ON THE IMPLEMENTATION OF DIFFERENT HYPERBOLIC TANGENT SOLUTIONS IN FPGA

Darío Baptista and Fernando Morgado-Dias

Madeira Interactive Technologies Institute and Centro de Competências de Ciências Exactas e da Engenharia, Universidade da Madeira
Campus da Penteada, 9000-039 Funchal, Madeira, Portugal.
Tel:+351 291-705150/1, Fax: +351 291-705199

Abstract: This paper describes the development of an artificial neuron using the blocks from the library of System Generator. The advantage of using System Generator's block is the ease of conversion to HDL language and the facility of implementation into FPGA. The first solution presented here consists of an efficient method of "query-by-table" (lookup table). In this method, the function's values are pre-calculated and stored in a table. The second set of solutions consists in implementing third degree polynomials. For the polynomial implementation three methods were developed using different hardware for the multiplication: Booth's algorithm, 2's complement and Digital Signal processors. The best solution describes the hyperbolic tangent with an error equal to $1x10^{-14}$.

Keywords: Artificial Neural Networks, System Generator, Hyperbolic Tangent.

1. INTRODUCTION

Almost since the beginning of Artificial Neural Networks' (ANN) use for modelling and control, a need for hardware implementation has arisen.
This need conducted to many experimental and academic prototypes and several commercial solutions. One of the first commercial implementations was developed by Nestor and Intel in 1993 for an Optical Character Recognition (OCR), which uses Radial Basis Functions (Morgado-Dias, 2004).
The need for physical implementations also arises in medical applications of ANN which work inside or in close connection with the body. The work referred in (Stieglitz, 2006) contains many examples of neural vision systems that cannot be used attached to computers.
A large share of the difficulty of implementing an ANN in hardware comes from the non-linear activation function, most of the times a hyperbolic tangent. Since this function cannot be easily calculated by elementary functions, it has to be replaced by an approximation that is both less resource consuming and does not delay the calculation too much. To this end many solutions have been tested: piecewise linear functions, polynomial approximations, Taylor Series, Elliot functions and the CORDIC algorithm. These solutions are only a few examples.
The best solutions found in the review of the literature are: (Pinto , 2006) that with a set of five 5th order polynomial approximations with a maximum error of $8x10^{-5}$ (using the sigmoid function) and (Ferreira, 2007) that with a piecewise linear approximation determined by an optimized algorithm which achieved $2.18x10^{-5}$ error in the hyperbolic tangent and a $5x10^{-8}$ error in a control loop simulation.
This paper presents an automated solution that makes use of System Generator of Xilinx to obtain an FPGA implementation of a neuron. This work is a development of the one published in (Reis et. al 2011), with improved solutions for the activation function.

2. THEORETICAL DESCRIPTION

A basic block for the construction of an artificial neuron is the appropriate choice of activation function. This paper is focused on the implementation of the hyperbolic tangent using the

System Generator block. In the first place, for building this activation function it is necessary to analyze its parity. Figure 1 shows that this function presents symmetry at the origin, i.e., it is an odd function.

$$f(-x) = -f(x) \qquad (1)$$

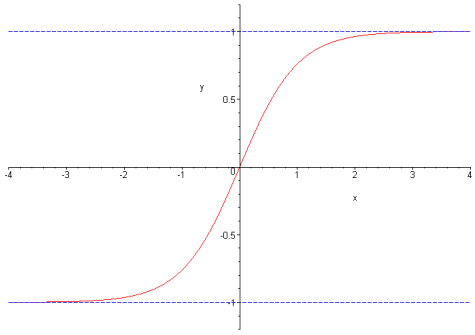

Fig. 1: Graphical representation of hyperbolic tangent

2.1 *Implementation of Hyperbolic Tangent using the ROM*

The first implementation of the hyperbolic tangent consists in storing, in a Read Only Memory (ROM), the function's values for input values between 0-6. For inputs values greater than 6 the function is equal to 1.
The implementation of the hyperbolic tangent function using block System Generator is represented in Figure 2 (Reis et. al, 2011).

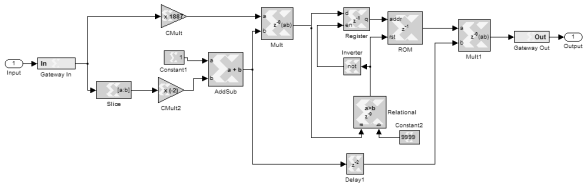

Fig. 2: Implementation of the hyperbolic tangent function using a ROM.

The ROM stores the values of the hyperbolic tangent but the inputs of the LUT are the memory addresses instead of the inputs of the hyperbolic tangent. The outputs will be fixed point values and these numbers will be unsigned. For the LUT to perform the required function several transformations are necessary. These changes are done by the subsystem represented in figure 2 (Reis et. al, 2011).
This block achieves its function using a ROM to store the values of the hyperbolic tangent. To make an efficient use of the memory the hyperbolic tangent is only represented between the positive values of 0 and 6 since this function is symmetrical and saturates.

2.2 *Implementation of Hyperbolic Tangent using an interpolating polynomial*

The second implementation of this function consisted of using a polynomial interpolator. To do this, a set of polynomials was determined that could describe the hyperbolic tangent function for input values between 0 and 6.

To determine these polynomials the Chebyshev interpolation was used. To achieve high accuracy the domain of the function was split in to smaller intervals. For this implementation 25 polynomials of third-degree were used, in order to describe the hyperbolic tangent with a Mean Square Error (MSE) equal to $1x10^{-14}$. It is noted that, for values of function's domain higher than 6, the function is interpolated by a constant value 1. Generally, each polynomial obtained has the following form:

$$P(x) = a + bx + cx^2 + dx^3 \qquad (2)$$

Note that the representation for all coefficients is done with 32 bits, where 2 bits are for the integer part and 30 bits are for fractional part.
If the polynomial was implemented exactly as written in eq.(2), 4 additions and 6 multiplication would have to be performed. However, it is possible to write the same polynomial, but using only 3 multiplications and 3 additions, which saves resources in the FPGA.

$$P(x) = a + (b + (c + dx)x)x \qquad (3)$$

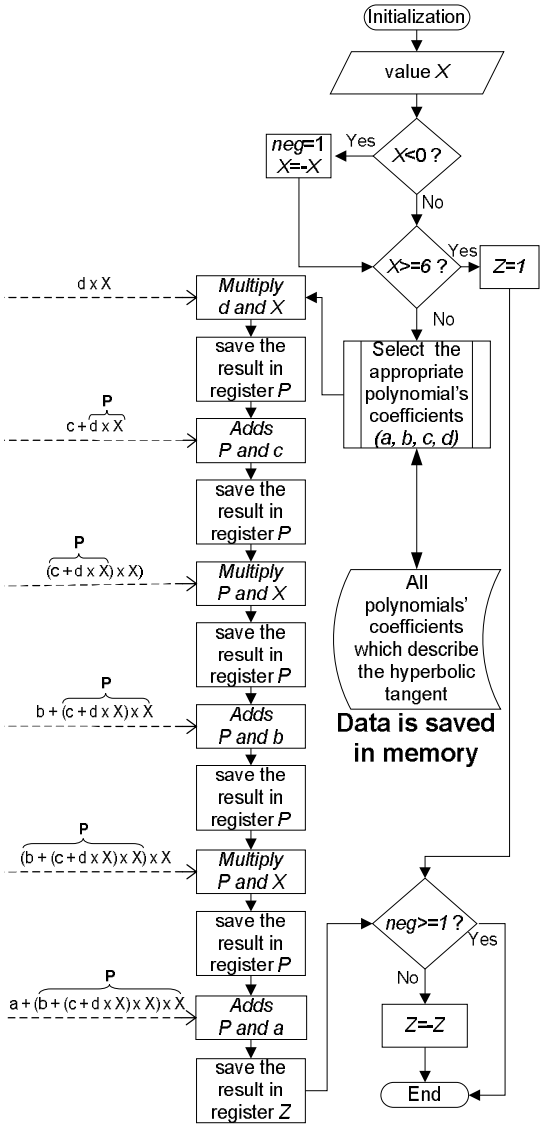

Fig. 3: Flowchart descriptive of hyperbolic tangent function using polynomial interpolation

Figure 3 shows the flowchart of the polynomial interpolator's implementation. It shows that to implement this idea it is necessary to use only two operations: addition and multiplication. The multiplication operation was implemented in three different forms: using a specific feature of the FPGA (DSP - Digital Signal Processor), using Booth's algorithm and using a 2's complement multiplication.

2.2.1 Multiplication using DSPs

The DSPs are devices which have the capacity to process large quantities of operations in a short time. This ability of DSPs makes this device perfect for applications where delay is not tolerable. When using an FPGA that has this resource available the user can establish whether they should be used for an operation or not.

2.2.2 Multiplication using 2's complement

This implementation allows a multiplication without the use of DSPs. Figure 4 shows the procedure to calculate a multiplication using 2's complement (Morgado Dias, 2010). In this example numbers of 4 bits are used. The multiplicand can have any sign and the multiplier must be positive.

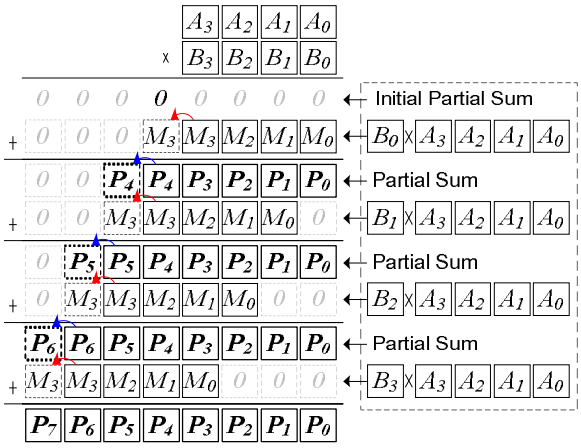

Fig. 4: Example of multiplication in 2's complement using 4-bit numbers.

Figure 4 shows that, at the beginning, an initial partial sum is applied (i.e. the partial sum is 0 because the accumulator is initiated to 0). The first multiplication, which is the result of the multiplication of the first digit of the multiplier (B0) by the multiplicand is placed underneath this partial sum (Morgado Dias, 2010).
The next step is to calculate the next partial sum where an extension of signal is applied. Underneath this last partial sum, we put the multiplication's result of second digit of the multiplier (B1) by the multiplicand, after shifted and signal extension. This procedure is repeated until meeting the last multiplication between last multiplier's digit (B3) by multiplicand, obtaining, in this way, the desired result (Morgado Dias, 2010).
Figure 5 shows the implementation of the algorithm using 2's complement.

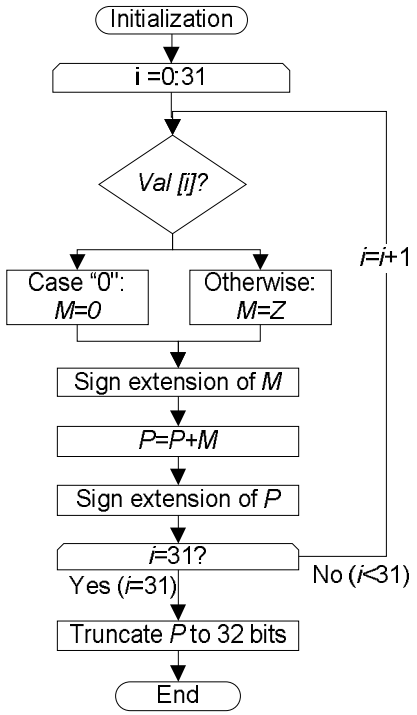

Fig. 5: Flowchart descriptive of the implementation of the multiplication using 2's complement.

2.2.3 Implementation of multiplication using Booth's algorithm.

This implementation also allows doing a multiplication without the use of DSPs. Figure 6 shows the procedure to calculate a multiplication using Booth's algorithm (Sêrro, 2007). This procedure uses a 4-bit numbers with a positive multiplier and a multiplicand of any signal.

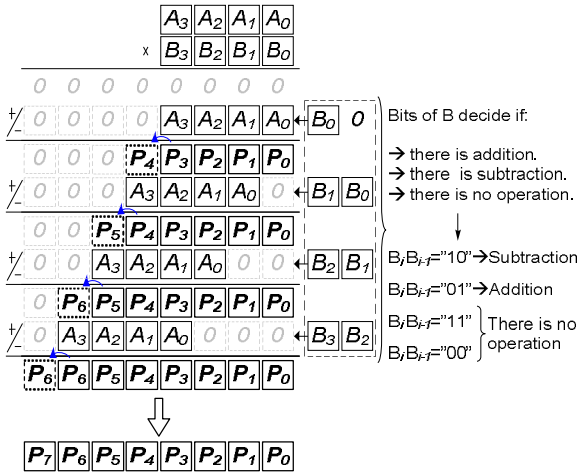

Fig. 6: Example of multiplication using Booth's algorithm.

Figure 6 shows that, at the beginning, an initial partial sum is applied (i.e. the initial partial sum is equal to 0 because the accumulator is initiated with 0) and a copy of the multiplicand is placed underneath this partial sum. The operation held between the partial sum and multiplied is determined according to the pair of bits "B_i, B_{i-1}" Table 1 shows all possible combinations of the pair "B_i, B_{i-1}" and their respective operations (Sêrro, 2007).

Table 1: Possible combinations of the pair "B_i, B_{i-1}" and their respective operations.

B_i ; B_{i-1}	Operation between the partial sum and multiplied
"10"	Subtraction
"01"	Addition
"00"	There is no operation
"11"	There is no operation

To get the next partial sum sign extension is applied. Next, the multiplicand is placed under the partial sum after shifted. One more time, table 1 must be consulted to determine the operation This is repeated until the last pair of bits of the multiplier ("B3, B2") is reached, obtaining the desired result (Sêrro, 2007). Figure 7 shows the implementation of the Booth's algorithm.

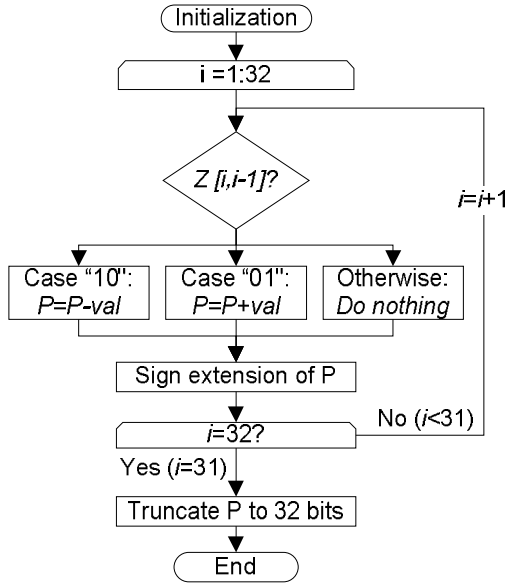

Fig. 7: Flowchart descriptive of implementation of multiplication using Booth's algorithm.

3. RESULTS

To compare all these solutions, each system was compiled and a co-simulation block created. This allows to simulate the system's behaviour when it is inserted into the FPGA.
This co-simulation block was created for the FPGA Virtex 5 XC5VFX130T-1FF1738. Table 2 presents the resources used in all the solutions presented.
Figure 8 shows the system created to do the test with the co-simulation block. To test the implementations an input consisting of 1200 values ranging between -6 and 6 was used.
Figure 9 and 10 show the error between the hyperbolic tangent values and the response of each system. It is important to note which the figure 9 is the same to the three solutions using the polynomial interpolation.
Comparing figures 9 and 10 it can be seen that the system which uses ROM presents an error 10 times higher than the system which uses the polynomial interpolator.

Analysing table 2 it can be seen that between the solutions which use the polynomial interpolator, the use of DSPs saves the other resources of the FPGA.
The implementation that uses the ROM to hold the hyperbolic tangent values contains 10000 values and the representation for each theses values is done with 32 bits.
The implementation where the multiplication is done with the Booth's algorithm is the solution that uses more resources. On the other hand, this alternative and 2's complement do not require the use of DSP. The latter solution saves resources when compared with the algorithm of Booth. So, if the intention is to save DSPs, it will be better to use the multiplication with 2's complement.
The solution which uses the ROM can save more resources than the any other solution, although the error is higher and more memory is used.

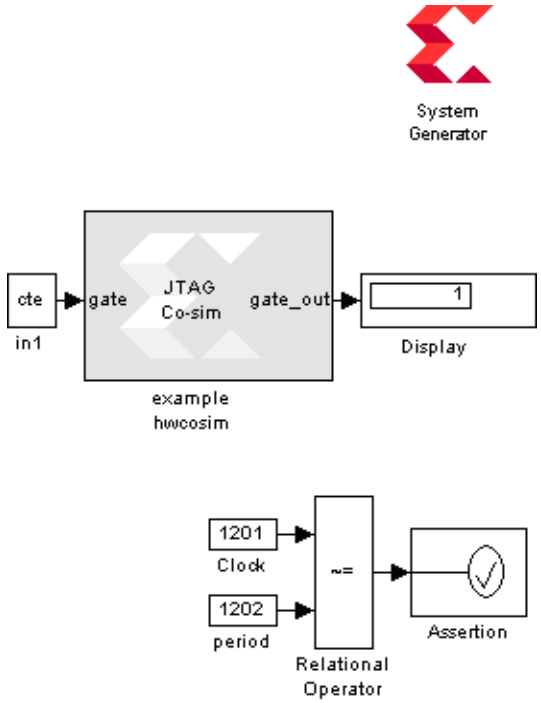

Fig. 8: Test system for co-simulation block.

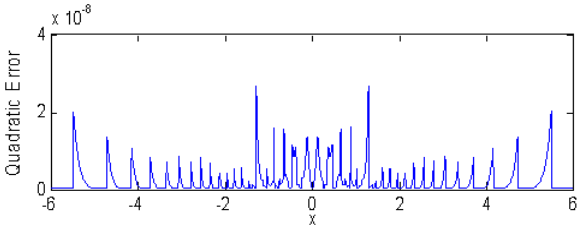

Fig. 9: Error between the values of the hyperbolic tangent and the system response using the polynomial interpolation.

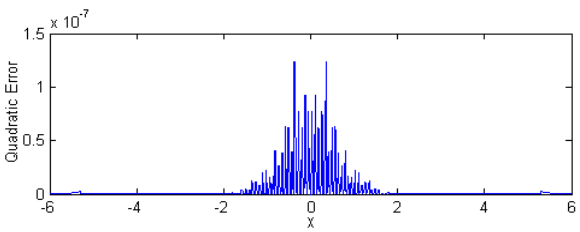

Fig. 10: Error between the values of hyperbolic tangent and the system response using ROM

Table 2: The resources of FPGA used for each system

		Design Summary of Hyperbolic Tangent			
		ROM	Polynomial using multiplication with:		
			DSP	2's Complement	Algorithm of Booth
Slice Logic Utilization	**Number of Slice Registers**	642 out of 81,920 (1%) *Number used as Flip Flops: 641* *Number used as Latch-thrus: 1*	424 out of 81,920 (1%) *Number used as Flip Flops: 424*	424 out of 81,920 (1%) *Number used as Flip Flops: 424*	424 out of 81,920 (1%) *Number used as Flip Flops: 424*
	Number of Slice LUTs	739 out of 81,920 (1%)	1,003 out of 81,920 (1%)	2,558 out of 81,920 (3%)	5,444 out of 81,920 (6%)
	Number used as logic	637out of 81,920 (1%) *Number using O6 output only: 404* *Number using O5 output only: 146* *Number using O5 and O6: 87*	996 out of 81,920 (1%) *Number using O6 output only: 670* *Number using O5 output only: 86* *Number using O5 and O6: 240*	2,551 out of 81,920 (3%) *Number using O6 output only: 2,220* *Number using O5 output only: 87* *Number using O5 and O6: 244*	5,437 out of 81,920 (6%) *Number using O6 output only: 5,063* *Number using O5 output only: 87* *Number using O5 and O6: 287*
	Number used as Memory:	92 out of 25,280 (1%) *Number using O6 output only: 90* *Number using O6 output only: 2*	--	--	--
	Number used as exclusive route-thru	10	7	7	7
	Number of route-thrus	156 *Number using O6 output only: 156*	93 *Number using O6 output only: 93*	93 *Number using O6 output only: 93*	94 *Number using O6 output only: 93* *Number using O5 and O6: 1*
Slice Logic Distribution	**Number of occupied Slices**	303 out of 20,480 (1%)	419 out of 20,480 (2%)	766 out of 20,480 (3%)	1,730 out of 20,480 (8%)
	Number of LUT Flip Flop pairs used	962 *Number with an unused Flip Flop: 320 out of 962 (33%)* *Number with an unused LUT: 223 out of 962 (23%)* *Number of fully used LUT-FF pairs: 419 out of 962 (43%)*	1,158 *Number with an unused Flip Flop: 734 out of 1,158 (63%)* *Number with an unused LUT: 155 out of 1,158 (13%)* *Number of fully used LUT-FF pairs: 269 out of 1,158 (23%)*	2,685 *Number with an unused Flip Flop: 2,261 out of 2,685 (84%)* *Number with an unused LUT: 127 out of 2,685 (4%)* *Number of fully used LUT-FF pairs: 297 out of 2,685 (11%)*	5,591 *Number with an unused Flip Flop: 5,167 out of 5,591 (92%)* *Number with an unused LUT: 147 out of 5,591 (2%)* *Number of fully used LUT-FF pairs: 277 out of 5,591 (4%)*
	Number of unique control sets	26	21	21	21
	Number of slice register sites lost to control set restrictions	43 out of 81,920 (1%)	32 out of 81,920 (1%)	32 out of 81,920 (1%)	32 out of 81,920 (1%)
IO Utilization	**Number of bonded IOBs**	1 out of 840 (1%) *Number of LOCed IOBs: 1 out of 1 (100%)*	1 out of 840 (1%) *Number of LOCed IOBs: 1 out of 1 (100%)*	1 out of 840 (1%) *Number of LOCed IOBs: 1 out of 1 (100%)*	1 out of 840 (1%) *Number of LOCed IOBs: 1 out of 1 (100%)*
Specific Feature Utilization	**Number of BlockRAM/FIFO**	12 out of 298 (4%) *Number of 36k BlockRAM used: 9* *Number of 18k BlockRAM used: 5*	2 out of 298 (1%) *Number of 36k BlockRAM used: 2*	2 out of 298 (1%) *Number of 36k BlockRAM used: 2*	2 out of 298 (1%) *Number of 36k BlockRAM used: 2*
	Total Memory used (KB)	414 out of 10,728 (3%)	72 out of 10,728 (1%)	72 out of 10,728 (1%)	72 out of 10,728 (1%)
	Number of BUFG/BUFGCTRLs	4 out of 32 (9%) *Number used as BUFGs: 2* *Number used as BUFGCTRLs: 2*	3 out of 32 (9%) *Number used as BUFGs: 2* *Number used as BUFGCTRLs: 1*	3 out of 32 (9%) *Number used as BUFGs: 2* *Number used as BUFGCTRLs: 1*	3 out of 32 (9%) *Number used as BUFGs: 2* *Number used as BUFGCTRLs: 1*
	Number of BSCANs	1 out of 4 (25%)	1 out of 4 (25%)	1 out of 4 (25%)	1 out of 4 (25%)
	Number of DSP48Es	12 out of 320 (3%)	12 out of 320 (3%)	--	--

4. CONCLUSION

This paper presents different implementations of hyperbolic tangent where the primary aim is to find the best solution which allows to save resources and get the best performance.

The first implementation consists in storing the values of hyperbolic tangent in a ROM. This block works like a lookup table because it is necessary to insert a memory's address to get the value of function. This implementation is a solution which allows to save more resources than the second implementation (using an interpolation polynomial). However, the first solution presents a disadvantage because it presents a higher MSE.

The second implementation consists in using the Chebyshev interpolation. For the specified error 25 polynomials of third order were needed. To implement the polynomials in the FPGA adders and multipliers are needed. The multipliers can be implemented using DSPs. However, the multiplication can be done without DSPs. Two solutions were tested: Booth's algorithm and 2's complement. The idea to implement three different ways to perform a multiplication consists in finding the best balance between performance and resource use.

After analyzing the implementations it can be stated that: using DSPs frees other resources; 2's complement is less resource demanding than Booth's algorithm and both these solutions do not use DSPs.

Therefore, the reader could ask himself the following: among the three implementations using the interpolation polynomial, what are the possible cases for more effective implementation? One plausible answer could be found if the structure of the intended ANN is analyzed. If the ANN has many inputs or many neurons in the hidden layer, it is advisable to use the algorithm of 2's complement. This implementation allows saving DSPs which will be necessary to do the sum of products between the weights and the inputs. Otherwise, it is better to use the multiplication's implementation using DSPs.

5. ACKNOWLEDGMENTS

The authors would like to acknowledge the Portuguese Foundation for Science and Technology for their support for this work through project PEst-OE/EEI/LA0009/2011.

6. REFERENCES

Morgado Dias, F. (2010). *Sistemas Digitais – Princípios e Prática.* Primeira Edição. FCA.

Sêrro, C. (2007). Multiplicação e Divisão inteira. Acessed: 6th March 2012. [Online] http://comp.ist.utl.pt/ec-ac/Files/Powerpoints/Aula05_MultDiv.pdf

Stieglitz, T. and Meyer, J. (2006). Biomedical Microdevices for Neural Implants, BIOMEMS Microsystems. Vol. 16, pp. 71-137.

Morgado-Dias F., Antunes A. and Mota A. (2004). Artificial neural networks: a review of commercial hardware, in Engineering Applications of Artificial Intelligence, Vol.17, pp. 945-952.

Pinto J. O. P., Bose B.K., Leite L. C., Silva L. E. B. and Romero M. E.(2006). Field Programmable Gate Array (FPGA) Based Neural Network Implementation of Stator Flux Oriented Vector Control of Induction Motor Drive, in IEEE International Conference on Industrial Technology.

Ferreira, P., Ribeiro, P., Antunes, A. and Morgado Dias, F. (2007). A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, in Neurocomputing, Vol.71, pp 71-77.

Reis, L., Aguiar, L., Baptista, D., Morgado-Dias, F. (2011), ANGE – Automatic Neural Generator, International Conference on Artificial Neural Networks, Espoo, Finland.