Improving Accuracy in Mitchell’s Logarithmic Multiplication Using Operand Decomposition

V. Mahalingam, Student Member, IEEE, and Nagarajan Ranganathan, Fellow, IEEE

Abstract—Logarithmic number systems (LNS) offer a viable alternative in terms of area, delay, and power to binary number systems for implementing multiplication and division operations for applications in signal processing. The Mitchell algorithm (MA), proposed in [15], reduces the complexity of finding the logarithms and the antilogarithms using piecewise straight line approximations of the logarithm and the antilogarithm curves. The approximations, however, result in some loss of accuracy. Thus, several methods have been proposed in the literature for improving the accuracy of Mitchell’s algorithm. In this work, we investigate a new method based on Operand Decomposition (OD) to improve the accuracy of Mitchell’s algorithm when applied to logarithmic multiplication. In the OD technique proposed in [9] for reducing the amount of switching activity in binary multiplication, the two inputs to be multiplied are together decomposed into four binary operands and the product is expressed as the sum of the products of the decomposed numbers. We show that applying operand decomposition to the inputs as a preprocessing step to Mitchell’s multiplication algorithm significantly improves the accuracy. Experimental results indicate that the proposed algorithm for logarithmic multiplication reduces the error percentage of Mitchell’s algorithm by 44.7 percent on the average. It is also shown that the OD method yields further improvement when combined with the other correction methods proposed in the literature.

Index Terms—Computer arithmetic, error analysis, logarithmic number system, interpolation.

1 INTRODUCTION

Digital signal processing applications use tasks such as Finite Impulse Response filtering, Fast Fourier Transform, and Discrete Cosine Transform, which involve heavy use of arithmetic operations such as addition, multiplication, and division. The arithmetic operations, such as multiplication and division in a binary number system, are computationally complex in terms of area, delay, and power. While the fixed point number system could have the problem of overflow and scaling due to limited precision [4], the floating-point number system provides better precision and scaling, however, incurring significantly more overhead. Logarithmic Number Systems (LNS) offer a viable alternative combining the simplicity of a fixed point number system and the precision of a floating-point number system. In LNS-based systems, it is possible to implement operations like multiplication and division using addition and subtraction operations on the logarithms of the input data. However, the computations in the LNS domain result in some loss of accuracy and, thus, are mostly limited to those signal processing applications in which a certain amount of error is tolerable. Consequently, an exact result after multiplication is not always required and a rounded product is acceptable in many cases. Hence, LNS arithmetic is preferred in many signal processing systems due to its area, delay, and power advantages in spite of some loss of accuracy.

Swartzlander [21] was the first to develop a four function (addition, subtraction, multiplication, and division) arithmetic processor based on the logarithmic number system. Subsequently, the use of LNS has been demonstrated further in other works [10], [18], [20]. Logarithms are usually computed using the Taylor’s series or by storing a complete table of logarithms for all the numbers. These methods are inefficient in terms of the amount of hardware required as well as speed and power. Several approaches have been proposed in the literature for improving the efficiency of logarithm based arithmetic. They can be broadly classified as 1) Look Up Table (LUT)-based interpolation [3], [5] and 2) Mitchell’s algorithm-based logarithm computation [15]. Earlier studies have shown that interpolation-based methods can implement logarithms with high precision but require a considerable amount of computation and storage [5], [12], [13], [22]. On the other hand, Mitchell’s algorithm does not require any ROM and targets efficiency at the loss of some accuracy. Thus, there is a strong interest toward using Mitchell’s algorithm to implement logarithmic computations for use in DSP applications.

Mitchell’s algorithm for multiplying two numbers using logarithms is straightforward. The logarithms of the input numbers are added and the antilogarithm of the sum is determined. The method used to find the logarithm and the antilogarithm impacts the accuracy. Mitchell presented a simple method to approximate the logarithm and antilogarithm calculations [15] using piecewise straight line approximations of the logarithm and the antilogarithm curves. The actual logarithm values and Mitchell’s logarithm values are plotted in Fig. 1. The error in Mitchell’s logarithm is due to the fractional part of the logarithm. Hence, whenever the fraction part is zero, the Mitchell log curve intersects the actual logarithm curve. The logarithm...
error due to Mitchell’s approximation is in the range $0 \leq \text{Error} \leq 0.08639$ and attains the maximum value when the fraction part of the logarithm is equal to 0.44 [15]. The maximum possible error for Mitchell’s logarithm multiplication is around 11.1 percent and the average error is around 3.8 percent.

Mitchell’s method [15] has a high amount of error because of the single straight line approximation. In the same paper, Mitchell discussed the derivation of an analytical correction term that can be added to the final result to reduce the error. However, the computation of the correction term itself required the use of Mitchell’s algorithm. Later, several methods based on divided straight line approximation were proposed in the literature for improving the accuracy of the result from Mitchell’s algorithm [1], [2], [6], [8], [19]. In [7], a simulation-based correction strategy was developed for use with Mitchell’s algorithm. The method uses a fixed table of correction values for the different fractional regions of the logarithms.

In this work, we propose a new method based on operand decomposition (OD) for improving the accuracy of Mitchell’s logarithmic multiplication. The OD technique was first proposed in [9] for reducing the amount of switching activity in binary multiplication. The operand decomposition reduces the number of “1” bits in the decomposed operands which, in turn, decreases the chances of a carryover from the mantissa part to the integer part during the logarithm summation step. It is shown mathematically in [15] that the accuracy of Mitchell’s algorithm can be improved by avoiding the carryover during the summation step. We show that applying operand decomposition to the inputs as a preprocessing step to Mitchell’s algorithm significantly improves the accuracy. Further, since the OD method does not require the addition of a correction term, we show that it can be easily combined with other methods to obtain even better accuracy.

The remainder of the paper is as follows: The various related works are summarized in Section 2 and the proposed OD method is discussed in Section 3. Experimental results for random and biased input data as well as for image data are provided in Section 4 and conclusions in Section 5.

2 RELATED WORK
Swartzlander [20] and Kingsbury [10], were the first to investigate the use of LNS in digital filtering and Fast Fourier Transform, respectively. LNS-based multiplication methods are of two categories: 1) methods that use lookup tables and interpolation and 2) methods based on Mitchell’s algorithm that computes the logarithms through shift and count operations. While the lookup table-based methods involve memory overhead, Mitchell’s algorithm does not require memory; however, it incurs some loss of accuracy. Thus, several researchers have proposed methods to improve the accuracy in Mitchell’s algorithm. The various related works in logarithm-based multiplication are categorized in the taxonomy diagram given in Fig. 2. These methods are briefly explained in the rest of the section.

2.1 LUT-Based Logarithmic Multiplication
The traditional approach to logarithmic multiplication is to use complete tables of logarithm values stored in memory, which is inefficient in terms of memory requirements. Thus, several researchers investigated the idea of using smaller lookup tables and compensating with interpolation methods [5], [11], [14], [18]. The objective was to identify LUT sizes corresponding to varying logarithmic accuracy levels,
trading off accuracy versus memory overhead. In [16], [17], the multidimensional logarithmic number system (MDLNS) is investigated in which the memory requirements are reduced at the cost of additional computations. In spite of the attempts at reducing the amount of memory to store lookup tables, there is strong interest in Mitchell’s algorithm, which completely eliminates the use of lookup tables.

2.2 Mitchell’s Algorithm-Based Logarithm Multiplication

In this section, we briefly describe Mitchell’s algorithm for logarithm-based multiplication, which is illustrated in Fig. 3. Mitchell’s logarithm and antilogarithm calculations require only shifting and counting operations. The zero detector is to ensure a zero output if any of the inputs is zero. Mitchell’s algorithm [15] for multiplying two inputs A and B is given in Table 1. The position of the leading 1 bit in each input is identified by shifting the input bits left until the most significant bit is a “1,” decrementing a counter each time, which is initially loaded with the word size. The final values of these counters, \( k_A \) and \( k_B \), form the integer part of the logarithms of A and B. The remaining sequences of bits form the corresponding mantissa parts, \( m_A \) and \( m_B \). The corresponding integer and mantissa parts can be appended to form the logarithm values, \( f_A \) and \( f_B \), of the given inputs A and B. These logarithm values are added and the antilogarithm of the sum, \( f_{AB} \), is determined as follows: The value of the characteristic part of the sum, \( k_{AB} \), is decoded to determine the position in the final product, where a leading 1-bit is inserted. The mantissa part, \( m_{AB} \), of the sum is simply appended immediately to the right of the inserted leading 1-bit padded by 0-bits in the remaining positions to form the final result, which is the product \( A \times B \). A step-by-step example illustrating the above procedure for using the logarithmic multiplication of decimal numbers 18 and 58 is shown in Fig. 4.

It can be observed that Mitchell’s logarithm multiplication algorithm is simple to perform and requires only shifting and counting operations. The example shown has an error of 5 percent. Mitchell shows in [15] that the error in the multiplication is due to the fraction part of the logarithms and provides a detailed error analysis. We briefly outline only the relevant parts of the error analysis next, which is required to understand the working of the corrections methods.

2.2.1 Error Analysis for Mitchell’s Multiplication Algorithm

Consider a binary number N in the range \( 2^{k+1} > N \geq 2^k \) and \( k \geq j \). The number N can be written as

\[
N = \sum_{i=j}^{k} 2^i Z_i. 
\]  
(1)

Since \( Z_k \) is the most significant bit, we may assume \( Z_k = 1 \) for any valid \( k \geq j \), without loss of generality. Factoring out the value \( 2^k \) from N, we get

\[
N = 2^k \left( 1 + \sum_{i=j}^{k-1} 2^{i-k} Z_i \right). 
\]  
(2)

Let the second term in the above equation be referred to by \( x \), as given below.

\[
\sum_{i=j}^{k-1} 2^{i-k} Z_i = x. 
\]  
(3)

Since \( k \geq j \), the value of \( x \) will be in the range \( 0 \leq x < 1 \) and will be referred to as the fraction term of the binary number N in the rest of the paper. The number N can now be represented as

\[
N = 2^k (1 + x). 
\]  
(4)

Table 1

<table>
<thead>
<tr>
<th>Mitchell’s Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Calculate ( k_A ) ← position of leading 1-bit in input A;</td>
</tr>
<tr>
<td>Step 2: Calculate ( k_B ) ← position of leading 1-bit in input B;</td>
</tr>
<tr>
<td>% ( k_A ) and ( k_B ) form the characteristic parts of the logarithms of A and B;</td>
</tr>
<tr>
<td>Step 3: Assign ( m_A ) ← the bits to the right of the leading 1-bit in A;</td>
</tr>
<tr>
<td>Step 4: Assign ( m_B ) ← the bits to the right of the leading 1-bit in B;</td>
</tr>
<tr>
<td>% ( m_A ) and ( m_B ) form the mantissas of the logarithms of A and B;</td>
</tr>
<tr>
<td>Step 5: Append as floating point numbers ( f_A \leftarrow (k_A, m_A) ) and ( f_B \leftarrow (k_B, m_B) );</td>
</tr>
<tr>
<td>% add the logarithm values and then calculate the antilogarithm;</td>
</tr>
<tr>
<td>Step 6: ( f_{AB} \leftarrow f_A + f_B );</td>
</tr>
<tr>
<td>Step 7: Assign product([k_{AB}] \leftarrow 1 ) and append ( m_{AB} ) immediately after this 1-bit;</td>
</tr>
</tbody>
</table>
Let $A = 18 = (00010010)$ and $B = 58 = (00111010)$

Step 1: $k_A = (100)$
Step 2: $k_B = (101)$
Step 3: $m_A = (0010)$
Step 4: $m_B = (1101)$
Step 5: $f_A = (000.0010)$ and $f_B = (101.1101)$
Step 6: $f_{AB} = (000.0010) + (101.1101) = 1001.1111$
Step 7: product $= 1111100000 = 992$

Fig. 4. Mitchell’s logarithm multiplication example.

The true logarithm of this binary number and Mitchell’s approximation is shown in the following two equations. The $lg$ here refers to logarithm to the base 2.

\[
lg(N)_{\text{true}} = k + lg(1 + x), \tag{5}
\]

\[
lg(N)_{\text{approx}} = k + x. \tag{6}
\]

It can be seen from the above equation that Mitchell’s method approximates the $lg(1 + x)$ value with the value of the fraction $x$, hence eliminating the need for lookup tables. The logarithmic summation steps for each case, 1) the actual logarithms and 2) Mitchell’s logarithms, are expressed as the following set of equations:

\[
lg(N_1 + N_2) = k_1 + log_2(1 + x_1) + k_2 + log_2(1 + x_2), \tag{7}
\]

\[
lg(N_1 + N_2)_{MA} = k_1 + x_1 + k_2 + x_2. \tag{8}
\]

The antilogarithmic approximation of the above summation gives the final product $P$.

\[
P_{MA} = 2^{k_1 + k_2} (1 + x_1 + x_2). \tag{9}
\]

The error $E_m$ in Mitchell’s logarithm multiplication is then defined as

\[
E_m = \frac{P_{\text{true}} - P_{MA}}{P_{\text{true}}}, \tag{10}
\]

where $P_{\text{true}}$ is defined as the actual product calculated using standard binary multiplication. The maximum possible multiplication error using Mitchell’s method is around 11.1 percent and will occur when both of the fraction parts are equal to 0.5. Another important observation is that the error is always positive in Mitchell’s multiplication. Hence, the error can be compounded if successive multiplication is performed on the data. Several researchers have attempted to reduce this error percentage either by using separate approximation in different ranges or adding an offset value to Mitchell’s result.

### 2.3 Divided Approximation-Based Correction

In [15], Mitchell approximated the logarithm curve using a piecewise linear curve with the intervals being defined based on successive powers of two. This can be observed in the curves plotted in Fig. 1. In [1], [8], and [2], methods are described that further divide these intervals into smaller regions and a set of equations is derived to form correction terms for the various regions that can be used to improve the accuracy of the result.

#### 2.3.1 Hall’s Correction Coefficients

In [6], the range $[0, 1]$ of mantissa $x$ was partitioned into four parts and a separate approximation equation was identified by the trial and error method for each subinterval. In [8], Hall et al. developed a set of coefficients for the correction equations with the objective of minimizing the error, which, in turn, improved the results significantly. The coefficients were carefully chosen to be fractions with integer numerators and the denominators being powers of two values. Hall et al.’s correction equations for the four intervals are given below.

\[
lg(1 + x) = x + \frac{37}{128}x, \quad x \in [0.00, 0.25], \tag{11}
\]

\[
lg(1 + x) = x + \frac{3}{64}x + \frac{1}{16}, \quad x \in [0.25, 0.50], \tag{12}
\]

\[
lg(1 + x) = x + \frac{7}{64}x' + \frac{1}{32}, \quad x \in [0.50, 0.75], \tag{13}
\]

\[
lg(1 + x) = x + \frac{29}{128}x', \quad x \in [0.75, 1.00], \tag{14}
\]

where $x' = (1 - x)$.

Similarly, Hall et al. [8] derived the approximation equations for antilogarithm calculation for each region. Hall et al.’s method reduced the maximum error percentage of logarithmic multiplication from 10.1 in Mitchell’s method to 1.3. However, Hall et al.’s method has a large hardware overhead since it uses all the bits of the mantissa and the coefficients are not powers of two, which is addressed in the next method by Abed and Siferd in [1].

#### 2.3.2 Abed and Siferd’s Correction Coefficients

Abed and Siferd [1], [2] developed correction equations that required small and fast circuitry while maintaining accuracy close to Hall et al.’s method. Three different correction strategies based on equations for two, three, and six regions were proposed with varying hardware complexity and accuracy. The correction equations use only the three most significant bits of the mantissa and the coefficients are restricted to powers of two. The equations for the 2-region correcting algorithm are given below. The equations correspond to two separate approximations for different mantissa ranges of the logarithm.

\[
log_2(1 + x) = x + \frac{1}{4}x_{3\text{MSB} \text{bits}}, \quad x \in [0.0, 0.5], \tag{15}
\]

\[
log_2(1 + x) = x + \frac{1}{2}x_{3\text{MSB} \text{bits}}, \quad x \in [0.5, 1.0], \tag{16}
\]

where $x' = (1 - x_{3\text{MSB} \text{bits}} - 2^{-3})$. The correction equations for antilogarithms proposed in [2] also use only three bits of the mantissa and the coefficients that are powers of two. Thus, their method improved both the accuracy and efficiency of implementation over Hall et al.’s method.
2.4 Correction Term-Based Methods
The correction term-based methods involve adding a correction term either to the final product as in [15] or to the logarithm summation as in [7].

2.4.1 Correction Term Added to Final Product
Mitchell in [15] developed analytical expressions for calculating correction values to be added to the product result in order to improve the accuracy of the logarithmic multiplication algorithm. The carryover bit from the mantissa part to the integer part determines which one of the following two equations is to be used for correction:

\[ P_{MEC} = P_{MA} + 2^{k_1+k_2}x_1x_2, \quad x_1 + x_2 < 1, \]  
\[ P_{MEC} = P_{MA} + 2^{k_1+k_2}y_1y_2, \quad x_1 + x_2 \geq 1, \]

where \( y_1 = 1 - x_1 \) and \( y_2 = 1 - x_2 \). Generating the correction expressions requires the following steps: 1) Calculate either \( x_1x_2 \) or \( y_1y_2 \) depending on the carryover bit, 2) scale the correction term by the factor \( 2^{k_1+k_2} \), and 3) add the correction to the product computed using the Mitchell Algorithm.

2.4.2 Table of Correction Values
Duncan, in [7], devised a method of adding a correction value to the logarithm summation. In this approach, the mantissa part of the two input logarithms is partitioned into eight subintervals in steps of 0.125, and a fixed correction term was calculated for each range of the input operands. The 64 values of the fixed correction tables are identified by averaging out error values after extensive simulations with average error percentage as the optimization objective.

3 Proposed Approach
In this section, we describe a new approach which uses operand decomposition before applying Mitchell’s algorithm for logarithmic multiplication. Since the proposed method does not require the addition of any correction term, it can be used in combination with other methods to get further improvement in accuracy.

3.1 Operand Decomposition (OD)
In this section, we explain the operand decomposition approach for multiplying two binary numbers.

Consider two n-bit binary numbers X and Y of the form

\[ X = [x_{n-1}x_{n-2} \ldots x_2x_1x_0] \]  
\[ Y = [y_{n-1}y_{n-2} \ldots y_2y_1y_0]. \]

The operands X and Y are decomposed into the following four operands, A, B, C, and D:

\[ A = [a_{n-1}a_{n-2} \ldots a_2a_1a_0], \]
\[ B = [b_{n-1}b_{n-2} \ldots b_2b_1b_0], \]
\[ C = [c_{n-1}c_{n-2} \ldots c_2c_1c_0], \]
\[ D = [d_{n-1}d_{n-2} \ldots d_2d_1d_0], \]

where the individual bits in the decomposed operands are calculated using the following equations: \( a_i = x_i \land y_i \), \( b_i = x_i \lor y_i \), \( c_i = x_i \land y_i \), and \( d_i = x_i \lor y_i \). The product is then computed from the decomposed operands using the following property:

\[ X \ast Y = (C \ast D) + (A \ast B). \]

The equation can be easily verified for correctness using simple substitution and a simple proof is given in [9]. The approach increases the number of 0 bits in the decomposed multiplications and hence decreases the switching power in the multiplication operation. Further, it has been shown analytically in [9] that the decomposition reduces the probability of 1 bits occurring in the decomposed operands to \( (1/4) \) from \( (1/2) \). An example illustrating the operand decomposition approach for binary multiplication is given in Fig. 5. The example shows the logarithmic multiplication of integers 18 and 60.

The OD process reduces the number of “1” bits in the operands which, in turn, decreases the chances of a carryover from the mantissa part to the integer part during the addition of logarithms. It is shown mathematically in [15] that the accuracy of Mitchell’s algorithm can be improved by avoiding the carryover during the summation step. We show that applying operand decomposition to the inputs as a preprocessing step to Mitchell’s algorithm significantly improves the accuracy. In the next section, we explain the Operand Decomposed (OD)-Mitchell’s algorithm.

3.2 Functional Architecture of OD-Mitchell
The block diagram of the OD-Mitchell algorithm is shown in Fig. 6. The computations consist of six major blocks: the decomposition unit, the logarithm unit, the antilogarithm unit, two binary adders, and a zero detector unit. The decomposition unit is simple to design and only requires 6*\( n \) 2-input (and, or, not) gates, where \( n \) is the number of input bits in the multiplication. The logarithm and the antilogarithm units can be designed as combinational logic blocks [1], [2]. The logarithm unit has a leading one detector circuit, an \( (n \times \log_2 n) \) ROM, and a logarithmic shifter. The antilogarithm unit requires only a single logarithmic shifter. The gate level implementation details of these internal units are explained in detail in [1], [2]. The Operand Decomposed (OD)-Mitchell’s logarithmic multiplication algorithm is...
shown in Table 2. Here, the operands X and Y are decomposed to A, B, C, and D using the procedure given in Section 3.1. The decomposed operands are then multiplied separately and added together to obtain the final product.

An example is given in Fig. 7 to illustrate the working of the operand decomposed logarithmic multiplication. As shown in the figure, the multiplicands X = d’18 and Y = d’60 are decomposed into four numbers, A = d’62, B = d’16, C = d’44, and D = d’02. The decomposed operands are then multiplied separately using Mitchell’s logarithm multiplication. The results of these decomposed products, \( \text{sum1} = d’992 \) and \( \text{sum2} = d’88 \), are then added to get the final product, \( X \times Y = d’1080 \). The same logarithmic multiplication example is shown in Fig. 8 without operand decomposition. The final product in this case is equal to \( d’1024 \) with an error of approximately 5.2 percent. The average error percent of Mitchell’s logarithmic multiplication after operand decomposition reduced to 2.1 percent, an improvement of around 45 percent. Next, we investigate the combination of OD with other methods to further improve the accuracy of Mitchell’s algorithm.

### 3.3 Combining OD with Divided Approximation

In this section, we describe a new algorithm that combines operand decomposition with Abed and Siferd’s [1] divided approximation approach for logarithmic multiplication. Here, the operands X and Y are decomposed to A, B, C,
and D using the procedure given in Section 3.1. The decomposed operands A, B and C, D are then multiplied separately using Mitchell’s logarithmic multiplication procedure. The products of these multiplications, \( P_{AB} \) and \( P_{CD} \), are then added to get the final product \( X \times Y \). Simulations indicate that the same approximation equations for both the decomposed multiplications provide better error range. Hence, in this work, we use the following 2-region functions for the decomposed multiplications. The coefficients for these 2-region functions are identified from extensive trial and error simulations. These simulations take the values of product \( P_{AB} \) and product \( P_{CD} \) into account with average error percentage and error range as the improvement metric. The proposed OD algorithm with divided approximation is shown in Table 3. The combination of operand decomposition with the proposed approximation equations reduces the average error percent to less than 0.2 percent.

### 3.4 Combining OD with Error Correction-Based Approaches

In this section, we investigate combining the operand decomposition with the table of correction values (TCV)-based approach and Mitchell’s error correction (MEC) approach. The decomposed multiplications \( A \times B \) and \( C \times D \) in the TCV and MEC-based approach use similar correction tables and equations.

#### 3.4.1 OD with Table of Correction Values (TCV)

The TCV method is a simulation-based approach. Here, a constant list of values to be added to Mitchell’s logarithmic summation is stored in a table. The fractional value of the logarithms is used to access the entries in the table. Steps 15-26 in Table 4 illustrate the method to access the appropriate correction term from the \( TCV_{array} \) given the fraction values. Here, we develop a new set of correction values for the entries, depending on the ratio of decomposed products, \( A \times B \) and \( C \times D \). The same table of correction values is used for both the multiplications \( A \times B \) and \( C \times D \). The fraction values \( m_A \), \( m_B \), \( m_C \), and \( m_D \) are used to access the table.

#### TABLE 3

**OD-DA Mitchell’s Logarithmic Multiplication Algorithm**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Decompose the numbers X and Y into A, B, C, and D;</td>
</tr>
<tr>
<td>2</td>
<td>Calculate ( k_A ) ← position of leading 1-bit in input A;</td>
</tr>
<tr>
<td>3</td>
<td>Calculate ( k_B ) ← position of leading 1-bit in input B;</td>
</tr>
<tr>
<td>4</td>
<td>Assign ( m_A ) ← the bits to the right of the leading 1-bit in A;</td>
</tr>
<tr>
<td>5</td>
<td>Assign ( m_B ) ← the bits to the right of the leading 1-bit in B;</td>
</tr>
<tr>
<td>6</td>
<td>Calculate ( m_A ) ← 2-region improve(( m_A ));</td>
</tr>
<tr>
<td>7</td>
<td>Calculate ( m_B ) ← 2-region improve(( m_B ));</td>
</tr>
<tr>
<td>8</td>
<td>Append as floating point numbers ( f_A ) ← ( (k_A, m_A) ) and ( f_B ) ← ( (k_B, m_B) );</td>
</tr>
<tr>
<td>9</td>
<td>Add the logarithm values and then calculate the antilogarithm;</td>
</tr>
<tr>
<td>10</td>
<td>Assign ( P_{AB}[k_{AB}] ) ← 1 and append ( m_{AB} ) immediately after this 1-bit;</td>
</tr>
<tr>
<td>11</td>
<td>Perform steps 2 and 3 to obtain integer part of C and D and store the values in ( k_C ) and ( k_D );</td>
</tr>
<tr>
<td>12</td>
<td>Calculate ( m_C ) and ( m_D ) as in steps 4, 5, 6, and 7;</td>
</tr>
<tr>
<td>13</td>
<td>Append ( f_C ) ← ( (k_C, m_C) ) and ( f_D ) ← ( (k_D, m_D) ) as a number;</td>
</tr>
<tr>
<td>14</td>
<td>( f_{CD} ) ← ( f_C + f_D );</td>
</tr>
<tr>
<td>15</td>
<td>Assign ( P_{CD}[k_{CD}] ) ← 1 and append ( m_{CD} ) immediately after this 1-bit;</td>
</tr>
<tr>
<td>16</td>
<td>Product: ( X \times Y ) ← ( P_{AB} + P_{CD} );</td>
</tr>
<tr>
<td>17</td>
<td>function 2-region improve(( m ));</td>
</tr>
<tr>
<td>18</td>
<td>2 region approximation for the mantissa of operands;</td>
</tr>
<tr>
<td>19</td>
<td>if (( m &lt; 0.5 ))</td>
</tr>
<tr>
<td>20</td>
<td>( m ) ← ( m + (1/8)m )</td>
</tr>
<tr>
<td>21</td>
<td>else</td>
</tr>
<tr>
<td>22</td>
<td>( m ) ← ( m + (1/8)(1 - m - 1/8) )</td>
</tr>
</tbody>
</table>

---

**Fig. 8. Mitchell’s example.**

\[
\begin{align*}
X &= b’{00010010} = d’18 \\
Y &= b’{00111100} = b’60 \\
\lg X &= 0100.001000 \\
\lg Y &= 0101.111000 \\
\text{Sum} &= \lg x + \lg Y = 1010.000000 \\
X \times Y &= \text{Antilog (Sum)} = 0000001000000000 \\
X \times Y &= d’1024
\end{align*}
\]
which has 64 entries. The entries in the table are developed using extensive trial and error simulations. The combination of operand decomposition with table of correction values (OD-TCV) is shown in Table 4. The algorithm is similar to Mitchell’s calculation, except that a correction (TCV) value is added during the summation of the logarithms. The values in the table correspond to the correction term that is to be added to the logarithmic summation. As an example illustration, consider two logarithm values, log \( A = 3.4 \) and log \( B = 2.8 \). The mantissa part of these logarithms is \( m_A = 0.4 \) and \( m_B = 0.8 \), hence a value of 0.0537 corresponding to these mantissa ranges in TCV would be added to the logarithmic summation. The approach produces a very good error range and decreases the average error percentage to about 0.01 percent.

### 3.4.2 Combining OD with Mitchell’s Correction Term

Mitchell, in [15], presented a simple correction approach requiring only two extra additions. The approach adds a correction term to the original product, generated by Mitchell’s algorithm. In this work, we combine operand decomposition with Mitchell’s correction method without any modification to the basic approach. The equations for these correction terms are given in Section 2. This method significantly improves the error range as 99 percent of the logarithm multiplications now have an error value of less than 1 percent.

### 4 EXPERIMENTAL RESULTS

In this section, we first discuss the experimental setup and then present the results. The algorithms were implemented in C and compiled using the gcc compiler under Sun Solaris. The algorithms were tested with random input data generated using the rand() function in stdio.h. In the domain of logarithmic multiplication, two metrics are commonly used for comparison which are the maximum possible error and the average error percentage. The proposed operand decomposition approach does not reduce the maximum possible error except for certain methods. The operand decomposition approach gives the error curve a Gaussian shape and, hence, improves on the
average error percent. This average error percentage is important in many digital signal processing applications, including 2D-Correlation, FIR filtering, IIR filtering, and many other applications involving the convolution of two matrices. The average error percent (AEP) is defined as follows:

$$\text{Error Percent} (EP) = \frac{TV - LV}{TV} \times 100,$$

(26)

where TV refers to the true value obtained using binary multiplication and LV refers to value obtained using the proposed logarithmic multiplication.

$$\text{Average Error Percent} (AEP) = \frac{\sum_{i=1}^{N} EP_i}{N},$$

(27)

where N is the number of multiplications performed. The generated results are then compared with the error percentage of the previous approaches. These methods are also implemented and applied with the same input patterns for the sake of fairness. Second, the paper also gives a distribution of the number of values below a particular error value. For example, in the proposed operand decomposition approach, 45 percent of the multiplications have an error of less than 1 percent, compared to only 21 percent in Mitchell’s algorithm. A similar distribution is presented for the various operand decomposition combined error correction approaches. The proposed operand decomposition approach gives better results with both metrics in all cases. The simulation results are presented in the next section.

4.1 Operand Decomposition versus Mitchell’s Logarithm

Mitchell’s original method due to its piecewise linear approximation in power of two intervals has an average error of 3.88 percent. On combining Mitchell’s logarithm with operand decomposition, the average error percentage reduces to 2.1 percent for 32 bit width multiplicands. The simulations were performed with 1,000, 10,000, and 100,000 random input vectors. The error percentage of the tested approaches did not vary much with the input size. Figs. 9 and 10 show the error spread of 500 random input multiplications for Mitchell’s and OD-Mitchell’s multiplication for 32-bit width inputs. The dots in the figures represent the error percent for individual multiplication and the curve represents the moving average over different random blocks. The same convention is followed in all the error spread graphs presented in this section. It can be clearly seen that, for the OD-Mitchell’s method, more errors are near zero. Further, the percentage of values below a certain error range for 100,000 multiplications are shown in Table 6. The average error percentage for various input widths and 100,000 multiplications is shown in Table 7.

The table shows the average error percent for four different multiplicand widths and two different numbers of multiplications (N). It can be clearly seen from these tables that our method improves the error range and the average error percentage of Mitchell’s logarithmic multiplication by 45 percent. The proposed operand decomposition approach was also tested with biased random inputs. The random inputs generated using the rand() function in the C programming environment is 1) biased to have more bits with value “1” and
2) biased with more bits with value “0.” The Mitchell algorithm calculation with the biased schemes had an average error percentage of 3.4 and 3.3 for biased “1” and biased “0” schemes, respectively. The OD-Mitchell algorithm’s average error percentages for the biased “1” and biased “0” schemes were 1.9 and 1.8, respectively. Hence, the biased inputs had changed the average error percentage, but the improvement of operand decomposition method was once again close to 45 percent.

4.2 Combination of OD with Other Correction Approaches

In this section, we present the results of combining the proposed operand decomposition with previously published error correction approaches to logarithmic multiplication.

4.2.1 Divided Approximation

The operand decomposition (OD) when combined with previous error correction approaches improves the average error percentage and also the percent error range. We show the error spread of 500 random input multiplications for divided approximation and OD-divided approximation methods in Figs. 11 and 12, respectively. In this case, the errors are more evenly distributed above and below zero, hence giving the error curve a Gaussian shape. The average error percent also achieves good improvement because of this. The coefficients for the linear equations in divided approximation were suitably selected to reduce the average error percentage and the percent error range optimally. Table 6 shows the percentage of errors below a certain value for divided approximation. The average error percentage for the divided approximation is shown in Table 7. It can be observed that, while DA has an average error percent ranging from 2.45 percent to 2.71 percent, the OD-DA method reduces the average error percent to within 0.10 percent to 0.17 percent. The negative sign in the average error percent for DA indicates that the error occurs more often on the negative side of the actual logarithm. Once again, the operand decomposition method shows improvement in both of the error metric calculations.

4.2.2 Correction Term-Based Methods

The results on the error spread for the table of correction values and the OD-table of correction values are shown in Figs. 13 and 14, respectively. Mitchell’s error correction term-based approach has the best error range and is mostly positive. Figs. 15 and 16 show the error spread for Mitchell’s correction term (MEC) and the OD-Mitchell’s correction term. Finally, the error percent range and the average error percentage for these two correction term-based approaches are shown in Tables 6 and 7, respectively.

4.3 Image Processing Filters

The kernel elements in the Gaussian smoothing filters have a reasonably weighted element set, which is important to show the effectiveness of the operand decomposition approach. Since, with convolution kernels having small weights, the decomposition results in a single multiplication as one of the operands becomes zero and, hence, the operation requires only a single application of Mitchell’s algorithm. The Gaussian smoothing operation is a two-dimensional convolution-based filter, similar to the mean filter, but uses a kernel matrix that represents the shape of a Gaussian (bell-shaped hump), which detects edges more accurately. A circularly symmetric Gaussian distribution in two dimensions has the form

![Graph](image-url)
where $\sigma$ is the standard deviation of the distribution.

In theory, the Gaussian is a continuous distribution and would require a large convolution kernel. However, it has been shown in practice that three standard deviations from the mean effectively approximates the Gaussian distribution. In this paper, we use a convolution kernel matrix for Gaussian distribution with a standard deviation ($\sigma$) of 1.0. Once a suitable kernel has been calculated, the Gaussian smoothing operation is performed using standard convolution methods. The Gaussian filter is also useful in some computational biology applications, as some cells in the visual pathways of the brain have an approximate Gaussian response.

4.3.1 Operand Decomposed Gaussian Smoothing

Fig. 17 shows the base image obtained from Heriot-Watt University [23] for evaluating the operand decomposition approach. The base image in Fig. 17 is corrupted with salt and pepper noise, which is also referred to as intensity spikes, speckle, or data drop-out noise in image filtering contexts. The noise can occur due to error in data transmission, dust in the camera lens, and faulty memory elements. The corrupted pixels are either set to the maximum value or have single bits flipped over. In some cases, single pixels are set alternatively to zero or to the maximum value, giving the image a salt and pepper-like appearance. The noise is usually quantified by the percentage of pixels which are corrupted. We introduced salt and pepper noise with a probability of 1 percent (i.e., the individual bits have been flipped with probability 1 percent), as shown in Fig. 17b.

The multiplications for the convolution operation during smoothing are performed using standard multiplication and the various proposed methods. The resulting images of the Gaussian smoothing operation for standard multiplication, basic Mitchell Algorithm, and Operand Decomposed Mitchell Algorithm alone are shown in Fig. 17c, Fig. 17d, and Fig. 17e, respectively. Here, we use mean square error (MSE) and average error percent (AEP) for showing the effectiveness of operand decomposition in the Gaussian smoothing. The error comparison for the proposed approach is shown in Table 8. It can be inferred from Table 8 that the proposed operand decomposition approach clearly improves mean square error and the average error percentage for these image processing filters. Further, the Operand Decomposed Divided Approximation method has a better error percentage than any other OD-combination of correction approaches for the Gaussian smoothing application. The average error percentage and the mean square error of the OD-DA method is seen to be as good as using the standard multiplication for the convolution operation. Hence, the proposed operand decomposition strategy combined with the Mitchell algorithm-based multiplication can be used in image processing applications without much loss of accuracy.

4.4 Hardware Overhead

A 32-bit multiplier based on the proposed Operand Decomposed Mitchell algorithm was implemented using...
CMOS technology. The multiplier consisted of an operand decomposition unit, four leading one detector units, four 32 X 5-bit ROM, six logarithmic shifters, and three ripple carry adders. The input multiplicands are decomposed and then given to the leading one detector circuit. The position of the leading 1-bit is passed on to the ROM, which outputs the characteristic part of the logarithm. The remaining fraction part is shifted by the logarithmic shifters and appended to the characteristic value. The MA-based logarithms are then added and the sum is shifted using a logarithmic shifter for antilogarithmic approximation. The decomposed products are then added to get the final result. The logarithm and antilogarithm calculation for the decomposed operands occur in parallel. Simulation from extracted netlists indicates that the circuit requires 3,800λ x 1,050λ of chip area, when compared to 4,320λ x 1,260λ for a 32-bit fixed point array multiplier implemented with the same configuration. The power consumption of the proposed OD-MA multiplier estimated with a clock frequency of 100 MHz and a supply voltage of 1.08 V is 85 μW, which is only 30 percent of the power consumed by a standard 32-bit fixed point array multiplier. The Operand Decomposition-based Mitchell multiplier almost doubles the area and power required when compared to the original Mitchell multiplier and the delay overhead is unaffected as the pipeline stage delay is determined by the logarithm approximation stage. However, a sequential implementation of the two decomposed multiplications will double the delay overhead and area increases only slightly (less than 4 percent of the whole unit) due to the operand decomposition hardware.

The divided approximation approach [1] requires only a simple shifter and adder and the hardware overhead for this approach is small since we employ only a two region correction scheme and three mantissa bits for correction. The average error percentage of the operand decomposed divided approximation stage is around 0.15 percent. The OD-TCV and OD-MEC approach reduces the average error percentage to 0.02 percent and 0.03 percent, respectively, but the area and power overheads for these correction approaches are significantly higher compared to the divided approximation-based correction. The TCV and MEC approach required 50 percent more hardware when compared to the Mitchell logarithm-based algorithm and the 2-region divided approximation with three mantissa bits required less than 2 percent hardware. Hence, a particular OD-combination algorithm can be used depending on the accuracy and overhead requirement of the user.

5 CONCLUSIONS

Mitchell’s algorithm-based logarithm multiplication [15] is desirable for digital signal processing applications due to its low overhead property. However, the piecewise straight line approximation in Mitchell’s algorithm has a high error
percentage. In this paper, we described a novel approach to improve the accuracy in Mitchell-based logarithmic multiplication using operand decomposition. A slightly modified operand decomposition was used in [9] to reduce the switching transitions in binary multiplication. The operand decomposition approach improves the average error percentage and the error range of Mitchell algorithms by around 45 percent. The operand decomposition approach also combines well with previously proposed error correction methods. Efficient algorithms were developed to optimally combine operand decomposition with divided approximation, table of correction values, and Mitchell’s error correction equations method. New powers of two coefficients are developed for correction equations in divided approximation. Similarly, we also develop a new set of correction values for the table of correction values approach for minimizing the error range and the average error percentage of the multiplication algorithm. Thus, we conclude that operand decomposition is a powerful means to achieving both accuracy and efficiency in logarithm-based arithmetic.

REFERENCES


V. Mahalingam received the BE degree in computer science and engineering from Sri Venkateswara College of Engineering (SVCE), University of Madras, India, in 2003 and the master’s degree in computer engineering from the University of South Florida, Tampa, in 2005, where he is currently working toward the PhD degree in computer engineering. In 2002-2003, he worked as a part-time research assistant at Waran Research Foundation (WARFT), Chennai, India. His research interests include design automation, low power logic synthesis, uncertainty aware optimization, and VLSI testing. He is a student member of the IEEE and the IEEE Computer Society.

Nagarajan Ranganathan (S’81-M’88-SM’92-F’02) received the BE (Honors) degree in electrical and electronics engineering from the Regional Engineering College, Tiruchirapalli, University of Madras, India, in 1983 and the PhD degree in computer science from the University of Central Florida, Orlando, in 1988. He is currently a professor in the Department of Computer Science and Engineering at the University of South Florida, Tampa. During 1998-1999, he was a professor of electrical and computer engineering at the University of Texas at El Paso. His research interests include VLSI system design, design automation, energy and power optimization, biomedical information processing, crisis management and homeland security applications. He has developed many special purpose VLSI systems for computer vision, image processing, pattern recognition, data compression and signal processing applications. He has published more than 200 papers in reputed journals and conferences and is a co-owner of five US patents. Dr. Ranganathan is a fellow of the IEEE and a member of the IEEE Computer Society and the IEEE Circuits and Systems Society. He served on the editorial boards for the journals Pattern Recognition (1993-1997), VLSI Design (1994-present), IEEE Transactions on VLSI Systems (1995-1997), IEEE Transactions on Circuits and Systems (1997-1999), and the IEEE Transactions on Circuits and Systems for Video Technology (1997-2000). He was the chair of the IEEE Computer Society Technical Committee on VLSI during 1997-2001. He served as the steering committee chair of the IEEE Transactions on VLSI Systems during 2001-2002 and is serving as the editor-in-chief for 2003-2006. He was elected as fellow of IEEE in 2002 for his contributions to algorithms and architectures for VLSI Systems. He has received three best paper awards and the Theodore and Vanette Askounes-Ashford Distinguished Scholar Award from the University of South Florida, Tampa.