Symmetric-Mapping LUT-Based Method and Architecture for Computing X^Y-Like Functions

Hui Chen, *Graduate Student Member, IEEE*, Heping Yang, Wenqing Song, *Student Member, IEEE*, Zhonghai Lu, *Senior Member, IEEE*, Yuxiang Fu, *Member, IEEE*, Li Li, and Zongguang Yu

***Abstract*— We propose a new method and hardware architecture to compute the functions expressed as X^Y (X and Y are arbitrary floating-point numbers), which can support arbitrary Nth root, exponential and power operations. Because of the complexity of direct computation, we usually convert it to logarithm, multiplication, and antilogarithm operations. Traditional approaches suffer from long latency, large area and high power consumption. To solve this problem, we propose a symmetric-mapping lookup table (SM-LUT) to be capable of computing $\log_2 x$ ($x \in [1, 2]$) and 2^x ($x \in [0, 1]$) simultaneously. It lays the foundation for computing X^Y. To further improve hardware performance of our architecture, we propose a multi-region address searcher to speed up the calculation of SM-LUT. In addition, we use an optimized Vedic multiplier to shorten the critical path and improve the efficiency of multiplication, which is included in computing X^Y. Under the TSMC 40nm CMOS technology, we design and synthesize a reference circuit to compute X^Y with a maximum relative error of 10^{-3}. The report shows that the reference circuit achieves the area of 14338.50 μm^2 and the power consumption of 4.59 mW at the frequency of 1 GHz. In comparison with the state-of-the-art work under the same input range and similar precision, it saves 78.57% area and 80.42% power consumption for $\sqrt[N]{R}$ computation and 82.89% area and 81.89% power consumption for R^N computation averagely. On top of that, our architecture reduces the computation latency by 62.77% averagely and has one more order of magnitude of energy efficiency than others.**

***Index Terms*— Floating-point, X^Y-like functions, symmetric-mapping LUT, multi-region address searcher, Vedic multiplier.**

I. Introduction

FUNCTIONS expressed as X^Y are important operations in applications such as scientific computing, digital signal processing (DSP) and computer 3D graphics [1]. Although software routines can provide accurate results, they are often too slow for numerically intensive and real-time applications [2]. The timing constraints of these applications have led to the development of customized hardware design for computing functions like X^Y. But computing arbitrary X^Y in a unified hardware circuit and meeting the real-time requirements are considered difficult due to its complexity and prohibitive hardware requirements. On the other hand, the rapid development of integrated circuit industry leads to the emergence of the reconfigurable domain-specific accelerators [3] and DSP processors [4]. As the basic operations, N*th* root, power operations and exponential operations are often included at the same time, such as reconfigurable domain-specific processor [3]. If they are implemented separately, the hardware cost could be very high. Currently, the approaches to compute N*th* root, power operations, and exponential operations can be divided into the following categories: LUT-based [5], [6], functional iteration [7], [8], digit-recurrence [9]–[11] and CORDIC-based [12]–[14]. In addition to ASIC implementation, there is also FPGA implementation, such as [13], [15].

However, there is not much research on universal computing. Because some approaches are very restrictive, such as the LUT-based approach, where supporting large input ranges can be costly in terms of hardware. In the latest research, Luo *et al.* [12] firstly proposed a traditional-CORDIC based approach to compute N*th* root for any fixed-point number, while Wang *et al.* [14] aimed at any floating-point number for N*th* computation based on the generalized hyperbolic CORDIC (GH-CORDIC). Mopuri and Acharyya [13] also did similar work. Although his approach is based on the binary hyperbolic CORDIC, it is actually a special form of GH-CORDIC (base is 2). Therefore, Suresh's approach to compute N*th* root is similar to [14]. Besides, the input data types (R and N) are not uniform in Suresh's work. R is a floating-point number but N is a fixed-point number, a combination that may not be applicable in practice.

From the above analysis, it can be seen that the general architectures of computing X^Y-like functions are all based on CORDIC. However, CORDIC requires more iterations in order to achieve higher accuracy and sometimes needs to extend the negative iterations to achieve larger input range. Even if the CORDIC-based approaches to build such a general architecture are feasible, the area and power consumption are large, especially the long computation latency, which are often not acceptable in practical applications. To address this

Manuscript received July 12, 2020; revised September 13, 2020; accepted December 20, 2020. Date of publication January 8, 2021; date of current version February 23, 2021. This work was supported in part by the National Nature Science Foundation of China under Grant 61176024 and in part by the Nanjing University Technology Innovation Fund. This article was recommended by Associate Editor T. Hanyu. *(Corresponding authors: Yuxiang Fu; Li Li.)*

Hui Chen, Heping Yang, Wenqing Song, Yuxiang Fu, Li Li, and Zongguang Yu are with the School of Electronic Science and Engineering, Nanjing University, Nanjing 210023, China (e-mail: winnerchan@smail.nju.edu.cn; yuxiangfu@nju.edu.cn; lili@nju.edu.cn).

Zhonghai Lu is with the Department of Electrical Engineering, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 16440 Stockholm, Sweden.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2020.3046783.

Digital Object Identifier 10.1109/TCSI.2020.3046783

problem, we propose a new approach to compute arbitrary X^Y-like functions for floating-point numbers, achieving low area, low computation latency and high energy efficiency.

On the whole, this paper makes the following contributions.

- We develop a new method and hardware architecture to compute arbitrary floating-point X^Y (both X and Y are variables) for the first time. It includes three types of functionality: Nth root, exponential and power operations. In our approach, the core step of computing X^Y will be converted to the calculation of $log_2 x$ and 2^x.
- The key idea of our method is based on symmetric-mapping LUT (SM-LUT) to compute $log_2 x$ $(x \epsilon [1, 2])$ and 2^x $(x \epsilon [0, 1])$ simultaneously. This idea can be extended to any pair functions that are symmetric about the line $y = x$. The approach is superior as long as the input range is limited and both functions are used in the same architecture. It can not only save the area and power consumption, but also have the advantage of low computation latency.
- To further improve hardware performance of our architecture, a multi-region address searcher is proposed to quickly find the result data of SM-LUT, which has a lower latency than the serial sequential search or binary search method. In addition, we adopt the Vedic multiplier [16] to optimize the multiplication operation to shorten the critical path and greatly reduce the hardware area.
- With the same input range and similar accuracy, we evaluate our architecture and the state-of-the-art architectures from different aspects, which shows varying degrees of superiority.

The rest of this paper is organized as follows. Section II provides the necessary theoretical background. Section III proposes the methodology to compute arbitrary floating-point X^Y. Section IV details the important modules of the proposed architecture. Section V sets experimental comparison conditions, including the same input range and similar accuracy. Under this uniform condition, Section VI makes a comparative analysis of word length setting, computational complexity and latency with other latest approaches, including the most important hardware experimental results. Finally, Section VII draws the conclusions.

II. Theoretical Background

In all standard mathematical functions, there exist three operations: Nth root, exponential and power operations.

Nth root: $y = x^{\frac{1}{N}}$, where N ranges from integers greater than or equal to 2. If N is an even number, x has to be a positive number. Conversely, when N is an odd number, the range of x is any real number.

Exponential function: $y = a^x$. As a rule, a is a constant greater than 0 but not equal to 1, x is any real number.

Power function: $y = x^a$. In general, a is a rational number. Therefore, the function can be expressed as $y = x^{\frac{m}{n} \times (-1)^k}$, where m, n, and k are all positive integers. In particular, m and n are prime. Since the range of x is related to the values of m, n and k and is complicated, we will not discuss it here.

It can be seen from the above three mathematical operations, their forms of expression and input range are different. If all three operations are required for an accelerator, separate design will result in a long time. What's more, the variable input and output ranges can also make the design logic complex. The ideal way is to express all three operations in terms of X^Y, with an unlimited range of inputs for both X and Y. In hardware architectures, fixed-point values tend to represent a limited data range, whereas floating-point values represent a wider range, so our work focuses on floating-point inputs.

The generic approach for X^Y computation is based on the natural logarithm-exponential relation (LER): $X^Y = exp(ln(X) \times Y)$, where the logarithm and exponential operations are performed in various methods. The disadvantage of this approach is that it is difficult to directly calculate the logarithm of the floating-point number X, and the floating-point exponential operation is also complicated.

In fact, X^Y can be converted to LER with an arbitrary base. According to the rule of IEEE-754 standard, any floating-point number D can be expressed as

$$\begin{aligned} &D = (b_{31} b_{30} \cdots b_1 b_0)_2, \\ &D = (-1)^S \times 1.M \times 2^K, \quad K = E - 127, \end{aligned} \tag{1}$$

where S is the sign bit (b_{31}), M is the mantissa $(b_{22} b_{21} \cdots b_1 b_0)_2$, and E is the exponent $(b_{30} b_{29} \cdots b_{24} b_{23})_2$.

$$S = 0 or 1, \quad M \epsilon [0, 1), \quad E \epsilon [0, 255]. \tag{2}$$

Now, we detail the method to compute X^Y for two floating point operands: X and Y. Assuming that X and Y are both positive numbers, $X = 1.M_1 \times 2^{K_1}$ and $Y = 1.M_2 \times 2^{K_2}$. X^Y can be computed as follows:

$$\begin{aligned} X^Y &= 2^{log_2(X^Y)} = 2^{Y \times log_2(1.M_1 \times 2^{K_1})}, \\ &= 2^{Y \times (K_1 + log_2(1.M_1))}. \end{aligned} \tag{3}$$

For unity, the output should also be a floating-point number, that is, if $X^Y = 1.M_o \times 2^{K_o}$. To achieve this, we separate the result of $Y \times (K_1 + log_2(1.M_1))$ $(= R)$ and $R = R_I + R_F$, where R_I and R_F represent the integer and fractional parts of R, respectively. If so, X^Y can be further computed by

$$X^Y = 2^R = 2^{R_I + R_F} = 2^{R_F} \times 2^{R_I} = 1.M_o \times 2^{K_o}. \tag{4}$$

In practical terms, the result of X^Y must be in the floating-point range, so $R_I = K_o$ must be in $[-127, 128]$. Because $R_F \epsilon [0, 1)$, 2^{R_F} must be equal to $1.M_o$, $M_o \epsilon [0, 1)$.

Therefore, the computation process of X^Y can be decomposed into a series of operations consisting of the logarithm of the mantissa $1.M_1$, one addition by K_1 and one multiplication by Y, and the exponential of R_F.

Although the input Y is a floating-point number, we can use (1) to convert it to a fixed-point number, so the multiplication in (3) is a fixed-point operation. The integer bit width of the fixed-point multiplier depends on the value of Y and the result of the multiplication.

III. Proposed Methodology

From (3) and (4), we know that $log_2 x$ and 2^x calculations play the most important role in computing X^Y. In this section, we first introduce the symmetry property of $log_2 x$ and 2^x and its computation method. Second, we present the

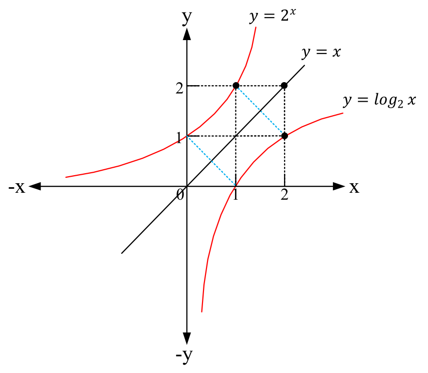


Fig. 1. Geometric relationship between the functions $y = 2^x$ and $y = log_2 x$.

theory of SM-LUT based method to calculate $log_2 x$ and 2^x efficiently. Since the area of $log_2 x$ and 2^x calculated by SM-LUT based method is small, in order to further improve the hardware performance of X^Y computation, we finally propose a multi-region parallel search method to quickly find the output result of SM-LUT.

A. $log_2 x$ and 2^x Computation

Mathematically, the inverse function of a logarithm function is an exponential function. Fig. 1 shows the geometric relationship between the two functions $y = 2^x$ and $y = log_2 x$. When x is a member of $[0, 1]$, the codomain of $y = 2^x$ is $[1, 2]$. And when x is in the range $[1, 2]$, the codomain of $y = log_2 x$ is $[0, 1]$. $y = x$ is the axis of symmetry of these two functions. In other words, $y = 2^x$ can actually be viewed as $x = log_2 y$. The symmetric relation of inverse functions of each other provides theoretical support for us to invent SM-LUT, which can be further used to compute X^Y.

In hardware, LUT is considered as an easy way to implement certain functions. If some functions can be implemented simply using LUT, it will have the advantages of low latency and reduced complexity. Furthermore, if the number of data stored in LUT is small, the hardware area is also low. Therefore, it is strict to implement a particular function entirely with LUT. The first condition is that it can be implemented with LUT, and the second is that the range of input and output cannot be large. Otherwise, the advantages of LUT will no longer exist, but become very clumsy.

On the other hand, a direct implementation with LUT will introduce some precision errors. In order to achieve high accuracy without affecting the whole logic function, we need to analyze the feasibility of LUT-based method. Precision, range, and area are the three elements that need to be weighed most when using the LUT-based method. They determine whether the method is worth adopting.

From the theoretical analysis of the previous section, only 2^x $(x \epsilon [0, 1))$ and $log_2 x$ $(x \epsilon [1, 2))$ need to be calculated when computing X^Y. These two functions are not only symmetric, but also the domain and the codomain are exactly reciprocal. This gives us the first advantage of using LUT to implement them. Second, small input and output ranges can make the hardware area smaller and also provide the possibility for high-precision implementation.

TABLE I
LUT SIZE ANALYSIS UNDER DIFFERENT PRECISION

OMP	AE	DA	B_{AD}	LUT size
10^{-2}	$2^{-4} = 6.25 \times 10^{-2}$	16	64 bits	8B
10^{-3}	$2^{-7} = 7.81 \times 10^{-3}$	128	896 bits	112B
10^{-4}	$2^{-10} = 9.76 \times 10^{-4}$	1024	10240 bits	1.25KB
10^{-5}	$2^{-14} = 6.10 \times 10^{-5}$	16384	229376 bits	28KB
⋮	⋮	⋮	⋮	⋮

1B (byte) = 8 bits, 1KB (Kilobyte) = 2^{10}B = 1024B = 8192 bits.
OMP: Order of Magnitude of Precision.

B. SM-LUT Based Method

In this subsection, we first analyze the storage costs of using LUT to implement functions of varying precision. Then, we detail how to use LUT to implement both 2^x and $log_2 x$ calculations involved in computing X^Y. Finally, we take advantage of the symmetry of 2^x and $log_2 x$, through mathematical transformation and algorithm requirements, to propose a new LUT-based method to implement them, which is called SM-LUT based method in this paper.

The principle of the LUT method is to store pre-computed data, and the bit width and amount of data determine the accuracy of the result. If the output range is $[OR_m, OR_n]$ and the absolute error (AE) is required to be 2^{-b} $(b = 1, 2, 3, \cdots)$, the data amount (DA) to be stored in LUT will be

$$DA = (OR_n - OR_m) \times AE^{-1} + 1. \quad (5)$$

When each data has B_I integer bits and B_F fractional bits, the bits of all data (B_{AD}) will be

$$B_{AD} = DA \times (B_I + B_F). \quad (6)$$

That is

$$B_{AD} = [(OR_n - OR_m) \times 2^b + 1](B_I + b), \quad (7)$$

where B_F is equal to b.

For instance, if we want to compute $log_2 x$ and 2^x with an accuracy of 6.25×10^{-2} $(=2^{-4})$, we need at least 4 bits of decimal place. Take the output range of $log_2 x$ for example, $OR_m = 0$, $OR_n = 1$ (1 is not included), DA will be 16. Each data does not need an integer bit, so $B_I = 0$ and B_F is 4. B_{AD} will be 64. Similarly, the details of other accuracies can be obtained, as shown in Table I (Case: $OR_m = 0$, $OR_n = 1$, 1 is not included). From the above table, we know that when the order of magnitude of accuracy is changed from -2 to -5, the size of LUT will be increased approximately by 3584 (229376/64) times.

Based on the above analysis, if 2^x $(x \epsilon [0, 1))$ and $log_2 x$ $(x \epsilon [1, 2))$ are implemented completely with LUT, and the accuracy is 9.76×10^{-4}, the size of LUT to store results will be 2.5KB. Each result corresponds to an input node. Therefore, if we also store these nodes in LUT, the total LUT size will be 5KB. The relationship between these nodes and the results is shown in Fig. 2(a). For example, y_{1a}, y_{1b}, y_{2a}, and y_{2b} are the results of 2^x and $log_2 x$, respectively, which are stored in 2^{nd} LUT and 4^{th} LUT. x_{1a}, x_{1b}, x_{2a}, and x_{2b} are the corresponding

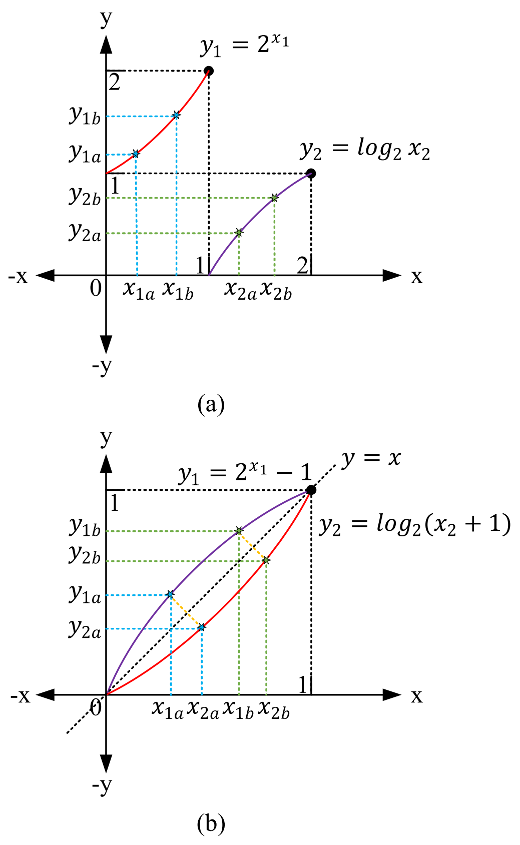


Fig. 2. Relationship between input nodes and output results in $y = 2^x$ and $y = log_2 x$. (a) The relationship under the original functions. (b) The relationship under the shifted functions.

TABLE II
DATA IN FOUR DIFFERENT LUTS OF 2^x AND $log_2 x$ COMPUTATION

Number	1st LUT	2nd LUT	3rd LUT	4th LUT
	x_1	y_1	x_2	y_2
1	0	1	1	0
2	$log_2(1+\frac{1}{DA})$	$1+\frac{1}{DA}$	$2^{(\frac{1}{DA})}$	$\frac{1}{DA}$
3	$log_2(1+\frac{2}{DA})$	$1+\frac{2}{DA}$	$2^{(\frac{2}{DA})}$	$\frac{2}{DA}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$log_2(1+\frac{i}{DA})$	$1+\frac{i}{DA}$	$2^{(\frac{i}{DA})}$	$\frac{i}{DA}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
DA-1	$log_2(1+\frac{DA-2}{DA})$	$1+\frac{DA-2}{DA}$	$2^{(\frac{DA-2}{DA})}$	$\frac{DA-2}{DA}$
DA	$log_2(1+\frac{DA-1}{DA})$	$1+\frac{DA-1}{DA}$	$2^{(\frac{DA-1}{DA})}$	$\frac{DA-1}{DA}$

$y_1 = 2^{x_1}$, $y_2 = log_2(x_2)$.

input values of 2^x and $log_2 x$, respectively, which are stored in 1st LUT and 3rd LUT. If the results of 2^x and $log_2 x$ are equally spaced, the input values will be non-equally spaced. They are shown in Table II, where $i = 1, 2, 3, \cdots, DA$, DA is computed by (5).

However, it is not an optimal solution to construct four LUTs directly. According to the particularity of the inverse function and the limitation of the computational range, we propose a better method to calculate them. Because

$$\begin{aligned} \psi &= 2^{\nu}, \quad \nu \,\epsilon\, [0, 1), \\ \nu &= log_2 \psi, \quad \psi \,\epsilon\, [1, 2), \end{aligned} \tag{8}$$

if $\psi = \phi + 1$, then

$$\begin{aligned} \phi + 1 &= 2^{\nu}, \quad \nu \,\epsilon\, [0, 1), \\ \nu &= log_2(\phi + 1), \quad \phi \,\epsilon\, [0, 1). \end{aligned} \tag{9}$$

On the basis of "left move plus and right move minus, up move plus and down move minus" in the mathematical function property, we shift down the curve 2^{x_1} in Fig. 2(a) by 1, and the curve $log_2 x_2$ is shifted 1 to the left, the two new curves are shown in Fig. 2(b).

Suppose that (x_{1a}, y_{1a}) and (x_{2a}, y_{2a}) are the inverse points of functions $(2^{x_1} - 1)$ and $log_2(x_2 + 1)$, so

$$x_{1a} = y_{2a}, \quad y_{1a} = x_{2a}. \tag{10}$$

The midpoint between these two points is

$$(\frac{x_{1a} + y_{1a}}{2}, \frac{y_{1a} + x_{1a}}{2}) or (\frac{x_{2a} + y_{2a}}{2}, \frac{y_{2a} + x_{2a}}{2}). \tag{11}$$

Apparently, the midpoint is right on the line $y = x$, so $(2^{x_1} - 1)$ and $log_2(x_2+1)$ are still symmetric about $y = x$. This property is important because it allows the number of LUTs to be reduced from 4 to 2 (the LUT size remains the same). The 1st LUT still stores equally spaced values but the 2nd LUT stores non-equally spaced values. Therefore, after optimization, the results of $log_2 x$ are evenly spaced, but the results of 2^x are non-evenly spaced. These data are shown in Table III (1st step). We can use this property to reduce the hardware area of computing 2^x and $log_2 x$ by half.

TABLE III
DATA IN SM-LUT

Number	1st step		2nd step	
	x_1 or y_2	y_1 or x_2	x_1 or y_2	y_1 or x_2
	$1^{st}LUT$	$2^{nd}LUT$	$1^{st}LUT$	$2^{nd}LUT$
1	0	1	0	0
2	$\frac{1}{DA}$	$2^{(\frac{1}{DA})}$	$\frac{1}{DA}$	$2^{(\frac{1}{DA})}$-1
3	$\frac{2}{DA}$	$2^{(\frac{2}{DA})}$	$\frac{2}{DA}$	$2^{(\frac{2}{DA})}$-1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$\frac{i}{DA}$	$2^{(\frac{i}{DA})}$	$\frac{i}{DA}$	$2^{(\frac{i}{DA})}$-1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
DA-1	$\frac{DA-2}{DA}$	$2^{(\frac{DA-2}{DA})}$	$\frac{DA-2}{DA}$	$2^{(\frac{DA-2}{DA})}$-1
DA	$\frac{DA-1}{DA}$	$2^{(\frac{DA-1}{DA})}$	$\frac{DA-1}{DA}$	$2^{(\frac{DA-1}{DA})}$-1

1st step: after the symmetric mapping.
2nd step: after the shift transformation.
$y_1 = 2^{x_1}$, $y_2 = log_2(x_2)$.

The shift transformation is beneficial since the hardware implementation only needs LUTs to store the data between [0, 1) instead of [0, 1) and [1, 2), which are shown in Table III (2nd step). When we need to calculate 2^x ($x \,\epsilon\, [0, 1)$), we take the 2nd LUT as the result lookup table and the 1st LUT as the judgment node lookup table. The LUT result needs to be added by 1 as the final calculation result. Reversely, when calculating

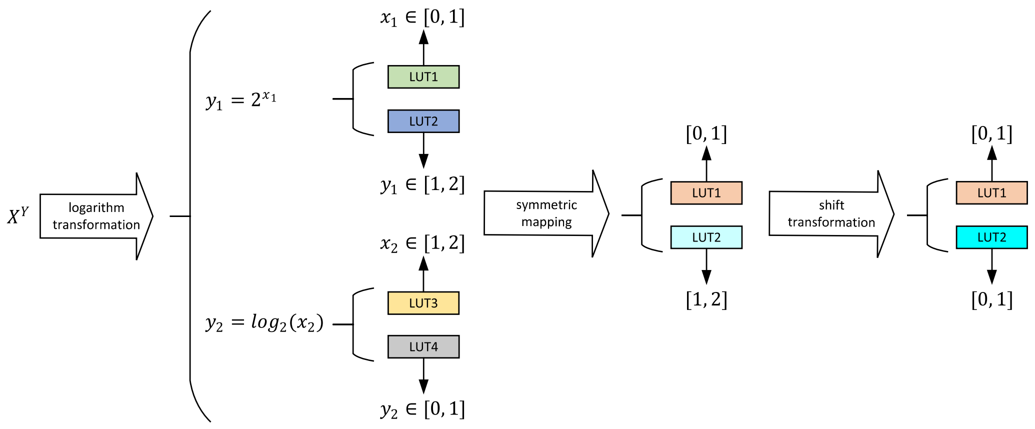
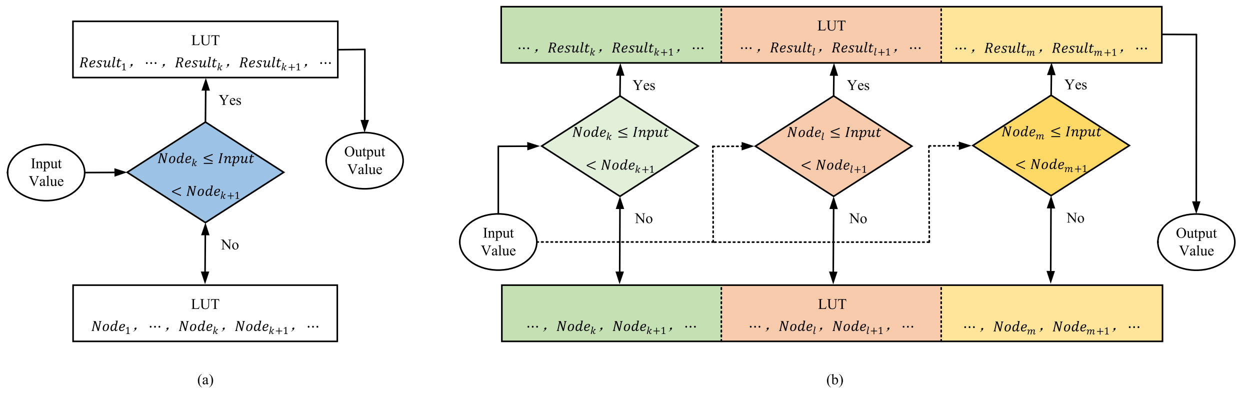


Fig. 3. Creation of SM-LUT.



LUT
$Result_1, \cdots, Result_k, Result_{k+1}, \cdots$
Yes
Input Value
$Node_k \le Input < Node_{k+1}$
Output Value
No
LUT
$Node_1, \cdots, Node_k, Node_{k+1}, \cdots$
(a)
LUT
$\cdots, Result_k, Result_{k+1}, \cdots$
$\cdots, Result_l, Result_{l+1}, \cdots$
$\cdots, Result_m, Result_{m+1}, \cdots$
Yes
$Node_k \le Input < Node_{k+1}$
$Node_l \le Input < Node_{l+1}$
$Node_m \le Input < Node_{m+1}$
No
Input Value
Output Value
LUT
$\cdots, Node_k, Node_{k+1}, \cdots$
$\cdots, Node_l, Node_{l+1}, \cdots$
$\cdots, Node_m, Node_{m+1}, \cdots$
(b)

Fig. 4. (a) The basic method: serial sequential search. (b) The multi-region method: parallel search.

$log_2 x$ ($x \in [1,2)$), the 2nd LUT is used as the judgment node lookup table, and the 1st LUT is used as the result lookup table. At this point, the input needs to be subtracted 1 to compare with the judgment node. In fact, Eq. (3) tells us that "minus 1" is not needed in the actual hardware implementation because the result is exactly the mantissa of X. The above creation process is illustrated in Fig. 3.

C. Multi-Region Parallel Search Method

When it comes to LUT-based methods, data search is inevitably involved, which usually includes three search methods: serial sequential search, binary search and parallel search. Serial sequential search method is often oriented to situations where there is a small amount of data in LUT. Although binary search method reduces the number of searches compared with serial sequential search method, there are worst-case and best-case scenarios for the search process, which are unstable. The resource consumption of parallel search is large, but the search speed is fast. Next, we briefly analyze the advantages and disadvantages of the three methods, and then discuss the proposed multi-region parallel search method.

The basic serial method is shown in Fig. 4(a), which compares the judgment nodes sequentially. Suppose that the judgment nodes and calculation results stored in LUTs are

$$LUT_1 : Node_1, \cdots, Node_k, Node_{k+1}, \cdots, Node_{DA},$$
$$LUT_2 : Result_1, \cdots, Result_k, Result_{k+1}, \cdots, Result_{DA}.$$

When the input value is determined to be between $Node_k$ and $Node_{k+1}$, the output value will be $Result_k$. This serial sequential search method is only suitable for a small size of LUT, because for DA judgment nodes, it needs at least one comparison and at maximum $(DA - 1)$ comparisons. Binary search method is similar to serial sequential search method, except that the judgment node selected each time is the middle point of the current possible interval. This method needs at least one comparison and at maximum $(DA/2)$ comparisons. Although the above methods use less logical resources in hardware implementation, the search time can vary a lot. Therefore, in order to achieve efficiency in both area and speed, we propose a multi-region parallel search method to quickly find the best result of 2^x or $log_2 x$.

As shown in Fig. 4(b), the multi-region search method is no longer a serial comparison, but to divide all the data into many regions in advance to narrow the search scope. At first, we determine which region the input value x is in. Then we do a parallel search within the small region that we locate. Suppose the judgment nodes in the LUT_1 are classified into $\Re$ parts and they are

$$RG_1 : \cdots, Node_k, Node_{k+1}, \cdots,$$
$$\vdots$$
$$RG_\hbar : \cdots, Node_l, Node_{l+1}, \cdots,$$
$$\vdots$$
$$RG_\Re : \cdots, Node_m, Node_{m+1}, \cdots,$$

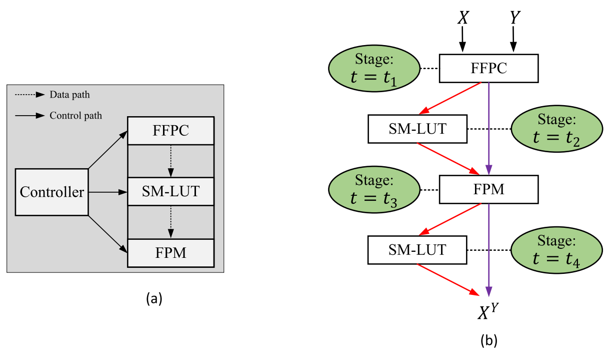


Fig. 5. (a) Hardware architecture of X^Y. (b) Computation process of X^Y.

where $RG_\hbar$ represents the $\hbar th$ region. The search process of this method is as follows:

1) Read the first judgment node from each region $(RG_1, \cdots, RG_\Re)$ at the same time, then compare them with the input value, respectively.

2) Express all first nodes as $Node_{RG_{11}}, \cdots, Node_{RG_{\Re 1}}$ in order, we can regularly find out which region $(RG_\hbar)$ x is in according to the condition $Node_{RG_{\hbar 1}} \leq x < Node_{RG_{(\hbar+1)1}}$.

3) Within the region $Node_{RG_{\hbar 1}}$ found in 2), read multiple nodes $Node_{RG_{\hbar 1}}, \cdots, Node_{RG_{\hbar k}}$ $(k = 1, 2, \cdots)$ in parallel, then compare them with the input value x, respectively.

4) When $Node_{RG_{\hbar k}} \leq x < Node_{RG_{\hbar(k+1)}}$, pick and output $Result_{RG_{\hbar k}}$ from LUT_2.

5) Once the calculation result is locked, it will be the final value of 2^x or $log_2 x$.

IV. Proposed Architecture

A. Overview

Fig. 5 describes the overall architecture of computing X^Y based on the proposed methodology. The controller module controls the three modules on the data path: floating-to-fixed point conversion (FFPC) module, SM-LUT module and fixed-point multiplier (FPM) module. It is not only used to cascade the three modules but also performs task scheduling. Its scheduling process is divided into the following four stages:

1) $t = t_1$: When the input X and Y are valid, the FFPC module first separates the exponent K_1 and mantissa M_1 from X, and convert the floating-point Y to a fixed-point Y. The fixed-point Y is sent to FPM module after a certain delay and M_1 is transferred to SM-LUT module for "$log_2(1 + M_1)$" calculation. The result of SM-LUT module is added to K_1 as another input to FPM module.

2) $t = t_2$: Suppose the judgment nodes of 2^x are stored in LUT_1 and the calculation results are stored in LUT_2. The SM-LUT module will first find the region where x is in LUT_1, then determine the specific address and extract the corresponding data for output from LUT_2.

3) $t = t_3$: The FPM module multiplies the fixed-point Y with $log_2(1 + M_1)$ to get a fixed-point result. The integer part of the result (R_I) plus 127 will be used as the exponent of the final floating-point result of X^Y, and the fractional part (R_F) will be fed to the SM-LUT module again.

4) $t = t_4$: The SM-LUT module at this stage will be used to calculate 2^{R_F}, where the data stored in LUT_1 will be deemed as the calculation results, and the data stored in LUT_2 will be regarded as the judgment nodes. The computational process is the same as that in the second stage $(t = t_2)$, the fractional part of the result will be directly used as the mantissa of X^Y.

Next, we will elaborate on these modules in detail.

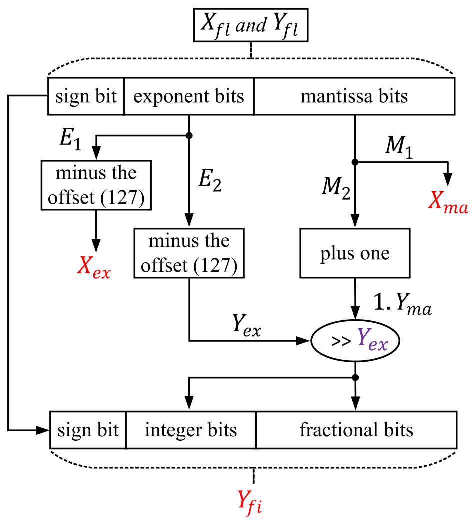


Fig. 6. Hardware architecture of the FFPC module.

B. FFPC Module

When the floating-point input X_{fl} $(b^x_{31}b^x_{30}\cdots b^x_1 b^x_0)$ and Y_{fl} $(b^y_{31}b^y_{30}\cdots b^y_1 b^y_0)$ are valid, the module first separates the exponent E_1 (E_2) and the mantissa M_1 (M_2) from X (Y). Assuming that the fixed-point outputs of FFPC module are Y_{fi} (1 sign bit, 3 integer bits and 27 fractional bits, this will be explained in the later section), exponent X_{ex} (1 sign bit and 7 integer bits), and mantissa X_{ma} (23 fractional bits). Their results are as follows:

$$\begin{aligned} X_{ex} &= (b^x_{30}\cdots b^x_{24}b^x_{23})_2 - 8'b01111111, \\ X_{ma} &= (b^x_{22}\cdots b^x_1 b^x_0)_2, \\ Y_{ex} &= (b^y_{30}\cdots b^y_{24}b^y_{23})_2 - 8'b01111111, \\ Y_{ma} &= (b^y_{22}\cdots b^y_1 b^y_0)_2, \\ Y_{fi}[30] &= b^y_{31}, \\ Y_{fi}[29:0] &= 1.\underbrace{(Y_{ma})_2} \gg \underbrace{(Y_{ex})_{10}}. \end{aligned} \tag{12}$$

From a hardware perspective, the architecture of this module is shown in Fig. 6. "127" is the offset of the exponent of a single-precision floating-point number. The operations involved are only addition, subtraction and shift.

C. SM-LUT Module

This module has dual operation function. In addition to the input x, it has the signal c to select the function. When $c = 0$, this module is used to calculate $log_2 x$ $(x \,\epsilon\, [1, 2))$; On the other hand, when $c = 1$, the module functions as 2^x $(x \,\epsilon\, [0, 1))$.

If the SM-LUT module implements $log_2 x$, its hardware architecture is shown in Fig. 7. In total, the architecture mainly consists of a multi-region address searcher and two LUTs. As described above, the data (D_1) stored in LUT_1 are all judgment nodes and LUT_2 stores all results (D_2).

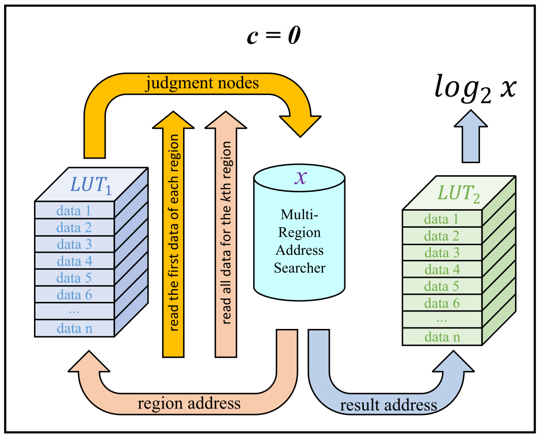


Fig. 7. Architecture of computing $log_2 x$ by SM-LUT module.

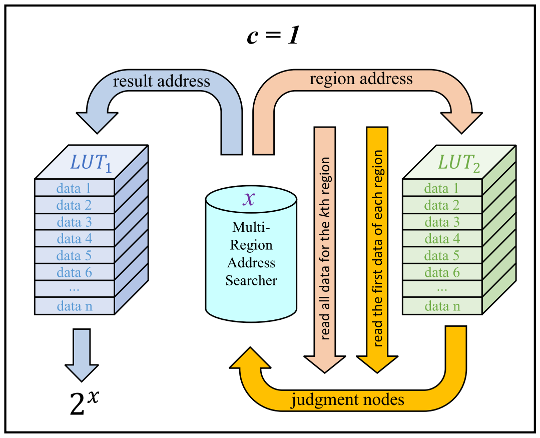


Fig. 8. Architecture of computing 2^x by SM-LUT module.

The calculation formulas for every LUT_1 data and LUT_2 data are as follows:

$$D_2(i) = \frac{i}{2^b}, \quad D_1(i) = 2^{D_2(i)} - 1, \ b = 1, 2, 3, \cdots. \quad (13)$$

Taking $b = 10$ as an example, the maximum absolute error of the result is $\frac{1}{2^{10}} = 9.76 \times 10^{-4}$.

$$LUT_1 : 0, \sqrt[1024]{2} - 1, \sqrt[1024]{2^2} - 1, \cdots, \sqrt[1024]{2^{1023}} - 1,$$
$$LUT_2 : 0, \frac{1}{1024}, \frac{2}{1024}, \frac{3}{1024}, \cdots, \frac{1023}{1024}.$$

Assuming that the multi-region address searcher uses 32 regions for parallel search, then the data in LUT_1 correspond to the following regions:

$$RG_1 : 0, \sqrt[1024]{2}, \sqrt[1024]{2^2}, \sqrt[1024]{2^3}, \cdots, \sqrt[1024]{2^{31}},$$
$$\vdots$$
$$RG_\hbar : \sqrt[1024]{2^{31\hbar-32}}, \sqrt[1024]{2^{31\hbar-31}}, \cdots, \sqrt[1024]{2^{31\hbar-1}},$$
$$\vdots$$
$$RG_{32} : \sqrt[1024]{2^{992}}, \sqrt[1024]{2^{993}}, \cdots, \sqrt[1024]{2^{1023}}.$$

When the input data x is valid, the whole computation process is shown in Section III-C. If the SM-LUT module is used to compute 2^x, its hardware architecture is transformed as shown in Fig. 8. The calculation process is the same as above, except that the mapping lookup tables are swapped.

From Fig. 7 and Fig. 8, we know that the core component of the SM-LUT module is multi-region address searcher. As discussed in Section III-C, we have decided to use a multi-region parallel search method. Fig. 9(a) shows the architecture of multi-region address searcher using 32 comparators. To further improve the performance, we adopt an improved architecture shown in Fig. 9(b), which replaces the comparator with the subtracter. The difference between the input x and each judgment node has one sign bit, which is marked as $sign\,1$, $sign\,2$, $\cdots$, and $sign\,32$. We can use the 32 sign bits to find the region address or the result address. The 32 mapping relationships are shown in Table IV below. In the hardware implementation, the multi-region address searcher is operated twice with different inputs. The first time is to find the region address using region addresses as inputs, and the second time the result address using the region address obtained from the first search as inputs.

TABLE IV
ADDRESS INDEX MAPPING RELATIONSHIP

$SIGN$	Region address	Result address
$32'b0111\cdots111$	$RG_1 \rightarrow [0, 31]$	RA_1
$32'b0011\cdots111$	$RG_2 \rightarrow [32, 63]$	RA_2
$32'b0001\cdots111$	$RG_3 \rightarrow [64, 95]$	RA_3
$\cdots$	$\cdots$	$\cdots$
$32'b000\cdot0\cdot111$	$RG_\hbar \rightarrow [32(\hbar-1), 32\hbar-1]$	$RA_\hbar$
$\cdots$	$\cdots$	$\cdots$
$32'b0000\cdots011$	$RG_{30} \rightarrow [928, 959]$	RA_{30}
$32'b0000\cdots001$	$RG_{31} \rightarrow [960, 991]$	RA_{31}
$32'b0000\cdots000$	$RG_{32} \rightarrow [992, 1023]$	RA_{32}

$SIGN = \{sign\,1, sign\,2, \cdots, sign\,32\}$.
$RG_\hbar$: The $\hbar^{th}$ region.
$RA_\hbar$: The $\hbar^{th}$ address in the $\hbar^{th}$ region.

From the above architecture, we only need 2 iterations (clock cycles) to search the final result from 1024 (32 × 32) numbers, and the index address can be reduced from the original 10 bits ($1024 = 2^{10}$) to 5 bits ($32 = 2^5$). In this way, the critical path is shorter than the serial search method and the area is lower than the method in Fig. 9(a).

D. FPM Module

The input of this module is two 38-bit fixed-point numbers, and the output result is obtained by a fixed-point multiplier. Then it intercepts the middle 38-bits as the result of this module. The choice of multiplier has an important influence on the timing, area and precision of the whole architecture. In general, we use the conventional array multiplier (CAM) [17] for fixed-point multiplication, but we all know that its critical path is long and it usually affects the frequency of whole architecture. Another common multiplier is called Vedic, although both are composed of ripple carry adders, the critical path of Vedic is shorter than that of CAM. For this reason, we select a 38-bit fixed-point Vedic multiplier (VM) to conduct the multiplication operation ($Y \times (K_1 + log_2(1.M_1))$) in (3). The architecture of VM can be referred to [16], [18].

To illustrate intuitively the cost of such a substitution, we compare their area in terms of the number of transistors. From [19], we know that one B-bit ripple carry

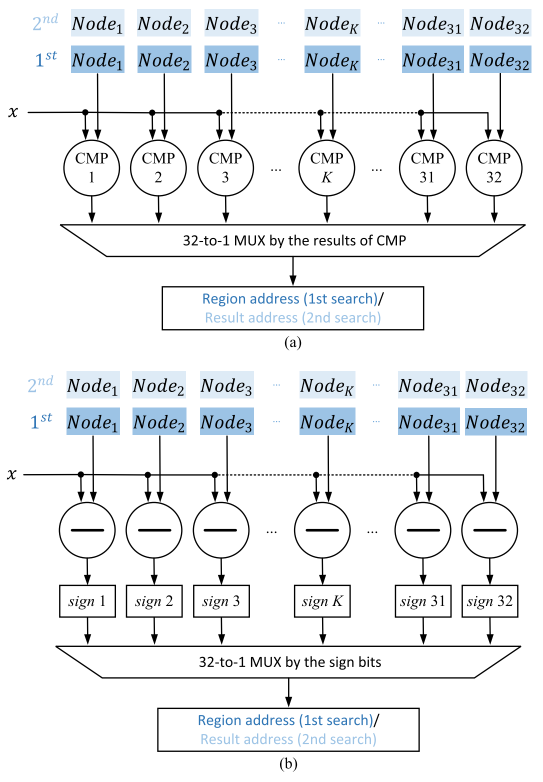

2nd Node1 Node2 Node3 NodeK Node31 Node32
1st Node1 Node2 Node3 NodeK Node31 Node32
x
CMP 1
CMP 2
CMP 3
CMP K
CMP 31
CMP 32
32-to-1 MUX by the results of CMP
Region address (1st search)/
Result address (2nd search)
(a)
2nd Node1 Node2 Node3 NodeK Node31 Node32
1st Node1 Node2 Node3 NodeK Node31 Node32
x
sign 1
sign 2
sign 3
sign K
sign 31
sign 32
32-to-1 MUX by the sign bits
Region address (1st search)/
Result address (2nd search)
(b)

Fig. 9. (a) Architecture of parallel search using comparator. (b) Architecture of parallel search using subtracter.

adder (RCA) needs $24B$ transistors and one B-bit CAM requires $6B \times (5B - 6)$ transistors. Therefore, the transistor count (TC) required for a 38-bit CAM is:

$$TC_{CAM} = 6 \times 38 \times (5 \times 38 - 6) = 41952. \tag{14}$$

According to the structure of Vedic multiplier (VM) [16], a 38-bit (2'b0 is added to the highest bit) VM consists of 4 20-bit VMs and 2 40-bit RCAs. A 20-bit VM consists of 4 10-bit VMs and 2 20-bit RCAs. A 10-bit VM consists of 4 5-bit VMs and 2 10-bit RCAs. A 5-bit VM consists of 10-bit full adders. Therefore, the TC of a 5-bit VM equals that of a 10-bit RCA. The TC of a 20-bit RCA equals that of 2 10-bit RCAs, and the TC of a 40-bit RCA equals that of 4 10-bit RCAs. In total, the number of 10-bit RCA required for a 38-bit VM is:

$$4(4(4+2) + 2 \times 2) + 2 \times 4 = 120. \tag{15}$$

Thus, the TC required for a 38-bit VM is:

$$TC_{VM} = 24 \times 10 \times 120 = 28800. \tag{16}$$

From (14) and (16), we know that the TC of 38-bit VM is lower than (save 31.35%) that of 38-bit CAM. Besides, VM has shorter critical path, so we choose VM as the operation unit of FPM.

In fact, we can do further optimization in the specific hardware implementation process. From the subsection VI-A, we know that the first input of VM is valid with 30 bits (Y_{fi}), and the second input is valid with 35 bits ($[K_1 + log_2(1.M_1)]$). In this case, some operations are not required, as shown in Fig. 10. The colors of the circles represent different input bit widths for each layer. The numbers on the left ("M_l") and right ("M_r") sides of the multiplier symbol represent the number of multiplicative factor. After multiplication ("$M_l \times M_r$"), we can get the number of multipliers required. The results of these multipliers in each layer need to be added by using adders to obtain the multiplication results of the previous layer. It requires half as many adders as multipliers. The numbers in the adder symbol represent the number of adders. The "X" in the last row means that the multiplicative factors in the fourth layer do not participate in the calculation since they are all zeros (namely zero-skipping), and the numbers in the circle represent the number of multiplicative factors that participate in the operation after optimization.

In principle, a 38-bit VM requires 64 (8×8) 5-bit VMs and 32 (8×4) 10-bit RCAs. Since the first 10 bits in Mul_1 and the first 5 bits in Mul_2 are 0, the multiplication of these parts is 0. Therefore, we can omit their operations. By zero-skipping for multiplication, a 38-bit VM only needs 42 (6×7) 5-bit VMs and 21 (3×7) 10-bit RCAs. Compared to the previous calculation, it saves 34.375% 5-bit VMs and 10-bit RCAs.

V. Experimental Setup

In order to compare our approach with other approaches under the same conditions, we first set the same input range. Based on this condition, we select a similar error range through MATLAB simulation. All these form a solid basis for the further comparative analysis in Section VI.

In MATLAB, we verify the correctness of the proposed methodology by evaluating its relative errors. The relative error (RE) is defined as

$$RE = \frac{|V_T - V_M|}{|V_T|}, \tag{17}$$

where V_T represents the theoretical value of X^Y and V_M stands for the measured value of X^Y using our approach. Therefore, the maximum relative error of X^Y is $max(RE_k)$ of k measurements. Another important criteria is the average relative error (ARE), which defined as follows:

$$ARE = \frac{\sum_{i=1}^{k} |V_T - V_M|_i}{|V_T| \times k}. \tag{18}$$

Next, we set the input range to compare. Currently, the state-of-the-art approaches to compute X^Y-like function are mainly based on high-radix iterative and CORDIC, such as the work in [2], [12]–[14]. Their input range depends on the parameter r, m, r refers to the radix [2] and m refers to the negative iterations of CORDIC [12]–[14] respectively. For $\sqrt[N]{X}$ computation, the CORDIC-based approaches [12]–[14] implement it by software and hardware for $X \epsilon [10^{-6}, 10^6]$ and $N \epsilon [2, 1002]$. For X^N computation, the input range of high-radix iterative-based approach [2] and CORDIC-based approach [13] is set to be $X \epsilon [10^{-2}, 10^2]$ and $N \epsilon [1, 5]$. To be fair, we also set the same input range for testing and the test quantity k is 40,000. Need of special note is that unlike other dedicated architectures of computing Nth root, the input of our architecture may not be an integer. For example, to compute a quadratic root, the input of our architecture is $N = 0.5$, but that for the dedicated architecture is $N = 2$.

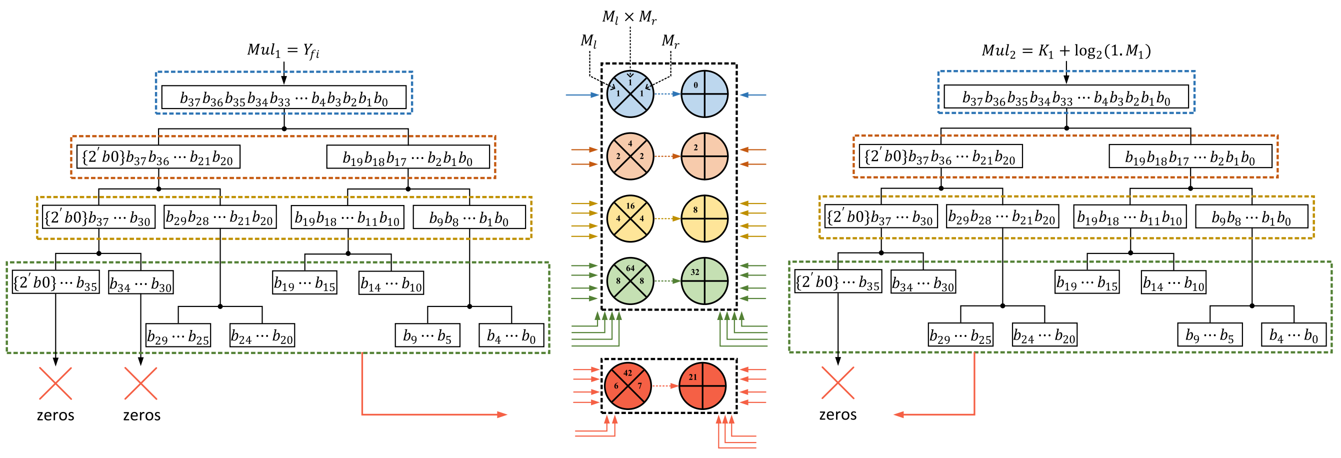


Fig. 10. Operation process and zero-skipping optimization of a 38-bit Vedic multiplier.

TABLE V
VERIFICATION OF THE PROPOSED METHODOLOGY BY MATLAB SIMULATION

Item	$\sqrt[N]{X}$				X^N		
	[12]	[13]	[14]	Proposed	[2]	[13]	Proposed
X	$[10^{-6}, 10^6]$	$[10^{-6}, 10^6]$	$[10^{-6}, 10^6]$	$[10^{-6}, 10^6]$	$[10^{-2}, 10^2]$	$[10^{-2}, 10^2]$	$[10^{-2}, 10^2]$
N	$[2, 1002]$	$[2, 1002]$	$[2, 1002]$	$[2, 1002]$	$[1, 5]$	$[1, 5]$	$[1, 5]$
$max(RE)$	1.993×10^{-3}	1.928×10^{-3}	7.373×10^{-3}	1.069×10^{-3}	2.321×10^{-2}	1.030×10^{-2}	5.272×10^{-3}
ARE	5.195×10^{-4}	5.464×10^{-4}	3.171×10^{-3}	4.160×10^{-4}	1.340×10^{-4}	2.875×10^{-3}	2.095×10^{-3}

Then we conduct a software simulation. Before we do that, we need to set the assumptions. In the traditional approaches, the precision is closely related to the positive iterations of CORDIC or high-radix iterative, while our approach is related to the size of SM-LUT. For a more impartial comparison of the hardware overhead below, we select an average relative error of the order of magnitude of 10^{-4}. To achieve the precision, our approach needs to store 1024 pieces of result data in either LUT_1 or LUT_2. Paper [12] needs to set radix r to be 128, Papers [12]–[14] need to set the number of iterations n to be 10, 10, 11, respectively. From Table V, in terms of $max(RE)$ and ARE, it is evident that our approach for $\sqrt[N]{X}$ computation is superior compared with the state-of-the-art approaches [12]–[14] and the proposed approach for X^N computation is also superior compared with the state-of-the-art approaches [2], [13]. Although the average relative error of [2] is smaller, its maximum relative error is relatively larger. In practical applications, the maximum relative error is more influential and significant.

In addition, we make an error distribution analysis. According to Fig. 11, the error of our approach in calculating $\sqrt[N]{X}$ is mostly concentrated in $10^{-3} > RE \geq 10^{-4}$, which accounts for 93.035% through the calculation. The errors of [12] and [13] are also concentrated in $10^{-3} > RE \geq 10^{-4}$, while the errors of [14] are mainly concentrated in $RE \geq 10^{-3}$, accounting for 83.465%, 77.263% and 84.143% respectively. For X^N computation, the errors of [13] and proposed approach are concentrated in $10^{-2} > RE \geq 10^{-3}$, which makes up 78.020% and 90.245% respectively. Although the errors of [2] are mainly concentrated in $10^{-3} > RE \geq 10^{-4}$, but the probability is only 43.595% and its errors are scattered from 10^{-2} to 10^{-5}. In terms of computational stability, the more concentrated the error probability is, the more stable it is. Therefore, from Fig. 11, we can conclude that the computation of X^Y-like functions by our approach is more stable.

VI. EVALUATION AND DISCUSSION

In order to reveal the superiority of our approach based on the above similar conditions, we make a comparative analysis from different aspects with other latest approaches to compute X^Y-like functions, including word length setting, hardware complexity (transistor count) and computation latency. Finally, we compare the hardware implementation results of different approaches under the same CMOS technology, so as to directly reflect the advantages of our approach.

A. Word Length Analysis

In this subsection, we analyze the word length required for each approach's hardware implementation based on the conditions of Section V, which is of critical importance. The longer the word length is, the better the precision is. However, overlong word length will consume more area and power. Sometimes, it also increases the critical path and lowers the frequency of the whole architecture. From the perspective of data composition, the fractional (or mantissa) part affects the precision of computation. According to a "rule of thumb" mentioned in [20], if we hope that the output has n-bit precision, the internal registers should have $log_2 n$ additional bits at the least significant bit (LSB) position. Therefore, since the input and output of our design are 32-bit floating-point numbers with 23-bit mantissa, we should expand the fractional part of all internal data to 27 bits with 4 ($\approx log_2 23$) additional bits during the computation process.

As for integer part, the FFPC module needs to output the fixed-point number Y_{fi}, the exponent X_{ex} and the

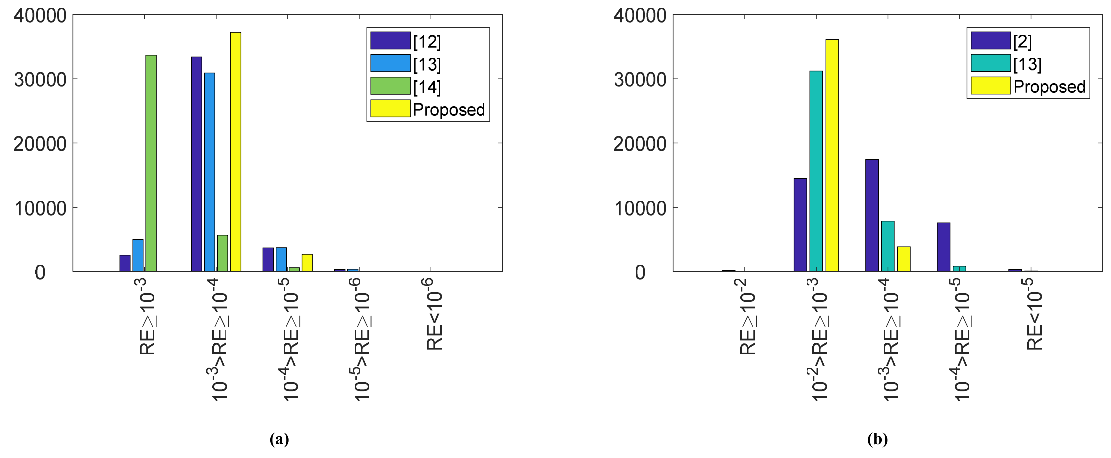


Fig. 11. Error distributions under different approaches. (a) For $\sqrt[N]{X}$ Computation. (b) For X^N Computation.

mantissa X_{ma}. $X_{ex} \epsilon [-127, 127]$ and $X_{ma} \epsilon [0, 1]$, so X_{ex} needs an integer part of 7 bits and X_{ma} doesn't. Because our design supports the $\sqrt[N]{X}$ computation of $N \epsilon [2, 1002]$ (Y_{fi} belongs to (0.001, 0.5]) and the X^N computation of $N \epsilon [1, 5]$ (Y_{fi} belongs to [1, 5]), the integer part of Y_{fi} needs at least 3 ($\approx log_2 5$) bits to support the maximum value. From (3), we know that the inputs of FPM are Y_{fi} and $K_1 + log_2(1.M_1)$, the result of $log_2(1.M_1)$ belongs to [0, 1), $K_1 \epsilon [-127, 127]$, so $[K_1 + log_2(1.M_1)] \epsilon (-126, 128)$ and $Y_{fi} \times [K_1 + log_2(1.M_1)] \epsilon (-630, 640)$. If so, we uniformly set the input and output of FPM to 38 bits, consisting of a 1-bit sign bit, a 10-bit integer bit, and a 27-bit fractional bit. Because the input and output of SM-LUT are between [0, 1), there is no need for integer bits.

To sum up, the word length settings of each module in our design are shown in Table VI. The other approaches' word lengths can be found in their respective work and are summarized in Table VI. By contrast, under the same input range, our approach can set shorter word lengths than the state-of-the-art approaches, which not only saves the area, but also increases the working frequency.

B. Transistor Count Analysis

In this subsection, we make an analysis on the performance of the proposed architecture and compare it with the state-of-the-art architecture [2], [12]–[14] in terms of the hardware complexity. From [13], we know that the total TC involved in the natural logarithm computation by HV CORDIC is

$$TC_{log\_e} = (8 \times (m + 1) + 6 \times n + 2) \times TC_{RCA}, \quad (19)$$

where m is the negative iterations and n is the positive iterations of CORDIC, TC_{RCA} represents the TC of RCAs and one B-bit RCA needs $24B$ transistors. In fact, Binary Hyperbolic CORDIC (BV mode and BR mode) [13] is the same as Generalized Hyperbolic CORDIC (GHV mode and GHR mode) [14] with base 2, the total TC required for the binary logarithm computation by them can be expressed as follows:

$$TC_{log\_2} = (6 \times n + 2) \times TC_{RCA} + TC_{RCA}. \quad (20)$$

Then, the total TC involved in the division computation by LV CORDIC is expressed as follows:

$$TC_{div} = 2 \times (m + n + 1) \times TC_{RCA}. \quad (21)$$

For the natural exponential computation, the total TC can be calculated by the following equation:

$$TC_{exp\_e} = (8 \times (m + 1) + 6 \times n + 1) \times TC_{RCA}. \quad (22)$$

In the same way, we can get the total TC involved in the binary exponential computation is given by:

$$TC_{exp\_2} = (6 \times n + 1) \times TC_{RCA}. \quad (23)$$

In addition to the above modules, we know from paper [14] that two 28-bit multipliers are used to calculate $\sqrt[N]{X}$ and the multiplier uses a two-stage pipelined structure. As described in Section IV-D, a 28-bit VM consists of 4 14-bit VMs and 2 28-bit RCAs. A 14-bit VM consists of 4 7-bit VMs and 2 14-bit RCAs. A 7-bit VM consists of 14-bit full adders. Therefore, the TC of a 7-bit VM equals that of a 14-bit RCA. The TC of a 28-bit RCA equals that of 2 14-bit RCAs. Thus, the total TC required for a 28-bit VM is:

$$TC_{mul_{28bit}} = 24 \times 14 \times [(4(4 + 2) + 2 \times 2)] = 9408. \quad (24)$$

Three extra adders are also included in the architecture of [14].

In contrast, it can be seen from Fig. 6-Fig. 9 that FFPC module involves 3 add operations and a shift. The input and output of SM-LUT module do not require additional add operation, and its internal consists of 32 add operations (two iterations). The SM-LUT module is used twice to compute $log_2 x$ and 2^x respectively. From (3), we need an extra add operation. To output the correct floating-point result, we also need an adder to calculate $K_o + 127$. In conclusion, the total TC involved in the whole design is:

$$\begin{aligned} TC_{pro} &= TC_{FFPC} + 2TC_{SM-LUT} + TC_{FPM} + 2TC_{RCA} \\ &= 24 \times 8 \times 2 + 24 \times 24 + 2 \times 2 \times 32 \times 24 \times 28 \\ &\quad + 28800 + 24 \times 8 \times 2 = 116160. \end{aligned} \quad (25)$$

The total TC of other state-of-the-art approaches can be calculated as shown in Table VII from the parameters derived in the previous two subsections. NTS refers to the number

TABLE VI
WORD LENGTH SETTINGS FOR DIFFERENT APPROACHES

Function	Approach	Type	Logarithm	Division / Multiplication	Exponential
$\sqrt[N]{X}$	[12]	Module	HV CORDIC	LV CORDIC	HR CORDIC
		S+I+F[1]	1+20+27	1+10+27	1+11+27
		Total Bits	48	38	39
	[13]	Module	BV CORDIC	LV CORDIC	BR CORDIC
		S+I+F	1+2+45	1+10+27	1+2+27
		Total Bits	48	38	30
	[14]	Module	GHV CORDIC	LV CORDIC and Multiplier	GHR CORDIC
		S+I+F	1+2+27	1+8+27 and 0+1+27	1+2+27
		Total Bits	30	36 and 28	30
X^N	[2]	Module	HV CORDIC	Multiplier	HR CORDIC
		S+I+F	1+7+27	1+5+27	1+34+27
		Total Bits	35	33	62
	[13]	Module	BV CORDIC	Multiplier	BR CORDIC
		S+I+F	1+2+32	1+5+27	1+2+27
		Total Bits	35	33	30
$\sqrt[N]{X}$ and X^N	Proposed	Module	SM-LUT	Multiplier	SM-LUT
		S+I+F	1+0+27	1+10+27	1+0+27
		Total Bits	28	38	28

[1] "S+I+F" stands for "sign bit + integer bit + fractional bit".

TABLE VII
TRANSISTOR COUNT OF IMPORTANT MODULES UNDER DIFFERENT ARCHITECTURES

Function	Approach	Total TC	NTS	P_{NTS}
$\sqrt[N]{X}$	[12]	214872	98712	45.94%
	[13]	151080	34920	23.11%
	[14]	150432	34272	22.78%
X^N	[2]	414906	298746	72.00%
	[13]	137634	21474	15.60%
$\sqrt[N]{X}$ and X^N	Proposed	116160	\	\

of transistor savings compared with other approaches and it is defined as:

$$NTS = TC_{oth} - TC_{pro}. \tag{26}$$

And compared with the state-of-the-art approaches, the percentage of NTS is:

$$P_{NTS} = \frac{NTS}{TC_{oth}} = \frac{TC_{oth} - TC_{pro}}{TC_{oth}} \times 100\%. \tag{27}$$

From Table VII, we can draw the following conclusions:

- 1) For $\sqrt[N]{X}$ computation, our approach saves 45.94%, 23.11% and 22.78% transistors compared with [12], [13] and [14], respectively.
- 2) For X^N computation, our approach saves 72.00% and 15.60% transistors compared with [2] and [13].

C. Timing Analysis

In this section, we analyze the computation latency of each approach to calculate $\sqrt[N]{X}$ or X^N. In the case that the working frequency of the whole architecture is consistent, the smaller the computation latency, the faster the computation speed. The timing of the analysis below is based on the conditions set in previous subsections: input range, precision, and word length.

From [13], we know that [2] needs 59 clock cycles to compute X^N, which will not be analyzed further here. And beyond that, we also know the total latency (TL) of [12] is given by

$$TL_{[12]} = 5m + 3n + 2[\frac{n-1}{3}] + 7. \tag{28}$$

Paper [14] shows that the TL of his architecture is expressed as follows:

$$TL_{[14]} = 2 \times TL_{CORDIC} + TL_{mul} + 2 + 2, \tag{29}$$

where $TL_{CORDIC} = n + [\frac{n-1}{3}]$ and $TL_{mul} = 2$ when the multiplier uses a two-stage pipelined structure, that is

$$TL_{[14]} = 2n + 2[\frac{n-1}{3}] + 6. \tag{30}$$

Mopuri and Acharyya [13] analyzes the number of clock cycles consumed by his proposed architecture in each module. In general, when calculating $\sqrt[N]{X}$, its TL is

$$TL_{[13]\_Nroot} = TL_{log\_2} + TL_{div} + TL_{exp\_2}, \tag{31}$$

where $TL_{log\_2} = n + 4$, $TL_{div} = m + n + 1$, $TL_{exp\_2} = n + 2$, when $n = 4, 13, 40, \cdots$, the binary hyperbolic CORDIC should be repeated and additional one clock cycle is required. Therefore, including the total extra latency TL_{extra}, $TL_{[13]\_Nroot}$ will be

$$TL_{[13]\_Nroot} = m + 3n + 7 + TL_{extra}. \tag{32}$$

When calculating X^N, the TL consumed by approach [13] is

$$TL_{[13]\_Npower} = TL_{log\_2} + TL_{mul} + TL_{exp\_2}, \tag{33}$$

where $TL_{mul} = 1$ when the multiplication operation is performed using CAM, that is

$$TL_{[13]\_Npower} = 2n + 7 + TL_{extra}, \tag{34}$$

Next, we focus on analyzing the computation latency of our approach. In theory, FFPC module needs only one clock

TABLE VIII
TOTAL COMPUTATION LATENCY FOR DIFFERENT APPROACHES

Function	Approach	Total TL	NTL	P_{NTL}
$\sqrt[N]{X}$	[12]	53	38	71.70%
	[13]	41	26	63.41%
	[14]	34	19	55.88%
X^N	[2]	59	44	74.58%
	[13]	29	14	48.28%
$\sqrt[N]{X}$ and X^N	Proposed	15	\	\

cycle to complete its task, but in order to shorten the critical path, we add an intermediate register to increase the working frequency, which makes the module need two clock cycles. In the SM-LUT module, one clock cycle is needed to select the judgment node and extract the result data, respectively. The multi-region address searcher adopts a 32-way parallel structure, and a total of two clock cycles are needed to find the index address of the result data. Therefore, the SM-LUT module needs four clock cycles totally. The FPM module uses 3-level segmentation operation, which needs three clock cycles. In addition, Eq. 3 also needs one clock cycle to calculate $K_1 + log_2(1.M_1)$, and the final floating-point result also needs one clock cycle to complete the 32-bit splicing. Therefore, the TL of our architecture is

$$\begin{aligned} TL_{pro} &= TL_{FFPC} + 2 * TL_{SM-LUT} + TL_{FPM} + 2 \\ &= 2 + 2 * 4 + 3 + 2 = 15. \end{aligned} \tag{35}$$

As analyzed in Section V, in order to achieve the calculation accuracy in Table V, the iterations (n) of CORDIC in [12]–[14] should be set to 10, 10, 11, respectively. m should be 2. In this case, we can calculate the TL for the other state-of-the-art approaches, as shown in Table VIII. NTL refers to the TL savings compared with other approaches and it is defined as:

$$NTL = TL_{oth} - TL_{pro}. \tag{36}$$

And compared with the state-of-the-art approaches [2], [12]–[14], the percentage of NTL is:

$$P_{NTL} = \frac{NTL}{TL_{oth}} = \frac{TL_{oth} - TL_{pro}}{TL_{oth}} \times 100\%. \tag{37}$$

From Table VIII, we can draw the following conclusions:

- 1) For $\sqrt[N]{X}$ computation, our approach saves 71.70%, 63.41% and 55.88% computation latency compared with [12], [13] and [14], respectively.
- 2) For X^N computation, our approach saves 74.58% and 48.28% computation latency compared with [2] and [13].

D. Implementation Results

Based on the above analysis, we code our architecture and the state-of-the-art architectures [2], [12]–[14] in Verilog HDL. Their ASIC implementations are all done under TSMC 40nm CMOS technology and clock frequency @1GHz with the help of Synopsis Design Compiler (DC). Their synthesis reports are shown in Table IX. The input range and expected accuracy are not necessarily limited to the above analysis, but for a fair comparison, we set the above conditions to further analyze their hardware costs, including area, power consumption, etc. The LUTs are excluded from our area report since they shall be generated by a memory compiler not synthesis. The maximum operating frequency (MOF) of our design can reach 2.2GHz. The number of logic gates is 12448 and that of FFs is 741.

To visually illustrate the advantages and disadvantages of the architecture, we also use the following metrics. The first one is computing time (CT), which is defined as follows:

$$CT = \frac{TL}{f}, \tag{38}$$

where TL refers to the total latency of the architecture, f stands for the working frequency. The second one is the floating-point operations per second per watt ($FOSW$) or the sampling rate per watt $SRPW$. Both are used to assess energy efficiency and defined as follows:

$$\begin{aligned} FOSW &= \frac{(10^3 \times f)Mflops}{(w)mW}, \\ SRPW &= \frac{(10^3 \times f)MSPS}{(w)mW}, \end{aligned} \tag{39}$$

where w is the power consumption of the architecture and $MSPS$ refers to “Million Samples Per Second”. Typically, $FOSW$ is used for floating-point input types [14], and $SRPW$ is used for other input types, such as fixed-point and integer data types [12], [13]. In addition, we compute the area savings (AS) and power consumption savings (PCS) compared with other approaches by the following definitions:

$$\begin{aligned} AS &= Area_{oth} - Area_{pro}, \\ PCS &= Power_{oth} - Power_{pro}. \end{aligned} \tag{40}$$

Compared with the state-of-the-art approaches, the percentage of AS or PCS is:

$$\begin{aligned} P_{AS} &= \frac{AS}{AS_{oth}} = \frac{AS_{oth} - AS_{pro}}{AS_{oth}} \times 100\%, \\ P_{PCS} &= \frac{PCS}{PCS_{oth}} = \frac{PCS_{oth} - PCS_{pro}}{PCS_{oth}} \times 100\%. \end{aligned} \tag{41}$$

From Table IX, the proposed design for $\sqrt[N]{R}$ computation saves 84.30%, 77.93% and 73.48% area and 84.32%, 79.56% and 77.37% power consumption respectively when it is compared with the state-of-the-art architectures [12]–[14]. For R^N computation, our design can save 90.57% and 75.21% area and 90.39% and 73.39% power consumption respectively compared with the state-of-the-art architectures [2], [13]. When the working frequency is 1GHz, we can get the computation time as shown in Table IX by combining the computation latency shown in Table VIII. After calculation, we know that our architecture saves 71.70%, 63.41% and 55.88% computation time respectively compared with [12]–[14] for $\sqrt[N]{X}$ computation. For X^N computation, our architecture saves 74.58% and 48.28% computation time compared with [2] and [13].

Then, we use $FOSW$ and $SRPW$ to evaluate the energy efficiency of different architectures. From the data in Table IX, our architecture is one order of magnitude more efficient than others. In terms of the precision of actual hardware architecture, we also make a statistical analysis. As with 40,000 data

TABLE IX
ASIC IMPLEMENTATION FOR THE PROPOSED AND STATE-OF-THE-ART ARCHITECTURES

Item	$\sqrt[N]{X}$				X^N		
	[12]	[13]	[14]	Proposed	[2]	[13]	Proposed
Input Type	X:fixed-point N:fixed-point	X:floating-point N:fixed-point	X:floating-point N:floating-point	X:floating-point N:floating-point	X:fixed-point N:fixed-point	X:floating-point N:fixed-point	X:floating-point N:floating-point
Area (μm^2)	91318.06	64968.98	54060.72	14338.50	152068.73	57850.59	14338.50
AS (μm^2)	76979.56	50630.48	39722.22	\	137730.23	43512.09	\
P_{AS}	84.30%	77.93%	73.48%	\	90.57%	75.21%	\
PC (mW)	29.27	22.46	20.28	4.59	47.76	17.25	4.59
PCS (mW)	24.68	17.87	15.69	\	43.17	12.66	\
P_{PCS}	84.32%	79.56%	77.37%	\	90.39%	73.39%	\
CT	53 ns @1GHz	41 ns @1GHz	34 ns @1GHz	15 ns @1GHz	59 ns @1GHz	29 ns @1GHz	15 ns @1GHz
Energy Efficiency	SRPW= 3.42×10^4	SRPW= 4.45×10^4	FOSW= 4.93×10^4	FOSW= 2.18×10^5	SRPW= 2.09×10^4	SRPW= 5.80×10^4	FOSW= 2.18×10^5
MOF (GHz)	2.08	1.98	2.38	2.22	1.63	1.63	2.22
$max(RE)$	5.22×10^{-3}	5.14×10^{-3}	9.06×10^{-3}	4.57×10^{-3}	6.39×10^{-2}	4.34×10^{-2}	8.16×10^{-3}
ARE	7.17×10^{-4}	6.97×10^{-4}	5.45×10^{-3}	6.83×10^{-4}	9.56×10^{-4}	6.37×10^{-3}	4.48×10^{-3}

TABLE X
COMPARISON OF PERFORMANCE INDICATORS USING FPGA

Design / Item	Paper [15]	Paper [13]	Proposed
FPGA Platform	Virtex-5	Virtex-6	Virtex-6
LUT	1828	12416	1547
FF	1260	314	755
DSP 48 Sclics	11	3	0
BRAMs	7	0	1
MOF (MHz)	214	380	405

from software tests, we find the maximum relative error and calculate the average relative error, as shown in Table IX. Although their errors are larger than the results of the software tests, they are of the same magnitude. Based on the maximum relative error, we can evaluate the actual effective bits of the hardware implementation. Take our architecture of computing X^N for example, the maximum RE is $8.16 \times 10^{-3} \approx 2^{-7}$, considering that the output is 32 bits and the fractional part is 23 bits, so 16 bits in the output are approximately accurate.

In addition, we also use FPGA to implement the architecture proposed in this paper, and its resource usage is shown in Table X. The two LUTs have a size of 10K bits each and 20K bits in total. They are configured by one BRAM block which has a size of 36K bits with an utilization of 55.56%. Overall, our architecture has great advantages compared with [13] and [15].

As a note, we would like to point out that the accuracy in [13] is actually evaluated according to relative error, while the whole paper describes it as absolute error, which is different. The most obvious one is formula (46), which is completely different in definition and expression. Secondly, Table XI in [13] says that the hardware implementation is based on TSMC 45nm, but the abstract and conclusion mention that it is evaluated under TSMC 40nm, which is also different. After verification in this paper, the hardware implementation technology is close to TSMC 40nm rather than TSMC 45nm.

VII. CONCLUSION

A new method and hardware architecture are proposed to compute arbitrary X^Y-like functions based on SM-LUT in this paper. Most importantly, SM-LUT is first proposed for simultaneous computation of log_2x ($x \in [1, 2]$) and 2^x ($x \in [0, 1]$). It plays a central role in computing X^Y. In the hardware implementation, we adopt various optimization methods to improve the hardware efficiency of the proposed architecture. In addition, other advanced approaches have also been analyzed with the same input range and similar precision to prove the superiority of our approach. Synthesized under the TSMC 40nm CMOS technology @1GHz frequency, our architecture saves 78.57% area and 80.42% power consumption on average when calculating $\sqrt[N]{R}$. For R^N computation, our architecture saves 82.89% area and 81.89% power consumption averagely. In fact, the most important advantage of our architecture is computing latency. Because the CORDIC-based hardware architecture requires multiple iterations to achieve the desired accuracy, sometimes it also needs to expand the input range by adding the negative iterations (such as [12]), which inevitably further increases the hardware complexity and computing latency. Therefore, the hardware architecture based on SM-LUT proposed in this paper can effectively solve the latency problem. What's more, our architecture is not only low hardware cost, but also high energy efficiency, which can be one order of magnitude higher than other approaches.

REFERENCES

[1] D. Harris, "A powering unit for an OpenGL lighting engine," in *Proc. Conf. Rec. 35th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2001, pp. 1641–1645.

[2] J.-A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, "High-radix iterative algorithm for powering computation," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Jun. 2003, pp. 204–211.

[3] F. Feng, L. Li, K. Wang, Y. Fu, G. He, and H. Pan, "Design and application space exploration of a domain-specific accelerator system," *Electronics*, vol. 7, no. 4, p. 45, Mar. 2018.

[4] B. Perach and S. Weiss, "SiMT-DSP: A massively multithreaded DSP architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 8, pp. 1413–1426, Aug. 2018.

[5] J. M. Muller, "Elementary functions: Algorithms and implementation," *Math. Comput. Educ.*, vol. 34, no. 1, pp. 21–52, 1997.

[6] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. Comput.*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[7] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7/sup TM/ microprocessor," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Apr. 1999, pp. 106–115.

[8] J.-A. Pineiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Trans. Comput.*, vol. 51, no. 12, pp. 1377–1388, Dec. 2002.

[9] J.-A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, "Algorithm and architecture for logarithm, exponential, and powering computation," *IEEE Trans. Comput.*, vol. 53, no. 9, pp. 1085–1096, Sep. 2004.

[10] E. Antelo, T. Lang, and J. D. Bruguera, "Very-high radix CORDIC vectoring with scalings and selection by rounding," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Apr. 1999, pp. 204–213.

[11] A. Vazquez and J. D. Bruguera, "Iterative algorithm and architecture for exponential, logarithm, powering, and root extraction," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1721–1731, Sep. 2013.

[12] Y. Luo, Y. Wang, H. Sun, Y. Zha, Z. Wang, and H. Pan, "CORDIC-based architecture for computing Nth root and its implementation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 12, pp. 4183–4195, Dec. 2018.

[13] S. Mopuri and A. Acharyya, "Low complexity generic VLSI architecture design methodology for N^{th} root and N^{th} power computations," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 12, pp. 4673–4686, Dec. 2019.

[14] Y. Wang, Y. Luo, Z. Wang, Q. Shen, and H. Pan, "GH CORDIC-based architecture for computing N^{th} root of single-precision floating-point number," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 4, pp. 864–875, Apr. 2020.

[15] F. de Dinechin, P. Echeverría, M. López-Vallejo, and B. Pasca, "Floating-point exponentiation units for reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 1, pp. 1–15, May 2013.

[16] 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.

[17] A. Bisoyi, M. Baral, and M. K. Senapati, "Comparison of a 32-bit vedic multiplier with a conventional binary multiplier," in *Proc. IEEE Int. Conf. Adv. Commun., Control Comput. Technol.*, May 2014, pp. 1757–1760.

[18] 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. 2690–2694, Nov. 2015.

[19] 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.

[20] D. R. Llamocca-Obregón and C. P. Agurto-Ríos, "A fixed-point implementation of the expanded hyperbolic CORDIC algorithm," *Latin Amer. Appl. Res.*, vol. 37, no. 1, pp. 83–91, 2007.

Hui Chen (Graduate Student Member, IEEE) received the B.S. degree in electronic information science and technology from the Jinling College, Nanjing University, Nanjing, China, in 2017, where he is currently pursuing the Ph.D. degree with the School of Electronic Science and Engineering. His current research interests include digital integrated circuit design, VLSI IP optimization, and reconfigurable computing.

Heping Yang received the B.S. degree in electronic information science and technology from the Jinling College, Nanjing University, Nanjing, China, in 2019, where he is currently pursuing the M.S. degree. His research interest includes digital integrated circuit design.

Wenqing Song (Student Member, IEEE) received the B.S. degree from the School of Electronic Science and Engineering, Southeast University (SEU), Nanjing, China, in 2017. She is currently pursuing the Ph.D. degree in electronic science and engineering with Nanjing University (NJU), Nanjing. Her current research interests include polar coding algorithms and efficient hardware architecture.

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 the 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 the area of 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 network, computer architecture, design automation, and real-time systems.

Yuxiang Fu (Member, IEEE) received the B.S. degree in microelectronics and solid state electronics and the Ph.D. degree in electronic science and technology from Nanjing University, Nanjing, China, in 2013 and 2018, respectively. In 2018, he joined the School of Electronic Science and Engineering, Nanjing University, where he is currently an Associate Research Professor. His current research interests include network-on-chip algorithms/architectures, low-power digital systems, and 3D IC design.

Li Li received the B.S. and Ph.D. degrees from the Hefei University of Technology, Hefei, China, in 1996 and 2002, respectively. She is currently a Professor with the VLSI Design Institute, School of Electronic Science and Engineering, Nanjing University, Nanjing, China. Her current research interests include VLSI design for digital signal processing systems, reconfigurable computing, and multiprocessor system-on-a-chip (MPSoC) architecture design methodology. She is a member of Circuits and Systems for Communications (CASCOM) TC of IEEE CAS Society.

Zongguang Yu received the B.S. and M.S. degrees in electronic science engineering from Xidian University, Xi'an, China, in 1985 and 1988, and the Ph.D. degree from Southeast University, Nanjing, China, in 1997. Since 2016, he has been an Adjunct Professor of Nanjing University, Nanjing. His current research interests include high-speed memory, CMOS analog, and mixed mode integrated circuit design.