Algorithm Design for a 30 bit Integrated Logarithmic Processor

David M. Lewis and Lawrence K. Yu

Department of Electrical Engineering, University of Toronto

Abstract

This paper describes the architecture of an integrated processor that is capable of performing addition and subtraction of 30 bit numbers with 20 fractional bits in the logarithmic number system. Previous techniques would require 70Mb of ROM to implement this processor, infeasible for a single chip implementation. The techniques presented here use a factor of 275 less memory. The key to this is the use of a linear approximation of the non-linear functions stored in the lookup tables. The functions involved are highly non-linear in some regions, so variable size regions are used for the approximation.

The use of linear approximation alone would still require over 565Kb of ROM. Further compression is obtained by using linear approximation with differential coding of each table. The compression is chosen to minimize ROM size, and obtains a further reduction of 55%. A total of 260K bits of ROM are required to implement the processor.

1. Introduction

This paper describes the algorithms used by an integrated 30 bit logarithmic number system (LNS) processor. The logarithmic number system (LNS) has been in use for several years [1]. Its advantages include uniform error characteristics across the entire range of values, and better accuracy than floating point representation using the same number of bits, as well as high speed multiplication and division.

The principal disadvantage of the LNS is the difficulty in performing addition and subtraction. Addition and subtraction require lookup tables with several times 2^F words, for F bits of fractional precision [2]. For this reason, most published implementations have been restricted to 8 to 12 bits of fractional precision [3,4]. To put the difficulty of implementing high precision LNS processors using recent architectures in perspective, the 30 bit design described here would require about 70Mb of ROM to implement with the techniques used by a previous implementation [4].

Linear approximation is a useful technique for reducing the size of a lookup table. Linear approximation [5], quadratic approximation [6], and linear approximation with a non-linear difference function in a PLA [7] have been used to advantage in the approximation of some functions, such as $\log(x)$. This is possible for $\log(x)$ because of its smooth nature. The function required for subtraction in the LNS is highly non-linear, so linear approximation has not been as successful in this application. One attempt [8] has achieved better precision, by using a modified linear approximation, but is limited to about 3.85 additional bits of precision. Furthermore, this method is only described for addition. This paper concluded that linear approximation was impractical from the hardware standpoint for logarithmic arithmetic.

Two problems were encountered in [8]. The first was the need for lookup tables with two inputs to perform the linear approximation. This led to the modified linear approximation scheme, which cannot achieve the full precision possible in the LNS. Secondly, although not stated in [8], the subtraction function in LNS is highly non-linear, and not easily adapted to a linear approximation. Thus, the most recent previous implementations of LNS addition do not use linear approximation.

This paper uses two techniques to greatly increase the precision possible for a given amount of ROM. First, a new technique for linear approximation is used to reduce the amount of table storage required. The use of this technique alone achieves substantial reductions in table space, with a modest increase in other components. Using linear approximation alone reduces the table space to 565K bits, a factor of 127 reduction compared to [4].

This is still too large for integration onto a single chip using the available technology. A second technique is therefore used to compress each table. The functions in the tables are too non-linear to allow linear approximation, so table compression based on linear approximation with differential coding is used. Each table analyzed in a manner that chooses the parameters for compression in a manner that minimizes ROM space. There is a considerable advantage to merging multiple tables into a single ROM, so slightly sub-optimal tables are used to minimize total chip area. The total ROM required after application of linear approximation plus differential coding is 260K bits, a factor of 275 reduction.

The paper is organized as follows. Section 2 introduces the LNS representation and the algorithms used. The chip level design is briefly described in section 3, followed by conclusions in section 4.

2. Algorithm Design

The logarithmic number system represents a number x by its sign and logarithm in some base b. Formally, the number x is represented by the pair $<s_x, e_x>$, such that $x = (-1)^{s_x} \times b^{e_x}$. In this paper we will only consider $b = 2$. e_x is an N-bit fixed point number with I integer bits and F fractional bits of precision, and $N = F + I$. We also assume

an excess-n representation of e_x, with $n = 2^{I-1} - 1$. If the bits of e_x are m_i, $i = -F, \cdots, I-1$, then the value of e_x is

$$e_x = \sum_{i=-F}^{I-1} m_i \times 2^i - 2^{I-1} + 1 \qquad (2.1)$$

The intended application of the LNS is as a replacement for floating point, so it must be possible to represent zero. For simplicity, this is done with a distinct bit, z_x. The value represented is zero if z_x is one. A more economical representation would be to use a distinct value of e_x to represent zero, but we are interested in the least complicated design.

The number x is thus represented by the triple $<s_x, z_x, e_x>$, and has the value

$$x = (1 - z_x) \times (-1)^{s_x} \times 2^{e_x} \qquad (2.2)$$

The processor described in this paper uses a 30-bit LNS format, with I = 8 and F = 20, and two extra bits for z_x and s_x. This format is shown in Figure 1.

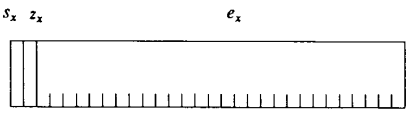

Figure 1. Operand Format

Multiplication and division in the LNS are trivial operations and will not be described. Addition and subtraction are more complicated, and are the focus of the processor described in this paper.

Letting a and b be two numbers represented in LNS format by $<z_a, s_a, e_a>$ and $<z_b, s_b, e_b>$, with $a \geq b$, and $b > 0$, addition and subtraction can be implemented using the formulae presented below:

Addition:
$$c = a + b$$
$$e_c = e_a + fa(e_b - e_a)$$
$$f_a(r) = \log(1 + 2^r)$$

Subtraction:
$$c = a - b$$
$$e_c = e_a + f_s(e_b - e_a)$$
$$f_s(r) = \log(1 - 2^r)$$

In this paper $\log(x)$ means the logarithm to base 2, $\ln(x)$ means the natural logarithm of x, and $\exp(x)$ means 2^x.

The central difficulty in LNS arithmetic is the implementation of the functions $f_a(r)$ and $f_s(r)$. The most recent implementation [4] represents each of these functions by a set of ROMs, using some bits of r as inputs to the ROM. The bits selected as inputs to the ROM depend on the derivative of the function stored in the ROM. In regions in which the derivative of the function is small, some of the least significant bits of the function parameter will have an insignificant effect (less than the accuracy requirement) on the function value. These bits do not need to be input to the ROM, reducing the size of the table. In the strategy used by this implementation, addition and subtraction each require F separate tables containing from 1 to 2^F words are required, containing a total of roughly 4×2^F words, each of F bits. Further optimization is used by [4] to reduce the table size by another 20%.

The exponential dependence of ROM size on F forms the primary limit in the precision attainable in the LNS. An implementation of acceptable chip area in 3 micron CMOS technology is limited to about 250K bits of ROM. With this limit, F cannot be made larger than 12.

The processor described in this paper uses new techniques to reduce the ROM requirements to an acceptable amount. The essential result of the application of these techniques is to replace the 2^F dependence of ROM size by a larger linear factor, but a $2^{F/2}$ dependence. This allows much more precision with a given amount of ROM.

2.1 Linear Extrapolation

The first technique used in this processor is linear approximation as a method of implementing $f_a(r)$ and $f_s(r)$. A linear approximation of some function $f(x)$ in the neighbourhood of x, is defined by (2.3).

$$\hat{f}(x + \Delta x) = f(x) + \frac{df(x)}{dx} \times \Delta x \qquad (2.3)$$

This formulation appears to require a separate ROM for $\frac{df(x)}{dx}$ and a multiplication, which is potentially expensive. Further consideration of the functions involved will show that both of these can be eliminated.

First, the multiplication can be eliminated by using logarithmic arithmetic, so (2.3) can be replaced by (2.4) if $\frac{df(x)}{dx} > 0$ and (2.5) if $\frac{df(x)}{dx} < 0$.

$$\hat{f}(x+\Delta x) = f(x) + sgn(\Delta x) \times \exp\left[\log(|\Delta x|) + \log\left[\frac{df(x)}{dx}\right]\right] \qquad (2.4)$$

$$\hat{f}(x+\Delta x) = f(x) - sgn(\Delta x) \times \exp\left[\log(|\Delta x|) + \log\left[\frac{-df(x)}{dx}\right]\right] \qquad (2.5)$$

The function $sgn(x)$ in (2.4) and (2.5) is the sign function.

The function $f_a(r)$ has a positive derivative, and is approximated using (2.4), while $f_s(r)$ always has a negative derivative, and is approximated using (2.5).

The advantage of (2.4) and (2.5) is not immediately clear. These appear to be more complex to implement than (2.3), replacing a table of $\frac{df(r)}{dr}$ by a table of $\log\left[\frac{df(r)}{dr}\right]$, and requiring additional logic to perform logarithms and exponentials. In fact, due to the particular functions under consideration, $f_a(r)$ and $f_s(r)$, (2.4) and (2.5) can be used as the basis for further simplifications. Considering the formulae for $\log\left[\frac{df_a(r)}{dr}\right]$ and $\log\left[-\frac{df_s(r)}{dr}\right]$, derived in (2.6) through (2.11), it can be seen that each of these is closely related to $f_a(r)$ and $f_s(r)$. Thus, each of these can be easily calculated, eliminating the need for lookup tables containing their values. Although additional ROMs are required for log() and exp(), it will be seen that these are relatively small.

$$\frac{df_a(r)}{dr} = \frac{2^r}{1+2^r} \qquad (2.6)$$

$$\log\left[\frac{df_a(r)}{dr}\right] = r - \log(1+2^r) \qquad (2.7)$$

$$\log\left[\frac{df_a(r)}{dr}\right] = r - f_a(r) \tag{2.8}$$

$$\frac{df_s(r)}{dr} = -\frac{2^r}{1-2^r} \tag{2.9}$$

$$\log\left[-\frac{df_s(r)}{dr}\right] = r - \log(1-2^r) \tag{2.10}$$

$$\log\left[-\frac{df_s(r)}{dr}\right] = r - f_s(r) \tag{2.11}$$

The remaining problem is how to choose x and Δx as some function of r. A simple technique for choosing these is to partition the binary representation of r into several parts, specifically r_i, r_h, r_l, and r_e, such that $r = r_i + r_h + r_l + r_e$. Also define $r_t = r_i + r_h$. The approximation will be performed with $x = r_t$, $\Delta x = r_l$, and r_e will be ignored. This leads to the approximation (2.12).

$$\hat{f}(r) = f(r_t) + r_l \times \frac{df(r_t)}{dr} \tag{2.12}$$

The values of each of these quantities are described by two integers p_l and p_e, $p_e \le p_l$ and $p_l < 0$, that partition the binary representation of r. p_l and p_e are not constants, but are functions of r (although not necessarily the same for both $f_a(r)$ and $f_s(r)$). Given some p_l and p_e, the corresponding values of r_i, r_h, r_l, and r_e are defined by (2.13) through (2.16).

$$r_e = r_{p_e-1} \cdots r_{-F} \tag{2.13}$$

$$r_l = r_{p_l} \cdots r_{p_e} - 2^{p_l} \tag{2.14}$$

$$r_h = r_{-1} \cdots r_{p_l+1} + 2^{p_l} \tag{2.15}$$

$$r_i = r_{l-1} \cdots r_0 \tag{2.16}$$

The notation $r_n \cdots r_m$ means the value of the binary representation of bits m through n inclusive of r.

The result is that r_h is a positive quantity, with $1 - p_l$ bits being dependent upon r, and r_l is a signed quantity with $p_l - p_e$ significant bits, and $|r_l| \le 2^{p_l}$. The choice of r_l as a signed quantity rather than an unsigned quantity eliminates one bit from r_h, while maintaining the same constraint on the absolute value of r_l. Finally, r_e is positive and $r_e < 2^{p_e}$, r_i is the integer part of r, so $r < r_i + 1$.

Combining this partitioning of r with the values of the derivatives in (2.8) and (2.11) and the approximations (2.4) and (2.5) leads to the formulae (2.17) and (2.18) as approximations for $f_a(r)$ and $f_s(r)$.

$$\hat{f}_a(r) = f_a(r_t) + sgn(r_l) \times \exp\left[\log(|r_l|) + r_t - f_a(r_t)\right] \tag{2.17}$$

$$\hat{f}_s(r) = f_s(r_t) - sgn(r_l) \times \exp\left[\log(|r_l|) + r_t - f_s(r_t)\right] \tag{2.18}$$

An overview of the architecture that implements addition and subtraction using linear approximation for $f_a(r)$ and $f_s(r)$ is shown in Figure 2. This data path does not show the calculation of r, since it is trivial. The box labeled split chooses p_e and p_l, while the partition box splits up r into the components r_t and $|r_l|$. The arithmetic performed by the remainder of the data paths can be best described by the algorithms shown in Figure 3.

Figure 2 does not show the precisions of the various data paths, for simplicity. The output of the $f_a()$ and $f_s()$ ROMs is assumed to have precision F_f, so data paths *frt* and *fd* have precision F_f. All references to precision refer only to the number of fractional bits, with the number of integer bits being determined by the range of values that must be represented. The data paths are *lrl*, *lcor*, and *cor* have precisions F_{lrl}, F_{lcor}, and F_{cor} respectively. Other data paths have precision that is equal to the maximum precision data path that is used to compute the value.

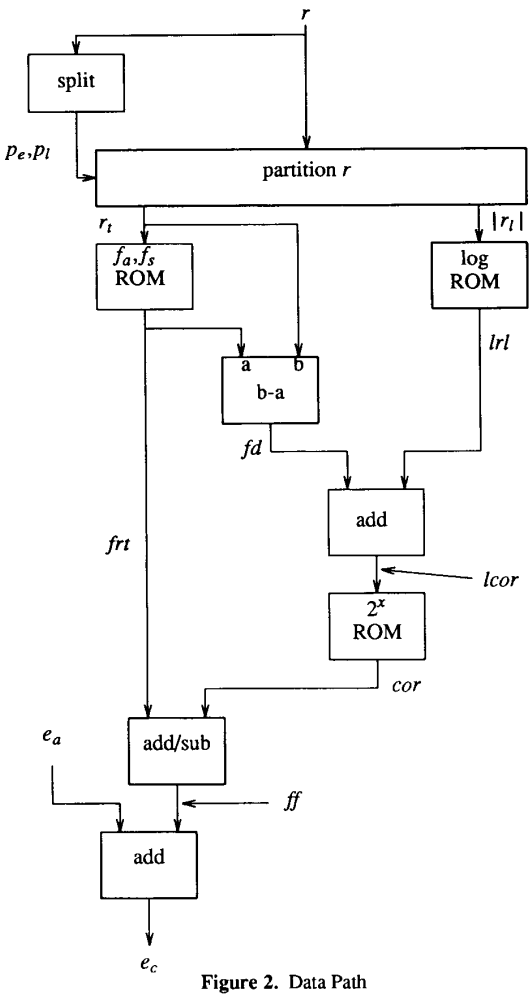

Figure 2. Data Path

The design problem for this level of the architecture is to determine the values of p_l and p_e, as well as the widths of all data paths. The details of these choices are complex, are presented in another paper [9], so only a brief overview will be given here.

The choice of p_l and p_e respectively determine the length of the region that it is possible to linearly approximate over and that part of r that may be ignored. Since $\Delta x = r_l$, and $|r_l| < 2^{p_l}$, linear approximation may be performed over a region up to 2^{p_l} in length in each direction (positive and negative). Also, r_e is not used in the approximation, causing an amount of up to 2^{p_e} to be ignored. Each of these

contributes some error to the final result. The finite precision of each word in the ROM and any rounding performed by the data paths also contributes to the error of the final result. This error can be mathematically expressed as a function of r, p_l, and p_e, and the data path widths F_f, F_{lcor}, F_{cor}, F_{lrl}, say $error(r,p_l,p_e,F_f,F_{lcor},F_{cor},F_{lrl})$.

The goal is design a processor that meets some accuracy criteria, say half of a least significant bit, described by the equation $|error(r,p_l,p_e,F_f,F_{cor},F_{lcor},F_{lrl})| < 2^{-F-1}$. By expressing this equation in terms of r, it is possible to obtain a relationship between the data path widths and p_l and p_e as a function of r. Ideally, the system could then be optimized to obtain the lowest cost implementation that meets the constraint on all of these parameters.

The mathematics is essentially intractable, so certain simplifications are used. The first replaces the error function by a function which is simpler than the actual error function, as well as being at least as large. If the new error function meets the accuracy criteria, then the actual error function necessarily will meet the criteria as well.

Addition:

$$e_c = e_a + f_a(r_i) + sgn(r_i) \times \exp(\log(|r_l|) + r_i - f_a(r_i))$$

$$\begin{array}{ll} frt = fa\_tbl(r_i) & \\ lrl = log\_tbl(|r_l|) & \\ fd = r_i - frt & \\ lcor = lrl + fd & \\ cor = exp\_tbl(lcor) & \\ ff = frt + cor & \text{if } r_l \geq 0 \\ ff = frt - cor & \text{if } r_l < 0 \\ e_c = e_a + ff & \end{array}$$

Subtraction:

$$e_c = e_a + f_s(r_i) - sgn(r_i) \times \exp(\log(|r_l|) + r_i - f_s(r_i))$$

$$\begin{array}{ll} frt = fs\_tbl(r_i) & \\ lrl = log\_tbl(r_i) & \\ fd = r_i - frt & \\ lcor = lrl + fd & \\ cor = exp\_tbl(lcor) & \\ ff = frt - cor & \text{if } r_l \geq 0 \\ ff = frt + cor & \text{if } r_l < 0 \\ e_c = e_a + ff & \end{array}$$

Figure 3. Addition and Subtraction Algorithm

The second simplification is to treat each error source separately. The error function can be viewed as the sum of the errors due to each contribution. The error goal can be met by allocating each of these separate error function some fraction of the total allowable error. This is done in a manner that attempts to minimize total ROM size. As a result, p_l and p_e and the data path widths are expressed as functions of F and r.

The resulting values of all architectural parameters are shown in Table 1. The values of p_l and p_e for addition are given in (2.19) and (2.20), for subtraction with $r \leq -1$ in (2.21) and (2.22), and for subtraction with $-2^{-l} \leq r < -2^{-l-1}$ in (2.23) and (2.24). Note that these precisions refer only to the fractional bits of each data path. The number of integer bits required depends upon the range of values that must be represented. Details are described in another paper [9].

While the data path widths are chosen to meet the worst case constraints of both addition and subtraction, the values of p_l and p_e chosen depending upon which operation is being performed. In general, for more negative values of r, both $f_a(r)$ and $f_s(r)$ become flatter, allowing the use of fewer bits of r as inputs to the lookup tables. For $f_a(r)$, the values of p_l and p_e can be conveniently chosen using r_i alone, using the formulae (2.19) and (2.20). For $f_s(r)$ in the region $r < -1$, this is also the case, using (2.21) and (2.22). The function $f_s(r)$, as r approaches 0, becomes highly non-linear, requiring the use of very small intervals for linear approximation. The domain of $f_s(r)$ is therefore subdivided into intervals $-2^{-l} \leq r < -2^{-l-1}$, which become smaller as r approaches 0. For example, $l = 4$ corresponds to an r with fractional bits of the form $.11110r_{-6}r_{-7}\cdots$. The value of l can be produced using a priority encoder on the fractional bits of r, since it directly corresponds to the most significant zero in the fractional bits of the binary representation of r.

As r approaches zero, the distance over which linear approximation is accurate become smaller, requiring tables with finer granularity. Simultaneously, the size of each interval requiring a given granularity becomes smaller. As a result, a fixed size of table is required for each distinct value of l.

data path	precision
F_f	$F+7$
F_{lrl}	$F+7$
F_{cor}	$F+7$
F_{lcor}	$\left\lceil \frac{F}{2} \right\rceil + 2$

TABLE 1. Data Path Precisions

$$p_l \equiv \left\lfloor \frac{-F - r_i - 1}{2} \right\rfloor \quad (2.19)$$

$$p_e \equiv -F - r_i - 5 \quad (2.20)$$

$$p_l \equiv \left\lfloor \frac{-F - r_i - 3}{2} \right\rfloor \quad (2.21)$$

$$p_e \equiv -F - r_i - 6 \quad (2.22)$$

$$p_l \equiv \left\lfloor \frac{-F - 3 - 2l}{2} \right\rfloor \quad (2.23)$$

$$p_e \equiv -F - 5 - l \quad (2.24)$$

The log and exp tables each use simple arithmetic identities to reduce their size.

The log table uses the identity $\log(x) = \log(x \times 2^n) - n$, choosing n such that the input to the log table is in the range [1, 2). A priority encoder determines the value of n, and a shifter produces an input to the logarithm ROM in the desired range. The value of $-n$ is concatenated to the fraction output by the log table, which is in the range [0, 1), producing the desired result. This halves the size of the logarithm table.

Similarly, the exponential table uses the identity $\exp(x) = \exp(x-n) \times \exp(n)$. Only the fractional bits of *lcor* are input to the exponential table, and a shifter on the output performs the multiplication by $\exp(n)$.

The resulting tables required to implement the algorithm for various values of r_i are shown in Tables 2 and 3. Values of r_i below certain thresholds are "essential zeroes" [4], for which $f_a(r)$ and $f_s(r)$ are zero, and are handled with other logic. The values of r deemed to be essential zeroes are smaller here than in [4] due to our use of conservative mathematical error models.

According to Table 2, a total of 22K words of varying precisions are required, with a total of 660,480 bits. This is a substantial amount of memory, and further techniques are required to reduce it. First, it is experimentally observed that it is possible to reduce F_f below the value $F + 7$. The proof of the value of F_f is conservative, and experimentation reveals that F_f as small as $F + 3$ can be used without causing any errors across the entire domain of r. All possible values of r are tested to insure that there are no errors. This reduces the total table size for $f_a(r)$ and $f_s(r)$ to 578,560 bits, still a considerable number to integrate onto a single chip. Another technique is required to reduce this into an acceptable quantity.

2.2 Non Linear Table Compression

A final technique is used to reduce the size of each of the function tables. This is non-linear table compression. The preceding error analysis has already made each of the non-linear functions stored in ROM as loosely specified as possible without violating the accuracy constraints, so each of these functions must be represented exactly in order to guarantee the accuracy of the final result. This constraint prevents any approximations being used to represent these functions.

The technique used to represent these functions is the combination of a linear approximation and differential code. That is, to represent a function $f(x)$ exactly, x is split into two parts x_b and x_e according to (2.25), and the arithmetic identity expressed by (2.26) and (2.27) is used.

$$x = x_b + x_e \tag{2.25}$$

$$f(x) = f(x_b) + \frac{df(x_b)}{dx} \times x_e + fd(x) \tag{2.26}$$

$$fd(x) = f(x) - f(x_b) + \frac{df(x_b)}{dx} \times x_e \tag{2.27}$$

The essence of this technique is that a linear approximation of the function is used. The linear approximation may not produce the exact result, so a correction table $fd(x)$ is used. This table stores the difference between the actual function value and the linearly approximated value. Since the linear approximation is a good approximation over a small interval, the values in $fd(x)$ are small compared to $f(x)$. It is necessary to represent the value of $f(x_b)$ and $\frac{df(x_b)}{dx}$ for each possible value of x_b, and the value of $fd(x)$ for each possible value of x.

The equations (2.26) and (2.27) correspond to the hardware implementation shown in Figure 4. Given some $f(x)$ with N_x fractional bits of precision in x, the non-linear table compression divides x into two parts, x_e containing the N_e least significant bits of x, used for linear approximation, and x_b containing the N_b most significant bits, used to look up the function value and derivative, with $N_x = N_e + N_b$. The value of x_b is used to address a ROM. Each word contains the value of $f(x_b)$, the value of $\frac{df(x_b)}{dx}$, and $2^{N_e} - 1$ correction words $fd(x)$. A multiplexer using x_e to select one of the inputs produces the value of $fd(x)$. A multiplier and adder together perform the linear approximation. The correction word is then added in to find the value of $f(x)$.

ri	fa tables		fs tables	
	inputs	size	inputs	size
-1	$r_{-1} \cdots r_{-9}$	512	see table 2(b)	
-2	$r_{-1} \cdots r_{-9}$	512	$r_{-1} \cdots r_{-10}$	1024
-3	$r_{-1} \cdots r_{-8}$	256	$r_{-1} \cdots r_{-9}$	512
-4	$r_{-1} \cdots r_{-8}$	256	$r_{-1} \cdots r_{-9}$	512
.				
.				
.				
-19	r_{-1}	2	$r_{-1}r_{-2}$	4
-20	r_{-1}	2	$r_{-1}r_{-2}$	4
-21	(none)	1	r_{-1}	2
-22	(none)	1	r_{-1}	2
-23			(none)	1
-24			(none)	1

(a) Tables for $f_a(r)$ and $f_s(r)$

li	inputs	size
0	$r_{-2} \cdots r_{-11}$	1024
1	$r_{-3} \cdots r_{-12}$	1024
2	$r_{-4} \cdots r_{-13}$	1024
.		
.		
.		
7	$r_{-9} \cdots r_{-18}$	1024
8	$r_{-10} \cdots r_{-19}$	1024
9	$r_{-11} \cdots r_{-20}$	1024
10	$r_{-12} \cdots r_{-20}$	512
.		
.		
.		
18	r_{-20}	2
19	(none)	1
20	(none)	1

(b) Tables for $f_s(r), -1 \le r < 0$

table	size
log	2048
exp	4096

(c) Log and Exp Tables

TABLE 2. Table Sizes for Linear Extrapolation

Optimization of this design requires calculating the number of bits of ROM. Let W_f be the maximum number of bits required to represent $f(x_b)$, W_d be the maximum number of bits required to represent $\frac{df(x_b)}{dx_b}$, and W_c be the number of bits required to represent the correction function $fd(x)$. The total width of the ROM is W, where W is given by (2.28).

$$W = W_f + W_d + (2^{N_e} - 1) \times W_c \tag{2.28}$$

The original uncompressed table requires $2^{N_e} \times W_x$ bits, and the compressed table requires $2^{N_b} \times W$ bits.

Minimization of table size requires selecting a particular value of N_e, which implies N_b. Ideally, for a given $f(x)$, one could construct a mathematical expression defining the total table size. This is impractical, since the function stored in the table is not an analytical function, but is an approximation of it after rounding to some precision. This makes the mathematics involved difficult.

Since each of the tables is relatively small, the simplest technique is to generate all of the tables for all values of N_e. It is then possible to select the table that contains the fewest bits.

An optimization program was written to find the optimal value of W that minimized total table size. Each table is allowed to use a different value of N_e, with the optimal values shown in Table 3.

A subtle point here is that the number of words in Table 3 is not exactly the number of words given in Table 2 divided by 2^{N_e}. The reason for this is some of the tables in Table 2 contain fewer than 2^{N_e} entries, and must be rounded up to a word in the compressed table.

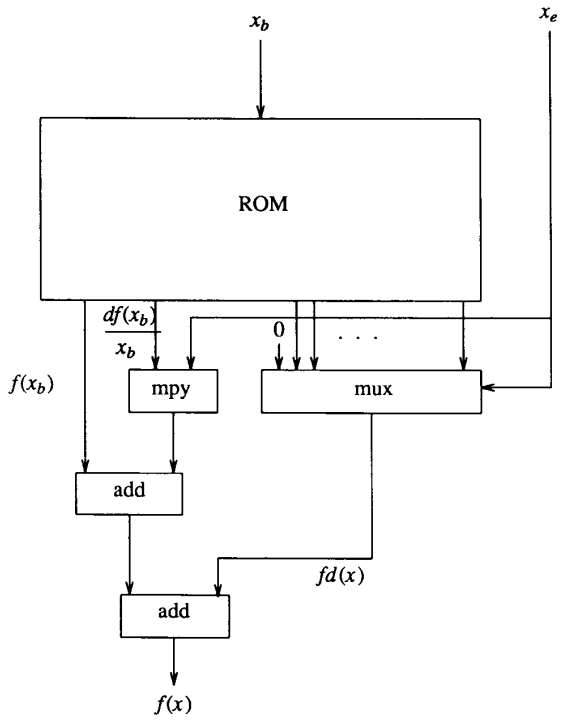

Figure 4. Non-Linear Function Compression

It is not possible to state in advance that the best implementation in terms of chip area is obtained by implementing each table in a unique ROM. In particular, the various f_a and f_s tables are candidates for merging into a single ROM, since only one of the tables will be accessed for each computation. It is therefore desirable to find other values of N_e, which, although sub-optimal, produce a word width that is close enough to other tables' word width that they can be implemented in a single ROM without wasting excessive space. Reasonable values for these tables are also shown in table 3.

Various combinations of merging tables into a single ROM are possible. As an example, the optimal f_a and f_s tables total 184,144 bits, while merging total cost of merging all of the ROMs into a single table with W=92 requires 201,388 bits, a 9.3% increase. This 9.3% increase must be compared to the overhead associated with implementing each table in its own ROM.

		optimal					
Name	Function	N_e	W_f	W_d	W_c	W	words
fa	$f_a(r)$	3	23	13	7	85	258
fsb	$f_s(r), r < -1$	3	23	13	7	85	386
fss1	$f_s(r), -1 < r,l < 10$	4	27	14	9	176	640
fss2	$f_s(r), -1 < r,l \geq 10$	3	28	22	11	127	132
log	$\log(x)$	4	23	12	5	110	128
exp	$\exp(x)$	4	24	13	7	142	256

(a) Optimal values of W

		actual values of W					
Name	Function	N_e	W_f	W_d	W_c	W	words
fa	$f_a(r)$	3	23	13	7	85	258
fsb	$f_s(r), r < -1$	3	23	13	7	85	386
fss1	$f_s(r), -1 < r,l < 10$	3	27	14	7	90	1280
fss2	$f_s(r), -1 < r,l \geq 10$	2	28	22	14	92	265
log	$\log(x)$	4	23	12	5	110	128
exp	$\exp(x)$	4	24	13	7	142	256

(b) Actual Values of W

TABLE 3. Compressed Table Sizes

In order to optimize chip area, an analytical expression for ROM area can be constructed. Figure 5 shows a representation of a ROM layout on an integrated circuit. Assume that a ROM contains a rectangular array of bit cells of height b_h and width b_w. If the ROM is approximately square, with a border on the left side of width d_w for the wordline drivers, and a border of width d_s on the bottom for sense amplifiers, then the total ROM area of an n bit ROM is

$$Area(n) = n \times b_h \times b_w + \sqrt{n} \times (b_h \times d_w + b_w \times d_s) + d_w \times d_s \quad (2.29)$$

Using this formula with parameters determined from actual circuit design, the overhead associated with each ROM is so large that there is a strong incentive to merge ROMs. In the final implementation, all of the f_a and f_s tables were merged into a single ROM of 2189 words by 90 bits. A small part of the fss2 table will not fit in this width, so a separate 265 word by 2 bit ROM, called the excess ROM, is used to store the extra information. This ROM is also altered slightly to simplify the implementation, as will be described in the next section. A summary of the architectural model of the ROMs is shown in Table 4.

3. Detailed Chip Design

A two chip design based upon the algorithms presented here is under way. This design is still in progress, so the description of it in this section is brief.

The chip is being designed in a conventional 2 level metal 3 micron P-well CMOS process. The design envisaged could be implemented in a die 10mm×10mm. The multi-project chip fabrication service available to us has a fixed maximum die size of 8.2mm per

side, so our implementation is using two chips. One of the chips contains the first several stages of logic and the log rom and F-rom, and is very dense. The second chip contains the remaining roms and stages, and occupies only a small fraction of the available area.

The total area of the processor is still well within the amount that can be fabricated on a single chip with acceptable yield, so this decision is purely due to the constraints of the chip manufacturer.

The interface provided by the chip is a pipelined processor, accepting two operands and producing one result per clock cycle. The clock cycle is conservatively targeted at 150ns, and circuit simulations to date suggest that this will be met. The only fundamental constraint on cycle time in a pipelined implementation is ROM access time, which is expected to be 100ns in 3 micron CMOS.

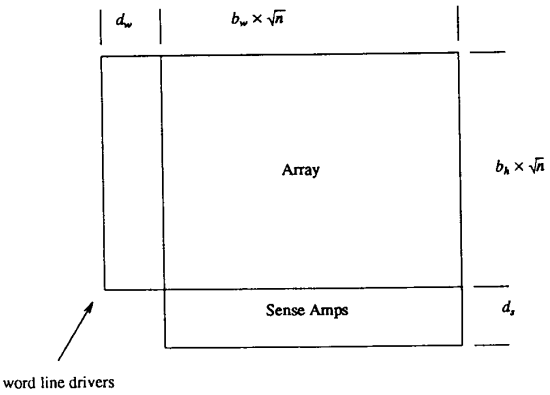

Figure 5. ROM Area Model

ROM	Words	Width	Total Bits
f	2189	90	197010
excess	265	2	530
log	128	110	14080
exp	256	142	36352
Total			247972

TABLE 4. ROM Sizes

The detailed logic design has been completed and simulated using a logic simulator. The logic simulator is integrated into a software driver that performs the arithmetic calculation and compares the result of simulating the logic design to the correct result. This enables large numbers of test cases to be run with little effort. Logic simulations of several thousand interesting cases have been performed without any errors.

The optimal ROM sizes presented above require complicated logic to map r into a ROM address. The complexity of the logic can be reduced by placing each table at an address in the ROM that simplifies the mapping function. This leaves "holes" in the address space of the ROM, but no additional area is consumed, since the corresponding words in the ROM are not implemented. This requires a word line decoder that decodes only those addresses that are implemented.

The ROMs also use a multi-level decoder tree that requires that each contiguous set of addresses be a multiple of 8 words. This slightly increases the size of the ROMs. Another factor that increases the size of the ROMs is the desire to have an approximately square circuit layout for large ROMs. This also requires rounding up the number of words in each ROM to an integral power of two multiple of some number that leads to a square layout. The resulting ROMs, as actually laid out, are described in Table 5. The excess ROM is so small that a layout with a large aspect ratio is tolerable. A total of 260K bits of ROM are actually present in the chip. Almost all of the f_a and f_s tables are grouped into a single ROM, called the f-ROM. A small additional ROM is used to store some bits of fss2 that will not fit in the f-ROM. Although the f-ROM is 90 bits wide, and the fss2 table only requires 92 bits, the excess ROM is organized as 265 words by 9 bits. An access to fss2 therefore produces 99 bits, more than apparently required. These extra bits are used because they reduce the amount of data steering logic required for selecting different widths of W_c and W_d. It is highly desirable to have each field for the various tables located at the same position in the output word of the ROM, a property that can only be achieved by using the larger excess ROM. In our implementation, the f-ROM stores 27 bits of W_f and 14 bits of W_d for each word, together with a 49 bit field that is interpreted as multiple values of W_d. This is not adequate for fss2, so one bit from the excess ROM is used for the larger value of W_f of the fss2 table, and eight additional bits are used for the larger value of W_d. The only data steering logic required is used to interpret 49 bits in the f-ROM either as seven offsets of seven bits, for fa, fsb, and fss1, or as three offsets of 14 bits for fss2. This leads to reduced implementation complexity at the cost of a small number of extra bits of ROM.

An unfortunate side effect of the complicated algorithms used by this processor is the large delay through the processor. A total of 10 pipeline stages are used in the present implementation. It is possible that this could be reduced by more careful allocation of functionality to each pipe stage. A faster implementation could also be achieved by not using the non-linear compression, at the cost of larger ROMs.

ROM	Words	Width	Bits
f	296	720	213120
excess	136	18	2448
log	64	284	14080
exp	128	220	36352
Total			266000

TABLE 5. ROM Implementations

4. Conclusions

This paper has described the algorithms used in an integrated processor for 30 bit logarithmic arithmetic. Two techniques are used to make this device possible. Linear approximation of the functions required is shown to be simple due to the particular functions involved. The linear approximation is performed using parameters that are chosen to meet the desired accuracy. Subsequent non-linear compression of each lookup table leads to a further reduction in table size. The non-linear functions are compressed using linear approximation, plus a correction function that results in exact implementation of the desired function. The result is that a factor of 275 reduction in table size is achieved, compared to previous techniques. The disadvantage of the techniques presented here is that they increase the

delay of the processor considerably, although the possibility improving the speed of the implementation by better pipeline partitioning has yet to be explored.

5. References

[1] J. N. Mitchell Jr, "Computer Multiplication and Division Using Binary Logarithms", in *IRE Trans. Electron. Comput.* Aug. 1962, pp 512-517

[2] E.E. Swartzlander and A. G. Alexopolous, "The Signed Logarithm Number System", in *IEEE Trans. Comput.*, Dec. 1975, pp 1238-1242

[3] J. H. Lang , C. A. Zukowski, R. O. LaMaire, and C. H. An, "Integrated-circuit Logarithmic Units", in *IEEE Trans. Comput.*, May, 1985, pp 475-483

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

[5] 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

[6] D. Marino, "New Algorithm for the Approximate Evaluation in Hardware of Binary Logarithms and Elementary Functions", in *IEEE Trans. Comp.*, Dec 1972, pp 1416-1421

[7] 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

[8] F. J. Taylor, "An Extended Precision Logarithmic Number System", in *IEEE Trans. Acoust., Speech, Signal Processing*, Jan. 1983, pp 232-234

[9] D. M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System", to appear in *IEEE Trans. on Comput.*