Algorithm and Architecture for Logarithm, Exponential, and Powering Computation

J.-A. Piñeiro, M.D. Ercegovac, *Fellow*, *IEEE*, and J.D. Bruguera, *Member*, *IEEE*

Abstract—An architecture for the computation of logarithm, exponential, and powering operations is presented in this paper, based on a high-radix composite algorithm for the computation of the powering function (X^Y). The algorithm consists of a sequence of overlapped operations: 1) digit-recurrence logarithm, 2) left-to-right carry-free (LRCF) multiplication, and 3) online exponential. A redundant number system is used and the selection in 1) and 3) is done by rounding except from the first iteration, when selection by table look-up is necessary to guarantee the convergence of the recurrences. A sequential implementation of the algorithm, with a control unit which allows the independent computation of logarithm and exponential, is proposed and the execution times and hardware requirements are estimated for single and double-precision floating-point computations. These estimates are obtained for radices from $r = 8$ to $r = 1,024$, according to an approximate model for the delay and area of the main logic blocks and help determining the radix values which lead to the most efficient implementations: $r = 32$ and $r = 128$.

Index Terms—Computer arithmetic, elementary function approximation, digit-recurrence algorithms, logarithm, exponential, powering, high-radix, LRCF multiplication, online arithmetic.

1 Introduction

Powering (X^Y), logarithm ($\log X$), and exponential (2^X) are important operations in computer 3D graphics, digital signal processing (DSP), scientific computing, artificial neural networks, logarithmic number systems (LNS), and multimedia applications [12], [16], [21], [26], [29]. As other elementary functions, such as square root, reciprocal square root, and trigonometric functions, they have been traditionally computed by software routines [3], [9], [10], [13]. These routines provide very accurate results, but are often too slow for numerically intensive or real-time applications. The timing constraints of these applications have led to the development of dedicated hardware for the computation of elementary functions, including the implementation of table-based algorithms [17], [25], [27], [28], functional iteration methods [11], [18], [20], and digit-recurrence algorithms [2], [7].

Accurately computing the floating-point powering function is considered difficult [17] and the prohibitive hardware requirements of a table-based implementation (note that X^Y is a 2-variable function) have led only to partial solutions, such as powering algorithms for a constant exponent p [21], [27] or for very low precision [12]. A direct implementation of a digit-recurrence algorithm for powering computation is not feasible due to its high intrinsic complexity.

In this paper, we give a detailed description of an optimized composite iterative algorithm for the computation of the powering function (X^Y), for a floating-point input operand $X = M_x 2^{E_x}$ and integer[1] b-bit operand Y, and extend it to powering operations with exponents of the type $Y = 1/q$, with q integer. An abbreviated previous version of our algorithm with integer exponent was presented in [23]. The final result, X^Y, was computed as $Z = M_z 2^{E_z} = e^{Y \ln(M_z)} 2^{Y E_z}$ through a sequence of overlapped operations. The first step consisted of computing $\ln(M_z)$ by using a high-radix ($r = 2^b$) digit-recurrence algorithm with selection by rounding [22]. An intermediate computation $Y_L \ln(M_z)$, with $Y_L = Y \log_2(e)$, was carried out by a high-radix left-to-right carry-free (LRCF) multiplication operation [5]. Another LRCF multiplication by $\ln 2$ was performed to guarantee the convergence of the algorithm and, as the last step, the exponential of the resulting product was computed by an online high-radix algorithm, with online delay $\delta = 2$ and selection by rounding [24].

In the optimized version of our algorithm, the final result X^Y can be computed directly as $Z = M_z 2^{E_z} = 2^{Y \log_2(M_x)} 2^{Y E_x}$, which allows avoiding the computation of $Y_L = Y \log_2(e)$ and eliminates the need for a second LRCF multiplication, reducing the overall latency by one cycle. The expressions of the recurrences in the computations of the logarithm and the exponential must be slightly modified and extra look-up tables storing $-l_j/\ln(2)$ and $-e_j/\ln(2)$ are now required.

In the stages computing such operations (logarithm and exponential), selection by table look-up is still performed in the first iteration to guarantee the convergence of both

- *J.-A. Piñeiro was with the University of Santiago de Compostela and is now with Intel Barcelona Research Center, Edif. Vertex II, c) Jordi Girona, 29-3A, 08034 Barcelona, Spain. E-mail: alex@dec.usc.es.*
- *M.D. Ercegovac is with the Department of Computer Science, University of California at Los Angeles, 4732E Boelter Hall, Los Angeles, CA 90095. E-mail: milos@cs.ucla.edu.*
- *J.D. Bruguera is with the Department of Electronic and Computer Engineering, University of Santiago de Compostela, 15782, Santiago de Compostela, Spain. E-mail: bruguera@dec.usc.es.*

Manuscript received 7 May 2003; revised 19 Dec. 2003; accepted 29 Jan. 2003. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0033-0503.

1. The computation of the powering operation with integer exponent in the range $[1, 128]$ is useful, for instance, in computer graphics lighting computations (the lighting specular component in the OpenGL graphics pipeline [12]).

algorithms, with the online delay of two cycles in the exponential scheme allowing addressing of the initial tables one cycle in advance and reducing the delay of the critical path in this stage.

A sequential architecture for powering computation was proposed in [23], with radix $r = 128$. Such an architecture can actually be implemented with any radix $r \geq 8$, although the overall hardware requirements increase with r and an analysis of the trade offs between area and speed is necessary for determining which radix values result in the most efficient implementations. We perform such an analysis in this paper for our optimized algorithm.[2]

The analysis performed is based on estimates obtained for single and double-precision computations and for radix values going from $r = 8$ to $r = 1,024$, according to an approximate model for the delay and area cost of the main logic blocks employed in the proposed architecture. The main results of our analysis are that a fast implementation can be obtained when using $r = 128$, but an implementation with $r = 32$ is more suitable for applications with tighter area constraints. There is no advantage in using values $r > 128$ since similar or slower execution times are achieved, with much higher hardware requirements.

Since the computations of logarithm and exponential are included in our algorithm for powering computation, some minor changes can be made to the architecture to allow for the independent computation of logarithm and exponential, with lower latencies than powering, making the architecture an interesting alternative when implementing a dedicated unit for elementary function computation.

The rest of this paper is structured as follows: We first focus on the algorithm for powering computation with integer exponent Y, giving a detailed description of the algorithm and its error analysis in Section 2 and detailing the architecture implementing the algorithm in Section 3, with an overview of the high-radix logarithm, LRCF multiplication, and online high-radix exponential stages. The evaluation of the proposed architecture for single and double-precision floating-point results, for radix values going from $r = 8$ to $r = 1,024$, is presented in Section 4. In Section 5, we explain the minor modifications to be made to the proposed architecture in order to allow for the independent computation of logarithm and exponential. We show how to extend our algorithm for powering computation to exponents of the type $Y = 1/q$, with q integer, in Section 6. Finally, the main contributions made in this paper are summarized in Section 7.

2 Algorithm for Powering Computation

In this section, we present a new composite iterative algorithm for the computation of the powering function (X^Y), for a floating-point operand X and an integer b-bit exponent Y, describing the algorithm and its error analysis.

2.1 Overview

The algorithm is based on the well-known identity

$$X^Y = 2^{Y \log_2(X)}. \tag{1}$$

Considering a floating-point[3] input operand $X = M_x 2^{E_x}$, with M_x the n-bit significand and E_x the exponent, and an integer b-bit[4] input operand Y:

$$\begin{aligned} X^Y &= 2^{Y \log_2(M_x 2^{E_x})} = 2^{Y(\log_2(M_x)+\log_2(2^{E_x}))} \\ &= 2^{Y \log_2(M_x)} 2^{Y \log_2(2^{E_x})} = 2^{Y \log_2(M_x)} 2^{\log_2(2^{YE_x})} \\ &= 2^{Y \log_2(M_x)} 2^{YE_x}. \end{aligned} \tag{2}$$

According to (2), the powering function can be calculated by a sequence of operations consisting of the logarithm of the significand M_x, a multiplication by Y, and the exponential of the resulting product. The form of the result is a significand $2^{Y \log_2(M_x)}$ and an exponent YE_x, typical of a floating-point operand.[5]

For an efficient implementation of the powering function, the computation of the operations involved must be overlapped, which requires a left-to-right most-significant digit first (MSDF) mode of operation and the use of a redundant number system.

A problem of the proposed algorithm is the range of the digit-recurrence exponential, which is $(-\ln 2, \ln 2)$, while the argument of the exponential here is $Y \log_2(M_x)$, with Y an integer. To extend the range of convergence of this algorithm and, thus, guarantee the convergence of the overall proposed method, we extract the integer I and fractional part F of the product serially, which requires Y to be a b-bit (or $2b$-bit) integer.

The exponential becomes:

$$2^{Y \log_2(M_x)} = 2^I 2^F \tag{3}$$

and, therefore, according to (2),

$$X^Y = 2^F 2^{(I+YE_x)}, \tag{4}$$

with I and F the integer and fractional parts of $Y \log_2(M_x)$, resulting in a bounded argument for the exponential.

In summary, as illustrated in Fig. 1 for single-precision computations with $r = 128$, our algorithm for the computation of the powering function consists of three steps. The number of iterations to be performed in each stage will be explained in Section 2.2.

1. Computing the logarithm of the input significand, $\log_2(M_x)$, by employing a high-radix digit-recurrence algorithm [22].
2. Computing the product $Y \log_2(M_x)$ by using a serial *left-to-right carry-free* (LRCF) multiplication scheme [5]. The integer I and fractional F parts of $Y \log_2(M_x)$ are extracted serially.
3. Computing the exponential $M_z = 2^F$ by employing an online high-radix algorithm [24].

The exponent E_z can be computed in parallel by an integer multiply-add unit. The output of the algorithm is the floating-point normalized result:

2. The exponent Y can be $2b$-bit wide (instead of b-bit wide) when a maximum value of $2^b - 1$ does not meet the application requirements. We have chosen, for our study, duplicating the size of the input operand Y for radix values $r < 128$, taking into account the range of the integer exponent in the OpenGL specular lighting computations.

3. The algorithm can be adapted to a normalized fixed-point input operand $X = M_x$, $1 \leq M_x < 2$, resulting in $X^Y = 2^{Y \log_2(M_x)}$.

4. The size of the exponent Y can be set to $2b$-bits when the range must be extended, for instance, to perform lighting computations if the radix is $r < 128$.

5. For fixed-point computations, the value $2^{Y \log_2(M_x)}$ must be shifted YE_x positions.

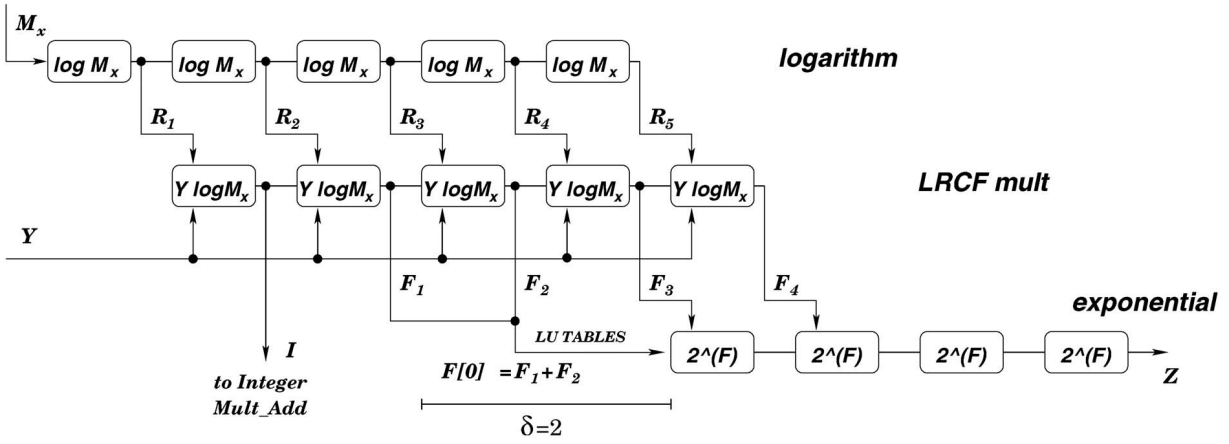


Fig. 1. Operation flow of the powering algorithm (single-precision computation, $r = 128$).

$$X^Y = M_z 2^{E_z} = 2^F 2^{(I+YE_x)}. \tag{5}$$

The use of redundancy results in $2^F \in (0.5, 2)$ and, therefore, a normalization of the final result may be necessary. However, the condition $F < 0$ can be determined in advance to the last iterations of the exponential and the final normalization can be performed with no extra delay.

The overall latency of the algorithm, as shown in Fig. 1, can be estimated as

$$latency = N_e + (\delta + 1) + 1 + \gamma, \tag{6}$$

with N_e the number of iterations of the online exponential, $(\delta + 1)$ cycles to accommodate the online delay, one cycle due to the LRCF multiplication, and γ extra cycles due to the computation of the integer part I of $Y \log_2(M_x)$. I can be obtained in a single cycle ($\gamma = 1$) when Y is a b-bit integer, while $\gamma = 2$ extra cycles are necessary when a range extension has been performed by allowing Y to be $2b$-bit wide.

A latency reduction by one cycle is achieved regarding the algorithm proposed in [19], [23] due to the elimination of the second LRCF multiplication, required in that case to guarantee the convergence of the algorithm, which was based on the identity $X^Y = e^{Y \ln X}$ instead of $X^Y = 2^{Y \log_2 X}$.

2.2 Error Analysis

The final error in the algorithm consists of the accumulation of the errors due to the cascaded implementation of a set of operations and must be bounded by 2^{-n-1} before the final rounding.

Let ϵ_l be the error in the computation of the logarithm of the input significand: $\ln(M_x) + \epsilon_l$. When the LRCF multiplication is performed, the associated error is:

$$\epsilon_{LRCF} = \epsilon_l Y + \epsilon_m, \tag{7}$$

with ϵ_m the error due to the LRCF multiplication scheme in the computation of the product.

The integer I and fractional F parts of the product $Y \log_2(M_x)$ are extracted serially, with no error affecting the integer part. The next operation to be performed is the exponential 2^F. The obtained result is:

$$2^F 2^{\epsilon_{LRCF}} + \epsilon_e, \tag{8}$$

with ϵ_e the error in the computation of the exponential.

The difference between the exact result and the obtained result must be bounded:

$$|2^F - 2^F 2^{\epsilon_{LRCF}} - \epsilon_e| < 2^{-n-1}. \tag{9}$$

Taking into account that $2^F \in (0.5, 2)$:

$$|1 - 2^{\epsilon_l Y + \epsilon_m} - \epsilon_e 2^{-1}| < 2^{-n-2}. \tag{10}$$

For a precision of n bits and a radix $r = 2^b$, a set of minimum values for ϵ_l, ϵ_m, and ϵ_e must be determined to guarantee a final result accurate to n bits. The values of the error parameters set the precision to be reached in each stage, n_l, n_e, and n_m. These parameters determine a minimum number of iterations of the logarithm (N_l) and the exponential (N_e) to be performed. As shown in Fig. 1, the number of iterations of the LRCF multiplication to be performed must be the same as those of the logarithm, N_l, because all the information must reach the exponential stage. The parameters N_l, N_e, n_l, n_e, and n_m set the size of the look-up tables, adders, and multipliers to be used.

Moreover, g_l, g_e, and g_m guard bits must be employed to guarantee that, in each stage of the powering algorithm, the iteration errors do not affect the achievement of the required precisions n_l, n_e, and n_m.

The critical parameter to be minimized first is ϵ_e since it is directly related to the required precision n_e to be reached in the exponential stage and, therefore, to N_e, the number of iterations of the online exponential algorithm to be performed.[6]

As an example, we show in Table 1 the set of minimum values for the considered parameters for single ($n = 24$) and double-precision ($n = 53$) computations, when the radix is $r = 128$. N_e and N_l are given in cycles, and n_e, n_l, and n_m are given in number of bits.

3 Implementation

In this section, a sequential architecture is proposed for the implementation of our high-radix powering algorithm. We outline here the main computations involved: 1) high-radix logarithm, 2) high-radix LRCF multiplication, and 3) online high-radix exponential, and then describe the main features

6. Note that N_e is the only parameter that affects the latency of the algorithm since $\delta = 2$ for any n and any $r \geq 8$, as shown in [24].

TABLE 1
Example of Parameters for Powering Computation ($r = 128$)

Precision	N_e (cycles)	N_l (cycles)	n_e (bits)	n_l (bits)	n_m (bits)
SP ($n = 24$)	4	6	26	36	27
DP ($n = 53$)	8	9	55	60	55

of the logic blocks employed. Fig. 2 shows the block diagram of the proposed architecture. Single thick lines denote long-word (around n bits) operands/variables in parallel form, single thin lines denote short-word (up to 11 bits) operands/variables in parallel form, and double lines denote single-digit (b bits) variables (R_j, I, and Fj).

3.1 High-Radix Logarithm

A high-radix digit-recurrence algorithm for the computation of $\ln(M_x)$ is described in detail in [19], [22]. Some modifications and optimizations have been made in the algorithm used here, according to the operation flow in the optimized algorithm for powering computation, and also a slightly different notation from that used in [22] is employed.

The algorithm for the computation of $\log_2(M_x)$ is based on the identity

$$\log_2(M_x) = \log_2\Big(M_x \prod f_j\Big) - \sum \log_2(f_j). \tag{11}$$

Provided that the following condition is satisfied:

$$M_x \prod f_j \to 1, \tag{12}$$

then

$$-\sum \log_2(f_j) \to \log_2(M_x). \tag{13}$$

The condition (12) can be achieved using a *multiplicative normalization*, which consists of determining a sequence f_j such that M_x is transformed to 1 by successive multiplications. To simplify the multiplications, it is convenient to define the constants as $f_j = 1 + l_j r^{-j}$, where $r = 2^b$ is the radix and l_j is a radix-r digit. This form of f_j allows the use of a *shift-and-add* implementation.

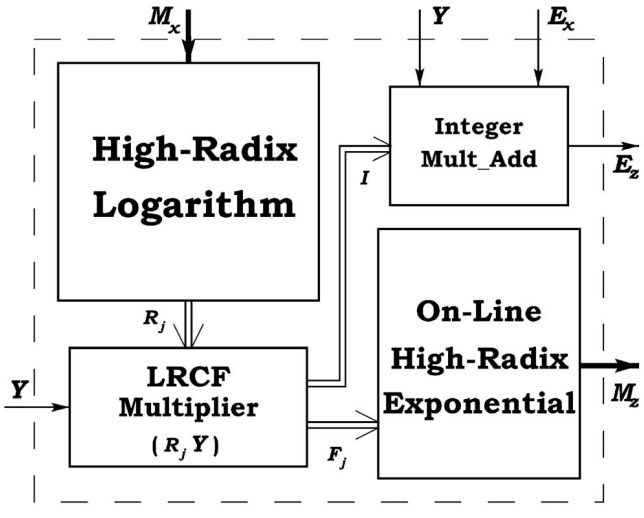


Fig. 2. Block diagram of the proposed architecture.

The recurrences for performing such multiplicative normalization and computing the logarithm digits are [22]:

$$\begin{aligned} W_l[j+1] &= r(W_l[j] + l_j + l_j W_l[j] r^{-j}) \\ R[j+1] &= rR[j] - r^{j-1}\log_2(1 + l_j r^{-j}) - rR_j, \end{aligned} \tag{14}$$

with $j \geq 1$, $W_l[1] = r(M_x - 1)$, $R[1] = 0$, and $R_1 = 0$. For a result precision of n_l bits, $N_l = \lceil n_l/b \rceil$ iterations are necessary. The scaled recurrence $R[j]$ has been defined as

$$R[j] = r^{j-2} L[j] \tag{15}$$

in order to extract a radix-r digit R_j per iteration from the same bit-positions in all iterations.

The block diagram of the high-radix logarithm stage is shown in Fig. 3, with double lines for SD operands, thin ones for single-digit operands, and thick lines for parallel operands. $TAB(rl_1)$ and $TAB(-\log_2 l_1)$ are the look-up tables storing rl_1 and $-\log_2(1 + l_1 r^{-1})$, respectively, addressed by the $b+1$ most significant bits of the input operand M_x.

The selection of the digits l_j in iterations $j \geq 2$ is done by rounding an estimate of the residual. Such an estimate is obtained by truncating the signed-digit representation of $W_l[j]$ to t fractional bits. The selection function is

$$l_j = -round(\hat{W}_l[j]). \tag{16}$$

The sign of the digit l_j is defined as opposite of the sign of $W_l[j]$ in order to satisfy a bound on the residual, thus assuring the convergence. The digit set for l_j is $\{-(r-1), \ldots, -1, 0, 1, \ldots, (r-1)\}$.

Iteration $j = 1$ does not converge with selection by rounding, so the selection of l_1 is performed by table look-up. This table is addressed by the $b+1$ most significant bits of the input operand M_x and the selection is done in such a way that the value of $|l_2|$ is bounded according to the convergence conditions [22]. However, this results in an overredundant digit l_1 ($b+1$ bits), increasing by one bit the size of the multiplier operand. The convergence conditions also determine a minimum value of $t = 2$ and the radix $r \geq 8$.

A multiply-add unit is used for the computation of the residual recurrence, unlike in the algorithm proposed in [22], where a separated SD multiplier and SDA4 were used. The digit l_1 is stored in the look-up table $TAB(rl_1)$ already in SD-4 recoded form to reduce the delay of the path containing the table and the multiply-add unit.

The logarithm constants are stored in a look-up table whose size grows exponentially with the radix. However, an approximation $-l_j r^{-1}/ln(2)$ can be used in iterations $j \geq \lceil N_l/2 \rceil + 1$ in order to reduce the overall hardware requirements of the algorithm, with a look-up table storing $-l_j/ln(2)$.

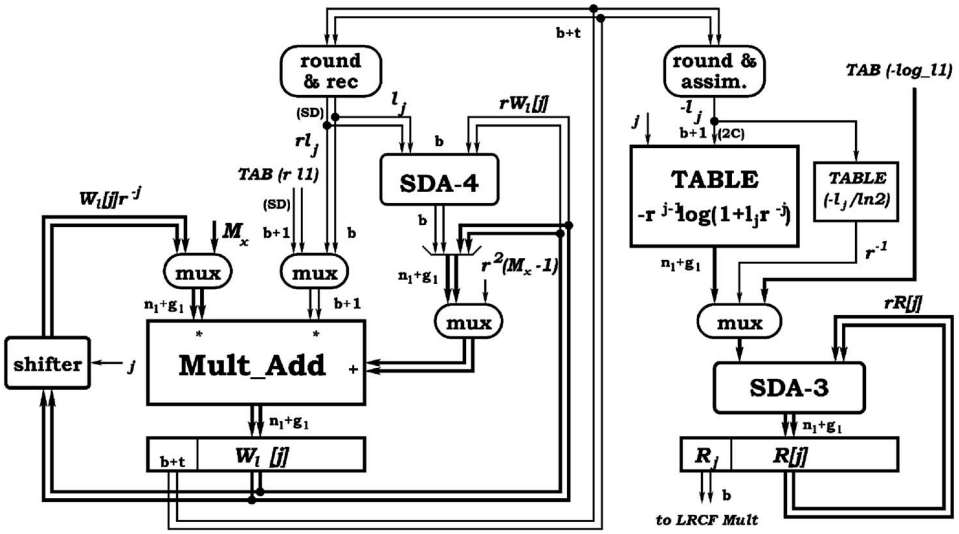


Fig. 3. Block diagram of the high-radix logarithm stage.

The use of redundant representation is mandatory in our algorithm for powering computation due to the left-to-right operation flow and results in faster execution times by making the additions independent of the precision.

3.2 LRCF Multiplication

The left-to-right carry-free (LRCF) multiplication, introduced in [5], produces the product digits from a redundant set in a most-significant-digit-first (MSDF) manner and performs the conversion on-the-fly of the product generated in a redundant form to the conventional form without using a carry-propagate adder and without additional delay. The resulting implementation is fast and regular and is very well-suited for VLSI implementations [14].

We adapt the LRCF multiplication to carry out the intermediate multiplication in the high-radix powering, utilizing its left-to-right operation flow, with redundant representation of the operands and recoding to a high-radix of the multiplier operand. However, since the high-radix exponential is of the online type, the digits produced by LRCF multiplier are used without conversion. The block diagram of the LRCF multiplication stage is shown in Fig. 4.

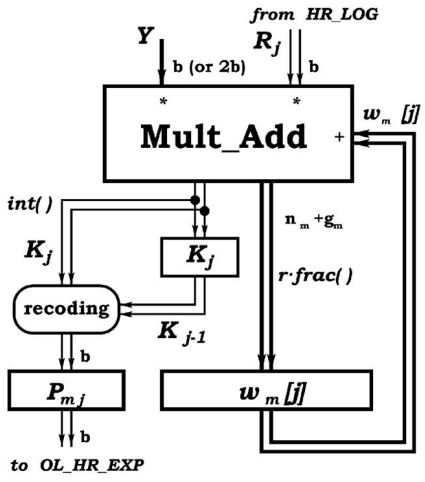


Fig. 4. Block diagram of the LRCF multiplication stage.

The adapted LRCF algorithm has two operands, A and D. A is the radix-2 representation of the operand a such that $a = \sum_{i=1}^{n_m} A_i 2^{-i}$, with $A_i \in \{0, 1\}$. D is the recoded radix-r representation of the multiplier d such that $d = \sum_{i=1}^{n_m/b} D_i r^{-i}$, with $D_i \in \{-r/2, \ldots, r/2\}$ and $r = 2^b$.

The recurrence produces a sequence of two accumulated products w and p as follows:

$$\begin{aligned} w[j+1] &= r(fraction(w[j] + aD_{j+1})) \\ K_{j+1} &= integer(w[j] + aD_{j+1}) \\ p[j+1] &= p[j] + K_{j+1} r^{-j-1}, \end{aligned} \tag{17}$$

with $j = 1, \ldots, n_m/b$ and initial values $w[0] = p[0] = 0$. The digits of the multiplier are used from most to least significant, unlike conventional multiplication schemes which use a right-to-left mode. After n_m/b steps, $p[n_m/b]$ is the most significant part of the product, while $w[n_m/b]$ is the least significant part.

A fast implementation of the LRCF multiplication scheme requires: 1) use of a redundant adder (either CS or SD) to compute the residual $w[j]$ and 2), since the maximum value of K_j in (17) is, in general, larger[7] than $(r-1)$, a recoding of K_j and K_{j+1} into P_j in the range $[-(r-1), (r-1)]$ is necessary. That is, the resulting digit will be $P_j = f(K_j, K_{j+1})$.

Regarding the implementation proposed in [5], the main modification in the LRCF multiplication stage in the architecture for powering computation is the use of a multiply-add unit for computing the recurrences, instead of separated redundant multiplier and adder.

3.3 Online High-Radix Exponential

An online high-radix algorithm for the computation of e^F is described in detail in [19], [24]. Some modifications and optimizations have been made in the algorithm used here, according to the operation flow in the optimized algorithm

7. The range of K_j is $|K_j| < 3r/2$.

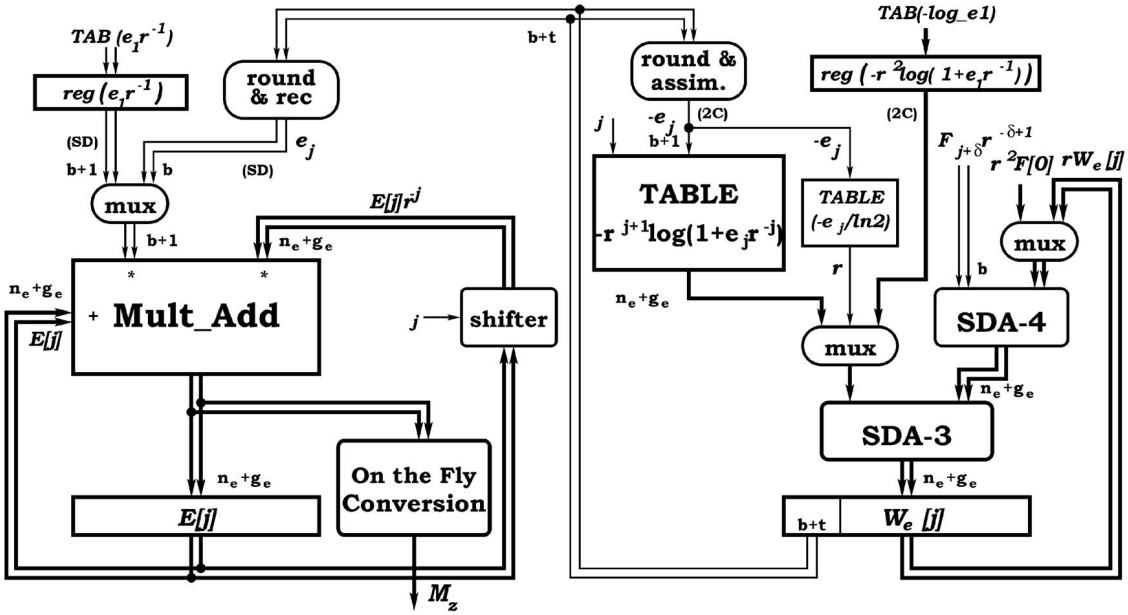


Fig. 5. Block diagram of the online high-radix exponential stage.

for powering computation, and also a slightly different notation is used.

The algorithm for the computation of 2^F is based on the identity

$$2^X = \left(\prod h_j\right) 2^{X-\sum \log_2(h_j)}, \tag{18}$$

with $h_j = 1 + e_j r^{-j}$, $r = 2^b$ being the radix and e_j a radix-r digit.

The exponential is computed on the basis of partial information:

$$F[0] = \sum_{j=1}^{\delta} F_j r^{-j}, \tag{19}$$

with δ the *online delay* and F_j the radix-r digits of the input operand F.

The recurrences for performing the *additive normalization* of F and computing the exponential are [24]:

$$\begin{aligned} W_e[j+1] &= r(W_e[j] - r^j \log_2(1 + e_j r^{-j}) + F_{j+\delta} r^{-\delta}) \\ E[j+1] &= E[j](1 + e_j r^{-j}), \end{aligned} \tag{20}$$

with $j \geq 1$, $W_e[1] = rF[0]$, and $E[1] = 1$. For a result precision of n_e bits, a total number of $N_e = \lceil n_e/b \rceil$ iterations are necessary. An approximation $-e_j r/\ln(2)$ to the logarithm constants can be used in iterations $j \geq \lceil N_e/2 \rceil + 1$, with a table storing $-e_j/\ln(2)$, in order to reduce the overall hardware requirements of the algorithm.

The block diagram of the online high-radix exponential stage is shown in Fig. 5, with double lines for SD operands, thin ones for single-digit operands, and thick lines for parallel operands. $TAB(e_1 r^{-1})$ and $TAB(-\log\_e_1)$ are the look-up tables storing $e_1 r^{-1}$ and $-r^2 \log_2(1 + e_1 r^{-1})$, respectively, addressed by the b most significant bits of the input operand F.

The selection of digits e_j is done in iterations $j \geq 2$ by rounding an estimate of the residual:

$$e_j = round(\hat{W}_e[j]), \tag{21}$$

with a digit set for e_j of $\{-(r-1), \ldots, -1, 0, 1, \ldots, (r-1)\}$. $\hat{W}_e[j]$ is obtained by truncating the signed-digit representation of $W_e[j]$ to t fractional bits.

The use of selection by table look-up is required in the first iteration. The table is addressed by the b most significant bits of the input operand F_1 and the selection is performed so that $-(r-3) \leq e_2 \leq (r-2)$, which results in an overredundant first digit e_1. The convergence conditions also determine a minimum value of $t = 2$, an online delay $\delta = 2$, and a bound on the radix $r \geq 8$, as shown in [24].

The look-up tables used in the first iteration are addressed one cycle in advance, since $F[0]$ is already known while $F_{\delta+1}$ is being computed, and the obtained values can be stored in the registers reg_e_1 and $reg\_\log e_1$.

The conversion from redundant to conventional representation of the final result is performed using an on-the-fly method [4], [6], avoiding the need for a carry-propagate addition and an increase in neither the cycle time nor the latency of the algorithm.

3.4 Implementation Details

The main features in the implementation of our architecture for powering computation are:

- All variables are in redundant representation (signed-digit) to allow faster execution of iterations, since the additions become independent of the precision. A similar approach could be proposed with the use of carry-save (CS) representation [15].
- All products of the type $Q[j]r^{-j}$ and $q_j r^{-j}$ (with $q_j = l_j$ or e_j) can be performed as shifts since $r = 2^b$.
- SDAα is a *signed-digit binary adder* with α input bit-vectors. The addition of two SD operands requires an SDA4 adder since the input operands are represented by two bit-vectors each. We use SDA3

TABLE 2
Execution Time and Total Area for Single-Precision Powering ($n = 24$)

Radix	latency	cycle time (τ)	exec. time (τ)	LOG area (fa)	EXP area (fa)	total area (fa)
8	9+6= 15	7.5	112.5	834	706	1620
16	7+6= 13	8.0	104.0	1067	878	2046
32	**6+6= 12**	**8.0**	**96.0**	**1408**	**948**	**2513**
64	5+6= 11	9.5	104.5	2201	1244	3621
128	**4+5= 9**	**10.0**	**90.0**	**2390**	**1412**	**3977**
256	4+5= 9	10.5	94.5	4075	2207	6471
512	3+5= 8	11.0	88.0	7408	3630	11278
1024	3+5= 8	11.5	92.0	11087	6567	17909

TABLE 3
Execution Time and Total Area for Double-Precision Powering ($n = 53$)

Radix	latency	cycle time (τ)	exec. time (τ)	LOG area (fa)	EXP area (fa)	total area (fa)
8	19+6= 25	8.5	212.5	2001	2107	4219
16	14+6= 20	8.5	170.0	1946	1979	4060
32	**11+6= 17**	**9.0**	**153.0**	**2790**	**2630**	**5622**
64	10+6= 16	9.5	152.0	4615	3033	7869
128	**8+5= 13**	**10.0**	**130.0**	**4921**	**4415**	**9563**
256	7+5= 12	10.5	126.0	8675	7514	16428
512	7+5= 12	11.0	132.0	14716	12166	27179
1024	6+5= 11	11.5	126.5	28484	15763	44559

adders for accumulating a SD operand and an operand in two's complement ($2C$) representation.

- An internal recoding from SD radix-2 to SD radix-4 representation of the multiplier operand is performed in the multiply-add units to reduce by half the number of partial products to be accumulated. However, when the multiplicand is represented in SD, the size of the unit is roughly double that of a regular multiplier with nonredundant multiplicand and one extra level in the accumulation tree is required.
- The *round&rec* units take the $(b + t)$ most significant bits of the residuals $W_l[j]$ and $W_e[j]$ ($t = 2$ in both cases) and produce an SD-4 representation of the digits l_j and e_j, to be used as multipliers in the multiply-add units.
- The *round&assim* units take the same $(b + t)$ input words, but produce a two's complement representation of the coefficient with opposite sign ($-l_j$ or $-e_j$) by rounding the input word and performing its assimilation into nonredundant representation. This $2C$ representation is used for addressing the look-up tables storing the logarithm constants and as inputs for the tables storing $-l_j/\ln(2)$ or $-e_j/\ln(2)$.

4 Evaluation and Comparison

In this section, we present estimates of the execution time and the area costs of the proposed architecture, for single and double-precision floating-point powering computations ($n = 24$ and $n = 53$ bits) with radix values $r = 2^b$ in the range from $r = 8$ to $r = 1,024$. These estimates are based on an approximate model for the cost and delay of the main logic blocks used.

The actual delays and area costs depend on the technology used and on the actual implementation. However, this model provides a good first-order approximation to the actual execution time and area values and has been widely used in technology-independent comparisons [8], [19], [22]. The units employed are the delay τ of a complex gate, a full-adder, and the area of a 1-bit full-adder (fa). A detailed explanation of the model can be found in [19], [22].

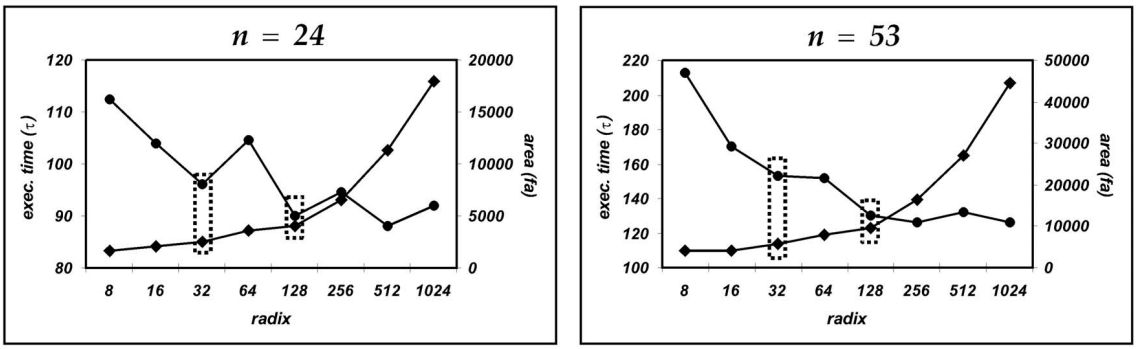
Tables 2 and 3 show estimates of the latency, cycle time, execution time, area of the logarithm stage, area of the exponential stage, and total area of the proposed architecture, for single and double-precision computations, respectively, when implemented with radix values going from $r = 8$ to $r = 1,024$. The latency is given in cycles, the cycle time and execution time are expressed in terms of τ, and the unit for the area estimates is fa. The dependence of both the execution times and the total area with the value of the radix r is graphically represented in Fig. 6.

The latency, as explained in (6), is $N_e + \gamma + 4$ cycles, with $\gamma = 1$ (with $r \geq 128$ and $\gamma = 2$ cycles for lower radices) since the online delay of the exponential stage is $\delta = 2$ in this case. The cycle time corresponds in most cases to the high-radix logarithm stage, where the critical path can be either the one composed of the initial table look-up for selecting the digit l_1, a multiplexer, the multiply-add unit and the register $W_l[j]$, or the one consisting of the *round&assim* unit, the look-up table storing the elementary logarithms, a 3:1 multiplexer, an SDA3 adder, and the register $R[j]$.

The total area of our architecture consists of the hardware requirements of the individual stages (high-radix logarithm, LRCF multiplication, and online high-radix exponential, shown in Figs. 3, 4, and 5), plus the integer $(b \times n_{exp})$-bit multiply-add unit to compute the exponent E_z, the control logic, and a b-bit CPA adder to assimilate F_1 into conventional representation before addressing the initial tables in the online high-radix exponential stage.

The analysis of the obtained results can be summarized in the following points:

- For radix values $r \geq 64$, the cycle time only depends on the radix, and not on the target precision, due to the use of redundant arithmetic.



(a) (a)

Fig. 6. Execution time and area cost estimates for the proposed architecture. (a) Single-precision powering. (b) Double-precision powering.

Delay estimates ($r = 512$)

path_A: 3τ (*round&assim*) + 4.5τ (TabLog_e_i) + 1τ (5:1 mux) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_c) = 11τ

path_B: 0.5τ (shifter) + 0.5τ (mux) + 5τ (Mult/Add_d) + 0.5τ (mux) + 1τ (reg_d) = 7.5τ

path_C: 3τ (*round&assim*) + 0.5τ (module) + 4.5τ (Tab_z) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_z) = 10.5τ

path_D: 1.5τ (*round&rec*) + 0.5τ (mux) + 0.5τ (inv) + 5τ (Mult/Add_w) + 0.5τ (mux) + 1τ (reg_w) = 9τ

path_E: 3τ (*round&assim*) + 0.5τ (module) + 4.5τ (TabLog_T_j) + 1τ (5:1 mux) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_c) = 11.5τ **(critical path)**

execution time: latency × cycle time = $10\ (4 + 3 + 3) \times 11.5\tau = \mathbf{115}\tau$

Delay estimates ($r = 128$)

path_A: 2τ (*round&assim*) + 4.5τ (TabLog_e_i) + 1τ (5:1 mux) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_c) = 10τ **(critical path)**

path_B: 1.5τ (shifter) + 0.5τ (mux) + 5τ (Mult/Add_d) + 0.5τ (mux) + 1τ (reg_d) = 8.5τ

path_C: 2τ (*round&assim*) + 0.5τ (module) + 4τ (Tab_z) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_z) = 9τ

path_D: 1.5τ (*round&rec*) + 0.5τ (mux) + 0.5τ (inv) + 5τ (Mult/Add_w) + 0.5τ (mux) + 1τ (reg_w) = 9τ

path_E: 2τ (*round&assim*) + 0.5τ (module) + 4τ (TabLog_T_j) + 1τ (5:1 mux) + 1τ (3:2 CSA) + 0.5τ (mux) + 1τ (reg_c) = 10τ **(critical path)**

execution time: latency × cycle time = $12\ (5 + 3 + 4) \times 10\tau = \mathbf{120}\tau$

Fig. 7. Delay and area estimates for a high-radix CORDIC computation ($n = 32$-bits).

- The main contribution to the total area for the proposed architecture comes from the logarithm stage due to the extra iteration (or two extra iterations for lower radices) necessary to obtain the final n-bit precision.[8] This extra iteration(s) result in bigger shifters and look-up tables for the logarithm stage.
- The main contribution to the total area for the proposed architecture comes from the combinational logic for small radices, but the area of the look-up tables becomes the primary factor for $r = 128$ in single-precision computations and for $r = 32$ in double-precision computations due to the exponential growth in the table size with the radix.

8. A precision of around $n_l = n + b$ or $n_l = n + 2b$ bits is typically necessary, as shown in Section 2.2.

- Using the identity $X^Y = 2^{Y \log_2(X)}$ instead of $X^Y = e^{Y \ln(X)}$ allows reducing the overall latency by one cycle regarding the algorithm proposed in [19], [23] due to the elimination of the second LRCF multiplication. Moreover, an extra multiplication $Y_L = Y \log_2(e)$ is also eliminated at the expense of requiring two extra look-up tables for storing $-l_j/\ln 2$ and $-e_j/\ln 2$. Since the size of these extra tables grows exponentially with the radix r, the impact in the total area is bigger for the higher radix values. However, the overall area requirements of our implementation are lower than or similar to those of the algorithm proposed in [19], [23] for any radix value $r \leq 128$.
- The cycle time increases with the radix, but the latency decreases. A good trade off between latency

TABLE 4
Area Estimates for the Tables Used in the HR Vectoring CORDIC Implementation [1]

Table	$r = 512$		$r = 128$	
	size ($Kbits$)	**area** (fa)	**size** ($Kbits$)	**area** (fa)
Tab_M_2	$2^{12} \times 10 = 40Kb$	1200	$2^{10} \times 10 = 10Kb$	350
TabLog_M_2	$2^{12} \times 36 = 144Kb$	4320	$2^{10} \times 36 = 36Kb$	1260
Tab_z	$2^{11} \times 36 = 72Kb$	2520	$2^{9} \times 36 = 18Kb$	630
TabLog_T_j	$2^{11} \times 36 = 72Kb$	2520	$2^{9} \times 36 = 18Kb$	630
Tab_e_1	$2^{11} \times 12 = 24Kb$	840	$2^{9} \times 12 = 6Kb$	210
TabLog_e_1	$2^{11} \times 36 = 72Kb$	2520	$2^{9} \times 36 = 18Kb$	630
TabLog_e_i	$2^{11} \times 36 = 72Kb$	2520	$2^{10} \times 36 = 36Kb$	1260
TOTAL	**496**Kb	**16440**	**142**Kb	**4970**

and cycle time can be achieved for specific values of r, leading to low execution times. The area requirements must also be taken into account when trying to determine the most efficient implementations. According to Tables 2 and 3 and Fig. 6, the best trade offs correspond to the radix values highlighted in the tables and the charts. Thus, in applications demanding high-speed processing, the most efficient implementations correspond to radix $r = 128$, while, for all other applications, implementations with radix $r = 32$ may be preferable due to their lower area requirements.

The main conclusion that can be drawn from this analysis is that little advantage, or no advantage at all, is obtained from using very-high radix values such as $r = 256$, $r = 512$, or $r = 1,024$ because the execution times are similar to those achieved with $r = 128$, but the area requirements are much higher.

4.1 Comparison with High-Radix CORDIC

For the sake of reference, we now give execution time and hardware requirement estimates for a high-radix vectoring CORDIC implementation [1]. This algorithm has been chosen as a reference because it allows the computation of two-variable functions (for instance, the modulus of a 2D vector) and high-radix algorithms have been proposed for its acceleration. To allow for a fair comparison, the same assumptions are made for both algorithms, redundant representation is used for the variables, and we consider a fixed-point implementation of our powering algorithm for $n = 32$ (details in [19]) since the algorithm proposed in [1] performs $n = 32$-bit computations. The main features of such an algorithm are the use of radix $r = 512$, with a small radix $R = 32$ for the first CORDIC microrotation, and two scaling operations to guarantee the convergence of the algorithm.

When computing the modulus, the $r = 512$ CORDIC algorithm requires four microrotations, two scaling operations (taking three cycles due to the complexity of the second scaling), and four extra iterations for compensating the scaling factor. This results in a latency of 10 cycles ($4 + 3 + 3$ since the first extra iteration can be overlapped with the last CORDIC microrotation). According to the approximate model used, the critical path has a delay of 11.5τ, as shown in Fig. 7. The critical path in this case consists of the *round&assim* unit, module computation, table storing the logarithms of the elementary angles (required for scale factor compensation), 5:1 multiplexer, 3:2 CSA adder, 2:1 multiplexer, and register.[9] The execution time can be therefore estimated as 115τ, similar to the execution time of the powering algorithm (about 100τ). However, the total area of the implementation proposed in [1] is about $18,000fa$, more than 3 times the area estimate of our powering algorithm. This is due to the huge size of the tables employed (around $500Kbits$), as shown in Table 4.

The conclusions drawn from the analysis of the trade offs between execution time and area costs of our architecture for powering computation can be applied to high-radix CORDIC as well and, therefore, we have included, in Fig. 7, the delay estimates for an implementation with radix $r = 128$. The main paths in this implementation are more compensated and the cycle time is significantly reduced from 11.5τ to 10τ. However, the latency in this case is higher since 12 cycles are necessary for computing the modulus (five cycles for microrotations, three cycles for the two prescaling operations, and five cycles for scaling factor compensation, the first one overlapped with the last microrotation). The execution time can therefore be estimated as 120τ, only 5τ slower than the $r = 512$ implementation, but the table size is now $142Kb$, leading to a total area about $6,000fa$, similar to the area estimate of our algorithm ($5,964fa$), and three times smaller than that of the implementation with radix $r = 512$.

Table 5 summarizes the comparison between the proposed architecture for powering computation and the CORDIC algorithm proposed in [1], for fixed-point $n = 32$-bit computations, showing the latency, cycle time, execution time, and total area estimates in both cases. We have included the implementations of high-radix CORDIC with both radix $r = 128$ and $r = 512$ to mark the importance of the conclusions drawn from the analysis performed.

We can conclude that, with our algorithm, the powering function can be computed efficiently in dedicated hardware with area cost and performance comparable to those of the evaluation of other elementary functions.

5 Computation of Logarithm and Exponential

The logarithm and exponential operations are included in the operation flow of our algorithm for powering computation in such a way that it is not possible to directly obtain them as an output result. However, some modifications can be made to the proposed architecture to allow the

9. In the estimates presented in [1], the registers and the look-up tables are not included, but we have shown [19] that, in this type of architecture, such tables determine the execution time since they belong to the slowest paths in the architecture.

TABLE 5
Comparison with High-Radix CORDIC Vectoring Algorithm ($n = 32$ Bits)

Scheme	latency	cycle time (τ)	exec. time (τ)	area(fa)
powering ($r = 128$)	10	10.0	100	5964
CORDIC ($r = 128$)	12	10.0	120	6000
CORDIC ($r = 512$)	10	11.5	115	18000

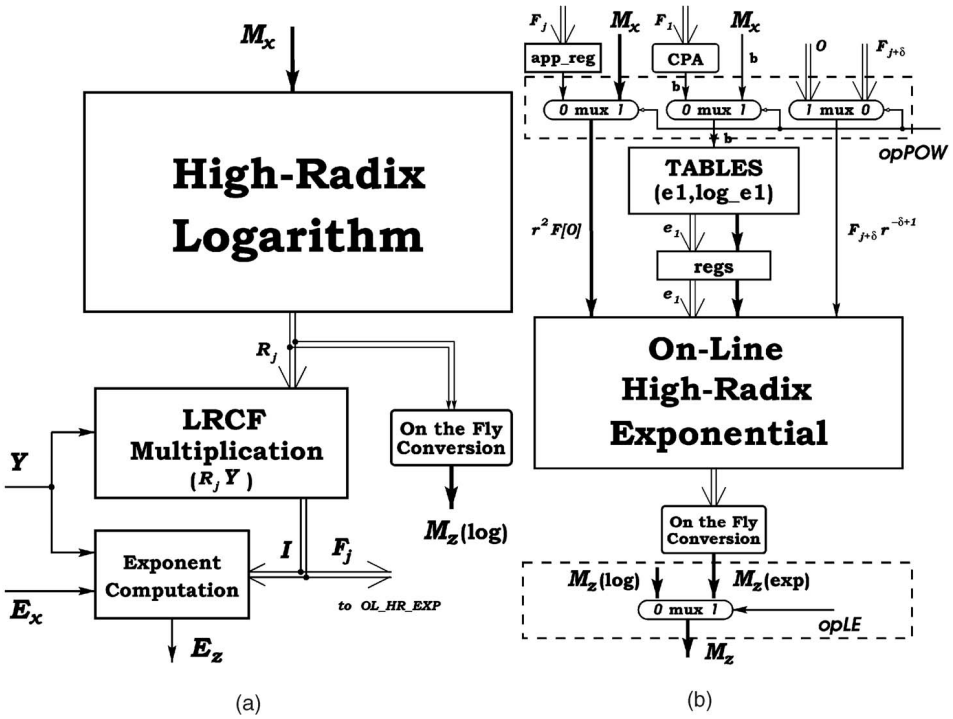


Fig. 8. Block diagrams of the modified architecture. (a) Logarithm and LRCF multiplication. (b) Online exponential.

independent computation of the three operations: logarithm, exponential, and powering. The resulting unit is more versatile than the original one and becomes an interesting alternative for elementary function computation in digital signal processing (DSP), computer graphics, and other applications.

Fig. 8 shows the main modifications to be performed to the previously proposed architecture to allow for this extra functionality. These modifications only affect the inputs and outputs of the main stages, whose internal composition remains unchanged. The extra multiplexers to be inserted are highlighted within a dashed box in Fig. 8b. Also, in this figure, *app_reg* stands for an append register, which is necessary in the computation of the powering since $r^2F[0] = r^2(F_1 + F_2)$.

A summary of the control decisions to be taken depending on the operations to be computed is shown in Table 6. Two control signals suffice for selecting the correct behavior of the architecture: $opPOW$ and $opLE$. The signal $opPOW$ is responsible for selecting between including the exponential stage in the powering operation flow or using it as an independent unit with parallel inputs and output. In order to do this, $opPOW$ must properly set the inputs of the look-up tables $TAB(e_1 r^{-1})$, $TAB(-\log\_e_1)$, the input parameter $r^2F[0]$, and the digits $F_{j+\delta}r^{-\delta+1}$. For logarithm computation, the values of these signals are not relevant. The second control signal, $opLE$, must distinguish between the computation of logarithm (taking the output operand from the *on-the-fly* conversion unit at the output of the logarithm stage) and the computation of the exponential and powering (in both cases, the output operand is the original *on-the-fly* conversion of the output of the exponential stage). According to Fig. 8, the computation of the logarithm admits two combinations of the control signals $opPOW$ and $opLE$: "00" and "10," while, for the computation of the exponential, they must take the value "11" and, for powering computation, a control signal "01" is necessary.

The computation of logarithm and exponential has a lower latency than the computation of powering and, therefore, the control unit must also allow the completion of the operation in fewer cycles than the original case. Table 7 shows the latency values for the computation of the three operations when performing single and double-precision computations. Only implementations with radix $r = 32$ and $r = 128$ for single-precision and $r = 32$ and 128 for double-precision, are considered since they have been shown in Section 4 to be the most efficient ones for powering computation.

The latency of the exponential computation is one cycle higher than that of the logarithm since, in this case, the first cycle is employed in addressing the initial look-up tables $TAB(e_1 r^{-1})$, $TAB(-\log\_e_1)$, and the first iteration of the algorithm is completed in the second cycle.

TABLE 6
Computations Performed in the Complete Unit Depending on the Operation

Operation	logarithm	exponential	powering
Input $TABs(e_1 r^{-1}), (-\log\_e_1)$	don't care	$\lfloor M_x \rfloor_b$	F_1
$r^2 F[0]$	don't care	$r^2 M_x$	$r^2(F_1 + F_2)$
$F_{j+\delta} r^{-\delta+1}$	don't care	0	$F_{j+\delta} r^{-\delta+1}$
Output M_z	$ofc\,[\log_2(M_x)]$	$ofc\,[2^{M_x}]$	$ofc\,[2^{Y\log_2(M_x)}]$
Control signals ($opPOW, opLE$)	00, 10	11	01

6 Powering Computation with Exponents $Y = 1/q$

Our algorithm for powering computation can be extended for the computation of exponents of the type $Y = 1/q$, with q integer (the qth root extraction), as shown next:

$$\begin{aligned} X^Y &= 2^{1/q\log_2(M_x 2^{E_x})} = 2^{1/q(\log_2(M_x)+\log_2(2^{E_x}))} \\ &= 2^{1/q[\log_2(M_x)+E_x]} = 2^{1/q\log_2(M_x)} 2^{1/qE_x}. \end{aligned} \tag{22}$$

The first part of the equation can be handled in the same way as in the original powering algorithm, taking the integer and fractional parts I' and F' of the product $1/q\log_2(M_x)$. In this case, the integer part is always $I' = 0$ since $1/q < 1$ and $\log_2(M_x) < 1$:

$$2^{1/q\log_2(M_x)} = 2^{F'}, \tag{23}$$

which allows reducing the overall latency of the algorithm by one cycle.

However, the term $2^{1/qE_x}$ must be handled differently since the exponent is not an integer. In this case, an integer division and modulus operation can be performed in order to again extract the integer part of E_x/q:

$$2^{1/qE_x} = 2^{\lfloor E_x/q \rfloor} 2^{\psi}, \tag{24}$$

with $\psi = E_x \% q$. The output of the algorithm is therefore, according to (22) and (24), the floating-point result:

$$X^Y = M_z 2^{E_z} = 2^{\psi} 2^{F'} 2^{\lfloor E_x/q \rfloor}. \tag{25}$$

The term $M_z = 2^{\psi} 2^{F'}$ can be obtained as the output of the exponential stage since the exponential algorithm computes αe^{β} when the input argument is β and $E[1] = \alpha$ is taken as the initial value for the $E[j]$ recurrence [19], [24] instead of $E[1] = 1$. For any small q, a table can be employed to store the q values 2^{ψ}, which go from $2^0, 2^{1/q}, \ldots$ to $2^{q-1/q}$ and then the appropriate value can be used to load and initialize the register $E[j]$ as 2^{ψ}, depending on the result of the modulus operation ψ. The exponent term E_z is $2^{\lfloor E_x/q \rfloor}$, with an integer argument in the exponential.

TABLE 7
Latency for the Computation of the Three Operations

Operation	Single-Precision		Double-Precision	
	$r = 32$	$r = 128$	$r = 32$	$r = 128$
logarithm	5	4	11	8
exponential	6	5	12	9
powering	12	9	17	13

The new hardware requirements in the architecture to allow for the computation of the powering function with exponents of the type $Y = 1/q$ are a look-up table where the values $1/q$ are stored with high-accuracy (if only a subset of q values is interesting for the application, a significant area reduction can be achieved since a small table can be used), the table to store the values 2^{ψ}, and the unit to perform the modulus operation. The term $\lfloor E_x/q \rfloor$ can be computed as $E_x \times 1/q$, taking the integer part of the result, once the table look-up of $1/q$ has been performed.

7 Conclusion

An architecture for the computation of the logarithm, exponential, and powering functions has been presented in this paper, based on a high-radix composite algorithm for computing the powering X^Y with integer exponent Y. The algorithm consists of computing X^Y as $2^{Y\log_2(M_x)} 2^{YE_x}$, through a sequence of overlapped operations: logarithm, multiplication, and exponential.

The computation of the logarithm is carried out by a high-radix digit-recurrence unit, with selection by rounding. The intermediate multiplication is computed by high-radix left-to-right carry-free (LRCF) multiplier. Finally, the exponential is computed by an online high-radix unit with selection by rounding. The computation of the exponent of the result is carried out in parallel by an integer multiply-add unit.

A sequential architecture implementing our algorithm has been proposed and estimates of the execution times and hardware requirements have been obtained, based on an approximate model for the delay and the area of the main components, for single and double-precision floating-point computations with radix values going from $r = 8$ to $r = 1,024$. The analysis of the trade offs between area and speed in the implementation of this architecture shows that the most efficient implementation corresponds to radix $r = 32$, although an implementation with radix $r = 128$ may be suitable when very high-speed processing is necessary, at the expense of greater hardware requirements. There is no advantage in the use of a very-high radix value such as $r = 256$, $r = 512$, or $r = 1,024$ since the execution times are similar to those achieved with $r = 128$, but the area requirements are much higher.

A comparison with a high-radix CORDIC architecture shows that the execution times and hardware requirements of the unit computing the powering are similar to those of other iterative algorithms for the computation of elementary functions.

Finally, the proposed architecture has been modified to allow the independent computation of logarithm and exponential, with lower latencies than those of powering and our algorithm has been extended to powering

computation with exponents of the type $Y = 1/q$ (i.e., the qth root extraction).

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their suggestions and contributions, which have led to some optimizations and additional results. This work was developed while J.-A. Piñeiro was with the University of California, Los Angeles (UCLA). J.-A. Piñeiro and J.D. Bruguera were partially supported by the Spanish Ministry of Science and Technology (MCyT-FEDER) under contract TIC2001-3694-C02.

REFERENCES

[1] E. Antelo, T. Lang, and J.D. Bruguera, "Very-High Radix CORDIC Vectoring with Scalings andSelection by Rounding," *Proc. 14th Int'l Symp. Computer Arithmetic (ARITH14),* pp. 204-213, Apr. 1999.
[2] J.-C. Bajard, S. Kla, and J.-M. Muller, "BKM: A New Hardware Algorithm for Complex Elementary Functions," *IEEE Trans. Computers,* vol. 43, no. 8, pp. 955-964, Aug. 1994.
[3] W. Cody and W. Waite, *Software Manual for the Elementary Functions.* Prentice Hall, 1980.
[4] M.D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventiona lRepresentations," *IEEE Trans. Computers,* vol. 36, no. 7, pp. 895-897, July 1987.
[5] M.D. Ercegovac and T. Lang, "Fast Multiplication without Carry Propagate Addition," *IEEE Trans. Computers,* vol. 39, no. 11, pp. 1385-1390, Nov. 1990.
[6] M.D. Ercegovac and T. Lang, "On-the-Fly Rounding," *IEEE Trans. Computers,* vol. 41, no. 12, pp. 1497-1503, Dec. 1992.
[7] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations.* Kluwer Academic, 1994.
[8] M.D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 628-637, July 2000.
[9] C.T. Fike, *Computer Evaluation of Mathematical Functions.* Prentice Hall, 1968.
[10] S. Gal and B. Bachelis, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *ACM Trans. Math. Software,* vol. 17, no. 1, pp. 26-45, Mar. 1991.
[11] R.E. Goldschmidt, "Applications of Division by Convergence," MS thesis, Electrical Eng. Dept., Massachussets Inst. of Technology (MIT), June 1964.
[12] D. Harris, "A Powering Unit for an OpenGL Lighting Engine," *Proc. 35th Asilomar Conf. Signals, Systems, and Computers,* pp. 1641-1645, 2001.
[13] J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.G. Thacher, and C. Witzgall, *Computer Approximations.* New York: Wiley, 1968.
[14] R.K. Kolagotla, H.R. Srinivas, and G.F. Burns, "VLSI Implementation of a 200-MHz 16*16 Left-to-Right Carry-Free Multiplier in 0.35 μm CMOS Technology for Next-Generation DSPs," *Proc. IEEE 1997 Custom Integrated Circuits Conf.,* pp. 469-472, 1997.
[15] P. Kornerup, "Reviewing 4-to-2 Adders for Multi-Operand Addition," *Proc. IEEE 13th Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP '02),* pp. 218-229, 2002.
[16] D.M. Lewis, "114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications," *IEEE J. Solid-State Circuits,* vol. 30, no. 12, pp. 1547-1553, 1995,
[17] J.-M. Muller, *Elementary Functions. Algorithms and Implementation.* Birkhauser, 1997.
[18] S.F. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor," *Proc. 14th Symp. Computer Arithmetic (ARITH14),* pp. 106-115, Apr. 1999.
[19] J.-A. Piñeiro, "Algorithms and Architectures for Elementary Function Computation," PhD dissertation, Dept. of Electronics and Computer Eng., Univ. Santiago de Compostela, June 2003, http://www.ac.usc.es/~alex.
[20] J.-A. Piñeiro and J.D. Bruguera, "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root," *IEEE Trans. Computers,* vol. 51, no. 12, pp. 1377-1388, Dec. 2002.
[21] J.-A. Piñeiro, J.D. Bruguera, and J.-M. Muller, "Faithful Powering Computation Using Table Look-Up and Fused Accumulation Tree," *Proc. IEEE 15th Int'l Symp. Computer Arithmetic,* pp. 40-47, 2001.
[22] J.-A. Piñeiro, M.D. Ercegovac, and J.D. Bruguera, "High-Radix Logarithm with Selection by Rounding," *Proc. IEEE 13th Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP '02),* pp. 101-110, July 2002.
[23] J.-A. Piñeiro, M.D. Ercegovac, and J.D. Bruguera, "High-Radix Iterative Algorithm for Powering Computation," *Proc. IEEE 16th Int'l Symp. Computer Arithmetic (ARITH16),* pp. 204-211, June 2003.
[24] J.-A. Piñeiro, M.D. Ercegovac, and J.D. Bruguera, "On-Line High-Radix Exponential with Selection by Rounding," *Proc. IEEE Int'l Symp. Circuits And Systems (ISCAS 2003),* May 2003.
[25] 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.
[26] H.C. Shin, J.A. Lee, and L.S. Kim, "A Minimized Hardware Architecture of Fast Phong Shader Using Taylor Series Approximation in 3D Graphics," *Proc. Int'l Conf. Computer Design, VLSI in Computers and Processors,* pp. 286-291, 1998.
[27] N. Takagi, "Powering by a Table Look-Up and a Multiplication with Operand Modification," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1216-1222, Nov. 1998.
[28] P.T.P. Tang, "Table Look-Up Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic,* pp. 232-236, June 1991.
[29] M. Zhang, S. Vassiliadis, and J.G. Delgado-Frias, "Sigmoid Generators for Neural Computing Using Piecewise Approximations," *IEEE Trans. Computers,* vol. 45, no. 9, pp. 1045-1050, Sept. 1996.

Jose-Alejandro Piñeiro received the PhD degree in computer engineering in 2003 and the MSc degree (1999) and BSc degree (1998) in physics (electronics), from the University of Santiago de Compostela, Spain. In February 2004, he joined Intel Barcelona Research Center, Intel Labs-UPC, whose research focuses on new microarchitectural paradigms and code generation techniques for both IA-32 and IPF families. His research interests are also in the area of computer arithmetic, VLSI design, and numerical processors.

Milos D. Ercegovac received the MS (1972) and PhD (1975) degrees in computer science from the University of Illinois, Urbana-Champaign, and the BS degree in electrical engineering (1965) from the University of Belgrade, Yugoslavia. He is a professor and chair of the University of California at Los Angeles Computer Science Department. He specializes in research and teaching in digital arithmetic, digital design, and computer system architecture. His research contributions have been extensively published in journals and conference proceedings. He is a coauthor of two textbooks on digital design and of a monograph in the area of digital arithmetic. Dr. Ercegovac has been involved in organizing the IEEE Symposia on Computer Arithmetic since 1978. He served as an editor of the *IEEE Transactions on Computers* and as a subject area editor for the *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE and a member of the ACM.

Javier D. Bruguera received the BSc degree in physics and the PhD degree from the University of Santiago de Compostela, Spain, in 1984 and 1989, respectively. Currently, he is a professor with the Department of Electronic and Computer Engineering at the University of Santiago de Compostela. His research interests are in the area of computer arithmetic, VLSI design for signal and image processing, and parallel architectures. He is a member of the IEEE.