High-Speed Function Approximation using a Minimax Quadratic Interpolator

Jean-Michel Muller, Stuart Oberman, Jose-Alejandro Pineiro, Javier Bruguera

High-Speed Function Approximation Using a Minimax Quadratic Interpolator

Jose-Alejandro Piñeiro, Stuart F. Oberman, *Member*, *IEEE Computer Society*, Jean-Michel Muller, *Senior Member*, *IEEE*, and Javier D. Bruguera, *Member*, *IEEE*

Abstract—A table-based method for high-speed function approximation in single-precision floating-point format is presented in this paper. Our focus is the approximation of reciprocal, square root, square root reciprocal, exponentials, logarithms, trigonometric functions, powering (with a fixed exponent p), or special functions. The algorithm presented here combines table look-up, an enhanced minimax quadratic approximation, and an efficient evaluation of the second-degree polynomial (using a specialized squaring unit, redundant arithmetic, and multioperand addition). The execution times and area costs of an architecture implementing our method are estimated, showing the achievement of the fast execution times of linear approximation methods and the reduced area requirements of other second-degree interpolation algorithms. Moreover, the use of an enhanced minimax approximation which, through an iterative process, takes into account the effect of rounding the polynomial coefficients to a finite size allows for a further reduction in the size of the look-up tables to be used, making our method very suitable for the implementation of an elementary function generator in state-of-the-art DSPs or graphics processing units (GPUs).

Index Terms—Table-based methods, reciprocal, square root, elementary functions, minimax polynomial approximation, single-precision computations, computer arithmetic.

1 Introduction

The increasing speed and performance constraints of computer 3D graphics, animation, digital signal processing (DSP), computer-assisted design (CAD), and virtual reality [7], [12], [14], [26], have led to the development of hardware-oriented methods for high-speed function approximation. An accurate and fast computation of division, square root, and, in some cases, exponential, logarithm, and trigonometric functions has therefore become mandatory in platforms such as graphics processing units (GPUs), digital signal processors (DSPs), floating-point units of general-purpose processors (FPUs), or application-specific circuits (ASICs) [18], [19], [27], [39].

The method presented in this paper is a table-driven algorithm based on an enhanced minimax quadratic approximation which allows such high-speed computations in single-precision (SP) floating-point format. High-fidelity audio and high-quality 3D graphics or speech recognition, among other applications, require the use of single-precision FP computations [7]. Thus, while 10-16 bit approximations were accurate enough in early graphics cards, with the evolution of graphics applications, higher-precision computations have become mandatory.

For instance, a frequent operation in *3D rendering* is the interpolation of attribute values across a primitive, colors and texture coordinates being common attributes requiring such interpolation. To obtain perspective correct results, it is necessary to interpolate a function of the attributes, rather than the attributes directly, and the interpolated result is later transformed by the inverse function to get the final value at a desired point in screen space [11]. Thus, the attributes at the primitive's vertices must first be divided through by the homogeneous coordinate w (the coordinate w itself is replaced by $1/w$) and these modified attributes are then linearly interpolated to determine the correct value at a given pixel within the primitive. At each pixel, the newly interpolated value is then divided by the interpolated $1/w$, via multiplication by w, to obtain the correct final value. This set of operations requires that, *per-pixel*, the newly interpolated inverse homogeneous coordinate $1/w$ must be reciprocated to reform w. The precision of this reciprocal operation must be *as close as possible to the working precision of the attributes being interpolated* to guarantee correct results. In the case of per-pixel texture coordinate interpolation, the coordinates in modern GPUs are computed and stored as single-precision floating-point values. The reciprocal operation itself, therefore, must return a value close to the exactly-rounded single-precision floating-point result. In practice, an error in the computation near 1 ulp for single-precision is sufficient, while errors greater than this can result in visible seams between textured-primitives.

Other transcendental functions are frequently used in modern vertex and pixel shading programs. Normalizing vectors using the reciprocal square root operator is a common function in lighting shaders. The instruction sets

- *J.-A. Piñeiro is with the Intel Barcelona Research Center, Intel Labs-UPC, Edif. Nexus II, c) Jordi Girona 29-3°, 08034 Barcelona, Spain. E-mail: alex@dec.usc.es.*
- *S.F. Oberman is with NVIDIA Corporation, USA, Santa Clara, CA 95050. E-mail: soberman@nvidia.com.*
- *J.-M. Muller is with CNRS, LIP, Ecole Normale Superieure de Lyon, 46 Allee d'Italie, 69364 Lyon Cedex 07, France. E-mail: Jean-Michel.Muller@inria.fr.*
- *J.D. Bruguera is with the Departamento de Electronica e Computacion, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain. E-mail: bruguera@dec.usc.es.*

Manuscript received 27 Nov. 2003; revised 30 June 2004; accepted 16 Sept. 2004; published online 18 Jan. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0230-1103.

of modern GPUs and APIs, such as Microsoft's DirectX9 [22], also incorporate $\exp 2$, $\log 2$, $\sin$, and $\cos$. These operations are important to many classes of shading programs and they are all required to produce results as close as possible to the processor's working precision.

The proposed method is suitable for approximating any function $f(X)$ and, therefore, can be applied to the computation of reciprocal, square root, square root reciprocal, logarithms, exponentials, trigonometric functions, powering[1] (with a fixed exponent p), or special functions. Furthermore, this method can be used for fixed-point DSP applications with a target precision of 16 to 32 bits as well or to obtain initial approximations (*seed values*) for higher precision computations (as done in [30] for double-precision computation of reciprocal, division, square root, and square root reciprocal).

The input operand X is split into two parts, the most and least significant fields: $X = X_1 + X_2$, X_1 being 6-7 bits wide, and the function $f(X)$ is approximated as a quadratic polynomial $C_2X_2^2 + C_1X_2 + C_0$. For each input subinterval, the coefficients of the minimax approximation C_2, C_1, and C_0, which only depend on X_1, are obtained through an iterative process that takes into account the effect of rounding such coefficients to a finite wordlength and are then stored in look-up tables. The evaluation of the polynomial is performed using redundant arithmetic and multioperand addition, while X_2^2 is generated by a specialized squaring unit with reduced size and delay regarding a standard squarer. The multioperand addition is carried out by a *fused accumulation tree*, which accumulates the partial products of $C_2X_2^2$ and C_1X_2, together with C_0, and requires less area than a $(n \times n)$-bit standard multiplier while having about the same delay. The rounding scheme is an enhancement of rounding to the nearest and consists of injecting a function-specific rounding-bias before truncating the intermediate result to the target precision. The output results can be guaranteed accurate to 1 ulp, 2 ulps, or 4 ulps, depending on the accuracy constraints of a specific application for any of the functions $f(X)$ considered.

The organization of this paper is the following: Background information about hardware-oriented methods is given in Section 2; the main features of our method, error computation, iterative method for selecting the polynomial coefficients, and efficient polynomial evaluation, are described in Section 3; an architecture for the computation of reciprocal, square root, exponential, and logarithm, together with delay and area cost estimates, is shown in Section 4; in Section 5, a comparison with bipartite tables methods and linear and quadratic approximations is presented; finally, the main contributions made in this work are summarized in Section 6.

2 Background

Hardware-oriented methods for function approximation can be separated into two main groups: iterative and noniterative methods. In the first group belong digit-recurrence and online algorithms [8], [9], based on the subtraction operation and with linear convergence, and hardware implementations of functional iteration methods, such as Newton-Raphson and Goldschmidt algorithms [10], [27], which are multiplicative-based and quadratically convergent. This type of method can be used for both low-precision and high-precision computations. On the other hand, we have direct table look-up, polynomial and rational approximations [17], [23], and table-based methods [2], [5], [6], [15], [34], [40], [42], suitable only for low-precision operations and usually employed for computing the *seed values* in functional iteration methods.

When aiming at single-precision (SP) computations, table-based methods have inherent advantages over other types of algorithms. First of all, they are halfway between direct table look-up, whose enormous memory requirements make them inefficient for SP computations, and polynomial/rational approximations, which involve a high number of multiplications and additions, resulting in long execution times. Table-driven methods combine the use of smaller tables with the evaluation of a low-degree polynomial, achieving both a reduction in size regarding direct table look-up and significant speed-ups regarding pure polynomial approximations. The fact of being noniterative makes table-based algorithms also preferable regarding digit-recurrence methods, due to the linear convergence of these conventional *shift-and-add* implementations, which usually leads to long execution times. Something similar happens to functional iteration methods, despite being quadratically convergent, since, for each iteration, a number of multiplications are involved, resulting again in long execution times. Moreover, functional iteration methods are oriented to division and square root computations only and do not allow computation of logarithm, exponential or trigonometric functions.

Table-based methods can be further subdivided, depending on the size of the tables employed and the amount of computations involved, into *compute-bound methods* [41], [42], *table-bound methods* [6], [24], [34], [35], [38], and *in-between methods* [1], [2], [5], [15], [40].

- *Compute-bound methods* use table look-up in a very-small table to obtain parameters which are used afterward in cubic or higher-degree polynomial approximations [41], [42]. The polynomial is usually evaluated using Horner's rule and an important amount of additions and multiplications is involved. This type of method is favored with the availability of a fused multiply-add unit, as happens in architectures such as PowerPC and Intel IA64 (now IPF) [7], [20], [21].
- *Table-bound methods* use large tables and just one or possibly a few additions. Examples of this type of method are *Partial Product Arrays* [38] (PPAs) and bipartite tables methods. Among the bipartite tables methods there are those consisting of just two-tables and addition, such as bipartite table methods (BTM) [6] and symmetric BTM (SBTM) [34], and their generalizations, multipartite table methods [24], [35], which employ more than two tables and a few (possibly redundant) additions. These methods are

1. Note that the computation of powering (X^p) implies the computation of reciprocal (X^{-1}), square root ($X^{1/2}$), square root reciprocal ($X^{-1/2}$), reciprocal square (X^{-2}), and cube root ($X^{1/3}$) among other functions, just by adjusting the parameter p to the desired value.

fast, but their use is limited to computations accurate up to 16-bits (maybe 20-bits) with current VLSI technology, due to the growth in the size of the tables with the accuracy of the result.[2]

- *In-between methods* use medium size tables and a significant yet reduced amount of computation (e.g., one or two multiplications or several small/rectangular multiplications). This type of method can be further subdivided into linear approximations [5], [40] and second-degree interpolation methods [1], [2], [15], [31], [36], depending on the degree of the polynomial approximation employed. The intermediate size of the look-up tables employed makes them suitable for performing single-precision computations, achieving fast execution times with reasonable hardware requirements.

As explained above, the best alternative when aiming at SP computations are *in-between methods* (*compute-bound methods* may be preferable if the area constraints of a specific application are really tight). Within this group, the main advantage of linear approximations is their speed since they consist of a table look-up and the computation of a multiply-add operation [5] (or a single multiplication, if a slight modification of the input operand is performed [40]), while the main advantage of quadratic approximations is the reduced size of the look-up tables (around 12-$15Kbits$ per function [1], [2], [15], [36] versus 50-$75Kb$ [5], [40] in linear approximations).

The synthesis of the main advantages of both linear and conventional quadratic approximations can be reached by using the small tables of a *quadratic* interpolator and by *emulating* the behavior of linear approximations, computing the second-degree polynomial with a similar delay to that of an SP multiply operation. Such acceleration can be achieved [29], [31], [36] by combining the use of redundant arithmetic, a specialized squaring unit, and the accumulation of all partial products in a multioperand adder.

3 Minimax Quadratic Interpolator

In this section, we propose a table-based method for high-speed function approximation in single-precision floating-point format. We consider, in this paper, the computation of reciprocal, square root, square root reciprocal, exponential (2^X), logarithm ($\log_2 X$), and sine/cosine operations, although the proposed method could be used for the computation of other trigonometric functions ($\tan$, $\arctan$, ...), powering (with a fixed exponent p), or special functions. The results can be guaranteed accurate to 1 ulp, 2 ulps, or 4 ulps for any of these operations, depending on the accuracy constraints of a specific application. This algorithm is a generalization and optimization of those previously proposed by Piñeiro et al. [29], [31] and by Muller [25].

3.1 Range Reduction and Notation

Three steps are usually carried out when approximating a function [17], [42]: 1) range reduction of the argument to a predetermined input interval, 2) function approximation of the reduced argument, and 3) reconstruction, normalization, and rounding of the final result. Thus, the approximation is performed in an input interval $[a, b)$ and range reduction is used for values outside this interval. For the elementary functions there are natural selections of the intervals $[a, b)$ which simplify both the steps of range reduction and function approximation, leading to smaller tables and faster execution times, and also allow avoidance of singular points in the function domain [4], [13], [36].

Reciprocal:
$\frac{1}{M_x} = (-1)^{s_x} \frac{1}{X} 2^{-E_x}$, $X \in (1, 2)$

Square Root:
$\sqrt{M_x} = \sqrt{X}\, 2^{E_x/2}$, $X \in (1, 2)$ if E_x even, or
$\sqrt{M_x} = \sqrt{2X}\, 2^{(E_x-1)/2}$, $X \in (2, 4)$ if E_x odd

Reciprocal Square Root:
$\frac{1}{\sqrt{M_x}} = \frac{1}{\sqrt{X}} 2^{-E_x/2}$, $X \in (1, 2)$ if E_x even, or
$\frac{1}{\sqrt{M_x}} = \frac{1}{\sqrt{2X}} 2^{-(E_x-1)/2}$, $X \in (2, 4)$ if E_x odd

Exponential:
$2^{M_x} = 2^{I+F} = 2^I 2^F$, $F \in (0, 1)$
after denormalizing M_x

Logarithm:
$\log_2(M_x) = \log_2(X) + E_x$, $X \in (1, 2)$ if $E_x \neq 0$, or
$\log_2(M_x) = (X-1)[\log_2(X)/(X-1)]$, $X \in (1, 2)$ if $E_x = 0$

Sine/Cos:
$\sin(M_x) = \sin(X)$,
after mapping angle M_x to $X \in [0, 1)$

Fig. 1. Range reduction schemes and approximation intervals for some functions.

Our method deals with the step of function approximation within a reduced input interval and standard techniques are applied for the range reduction and reconstruction steps. A summary of these well-known range reduction schemes is shown in Fig. 1, where the following notation is used: An IEEE single-precision floating-point number M_x is composed of a sign bit s_x, an 8-bit biased exponent E_x, and a 24-bit significand X, and represents the value:

$$M_x = (-1)^{s_x} X 2^{E_x}, \tag{1}$$

where $1 \leq X < 2$, with an implicit leading 1 (only the fractional 23-bits are stored). Leading zeros may appear when approximating the logarithm if $E_x = 0$, causing a loss of precision. In that case, $1 \leq \log_2(X)/(X-1) < 2$ is computed instead since it eliminates the leading zeros and the result is later multiplied by $(X-1)$, processing the exponent of the result to account for this normalization [36].

3.2 Overview of the Method

As explained in Section 2, a thorough analysis of the design space of hardware-oriented methods for function approximation shows that, when aiming at single-precision computations, the best trade off between area and speed is obtained by using a quadratic interpolator, provided that the degree-2 polynomial is efficiently computed [29], [31],

2. The size of the tables required for SP computations with the SBTM method is over $1,300Kbits$ per function [34].

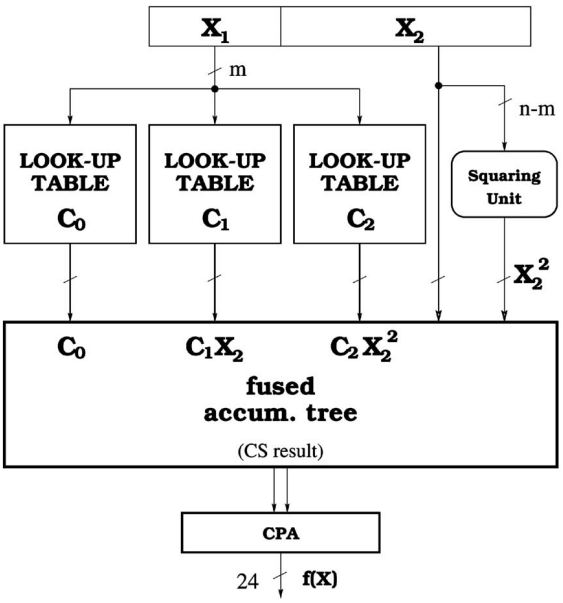

Fig. 2. Block diagram of the proposed table-based method.

[36]. Therefore, our method for high-speed function approximation consists of a quadratic polynomial approximation, followed by a fast evaluation of the polynomial. A specialized squaring unit, redundant arithmetic, and a multioperand adder are employed to carry out with such evaluation. On the other hand, the high accuracy of an *enhanced minimax approximation*, which takes into account the effect of rounding the coefficients to a finite wordlength, allows for a significant reduction in the size of the tables required to store the polynomial coefficients regarding similar quadratic interpolators.

As shown in Fig. 2, in our method, the n-bit binary significand of the input operand X is split into an upper part, X_1, and a lower part, X_2:

$$\begin{aligned} X_1 &= [.x_1 x_2 \dots x_m] \\ X_2 &= [.x_{m+1} \dots x_n] \times 2^{-m}. \end{aligned} \tag{2}$$

An approximation to $f(X)$ in the range $X_1 \le X < X_1 + 2^{-m}$ can be performed by evaluating the expression

$$f(X) \approx C_0 + C_1 X_2 + C_2 X_2^2, \tag{3}$$

where the coefficients C_0, C_1, and C_2 are obtained for each input subinterval using the computer algebra system **Maple** [43]. The method for obtaining those coefficients, corresponding to an *enhanced minimax approximation*, will be explained in Section 3.4 and is one of the fundamental contributions of this work.

The values of C_0, C_1, and C_2 for each function $f(X)$ depend only on X_1, the m most significant bits of X. Therefore, these polynomial coefficients can be stored in look-up tables of 2^m entries.[3] The exact value of m depends on the function to be computed and on the target precision. For instance, it will be shown in Section 3.4 that, for approximating the square root and exponential, with an accuracy of 1 ulp in a single-precision format, $m = 6$ suffices, while $m = 7$ is necessary for computing the reciprocal, logarithm, and reciprocal square root. Note that minimizing m is crucial to the area requirements of an interpolator since a high-accuracy approximation that allows for using $m' = m - 1$ would require look-up tables with 2^{m-1} entries, that is, half the number of entries in the table, and, therefore, tables with *roughly half the size* of the original ones.

The main features of our method can be summarized as:

- An enhanced minimax approximation is computed to obtain the sets of coefficients of a quadratic interpolation for the considered function, within the error bounds set by a specific application. Such an approximation consists of 3-passes of the minimax approximation in order to compensate for the rounding errors introduced by having finite-size coefficients. This enhanced minimax approximation is performed with the computer algebra system Maple.
  - The value of m is minimized to allow for a significant reduction in the size of the look-up tables storing the coefficients.
  - The wordlengths of the polynomial coefficients are also minimized, after m has been set, to allow for a further reduction in the table size and also in the size of the logic blocks to be used in the polynomial evaluation.
  - The range of the coefficients is noted in order to safely remove, if possible, some initial bits from the stored coefficients. These bits are treated as *implicit bits* and can be concatenated at the output of the look-up tables, allowing for a further slight reduction in their size.
- A high-speed evaluation of the degree-2 polynomial is carried out in order to obtain execution times similar to those of linear interpolation methods, which only require the computation of one multiplication (and, possibly, one addition).
  - A specialized squaring unit is used for generating X_2^2. Since this value is used as a multiplier operand in the generation of the partial products of $C_2X_2^2$, its assimilation to nonredundant representation (which would increase the overall delay of the polynomial evaluation) is avoided by using CS to SD recoding instead.
  - SD radix-4 recoding of the multipliers is used in the generation of the partial products of $C_2X_2^2$ and C_1X_2, which allows the reduction by half of the number of partial products to be accumulated. The alternative of using SD radix-8 recoding instead is discussed in Section 3.5.
  - The accumulation of all partial products of $C_2X_2^2$ and C_1X_2, together with the degree-0 term C_0, is carried out by a multioperand adder.
  - Assimilation into nonredundant representation is performed *just once*, by a fast adder (a CPA), at the output of the fused accumulation tree. If

3. In some cases, a bigger range is covered for ease of implementation and some extra bit(s) may be necessary for addressing the tables. For instance, when approximating the square root or square root reciprocal, the least significant bit of the exponent is employed to select between the tables covering the intervals $(1, 2)$ and $(2, 4)$.

two multiplications and two additions were performed, several sequential assimilations would be necessary, slowing down the polynomial computation significantly.

- Rounding is performed by injecting a *function-specific rounding bias* before performing truncation of the result to the target precision. This is an enhancement of rounding to the nearest since it subsumes the former, but also allows for reducing the maximum absolute error for each specific function and compensating for the effect of truncation errors in the polynomial evaluation, such as using finite wordlength in the computation of X_2^2. This rounding scheme is not suitable for higher-precision computations since exhaustive simulation is required for the choice of the rounding bias.

The results of the function approximation obtained with our quadratic interpolator can be guaranteed accurate to 1 ulp, which allows faithful rounding for the computation of reciprocal, square root, square root reciprocal, and exponential.[4] For many applications, guaranteeing faithful rounding suffices and speeds up the computations, while significantly reducing the hardware requirements of the circuit. For instance, it is a common practice in computer graphics applications not to perform exact rounding since the inherent loss of precision does not lead to degradation in the quality of the results, usually employed only for displaying purposes [14], [18].

3.3 Error Computation

The total error in the final result of the function approximation step can be expressed as the accumulation of the error in the result before rounding, ε_{interm}, and the rounding error, ε_{round}:

$$\varepsilon_{total} = \varepsilon_{interm} + \varepsilon_{round} < 2^{-r}, \tag{4}$$

where r depends on the input and output ranges of the function to be approximated and on the target accuracy and defines a specific bound on the final error.

The error on the intermediate result comes from two sources: the error in the minimax quadratic approximation itself, ε_{approx}, and the error due to the use of finite arithmetic in the evaluation of the degree-2 polynomial:

$$\varepsilon_{interm} \leq \varepsilon_{approx} + \varepsilon_{C_0} + \varepsilon_{C_1} X_2 + \varepsilon_{C_2} X_2^2 + |C_1|\varepsilon_{X_2} + |C_2|\varepsilon_{X_2^2}. \tag{5}$$

The error of the approximation, ε_{approx}, depends on the value of m and on the function to be interpolated. As pointed out in Section 3.2, the minimum value of m compatible with the error constraints must be used in order to allow for reduced size look-up tables storing the polynomial coefficients.

The Maple program we use for obtaining the coefficients (see Section 3.4) gives the contribution to the intermediate error of the minimax approximation performed with *rounded* coefficients and, therefore, we can define:

$$\varepsilon'_{approx} = \varepsilon_{approx} + \varepsilon_{C_0} + \varepsilon_{C_1} X_2 + \varepsilon_{C_2} X_2^2 \tag{6}$$

and, since $\varepsilon_{X_2} = 0$, in this case:

$$\varepsilon_{interm} \leq \varepsilon'_{approx} + \varepsilon_{squaring}, \tag{7}$$

with $\varepsilon_{squaring} = |C_2|\varepsilon_{X_2^2}$.

On the other hand, ε_{round} depends on how the rounding is carried out. Conventional rounding schemes are *truncation* and *rounding to the nearest*. If using truncation of the intermediate result at position 2^{-r}, the associated error would be bounded by $\varepsilon_{round} \leq 2^{-r}$, while, if performing rounding to the nearest by adding a one at position 2^{-r-1} before such truncation, the rounding error would be bounded by 2^{-r-1} instead.

The rounding scheme employed in our minimax quadratic interpolator is an enhancement of rounding to the nearest, made possible by exhaustive simulation, and consists of adding a function-specific *rounding bias* before performing the truncation of the intermediate result at position 2^{-r}. A judicious choice of the *bias*, based on the error distribution of the intermediate result for each specific function, allows reduction of the maximum absolute error in the final result while compensating for the truncation carried out in the squaring unit. The injection of such a *bias* constant can be easily performed within the accumulation tree, as will be explained in Section 3.5.

Summarizing, the total error in the final result can be expressed as:

$$\varepsilon_{total} = \varepsilon'_{approx} + |C_2|\varepsilon_{X_2^2} \pm \varepsilon_{round} < 2^{-r}, \tag{8}$$

with $\pm\varepsilon_{round}$ instead of $+\varepsilon_{round}$ due to the ability of our rounding scheme to compensate in some cases for the truncation errors in both the squaring computation and the enhanced minimax approximation.

3.4 Enhanced Minimax Approximation

In this section, we describe the algorithm for obtaining the coefficients of the quadratic polynomial to be stored in the look-up tables. This algorithm consists of three passes of the minimax approximation in order to compensate for the errors introduced by rounding each of the polynomial coefficients to a finite precision. More information about the minimax approximation can be found in [23], [25], [29], [31], [33].

Fig. 3 describes the heuristic employed to obtain the minimum value of m which guarantees results accurate to the target precision, the set of coefficients C_0, C_1, C_2 with minimum wordlengths (t, p, q), and the function-specific rounding bias. As *Step 2* in our heuristic, a 3-passes hybrid algorithm is computed by using Maple, as shown in Fig. 4. This hybrid algorithm will be explained in detail below. As *Step 3*, exhaustive simulation across the approximation interval is performed using a C program which describes the hardware functionality of our quadratic interpolator. Note that exhaustive simulation is possible due to the fact of dealing with single-precision computations.

When the configuration which leads to a lower table size has been obtained (*optimal configuration*), the binary representation of the coefficients is generated to be used in the synthesis process. All coefficients are kept in fixed-point form and the range of the coefficients is noted in order

4. Faithful rounding means that the returned value is guaranteed to be one of the two floating-point numbers that surround the exact value and such a condition leads, in many cases, to producing the exactly rounded result.

Step1:

The minimax quadratic approximation of the considered function with coefficients of unrestricted length is computed for the target interval, for different values of m.
Select the minimum value of m for which it holds: $\epsilon_{approx} < 2^{-r-1} + \delta$.

Step 2:

For such value of m, perform a 3-passes hybrid algorithm (see Figure 4) with large values for the parameters t, p, q (wordlengths of polynomial coefficients C_0, C_1 and C_2, respectively), for instance $t = 40$, $p = 30$, $q = 20$.

if $\epsilon'_{approx} \geq 2^{-r-1} + \delta$, do $m' = m + 1$ and go back to **Step 2**.

else proceed decreasing the values of t, p, q and **iterate** evaluating the 3-passes minimax algorithm, until obtaining the sets of coefficients for which the error bound is met, which minimize the value $(t + p + q)$.

Step 3:

Once a candidate set of coefficients has been obtained, exhaustive simulation across the target interval is used to compute the best *rounding constant* which can be injected before truncation to minimize the absolute error of the result. The following constraint must be met: $\epsilon_{total} < 2^{-r}$.

if there is no rounding constant which guarantees achieving the target accuracy, select the second candidate set of coefficients and go back to **Step 3**.

else the optimal configuration of coefficients has been found: the iterative process is finished.

Fig. 3. Heuristic for obtaining sets of coefficients.

to safely remove some bits from the stored coefficients (to be dealt with as implicit initial bits, which are later concatenated at the output of the tables), allowing a slight further reduction in the table sizes.

The parameter δ has been introduced in the heuristic described in Fig. 3 in order to be aggressive in the minimization of m and (t, p, q) since it prevents early discarding of configurations which might be allowed later in combination with a specific rounding bias. A value of δ which allows obtaining a good degree of optimization is $\delta = 2^{-r-2}$.

Note that, at the end of *Step 2* in Fig. 3, there may be several combinations of (t, p, q) which are candidates to be the optimal configuration. In such a case, priority could be given to those combinations which minimize $(p + q)$, allowing a higher value for t since the coefficient C_0 is not involved in any partial product generation and, therefore, it has a slightly lower impact on the size of the structures used within the accumulator tree. Alternatively, the bit-patterns of the coefficients for each configuration (in binary representation) can be analyzed to see if there is any configuration which allows treating a higher number of bits as implicit, thus saving some extra bits of storage.

```
computecoeffts := proc(t, p, q)
errmax:=0;
for i from 0 to 63 do
    pol1 := minimax(sqrt(1 + 1/64*i + x), x = 0 .
        .1/64, [2, 0], 1, 'err');
    C1 := 2^(-p)*round(coeff(pol1, x)*2^(p));
    a1 := coeff(pol1,x);
    a2 := coeff(pol1,x^2);
    aa2 := a2 + (a1-C1)*2^(6);
    C2 := 2^(-q)*round(aa2*2^(q));
    p0 := minimax(sqrt(1+1/64*i+x)-C1*x-C2*x^2,
        x = 0 .. 1/64, [0, 0], 1, 'err');
    C0 := round(2^(t)*( tcoeff(p0)) )*2^(-t);
    err := infnorm(sqrt(1+1/64*i+x)-C0-C1*x-
        C2*x^2, x=0..1/64);
    if errmax < err then errmax:=err fi;
od;
goodbits:=abs(ln(errmax))/ln(2.0);
print(goodbits,errmax);
end;
```

Fig. 4. Maple program for obtaining the coefficients (square root, $m = 6$).

An interesting artifact can occur for some configurations with specific functions: All values of a coefficient share some initial common bits (this may include the sign bit as well) except the one corresponding to the first subinterval. In such a case, the conflictive value can be speculatively substituted by the closest binary number sharing those initial common bits and the exhaustive simulation must be performed again, assuming this new value, to check whether the maximum absolute error has not increased. This trick allows the saving of extra storage of implicit bits in the look-up tables without affecting the accuracy of the approximation. In order to illustrate this artifact, the implicit bits for the optimal configurations obtained with 1 ulp accuracy for some elementary functions are shown in Table 1. It happens that the value of the coefficient C_0 in the first subinterval for $\sin(X)$, with $m = 6$ and configuration $(27, 18, 13)$, is $-2.235 \cdot 10^{-8}$, while all other values of C_0 are positive and belong to the interval $[0, 1)$. Keeping such a value for the first interval would make storage of both the sign and the integer bits for all C_0 coefficients in the corresponding look-up table necessary. However, this value can be safely substituted by 0 and, since the maximum error of the approximation does not belong in that first subinterval, a pattern $+0.xxxxx$ can be assumed for C_0 in that case. Thus, the initial $+0.$ need not be stored since it can be

TABLE 1
Implicit Bits in the Polynomial Coefficients for Some Functions (1 ulp)

Function	C_0	C_1	C_2
$1/X$	$+0.1xxxx\ldots xx$	$-0.xxxxx\ldots xx$	$+0.xxxxx\ldots xx$
$\sqrt{X} \in (1,2)$	$+1.0xxxx\ldots xx$	$+0.01xxx\ldots xx$	$-0.000xx\ldots xx$
$\sqrt{2X} \in (2,4)$	$+1.xxxxx\ldots xx$	$+0.10xxx\ldots xx$	$-0.00xxx\ldots xx$
$1/\sqrt{X} \in (1,2)$	$+0.1xxxx\ldots xx$	$-0.0xxxx\ldots xx$	$+0.0xxxx\ldots xx$
$1/\sqrt{2X} \in (2,4)$	$+0.10xxx\ldots xx$	$-0.0xxxx\ldots xx$	$+0.0xxxx\ldots xx$
2^X	$+1.xxxxx\ldots xx$	$+x.xxxxx\ldots xx$	$+0.0xxxx\ldots xx$
$\log_2(X)$	$+0.xxxxx\ldots xx$	$+x.xxxxx\ldots xx$	$-0.xxxxx\ldots xx$
$\sin(X)$	$+0.xxxxx\ldots xx$	$+x.xxxxx\ldots xx$	$-0.0xxxx\ldots xx$

concatenated at the output of the table with the stored $.xxxxx$ bits.

3.4.1 Three-Passes Hybrid Algorithm

A minimax approximation with polynomial coefficients (a_0, a_1, a_2) of unrestricted wordlength yields very accurate results. However, the coefficients must be rounded to a finite size to be stored in look-up tables and such rounding may significantly affect the precision of the original approximation. Let us illustrate this with an example: Consider the computation of the function $1/\sqrt{X}$ on the interval $(1,2)$, for a random value of m, say 8. At address i in the table, we will find the coefficients of the approximation for the interval $[1+i/256, 1+(i+1)/256)$. Let us see how to compute coefficients for $i = 37$.

The Maple input line

```
> minimax(1/sqrt(1+37/256+x),
x=0..1/256,[2,0],1,'err');
```

asks for the minimax approximation in that domain (the variable x represents X_2, the lower part of the input operand X). We get the approximation

```
> 0.93472998018 +
(- 0.40834453917 + 0.26644775593 x ) x,
```

with an error $\varepsilon_{approx} = 3.61 \times 10^{-9}$. However, if the linear coefficient a_1 is rounded to $p = 14$ bits to become C_1 and the quadratic coefficient a_2 is rounded to $q = 6$ bits to form C_2, the approximation error using C_1 and C_2 turns out to be $\varepsilon'_{approx} = 5.58 \times 10^{-8}$, which is much larger than ε_{approx}, the error of the approximation using a_1 and a_2.

A second pass of the minimax approximation allows for the computation of the best approximation among the polynomials with p-bit C_1 coefficients (this new polynomial has coefficients a'_0 and a'_2). After rounding the newly obtained coefficient a'_2 to $q = 6$ bits to form C_2, the approximation error has been reduced to $\varepsilon'_{approx} = 5.01 \times 10^{-8}$.

A third pass of the minimax approximation is necessary to take into account the effect of this rounding as well. The idea is now obtaining the best polynomial among those polynomials whose order-1 coefficient is C_1 and whose order-2 coefficient is C_2. This is done as follows:

```
> minimax(1/sqrt(1+37/256+x)-C1*x -
C2*x^2,x=0..1/256,[0,0],1,'err');
```

By doing this, a coefficient $a''_0 = 0.934730008279251$ is obtained. The error of this final approximation is $\varepsilon'_{approx} = 2.77 \times 10^{-8}$, which is half the error of the original approximation with a single pass of minimax, when the effect of rounding the coefficients to finite wordlengths was not taken into account. Finally, in both cases, the degree-0 coefficient must also be rounded to t bits to be stored in the look-up table corresponding to C_0.

In Piñeiro et al. [29], [31], an algorithm was proposed consisting of performing these 3-passes of the minimax approximation. However, some numerical problems arose in such an algorithm when considering specific configurations and/or specific functions, originated by the minimax approximation performed as second step.[5] In such a step, an approximation $a'_0 + a'_2\lambda^2$ to $(a_1 - C_1)\lambda$ is computed, which, defining $L = \lambda^2$, is equivalent to performing an order-1 approximation to $(a_1 - C_1)\sqrt{L}$. The conclusion is that certain functions are not readily amenable to being approximated by automated methods since the function needs to be expressable in a particularly simple form.

Those numerical problems referred to were later addressed by Muller [25] by showing that the degree-1 minimax approximation to $\sqrt{L}$ in the interval $[0, 2^{-2d}]$ corresponds to

$$2^{-d-3} + 2^d L, \tag{9}$$

with an error in the approximation of 2^{-d-3}, which makes available an analytical expression for performing the second pass. The algorithm proposed in [25], however, does not provide a higher overall accuracy since it performs only 2-passes of the minimax approximation instead of three.

Summarizing, we propose here a hybrid algorithm consisting of 3-passes of the minimax approximation, with the second pass performed by making use of the analytical expressions introduced in [25]:

1. Using minimax to find the original approximation, with nontruncated coefficients, and rounding the degree-1 coefficient to p bits to obtain C_1.

5. For instance, when trying to compute an approximation to the reciprocal with $m = 7$ and a configuration $(26, 16, 10)$ or to the logarithm with $m = 7$ and $(26, 15, 10)$, Maple gives the following error message: *Error, (in numapprox/remez) error curve fails to oscillate sufficiently; try different degrees.*

TABLE 2
Summary of Design Parameters for Some Functions (1 ulp)

Function	ϵ_{total}	m	*configuration*	accuracy	*Look–Up Table size*
$1/X$	$< 2^{-24}$	7	26, 16, 10	24.02	$2^7 \times (25+16+10) = 6.375Kb$
$\sqrt{X}$	$< 2^{-23}$	6	25, 15, 11	23.04	$2 \times 2^6 \times (25+15+9) = 6.125Kb$
$1/\sqrt{X}$	$< 2^{-24}$	7	26, 16, 10	24.15	$2 \times 2^7 \times (25+15+9) = 12.25Kb$
2^X	$< 2^{-23}$	6	25, 15, 11	23.02	$2^6 \times (25+16+10) = 3.1875Kb$
$\log_2 X$	$< 2^{-24}$	7	26, 15, 10	24.13	$2^7 \times (26+16+10) = 6.5Kb$
$\sin(X)$	$< 2^{-24}$	6	27, 18, 13	24.01	$2^6 \times (27+19+12) = 3.625Kb$

TABLE 3
Size of the Tables to Be Employed for Target Accuracies of 2 ulps and 4 ulps

Function	*Look–Up Table size (2 ulps)*	*Look–Up Table size (4 ulps)*
$1/X$	$2^7 \times (24+15+9) = 6Kb$	$2^6 \times (25+17+12) = 3.375Kb$
$\sqrt{X}$	$2 \times 2^5 \times (26+17+11) = 3.375Kb$	$2 \times 2^5 \times (23+14+9) = 2.875Kb$
$1/\sqrt{X}$	$2 \times 2^6 \times (24+14+11) = 6.125Kb$	$2 \times 2^6 \times (23+13+9) = 5.625Kb$
2^X	$2^5 \times (25+18+13) = 1.75Kb$	$2^5 \times (24+15+10) = 1.532Kb$
$\log_2 X$	$2^6 \times (27+19+14) = 3.75Kb$	$2^6 \times (24+15+10) = 3.063Kb$
$\sin(X)$	$2^6 \times (25+17+10) = 3.25Kb$	$2^6 \times (25+15+9) = 3.063Kb$

2. Computing a_2' using the analytical expression

$$a_2' := a_2 + (a_1 - C_1) \times 2^m, \tag{10}$$

where a_1, a_2 are the degree-1 and degree-2 coefficients of the original approximation, and rounding a_2' to q bits to obtain C_2.
3. Using minimax to compute the degree-0 coefficient, based on C_1 and C_2 above, and then rounding it to t bits to obtain C_0.

Fig. 4 shows a Maple program which implements our method, for the computation of square root, with $m = 6$, within the interval $(1, 2)$. The number of correct bits are obtained as *goodbits* and the error of the approximation, ε'_{approx}, is shown as *errmax*.

3.4.2 Table Sizes Obtained with Our Algorithm

Table 2 shows a summary of the parameters to be employed for the approximation of some elementary functions with our method, with an accuracy of 1 ulp. The error bounds in the prenormalized result are shown, together with the minimum values of m for each function, the number of fractional bits of the polynomial coefficients in the optimal configurations, the accuracies yielded by such optimal configurations, and the corresponding look-up table sizes for each function.

Remember that the number of fractional bits of the coefficients (*configuration*) is not coincident in all cases with the number of stored bits since the ranges of some coefficients allow removal of some initial bits, which are left as *implicit bits*, while, in other cases, it is necessary to store an integer bit as well, as shown in Table 1.

Smaller tables could be employed for approximations with looser accuracy constraints, as shown in Table 3 for the cases of 2 ulps and 4 ulps. Note that, when a decrement by one in the value of parameter m is made possible by lower accuracy constraints, a reduction by a factor of about 2 can be achieved in the size of the look-up tables. This happens, for instance, for reciprocal computations when going from 2 ulps to 4 ulps and for exponential (2^X) or reciprocal square root approximation when moving from 1 ulp to 2 ulps.

3.5 Fast Evaluation of the Quadratic Polynomial

As explained in Section 2, many quadratic interpolators [1], [2], [15] have, as a main drawback, their long execution times, in spite of the advantage of using small tables for storing the polynomial coefficients. In order to emulate the behavior of linear approximations, the polynomial evaluation must be carried out in an efficient manner. For that purpose, we use a specialized squaring unit, redundant arithmetic, and multioperand addition, as shown in Fig. 2.

3.5.1 Specialized Squaring Unit

Rather than having a multiplier to compute X_2^2, two techniques can be used to design a unit with significantly reduced area and delay [16], [37]: rearranging the partial product matrix and considering the leading zeros of X_2. The former is well-known and we refer the reader to [3], [29], [36] for more information. In the latter, advantage is taken of the fact that $X_2 = [.x_{m+1} \dots x_n] \times 2^{-m}$ (i.e., X_2 has m leading zeros) and, therefore, X_2^2 will have at least $2m$ leading zeros. Truncation can be employed in the partial product matrix of the squaring unit under a certain error bound, which is dependent on the function to be computed. When $m = 6$, truncating at position 2^{-28} results in a maximum error of $1.0 \cdot 2^{-25}$, while the maximum value of $\varepsilon_{X_2^2}$ is $0.94 \cdot 2^{-25}$ for $m = 7$ and does not affect the accuracy of the final result due to the injection of a function-specific bias in the rounding scheme. Truncation at position 2^{-28} when $m = 6$ results in a wordlength of 16 bits for X_2^2, while

TABLE 4
Use of Redundant Arithmetic in the Generation of the Partial Products

m	size X_2	size X_2^2(*)	pps X_2 (SD-4)	pps X_2^2 (SD-4)	pps X_2 (SD-8)	pps X_2^2 (SD-8)
6	17	16	9	8	6	6
7	16	14	8	7	6	5
8	15	12	8	6	5	4
9	14	10	7	5	5	4

$*$ truncating squaring at position 2^{-28}

14 bits must be kept for the square with $m = 7$. Only two levels of 4:2 adders, plus an initial *and* stage ($x_{ij} = x_i \cdot x_j$), are required to generate the CS representation of X_2^2, which is directly recoded to SD-4, to be used as multiplier operand in the fused accumulation tree.

When sharing the squaring unit for the approximation of several functions with different values of m, the squarer must be designed for the lowest m since all other cases are subsumed just by inserting leading zeros, as will be shown in Section 4.

3.5.2 Fused Accumulation Tree

Apart from the squaring evaluation, the other main problem to be overcome in order to guarantee high-speed function approximations is the computation of the quadratic polynomial once the values of C_0, C_1, and C_2 have been read from the look-up tables and X_2^2 has been calculated in parallel using the specialized squaring unit.

Following [36], we propose employing a unified tree to accumulate the partial products of both C_1X_2 and $C_2X_2^2$, plus the coefficient C_0. The use of redundant arithmetic can help to significantly reduce the number of partial products to be accumulated: SD radix-4 reduces such a number by half, while SD radix-8 may do it by a factor of 2/3.

The total number of partial products to be accumulated is the sum of the number of partial products of C_1X_2 (generated by X_2), the number of partial products of $C_2X_2^2$ (generated by X_2^2), plus 1 (the degree-0 coefficient C_0). Thus, if the total number of partial products goes from 10 to 12, a first level of 3:2 adders, plus two levels of 4:2 adders are necessary. If the range is 13 to 16 pps, three levels of 4:2 CSA adders are necessary and, for higher values, at least four levels of adders must be employed in the accumulation tree.

Table 4 shows different combinations of SD-4 and SD-8 representation for X_2 and X_2^2 which may be considered. When $m = 7$, the best alternative consists of using SD-4 for both X_2 and X_2^2 since their wordlengths of 16 and 14 bits, respectively, lead to the generation of $8 + 7 + 1 = 16$ partial products to be accumulated. No speed-up can be obtained from using SD-8 unless it is employed for both X_2 and X_2^2, which would increase the table sizes significantly since the multiples $(3\times)$ for C_1 and C_2 must be also stored in order not to increase the delay of the partial product generation scheme.

When $m = 6$, X_2 has 17 significant bits and X_2^2 has 12 leading zeros, with a wordlength of 16 bits if truncation at position 2^{-28} is performed in the specialized squaring unit. The first alternative, following [29], [31], consists of again using SD-4 representation for both X_2 and X_2^2, which results in the generation of nine partial products to compute C_1X_2 and eight partial products to compute $C_2X_2^2$. The number of operands to be accumulated is, therefore, 18 ($9 + 8 + 1$, the last one being the coefficient C_0) and, therefore, four levels of adders are required, but a minimum size for the look-up tables is guaranteed. The second alternative consists of using SD-8 representation for X_2^2, reducing the contribution of $C_2X_2^2$ from eight to six partial products, which leads to a total number of 16 pps to be accumulated and helps to reduce the delay of the accumulation tree by adjusting to just three levels of 4:2 adders. The selection between both alternatives depends on implementation and technology constraints.

In our method, the first alternative is employed because it allows a minimum table size, while the accumulation of partial products can be arranged so that the effective delay corresponds to only three levels of 4:2 adders. Note that, as will be shown in Section 4, the generation of the partial products of $C_2X_2^2$ usually takes longer than the generation of those corresponding to C_1X_2, due to the computation of the squaring and later recoding into SD-4 representation. Thus, a first level of three 3:2 CSA adders can reduce the nine partial products of C_1X_2 into six pps, in parallel with the recoding of X_2^2 and generation of $C_2X_2^2$, and, therefore, out of the critical path of the interpolator. Note also that, in the resulting accumulation tree, only the leftmost *carry-save* adders in levels second to fourth (see Fig. 6) must have full single-precision wordlength, while all other adders benefit from the reduced wordlength of the polynomial coefficients.

By using this strategy, the effective delay of the accumulation tree for $m = 6$ is exactly the same as that for $m = 7$, that is:

$$t_{accum\_tree} = t_{pp\_gen} + 3 \times t_{4:2\_CSA}, \tag{11}$$

with t_{pp_gen} the delay of the partial products generation stage, while the delay of a standard (24×24)-bit multiplier corresponds to

$$t_{std\_mult} = t_{pp\_gen} + t_{3:2\_CSA} + 2 \times t_{4:2\_CSA}. \tag{12}$$

Therefore, the proposed fused accumulation tree is only about 0.5τ slower than a standard single-precision multiplier (τ is the delay of a full-adder), with less area because of the reduced wordlengths of C_2 and C_1, the coefficients of our enhanced minimax polynomial approximation.

Fig. 5 shows the arrangement of the matrix of partial products to be accumulated for $m = 6$ and gives an idea of the effect of reducing the wordlengths of these two coefficients on the total area of the accumulation tree (C_2

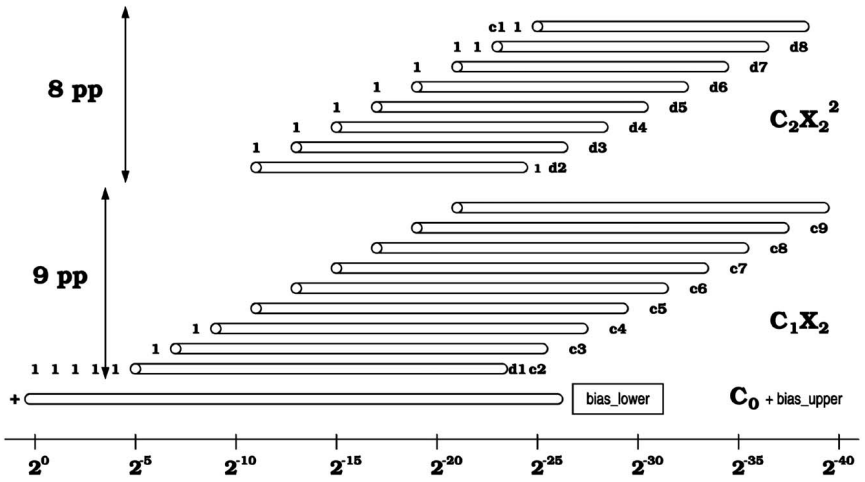

Fig. 5. Accumulation of the partial products (when $m = 6$).

affects eight partial products, while C_1 is involved in the generation of nine). Sign extension of the operands [28] is performed as shown in Fig. 5, with the circles at the beginning of each word meaning a complement of the sign bit. Bits c_i and d_j correspond to the LSB to be added to partial products i or j when multiples -1 and -2 are generated. c_is correspond to the partial products of C_1X_2, while d_js are related to the partial products of $C_2X_2^2$.

The injection of the function-specific rounding *bias* can be easily accomodated within the accumulation tree by a variety of techniques. As shown in Fig. 5, we have chosen to split such bias into *upper* and *lower* fields, adding the upper field to C_0 before storing it in the corresponding look-up table and appending to it the lower field within the accumulation tree.

4 Architecture

In this section, an architecture for the computation of reciprocal, square root, exponential, and logarithm, in single-precision floating-point format is proposed. The block diagram of such an unfolded architecture,[6] which implements our table-based method for function approximation, is shown in Fig. 6. Sign, exponent, and exception logic are not shown.

When implementing a single unit approximating several functions, only the look-up tables storing the polynomial coefficients must be replicated since the squaring unit, recoders, CPA, and multioperand adder can be shared for the different computations. Some kind of selection logic must be inserted to distinguish among the polynomial coefficients corresponding to different functions or, alternatively, a single ROM structure could be employed to store all coefficients, with the addressing acting as the multiplexing scheme.

Since $m = 6$ for square root and exponential and $m = 7$ for reciprocal and logarithm, the input operand X is split into X_1 and X_2 as follows:

$$X_1 = .x_1 \dots x_6 x_7, \tag{13}$$

with x_7 not affecting the coefficient selection for square root and exponential, and

$$X_2 = .x_7 x_8 \dots x_{23} \times 2^{-6}, \tag{14}$$

with x_7 set to 0 when approximating reciprocal and logarithm.

The size of the buses, the squaring unit, and the accumulation tree have been set according to the error computation in order to guarantee an accuracy of 1 ulp for all functions considered: 27 bits for C_0 (one integer and 26 fractional bits, with an implicit positive sign), 18 bits for C_1 (one sign, one integer, and 16 fractional bits), and 13 bits for C_2 (one sign, one integer, and 11 fractional bits), as shown in Fig. 6. On the other hand, the squaring unit and accumulation tree employed are those corresponding to $m = 6$ since the case $m = 7$ is thus subsumed.

The look-up tables storing the polynomial coefficients are addressed using X_1, two control bits (*op*) which select among the four functions to be approximated, and, for square root computations, also the least significant bit of the input exponent E_x, which determines whether the exponent is even or odd. In parallel, X_2 is recoded from binary representation to SD-4 and the computation of X_2^2 is carried out in *carry-save* form and then recoded to SD-4 representation. The sign, exponent, and exception logic also operate in parallel with the table look-up. The total table size is about $22.2Kb$, with a contribution of $2 \times 3.0625KB$ for the square root, $3.1875Kb$ for the exponential, $6.375Kb$ for the reciprocal, and $6.5Kb$ for the logarithm, as shown in Table 2.

All partial products are accumulated together using the multioperand adder, with a first level of 3:2 adders reducing the number of partial products of C_1X_2 from nine to six, and then three levels of 4:2 adders. Such a structure, as explained in Section 3.5, accomodates the delay of the required extra level of 3:2 adders and compensates for the delay of the path consisting of squaring computation and recoding. A function-specific compensation constant (*rounding bias*) is injected within the accumulation tree in order to perform the rounding while minimizing the maximum

6. Note that pipelining this architecture into a three-stage structure is straightforward, as shown in [29], [31].

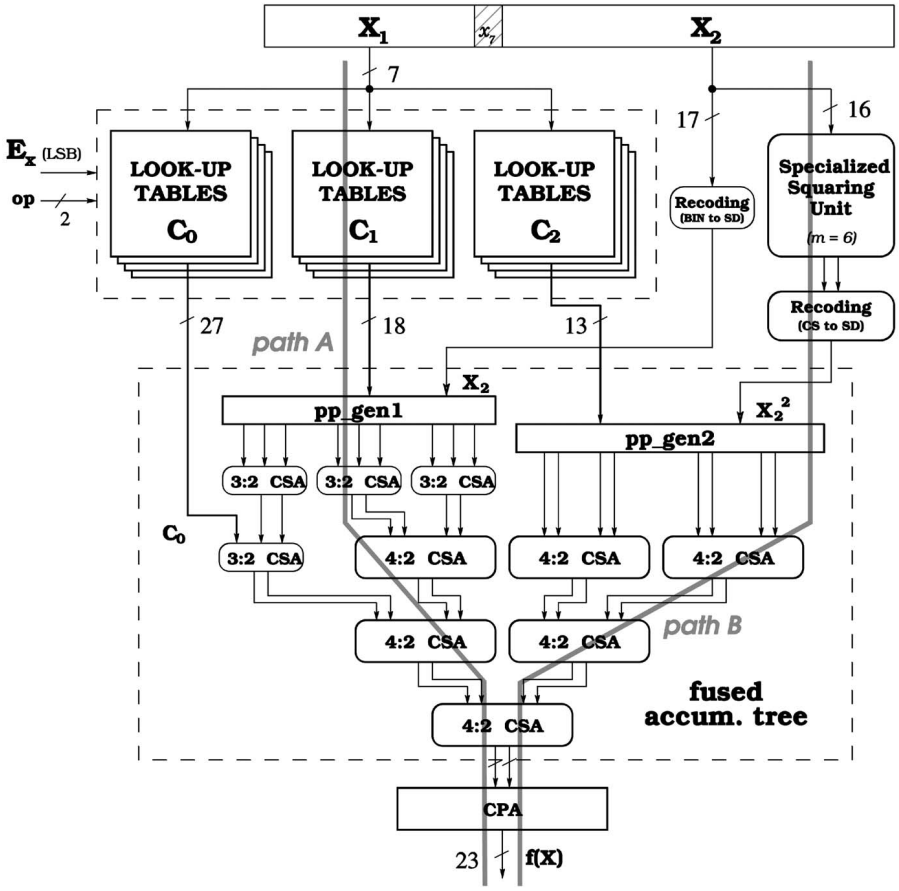

Fig. 6. Implementation of our minimax quadratic interpolator.

absolute error for each function. Finally, the assimilation from CS into nonredundant representation and normalization of the result are carried out by a fast adder.

4.1 Evaluation

Estimates of the cycle time and area costs of the proposed architecture are presented now, based on an approximate technology-independent model for the cost and delay of the main logic blocks employed. In this model, whose description can be found in [29], [30], the unit employed for the delay estimates is τ, while the area estimates are expressed as a multiple of fa, the delay and area, respectively, of a full-adder. Table 5 shows the area and delay estimates for the main logic blocks in our architecture, together with those corresponding to the logic blocks employed in the methods included in the comparison performed in Section 5. The delay estimates for the look-up tables already include an extra 0.5τ to account for the selection scheme required when several functions are approximated.

The critical path in our architecture is the slowest between two main candidates:

$$t_{path\_A} = t_{table\_C_1}(3.5\tau) + t_{pp\_gen}(1\tau) + t_{3:2\_CSA}(1\tau) + 3 \times t_{4:2\_CSA}(4.5\tau) + t_{CPA}(3\tau) + t_{reg}(1\tau) = 14\tau, \quad (15)$$

$$t_{path\_B} = t_{squaring}(3.5\tau) + t_{recoding}(1.5\tau) + t_{pp\_gen}(1\tau) + 3 \times t_{4:2\_CSA}(4.5\tau) + t_{CPA}(3\tau) + t_{reg}(1\tau) = 14.5\tau. \quad (16)$$

According to the delay estimates of the individual components shown in Table 5, the critical path in our architecture corresponds to $path\_B$ and the cycle time can be estimated as $t_{path\_B} = 14.5\tau$.

According to the area estimates of the individual components shown in Table 5, the total area can be estimated as $1,291fa$, with a contribution of $776fa$ from the $22.2Kbit$ look-up tables and of $515fa$ from the combinational logic blocks employed: a specialized squaring unit with $m = 6$ ($52fa$), a recoder from CS to SD-4 ($25fa$), a recoder from binary to SD-4 ($5fa$), the fused accumulation tree corresponding to $m = 6$ ($400fa$), a CPA ($21fa$), and registers ($12fa$). The exponent and sign logic, and the range reduction schemes, are not included in the evaluation.

5 Comparison

A comparison of the minimax quadratic interpolator with existing methods for function approximation is presented in this section. The area and delay estimates have been obtained using the same technology-independent model used for the evaluation of our architecture [29], [30], and are explained in detail in [32]. Table 5 shows the estimates for the main logic blocks employed in the compared methods.

TABLE 5
Area and Delay Estimates of Logic Blocks Used in the Compared Interpolators

Logic Block	Area (fa)	Delay (τ)
X_2 squaring ($m = 6/m = 8$)	52 / 35	3.5 / 3.0
recoding CS to SD	25	1.5
recoding BIN to SD	5	0.5
accum. tree ($m = 6/m = 8$)	400 / 350	$1 + 4.5 = 5.5$
24x24 mult.	471	2 + 4 + 3 = 9
(15x15)+26 Mult/Add	200	1.5 + 3 + 3 = 7.5
16x16 mult.	202	2 + 3 + 3 = 8
16x27 mult.	327	2 + 3 + 3 = 8
20x20 mult.	314	2 + 4 + 3 = 9
32-bit CPA	21	3
24-bit register	12	1
6-7 input-bit tables	35 fa/Kb	3.5
8-9 input-bit tables	35 fa/Kb	4.5
11-12 input-bit tables	30 fa/Kb	5.5
14-15 input-bit tables	25 fa/Kb	6.5

All considered interpolators employ similar tables and multipliers and, therefore, comparison results are technology-independent, with the relative values expressing trends among the different designs and making good first-order approximations to the actual execution times and area costs.

The comparison is organized in two parts. On the one hand, we compare the minimax quadratic interpolator with an optimized bipartite table method (SBTM) [34] and two linear approximation algorithms (DM97 and Tak98) [5], [40], when computing a single operation: reciprocal. This is due to the fact that DM97 [5] has been proposed only for performing such an operation and that Tak98 [40] has been proposed for powering computation with a fixed exponent (X^p, thus including the reciprocal when p is set to -1), but neither one is a general method for elementary function approximation. On the other hand, we compare the minimax interpolator with other quadratic methods (CW97, CWCh01, JWL93, and SS94) [1], [2], [15], [36] when employed as an elementary function generator, that is, when computing several operations (in this case, four operations) with the combinational logic shared for the different computations and a replicated set of look-up tables. In JWL93 [15] the functions approximated are reciprocal, square root, arctangent, and sine/cosine, CW97 [1] and CWCh01 [2] approximate reciprocal, square root, sin/cos and 2^X, and, in both SS94 [36] and our quadratic minimax interpolator, the functions computed are reciprocal, square root, exponential (2^X), and logarithm. We have chosen to conform to the original methods in order to obtain the area and delay estimates based on the table sizes reported by their respective authors.

Table 6 shows the table sizes, estimated area costs, and execution times of the minimax quadratic interpolator for the computation of the reciprocal when compared with those corresponding to SBTM [34], DM97 [5], and Tak98 [40]. Some assumptions have been made to allow for a fair comparison: Table sizes correspond to a final result accurate to 1 ulp and exponent and exception logic are not included in the area estimates.

The main conclusion to be drawn from this comparison is that bipartite table methods are simply unfeasible for single-precision computation due to the huge size of the tables to be employed. It is also noticeable that fast execution times can be obtained with linear approximation methods, but their hardware requirements are two to three times higher *per function* than those corresponding to the minimax quadratic interpolator, which, on the other hand, allows for similar execution times. When several functions must be approximated, the look-up tables storing the coefficients need to be replicated and the benefits of a fast quadratic interpolator regarding bipartite tables and linear approximation methods are significantly increased.

In order to compare the minimax quadratic interpolator with other second-degree approximation methods, when used as an elementary function generator, the total table size, execution time, and area costs, both from the look-up tables and from the combinational logic, are shown in Table 7. To allow for a fair comparison, table sizes correspond in all cases to final results accurate to 1 ulp. Note also that exponent and exception logic are not included in the area estimates and that execution times correspond to reciprocal and square root computation (for exponential, logarithm, or trigonometric functions, the delay of the range reduction and/or reconstruction steps should be added).

The analysis of approximations with Chebyshev polynomials of various degrees performed in [36] shows that a cubic approximation allows some area savings for 24-bits of precision and *exactly rounded* results. However, when results accurate to 1 ulp are allowed, a quadratic approximation is preferable in terms of both area and speed (see [36, Fig. 14]). Therefore, we have included in our comparison their quadratic interpolator (SS94) for the approximation of reciprocal, square root, exponential (2^X), and logarithm, with the table sizes reported in [36, Fig. 18], which correspond to a target precision of 24-bits, with 1 ulp accuracy.

The analysis of the estimates shown in Table 7 points out some interesting trends. Regarding execution time, JWL93,

TABLE 6
Architecture Comparison (Reciprocal Computation, 1 ulp Accuracy)

Scheme	*table size* (Kb)	*area (tables + logic)* (fa)	*exec. time*(τ)
SBTM [34]	1300	> 25000	**10**
Tak98 [40]	50	$1500 + 483 = 1983$	**15**
DM97 [5]	52	$1560 + 212 = 1772$	**13.5**
Enhanced Quadr. Minimax	6.4	$223 + 515 =$ **738**	**14.5**

TABLE 7
Architecture Comparison (Computation of Four Operations, 1 ulp Accuracy)

Scheme	*table size* (Kb)	*area (tables + logic)* (fa)	*exec. time*(τ)
JWL93 [15]	65.9	$2056 + 365 = 2670$	36
CW97 [1]	25	$875 + 663 = \mathbf{1538}$	31
CWCh01 [2]	17.2	$602 + 763 = \mathbf{1365}$	30
SS94 [36]	58	$2030 + 473 = 2503$	**17.5**
Enhanced Quadr. Minimax	22.2	$776 + 515 = \mathbf{1291}$	**14.5**

CW97, and CWCh01 show the main drawback of quadratic interpolators when the degree-2 polynomial is evaluated using a conventional scheme consisting of performing two serial multiplications and additions: low speed. Conversely, SS94 and the minimax quadratic interpolator use optimized schemes to deal with the polynomial evaluation, resulting in execution times similar to those achieved by linear approximation methods (about 14τ). In SS94, the two multiplications and additions are concurrently computed and the final result is obtained by using a 3-input multioperand adder, while, in our minimax quadratic interpolator, all partial products of the multiplications are added together using merged arithmetic and a final CPA assimilates the output of the fused accumulation tree into binary representation.

In terms of area, CW97 and CWCh01 methods show reduced hardware requirements (around $1,400fa$). This is due to the fact that those methods use a hybrid scheme where some function values are stored in the look-up tables instead of the polynomial coefficients, which allows a reduction in the total table size. However, such a reduction comes at the expense of computing the coefficients on-the-fly, which adds extra delay to the critical path and results in long execution times. The SS94 and JWL93 methods require bigger tables *per function* and, therefore, their hardware requirements are significantly larger.

The main difference between SS94 and the minimax quadratic interpolator is subtle, but important, and explains the different table size required for each method. While both methods rely on accurate mathematical approximations which outperform Taylor approximations (although, as shown by Piñeiro et al. [29], [31] and Muller [23], [25], the use of minimax yields slightly better accuracy than Chebyshev), the approximation in SS94 is computed without accounting for the effect of rounding the polynomial coefficients to a finite wordlength. However, three passes of the minimax approximation are performed in our algorithm, which allows compensating in each pass for the effect of rounding a coefficient to a finite wordlength (see Section 3.4). This property, combined with the intrinsic higher accuracy of the minimax approximation, results in a more accurate departing polynomial approximation, and makes feasible a reduction by one of the value of m for most functions. Note that a decrement in m by one results in a reduction by half in the size of the look-up tables to be employed, which grow exponentially with m (each table has 2^m entries), and, therefore, has a strong impact on the hardware requirements of the architecture. The final step of performing exhaustive simulation in order to minimize the width of the coefficients to be employed, which is common to both methods, has a much lower impact on the table size since marginal improvements can be achieved at that stage. Such a step is necessary, however, since any reduction in the wordlength of the coefficients may also help in reducing the size of the accumulation tree to be employed for the polynomial evaluation.

We can conclude, therefore, that our enhanced minimax interpolator shows the lowest overall area requirements (about $1,291fa$), similar to those of the best quadratic interpolators, which, on the other hand, have execution times over two times longer than those of our method. Such an area reduction, made possible by the high accuracy of the minimax approximation and to the 3-passes iterative algorithm employed to compute the polynomial coefficients, can be achieved without compromising speed since similar execution times to those of the linear approximations have been obtained. Moreover, when implementing an interpolator for the computation of different operations, the impact of reducing the table size in the overall area increases significantly since one set of look-up tables storing C_0, C_1, and C_2 must be used per function.

6 Conclusion

A table-based method for high-speed function approximation in single-precision floating-point format has been presented in this paper. Our method combines the use of an enhanced minimax quadratic approximation, which allows the use of reduced size look-up tables, and a fast evaluation of the degree-2 polynomial, which allows us to obtain execution times similar to those of linear approximations. An architecture implementing our method has been proposed for the computation of reciprocal, square root, exponential, and logarithm, with an accuracy of 1 ulp.

The significand of the input operand is split into two fields, X_1 and X_2, and an iterative algorithm which consists of 3-passes of the minimax approximation to compensate for the effect of rounding each coefficient to a finite wordlength is performed for each input subinterval. The coefficients of such quadratic approximation, C_0, C_1, and C_2, are stored in look-up tables addressed by the m-bit word X_1 (m being 6 for square root and exponential, while $m = 7$ for reciprocal and logarithm computation). In parallel with the table look-up, a specialized squaring unit with low area cost and delay performs the calculation of X_2^2. The values X_2 and X_2^2 are recoded into SD radix-4 representation in order to reduce the number of partial products of C_1X_2 and $C_2X_2^2$, which are accumulated together using a multioperand adder. Such an

accumulation tree has less area and a similar delay as a standard (24×24)-bit multiplier.

Estimates of the execution times and area costs for the proposed architecture have been obtained and a comparison with other quadratic interpolators, linear approximations, and bipartite tables methods has been shown, based on a technology-independent model for the area and delay of the main logic blocks employed. The conclusion to such a comparison is that our method combines the main advantage of linear approximations, the speed, and the main advantage of the second-degree approximations, the reduced size of the circuit, achieving the best trade off between area and performance.

The proposed method can be applied not only to the computation of reciprocal, square root, exponentials, and logarithms, but also for square root reciprocal, trigonometric functions, powering, or special functions, in single-precision format with an accuracy of 1 ulp, 2 ulps, or 4 ulps. It can also be employed for fixed-point computations or to obtain high-accuracy seeds for functional iteration methods such as Newton-Raphson or Goldschmidt, allowing significant latency reductions for those algorithms. All these features make our quadratic interpolator very suitable for implementing an elementary function generator in state-of-the-art DSPs or GPUs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their useful suggestions and contributions. J.-A. Piñeiro and J.D. Bruguera were supported by the Spanish Ministry of Science and Technology (MCyT-FEDER) under contract TIC2001-3694-C02. J.-A. Piñeiro was with the University of Santiago de Compostela when this work was performed.

REFERENCES

[1] J. Cao and B. Wei, "High-Performance Hardware for Function Generation," *Proc. 13th Int'l Symp. Computer Arithmetic (ARITH13),* pp. 184-188, 1997.

[2] J. Cao, B. Wei, and J. Cheng, "High-Performance Architectures for Elementary Function Generation," *Proc. 15th Int'l Symp. Computer Arithmetic (ARITH15),* pp. 136-144, 2001.

[3] T.C. Chen, "A Binary Multiplication Scheme Based on Squaring," *IEEE Trans. Computers,* vol. 20, pp. 678-680, 1971.

[4] W. Cody and W. Waite, *Software Manual for the Elementary Functions.* Prentice-Hall, 1980.

[5] D. DasSarma and D.W. Matula, "Faithful Interpolation in Reciprocal Tables," *Proc. 13th Symp. Computer Arithmetic (ARITH13),* pp. 82-91, 1997.

[6] D. DasSarma and D.W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1216-1222, Nov. 1998.

[7] K. Diefendorff, P.K. Dubey, R. Hochprung, and H. Scales, "Altivec Extension to PowerPC Accelerates Media Processing," *IEEE Micro,* pp. 85-95, Mar./Apr. 2000.

[8] M.D. Ercegovac and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications," *VLSI Signal Processing, III,* chapter 24, IEEE Press, 1988.

[9] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations.* Kluwer Academic, 1994.

[10] M.J. Flynn, "On Division by Functional Iteration," *IEEE Trans. Computers,* vol. 19, pp. 702-706, 1970.

[11] J. Foley, A. vanDam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice in C,* second ed. Addison-Wesley, 1995.

[12] D. Harris, "A Powering Unit for an OpenGL Lighting Engine," *Proc. 35th Asilomar Conf. Signals, Systems, and Computers,* pp. 1641-1645, 2001.

[13] J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.G. Thacher, and C. Witzgall, *Computer Approximations.* New York: Wiley, 1968.

[14] N. Ide et al. (Sony Playstation2), "2. 44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3D Graphics Computing," *IEEE J. Solid-State Circuits,* vol. 35, no. 7, pp. 1025-1033, July 2000.

[15] V.K. Jain, S.A. Wadecar, and L. Lin, "A Universal Nonlinear Component and Its Application to WSI," *IEEE Trans. Components, Hybrids, and Manufacturing Technology,* vol. 16, no. 7, pp. 656-664, 1993.

[16] T. Jayarshee and D. Basu, "On Binary Multiplication Using the Quarter Square Algorithm," *Proc. Spring Joint Computer Conf.,* pp. 957-960, 1974.

[17] I. Koren, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Trans. Computers,* vol. 40, pp. 1030-1037, 1990.

[18] A. Kunimatsu et al. (Sony Playstation2), "Vector Unit Architecture for Emotion Synthesis," *IEEE Micro,* vol. 20, no. 2, pp. 40-47, Mar./Apr. 2000.

[19] D.M. Lewis, "114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications," *IEEE J. Solid-State Circuits,* vol. 30, no. 12, pp. 1547-1553, 1995.

[20] P. Markstein, *IA-64 and Elementary Functions.* Hewlett-Packard Professional Books, 2000.

[21] C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* San Francisco: Morgan Kaufman, 1994.

[22] Microsoft Corp., *Microsoft DirectX Technology Review,* 2004, http://www.microsoft.com/windows/directx/default.aspx.

[23] J.-M. Muller, *Elementary Functions. Algorithms and Implementation.* Birkhauser, 1997.

[24] J.-M. Muller, "A Few Results on Table-Based Methods," *Reliable Computing,* vol. 5, no. 3, 1999.

[25] J.-M. Muller, "Partially Rounded Small-Order Approximations for Accurate, Hardware-Oriented, Table-Based Methods," *Proc. IEEE 16th Int'l Symp. Computer Arithmetic (ARITH16),* pp. 114-121, 2003.

[26] S. Oberman, G. Favor, and F. Weber, "AMD-3DNow! Technology: Architecture and Implementations," *IEEE Micro,* vol. 19, no. 2, pp. 37-48, Mar./Apr. 1999.

[27] S.F. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor," *Proc. 14th Symp. Computer Arithmetic (ARITH14),* pp. 106-115, Apr. 1999.

[28] A.R. Omondi, *Computer Arithmetic Systems. Algorithms, Architecture and Implementations.* Prentice Hall, 1994.

[29] J.-A. Piñeiro, "Algorithms and Architectures for Elementary Function Computation," PhD dissertation, Univ. of Santiago de Compostela, 2003.

[30] J.-A. Piñeiro and J.D. Bruguera, "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root," *IEEE Trans. Computers,* vol. 51, no. 12, pp. 1377-1388, Dec. 2002.

[31] J.A. Piñeiro, J.D. Bruguera, and J.-M. Muller, "Faithful Powering Computation Using Table Look-Up and Fused Accumulation Tree," *Proc. IEEE 15th Int'l Symp. Computer Arithmetic,* pp. 40-47, 2001.

[32] J.-A. Piñeiro, S. Oberman, J.-M. Muller, and J.D. Bruguera, "High-Speed Function Approximation Using a Minimax Quadratic Interpolator," technical report, Univ. of Santiago de Compostela, Spain, June 2004, http://www.ac.usc.es/publications.

[33] J.R. Rice, *The Approximation of Functions.* Reading, Mass.: Addison Wesley, 1964.

[34] M.J. Schulte and J.E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 842-847, Aug. 1999.

[35] M.J. Schulte and J.E. Stine, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. VLSI Signal Processing,* vol. 21, no. 2, pp. 167-177, 1999.

[36] M.J. Schulte and E.E. Swartzlander, "Hardware Designs for Exactly Rounded Elementary Functions," *IEEE Trans. Computers,* vol. 43, no. 8, pp. 964-973, Aug. 1994.

[37] M.J. Schulte and K.E. Wires, "High-Speed Inverse Square Roots," *Proc. 14th Int'l Symp. Computer Arithmetic (ARITH14),* pp. 124-131, Apr. 1999.
[38] E.M. Schwarz and M.J. Flynn"Hardware, Starting Approximation for the Square Root Operation" *Proc. 11th Symp. Computer Arithmetic (ARITH11),* pp. 103-111, 1993.
[39] H.C. Shin, J.A. Lee, and L.S. Kim, "A Minimized Hardware Architecture of Fast Phong Shader Using Taylor Series Approximation in 3D Graphics," *Proc. Int'l Conf. Computer Design, VLSI in Computers and Processors,* pp. 286-291, 1998.
[40] N. Takagi, "Powering by a Table Look-Up and a Multiplication with Operand Modification," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1216-1222, Nov. 1998.
[41] P.T.P. Tang, "Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic," *ACM Trans. Math. Software,* vol. 4, no. 16, pp. 378-400, Dec. 1990.
[42] P.T.P. Tang, "Table Look-Up Algorithms for Elementary Functions and Their Error Analysis," *Proc. IEEE 10th Int'l Symp. Computer Arithmetic (ARITH10),* pp. 232-236, 1991.
[43] Waterloo Maple Inc., *Maple 8 Programming Guide,* 2002.

Jose-Alejandro Piñeiro received the BSc degree (1998) and the MSc degree (1999) in physics (electronics) and the PhD degree in computer engineering in 2003 from the University of Santiago de Compostela, Spain. Since 2004, he has been with Intel Barcelona Research Center (IBRC), Intel Labs-UPC, whose research focuses on new microarchitectural paradigms and code generation techniques for the IA-32, EM64T, and IPF families. His research interests are also in the area of computer arithmetic, VLSI design, computer graphics, and numerical processors.

Stuart F. Oberman received the BS degree in electrical engineering from the University of Iowa, Iowa City, in 1992 and the MS and PhD degrees in electrical engineering from Stanford University, Palo Alto, California, in 1994 and 1997, respectively. He has participated in the design of several commercial microprocessors and floating-point units. From 1995-1999, he worked at Advanced Micro Devices, Sunnyvale, California, where he coarchitected the 3DNow! multimedia instruction-set extensions, designed floating-point units for the K6 derivatives, and was the architect of the Athlon floating-point unit. Since 2002, he has been a principal engineer at NVIDIA, Santa Clara, California, where he designs Graphics Processing Units. He has coauthored one book and more than 20 technical papers. He holds more than 30 granted US patents. He is a Tau Beta Pi Fellowship recipient and a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and the IEEE Computer Society.

Jean-Michel Muller received the PhD degree from the Institut National Polytechnique de Grenoble in 1985 and became a full-time researcher in the Centre National de la Recherche Scientifique in 1986. He is now chairman of LIP, which is a common computer science laboratory of the Ecole Normale Supérieure de Lyon, INRIA, CNRS, and Universite Claude Bernard. He served as coprogram chair of the 13th IEEE Symposium on Computer Arithmetic (ARITH-14, Asilomar, California) in 1997, as general chair of ARITH-15 (Adelaide, Australia) in 1999, and as an associate editor of the *IEEE Transactions on Computers* from 1996 to 2000. He is a senior member of the IEEE.

Javier D. Bruguera received the BS degree in physics and the PhD degree from the University of Santiago de Compostela, Spain, in 1984 and 1989, respectively. Currently, he is a professor in the Department of Electronic and Computer Engineering at the University of Santiago de Compostela. Previously, he was an assistant professor in the Department of Electrical, Electronic, and Computer Engineering at the University of Oviedo, Spain, and an assistant professor in the Department of Electronic Engineering at the University of A Coruña, Spain. He was a visiting researcher in the Application Center of Microelectronics at Siemens in Munich, Germany, and in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. His primary research interests are in the area of computer arithmetic, processor design, digital design for signal and image processing, and parallel architectures. He is a member of the IEEE.