Algorithm and Architecture for on-line Decimal Powering Computation

Mahmoud Y. Hassan*, Tarek ElDeeb* and Hossam A. H. Fahmy†
*SilMinds, Maadi, 11431, Helwan, Egypt
Email: mahmoud.yousef@silminds.com
†Electronics and Communications Department, Cairo university, Egypt
Email: hfahmy@stanfordalumni.org

***Abstract*—An architecture for the computation of a decimal powering function is presented in this paper. The algorithm consists of a sequence of overlapped operations: 1) digit recurrence logarithm, 2) sequential multiplication, and 3) on-line antilogarithm. A correction scheme is introduced between the overlapped operations to guarantee correct on-line calculations. Execution times are estimated for decimal64 and decimal128 formats of the IEEE 754-2008 standard for floating point arithmetic.**

I. Introduction

Interest in decimal arithmetic increased considerably in recent years. There are many commercial demands for Decimal Floating Point (DFP) arithmetic operations such as tax calculations, currency conversion and phone billing. Powering (X^Y) is an important function in financial applications such as interest rate calculations. Decimal Powering function pow(x,y)= x^y where $x \in [-\infty, +\infty]$ and $y \in [-\infty, +\infty]$ is defined in the new IEEE 754-2008 standard [1] and is implemented in this work. The standard also defines other functions such as: compound(x,n)= $(1+x)^n$, pown(x,n)= x^n where n is an integer, powr(x,y)= x^y (derived by considering only exp($y \times log(x)$) where $x \in [0, +\infty]$ and $y \in [-\infty, +\infty]$.

Most elementary functions are implemented using software libraries due to the advantage of using large look up tables and providing more accurate results. However, software algorithms are not suitable for numerical intensive and real time applications due to their slow operation, 100 to 1000 slower than what can be implemented in hardware [2]. Therefore, new algorithms and hardware implementations for elementary decimal floating point arithmetic functions have been introduced recently [3] [4] [5]. The accurate computation of the floating point powering function is difficult and requires wider calculating precisions as explained by the Table Maker's Dilemma [6]. A composite iterative algorithm for the computation of binary powering function is introduced in [7]. In this paper, we give a detailed description of an iterative algorithm for the computation of the decimal powering function (X^Y). Three overlapped operations: *logarithm*, *multiplication* and *antilogarithm* are done iteratively combined with a correction scheme between subsequent operations. The final result is faithfully rounded, that is the result is one of the immediate floating point neighbors to the exact result. To the best of the authors knowledge, this is the first work to implement a decimal floating point powering function on hardware.

The paper is organized as follows: section II introduces the decimal powering. Section III shortly presents the testing and implementation results while section IV gives the conclusions.

The project "Promoting Egypt as the first decimal Arithmetic Intellectual property cores provider for financial applications in the world" (grant number C2/S1/163) is funded by the RDI Programme through the EU-Egypt Innovation Fund (EEIF). The RDI Programme is a programme of the Ministry of Higher Education and Scientific Research funded by the European Union.

II. Algorithm

The algorithm is based on the mathematical identity:

$$X^Y = 10^{Y log_{10} X} \qquad \text{for } X > 0 \tag{1}$$

For $X < 0$, invalid operation is signaled out for finite non-integer Y. A negative final result is obtained for $X < 0$ and an odd integer Y. For $X < 0$ and an even integer Y, the final result is positive. Both inputs X and Y are decimal floating point operands and their absolute values are denoted $\hat{M}_x 10^{E_x}$ and $\hat{M}_y 10^{E_y}$ respectively. Hence,

$$\begin{aligned} X^Y &= 10^{\hat{M}_y 10^{E_y} log_{10}(\hat{M}_x 10^{E_x})} \\ &= 10^{\hat{M}_y 10^{E_y}(log_{10}(\hat{M}_x)+E_x)} \end{aligned} \tag{2}$$

And when both $\hat{M}_x$ and $\hat{M}_y$ are normalized, then

$$X^Y = 10^{M_y 10^{E_y+L}(log_{10}(M_x)+E_x+K)} \tag{3}$$

Where L is the leading zero count of $\hat{M}_y$ and K is the integer part of the logarithm's result when normalizing $\hat{M}_x$ and can be calculated by subtracting leading zero count of $\hat{M}_x$ from its precision. Thus, to calculate the final result, three composite operations have to be performed consecutively, *logarithm*, *multiplication* and *antilogarithm*. Moreover, in order to reduce the latency of the powering function, the sub operations have to be computed on-line [7]. That is, every stage starts its operation based on a calculated digit in the previous stage. Special data NaN, Inf and 0 are handled according to the IEEE standard's specifications for floating point arithmetic [1].

A. Logarithm (log_{10})

Logarithm is computed iteratively using the digit recurrence algorithm with selection by rounding. Pineiro et al. [8] introduced an algorithm for the computation of logarithms for binary floating point numbers. Chen et al. [3] adapted the same algorithm to decimal floating point numbers.

TABLE I
LOOK UP TABLE FOR e_1 SELECTION

Ranges of m	e_1
$[0.96, 1.00)$	0
$[0.88, 0.95]$	1
$[0.81, 0.87]$	2
$[0.76, 0.80]$	3
$[0.70, 0.75]$	4
$[0.66, 0.69]$	5
$[0.62, 0.65]$	6
$[0.59, 0.61]$	7
$[0.56, 0.58]$	8
$[0.50, 0.55]$	9

1) Algorithm: A logarithmic result $log_{10}(x)$ can be achieved according to [3] based on the mathematical identity:

$$log_{10}(m) = log_{10}(m \prod f_j) - \sum log_{10}(f_j) \tag{4}$$

So $log_{10}(m) = -\sum_{j=1}^{\infty} log_{10}(f_j)$ if the first term converges to zero. f_j is defined as $f_j = 1 + e_j 10^{-j}$, this form allows the use of a *shift-and-add* implementation. The corresponding recurrences for computing the logarithm are:

$$E(j+1) = E[j](1 + e_j 10^{-j}) \tag{5}$$
$$L(j+1) = L(j) - log_{10}(1 + e_j 10^{-j}) \tag{6}$$

Where, $E[1] = m$ and $L[1] = 0$. The digits e_j are selected so that $E(j+1)$ converges to 1. To have a selection function for e_j, a scaled remainder is defined as ,

$$W[j] = 10^j (1 - E[j]) \tag{7}$$

Substituting (7) in (5) yields,

$$W[j+1] = e_j W[j] 10^{-j+1} + 10(W[j] - e_j) \tag{8}$$

And e_{j+1} digits are selected by rounding $W[j+1]$ to the integer part in every iteration.

$$e_{j+1} = round(W[j+1]) \tag{9}$$

The logarithmic result is achieved through sequential additions according to (6), where the values of $log(1 + e_j 10^{-j})$ are stored in a look up table. The series expansion of the logarithm function :

$$log_{10}(1+x) = \frac{(x - \frac{x^2}{2} + \dots)}{ln(10)} \tag{10}$$

is used to reduce the size of this look up table. After iteration $j = k$ the values of $log_{10}(1 + e_j 10^{-j})$ can be approximated by $\frac{e_j 10^{-j}}{ln(10)}$. The selection of the k value depends on achieving the constraint $\frac{e_j^2 10^{-2j}}{2ln(10)} < 10^{-n}$, where n is the required accurate precision to be calculated. According to [3], the value of e_1 is obtained by look up table (I), where the input is in the range $0.5 \leq m < 1$ and the number in the range $0.1 \leq m < 0.5$ needs to be adjusted. The adjustment can be done by multiplying the input with (2, 3 or 5) and adding a value of ($log(2)$, $log(3)$ or $log(5)$) to the logarithm result. e_{j+1} values are selected by rounding the scaled remainder for iterations $j \geq 2$.

2) Error analysis:

a) Inherent Error of Algorithm: This error results from the difference between the logarithm result obtained from finite iterations and the exact result obtained from infinite iterations. Since the decimal logarithmic result is achieved after n^{th} iteration, ε_i can be defined as:

$$\varepsilon_i = -\sum_{j=n}^{\infty} log_{10}(1 + e_j 10^{-j}) \tag{11}$$

Since e_j represents the error between two successive iterations and $e_j \in [-9, 9]$,we choose the worst cases ($e_j = 9\ or\ -9$) to analyze maximum ε_i

$$\varepsilon_i = -\sum_{j=n}^{\infty} log_{10}(1 \pm 9 \times 10^{-j}) \tag{12}$$

According to (12), maximum ε_i is in the range:

$$-4.34 \times 10^{-n} \leq \varepsilon_i \leq 4.34 \times 10^{-n} \tag{13}$$

b) Approximation Error: The approximate value $\frac{e_j 10^{-j}}{ln(10)}$ is used to estimate $log(1 + e_j 10^{-j})$ from the k^{th} to the n^{th} iteration. According to the series expansion of logarithm function in (10), Since the value of $log(1+x)$ is approximatd by $\frac{x}{ln(10)}$ after the k^{th} iteration. Putting $x = e_j 10^{-j}$, this produces an approximation error, ε_a:

$$\varepsilon_a = \sum_{j=k}^{n} \frac{-\frac{(e_j 10^{-j})^2}{2} + \frac{(e_j 10^{-j})^3}{3} - \dots}{ln(10)} \tag{14}$$

Since

$$\sum_{j=k}^{n} \frac{\frac{(e_j 10^{-j})^3}{3} - \dots}{ln(10)} \ll 10^{-n} \tag{15}$$

So higher order terms can be neglected with respect to $-\frac{(e_j 10^{-j})^2}{2ln(10)}$,

$$\varepsilon_a \approx \sum_{j=k}^{n} -\frac{(e_j 10^{-j})^2}{2ln(10)} \tag{16}$$

Considering the worst cases ($e_j = 9\ or\ -9$) we obtain the maximum ε_a:

$$\varepsilon_a \leq 1.78 \times 10^{-(2k-1)} \tag{17}$$

c) Quantization Error: Since only those intermediate values who have finite precisions are operated in the hardware-oriented algorithm, three quantization errors occur. First, the logarithm results are achieved by accumulating the n-digits rounding values of $-log_{10}(1 + e_j 10^{-j})$ from the 1^{st} to the k^{th} iteration. In each iteration, the maximum rounding error of $-log_{10}(1 + e_j 10^{-j})$ is 0.5×10^{-n} therefore the maximum ε_{q1} is:

$$\varepsilon_{q1} \leq \sum_{j=1}^{k} 0.5 \times 10^{-n}$$
$$\leq 0.5k \times 10^{-n}$$

Second, the logarithm results are achieved by accumulating the $n-$digit rounded values of $\frac{-e_j 10^{-j}}{ln(10)}$ from the k^{th} to the

n^{th} iteration. Since the values of $\frac{-e_j}{ln(10)}$ are precomputed and stored in a Look Up Table (LUT) for $e_j \in [-9, 9]$ followed by a shifter to achieve the multiplication by 10^{-j}, the maximum rounding error of $\frac{-e_j 10^{-j}}{ln(10)}$ is 0.5×10^{-n} therefore the maximum ε_{q2} is:

$$\begin{aligned} \varepsilon_{q2} &\leq \sum_{j=k+1}^{n} 0.5 \times 10^{-n} \\ &\leq 0.5 \times (n-k-1) \times 10^{-n} \end{aligned}$$

Third, the logarithm result $log_{10}(M_x^{'})$ is adjusted by a finite n-digit rounded constant (0, $log_{10}(2)$, $log_{10}(3)$ or $log_{10}(5)$) in the last iteration, so the maximum ε_{q3} is:

$$\varepsilon_{q3} \leq 0.5 \times 10^{-n}$$

Therefore, the maximum quantization error ε_q is:

$$\begin{aligned} \varepsilon_q &\leq \varepsilon_{q1} + \varepsilon_{q2} + \varepsilon_{q3} \\ &\leq 0.5k \times 10^{-n} + 0.5 \times (n-k-1) \times 10^{-n} + 0.5 \times 10^{-n} \\ &\leq 0.5 \times n \times 10^{-n} \end{aligned}$$

d) Error Estimation: Since the final logarithm result is faithfully rounded, it has a maximum rounding error of $\varepsilon_r = 1 \times 10^{-n}$. Hence the total error introduced is due to ε_i, ε_a, ε_q and ε_r:

$$\begin{aligned} E_{total} &= \varepsilon_i + \varepsilon_a + \varepsilon_q + \varepsilon_r \qquad (18) \\ &= (5.34 + 0.5n) \times 10^{-n} + 1.78 \times 10^{-(2k-1)} \end{aligned}$$

According to equation (10), approximation can be done after the k^{th} iteration where,

$$\begin{aligned} k &= -\frac{1}{2} log_{10}\left(\frac{2ln(10) \times 10^{-n}}{81}\right) \qquad (19) \\ &= \frac{n}{2} + C \end{aligned}$$

And C is a constant equal to $-\frac{1}{2}(log_{10}(2ln(10)) - log_{10}(81))$. Substituting (19) into (18) leads to:

$$E_{total} = (5.34 + 0.5n) \times 10^{-n} + 1.78 \times 10^{-2C+1} \times 10^{-n}$$

Hence, $E_{total} = (A + 0.5n) \times 10^{-n}$ where A is a constant equal to $(5.34 + 1.78 \times 10^{-2C+1})$. Hence, $E_{total} \times 10^n$ is proportional to the calculation precision n. Moreover, $E_{total} < 0.5 \times 10^{n-2}$ up to $n = 88$ digits. That is, if a calculating precision of $n < 88$ is used, a total maximum error of $0.5 \times 10^{n-2}$ is guaranteed according to this error analysis.

3) Architecture: The hardware implementation of the logarithmic converter circuit includes two stages. *Stage 1* is to obtain e_{j+1} with selection by rounding. After e_{j+1} is achieved, the logarithm result is produced in *stage 2*. The block diagrams illustrating the architecture of *stage 1* and *stage 2* are shown in figure(1) and figure(2) respectively. The used *Multiplier* and *CPA* are both implemented based on [9] [10]. In the first stage, a multiplier by one digit using Signed Digit (SD) recoding for e_{j+1} produces two partial products. Two barrel shifters are used to achieve the first term of equation(8) $e_j W[j] 10^{-j+1}$. The other term includes an addition operation, but it can be implemented by a simple logic as e_{j+1} is the integer one digit resulting from rounding the scaled remainder $W[j]$. Thus, the result of the subtraction is always the fraction part of $W[j]$. Table(II) illustrates the operation by an example, where either the fraction of $W[j]$ or its complement is selected when the sign of $W[j]$ is 0 or 1 respectively. In the second stage, e_j values are used to choose the value of $log(1 + e_j 10^{-j})$ from the look up table which will be added sequentially to achieve the final logarithm result. Note that e_1 is obtained from look up table (I). In the approximation phase, a multiplication by $\frac{1}{ln(10)}$ is needed according to equation(10). This operation can be achieved by storing the values of $\frac{e_j}{ln(10)}$ in a ROM followed by a shifter to compute $\frac{e_j 10^{-j}}{ln(10)}$.

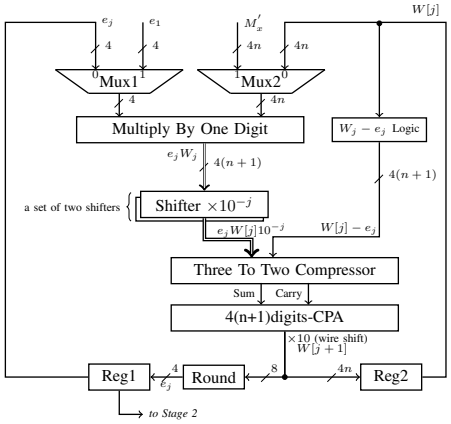

Fig. 1. Architecture of Logarithm's First stage

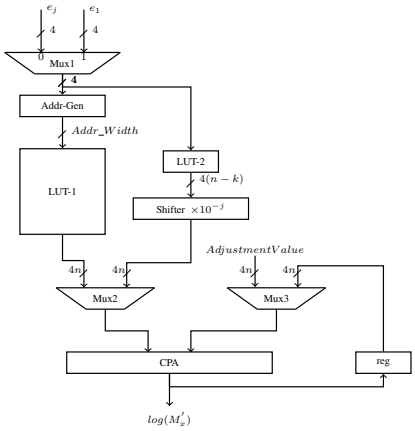

Fig. 2. Architecture of Logarithm's Second stage

4) Logarithm Extension and counting leading zeros: Recall that our proposed power computing algorithm is based on (3):

$$X^Y = 10^{M_y 10^{E_y + L}(log_{10}(M_x) + E_x + K)}$$

where both M_x and M_y are normalized. If $E_x + k$ happens to be $zero$ and M_x is all 9s, then

$$log_{10}(0.\underbrace{999\cdots999}_{p}) = 0.\underbrace{000\cdots000}_{p}xxxx$$

TABLE II
$W[j] - e_j$ LOGIC

cases	$W[j] - e_j$	W[j] sign	$(W[j] - e_j)$ fraction
$-2.4 \rightarrow -2$	-0.4	1	complement
$-2.7 \rightarrow -3$	0.3	1	complement
$2.4 \rightarrow 2$	0.4	0	fraction
$2.7 \rightarrow 3$	-0.3	0	fraction

TABLE III
NUMERICAL EXAMPLE

$log_{10}(0.2847737800173746)$		
j	e_j	$result$
1^{st}	2	$0.\mathbf{5}66302500767\cdots$
2^{nd}	-3	$0.5\mathbf{4}3074235033\cdots$
3^{rd}	6	$0.54\mathbf{5}672215753\cdots$
4^{th}	-4	$0.5454\mathbf{9}8463207\cdots$
5^{th}	0	$0.5454\mathbf{9}8463207\cdots$
6^{th}	4	$0.545500\mathbf{0}0382\cdots$
7^{th}	-5	$0.5454599\mathbf{9}83235\cdots$

Now, assume a positive value of E_y+L, then log result will be right shifted by E_y+L amount before feeding the result to the next stage. So, in the case of positive E_y+L, the leading zeros appearing in log operation result need to be eliminated and in the same time an extended precision needs to be calculated. In the case of decimal-64 and decimal-128 formats, the maximum number of leading zeros that may appear in the result is 16 and 34 zeros respectively.

5) Alternating behavior of log results: Here we show how the log stage's result can affect the subsequent stage due to on-line operation. Table (III) shows the iterations results for $log_{10}(0.2847737800173746)$. One digit is fed on-line to the next stage every iteration. For the example shown , in 4^{th} iteration, the on-line digit 4 switches to 5 in 6^{th} iteration and switches again to 4 in 7^{th} iteration. This switching action has to be accounted for in the next stage iteratively and is handled in section(II-B).

B. Sequential multiplication

Multiplication is the second stage in the decimal power computing algorithm. It is done from left to right on-line and iteratively. The multiplication operation is based on Signed Digit (SD) format to reduce the number of generated multiples [9]. The operands of the multiplication are: input M_y and $log_{10}(M_x) + E_x + k$, where both inputs M_x and M_y are normalized. Since M_x is normalized, the result of $log_{10}(M_x)$ is always a negative fraction. Thus, depending on the sign of E_x+k, which is always integer, $log_{10}(M_x)$ or its complement is combined with E_x+k to achieve $log_{10}(M_x)+E_x+k$. The log-stage outputs a correct digit with an error $< 0.5ulp$ per iteration. Hence, we choose to use the SD redundant format for log-stage results. As shown in table IV, where *l* is a $digit < 5$ and *h* is a $digit \geq 5$, two numbers which differ in *0.5ulp* are coded to the same value. For example, $\underline{2}6$ and $\underline{3}2$ are both coded into 3. This facilitates the on-line operation of $mult$-stage after log-stage as follows:

TABLE IV
SD RECODING

digits		SD	digits		SD
0	$0l$	0	5	$5l$	$\overline{5}$
	$0h$	1		$5h$	$\overline{4}$
1	$1l$	1	6	$6l$	$\overline{4}$
	$1h$	2		$6h$	$\overline{3}$
2	$2l$	2	7	$7l$	$\overline{3}$
	$2h$	3		$7h$	$\overline{2}$
3	$3l$	3	8	$8l$	$\overline{2}$
	$3h$	4		$8h$	$\overline{1}$
4	$4l$	4	9	$9l$	$\overline{1}$
	$4h$	5		$9h$	$\overline{0}$

1) Sources of error and correction scheme: The numbers which can result in a multiplication error, due to a change in a log-stage result, are the values which are coded to the same SD value but with opposite signs such as: $4h$, $5l$, $9h$, $0l$. Also, the values at the boundary of 0.5 such as: $3l, 3h$...etc

a) Same value but opposite sign: The ten's complement of the partial product can be formed of its nine complement with a predicted tail digit of 1 in the previous partial product. This tail digit value is decided upon the next digit in the multiplicand (log-stage value). If the value of log-stage result switches from $999\cdots999$ to $1000\cdots000$ (a wrong prediction of a tail digit), a tail digit of 1 needs to be eliminated. In the same way, a tail digit of 1 needs to be added if the log-stage result switched from $1000\cdots000$ to $0999\cdots999$ (missed tail digit). This error can be corrected by using two shift registers with an initial values of $999\cdots999$ and $000\cdots0001$. These values are added to the result of multiplication whenever this case is detected. In case of $5l$ and $4h$, no correction is needed because 5 and its ten's complement have the same value.

b) Boundary of 0.5*:* This case can be best shown by an example. Assume a result from log-stage while operation to be $\underline{64}9\cdots$ which is switched to $6\underline{50}\cdots$ in the next iteration, and is needed to be multiplied by $M_y = 1$. In the first iteration, the result of the multiplication will be $\overline{4}$ (which can be thought of as 6). In the second iteration the result will be $\overline{4} + 0.\overline{5}$ (which can be thought of as 5.5). Noting that the correct result is obtained in the second iteration, the actual result then is $\overline{3}.\overline{5}$(which can be thought of as 6.5). Hence, whenever this case is detected, the ten's complement of the next partial product is negated, that is, according to the same example, in the second iteration the result of multiplication will be $\overline{4}+0.5$ instead of $\overline{4}+0.\overline{5}$. In case of $\underline{55}0\cdots$ switching to $5\underline{49}\cdots$ no correction is needed because the value of 5 is equal to the value of $\overline{5}$.

2) Architecture: The architecture of the proposed sequential multiplier combined with the correction scheme for the on-line results is shown in figure(3).

Firstly, log-stage result is shifted left to obtain the current and next slice. Current and next slice (each slice is two digits) are passed to the SD recoding blocks. A comparison between current slice and a delayed (by one clock cycle) version of it is done to determine the sign of the current partial product according to the proposed correction scheme. The value of SD1 and comparator output select one of the generated PP

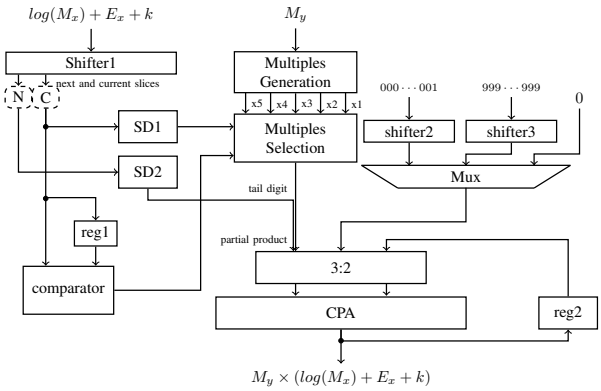

Fig. 3. Architecture of sequential multiplier

or its complement respectively. SD2 determines the value of the tail digit based on the next slice. A correction vector of $999\cdots999$ or $000\cdots001$ is selected to correct the error due to a wrong tail digit. Finally, A $ThreeToTwoCompressor$ followed by a CPA are used to obtain the value of $M_y \times (log_{10}(M_x) + E_x + k)$.

C. on-line Antilogarithm

A detailed implementation of a decimal antilogarithm circuit based on digit recurrence with selection by rounding can be found in [4]. A detailed error analysis for on-line binary antilogarithm can be found in [7]. A delay of 2-digits before calculation is chosen for our proposed implementation. The recurrence equations for the on-line antilogarithm are:

$$W[j+1] = 10W[j] - 10^{j+1}ln(1 + e_j 10^{-j}) + X_{j_{new}}10^{-1} \quad (20)$$

where,

$$W[j] = 10^j \times L[j] \quad (21)$$

And

$$L[j+1] = L[j] - ln(1 + e_j 10^{-j}) \quad (22)$$

$$E[j+1] = E[j] \times (1 + e_j 10^{-j}) \quad (23)$$

And, e_j is selected by rounding $W[j+1]$ to the integer part in every iteration.

1) Sources of error: The previous multiplication operation is carried out from left to right. Thus, in every iteration a new shifted partial product is added to the previous result. The current digit in the final result is maximally affected by one carry resulting from the nearest digit addition.

$$\begin{array}{r} 9\mathbf{9}999\ + \\ \mathbf{9}9999 \\ \hline \mathbf{10}9998\mathbf{9} \end{array}$$

Hence, a comparison which tracks any increment or decrement that occurs in the previous $multiplication$-stage digit is needed. Once a change is detected, the value of $X_{j_{new}}$ has to be adjusted to compensate the error due to a missed calculated carry or borrow. The correction of $X_{j_{new}}$ is carried out by adding (or subtracting) a value of one. Note that the value of $X_{j_{new}}$ is always multiplied by 10^{-1} so an integer part of '1' is combined with the new digit or the new digit ten's complement is used whenever an increment or decrement is detected respectively.

III. Testing and Results

X^Y is a bivariate function. So an exhaustive testing for all combinations of input is considered relatively impossible. A simple software model targeting the presented sources of errors is built and used to generate (200,000) test cases. IBM DecNumber library is used as reference. The test proved correct results for faithfully rounded decimal power function. The minimum number of iterations required for decimal powering computing are 40 and 60 clock cycles for decimal64 and decimal128 formats respectively. The maximum number of iterations required when an extended precision of logarithm is needed are 56 and 84 iterations for decimal64 and decimal128 formats respectively.

IV. Conclusion

In this paper, we have presented an on-line algorithm for decimal powering computation. Three overlapped operations are done iteratively: *logarithm*, *multiplication* and *antilogarithm*. A correction scheme between adjacent stages is introduced and explained.

References

[1] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, aug. 2008.

[2] M. Cowlishaw, "Decimal floating-point: algorism for computers," in *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, 15-18 2003, pp. 104 – 111.

[3] D. Chen, Y. Zhang, Y. Choi, M. H. Lee, and S.-B. Ko, "A 32-bit decimal floating-point logarithmic converter," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, 8-10 2009, pp. 195 –203.

[4] D. Chen, Y. Zhang, D. Teng, K. Wahid, M. H. Lee, and S.-B. Ko, "A new decimal antilogarithmic converter," in *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, 24-27 2009, pp. 445 –448.

[5] C. A. Marius Cornea. (2010, January) Intel decimal floating-point math library. Intel Corporation. [Online]. Available: http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library/

[6] V. Lefevre, J.-M. Muller, and A. Tisserand, "Towards correctly rounded transcendentals," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, 6-9 1997, pp. 132 –137.

[7] J.-A. Pineiro, M. Ercegovac, and J. Bruguera, "Algorithm and architecture for logarithm, exponential, and powering computation," *Computers, IEEE Transactions on*, vol. 53, no. 9, pp. 1085 – 1096, sept. 2004.

[8] ——, "High-radix logarithm with selection by rounding," in *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*, 2002, pp. 101 – 110.

[9] R. Raafat, A. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, and H. Fahmy, "A decimal fully parallel and pipelined floating point multiplier," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, 26-29 2008, pp. 1800 –1804.

[10] H. Fahmy, R. Raafat, A. Abdel-Majeed, R. Samy, T. ElDeeb, and Y. Farouk, "Energy and delay improvement via decimal floating point units," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, 8-10 2009, pp. 221 –224.