International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622
National Level Technical Symposium On Emerging Trends in Engineering & Sciences
(NLTSETE&S- 13th & 14th March 2015)

RESEARCH ARTICLE OPEN ACCESS

Computations Of Elementary Functions Based On Table Lookup And Interpolation

Syed Aliasgar[1], Dr.V.Thrimurthulu[2], G Dillirani[3]
[1]Assistant Professor, Dept.of ECE, CREC, Tirupathi, A.P, India
[2] Professor, Head of ECE Dept., CREC, Tirupathi, A.P, India
[3]Assistant Professor, Dept.of ECE, CREC, Tirupathi, A.P, India
[1]aliasgarsyed@gmail.com, [2]vtmurthy.v@gmail.com, gdillirani@gmail.com

Abstract
In this paper, we are focussing on the realization of functions such as log ( ) and antilog ( ) in hardware due to their importance in several computing applications.
In this paper, we are trying to show low memory requirement as compared with other methods to provide better accuracies. The idea of this project was 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 ( ). This method scaled very well with an increase in the required accuracy, compared to existing techniques. The proposed algorithm is implemented and tested using verilog hardware description language.
Keywords*:* Lookup Table,Interpolation, FPGA.

I. INTRODUCTION

The generation of elementary functions such as log() and antilog() finds uses in many areas such as Digital Signal Processing, 3D computer graphics, scientific computing, artificial neural networks, logarithmic number systems andother multimedia applications. In fact the fast generation of these functions is critical to performance in many of these applications. Using software algorithms to generate theseelementary functions is often not fast enough [1], [2]. 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 which are well researched and used for the generation of the logarithm function are digit-recurrence algorithms and look up table 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 [3]. The look up table based methods are widely used to approximate logarithm and antilogarithm functions. Some of the previous works involving look up table based methods include, table lookups combined with polynomial approximations [2], [4], symmetric bipartite table based approximations [5], [6] etc. The main objective of all these works is to utilize minimum circuit area while retaining the accuracy of the approximation. Our approach combines a table lookup with a linear interpolation, implemented in a manner that optimizes the area 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 work, the number format used is similar to the IEEE 754 single-precision floating point format which 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 represented in this format is given by

_1:M _ 2E□127: (1)

We use a similar number format representation, but assume the number of bits k in the mantissa to be variable. We target 13 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, and also provides an error analysis of the approximation. In Section V we present some estimates on the area of the architecture, and compare it with other relevant works. We conclude in Section VI.

II. OUR APPROACH

This work uses a look up table 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 reduction. The idea is describedn the following sections.

A. Interpolation Approach

Mitchell's approximation [7] is given by the equation.

$$\log(1 + m) = m \quad (2)$$

The error due to this approximation is given by

$$EM = \log(1 + m) - m \quad (3)$$

The error curve shown in Figure 1 is sampled at 2t points (depending on the size of the table required) and the sampled values are stored in the look up table. The look up table is addressed by the first t bits of the mantissa portion ofthe number. Now we investigate the option of interpolatingbetween the values stored in the table. This is done by the following equation.

$$\log(1 + m) = m + a + (b - a) \_ n12k - t \quad (4)$$

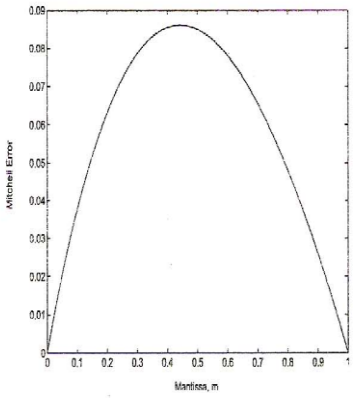

Fig. 1. Error due to Mitchell Approximation

Here m is the mantissa part, a is the error value from thetable accessed by the first t bits and b is the next table value adjacent to a. k is the total number of bits used to represent the mantissa and n1 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. The third term in Equation 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 Bipartitioning
STEP 1: The first t bits 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 Figure 2 This gives us 14 accurate bits. 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 above algorithm is implemented for t, t + 1 and so on upto 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. The depth of the look up table used is 64 words and the size of each word is 16 bits. 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 n1. 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 sections(s) that our approach gives similar error performance as the 7 bit interpolation, with smaller area and lower delay.

# of Interpolations	0	1	2	3	4	5	6	7	8
Maximum Error $\times 10^{-3}$	3.4147	1.7182	0.8834	0.464	0.2583	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 I
MAXIMUM ERROR FOR A LIMITED INTERPOLATION APPROACH

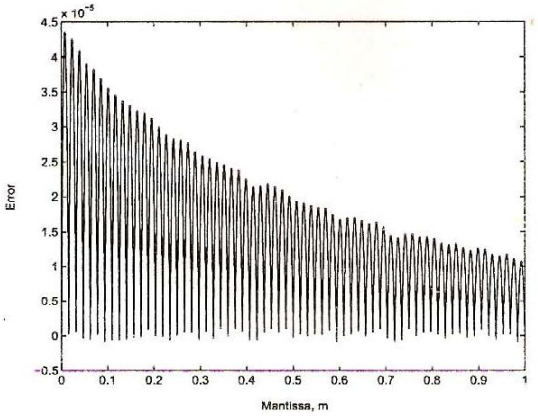

Fig. 2. Error Performance of Algorithm 1

B. More Effective Approach

In this section, we propose a more efficient approach to do interpolation without the multiplication of (b□a) and n1 in Equation 4. The essential idea is that the multiplication of (b □ a) and n1 is simplified by taking the antilogarithm of the sum of the logarithm of (b □ a) and the logarithm of n1. In order to perform this operation with a small delay, we consider the following options:

1) log2(b □ a) may be approximated by

a) Mitchell approximation

b) Table look up : For the table look up option the constants log2(b □ a) for each of the intervals is stored in the original look up table. Recall that we stored the error values obtained by using a Mitchel approximation for the log function. The log2(b□a) values are stored along with the error values and are indexed by the same address lines.

2) Antilog of (log2(n1) + log2(b □ a)) may be approximated by

a) Mitchell approximation

b) Table look up : To obtain the antilogarithm of a number by this method, we need to construct another similar look up table. The error due to the Mitchell approximation of the antilogarithm function is stored in a lookup table as shown inSection IV. This antilog lookup table is utilized to compute the logarithm of a number, since the multiplication of (b □ a) and n1 is performed by taking the antilogarithm of the sum of the logarithm of (b □ a) and the logarithm of n1 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 II.

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

TABLE II
MAXIMUM ERROR x 10□3 FOR VARIOUS APPROACHES

From Table II we see that the combination of 1:b) and 2:b) has the best error performance. Therefore, to perform the interpolation described in Equation 4, it makes sense to find the antilogarithm using a lookup table. The additional advantage of this is that the same lookup table 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 trade-off between the accuracy and the number of values stored in the table.

C. Architecture of Implementation

The Figures 3 and 4 show the architecture of the implementation of our scheme. The implementation is pipelined, with 7 stages in the pipeline. We use a 23-bit mantissa and a 26 by 16 bit lookup table as an example. The width of each word in the table is chosen as 16 bits so that the error due to truncation does not dominate the overall error. Recall that the overall error, from Table II, was 13.74 bits in this case. Since the exponent part of the input number trivially becomes the decimal part of the logarithm, we only show the operations on the mantissa. One of the adders is a three input fixed point adder as shown in Figure 4. The output of the antilog lookup is shifted by a value equal to the the decimal value of the output of this 3-input adder. This operation is not shown in Figure 4. 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(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 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 Figure 4. 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 LOD or leading One Detector block detects the first bit that has a value 1. It then uses the mantissa given by the remaining bits to access the lookup table. The decimal part of log n is directly sent to the three input fixed point adder. Also the 27 term in the denominator of Equation 4 is trivially accounted for after the antilog look up stage, by a constant right shift of 7 bits.

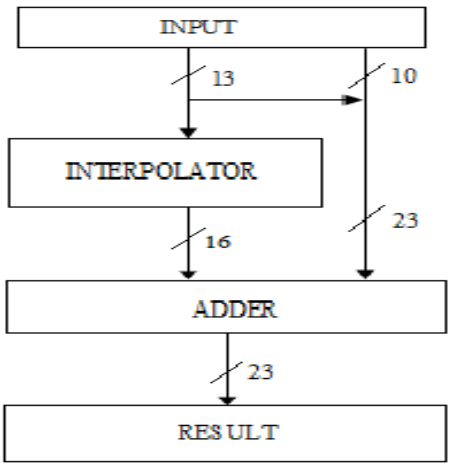

Fig. 3. Block Diagram of the Log Engine

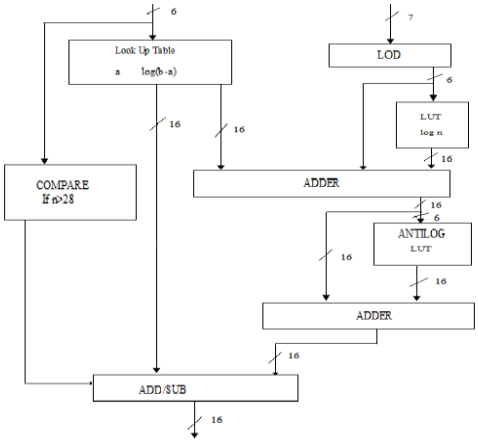

Fig. 4. Architecture of the Interpolator

D. Error Analysis

The expression for error due to a Mitchell approximation of the logarithm is given by Equation 3. The error curve is plotted in Figure 1

As mentioned before, this curve 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 the table, the maximum error is bound to occur when the difference between the two error values is largest.

The largest error occurs for the first pair of points in the lookup table. The size of this largest error (assuming a table of size 6 and width 16) is given by:

Smax = log2(1+2□6)□2□6□log2(1+0)□0 = 6:7428_10□3 (5)

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

E = log2(1 +m) □m □ a □(b □ a) _ n1 2k□t (6)

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,

Etot = Etr + Eint + Eanti (7)

where Etr is the error due to truncation of bits stored in the lookup table, Eint is the error due to interpolation and Eanti is the error incurred due to the use of the anti-logarithm function during interpolation. The input number is split into three different ranges for the error analysis.

Case 1: Inputs from 0 to 27-1 (i.e. mantissa with 7 bits): Here the error is due to truncation 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 truncation error is given by 2□17.

Case 2: Inputs from 27 to 214-1 (i.e. mantissa with 14 bits): The error has contributions from all the three error sources mentioned above. Interpolation is done by using Equation 8. In this case the first value of the interval is a = 0 and the last value is b = Smax. Also the first 6 bits of the mantissa rezeros since we are in the first of 26 intervals, and the value of n1 is given by the decimal value of the next 7 bits. This can be expressed as n1 = m _ 213. The interpolation error term is given by

Interpolation term = a + (b □ a) _ n1 27 (8)

Substituting b, a and n1 in 8 we get the interpolation term in the first interval as Interpolation term = Smax(m_ 213)=27: (9)

If the error due to anti-log is assumed to be zero, we can find that the error expression due to interpolation by using the following equation.

Eint = log2(1 +m) □m □ Smax _ (m _ 26): (10)

The maximum error is found by differentiating this equation, and setting it to zero. We get the maximum error as

Emaxint =4:33 _ 10□5 at m = 7:7929_ 10□3.

The anti-log error depends on the values of log2(b □ a) which is stored in the table and log2(n1) . We find the error due to antilog by simulating the lookup table based antilog approximation for these particular values and find the maximum anti-log error to be 8:4433_ 10□6.

The results of the simulation are shown in Figure 5 for all possible values of m ranging from 0 to 1.

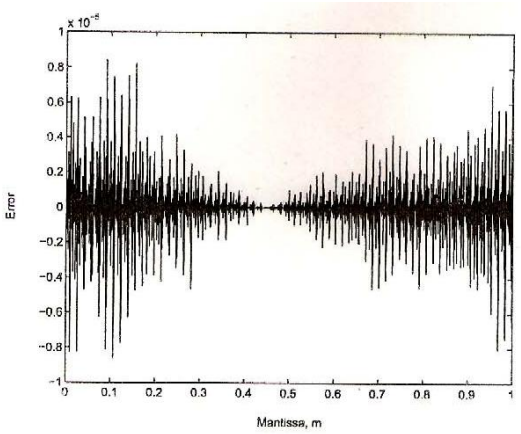

Fig. 5. Anti-log Error during Interpolation

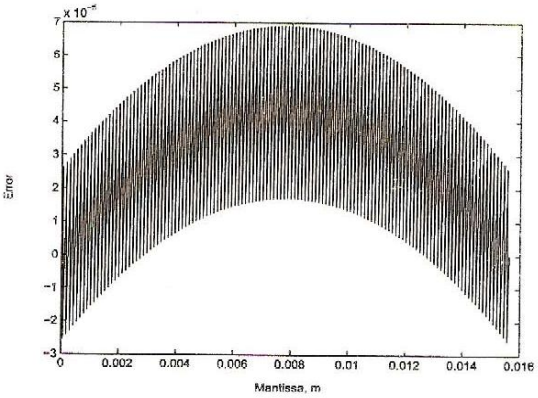

Fig. 6. Interpolation Error for case 3

Case 3: For numbers from 214 to 2k-1 (i.e. mantissa with higher than 14 bits precision): The truncation 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 n1 is given by n1 = round(m_213),where the round() function represents a round-off to the closest integer. The error function for this case is given by:

Eint = log2(1 +m) □m□Smax _ round(m _ 213)27

Plotting this expression from m = 0 to 2□6 in Figure 6, we find the maximum error as 6:9276_10□5. Out of all the above cases the third case has the worst error due to interpolation, while the errors due to truncation and antilog approximation remain the same. Hence the error bound, given by plugging in the maximum values of each of the error components in 7 is

Etot =8:5348 _ 10□5:

III. ANTILOGARITHM COMPUTATION

The Mitchell approximation to find the binary antilogarithm of a number is given by the expression:

2m = 1 +m (12)

The error due to this expression is given by

EM = 2m □ (1 +m) (13)

We sample the error curve at as many points as required, and store these values in the first entry of a lookup table. If c; d are any two adjacent values stored in the memory, we store log2(c □ d) in the second entry of the lookup table. The architecture used for the implementation of the antilog function is very similar to the logarithm implementation except for these differences. In Figure 3, instead of the adder block we use a subtractor 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 Figure 4 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 Size	64	128	256
Maximum Error ×10^{-5}	5.4806	1.4214	0.3983
Accurate Bits	14.16	16.1	17.94

TABLE III
WORST CASE ANTILOG ERROR

IV. EXPERIMENTAL RESULTS AND COMPARISONS

In this section, we make comparisons with existing approaches that approximate the logarithm function. In comparison to the 6-region error correction algorithm described in [8], our approach is more flexible in that it supports better accuracies by increasing the LUT size and adding more bits to the adders/subtractors involved. The 6-region error correcting method has an absolute maximum error of 0.0132 which givesan accuracy of 6 bits. We also analyze the LUT based approximation methods. Table IV compares the accuracy of methods given by Brubaker [11], Maenner [12], and Kmetz [13] for a fixed table size of 2048 bits. *We note that if these methods were to provide an accuracy comparable to ours, they need much bigger table sizes*. The other methods shown in Table IV implement only the log function. Since our method implements both the log and the antilog function, we report only half the total lookup table size needed for our approach. In Table V, we compare the lookup table size required by our method and the symmetric bipartite table method [6]. The symmetric bipartite table method involves two parallel table lookups and an addition. From Table V, we note that our approach requires far less memory than the SBTM approach. Also when the required accuracy increases in bits, 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. For a 2 bit increase in the accuracy, the SBTM LUT size increases almost by a factor of a power of 3 whereas the LUT size of our method increases by a factor of a power of 2. 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 [6]) using the Xilinx ISE Foundation tool [15]. We used the XUPV2P board for this experiment. Table VI reports the outcome of this experiment, for outputs accurate upto 13 bits and 15 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 XCV2P30 FPGA with 30000 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 13 bits of accuracy to 15 bits we need a minimal increase in the logic resources and twice the memory resources as for 13 bits. The SBTM also needs a very small increase in the logic utilization, but needs close to three times more memory than for 13 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. Note that for 13 bits of accuracy we need two tables with 2048 bits each to generate the log and the antilog function. Also while computing the log function, half of the antilog look up table is required and vice versa during the antilog computation. The XUPV2P board used for the experiments supports dual port block RAM memory and so both log and antilog functions can be implemented simultaneously, without creating conflicts during memory access.

	Table Size	Accuracy in Bits
Our approach	2048	13.74
Brubaker	2048	7.71
Maenner	2048	8.47
Kmetz	2048	9.82

TABLE IV
COMPARISON OF ACCURACY

Method	Accuracy,bits	LUT Size, bits
Our Method	13.74	2048
	15.51	4096
	17.15	8192
SBTM Method	13	11776
	14	20992
	15	34816
	16	55296
	17	98304

TABLE V
COMPARISON OF TABLE SIZES

FPGA Resources	Our method		SBTM	
Accuracy (bits)	13.74	15.51	13	15
Flpi Flops	243	247	30	34
4-Input LUTs	182	188	19	22
Slices	174	178	11	12
Block Ram Size	2048bits	4096bits	11776bits	34816bits

TABLE VI
FPGA RESOURCE UTILIZATION

V. CONCLUSION

In this paper, The simulated results and the computed results always have at least 10 bits of accuracy which is the target of the implementation.In this project, presented approach is to compute log ( ) and antilog ( ) in hardware. This approach is based on a LUT, followed by an interpolation step.We find log ( ) and antilog ( ) of a number efficiently using interpolation, without the need to explicitly perform multiplication or division.Performed the multiplication which is required during the interpolation step, by utilizing an antilogarithm operation. The antilogarithm operation is also performed by utilizing a LUT. This approach results in significantly lower memory utilization for the same accuracy. This method scales extremely well to accommodate higher accuracies.

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 parallelacting method of generating functions defined by power series,including logarithm, exponential, and sine, cosine,*" IEEE Trans. ParallelDistrib. Syst., vol. 7, no. 1, pp. 33–45, Jan. 1996.

[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] *"A Fast Hardware Approach for Approximate, Efficient Logarithm and antilogarithm Computations"*, ieee transactions on VLSI systems vol.17 No2,feb2009.