

DISCRETE DEVICE REALIZED BY NEURAL NETWORKS

R. Kohut, B. Steinbach

Freiberg University of Mining and Technology, Freiberg, Germany

This paper presents a discrete device for neural network realized on field-programmable gate arrays (FPGA). A basic element of the implemented neural network is new type of neuron, called Boolean neuron that may be mapped directly to configurable logic blocks (CLB) or to look up table (LUT) of FPGAs. The structure and logic of the Boolean neuron allows a direct representation of the Boolean neural network (BNN) architecture to FPGAs. A new training algorithm for BNNs is suggested. This training allows the restriction of the number of inputs of Boolean neurons to the inputs number in a logic block of FPGA. The huge benefit compared to existing approaches of neural network implementations on FPGAs was achieved.

Introduction

The majority of artificial neural networks (NN) applications are implemented in software running on a single processor. The reason for that is the flexibility of software for validation and debugging. However, although the NNs are commonly used in software implementation, specialized hardware offers tangible advantages in several situations [1]. The most important reasons for using specialized NN hardware are higher speed or lower cost.

Parallelism, modularity and dynamic adaptation of neural networks are successfully used in hardware applications of NNs [1]. The basic elements of hardware devices work in parallel. The behavior of these elements can be programmed independently from each other or selected by the types of the gates. The functional elements are connected via interconnection elements in parallel structures. Interconnections may be realized by fixed wires or customized using preconfigured wires and switches.

There are three types of hardware NN implementations: digital, analogue and hybrid [1]. The digital devices are most commonly used in a practice. As core of run-time reconfigurable (RTR) computer architectures field programmable gate arrays (FPGA) become increasingly popular for prototyping and design of complex hardware systems. One reason for that is the growing efficiency and availability of FPGAs and associated boards. The flexibility, capacity, and performance of FPGA devices influence strongly the run-time of reconfigurable systems. Therefore, the hardware NNs are commonly realized as discrete devices on FPGA [2]. Boolean Neuron (BN) and Boolean Neural Networks (BNN) presented in this paper are very well suited for implementation on FPGA, too.

However, years after the first NN application of NN on FPGA, the FPGA realization of NN with a large number of neurons is still a challenging task. NN algorithms require many multiplications and it is relatively expensive to implement multipliers on fine-grained FPGAs [1]. The number of logic gates implemented on a FPGA restricts the systems size whereas neural computations require area-consuming operators like multipliers [2]. Connectivity problems are not solved in general for reconfigurable FPGAs [3, 4, 5, and 6]. FPGA implementations of NNs face to a wide gap between the capabilities offered by the hardware and the availability of mature tools for the development of their targeted application domains.

In order to solve these problems, we developed a new neuron structure and employed this new type of neuron for the building of neural networks. These neural networks can be optimally mapped into a logic structure of FPGA. The presented neuron can be implemented directly into a single LUT. The whole neural network is realized as discrete device on FPGA.

1. Boolean Neuron

1.1 Motivation

Disadvantages of existing FPGA approaches were described in several papers. The realization of one usual neuron requires tens or hundreds of configurable logic blocks (CLB). Well-known implementation of a simple three layer feed forward network GANGLION [2] needs 640 to 784 CLBs for a single neuron. Michael Gschwind suggests in [7] a bit-stream approach that requires 22 CLBs to implement a neuron. In the report [8] 51 CLBs are used for implementation a single neuron. Recently it was presented in [9] a FPGA implementation of a Hopfield neural network containing 64 neurons. Each neuron requires 26 CLBs and the whole neural net requires 1664 CLBs. One additional design problem is the mapping of one single neuron to a multilevel structure of CLBs. Obviously such an implementation slows down the computation speed neural network. Let us analyze this problem.

The structure of a FPGA is an “array of logic blocks” connected together via programmable interconnections. The basic elements of FPGAs are configurable logic blocks that typically contain a number of lookup tables (LUT), an optional D-Flip-Flop (DFF) and additional control logic. In our case 4 LUTs are included in one CLB. A LUT is a 4 input logic block that can implement any Boolean function of four variables.

The input signals and weight coefficients of a usual neuron are real numbers. The output of a neuron is determined by a transfer function. It can be a real, an integer or a Boolean function, but in any case this function operates with data not belonging to the Boolean type. Since the elements of FPGAs operate with Boolean signals, the using of other types of values for the parameters of neurons is inefficient. Unnecessary memory expenses occur for the mapping of such neurons to FPGAs. Moreover, the number of inputs of a neuron is much larger than a number of inputs of a LUT. This analysis discloses three main sources of the inefficient mapping of a classical NN into an FPGA, which are

- data type real of input and outputs data of neurons and networks,
- difficult transfer function of neurons and
- large number of inputs in the neurons.

In order to decrease the number of CLBs, required for the implementation of a single neuron, we suggested functional changes of the usual neural element and developed a Boolean neuron.

1.2 Boolean Neuron

A Boolean neuron (or Boolean neural element) operates with Boolean signals and uses only Boolean operations. The output signal of the Boolean neuron is defined as:

$$y = f(\mathbf{x}, \mathbf{w}), \quad (1)$$

where $\mathbf{x} = \{x_1, x_2, \dots, x_{N_x}\}$,
 $\mathbf{w} = \{w_1, w_2, \dots, w_{N_x}\}$,
 f - Boolean transfer function, $x_i, w_i, f, y \in \{0, 1\}$.

The change of the real domain of neurons to Boolean values and to a Boolean transfer function reduces the time for converting the input signal vector into the output signal significantly. The second advantage of the Boolean neuron is the decrease of necessary memory size.

The structure of such a Boolean neuron is unchanged comparing to classical neurons. The general computational process and the structure of the neural network are unchanged, too. Since the structure of the Boolean neuron of 4 inputs has same logic structure as a lookup table (see Fig. 1), the restriction of a Boolean neuron to 4-inputs allows a direct representation of such BN to a single LUT.

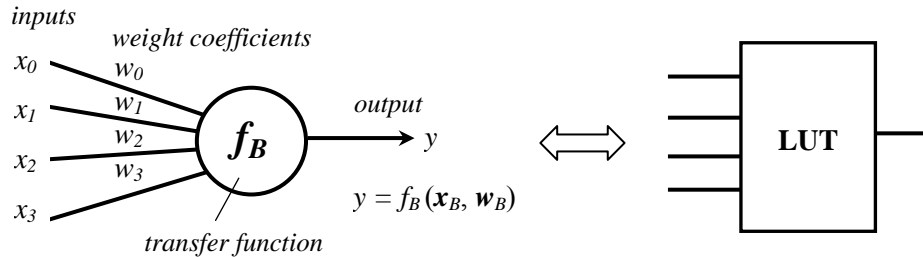


Fig. 1. General structures of BN and LUT

The following BNN with restricted Boolean neurons can be mapped directly to the FPGA structure. In this case the network of LUTs of an FPGA fits well to a Boolean neural network (see Fig. 2).

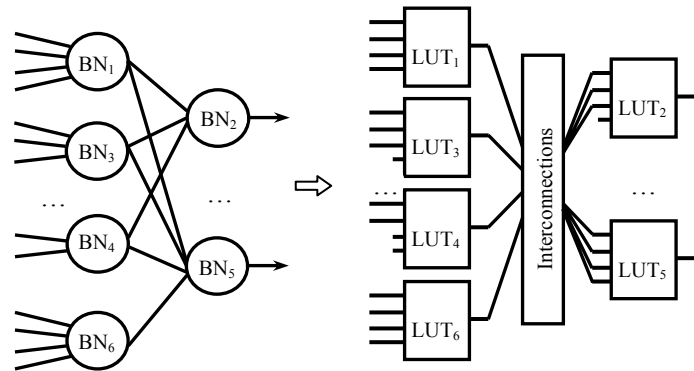


Fig. 2. Mapping of BNN to FPGA

2. Boolean neural networks

2.1 Training

Replacing classical neural elements by Boolean neurons we created a new type of neural network, called Boolean neural network (BNN). Boolean neural networks belong to class of feed forward neural networks. For the training of BNN a sequential training algorithm was developed. This training algorithm adds hidden layers and hidden neurons during the training process and each added neuron has not more than 4 inputs. The Boolean neurons on the hidden layer have different transfer functions. Each Boolean neuron on the hidden layer possesses a Boolean transfer function that is defined on the whole set or subset of input variables and is different from the Boolean transfer function of all other neurons in this layer. These Boolean functions are determined in the training process.

The starting point of the training process is the function table of the set of Boolean functions. This table is also called implementation matrix that covers the input and output signals as complete training set of the neural network [10, 11]. The main idea of the suggested training consists in the sequential decomposition of all Boolean functions of the function set. Each Boolean function from the set will be represented by simpler Boolean functions unknown

initially. The result of training should be a completely trained Boolean neural network having defined parameters for all neurons. Before the training is started, the base operation for this training should be selected. It can be any Boolean operation such as “AND”, “OR”, “equivalence” or “EXOR”. The training process assumes a given Boolean base operation as a transfer function for all neurons of the output layer.

The algorithm starts with an insertion of the input layer into the network, see Alg. 1 - Line 1. The number of neurons in the input layer is equal to the number of all arguments of the set of Boolean functions. Then the hidden layer is added iteratively. The index of the first hidden neuron is 0. Each function from the initial set is selected and returned as result of the function `SELECTFROMYSET()`, Alg. 1 - Line 4. While the selected Boolean function is not decomposed completely, Alg. 1 – line 5, the next steps will be repeated in a loop, Alg. 1 – lines 6-10. The condition in the loop depends on the selected base operation and will be tested by the function `ISNOTDECOMPOSED()`.

Algorithm 1 Train – Training of BNN

Input: $Yset$ – set of Boolean functions
 $limit$ – largest number of inputs allowed for neurons

Result: $net = \{Kset, W\}$ – network

```

TRAIN( $Yset, limit$ )
1  insert the input layer
   //Start the training of hidden layer
2   $i \leftarrow 0$ 
3  for ( $func \leftarrow 0, \dots, Ny-1$ ) //  $Ny$  – amount of functions in the initial set
4      $bf \leftarrow SELECTFROMYSET(func)$ 
5     while ISNOTDECOMPOSED( $bf$ ) do
6         ( $k[i], flag$ )  $\leftarrow FINDK(bf, limit)$ 
7          $w[i] \leftarrow REDUCTIONOFYSET(k[i])$ 
8          $Kset \leftarrow ADDTOSET(k[i])$ 
9          $W \leftarrow ADDTOSET(w[i])$ 
10         $i \leftarrow i + 1$  // prepare the insertion of a new hidden neuron
   //split the output neurons
11  $W \leftarrow SUPERPOS(W, limit)$ 
12 return  $net$ 

```

If a further decomposition is necessary, a new hidden neuron is added, an appropriate transfer function $k[i]$ is searched by the function `FINDK()` (Alg. 1 – line 6) and, if possible it will be separated from all functions in the function set (Alg. 1 – line 7). Thereby a weight vector $w[i]$ will be received. The functions `FINDK()` and `REDUCTIONOFYSET()` (Alg. 1 – lines 6, 7) are described in detail in the algorithms 2 and 3. Each found transfer function and the new vector of weight coefficients are added to the appropriate set and matrix (Alg. 1 – lines 8, 9). If all Boolean functions were decomposed, i.e. the sets of k -functions and weights coefficients were found, the algorithm of a superposition `SUPERPOS()` for the output layer is used (Alg. 1 – line 11). The aim of this step is to restrict the number of inputs in the output neurons to the given limit. It allows a direct mapping of the BNN into a FPGA structure.

In the algorithm 1 – line 6 the function `FINDK()` is called to define the transfer function for the new inserted hidden neuron. The algorithm 2 describes this function in details. The function `FINDK()` gets the Boolean function bf and the permitted number of inputs of the hidden neuron as input parameters. The value $limit$ is the desired number of arguments for the searched transfer function of this neuron. The searched function should be a subfunction of the specified function bf defined on $limit$ Boolean variables, if possible. The algorithm returns the Boolean transfer function of the added hidden neuron k and the flag $flag$. If the found transfer function depends on $limit$ Boolean variables, the $flag = 1$, otherwise $flag = 0$.

The algorithm FINDK works as follows. First, a difference *diff* between the amount of variables in the function *bf* and the permitted number of variables in the function *k* is computed. The variables *flag* and *i* are initialized, Alg. 2 – line 1. Then, while *flag* = 0 and *i* < N_x , a subfunction of the given function *bf* is searched in the **while** loop (Alg. 2 – lines 2-6). The searched subfunction should be depending on N_x-1 Boolean variables.

Algorithm 2 FindK – Search the transfer function for the new hidden neuron	
Input:	<i>bf</i> – Boolean function <i>limit</i> – permitted number of inputs in the hidden neuron
Result:	<i>k</i> – found transfer function $flag \in \{1, 0\}$

```

FINDK(bf, limit)
1  diff ←  $N_x - limit$ ; flag ← 0; i ← 0 //  $N_x$  – number of variables in bf
2  while (flag = 0 and i <  $N_x$ ) do
3      if (  $\min_{x_i} bf \neq 0$ ) then
4           $k \leftarrow \min_{x_i} bf$ 
5          flag ← 1
6          i ← i + 1
7      if (flag = 1 and diff > 1) then
8          (k2, flag2) ← FINDK (k2, limit) // k2 – half table of the k
9          if (flag2 = 1) then k ← chain (k2, k2) // – chain k2 by itself
10     if (flag=1) then return (k, flag)
11     else return (bf, flag)

```

While such a subfunction not found (*flag* = 0) and the parameter *i* < N_x , the partial minimum of the function *bf* with regard to x_i is computed. If the minimum is not a constant function 0, Alg. 2 - line 3, the subfunction *k* of the function *bf* will be found in line 4 - Alg. 2, and the flag *flag* is set to 1, Alg. 2 – line 5. If the difference *diff* > 1, the function FINDK() will be called recursively for a half function $k_{/2}$ of the function *k* created in the **while** loop, Alg. 2 –lines 7, 8. The half function $k_{/2}$ is selected by one half of the truth table of the subfunction *k*. Note, that half function $k_{/2}$ depends on N_x-1 Boolean variables, since it consists of one half of the truth table of the subfunction *k*.

The function FINDK() in line 8 returns the function *k2* and *flag2*. If *flag2* is equal to 1, the function *k* will be defined by chaining of the truth table of the function *k2*, Alg. 2 – line 9. Depending on the value of *flag* either the founded subfunction *k* or the function source function *bf* will be returned together with the flag, Alg. 2 – lines 10-11.

Algorithm 2 describes the function FINDK() for OR- and/or EXOR as base operation. These base operations affect the lines 3-4 of the algorithm, where the partial minimum of the function *bf* is computed. For the AND and/or EXAND as base operation the algorithm keeps unchanged, generally. Only instead of the partial minimum the partial maximum of the function *bf* is computed in the lines 3-4 and compared with 1 in line 3.

Since the function FINDK() was explained for OR- and/or EXOR as base operations above, algorithm 3 states the function REDUCTIONOFYSET() for the same operations. It is checked in the function REDUCTIONOFYSET() whether a given Boolean function *k* can be separated from each Boolean function of the initial set. Additionally, the vector of weight coefficients for the output neurons is determined.

The algorithm 3 executes calculations for each Boolean function from the initial set in a loop, Alg. 3 – line 1. At first, the weight coefficient for the selected function is set to 1, Alg. 3 – line 2. If there is at least one combination of a zero value of the selected function and a one value of the function *k*, Alg. 3 – line 4, than the corresponding weight coefficient is set

to 0, Alg. 3 – line 5. Otherwise, Alg. 3 – line 6, the values of the selected function are set to 0 for each one value of the k function, Alg. 3 – lines 8, 9. As result the vector of weight coefficients w is returned, Alg. 3 – line 10.

Algorithm 3. Reduction of $Yset$ – split the function k from all functions in the set $Yset$

Input: k – transfer function of a neuron
Result: w – vector of weight coefficients for output neurons

REDUCTIONOFYSET(k)

```

1  for ( $func \leftarrow 0, \dots, Ny-1$ )           //  $Ny$  – number of functions in the set
2       $i \leftarrow func; w[i] \leftarrow 1$ 
3      for ( $j \leftarrow 0, \dots, Np-1$ )
4          if ( $k[j] = 1$  and  $func[j] = 0$ ) then
5               $w[i] \leftarrow 0$ 
6      if ( $w[i] = 1$ ) then
7          for ( $j \leftarrow 0, \dots, Np-1$ )
8              if ( $k[j] = 1$ ) then
9                   $func[j] \leftarrow 0$ 
10 return  $w$ 

```

In this paper we do not consider the algorithm for the function SUPERPOS() in detail. This algorithm based on the law of superposition for Boolean data. A possible implementation of this function is described in [10].

The result of the training of a BNN is a network of neurons having defined transfer functions and weight coefficients for all neurons on the hidden and output layers.

In order to map the trained BNN to FPGA automatically, the MOCCA compiler [3, 8] is used. For that we have to describe the network as an UML (Unified Modeling Language) model. Fig. 6 illustrates a respective system-level design of a BNN using UML.

2.2 Example 1

Assume a LUT has 2 inputs. The set of three very simple Boolean functions y_1, y_2 and y_3 (2) are given as input data for the training algorithm described above. The operations used in (2) are defined as $a \oplus b = \overline{a}b \vee a\overline{b}$ and $a \odot b = ab \vee \overline{a}\overline{b}$. We take the functions y_1, y_2 and y_3 , because they have small function tables that can be represented in this paper completely.

$$y_0 = \overline{x_2}x_3 \vee x_1x_2 \oplus x_3; \quad y_1 = \overline{x_2} \vee (x_1 \oplus x_3); \quad y_2 = (\overline{x_1} \vee \overline{x_2}) \odot x_3 \quad (2)$$

Since the given functions depend on three Boolean variables x_1, x_2 and x_3 , according to the algorithm 1 first the input layer with 3 neurons is inserted in the network. The number of hidden neurons is set to 0.

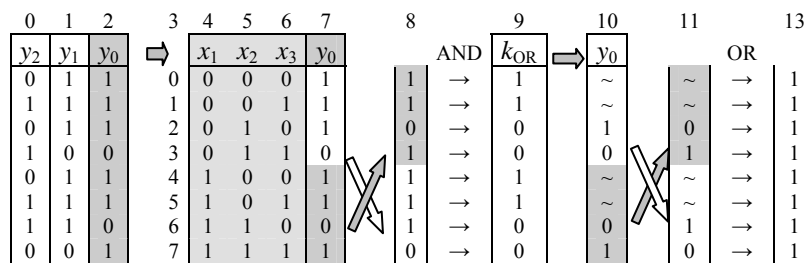


Fig. 3. Search of a transfer function k for the first hidden neuron

For the development of the hidden layer of the network the function y_0 is selected from the functions set. There are values one in the truth table of this function, Fig. 3 – column 7, therefore the training continues. Next, the truth table of the function y_0 is divided into two

halves and these halves are exchanged, Fig. 3 - columns 7 and 8. Each value of the function y_0 from column 7 is assigned to the corresponding value of the column 8. The conjunction of columns 7 and 8 computes the partial minimum of function y_0 with regard to x_1 . Since not all values in the column 9 - Fig. 3 are equal to 0, the computed minimum is a transfer function for the first hidden neuron. The training algorithm can be improved if FINDK() functions for different operations (e.g. OR and AND) are applied. This is explained in our example.

	x_1	x_2	x_3	k_1	k_2	k_3	k_4	k_5
0	0	0	0	1	1	0	0	0
1	0	0	1	1	0	1	1	1
2	0	1	0	0	1	0	0	1
3	0	1	1	0	0	1	0	1
4	1	0	0	1	0	1	0	0
5	1	0	1	1	1	0	1	1
6	1	1	0	0	0	1	0	1
7	1	1	1	0	1	0	0	1

	k_1	k_2	k_3	k_4	k_5
y_2	0	0	1	1	1
y_1	1	0	1	0	0
y_0	1	1	0	0	0

	y_2	y_1	y_0
0	1	1	1
1	1	1	1
2	0	0	1
3	1	1	0
4	0	1	1
5	1	1	1
6	1	1	0
7	0	0	1

Fig. 4. The transfer functions k_1, \dots, k_5 of hidden neurons, the weight coefficients of output neurons and initial functions y_0, y_1 and y_2

The function REDUCTIONOFYSET() is called after the first hidden neuron gets its transfer function. The truth table of the function shown in the column 10 – Fig. 3 is the result of separating k_{OR} from the function y_0 . It means, each one value of the function y_0 is set on ‘~’, if the correspond value of the function k_{OR} is equal 1. In such a way a lattice of functions is received, Fig. 3 - column 10. The “don't care” values can be used for both 0 and 1. Since the truth table of y_0 contains both one and zero values in column 10, its truth table is divided again in two halves, Fig. 3 - column 11. But now, the partial maximum of the function y_0 with regard of x_1 is computed, Fig. 3 - column 13. The received partial maximum is not a transfer function of the next hidden neuron, because it contains no zero value. Then the partial minimum and maximum of the function y_0 with regard to another variable is computed, until the function y_0 is decomposed. After that the next function from the initial set will be selected and decomposed. The created transfer functions of hidden neurons and the vectors of weight coefficients computed by the function REDUCTIONOFYSET() are added in the corresponding sets.

Having all functions in the initial function set decomposed, the hidden layer of the BNN is built. The transfer functions of hidden neurons (Fig. 4 left) and the weight coefficients of Boolean neurons in the output layer (Fig. 4 middle) were received. All transfer functions on the hidden layer depend on 2 input signals.

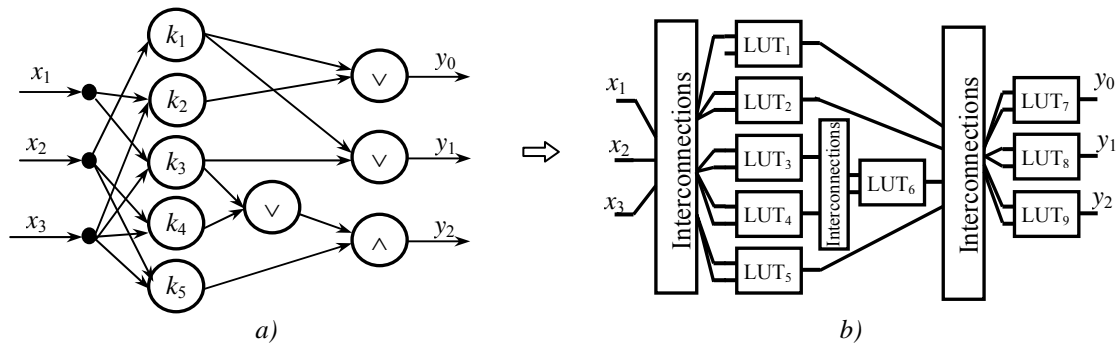


Fig. 5. Mapping of BNN to the network consisting of 2 inputs LUTs

One neuron in the output layer depends on 3 inputs, because three weight coefficients of this neuron are equal to 1 (see weight coefficients for the function y_2 in Fig. 4). Since all neurons are restricted to 2 inputs according to the settings of the task, this neuron must be divided into 2 neurons by a superposition. Now training is finished and the Boolean neural network is trained completely. The produced structure of the neural network is shown in Fig. 5a. Each

neuron in the hidden layer has its own transfer function, which differs from the transfer functions of all other hidden neurons. The 5 functions k_i are needed for the mapping the 3 initial Boolean functions, but each of these functions k_i depends on 2 Boolean variables only. This allows a direct mapping of the transfer functions of Boolean neurons into FPGAs that have LUTs with 2 inputs only. The corresponding structure of a network of LUTs is shown in Fig. 5b.

2.3 Example 2

The suggested training algorithm for the optimal mapping of Boolean neural network was verified on the some examples from MCNC benchmarks. The results of the OR- and AND-training algorithms for some MCNC benchmarks are shown in Table 1.

Table 1
Results of training for MCNC benchmarks using LUTs having 4 inputs

used benchmark	number of		numbers of hidden neurons created by	
	inputs	outputs	OR-algorithm	AND-algorithm
5xp1	7	10	46	51
9sym1	9	1	20	36
bw	5	9	25	25
b12	15	8	49	27
con1	7	2	12	10
alcom	15	38	40	74

3. Mapping of BNN to FPGA

3.1 Approach

There are many methods for the implementation of BNN into FPGAs. The system to be realized in FPGA should be programmed. For this some hardware specification language, a graphic input of a circuit diagram or a finite state machine can be used. The hardware specification languages such VHDL or Verilog are commonly used in a practice. Using such the descriptions, it follows as next steps the functional simulation, the synthesis and the implementation. After that the FPGA can be configured using the produced bit stream. Although there are progresses in high level synthesis technology, a direct specification of BNN for a FPGA implementation using VHDL is expensive. An efficient methodology for the realization of systems in FPGA was suggested in [8, 12]. This methodology connects the paradigm of the Hardware/Software CoDesigns with the concept of system development in a model driven architecture (MDA Model Driven Architecture) [5, 13 and 14]. For the description of the BNN the UML 2.0 is used. The system specified by UML can be mapped automatically in an executable application by a special tool, called the MOCCA compiler (MOdel Compiler for reConfigurable Architectures) [14, 15]. Next we explain an example of mapping of BNN in FPGA using the UML and the MOCCA compiler.

3.2 Example

As an example of the Hardware/Software CoDesigns a very simple trained Boolean neural network is used. The transfer functions of hidden neurons and the corresponding connections of functions k that are presented by the weight coefficients are shown in Fig. 6a, and 6b. The trained BNN represents a set of Boolean functions y_0, y_1, \dots, y_9 . Each of them is defined on 3 variables x_1, x_2 and x_3 . The corresponding UML Design model of this BNN is presented in Fig. 6c.

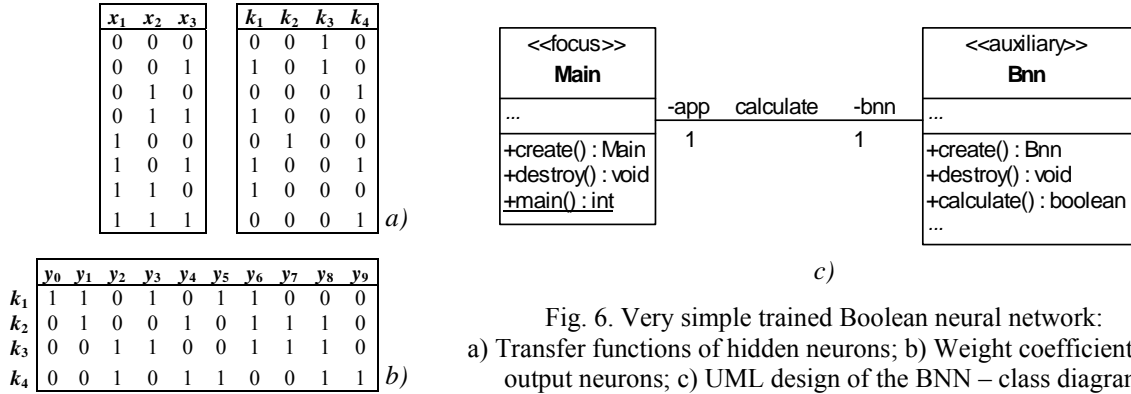


Fig. 6. Very simple trained Boolean neural network:
 a) Transfer functions of hidden neurons; b) Weight coefficients for output neurons; c) UML design of the BNN – class diagram

The system consists of the two classes **Main** and **Bnn**. The trained BNN is specified by the method *calculate()* of the class **Bnn**. The object of the class **Main** instantiates the object of the class **Bnn**, defines the input signals and receives the result signals. The class **Bnn** has for each transfer function of neurons on the hidden and output layers of Boolean neural network a method, which should be mapped each into a single LUT. The complete BNN will be evaluated in the FPGA by the invocation *calculate()* of the *main()* method.

In order to map design model of BNN automatically, the MOCCA compiler using C++ and VHDL-RTL implementations platforms was applied. The overall compilation/synthesis time of the design model into the final hardware/software modules takes approximately 0.5 to 5 minutes, depending on the degree of optimization. The Boolean neural network was tested on a hardware platform comprising a Pentium IV processor running at 2.4 GHz and a Xilinx Virtex-II FPGA [16] with approximately 3 million gate equivalents running at 100 MHz. The majority of the resources is consumed by the *calculate()* method, which represents the full structure of the BNN after inlining all methods representing functions *k* and *y*, respectively. The FPGA implementation of the BNN, including the communication interfaces and register file requires 4 Flip Flops to store the input and output data, and the FSM states. For its realization 15 LUTs are used. The execution time of *calculate()* takes 50 ns.

Conclusion

FPGA-based reconfigurable architectures are well suited to implement artificial neural networks. However the direct mapping of classical neural networks into FPGAs structures needs a large number of logic gates for each neuron. Therefore in this paper a new type of neural elements was suggested. It is called Boolean neuron and can be realized by a single lookup table (LUT) of configurable logic blocks (CLB). A logic block of an FPGA includes 4 LUTs. Thus, a CLB can realize 4 Boolean neurons controlled by 4 inputs each. The Boolean neurons allow an optimal FPGA implementation of neural networks.

We have developed algorithms for the training of Boolean neural network that restrict the number of inputs of Boolean neurons. Due to the limitation on 4 inputs of Boolean neurons in Boolean neural networks, efficient FPGA implementations of BNNs, in terms of performance and gate count was found. In comparison to classical neural networks the total number of neurons in the Boolean neural network may be increased, but because only one LUT is required for Boolean neuron the common effort is decreased. That is huge benefit compared to existing approaches of neural network implementations on FPGAs. The suggested Boolean neuron extends the possible approaches of neural network implementation in circuit design significantly.

The presented Boolean neural network is one of the possible types of neural networks that based on Boolean neurons. The Boolean neuron can be used for the synthesis of various structures of neural network. Using such Boolean neurons we could decrease the calculation time for both training and utilization of BNNs significantly. An additional advantage of the Boolean neuron consists in the reduction of necessary memory size. The example of the UML-based automated hardware-software co-design for run-time reconfigurable architectures showed an efficient method for mapping of Boolean neural network into a discrete device.

In future, we continue our research for optimal mapping of Boolean neural network to FPGA devices. Based on the results described in this paper we will design and develop a mapping methodology for any structures of Boolean neural networks both without on-chip learning and with on-chip learning.

References

1. Caviglia D. D.: The Italian Managing Node of NeuroNet. // available on Web Page - URL: <http://www.dibe.unige.it/neuronet/>
2. Cox Ch.E.; Blanz W.E.: GANGLION – a fast field-programmable gate array implementation of a connectionist classifier. // IEEE Journal of Solid-State Circuits, 27(3), March 1992, pp. 288 – 299.
3. Bade S.L.; Hutchings R.L.: FPGA-based stochastic neural networks-implementation. // Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1994, pp. 189 – 198.
4. Eldredge J.G.; Hutchings B.L.: RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. // Proceeding of the IEEE World Conference on Computational Intelligence, 1994, pp 77 – 80.
5. Gajski, D.; Vahid, F.: Specification and Design of Embedded Hardware-Software Systems.// IEEE Design and Test of Computers, 1995, pp. 53 – 66.
6. Kohut, R.; Steinbach, B.: The Structure of Boolean Neuron for the Optimal Mapping to FPGAs. Proceedings of the VIII-th International Conference CADSM 2005. Lviv – Polyana, Ukraine, 2005, pp. 469 – 473.
7. Gschwind M.; Salapura V.; Maischberger O.: Space efficient neural net implementation. // Proceedings of the Second International ACM/SIGDA Workshop on Field-Programmable Gate Arrays, Berkeley, CA, USA, February 1994.
8. Steinbach, B., Beierlein, T., Fröhlich, D. UML-Based Co-Design for Run-Time Reconfigurable Architectures. // Grimm, Ch.: Languages for System Specification, Kluwer Academic Publishers, Boston, Dordrecht, London, Printed in The Netherlands, 2004, pp. 5 – 19.
9. Gschwind M.; Salapura V.; Maischberger O.: A Generic Building Block for Hopfield Neural Networks with On-Chip Learning. // IEEE International Symposium on Circuits and Systems, Atlanta, USA, May 1996.
10. Kohut, R.: Neuronale Netze als Modell Boolescher Funktionen, Doctor thesis in German, Freiberg University of Mining and Technology, Freiberg, 2006.
11. Kohut, R.; Steinbach, B.: Boolean Neural Networks. Transactions on Systems, Issue 2, Volume 3, April 2004, pp. 420 – 425.
12. Steinbach, B., Fröhlich, D., Beierlein, T.: Hardware/Software Codesign of Reconfigurable Architectures Using UML. // Grant, M.; Müller, W.: UML for SOC Design, Springer, Dordrecht, Printed in The Netherlands, 2005, pp. 89 – 117.
13. Gajski, D.D.: Principles of Digital Design. Prentice Hall, Munich, London, Mexico, New York, Singapore, Sydney, Toronto, 1997.
14. Object Management Group - Architecture Board ORMSC (2001). Model driven architecture - a technical perspective (MDA). // Web Page <http://www.omg.org>, Web-Pages, 2004.
15. MOCCA Project. The MOCCA-compiler for run-time reconfigurable architectures. <http://www.htwm.de/lec/mocca>, Web-Pages, Aug. 2006.
16. Xilinx. The Programmable Logic Data Book. Xilinx, Inc., San Jose, CA, USA, 1993.