Iterative Algorithm and Architecture for Exponential, Logarithm, Powering and Root Extraction

Álvaro Vázquez, and Javier D. Bruguera

Iterative Algorithm and Architecture for Exponential, Logarithm, Powering and Root Extraction

Álvaro Vázquez, *Member, IEEE* and Javier D. Bruguera, *Member, IEEE*

Abstract—An algorithm and architecture for powering computation and root extraction, with fixed–point and floating–point exponents, is presented in this paper. The algorithm is based on an optimized iterative sequence of parallel and/or overlapped operations: (1) reciprocal, (2) high–radix digit–recurrence logarithm, (3) left–to–right carry–free multiplication and (4) high–radix on–line exponential. A redundant number system is used to allow for the overlapping of the different operations of the algorithm. As the logarithm and exponential are part of the sequence of operations, some minor changes are made to allow for the independent computation of the logarithm and exponential functions. A sequential implementation of the algorithm is proposed and the execution times and hardware requirements are estimated for single and double-precision floating-point computations. These estimates are obtained for several radices, according to an approximate model for the delay and area of the main logic blocks, and help to determine the radix values which lead to the most efficient implementations.

Index Terms—Elementary functions computation, digit–recurrence algorithms, high–radix algorithms, floating–point representation

1 Introduction

Microprocessor performance has grown rapidly during the last two decades, enabled by transistor speed scaling and microarchitecture technology, in such a way that current microprocessors have adopted a multicore architecture. However, if chip architects try to improve performance simply by adding more cores, the power consumption of chips would be prohibitive [2]. Therefore, multicore processors with uniform instruction set but heterogeneous implementations, including fixed-function and multi–function accelerators, seem to be the most attractive approach [2], [10]. Then, the design of new algorithms and architectures for elementary functions of interest in some areas, such as computer graphics, digit signal processing, communication, scientific computation and simulation, seem a promising approach for future microprocessors.

Powering (X^Y), logarithms ($\log_2 X$, $\ln X$, $\log_{10} X$, ...), exponentials (2^X, e^X, 10^X, ...) and root extraction ($X^{1/Y}$) are frequent operations in computer graphics, digital signal processing (DSP), communications, physical simulation, and scientific computing, among others [1], [8], [14], [23]. Note that the powering and the root extraction include a number of very frequent operations in those areas, such as square root ($X^{1/2}$), inverse square root ($X^{-1/2}$), cubic root ($X^{1/3}$), inverse cubic root ($X^{-1/3}$), squaring (X^2), inverse squaring (X^{-2}), reciprocal (X^{-1}), and some other less frequent but also important operations.

These functions, together with some other elementary functions, have been traditionally computed by software routines, which provide very accurate results but are often too slow for numerically intensive or real–time applications. Therefore, the development of dedicated hardware for the computation of those elementary functions has been a challenging task for years.

• *The authors are with the "Centro de Investigación en Tecnoloxías da Información (CITIUS)", University of Santiago de Compostela, Spain; both authors are members of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).*
E-mail: alvaro.vazquez@usc.es; jd.bruguera@usc.es

The two main approaches which have been used for the computation of elementary functions are table–driven and digit–recurrence algorithms [7], [12]. There are a number of architectures for the computation of the exponential and logarithm; however, accurately computing the floating–point powering function and the root extraction is difficult. The prohibitive hardware requirements of a table–driven implementation (note that both are 2–variable functions) and the high intrinsic complexity of a digit–recurrence based algorithm have lead only to partial solutions, such as powering or root extraction for a constant exponent or for very low precision [3], [16], [20], [11].

Of course, the table–driven polynomial approximations can be adapted to compute more than just one exponent or elementary function, but this needs the replication of the look–up tables [3], [16], [20]. On the other hand, a digit–recurrence methodology for the q–th root extraction, q being an integer, is presented in [11], but the architecture for the computation of large q–th roots seems difficult to implement because of its complexity depends on q.

In any case, the methods above cannot be considered as general methods for the calculation of any powering function or q–th root extraction. Moreover, all the previous implementations deal with fixed–point exponents.

In this paper we present a detailed description of two variations of an optimized iterative algorithm for powering calculation and root extraction. First, an algorithm for the calculation of X^y or $X^{1/y}$, for a floating–point operand X, y being an integer operand, with $|y| \geq 2$ for root extraction, is presented. After that, the algorithm is modified to allow the computation of X^Y and $X^{1/Y}$ for a floating–point exponent Y. In this case, due to implementation and representation issues, the dynamic range of the exponent is limited although the algorithm is valid for any exponent.

Powering and root extraction functions are computed through a sequence of overlapped operations: high–radix digit–recurrence logarithm, high–radix left–to–right carry–free multiplication and on–line high–radix exponential. As the computation of logarithm and exponential are part of the algorithm some minor changes, which basically consist of the introduction of additional input or output path, are made to allow for the independent computation of logarithm and exponential.

A preliminary version of our algorithm for the computation of $X^{1/q}$, q being an integer, has been presented in [21], which is based on a previous architecture for the computation of the powering function X^p with integer exponent p [15]. In this extended version, we propose an integrated dataflow for the root extraction and the powering function and the extension to powering and root extraction with a floating–point exponent.

2 Elementary Functions in the IEEE 754-2008 Standard

The revised version of the IEEE 754 standard for floating-point (IEEE 754-2008) [9] recommends the implementation of the correctly rounded powering function X^y with three variants:

- **pown(X, y)**, X^y with y integer and X a floating–point number.
- **powr(X, Y)**, X^Y with Y and $X \geq 0$ floating–point numbers.
- **pow(X, Y)**, X^Y with Y and X floating–point numbers.

The architectures proposed in this paper only accept arguments $X \geq 0$, so that negative arguments must get special treatment. For example, the function **pown(X, y)** is defined as

$$pown(X,y) = \begin{cases} X^y & \text{if } X \geq 0 \\ (-1)^y \times X^y & \text{otherwise} \end{cases}$$

In case of a floating–point exponent Y, function $pow(X,Y)$ must signal the invalid operation exception when $X < 0$ and Y not integer,

$$pow(X,Y) = \begin{cases} |X|^Y & \text{if } X \geq 0 \\ (-1)^Y \times |X|^Y & \text{if } X < 0 \ \& \\ & Y \text{ integer} \\ NaN & \text{otherwise} \end{cases}$$

Concerning the root computation, the standard only recommends the implementation of integral roots, **rootn(X, y)**. However, we extend

the algorithm and architecture for the powering computation to evaluate also function $|X|^{1/Y}$, where Y is a floating–point number. Then,

$$root(X,Y) = \begin{cases} |X|^{1/Y} & \text{If } X \geq 0 \\ (-1) \times |X|^{1/Y} & \text{If } X < 0 \ \& \\ & Y \text{ odd int.} \\ NaN & \text{otherwise} \end{cases}$$

Moreover, the architecture we propose also evaluates other floating-point functions recommended by the standard, such as logarithms and exponentials.

Correct rounding of elementary functions in hardware is too much expensive because an approximation of the exact result to a sufficient (large) precision is required; the problem of determining this precision is known as the Table Maker's Dilemma [13]. In this case, providing accurate results with less restrictive requirements would be a more feasible hardware solution.

Therefore, instead of correct rounding, faithful rounding is provided for the functions implemented in this paper. The computed result of a function is faithfully rounded if it is one of the two floating-point numbers that surround the exact result [17]. Note that if the exact result is a floating-point number, the system must return this one.

3 Algorithm for Powering and Root Extraction with a Fixed–Point Exponent

The function to be computed is X^Z, where X is a floating–point number, $X = (-1)^{s_x} \times M_x \times 2^{E_x}$, M_x being the n-bit significand[1] and E_x the n_{E_x}-bit signed exponent, and Z is a $n_z + 1$–bit fixed–point exponent of the form

$$Z = \begin{cases} y & \text{in powering computation} \\ 1/y & \text{in root extraction} \end{cases}$$

y being a signed integer operand of $n_y + 1$ bits, with $|y| \geq 2$ for root extraction. This algorithm provides an integrated formulation for the computation of the powering function and root extraction. It is an extension of a recent conference paper [21], where an algorithm for the q–th root extraction has been presented. The algorithm and implementation in [21] were based on the algorithm for powering computation presented in [15], although there are important differences between both algorithms.

Though we only consider here normal floating-point values X, denormals can be easily handled as done in [22] for logarithm computation. In this case, an extra iteration is required to normalize the significand.

3.1 Overview

The algorithm is based on the identity

$$X^Z = 2^{\log_2(X^Z)} = 2^{Z \log_2(X)}. \quad (1)$$

Considering that X is a floating–point operand,

$$X^Z = 2^{Z \log_2(M_x \times 2^{E_x})} = 2^{Z(\log_2 M_x + E_x)} \quad (2)$$

defining

$$S = \log_2 M_x + E_x \quad (3)$$

which is the concatenation of the digits of the integer value E_x and of $\log_2 M_x \in [0,1)$, we obtain

$$X^Z = 2^{Z \times S}. \quad (4)$$

According to Equations (3) and (4), X^Z can be calculated as a sequence of operations: (1) logarithm of the significand M_x ($\log_2 M_x \in [0,1)$), (2) addition of E_x and $\log_2 M_x$ (concatenation of binary strings), (3) multiplication by Z, and (4) exponential of the result of the multiplication. For an efficient implementation, the operations involved must be overlapped. This requires a left–to–right most–significant digit first (MSDF) mode of operation and the use of a redundant representation[2].

A problem of the algorithm above is the range of the exponential function $2^{Z \times S}$. Digit–recurrence exponential algorithms require the argument to be in the interval $(-1,1)$, while $Z \times S$ is out of the range. To extend the range of convergence and guarantee the convergence of the algorithm, the integer and fractional parts of $Z \times S$

1. The n bits of the significand include the hidden bit; that means the least–significant bit (LSB) has a weight $2^{-(n-1)}$.

2. We use a radix–r signed–digit representation with a maximally redundant digit set $\{-(r-1), \ldots, 0, \ldots (r-1)\}$.

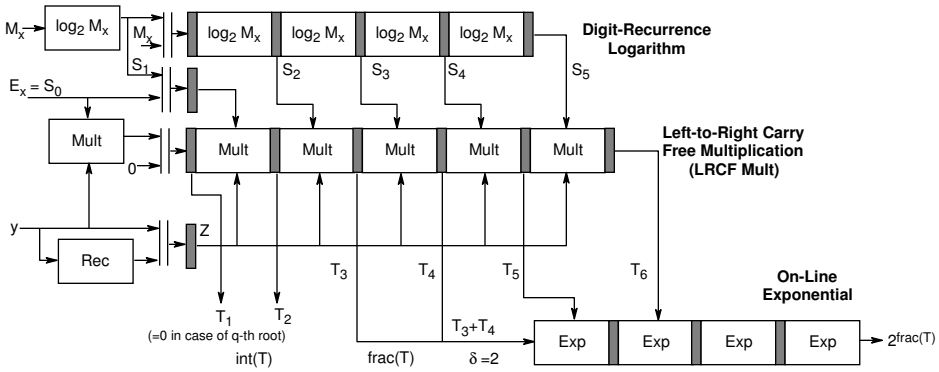

Fig. 1. Sequence of operations of the algorithm with a fixed–point exponent

must be extracted serially and Equation (4) must be rewritten,

$$X^Z = 2^{Z\times S} = 2^{int(Z\times S)} \times 2^{frac(Z\times S)}. \quad (5)$$

Therefore, according to Equation (5) and considering $F = X^Z = M_f \times 2^{E_f}$, the significand M_f and the exponent E_f of X^Z are

$$M_f = 2^{frac(Z\times S)}, \quad E_f = int(Z \times S). \quad (6)$$

The argument of the exponential $2^{frac(Z\times S)}$ is now in $(-1, 1)$.

Note that the number of integer bits of $Z\times S$ is larger for X^y than for $X^{1/y}$. In case of root extraction, the number of integer bits depends only on E_x; but in powering it depends moreover on y.

In summary, as illustrated in Figure 1 for single–precision (or *binary32* [9]) computation, radix $r = 2^b = 128$ and an 8–bit input operand y ($n_y = 7$), the algorithm[3] consists of the following steps below, and the precision and latency required for each step are analyzed in Section 5.

As will be shown later in Section 5, the number of stages of the logarithm and the multiplication are different for powering and root extraction; in fact, in this case the calculation of the powering function needs one more logarithm and multiplication stage than the root extraction.

3. The algorithm is valid for any precision in X and for any radix $r \geq 8$; to make the explanations more precise we particularize for $r = 128$. Later, the influence of the radix on the cycle time and area will be discussed.

In order to accommodate these two different datapaths, with different number of stages for logarithm and multiplication, and different number of integer digits, several multiplexers have been placed in the first stage.

Therefore, the steps of the algorithm are:

1) Evaluation of $Z = (-1)^{s_y} \times 1/|y|$ (only if root is being extracted, module *rec* in Figure 1), s_y being the sign of y. For practical cases, a low precision value for $|y|$, $n_y \leq 12$, is enough and a lookup table (LUT) of n_y inputs and n_z outputs (n_z fractional bits, non–redundant binary representation) is preferable for the computation of $1/|y|$. For larger precision y, $n_y > 12$, a digit–recurrence algorithm is required for the computation of $1/y$ [21] or the algorithm for powering and root calculation with a floating–point exponent, described in Section 4, might be used by representing y as floating–point operand.
2) Evaluation of the logarithm $L = \log_2 M_x \in [0, 1)$ to a precision of n_l bits using a high-radix digit-recurrence algorithm. The logarithm is in a signed–digit radix–r representation. Note that, as the logarithm in the powering function needs one more stage than in root extraction, the first stage is skipped in the case of root extraction.
3) Multiplication $T = Z\times S$. Operand $S = E_x + L = \sum_{i=-\lceil (n_{E_x}-1)/b\rceil+1}^{\lceil n_l/b\rceil} S_i \; r^{-i}$ is obtained by concatenating the digits of E_x (integer digits), recoded to a signed–digit radix–r representation, and L (fractional digits). The

multiplication is evaluated using an LRCF (left-to-right carry-free) multiplier [5].

4) Serial extraction of the integer $int(T)$ and fractional $frac(T)$ parts of T, and on-the-fly conversion of $int(T)$ to a non-redundant representation. The number of integer digits is $\lceil (n_{E_x} - 1 + n_y)/b \rceil$ for powering and $\lceil (n_{E_x} - 1)/b \rceil$ for root extraction.
5) On-line high-radix exponential $2^{frac(T)} \in (0.5, 2)$ with $frac(T) \in (-1, 1)$, precision of n_e bits, and on-line delay $\delta = 2$. The redundant result is normalized and rounded to n bits using an on-the-fly rounding unit [6].

For the example in Figure 1 ($b = 7$, $n_y = 7$, $n_{E_x} = 8$) the number of digits in the integer part is 2 in the case of powering and 1 in the case of root extraction. Since root extraction needs to compute $Z = 1/y$, the number of cycles required to obtain the integer part of both algorithms is the same. This holds for any radix and precision combination if $n_y \leq b$. The case $n_y > b$ is not adequate for this combined architecture in terms of latency, but it can be avoided by selecting a larger radix.

Consequently, the total latency for $n_y \leq b$ is given by

$$N = (\lceil (n_{E_x} - 1)/b \rceil + 1) + (\delta + 1) + N_e \qquad (7)$$

where $N_e = \lceil n_e/b \rceil$ is the latency of the exponential $2^{frac(T)}$.

Table 1 shows a simulation example of the algorithm for root extraction, S_i, T_i and F_i being the digits of the logarithm, the LRCF multiplication and the exponential, respectively. As it can be seen, the obtained result has the accuracy required for the simple–precision representation.

3.2 Architecture

Figure 2 shows the general block diagram of the architecture implementing the proposed algorithm. This Figure only shows the main blocks involved in the computation, without any time and overlapping considerations among blocks. Those considerations are illustrated in Figure 1.

Single thick lines represent long-word operands (around n bits), single thin lines represent short-word operands (around b or n_{E_x} bits), and double lines represent redundant signed radix–r digits. The high-radix logarithm, LRCF multiplication, and on-line high-radix exponential units are similar to those implemented in [15]. A detailed description of an on-the-fly conversion unit can be found in [6].

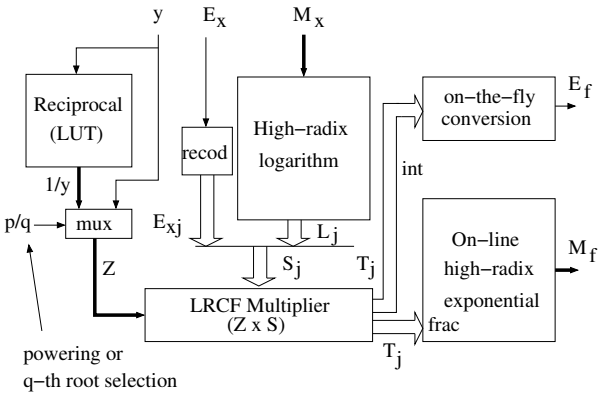

Fig. 2. Block diagram of the architecture with fixed–point exponent

To allow faster execution of iterations in these units we opted for representing all variables in a redundant borrow-save representation. This representation allows an easier conversion of signed radix–r digits than the carry–save representation. Moreover, a borrow-save adder can be implemented as a carry-save adder with some inverted inputs and outputs

4 Algorithm for Powering and Root Calculation with a Floating–Point Exponent

In this section the algorithm for powering and root extraction, described in Section 3, is extended to deal with a floating–point exponent. This way, the function to be computed is X^Y or $X^{1/Y}$, X and Y being floating–point numbers, $X = (-1)^{s_x} \times M_x \times 2^{E_x}$, $Y = (-1)^{s_y} \times M_y \times 2^{E_y}$.

Note that depending on the sign of exponent E_y the operation being computed can be powering or root extraction in both cases.

Denormals are easily handled as well (see [22] for details); in this case no additional iteration is required.

TABLE 1
Simulation of single–precision root extraction for radix $r = 128$, with $X = 0.039632357657$ and $y = 5$

Simulated Results for Intermediate Operations ($z = 1/y = 0.200000$)														
$S = E_x + \log_2(M_x) = -5 + 0.3428226009$					$T = z \times S = -0.9314354807$						$F = 2^{frac(T)} = 0.5243363641$			
S_1	S_2	S_3	S_4	S_5	T_1	T_2	T_3	T_4	T_5	T_6	F_0	F_1	F_2	F_3
-5	43	113	-24	-114	0	-1	8	99	46	28	67	14	93	-7
Simulated SP Rounded Result $RN(F) = 0.5243363380$, Exact result $X^{1/y} = 0.5243363683\ldots$, Error $< 2^{-23}$.														

4.1 Algorithm

Our starting point is Equation (1). Replacing the exponent by a floating–point exponent Y,

$$|X|^Y = 2^{(-1)^{s_y} \times M_y \times \log_2 |X| \times 2^{E_y}}. \quad (8)$$

Similarly,

$$|X|^{1/Y} = 2^{(-1)^{s_y} \times (1/M_y) \times \log_2 |X| \times 2^{-E_y}}. \quad (9)$$

In order to use the same multiplier for both operations, $1/M_y \in (0.5, 1]$ is normalized in $[1, 2)$,

$$|X|^{1/Y} = 2^{(-1)^{s_y} \times (2/M_y) \times \log_2 |X| \times 2^{-(E_y+1)}}. \quad (10)$$

As for the fixed–exponent case, to guarantee the convergence of the algorithm, the integer and fractional parts are extracted serially,

$$|X|^Z = M_f \times 2^{E_f} = 2^{frac(T)} \times 2^{int(T)} \quad (11)$$

with $Z = Y$ or $Z = 1/Y$. Defining an intermediate significand and exponent (M_z, E_z) pair as

$$(M_z, E_z) = \begin{cases} (M_y, E_y) & \text{for powering} \\ (2/M_y, -(E_y + 1)) & \text{for root} \end{cases} \quad (12)$$

then, for powering and root extraction,

$$T = (-1)^{s_y} \times M_z \times \log_2 |X| \times 2^{E_z}. \quad (13)$$

The main difference with the fixed–point exponent case is the additional term 2^{E_z}, which is a left or right shift of $\log_2 |X|$. The maximum shift amount depends basically on the position of the left-most significant (non-zero) digit of $\log_2 |X|$ (see Section 4.2).

With these considerations, the sequence of operations for single precision and $r = 128$ ($b = 7$) is shown in Figure 3. The precision required in every step is obtained in Section 5 and the sequence of operations is summarized in the steps below[4]:

4. As these steps are somewhat similar to the steps in Section 3.1, they are described with fewer details.

1) Evaluation of $R = (2/My)$, only in case of root extraction, by means of a digit–recurrence algorithm [21]. The latency is $N_r = \lceil n_r/b \rceil$ for n_r bits of accuracy.
2) Digit–by–digit computation of the logarithm $L = E_x + \log_2 M_x$. A low–latency radix–10 digit–recurrence algorithm for the computation of the logarithm has been proposed in [4]; we have extended it to binary and any radix [22]. When $|X|$ is close to 1, $\log_2 |X|$ has a significant number of leading zeros (if $|X| \geq 1$) or ones (if $|X| < 1$); this number can be estimated and the corresponding iterations of the logarithm skipped.
   Hence, an initial iteration, range reduction step in the figure, is introduced to determine the leading zeros/ones of the fractional part of L, l_x, and the number of zero digits of the integer part K_{E_x} by using Leading–Zero detectors (LZD) or Leading–One detectors (LOD), and the number of skipped iterations $K = \lfloor (l_x - 1)/b \rfloor$. After that, the logarithm is computed with $n_l = n + n_{E_x} + 6 + b$ precision bits; this requires $N_l = \lceil (n + n_{E_x} + 6)/b \rceil + 1$ iterations.
3) Shifting L by 2^{E_z}, $S = L \times 2^{E_z}$. The shift implementation is described in Section 4.3.
4) Left–to–right carry–free multiplication $T = M_z \times S$, starting in cycle 5 with on–line delay $\delta_m = 1$. An additional most significant digit T_0 is computed for detecting overflow ($T_0 \neq 0$ for overflow, see Section 4.2).
5) On–line exponential $2^{frac(T)}$, starting in cycle 7, because the on–line delay of the exponential is $\delta = 2$.

The latency of the algorithm is

$$N = 5 + \gamma + \delta_m + \delta + N_e \quad (14)$$

where $\delta = 2$, $\delta_m = 1$ (for root extraction and the combined operation), $\gamma = \lceil (n_{E_x} - 1)/b \rceil$ and N_e is the latency of the exponential operation.

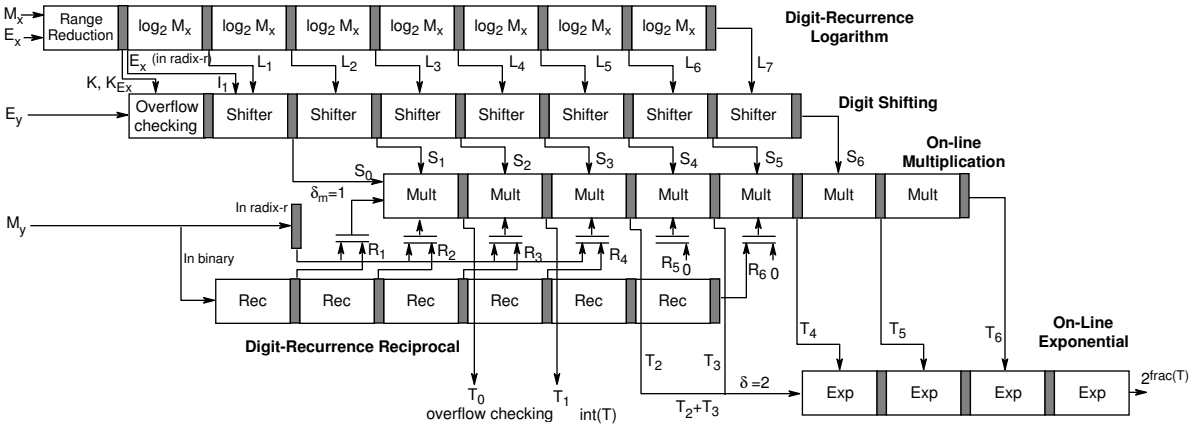

Fig. 3. Block diagram for the computation of X^Y and $X^{1/Y}$, X and Y being single-precision floating–point numbers, with a radix 128.

4.2 Bounds for E_y

Shift 2^{E_z} (see Equation (12) and Equations (8) and (10)) imposes a limitation to the range of supported Y values. That is, the shift cannot produce either a result larger than the maximum or lower than the minimum representable floating–point number. The practical values for E_z can be limited as follows:

- Examining the overflow (if $T \geq 0$) or underflow (if $T < 0$) conditions it is possible to determine the maximum value for $|E_z|$. Overflow and underflow occur if $E_f > E_{max}$ and if $E_f < E_{min}$, respectively.
  The most restrictive case is the overflow when $|X| \in \{1 - 2^{-l_x-1}, 1 + 2^{-l_x-1}\}$, with l_x the number of fractional leading zeros or ones of M_x, which leads to the following condition,
  $$int(M_z \times \log_2 |X| \times 2^{E_{z\ max}+1}) > 2^{n_{E_x}-1} - 1. \quad (15)$$
  Then, taking into account that $\log_2(1 + 2^{-l_x-1}) \geq 2^{-l_x-1}$, and $|\log_2(1 - 2^{-l_x-1})| \geq 2^{-l_x-1}$, and $M_z = 1$,
  $$E_{z\ max} = n_{E_x} + l_x \leq n + n_{E_x} - 2. \quad (16)$$
  Larger values of E_z will produce overflow in the exponent of the result. Obviously, lower values of E_z could also produce an overflow depending on the value of X.
- The minimum value for E_z is determined by the condition $frac(T) \geq 2^{-n_m}$, n_m being the precision of $frac(T)$. Therefore,
  $$frac(M_z \times \log_2 |X| \times 2^{E_z}) \geq 2^{-n_m}.$$
  Replacing $M_z = 2$, $E_x = E_{max}$ and $M_x = 2$,
  $$E_{z\ min} = -(n_{E_x} + n_m). \quad (17)$$

Therefore, from Equations (16) and (17), the practical range of E_y for powering is limited to

$$-(n_{E_x} + n_m) \leq E_y \leq n + n_{E_x} - 2. \quad (18)$$

For root extraction, the practical range of E_y is limited to

$$-(n + n_{E_x} - 1) \leq E_y \leq n_{E_x} + n_m + 1. \quad (19)$$

Consequently, $-69 \leq E_y \leq 61$ ($-62 \leq E_y \leq 70$) and $-37 \leq E_y \leq 29$ ($-30 \leq E_y \leq 38$) for powering (root extraction) in double–precision and single–precision floating–point representation, respectively.

4.3 Shifting the logarithm

Figure 4(a) shows the format of $L = E_x + \log_2 M_x$. Due to the addition of E_x, there is an integer part of $\gamma = \lceil (n_{E_x} - 1)/b \rceil$ radix–r digits, the leading K_{Ex} of which are zeros. The fractional part has $K = \lfloor (l_x - 1)/b \rfloor$ radix–r leading zeros followed by N_l digits. The non–zero radix–r digits of the integer and fractional parts are denoted by $I_1, \ldots, I_{\gamma - K_{Ex}}$ and $L_1, \ldots, L_{N_l}$, respectively. Note that the leading zeros of the $\log_2 M_x$ are skipped over during its computation then these digits are

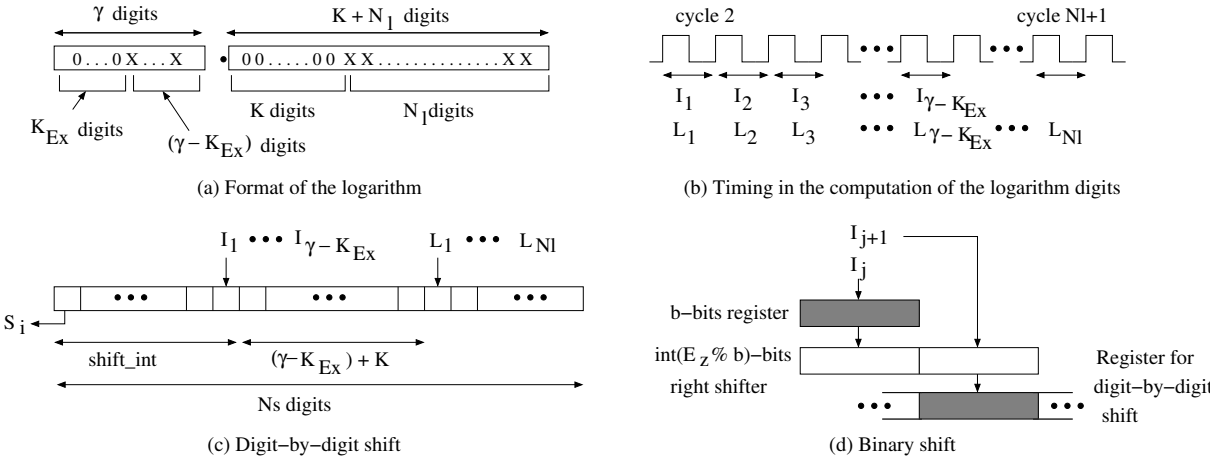

Fig. 4. Shifting the logarithm

not computed but are represented in the figure for a better comprehension of the shifting. The first non zero digit of the integer and fractional parts of L are obtained simultaneously in cycle 2 (Figure 4(b)).

The E_z–bit left or right shift is implemented as a right shift of L, with L prealigned $K_{E_x}+1$ (if there is a non-zero integer part) or $\gamma+K+1$ (if the integer part is zero) digits to the left, the possible maximum left shift. The shift is split in two parts: (1) a digit–by–digit right shift of $(K_{E_x}+1)-\lfloor E_z/b \rfloor$ or $(K+\gamma+1)-\lfloor E_z/b \rfloor$ radix–r digits and (2) a binary left shift of $E_z\%b$ bits ($E_z\%b$ is always positive).

The digit–by–digit shift is carried out in a displacement register with N_s digits (Figure 4(c)), where N_s is roughly equal to N_l. All the integer digits I_j enter at the same position of the register but in consecutive cycles. The same for the fractional digits L_j. On the other hand, digit L_j enters $(\gamma-K_{E_x})+K+1$ positions to the right of digit I_j. The digits are left shifted out, one digit every cycle.

The position where the I_j digits input the register is determined in terms of K_{E_x} and E_z. Two different cases are identified:

1) The integer part is not zero, $\gamma \neq K_{E_x}$. The maximum allowed left shift is K_{E_x}. Then, digits I_j input the register in position $K_{E_x}-\lfloor E_z/b \rfloor+1$ and the output of the register has $K_{E_x}-\lfloor E_z/b \rfloor$ leading zeros/ones digits.
2) The integer part is zero, $\gamma = K_{E_x}$. The maximum allowed left shift is $\gamma+K$. Then, the L_i digits are introduced at position $\gamma+K+1-\lfloor E_z/b \rfloor$. The output of the register produces $\gamma+K-\lfloor E_z/b \rfloor$ leading zeros/ones digits.

The shifted logarithm S has $N_s \leq N_l+1$ digits. The most significant digit S_0 is for detecting overflow[5], the following γ radix–r digits correspond to the integer part of the shifted logarithm and the remaining $K+N_l$ radix–r digits correspond to the fractional part.

As shown in Figure 4(d), the $E_z\%b$–bit binary shift is carried out by storing two consecutive digits, I_j and I_{j+1} or L_j and L_{j+1}, in a $2b$–bit left shifter, and shifting the $2b$ bits to the left and discarding the b least–significant bits. The maximum binary shift amount is b bits.

5 Error Analysis

In this section, the error analysis for the computation of the powering and root functions with integer and floating–point exponent and a radix $r \geq 8$ is presented. A direct result obtained from this error analysis is the precision and minimum latency required for every intermediate operation and the global latency.

5. If $T_0 = S_0 \times M_z \neq 0$ or $E_z > E_{z\ max}$, then the result overflows.

We denote $F = X^Z$ the operation to be evaluated, Z being a fixed–point or a floating–point exponent. F is evaluated with Equations (5) and (11) and an approximation $\widehat{F}$ is obtained[6] with the following contributions to the error, represented by the different ε's:

1) Reciprocal (*only in case of root extraction*). The output of the reciprocal unit is $1/y$ $(or\ 1/M_y) + \varepsilon_{rec}$.
2) Logarithm $L = E_x + \log_2 M_x$. The output is $\widehat{L} = L + \varepsilon_{log}$.
3) Shift $S = L \times 2^{E_z}$ (*only in case of floating–point exponent*). The output is given by $\widehat{S} = S + \varepsilon_{log} \times 2^{E_z} + \varepsilon_{sh}$.
4) Multiplication $T = Z \times S$ (fixed–point exponent) or $T = M_z \times S$ (floating–point exponent). The result is $\widehat{T} = T + \varepsilon_t$, ε_t being different for fixed–point exponent,

$$\varepsilon_t = \begin{cases} Z \times \varepsilon_{log} + \varepsilon_{mul} & \text{for powering} \\ Z \times \varepsilon_{log} + S \times \varepsilon_{rec} + \varepsilon_{log}\varepsilon_{rec} + \varepsilon_{mul} & \text{for root extraction} \end{cases} \quad (20)$$

or floating–point exponent,

$$\varepsilon_t = \begin{cases} \varepsilon_{log} \times M_z \times 2^{E_z} + M_z \times \varepsilon_{sh} + \varepsilon_{mul} & \text{for powering} \\ \varepsilon_{log} \times M_z \times 2^{E_z} + M_z \times \varepsilon_{sh} + \\ +\varepsilon_{rec}(S + \varepsilon_{log} \times 2^{E_z} + \varepsilon_{sh}) + \varepsilon_{mul} & \text{for root extraction.} \end{cases} \quad (21)$$

5) Separation of the integer and fractional parts of $\widehat{T}$

$$\begin{aligned} int(\widehat{T}) &= int(T + \varepsilon_t) \\ fract(\widehat{T}) &= T + \varepsilon_t - int(T + \varepsilon_t). \end{aligned} \quad (22)$$

6) Exponentiation $2^{frac(T)}$. The output of the exponential is $2^{frac(\widehat{T})} + \varepsilon_{exp}$.

Note that the only difference between the analysis of the algorithms with fixed–point and floating–point exponents is the error due to the shifter in ε_t (see Equations (20) and (21)).

Then, taking into account $int(T + \varepsilon_t) = int(T) + int(frac(T) + \varepsilon_t)$ and Equation (22)

$$\begin{aligned} \widehat{F} &= (2^{frac(T_\alpha)} + \varepsilon_{exp}) \times 2^{int(T_\alpha)} \\ &= 2^{T+\varepsilon_t} + \varepsilon_{exp} \times 2^{int(T+\varepsilon_t)} \\ &= F \times (2^{\varepsilon_t} + \varepsilon_{exp} \times 2^{int(frac(T)+\varepsilon_t)-frac(T)}). \end{aligned}$$

Then,

$$|\widehat{F} - F| = |F \times (-1 + 2^{\varepsilon_t} + \varepsilon_{exp} \times 2^{int(frac(T)+\varepsilon_t)-frac(T)})|. \quad (23)$$

Since we want to provide faithful rounding [17], the rounded result $RN(\widehat{F})$ must be at a distance less than $1\ ulp$ of the exact result F when $(1 + (1/4)2^{-n+1}) \leq M_f < 2$ and less than $0.5\ ulp$ when $1 \leq M_f < (1 + (1/4)2^{-n+1})$ [22]. Note that $ulp(F) = 2^{-(n-1)} \times 2^{E_f}$.

Assuming rounding to the nearest (even), $|\widehat{F} - RN(\widehat{F})| \leq 0.5\ ulp(\widehat{F})$ and then, the necessary and sufficient conditions for faithful rounding are

$$|\widehat{F} - F| < \begin{cases} \frac{1}{2} ulp(F) & \text{if } 1 + 2^{-n-1} \leq M_f < 2 \\ \frac{1}{4} ulp(F) + (F - 2^{E_f}) & \text{if } 1 \leq M_f < 1 + 2^{-n-1}. \end{cases} \quad (24)$$

Since $frac(T) \in (-1, 1)$ is expressed in signed-digit representation, then $2^{frac(T)} \in (0.5, 2)$ is not normalized. The normalized significand and the exponent are given by $M_f = 2^{frac(T)+\sigma}$ and $E_f = int(T) - \sigma$, where $\sigma = 1$ if $frac(T) < 0$ and $\sigma = 0$ if $frac(T) \geq 0$.

Next, we compute an upper bound for Equation (24). Note that $int(frac(T) + \varepsilon_f) \leq 1$ for $\sigma = 0$, and $int(frac(T) + \varepsilon_f) \leq 0$ for $\sigma = 1$. Therefore, $int(frac(T) + \varepsilon_t) - frac(T) \leq 1 - (frac(T) + \sigma)$. Moreover, considering $\varepsilon_t << 1$, the approximation $2^{\varepsilon_t} = e^{\ln(2)\varepsilon_t} \approx 1 + \ln(2)\varepsilon_t + O(\varepsilon_t^2) \leq 1 + \varepsilon_t$ holds; therefore,

$$|\widehat{F} - F| \leq |F \times (\varepsilon_t + (2/M_f) \times \varepsilon_{exp})|. \quad (25)$$

Introducing this bound in (24) and using $M_f < 2$ in the first equation and $M_f \geq 1$ in the second one, the following worst-case condition for faithful rounding is obtained,

$$|\varepsilon_t + 2 \times \varepsilon_{exp}| < 2^{-(n+1)}. \quad (26)$$

The critical parameter to be minimized first is ε_{exp} since it is directly related to the latency of

6. The *hat* is used to denote an approximated value.

the algorithm. Hence, the minimum upper bound for ε_{exp} is:

$$|\varepsilon_{exp}| \leq 2^{-(n+3)} \quad (27)$$

then, replacing in Equation (26), we get

$$|\varepsilon_t| < 2^{-(n+2)}. \quad (28)$$

There are different contributions to ε_t depending on the function being computed and the exponent type (see Equations (20) and (21)):

- **Powering with fixed–point exponent**. The highest value of ε_t is obtained with $|Z| = 2^{n_y} - 1$, then

$$|(2^{n_y} - 1) \times \varepsilon_{log} + \varepsilon_{mul}| < 2^{-(n+2)}. \quad (29)$$

  A good trade-off between error components is achieved with $\varepsilon_{log} \leq 2^{-n_y} \times \varepsilon_{mul}$ and consequently

$$|\varepsilon_{mul}| \leq 2^{-(n+3)} \quad (30)$$
$$|\varepsilon_{log}| \leq 2^{-(n+n_y+3)}. \quad (31)$$

- **Root extraction with fixed–point exponent**. Considering $|S| < 2^{n_{E_x}-1}$, $|Z| \leq 1$, we have the same upper bound ε for ε_{log} and ε_{mul}, and $\varepsilon_{rec} \leq 2^{-(n_{E_x}-1)} \times \varepsilon$. Then,

$$|(3 + \varepsilon 2^{-(n_{E_x}-1)}) \times \varepsilon| < 2^{-(n+2)} \quad (32)$$

  hence,

$$|\varepsilon_{log}| \leq 2^{-(n+4)} \quad (33)$$
$$|\varepsilon_{mul}| \leq 2^{-(n+4)} \quad (34)$$
$$|\varepsilon_{rec}| \leq 2^{-(n+n_{E_x}+3)}. \quad (35)$$

- **Powering with floating–point exponent**. Considering $M_z < 2$, $2^{E_z} \leq 2^{E_{z\ max}}$, $\varepsilon_{mul} \leq \varepsilon$, $\varepsilon_{sh} \leq \varepsilon$, and $\varepsilon_{log} \leq 2^{-(E_{z\ max}+1)} \times \varepsilon$ in Equation (21),

$$\varepsilon_{mul} \leq 2^{-(n+4)} \quad (36)$$
$$\varepsilon_{sh} \leq 2^{-(n+4)} \quad (37)$$
$$\varepsilon_{log} \leq 2^{-(n+5+E_{z\ max})}. \quad (38)$$

- **Root Extraction with floating–point exponent**. The bounds for ε_{mul} and ε_{sh} are as in equations (36) and (37). Finally, considering $|S| < 2^{n_{E_x}-1}$, $\varepsilon_{log} \leq 2^{-(E_{z\ max}+2)} \times \varepsilon$, and $\varepsilon_{rec} \leq 2^{-n_{E_x}}\varepsilon$,

$$\varepsilon_{log} \leq 2^{-(n+6+E_{z\ max})} \quad (39)$$
$$\varepsilon_{rec} \leq 2^{-(n+4+n_{E_x})}. \quad (40)$$

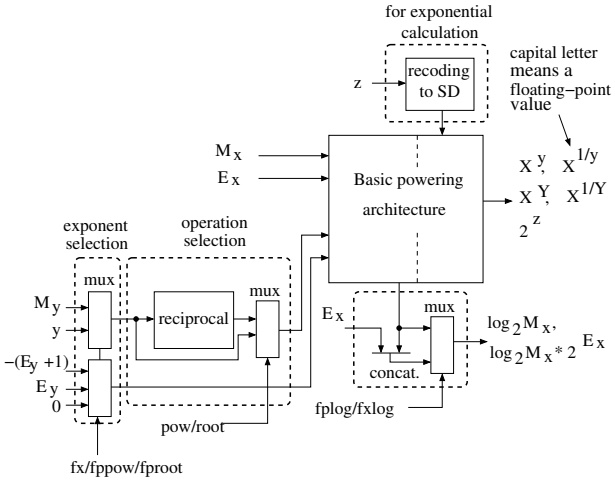

Fig. 5. Block diagram of the top–level architecture

The required precision and minimum latency values for each intermediate operation and the latency for the complete operation are shown in Tables 2(a) to 2(d). These values are provided for single (SP) and double (DP) precision with $r = 128$. To estimate the precision and minimum latency of the logarithm (n_l, N_l) we use the bound $E_{z\ max} \leq n_{E_x} + l_x$ in expressions (38) and (39).

6 Top Level Architecture

Figure 5 shows the block diagram of the architecture for powering and root extraction, together with some other functions that can be easily evaluated with the architecture. The functions that the proposed architecture can evaluate are powering and root extraction with fixed–point (integer) exponent or with floating–point exponent, exponential with integer exponent[7] (2^z), and base–2 logarithm with fixed–point or floating-point argument.

Additionally, the architecture can be extended to the computation of logarithms to any other base [22], that is,

$$\log_a X = \frac{1}{\log_2 a} \times \log_2 X.$$

7. Note that the exponential can be computed as a particular case of the powering function. However, the use of the on–line exponential implicit in the architecture allows us to obtain that function with a lower latency

TABLE 2
Precision and latency for powering computation and root extraction for $r = 128$, $b = log_2(r)$, $\delta = 2$, $\delta_m = 1$. ($(n_r,\ N_r)$, $(n_l,\ N_l)$, $(n_m,\ N_m)$, $(n_s,\ N_s)$, $(n_e,\ N_e)$: precision and latency for reciprocal, logarithm, multiplication, shift, and exponential, respectively.)

Target	Precision (bits)			Latency (cycles)			
	n_l	n_m	n_e	N_l	N_m	N_e	N_{pow}
$(n,\ n_{E_x},\ n_y)$	$n + n_y + 3$	$n + 3$	$n + 3$	$\lceil n_l/b \rceil$	$\gamma + \lceil n_m/b \rceil$	$\lceil n_e/b \rceil$	$1 + \delta + \gamma + \lceil n_e/b \rceil$
SP (24, 8, 7)	34	27	27	5	6	4	9
DP (53, 11, 7)	63	56	56	10	11	8	14

(a) Powering with fixed–point exponent ($\gamma = \lceil (n_{E_x} - 1 + n_y)/b \rceil$, $n_y \leq b$)

Target	Precision (bits)				Latency (cycles)				
	n_r	n_l	n_m	n_e	N_r	N_l	N_m	N_e	N_{root}
$(n,\ n_{E_x})$	$n + n_{E_x} + 3$	$n + 4$	$n + 4$	$n + 3$	1	$\lceil n_l/b \rceil$	$\gamma + \lceil n_m/b \rceil$	$\lceil n_e/b \rceil$	$2 + \delta + \gamma + \lceil n_e/b \rceil$
SP (24, 8)	35	28	28	27	1	4	5	4	9
DP (53, 11)	67	57	57	56	1	9	11	8	14

(b) Root extraction with fixed–point exponent ($\gamma = \lceil (n_{E_x} - 1)/b \rceil$, $n_y \leq b$)

Target	Precision (bits)			Latency (cycles)			
	n_l	n_s & n_m	n_e	N_l	N_s & N_m	N_e	N_{pow}
$(n,\ n_{E_x})$	$n + n_{E_x} + 5 + b$	$n + 5$	$n + 3$	$\lceil n_l/b \rceil$	$\gamma + 1 + \lceil n_m/b \rceil$	$\lceil n_e/b \rceil$	$5 + \gamma + \delta + N_e$
SP (24, 8)	44	29	27	7	7	4	12
DP (53, 11)	76	58	56	11	12	8	17

(c) Powering with floating–point exponent ($\gamma = \lceil (n_{E_x} - 1)/b \rceil$)

Target	Precision (bits)			Latency (cycles)			
	n_l	n_s & n_m	n_e	N_l	N_s & N_m	N_e	N_{root}
$(n,\ n_{E_x})$	$n + n_{E_x} + 6 + b$	$n + 5$	$n + 3$	$\lceil n_l/b \rceil$	$\gamma + 1 + \lceil n_m/b \rceil$	$\lceil n_e/b \rceil$	$5 + \gamma + \delta_m + \delta + N_e$
SP (24, 8)	45	29	27	7	7	4	13
DP (53, 11)	77	58	56	11	12	8	18

(d) Root extraction with floating–point exponent ($\gamma = \lceil (n_{E_x} - 1)/b \rceil$). The precision and latency required in the digit–recurrence reciprocal are $n_r = n + n_{E_x} + 4$ and $N_r = \lceil n_r/b \rceil$, which result in 36 and 68 bits of precision for SP and DP, respectively and 6 and 10 cycles for SP and DP, respectively

A look-up table (LUT) to store constants $1/\log_2 a$ and a multiplier (not shown in the Figure) are required. Moreover, e^X function, X being a floating–point or fixed–point number, can be calculated as well with a dual pass through the architecture,

$$e^X = (2^X)^{\log_2 e}.$$

Coming back to Figure 5, note that several muxes have been incorporated to select the exponent type (*fx/fppow/fproot*), the operation (*pow/root*) and the argument type for the logarithm (*fplog/fxlog*). The output of the *basic powering architecture*, which basically corresponds to the sequence of operations in Figure 3, provides one of the five operations. Note that in the case of root extraction with a floating–point exponent, the reciprocal calculation is integrated in the basic architecture and the *reciprocal* module is not used.

The number of iterations in the logarithm and the LRCF multiplier can be different for the computation with fixed–point or floating–point exponent; therefore, some logic, not shown in the Figure, must be introduced in *basic powering architecture* to accommodate both operations in the same architecture

For the exponential operation 2^z, as the algorithm for the powering computation uses an on-line exponential, z must be recoded to a signed–digit representation.

TABLE 3
Area and delay of the combined architecture with 8-bit signed integer exponent

Radix	Cycle (♯FO4)	Single–precision			Double–precision		
		Latency	Exec. time (♯FO4)	Total area (NAND2)	Latency	Exec. time (♯FO4)	Total area (NAND2)
8	31	17	527	14,000	28	868	30,000
16	33	14	462	17,000	22	726	34,000
32	33	12	396	19,000	18	594	45,000
64	34	11	374	24,000	16	544	54,000
128	34	9	306	32,000	14	476	78,000
256	36	9	324	50,000	13	468	121,000
512	36	8	288	75,000	13	468	206,000
1024	37	8	296	135,000	11	407	332,000

TABLE 4
Area and delay of the combined architecture with a floating-point exponent

Radix	Cycle (♯FO4)	Single–precision			Double–precision		
		Latency	Exec. time (♯FO4)	Total area (NAND2)	Latency	Exec. time (♯FO4)	Total area (NAND2)
8	31	20	620	24,000	31	961	50,000
16	33	17	561	29,000	25	825	59,000
32	33	16	528	33,000	21	693	73,000
64	34	15	510	40,000	20	680	89,000
128	34	13	442	52,000	18	614	122,000
256	36	13	468	74,000	17	612	190,000
512	36	12	432	107,000	17	612	300,000
1024	37	12	444	185,000	15	555	460,000

7 Evaluation and comparison

In this section we present estimates of the execution time and hardware cost for the proposed architectures and a comparison with other representative implementations.

7.1 Evaluation

The estimates are obtained with an approximate model for the cost and delay of the main logic blocks used, based on a simplification of the logical effort method [19]. 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 and has been widely used in technology-independent comparisons. The delays are expressed in FO4 units (delay of an inverter with a fanout of 4 inverters), and the area in numbers of equivalent minimum size NAND2 gates.

To give a better understanding, the execution time, cycle time and area of a single-precision floating–point multiply-add fused (MAF) unit [7], obtained using the logical effort model, is provided. Thus, the execution time and the cycle time of the MAF unit are 70 and 35 FO4 units, respectively, assuming a latency of 2 cycles, and the area is 8780 NAND2 gates.

Tables 3 and 4 show estimates of area and latency of the architectures for powering computation and root extraction for several radices. The latency of the radix–128 implementations can be derived almost directly from Figures 1 and 3. The latencies for the other radices are obtained with Equations (7) and (14). The delay and area of the different modules in the architectures have been obtained in [15].

In both cases, fixed–point and floating–point exponent, the cycle time corresponds to the critical path delay of the logarithm unit. The main contribution to the total area comes from both the high-radix logarithm and on–line exponential units. We observe that very little advantage in execution time and area is obtained from using very high radix values (over $r = 128$).

TABLE 5
Comparison for single–precision floating–point
($r = 128$, $n_y = b = 7$, $cycle = 34\ FO4$)

Architecture	Latency	Delay (FO4)	Area (NAND2)
Fixed–point exponent	9	306	32,000
Floating–point exponent	13	442	52,000
X^p with p integer [15]	9	306	29,827
$X^{1/q}$ with q integer [21]	9	306	28,300

7.2 Comparison

The comparison of our powering and root extraction method with previous alternatives is not easy. As far as we know, no other previously proposed algorithm and architecture allows the computation of the powering and root extraction for any fixed–point of floating–point exponent value.

Our powering algorithm is based on the implementation for an integer (fixed–point) exponent presented in [15], and its extension to the q–th root extraction in [21], q being an integer. The other architectures in the literature, based on table-driven polynomial approximations [3], [16], [18], [20] or digit–recurrence algorithms [11], are derived for a given value of the fixed–point exponent and changing it implies making a different implementation.

Therefore, in order to evaluate our architecture, we compare it with the architectures for the computation of X^p [15] and $X^{1/q}$ [21], with p and q integers. Table 5 shows the latency, delay and area estimate of every algorithm and implementation. We have considered a single–precision floating–point representation for the floating–point operands and radix $r = 128$. The numbers for our architecture and the architecture in [15] are quite similar, because these architectures are based on similar optimizations of the naive implementation.

8 Conclusions

An architecture for the computation of powering and root extraction functions, with fixed–point and floating–point exponents, logarithm and exponential, based on a high-radix composite algorithm for powering and root extraction, has been presented.

The algorithm consists of computing X^Z as $2^{Z \times (log_2 M_x + E_x)}$, Z being a fixed–point exponent, integer ($Z = y$) or fractional ($Z = 1/y$), or a floating–point exponent, through a sequence of overlapped operations: reciprocal, logarithm, multiplication, and exponential. The reciprocal, when needed, is obtained by means of a look–up table or a digit–recurrence algorithm for a fixed–point or floating–point exponent, respectively. The logarithm is computed in a high–radix digit–recurrence unit with selection by rounding. A high–radix left–to–right carry–free (LRCF) multiplier is used to obtain the intermediate multiplication. Finally, the exponential is computed by an on–line high-radix unit with selection by rounding. The integer part of the computation is used to obtain the result exponent. A sequential architecture has been proposed

Estimates of the execution times and hardware requirements have been obtained. These estimates are based on an approximate model for the delay and the area of the main components, for single and double-precision floating-point representations and several radices, from $r = 8$ to $r = 1024$. The analysis of the trade–offs between area and speed shows that the most efficient implementations correspond to a radix between $r = 32$ and $r = 128$; an implementation with radix $r = 128$ may be suitable when very high-speed processing is necessary, at the expense of greater hardware requirements. Larger radices require much higher hardware complexity and the execution time is not reduced so much.

Finally, the proposed architecture has been modified to allow the independent computation of logarithm and exponential, with lower latencies than those of powering and root extraction.

Acknowledgments

Work supported in part by Ministry of Science and Innovation of Spain, co-funded by the FEDER funds of the European Union, under contract TIN2010-17541, and by the Xunta de Galicia, Program for Consolidation of Competitive Research Groups ref. 2010/28.

REFERENCES

[1] D. Blythe, "Rise of the Graphics Processor". *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[2] S. Borkar and A.A. Chien, "The Future of Microprocessors". *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[3] J. Cao, B.W.Y. Wei, and J. Cheng, "High-Performance Architectures for Elementary Function Generation", *Proc. 15th IEEE Symp. Computer Arithmetic*, pp. 136–144, Jun. 2001.

[4] D. Chen, L. Han, Y. Choi, and S.-B. Ko, "Improved Decimal Floating-Point Logarithmic Converter Based on Selection by Rounding", *IEEE Trans. Computers*, vol. 61, no. 5, pp. 607–621, May 2012.

[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, "Digital Arithmetic", Morgan Kaufmann, Jun. 2003.

[8] D. Harris, "A Powering Unit for an OpenGL Lighting Engine", *Proc. Asilomar Conf. Signals, Systems and Computers*, pp. 1641–1645, Nov. 2001.

[9] IEEE Std 754(TM)-2008, "IEEE Standard for Floating-Point Arithmetic", *IEEE Computer Society*, Aug. 2008.

[10] A. Marovka, "Back to Thin–Core Massively Parallel Processors", *IEEE Computer Magazine*, vol. 44, no. 12, pp. 49–54, Dec. 2011.

[11] P. Montuschi, J.D. Bruguera, L. Ciminiera, and J.A. Piñeiro, "A Digit-by-Digit Algorithm for m–th Root Extraction", *IEEE Trans. Computers*, vol. 56, no. 12, pp. 1696–1706, Dec. 2007.

[12] J.-M. Muller, "Elementary Functions, 2nd Ed.", Birkhäuser, Nov. 2005.

[13] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlè, and S. Torres. *Handbook of Floating–Point Arithmetic*, Birkhäuser, Dic. 2009.

[14] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Philips, "GPU Computing", *Proc. IEEE*, vol. 96, no.5, pp. 879–899, May 2008.

[15] J.A. Piñeiro, M.D. Ercegovac, and J.D. Bruguera, "Algorithm and Architecture for Logarithm, Exponential and Powering Computation", *IEEE Trans. Computers*, vol. 53, no. 9, pp. 1085–1096, Sep. 2004.

[16] J.A. Piñeiro, S. Oberman, J.-M. Muller, and J.D. Bruguera, "High-Speed Function Approximation using a Minimax Quadratic Interpolator", *IEEE Trans. Computers*, vol. 54, no. 3, pp. 304-318, Mar. 2005.

[17] S.M. Rump, T. Ogita, and S. Oishi, "Accurate Floating-Point Summation. Part I: Faithful Rounding", *SIAM J. Sci. Comput.*, vol. 31 , no. 1, pp. 189–224, Oct. 2008.

[18] A.G.M. Strollo, D. De Caro, and N. Petra, "Elementary Functions Hardware Implementation Using Constrained Piecewise-Polynomial Approximations", *IEEE Trans. Computers*, vol. 60, no. 3, pp. 418–432, Mar. 2011.

[19] I.E. Sutherland, R.F. Sproull, and D. Harris, "Logical Effort: Designing Fast CMOS Circuits", Morgan Kaufmann, Feb. 1999.

[20] 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.

[21] A. Vázquez and J.D. Bruguera, "Composite Iterative Algorithm and Architecture for q–th Root Calculation", *Proc. 20th IEEE Symp. Computer Arithmetic*, pp. 52–61, 2011.

[22] A. Vázquez and J.D. Bruguera, "Reducing the Latency of Digit–by–Digit Floating–Point Logarithms", *Internal Report. Univ. Santiago de Compostela (SPAIN)*, Jun. 2012. www.ac.usc.es/system/files/TR-logarithm.pdf.

[23] 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, Sep. 1996.

Alvaro Vazquez graduated in Physics in 1997 and received the PhD. degree in Electronic and Computer Engineering in 2009 from the University of Santiago de Compostela, Spain. Currently, he is a postdoctoral researcher at the "Centro de Investigación en Tecnologías de la Información" (CITIUS), University of Santiago de Compostela. In 1998 he joined the Departamento de Electronica e Computacion at the University of Santiago de Compostela. He was a research visitor in the FPU Design Team at IBM R&D, Boeblingen, Germany, in 2008, for five months. From October 2009 to March 2011 he was an INRIA postdoc at the Laboratoire de l'Informatique du Parallelisme, ENS Lyon, France. His research interests are decimal floating-point arithmetic, design of high-speed and low-power numerical processors, FPGA-specific floating-point operators for reconfigurable computing, and algorithms and architectures for computing elementary functions. Dr. Vazquez is member of the IEEE and the IEEE Computer Society.

Javier D. Bruguera received the graduated 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 in the Department of Electronic and Computer Science at the University of Santiago de Compostela, Spain, and he is also with the "Centro de Investigación en Tecnologías de la Información" (CITIUS), University of Santiago de Compostela. Previously, he was an assistant professor at the University of Oviedo, Spain, from 1984 to 1986, and an assistant professor at the University of A Corunna, Spain, from 1987 to 1990. Dr. Bruguera has been serving as Chair of the Department between 2006 and 2010. He was a research visitor in the Application Center of Microelectronics at Siemens, Munich, Germany, in 1993, for five months, and in the Department of Electrical Engineering and Computer Science at the University of California at Irvine, from October 1993 to December 1994.

His primary research interests are in the area of computer arithmetic, processor design, and computer architecture. He is author/coauthor of nearly 150 research papers on journals and conferences. Dr. Bruguera has served on program committees for several IEEE, ACM, and other meetings. Dr. Bruguera is member of the IEEE, the IEEE Computer Society and ACM.