CORDIC-Based Architecture for Computing Nth Root and Its Implementation

Yuanyong Luo, Yuxuan Wang, Huaqing Sun, Yi Zha, Zhongfeng Wang, *Fellow, IEEE*, and Hongbing Pan

***Abstract*— This paper presents a COordinate Rotation Digital Computer (CORDIC)-based architecture for the computation of Nth root and proves its feasibility by hardware implementation. The proposed architecture performs the task of Nth root simply by shift-add operations and enables easy tradeoff between the speed (or precision) and the area. Technically, we divide the Nth root computation into three different subtasks, and map them onto three different classes of the CORDIC accordingly. To overcome the drawback of narrow convergence range of the CORDIC algorithm, we adopt several innovative methods to yield a much improved convergence range. Subsequently, in terms of convergence range and precision, a flexible architecture is developed. The architecture is validated using MATLAB with extensive vector matching. Finally, using a pipelined structure with fixed-point input data, we implement the example circuits of the proposed architecture with radicand ranging from zero to one million, and achieve an average mean of approximately 10^{-7} for the relative error. The design is modeled using Verilog HDL and synthesized under the TSMC 40-nm CMOS technology. The report shows a maximum frequency of 2.083 GHz with 197421.00 μm^2 area. The area decreases to 169689.98 μm^2 when the frequency lowers to 1.00 GHz.**

***Index Terms*— Nth root, CORDIC, convergence range, fixed-point, pipelined structure, high speed.**

I. Introduction

IN VLSI domain, the design of the computation of the Nth root is a challenging task. The Nth root computation includes some frequently used operations, such as square root and cube root. Therefore, loads of algorithms and implementations for square root and cube root have been proposed. The most famous method to calculate these two roots might have been Newton-Raphson(NR) method. Literatures [1]–[5] perform the square root extraction by NR method. Meanwhile, they also perform the division with square root together because these two tasks can share some common circuits. A floating point cube root has been implemented on FPGA using NR method in paper [6]. Its highest frequency is up to 149 MHz on Virtex5. The merit of NR method is that its quadratic convergence leads to less iterations. The drawback of NR method is that it requires an initial guess. Random guesses result in different precision of the outputs. Meanwhile, it requires lots of multiplication operations. In addition to NR method, some other linear convergence and digit-by-digit algorithms for square and cube roots have also been proposed. Shift-add-multiply based square root finding algorithms have been presented in [7] and [8]. Other two multiplications-required cube root algorithms were implemented on FPGA in [9] and [10]. Those multiplication-based algorithms cannot achieve high processing speed. Besides, a CORDIC based algorithm for square root extraction deserves mentioning [11]–[14]. By applying the simple shift-add operations, this method can obtain high processing speed. Due to its distinct advantages, it's widely deployed in many digital signal processing systems. Overall, square root and cube root attract most of the attentions. However, other higher order roots are also needed in some special areas such as volume shading for computer graphics, atmospheric models, radiance and luminance and so forth [15]. Hence, this paper aims at a general architecture and fixed circuits to compute arbitrary Nth root. This work is very complex because square and cube roots finding algorithms are hard to merge into an integrated one. Theoretically, NR method can compute any root. However, the complexity of the circuits rapidly increases along with the augment of the N and a fixed implementation cannot be used to calculate other roots.

Some researchers have been putting their efforts into the computation of general Nth root. A general digit-recurrence algorithm for the calculation of the Nth root is presented in [16]. Like NR method, the implementation of this algorithm depends on N, i.e., the larger N the larger the complexity. A fixed implementation is targeted at a given N so that this method cannot be considered as a general method for the computation of any Nth root. In paper [17], based on NR method, a high-level synthesis and verification tool for specific Nth root processing engine is proposed. The design can be used for modeling different architectures based on an assigned N value, such as square root for N=2, cube root for N=3 and so on. Again, it is not a general approach for the calculation of Nth root. In paper [18], according to the equation $R^{\frac{1}{N}} = 2^{(\frac{1}{N})log_2^R}$, a top-level architecture comprised a reciprocal-logarithm-multiplication-exponential chain to calculate Nth root is presented. Neither detailed architectures of sub-modules nor experiment results were provided in paper [18]. We are not sure about the exact performance of the proposed design. Thus, we try to derive a fixed architecture to calculate arbitrary Nth root using the idea of task dividing to overcome all the above-mentioned shortcomings in the existing works.

Manuscript received January 21, 2018; revised April 11, 2018; accepted May 3, 2018. This paper was recommended by Associate Editor M. M. Kermani. *(Corresponding author: Hongbing Pan.)*

The authors are with Nanjing University, Nanjing 210008, China (e-mail: phb@nju.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2018.2835822

In this paper, a CORDIC-based architecture for the calculation of the Nth root is proposed and implemented. CORDIC was first invented by Volder [19], [20] in 1959 for the evaluation of trigonometric functions, multiplication and division. Then, John Walter extended its computation capacities by changing a few parameters to calculate logarithms, exponentials and square roots [21], [22]. Up till now, a number of attempts have been made to design different algorithms using CORDIC in pursuit of high performance and low hardware cost. It is summarized in [14] that CORDIC has been used for applications which include complex multiplication, eigenvalue computation, matrix inversion and so on. In addition, CORDIC is also used for complex division and square root in [23] and [24]. Some architectures of mathematical transformation including discrete Fourier transform [25] and discrete Mellin transform [26] are also modeled using CORDIC. It has even been used for simulating the Fire Neuron [27]. Overall, CORDIC is a powerful iterative algorithm which can be assembled to complete lots of tasks. Inspired by the idea of task dividing [18] and the functionalities of CORDIC [14], we separate the Nth root computation into three different subtasks according to the equation $R^{\frac{1}{N}} = exp(\frac{ln(R)}{N})$. The first step is to calculate $ln(R)$ using the Hyperbolic CORDIC of Vectoring mode(HV). The second step is a division operation for the computation of $\frac{ln(R)}{N}$ via the Linear CORDIC of Vectoring mode(LV). The final step is to complete the exponential operation of $exp(\cdot)$ through the Hyperbolic CORDIC of Rotation mode(HR). Unfortunately, it is shown that with the standard CORDIC, the proposed architecture only has the flexibility in precision. The convergence range of radicand R is too narrow to satisfy various requirements. Thus, we make efforts to expand the convergence range of the CORDIC. Several effective methods are presented to realize this requirement in [28]. By using these innovative methods, we make the proposed architecture have the flexibility in both precision and convergence range. We verify the architecture on the platform of MATLAB. Software test shows that the proposed architecture is correct and efficient, and it demonstrates the flexibility of precision and convergence range as well. Finally, we implement the example circuits in hardware to further prove the feasibility of our architecture. The circuits are modeled by Verilog HDL and synthesized under the TSMC 40-nm CMOS technology. We verify the outputs of the circuits by comparing with the MATLAB's simulated results. And it shows that the order of magnitude of error keeps consistent with that of software test. The highest frequency is up to 2.083 GHz with area 197421.00 $\boldsymbol{\mu m^2}$. The area decreases to 169689.98 $\boldsymbol{\mu m^2}$ when the frequency is only 1.00 GHz.

The rest of this paper is organized as follows. Section II presents the proposed architecture with standard CORDIC and analyzes its weakness of narrow range of radicand R. Section III introduces the methods to expand the convergence range and consummates the proposed architecture by enabling the flexibility in convergence range. The software test is performed using MATLAB with extensive vector matching in this part. Section IV implements the example circuits in hardware and compares it with a Vedic multiplier [29]–[31]. The timing, transistor count and error analyses are also carried out in this part. Section V discusses the prior attempts and proposes a second version of the proposed architecture. Section VI concludes our work with a thorough discussion of the merit and weakness of the proposed architecture as well as its implementation.

II. Architecture Consisting of the Standard CORDIC

In this section, we will briefly introduce the standard CORDIC and elaborate on the architecture for computing Nth root. The convergence range will be analyzed to show the shortcomings of this primitive architecture.

A. Introduction to the Standard CORDIC

Since we only use the Hyperbolic CODIC and the Linear CORDIC in this paper, the introduction to the Circular CORDIC is omitted. In the following, the iterative formulas of HV, HR and LV classes of CORDIC will be presented.

The iterative formulas of HV:

$$x^{i+1} = x^i - sign(y^i)(2^{-i}y^i), \tag{1a}$$

$$y^{i+1} = y^i - sign(y^i)(2^{-i}x^i), \tag{1b}$$

$$z^{i+1} = z^i + sign(y^i)tanh^{-1}(2^{-i}), \tag{1c}$$

where i starts with 1 and is an integer.

The iterative formulas of HR:

$$x^{i+1} = x^i + sign(z^i)(2^{-i}y^i), \tag{2a}$$

$$y^{i+1} = y^i + sign(z^i)(2^{-i}x^i), \tag{2b}$$

$$z^{i+1} = z^i - sign(z^i)tanh^{-1}(2^{-i}), \tag{2c}$$

where i starts with 1 and is an integer.

The iterative formulas of LV:

$$x^{i+1} = x^i, \tag{3a}$$

$$y^{i+1} = y^i - sign(y^i)(2^{-i}x^i), \tag{3b}$$

$$z^{i+1} = z^i + sign(y^i)(2^{-i}), \tag{3c}$$

where i starts with 0 and is an integer.

For the Hyperbolic CORDIC, when the iterative sequence number i equals (3n+1), i.e., when $i = 4, 7, 10, 13\ldots$, one more iteration is needed, otherwise, the CORDIC will not converge [28]. After several iterations, the outputs of the above-mentioned CORDIC will converge to some special functions shown as Table I [14]. The scale-factor K_h of Hyperbolic CORDIC is given by

$$K_h = \Pi_{i=1}^{n}\left(\sqrt{1-2^{-2i}}\right). \tag{4}$$

Actually, the scale-factor brings no trouble to our architecture of the Nth root computation because we can avoid it by the special inputs, which will be seen in the next section. Table I indicates that the CORDIC is capable of calculating inverse hyperbolic tangent, hyperbolic sine, hyperbolic cosine, and division. These lay the foundation for the realization of the Nth root evaluation.

TABLE I
OUTPUTS OF THE CORDIC

CORDIC	OUTPUTS
HV	$x_n = K_h\sqrt{x_0^2 - y_0^2}$
	$y_n = 0$
	$z_n = z_0 + \tanh^{-1}(\frac{y_0}{x_0})$
HR	$x_n = K_h(x_0 \cosh z_0 - y_0 \sinh z_0)$
	$y_n = K_h(y_0 \cosh z_0 + x_0 \sinh z_0)$
	$z_n = 0$
LV	$x_n = x_0$
	$y_n = 0$
	$z_n = z_0 + y_0/x_0$

B. Top-Level Architecture for Computing Nth Root

Consider two positive real numbers R and N. The essence of our architecture is the fact that the following equation always holds true.

$$R^{\frac{1}{N}} = exp(\frac{ln\,(R)}{N}). \tag{5}$$

The proposed architecture has been divided into three steps. The first step is the calculation of the logarithm $ln\,(R)$ using the CORDIC of HV class. In order to complete this task, the inputs of the HV should be initialized as follows,

$$x_0 = R + 1; \quad y_0 = R - 1; \; z_0 = 0. \tag{6}$$

Thus, the output of z_n can be formulated as

$$z_n = tanh^{-1}\left(\frac{R-1}{R+1}\right) = \frac{1}{2}ln(R). \tag{7}$$

Then, the z_n is left shifted by 1 bit to get the actual value of $ln(R)$.

The next step in the proposed architecture is the division operation via the CORDIC of LV class. The inputs to LV are making $x_0 = N$, $y_0 = ln(R)$ and $z_0 = 0$ so that the output z_n can converge to $ln(R)/N$.

The final step is the calculation of exponential $exp(\cdot)$. This can be done by considering $x_0 = 1/K_h$, $y_0 = 0$ and $z_0 = ln(R)/N$, and operating the CORDIC of HR class. The outputs of this step can be expressed as

$$x_n = cosh\left(\frac{ln(R)}{N}\right), \quad y_n = sinh(\frac{ln(R)}{N}).$$

So the arbitrary real root can be computed by an identical equation

$$exp(x) = cosh(x) + sinh(x).$$

An add operation is needed subsequently to complete the whole task which can be expressed as

$$R^{\frac{1}{N}} = \exp\left(\frac{\ln(R)}{N}\right) = \cosh\left(\frac{\ln(R)}{N}\right) + \sinh\left(\frac{\ln(R)}{N}\right).$$

Fig. 1 illustrates the entire computing flow.

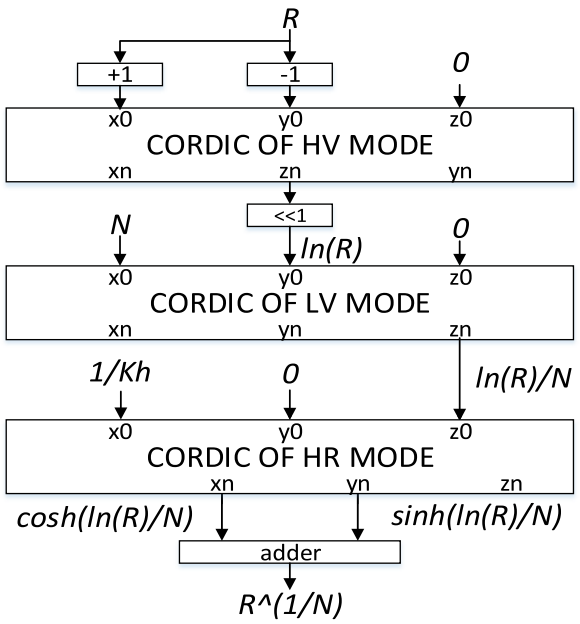


Fig. 1. The computing flow of the Nth root.

TABLE II
CONVERGENCE RANGE OF THE STANDARD CODRIC

CORDIC OF HV	CORDIC OF LV	CORDIC OF HR
$x_0 > 0$ $\lvert\tanh^{-1}\left(\frac{y_0}{x_0}\right)\rvert \leq$ 1.1182	$\lvert\frac{y_0}{x_0}\rvert \leq 2$	$\lvert z_0\rvert \leq 1.1182$

C. Convergence Range Analysis for Primitive Architecture

After the construction of the primitive Nth root architecture, we need to analyze the range of the inputs R and N. This can be done by extensive analyses for each CORDIC comprised in our top-level architecture shown as Fig. 1. For the standard CORDIC, the convergence range is summarized in Table II [28]. Now, we apply those constraints to the proposed architecture based on the standard CORDIC.

First, for the CORDIC of HV class, because

$$\lvert\tanh^{-1}\left(\frac{y_0}{x_0}\right)\rvert \leq 1.1182,$$

we can derive that

$$\left|\frac{y_0}{x_0}\right| \leq \tanh(1.1182) = 0.807.$$

Considering (6), (7) and above expression, the following inequality can be obtained.

$$\lvert\frac{R-1}{R+1}\rvert \leq 0.807. \tag{8}$$

From Table II, the input x_0 of HV should be positive. According to (6), we can get that

$$R + 1 > 0. \tag{9}$$

Combining (8) and (9), the range of R can be figured out as

$$R \in [\frac{1}{9.36}, 9.36]. \tag{10}$$

Next, consider the CORDIC of HR class. Based on Table II we know that

$$|z_0| \leq 1.1182. \tag{11}$$

Because the input $z_0 = ln(R)/N$, (10) and (11), we can obtain the range of N as follow,

$$N \geq \frac{ln(9.36)}{1.1182} = 2. \tag{12}$$

As for the CORDIC of LV class, since

$$\max\left(\left|\frac{y_0}{x_0}\right|\right) = \frac{\ln(9.36)}{2} = 1.1182 \leq 2,$$

the constraint in Table II for LV has been automatically met.

In summary, the convergence range of this primitive architecture can be specified as in (10) and (12). The range of N has the lower bound 2, which satisfies the definition of Nth root. However, the range of R is too narrow to meet the requirement, so it is necessary to expand it.

III. The Architecture With Improved CORDIC

In this section, we aim at expanding the convergence range of the proposed primitive architecture. Because the proposed architecture is developed based on the CORDIC, we take several measures to extend the convergence range of the Hyperbolic and the Linear CORDIC. An improved CORDIC-based architecture for computing Nth root, which has the flexibility in both precision and convergence range, will be presented and tested in the following.

A. Ways for Expanding the Convergence Range of CORDIC

The Linear CORDIC is the simplest algorithm of the CORDIC classes. As described in the paper [28], we could choose to expand the set of iteration indexes from $i = 0, 1, 2, \cdots, n$ to $i = -m, -m+1, \cdots, n$ and maintain the same basic iterative formulas of (3). Then instead of the constraint for LV described in Table II, the new convergence range is given by

$$|\frac{y_0}{x_0}| \leq 2^{m+1}. \tag{13}$$

For the Hyperbolic CORDIC described by (1) and (2), we can also expand the iteration indexes to include non-positive numbers as $i = -m, -m+1, \cdots, n$. However, the iterative formulas of (1) and (2) can't be maintained the same for $i \leq 0$. As mentioned in [28], all the terms 2^{-i} in (1) and (2) need to be replaced by $(1-2^{-2^{-i+1}})$ when $i \leq 0$. As a result, for the non-positive iteration indexes, the iterative formulas of Hyperbolic CORDIC are given as follows.

The iterative formulas of HV for $i \leq 0$:

$$x^{i+1} = x^i - sign(y^i)(1-2^{-2^{-i+1}})y^i, \tag{14a}$$
$$y^{i+1} = y^i - sign(y^i)(1-2^{-2^{-i+1}})x^i, \tag{14b}$$
$$z^{i+1} = z^i + sign(y^i)\ tanh^{-1}(1-2^{-2^{-i+1}}). \tag{14c}$$

TABLE III
The Features of the Improved Hyperbolic CORDIC

m	θ_{max}	$1/K_h$
0	2.099933524278612	1.825620556590062
1	3.816927126521185	5.246258103242353
2	6.935111921623039	59.412683447074620
3	12.826859141670756	$1.075488553331324\times 10^4$

The iterative formulas of HR for $i \leq 0$:

$$x^{i+1} = x^i + sign(z^i)(1-2^{-2^{-i+1}})y^i, \tag{15a}$$
$$y^{i+1} = y^i + sign(z^i)(1-2^{-2^{-i+1}})x^i, \tag{15b}$$
$$z^{i+1} = z^i - sign(z^i)\ tanh^{-1}(1-2^{-2^{-i+1}}). \tag{15c}$$

The convergence range of the above improved Hyperbolic CORDIC depends on the max value of the summation of the rotation angles, which can be defined as

$$\theta_{max} = \sum_{i=-m}^{0} tanh^{-1}(1-2^{-2^{-i+1}}) + \sum_{i=1}^{n} tanh^{-1}(2^{-i}). \tag{16}$$

In (16), the repeated iterations ($i = 4, 7, 10, 13\ldots$) are accumulated twice. For instance, when $i = 4$,

$$\theta_{max} = \theta_{max} + tanh^{-1}(2^{-4}) + tanh^{-1}(2^{-4}).$$

Then, the constraint of the HV shown in Table II can be updated by $|\tanh^{-1}(\frac{y_0}{x_0})| \leq \theta_{max}$ and that of the HR can be updated by $|z_0| \leq \theta_{max}$.

In the proposed architecture, $1/K_h$ is a constant input to the CORDIC of HR class, so it is necessary to compute this number in advance. After including the additional non-positive indexed iterations, the scale-factor K_h can be redefined as

$$K_h = \Pi_{i=-m}^{0}\sqrt{1-\left(1-2^{-2^{-i+1}}\right)^2}\,\Pi_{i=1}^{n}\sqrt{1-\left(2^{-i}\right)^2}. \tag{17}$$

In (17), the repeated iterations need to multiply twice.

Consider the iteration indexes as $i = -m, -m+1, \cdots, 20$. Now, based on (16) and (17), the features of the improved Hyperbolic CORDIC can be concluded in Table III. Based on Table III, we can see that the convergence range of the Hyperbolic CORDIC can be dramatically expanded by adding limited number of additional iterations.

B. Improved CORDIC-Based Architecture

In Section III.A, we have elaborated on how to expand the convergence range of the CORDIC. Now, we will detail an example to show how the improved CORDIC applies to our architecture. After that, we will present the CORDIC-based architecture for computing Nth root systematically.

Let's go through our example. Consider the max positive iteration index as 20 for all the CORDIC used in our architecture.

First, choose m=2 for the Hyperbolic CORDIC. Based on the constraint of HV and the θ_{max} provided in TABLE III, we have

$$\left|\tanh^{-1}(\frac{y_0}{x_0})\right| \leq \theta_{max} = 6.935111921623039.$$

Then, we have the following inequality.

$$\left|\frac{y_0}{x_0}\right| \leq tanh(6.935111921623039) = 0.999998106488676. \quad (18)$$

According to (6) and (18), we can obtain that

$$R \in [\frac{1}{1.056\times 10^6}, 1.056\times 10^6] = [9.467\times 10^{-7}, 1.056\times 10^6]. \quad (19)$$

Next, we pay attention to the CORDIC of HR class used in the proposed architecture. Since $m = 2$, from Table II and III we have

$$|z_0| \leq \theta_{max} = 6.935111921623039. \quad (20)$$

Considering the input $z_0 = ln(R)/N$, (19) and (20), we can calculate the range of N as

$$N \geq \frac{ln\left(1.056\times 10^6\right)}{6.935111921623039} = 2. \quad (21)$$

Finally, the CORDIC of LV class needs to be addressed. The inputs of LV are that $x_0 = N$, $y_0 = ln(R)$ and $z_0 = 0$. Combining with (13), (19) and (21), we can derive that

$$\max\left|\frac{\ln(R)}{N}\right| = \frac{\ln(1.056\times 10^6)}{2} \leq 2^{m+1},$$

i.e., $m \geq 2$. So we can choose *m=2* for the CORDIC of LV class. Up till now, an example of our proposed architecture for computing Nth root has been clarified.

Now, it is time to show the proposed CORDIC-based architecture for computing Nth root in a general way. Consider the iteration index *i* for various CORDICs used in the architecture has the same range, $i = -m, -m+1, \cdots, n$. Then we denote the proposed architecture as $Arch.(m, n)$, in which m represents the maximum non-positive iteration sequence number while n means the maximum positive iteration sequence number. When the iteration index is non-positive, Hyperbolic CORDIC in $Arch.(m, n)$ adopts the iterative formulas described by (14) and (15), otherwise it adopts the formulas described by (1) and (2). As for the CORDIC of LV mode in $Arch.(m, n)$, it always adopts the iterative formulas (3).

Therefore, above derivation has clarified a special case $Arch.(2, 20)$. When choosing other value of m, the features of $Arch.(m, 20)$ can also be calculated in the same way as the above process. Table IV summarizes the detail features of $Arch.(2, 20)$ and $Arch.(3, 20)$. We can see that the lager value of m, the lager convergence range of R is. In other words, m in $Arch.(m, n)$ indicates the flexibility in convergence range of R in our proposed architecture. Actually, n in $Arch.(m, n)$ indicates the flexibility in precision, which will be shown in Section III.C. According to the extensive calculations, the lower bound of N in our proposed architecture always keeps as 2, which satisfies the definition of Nth root. The top-level architecture keeps the same as Fig. 1. The structure of each CORDIC is different from conventional CORDIC since we apply the improved CORDIC [28] to our design. The pipelined structure of our proposed architecture will be detailed in Section IV.

TABLE IV
FEATURES OF THE EXAMPLE ARCHITECTURES

Features		***Arch.*** $(2, 20)$	***Arch.*** $(3, 20)$
Iteration index	HV	$i = -2, -1, \cdots, 20$	$i = -3, -2, \cdots, 20$
	LV	$i = -2, -1, \cdots, 20$	$i = -3, -2, \cdots, 20$
	HR	$i = -2, -1, \cdots, 20$	$i = -3, -2, \cdots, 20$
Iterative formula	HV	(14) for $i \leq 0$, (1) for $i > 0$	
	LV	(3)	
	HR	(15) for $i \leq 0$, (2) for $i > 0$	
R		$[9.467\times 10^{-7}, 1.056\times 10^6]$	$[7.22\times 10^{-12}, 1.38\times 10^{11}]$
N		≥ 2	≥ 2
$1/K_h$		59.412683447074620	1.0754885533313$\times 10^4$

a. For the Hyperbolic CORDIC, one more iteration is needed when i = 4,7,10,13 ....
b. R and N are real numbers.

C. Software Test for the Proposed Architecture

Before implementing our example architecture in hardware, it is essential to create the benchmark via software simulation. We code the $Arch.(m, n)$ on MATLAB and simulate their relative errors. First, we test the features of $Arch.(2, 20)$ and $Arch.(3, 20)$ to justify the correctness of our proposed architecture. Then, we test $Arch.(0, n)$, $Arch.(2, n)$ and $Arch.(3, n)$ with different n, to show the flexibility in precision of our proposed architecture.

The relative error is defined as

$$Err_r = |\frac{T - C}{T}|. \quad (22)$$

In (22), T means the true value of the Nth root and C represents the result derived from the proposed architecture.

Suppose that the number of the Nth root we will test is denoted as *Num*. Then another important criterion for measuring the proposed architecture is the average value of the relative error, which is given by

$$Avg\_Err_r = \frac{\sum_{j=1}^{Num} Err_r}{Num}. \quad (23)$$

The final significant criterion defined in this paper is the max relative error which is described by

$$Max\_Err_r = \max\{Err\}_r. \quad (24)$$

After the introduction of the criteria for measuring the proposed architecture, we set the Nth root to be 1 million for each$Arch.(0, n)$ and $Arch.(2, n)$ test, and 5 million for each $Arch.(3, n)$ test. The inputs R and N are generated by random number. Actually, there is no upper bound for input N. For the convenience of software test, we set that upper bound to be 1002.

Table V presents the test results for the $Arch.(2, 20)$ and $Arch.(3, 20)$. The probability distribution of the relative error is also included in this table. The relative errors are mostly located near the Avg_Err_r and the max value of relative error all approaches 3×10^{-6}. Thus, the software test results have proven the correctness and the accuracy of the proposed architecture for the Nth root computation.

Table VI shows the tests of average value of relative error for $Arch.(0, n)$, $Arch.(2, n)$ and $Arch.(3, n)$, where n is

TABLE V
SOFTWARE TEST FOR $Arch.(2, 20)$ AND $Arch.(3, 20)$

Criterion	$\boldsymbol{Arch.(2,20)}$	$\boldsymbol{Arch.(3,20)}$
Num	1000,000	5000,000
Input range of R	$[10^{-6}, 10^{6}]$	$[10^{-10}, 10^{11}]$
Input range of N	$[2, 1002]$	$[2, 1002]$
Avg_Err_r	6.9×10^{-7}	6.9×10^{-7}
Max_Err_r	2.8×10^{-6}	3.1×10^{-6}
$P(Err_r \geq 10^{-6})$	26.52%	25.87%
$P(10^{-7} < Err_r < 10^{-6})$	64.27%	65.01%
$P(Err_r \leq 10^{-7})$	9.21%	9.12%

a. Input R and N are real numbers.
b. P(·) denotes the probability.

TABLE VI
SOFTWARE TEST FOR $Arch.(mn)$

Arc. / n	$\boldsymbol{Arch.(0,n)}$		$\boldsymbol{Arch.(2,n)}$		$\boldsymbol{Arch.(3,n)}$	
	Avg_Err_r	*latency*	Avg_Err_r	*latency*	Avg_Err_r	*latency*
3	6.865×10^{-2}	16	7.213×10^{-2}	26	7.397×10^{-2}	31
6	1.107×10^{-2}	27	1.120×10^{-2}	37	1.073×10^{-2}	42
9	1.324×10^{-3}	38	1.371×10^{-3}	48	1.413×10^{-3}	53
12	1.685×10^{-4}	49	1.692×10^{-4}	59	1.679×10^{-4}	64
15	2.196×10^{-5}	60	2.281×10^{-5}	70	2.173×10^{-5}	75
18	2.737×10^{-6}	71	2.725×10^{-6}	81	2.763×10^{-6}	86
21	3.504×10^{-7}	82	3.577×10^{-7}	92	3.451×10^{-7}	97

a. Input range of N for all $Arch.(m, n)$: [2,1002].
b. Input range of R for $Arch.(0, n)$: [0.015,66.77].
c. Input range of R for $Arch.(2, n)$: $[10^{-6}, 10^{6}]$.
d. Input range of R for $Arch.(3, n)$: $[10^{-10}, 10^{11}]$.
e. Latency=$5m + 3n + 2\left\lfloor\frac{n-1}{3}\right\rfloor + 7$ clock cycles, refer to Section IV.B.

set as different numbers. By massive calculations like the process provided in Section III.B, we find that different n in $Arch.(m, n)$ results in nearly the same convergence range of R, as long as m is fixed. For instance, the convergence range of R in $Arch.(0, n)$ always keeps the same as [0.015, 66.77], and that in $Arch.(1, n)$ always keeps as $[4.838\times10^{-4}, 2.067 \times 10^{3}]$. Therefore, in Table VI, input range of R is the same when m is fixed. From Table VI, it is evident that different n leads to different precision (Avg_Err_r) and different m leads to different convergence range of R. It makes sense that we conclude the proposed CORDIC-based architecture for computing Nth root, has the flexibility in both precision and convergence range. When each sub-module of our proposed architecture is implemented in a pipelined manner (refer to Section IV.A), the corresponding latency (refer to Section IV.B) for $Arch.(m, n)$ is also presented in Table VI. By observation, it is clear that the lower m or n is, the lower latency is. In hardware implementation, Table IV explains the trade-off between precision (or convergence range of R) and latency (or area, power, frequency).

IV. HARDWARE IMPLEMENTATION

In this section, to prove the feasibility of the proposed architecture, the special case $Arch.(2, 20)$ is coded in Verilog HDL for the fixed-point implementation. In order to achieve high sampling rate, a pipelined structure is adopted. Instead of the conventional pipelined structure for the standard CORDIC, we propose an innovative one for the additional iterations of the improved Hyperbolic CORDIC without lengthening the critical path. We also leverage the nature of the very different range of the input data of each CORDIC to design different input word lengths for each cascaded CORDIC, which can massively reduce the area consumption. In the following, details of the hardware design, timing analysis, results of the implementation, transistor count analysis, and the error analysis are presented.

A. Details of the Hardware Design

The top-level architecture stays the same as illustrated in Fig. 1. For the improved CORDIC of LV class, the iterative formulas maintain the original form as (3). Therefore, the pipelined structure for this case is consistent with the conventional methodology. However, the improved Hyperbolic CORDIC has different iterative formulas when the iteration index is non-positive as given by (14) and (15). So it is necessary to design a new iteration structure for them. Fig. 2 depicts the pipelined structure for the CORDIC of HR class used in $Arch.(2, 20)$. All the elements of the CORDIC of HR class are cascaded with each other to form a pipelined structure. When the iteration index $i > 0$, the conventional iteration structure is adopted as shown in the lower right of Fig. 2. When the iteration index $i \leq 0$, the proposed structure for (15) is shown in the lower left of Fig. 2. It is well known that the critical path of the conventional CORDIC is a shift and an add operations, while (15a) or (15b) needs one shift and two add operations, which lengthens the critical path by an additional add. Table IV tells that the number of non-positive indexed iteration is much less than that of positive indexed iteration, so it is not worth sacrificing speed due to a small number of non-positive indexed iteration. As a result, we add one-stage pipeline for the implementation of (15) to maintain the critical path as one shift and one add operations as it is shown in the lower left of Fig. 2. Observing (14) and (15), it is intuitive to find that the only difference between them is the judge condition, or the sign function, resulting in the nearly same implementation in hardware. Replacing $sign(z^i)$ with $sign(y^i)$ in Fig. 2, we can get the pipelined structure for the CORDIC of HV class. Up till now, the pipelined structure of the proposed CORDIC-based architecture for computing Nth root has been clarified.

Next, we analyze the required input word length for each improved CORDIC used in $Arch.(2, 20)$. In order to meet the average precision (10^{-7}) and the lower bound of input radicand R(shown in Table IV: 10^{-7}), we set the fractional part of every input dada to be 27 bits. For the first CORDIC of HV class illustrated in Fig. 1, the input data is R. Observing from Table IV we know that the upper bound of R is 1.056×10^{6}, so we set the integral part of R to be 20 bits which limits the max value of Rlower than $2^{20}+1 = 1,048,577$. Then consider the input data of second CORDIC of LV class, N and $ln(R)$. The max value of $ln(R)$ approximates $ln\,(1048577) = 13.863$,

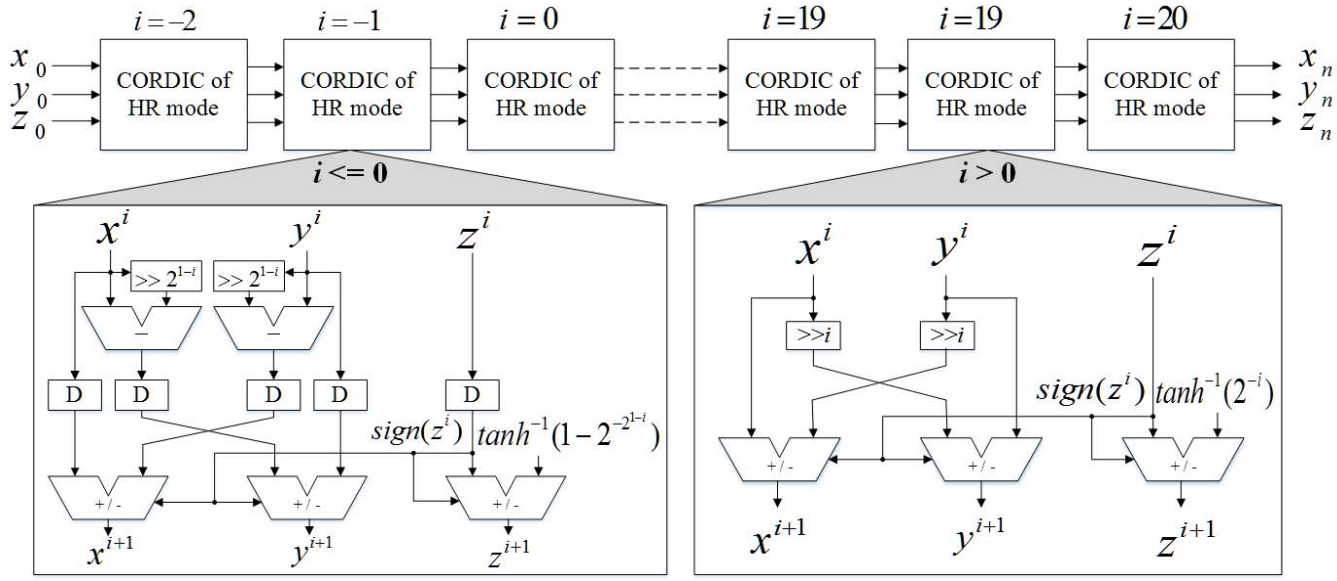


Fig. 2. Pipelined architecture for the improved CORDIC of HR class. When i = 4,7,10,13…, the iteration should be cascaded twice. For instance, when i=19, the iteration is cascaded twice as shown in this picture.

TABLE VII
WORD LENGTH SETTING

CORDIC	Item	Sign bit	Integral bit	Fractional bit	Total bit
HV	R	1	20	27	48
LV	$N, ln(R)$	1	10	27	38
HR	$1/K_h, ln(R)/N$	1	11	27	39
adder	$\cosh(\cdot), \sinh(\cdot), R^{1/N}$	1	11	27	39

a small number. Thus, the integral part of the input data for the second CORDIC depends on N. In practice, we do not require N to be very large. Therefore, we set the integral part of N and $ln(R)$ to be 10 bits, which limits the max value of N lower than $(2^{10}+1=1025)$. For the last CORDIC of HR class shown in Fig. 1, the max value of the outputs $\sinh(\cdot)$ and $\cosh(\cdot)$ all approach

$$\frac{\exp\left(\max\left(\frac{\ln(R)}{N}\right)\right)}{2}=\frac{\exp\left(\frac{\ln(1048577)}{2}\right)}{2}=512.00024.$$

Based on the above analysis, it seems to make sense to set the integral part of the input data to be 10 bits. However, due to the add operations in (15a) and (15b), the outputs may exceed 10 bits. Thus we actually set the integral part of the input data to be 11 bits for the last CORDIC. Except the fractional part and the integral part, a sign bit is added in front of every input data. The word length setting for every CORDIC and the final adder is outlined in Table VII. By the means of setting different word length for each CORDIC, almost one fifth hardware area can be saved comparing with setting all data 48 bits.

B. Timing Analysis

Consider a general design of the proposed pipelined architecture denoted as $Arch.(m,n)$. So the iteration indexes of the improved CORDIC used in our architecture are all set as $i=-m,-m+1,\cdots,n$. First, the CORDIC of LV mode maintains the standard iterative formulas described by (3), so that each stage of LV requires one clock cycle. The total stages of LV are $(m+n+1)$. As a result, we need $(m+n+1)$ clock cycles to complete the division task. Then, every Hyperbolic CORDIC requires (m+1) non-positive indexed iterations. As shown in the lower left of Fig. 2, each non-positive indexed iteration needs two clock cycles because we have performed one-stage pipeline technique. Consequently, the entire non-positive indexed iterations need (2m+2) clock cycles. For the positive indexed iterations of Hyperbolic CORDIC as shown in the lower right of Fig. 2, each cascaded stage requests one clock cycle. Considering that repeated iterations $(i=4,7,10,13\ldots)$ are cascaded twice (example: $i=19$ in Fig. 2), so that there are $(n+\lfloor\frac{n-1}{3}\rfloor)$ stages for the positive indexed iterations of Hyperbolic CORDIC, which also means we need $(n+\lfloor\frac{n-1}{3}\rfloor)$ clock cycles in this situation. In summary, every Hyperbolic CORDIC used in $Arch.(m,n)$ requests $(2m+2+n+\lfloor\frac{n-1}{3}\rfloor)$ clock cycles. As Fig. 1 depicts, additional add operations outside the CORDIC also demand 2 additional clock cycles. To sum up, the total latency of the proposed architecture is given by

$$\begin{aligned}T_{all}&=T_{HV}+T_{LV}+T_{HR}+2\\&=\left(2m+2+n+\left\lfloor\frac{n-1}{3}\right\rfloor\right)+(m+n+1)\\&\quad+\left(2m+2+n+\left\lfloor\frac{n-1}{3}\right\rfloor\right)+2\\&=5m+3n+2\left\lfloor\frac{n-1}{3}\right\rfloor+7.\end{aligned}\tag{25}$$

As described in Section III.C, in (25), m determines the convergence range of the radicand R, the larger the m is,

the larger the convergence range; $n$determines the computing precision, the larger the n is, the higher the precision.

For the implemented special case $Arch.(2, 20)$, consider $m = 2$, n=20, and (25), the latency can be computed as

$$T_{all} = 5 \times 2 + 3 \times 20 + 2 \left\lfloor \frac{20-1}{3} \right\rfloor + 7 = 89 \text{ clock cycles.}$$

A detail latency requirement for general $Arch.(m, n)$ is outlined in Table VI. It tells us that if we do not dictate large convergence range of radicand R and high precision of computing in $Arch.(m, n)$, the latency can be massively reduced. Accordingly, the frequency will increase since the word lengths are going to decrease for lower precision computing and lower range of R, which leads to the shorter carry-chain (significant part of the critical path) for adders.

As for the critical path, it only consists of one shift operation and one add operation. Assuming the time of a shift operation is t_s ns and that of an add operation is t_a ns, the sampling rate (MSPS: Million Samples Per Second) of the pipelined structure for the proposed architecture can be computed as

$$\frac{1000}{t_s + t_a} \text{ MSPS.}$$

If the system does not require high processing speed, we can use partially folding technique to further save the area. Assume the folding factor is F, then the sampling rate is given by

$$\frac{1000}{F(t_s + t_a)} \text{ MSPS.}$$

For example, consider the CORDIC of LV mode in a special case $Arch.(1, 2)$. The iteration index is $i = -1, 0, 1, 2$. So the CORDIC of LV used in $Arch.(1, 2)$ requires 4 cascaded stages as shown in the left of Fig. 3. Now, as shown in the right of Fig. 3, we perform the partially folding technique with folding factor F as 2. With folding technique, 4 stages of CORDIC lowers to 2 stages, which can save almost half area. However, since we can only get the results at odd clock cycle, the folded structure also halves the sampling rate (frequency stays the same).

C. Implementation Results

The special case $Arch.(2, 20)$ of the proposed architecture is coded in Verilog HDL and synthesized under the TSMC 40-nm CMOS technology. The top-level architecture can be seen in Fig. 1 and the pipelined structure of sub-modules can be seen in Fig. 2. The word length setting is outlined in Table VII. This is the first time that a general Nth root computation architecture has been implemented in hardware, so we cannot find any counterpart to compare with the performance of our implementation. The prior attempts [17], [18] all require the multiplication operations, thus, we decide to use a multiplier as the benchmark. In reality, it may be better to choose a well-known multiplier, such as Conventional Array Multiplier (CAM), in this way most of the readers can directly recognize the performance of our architecture. But the critical path of CAM may be too long to keep up with the frequency of our implementation. Though Vedic and CAM are both constructed by Ripple Carry Adders (RCA), the critical path

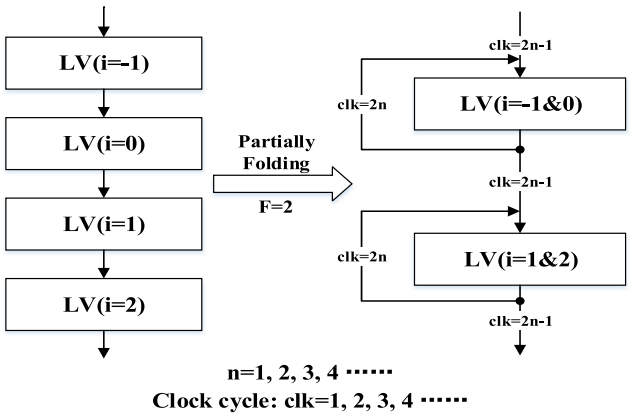


Fig. 3. Example of partially folding with folding factor F=2. The folded structure inputs a sample and outputs a result at every odd clock cycle. Two angles (constants) are pre-stored in each CORDIC of LV mode for the folded structure. At even clock cycle, the outputs of each CORDIC are returned to the input interfaces of this CORDIC to reuse the circuits.

of Vedic is shorter than CAM. Consequently, we select a 48-bit fixed-point Vedic multiplier capable of achieving high frequency, as the benchmark. The design details of Vedic can be found in [29]–[31]. In Section IV.D, we also theoretically compare the area of our example circuits with that of CAM, in case some readers may not be familiar with Vedic.

In order to evaluate the performance of our circuits, some criteria need to be specified. When the circuits of $Arch.(2, 20)$ and Vedic are running at the same frequency, we define three criteria. The first one is the area ratio A_r. Assuming the area of $Arch.(2, 20)$ and Vedic are A_I and A_V respectively, then A_r can be given by

$$A_r = \frac{A_I}{A_V}.$$

A_r denotes at a certain frequency, how much area the special case $Arch.(2, 20)$ occupies compared with a Vedic multiplier. The second criterion is the power ratio P_r. Assuming the power of $Arch.(2, 20)$ and Vedic are P_I and P_V respectively, then P_r can be given by

$$P_r = \frac{P_I}{P_V}.$$

P_r denotes at a certain frequency, how much power the special case $Arch.(2, 20)$ consumes compared with a Vedic multiplier. The third criterion is the power ratio per unit area P_r/A_r, due to the equation

$$\frac{P_r}{A_r} = \frac{(\frac{P_I}{P_V})}{(\frac{A_I}{A_V})} = \frac{(\frac{P_I}{A_I})}{(\frac{P_V}{A_V})}.$$

P_r/A_r denotes at a certain frequency, how much power per unit area of the special case $Arch.(2, 20)$ consumes compared with per unit area of a Vedic multiplier.

Another criterion is the sampling rate per watt of $Arch.(2, 20)$. Assuming the frequency is fGHz, then the sampling rate equals $1000 \times f$ MSPS (Million Samples Per Second). We further assume that the power is w mW. Thus, the

TABLE VIII
Arch.(2, 20) VERSUS VEDIC MULTIPLIER

Freq.	Item	*Arch.*(2,20)	Vedic	Ratio		P_r/A_r
1 GHz	Area(μm^2)	169689.98	27906.24	A_r	6.081	0.836
	Power(mW)	53.2978	10.4817	P_r	5.085	
1.89 GHz	Area(μm^2)	187608.46	46926.16	A_r	3.998	0.827
	Power(mW)	97.0004	29.3504	P_r	3.305	
2.083 GHz	Area(μm^2)	197421.00	52997.62	A_r	3.725	0.797
	Power(mW)	109.7480	36.9760	P_r	2.968	

a. Area ratio: A_r.
b. Power ratio: P_r.
c. Power ratio per unit area: P_r/A_r.

TABLE IX
SAMPLING RATE PER WATT FOR *Arch.*(2, 20)

FREQUENCY(GHz)	1	1.89	2.083
MSPS/W	1.8762×10^4	1.9484×10^4	1.8980×10^4

a. MSPS/W: million samples per watt.

sampling rate per watt can be given by

$$\frac{(1000 \times f)MSPS}{(w)mW} = \frac{10^6 \times f}{w} MSPS/W. \tag{26}$$

The circuits are synthesized by Design Complier and the results are presented in Table VIII. The highest frequency of the special case $Arch.(2, 20)$ is up to 2.083GHz, which means it can process 2.083 billion Nth root per second. At the highest frequency, the area consumption of $Arch.(2, 20)$ is 3.725 times than that of a Vedic multiplier; while the NR method needs many multipliers when computing higher order Nth root. With frequency increasing, area ratio A_r, power ratio P_r, and the power ratio per unit area P_r/A_r, all decline, compared with a Vedic multiplier. So we can conclude that the higher frequency the proposed architecture runs at, the higher area and power efficiency it has when compared with those architectures requesting multiplication operations.

According to the frequency and power provided in Table VIII and the formula (26), we calculate the sampling rate per watt for $Arch.(2, 20)$ and present the results in Table IX. Table IX indicates the different energy efficiency when $Arch.(2, 20)$ runs at different frequency. It tells that at the frequency 1.89GHz, the energy efficiency is better than the other two situations. It can process 19.484 billion Nth root per second and per watt, i.e., 19.484 billion Nth root per joule. Although at highest frequency, $Arch.(2, 20)$ can provide the highest sampling rate (processing speed), the corresponding energy efficiency is not the best. As a result, if we pursue the highest energy efficiency, we may not let our proposed architecture $Arch.(m, n)$ run at its highest frequency.

D. Transistor Count Analysis

In the following, we perform the transistor count (TC) analysis for $Arch.(2, 20)$ and compare with Vedic multiplier and Conventional Array Multiplier (CAM) as shown in [24], in case some readers are not familiar with Vedic. A b-bit Ripple Carry Adder (RCA) requires 24b transistors and a b-bit CAM needs 6b(5b-6) transistors (refer to [24]). So TC respect to a 48-bit CAM can be computed as

$$TC_{CAM} = 6 \times 48 \times (5 \times 48 - 6) = 67392.$$

As for TC respect to a 48-bit Vedic, it is a little bit complex. According to the structure of Vedic (see [31]), a 48-bit Vedic consists of 4 24-bit Vedic multipliers and 2 48-bit RCA adders. A 24-bit Vedic consists of 4 12-bit Vedic multipliers and 2 24-bit RCA adders. A 12-bit Vedic consists of 4 6-bit Vedic multipliers and 2 12-bit RCA adders. A 6-bit Vedic consists of 4 3-bit Vedic multipliers and 2 6-bit RCA adders. A 3-bit Vedic consists of 6 full adders. The transistor number of a 3-bit Vedic equals that of a 6-bit RCA adder. Consider TC of a 12-bit RCA equals that of 2 6-bit RCA adders, TC of a 24-bit RCA equals that of 4 6-bit RCA adders, and TC of a 48-bit RCA equals that of 8 6-bit RCA adders. Therefore, a 48-bit Vedic can be seen as using following number of 6-bit RCA adders.

$$4\,(4\,(4\,(4+2)+2\times 2)+2\times 4)+2\times 8 = 496.$$

Thus, TC respect to a 48-bit Vedic is

$$TC_{Vedic} = 496 \times 24 \times 6 = 71424.$$

Here comes the TC of $Arch.(2, 20)$. First, we assume adders used in $Arch.(2, 20)$ are RCA adders just like Vedic and CAM. By counting the number of different bit adders used in $Arch.(2, 20)$, there are 182 48-bit RCA adders, 92 38-bit RCA adders, and 181 39-bit RCA adders (refer to Fig. 1, Fig. 2 and Table VII). To sum up, TC respect to $Arch.(2, 20)$ is

$$\begin{aligned} TC_{Arch.(2,20)} &= (182 \times 48 + 92 \times 38 + 181 \times 39) \times 24 \\ &= 462984. \end{aligned}$$

Table X summarizes the results of transistor count analysis. It shows that TC of Vedic is only a bit more than that of CAM, which results in nearly the same values of TC ratio between $Arch.(2, 20)$ and multiplier. Comparing with Table VIII, we can find that at low frequency 1 GHz, the TC ratio (6.48) between $Arch.(2, 20)$ and Vedic almost equals the area ratio A_r (6.081). While increasing the frequency, A_r drastically decreases. At highest frequency, A_r lowers to 3.725. This is caused by different critical paths between $Arch.(2, 20)$ and Vedic. Due to the shorter critical path of the proposed architecture, its area increases slower than that of Vedic when increasing the frequency (the constraints become tighter).

TABLE X
TRANSISTOR COUNT (TC) ANALYSIS

TC_{CAM}	TC_{Vedic}	$TC_{Arch.(2,20)}$	$\frac{TC_{Arch.(2,20)}}{TC_{CAM}}$	$\frac{TC_{Arch.(2,20)}}{TC_{Vedic}}$
67392	71424	462984	6.87	6.48

TABLE XI
RESULTS OF THE HARDWARE TEST

Criterion	$Arch.(2,20)$
Num	200,000
Input range of R	$[10^{-6}, 10^{6}]$
Input range of N	$[2, 1002]$
Avg_Err_r	7.0544×10^{-7}
Max_Err_r	7.1630×10^{-6}
$\mathrm{P}(Err_r \geq 10^{-6})$	28.95%
$\mathrm{P}(10^{-7} < Err_r < 10^{-6})$	61.25%
$\mathrm{P}(Err_r \leq 10^{-7})$	9.80%

a. Input range of R and N are real numbers.
b. $\mathrm{P}(\cdot)$ denotes the probability.

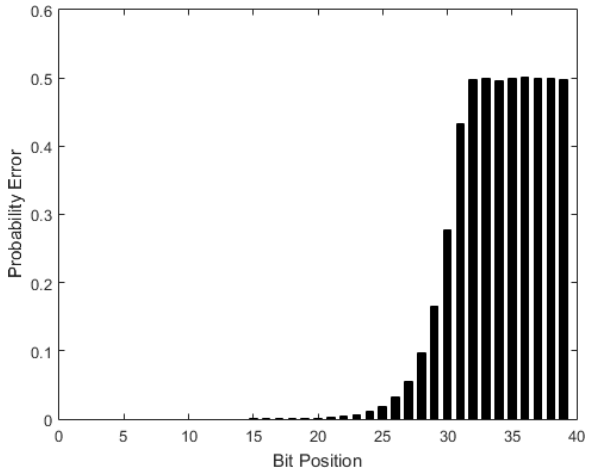


Fig. 4. Bit position error.

E. Error Analysis

The accuracy of the special case $Arch.(2, 20)$is evaluated by comparing the outputs from ModelSim with MATLAB's "$R^{1/N}$" function. Two sets of 200,000 randomly generated real numbers for N and R are taken as the inputs to the example circuits. First, as mentioned in the software test, we adopt the relative error to judge the accuracy of the circuits. The results are concluded in Table VIII. According to the results of the hardware test, the feasibility of the proposed architecture $Arch.(m, n)$ has been proven. Comparing Table XI with Table V, we can find out that the order of magnitude for Avg_Err_r and Max_Err_r keeps the same for both hardware and software tests. As for the distribution of relative error, almost 2.5% transfers from the order of magnitude -7 to that of -6, which is mainly incurred by finite word length setting in hardware.

Another commonly used metric for measuring the error is the "*bit position error*" [24]. The output of our circuits is 39-bit fixed-point data as shown in Table VII. Fig. 4 presents the results of the "*bit position error*" experiment. It is evident from Fig. 3 that precision up to 31 out of 39 bits is somewhat accurate. Although the bit error probability of the rest 8 bits all approach 0.5, it does not demonstrate those bits can be removed without affecting the precision of the implementation.

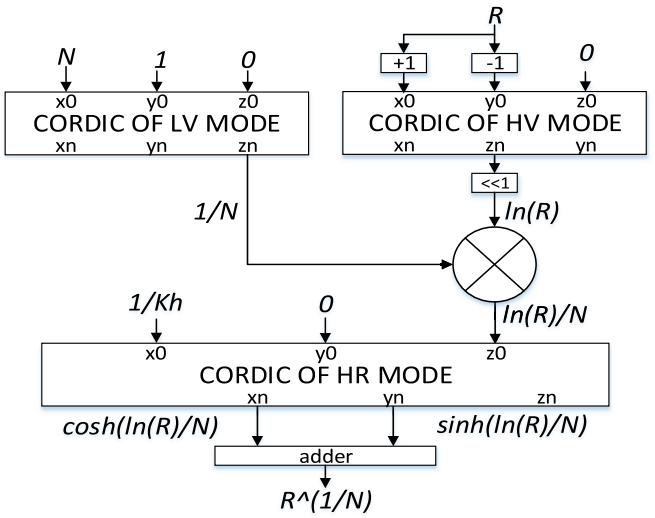


Fig. 5. Top-level architecture for $Arch^{*}.(m, n)$.

The sub-modules of the architecture need those bits to resist the propagation of error. For the other 31 bits, the most significant 14 bits are absolutely accurate. The remaining 17 bits has a possibility of being wrong. These characteristics match with that of CORDIC. In practical situation, we may only select the front 31 bits as the output, because the other 8 bits are useless in the next module or application.

V. DISCUSSION

In this section, we discuss about the existing methods for Nth root calculation and try our best to make comparison with the proposed $Arch.(m, n)$. Digit-recurrence method [16] and NR method [17] cannot guarantee a fixed VLSI implementation to compute arbitrary Nth root. Therefore, when targeted at a given N and implemented in VLSI system, we analyze their algorithms and estimate their possible hardware complexities as well as latency. Paper [18] proposes a top-level architecture for Nth root computation but is vague on details of sub-modules and lacks experiment results. Despite all this, its top-level architecture inspires us to provide a second version of $Arch.(m, n)$ which can save the latency of LV (see Fig. 5) by introducing a multiplier.

A. Discussion About Digit-Recurrence Method

Digit-recurrence method has the characteristic of producing a new digit on the completion of each iteration and it is absolutely correct for the computed bit. The mathematical essence is based on the concept of completing Nth power. Assuming S is the value of Nth root, i.e., $S = R^{1/N}$, then we set a iteration formula as

$$w\,(i) = R - S(i)^{N}.$$

In each iteration, we regulate $S(i)$ to let $w\,(i)$ approximates to zero more. After several iterations, the value of Nth root $S(i)$, can meet the precision requirement. Above is a simple explain for this method. According to paper [16], the practical

iteration formulas for radix-2 are

$$S(i) = S(i-1) + s_i 2^{-i}, \tag{27a}$$

$$w(i) = (R - S(i)^N)2^i, \tag{27b}$$

where $s_i \in \{1, 0, -1\}$. The selection of s_i is determined by the intervals that $w(i-1)$ belongs to. For different Nth root, the selection intervals are different accordingly (see [16] for more details). The convergence range of R for this method is very narrow to be $[2^{-N}, 1]$. From (27b), it is clear that for different order of Nth root, this method requires different number of multiplication operations. In addition to this, the selection intervals also change respect to different N. Hence, a fixed VLSI implementation for this method can only address the Nth root of a given N.

Here comes the hardware complexity analysis and latency analysis. For each iteration, (27) requires two adders and N-1 multipliers. Assuming the iteration number is M, then we need 2M adders and M(N-1) multipliers. For high order Nth root and high precision computation, the hardware complexity of this method is too large to deserve. In order to shorten latency, we assume the structure of multiplication operations for each iteration is constructed as multiplier tree (just like adder tree). Thus, the latency of multiplications reduces to $\lceil log_2^N \rceil$. Taken the latency of other two add operations into consideration, the latency for each iteration is $2 + \lceil log_2^N \rceil$. For M iterations, we need M$(2 + \lceil log_2^N \rceil)$ clock cycles.

Now, we try to compare with our example circuits $Arch.(2, 20)$. According to TABLE XI, the average value of relative error of the example circuits approximates $\times 10^{-7}$. It requires almost 20 iterations for digital-recurrence method to achieve this precision, i.e., M=20. Consider a high order Nth root as N=64. Then the latency of this method is

$$20\left(2 + \left\lceil log_2^{64} \right\rceil\right) = 160 \text{ clock cycles},$$

while $Arch.(2, 20)$ only requires 89 clock cycles for arbitrary order of Nth root. As for hardware complexity, it requires 20 adders and 1220 multipliers, which is much more complex than our example circuits.

B. Discussion About Newton-Raphson Method

NR method has somewhat similarity with digital-recurrence method. Compared to evaluating inverse Nth root, direct using NR method for Nth root will double the number of multiplication operations and introducing a division operation in each iteration (see paper [17] for details). Therefore, [17] first calculates inverse Nth root $(\frac{1}{R^{1/N}})$ using NR method and then performs the division operation to compute Nth root via NR method again.

Assuming $S = \frac{1}{R^{1/N}}$, then the iterative formula can be given as

$$S_{i+1} = S_i\left(\frac{1 + N - RS_i^N}{N}\right). \tag{28}$$

Performing several iterations, S_i will converge to inverse Nth root (refer to paper [17] for details). After the computation of inverse Nth root denoted as S, [17] uses NR method to calculate Nth root denoted as Y. The iterative formula for the division operation is

$$Y_{i+1} = Y_i(2 - SY_i). \tag{29}$$

Y_i will converge to $\frac{1}{S}$ through several iterations. In other words, Y_i will converge to Nth root $R^{1/N}$. For the same reason like Digit-recurrence method, NR method cannot compute arbitrary order of Nth root through a fixed VLSI implementation.

Following presents the hardware complexity analysis and latency analysis. For each iteration, (28) needs one adder and N+1 multipliers. If using multiplier tree to reduce latency, in parenthesis of (28), we need $\lceil log_2^{N+1} \rceil$ clock cycles to complete multiplication operations and one clock cycle to perform the add operation. Outside the parenthesis, one more clock cycle is needed for multiplication. In total, (28) needs $2 + \lceil log_2^{N+1} \rceil$ clock cycles. For each iteration, (29) requires one adder and two multipliers. It also needs 3 clock cycles to complete the iteration. Assuming iteration number for (28) and (29) is both M, then NR method needs 2M adders and M(N+3) multipliers. The latency is M$(5 + \lceil log_2^{N+1} \rceil)$.

In paper [17], the input range of R is set as [0.5, 1.5], and initial guess is between [0.95,1.05]. By software test provided in [17], setting iteration number as 5 for both processes produces the absolute error value as $\times 10^{-8}$. If setting iteration number as 4, the absolute error increases to $\times 10^{-6}$. Thus, in order to match the precision of our example circuits (10^{-7}), we consider the iteration number M for (28) and (29) both as 5. When computing 64^{textth} root (N=64), NR method needs 10 adders and 335 multipliers for a pipelined VLSI implementation, which is much more complex than $Arch.(2, 20)$. As for latency, it requires

$$5\left(5 + \left\lceil log_2^{64+1} \right\rceil\right) = 60 \text{ clock cycles}.$$

Benefited from NR method's quadratic convergence, its latency is less than our example circuits (89).

In reality, the computing precision of NR method is unstable for different initial guesses and different radicands R. Paper [17] is too ideal narrowing the input range of R as [0.5,1.5] and initial guess between [0.95,1.05]. If the input range of R is much larger, in order to achieve a certain precision, the iteration number (latency) is uncontrollable, especially for high order Nth root.

C. Second Version of the Proposed Architecture

The mathematical essence of paper [18] comes from the equality $R^{\frac{1}{N}} = 2^{(\frac{1}{N})log_2^R}$, which is much like ours. Therefore, its top-level architecture is similar to Fig. 1 except paper [18] computes logarithm and reciprocal in a parallel manner followed by a multiplication. Neither detailed architectures of sub-modules nor experiment results were provided. Thus, no comparison with this work is given in this paper.

The difference of our top-level architectures is an excellent example of trade-off in VLSI design domain. Compared to Fig. 1, paper [18] sacrifices area (introduce a multiplier) in hardware to achieve lower latency. Inspired by paper [18], we also present a second version of our proposed architecture,

which is denoted as $Arch^*.(m, n)$. Fig. 5 shows the top-level architecture of $Arch^*.(m, n)$. Comparing Fig. 5 to Fig. 1, it is clear that the latency of LV has been saved by introducing a multiplier.

Following analyzes the latency of $Arch^*.(m, n)$. According to Section IV.B, the latency of LV is less than that of HV. Therefore, the latency of $Arch^*.(m, n)$ can be computed by formula (25) subtracting the latency of LV and adding the latency of a multiplier. Supposing the latency of a multiplier used in Fig. 5 is one clock cycle, we can calculates the latency of $Arch^*.(m, n)$ denoted as T^*_{all}.

$$\begin{aligned} T^*_{all} &= T_{all} - T_{Lv} + T_{multiplier} \\ &= \left(5m + 3n + 2\left\lfloor \frac{n-1}{3} \right\rfloor + 7\right) - (m + n + 1) + 1 \\ &= 4m + 2n + 2\left\lfloor \frac{n-1}{3} \right\rfloor + 7. \end{aligned} \tag{30}$$

If our example circuits $Arch.(2, 20)$ are implemented using $Arch^*.(2, 20)$ shown as Fig. 5, with the penalty of introducing a multiplier, its latency will reduce to 67 clock cycles.

VI. Conclusion

In this paper, based on the improved CORDIC [28], we have proposed a CORDIC-based architecture $Arch.(m, n)$ for the general computation of Nth root and implemented the example circuits in hardware to prove its feasibility and efficiency. Under the TSMC 40-nm CMOS technology, the example circuits can achieve a frequency up to 2.083 GHz with 3.725 times the area than that of a Vedic multiplier [29]–[31]. Compared to the existing works, this is the first time that a fixed VLSI implementation can compute arbitrary Nth root with acceptable area and latency. The proposed architecture for the Nth root computation is beyond the conventional concept, because the N of $R^{\frac{1}{N}}$ is real number rather than only integer. As shown in Table VI, the proposed architecture has the flexibility in both convergence range and precision. Changing the max positive iteration index (n) results in different precision. Changing the number of non-positive iterations (m+1) leads to different convergence range. Therefore, the proposed architecture is easy to adjust for fitting the different requirements. In hardware, if speed can be compromised, a folded architecture can be chosen to save area. If high speed is required, parallel and pipelined architecture can be implemented to achieve high sampling rate. If low latency is expected, second version of the proposed architecture $Arch^*.(m, n)$ offers an alternative.

The weakness of the proposed architecture is the long latency for low order Nth root computation. The example circuits require the latency of three cascaded CORDICs. Conventional CORDIC-based square root computation only contains one CORDIC, which results in smaller latency. While computing a high-order Nth root, the fixed latency becomes a merit since prior attempts such as digit-recurrence method [16] needs much more clock cycles to generate the first output.

High order Nth roots are needed in some special areas such as volume shading for computer graphics, atmospheric models, radiance and luminance and so forth [15]. Our proposed architecture and its second version offer promising hardware-based solutions for those problems.

References

[1] C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim, "Some properties of iterative square-rooting methods using high-speed multiplication," *IEEE Trans. Comput.*, vol. TC-21, no. 8, pp. 837–847, Aug. 1972.

[2] H. Kabuo *et al.*, "Accurate rounding scheme for the Newton–Raphson method using redundant binary representation," *IEEE Trans. Comput.*, vol. 43, no. 1, pp. 43–51, Jan. 1994.

[3] M. Allie and R. Lyons, "A root of less evil," *IEEE Signal Process. Mag.*, vol. 22, no. 2, pp. 93–96, Mar. 2005.

[4] A. Seth and W.-S. Gan, "Fixed-point square roots using L-b truncation," *IEEE Signal Process. Mag.*, vol. 28, no. 6, pp. 149–153, Nov. 2011.

[5] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1059–1070, Apr. 2012.

[6] C. M. Guardia and E. Boemo, "FPGA implementation of a binary32 floating point cube root," in *Proc. 9th Southern Conf. Programm. Logic (SPL)*, Nov. 2014, pp. 1–6.

[7] J. Prado and R. Alcantara, "A fast square-rooting algorithm using a digital signal processor view document," *Proc. IEEE*, vol. 75, no. 2, pp. 262–264, Feb. 1987.

[8] N. Mikami, M. Kobayashi, and Y. Yokoyama, "A new DSP-oriented algorithm for calculation of the square root using a nonlinear digital filter," *IEEE Trans. Signal Process.*, vol. 40, no. 7, pp. 1663–1669, Jul. 1992.

[9] Y. Li and W. Chu, "On the improved implementations and performance evaluation of digit-by-digit integer restoring and non-restoring cube root algorithms," in *Proc. Int. Conf. Comput., Inf. Telecommun. Syst. (CITS)*, Jul. 2016, pp. 1–5.

[10] R. V. W. Putra and T. Adiono, "Optimized hardware algorithm for integer cube root calculation and its efficient architecture," in *Proc. Int. Symp. Intell. Signal Process. Commun. Syst. (ISPACS)*, Nov. 2015, pp. 263–267.

[11] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Boston, MA, USA: Birkhäuser, 2006.

[12] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," *IEEE Signal Process. Mag.*, vol. 9, no. 3, pp. 16–35, Jul. 1992.

[13] F. Angarita, A. Perez-Pascual, T. Sansaloni, and J. Vails, "Efficient FPGA implementation of CORDIC algorithm for circular and linear coordinates," in *Proc. Int. Conf. Field Programm. Logic Appl.*, Aug. 2005, pp. 535–538.

[14] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of CORDIC: Algorithms, architectures, and applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 9, pp. 1893–1907, Sep. 2009.

[15] A. S. Glassner, *Principles of Digital Image Synthesis*. San Mateo, CA, USA: Morgan Kaufmann, 1995.

[16] P. Montuschi, J. D. Bruguera, L. Ciminiera, and J. A. Piñeiro, "A digit-by-digit algorithm for mth root extraction," *IEEE Trans. Comput.*, vol. 56, no. 12, pp. 1696–1706, Dec. 2007.

[17] S. Aslan, H. Salamy, and J. Saniie, "A high-level synthesis and verification tool for application specific k^{th} root processing engine," in *Proc. Int. Midw. Symp. Circuits Syst. (MWSCAS)*, Aug. 2013, pp. 1051–1054.

[18] Á. Vázquez and J. D. Bruguera, "Composite iterative algorithm and architecture for q-th root calculation," in *Proc. IEEE Symp. Comput. Arith. (ARITH)*, Jul. 2011, pp. 52–61.

[19] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.

[20] J. E. Volder, "The birth of CORDIC," *J. VLSI Signal Process.*, vol. 25, no. 2, pp. 101–105, 2000.

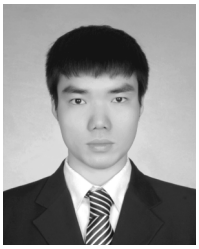
[21] J. S. Walther, "A unified algorithm for elementary functions," in *Proc. 38th Spring Joint Comput. Conf.*, Atlantic City, NJ, USA, 1971, pp. 379–385.

[22] J. S. Walther, "The story of unified CORDIC," *J. VLSI Signal Process.*, vol. 25, no. 2, pp. 107–112, Jun. 2000.

[23] B. Yang, D. Wang, and L. Liu, "Complex division and square-root using CORDIC," in *Proc. Int. Conf. Consum. Electron., Commun. Netw. (CECNet)*, Apr. 2012, pp. 2464–2468.

[24] S. Mopuri and A. Acharyya, "Low-complexity methodology for complex square-root computation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 11, pp. 3255–3259, Nov. 2017.

[25] T. Kulshreshtha and A. S. Dhar, "CORDIC-based Hann windowed sliding DFT architecture for real-time spectrum analysis with bounded error-accumulation," *IET Circuits, Devices Syst.*, vol. 11, no. 5, pp. 487–495, Sep. 2017.
[26] K. C. Ray and A. S. Dhar, "CORDIC-based parallel architecture for one dimensional discrete Mellin transform," in *Proc. IEEE Region 10th Conf. (TENCON)*, Nov. 2016, pp. 1638–1643.
[27] M. Heidarpour, A. Ahmadi, and R. Rashidzadeh, "A CORDIC based digital hardware for adaptive exponential integrate and fire neuron," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 63, no. 11, pp. 1986–1996, Sep. 2016.
[28] X. Hu, R. G. Harber, and S. C. Bass, "Expanding the range of convergence of the CORDIC algorithm," *IEEE Trans. Comput.*, vol. 40, no. 1, pp. 13–21, Jan. 1991.
[29] A. P. James, D. S. Kumar, and A. Ajayan, "Threshold logic computing: Memristive-CMOS circuits for fast Fourier transform and vedic multiplication," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 11, pp. 1694–2690, Jan. 2015.
[30] A. Kanhe, S. K. Das, and A. K. Singh, "Design and implementation of floating point multiplier based on vedic multiplication technique," in *Proc. Int. Conf., Commun., Inf., Comput. Technol. (ICCICT)*, Oct. 2012, pp. 1–4.
[31] Y. Bansal, C. Madhu, and P. Kaur, "High speed vedic multiplier designs-A review," in *Proc. Recent Adv. Eng. Comput. Sci. (RAECS)*, Mar. 2014, pp. 1–6.



Yuanyong Luo received the B.S. degree in applied physics from Jilin University, Changchun, China, in 2016. He is currently pursuing the Ph.D. degree with the School of Electronic Science and Engineering, Nanjing University, China. From 2014 to 2015, he hosted an engineering project, the system of aerial photography with three cooperative and unmanned aerial vehicles, at Jilin University. His current research interests include digital integrated circuit design and VLSI implementation of signal processing algorithms, machine learning algorithms, and digital communications.

Yuxuan Wang received the B.S. degree in electronic science and engineering from Nanjing University, Nanjing, China, in 2015. He is currently pursuing the M.Sc. degree with the School of Electronic Science and Engineering, Nanjing University. His current research interests include VLSI computing IP optimization and the coordinated acceleration of software and hardware in machine learning.

Huaqing Sun received the B.S. degree in electronic information engineering from Sichuan University, Chengdu, China, in 2017. He is currently pursuing the master's degree with the School of Electronic Science and Engineer, Nanjing University, China. His current research interests include reconfigurable computing and the efficient hardware implementations for machine learning algorithms.

Yi Zha received the B.S. degree in electronic engineering from Jinling College, Nanjing, China, in 2017. She is currently pursuing the master's degree with the School of Electronic Science and Engineering, Nanjing University, China. From 2011 to 2015, she participated in two robot competitions, achieved the ideal ranking, and published an academic paper. Her current research interests include VLSI for digital signal processing systems, reconfigurable computing, and the hardware implementations for machine learning algorithms.

Zhongfeng Wang (F'16) received the B.E. and M.S. degrees from Tsinghua University, Beijing, China, and the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, in 2000. He was with National Semiconductor Corporation, Santa Clara, USA. He was an Assistant Professor with the School of EECS, Oregon State University, Corvallis. He joined Nanjing University in 2016 as a Distinguished Professor through the State's 1000-talent plan after serving Broadcom Corporation as a leading VLSI Architect for nearly nine years.

He is a world-recognized expert on VLSI for Signal Processing Systems. In the current record (since 2007), he has had five papers ranked among top 20 most downloaded manuscripts in the IEEE TRANSACTIONS ON VLSI SYSTEMS. During his tenure at Broadcom, he has contributed significantly on 10 Gbps and beyond high-speed networking products. He has made critical contributions in designing FEC coding schemes for 100 and 400 Gbps Ethernet standards. His technical proposals have been adopted by many international networking standards. He has published over 160 technical papers, edited one book (VLSI), and filed tens of U.S. patent applications and disclosures. He was a recipient of the IEEE Circuits and Systems Society VLSI Transactions Best Paper Award in 2007.

His current research interests are in the area of digital communications, machine learning, and efficient VLSI implementation. He has served as a technical program committee member (or co-chair), the session (or track) chair, and a review committee member for tens of international conferences. Since 2004, he has served as an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS-I (TCAS-I), TCAS-II, and the IEEE TRANSACTIONS ON VLSI SYSTEMS, for numerous terms. He was a Guest Editor for a special issue of the IEEE JOURNAL ON EMERGING AND SELECTED TOPICS IN CIRCUITS AND SYSTEMS in 2016. In 2013, he served in the Best Paper Award Selection Committee for the IEEE Circuits and System Society.

Hongbing Pan received the B.S. degree in applied physics and the Ph.D. degree in microelectronics and solid state electronics from Nanjing University, 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, Nanjing, China. He has authored over 40 articles. His research interests include VLSI design, CMOS sensors, reconfigurable computing, and artificial intelligence.