MODELLING A NEURON USING A CUSTOM MATH LIBRARY *sfloat24* – IMPLEMENTATION OF A SIGMOID FUNCTION ON A FPGA DEVICE –

M. C. Miglionico
Department of Culture of the Project – Second University of Napoli
Via S. Lorenzo, Monastero di San Lorenzo, I-81031 Aversa (CE) – ITALY
BENECON Scarl – Member of UNESCO
E-mail: mcristina.miglionico@unina2.it

F. Parillo*
Department of Automation, Electromagnetism, Computer Science and Industrial Mathematics
University of Cassino, Via G. Di Biasio 43, I-03043 Cassino (FR) - ITALY
E-mail: f.parillo@unicas.it

ABSTRACT

Artificial neuron networks base their processing capabilities in a parallel architecture. This makes them useful to solve pattern recognition, system identification and control problems. In this paper is described an artificial neuron implementation on a FPGA device by using a custom *sfloat24* math library. Using this library have been implemented some activation functions of an artificial neuron. In particular this work discusses the analytical formulation of a sigmoid activation function and the consequent implementation on a FPGA device, using the custom built *sfloat24* math library.
The simulation results show that the *sfloat24* floating point math library and the implementation of an artificial neuron network are feasible, and their outperform by high-speed response.

Keywords: Artificial Neuron Networks (ANN), Field Programmable Gate Array (FPGA).

1. Introduction

Artificial Neuron Networks (ANN) are used with success in pattern recognition problems, function approximation, control, etc. Their processing capabilities are based on a parallel architecture. There are different kind of electronic implementation of ANN, digital, analog and hybrid (M.A. Banuelos, j. Castillo Hernandez, S. Quintana Thierry, R. Damian Zamacoma, J. Valeriano Assem, R.E. Cervantes, R. Fuentes Conzalez, G. Calva Olmos, J.L. Perez Silva, 2003) and each one has specific advantage and disadvantages depending on the type and configuration of the network, training method and application.
For a digital implementation of ANN there are different alternatives, custom design, digital signal processors, programmable logic, etc. Programmable logic offers low cost, powerful software tools and true parallel implementations. Field Programmable Gate Arrays ($FPGA_s$) consist in a family of a programmable logic devices based in an array of configurable logic blocks, which gives a great flexibility in the development of digital systems. The amount of their available resources for digital system design gives a great flexibility for implementing complex systems, such as a specific purpose microprocessor, network controllers, digital video systems, and for implementing digital ANNs. (Bogdan M. Wilamowski, 2009).

* Corresponding author

In this paper the authors present the design of an artificial neuron on the ALTERA® Cyclone III EP3C25F324C8 FPGA evaluation board. In general a neuron comprises an input aggregation stage and an output activation function one.
The input stage of implemented neuron is based on a *sfloat24* adder. Fundamentally three types of activation functions have been used: step, ramp-saturation, and sigmoid. For implementing the sigmoid activation function the well known CORDIC (Ray Andraka, 1998) theory is used. The obtained results will be used, as a starting point, for the generation of complex ANN for application requiring parallel computing.
The authors was developed a floating point math library for FPGAs, called *sfloat24* (M.C. Miglionico, F. Parillo, 2010), which is used in this paper for the Artificial Neuron implementation. Respect to the classical IEEE 754 (IEEE Standard for Binary Floating-Point Arithmetic, 1985) (W. Kahan, 1996) floating point number format, the numbers are represented by 24 bit word length. This reduced representation is useful when a small FPGA is used.
The implementation of the Artificial Neuron has been based on the definition of the basic arithmetic operations and of the comparison operator (C. Attaianese, F. Parillo, G. Tomasso, 2010), successively applying the CORDIC theory in order to define the exponential function.

2. Modelling of an Artificial Neuron

A generic neuron, as well known, consist of several weighted inputs, an adder function and an activation function as shown in Figure 1.

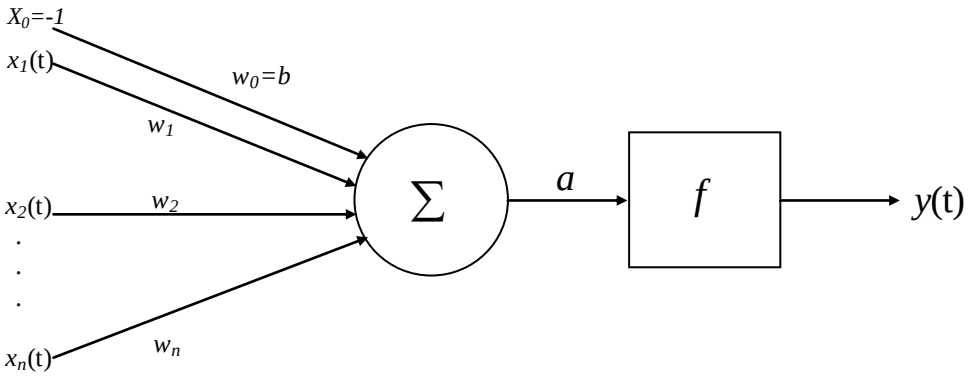

Figure 1. Block diagram of an artificial neuron.

where $y(t)$ is the output of the neuron. W is the input vector, $x_i(t),\ i \in [0;n]$ are the inputs and f is a non linear activation function. This neuron model correspond to the following generalization:

$$y = f(a) = f\left(\sum_{i=0}^{n} w_i \cdot x_i\right) = f(W \otimes X) \tag{1}$$

with $X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$ and $W = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \cdot \\ \cdot \\ w_n \end{bmatrix}$

where: x_0 is equal to -1 and $w_0=b$, where b is a bias and/or a threshold.

The simplest activation function is the step activation function, defined as follow:

$$y(a) = \begin{cases} 0 & if\, a < \theta \\ 1 & if\, a >= \theta \end{cases} \tag{2}$$

where *a*, see Figure 1, is the sum of all input signal and θ is a threshold level.
Another used activation function is the ramp saturation activation function that produces an output gradually growing from zero to a fixed maximum level. This function can be expressed as:

$$y(a) = \begin{cases} a & if\, a < \theta \\ \vartheta & if\, a >= \theta \end{cases} \tag{3}$$

where θ is the saturation level.
The back-propagation (BP) learning process is widely adopted as a successful learning rule to find the appropriate values of the weights for ANNs. For example the Multi-Layer networks consists of various layers: an input and output ones between which lie one or several hidden ones whose outputs are not observable. These layers are based upon some processing unit (neurons) interconnected by means of feed-forward pondered links as depicted in Figure 2. (Mellit, S. Shaari, H. Mekki, N. Khorissi, 2008). The BP learning process requires that the activation function is derivable. An example of a derivable function is the sigmoid.

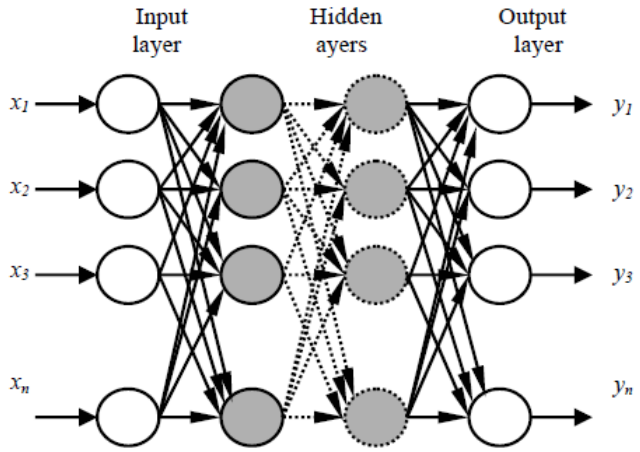

Figure 2. Schematic diagram of a multi-layer feed-forward neural network.

A sigmoid function has the following expression:

$$f(x) = \frac{1}{(1+e^{-x})} \tag{4}$$

This function has been implemented by means the developed custom *sfloat24* math library and using the CORDIC theory for the implementation of the exponential function.
Various methods exist to implement a trigonometric/hyperbolic math function. These include Taylor series; various curve fitting algorithms and the CORDIC (COordinate Rotation DIgital Computer) algorithm. The CORDIC algorithm often offers the most elegant solution to the problem, and it is astounding in its simplicity of implementation, efficiency and elegance (Samuel Ginsberg, 2002).
The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shift and adds. Usually the CORDIC theory is used to implement trigonometric functions. The implementation of the Sine and Cosine functions is quite immediate.

Applying this theory and assuming 12 step system, this yields 12 bits of accuracy in the final answer. Note that the $Konst = Cos\vartheta$, constant for a 12 step algorithm, is given by:

$$K = \prod_{i=0}^{11} \cos\theta_i = 0.6075$$

where:

$$\theta_i = \tan^{-1}(2^{-i})$$

At begin, off line stage, has been built up two small look up tables, in this case with a length of 12 elements, in the one is stored the angles in power of 2, starting from $\vartheta[0] = \frac{\pi}{4}$, and in the other the reciprocal of the power of 2. The second look up table is useful to avoid the divide by 2 operation during the execution of the algorithm (real time stage).
The first look up table is denoted *fract_teta[12],* where the stored angles are in radiants, and the second *reciprocal_pow2[12]*. In the following is depicted a fragment of C/C++ code to implement the sine and cosine functions.

Initialisation**:**
Set ϑ, '*teta_rad'* is the input angle, expressed in radians;
set *y=0;*
set *x=Konst=0.60725;*

Computation**:**
y=0.0;
dx=0.0;dy=0.0;

```
for(i=0;i<12;i++)
{
  dx=x*reciprocal_pow2[i];
  dy=y*reciprocal_pow2[i];
  dteta_rad=fract_teta[i];
  if(teta_rad>=0) d=-1.0; else d=1.0;
  x=x+d*dy;
  y=y-d*dx;
  teta_rad=teta_rad+d*dteta_rad;
}
```

The trigonometric functions of the desired angle are present in the x and y variable. In this implementation is assumed $-\frac{\pi}{2} < \vartheta < \frac{\pi}{2}$ in order to extend the computation for any value of ϑ it is sufficient to add few instructions, as well known in the literature.
In the case of the implementation of the hyperbolic function the expressions, using always the CORDIC theory, can be written as follow:

$$\vartheta_{i+1} = \vartheta_i - d_i \tanh^{-1}(2^{-i}) \tag{5}$$

$$\begin{aligned} x_{i+1} &= x_i + y_i d_i 2^{-i} \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ \vartheta_{i+1} &= \vartheta i - d_i \tanh^{-1}(2^{-i}) \end{aligned} \tag{6}$$

where $d_i = -1$ if $\vartheta_i < 0$, +1 otherwise, the integer i varies as described below.

The hyperbolic functions, considering the expressions (5) and (6), can be implemented in similar manner as the sine and cosine functions. In fact, the following code is quite similar:

Initialisation**:**
Set ϑ, '*teta_rad'* is an input variable;
set *y=0;*
set *x=Konst=7.5495e+011;*

Computation**:**
```
y=0.0;
dx=0.0;dy=0.0;
for(i=0;i<24;i++)
{
  dx=x*reciprocal_pow2[i];
  dy=y*reciprocal_pow2[i];
  dteta_rad=fract_teta[i];
  if(teta_rad>=0) d=-1.0; else d=1.0;
  x=x+d*dy;
  y=y+d*dx;
  teta_rad=teta_rad-d*dteta_rad;
}
```

reciprocal_pow2[i] is given by:

$$/*[1-2^{-13},1-2^{-12},1-2^{-11},1-2^{-10},1-2^{-9},1-2^{-8},1-2^{-7},1-2^{-6},1-2^{-5},1-2^{-4},1-2^{-3},1-2^{-2},$$
$$2^{-1},2^{-2},2^{-3},2^{-4},2^{-5},2^{-6},2^{-7},2^{-8},2^{-9},2^{-10},2^{-11},2^{-12}]*/$$

and *fract_teta[i]* is given by the inverse hyperbolic tangent of the above commented values.
The hyperbolic functions are present in the x and y variables. In this case the steps are 24 because the input variable can assume any value, the hyperbolic are not circular math functions.
At last the exponential functions can be easily derived from the hyperbolic functions as following:

$$e^{x} = \sinh(\vartheta) + \cosh(\vartheta) \quad e^{-x} = -(\sinh(\vartheta) + \cosh(\vartheta)) \tag{7}$$

3. FPGA Implementation – Simulation results –

It is possible, on the basis of expressions (5),(6) and (7), to build the sigmoid function. The code of the hyperbolic functions has been written in VHDL code, similar at above depicted C code.
The code has been implemented using the ALTERA® Quartus II 9.1 software. The *sfloat24* sigmoid activation function occupies only 44% of total logic elements of the used Cyclone® III EP3C25F324C8 device and occupying under 1% of the dedicated logic registers.
In the following is shown the obtained sigmoid function compared to the same function built with Simulink® blocks. It is important to underline that the Matlab® operates with double precision floating (64 bit) point numbers. The simulations have been performed using the ALTERA® DSP Builder tool, in particular using the HDL import block, as depicted in Figure 3.
Figure 4 shows the simulation results when the sampling time T_s has been fixed to a 2.56 μS, in this case the output error varies in the range ±0.006 about. Other simulations have been performed with different sampling times. In all the tests results that the entire system has a latency time at maximum of 6 clock cycles. Improvement of the overall performances could be reached by using a sample time less than 2.56 μS. For a common industrial application the used sample time produces good results.

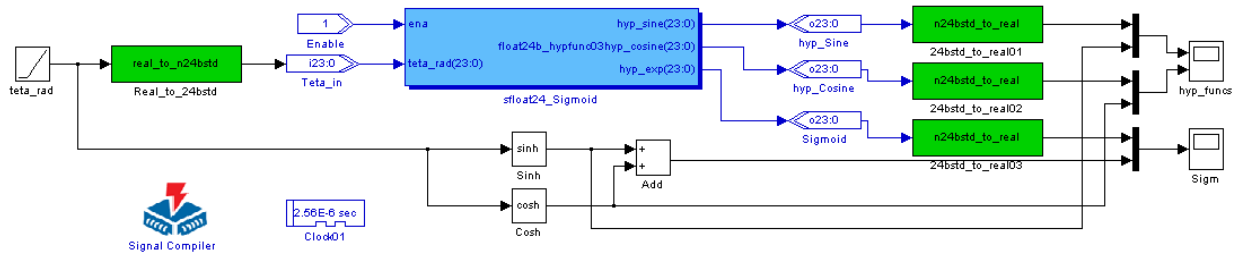

Figure 3. Hyperbolic and Sigmoid functions Simulink® simulation layout.

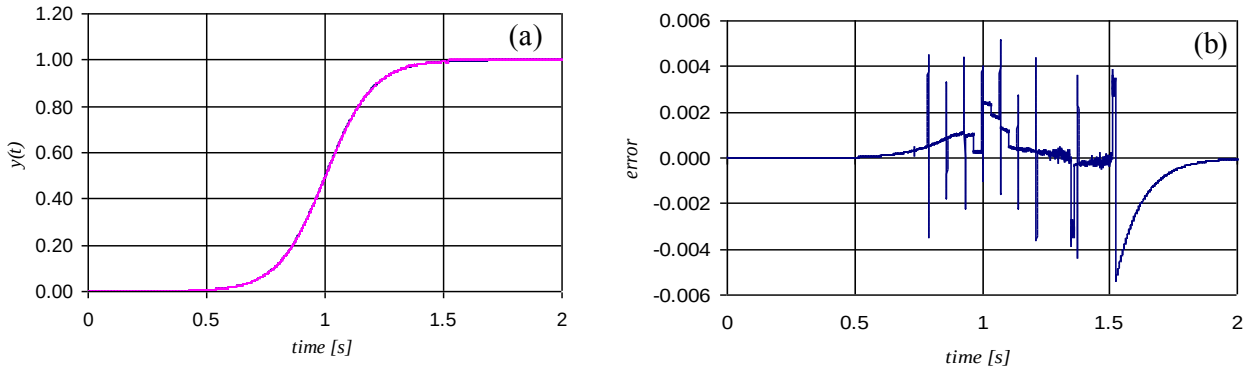

Figure 4. Simulation results – (a) Sigmoid activation function and (b) the absolute error between Sigmoid generated by the *sfloat24* math library and one generated by Simulink® math function block.

The system presents a stable performance comparing to the any external disturbance. In fact, FPGA$_S$ offer the stability of a typical digital implementation on a standard microprocessor and performances comparable to an analog implementation. Two additional routines allow to test the implemented ANN performances. The first routine converts a given floating point number in to *sfloat24* number layout format. The second one converts a *sfloat24* number to a double precision floating number.
These routine, due to their complexity, have been written in C/C++ language and implemented as S-Function as depicted in (M.C. Miglionico, F. Parillo, 2010).

4. Conclusions

The obtained results show the validity and the feasibility of implementing a Sigmoid activation function using either the custom *sfloat24* math library and the CORDIC theory. On the used FPGA device is possible only to implement a single artificial neuron. These results show that it is possible implement a full Artificial Neural Network similar the one shown in Figure 2. In this case a ALTERA® Stratix FPGA is required. The authors are working on an optimized version of the proposed algorithm, in order to reduce the area device occupation, in terms of logic elements, on any used FPGA. Furthermore, the speed or execution or latency of the ANN can be precisely controlled with the amount of reuse of *sfloat24* arithmetic elements.
Since on the more advanced families of FPGAs it is possible to implement a network with an interesting number of neurons working in parallel using only a single chip.
The implemented sigmoid module is highly accurate and can easily be reconfigured for any sigmoid slope and any desired error tolerance.

REFERENCES

M.A. Banuelos, j. Castillo Hernandez, S. Quintana Thierry, R. Damian Zamacoma, J. Valeriano Assem, R.E. Cervantes, R. Fuentes Conzalez, G. Calva Olmos, J.L. Perez Silva. (2003). Implementation of a Neuron Using FPGAS. Journal of Applied Research and Technology, Vol. I numero 003, 3 October 2003, Universitad Nacional Autonoma de Mexico Distrito Federal, Mexico, 248 - 255.

Bogdan M. Wilamonwski. (2009). Neural Network Architectures and Learning Algorithms. IEEE Industrial Electronics Magazine, December 2009.

Ray Andraka. (1998). *A survey of CORDIC algorithms for FPGA based computers*. In: International Symposium on Field Programmable Gate Arrays, Monterey, California, United States 1998 , 191 - 200.

M.C. Miglionico, F. Parillo. (2010). FPGA implementation of *sfloat24* digital PI. IEEE Conference PEDES 2010 Power India, New Delhi, 20-23 December 2010.

IEEE Standard for Binary Floating-Point Arithmetic. (1985). ANSI/IEEE 754 1985.

W. Kahan. (1996). Lecture notes on the Status of IEEE Standard 754 for Binary Floating Point Arithmetic. Electrical Engineering and Computer Science – University of California Berkeley CA 94720-1776, 31 May 1996.

C. Attaianese, F. Parillo, G. Tomasso. (2010). Dual Boost High Performances control strategy on a Power Factor Correction (PFC) implementation by using a 24 bit custom floating point library. Journal of Electrical Engineering [http://www.jee.ro], Vol. 10 edition 4, 23 December 2010.

Mellit, S. Shaari, H. Mekki, N. Khorissi. (2008). FPGA-based Artificial Neural Network for Prediction of Solar Radiation Data from Sunshine Duration and Air Temperature. Proceedings 2008 IEEE Region 8 International Conference on Computational technologies in Electrical and Electronic engineering. "SIBIRCON 2008", 118-123.

Samuel Ginsberg. (2002). Compact and Efficient Generation of Trigonometric Functions using a CORDIC algorithm. Cape Town, South Africa, January 2002.