48

Data · June 2014

CITATIONS
0

READS
564

3 authors, including:

Hanen Chehimi
National Engineering School of Monastir
8 PUBLICATIONS **13** CITATIONS
SEE PROFILE

M'Sahli Faouzi
University of Monastir
257 PUBLICATIONS **1,510** CITATIONS
SEE PROFILE

Optimized FPGA Implementation of an Artificial Neural Network for Function Approximation

Rabii Elmissaoui, Anis Sakly, Faouzi M'Sahli

Research Unit on Study of Industrial Systems and Renewable Energy,
National Engineering School of Monastir, Tunisia

ABSTRACT
This paper describes a generalized technique for designing, simulating and implementing an Artificial Neural Network (ANN) using Field Programmable Gate Array (FPGA). For a hardware implementation of the network, FPGA can exploit the parallelism potential of neural networks. By implementing all the concurrent tasks that can be executed simultaneously, the processing time can be reduced to a significant degree. Network modeling is developed from the model of a single neuron whose effectiveness of implementation plays an important role on the performance of the network, especially in terms of number of response cycles. This work also includes a description of the method used to integrate a hyperbolic tangent sigmoid (Tansig) activation function and its influence on FPGA memory consumption which we try to optimize. Two examples of function approximation are implemented to validate the different results found in practice and simulation.

Key words: Artificial neural network, FPGA, Training, Tansig activation function, BRAM, Backpropagation algorithm, VHDL, Identification.

Corresponding Author: Rabii Elmissaoui elmissaoui.rabii@gmail.com

1 INTRODUCTION

Applications of neural networks cover a wide range of problems involving complex nonlinear relationships where traditional solutions are not valid. The reason for this diversity is justified through the benefits of this type of architecture. Indeed, neural networks offer simplicity of implementation and good adaptation with the parametric and environment variations. In addition, they have a parallel processing structure and an ability to learn through examples [1]. ANNs can be implemented easily in software applications since this method does not require detailed knowledge about the inner workings of the various network elements. However, a disadvantage in real-time applications of software-based ANNs is slower execution compared with other hardware-based ANNs [2]. FPGAs are reconfigurable and programmable devices that give a great flexibility in the development of many electronic circuits which make them suitable for hardware implementation of neural networks [3]. One of the important challenges in this work is to design a neuron model, using serial processing input mode, which is capable to offer a high speed response performance without deteriorating the memory usage. The results of the synthesis are compared to other results in related works. The comparison is done in terms of time and memory with the same serial processing mode [3], partial parallel processing [4, 5] and full parallel processing [5].

This paper is organized as follows. In Section 2, we introduce the artificial model of the neuron describing the principal function of each part, the coding method to represent the

different inputs/outputs and the chosen technique to implement the Tansig activation function. This model is then used to create a simple artificial neural architecture with a single hidden layer. In this section, we also presents the algorithm and the tool used to train the ANN. Section 3 describes the simulation of a single neuron to demonstrate how the internal data are processed to generate the final output. Then, using the same tool, we simulates the hole ANN applied to learn two different examples with two different structures in the first example (XOR bipolar function) and a complementary test in the second example (dynamic system). Section 4 deals with synthesizing a single neuron and comparing the different results with other related work. After that, it deals with synthesizing and describing the implementation of the network. The neural architecture proposed in this study is implemented in Xilinx XC3S200 Spartan3 FPGA using Very high speed integrated circuit Hardware Description Language (VHDL). Finally, in Section 5, we perform a comparison between the different obtained results in terms of training speed and output precision.

2 MODELING OF THE ANN

2.1 Neuron Model

The artificial neuron is a computational model inspired from the operation of the biological neuron. An ANN consists of a set of artificial neurons. The general model of an artificial neuron [6] is shown in Fig 1.

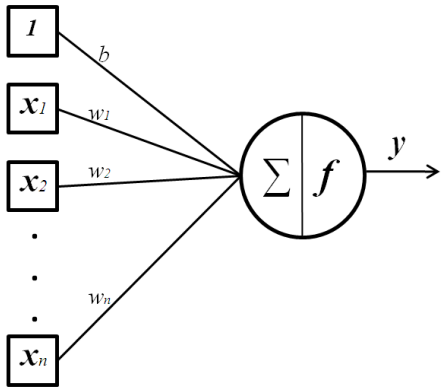

Fig 1: General model of the artificial neuron

f is the activation function given by:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) = f(z) \quad (1)$$

The VHDL model of the artificial neuron used in this work is made using components synchronized with a single clock (Fig 2). The signals are coded with signed fixed-point coding. Each signal contains two parts. The first one is the Integer Part (IP) including the sign bit and the second is the Fractional Part (FP). Compared to floating-point coding, fixed-point offers the advantage of simplicity in implementation since it reduces the computation time and memory consumption. There are a lot of libraries and documentations on VHDL coding rules for handling fixed-point. For example, the library IEEE_proposed [7] is quite useful in the simulation to make conversions between many different types like *real* (real value), *std_logic_vector* (normal binary vector) and also *sfixed* (fixed-point binary vector). Besides,

the operations of addition and multiplication are similar to the simple binary coding operations. But, it is necessary to have an attention to some rules for the point position of the generated output to not have wrong values. Some of these rules are noted in Table 1.

Table 1. Some fixed-point rules

Operation	Result Range	
	IP	FP
A+B	Max(A'IP, B'IP) + 1	Min(A'FP, B'FP) + 1
A-B	Max(A'IP, B'IP) + 1	Min(A'FP, B'FP) + 1
AxB	A'IP + B'IP + 1	A'FP + B'FP

The proposed neuron contains four components (Fig 2). Each one has a logic clock input to synchronize it with all other components, a chip select input to be activated by the previous component and a chip select output to activate the next component.

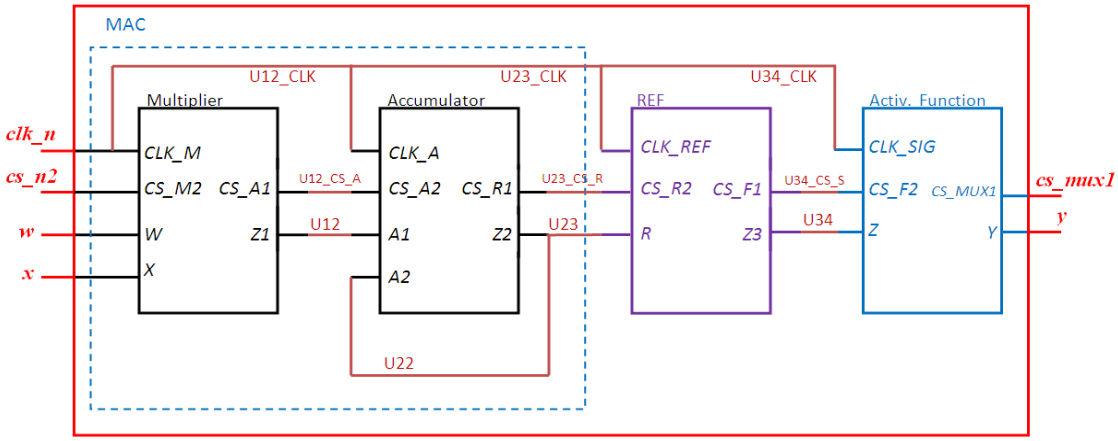

Fig 2: VHDL model of the artificial neuron

2.1.1 Multiplier-Accumulator

The Multiplier-Accumulator (MAC) is the processing stage that can be designed using three different techniques like shown in Fig 3. The first technique is the Serial Processing (SP). The multiplication and addition are both established in serial mode. It uses a single multiplier and a single accumulator. It consists of multiplying one input by its equivalent weight and the result is then added to the previous accumulated one. The second technique is the Partial Parallel Processing (PPP) which uses several multipliers to perform parallel multiplication. The outputs of each two multipliers are connected to the inputs of an accumulator. Then, the outputs of each two accumulators are also injected into the inputs of another accumulator. The process is repeated until reaching one final output. The third technique is the Full Parallel Processing (FPP) in which the multiplication is performed in parallel mode like in PPP. The process of accumulation is also performed in parallel mode using a single multi-input accumulator. The first technique is the one used in this work because, generally, it consumes less memory than the other two ones. The only constraint is in the response time. It is adequate for a small input number but the more the number of input increases the more the response time increases. So in this work, we try to minimize this time by reducing the number of response cycles.

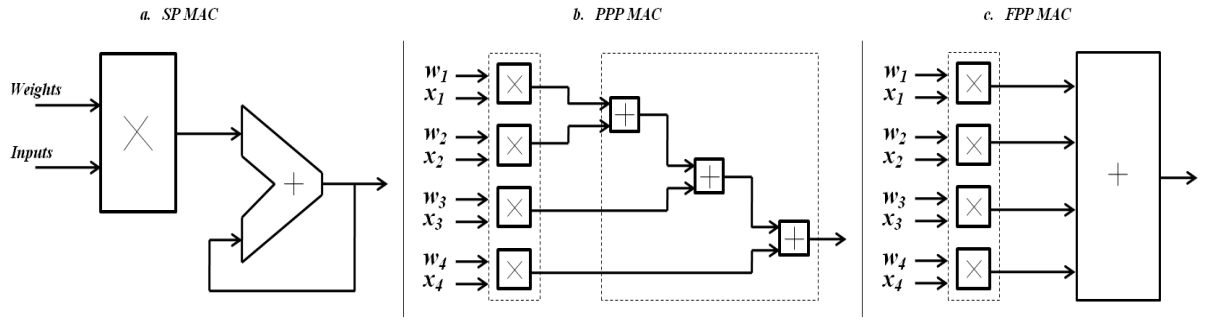

Fig 3: Techniques of MAC design

The multiplier uses a serial mode for the acquisition of the pair (*w, x*) then multiplies them. The result is transmitted to the output pin *Z1*. It uses the embedded multiplier MULT 18X18 of XC3S200 Spartan3 FPGA. The weight *w* is coded in 10 bits with 7 bits in FP. This coding is for signed values in the range [-4,4] with a step of 0.01. The choice of this interval depends mainly on the studied example and generated weights after training. In this work, we consider two different versions for coding the input signal *x*. The first version is used specially for the neurons of the first hidden layer. It consists of using 16 bits with 1 bit for the sign, 4 bits in IP and 11 bits in FP. This coding is necessary in order to take into account the input values that are outside the interval [-1,1] (Tansig function output range). The second coding version uses 16 bits with 14 bits in FP, largely sufficient for values in the interval [-1,1] with a step of 0.0001. Both of them work with the same principal and use the same resources in memory and the same number of response cycles. In the rest of the paper, we describe the work based on the first coding version.

The accumulator performs the addition of the weighted sum of various outputs already calculated by the multiplier. It starts operations at the falling edge just after the multiplier output. It has an internal counter that counts the number of inputs. Each time it adds the final amount calculated with the new value in *A1* coded in 26 bits.

2.1.2 Reconstitution of the Form

The Reconstitution of the Form (REF) is a block that has for mission to adapt the form of the post-synaptic signal. The output of the MAC unit (Multiplier-Accumulator) requires a 28 bits coding and more specifically 9 bits in IP to satisfy the conditions of fixed-point math operations. But in practice, two bits in the IP are enough to reach the interval [-4,4] chosen for the activation function input. In addition, 7 bits in the FP (instead of 18 bits) are sufficient to code 2 digits after the point. Hence, the role of this block becomes more clear; it makes a truncation (without changing the value) of MAC fixed-point output to adapt it for the activation function input.

2.1.3 Activation Function

This block is the final sub-component of the neuron which is the global component. Hidden layers use Tansig function. Two types of activation function are used in this study:

- Purelin Function: $y = z$ (2)
- Tansig Function: $y = (1 - exp(-2z)) / (1 + exp(2z))$ (3)

The Purelin function is mainly used in the last layer in order to extend the ANN output beyond the interval [-1,1]. Its output is coded in 16 bits with 4 bits in IP. However, there is no

need to use the REF bloc with a neuron model using Purelin as an activation function. The structure using Tansig function in the hidden layer and a Purelin function in the output layer is capable to approximate any mathematical function.

The Tansig input is coded in 11 bits. The output is coded in 16 bits like the input signal *x* of the multiplier, in order to take into account its injection in the input of another neuron in the network. This component uses a look-up table of values stored in memory. For example, for 256 values, the step is 4/128 = 0.03125 to browse the interval [-4,4]. This table can be stored using two different techniques. The first technique stores values directly in the SRAM distributed memory (the LUTs) of Xilinx XC3S200 Spartan3 FPGA. But, it consumes a lot of memory. The second technique stores the values in Block-RAM (BRAM). Xilinx XC3S200 Spartan3 FPGA contains 12 blocks of 18 Kbits RAM (216 Kbits). It calls the VHDL model of this block with the Xilinx synthesis tool (ISE Webpack) and inserts it into the component of the activation function as if it were a sub-component. This technique saves the distributed memory for the rest of the artificial network architecture.

Xilinx offers single-port and dual-port BRAMs. Like shown in Fig 4, in this work we use the single-port model. Each memory port is synchronized with its own clock (CLK) and has a memory enable bit (EN), write enable bit (WE), input data signal (DI), output data signal (DO) and a signal containing the address of the data to write/read (ADDR). Depending on the size of values to be stored and read into memory, there are several configurations of a BRAM memory. In our case, we want to read 16 bits values. This means that the BRAM acts in ROM mode. Then, it is preferable to read 16 bits at one time using the 16Kx1 configuration.

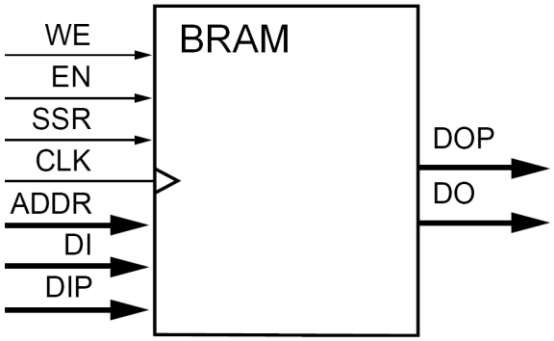

Fig 4: Xilinx single-port BRAM model

The name of the single-port BRAM used in this work is RAMB16_S18. This memory is divided into four blocks of 16 lines. Each line contains 16 values. Each real value is coded with 4 hexadecimal values. Thus, the total memory is 16x16x4 = 1024 of 16 bits values. In the previous code, only 64 values are filled. The real values approximating the Tansig function must be converted into hexadecimal and inserted correctly in BRAM.

2.2 ANN Model

The implemented neural architecture (Fig 5) is a simple two-layer network with two neurons in the first layer (hidden layer) and a single neuron in the second layer (output layer). The network also has two inputs x_1 and x_2 (without considering the bias input). In this study, two main structures (f_1/f_2) are developed: Tansig/Tansig and Tansig/Purelin. The goal is to establish a comparative study in Training speed and output precision.

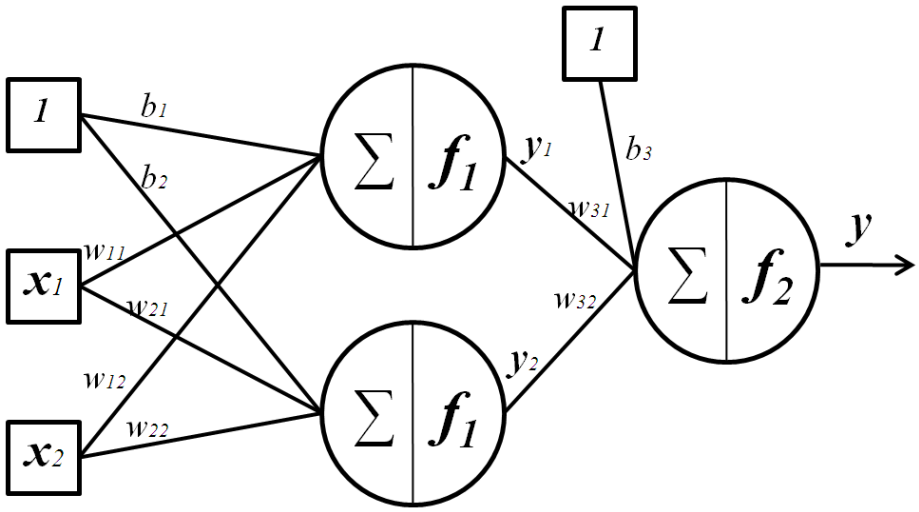

Fig 5: Neural architecture model

The VHDL model of the ANN is shown in Fig 6. This network has two inputs x_1 and x_2 and a single output y. The connection between the two neurons N1 and N2 of the first layer neuron with the neuron N3 of the second layer is provided via a multiplexer [2]. The input x_i is generated from a unit of reception and transmission that communicates by providing the inputs (switches) and transferring of the outputs (LEDs or seven segment display).

2.2.1 Unit of Reception and Transmission

This Unit of Reception and Transmission (URT) is divided into two components. The first component is called W_X_Gen which ensures the synchronization of weights transmission at specific moments in the network depending on the input x_i. The inputs are generated from switches (*rx_d0* to *rx_d6*). The eighth switch is *rx_d7* which control the display of output LEDs of Spartan3 FPGA board. The second component is named Y_Gen. It acquires the 16 bits output of the network. Then, it generates the first 8 bits or the second ones of the output, depending on the value transmitted to the pin TX_LH from *rx_d7*. It is also possible to configure the unit to display the output values with LEDs and/or seven segment display module of the Spartan3 FPGA Board (with SS_Display component).

2.2.2 Multiplexer

This component (Mux) is essential for the network. Indeed, it is not possible to send the two neuron outputs in the hidden layer directly to the single neuron input in the output layer. Indeed, in this work we are using an SP MAC version integrated in each neuron. Therefore, a multiplexer component is added to synchronize not only the transmission of the inputs of the third neuron N3 but also its weights. It has two inputs pins CS_MUX1 and CS_MUX2 to identify successively, the presence of neurons N1 and N2 outputs. Once the two pins go from '0' to '1' state simultaneously, the multiplexer starts sending inputs and weights to N3 as W_X_Gen with N1 and N2.

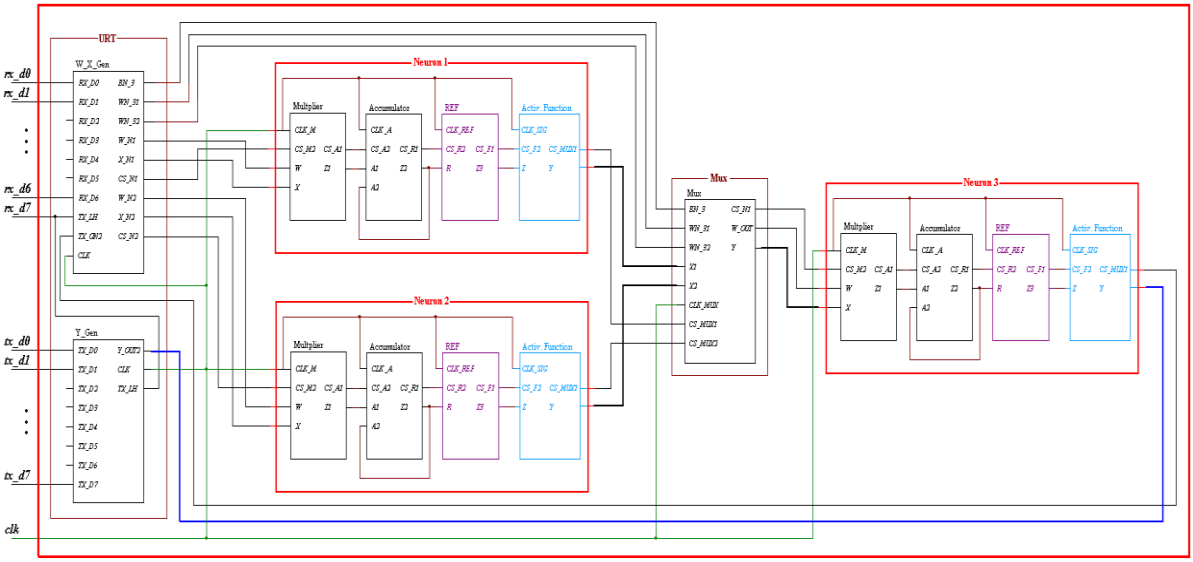

Fig 6: Model of the ANN with Tansig/Tansig structure

2.3 Training of the ANN

The training of the ANN is a supervised one in which we try to impose desired outputs to the network for a given inputs by modifying the synaptic weights values. The network then acts as a filter whose transfer settings are adjusted from the couples presented inputs/outputs. The adaptation parameters can be performed with an optimization algorithm in which, the initialization of synaptic weights are usually random. The training algorithm used is the error backpropagation algorithm which is a generalization of the "Delta" rule of Widrow-Hoff. This rule is also called the instantaneous gradient method and it is based on the minimization of the squared error criterion:

$$E(t) = \frac{1}{2}(y_d(t) - y(t))^2 \tag{4}$$

where $y_d(t)$ is the desired ANN output and $y(t)$ is its real output. The instantaneous gradient method adjusts the weights according to the derivative. It is defined by the following rule of weights updating [8]:

$$w_{ij}(t) = w_{ij}(t-1) - \alpha \frac{\partial E(t)}{\partial w_{ij}(t)} \tag{5}$$

The application of the instantaneous gradient method for multilayer networks is used to define the error backpropagation algorithm. For a layer q, we obtain:

$$w_{ij}^{(q)}(t) = w_{ij}^{(q)}(t-1) - \alpha \frac{\partial E(t)}{\partial w_{ij}^{(q)}(t)} \tag{6}$$

$$b_i^{(q)}(t) = b_i^{(q)}(t-1) - \alpha \frac{\partial E(t)}{\partial b_i^{(q)}(t)} \tag{7}$$

To simplify the training calculations, Neural Network Toolbox (NNToolbox) of Matlab is used. This tool provides a complete set of functions and a graphical user interface for the design, implementation, visualization and simulation of neural networks. It supports most commonly used architectures of supervised and unsupervised networks and a comprehensive set of training functions.

3 SIMULATION OF THE ANN

3.1 Single Neuron Simulation

The main simulation is performed with ModelSim software. The library unisim is necessary to simulate the neuron model using BRAM for the activation function. Fig 7 shows the simulation of the neuron using two random inputs and weights. The multiplier starts the first time by acquiring the bias b with its internal input $x_0 = 1$. Thus, it is actually an accumulation of three inputs before reaching the REF block providing the input to the function TANSIG.

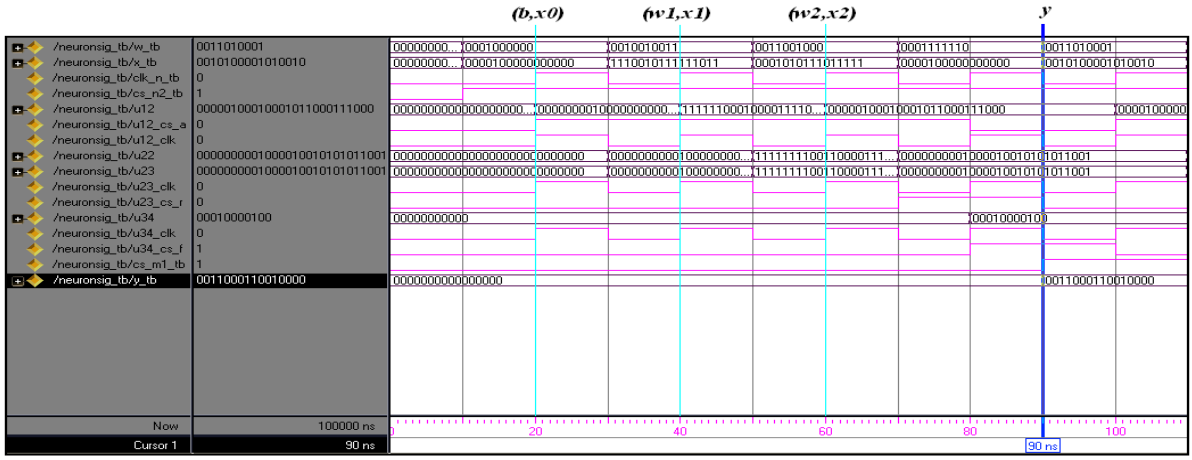

Fig 7: Simulation of the neuron

The clock period for this simulation is equal to 20 ns. The first acquisition starts at 20 ns and ends at 90 ns. Thus, for a single neuron, the final output is validated after 4.5 clock cycles from the first acquisition. Moreover, for a neuron with Purelin function, the number of cycles is only 4 because there is no use of REF bloc.

3.2 ANN Simulation

3.2.1 Example 1 (Bipolar XOR Function)

This chosen example is a very simple one used to test the ANN implementation and operation, whose truth table is given in Table 2.

Table 2. Bipolar XOR Function

Input x_1	Input x_2	Output y
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

The ANN simulation is also performed with the software ModelSim. The maximum number of values in the Tansig table is 1024. Whether we use distributed memory or BRAM to store them, we find the same results. The first simulation is performed with Tansig/Purelin structure (Fig 8). This architecture is able to generate the output of a single neuron after 3.5 clock cycles and the overall output of the ANN after 9.5 clock cycles. The real outputs are

remarkably similar to desired ones. In this case, the value 0.9916 and 0.9877 approximates 1 and -0.9970 approximates -1. Using Tansig/Tansig structure, the approximations of the output *y* are -0.9916 for -1 and 0.9885 for 1 (Fig 9). In both structures, the two input signals x_1 and x_2 of the ANN are transmitted using two signals (*rx_d0_tb* and *rx_d1_tb*) simulating the two switches (*rx_d0* and *rx_d1*) in the FPGA board. Another simulation is also performed with Matlab NNToolbox using the same mean squared error *e* (Table 7).

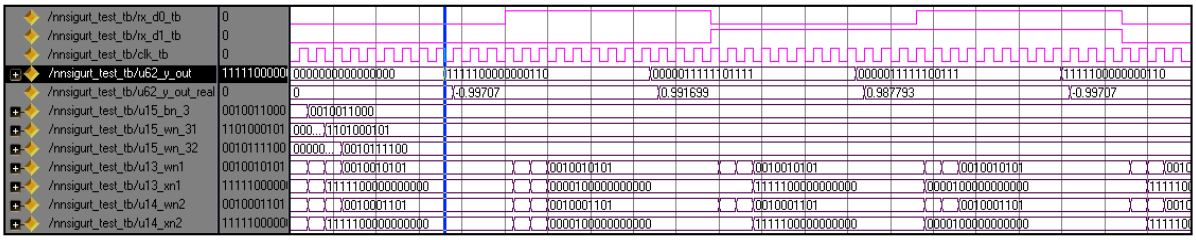

Fig 8: ANN simulation using Tansig/Purelin structure with bipolar XOR function

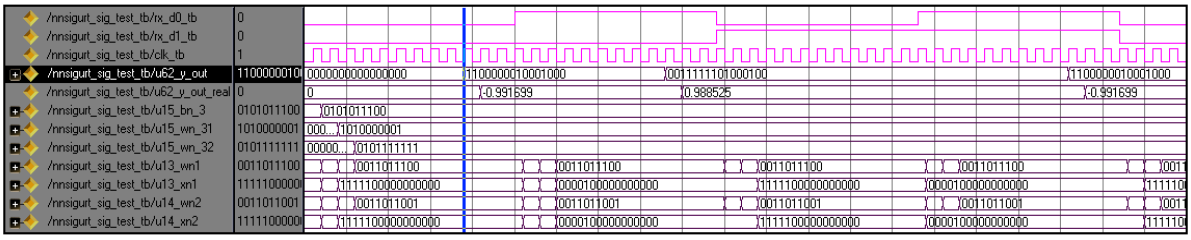

Fig 9: ANN simulation using Tansig/Tansig structure with bipolar XOR function

3.2.2 Example 2 (Bipolar XOR Function)

In this case, there are two steps to achieve. Primarily, the network training is applied to identify a simple dynamic system. Secondly, it is about performing a test with a unit step input to verify the dynamic response values of this system described by the following recursive equation:

$$y(k)=u(k)-0.5y(k-1) \quad (8)$$

where *k* (non-zero positive integer) is the sample number, *y(k)* is the output of the system and *u(k)* is its input. This is a first order digital filter with infinite impulse response (IIR). The output is determined by a linear combination of the previous input and output (feedback). In theory, the impulse response of such a filter never dies out completely, though in practice, this is not true given the finite resolution of computer arithmetic [11]. A digital filter is a system that performs mathematical operations on a sampled discrete-time signal to reduce or enhance certain aspects of that signal. Digital filter implementations, compared to analog ones, offer lots of advantages, such as high accuracy, small physical size. In addition, multi-rate operation digital filtering can be applied to very low frequency signals, such as those occurring in biomedical applications, very efficiently [12].

The system is excited by a sinusoidal signal defined as: $u(k) = sin(\pi/4)$. The used neural network is a recurrent network which has two inputs $x_1 = u(k)$ and $x_2 = y(k-1)$ and an output *y(k)*. The identification of this dynamic system is thus achieved with ANN training in which we use, for simplicity, only 8 samples of the signal *u(k)* as presented in Table 3. It is possible to increase the number of samples (for example to 80) to improve the accuracy and avoid

confusion with other values. Fig 10 shows the set of simulated values with this system (identification) using the same ANN (with feedback), having a Tansig/Purelin structure.

Table 3. Dynamic system samples

k	*u(k)*	*y(k-1)*	*y(k)*
1	0.71	0	0.71
2	1	0.71	0.65
3	0.71	0.65	0.38
4	0	0.38	-0.19
5	-0.71	-0.19	-0.61
6	-1	-0..61	-0.71
7	-0.71	-0.71.	-0.36
8	0	-0.36	0.18

Then (after training), a simple test is performed using a unit step signal (*u(k)* = 1). According to Table 3, the output must converge to the value *y(k)* = 0,65. After the sixth recurrence, the network output converges to the value *y(6)* = 0,6562 quite close to 0,65. To verify this value, a network simulation with ModelSim is performed as shown in Fig 11. The results are also compared to those found with Matlab simulation and practice test (see Tables 9 and 10). The simulation is performed using a single switcher (*rx_d0*) to enable/disable the connection between the output signal *y* and the input signal x_2. In practice, we must delay the output (for example to one second) in order to view the rapid variation of the dynamic response using seven segment display module.

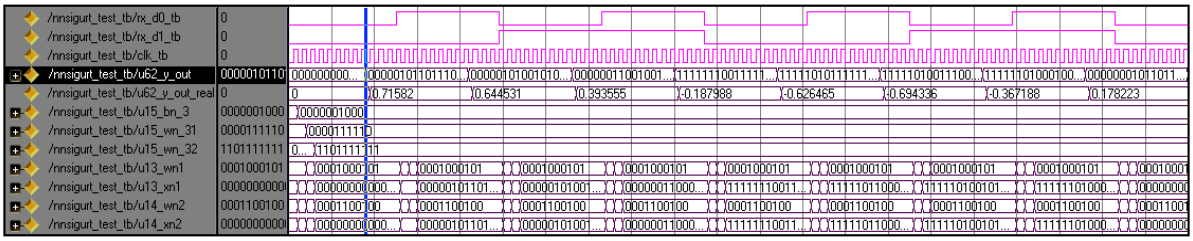

Fig 10: ANN simulation using Tansig/Purelin structure (Example 2 identification)

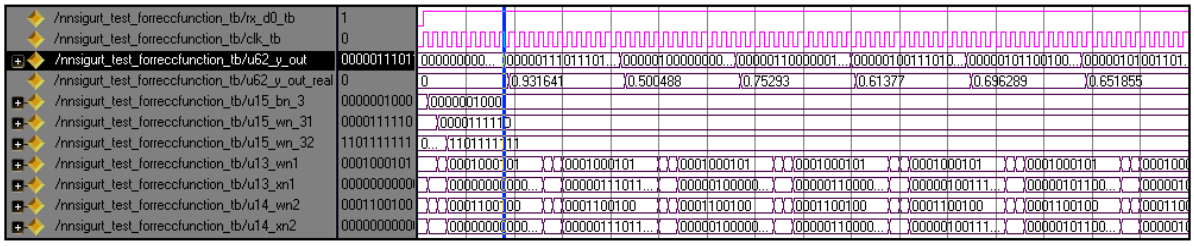

Fig 11: ANN simulation using Tansig/Purelin structure (Example 2 test)

4 SYNTHESES AND IMPLEMENTATION

4.1 Neuron Model Synthesis

The neuron RTL and logic synthesis are performed with the ISE Webpack software from Xilinx and using XC3S200 Spartan3 FPGA. In this work, we synthesized two different

neuron models. The first one uses the simple Purelin activation function. The second one uses Tansig activation function which is more complex to synthesis. The principal method of Tansig function approximation is performed with look-up table saved in one BRAM of the FPGA. The different resources of each neuron model are presented in Table 4. It is obviously clear that BRAM usage highly reduces the number of slices and LUTs of the FPGA. However, the number of approximation values and the FPGA BRAMs limits the number of neurons. Spartan3 FPGA contains 18 blocks of 18 Kbits BRAM which largely sufficient to implement a single neuron or simple ANN. Another advantage of using BRAM is increasing the maximum working frequency of the neuron circuit. The Purelin model, offers the smallest cycles number and resources since it doesn't require a REF block and a look-up table for the activation function.

Table 4. Resources of the neuron model using XC3S200 Spartan3 FPGA

Resources	Purelin neuron	Tansig neuron	
		Without BRAM	With BRAM
Slices	31	256	42
FFs	50	58	83
LUTs	61	511	81
IOBs	45	45	45
Multipliers	1	1	1
BRAMs	0	1	1
Max Frequency (MHz)	78.489	41.723	81.112
Min Period (ns)	12.741	23.968	12.329
Cycles	4	4.5	4.5

In Table 5, we present a comparison between the different resources of the proposed model and some models discussed in related works. To perform this comparison, our model is synthesized with the same FPGA board, the same number of inputs and the same Tansig activation function.

Table 5. Resources comparison of the neuron model using Tansig activation function

Resources	Our work (1) Spartan3E SP	[3] Spartan3E SP	Our work (2) Virtex6 SP	[4] Virtex6 PPP	Our work (3) Spartan3E SP	[5] Spartan3E		
						SP	PPP	FPP
Inputs	3	3	3	3	12	12	12	12
Inputs bits	16	4	16	32	16	16	16	16
Weight bits	10	4	10	32	10	16	16	16
Slices	42	39	44	1862	46	41	305	270
FFs	83	8	81	7449	87	55	235	207
LUTs	81	72	94	6052	92	76	325	296
IOBs	45	25	45	45	45	34	210	210
Multipliers	1	4	(1DSP)	(1DSP)	1	2	13	13
BRAMs	1	0	1	0	1	0	0	0
Max freq (MHz)	81.112	27.500	218.662	405.022	80.758	61	72	78
Min Period (ns)	12.328	36.363	4.573	2.469	12.383	16.393	13.888	12.820
Cycles	4.5	3	4.5	126	13.5	14	6	4

In the first comparison, Spartan3E board is used with three inputs. The neuron in [3] uses also an SP MAC, a fixed-point coding and a direct digital technique to approximate Tansig function. As we can see, it offers better resources usage in general except the number of multipliers and the maximum frequency. However, it only uses 4 bits to code both of inputs and weights versus 16 bits inputs and 10 bits weights. This means that our model offers a better precision. In the second comparison, the used board is Virtex6. The neuron in [4] has also three inputs but uses a PPP MAC with 32 bits floating-point coding. It uses a COordinate Rotation Digital Computer (CORDIC) based exponent calculator to approximate Tansig function. Our model offers a clear better performance in slices and LUTs resources and especially in clock cycles number with 4.5 versus 126 cycles. The last comparison is performed with 12 inputs using Spartan3E FPGA. The neuron in [5] uses 16 bits fixed-point coding and three different types of MAC. The results of memory and time resources in SP MAC are close to our results. However, PPP and FPP MACs offer better timing results but consumes more memory resources.

The RTL schematic of any other component or sub-component can be easily generated and visualized after finishing the process of synthesis with ISE Webpack. For example, the internal structure of TANSIG activation function using BRAM is shown in Fig 12. The encircled internal part is the BRAM RAMB16_S18 module storing 1024 values.

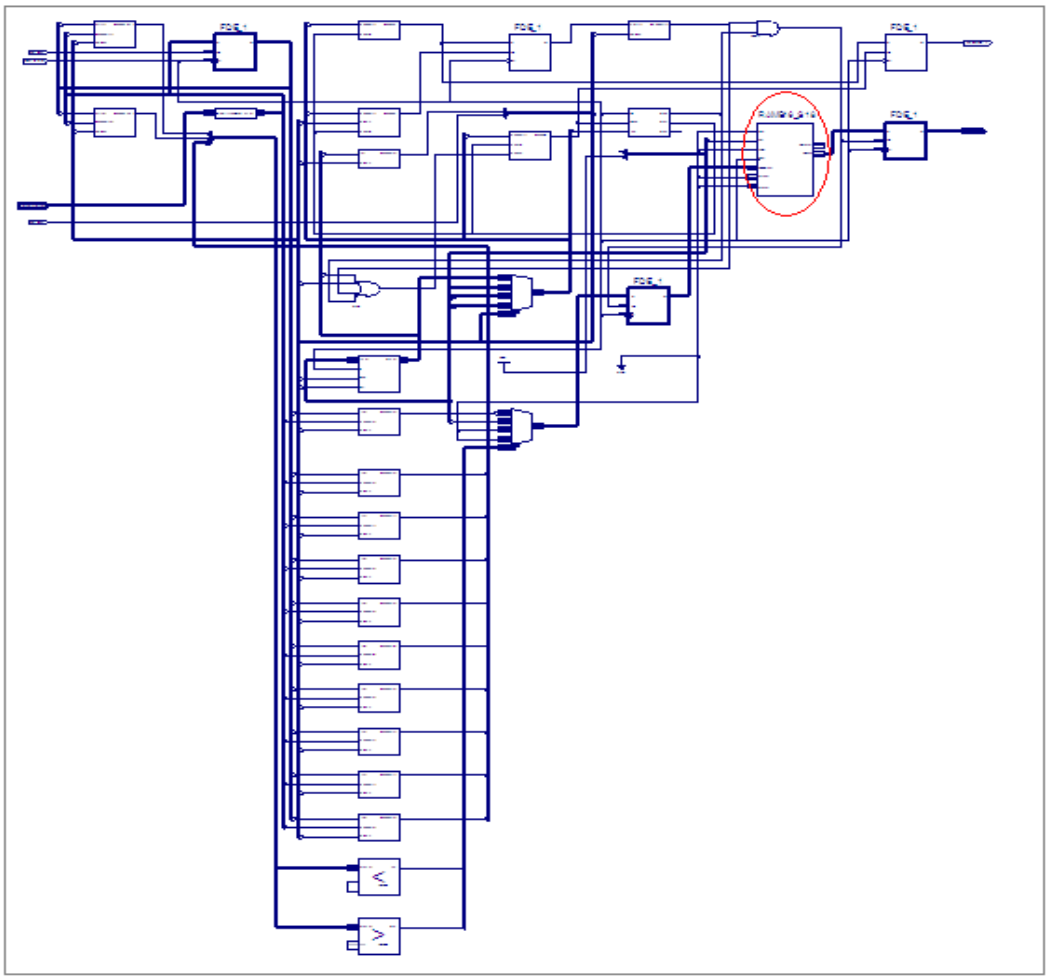

Fig 12: RTL schematic of the Tansig function using BRAM

The RTL schematic of the neuron model is shown in Fig 13. We can note the connection between the accumulator output with its second input using an external wire. Simply, we can verify that the Neuron RTL Schematic reflects the desired neuron model presented in Fig 2.

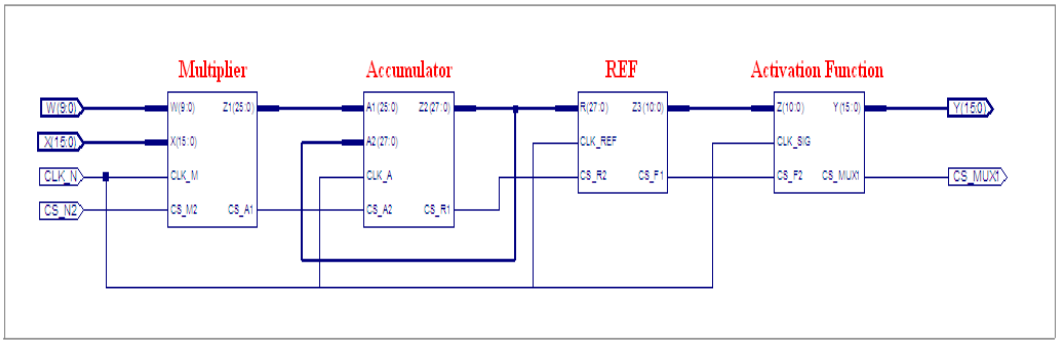

Fig 13: RTL schematic of the neuron

4.2 ANN Synthesis and Implementation

The implantation of the ANN is performed in Spartan3 FPGA. The ANN used in example 2 is the same one used in example 1 with just a little modification in the feedback that lightly changes the RTL schematic and the memory consumption. In this step, we can also make a comparison in terms of memory usage with and without BRAM. In both cases, the number of values stored of the Tansig function is 1024. The distributed memory usage comparison is shown in Table 6.Without BRAM, the main resources are 1475 slices and 2592 for LUTs, against respectively 728 and 1246 using BRAM. These values show that working with BRAM offer an important advantage when using complex architecture consuming large amount of distributed memory that some FPGA boards cannot stand.

Table 6. Resources of the ANN

Resources	Tansig/Purelin ANN (With BRAM)	Tansig/Tansig ANN	
		Without BRAM	With BRAM
Slices	733	1475	728
FFs	389	346	423
LUTs	1254	2592	1246
IOBs	24	24	24
Multipliers	3	3	3
BRAMs	2	0	3
Max Frequency (MHz)	11.697	11.867	12.049
Min Period (ns)	85.494	84.265	82.996
Cycles	9.5	10	10

The RTL schematic of the developed network is shown in Fig 14. It reflects the one presented in Fig 6. The final implementation in FPGA board is done with the iMPACT tool of Xilinx allowing loading the bit stream into the FPGA memory.

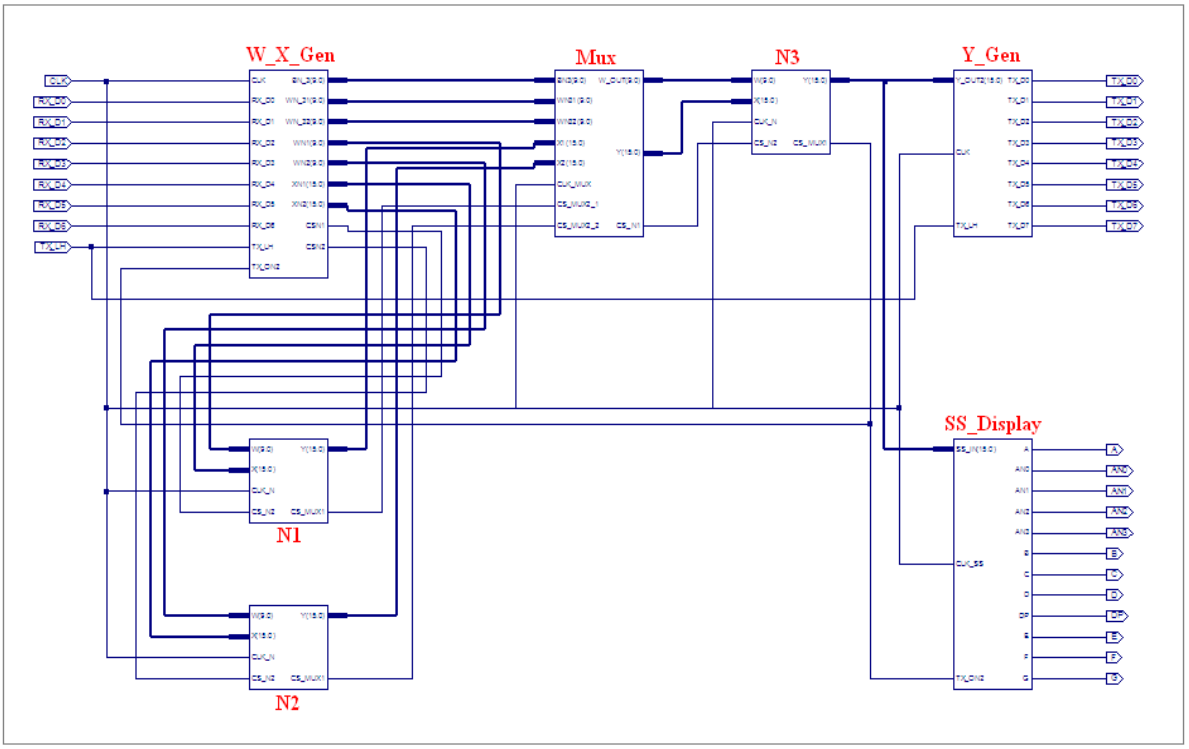

Fig 14: RTL schematic of the ANN

Fig 15 shows two photos taken during the practical test of the neural network implementation of example 1 with XC3S200 Spartan3 FPGA. There are two display modes (with the component SS_Display). In the first mode (Fig 15.a), one digit of the seven segment display module is reserved for the value sign, allowing easy reading. In the second mode (Fig 15.b), the display is made with four digits to improve the precision. However, it is not possible to directly identify the sign, unless using a single LED or by viewing the same result with the 8 LEDs at the same time.

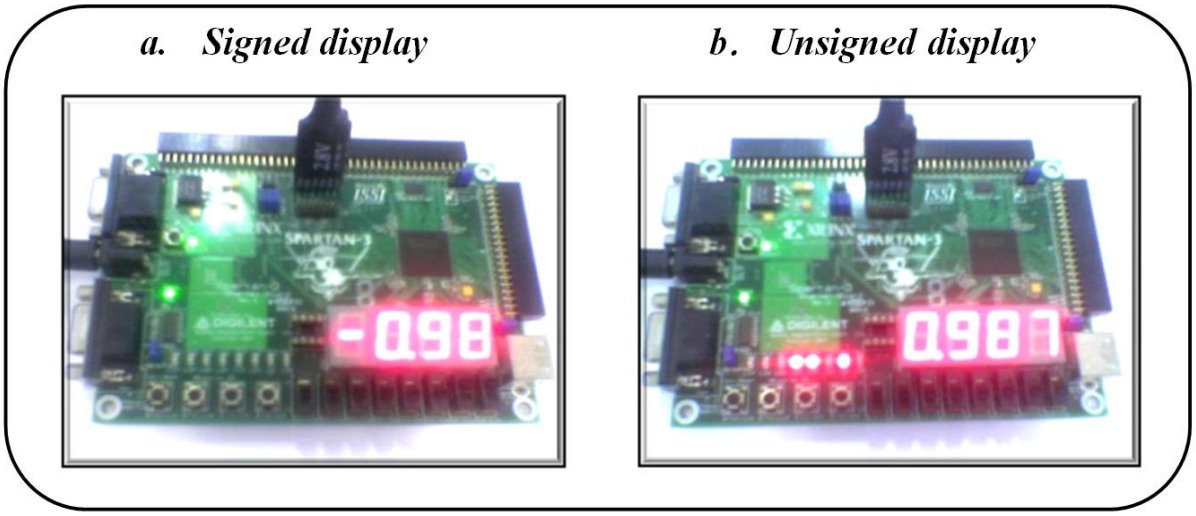

Fig 15: Signed and unsigned display using seven segment display module of Spartan3 FPGA board

5 Comparison of the Different Results

As indicated in Table 7, it is clear that the structure Tansig/Purelin is showing the smallest number of iterations (epochs) and then the fastest training. We can also note that it is possible to generalize this result not only for this example, but also for other examples with higher number of hidden layers [10].

Table 7. Training speed comparison by the number of epochs of the example 1

e	Tansig/Purelin	Tansig/Tansig
10^{-4}	121	2155
10^{-3}	98	2231
10^{-2}	74	305

Table 8 summarizes the different results produced with the two network structures. This involves comparing the different values of network outputs, found in Matlab and ModelSim, with practical ones viewed using signed mode with seven segment display module. The visualized values in practice are highly consistent with those found in simulation. It can be inferred that using Tansig/Purelin structure offers generally more accurate results than the Tansig/Tansig structure. These values are remarkably similar with those found in practice. The ANN has a simple structure containing two neurons in one single hidden layer, which help to decrease the training time and number of epochs. Besides, increasing the number of hidden neurons can reduce the error *e* value [9]. However, not only the training time increases but also the memory usage.

Table 8. Output precision comparison of the example 1

Structure	Output	Theory	Matlab $e = 10^{-4}$	Matlab $e = 10^{-3}$	Matlab $e = 10^{-2}$	ModelSim $e = 10^{-4}$ $n = 64$	ModelSim $e = 10^{-4}$ $n = 256$	ModelSim $e = 10^{-4}$ $n = 1024$	Practice $e = 10^{-4}$ $n = 1024$
Tansig/Purelin	y_1	-1	-0.9933	-0.9791	-0.9320	-0.9594	-0.9819	-0.9970	-0.99
	y_2	1	0.9875	0.9612	0.8759	0.9765	0.9882	0.9916	0.99
	y_3	1	0.9880	0.9629	0.8823	0.9648	0.9882	0.9877	0.98
	y_4	-1	-0.9933	-0.9791	-0.9320	-0.9594	-0.9819	-0.9977	-0.99
Tansig/Tansig	y_1	-1	-0.9917	-0.9736	-0.9156	-0.9895	-0.9913	-0.9916	-0.99
	y_2	1	0.9885	0.9639	0.8863	0.9866	0.9881	0.9885	0.98
	y_3	1	0.9885	0.9640	0.8873	0.9866	0.9881	0.9885	0.98
	y_4	-1	-0.9917	-0.9736	-0.9156	-0.9895	-0.9913	-0.9916	-0.99

The only structure used in example 2 is Tansig/Purelin. For the identification, Table 9 shows the results obtained in simulation using Matlab and ModelSim. The best ones are obtained with the smallest error ($e = 10^{-5}$) and the highest number of activation function samples ($n = 1024$). As we can see, even if we reduce the mean squared error, the number of activation function samples still plays a vital role in the precision. However, apart from consuming more memory, generating more samples in BRAM can increase the difficulty of the approximation itself, especially when we need to use several BRAMs for the same look-up table.

Table 9. Output precision comparison (identification) of the example 2 using Tansig/Purelin structure

Output	Theory	Matlab $e = 10^{-5}$	Matlab $e = 10^{-4}$	Matlab $e = 10^{-3}$	ModelSim $e = 10^{-5}$ $n = 64$	ModelSim $e = 10^{-5}$ $n = 256$	ModelSim $e = 10^{-5}$ $n = 1024$	Practice $e = 10^{-5}$ $n = 1024$
y(1)	0.71	-0.7096	-0.7028	0.6584	0.6464	0.7089	0.7158	0.71
y(2)	0.65	0.6470	0.6478	0.6539	0.6166	0.6254	0.6445	0.64
y(3)	0.38	0.3827	0.3904	0.4175	0.3310	0.3623	0.3935	0.93
y(4)	-0.19	-0.1891	-0.2019	0.1868	-0.1245	-0.1811	-0.1879	-0.18
y(5)	-0.61	-0.6137	-0.6098	-0.5853	-0.5771	-0.6142	-0.6264	-0.62
y(6)	-0.71	-0.7054	-0.6974	-0.6727	-0.6064	-0.6860	-0.6964	-0.69
y(7)	-0.36	-0.3643	-0.3739	-0.4010	-0.3315	-0.3520	-0.3671	-0.36
y(8)	0.18	0.1832	0.1919	0.1659	0.1274	0.1579	0.1782	0.17

In the unit step input test, like in identification, several simulations are performed using Matlab with different values of the mean squared error *e*. The results of each simulation are then compared to those found in ModelSim with the smallest error *e* and different values of samples number *n*. According to Table 10, the best results are generated with $e = 10^{-5}$ and $n = 1024$. This is again quite logical since the error is reduced and the accuracy is increased and it confirms the proper functioning of the network. In example 2, the results obtained in practice, whether they are for identification or for the test with the unit step input, are reasonably close to those obtained with the two simulations.

Table 10. Output precision comparison (unit step input test) of the example 2 using Tansig/Purelin structure

Output	Theory	Matlab $e = 10^{-5}$	Matlab $e = 10^{-4}$	Matlab $e = 10^{-3}$	ModelSim $e = 10^{-5}$ $n = 64$	ModelSim $e = 10^{-5}$ $n = 256$	ModelSim $e = 10^{-5}$ $n = 1024$	Practice $e = 10^{-5}$ $n = 1024$
y(1)	1.0000	0.9338	0.9097	0.8286	0.8110	0.9208	0.9409	0.94
y(2)	0.5000	0.4942	0.5336	0.6092	0.5566	0.4995	0.5034	0.50
y(3)	0.7500	0.7700	0.7349	0.6883	0.6645	0.7646	0.7812	0.78
y(4)	0.6250	0.6082	0.6344	0.6616	0.5034	0.6220	0.6176	0.61
y(5)	0.6875	0.7085	0.6868	0.6708	0.6044	0.6918	0.7148	0.71
y(6)	0.6562	0.6480	0.6600	0.6677	0.6645	0.6381	0.6577	0.65

6 CONCLUSION

In this work, optimized hardware implementation of a neural architecture is developed. The Tansig activation function is mainly performed using FPGA BRAM to save the SRAM distributed memory. The proposed neuron model uses a serial processing MAC in order to economize more the available memory resources of the FPGA. This model offers also a good timing performance (number of clock cycles) compared to other related works in the same domain. From this model, we designed and succeeded to implement a simple two-layer ANN. Then, to test this implementation, we used two simple examples that are the bipolar XOR function and the first order digital filter. Moreover, the range of the input layer values can be further extended to approximate other functions, especially with the structure Tansig/Purelin

offering fast training and consuming less memory. In addition, this same architecture can be extended by increasing the number of hidden layers of the ANN. This can increase the number of weights combinations, reduce the mean squared error and can serve to implement more complex and efficient neural architectures.

REFERENCE

[1] Saumya B, Kreeti J, Neeti J (2011) Artificial Neural Networks. International Journal of Soft Computing and Engineering (IJSCE), vol 1, p 27
[2] Haitham KA, Esraa ZM (2010) Design of an Artificial Neural Network Using FPGA. International Journal of Computer Science and Network Security (IJCSNS), vol 10, p 88
[3] Rafid AK, Sa'ad AA (2009) Digital Hardware Implementation of Artificial Neurons Models Using FPGA. Al-Rafidain Engineering, vol 17(3), p 13
[4] Sahin I, Koyuncu I (2012) Design and Implementation of Neural Networks Neurons with RadBas, LogSig, and TanSig Activation Functions on FPGA. Electronics and Electrical Engineering, vol 4(120), p 53
[5] Sa'ad AA, Rafid AK (2008) FPGA Implementation of Artificial Neurons: Comparison Study. IEEE, Information and Communication Technologies: From Theory to Applications (ICTTA), 3rd International Conference, p 6
[6] Milene BC, Alexandre MA, Luiz ESR, Carlos APSM, Petr E (2005) Artificial Neural Network Engine Parallel and Parameterized Architecture Implemented in FPGA. Springer, Lecture Notes in Computer Science (LNCS), p 295
[7] David B (2008) Fixed Point Package User's Guide. VHDL-2008 Support Library, pp 2-12
[8] Tung-Chueng C, Ru-Jen C (2006) Application of Back-Propagation Networks in Debris Flow Prediction. Elsevier, Engineering Geology, vol 85, p 275
[9] Gaurang P, Amit G, Kosta YP, Devyani P (2011) Behaviour Analysis of Multilayer Perceptrons with Multiple Hidden Neurons and Hidden Layers. International Journal of Computer Theory and Engineering (IJCTE), vol 3(2), p 337
[10] Joseph A, Bong DBL, Mat DAA (2009) Application of Neural Network in User Authentication for Smart Home System. World Academy of Science, Engineering and Technology (WASET), vol 53, pp 1295-1298
[11] Amir AB, Sunusi SA (2012) Design of Digital Recursive Filter Using Artificial Neural Network. Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS), vol 1, p 1
[12] Gurleen K, Ranjit K (2007) Design of Recursive Digital Filters using Multiobjective Genetic Algorithm. International Journal of Engineering Science and Technology (IJEST), vol 3(7), p 5614