# A 32-Bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point

J.N. Coleman and E.I. Chester
Department of Electrical & Electronic Engineering, The University, Newcastle upon Tyne, NE1 7RU, United Kingdom.
j.n.coleman@ncl.ac.uk

***Abstract***

*As an alternative to floating-point, several papers have proposed the use of a logarithmic number system, in which a real number is represented as a fixed-point logarithm. Multiplication and division therefore proceed in minimal time with no rounding error. However, the system can only offer an overall advantage if addition and subtraction can be performed with speed and accuracy at least equal to that of floating-point, but these operations require the interpolation of a non-linear function which has hitherto been either time-consuming or inaccurate. We present a procedure by which additions and subtractions can be performed rapidly and accurately, and show that these operations are thereby competitive with their floating-point equivalents. We then show that the average performance of the logarithmic system exceeds floating-point, in terms of both speed and accuracy.*

## 1. Introduction

As an alternative to floating-point, several papers have proposed the use of a logarithmic number system for the representation of real numbers. In a LNS a number $x$ is represented as the fixed-point value $i = \log_2 x$, with a special arrangement to indicate zero $x$ and an additional bit to show its sign. For $i = \log_2 x$ and $j = \log_2 y$, and assuming without loss of generality that $j \le i$, LNS arithmetic involves the computations

$$\log_2(x + y) = i + \log_2(1 + 2^{j-i}) \quad (1)$$

$$\log_2(x - y) = i + \log_2(1 - 2^{j-i}) \quad (2)$$

$$\log_2(x * y) = i + j$$

$$\log_2(x / y) = i - j$$

By suitable choice of position for the binary point, a LNS can have similar range and precision to a FLP system of equal wordlength. The most valuable advantage of the LNS is that multiplication and division can be performed much more rapidly than in FLP, as they require only one fixed-point addition time. These operations are also infinitely more accurate, returning an exact result where FLP has potentially a half-bit rounding error. If LNS addition and subtraction could be performed with speed and accuracy at least equivalent to FLP, then the average performance of the LNS would significantly exceed that of FLP, but unfortunately these operations require the evaluation of a non-linear function $F = \log_2(1 \pm 2^{j-i})$, illustrated in fig. 1.1. For practical purposes, this can only be held in a lookup table, and for practical wordlengths it is not possible to store the entire range of values, so an interpolation must be performed. The function is irrational and thereby subject to a half-bit rounding error, the interpolation procedure tends to introduce additional error, and the entire process is time-consuming. Designs for LNS addition operations have therefore so far tended to be slower or less accurate than FLP, or to be restricted to short wordlengths. No designs to date appear to offer both the speed and the accuracy of FLP at 32-bits precision.

In this work we present such a design. Section 2 describes two original algorithms for the evaluation of addition and subtraction with equivalent accuracy to FLP. It is shown in section 3 that a VLSI implementation of these algorithms will have speed approximately equal to that of an equivalent FLP unit. As the LNS device also maintains the intrinsic advantages of LNS multiplication and division in terms of both speed and accuracy, the average performance of this unit is considerably better than that of a 32-bit FLP unit. The results of some systematic larger-scale simulations designed to compare the accuracy of the two systems are given in section 4.

**Fig. 1.1. LNS add and subtract functions**

## 2. Arithmetic algorithms

J.N. Coleman

The use of 16-bit LNS arithmetic was originally proposed as an alternative to 16-bit fixed-point by Kingsbury and Rayner in 1971 [1]. Simulated LNS

arithmetic demonstrated that the greater dynamic range of this system yielded a very significant improvement in the response of a digital filter. Attention turned some years later to the better roundoff-error characteristics of the LNS when compared to FLP, and clear improvements in noise-to-signal ratio were demonstrated in a filter [2], and FFT [3].

Implementation work began with a 1975 paper which suggested a 12-bit device [4], whilst a 1988 proposal extended this to 20-bits [5]. Both designs were direct implementations of eqns 1 and 2, with a lookup table covering all possible values (omitting those which quantise to zero) of $F(r = j - i)$. It is of course apparent that as the wordlength increases, the table sizes increase exponentially, which limits the practical utility of this approach to about 20 bits. A 1991 design [6] extended the wordlength to 28 bits by implementing the lookup table throughout $r$ only at intervals of $\Delta$, where $r = -n\Delta - \delta$. Intervening values were obtained by interpolation using the Taylor series

$$F(r) = F(-n\Delta) + [D(-n\Delta) / 1!]\,\delta + [D'(-n\Delta) / 2!]\,\delta^2 + \ldots$$

This design used only the first-order term, requiring the additional storage of a table of derivatives $D(r)$, and introducing a multiplication into the critical speed path. The scheme also exposed a further problem intrinsic to LNS arithmetic: the difficulty of interpolating $F(r)$ for subtractions in the region $-1 < r < 0$. Here, as is evident from fig. 1.1, $D(r) \rightarrow -\infty$. To maintain accuracy in the face of the rapidly changing derivative, it is necessary to implement a large number of successively smaller intervals as $r \rightarrow 0$. The large memory with its irregular organisation is cumbersome to design, and complicates the VLSI floor-planning. Although the accuracy achieved by [6] was not reported, it would have been limited by the rounding error in the function, plus any additional error introduced by the interpolation procedure.

A separate proposal in 1994 [7] involved the use of a higher-order polynomial interpolator, with a novel scheme for interleaving the stored values so as to reduce the overall memory requirement. A design for a 32-bit unit with a 287 Kbit lookup table was presented, and its addition accuracy shown by simulation to be within FLP limits. A variant had a smaller table with larger errors. The critical speed path included a ROM, two multipliers, three barrel-shifters, and five carry-propagate adders / subtractors. Fabrication of the lower-accuracy variant was done in 1995 [8], with a latency of 158 ns in add / subtract operations, and 13 ns for multiply / divide.

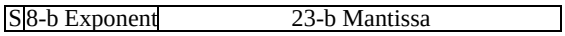
Some novel arithmetic transforms which may have the potential to simplify the addition and subtraction algebra are given in [9]. This work also includes a review of recent practical applications of LNS arithmetic from the U.S.A. and Japan. Areas cited include image processing and DSP applications, graphics, and aircraft controls. Fujitsu are making some use of the technique in their microprocessors.

The aim of the current work is to develop a 32-bit adder / subtractor with speed and accuracy comparable to that of a FLP system. We took as our starting-point a first-order Taylor-series approximation, the critical delay path of which comprises a ROM, a multiplier, and two carry-propagate adders, which is at the limit of what can be implemented without significantly exceeding the delay of a FLP addition. We therefore faced two problems. First, we had the difficulty, referred to above, of interpolating the subtraction function in the region $-1 < r < 0$. In 1995 we published an algorithm [10] for transforming such a subtraction into one for which $r < -1$, with little extra time delay. The second problem is that, at 32-bit wordlengths, a first-order Taylor interpolation yields far too high an error to meet the desired accuracy without recourse to unreasonably large lookup table sizes. A natural way to improve the accuracy would be to evaluate the second order term, but unfortunately this would considerably extend the delay path. An alternative technique for function evaluation without interpolation uses bipartite tables [11]. With this method, it is possible to replace a table access with $n$ address bits with two simultaneous accesses each with approximately 2/3 $n$ address bits. For $n = 28$ bits as required here, however, even this would yield an unacceptably large table size.

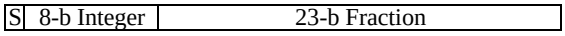
Using a crude first-order approximation procedure, we now describe a new algorithm by which the error it yields can be simultaneously evaluated, then accumulated into the result, thereby correcting the error with a delay of only one extra carry-save stage. We will show that a unit designed to implement this algorithm will perform 32-bit addition and subtraction operations with speed and accuracy roughly equal to that of a FLP unit.

## 2.1. Data format, range and precision

IEEE standard FLP representation uses a sign, 8-bit biased exponent, and 23-bit significand. The latter has an implied binary point immediately to its left, and a hidden '1' to the left of the point. Extreme exponent values (0 and 255) are used to represent special cases, thus this format holds values in the range $\pm(1.0 * 2^{-126})$ to $(1.111... * 2^{+127})$, ≈ ±1.2E–38 to 3.4E+38.




In the equivalent LNS representation used throughout this work, the integer and fractional parts form a coherent two's complement fixed-point value in the range ≈ –128 to +128. The real numbers represented are in the range $\pm 2^{-128}$ to $2^{+128}$, ≈ ±2.9E–39 to 3.4E+38. One value, all ones, is used to represent the real number zero.




## 2.2. Measurement of accuracy

**Definitions** In a FLP implementation with $f$-bit significand, assuming that both significands represent exact values, a result $\hat{A}$ as returned by a practical implementation can be regarded as an approximation to the corresponding exact result $A$. Each result is thus in error by $e = \hat{A} - A$. The errors throughout the range of significands are generally quoted relative to the weight of the LSB.

$$e_{\text{max rel}} = \frac{\max(\hat{A} - A)}{2^{-f}};$$

similarly $e_{\text{min rel}}$, $|e|_{\text{max rel}}$, $e_{\text{av rel}}$, $|e|_{\text{av rel}}$.

In the equivalent LNS (i.e. with $f$-bit fractional part), it

may be correspondingly assumed that both logarithms forming the inputs to an operation are exact. Each result $\hat{I}$ returned by the implementation is an approximation to the corresponding exact logarithm $I$, and is thereby in error by $e_{\log} = \hat{I} - I$. Thus

$$e_{\text{max rel log}} = \frac{\max(\hat{I} - I)}{2^{-f}};$$

similarly $e_{\text{min rel log}}$, $|e|_{\text{max rel log}}$, $e_{\text{av rel log}}$, $|e|_{\text{av rel log}}$.

Since, however, the user is concerned not with the logarithm *per se*, but rather with the value that it represents, it is also possible to quote this error in a way which allows direct comparison with the error returned by FLP. The error in the value represented is then expressed in terms of the weight of the LSB of an equivalent FLP operation, i.e.

$$e'_{\text{max rel}} = \frac{2^{\max(\hat{I} - I)} - 1}{2^{-f}};$$

similarly $e'_{\text{min rel}}$, $|e'|_{\text{max rel}}$, $e'_{\text{av rel}}$, $|e'|_{\text{av rel}}$.

For $f = 23$, FLP arithmetic has $e_{\text{max rel}} = 0.5$, $e_{\text{min rel}} = -0.5$, $e_{\text{av rel}} = 0$, and $|e|_{\text{av rel}} = 0.1733$, whilst logarithmic addition and subtraction has best-case values of $|e|_{\text{max rel log}} = 0.5$, $|e|_{\text{av rel log}} = 0.25$, $e'_{\text{max rel}} = 0.3464$, $e'_{\text{min rel}} = -0.3464$, $e'_{\text{av rel}} = 0$, and $|e'|_{\text{av rel}} = 0.1733$. In LNS multiplication and division all values are 0.

**Theorem 1** *If the operations defined in eqns 1 and 2 yield errors within a given $e_{\text{max rel log}}$ over all negative values of j for i = 0, then they yield the same $e_{\text{max rel log}}$ over all values of j for all values of i. An implementation can thus be regarded as fully verified if it can be verified over this subset.*

**Proof** Let $r = j - i$. Then $I = i + F(r)$ and $\hat{I} = i + \hat{F}(r)$. Thus

$$e_{\text{max rel log}} = \frac{\max\left[\hat{F}(r) - F(r)\right]}{2^{-f}}$$

Let $i$ and $j \in \{\text{minlog} .. 0 .. \text{maxlog}\}$. By definition $j \le i$, $\therefore\ r \in \{\text{minlog} .. 0\}$. Then for $\{(i = 0, j = \text{minlog}) .. (i = 0, j = 0)\}$, also $r \in \{\text{minlog} .. 0\}$.

## 2.3. Conventional LNS addition and subtraction

The function $F(r)$ was shown in fig. 1.1. To minimise its storage, $\Delta$ is progressively increased as the function becomes more linear with decreasing $r$, and an intervening value of $r$ lying in the $n$th interval is thus correctly expressed as

$$r = -\delta \ \{n = 0\};\ r = \sum_{0}^{n-1}(-\Delta_n) - \delta \ \{n > 0\}$$

In a typical system, the range of $r$ will be partitioned at each power of 2, and $F(r)$ implemented with a small table of equal size in each segment; i.e. $\Delta$ will be doubled at each increasing power of 2. For clarity of explanation, however, the following text will omit further reference to the variation in $\Delta$, and abbreviate this expression to

$$r = -n\Delta - \delta$$

In the programmed simulations the correct treatment of $\Delta$ will nevertheless be maintained, so that their results may be regarded as realistic of a practical implementation.

Together with each value of $F(r)$ is stored its derivative $D(r)$. The function of the intervening value of $r$ is then obtained by interpolation

$$F(-n\Delta - \delta) \approx F(-n\Delta) - \delta D(-n\Delta)$$

A conventional implementation is shown outside the dotted lines in fig. 2.1. Following the initial subtraction to obtain $r$, the latter is partitioned, effectively dividing by $\Delta$. The high-order bits represent $n$, and are used to access the F and D tables, whilst the low-order bits represent $\delta$. $F(-n\Delta)$ is then added to the product $\delta D(-n\Delta)$ to obtain the approximation to $F(r)$, which is added to $i$ to yield the result.

**Fig. 2.1. LNS implementation (new components within dotted lines)**

The interpolation yields an error, as shown in the inset to fig. 1.1

$$\varepsilon(n, \delta) = F(-n\Delta) - \delta D(-n\Delta) - F(-n\Delta - \delta)$$

For each $n$, $\varepsilon$ increases with $\delta$ to a maximum

$$E(n) \approx F(-n\Delta) - \Delta D(-n\Delta) - F(-n\Delta - \Delta)$$

An example of the errors arising during additions using this interpolation method is shown in the upper curve of fig. 2.2. These simulations were based on a 40-bit LNS, with 8-bit integer part and 32-bit fraction. The non-zero range of $r$ was partitioned as described above, into 7 segments covering the sub-ranges 0..–1, –1..–2, ... –32..–64, each segment being divided into a fixed number of intervals. Additions were simulated for all values of $j$ for $i = 0$, and the results compared with accurate values obtained from 80-bit FLP arithmetic. The graph shows the maximum value, anywhere throughout the range of $r$, of $|\varepsilon|$ in terms of the LSB, i.e. $|e|_{\text{max rel log}}$. From theorem 1, these results are applicable to all combinations of operands. The simulation was repeated, varying the number of intervals in each segment from 128 to 4096. For 128-interval segments, $\Delta$ is $2^{25}$ within the first segment at the RHS of $F(r)$, and doubles at each power of 2. It is evident that the maximum error reduces

by a factor of 4 with each doubling of the table size; this because each time $\Delta$ is halved, the accuracy of both $F(r)$ and $D(r)$ improves by one bit.

**Fig. 2.2. Error in 40-b add operations**

**Fig. 2.3. $\alpha$ and $\varepsilon$ vs $n$ and $\delta$**

**Fig. 2.4. Storage requirement**

## 2.4. Error correction algorithm

In this paper we present a means to correct the error $\varepsilon$. It is based on the observation that, for a given $\delta$, the ratio $P = \varepsilon(n, \delta) / E(n)$ is roughly constant for all $n$. It is therefore possible to store for one $n$ a table P of the error at successive points throughout $\Delta$, 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, its value of $E$. The error $\varepsilon$ is then obtained for any $(n, \delta)$ as

$\varepsilon(n, \delta) = E(n)P(\delta)$

This is added to the result of the interpolation, thereby correcting the error.

This scheme has the major practical advantage that the lookups of $E(n)$ and $P(\delta)$ can be performed at the same time as those of $F(n)$ and $D(n)$, and their product can be evaluated simultaneously with the multiplication in the interpolation. Since the interpolation already requires a final addition, this can be combined with the addition of $\varepsilon$ by using a carry-save stage. The entire correction procedure can therefore be performed with only a few extra gate delays. The extra hardware is shown within the dotted lines in fig. 2.1.

The error $e_{\log}$ in the final corrected result (assuming $P$ calculated at $n = 0$) is

$e_{\log} = E(n)P(0, \delta) - E(n)P(n, \delta)$, which can be written

$e_{\log} = \alpha(n, \delta)\varepsilon(n, \delta)$ where

$$\alpha(n, \delta) = \frac{P(0, \delta)}{P(n, \delta)} - 1$$

The value $\alpha(n, \delta)$ represents the factor of the final $e_{\log}$ arising because of the approximation of $P(n, \delta)$ as $P(0, \delta)$. It is difficult to derive an analytical expression for the bounds of $\alpha\varepsilon$, and such an exercise would be of little benefit because a practical implementation will include a further source of error to be described in the next paragraph. However this can be determined numerically, and we shall present such a computational verification below. We first illustrate with an example the behaviour of $\alpha$ and $\varepsilon$. Their maximum magnitudes ($\varepsilon$ is shown relative to the LSB) within individual intervals of $n\Delta = 1$ are plotted against $n$ in fig. 2.3. Plots are given for two values of $\delta$, at a small offset $\gamma = 2^9$ LSB from each extreme of $\Delta$, i.e. $\delta = \gamma$, and $\delta = \Delta - \gamma$. The value of $\Delta$ has been increased at successive powers of 2 throughout $r$ (resulting in discontinuities) with 512-word tables in each segment. The remaining small discontinuities appear because although the terms are fairly evenly distributed across zero, the maximum magnitude is plotted. It is evident that with respect to both controlling variables $n$ and $\delta$, the two sources of error $\alpha$ and $\varepsilon$ are anticorrelated. As $-n\Delta \downarrow \alpha \uparrow$ but $\varepsilon \downarrow$. As $\delta \uparrow \alpha \downarrow$ and $\varepsilon \uparrow$. Thus $e_{\log} = \alpha\varepsilon$ is held at a relatively low value throughout.

A more substantial source of error arises from the fact that for practical values of $\Delta$ it is not feasible to store $P$ for all values of $\delta$. Instead, the P table is implemented at subintervals within $\Delta$, with only the high-order bits of $\delta$ being used to form the address to P. The simulations described in section 2.3, to which theorem 1 applies, were now repeated using the correction algorithm with a P table of varying size. The results are shown in the lower set of curves in fig. 2.2, which indicate that, for small numbers of intervals per segment, the maximum error is halved as the P table is doubled in size. With 512-word F, D and E tables, a P table of 4 Kwords will reduce $|e|_{\max \text{ rel } \log}$ by over 3 orders of magnitude, from 5022 to 3.9, i.e. it will leave two bits in error. However, the algorithm is unable to make an exact correction, and even with a large number of segments per interval, a small error of about 2 still remains.

The new P and E tables increase the overall storage requirement. For the range of values shown in fig. 2.2,

these effects are illustrated in fig. 2.4. For the purpose of this calculation, the F, D and P tables have been assumed to comprise 32-bit words, and the E tables 16-bit words, except in the 128-interval configuration where the larger errors would require a 32-bit E table. For the example illustrated above (512 intervals per segment, P = 4K), the total storage requirement is roughly doubled, from 7168 words (229,376 bits) for the F and D tables only, to 14,848 words (417,792 bits) for the F, D, E and P tables. In practice, these values would be reduced somewhat because the tables would be implemented with only the maximum necessary number of bits per segment, which would gradually reduce through the range of *r*. This would have to be calculated in detail for each individual implementation, after the table sizes had been fixed. The exact sizes of ROM for two possible implementations are given later in this section.

A number of simplifications are possible, which substantially reduce the complexity and delay time of an implementation. In the add / subtract unit, the two multipliers would be realised as parallel arrays. In both cases, both operands have fractional parts, and their results therefore have twice the required number of fractional bits. Rather than rounding the results, the low-order bits may be truncated. This has a negligible effect on accuracy, but does allow more than 50% reduction in the number of multiplier cells. Second, the final carry-propagate stage within both multipliers can be omitted, and separate sum and carry vectors are passed to an enlarged tree of carry-save adders. This saves the majority of the carry-propagate addition time. The simulations in section 2.6 will take these optimisations into account.

Finally, in order to avoid having signed multipliers, the algebra can be rearranged such that the D, E and P tables carry only positive values. The values of *r*, and therefore δ, are similarly always positive.

## 2.5. Subtractor range shifter

From [10], with acknowledgement to the IEE

The range shifter (fig. 2.5) simplifies the subtraction operation by obviating the interpolation in the region $-1 < r < 0$, eliminating table D and reducing substantially the size and complexity of table F, using instead two much smaller and regularly organised tables. At and below $r = -1$, the F and D tables are implemented as before, but in this region are small. It relies on the replacement of subtraction $2^i - 2^j$ with two successive subtractions

$2^i - 2^j = (2^i - 2^{j+k1}) - 2^{j+k2}$, where

$2^{k1} + 2^{k2} = 1$, i.e. $k2 = \log_2(1 - 2^{k1})$

Factor $2^{k1}$ is individually chosen for each combination of operands such that the index *r1* for the inner subtraction falls on the nearest modulo-Δ*1* boundary beneath $j - i$, where Δ*1* is now fixed at a large value. $F(r1)$ can therefore be obtained directly from lookup table F1, which contains $F(r)$ for $-1 < r < -\Delta 1$ at modulo-Δ*1* intervals. Factor *k1* is constrained to lie in the range $-\Delta 1 \le k1 < 0$, and can therefore be used to index another lookup table F2, containing $F(r)$ for all possible values of *r* between $-\Delta 1 < r < 0$, to obtain *k2*. Since $2^{k1} \approx 1$, *k2* is a large negative value. This has the effect of increasing the magnitude of the index for the outer subtraction, *r2*, such that $r2 < -1$. It therefore falls in the linear region of $F(r2)$, and can be obtained by interpolation from the small remaining F and D tables covering this region.

Thus

$r1 = (((j - i) \text{ DIV } \Delta 1) - 1) * \Delta 1 = j + k1 - i$

$k1 = i - j + r1 = -(((j - i) \text{ MOD } \Delta 1) + \Delta 1)$

The subtraction becomes

$2^i - 2^j = 2^{i+F(r1)} - 2^{j+F(k1)}$, which generates an index *r2*

$r2 = j - i + F(k1) - F(r1)$

$= j - i + \log_2[(1 - 2^{i-j+r1}) / (1 - 2^{r1})]$

The value of *r2* can be considered in three regions, depending on the original operands *i* and *j*. For $j - i \le -1$, *r2* is taken directly as $j - i$, and will lie in the linear region of *F* from which $F(r)$ can be obtained by interpolation. For $-1 < j - i < -\Delta 1$, *r2* is derived as shown above, and has a maximum of approximately $-(1 + \Delta 1)$. Thus it also lies in the linear region of *F*, and $F(r)$ is similarly obtained by interpolation. For the third region, $-\Delta 1 \le j - i < 0$, the derived value of *r2* rises above −1. However, this range is covered by the F2 table, and $F(r)$ is therefore already available as *k2*. Appropriate multiplexing paths for the three regions can be arranged either explicitly or by zeroing one of the inputs to an arithmetic element.

The modified values of *j* and *i*, known as *j2* and *i2*, are then subtracted, yielding a new value of *r2* which is guaranteed to fall in the linear region of $F(r)$ below −1. Together with *i2*, this is passed to the adder / subtractor for completion of the outer subtraction.

It is shown in [10] that the combined size of the F1 and F2 tables is about one-seventh of that of the F and D tables that would be required to yield an interpolation of similar accuracy. In the current work, F1 and F2 are 2048 and 4096 words respectively.

**Fig. 2.5. Subtractor range shifter**

A number of simplifications are possible in an implementation of this scheme. In the calculation of *k1* and *r1*, the subtraction $j - i$ is not necessary because this term is already available as *r*. Operation DIV returns a

truncated result, and since $\Delta 1$ is a power of 2 the DIV, MOD and $*$ operations involve only bit-partitioning and concatenation of zeroes. Thus the only arithmetic operations required in these calculations are the addition or subtraction of the single-bit constants $\Delta 1$ and 1. However, it will be noted that there is a deterministic relationship between the bit-partitioned values of $r$ and the functions $k1$ and $r1$ which form the indices to the F1 and F2 tables. The addition and subtraction can therefore be avoided completely by rearranging the mapping of addresses to function values in these tables. Calculation of the modulo $r1$ and coefficient $k1$ can thus be done with no time overhead at all. Finally, the subtraction to obtain $r2$ can be rearranged in order to use the precalculated value of $r$, and to use cumulative additions instead of an addition and a subtraction. The entire unit can thus be implemented with a worst-case delay of one ROM access, a carry-propagate adder, and a carry-save stage.

## 2.6. Adder / subtractor design and evaluation

Our objective was to produce a 32-bit design with 8-bit integer part, 23-bit fraction and sign, having the same or less error than FLP. To accomplish this, we have made a detailed study of two separate designs, one optimised for the maximum possible accuracy, and one for the smallest ROM size. These are referred to as the high and low accuracy designs.

For the high-accuracy variant we have used the 40-bit design described above, taking F, D and E at 512 words and P at 4 Kwords. The incoming 32-bit operands are expanded to 40 bits by concatenating zeroes, and the 40-bit result rounded back to 32 bits with a rounder incorporated into the final adder. Essentially, the 9 extra internal bits are used as guard bits. After rounding, the $|e|_{\text{max rel log}}$ of $\approx 4$ observed in the 40-bit implementation is vastly reduced. Following simulation in accordance with theorem 1, the errors found are given in the upper row of table 2.1.

The same algorithm was also used to correct the error in the subtraction operation. Since interpolation is difficult when $-1 < r < 0$, we have used the algorithm described in [10] to avoid interpolations in this region. It must be noted that when this algorithm is invoked, the subtraction is not done in accordance with eqn 2, since the operands $i$ and $j$ are transformed into new values. As a consequence, theorem 1 does not apply within this restricted range. To have simulated a subtraction for all combinations of $i$ and $j$ yielding $-1 < r < 0$ would have been impractical, so here the operation was simulated for all values of $j$ over a limited subset of $i$, including values around each power of 2 and random values in-between. Results are also given in table 2.1.

The storage requirement for this variant has been calculated by taking the maximum number of bits for the largest value held in each F, D and E table segment individually. The segment sizes therefore reduce with decreasing $r$. This optimised value is thus somewhat less than the estimate presented in fig. 2.4. Total storage is 34,304 words or 856,064 bits, a size that can easily be accommodated in a small area with contemporary fabrication.

In the low-accuracy variant, we have made a number of simplifications to reduce the ROM size, whilst still keeping a maximum error better than FLP. The number of guard bits has been reduced from 9 to 4, which reduces the size of all the tables commensurately. The F, D and E tables are reduced to 256 words per segment, and P tables to 1024 words. At this reduced level of precision, the final segment (−32 .. −64) is not required, so the number of segments is also reduced by one. The accuracy of this variant is presented in the lower row of table 2.1. Its storage requirement is 5632 words (108,032 bits) for addition and 11,088 words (289,280 bits) for subtraction, a total of 34,304 words (397,312 bits). This total is a substantial reduction from that of the high-accuracy variant, and is comparable with that used in [7].

It is the low-accuracy variant which has been chosen for further consideration. **The hardware design and simulations presented in subsequent sections of this paper are all based on this low-accuracy variant.**

| Variant | Operation | $\|e\|_{\text{max rel log}}$ | $\|e\|_{\text{av rel log}}$ | $e'_{\text{max rel}}$ | $e'_{\text{min rel}}$ | $e'_{\text{av rel}}$ | $\|e'\|_{\text{av rel}}$ |
|---|---|---|---|---|---|---|---|
| High acc. | Add | 0.5046 | 0.2509 | 0.3489 | –0.3498 | +0.0066 | 0.1739 |
| | Subtract | 0.5074 | 0.2509 | 0.3517 | –0.3493 | –0.0067 | 0.1739 |
| Low acc. | Add | 0.6556 | 0.2563 | 0.4544 | –0.4233 | +0.0447 | 0.1776 |
| | Subtract | 0.7193 | 0.2563 | 0.4414 | –0.4986 | –0.0456 | 0.1777 |

**Table 2.1. Errors in 32-bit LNS addition and subtraction operations**

## 2.7. Discussion

Apparently we have identified a useful property of the logarithmic addition and subtraction functions. After performing a Taylor interpolation, the shapes of the curves of the errors within each interval remain sufficiently similar throughout the range to permit a good approximation to the curve for any interval to be made by scaling from a template obtained from another interval. This approximation technique has the major practical advantage that the calculation of the interpolation error can be performed at the same time as the interpolation itself. The error can then be corrected by accumulating this term, incurring a delay of only one extra carry-save stage.

Typically this scheme will reduce the maximum interpolation error by several thousand times. It thus yields accuracy equivalent to that which would be obtained by increasing the size of the F and D tables by some 64 - 128 times. However, it does so with only about a twofold increase over the F and D table size. The correction algorithm does not yield an exact result; typically it is inaccurate by a few times the LSB, but this can be overcome at minimal cost by performing the arithmetic to more places than required and rounding the result.

We have illustrated two examples in detail. In the high-accuracy variant, $|e|_{\text{max rel log}}$ for additions in a 40-bit

LNS was reduced by three orders of magnitude by using a 4 Kword P table. This 40-bit system was then used as the basis for a 32-bit implementation, which yielded $|e'|_{\text{max rel}}$ of 0.3517, an increase in accuracy over FLP of more than half a bit. Values of $|e'|_{\text{av rel}}$ were also comparable to FLP, although a small bias did persist which is not present in FLP, seen in the non-zero values of $e'_{\text{av rel}}$. However, this bias is equal and opposite in add and subtract operations, and so is eliminated in signed arithmetic or in code using add and subtract in roughly equal proportions. ROM size for this variant was 856 Kbits.

In the low accuracy variant, the ROM sizes were reduced as far as possible, whilst keeping $|e'|_{\text{max rel}}$ within the FLP limit of 0.5. Once again, there was a more-or-less equal and opposite bias in adds and subtracts, so the system would be unbiased in normal use. The total ROM was now 397 Kbits.

A LNS multiply / divide system returns $|e'|_{\text{max rel}}$ of 0, whereas the corresponding value for FLP is 0.5. A realistic computation will involve both addition and multiplication operations, and the accuracy of such a computation will therefore lie between these extremes, probably approximating to the average.

## 3. Physical design and speed

This is an implementation of the low-accuracy variant defined in section 2. It offers addition, subtraction, multiplication and division operations on LNS format operands. There are two data paths, for add / subtract and multiply / divide, as shown in fig. 3.1.

The two operands are passed directly to three units: magnitude comparison and difference; zero, sign and control; and multiply / divide.

The magnitude comparator returns *i*, *j*, and *r*, and also detects equality. It passes its results regarding the relative magnitude of the two operands to the zero, sign and control unit.

Following the comparator, add / subtract operations are completed in two specialised blocks of circuitry. Additions are passed directly to the add / subtract unit, which interpolates the result using the Taylor expansion with the correction algorithm described in section 2. In the case of subtractions, however, the data will follow one of two paths. If $-1 < r < 0$, the operands will be processed by the range shifter, which will modify their values such that this is no longer the case. After modification, or if $r \leq -1$ initially in which case the range shifter is bypassed, the operands are forwarded to the adder / subtractor where the operation is completed.

The zero, sign and control logic can detect zero operand values, and is supplied with information about the relative magnitude of the operands by the magnitude comparator. In cases involving one or more zero operands, or in cases where the two operands are equal, the result may either follow one of the operands, or be zero itself. The two operands and a 'hot zero' are therefore available to the final multiplexor, the setting of which is determined by the control logic. Meanwhile the sign logic determines the sign of the result in accordance with the normal procedures for sign-and-magnitude arithmetic, and the sign bit is appended as the result leaves the ALU.

**Fig. 3.1. LNS ALU**

### 3.1. VLSI Implementation

A device has been designed to implement the circuits described in this section, taking operands of the format given in section 2.1. It was designed in a 0.7μ two-level-metal standard-cell system, within the CADENCE framework. In order to make the design as realistic as possible, the designer had the leeway to use the system in the most appropriate way, which meant that in some cases he did not follow the circuits exactly as they are presented above. In particular, as in all standard cell systems, better results could sometimes be obtained by synthesising blocks of logic than by designing by hand. Typical delays through the worst-case speed paths were measured with the timing simulator, and the results are given in table 3.1. Two values are shown for subtraction, depending on whether the range shifter is used.

For comparison, a 32-bit FLP unit was designed in the same way, and like the LNS unit was optimised for speed. As far as possible, blocks were re-used from the LNS design. This unit took operands of the format shown in section 2.1, and its results were rounded to the nearest. Since, however, the objective was simply to determine the delay, no attempt was made to refine this design for strict IEEE compliance.

A 32-bit fixed-point unit was also designed. Here, the add / subtract path was almost identical to the LNS multiply / divide unit, and the signed multiplier was produced automatically with the vendor-supplied macrocell generator. Since fixed-point multiplications with fractional parts always require rescaling, a final variable-length shift and round stage was included

following the multiplier. If rounding by truncation were assumed, then the multiplication delay here would be about 4 ns less. We did not design a FLP or FXP divider, but a common rule-of-thumb is that division takes about 3 times as long as multiplication

| | A | S | M | D |
|---|---|---|---|---|
| FLP | 28 | 28 | 22 | |
| FXP | 4 | 4 | 32 | |
| LNS | 28 | 28 / 42 | 4 | 4 |

**Table 3.1. Delay times of VLSI devices (ns)**

### 3.2. Discussion

The times derived for FLP operations are broadly comparable with commercial devices also designed in 0.7μ two-level-metal technology, e.g. the 30MHz TMS320C44 [12]. Including its register and pipeline overhead, this device has add and multiply times of 33ns.

In code comprising a roughly equal proportion of adds and multiplies, it would be expected that the LNS implementation would perform in about 64% of the time taken by FLP. However, a number of assumptions have to be made in the interpretation of these results to allow for practical operating conditions. Examination of large-scale algorithms has suggested that the subtractor range shifter is used, on average, in somewhat less than half of subtractions. Assuming that a roughly equal number of additions and subtractions be executed, this would raise the average add / subtract time to about 31 ns. In the FLP unit the add and multiply times are so close that in any practical device they would be equalised to the larger one in order to allow a constant cycle time. Finally, all these times would be subject to a register and pipeline overhead, and would then be increased if necessary to a multiple of some clock period. These considerations make an exact comparison difficult, but in approximate terms the LNS unit would reach twice the speed of FLP at an add / multiply ratio of about 40% / 60%.

Since the FXP adder is the same as the LNS multiplier, and the FXP multiplier has approximately the same delay as the LNS adder, it may be inferred that in code comprising a roughly equal proportion of adds and multiplies the LNS and FXP implementations would perform in about the same time.

## 4. Simulation of accuracy

J.N. Coleman

The objective was to compare the error produced by the 32-bit FLP system with that from the low-accuracy implementation of the LNS. A version of the simulator used in section 2 was prepared in which operations were represented as procedures, which could be called from a mainline program executing an algorithm. Two further copies of each algorithm were written to operate on the intrinsic (Pentium) 32-bit and 80-bit FLP data types respectively. In each trial, the 80-bit FLP algorithm was regarded as yielding the standard result.

Each algorithm performed the same operation on consecutive items of input data taken from a large, randomly-generated file of 80-bit data. Variates in the file were allowed to range between set limits, and each algorithm was evaluated with variations in this range. The file was generated as follows. A random positive fraction between 0 and 1, was obtained. This represented the value of the variate. A second random positive fraction was then taken, and multiplied by an odd integer constant $p$ representing the number of powers of 10 of the permitted range. This represented the magnitude of the variate; it was used to raise the value to a power of 10 centred at 0. E.g. for $p = 5$, the value would be multiplied at random by any of 0.01, 0.1, 1, 10 or 100. The variate $v$ would thus lie in the range

$$v = (0 \,..\, 1) * 10^{[-(p-1)/2 \,..\, (p-1)/2]}$$

In the results, plots are given for a range of permitted magnitudes, varying from $p = 1$, $v = (0 \,..\, 1) * 1$ at the left, to $p = 65$, $v = (0 \,..\, 1) * 10^{\pm 32}$ at the right. Some of the algorithms were also run in signed form, in which case $v$ was also multiplied randomly by +1 or −1.

Since 32-bit logarithmic and 32-bit FLP values do not quantise identically, these effects were eliminated by preceding each simulation with a quantisation phase. This took the 80-bit random input, quantised it to the nearest 32-bit logarithmic / single FLP value, and converted the latter back to 80-bit FLP representation. The LNS / 32-bit FLP value was used as input to the 32-bit implementation under test, while the 80-bit value was input to its 80-bit counterpart. In many cases, therefore, the two systems were not taking identical input data, but were taking the nearest available values to the original data within their own representation. In each case the 80-bit algorithm returned an accurate result for the quantisation used, and from this the error yielded by the 32-bit system was calculated. Values of $|e|_{\text{av rel}}$ for FLP and $|e'|_{\text{av rel}}$ for LNS were calculated over the entire result file. The procedure is illustrated for LNS in fig. 4.1.

**Fig. 4.1. Data flow in LNS simulations**

### 4.1. Results

The algorithms evaluated are shown in fig. 4.2. The number of random samples in each input file was taken such as to yield 5000 separate evaluations of each algorithm (e.g. SUM required 10,000 samples). This was found to yield reasonably asymptotic behaviour of the average error in every case except SIGNED MAC, where 20,000 evaluations (60,000 samples) were taken.

It can be verified from the graph SUM that we have achieved the objective of designing a logarithmic addition algorithm with substantially the same error as

FLP. Results for DIFFERENCE show that the logarithmic subtraction algorithm is also comparable to FLP, except where the operands are closely matched where there is a discrepancy. As the LNS add and subtract curves are almost identical, it is probable that this derives from a particular strength of FLP in this region, rather than a weakness in LNS. This effect is much less significant when randomly signed operands are used, as in SIGNED SUM, which exercises the addition and subtraction operations equally.

The addition and multiplication algorithms are jointly exercised in MAC and SOP. In all cases the LNS error is less than that of FLP, particularly where the operands have a wide dynamic range, when the LNS error reduces to between a half and a quarter of that yielded by FLP. Even more striking are the results for SIGNED MAC and SIGNED SOP, where the error yielded by the LNS is around a quarter of that of FLP for operands of similar magnitude. The instability in the behaviour of the FLP error in signed MAC around a dynamic range of $10^{16}$ is puzzling. It is possible that this effect is related to the fact that at this range the most significant exponent bit is just coming into use, and that the quantisation intervals are therefore reaching their maximum. The LNS implementation does not exhibit this behaviour.

The final graph, Gauss-Jordan, is a larger numeric kernel taken from [13], and dominated by a roughly equal proportion of subtract and multiply instructions. It was run using a randomly generated matrix of coefficients in the range $\pm(0 .. 1) * 1$, for $(N * N)$, $(N * 1)$ matrices. The average error over 100 trials for each matrix size is shown. Evidently the LNS offers a clear improvement in accuracy over FLP; in the examples studied it yielded, on average, 66% of the FLP error.

### 4.2. Discussion

These results suggest that the LNS will offer a broadly twofold improvement in accuracy over FLP, which is consistent with what would be expected from the fact that in a roughly equal mixture of adds and multiplies the adds will still have a half-bit rounding error whereas the multiplies will be infinitely accurate. The benefits to be gained by using LNS will vary, depending on the ratio of add / subtract to multiply / divide operations, and the sign and range of the operands.

## 5. Conclusion

Hitherto, LNS arithmetic devices had offered either better speed or better accuracy than FLP but not both. Alternatively they had been restricted to short wordlengths. We have now demonstrated that it is possible to design a 32-bit LNS arithmetic unit which will perform with substantially better speed and accuracy than FLP. The exact speed improvement depends on the ratio of adds to multiplies, but is about twofold at 40% to 60%. The improvement in accuracy is harder to predict, as it depends also on the range of the operands, but is generally around twofold in simulated arithmetic kernels.

A further advantage is that a LNS ALU requires the design of only one substantial piece of hardware, whereas the design of a FLP unit involves separate consideration of the adder, multiplier and divider.

Algorithmic complexity is exploding in almost all areas of advanced computation, and a great many applications are now becoming bounded by the limits currently imposed by FLP execution. Examples include real-time applications such as the large class of RLS-based algorithms and sub-space methods which will be required in broadcasting and cellular telephony; Kalman filtering and Riccati-like equations used in advanced real-time control; and graphics systems. Ways are urgently being sought to bypass this limitation by improving the speed at which the basic arithmetic operations can be performed. The results we have presented here suggest that a logarithmic arithmetic unit will offer a valuable means by which to achieve this objective.

**Acknowledgement**

This work is supported by the ESPRIT Long-Term Research programme, grant nos 23544 and 33544. We are grateful to Dr Jiri Kadlec of the Czech Academy of Sciences, Prague, for his many helpful comments on the first draft of this work.

**References**

[1] N.G. Kingsbury and P.J.W. Rayner, Digital Filtering Using Logarithmic Arithmetic, *Electronics Letters* Vol. 7 (1971) 56-58.

[2] T. Kurokawa, J.A. Payne and S.C. Lee, Error Analysis of Recursive Digital Filters Implemented with Logarithmic Number Systems, *IEEE Trans. Acoustics, Speech and Signal Processing* Vol. ASSP-28 (1980) 706-715.

[3] E.E. Swartzlander, D.V.S. Chandra, H.T. Nagle and S.A. Starks, Sign / Logarithm Arithmetic for FFT Implementation, *IEEE Trans. Computers* Vol. C-32 (1983) 526-534.

[4] E.E. Swartzlander and A.G. Alexopoulos, The Sign / Logarithm Number System, *IEEE Trans. Computers* (December 1975) 1238 -1242.

[5] F.J. Taylor, R. Gill, J. Joseph and J. Radke, A 20 Bit Logarithmic Number System Processor, *IEEE Trans. Computers* Vol. 37 (1988) 190-200.

[6] L.K. Yu and D.M. Lewis, A 30-b Integrated Logarithmic Number System Processor, *IEEE J. Solid-State Circuits* Vol. 26 (1991) 1433-1440.

[7] D.M. Lewis, Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit, *IEEE Trans. Computers* Vol. 43 (1994) 974-982.

[8] D.M. Lewis, 114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications, *IEEE J. Solid-State Circuits* Vol. 30 (1995) 1547-1553.

[9] M.G. Thomas, T.A. Bailey, J.R. Cowles and M.D. Winkel, Arithmetic Co-Transformations in the Real and Complex Logarithmic Number Systems, *IEEE Trans. Computers* Vol. 47 (1998) 777 - 786.

[10] J.N. Coleman, Simplification of table structure in logarithmic arithmetic, *Electronics Letters*, Vol. 31 (1995) 1905-1906, and erratum (1996) 2103.

[11] M.J. Schulte and J.E. Stine, Symmetric Bipartite Tables for Accurate Function Approximation, *Proc. 13th Symposium on Computer Arithmetic*, IEEE, 1997.

[12] Texas Instruments, TMS320C44 Digital Signal Processor, 1994.

[13] W.H. Press et al, Numerical Recipies in Pascal, Cambridge, 1989.

**Fig. 4.2. Error in simulated arithmetic kernels**