EE552

Hardware Implementation of a Neural Network Trainer and Associated Neural Network

Final Report

Prepared For: Dr. Duncan Elliott

Group:

Darren Gonek	dgonek@ualberta.ca
Guillermo Barreiro	guille@ee.ualberta.ca
Andrew Ling	acling@ee.ualberta.ca
Shyam Chadha	shyam_courses@hotmail.com
Timmy Li	tli@nanogatesystems.com
Reid Orsten	rorsten@ieee.org

March 21, 2002

ABSTRACT

An artificial neural network trainer and an associated neural network designed to solve a two-dimensional spatial recognition problem was designed for this project. The project was selected to fulfill the goals of the course, to develop the necessary skills to design complex digital application specific integrated circuits (ASIC) and systems on a chip (SoC) using CAD synthesis tools. Working in a group, this significant digital system using a field programmable gate array (the ALTERA FLEX 10K70RC240-4) was designed and implemented.

Artificial Neural Networks attempt to recreate the behavior of a biological brain in logic to solve complex problems. They consist of nodes (neurons) in which inputs are multiplied by pre-trained weights and are summed together to produce overall outputs. These neurons must be trained before use, generally through a tedious procedure done in software. A hardware implementation using a Field Programmable Gate Array could take advantage of the high speeds achievable using hardware, and as a result would be a beneficial and economic investment for designs requiring artificial Neural Networks.

The system's implementation was divided into a number of individual modules that were combined to produce the finished product. These modules included: a software input and output device driver; an RS 232 serial connection to the FPGA and its interface; the Neural Network Trainer and associated Univariate Randomly Optimized Neural Network composed of a control unit, an error calculator, three networked neurons, a bus controller (used for the transportation of data), LPM RAM, and a pseudo-random number generator.

This report describes the implementation of these components including: design, uses, issues pertaining to their affect on the final project, and the actual resource requirements.

ACHIEVEMENTS

The foremost achievement of this group was the development of a strong team of individuals with diverse backgrounds and knowledge bases, who came together to form a successful project of this magnitude. This project developed and improved the teamwork and organizational skills of all members, and this was seen as a major achievement.

The SoC ASIC design of the Hardware Neural Network Trainer and Associated Neural Network consisted of the following major components:

- A Software Device Driver
- An RS232 Communications Interface (including FIFO Queues)
- A Hardware Device Control Unit
- A Network of individual Neurons interconnected using a data bus (the hardware implementation of the software algorithm)
- The Data Bus Control Datapath
- An Error Calculator
- A Pseudo-Random Number Generator.

The Device Driver was the software component of the system, used to control the system, which acted as a computer system peripheral. This driver was modified from a Microsoft open source MTTTY program to send the required formats of data to the hardware device using serial communication.

The RS232 Communications Interface utilized FIFO Queues to attempt to reduce bottlenecks in the design due to serial communications speed. The overall goal of the project was the Neural Network device, and thus a design decision to use a serial connection was made.

The Hardware Control Unit consisted of a complex state machine, which controlled all hardware aspects of the device.

The neurons were implemented in a number of ways, and the most suitable design was selected based on speed vs. size considerations. The Data BUS control datapath, network, error calculator and RAM were designed using CAD tools and implemented in Hardware on the FPGA.

The Pseudo-Random Number Generator was modified from a hardware design supplied in the labs.

The actual Neural Network algorithm implemented did not train faster then software as per the initial goal. It was concluded that this was due to the neuron design.

A final achievement of this group was a combined weight increase of 40 lbs. At the beginning of the project the net weight of the group was 1081 lbs, and at the time of completion the net weight was 1121 lbs. Needless to say, the group has learned something about the consumption of Tim Horton's donuts.

TABLE OF CONTENTS

INTRODUCTION

Artificial Neural Networks are an attempt to recreate the behavior of a biological brain in logic via software or, for the purposes of this project, hardware. These systems can be used to solve complex problems with a comparatively small amount of effort. As neural networks must initially be trained using a tedious procedure with a large amount of training input data before they can be used, a faster means of training a neural network would be valuable. In general, the training and use of neural networks is currently done using software. This hardware implementation of a neural network trainer and associated network using a Field Programmable Gate Array would take advantage of the high speeds achievable using hardware, and as a result would be a valuable investment for designs requiring artificial Neural Networks.

The specific application of this device is to train a neural network for area recognition/classification on an x-y plane. The neural network accepts two coordinates as input (x and y) and determines whether or not the point is in a specified area of the x-y plane (as determined from the training set). The x and y values are 8-bit integers in the range [0,255]. The network is trained by defining the area using a training data set consisting of a number of points classified as being in, and not in the area. The data will consist of three values: two coordinates (x and y) and the target. The target is a classification of 1 or –1, which is an indication of whether the coordinate is in the area (1) or not (-1).

IMPLEMENTATION AND DESCRIPTION OF OPERATION

The design is implemented both in Hardware and Software, and consists of three key components that interact with one-another as shown in FIGURE 1 (below). These components include: the Software Device Driver, which allows the user to specify the mode of operation (training mode or evaluation mode) and passes data to and from the device; and the actual Neural Network and Trainer consisting of a control unit and datapath implemented in hardware using the 10K70RC240-4 FPGA.

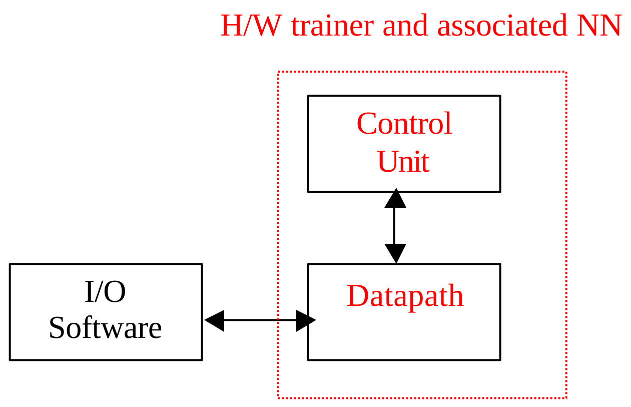

FIGURE 1. HIGH-LEVEL DIAGRAM OF DEVICE

INPUT/OUTPUT SOFTWARE DEVICE DRIVER

The user has the ability to specify the operational mode of the device (as "Training Mode" or "Evaluation Mode"). If the user selects training mode, they are required to provide a "training data" file containing a training data set in a specified format (see Appendix D page D3), a "weights" file name into which the final evaluated neuron weights will be stored, and a number of training iterations that the data set will be used to train the network. If the user selects evaluation mode, they are required to provide an "input data" file containing data in a specified format, a "weights" file containing the neuron weights, and an "output" file name into which the calculated output are stored.

When in training mode, this system trains the on-chip neural network by passing training data from the training data file provided by the user over an RS232 serial connection to the neural network (in training mode). The data is processed by the network, which is specifically designed in the FPGA for the requirements of the artificial neural network's application. The system trains the network using the provided data for a user-specified number of iterations. When the Neural Network has been trained, the final weights are passed back to the software driver via the serial connection, to be permanently stored in the weights file (as they are volatile on the RAM), and the training of the Neural Network is complete.

When in evaluation mode, the system initializes the weights of the on-chip neural network using the weights from the user provided by the user. The system then uses the on-chip neural network to evaluate the output of an input data set provided by the user, and passes the output back to the software driver.

The software driver controls the device via 4-byte signals. The first byte consists of a Command that tells the Hardware Neural Network trainer which functions to perform. The next three bytes may contain data, depending on the command. See APPENDIX B, page numbers B11 and B12 for a complete listing of signal possibilities.

CONTROL UNIT

The control unit controls the operation of the neural network trainer and associated network based on commands provided by the software driver. The control unit retrieves the control signal from the in-FIFO, and decodes the control packet to determine the mode of operation: train from serial data, train from RAM, return weights, load weights, and evaluate. It begins and ends each mode in an idle state ready to retrieve the next instruction. The control unit design consists of a state machine that traverses the states and sends and receives control signals to the Datapath.

Two possible training modes exist: the first to train from data provided by the RS232 and store it to on-chip RAM; and the second to train using values stored in RAM allowing subsequent epochs to be completed much faster then possible with the transmission delay of the first epoch. During a training run, a random weight is generated for one weight in the neural net, and the output is evaluated. If this new weight improves the performance of the system it replaces the old weight; otherwise the original weight is kept. The training algorithm used performs this calculation for each input set 6 times per neuron, resulting in a final best weight for the neurons. When the trainer has completed the specified number of training runs it enters the idle state awaiting the next instruction. When the control unit decodes the return weights command, it loops through all of the neurons and weights and returns them via the serial connection. This functionality is useful because the weights are volatile (not stored on power down) and it removes the requirement to retrain the network on every use.

When the load weights command is received, the control unit loads the specified weight to the indicated location it is to be loaded to. This functionality is useful to reload the weights upon startup to remove the requirement to retrain the network on every use.

When the control unit receives the evaluate command, it runs the inputs through the network to produce an output (classification), and returns this value to the software.

HARDWARE DATAPATH

The control unit controls the functionality of the device using control signals. The datapath consists of a number of components including:

- The RS-232 Asynchronous Serial Connection
- FIFO Queues
- The Memory Interface
- The Data Bus Controller
- The Neural Network
- The Neurons
- The Error Calculator
- The Random Number Generator

RS-232 ASYNCHRONOUS SERIAL CONNECTION

The RS232 asynchronous communications interface transmits packets of information to and from the device via the PC's RS232 port and the FPGA. The packets consist of an 8-bit payload contained within a least-significant start bit ('0') and a most significant stop bit ('1'). This framing is done by the software and is used as a transmission protocol by the FPGA to identify the payload. Thus, each control signal is four packets in length.

After being passed through the serial connection, the packet is passed through the MAX232 converter, which adjusts the voltage level from the TTL/CMOS values (+5.0, -5.0) found on the FPGA to that of the RS232 serial port (+25.0, -25.0). The FPGA then accepts the data via a General Purpose Input Output pin, verifies its integrity, and stores the values into four data storage registers.

The transmitter and receiver must agree on a common word length, and must also use the same baud rate to ensure transmission synchronization. The transmission protocol used is the same as the RS232 application note [6].

The software and hardware components are custom designed to communicate correctly with one-another. The software driver is "aware" of the structure of the Neural Network, and thus is able to ensure that the correct data is transmitted and received, and can control the hardware correctly. Also, both the software and hardware "know" what to expect of one another when a command is sent by the software to the hardware.

FIFO QUEUES

The FIFO queues acts as receive and transmit buffers for the RS232 link between the hardware and the software. A receive and transmit FIFO queue connect the control Unit to the external RS232 port. The receive FIFO currently has a size of four 32-bit words and the transmit FIFO has six 16-bit words.

The FIFO's are in control of handshaking signals between the software and the control unit. When the receive FIFO is full, the software is informed to stop sending data via a CTS signal. The control unit checks an rx_empty signal to see if the receive FIFO is empty; and if not, takes data from it as required. When the transmit FIFO is not full, the control unit is free to add data to the transmit FIFO. The transmit FIFO then signals the RS232 port to send the data at the front of the queue.

THE MEMORY INTERFACE

This component handles the storage of the training data into RAM during the first epoch from the FIFO queue. The memory interface also recalls the training data sets for the control unit during the subsequent training epochs.

THE DATA BUS CONTROLLER

This component is designed to get and set the weights and to activate the neural network during operational mode. The data bus controller controls the transfer of data through the neural network, and supplies the network with new random weights as well as the inputs. The datapath entity is instantiated for each neuron. Subsequently, each neuron will have a unique identifier corresponding to the number in which it was instantiated. A comparator uses this identifier to ensure that the value being passed in on the address bus lines matches the identifier of the neuron. If a match is detected, then it sets the equal signal to high, and the logic gates for the neuron can be used.

Along with the address bus and the data bus, there exists a command bus and select bus. The command bus provides the neuron with one of four commands utilizing 2 bits, which also act as inputs to the combinational logic:

00 - read the weights of the neurons, i.e. send to the datapath
01 - write weights of the neurons, i.e. take from datapath and store
10 - idle
11 – begin forward calculation

The select bus tells the neuron which weight it is dealing with. The combinational logic (AND gates) determines from the bus inputs which weight is being used and what operation to perform. Each weight is stored in a storage unit, controlled by an enable (write the weight) and q_enable (read the weight).

The bi-directional 8-bit data bus exists to provide the weight storage units with their values during storage and to take the values from the storage units when reading. The address bus is used to inform the neural network which neuron it should be performing a read or write on. The schematic for the datapath is included in Appendix B.

THE NEURAL NETWORK

This component consists of the neurons and their connections into a network and to the data bus.

THE NEURONS

The algorithm chosen for implementation was the *Univariate randomly optimized Neural Network (uronn)* (Reference 1, pages 211-212). This Algorithm searches for the weights that best fit the neural network by randomly changing them. This algorithm is a very advantageous first approach in the implementation of a neural network trainer because few calculations have to be performed in each epoch and 8-bit signed integers can be used as values for weights and inputs instead of 32-bit floating points values. This algorithm is also very handy for comparing training speeds between software and hardware. The pseudocode for this algorithm is provided in Appendix B.

Each neuron has a "hold_weights" component that does all the interaction with the datapath. This component is composed of two blocks: a storage unit for the weights and the actual Hebbian Neuron. The first block stores the actual weight and eventually stores the new weight values to be tested during training. The second block calculates the output of the whole component. The block diagrams for the datapath of the neuron and its interaction, schematic diagrams for both components, and the state diagram for the neuron are provided in Appendix B.

The datapath contains three neurons, two for the hidden layer and one for the output layer, and are connected in a traditional feed-forward architecture. For this implementation input neurons were not considered, as their purpose is to merely distribute the inputs among the neurons at the hidden layer. Thus, in the design the inputs come directly from the data bus into the neurons at the hidden layer.

Two different prototypes for this datapath were designed: one with a bi-directional data bus and one with an independent input data bus and an output data bus. Although the first prototype is smaller and uses the FPGA's resources better, its implementation is not as modular as the second, and a future scaling of the neural network if this prototype was used would require more work than using the second prototype.

THE ERROR CALCULATOR

This component compares the output evaluated by the neural network from the inputs provided with the current weights, and compares it to the expected target output. If the evaluated output is closer to the target value, the current random weight being tested replaces the previous weight value.

THE RANDOM NUMBER GENERATOR

The random number generator generates the random weights used by the neural network.

FPGA USER I/O SIGNALS

Pin Description	Pin Type	Pin Mapping	Pin Description
Clock	Input	Pin = 91	System Clock at 25.175 MHz
Reset	Input	Pin = 28 (Push Button 1)	Used to reset the system, Active Low
rx_full	Output	Pin = 55 (Hole Number 23)	Asserts to High when the rx FIFO is full, tells the software to stop sending data
CmpSerialIn	Input	Pin = 61 (Hole Number 25)	Receive
CmpSerialOut	Output	Pin = 63 (Hole Number 27)	Used to send packets of 8bit data from the FPGA to the Computer

TABLE 1: FPGA User I/O Signals

In deciding on what I/O pins were needed, three areas were given focus: data entry, data exit, and user interfaces. Currently, there is only one line needed for data entry and one line for data exit. This creates two need I/O pins which connect directly to the PC via the interface circuit described in the sections above. Another output line is needed for flow control of data entry telling the software to stop sending data if the rx FIFO buffer is full.

Of course, there needs to be a universal clock synchronizing events in the design and a universal reset pin. That makes a total of 5 I/O pins needed for the Neural Net.

EXPERIMENTS/CHARACTERIZATION

RS-232

When testing the RS232 port, various speeds were chosen, from a 9600-baud rate to a max transmission speed of 115200 bps. The only reason for the upper limit is due to the PC side being unable to transmit data any faster through its COM ports. Integration of the RS232 port with the Neural net was done with a 9600-baud rate to reduce the probability of errors due to corrupted data. However, during the demonstration a max speed of 115200 bps will be used if possible.

Neuron

Because the Neuron has a multiplier component inside it, the measurements of size and speed change considerably depending on the implementation used for it. Four different prototypes were designed, taking into consideration size constraints.

Prototype 1: Booth Algorithm multiplier

A classical Booth algorithm was implemented as the multiplier unit. Some states were added to the state machine, but in general there were no huge modifications.

- Advantages: Small, multiplies signed and unsigned numbers without changing any part of the component, easy to control.
- Disadvantages: Slow performance, calculates multiplication in different time for different input numbers, causing the system to be idle if one neuron has not finished its task while the others finished already, the architecture of the datapath has to be slightly changed.

Prototype 2: Carry-Save-Adder

A parallel carry-add-adder was implemented.

- Advantages: Regular structure, fast, could be optimized for better performance through pipelining.
- Disadvantages: The architecture implemented only works with unsigned numbers, critical path could delay considerably the performance if the number of bits of the inputs and weights is increased, and modifications could be somehow complicated.

Prototype 3: Multiplier using lpm_mult megafunction :

The lpm_mult megafunction was included in the datapath replacing the multiplier component. Its pipelining parameter was set to 1, in order to reduce space consumption.

- Advantage: A function in a library always works, easy implementation of pipelining and parameterization (change width of variables, etc.), easy reference and no problem with debugging.
- Disadvantage: Synchronization of the clock is difficult, big in size when synthesized, has problems with delay times when connecting the control unit.

<u>Prototype 4: Neuron architecture implemented using behavioral architecture</u>
A behavioral description of a Hebbian Neuron was implemented in this design

- Advantage: fast, easy to implement, easy to add changes, easy to modify.
- Disadvantages: extremely big, uncertainty on structure implemented by the synthesizer, its performance is lost when connected to another components.

The following tables summarize the measurements for size and speed, the throughput of each model and a measurement of frequency/area used to see which prototype has better usage of resources. These experiments were performed on both the Flex10K20 and Flex10K70.

	10K20RC240-4		
Prototype Number	Clock Period (ns)	Frequency (MHz)	Percentage of LC used (%)*
1	62.1	16.10	19
2	66.6	15.01	21
3	57.0	17.54	26
4	8	125	32

TABLE 3: Experimental Measurement of Size and Speed on Flex10k20

	10K70RC240-4		
Prototype Number	Clock Period (ns)	Frequency (MHz)	Percentage of LC used (%)*
1	69.2	14.45	6
2	93.7	10.67	6
3	77.1	12.97	8
4	8	125	10

TABLE 4: Measurement of Size and Speed on Flex10k70

	10K20RC240-4	
Prototype Number	Throughput (us)	Resource Usage (MHZ/LC)
1	4.95	0.070
2	0.718	0.061
3	0.870	0.057
4	0.1	0.336

TABLE 5: Measurement of Throughput and Resource Usage on Flex10k20

	10K70RC240-4	
Prototype Number	Throughput (us)	Resource Usage (MHZ/LC)
1	4.95	0.061
2	0.718	0.044
3	0.870	0.042
4	0.1	0.331

TABLE 6: Measurement of Size and Speed on Flex10k70

As seen on the tables, prototype 4 has the best throughput, latency and resource usage. Unfortunately the control over the components inside these designs is not complete. Still for this first integration of all the components of the project the prototype 3 was chosen, for it allows a clear visualization of the data flow and an easier manipulation of the inputs and weights.

To see the effectiveness of the algorithm two implementations in software tools were programmed, one in Matlab and one in C.

In both cases the algorithm showed to be very slow and to have a high error percentage with a non-normalized training database. However this experiment was performed for statistical uses only, and to have a comparison point for the neural training that is being implemented. This is an ideal study case to compare speed on software-developed systems and hardware-developed systems. The following table summarizes the results obtained using the software-implemented algorithms trying to train a simple x-y position exercise.

	Matlab		C	
Number of Iterations	Time	Final Error	Time	Final Error (%)
1000	21 seconds	0.2066	1.3 seconds	0.2103
2000	46 seconds	0.2080	2 seconds	0.2106
5000	2:15 minutes	0.2075	5 seconds	0.2088
10000	5:10 minutes	0.2072	10 seconds	0.2094

TABLE 7: Software Developed Algorithms for Experimentation

The full problem and all of its data as described in the introduction, are included in Appendix D.

Tristate buses

One of the first implementations of the Hold_weights component was built using little components that would allow each individual weight to communicate with the bi-directional bus. This component (weights_io) used tristate buffers for communication purposes with the outside world. The schematic diagram is shown in Appendix B.

Unfortunately when trying to scale up the design by building another components based on this component, it was found to be non-compatible because the inout port was being driven by more than one source. Thus this design had to be dropped and a new implementation of the Hold_weights component had to be done. The code and simulation for the Weights_io component are provided.

A tristate buffer is used among the components used in the Hold_weights component that is being used right now. Because this design brings little scalability the tristate buffer will be replaced by a multiplexer.

DESCRIPTION OF TEST CASES/SIMULATION

RS-232 Port and FIFO buffers

The difficulty in testing the RS232 port relates to its asynchronous behavior. It is common knowledge that asynchronous programs and chips are much harder to debug since they are driven by events. Testing was accomplished two ways, running the hardware in real-time and using the MaxPlus II waveform editor.

The real-time testing involved connecting the FPGA to a PC and getting the two devices to send and receive serial data. The PC would send data from its keyboard. This data would be received by the FPGA and stored into its rx FIFO buffer. Data sent from the FPGA was accomplished by looping back the rx FIFO to the tx FIFO. On a push button command, data from the rx FIFO was entered to the tx FIFO and sent out to the RS232 port. The PC Software receives the data and converts it to an ASCII representation so it is readable by any text viewer.

Simulations involved manually entering input data in waveforms. For the RS232 port, two cases were tested for receiving data and four cases were tested for sending data.

Receiving: Accepting Valid Data, Accepting Invalid Data
Sending: Overflow of transmission stream, Disabling the transmit enable signal,
Asserting the transmit reset, Sending consecutive data.

As for the tx and rx FIFO's, data was simply entered and read to ensure the FIFO behaved properly.

Datapath

The testing of the Neural Network was very trivial. Essentially, a combination of the CMD, SEL, and ADDR inputs were tested to ensure that their functionality was correct and that the correct things occurred when the different situations arose. For example, if CMD was set to 00, it was ensured that a data value was "read" by asserting the read output to be the data. The two prototypes were tested and the proper behavior of each one is included in Appendix E.
A test bench for the Neural Network Datapath with bi-directional databus is included in Appendix C.

Neuron

All the neuron simulation waveforms are marked according to the prototype used and the expected output (either positive or negative one). Each simulation shows the Done signal outputted by the component and the final answer on the accumulator register.

A waveform of the Random Number Generator is also attached. It shows how the numbers generated are random, although if the generator starts at the reset state the numbers generated will always be the same. This problem will be fixed with the control unit.

Waveforms of the Hold_weights component are attached to see how it behaves properly. It shows how the component reads and writes properly to the assign weight. However this component will have to go through some modifications to male it reusable.

A waveform of the weights_io experimental component is also attached. It shows how the component reads and writes properly from one register. All the waveforms for all 3 components are included in Appendix E.

CONCLUDING REMARKS

The main advantage of algorithm implementation in hardware over software is speed of execution. The propose of a hardware implementation of a Neural Network Trainer and associated Neural Network, is that the device could take advantage of this characteristic, making it more attractive then currently available software implementations. Although after comparing the implementation of this artificial NN in hardware to implementations in software (C and Matlab), it was expected that the design would execute faster, the hardware design was in-fact slower. This project did achieve full functionality, and with more time, the design could have been modified to increase its overall speed possibly by pipelining the multipliers used, and using a different form of communication between the software and hardware (parallel communication rather then serial communication).

Developing the interface between the PC and our Neural Net gave much insight on the benefits and difficulties that come with serial communication. Like with all hardware designs, the tradeoff between speed and hardware were debated, as was the simplicity of design and error detection.

Our Serial Port has very limited error checking, simply ignoring packets that don't follow a standard RS232 format. As for speed, since there is only one tx data line and one rx data line, the transmission of data was bottlenecked; however, the benefit of the simplicity in the hardware came with that latency. RS232 protocol is probably not the best solution for communication between the device and the PC, but it was chosen due to its simplicity. Its simplicity made it easy to debug and design, thus focusing our efforts on the Neural Net itself, the core of the project.

The main advantages of RS232 communication is its adaptability to many devices, and its asynchronous nature. However, with other protocols such as USB becoming the norm, we feel that our Neural Net would probably abandon RS232 in future upgrades to the product.

Creating a neural network trainer in hardware is not as beneficial as originally anticipated. The complexity of the traning algorithms makes even an extremely simplistic one a significant undertaking requiring lots of hardware. For the device to be worthwhile, it would have to be full custom to achieve performance significantly above that of a modern workstation. Even as a full custom chip, communication with software could remain a bottleneck.

By analyzing the current implementations of the compiled components and simulations, it has been determined that the design would indeed have been too large to fit on the 10K20 FPGA, and it currently requires 60% of the 10K70.

Testbenches were used to test the individual components, in addition to the extensive testing that was done on each of the components individually, to identify and remove a number of bugs. The results of the simulations show that although the speed requirements were not met, the functionality requirements certainly were.

As expected the issues of interfacing between hardware and software, the neurons into the network, and the actual network to a trainer implemented in hardware provided many challenges. We have encountered some of these challenges and working as a team, such as clock skew, and interfacing design issues that were created due to miscommunications. These challenges were overcome, allowing us to realize the importance of communication in a group work setting. Through teamwork, a fully functioning prototype was completed by the required deadline.

REFERENCES

1) Looney, Carl: “Pattern Recognition Using Neural Networks: Theory and Algorithms for Engineers and Scientists”, Oxford University press, New York, 1997. Chapters 3, 4 and 6.

2) Perez-Uribe, Sanchez: “FPGA implementation of a Network of Neuronlike Adaptive elements”, International conference on Artificial Neural Networks ICANN, 1997, LNCS 1327, pp 1247-1252. http://lslwww.epfl.ch/~aperez/ps/PerezSanchez_icann97.pdf

3) Beuchat, Haenni, Sanchez: “Hardware Reconfigurable Neural Networks” http://ipdps.eece.unm.edu/1998/raw/haenni.pdf

4) Chapman, Sutankayo: “Implementing Artificial Neural Network Designs: Final Report” http://www.ee.ualberta.ca/~elliott/ee552/projects/1998_w/NeuralNets/inndfr.pdf

5) Ossoinig, Reisinger, Steger, Weiss: “Design and FPGA-implementation of a Neural Network” http://www.icspat.com/papers/493mfi.pdf

6) Bensler, Chan: “RS232 Serial Port” http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999_w/RS232/

7) http://www.cs.unbc.edu/hdp/VHDL/samples/samples.html