IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received February 17, 2020, accepted February 29, 2020, date of publication March 4, 2020, date of current version March 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2978399

A Universal Approximation Method and Optimized Hardware Architectures for Arithmetic Functions Based on Stochastic Computing

ZIDI QIN[1], YUOU QIU[1], MUHAN ZHENG[1], HONGXI DONG[1], ZHONGHAI LU[2], (Senior Member, IEEE), ZHONGFENG WANG[1], (Fellow, IEEE), AND HONGBING PAN[1]

[1]School of Electronic Science and Engineering, Nanjing University, Nanjing 210023, China
[2]School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 10044 Stockholm, Sweden

Corresponding author: Hongbing Pan (phb@nju.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 61376075 and Grant 41412020201.

ABSTRACT Stochastic computing (SC) has been applied on the implementations of complex arithmetic functions. Complicated polynomial-based approximations lead to large hardware complexity of previous SC circuits for arithmetic functions. In this paper, a novel piecewise approximation method based on Taylor series expansion is proposed for complex arithmetic functions. Efficient implementations based on unipolar stochastic logic are presented for the monotonic functions. Furthermore, detailed optimization schemes are provided for the non-monotonic functions. Using NAND and AND gates as main computing elements, the optimized hardware architectures have extremely low complexity. The experimental results show that a broad range of arithmetic functions can be implemented with the proposed SC circuits, and the mean absolute errors can achieve the order of 1×10^{-3}. Compared with the state-of-the-art works, the approximation precision for some typical functions can be increased by more than $8\times$ with our method. In addition, the proposed circuits outperform the previous methods in hardware complexity and critical path significantly.

INDEX TERMS Stochastic computing, arithmetic functions, approximation, VLSI architecture.

I. INTRODUCTION

Stochastic computing (SC) is an attractive approximate computing method which performs basic computing operations based on probabilities [1]–[3]. Using very simple and low-cost arithmetic units, SC circuits have been exploited in many applications, such as neural networks [4]–[8], image processing [9] and low-density parity check (LDPC) decoding [10]. However, the computation based on probabilities can cause performance degradation, so SC is suitable for applications with low requirement of numerical precision.

SC uses the probability of one's occurrence in a bit-stream to represent the value of a number, and processes random bit-streams called *stochastic numbers* (SNs) instead of deterministic numbers. To represent a real number x, a stochastic number generator (SNG) is required to produce a random sequence X of length K, and there are K_1 ones in X. In unipolar format, the value of x equals $p_x = K_1/K$ and $x \in [0, 1]$, where p_x is the probability of ones in the sequence X. In bipolar format, the value of x is represented with $2p_x - 1$ and $x \in [-1, 1]$.

Since traditional binary arithmetic is converted into probability calculation of SNs, complex arithmetic operations can be performed with very simple logic circuits in SC [11]. In unipolar format, the multiplication of two SNs can be implemented by using a single AND gate instead of a complex multiplier. Additions can be performed by multiplexers (MUXs), but the outputs from MUXs are scaled results which have some precision loss. A NOT gate is used to implement $1 - x$ in unipolar format. A NAND gate can perform

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

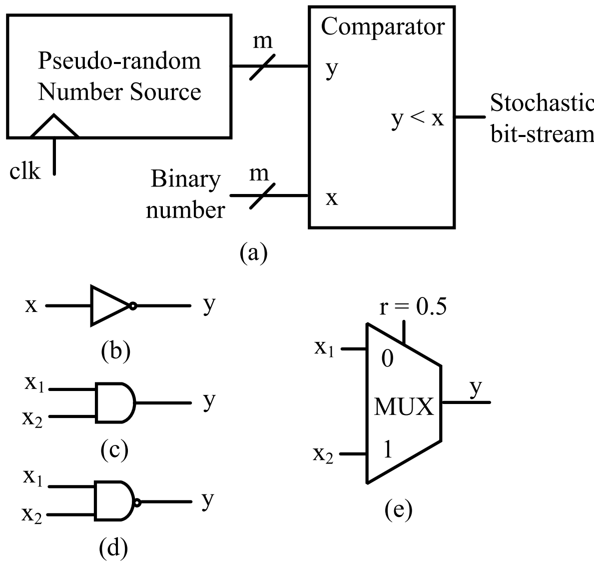


FIGURE 1. Basic components in SC circuits. (a) stochastic number generator (b) NOT gate: $y = 1 - x$ (c) AND gate: $y = x_1x_2$ (d) NAND gate: $y = 1 - x_1x_2$ (e) Multiplexer: $y = (x_1 + x_2)/2$.

$y = 1 - x_1x_2$ in unipolar format. These simple logic computing elements can significantly save hardware resources and power consumption in digital systems.

Transforming deterministic numbers into random bit-streams [12], [13], SNGs are important components in SC circuits. The hardware implementations of SNGs can influence the computing accuracy and hardware complexity of SC systems [14]. A typical architecture of an SNG is shown in Fig.1 (a). By comparing an m-bit binary number x with a series of m-bit random numbers, an SNG can produce an SN of length N. A linear feedback shift register (LFSR) and a comparator are the basic components in an SNG. The LFSR can generate pseudo random numbers for comparisons with x. In every clock cycle, if x is bigger than a pseudo random number, 1 is generated as one bit of the output. To represent the computing results with high precision, long bit-streams are usually required, which can lead to long computing latency.

Taking advantage of the above-mentioned features of SC, complex arithmetic functions can be implemented by using low-complexity circuits. In [11], based on finite-state machines (FSMs), the stochastic implementation method for exponential and hyperbolic tangent functions has been proposed. In [15], [16], FSM-based method has been further explored to synthesize more types of arithmetic functions. Higher approximation precision can be achieved by increasing the state numbers of FSMs. However, the hardware complexity increases in proportion to the state numbers. Bernstein polynomials have been employed for the approximation and stochastic implementations of arithmetic functions [17], [18]. The approximation accuracy of this method is related to the degrees of Bernstein polynomials. Higher degrees can improve approximation accuracy, but lead to a significant increase in hardware complexity. On the basis of truncated Maclaurin series polynomials and Horner's rule, complex arithmetic functions can be implemented with multiple levels of NAND gates [19]. The method in [19], however, requires complicated polynomial expansions or factorizations, which leads to large hardware complexity. Recently, [4], [20] used stochastic circuits to implement piecewise linear (PWL) approximation of arithmetic functions. To implement the functions with simpler stochastic circuits, [20] proposed to transform the approximated linear functions by analyzing the values of parameters in different segments. However, the method in [20] is only suited for a narrow range of functions, because the transformations are constrained by the values of parameters.

This paper provides a novel approximation method and low-complexity hardware architectures for arithmetic functions. The main contributions of this paper are as follows.

1) On the basis of piecewise approximations, truncated Taylor expansion is used to formulate the approximate computation of arithmetic functions. To simplify the hardware implementations, universal equations based on unipolar format are further developed for different kinds of functions.

2) Using NAND and AND gates as the basic computing components, very low-complexity hardware architectures are provided for different kinds of arithmetic functions.

3) The optimized implementation schemes for non-monotonic functions are investigated in detail. An efficient scheme for improving the approximation precision is further presented. We have verified the proposed method on a wide range of arithmetic functions, some of which are investigated for the first time.

The rest of this paper is organized as follows. Section II introduces the proposed method for arithmetic functions approximation in SC. Section III provides the efficient hardware architectures for arithmetic functions. Experimental results are introduced in Section IV. Comparisons with previous works are presented in Section V. Finally, we conclude this paper in Section VI.

II. PROPOSED APPROXIMATION METHOD FOR ARITHMETIC FUNCTIONS BASED ON STOCHASTIC LOGIC

In this section, the basic approximation method for arithmetic functions is introduced. On the basis of truncated Taylor expansion and piecewise approximation, an approximation scheme is derived for continuously differentiable arithmetic functions. Then stochastic implementation schemes are further provided for the functions using stochastic unipolar format.

A. THE BASIC APPROXIMATION METHOD

Assume that a continuously differentiable function $f(x)$ is approximated by $f^*(x)$, and the input x is in the interval $[\alpha, \beta]$. Firstly, we divide input range $[\alpha, \beta]$ into S equal sub-intervals. Thus, the length of each sub-interval is $h = |\alpha - \beta|/S$. In $[\alpha, \beta]$, the first sub-interval is denoted by $[\alpha_0, \alpha_1]$ and the ith sub-interval is denoted by $[\alpha_{i-1}, \alpha_i]$, where i is the index of a sub-interval, $i = 1, \ldots, S$.

In the unipolar format, the input range and output range in SC are both [0, 1], so the functions to be approximated should satisfy these conditions. In the following, we only discuss the functions owning the input range and output range of [0, 1].

In the PWL method [20], the original function is approximated by a linear function in each segment. The straightforward way is to determine the approximated linear function as $y = a_i x + b_i$, where i denotes the ith segment. The detailed scheme of how to determine parameters of the functions is introduced in [20].

We propose a novel approach based on truncated Taylor expansion to approximate the curves in each sub-interval. According to the Taylor series expansion, $f(x)$ can be expanded at $x = x_0$ as follows:

$$f(x) = f(x_0) + f'(x_0)\triangle x + \frac{1}{2!}f''(x_0)(\triangle x)^2 + \ldots + \frac{1}{n!}f^n(x_0)(\triangle x)^n, \quad (1)$$

where $\triangle x = x - x_0$.

In a small interval, a function can be approximated by Taylor series expansion with high precision. When S is large, the length of a sub-interval can be very small. For instance, when S is 16 for the range [0, 1], the length of a sub-interval is 0.0625. Higher order expansion can provide better approximation performance, but can also increase hardware complexity significantly. Plus, when $\triangle x$ is small, the high order terms in Eq. (1) can only bring trivial improvement in precision.

To achieve a balance between hardware complexity and precision, we use the first order Taylor expansion in a small sub-interval. Then $f^*(x)$ in a sub-interval can be expressed as follows:

$$f^*(x) = f(x_0) + f'(x_0)\triangle x = f(x_0) + f'(x_0)(x - x_0). \quad (2)$$

In a sub-interval $[\alpha_{i-1}, \alpha_i]$, we set $x_0 = \alpha_{i-1}$, and put $\alpha_{i-1} = (i-1) \times h$ and $\alpha_i = i \times h$ into Eq. (2):

$$\begin{aligned} f^*(x) &= f(\alpha_{i-1}) + f'(\alpha_{i-1})\triangle x \\ &= f(\alpha_{i-1}) + f'(\alpha_{i-1})(x - \alpha_{i-1}) \\ &= f((i-1)h) + f'(\alpha_{i-1})(x - (i-1)h). \end{aligned} \quad (3)$$

For an arbitrary input x, as long as index i is determined, Eq. (3) can be used for approximating $f(x)$. The corresponding value of index i can be determined by Eq. (4):

$$i = \lfloor x/h \rfloor + 1, \quad (4)$$

where $\lfloor x/h \rfloor$ is the integer part of x/h and is denoted by p.

Furthermore, when the length of $[\alpha_{i-1}, \alpha_i]$ is small, the segment in this sub-interval can be seen as a straight line. Thus, the gradient $f'(x)$ can be approximated as a constant. We propose to approximate the gradient $f'(\alpha_{i-1})$ by the following equation:

$$f'(\alpha_{i-1}) \approx \frac{f(\alpha_i) - f(\alpha_{i-1})}{h} \approx \frac{f(ih) - f((i-1)h)}{h}. \quad (5)$$

Then Eq. (3) can be transformed into:

$$f^*(x) = f((i-1)h) + \frac{f(ih) - f((i-1)h)}{h}(x - (i-1)h). \quad (6)$$

As $i = p + 1$, Eq. (6) can be further simplified into:

$$f^*(x) = f(ph) + (f((p+1)h) - f(ph))(\frac{x}{h} - p). \quad (7)$$

Then we denote that

$$\lambda = f(ph), \quad (8)$$

$$\mu = f((p+1)h) - f(ph), \quad (9)$$

and

$$q = \frac{x}{h} - p, \quad (10)$$

where q is the decimal part of x/h and $p = 0, 1, \ldots, S-1$. Thus, Eq. (7) can be expressed by

$$f^*(x) = \lambda + \mu \times q, \quad (11)$$

where the values of λ, q and $|\mu|$ belong to [0, 1]. Eq. (11) is the basic approximation equation. However, it cannot be implemented with unipolar stochastic logic directly, because the value of μ can be negative when $f((p+1)h) < f(ph)$. The monotonicity of $f(x)$ in [0, 1] determines whether the value of μ is positive or not. Thus, we further propose optimized implementation schemes according to the monotonicity of target function in [0, 1].

B. UNIVERSAL IMPLEMENTATIONS FOR MONOTONIC FUNCTIONS BASED ON STOCHASTIC COMPUTING

In input range [0, 1], target functions can be divided into three types, monotonically decreasing functions, monotonically increasing functions and non-monotonic functions. Thus, approximation equations for these three kinds of functions are presented in the following.

1) MONOTONICALLY DECREASING FUNCTIONS

Assume that $f(x)$ is a decreasing function, and we further transform Eq. (7) into:

$$f^*(x) = f(ph)[1 - (1 - \frac{f((p+1)h)}{f(ph)})(\frac{x}{h} - p)]. \quad (12)$$

Since $(p+1)h > ph, f((p+1)h) < f(ph)$. It can be derived that $f((p+1)h)/f(ph)$ and $1 - f((p+1)h)/f(ph)$ both belong to range (0, 1]. We denote $\phi = 1 - f((p+1)h)/f(ph)$ and then can get

$$f^*(x) = \lambda \times (1 - \phi \times q). \quad (13)$$

In SC circuits, $1 - \phi \times q$ can be implemented with only one NAND gate, and the multiplication between $1 - \phi \times q$ and λ can be performed with only one AND gate in unipolar format.

When the number of segments, S, is determined, λ and ϕ are fixed values, so they can be computed in advance. Plus, to simplify the implementation of $f^*(x)$, S can be chosen as 2^m, where m is a positive integer. Thus, $x/h = x \times 2^m$ can be computed with shift operations in hardware implementations.

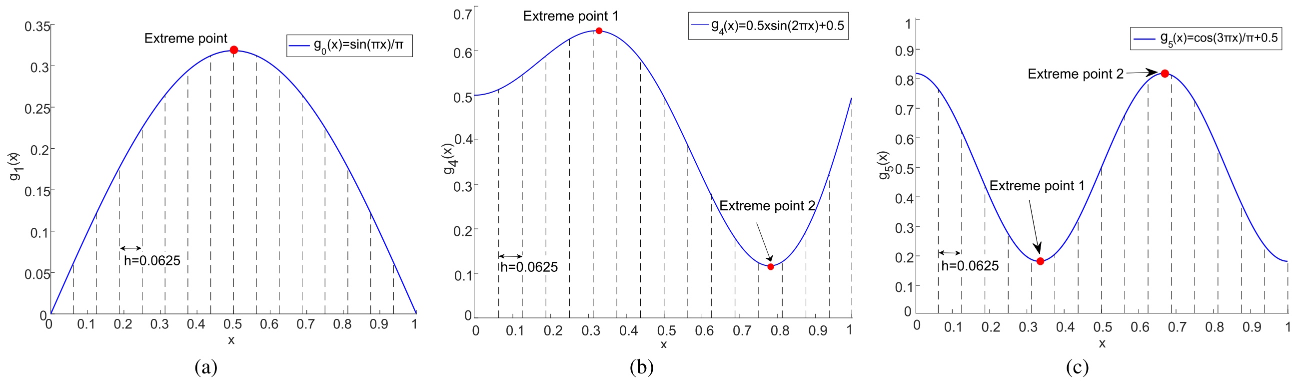


FIGURE 2. Examples of non-monotonic functions. (a) $g_0(x) = sin(\pi x)/\pi$ and its extreme point (b) $g_4(x) = 0.5xsin(2\pi x) + 0.5$ and its extreme points (c) $g_5(x) = cos(3\pi x)/\pi + 0.5$ and its extreme points.

The number of segments determines the approximating precision of the proposed method. The more segments are used, the higher precision can be achieved. For most functions, 8 or 16 segments are balanced choices between precision and hardware complexity.

2) MONOTONICALLY INCREASING FUNCTIONS

For increasing functions, Eq. (13) cannot be directly used, because $\phi < 0$ and $f(x)$ cannot be computed with unipolar format. To compute $f^*(x)$ with unipolar format, we propose some transformation to Eq. (2):

$$f^*(x) = f(x_0) + f'(x)\triangle x = f(x_0) - f'(x)(x_0 - x), \quad (14)$$

where $\triangle x$ is substituted with $-\triangle x$. In a sub-interval $[\alpha_{i-1}, \alpha_i]$, we set x_0 as α_i and we can get

$$\begin{aligned} f^*(x) &= f(\alpha_i) - f'(x)(\alpha_i - x) \\ &= f(\alpha_i) - \frac{f(\alpha_i) - f(\alpha_{i-1})}{h}(\alpha_i - x). \end{aligned} \quad (15)$$

Then put $a_i = ih$, $i = p + 1$ and $q = x/h - p$ into Eq. (15) and we can get

$$f^*(x) = f((p+1)h) - [f((p+1)h) - f(ph)](1 - q). \quad (16)$$

To make the computation of Eq. (16) suitable for unipolar format, we further do some transformation as follows:

$$f^*(x) = f((p+1)h)[1 - (1 - \frac{f(ph)}{f((p+1)h)})(1 - q)], \quad (17)$$

where $f((p+1)h)$ is denoted by λ', $1 - f(ph)/f((p+1)h)$ is denoted by ϕ'. Then we can get

$$f^*(x) = \lambda' \times (1 - \phi' \times (1 - q)), \quad (18)$$

where $\phi' \in (0, 1]$, $\lambda' \in [0, 1]$ and $q \in [0, 1)$.

In SC circuits, $1 - q$ can be implemented with a NOT gate. $1 - \phi' \times (1 - q)$ and the multiplication of $1 - \phi' \times (1 - q)$ with λ' can be performed with a NAND gate and an AND gate, respectively. When the number of segments, S, is determined, λ' and ϕ' are fixed values, which can be computed as parameters in hardware implementations.

C. IMPLEMENTATION SCHEMES FOR NON-MONOTONIC FUNCTIONS BASED ON STOCHASTIC COMPUTING

Non-monotonic functions have more than one monotonic intervals in [0, 1]. When dealing with a non-monotonic function, its extreme points should be found first to determine the monotonic intervals, where coordinates of the extreme points are denoted by $\epsilon_0(x_0, y_0), \epsilon_1(x_1, y_1), \ldots, \epsilon_j(x_j, y_j)$. Then choose a suitable length of h and divide the range [0, 1] into $S = 2^m$ segments. Notably, the length of a sub-interval should be shorter than that of a monotonic interval. Next, observe the positions of the extreme points. The abscissa of an extreme point may locate on the dividing points between two sub-intervals or inside a sub-interval. Here, we denote the functions owning no extreme points inside a sub-interval as Type I, and the functions owning extreme points inside a sub-interval as Type II.

The examples of several non-monotonic functions are shown in Fig. 2, where the input range [0, 1] is divided into 16 segments. $g_0(x) = sin(\pi x)/\pi$ has an extreme point $(0.5, g_0(x))$, and the abscissa of $(0.5, g_0(x))$ is an integer multiple of $1/S = 1/16$. Thus, $g_0(x)$ belongs to Type I. $g_4(x) = 0.5xsin(2\pi x) + 0.5$ and $g_5(x) = cos(3\pi x)/\pi + 0.5$ belong to Type II, because they all have extreme points inside the sub-intervals, which are shown in Fig. 2 (b) and (c). In the following, the detailed approximation schemes are discussed with some examples of Type I and Type II.

1) SCHEMES FOR NON-MONOTONIC FUNCTIONS OF TYPE I

For functions of Type I, the curve in every sub-interval is monotonic, so Eq. (13) or Eq. (18) can be applied in each sub-interval according to the monotonicity flexibly. We take some non-monotonic functions as examples to illustrate the method in detail. The investigated functions include $g_0(x) = sin(\pi x)/\pi$, $g_1(x) = 0.25sin(2\pi x/3)$, $g_2(x) = xln(0.5x) + 1$, $g_3(x) = cos(\pi x - 0.6)/\pi + 0.4$.

First, we take $g_0(x) = sin(\pi x)/\pi$ as an example to explain how to use the proposed method. $g_0(x)$ is an increasing function in the input range [0, 0.5) and a decreasing function in the

input range [0.5, 1]. Thus, we can use Eq. (18) in the range [0, 0.5) and Eq. (13) in the range [0.5, 1]. Then $g_0(x)$ can be approximated by the following function:

$$g_0^*(x) = \begin{cases} \lambda' \times (1 - \phi' \times (1 - q)), & 0 \le x < 0.5, \quad (19a) \\ \lambda \times (1 - \phi \times q), & 0.5 \le x \le 1. \quad (19b) \end{cases}$$

In hardware implementations, the computations in the range [0, 0.5) and [0.5, 1] can reuse the same NAND gates and AND gates, so Eq. (19) requires nearly the same hardware complexity with the monotonic functions. $g_0(x)$ is symmetric with respect to $x = 0.5$, so λ equals λ', and ϕ equals ϕ'. Therefore, only half of the parameters are required in hardware implementation.

The computations of $g_1(x) = 0.25sin(2\pi x/3)$ and $g_2(x) = xln(0.5x) + 1$ are similar to $g_0(x)$, but $g_1(x)$ and $g_2(x)$ have different monotonic intervals. $g_1(x)$ is an increasing function in [0, 0.75) and a decreasing function in [0.75, 1]. Therefore, the approximation function of $g_1(x)$ is as follows:

$$g_1^*(x) = \begin{cases} \lambda' \times (1 - \phi' \times (1 - q)), & 0 \le x < 0.75, \quad (20a) \\ \lambda \times (1 - \phi \times q), & 0.75 \le x \le 1. \quad (20b) \end{cases}$$

$g_2(x) = xln(\frac{1}{2}x) + 1$ is a decreasing function in [0, 0.75) and an increasing function in [0.75, 1]. Therefore, the approximation function of $g_2(x)$ is as follows:

$$g_2^*(x) = \begin{cases} \lambda \times (1 - \phi \times q), & 0 \le x < 0.75, \quad (21a) \\ \lambda' \times (1 - \phi' \times (1 - q)), & 0.75 \le x \le 1. \quad (21b) \end{cases}$$

The hardware implementations of $g_1(x)$ and $g_2(x)$ are similar to $g_0(x)$, but the values of λ, ϕ, λ' and ϕ' cannot be reused. Assume that $S = 16$. When $p = 0, 1 \ldots 11$, $g_1(x)$ and $g_2(x)$ are approximated by Eq. (20a) and Eq. (21a), respectively; When $p = 12, 13 \ldots 15$, $g_1(x)$ and $g_2(x)$ are approximated by Eq. (20b) and Eq. (21b), respectively. As a result, multiplexers are required to choose $1-q$ or q according to the value of p in hardware implementations. The detailed hardware architectures are provided in Section III.

2) SCHEMES FOR NON-MONOTONIC FUNCTIONS OF TYPE II

For functions of Type II, the sub-intervals owning extreme points are not monotonic intervals. We provide two available schemes to deal with this case. As shown in Fig. 3 (a), Scheme I approximates the special sub-interval with a monotonic function directly.

For example, $g_3(x) = cos(\pi x - 0.6)/\pi + 0.4$ has an extreme point $[0.19, g_3(0.19)]$. When $S = 16$ and $h = 0.0625$, 0.19 is not the integer multiple of h, so $g_3(x)$ belongs to Type II. $p = \lfloor 0.19/h \rfloor = 3$, so the extreme point locates in the range $[3h, 4h]$. Then compare the values of $g_3(3h)$ and $g_3(4h)$. $g_3(3h) > g_3(4h)$, so we approximate the segment in $[3h, 4h]$ with a decreasing function. As a result, $g_3(x)$ is approximated as an increasing function in [0, 0.1875)

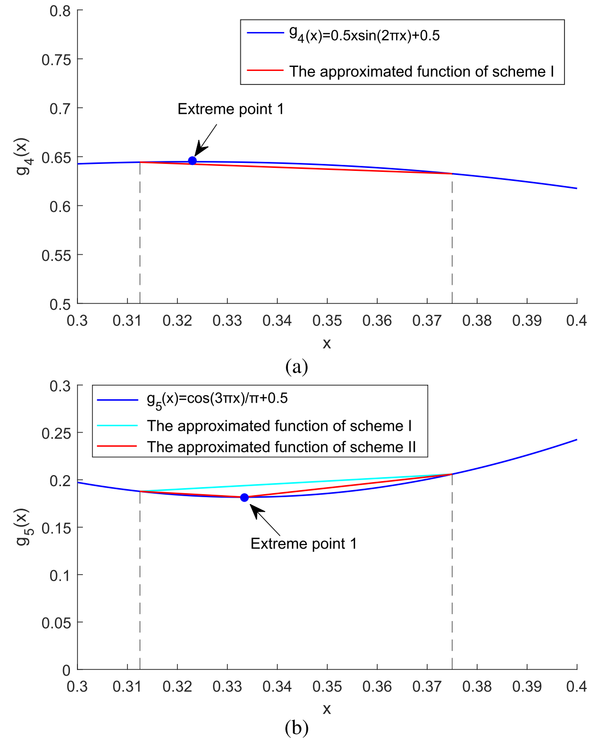


FIGURE 3. Proposed schemes for non-monotonic functions of type II. (a) Scheme I for the example function (b) Scheme II for the example function.

and a decreasing function in [0.1875, 1]. The approximation function of $g_3(x)$ is as follows:

$$g_3^*(x) = \begin{cases} \lambda' \times (1-\phi' \times (1 - q)), & 0 \le x < 0.1875, \quad (22a) \\ \lambda \times (1-\phi \times q), & 0.1875 \le x \le 1. \quad (22b) \end{cases}$$

Then we give another example: $g_4(x) = 0.5xsin(2\pi x) + 0.5$. $g_4(x)$ has three monotonic intervals in [0, 1], which is shown in Fig. 2 (b). When $S = 16$ and $h = 0.0625$, the extreme points locate in $[5h, 6h]$ and $[12h, 13h]$. Then we approximate the segments in $[5h, 6h]$ and $[12h, 13h]$ with decreasing functions, because $g_4(5h) > g_4(6h)$ and $g_4(12h) > g_4(13h)$. Therefore, $g_4(x)$ can be approximated by

$$g_4^*(x) = \begin{cases} \lambda \times (1 - \phi \times q), & 0 \le x < 0.375, \quad (23a) \\ \lambda' \times (1-\phi' \times (1-q)), & 0.375 \le x < 0.75, \quad (23b) \\ \lambda \times (1 - \phi \times q), & 0.75 \le x \le 1. \quad (23c) \end{cases}$$

Applying Scheme I for functions of Type II, the hardware implementations of $g_3(x)$ and $g_4(x)$ are similar to functions of Type I.

3) SCHEME II: OPTIMIZATION SCHEME FOR NON-MONOTONIC FUNCTIONS OF TYPE II

Scheme I is a direct way to implement functions of Type II. However, in some sub-intervals, the gradient of a function changes fast and one segment may be not enough to achieve

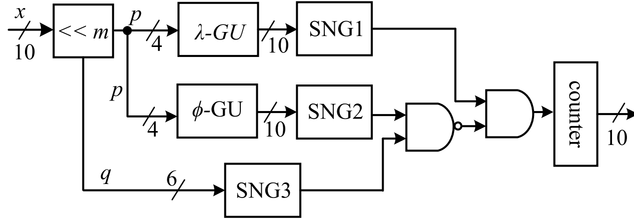


FIGURE 4. Arch A: Proposed architecture for monotonically decreasing functions (16 segments).

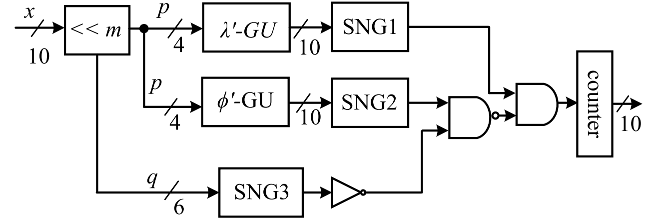


FIGURE 5. Arch B: Proposed architecture for monotonically increasing functions (16 segments).

the required approximation precision. In addition, increasing the total numbers of segments of the whole input range will lead to larger hardware complexity. As a result, to improve the approximation precision, we propose to partition this kind of sub-intervals into more segments and further approximate the segments with the proposed equations.

For example, $g_5(x) = cos(3\pi x)/\pi + 0.5$ is a function of Type II and has three monotonic intervals in [0, 1]. As shown in Fig. 2 (c), when $S = 16$ and $h = 0.0625$, an extreme point $(0.33, g_5(0.33))$ is in the sub-interval $[5h, 6h]$. As shown in Fig. 3 (b), adopting Scheme I in $[5h, 6h]$ can bring much approximation errors, so we use Scheme II to improve the approximation precision and partition the interval according to the position of the extreme point. Then the segment in $[5h, 0.33)$ is approximated as a decreasing function and the segment in $[0.33, 6h]$ is approximated as an increasing function. In $[10h, 11h]$, there is an another extreme point, $[0.66, g_5(0.66)]$. Scheme II is also applied in this sub-interval and we can get the approximation function of $g_5(x)$ as follows:

$$g_5^*(x) = \begin{cases} \lambda \times (1 - \phi \times q), & 0 \le x < 0.33, \quad (24a) \\ \lambda' \times (1 - \phi' \times (1 - q)), & 0.33 \le x < 0.66, \quad (24b) \\ \lambda \times (1 - \phi \times q), & 0.66 \le x \le 1. \quad (24c) \end{cases}$$

Compared with Scheme I, the hardware implementation of $g_5(x)$ requires additional two segments, which leads to slightly larger hardware complexity. The detailed architecture of Scheme II is presented in Section III-C.

III. THE PROPOSED HARDWARE ARCHITECTURES

In this section, the hardware architectures for the three kinds of arithmetic functions are introduced in detail.

A. HARDWARE ARCHITECTURE FOR DECREASING FUNCTIONS

For decreasing functions, the architecture denoted by Arch A is shown in Fig. 4, where 16 segments are used. Firstly, according to Eq. (12), a shift operation is performed on the input value x to compute $\frac{x}{h} = x \times 2^m$, where the shift bits are decided by the number of segments.

Next, the result of $x \times 2^m$ is divided into an integer part p and a decimal part q, where p is a 4-bit number and q is a 6-bit number. According to Eq. (13), p is sent to λ-generating unit (λ-GU) and ϕ-generating unit (ϕ-GU) to generate corresponding λ and ϕ, respectively. In the generating units, each bit of the outputs is computed by the logic expressions of input bits, and implemented with combinational logic circuits. Then λ, ϕ and q are sent to SNGs to generate stochastic bit-streams.

The NAND gate is used to implement the expression $y = 1 - \phi \times q$. Then the AND gate is adopted to implement the multiplication between $1 - \phi \times q$ and λ. Finally, the bit-stream from the AND gate will be converted to a 10-bit binary number by a counter.

B. HARDWARE ARCHITECTURE FOR INCREASING FUNCTIONS

For increasing functions, the architecture denoted by Arch B is shown in Fig. 5. The overall architecture to implement Eq. (17) is similar to that of decreasing functions. λ'-generating unit (λ'-GU) and ϕ'-generating unit (ϕ'-GU) are used to generate corresponding λ' and ϕ', respectively. The only difference is that the bit-stream of q should be sent to a NOT gate to realize the computation of $1 - q$. Then the NAND gate is used to implement the expression $y = 1 - \phi' \times (1 - q)$.

C. HARDWARE ARCHITECTURES FOR NON-MONOTONIC FUNCTIONS

In this section, we present the hardware architectures for non-monotonic functions. Respective architectures are developed for implementations with Scheme I and II.

1) HARDWARE ARCHITECTURE FOR SCHEME I

For Scheme I, the hardware architecture is denoted by Arch C and shown in Fig. 6 (a). The input x is first processed by a shift operation to compute $x/h = x \times 2^m$, where the integer part and the decimal part of $x \times 2^m$ are p and q, respectively. p is sent to λ/λ'-generating unit (λ/λ'-GU) and ϕ/ϕ'-generating unit (ϕ/ϕ'-GU) to compute corresponding λ/λ' and ϕ/ϕ', respectively. p denotes the sub-intervals in which x is located, so λ/λ' and ϕ/ϕ' are generated according to the value of p.

To generate a selecting signal *Sel* for the MUX, p is sent to the selection-generating unit (Sel-GU). *Sel* is generated according to the value of p and is used for choosing q or $1-q$. When $Sel = 1$, the output of the MUX is q. When $Sel = 0$, the output of the MUX is $1 - q$.

For example, according to Eq. (19), the computing of $sin(\pi x)/\pi$ uses different equations in the range $[0, 0.5)$ and the range $[0.5, 1]$, respectively. As the point $(0.5, g(0.5))$ is

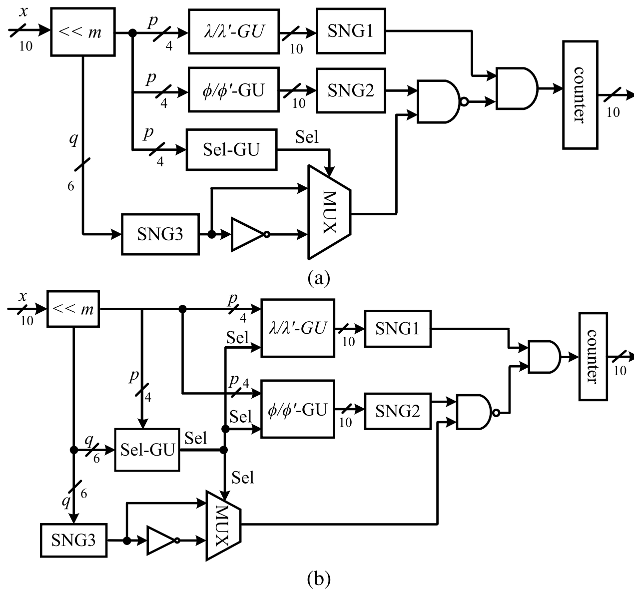


FIGURE 6. Proposed architectures for non-monotonic function (16 segments). (a) Arch C: architecture based on Scheme I (b) Arch D: architecture based on Scheme II.

TABLE 1. Logic expressions of *Sel* for different functions. p_3 is the MSB of *p*.

Functions	Sel
$g_0(x)$	$=\bar{p_3}$
$g_1(x)$	$=p_3p_2$
$g_2(x)$	$=\overline{p_3p_2}$
$g_3(x)$	$=p_3+p_2+p_1p_0$
$g_4(x)$	$=p_3\bar{p_2}\bar{p_1}+p_3\bar{p_2}+\bar{p_3}p_2p_0+\bar{p_3}p_2p_1$

the symmetric point of $sin(\pi x)/\pi$, the first eight sub-intervals are in range $[0, 0.5)$, when $S = 16$. $p[p_3 : p_0]$ is a 4-bit number, where p_3 is the most significant bit (MSB). Then p_3 can be used to determine whether x is in the range $[0, 0.5)$ or not. If MSB of p is 0, x is in the range $[0, 0.5)$; If MSB of p is 1, x is in the range $[0.5, 1]$. Notably, only half of the parameters are required to be generated because of the symmetric characteristics of $sin(\pi x)/\pi$, which can reduce the hardware resources.

For different functions, the main difference is the design of Sel-GU, and the structure of Sel-GU should be designed specially. The logic expressions of Sel for $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ are presented in Table 1. As can be seen, Sel can be generated by simple combinational logic of different bits of p.

2) HARDWARE ARCHITECTURE FOR SCHEME II

For Scheme II, the proposed hardware architecture is denoted by Arch D and shown in Fig. 6 (b). Compared with Arch C, the main differences are the designs of Sel-GU and generating units. Take $g_5(x) = cos(3\pi x)/\pi + 0.5$ as an example. As described in Section II-C3, the sub-intervals, $[5h, 6h]$ and $[10h, 11h]$, have extreme points: $(0.33, g_5(0.33))$ and $(0.66, g_5(0.66))$. Two more segments are required for the sub-intervals, so there are total 18 segments in the input range, when $S = 16$. In the Sel-GU, the Sel signal can be generated according to the values of p and q. If $p = 5$ and $q < 0.015625$, $Sel = 1$; If $p = 5$ and $q > 0.015625$, $Sel = 0$, where $0.015625 = 0.328125 - 5h$. Similarly, if $p = 10$ and $q < 0.03125$, $Sel = 0$; If $p = 10$ and $q > 0.03125$, $Sel = 1$, where $0.03125 = 0.65625 - 10h$. When $Sel = 1$, the output of the MUX is q. When $Sel = 0$, the output of the MUX is $1 - q$. Next, the values of λ/λ' and ϕ/ϕ' are generated according to the values of p and Sel in the generating units.

IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results of the proposed approximation method and corresponding hardware implementations based on SC.

To prove the universality of the proposed method, we have investigated multiple kinds of arithmetic functions. We have selected representative arithmetic functions which are decreasing functions, increasing functions or non-monotonic functions in the input range $[0, 1]$. If outputs of the investigated functions belong to $[0, 1]$, the functions can be implemented with the unipolar format directly. If the outputs of a function do not locate inside $[0, 1]$, the function can be scaled into range $[0, 1]$ by simple multiplications and additions, and the scaled version of the function can be implemented with the proposed method. For example, the values of $cos(\pi x)$ are in range $[-1, 1]$, so we implement the scaled version of $cos(\pi x)$: $0.5cos(\pi x)+0.5$. Scaled versions are also used for other functions such as $arccos(x)$, $sinh(x)$, $sec(x)$, $cosh(x)$, $arcsin(x)$, $arctan(x)$ and so on.

MATLAB is used to simulate the proposed hardware implementations, where hardware truncation errors are taken into consideration. In our simulations, the inputs of target functions and other parameters (ϕ, ϕ', λ and λ') are all 10-bit numbers. The simulation results of precision are evaluated with output mean absolute error (MAE) results, which are acquired by using Monte Carlo experiments for different inputs. For each input, 1000 Monte Carlo runs were performed. A 10-bit LFSR is used and the length of stochastic bit streams is 1024.

The investigated functions are implemented by the proposed architectures which are presented in Section III. The hardware architectures have been implemented in Verilog HDL and synthesized using the Synopsys Design Compiler (DC) under the TSMC 40 *nm* CMOS technology. A frequency of 500 *MHz* has been achieved.

The approximation precision, hardware complexity, critical path delay and power results for different kinds of functions are shown in Table 2, 3 and 4. The hardware complexity is evaluated by the cell area which is given in terms of equivalent numbers of 2-input NAND gates. All SNGs and counters are included in synthesis, and we also synthesize the architecture of an SNG. The cell area, critical path delay and power of an SNG are 101, 0.79 *ns* and $36\mu W$, respectively.

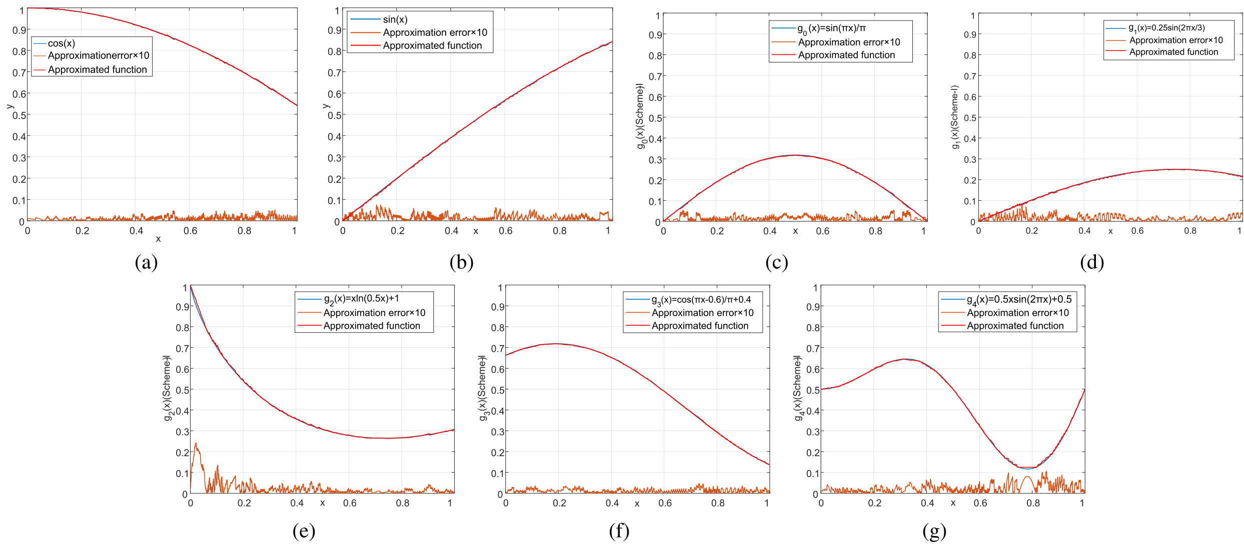


FIGURE 7. Simulation results for example functions, where the number of segments is 16. (a) Simulation results of $cos(x)$ (b) Simulation results of $sin(x)$ (c) Simulation results of $g_0(x) = sin(\pi x)/\pi$ (d) Simulation results of $g_1(x) = 0.25sin(2\pi x/3)$ (e) Simulation results of $g_2(x) = xln(0.5x) + 1$ (f) Simulation results of $g_3(x) = cos(\pi x - 0.6)/\pi + 0.4$ (g) Simulation results of $g_4(x) = 0.5xsin(2\pi x) + 0.5$.

TABLE 2. MAE results and hardware implementation results of decreasing functions using the proposed method. The functions are implemented with Arch A.

Decreasing Functions	No. of Segments	MAE	Cell Area	Delay (ns)	Power (μw)
cos(x)	8	0.0026	379	0.91	125.41
	16	0.0017	423	0.94	120.24
$0.5cos(\pi x)$ $+0.5$	8	0.0056	391	0.93	126.56
	16	0.0030	446	0.93	127.18
$0.5cos(1.5x)$ $+0.5$	8	0.0026	387	0.93	128.76
	16	0.0019	423	0.94	126.88
$0.5arccos(x)$	8	0.0069	386	0.93	127.16
	16	0.0026	453	0.94	125.64
e^{-x}	8	0.0027	357	0.94	114.73
	16	0.0020	383	0.93	122.35
e^{-2x}	8	0.0038	360	0.92	115.50
	16	0.0024	387	0.88	117.00
$e^{(-1.5x-5)}$	8	0.00061	336	0.92	111.89
	16	0.00053	335	0.94	110.84

A. ANALYSIS OF MONOTONIC FUNCTIONS

Table 2 shows the approximation precision and corresponding synthesis results of decreasing functions. The functions in Table 2 are all implemented with Arch A in Fig. 4. It can be seen that using 16 segments for approximation can achieve slightly higher precision than using 8 segments. We take $cos(x)$ as an example to show the simulation results in Fig. 7 (a), where 16 segments are used in the simulation, and the absolute error is magnified by $10\times$.

Table 3 shows the approximation precision and corresponding synthesis results of increasing functions. The functions in Table 3 are all approximated with Eq. (18) and implemented with Arch B in Fig. 5. Similarly, using 16 segments for approximation can improve the precision slightly compared with using 8 segments. The simulation results of $sin(x)$ is shown in Fig. 7 (b), where 16 segments are used in the simulation, and the absolute error is magnified by $10\times$.

As can be seen from Table 2 and 3, three SNGs occupy about 303 cells, which dominates the hardware complexity of the proposed architectures. For most of the functions, the hardware complexity increases when 16 segments are used, because more parameters require to be generated. The number of segments, however, does not have clear influence on the critical path delay and dynamic power. The critical path delays of the functions with different segments are almost the same. The dynamic power of the implementations using 8 segments are usually slightly lower than those using 16 segments.

B. ANALYSIS OF NON-MONOTONIC FUNCTIONS

Table 4 shows the approximation precision and corresponding synthesis results of non-monotonic functions. As described in Section II-C, we approximate the functions including $g_0(x)$, $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ with Scheme I, and implement them with Arch C in Fig. 6 (a). $g_5(x)$ is approximated with Scheme II, and implemented with Arch D in Fig. 6 (b). The simulation results of $g_0(x)$, $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ are shown in Fig. 7 (c), (d), (e), (f) and (g), respectively. The simulation results of $g_5(x)$ is shown in Fig. 8 (a). For comparisons, we give the simulation result of $g_5(x)$ using Scheme I in Fig. 8 (b). In the simulation, 16 segments are used and the absolute error is magnified by $10\times$.

It can be seen from Table 4 and Fig. 7, increasing the number of segments leads to significant improvement in approximation precision of the non-monotonic functions, especially for functions such as $g_4(x)$ and $g_5(x)$. As shown

TABLE 3. MAE results and hardware implementation results of increasing functions using the proposed method. The functions are implemented with Arch B.

Increasing Functions	No. of Segments	MAE	Cell Area	Delay (ns)	Power (μw)
$sin(x)$	8	0.0036	387	0.93	120.50
	16	0.0027	450	0.93	127.00
$sin(1.5x)$	8	0.0041	402	0.93	128.19
	16	0.0028	481	0.93	135.25
$0.5sinh(x)$	8	0.0032	412	0.88	127.39
	16	0.0026	490	0.93	130.63
$sec(x)-1$	8	0.0049	397	0.93	120.36
	16	0.0026	472	0.93	129.14
$cosh(x)-1$	8	0.0032	402	0.93	125.57
	16	0.0023	473	0.94	129.37
$0.5arcsin(x)$	8	0.0068	413	0.94	126.22
	16	0.0032	478	0.93	122.65
$0.5arctan(x)$	8	0.0035	388	0.93	119.11
	16	0.0026	463	0.89	128.82
$x^{2.5}$	8	0.0045	392	0.94	120.60
	16	0.0027	453	0.93	126.05
$\sqrt{x}$	8	0.0105	425	0.93	120.91
	16	0.0050	461	0.93	128.53
$0.2\sqrt{x+3}$	8	0.0013	405	0.93	126.89
	16	0.0026	497	0.93	132.91
$ln(1+x)$	8	0.0034	398	0.93	127.50
	16	0.0026	447	0.92	127.00
$tanh(x)$	8	0.0035	384	0.93	116.50
	16	0.0026	435	0.93	129.00
$tanh(4x)$	8	0.0059	393	0.92	121.22
	16	0.0024	473	0.93	128.02
$tanh(1.5x+0.5)$	8	0.0030	388	0.93	126.70
	16	0.0020	456	0.94	131.31
$sigmoid(x)$	8	0.0022	353	0.93	114.94
	16	0.0016	369	0.94	122.54
$e^{(x-1)}$	8	0.0027	364	0.94	124.50
	16	0.0020	359	0.94	124.00

in Fig. 8, the proposed Scheme II can improve the approximation precision of $g_5(x)$ in the sub-intervals owning an extreme point. For $g_5(x)$ based on Scheme I, the MAE and max absolute error in $[5h, 6h]$ are 0.0071 and 0.1559, respectively. When scheme II is used, the MAE and max absolute error in $[5h, 6h]$ can be reduced to 0.0068 and 0.1334, respectively. As a result, Scheme II is an efficient way to improve the precision. If higher precision is required, Scheme II with more segments can be used in the intervals owning large approximation errors.

As can be seen from Table 4, different non-monotonic functions have similar hardware complexity, and the hardware complexity increases slightly, when more segments are used. Compared with other functions, the hardware complexity of $g_5(x)$ based on Scheme II is the largest, but the critical path delay and dynamic power have little changes.

TABLE 4. MAE results and hardware implementation results of non-monotonic functions using the proposed method. $g_5(x) = \frac{1}{\pi}cos(3\pi x) + \frac{1}{2}$ is implemented with Arch D and the other functions are implemented with Arch C.

Non-monotonic Functions	No. of Segments	MAE	Cell Area	Delay (ns)	Power (μw)
$g_0(x) = sin(\pi x)/\pi$	8	0.0040	369	0.93	121.00
	16	0.0025	406	0.94	116.50
$g_1(x) = \frac{1}{4}sin(\frac{2\pi x}{3})$	8	0.0025	385	0.87	128.40
	16	0.0018	424	0.93	132.57
$g_2(x) = xln(\frac{x}{2})+1$	8	0.0075	391	0.93	129.51
	16	0.0032	441	0.92	130.41
$g_3(x) = \frac{1}{\pi}cos(\pi x - \frac{3}{5}) + \frac{2}{5}$	8	0.0046	380	0.94	130.24
	16	0.0025	429	0.92	130.50
$g_4(x) = \frac{1}{2}xsin(2\pi x) + \frac{1}{2}$	8	0.0114	398	0.94	131.94
	16	0.0041	437	0.93	135.14
$g_5(x) = \frac{1}{\pi}cos(3\pi x) + \frac{1}{2}$	8	0.0206	404	0.93	127.43
	16	0.0068	451	0.94	136.28

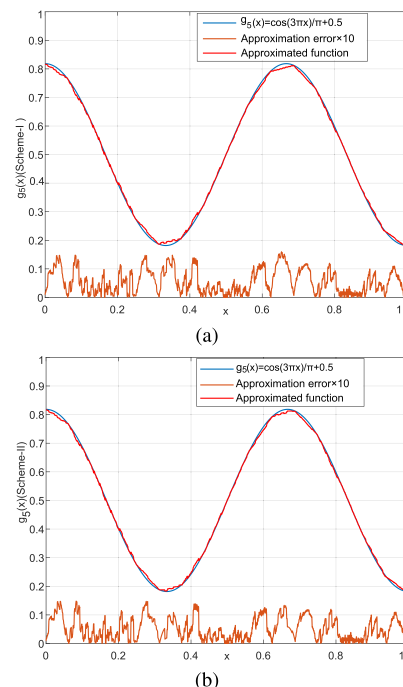


FIGURE 8. Comparisons between Scheme I and Scheme II. (a) Simulation results of $g_5(x)$ using Scheme I (b) Simulation results of $g_5(x)$ using Scheme II.

V. COMPARISONS WITH RELATED WORKS

In Section IV, we have investigated many functions to prove the universality of the proposed approximation method and

TABLE 5. MAE results of stochastic implementations for arithmetic functions using the proposed method, PWL-based method, Horner's Rule-based method and FSM-based method.

Types	Functions		Proposed		PWL [20]		Horner's Rule [19]	FSM [19], [21]
			8 segments	16 segments	8 segments	16 segments		
Decreasing Functions (Arch A)	cos(x)	order	-	-	-	-	6	8-state
		Error	0.0026	0.0017	0.0087	0.0063	0.0023	0.0053
	e^{-x}	order	-	-	-	-	6	8-state
		Error	0.0027	0.0020	0.010	0.0064	0.0008	0.0154
	e^{-2x}	order	-	-	-	-	6	8-state
		Error	0.0038	0.0024	0.0766	0.0678	0.0009	0.0423
Increasing Functions (Arch B)	sin(x)	order	-	-	-	-	7	8-state
		Error	0.0036	0.0027	0.0013	0.0012	0.0034	0.0025
	ln(1+x)	order	-	-	-	-	7	8-state
		Error	0.0034	0.0026	0.0026	0.0011	0.0081	0.0186
	tanh(x)	order	-	-	-	-	7	8-state
		Error	0.0035	0.0026	0.0012	0.0012	0.0140	0.0351
	tanh(4x)	order	-	-	-	-	6	8-state
		Error	0.0059	0.0024	-	-	0.0191	0.0046
	sigmoid(x)	order	-	-	-	-	5	8-state
		Error	0.0022	0.0016	0.0043	0.0012	0.0046	0.0198
	e^{x-1}	order	-	-	-	-	-	8-state
		Error	0.0027	0.0020	-	-	-	0.0163
Non-monotonic Functions (Arch C)	sin(πx)/π	order	-	-	-	-	11	8-state
		Error	0.0040	0.0025	0.0288	0.0288	0.0487	0.4716

hardware architectures. Functions which can be realized by the proposed method are not limited in the functions in Section IV. Some of the functions have been implemented in existing works, but some of them have not been mentioned in previous works. We use the implementation results of the functions which have been investigated in the existing works to make fair comparisons. The implementation results of these functions should be enough for us to make sufficient comparisons. In this section, we compare the experimental results of performance test and hardware implementations with previous works. Results of PWL-based method [20], Horner's Rule-based method [19] and FSM-based method [15], [16] are used for comparisons.

A. APPROXIMATION PERFORMANCE COMPARISONS

Table 5 shows the comparisons with previous methods in terms of MAE results. In general, the MAE results of the proposed method are around 1×10^{-3}, which can be considered as high-precision approximations for many applications.

Compared with FSM-based method, the approximation precision of our method is higher for most of the functions except for $sin(x)$. For $sin(\pi x)/\pi$, the precision can be improved by 188× with our method. As described in [15], the precision of FSM-based method can be improved by increasing state numbers. However, the hardware complexity will increases in proportion to the state numbers of FSM. The 8-state FSM is a balanced choice between accuracy and hardware complexity. Thus, compared with FSM-based method with 8 states, high-precision approximations can be implemented with less hardware complexity.

For $ln(1+x)$, $tanh(x)$, $tanh(4x)$, $sigmoid(x)$ and $sin(\pi x)/\pi$, the proposed method achieves higher approximation precision than Horner's Rule-based method [19]. In Horner's Rule-based method, some arithmetic functions can be implemented by using multiple levels of NAND gates based on Horner's rule. This method, however, has its limitations. For example, functions like $tanh(4x)$ cannot be implemented directly using Maclaurin expansion, when a is greater than 1. Thus, $tanh(4x)$ is implemented based on the circuits of e^{-2ax} and a JK flip-flop [19], leading to larger approximation errors. The precision of $tanh(4x)$ can be improved by 8× with the proposed method.

For $sin(x)$, $ln(1+x)$ and $tanh(x)$, the approximation precision of PWL-based method is slightly higher than our method. For functions such as e^{-2x} and $sin(\pi x)/\pi$, the precision can be increased by 25× and 11.5×, respectively. In [20], to avoid using the scaled version of additions, the approximated function, $f^*(x) = a_i x + b_i$, are transformed according to the relationships of a_i and b_i. However, the transformations are limited by the relationships of a_i and b_i. For example, when the absolute value of a_i is much larger than b_i, the implementations of approximated functions require subtractions, which can cause more precision loss. As a result, the method in [20] has poor approximation precision for e^{-2x}. Compared with PWL-based method, our method has universal equations rather than analyzing the parameters of each segment. In addition, the proposed method can implement high-precision approximations for more kinds of functions.

B. HARDWARE COMPLEXITY AND CRITICAL PATH DELAY COMPARISONS WITH RELATED WORKS

Table 6 shows the hardware complexity, delay and power comparisons with other works. The hardware complexity is

TABLE 6. Hardware complexity and critical path delay of stochastic implementations for arithmetic functions using the proposed method, PWL-based method, Horner's Rule-based method and FSM-based method. The clock frequency of the proposed circuits is 500 *MHz*, [20] and [19] do not provide the information of the clock frequency.

Types	Functions		Proposed		PWL [20]		Horner's Rule [19]	FSM [19]
			8 segments	16 segments	8 segments	16 segments		
Decreasing Functions (Arch A)	cos(x)	Cell Area	379	423	489	525	769	1268
		Delay (ns)	1.48[a]	1.53[a]	3.07	3.11	2.86	2.96
		Power (μw)	203.94[a]	195.00[a]	13.66	13.68	21.56	36.44
	e^{-x}	Cell Area	357	383	491	530	763	2489
		Delay (ns)	1.53[a]	1.51[a]	2.73	2.97	2.82	3.09
		Power (μw)	186.44[a]	198.81[a]	13.06	13.9	21.48	41.29
	e^{-2x}	Cell Area	360	387	504	540	1199	1269
		Delay (ns)	1.50[a]	1.43[a]	3.08	3.32	2.87	2.71
		Power (μw)	187.69[a]	190.13[a]	13.35	14.13	21.78	35.57
Increasing Functions (Arch B)	sin(x)	Cell Area	387	450	601	650	2671	1269
		Delay (ns)	1.51[a]	1.51[a]	2.87	2.99	3.09	2.71
		Power (μw)	195.81[a]	206.38[a]	16.58	17.56	67.65	35.49
	$ln(1+x)$	Cell Area	394	449	607	650	763	1269
		Delay (ns)	1.51[a]	1.50[a]	2.92	3.01	2.89	2.78
		Power (μw)	207.19[a]	206.38[a]	16.73	17.50	21.42	34.79
	tanh(x)	Cell Area	381	438	603	657	674	1270
		Delay (ns)	1.51[a]	1.51[a]	2.97	3.12	2.89	2.71
		Power (μw)	189.31[a]	209.63[a]	16.62	17.64	18.82	35.52
	tanh(4x)	Cell Area	393	473	-	-	729	254
		Delay (ns)	1.50[a]	1.51[a]	-	-	4.07	2.54
		Power (μw)	196.98[a]	208.04[a]	-	-	-	-
	sigmoid(x)	Cell Area	353	361	600	640	758	1489
		Delay (ns)	1.51[a]	1.53[a]	2.86	3.07	2.79	3.09
		Power (μw)	186.77[a]	199.12[a]	16.54	17.29	21.15	41.23
	e^{x-1}	Cell Area	364	359	-	-	-	1144
		Delay (ns)	1.53[a]	1.53[a]	-	-	-	2.76
		Power (μw)	202.31[a]	201.50[a]	-	-	-	-
Non-monotonic Functions (Arch C)	sin(πx)/π	Cell Area	369	405	600	647	680	1269
		Delay (ns)	1.51[a]	1.53[a]	3.11	3.16	2.81	2.71
		Power (μw)	196.63[a]	189.31[a]	16.30	17.49	19.11	35.52

[a] This is a scaled result of from $40nm$ technology.

given in terms of equivalent numbers of 2-input NAND gates. The results of other works are given in [20] under 65 *nm* CMOS technology, so we scale the results of delay and power to 65 *nm* for fair comparisons.

To achieve the same order of precision, FSM-based method and Horner's Rule-based method require more SNGs for generating parameters. Using a 8-state FSM, the FSM-based method requires at least 10 SNGs, a MUX with 8 inputs, and more than 8 delay elements. For the functions in Table 6, Horner's Rule-based method requires 4 or 5 SNGs, and several delay elements for square operations. As only 3 SNGs are required, our circuits can achieve lower hardware complexity. With less gates in the critical path, the proposed circuits have shorter critical path than these two works. The FSM-based implementations contain longer computations inside feedback loops. Thus, compared with FSM-based method, 46% of critical path delays can be saved on average with our method. Horner's Rule-based method has a comparator and more than 5 level of NAND gates in the critical path, which is in proportion to the Maclaurin polynomial's order. Compared with Horner's Rule-based method, 49% of critical path delays can be saved on average. The detailed work frequency has not been given, so it is unfair to compare the power numbers according to the synthesis results. Since more SNGs are required, more dynamic power should be consumed in Horner's Rule-based and FSM-based methods than the proposed method. Plus, the critical path of our method is significantly shorter than these two works. As a result, lower power consumptions can possibly be achieved, if the proposed circuits work under a lower supplied voltage.

The hardware complexity of our work is at the same order of magnitude as PWL-based method in [20]. For some functions such as $sin(\pi x)/\pi$, by taking advantage of the functions' symmetry, less parameters are required to be generated. Thus, for this kinds of functions, our method is superior than the work in [20]. In terms of critical path, our method has clear advantages than PWL-based method. The critical path of our method can be further reduced by pipelining technique. The detailed work frequency has not been given in [20], so it is unfair to compare the power according to the synthesis results. With the same numbers of SNGs, the power of PWL-based method and our method should be at the

same order. Notably, the critical path of our method is shorter than that of PWL-based method, so lower power consumption can possibly be achieved, if the proposed circuits work under a lower supplied voltage.

In general, our method has significant advantages in critical path. SNGs occupy about 75% of the total area and about 85% of the total power in the proposed circuits. To decrease power consumptions and area, schemes such as LFSR-sharing can be further applied for optimizing SNGs. The optimization schemes are beyond the discussion range of our work and the details can be found in [12], [22], [23].

VI. CONCLUSION

Based on PWL technique and truncated Taylor expansion, a novel approximation method for arithmetic functions has been presented in this paper. The detailed approximation equations have been derived for monotonic functions and non-monotonic functions. For different types of functions, optimized hardware architectures have been developed. The proposed circuits can implement a broad range of arithmetic functions using unipolar stochastic logic. According to the experimental results, high-precision approximations for arithmetic functions can be implemented with low-complexity circuits. Compared with the previous PWL-based method, our method can be generalized for many other functions. Compared with the FSM-based and Horner's Rule-based methods, the proposed circuits has significant advantages in computation delay and hardware complexity.

REFERENCES

[1] M. Alawad and M. Lin, "Survey of stochastic-based computation paradigms," *IEEE Trans. Emerg. Topics Comput.*, vol. 7, no. 1, pp. 98–114, Jan. 2019.

[2] S. Abdallah, A. Chehab, I. H. Elhajj, and A. Kayssi, "Stochastic hardware architectures: A survey," in *Proc. Int. Conf. Energy Aware Comput.*, Dec. 2012, pp. 1–6.

[3] J. P. Hayes, "Introduction to stochastic computing and its challenges," in *Proc. 52nd Annu. Des. Autom. Conf. (DAC)*, Dec. 2015, pp. 1–3.

[4] V.-T. Nguyen, T.-K. Luong, H. Le Duc, and V.-P. Hoang, "An efficient hardware implementation of activation functions using stochastic computing for deep neural networks," in *Proc. IEEE 12th Int. Symp. Embedded Multicore/Many-Core Syst. Chip (MCSoC)*, Sep. 2018, pp. 233–236.

[5] J. Li, Z. Yuan, Z. Li, C. Ding, A. Ren, Q. Qiu, J. Draper, and Y. Wang, "Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 1230–1236.

[6] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI implementation of deep neural network using integral stochastic computing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2688–2699, Oct. 2017.

[7] T. Hirtzlin, B. Penkovsky, M. Bocquet, J.-O. Klein, J.-M. Portal, and D. Querlioz, "Stochastic computing for hardware implementation of binarized neural networks," *IEEE Access*, vol. 7, pp. 76394–76403, 2019.

[8] Z. Li, J. Li, A. Ren, R. Cai, C. Ding, X. Qian, J. Draper, B. Yuan, J. Tang, Q. Qiu, and Y. Wang, "HEIF: Highly efficient stochastic computing-based inference framework for deep neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 8, pp. 1543–1556, Aug. 2019.

[9] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Proc. 50th Annu. Des. Autom. Conf. (DAC)*, May 2013, pp. 1–6.

[10] I. Perez-Andrade, S. Zhong, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "Stochastic computing improves the timing-error tolerance and latency of turbo decoders: Design guidelines and tradeoffs," *IEEE Access*, vol. 4, pp. 1008–1038, 2016.

[11] B. D. Brown and H. C. Card, "Stochastic neural computation. I. Computational elements," *IEEE Trans. Comput.*, vol. 50, no. 9, pp. 891–905, Sep. 2001.

[12] H. Ichihara, S. Ishii, D. Sunamori, T. Iwagaki, and T. Inoue, "Compact and accurate stochastic circuits with shared random number sources," in *Proc. IEEE 32nd Int. Conf. Comput. Des. (ICCD)*, Oct. 2014, pp. 361–366.

[13] A. Alaghi and J. P. Hayes, "Fast and accurate computation using stochastic circuits," in *Proc. Des., Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2014, pp. 1–4.

[14] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 2, pp. 1–19, May 2013.

[15] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, Nov. 2012, pp. 480–487.

[16] P. Li, W. Qian, and D. J. Lilja, "A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic," in *Proc. IEEE 30th Int. Conf. Comput. Des. (ICCD)*, Sep. 2012, pp. 303–308.

[17] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Proc. 45th Annu. Conf. Des. Autom. (DAC)*, 2008, pp. 648–653.

[18] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 93–105, Jan. 2011.

[19] K. K. Parhi and Y. Liu, "Computing arithmetic functions using stochastic logic by series expansion," *IEEE Trans. Emerg. Topics Comput.*, vol. 7, no. 1, pp. 44–59, Jan. 2019.

[20] T.-K. Luong, V.-T. Nguyen, A.-T. Nguyen, and E. Popovici, "Efficient architectures and implementation of arithmetic functions approximation based stochastic computing," in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2019, pp. 281–287.

[21] K. K. Parhi, "Stochastic logic implementations of polynomials with all positive coefficients by expansion methods," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 11, pp. 1698–1702, Nov. 2018.

[22] F. Neugebauer, I. Polian, and J. P. Hayes, "Building a better random number generator for stochastic computing," in *Proc. Euromicro Conf. Digit. Syst. Des. (DSD)*, Aug. 2017, pp. 1–8.

[23] K. Kim, J. Lee, and K. Choi, "An energy-efficient random number generator for stochastic circuits," in *Proc. 21st Asia South Pacific Des. Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 256–261.

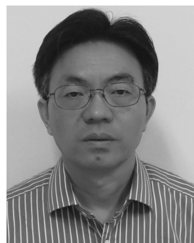
ZIDI QIN received the B.S. degree in telecommunication engineering from Chongqing University, Chongqing, China, in 2015. She is currently pursuing the Ph.D. degree with the School of Electronic Science and Engineering, Nanjing University, China. Her current research interests include network compression and hardware acceleration for deep learning algorithms, VLSI design, and reconfigurable computing.

YUOU QIU received the B.S. degree in electronic information science and technology from Nanjing University, Nanjing, China, in 2018, where she is currently pursuing the master's degree with the School of Electronic Science and Engineering. Her current research interests include digital integrated circuit design, reconfigurable computing, and VLSI implementation of neural networks.

MUHAN ZHENG received the B.S. degree in electronic information science and technology from Nanjing University, Nanjing, China, in 2019, where she is currently pursuing the master's degree with the School of Electronic Science and Engineering. Her current research interests include digital integrated circuit design and VLSI implementations of neural networks.

HONGXI DONG received the B.S. degree in electronic information engineering from the Nanjing University of Aeronautics Astronautics, Nanjing, China, in 2019. She is currently pursuing the master's degree with the School of Electronic Science and Engineering, Nanjing University, Nanjing. Her current research interests include digital integrated circuit design, reconfigurable computing, and VLSI implementation of machine learning algorithms.

ZHONGHAI LU (Senior Member, IEEE) received the B.S. degree in radio and electronics from Beijing Normal University, Beijing, China, in 1989, and the M.S. degree in system-on-chip design and Ph.D. degree in electronic and computer system design from the KTH Royal Institute of Technology, Stockholm, Sweden, in 2002 and 2007, respectively. He was an Engineer in electronic and embedded systems, from 1989 to 2000. He is currently a Professor with the School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology. He has authored more than 180 peer-reviewed articles. His current research interests include interconnection networks, computer architecture, design automation, and real-time systems.



ZHONGFENG WANG (Fellow, IEEE) received the B.S. and M.S. degrees from Tsinghua University, and the Ph.D. degree from the University of Minnesota, Minneapolis, in 2000. He worked with Oregon State University and National Semiconductor Corporation. He was with Broadcom Corporation, CA, USA, from 2007 to 2016, as a Leading VLSI Architect. He has been working with Nanjing University, China, as a Distinguished Professor, since 2016. He is a World-Recognized Expert on Low-Power High-Speed VLSI Design for Signal Processing Systems. He has published over 200 technical articles with multiple best paper awards received from the IEEE technical societies, among which is the *VLSI Transactions* Best Paper Award of 2007. He has edited one book VLSI and held more than 20 U.S. and China patents. His current research interests include optimized VLSI design for digital communications and deep learning. In the current record, he had many articles ranking among top 25 most (annually) downloaded manuscripts in the IEEE Transactions on VLSI Systems. In the past, he has served as an Associate Editor for the IEEE Transactions on Circuits and Systems I, T-CAS-II, and T-VLSI Systems for many terms. He has also served as a TPC Member and various chairs for tens of international conferences. Moreover, he has contributed significantly to the industrial standards. So far, his technical proposals have been adopted by more than 15 international networking standards. In 2015, he was elevated to the Fellow of IEEE for contributions to VLSI design and implementation of FEC coding.

HONGBING PAN received the B.S. degree in applied physics and Ph.D. degree in microelectronics and solid state electronics from Nanjing University, Nanjing, China, in 1994 and 2005, respectively.

From 2006 to 2012, he was an Associate Professor with the Institute of VLSI Design, Nanjing University. Since 2013, he has been a Professor with the School of Electronic Science and Engineering, Nanjing University. He is the author of more than 40 articles. His research interests include VLSI design, CMOS sensors, reconfigurable computing, and artificial intelligence.