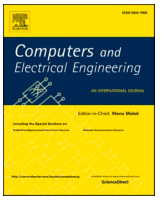
Computers and Electrical Engineering 87 (2020) 106800

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng



Computing with logarithmic number system arithmetic: Implementation methods and performance benefits

Behrooz Parhami *

Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560, USA

ARTICLE INFO

Keywords:
ALU design
Computation errors
Interpolation
Logarithmic arithmetic
Performance per watt
Real arithmetic

ABSTRACT

The logarithmic number system (LNS) has found appeal in digital arithmetic because it allows multiplication and division to be performed much faster and more accurately than with widely-used floating-point (FP) number formats. We present a comprehensive review and comparison of various techniques and architectures for performing arithmetic operations efficiently in LNS, using the sign/logarithm format, and focus on the European Logarithmic Microprocessor (ELM), a device built in the framework of a research project launched in 1999, as an important case study. Comparison of the arithmetic performance of ELM with that of a commercial superscalar pipelined FP processor of the same vintage and technology confirms that LNS has the potential for successful deployment in general-purpose systems. Besides paying due attention to LNS attributes beyond computational speed and accuracy, novel contributions of this survey include an exploration of the relationship of LNS with the emerging field of approximate computing and a discussion of the discrete logarithmic number system.

1. Introduction

Proposals for the logarithmic number system (LNS), a representation scheme where the representation of a number consists of the sign of the number appended to the logarithm of its absolute value, emerged in the 1970s. In 1971, N. G. Kingsbury and P. J. W. Rayner introduced "logarithmic arithmetic" for digital signal processing. Later, in 1975, a similar logarithmic number system was proposed by E. E. Swartzlander and A. G. Alexopoulos, who used scaling to avoid negative logarithms. Independently, S. C. Lee and A. D. Edgar proposed a similar number representation method in 1977. These proposals were all motivated by simple multiplication and division in LNS as well as its superior error characteristics compared with both fixed- and floating-point arithmetic. By the 2000s, features and benefits of LNS were being described in textbooks and technical reference books [1,2].

The sign/logarithm number system speeds up multiplication and division relative to conventional weighted numbers, while also avoiding the problems inherent in a residue number system (RNS), including very complex/slow division and reconversion to binary. This advantage, however, is offset by the fact that addition and subtraction operations require relatively complicated procedures. Despite its attractive properties, until recently, few implementations of LNS arithmetic were attempted, all of which were restricted to low-precision applications, the difficulty in performing addition and subtraction on long words being the principal reason. LNS addition and subtraction require lookup tables whose size grows exponentially with the logarithm width. For this reason, implementations described in the early literature were limited to 8–12 bits of fractional precision [3].

* Corresponding author.
E-mail address: parhami@ece.ucsb.edu.

https://doi.org/10.1016/j.compeleceng.2020.106800
Received 24 August 2019; Received in revised form 24 July 2020; Accepted 28 July 2020
Available online 29 August 2020

It was recognized early on that LNS can offer an advantage over floating-point (FP) representation only if LNS addition and subtraction can be performed with the speed and accuracy comparable to those of FP. However, achieving this goal is complicated by the fact that these operations entail the evaluation of nonlinear functions. Interpolation is useful for reducing the size of a lookup table. Linear approximation, quadratic approximation, and linear approximation with a nonlinear function in a PLA have been used to advantage in the approximation of the function $\log x$. The latter method is only described for addition. Taylor's conclusion in [3] was that a hardware implementation of linear approximation for logarithmic arithmetic is impractical. Nevertheless, Lewis [4] presented an algorithm for logarithmic addition and subtraction using 32-bit representations, with 30-bit exponents containing 22 fractional bits.

Development of LNS-based commercial systems has long been contemplated. D. Yu and D. M. Lewis, V. Paliouras et al., and M. G. Arnold proposed architectures for LNS-based processors, but did not present finished designs or extensive simulation results. At nearly the same time, a European project, initiated by Coleman et al. [5,6], laid down the foundations for the development of a commercial digital system, dubbed the European Logarithmic Microprocessor (ELM). Comparison of ELM with TMS320C6711, a commercial superscalar pipelined FP processor of the same vintage and technology, provided encouraging results in regards to speed and accuracy of the arithmetic operations [7], establishing the project's success.

Modern computation-intensive applications are characterized by increased algorithmic complexity as well as a rise in problem and data sizes. Thus, a great many applications are becoming bounded by the speed of FP operations. Memory and I/O are also becoming major bottlenecks, but these problems are independent of the choice of LNS. Real-time applications in this class are exemplified by RLS-based algorithms (see Section VIII), subspace methods required in broadcasting and cellular telephony, Kalman filtering, and Riccati-like equations for advanced real-time control. Graphics systems provide another case in point. Ways are urgently being sought to overcome this limitation, and LNS seems to provide a viable solution.

The Gravity Pipe special-purpose supercomputer (GRAPE), that won the Gordon Bell Prize in 1999, used LNS representation and arithmetic. LNS is commonly used as part of hidden Markov models, exemplified by the Viterbi algorithm, with applications in speech recognition and DNA sequencing. A substantial effort to explore the applicability of LNS as a viable alternative to FP for general-purpose processing of single-precision real numbers has been expended by researchers over the past two decades, leading to the demonstration of LNS as an alternative to floating-point, with improved accuracy and speed. Surveying these efforts and the current ongoing research in the field has been a major motivation in writing this paper, which not only updates previous survey papers but also covers the relationship of LNS with the emerging field of approximate computing, fueling novel applications in machine learning and neural networks, and includes a discussion of the discrete logarithmic number system.

It is worth noting the long history and intuitive appeal of logarithmic number representation. For decades, before the age of electronic digital calculators, log-scale-based slide rules were companions to engineers and helped them perform back-of-the-envelope calculations quickly and efficiently. Because of the pioneering work of A. Niedler and E. K. Miller, there is now broad agreement that the mental number line in humans and primates is logarithmic rather than linear.

Here is a roadmap for the rest of this paper, a short version of which has been published previously [8]. Arithmetic algorithms for LNS and their comparison (in terms of circuit complexity, power consumption, and error characteristics) with FP arithmetic are discussed in Section II. Section III outlines table-lookup methods for interpolating nonlinear functions, as required for LNS addition and subtraction, followed by the relatively recent methods of using Taylor series and similar expansions. Error compensation algorithms, used after interpolation, are also discussed. Section IV demonstrates that VLSI realizations of these algorithms entail latencies that are essentially comparable to those of an equivalent FP unit. Section V covers the architecture and philosophy adopted by the European research team that designed ELM. Section VI details the physical realization of ELM and compares its speed and accuracy with those of a commercial superscalar and pipelined FP processor. Section VII outlines other performance criteria, such as energy efficiency in terms of computations per watt. Section VIII touches upon some application domains where LNS has been used and compares the resulting performance with that of a comparable FP system. Section IX contains a brief review of the good fit of LNS to a number of low-precision applications that serve to connect it to the emerging field of approximate computing. Section X introduces the notion of discrete logarithms and discusses their use in number representation. Section XI concludes the paper and discusses possible areas for future work.

2. Logarithmic number system

2.1. Number representation

The sign/logarithm representation of a number consists of the sign of the number appended to the logarithm of the absolute value of the number, with special arrangement to show the zero value. This system thus avoids the classic problem with the basic logarithmic number systems: the inability to represent negative numbers. In the sign/logarithm number system, a number x is represented by its sign S_x and the binary logarithm of its magnitude L_x as:

$$S_x = \begin{cases} 1 & x < 0 \\ 0 & x > 0 \end{cases} \qquad L_x = \log_2 |x| \tag{1}$$

Logarithmic numbers can be viewed as particular instances of a FP number system, where the significand M_x in the floating-point number $(-1)^{Sx} \times M_x \times 2^{Ex}$ is always 1.0 and the exponent E_x has a fractional part rather than being an integer. The number x with sign S_x and fixed-point logarithm L_x has a value of:

$$x = (-1)^{Sx} \times 2^{Lx} \tag{2}$$

The logarithm L_x is a 2′s-complement fixed-point number, with its negative values corresponding to numbers of magnitude less than 1.0. In this way LNS numbers can represent both very large and very small numbers, as is the case for FP numbers.

Fig. 1 shows an LNS representation equivalent to the 32-bit IEEE standard FP representation [5]. Here, the integer and fractional parts form a 2′s-complement fixed-point value ranging from –128 to approximately +128. The real numbers represented are signed and have magnitudes ranging from $2^{-128}$to $\sim 2^{+128}$ (i.e., from 2.9×10^{-39} to $3.4 \times 10^{+38}$). The smallest representable positive value, 40000000_{16}, is used as a special code for zero, while $C0000000_{16}$ is dedicated to represent NaN. Thus, we have an LNS representation that covers the entire range of the corresponding IEEE FP short format.

The primary motivation for using LNS over FP is the fact that multiplication and division in LNS are extremely fast and simple compared with the FP system. The use of LNS is thus attractive in applications that entail a large number of multiplications and divisions. Another advantage is LNS's uniform geometric error characteristics across the entire range of values. This leads to roughly an additional 1/2 bit of precision compared with a FP representation using the same number of bits. Thus, in signal-processing applications, LNS offers better signal-to-noise ratio as well as a better dynamic range.

2.2. Arithmetic operations

Let's assume that we want to derive a sign/logarithm result by performing each of the basic arithmetic operations on a pair of sign/logarithm operands (S_x, L_x) and (S_y, L_y). Multiplication becomes a simple operation in LNS. The logarithm of the product is computed by adding the two fixed-point logarithms. This is evident from the following logarithmic property:

$$\log_2(x \times y) = \log_2(x) + \log_2(y) \tag{3}$$

The sign is the XOR of the multiplier's and multiplicand's sign bits. Since the logarithmic numbers are 2′s-complement fixed point numbers, addition is an exact operation if there is no overflow, while overflow events (correctly) result in $\pm\infty$. For division:

$$\log_2(x / y) = \log_2(x) - \log_2(y) \tag{4}$$

The only difference here is the possibility of underflow, which occurs when the difference of the logarithms is a negative number with too large a magnitude to be represented in the available word width.

Thus, LNS multiplication and division operations are defined as:

$$\text{Multiplication} : S_p = S_x \oplus S_y, \quad L_p = L_x + L_y \tag{5}$$

$$\text{Division} : S_q = S_x \oplus S_y, \quad L_q = L_x - L_y \tag{6}$$

The ease of the LNS calculations above is in stark contrast to what is required to implement the addition and subtraction operations, to be described next. Given two LNS numbers x and y, with $|x| \geq |y|$, we can compute $z = x \pm y$ as follows:

$$S_z = S_x \tag{7}$$

$$L_z = \log_2|x \pm y| = \log_2|x(1 \pm y/x)| = \log_2|x| + \log_2|1 \pm y/x| = \log_2|x| + \log_2\left|1 \pm 2^{(Ly - Lx)}\right| \tag{8}$$

Let $d = L_y - L_x \leq 0$ and $\phi^{\pm}(d) = \log_2|1 \pm 2^d|$. The value of $\phi^{\pm}(d)$, shown in Fig. 2, can be calculated and stored in a ROM for the two cases of addition and subtraction, but this is not feasible for wide words. Other realizations, based on interpolating the nonlinear functions $\phi^{\pm}(d)$, are discussed in Section III. For now, we will assume that a ROM is sufficient for our purpose.

Realization of Eqns. (5), (6), and (8) by means of a comparator, an adder, a subtractor, a read-only memory (ROM) table, two multiplexers, and a small amount of peripheral logic can result in a simple four-function ALU [1], as shown in Fig. 3. For addition/subtraction, the left adder-subtractor computes $L_y - L_x$, the lookup table provides the value of $\log_2|1 \pm 2^{(Ly - Lx)}|$, and the right adder-subtractor perform the addition of Eq. (8). For multiplication and division, the left adder-subtractor performs the required addition or subtraction of logarithms, as specified in Eqs. (3) and (4), and the right adder-subtractor takes care of the scale factor: Each number is scaled up by m to obviate the need for negative logarithms, so $\log m = L_m$ must be subtracted from the multiplication result and added to the division result, so as to produce a properly-scaled output.

The latency of the ALU in Fig. 3 is $T_{OP} = T_{COMP} + 2T_{ADD} + T_{ROM}$, where T_{COMP} is the delay of a comparator. The peripheral logic delay is assumed to be negligible. When the comparator is implemented with a subtractor, we have $T_{COMP} = T_{ADD}$, and:

$$T_{OP} = 3T_{ADD} + T_{ROM} \tag{9}$$

2.3. Forward and reverse conversions

Conversion from binary to logarithmic representation (forward conversion) entails computation of logarithm. The reverse

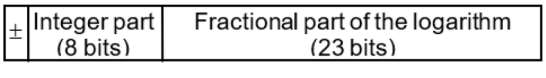


Fig. 1. A 32-bit LNS number format

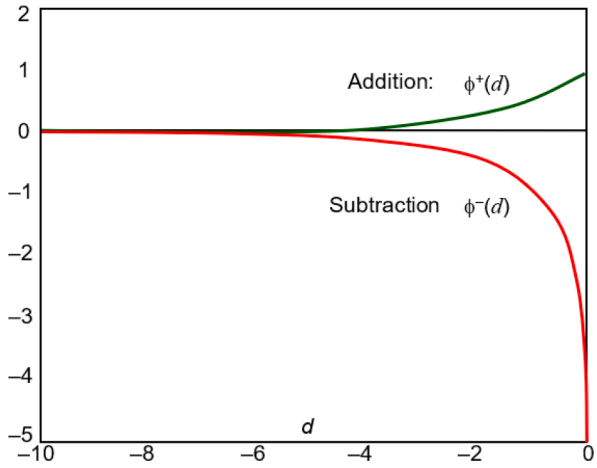


Fig. 2. Plots of $\phi^+(d)$ and $\phi^-(d)$ as functions of d.

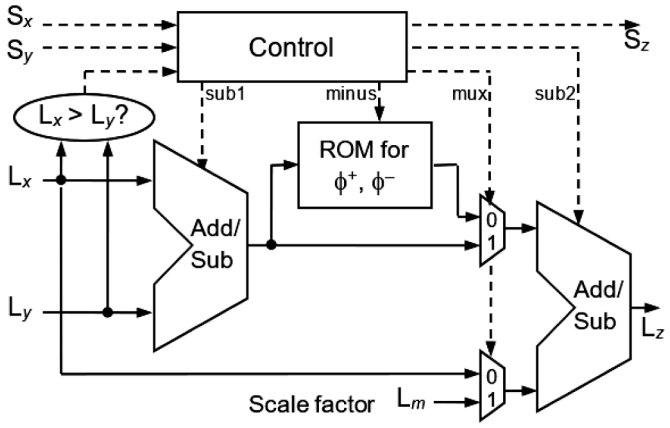


Fig. 3. A complete four-function ALU for LNS [1].

conversion, needed at the end of a sequence of computations to present the result in binary format consists of antilogarithm computation or exponentiation. Both of these functions have been studied extensively, as they are needed in many other contexts besides logarithmic number representation. Whereas accuracy is important in both conversions, speed of conversion and circuit complexity are perhaps even more important, in order to keep the conversion time and circuit overheads within reason. Each conversion scheme proposed typically accompanies error analysis.

First, let us review a method for forward conversion as an example of the wide array of available schemes. The logarithmic converter of Kim et al. [9] uses 8-way partitioning of the interval [0, 1] to improve upon the linear method of J. N. Mitchell and a somewhat more accurate, but still crude, method of T. B. Juang et al., introducing the relative errors of 5.9% and 2.9%, respectively. Coefficients for the linear approximations $a_i x + b_i$ of the 8 intervals are stored in 8-entry tables.

Optimal coefficient values are determined and near-optimal values are chosen for a and b to reduce the hardware and power requirements. Shift-and-add is used for computing $a_i x$. The hardware needed for the entire computation consists of a 32-bit leading 0s counter, a variable shifter, an integer unit producing the 6-bit whole part of the result, and a fractional unit yielding the remaining 26 bits of the logarithm. The latter unit, accounting for most of the latency, uses three 26-bit carry-save adders, arranged in 3 levels (the inputs to which come from look-up tables and hardwired shifters), feeding a 26-bit carry-propagate adder. Errors for this converter range from –0.190%, occurring in the first subinterval [0, 1/8), to +0.103%, occurring at 3/8.

Other designs for forward-converters abound, so we cite just a few examples. M. Chaudhary and P. Lee present a design suitable for FPGA realization, as it uses less than 2 Kb of ROM, along with 3 multipliers. Other proposals include decimal converters, conversion schemes for use in binary multiplication/division via logarithmic conversions [10], approximation methods [11], and domain-specific designs tuned to the requirements of a particular application or class of applications.

We next turn to reverse conversion via computing the function 2^x, again using the antilogarithmic converter of Kim et al. [9], as an example. This unit also uses 8-way partitioning of the interval [0, 1] for piecewise linear approximations $c_i x + d_i$. Optimal values for c_i and d_i are determined and near-optimal values are chosen to reduce complexity. Shift-and-add operations are used for computing $c_i x$.

Errors for this reverse converter range from –0.070%, occurring at 3/8, to +0.082%, occurring in the sixth subinterval [5/8, 3/4). A minor modification of this scheme has been shown to reduce the relative error at negligible cost [12].

A number of authors have dealt with both the forward and reverse conversion problems [13].

Thus far, we have shown forward and reverse conversions to be feasible with acceptable latency and circuit cost. Given the importance of these conversions to LNS applications, we devote the rest of this subsection to an in-depth review on the subject.

Methods for computing logarithms and exponentials have a long history, dating back to the mid-1600s. Initially, the objectives were to make manual calculations easier, faster, and more accurate. With the advent of programmable electronic computers, software algorithms became the main focus, given that only adders/subtractors, and later multipliers/dividers, were to be used. With the invention of the CORDIC method [1], custom hardware provided greater speed but required its own design strategies and algorithms. Use of reconfigurable circuit components brought about the need for a reexamination of design methodologies, as did more stringent requirements for energy efficiency.

A recent contribution to the field of logarithmic conversion [14] also traces the previous efforts in this area and provides circuit-complexity and power comparisons for various methods and implementations. The key to improved accuracy and power dissipation is non-uniform partitioning of the function domain, depending on the range and precision of the input. Even though the method has been devised and evaluated for integer inputs, similar techniques are applicable to FP logarithmic converters.

Specific examples from [9] might be helpful in understanding the nature of the method used. Consider the computation of $\log_2[2^k(1+x)]$, where $0 \le x < 1$. The required logarithm is $k + \log_2(1+x)$. The second term can be approximated as $x + \lambda(x)$, with the determination of $\lambda(x)$ being the primary computational task. An algorithm that needs a few seconds of running time is used to find the optimal number of segments for a given target maximum error, with the number of segments obtained typically being an order of magnitude or more below the number needed with uniform segmentation.

For instance, setting the result error at 4.6% leads to only 4 segments, whose widths, expressed in units of 1/8 (looking at the 3 most-significant bits of x) are: 1, 3, 2, 2. Setting the result error at the more stringent 0.42% leads to the use of 11 non-uniform segments as follows, where the length of each segment is expressed as a multiple of 1/64 (examining the 6 most-significant bits of x): 1, 2, 4, 6. 10, 11, 11, 9, 6, 3, 1 (see Fig. 4 in [14]). Notice that segments are narrower in the vicinity of $x = 0$ and $x = 1$, an outcome that could have been anticipated, given the shape and the curvature of the logarithm function.

3. Addition and subtraction algorithms

The focus in this section is on the efficient calculation of the nonlinear term $\phi^{\pm}(d) = \log_2|1 \pm 2^d|$ for $d = L_y - L_x$ (or $d = j - i$, taking L_y as j and L_x as i for simplicity). Implementation work began by E. E. Swartzlander and A. G. Alexopoulos in 1975, entailing a 12-bit device. In 1988, F. J. Taylor et al. extended the scheme to 20 bits. Both designs used direct implementations of Eqn. (8), with a lookup table or ROM covering all possible values of $\phi^{\pm}(d)$, omitting those which quantize to 0.

It is apparent that, as the word width increases, table sizes increase exponentially, which limits the practical utility of this approach to about 20 bits. One solution to the problem of exponential complexity is to have a lookup table only for selected intermediate values and interpolate the intervening values using linear expansion and approximation. A design by D. Yu and D. M. Lewis extended the word width to 28 bits by implementing the lookup table for d values only at intervals of Δ. Any negative value of d satisfies $d = -h\Delta - \delta$ for some integer value h, leading to the Taylor-series expansion of $F(d)$:

$$F(d) = F(-h\Delta) - \frac{D(-h\Delta)\delta}{1!} + \frac{D'(-h\Delta)\delta^2}{2!} - \ldots \tag{10}$$

The latter design used only the first-order term, requiring the additional storage of a table of derivatives $D(d)$ and introducing a multiplication into the circuit's critical path. The scheme also exposed a further problem intrinsic to LNS arithmetic: the difficulty of interpolating $\phi^-(d)$ in the region $-1 < d < 0$. It can be seen from the graph in Fig. 2 that $\phi^-(d)$ and its first- and second-order derivatives tend to $-\infty$ as d goes to 0. To maintain accuracy, it is necessary to implement a large number of successively smaller intervals as $d \to 0$.

Coleman et al. [5] published the design of a 32-bit logarithmic adder/subtractor (Fig. 4) with speed and accuracy comparable to that of the FP system. Their starting point was the first-order Taylor-series approximation, the critical path of which contains a ROM, a

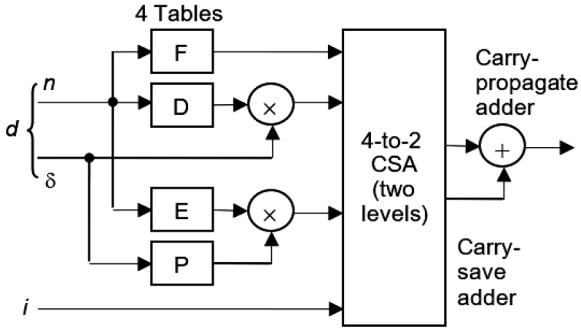


Fig. 4. An LNS adder/subtractor implementation [6].

multiplier, and two carry-propagate adders, which is at the limit of what can be implemented without significantly exceeding the delay of FP addition. To counter the difficulty of subtraction for $-1 < d < 0$, they employed a range-shifter (see Section 4) to transform such a subtraction into one for which $d < -1$, with an extra time delay of only one ROM stage, a carry-propagate adder, and a carry-save stage. Another major problem is that at 32-bit word width, a first-order Taylor approximation results either in too high an error or too large a table. A possible solution [5] entails the use of a crude first-order approximation, along with simultaneous assessment of the resulting error, which is then incorporated into the result as a correction term. This correction entails only one extra carry-save addition stage.

As discussed earlier, to minimize the lookup table size, Δ can be progressively increased as the function becomes more linear with decreasing d (toward more-negative values). An intervening value of d lying in the hth interval, $0 \le h < n$, is expressed as:

$$d = -\delta \{h = 0\}; \tag{11}$$

$$d = \sum_{h=0}^{n-1} (-\Delta_h) - \delta \{h > 0\}$$

In a typical implementation, Δ is doubled at the next power of 2. For simplicity, however, we ignore the variation in Δ and use:

$$d = -h\Delta - \delta \tag{12}$$

Together with each value of $F(d)$, is stored its derivative $D(d)$. The function at an intervening value of d is then obtained by linear interpolation:

$$F(-h\Delta - \delta) \approx F(-h\Delta) - D(-h\Delta)\delta \tag{13}$$

Following the initial subtraction to obtain d, the latter is partitioned via division by Δ. The high-order bits represent h and are used to access the F and D tables, while the low-order bits represent δ. $F(-h\Delta)$ is then combined with the product $D(-h\Delta)\delta$ to obtain the approximation to $F(d)$, which is added to i to yield the result. The circuit consisting of the tables E and P, along with the multiplier to their right and one level of the carry-save adder (CSA) in Fig. 4 is for error-correction, as described next.

The interpolation scheme just discussed doesn't give an exact result and leaves an error given by:

$$\varepsilon(h, \delta) = F(-h\Delta) - D(-h\Delta)\delta - F(-h\Delta - \delta) \tag{14}$$

For each value of h, the absolute error ε increases with δ to a maximum value $\varepsilon_{max}(h) = E(h)$ given as:

$$E(h) = F(-h\Delta) - D(-h\Delta)\Delta - F(-h\Delta - \Delta) \tag{15}$$

The error in the estimation increases from zero when the required value lies on a stored point, to $E(h)$ when it falls just slightly to the left of the next stored point. It was observed that the shape of this error curve is very similar in all the intervals of both curves. A separate table stores the normalized shape of the common error curve, from 0 when the required value lies on an interpolation point, to 1. This table is known as the P (proportion) function. The error is calculated by multiplying $E(h)$ by P and is then added to the result of the interpolation, as shown in Fig. 4.

The error-correction algorithm of Coleman et al. [5] is based on the fact that, for a given δ, the ratio $P(h, \delta) = \varepsilon(h, \delta)/E(h)$ is roughly constant for all h. It is therefore possible to store, for one h, a table P of the error at successive points throughout Δ, expressed as a proportion of the maximum error E attained in that interval. It is also necessary to store, together with F and D for each interval, the value of the maximum error E. The error ε is then obtained for any (h, δ) as $\varepsilon(h, \delta) \approx E(h) \times P(c, \delta)$, where c is a constant. The lookups of $E(h)$ and $P(c, \delta)$ can be performed at the same time as those of $F(h)$ and $D(h)$, with their product evaluated in parallel with the multiplication in the interpolation. The expression $\varepsilon(h, \delta) \approx E(h) \times P(c, \delta)$ evaluates to:

$$E(h)P(c, \delta) = \left[\frac{D'(-h\Delta)\delta^2}{2!} + \frac{D''(-h\Delta)\delta^2\Delta}{3!} + \frac{D'''(-h\Delta)\delta^2\Delta^2}{4!} + \ldots\right] A(c, \delta) \tag{16}$$

$$A(c, \delta) = \left[\frac{D'(c) + \frac{1}{3}D''(c)\delta + \frac{1}{12}D'''(c)\delta^2 + \ldots}{D'(c) + \frac{1}{3}D''(c)\Delta + \frac{1}{12}D'''(c)\Delta^2 + \ldots}\right] \tag{17}$$

For $c = 0$, $A(c, \delta) \approx 1$ for all values of δ. The reader should consult reference [15] for details on this error correction scheme, along with its theoretical justification and experimental evaluation.

4. LNS arithmetic unit design

4.1. LNS ALU overall design

As noted by Coleman et al. [5], interpolation is difficult for subtractions in the region $-1 < d < 0$. Because the range $-1 < d < -0.5$ is not as problematic, the smaller range $-0.5 < d < 0$ is targeted for range shifting to reduce the required table size. The range-shift algorithm obviates such problematic subtractions by transforming i and j into new values that yield $d < -1$ (see [15] for details). It has a delay of one ROM access, a carry-propagate addition, and a carry-save addition stage.

The ALU for ELM has two separate circuits for add/subtract and multiply/divide. The add/subtract circuit is preceded by a range shifter and associated control logic, followed by a mux to select either the unmodified or range-shifted inputs, depending on the value of *d*. In the special cases involving one or two zero operands, or when the two operands are equal, the result may either follow one of the operands or be zero itself. The two input operands and the value 0 are therefore made available to a final 4-way mux, the setting of which is determined by the control logic.

4.2. VLSI implementation

The unit just described was designed in a 0.7μ 2-level-metal standard-cell system in Cadence and simulated for results for its VLSI implementation. This version of the design was made only for the purpose of comparison with FP system, and a contemporary variant was used in ELM. Typical delays through the critical paths were measured with the timing simulator, with results presented in Table 1. The cell library did not include an asynchronous ROM, so Coleman et al. quoted times for unrouted designs, incorporating an assumed 5 ns access time for suitable ROM devices.

For comparison, Coleman et al. also designed a 32-bit FP unit and a 32-bit fixed-point unit on the same lines as the LNS unit. As far as possible, blocks were reused from the LNS design. They didn't design a FP or fixed-point divider for comparing division times with LNS, but it is known that division typically takes 2–3 times as long as multiplication.

Depending on whether the range-shifter is required in a particular subtraction, two timing values are shown for subtraction in Table 1. Assuming that the range-shifter comes into play in 50% of subtractions on average, the mean subtraction time is 35 ns. With an equal mix of additions and subtractions, the average add/subtract time is about 31 ns, and the average multiply/divide time is 4 ns. Thus, the LNS unit would reach roughly twice the speed of FP at an add/multiply ratio of about 40/60 percent. If the add/multiply ratio is changed to 50/50, the LNS unit will have a performance comparable to that of fixed point.

4.3. Other design considerations

Over the years, many complete LNS arithmetic processors have been designed for real and complex computations. There has also been some discussion of design tools for LNS, although there remains much room for contributions in this area. Memory function interpolation schemes, their FPGA realizations, and, more generally, table-lookup schemes for function evaluation [16] should be examined for possible application to LNS addition and subtraction operations.

5. Architectural considerations

LNS ALUs have markedly different characteristics from their FP counterparts and therefore require some reevaluation of the surrounding microprocessor design to deploy them to best advantage. The optimal structure for memory and other subsystems may be different for LNS-based processors than for conventional platforms.

V. Paliouras et al. used Lewis's interleaved memory function interpolators [4] as the basis for the addition unit in a proposed very long instruction word (VLIW) device optimized for filtering and comprised of two independent ALUs, each with a 4-stage pipelined adder and a single-cycle multiplier. M. G. Arnold also proposed a VLIW device with one single-cycle multiplier and a 4-stage pipelined adder, suggesting that large-scale replication of functional units could lead to difficulties with the complexity of the multiplexing paths leading to and from the registers.

Coleman et al. [5] noted that the small size of the LNS multiplier-integer unit in their ELM proposal encouraged its replication, leading to a design with 4 single-cycle multipliers and 2 multicycle adders. They maintain, based on simulations results, that the basic addition/subtraction unit would incur about 3 times the delay of a fixed-point addition and that the range shifter, when invoked, would delay it by an additional cycle. This leaves the LNS adder/subtractor as the only circuit block with a delay greater than a cycle and it does not seem worthwhile to devise a pipelining scheme for this operation alone.

The variable delay time would also complicate the pipeline control algorithm. Hence, Coleman et al. [5] decided against pipelining the adder unit. Instead, they suggested a fully interlocked pipeline (Fig. 5) in which a single-cycle ALU executes all operations except LNS addition and subtraction. The latter are handled by the multicycle ALU, which executes with a 3-cycle latency. There is a possibility that both ALUs may complete at the same time, so the register file update paths are designed with enough bandwidth to accommodate this. ELM's instruction pipeline, shown in Fig. 5, begins with the standard early stages of instruction-cache access, instruction issue, and data-cache access. The pipeline ends with the ALU stage containing four single-cycle multipliers and two multicycle adders, as described above, to complete instruction execution and to send computed results to the general registers.

To boost throughput, functional unit replication has been employed here. Unlike functional unit pipelining, this is a very

Table 1
Latencies of VLSI arithmetic circuits (ns), according to the study by Coleman et al. [5].

Operation	Fixed	FP	LNS
Add	4	28	28
Subtract	4	28	28 / 42
Multiply	32	22	4
Divide	–	–	4

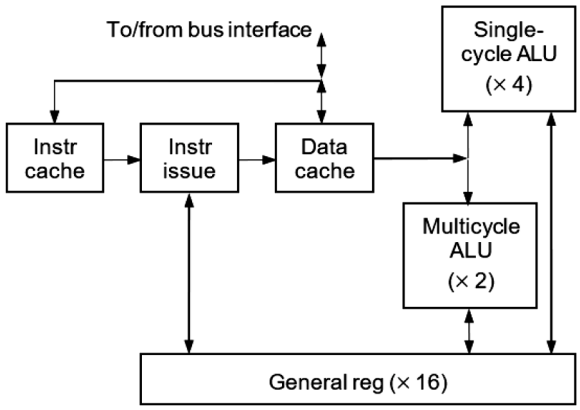


Fig. 5. ELM pipeline [6].

appropriate strategy for ELM, where the single-cycle multiply/divide unit is so small that it can readily be replicated four times. A vector capability is thereby provided, permitting the execution of four integer, logical, or LNS multiplication, division, or square root operations in one clock cycle. The vast majority of the silicon area used for the arithmetic circuitry can thus be reserved for the more substantial multicycle ALU. Two such units are provided, allowing two additions or subtractions to proceed in three cycles.

Mainstream computers favor a register-register architecture, with the data cache decoupled from the pipeline via a load/store unit to permit its independent concurrent operation. However, ELM designers focused on a dual RR/RM (register-register/register-memory) instruction-set architecture, with a cache incorporated into the pipeline. Accordingly, all instructions are available in either scalar or vector form, the former operating on only one memory location or register, and the latter on a set of four consecutive locations or registers for multiplication and division (two in the case of addition and subtraction). A hybrid mode is available to apply a scalar to all elements of a vector.

To maintain the required memory bandwidth, instruction and data caches supply, respectively, one and four 32-bit words to the processor per cycle. Each cache is 8 KB in size and is 2-way set-associative. A 64-bit asynchronous external bus interface is provided, with a minimum transaction time of 3 cycles per read and 4 per write. Memory-mapped I/O devices are accessed by polling and transfer 32 bits per operation. They also have recourse to a single-level non-maskable interrupt.

ELM has an interlocked pipeline and also an RM ISA. The two features together make for a relatively straightforward assembly language interface, which is currently the only way of programming the device [17]. On ELM, it is possible to launch two logarithmic additions in one cycle and then to process four loads and four multiplications in the remaining two cycles before the addition completes. The four multiplications perform implicit loads, so in all, eight words are loaded in the two cycles. The maximum execution rate on this device is thus two additions, four multiplications, and eight loads in three cycles.

6. Accuracy and speed

6.1. Accuracy analysis

Coleman et al. [5] used a simulation model of their LNS unit (described in Section IV), to compare the error produced by the 32-bit LNS and FP systems. For LNS, operations were represented as procedures, which could be called from an algorithm's mainline program. Two other versions of each algorithm were written to operate on Pentium 32-bit and 80-bit FP data types. In each trial, the 80-bit

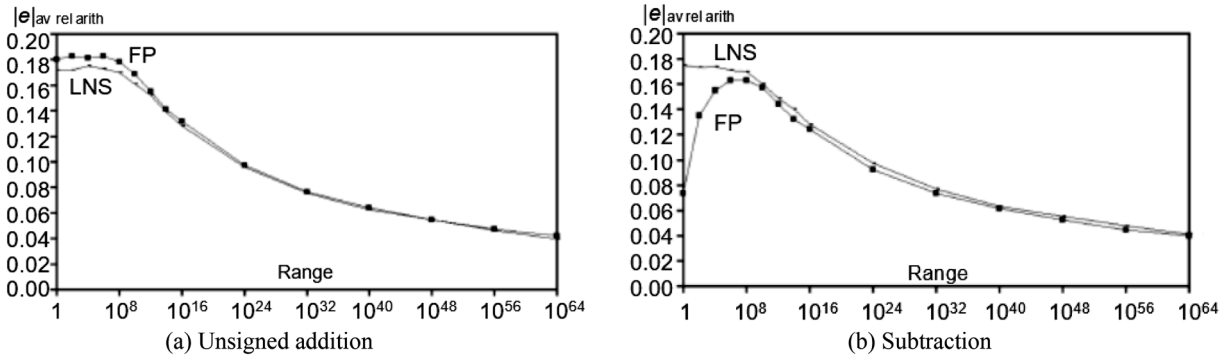


Fig. 6. Error characteristics of LNS compared with FP. Signed addition exhibits variations similar to subtraction.

FP algorithm was regarded as the gold model, yielding the standard result.

The LNS and 32-bit FP value was used as input to the 32-bit implementation under test, while the 80-bit value was supplied to its 80-bit counterpart. In each case, the 80-bit algorithm returned an accurate result and from this, error in the 32-bit system was derived. Values of $|e|_{\text{av rel arith}}$ were calculated for both the 32-bit LNS and 32-bit FP implementations over the entire result file.

We see from Fig. 6 that the objective of designing a logarithmic addition algorithm with substantially the same error as FP has been achieved, the only discrepancy occurring for subtraction with closely matched operands. Addition and multiplication are jointly exercised in MAC and SOP (Fig. 7). In all cases, the LNS error is less than that of FP, particularly where the operands have a wide dynamic range, when the LNS error reduces to between 1/2 and 1/4 of FP errors.

For error analysis on the actual ELM, Coleman et al. developed ELM assembly language programs to sweep through all representable values of the logarithmic domain difference of the two operands in addition or subtraction. Analysis results showed both average and worst-case errors to be comparable for both addition and subtraction operations.

The final hardware implementation of the algorithms differs slightly from that already published [5], leading to marginally better error performance. The algorithms used for simulation, when implemented in hardware as well on the ELM evaluation board, yield virtually identical results. Evidently, LNS has accuracy advantages over FP. In the examples discussed above, it produced, on average, 65 percent of the FP error (error reduction of about 1/3).

In addition to worst-case error analysis for various operations, it is important to study the distribution of errors, both in application-independent studies and in application-specific contexts. As in any number representation system, there is an accuracy-performance trade-off in choosing pertinent LNS parameters and hardware implementations. Use of wider logarithms leads to improved accuracy, but it increases the LNS adder/subtractor complexity almost exponentially.

6.2. Speed analysis

ELM was fabricated in 0.18 μm technology and all of the measurements reported by Coleman et al. [6] were taken from the system running at 125 MHz. As the FP comparison basis, they chose the Texas Instruments TMS320C6711 DSP chip, which is very similar to ELM evaluation board in every respect apart from its arithmetic system. It runs at 150 MHz, the fastest speed grade available to them in 0.18 μm technology. TMS is a superscalar device with a VLIW organization, an RR instruction set, two independent arithmetic units, each with 16 registers, parallel pipelined FP add and multiply units, and integer and reciprocal-approximation units, and has an optimizing C compiler. ELM on the other hand, is a scalar device with an RM instruction set and 16 general registers. It has an interlocked pipeline, and is programmed in assembly (a source of difficulty in comparisons). The resources available to each device are detailed in Table 2.

The devices seem to be comparable in terms of fabrication technology and clock speed. They are also broadly comparable in terms of memory bandwidth, but do have different cache arrangements. TMS has a unified L2 cache, which is not present on the ELM. There are also differences in the L1 caches, which are to some extent direct consequences of the different arithmetic techniques. To test the performance difference between the two, various arithmetic and logical operations were run on both and the time taken was noted. Latencies and vector throughputs are summarized in Table 3.

Measuring by time and recognizing the slightly slower clock speed of the ELM implementation, additions and subtractions are still marginally faster than on TMS, except when the range shifter is deployed. Multiplications are 3.4 times as fast, while divisions and square-roots complete in a small fraction of the time as compared to TMS. In vector sequences, TMS is at an advantage. However, it attains this level of performance only when a continuous pipeline flow is maintained. This will be difficult in code with data dependencies or with short vectors. ELM has a lower latency than TMS in almost all cases, leading to a significantly higher scalar throughput. Overall, ELM can be expected to offer around twice the performance of TMS, with the advantage increasing for computations involving a significant proportion of division and/or square-root operations.

7. Other performance criteria

Power consumption has become a major design criterion all through the performance scale. For instance, cooling of supercomputers and extending battery life in personal electronics dominate design efforts at the two ends of the performance spectrum. So, it is natural to inquire about the power consumption implications of using logarithmic arithmetic. It turns out that logarithmic arithmetic is

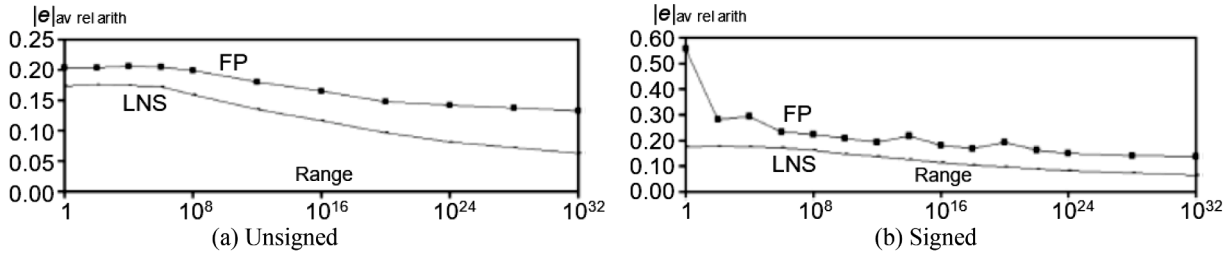


Fig. 7. Error characteristics of multiply-accumulate, $ax + y$. Sum-of-products, $ax + by$, exhibits fairly similar variations.

Table 2
Hardware resources of TMS and ELM devices [6].

System attribute	ELM	TMS
Technology:		
Feature size (μm)	0.18	0.18
Metal layers	6	5
Unit multiplicity:		
Real add/sub	2	2
Real mul/div	4 in all	2 each
Registers	16	32
L1 data cache:		
Size (KB)	8	4
Write policy	W-alloc, WB	W-alloc
Org (B, ways, lines)	32, 2, 128	32, 2, 64
Read BW (GB/s)	2.0	2.4
Write BW (GB/s)	1.0	1.2
L1 instr cache:		
Size (KB)	8	4
Org (B, ways, lines)	32, 2, 128	64, 1, 64
External bus:		
Width (b)	64	32
Protocol	Asynch	Synch
Bandwidth (MB/s)	250	340

Table 3
Latency and throughput of TMS and ELM devices [6].

Operation	Latency (Cycles)	Latency (ns, 125 MHz)	Scalar xput (op/cycle)	Vector xput (op/cycle)
ELM:				
Add	3	24	0.33	0.67
Subtract	3 (4)	24 (32)	0.33 (0.25)	0.67
Multiply	1	8	1	4
Divide	1	8	1	4
Square-root	1	8	1	4
Load	1	8	1	4
Store	2	16	0.5	2
MAC	7	–	0.14	0.5
SOP	7	–	0.14	0.5
TMS:				
Add	4	27	0.25	2
Subtract	4	27	0.25	2
Multiply	4	27	0.25	2
Divide	~30	~200	~0.033	–
Square-root	~40	~267	~0.025	–
Load	4	27	0.25	4
Store	4	27	0.25	2
MAC	16	–	0.0625	0.67
SOP	16	–	0.0625	0.67

by nature low-power.

The energy efficiency advantage of logarithmic number representation was known for quite some time via episodic evidence, as various designers demonstrated low-power circuits for particular applications running on associated fine-tuned hardware platforms. Subsequently, Paliouras and Stouraitis [18] published a study of signal transitions in logarithmic arithmetic versus fixed-point arithmetic, showing 50% or more reduction in activity with linear (uniform and correlated Gaussian) input data. Power savings from reduced activity are augmented by reduction in circuit complexity due to the fact that for LNS, multipliers, dividers, squarers, and square-rooters are replaced by add, subtract, left-shift, and right-shift circuitry, offering a factor of about 2 power savings in arithmetic operations.

For many practical, compute-intensive applications, the savings from reduced activity and circuit simplifications are substantial enough to compensate for the energy requirements of conversion and reconversion processes, needed to interface the system with non-logarithmic data sources and sinks, and the somewhat greater complexity of addition and subtraction. The accrued benefits are, of course, dependent on implementation parameters such as word width, precision (location of the radix point in the logarithm field), and logarithm base.

The overhead for forward and reverse conversions can become insignificant as more computation is carried out between conversions. However, even in the extreme of a single arithmetic operation for each forward and reverse conversion, advantages have been shown to accrue. The extent of advantages will depend on the application and implementation technology. Following Kim et al., who implemented an energy-efficient 32-bit logarithmic arithmetic unit [9], a variety of new applications and associated circuit

realizations have confirmed the energy benefits of logarithmic arithmetic, as suggested earlier [12]. Proposals include both ROM-based designs, perhaps with reconfigurable devices such as FPGAs [7], and several ROM-free implementations [19].

The ultimate in energy efficiency is the use of reversible computation. Quantum computing is inherently reversible, but there are other ways of ensuring reversibility, e.g. via adiabatic design based on CMOS technology.

8. Case studies

As discussed in Section VI, an LNS ALU performs various arithmetic operations faster and with less error than a FP ALU. Also, the results of these discrete operations as performed on hardware were shown, with ELP outperforming a similar commercial pipelined FP device. But for the system to be truly advantageous, the performance gains suggested by these small-scale studies must be shown to also exist in practical, production-scale systems/algorithms.

Examples of LNS-based computation-intensive applications are found in digital filtering, fast Fourier transform, evaluation of elementary functions, and graphics computations. J. N. Coleman et al. demonstrated the LNS advantages using two numeric-intensive application domains as benchmarks: digital signal processing and numerical methods.

The latter experiments, detailed in the following two subsections, were applied to 32-bit LNS and FP designs, with relative accuracy and difference in execution speed as the desired outcomes of the activity. Divisions were used only where essential, and random input data were employed to the extent possible. To calculate total latency, a count of the number of executed real instructions of each type was kept, and this was multiplied with the latency times of each instruction given in Table 1. Thus, we can estimate the speed-up to be gained for each application by using the LNS implementation over the FP implementation.

8.1. Recursive least-squares (DSP)

The recursive least-squares (RLS) algorithm is used in a variety of DSP and communication domains, such as echo and noise cancelation and adaptive control. The objective of this algorithm is to estimate the weights of a finite impulse response (FIR) filter, a third-order instance of which is defined by:

$$y(t) = w_1 \times u(t) + w_2 \times u(t-1) + w_3 \times u(t-2) + e(t) \tag{18}$$

RLS identification uses $y(t)$ and $u(t)$ to estimate the weights w_1, w_2, w_3 by minimizing the sum of squares of prediction errors [20]. It works with square-root-free factorization of the extended covariance matrix:

$$C(t) = L(t) \times D(t) \times L^{'}(t) \tag{19}$$

The simulation procedure used was along the same lines as that discussed in Section IV. The experiment was repeated with a varying dynamic range of the input. The results as conveyed in [20] are plotted in Fig. 8 in terms of signal-to-noise ratio (SNR), showing a significant gain in accuracy for signals of wide dynamic range. Over the narrower portion of the range, LNS outperformed FP by an average of 2.7 dB. For the wider half, this gain was 10.5 dB. Since 6 dB corresponds roughly to 1 bit, this represents a gain in accuracy of 0.5–1.5 bits. The instruction counts and execution times of a run for 1000 input samples on the 32-bit simulator are given in Table 4. The results show a speedup ratio of 2.2 of LNS over the FP implementation.

8.2. Laguerre algorithm (Numerical methods)

This is the second application discussed by Coleman et al. [20] for comparative performance evaluation of FP versus LNS. This algorithm, taken directly from published numerical recipes, computes the roots of a polynomial. The algorithm requires square-rooting. In the FP version, this was evaluated using an initial approximation from a 4 K word lookup table (covering the entire FP range), followed by two Newton-Raphson iterations.

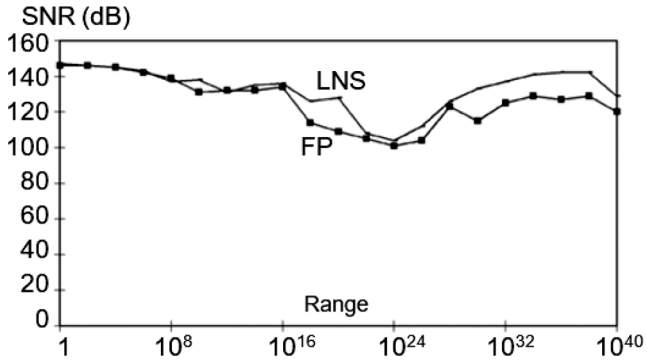


Fig. 8. Simulation results for RLS [20].

Table 4
Operation counts and run times for RLS [20].

Operation	FP	LNS
Add	16,000	17,389
Subtract	15,000	13,611
Multiply	38,000	38,000
Divide	13,000	13,000
Square-root	0	0
Time (ms)	**2.56**	**1.17**

The simulation procedure was the same as discussed earlier. A randomly generated array of signed coefficient data was used and the experiment was repeated for varying dynamic range and for varying n. Each result was averaged over 1000 trials. As the order n was increased, the range had to be reduced. In all cases, LNS error was between 0.74 and 0.79 that of FP.

A similar consistency was observed with increasing n, as shown in Fig. 9, from which it is evident that the LNS implementation induced an average error of 73 percent that of FP. At $n = 10$, for example, the error is reduced from about 80 to 50 *ulp* (units in least position). This improvement is equivalent to about 0.5 bit of added precision. Total execution times for the 1000 trials at $n = 10$ are shown in Table 5. The LNS execution time is 0.41 of that of FP, a speed-up ratio of 2.5.

9. Low-Precision applications

As noted in the introduction, the idea of logarithmic number systems came about from arithmetic-intensive DSP applications that required fairly low precision, thus making table-based realization of addition/subtraction feasible at a time when memory was fairly expensive. Thus far, we have focused on LNS as a replacement for traditional fixed- and floating-point arithmetic needed in general-purpose applications. In certain special-purpose domains, low-precision arithmetic may be acceptable, particularly if it comes with correspondingly lower circuit cost and energy consumption.

Low-precision application domains are expanding as computers increasingly handle multimedia and signal processing workloads instead of conventional numerical computations. Progress in this domain entails the identification and deployment of novel applications, given that techniques for low-precision LNS arithmetic are already highly developed.

Let us speculate on a number of application domains, outside traditional DSP, that may benefit from low-precision LNS arithmetic. Approximate computing, sometimes also referred to as inexact design [21] is an emerging area of investigation that deals with trading off precision for improved speed and power consumption. This is closely related to precision-on-demand methods, where data-path widths are optimized according to the criticality of a computation segment to the accuracy of the final result [22]. For example, there is really no need for every computation segment to be performed with uniform precision (same bit-width) if the less-significant bits will disappear subsequently because of a number being multiplied by a small factor. It is noteworthy that even with uniform data-path width, certain circuit optimizations, such as the use of truncated multipliers with significantly lower cost and power consumption can improve the cost-effectiveness of application-specific systems.

Another example is provided by the field of cognitive computing, which seeks to develop a unified computational theory of the mind. Current, near-real-time cortical simulators at the mammalian scale require the use of the most powerful supercomputers. Perhaps the higher efficiency and lower power of LNS can be used to develop special-purpose simulators at much lower cost and complexity. Neural networks provide related application domain where accuracy requirements may not be very stringent.

When dealing with low-precision computations, accuracy requirements for forward and reverse conversions can also be relaxed to save on circuit cost and energy consumption. Simple, fast, and low-precision approximations of the logarithm and exponential functions can be tapped for implementing these conversions, with a standby precision-improvement mechanism provided for rare

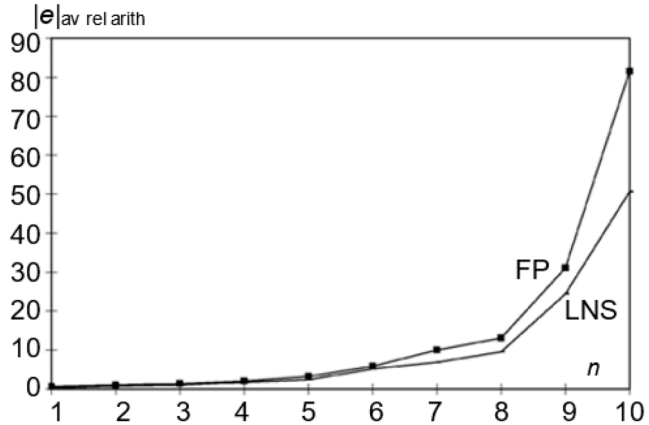


Fig. 9. Simulation results for Laguerre [20].

Table 5
Operation counts and run times of the Laguerre algorithm [20].

Operation	FP	LNS
Add	2,786,839	2,480,039
Subtract	2,868,063	1,664,927
Multiply	5,558,119	4,048,183
Divide	1,218,801	463,833
Square-root	0	377,217
Time (ms)	**361**	**147**

cases when greater precision might be needed.

Finally, low-precision logarithmic arithmetic can be used in conjunction with high-precision representations for low-cost error checking in critical computations, in much the same way that residue channels are used [23].

10. Discrete logarithmic representation

For completeness, we present a brief review of the discrete counterpart [24] to logarithmic representation, even though its applications are rather limited. Given the set $S = \{0, 1, 2, \ldots, p-1\}$, where p is a prime number, modulo-p exponentiation of element of S can be performed either by using standard exponentiation followed by reduction of the final integer modulo p or else via modular multiplications. For example, to compute $3^8 \bmod 17$, we either compute $3^8 = 6561$ and then find $6561 \bmod 17 = 16$, or we perform three modular multiplications, as follows: $3 \times 3 \bmod 17 = 9$; $3^4 \bmod 17 = 9 \times 9 \bmod 17 = 13$, and finally $3^8 \bmod 17 = 13 \times 13 \bmod 17 = 16$. When the exponent is not a power of 2, modular multiply-add operations can be used to compute the exponentiation result via Horner's rule.

Just as ordinary logarithms are inverse functions of exponentiation, discrete logarithms are inverse functions of discrete exponentiation. Taking the example in the preceding paragraph, we can say that 16 is the mod-17 discrete logarithm of 8 in base 3, that is, $3^8 = 16 \bmod 17$ implies that $\text{dlog}_3 16 = 8 \bmod 17$. The solution, however, is not unique, given that the equality $3^{16} \bmod 17 = 1$ leads to any number of the form $8 + 16k$ also being a solution. So, the range must be appropriately limited.

Similar to representations with real-valued logarithms, multiplication, division, powering, and root extraction become simple operations with discrete logarithmic representations, whereas addition and subtraction are complicated operations that must be implemented with tabular assist. Conversion to/from binary format are also shown to be feasible for tabular implementation, requiring tables of size and width $O(k)$.

Extension of LNS to the use of more than one base is an interesting possibility [25]. Fit-Florea et al. [15] use N. F. Benschop's patented result, that any k-bit integer x can be represented by an exponent triple (s, p, e) satisfying $x = (-1)^s 2^p 3^e \bmod 2^k$, to devise a 2D discrete logarithmic number representation system. The representation has the standard sign bit s and two integer exponents p and e. Multiplication is then converted to two additions, while add/subtract is simplified because of independent operations on base-2 and base-3 values. Further extension to 3 or more bases is also possible.

One benefit of discrete logarithmic representation is that it allows exact (integer) arithmetic, which is useful in certain applications such as cryptography. Fit-Florea et al. [15] demonstrate the advantages of the scheme for a powering computation, involving input and output in standard binary format and discrete logarithmic representation internally. Despite the conversion and reconversion overheads, the overall scheme performs the exponentiation with $O(k)$ additions and no multiplications for a k-bit operand, leading to area and power-dissipation advantages over the fastest standard method, with the savings being comparable to those achieved by LNS over floating-point.

11. Conclusion and future work

We have presented a detailed overview of logarithmic number representation, recent advances that have made the scheme even more competitive, and advantages in some application domains. The main advantage of LNS is that multiplication/division is achieved by simply adding/subtracting the logarithms. However, this advantage may be offset by the fact that adding two numbers, a nonlinear operation in LNS, is complicated. LNS disadvantages in performing addition/subtraction are mitigated when low-precision arithmetic is sufficient for application needs, as detailed in Section IX. The presence of many papers pertaining to the design of low-cost, low-power LNS computations tailored to the needs of machine-learning and neural-networks applications in the publication queues of multiple IEEE journals is thus not surprising [26-28].

In numerically intensive applications, the operations of multiplication, division, exponentiation, and root extraction are frequent enough to give LNS an overall edge in performance and power consumption, even when, owing to area and cost constraints, addition and subtraction are not realized with the fastest known methods. Besides simple table lookup, techniques/architectures for interpolation of data using expansion series have been investigated, and ways to minimize the error have been devised. It appears that with suitable approximation and interpolation techniques, additions and subtractions can be performed accurately and rapidly, and are thereby competitive with their FP counterparts.

Much of the discussion in this paper was in terms of general-purpose computation of the kind typically run on commodity microprocessors. In fact, LNS was originally developed for applications in signal processing and, as of this writing, is still better suited to that domain and other application areas that are similarly intensive in multiplications, divisions, exponentiation, and root-extraction.

It is thus worth stressing that performance gains, hardware simplification, and energy efficiency advantages of LNS, discussed in the preceding pages, carry over directly to any custom hardware realization of application-specific systems. Furthermore, modern signal processing tasks require a great deal of scheduling and housekeeping operations that need more than just number-crunching and can thus benefit from general-purpose hardware, perhaps with special-purpose hardware assist.

In closing, as alluded to in Section 1, our ability to deal with algorithmic complexity is limited by instruction execution times (modern memory and I/O bottlenecks must also be tackled, independent of arithmetic design and number representation scheme). Results of various studies compiled in this paper suggest that by providing additional options and design points, logarithmic arithmetic might offer valuable means for dealing with this limitation in future digital systems. Various combinations of logarithmic and other representation strategies may play a role in overcoming some of the current obstacles. Any advantage will be amplified when some data is already sensed or otherwise made available in logarithmic form [29].

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The author thanks Mr. Manik Chugh for his help with the preparation of an early version of this paper [8]. The review herein is extensive but not exhaustive. An extended version of this paper, with ~100 references (that put it beyond this journal's limit on the number of reference citations), is available on-line [30]. Even the extended version does not include all the papers published over the five decades of research covered. Besides published papers that may have been missed, there are quite a few patents on LNS and its applications. A February 2020 Google Scholar search for "logarithmic arithmetic" yielded about 15,000 more hits when patents were also included among the results.

References

[1] Parhami B. Computer arithmetic: algorithms and hardware designs. 2nd ed. Oxford; 2010. 1st ed., 2000.
[2] Benschop NF. Associative digital network theory: an associative algebra approach to logic, arithmetic and state machines. Springer; 2009 (Chapter 11: "Log-Arithmetic, with Single and Dual Base").
[3] Taylor FJ. An Extended Precision Logarithmic Number System. IEEE Trans. Acoustics, Speech, and Signal Processing 1983;31:232–4.
[4] Lewis DM. An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System. IEEE Trans. Computers 1990;39:1326–36.
[5] Coleman JN, Chester EI, Softley C, Kadlec J. Arithmetic on the European Logarithmic Microprocessor. IEEE Trans. Computers 2000;49:702–15. erratum, Vol. 49, p. 1152, 2000.
[6] Coleman JN, Softley CI, Kadlec J, Matousek R, Tichy M, Pohl Z, Hermanek A, Benschop NF. The European Logarithmic Microprocessor. IEEE Trans. Computers 2008;57:532–46.
[7] Fu H, Mencer O, Luk W. Comparing Floating-Point and Logarithmic Number Representations for Reconfigurable Acceleration. Proc. IEEE Conf. Field-Programmable Technology June 2010:337–40.
[8] Chugh M, Parhami B. Logarithmic Arithmetic as an Alternative to Floating-Point: a Review. Proc. 47th Asilomar Conf. Signals, Systems, and Computers November 2013. to appear.
[9] Kim H, Nam BG, Sohn JH, Woo JH, Yoo HJ. A 231-MHz, 2.18-mW 32-bit Logarithmic Arithmetic Unit for Fixed-Point 3-D Graphics System. IEEE J. Solid-State Circuits November 2006;41(11):2373–81.
[10] Nandan D, Kanungo J, Mahajan A. An Efficient VLSI Architecture Design for Logarithmic Multiplication by Using the Improved Operand Decomposition. Integration 2017;58:134–41.
[11] Liu C-W, Ou S-H, Chang K-C, Lin T-C, Chen S-K. A Low-Error, Cost-Efficient Design Procedure for Evaluating Logarithms to Be Used in a Logarithmic Arithmetic Processor. IEEE Trans. Computers April 2016;65(4):1158–64.
[12] Lastras M, Parhami B. A Logarithmic Approach to Energy-Efficient GPU Arithmetic for Mobile Devices. Proc. 47th Asilomar Conf. Signals, Systems, and Computers November 2013. to appear.
[13] Paul S, Jayakumar N, Khatri SP. A Fast Hardware Approach for Approximate, Efficient Logarithm and Antilogarithm Computations. IEEE Trans. VLSI Systems February 2009;17(2):269–77.
[14] De Caro D, Genovese M, Napoli E, Petra N, Strollo AGM. Accurate Fixed-Point Logarithmic Converter. IEEE Trans. Circuits and Systems II July 2014;61(7):526–30.
[15] Fit-Florea A, Li L, Thornton MA, Matula DW. A Discrete Logarithm Number System for Integer Arithmetic Modulo 2^k: algorithms and Lookup Structures. IEEE Trans. Computers February 2009;58(2):163–74.
[16] Parhami B. The Return of Table-Based Computing. Proc. 52nd Asilomar Conf. Signals, Systems, and Computers 2018:115–9.
[17] Coleman JN, Chester EI. A 32-Bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point. Proc. 14th IEEE Symp. Computer Arithmetic 1999:142–51.
[18] Paliouras V, Stouraitis T. Low-Power Properties of the Logarithmic Number System. Proc. 15th IEEE Symp. Computer Arithmetic 2001:229–36.
[19] Ismail RC, Coleman JN. ROM-less LNS. Proc. 20th IEEE Symp. Computer Arithmetic 2011:43–51.
[20] Coleman JN, Softley CI, Kadlec J, Matousek R, Licko M, Pohl Z, Hermanek A. Performance of the European Logarithmic Microprocessor. Proc. SPIE Annual Meeting 2003:607–17.
[21] Anthes G. Inexact Design: beyond Fault-Tolerance. Commun ACM April 2013;56(4):18–20.
[22] Lee DU, Villasenor JD. A Bit-Width Optimization Methodology for Polynomial-Based Function Evaluation. IEEE Trans. Computers April 2007;56(4):567–71.
[23] Massey JL. Survey of Residue Coding for Arithmetic Errors. ICC Bulletin October 1964;3(4):4–17.
[24] Schirokauer O, Weber O, Denny T. Discrete logarithms: the effectiveness of the index calculus method. Algorithmic Number Theory 1996;1122:337–61. Lecture Notes in Computer Science.
[25] Azarmehr M, Ahmadi M, Jullien GA, Muscedere R. High-Speed and Low-Power Reconfigurable Architectures of 2-Digit Two-Dimensional Logarithmic Number System-Based Recursive Multipliers. IET Circuits, Devices & Systems 2010;4(5):374–81.
[26] M.S. Ansari, B.F. Cockburn, and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," IEEE Trans. Computers, to appear.

[27] M. Loukrakpam and M. Choudhury, "Error-Aware Design Procedure to Implement Hardware-Efficient Logarithmic Circuits," IEEE Trans. Circuits and Systems II, to appear.
[28] P. Yin, C. Wang, H. Waris, W. Liu, Y. Han, and F. Lombardi, "Design and Analysis of Energy-Efficient Dynamic Range Approximate Logarithmic Multipliers for Machine Learning," IEEE Trans. Sustainable Computing, to appear.
[29] Kavadias S, Dierickx B, Scheffer D, Alaerts A, Uwaerts D, Bogaerts J. A Logarithmic Response CMOS Image Sensor with on-Chip Calibration. IEEE J. Solid-State Circuits August 2000;35(8):1146–52.
[30] B. Parhami, "Computing with Logarithmic Number System Arithmetic (Extended Online Version with More Reference Citations)," August 2020. https://web.ece.ucsb.edu/~parhami/pubs_folder/parh20-caee-comput-w-lns-arith.pdf.