# A Fast Hardware Approach for Approximate, Efficient Logarithm and Antilogarithm Computations

Suganth Paul, Nikhil Jayakumar, and Sunil P. Khatri

***Abstract*—The realization of functions such as log() and antilog() in hardware is of considerable relevance, due to their importance in several computing applications. In this paper, we present an approach to compute log() and antilog() in hardware. Our approach is based on a table lookup, followed by an interpolation step. The interpolation step is implemented in combinational logic, in a field-programmable gate array (FPGA), resulting in an area-efficient, fast design. The novelty of our approach lies in the fact that we perform interpolation efficiently, without the need to perform multiplication or division, and our method performs both the log() and antilog() operation using the same hardware architecture. We compare our work with existing methods, and show that our approach results in significantly lower memory resource utilization, for the same approximation errors. Also our method scales very well with an increase in the required accuracy, compared to existing techniques.**

***Index Terms*—Field-programmable gate arrays (FPGAs), floating point arithmetic, logarithmic arithmetic, VLSI.**

## I. Introduction

THE generation of elementary functions such as log() and antilog() finds uses in many areas such as digital signal processing (DSP), 3-D computer graphics, scientific computing, artificial neural networks, logarithmic number systems (LNS), and other multimedia applications [1]. Our approach provides a good solution for field-programmable gate array (FPGA)-based applications that require high accuracy with a low cost in terms of required lookup table (LUT) size. Such applications include LNS, DSP cores, etc. In fact the fast generation of these functions is critical to performance in many of these applications. Using software algorithms to generate these elementary functions [2], [3] is often not fast enough as stated in [1]. Hence, the use of dedicated hardware to compute log() and antilog() is of great value. Over the past few decades, many authors have proposed various hardware approaches to approximate these elementary functions in an area-efficient manner, while maintaining high speed and accuracy.

Two methods that are well researched and used for the generation of the logarithm function are digit-recurrence algorithms and LUT-based approaches. Out of these methods the digit-recurrence methods are efficient from an area and accuracy perspective, but have longer latencies and convergence problems [1], [4], [5]. The LUT-based methods are widely used to approximate the logarithm and antilogarithm functions. Some of the previous works involving LUT-based methods include, LUTs combined with polynomial approximations [3], [6], symmetric bipartite table-based approximations [7], [8] etc. The main objective of all these works is to utilize minimum circuit area while retaining the accuracy of the approximation. The main idea of our approach is to use LUTs along with linear or quadratic interpolation and approximates the multiplication required for interpolation using approximate log() and antilog() functions while computing a more accurate log() and antilog(). We show that the most cost effective implementation is a LUT with a linear interpolation, implemented in a manner that optimizes the area and delay while providing good accuracy. We apply our method to generate the logarithm of a number and also show that a similar methodology can be used to generate the antilogarithm of a number.

In this paper, the number format used is similar to the IEEE 754 single-precision floating point format that has 32 bits. The leading bit is the sign bit, followed by an 8-bit exponent $E$ and a 23 bit mantissa $M$. The value of a number, $X$ represented in this format is given by

$$X = \pm 1.m * 2^{E-127}, \quad 0 \leq m < 1. \tag{1}$$

We use a similar number format representation, but assume the number of bits in the mantissa to be variable. We also assume that the number is positive since the logarithm of a negative number does not exist. We target 10 or more bits of accuracy in this work.

The remainder of this paper is organized as follows. Some previous work in this area is described in Section II. Section III elucidates our approach to efficiently find the logarithm of a number through linear interpolation, and also provides an error analysis of the approximation. Section IV shows how the antilogarithm of a number is computed through linear interpolation. Section V presents an improvement to the linear interpolation approach. Section VI investigates the computation of logarithm through quadratic interpolation. A summary of the various approaches explored to approximate the logarithm is given in Section VII. We also present some estimates on the area and delay of the linear interpolation approach, and compare it with other relevant works in this section. We conclude in Section VIII.

## II. Previous Work

One of the earliest approaches to approximate the binary logarithm of a number was given by Mitchell [9]. In his method, the

Manuscript received July 08, 2007; revised October 21, 2007 and January 27, 2008. First published December 02, 2008; current version published January 14, 2009.

S. Paul and S. P. Khatri are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station TX 77843 USA (e-mail: suganthpaul@gmail.com; sunilkhatri@tamu.edu).

N. Jayakumar is with Texas Instruments, Dallas, TX 75243 USA (e-mail: nikhil.jayakumar@ti.com).

Digital Object Identifier 10.1109/TVLSI.2008.2003481

1063-8210/$25.00 © 2008 IEEE

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

logarithm of a number is found by attaching the mantissa part of the number to the exponent part. This method is extremely easy to implement but gives an absolute error as high as 0.086 which is only 3.5 bits of accuracy. There are various authors who have reduced this error by using error correction techniques implemented by simple logic gates without involving multiplications or divisions [10]–[13]. Although these methods are better than Mitchell's approach, they all give less than seven accurate bits (which might be adequate for some applications). Compared to these methods, our approach is applicable for applications that require a much higher accuracy. [14] gives an approach for logarithmic multiplication, but they provide results based on random input vectors and compute the average error. We find the worst case error over all possible inputs.

In recent times, most elementary functions are generated using LUTs. This approach, initially proposed by Brubaker in [15], involves the computation of a function using a single LUT. The accuracy of the approach depends solely on the size of the LUT used. Another method given in [16] improves Brubaker's method by concatenating the lower order bits of the mantissa to the value looked up from the table. However, this gives only a small error improvement of one bit. Kmetz [17] proposes to store the error that occurs due to the Mitchell approximation in the table and add it to the mantissa of the number. This results in a further improvement in error (by 3 bits) with the overhead of an add operation. In our approach, we follow the same scheme of storing the error values of the Mitchell approximation in a table. Other methods like [7] and [8] make use of bigger LUTs to give more accurate results, with a good speed of computation. In our approach, we try to find the optimum table size to use for a required accuracy.

Apart from these simple methods, there are several other complex methods for the generation of these elementary functions. References [3] and [6] use a LUT combined with a polynomial approximation to interpolate the function between many small intervals. Reference [3] also presents a trade off between the table sizes and the degree of the polynomial to use while interpolating. The problem with these methods is that there are multiplications and divisions involved in the computation. *Our approach focuses on using a smaller size table along with a simple linear function to interpolate between the table values. The main contribution of this work is to approximate the multiplication required for the interpolation by a method that is fast and results in a small error.* This also allows us to pipeline the implementation and achieve a higher throughput. The multipliers in our target FPGA are restricted to a speed of 135 MHz, and by avoiding their use, we can achieve much greater speeds, as our results indicate. There are many papers on LNS [18]–[24]. LNS systems also require the computation of log() quantities while finding the approximate value of some functions like addition and subtraction. The papers [18], [20], and [21] use multipliers to compute the log(). Reference [23] uses only ROMs to compute the log() without any interpolation. The paper [22] use ROMs and linear tangent and/or secant interpolation. We are different from [19] and [22] in that we explore linear least squares based interpolation which gives us one extra bit of accuracy over our own linear secant implementation. Also our method lends itself elegantly to perform antilog() operation using the same hardware architecture and accuracy as log(), whereas [19], [22] do not talk about antilog() computation using the same hardware architecture as the log() computation.

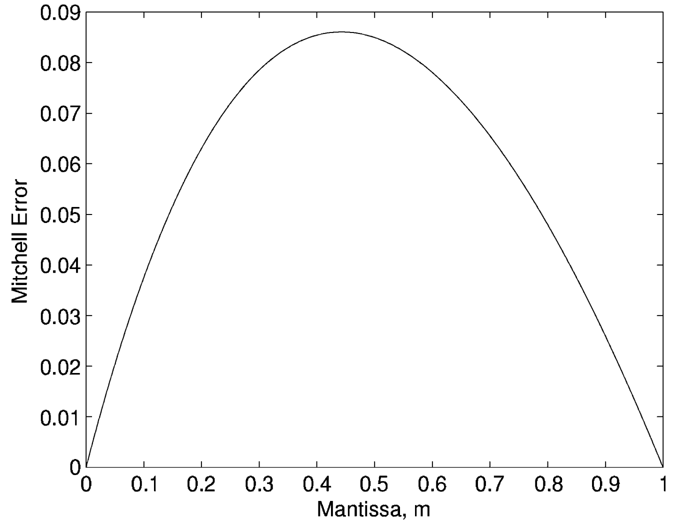


Fig. 1. Error due to Mitchell approximation.

Reference [25] presents an analysis of the errors and hardware implementation costs of some of the LUT-based methods discussed above. In our paper, we show that a good improvement in error performance can be achieved with the use of a small LUT and some additional hardware. We compare our results with those presented in [25].

## III. Our Approach

This work uses a LUT-based approach combined with a linear interpolation to generate the logarithm of a number. The multiplication required in this linear interpolation is avoided, resulting in an area and delay reduction. The idea is described in the following sections.

### A. Interpolation Approach

Mitchell's approximation [9] is given by the following equation:

$$\log_2(1+m) \simeq m, \quad 0 \leq m < 1. \tag{2}$$

The error due to this approximation is given by

$$E_M = \log_2(1+m) - m, \quad 0 \leq m < 1. \tag{3}$$

The error curve shown in Fig. 1 is sampled at $2^t$ points (depending on the size of the LUT required). These values are rounded depending on the width of the word required and stored in the LUT. The LUT is addressed by the first $t$ bits of the mantissa portion of the number. Now we investigate the option of interpolating between the values stored in the table. This is done by the following equation:

$$\log_2(1+m) \simeq m + a + \frac{(b-a)\cdot n_1}{2^{k-t}}, \quad 0 \leq m < 1. \tag{4}$$

Here $m$ is the mantissa part, $a$ is the error value from the table accessed by the first $t$ bits and $b$ is the next table value adjacent

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

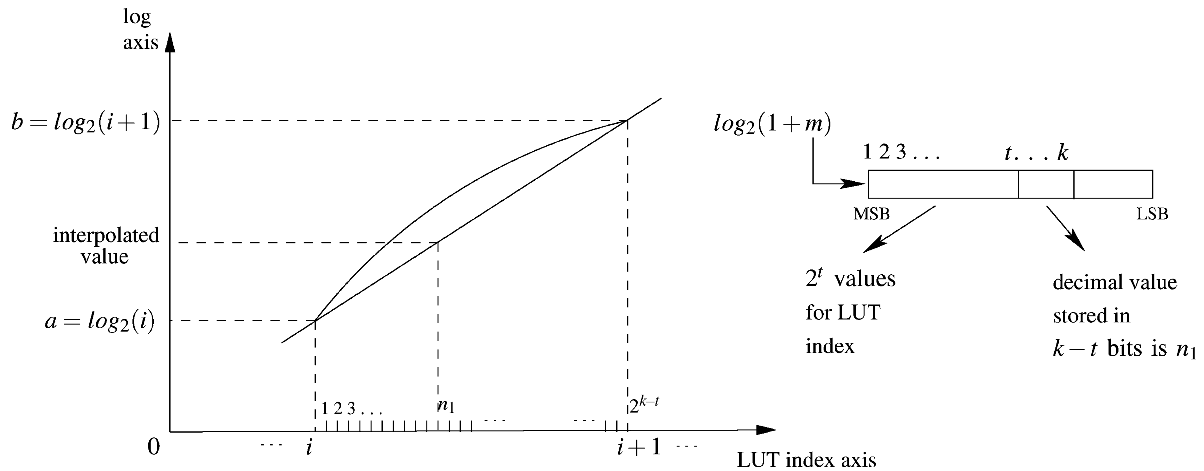


Fig. 2. Interpolation to find the log().

to $a$. Also $k$ is the total number of MSBs in the mantissa used for the interpolation and $n_1$ is the decimal value of the last $k-t$ bits of the mantissa. Essentially we find an error value from the table based on the first $t$ bits and interpolate between this value and the next value based on the remaining bits. This is illustrated in Fig. 2.

The third term in (4) requires a multiplication. In order to circumvent the multiplication (which is expensive in terms of area and delay), we investigate the option of interpolating repeatedly between any two adjacent values stored in the table. This is done using Algorithm 1.

**Algorithm 1** Recursive Bi-partitioning

STEP 1: The first $t$ bits of the mantissa address the table to obtain the stored *left* value $a$ and the adjacent *right* value $b$

STEP 2: Bisect the two values obtained in the previous step and find the *middle* value

**if** the next bit in the mantissa is a 1 **then**

    Keep the *middle* and *right* values

**else**

    Keep the *left* and *middle* values

**end if**

**if** the last bit of the mantissa is not reached **then**

    Goto STEP 2

**else**

    Choose the *left* or *right* value based on, if the last bit is a 0 or 1 respectively

**end if**

The error performance of Algorithm 1 is shown in Fig. 3. The maximum error is $4.35\times10^{-5}$. This gives us 14 bits of accuracy. The only problem with this approach is that there are too many steps involved as all the mantissa bits are considered. Trying another approach, we investigate the case where a limited number of interpolations are done. We tabulate the maximum error incurred when the previous algorithm is implemented for $t$, $t+1$, and so on until $t+8$ mantissa bits and ignoring the rest. This is the same as doing different levels of interpolation from 0 to 8. The maximum error for this approach is shown in Table I. In this case, the size of the LUT used is 64 words and the width of each word is 16 bits. The width of each word in the table is chosen in such a way that the accuracy is not reduced due to rounding.

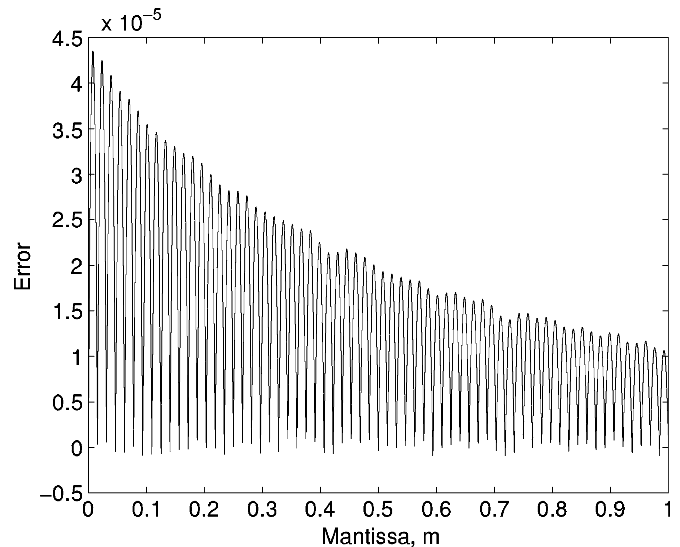


Fig. 3. Error performance of Algorithm 1.

From Table I, we see that 1 or 2 interpolations are not enough to give a good error performance. Reasonable accuracy is obtained for either 7 or 8 bits, but this requires as many interpolations, and therefore results in larger delays in computing the logarithm. In order to obtain better accuracy, we need to implement the multiplication of $(b-a)$ and $n_1$. However, implementing multiplication is expensive in terms of area and delay. Therefore, we approximate the multiplication, so as to obtain good error performance as well as low delay and area utilization. We will show in the following section(s) that our approach

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

TABLE I
MAXIMUM ERROR, FOR A LIMITED INTERPOLATION APPROACH

| # of Interpolations | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Maximum Error $\times$ $10^{-3}$ | 3.4147 | 1.7182 | 0.8834 | 0.4640 | 0.2538 | 0.1486 | 0.0959 | 0.0693 | 0.0564 |
| Accurate Bits | 8.19 | 9.19 | 10.14 | 11.07 | 11.94 | 12.72 | 13.35 | 13.82 | 14.11 |

TABLE II
MAXIMUM ERROR $\times 10^{-3}$, FOR VARIOUS APPROACHES

| | Number of Words in the Table | | | | | |
|---|---|---|---|---|---|---|
| Method | 64 | | 128 | | 256 | |
| | Max Error | Bits | Max Error | Bits | Max Error | Bits |
| 1.a),2.a) | 2.8579 | 11.77 | 1.2720 | 12.94 | 0.6228 | 13.97 |
| 1.a),2.b) | 3.4490 | 11.5 | 1.6753 | 12.54 | 0.0843 | 13.53 |
| 1.b),2.a) | 3.5760 | 11.45 | 1.7384 | 12.49 | 0.8467 | 13.52 |
| 1.b),2.b) | 0.7320 | 13.74 | 0.2142 | 15.51 | 0.0687 | 17.15 |

gives similar error performance as the 7 bit interpolation, however with lower delay.

### B. More Effective Approach

In this section, we propose a more efficient approach to do interpolation without the multiplication of $(b-a)$ and $n_1$ in (4). The essential idea is that the multiplication of $(b-a)$ and $n_1$ is simplified by taking the antilogarithm of the sum of the logarithm of $(b-a)$ and the logarithm of $n_1$. In order to perform this operation with a small delay, we consider the following options.

1) $\log_2(b-a)$ may be approximated by either of the following two options:
   a) Mitchell approximation;
   b) LUT: For the LUT option the constants $\log_2(b-a)$ for each of the intervals is stored in the original LUT. Recall that we stored the error values obtained by using a Mitchel approximation for the log function. The $\log_2(b-a)$ values are stored along with the error values and are indexed by the same address lines.
2) Antilog of $(\log_2(n_1)+\log_2(b-a))$ may be approximated by either of the following two options:
   a) Mitchell approximation;
   b) LUT: To obtain the antilogarithm of a number by this method, we need to construct another LUT. The error due to the Mitchell approximation of the antilogarithm function is stored in this LUT as shown in Section IV. This antilog LUT is utilized to compute the antilogarithm of a number, since the multiplication of $(b-a)$ and $n_1$ is performed by taking the antilogarithm of the sum of the logarithm of $(b-a)$ and the logarithm of $n_1$.

The maximum error in the logarithm of a number incurred by using each of these options along with the number of accurate bits in the result is shown in Table II.

From Table II we see that the combination of 1b) and 2b) has the best error performance. Therefore, to perform the interpolation described in (4), it makes sense to find the antilogarithm using a LUT. The additional advantage of this is that the same LUT can be used while computing the antilogarithm of a number as well. Table II also shows that our approach allows scalability of the system, by making a tradeoff between the accuracy and the number of values stored in the table.

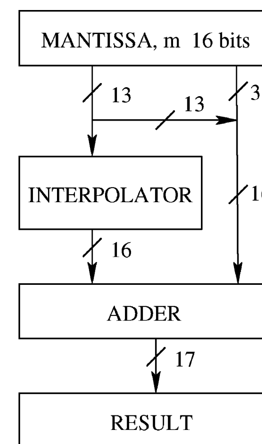


Fig. 4. Block diagram of the log engine.

### C. Architecture of Implementation

Fig. 4 shows the block diagram of the log() engine which is essentially an implementation of (4). The architecture of the interpolator block is shown in Fig. 5. The exponent part of the input number trivially becomes the decimal part of the logarithm. This is because we assume that $0 \leq m \leq 1$ in (1). Also, we assume that the number we operate on (which is expressed as in (1) is positive. Here we only show the operations on the mantissa, $m$. The implementation is pipelined, with 12 stages in the pipeline. We use a 16-bit mantissa and a $2^6$ by 16 bit LUT as an example. The width of each word in the table is chosen as 16 bits so that the error due to rounding does not dominate the overall error. Rounding of a number, $x$ to $t$ bits is done as

$$\mathrm{round}(x,t) = \lfloor x2^t + 0.5 \rfloor 2^{-t}. \tag{5}$$

Recall that the accuracy, from Table II, was 13.74 bits in this case. One of the adders is a three input fixed point adder as shown in Fig. 5. It is implemented as 2 adders. The width of the mantissa bits processed by each block in the architecture is shown in the diagrams. Since $(b-a)$ takes both negative and positive values for $0 < m < 1$, the $\log_2(b-a)$ values stored in the lookup tables are actually the logarithm of the absolute values of $(b-a)$. It is found that $(b-a)$ changes sign from

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

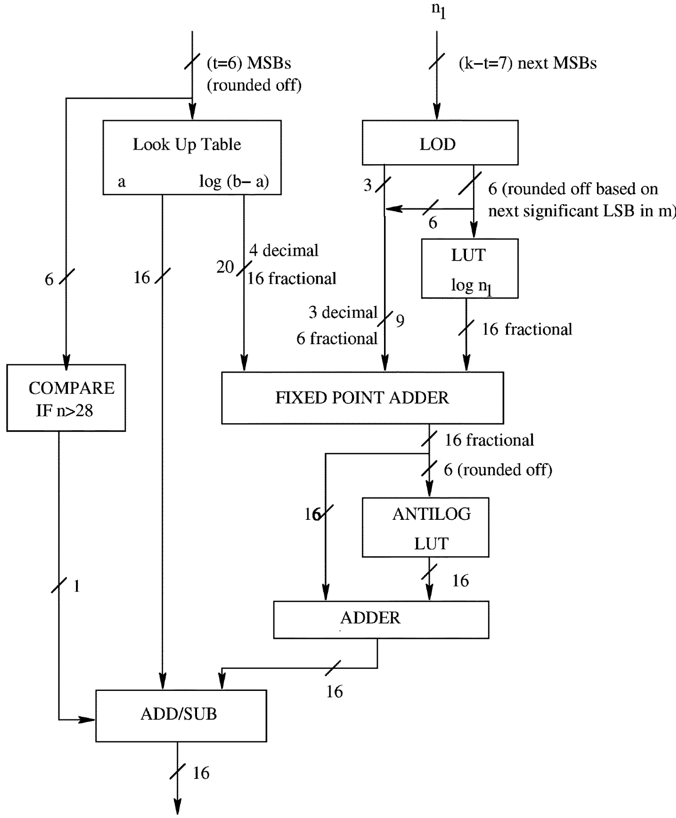


Fig. 5. Architecture of the interpolator for $k = 13, t = 6$.

positive to negative for $m > 0.4427$. This is equivalent to comparing the decimal value of the first six bits of the mantissa with 28, as shown in Fig. 5. Hence, if the first six bits have a value greater than 28, the comparator block sends a control signal to the ADD/SUB block instructing it to perform a subtract operation. The leading one detector (LOD) block is used to find the $\log_2(n_1)$. The LOD detects the first bit that has a value 1. It then uses the remaining bits as the mantissa portion to access the lookup table. Since there are 7 bits given as input to the LOD in this example, the LOD finds the first position that has a value 1 and the remaining bits which can be as wide as 6 bits is used to access the LUT. The decimal part of $\log_2(n_1)$ which is indicated by the position of the first bit with value 1, is directly sent to the three input fixed point adder to be added to the decimal part of $\log_2(b-a)$. The output of the adder after the antilog stage has to be shifted to the right or left depending on the decimal output of the fixed point adder. Also there is a $2^7$ term in the denominator of (4) and this is accounted for by a constant right shift of 7 bits at the output of the adder of the antilog LUT. In Fig. 5, the blocks representing a LUT for $\log_2(n_1)$ and a LUT for $a$ are identical and can be implemented using a dual port RAM. Also note that, the address of each of the three LUTs shown in Fig. 5 has a width of 6 bits. In each case the 6 bit address word is obtained by rounding off the value of the next LSB in the word from which the address word is derived. The round() operation is implemented using an adder as shown in (5). All the quantities that are rounded off in this manner are annotated as such in Fig. 5.

## D. Error Analysis

The expression for error due to a Mitchell approximation of the logarithm is given by (3). As mentioned before, the error curve due to the Mitchel approximation is sampled at various points, and these samples are stored in the lookup table. It is observed that while interpolating between any two adjacent values in this table, the maximum error is bound to occur when the difference between these two adjacent error values is largest. The largest difference in error values occurs for the first pair of points in the lookup table. The size of this largest difference (assuming a table of size $2^6$ and width 16) is given by

$$S_{\max} = \log_2(1+2^{-6})-2^{-6}-\log_2(1+0)-0 = 6.7428\times10^{-3}. \tag{6}$$

The expression for error due to our approximation is given from (4) as

$$E = \log_2(1+m) - m - a - \frac{(b-a)\cdot n_1}{2^{k-t}}. \tag{7}$$

There are three sources of error in our method. An error upper bound is obtained by adding the maximum errors due to all these sources. In other words

$$E_{\text{tot}} = E_{\text{round}} + E_{\text{int}} + E_{\text{anti}} \tag{8}$$

where $E_{\text{round}}$ is the error due to rounding of bits stored in the lookup table, $E_{\text{int}}$ is the error due to interpolation, and $E_{\text{anti}}$ is the error incurred due to the use of the antilogarithm function during interpolation.

The input number or mantissa is split into three different ranges in terms of bit width, in order to analyze the error. For example, a mantissa of bit width 5 implies that the remaining LSBs take on a value of 0.

*Case 1—Mantissa Width of 0–6 bits:* Here the error is due to rounding alone as the error values for all 6-bit numbers have already been stored in the lookup table. For a table in which each word is 16 bits wide, the rounding error is given by $2^{-17}$.

*Case 2—Mantissa Width of 7 to 13 bits:* The error has contributions from all the three error sources mentioned above. Interpolation is done by using (9). In this case, the first value of the interval is $a = 0$ and the last value is $b = S_{max}$. Also the first 6 bits of the mantissa are zeros since we are in the first of $2^6$ intervals, and the value of $n_1$ is given by the decimal value of the next 7 bits. This can be expressed as $n_1 = m\cdot 2^{13}$. The interpolation error term is given by

$$\text{Interpolation term} = a + \frac{(b-a)\cdot n_1}{2^7}. \tag{9}$$

Substituting $b$, $a$, and $n_1$ in (9), we get the interpolation term in the first interval as

$$\text{Interpolation term} = S_{\max}(m\cdot 2^{13})/2^7. \tag{10}$$

If the error due to antilog is assumed to be zero, we can find that the error expression due to interpolation by using the following equation:

$$E_{\text{int}} = \log_2(1+m) - m - S_{\max}\cdot(m\cdot 2^6). \tag{11}$$

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

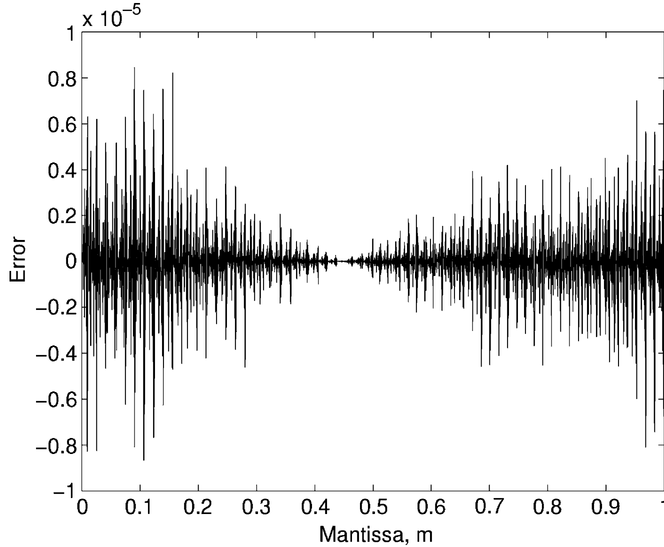


Fig. 6. Antilog error during interpolation.

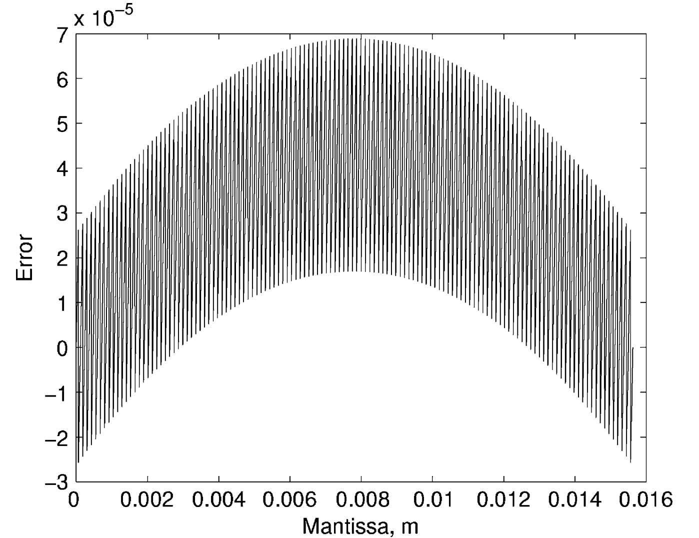


Fig. 7. Interpolation error for Case 3.

The maximum error is found by differentiating this equation and setting it to zero. We get the maximum error as $E_{\text{int}}^{\max} = 4.33 \times 10^{-5}$ at $m = 7.7929 \times 10^{-3}$.

The antilog error depends on the values of $\log_2(b-a)$ which is stored in the table and $\log_2(n_1)$. We find the error due to antilog by simulating the lookup table-based antilog approximation for these particular values and find the maximum antilog error to be $8.4433 \times 10^{-6}$.

The results of the simulation are shown in Fig. 6 for all possible values of $m$ ranging from 0 to 1.

*Case 3—Mantissa Width Greater Than 13 bits:* The rounding and antilog errors are the same as above. As for the interpolation error, we proceed in a fashion similar to Case 2. Here the value of $n_1$ is given by $n_1 = \text{round}(m \cdot 2^{13})$, where the $\text{round}()$ function represents a round-off to the closest integer given by (5). The error function for this case is given by

$$E_{\text{int}} = \log_2(1+m) - m - \frac{S_{\max} \cdot \text{round}(m \cdot 2^{13})}{2^7}. \quad (12)$$

Plotting this expression from $m = 0$ to $2^{-6}$ in Fig. 7, we find the maximum error as $6.9276 \times 10^{-5}$. Out of all the previous cases, the third case has the worst error due to interpolation, while the errors due to rounding and antilog approximation remain the same. Hence, the error bound, given by plugging in the maximum values of each of the error components in (8) is $E_{\text{tot}} = 8.5348 \times 10^{-5}$.

## IV. Antilogarithm Computation

The Mitchell approximation to find the binary antilogarithm of a number, $m$ is given by the expression

$$2^m \simeq 1 + m, \quad 0 \leq m < 1. \quad (13)$$

The error due to this expression is given by

$$E_M = 2^m - (1+m), 0 \leq m < 1. \quad (14)$$

We sample the error curve at as many points as required, and store these values in the first entry of a LUT. If $c, d$ are any two adjacent values stored in the memory, we store $\log_2(c-d)$ in the second entry of the LUT. The architecture used for the implementation of the antilog function is very similar to the logarithm implementation except for these differences. In Fig. 4, instead of the adder block we use a subtracter as the Mitchell approximation is always higher than the actual antilog value. Also the polarity of $c - d$ changes from positive to negative when the mantissa $m > 0.5288$. Hence while using a table with 6 address bits (i.e., 64 values), the comparison done in Fig. 5 will be to check if the decimal value of the first six bits of the mantissa is greater than 34. The antilogarithm computation was simulated, and the worst case errors and the accurate number of bits are shown in Table III for different table sizes.

TABLE III
Worst Case Antilog Error

| Table Size | 64 | 128 | 256 |
|---|---|---|---|
| Maximum Error $\times 10^{-5}$ | 5.4806 | 1.4214 | 0.3983 |
| Accurate Bits | 14.16 | 16.10 | 17.94 |

## V. Interpolation Using a Least Square-Based Linear Polynomial

An improvement can be made on the linear interpolation-based logarithm and antilogarithm computation with no additional increase in hardware. In this method, the error curve in Fig. 1 is split up into many smaller segments. There are as many segments as the number of entries in the LUT. Assume that the number of entries in the LUT is $2^t$. Now each of these $2^t$ segments is fitted using a first-order polynomial in a least square sense. The coefficients of these $2^t$ polynomials are stored in the LUT. The logarithm is now approximated as

$$\log_2(1+m) \simeq m + a \cdot n_1 + b, \quad 0 \leq m < 1. \quad (15)$$

Here $a, b$ are the coefficients of the least square polynomial. If $k$ is the total number of bits in the mantissa, the first $t$ bits are used to access the LUT. $n_1$ is the value represented by the last $k-t$ bits of the mantissa. We store the values $\log_2(a)$ and $b$ in two LUTs and the multiplication $a$ times $n_1$ is implemented as before using LUTs for logarithm and antilogarithm. While

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

TABLE IV
LOG AND ANTILOG ACCURACY FOR LEAST SQUARE-BASED LINEAR INTERPOLATION

| No. of words in LUT | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|
| Width of LUT word, bits | 12 | 14 | 16 | 18 | 20 | 22 |
| Log Accuracy, bits | 10.73 | 12.76 | 14.67 | 16.68 | 18.52 | 20.66 |
| AntiLog Accuracy bits | 10.51 | 12.44 | 14.49 | 16.42 | 18.37 | 20.32 |

TABLE V
ACCURACY OF LOGARITHM THROUGH LEAST SQUARE-BASED QUADRATIC INTERPOLATION

| No. of words in LUT | 64 | 128 | 256 |
|---|---|---|---|
| Quadratic Interpolation Accuracy | 15.29 | 16.96 | 19.17 |

performing the inner logarithm, we leave out the multiplication term in (15). Only the coefficient $b$ is looked up and added to the mantissa $m$.

The antilogarithm is also computed the same way. It is approximated as

$$2^m \simeq 1 + m + c \cdot n_1 + d, \quad 0 \leq m < 1. \tag{16}$$

The coefficients $c, d$ in this case are found by fitting a polynomial to different segments of the curve given in (14). The improvement in errors is shown in Table IV.

Note that the width of each word in the LUT has to be higher than the accuracy obtained. Otherwise the accuracy will be affected by errors due to rounding of LUT words.

## VI. INTERPOLATION USING A LEAST SQUARE-BASED QUADRATIC POLYNOMIAL

An interesting experiment is to explore how much improvement in accuracy can be obtained by interpolation using a quadratic polynomial. In this case, the logarithm would be approximated as

$$\log_2(1+m) \simeq m + a \cdot n_1^2 + b \cdot n_1 + c, \quad 0 \leq m < 1. \tag{17}$$

Here $a, b, c$ are the coefficients of the least square quadratic polynomial. If $k$ is the total number of bits in the mantissa, the first $t$ bits are used to access the LUT. $n_1$ is the value represented by the last $k - t$ bits of the mantissa.

We implement (17) by extending our approach for the linear least square polynomial interpolation. The quadratic term is realized as the antilog of the sum of twice the logarithm of $n_1$ and the logarithm of $a$. The linear term in the equation is implemented as before. The values $\log_2(a)$ and $\log_2(b)$ are stored in an LUT and the coefficient $c$ is stored in another LUT. To find the $\log_2(n_1)$ we implement (17) after neglecting the linear and the quadratic term, i.e., by looking up the constant term $c$. The hardware resources as compared to the linear interpolation method differs by one extra LUT to store the coefficient of the quadratic term and an extra adder to add the quadratic term in the approximation polynomial. The accuracy due to this is shown in Table V.

From Table V, it can be seen that quadratic interpolation yields only about one extra bit of accuracy as compared to linear interpolation. The accuracy of quadratic interpolation is limited by the way in which the multiplications in the interpolation step are implemented. Table V also shows that, if the multiplications in the interpolation step were implemented in the perfect sense, quadratic interpolation would have yielded a much better accuracy than linear interpolation. The accuracy of quadratic interpolation is limited by implementing the multiplication as a logarithm and antilogarithm (obtained through a single LUT). This is explained further in Section VII.

## VII. EXPERIMENTAL RESULTS AND COMPARISONS

We summarize the various approaches investigated to find the logarithm of a number. These include LUT only, LUT with linear interpolation, LUT with linear/quadratic interpolation based on polynomials obtained using a least square fit. The accuracies of these approaches are shown in Table VI. This table also shows the accuracy of linear and quadratic interpolation implemented with perfect multipliers for reference purposes.

It is seen that our approximate approach using linear least square polynomial interpolation is almost as accurate as perfect linear interpolation. The approximate quadratic least square polynomial interpolation is significantly less accurate than perfect quadratic interpolation. This is due to the fact that the multiplication required for interpolation is implemented by a single LUT which has only 7–9 bits of accuracy. Intuitively the approximate polynomial interpolation is implemented in two steps of single LUT lookups which give an accuracy of around 14–18 bits. Quadratic interpolation is also implemented in two steps and has almost the same accuracy. Logarithm computation using an LUT with a linear least square based polynomial has a very good accuracy and uses the same resources as linear interpolation. The Antilogarithm is also computed using the same technique.

In the remainder of this section, we compare our linear least square polynomial approach with existing approaches that approximate the logarithm function. We also analyze the LUT-based approximation methods and compare the memory requirement for these methods in bits.

While computing the LUT size for our method, we optimize the memory requirement for storing the linear polynomial coefficients as follows. The width of the words used in the LUT is seen in Table IV. For the constant term stored in the LUT in $2^t$ locations, the first 3 bits of all $2^t$ locations are zero. Hence the first 3 bits can be omitted in all locations. This holds for both logarithm and antilogarithm implementations.

In Table VII, we compare the LUT size required by our method with the symmetric bipartite table method (SBTM) [8]. The SBTM involves two parallel table lookups, an addition and two XOR operations. From Table VII, note that our approach requires far less memory than the SBTM approach. Also when the required accuracy increases, we see that the LUT size for the SBTM method needed to support this accuracy increases at a much faster rate than for our method. We also compare

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

TABLE VI
ACCURACY OF VARIOUS APPROACHES TO FIND log(), IN BITS

| No. of words in LUT | 64 | 128 | 256 |
|---|---|---|---|
| Width of each word, bits | 16 | 18 | 20 |
| LUT only | 7.21 | 8.19 | 9.18 |
| LUT with perfect linear interpolation | 14.93 | 16.90 | 18.76 |
| LUT with approximate linear interpolation | 14.67 | 16.68 | 18.52 |
| LUT with least Square linear polynomial interpolation | 14.49 | 16.42 | 18.37 |
| LUT with perfect quadratic interpolation | 23.41 | 26.25 | 29.36 |
| LUT with least square quadratic polynomial interpolation | 15.29 | 16.96 | 19.17 |

TABLE VII
COMPARISON OF TABLE SIZES, IN BITS

| Accuracy in Bits | Our method | SBTM | | Kmetz | | Maenner | | Brubaker | |
|---|---|---|---|---|---|---|---|---|---|
| | Table Size | Table Size | Table Size ↑ | Table Size | Table Size ↑ | Table Size | Table Size ↑ | Table Size | Table Size ↑ |
| 10 | 528 | 2432 | 4.61× | 2560 | 4.8× | 10240 | 19.39× | 20480 | 38.79× |
| 12 | 1280 | 7424 | 5.80× | 12288 | 9.60× | 49152 | 38.40× | 98304 | 76.80× |
| 14 | 2944 | 20992 | 7.13× | 57344 | 19.48× | 229376 | 77.91× | 458752 | 155.83× |
| 16 | 6656 | 55296 | 8.31× | 262144 | 39.38× | 1048576 | 157.54× | 2097152 | 315.08× |
| 18 | 14848 | 159744 | 10.76× | 1179648 | 79.45× | 4718592 | 317.79× | 9437184 | 635.59× |
| 20 | 32768 | 450560 | 13.75× | 5242880 | 160.00× | 20971520 | 640.00× | 41943040 | 1280.00× |

our approach with methods given by Brubaker [15], Maenner [16], and Kmetz [17]. The LUT size for all these methods were chosen to be the least LUT size to give the required accuracy. *Note that if these methods were to provide an accuracy comparable to ours, they need much bigger table sizes.*

The extra accuracy we obtain in our method is traded-off with the need to implement a modest number of additional components for interpolation. We quantified this overhead by implementing our method (as well as the SBTM method [8]) using the Xilinx ISE Foundation tool [26]. We used an XC2VP30 FPGA as the target for this experiment. Table VIII reports the outcome of this experiment, for outputs accurate up to 14 and 16 bits. Although our method has a bigger logic overhead in terms of flip-flops, 4—input LUTs, and slices, these numbers are insignificant when compared to available logic resources on a conventional FPGA. For example in the XC2VP30 FPGA with 30 000 slices, both methods utilize less than 1% of the FPGA resources. Also our method is far more conservative with respect to the memory resources utilized. Note that our method scales extremely well to obtain higher accuracies. To scale from 14 bits of accuracy to 16 bits we need a minimal increase in the logic resources and twice the memory resources as for 14 bits. The SBTM also needs a very small increase in the logic utilization, but needs close to three times more memory than for 14 bits. This trend in resource utilization holds as we scale to higher accuracies. The SBTM thus presents a bottleneck in its memory requirement for higher accuracies. We compared the maximum clocking speed of our method and the SBTM. Both the methods were pipelined to have as high a throughput as possible. Our architecture was pipelined to have a latency of 12 clock cycles. After placement and routing on the target FPGA both methods were able to support clock speeds of a little over 350 MHz. This frequency of operation was verified by performing post place and route simulation using the Xilinx ISE simulator tool. Both setup time and hold time constraints were satisfied at the said speed.

TABLE VIII
FPGA RESOURCE UTILIZATION, POWER CONSUMPTION

| FPGA Resources | Our Method | | SBTM | |
|---|---|---|---|---|
| Accuracy (bits) | 14.67 | 16.68 | 14 | 16 |
| Flip Flops | 415 | 415 | 80 | 89 |
| 4 - Input LUTs | 190 | 210 | 43 | 50 |
| Slices | 260 | 287 | 50 | 52 |
| Block RAM size (bits) | 2944 | 6656 | 20992 | 55296 |
| Power Consumed (mW) | 174 | 216 | 127 | 129 |

## VIII. CONCLUSION

The implementation of functions such as log() and antilog() in hardware is important, due to their prevalence in several computing applications. In this paper, we present an approach to compute log() and antilog() in hardware. Our approach is based on a LUT, followed by an interpolation step. The novelty of our approach lies in the fact that we find the log() of a number efficiently using interpolation, without the need to explicitly perform multiplication or division. While computing the logarithm of a number, we perform the multiplication required during the interpolation step, by utilizing an antilogarithm operation. The antilogarithm operation is also performed by utilizing a LUT. We compare our work with existing methods, and show that our approach results in significantly lower memory utilization for the same accuracy. Also our method scales extremely well to accommodate higher accuracies. One interesting addition to our method would be to explore the use of LUTs of sizes other than powers of 2. This will give our method more scalability.

## REFERENCES


[1] J.-A. Pineiro, "Algorithm and architecture for logarithm, exponential, and powering computation," *IEEE Trans. Computers* vol. 53, no. 9, pp. 1085–1096, Sep. 2004.

[2] D. M. Mandelbaum and S. G. Mandelbaum, "A fast, efficient parallel-acting method of generating functions defined by power series, including logarithm, exponential, and sine, cosine," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 1, pp. 33–45, Jan. 1996.


Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.

[3] M. J. Schulte and J. E. E. Swartzlander, "Hardware designs for exactly rounded elementary functions," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 964–973, Aug. 1994.

[4] J. A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, "High-radix logarithm with selection by rounding: Algorithm and implementation," *J. VLSI Signal Process. Syst.* vol. 40, pp. 109–123, May 2005.

[5] D. K. Kostopoulos, "An algorithm for the computation of binary logarithms," *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1267–1270, Nov. 1991.

[6] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *Proc. 10th Symp. Comput. Arithmetic*, Jun. 1991, pp. 232–236.

[7] J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *J. VLSI Signal Process.*, vol. 21, pp. 167–177, Jun. 1999.

[8] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. Computers*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[9] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Trans. Electron. Computers*, vol. 11, pp. 512–517, Aug. 1962.

[10] K. H. Abed and R. E. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter," *IEEE Trans. Computers*, vol. 52, no. 11, pp. 1421–1433, Nov. 2003.

[11] C. Layer, H. J. Pfleiderer, and C. Heer, "A scalable compact architecture for the computation of integer binary logarithms through linear approximation," in *Proc. Int. Symp. Circuits Syst.*, May 2004, pp. 421–424.

[12] S. L. SanGregory, C. Brothers, D. Gallagher, and R. E. Siferd, "A fast low-power logarithm approximation with CMOS VLSI implementation," in *Proc. IEEE Midw. Symp. Circuits Syst.*, Aug. 1999, vol. 1, pp. 388–391.

[13] K. H. Abed and R. E. Siferd, "VLSI implementation of a low-power antilogarithmic converter," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1221–1228, Sep. 2003.

[14] V. Mahalingam and N. Ranganathan, "An efficient and accurate logarithmic multiplier based on operand decomposition," in *Proc. 19th Int. Conf. VLSI Des. held jointly with 5th Int. Conf. Embedded Syst. Des. (VLSID)*, Washington, DC, 2006, pp. 393–398.

[15] T. A. Brubaker and J. C. Becker, "Multiplication using logarithms implemented with read-only memory," *IEEE Trans. Computers*, vol. C-24, no. 8, pp. 761–766, Aug. 1975.

[16] R. Maenner, "A fast integer binary logarithm of large arguments," in *Proc. IEEE MICRO*, Dec. 1987, vol. 7, pp. 41–45.

[17] G. L. Kmetz, "Floating point/logarithmic conversion system," U.S. Patent 4 583 180, Apr. 15, 1986.

[18] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European logarithmic microprocessor," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 702–715, Jul. 2000.

[19] M. G. Arnold and M. D. Winkel, "A single-multiplier quadratic interpolator for LNS arithmetic," in *Proc. Int. Conf. Comput. Des.: VLSI Comput. Process. (ICCD)*, Washington, DC, 2001, p. 178.

[20] D. M. Lewis, "114 MFLOPS logarithmic number system arithmetic unit for DSP applications," in *Proc. Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1995, pp. 1547–1553.

[21] D. M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit," *IEEE Trans. Computers* vol. 43, no. 8, pp. 974–982, Aug. 1994.

[22] D. M. Lewis and L. Yu, "Algorithm design for a 30-bit integrated logarithmic processor," in *Proc. 9th Symp. Comput. Arith.*, Sep. 1989, pp. 192–199.

[23] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 bit logarithmic number system processor," *IEEE Trans. Computers* vol. 37, no. 2, pp. 190–200, Feb. 1988.

[24] V. Paliouras and T. Stouraitis, "Low-power properties of the logarithmic number system," in *Proc. 15th IEEE Symp. Comput. Arith. (ARITH)*, Washington, DC, 2001, p. 229.

[25] M. Arnold, T. Bailey, and J. Cowles, "Error analysis of the Kmetz/Maenner algorithm," *J. VLSI Signal Process. Syst.*, vol. 33, pp. 37–53, Jan.–Feb. 2003.

[26] Xilinx, San Jose, CA, "ISE Foundation," 2008. [Online]. Available: http://www.xilinx.com/ise/logic_design_prod/foundation.htm

**Suganth Paul** received the B.E. degree in electronics and communication engineering from the College of Engineering Guindy, Anna University, India, and the Master of Science degree in electrical and computer engineering from Texas A&M University, College Station.

He is currently a Circuit Design Engineer with Intel Inc., Austin, TX. During his graduate studies, he has done research on various VLSI topics including fast computer arithmetic, high speed implementation of digital circuits, subthreshold circuit design for communication systems, and FPGA platforms for weather radar signal processing.

**Nikhil Jayakumar** received the Bachelor's degree in electrical and electronics engineering from the University of Madras, India, the Masters degree in electrical engineering from the University of Colorado at Boulder, Boulder, and the Doctoral degree in computer engineering from the Texas A&M University, College Station.

He is currently with Texas Instruments, Inc., Dallas, TX. During his graduate and doctoral studies, he has done research and published several papers in many aspects of VLSI including formal verification, clock network design, routing, structured ASIC design, radiation-hard design, logic synthesis, LDPC decoder architectures, Statistical timing, low leakage power design techniques, and subthreshold circuit design.

**Sunil P. Khatri** received the B.Tech. degree in electrical engineering from Indian Institute of Technology Kanpur, Kanpur, India, the M.S. degree in electrical and computer engineering from the University of Texas, Austin, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley.

He was with Motorola, Inc., for four years, where he was a member of the design teams of the MC88110 and PowerPC 603 RISC microprocessors. He is currently an Assistant Professor with the Department of Electrical Computer Engineering, Texas A&M University, College Station. His research interests include logic synthesis, novel VLSI design approaches to address issues such as power, cross-talk, hardware acceleration of CAD algorithms, and cross-disciplinary applications of these topics. He has coauthored about 125 technical publicatons, 5 U.S. Patents, one book, and a book chapter.

Dr. Khatri was a recipient of two Best Paper Awards and two Best Paper Nominations. His research has been supported by Intel Corporation, Lawrence Livermore National Laboratories, the National Securities Agency, the National Science Foundation, and Nascentric, Inc.

Authorized licensed use limited to: Texas A M University. Downloaded on June 24, 2009 at 02:43 from IEEE Xplore. Restrictions apply.