2011 20th IEEE Symposium on Computer Arithmetic

Composite Iterative Algorithm and Architecture for q-th Root Calculation*

Álvaro Vázquez
Laboratoire LIP, CNRS-ENSL-INRIA-UCBL
INRIA, France
Alvaro.Vazquez.Alvarez@ens-lyon.fr

Javier D. Bruguera
Dept. Electronic and Computer Science
Centro de Investigación en Tecnologías de la Información (CITIUS)
University of Santiago de Compostela, Spain
jd.bruguera@usc.es

***Abstract*—An algorithm for the q-th root extraction, q being any integer, is presented in this paper. The algorithm is based on an optimized implementation of $X^{1/q} = 2^{(1/q)\log_2(X)}$ by a sequence of parallel and/or overlapped operations: (1) reciprocal, (2) digit–recurrence logarithm, (3) left–to–right carry–free multiplication and (4) on–line exponential. A detailed error analysis and two architectures are proposed, for low precision q and for higher precision q. The execution time and hardware requirements are estimated for single precision floating–point computations for several radices; this helps to determine which radices result in the most efficient implementations. The architectures proposed improve the features of other architectures for q–th root extraction.**

I. Introduction

The design of functional units for the computation of q–th roots ($X^{1/q}$) has been a challenging task for years. The q–th root extraction includes some very frequent operations in computer graphics, digital signal processing and scientific computation, such as square root ($X^{1/2}$), reciprocal (X^{-1}), inverse square root ($X^{-1/2}$), cubic root ($X^{1/3}$) and inverse cubic root ($X^{-1/3}$), and some other less frequent but also important functions.

The traditional approximation to q–th roots extraction has been the development of functional units for the computation of a given root. This way, there is a number of algorithms and implementations for the two most frequent roots, the square root and the inverse square root calculation, including linear convergence digit–recurrence algorithms and quadratic convergence multiplicative-based methods, such as Newton-Raphson and Goldschmidt algorithms [4]. There are also several approaches for the calculation of other roots derived from the application of general methods for function evaluation to the case of root extraction.

In general, for the calculation of a q–th root with very low precision, it is possible to employ direct table look–up, but its high memory requirements make it an inefficient method for single– or double–precision floating–point formats. Polynomial and rational approximations are another way of implementing the q–th root extraction [7]. However, one of the most efficient methods in floating–point representation is table–driven algorithms, which are halfway between direct table look–up and polynomial and rational approximations. The use of a polynomial approximation allows the table size to be reduced and the table look–up allows us to reduce the degree of the polynomial.

A first order piecewise linear approximation based on a Taylor expansion for the calculation of X^p with $p = \pm 2^k$ or $p = \pm 2^{k_1} \pm 2^{-k_2}$, where k and k_1 are any integer and k_2 any nonnegative integer, is presented in [12]. A limited number of roots, square root, reciprocal square root, fourth root, etc., are included. This implementation requires, besides the table to store the coefficients, just one multiplier. Alternatively, second order polynomial approximations for the calculation of X^p, for any p, which includes any q–root, are presented in [1], [8]. In these cases, besides the look–up table, one or two multipliers and several adders are required.

On the other hand, a digit–recurrence method for the q–th root extraction is presented in [6] and particularized to the radix 2 cube root computation in [10]. The complexity of the resulting architecture depends on q, such as the larger q the larger the complexity, in such a way that the architecture for the computation of large q–th roots seems difficult to implement. Other digit–recurrence implementations for both square and cube root computations are described in [5], [13].

It has to be pointed out that all the methods outlined above for the extraction of a q–root are targeted for a given q. That means that the resulting architecture cannot be used for the calculation of a root different to that it has been designed for. To adapt the architecture to a different q–th root requires to change the look–up tables in the case of table–driven polynomial approximations, or to design a completely new architecture, in the case of the digit–recurrence method. Of course, the table–driven polynomial approximations can be adapted to compute more than just one q–th root, but this needs the replication of the look–up tables. In any case, the methods above cannot be considered as general methods for the calculation of any q–th root.

The only architecture in the literature for the q–th root extraction for any q, except the naive implementation of $X^{1/q}$ into a cascaded reciprocal-logarithm-multiplication-exponential chain, has been presented in [9]. This architecture was designed for the computation of the powering function X^p, with p any integer, based on a logarithm-

*Work supported in part by Ministry of Education and Science of Spain and FEDER funds under contract TIN 2007-67537-C03. J. D. Bruguera is a member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.16

Table I
SYMBOLS AND NOTATION

Symbol	Definition	Symbol	Definition
X	floating-point radicand $X = M_x \times 2^{E_x}$	n	precision bits of radicand and result
q	root degree ($q \in N$, $q \neq 0$)	n_{Ex}	precision bits of the exponent of X
M_q	normalized significand of q, $M_q \in [0.5, 1)$	E_q	Normalized exponent of q, $E_q \geq 0$
$X^{1/q}$	q-th root function computed as $2^{(1/q)\log_2(X)}$	n_q	precision bits of $\|q\|$ or M_q
M_z	Significand of the result	E_z	Exponent of the result
T	argument of the q-th root function $T = (1/q)\log_2(X)$	r	radix $r = 2^b$ ($b \in N^+$, $b \geq 3$)
$int(T)$	integer part of T	D_j, L_j, S_j^*, T_j, E_j	high-radix digits of intermediate operands
$frac(T)$	fractional part of T		
D	reciprocal of the root degree $D = 1/q$	n_d, n_l, n_s, n_m, n_e	precision bits of intermediate operands
S	logarithm of the radicand $S = \log_2(X) = E_x + \log_2(M_x)$		
S^*	logarithm of the radicand shifted by 2^{-E_q}	N_d, N_l, N_s, N_m, N_e	latencies of intermediate operations
L	logarithm of the radicand significand $\log_2(M_x)$		

Notation for operations: d or D for reciprocal, l or L for logarithm, m or T for multiplication, s or S^* for shifting, e or E for exp.

multiplication-exponential chain implementation speeded–up by using redundancy and on–line arithmetic, and extended to the computation of $X^{1/q}$. However, the extended architecture for the q–th root extraction is hard to implement, because in addition to the operations in the chain, it includes an integer division and requires the calculation of the remainder of the division.

In this paper we give a detailed description of an optimized composite iterative algorithm for the computation of $X^{1/q}$, for a floating–point operand $X = (-1)^{s_x} \times M_x \times 2^{E_x}$ and an integer operand q. The algorithm is based on the architecture presented in [9] for the computation of X^p, p being any integer, and the final result is computed as $X^{1/q} = 2^{(1/q)\times(E_x + \log_2 M_x)}$ 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. This formulation avoids the integer division and the remainder operation of the extension of the algorithm in [9] to the calculation of the q–th root. Besides, the proposed q-th root computation could be easily integrated into the dataflow of the powering architecture [9], allowing the evaluation of numerous powering and q-th root operations, or even logarithms and exponentials.

We propose two different implementations of the algorithm. First, we propose an algorithm for the computation of $X^{1/q}$ with low precision values of q, in such a way that $1/q$ can be obtained from a look–up table. Later, we modify the algorithm to overcome this limitation in the precision of q. We perform a detailed error analysis to determine the size of the intermediate operands and an analysis of the tradeoffs between area and speed for determining which radices result in the most efficient implementations.

The rest of the paper is structured as follows: we first focus on the algorithm for q–th root extraction for low precision q, giving a detailed description of the algorithm and its error analysis in Section II. The architecture implementing the algorithm is presented in Section III. In Section IV, we present the modification of the algorithm and the architecture to larger precision q values. The evaluation of the architecture for several different radices and the comparison with other implementations is given in Section V. Finally, the main conclusions are summarized in Section VI.

II. ALGORITHM FOR q-TH ROOT CALCULATION

In this Section we present a new composite algorithm for the computation of the q-th root function $X^{1/q}$ and its error analysis, X being a floating-point number and $q = (-1)^{s_q} abs(q)$ a $n_q + 1$-bit signed integer exponent (see Table I for symbols and notation used). This algorithm is based on the algorithm presented in [9] for the computation of the powering function X^q, although there are important differences between both algorithms.

The algorithm presented in this Section is intended for low precision values of q, or when only a reduced subset of q values is interesting for the application, such that $1/q$ can be obtained efficiently from a look-up table. In Section IV we propose a modification of the algorithm that avoids this limitation on the precision of q.

A. Overview

The algorithm for the q-th root calculation ($q \neq 0$) is derived as follows

$$X^{1/q} = 2^{\log_2(X^{1/q})} = 2^{(1/q)\log_2(X)} \quad (1)$$

Considering a floating–point operand $X = M_x \times 2^{E_x}$, M_x being the n-bit significand and E_x the n_{Ex}-bit signed exponent,

$$\begin{aligned} X^{1/q} &= 2^{(1/q)\log_2(M_x \times 2^{E_x})} \\ &= 2^{(1/q)\log_2(M_x)} \times 2^{E_x/q} \end{aligned} \quad (2)$$

Equation (2) could be taken as the floating–point result for the q-th root calculation, where $2^{(1/q)\log_2(M_x)}$ is the significand and E_x/q the exponent; however, the main problem for this interpretation is that the exponent is not

an integer. In [9], it has been suggested that this situation can be handled by decomposing the integer division E_x/q into integer and fractional part, in such a way that $E_x/q = \lfloor E_x/q \rfloor + (1/q) \times E_x \% q$, where $\%$ is the remainder of the division. This way, equation (2) could be rewritten as follows

$$X^{1/q} = 2^{(1/q)\log_2(M_x)} \times 2^{(1/q)\times E_x \% q} \times 2^{\lfloor E_x/q \rfloor} \quad (3)$$

where $2^{(1/q)\log_2(M_x)} \times 2^{(1/q)\times E_x \% q}$ and $\lfloor E_x/q \rfloor$ are the significand and the exponent, respectively. However, equation (3) is hard to implement due to integer division and remainder operations.

Therefore, another transformation to equation (2) must be used. To avoid the integer division E_x/q equation (2) can be rewritten as

$$\begin{aligned} X^{1/q} &= 2^{(1/q)\log_2(M_x \times 2^{E_x})} \\ &= 2^{(1/q)\times S} \end{aligned}$$

where $S = E_x + \log_2(M_x)$ is the concatenation of the digits of E_x (integer value) and $\log_2(M_x) \in [0, 1)$.

Therefore,

$$X^{1/q} = 2^{S/q} \quad (4)$$

According to equation (4) the q-th root can be calculated as a sequence of operations: logarithm of the significand M_x ($log_2(M_x) \in [0, 1)$), addition of E_x and $log_2(M_x)$ (concatenation of binary strings), division by q and exponential of the result of the division. For an efficient implementation, the computation of 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.

A problem of the algorithm above is the range of the exponential function $2^{S/q}$. Digit–recurrence exponential algorithms require the argument to be in the interval $(-1, 1)$, while S/q is out of the range. To extend the range of convergence and guarantee the convergence of the algorithm, the integer and fractional parts of the argument of the exponential must be extracted serially and equation (4) must be rewritten.

If $T = S/q$ is computed and its integer $int(T)$ and fractional $frac(T)$ parts are extracted, 2^T becomes

$$2^T = 2^{int(T)} \times 2^{frac(T)} \quad (5)$$

so that the argument of the exponential $2^{frac(T)}$ is now in $(-1, 1)$.

Then, replacing in equation (4), the q-th root is obtained as follows

$$X^{1/q} = M_z \times 2^{E_z} \quad (6)$$

with

$$M_z = 2^{frac(T)} \quad , \quad E_z = int(T) \quad (7)$$

where M_z and E_z are the significand and exponent of $X^{1/q}$. This results in a bounded argument for the exponential.

We have first implemented the q-th root algorithm for low precision values of q and a generic radix $r = 2^b$. An example is shown in Fig. 1 for a single-precision operand X and radix $r = 128$. The sequence of operations is as follows:

1) Evaluation of $D = (-1)^{s_q} \times abs(1/q)$ using a lookup table of n_q inputs (low precision) and n_d outputs (1 integer bit and $n_d - 1$ fractional bits), with $abs(1/q) \in (0, 1]$ and $E_q > 0$. Operand D is obtained in a non-redundant binary form (see Section III).
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 (see Section III).
3) Multiplication $T = D \times S$ (see Section III). Operand

$$S = \sum_{i=-\lceil n_{Ex}/b \rceil}^{\lceil n_l/b \rceil} S_i \; r^{-i} = E_x + L$$

is obtained serially by concatenating the digits of E_x and L, with $E_x \in [-2^{n_{Ex}-1}, 2^{n_{Ex}-1} - 1]$ and L expressed in a signed-digit radix-r form. We evaluate this multiplication using a LRCF (left-to-right carry-free) multiplier [2]. The product of this fixed point multiplication has at most n_{Ex} significant integer bits. The maximum number of accurate fractional bits required is calculated in Section II-B.
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 integer part corresponds to the $\gamma = \lceil n_{Ex}/b \rceil$ leading digits of T.
5) Online high-radix exponential $2^{frac(T)} \in (0.5, 2)$ with argument $frac(T) \in (-1, 1)$, precision of n_e bits, and online delay $\delta = 2$ (see Section III). The redundant result is normalized and rounded to n bits using an on-the-fly rounding unit [3].

The latency of the algorithm for $r = 2^b \geq 8$ is given by:

$$N_{q-root} = 1 + \gamma + (\delta + 1) + N_e$$

where $\delta = 2$ and $N_e = \lceil n_e/b \rceil$ are respectively the online delay and the latency of the exponential $2^{frac(T)}$, and $\gamma = \lceil n_{Ex}/b \rceil$ is the number of radix-r integer digits of T. We have performed an error analysis to obtain an estimation of the precisions and latencies for the intermediate operations.

B. Error Analysis

If we evaluate $X^{1/q}$ using the previous sequence of operations, we get an approximation $X_\alpha^{1/q}$ (we use a subindex α to indicate an approximated value) with the following contributions to the error, represented by the different ε's:

1) Reciprocal $D = 1/q$. The output of the reciprocal unit is $D + \varepsilon_{rec}$
2) Logarithm $L = log_2(M_x)$: The output of the module is $L + \varepsilon_{log}$

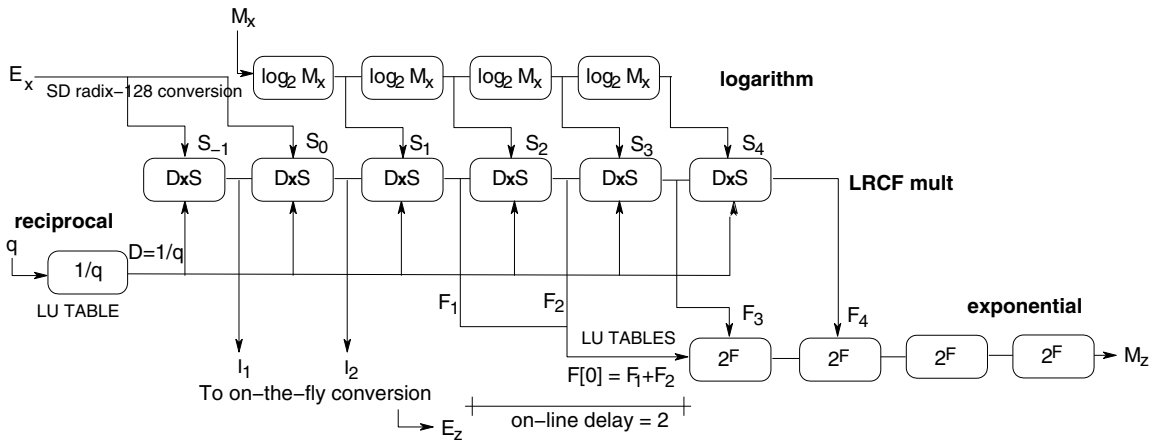

Figure 1. Sequence of operations of the algorithm.

3) Multiplication $T = D \times S$ with $S = E_x + L$. The output of the multiplier is given by

$$T_\alpha = T + D \times \varepsilon_{log} + S \times \varepsilon_{rec} + \varepsilon_{log} \times \varepsilon_{rec} + \varepsilon_{mul}$$

In order to simplify the previous expression we use

$$\varepsilon_f = D \times \varepsilon_{log} + S \times \varepsilon_{rec} + \varepsilon_{log} \times \varepsilon_{rec} + \varepsilon_{mul} \quad (8)$$

so

$$T_\alpha = T + \varepsilon_f$$

4) Extraction of the integer and fractional parts of $T = D \times S$. The integer part is of the form $int(T_\alpha) = int(T + \varepsilon_f)$. The fractional part is given by

$$frac(T_\alpha) = T + \varepsilon_f - int(T + \varepsilon_f)$$

5) Operation $2^{frac(T)}$. The output of the exponential computation is $2^{frac(T_\alpha)} + \varepsilon_{exp}$.

So the approximation of $X^{1/q}$ is as follows

$$\begin{aligned} X_\alpha^{1/q} &= (2^{frac(T_\alpha)} + \varepsilon_{exp}) \times 2^{int(T_\alpha)} \\ &= (2^{frac(T+\varepsilon_f)} + \varepsilon_{exp}) \times 2^{int(T+\varepsilon_f)} \\ &= 2^{T+\varepsilon_f} + \varepsilon_{exp} \times 2^{int(T+\varepsilon_f)} \\ &= X^{1/q} \times 2^{\varepsilon_f} + \varepsilon_{exp} \times 2^{int(T+\varepsilon_f)} \end{aligned} \quad (9)$$

with $X^{1/q} = 2^T$.

Since we want to provide faithfully rounded q-th roots, the rounded result $RN(X_\alpha^{1/q})$ must be within 1 ulp of the exact result $X^{1/q}$. Assuming rounding to the nearest even, the error bound for $|X_\alpha^{1/q} - RN(X_\alpha^{1/q})|$ is 0.5 ulp, and the condition for faithful rounding is given by $|X^{1/q} - X_\alpha^{1/q}| \leq 0.5\ ulp$.

Since $frac(T) \in (-1, 1)$ is expressed in signed-digit, then $2^{frac(T)} \in (0.5, 2)$ and we are considering n precision bits for the final normalized result, this implies that $1\ ulp = 2^{-n-\sigma} \times 2^{int(T)}$, where $\sigma = 1$ if $frac(T) < 0$ and $\sigma = 0$ if $frac(T) \geq 0$. Then, it must be verified that

$$|X^{1/q} - X_\alpha^{1/q}| \leq 2^{-(n+\sigma)} \times 2^{int(T)}$$

Replacing $X_\alpha^{1/q}$ by expression (9), the previous condition is transformed into

$$|X^{1/q} \times (1 - 2^{\varepsilon_f}) - \varepsilon_{exp} \times 2^{int(T+\varepsilon_f)}| \leq 2^{-(n+\sigma)} \times 2^{int(T)}$$

Using $X^{1/q} = 2^{frac(T)} \times 2^{int(T)}$ and

$$int(T + \varepsilon_f) = int(T) + int(frac(T) + \varepsilon_f)$$

and taking out the common factor $2^{int(T)}$, the previous condition can be expressed as

$$|2^{frac(T)} \times (1 - 2^{\varepsilon_f}) - \varepsilon_{exp} \times 2^{int(frac(T)+\varepsilon_f)}| \leq 2^{-(n+\sigma)}$$

Next, we compute an upper bound for the left term of this inequality. Considering $\varepsilon_f < 2^{-n}$ we use the approximation $2^{\varepsilon_f} = e^{\ln(2)\varepsilon_f} \approx 1 + \ln(2)\varepsilon_f + O(2^{-2n}) \leq 1 + \varepsilon_f$. Moreover, we can use the same upper bound $2^{1-\sigma}$ for both terms $2^{frac(T)}$ and $2^{int(frac(T)+\varepsilon_f)}$. We introduce these bounds in the previous expression obtaining

$$|\varepsilon_f + \varepsilon_{exp}| \leq 2^{-(n+1)}$$

The critical parameter to minimize first is ε_{exp}, since it is directly related to the latency of the algorithm. The minimum possible value for the upper bound for the exponential error is:

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

Then we have

$$|\varepsilon_f| \leq 2^{-(n+2)}$$

To obtain the highest contribution to the error ε_f given by expression (8) we replace $D = 1/q \leq 1$ by 1 and $S = E_x + \log_2(M_x) \leq 2^{n_{Ex}}$ by $2^{n_{Ex}}$. Then,

$$|\varepsilon_{log} + 2^{n_{Ex}} \times \varepsilon_{rec} + \varepsilon_{log} \times \varepsilon_{rec} + \varepsilon_{mul}| \leq 2^{-(n+2)}$$

To simplify we put the same upper bound ε for the errors of the logarithm and multiplication, that is, $\varepsilon_{log} \leq \varepsilon$, $\varepsilon_{mul} \leq$

Table II
EXAMPLE OF PARAMETERS FOR q-TH ROOT COMPUTATION ($r = 128$, $\delta = 2$, $\gamma = \lceil n_{Ex}/b \rceil$)

Target Precision	Precision (bits)				Latency (cycles)				
	n_d	n_l	n_m	n_e	N_d	N_l	N_m	N_e	N_{q-root}
(n, n_{Ex})	n_{Ex}+n+4	n+4	n_{Ex}+n+4	n+2	1	$\lceil (n+4)/b \rceil$	$\gamma+\lceil (n+4)/b \rceil$	$\lceil (n+2)/b \rceil$	$2+\delta+\gamma+\lceil (n+2)/b \rceil$
SP (24,8)	36	28	36	26	1	4	2+4	4	10
DP (53,11)	68	57	68	55	1	9	2+9	8	14

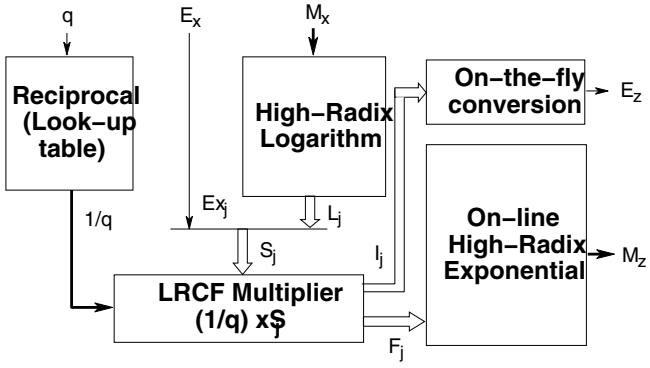

Figure 2. Block diagram of the architecture.

ε, while for the error of the reciprocal we consider $\varepsilon_{rec} \leq 2^{-n_{Ex}} \times \varepsilon$. Then we get

$$|(3 + 2^{-n_{Ex}}) \times \varepsilon| \leq 2^{-(n+2)}$$

so that an upper bound for the error of the logarithm and multiplication is

$$|\varepsilon| \leq 2^{-(n+4)}$$

For the reciprocal we get

$$|\varepsilon_{rec}| \leq 2^{-(n+n_{Ex}+4)}$$

The required precision (integer+fractional bits) and minimum latency values for each intermediate operation (n_d and N_d for exact rounded reciprocal, n_l and N_l for logarithm, n_m and N_m for multiplication, and n_e and N_e for exponential) are shown in Table II parametrized as a function of n and n_{Ex}.

We also show the latency for the q-th root computation and the corresponding values for single (SP) and double (DP) precision with $r = 128$.

III. IMPLEMENTATION

We propose a sequential architecture for the implementation of the algorithm described in Section II. Fig. 2 shows the general block diagram of the architecture. Single thick lines represent long-word operands (around n bits), single thin lines represent short-word operands (around b or n_{Ex} bits), and double lines represent redundant signed radix-r digits in a borrow-save format (or signed-digit radix 2). The high-radix logarithm, LRCF multiplication, and on-line high-radix exponential units are similar to those implemented in [9]. A detailed description of an on-the-fly conversion unit can be found in [3].

Table III
SIZE (IN KBITS) OF LOOK-UP TABLES FOR THE RECIPROCAL

(n, n_{Ex})	n_q							
	5	6	7	8	9	10	11	12
SP (24,8)	1.15	2.30	4.6	9.2	18.4	37	74	148
DP (53,11)	2.18	4.35	8.7	17.4	34.8	70	139	278

To allow faster execution of iterations in these units we opted for representing all variables in a redundant borrow-save representation. An advantage of borrow-save over carry-save representation is an easier conversion of signed radix-r digits. Moreover, a borrow-save adder can be implemented as a carry-save adder with some inverted inputs and outputs. Next, we outline the main computations involved.

Reciprocal. If q is a low-precision operand (precision of $abs(q)$ $n_q \leq 12$ bits), we can use a look-up table to compute efficiently its reciprocal $abs(1/q) \in (0, 1]$. As we have detailed in Section II-B, we need $n+n_{Ex}+4$ precision bits to represent the correctly rounded reciprocal $abs(1/q)$. Therefore, the size of the look-up table is $2^{n_q} \times (n+n_{Ex}+4)$ bits. We assume a 1-cycle latency for the look-up table. In Table III we show the size of the corresponding look-up table for single and double precision results and several values of n_q.

Logarithm. For the computation of $\log_2(M_x)$ we use an optimized high-radix digit-recurrence algorithm similar to that implemented in [9]. This algorithm is based on the identity

$$\log_2(M_x) = \log_2(M_x \prod f_j) - \sum \log_2(f_j)$$

See [9] for more details.

Multiplication. The left-to-right carry-free (LRCF) multiplication [2], produces the product digits from a redundant set in a most-significant-digit-first (MSDF) manner. In our case, the digits produced by the LRCF multiplier are used without conversion. See [2], [9] for more details.

Exponential. The online high-radix algorithm for the computation of 2^F is similar to that implemented in [9]. This algorithm is based on the identity

$$2^F = (\prod h_j) 2^{F - \sum \log_2(h_j)}$$

See [9] for more details.

IV. Implementation for higher precision q

The main limitation of the algorithm proposed in Section II is the range of q. If the precision n_q of $abs(q)$ is significantly high ($n_q > 12$), a direct extraction of $abs(1/q)$ from a look-up table is not efficient. In this case, the algorithm of Section II needs to be modified.

For the computation of the reciprocal we propose a high-radix iterative algorithm with multiplicative decomposition [4]. The divisor is required to be in the convergence range of the algorithm, which in our case is $[0.5, 1)$. Therefore $abs(q) \in [1, 2^{n_q} - 1]$ is normalized into $M_q \in [0.5, 1)$, such that $abs(q) = M_q 2^{E_q}$, with $E_q > 0$. Thus, the computation of $1/q$ is replaced by the evaluation of $(-1)^{s_q} \times (1/M_q) \times 2^{-E_q}$.

An example of the operation flow of the modified q-th root algorithm for single precision and $r = 128$ is shown in Fig. 3. The sequence of operations of the algorithm for $n_q > 12$ is as follows:

1) Evaluation of the number of leading zeros $lz_q \in [0, n_q - 1]$ of q using a LZD. The exponent E_q is given by $E_q = n_q - lz_q$.
2) Normalization of $abs(q)$ into the range $[0.5, 1)$. A n_q-bit barrel shifter is used to obtain the normalized divisor $M_q \in [0.5, 1)$ by shifting q an amount of lz_q bits to the left.
3) Evaluation of $D = (-1)^{s_q} \times (1/M_q) \in (-2, 2)$ using the high-radix iterative algorithm with multiplicative decomposition (see [15]). The digits D_j of the reciprocal are in a signed-digit radix-r redundant form.
4) Evaluation of the logarithm $L = \log_2(M_x) \in [0, 1)$ to a precision of n_l bits using a high-radix algorithm (see [15]).
5) Multiplication of $S^* = S \times 2^{-E_q}$, with $S = E_x + \log_2(M_x)$. Since S is obtained digit by digit in a signed-digit radix-r form, this multiplication is implemented as a composition of two different right shifts: a first shift of S by $\lfloor E_q/b \rfloor$ radix-r digits and a second shift of each digit S_i of S by $E_q \% b$ bits. A signed-digit shifter that implements this operation serially is described in [15].
6) Multiplication $T = D \times S^*$ (see [15]). Since the digits of the reciprocal D (D_i) and the operand S^* ($S^*{}_j$) are obtained serially, we implement this multiplication using an online multiplier [14] with online delay $\delta = 2$.
7) 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.
8) On-line high-radix exponential $2^{frac(T)} \in (0.5, 2)$ with argument $frac(T) \in (-1, 1)$, precision of n_e bits, and online delay $\delta = 2$ (see [15]). The redundant result is normalized and rounded to n bits using an on-the-fly rounding unit [3].

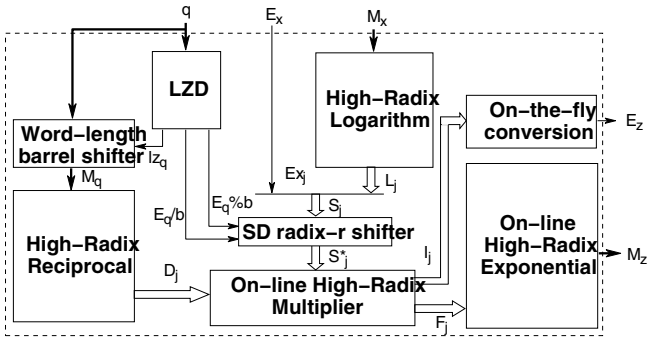

Figure 4. Block diagram of the modified architecture.

To obtain the required precisions for the different intermediate operations, we have performed an error analysis similar to the analysis in Section II-B. The required precisions for each operation (reciprocal, logarithm, signed-digit shifting, multiplication and exponential) are shown in Table IV. These parameters determine the latencies of the intermediate operations for radix $r = 2^b$. Table IV also shows the required precision and latency of each operation for single and double precision and $r = 128$. Besides, we consider a 1-cycle latency for the LZD and coding of shifting amount E_q/b and $E_q \% b$, one cycle for the normalization of q and $N_d = \lceil n_d/b \rceil$ iterations for the high-radix fixed-point reciprocal unit.

The precision of q only has direct impact in the latency and area of the LZD and the barrel shifter used for normalization, but not in the latency of the high-radix iterative reciprocal unit.

The latency of the algorithm for $r = 2^b \geq 8$ is given by:

$$N_{q-root} = 1 + \gamma + 2 \times (\delta + 1) + N_e$$

where $\delta = 2$ and $N_e = \lceil n_e/b \rceil$ are respectively the online delay and the latency of the exponential $2^{frac(T)}$, and $\gamma = \lceil n_{Ex}/b \rceil$ is the maximum number of radix-r integer digits of the multiplication product.

A sequential architecture for the implementation of the modified algorithm is shown in Fig. 4. The architecture of each unit is detailed in [15].

V. Evaluation and Comparison

In this Section, we present estimates of the execution time and hardware cost for the proposed low and high precision q architectures described in Sections III and IV. First, we describe the evaluation model used to obtain the area and delay estimates. Next, we particularize for single ($n = 24$, $n_{Ex} = 8$) and double precision ($n = 53$, $n_{Ex} = 11$) formats with radix values $r = 2^b$ ranging from $r = 8$ to $r = 1024$. Finally, we present a comparison with other representative implementations in Section V-A.

We use an area and delay evaluation model based on a simplification of the logical effort method [11] that allows

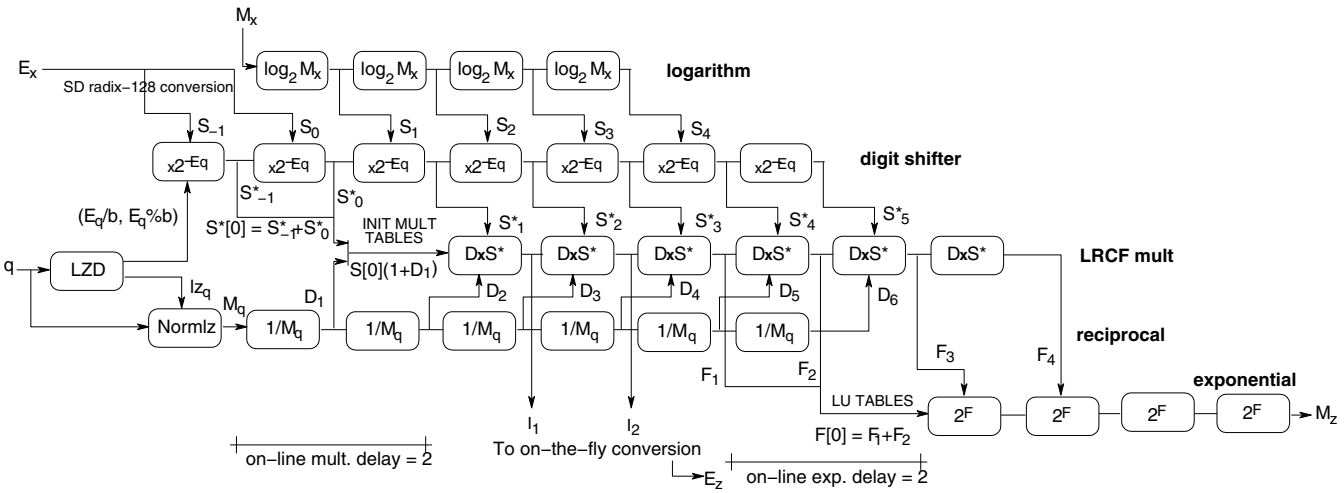

Figure 3. Sequence of operations of the modified algorithm.

Table IV
EXAMPLE OF PARAMETERS FOR q-TH ROOT COMPUTATION ($r = 128$, $\delta = 2$, $\gamma = \lceil n_{Ex}/b \rceil$)

Target Precision	**Precision of intermediate operations (bits)**					**q–th root precision**
	n_d	n_l	n_s	n_m	n_e	n_{q-root} (bits)
$(n,\ n_{Ex},\ n_q)$	n_{Ex}+n+4	n+4	n_{Ex}+n+5	n_{Ex}+n+4	n+2	n
SP (24,8,32)	36	28	37	36	26	24
DP (53,11,64)	68	57	69	68	55	53
Target Precision	**Latency of intermediate operations (cycles)**					**q–th root latency**
	N_d	N_l	N_s	N_m	N_e	N_{q-root} (cycles)
$(n,\ n_{Ex},\ n_q)$	$\lceil n_d/b \rceil$	$\lceil n_l/b \rceil$	γ+$\lceil (n$+5$)/b \rceil$	γ+$\lceil (n$+4$)/b \rceil$	$\lceil n_e/b \rceil$	3+2δ+γ+$\lceil n_e/b \rceil$
SP (24,8,32)	6	4	2+5	2+4	4	13
DP (53,11,64)	10	9	2+9	2+9	8	17

for faster hand calculations (see [15] for further details). The delays are expressed in FO4 units (delay of an 1x inverter with a fanout of 4 inverters), and the area in number of equivalent minimum size NAND2 gates. We do not expect this rough model to give very precise area-delay figures, but it provides good first-order area and delay estimations to be used in technology-independent comparisons.

Table V shows estimates of the latency (in number of cycles), cycle time and execution time (in FO4 units) of the q-th root computation, area of the high radix-r logarithm and on–line exponential units (in NAND2 units), and total area of the proposed low precision q architecture with $n_q = 8$, for single precision computations. The estimates for double precision are not presented here because the general trend is similar regardless of the precision adopted (see [15]). The implemented radix values go from $r = 8$ to $r = 1024$. To give a better understanding of the relative delay, cycle time and area of the proposed architecture, the execution time, cycle time and area of a single-precision floating–point FMA, obtained using the logical effort model, is provided. Thus, the execution time and the cycle time of the FMA unit are 70 and 35 FO4 units, respectively, assuming a latency of 2 cycles, and the area is 8780 NAND2 gates.

The latency of a q–root computation for the low precision q implementation was calculated in accordance with the formula $2 + \gamma + \delta + N_e$, with $\gamma = 2$ and $\delta = 2$. The cycle time corresponds to the critical path delay of the logarithm unit, and is the sum of the delays of the round unit, a multiplexer and the multiply-add unit. The main contribution to the total area of the q-root unit comes from both the high-radix logarithm and on–line exponential units, and this is significantly high for radix values $r > 128$. In addition, we observe that, very little advantage in execution time is obtained from using very high radix values (over $r = 128$). On the other hand, the estimated area look–up table for the reciprocal with $n_q = 8$ is only 2360 NAND2 gates for single precision. However, as we show later, for higher precision values of q, such as $n_q > 10$, the contribution of this look–up table to the total area is very significant, and the use of a look–up table to compute the reciprocal may be not justified.

Table VI shows the corresponding estimates for the high–precision q architecture, with $n_q = 32$ for single precision computations (see [15] for area and latency estimates for double precision and $n_q = 64$). In addition to the previous estimates, we show the area of the reciprocal unit for the different radix values. We include in this area estimation the fixed point high-radix iterative unit, the leading zero detector and the related shifters. Though we only present estimates

Table V
AREA AND DELAY OF THE LOW PRECISION q ARCHITECTURE WITH $n_q = 8$ AND SINGLE PRECISION ($n = 24$, $n_{Ex} = 8$). SP FMA UNIT: EXC. TIME = 70 FO4 UNITS, CYCLE TIME = 35 FO4 UNITS, LATENCY = 2, AREA = 8780 NAND2 GATES

Radix	Latency	Cycle Time (♯FO4) (♯FO4)	Exec. time (♯FO4)	LOG Area (NAND2)	Exp. Area (NAND2)	Total Area (NAND2)
8	16	30	480	4500	4200	13650
16	13	32	416	5700	5400	16450
32	12	32	384	6600	6200	18200
64	11	34	374	9000	8500	23300
128	10	34	340	11400	10800	28250
256	9	36	324	18700	17600	42800
512	8	36	288	32600	30700	69900
1024	8	37	296	60500	57000	124650

Table VI
AREA AND DELAY OF THE HIGH PRECISION q ARCHITECTURE WITH $n_q = 32$ AND SINGLE PRECISION ($n = 24$, $n_{Ex} = 8$). SP FMA UNIT: EXC. TIME = 70 FO4 UNITS, CYCLE TIME = 35 FO4 UNITS, LATENCY = 2, AREA = 8780 NAND2 GATES

Radix	Latency	Cycle Time (♯FO4) (♯FO4)	Exec. time (♯FO4)	Rec. Area (NAND2)	LOG Area (NAND2)	Exp. Area (NAND2)	Total Area (NAND2)
8	19	30	570	6200	4500	4200	21100
16	16	32	512	7050	5700	5400	25200
32	15	32	480	7500	6600	6200	27700
64	14	34	476	8700	9000	8500	34400
128	13	34	442	9700	11400	10800	40500
256	12	36	432	12000	18700	17600	57800
512	11	36	396	15100	32600	30700	88400
1024	11	37	407	20650	60500	57000	149100

for a given precision ($n_q = 32$), moderate variations in the precision n_q have a negligible impact on the total area or execution time.

We want to determine a threshold value for n_q such that the high precision q architecture presents a hardware cost advantage over the low precision q architecture. We present in Table VII area estimations of the low precision q architecture, for single precision computations, and for different values of n_q from 5 up to 12. We choose a radix $r = 128$ since it seems to be the most advantageous in terms of the product *execution time* $\times$*area*. We also present the area estimations for the high-precision q architecture for a radix value $r = 128$ and $n_q = 32$. We observe that the area of the low precision q architecture with $n_q = 11$ is slightly higher than the area of the higher precision q architecture, although the latency is 3 cycles lower. For $n_q < 11$, the low precision q architecture is more advantageous in terms of both area and latency. On the other hand, the huge area penalty of the lookup table makes the high-precision q architecture more attractive for $n_q > 11$.

A. Comparison

The comparison of our q-th root computation method with previous alternatives is not easy. As far as we know, no other previously proposed algorithm and architecture, but a naive implementation of $X^{1/q} = 2^{(1/q)\log_2(X)}$ and the extension of the powering architecture in [9], allows the computation of the q-th root for any value of q; that is, the other architectures in the literature are derived for a given value of q and changing q implies making a different implementation. On the contrary, the proposed architecture allows the computation of any q–th root without any additional modification. It has to be pointed out that the complexity of the extended algorithm in [9] which implements Equation (3), makes it very hard and inefficient to implement more than a small set of q–th roots.

Basically, there are two types of algorithms for the computation of the q-th root. Algorithms based on table-driven polynomial approximations [1], [8], [12] and digit–recurrence algorithms [6]. Among the former, in [12] a method for generating X^p for a given p is proposed, applicable to values of $p = \pm 2^k$ or $p = \pm 2^{k_1} \pm 2^{k_2}$, where k_1 is an integer and k_2 is a non-negative integer. This includes a limited number of roots, such as square root, fourth root, eighth root, etc. The powering function is computed by a piecewise linear approximation based on modified first–order Taylor expansion. The first–order Taylor expansion is rewritten as $X^p = C \times X'$, where $X' = X_1 + 2^{-m-1} + p \times (X_2 - 2^{-m-1})$, and X_1, X_2 are the upper m-bit part and the lower part of X, respectively. C can be read through a table look-up addressed by X_1 and, for special p's, X' is easily obtained by modifying X. Only one multiplication is required to evaluate the modified Taylor expansion.

A second–order minimax approximation is presented in [8], which allows the computation of X^p for any given p. This includes every q-th root. X^p is approximated as $C_2 \times X_2^2 + C_1 \times X_2 + C_0$. The three coefficients C_2, C_1 and C_0 are

Table VII
DETERMINING THE BEST UNIT IN TERMS OF COST FOR $r = 128$ AS A FUNCTION OF n_q

q-th root unit with reciprocal by lookup table								
	Single precision ($n = 24$, $n_{Ex} = 8$, 10 cycles latency) n_q							
	5	6	7	8	9	10	11	12
Area $1/q$ (♯NAND2)	335	675	1180	2360	4725	9450	18900	32400
Total Area (♯NAND2)	26200	26600	27100	28250	30600	35350	44800	58300

q-th root unit with reciprocal by high-radix digit-recurrence			
	Latency (cycles)	Area $1/q$ (NAND2)	Total Area (NAND2)
SP ($n = 24$, $n_{Ex} = 8$, $n_q = 32$)	13	10850	40500

stored in look–up tables and selected by X_1. The size of the tables is optimized by carefully minimizing the coefficients wordlength for the required precision. The evaluation of the powering requires, besides the look–up tables, a squaring unit and a fused accumulation tree.

Another second–order interpolation for the evaluation of elementary functions, including q-th roots, is presented in [1]. The table sizes are reduced by storing the function values and one coefficient for each interpolation subinterval, instead of storing all the three coefficients as in the proposal above. The two remaining coefficients are computed from the function values. This way, the memory requirements are reduced by one third. Additionally, some multipliers and adders are needed to complete the powering computation.

Note that the three algorithms and architectures above for the computation of $X^{1/q}$ are targeted for a given q. To adapt the architecture to other different q, when possible, requires changing the look–up tables.

On the other hand, a general digit–recurrence algorithm for the computation of the q-th root has been presented in [6]. The result is a general algorithm that must be particularized for each different q, and the larger q the larger the complexity. This general algorithm has been used to implement a cube root unit [10].

In order to evaluate our architecture, we compare the algorithms based on a table-driven polynomial approximation and the digit–recurrence algorithm outlined above. However, it has to be kept in perspective that, unlike our algorithm, all these algorithms have to be particularized for a given q when implementing the q-th root unit. Moreover, we include in the comparison the naive implementation of $X^{1/q} = e^{(1/q)ln(X)}$ and the architecture for the computation of X^p and its extension to $X^{1/q}$, with p and q integers, presented in [9].

Table VIII shows the latency, delay and area estimate of every algorithm and implementation. We have considered a single–precision floating–point representation for X and low–precision q; in particular, we consider $n_q = 8$ for low precision q and $n_q = 32$ for our architecture with higher precision q. For the architectures based on composite logarithm-multiplication-exponential algorithms, we have used a radix $r = 128$.

Note that, although the table-driven polynomial approximation are implemented for a given root, the latency, delay and area are roughly the same for any root. However, the features of the digit–recurrence architecture are closely related to the value of q; so, we show the features for the computation of the radix 2 cube root described in [10].

As expected, the latency, total delay and area of the table–driven polynomial approximation architectures is significantly smaller than in the architecture we have proposed in this paper, because those architectures can compute only a given root. Of course, those architectures could be extended to compute more than just one given root. This means that tables to store the coefficients or the function values need to be replicated to include one set of tables for each root. This affects the total area and the cycle time.

Similarly, the digit–recurrence architecture is tailored to a given root, as well. The computation of a different root implies the development of completely different architecture. The larger the q the larger the complexity.

Finally, the features of our architecture and the architecture in [9] are quite similar. Both architectures are based on similar optimizations of the naive implementation of $X^{1/q} = 2^{(1/q)\log_2(X)}$; however, the extension of the architecture in [9] to the computation of a q-th root, is quite inefficient. This is mainly due to the huge table required to obtain $2^{(E_x\%q)/q}$ once the remainder of the integer division $E_x\%q$ has been evaluated, even for low precision q.

VI. CONCLUSION

An algorithm for q–th root extraction has been presented, based on a high–radix composite algorithm. The algorithm consists of computing $X^{1/q}$ as $2^{(1/q)(E_x+log_2(M_x)}$ through a sequence of parallel and /or overlapped operations: reciprocal, logarithm, multiplication and exponential. A detailed error analysis has been carried out to determine the intermediate wordlengths. The algorithm is based on a previous algorithm for the computation of the powering function X^p, with p any integer, which was extended for the computation of q–th roots. However, the extended algorithm seems hard to implement since it is necessary to compute an integer division and a remainder operation. Our algorithm avoids these two operations, resulting in a much simpler algorithm.

Two architectures have been proposed. First, an architecture for low precision q values, less than 12 bits, where the

Table VIII
ARCHITECTURE FEATURES COMPARISON FOR SINGLE–PRECISION FLOATING–POINT REPRESENTATION AND LOW PRECISION q

Architecture	Latency (cycles)	cycle time (FO4)	Delay (FO4)	Area (NAND2)
Composite LOG-MUL-EXP Algorithms ($r = 128$)				
Our architecture ($n_q = 8$)	10	34	340	28250
Our architecture ($n_q = 32$)	13	34	442	40500
Naive ($n_q = 8$)	15	34	510	33015
X^p with p integer [9]	9	34	306	29827
$X^{1/q}$ ($n_q = 8$) [9]	9	34	306	> 500000
Linear approx. [12]	1	51	51	26122
2nd–order interp. [8]	3	18.7	56.1	10170
2nd–order interp. [1]	2	54.4	108.8	10612
Digit–recurr. ($q = 3$, $r = 2$) [10]	52	119	6188	9035

reciprocal $1/q$ is obtained directly from a look–up table; after that, an alternative architecture for higher precision values of q, where a high–radix iterative algorithm has been used for the computation of the reciprocal. Both architectures have been evaluated and estimates of the execution time, the latency and the area have been obtained, based on an approximated model for the delay and the area of the main building blocks, for single precision floating–point representation and several radices. The analysis of the tradeoffs between area and speed allows us to determine the better radix for every implementation: radix $r = 128$ might be suitable for high speed implementations. Larger radices result in similar execution times with much larger area requirements.

The comparison with other previous algorithms is not easy. As far as we know, no other previously proposed methods, except the extension of the architecture for the computation of X^p, allow the computation of the q–th root for any value of q. Even so, we have discussed the area and time figures for several non–general implementations for q–th root calculation and the powering architecture to determine the suitability of our implementation. The conclusion is that the execution times and hardware requirements are better than those of the powering function calculation and, although obviously are worse than those of the non–general q–th root extraction architectures, the flexibility of our implementation makes it an interesting alternative.

Work is in progress to integrate the proposed q–th root computation with the previous powering function algorithm into a combined unit.

REFERENCES

[1] J. Cao, B. W. Y. Wei and J. Cheng, *High-performance architectures for elementary function generation*, Proc. 15th IEEE Symp. on Computer Arithmetic, pp. 136–144, Jun. 2001.

[2] M. D. Ercegovac and T. Lang, *Fast Multiplication Without Carry-Propagate Addition*, IEEE Trans. on Computers, vol.39, no.11, pp. 1385–1390, Nov. 1990.

[3] M.D. Ercegovac and T. Lang, *On-the-Fly Rounding*, IEEE Trans. on Computers, vol.41, no.12, pp.1497–1503, Dec. 1992.

[4] M.D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2004.

[5] M.D. Ercegovac, *Digit-by-Digit Methods for Computing Certain Functions*, 41st Asilomar Conference on Signals, Systems and Computers, pp. 338–342, Nov. 2007.

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

[7] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, Birkhäuser, 1997.

[8] J.-A. Piñeiro, J.D. Bruguera and J.-M. Muller, *Faithful Powering Computation Using Table Lookup and Fused Accumulation Tree*, Proc. 15th IEEE Symp. on Computer Arithmetic, pp. 40–47, Jun. 2001.

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

[10] A. Piñeiro, J.D. Bruguera, F. Lamberti, P. Montuschi, *A Radix-2 Digit-by-Digit Architecture for Cube Root*, IEEE Trans. on Computers, vol. 57, no. 4, pp. 562–566, Apr. 2008.

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

[12] N. Takagi, *Powering by a Table Look-Up and a Multiplication with Operand Modification*, IEEE Trans. on Computers, vol. 47, no. 11, pp. 1216–1222, Nov. 1998.

[13] N. Takagi, *A Digit-Recurrence Algorithm for Cube Rooting*, IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, vol. E84-A, no. 5, pp. 1309–1314, May 2001.

[14] K.S. Trivedi and M.D. Ercegovac, *On-Line Algorithms for Division and Multiplication*, IEEE Trans. on Computers, vol. C-26, no. 7, pp. 681–687, Jul. 1977.

[15] A. Vazquez, J.D. Bruguera, *Composite Iterative Algorithm and Architecture for q-th Root Calculation*, INRIA Research Report RR-7564, Mar. 2011. Available at http://hal.inria.fr/inria-00575573/PDF/RR-7564.pdf.