[9] C.-T. Ho, " Full bandwidth communications on folded hypercubes," in *Proc. Int. Conf. Parallel Processing*, vol. I. Penn State, 1989, pp. 276–280.

[10] ——, "An observation on the bisectional interconnection networks," *IEEE Trans. Comput.*, vol. 41, no. 7, pp. 873–877, July 1992.

[11] S. Latifi and A. El-Amawy, "on folded hypercubes," in *Proc. Int. Conf. Parallel Processing*, vol. I. Penn State, 1989, pp. 180–187.

[12] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam: North-Holland, 1977.

[13] M. E. Watkins, "Connectivity of transitive graphs," *J. Combinat. Theory*, vol. 8, 23–29, 1970.

Implementation of Four Common Functions on an LNS Co-Processor

Debasish Das, Krishnendu Mukhopadhyaya, and Bhabani P. Sinha

Abstract—We propose a scheme for evaluating four commonly used functions namely, 1) inverse trigonometric functions, 2) trigonometric functions, 3) the exponential function, and 4) the logarithmic function with the help of a logarithmic number system (LNS) processor. A novel idea of *series folding* has been introduced for computing the above functions, expressed in the form of infinite series. We also show that with a suitable choice of the radix for the LNS we can evaluate exponential and logarithmic functions without using any extra hardware.

Index Terms— Logarithmic number system, series folding, inverse trigonometric functions, arctangent function, trigonometric functions, exponential function, logarithmic function.

I. INTRODUCTION

Many real time applications in the areas of signal processing, process control, etc., require very fast evaluation of a large number of mathematical functions. Contemporary Arithmetic Logic Units (ALU) of a general purpose computer may often find it difficult to meet this requirement of massive real-time computations. One of the main reasons for not achieving such high rate of arithmetic computations is the relative inefficiency associated with floating-point operations.

To overcome this difficulty, the idea of the Logarithmic Number System (LNS) for the representation and manipulation of numbers was proposed [1]. LNS offers several advantages compared to the floating-point representation. A fast arithmetic unit was developed [2] for obtaining very high computational data rates. Such processors are very efficient in performing operations like multiplication, division, squaring and square-rooting, but slow for operations like addition and subtraction. Addition and subtraction operations in LNS need to perform a table look-up and this table is to be stored as part of the LNS processor. A technique for reducing the size of the look-up tables has been proposed in [3].

One of the problems associated with the design of the LNS processor was the conversion of a number from the binary floating-point system to LNS and vice-versa. In the design given in [2], the authors proposed the use of the look-up tables to perform these operations. However, methods for generating the logarithm of a number using PLAs have evolved, as given in [4] and have solved this

problem too. To reduce the number of conversions from floating-point to LNS, it is reasonable to think about an LNS co-processor which can perform all the arithmetic operations performed by a commercially available numeric co-processor and with the same precision. The LNS co-processor as designed in [2] was of very limited application. The processor could perform only six basic arithmetic operations 1) addition, 2) subtraction, 3) multiplication, 4) division, 5) squaring, and 6) square-rooting. It is difficult for a co-processor of such limited capability to satisfy the demanding needs of computing other mathematical functions involved in many real-time applications.

In this brief contribution, we propose the idea of implementing four other general purpose functions in such a co-processor, by using suitable algorithms and a little amount of extra hardware. The four functions that we have chosen for implementation are very fundamental and frequently needed. They are 1) inverse trigonometric functions, 2) trigonometric functions, 3) the exponential function, and 4) the logarithmic function. Each of these functions is first expressed as a power series and then evaluated by the LNS co-processor. In evaluating such infinite power series of a variable x, the primary problem is that the number of terms which are to be evaluated and then summed up, depends on the precision and the value of the argument x. If x is a small fraction close to zero, the number of terms to be summed up will be small; otherwise it will be large. The proposed technique is centered around developing an algorithm based on an idea of *series folding*, such that by suitably transforming the argument, we can compute the value of the function, by evaluating a small number of terms for a given precision. Since the number of multiplications/divisions to be performed in a power series evaluation soon overrides the number of additions/subtractions, and also because the multiplication/division dominated computations can be executed at a faster rate on an LNS co-processor, the proposed technique helps to compute these functions very quickly with a reasonable accuracy.

In the final part of our work, we will show that a suitable choice of the radix r of the LNS, may eliminate the requirement of any extra computation other than table look-up for the exponential and logarithmic functions, again without affecting the precision.

II. LOGARITHMIC NUMBER SYSTEM AND ASSOCIATED ARITHMETIC

In LNS, a number x is represented in signed magnitude form, i.e., as a pair (S, e), where $x = (-1)^S (r)^e$, S being the sign bit (which is either 0 or 1 according to the sign of x) and e being the signed exponent of the radix r. The exponent e is expressed in fixed point binary mode with say, I bits for the integer part and F bits for the fractional part and one bit for the sign of the exponent, i.e., with a total of $(I + F + 1)$ bits. If the radix is considered to be 2, then the smallest number that can be represented using the scheme is 2^{-N}, where $N = (2^I - 1) + (1 - 2^{-F}) = (2^I - 2^{-F})$. The ratio between two consecutive numbers is equal to $r^{2^{-F}}$, and the corresponding precision ϵ is roughly $(\ln r) 2^{-F}$. Typically, if $I = 5$, $F = 26$, and $r = 2$, we can have a precision of 26 bits in radix 2. However, for the purpose of comparison with the precision of floating-point representation, ϵ will be assumed as $2^{-23} (\approx 10^{-7})$.

Arithmetic operations involving manipulation of the exponent part only can very easily be performed using such a representation. Assume that two numbers A and B are represented in the LNS format as the tuples $(S(A), e(A))$ and $(S(B), e(B))$ respectively where $A = (-1)^{S(A)} r^{e(A)}$ and $B = (-1)^{S(B)} r^{e(B)}$. Now, if $C = A \times B$, and C is represented in the LNS form as the pair $(S(C), e(C))$, then $e(C) = e(A) + e(B)$ and $S(C) = S(A) \odot S(B)$. For $C = \frac{A}{B}$,

everything will be same except that $c(C) = c(A) - c(B)$. So it is clear that these operations can be performed in a single addition or subtraction time of a fixed-point number. If $C = A + B$ and $S(A) = S(B)$, then for $A \leq B$, $S(C) = S(B)$ and $c(C) = c(A) + \Phi_a(V)$, where $V = c(B) - c(A)$, and $\Phi_a(V) = \log_r(1 + r^V)$. For $C = A - B$ with $A \leq B$, and $S(A) = S(B)$ everything remains the same except that Φ_a is changed to $\Phi_s = \log_r(1 - r^V)$. Evaluating Φ_a or Φ_s involves a table look-up. Hence, the addition and subtraction operations are slower in the LNS than those in the floating-point number system. Also, subtraction may introduce a large amount of error when the two operands are nearly equal.

III. INFINITE SERIES EVALUATION USING AN LNS CO-PROCESSOR

To minimize the number of conversions from floating-point to LNS and back, the LNS processor is normally used as a dedicated co-processor for the evaluation of mathematical functions where the intermediate results can remain in the LNS format. Data conversion would be performed only on the operands for bringing them into the co-processor from the main processor and then also on the results taken out of the LNS co-processor to the main processor. In such an environment, the number of data inputs and outputs should also be preferably restricted to a small value. The class of computational problems which are dominated by operations such as multiplication, squaring, square-rooting would yield significantly faster results using the LNS without introducing a significant amount of error.

All these features of an LNS co-processor make it suitable for evaluating functions which can be expressed as infinite series. For practical purposes, we can truncate the series after some fixed number of terms when we achieve the desired level of precision.

IV. EVALUATION OF INVERSE TRIGONOMETRIC FUNCTIONS

We consider here the evaluation of the arctan function, i.e., $\tan^{-1}(x)$ for an argument x. Other inverse trigonometric functions can very easily be evaluated from $\tan^{-1}(x)$. For example, suppose $\sin^{-1}(x)$ is to be evaluated. We first evaluate $y = \frac{x}{\sqrt{(1 - x^2)}}$. Then $\tan^{-1}(y)$ gives the value of $\sin^{-1}(x)$. The Taylor series expansion of $\tan^{-1}(x)$ for $|x| \leq 1$ is given by,

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \qquad (1)$$

We can use this series in evaluating $\tan^{-1}(x)$ for all x, $0 \leq x \leq 1$. However, for $-1 \leq x \leq 0$ we will evaluate $\tan^{-1}(-x)$ since $\tan^{-1}(x) = -\tan^{-1}(-x)$. Similarly if $|x| > 1$, we compute $y = \frac{1}{x}$ and then compute $\tan^{-1}(y)$. Since $\tan^{-1}(y) = \cot^{-1}(x) = \frac{\pi}{2} - \tan^{-1}(x)$, we get,

$$\tan^{-1}(x) = \frac{\pi}{2} - \tan^{-1}(y) \qquad (2)$$

Thus, we can evaluate $\tan^{-1}(x)$ for all x by computing different terms in the appropiate series expansion and summing them up until a desired precision is achieved. The LNS co-processor can very efficiently be used for computing each individual term. However, there are two major problems in the approach. First, it needs to be determined how many terms are to be evaluated to achieve the desired precision. For x close to 1, the number of terms may be very large and the computation will then be highly inefficient. Secondly, there are alternate positive and negative terms, causing slower convergence of the series. Subtractions may also introduce a very large amount of error in the computation when the two operands are nearly equal. For the first problem, we note that we can terminate whenever any term becomes less than a predefined value. To overcome the second

problem, we see that after a little manipulation $\frac{\tan^{-1}(x)}{(1 - x^2)}$ can be expressed as

$$\frac{\tan^{-1} x}{(1 - x^2)} = x + (1 - \frac{1}{3})x^3 + (1 - \frac{1}{3} + \frac{1}{5})x^5 + \cdots \qquad (3)$$

Thus, we have obtained a series expansion of $\frac{\tan^{-1}(x)}{(1 - x^2)}$ as $\sum_{p \geq 1} A_{p-1} x^{2p-1}$, where $A_{p-1} = (1 - \frac{1}{3} + \frac{1}{5} + \cdots + \frac{(-1)^{p-1}}{(2p-1)})$. Note that A_{p-1} is always positive and converges to the value $\frac{\pi}{4}$ as p tends to infinity. Thus the second problem is solved. Instead of evaluating the A_{p-1}'s each time, we can store them in a table for a direct look-up. However, to implement it in practice, we must have an idea of the size of the table, i.e., we are to know the upper bound p_{max} on the number of terms that are to be evaluated.

A. The Upper Bound on p

To calculate the upper bound on the number of terms to be evaluated, we proceed as follows.

Lemma 1: Let $T_p(x) = A_{p-1} x^{2p-1}$, and let $t_p(x) = (1 - x^2)T_p(x)$. Then for all $p \geq 1$, $t_{p+1}(x) < t_p(x)$.

Proof: Omitted.

By lemma 1 the successive terms in eqn.(3) are monotonically decreasing. Our goal is to find a value of p such that the LNS representation of t_p is less than a predefined value η. We may choose η to be equal to -23 in radix 2, which would turn out to be $2^{-23}(\eta = \log_2 \epsilon)$ after conversion to its floating-point representation. Using lemma 1, we can claim that if the LNS representation of $t_p(x)$ is less than -23, then the LNS representation of $t_{p-1}(x)$ will also be less than -23. In this case, we will evaluate only $(p - 1)$ terms of the series. Hence, the maximum number of terms p_{max}, that needs to be evaluated is one less than the smallest value of p satisfying the inequality

$$\log_2(t_p(x)) < -23 \qquad (4)$$

i.e.,

$$\log_2(1 - \frac{1}{3} + \frac{1}{5} + \cdots - \frac{1}{(2p - 1)})$$
$$+ (2p - 1)\log_2 x + \log_2(1 - x^2) < -23 \qquad (5)$$

Examining the inequality (5), one can easily conclude that the value of p_{max} is primarily dependent on the value of $(-23)/\log_2(x)$. For $x = 1$, we see that an infinite number of terms are to be evaluated for evaluating the series correctly. As x decreases, p_{max} will also decrease. For a successful implementation of the series evaluation scheme, we must have a sufficiently small value of p_{max}. However, even for $x = 0.5$, p_{max} can be as large as 12. The only way to further reduce the value of p_{max} is to restrict the maximum value of x to a still smaller value. In the next section, we will discuss a method which would enable us to achieve this goal when evaluating the series, yet finding the value of $\tan^{-1}(x)$ for any value of the argument x, $0 \leq x \leq 1$.

Lemma 2: For any p satisfying the inequality (5), $\sum_{i \geq 1} t_{p+i}(x) < 2\eta$, when $x \leq 0.5$.

Proof: Omitted.

Lemma 3: The total series truncation error in evaluating $\tan^{-1}(x)$ will be less than η, if we take $p_{max}(= p_0 + 1)$ terms of the series (3), where p_0 is the smallest value of p satisfying the inequality (5).

Proof: Omitted.

We now introduce the idea of *series folding* in the following section to reduce the value of p_{max} by a suitable transformation on the argument x without losing the precision. The reduction in x will, in turn, reduce the residual error.
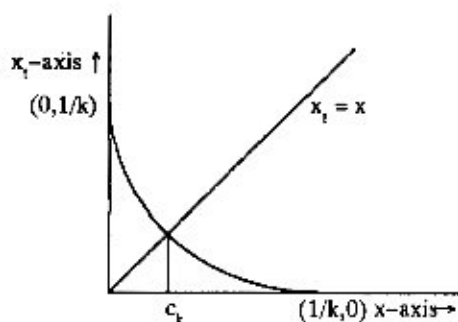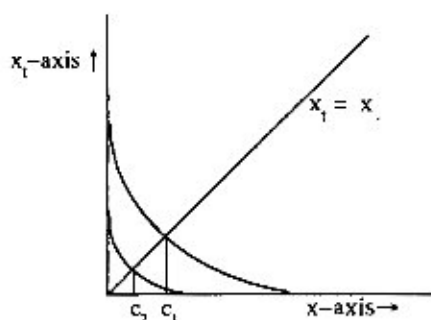
Fig. 1. Curves showing the value of c_k.

Fig. 2. Curves showing c_1 and c_2.

B. Series Folding

Let us have a transformation on x defined by $x_t = \frac{(1-kx)}{(k-x)}$, where k is a positive integer. The expressions $\tan^{-1}(x_t)$ and $\tan^{-1}(x)$ are then related as,

$$\tan^{-1}(x_t) = \tan^{-1}\left(\frac{1}{k}\right) - \tan^{-1}(x). \tag{6}$$

Now, we shall show that for certain values of x, the transformed value x_t is less than x. The graph of x_t versus x assumes the shape as shown in Fig. 1. We have also drawn the straight line $x_t = x$ on Fig. 1, which intersects the curve $f(x) = \frac{(1-kx)}{(k-x)}$, at a point $x = c_k$ where, $c_k = \frac{(1-kc_k)}{(k+c_k)}$, i.e.,

$$c_k = \left(\sqrt{(k^2+1)} - k\right) \tag{7}$$

We call c_k the critical value corresponding to k. We note that for $x > c_k$, $x_t < x$ and for $x < c_k$, $x_t > x$. If we go on changing the values of k, a number of such critical values c_1, c_2, c_3 etc., would be obtained for $k = 1, 2, 3, \cdots$, respectively. Specifically, $c_1 = \sqrt{2}-1$, $c_2 = \sqrt{5} - 2$, etc. This is illustrated in Fig. 2. We define $c_0 = 1$.

Now, we can start folding the series as follows. If $c_1 < x \leq c_0$ we transform x to x_1, where $x_1 = \frac{(1-x)}{(1+x)}$ (for $k = 1$). This transformed value x_1 will be less than c_1. If $c_2 < x \leq c_1$, then we can transform x_1 to x_2 as $x_2 = \frac{(1-2x_1)}{(2+x_1)}$ (for $k = 2$). In general, we divide the interval [0, 1] into several subintervals $I_1, I_2, \cdots$, where the ith subinterval I_i is defined by $I_i = (c_i, c_{i-1})$ for $i \geq 1$. For a given input argument x, we check which interval x lies in and transform it

accordingly. For example, if it falls in I_q we use the transformation with k = q and obtain the transformed argument $x_q = \frac{(1-qx)}{(q+x)}$, so that $x_q \leq c_q$. We can repeat this process to bring down the transformed value to less than or equal to any desired c_m, by at most m successive transformations.

Remark: As a special case, for $x > c_0$, the transformation on x is, $x_0 = \frac{1}{x}$ (for $k = 0$), which we have already seen in (2).

Remark: Fixing the maximum number of transformations to some value of m will also determine the value of p_{max}.

After evaluating the arctan of the transformed argument we can get back $\tan^{-1}(x)$ by using (6). For this retransformation we have to do only few subtractions. If we fix that at most m transformations will be used so that the value of the transformed argument will be below c_m, then we have to do a maximum of m subtractions. The values of $\tan^{-1}(\frac{1}{k})$ for $k = 0, 1, 2, \cdots$ are stored to facilitate this process (for $k = 0$, we set $\tan^{-1}(\frac{1}{k}) = \frac{\pi}{2}$).

The proposed algorithm including the transformations and retransformations can be precisely described for a given input argument x and a given value of the maximum number of transformations as found at the bottom of the page.

One may find a remote similarity between this idea of series folding and the transformation technique described in [5, pp. 123–125].

In terms of m, the maximum number of transformations, the inequality (5) can be rewritten as,

$$\log_2\left(1 - \frac{1}{3} + \frac{1}{5} - \cdots - \frac{(-1)^{p-1}}{(2p-1)}\right)$$
$$+ (2p - 1)\log_2\left(\sqrt{m^2+1} - m\right) - \log_2(1 - x^2) < -23. \tag{8}$$

Thus, as we increase the value of m, the value of p_0 obtained from the inequality (8) (hence the value of $p_{max} = p_0 + 1$) would decrease. It is clear from the inequality (8) that as $m \to \infty$, $p_0 \to 0$. Table I shows the dependency of p_{max} on m. If we increase m, the number of terms p_{max} required for computing $\tan^{-1}(x)$ is reduced, resulting in lesser amount of computational error. On the other hand, increasing the value of m has the effect of introducing error in the argument itself. Similarly, if p_{max} is reduced then the time for evaluating the terms of the series and summing them up will decrease while that for transformation and retransformation will increase. It is expected that there may be some optimum value of m for which the product of error and computational time will be minimum.

C. Choice of m

In this section, our objective is to find the optimum number of transformations. To do so, we first find the error introduced in the computed value of $\tan^{-1}(x)$ for different values of m, during the whole process of argument transformations, evaluations and summing of terms, and retransformations. To estimate the error, we use a simulation program. We next estimate the required computational time for different values of m. Finally, we choose the value of m for which the product of the total error and the computational time is minimum.

1) Simulation Results: The proposed technique was simulated with $\epsilon = 2^{-23}$ for uniformly distributed ten million values of x between 0 and 1. The results were compared with the values of $\tan^{-1}(x)$ calculated in floating-point with double precision. The

---

Step 1 : Set $k = 0$.
Step 2 : Compute the function $g(x, k, m)$, where

$$g(x, k, m) = \begin{cases} \sum_{p=1}^{p_{max}} t_p(x), & \text{for } k > m \\ g(x, k+1, m), & \text{for } (x \leq C_k) \text{ and } (k \leq m) \\ \tan^{-1}(\frac{1}{k}) - g\left((1-kx)/(k+x), k+1, m\right), & \text{for } (x > C_k) \text{ and } (k \leq m). \end{cases}$$

TABLE I

Maximum Number of Transformations (m)	Maximum Number of Terms(p_{max})
1	9
2	6
3	5
4	4
5 - 9	4
10 - 12	3
13	2

TABLE II

m	Average Error	Maximum Error
1	1.0 E-7	5.8E-7
2	5.4E-8	2.6E-7
3	4.8E-8	2.2E-7
4	1.0E-7	4.8E-7
5	5.1E-8	2.2E-7
6	4.3E-8	2.1E-7
7	4.1E-8	1.8E-7
8	4.0E-8	1.8E-7
9	3.0E-7	6.1E-7
10	1.8E-7	4.2E-7
11	1.2E-7	3.2E-7
12	8.5E-8	2.6E-7
13	6.3E-8	2.2E-7

TABLE III

m	T_c (in terms of t_c)
1	75
2	64
3	67
4	70
5	80
6	90
7	100
8	110
9	120
10	123
11	133
12	143
13	146

observed differences are listed in Table II. The proposed algorithm provides good accuracy for any reasonable value of m. Both the average and the maximum errors attain their minimum at $m = 8$.

2) Estimation of the Computational Time: As mentioned in Section II, multiplication and division operations can be done at a faster speed than addition and subtraction operations in LNS. The time required for one multiplication or division is no more than the time required for one fixed-point addition or subtraction. However, one addition or subtraction in the LNS requires two fixed-point additions or subtractions and one table look up [2]. For addition or subtraction with a very high precision, one may even need a Taylor series approximation in addition to the table look-up [3]. Thus, if we assume that the time required for a table look-up is same as that for one fixed-point addition, then the time required for one addition (subtraction) operation is approximately three times than that taken for one multiplication operation. If we denote the time needed for one fixed-point addition (subtraction) by t_f, then the time needed for an LNS multiplication or division is equal to t_f and that for one addition or subtraction is $3t_f$. However, as a special case, $1 + x$ or $1 - x$ can be computed in t_f time, since the LNS representation of 1 is 0. We will now find out the maximum time required for evaluating the arctangent function.

During one transformation, we evaluate $\frac{(1-xt)}{(1-x)}$, i.e., we need to do one multiplication (t_f time), one division (t_f time), and one addition ($3t_f$ time) and finally one special subtraction (t_f time). Thus, the time needed for one transformation is $6t_f$. For retransformation, we evaluate $\tan^{-1}(x) = \tan^{-1}(x) - \tan^{-1}(\frac{1}{k})$, where $\tan^{-1}(\frac{1}{k})$ is stored in a table. Thus, one subtraction and one table look-up are needed, making the time for one retransformation as $4t_f$.

The pth term of the modified series for $\tan^{-1}(x)$ is written as $t_p(x) = A_{p-1}x^{2p-2}(1-x^2)$. The factor $(1-x^2)$ can be evaluated once for all (by one multiplication and one subtraction) and stored in a register. The LNS representation of A_{p-1} will be stored in a table. The term x^{2p-1} can be evaluated from x^{2p-3} by multiplying it with x^2. These three factors can then be multiplied (2 multiplications) to generate $t_p(x)$. Hence, evaluation of each term needs three multiplications and one table look up. Finally, this term is to be added to the stored result. Thus, the total time needed for evaluation of one term and adding it to the stored result is $7t_f$.

If m is the number of transformations and p_{max} is the corresponding number of terms of the series expansion to be evaluated, then the total time needed for computing $\tan^{-1}(x)$ is $T_c = (10m - 7p_{max})t_f + 2t_f$, where the term $2t_f$ accounts for the evaluation of the factor $(1-x^2)$. Using Table I, we find T_c for different values of m, which are listed in Table III.

Examining Table II and Table III, we see that the product of the maximum error and the total time needed for computing $\tan^{-1}(x)$ is

minimum for $m = 3$. Hence, we conclude that the optimum number of transformations for computing $\tan^{-1}(x)$ is equal to 3. We also note that both the average error and the maximum error for $m = 3$ are not significantly higher than their respective minimum values (which correspond to $m = 8$).

3) Transformations on Different Values of the Argument: In Table IV, we show the specific transformations that the input arguments must undergo. From the table it is clear that the estimation of computational time, as reported in section IV-C-2 is rather conservative. Only a small range of values would undergo all the three transformations, while the rest would require less than three transformations.

V. EVALUATION OF TRIGONOMETRIC FUNCTIONS

We consider here only the evaluation of the Sine function. Other trigonometric functions can be evaluated using this function.

The series by which we can evaluate $\sin(x)$ is very much similar in nature to the series for evaluation of $\tan^{-1}(x)$ and is given by

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots, \qquad (9)$$

where $|x| \leq 1$. We would modify the series as in Section III, as $\sin(x) = \sum_0^p \left(1 - \frac{1}{3!} + \frac{1}{5!} - \cdots + \frac{(-1)^p}{(2p+1)!}\right)x^{2p+1}(1-x^2)$. Since $\sin(-x) = -\sin(x)$, evaluating $\sin(x)$ is enough to get $\sin(-x)$. Again, we can always express x as $x = n\frac{\pi}{2} - y$ where $y < \frac{\pi}{2}$ and $\sin(x) = \pm\sin(y)$ or $\pm\cos(y)$, depending on the values of n. Thus,

TABLE IV

Range of Input Argument	Transformation		
	First	Second	Third
1.000 - 0.721	Yes	No	No
0.720 - 0.619	Yes	No	Yes
0.618 - 0.524	Yes	Yes	Yes
0.523 - 0.415	Yes	Yes	No
0.414 - 0.313	No	Yes	No
0.312 - 0.237	No	Yes	Yes
0.236 - 0.163	No	No	Yes
0.162 - 0	No	No	No

TABLE V

Maximum Number of Transformations (m)	Maximum Number of Terms (p_{max})
1	32
2	9
3	5
4	3
5	3
6	3
7	3
8	3
9	2
10	2
11	1
12	1

TABLE VI

m	Average Error	Maximum Error
1	1.9E-6	2.8E-5
2	6.8E-8	4.2E-7
3	4.4E-8	2.3E-7
4	6.2E-8	3.2E-7
5	3.7E-8	2.2E-7
6	3.7E-8	2.2E-7
7	3.6E-8	2.2E-7
8	3.8E-8	2.0E-7
9	3.6E-8	2.0E-7
10	3.6E-8	2.0E-7
11	3.6E-8	2.0E-7
12	3.6E-8	2.0E-7

TABLE VII

m	T_c (in terms of t_f)
1	241
2	95
3	82
4	83
5	98
6	113
7	128
8	143
9	151
10	166
11	174
12	189

it is sufficient to evaluate $\sin(y)$ for $y < \frac{\pi}{2}$. But $y < \frac{\pi}{2}$ does not always guarantee that y will be less than 1 and this prevents us from using the Taylor series expansion of $\sin(x)$. So here again, we take the idea of folding as in Section IV to reduce the number of terms those are to be evaluated for computing $\sin(x)$.

If $y > \frac{\pi}{4}$ we transform y to y_1 as $y_1 = \frac{\pi}{2} - y$. Thus $\sin(y_1) = \cos(y)$ or, $\sin(y) = \sqrt{1 - \sin^2(y_1)}$. Again if $y_1 > \frac{\pi}{8}$, we use the transformation $y_2 = \frac{\pi}{4} - y_1$. Thus, $y_1 = \frac{\pi}{4} - y_2$ and hence, $\sin(y_1) = \frac{1}{\sqrt{2}}(\cos(y_2) - \sin(y_2))$. The terms y_m and y_{m-1} are related as $y_{m-1} = \frac{\pi}{2^m} - y_m$. Hence, $\sin(y_{m-1}) = \sin(\frac{\pi}{2^m})\cos(y_m) - \cos(\frac{\pi}{2^m})\sin(y_m)$. We assume that the values of $\sin(\frac{\pi}{2^m})$ and $\cos(\frac{\pi}{2^m})$ are stored in a table. Hence $\sin(y_{m-1})$ can be computed, if we can evaluate $\sin(y_m)$ from the truncated infinite series.

In terms of m, the maximum number of transformations, the inequality (5) takes the form for Sine function as,

$$\log_2\left(1 - \frac{1}{3!} + \frac{1}{5!} - \cdots + \frac{(-1)^p}{(2p+1)!}\right)$$
$$+ (2p-1)\log_2(\frac{\pi}{2^m}) + \log_2(1 - x^2) < -23. \quad (10)$$

Table V shows the different values of p_{max} for different m. To estimate the computational error for various m, we use a simulation program with $\epsilon = 2^{-23}$, for uniformly distributed ten million values of x between zero and one. The results were compared with the values of $\sin(x)$ calculated in floating-point with double precision. The observed differences are listed in Table VI. The proposed technique provides good accuracy for any reasonable value of m.

To estimate the time needed for evaluating $\sin(x)$, we follow the notations used in Section IV-C-2. In this case, the transformation involves one subtraction and one division (to compute $\frac{\pi}{2^m}$ from $\frac{\pi}{2^{m-1}}$). Thus the time needed for one transformation is $4t_f$. The retransformation involves two table look-up operations (for getting $\sin(\frac{\pi}{2^m})$ and $\cos(\frac{\pi}{2^m})$), 2 subtractions (one with $3t_f$ time and the other with t_f time), 3 multiplications, and 1 square-rooting. Assuming that the time needed for square-rooting is equal to t_f, the total time needed for one retransformation is $11t_f$.

The time needed for evaluation of a term and adding it to the stored result remains same as that for $\tan^{-1}(x)$, i.e., $7t_f$. If m is the number of transformations and p_{max} is the corresponding number of terms of the series expansion to be evaluated, then the total time for computing $\sin(x)$ is $T_c = (15m + 7p_{max})t_f + 2t_f$. Using Table V, we find the time T_c for different values of m, as listed in Table VII. We see from this table that the time to compute $\sin(x)$ is minimum for $m = 3$.

Examining Table VI and VII, we find that the product of the maximum error and the total computational time is minimum for $m = 3$. Hence, we conclude that the optimum number of transformations for evaluating Sine function is 3.

Table VIII shows the transformations undergone by the different values of the input argument. It is readily seen that only a small range of the input argument values undergo all the three transformations, while the rest would require less than three transformations.

VI. EVALUATION OF EXPONENTIAL AND LOGARITHMIC FUNCTIONS

LNS co-processors can also be used to evaluate exponential and logarithmic functions from their infinite series representations, using the idea of series folding. For example, to compute e^x, where $x > 1$, x is first transformed to a sufficiently small value $y = \frac{x}{2^k}$. Then e^x is evaluated from the series and finally e^y is squared k times to get e^x.

TABLE VIII

Range of Input Argument	Transformation		
	First	Second	Third
1.000 - 0.982	Yes	Yes	Yes
0.981 - 0.786	Yes	Yes	No
0.785 - 0.590	No	Yes	No
0.589 - 0.393	No	Yes	Yes
0.392 - 0.197	No	No	Yes
0.196 - 0	No	No	No

Fig. 3. Block diagram of the proposed hardware.

TABLE IX

m	T_c (in terms of t_p) (using parallel LNS unit)
1	36
2	34
3	38
4	42
5	49
6	56
7	63
8	70
9	77
10	81
11	88
12	95
13	99

Similarly, to compute $\log_2 x$, we can transform x to a sufficiently small value $y = \frac{x}{2^k}$. Evaluation of $\log_2 y$ can be done from the series expansion of $\log_2 y$ ($y < 1$), with the help of the LNS co-processor. Hence, $\log_2 x$ can be computed by adding k to $\log_2 y$.

However, we can do away with all such computations in evaluating exponential and logarithmic functions by a proper choice of the radix r, as explained below.

The conversion at the output of the LNS co-processor converts a number from the LNS format to the floating-point binary mode. Thus, given the integral part I and the fractional part F of the number in LNS, the output conversion is done to get $x = r^{I+F}$. This is done by a table look-up. Choosing r = e instead of the usual value 2, we can directly evaluate the exponential function by the conversion unit.

Similarly, the conversion process at the input of the LNS co-processor will directly give the value of $\log_e x$ for a given binary floating-point number x. The input conversion process in this case will, however, be a bit more complex. Consider, for example, a floating-point binary number $x = M2^E$. To convert x to LNS radix 2, we need to evaluate the integral part I and the fractional part F in the LNS format where, $2^{I+F} = x = M2^E$ i.e., $I + F = \log_2 M + E$. Now, $\log_2 M$ can be obtained by a table look-up. On the other hand, if r = e, then we would have, $e^{I+F} = x$, i.e., $I + F = \log_e M + E \log_e 2$.

The multiplication operation between E and $\log_e 2$ can be overlapped in time with the table look-up operation to find $\log_e M$. If we allow this added complexity in the input conversion process from floating-point to LNS, then logarithmic functions can be evaluated with no extra computation.

VII. HARDWARE IMPLEMENTATION

The implementation of all these functions would require some hardware support. To illustrate the case, we propose the hardware that is required to evaluate $\tan^{-1}(x)$ as shown in Fig. 3. It consists of two parts : 1) Term evaluator unit and 2) Summing unit. The advantage of this scheme is that the extra hardware can be fabricated using the existing LNS adder or the multiplier that is actually present in the LNS co-processor. The proposed scheme is heavily pipelined.

A. Term Evaluator Unit

The term evaluator unit consists only of 3 registers R1, R2 and R3; 3 multipliers M1, M2 and M3; and one table T_1 containing ($p_{max} - 1$) entries (LNS form of A_0, that is 0, is not stored). The registers R1, R2 and R3 are first fed with the values x, x^2 and ($1-x^2$) respectively. The table contains the LNS forms of the coefficients, A_{p-1} i.e., $(1 - \frac{1}{3})$, $(1 - \frac{1}{3} + \frac{1}{5})$, etc., up to p_{max} such terms. The table is a set of registers with a table pointer which increases itself
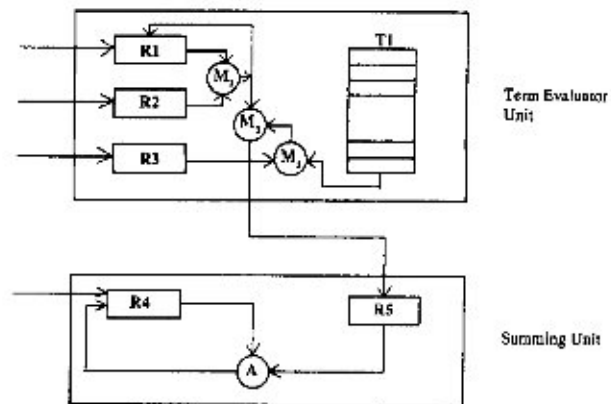
by 1, whenever one access to the table is made. The pointer initially points to the first entry of the table.

In the first time unit the multiplier M_1 multiplies the contents of the registers R1 and R2, and the multiplier M_2 multiplies the contents of the register R3 and the first entry of the table T_1. The output of M_1 is fed back to the register R1 and also fed to the input of the multiplier M_3, while the other input of M_3 comes from the output of M_2. During the second time instant M_3 computes the product of its two inputs while M_1 multiplies x^3 with the content of R2, and M_2 multiplies $(1 - x^2)$ with the second entry of the table. Recalling that $t_p = A_{p-1}x^{2p-1}(1 - x^2)$, M_1 generatres the portion x^{2p-1} of t_p and M_2 generates the product $A_{p-1}(1 - x^2)$. Finally, M_3 multiplies the two results generated by M_1 and M_2 to produce the term t_p.

B. Summing Unit

The summing unit contains 2 registers R4 and R5 and one LNS adder. The register R4 is fed with the value $x(1 - x^2)$ and R5 gets the output of the term evaluator unit. As the term evaluator unit is pipelined and it can generate the output in every time instant, the summing unit can operate at the same speed with the term evaluator unit. The adder adds the contents of R4 and R5 and puts the result back in R4. The final output can be taken from the register R4 itself. As we have seen that the addition in LNS is a slower process than multiplication, we may require a buffer between the term evaluator unit and the summing unit.

With such a pipelined implementation, we can easily verify that the time needed for computing $\tan^{-1}(x)$ can be reduced to $T_c = (7m + 3p_{max} + 2)t_j$. Table IX shows the values of T_c for different values of m. The time-error product is still minimum for $m = 3$.

Using the very same pipeline, just changing the contents of the table T_1, the whole setup can be used to evaluate $\sin(x)$ also.

VIII. CONCLUSION

We have shown that the four common functions namely, inverse trigonometric functions, trigonometric functions, the exponential function, and the logarithmic function that are very frequently used, can be implemented by using an LNS co-processor. The advantage of our idea based on series folding is that as the number of iterations required can be known *a priori*, the time for computing a given function can easily be estimated. Further works are also being carried out to evaluate some other functions like the hyperbolic trigonometric functions or the fast Fourier transform in a similar framework.

REFERENCES

[1] E. E. Swartzlander and A. G. Alexopoulous, "The signed logarithm number system," *IEEE Trans. Comput.*, vol. C-24, pp. 1238–1242, Dec. 1975.

[2] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 bit logarithmic number system processor," *IEEE Trans. Comput.*, vol. 37, pp. 190–200, Feb. 1988.

[3] D. M. Lewis, "An architecture for addition and subtraction of long word length numbers in the logarithmic number system," *IEEE Trans. Comput.*, vol. 39, pp. 1325–1336, Nov. 1990.

[4] H. Y. Lo and Y. Aoki, "Generation of a precise binary logarithm with difference grouping programmable logic array," *IEEE Trans. Comput.*, vol. C-34, pp. 681–691, Aug. 1985.

[5] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Meszlenyi, J. R. Rice, H. G. Thacher, Jr., and C. Witzgall, *Computer Approximations*. New York: John Wiley, 1968.