An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator

David M. Lewis

Department of Electrical Engineering
University of Toronto

Abstract

This paper describes a logarithmic number system (LNS) arithmetic unit using a new method for polynomial interpolation in hardware. The use of an interleaved memory reduces storage requirements by allowing each stored function value to be used in interpolation across several segments. This strategy can be shown to always use fewer words of memory than an optimized polynomial with stored polynomial coefficients. Many accuracy requirements for the LNS arithmetic unit are possible. Although a round to nearest would be desirable, is cannot be easily achieved. The goal suggested here is to insure that the worst case LNS relative error is smaller than the worst case FP relative error. Using the interleaved memory interpolator, the detailed design of an LNS arithmetic unit is performed using a second order polynomial interpolator including approximately 91K bits of ROM.

1. Introduction

Logarithmic number system (LNS) arithmetic has been the subject of considerable attention for low and moderate precision hardware implementations, but has met with less success in higher precision implementations. This paper describes techniques used to design an LNS arithmetic unit with worst case relative error better than the worst case relative error of IEEE 754 single precision arithmetic [12]. The analysis of the data path requirements will also show that this kind of accuracy requires more accurate, and hence more expensive, approximation of the LNS arithmetic functions than has previously been provided by LNS arithmetic units.

Achieving this level of accuracy requires better methods for approximation of the LNS arithmetic functions, which is made possible by a new strategy for function interpolators. The principles of an interleaved memory function interpolator are summarized in this paper. This paper shows how to use a second order interpolator to perform LNS arithmetic with about 91K bits of ROM and two multipliers.

This work was supported by Defence Research Establishment Atlantic.

The remainder of the paper is organized as follows. Section 2 introduces terminology and reviews related work. Section 3 introduces the interpolator using interleaved memory, and section 4 provides the design analysis for a 32-bit LNS arithmetic unit. Section 5 concludes the paper.

2. Terminology and Previous Work

Before describing the details of this work, an introduction to LNS arithmetic will be provided and previous implementations of LNS arithmetic will be reviewed.

2.1 LNS Representation and Arithmetic

Base-two LNS represents a number a by the pair $<s_a,e_a>$, where s_a is a sign bit, and e_a is an N-bit fixed point number. The value represented is $a = (-1)^{s_a} \times 2^{e_a}$. Special representations, such as 0, are ignored.

FP and LNS representations have different error characteristics. The representation error of an FP number depends on the particular number, while for LNS the error is independent of the number. Both systems have the same average error, but LNS has a smaller worst-case error. On the other hand, roughly half of the FP numbers have smaller error than LNS. We use the worst case error as the property of interest, since this is all that can be guaranteed in any computation. The combination of the constant error and better worst-case behaviour of LNS make it attractive as an alternative to FP.

This paper uses relative error to consistently compare different alternatives. Comparing a F fractional bit LNS representation to a FP representation with an F bit significand and hidden bit, we first observe that the FP representation has relative error ranging from $\varepsilon_{FP} = 2^{-F-2}$, to $\varepsilon_{FP} = 2^{-F-1}$. The latter represents the worst case error. LNS represents e_a to an absolute accuracy of 2^{-F-1}, corresponding to a constant relative accuracy of $\varepsilon_{LNS} = 2^{2^{-F-1}} - 1 \approx \ln(2) \times 2^{-F-1} \approx .6931 \times 2^{-F-1} \approx 2^{-F-1.528}$, about half a bit more precision than the worst case error in FP.

1063-6889/93 $03.00 © 1993 IEEE

An accurate representation is of little use unless the arithmetic operations produce results with a similar level of accuracy. Multiplication and division in LNS are exact operations, as they require addition or subtraction of fixed point numbers, but addition and subtraction can introduce errors. Since FP performs rounding on all operations, LNS arithmetic is potentially more accurate than FP (in worst case) using the same number of bits, provided that addition and subtraction are implemented accurately.

Addition and subtraction of a and b, represented respectively by $<s_a,e_a>$ and $<s_b,e_b>$, can be performed without loss of generality by assuming that a and b are both positive and $e_a > e_b$. The standard algorithm to compute e_c for LNS addition, such that $c = a + b$ and subtraction where $c = a - b$, where c is represented by $<s_c,e_c>$ is shown in (2.1), using $f_a(r)$ as defined in (2.2), and $f_s(r)$ for subtraction as defined in (2.3).

$$e_c = e_a + f_a(r), \quad r = e_b - e_a \tag{2.1}$$

$$f_a(r) = \log_2(1 + 2^r) \tag{2.2}$$

$$f_s(r) = \log_2(1 - 2^r) \tag{2.3}$$

The central problem in LNS arithmetic is finding a low cost, accurate method of computing $f_a(r)$ and $f_s(r)$.

2.2 Previous Related Work

Most previous approaches for LNS arithmetic use polynomials to approximate the $f_a(r)$ and $f_s(r)$, although their descriptions are not usually stated in these terms. Interpolation approximates some function $f(x)$ by an approximation $\hat{f}(x)$. The range of x is divided into a set of subintervals $[x_i,x_{i+1})$, with $h = x_{i+1} - x_i$. Within each subinterval, a polynomial $\hat{f}(x)$ is used to approximate $f(x)$ using $\hat{f}(x) = \sum_{j=0}^{k} a_j \times x^j$. A direct lookup table, storing $f(x)$ for every value of x is equivalent to a 0th order polynomial with $h = 2^{-F}$. 0-order approximation with variable h in different regions has been used in the design of an LNS arithmetic unit [2]. The value of h is chosen to be as large as possible while still meeting the error constraint. Linear approximation [3], and linear approximation with a difference correction array have also been used [4]. Another approach has been to perform multiplication and division using LNS, and addition and subtraction in FP, converting between the two systems using a linear interpolator [5]. A hardware architecture for general kth order polynomial approximation of functions has also been described [6]. In this approach, a memory is used to store $(k + 1)$ polynomial coefficients a_i for each of the polynomials in each subinterval, and a datapath performs the polynomial evaluation.

3. Interpolation Using Stored Function Values

To compare the approaches for interpolating functions using polynomials, some background information on polynomial interpolation will be presented here.

In function interpolators with multiple polynomials for distinct intervals $[x_i,x_{i+1})$ it is common to perform the translation $x_e = x - x_i$, which translates the interval $[x_i,x_{i+1})$ to $[0,h)$. This translation can be easily performed in hardware by requiring that h be a integral power of 2, and splitting the binary representation x into two parts; the upper bits, appropriately shifted, produce the value of $i = \left\lfloor \frac{x}{h} \right\rfloor$ which is used for addressing the memory. The remaining lower bits produce x_e. The function $\hat{f}(x)$ is now computed using x_e as $\hat{f}(x) = \sum_{j=0}^{k} a_j \times x_e^j$. Obviously the coefficients a_j must be altered with respect to those in section 2.2 to compute the translated function. This form of computation reduces the widths of the data paths for performing the computation, since $x_e \in [0,h)$ requires fewer bits to represent than does x.

The stored function value interpolator uses a slightly different approach for interpolation intervals. The interleaved memory interpolator uses a ROM containing the actual function values $f(x_i)$, rather than stored polynomial coefficients a_i. To develop the interleaved memory approach, we will begin with a non-interleaved stored function value interpolator, shown in Figure 1. In this interpolator, the function values are stored in a $\frac{n}{k}$ word memory, with each memory word storing the k+1 data words, each of which is a set of function values $f(x_i), \cdots, f(x_{i+k})$. In this case, i is chosen according to (3.1), and $x_e \in [0,k \times h)$.

$$i = \left\lfloor \frac{x}{k \times h} \right\rfloor \times k \tag{3.1}$$

It is convenient to consider the polynomial interpolation with stored function values in terms of the translated function on $u = \frac{x_e}{h}$, with $g(u) = f(x)$. The interpolation is performed using the standard Lagrange formulation [8], which will be presented here in sufficient detail to describe the alterations made to it later in this paper.

Lagrange interpolation determines the polynomial coefficients that fit a kth order polynomial to a number of points $(u_j,g(u_j))$, $j = 0,1,...,k$. This is expressed as the system of equations (3.2)

$$a_0 + a_1 \times u_j + \cdots + a_k \times u_j^k = g(u_j), \quad u_j = 0,1,...,k \tag{3.2}$$

Using the translation above, $u_j = j$, and the system of equations can be defined using the Vandermonde matrix

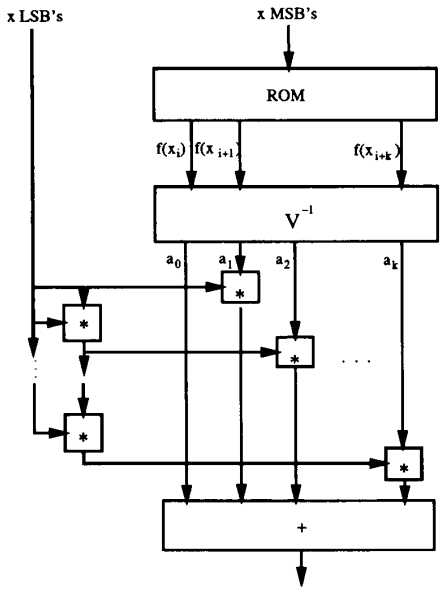

Figure 1. Polynomial interpolator with stored function values

V_k. V_k is a $(k+1)\times(k+1)$ matrix that can be used to express (3.2) in matrix form, as given in (3.3).

$$V_k \times \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ a_k \end{bmatrix} = \begin{bmatrix} g(0) \\ g(1) \\ \cdot \\ \cdot \\ g(k) \end{bmatrix} \tag{3.3}$$

The entries of V_k, $v_{k,ij}$, are given by u_{i-1}^{j-1}, or for the transformation given, $(i-1)^{(j-1)}$. 0^0 is taken to be 1. The interpolation uses a_i determined by (3.4).

$$\begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ a_k \end{bmatrix} = V_k^{-1} \times \begin{bmatrix} g(0) \\ g(1) \\ \cdot \\ \cdot \\ g(k) \end{bmatrix} \tag{3.4}$$

The purpose of transforming into u is to have a set of u_j which are constant regardless of h or i, so V_k is also constant regardless of the particular x. Since V_k is constant, so is V_k^{-1}, and the a_i can be calculated by datapaths designed to perform multiplication by V_k^{-1}. This is much less complicated than general matrix multiplication. Each of the multiplications by integers in V_k^{-1} can be performed by a specialized constant multiplier, which is much less complex than a general integer multiplier. For example, multiplication by 6 would be performed using a single adder with the multiplier shifted by one and two bits as its respective inputs. For example, V_2 and V_4 are shown in Figure 2, together with V_2^{-1} and V_4^{-1}.

$$V_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \qquad V_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{2} & 2 & -\frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{bmatrix}$$

$$V_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \end{bmatrix} \qquad V_4^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{25}{12} & 4 & -3 & \frac{4}{3} & -\frac{1}{4} \\ \frac{35}{24} & -\frac{13}{3} & \frac{19}{4} & -\frac{7}{3} & \frac{11}{24} \\ -\frac{5}{12} & \frac{3}{2} & -2 & \frac{7}{6} & -\frac{1}{4} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{bmatrix}$$

Figure 2. Example matrices V_2 and V_4 with inverses

The memory values are prescaled to eliminate the need for division by factors other than powers of 2 (by 1/3 in the case of V_4^{-1}) and a dedicated multiplier uses the minimal number of adders for each multiplication. Multiplication by V_2^{-1} requires 6 operations and multiplication by $3 \times V_4^{-1}$ requires a total of 36 addition or subtraction operations,

A minor change can further reduce hardware cost. The complexity of (3.4) depends on the binary representation of the numbers in V_k^{-1}. This in turn is due to the range of integers in V_k, which range from 0 to k^k. The transformation $u = \frac{x_e}{h} - \delta$, for some specified δ, uses the same set of points, but results in u having a smaller range. u now has the range $[-\delta, k-\delta)$, and the Vandermonde matrix is redefined as $V_{k,\delta}$, with entries $v_{k,\delta,ij} = (i-1-\delta)^{(j-1)}$. The magnitude of the entries in $V_{k,\delta}$ can be minimized and constrained to be integers by choosing $\delta = \left\lfloor \frac{k}{2} \right\rfloor$. The matrices $V_{2,1}$ and $V_{4,2}$ and their inverses are shown in Figure 3. Multiplication by $V_{2,1}^{-1}$ requires 3 additions, and $3 \times V_{4,2}^{-1}$ requires 15 additions, less hardware than a typical integer multiplier.

This structure can be modified to use interleaved memory, which reduces the amount of ROM storage required. Instead of using each distinct set of function values $f(x_i) \cdots f(x_{i+k})$ for the interpolation of $f(x)$ with $x \in [x_i, x_{i+k})$, each distinct set of function values is used only for interpolation for $x \in [x_{i+\lambda}, x_{i+\lambda+1})$. The motivation for this is the observation that the maximum error differs between subintervals $[x_j, x_{j+1})$, where $i \le j < i+k$. Choosing $\lambda = \left\lfloor \frac{k}{2} \right\rfloor$ will always result in the selection of

$$V_{2,1} = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad V_{2,1}^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{bmatrix}$$

$$V_{4,2} = \begin{bmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{bmatrix} \qquad V_{4,2}^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ -\frac{1}{24} & \frac{2}{3} & -\frac{5}{4} & \frac{2}{3} & -\frac{1}{24} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{bmatrix}$$

Figure 3. Sample values of $V_{k,\delta}$ and inverses

the interval that minimizes interpolation error.

This modification on its own appears to increase the amount of storage required, since each set of $k + 1$ function values is used now to span the interval an interval h wide instead of $k \times h$. This potential problem is avoided by storing the function values in an interleaved memory. The interleaved memory allows simultaneous access to any $k + 1$ consecutive words, and reduces the amount of memory required.

The hardware implementation of interleaved memory replaces the ROM in Figure 1 with a collection of $P = 2^p \geq k + 1$ ROMs, as shown in Figure 4. P is required to be a power of 2 so that the computation of a value modulo P is easy to perform. Each of the values $f(x_i)$ through $f(x_{i+k})$ are located in words $\left\lfloor \frac{i}{P} \right\rfloor$ or $\left\lfloor \frac{i}{P} \right\rfloor + 1$. An incrementer is used to generate the latter value, and a collection of multiplexers selects the appropriate address for each ROM. This method reduces the total amount of ROM by a factor of $k + 1$.

A better approach in VLSI is to perform the interleaving on a smaller collection of words. A single ROM, with $P + k$ function values stored in each word can be used, as shown in Figure 5. A set of multiplexers can be used to select the appropriate function values. This increases the memory requirement by a factor of $\frac{P+k}{P}$, but can reduce area in a VLSI circuit because of the merging of multiple ROMs into a single ROM.

It can be shown that the optimal choice for λ is exactly the same as the optimal choice for δ. This coincidence can be exploited to redefine the interpolation equations in a simpler manner as shown in (3.5) through (3.7). This defines an interpolation on the interval $[x_i, x_{i+1})$ using $f(x_{i-\lambda})$ through $f(x_{i+k-\lambda})$, which is equivalent to the previous description. It is also clear

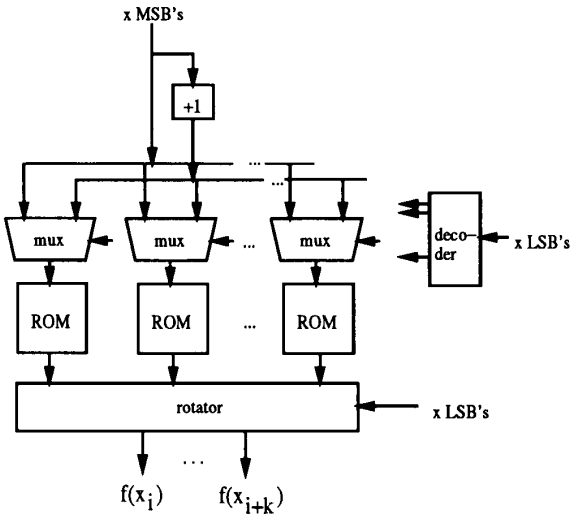

Figure 4. Interleaved memory for coefficient storage

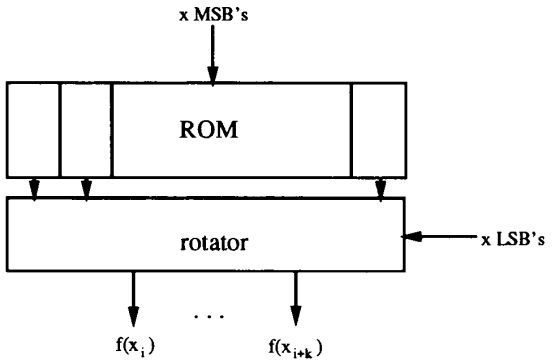

Figure 5. Single ROM implementation of interleaved memory

from (3.5) and (3.6) that $u \in [0,1)$, reducing the width of the data paths compared to the previous interpolator.

$$i = \left\lfloor \frac{x}{h} \right\rfloor \tag{3.5}$$

$$u = \frac{x}{h} - i \tag{3.6}$$

It can be shown that, for a given amount of memory, interleaved memory function interpolators always achieve higher accuracy than do stored coefficient interpolators, even compared to stored coefficient interpolators that are optimized to minimize interpolation error [7]. Furthermore, it is possible to optimize the stored function values to reduce error further. The details of this are beyond the

$$\hat{f}(x) = \left[V_{k,\delta}^{-1} \times \left[\begin{array}{c} f(x_{i-\lambda}) \\ \cdot \\ \cdot \\ \cdot \\ f(x_{i+k-\lambda}) \end{array} \right] \right]^T \times \left[\begin{array}{c} 1 \\ u \\ \cdot \\ \cdot \\ \cdot \\ u^k \end{array} \right] \qquad (3.7)$$

scope of this paper.

4. Design of Accurate LNS Arithmetic Unit Using Interleaved Memory Interpolator

Applying these techniques to the design of the LNS arithmetic unit first requires the analysis of the allowable error in the approximation of $f_a(r)$ and $f_s(r)$. While many implementations of LNS arithmetic have been described, we are unaware of a precise evaluation of the accuracy requirements.

4.1 Accuracy Requirements in LNS Arithmetic

An important point is that LNS arithmetic algorithms using approximations to $f_a(r)$ and $f_s(r)$ are cannot use an error criteria with the same mathematical simplicity as FP arithmetic. FP arithmetic can be implemented exactly, computing an infinitely precise result, followed by a rounding step to F bits, for a total relative error no greater than 2^{-F-1}. An attractive error criteria for LNS would be an absolute error in $f_a(r)$ and $f_s(r)$ of no more than 2^{-F-1}, achieving the full possible precision of LNS representation. Unfortunately, $f_a(r)$ and $f_s(r)$ are transcendental functions, so any approximation of them in a finite number of bits (other than a direct lookup table) introduces some error. An approximation of $f_a(r)$ and $f_s(r)$ will also incur a rounding error as this is rounded to F bits. If the approximation of $f_a(r)$ and $f_s(r)$ has an error of ε_{dp} (datapath error), then the total error in e_c will be $\varepsilon_{dp} + 2^{-F-1}$. The total relative error is then given by ε_{LNS} in (4.1).

$$\varepsilon_{LNS} = 2^{\left[\varepsilon_{dp} + 2^{-F-1}\right]} - 1 \approx \ln(2) \times \left[\varepsilon_{dp} + 2^{-F-1}\right] \qquad (4.1)$$

It is possible to choose any non-zero ε_{dp}, with increasing cost as the error constraint is made smaller. This means that an LNS arithmetic unit can be designed with any given ε_{LNS} strictly larger, but not equal to, than $\ln(2) \times 2^{-F-1}$. One possible goal, used here, is to require that for all possible inputs, the worst case error of LNS arithmetic be no greater than the worst case error of FP arithmetic. This requires that the relative error be less than ε_{FP}, as stated in (4.2). This can be used to derive a constraint on ε_{dp}, as done in (4.3), and approximately in (4.4), by noting that $\left[\frac{1}{\ln(2)} - 1\right] \approx 1.771 \times 2^{-2}$.

$$\ln(2) \times \left[\varepsilon_{dp} + 2^{-F-1}\right] \le 2^{-F-1} \qquad (4.2)$$

$$\varepsilon_{dp} \le \left[\frac{1}{\ln(2)} - 1\right] \times 2^{-F-1} \qquad (4.3)$$

$$\varepsilon_{dp} \le 1.771 \times 2^{-F-3} \approx 2^{-F-2.18} \qquad (4.4)$$

The final constraint, eqn. (4.4), is somewhat surprising. It indicates that approximation of $f_a(r)$ and $f_s(r)$ must be performed to almost three more bits of precision than an FP arithmetic unit to achieve the above goal of comparable worst case accuracy. This allowable error includes all sources of error: interpolation error, ROM word finite precision rounding error, and data path rounding or truncation errors. Previous LNS arithmetic units have adopted weaker constraints.

The precision of FP arithmetic is conventionally constrained to be a fixed relative error compared to the infinite precision result. This has the virtue of mathematical simplicity, but leads to excessive cost for LNS arithmetic units. In particular, $f_s(r) \to -\infty$ as $r \to 0$, making approximation difficult. This corresponds to subtracting two nearly equal numbers. While FP incurs no extra costs for infinitely precise computations in this region, the nonlinearity of $f_s(r)$ requires small h and makes the memory cost excessive. Since the phenomenon of catastrophic cancellation makes the result progressively less meaningful as it approaches 0, we use a "weak" error approach such that the result for subtraction of two nearly equal numbers is only be computed to a given relative accuracy of the larger operand (also suggested in [9].) Here, "nearly equal" means two numbers such that $r \ge -1$, i.e., the numbers differ in magnitude by a factor of two or less.

For such numbers, the absolute error allowable in the result of $a - b$, assuming $a \ge b$ is $a \times 2^{-F-1}$, or an allowable relative error in the result of $\frac{a \times 2^{-F-1}}{a - b}$. We define $\varepsilon_{weak} = \varepsilon \times \frac{a - b}{a}$ where ε is the relative error in the computation of $a - b$.

4.2 Design of Datapaths for LNS Adder/Subtractor

The most complex part of the design is the analysis of the data path requirements for performing LNS addition and subtraction. For an $F = 23$ data path, comparable to IEEE 754 single precision FP [12], the total error must be less than 1.771×2^{-26}. This error budget must be allocated to the various sources of error in the hardware, while minimizing total cost. Following [10], it is possible to allocate some fraction to each source of error and form a set of equations constraining h and the widths of the datapaths. While this offers an explicit design procedure, it is complicated to perform. It also does not directly yield the minimal cost design, since the use of worst case error estimates often lead to a pessimistic result and excessive hardware cost. Various error sources may be

smaller than the worst case bound, but and some may be anti-correlated, having smaller worst case total error than the sum of individual worst case errors. This suggests a first-order mathematical approach, followed by iterative improvements to the design.

The first design decision is the order of the interpolator. Based on previous designs [10], a linear ($k = 1$) interpolator is capable of meeting the requirements with approximately 2MB of ROM. The use of a second order interpolator can reduce this memory requirement with the inclusion of two multipliers, which can be expected to occupy less area than the ROM. Further increasing the order of the interpolator can reduce the amount of ROM, but requires more multipliers. For this application, an approximate estimate of implementation complexity suggested that a second order, $k = 2$, interpolator minimizes total circuit cost, including both ROMs and multipliers.

The calculation for second order interpolation, which uses $\lambda = 1$ and $\delta = 1$, appear in (4.5) through (4.8)

$$a_0 = f(x_i) \tag{4.5}$$

$$a_1 = \frac{1}{2} \times \left[f(x_{i+1}) - f(x_{i-1}) \right] \tag{4.6}$$

$$a_2 = \frac{1}{2} \times \left[f(x_{i+1}) - 2 \times f(x_i) + f(x_{i-1}) \right] \tag{4.7}$$

$$\hat{f}(x) = a_0 + u \times (a_1 + u \times a_2) \tag{4.8}$$

The data paths for this interpolator are shown in Figure 6. The design details consist of selecting the value of h for the functions $f_a(r)$ and $f_s(r)$, the sizes of the multipliers, the accuracy of representation of $f(x_i)$ in the ROM, and the widths of the data paths. The multipliers are not necessarily capable of handling the full precision data that occurs in the interpolation calculation. Instead, shifters are placed on the inputs and outputs of each multiplier to select as many significant bits as possible. The sizes of the multipliers must be selected to keep total error within bounds. The design parameters are shown in Table 1.

After initial studies, it was found that m1h and m2h were similar, so detailed hardware design was simplified by constraining the height of all multipliers to be the same.

Because of the highly non-linear nature of $f_a(r)$ and $f_s(r)$, the domain of these functions is divided into a set of intervals, as in [10], and h is chosen independently for each of these intervals. The intervals are defined as shown in Table 2. Note that the singularity of $f_s(r)$ at $r = 0$ requires progressively smaller intervals near 0. For this reason, the domain of the function $f_s(r)$ is split into two regions, introducing $f_{ss,i}$ and $f_{ssx,i}$ for r near 0. Table 2 also shows the number of distinct points in each interval. Intervals $f_{ss,i}$ and $f_{ssx,i}$ are identical; the difference will be that we use the exact error model for $f_{ssx,i}$ and the weak accuracy error model for $f_{ss,i}$. Either $f_{ss,i}$ or $f_{ssx,i}$ must be used in an arithmetic unit, but not both.

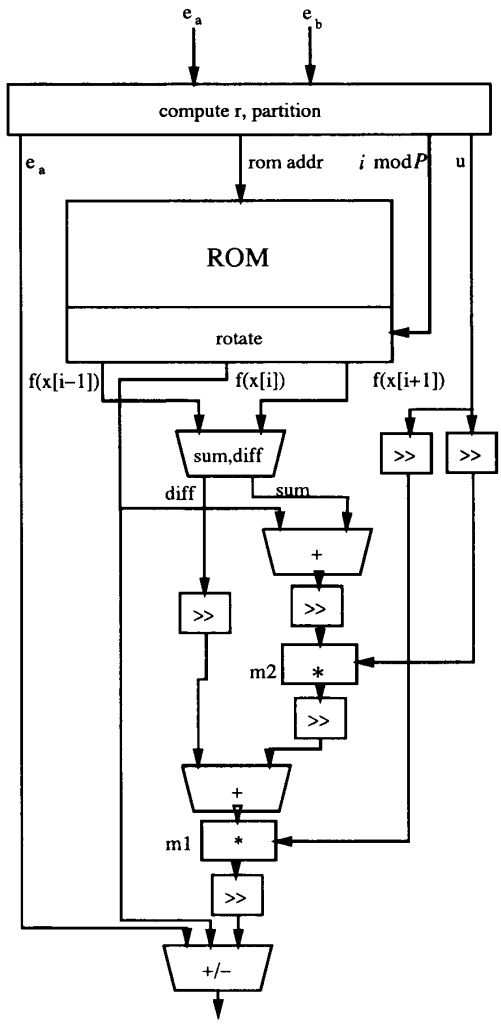

Figure 6. Datapaths for 32 bit LNS arithmetic unit

h	h for interpolation
F_{ROM}	precision of ROM
F_{DP}	precision of data paths
m1w,m1h	width and height of m1 multiplier
m2w,m2h	width and height of m2 multiplier

TABLE 1. Data Path Design Parameters

To reduce the complexity of the design procedure, a hybrid approach was used. Each source of error was modeled, but the design was optimized with an empirical analysis. The initial design was made conservatively, and meets the error specification. The accuracy of various data paths were then reduced as much as possible without

interval name	function and interval	points per interval
$f_{a,i}$	$f_a(r),\ -i-1 \le r < -i$	2^F
$f_{s,i}$	$f_s(r),\ -i-1 \le r < -i,\ i \ge 1$	2^F
$f_{ss,i}\, f_{ssx,i}$	$f_s(r),\ -2^{-i} \le r < -2^{-i-1}$	2^{F-i-1}

TABLE 2. Intervals for Approximating Functions

violating the error specification. Because it is essential that the design meet the error specification for every value of r, a functional model of the design has been constructed to test the algorithm. This model is a C language program that provides an exact functional model of the hardware design of Figure 6, and computes the relative error of the hardware for any given addition or subtraction. The program uses integers to represent all values in the datapath, truncated to the appropriate widths at each stage, and compares the result to that computed using double precision IEEE FP arithmetic. It is parameterized so that the key design parameters in Table 1 can be specified. The program is fast enough that all possible input values of r for both addition and subtraction (a total of $2 \times 24 \times 2^{23} \simeq 4 \times 10^8$ cases) can be exhaustively verified in about 6 CPU hours on a 20-MIP machine. A quick verification of 1/249 of the cases can be performed in two minutes, and the exhaustive verification performed if the short one does not produce any errors. The final design meets the error specification for every input value of r.

Detailed design took place by calculating the value of h required for each interval. As an initial design specification, we allowed the interpolation error to be up to 2^{-26}, allowing $.771 \times 2^{-26}$ for the rest of the data path errors. The simplest approach is to choose successively smaller powers of 2 for values of h, until the computed interpolation error meets the bound. We also assumed that the error of interpolating each function is monotonic in each interval, so used the smaller h of the two endpoints. Detailed data path design took place by using mathematical analysis of the maximum values of the a_i and determining allowable error at each stage. Tradeoffs were made to minimized expected VLSI implementation area.

Total table space is 2218 words or 68758 bits for the weak error model, and 9284 words or 287804 bits for the precise error model. As speculated above, there is a dramatic increase in table size by using the exact error model, which we do not regard as worth the minimal increase in accuracy. These numbers must be increased by 25% to account for the $P = 8$-way interleaving of the memory, with $k = 2$ extra words in each ROM word. After rounding each table segment up to a multiple of 8 words, then adding 25%, a total of 95680 bits of ROM are required.

i	Table Words for Interval			
	$f_{a,i}$	$f_{s,i}$	$f_{ss,i}$	$f_{ssx,i}$
0	128	-	128	512
1	128	256	128	512
2	128	128	64	512
3	64	128	64	512
.				
.				
.				
24	2	2	2	64
total	770	834	614	7680

TABLE 3. Numbers of words, $1/h$ for LNS arithmetic unit

Table 5 shows the minimum and maximum relative errors found across all of the intervals, together with average error and average absolute error. The error for $f_{ss,i}$ is stated in terms of the weak error ε_{weak}, while all others are relative error. These errors, in terms of least significant bits, show that the arithmetic unit outperforms FP arithmetic, which would have an error range of -.5 to .5, and average absolute error of .25. The average absolute error for $f_{a,i}$ and $f_{s,i}$ is about log(2)×.25, which would be expected for a precise LNS arithmetic unit. The average absolute error for $f_{ss,i}$ is much smaller. This is because $f_s(r)$ is highly non-linear in each $f_{ss,i}$, and most values of r result in an error much smaller than the worst case. On the other hand, the average absolute error is roughly log(2)×.5 for $f_{ssx,i}$ because larger absolute error in $f_s(r)$ occurs where the result is larger, keeping the relative error roughly constant. The error has a slight bias, due to design for minimum absolute error instead of minimum bias, and due to the use of truncation in the datapaths.

F_{ROM}	26
F_{DP}	30
m1w,m1h	19 , 16
m2w,m2h	12 , 16

TABLE 4. Data Path Parameters for Optimized Design

regions	ε_{min}	ε_{max}	$\bar{\varepsilon}$	$\overline{\lvert\varepsilon\rvert}$
$f_{a,i}$	-.495	.464	.0056	.173
$f_{s,i}$	-.459	.440	-.0063	.168
$f_{ss,i}$	-.285	.456	.0134	.059
$f_{ssx,i}$	-.400	.475	.0103	.174

TABLE 5. Relative Error Characteristics of LNS Arithmetic Unit Design for all Possible Inputs in Units of 2^{-23}

Figure 7 shows a histogram of the relative error of the processor for every possible input value of r. For $f_{a,i}$ and $f_{s,i}$, the error distribution is relatively flat, while for $f_{ss,i}$

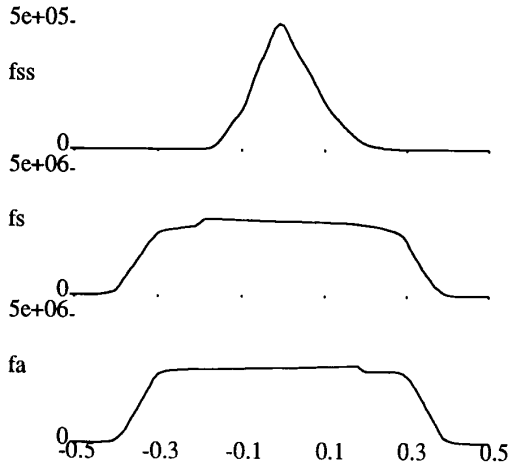

Figure 7. Histogram of relative error distribution of LNS arithmetic processor

the distribution is concentrated towards the center. This is due to the fact $f_s(r)$ is highly non-linear near $r = 0$ and the error varies greatly across each interval, while the weak error model compares the relative error to a constant input value.

A detailed logic design of the arithmetic unit has been constructed and simulated, and verifies the expected accuracy of the arithmetic unit.

4.3 Comparison to Previous Implementations

Table 6 compares the implementation of this paper to some other implementations. It uses the number of bits of ROM and multiplier cells as an estimate of complexity. All other implementations have worse error relative characteristics in terms of 2^{-F} than this implementation. Reference [5] describes the only other implementation with comparable precision and complexity, but has error more than five times greater than this implementation.

reference	F	ROM(k)	mpy
[2]	12	154	0
[11]	20	251	166
[5]	23	198	432
this work	23	91	496

TABLE 6. Comparison to other LNS implementations

5. Conclusions

This paper has introduced an architecture for polynomial function interpolation, using interleaved memories with stored function values. This architecture, which has low hardware cost and interpolation error, can be applied to LNS arithmetic units. A particular example of an arithmetic unit with worst case relative error better than the worst case relative error of single precision floating point has been described. Design verification on the architectural and logic design levels is complete, and shows no errors. Using these algorithms, LNS arithmetic may soon be seen to be competitive with FP arithmetic.

6. Acknowledgements

Mark Arnold suggested a linear interleaved memory interpolator as an improvement to the LNS unit described in [11].

7. References

[1] M.G. Arnold, T.A. Bailey, J.R. Cowles, and M.D. Winkel, "Applying Features of IEEE 754 to Sign Logarithm Arithmetic", in *IEEE Trans. Comput.*, Aug 1992, pp 1040-1050

[2] F. J. Taylor , R. Gill, J. Joseph, and J. Radke, "A 20 bit Logarithmic Number System Processor" in *IEEE Trans. Comput.*, Feb. 1988, pp 190-200

[3] M. Combet, H. Van Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers", in *IEEE Trans. Electron. Comp.*, Dec 1965, pp 863-867

[4] H-Y Lo and Y. Aoki, "Generation of a Precise Binary Logarithm with Difference Grouping Programmable Logic Array", in *IEEE Trans. Comput.*, Aug. 1985, pp 681-691

[5] F.-s. Lai and C.-F.E. Wu, "A Hybrid Number System Processor with Geometric and Complex Arithmetic Capabilities", in *IEEE Trans. Comput.*, Aug. 1991, pp 952-962

[6] A.S. Noetzel, "An Interpolating memory Unit for Function Evaluation: Analysis and Design", in *IEEE Trans. Comput.*, Mar. 1989, pp 377-384

[7] L. Fox and I. Parker, **Chebyshev Polynomials in Numerical Analysis**, Oxford University Press, 1968

[8] E.W. Cheney, **Introduction to Approximation Theory**, McGraw-Hill, 1966

[9] M.G. Arnold, T.A. Bailey, J.R. Cowles, and J.J Cupal, "Redundant Logarithmic Arithmetic", in *IEEE Trans. Comput.*, Aug. 1990, pp 1077-1086

[10] D. M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System", in *IEEE Trans. Comput.*, Nov. 1990, pp 1326-1336

[11] D. M. Lewis and L. K. Yu, Algorithm Design for a 30 bit Integrated Logarithmic Processor, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, Sept, 1989, pp 192-199

[12] IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Std 754-1985, IEEE, 1985