electronics

Article

Low-Latency and Minor-Error Architecture for Parallel Computing X^Y-like Functions with High-Precision Floating-Point Inputs

Ming Liu [1,*], Wenjia Fu [2] and Jincheng Xia [2]

1 School of Microelectronics, Shenzhen Institute of Information Technology, Shenzhen 518000, China
2 Shenzhen Key Laboratory of IoT Key Technology, Harbin Institute of Technology, Shenzhen 518000, China; wenjiafu@stu.hit.edu.cn (W.F.); jinchengxia@stu.hit.edu.cn (J.X.)
* Correspondence: lium@sziit.edu.cn or lm_hit_1986@126.com; Tel.: +86-0755-8922-6908

Abstract: This paper proposes a novel architecture for the computation of X^Y-like functions based on the QH CORDIC (Quadruple-Step-Ahead Hyperbolic Coordinate Rotation Digital Computer) methodology. The proposed architecture converts direct computing of function X^Y to logarithm, multiplication, and exponent operations. The QH CORDIC methodology is a parallel variant of the traditional CORDIC algorithm. Traditional CORDIC suffers from long latency and large area, while the QH CORDIC has much lower latency. The computation of functions $\ln x$ and e^x is accomplished with the QH CORDIC. To solve the problem of the limited range of convergence of the QH CORDIC, this paper employs two specific techniques to enlarge the range of convergence for functions $\ln x$ and e^x, making it possible to deal with high-precision floating-point inputs. Hardware modeling of function X^Y using the QH CORDIC is plotted in this paper. Under the TSMC 65 nm standard cell library, this paper designs and synthesizes a reference circuit. The ASIC implementation results show that the proposed architecture has 30 more orders of magnitude of maximum relative error and average relative error than the state-of-the-art. On top of that, the proposed architecture is also superior to the state-of-the-art in terms of latency, word length and energy efficiency (power $\times$ latency $\times$ period /efficient bits).

Keywords: floating point; X^Y-like functions; QH CORDIC; high accuracy; low latency

check for updates

Citation: Liu, M.; Fu, W.; Xia, J. Low-Latency and Minor-Error Architecture for Parallel Computing X^Y-like Functions with High-Precision Floating-Point Inputs. *Electronics* **2022**, *11*, 69. https://doi.org/10.3390/electronics11010069

Academic Editor: Shinichi Yamagiwa

Received: 22 November 2021
Accepted: 24 December 2021
Published: 27 December 2021

1. Introduction

X^Y-like functions usually find their place in engineering and scientific applications such as digital signal processing, real-time 3D (three dimensions) graphics, scientific computing and so forth [1]. Currently, customized hardware designs for X^Y-like functions are becoming more promising due to the demanding timing constraints of these applications [2].

For most scientific applications, 64-bit floating-point (FP) numbers conforming to the IEEE-754(2008) standard are extensively applied. However, a rapidly growing number of scientific applications such as climate modeling, fluid mechanics, and economic analysis require a higher level of numerical precision [3]. This means that hundreds or more digits, such as 128 bits, are needed to gain valid numerical results.

Since $X^Y = e^{Y\ln X}$, the computation of function X^Y can be decomposed into the logarithmic computation of $\ln X$, the multiplication of $\ln X$ and Y, and the exponential computation $e^{Y\ln X}$. Therefore, the computation of X^Y-like functions can be converted to the computation of logarithmic and exponential functions. The hardware methods of computing X^Y-like functions directly or indirectly can be divided into four categories: digit-recurrence [4–6], functional iteration [7–9], LUT (Look-Up Table)-based [10–12], and CORDIC-based [13–15]. In fact, CORDIC is one of the digit-recurrence algorithms.

The LUT method is simple and convenient, since look-up tables give the stored data according to their address index. Chen et al. [12] proposed a symmetric-mapping LUT-based method and an architecture to directly compute X^Y-like functions. The size of LUTs has a linear relation with the word length of address indexes and an exponential relation with the word length of storage data. It can be seen that the LUT method performs well in the case of low precision. However, as the precision of inputs increases, the size of LUTs expands dramatically. Under the requirement of high-precision inputs such as 128-bit FP inputs, the size of LUTs seems obviously unreasonable.

The functional iteration method usually computes $e^{Y\ln X}$ or $2^{Y\log 2X}$ indirectly instead of computing X^Y-like functions directly. Its principle is to transform logarithms and exponentials into a sum of a series of power exponents. Refs. [16–19] evaluate binary logarithms and exponentials via simple piecewise linear approximation. The main shortcoming of a simple linear approximation method is the high relative error with limited lookup tables. Paul et al. [20] use a second-order polynomial approximation method to reduce the relative error. In addition to binary logarithms and exponentials, a piecewise polynomial approximation for natural logarithms and exponentials is presented in [21]. To approximate natural logarithms and exponentials, Langhammer and Pasca [9] adopt a third-order piecewise Taylor expansion approximation method, which yields a faster processing speed than in [21]. All the functional iteration methods above require LUTs. LUTs of larger size contribute to higher computing precision. However, the improvement of calculation precision calls for more terms of power exponents and more LUTs with much larger sizes, which leads to an increase in both the computing complexity and timing latency. Therefore, the functional iteration method is likewise not suitable for X^Y-like functions with high-precision inputs.

Compared with the LUT and functional iteration methods, the digit-recurrence method is more suitable for hardware implementation of X^Y-like functions with wide word length. The characteristics of digit recurrence are bit-by-bit approximation and repeated calculation, which can be concisely constructed with logical and control blocks. The precision of the digit-recurrence method increases with the number of iterations. Pineiro et al. [4] exploits this method with high-radix arithmetic units to reduce the number of iterations when extracting binary logarithms and exponentials. Today, many commercial applications demand decimal floating-point arithmetic units. Therefore, Chen et al. [22,23] propose architectures for decimal logarithms and exponentials using the digit-recurrence method.

There are many kinds of digit-recurrence methods, including the CORDIC algorithm, the Gold–Schmidt (GS) iteration method, the Newton–Raphson (NR) iteration method, and so on. The convergence speed of GS iteration depends on the precision of the seed values corresponding to inputs. LUTs employed in Gold-Schmidt iteration to store seed values will expand sharply for high-precision inputs, with a similar effect on timing latency. The NR iteration method requires an initial guess, which may result in different precisions in the outputs [24–26]. The hardware complexity of the NR method increases with the increasing value of N.

Contrary to GS and NR iterations, the CORDIC algorithm is simple to implement, only includes shift and addition operations, and has a stable convergence speed and relatively fixed sizes of LUTs. Therefore, the CORDIC algorithm is widely used in the realization of transcendental functions.

The majority of CORDIC architectures for computing X^Y-like functions are based on the hyperbolic CORDIC algorithm. Traditional hyperbolic CORDIC suffers from high latency, large circuit area, and costly power consumption for the sake of its linear convergence speed.

In the latest literatures, Mack et al. [27] proposed an expanded hyperbolic CORDIC algorithm to compute powering functions for any FP number. Mopuri and Acharyya [14] did work on N^{th} power computations based on the binary hyperbolic CORDIC, which is a special case of the generalized hyperbolic CORDIC [13]. Luo et al. [28] employed the generalized hyperbolic CORDIC to directly compute logarithms and exponentials with an arbitrary fixed base.

However, there is not much research on general computation for X^Y-like functions with high-precision FP inputs. Such a generic approach for N^{th} power computation is proposed in [1] based on the natural logarithm–exponent relation, i.e., $X^Y = e^{Y\ln X}$.

Duprat and Muller [29] introduced the Branching CORDIC algorithm, which enables a fast implementation of the CORDIC algorithm by performing two basic CORDIC rotations in parallel in two separate modules. D.S. Phatak [30] has improved the algorithm and proposed a double step branching CORDIC algorithm where two circular mode rotations are performed in a single step with little additional hardware. To achieve the goal of high-precision, high-accuracy, and low-latency in computing X^Y-like functions, this paper adopts the QH CORDIC methodology [31], which is inspired by the double step branching CORDIC algorithm. In this paper,

- We propose a parallel computing architecture with low-latency based on the QH CORDIC methodology;
- We enlarge the feasible range of FP inputs of the proposed architecture with specific techniques to make sure the proposed architecture applies to high-precision computing;
- We conduct hardware modeling on the proposed architecture to achieve the lowest possible circuit complexity and resource consumption;
- We compare the hardware implementation results with related works to show the minor-error and high-accuracy features of the proposed architecture.

The rest of this paper is organized as follows. Section 2 provides the necessary theoretical background of the QH CORDIC methodology. Section 3 introduces the hardware modeling of X^Y-like functions for 128-bit FP numbers. Section 4 shows the ASIC implementation results of the proposed architecture and compares it with the state-of-the-art in terms of correctness, word length, timing, and power. Section 5 concludes this paper.

2. QH CORDIC-Based Methodology of X^Y-Like Functions

In Section 2, emphasis is placed on the feasibility of logarithmic function ln$x$ and exponential function e^x computing with the QH CORDIC methodology.

We first review the QH CORDIC methodology in terms of iterative formulae. Then, we discuss the range of convergence (ROC) of the QH CORDIC methodology. Given the ROC of the QH CORDIC, the validity of computing of logarithmic function ln$x$ and exponential function e^x based on the QH CORDIC is analyzed.

2.1. Iterative Formulae of QH CORDIC Methodology

Based on shift-addition and vector rotation, the hyperbolic CORDIC algorithm is simple and efficient. However, hyperbolic CORDIC only generates one accurate bit per iteration, which is an apparent drawback for real-time scientific computing. Unlike traditional hyperbolic CORDIC, the QH CORDIC methodology combinates four sequential iterations into a single integrated iteration, greatly cutting down the quantity of iterations.

The iterative formulae of hyperbolic CORDIC are shown in Equation (1):

$$\begin{aligned} x_{n+1} &= x_n + \sigma_n 2^{-n} y_n \\ y_{n+1} &= y_n + \sigma_n 2^{-n} x_n \\ z_{n+1} &= z_n - \sigma_n \theta_n \end{aligned} \tag{1}$$

where σ is the rotation direction of the n-th rotation, $\theta_n = \tanh^{-1}(2^{-n})$, and n starts from 1.

Accordingly, the iterative formulae of the QH CORDIC are shown in Equation (2):

$$\begin{aligned}
x_{n+4} &= \Big\{1 + \sigma_{n+3}\sigma_{n+2}\sigma_{n+1}\sigma_n * 2^{-(4n+6)} \\
&+[16\sigma_{n+1}\sigma_n + 8\sigma_{n+2}\sigma_n + 4\sigma_{n+2}\sigma_{n+1} + 4\sigma_{n+3}\sigma_n + 2\sigma_{n+3}\sigma_{n+1} + \sigma_{n+3}\sigma_{n+2}] * 2^{-(2n+5)}\Big\} * x_n \\
&+\Big\{[8\sigma_n + 4\sigma_{n+1} + 2\sigma_{n+2} + \sigma_{n+3}] * 2^{-(n+3)} \\
&+[8\sigma_{n+2}\sigma_{n+1}\sigma_n + 4\sigma_{n+3}\sigma_{n+1}\sigma_n + 2\sigma_{n+3}\sigma_{n+2}\sigma_n + \sigma_{n+3}\sigma_{n+2}\sigma_{n+1}] * 2^{-(3n+6)}\Big\} * y_n \\
y_{n+4} &= \Big\{1 + \sigma_{n+3}\sigma_{n+2}\sigma_{n+1}\sigma_n * 2^{-(4n+6)} \\
&+[16\sigma_{n+1}\sigma_n + 8\sigma_{n+2}\sigma_n + 4\sigma_{n+2}\sigma_{n+1} + 4\sigma_{n+3}\sigma_n + 2\sigma_{n+3}\sigma_{n+1} + \sigma_{n+3}\sigma_{n+2}] * 2^{-(2n+5)}\Big\} * y_n \\
&+\Big\{[8\sigma_n + 4\sigma_{n+1} + 2\sigma_{n+2} + \sigma_{n+3}] * 2^{-(n+3)} \\
&+[8\sigma_{n+2}\sigma_{n+1}\sigma_n + 4\sigma_{n+3}\sigma_{n+1}\sigma_n + 2\sigma_{n+3}\sigma_{n+2}\sigma_n + \sigma_{n+3}\sigma_{n+2}\sigma_{n+1}] * 2^{-(3n+6)}\Big\} * x_n \\
z_{n+4} &= z_n - \sigma_{n+3}\theta_{n+3} - \sigma_{n+2}\theta_{n+2} - \sigma_{n+1}\theta_{n+1} - \sigma_n\theta_n
\end{aligned} \tag{2}$$

where σ_n, σ_{n+1}, σ_{n+2}, and σ_{n+3} are rotation directions of the n-th, (n + 1)-th, (n + 2)-th, and (n + 3)-th rotations; $\theta_n = \tanh^{-1}(2^{-n})$, $\theta_{n+1} = \tanh^{-1}[2^{-(n+1)}]$, $\theta_{n+2} = \tanh^{-1}[2^{-(n+2)}]$, and $\theta_{n+3} = \tanh^{-1}[2^{-(n+3)}]$; and n starts from 1.

The key of the QH CORDIC lies in the prediction of σ in four sequential steps, which is also the necklace of traditional hyperbolic CORDIC. In basic hyperbolic CORDIC, the value of σ is either -1 (rotating in a clockwise direction) or 1 (rotating in a counterclockwise direction). For a group of four sequential steps, the corresponding $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ has 16 possible cases for its values, from $\{-1, -1, -1, -1\}$ to $\{1, 1, 1, 1\}$.

Substitute the 16 possible cases of $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ into (2) and obtain the 16 corresponding simplified expressions for x_{n+4}, y_{n+4} and z_{n+4}. Table 1 details the corresponding iterative formulae of y_{n+4} when $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ ranges from $\{-1, -1, -1, -1\}$ to $\{1, 1, 1, 1\}$. Since the corresponding iterative formulae of x_{n+4} are almost the same as those of y_{n+4}, a table that lists the iterative formulae of x_{n+4} is omitted.

Table 1. Corresponding iterative formula of y_{n+4}.

Case	σ_n	σ_{n+1}	σ_{n+2}	σ_{n+3}	Iterative Formula of y_{n+4}
1	−1	−1	−1	−1	$y_{n+4} = [1 + 2^{-(4n+6)} + 35 \times 2^{-(2n+5)}] \times y_n + [201315 \times 2^{-(n+3)} - 15 \times 2^{-(3n+6)}] \times x_n$
2	−1	−1	−1	1	$y_{n+4} = [1–2^{-(4n+6)} + 21 \times 2^{-(2n+5)}] \times y_n + [-13 \times 2^{-(n+3)} - 2^{-(3n+6)}] \times x_n$
3	−1	−1	1	−1	$y_{n+4} = [1–2^{-(4n+6)} + 9 \times 2^{-(2n+5)}] \times y_n + [-11 \times 2^{-(n+3)} + 7 \times 2^{-(3n+6)}] \times x_n$
4	−1	1	−1	−1	$y_{n+4} = [1–2^{-(4n+6)} - 9 \times 2^{-(2n+5)}] \times y_n + [–7 \times 2^{-(n+3)} + 11 \times 2^{-(3n+6)}] \times x_n$
5	1	−1	−1	−1	$y_{n+4} = [1–2^{-(4n+6)} - 21 \times 2^{-(2n+5)}] \times y_n + [2^{-(n+3)} + 13 \times 2^{-(3n+6)}] \times x_n$
6	−1	−1	1	1	$y_{n+4} = [1 + 2^{-(4n+6)} - 2^{-(2n+5)}] \times y_n + [–9 \times 2^{-(n+3)} + 9 \times 2^{-(3n+6)}] \times x_n$
7	−1	1	−1	1	$y_{n+4} = [1 + 2^{-(4n+6)} - 15 \times 2^{-(2n+5)}] \times y_n + [–5 \times 2^{-(n+3)} + 5 \times 2^{-(3n+6)}] \times x_n$
8	−1	1	1	−1	$y_{n+4} = [1 + 2^{-(4n+6)} - 19 \times 2^{-(2n+5)}] \times y_n + [–3 \times 2^{-(n+3)} - 3 \times 2^{-(3n+6)}] \times x_n$
9	1	−1	−1	1	$y_{n+4} = [1 + 2^{-(4n+6)} - 19 \times 2^{-(2n+5)}] \times y_n + [3 \times 2^{-(n+3)} + 3 \times 2^{-(3n+6)}] \times x_n$
10	1	−1	1	−1	$y_{n+4} = [1 + 2^{-(4n+6)} - 15 \times 2^{-(2n+5)}] \times y_n + [5 \times 2^{-(n+3)} - 5 \times 2^{-(3n+6)}] \times x_n$
11	1	1	−1	−1	$y_{n+4} = [1 + 2^{-(4n+6)} - 2^{-(2n+5)}] \times y_n + [9 \times 2^{-(n+3)} - 9 \times 2^{-(3n+6)}] \times x_n$
12	−1	1	1	1	$y_{n+4} = [1–2^{-(4n+6)} - 21 \times 2^{-(2n+5)}] \times y_n + [–2^{-(n+3)} - 13 \times 2^{-(3n+6)}] \times x_n$
13	1	−1	1	1	$y_{n+4} = [1–2^{-(4n+6)} - 9 \times 2^{-(2n+5)}] \times y_n + [7 \times 2^{-(n+3)} - 11 \times 2^{-(3n+6)}] \times x_n$
14	1	1	−1	1	$y_{n+4} = [1–2^{-(4n+6)} + 9 \times 2^{-(2n+5)}] \times y_n + [11 \times 2^{-(n+3)} - 7 \times 2^{-(3n+6)}] \times x_n$
15	1	1	1	−1	$y_{n+4} = [1–2^{-(4n+6)}\ 21 \times 2^{-(2n+5)}] \times y_n + [13 \times 2^{-(n+3)} + 2^{-(3n+6)}] \times x_n$
16	1	1	1	1	$y_{n+4} = [1 + 2^{-(4n+6)} + 35 \times 2^{-(2n+5)}] \times y_n + [15 \times 2^{-(n+3)} + 15 \times 2^{-(3n+6)}] \times x_n$

Table 2 details the corresponding iterative formulae of z_{n+4} when $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ ranges from $\{-1, -1, -1, -1\}$ to $\{1, 1, 1, 1\}$.

Table 2. Corresponding iterative formula of z_{n+4}.

Case	σ_n	σ_{n+1}	σ_{n+2}	σ_{n+3}	Iterative Formula of z_{n+4}
1	−1	−1	−1	−1	$z_{n+4} = z_n + \theta_n + \theta_{n+1} + \theta_{n+2} + \theta_{n+3}$
2	−1	−1	−1	1	$z_{n+4} = z_n + \theta_n + \theta_{n+1} + \theta_{n+2} - \theta_{n+3}$
3	−1	−1	1	−1	$z_{n+4} = z_n + \theta_n + \theta_{n+1} - \theta_{n+2} + \theta_{n+3}$
4	−1	1	−1	−1	$z_{n+4} = z_n + \theta_n - \theta_{n+1} + \theta_{n+2} + \theta_{n+3}$
5	1	−1	−1	−1	$z_{n+4} = z_n - \theta_n + \theta_{n+1} + \theta_{n+2} + \theta_{n+3}$
6	−1	−1	1	1	$z_{n+4} = z_n + \theta_n + \theta_{n+1} - \theta_{n+2} - \theta_{n+3}$
7	−1	1	−1	1	$z_{n+4} = z_n + \theta_n - \theta_{n+1} + \theta_{n+2} - \theta_{n+3}$
8	−1	1	1	−1	$z_{n+4} = z_n + \theta_n - \theta_{n+1} - \theta_{n+2} + \theta_{n+3}$
9	1	−1	−1	1	$z_{n+4} = z_n - \theta_n + \theta_{n+1} + \theta_{n+2} - \theta_{n+3}$
10	1	−1	1	−1	$z_{n+4} = z_n - \theta_n + \theta_{n+1} - \theta_{n+2} + \theta_{n+3}$
11	1	1	−1	−1	$z_{n+4} = z_n - \theta_n - \theta_{n+1} + \theta_{n+2} + \theta_{n+3}$
12	−1	1	1	1	$z_{n+4} = z_n + \theta_n - \theta_{n+1} - \theta_{n+2} - \theta_{n+3}$
13	1	−1	1	1	$z_{n+4} = z_n - \theta_n + \theta_{n+1} - \theta_{n+2} - \theta_{n+3}$
14	1	1	−1	1	$z_{n+4} = z_n - \theta_n - \theta_{n+1} + \theta_{n+2} - \theta_{n+3}$
15	1	1	1	−1	$z_{n+4} = z_n - \theta_n - \theta_{n+1} - \theta_{n+2} + \theta_{n+3}$
16	1	1	1	1	$z_{n+4} = z_n - \theta_n - \theta_{n+1} - \theta_{n+2} - \theta_{n+3}$

For an iteration step of the QH CORDIC in vectoring mode, parallelly compute 16 iterative formulae of y_{n+4} shown in Table 1 and obtain a group of 16 different values. Sort the closest-to-zero value out from the 16 y_{n+4} values and take it as the output of y_{n+4} in the current iteration step of the QH CORDIC. Simultaneously, take $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ corresponding to the iterative formula of the output of y_{n+4} as rotation directions in the current iteration step. Then, the computer outputs x_{n+4} and z_{n+4} with the iterative formulae of x_{n+4} and z_{n+4} corresponding to the rotation directions, respectively.

For an iteration step of the QH CORDIC in rotating mode, parallelly compute 16 iterative formulae of z_{n+4} shown in Table 2 and obtain a group of 16 different values. Sort the closest-to-zero value out from the 16 z_{n+4} values and take it as the output of z_{n+4} in the current iteration step of the QH CORDIC. Simultaneously, take $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ corresponding to the iterative formula of the output of z_{n+4} as the rotation directions in the current iteration step. Then, the computer outputs x_{n+4} and y_{n+4} with the iterative formulae of x_{n+4} and y_{n+4} corresponding to the rotation directions, respectively.

It can be seen in Table 1 that the eight upper iterative formulae of y_{n+4} (Cases 1–8) are partly symmetric to the eight lower iterative formulae (Cases 9–16). Such elaborate symmetry also exists with the iterative formulae of x_{n+4}. To reduce the computational burden for every iteration step of the QH CORDIC, multiplications with the same absolute value of the coefficients can be simplified. By merging repeated multiplications into one multiplication and one sign-inversing operation, it takes 34 additions, 12 multiplications, and four shifts to parallelly finish the computation of the 16 iterative formulae of y_{n+4} shown in Table 1. Similarly, it takes 34 additions, 12 multiplications, and four shifts to parallelly finish the computation of the 16 iterative formulae of x_{n+4}.

2.2. Range of Convergence of QH CORDIC Methodology

The ROCs of traditional hyperbolic CORDIC [32] are showed in Equation (3):

$$\begin{cases} \left|\tan^{-1}\frac{y_1}{x_1}\right| \le \theta_N + \sum\limits_{n=1}^{N-1}\theta_n \\ \left|\tanh^{-1}\frac{y_1}{x_1}\right| \le 1.1182, N \to \infty \\ \left|\frac{y_1}{x_1}\right| \le 0.80694, N \to \infty \end{cases} \tag{3}$$

where y_1 and x_1 are initial inputs. It can be inferred that the angle of an input vector in radians for traditional hyperbolic CORDIC must be located in (−1.1182, 1.1182).

Similar to traditional hyperbolic CORDIC, constraints on the ROC of the QH CORDIC also exist.

Since the logarithmic function ln$u$ cannot be attained directly by the QH CORDIC, the computation of function ln$u$ is done through Equation (4):

$$\ln u = 2\tanh^{-1}\left(\frac{u-1}{u+1}\right). \tag{4}$$

The initial conditions and terminated statuses for QH CORDIC-based computation of ln$u$ are listed in Equations (5) and (6), respectively:

$$\begin{cases} x_1 = u-1 \\ y_1 = u+1 \\ z_1 = 0 \end{cases} \tag{5}$$

$$\begin{cases} x_\infty = \frac{2\sqrt{u}}{K_\infty} \\ y_\infty = 0 \\ z_\infty = \tanh^{-1}\left(\frac{u-1}{u+1}\right) \\ K_\infty = \prod\limits_{i=1}^{\infty} \frac{1}{\sqrt{1-2^{-2i}}} \end{cases} \tag{6}$$

Accompanied by Equation (3), the ROC of input u for function ln$u$ is (0.11, 9.51).

As for the exponential function e^v, the computation of function e^v is done through Equation (7):

$$e^v = \frac{e^v + e^{-v}}{2} + \frac{e^v - e^{-v}}{2} = \cosh v + \sinh v. \tag{7}$$

The initial conditions and terminated statuses for QH CORDIC-based computation of e^v are listed in Equations (8) and (9), respectively:

$$\begin{cases} x_1 = K_\infty \\ y_1 = 0 \\ z_1 = v \\ K_\infty = \prod\limits_{i=1}^{\infty} \frac{1}{\sqrt{1-2^{-2i}}} \end{cases} \tag{8}$$

$$\begin{cases} x_\infty = \cosh v \\ y_\infty = \sinh v \\ z_\infty = 0 \end{cases} \tag{9}$$

According to Equation (3), the ROC of input v for function e^v is (−1.1182, 1.1182).

Furthermore, when iteration times come to be 4, 13, 40, 121, ⋯, $(3^{i+2}-1)/2$, ⋯ where i starts from 0, repeated iterations are necessary in order to ensure the convergence of the QH CORDIC [33]. In this manner, the actual sequence of iteration times of the QH CORDIC is 1, 2, 3, 4, 4, 5, ⋯, 12, 13, 13, ⋯.

2.3. Validity of Computation for Logarithmic Function and Exponential Function with QH CORDIC

As $X^Y = e^{Y\ln X}$, it is necessary to study the validity of the computation for logarithmic function ln$u$ and exponential function e^v with the QH CORDIC, that is to say, to enlarge ROC of the QH CORDIC for logarithmic and exponential functions.

Since the inputs of the proposed architecture for X^Y-like functions are all FP numbers, suppose that the input FP number u is $(-1)^S \times M \times 2^E$, where S is sign of u, E is the exponent of u after correcting bias, M is the mantissa of u after complementing the implicit bit, and $M \in [1, 2)$.

There is nothing ambiguous about $S = 0$ because the input FP number u for the logarithmic function ln$u$ is bound to be positive. So, we obtain $u = M \times 2^E$. Perform natural logarithmic computation of both sides of $u = M \times 2^E$ to obtain

$$\ln u = \ln M + e \times \ln 2 \quad (10)$$

To adjust M into the range of (0.11, 9.51), right-shift M one bit. Represent the right-shifted M as M'. Equation (10) is updated as

$$\ln u = \ln M' + (e + 1) \times \ln 2 \quad (11)$$

From Equation (11) we can see that the computation of ln$u$ can be split into one logarithmic operation and one constant multiplication, as well as one addition.

In a similar way, the validity of the computation for exponential function e^v with the QH CORDIC can also be ensured [15].

3. Hardware Modeling of X^Y-Like Functions with QH CORDIC

The QH CORDIC can be applied to both fixed-point and FP operations. Based on the QH CORDIC, this paper presents a quad-precision (128 bits) FP hardware modeling of X^Y-like functions.

The overall architecture of the quad-precision FP X^Y-like functions is illustrated in Figure 1. The proposed architecture is divided into three parts. As for X^Y-like functions, X is base while Y is index. Inputs are two quad-precision FP numbers: *base* and *index* (X and Y) and two control signals: *clk* and *rst_n*. Outputs are *power_result* and *finish*, which are a 128-bit calculated result of function X^Y and a completed signal of function X^Y, respectively. There are three segments in the overall hardware architecture for X^Y-like functions, the preprocessing module, the QH module, and the postprocessing module.

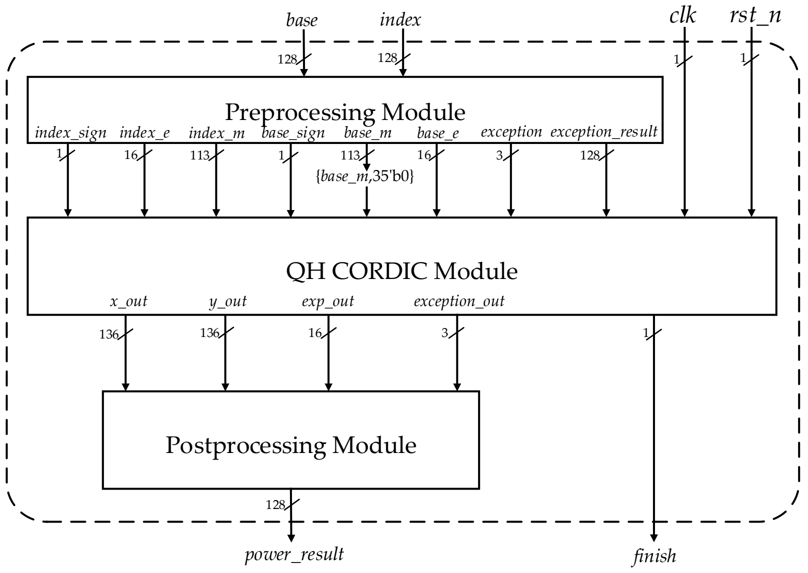


Figure 1. Overall architecture of quad-precision FP X^Y-like functions.

3.1. Preprocessing Module

The preprocessing module judges whether exception situations exist after breaking down the two FP inputs *base* and *index* into three portions: sign (*base_sign* and *index_sign*), exponent (*base_e* and *index_e*) and mantissa (*base_m* and *index_m*). The program flowchart of the preprocessing module is presented in Figure 2. As it is shown in Figure 2, for the quad-precision FP input *base*, *base_sign* is a 1-bit sign; *base_e* is a 15-bit exponent of *base* after correcting bias, and *base_m* is a 113-bit mantissa of *base* after complementing an implicit bit "1". Meanwhile, *index_sign*, *index_e*, and *index_m* are generated in the same way.

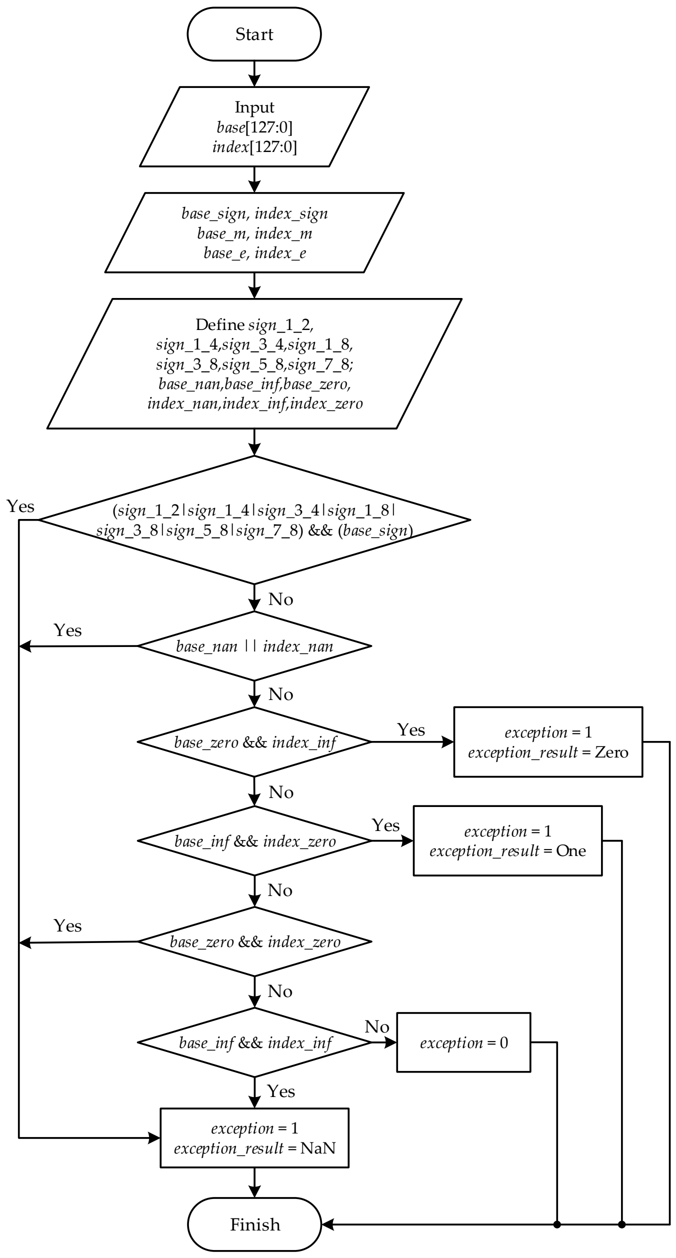


Figure 2. Program flowchart of the preprocessing module.

The new definitions *sign_1_2*, *sign_1_4*, *sign_3_4*, *sign_1_8*, *sign_3_8*, *sign_5_8*, and *sign_7_8* in Figure 2 denote that *index* equals 1/2, 1/4, 3/4, 1/8, 3/8, 5/8, and 7/8, respectively. Although the inputs *base* and *index* have quite a wide range around their numerical values, both of them are still rational numbers, which means that *base* and *index* can be expressed by p/q where p and q are two integers with the denominator q not equal to 0. Denote *base* = p_1/q_1 and *index* = p_2/q_2. When the denominator q_2 is even, *base* must be nonnegative. This is difficult to check in an actual implementation, as p_2 and q_2 of *index* are difficult to confirm when *index* is an irregular high-precision FP number. In this paper, we only focus on eight common cases: *index* equals 1/2, 1/4, 3/4, 1/8, 3/8, 5/8, and 7/8. When *index* = 1/2, 1/4, 3/4, 1/8, 3/8, 5/8, or 7/8, and, in the meantime, *base* is negative (*base_sign* = 1), the 1-bit exception judgement signal *exception* of the preprocessing module is set to be 1 and the 128-bit exception output signal *exception_result* is set to be NaN (128′h7fff_8000_0000_0000_0000_0000 _0000_0000).

Another set of new definitions *base_nan*, *base_inf*, and *base_zero* (*index_nan*, *index_inf*, and *index_zero*) also appears in Figure 2. To be specific, *base_nan* (*index_nan*) means that the

FP input number *base* (*index*) is not a number (NaN); *base_inf* (*index_inf*) means that *base* (*index*) is infinite either negatively or positively; *base_zero* (*index_zero*) means that *base* (*index*) equals to zero.

According to Figure 2, there exist certain cases the where exception judgement signal *exception* is set to be 1 and exception output signal *exception_result* is set to a corresponding numerical value. Otherwise, *exception* is set to be 0, which means that there is no exceptional situation in the preprocessing module. After the preprocessing module, the signals *base_sign*, *index_sign*, *base_e*, *index_e*, *base_m*, *index_m*, *exception*, and *exception_result* are transferred to next module, the QH module.

3.2. QH Module

In this paper, hardware modeling of the function X^Y is done through the logarithmic function $\ln X$, the multiplication of $\ln X$ and Y, and the exponential function $e^{Y\ln X}$. In the QH module, the implementation of the logarithmic function and the exponential function with QH CORDIC methodology in Section 2.1 is abstracted as in Figure 3.

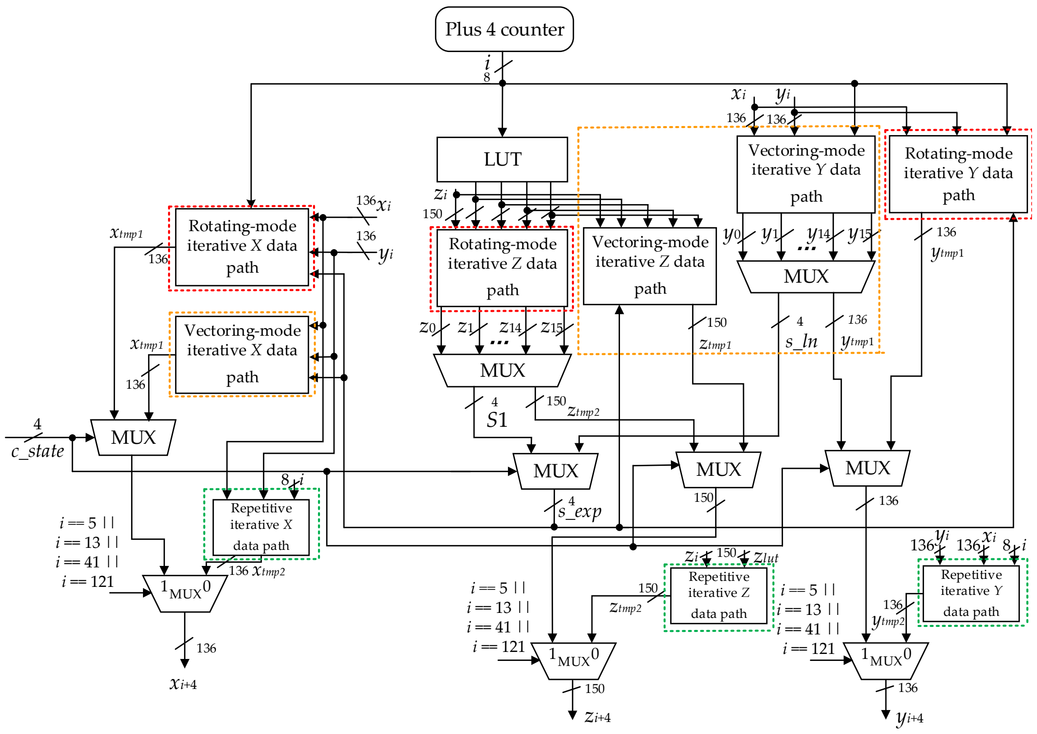

Plus 4 counter
LUT
Rotating-mode iterative X data path
Vectoring-mode iterative X data path
Rotating-mode iterative Z data path
Vectoring-mode iterative Z data path
Vectoring-mode iterative Y data path
Rotating-mode iterative Y data path
Repetitive iterative X data path
Repetitive iterative Z data path
Repetitive iterative Y data path
MUX
c_state
s_ln
s_exp
S1
i == 5 || i == 13 || i == 41 || i == 121
x_{i+4}
z_{i+4}
y_{i+4}

Figure 3. Vectoring mode and rotating mode of QH CORDIC.

In Figure 3, the vectoring mode of the QH CORDIC is for the logarithmic function $\ln X$, while the rotating mode of the QH CORDIC is for exponential function $e^{Y\ln X}$. The vectoring mode of the QH CORDIC bears a close resemblance to the rotating mode of the QH CORDIC because they both have three major data paths and seven submodules. Their differences mainly lie in the signal that determines the rotation direction of the next iteration (signal *s_ln* and signal *s_exp*). As explained in Section 2.2, in order to ensure the ROC of QH CORDIC, when i = 5, 13, 41, 121, ⋯, $(3^{i+2} - 1)/2$, ⋯ where i starts from 0, repeated iterations are needed. Therefore, except for the iterative $X/Y/Z$ data path that performs the iterative formulae $x_{n+4}/y_{n+4}/z_{n+4}$, the iterative $X/Y/Z$ data paths when i = 5, 13, 41, 121 are also listed.

The actual implementation of the QH module establishes a finite state machine (FSM), which has twelve states: INIT_LN, ITE_LN, ONE_STEP_1_LN, ONE_STEP_2_LN, INNER_DEAL_0, INNER_DEAL_1, INNER_DEAL_2, INNER_DEAL_3, INIT_EXP, ITE_EXP, ONE_STEP_1_EXP, and ONE_STEP_2_EXP.

As for the twelve states, the states INIT_LN, ITE_LN, ONE_STEP_1_LN, and ONE_STEP_2_LN belong to the logarithmic function calculating part; the states INIT_EXP, ITE_EXP, ONE_STEP_1_EXP and ONE_STEP_2_EXP belong to the exponential function calculating part; the state INNER_DEAL_0 performs the post-processing of logarithmic function; the state INNER_DEAL_1 belongs to multiplication part; the state INNER_DEAL_2 performs the multiplication in preprocessing of exponential function; the state INNER_DEAL_3 performs the additions in preprocessing of exponential function. A detailed state transition diagram of the FSM is presented in Figure 4.

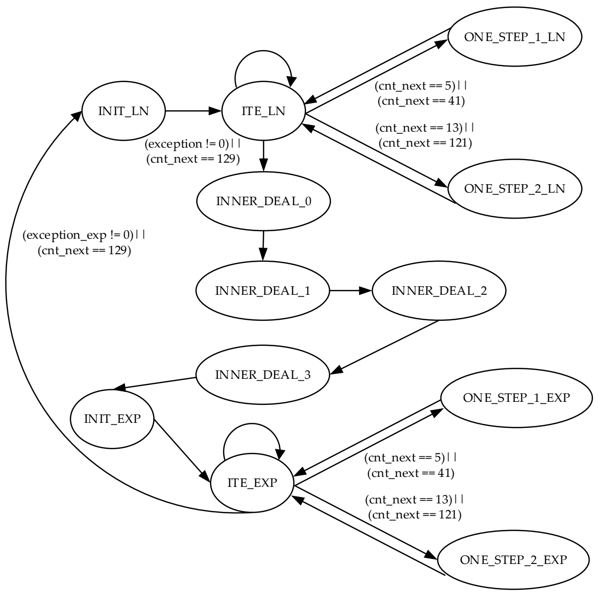


Figure 4. State transition diagram of the finite state machine.

To some extent, the architectures of the twelve states are parallelly similar. Figure 5 shows the initialization process of the logarithmic function and the exponential function. In Figure 5a, *z_in* = {*base_m*, 35′b0} where *base_m* is one of the outputs of the preprocessing module. In Figure 5b, *K_inv* is a constant and its value is as Equation (12).

$$K\_inv = 1/K_{\infty} = 1/(\prod_{i=1}^{\infty} \frac{1}{\sqrt{1-2^{-2i}}}) \tag{12}$$

Input *exp_pre_mantissa* is one of the outputs of the state INNER_DEAL. The architecture of the states INNER_DEAL_0, INNER_DEAL_1, INNER_DEAL_2, and INNER_DEAL_3 is shown in Figure 6.

In Figure 6, the external inputs are *index_s*, *index_m*, *index_e*, *base_e*, *ite_z_ln*, and a constant $1/ln2$. Among them, *index_s*, *index_m*, *index_e*, and *base_e* are the outputs of the preprocessing module, while *ite_z_ln* is one of the outputs of the module z_pre in the state ITE_LN.

The state INNER_DEAL_0 in Figure 6 realizes the normalization of the calculated results *base_e* and *ite_z_ln* after the states INIT_LN, ITE_LN, ONE_STEP_1_LN, and ONE_STEP_2_LN, while the state INNER_DEAL_0 turns the results of logarithmic function $\ln X$ into a normalized 128-bit FP number {*ln_sign*, *ln_e*, *ln_m*}.

INNER_DEAL_1 in Figure 6 computes the multiplication of two 128-bit FP numbers, $\ln X$ and Y. The result of INNER_DEAL_1 is also a normalized 128-bit FP number {*rslt_s*, *rslt_e*, *rslt_e*}.

The states INNER_DEAL_2 and INNER_DEAL_3 realize predealing of the 128-bit FP input {*rslt_s*, *rslt_e*, *rslt_e*} of exponential function $e^{Y \ln X}$, including exception checking and ensuring the computational validity problem of the exponential function. Generally, state

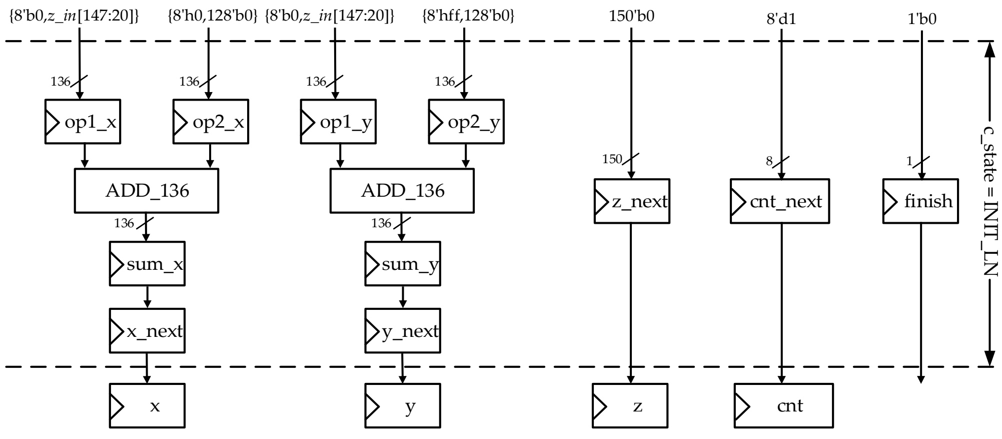
INNER_DEAL_2 performs the former multiplication operation while state INNER_DEAL_3 performs the later multiplication operation and an addition operation.

Figure 7a,b demonstrate the architectures of state ITE_LN and state ITE_EXP, respectively. The yellow box in Figure 7a corresponds to the yellow box in Figure 3, while the red box in Figure 7b corresponds to the red box in Figure 3. Module x_pre in Figure 7a,b is the same, so it is with the module y_pre and the module z_pre. The three important modules x_pre, y_pre, and z_pre consist of iterative data paths of x_{n+4}, y_{n+4}, and z_{n+4} in Equation (2). Furthermore, the signal in the register *cnt_next* and signal *exception* determine the next state after ITE_LN or ITE_EXP.

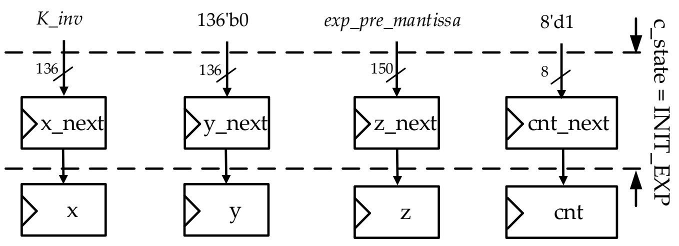
The architecture of module x_pre is shown in detail in Figure 8. The module x_pre is performs iterative X data path calculation and involves many multiplications with single integer constants. Binary decomposition is used to encode single integer constants to reduce delay. Module x_pre divides the iterative X data path into three layers. Layer1 evaluates 2, 4, 8, 16, and 32 times of the inputs x and y with shift operations. Layer2 uses the results of layer1 and the binary decomposed results of single integer constants with a series of compressors and adders to obtain 9, 15, 19, 21, and 35 times the input x, and 3, 5, 7, 9, 11, 13, and 15 times the input y. Layer3 performs shift and compress operations on the results of layer2 according to Equation (2).

Architecture of the module y_pre is quite similar to module x_pre. Iterative Z data path calculation is also optimized and its architecture is shown in Figure 9.

The iterations of the Z data path require 16 calculations for the inputs $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\}$ from 0000 to 1111 to obtain 16 results. The calculating process of each result is quite similar. Figure 9 takes $\{\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}\} = \{0, 0, 0, 0\}$ as an example.



(**a**)



(**b**)

Figure 5. (**a**) Architecture of the state INIT_LN; (**b**) architecture of the state INIT_EXP.

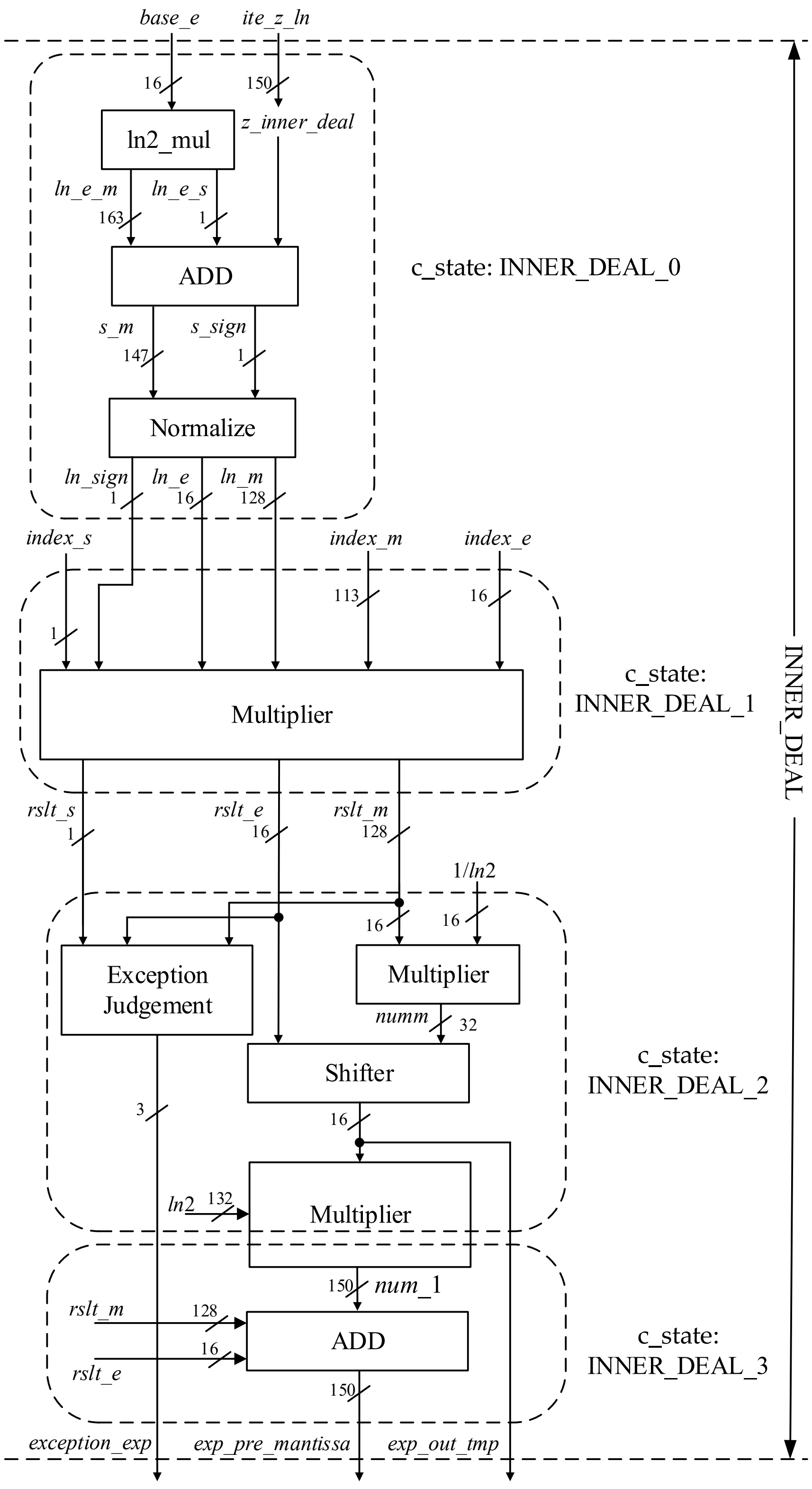


Figure 6. Architecture of states INNER_DEAL_0, INNER_DEAL_1, INNER_DEAL_2, and INNER_DEAL_3.

Figure 10a,b demonstrate architectures of the state ONE_STEP_1_LN and state ONE_STEP_2_LN, respectively. The two green boxes in Figure 10a,b jointly make up the repetitive iterative x_{n+4}, y_{n+4} and z_{n+4} data path for logarithmic functions. However, when register *cnt* is 5 or 41, the FSM jumps to state ONE_STEP_1_LN; when register *cnt* is 13 or 121, the FSM jumps to state ONE_STEP_2_LN.

Figure 11a,b demonstrate architectures of State ONE_STEP_1_EXP and state ONE_STEP_2_EXP, respectively. The two blue boxes in Figure 11a,b jointly make up the repetitive iterative x_{n+4}, y_{n+4} and z_{n+4} data path for exponential functions.

3.3. Postprocessing Module

After the QH module, the outputs *x_out*, *y_out*, *exp_out*, and *exception_out* are generated. As Figure 12 shows, the signal *exp_pre_exp* is just *exp_out* of the QH module and signal *exception_out_inner* is *exception_out* of the QH module. The postprocessing module mainly serves to merge a normalized 128-bit FP output of powering function X^Y.

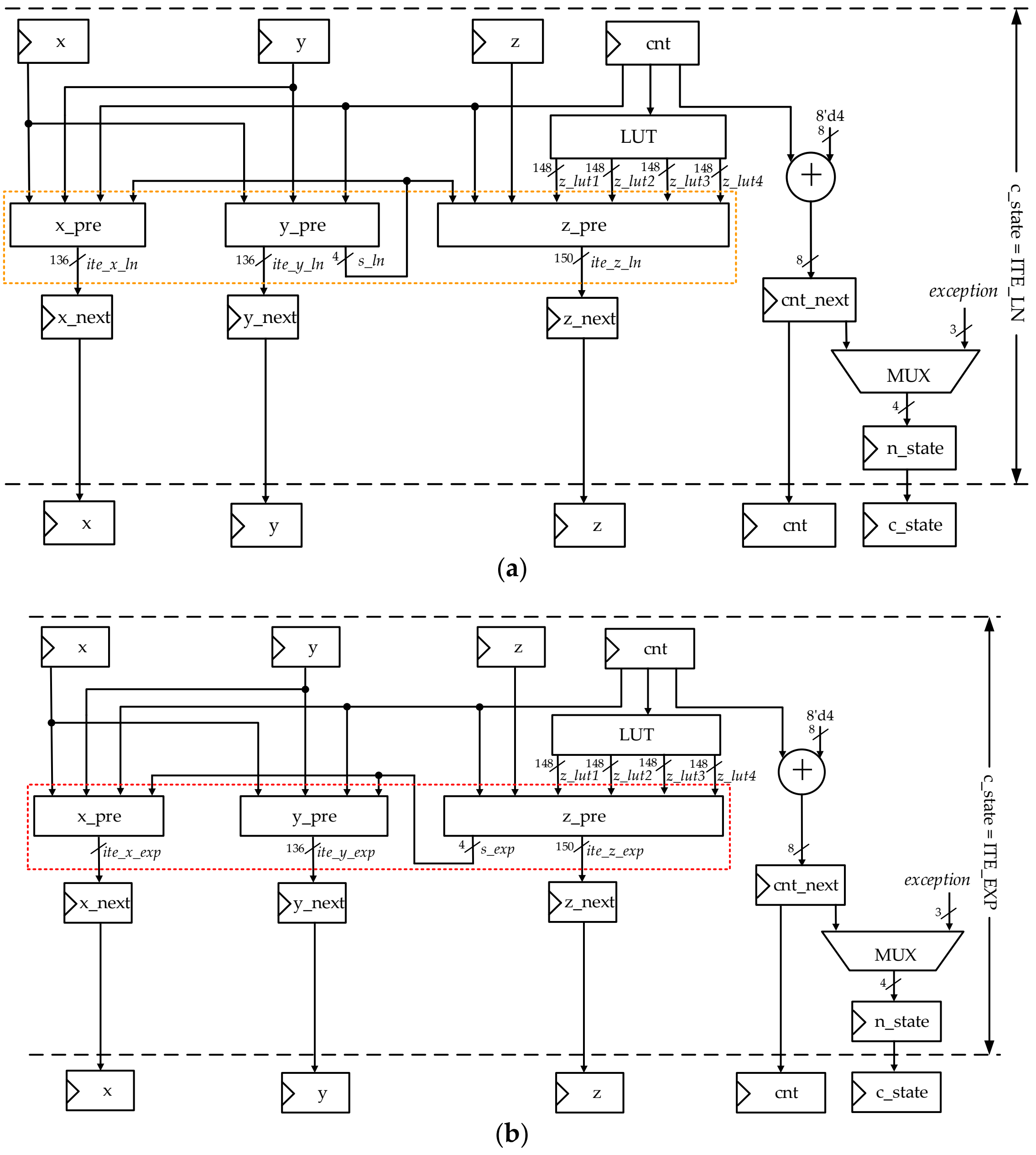


Figure 7. (**a**) Architecture of the state ITE_LN; (**b**) architecture of the state ITE_EXP.

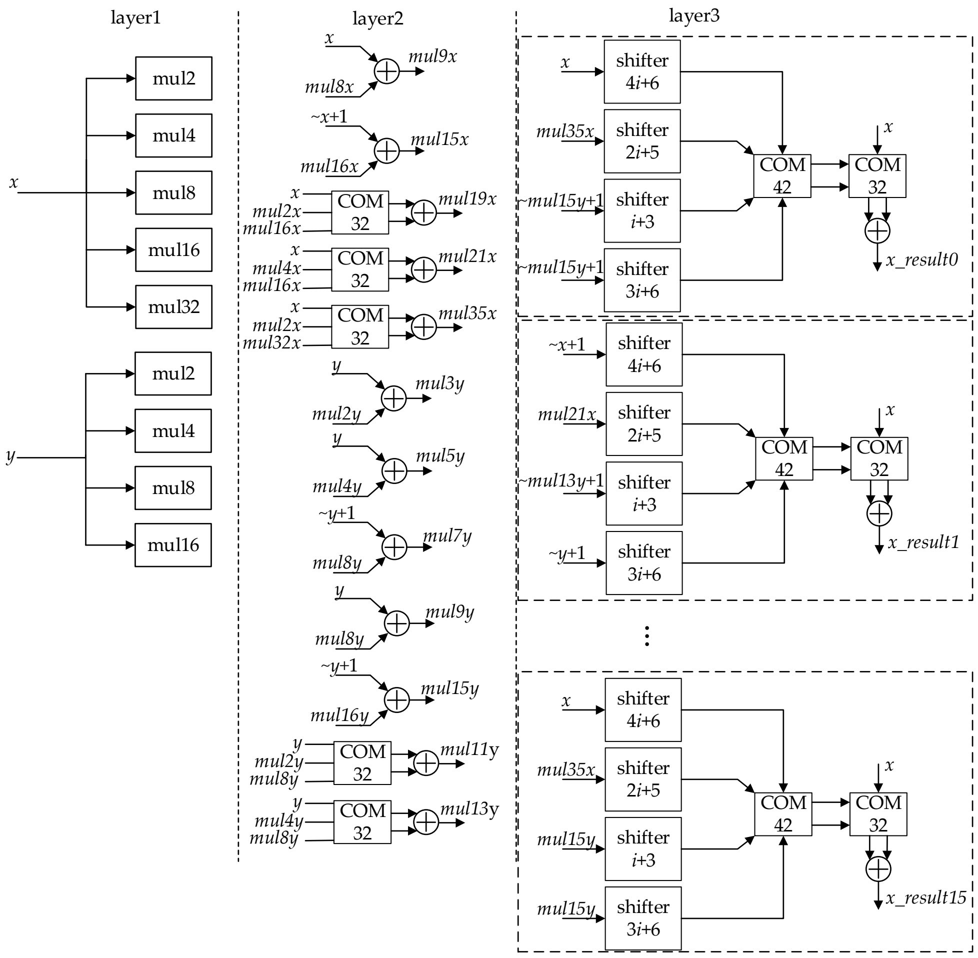
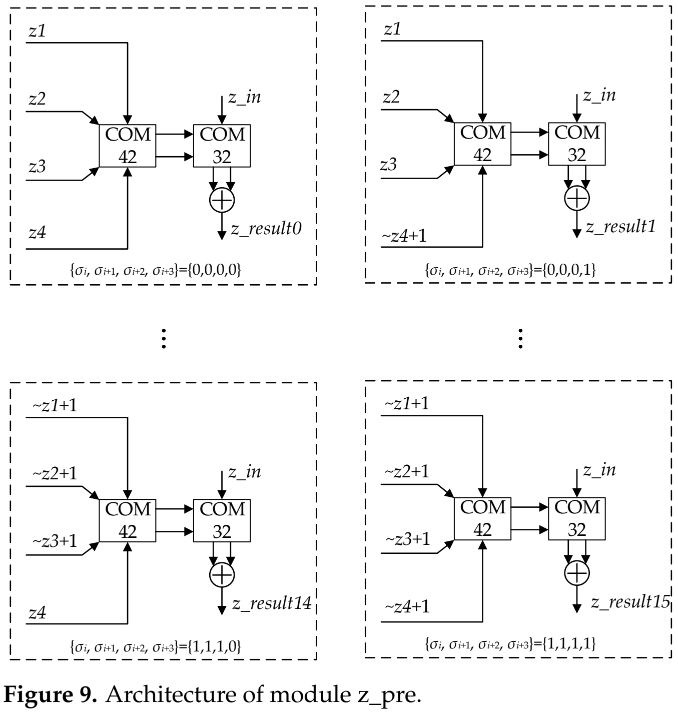


Figure 8. Architecture of the module x_pre.



z1
z2
z3
z4
~z4+1
~z1+1
~z2+1
~z3+1
z_in
COM 42
COM 32
z_result0
z_result1
z_result14
z_result15
{σi, σi+1, σi+2, σi+3}={0,0,0,0}
{σi, σi+1, σi+2, σi+3}={0,0,0,1}
{σi, σi+1, σi+2, σi+3}={1,1,1,0}
{σi, σi+1, σi+2, σi+3}={1,1,1,1}

Figure 9. Architecture of module z_pre.

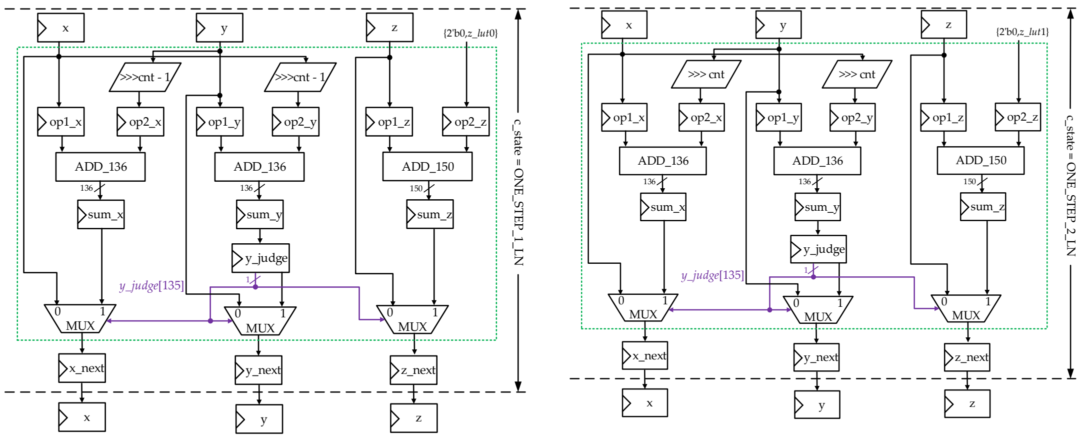
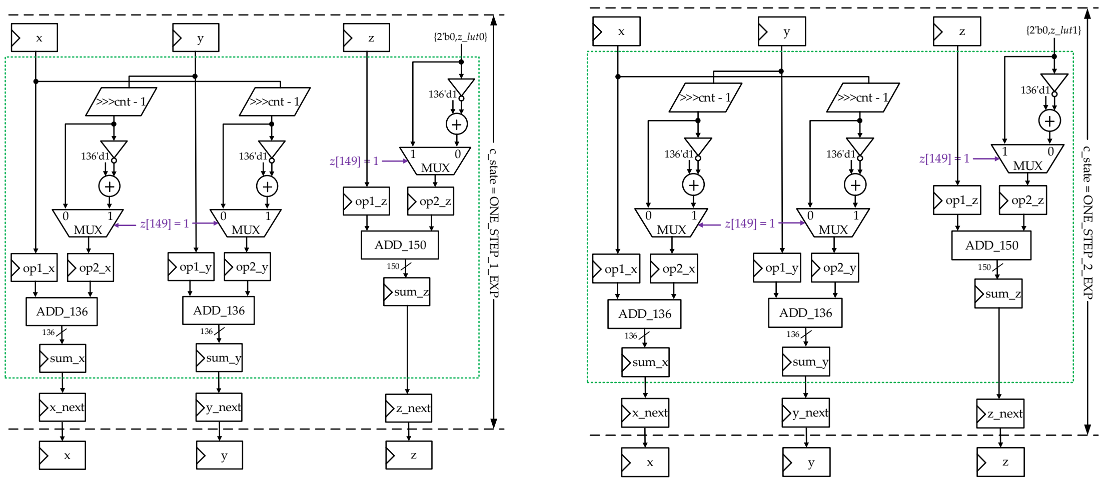


Figure 10. (**a**) Architecture of the state ONE_STEP_1_LN; (**b**) architecture of the state ONE_STEP_2_LN.



(**a**) (**b**)

Figure 11. (**a**) Architecture of the state ONE_STEP_1_EXP; (**b**) architecture of the state ONE_STEP_2_EXP.

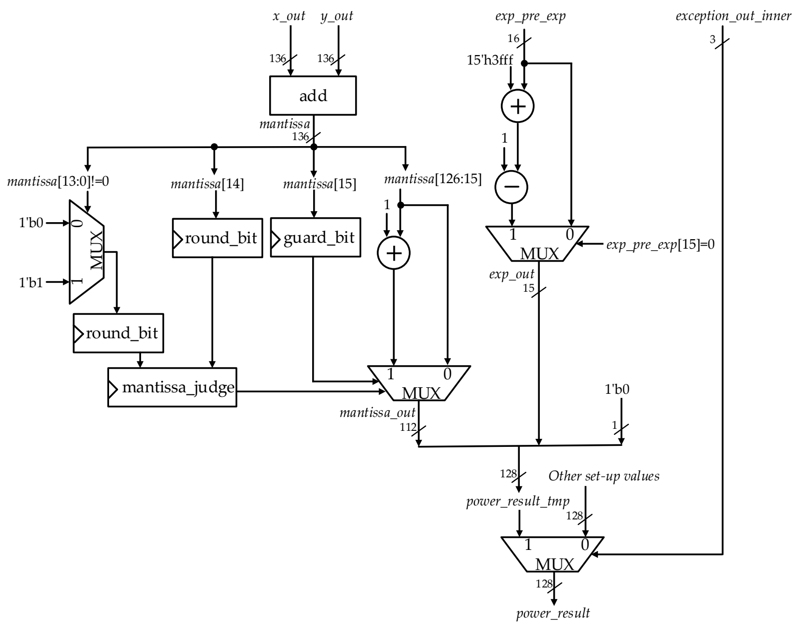


Figure 12. Architecture of the postprocessing module.

4. Implementation Results and Comparisons

4.1. ASIC Implementation Results of the Proposed Architecture

The proposed architecture was coded in Verilog HDL and synthesized with the TSMC 65 nm standard cell library, using Synopsys Design Compiler. The proposed architecture was synthesized with the best achievable timing constraints, with the constraint of the max-area set to zero and a global operating voltage of 0.9 V. The ASIC implementation details are shown in Table 3.

Table 3. ASIC Implementation Details @ TSMC 65 nm.

Item	-
Period (ns)	3.3
Latency (cycle)	76
Area (μm^2)	1417366
Power (mW)	36.2189
Precision (bit)	113
Total time (ns) [1]	250.8
ATP (mm^2·ns) [2]	355.4754
Total energy (fJ) [3]	9083.7001
Energy efficiency (fJ/bit) [4]	80.3867
Area efficiency (bit/(mm^2·ns)) [5]	0.3179

[1] Total time = latency × period. [2] ATP = area × total time. [3] Total energy = power × total time. [4] Energy efficiency = total energy/precision. [5] Area efficiency = precision/ATP.

ATP and total energy are usually used to properly and roundly evaluate ASIC performance. The smaller the ATP and total energy are, the better the ASIC design. In a similar way, according to the definitions of energy efficiency and area efficiency, the smaller the energy efficiency is and the larger the area efficiency is, the better the ASIC design.

4.2. Evalutation and Comparative Analysis

In order to reveal the superiority of our approach based on the same conditions, this paper makes a comparative analysis using different indicators against other state-of-the-art approaches for computing X^Y-like functions, including computation correctness, word length, computation latency, and power consumption.

4.2.1. Computational Correctness

In Python, we verify the computational correctness of the proposed architecture and other approaches by evaluating their relative errors. The definition of the indicator relative error (RE) is as Equation (13):

$$RE = \frac{|V_T - V_M|}{|V_T|} \tag{13}$$

where V_T stands for the theoretical value of function X^Y and V_M stands for the measured value of function X^Y with the proposed and other approaches. The maximum relative error is denoted as $max(RE_k)$ of k test quantities. Another important indicator, the average relative error (ARE), is defined as Equation (14):

$$ARE = \frac{\sum_{i=1}^{k} |V_T - V_M|_i}{|V_T| \times k} \tag{14}$$

where k stands for test quantity of function X^Y.

Currently, only [12,14] in the state-of-the-art approaches implement both $\sqrt[N]{X}$ and X^N computation. For [12,14], the test quantity k is 40,000. For [14], the number of iterations n is set to be 10. For [12], 1024 pieces of result data are stored in either LUT1 or LUT2. Software verification of [12,14] and the proposed architecture for function X^Y is presented in Table 4.

Table 4. Verification of [12,14] and the proposed architecture by Python simulation.

Item	$\sqrt[N]{X}$			X^N		
	[14]	**[12]**	**Proposed**	**[14]**	**[12]**	**Proposed**
X	$[10^{-6}, 10^{6}]$	$[10^{-6}, 10^{6}]$	$(2^{-16382}, 2^{16383})$	$[10^{-2}, 10^{2}]$	$[10^{-2}, 10^{2}]$	$(2^{-16382}, 2^{16383})$
N	[2, 1002]	[2, 1002]	$(2^{-16382}, 2^{16383})$	[1, 5]	[1, 5]	$(2^{-16382}, 2^{16383})$
k	40,000	40,000	100,000	40,000	40,000	100,000
$max(RE)$	1.928×10^{-3}	1.069×10^{-3}	1.688×10^{-34}	1.030×10^{-2}	5.272×10^{-3}	1.610×10^{-34}
ARE	5.464×10^{-4}	4.160×10^{-4}	1.446×10^{-34}	2.875×10^{-3}	2.095×10^{-3}	1.442×10^{-34}

From Table 4, in terms of $max(RE)$ and ARE, it is evident that the proposed architecture for $\sqrt[N]{X}$ computation is superior compared with the state-of-the-art approaches [12,14]. The proposed architecture for X^N computation is also superior compared with the state-of-the-art approaches [12,14].

4.2.2. Word Length

In this subsection, this paper analyzes the word length required for each approach's hardware implementation based on the conditions in Section 4.2.1. The longer the word length is, the better the precision.

The word lengths of each module in [12,14], and the proposed architecture are shown in Table 5. By contrast, the word length of the proposed architecture is much longer than the state-of-the-art approaches, which means that the proposed architecture has the characteristic of high precision. However, the long word length consumes more area and power, increases the critical path, and lowers the working frequency to some extent.

Table 5. Word length for [12,14] and the proposed architecture.

Function	**Architecture**	**Type**	**Logarithm**	**Division/Multiplication**	**Exponential**
$\sqrt[N]{X}$	[14]	Module	HV CORDIC	LV CORDIC	HR CORDIC
		S + I + F [1]	1 + 2 + 45	1 + 10 + 27	1 + 2 + 27
		Total Bits	48	38	30
X^N	[14]	Module	BV CORDIC	LV CORDIC	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	[12]	Module	SM-LUT	Multiplier	SM-LUT
		S + I + F	1 + 0 + 27	1 + 10 + 27	1 + 0 + 27
		Total Bits	28	38	28
	Proposed	Module	states INIT_LN, ITE_LN, ONE_STEP_1_LN, ONE_STEP_2_LN	states INNER_DEAL_0, INNER_DEAL_1, INNER_DEAL_2, INNER_DEAL_3	states INIT_EXP, ITE_EXP, ONE_STEP_1_EXP, ONE_STEP_2_EXP
		S + I + F	1 + 15 + 112	1 + 15 + 112	1 + 15 + 112
		Total Bits	128	128	128

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

4.2.3. Timing Analysis and Power Analysis

In this section, this paper first analyzes computation latency of [14,15] and the proposed architecture to calculate $\sqrt[N]{X}$ or X^N.

NL refers to latency savings compared with other architectures and it is defined as Equation (15):

$$NL = L_{oth} - L_{pro} \tag{15}$$

where L is latency. Compared with [14,15], the percentage of NL is as Equation (16):

$$P_{NL} = \frac{NL}{L_{oth}} = \frac{L_{oth} - L_{pro}}{L_{oth}} \times 100\% \tag{16}$$

Under the circumstances, we can calculate NL, P_{NL} for [14,15] and the proposed architecture, as shown in Table 6.

Table 6. Timing analysis for [14,15] and the proposed architecture.

Function	Architecture	Latency (Cycle)	Period (ns)	Total Time (ns) [1]	NL (Cycle)	P_{NL} (%)	Throughput (bit/ns) [2]
$\sqrt[N]{X}$	[14]	78	0.5	39	2	2.56	0.333
X^N	[14]	79	0.5	39.5	3	3.79	0.329
$\sqrt[N]{X}$ and X^N	[15]	87	1	87	11	12.64	0.437
	Proposed	76	3.3	250.8	\	\	0.451

[1] Total time = latency × period. [2] Throughput = precision/total time; the precision of [14,15], and proposed architecture is 13, 38 and 113 respectively.

From Table 6, for $\sqrt[N]{X}$ computation, the proposed architecture saves 2.56% and 12.64% latency compared with [14,15], respectively. For X^N computation, the proposed architecture saves 3.79% and 12.64% latency, respectively. The direct comparison of latency in terms of cycle is not fair, as the periods of [14,15] and the proposed architecture may be different. Hence, the indicator total time is employed to measure the timing efficiency of the three architectures. According to Table 6, the total time of the proposed architecture is about 6 times that of [14] and about 2.8 times that of [15]. It should be noticed that the comparison is done in terms of total time without taking into account the fact that the three architectures run on different technologies.

For fairness, the indicator throughput was also measured. From Table 6, for $\sqrt[N]{X}$ computation, the proposed architecture has ≈35.4% and ≈3.2% throughput overhead over [14] and [15], respectively; for X^N computation, the proposed architecture has ≈37.1% and ≈3.2% throughput overhead over [14,15], respectively. It can be inferred that the throughput of our proposed architecture is larger, as our precision is much higher. For high-precision applications, the proposed architecture can achieve better timing performance.

Next, this paper focuses on the power consumption to calculate $\sqrt[N]{X}$ or X^N for [14,15] and the proposed architecture. Similar with Section 4.1, this paper employs energy efficiency to compare with other architectures.

As seen in Table 7, the proposed design for $\sqrt[N]{X}$ computation saves 25.44% and 34.18% energy efficiency when it is compared with [14,15], respectively. For X^N computation, our design can save 21.58% and 34.18% energy efficiency compared with [14] and [15], respectively.

Table 7. Power analysis for [14,15] and the proposed architecture.

Function	Architecture	Technology	Power (mW)	Period (ns)	Precision (bit)	Energy Efficiency (fJ/bit)
$\sqrt[N]{X}$	[14]	45 nm	35.9	0.5	13	107.7
X^N	[14]	45 nm	33.7	0.5	13	102.4
$\sqrt[N]{X}$ and X^N	[15]	TSMC 40 nm	53.2978	1	38	122.0
	Proposed	TSMC 65 nm	36.2189	3.3	113	80.3

5. Conclusions

In this paper, a novel hardware architecture is proposed based on the QH CORDIC methodology to compute X^Y-like functions. The computation of function X^Y is divided into the computation of function ln$X$, a multiplication operation, and function e^x, of which function ln$x$ and function e^x are calculated using the QH CORDIC.

The QH CORDIC methodology merges four single iterations into a whole parallel iteration, simultaneously computes a total of 16 possible values of x_{n+4}, y_{n+4}, and z_{n+4} respectively, and picks up rotation directions of the next iteration according to z_{n+4} for rotating mode or y_{n+4} for vectoring mode. Compared with traditional hyperbolic CORDIC algorithms, the QH CORDIC only needs 32 clock cycles to achieve 128-bit FP data accuracy, which greatly reduces the circuit latency.

The proposed architecture solves the problem of limited ROC of CORDIC algorithm and explains the validity of computing function ln$x$ and e^x with the QH CORDIC, ensuring that the domain of inputs for the proposed architecture are high-precision (128 bits) floating-point numbers.

The ASIC implementation results show that the proposed architecture's circuit latency, area, and power consumption are 76 cycles, 1417366 μm^2, and 36.2189 mW, respectively, under a working frequency of 300 MHz and a precision of at least 113 bits. The conspicuous timing performances of the proposed architecture with high-precision inputs and dramatically accurate outputs are achieved at the cost of area and power consumption. This is related to the adopted QH CORDIC methodology to a large extent, as the QH CORDIC follows the parallel computing strategy.

Compared with other architectures, the entire logic path turned out to be minor error and high accuracy. The proposed architecture is about 30 orders of magnitude superior to [12,14] in terms of $max(RE)$ and ARE. The proposed architecture was also proved to be low-latency. For $\sqrt[N]{X}$ computation, the proposed architecture has 2.56% and 12.64% latency overhead over [14,15], respectively. For X^N computation, it has 3.79% and 12.64% latency overhead over [14,15], respectively. In addition, the proposed architecture outmatches [14,15] in terms of indicator energy efficiency and throughput. The word length of the proposed architecture is 128 bits, three or four times that of [12,14].

Therefore, the proposed architecture is highly favored to perform high-precision floating-point computing of X^Y-like functions. However, the long word length may weaken the advantages in terms of area and power. Our focus in the future will firstly be rigorous error analysis that cuts down on the word length of x_{n+4}, y_{n+4}, and z_{n+4} during the QH CORDIC iterations. Secondly, the word length of the two multipliers Y and $\ln X$ can be shortened with the accuracy of product $Y\ln X$ maintained. In this way, the computation of $Y\ln X$ in $e^{Y\ln X}$ can be simplified. These improvements will help to reduce the total area and power of the circuit design.

In addition, the proposed hardware architecture can be modified into a low-latency and minor-error architecture for X^Y-like functions whose inputs are configurable in terms of precision. In line with the precisions of current scientific computing applications, not only quadruple precision (128 bits), but also half-precision (16 bits), single precision (32 bits), and double precision (64 bits) are accessible in the configurable architecture.

Author Contributions: Conceptualization, M.L.; methodology, M.L.; software, J.X. and M.L.; validation, J.X. and M.L.; writing—original draft preparation, W.F.; writing—review and editing, W.F. and M.L.; funding acquisition, M.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by a project of the Educational Commission of Guangdong Province of China (2019GKQNCX122) and Scientific Research Project in School-level (SZIIT2019KJ026).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pineiro, J.-A.; Ercegovac, M.D.; Bruguera, J.D. High-radix iterative algorithm for powering computation. In Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain, 15−18 June 2003; pp. 204–211.
2. Harris, D. A powering unit for an Open GL lighting engine. In Proceedings of the 35th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 4–7 November 2001; pp. 1641–1645.
3. Zuras, D.; Cowlishaw, M.; Aiken, A.; Applegate, M.; Bailey, D.; Bass, S.; Bhandarkar, D.; Bhat, M.; Bindel, D.; Boldo, S.; et al. IEEE Standard for Floating-Point Arithmetic. *IEEE Std.* **2008**, *754*, 1–70.
4. Pineiro, J.-A.; Ercegovac, M.D.; Bruguera, J.D. Algorithm and architecture for logarithm, exponential, and powering computation. *IEEE Trans. Comput.* **2004**, *53*, 1085–1096. [CrossRef]
5. Antelo, E.; Lang, T.; Bruguera, J.D. Very-high radix CORDIC vectoring with scalings and selection by rounding. In Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, SA, Australia, 14−16 April 1999; pp. 204–213.
6. Vazquez, A.; Bruguera, J.D. Iterative algorithm and architecture for exponential, logarithm, powering, and root extraction. *IEEE Trans. Comput.* **2013**, *62*, 1721–1731. [CrossRef]
7. Oberman, S.F. Floating point division and square root algorithms and implementation in the AMD-K7/sup TM/ microprocessor. In Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, SA, Australia, 14−16 April 1999; pp. 106–115.

8. Pineiro, J.-A.; Bruguera, J.D. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Trans. Comput.* **2002**, *51*, 1377–1388. [CrossRef]
9. Langhammer, M.; Pasca, B. Single precision logarithm and exponential architectures for hard floating-point enabled FPGAs. *IEEE Trans. Comput.* **2017**, *66*, 2031–2043. [CrossRef]
10. Muller, J.M. Elementary functions: Algorithms and implementation. *Math. Comput. Educ.* **1997**, *34*, 21–52.
11. Schulte, M.J.; Stine, J.E. Approximating elementary functions with symmetric bipartite tables. *IEEE Trans. Comput.* **1999**, *48*, 842–847. [CrossRef]
12. Chen, H.; Yang, H.; Song, W.; Lu, Z.; Fu, Y.; Li, L.; Yu, Z. Symmetric-Mapping LUT-Based Method and Architecture for Computing XY-Like Functions. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2021**, *68*, 1231–1244. [CrossRef]
13. Luo, Y.; Wang, Y.; Sun, H.; Zha, Y.; Wang, Z.; Pan, H. CORDIC-based architecture for computing Nth root and its implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2018**, *65*, 4183–4195. [CrossRef]
14. Mopuri, S.; Acharyya, A. Low complexity generic VLSI architecture design methodology for Nth root and Nth power computations. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2019**, *66*, 4673–4686. [CrossRef]
15. Wang, Y.; Luo, Y.; Wang, Z.; Shen, Q.; Pan, H. GH CORDIC-based architecture for computing Nth root of single-precision floating-point number. *IEEE Trans. Very Large Scale Integr. Syst.* **2020**, *28*, 864–875. [CrossRef]
16. Combet, M.; van Zonneveld, H.; Verbeek, L. Computation of the base two logarithm of binary numbers. *IEEE Trans. Electron. Comput.* **1965**, *EC-14*, 863–867. [CrossRef]
17. Hall, E.L.; Lynch, D.D.; Dwyer, S.J. Generation of products and quotients using approximate binary logarithms for digital filtering applications. *IEEE Trans. Comput.* **1970**, *C-19*, 97–105. [CrossRef]
18. Abed, K.H.; Siferd, R.E. CMOS VLSI implementation of a low-power logarithmic converter. *IEEE Trans. Comput.* **2003**, *52*, 1421–1433. [CrossRef]
19. Abed, K.H.; Siferd, R.E. VLSI implementation of a low-power antilogarithmic converter. *IEEE Trans. Comput.* **2003**, *52*, 1221–1228. [CrossRef]
20. Paul, S.; Jayakumar, N.; Khatri, S.P. A fast hardware approach for approximate, efficient logarithm and antilogarithm computations. *IEEE Trans. Very Large Scale Integr. Syst.* **2009**, *17*, 269–277. [CrossRef]
21. De Dinechin, F.; Pasca, B. Floating-point exponential functions for DSP-enabled FPGAs. In Proceedings of the IEEE International Conference on Field-Programmable Technology, Beijing, China, 8−10 December 2010; pp. 110–117.
22. Chen, D.; Han, L.; Ko, S.B. Decimal floating-point antilogarithmic converter based on selection by rounding: Algorithm and architecture. *IET Comput. Digit. Technol.* **2012**, *6*, 277–289. [CrossRef]
23. Chen, D.; Han, L.; Choi, Y.; Ko, S.-B. Improved decimal floating-point logarithmic converter based on selection by rounding. *IEEE Trans. Comput.* **2012**, *61*, 607–621. [CrossRef]
24. Liu, W.; Nannarelli, A. Power efficient division and square root unit. *IEEE Trans. Comput.* **2012**, *61*, 1059–1070. [CrossRef]
25. Seth, A.; Gan, W.-S. Fixed-point square roots using L-b truncation. *IEEE Signal Process. Mag.* **2011**, *28*, 149–153. [CrossRef]
26. Kabuo, H.; Taniguchi, T.; Miyoshi, A.; Yamashita, H.; Urano, M.; Edamatsu, H.; Kuninobu, S. Accurate rounding scheme for the Newton-Raphson method using redundant binary representation. *IEEE Trans. Comput.* **1994**, *43*, 43–51. [CrossRef]
27. Mack, J.; Bellestri, S.; Llamocca, D. Floating point CORDIC-based architecture for powering computation. In Proceedings of the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Riviera Maya, Mexico, 7–9 December 2015; pp. 1–6.
28. Luo, Y.; Wang, Y.; Ha, Y.; Wang, Z.; Chen, S.; Pan, H. Generalized Hyperbolic CORDIC and Its Logarithmic and Exponential Computation with Arbitrary Fixed Base. *IEEE Trans. Very Large Scale Integr. Syst.* **2019**, *27*, 2156–2169. [CrossRef]
29. Duprat, J.; Muller, J.M. The CORDIC algorithm: New results for fast VLSI implementation. *IEEE Trans. Comput.* **1993**, *42*, 168–178. [CrossRef]
30. Phatak, D.S. Double step branching CORDIC: A new algorithm for fast sine and cosine generation. *IEEE Trans. Comput.* **1998**, *47*, 587–602. [CrossRef]
31. Fu, W.; Xia, J.; Lin, X.; Liu, M.; Wang, M. Low-Latency Hardware Implementation of High-Precision Hyperbolic Functions Sinh$x$ and Cosh$x$ Based on Improved CORDIC Algorithm. *Electronics* **2021**, *10*, 2533. [CrossRef]
32. Llamocca-Obregón, D.R.; Agurto-Ríos, C.P. A fixed-point implementation of the expanded hyperbolic CORDIC algorithm. *Lat. Am. Appl. Res.* **2007**, *37*, 83–91.
33. Hao, L.; Ming-Jiang, W.; Mo-Ran, C.; Ming, L. A VLSI Implementation of Double Precision Floating-Point Logarithmic Function. In Proceedings of the 2019 IEEE 4th International Conference on Signal and Image Processing (ICSIP), Wuxi, China, 19–21 July 2019; pp. 345–349.