This is to certify that the

thesis entitled

EFFICIENT ALGORITHMS FOR GENERATING
HIGH PRECISION BINARY LOGARITHMIC NUMBERS

presented by

Yi Wan

has been accepted towards fulfillment
of the requirements for

MS degree in Electrical Eng

Major professor

Date Oct 20, '97

Efficient Algorithms for Generating High Precision Binary Logarithmic Numbers

by

Yi Wan

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering
1997

Abstract

Efficient Algorithms for Generating Precise Binary Logarithmic Numbers

by

Yi Wan

Two algorithms for fast evaluation of binary logarithm are developed in this paper. They provide advantages over other recent work on this problem. The first algorithm — Two Layer Factorization Algorithm (TLFA), uses a division and a newly discovered nonlinear approximation method to provide an improvement over the often quoted difference grouping algorithm and other recent work on this problem, both in speed and implementation structure. The second algorithm — Continuous Factorization Algorithm (CFA), is proposed to suit high speed and high precision conversions. It uses a continuous factorization process to reduce the problem to a division one and employs the SRT division technique to enhance speed. The resulting conversion time is only about three additions irrespective of word length. The error analysis and implementation structure are developed for this algorithm.

ACKNOWLEDGMENT

I would like to thank Dr. Chin-Long Wey for his encouragement, support, and care, and many of his insights I got during our discussions together.

Contents

List of Figures

List of Tables

1 Introduction

Logarithmic number system (LNS) is an attractive alternative to the conventional number system when the data need to be manipulated at very high rate over a wide data range. The LNS can simplify multiplication, division, root, and power operations [2]. When logarithm is used, multiplication and division are reduced to addition and subtraction, and power and root operations are reduced to multiplication and division. On the other hand, addition and subtraction operations become more complex. Another major problem is deriving logarithms and anti-logarithms quickly and accurately enough to allow conversions to and from the conventional number representations. These conversions always involve slow-speed approximations. Therefore, binary logarithms can be useful only in arithmetic units dedicated to special applications, where very few conversions are required but many multiplications and divisions are executed; e.g., real-time digital filters [3]. The objective of this thesis is to develop efficient algorithms that convert the conventional number representation to binary logarithmic representation. More specifically, given a non-zero binary number X, evaluate $\log X$. Note that such an X can always be normalized as

$$X = 1.x_1x_2\ldots x_n \times 2^k$$

for some integer k and then

$$\log X = \log(1 + .x_1 x_2 \ldots x_n) + k$$

Therefore, this conversion problem can be re-stated as:

Given an n-bit fractional binary number

$$x = .x_1 x_2 \ldots x_n$$

how to efficiently generate the precise binary logarithmic value of $(1 + x)$, i.e.,

$$y = \log(1 + x) \tag{1}$$

Here $y = .y_1 y_2 .. y_n$ is also an n-bit fractional binary number, i.e., $y \in [0, 1)$.

In the LNS, for logarithmic addition, let

$$a = \log(A) \qquad \text{and} \qquad b = \log(B)$$

i.e., $A = 2^a$ and $B = 2^b$.

Suppose $A < B$. Let $r = A/B$, then $0 \leq r < 1$ and

$$A + B = A(1 + \frac{A}{B})$$
$$= A(1 + r)$$

hence

$$\log(A + B) = \log[A(1 + r)]$$
$$= \log(A) + \log(1 + r)$$
$$= a + \log(1 + r)$$

The problem of addition in LNS then involves the following important step:

Given an n-bit binary number $r \in [0, 1)$, it is to generate the binary logarithmic value of $(1 + r)$.

Therefore, the efficient algorithms developed for conversion in Eq. 1 can also be used for the logarithmic addition problem. Recently a number of new binary logarithmic conversion algorithms, e.g., [1][2][4][5], have been proposed. The existing conversion algorithms can be roughly classified into two categories: look-up table approach and computation approach. The former approach adopts various approximation schemes such as linear

approximation methods so that moderate-size look-up table can be implemented [1]. Look-up table approach generally has the advantage of fast speed. However, the look-up table size often increases drastically as the word length increases. The latter approach converts the binary logarithm with multiplication and/or division operations [2][4]. However, the applicability of such approach is limited by its slow operation speed. The trade-off between both approaches is the speed and precision. Thus, a feasible solution is the combination of both approaches. This study develops two efficient algorithms using factorization scheme for binary logarithm conversion. The first algorithm – Two Layer Factorization Algorithm (TLFA), uses a factorization approach to reduce the look-up table size and employs a nonlinear approximation method to reduce the computational complexity. Simulation results on IEEE single precision (23 bits) conversion shows that the algorithm requires only a reasonable small-size ROM. For handling long word length numbers, such as IEEE double precision (53 bits), the second algorithm – Continuous Factorization Algorithm (CFA) is developed. It uses a continuous factorization process to further reduce the look-up table size and incorporates the SRT division technique [2] to reduce computational complexity.

In the next chapter, some recent conversion algorithms are briefly reviewed. Chapter 3 presents the proposed conversion algorithms with experimental results. Finally, a concluding remark is given in Chapter 4.

2 Background

This chapter briefly reviews some existing conversion algorithms. The basic easy method in the look-up table approach is the use of a simple linear function $y = x$ to approximate $y = \log(1 + x)$, as shown in Fig. 1 [1].

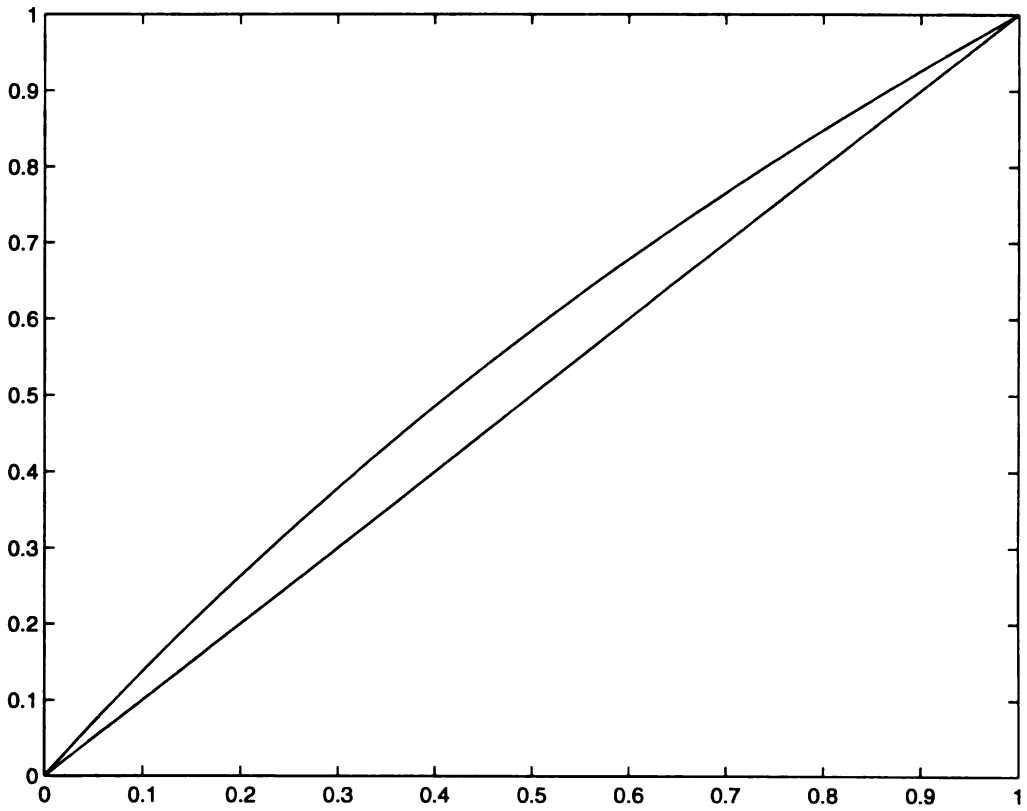

Figure 1: Graphs of $y = x$ and $y = \log(1 + x)$

In this scheme, if we let the approximation error

$$\Delta = \log(1 + x) - x$$

then the maximal approximation error $\Delta_{max} = 0.086071$ occurs at $x = 0.442695$. The number of different groups is defined by $\lceil 2^n \cdot \Delta_{max} \rceil$, where $\lceil * \rceil$ is the ceiling function and n is the word length. Thus, it requires 23 different groups for 8-bit word length. The conversion uses a ROM with 8 inputs and 5 outputs. However, the number of different groups increases exponentially as the word length increases. So, even though this algorithm has very fast speed, it is only suitable for very low conversion precision and hence has very limited use.

A multiplicative normalization algorithm is developed in [2] which uses a relative simple convergence procedure for the conversion process. The recursive procedure is:

At step i,

$$x_{i+1} = x_i \cdot b_i \qquad \text{and} \qquad y_{i+1} = y_i - \log(b_i)$$

The initial conditions are $x_0 = x$ where x is a fraction and $y_0 = 0$.

For the ease of using look-up table, each b_i is chosen as either 1 or $(1 + 2^{-i})$. It shows that if $x_{n+1} = 1$, then

$$\begin{aligned} y_{n+1} &= \log(x) \\ &= -\sum_{k=1}^{n} \log(b_k) \end{aligned}$$

In fact, each y_i is a truncated conversion result of i-bit length. The algorithm uses a small look-up table to store the values $\log(1 + 2^{-i})$. In addition, only the $\log(b_i), i = 1, 2, \ldots, n$, need to be stored for $2n$-bit word length. However, this algorithm needs to compute the multiplication $x_i \cdot b_i$ and the subtraction $y_i - \log(b_i)$. The former operation can be simplified using shift-add operations because b_i is either 1 or $(1 + 2^{-i})$. Carry propagation is a speed-slowing factor in this approach. The latter operation can be improved using carry-save adder.

An ATA (add-table look up-add) method [4] uses the truncated Taylor series and a difference grouping method. Suppose the function f is smooth and the 24-fraction-bit input

$$X = x_0 + \lambda x_1 + \lambda^2 x^2 + \lambda 3 x^3$$

where $\lambda = 2^{-6}$ and $x_i < 1$, $i = 0, 1, 2, 3$, are 6-bit in length each.

The Taylor series of $f(X)$ is

$$f(X) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0 + \lambda x_1)}{n!} (\lambda^2 x_2 + \lambda^3 x_3)^n$$

which, after truncating the high order terms and further expanding the second term, can be approximated by

$$\begin{aligned} \tilde{f}(X) = & f(x_0 + \lambda x_1) + \\ & \frac{\lambda}{2}\{f(x_0 + \lambda x_1 + \lambda x_2) - f(x_0 + \lambda x_1 - \lambda x_2)\} + \\ & \frac{\lambda^2}{2}\{f(x_0 + \lambda x_1 + \lambda x_3) - f(x_0 + \lambda x_1 - \lambda x_3)\} + \\ & \lambda^4\{\frac{x_2^2}{2} f^{(2)}(x_0) - \frac{x_2^3}{6} f^{(3)}(x_0)\} \end{aligned}$$

The computation of $\tilde{f}$ needs only add/sub/shift operations, including at least 5 full-length and 4 half-length addition operations. The paper mentions the conversion error of less than 2^{-29}, which means the last term

$$\lambda^4\{\frac{x_2^2}{2} f^{(2)}(x_0) - \frac{x_2^3}{6} f^{(3)}(x_0)\}$$

in the expression of $\tilde{f}(X)$ cannot be omitted. In this case more look-up tables or multiplications are needed.

In [5], digit partition (DP) is used to divide a long word and Iterative Difference by Linear Approximation (IDLA) is used to compute an exponential function needed in the conversion.

Suppose that a fractional input

$$x = 0.x_0x_1 \ldots x_{22}$$

of 23-bit length produces a 23-bit output

$$y = 0.y_0y_1 \ldots y_{22}$$

The output is partitioned as

$$\begin{aligned} y &= 0.y_0y_1 \ldots y_{11}y_{12} \cdots y_{22} \\ &= 0.YZ \end{aligned}$$

where Y represents the first 12 bits $y_0y_1 \ldots y_{11}$ and Z the remaining bits $y_{12}y_{13} \cdots y_{22}$.

By experiment or simple analysis proof, Y is determined only by $x_0x_1 \ldots x_{12}$. Hence a ROM with 13 inputs can be used to store the first 12-bit output Y. The remaining 11-bit output $Z = y_{12}y_{13} \ldots y_{22}$ is

calculated as follows.

$$2^{0.0..0Z} = (1.x_0x_1\ldots x_{12}) \times (2^{-0.Y}) + (0.0\ldots 0x_{13}x_{14}\ldots x_{22}) \times 2^{-0.Y}$$

In the above expression, the term $(1.x_0x_1\ldots x_{12}) \times (2^{-0.Y})$ can be obtained by a PLA with 13-bit input $x_0x_1\ldots x_{12}$, and the second term can be rewritten as

$$(0.00\ldots 0x_{13}x_{14}\ldots x_{22}) \times 2^{-0.Y} = 2^{\log(0.x_{13}x_{14}\ldots x_{22})-13-0.Y}$$

and eventually

$$2^{0.0\ldots 0Z} = (1.x_0x_1\ldots x_{12}) \times (2^{-0.Y}) + 2^{-13} \times 2^{\log(0.x_{13}x_{14}\ldots x_{22})-0.Y}$$

The output bits $Z = y_{12}y_{13}\ldots y_{22}$ can be evaluated by the IDLA algorithm.

However, the exponential evaluation is a time-consuming process, and a full-length addition is needed for summing up the right-hand-side of the last equation. In addition, we still need to perform a logarithmic conversion to determine Z eventually.

A simple linear approximation plus second stage piecewise linear approximation is developed in [7].

Suppose $y = y_1 + y_2$ is a 23-bit fraction, where y_1 is the first 11-bit part and y_2 is the last 12-bit part of y, then

$$\log(1 + y) \approx y + E_y \pm \Delta E_y \cdot y_2$$

where

$$E_y = \log(1 + y_1) - y_1$$

and

$$\Delta E_y = E_y(y_1 + 2^{-11}) - E_y(y_1)$$

This algorithm requires two look-up tables to store E_y and ΔE_y. In addition, it also requires a 12-bit multiplication which may degrade the speed performance.

A direct computation method for converting $y = \log(x)$ is proposed in [8].

Suppose

$$\log x = y$$

$$= y_{n-1}y_{n-2}\cdots y_1 y_0 . y_{-1}y_{-2}\cdots$$

then

$$\begin{aligned} x &= 2^{y} \\ &= 2^{y_{n-1}y_{n-2}\cdots y_1 y_0 \cdot y_{-1}y_{-2}\cdots} \\ &= 2^{y_{n-1}\cdot 2^{n-1}} \cdot 2^{y_{n-2}\cdot 2^{n-2}} \cdots 2^{y_1 \cdot 2} \cdot 2^{y_0} \cdot 2^{y_{-1}\cdot 2^{-1}} \cdots \end{aligned}$$

The developed algorithm starts with $i = n - 1$ and does the following recursively

If $x \geq 2^{2^i}$, then $y_i = 1,\ x = x \times 2^{-2^i}$.

Otherwise, $y_i = 0$.

The procedure is repeated by setting $i = i - 1$. In this scheme, both the comparison and the multiplication can be efficiently operated for $i \geq 0$. The former compares the 2^ith bit of x, while the latter shifts x to the left by 2^i bits. However, for $i < 0$, 2^i is a fraction and 2^{2^i} involves root operation, which is considered too time consuming. This problem can be solved by using look-up table to store $\{2^{2^{-1}},\ 2^{2^{-2}}, \ldots\}$. But a full length subtraction is still needed here for comparison. Since $2^{-2^i} = 1/2^{2^i}$, a division is needed in the multiplication step $x = x \times 2^{-2^i}$. Again to avoid this division another look-up table has to be used. Then a full length

multiplication has to be performed to get the product, which seriously degrades the speed performance.

As can be seen from the above review, current work on this conversion problem is not very satisfactory in dealing with conversion speed and high precision even though some involve complicated structure. The work developed in the next chapter attempts to make further improvement on these problems.

3 Algorithm Development

This chapter presents the development of both TLFA and CFA for binary logarithmic conversion. The former uses a two-layer factorization approach, while the second algorithm employs a continuous factorization process for long word conversion. We first propose the TLFA, then develop the CFA.

3.1 Two Layer Factorization Algorithm

This section describes the development of an efficient algorithm that generates the precise binary logarithm $\log(1+x)$ for an n-bit fractional binary number $x = .x_1x_2\ldots x_n$. Without loss of generality, n is assumed to be an even number, i.e., $n = 2m$ for some integer m. The term 1+x can be factorized as

$$\begin{aligned} 1+x &= 1+.x_1x_2\ldots x_n \\ &= 1+.x_1x_2\ldots x_m x_{m+1}x_{m+2}\ldots x_{2m} \\ &\approx (1+.x_1x_2\ldots x_m)(1+.00\ldots0c_1c_2\ldots c_m) \\ &= (1+.x_1x_2\ldots x_m)(1+.c_1c_2\ldots c_m\cdot 2^{-m}) \end{aligned} \tag{2}$$

Let

$$a = .x_1 x_2 \ldots x_m$$

$$b = .x_{m+1} x_{m+2} \ldots x_{2m}$$

$$b' = b \cdot 2^{-m}$$

$$c = .c_1 c_2 \ldots c_m$$

$$c' = c \cdot 2^{-m}$$

then $(1 + x)$ can be expressed as

$$\begin{aligned} 1 + x &= 1 + a + b' \\ &= (1 + a)(1 + c') \end{aligned} \tag{3}$$

By Eq. 3, c can be computed as

$$c = \frac{b}{1 + a} \tag{4}$$

then the logarithmic value of (1+x) is derived as

$$\log(1 + x) \approx \log(1 + a) + \log(1 + c') \tag{5}$$

ROM look-up table approach has been an efficient way to generate binary logarithms. For generating the logarithm of an n-bit binary number, a look-up table can be implemented with a $2^n \times n$ ROM.

By Eq. 5, two look-up tables need to be used:

one table – ROM1, for

$$\log(1 + .x_1 x_2 \ldots x_m)$$

and the other – ROM2, for

$$\log(1 + .00...0 c_1 c_2 \ldots c_m)$$

where each table is implemented with at most a $2^m \times n$ ROM.

In other words, instead of using a $2^n \times n$ ROM in the conventional implementation, this approach uses two $2^m \times n$ ROMs, whose total size is only a small fraction of the original one, namely, $\frac{1}{2^{m-1}}$.

For example, for $n = 16$ and $m = 8$, the ROM table size is reduced from $2^{16} \times 16$, or 1M bits, to two $2^8 \times 16$, or a total of 8K bits. The reduction is significant. Table 1 summarizes the ROM size reduction for various values of n.

n	m	$2^n \cdot n$	$2 \cdot 2^m \cdot n$
10	5	10K	640
12	6	48K	1.5K
14	7	112K	3.5K
16	8	1M	8K
18	9	4.5M	18K
20	10	20M	40K

Table 1: TLFA ROM size comparison

However, the above approach requires the computation of c as in Eq. 4, which involves a slow division process. As a result, the speed improvement gained by the use of look-up tables for Eq. 5 may be offset by the slow division process. Therefore, attempts are made to get around the division in Eq. 4 to improve the speed performance. Taking advantage of the existing look-up table ROM1 used for $\log(1 + .x_1x_2 \ldots x_m)$, the slow division process

can be simplified. More specifically, by Eq. 4, c can be expressed as

$$\begin{aligned} c &= \frac{b}{1+a} \\ &= \frac{1+b}{1+a} - \frac{1}{1+a} \\ &= 2^{\log(1+b)-\log(1+a)} - 2^{-\log(1+a)} \\ &= 2^{B-A} - 2^{-A} \end{aligned} \tag{6}$$

where $A = \log(1+a)$ and $B = \log(1+b)$.

Note that the values of both A and B can be quickly retrieved from ROM1. Therefore, the only problem remaining is the evaluation of the function 2^z.

Interestingly, the functions $y = (2^z - 1)$ and $y = 1 - \log(2-z)$, as shown in Fig. 2, are very close to each other when $0 \leq z \leq 1$. i.e.,

$$2^z - 1 \approx 1 - \log(2-z)$$

or

$$2^z \approx 2 - \log(2-z) \tag{7}$$

In other words, the function 2^z can be approximated by the nonlinear function $2 - \log(2-z)$ for all $z \in [0,1]$.

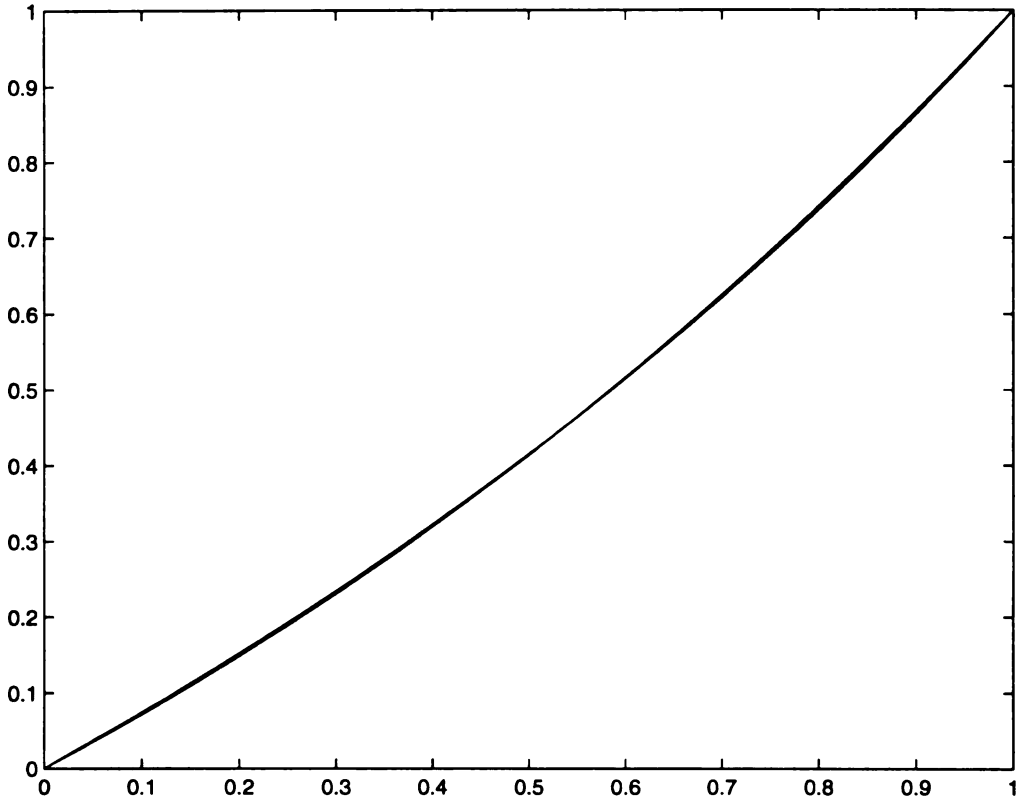

Figure 2: The graphs of $2^z - 1$ and $1 - \log(2 - z)$

Since

$$\log(2 - z) = \log[1 + (1 - z)],$$

its value can be found from the look-up table ROM1 because $(1 - z) \in [0, 1]$ for all $z \in [0, 1]$.

Fig. 3 plots the approximation error between $2^z - 1$ and $1 - \log(2 - z)$ for all $z \in [0, 1]$. Results show that the maximum error is 0.004198, which is much better than 0.086071 with a linear function approximation in [1]. Note that the maximum approximation error of 0.004198 is equivalent to 7 bit accuracy. To further reduce the maximum approximation error, we use the function

$$1 - \log(2 - z) + 2^{-12} + 2^{-13} \tag{8}$$

to approximate $2^z - 1$.

As such, the maximum approximation error can be reduced to 0.0038038, which is less than 2^{-8}. So $m = 8$ is the appropriate value suggested with this implementation.

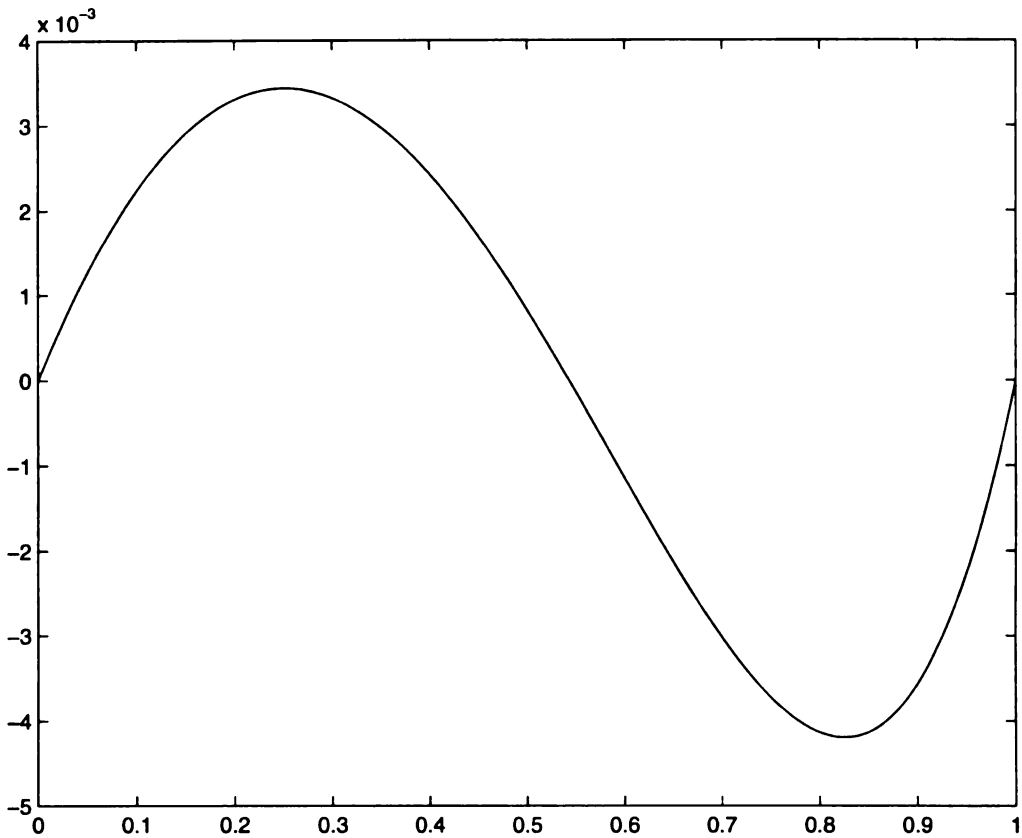

Figure 3: Difference between $2^z - 1$ and $1 - \log(2 - z)$

It should be mentioned that, since the approximation to 2^z in Eq. 7 is valid only for all $z \in [0, 1]$, the exponents in both terms of Eq. 6 must be ranged between 0 and 1.

Notice that A and B as in Eq. 6 are in the range of $[0, 1]$. Hence Eq. 6 is re-written as

$$c = 2^{B-A} - 2^{1-A}/2 \qquad \text{if } A < B \tag{9a}$$

or

$$c = 2^{1+B-A}/2 - 2^{1-A}/2 \qquad \text{if } A \geq B \tag{9b}$$

In Eq. 9a, by Eq. 7, the second term 2^{1-A} is approximated by the following

$$\begin{aligned} 2^{1-A} &\approx 2 - \log(1 + A) \\ &= 2 - A' \end{aligned} \tag{10}$$

where $A' = \log(1 + A)$

and the first term 2^{B-A} is approximated by

$$2^{B-A} \approx 2 - \log(2 - (B - A))$$

Let $p = 1 - B + A$, then $p < 1$ and

$$2 - \log(2 - (B - A)) = 2 - \log(1 + p)$$

Similarly, the first term 2^{1+B-A} in Eq. 9b is approximated by

$$\begin{aligned} 2^{1+B-A} &\approx 2 - \log(2 - (1 + B - A)) \\ &= 2 - \log(p) \end{aligned}$$

where $p \geq 1$ because $A \geq B$ in Eq. 9b.

Denote $p = p_0 + p_f$, where p_0 and p_f are the integral and fractional parts of p respectively. If $p \geq 1$, then $p_0 = 1$ and $p - 1 = p_f$. On the other hand, if $p < 1$, then $p_0 = 0$ and $p = p_f$.

In Eq. 9a, $A < B$, or $p < 1$,

$$\begin{aligned} 2^{B-A} &\approx 2 - \log(1 + p) \\ &= 2 - \log(1 + p_f) \\ &= 2 - P \end{aligned}$$

where $P = \log(1 + p_f)$.

From Eq. 10,

$$c = 2^{B-A} - 2^{1-A}/2$$

$$\approx (2 - P) - (2 - A')/2$$

i.e.,

$$c = (1 - P) + A'/2 \qquad \text{if } p_0 = 0 \tag{11a}$$

Similarly, for $A \geq B$, or $p \geq 1$, in Eq. 9b,

$$2^{1+B-A} \approx 2 - \log(p)$$

$$= 2 - \log(1 + p_f)$$

$$= 2 - P$$

From Eq. 10,

$$c = 2^{1+B-A}/2 - 2^{1-A}/2$$

$$\approx (2 - P)/2 - (2 - A')/2$$

i.e.,

$$c = (-P)/2 + A'/2 \qquad \text{if } p_0 = 1 \tag{11b}$$

Note that the above simplification makes the comparison of A and B unnecessary in the implementation and hence saves a subtraction of length m. Algorithm 1.1 summarizes the procedure described above.

Algorithm 1.1

Step 0. Generate look-up tables, $2^m \times n$,

ROM1(x) for $\log(1.x_1x_2 \ldots x_m)$ and

ROM2(c) for $\log(1.00...0c_1c_2 \ldots c_m)$

$$x = .x_1x_2 \ldots x_m x_{m+1} x_{m+2} \ldots x_{2m}$$

$$= a + b \cdot 2^{-m}$$

$$c = .c_1c_2 \ldots c_m$$

Step 1. $A = \log(1 + a) = \text{ROM1}(a)$

$B = \log(1 + b) = \text{ROM1}(b)$;

Step 2. Calculate c from Eq. 11a or Eq. 11b

2.1 $p = 1 - B + A = p_0 + p_f$.

2.2 $P = \text{ROM1}(p_f)$; $A' = \text{ROM1}(A)$;

2.3 IF $p_0 = 0$, THEN $c = 1 - P$; ELSE $c = (-P)/2$;

2.4 $c = c + A'/2$.

Step 3. $C = \log(1 + c \cdot 2^{-m}) = \text{ROM2}(c)$;

$$\log(1 + x) = \log(1 + a) + \log(1 + c \cdot 2^{-m})$$

$$= A + C$$

The following examples illustrate the stepwise procedure of Algorithm 1.1.

Example 1.1

Consider a 16-bit binary number $x = .10111011\ 11101010$.

$$a = .10111011 \text{ and } b = .11101010$$

From ROM1,

$$A = \log(1 + a) = .11001011$$

$$A' = \log(1 + A) = .11011000$$

and

$$B = \log(1 + b) = .11110000$$

By Eq.11a,

$$p = 1 - \log(1 + b) + \log(1 + a)$$

$$= .11011011$$

Note that $p_0 = 0$ *and* $p_f = p$.

Since $p_0 = 0,\ c = .10001000.$

From ROM2,

$$C = \log(1 + c \cdot 2^{-m})$$

$$= \text{ROM2}(c)$$

$$= .0000000011000100$$

This results in

$$\log(1 + x) = .1100101101001110$$

The actual value is $\log(1 + x) = .1100101101001101$. *The approximation error is 1ulp (unit in the last position).*

Example 1.2

Consider again a 16-bit binary number $x = .10110100\ 01011011.$

$$a = .10110100 \text{ and } b = .01011011$$

From ROM1,

$$A = \log(1 + a) = .11000101$$

$$A' = \log(1 + A) = .11010011$$

and

$$B = \log(1 + b) = .01110000$$

By Eq. 11b,

$$p = 1 - \log(1 + b) + \log(1 + a)$$

$$= 1.01010101$$

In this case $p_0 = 1$ *and* $p_f = p - 1$.

Since $p_0 = 1$, $c = .00110100$.

By ROM2,

$$C = \log(1 + c \cdot 2^{-m})$$

$$= \text{ROM2}(c)$$

$$= .0000000001001011$$

This results in

$$\log(1+x) = .1100010011110011$$

The actual value of $\log(1+x) = .1100010011110101$. *The approximation error is -2ulp.*

To demonstrate the effectiveness of algorithm 1.1, approximation errors have been analyzed by comparing the approximated value with the actual value of $\log(1+x)$. The simulation assumes that $m = 8$ and $n = 16$. Hence the input fraction takes the form of $x = .x_1x_2 \ldots x_{16}$. All possible $(2^{17} - 1)$ combinations of $x_1x_2 \ldots x_{16}$ are simulated for approximated values and compared with the true values. Let $DIFF$ denote the absolute value of the difference between the approximated value and the actual conversion value. Simulation results are shown in Table 2.

It can be seen that the maximum approximation error is $4ulp$, or 2^{-14}. The number of combinations with $DIFF = 2^{-14}$ is 27 out of $(2^{17} - 1)$, which is 0.041%. Because of nonlinear approximation error and look-up table truncation, the computation of c as described in the algorithm can have an

$DIFF(ulp)$	Number of Cases	Percentage
0	22426	34.2
1	31388	47.9%
2	10417	15.9%
3	1278	1.95%
4	27	0.041%
≥ 5	0	0

Table 2: Simulation result of TLFA 1 with $n = 16$

error of up to 1-2*ulp*, this error can be magnified by a factor of $(1 + a)$ as in Eq.3, which can then be further carried into the final stage of taking logarithm from the look-up tables.

Since none of the conversion results of these combinations produce $DIFF > 2^{-14}$, in other words, all $DIFF \leq 2^{-13}$. This concludes that the developed algorithm achieves an accuracy of 13 bits.

The two tables — ROM1 and ROM2, are used and each has a size of

$2^8 \times 16$, or 4k bits. Note that

$$\log(1+x) < 2x$$

for all $x > 0$.

It follows that the first 7 bits of the output of ROM2 must be 0. So the size of $2^8 \times 9$, or 2.25k bits is actually enough for ROM2.

It should be mentioned that the slow division process in Eq. 4 is improved by approximating 2^z using the existing ROM table. However, the nonlinear approximation approach developed in this algorithm limits to $m = 8$ and the accuracy of $\log(1+x)$ to 13 bits. The accuracy can be improved by either selecting better nonlinear approximation functions, or alternatively, using additional ROM table for the values of 2^z, where $z \in [0, 1)$.

The first alternative seems difficult due to the fact that any nonlinear approximation function to the function 2^z itself need to be effectively evaluated first. In this study a connection between 2^z and available logarithmic function is discovered and hence there is not much hardware overhead involved. But in general, it's not easy to find a very convenient approximation function. As discussed in chapter 2, linear function and

truncated Taylor series have been tried before but are not very satisfactory because of either low precision or heavy computation involved in the evaluation of the approximation function.

Difference grouping technique can be used in the second alternative. Let ROM3 be the look-up table for the difference between $2^z - 1$ and the nonlinear approximation function (8) with the size of $2^m \times m$ bits. Thus, the approximation of 2^z can be expressed as

$$2^z \approx 2 - \log(2 - z) + \Delta(z) \qquad \text{for } z \in [0, 1] \tag{12}$$

where $\Delta(z)$ is the difference error by the first stage nonlinear approximation and can be stored in ROM3.

The value c in Eq. 10 can then be calculated as

$$c = \begin{cases} (1 - P) + \Delta(1 - p) + \frac{A' + \Delta(1-A)}{2} & \text{if } A \leq B \\ \frac{-P + \Delta(2-p)}{2} + \frac{A' + \Delta(1-A)}{2} & \text{if } A > B \end{cases} \tag{13}$$

Algorithm 1.2 summarizes the procedure for this implementation. It has been developed and simulated for IEEE single precision conversion (23 bits)

with three internal guarding digits, making the total number of bits in a word to be 26, which corresponds to m=13. In this case, ROM1 has size $2^{13} \times 26$ bits, ROM2 has size $2^{13} \times 14$ bits since the first 12 bits of $\log(1+c')$ are all 0, and ROM3 has size $2^{13} \times 6$ since, by previous analysis, $\Delta(z)$ is less than 2^{-8} and hence only the last 5 bits plus a sign bit need to be stored.

Algorithm 1.2

Step 0. (ROM1(x) and ROM2(c) are the same as in Algorithm 1.1)

ROM3(x) for $2^x - 2 + \log(2-x)$ with size of $2^m \times m$

Step 1. (Same as Algorithm 1.1)

Step 2. Calculate c from Eq. 13

2.1 $p = 1 - B + A = p_0 + p_f$.

2.2 $P = \text{ROM1}(p_f)$; $A' = \text{ROM1}(A)$;

2.3 IF $p_0 = 0$

THEN $c = 1 - P + \text{ROM3}(1-p) + \text{ROM1}(1-A)/2$;

ELSE $c = (-P + \text{ROM3}(2-p) + \text{ROM3}(1-A))/2$;

2.4 $c = c + A'/2$.

Step 3. (Same as Algorithm 1.1)

Simulation results show that the maximum conversion error is 2^{-24} after truncating the last two guarding digits. Note that the look-up table size can be further reduced by using PLA and other look-up table compactification techniques. The use of PLA in [1] reduces the size to about 16% of the original ROM size, similar effect can also be achieved by using PLA in this scenario.

3.2 Continuous Factorization Algorithm

In the previous section, nonlinear approximation method is proposed to reduce the size of ROM for look-up table. The limitation of the scheme is that its reduction of ROM size is only one-step and hence is not suitable for high precision conversion. Limited by the look-up table size, a natural alternative to increase conversion speed is to use parallel processing. The Continuous Factorization Algorithm (CFA) is proposed that uses look-up table and SRT division technique [2] to achieve fast high precision conversion. We first formulate the problem in this approach.

For an n-bit fractional binary number

$$x = .x_1 x_2 \ldots x_n$$

we factorize $1 + x$ as

$$\begin{aligned} 1 + x &= 1 + .x_1 x_2 \ldots x_n \\ &\approx (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_n) \end{aligned} \tag{14}$$

where each q_i is either 0 or 1.

Then $\log(1 + x)$ can be approximated as

$$\log(1 + x) \approx \log(1 + .q_1) + \log(1 + .0q_2) + \ldots + \log(1 + .0 \ldots 0q_n) \tag{15}$$

Since each q_i is either 0 or 1, the term $\log(1 + .0 \ldots 0q_i)$ is either 0 or $\log(1 + .0 \ldots 01)$, which can be pre-stored in a look-up table. Therefore this approach employs a look-up table of at most $n \times n$ bits. For example, if n=64, then the ROM size is no more than $64 \times 64 = 4K$ bits. The question now is how to determine the q_i's.

In principle the factorization process of $1 + x$ can be carried out as follows.

First

$$\begin{aligned} 1 + x &= 1 + .x_1 x_2 \ldots x_n \\ &\approx (1 + .x_1)(1 + .0a_{1,2}a_{1,3} \ldots a_{1,n}) \end{aligned} \tag{16}$$

where $q_1 = x_1$ and

$$.a_{1,2}a_{1,3} \ldots a_{1,n} = \frac{.x_2 x_3 \ldots x_n}{1 + .x_1} \tag{17}$$

In a similar way,

$$1 + .0a_{1,2}a_{1,3} \ldots a_{1,n} \approx (1 + .0a_{1,2})(1 + .00a_{2,3}a_{2,4} \ldots a_{2,n}) \tag{18}$$

where $q_2 = a_{1,2}$ and

$$.a_{2,3}a_{2,4} \ldots a_{2,n} = \frac{.a_{1,3}a_{1,4} \ldots a_{1,n}}{1 + .0a_{1,2}} \tag{19}$$

In general, at step j, $j = 1, 2, \ldots, n-1$,

$$1 + .00\ldots0a_{j,j+1}a_{j,j+2}\ldots a_{j,n} \approx (1 + .00\ldots0a_{j,j+1}) \times (1 + .00\ldots0a_{j+1,j+2}a_{j+1,j+3}\ldots a_{j+1,n}) \quad (20)$$

where $q_j = a_{j,j+1}$ and

$$.a_{j+1,j+2}a_{j+1,j+3}\ldots a_{j+1,n} = \frac{.a_{j,j+2}a_{j,j+3}\ldots a_{j,n}}{1 + .00\ldots0a_{j,j+1}} \quad (21)$$

For example, consider $n = 6$ and $x = .111101$, By Eq. 21,

$$\begin{aligned}
1.111101 &\approx (1.1) \cdot (1.010011) && q_1 = 1 \\
1.010011 &\approx (1.01) \cdot (1.000011) && q_2 = 1 \\
1.000011 &\approx (1.000) \cdot (1.000011) && q_3 = 0 \\
1.000011 &\approx (1.0000) \cdot (1.000011) && q_4 = 0 \\
1.000011 &\approx (1.00001) \cdot (1.000001) && q_5 = 1 \\
1.000001 &\approx (1.000001) \cdot (1.000000) && q_6 = 0
\end{aligned}$$

As a result,

$$(1.1)(1.01)(1.000)(1.0000)(1.00001)(1.000001) = 1.11101111$$

In the next section we analyze the approximation error and ways to reduce look-up table size and computation steps. In that section we give a sufficient condition that guarantees the conversion precision. Following that we first develop a parallel processing structure that reduces the computation steps from $O(n^2)$ to $O(n)$, then we develop a more effective implementation algorithm which is based on the SRT division spirit.

3.2.1 CFA Approximation Error Analysis

In this section we analyze the various approximation errors in the CFA conversion scheme which is based on the factorization process. We assume the number of fractional bits is n and use *ulp* (unit of last position) to denote 2^{-n}. Also we assume that in any division if the word length of the divisor is less than that of the divident, then the divisor is truncated. This makes the hardware structure simpler.

Theorem 1. *Suppose the register has fractional length of n, then*

$$\frac{.0\ldots0a_{k,k+1}\ldots a_{k,n}}{(1+.0\ldots0q_k)}$$

gives an n-bit fraction with error less than ulp.

Proof. If $q_k = 0$, then clearly the error is 0. If $q_k = 1$, note that in the division process, the quotient bits are accurate until the last k fractional

bits when the divisor $1 + .0\ldots0q_k$ becomes 1 due to truncation. Hence

$$\begin{aligned}
\text{error} &= \left| \frac{.0\ldots0y_{n-k+1}y_{n-k+2}\cdots y_n}{1+.0\ldots01} - \frac{.0\ldots0y_{n-k+1}y_{n-k+2}\cdots y_n}{1} \right| \\
&= \frac{(.0\ldots0y_{n-k+1}y_{n-k+2}\ldots y_n)(.0\ldots01)}{1+.0\ldots01} \\
&< 2^{-(n-k)} \cdot 2^{-k} \\
&= 2^{-n} \\
&= ulp
\end{aligned}$$

□

Theorem 2. *For any fixed $n = 2m > 0$, the approximation error in the following*

$$1 + .0\ldots0x_{m+1}x_{m+2}\ldots x_{2m} \approx (1 + x_{m+1}2^{-m-1})(1 + x_{m+2}2^{-m-2})\ldots(1 + x_{2m}2^{-2m}) \quad (22)$$

is less than $\frac{1}{2}ulp$. Here each x_i is either 0 or 1.

Proof. Clearly the maximum approximation error occurs when $x_{m+k} = 1$ for all $1 \leq k \leq m$. We assume this and proceed as follows:

For $1 \leq k \leq m$, let r_k denote the remainder (error) of the product of the

first k terms on the right side of the Eq. 22, i.e.,

$$r_k = (1 + 2^{-m-1})(1 + 2^{-m-2}) \ldots (1 + 2^{-m-k})$$
$$- (1 + .\overbrace{0 \ldots 0}^{m}\overbrace{11 \ldots 1}^{k})$$

Note that r_m equals the maximum error stated in the theorem.

The first few r_k's are easy to compute. For example,

$$1 + 2^{-m-1} = 1 + 2^{-m-1} + 0$$
$$\Longrightarrow \quad r_1 = 0$$

and

$$(1 + 2^{-m-1})(1 + 2^{-m-2}) = (1 + 2^{-m-1} + 2^{-m-2}) + 2^{-2m-3}$$
$$\Longrightarrow \quad r_2 = 2^{-2m-3}$$

It can also be checked that

$$r_3 = 2^{-2m-3} + 2^{-2m-4} + 2^{-2m-5}$$
$$< 2^{-2m-2}$$

So the theorem holds for $m \leq 3$.

To estimate the bound on r_k in general, first note the following facts.

Fact 1: $1 + x < e^x$ for all $x > 0$.

Fact 2: $e^x < 2x + 1$ for all $x \in (0, \frac{1}{2}]$.

For $m \geq k \geq 4$,

$$
\begin{aligned}
& r_k - r_{k-1} \\
&= [(1 + 2^{-m-1})(1 + 2^{-m-2}) \dots (1 + 2^{-m-k}) - (1 + .\overbrace{0 \dots 0}^{m}\overbrace{11 \dots 1}^{k})] \\
&\quad - [(1 + 2^{-m-1})(1 + 2^{-m-2}) \dots (1 + 2^{-m-k+1}) - (1 + .\overbrace{0 \dots 0}^{m}\overbrace{11 \dots 1}^{k-1})] \\
&= (1 + 2^{-m-1})(1 + 2^{-m-2}) \dots (1 + 2^{-m-k+1})2^{-m-k} - 2^{-m-k} \\
&= 2^{-m-k}[(1 + 2^{-m-1})(1 + 2^{-m-2}) \dots (1 + 2^{-m-k+1}) - 1]
\end{aligned}
$$

The term $(1+2^{-m-1})(1+2^{-m-2})\dots(1+2^{-m-k+1})$ is estimated as follows,

$$\begin{aligned}
&(1+2^{-m-1})(1+2^{-m-2})\dots(1+2^{-m-k+1}) \\
&< \prod_{i=1}^{\infty}(1+2^{-m-i}) \\
&< \prod_{i=1}^{\infty} e^{2^{-m-i}} \\
&= e^{\sum_{i=1}^{\infty} 2^{-m-i}} \qquad \text{(by Fact 1)} \\
&= e^{2^{-m}} \\
&< 2^{-m+1}+1 \qquad \text{(by Fact 2)}
\end{aligned}$$

Thus

$$\begin{aligned}
r_k - r_{k-1} &= 2^{-m-k}[(1+2^{-m-1})(1+2^{-m-2})\dots(1+2^{-m-k+1})-1] \\
&< 2^{-m-k}\cdot 2^{-m+1} \\
&= 2^{-2m-k+1}
\end{aligned}$$

and it follows that for $m \geq 4$,

$$\begin{aligned} r_m &= r_3 + \sum_{k=4}^{m} (r_k - r_{k-1}) \\ &< r_3 + \sum_{k=4}^{\infty} (r_k - r_{k-1}) \\ &< r_3 + \sum_{k=4}^{\infty} 2^{-2m-k+1} \\ &= r_3 + 2^{-2m-2} \\ &< 2^{-2m-2} + 2^{-2m-2} \\ &< 2^{-2m-1} \\ &= \frac{1}{2} ulp \end{aligned}$$

□

Corollary 1. *For any $n = 2m$-bit fraction factorization as in Eq. 15, only the first m steps are needed and*

$$q_{m+k} = a_{m,m+k}, \ 1 \leq k \leq m$$

The error caused by this is less than ulp.

Proof. By theorem 2,

$$\begin{aligned} \text{error} &< (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_m) \cdot r_m \\ &< 2r_m \\ &< ulp \end{aligned}$$

□

Theorem 3. *For each $n = 2m$-bit fractional input x, at most $m(m-1)/2$ quotient bit operations are needed and the maximal error resulting from factorization is less than $n \cdot ulp$.*

Proof. From corollary 2, only m divisions are needed. Because of truncation at the end of each division, for $1 \leq k \leq m$, only $m - k$ quotient bit operations are needed. So the total number is

$$\sum_{k=1}^{m-1} k = m(m-1)/2$$

Note that the error caused by $\frac{1+.x}{1.1}$ is at most $\frac{1}{2}ulp < ulp$ and is less than $2ulp$ for each divisor $1 + .0 \ldots 0q_k,\ 2 \leq k \leq m$.

By theorem 2,

$$\begin{aligned} \text{total error} &< ulp + 2ulp(m-1) + ulp \\ &= 2m \cdot ulp \\ &= n \cdot ulp \end{aligned}$$

□

Corollary 2. *The final conversion error due to the factorization process is less than $\frac{n \cdot ulp}{\ln 2}$.*

Proof. Since the function log(1+x) is continuously differentiable for $x \in [0,1]$, for any x and y such that $0 \leq x < y \leq 1$,

$$\begin{aligned} \log(1+y) - \log(1+x) &\leq \underset{x \in (0,1)}{\text{Max}} \left| \frac{d \log(1+x)}{dx} \right| \cdot (y-x) \\ &= \frac{y-x}{\ln 2} \\ &< \frac{n \cdot ulp}{\ln 2} \end{aligned}$$

□

Corollary 3. *In order to achieve n-bit conversion accuracy, i.e., for each n-bit fractional input, the logarithmic conversion error is less than ulp, the following condition on the number of internal guarding digits g is sufficient.*

$$\log(n+g) - \log(\ln 2) < g$$

Proof. Note that the error caused by factorization is positive and the error caused by look-up table truncation is negative. So we can just restrict the error from factorization and error from look-up table truncation to be both less than *ulp*.

From Corollary 2, we have

$$\frac{n+g}{\ln 2} 2^{-(n+g)} < 2^{-n}$$

which leads to

$$\log(n+g) - \log(\ln 2) < g$$

For the look-up table truncation error, it is less than $(n+g) \cdot ulp$. So we have

$$(n+g)2^{-(n+g)} < 2^{-n}$$

which leads to

$$\log(n+g) < g$$

By comparing the above two conditions we have

$$\log(n+g) - \log(\ln 2) < g \tag{23}$$

□

Example 2.1

Suppose $n = 56$, then by Eq. 23 we can choose $g = 7$ to guarantee the conversion accuracy.

Note that the condition developed in this section is just sufficient and may not be optimal. Some estimates in the analysis can be further refined. For

example, it can be easily shown that not all q_i's can be 1 whenever $n > 3$; the bounds in the proof of Theorem 1 can also be lowered. Nevertheless, as shown in the above example, the number of guarding digits is not big, especially in long word case. So the result can be used in designing high precision conversion structure.

The next two theorems are developed to reduce the size of the look-up table used to store $\log(1 + 2^{-i})$, $i = 1, 2, \ldots$.

Theorem 4. *For each $i > 1$, $\log(1 + 2^{-i})$ is a fraction with (i-1) leading 0's.*

Proof. From the identity

$$\log(1 + x) < \frac{x}{\ln 2}$$

we have

$$\begin{aligned} \log(1 + 2^{-i}) &< \frac{2^{-i}}{\ln 2} \\ &< 2^{-(i-1)} \end{aligned}$$

□

Theorem 5. *Suppose $n = 2m$, then for $k > m$,*

$$\log(1 + 2^{-k}) \approx \frac{1}{2}\log(1 + 2^{-k+1})$$

Proof. From calculus technique, we have that for $x \in (0, \frac{1}{2}]$,

$$\begin{aligned}
&\left|\frac{1}{2}\log(1+x) - \log(1+\frac{x}{2})\right| \\
&= \left|\log\sqrt{1+x} - \log(1+\frac{x}{2})\right| \\
&= \left|\log\frac{\sqrt{1+x}}{1+\frac{x}{2}}\right| \\
&= \left|\log(1 - \frac{\frac{1}{8}(1+c)^{-\frac{3}{2}}}{1+\frac{x}{2}}x^2)\right| \quad \text{where } 0 < c < x \\
&< \frac{1}{2}x^2
\end{aligned}$$

Then for $x = 2^{-k}$ as in the theorem, we have that

$$\left|\frac{1}{2}\log(1 + 2^{-k}) - \log(1 + 2^{-k-1})\right| < 2^{-2k-1}$$

□

The implication of the above theorem is that only the first half of the look-up table for

$$\log(1 + .0\ldots01)$$

need to be stored. For the second half of the look-up table, the result is only a shift of the middle entry.

Example 2.2

The following table for $n = 8$ *shows that for* $k \geq 5$, $\log(1 + 2^{-k})$ *is just a right shift of* $\log(1 + 2^{-k+1})$, *as shown in the previous theorem.*

i	$\log(1 + 2^{-i})$
1	.10010101
2	.01010010
3	.00101011
4	.00010110
5	.00001011
6	.00000101
7	.00000010
8	.00000001

Table 3: 8-bit look-up table for $\log(1 + 2^{-i})$

From these two theorems we get the following corollary on the size of the look-up table.

Corollary 4. *For $n = 2m$ word length conversion, the logarithmic look-up table size can be as small as $\frac{m(3m+1)}{2}$ bits.*

Proof. This is a simple result from Theorems 4 and 5. □

Example 2.3

If n=64, then the look-up table size is at most

$$\frac{32 \cdot (3 \cdot 32 + 1)}{2} \quad bits$$

which is less than 1.5K bits.

In the following subsections we first propose a procedure based directly on the factorization process described before. Then we incorporate the SRT division technique to improve the speed.

3.2.2 Restoring Approach

To implement the factorization process described at the beginning of this section

$$1 + .x_1x_2 \ldots x_n = (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_n)$$

we can do n divisions with the divisor of the form $1 + 2^{-j}$ at each division step j. Or we can do n multiplications to change the divisor at step j to be of the form $(1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_j)$, which reduces the division to a single comparison.

In the first approach, because of the special form of the divisors, at each division step j, after certain number of quotient bits are determined, the next division process can start. The following example shows this in detail.

Example 2.4

Suppose a fraction

$$.a_{1,1}a_{1,2}\ldots = .x_1x_2\ldots$$

At step 1, we do the division

$$\frac{.0a_{1,2}a_{1,3}\cdots}{1.a_{1,1}} = .0a_{2,2}a_{2,3}a_{2,4}a_{2,5}\ldots$$

at step 2, we do the division

$$\frac{.00a_{2,3}a_{2,4}\cdots}{1.0a_{2,2}} = .00a_{3,3}a_{3,4}a_{3,5}a_{3,6}\ldots$$

Step 2 doesn't have to wait for the completion of step 1 to start. In fact, after $a_{2,5}$ *in step 1 is determined, step 2 can be carried out. Each additional determined quotient bit from step 1 immediately participates in step 2 division in a synchronized way. Similarly, step 3 can start after* $a_{3,7}$ *is determined in step 3.*

The parallel processing structure illustrated by the above example can increase speed dramatically in high precision situation. In fact, it can be proved that the number of total quotient bit operations is reduced from $O(n^2)$ to $O(n)$. However, this erases the recursive property of the algorithm in the hardware implementation and causes large array-like structure. So

the trade-off here has to be weighed depending on the application requirement.

In the rest of the section we develop the second implementation approach. As will be seen in later subsections, it allows the design of high speed implementation with still low hardware complexity.

We utilize the standard restoring division procedure [2] which uses comparison and subtraction operations. Define the partial quotient D_j and partial remainder R_j, $j = 1, 2, \ldots, n$, as

$$D_j = (1 + .q_1)(1 + .0q_2)\ldots(1 + .0\ldots0q_j)$$

and

$$R_j = (1 + .x_1x_2\ldots x_n) - D_j$$

with $D_0 = 0$ and $R_0 = .x_1x_2\ldots x_n$.

Note that the choice of each q_j is to try to make D_j to increase monotonically from 1 to $1 + .x_1x_2\ldots x_n$ which also corresponds to the partial remainder R_j decreasing from $.x_1x_2\ldots x_n$ to 0. So at each step j, we

can first try $q_j = 1$. If D_j turns out to be too big ($D_j > 1 + .x_1x_2 \ldots x_n$ or $R_j < 0$), then we just reset $q_j = 0$ and try to choose a smaller factor $(1 + q_{j+1}2^{-(j+1)})$ and so on.

Since

$$\begin{aligned} R_j &= (1 + .x_1x_2 \ldots x_n) - D_j \\ &= (1 + .x_1x_2 \ldots x_n) - (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_j) \\ &= (1 + .x_1x_2 \ldots x_n) - (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_{j-1}) \\ &\quad - (1 + .q_1)(1 + .0q_2) \ldots (1 + .0 \ldots 0q_{j-1}) \cdot q_j 2^{-j} \\ &= R_{j-1} - D_{j-1} \cdot q_j 2^{-j} \end{aligned} \tag{24}$$

we have the following algorithm.

Algorithm 2.1

Step 0: Let $R_0 = .x_1x_2 \ldots x_n$, $D_0 = 1$.

Step j: j=1,2,...,n

Let $R_j = R_{j-1} - D_{j-1} \cdot 2^{-j}$

If $R_j \geq 0$, $q_j = 1$, $D_j = D_{j-1} + D_{j-1}2^{-j}$

If $R_j < 0$, $q_j = 0$, $R_j = R_{j-1}$, $D_j = D_{j-1}$

Example 2.5

Table 4 shows the detailed steps for

$$R_0 = x = .111101$$

implemented by algorithm 2.1.

j	R_j	q_j	D_j
1	.011101	1	1.1
2	.000101	1	1.111
3	.000101	0	1.111
4	.000101	0	1.111
5	.000010	1	1.111011
6	.000001	1	1.111100

Table 4: Example 2.5

Note that because of truncation D_6 is different from x in the above example, the error estimation is done in the previous section.

The restoring division nature of the procedure can be improved by a non-restoring procedure with the use of signed digits. In fact, it will be shown that the SRT division technique can be used to achieve the computation complexity of $O(N)$ with still small chip area. In the rest of this section the development is described.

3.2.3 Modified Restoring Approach

In this subsection we develop another version of algorithm 2.1 which is more suitable for hardware design and also serves as a basis for the development of SRT division techniques in the next subsection.

First note that in Algorithm 2.1 $R_j < 2^{j+1}$ for each $j = 0, 1, \ldots$. This implies that for large value of j the first $j - 1$ fractional bits in R_j are all 0. Secondly, R_j is only involved in the computation of

$$R_j - D_j \cdot 2^{-(j+1)}$$

where $D_j \cdot 2^{-(j+1)}$ has the first j fractional bits as 0. So it can be seen that the first j-1 fractional bits of 0's in

$$R_j - D_j \cdot 2^{-(j+1)}$$

doesn't really contribute to the result. Plus considering the hardware implementation structure which often uses shift registers in this situation, we do the following modification of algorithm 2.1.

Let $F_j = 2^j R_j$, then from Eq. 24,

$$\begin{aligned} F_j &= 2^j R_j \\ &= 2^j (R_{j-1} - D_{j-1} \cdot q_j 2^{-j}) \\ &= 2^j R_{j-1} - D_{j-1} \cdot q_j \\ &= 2F_{j-1} - D_{j-1} \cdot q_j \end{aligned} \tag{25}$$

Thus we have the following algorithm which is more suited to hardware implementation.

Algorithm 2.2

Step 0: $F_0 = .x_1x_2 \ldots x_n$ and $D_0 = 1$

Step 1: For each $j = 1, 2, \ldots, n$,

1.1 $F_j = 2F_{j-1} - D_{j-1}$

1.2 If $F_j \geq 0$, then $\{q_j = 1;\ D_j = D_{j-1} + 2^{-j}D_{j-1}\}$

1.3 else $\{q_j = 0;\ F_j = 2F_{j-1};\ D_j = D_{j-1}\}$

Example 2.6

Consider a fractional number $x = .111101$, *the stepwise procedure of the algorithm 2.2 is tabulated in Table 5.*

Finally, (1.1)(1.01)(1.000)(1.0000)(1.00001)=1.11101111.

j	$2F_{j-1}$	D_{j-1}	q_j	F_j	D_j
1	1.111010	1.000000	1	0.111010	1.100000
2	1.110100	1.100000	1	0.010100	1.111000
3	0.101000	1.111000	0	0.101000	1.111000
4	1.010000	1.111000	0	1.010000	1.111000
5	10.100000	1.111000	1	0.110000	1.111011
6	1.100000	1.111011	0	1.100000	1.111011

Table 5: Example 2.5

3.2.4 Non-restoring Approach

For an n-bit factorization process, algorithm 2.2 requires n steps, where each step needs two n-bit addition/subtraction operations. In the whole process the most time consuming steps are the full length subtraction $F_j = 2F_{j-1} - D_{j-1}$ and the full length addition $D_j = D_{j-1} + D_{j-1}2^{-j}$ because of carry propagation. Note that at each step j in algorithm 2.2 we are essentially doing the following division problem

$$\frac{2F_{j-1}}{D_{i-1}} = q_j + \dots \tag{26}$$

in which the dividend is $2F_{j-1}$ and the divisor is D_{i-1}. The SRT division technique was first invented to avoid full length subtraction and make the quotient bit selection rule be data independent [2]. In other words, we can make a simpler selection rule which allows the dividend and the divisor to be in a certain range instead of being exact and still guarantees the convergence. The use of this technique solves the carry propagation problems just mentioned. The price paid for this technique is that sign digit has to be used, which means that in the radix–2 case, the quotient bit q_j has to be allowed to assume value from the set {-1,0,1}. This result in the look-up table to be doubled to also include the values $\log(1 - 2^{-j})$.

This increase is not significant since the original look-up table has a very small size. On the other hand, the speed performance is improved dramatically, especially in long word length situation.

In the rest of this subsection the detail of the application of SRT division technique is developed.

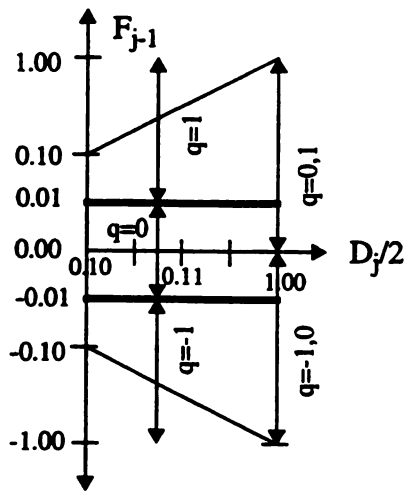

Figure 4: Radix-2 P-D plot

Let the quotient digit set be {-1, 0, 1}. For simplicity of notation, let "$\bar{1}$" denote "-1". Consider the P-D plot in Fig. 4, to guarantee $|P - qD| < D$,

the ranges for q is

$$(-1+q)D \leq P \leq (1+q)D, \quad \text{where } 0.5 \leq D < 1$$

i.e., $0 \leq P \leq 2D$ for $q = 1$; $-D \leq P \leq D$ for $q = 0$; and $-2D \leq P \leq 0$ for $q = -1$. As a result, the ranges of the overlapping area $P_{0,1}$ for both $q = 0$ and $q = 1$ is: $0 \leq P_{0,1} \leq D$, and the ranges of the overlapping area $P_{-1,0}$ for both $q = -1$ and $q = 0$ is: $-D \leq P_{0,1} \leq 0$. Therefore, two comparison constants, $\frac{1}{4}$ and $-\frac{1}{4}$, can be chosen to determine q for divisor D between 0 and 1, and between 0 and -1 respectively.

In the implementation of Eq. 26, the divisor at each step j is $2F_{j-1}$ and the dividend D_{j-1} is always in the range [1,2).

Thus, we consider $D = \frac{D_{j-1}}{2}$ so that $1 > D \geq 0.5$. The division

$$\frac{2F_{j-1}}{D_{i-1}} = \frac{F_{j-1}}{\frac{D_{i-1}}{2}}$$
$$= \frac{F_{j-1}}{D}$$

defines $P = F_{j-1}$.

This implies that D is ranged between 0.5 and 1.0 and P is between -1.0 and 1.0. As shown in Fig. 4, two comparison constants $c_1 = 0.25_{10} = 0.01_2$

and $c_{-1} = -0.25_{10} = 1.11_2$ (2's complement representation) can be used.

Thus, the quotient digit q_j is estimated by

$$q_j = \begin{cases} 1 & \text{if } F_{j-1} > c_1 \\ 0 & \text{if } c_{-1} \leq F_{j-1} \leq c_1 \\ -1 & \text{if } F_{j-1} < c_{-1} \end{cases} \tag{27}$$

The improved continuous factorization scheme is summarized in the following.

Algorithm 2.3

Step 0: $F_0 = .x_1 x_2 \ldots x_n$ and $D_0 = 1$

Step 1: For each $j = 1, 2, \ldots, n$,

1.1 Estimate q_j from Eq. 27

1.2 $F_j = 2F_{j-1} - q_j D_{j-1}$

1.3 $D_j = D_{j-1} + q_j \cdot 2^{-j} D_{j-1}$

Example 2.7

j	$2F_{j-1}$	D_{j-1}	q_j	F_j	D_j
1	1.111010	1.000000	1	0.111010	1.100000
2	1.110100	1.100000	1	0.010100	1.111000
3	0.101000	1.111000	0	0.101000	1.111000
4	1.010000	1.111000	1	$0.\bar{1}0\bar{1}000$	1.111111
5	$\bar{1}.0\bar{1}0000$	1.111111	$\bar{1}$	0.101111	1.111100
6	1.011110	1.111100	1	$0.\bar{1}00010$	1.111101

Table 6: Example 2.6

Let's reconsider the previous example in this algorithm as shown in Table 6. In this example $D_6 = x$. In general there could be some error whose range is analyzed in the section 3.2.1.

Note that steps 1.2 and 1.3 in algorithm 2.3 can be completed very fast by the use of carry-save-adder (CSA), which eliminates carry propagation and produces an approximate value, whose accuracy determines the number of bits kept for a short length full addition.

3.2.5 Modified Error Analysis and Hardware Implementation Structure

Because of the simplication involved in algorithm 2.3, the error analysis in subsection 3.2.1 needs to be modified. It is done in this subsection and the schematic structure to implement the developed algorithm 2.3 is also proposed.

Suppose the input word length is n bits and let g be the number of guarding digits needed to ensure conversion accuracy of also n bits. This means the internal register length is $n + g$. Let $ulp = 2^{-(n+g)}$. Note that in algorithm 2.3 the only error is caused by the truncation in calculating D_j at each step j, which result in an error no larger than ulp. It then follows easily that we get exactly the same result as given by Corollary 3 in subsection 3.2.1, under the additional condition that $\log(1 + 2^{-j})$ is stored in truncated form and $\log(1 - 2^{-j})$ in round-up form.

In the following the implementation structure of algorithm 2.3 is developed. Suppose the input word length is n, then the look-up table used to store $\log(1 \pm 2^{-j})$ will have the size of $2n^2$. As an example, if $n = 64$, then

$2n^2 = 8K(\text{bits})$. It's interesting to note that by the look-up table reduction technique developed in subsection 3.2.1, and the fact that $\log(1-x) \approx -\log(1+x)$, the look-up table size can be just n^2, but this involves a little hardware overhead. Three CSA's are needed. one for the addition of

$$\sum_{j=1}^{n} \log(1 \pm q_j 2^{-j})$$

and two others for the factorization process. One is used for the dividend F_j, the other for the divisor D_j.

Note that $q_1 = x_1$ is directly determined. For the comparison of partial remainder F_j with comparison constant c, note from the P-D plot that the allowable error is $\frac{1}{4}$. To ensure convergence in the algorithm, we need to retain 4 fractional bits of the CSA for D_j, plus the first integral digit and sign bit, we need to do a full addition of 6 bits for the comparison step 2.1 in algorithm 2.3, which can be implemented by a carry-look-ahead adder to improve speed or a usual full adder to save chip area if the carry propagation delay is bearable in the application.

s	a	f_1	$f_2 - f_4$	q	q_s	q_a
0	0	0	* * *	0	0	0
0	0	1	* * *	1	0	1
0	1	*	* * *	1	0	1
1	0	*	* * *	$\overline{1}$	1	1
1	1	0	* * *	$\overline{1}$	1	1
1	1	1	* * *	0	0	0

Table 7: Truth table for q

After the result $sa.f_1f_2f_3f_4$ of full addition is produced the quotient bit q can be determined. Table 7 shows the encoding of q and its logic function depending on $sa.f_1f_2f_3$.

The logic functions of q_s and q_a are then derived as

$$q_s = s(\overline{a} + \overline{f}_1)$$

$$q_a = a \oplus s + a \oplus f_1$$

The implementation structure is shown next.

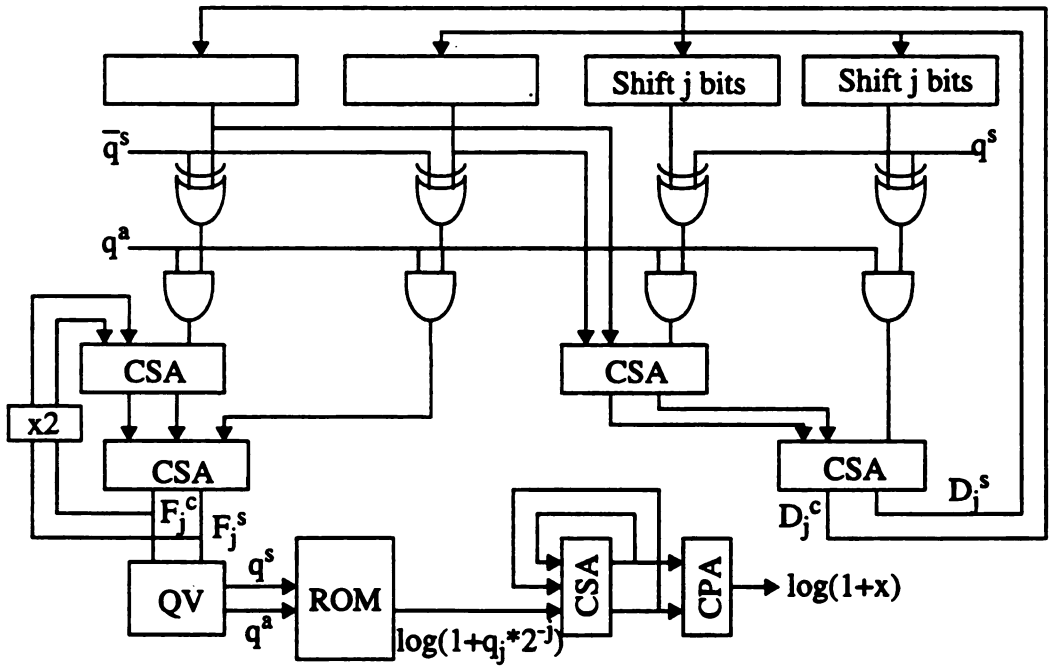

Figure 5: Block diagram of the implementation of algorithm 2.3

4 Conclusion

Logarithmic number system (LNS) is an attractive alternative to the conventional number system when the data need to be manipulated at very high rate over a wide data range. The major problems in dealing with the logarithmic number system are to derive logarithm and anti-logarithm quickly and accurately enough to allow conversion to and from the conventional number representation.

This thesis study develops two efficient algorithms for binary logarithmic conversion : TLFA (Two Layer Factorization Algorithm) and CFA (Continuous Factorization Algorithm). The former uses a factorization approach to reduce the look-up table size and employs a nonlinear approximation method to reduce the computational complexity. Simulation results on IEEE single precision conversion (23 bits) show that the algorithm requires only one ROM table of $2^{13} \times 26$ bits, one of $2^{13} \times 14$ bits, and one of $2^{13} \times 5$ bits, or a total of 360K bits. This algorithm has advantages over other work on this problem in both speed and implementation structure which is mainly due to the discovery of a convenient nonlinear approximation function. For handling long word

length numbers, such as in IEEE double precision (53 bits), CFA uses a continuous factorization process to further reduce the look-up table size and reduces the problem to a division one which can be implemented using the SRT division technique [2] to give a high-speed yet simple structure. More specifically, the conversion time is about three additions irrespective of word length.

The speed of SRT-based divisions is mainly determined by the complexity of the quotient-digit selection [2, 15, 16]. To speed up the division process, one may reduce the number of iteration steps by increasing the radix β of the sign digit number system used in the process. Selecting $\beta = 2^m$ allows the generation of m quotient bits at each step and the number of steps can be reduced to $\lceil \frac{n}{m} \rceil$ where n is the word length. However, the complexity of the quotient bit selection and remainder updating increase for high radices, offsetting the advantages of the reduction in the number of iterations [2]. Therefore, exploring the speed performance in this logarithmic conversion problem using the high-radix SRT division scheme is an interesting topic for future work.

Appendices

Appendix

A Source code of simulation program for TLFA 1.1 (in Java)

```
// This program simulates algorithm 1.1 for all
// possible 16-bit input combinations.


class misc {
      static int N=8;
      static long one = (long) (Math.pow(2,N)+.5);
      static double e=Math.pow(2,-N);


    public static void toBinary(long x, int n)  {
      int i;
      int [] a=new int [128];
      for (i=n-1; i>=0; i--)  {
```

```
    long t=x&1;
    x /= 2;
    a[i]= (int) t;
  }
  for(i=0; i<n; i++)
    System.out.print(a[i]);
  System.out.print("   ");
}

public static long expconv(long x)  {
  double yd=Math.pow(2,x*e)-1;
  return (long)(yd/e+.5);
}

public static long lt1(long x, int tag)  {
  double xd = (double) x * e;
  double yd = Math.log(1+xd)/Math.log(2);
  long  y = (long) (yd/e/e+.5);
  switch(tag)  {
```

```
    case 0: return y;
    case 1: return (y*2/one+1)/2;
    case 2: return y%one;
  };
  return -1;
}

public static long lt2(long x, int tag)  {
  double xd = (double)x*e*e;
  double yd = Math.log(1+xd)/Math.log(2);
  long  y = (long) (yd/e/e+.5);
  switch(tag)  {
    case 0: return y;
    case 1: return y/one;
    case 2: return y%one;
  };
  return -1;
}
```

```
    public static long lt(long x)  {
      double xd = (double)x*e*e;
      double yd = Math.log(1+xd)/Math.log(2);
      long  y = (long) (yd/e/e+.5);
      return y;
    }

};

class s  {
  public static void main(String[] args) {
    int N=8;
    int[] diff=new int[N];
    long c,C,x,y,A,Aprime,B,p,pf,P,r,t,t1,cnt=0,err=0;
    long one = (long) (Math.pow(2,N)+.5);
    long two = one*2;
    double xd,yd;
    double e=1/(double) one;
```

```
    for (x=0; x<one; x++)  {
      A = misc.lt1(x,1);
      Aprime = misc.lt1(A,1);
      for (y=0; y<one; y++)  {
        B = misc.lt1(y,1);
        p = one-B+A;
        pf=p%one;
P = misc.lt1(pf,1);
if (p<one) c = (one-P)+Aprime/2;
        else c = -P/2 + Aprime/2;
C = misc.lt2(c,0);
        r = misc.lt1(x,0)+C;
        t = misc.lt(x*one+y);
        t1 = r-t;
        diff[(int)Math.abs(t1)]++;
      }
    }
```

```
    for (int i=0; i<N; i++)
     System.out.println(i + "  " + diff[i]);
  }
}
```

B Source code of simulation program for TLFA 1.2 (in Java)

```
// This program simulates algorithm 1.2 in IEEE single precision.


class misc {

      static int N=13;

      static long one = (long) (Math.pow(2,N)+.5);

      static double e=Math.pow(2,-N);


    public static void toBinary(long x, int n)  {

      int i;

      int [] a=new int [128];

      for (i=n-1; i>=0; i--)  {

        long t=x&1;

        x /= 2;

        a[i]= (int) t;

      }

      for(i=0; i<n; i++)
```

```
    System.out.print(a[i]);
  System.out.print("   ");
}

public static long expconv(long x)  {
  double yd=Math.pow(2,x*e)-1;
  return (long)(yd/e+.5);
}

public static long lt1(long x, int tag)  {
  double xd = (double)x*e;
  double yd = Math.log(1+xd)/Math.log(2);
  long  y = (long) (yd/e/e+.5);
  switch(tag)  {
    case 0: return y;
    case 1: return y/one;
    case 2: return y%one;
  };
  return -1;
```

```
}

public static long lt2(long x, int tag)  {
  double xd = (double)x*e*e;
  double yd = Math.log(1+xd)/Math.log(2);
  long  y = (long) (yd/e/e+.5);
  switch(tag)  {
    case 0: return y;
    case 1: return y/one;
    case 2: return y%one;
  };
  return -1;
}

public static long lt(long x)  {
  double xd = (double)x*e*e;
  double yd = Math.log(1+xd)/Math.log(2);
  long  y = (long) (yd/e/e+.5);
  return y;
```

```
    }


};


class s  {
  public static void main(String[] args) {
    int N=13;
    long c,x,y,x1,y1,t,t1,t2,err=0;
    long one = (long) (Math.pow(2,N)+.5);
    long two = one*2;
    double xd,yd;
    double e=Math.pow(2,-N);



    for (x=0; x<one; x++)  {
      x1 = misc.lt1(x,1);
      t2 = misc.expconv(one-x1)/2;
      for (y=0; y<one; y++)  {
//        misc.toBinary(x,N);
```

```
//        misc.toBinary(y,N);
          y1 = misc.lt1(y,1);
          t1 = y1-x1;
          if(t1>=0) c=misc.expconv(t1)+one/2-t2;
          else  c=misc.expconv(one+t1)/2-t2;
          y1 = misc.lt1(x,0)+misc.lt2(c,0);
          t = misc.lt(x*one+y);
//        misc.toBinary(y1,N); misc.toBinary(t,N);
          t1 = y1-t;
          if (Math.abs(t1)>err) err = Math.abs(t1);
//        System.out.println(t1 + "   " + err);
        }
      }
      System.out.println("Maxerror=" + err);
    }
}
```

References

[1] H.Y. Lo and Y. Aoki, "Generation of a precise binary logarithm with difference grouping programmable logic array," IEEE Trans. on Computers, C-34(8) pp.681-691, 1985

[2] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

[3] N.G. Kingsbury, "Digital filtering using logarithmic arithmetic", Electron. Lett., Vol 7, pp. 56-58, 1971

[4] W.F. Wong and E. Goto, "Fast evaluation of the elementary functions in single precision", IEEE Transactions on Computers, Vol 44, No. 3, pp. 453-457, 1995

[5] S.-C. Huang and L.G. Chen, "The chip design of A 32-b Logarithmic Number System," Proc. IEEE International Symp. on Circuits and Systems, pp. 167-170, 1994.

[6] M.G. Arnold, T.A. Bailey, J.R. Cowles and M.D. Winkel, "Applying features of IEEE 754 to sign/logarithm arithmatic", IEEE Transactions on Computers, Vol 41, No. 8, pp. 1040- 1049, 1992

[7] F.-S. Lai and C.-F. E. Wu, "A hybrid number system processor with geometric and complex arithmetic capabilities", IEEE Transactions on Computers, Vol 40, No. 8, pp. 952-962, 1991

[8] D.K. Lostopoulos, "An algorithm for the computation of binary logarithms", IEEE Transactions on Computers, Vol 40, No. 11, pp. 1267-1270, 1991

[9] I. Koren and O.Zinaty, "Evaluating elementary functions in a numerical coprocessor based on rational approximations", IEEE Transactions on Computers, Vol. 39, No. 8, pp. 1030- 1037, 1990

[10] D.M. Lewis, "An architecture for addition and subtraction of long word length numbers in the logarithmic number system", IEEE Transactions on Computers, Vol 39, No. 11, pp. 1325-1336, 1990

[11] T. Stouraitis and F.J. Taylor, "Floating-point to logarithmic encoder error analysis", IEEE Transactions on Computer, Vol 37, NO. 7, pp. 858-863, 1989.

[12] T. Kurokawa, J. Payne and S. Lee, "Error analysis of recursive digital filters implemented with logarithmic number systems", IEEE Trans. Accost., Speech, Signal processing, Vol ASSP-28, pp.706-715, 1980

[13] F.J. Taylor, "A hybrid floating-point logarithmic number system processor", IEEE Trans. Circuit System, Vol CAS-32, pp. 92-95, 1985

[14] F.J. Taylor, R. Gill, J. Joseph and J. Radke, "A 20-bit logarithmic number system processor", IEEE Transactions on Computer, Vol 37, pp. 190-199, Feb. 1988

[15] T.-H. Pan, H.-S. Kay, Y. Chun and C.L. Wey, "High-Radix SRT division with speculation of quotient digits," Proc. IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD'95), Austin, TX, pp. 479-482, Oct. 1995

[16] C.L. Wey and C.-P. Wang, "VLSI implementation of a fast radix-4 SRT division," Proc. of 39th Midwest Symp. on Circuits and Systems, Iowa, 65-68, August 1996