An FPGA-Based General Purpose Neural Network Chip With On-Chip Learning

ARTICLE · JANUARY 2005

READS
21

2 AUTHORS, INCLUDING:

Yaser Khalifa
ABB
34 PUBLICATIONS **106** CITATIONS

SEE PROFILE

An FPGA-Based General Purpose Neural Network Chip With On-Chip Learning

Yaser M.A. Khalifa
State University of New York at New Paltz
75 South Manheim Blvd.
(845) 257-3764
yaserma@engr.newpaltz.edu

Yu Jen Fan
State University of New York at New Paltz
75 South Manheim Blvd.
(845) 257-3746
fan00@newpaltz.edu

ABSTRACT
In this paper, a description of a general purpose neural network chip with on-chip learning is given. The design is implemented using Xilinx Vertex II XCV 1000 Field Programmable Gate Array (FPGA). An XOR gate simulation was used as a testing application. Results and comparison of both software and hardware implementations are listed. A second testing application in noise cancellation and voice recognition is currently under development.

Keywords
Neural Networks, Field Programmable Gate Array, Hardware Description Language.

1. INTRODUCTION
Neural networks can be implemented in software and hardware. Since implementations on a conventional host computer are easier and cheaper comparing to hardware, learning algorithms are usually implemented in software for low-level applications. The performance of a conventional processor, e.g. the Intel Pentium series, continues to improve dramatically but they are still far away from the required performance. Even the fastest sequential processor can not provide real-time response for neural networks with large number of neurons and synapses. Therefore speed becomes the primary reason of using hardware implementations for neural network. In this paper, we provided the result from both software and hardware implementation.

2. OVERVIEW OF NEURAL HARDWARE
Implementation of Neural Networks can be more practical and attractive when high-speed and cost-effective neural hardware is available. There are some attributes to categorize these hardware devices, depending on their purposes, architectures, and learning types. The graph of taxonomy is shown in Figure 1.

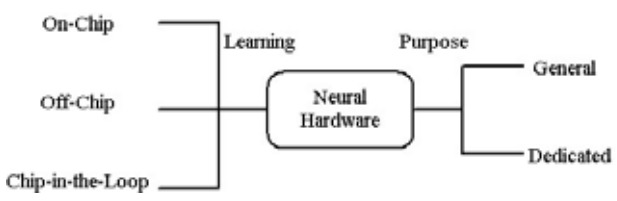

Figure 1. Taxonomy of neural hardware

General purpose designed neurochips can be developed by either single processor or multiple processor systems, and require a huge number of computations and communications resources to perform a neural function. Most general purpose neurochips are digital so that the processors would be able to execute larger neural networks in a parallel environment. The advantages of using general designed neuro-computers are ease to implement, wide availability, and lower cost.

Dedicated purpose neural hardware design usually has a higher performance because the direct mapping from hardware structure to silicon is available. However, a direct mapping in the interconnection also implies massive communication cost. Therefore the dedicated design usually depends on the application requirements and usually a custom-design. The drawback of this design is the lack of flexibility.

Neural Network hardware also can be classified by its learning type: on-chip, off-chip, and chip-in-the-loop learning. On-chip learning provides the highest speed and precision. The training algorithm and the weight updates are completely done on chip. Off-chip learning is performed by a host computer. The weight updates from the training process are quantized and then downloaded on the chip. Obviously, off-chip learning can reduce the silicon space, but it is much slower than on-chip learning. Chip-in-the-loop learning uses the hardware for only forward propagation. The calculations of the new weights are done by a host computer, which means the updated weights will be downloaded to the chip after a training cycle. In this paper we describe an on-chip learning general purpose neural network chip using Xilinx FPGA, which means we downloaded the program into FPGA and performs without any other devices.

3. LEARNING AND TRAINING
This section introduces the concept of training a network to perform a given task. In order for a TLU to perform a given classification, it must have the desired decision surface. Since this is done by the weight vector and threshold, it is necessary to adjust these to bring about the required functionality. In general terms, adjusting the weights and thresholds in a network is usually done via an iterative process of repeated presentation of examples of the required task. At each presentation, small changes are made to weights and thresholds to make them more close to their desired values. This process is known as training the net. From the network's viewpoint, it undergoes a process of learning, and the prescription for how to change the weights at each step is the learning rule (or the learning algorithm).

There are two types of neural network (classified by structure), one is feed-forward network and the other is feed-back network. Both of these two networks can be expressed as a supervised learning.

3.1 DELTA Rule

The unit perceptron is constructed by single layer network, collect the input data to form an activation function, and classified the output using threshold. To make the perceptron learning, we use the "DELTA rule" to train nodes. The formula is given:

$$W_i(t+1) = W_i(t) + \Delta W_i \quad (1)$$

$$\Delta W_i = \alpha * \delta * X_i \quad (2)$$

$$\delta = t_i - a_i \quad (3)$$

Here, W_i is the synaptic weight from input neuron to output neuron, t_i is the target output, a_i is the actual output, and α is a positive factor of proportionality called learning rate. The rule tells us that we get the new value of a weight by adding a certain, weight-specific "delta". In other words, the change of a weight feeding into an output unit is the bigger the more the actual output deviates from the target output. If the difference between actual and target output is zero, no changes need to be made to this weight. This rule works well for single-layer networks (i.e. perceptron), but can not be applied to general feed-forward nets, since there is no information concerning "targets" for hidden units, so the delta rule does not tell us how to change the weights feeding into these units. On the other hands, we can not limit ourselves to networks without hidden units, since it has been proved that they are necessary for some tasks, and that is why we need feed-back network which represented by back-propagation algorithm to change the weights in hidden layer.

3.2 Back-Propagation Algorithm

Back-propagation algorithm is the most important and popular learning algorithm [5]. The algorithm works with a generalization of the delta algorithm to multi-layer feed-forward networks, that means in particular to networks with hidden units. The algorithm is called back-propagation, and it is based in the mathematical method of gradient descent. Rumelhart begins his original paper by presenting a derivation of the delta rule that shows the delta rule also implements gradient descent. The idea is to define an error function:

$$E = \frac{1}{2}\sum_i (t_i - a_i)^2 \quad (4)$$

Where i varies over all output units and all input/output patterns. This function is assumed to be a function of the weights, and the ultimate goal is to minimize this function (i.e. minimize the error) by adjusting the weights. There is no formula for the error function, the only known pieces of information are the current point in weight space and the current output, or its deviation from the desired output, respectively. The optimal vector is then found by minimizing this function by gradient descent as shown schematically in figure 2.

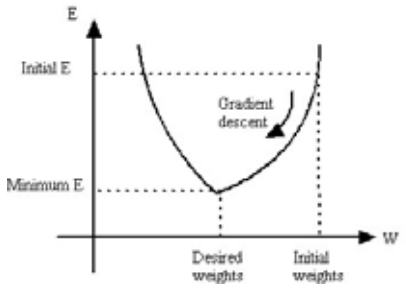

Figure 2. Gradient Descent for a Network

In this case there is an extra term in the delta rule that is the derivative of the sigmoid $d\sigma(a)/da$. It is convenient occasionally to denote derivatives by a dash or prime symbol so putting $d\sigma(a)/da = \sigma'(a)$, we obtain:

$$\Delta W_i = \alpha\sigma'(a) * (t_i - a_i) * X_i \quad (5)$$

Where $\sigma(a)$ is the activation function and $\sigma'(a)$ is the slope of the sigmoid function. They are showed in figure 3.

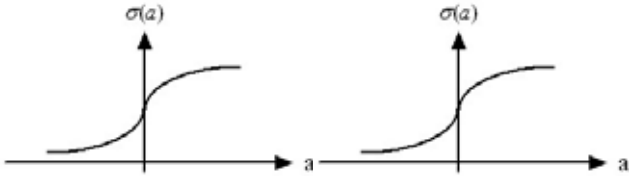

Figure 3. The sigmoid function and its slope

It is now possible to write the hidden node learning rule for the k *th* hidden unit as:

$$\Delta W_{ki} = \alpha\sigma'(a_k)\delta^k X^p{}_{ki} \text{ where } \delta^k = \sum_{j\in I_k} \delta^j W_{jk} \quad (6)$$

Therefore, for any node *k* (hidden or output) we may write:

$$\Delta W_{ki} = \alpha * \delta^k * X^p{}_{ki} \quad (7)$$

For the output nodes:

$$\delta_k = \sigma'(a_k)(t^p{}_k - a^p{}_k) \quad (8)$$

For the hidden nodes:

$$\delta_k = \sigma'(a_k)\sum_{j\in I_k} \delta^j W_{jk} \quad (9)$$

In line with convention, this is the usage that will be adopted subsequently. It remains to develop a training algorithm around the rules we have developed. In [2], it will be clearer to expand the main step of back-propagation algorithm in following steps

1. Present the pattern at the input layer
2. Let the hidden units evaluate their output using the pattern.

3. Let the output units evaluate their output using the result in step 2 from the hidden units.
4. Apply the target pattern to the output layer.
5. Calculate the δ's on the output nodes.
6. Train each output node using gradient descent (7)
7. For each hidden node, calculate its δ according to (9)
8. For each hidden node, calculate its δ according to (6)

The steps 1-3 are collectively known as the forward pass since information is following forward through the network in the natural sense of the nodes' input-output relation. Steps 4-8 are collectively known as the backward pass. Step 7 involves propagating the δs back from the output nodes to the hidden units-hence the name back-propagation. The networks that get trained like this are sometimes known as multilayer perceptron or MLP. A simulation from Matlab is shown in Figure 4.

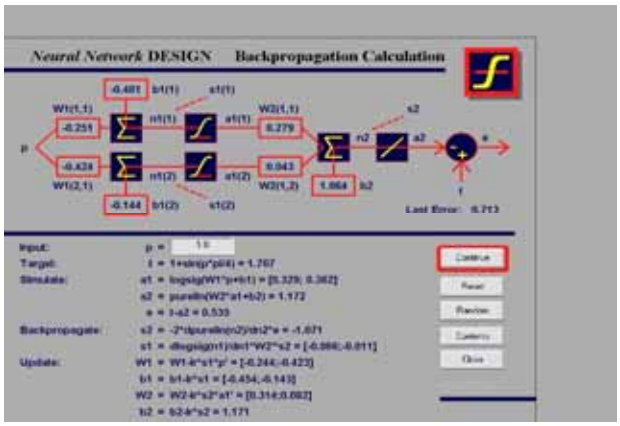

Figure 4. Back-Propagation algorithm

4. RESULT FROM SOFTWARE

C language is one of the most popular environments for designers. In our project, we used Visual C++ to simulate the XOR gate using back-propagation algorithm. The training process basically checks the result for every possible model ("00", "01", "10", and "11"). The updated weights must be satisfied by all of these sets. If a set of weights are qualified for part of sets such as "00" and "01", but fails to qualify the "10", the process requires resetting the counter and running through from "00" again until all the weights are satisfied by these sets. The procedure takes a long time to simulate while the error is limited to 5%.

5. Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is a completely reconfigurable logic chip. This logic chip consists of millions of configurable logic block (CLB) which can be linked together to form complex digital logic implementations. The individual units are interconnected by a matrix of wires and programmable switches. A user's design is implemented specifying the simple logic function for each logic block and selectively closing the switches in the interconnect matrix. The array of blocks and interconnects figure a fabric of basic building blocks for logic design circuits. A typical Computer Aided Design (CAD) for an FPGA would include software for certain tasks like the behavioral design, functional simulation, verification, placing and routing. We used Xilinx ISE for simulation environment, and VHDL (VHSIC Hardware Description Language) also provides a consistent and portable design instrument pointing FPGAs.

5.1 FPGA Logic Block Architecture

The basic idea underlying the architecture of an FPGA is very simple. In general combinational and sequential circuits can be implemented directly in silicon. A generic FPGA consists of numerous programmable *logic blocks* which have the capability to implement some digital functions. Logic blocks comprise programmable routing switches which connect the input and output pins of each logic block. When a circuit is implemented in an FPGA, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block. An illustration of a typical FPGA architecture is showed in Figure 5 [1]. The I/O pads are evenly distributed around the perimeter of the FPGA.

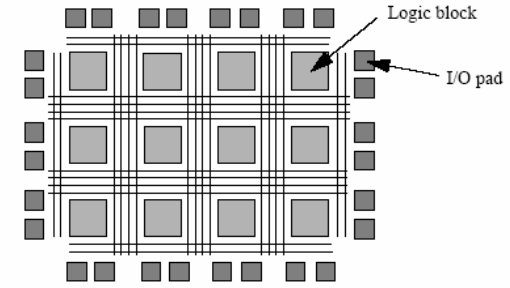

Figure 5. Structure diagram of FPGA

The advantages of FPGA over a microprocessor chip for neural computing can be listed as [3]:

1. Developing hardware systems using design tools for FPGAs is as easy as developing a software system
2. FPGAs can be re-programmed on the fly
3. The new FPGAs on the market support hardware that requires more than 1 million gates (a small scale processor can be implemented using such an FPGA)
4. A custom circuit built on an FPGA operates faster than a microprocessor chip

These advantages make FPGAs now viable alternatives to other technology implementations for high-speed neural applications. The structure of a FPGA can be described as an array of blocks connected together via programmable interconnections. The amount of logic that each logic block can implement depends on which family of FPGAs is being used. The most important advantage of FPGAs is the flexibility that they provide. An engineer can change and refine his design by exploiting the device's re-programmability [4].

5.2 FPGA Design Flow

The design flow broadly refers to the sequence of activities encompassing various design tools that begin with some abstract specification of a design and ends with a configured FPGA. The design flow described in this section is that of the Xilinx ISE

environment and is illustrated in Figure 6. However most of the activities will have a counterpart in any vendors' design flow [6].

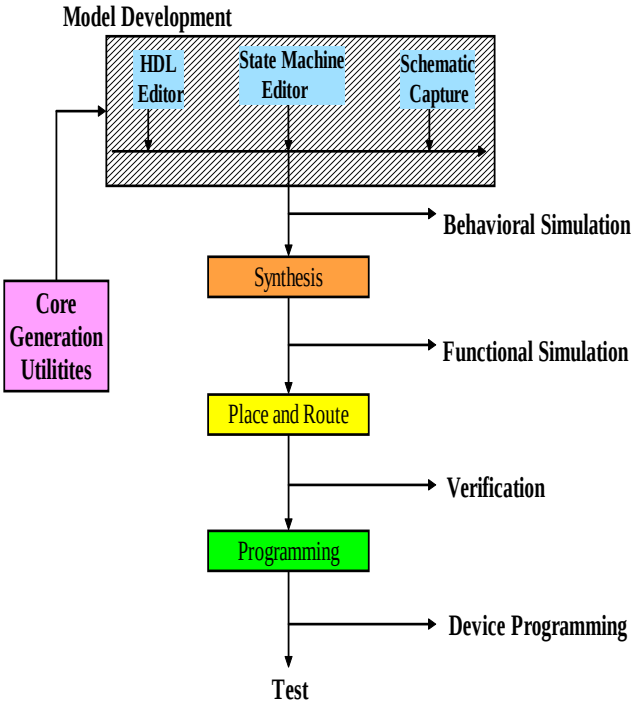

Figure 6. FPGA Design Flow

The core generation utilities are divided in three categories: HDL editor, state machine editor, and schematic capture. Numbers of EDA tools provide these functionalities, such as Xilinx ISE, ModelSim, Synplify, and MaxplusII. In VHDL environment, behavioral design and synthesis are necessary and help us to check the correctness and logicality of the program. After functional simulation, we can start to do verification and placing and routing on the FPGA. Once the verification is completed, the program can be downloaded onto the FPGA.

6. RESULT

The complete circuit diagram shown in Figure 7 corresponds to our neuron-model. The implementation platform was Xilinx AFX BG560-100 Prototype Board comes with Xilinx Virtex XCV1000 FPGA device. The board also includes XC1800 configuration PROMs in PC44 packages and can be reprogram via JTAG.

Before synthesizing the circuitry, the simulation is carried out by the Xilinx ModelSim XE. Finally, the design was synthesized using Xilinx Project Navigator.

7. CONCLUSION

In this paper the neuron model is designed, simulated, and synthesized in Xilinx Vertex II, XCV1000 FPGA. This is achieved using Xilinx ISE software packages and AFX BG560-100 Prototype Board that instrumented with download and upload utilities.

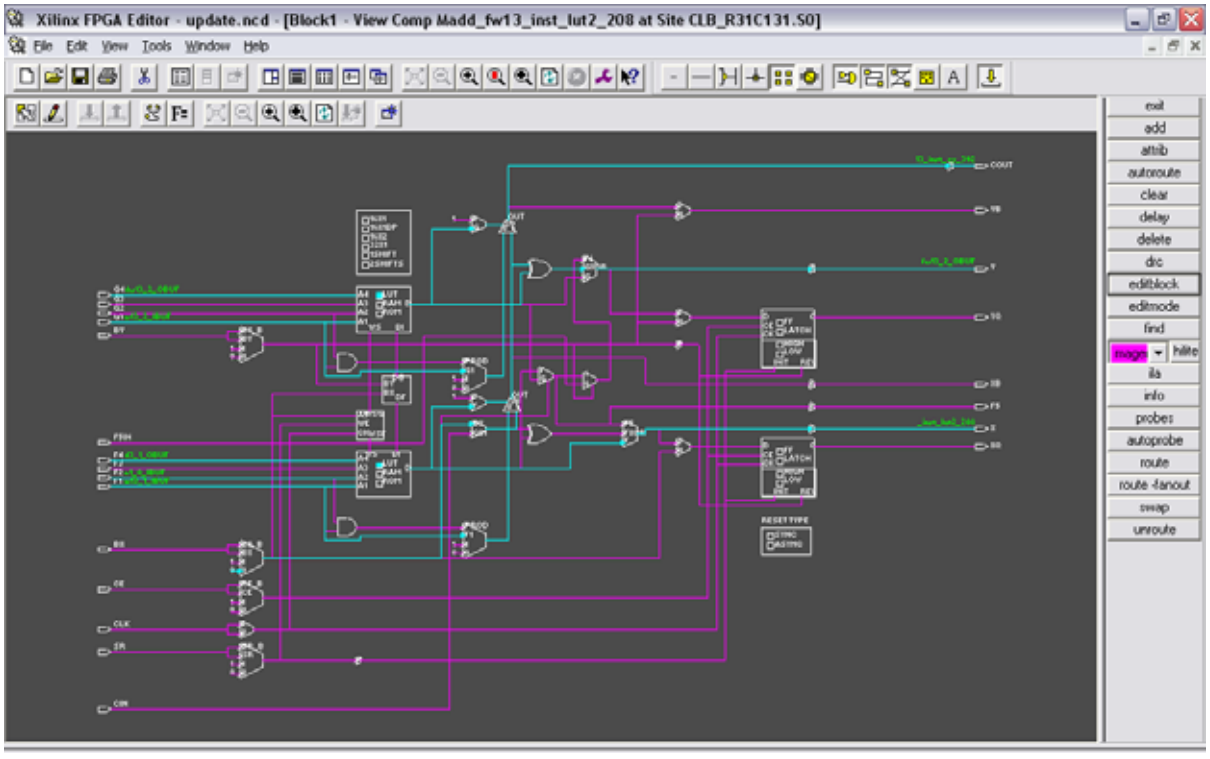

Figure 7. Circuit diagram of the design

Since the primary performance for this design is on-chip learning for general purpose, training is done completely on the chip without any other devices. Currently, we are working on using this chip in Adaptive Noise Cancellation for speech recognition applications. We used the neuron-model as a filter, and adjust the weights to minimize the error to get the optimized output. This work is under construction and hopefully more result can be brought to the conference.

8. REFERENCES

[1] Ahmed, E. and Rose, J. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *ACM Symp. FPGAs*, pp. 3-12, 2000.

[2] Gurney, K. An Introduction to Neural Networks, *Press, 1997*.

[3] McKenna, M. and Wilamowski, B. 2001. Implementing a Fuzzy System on a Field Programmable Gate Array. *International Joint Conference on Neural Networks* (IJCNN'01), pp. 189-194, 2001.

[4] Özmen, A., Tekçe, F., and Vardar, K. Hardware Implementation of a Neural-Model. *Proceeding of International Conference on Signal Processing*, Vol. 1, No. 2, 2003.

[5] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning Internal Representations by Error Propagation. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1, Foundations, eds. D.E, 1986.

[6] Yalamanchili, S. Introductory VHDL: From Simulation To Synthesis. *Prentice Hall*, 2001.