Composite Iterative Algorithm and Architecture for q-th Root Calculation

Alvaro Vazquez, Javier Bruguera

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Composite Iterative Algorithm and Architecture for q-th Root Calculation

Álvaro Vázquez — Javier D. Bruguera

N° 7564

March 2011

ISSN 0249-6399 ISRN INRIA/RR--7564--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Composite Iterative Algorithm and Architecture for q-th Root Calculation

Álvaro Vázquez *, Javier D. Bruguera †

Thème : Algorithms, Certification, and Cryptography
Équipe-Projet Arénaire

Rapport de recherche n° 7564 — March 2011 — 30 pages

Abstract: An algorithm for the q-th root extraction, being q 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 and double 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.

Key-words: Integer rooting, high-radix digit-by-digit algorithms, on-line algorithms, elementary function evaluation

* INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon, France, Alvaro.Vazquez-Alvarez@inrialpes.fr

† Universidade de Santiago de Compostela, Departamento de Electrónica e Computación, Spain, jd.bruguera@usc.es. Work supported in part by Ministry of Education and Science of Spain 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)

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Algorithme itératif composé et architecture pour le calcul des racines q-ièmes

Résumé : Dans cet article, nous présentons un algorithme matériel pour l'extraction de la racine q-ième d'un nombre X, où q est un entier naturel non nul. Cet algorithme est basé sur une implantation optimisée de la fonction $X^{1/q} = 2^{(1/q)\log_2(X)}$ par une séquence d'opérations parallèles et/ou superposées: (1) réciproque, (2) logarithme chiffre par chiffre, (3) multiplication de gauche-à-droite sans propagation de retenue et (4) exponentielle en ligne. Une analyse détaillée des erreurs et deux architectures sont proposées, pour q de basse précision et pour q de précision plus haute. Le temps d'exécution et les composants matériels à utiliser sont estimés pour des calculs en virgule flottante simple et double précision et pour plusieurs bases. Cette étude aide à déterminer quelles bases mènent aux implantations les plus efficaces. Les architectures proposées améliorent les caractéristiques d'architectures précédentes destinées à l'extraction des racines.

Mots-clés : Calcul des racines entières, méthodes de calcul chiffre par chiffre, grande base, algorithmes en ligne, évaluation des fonctions élémentaires

1 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 [5]. 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 [9]. 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 $p = \pm 2^{k_1} \pm 2^{-k_2}$, being k and k_1 any integer and k_2 any nonnegative integer, is presented in [15]. 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^y, being y any rational, are presented in [2, 10]. 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 [8] and particularized to the radix 2 cube root computation in [13]. 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 [6, 16].

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

Symbol	Definition
$X = M_x \times 2^{E_x}$	floating-point radicand
n	precision bits of radicand and result
n_{Ex}	precision bits of the exponent of X
r	radix $r = 2^b$ ($b \in N^+$, $b \geq 3$)
q	root degree ($q \in N$, $q \neq 0$)
M_q	normalized significand of q, $M_q \subset [0.5, 1)$
E_q	Normalized exponent of q, $E_q \geq 0$
$X^{1/q} = 2^{(1/q)\log_2(X)}$	q-th root function
n_q	precision bits of $\|q\|$ or M_q
M_z	Significand of the result
E_z	Exponent of the result
$T = (1/q)\log_2(X)$	argument of the q-th root function
I	integer part of T
F	fractional part of T
$D = 1/q$	reciprocal of the root degree
$S = \log_2(X) = E_x + \log_2(M_x)$	logarithm of the radicand
S^*	logarithm of the radicand shifted by 2^{-E_q}
$L = \log_2(M_x)$	logarithm of the radicand significand
D_j, L_j, S^*_j, T_j, E_j	high-radix digits of intermediate operands
$D[j]$, $L[j]$, $T[j]$, $E[j]$	intermediate results at iteration j
$w_d[j]$, $w_l[j]$, $w_m[j]$, $w_e[j]$	intermediate residuals at iteration j
n_d, n_l, n_s, n_m, n_e	precision bits of intermediate operands
g_d, g_l, g_m, g_e	guard bits of intermediate operans
N_d, N_l, N_s, N_m, N_e	latencies of intermediate operations

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 exponential.

Table 1: Symbols and notation

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 [12]. This architecture was designed for the computation of the powering function X^p, with p any integer, based on a logarithm-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 modulus 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 [12] for the computation of X^p, being p 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 expo-

nential. This formulation avoids the integer division and the modulus operation of the extension of the algorithm in [12] 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 [12], 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 2. The architecture implementing the algorithm is presented in Section 3. In Section 4, 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 5. Finally, the main conclusions are summarized in Section 6.

2 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, being X a floating-point number and $q = (-1)^{s_q} abs(q)$ a $n_q + 1$-bit signed integer exponent (see Table 1 for symbols and notation used). This algorithm is based on the algorithm presented in [12] 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 4 we propose a modification of the algorithm that avoids this limitation on the precision of q.

2.1 Overview

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

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

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

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

Equation (2) could be taken as the floating–point result for the q-th root calculation, being $2^{(1/q)\log_2(M_x)}$ the significand and E_x/q the exponent; however, the main problem for this interpretation is that the exponent is not an integer. In [12], 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 $E_x/q = \lfloor E_x/q \rfloor + (1/q) \times E_x\%q$ being % the modulus of the division. This way, equation (2) could be rewritten as follows

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

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

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

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

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

Therefore,

$$X^{1/q} = 2^{S/q} \tag{4}$$

According equation (4) the q-th root can be calculated as a sequence of operations: logarithm of the significand M_x ($log_2(M_x) \subset [0, 1)$), 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 I and fractional F parts are extracted, 2^T becomes

$$2^T = 2^I \times 2^F \tag{5}$$

so that the argument of the exponential 2^F is now in $(-1, 1)$.

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

$$X^{1/q} = 2^F \times 2^I \tag{6}$$

with

$$M_z = 2^F \quad , \quad E_z = I \tag{7}$$

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

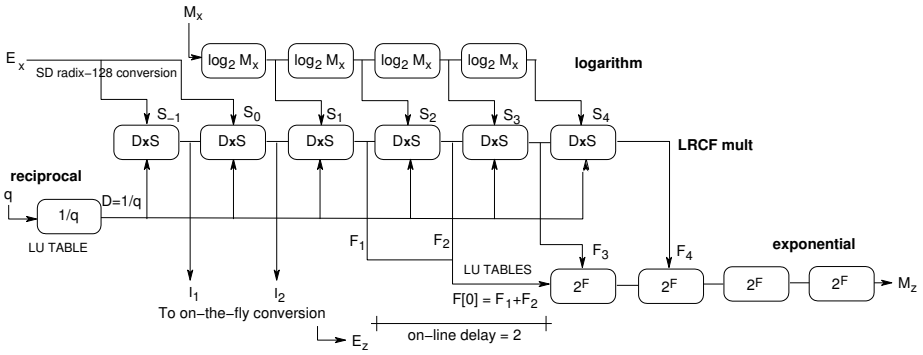

Figure 1: Sequence of operations of the algorithm.

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 $abs(1/q) \subset (0, 1]$ and $E_q > 0$. Operand D is obtained in a non-redundant binary form (Section 3.1).

2. Evaluation of the logarithm $L = \log_2(M_x) \subset [0, 1)$ to a precision of n_l bits using a high-radix algorithm (see Section 3.2).

3. Multiplication $T = D \times S$ (see Section 3.3). Operand

$$S = \sum_{i=-\lceil n_{Ex}/b \rceil + 1}^{\lceil n_l/b \rceil - 1} S_i \; r^{-i} = E_x + L$$

   is obtained serially by concatenating the digits of E_x and L, with $E_x \subset [-2^{n_{Ex}-1}, 2^{n_{Ex}-1} - 1]$ and L expressed in a signed-digit radix-r form. We evaluate this multiplication using a LRCF (left-to-right carry-free) multiplier [3]. 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 2.2.

4. Serial extraction of the integer I and fractional F parts of T, and on-the-fly conversion of I to a non-redundant representation. The integer part I corresponds to the $\gamma = \lceil n_{Ex}/b \rceil$ leading digits of T.

5. Online high-radix exponential $2^F \subset (0.5, 2)$ with argument $F = frac(T) \subset (-1, 1)$, precision of n_e bits, and online delay $\delta = 2$ (see Section 3.4). The redundant result is normalized and rounded to n bits using an on-the-fly rounding unit [4].

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^F, 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. For the example of Fig. 1, the number of cycles of the logarithm, multiplication and exponential are respectively $N_l = 4$, $N_m = 6$ and $N_e = 4$, while the latency of the q-th root algorithm is $N_{q-root} = 10$ cycles.

2.2 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 $L + \varepsilon_{log}$
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

$$\varepsilon_f = D \times \varepsilon_{log} + S \times \varepsilon_{rec} + \varepsilon_{log} \times \varepsilon_{rec} + \varepsilon_{mul} \tag{8}$$

   so

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

4. Extraction of the integer I and fractional F parts of $T = D \times S$. The integer part is of the form $I_\alpha = \lfloor T_\alpha \rfloor = I + \lfloor F + \varepsilon_f \rfloor$. The fractional part is given by

$$F_\alpha = frac(T_\alpha) = F + \varepsilon_f - \lfloor F + \varepsilon_f \rfloor$$

5. Operation 2^F. The output of the exponential computation is $2^{F_\alpha} + \varepsilon_{exp}$.

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

$$\begin{aligned} X_\alpha^{1/q} &= (2^{F_\alpha} + \varepsilon_{exp}) \times 2^{I_\alpha} \\ &= (2^{F+\varepsilon_f - \lfloor F+\varepsilon_f \rfloor} + \varepsilon_{exp}) \times 2^{I+\lfloor F+\varepsilon_f \rfloor} \\ &= (2^{F+\varepsilon_f} + \varepsilon_{exp}) \times 2^I \\ &= X^{1/q} \times 2^{\varepsilon_f} + \varepsilon_{exp} \times 2^I \end{aligned} \tag{9}$$

with $X^{1/q} = 2^F \times 2^I$.

Since $F \subset (-1, 1)$ is expressed in signed-digit, then $2^F \subset (0.5, 2)$ and we are considering n precision bits for the final normalized result, this implies that 1 $ulp = 2^n$. Assuming rounding to the nearest even, the error bound for the exponential is $2^{-(n+1)}$. Then, it must be verified that

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

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

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

Using $X^{1/q} = 2^F \times 2^I$ and taking out the common factor 2^I, the previous condition can be expressed as

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

Next, we compute an upper bound for the left term of this inequality. Since $2^F \subset (0.5, 2)$, we replace it by 2, and considering $\varepsilon_f << 1$ we use the approximation $2^{\varepsilon_f} \approx 1 + (\ln(2)/2)\varepsilon_f + O(\varepsilon_f^2) \leq 1 + \varepsilon_f/2$. 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, $\varepsilon_{log} \leq \varepsilon$, $\varepsilon_{mul} \leq \varepsilon$, 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

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

Considering the bits required for representing the integer parts (1 bit for the reciprocal $abs(1/q)$, n_{Ex} bits for the multiplication), the required precision and minumum 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 2 parametrized as a function of n and n_{Ex}. We also show the latency for the q–root computation and the corresponding values for single (SP) and double (DP) precision with $r = 128$. These values will be used to determine the number of guard bits g_l, g_m and g_e to guarantee the required precisions n_l, n_m, n_e in the corresponding units.

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

Table 2: Example of Parameters for q-th root Computation ($r = 128$, $\delta = 2$, $\gamma = \lceil n_{Ex}/b\rceil$)

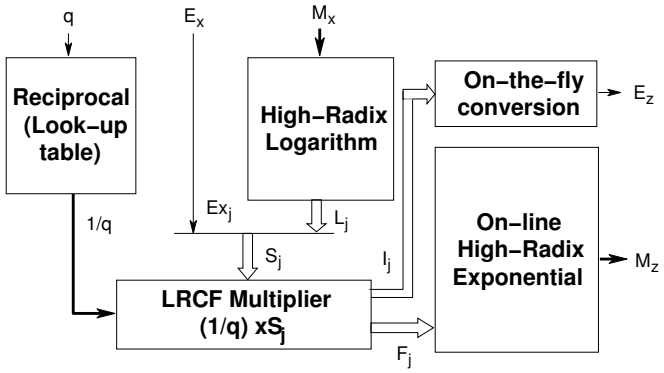

Figure 2: Block diagram of the architecture.

3 Implementation

We propose a sequential architecture for the implementation of the algorithm described in Section 2. 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 [12]. A detailed description of an on-the-fly conversion unit can be found in [4].

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.

$(n,\ n_{Ex})$	n_q							
	5	6	7	8	9	10	11	12
SP (24,8)	1.12	2.25	4.5	9	18	36	72	144
DP (53,11)	2.12	4.25	8.5	17	34	68	136	272

Table 3: Size (in Kbits) of look-up tables for reciprocal computation.

3.1 Table Lookup for 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 $abs(1/q) \subset (0, 1]$. As we have detailed in Section 2.2, 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 3 we show the size of the corresponding look-up table for single and double precision results and several values of n_q.

3.2 High-Radix Logarithm

For the computation of $\log_2(M_x)$ we use an optimized high-radix digit-recurrence algorithm similar to that implemented in [12]. This algorithm is based on the identity

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

such that if $M_x \prod f_j \leftarrow 1$ then $-\sum \log_2(f_j) \leftarrow \log_2(M_x)$, where constant factors of the form $f_j = (1 + l_j r^{-j})$ allow the use of a shift-and-add implementation.

For radix $r = 2^b$ the recurrences for the logarithm $L[j]$ and the residual $w_l[j+1]$ result in

$$\begin{aligned} L[j+1] &= L[j] - \log_2(1 + l_j r^{-j}) \\ w_l[j+1] &= r(w_l[j] + l_j + l_j w_l[j] r^{-j}) \end{aligned}$$

The number of iterations required to obtain the logarithm with n_l bits of accuracy is $N_l = \lceil n_l / b \rceil$.

The block diagram of an implementation of the previous recurrences is shown in Fig. 3. We use selection by rounding to obtain the high-radix digits l_j, with $|l_j| \leq (r-1)$, except for the first digit l_1, which is obtained by a lookup table.

The rounding is performed on an estimate $w_l[\widehat{j+1}]$ obtained by truncating to t fractional bits the borrow-save residual $w_l[j+1]$. The constants $\log_2(1 + l_j r^{-j})$ are stored in lookup tables. In order to simplify these tables, the logarithm constants are approximated by $-l_j/ln(2)$ from iteration $1 + \lceil N_l/2 \rceil$.

At the end of each iteration $j \geq 1$ we obtain a high-radix digit L_j of the logarithm in a borrow-save form. The recurrence $L[j]$ is scaled by r so that each digit L_j is obtained by the extraction of the b leading bits of the scaled borrow-save $L[j]$.

3.3 LRCF Multiplication

The left-to-right carry-free (LRCF) multiplication [3] produces the product digits T_j from a redundant set in a most-significant-digit-first (MSDF) manner.

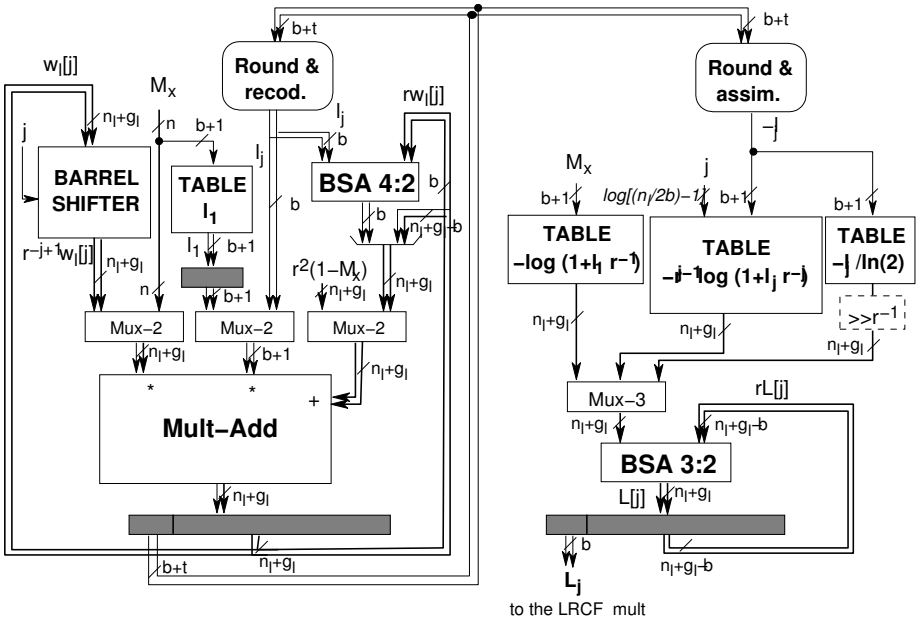

Figure 3: Block diagram of the logarithm stage.

We use a LRCF multiplier to perform the intermediate multiplication in the low precision q-th root algorithm. We adapt the original architecture to fit our requirements as in [12]. This stage computes the following two recurrences for the residual $w_m[j+1]$ and the product $T[j]$:

$$\begin{aligned} w_m[j+1] &= r(frac(w_m[j] + D \times S_{j+1}) \\ K_j &= \lfloor w_m[j] + D \times S_{j+1} \rfloor \\ T[j+1] &= T[j] + K_j r^{-j-1} \end{aligned}$$

where the maximum value of $|K_j| < 3r/2$ can be larger than $(r-1)$. Before starting the computation of iterations, the radix points of operands D and S are adjusted such that $|D| < 1$ and $|S| < 1$.

The block diagram of the LRCF multiplication stage is shown in Fig. 4.

The LRCF multiplier consists of a multiply-add unit which computes $w_m[j] + D \times S_{j+1}$ and a recoding block to obtain T_j from K_j and K_{j-1}. The inputs to the multiply-add unit are the operand $D = 1/q$, the high-radix digits S_i (in borrow-save format) of the operand $S = E_x + log_2(M_x)$ and the residual $w_m[j]$. The digits S_j come either from the high-radix representation of E_x or from the evaluation of the logarithm (digits L_j). In each iteration the operand D is multiplied by a high-radix digit S_i and accumulated to the previous residual $w_m[j]$. The next residual is obtained by scaling by r the fractional part of the result of the multiply-add operation.

The product T has integer I and fractional F parts. Since we have consider similar error bounds for the logarithm and multiplication (see Section 2.2), we need to compute the result within $n_m = n_{Ex} + n_l$ bits of accuracy. The integer part consists of the $\gamma = \lceil n_{Ex}/b \rceil$ most significant digits T_j obtained as $T_j = recod(K_j, K_{j-1})$. These digits are passed to a on-the-fly conversion unit.

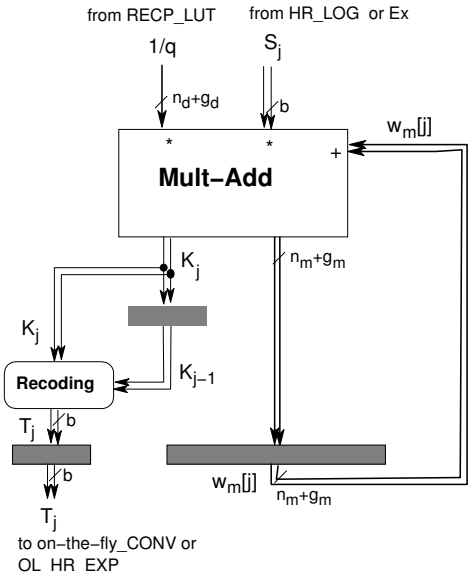

Figure 4: Block diagram of the LRCF multiplier.

Since the fractional part is obtained with n_l bits of accuracy, in the remaining iterations we compute $N_l = \lceil n_l/b \rceil$ digits of the fractional part of T, which are used by the exponential unit without conversion. The total number of iterations of the LRCF multiplication is $N_m = \gamma + N_l$.

3.4 On-line high-radix exponential

The online high-radix algorithm for the computation of 2^F is detailed in [12]. This algorithm is based on the identity

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

with $h_j = (1 + e_j r^{-j})$. The input operand F is only available up to the $j + \delta$ digit at iteration j, as $F[0] = \sum_{i=1}^{\delta} F_j r^{-j}$, with δ the online delay and F_j the digits of F. The recurrences for the residual and the exponential are given by:

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

with $j \geq 1$, $E[1] = 1$ and $w_e[1] = rF[0]$. The number of iterations required to obtain the exponential with n_e bits of accuracy is $N_e = \lceil n_e/b \rceil$.

The block diagram of the exponential unit is shown in Fig. 5. We use selection by rounding to obtain the high-radix digits e_j, with $|e_j| \leq (r-1)$, except for the first digit e_1, which is obtained by a lookup table. The lookup tables used in the first iteration are addresses one cycle in advance, since $F[0]$ is known while $F_{1+\delta}$ is being computed. The rounding is performed on an estimate $w_e\widehat{[j+1]}$ obtained by truncating to t fractional bits the borrow-save residual $w_e[j+1]$.

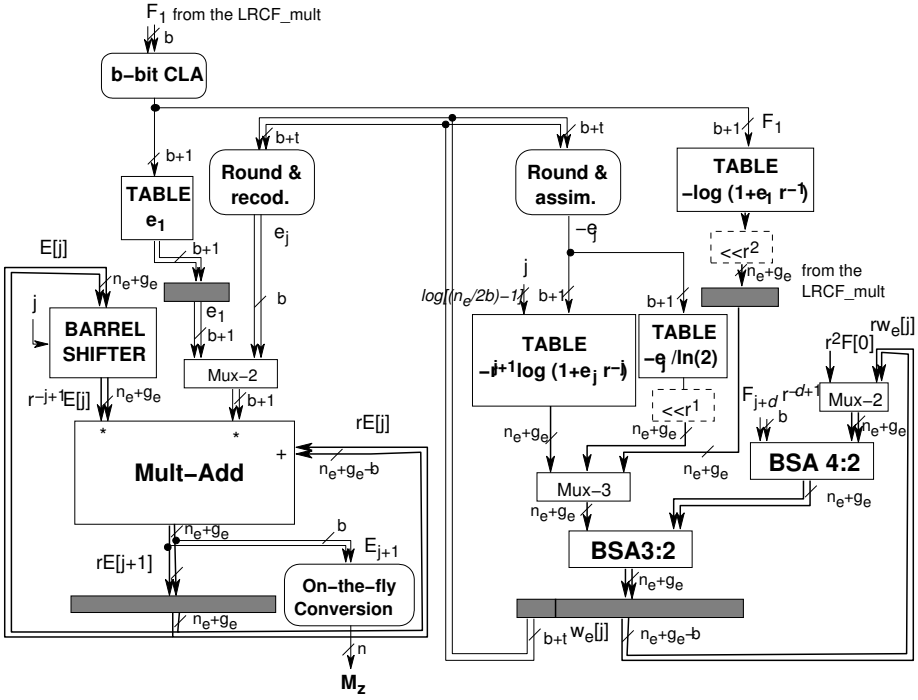

Figure 5: Block diagram of the exponential stage.

The constants $\log_2(1+e_j r^{-j})$ for iterations 2 to $\lceil N_e/2 \rceil$ are stored in lookup tables. The exponential constants are approximated by $-re_j/ln(2)$ from iteration $1+\lceil N_e/2 \rceil$. At the end of each iteration we obtain a high-radix digit E_j of the logarithm in a borrow-save form. The recurrence $E[j]$ is scaled by r so that each digit E_j is obtained by the extraction of the b leading bits of the scaled borrow-save $E[j]$.

The conversion of the high-radix digits E_j of the exponential to the non redundant binary result is performed in an on-the-fly rounding unit [4], avoiding an increase in neither the latency of the algorithm nor the cycle time.

4 Implementation for higher precision q

The main limitation of the algorithm proposed in Section 2 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 2 needs to be modified.

For the computation of the reciprocal we propose a high-radix iterative algorithm described in Section 4.1. The divisor is required to be in the convergence range of the algorithm, which in our case is $[0.5, 1)$. Therefore $abs(q) \subset [1, 2^{n_q} - 1]$ is normalized into $M_q \subset [0.5, 1)$, 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 $D \times 2^{-E_q}$ with $D = (-1)^{s_q} \times (1/M_q)$, so that the algorithm performs $2^{D \times 2^{-E_q} \times S}$ with

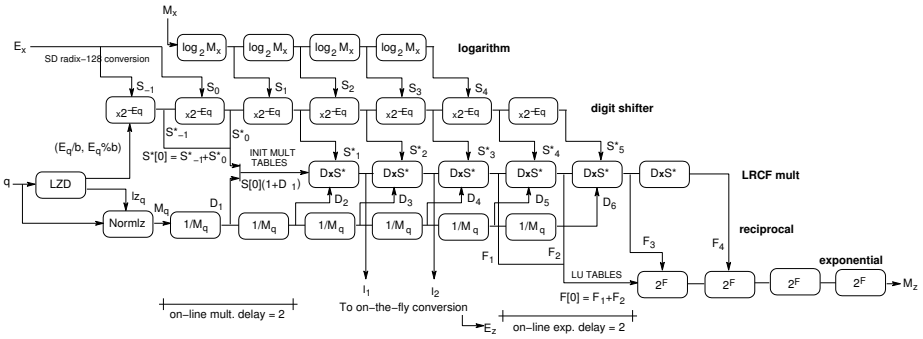

Figure 6: Sequence of operations of the modified algorithm.

$S = E_x + \log_2(M_x)$. In order to reduce the latency of the algorithm, the factor 2^{-E_q} is first multiplied by S and the result by D.

An example of the operation flow of the modified q-th root algorithm for single precision and $r = 128$ is shown in Fig. 6. The sequence of operations of the algorithm for $n_q > 12$ is as follows:

1. Evaluation of the number of leading zeros $lz_q \subset [0, n_q - 1]$ of q using a leading zero detector (LZD). The LZD is enhanced (see Section 4.2) to allow the computation of $\lfloor E_q/b \rfloor$ and $E_q\%b$ in parallel, with the exponent E_q 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 $M_q \subset [0.5, 1)$ by shifting q an amount of lz_q bits to the left.

3. Evaluation of $D = (-1)^{s_q} \times (1/M_q) \subset (-2, 2)$ using a high-radix iterative algorithm with multiplicative decomposition (see Section 4.1). The digits D_j of the reciprocal are in a signed-digit radix-r redundant form.

4. Evaluation of the logarithm $L = \log_2(M_x) \subset [0, 1)$ to a precision of n_l bits using the high-radix algorithm described in Section 3.2.

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 Section 4.2.

6. Multiplication $T = D \times S^*$ (see Section 4.3). 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 [17] with online delay $\delta = 2$.

7. Serial extraction of the integer I and fractional F parts of T, and on-the-fly conversion of I to a non-redundant representation.

Target Precision	**Precision of intermediate operations (bits)**				
	n_d	n_l	n_s	n_m	n_e
$(n,\ n_{Ex},\ n_q)$	n_{Ex}+n+4	n+4	n_{Ex}+n+5	n_{Ex}+n+4	n+2
SP (24,8,32)	36	28	37	36	26
DP (53,11,64)	68	57	69	68	55
Target Precision	**Latency of intermediate operations (cycles)**				
	N_d	N_l	N_s	N_m	N_e
$(n,\ n_{Ex},\ n_q)$	$\lceil n_d/b \rceil$	$\lceil n_l/b \rceil$	γ+$\lceil (n_l$+1$)/b \rceil$	γ+$\lceil n_l/b \rceil$	$\lceil n_e/b \rceil$
SP (24,8,32)	6	4	2+5	2+4	4
DP (53,11,64)	10	9	2+9	2+9	8
Target Precision	n_{q-root} (bits)		**q–th root latency**		
$(n,\ n_{Ex},\ n_q)$	n		3+2δ+γ+$\lceil n_e/b \rceil$		
SP (24,8,32)	24		13		
DP (53,11,64)	53		17		

Table 4: Example of Parameters for q-th root Computation ($r = 128$, $\delta = 2$, $\gamma = \lceil n_{Ex}/b \rceil$)

8. On-line high-radix exponential $2^F \subset (-2, 2)$ with argument $F = frac(T) \subset (-1, 1)$, precision of n_e bits, and online delay $\delta = 2$ (see Section 3.4). The redundant result is normalized and rounded to n bits using an on-the-fly rounding unit [4].

To obtain the required precisions for the different intermediate operations, we have performed an error analysis similar to the analysis in Section 2.2. The required precisions for each operation (reciprocal, logarithm, signed-digit shifting, multiplication and exponential) are shown in Table 4. These parameters determine the latencies of the intermediate operations for radix $r = 2^b$. Table 4 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 amounts 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 on-line delay and the latency of the exponential 2^F, 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. 7.

Next, we only describe the new units, that is, high-radix reciprocal unit, the LZD, the signed-digit shifter and the on-line multiplier. The high-radix logarithm and on-line high-radix exponential units were detailed in Section 3.

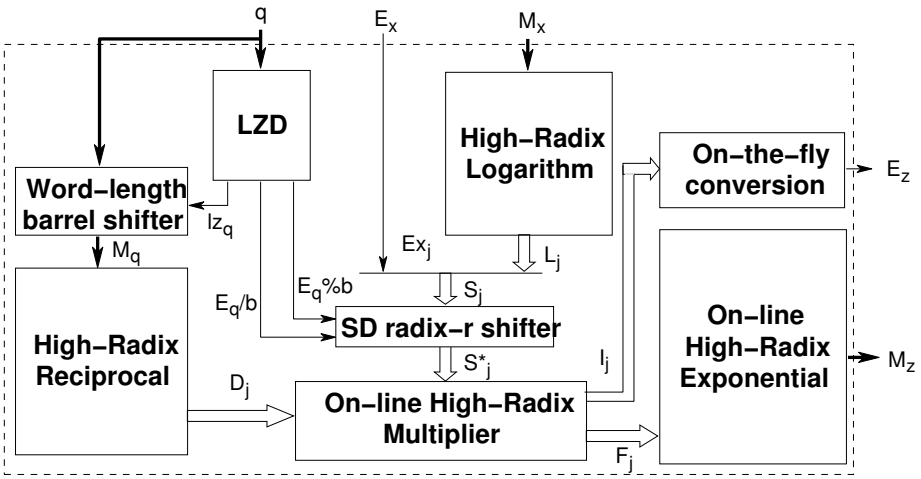

Figure 7: Block diagram of the modified architecture.

4.1 High-radix digit-recurrence reciprocal

High-radix SRT methods [5] use an additive decomposition for the reciprocal. To allow selection by rounding for the quotient digits in these type of methods, the initial residual needs to be scaled, which introduces additional delay. When a correct rounded reciprocal is not a requirement, we can use a multiplicative decomposition of $1/M_q$ instead of a sum of terms to avoid the initial scaling [5]. Thus, we decompose the reciprocal in a multiplication of factors determined by a digit, that is, $1/M_q = \prod_{i=1}^{\infty}(1 + d_i r^{-i})$, such that at iteration j the reciprocal is approximated by $Q[j] = \prod_{i=1}^{j}(1 + d_i r^{-i})$.

Observing that $M_q Q[j] \rightarrow 1$ when $j \rightarrow \infty$, we define a scaled residual as $w_d[j] = r^j(1 - M_q D[j])$, obtaining the following recurrences for the reciprocal and the residual:

$$\begin{aligned} Q[j+1] &= Q[j](1 + d_{j+1} r^{-j-1}) \\ w_d[j+1] &= r w_d[j] - d_{j+1} + d_{j+1} w_d[j] r^{-j} \end{aligned}$$

with $Q[0] = 1$. As in the SRT algorithms, the convergence is linear, obtaining (roughly) one digit of the result in each iteration. For $N_d = \lceil n_d/b \rceil$ iterations we obtain n_d precision bits of the reciprocal ($|\varepsilon_d| \leq 2^{-n_d}$). Instead of $Q[j]$, a scaled reciprocal $D[j] = Q[j]r^j$ is computed in order to extract a radix-r digit D_j per iteration from the leading position in all iterations $j \geq 1$ (digit $D_0 = 1$ is implicit).

The selection of digits $d_{j+1} \in \{-(r-1), \ldots, r-1\}$ is performed as $d_{j+1} = round(\widehat{rw[j]})$, where $\widehat{rw[j]}$ denotes the r-shifted residual $w[j]$ truncated to t fractional bits. To ensure convergence with selection by rounding for $j > 1$ and $M_q \subset [0.5, 1)$, d_1 is selected from the over-redundant digit set $\{-(2r-1), \ldots, 2r-1\}$ by table look-up. This table is addressed by the $b+2$ most significant bits of M_q.

The block diagram of this unit is shown in Fig. 8. A more detailed description of this unit can be found in [18].

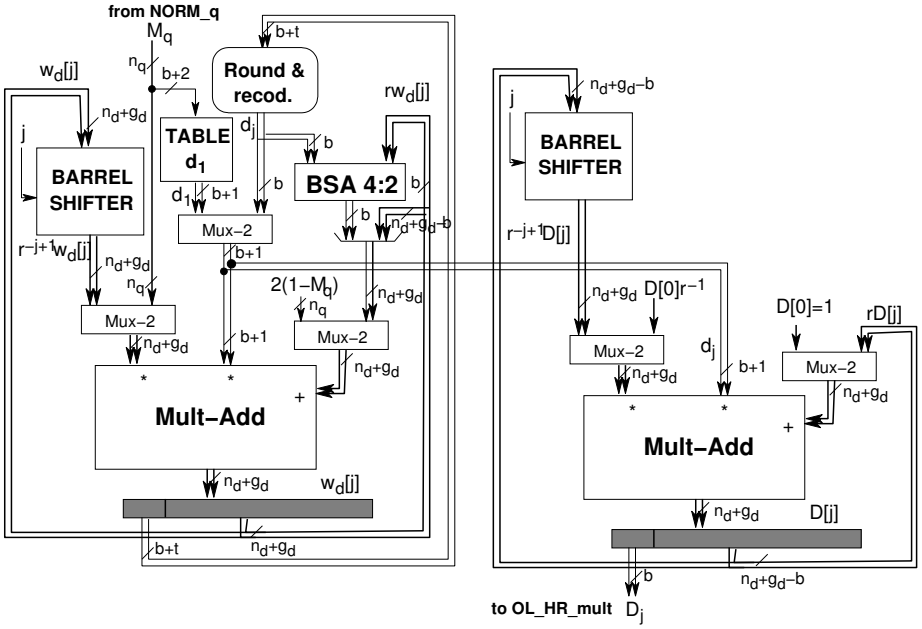

Figure 8: Block diagram of the reciprocal unit.

4.2 Enhanced LZD and signed-digit shifter

Since $S = E_x + \log_2(M_x)$ is obtained in the form of a sequence of $\gamma + N_l$ radix-2^b digits, the multiplication $S^* = S \times 2^{-E_q}$ can only be performed as a digit right shift of S when E_q is an integer multiple of b. A more general solution is to decompose E_q in two terms as

$$E_q = \lfloor E_q/b \rfloor \times b + E_q \% b$$

so that the operation is computed as a compound right shift of $\lfloor E_q/b \rfloor$ radix-r digits plus $E_q\%b$ bits. Provided that the maximally redundant radix-r digits S_i are represented in a borrow-save (or carry-save) form and the radix is an integer power of 2 ($r = 2^b$), the right shift of $E_q\%b$ bits can be implemented as a regular binary shift [1]. The serial architecture proposed is shown in Fig. 9. The unit processes a digit ${S^*}_i$ per cycle ($0 \le i < N_s$). First, a right shift of $E_q\%b < b$ bits is performed over S_j ($-\gamma < j < N_l$) using two barrel shifters of $2b$ bits. Since this binary shift crosses the digit boundaries, the previous input digit S_{j-1} is latched and then concatenated at cycle i to the left of S_j. The initial value of this latch is 0. The b most significant bits of the two barrel shifter output are passed to a level of latched 2:1 multiplexers controlled by a signal[2] $rsa = \lfloor E_q/b \rfloor < N_s$.

This signal represents the number of radix-r digit shifted to the right in a hot-one code. The output of the barrel shifters is stored in the latch which corresponds to the number of shifted positions counted from the right (from

[1] That is, a bit shifted out to the right of a radix-r digit halves its value.

[2] This bound is ensured by error analysis, so that S^* is computed with enough precision bits n_s.

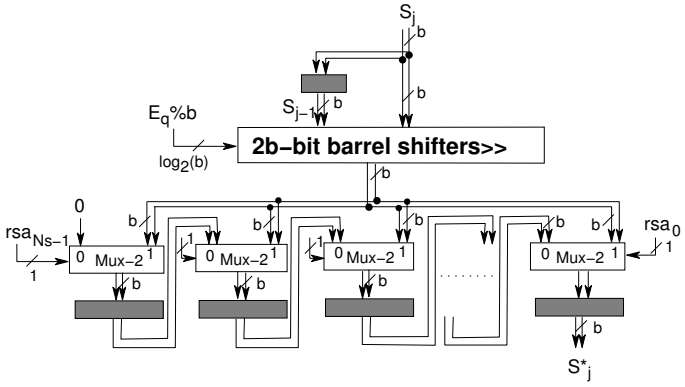

Figure 9: Block diagram of the serial signed-digit shifter.

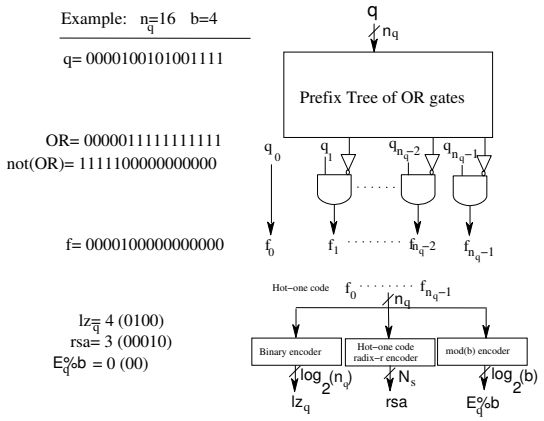

Figure 10: Enhanced leading zero detector.

$N_s - 1$ to 0). The latch outputs are shifted one radix-r digit to the right each cycle, while S_i^* corresponds to the value stored in the rightmost latch.

The shifting amounts signals rsa and $E_q\%b$ are obtained with a slight modification of the decoding logic in the LZD using the relation $E_q = n_q - lz_q$. The architecture of this enhanced LZD is shown in Fig. 10. First, a hot-one code string $f_0 \dots f_1 f_2 f_{n_q-1}$ indicating the position of the leading one of $q = \sum_{i=0}^{n_q-1} q_i \; 2^{n_q-1-i}$ is obtained using a tree of OR gates (prefix tree, carry lookahead...) and a level of AND-2 gates with an inverted input. These signals are computed as

$$f_i = q_i \cdot \overline{OR_{j=0}^{i-1} \; q_j}$$

Then, lz_q , $rsa = \lfloor E_q/b \rfloor$ and $E_q\%b$ are computed as

$$\begin{aligned} lz_q &= \sum_{i=0}^{n_q-1} i \times f_i \\ rsa &= \sum_{i=0}^{n_q-1} \lfloor (n_q - i)/b \rfloor \times f_i \end{aligned}$$

$$(E_q\%b) \quad = \quad \sum_{i=0}^{n_q-1} (n_q - i)mod(b) \times f_i$$

with an appropriate decoding of the hot-one code signals f_i using OR gates. Namely, the signal lz_q is decoded to a binary operand of $\lceil \log_2 n_q \rceil$ bits width, the signal rsa into a hot-one radix-r operand of N_s bits width, and the signal N_s into a binary (modulo b) operand of $\lceil \log_2 b \rceil$ bits width. We show a toy example for $n_q = 16$ $q = 2383$ and $b = 4$ (radix-16) in the left side of Fig. 10. We obtain $lz_q = 4$, $rsa = 3$ and $(E_q\%b) = 0$ as follows:

1. We compute signals f_0 to f_{15} using the prefix tree of OR gates ($\lceil \log_2(16) \rceil$-1=3 levels of two-input gates), and the level of NOT-AND2 gates, obtaining $f_4 = 1$.

2. We obtain $lz_q = 4$ (0100_2) from the binary encoder, implemented using the following boolean expressions:

$$\begin{aligned} lz_q(3) &= f_8 \vee f_9 \vee f_{10} \vee f_{11} \vee f_{12} \vee f_{13} \vee f_{14} \vee f_{15} \\ lz_q(2) &= f_4 \vee f_5 \vee f_6 \vee f_7 \vee f_{12} \vee f_{13} \vee f_{14} \vee f_{15} \\ lz_q(1) &= f_2 \vee f_3 \vee f_5 \vee f_7 \vee f_{10} \vee f_{11} \vee f_{14} \vee f_{15} \\ lz_q(0) &= f_1 \vee f_3 \vee f_5 \vee f_7 \vee f_9 \vee f_{11} \vee f_{13} \vee f_{15} \end{aligned}$$

3. The hot-one code radix-16 encoder computes the value $rsa = \lfloor (16 - lz_q)/4 \rfloor \in [0, 4]$, producing an output $rsa = 3$ expressed in a hot-one code (00010). The logical expressions of this block are given by

$$\begin{aligned} rsa(4) &= f_0 \\ rsa(3) &= f_1 \vee f_2 \vee f_3 \vee f_4 \\ rsa(2) &= f_5 \vee f_6 \vee f_7 \vee f_8 \\ rsa(1) &= f_9 \vee f_{10} \vee f_{11} \vee f_{12} \\ rsa(0) &= f_{13} \vee f_{14} \vee f_{15} \end{aligned}$$

4. The modulo b=4 encoder produces a binary string of $log_2(4) = 2$ bits which represents the value $(16 - lz_q)\%4 \in [0, 3]$. The logical expressions of this block are given by

$$\begin{aligned} E_q\%4(1) &= f_1 \vee f_2 \vee f_5 \vee f_6 \vee f_9 \vee f_{10} \vee f_{13} \vee f_{14} \\ E_q\%4(0) &= f_1 \vee f_3 \vee f_5 \vee f_7 \vee f_9 \vee f_{11} \vee f_{13} \vee f_{15} \end{aligned}$$

   For the example of Fig. 10, since $lz_q = 4$, then $f_4 = 1$ and $(16 - lz_q)\%4 = 0$.

4.3 On-line multiplier

On-line multiplication [17] of $P = X \times Y$ is defined by the following recurrence equation for the scaled partial product $w_m[j] = r^j X[j]Y[j]$:

$$w_m[j+1] = r(w_m[j] - P_j) + r^{-\delta}(X_{j+\delta+1}Y[j] + Y_{j+\delta+1}X[j+1]) \qquad (10)$$

with $\delta = 2$ for $r > 2$, $X[j] = \sum_{i=1}^{j+\delta} X_i r^{-i}$, $Y[j] = \sum_{i=1}^{j+\delta} Y_i r^{-i}$. The initial condition for the algorithm is $w_m[0] = X[0] \times Y[0]$ and $P_0 = 0$, where the

product digits $P_j \subset \{-(r-1)\ldots,0,\ldots,r-1\}$ are obtained in our case by a selection function $P_{j+1} = round(w_m\widehat{[j+1]})$, with $w_m[j+1]$ truncated to t bits.

After iteration $j \geq 0$, the product is given by

$$P = P[j+1] + (w_m[j+1] - P_{j+1})r^{-j-1}$$

with $P[j+1] = \sum_{i=1}^{j+1} P_i r^{-i}$. For convergence we have to verify the following conditions:

- General condition of convergence for the iteration: $|w_m[j+1] - P_{j+1}| \leq r^{i+1}(P - P[j+1])$. Since $|P_j| \leq (r-1)$, and

$$P - P[j+1] \leq \sum_{i=j+2}^{\infty} (r-1)r^{-i} \leq r^{-j-1}$$

  this condition can be formulated as $|w_m[j+1]| \leq (r-1) + r^{i+1}r^{-j-1}$, that is $|w_m[j+1]| \leq r$.

- Condition of convergence for selection by rounding: $|w_m[j+1]| < (r-1) + 1/2$, that is $|w_m[j+1]| < r - 1/2$.

To obtain upper bounds for the values of X and Y we check the second condition (more restrictive) for $j = 0$. The scaled partial product $w_m[1]$ given by recurrence (10) with $j = 0$ can be expressed as

$$\begin{aligned} w_m[1] &= rw_m[0] + r^{-\delta}(X_{\delta+1}Y[0] + Y_{\delta+1}X[1]) \\ &= r(X[0]Y[0]) + r^{-\delta}(X_{\delta+1}Y[0] + Y_{\delta+1}X[1]) \\ &= Y[0]r(X[0] + X_{\delta+1}r^{-\delta-1}) + X[1]r(Y_{\delta+1}r^{-\delta-1}) \\ &= rY[0]X[1] + X[1]r(Y_{\delta+1}r^{-\delta-1}) \\ &= rX[1](Y[0] + Y_{\delta+1}r^{-\delta-1}) = rX[1]Y[1] \end{aligned} \tag{11}$$

Using $r|X[1]Y[1]| \leq r(|X \times Y|)$ in the previous expression, the condition of convergence for selection by rounding limits the values of the input operands to

$$|X \times Y| < 1 - 1/2r$$

To obtain a value for the minimum number of fractional bits t required for the estimate $w_m\widehat{[j+1]}$, we use the condition of convergence $|w_m[j+1]| < r - 1/2$ for $j > 0$. In this case, a bound for $w_m[j+1]$ is given by

$$w_m[j+1] \leq r(1/2 + 2^{-t}) + r^{-\delta}2(r-1)(1-1/2r)^{1/2} \tag{12}$$

obtained by introducing the following bounds in expression (10): $|w_m[j] - P_j| \leq 1/2 + 2^{-t}$, $|X_{j+\delta+1}| \leq (r-1)$, $|Y_{j+\delta+1}| \leq (r-1)$, $|Y[j-1]| \leq (1-1/2r)^{1/2}$, and $|X[j]| \leq (1-1/2r)^{1/2}$.

With the previous bound for $|w_m[j+1]|$, the condition of convergence results in

$$r(1/2 + 2^{-t}) + r^{-\delta}2(r-1)(1-1/2r)^{1/2} < r - 1/2 \tag{13}$$

obtaining a minimum value of $t = 1$ for $r \geq 4$ and $\delta = 2$.

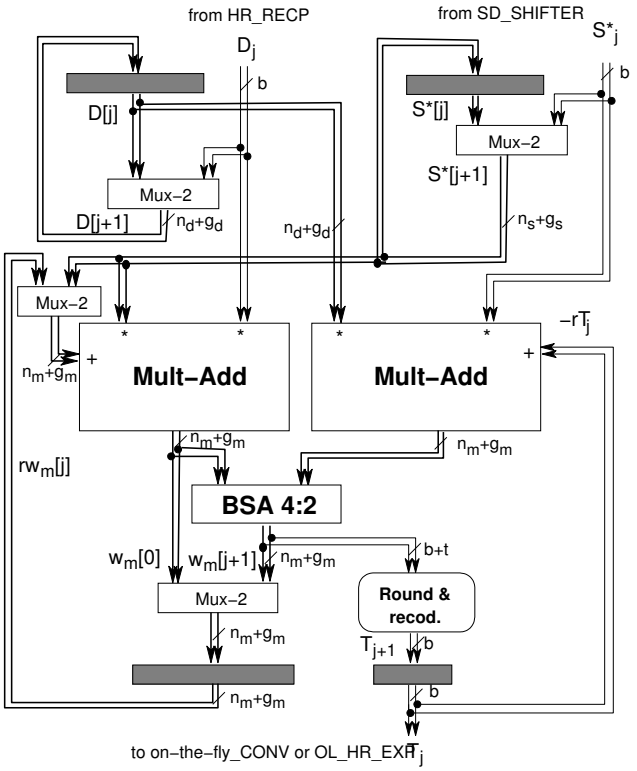

Figure 11: Block diagram of the on-line multiplier.

We perform the operation $T = D \times S^*$ using the online multiplier of Fig. 11. Since operands D and S^* are of the form $D = \sum_{i=0}^{\lceil (n_d+g_d)/b \rceil} D_i r^{-i}$ with $D_0 = 1$, and $S^* = \sum_{i=0}^{\lceil n_{Ex}/b \rceil - 1} {S^*}_i r^i + \sum_{i=1}^{\lceil (n_s+g_s)/b \rceil} {S^*}_i r^{-i}$, they need to be scaled by a constant to verify $X \times Y < 1 - 1/2r$. Thus, we scale both operands as follows, $X = Dr^{-1}$ ($X_1 = 1$), and $Y = S^* r^{-\lceil n_{Ex}/b \rceil}$, so that $X \times Y \geq 2/r$ and $X \times Y < 1 - 1/2r$ is verified for $r \geq 4$. The product is given by $T = Pr^{\lceil n_{Ex}/b \rceil + 1}$.

We need an initial cycle to compute $w_m[0] = D[0]S^*[0] = S^*[0] + D_1 S^*[0]$, and $N_m = \gamma + N_l$ iterations of the recurrence (10) followed by the selection of $P_{j+1} = round(\widehat{w_m[j+1]})$ to get an accuracy of n_m bits (n_{Ex} integer, n_l fractional) of the product T.

5 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 Section 3 and 4. 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 5.1.

Component	Delay ($FO4$)	Area (NAND2)
Nand2	0.7	1
Xor2	1.3	2.5
FA	3.2	7.5
1-bit 4:2 CSA	5	15
4-bit CLA	5.5	35
1-bit D-latch	2.5	3.5

Table 5: Area and delay values for CMOS components.

	n_q							
	5	6	7	8	9	10	11	12
Single Precision (n, n_{Ex})=(24,8)								
Size (Kbits)	1.12	2.25	4.5	9	18	36	72	144
Area (♯Nand2)	335	675	1180	2360	4725	9450	18900	32400
Delay (♯FO4)	6.4	8	9.6	11.2	12.8	14.4	14.4	16
Double Precision (n, n_{Ex}) =(53, 11)								
Size (Kbits)	2.12	4.25	8.5	17	34	68	136	272
Area (♯Nand2)	635	1275	2230	4460	8925	17850	35700	61200
Delay (♯FO4)	6.4	8	9.6	11.2	12.8	14.4	14.4	16

Table 6: Size, area and delay of look-up tables for reciprocal computation

We use an area and delay evaluation model based on a simplification logical effort method [14] that allows for faster hand calculations. It considers the different input and output gate loads, but neither interconnections nor gate sizing optimizations. Instead, we use other optimization techniques, such as buffering or cloning (gate replication) to drive high loads. The total stage delay is obtained as the sum of the delays of the gates on the critical path. 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. In Table 5 we detail the delay and area of some common CMOS gates and logic components.

The hardware complexity and delay figures for look-up tables shown in Table 6 were extracted from synthesis evaluations as in [12] (see Apendix A in [11] for more details).

Tables 7 and 8 show 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 and double precision computations respectively. The implemented radix values go from $r = 8$ to $r = 1024$.

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

Radix	Latency (♯cycles)	Cycle T. (♯FO4)	Exec. T. (♯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 7: Area and delay of the low precision q architecture with $n_q = 8$ and single precision ($n = 24$, $n_{Ex} = 8$).

Radix	Latency (♯cycles)	Cycle T. (♯FO4)	Exec. T. (♯FO4)	LOG Area (NAND2)	Exp. Area (NAND2)	Total Area (NAND2)
8	27	31	837	11350	11000	31400
16	21	33	693	12700	12300	34800
32	18	33	594	18100	17500	45700
64	16	34	544	20600	20000	51400
128	14	34	476	32900	31800	75700
256	13	36	468	58000	56100	125800
512	13	36	468	98000	94800	204800
1024	12	37	444	133300	129100	275200

Table 8: Area and delay of the low precision q architecture with $n_q = 8$ and double precision ($n = 53$, $n_{Ex} = 11$).

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 and 4460 NAND2 gates for double 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.

Tables 9 and 10 show the corresponding estimates for the high–precision q architecture, with $n_q = 32$ and $n_q = 64$ for the single and double precision computations respectively.

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. The contribution of this unit to the total area is more significant in percentage for low radix values. Thus, for a given radix, the area of this architecture is higher than the area of the low precision q architecture for n_q. Besides, there is a latency overhead of 3 cycles. On the other hand, a variation in the precision n_q has a negligible impact on the total area or execution time.

Radix	Latency (♯cycles)	Cycle T. (♯FO4)	Exec. T. (♯FO4)	Area (NAND2)			
				Recip.	Log.	Exp.	Total
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

Table 9: Area and delay of the high precision q architecture with $n_q = 32$ and single precision ($n = 24$, $n_{Ex} = 8$).

Radix	Latency (♯cycles)	Cycle T. (♯FO4)	Exec. T. (♯FO4)	Area (NAND2)			
				Recip.	Log.	Exp.	Total
8	30	31	930	11200	11350	11000	44700
16	24	33	792	12500	12700	12300	50250
32	21	33	693	13200	18100	17500	62100
64	19	34	646	15000	20600	20000	70400
128	17	34	578	16200	32900	31800	96100
256	16	36	576	18800	58000	56100	149700
512	16	36	576	22300	98000	94800	232500
1024	15	37	555	28500	133300	129100	310000

Table 10: Area and delay of the high precision q architecture with $n_q = 64$ and double precision ($n = 53$, $n_{Ex} = 11$).

Thus, 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 11 area estimations of the low precision q architecture, for single and double 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 architecture for a radix value $r = 128$. We observe that for both single and double precision computations, 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.

5.1 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 [12], 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

q-th root unit with reciprocal by lookup table: Area (♯NAND2)								
Single precision ($n = 24$, $n_{Ex} = 8$, 10 cycles latency)								
n_q	5	6	7	8	9	10	11	12
Recip.	335	675	1180	2360	4725	9450	18900	32400
Total	26200	26600	27100	28250	30600	35350	44800	58300
Double precision ($n = 53$, $n_{Ex} = 11$, 14 cycles latency)								
n_q	5	6	7	8	9	10	11	12
Recip.	635	1275	2230	4460	8925	17850	35700	61200
Total	71900	72500	73500	75700	80200	89100	107000	132500

q-th root unit with reciprocal by high-radix digit-recurrence			
	Latency (cycles)	Area $1/q$ (NAND2)	Total Area (NAND2)
SP	13	10850	40500
DP	17	18600	96100

Table 11: Determining the best unit in terms of cost for $r = 128$ as a function of n_q. SP (n=24, n_{Ex}=8, n_q=32), DP (n=53, n_{Ex}=11, n_q=64).

of any q–th root without any additional modification. It has to be pointed out that the complexity of the extended algorithm in [12] 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 [2, 10, 15] and digit–recurrence algorithms [8]. Among the former, in [15] a method for generating X^p for a given p is proposed, applicable to values of $p = \pm 2^k$ or $p = \pm 2^{k_1} \pm 2^{k_2}$ being k_1 an integer and k_2 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 $X^p = C \times X'$ being $X' = X_1 + 2^{-m-1} + p \times (X_2 - 2^{-m-1})$, and X_1 and X_2 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 [10], 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 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 [2]. 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 need to complete the powering computation.

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 [12]	9	34	306	29827
$X^{1/q}$ ($n_q = 8$) [12]	9	34	306	> 500000
Linear approx. [15]	1	51	51	26122
2nd–order interp. [10]	3	18.7	56.1	10170
2nd–order interp. [2]	2	54.4	108.8	10612
Digit–recurr. ($q = 3$, $r = 2$) [13]	52	119	6188	9035

Table 12: Architecture features comparison for single–precision floating–point representation and low precision q.

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 [8]. 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 [13].

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 [12].

Table 12 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 $r = 128$.

As shown in Tables 7-10, a good tradeoff between latency, cycle time and area can be achieved for specific values of the radix. Although it is difficult to select just one radix for every implementation, in applications demanding high-speed processing one of the most efficient implementations corresponds to 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 [13].

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 [12] 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 [12] 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 modulus of the integer division $E_x\%q$ has been evaluated, even for low precision q.

6 Conclusion

An algorithm for q–th root extraction has been presented, based on a high–radix composite algorithm. It consists of computing $X^{1/q}$ as $2^{(1/q)\times(E_x+log_2(M_x))}$ through a sequence of parallel and /or overlapped operations: reciprocal, high–radix digit–recurrence logarithm, high–radix left–to–right carry–free multiplication and on–line high–radix 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 modulus 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 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 and double precision floating–point representations 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 which would be able to compute any function of type $X^{p/q}$.

References

[1] J. Cao and B.W.Y Wei, *High-performance hardware for function generation*, Proc. 13th IEEE Symposium on Computer Arithmetic, pp.184–188, Jul. 1997.

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

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

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

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

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

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

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

[9] J.-M. Mueller, *Elementary Functions, Algorithms and Implementation*, Birkhäuser, 1997.

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

[11] J.-A. Piñeiro, *Algorithms and Architectures for Elementary Function Computation*, Ph.D. Dissertation, Dept. of Electronics and Computer Engineering, University of Santiago de Compostela, Spain, Jun. 2003. Available at http://www.ac.usc.es.

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

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

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

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

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

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

[18] A. Vazquez, E. Antelo and T. Lang, *Combined Division/Square-Root/Reciprocal Square-Root Unit for 3D Graphics Geometry Processing*, Tech. Report, Department of Electronic and Computer Science. University of Santiago de Compostela, Spain. Sep. 2000. Available at http://www.ac.usc.es.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399