Fast Algorithm of A 64-bit Decimal Logarithmic Converter

Ramin Tajallipour, Md. Ashraful Islam, and Khan A. Wahid
Dept. of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, Saskatchewan, Canada
Email: rat177@mail.usask.ca, mdi142@mail.usask.ca, khan.wahid@usask.ca

Abstract— **The paper presents an efficient algorithm to compute base-10 logarithm of a decimal number. The algorithm uses a 64-bit floating-point arithmetic, and is based on a digit-by-digit iterative computation that does not require look-up tables, curve fitting, decimal-binary conversion, or division operations. It is the first FPGA prototype of its kind that uses a 64-bit (decimal 16-digit) precision. Two numerical examples have been presented for the purpose of illustration. The algorithm produces very accurate result with a maximum absolute error of 3.53×10^{-14}. The architecture is pipelined and implemented on to the Xilinx Virtex2p FPGA. It costs 6,752 logic cells, outputs at a minimum rate of 51 mega-samples/sec, and consumes 125.7 mW of power. The scheme is very suitable for timing and accuracy critical applications and compliant with the IEEE754-2008 standard (*decimal*64 format).**

Index Terms— **Decimal logarithm converter, floating-point arithmetic, iterative computation, IEEE754-2008**

I. INTRODUCTION

Elementary functions such as the logarithm and the exponential operations have become very useful in many applications such as, financial analysis, tax calculation, internet based applications, and ecommerce [1], where these operations are used to avoid hardware-expensive multiplication and division operations. In the past, several hardware-efficient methods have been proposed for computing the base-2 logarithm of binary numbers [4][5][11]-[16]. However, after the inclusion of decimal floating-point (FP) operation in the latest IEEE754-2008 standard [6], more researchers have devoted their effort in developing decimal FP algorithms and architectures to efficiently compute logarithms [7][17], exponentiation [28], trigonometric operations, etc.

A study has shown that 55% of the numbers stored in the database of 50 big organizations is decimal [21]. There are several software packages available to customer to compute decimal numbers using decimal arithmetic to minimize error [23], but the software-implemented decimal arithmetic requires much longer time to execute than the hardware version [1], which led momentum to its implementation in hardware. IBM has recently implemented decimal FP architecture in their POWER6 [3][19], z9 [29], and z10 microprocessors [20]. Several decimal architecture of multi-operand carry-save adder [24][25], carry look-ahead adder [26], parallel BCD adder [27], signed-digit adder [18], etc. have been proposed.

There are several applications which require the direct computation of decimal (or radix-10) logarithm, such as, to measure the pH in chemistry, the earthquake intensity in Richter scale, the optical density in spectrometry and optics, the brightness of stars in astronomy, etc. [2]. Moreover, the radix-10 logarithm is widely used in computing the ratio of voltage and power levels (called *bel*) in telecommunications, electronics and acoustics. In most base-10 logarithmic converters, the decimal input is first converted to binary followed by base-2 logarithm computation; after the completion, the result is converted back to decimal radix – these back and forth conversions of bases introduce errors on the system. A generalized iterative algorithm to compute base-k logarithm has been presented in [8]; however, the division operation in that work limits the performance by incurring erroneous computation. Moreover, the use of lookup tables and the lack of user control on the number of iteration make this algorithm very inefficient for hardware implementation.

We have recently presented a 32-bit decimal logarithm (in short, *log*) converter [22]. While the *decimal*32 format, as defined in the IEEE754-2008 standard [6], is only used for storage, the *decimal*64 and *decimal*128 are used for more accurate decimal computation. Being motivated by the fact, in this paper, we extend the algorithm to compute the radix-10 log using *decimal*64 precision, which is the first FPGA prototype of its kind. The algorithm is based on a digit-by-digit iterative computation that does not require error correction circuitry, look-up tables, curve fitting, or division operations. The number of iterations of the log converter depends on the user defined precision. The previous 32-bit design [22] suffers from high latency (e.g. 40 clock cycles) due to an inefficient power-10 algorithm and un-pipelined operation. Here, we present a very efficient power-10 module with a pipelined architecture that may take only 4 clock cycles to produce the log result with a high process throughput of 51 mega-samples/sec. The error analysis shows that the proposed scheme produces very accurate result. The architecture is developed based

Manuscript received January 1, 2009; revised June 1, 2009; accepted July 1, 2009.

doi:10.4304/jcp.5.12.1847-1855

on 64-bit binary coded decimal (BCD) representation and compliant with the IEEE *decimal*64 FP standard.

The paper is organized as follows: Section II presents the background. In section III, the digit-by-digit algorithm is presented. The pseudo-code of the algorithm and two examples are also presented for illustration. A detailed hardware implementation is discussed in section IV, where the description of different internal modules is presented. Section V discusses the performance analysis with a comparison of hardware among related log converter designs. The paper is concluded in section VI.

II. BACKGROUND

The general form of any positive number, L can be expressed as:

$$L = \sum_{i=-\infty}^{+\infty} C_i R^i = ... + C_2 R^2 + C_1 R^1 + C_0 R^0 + C_{-1} R^{-1} + C_{-2} R^{-2} + ... \quad (1)$$

Where, R is the numerical base, and C_i is the coefficient for the i^{th} power of that base, ranging from 0 to $R-1$. For a decimal base, R equals to 10. After taking the logarithm of any positive decimal number, P, (1) results in the following (2):

$$L = log_{10} P = ... + C_i \times 10^i + C_{i-1} \times 10^{i-1} ... + C_1 \times 10^1 + C_0 \times 10^0 + C_{-1} \times 10^{-1} + C_{-2} \times 10^{-2} ..., \; P > 0 \quad (2)$$

The coefficients in (2) can now be divided into two categories: integer (or "character": $..., C_i, C_{i-1}, ..., C_1, C_0$) and fraction (or "mantissa": $C_{-1}, C_{-2}, C_{-3}, ...$). The procedure to compute these coefficients is described in the following section.

III. ALGORITHM FOR DECIMAL64 LOG

A. Decimal64 format in IEEE754-2008

The IEEE 754-2008 decimal FP arithmetic supports the *decimal*32, *decimal*64, and *decimal*128 computation and data interchange formats, and implements all the operations and conversions [6]. The basic decimal FP format is illustrated in Fig. 1.

The *Sign* is a 1-bit field and indicates the sign of the number where *S* is 0 or 1. The *combination field* is a *w*+5-bits field that encodes two most significant bits (MSBs) of the exponent and the most significant digit (MSD) of the coefficient. The Not-a-Number (NaN) and Infinite number (Inf) are indicated in the *Combination Field*. The biased exponent is a *w*+2 bit quantity, where the value of the first two bits of the biased exponent taken together is 0, 1, or 2. The whole encoded exponent is an unsigned binary integer with the largest unsigned value. The value of the exponent is calculated by subtracting an *exponent bias* from the value of the encoded exponent, to be able to represent both negative and positive exponents.

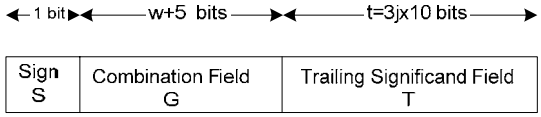


Figure 1. Decimal floating-point number format in IEEE754-2008

The *Tailing Significand Field* (3*j* x 10 bits) is formed by appending the decoded continuation digits (*j*-bit) as a suffix to the most significant digit (MSD) derived from the combination field. Each 10-bit group represents three decimal digits, using Densely Packed Decimal (DPD) encoding [30]. The format encodes a total of *p=3j+1* decimal digits, where *p* = the number of digits in the *significand* (precision). For *decimal*64 format: *w* = 8; *j* = 5, *exponent bias* = 398, and *p* = 16.

B. Proposed iterative algorithm

The IEEE754-2008 standard [6] defines any non-normalized unsigned decimal fraction as: $d_0.d_1d_2d_3...d_{15}$, where $0 \le d_i \le 9$. To be compatible with the standard, we extend our input decimal number, *P* as follows:

$$P = (-1)^S \times 10^a \times P_c \quad (3)$$

Where, *S* is the sign, *a* is the exponent, and P_c is the coefficient (in integer form). Taking a 64-bit log of (3) results in (4):

$$L = \log_{10}(P) = \log_{10}(10^a) + \log_{10}(P_c) \quad (4)$$

Here, $0000000000000001 \le P_c \le 9999999999999999$. The computation of $\log_{10}(P_c)$ follows the iterative algorithm. It starts with the computation of the upper limit of *i*, called $i_{\max}$, which is the number of mantissa digits desired in the final converted answer. $i_{\max}$ is set by the user and defines the number of iterations.

In order to perform the initial range reduction, we extend (4) further as given below:

$$L = \log_{10}(P) = a + b + \log_{10}(k) \quad (5)$$

Where, b (range: $1 \le b \le 15$) is the characteristics of $\log_{10}(P)$ and is obtained by detecting leading zeros; *k* is a decimal fraction (range: $0.1 \le k < 1$). After separating *a,* and combining (2) and (5), we get the following (6):

$$P = 10^b \times 10^{C_{-1} \times 10^{-1}} \times 10^{\sum_{i=-\infty}^{-2} C_i \times 10^i} \quad (6)$$

The intermittent data is accumulated into a temporary variable, A, where $1 < A < 10$:

$$A = \frac{P}{10^b} = 10^{C_{-1}\times 10^{-1}} \times 10^{\sum_{i=-\infty}^{-2} C_i \times 10^i} \quad (7)$$

This division (by 10) operation can be easily implemented by right shifting the input digits. In order to determine the fractional parts (e.g., $C_{-1}, C_{-2}, C_{-3}, \ldots$), we take the power-10 of (7) as shown below:

$$A^{10} = (10^{C_{-1}\times 10^{-1}} \times 10^{\sum_{i=-\infty}^{-2} C_i \times 10^i})^{10} \quad (8)$$
$$= 10^{C_{-1}} \times 10^{\sum_{i=-\infty}^{-1} C_{i-1} \times 10^i}$$

Now (8) has a structure similar to (7). The first mantissa coefficient, C_{-1}, is computed by simply counting the number of integers in (8). The temporary value (stored in A^{10}) undergoes another power range reduction and is accumulated back into *A*. The process continues for the remaining mantissa coefficients until the number of iteration reaches $i_{\max}$, set earlier by the user.

For cases where the input lies between 0 and 1, the log produces negative result. Interestingly, the proposed algorithm is capable of handling such cases. For this purpose, we first adjust he decimal point as follows:

$$P = m \times 10^n, \text{ where } m > 1, \quad n < 0 \quad (9)$$

Taking the 64-bit radix-10 log in both sides of (9) leads to the following (10):

$$L = \log_{10} P = -|n| + \log_{10} m \quad (10)$$

Now, the computation of $\log_{10} m$ follows the procedure described in (5) – (8).

C. Pseudocode of the algorithm

The pseudo-code of the proposed algorithm summarized below and illustrated in Fig. 2:

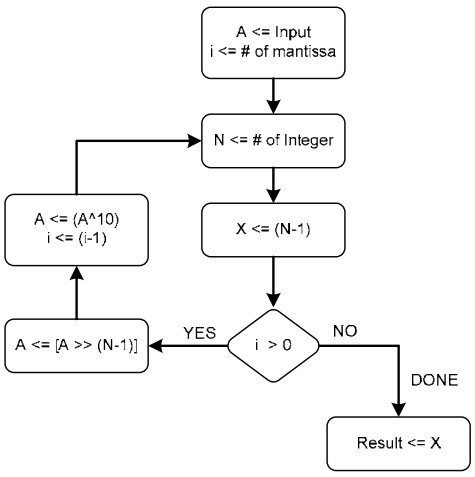


Figure 2. Flow graph of the decimal log algorithm

1. Read decimal input, *P* and the number of mantissa in the log result, *i*
2. Transfer *P* to *A* and perform initial range reduction
3. Detect the number of integers in *A*
4. Start computing the mantissa coefficients, C_i
5. Compute power-10 of *A*, and perform power range reduction
6. Decrease *i* by 1
7. If *i* > 0, repeat steps 3-6

D. Examples

In order to better illustrate the algorithm, we present two examples in the following section:

1) Determine the logarithm of decimal number, P = 123456789.123456 up to three fractional digits.

The computation steps are as follows:

- Here the user sets the number of fractional bits; so, $i_{\max} = 3$; hence, the computation process will continue up to the computation of C_{-3}, and the final answer will be in the format: $C_0.C_{-1}C_{-2}C_{-3}$
- The number of integer in *P*: $N = 9$; hence, $C_0 = (9-1) = 8$
- Right shift the digits of *P* by eight digits: $A \leftarrow (P >> 8) = 1.23456789123456$
- Compute power-10 of *A* and accumulate the result: $A \leftarrow A^{10} = (1.23456789123456)^{10}$
  $= 8.225262673723126411578214087 1879$
- Compute the number of integers in *A*: $N = 1$; hence, $C_{-1} = (1-1) = 0$
- Right shift the digits of *A* by zero digit: $A \leftarrow (8.225262673723126411578214087 1879 >> 0)$
  $= 8.225262673723126411578214087 1879$
- Compute power-10 of *A* and accumulate the result: $A \leftarrow A^{10} = (8.225262673723126411578214087 1879)^{10}$
  $= 1417417401.844385939491113689 3318$
- Compute the number of integers in *A*: $N = 10$; hence, $C_{-2} = (10-1) = 9$
- Right shift the digits of *A* by nine digits: $A \leftarrow (1417417401.844385939491113689 3318 >> 9)$
  $= 1.417417401844385939491113689 3318$
- Compute power-10 of *A* and accumulate the result: $A \leftarrow A^{10} = (1.417417401844385939491113689 3318)^{10}$
  $= 32.732381445138501594634638993872$
- Once again, compute the number of integers in *A*: $N = 2$; hence, $C_{-3} = (2-1) = 1$
- The iteration stops, and the final answer is: $L = 8.091$

2) Determine the logarithm of decimal number, P = 0.00000123456789 up to two fractional digits.

The computation steps are as follows:

- Here the input is a fraction $(0 < P < 1)$; so, $i_{\max} = (2+1) = 3$; hence, the computation process will

continue up to the computation of C_{-3}, and the final answer will be in the format: $-C_0.C_{-1}C_{-2}C_{-3}$

- Adjust the decimal point: $P = 1.23456789 \times 10^{-6}$; m = 1.23456789 , n = -6
- The number of integer in P: $N = 1$; hence, $C_0 = (1-1) = 0$
- Right shift the digits of P by zero digit: $A \leftarrow (P >> 0) = 1.23456789$
- Compute power-10 of A and accumulate the result: $A \leftarrow A^{10} = (1.23456789)^{10}$ $= 8.225262591471025795047611436615 4$
- Compute the number of integers in A: $N = 1$; hence, $C_{-1} = (1-1) = 0$
- Right shift the digits of A by zero digit: $A \leftarrow (8.225262591471025795047611436615 4 >> 0)$ $= 8.225262591471025795047611436615 4$
- Compute power-10 of A and accumulate the result: $A \leftarrow A^{10} = (8.225262591471025795047611436615 4)^{10}$ $= 1417417260.1035587702142524239761$
- Compute the number of integers in A: $N = 10$; hence, $C_{-2} = (10-1) = 9$
- Right shift the digits of A by nine digits: $A \leftarrow (1417417260.1035587702142524239761 >> 9)$ $= 1.417417260103558770214252423976 1$
- Compute power-10 of A and accumulate the result: $A \leftarrow A^{10} = (1.417417260103558770214252423976 1)^{10}$ $= 32.732348712982628176658128707107$
- Once again, compute the number of integers in A: $N = 2$; hence, $C_{-3} = (2-1) = 1$
- The iteration stops, and the final answer is: $L = -6 + 0.091 = -5.909$

Thus, it can be seen that the algorithm does not require any lookup tables, curve fitting, FP division operations, or error correction circuitry. The following section describes the hardware implementation of the proposed scheme.

IV. HARDWARE IMPLEMENTATION

The architecture of the radix-10 log converter is shown in Fig. 3. It consists of two major units both connected to a controller: Synchronous register-Counter and Core unit. The converter accepts two inputs: a 16-digit decimal number (*P*, in BCD) including the decimal point (inputted as hex '10') and the desired number of digits after the decimal radix point (*i*, in binary) in the final log result. Depending on *i*, a down counter is set which defines the number of iterations. The core unit performs the fundamental computation supervised by the Controller. The width of the data lines in all the following figures is in decimal digit, unless otherwise specified.

The architecture of the core unit is shown in Fig. 4. It does not show the interaction with the controller. The core architecture is developed using unsigned BCD representation with an internal precision of 16 digits (64-bit binary). The DPS (decimal-point separator) module detects and separates the DP, and then stores the unsigned magnitude to a temporary register. The DP follows a separate path (DP Accumulator – DP Update) that is parallel to the core computation. The 'DP Update' module tracks the position of the decimal-point and updates it after every computation step.

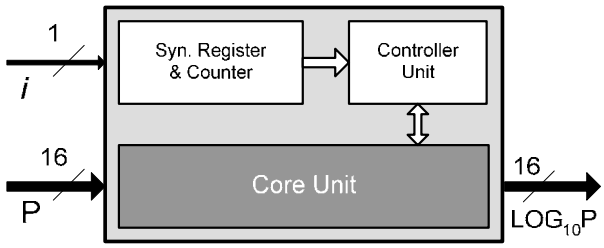


Figure 3. Block diagram of the entire system

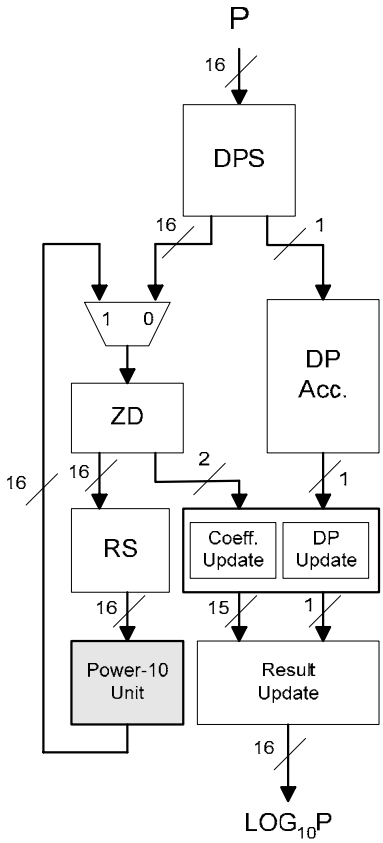


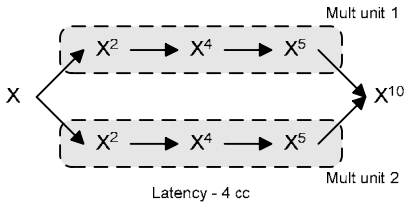
Figure 4. Architecture of the core unit

The 16-digit unsigned input is passed to the 'Zero Detector' (ZD) module that determines the number of integers, which is the first digit (or coefficient) of the final log result. The coefficient is updated at the same time in the 'Coefficient Update' module. The right-shift (RS) operation to be performed on *A* is also achieved at the same clock cycle by simply updating the position of the DP in the 'DP Update' module. The intermittent data is passed into the 'Power-10 Unit' and fed back to ZD for further processing. The data flow to ZD is controlled by

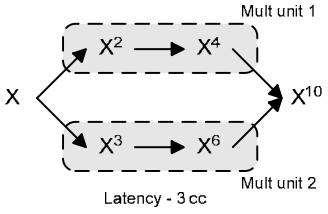
the controller (lines not shown) through a 2:1 multiplexer. The process continues until the counter reaches to zero and the controller then stops the computation. The final result is accumulated in the 'Result Update' unit after combining the outputs of 'Coefficient Update' and 'DP Update'.

A. Power-10 architecture

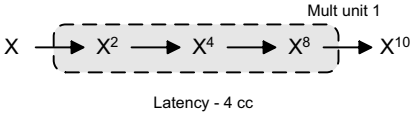
The Power-10 module is a key unit of the log converter and the accuracy of the final result greatly depends on its efficient implementation. Because of its complexity, we have explored several options based on "divide and conquer" algorithm to efficiently implement the unit which are shown in Fig 5. Considering the tradeoff between hardware cost and speed, we have chosen option 3 for our implementation. The algorithm is based on a recursive powering that requires one parallel multiplier unit and 4 cc to complete. This is a significant improvement over the previous implementation [22] which had taken 40 cc for such computation.



(a) Option 1



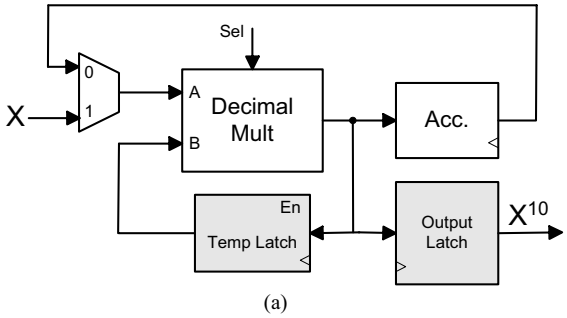
(b) Option 2



(c) Option 3

Figure 5. Power-10 algorithms and the cost of implementation

The overall architecture of the Power-10 unit is shown in Fig. 6(a) where the width of all data lines is 16-digit. It consists of a 16-digit combinational multiplier, an accumulator, and a few latches. The selection bit (*Sel*) dictates the multiplication operation: 0 for A*A; 1 for A*B. A key step of the proposed algorithm is to count the number of integers before the decimal point to evaluate the coefficients, C_i, which may take any value between 0 and 9 (where, $i < 0$). Inside the multiplier, the most significant 16 digits are retained and accumulated for further processing. The data flow is controlled using a 2:1 multiplexer by the controller (lines not shown). After the first multiplication stage, the output (i.e., X^2) is stored in a temporary latch so that it can be used later at the fourth multiplication stage. The timing diagram for each clock pulse and the selection/control sequence are shown in Fig 6(b) [here, *n* indicates the instance of the clock pulse at any given time, *t*].



(a)

Clock Pulse	n+0	n+1	n+2	n+3	n+4
Output Latch	--	X^2	X^4	X^8	X^{10}
Temp Latch (B)	--	X^2	X^2	X^2	X^2
Sel	0	0	0	1	0
En	0	1	0	0	0

(b)

Figure 6. (a) Architecture of the Power-10 unit; (b) Timing diagram

B. 16-digit decimal multiplier

Several efficient methods for decimal multiplication have been proposed in the past [9][10]. Here, we have used a general purpose 16-digit combinational multiplication algorithm which is a modified version of [9]. This is another improvement over the previous 32-bit design [22] where a sequential multiplier was used. The multiplier architecture (as shown in Fig. 7) is optimized to reach the desired throughput. The multiplier input is recoded and the partial products are first kept in a redundant format and then accumulated by a tree of redundant adders. Finally the 32-digit product is obtained by converting the carry-save tree's outputs into BCD format. The presented combinational architecture results in low latency and that is why it is chosen for the log converter.

The product, *p* is given below, where *A* and *B* are the signed multiplier and the multiplicand respectively:

$$p = A.B = \sum_{i=0}^{n-1} A \times B_i \times 10^i \quad , \quad B \in [0,9] \quad (11)$$

In order to make computation simple, B_i is recorded into two groups $B_h = \{0,5,10\}$ and $B_l = \{-2,1,0,1,2\}$, where $B_i = B_h + B_l$. Negative numbers are represented in radix-10 complement and are implemented by performing the 9's complement of the BCD digit and adding a '1' with the least significant digit. In order to compute '2X' (two times input), we first duplicate each digit of the input and record it using 5 bits

– one carry (1 bit) and one digit (4-bits). For a 16-digit input, it generates 16 digits and 16 carry bits. In the second step, the carry bits are added; the carry is not propagated to the next digit. The generation of 5X (five times input) is performed by first computing 10X (ten times input) and a simple division operation. The overall architecture of the partial product generator (PPG) is shown in Fig 8.

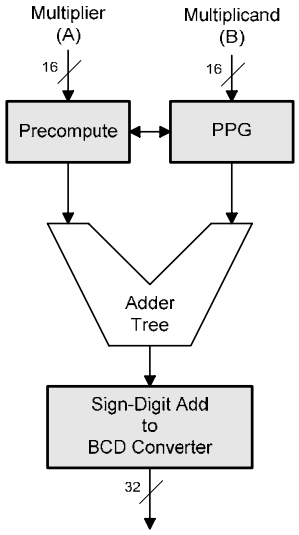


Figure 7. 16-digit general purpose BCD multiplier

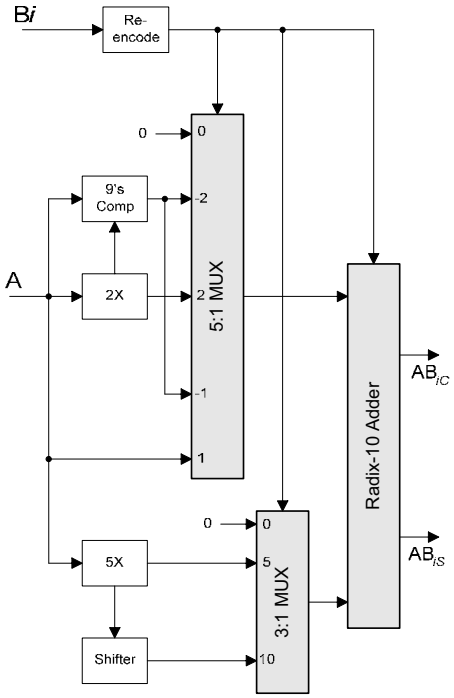


Figure 8. Block diagram of the partial product generator (PPG)

The partial products are added using adder tree and converted to BCD. For this conversion, we have used an efficient signed-digit decimal adder [18] which has the benefit of carry-free addition; however a carry-propagation adder (CPA) must be used to transform the signed-digit sum into an unsigned sum. The operation is shown in Fig. 9 and described as follows by (12):

$$S_i = u_i - 10 \times c_i + c_{i-1} \qquad (12)$$

where $c_i = \begin{cases} -1 & if \quad u_i < -1 \\ +1 & if \quad u_i > +1 \\ 0 & otherwise \end{cases}$.

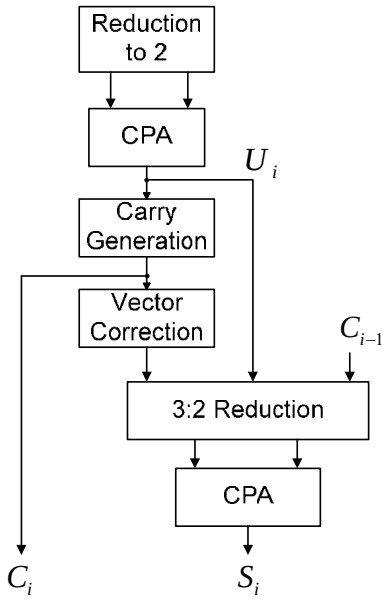


Figure 9. Final conversion to BCD using signed-digit adder

V. PERFORMANCE EVALUATION

The architecture of the 64-bit log converter has been prototyped using Verilog and synthesized onto Xilinx Virtex2p FPGA (xc2vp30ff1152-7). The breakdown of the cost of different units is shown in Table I. It can be seen that the Power-10 unit (including a combinational decimal multiplier) consumes the most resources of the entire system.

TABLE I. HARDWARE COST OF THE PROPOSED 64-BIT DECIMAL LOG CONVERTER

Module	Sub-module	Bit length	Reg.	Logic cells
Core Unit (inc. Power-10)	Others	64	269	213
	Decimal mult	64	0	6,403
Controller and others		16	73	136
Total		64	342	6,752

A. Error analysis

For the targeted 64-bit decimal FP applications, the proposed log converter must be able to achieve the minimum accuracy (i.e., 32-bit binary precision as defined in [31]) to guarantee correct operation. In order to compute the maximum error of the converted log result, we have performed an error analysis, where a long test vector comprising of 100 16-digit positive decimal numbers (ranging from 0000000000000000 to

9999999999999999 with arbitrary position of decimal point) is used. The error is computed taking a precision of 22-digit (written in Matlab) as reference. Fig. 10 shows the normalized error plot (in log scale) for only five arbitrary samples for different internal digit precision. In the x-axis, the number of decimal digits retained after one power-10 iteration is shown. It can be seen from the plot that the algorithm produces less error if higher number of digits is retained. The proposed architecture is based on a 64-bit binary precision, and thus can only handle up to 16-decimal digits. As a result, the error, as seen in the plot, is fixed after 16 digits. The maximum normalized error at this precision is estimated to be 3.53×10^{-14}.

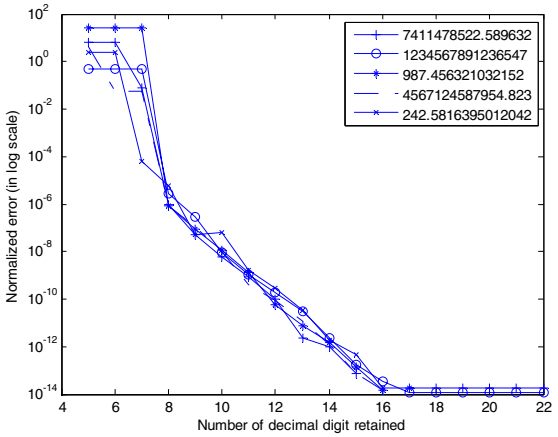


Figure 10. Normalized error of log converter for different precision

B. Hardware comparisons

Table II compares the results of the proposed log converter with other similar designs. In the cases, where the information of logic cells is not available, we have computed it from the count of slices (e.g., one slice is equivalent to two logic cells). The latency indicates the minimum number of clock cycles required to produce decimal 1-digit output. In all cases, similar Xilinx FPGA technology is used and the maximum absolute error along with the number of digit accuracy is presented.

We start with two binary designs [12][13] which give us a rough estimate about the relative comparison between binary and radix-10 designs. Note that, the results presented in [12] and [13] are based on the synthesis of HDL code that was originally generated automatically by C++ program.

The work in [7] is based on a curve fitting (linear approximation) algorithm. Due to the use of look-up-table (i.e., ROM mapping), the design takes only 1 cc to produce the output, but the crude approximation algorithm results in large error (e.g. maximum absolute error is 0.09 with only 3 digit accuracy). There are 16 partition regions used to achieve such accuracy and a complex error-correction circuitry is required at the end. The work in [17] is based on a digital recurrence algorithm. With the use of large look-up-tables and complex mapping, the error is largely minimized, but at the expense of low operational frequency and reduced throughput (e.g. latency is 18 cc), which makes the scheme unsuitable for time critical applications. The decimal digit accuracy is 14, which is still lower than the proposed algorithm. [17] also discusses briefly the extension to 64-bit design, but the cost of actual FPGA implementation is not reported. As a result, we have estimated the cost from the 7-digit core. In [22], the authors have presented an iterative scheme with a sequential multiplier unit that results in consuming relatively low recourses; however the sequential nature of the architecture and the un-pipelined operation limit the performance by yielding a very large latency and low throughput.

Compared to all existing designs, the proposed scheme has much lesser computation error and higher digit accuracy (very naturally as it uses higher precision), lesser hardware cost, and higher frequency of operation. Compared to [22], the hardware cost of the proposed 64-bit scheme is higher (and so is the estimated power consumption) because of the two following reasons: (1) use of a combinational multiplier; (2) use of a much higher digit precision. However, the (minimum) latency is 4 cc which makes the proposed scheme very suitable for time and accuracy critical applications. The computation algorithm is generalized and scalable, which means that the architecture can be extended for *decimal*128 format without causing large increase in the complexity and hardware cost – this is another advantage of the proposed scheme. As an example, for compliance with the IEEE754-2008 *decimal*64 format, [17] requires a significant increase in two LUTs from 14-digit to 34-digit, and moderate increase in other processing blocks with a large increase in latency by at least two times.

VI. CONCLUSION

The paper presents a fast algorithm and efficient implementation for computing decimal logarithm using 64-bit floating-point arithmetic that complies with the IEEE754-2008 standard. The algorithm is based on a digit-by-digit iterative computation that does not require look-up tables, curve fitting, decimal-binary conversion, or division algorithms. The final logarithmic output is very accurate with a maximum absolute error of $3.53\text{x}10^{-14}$; no correction or rounding circuitry is required that makes the scheme suitable for timing and accuracy critical applications. The architecture is generalized and scalable – can be extended for *decimal*128 format. Future research is directed towards such extension, as well as the VLSI implementation of the algorithm.

TABLE II. HARDWARE COMPARISONS OF DIFFERENT LOG CONVERTERS

	Base	Bit-length (binary)	Scheme	Slices	Logic cells	Freq. (MHz)	Power (mW)	Latency (cc)	No. of digit accuracy	Max. abs. error
Detry et al. [12] [1]	2	31	ROM	881	1,762	11.4	--	30	--	--
Detry et al. [13] [1]	2	31	ROM	1,893	2,736	14.5	--	16	--	--
Dongdong et al.[7]	10	32	Curve fitting	999	1,998	50.9	108	1	3	0.9×10^{-1}
Dongdong et al.[17] [2]	10	32	Digit recurrence	2,842	5,684	47.7	--	18	14	0.66×10^{-14}
Ramin et al. [22] [3]	10	32	Iterative	526	1,053	44	79	40	10	0.1×10^{-11}
Dongdong et al.[17]	10	64 [4]	Digit recurrence	4,603	9,602	--	--	36	--	--
Proposed	10	64	Iterative	3,376	6,752	51	125.7	4	16	3.53×10^{-14}

[1]using Xilinx VirtexII FPGA (xc2v1000-4) device; [2]the max. abs. error reported is based on theoretical calculation; [3]using VirtexII FPGA (xc2v1000-6) device; [4]the estimated hardware count for 64-bit, since the actual cost was not reported

ACKNOWLEDGMENT

The authors would like to acknowledge the Natural Science and Engineering Research Council of Canada (NSERC) for its support to this research work.

REFERENCES

[1] M. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," Proc. of the IEEE Symposium on Computer Arithmetic, pp. 104-111, 2003.
[2] Wikipedia, [Online], Available: http://en.wikipedia.org/wiki/Logarithm, November 2009.
[3] "IBM Power6", IBM Corporation, May 2007.
[4] J. Mitchell, "Computer Multiplication and Division Using Binary Logarithms", IRE Trans. Electron. Computer, pp. 512-517, 1962.
[5] D. Kostopoulos, "An Algorithm for the Computation of Binary Logarithms", IEEE Trans. on Computers, vol. 40, no. 11, pp. 1267-1270, 1991.
[6] The IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008), IEEE Computer Society, Aug 2008.
[7] D. Chen, Y. Choi, Li Chen, D. Teng, K. Wahid, S. Ko, "A Novel Decimal-to-decimal Logarithmic Converter", Proc. of the IEEE Int. Symposium on Circuits and Systems, pp. 688-691, 2008.
[8] H. Lo and J. Chen, "A Hardwired Generalized Algorithm for Generating the Logarithm Base-k by Iteration," IEEE Trans. Computer, vol. C-36, pp.1363 – 1367, 1987.
[9] T. Lang and A. Nannarelli, "A Radix-10 Combinational Multiplier", Proc. of the Asilomar Conference on Signals, Systems and Computers, pp. 313-317, 2006.
[10] H. C. Neto and M. P. Vestias, "Decimal Multiplier on FPGA Using Embedded Binary Multipliers", Proc. of the Int. Conf. on Field Programmable Logic and Applications, pp.197-202, 2008.
[11] M. Ercegovac, "Radix-16 Evaluation of Certain Elementary Functions", IEEE Trans. on Computers, vol. C-22(6), pp. 561–566, 1973.
[12] J. Detrey, F. Dinechin, and X. Pujol, "Return of the Hardware Floating-Point Elementary Function", Proc. of the IEEE Symposium on Computer Arithmetic, pp. 161-168, 2007.
[13] J. Detrey and F. de Dinechin, "A Parameterizable Floating-point Logarithm Operator for FPGAs", Proc. of the 39th Asilomar Conf. on Signals, Systems & Computers, pp. 1186-1190, 2005.
[14] P. T. P. Tang, "Table-driven Implementation of the Logarithm Function in IEEE Floating-point Arithmetic", ACM Trans. on Mathematical Software, vol. 16(4), pp. 378 – 400, 1990.
[15] C. Wrathall and T. C. Chen, "Convergence Guarantee and Improvements for a Hardware Exponential and Logarithm Evaluation Scheme", Proc. of the IEEE Symposium on Computer Arithmetic, pp. 175–182, 1978.
[16] W. Wong and E. Goto, "Fast Hardware-based Algorithms for Elementary Function Computations using Rectangular Multipliers", IEEE Trans. on Computers, vol. 43(3), pp. 278–294, 1994.
[17] Dongdong Chen, Yu Zhang, Younhee Choi, Moon Ho Lee, Seok-Bum Ko, "A 32-bit Decimal Floating-Point Logarithmic Converter," Proc. of the IEEE Symposium on Computer Arithmetic, pp. 195-203, 2009.
[18] J. Rebacz, E. Oruklu, and J. Saniie, "Performance Evaluation of Multi-Operand Fast Decimal Adders", Proc. of the IEEE Int. Midwest Symposium on Circuits and Systems, pp. 535-538, 2009.
[19] E. Schwarz and S. Carlough, "Power6 decimal divide," Proc. of the IEEE Int. Conf. on Application-specific Systems, Architectures and Processors, pp. 128-133, 2007.
[20] C. Webb, "IBM z10: The next-generation mainframe microprocessor," IEEE Micro, vol. 28, no. 2, pp. 19–29, 2008.
[21] A. Tsang and M. Olschanowsky, "A study of database 2 customer queries," IBM Santa Teresa Laboratory, San Jose, CA, USA, Tech. Rep. TR-03.413, Apr. 1991.
[22] R. Tajallipour, D. Teng, S-B Ko, and K. Wahid, "On the Fast Computation of Decimal Logarithm", Proc. of the IEEE Int. Conf. on Computer and Information Technology, pp. 32-36, 2009.
[23] BigDecimal, [Online], Available: http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html, November 2009
[24] R. Kenney and M. Schulte, "High-speed multioperand decimal adders," IEEE Trans. on Computers, vol. 54, no. 8, pp. 953–963, 2005.
[25] I. D. Castellanos and J. E. Stine, "Compressor trees for decimal partial product reduction," Proc. of the 18th ACM Great Lakes Symposium on VLSI, pp. 107–110, 2008.

[26] A. Bayrakci and A. Akkas, "Reduced delay BCD adder," Proc. of the IEEE Int. Conf. in Application-specific Systems, Architectures and Processors, pp. 266–271, 2007.
[27] L. Dadda, "Multioperand parallel decimal adder: A mixed binary and bcd approach," IEEE Trans. on Computers, vol. 56, no. 10, pp. 1320–1328, 2007.
[28] D. Chen, Y. Zhang, D. Teng, K. Wahid, M. Lee, and S-B. Ko, "A New Decimal Antilogarithmic Converter", Proc. of the IEEE Int. Symposium on Circuits and Systems, pp. 445-448, 2009.
[29] A. Duale, M. Decker, H. Zipperer, M. Aharoni, and T. Bohizic, "Decimal Floating-Point in z9: An Implementation and Testing Perspective". Journal on IBM Res. and Dev., Jan 2007.
[30] M. Cowlishaw, "Densely Packed Decimal Encoding", IEEE Computers and Digital Techniques, pp. 102-104, 2002.
[31] The IEEE Standard for Binary Floating-Point Arithmetic (IEEE754-1985), IEEE Computer Society.

Ramin Tajallipour completed his B.Sc. in Electrical Engineering from Azad South Tehran University in 2006. After graduation with help of two other as a group made a testing ring instrument which achieved a patent number and already is using at well-known company, SAIPA automobile manufacturing company in Iran. Then in 2007, he joined to Qom power station in Iran as an automation expert and worked about 1.5 years. He started his M.Sc. program with Dr. Khan A. Wahid at Electrical Engineering department of University of Saskatchewan in fall 2008. He is currently working in "Digital Systems Research Group" and his research interest is in the field of video and image compression and processing, digital designing, FPGA, and real-time embedded systems.

Md. Ashraful Islam Md Ashraful Islam: Received his B.Sc. degree in Electrical and Electronic Engineering from the Bangladesh University of Engineering & Technology, Bangladesh in 2005.He is currently an M.Sc. candidate in Electrical & Computer Engineering Department at University of Saskatchewan. Prior to that, he worked as a Lecturer in Southeast University, Bangladesh, as a Radio planner in Grameenphone Ltd, Bangladesh & Telecom Malaysia in Bangladesh respectively. He is a member of "Digital Systems Research Group", at the College of Engineering, University of Saskatchewan. His research interests include high-performance digital circuit, FPGA and ASIC design, and VLSI architectures for image processing. He has designed and co-designed two IC chips of area and power efficient architectures in 0.18um CMOS (TSMC) technology that has been reported to his publications.

Khan A. Wahid earned his B.Sc. degree from Bangladesh University of Engineering and Technology (BUET) in 2000. He received his M.Sc. (2003) and Ph.D. (2007) from the University of Calgary. He was the recipient of numerous prestigious awards and scholarships including the most distinguished "Killam Scholarship" and the "NSERC Canada Graduate Scholarship" for his doctoral research.

Dr. Wahid has been working as an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Saskatchewan since July 2007. He has authored over 40 peer-reviewed journal and international conference papers in the field of digital arithmetic techniques, FPGA and ASIC design, real-time embedded systems, video and image compression, and biomedical imaging systems. He has been serving as a reviewer for the IEEE Transactions on Circuit and Systems for Video Technology, Biomedical Engineering Online, EURASIP Journal on Signal Processing, and Elsevier Journal on Computers and Electrical Engineering since 2006. He is a registered as a Professional Engineer in the province of Saskatchewan, Canada, and a Member of the Institute of Electrical and Electronics Engineers (IEEE).