electronics

MDPI

Article

Accelerating Pattern Recognition with a High-Precision Hardware Divider Using Binary Logarithms and Regional Error Corrections

Dat Ngo [1,†], Suhun Ahn [2,†], Jeonghyeon Son [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 Electronics Engineering, Dong-A University, Busan 49315, Republic of Korea; yusu@donga.ac.kr (S.A.); 1922913@donga.ac.kr (J.S.)
* Correspondence: bongsoon@dau.ac.kr; Tel.: +82-51-200-7703
† These authors contributed equally to this work.

Abstract: Pattern recognition applications involve extensive arithmetic operations, including additions, multiplications, and divisions. When implemented on resource-constrained edge devices, these operations demand dedicated hardware, with division being the most complex. Conventional hardware dividers, however, incur substantial overhead in terms of resource consumption and latency. To address these limitations, we employ binary logarithms with regional error correction to approximate division operations. By leveraging approximation errors at boundary regions to formulate logarithm and antilogarithm offsets, our approach effectively reduces hardware complexity while minimizing the inherent errors of binary logarithm-based division. Additionally, we propose a six-stage pipelined hardware architecture, synthesized and validated on a Zynq UltraScale+ FPGA platform. The implementation results demonstrate that the proposed divider outperforms conventional division methods in terms of resource utilization and power savings. Furthermore, its application in image dehazing and object detection highlights its potential for real-time, high-performance computing systems.

Keywords: high-precision divider; binary logarithm; regional error correction

Academic Editor: George A. Tsihrintzis

Received: 31 January 2025
Revised: 28 February 2025
Accepted: 5 March 2025
Published: 7 March 2025

Citation: Ngo, D.; Ahn, S.; Son, J.; Kang, B. Accelerating Pattern Recognition with a High-Precision Hardware Divider Using Binary Logarithms and Regional Error Corrections. *Electronics* **2025**, *14*, 1066. https://doi.org/10.3390/electronics14061066

1. Introduction

Deep neural networks (DNNs) have become fundamental in pattern recognition applications due to their ability to model complex data distributions and achieve state-of-the-art performance in tasks such as image classification, speech recognition, and natural language processing [1–3]. By leveraging hierarchical feature extraction and deep architectures, DNNs effectively capture intricate patterns and relationships within data. However, this high accuracy comes at the cost of substantial computational complexity. The large number of parameters, layers, and operations required for training and inference demands considerable computational resources, often relying on high-performance hardware such as GPUs and TPUs. Among these operations, multiplication and division play a critical role in matrix computations and activation functions, making their efficient execution essential for achieving optimal performance. This necessity highlights the importance of hardware accelerators designed to optimize these operations, particularly for resource-constrained platforms such as mobile devices and embedded systems, where power efficiency is a key concern.

The computational demands of multiplication and division vary significantly across hardware architectures. While multiplication is relatively efficient due to specialized hardware units in both CPUs and GPUs, division remains a computationally expensive operation. Unlike multiplication, division lacks dedicated hardware support in most processing units [4,5] and typically relies on iterative approximation methods such as Newton–Raphson or Goldschmidt iterations [6]. As a result, division operations are often emulated using a combination of multiplications, shifts, and additions, leading to increased latency and power consumption. On GPUs, where parallelism is optimized for massively concurrent computations, the absence of dedicated division units further exacerbates performance bottlenecks in applications requiring frequent division operations, such as deep learning and numerical simulations. This inefficiency underscores the need for optimized division algorithms and hardware accelerators to enhance performance and energy efficiency in such platforms.

Binary logarithm-based division offers a promising alternative by transforming division into subtraction through logarithmic and exponential relationships, significantly reducing computational complexity. While early approaches, such as Mitchell's algorithm [7], introduce approximation errors of around 12.5%, advancements in error correction techniques have demonstrated the potential to reduce this error to below 1%, making logarithm-based division a viable alternative for many applications. This method is particularly beneficial in hardware-constrained environments, where traditional division is costly in terms of latency and power consumption. By refining approximation techniques and integrating error correction mechanisms, binary logarithm-based division can provide a balance between computational efficiency and precision, making it suitable for performance-critical and energy-efficient systems.

In this work, we propose an efficient binary logarithm-based division that enhances computational accuracy while maintaining hardware efficiency. To address the inherent approximation errors, we introduce a regional error correction mechanism that adjusts results based on input-specific characteristics. This refinement significantly improves the accuracy of division while preserving its computational advantages. Additionally, we present a six-stage pipelined hardware architecture that was synthesized and validated on a Zynq UltraScale+ FPGA platform. We further demonstrate the effectiveness of the proposed divider in image dehazing and object detection applications, achieving substantial reductions in hardware utilization and power consumption while maintaining high performance.

2. Related Work

2.1. Digit Recurrence Division Methods

Digit recurrence division is a widely used algorithm, particularly suited for implementation on resource-constrained edge devices. It follows the paper-and-pencil division method, where the dividend is processed digit-by-digit (or bit-by-bit) from left to right, producing the corresponding quotient digit (or bit). Mathematically, division can be expressed as follows:

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder}, \tag{1}$$

where $0 \leq \text{Remainder} \leq \text{Divisor}$.

The implementation of digit recurrence division involves addition, shifting, and multiplication, making it well-suited for commercial applications. The common types of dividers based on this algorithm include restoring, non-restoring [8], and SRT [9] (named after its inventors).

2.1.1. Restoring Division

Let N_1, N_2, Q, and P represent the dividend, divisor, quotient, and remainder, respectively. Assume all numbers are binary, with Q having a wordlength of w. Each quotient bit is denoted as q_i, where $i = w-1, w-2, ..., 1, 0$. The partial remainder corresponding to q_i is denoted as $P(i)$. The restoring division process begins by initializing $P(w) = N_1$ and determining q_{w-1} using the following:

$$P(w-1) = P(w) - q_{w-1} \cdot N_2 \cdot 2^{w-1}. \tag{2}$$

If $P(w-1) \geq 0$, then $q_{w-1} = 1$; otherwise, $q_{w-1} = 0$ and $N_2 \cdot 2^{w-1}$ is added back to restore the remainder. This restoration step ensures $P(w-1) \geq 0$ before proceeding to the next quotient bit q_{w-2}. The general recurrence relation is as follows:

$$P(i) = P(i+1) - q_i \cdot N_2 \cdot 2^i. \tag{3}$$

Subtraction continues until the partial remainder becomes negative, requiring restoration before computing the next quotient bit. Algorithm 1 summarizes the restoring division approach, while two hardware architectures (serial and pipelined) are presented in [10].

Algorithm 1 Restoring Division

Input: Dividend N_1 and divisor N_2
Output: Quotient Q and remainder P
Begin
1: $P = N_1$, $N_2 = N_2 \cdot 2^w$
2: for $i = w-1 \rightarrow 0$ do
3: $P = 2P - N_2$
4: if $P \geq 0$ then
5: $q_i = 1$
6: else
7: $q_i = 0$
8: $P = P + N_2$
9: end if
10: end for
End

2.1.2. Non-Restoring Division

Restoring division may require up to $2w$ clock cycles to compute all quotient bits: w cycles for subtractions and up to w additional cycles for restorations. The non-restoring division method eliminates the need for restoration by performing a single decision and addition (or subtraction) per quotient bit, as summarized in Algorithm 2.

This method produces a quotient and remainder in a non-standard form, requiring an additional conversion step. Xilinx, the leading FPGA manufacturer, offers two versions of non-restoring dividers: Radix-2 and High-Radix [11]. The Radix-2 divider is recommended for integer operands with wordlengths below 16 bits, while the High-Radix divider incorporates prescaling, making it suitable for operands exceeding 16 bits.

Algorithm 2 Non-Restoring Division

Input: Dividend N_1 and divisor N_2
Output: Quotient Q and remainder P
Begin
$P = N_1$, $N_2 = N_2 \cdot 2^w$
for $i = w - 1 \rightarrow 0$ do
  if $P \geq 0$ then
    $q_i = 1$
    $P = 2P - N_2$
  else
    $q_i = 0$
    $P = 2P + N_2$
  end if
end for
$Q = Q - \bar{Q}$
if $P < 0$ then
  $Q = Q - 1$
  $P = P + N_2$
end if
End

2.1.3. SRT Division

Using the same notation as before, let r be the radix, typically chosen as a power of two. The SRT division method follows the following recurrence:

$$rP(w) = N_1, \quad (4)$$

$$P(i+1) = rP(i) - q_{i+1}N_2. \quad (5)$$

At each iteration, one quotient digit is determined using the following selection function:

$$q_{i+1} = \text{SEL}(rP(i), N_2). \quad (6)$$

The quotient digit is chosen such that the next partial remainder satisfies $|P(i+1)| < N_2$. The complexity of SEL$(\cdot)$ depends on the radix, redundancy, and wordlength of divisor and remainder estimates. Interested readers are referred to [12] for a detailed analysis. Each iteration comprises three steps:

- Selecting the next quotient digit q_{i+1}.
- Computing the product $q_{i+1}N_2$.
- Updating the remainder: $P(i+1) = rP(i) - q_{i+1}N_2$.

2.2. Functional Iteration Division Methods

Unlike digit recurrence methods, which compute one quotient digit per iteration, functional iteration methods estimate the quotient directly, allowing multiple digits to be computed per iteration. This approach relies on multiplication instead of subtraction, reducing latency at the cost of precision.

2.2.1. Newton–Raphson Division

This method [13] estimates the divisor's reciprocal and multiplies it by the dividend:

- Computing an initial estimate X_0 of 1 / N_2.
- Refining the estimate iteratively: $X_1, X_2, ..., X_f$.
- Computing the quotient: $Q = N_1 X_f$.

In order to refine the estimate iteratively, it is essential to find a function $g(X)$ that has a zero at $X = 1 / N_2$. One of such function is $g(X) = (1 / X) - N_2$, for which the Newton–Raphson iteration can be applied:

$$X_{i+1} = X_i - \frac{g(X_i)}{g'(X_i)} = X_i(2 - N_2 X_i). \tag{7}$$

Despite initial slow convergence, the method exhibits quadratic convergence, approximately doubling the number of correct digits in each iteration. Xilinx implements this method in its LutMultA divider [11].

2.2.2. Goldschmidt Division

Goldschmidt division [14] iteratively multiplies both the dividend and divisor by a common factor F_i, expressed as follows:

$$Q = \frac{N_1}{N_2}\frac{F_0}{F_0}\frac{F_1}{F_1}\ldots\frac{F_f}{F_f}, \tag{8}$$

where the factor is chosen to drive N_2 toward 1, thereby transforming N_1 into the final quotient. Unlike Newton–Raphson, this method allows parallel multiplication, leading to its adoption in AMD Athlon processors [15].

2.3. Logarithm-Based Division Methods

Figure 1 presents a typical block diagram for division using binary logarithms, where the red-dashed blocks correspond to logarithm and antilogarithm computations. These computations introduce errors into the quotient, with the primary source of error stemming from the inverse relationship between the two operations.

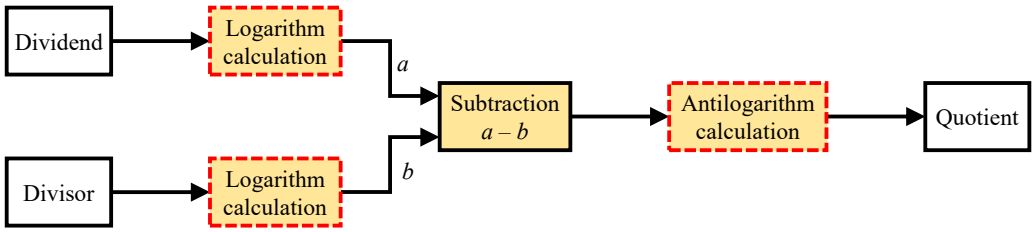


Figure 1. Block diagram of binary logarithm-based division. The red-dashed blocks require approximation techniques that introduce errors into the quotient.

For a given binary number N represented as follows:

$$N = n_k \ldots n_3 n_2 n_1 n_0 . n_{-1} n_{-2} \ldots n_j = \sum_{i=j}^{k} 2^i n_i, \tag{9}$$

where n_k and n_j represent the most significant and least significant bits, respectively. Each n_i is either 0 or 1. Without loss of generality, we assume $n_k = 1$, allowing N to be rewritten as follows:

$$N = 2^k \left(1 + \sum_{i=j}^{k-1} 2^{i-k} n_i\right) = 2^k (1 + x), \tag{10}$$

where $x = \sum_{i=j}^{k-1} 2^{i-k} n_i$ is the fractional part, constrained by $0 \leq x < 1$. The binary logarithm of N can then be expressed as $\log_2(N) = k + \log_2(1 + x)$. Thus, computing $\log_2(N)$ involves the following:

- Determining the index k of the most significant nonzero bit n_K.
- Computing $\log_2(1 + x)$, where x represents the fractional component of N.

Approximating $\log_2(1+x)$ is computationally simpler than directly approximating $\log_2(N)$. Common methods include the look-up table (LUT) approach [16,17] and Taylor series expansion [18]. The LUT method precomputes logarithm values and stores them for quick retrieval, while the Taylor series method approximates the logarithm through an infinite sum, truncated at a desired accuracy level. However, both approaches exhibit a trade-off between precision and computational complexity, making them less suitable for high-accuracy applications.

2.3.1. Mitchell's Algorithm

Mitchell's algorithm [7] simplifies logarithm computation by approximating $\log_2(1+x)$ as a linear function $\log_2(1+x) \approx x$. This leads to an approximate logarithm $\log_2(N') = k + x$, introducing an approximation error defined as $R = \log_2(1+x) - x$, which lies within the range $[0, 0.08639]$. This error propagates into division operations using binary logarithms.

For two numbers N_1 and N_2, the quotient in logarithmic form is as follows:

$$\begin{aligned} \log_2(Q) &= \log_2(N_1) - \log_2(N_2) && (11)\\ &= k_1 + \log_2(1+x_1) - k_2 - \log_2(1+x_2) && (12)\\ \Rightarrow Q &= \frac{2^{k_1+k_2}(1+x_1)}{1+x_2}. && (13) \end{aligned}$$

Applying Mitchell's approximation results:

$$\begin{aligned} \log_2(Q') &= \log_2(N_1') - \log_2(N_2') && (14)\\ &= k_1 + x_1 - k_2 - x_2 && (15)\\ &= \begin{cases} (k_1-k_2) + (x_1-x_2) & x_1 - x_2 \geq 0 \\ (k_1-k_2-1) + (1+x_1-x_2) & x_1 - x_2 < 0 \end{cases} && (16)\\ \Rightarrow Q' &= \begin{cases} 2^{k_1-k_2}(1+x_1-x_2) & x_1 - x_2 \geq 0 \\ 2^{k_1-k_2-1}(2+x_1-x_2) & x_1 - x_2 < 0 \end{cases}, && (17) \end{aligned}$$

and the resulting division error E_d is as follows:

$$E_d = \frac{Q'-Q}{Q} = \begin{cases} \frac{(1+x_1-x_2)(1+x_2)}{1+x_1} - 1 & x_1 - x_2 \geq 0 \\ \frac{(2+x_1-x_2)(1+x_2)}{2(1+x_1)} - 1 & x_1 - x_2 < 0 \end{cases}. \quad (18)$$

The error analysis is divided into two cases.

Case 1: $x_1 - x_2 \geq 0$. For this case, the error is rearranged as follows:

$$E_d(x_1 - x_2 \geq 0) = \frac{(1+x_1-x_2)(1+x_2)}{1+x_1} - 1 = \frac{x_2(x_1-x_2)}{1+x_1}. \quad (19)$$

Given that $0 \leq x_1 < 1$ and $0 \leq x_2 < 1$, the maximal error occurs when $x_1 = 1$. Substituting $x_1 = 1$ into Equation (19) and differentiating with respect to x_2, we obtain the following:

$$\begin{aligned} E_d(x_1 - x_2 \geq 0) &= \frac{x_2(1-x_2)}{2} && (20)\\ \frac{\partial E_d(x_1 - x_2 \geq 0)}{\partial x_2} &= 1 - 2x_2. && (21) \end{aligned}$$

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

Case 2: $x_1 - x_2 < 0$. For this case, the error is rearranged as follows:

$$E_d(x_1 - x_2 < 0) = \frac{(2 + x_1 - x_2)(1 + x_2)}{2(1 + x_1)} - 1 = \frac{(x_2 - x_1)(1 - x_2)}{2(1 + x_1)}. \tag{22}$$

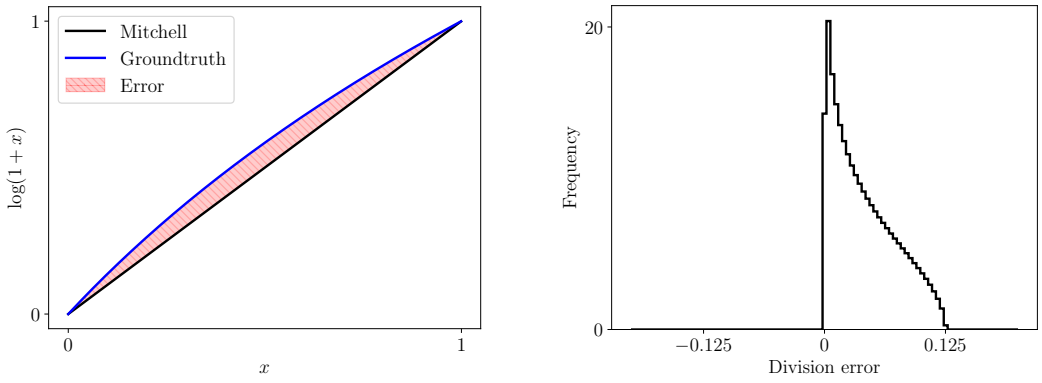
Here, the maximal error occurs when $x_1 = 0$. Substituting $x_1 = 0$ into Equation (22) and differentiating with respect to x_2, we obtain the following:

$$E_d(x_1 - x_2 < 0) = \frac{x_2(1 - x_2)}{2} \tag{23}$$

$$\frac{\partial E_d(x_1 - x_2 < 0)}{\partial x_2} = 1 - 2x_2. \tag{24}$$

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

Figure 2 illustrates the two types of errors introduced by Mitchell's algorithm. The first error, shown in Figure 2a, arises from the approximation $\log_2(1 + x) \approx x$. The second error, demonstrated in Figure 2b, results from applying this approximation to division operations. Specifically, Figure 2b depicts the distribution of division errors, aligning with the aforementioned analysis. The error ranges from a minimum of 0 to a maximum of 0.125. Clearly, improving the approximation in Figure 2a directly reduces the division error shown in Figure 2b, which serves as the primary focus of the subsequent section.



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

Figure 2. Illustration of errors introduced by Mitchell's algorithm. (**a**) Error resulting from the approximation $\log_2(1 + x) \approx x$. (**b**) Distribution of division errors when applying Mitchell's algorithm.

2.3.2. Discontinuous Piecewise Linear Approximation

To enhance accuracy, Ha and Lee [19] proposed a piecewise linear approximation for $\log_2(1 + x)$ instead of using a single straight-line approximation over the entire interval $0 \leq x < 1$. They partitioned the range into a predefined number of unequally spaced regions and applied linear approximations within each region. Specifically, each region was further divided into k sub-regions, with a separate straight-line approximation for $\log_2(1 + x)$ in each sub-region. To facilitate hardware implementation, all k lines within a given region shared the same slope. As an example, Ha and Lee [19] used two regions ($0 \leq x < 0.4142$ and $0.4142 \leq x < 1$), each containing three sub-regions ($k = 3$), approximating $\log_2(1 + x)$ as follows:

$$\log_2(1 + x) \approx \begin{cases} 1.2071x + 0.0144 & 0.0796 \leq x < 0.3187 \\ 1.2071x + 0.0072 & 0 \leq x < 0.0796 \text{ or } 0.3187 \leq x < 0.4142 \\ 0.8536x + 0.1609 & 0.5268 \leq x < 0.8649 \\ 0.8536x + 0.1537 & 0.4142 \leq x < 0.5268 \text{ or } 0.8649 \leq x < 1 \end{cases}. \tag{25}$$

The primary limitation of this method is its manual design and optimization for hardware implementation. No systematic methodology was proposed for extending the approach to more general cases.

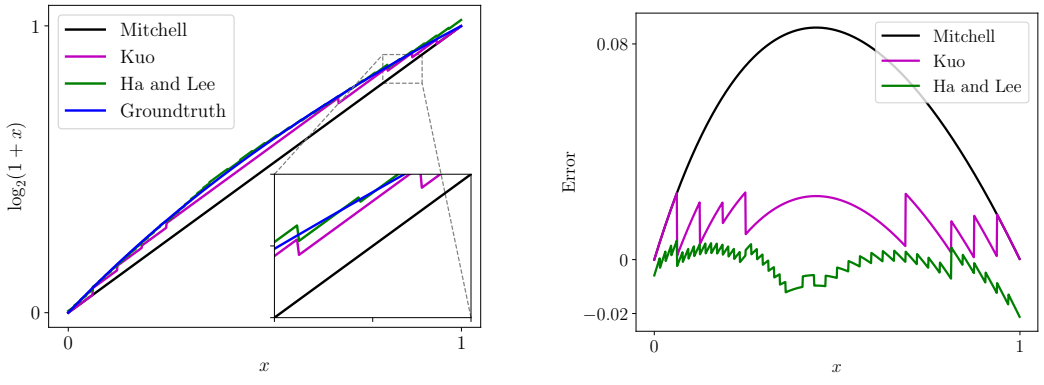
2.3.3. Non-Uniform Multi-Region Constant Adder Correction

Kuo [20] introduced another approach to reduce approximation error by dividing the range into a predefined number of equally spaced regions. Within each region, a constant was added to the straight-line approximation used in Mitchell's algorithm, effectively creating parallel lines that resulted in smaller approximation errors. To further optimize the method, neighbouring regions with identical constant values were merged, forming a non-uniform multi-region constant adder correction scheme.

However, like Ha and Lee's approach [19], Kuo's method also lacks a systematic procedure for deriving the approximation formula. As an example, Kuo [20] divided the range into nine regions and approximated $\log_2(1+x)$ as follows:

$$\log_2(1+x) \approx \begin{cases} x & 0 \le x < 0.0625 \\ x + 0.0234375 & 0.0625 \le x < 0.125 \\ x + 0.0390625 & 0.125 \le x < 0.1875 \\ x + 0.046875 & 0.1875 \le x < 0.25 \\ x + 0.0625 & 0.25 \le x < 0.6875 \\ x + 0.04296875 & 0.6875 \le x < 0.8125 \\ x + 0.03125 & 0.8125 \le x < 0.875 \\ x + 0.015625 & 0.875 \le x < 0.9375 \\ x & 0.9375 \le x < 1 \end{cases}. \tag{26}$$

Figure 3 compares the approximation lines used in Mitchell's [7], Ha and Lee's [19], and Kuo's [20] methods, with the corresponding approximation errors shown in Figure 3b. It is evident that Ha and Lee's [19] method and Kuo's [20] method significantly reduce approximation errors compared to Mitchell's algorithm. Among these, Ha and Lee's [19] approach exhibits the best performance, closely approximating the $\log_2(1+x)$ curve.



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

Figure 3. Comparison of methods improving upon Mitchell's algorithm. (**a**) Approximation lines used in each method, with the region $0.8 \le x \le 0.9$ enlarged for better visualization. (**b**) Corresponding approximation errors.

3. Proposed Method

We develop the proposed method by adopting a similar approach to that of Kuo [20], partitioning the fraction into equally spaced regions and determining an offset for each region to minimize the approximation error. Furthermore, we present a systematic methodology for extending the proposed method to general cases.

Let N be a number whose binary logarithm, as computed by Mitchell's algorithm, is given by $\log_2(N') = k + x$. The associated error is as follows:

$$R(x) = \log_2(1 + x) - x. \tag{27}$$

If $R(x)$ is added to $\log_2(N')$, the exact logarithm is obtained as $\log_2(N) = k + \log_2(1 + x)$. Let z denote a point within the fractional range. If $R(z)$ could be computed for every possible z, the exact logarithm could be determined. However, this approach is impractical due to the limited representation capabilities of computer systems, where numbers must be represented with a fixed number of bits.

To address this issue, we partition the fraction into M equally spaced regions, with M chosen as a power of two to simplify hardware implementation. The i-th region is defined as follows:

$$S_i = \left\{ x \mid \frac{i-1}{M} \leq x < \frac{i}{M} \right\}, \tag{28}$$

where $i = 1, 2, ..., M$. Based on Mitchell's algorithm, we approximate $\log_2(1 + x)$ as follows:

$$\log_2(1 + x) \approx x + \Delta(i), \tag{29}$$

where the offset $\Delta(i)$ is specific to S_i. Several methods can be used to define this offset. The simplest approach is to use the error at the region boundary. For example, defining the offset using the right-end boundary error yields the following:

$$\Delta_{\text{right}}(i) = R\left(\frac{i}{M}\right) = \log\left(1 + \frac{i}{M}\right) - \frac{i}{M}. \tag{30}$$

This ensures that the approximation error at the right end is zero, but the error increases toward the left end. Alternatively, the offset can be defined using the error at the central point of the region:

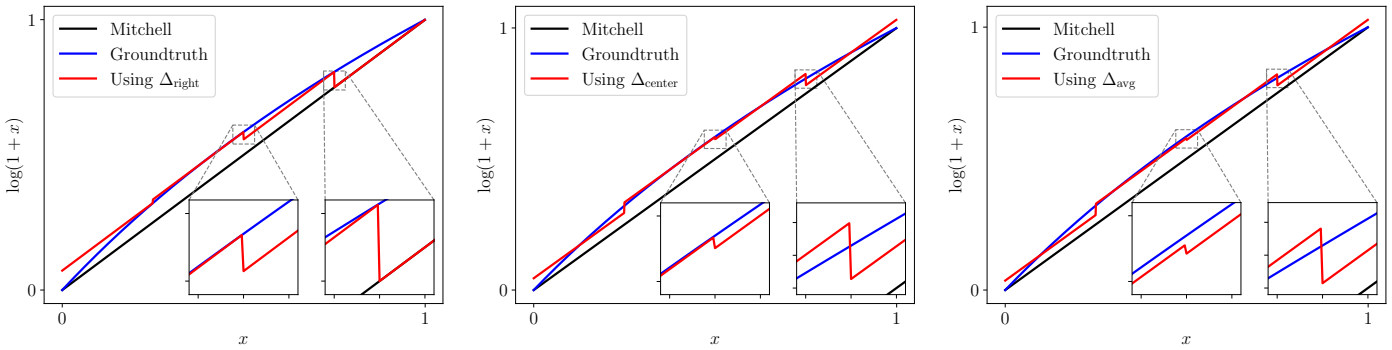
$$\Delta_{\text{center}}(i) = R\left(\frac{2i-1}{2M}\right) = \log\left(1 + \frac{2i-1}{2M}\right) - \frac{2i-1}{2M}. \tag{31}$$

This distributes the error more evenly within the region, but due to the nonlinearity of logarithm and antilogarithm functions, the errors at the two ends are unequal.

To ensure equal errors at both region boundaries, we define the offset as the average of the errors at the left and right boundaries:

$$\Delta_{\text{avg}}(i) = \frac{1}{2}\left[R\left(\frac{i-1}{M}\right) + R\left(\frac{i}{M}\right)\right]. \tag{32}$$

Figure 4 illustrates the three approximation lines corresponding to these offset definitions. The fractional range is divided into four regions, with an enlarged view of the third region for better visualization of approximation errors. When using Δ_{right}, the error is zero at the right end but increases toward the left end, reaching 0.0276. Using Δ_{center}, the errors are more evenly spread; however, they remain unequal at the two ends (left: 0.0095, right: 0.0181). The proposed method, which employs Δ_{avg}, ensures that errors at both ends are equal, yielding an error of 0.0138 for the third region.



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

Figure 4. Approximation lines corresponding to different offset definitions. (**a**) Δ_{right}. (**b**) Δ_{center}. (**c**) Δ_{avg}. The fraction is divided into four regions, with an enlarged view of the third region for clarity.

The steps for computing the binary logarithm of a number N using the proposed method are summarized in Algorithm 3. Figure 5a compares the approximation error of the proposed method ($M = 32$) against Mitchell's [7], Ha and Lee's [19], and Kuo's [20] methods. Figure 5b further illustrates how the approximation error decreases as M increases from 8 to 16, 32, and 1024.

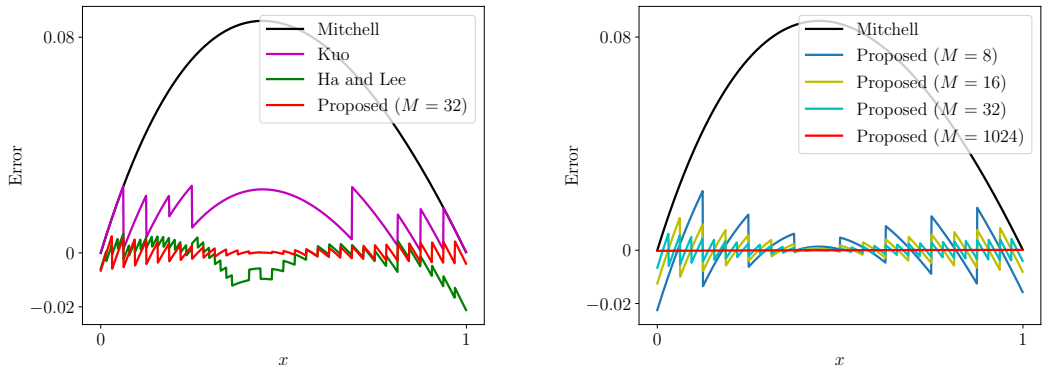
Summary statistics, including minimum, maximum, mean, and standard deviation of the errors, are presented in Table 1. The results indicate that errors in Mitchell's [7] and Kuo's [20] methods are all positive, signifying an uneven error distribution. In contrast, Ha and Lee's [19] method exhibits a distribution skewed toward negative values. The proposed method achieves a more balanced error distribution, with both the mean and standard deviation approaching zero as M increases, making it the most accurate approach for logarithm computation.

Algorithm 3 Regional Error Correction For Binary Logarithm Calculation

Input: Integer N
Parameter(s): Number of regions M
Output: Logarithm value $\log(N)$
Begin
1: Rearrange N as $N = 2^k(1+x)$, extracting k and x
2: Determine the region index $i = \lfloor M \cdot x \rfloor + 1$
3: Compute the offset $\Delta_{\text{avg}}(i) = \frac{1}{2}\left[R\left(\frac{i-1}{M}\right) + R\left(\frac{i}{M}\right)\right]$
4: Compute the logarithm $\log(N) = k + x + \Delta_{\text{avg}}(i)$
End

Table 1. Summary statistics of approximation errors for different methods. "Std." denotes standard deviation.

Method		Minimum	Maximum	Mean	Std.
Mitchell		0.0000	0.0861	0.0573	0.0257
Kuo		0.0000	0.0249	0.0148	0.0065
Ha and Lee		−0.0212	0.0070	−0.0024	0.0055
Proposed	$M = 8$	−0.0225	0.0222	0.0009	0.0072
	$M = 16$	−0.0125	0.0121	0.0002	0.0036
	$M = 32$	−0.0066	0.0062	0.0001	0.0018
	$M = 1024$	−0.0002	0.0001	0.0000	0.0001



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

Figure 5. Approximation error analysis of the proposed method. (**a**) Comparison of errors among different methods. (**b**) Approximation errors of the proposed method for varying values of M.

The method for computing the antilogarithm follows a similar approach and is summarized in Algorithm 4. Recall that for a given number N, the logarithm and antilogarithm computed using Mitchell's algorithm are $k + x$ and 2^{k+x}, respectively, while the exact values are $k + \log_2(1 + x)$ and $2^k(1 + x)$. Mitchell's approximation assumes $\log_2(1 + x) \approx x$ or $2^x \approx 1 + x$. Consequently, the error in computing the antilogarithm using Mitchell's algorithm is given by the following:

$$A(x) = 2^x - 1 - x. \tag{33}$$

Given that the fraction x belongs to the i-th region, the offset $\nabla_{\text{avg}}(i)$ is defined analogously as follows:

$$\nabla_{\text{avg}}(i) = \frac{1}{2}\left[A\left(\frac{i-1}{M}\right) + A\left(\frac{i}{M}\right)\right], \tag{34}$$

$$2^x \approx 1 + x + \nabla_{\text{avg}}(i). \tag{35}$$

Algorithm 4 Regional Error Correction For Antilogarithm Calculation

Input: Logarithm value $\log(N)$
Parameter(s): Number of regions M
Output: Antilogarithm value N
Begin
1: Compute the integer $k = \lfloor \log(N) \rfloor$ and fraction $x = \log(N) - k$
2: Determine the region index $i = \lfloor M \cdot x \rfloor + 1$
3: Compute the offset $\nabla_{\text{avg}}(i) = \frac{1}{2}\left[A\left(\frac{i-1}{M}\right) + A\left(\frac{i}{M}\right)\right]$
4: Compute the antilogarithm $N = 2^{k+1+x+\nabla_{\text{avg}}(i)}$
End

4. Error Analysis

We have introduced the proposed method for computing binary logarithms and antilogarithms and demonstrated its superiority over benchmark methods. In this section, we analyze the division error and investigate the impact of two key parameters: the partitioning parameter M and the wordlength W of the offsets $\Delta_{\text{avg}}(i)$ and $\nabla_{\text{avg}}(i)$.

Let Q' denote the quotient obtained by using the proposed method, and let Q denote the reference quotient computed using the standard digit recurrence method [10]. Although digit recurrence is the slowest among the three division techniques discussed in Section 2,

it is also the most accurate. Therefore, we use digit recurrence division as the reference method in this analysis. The division error is defined as follows:

$$E = \frac{Q' - Q}{Q}. \tag{36}$$

Table 2 presents the error variation when the wordlength is fixed at 10 bits, and the partitioning parameter is varied. The analysis also considers different wordlengths for the dividend, divisor, and quotient. To represent wordlength configurations, we use the notation "Input/Output," where "Input" specifies the wordlength of the dividend and divisor, while "Output" indicates the wordlength of the quotient. For example, the notation "8/16" in Table 2 signifies that both the dividend and divisor are 8-bit numbers, while the quotient is represented using 16 bits.

Table 2. Error analysis for varying partitioning parameter M. The wordlength W of the offsets $\Delta_{\text{avg}}(i)$ and $\nabla_{\text{avg}}(i)$ is fixed at 10 bits.

Metric	M	**Input/Output Wordlength (Bits)**								
		8/16	**9/18**	**10/20**	**11/22**	**12/24**	**13/26**	**14/28**	**15/30**	**16/32**
E (%)	8	3.493	3.657	3.657	3.698	3.719	3.729	3.724	3.722	3.721
	16	1.774	1.951	1.989	1.994	2.035	2.034	2.040	2.042	2.045
	32	0.971	0.859	0.952	0.997	1.010	1.021	1.021	1.024	1.023
	1024	0.103	0.103	0.110	0.112	0.111	0.112	0.112	0.108	0.111
	2048	0.120	0.100	0.120	0.112	0.102	0.102	0.098	0.103	0.103
	4096	0.098	0.100	0.094	0.112	0.103	0.098	0.103	0.112	0.112

From Table 2, it is evident that the error decreases significantly as the partitioning parameter increases. When the fraction is divided into 1024 regions, the error is approximately 0.1%, indicating that division using the proposed method closely approximates the standard digit recurrence method. Even with a relatively low partitioning parameter of $M = 8$, the error ranges between 3.5% and 3.7%, which is a substantial improvement over the 12.5% error observed with Mitchell's algorithm.

Table 3 examines the error variation when the partitioning parameter is fixed at 1024 and the wordlength is varied. Similar to Table 2, this analysis evaluates different wordlengths for the dividend, divisor, and quotient. The results confirm that the error decreases as the wordlength of the offsets increases. However, beyond 10 bits, the reduction in error becomes negligible. As demonstrated in Table 2, the division error associated with the proposed method is significantly lower than that of Mitchell's algorithm.

Table 3. Error analysis for varying wordlength W of the offsets $\Delta_{\text{avg}}(i)$ and $\nabla_{\text{avg}}(i)$. The partitioning parameter M is fixed at 1024.

Metric	W	**Input/Output Wordlength (Bits)**								
		8/16	**9/18**	**10/20**	**11/22**	**12/24**	**13/26**	**14/28**	**15/30**	**16/32**
E (%)	8	0.452	0.395	0.452	0.452	0.452	0.452	0.403	0.398	0.417
	10	0.103	0.103	0.110	0.112	0.111	0.112	0.112	0.108	0.111
	12	0.044	0.044	0.045	0.037	0.041	0.040	0.042	0.039	0.042
	14	0.034	0.032	0.037	0.034	0.034	0.034	0.031	0.028	0.031
	16	0.031	0.034	0.033	0.032	0.034	0.034	0.028	0.028	0.030
	18	0.032	0.033	0.034	0.032	0.034	0.032	0.028	0.027	0.031

Although increasing M and W reduces division error, higher values of these parameters also increase hardware complexity. Fortunately, from Tables 2 and 3, it is evident that

no significant improvements are observed beyond $M = 1024$ and $W = 10$. Based on this observation, we adopt $M = 1024$ and $W = 10$ for the hardware implementation discussed in Section 5, demonstrating that the proposed divider is significantly more compact and energy-efficient than benchmark designs.

This error analysis validates the effectiveness of the proposed method for division operations. It maintains the computational simplicity inherent in logarithm-based division while significantly enhancing precision. The subsequent section presents the hardware implementation and evaluates its computational efficacy.

5. Hardware Implementation

5.1. Hardware Architecture

Figure 6 illustrates the hardware architecture for implementing the proposed divider. The wordlengths of all signals are configured for an example case with an 8-bit dividend, 8-bit divisor, and 16-bit quotient.

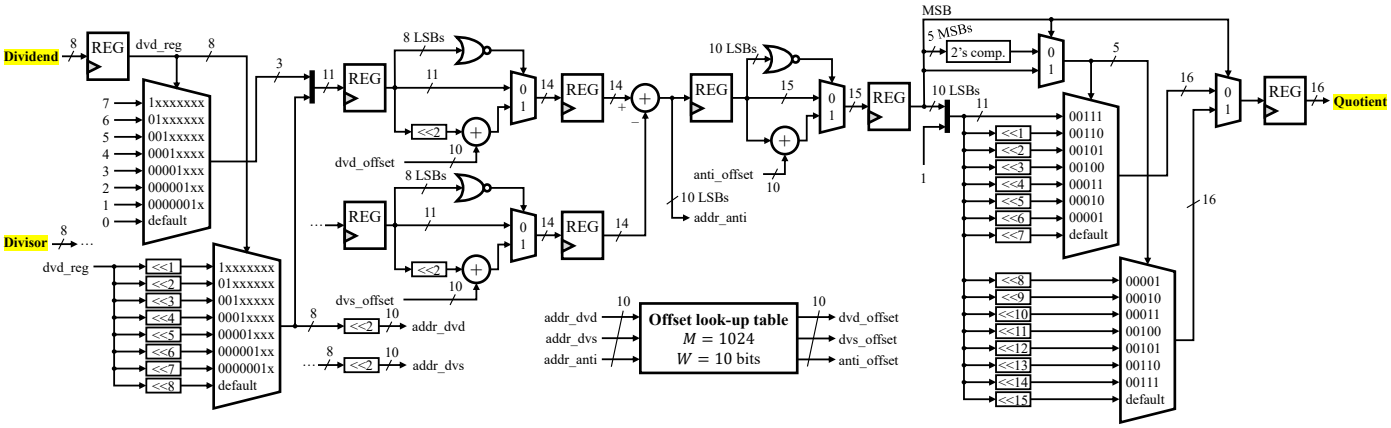


Figure 6. Hardware architecture of the proposed divider. REG, MSB, and LSB denote register, most significant bit, and least significant bit, respectively. The "..." symbol indicates that the data path for the divisor is identical to that of the dividend.

The computation begins by extracting the integer part k and the fraction part x from the input numbers. The proposed design scans from the most significant bit (MSB) rightward to locate the first nonzero bit, which determines k (with the least significant bit (LSB) assigned position zero). The input number is then shifted left until the first nonzero MSB is removed, leaving the fraction x. In Figure 6, two priority encoders perform this operation.

Next, the integer and fraction are concatenated, and the logarithm of the divisor is subtracted from that of the dividend. Borrow propagation in binary subtraction naturally accounts for the two cases in Equation (17). A 2-to-1 multiplexer determines whether the fraction is zero, as the offset is applied only when the fraction is nonzero.

The logarithm offset $\Delta_{\text{avg}}(i)$ and antilogarithm offset $\nabla_{\text{avg}}(i)$ are precomputed for $M = 1024$ and $W = 10$ bits. These values are stored in a small LUT, which is mapped to logic circuits on the target FPGA. To retrieve the offsets, the fractions of the dividend, divisor, and quotient are zero-padded to 10 bits and used as LUT addresses.

Once the logarithm of the quotient is obtained, the integer and fraction are extracted via bit slicing. The integer serves as a selection signal, while the fraction is padded with a leading 1 and shifted according to the integer value.

The entire hardware architecture completes the division in just six clock cycles, a significant improvement over the standard digit recurrence and functional iteration dividers, which requires 16 and 8 clock cycles, respectively.

5.2. FPGA Implementation and Performance Evaluation

The hardware architecture was implemented using Verilog HDL (IEEE standard 1364-2005) [21] and targeted to the XCZU7EV-2FFVC1156 FPGA, part of the Zynq UltraScale+ family. This chip features an Arm Cortex-A53 quad-core processor, a Cortex-R5F dual-core real-time processor, a Mali-400 GPU, 460, 800 registers, 230, 400 LUTs, 11Mb block RAM, 27 Mb UltraRAM, 1728 DSP slices, and a video encoder/decoder unit. The implementation was synthesized using Xilinx Vivado v2024.2 [22], and the results are summarized in Table 4.

To assess scalability, dividers with varying input/output wordlengths were implemented and compared against restoring digit recurrence, Radix-2, High-Radix, and LutMultA dividers. Details of the restoring divider implementation are provided in Appendix A of [10], while the Radix-2, High-Radix, and LutMultA dividers are described by Xilinx [11].

The performance evaluation considered five key metrics:

- Number of registers;
- Number of LUTs;
- Maximum operating frequency ($f_{\max}$);
- Power consumption (P_{con});
- Latency.

For High-Radix and LutMultA dividers, the number of block RAMs (BRAMs) and the number of DSP slices (DSP48s) were also analyzed.

The results indicate that:

- Restoring and Radix-2 dividers exhibit similar resource utilization, power consumption, and latency. However, Radix-2 dividers require more registers, leading to faster processing speeds for large wordlengths (24/48 and 32/64), albeit with higher power consumption.
- High-Radix dividers significantly reduce the latency for large wordlengths while utilizing block RAMs and DSP slices, resulting in lower registers and LUT usage compared to restoring and Radix-2 dividers.
- LutMultA dividers are more suitable for small wordlengths. For input wordlength less than 12 bits, they require only 8 clock cycles. Although they are slower than restoring and Radix-2 dividers, their processing speed remains sufficient for real-time processing. However, as Xilinx's divider generator does not support LutMultA division for wordlengths greater than or equal to 12 bits [11], implementation results for these cases are marked as NA (not available).

The proposed dividers are more compact and energy-efficient than all other designs, despite operating at slightly lower frequencies. For example, an 8× increase in input wordlength (from 8 to 64 bits) leads to 12.7× increase in registers, 13.8× increase in LUTs, and 2.3× increase in power consumption for restoring dividers. The corresponding increases for Radix-2 dividers are 15.4×, 12.8×, and 2.7×, respectively. As High-Radix and LutMultA dividers leverage block RAMs and DSP slices, direct comparisons are not applicable. The proposed dividers exhibit only 2.7×, 3.8×, and 1.4× increases, respectively, demonstrating superior scalability.

Moreover, the proposed divider maintains a constant latency of six clock cycles, the lowest among all dividers. Although the proposed divider is slightly slower than restoring and Radix-2 dividers, it still achieves a minimum $f_{\max}$ of 480.077 MHz, which is sufficient for real-time pattern recognition systems.

Table 4. Hardware implementation results of different dividers. The proposed hardware uses $M = 1024$ and $W = 10$. The absence of BRAM and DSP48 usage in restoring, Radix-2, and the proposed dividers indicates that they do not consume block RAMs or DSP slices. NA stands for not available.

Method	Metric *	Input/Output Wordlength (bits)										
		8/16	**9/18**	**10/20**	**11/22**	**12/24**	**13/26**	**14/28**	**15/30**	**16/32**	**24/48**	**32/64**
Restoring	Registers	350	433	521	616	722	834	951	1081	1216	2573	4447
	LUTs	320	403	485	582	675	761	865	1037	1148	2505	4404
	f_{max}	775.194	775.194	775.194	775.194	775.194	775.194	775.194	775.194	775.194	672.043	627.746
	P_{con}	0.750	0.764	0.964	0.889	0.921	1.075	1.117	1.160	1.198	1.452	1.726
	Latency	17	19	21	23	25	27	29	31	33	49	65
Radix-2	Registers	438	557	681	882	968	1139	1315	1504	1706	3806	6738
	LUTs	175	215	260	330	359	415	475	540	608	1297	2241
	f_{max}	775.194	775.194	771.605	771.605	775.194	775.194	771.605	775.194	769.823	775.194	771.605
	P_{con}	0.721	0.739	0.760	0.932	0.964	1.012	1.048	1.082	1.129	1.492	1.919
	Latency	18	20	22	24	26	28	30	32	34	50	66
High-Radix	Registers	558	656	689	724	724	795	873	908	1017	888	1136
	LUTs	386	397	431	442	459	473	533	543	727	554	710
	BRAMs	1	1	1	1	1	1	1	1	1	1	1
	DSP48s	5	5	5	5	5	5	5	5	7	9	11
	f_{max}	673.854	632.111	627.746	628.931	628.931	626.566	591.716	588.582	592.066	591.716	592.417
	P_{con}	0.763	0.799	0.801	0.807	0.810	1.041	1.025	1.027	1.054	1.192	1.383
	Latency	20	21	21	21	21	21	25	25	25	31	35
LutMultA	Registers	170	202	218	225	NA	NA	NA	NA	NA	NA	NA
	LUTs	300	308	467	437	NA	NA	NA	NA	NA	NA	NA
	BRAMs	0.5	0.5	1	2	NA	NA	NA	NA	NA	NA	NA
	DSP48s	0	0	0	0	NA	NA	NA	NA	NA	NA	NA
	f_{max}	532.198	528.541	504.796	437.828	NA	NA	NA	NA	NA	NA	NA
	P_{con}	0.694	0.903	0.920	0.880	NA	NA	NA	NA	NA	NA	NA
	Latency	8	8	8	8	NA	NA	NA	NA	NA	NA	NA
Proposed	Registers	140	152	158	164	174	184	194	204	214	300	380
	LUTs	305	391	429	461	469	679	677	717	581	846	1166
	f_{max}	724.638	676.590	685.401	672.043	645.161	529.381	534.474	537.634	573.723	531.915	480.077
	P_{con}	0.689	0.692	0.699	0.713	0.784	0.787	0.794	0.810	0.825	0.902	0.967
	Latency	6	6	6	6	6	6	6	6	6	6	6

* Registers, LUTs, BRAMs, DSP48s are measured as the quantity utilized. f_{max} represents the maximum frequency, measured in MHz. P_{con} denotes the power consumption, measured in watts. Latency is expressed in clock cycles.

5.3. Practical Application in Image Processing

To demonstrate the practical advantages of the proposed divider, we implemented two versions of the Image-Fusion-based DeHazing (IFDH) algorithm from [23]: standard version using digit recurrence dividers, and proposed version using the proposed dividers. Table 5 shows that the proposed version achieves a 9.12% reduction in register usage, a 6.73% reduction in LUT usage, and a 6.57% lower power consumption, with only a negligible 0.41% reduction in f_{max}. These results underscore the potential of the proposed divider in real-time pattern recognition systems, significantly reducing hardware resource usage and power consumption.

Table 5. Hardware implementation results for two versions of the Image-Fusion-based DeHazing (IFDH) algorithm. "Standard" refers to the version using standard digit recurrence dividers, and "Proposed" refers to the version using the proposed dividers.

Metric *	Available	Standard		Proposed	
		Used	**Utilization**	**Used**	**Utilization**
Registers	460,800	34,566	7.50%	31,413	6.82%
LUTs	230,400	28,718	12.46%	26,785	11.64%
BRAMs	312	66	21.15%	66	21.15%
f_{max}	-	373.276		371.747	
P_{con}	-	2.757		2.576	

* Registers, LUTs, and BRAMs are measured as the quantity utilized. f_{max} represents the maximum frequency, measured in MHz. P_{con} denotes the power consumption, measured in watts.

Figure 7 showcases the application of IFDH with both divider types. Input aerial images with varying haze levels (thin, moderate, and dense) are processed by IFDH for dehazing before object detection using YOLOv9 [24]. As demonstrated in Section 4, the proposed divider achieves a 0.1% error for $M = 1024$ and $W = 10$ bits, ensuring that replacing the standard dividers does not degrade performance. This is further validated by the YOLOv9 detection results in Figure 7.

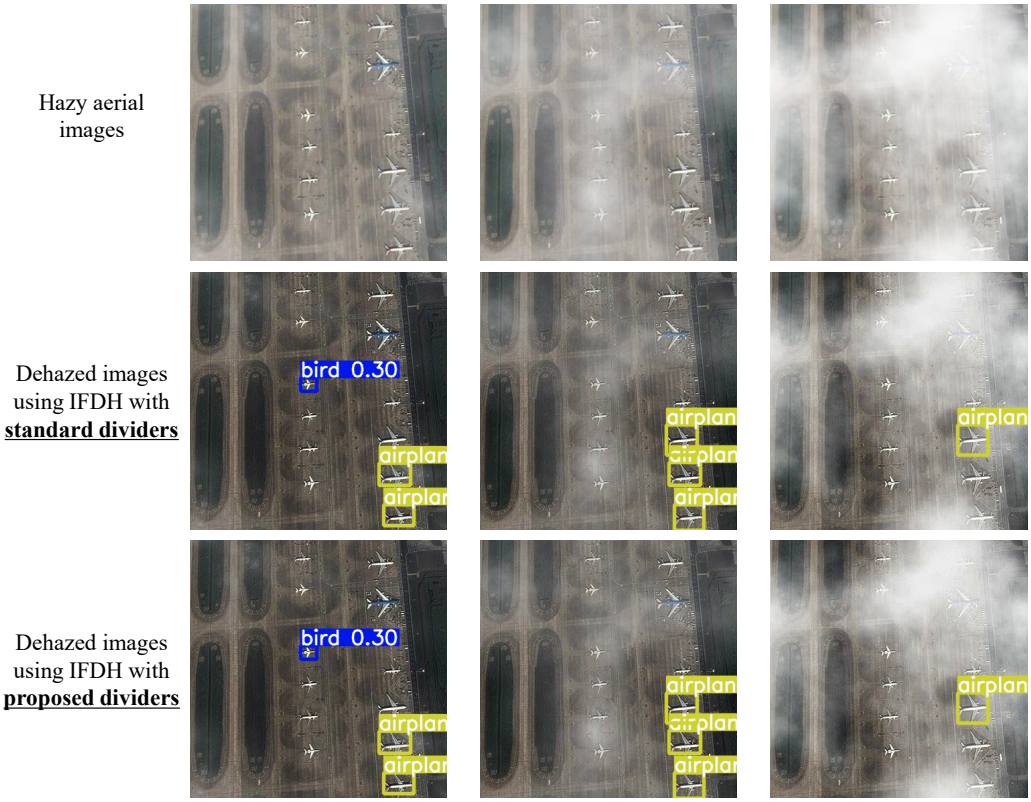


Figure 7. YOLOv9 object detection results on aerial images under varying haze levels using IFDH. Yellow labels represent airplanes, and blue labels represent birds.

6. Conclusions

In this paper, we introduced a novel method for computing binary logarithms and antilogarithms using a regional error correction mechanism, which can be easily extended to general cases. The proposed approach divides the fractional part of the input number into equally spaced regions and precomputes logarithm and antilogarithm offsets as the average of two error boundaries for each region. We analysed the approximation error and compared our method with benchmark techniques to validate its effectiveness.

To demonstrate practical applicability, we developed a six-stage pipelined architecture for implementing the proposed divider. FPGA-based hardware implementation results confirmed its superiority over benchmark dividers in terms of resource utilization and power savings. Furthermore, we integrated the proposed divider into an image-fusion-based dehazing system and the YOLOv9 object detection framework, achieving notable reductions in hardware resource utilization and power consumption while maintaining detection accuracy. These results underscore the potential of the proposed divider in optimizing real-time pattern recognition systems by reducing hardware overhead, latency, and energy consumption.

Despite its advantages, the proposed divider is currently limited to integer and fixed-point division. While these division methods remain essential for low-power, high-performance computing, extending the approach to floating-point division is a crucial next

step for broader applicability in future computing systems. We leave this extension as a direction for future research.

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

Funding: This research was funded by research funds from Dong-A University (No. 20250131), Busan, Republic of Korea.

Data Availability Statement: The dataset is available on request from the authors.

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

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

References

1. Zhang, G.; Chen, Y.; Zheng, Y.; Martin, G.; Wang, R. Local-enhanced representation for text-based person search. *Pattern Recognit.* **2025**, *161*, 111247. [CrossRef]
2. Wang, Y.; Wei, W. Local and global feature attention fusion network for face recognition. *Pattern Recognit.* **2025**, *161*, 111227. [CrossRef]
3. Zhang, Z.; Yang, L.; Wang, K.; Xi, X.; Nie, X.; Yang, G.; Yin, Y. Consistency and label constrained transfer low-rank representation for cross-light finger vein recognition. *Pattern Recognit.* **2025**, *161*, 111208. [CrossRef]
4. Fog, A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. Available online: https://www.agner.org/optimize/instruction_tables.pdf (accessed on 26 December 2024).
5. NVIDIA. CUDA Binary Utilities. Available online: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell-pascal (accessed on 26 December 2024).
6. Rodeheffer, T. Software Integer Division. Available online: https://www.microsoft.com/en-us/research/wp-content/uploads/2008/08/tr-2008-141.pdf (accessed on 16 December 2024).
7. Mitchell, J.N. Computer Multiplication and Division Using Binary Logarithms. *IRE Trans. Electron. Comput.* **1962**, *EC-11*, 512–517. [CrossRef]
8. Shaw, R. Arithmetic Operations in a Binary Computer. *Rev. Sci. Instrum.* **1950**, *21*, 690. [CrossRef]
9. McCann, M.; Pippenger, N. SRT Division Algorithms as Dynamical Systems. *SIAM J. Comput.* **2005**, *34*, 1279–1301. [CrossRef]
10. Lee, S.; Ngo, D.; Kang, B. Design of an FPGA-Based High-Quality Real-Time Autonomous Dehazing System. *Remote Sens.* **2022**, *14*, 1852. [CrossRef]
11. 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).
12. Oberman, S.; Flynn, M. Measuring the Complexity of SRT Tables. Available online: http://i.stanford.edu/pub/cstr/reports/csl/tr/95/679/CSL-TR-95-679.pdf (accessed on 19 February 2025).
13. Rodriguez-Garcia, A.; Pizano-Escalante, L.; Parra-Michel, R.; Longoria-Gandara, O.; Cortez, J. Fast fixed-point divider based on Newton-Raphson method and piecewise polynomial approximation. In Proceedings of the 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 9–11 December 2013; pp. 1–6. [CrossRef]
14. Goldschmidt, R. Applications of Division by Convergence. Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1964.
15. Soderquist, P.; Leeser, M. Division and square root: Choosing the right implementation. *IEEE Micro* **2002**, *17*, 56–66. [CrossRef]
16. 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]
17. Ngo, D.; Kang, B. Taylor-Series-Based Reconfigurability of Gamma Correction in Hardware Designs. *Electronics* **2021**, *10*, 1959. [CrossRef]
18. Arnold, M.G.; Collange, C. A Real/Complex Logarithmic Number System ALU. *IEEE Trans. Comput.* **2011**, *60*, 202–213. [CrossRef]
19. Ha, M.; Lee, S. Accurate Hardware-Efficient Logarithm Circuit. *IEEE Trans. Circuits Syst. II Express Briefs* **2017**, *64*, 967–971. [CrossRef]

20. 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]
21. *1364-2005*; IEEE Standard for Verilog Hardware Description Language. IEEE (Institute of Electrical and Electronics Engineers): Piscataway, NJ, USA, 2006; pp. 1–590. [CrossRef]
22. Xilinx. Vivado Design Suite User Guide: Designing with IP (UG896). Available online: https://docs.amd.com/viewer/book-attachment/21Juiels_eENy0SgK2kr7g/3ocj~oULvr~9S5RyFlBM3g-21Juiels_eENy0SgK2kr7g (accessed on 22 February 2025).
23. Ngo, D.; Lee, S.; Nguyen, Q.H.; Ngo, M.; Lee, G.D.; Kang, B. Single Image Haze Removal from Image Enhancement Perspective for Real-Time Vision-Based Systems. *Sensors* **2020**, *20*, 5170. [CrossRef] [PubMed]
24. Wang, C.Y.; Liao, H.Y.M. YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information. *arXiv* **2024**, arXiv:2402.13616.