Hindawi Publishing Corporation
Advances in Artificial Neural Systems
Volume 2011, Article ID 189368, 9 pages
doi:10.1155/2011/189368

Research Article

An Optimal Implementation on FPGA of a Hopfield Neural Network

W. Mansour,[1] R. Ayoubi,[2] H. Ziade,[3] R. Velazco,[1] and W. EL Falou[3, 4]

[1] *TIMA Laboratory, 46 avenue Félix Viallet, 38031 Grenoble, France*
[2] *Department of Computer Engineering, University of Balamand, Tripoli, Lebanon*
[3] *Electrical and Electronics Department, Faculty of Engineering I, Lebanese University, El Arz Street, El Kobbe, Tripoli, Lebanon*
[4] *Lebanese French University of Technology and Applied Sciences, Tripoli, Lebanon*

Correspondence should be addressed to W. EL Falou, wafalou99@hotmail.com

Received 3 March 2011; Revised 30 April 2011; Accepted 4 June 2011

Academic Editor: Paolo Del Giudice

The associative Hopfield memory is a form of recurrent Artificial Neural Network (ANN) that can be used in applications such as pattern recognition, noise removal, information retrieval, and combinatorial optimization problems. This paper presents the implementation of the Hopfield Neural Network (HNN) parallel architecture on a SRAM-based FPGA. The main advantage of the proposed implementation is its high performance and cost effectiveness: it requires $O(1)$ multiplications and $O(\log N)$ additions, whereas most others require $O(N)$ multiplications and $O(N)$ additions.

1. Introduction

Artificial Neural Networks (ANN's) have become a subject of very dynamic and extensive research [1–4]. One important factor is the progress in VLSI technology, which makes easier the implementation and testing of ANNs in ways not available in the past. Indeed, the improvement of VLSI technology makes feasible the implementation of massively parallel systems with thousands of processors. Another important factor is the resurging of ANNs as a powerful paradigm for complex classification and pattern recognition applications.

There are many publications in the literature concerning the implementation of Hopfield Neural Network (HNN) in FPGAs (Field Programmable Gates Arrays). In [2] an implementation of HNN on Xilinx VirtexE is used for block truncation coding for image/video compression. Reference [4] describes an implementation of HNN on FPGA (Virtex-4LX160) for the identification of symmetrically structured DNA motifs in Alpha Data, which has better performance than the same algorithms implemented in C++ on a IBM X260 Server. In another work [1] is studied the implementation of an associative memory neural network (AMNN) using reconfigurable hardware devices such as FPGA and its applications in image pattern recognition systems. In reference [5], the authors use a modified rule training (simultaneous perturbation learning rule) for HNN and showe its implementation in an FPGA.

The basic *associative memory* paradigm can be defined as the storage of a set of patterns in such a way that if a new pattern X is presented, the response is a pattern among the stored patterns which closely resembles X. This implies that it is possible to recall the complete pattern even if only part of it is available. This powerful concept can be utilized in many applications such as pattern recognition, image reconstruction from a partial image, noise removal, and information retrieval. The Hopfield Neural Network, a very interesting model of ANNs which was discovered by Hopfield in the 80's [6], can be used as an associative memory and as a solver of combinatorial optimization problems.

HNN consists basically of a number, usually large, of simple processing units (neurons) with small local memory per neuron. These neurons are interconnected via the so-called synapses. Thus, highly parallel computing systems with simple processing elements and point-to-point communication are typical target architectures for efficiently implementing ANNs.

Several mapping schemes have been reported to implement neural network algorithms on parallel architectures.

Examples can be found in [7–17]. In this paper, an implementation of HNN into an SRAM-based FPGA is shown.

In Section 2, the theoretical aspects related to the studied HNN are presented. Section 3 describes the implementation of the HNN according to our approach. Section 4 shows the results of the simulations performed using the ModelSim Simulation tool. In Section 5 our implementation is compared with previous work. The conclusions and planned future researches are discussed in Section 6.

2. Description of the Proposed Algorithm

Hopfield Neural Networks are recurrent artificial neural networks. In this type of ANN, the processing elements are the neurons and every output of each neuron is connected to the input of all other neurons via synaptic weights. All weights are calculated using the Hebbian rule defined as follows:

$$\begin{aligned} &\text{if } i \neq j \quad w_{ij} = \sum_{p=0}^{r} x_i^P x_j^P, \\ &\text{if } i = j \quad w_{ij} = 0, \end{aligned} \tag{1}$$

where $0 < i, j < N + 1$, $X^P = \{x_1^P, x_2^P, \ldots, x_N^P\}$, and $x_i^P \in \{-1, 1\}$; P is number of bits in pattern X and r is number of patterns set.

To provide an efficient design, with less hardware and much faster than existing state-of-the-art designs, we assume that the input patterns, "$a[i]$", are binary numbers. Then we apply 2 to come up with new input patterns, and we continue the iterations until the new input patterns are exactly equal to the old ones

$$a_i[t] = f(h_i[t]) = f\left(\sum_j w_{ij} a_j[t-1]\right). \tag{2}$$

This equation proposes the following computation steps which are performed for $1 \leq j \leq N$.

Step 1. Distribute $a_j[t-1]$ to all elements of column j in the weight matrix $W[t]$.

Step 2. Multiply $a_j[t-1]$ and W_{ij}.

Step 3. Sum the result of the above multiplication along each row of W to compute the weighted sums $h_i[t]$.

Step 4. Apply the activation function $f(h_i[t])$.

Repeat the above steps except that the distribution of $a_j[t-1]$ in Step 1 should be done to all elements of row j instead of column j. This is possible because the weight matrix is symmetric.

Each of Steps 2 and 4 can be performed concurrently in all active Processing Elements (PEs), and therefore, each operation will be performed in parallel taking only one cycle. Figure 1 shows how the multiplications of Step 2 are done in parallel for four nodes HNN. Each node (represented by a circle) corresponds to a PE and is numbered based on the index of the weight (e.g., PE_{23} corresponds to W_{23}).

Considering Step 3, the summations will be performed as follows: each odd PE will add the previous result to its even neighbor and store the result in its even PE. That is, $\text{PE}_{ij} \leftarrow \text{PE}_{ij} + \text{PE}_{i(j+1)}$, where j takes on even values. Then, each PE that is distant of a multiple of four will add its result to the PE two positions apart; that is, $\text{PE}_{ij} \leftarrow \text{PE}_{ij} + \text{PE}_{i(j+2)}$, where j is multiple of four. In step k, each PE that is distant of a multiple of 2^k apart will add its result to the PE 2^{k-1} positions apart to the right; that is, $\text{PE}_{ij} \leftarrow \text{PE}_{ij} + \text{PE}_{i(j+2^{k-1})}$, where j is multiple of 2^k. In this way, for each row the weighted sum $(h_i[t]/1 \leq i \leq N)$ will be computed and stored in each PE of column 0 in $\log N$ steps. The case for the next iteration is similar to the previous case except that the addition of the PEs previous results is performed columnwise instead of rowwise. The process of alternating between rowwise and columnwise summation from iteration to another is repeated until the HNN converges to a solution. The summation method is illustrated in Figure 2 for a row of 8 neurons. Notice that after $\log N$ summation steps each PE in column 1 will have the corresponding weighted sum. As shown, after three summation steps (i.e., $\log N$ steps), the final sum is stored in the PE_{i1} of the corresponding row. A detailed description of the algorithm can be found in [7, 8].

3. Implementation on an FPGA

Considering that the HNN follows two phases in order to effectively complete the work, learning and recognition phases are to be implemented.

3.1. Learning Architecture. The HNN must learn all the combinations that will be stored. The learning process will be executed serially for each of the combinations, thus, for N combinations N clock cycles are needed to finish this process.

An HNN of N-nodes requires N^2 learning units to be capable of calculating the $N \times N$ weight matrix calculated using the Hebbian rule given in (1). Following the same equation, the weight W_{ij} can be calculated by multiplying a_i by a_j and then summing the result with the previously saved weight. Since our inputs are binary, and the learning is bipolar $\{-1, 1\}$, the result of the multiplication should be $1(0 \times 001)$ if a_i is equal to a_j and $-1(0 \times \text{FFF})$ otherwise. Then the adder will add either -1 or 1 to the previously saved weight (Figure 3).

The fact that the weight matrix is symmetric with all W_{ii}'s equal to zero will allow to implement a learning scheme that requires only $(N^2/2) - N$ learning units.

3.2. Recognition Architecture. Two types of cells are used for the implementation of the recognition phase of the proposed algorithm. The first one, the Serial Node (SN), is responsible for both the multiplication of the weights by the input patterns and the addition of two serial data. It is important to note that SNs are used only in Step 1. The second one, called Master Node (MN), is used in the second step to provide the

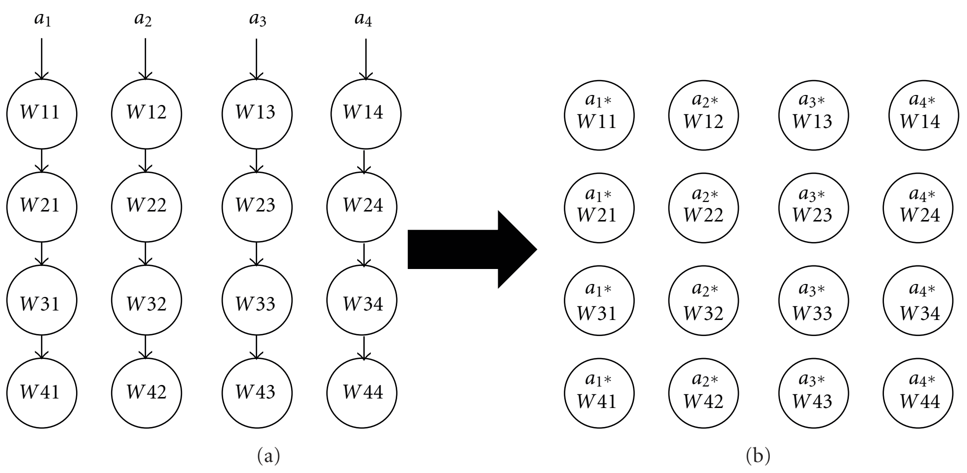

Figure 1: Parallel multiplication of four HNN nodes.

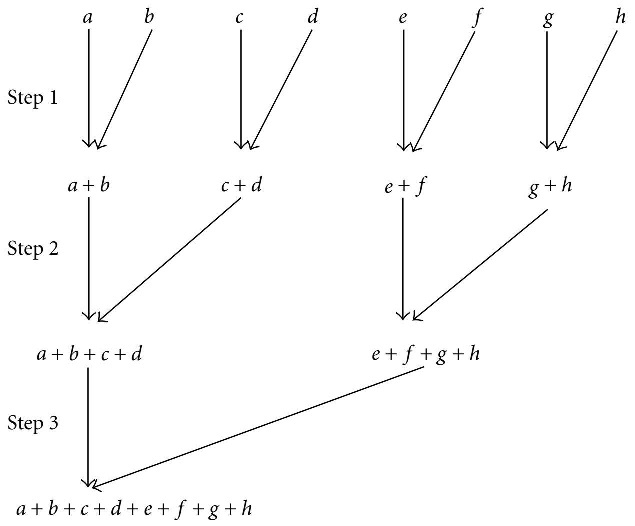

Figure 2: Addition of 8 cells.

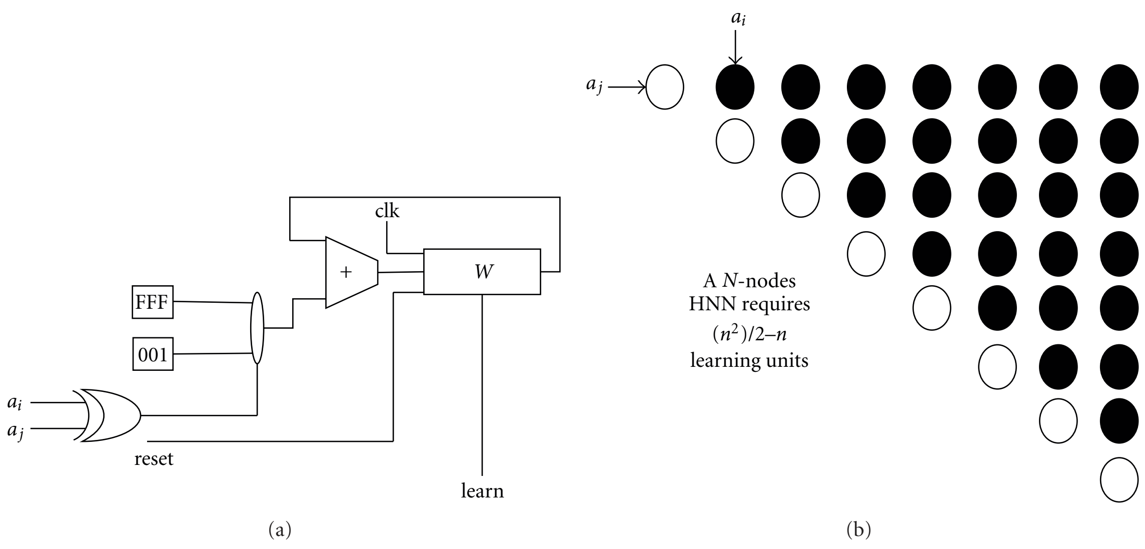

Figure 3: Architecture of the learning unit and its distribution over the network.

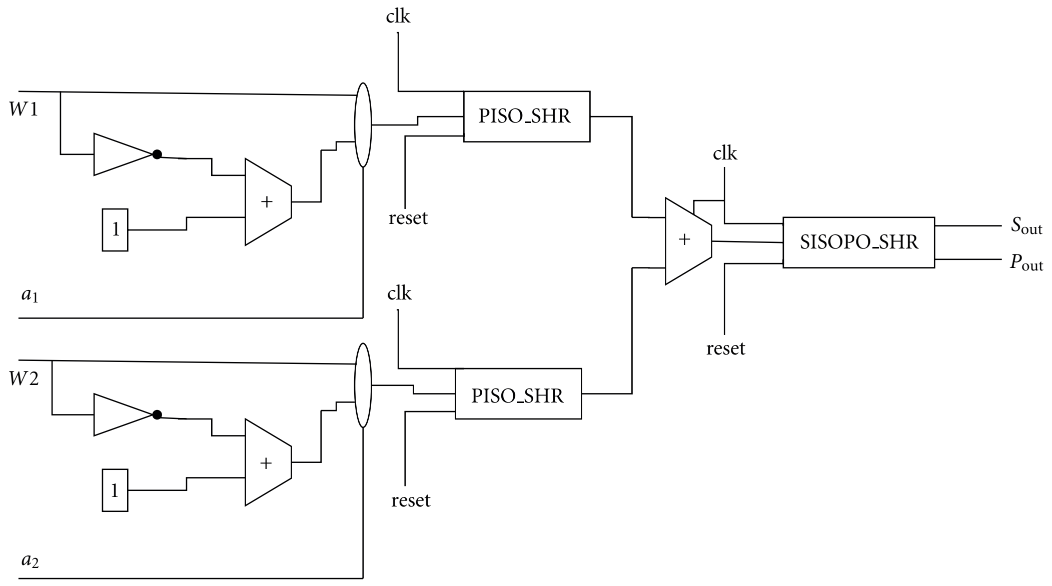

Figure 4: Architecture of the serial node (SN).

addition of two consecutive SNs, and in subsequent steps to provide the addition of two consecutive master cells.

To deal with larger networks, this chain of cells can be increased in a modular way. This would be done at the cost of adding extra hardware and using more cycles to obtain the final result. Each time the number of cells in a row increases by two, one master cell should be added.

3.2.1. The Serial Node (SN). Before going into the implementation details, it is worth mentioning that in order to avoid overflows the calculated weights are stored in 12-bit registers even though each weight can fit in smaller register. This is true even for large size networks. For instance, in the case of a network of 100 patterns, the range of the weights would be between -100 and $+100$. Therefore, using 12-bit registers should be more than enough to avoid overflows.

Owing that a_i are binary inputs and the calculated weights are coded with 12 bits, outputs of the learning unit and the multiplication task can be performed via a multiplexer that selects either the weight or its 2's complement. The first task of SN is to multiply the input pattern by the weight. The result is saved in a 12-bit parallel-in serial-out shift register (PISO_SHR). Then the serial output of this shift register will be added to another serial data. At the end, the result of this addition is saved in a serial-in serial-out parallel-out shift register (SISOPO_SHR) as shown in Figure 4.

3.2.2. The Master Node (MN). Since HNNs are supposed to be implemented on large networks, in Step 1 one MN should be used for every pair of adjacent SNs, while in subsequent steps another MN is used for every pair of adjacent master nodes.

As shown in Figure 5, the basic task of MN is to add two one bit inputs and store the result in a SISOPO_SHR similar to the one used in SNs.

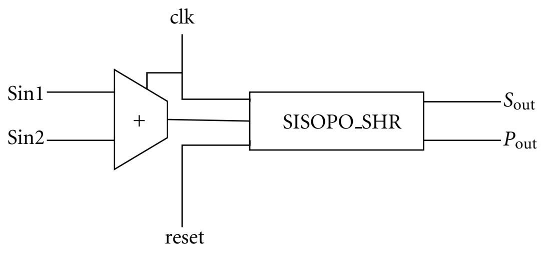

Figure 5: Architecture of the master node.

3.2.3. One Row Architecture. For the calculation of a row consisting of N nodes, SNs are always used in the first step while in all the other steps we use the MN. Figures 6 and 7 illustrate rows of four and eight nodes, respectively. Note that P_{out} outputs of cells other than master node are not used. Therefore they are unconnected. This is not a serious issue because the synthesis tools will optimize them away.

3.2.4. The Last Iteration. In order to stop the process one should wait until the previous input patterns are exactly equal to the new ones. The initial input patterns are used only in the first iteration whereas the recalculated patterns should be used in the following ones. This work is done by a state machine that reads the output of the network, checks it, calculates the new input patterns, and sends them to the network until the stopping condition is satisfied. Figure 8 presents an 8×8 mesh connected to the state machine (SM).

4. Simulation Results

The simulation was performed in many steps to ensure the correctness of the design. First of all, a simulation of the learning process has been done to train a 32-node HNN on three patterns: 0 x 10287C82, 0 x 243C2424, and 0 x

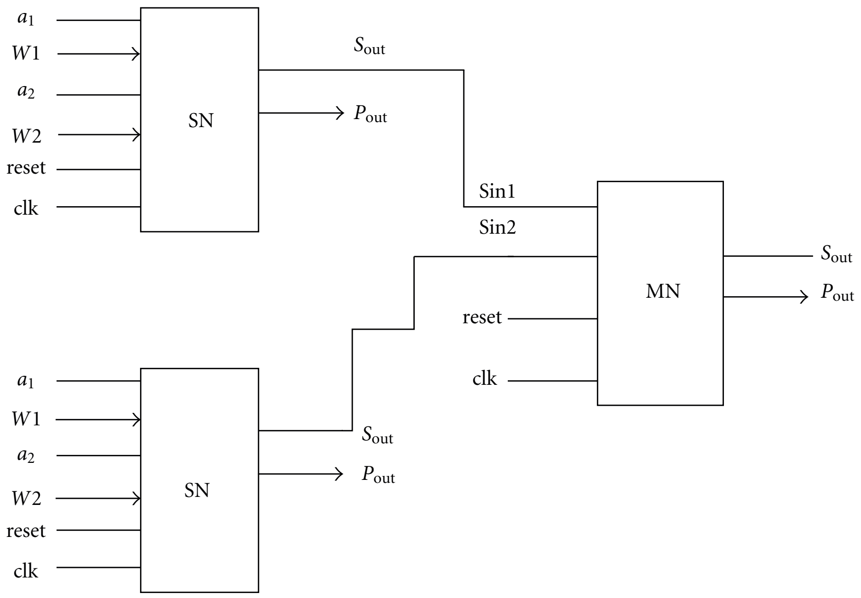

FIGURE 6: A row of eight nodes.

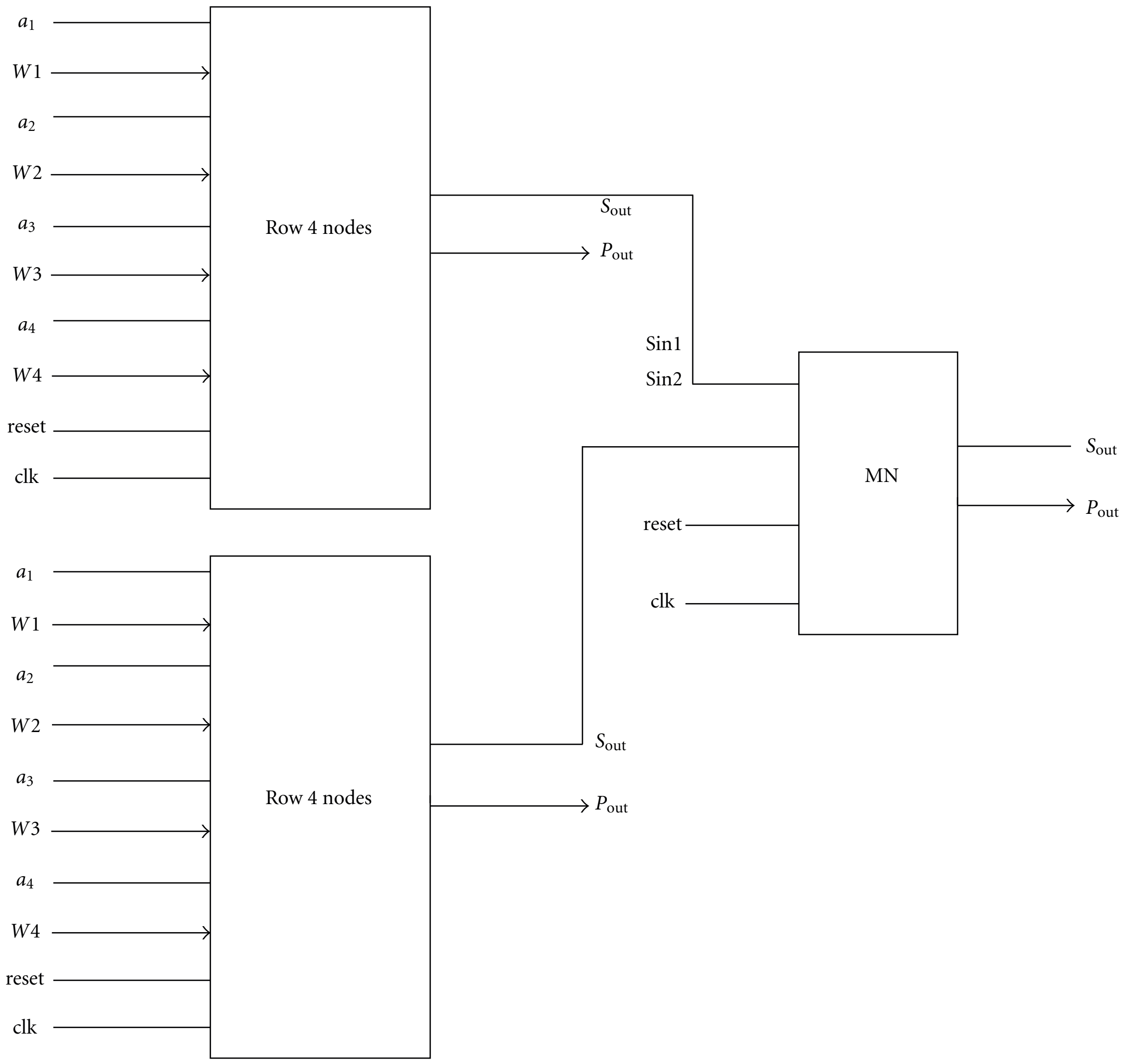

FIGURE 7: A row of four nodes.

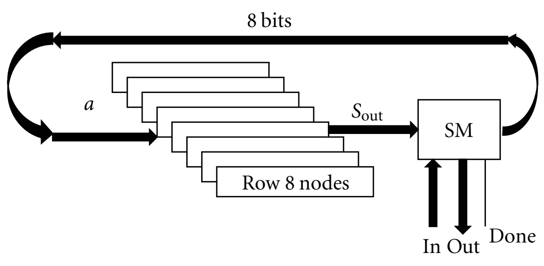

FIGURE 8: 8 × 8 mesh architecture.

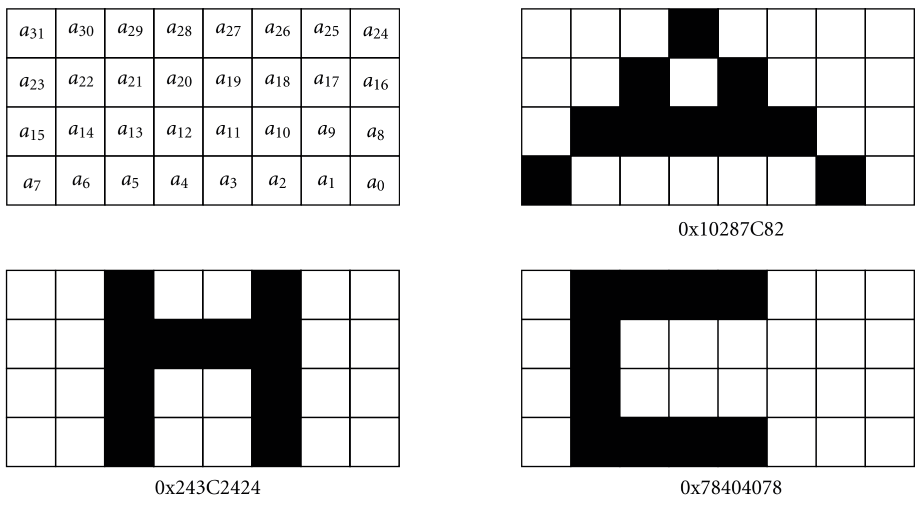
a_{31} a_{30} a_{29} a_{28} a_{27} a_{26} a_{25} a_{24}
a_{23} a_{22} a_{21} a_{20} a_{19} a_{18} a_{17} a_{16}
a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
0x10287C82
0x243C2424
0x78404078

FIGURE 9: 4 × 8 patterns and the three training sets; black = 1, white = 0.

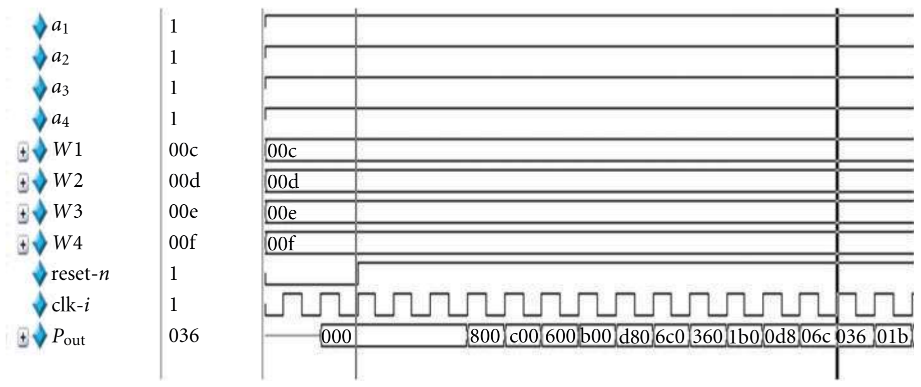

FIGURE 10: Simulation of a HNN's row of four nodes (the result 0 x 36 is the sum of the weights and it's done after 13 cycles).

78404078 (see Figure 9). The Ws correspond to the weights of one row consisting of 384 bits that is sixteen 12-bit each. The result is compared with another obtained by a training software implemented in C# to ensure the correctness of the design.

Figures 10 and 11 show the results of the simulation of a row of 4 nodes and 8 nodes, respectively. One can notice that a row that is twice larger takes two more cycles (serial adder + P_{out} register). The result of W_i is stored in P_{out} register.

After performing the training of the system, a simulation of the overall HNN 32 nodes network ("simmn32") was performed. "sys_o" is the output pattern after a "done" signal is set, while "na" is the next input pattern and the "reset_n_o" is the reset signal of the next MN. Figure 12 shows the simulation of the HNN with correct input patterns. In addition to that, the simulation given in Figure 13, performed with faulty input patterns, shows that the system converges to the expected pattern after two iterations.

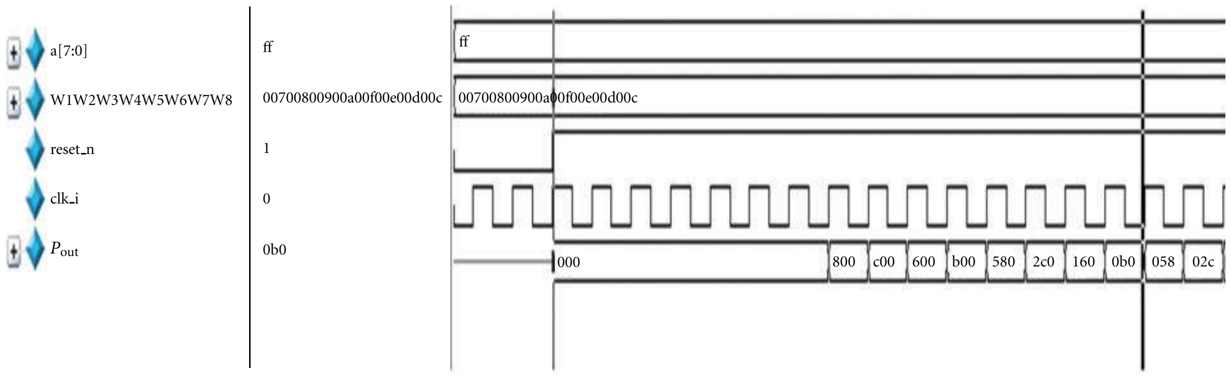

Figure 11: Simulation of a HNN's raw of eight nodes (the result 0 x 58 is the sum of the weights and it is done after 15 = 13 + 2 cycles).

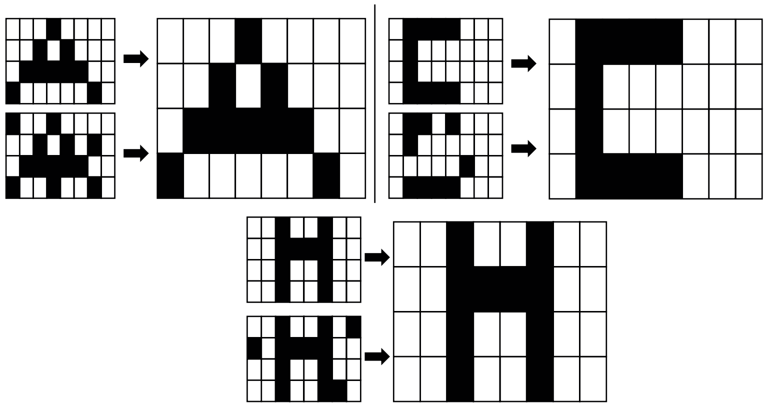

Figure 12: Real data example; good and disturbed inputs versus outputs.

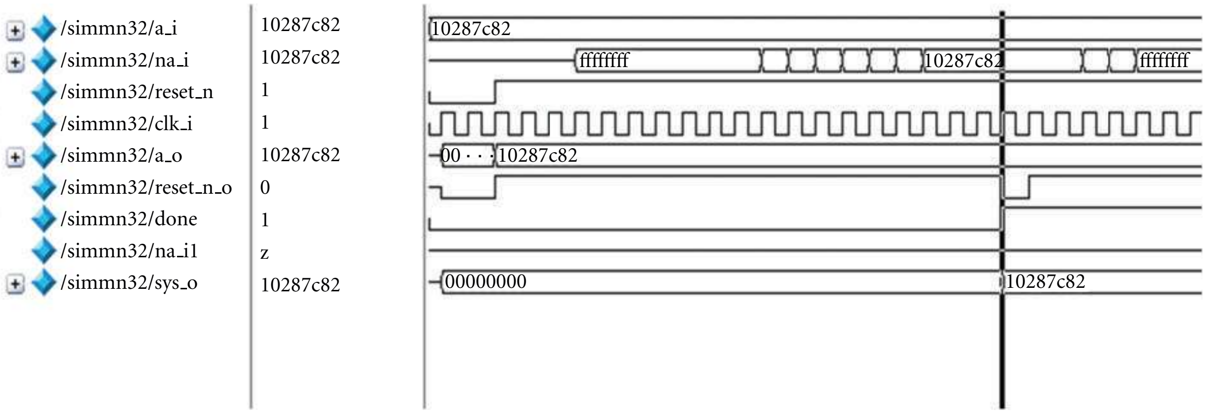

Figure 13: Simulation of 32-nodes HNN.

From Figures 12, 13, and 14, it can be seen that for both input patterns (values 0 x 10287C82 and 0 x 24BC2426), the HNN converges to patterns 0 x 10287C82 and 0 x 243C2424 memorized in the network. Thus, disturbed, missing, and correct patterns can be recovered correctly.

5. Performance and Comparison with Previous Work

From the simulation results presented in the previous section it can be clearly shown that a HNN of four nodes will only

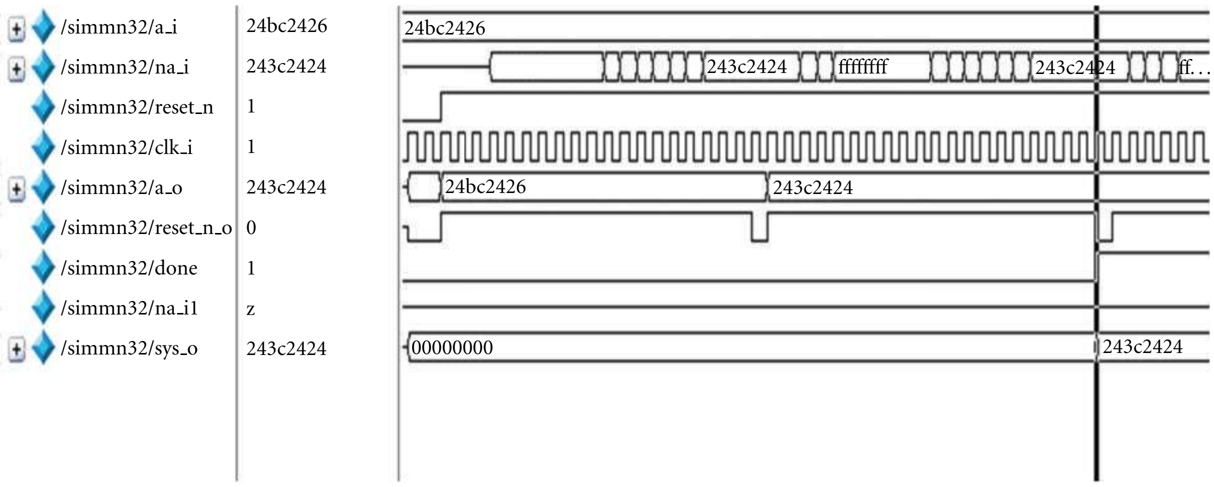

Figure 14: Simulation of 32 nodes HNN for disturbed input patterns.

take 13 cycles. A network of eight nodes will take $13 + 2 = 15$ cycles and in general a network of N nodes (N power of 2) takes $13 + 2 \times \log_2(N/4)$.

Many techniques for mapping ANNs onto parallel architectures have been proposed in the literature. Many of these techniques have been implemented on general-purpose parallel machines. Others were implemented on FPGA architectures.

A comparison with previous work relating general-purpose parallel machines shows the performance superiority of our implementation over known implementations on planar architectures. Most known implementations [11, 13, 14, 16, 17] require $O(N)$ time complexity, whereas the proposed implementation requires $O(\log N)$ time complexity. Implementations on nonplanar architectures, such as hypercube, show a minor performance gain over our design at the cost of much more complex interconnection network [10, 15]. This nonplanar architecture, when implemented on FPGA, requires a complex interconnectivity, which leads to more hardware resources and a lower time performance. This in turn could offset any performance gain.

Several FPGA implementations of ANNs have been reported in the literature [1–5]. Of special interest is the FPGA design proposed by Leiner et al. [1], because it implements the same Hopfield neural model. Both implementations targeted the same FPGA device, which is a Spartan2 chip the XC2S200. The implementation by Leiner et al. takes $O(N)$ multiplication and summation steps. Therefore, our implementation shows a higher performance. Moreover, we were able to achieve a clock rate of more than 157.604 MHz in contrast to a 50 MHz by Leiner et al.

6. Conclusions and Future Work

The implementation of an efficient algorithm for mapping into an FPGA the operation of the Hopfield associative memory was presented in detail. The time complexity is of order $O(\log N)$, which is better than any known algorithm on planar architecture and achieves the same performance of higher degree architecture such as hypercubes without the added cost.

Future work will explore a similar design of an HNN including fault tolerance.

References

[1] B. J. Leiner, V. Q. Lorena, T. M. Cesar, and M. V. Lorenzo, "Hardware architecture for FPGA implementation of a neural network and its application in images processing," in *Proceedings of the 5th Meeting of the Electronics, Robotics and Automotive Mechanics Conference (CERMA '08)*, pp. 405–410, October 2008.

[2] S. Saif, H. M. Abbas, S. M. Nassar, and A. A. Wahdan, "An FPGA implementation of a hopfield optimized block truncation coding," in *Proceedings of the 6th International Workshop on System on Chip for Real Time Applications (IWSOC '06)*, pp. 169–172, December 2006.

[3] S. Saif, H. M. Abbas, S. M. Nassar, and A. A. Wahdan, "An FPGA implementation of a neural optimization of block truncation coding for image/video compression," *Microprocessors and Microsystems*, vol. 31, no. 8, pp. 477–486, 2007.

[4] M. Stepanova and F. Lin, "A hopfield neural classifier and its FPGA implementation for identification of symmetrically structured DNA motifs," *Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 239–254, 2007.

[5] Y. Maeda and Y. Fukuda, "FPGA implementation of pulse density hopfield neural network," in *Proceedings of the International Joint Conference on Neural Networks*, Orlando, Fla, USA, August 2007.

[6] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, 1982.

[7] R. A. Ayoubi and M. A. Bayoumi, "An efficient implementation of multi-layer perceptron on mesh architecture," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 2, pp. 109–112, May 2002.

[8] R. A. Ayoubi, H. A. Ziade, and M. A. Bayoumi, "Hopfield associative memory on mesh," in *Proceedings of the IEEE International Symposium on Cirquits and Systems*, pp. 800–803, May 2004.

[9] J. Hwang and S. Kung, "Parallel algorithms/architectures neural networks," *Journal of VLSI Signal Processing*, 1982.

[10] K. Kim and V. K. P. Kumar, "Efficient implementation of neural networks on hypercube SIMD arrays," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, pp. 614–617, Washington, DC, USA, 1989.

[11] Y. Kim, M. J. Noh, T. D. Han, and S. D. Kim, "Mapping of neutral networks onto the memory processor integrated architecture," *Neutral Networks*, no. 11, pp. 1083–1098, 1988.

[12] S. Y. Kung, "Parallel architectures for artificial neural nets," in *Proceedings of the International Conference on Systolic Arrays*, vol. 1, pp. 163–174, San Diego, DC, Calif, USA, 1988.

[13] S. Y. Kung and J. N. Hwang, "A unified systolic architecture for artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 6, no. 2, pp. 358–387, 1989.

[14] W. M. Lin, V. K. Prasanna, and K. W. Przytula, "Algorithmic mapping of neural network models onto parallel SIMD machines," *IEEE Transactions on Computers*, vol. 40, no. 12, pp. 1390–1401, 1991.

[15] Q. M. Malluhi, M. A. Bayoumi, and T. R. N. Rao, "Efficient mapping of ANNs on hypercube massively parallel machines," *IEEE Transactions on Computers*, vol. 44, no. 6, pp. 769–779, 1995.

[16] S. Shams and J. L. Gaudiot, "Implementing regularly structured neural networks on the DREAM Machine," *IEEE Transactions on Neural Networks*, vol. 6, no. 2, pp. 407–421, 1995.

[17] S. Shams and K. W. Przytula, "Mapping of neural networks onto programmable parallel machines," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 2613–2617, New Orleans, La, USA, May 1990.