Received 8 February 2023, accepted 28 February 2023, date of publication 6 March 2023, date of current version 14 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3253668

APPLIED RESEARCH

Very High Accuracy Hyperbolic Tangent Function Implementation in FPGAs

ZBIGNIEW HAJDUK AND GRZEGORZ RAFAŁ DEC
Department of Computer and Control Engineering, Faculty of Electrical and Computer Engineering, Rzeszów University of Technology, 35-959 Rzeszów, Poland

Corresponding author: Zbigniew Hajduk (zhajduk@kia.prz.edu.pl)

This work was supported in part by the Maintenance of Research Capacity Fund of the Department of Electrical and Computer Engineering, Rzeszów University of Technology, under Grant PB23.EA.23.001; and in part by the Minister of Education and Science of the Republic of Poland within the ‘‘Regional Initiative of Excellence’’ Program, from 2019 to 2023, under Project 027/RID/2018/19, amount granted 11 999 900 PLN.

ABSTRACT The paper presents in detail a relatively simple implementation method of the hyperbolic tangent function, particularly targeted for FPGAs. The research goal of the proposed method was to examine the usage of the approximation of ordinary or Chebyshev polynomials for the implementation of the function. Several miscellaneous implementation versions have been considered. They differ in the polynomial degree, number of intervals for which the domain of the function is divided, etc. Both floating-point and fixed-point implementations have been presented. An impact on the FPGA resources utilization and calculations time for the implementation versions has also been briefly analyzed. Special attention has been paid to the accuracy of the calculations of the function. It turned out that applying the proposed method, a very high calculations accuracy can be achieved, while simultaneously maintaining reasonable resources utilization and short calculations time. The proposed method can be an effective alternative to other encountered implementation methods such as CORDIC. Additionally, the presented hardware architecture is more versatile and can be easily adapted for the implementation of other mathematical functions.

INDEX TERMS Activation function, artificial neural network, FPGA.

I. INTRODUCTION

The concept of Artificial Neural Networks (ANNs) was first introduced in the middle of the 20th century. Through an increase of computation power and number of collected data, ANNs have managed to undergo rapid development in recent years. There are many applications of ANNs found in areas including pattern recognition, image processing, speech recognition, control systems, predictions, etc. [1], [2], [3]. Depending on the specific architecture, ANNs may consist of large number of neurons clustered into layers. If neurons within the layer are able to process the signal in parallel, this directly leads to significant acceleration of the calculations speed, which might be important in certain applications of ANNs. Parallelism of operations is the natural feature of Field Programmable Gate Array (FPGA) technology. Therefore, FPGAs are promising candidates for ANNs implementations in the areas where calculations speed is the pivotal factor. The other advantage of FPGAs might include the reduced cost of implementation [4], particularly compared to high-performance computers or GPUs.

The most important, expensive and hard to implement part of any hardware realization of ANNs is a neuron's non-linear activation function [5]. The hyperbolic tangent is commonly used activation function for ANNs. Besides the calculations speed, the implementation method of the hyperbolic tangent function has also an essential impact on the calculations accuracy of the whole ANN. The activation function accuracy might be very important particularly in the case where the ANN is trained using PC software, such as Matlab, and then the calculated neurons' weights and biases are used in the FPGA implementation of this ANN. High accuracy of the activation function is also needed for the correct implementation of a learning algorithm (e.g., the backpropagation) for an ANN [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Ludovico Minati.

There are a number of papers considering a hardware implementation of activation functions. They differ in an obtained accuracy, applied approximation method, implementation's cost, type of used arithmetic (fixed or floating point), etc. A frequently used approximation method for the hyperbolic tangent function implementation is the piecewise linear (PWL) interpolation [3], [4], [7], [8]. In [9], the 9 segments of Simplicial Canonical Piecewise Linear model and Grey Wolf Optimizer has been used in order to approximate the hyperbolic tangent function. A look-up table (LUT) with 8192 elements has been exercised for the hyperbolic tangent function implementation in CompactRIO hardware platform [10]. A similar technique – a LUT with linear interpolation between LUT's points has been applied in [1]. A Taylor series approximation of the hyperbolic tangent function is considered in [11], [12]. The usage of the COordinate Rotation DIgital Computer (CORDIC) algorithm for implementation of the hyperbolic tangent function is featured in [5], [13]. The implementation of this function based on the Discrete Cosine Transform Interpolation Filter (DCTIF) is presented in [14]. An approximation of the hyperbolic tangent function exercising Lagrange, Chebyshev and least square method is considered in [15]. In [16], a piecewise 2nd degree polynomial approximation is applied for the hyperbolic tangent function implementation. Four different approaches for the hyperbolic tangent function implementation in FPGAs, namely the Taylor series expansion for the exponential function, Elliot-2, Elliot-93, and CORDIC LUT-based are examined in [17]. An interesting low-power analog design for the hyperbolic tangent activation function, dedicated for an implementation in CMOS technology, is presented in [18]. A kind of a direct implementation of hyperbolic tangent and sigmoid activation functions is featured in [19], which is the authors' previous work. In this case, the main realization difficulty has been shifted to the approximation of the exponent function. McLaurin series as well as Padé polynomials combined with LUTs have been proposed in order to accomplish this approximation.

It is worth noting that some of the papers (e.g., [7], [15]) did not revealed any details of the proposed implementation method. Some others only portrayed a rough idea of how the method was actually implemented (e.g., only simplified block diagram of a data-path was featured). In this paper the detailed FPGA implementation method of the hyperbolic tangent function is presented. The proposed method is based on the direct polynomial as well as Chebyshev polynomial approximation. A number of slightly different realizations are also featured in this paper. They differ in the type of applied arithmetic (both fixed-point as well as floating-point arithmetic are considered), and the degree of the applied polynomial. Different number of intervals (including relatively high number of intervals, exceeding 200) in which the domain of the approximated function is divided, and its impact on the implementation results has also been examined. The proposed method is easy to realize and allows reasonable implementation results to be obtained, both in terms of relatively low number of FPGA resources requirement and short calculations time. The other substantial feature of the proposed method is the ability to achieve both typically reported calculations accuracy (i.e., at the level of E-3, E-4) and very high accuracy (i.e., at the level of E-8 for single precision floating-point arithmetic or higher for fixed-point arithmetic), hardly encountered in other publications. Additionally, the proposed method and hardware architecture described in detail in the paper are more versatile and can easily be adapted to the FPGA realization of other mathematical functions.

II. IMPLEMENTATION METHOD

The hyperbolic tangent activation function is given by the following formula:

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

The proposed implementation method of the hyperbolic tangent function relies on direct interpolation of this function utilizing polynomials. Both ordinary (direct) and Chebyshev polynomials have been taken into consideration. Chebyshev polynomials are typically applied for an approximation of trigonometric, logarithmic, and other functions [20]. Ordinary polynomials were chosen, in turn, due to the simplicity of the calculations. Since for an approximation these polynomials are determined within the narrow range of arguments, namely $[-1, 1]$, the domain of the interpolated function has to be divided into several intervals. In order to achieve reasonable implementation results of the hyperbolic tangent function, i.e., low FPGA resources requirement, short calculations time, and high calculations accuracy, the number of the aforementioned intervals, as well as the polynomial degree, should be carefully determined. The increase of the polynomial degree also increases the accuracy but simultaneously decreases the calculations time and increases the resources requirement. Similarly, increasing the number of intervals improves the accuracy but also increases the resources requirement.

The other factor which influences the choice of the number of intervals is the length of the subsequent intervals. Approximating a $f(x)$ function, where $x \in [a, b]$, it is required to convert the $[a, b]$ domain into the $z \in [-1, 1]$ interval. It is achieved by the formula:

$$z = \frac{2 \cdot x - b - a}{b - a}. \tag{2}$$

In order to avoid the division operation in (2), which is costly in digital implementation in terms of both calculation time and resources requirement, the interval length (the denominator value in (2)) should be an integer power of 2. In this case, the division operation can be replaced either by a simple bit-shift operation or by arithmetic addition, depending on the type of applied arithmetic (fixed or floating-point).

It is also worth noting that the hyperbolic tangent function is symmetric in the reference to the origin of the coordinate system, and the function's value is virtually constant (with suitable accuracy) for arguments range $|x| \geq 10$. Therefore, the arguments range of the hyperbolic tangent function, considered for the proposed implementation method, can be limited to the [0, 10) interval.

In order to determine the number of intervals for which the domain of the hyperbolic tangent function should be divided, as well as the lowest polynomial degree resulting in the highest possible accuracy, several computational experiments have been carried out taking advantage of the Matlab software.

The [0, 0.25) interval was arbitrary chosen as the starting point. In the next step, using Matlab standard functions, the hyperbolic tangent function was approximated in this interval by an ordinary polynomial of degree from 2 to 7. For every polynomial degree, the maximum absolute error of the approximated function was calculated. The values of the approximated function were calculated using single precision floating-point arithmetic (the coefficients of the polynomial as well as the arguments of the approximated function were converted to single precision variables, and using this type of variables, the polynomial values were calculated). It turned out that the maximum absolute error no lower than at the level of E-8 can be achieved, whereas the lowest polynomial degree at this error level accounts for 5. Therefore, for further experiments, the 5th polynomial degree was chosen. In the next steps, the subsequent intervals were sought so that the whole range [0, 10) was covered. The boundaries of the subsequent intervals were subjected to the following rules: 1) the initial value of the next interval is the same as the final value for the previous interval, 2) the final value of the next interval is a sum of the initial value and the length of the interval. The interval length, in turn, had to meet 2 conditions: a) it had to be an integer power of 2, and b) it had to be as long as possible while simultaneously maintaining the maximum absolute error of the single precision approximation of the hyperbolic tangent function at the level of E-8. To find an interval that fulfils the second condition (b), the maximum absolute error of the approximation has to be calculated for at least 2 or more subsequent intervals meeting the first condition (a). Finally, using this procedure, the following 15 intervals were found: [0, 0.25), [0.25, 0.5), [0.5, 0.75), [0.75 1.0), [1.0, 1.25), [1.25, 1.5), [1.5, 2.0), [2.0, 2.5), [2.5, 3.0), [3.0, 3.5), [3.5, 4.0), [4.0, 5.0), [5.0, 6.0), [6.0, 8.0), [8.0, 10.0).

In the next subsections, the details of the hardware implementation of the hyperbolic tangent function are presented. The described implementations are based on the aforementioned, preliminary determined approximation parameters.

A. CHEBYSHEV APPROXIMATION

A function approximation utilizing Chebyshev polynomials is given by the formula:

$$f(z) = \sum_{k=0}^{N-1} c_k \cdot T_k(z), \tag{3}$$

where $c_0 \dots c_{N-1}$ are constant coefficients, and $T_k(z)$ is the Chebyshev polynomial such as:

$$T_k(z) = cos(k \cdot arccos(z)), \ -1 \leq z \leq 1. \tag{4}$$

Using some algebraic identities and manipulation, the Chebyshev polynomial can be expressed in the following form:

$$T_k(z) = \begin{cases} 1 \iff k = 0, \\ z \iff k = 1, \\ 2 \cdot z \cdot T_{k-1}(z) - T_{k-2}(z) \iff k \geq 2. \end{cases} \tag{5}$$

Calculating a function value using (1) and (3) might still be a little inconvenient. Yet, the Clenshaw iterative formula [21] can be applied instead:

$$\begin{gathered} d_{N+1} = d_N = 0, \\ d_k = 2 \cdot z \cdot d_{k+1} - d_{k+2} + c_k, \ for\ k = N-1, N-2, \dots, 1, \\ f(z) = z \cdot d_1 - d_2 + c_0. \end{gathered} \tag{6}$$

For example, for $N = 6$ the value of the approximated function can be calculated as:

$$\begin{aligned} d_5 &= c_5, \\ d_4 &= 2 \cdot z \cdot d_5 + c_4, \\ d_3 &= 2 \cdot z \cdot d_4 - d_5 + c_3, \\ d_2 &= 2 \cdot z \cdot d_3 - d_4 + c_2, \\ d_1 &= 2 \cdot z \cdot d_2 - d_3 + c_1, \\ f(z) &= z \cdot d_1 - d_2 + c_0. \end{aligned} \tag{7}$$

The essential issue pertaining to the approximation of the function is the calculation of coefficients $c_0 \dots c_{N-1}$. A separated set of values of coefficients has to be calculated for every interval for which the considered domain of the approximated function is divided. In order to accomplish the calculation of coefficients, several scripts in Matlab and Python have been developed and applied. These scripts directly generate a code in the Hardware Description Language (Verilog HDL has been applied in this case), which describes a read-only memory containing all of the coefficients values.

Both floating-point and fixed-point implementations of hyperbolic tangent function utilizing Chebyshev polynomials approximation have been developed. In this subsection, only the floating-point implementation will be described in detail.

A simplified block diagram of the digital circuit performing the calculation of the hyperbolic tangent function value is depicted in Fig. 1.

Since single-precision floating-point arithmetic is applied, most of signal lines visible on Fig. 1 are 32-bit buses. The X line is the input argument of the hyperbolic tangent function, whereas the Y line represents the calculated value of the function. The scalar ND signal initiates the calculations. When the result is ready for reading, the RDY signal is

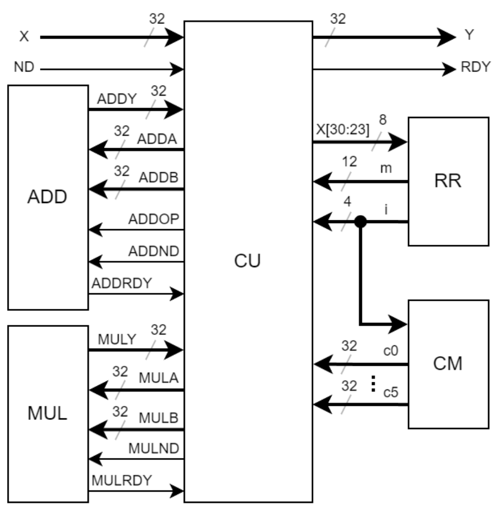

FIGURE 1. General architecture of the floating-point Chebyshev approximation module.

TABLE 1. Truth table describing the RR block.

Range of X	X[30:23]	X[22]	X[21]	i	m
$0 \leq \|X\| < 0.25$	$\leq 7Ch$	x	x	0	$BE8h$
$0.25 \leq \|X\| < 0.5$	$7Dh$	x	x	1	$BF4h$
$0.5 \leq \|X\| < 0.75$	$7Eh$	0	x	2	$BFAh$
$0.75 \leq \|X\| < 1.0$	$7Eh$	1	x	3	$BFEh$
$1.0 \leq \|X\| < 1.25$	$7Fh$	0	0	4	$C01h$
$1.25 \leq \|X\| < 1.5$	$7Fh$	0	1	5	$C03h$
$1.5 \leq \|X\| < 2.0$	$7Fh$	1	x	6	$C06h$
$2.0 \leq \|X\| < 2.5$	$80h$	0	0	7	$C09h$
$2.5 \leq \|X\| < 3.0$	$80h$	0	1	8	$C0Bh$
$3.0 \leq \|X\| < 3.5$	$80h$	1	0	9	$C0Dh$
$3.5 \leq \|X\| < 4.0$	$80h$	1	1	10	$C0Fh$
$4.0 \leq \|X\| < 5.0$	$81h$	0	0	11	$C11h$
$5.0 \leq \|X\| < 6.0$	$81h$	0	1	12	$C13h$
$6.0 \leq \|X\| < 8.0$	$81h$	1	x	13	$C16h$
$8.0 \leq \|X\| < 10.0$	$82h$	0	0	14	$C19h$
$\|X\| \geq 10.0$	other cases			15	$0h$

activated for a single clock cycle (the clock signal has not been shown in the diagram from Fig. 1). The module in Fig. 1 contains two arithmetic blocks ADD and MUL executing the addition and multiplication operations, respectively, the range recognition (RR) block responsible for the determination of the intervals for which the input value is belonging to, the coefficient memory (CM) block, and the control unit (CU).

For the implementation of the module from Fig. 1, the ADD and MUL blocks can be obtained as IP cores delivered by the vendor of the applied FPGA chip. For the ADD block, the ADDA and ADDB lines represent two operands of the addition operation, whereas the ADDY line contains the operation result. The ADDND signal, activated for a single clock cycle, initiates the operation. The ADDRDY indicates whether the addition operation has been completed. The ADD block can also perform a subtraction operation. In this case, the ADDOP signal should be activated. Otherwise, the addition operation is executed by the ADD block. Analogous naming convention of signals lines applies to the MUL block from Fig. 1.

The RR block is a purely combinational circuit. Its input is a bit selection of the input value X (the 8 bits of the exponent and the 2 most significant bits of the mantissa in the floating-point representation of the X). This block delivers two outputs: the interval identifier (the i signal) and the value referring to the "$-b-a$" expression in (2), which constitutes the m signal line. The truth table describing the operation of the RR block is presented in Table 1. The CM block takes the interval identifier from the RR block and delivers a set of coefficients assigned to the specific interval. It is implemented as a number of distributed read-only memories (a single memory for a single coefficient).

The control unit block performs essential operations of the module from Fig. 1. It is implemented as a state machine. The sequence of operations performed by the CU block is depicted in Fig. 2 as a form of an algorithmic state machine (ASM) diagram. The diagram adopts syntax elements similar to those used in Verilog HDL. Particularly, Verilog operators (e.g., arithmetic, logic, concatenation, bit selection operators) are directly used in the diagram. For example, the expression "$\{1'b0, X[30:23]+1, X[22:0]\}$" denotes the concatenation of the 1-bit binary constant equal to 0, partial selection of the bits for 30 down to 23 of the X vector incremented by 1 and partial selection of the bits for 22 down to 0. The signals names denoted using capital letters relate to external input/output buses and signals, whereas non-capital letters are exploited to mark internal variables (registers).

The sequence of operations described by this ASM directly refers to the case where $N = 6$ and the 15 intervals mentioned above are used. After activation of the ND signal, the CU block goes to the S1 state. In this state, the calculation of the numerator of (2) is initiated. It is worth noting that the multiplication by 2 of a floating-point number is equal to the addition of 1 to the exponent value (bits 23…30 of the floating-point representation of this number). Therefore, the first operand (ADDA) for the addition module receives the input value (X) multiplied by 2, whereas the second operand (ADDB) captures the value of the expression "$-b - a$" in (2). In the S2 state, the final value of (2) is established. This value depends on the "$b - a$" expression (the denominator of (2)); thereby, it depends on the interval identifier (the i variable) delivered by the RR block. Since the value of the "$b - a$" expression is always divisible by 2, the result of the division in (2) can be obtained by a simple addition of an integer number to the exponent of the result of the addition operation, initiated in the S1 state. Along with the calculation of the z variable, its doubled value is also calculated in the same state and assigned to the MULB (the second operand of the multiplication operation). However, when the absolute value of the input argument X is higher than or equal to 10.0 (in this case the interval identifier i amounts to 15), nothing is calculated in the S2 state and the ASM goes to the S13 state. In this state, the final value is assigned to the output Y, namely -1.0 or 1.0 depending on the arithmetic sign of the input argument. In the S3 state, the ASM waits until the multiplication of two operands, namely the doubled value of the z variable and c_5 coefficient, is performed and initiates the

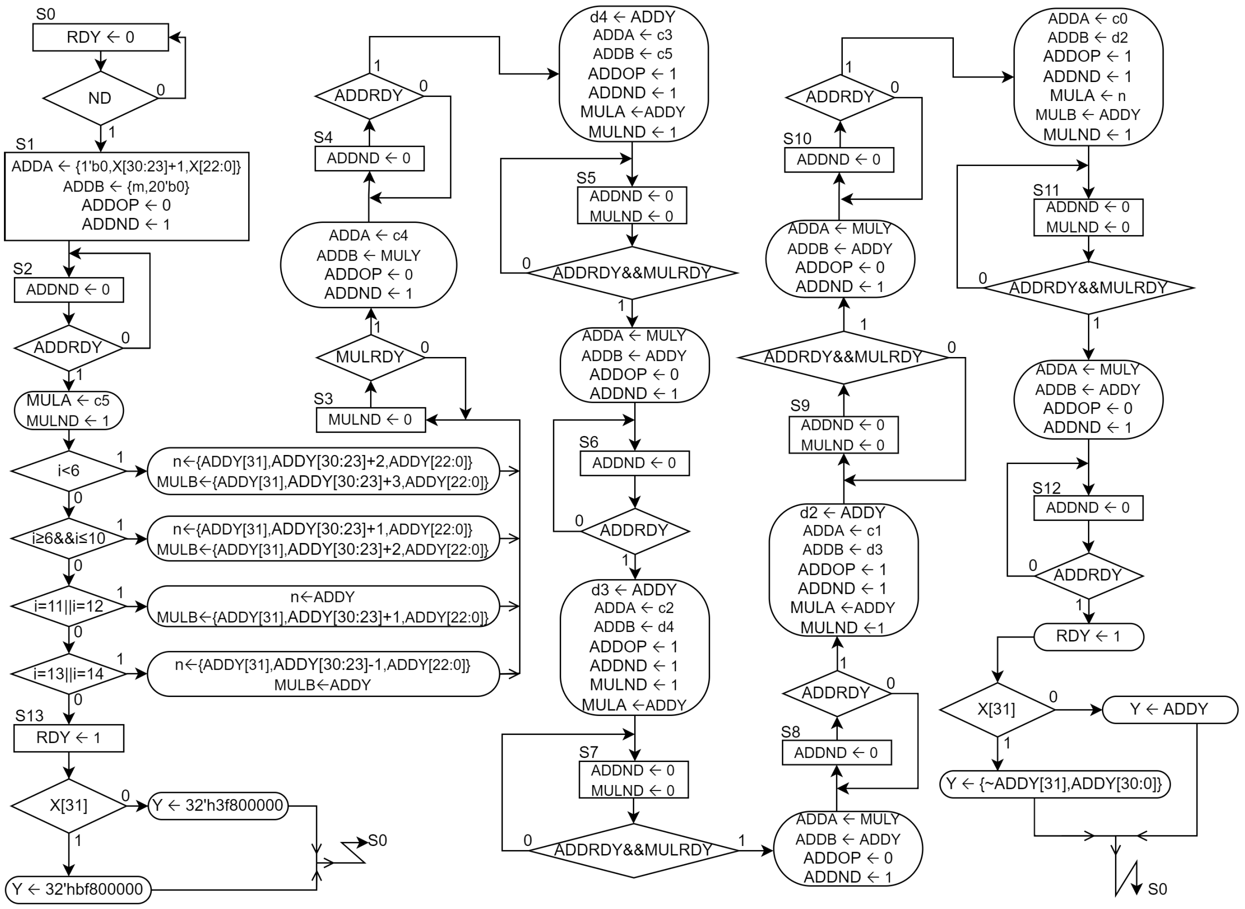

FIGURE 2. ASM diagram describing the control unit block from Fig. 1.

addition which leads to the calculation of the d_4 in (7). When the aforementioned addition is completed, two simultaneous arithmetic operations are initiated in the S4 state, namely the calculation of the expressions "$-d_5+c_3$" (note that $d_5 = c_5$) and "$2 \cdot z \cdot d_4$" (note that the value of "$2 \cdot z$" is still stored in the MULB), which leads to the calculation of the value of d_3 in (7). Similarly, the values of d_2, d_1, and finally $f(z)$ are computed in the states S5…S12 of the ASM. In the state S12, the RDY signal is activated, as well as the final result is established, depending on the sign of the input argument (if the input argument is a negative value, the final result of the calculation of the hyperbolic tangent function should also be a negative value).

B. DIRECT POLYNOMIAL APPROXIMATION

Apart from the Chebyshev polynomial, a direct polynomial has also been examined for the hyperbolic tangent function approximation and hardware implementation. This approximation can be performed using the simple formula:

$$f(z) = \sum_{k=0}^{N-1} a_k \cdot z^k, \tag{8}$$

where $N-1$ is the degree of the polynomial and $a_0 \ldots a_{N-1}$ are the coefficients of the polynomial. A more convenient

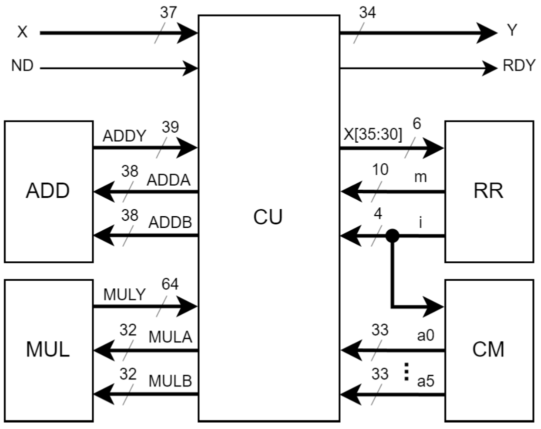

FIGURE 3. General architecture of the fixed-point direct polynomial approximation module.

formula for the calculation of a polynomial value is the Horner scheme:

$$\begin{aligned} s_{N-1} &= a_{N-1}, \\ s_k &= s_{k+1} \cdot z + a_k, \; for\; k = N-2, N-1, \ldots 0, \\ f(z) &= s_0. \end{aligned} \tag{9}$$

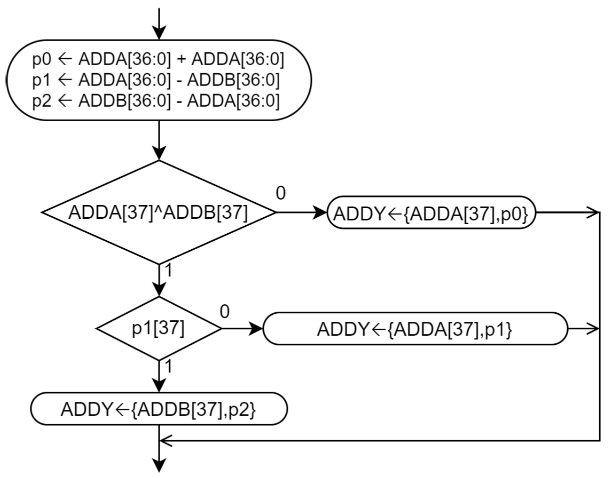
p0 ← ADDA[36:0] + ADDA[36:0]
p1 ← ADDA[36:0] - ADDB[36:0]
p2 ← ADDB[36:0] - ADDA[36:0]
ADDA[37]^ADDB[37]
0
1
ADDY←{ADDA[37],p0}
p1[37]
0
1
ADDY←{ADDA[37],p1}
ADDY←{ADDB[37],p2}

FIGURE 4. Diagram describing the behavior of the ADD block from Fig. 3.

For example, using (9) for $N = 6$, the polynomial can be expressed as follows:

$$f(z) = ((((a_5 \cdot z + a_4) \cdot z + a_3) \cdot z + a_2) \cdot z + a_1) \cdot z + a_0. \tag{10}$$

Similarly to the Chebyshev approximation, the values of the polynomial's coefficients can be computed using the Matlab (or Python) software.

Contrary to Chebyshev approximation, in the case of direct polynomial approximation, the fixed-point hardware implementation of the hyperbolic tangent function will be presented. A general architecture of the digital module implementing this approximation is depicted in Fig. 3. The architecture is very similar to the one from Fig. 1. However, the essential difference is the word length of subsequent signals. The other difference is that the addition and multiplication blocks are purely combinational circuits. Therefore, they do not require synchronization signals such as ADDND, ADDRDY, MULND, MULRDY, etc.

In order to obtain high calculations accuracy, the 32-bit fraction part of the fixed-point variables has been applied. Additionally, the sign-magnitude representation has been used. Therefore, for example, the total bit length of the input word (the X signal) is 37 bits, which consists of the 1 sign bit, 4 bits for the integer part, and 32 bit for the fraction part. Since the value of the hyperbolic tangent function is limited to the narrow $[-1.0, 1.0]$ interval, the Y output line needs only 1 bit for the integer part.

The fixed-point addition block (ADD) has two 38-bit operands (ADDA, ADDB) and a 39-bit result line (ADDY). The behavior of the ADD block is not straightforward. The details are described in Fig. 4 (similarly to the ASM, the diagram also applies Verilog HDL operators). The variables $p0 \ldots p2$ have 38 bits. The values of these variables are either the addition of the magnitude of the ADDA and ADDB operands or their subtraction. If the arithmetic signs of the operands (the most significant bit - 37) are the same, the magnitude of the whole addition result is the sum of the

TABLE 2. Truth table describing the RR block from Fig. 3.

Range of X	X[35-33]	X[32]	X[31]	X[30]	i	m
$0 \leq \lvert X\rvert < 0.25$	$000b$	0	0	0	0	$204h$
$0.25 \leq \lvert X\rvert < 0.5$	$000b$	0	0	1	1	$20Ch$
$0.5 \leq \lvert X\rvert < 0.75$	$000b$	0	1	0	2	$214h$
$0.75 \leq \lvert X\rvert < 1.0$	$000b$	0	1	1	3	$21Ch$
$1.0 \leq \lvert X\rvert < 1.25$	$000b$	1	0	0	4	$224h$
$1.25 \leq \lvert X\rvert < 1.5$	$000b$	1	0	1	5	$22Ch$
$1.5 \leq \lvert X\rvert < 2.0$	$000b$	1	1	x	6	$238h$
$2.0 \leq \lvert X\rvert < 2.5$	$001b$	0	0	x	7	$248h$
$2.5 \leq \lvert X\rvert < 3.0$	$001b$	0	1	x	8	$258h$
$3.0 \leq \lvert X\rvert < 3.5$	$001b$	1	0	x	9	$268h$
$3.5 \leq \lvert X\rvert < 4.0$	$001b$	1	1	x	10	$278h$
$4.0 \leq \lvert X\rvert < 5.0$	$010b$	0	x	x	11	$298h$
$5.0 \leq \lvert X\rvert < 6.0$	$010b$	1	x	x	12	$2B0h$
$6.0 \leq \lvert X\rvert < 8.0$	$011b$	x	x	x	13	$2E0h$
$8.0 \leq \lvert X\rvert < 10.0$	$100b$	x	x	x	14	$320h$
$\lvert X\rvert \geq 10.0$	other cases				15	$0h$

magnitudes of both operands (the $p0$ variable). Otherwise, depending on which operand has higher magnitude (the arithmetic sign of the $p1$ variable), the magnitude of the result is the subtraction of the magnitudes of the operands (either the $p1$ or $p2$ variable).

The MUL block is a typical fixed-point multiplier with 32-bit operands and 64-bit multiplication result. This block can be implemented using dedicated DSP blocks inside an FPGA.

The truth table describing the operation of the RR block is presented in Table 2. Based on the 6 most significant bits of the input vector (excluding the sign bit), the RR block designates the interval identifier (the i signal) as well as the value referring to the "$-b - a$" expression in (2), which constitutes the m signal line.

Similarly to the Chebyshev realization, the CM block, storing all polynomial coefficients, is implemented as a distributed read-only memory. Since the absolute values of all coefficients are less than 1.0 (they do not have integer parts), 33 bits suffice to represent their values (1 bit is needed for the arithmetic sign, and the following 32 bits represent the fraction part).

The ASM describing all operations performed by the CU block is presented in Fig. 5. After the activation of the ND signal in the S0 state, the operands values for the ADD block are determined. This operation is related to the calculation of the "$2 \cdot x - b - a$" expression in (2). The multiplication by 2 is obtained by the 1 bit shift left operation, which is equivalent to the specific part select operation within a concatenation operator for the ADDA operand. In the next state (S1), the addition result is ready. The arithmetic sign of the result is stored in the 1-bit w variable. In the same state, the final value of the 39-bit z variable is established depending on the denominator value in (2).

In the S2 state, the previously calculated value of the z variable is assigned to the MULA operand. The ADDB operand receives the value of the a_4 coefficient, which is related to the calculation of the "$a_5 \cdot z + a_4$" expression in (10). Since the ADDB (as well as ADDA) is a 38-bit variable and the word length of the a_4 coefficient is 33 bits, the five of the least significant bits of the ADDB are filled with zeros.

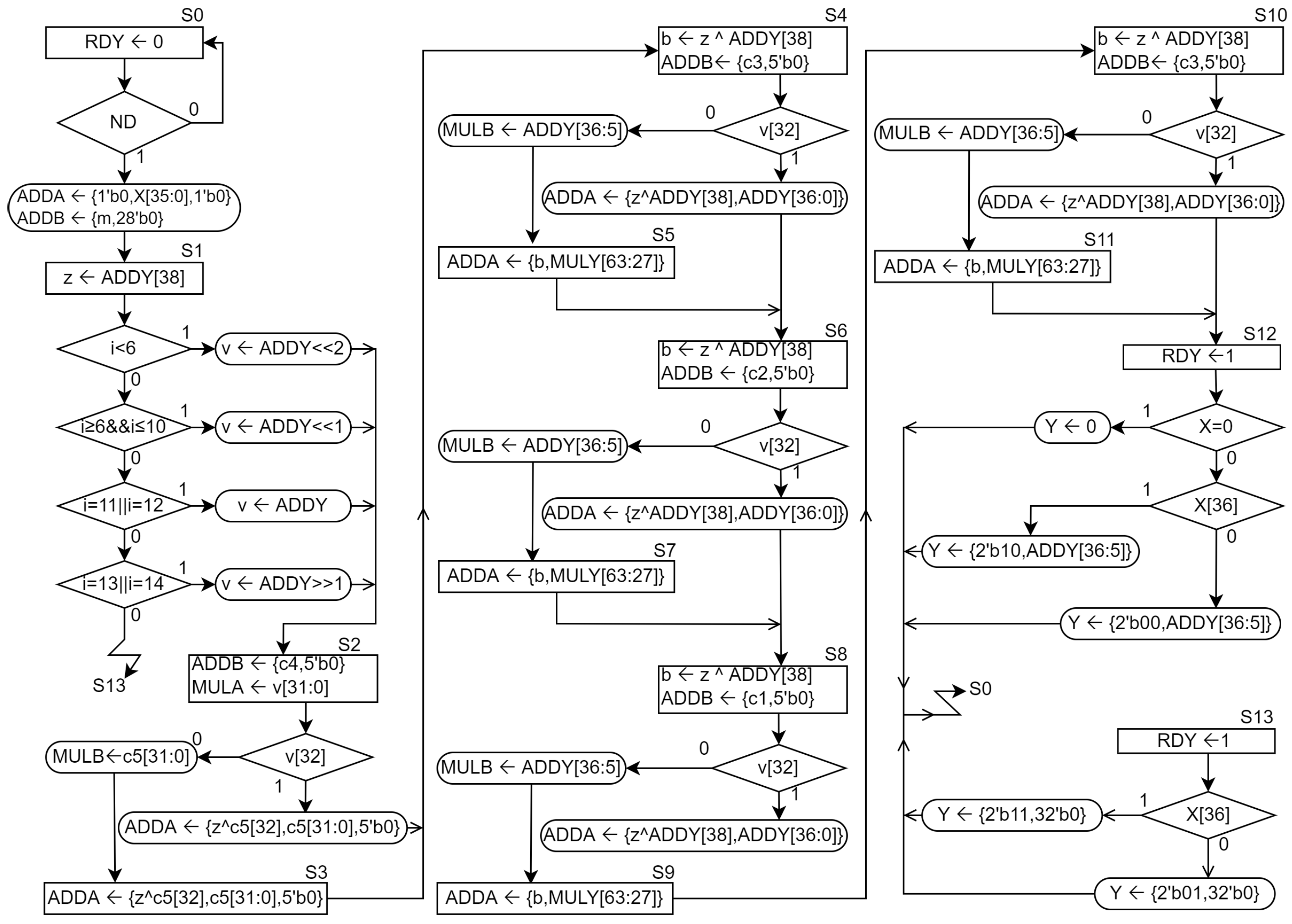

FIGURE 5. ASM diagram describing the control unit block from Fig. 3.

If the value of the z variable is equal to 1.0 (the thirty second bit of the z variable is 1), the value of the a_5 coefficient is directly assigned to the ADDA operand. Otherwise, the multiplication "$a_5 \cdot z$" is accomplished first, and then the multiplication result is assigned to the ADDA operand (only 37 the most significant bits of the multiplication result are taken into consideration).

In the S4 state, the expression "$a_5 \cdot z + a_4$" is finally calculated (the ADDY stores the result), and further calculations, such as the multiplication of the previous addition result by the z variable and the addition of the a_3 coefficient to the multiplication result, etc., are performed. Since the sign-magnitude representation of the fixed-point numbers has been applied, special care must be paid to the calculation of the arithmetic sign for the subsequent variables. For example, the 1-bit q variable, calculated in the S4 state as the XOR operation of the w variable and the sign bit (38) of the addition result, is used as the sign bit for the ADDA operand in the S5 state.

The final result of the calculations of the direct polynomial approximation of the hyperbolic tangent function is established in the S12 state. The RDY signal is activated for a single clock cycle in order to signal the end of calculations.

Although the presented ASMs from Fig. 2 and Fig. 5 relate to the specific case where $N = 6$ and 15 intervals are involved, it is worth noting that they can easily be altered to deal with other polynomial degree and different number of intervals.

III. IMPLEMENTATION RESULTS AND DISCUSSION

Taking advantage of the approximation methods described in the previous section, several miscellaneous implementations have been prepared and tested. Both single-precision floating-point and fixed-point implementations have been considered. These implementations also differ in the degree of polynomial and in the number of applied intervals. The obtained implementations results are featured in this section.

A. FLOATING-POINT IMPLEMENTATION RESULTS

Tables 3 and 4 show the number of obtained FPGA resources, achieved accuracy, number of required clock cycles, maximum clock frequency, and calculations time for the single-precision floating-point hyperbolic tangent function implementation. The 15 intervals introduced in the previous section have been applied for the approximation. For all cases considered in this section, the Artix-7 XC7a100t-

3csg384 FPGA chip was applied. The FPGA resources are indicated in terms of the number of look-up tables (LUTs) and flip-flops (FFs) necessary for the particular implementation. All implementations from Table 3 and 4 also require 4 FPGA-specific digital signal processing (DSP) blocks. Two of four of these blocks are used for the floating-point multiplier IP-core block. The other two DSPs are exploited by the adder IP-core block. The accuracy of the calculations is perceived here as the value of the maximum absolute difference between the reference value calculated by the Matlab "tanh()" function and the result of behavioral simulation of the Verilog code describing the considered implementations. The 1E6 values equally spaced within the [−10, 10] interval were used for accuracy calculations. The fifth column of Tables 3 and 4 indicates the number of clock cycles required to complete the output value. It was assumed that for all implementations from these Tables, the multiplier and adder IP-core blocks need 2 clock cycles in order to calculate the result. The maximum allowable clock frequency was obtained based on the post-route static timing report delivered by the FPGA implementation tool (the Xilinx ISE Design Suite was used). The default balanced optimization strategy was applied for the implementation tool. The calculations time was designated, in turn, as the multiplication of the minimum clock period time (the reciprocated value of the maximum clock frequency) and the number of clock cycles.

The data from Tables 3 and 4 indicate that implementations based on the Chebyshev approximation require a slightly higher number of FPGA resources. They can also be clocked with lower frequency, which entails longer calculations time (the number of clock cycles required for the calculation of the result is the same for both types of the approximation). However, up to the fifth polynomial degree, the accuracy of the implementations with the Chebyshev approximation is a little bit better than for the implementations with the direct polynomial approximation. Moreover, the software simulations results carried out using Matlab suggest that the accuracy of the Chebyshev approximation, for the polynomial degree equal to or higher than five, should be approximately as good as for the direct polynomial approximation. Unfortunately, probably due to the rounding errors of floating-point computations, particularly involving the floating-point arithmetic IP-cores form Xilinx, the actual accuracy of the FPGA implementations of the direct polynomial approximation is considerably better for these degrees of polynomial. The data also show that the increase of the polynomial degree beyond the 5 degree virtually does not improve the accuracy of the single-precision floating-point implementations of the hyperbolic tangent function.

Apart from the 15 basic intervals for which the domain of the approximated function is divided, the select number of other intervals has also been considered for the hyperbolic tangent function implementation. Only the direct polynomial approximation has been included for this consideration. As the data from Table 5 indicate, adding two additional intervals (i.e., [1.5, 1.75) and [2.0, 2.25)) for the third degree polynomial allows a noticeable increase of the accuracy. Surprisingly, in this case, the number of required LUTs is also lower than for the 15 intervals. A similar situation takes place for the polynomial of the second degree where the same 17 intervals were applied. Using the 19 intervals (the [0, 0.125) and [0.125, 0.375) intervals were added to the 17 previously established intervals) for the polynomial of the second degree causes a substantial increase of the accuracy, maintaining a similar number of LUTs in reference to the case where the basic number of intervals is involved.

Observing the obvious tendency of the increase of the accuracy along with the increase of the number of intervals,

TABLE 3. Implementation results of floating-point direct polynomial approximation for the 15 intervals.

Poly-nomial degree	Number of LUTs	Number of FFs	Accuracy	Number of clock cycles	Max clock frequency [MHz]	Calc. time [ns]
7	821	341	$5.895E-8$	46	130.3	352.9
6	781	341	$5.896E-8$	40	121.2	329.9
5	726	339	$6.656E-8$	34	125.5	270.9
4	672	339	$5.216E-7$	28	121.6	230.2
3	650	339	$2.121E-5$	22	129.0	170.4
2	564	338	$3.283E-3$	16	127.5	125.4

TABLE 4. Implementation results of floating-point chebyshev approximation for the 15 intervals.

Poly-nomial degree	Number of LUTs	Number of FFs	Accuracy	Number of clock cycles	Max clock frequency [MHz]	Calc. time [ns]
7	899	355	$1.192E-7$	46	112.5	408.7
6	829	350	$1.192E-7$	40	121.3	329.7
5	757	353	$1.192E-7$	34	116.9	290.8
4	744	347	$2.980E-7$	28	112.9	247.8
3	692	323	$1.180E-5$	22	113.9	193.0
2	566	288	$2.700E-4$	16	118.5	135.0

TABLE 5. Implementation results of floating-point direct polynomial approximation for different number of intervals.

Number of inter-vals	Poly-nomial degree	Number of LUTs /FFs	Accuracy	Number of clock cycles	Max clock frequency [MHz]	Calc. time [ns]
17	3	616/340	$9.353E-6$	22	129.8	169.4
17	2	553/339	$2.439E-4$	22	124.6	128.4
19	2	572/339	$8.630E-5$	16	121.9	131.2
59	3	744/341	$7.630E-8$	22	129.4	169.9
186	2	923/324	$1.213E-7$	16	124.7	128.3
214	2	951/341	$9.968E-8$	16	129.8	123.2
244	2	953/342	$8.498E-8$	16	126.8	126.2

TABLE 6. Implementation results of direct polynomial approximation for different parameters of floating-point arithmetic blocks.

No. of clock cycles	Number of LUTs	Number of FFs	Number of DSPs	Overall no. of clock cycles	Max clock frequency [MHz]	Calc. time [ns]
0	874	203	2	13	57.2	227.1
0	689	203	4	13	52.6	247.2
1	857	290	2	23	63.8	360.5
1	677	270	4	23	63.1	364.3
2	912	327	2	34	123.5	275.3
2	726	339	4	34	125.5	270.9
3	902	425	2	45	169.1	266.1
3	742	363	4	45	154.6	291.1
4	951	481	2	56	204.9	273.2
4	787	429	4	56	208.3	268.8

one could ask the question of how many intervals are needed for the polynomial of the third and second degree in order to achieve the accuracy similar to that of the polynomial of the fifth degree. In search of an answer to this question, the dedicated Matlab script has been prepared. Based on the required accuracy and the polynomial degree (the input parameters), this script automatically generates the intervals that comply with the rules described in the previous section. The script also directly generates the Verilog code describing the RR and CM blocks (Fig. 1, Fig. 3), which greatly facilitate the implementation.

Using this script and then conducting the behavioral simulation of the Verilog code describing the particular implementation, it turned out that the 59 intervals suffice for the 7.6E-8 accuracy (for the fifth polynomial degree, the accuracy amounts to 6.7E-8). This allows a significant reduction of the calculations time (170ns vs. 271ns) under virtually negligible increase of the number of required LUTs (744 vs. 726).

For the second degree of polynomial, the following three cases have been considered, namely the 186, 214, and 244 intervals. For the latter case, the accuracy reaches the 8.5E-8 value, and calculations time drops to 126ns (more than 2 times shorter). However, the number of required LUTs is noticeable higher (953 vs. 726) compared to the fifth polynomial degree with the basic 15 intervals.

For all of the implementations from Tables 3-5, the floating-point multiplier and adder blocks needed 2 clock cycles for the calculation of the result. Each block also utilized 2 DSPs. Table 6 illustrates the impact of the different number of clock cycles and DSP blocks on the required FPGA resources and overall calculations time for the implementation of the direct fifth degree polynomial approximation with the 15 intervals. Apart from the base case where each multiplier and adder block needed 2 DSPs (4 DSPs are required for an implementation), only the case where no DSP blocks are required for the adder block (2 DSPs are needed for an implementation).

As expected, if more DSP blocks are used, the particular implementation needs a lower number of LUTs (since an FPGA chip usually contains two times more FFs than LUTs, an attention can only be focused on the utilization of LUTs). However, it seems that the utilization of the DSPs by both of the arithmetic blocks does not noticeably improve the maximum clock frequency. For some cases, as a matter of fact, the maximum clock frequency for the 4 DSPs is even lower than for the 2 DSPs, which might be counterintuitive.

The maximum clock frequency is, in turn, strongly determined by the number of clock cycles needed by the arithmetic blocks. The more clock cycles are applied, the higher is the maximum clock frequency. However, the total number of clock cycles needed for the calculation of the result of the hyperbolic tangent function is also higher. Therefore, applying more than 2 clock cycles for the arithmetic blocks virtually does not improve the overall calculations time.

TABLE 7. Implementation results of fixed point direct polynomial approximation.

Poly-nomial degree	Number of LUTs /FFs/DSPs	Accuracy	Number of clock cycles	Calc. time [ns]	Max clock frequency [MHz]
8	690/161/8	$6.733E-11$	18	175.9	102.3
7	656/159/8	$4.020E-10$	16	157.7	101.4
6	558/152/4	$4.600E-9$	14	105.0	133.3
5	540/152/4	$3.922E-8$	12	91.2	131.5
4	526/152/4	$5.052E-7$	10	74.7	133.9
3	478/152/4	$2.123E-5$	8	59.7	134.0
2	532/188/4	$4.159E-4$	6	45.4	132.2

TABLE 8. Implementation results of fixed point chebyshev approximation.

Poly-nomial degree	Number of LUTs /FFs/DSPs	Accuracy	Number of clock cycles	Calc. time [ns]	Max clock frequency [MHz]
8	1055/334/8	$8.750E-11$	20	199.3	100.3
7	844/267/8	$2.158E-10$	18	194.2	92.7
6	684/269/4	$3.890E-9$	16	137.6	116.3
5	665/285/4	$1.925E-8$	14	123.3	113.5
4	656/234/4	$2.540E-7$	12	106.6	112.9
3	571/162/4	$1.200E-5$	10	77.0	129.8
2	550/173/4	$2.700E-4$	8	61.0	131.2

The overall shortest calculations time is achieved for the case where the arithmetic blocks are purely combinational. However, in this case, the maximum clock frequency drops to a very low value, which might not be acceptable in a typical synchronous system (e.g., the hyperbolic tangent block may unnecessary slow down the calculations speed of a whole ANN implementation).

It is also worth noting that the maximum clock frequency achieved by the floating-point implementations from Tables 3-5, where 2 clock cycles are chosen for the arithmetic blocks, is quite similar to the value attained by the fixed-point implementations.

B. FIXED-POINT IMPLEMENTATIONS RESULTS

Apart from floating-point, the fixed-point implementations of the hyperbolic tangent function have also been considered. The results of these implementations for the direct polynomial and Chebyshev approximation with the 15 intervals are presented in Tables 7-9. The implementations from Tables 7-8 involve signed-magnitude coding for the fixed-point representation, described in the previous section. For a comparison purpose, Table 9 presents, in turn, the implementations results of the Chebyshev approximation for the two's complement coding. It is important to note that for all implementations from Table 7-9, for the polynomial degrees 2…6, the word length of polynomial's coefficients amounted to 33 bits. However, in order to achieve higher accuracy for the 7th and 8th polynomial degrees, the word lengths of the coefficients had to be increased to 36 and 38 bits, respectively. For these two cases, the fraction length of the output word was also higher and accounted for 35 bits.

As the results suggest, the direct polynomial approximation implementations require a noticeable lower number of FPGA resources, both in terms of LUTs and FFs. However,

TABLE 9. Implementation results of fixed point chebyshev approximation for two's complement coding.

Poly-nomial degree	Number of LUTs /FFs/DSPs	Accuracy	Number of clock cycles	Calc. time [ns]	Max clock frequency [MHz]
8	771/331/8	$5.595E-11$	20	202.6	98.7
7	677/295/8	$2.141E-10$	18	171.6	104.9
6	633/247/4	$2.973E-9$	16	122.9	130.2
5	619/253/4	$1.951E-8$	14	106.2	131.8
4	548/255/4	$2.482E-7$	12	90.7	132.3
3	497/252/4	$1.177E-5$	10	75.2	132.9
2	462/216/4	$2.699E-4$	8	61.0	131.2

TABLE 10. Implementation results of fixed point direct polynomial approximation for different number of intervals.

Number of inter-vals	Poly-nomial degree	Number of LUTs /FFs/DSPs	Accuracy	Number of clock cycles	Calc. time [ns]	Max clock frequency [MHz]
17	3	527/152/4	$9.347E-6$	8	58.8	136.0
17	2	559/188/4	$2.439E-4$	6	46.0	130.4
19	2	595/189/4	$8.631E-5$	6	44.7	134.2
59	6	676/167/8	$7.500E-11$	14	131.1	106.8
59	5	610/151/8	$3.515E-10$	12	115.3	104.1
59	4	531/145/4	$4.122E-9$	10	74.3	134.5
59	3	514/145/4	$3.723E-8$	8	59.8	133.8
59	2	568/181/4	$1.080E-5$	6	44.8	133.9
186	6	1055/168/8	$7.034E-11$	14	137.4	101.9
186	5	958/151/8	$3.032E-10$	12	118.9	99.1
186	4	940/151/8	$1.351E-9$	10	97.5	102.6
186	3	811/145/4	$6.579E-9$	8	60.4	132.5
186	2	837/181/4	$6.937E-8$	6	44.1	136.0
244	6	1137/167/8	$1.376E-10$	14	135.6	103.5
244	5	1099/165/8	$1.376E-10$	12	119.2	100.6
244	4	992/151/8	$6.178E-10$	10	97.3	102.8
244	3	869/145/4	$3.237E-9$	8	60.2	132.9
244	2	905/181/4	$2.998E-8$	6	45.5	131.9

with an exception of the 8th polynomial degree for the signed-magnitude coding, the Chebyshev approximation allows a slightly higher accuracy to be achieved. Due to the fact that the implementations based on the Chebyshev approximation need 2 more clock cycles for the same polynomial degree, and the maximum clock frequency is also slightly lower, the overall calculations time is higher for these implementations.

It is worth noting that - as the data from Tables 8 and 9 show - the implementations with the two's complement coding require lower number of FPGA resources than the implementations with the signed-magnitude coding.

Similarly to the floating-point implementations, the impact of the select number of the other intervals on the results of fixed-point implementations has also been considered. The results for the direct polynomial approximation are presented in Table 10. As the data indicate, the highest accuracy is attained for the polynomial of the 6th degree with 186 and 59 intervals. Similar accuracy can be attained for the base 15 intervals and the 8th degree of the polynomial. However, for the case where the 59 intervals are involved, both the overall calculations time and the resources utilization are better than for the other cases with similar accuracy.

It is also a bit surprising that the increase of the number of intervals from 186 to 244 does not improve the accuracy for the polynomial of the 6th degree.

If the short calculations time of the hyperbolic tangent function would be an important factor, then again, the

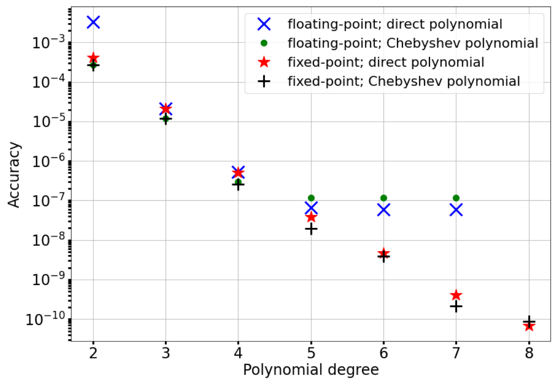

FIGURE 6. Calculations accuracy versus polynomial degree for the 15 intervals.

implementation with the 59 intervals and the polynomial of the 2nd degree can be the choice. A much higher accuracy with a very similar calculations time is attained, in fact, for the implementation with the 244 intervals. However, this implementation requires substantially higher number of FPGA resources.

Figure 6 presents the relation between the calculations accuracy and polynomial degree for the floating-point and fixed-point implementations, including the direct polynomial and Chebyshev approximations. A logarithmic scale was used on the ordinate axis. As seen, for the same number of intervals (i.e. the 15 intervals in this case), the increase of the polynomial degree entails the increase of the accuracy, regardless of the applied arithmetic and approximation type. However, for the floating-point implementations, the increase ends on the 5 polynomial degree, whereas for the fixed-point implementations a higher accuracy for a higher polynomial degree requires also the increase of the word-length for the used variables.

C. COMPARISON WITH OTHER WORKS

The main design goal of the proposed implementation methods of the hyperbolic tangent activation function was to achieve high calculations accuracy. Therefore, the accuracy (perceived as the maximum absolute error) constitutes an essential criterion for the comparison with other published results. Table 11 features such a comparison. The Table includes solutions with the highest accuracy that the authors were able to find. In fact, in [16] the maximum error, which is not explicitly given but can be read from the graph, drops to a very low value of around 2E-8 for single-precision floating-point arithmetic. However, it seems that the errors were calculated there based on software simulations instead of tangible hardware implementation simulations. The other work [15] did not revealed any implementation details (yet, reported accuracy reached the 1.9E-7 value). Therefore, these works were not included in Table 11.

As Table 11 suggests, the accuracy achieved by the proposed implementation method is slightly higher for the single-precision floating-point implementation and significantly higher for the fixed-point implementation. It is

TABLE 11. Comparison of the accuracy of the hyperbolic tangent function calculation for different implementations methods.

Method	Accuracy
LUT with linear approximation; fixed-point [1]	$1.60E-7$
PWL; floating-point [3]	$7.40E-3$
PWL; floating-point [4]	$2.18E-5$
Linear approximation; fixed-point [8]	$1.92E-4$
CORDIC; fixed-point [5]	$1.70E-7$
CORDIC; floating-point [13]	$2.47E-7$
DCT interpolation; double-precision floating-point [14]	$1.00E-5$
Direct with McLaurin series; floating-point [19]	$1.79E-7$
Proposed, direct polynomial approximation; floating-point	$5.89E-8$
Proposed, Chebyshev approximation; fixed-point	$5.59E-11$

worth noting that, for example, a high calculations accuracy was achieved for the proposed fixed-point implementation method for an essentially shorter word length than in [16], where the 64-bit precision was applied.

It is also important to note that the majority of the published solutions considered a shorter range of input values of the hyperbolic tangent function. With the exception of [19], the widest input range spanned from -8 to 8. The proposed implementation (as well as [19]) applies a wider range, namely $(-10, 10)$. It is worth noting that the usage of a wider input range usually entails a higher implementation cost (e.g., more FPGA resources are needed), and it is harder to achieve high calculations accuracy.

Apart from the calculations accuracy, the other implementation parameters include the calculations time and resources utilization. Unfortunately, these parameters are not always featured in published papers.

Moreover, existing solutions apply different FPGAs with different architectures, which have an important impact on the achieved calculations time and number of required FPGA resources. These two factors make the comparison between solutions more difficult. Nevertheless, a very short calculations time for the hyperbolic tangent function is reported in [14]. However, the accuracy of this solution is substantially lower than the proposed implementation and amounts to 1.0E-4 for the reported calculations time of less than $10ns$ (fixed-point arithmetic has been applied). The solution also needs a relatively high amount of bits of the internal Block RAM memory. A relatively short calculations time ($35ns$) is also the feature of [9]. Yet, the accuracy in this case is even lower than [14] and accounts for 1.5E-2.

The proposed implementations are definitely more effective than the solutions based on the CORDIC algorithm [5], [13] and the direct method with the McLaurin series [19], both in terms of the calculations time and resources utilization. For example, for [5], the calculations time amounts to $2.7\mu s$ and 1687 LUTs are required. For [19] these parameters accounted for $0.97\mu s$ and 1916 LUTs, respectively. The proposed solution requires, in turn, only 514 LUTs and produces the result in 60ns for fixed-point implementation with the accuracy even better than [5] (3.7E-8 vs. 1.7E-07). For floating-point implementation 744 LUTs are needed, and the calculations time accounts for 170ns under the 7.6E-8 accuracy, which is substantially better than for [19].

TABLE 12. Classification performance of 88 data records.

Activation function accuracy	Number of falsely classified data records
$1.0E-3$	46
$4.9E-4$	3
$2.4E-4$	1
$1.2E-4$ or higher	0

It is also worth noting that the accuracy of the single precision software implementation in the Visual C++ environment of the hyperbolic tangent function, measured using the same methodology, amounts to 4.86E-8. It is very similar to the accuracy obtained by the proposed FPGA implementation.

D. IMPACT OF THE ACTIVATION FUNCTION ACCURACY

The proposed FPGA implementation method of the hyperbolic tangent activation function allows very high calculations accuracy to be achieved. Based on the authors' previous works, the impact of the accuracy of the activation function on the classification performance of various neural networks has also been briefly examined.

In [22] the LSTM neural network performing a classification task for failure detection in the cold forging process [23], targeted for FPGA implementation, has been presented. The network consisted of two layers. The first layer included 44 LSTM cells, whereas the second contained a single neuron. The activation function was implemented taking advantage of the CORDIC based algorithm [13]. Various Verilog code simulations have been carried out in order to determine the accuracy of the classification of the previously developed LSTM network, depending on the accuracy of the activation function. The activation function accuracy was adjusted by an alteration of the number of the CORDIC algorithm iterations. The 88 test data records have been considered for the determination of the LSTM network classification accuracy. The results presented in Table 12 show that the 1.2E-4 activation function accuracy (or higher) is required for a classification without any error.

The other conducted experiment involved the training process of the same LSTM network. The outline of the training method of this LSTM, targeted for FPGA implementation, has already been briefly described in [24]. For the determination of the impact of the activation function accuracy on the training process, a Python script utilizing standard libraries has been prepared. For the LSTM training (the backpropagation trough time algorithm has been used), 176 data records - other than 88 previously applied records - have been exercised. The training process itself consisted of 15 epochs and 20 reiterations for each accuracy value of the activation function. Once the computations for the training process were completed, the validation was carried out with the 88 test data records and the classification accuracy was determined. For validation, the same activation function accuracy was applied as for LSTM training. The obtained classification accuracy of the LSTM network is

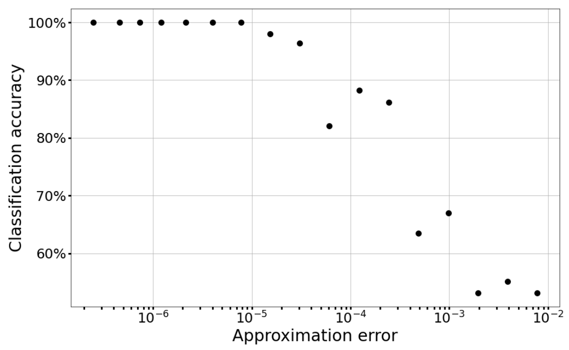

FIGURE 7. Relation of the activation function accuracy and classification accuracy.

presented in Fig. 7. The highest classification accuracy of the 20 aforementioned reiterations was depicted in the figure. As data indicate, 100% classification accuracy is achieved for the activation function accuracy that amounted to 7.75E-6 or higher. This is the accuracy value that is not often encountered for the implementation of the hyperbolic tangent function. A similar conclusion, namely that the activation function accuracy must be high for the correct implementation of the training algorithm (i.e., the backpropagation) for neural networks, is contained in [6]. However, the authors of this work applied an implementation of the activation function with the maximum absolute error accounted for 5E-4 (not so high accuracy).

The impact of the activation function accuracy has also been examined for the auto-associative neural network performing novelty detection in the milling process [25]. Another auto-associative neural network has also been developed and examined for the cold forging process [23]. Contrary to the previous findings, in both cases it turned out that the activation function accuracy is virtually negligible due to the fact that the arguments of the hyperbolic tangent function took values for which the function value was limited to 1.0 or −1.0. Similarly, a small impact of the hyperbolic tangent activation function implementation with the maximum absolute error around 5E-2 on the classification accuracy for the MNIST and Fashion-MNIST datasets has also been reported in [18].

The above considerations may indicate that the accuracy level of the activation function implementation, which is needed for a reliable solution of a problem, depends on the particular case and the processed data. Therefore, running the simulation of the impact of the activation function accuracy on the final results produced by ANNs seems to be useful in order to make a choice between different activation function implementation versions. However, there are also versatile implementation methods of feed-forward ANNs, such as [26], where the structure of the ANN (i.e. number of layers, number of neurons in subsequent layers, weights values, biases, etc.) does not have to be statically determined before FPGA synthesis and implementation process, but can be changed even during system operation by a simple alteration of the BlockRAM memory content. These implementations are able to deal with different ANNs for which the actual requirement of activation function accuracy may be unknown (potentially including a high accuracy requirement). Therefore, the accuracy of the activation function for these implementations should be high enough in order to meet the requirements for a wide range of applications. This is also the area where the proposed implementation might be useful.

IV. CONCLUSION

It has been shown that the direct polynomial or Chebyshev approximation allows for relatively simple and effective implementation of the hyperbolic tangent function in FPGAs. A very high calculations accuracy, relatively short calculations time, and moderate FPGA resources utilization can be achieved applying this implementation method.

The conducted experiments show that implementations based on direct polynomial approximation ensure lower FPGA resources utilization than implementations exercising the Chebyshev approximation. The latter implementations enable, in turn, slightly higher accuracy to be achieved for the same polynomial degree, particularly where fixed-point arithmetic is applied. An exception constitutes the floating-point implementations with the degree of polynomial equal to or higher than 5, where implementations based on the direct polynomial approximation feature considerably higher calculations accuracy.

Taking the number of intervals into account, the implementation results suggest that the increase of this number within a certain range (e.g. from 15 to 59) does not cause a relevant increase of the FPGA resources utilization. Therefore, in cases where the low resources utilization and the high accuracy constitute important factors, the 59 intervals would be recommended. When a short calculations time as well as high accuracy are needed, and the resources utilization is less important, the higher number of intervals (i.e., 186 or 244) might be the choice. However, for a higher number of intervals, the polynomial's coefficients can be stored in dedicated RAM blocks inside FPGA (if available), instead of utilizing LUTs as a distributed memory. This would allow the decrease of a number of required LUTs. It is also worth noting that applying a higher number of intervals does not deteriorate the timing of the circuit - the maximum allowable clock frequency remains almost unaffected, particularly for low degrees (i.e. 2 or 3) of polynomial.

As expected, the fixed-point implementations require a lower number of LUTs and FFs than the floating-point implementations of the polynomial of the same degree. They also enables considerably shorter (even more than two times) calculations time to be achieved. However, it turned out that the floating-point implementations are able to be clocked with higher clock frequency, particularly in the case where more clock cycles are applied for the floating-point arithmetic IP-core blocks.

The fixed-point implementations allow higher calculations accuracy to be achieved in comparison with single-precision floating-point implementations. It also turned out that the fixed-point implementations with two's complement coding seem to require a lower number of FPGA resources than the implementations with sign-magnitude coding.

It is worth noting that the obtained accuracy of the hyperbolic tangent function calculation for the proposed implementation method is very high in comparison with other published solutions. In fact, for the fixed-point implementations, the obtained accuracy is virtually the highest reported so far.

On account of a high calculations accuracy as well as a relatively short calculations time and moderate resources utilization, the proposed implementation method might be an effective alternative to the other methods of activation functions implementation, particularly CORDIC-based methods. The direct or Chebyshev polynomial approximation method may also be easily applied for the implementation of not only the other activation functions (e.g., sigmoid function) but also for various mathematical functions. In these cases, only the RR and CM blocks need to be adapted. The presented hardware architecture as well as the essential part of the ASM describing the control unit can remain unchanged.

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. Ind. Informat.*, vol. 7, no. 1, pp. 78–89, Feb. 2011.

[2] K. Chen, L. Huang, M. Li, X. Zeng, and Y. Fan, "A compact and configurable long short-term memory neural network hardware architecture," in *Proc. 25th IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2018, pp. 4168–4172.

[3] S. Bouguezzi, H. Faiedh, and C. Souani, "Hardware implementation of tanh exponential activation function using FPGA," in *Proc. 18th Int. Multi-Conf. Syst., Signals Devices (SSD)*, Mar. 2021, pp. 1020–1025.

[4] P. Ferreira, P. Ribeiro, A. Antunes, and F. M. Dias, "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function," *Neurocomputing*, vol. 71, nos. 1–3, pp. 71–77, Dec. 2007.

[5] V. Tiwari and N. Khare, "Hardware implementation of neural network with sigmoidal activation functions using CORDIC," *Microprocess. Microsyst.*, vol. 39, no. 6, pp. 373–381, 2015.

[6] F. Ortega-Zamorano, J. M. Jerez, D. U. Muñoz, R. M. Luque-Baena, and L. Franco, "Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 9, pp. 1840–1850, Sep. 2016.

[7] A. Armato, L. Fanucci, G. Pioggia, and D. De Rossi, "Low-error approximation of artificial neuron sigmoid function and its derivative," *Electron. Lett.*, vol. 45, no. 21, pp. 1–2, 2009.

[8] A. Armato, L. Fanucci, E. P. Scilingo, and D. De Rossi, "Low-error digital hardware implementation of artificial neuron activation functions and their derivative," *Microprocess. Microsyst.*, vol. 35, no. 6, pp. 557–567, Aug. 2011.

[9] H. M. Al-Rikabi, M. A. Al-Ja'afari, A. H. Ali, and S. H. Abdulwahed, "Generic model implementation of deep neural network activation functions using GWO-optimized SCPWL model on FPGA," *Microprocessors Microsyst.*, vol. 77, Sep. 2020, Art. no. 103141.

[10] T. Orlowska-Kowalska and M. Kaminski, "FPGA implementation of the multilayer neural network for the speed estimation of the two-mass drive system," *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 436–445, Aug. 2011.

[11] I. del Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled accuracy approximation of sigmoid function for efficient FPGA-based implementation of artificial neurons," *Electron. Lett.*, vol. 49, no. 25, pp. 1598–1600, 2013.

[12] S. Baraha and P. K. Biswal, "Implementation of activation functions for ELM based classifiers," in *Proc. Int. Conf. Wireless Commun., Signal Process. Netw. (WiSPNET)*, Mar. 2017, pp. 1038–1042.

[13] G. R. Dec, "LSTM cell implementation on FPGAs," *Parallel Process. Lett.*, vol. 31, no. 2, Jun. 2021, Art. no. 2150011.

[14] A. M. Abdelsalam, J. M. P. Langlois, and F. Cheriet, "A configurable FPGA implementation of the tanh function using DCT interpolation," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 168–171.

[15] D. Baptista and F. Morgado-Dias, "Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks," *Neural Comput. Appl.*, vol. 23, nos. 3–4, pp. 601–607, Sep. 2013.

[16] B. Pasca and M. Langhammer, "Activation function architectures for FPGAs," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 43–437.

[17] I. Koyuncu, "Implementation of high speed tangent sigmoid transfer function approximations for artificial neural network applications on FPGA," *Adv. Electr. Comput. Eng.*, vol. 18, no. 3, pp. 79–86, 2018.

[18] F. M. Shakiba and M. Zhou, "Novel analog implementation of a hyperbolic tangent neuron in artificial neural networks," *IEEE Trans. Ind. Electron.*, vol. 68, no. 11, pp. 10856–10867, Nov. 2021.

[19] Z. Hajduk, "Hardware implementation of hyperbolic tangent and sigmoid activation functions," *Bull. Polish Acad. Sci., Tech. Sci.*, vol. 66, no. 5, pp. 563–577, 2018.

[20] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin, Germany: Springer, 2007.

[21] W. Press, W. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*. Cambridge, U.K.: Cambridge Univ. Press, 1992.

[22] G. R. Dec, "FPGA-based neural net for failures prediction in the cold forging process," *Parallel Process. Lett.*, vol. 32, nos. 1–2, Mar. 2022, Art. no. 2150023.

[23] T. Zabinski, T. Maczka, J. Kluska, M. Kusy, Z. Hajduk, and S. Prucnal, "Failures prediction in the cold forging process using machine learning methods," in *Artificial Intelligence and Soft Computing*, L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh and J. M. Zurada, Eds. Cham, Switzerland: Springer, 2014, pp. 622–633.

[24] G. R. Dec, "FPGA-based learning acceleration for LSTM neural network," *Parallel Process. Lett.*, early access, doi: 10.1142/S0129626423500019. Unfortunately, this publication still seems to be in an "early access" stage. Therefore, we are unable to provide detailed bibliographic parameters.

[25] T. Zabinski, Z. Hajduk, J. Kluska, and L. Gniewek, "FPGA-embedded anomaly detection system for milling process," *IEEE Access*, vol. 9, pp. 124059–124069, 2021.

[26] Z. Hajduk, "Reconfigurable FPGA implementation of neural networks," *Neurocomputing*, vol. 308, pp. 227–234, Sep. 2018.

ZBIGNIEW HAJDUK received the Ph.D. degree in computer engineering from the University of Zielona Góra, Poland, in 2006, and the D.Sc. degree from the Czestochowa University of Technology, Poland, in 2019.

He is currently an Associate Professor with the Department of Computer and Control Engineering, Rzeszów University of Technology. His research interest includes digital systems design with FPGAs.

GRZEGORZ RAFAŁ DEC received the B.S. degree in automation and robotics from the Rzeszów University of Technology, Poland, in 2016, and the M.Sc. degree in electrical engineering from the Lublin University of Technology, Poland, in 2017. He is currently pursuing the Ph.D. degree in computer science with the Rzeszów University of Technology.

He is currently working on his Ph.D. thesis with Rzeszów University of Technology. His research interests include control theory, data science, and digital systems design.