symmetry

MDPI

Article

A Unified Hardware Design for Multiplication, Division, and Square Roots Using Binary Logarithms

Dat Ngo [1], Siyeon Han [2] and Bongsoon Kang [2,*]

1 Department of Computer Engineering, Korea National University of Transportation, Chungju 27469, Republic of Korea; datngo@ut.ac.kr
2 Department of Electronic Engineering, Dong-A University, Busan 49315, Republic of Korea; 1923602@donga.ac.kr
* Correspondence: bongsoon@dau.ac.kr; Tel.: +82-51-200-7703

Abstract: Multiplication, division, and square root operations introduce significant challenges in digital signal processing (DSP) systems, traditionally requiring multiple operations that increase execution time and hardware complexity. This study presents a novel approach that leverages binary logarithms to perform these operations using only addition, subtraction, and shifts, enabling a unified hardware implementation—a marked departure from conventional methods that handle these operations separately. The proposed design, involving logarithm and antilogarithm calculations, exhibits an algebraically symmetrical pattern that further optimizes the processing flow. Additionally, this study introduces innovative log-domain correction terms specifically designed to minimize computation errors—a critical improvement over existing methods that often struggle with precision. Compared to standard hardware implementations, the proposed design significantly reduces hardware resource utilization and power consumption while maintaining high operational frequency.

Keywords: unified hardware; multiplication; division; square root; binary logarithm

check for updates

Citation: Ngo, D.; Han, S.; Kang, B. A Unified Hardware Design for Multiplication, Division, and Square Roots Using Binary Logarithms. *Symmetry* **2024**, *16*, 1138. https://doi.org/10.3390/sym16091138

Academic Editors: Yunfei Fang and Peng Wu

Received: 31 July 2024
Revised: 29 August 2024
Accepted: 1 September 2024
Published: 2 September 2024

1. Introduction

Digital signal processing (DSP) systems perform tasks such as discrete cosine transform, fast Fourier transform, and image filtering, which require intensive use of multiplication, division, and square root operations. In a binary number system (BNS), standard hardware implementations of these computationally complex operations are expensive in terms of area, delay, and power consumption. Although fixed-point number representation can reduce these complexities, it is prone to overflow and scaling issues. On the other hand, floating-point number representation offers better precision and scaling but introduces more overhead.

The logarithmic number system (LNS) combines the advantages of fixed-point and floating-point number representations, namely, simplicity and precision. In the logarithmic domain (hereinafter referred to as the log-domain), multiplication and division are transformed into addition and subtraction, significantly simplifying hardware implementation. Despite inevitable errors in computing the logarithms of input data, log-domain arithmetic remains preferable in many DSP applications due to its benefits in reducing area, delay, and power consumption.

Two traditional methods are widely used to compute logarithms: the Taylor's series expansion and the look-up table (LUT) [1,2]. The Taylor's series method expresses the logarithm as an infinite sum of terms, with the first $(n+1)$ terms constituting the nth Taylor polynomial. Higher degrees of n yield more accurate approximations to the logarithm. The LUT method, in contrast, uses a complete table of logarithms for all numbers. Both methods require substantial memory, rendering them computationally inefficient. To address this, Mitchell [3] proposed a straightforward method for computing logarithms and antilogarithms using piece-wise linear approximations, which is memory-efficient and hardware-friendly, albeit at the cost of some accuracy. The logarithm error in Mitchell's approximation ranges from 0 to 0.08639, leading to corresponding absolute errors in log-domain arithmetic operations: 11.1% for multiplication, 12.5% for division, and 2.9% for square root.

Mitchell [3] also derived a correction term to reduce the error, but this term is computationally complex as it requires the use of Mitchell's algorithm for input data multiplication. Several approaches have been proposed to improve the accuracy of Mitchell's approximation, broadly categorized into shift-and-add-based [4–6], LUT-based [7], and interpolation-based [8] methods. These approaches divide the fraction of the logarithms into uniform or non-uniform regions and compute a corresponding correction term for each region.

In this work, we investigate Mitchell's algorithm and propose a unified hardware for computing multiplication, division, and square root operations—an approach that, to the best of our knowledge, has not been reported in the literature. Additionally, we propose a method for correcting results in the log-domain, significantly simplifying hardware design. Our contributions are twofold:

- We propose a unified and algebraically symmetrical hardware architecture capable of performing multiplication, division, and square root operations.
- We introduce a log-domain correction scheme that enhances the accuracy of these operations.

The remainder of this paper is organized as follows: Section 2 summarizes related work, Section 3 details the proposed design and presents hardware implementation results, and Section 4 concludes the paper.

2. Related Work

Figure 1 illustrates the typical block diagram of log-domain arithmetic operations. Logarithm and antilogarithm calculations are responsible for BNS-to-LNS and LNS-to-BNS conversions, respectively, forming the basis of the algebraically symmetrical characteristic in these systems. It is noted that existing hardware designs typically support only one type of operation—multiplication, division, or square root. Therefore, depending on the operation, a corresponding circuit (adder, subtractor, or shifter) processes the logarithms to yield the final result.

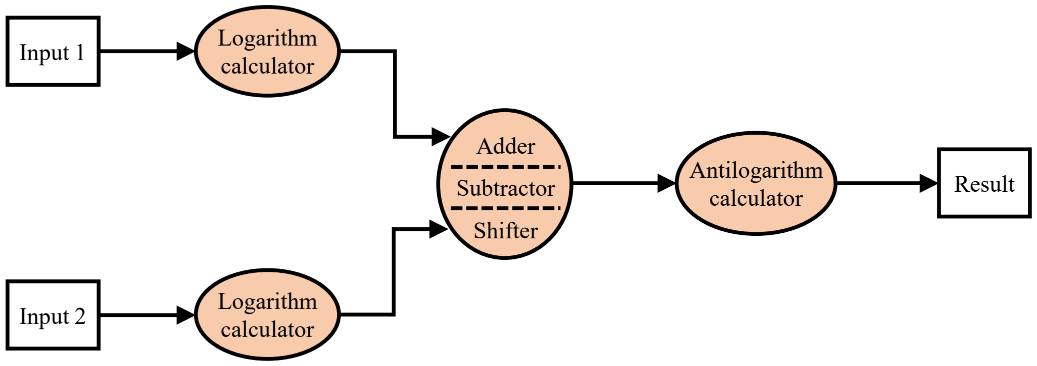

Figure 1. Block diagram of log-domain arithmetic operations.

Ahmed and Srinivas [9] utilized Mitchell's correction term to design an iterative multiplier. By observing that truncating the least significant bits of fractional parts can reduce the hardware complexity without significantly compromising precision, they developed a fractional predictor to facilitate the computation. Also based on Mitchell's correction term, Wu et al. [10] presented an approximate multiplier with a similar operating principle, iteratively compensating for the multiplication error. However, as mentioned earlier, the computation of Mitchell's correction term involves input data multiplication, which increases hardware size. Additionally, these two LNS multipliers are inefficient in terms of processing speed owing to their iterative nature, requiring multiple iterations to achieve a tolerable error level.

Joginipelly and Charalampidis [11] presented an LNS multiplier optimized for filtering applications. This work could be viewed as an improvement upon the previously discussed iterative multiplier. To derive the hardware architecture, the authors sought filter weights that minimize the mean square error between the approximate and true products. The use of fixed filter weights aids in achieving a low error without increasing hardware size. However, this approach also limits its applicability to general filtering applications.

Subhasri et al. [12] presented an LNS divider in which they sacrificed precision to reduce hardware complexities. More specifically, they devised an inexact subtractor, which is smaller than the standard subtractor as it ignores the carry bits from the least significant bits of the fractions. However, the reduction in hardware utilization is subtle, and the error slightly increases compared to Mitchell's algorithm.

More recently, Niu et al. [13] and Kim et al. [14] extended Mitchell's algorithm to floating-point numbers. They presented single-precision and double-precision floating-point multipliers and demonstrated their application in JPEG image compression and neural network inference. Similarly, Norris and Kim [15] implemented an iterative multiplier for single-precision floating-point numbers. They used histogram stretching to demonstrate the effectiveness of employing iterative multipliers instead of exact multipliers, showing a 13% reduction in processing time for a 352 $\times$ 288 video.

While much research has been conducted on multipliers, most studies focus on sacrificing precision to reduce power consumption and hardware complexity. Vakili et al. [16] proposed an approach that converts fixed-point inputs to floating-point format to preserve dynamic range. They utilized LUTs to approximate multiplication in floating-point format, with a decoder converting the multiplication result back to fixed-point format. Compared to the standard multiplier across four deep learning benchmarks, their approach achieved a 64% reduction in LUT utilization with a minimal accuracy loss of less than 0.29%.

Ahmad et al. [17] leveraged two-dimensional pseudo-Booth encoding to design floating-point pseudo-Booth and floating-point iterative pseudo-Booth multipliers. They also enhanced conventional iterative multipliers with a steering circuit to reduce power consumption. These two multipliers, compared to exact floating-point multipliers, exhibited 98.9% and 67.5% reductions in power consumption in TSMC 180 nm CMOS technology.

AMD Xilinx and Intel FPGA (Altera), the two largest FPGA manufacturers, currently provide arithmetic IP cores for multipliers, dividers, and square rooters. However, due to the insufficient documentation from Intel FPGA regarding the implementation of these IP cores [18], we focused on the solutions provided by AMD Xilinx [19,20]. Although the specific implementation details of their multipliers are not disclosed, the dividers utilize the radix-2 non-restoring division method, and the square rooters are limited to floating-point format. While using these IP cores can greatly reduce design time and accelerate time to market, they are not always optimal in terms of delay and area. In some cases, they may even become bottlenecks in the processing pipeline.

Interestingly, no studies on LNS square rooters have been reported in the literature. Existing research has primarily focused on developing LNS multipliers and dividers separately. In response to this gap, the following section introduces a unified hardware design featuring a multiplication-division-square root (MDS) adder capable of performing addition, subtraction, and shifting operations. This integration allows for the consolidation of all primary LNS arithmetic operations into a single design. Additionally, we derive correction terms for each operation and demonstrate that our proposed design substantially reduces error compared to Mitchell's algorithm.

3. Proposed Design

3.1. Mitchell's Algorithm-Based Logarithm Multiplication, Division, and Square Root

Let N be a binary number such that $2^j \leq N < 2^{k+1}$, where $j, k \in \mathbb{Z}$ and $j \leq k$. The binary representation of N is as follows:

$$N = n_k \dots n_3 n_2 n_1 n_0 . n_{-1} n_{-2} \dots n_j, \tag{1}$$

where $n_i \in \{0, 1\}$, n_k is the most significant bit, and n_j is the least significant bit. Without loss of generality, let us assume $n_k = 1$ and rewrite the number N as below:

$$N = \sum_{i=j}^{k} 2^i n_i = 2^k \left(1 + \sum_{i=j}^{k-1} 2^{i-k} n_i\right) = 2^k (1 + x), \tag{2}$$

where $x = \sum_{i=j}^{k-1} 2^{i-k} n_i$. As $j \leq k$, x is in the range $0 \leq x < 1$ and is called the fractional part (or fraction).

The logarithm of N is $lg(N) = k + lg(1 + x)$, where $lg(\cdot)$ denotes the base-2 logarithm. Mitchell [3] utilized a straight line to approximate $lg(1 + x)$ as $lg(1 + x) \approx x$, significantly reducing computational complexities. Compared with the true logarithm, Mitchell's approximation results in an error in the range $0 \leq R \leq 0.08639$.

The product of two numbers N_1 and N_2 can be expressed in the log-domain as follows:

$$\begin{aligned} lg(P) &= lg(N_1) + lg(N_2) = k_1 + lg(1 + x_1) + k_2 + lg(1 + x_2) && (3) \\ P &= 2^{k_1 + k_2} (1 + x_1)(1 + x_2). && (4) \end{aligned}$$

Using Mitchell's approximation, the product P can be approximated as:

$$lg(P') = k_1 + x_1 + k_2 + x_2. \tag{5}$$

Depending on whether a carry bit occurs when adding the fractional parts (x_1 and x_2), $lg(P')$ can be expressed as follows:

$$lg(P') = \begin{cases} k_1 + k_2 + (x_1 + x_2) & x_1 + x_2 < 1 \\ 1 + k_1 + k_2 + (x_1 + x + 2 - 1) & x_1 + x_2 \geq 1 \end{cases}. \tag{6}$$

According to Mitchell's approximation, $lg(1 + x) \approx x$ for $0 \leq x < 1$. Taking the antilogarithm yields $2^x \approx 1 + x$. As a result, the antilogarithm of Equation (6) is:

$$P' = \begin{cases} 2^{k_1 + k_2} (1 + x_1 + x_2) & x_1 + x_2 < 1 \\ 2^{1 + k_1 + k_2} (x_1 + x_2) & x_1 + x_2 \geq 1 \end{cases}. \tag{7}$$

The multiplication error E_m is defined as:

$$E_m = \frac{P' - P}{P} = \frac{P'}{P} - 1 = \begin{cases} \dfrac{1 + x_1 + x_2}{(1 + x_1)(1 + x_2)} - 1 & x_1 + x_2 < 1 \\ \dfrac{2(x_1 + x_2)}{(1 + x_1)(1 + x_2)} - 1 & x_1 + x_2 \geq 1 \end{cases}. \tag{8}$$

The quotient of two numbers N_1 and N_2 in the log-domain is the difference between $lg(N_1)$ and $lg(N_2)$:

$$\begin{aligned} lg(Q) &= k_1 + lg(1 + x_1) - k_2 - lg(1 + x_2) && (9) \\ Q &= \frac{2^{k_1 + k_2} (1 + x_1)}{1 + x_2}. && (10) \end{aligned}$$

The approximation of Q is:

$$lg(Q') = k_1 + x_1 - k_2 - x_2. \tag{11}$$

Similar to the multiplication case, there are two expressions of $lg(Q')$ depending on the presence or absence of a borrow from the integer part.

$$lg(Q') = \begin{cases} k_1 - k_2 + (x_1 - x_2) & x_1 - x_2 \geq 0 \\ k_1 - k_2 - 1 + (1 + x_1 - x_2) & x_1 - x_2 < 0 \end{cases} \tag{12}$$

$$Q' = \begin{cases} 2^{k_1-k_2}(1 + x_1 - x_2) & x_1 - x_2 \geq 0 \\ 2^{k_1-k_2-1}(2 + x_1 + x_2) & x_1 - x_2 < 0 \end{cases}. \tag{13}$$

The error E_d in division is defined as:

$$E_d = \frac{Q' - Q}{Q} = \frac{Q'}{Q} - 1 = \begin{cases} \dfrac{(1 + x_1 - x_2)(1 + x_2)}{1 + x_1} - 1 & x_1 - x_2 \geq 0 \\ \dfrac{(2 + x_1 - x_2)(1 + x_2)}{2(1 + x_1)} - 1 & x_1 - x_2 < 0 \end{cases}. \tag{14}$$

In the case of the square root, let N be the radicand. The logarithm of a square root is:

$$lg(S) = \frac{1}{2}[k + lg(1 + x)] \tag{15}$$

$$S = 2^{k/2}\sqrt{1 + x}. \tag{16}$$

The approximation is:

$$lg(S') = \frac{1}{2}(k + x) \tag{17}$$

$$S' = 2^{k/2}2^{x/2}. \tag{18}$$

The error E_s in the square root is defined as:

$$E_s = \frac{S' - S}{S} = \frac{S'}{S} - 1 = \frac{2^{x/2}}{\sqrt{1 + x}} - 1. \tag{19}$$

3.2. Error Analysis

It is evident from Equations (8), (14), and (19) that errors in multiplication, division, and square root operations stem from the fractional parts. First, consider the multiplication error in the case where $x_1 + x_2 < 1$, and let $a = x_1 + x_2$:

$$E_m(x_1 + x_2 < 1) = \frac{1 + a}{1 + a + x_2(a - x_2)} - 1. \tag{20}$$

Taking the partial derivative with respect to x_2:

$$\frac{\partial E_m(x_1 + x_2 < 1)}{\partial x_2} = \frac{-(1 + a)(a - 2x_2)}{[1 + a + x_2(a - x_2)]^2} = 0. \tag{21}$$

Solving this equation yields:

$$a = 2x_2 \tag{22}$$

$$x_1 = x_2. \tag{23}$$

The variable $a = x_1 + x_2$ is in the range $0 \leq a < 1$. At the two extremes, $a = 0$ and $a = 1$, the multiplication error is $E_m = 0$ and $E_m = -1/9$, respectively. The negative sign implies that the approximated product is always less than the true product. The error is maximum when $x_1 = x_2 = 1/2$, and the maximum error is $E_m = -1/9 \approx -11.1\%$.

Similarly, consider the case $x_1 + x_2 \geq 1$, with $a = x_1 + x_2$ now in the range $1 \leq a < 2$:

$$\begin{aligned} E_m(x_1 + x_2 \geq 1) &= \frac{2a}{1 + a + x_2(a - x_2)} - 1 && (24) \\ \frac{\partial E_m(x_1 + x_2 \geq 1)}{\partial x_2} &= \frac{-2a(a - 2x_2)}{[1 + a + x_2(a - x_2)]^2} = 0 && (25) \\ a &= 2x_2 && (26) \\ x_1 &= x_2. && (27) \end{aligned}$$

At the two extremes, $a = 1$ and $a = 2$, the multiplication error is $E_m = -1/9$ and $E_m = 0$, respectively. Similar to the previous case, the maximum error of -11.1% occurs when $x_1 = x_2 = 1/2$.

For the division error, we investigate two cases: $x_1 - x_2 \geq 0$ and $x_1 - x_2 < 0$. For $x_1 - x_2 \geq 0$:

$$E_d(x_1 - x_2 \geq 0) = \frac{(1 + x_1 - x_2)(1 + x_2)}{1 + x_1} - 1 = \frac{x_2(x_1 - x_2)}{1 + x_1}. \quad (28)$$

Given $0 \geq x_1, x_2 < 1$, the division error is maximized when x_1 approaches 1. Substituting $x_1 = 1$ into Equation (28) yields:

$$\begin{aligned} E_d(x_1 - x_2 \geq 0) &= \frac{x_2(1 - x_2)}{2} && (29) \\ \frac{\partial E_d(x_1 - x_2 \geq 0)}{\partial x_2} &= 1 - 2x_2 = 0 && (30) \\ x_2 &= 1/2. && (31) \end{aligned}$$

Here, the maximum error is $E_d = 1/8 = 12.5\%$ when $x_1 = 1$ and $x_2 = 1/2$. The minimum error is zero when $x1 = x_2$ or when $x_2 = 0$.

For $x_1 - x_2 < 0$:

$$E_d(x_1 - x_2 < 0) = \frac{(2 + x_1 - x_2)(1 + x_2)}{2(1 + x_1)} - 1 = \frac{(x_2 - x_1)(1 - x_2)}{2(1 + x_1)}. \quad (32)$$

Similarly, the error is maximized when $x_1 = 0$:

$$\begin{aligned} E_d(x_1 - x_2 < 0) &= \frac{x_2(1 - x_2)}{2} && (33) \\ \frac{\partial E_d(x_1 - x_2 < 0)}{\partial x_2} &= 1 - 2x_2 = 0 && (34) \\ x_2 &= 1/2. && (35) \end{aligned}$$

Thus, the maximum error $E_d = 1/8 = 12.5\%$ is achieved when $x_1 = 0$ and $x_2 = 1/2$. The minimum error is zero when $x_1 = x_2$ or when $x_2 = 1$.

The analysis of the square root error is as follows:

$$\begin{aligned} \frac{\partial E_s}{\partial x} &= \frac{[ln(2) \cdot (1 + x) - 1]2^{0.5x - 1}}{(1 + x)^{1.5}} = 0 && (36) \\ x &= \frac{1}{ln(2)} - 1 \approx 0.443. && (37) \end{aligned}$$

The maximum error is approximately $E_s \approx -0.029 = -2.9\%$. The minimum error is zero when $x = 0$ or when $x = 1$.

To summarize, computation errors can be as high as −11.1% for multiplication, 12.5% for division, and −2.9% for square root. The approximated product and square root are always less than the true product and square root, as indicated by the negative sign. In contrast, the approximated quotient is always greater than the true quotient. Therefore, in this work, we correct the errors using the following equations, inspired by the work of Mclaren [21]:

$$\begin{aligned} P_{correct} &= \frac{P'}{1-E_m} && (38) \\ Q_{correct} &= Q'(1-E_d) && (39) \\ S_{correct} &= \frac{S'}{1-E_s}. && (40) \end{aligned}$$

Notably, performing the above corrections in the log-domain is more computationally efficient, as the division and multiplication transform into addition and subtraction:

$$\begin{aligned} lg(P_{correct}) &= lg(P') + lg\left(\frac{1}{1-E_m}\right) && (41) \\ lg(Q_{correct}) &= lg(Q') - lg\left(\frac{1}{1-E_d}\right) && (42) \\ lg(S_{correct}) &= lg(S') + lg\left(\frac{1}{1-E_s}\right). && (43) \end{aligned}$$

The calculation of log-domain correction terms is challenging due to its reliance on division and logarithmic operations. Utilizing Michell's approximation can lead to significant errors, making it less desirable for accurate computations. In this study, we propose a method that involves partitioning the fractional parts into 2^M equally spaced regions using the M most significant bits (MSBs) of the fractions. For example, with three MSBs ($M = 3$), the fraction x is divided into eight ($2^3 = 8$) equally spaced intervals: $0.000 \to 0.125$, $0.125 \to 0.250$, $0.250 \to 0.375$, $0.375 \to 0.500$, $0.500 \to 0.625$, $0.625 \to 0.750$, $0.750 \to 0.875$, and $0.875 \to 1.000$.

For each specific region $x_i \leq x \leq x_{i+1}$, we compute the average correction term and store it in an LUT. Consequently, there are 2^{2M} correction terms for multiplication and division, and 2^M correction terms for square root operations. Tables 1–3 illustrate these correction terms for multiplication, division, and square root when $M = 3$. As shown in Table 1, the correction terms for multiplication form a symmetric matrix; therefore, the number of correction terms that need to be stored in the LUT can be further reduced to $2^{M-1}(2^M + 1)$. An example calculation involves dividing 15 by 3, demonstrating that the proposed correction term reduces the division error from 10% to 1.25%.

$$\begin{aligned} \text{Dividend} \quad N_1 &= 15 = 00001111_b \Rightarrow k_1 = 3 = 11_b,\ x_1 = 7 = 111_b \\ \text{Divisor} \quad N_2 &= 3 = 00000011_b \Rightarrow k_2 = 1 = 1_b,\ x_2 = 1 = 1_b \\ lg(N_1) &= 11.1110000_b \\ lg(N_2) &= 01.1000000_b \\ lg(N_1) - lg(N_2) &= 10.0110000_b \\ Q' &= 101.10000_b = 5.5 \Rightarrow 10\%\ \text{error} \\ \text{Correction term} \quad lg\left(\frac{1}{1-E_d}\right) &= 0.10743 = 00.0001110_b \\ lg(Q_{correct}) &= 110.0100010_b \\ Q_{correct} &= 101.00010_b = 5.0625 \Rightarrow 1.25\%\ \text{error}. \end{aligned}$$

Table 1. Correction terms for multiplication errors using the three most significant bits (MSBs) to partition the fractional parts.

$x_1 \backslash x_2$	0.000 ↓ 0.125	0.125 ↓ 0.250	0.250 ↓ 0.375	0.375 ↓ 0.500	0.500 ↓ 0.625	0.625 ↓ 0.750	0.750 ↓ 0.875	0.875 ↓ 1.000
0.000 → 0.125	0.00475	0.01304	0.01979	0.02540	0.03013	0.03417	0.03767	0.02669
0.125 → 0.250	0.01304	0.03592	0.05472	0.07042	0.08375	0.09520	0.09112	0.03307
0.250 → 0.375	0.01979	0.05472	0.08361	0.10792	0.12865	0.13252	0.08136	0.02522
0.375 → 0.500	0.02540	0.07042	0.10792	0.13963	0.15278	0.10957	0.06033	0.01879
0.500 → 0.625	0.03013	0.08375	0.12865	0.15278	0.11886	0.07757	0.04292	0.01342
0.625 → 0.750	0.03417	0.09520	0.13252	0.10957	0.07757	0.05087	0.02826	0.00886
0.750 → 0.875	0.03767	0.09112	0.08136	0.06033	0.04292	0.02826	0.01575	0.00495
0.875 → 1.000	0.02669	0.03307	0.02522	0.01879	0.01342	0.00886	0.00495	0.00156

Table 2. Correction terms for division errors using the three MSBs to partition the fractional parts.

$x_1 \backslash x_2$	0.000 ↓ 0.125	0.125 ↓ 0.250	0.250 ↓ 0.375	0.375 ↓ 0.500	0.500 ↓ 0.625	0.625 ↓ 0.750	0.750 ↓ 0.875	0.875 ↓ 1.000
0.000 → 0.125	0.01426	0.07090	0.12186	0.15092	0.15671	0.13896	0.09853	0.03727
0.125 → 0.250	0.00779	0.01413	0.05310	0.08792	0.10306	0.09790	0.07265	0.02834
0.250 → 0.375	0.01567	0.02431	0.01409	0.03891	0.06109	0.06552	0.05206	0.02118
0.375 → 0.500	0.02221	0.04624	0.03814	0.01410	0.02734	0.03934	0.03528	0.01530
0.500 → 0.625	0.02774	0.06492	0.07246	0.04990	0.01413	0.01772	0.02135	0.01038
0.625 → 0.750	0.03248	0.08104	0.10236	0.09521	0.06001	0.01417	0.00959	0.00622
0.750 → 0.875	0.03657	0.09508	0.12864	0.13545	0.11513	0.06880	0.01422	0.00264
0.875 → 1.000	0.04015	0.10743	0.15193	0.17144	0.16492	0.13273	0.07652	0.01428

Table 3. Correction terms for square root errors using the three MSBs to partition the fractional parts.

x	0.000 ↓ 0.125	0.125 ↓ 0.250	0.250 ↓ 0.375	0.375 ↓ 0.500	0.500 ↓ 0.625	0.625 ↓ 0.750	0.750 ↓ 0.875	0.875 ↓ 1.000
$lg[1\backslash(1-E_s)]$	0.01198	0.02983	0.03961	0.04280	0.04050	0.03356	0.02265	0.00829

Figure 2 illustrates the distribution of errors for multiplication, division, and square root operations using Mitchell's [3], Ha and Lee's [5], Kuo's [6] methods, and our proposed approach. The error distribution for Mitchell's algorithm shows a left-skewed pattern in multiplication, while division and square root errors exhibit a right-skewed distribution. The corresponding mean and standard deviation pairs are $(-0.038, 0.029)$, $(0.041, 0.032)$, and $(-0.020, 0.009)$, respectively. Ha and Lee [5], as well as Kuo [6], have refined Mitchell's approximation, leading to reduced errors in logarithm computations. However, their impact on log-domain arithmetic operations, depicted in Figure 2 and Table 4, shows modest improvement.

With the introduction of correction terms in our proposed method, the mean and standard deviation pairs improve to $(-0.001, 0.015)$ for multiplication, $(-0.004, 0.013)$ for division, and $(0.004, 0.004)$ for square root. These adjustments result in error reductions of 97.34% for multiplication, 90.24% for division, and 80% for square root. The error distributions now closely resemble a Gaussian shape, indicating that errors are predominantly centered around zero. This underscores a notable enhancement in accuracy compared to the methods of Mitchell [3], Ha and Lee [5], and Kuo [6].

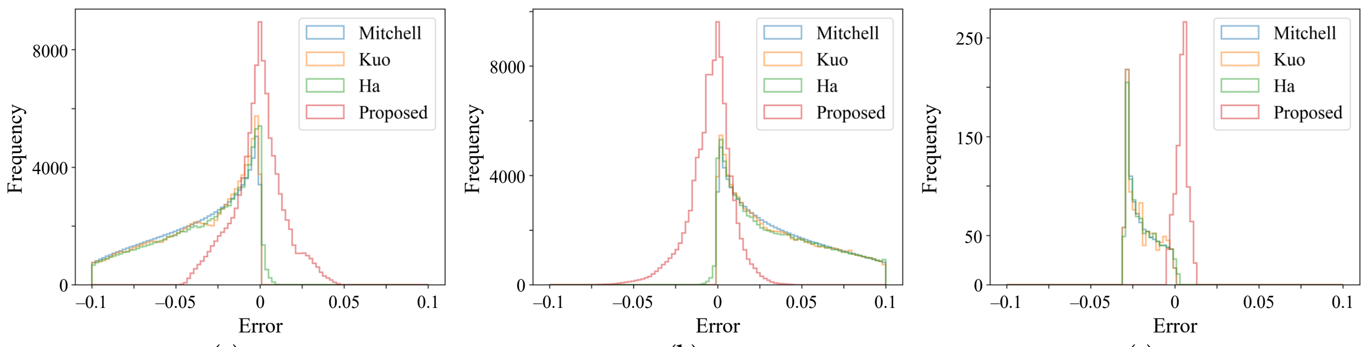

Figure 2. Distribution of multiplication, division, and square root errors using methods by Mitchell [3], Ha and Lee [5], Kuo [6], and the proposed method. (**a**) Multiplication error. (**b**) Division error. (**c**) Square root error.

Table 4. Summary statistics of multiplication, division, and square root errors using methods by Mitchell [3], Ha and Lee [5], Kuo [6], and the proposed method.

Operation	**Mitchell [3]**		**Ha and Lee [5]**		**Kuo [6]**		**Proposed Method**	
	Mean	**Std.**	**Mean**	**Std.**	**Mean**	**Std.**	**Mean.**	**Std.**
Multiplication	−0.038	0.029	−0.036	0.030	−0.037	0.030	−0.001	0.015
Division	0.041	0.032	0.041	0.034	0.041	0.033	−0.004	0.013
Square root	−0.020	0.009	−0.019	0.009	−0.019	0.009	0.004	0.004

Figure 3 illustrates the variation in error distributions for multiplication, division, and square root operations as the number of MSBs M varies. When only one bit is utilized, the error distributions remain skewed, resulting in minimal accuracy improvement. However, for $M \geq 3$, the errors exhibit a Gaussian-like distribution with the mean approaching zero, indicating a substantial enhancement in accuracy. Using larger values of M further improves accuracy but comes at the expense of an exponential increase in the size of the LUT, which is impractical for hardware implementation due to its exponential growth. Therefore, we opt to use $M = 3$ for our unified hardware design.

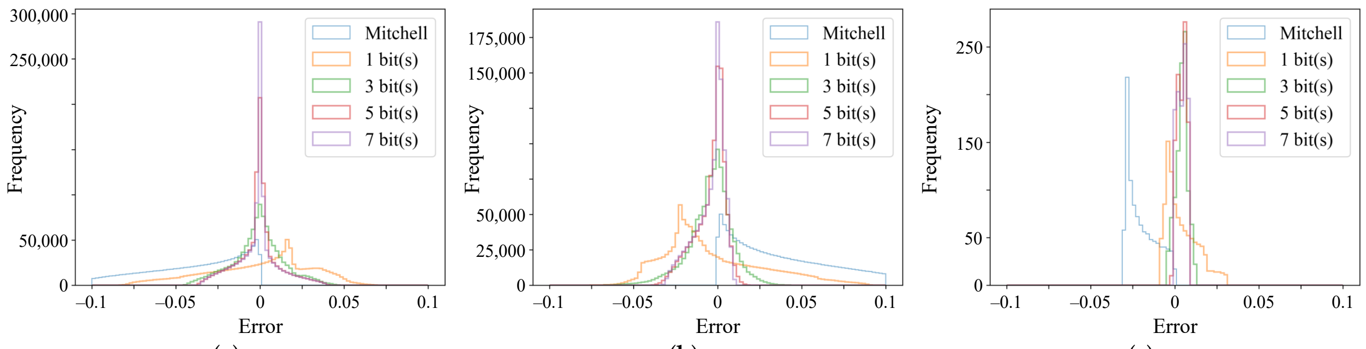

Figure 3. Distribution of multiplication, division, and square root errors using Mitchell's method [3] and the proposed method with M most significant bits, $M \in \{1, 3, 5, 7\}$. (**a**) Multiplication error. (**b**) Division error. (**c**) Square root error.

3.3. Unified Hardware Design

The proposed unified hardware follows a similar block diagram structure as depicted in Figure 1, thus retaining the inherent characteristic of algebraic symmetry. A key enhancement is the design of an MDS adder capable of executing addition, subtraction, and shifting operations, thereby establishing a unified architecture for multiplication, division, and square root computations. Figure 4 illustrates the detailed block diagram of our design, which accepts two inputs of L bits and produces an output of $2L$ bits to accommodate multiplication operations.

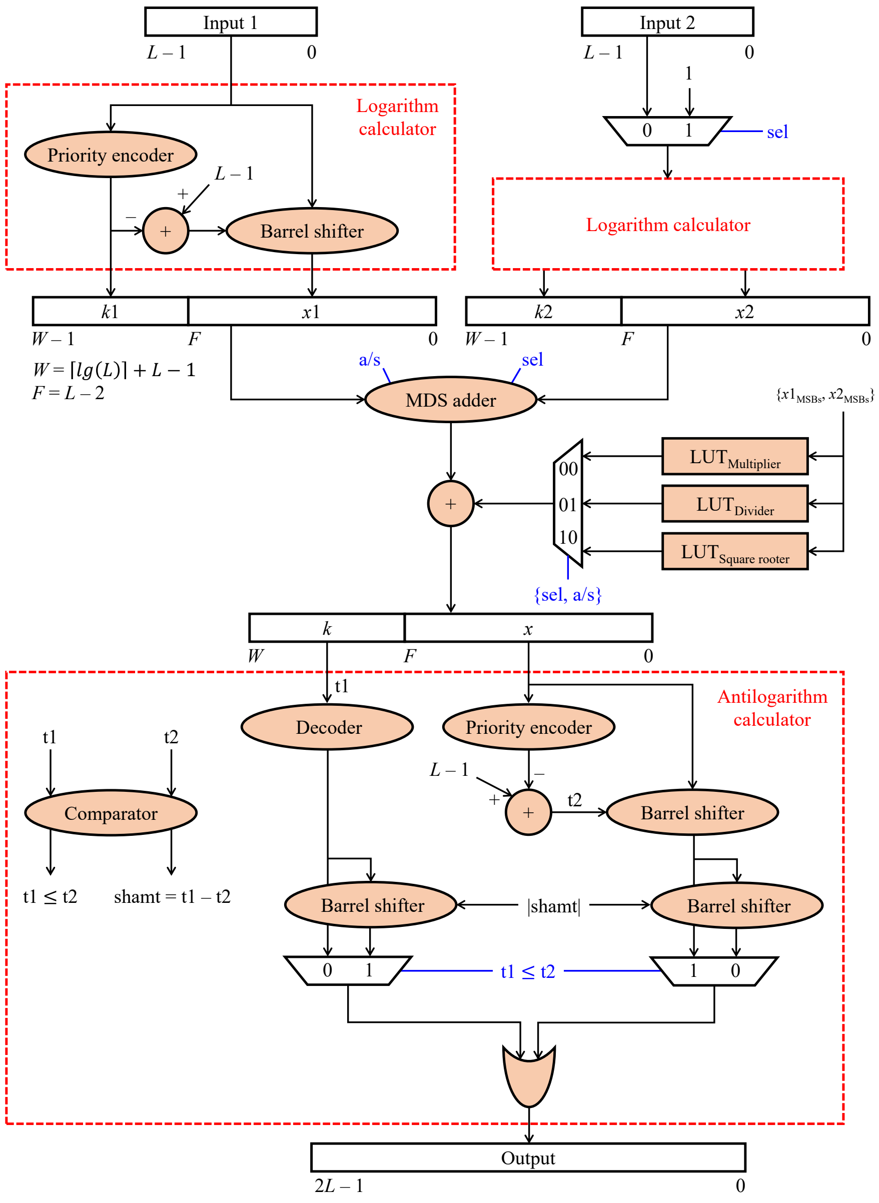

Figure 4. Block diagram of the proposed unified hardware design for multiplication, division, and square root.

The logarithm calculator comprises a priority encoder and a barrel shifter, implementing Mitchell's approximation method. The computed logarithm is stored in a W-bit register, where the integer part occupies $W-F$ bits and the fractional part consists of $F+1$ bits.

$$W = \lceil lg(L) \rceil + L - 1 \tag{44}$$
$$F = L - 2. \tag{45}$$

To specify which operation is to be performed, we employ a 2-bit command, which is further split into two 1-bit signals, denoted as sel and a/s. For binary operations, such as multiplication and division, sel $= 0$ routes the second input into the logarithm calculator. For a unary operation, like square root, sel $= 1$ selects a value of one, indicating the disregard of the second input. Regarding a/s, the MDS adder performs addition when a/s = 0; otherwise, it performs subtraction. Table 5 summarizes the operations of the proposed unified hardware design.

Table 5. Summary of operation modes of the proposed unified hardware design.

Command	sel	a/s	Operation	Description
00	0	0	Multiplication	$lg(N_1) + lg(N_2)$
01	0	1	Division	$lg(N_1) - lg(N_2)$
10	1	x	Square root	$0.5 \cdot lg(N_1) + lg(1)$
11	x	x	No operation	No description

In a naive implementation, adding or subtracting two logarithms would typically require at least two adders: one for negating the second operand in the case of subtraction, and another for adding two operands. However, in this study, we leverage the properties of two's complement number representation to enable a single adder to handle both addition and subtraction. Figure 5 depicts the block diagram of the proposed MDS adder, where B denotes the bit size of the input data.

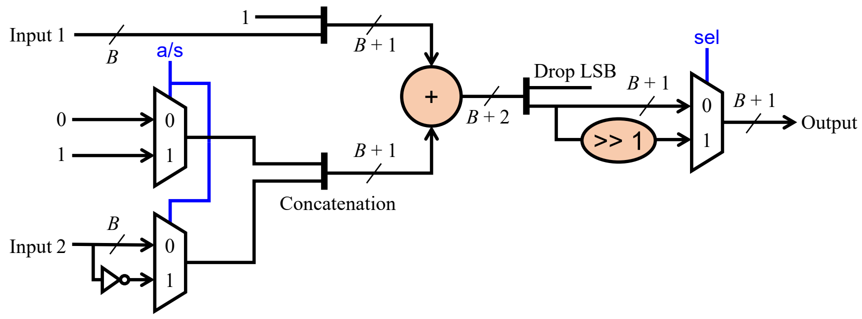

Figure 5. Block diagram of the proposed multiplication-division-square root (MDS) adder.

In two's complement representation, the subtraction $a - b$ is equivalent to $a + \bar{b} + 1$, where $\bar{b}$ is the bitwise inversion of b. In the proposed MDS adder, we pad the first input a with a 1 bit in the least significant bit (LSB) position. Similarly, after inverting the second input b, we pad $\bar{b}$ with a 1 bit in the LSB position. Adding these two padded inputs ensures that a carry-in of 1 is generated, resulting in $a + \bar{b} + 1$ as the output of the adder. For example, considering subtracting two 4-bit numbers $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$:

$$\begin{aligned} \text{Padding } a_3a_2a_1a_0 &\rightarrow a_3a_2a_1a_0 1 \\ \text{Inverting and padding } b_3b_2b_1b_0 &\rightarrow \bar{b}_3\bar{b}_2\bar{b}_1\bar{b}_0 1 \\ \text{Performing addition } c_4c_3c_2c_1c_0 &= a_3a_2a_1a_0 1 + \bar{b}_3\bar{b}_2\bar{b}_1\bar{b}_0 1 \\ \text{Result } &= c_4c_3c_2c_1. \end{aligned}$$

Another example involves adding two 4-bit numbers $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$:

$$\begin{aligned} \text{Padding } a_3a_2a_1a_0 &\rightarrow a_3a_2a_1a_0 1 \\ \text{Padding } b_3b_2b_1b_0 &\rightarrow b_3b_2b_1b_0 0 \\ \text{Performing addition } c_4c_3c_2c_1c_0 &= a_3a_2a_1a_0 1 + b_3b_2b_1b_0 0 \\ \text{Result } &= c_4c_3c_2c_1. \end{aligned}$$

In the first example, adding 1 and 1 at the LSB position generates a carry-in of 1, implying that $c_4c_3c_2c_1 = a_3a_2a_1a_0 + \bar{b}_3\bar{b}_2\bar{b}_1\bar{b}_0 + 1$. In the second example, adding 1 and 0 at the LSB position generates a carry-in of 0, implying that $c_4c_3c_2c_1 = a_3a_2a_1a_0 + b_3b_2b_1b_0$. Therefore, the proposed MDS adder is capable of performing both addition and subtraction. In the case of square root operations, the output of the adder will be shifted to the right by one position.

Next, the output of the MDS adder is combined with a correction term obtained from one of three LUTs for multiplication, division, and square root operations, respectively. The three MSBs of the fractional parts x_1 and x_2 serve as an address for accessing the LUTs. Subsequently, the result is processed by the antilogarithm calculator. In the left branch, a decoder and a barrel shifter determine the position of the most significant 1 bit. On the right branch, a priority encoder and two barrel shifters adjust the fraction to ensure that the bits are correctly positioned. The shift amount shamt specifies the position of the binary point, separating the integer and fractional parts. In the antilogarithm calculator, the signals t_1 and t_2 are used to represent intermediate results.

3.4. Implementation Results

We utilized Verilog HDL [22] (IEEE Standard 1364-2005) to implement the proposed unified hardware design and Xilinx Vivado v2024.1 [23] to obtain the implementation results. The target FPGA device is XCZU7EV-2FFVC1156 on a Zynq UltraScale+ MP-SoC ZCU106 Evaluation Kit [24]. This device comprises a processing system (PS) and programmable logic (PL) within the same unit. The PS features an Arm® Cortex®-A53 quad-core processor, a Cortex-R5F dual-core real-time processor, and a Mali-400 graphics processing unit. The PL includes abundant configurable logic blocks (460,800 registers, 230,400 LUTs, 11Mb block RAM, 27 Mb UltraRAM, and 1728 DSP slices) and a video encoder/decoder unit, making it well-suited for high-performance computing applications.

In Section 3.2, we analyzed errors in logarithm-based multiplication, division, and square root operations, discovering that these errors originated from the approximation of $lg(1+x)$ in Equation (2). Methods proposed by Ha and Lee [5] and Kuo [6] improve the approximation of $lg(1+x)$, thereby reducing the errors in multiplication, division, and square root operations. Also in Section 3.2, we presented a method for reducing these errors and demonstrated its superiority over the methods by Mitchell [3], Ha and Lee [5], and Kuo [6]. Notably, our proposed unified design is the first capable of performing multiplication, division, and square root operations via a single hardware architecture, which is a definite advantage but complicates direct comparisons. Therefore, we compare our proposed design against the standard multiplier, divider, and square rooter widely utilized in the industry.

Table 6 summarizes the implementation results for 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit operands. The multiplier is designed based on the split mechanism detailed in [25], while the divider and square rooter are designed following the pipeline parallelism described in [26]. Although Xilinx provides DSP macros that can synthesize multipliers, dividers, and squarers (not square rooters), these computing resources should be reserved for applications requiring intensive computations, such as neural network inference. Additionally, utilizing slice logic (registers and LUTs) instead of DSP macros not only facilitates a straightforward comparison between the proposed design and standard ones but also results in a more optimal implementation. We also included the Xilinx divider IP core in the comparison, with resource utilization information for this divider on the same Zynq UltraScale+ family obtained from [19].

First, it is noteworthy that all designs in Table 6 follow pipeline parallelism. Therefore, the latency is the number of clock cycles required to fill the pipeline, after which new results are produced in every clock cycle. While the multiplier's latency is two clock cycles thanks to the split mechanism, the latencies of the divider and square rooter depend on the bit size used to represent the result. To conduct a fair comparison, we use the same output bit size for all designs: specifically, 8, 16, 32, 64, and 128 bits to represent the output of 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit operations, respectively. Consequently, the latencies of the divider and square rooter can be explained as follows: one clock cycle for clocking input data and 8/16/32/64/128 clock cycles for 4-bit/8-bit/16-bit/32-bit/64-bit division and square root operations, resulting in latencies of 9/17/33/65/129 clock cycles for these two designs, as well as AMD Xilinx's divider. In sharp contrast, our proposed design only requires six clock cycles, regardless of whether the operation is multiplication, division, or square root. This efficiency is attributed to the use of logarithms, which transforms multiplication, division, and square root operations into addition, subtraction, and shift operations, which can be executed in a single clock cycle.

Table 6. Hardware implementation results of the standard multiplier, divider, square rooter, and the proposed unified design. NA stands for not available.

		Registers ↓ (#)	LUTs ↓ (#)	Latency ↓ (# Clock Cycles)	Power Consumption ↓ (W)	Maximum Frequency ↑ (MHz)
4-bit	Multiplier [25]	16	20	2	0.619	1272.265
	Divider [26]	97	61	9	0.634	1053.741
	AMD Xilinx's divider [19]	NA	NA	NA	NA	NA
	Square rooter [26]	76	75	9	0.644	964.320
	Proposed design	71	78	6	0.642	1005.025
8-bit	Multiplier [25]	32	80	2	0.659	603.136
	Divider [26]	343	308	17	0.704	834.028
	AMD Xilinx's divider [19]	262	124	17	NA	636.000
	Square rooter [26]	317	341	17	0.732	865.052
	Proposed design	127	198	6	0.675	651.042
16-bit	Multiplier [25]	64	323	2	0.793	373.413
	Divider [26]	1210	1131	33	0.999	776.398
	AMD Xilinx's divider [19]	NA	NA	NA	NA	NA
	Square rooter [26]	1273	1396	33	1.215	786.782
	Proposed design	234	473	6	0.838	591.716
32-bit	Multiplier [25]	175	1303	2	0.915	250.000
	Divider [26]	4444	4359	65	1.399	644.330
	AMD Xilinx's divider [19]	3334	1279	65	NA	604.000
	Square rooter [26]	5152	5935	65	1.971	604.230
	Proposed design	431	1121	6	0.993	506.842
64-bit	Multiplier [25]	440	5225	2	1.175	188.679
	Divider [26]	17,038	17,099	129	2.998	533.903
	AMD Xilinx's divider [19]	NA	NA	NA	NA	NA
	Square rooter [26]	29,429	24,542	129	4.094	385.208
	Proposed design	883	2394	6	1.415	500.000

Regarding slice logic utilization, it is evident from Table 6 that the multiplier occupies the least amount of slice logic and consumes the least power because multiplication is a relatively simple operation. However, as the operand bit size increases, the maximum frequency decreases sharply, from 1272.265 MHz for 4-bit multiplication to 188.679 MHz for 64-bit multiplication. Division and square root operations are more complex and thus require a substantial amount of slice logic for implementation. These two designs consume more power than the other designs but operate faster. Except for the simple multiplier, our proposed design is superior to the divider and square rooter in terms of slice logic utilization and power consumption. Although it is inferior to these two in terms of maximum frequency, it is noteworthy that the maximum frequency attained by our proposed design is adequate for most high-performance computing applications.

Overall, the implementation results demonstrate the efficacy and necessity of our proposed unified design.

4. Conclusions

In this paper, we presented a unified and algebraically symmetrical hardware design capable of performing multiplication, division, and square root operations by leveraging the properties of binary logarithms. To address the errors caused by approximations in logarithm calculations, we proposed the use of correction terms, which resulted in significant improvements in accuracy. We implemented the proposed unified design and compared it with standard multipliers, dividers, and square rooters. The implementation results demonstrated that our design is more efficient in terms of slice logic utilization and power consumption while maintaining operation at an acceptably high frequency, making it highly suitable for high-performance DSP applications.

While the proposed correction terms were calculated as average values over specific intervals, it is possible to further reduce computation errors by narrowing the interval or even using point-wise correction terms. However, this approach poses a significant challenge due to the substantial number of LUTs required for storage. Another direction for future work is to refine the approximation of $lg(1+x)$ using polynomial or piece-wise linear regression. The regression model could provide a rough initial estimate of the logarithm, which can then be refined using an iterative method like Newton–Raphson. In future work, we will explore all these potential directions and seek a most computationally efficient way to further reduce computation errors.

Author Contributions: Conceptualization, B.K.; software, D.N.; validation, D.N.; data curation, S.H.; writing—original draft preparation, D.N.; writing—review and editing, D.N., S.H. and B.K.; supervision, B.K. All authors have read and agreed to the published version of the manuscript.

Funding: This was supported by Korea National University of Transportation Industry-Academy Cooperation Foundation in 2024.

Data Availability Statement: Dataset available on request from the authors.

Acknowledgments: The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

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

References

1. Arnold, M.G.; Collange, C. A Real/Complex Logarithmic Number System ALU. *IEEE Trans. Comput.* **2011**, *60*, 202–213. [CrossRef]
2. Chaudhary, M.; Lee, P. An Improved Two-Step Binary Logarithmic Converter for FPGAs. *IEEE Trans. Circuits Syst. II Express Briefs* **2015**, *62*, 476–480. [CrossRef]
3. Mitchell, J.N. Computer Multiplication and Division Using Binary Logarithms. *IRE Trans. Electron. Comput.* **1962**, *EC-11*, 512–517. [CrossRef]
4. Kuo, C.; Juang, T. Design of fast logarithmic converters with high accuracy for digital camera application. *Microsyst. Technol.* **2018**, *24*, 9–17. [CrossRef]
5. Ha, M.; Lee, S. Accurate Hardware-Efficient Logarithm Circuit. *IEEE Trans. Circuits Syst. II Express Briefs* **2017**, *64*, 967–971. [CrossRef]
6. Kuo, C. Design and realization of high performance logarithmic converters using non-uniform multi-regions constant adder correction schemes. *Microsyst. Technol.* **2018**, *24*, 4237–4245. [CrossRef]
7. Jana, B.; Roy, A.S.; Saha, G.; Banerjee, S. A Low-Error, Memory-Based Fast Binary Logarithmic Converter. *IEEE Trans. Circuits Syst. II Express Briefs* **2020**, *67*, 2129–2133. [CrossRef]
8. Makimoto, R.; Imagawa, T.; Ochi, H. Approximate Logarithmic Multipliers Using Half Compensation with Two Line Segments. In Proceedings of the 2023 IEEE 36th International System-on-Chip Conference (SOCC), Santa Clara, CA, USA, 5–8 September 2023; pp. 1–6. [CrossRef]
9. Ahmed, S.; Srinivas, M. An Improved Logarithmic Multiplier for Media Processing. *J. Signal Process. Syst.* **2019**, *91*, 561–574. [CrossRef]
10. Wu, X.; Wei, Z.; Ko, S.B.; Zhang, H. Design of Energy Efficient Logarithmic Approximate Multiplier. In Proceedings of the 2023 5th International Conference on Circuits and Systems (ICCS), Huzhou, China, 27–30 October 2023; pp. 129–134. [CrossRef]
11. Joginipelly, A.; Charalampidis, D. An efficient circuit for error reduction in logarithmic multiplication for filtering applications. *Int. J. Circuit Theory Appl.* **2020**, *48*, 809–815. [CrossRef]
12. Subhasri, C.; Jammu, B.; Harsha, L.; Bodasingi, N.; Samoju, V. Hardware-efficient approximate logarithmic division with improved accuracy. *Int. J. Circuit Theory Appl.* **2020**, *49*, 128–141. [CrossRef]
13. Niu, Z.; Zhang, T.; Jiang, H.; Cockburn, B.F.; Liu, L.; Han, J. Hardware-Efficient Logarithmic Floating-Point Multipliers for Error-Tolerant Applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2024**, *71*, 209–222. [CrossRef]
14. Kim, S.; Norris, C.J.; Oelund, J.I.; Rutenbar, R.A. Area-Efficient Iterative Logarithmic Approximate Multipliers for IEEE 754 and Posit Numbers. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2024**, *32*, 455–467. [CrossRef]

15. Norris, C.J.; Kim, S. A Use Case of Iterative Logarithmic Floating-Point Multipliers: Accelerating Histogram Stretching on Programmable SoC. In Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 21–25 May 2023; pp. 1–5. [CrossRef]
16. Vakili, S.; Vaziri, M.; Zarei, A.; Langlois, J.P. DyRecMul: Fast and Low-Cost Approximate Multiplier for FPGAs using Dynamic Reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* **2024**. [CrossRef]
17. Towhidy, A.; Omidi, R.; Mohammadi, K. On the Design of Iterative Approximate Floating-Point Multipliers. *IEEE Trans. Comput.* **2023**, *72*, 1623–1635. [CrossRef]
18. Intel. Integer Arithmetic Intel FPGA IP Cores User Guide. Available online: https://www.intel.com/content/www/us/en/docs/programmable/683490/24-1/integer-arithmetic-cores.html (accessed on 26 August 2024).
19. Xilinx. Divider Generator v5.1 Product Guide (PG151). Available online: https://docs.amd.com/v/u/en-US/pg151-div-gen (accessed on 26 August 2024).
20. Xilinx. Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator (UG958). Available online: https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref (accessed on 26 August 2024).
21. Mclaren, D. Improved Mitchell-based logarithmic multiplier for low-power DSP applications. In Proceedings of the 2003 IEEE International Systems-on-Chip SOC Conference, Portland, OR, USA, 17–20 September 2003; pp. 53–56. [CrossRef]
22. *IEEE Std 1364-2005 (Revision of IEEE Std 1374-2001)*; IEEE Standard for Verilog Hardware Description Language; IEEE: Piscataway, NJ, USA, 2006; [CrossRef]
23. Xilinx. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973). Available online: https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license/Release-Notes (accessed on 21 April 2024).
24. Xilinx. ZCU106 Evaluation Board: User Guide (UG1244). Available online: https://docs.xilinx.com/v/u/en-US/ug1244-zcu106-eval-bd (accessed on 25 July 2023).
25. Ngo, D.; Kang, B. Taylor-Series-Based Reconfigurability of Gamma Correction in Hardware Designs. *Electronics* **2021**, *10*, 1959. [CrossRef]
26. Lee, S.; Ngo, D.; Kang, B. Design of an FPGA-Based High-Quality Real-Time Autonomous Dehazing System. *Remote Sens.* **2022**, *14*, 1852. [CrossRef]