BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 66, No. 5, 2018
DOI: 10.24425/124272

Hardware implementation of hyperbolic tangent and sigmoid activation functions

Z. HAJDUK*

Rzeszów University of Technology, ul. Powstańców Warszawy 12, 35-959 Rzeszów, Poland

Abstract. This paper presents the high accuracy hardware implementation of the hyperbolic tangent and sigmoid activation functions for artificial neural networks. A kind of a direct implementation of the functions in a few different versions is proposed and investigated both by software and hardware modeling. A single precision floating point arithmetic is applied. Apart from conventional design style with hardware description language coding, high level synthesis design techniques with the Matlab HDL coder and Xilinx Vivado HLS have also been investigated.

Key words: FPGA, hyperbolic tangent, sigmoid, floating point arithmetic.

1. Introduction

Artificial neural networks (ANNs) have found many applications in areas including pattern recognition, prediction, image processing, data filtering, features selection, optimization, control systems, etc. [1, 2]. ANNs can be implemented exercising software solutions (e.g. PC platforms, microprocessors systems) as well as hardware platforms, particularly field programmable gate arrays (FPGAs). Hardware implementation of ANNs may deliver a faster computing speed, which is related to the parallelism offered by hardware. Yet, due to a large number of computations the hardware realization of ANNs is more difficult than software, and it requires much more efforts on the part of the designer.

The most important, expensive and hard to implement part of any hardware realization of ANNs is the non-linear activation function of a neuron [3]. Commonly applied activation functions are the sigmoid and the hyperbolic tangent [4]. There are a number of papers considering a hardware implementation of activation functions. They differ in the accuracy obtained, applied approximation method, cost of implementation, type of arithmetic used (fixed or floating point), etc. A frequently used approximation method for the sigmoid and hyperbolic tangent functions is the piece-wise linear (PWL) interpolation, which divides the approximated function into numerous linear sections. This method has been applied in works [2, 4–7]. In [8] the hyperbolic tangent function has been realized exercising a look-up table (LUT) with 8192 elements and implemented using the LabView software and the CompactRIO hardware platform. A similar technique – a LUT with linear interpolation between LUT's points has been applied in [1]. A study of polynomial approximation of the hyperbolic tangent function, exercising Lagrange, Chebyshev and least square method, is presented in [9]. A Taylor series approximation is considered in [10]. The usage of the coordinate rotation digital computer (CORDIC) algorithm for implementation of activation functions is featured in [3]. An implementation based on the discrete cosine transform (DCT) interpolation is portrayed in [11]. Direct realization of the Gaussian type activation function for probabilistic neural networks is proposed in [12].

With the exception of [6, 9, 12], all of the papers mentioned above have considered fixed point arithmetic for implementation of the activation function. Some of the papers (e.g. [3, 9]) did not reveal any implementation details. The others (e.g. [4, 10, 12]) only presented a rough idea of how the proposed method has been implemented (only a simplified block diagram of a data-path has been featured).

In this paper a kind of a direct implementation of the hyperbolic tangent and sigmoid activation functions is proposed. In this approach, the main implementation difficulty is shifted to the approximation of the exponent function. Applying a LUT with either the McLaurin series or Padé polynomials is proposed in order to accomplish the approximation. Contrary to the traditional approach for FPGA design, which is based on fixed point arithmetic [13], a single precision floating point arithmetic is used. Applying this kind of arithmetic for the implementation of the activation function and, consequently, a whole ANN, allows an ANN's learning process to be conducted by exercising a PC software (e.g. Matlab) and then the calculated weights can be directly transferred to the FPGA. However, obtaining consistent results between PC-realized and trained ANN, and its FPGA-based counterpart, requires a high accuracy of the activation function calculation. Therefore, the important goal of the proposed solution was to attain as accurate of a calculation of the activation function as possible.

Unlike many other already published works, the paper is not concentrated on the mathematical side of the activation function realization, but it is focused on its actual digital implementation (both data-path and detailed control-path have been presented). The basic idea of the proposed implementation method has already been briefly portrayed in [14]. This paper describes

*e-mail: zhajduk@kia.prz.edu.pl

Manuscript submitted 2017-06-29, revised 2017-11-10, initially accepted for publication 2017-12-04, published in October 2018.

the proposed method in details introducing numerous new elements, such as considering 10 slightly different implementation versions, analyzing the absolute and relative errors, analyzing the FPGA resources requirement and calculation speed of these versions, and utilizing the software modeling for the estimation of the properties of the versions proposed. Additionally, other design techniques for the activation function realization, such as the usage of the Matlab HDL coder and Xilinx Vivado HLS tools, have also been investigated.

The rest of the paper is structured as follows. Section 2 describes a general idea of the proposed implementation method and presents some accuracy issues related to the developed software model of the proposed algorithm. Details of the FPGA implementation of the method along with the obtained accuracy, calculation speed and resources requirement are included in Section 3. Section 4 discusses, in turn, the feasibility of the usage of high level synthesis design tools, whereas Section 5 concludes the paper.

2. Realization method

The sigmoid activation function is given by the following formula:

$$S(x) = \frac{1}{1+e^{-x}}, \tag{1}$$

whereas the hyperbolic tangent function is expressed by the equation [4]:

$$T(x) = \frac{e^{2x}-1}{e^{2x}+1} = \frac{2}{1+e^{-2x}} - 1. \tag{2}$$

Direct realization of equation (1) or (2) requires the calculation of the exponent function, which is not straightforward. An interpolation of the exponent function by means of a McLaurin series and Padé polynomials has been proposed and investigated.

2.1. McLaurin interpolation. The McLaurin interpolation of the exponent function is given by the following formula:

$$e^x \approx 1 + \sum_{i=1}^{N} \frac{x^i}{i!}, \tag{3}$$

where N denotes the degree of the polynomial. Equation (3) can also be rewritten in a different form, more convenient for a practical implementation, e.g. for $N = 7$ we obtain:

$$e^x \approx 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(1 + \frac{x}{4}\left(1 + \frac{x}{5}\left(1 + \frac{x}{6}\left(1 + \frac{x}{7}\right)\right)\right)\right)\right)\right), \tag{4}$$

It is important to note that equations (3) and (4) are only valid for the exponent arguments of the function limited to a narrow interval [10], i.e. $x \in (-1, 1)$. However, for a wider range of the arguments we can use a simple formula:

$$e^x = e^{p+f} = e^p \cdot e^f, \tag{5}$$

where $p + f = x$, p is an integer part of x, whereas f denotes a fraction, $f \in (-1, 1)$. The expression e^f can be directly calculated using equation (3), while e^p may derive from a look-up table. The number of elements of a LUT requires, however, a separate consideration. We can note that either the sigmoid or hyperbolic tangent values of the function become constant for arguments $|x| \geq M$, where M is an unsigned integer value. Since the exponent values of the function for both positive and negative arguments need to be considered, the size of a LUT is equal to $2(M-1)$. The size can be reduced to M $-$ 1, however, if the activation symmetry feature of the function is exploited.

Having solved the problem of the exponent function approximation, the sigmoid and hyperbolic tangent function can be directly calculated using equations (1) or (2). Yet, the values of the M and N parameters must be determined first. In order to do this, the proposed algorithm has been modeled using the Matlab software as well as the C programming language. The C language turned out to be more convenient for the sake of the existence of an embedded single precision floating point data type (using this data type in Matlab is not as simple as applying the C language) and a lower calculation time. The modeled function has been sampled using 1E6 points, equally spaced within the interval $[-10, 10]$. The values returned by the software modeled function has been compared, in terms of relative and absolute errors, with the results originated from the calculation of equations (1) or (2) where the exponent function came from the C compiler library or Matlab set of embedded functions. The errors have been obtained using the following formulas:

$$\begin{aligned}
E^A_{\max} &= \max_{i=0,\ldots,10^6-1} \left|y(x_i) - \hat{y}(x_i)\right|, \\
E^A_{avg} &= \left(\sum_{i=0}^{10^6-1} \left|y(x_i) - \hat{y}(x_i)\right|\right) \cdot 10^{-6}, \\
E^R_{\max} &= \max_{i=0,\ldots,10^6-1} \left|\frac{y(x_i) - \hat{y}(x_i)}{y(x_i)}\right|, \\
E^R_{avg} &= \left(\sum_{i=0}^{10^6-1} \left|\frac{y(x_i) - \hat{y}(x_i)}{y(x_i)}\right|\right) \cdot 10^{-6},
\end{aligned} \tag{6}$$

where $E^A_{\max}$, E^A_{avg}, $E^R_{\max}$, E^R_{avg}, stands for the maximum absolute error, average absolute error, maximum relative error and average relative error respectively, whereas $y(x_i)$ means the accurate function value and $\hat{y}(x_i)$ is the function value obtained by means of the proposed realization method.

Table 1 and Table 2 show the absolute and relative errors of the proposed direct realization of the hyperbolic tangent function, acquired for different values of the two parameters. It can be noted that increasing the values beyond $N = 10$ and $M = 18$ yields very little impact on the improvements of the accuracy. Therefore, these values were admitted for the basic version of the hardware realization of the hyperbolic tangent function.

Table 1
Errors of the hyperbolic tangent function calculations for different values of the N parameter ($M = 18$)

N	Absolute error		Relative error	
	Max	*Avg.*	*Max*	*Avg.*
6	1.882E-04	1.321E-06	4.073E-04	2.448E-06
7	2.396E-05	1.482E-07	5.185E-05	2.726E-07
8	2.861E-06	1.565E-08	2.861E-05	2.919E-08
9	4.768E-07	2.420E-09	2.861E-05	5.211E-09
10	2.384E-07	1.482E-09	2.861E-05	3.501E-09
11	2.384E-07	1.473E-09	2.861E-05	3.488E-09
12	2.384E-07	1.475E-09	2.861E-05	3.492E-09

Table 2
Errors of the hyperbolic tangent function calculation for different values of the M parameter ($N = 10$)

M	Absolute error		Relative error	
	Max	*Avg.*	*Max*	*Avg.*
15	7.153E-07	2.929E-08	2.861E-05	3.131E-08
16	2.384E-07	1.034E-08	2.861E-05	1.236E-08
17	2.384E-07	2.970E-09	2.861E-05	4.989E-09
18	2.384E-07	1.482E-09	2.861E-05	3.501E-09
19	2.384E-07	1.449E-09	2.861E-05	3.469E-09

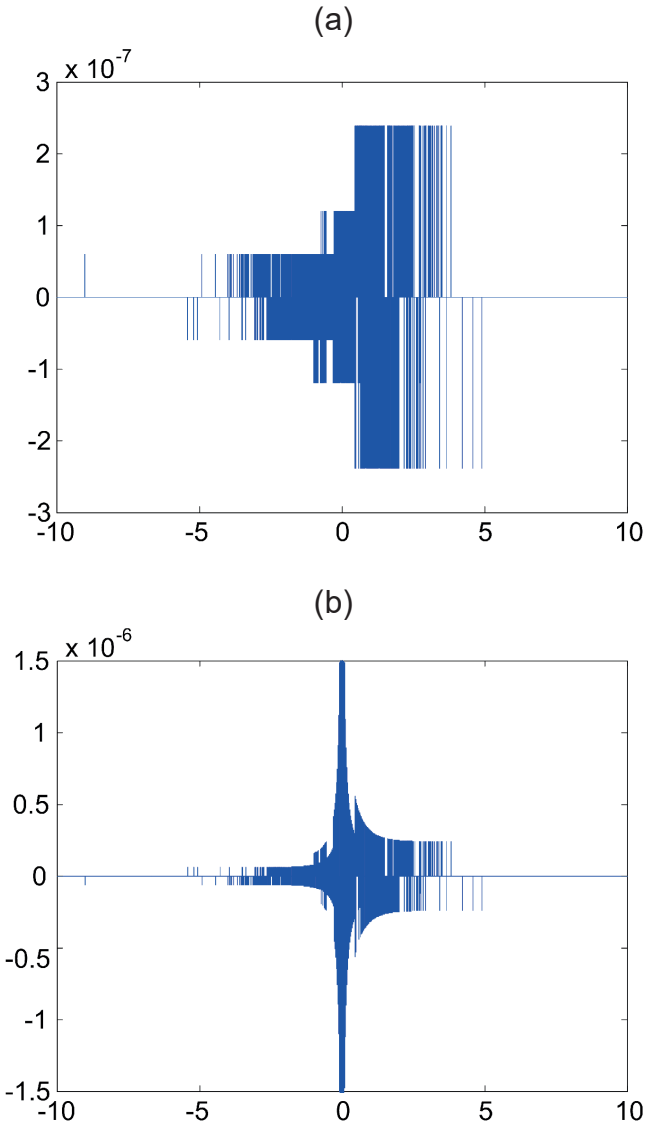

Fig. 1. Distribution of absolute (a) and relative (b) errors for the software modeled hyperbolic tangent function ($N = 10$, $M = 18$)

A distribution of absolute and relative errors for the proposed realization method of the hyperbolic tangent function is presented in Fig. 1. As seen, the values of the absolute errors seem to be strongly quantized with the lowest level of quantization reaching to approx. 0.7E-7 (actually, it slightly fluctuates from 6.801E-08 to 7.192E-08 with an average value of 6.979E-08). The relative error values are significantly higher for the arguments of the function coming from a narrow interval around the beginning of the coordinate system. Therefore, for Fig. 1b, the y-axis has been limited to the interval [–1.5E-6, 1.5E-6], whereas the full range of the function's values include [–2.86E-5, 7.82E-6].

It can also be noted that the values of the absolute as well as relative errors are significantly higher for the positive values of the arguments. Since the hyperbolic tangent function has a symmetry point located at the beginning of the coordinate system, the modeled values of the function can be calculated only for negative arguments and then properly adjusted by changing the sign for positive arguments. The results of the calculations carried out in this way are presented in Table 3 (option A) and Fig. 2. As expected, the maximum absolute error has slightly dropped, yet the maximum relative error has significantly increased by almost two orders of magnitude. The average error values have also significantly increased, which can be observed in Fig. 2, particularly for positive argument values.

Table 3
Errors of the hyperbolic tangent function calculation with the usage of the symmetry feature ($N = 10$, $M = 18$)

Option	Absolute error		Relative error	
	Max	*Avg.*	*Max*	*Avg.*
A	1.788E-07	2.744E-08	1.776E-03	4.903E-08
B	2.384E-07	2.771E-08	2.252E-03	5.020E-08

It is important to note that Fig. 2b shows only relative error values restricted to the interval [–1.5E-6, 1.5E-6], whereas the range of the values includes a much higher interval [–1.8E-3, 3.2E-4]. As Fig. 2b shows, relative errors are very high within a narrow interval, located close to the symmetry point. A similar situation also takes place when the function's values are calculated only for positive arguments and analogously adjusted for negative arguments, which is documented in Table 3 option B.

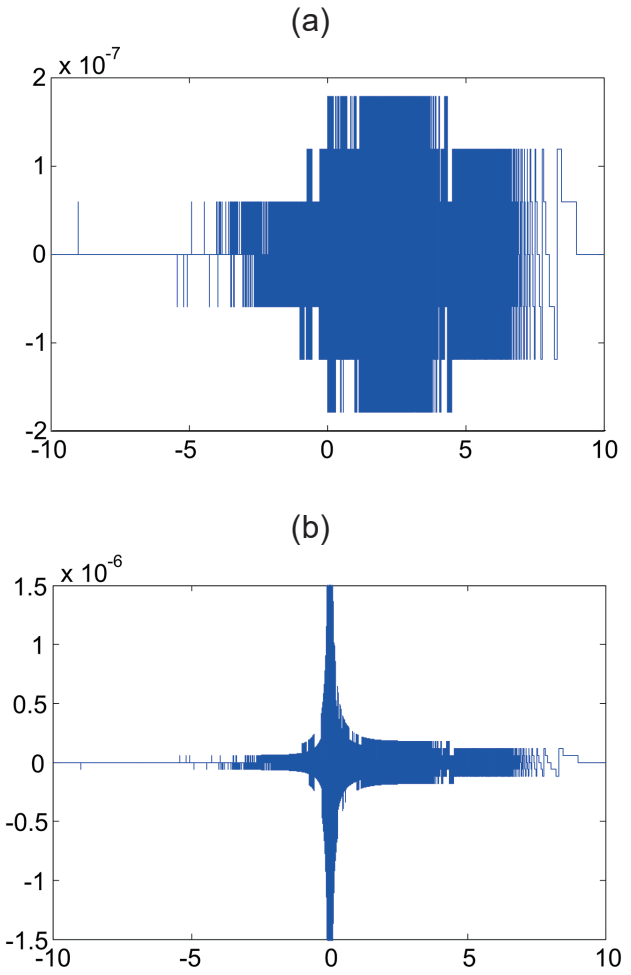

Fig. 2. Distribution of absolute (a) and relative (b) errors for the hyperbolic tangent function with the usage of a symmetry feature ($N = 10$, $M = 18$)

Another line of investigation has regarded a possibility of the usage of a lower degree of the McLaurin series (lower N parameter), keeping a similar accuracy at the same time. This would allow the reduction of the overall calculations time. The decrease of the N parameter value can be attained, for example, by restricting the interval of the exponent function argument from (–1, 1) to (–0.5, 0.5). Thus, the expression e^f from equation (5) can be replaced by the following product:

$$e^f = e^g \cdot e^h, \tag{7}$$

where

$$g = \begin{cases} f \;\Leftrightarrow\; |f| < 0.5 \\ f - 0.5 \;\Leftrightarrow\; f \geq 0.5 \\ f + 0.5 \;\Leftrightarrow\; f \leq 0.5 \end{cases}, \quad h = \begin{cases} 0 \;\Leftrightarrow\; |f| < 0.5 \\ -0.5 \;\Leftrightarrow\; f \geq 0.5 \\ 0.5 \;\Leftrightarrow\; f \leq -0.5 \end{cases}. \tag{8}$$

The constant values of the expression e^h from equation (7) can be stored in a another LUT, whereas e^g would be calculated using equation (3).

The results of the hyperbolic tangent function calculation, carried out in this way, are presented in Table 4. It can be noted that a quite good accuracy is attained for $N = 7$, while using a wider interval of the $N = 10$ argument of the function was needed. The maximum relative error is also lower than for $N = 10$. However, apart from the usage of a second, two-element LUT, the reduction of the polynomial degree has been attained at the cost of performing additional subtraction and multiplication operations. Moreover, these operations cannot be accomplished in parallel, as far as hardware implementation is concerned.

Table 4
Errors for the hyperbolic tangent function calculations with restricted range of exponent's argument and different N values ($M = 18$)

N	**Absolute error**		**Relative error**	
	Max	***Avg.***	***Max***	***Avg.***
5	1.311E-05	6.598E-08	2.839E-05	1.180E-07
6	1.073E-06	5.448E-09	2.334e-06	1.003E-08
7	2.384E-07	1.631E-09	1.109E-06	3.269E-09
8	2.384E-07	1.525E-09	1.109E-06	3.081E-09
9	2.384E-07	1.520E-09	1.109E-06	3.073E-09

Similar evaluations have also been carried out for the sigmoid activation function. It turned out that for the following values of the parameters $N = 10$, $M = 11$ and the interval [–10, 10], the maximum absolute and relative errors are smaller than for the hyperbolic tangent function, and amount to 1.192E-07 and 2.728E-07 respectively. Contrary to the hyperbolic tangent, the usage of the symmetry feature of the sigmoid function does not lead to excessive values of relative errors; however, it requires performing an additional subtraction operation.

2.2. Padé approximation. The Padé approximation of order (m, n) of the exponent function can be expressed in the fractional form [15]:

$$e^{-x} \approx \frac{P_m(x)}{Q_n(x)}, \tag{9}$$

where $P_m(x)$ and $Q_n(x)$ are the following polynomials:

$$P_m(x) = \sum_{k=0}^{m} \frac{(m+n-k)!\,m!}{(m+n)!\,k!\,(n-k)!}(-x)^k,$$
$$Q_n(x) = \sum_{k=0}^{n} \frac{(m+n-k)!\,n!}{(m+n)!\,k!\,(n-k)!}(-x)^k. \tag{10}$$

It is of note that for $n = m$ we have $P_n(x) = Q_n(-x)$, which means that the numerator and denominator coefficients have the same absolute value. This fact allows the number of multiplication operations to be significantly reduced which is an important feature, particularly in terms of hardware implementation.

Table 5 shows the absolute and relative errors of the direct realization of the hyperbolic tangent activation function with the Padé approximation of the exponent function. Apart from the presented results, other evaluations for different orders of the numerator and denominator have also been carried out. It turned out, however, that the approximation errors are significantly higher in these cases. Therefore, these results were not included in the table.

Table 5
Errors of the hyperbolic tangent function calculation using Padé approximation of the exponent

Approx. order	Absolute error		Relative error	
	Max	*Avg.*	*Max*	*Avg.*
(3,3)	4.172E-06	5.207E-08	2.252E-03	1.094E-07
(4,4)	2.384E-07	1.878E-09	2.252E-03	1.436E-08
(5,5)	2.384E-07	1.811E-09	3.623E-03	1.866E-08

The results from Table 5 suggest that the increase of the approximation order beyond the (4.4) value does not bring the decrease of the approximation errors. Thus, this approximation order and the following formula of the exponent function calculation have been admitted for further consideration:

$$e^{-x} \approx \frac{1680 - 840x + 180x^2 - 20x^3 + x^4}{1680 - 840x + 180x^2 - 20x^3 + x^4}. \tag{11}$$

It is important to note that, similar to the McLaurin interpolation, Padé approximation is also valid within a narrow interval of the function's argument, i.e. $x \in (-1, 1)$. Therefore, equation (5) and a LUT are still needed for a wider range of the exponent arguments of the function. The number of the elements of LUT should also be the same as for the McLaurin interpolation, i.e. $M = 18$. This value was used for the errors calculations from Table 5.

The usage of the symmetry feature for the Padé approximation has also been examined. Yet, contrary to the McLaurin interpolation, it does not bring an improvement of the accuracy. It is also worth noting that for the Padé approximation the maximum relative error is higher than for the McLaurin interpolation.

The Padé approximation of order (4,4) applied for the direct realization of the sigmoid activation function brings the following maximum absolute and relative errors: 1.192E-07 and 2.917E-07 respectively. Virtually, the errors are the same as for the McLaurin interpolation.

3. FPGA implementation

Once the software has been modeled and the algorithm of the activation function realization has been tested, the solution can be implemented in hardware, which is described in the following subsections.

3.1. Activation function module. An FPGA implementation of the method described in the previous section requires the usage of floating point (FP) components, accomplishing three basic arithmetic operations: the multiplication, addition and division. These components may come from FPGA vendors as intellectual property (IP) cores or they can be specially designed, e.g. such as in [16]. Having the FP components, the general block diagram of the hardware module, performing calculations of the activation function, can be very simple as it is shown in Fig. 3.

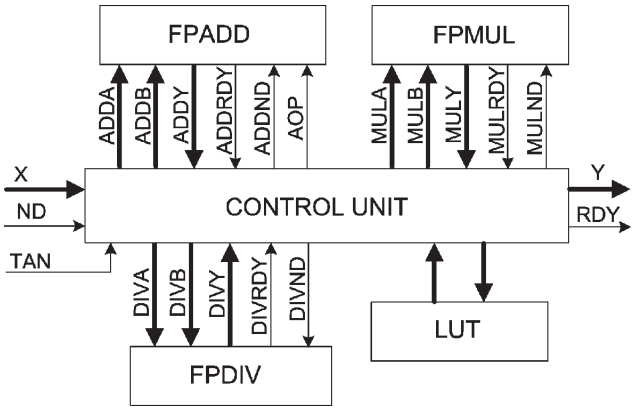

Fig. 3. Block diagram of the activation function module

Apart from the FP adder (FPADD), multiplier (FPMUL), divider (FPDIV) blocks, it contains a LUT table (physically implemented as a distributed ROM) for the e^p constant values from equation (5), and a control unit (CU). Each of the FP blocks has three 32-bit data buses dubbed *xA* (the first operand), *xB* (the second operand), *xY* (the operation result), and two control signals named *xND* (assertion of new data) and *xRDY* (completion of the operation), where here the *x* prefix denotes the type of the operation (ADD, MUL, DIV). The AOP signal for FPADD block determines when the addition (AOP = 0) or subtraction (AOP = 1) operation should be performed. The control unit has two external data buses: *X* for the activation argument of the function and *Y* for calculations result, and three control signals: *ND* (new data are present), *RDY* (the result is ready) and *TAN* (the type of the activation function).

3.2. Control unit specification. The control unit from Fig. 3 plays a pivotal role in the calculations of the activation func-

tion. The detailed algorithmic state machine (ASM) diagram, describing the operations performed by the control unit for the McLaurin interpolation of the exponent function, is presented in Fig. 4. The signals names denoted using capital letters relate to external input/output buses and signals, whereas non-capital letters are used to mark internal variables (registers).

The diagram also applies syntax elements similar to those used in the Verilog hardware description language (HDL). For example the expression {~X[31], X[30:0]} denotes the concatenation of the inverted 31-st bit of the X vector and the partial selection of the bits for 30 down to 0 of the same vector. It is also important to recall a binary representation of floating point numbers. The representation is as follows:

$$(-1)^s \times 1.m \times 2^{e\text{-}bias}, \tag{12}$$

where s is the sign bit of a number (the most significant bit of the binary representation), m is the mantissa and e is the exponent. For single precision floating point numbers, the bias amounts to 127, the mantissa has 23 bits whereas the exponent

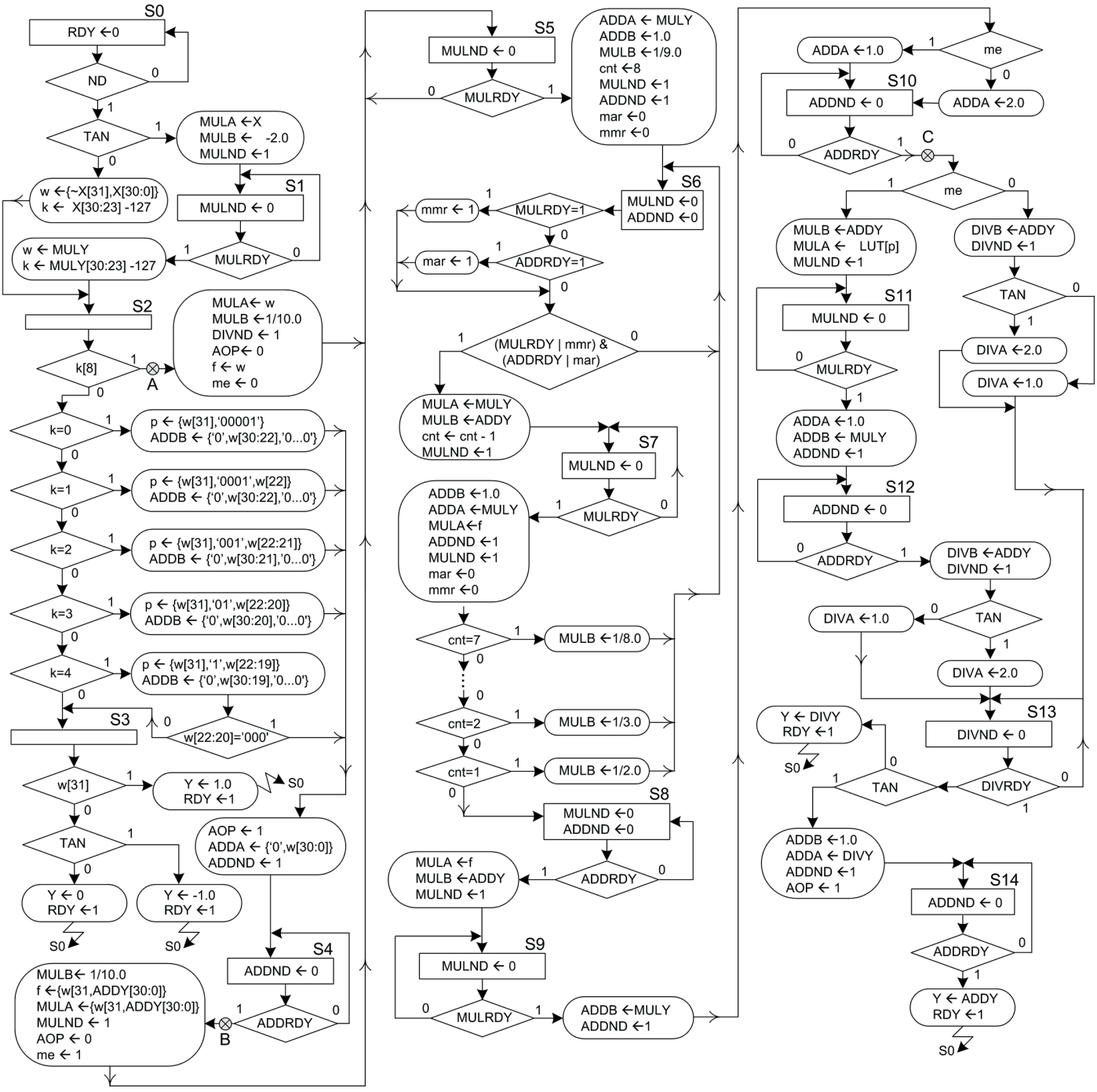

Fig. 4. Basic version of the ASM diagram describing the control unit operations for the McLaurin interpolation

counts 8 bits. Thus, the operation w <– {∼X[31], X[30:0]}, performed in the state S0 of the ASM, means that the activation function's argument with inverted sign is assigned to the internal 32 bits *w* variable.

The ASM from Fig. 4 considers both types of activation functions. The external signal TAN determines whether the sigmoid ($TAN = 0$) or hyperbolic tangent ($TAN = 1$) function should be calculated. If the hyperbolic tangent function is taken into account then the function's argument should be multiplied by –2.0, which is performed in the states S0 and S1 of the ASM. The next step of the proposed implementation method includes the determination of an integer part (and a little later, a fractional part) of the exponent function's argument, and the determination whether the integer part is less than a selected constant value (the *M* parameter from the previous section). In order to accomplish this, the value of the expression *e-bias* from notation (5) is calculated and assigned to the 9 bits *k* variable. If the calculation result is negative than this means that the absolute value of the exponent function's argument is less than 1.0, and the exponent can be directly calculated using expression (3) or (4). Otherwise, the value of the *k* variable is tested and the integer part (the 6 bits *p* variable) is constructed accordingly, exercising specific features of the floating point notation. Along with the determination of the *p* variable, the floating point representation of an integer part is also constructed and asserted to the second input (ADDB) of the FP adder block. If the integer part of the *w* variable is less than 18, which is the case when $k < 4$ or $k = 4$ and the three most significant bits of the mantissa of the *w* variable are not set, then the fraction part is calculated by subtracting the floating point representation of the integer part from the *w* variable. The subtraction result is assigned to the 32 bits *f* variable subsequently. When the integer part is greater or equal to 18, the ASM goes to state S3. In this state the final result of the activation function calculation (the *Y* variable) is determined, depending on the sign of the *w* variable and the actual type of the activation function.

After the determination of the fractional part (the *f* variable), the exponent function can be calculated using equation (3) or (4). The ASM from Fig. 4 exercises equation (4) with 10 terms of the McLaurin series. The calculations are conducted through states S5 ... S10 of the ASM. Instead of performing divisions from equation (4), multiplications by reciprocated values are accomplished. This ensures faster execution of the algorithm – the FP division block requires considerable more pipeline stages (more clock cycles) than the FP multiplier block, in order to achieve similar clock frequency (see subsection 3.4).

It is also important to note that some arithmetic operations are performed in parallel. When the multiplication of the exponent function's argument (the *f* variable) by a constant value is accomplished, the addition of the 1.0 value to a previous intermediate calculations result is performed at the same time. This shortens the overall calculations time. Yet, it should be noted that two operations in parallel are performed alternatively with a single operation at a time, i.e., the multiplication and addition are accomplished in state S6 whereas only the addition is performed in state S7.

After completion of the exponent function calculations for the fractional part as an argument (the e^f expression from equation (5)), the calculations result should be multiplied by a constant value, which represents the value of the exponent function for the integer part as an argument (the e^p expression from equation (5)). The multiplication is not necessary when the integer part is equal to 0 (the 1 bit *me* variable is also equal to zero in this case). The mentioned operation is accomplished in states S10 and S11 of the ASM. Further operations include the addition of the 1.0 constant value to the overall result of the exponent function calculations (only if the integer part differs from zero, otherwise the constant is added in state S9) as well as division and subtraction, according to equation (1) or (2).

The ASM from Fig. 4 exercises equation (4) for which only two operations are performed concurrently. Yet, the calculations can be rearranged using equation (3), which allows the utilization of all three arithmetic blocks of the control unit at the same time. The altered fragment of the ASM, performing calculations in this way, is featured in Fig. 5. The fragment can be integrated with the ASM from Fig. 4 by replacing the part of the ASM marked with the A, B and C symbols.

The fragment from Fig. 5 seems to have a simpler architecture than the analogous section from Fig. 4. It can be proved,

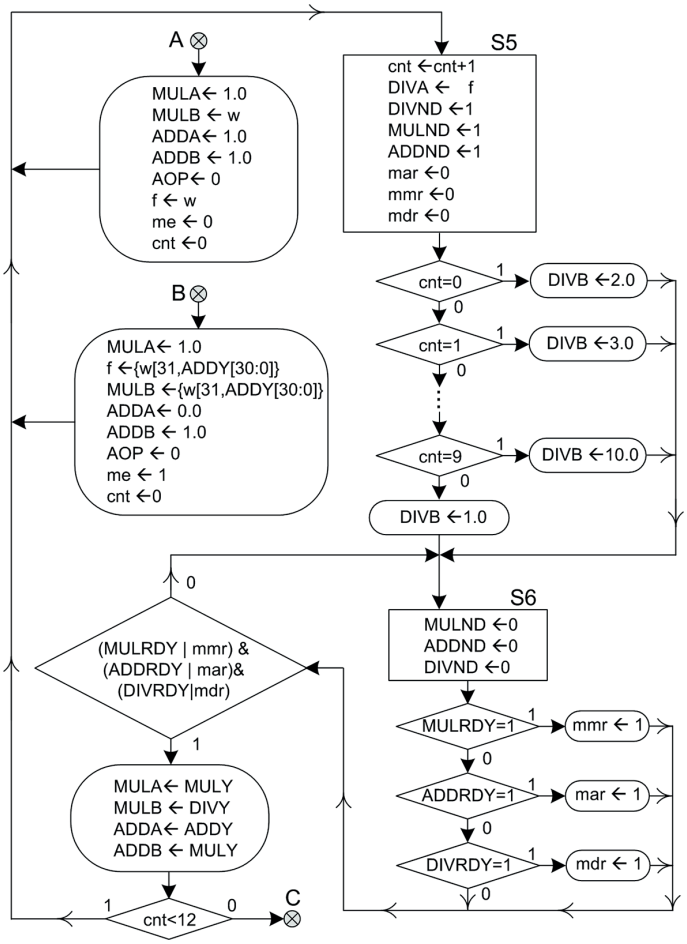

Fig. 5. Modification of the ASM from Fig. 4 allowing the calculation of the exponent function using equation (3)

however, that the fragment correctly implements equation (3) for $N = 10$. Since the divider block requires more clock cycles to complete calculations than the multiplier and adder, it would be more appropriate to use an additional multiplier block instead of the divider. This increases the requirement for FPGA resources, but enables a remarkable shortening in the overall calculations time.

All of the previous consideration has regarded the McLaurin interpolation of the exponent function. As far as the Padé approximation is concerned, it can be applied analogously – the part of the ASM from Fig. 4 comprising of states S5 ... S10 needs to be replaced by an ASM implementing equation (11). However, applying the additional FP multiplier and adder would allow more operations to be performed simultaneously. This, in turn, would lead to the reduction of the calculations time. The calculation sequence involving parallel computation of the Padé polynomials with the usage of two FP multipliers and two adders describes the ASM diagram depicted in Fig. 6. In order to fully describe the calculation of the sigmoid and hyperbolic tangent activation function, the ASM from Fig. 6 should be connected to the points marked with the A and B symbols on the ASM from Fig. 4, replacing the remaining part of the basic version of the ASM. The architectures of the operational blocks dubbed SUB1, SUB2 and SUB3 on the ASM from Fig. 6a are

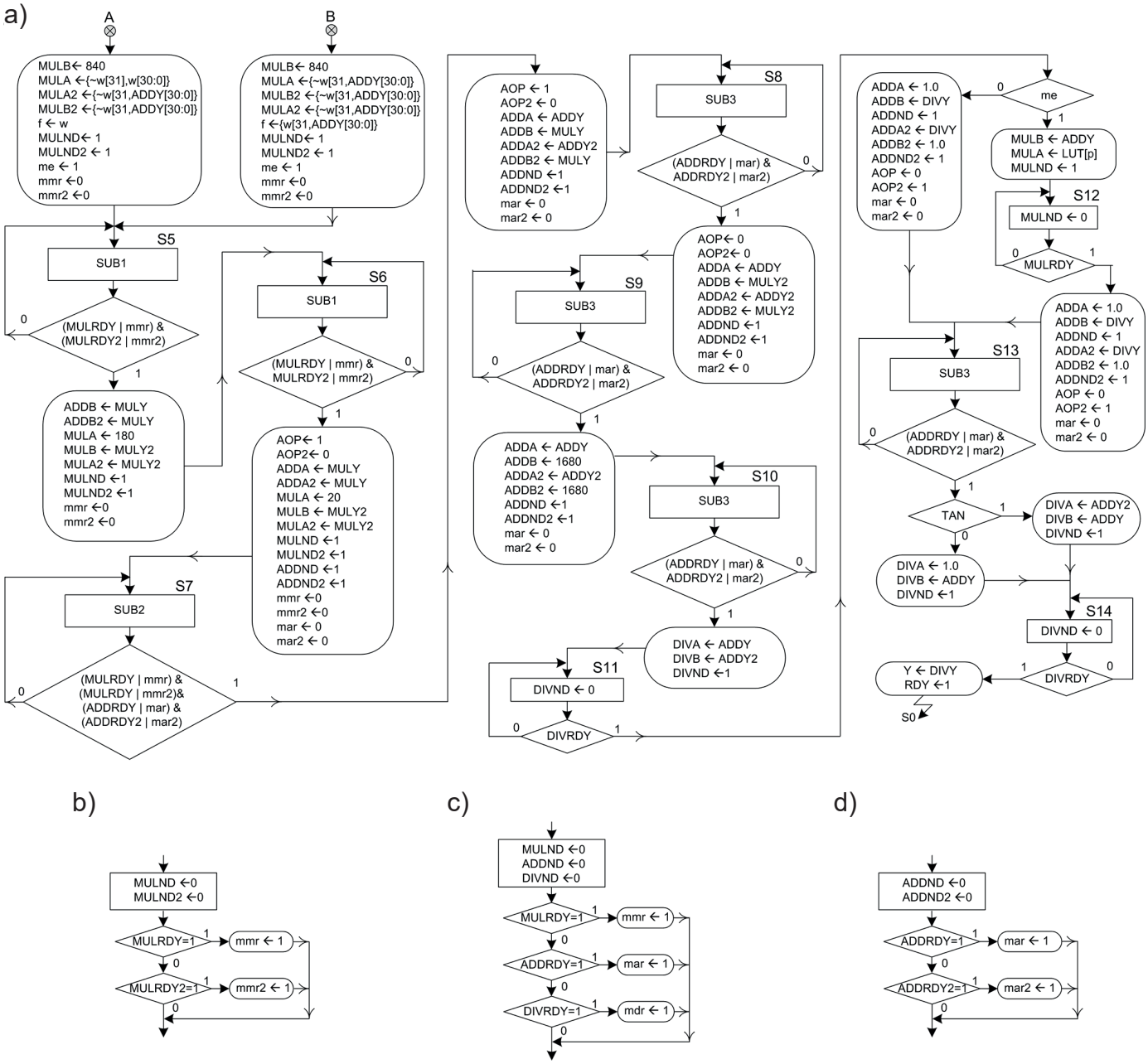

Fig. 6. Continuation of the ASM from Fig. 4 for the Padé approximation of the exponent (a), and architectures of the SUB1 (b), SUB2 (c) and SUB3 (d) operational blocks

presented in Fig. 6b, 6c and 6d respectively. These blocks deal with the auxiliary flags (e.g. *mmr*, *mar*) which store the value of the operation completion signals coming from the arithmetic blocks (the operation completion signals are only set within a single clock cycle, therefore they should be memorized for the detection of the completion of parallel computations performed by more than one arithmetic block).

The general idea of the paralleled computations described by the ASM from Fig. 6 relies on the parallel calculation of the numerator and denominator from equation (11). The primary FP multiplier deals with the multiplication by subsequent coefficients of the Padé polynomials, whereas the secondary FP multiplier, involving the data and control buses whose names have the "2" suffix, calculates the subsequent powers of the exponent argument of the function. The primary FP adder calculates in turn the value of the numerator from equation (11), whereas at the same time, the secondary FP multiplier calculates the denominator value.

It is important to note that, according to the ASM from Fig. 6, the 1680 constant value is added at the end of the calculation sequence of the value of the numerator and denominator. This addition could also be accomplished in earlier stages of the calculations, namely in state S5 where it could be performed in parallel with the multiplication. In this case, however, due to cumulative rounding errors, the overall maximum absolute error of the activation function calculation noticeably increases. Therefore, in order to match the errors level from Table 5, the calculation sequence described by the ASM from Fig. 6 should be applied.

It is also important to note that for the hyperbolic tangent activation function calculation, the ASM from Fig. 6 uses the formula with two exponent functions – the left hand side of the equation (2). Since the values of the numerator and denominator can be calculated simultaneously by two FP adders, the usage of the formula with two exponents allows the calculations time to be slightly decreased. It requires, however, an introduction of small amendment to state S0 of the ASM from Fig. 4 – instead of the negative value, the positive 2.0 constant should be assigned to the MULB variable.

3.3. Accuracy of the implementation. In order to verify the accuracy of the proposed implementation method of the activation function, the ASM from Fig. 4 has been described using Verilog hardware description language and implemented in Xilinx FPGAs. IP cores from Xilinx Core Generator software have been used for the FP arithmetic blocks implementation from Fig. 3. The accuracy of the activation function implementation has been tested in two ways: by simulations and using an FPGA board. The simulations for 1E6 points took a very long time (more than 11 hours for the Xilinx ISim simulator and a PC with an Core-i7 CPU @3.1GHz), thus most of the experiments have been conducted exercising a hardware platform. As an FPGA board, the main module (with Xilinx Spartan-6 XC6SLX100 chip) of the previously developed multiprocessor programmable controller [17] has been used. Apart from the activation function block, described by the ASM from Fig. 4, the implementation also included a specially designed communication module, responsible for PC communication. An architecture of the communication module was based on the similar module, dedicated for the P1-TS fuzzy system [16]. The module uses a simple RS-232 interface with 115.2 kbit/s baud rate. A dedicated PC application, written in Visual C++, has also been developed for sending data to the FPGA board and analyzing results.

The obtained accuracy of the FPGA implementation of the hyperbolic tangent and sigmoid functions, calculated for 1E6 points equally spaced within the [–10, 10] interval, is presented in Table 6 and 7. Besides the basic version of the functions implementation, described by the ASM from Fig. 4, a few of its modifications have been considered as well. The versions are as follows: A – basic version; B – the symmetry feature are used (negative arguments are calculated); C – restricted range of argument values, described by equations (6) and (7) are used, $N = 7$; D – restricted arguments as in the C version with the symmetry feature are exploited; E – the exponent is calculated using the ASM form Fig. 5; F – the same as E, but instead of the divider block, an additional multiplier is used; G – the

Table 6
Accuracy of the hyperbolic tangent function implementation

Version	*Absolute error*		*Relative error*	
	Max	***Avg.***	***Max***	***Avg.***
A	2.384E-07	1.728E-09	3.623E-03	2.414E-08
B	1.788E-07	2.758E-08	3.623E-03	6.102E-08
C	2.384E-07	1.732E-09	3.623E-03	2.416E-08
D	2.384E-07	2.790E-08	3.623E-03	6.173E-08
E	3.576E-07	4.136E-09	3.623E-03	4.067E-08
F	3.576E-07	4.134E-09	3.623E-03	4.067E-08
G	2.384E-07	2.905E-08	3.623E-03	7.270E-08
H	2.384E-07	2.903E-08	3.623E-03	7.247E-08
I	2.384E-07	3.341E-09	2.252E-03	2.269E-08
J	2.384E-07	3.997E-08	8.326E-03	8.842E-08

Table 7
Accuracy of the sigmoid function implementation

Version	**Absolute error**		**Relative error**	
	Max	***Avg.***	***Max***	***Avg.***
A	1.192E-07	1.706E-09	2.728E-07	1.603E-08
B	1.192E-07	1.453E-08	2.728E-07	2.96E-008
C	1.192E-07	1.699e-09	2.955E-07	1.766e-08
D	1.192E-07	1.486E-08	2.955E-07	3.305E-08
E	2.384E-07	4.166E-09	4.723E-07	3.040E-08
F	2.384E-07	4.136E-09	4.723E-07	3.036E-08
G	1.788E-07	1.596E-08	4.723E-07	4.248E-08
H	1.788E-07	1.602E-08	3.756E-07	4.303E-08
I	1.192E-07	3.268E-09	4.512E-07	3.428E-08
J	1.192E-07	3.268E-09	4.512E-07	3.428E-08

symmetry feature is added to the F version; H – restricted range of argument values is added to the G version, N=7; I – the Padé approximation of the exponent is introduced (single FP multiplier and single FP adder is exploited); J – the Padé approximation described by the ASM from Fig. 6 is applied (two FP multipliers and two FP adders are involved). For all of the versions the M parameter, related to the number of elements of LUT, has been set to 18.

As seen, the lowest maximum absolute error for the hyperbolic tangent function amounts to 1.788E-7 and is obtained for the basic version with the symmetry feature exploited. The error for the sigmoid function is slightly smaller (1.192E-7) and does not change within the versions, which exploit equation (4) or (11) for the exponent function calculation. However, we can notice that the maximum relative error for the hyperbolic tangent function with the McLaurin interpolation of the exponent is considerably higher than the one obtained from the software model (see Table 1). The detailed investigation revealed that this was caused by a rounding effect of the addition operation in state S9, where the 2.0 constant value is added to the intermediate exponent calculation in the case that the function's argument has no integer part (the *me* variable is equal to zero). The software model performs the same operation in two steps adding the 1.0 value in each of the steps. Yet, the calculations of the ASM from Fig. 4 can be slightly rearranged in order to mimic the calculation sequence of the software model. The particular modification, which involves states S9, S10 and S11 of the ASM, is shown in Fig. 7. This increases the overall calculations time (a few more clock cycles are needed), but it makes it so that the accuracy is exactly the same as in the software model.

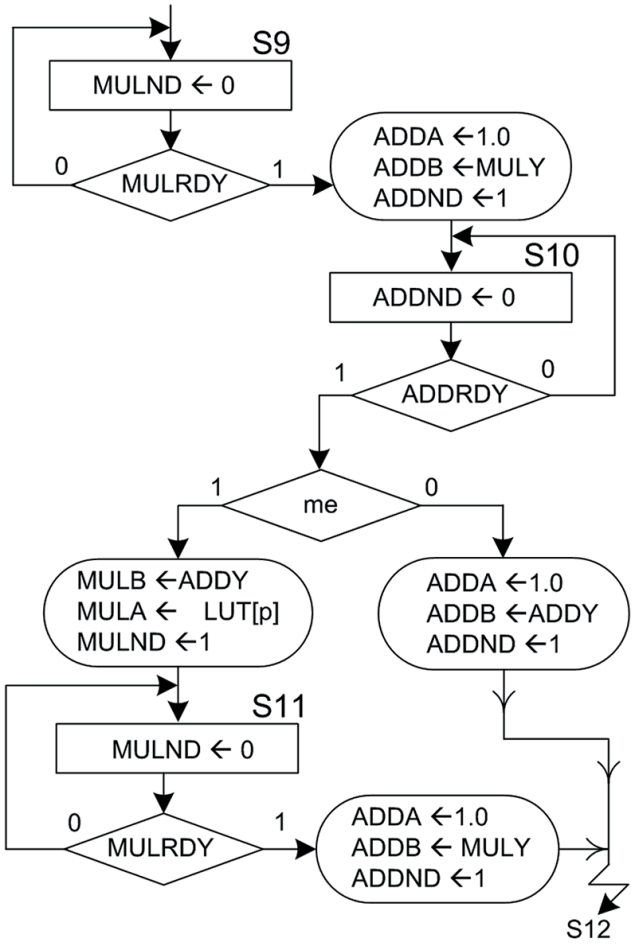

Fig. 7. Altered calculations sequence for the ASM from Fig. 4 enabling the same errors level as in the software model to be obtained

It is of note that the average absolute errors as well as the relative errors for the Padé approximation of the exponent function, applied for the hyperbolic tangent activation function calculation are different for the two considered implementation versions. This is related to the fact that the J version exploits the formula with two exponents (the left hand side of equation (2)), whereas the I version as well as the other versions with McLaurin interpolation, use the formula with a single exponent function.

Table 8 shows, in turn, a comparison of the obtained accuracy with other published results. The proposed implementation method gives a slightly lower accuracy only in comparison with the CORDIC implementation [3] for 64 bits precision (the proposed implementation exercised 32 bits, single precision floating point format). It is also worth to note that the methods described in [3, 6, 9] present the best accuracy the author was able to find in published papers.

Table 8
Comparison of maximum absolute errors for different implementation methods

Method	Sigmoid	Hyperbolic tangent
Chebyshev interpolation [9]		1.929E-07
CORDIC realization, 64 bits [3]	9.97E-11	1.695E-07
CORDIC realization, 32 bits [3]	4.77E-05	1.153E-02
PWL with modifications [6]		2.18E-05
PWL [4]	7.470E-03	
Proposed method	1.192E-07	1.788E-07

3.4. Speed and resources requirement. The overall calculation time of the proposed implementation method depends on the ratio of the number of clock cycles needed to complete the calculations and the actual clock frequency. The number of clock cycles as well as the maximum allowable clock frequency (the minimum clock period) is, in turn, strongly related to the latencies introduced by the FP arithmetic blocks. The latencies can be configured by the user within the Core Generator software.

Table 9 shows selected results of the conducted experiments regarding different latencies assigned to the divider, multiplier and adder blocks. The ASM from Fig. 4 and the Xilinx Spartan-6 XC6SLX100 FPGA chip have been taken into consideration for the experiments. Based on the obtained results,

particularly taking the lowest calculations time into account, the latency of 2 clock cycles for the multiplier and adder block, and 8 clock cycles for the divider has been chosen.

Table 9
Calculations time for different latency of FP blocks

Latency		Min. clock period [ns]	Max clock cycles	Calculations time [ns]
DIV	MUL/ADD			
6	1	20.9	60	1254.5
6	2	14.9	85	1268.6
8	2	11.0	87	958.0
12	2	11.0	91	1002.1
12	3	17.6	116	2044.7
12	4	9.0	141	1269.0
8	4	10.7	137	1472.6
12	5	8.6	166	1433.1

Table 10 shows the FPGA resources requirement and the minimum allowable clock period for the different implementation versions. The versions roughly require a similar number of FPGA flip-flops (FFs) and LUTs, which constitutes 2.3% ... 4.1% of the total available resources of the selected chip. Yet, the basic version (A) needs the least of the resources, whereas the version with parallel calculation of the Padé polynomials (J) requires the most of them. The versions with two FP multiplier blocks require twice more embedded digital signal processing (DSP) blocks. The differences regarding a minimum clock period are also insignificant between the versions.

Table 10
FPGA resources requirement and minimum clock period

Version	LUTs	FFs	DSPs	Period [ns]
A	1885	792	4	11.0
B	1916	792	4	11.2
C	2066	796	4	10.4
D	2039	797	4	10.1
E	1903	820	4	11.3
F	1960	910	8	10.3
G	1976	907	8	10.2
H	2034	911	8	10.3
I	2080	917	4	10.3
J	2624	1059	8	10.3

More considerable differences can be observed in terms of the number of clock cycles needed to complete the calculations. The details are presented in Table 11. The number of clock cycles depends on the argument value of the activation function. For example, when the hyperbolic tangent function and the H version of its implementation are taken into consideration, the relation between the number of clock cycles (L) and the function's argument value (x) is as follows:

$$L = \begin{cases} 6 \Leftrightarrow |x| \geq 9, \\ 55 \Leftrightarrow |x| < 0.25, \\ 61 \Leftrightarrow 0.5 > |x| \geq 0.25, \\ 65 \Leftrightarrow 9 > |x| \geq 0.5 \;\&\; |frac(2x)| < 0.5, \\ 68 \Leftrightarrow 9 > |x| \geq 0.5 \;\&\; |frac(2x)| \geq 0.5, \end{cases} \tag{13}$$

where $frac(x)$ means a fractional part of the x variable. The other cases from Table 11 can be considered analogously.

Table 11
Number of clock cycles and calculation times

Version	Hyperbolic tangent		Sigmoid	
	Cycles	Time [ns]	Cycles	Time [ns]
A	6/77/87	958.0	3/71/81	891.0
B	6/77/87	974.6	3/74/84	940.8
C	6/81/87/94	977.6	3/75/81/85	884.0
D	6/63/69/73/76	974.4	3/60/66/70	707.0
E	6/129/139	1570.7	3/122/132	1491.6
F	6/63/73	751.9	3/56/66	679.8
G	6/63/73	744.6	3/59/69	703.8
H	6/55/61/65/68	700.4	3/51/57/61	628.3
I	6/56/63	648.9	3/57/50	587.1
J	6/44/51	525.3	3/41/48	494.4

The calculation times, given in Table 11, were determined for the highest number of clock cycles and the minimum clock periods from Table 10. The shortest calculation time for both of the activation functions was attained for the J implementation version involving parallel calculation of the Padé polynomials. The second shortest calculation time and the shortest time for the category of the McLaurin interpolation characterize the H implementation version (a restricted range of the argument values, the symmetry feature, equation (3) for $N = 7$, the ASM fragment from Fig. 5, and two FP multipliers were applied). Yet, the accuracy of the H version is slightly compromised.

It is worth to note that the number of clock cycles required for the calculation of the activation function using the CORDIC algorithm [3] is significantly higher than for the proposed method (i.e., for the hyperbolic tangent function, 157 and 273 cycles are required for 32 and 64 bits of the CORDIC implementation, whereas 87 cycles are needed for the proposed method with the best accuracy version).

The calculation speed of the proposed FPGA implementation of the activation function has also been compared with the calculation speed offered by standard CPUs. The conducted tests

have involved the calculations of the hyperbolic tangent function values for 1E6 points equally spaced within the [–10, 10] interval. The obtained results are presented in Table 12.

Table 12
Calculation times of 1E6 values of the hyperbolic tangent function for different platforms

Platform and processor type	Calculations time [ms]
PC desktop *Intel Core i7–950 @ 3.1GHz*	105
PC tablet *Intel Atom x5-Z8500 @ 1.4GHz*	257
Raspberry Pi 2 *ARM Cortex-A7 @ 900MHz*	460.7
Xilinx Zynq (Processing System part) *ARM Cortex-A9 @ 667MHz*	414.8
Arduino Uno R3 *Atmel AVR (ATMEGA328) @ 16MHz*	228 284
Xilinx Spartan-6 *Xilinx IP core, MicroBlaze MCS @ 131.2MHz*	380 924
Xilinx Spartan-6 *IP core, PIC16F87x-based @ 67.8MHz*	169 998
Xilinx Spartan-6 *Proposed method, version B @ 87.5MHz*	917.7
Xilinx Zynq (Programmable Logic part) *Proposed method version B @ 118.1MHz*	679.5
Xilinx Spartan-6 *Proposed method, version J @ 88.4MHz*	543.7
Xilinx Zynq (Programmable Logic part) *Proposed method, version J @ 116.6MHz*	412.2

Apart from two 64-bit processors from Intel dedicated for PC computers other processors have been considered. These include two 32-bit processors with the ARM core (the Broadcom BC2836 encountered in the Raspberry Pi 2 board and the Xilinx Zynq XC7Z020 chip whitch integrates an ARM Cortex-A9 CPU and the Artix-7 FPGA, exploited by the ZedBoard evaluation board) and one 32-bit soft-microcontroller delivered by Xilinx as an IP core (the MicroBlaze MCS implemented in the FPGA board with a Spartan-6 chip), as well as two 8-bit microcontrollers (an Atmel AVR and the soft-microcontroller based on the Microchip PIC16F87x, described in [18] and implemented in a Spartan-6 FPGA chip) have been considered.

Table 12 also contains the calculation times for the proposed FPGA implementation method of the activation function. Two versions have been taken into account: the most accurate (B) and the fastest (J), implemented in two different FPGA chips from Xilinx (the Spartan-6 XC6SLX100 and Zynq XC7Z020). The FPGA implementations, including the soft-microcontrollers, have been clocked with the maximum allowable clock frequency reported by the design tool. It is of note that, for the same design, Zynq FPGAs enable the application of higher clock frequency than Spartan-6 FPGAs. The calculation times for all of the platforms, with exception of the Intel processors, have been measured by the external universal counter. For the Intel processors and PC platforms, the calculation times have been obtained by reading the system time before and after the calculations.

The obtained results revealed that the calculation time of the test algorithm for the most accurate version of the proposed realization method, implemented in the Xilinx Zynq FPGA chip, in comparison with the calculation time attained by the Intel processors and processors with the ARM core is 6.5, 2.6, 1.5 and 1.6 times longer respectively. However, the most powerful feature of FPGA technology is that a single computational block can be replicated multiple times within a FPGA chip. Each block, in turn, can accomplish its computations in parallel with the others. Therefore, FPGA implementation of a neural network comprised of more than 7 neurons using the hyperbolic tangent activation functions, realized according to the proposed method, should offer a shorter calculation time than the same neural network implemented by software with the usage of the Intel Core i7–950 CPU. Comparing to the processors with the ARM core, the calculation time of the FPGA implementation of a neural network consisting of more than 2 neurons should be shorter. It is of note that the above estimations are valid under the assumption that the ratio of the calculations time of the single neuron output, delivered by a CPU and the FPGA implementation, is very close to the calculation time ratio for the hyperbolic tangent activation function.

The calculation times obtained with the usage of the MicroBlaze MCS soft-microcontroller (which turned out to be surprisingly slow), AVR microcontroller and PIC16F87-x based soft-microcontroller are much longer than for the FPGA implementation (560, 336 and 250 times respectively). This means that the FPGA implementation of any neural network, as compared to software implementations using these microcontrollers, should result in an excellent performance.

The fastest FPGA implementation version of the activation function with the parallel calculations of the Padé polynomials brings 39.4% shorter calculation time of the test algorithm in comparison with the most accurate version. However it is achieved at the price of the 36.9% higher FPGA resources requirement and the 33% lower accuracy perceived in terms of the maximum absolute error.

4. Other design techniques

Manual HDL coding of any computational algorithm is not a straightforward process. Compared to a software development process, it always consumes much more time and requires a significant effort from the designer. Fortunately, in recent years new tools have become available, which allow the automatic generation of HDL code from programming languages such as C or Matlab. Since the proposed implementation method of activation function implementation has been developed and tested using the Matlab environment and C programming language, automatic HDL code generation techniques have also

been investigated. Only the basic version of the proposed implementation method with the McLaurin interpolation of the exponent, applied for the hyperbolic tangent activation function implementation, has been taken into consideration.

4.1. Matlab HDL Coder. The Matlab HDL Coder, introduced some time ago by MathWorks company, automatically generates HDL code (Verilog or VHDL) from Matlab functions, Simulink blocks or System Objects [19]. Yet, it only supports fixed point arithmetic. Also some mathematical functions, such as the exponent, are not supported for the HDL code generation. Therefore, equations (1) or (2) cannot be directly used and the method described in section 2 still must be applied.

In order to facilitate a fixed point design, the HDL coder provides automatic conversion from fixed to floating point arithmetic. Yet, in the case of the proposed implementation method of the activation function, the fixed point code, generated by the HDL coder required some manual modifications – the simulation of the code yielded incorrect results. It turned out that the problem was caused by the division operation, which in a fixed point code version was denoted by the HDL coder as a purely integer division without a reminder. After a proper alteration of the fixed point code, the simulations' results were correct. However, another problem was encountered after generating Verilog HDL code and it was still related to the division operation. The Verilog code included a simple division operator in a place where the fixed point division operation was needed. For the Xilinx FPGA synthesis tools, the division operator can only be used when the divisor is a constant and is a power of 2. Therefore, the generated Verilog code had to be altered by the usage of the divider IP core instance from the Xilinx Core Generator, instead of a simple division operator. The important issue was to properly adjust the bit width of the divisor, dividend, quotient and fractional part of the divider IP block. After introducing modifications, the code was fully synthesizable and calculations were conducted correctly.

Table 13 shows the implementation results of the code generated by the HDL coder, obtained from Xilinx ISE Design Suite for the Spartan-6 XC6SLX100 FPGA chip. Minimum clock period time is given in nanoseconds. Two versions have been considered: A – with an active option of the synthesis tool regarding the usage of DSP blocks, B – the option was inactive.

Table 13
Implementation results of the HDL coder generated code

Version	LUTs	FFs	DSPs	RAMB	Period	Cycles
A	6102	763	80	1	227.1	9
B	27892	831	17	1	245.2	9

The implementation of the code requires many more FPGA resources than implementation of the same algorithm with manual HDL coding (see Table 10, version A). It is particularly observed in the case of version B, when no DSP usage is chosen (yet, the DSP blocks are still utilized by the divider IP block). Compared to the manual HDL coding, the implementation needs an additional Block RAM block (RAMB) as well.

The achieved minimum clock period time is also many times higher than for manual HDL coding. The detailed analysis of the Verilog code revealed that the HDL coder creates an almost purely combinational description of a digital circuit. Although there are available options in the HDL coder allowing insertion of pipeline registers, but they are placed only at the inputs or the outputs of the generated combinational circuit. Thus, they have almost no influence on the minimum clock period time. In the considered implementation, one input and one output pipeline register has been chosen. The divider block needs 7 clock cycles (this is the minimum available value for this IP block), therefore the total number of clock cycles required for completion of the calculations accounts for 9 cycles. This gives 2043.9 ns of total calculation time. The accuracy of the implementation is no different than for the software model – the maximum absolute error amounts to 2.384E-07.

4.2. Vivado HLS. Another way of automatic HDL code generation from high-level programming languages is the usage of Vivado HLS software as a part of the Vivado Design Suite [20]. This is a relatively new tool from Xilinx, which allows for algorithm specification in C/C++ or System C languages. The Vivado HLS, unlike the Matlab HDL coder, also supports floating point data types. It provides the ability of automatic usage of Block RAM blocks, DSP blocks and floating point libraries. Unfortunately, the Vivado HLS is only dedicated for newer Xilinx FPGAs families, such as Virtex-7 or Kintex-7, and does not support older Spartan-6 chips, considered in the previous section.

Like the HDL coder, unfortunately the Vivado HLS also does not support the exponent function. Yet, the previously prepared C function, modeling the proposed algorithm, can be almost directly used with the Vivado HLS. The only change which was needed in the C function code was related to the parameters of the function (for the Vivado HLS, returned values should by conveyed by pointers).

Table 14 shows implementation results of the C code specified hyperbolic tangent activation function, and – for comparison – the manually HDL coded analogous algorithm, described by the ASM from Fig. 4. The XC7K160 chip from the Kintex-7 FPGA family has been taken into consideration. The minimum clock period time is given in nanoseconds.

Table 14
Implementation results of the Vivado HLS and manually HDL coded algorithms for Xilinx Kintex-7 family

Implement	LUTs	FFs	DSPs	Period	Cycles
Vivado HLS	3077	2148	16	8.4	6/174/184
Man. coded	1852	775	2	7.7	6/77/87

As the results indicate, the implementation with manual HDL coding gives considerably better results both in terms of the FPGA resources requirement and calculations speed. In

particular, the number of clock cycles needed to conduct the calculations is more than twice lower for the manually HDL coded algorithm. Conducted simulations also indicate that the obtained accuracy is the same as for the C function modeling the proposed algorithm.

5. Conclusions

It has been shown that a high accuracy of a FPGA implementation of the hyperbolic tangent and sigmoid activation function can be obtained by applying the direct realization method with the exponent function interpolation using the McLaurin series or Padé polynomials. Conducted experiments with slightly different implementation versions revealed that, due to cumulative rounding errors, an applied arithmetic operation sequence has a considerable impact on the overall calculations accuracy. For the hyperbolic tangent activation function, the best obtained accuracy of the proposed implementation versions with a 32 bits single precision floating point arithmetic is very close to the most accurate solution reported so far – the 64 bits fixed precision CORDIC implementation (1.788E-07 vs. 1.694E-07 of the maximum absolute error). Yet, the proposed method is faster than the CORDIC (974.6ns/Spartan-6 vs. 2730ns/Virtex-5 of the total calculations time), under comparable FPGA resources requirement.

Compared to the PWL approximation or DCT interpolation, a disadvantage of the proposed implementation method is a relatively long calculation time. The method requires a significant number of floating point operations, particularly for the McLaurin interpolation of the exponent function: 21/16 multiplications, 13/11 additions for the hyperbolic tangent/sigmoid function, and 1 division. Yet, some of the operations can be conducted in parallel. The number of operations for the implementations with the Padé approximation of the exponent function is smaller and accounts for 8/7 multiplications, 11/10 additions and 2 divisions respectively. The implementation of the PWL approximation or DCT interpolation may require only a few clock cycles to complete calculations. This leads to a very short calculation time of the activation function (e.g., in the case of [11] the calculations time of the hyperbolic tangent function as short as 7.6 ns for the Xilinx Virtex-7 FPGA is reported). However, the accuracy of the aforementioned methods is a few (3 ... 5) orders of magnitude lower than the proposed method. It is of note that the issues related to the calculations throughput are very rarely considered in the published solutions (they are strongly focused on the obtained calculations accuracy instead). Therefore, a solid comparison in terms of the calculations speed between the proposed method and other solutions is difficult to be carried out.

Despite the substantial number of operations, the accuracy seems to be a significant advantage of the presented method. Other conducted software simulations show that exercising the proposed method for 64 bits double precision floating point arithmetic, the maximum absolute error can be as low as 2.980E-08 and 2.221E-16 for the hyperbolic tangent and sigmoid functions respectively. It is also important to note that for the hyperbolic tangent activation function and the McLaurin interpolation of the exponent function, the formula with a single exponent function – the right hand side of equation (2) – has been taken into consideration. Such a formula is used e.g. by the Matlab *tansig* function. Yet, if the formula from the left hand side of equation (2) with two exponents is applied, then the feasible accuracy can be remarkably higher. The maximum absolute error can be as low as 5.960E-08 and 3.331E-16 for a single and double precision floating point arithmetic respectively. Unfortunately, it turned out that such accuracy cannot be attained using the Padé approximation of the exponent function. However, the advantage of the implementation method with the Padé approximation is that it ensures shorter calculation time (even 33% ... 46%) than the method with the McLaurin interpolation of the exponent.

The calculation speed comparison of the FPGA implementation of the hyperbolic tangent function and its realization with standard CPUs suggests that the parallel FPGA implementation of a neural network, consisting of a few neurons with the activation function realized according to the proposed method, can deliver shorter calculation time than the neural network implemented by software executed on a relatively high performance PC computer. The more neurons a neural network contains, the more considerable the advantage of the calculation speed of the FPGA implementation.

As far as design tools allowing automatic HDL code generation from high level programming languages are concerned, it seems they have significant potential. They provide a fast prototyping feature and allow for the implementation of complicated algorithms without a challenging and time-consuming manual HDL coding process. Yet, a manual HDL coding still allows obtaining better implementation results in terms of calculations speed and FPGA resources requirement. The considerable drawback of the tools is a lack of a support for some mathematical functions, such as the “exponent”.

References

[1] A. Gomperts, A. Ukil, and F. Zurfluh, “Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications”, *IEEE Trans. on Industrial Informatics* 7(1), 78–89 (2011).

[2] A. Armato, L. Fanucci, G. Pioggia, and D. De Rossi, “Low-error approximation of artificial neuron sigmoid function and its derivative”, *Electronics Letters* 45(21), 1–2 (2009).

[3] V. Tiwari and N. Khare, “Hardware implementation of neural network with Sigmoidal activation functions using CORDIC”, *Microprocessors and Microsystems* 39, 373–381 (2015).

[4] A. Armato, L. Fanucci, E.P. Scilingo, and D. De Rossi, “Low-error digital hardware implementation of artificial neuron activation functions and their derivative”, *Microprocessors and Microsystems* 35, 557–567 (2011).

[5] M.T. Tommiska, “Efficient digital implementation of the sigmoid function for reprogrammable logic”, *IEEE Proc. Comput. Digit. Tech.* 150(6), 403–411 (2003).

[6] P. Ferreiraa, P. Ribeiroa, A. Antunesa, and F. Morgado Dias, “A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function”, *Neurocomputing* 71(1–3), 71–77 (2007).

[7] M. Bajger and A. Omondi, "Low-error, High-speed Approximation of the Sigmoid Function for Large FPGA Implementations", *Journal of Signal Processing Systems* 52, 137–151 (2008).
[8] T. Orlowska-Kowalska and M. Kaminski, "FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System", *IEEE Trns. on Industrial Informatics* 7(3), 436–445 (2011).
[9] D. Baptista and F. Morgado-Dias, "Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks", *Neural Computing and Applications* 23(3), 601–607 (2013).
[10] I. del Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled accuracy approximation of sigmoid function for efficient FPGA-based implementation of artificial neurons", *Electronics Letters* 49(25), 1598–1600 (2013).
[11] A.M. Abdelsalam, J.M. Pierre Langlois, and F. Cheriet, "A Configurable FPGA Implementation of the Tanh Function using DCT Interpolation", *IEEE 25th Annual Int. Symp. on Field Programmable Custom Computing Machines*, 168–171, (2017).
[12] F. Zhoua, J. Liua, Y. Yua, X. Tiana, H. Liub, Y. Haoa, S. Zhanga, W. Chena, J. Daia, and X. Zhenga, "Field-programmable gate array implementation of a probabilistic neural network for motor cortical decoding in rats", *Journal of Neuroscience Methods* 185, 299–306 (2010).
[13] D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo, I. Gonzalez, F.J. Gomez-Arribas, J. Aracil, and F. Palacios, "High-Level Languages and Floating-Point Arithmetic for FPGA Based CFD Simulations", *IEEE Design & Test of Computers* 28(4), 28–37 (2011).
[14] Z. Hajduk, "High accuracy FPGA activation function implementation for neural networks", *Neurocomputing (Brief papers)* 247, 59–61 (2017). DOI: 10.1016/j.neucom.2017.03.044
[15] M. Vajta, "Some remarks on Padé-approximations", *Tempus-Intcom Symposium*, Sept. 2000, 1–6 (2000).
[16] J. Kluska and Z. Hajduk, "Hardware implementation of P1-TS fuzzy rule-based systems on FPGA", *Proc. Artif. Intell. Soft Comput.* 7894, 282–293 (2013).
[17] Z. Hajduk, B. Trybus, and J. Sadolewski, "Architecture of FPGA Embedded Multiprocessor Programmable Controller", *IEEE Trans. Ind. Electron.* 62(5), 2952–2961 (2015).
[18] Z. Hajduk, "An FPGA embedded microcontroller", *Microprocessors and Microsystems* 38(1), 1–8 (2014).
[19] C. Maxfield, "MathWorks' MATLAB now supports HDL code generation", EETimes, 2012, available at http://www.eetimes.com/document.asp?doc_id=1317035
[20] M. Santarini, "Xilinx unveils vivado design suite for the next decade of 'all programmable' devices", *Xcell Journal* 79, 8–13 (2012).