Hardware implementation of different neural network with on-chip learning

ALIN TISAN, STEFAN ONIGA, CIPRIAN GAVRINCEA
Electrotechnical Department
North University of Baia Mare
Victor Babes st., no 62A, Baia Mare
ROMANIA
atisan@ubm.ro http://ea.ubm.ro/ee

Abstract: - In this paper we propose a method to implement different neural network in FPGA circuits: a feedforward neural network with on-chip delta and Hebbian rule learning algorithm. The method implies the building of a neural network by generic blocks designed in Mathworks' Simulink environment. The main characteristics of this solution are on-chip learning algorithm implementation and high reconfiguration capability and operation under real time constraints.

Key-Words: - MLP, learning on-chip, Delta rule, Hebbian rule, ANN, FPGA

1 Introduction

In response to highly parallelism, modularity and dynamic adaptation, the artificial neural network (ANN) become the most explored data processing algorithms. In addition to this, the digital hardware implementation of ANNs in reconfigurable computing architectures like FPGAs circuits, become the easiest and fastest way to reconfigure in order to adapt the weights and topologies of an ANN.

In this paper we present an extendable digital architecture for the implementation of a multilayer feedforward networks (MLF) and Hebbian neural network using field programmable gate arrays (FPGAs) and propose a design methodology that allows the system designer to concentrate on a high level functional specification. For this reason we developed a new library Simulink block set constituted by Simulink Xilinx blocks and VHDL blocks. With these new created blocks, the designer will be able to develop the entire neural network by parameterize the ANN topologies as number of neurons and layers.

The implementation goal is achieved using the Mathworks' Simulink environment for functional specification and System Generation to generate the VHDL code according to the characteristics of the chosen FPGA device.

2 Hebbian Neural Network

The Hebbian neural network is a multilevel model of perception and learning. Its main characteristic is given by the increasing of the coupling between coactive cells so that they could be linked in growing assemblies. Denoting the neurons by n_i and n_j and the weight that connect the n_j and n_i by w_{ij} and if neuron n_i receives positive input x_j while producing a positive output y_i, the hebbian rule states that for some learning rate $\eta > 0$:

$$w_{ij} := w_{ij} + \eta y_i x_j, \quad (1)$$

Here

$$y_j = f(net) = f\left(\sum_{i=1}^{N} w_{ij} x_i - b\right) \quad (2)$$

where f(net) is defined by the discontinuous threshold activation function sgn(net):

$$\mathrm{sgn}(net) = \begin{cases} 1 \text{ if } net \geq 0 \\ -1 \text{ if } net < 0 \end{cases} \quad (3)$$

Of all the learning rules, Hebbian learning is probably the best known. It established the foundation upon which many other learning rules are based.

3 Multilayer Feedforward Network

The MLP network generally gives quickly results, is efficient with information processing, and learns by presenting examples; but sometimes is difficult to choose the optimal network parameters and training procedures for a given situation. From this reason, the building of a neural network with customizable blocks can give a higher reconfiguration capability and operation of the neural network under real time constraints.

The most difficulty parts in FPGA implementation of the MLP network are the sigmoid function and the calculus algorithm (Delta rule) of the weights, [1].

In MLP networks, the basic units, the neurons, are organized in, at least, 3 layers: one input layer, one output layer and one or more intermediate, hidden layers. Networks are typically fully connected, i.e., all outputs of a layer are connected by synapses to all inputs of the following layer. Only the hidden and the output layers include processing units, whereas the input layer is used just for data feeding.

The adopted neuron model is classical one's and is basically made of two blocks: First block is responsible for calculating the summation of all inputs and bias, after multiplying each one by it's weight, giving the *net* value:

$$net = bias + \sum_{k=1}^{N} w_k x_k \quad (4)$$

The later block computes the neuron output, based on its net value. It models the firing nature of the neuron that activates once the sum is above of a given threshold. Because the backpropagation learning algorithm requires an activation functions that is continuous and differentiable, the sigmoid function was chosen, [2]:

$$output(net) = \frac{1}{1+e^{-net}} \quad (5)$$

The backpropagation learning algorithm consists of four different steps: (forward) propagation, error check, back propagation (BP) and weights update.

First, the input values are propagated forward, using equations (4) and (5), to obtain the actual outputs.

Second step requires calculating the average of the total as a sum of squared individual errors.

$$E = \sum_p E^p = \frac{1}{2}\sum_p (T^p - O^p)^2 \quad (6)$$

where *O* is the actual output, *T* is the desired value for a given pattern *p*.

Next, if the total error is greater then a given threshold the value of the weights must be adjusted in order to minimize the error function. This minimization is done in idea to make changes in the weight proportional to the negative of the derivative of the error - backpropagation step, [3], [4], [5], [6].

In the backpropagation step, the first thing to do is to obtain the gradient of the output neurons. This is done through:

$$\delta = -\frac{\partial E}{\partial net_k} = -\frac{\partial E}{\partial O_k}\frac{\partial O_k}{\partial net_k} \quad (7)$$

For

$$f'(net) = \frac{\partial O_k}{\partial net_k} \quad (8)$$

and

$$\frac{\partial E}{\partial O_k} = -(T_k - O_k) \quad (9)$$

we obtained

$$\delta_k = (T_k - O_k) f'(net) \quad (10)$$

where *O* is the actual output, *T* is the desired value and *f'(net)* is the first derivative of the activation function and *k* is the number of the output neurons.

In the case of the sigmoid function the derivative is given by:

$$f'(net) = f(net)(1 - f(net)) = O(1 - O) \quad (11)$$

This value is then propagated backwards in order to obtain the gradient for each of the hidden layer neurons. For a given hidden neuron, its gradient is given by

$$\delta_j = f_j'(net_j)\sum_{k=1}^{K} \delta_k w_{kj} \quad (12)$$

The last step consists of updating the weights in order to minimize the error.

In the case of the neurons from the output layer the formula that gives the change of corresponding weight is:

$$\Delta w_k = \eta \delta_k y_j \quad (13)$$

where y_j is the output of the neuron *j* from the previous layer.

In the case of the neurons from the hidden layers the change of corresponding weight is:

$$\Delta v_k = \eta \delta_j z_i \quad (14)$$

where z_i is the output of the neuron *i* from the previous layer.

Because the BP learning algorithm is an iterative process the new epochs are repeated until the network is trained enough, i.e., until the error between the actual and the desired outputs is lower than a given threshold.

4 Neural Network Blockset Design

In order to learn on-chip, the Mc Culloch - Pitts neuron model, i.e. each of the input vector components x_i is multiplied with the corresponding weight w_{ij}, and these products are summed up yielding the net linear output, upon which the threshold activation function is applied to obtain the activation, was modified to make the calculus of the weights according to a certain learning rule and to update the new weights into a weight memory block, figure 1.

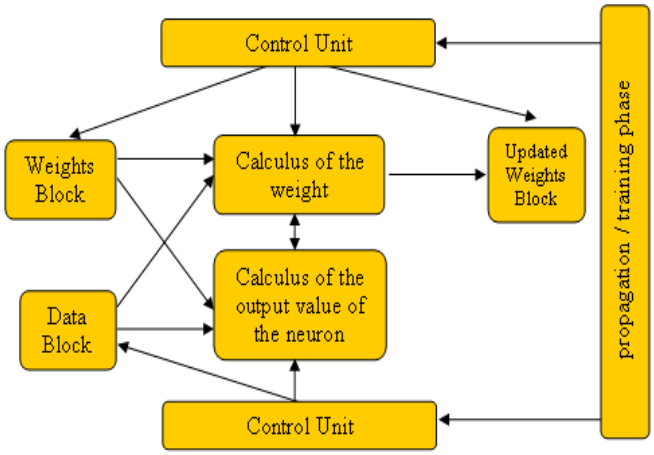

Fig.1. Block level representation of the neuron with on-chip learning

The parallelism adopted is a node parallelism one and requires one multiplier per neuron; therefore all neurons will work in parallel. If data inputs are memorized in a

single memory block, the weights storage will be private for each neuron because all the neurons have to access their correspondent weight memories at the same time.
The proposed model is constituted by two major blocks: a control logic bloc and a processing block.
The control logic block will manage the control signal of the processing bloc in order to initialize and command the processing components.
The processing block is designed to calculate the neural output, the weights according to learning rule adopted, in this case Delta and Hebbian rule, and to update these weights.

4.1 The Hebbian network design

4.1.1 Control logic block

The control logic block is described in VHDL code and is incorporated into design by a black box HDL, figure 2.
The role of this bloc is to load from Mathlab workspace the following variables: the number of vectors used for training and the number of bits used for data representation.
Depending on these variables, the control logic block will configure the size of the RAMs used for data and weights storage and will manage the enable signals of the processing elements of the processing block in order to run the processing block in a propagation phase or in a training phase

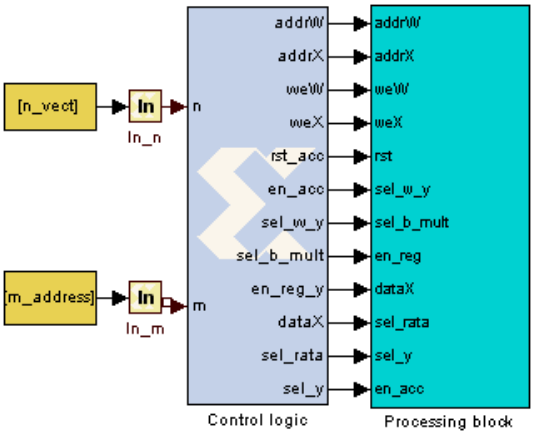

Fig. 2. Blockset architecture of neuron

4.1.2 Processing blocks

The processing block is the main block of the design. It incorporates both the artificial neuron and the logic for on-chip learning algorithm.
The structure of the artificial neuron consist in two memory blocks, one for data samples and one for weight coefficients, and one MAC unit, figure 3.
The logic for the learning algorithm requires a MAC unit too, but in order to save hardware resources we decide to use the MAC unit of the artificial neuron for the implementation of the learning algorithm. To achieve this, the design requires a number of multiplexer blocks and special control logic.

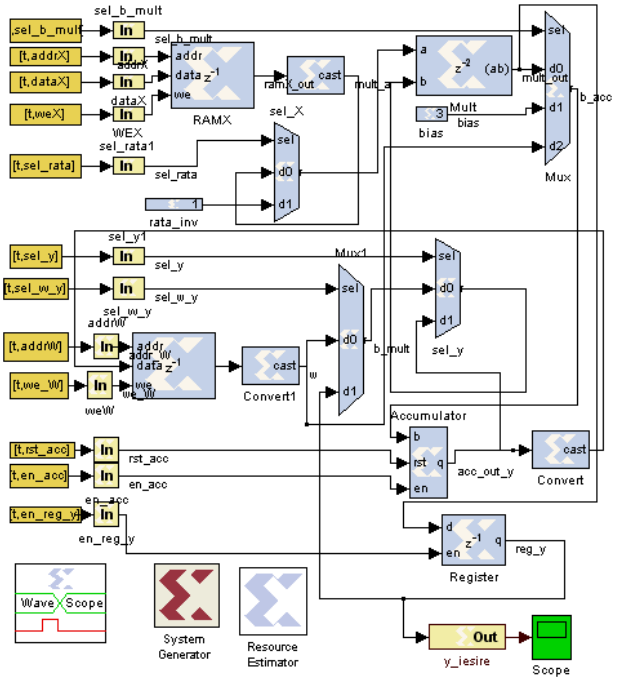

Fig. 3. Architecture of the processing block.

Artificial neuron with on-chip learning has two modes of operation: propagation mode on which the design acts as a regular artificial neuron and learning

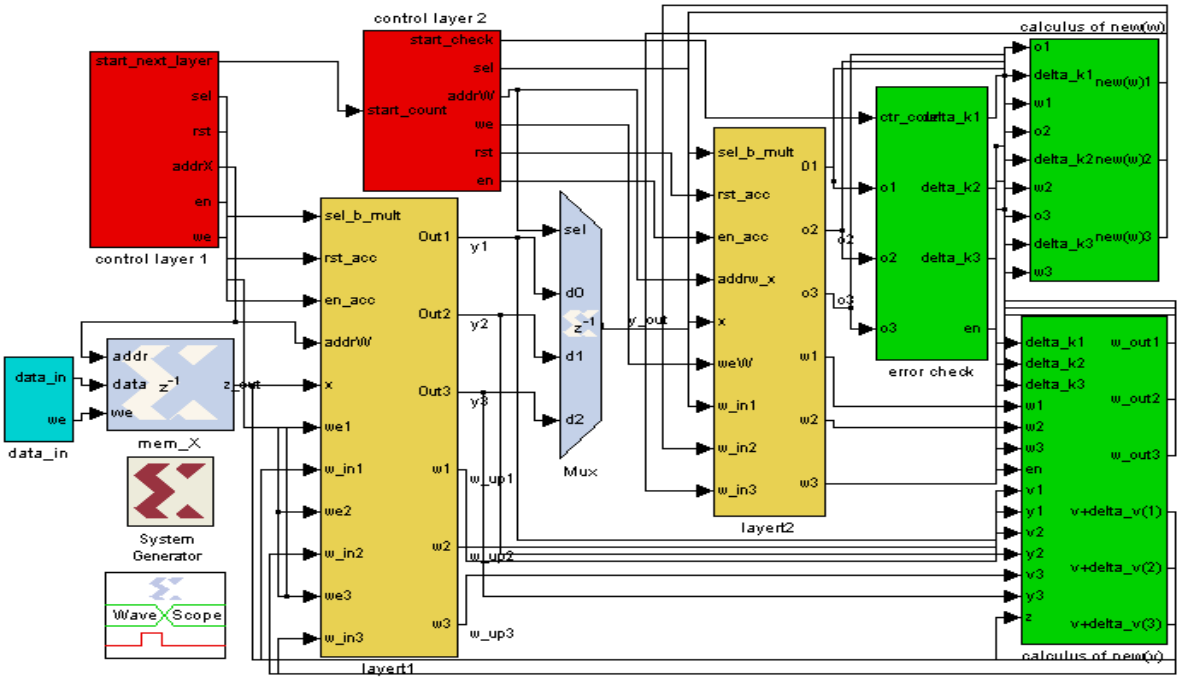

Fig. 4 Neural network with one hidden layer

mode. On learning mode there are two stages: on the first stage an output of the artificial neuron is calculated based on the data provided by training vector and on the second stage weight coefficients are recalculated based on Hebbian learning rule.
By using one MAC unit it saves hardware resources but execution time slightly increases. This happens only on the learning mode and doesn't affect the propagation mode. For an artificial neural network time constrains are important only on propagation mode of operation, and having a longer learning period doesn't affect the performance of the artificial neural network.

4.2 The MLP network design

The overview architecture of the MLP neural network is presented in figure 4.

4.2.1 Control logic block

The control logic block is designed to control the processing units, the neurons, from neural layers. In this way this block will gives the control signal to the multipliers, summation and RAM blocks in order to compute the *net* value of the neuron.
Because, hidden layer has to subordinate to the previous layer there are 2 different type of control logic block: one for the first layer (first hidden layer) and one for the following layers (the following hidden layers and for the output layer), fig 5 and fig 6.

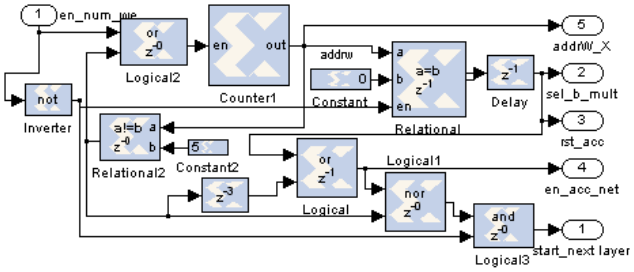

Fig 5. Control logic block for the first layer

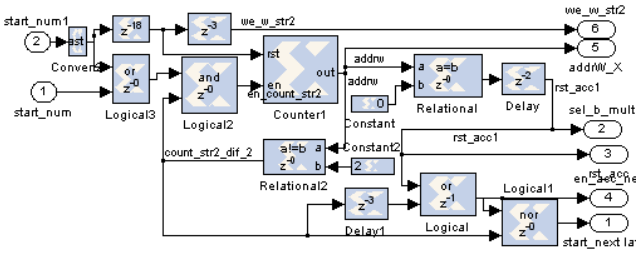

Fig 6. Control logic block for the subsequent layers

The control logic block will also control the data memory, used as a buffer for data that comes from outside (sensors output), in order to give the right address of the data that will fed the input layer of the neural network.

4.2.2 Processing blocks

The processing blocks are the main blocks of the design. Its incorporate both the artificial neuron and the logic for on-chip learning algorithm.

4.2.1 Neuron model

The structure of the artificial neuron consist in one memory block, for data samples, one MAC unit and an activation unit, fig 7

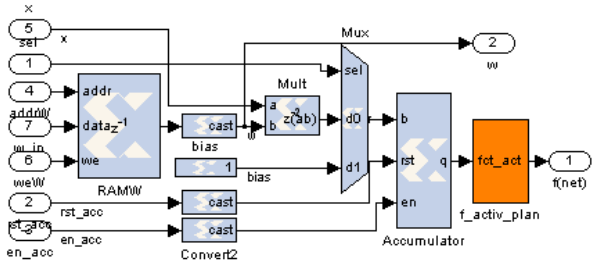

Fig 7. Architecture of the neuron

For implementation in FPGA of the activation function was used the PLAN approximation (Piecewise Linear Approximation of a Nonlinear function), proposed by Amin, Curtis and Hayes–Gill [7]. The PLAN approximation uses digital gates to directly transform from x to y. The approximation of the sigmoid function is presented in Table 1. The calculations need only be performed on the absolute value of the input x.

Condition	Approximation
$\|x\| \geq 5$	$y = 1$
$2.375 \leq \|x\| < 5$	$y = 0.03125 \cdot \|x\| + 0.84375$
$1 \leq \|x\| < 2.375$	$y = 0.125 \cdot \|x\| + 0.625$
$0 \leq \|x\| < 1$	$y = 0.25 \cdot \|x\| + 0.5$

Table 1

The error function of the sigmoid function approximations is presented below, fig. 8.[8]

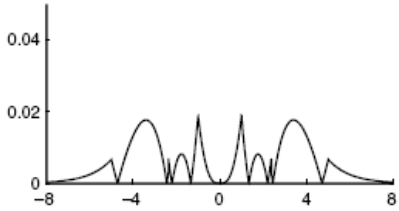

Fig 8, Error function of sigmoid function approximations

In figure 9 is shown the hardware architecture of the PLAN approximation of sigmoid function. The architecture includes beside comparators, multipliers, multiplexers and adders blocks, a black box that contains a VHDL file that control the multiplexer outputs data.

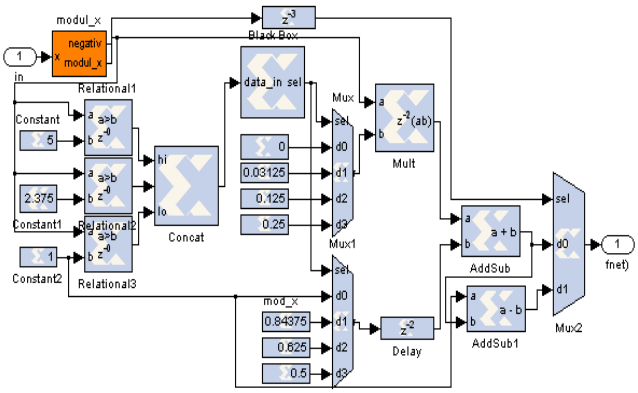

Fig. 9. The hardware architecture of the PLAN approximation of sigmoid function

3.3.2 Error check block

Another important block is the Error check block that deals with the calculus of the total error, the comparing with the given threshold E_{max} and the calculus of the error signal of the *k*th neuron, (delta_k), fig. 10

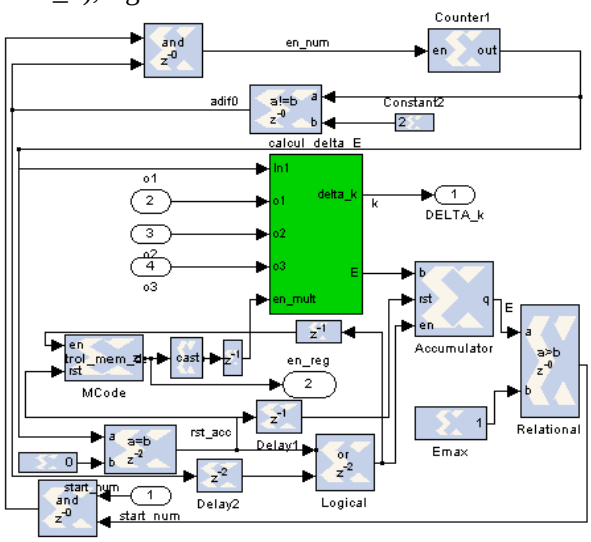

Fig. 10 Hardware architecture of the Error check block

3.3.3 Calculus block of the output layer weights

This block deals with the calculus of the new weights of the output layer.

The block implements the calculus formula of the weights according to the above formula (13). The parallelism is the layer's one because the calculus is done in the same time for all the neurons from the same layer, fig. 11.

The block is reconfigurable and resizable and depends of number of neurons from the output layer. Because the weight from the previous layers depends on the former weights of the output layer this block will begin to calculate and update the new weights only after the weights from the previous layer ware calculated.

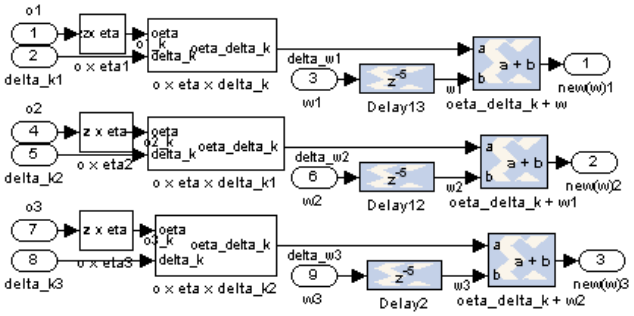

Fig. 11 Hardware implementation of the weights calculus block

3.3.3 Calculus block of the hidden layer weights

This block contains in fact other three blocks that will calculate the weights according to the above formulas ec. (12), (13) and (14), fig. 12

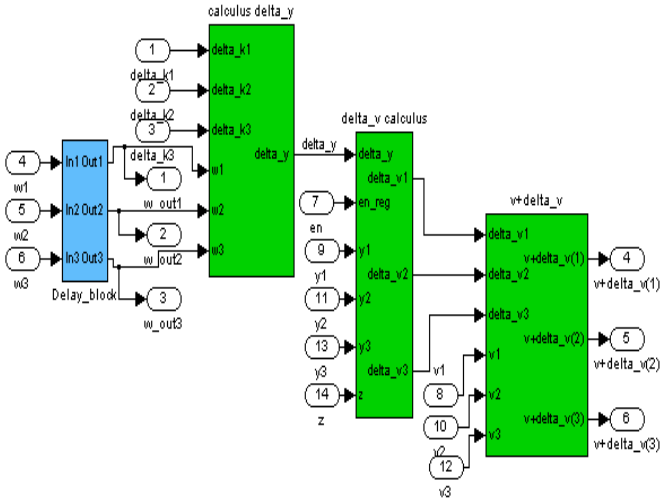

Fig. 12 Block configuration of the Calculus block of the hidden layer weights

All blocks implied in the calculus are resizable and can be changed by the parameters given from outside.

Every calculus blocks strictly respects the calculus algorithm presented above for the each layer apart

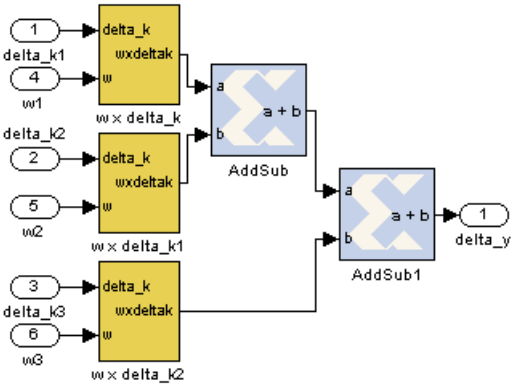

Fig. 13 Hardware implementation of the Delta_y calculus block

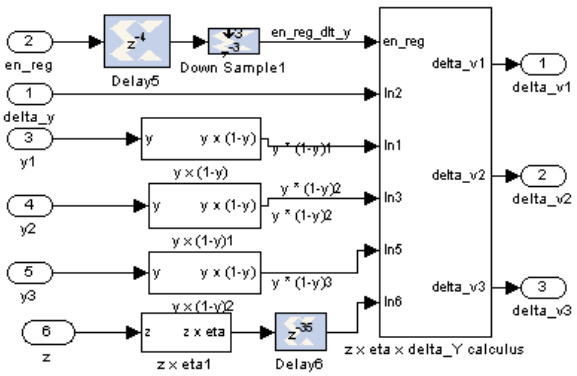

Fig. 14 Hardware implementation of the Delta_v calculus block

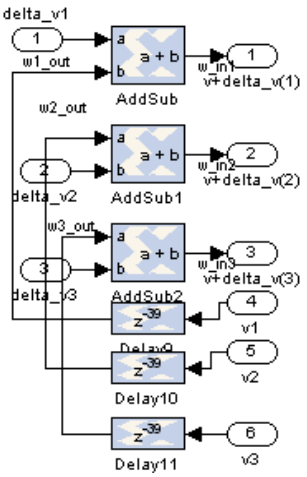

Fig. 15 Hardware implementation of the changed weights

The above figures presents the implementation of the following formulas:

$$\delta_y = y_j(1-y_j)\sum_{k=1}^{K}(T_k - O_k)(1-O_k)O_k w_{kj} \quad (15)$$

for the Delta_y calculus block, figure 13,

$$\Delta v = \eta \delta_{ok} z_i \quad (16)$$

for the Delta_v calculus block, where z_i are the values of the input pattern, figure 14
and

$$v^{new} = v^{old} + \Delta v \quad (17)$$

for the calculus of the updated weight, figure 15.

4 Conclusion

We have presented hardware architecture of artificial neuron with on-chip learning controlled by a generic control unit described in VHDL code. This method uses minimal hardware resources for implementation of this kind of artificial neuron. The main advantage of this solution is high modularity and versatility in neural network designing.

In order to design and to implement the neuron we used the Mathworks' Simulink environment for functional specification, System Generation to generate the VHDL code according to the characteristics of the chosen FPGA device and ISE Xilinx to simulate the design at different stages of implementation and to generate the bit file.
The neuron designed is a generic module and can be used to design neural networks that have the following features:

- the training is on-line;
- the learning is on-chip;
- all weights have been initialized prior to the start of the learning process;
- the learning parameter must be specified precisely;
- there must be some type of normalization associated with the increase of the weight or else wij can become infinite;
- the initialization of the neurons number per layer, number of layers, data and weights RAMs must be done by setting the variables from the Function Block Parameters from Matlab environment.

Both designs are implemented into Digilab 2E (D2SB) development board featuring the Xilinx Spartan 2E XC2S200EPQ208-6 FPGA. This chip has 2352 slices (control unit which includes two 4-inputs look-up tables (LUT) and two flip-flops) and 14 block RAMs. The resources were estimated with Simulink Resource Estimator Block and are shown in fig. 16 and 17

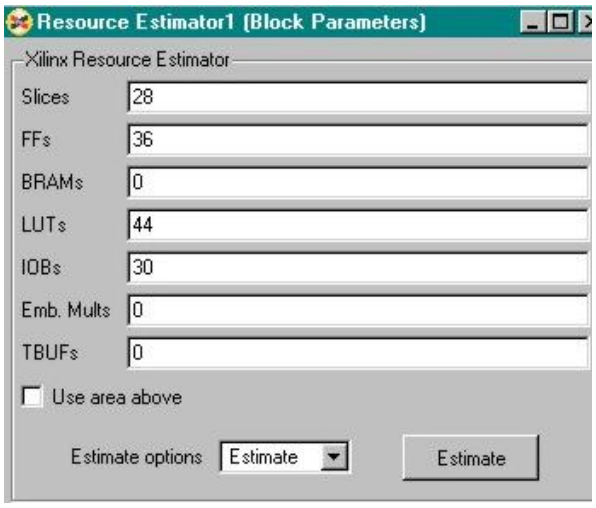

Fig. 16. The resource estimation by Simulink Resource Estimator of a single neuron with Hebbian learning rule

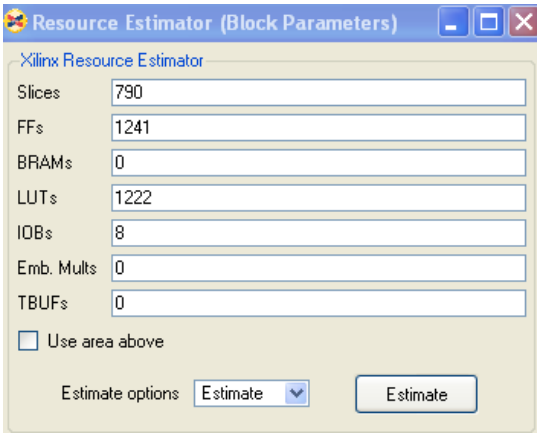

Fig 17. The resource estimation by Simulink Resource Estimator of a neural network with 2 layers with 3 neurons per layer

References:

[1] E. Fiesler, R Beale *Handbook of neural network*, Oxford University Press, 1997.

[2] A. Singh, *Design & Implementation of Neural Hardware* University School of Information Technology, GGS Indraprastha University, Delhi http://www.geocities.com/aps_ipu/papers/synopsis.pdf, 2005.

[3] A., Bernatzki, W, Eppler, *Interpretation of Neural Networks for Classification Tasks.* Proceedings of EUFIT 1996, Aachen, Germany, http://fuzzy.fzk.de/eppler /postscript/eufit.ps.2005

[4] A. Savran, S. Unsal. *Hardware Implementation of a Feedforward Neural Network using FPGAs*. International Conference on Electrical and Electronics Engineering. Bursa, December 2003.

[5] Yihua Liao. *Neural Networks in Hardware: A Survey*. Department of Computer Science, University of California, Davis.

[6] J. Zhu, P. Sutton. *FPGA Implementations of Neural Networks – a Survey of a Decade of Progress*. Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, Sep 2003.

[7] Amin, H., Curtis, K.M., and Hayes–Gill, B.R.: *Piecewise linear approximation applied to nonlinear function of a neural network*, IEE Proc. Circuits, Devices Sys., 1997, 144, (6), pp. 313–317

[8] M.T. Tommiska *Efficient digital implementation of the sigmoid function for reprogrammable logic*. IEE Proceedings – Computers and Digital Techniques 150, number 6, pages 403-411.