High-Speed Double-Precision Computation of Reciprocal, Division, Square Root, and Inverse Square Root

José-Alejandro Piñeiro, *Student Member*, *IEEE*, and Javier Díaz Bruguera, *Member*, *IEEE*

Abstract—A new method for the high-speed computation of double-precision floating-point reciprocal, division, square root, and inverse square root operations is presented in this paper. This method employs a second-degree minimax polynomial approximation to obtain an accurate initial estimate of the reciprocal and the inverse square root values, and then performs a modified Goldschmidt iteration. The high accuracy of the initial approximation allows us to obtain double-precision results by computing a single Goldschmidt iteration, significantly reducing the latency of the algorithm. Two unfolded architectures are proposed: the first one computing only reciprocal and division operations, and the second one also including the computation of square root and inverse square root. The execution times and area costs for both architectures are estimated, and a comparison with other multiplicative-based methods is presented. The results of this comparison show the achievement of a lower latency than these methods, with similar hardware requirements.

Index Terms—Computer arithmetic, Goldschmidt iteration, table-based methods, double-precision operations, division, square root, inverse square root.

1 Introduction

Reciprocal, division, square root, and inverse square root are important operations for several applications such as digital signal processing, multimedia, computer graphics, and scientific computing [7], [12], [18], [22]. Although they are less frequent than the two basic arithmetic operations, the poor performance of many processors when computing these operations can make their overall execution time comparable to the time spent performing addition and multiplication [16].

For a low precision computation of these functions, it is possible to employ direct table look-up, bipartite tables [2], [21] (table look-up and addition), or low-degree polynomial or rational approximations [11], [15], [23], [24], but the area requirements become prohibitive for table-based methods when performing high precision computations (such as the 53-bit accuracy double-precision floating-point format). Iterative algorithms are employed for these calculations. On one hand, digit-recurrence methods [4], [8], such as the SRT algorithm, result in small units, but their linear convergence sometimes leads to long latencies and makes them inadequate methods for these computations. High-radix digit-recurrence methods [13] result in faster but bigger designs. On the other hand, multiplicative-based methods [3], [6], [7], such as the Newton-Raphson and Goldschmidt algorithms, have quadratic convergence, which leads to faster execution times, usually at the expense of greater hardware requirements. These methods employ an initial table-based low-precision approximation (*seed* value) of the final result to reduce the number of iterations to be performed, thus reducing the overall latency of the algorithm.

A new multiplicative-based method is proposed in this paper. Our method combines an efficient second-degree minimax polynomial approximation [20] to obtain the *seed* values and a modified Goldschmidt iteration to provide double-precision floating-point results. The high accuracy (about 30-bits of precision) of the second-degree approximation allows the computation of a single iteration, significantly reducing the overall latency of the standard Goldschmidt algorithm. On the other hand, the modification performed in the iteration allows an important reduction on the hardware requirements. We have chosen this algorithm instead of Newton-Raphson due to its higher intrinsic parallelism, which leads to lower execution times.

The second-degree minimax polynomial approximation [20] consists of three look-up tables storing C_0, C_1, and C_2, the coefficients of a second-degree polynomial. This approximation combines the speed of linear approximations and the reduced area of the second-degree interpolations. The look-up tables are addressed with the upper part of the significand input, X_1 (having a wordlength of $m_1 = 9$ bits), and the evaluation of the quadratic polynomial is carried out by a specialized squaring unit and a multioperand adder. When obtaining the initial approximation for several different functions, only the look-up tables storing the coefficients must be replicated because the squaring unit and multioperand adder can be shared for the different computations.

Since there are two different implementations of the Goldschmidt algorithm for division and square root functions, we propose two architectures in this paper. The first one deals only with the computation of reciprocal and

• *The authors are with the Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain. E-mail: {alex, bruguera}@dec.usc.es.*

Manuscript received 20 Apr. 2001; revised 25 Jan. 2002; accepted 21 Mar. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114026.

division operations, with a very low latency and reasonable hardware requirements. The second proposed architecture also deals with the computation of square root and inverse square root, with an increased latency (but still a fast execution time compared with previous methods) and about the same amount of hardware as the first scheme. The modification performed in the Goldschmidt iteration for computing square root and inverse square root allows the reutilization of the logic blocks employed in the first architecture, saving an important amount of area.

The rest of this paper is structured as follows: In Section 2, a brief description of the algorithm is presented, including an overview of the second-degree minimax approximation; the two unfolded architectures are proposed and implementation details are explained in Section 3; estimates of the execution time and hardware requirements for our proposed architectures are presented and a comparison with some previous multiplicative-based methods is outlined in Section 4; finally, the main contributions made in this work are summarized in Section 5.

2 Algorithm

The method proposed in this paper deals with the computation of the reciprocal function $(1/X)$, division (Y/X), square root $(\sqrt{X})$, and inverse square root $(1/\sqrt{X})$ for input operands in the IEEE double-precision floating-point format. With this format, a floating-point number M is represented using a sign bit s_m, an 11-bit biased exponent e_m, and a 53-bit significand X. If M is a normalized number, it represents the following value:

$$M = (-1)^{s_m} \times (1 + f_m) \times 2^{e_m - 1023},$$

where $X = 1 + f_m$, $1 \leq X < 2$, and f_m is the fractional part of the normalized number (the 52-bit stored word).

The computation of these functions is performed only for the input significand, $Z = f(X)$, since the sign and exponent treatment is straightforward and can be performed in parallel.

Our method consists of the following steps:

- Computing an initial approximation $R_f \approx f(\hat{X})$, with $\hat{X}$ a truncated version of the input operand, accurate to 30 bits for the reciprocal and to 29 bits for the inverse square root,[1] by employing a second-degree minimax polynomial approximation.
- Performing a modified Goldschmidt iteration, employing R_f as a seed, to produce the final double-precision result $Z = f(X)$.

In the next subsection, we briefly describe the second-degree minimax approximation employed (a detailed description can be found in [20]). After this description, the modified Goldschmidt algorithm for the computation of reciprocation/division and also the one for the computation of square root/inverse square root will be presented.

1. These values will be justified in Sections 2.2 and 2.3, where the error computation for the Goldschmidt algorithm is explained.

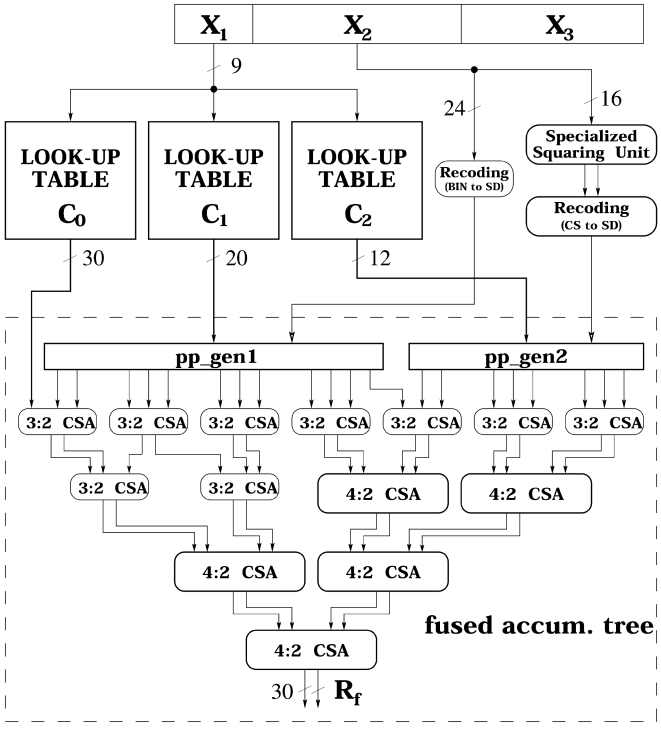

Fig. 1. Block diagram of the second-degree minimax approximation R_f.

2.1 Second-Degree Minimax Polynomial Approximation

A new method for the computation of powering function (X^p), in a single-precision floating-point format, by means of a second-degree minimax polynomial approximation with table look-up and multioperand accumulation, has been proposed in [20]. This algorithm allows an important reduction in the total area regarding linear approximations, with no delay increase. It combines the speed of first-degree approximation methods [23] and the reduced size of second-degree interpolation algorithms [1], [10].

Since reciprocal and inverse square root functions are specific cases of powering, this second-degree approximation can be effectively employed to obtain accurate initial estimates for both functions. These estimates are the seed values required for the modified Goldschmidt algorithms ($R_d = 1/\hat{X}$ will be the 30-bit seed value for division and reciprocal computation, while $R_s = 1/\sqrt{\hat{X}}$ will be the 29-bit initial value employed for the square root and inverse square root computation).

Since the minimax approximation is known to be the optimal polynomial approximation of a function [15], we take advantange of its high accuracy to significantly reduce the wordlengths of the coefficients to be employed, reducing the size of the required look-up tables.

As shown in Fig. 1, the n-bit binary input significand X is split into an upper part X_1, a middle part X_2, and a lower part X_3.

$$X_1 = [1.x_1x_2 \ldots x_{m_1}]$$
$$X_2 = [.x_{m_1+1} \ldots x_{m_2}] \times 2^{-m_1}$$
$$X_3 = [.x_{m_2+1} \ldots x_n] \times 2^{-m_2}.$$

An approximation to X^p in the range $X_1 \leq X < X_1 + 2^{-m_1}$ can be performed evaluating the expression

$$X^p \approx C_0 + C_1X_2 + C_2X_2^2, \quad (1)$$

where the coefficients C_0, C_1, and C_2 are obtained employing the computer algebra system **Maple** [25], which performs minimax approximations using the Remez algorithm. X_3 is not involved in any of these computations, having no effect on the value of the initial approximation R_f.

The values of these coefficients depend only on the value of X_1, the m_1 most significant bits of X, and on the parameter p. Therefore, C_0, C_1, and C_2 can be stored in look-up tables of 2^{m_1} input values.[2]

Apart from the table look-up, the quadratic polynomial (1) has to be efficiently evaluated. There are two main problems to be overcome: the squaring (X_2^2) and the two multiplications plus two additions to be computed. Rather than having a multiplier to compute the squaring function, some important properties can be used to significantly reduce the amount of area of the resulting unit and its associated delay. The techniques employed to obtain this area and delay reduction are considering the leading zeros of X_2, rearranging the matrix of partial products, and truncating this matrix within the error bounds. These strategies allow the achievement of lower delays and of a total area slightly over half of the area of the corresponding multiplier unit [20]. For the computation of the quadratic polynomial, we propose employing a unified multioperand accumulation tree. The inputs of this adder are the partial products of both C_1X_2 and $C_2X_2^2$, plus the coefficient C_0. *Signed-digit radix 4* (SD-4) representation [14] of the multiplier operands is employed to reduce the number of partial products to be accumulated.

2.1.1 Error Computation

The precision of the initial estimates obtained by computing the second-degree minimax approximation can be calculated taking into account the two main error sources: the error in the minimax approximation itself (ϵ_{approx}) and the error due to the employment of finite arithmetic in the computation of the quadratic polynomial. The error in the initial approximation is:

$$\epsilon_{R_f} \leq \epsilon_{approx} + \epsilon_{C_0} + \epsilon_{C_1}X_2 + \epsilon_{C_2}X_2^2 + |C_1|_{max}\epsilon_{X_2} + |C_2|_{max}\epsilon_{X_2^2}.$$

Since the error due to the finite wordlength of the coefficients is obtained as a result of the Maple program implementing our method [20]:

$$\epsilon'_{approx} = \epsilon_{approx} + \epsilon_{C_0} + \epsilon_{C_1}X_2 + \epsilon_{C_2}X_2^2,$$

2. $m_1 = 9$ and $m_2 = 33$ in this case, as will be shown in the error computation section.

therefore:

$$\epsilon_{R_f} \leq \epsilon'_{approx} + |C_1|_{max}\epsilon_{X_2} + |C_2|_{max}\epsilon_{X_2^2}.$$

The minimum value of m_1 that allows the achievement of the target precision ($\epsilon_{R_d} < 2^{-30}$ and $\epsilon_{R_s} < 2^{-29}$) is $m_1 = 9$ for both cases. Once this value has been set, the minimax approximation is computed by employing Maple, minimizing the wordlengths of the coefficients C_0, C_1, and C_2 within the error bounds.

The wordlength of the coefficients is 30, 20, and 12 bits, respectively, when computing the reciprocal approximation, while 29, 18, and 10 bits when computing the inverse square root. Therefore, the bus sizes have to be set to 30, 20, and 12 bits. The size of the tables to be employed for the reciprocal computation is $2^9 \times (30 + 20 + 12) = 31Kb$, i.e., less than $4KB$. For the inverse square root computation, since the input operand X is in the range $1 \leq X < 4$, two sets of tables are needed, resulting in a table size of $2 \times 2^9 \times (29 + 18 + 10) = 57Kb$, i.e., about $7KB$. The total table size is therefore of $31 + 57 = 88Kb$, i.e., $11KB$.

The target precision in the worst case (reciprocal approximation, with $|C_1|_{max} = |C_2|_{max} = 1$) sets a minimum value both of m_2 and of the position j where the squaring operation X_2^2 is truncated [20]. These values are $m_2 = 33$, which makes X_2 to have a wordlength of $m_2 - m_1 = 24$ bits and $j = 34$, which means that no bits of X after the position with weight 2^{-25} are required for the squaring computation. The wordlength of X_2^2 will therefore be 16 bits.

In spite of the fact that the wordlengths of the coefficients C_1 and C_2 are shorter than those of X_2 and X_2^2, respectively, we employ these values as multipliers in the partial product generation for various reasons: The most important one is that the assimilation of X_2^2 from CS to binary representation can be avoided and substituted by a CS to SD-4 recoding; it is also interesting that the recoding of X_2 from binary to SD-4 representation can be performed in parallel with the squaring operation and the table look-up, introducing no extra delay in the critical path. The size of the fused accumulation tree remains the same, regardless of which values are employed as multipliers and which ones as multiplicands.

The total number of partial products to be accumulated is 21 (eight from the $C_2X_2^2$ product, 12 from C_1X_2, plus the independent coefficient C_0), which causes the multioperand adder tree to have four levels of carry-save adders, as shown in Fig. 1, with a total delay of

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

with t_{pp_gen} the delay of the partial product generation stage. The result is provided in CS representation.

2.2 Modified Reciprocal/Division Goldschmidt Algorithm

Assume two n-bit inputs Y and X satisfying $1 \leq Y, X < 2$. The Goldschmidt algorithm [3] for computing the division operation ($Z = Y/X$) consists of finding a sequence $K_1, K_2, K_3, \ldots$ such that

$$r_i = XK_1K_2 \ldots K_i \rightarrow 1$$

and, therefore,

$$z_i = YK_1K_2 \ldots K_i \rightarrow Y/X.$$

The reciprocal ($Z = 1/X$) can be computed as a specific case of division:

$$z_i = K_1K_2 \ldots K_i \rightarrow 1/X.$$

The high accuracy (30 bits) of the first factor $K_1 = R_d$, obtained by employing the second-degree minimax approximation, guarantees that a single Goldschmidt iteration suffices to obtain double-precision results. This is due to the fact that the error ϵ_i on the iteration i follows the quadratic relation [3]:

$$\epsilon_i = XY\epsilon_{i-1}^2$$

for the division operation.

Our method for computing reciprocal and division operations consists of a reorganization of the steps to be performed in the traditional Goldschmidt algorithm and can be summarized as follows:

- Obtaining a seed value $R_d = K_1$ by performing a second-degree minimax approximation of the reciprocal ($1/\hat{X}$). Since R_d has an accuracy of 30 bits, $\epsilon_{R_d} < 2^{-30}$.
- Computing $G_d = R_dY$. For the division computation, $z_1 = G_d$, while $z_1 = R_d$ when computing the reciprocal operation.
- Computing $V_d = 1 - R_dX$. This value is guaranteed to be bounded by $|V_d| \leq 2^{-29}$. Note that G_d and V_d can be computed in parallel.
- Computing $Z = z_2 = z_1K_2$. Since $K_2 = 1 + V_d$, $Z = G_d(1 + V_d) = G_d + G_dV_d$ for division and $Z = R_d(1 + V_d) = R_d + R_dV_d$ when computing the reciprocal.

After performing the last computation, the maximum absolute error ϵ_Z is proportional to $\epsilon_{R_d}^2$, but also includes the error (ϵ_{comp}) due to the employment of finite arithmetic in the computation of the steps of our method:

$$\epsilon_Z = XY\epsilon_{R_d}^2 + \epsilon_{comp}. \tag{2}$$

Let ϵ_{Gt}, ϵ_{Vt}, and ϵ_{Zt} be the errors introduced by employing units with wordlength t. The value of t has to be set so that double-precision results accurate to 1 ulp are guaranteed. The value of ϵ_{comp} is:

$$\epsilon_{comp} = \epsilon_{Gt} + R_dY\epsilon_{Vt} + Y\epsilon_{Vt}\epsilon_{R_d} + X\epsilon_{Gt}\epsilon_{R_d} + \epsilon_{Vt}\epsilon_{Gt} + \epsilon_{Zt}. \tag{3}$$

Since the range of the results for the division operation is $0.5 < Z < 2$, the error in the final result must be bounded by:

$$\begin{aligned}\epsilon_Z = XY\epsilon_{R_d}^2 + \epsilon_{Gt} + R_dY\epsilon_{Vt} + Y\epsilon_{Vt}\epsilon_{R_d} + X\epsilon_{Gt}\epsilon_{R_d} \\ + \epsilon_{Vt}\epsilon_{Gt} + \epsilon_{Zt} < 2^{-54}.\end{aligned} \tag{4}$$

Assuming that Z is obtained in the last step after performing rounding to the nearest of the final result ($\epsilon_{Zt} \leq 2^{-55}$), the target precision can only be met, according to (4), if $\epsilon_{Gt} \leq 2^{-57}$ and $\epsilon_{Vt} \leq 2^{-57}$, with $\epsilon_{R_d} < 2^{-30}$:

$$\begin{aligned}\epsilon_Z = XY\epsilon_{R_d}^2 + \epsilon_{Gt} + R_dY\epsilon_{Vt} + Y\epsilon_{Vt}\epsilon_{R_d} + X\epsilon_{Gt}\epsilon_{R_d} \\ + \epsilon_{Vt}\epsilon_{Gt} + \epsilon_{Zt} \\ < 2^{-58} + 2^{-57} + 2^{-56} + 2^{-86} + 2^{-86} + 2^{-114} + 2^{-55} < 2^{-54}.\end{aligned}$$

The bounds on ϵ_{Gt} and ϵ_{Vt} can be met by employing truncation on the multipliers which carry out with G_d and V_d computation if a $t = 57$-bit wordlength is employed. Another alternative is employing rounding to nearest and a $t = 56$-bit wordlength, which guarantees the same bound on the errors.

2.3 Modified Square Root/Inverse Square Root Goldschmidt Algorithm

Assume an n-bit input operand X satisfying $1 \leq X < 2$. In this case, a sequence $K_1, K_2, K_3, \ldots$ should be found such that

$$z_i = K_1K_2 \ldots K_i \rightarrow 1/\sqrt{X}$$

and, therefore,

$$z_i = XK_1K_2 \ldots K_i \rightarrow \sqrt{X}.$$

We follow the same strategy in this case as that employed for the reciprocal/division computation: providing an accurate initial approximation obtained by performing a second-degree minimax approximation and then performing a single Goldschmidt iteration to obtain the double-precision results $Z = \sqrt{X}$ or $Z = 1/\sqrt{X}$. This is possible due to the quadratic convergence of the Goldschmidt algorithm [3]:

$$\epsilon_i = \left(3\sqrt{X}\epsilon_{i-1}^2 + X\epsilon_{i-1}^3\right)/2$$

for inverse square root computation and

$$\epsilon_i = \left(3X\sqrt{X}\epsilon_{i-1}^2 + X^2\epsilon_{i-1}^3\right)/2$$

for square root computation.

In the traditional Goldschmidt algorithm for square root/inverse square root computation, both the seed value $K_1 = 1/\sqrt{\hat{X}}$ and its square $1/\hat{X}$ are required. Two look-up tables can be employed to obtain these values, but these tables must be designed so that each table look-up value $1/\hat{X}$ corresponds at full target precision to the square of the table look-up value K_1 (this constraint can only be met by direct table look-up, excluding the possibility of employing table-based methods). Another way of obtaining these values is employing a table-driven method for K_1 (which results in a very important area reduction regarding direct table look-up) and then computing the squaring operation $R_d = (K_1)^2$.

Instead of employing a look-up table or a specialized squaring unit, we have arranged the sequence of computations to be performed so that the hardware employed for the reciprocal/division computation can be reutilized for the square root/inverse square root computation.

Summarizing, our method consists of performing the following steps:

- Obtaining $R_s = K_1$ by performing a second-degree minimax approximation of the inverse square root

$(1\sqrt{\hat{X}})$. Since R_s has an accuracy of 29 bits, $\epsilon_{R_s} < 2^{-29}$.

- Computing $G_s = R_s X$. For the square root computation, $z_1 = G_s$, while, for the inverse square root, $z_1 = R_s$.
- Computing $V_s = 1 - R_s G_s$. This value is guaranteed to be bounded by $|V_s| \leq 2^{-28}$. In this case, G_s and V_s obviously cannot be computed in parallel since G_s is employed in the computation of V_s. The employment of G_s in V_s computation allows us to save an important amount of area since neither a squaring unit nor another set of initial look-up tables are necessary to obtain $1/\hat{X}$.
- Computing $Z = z_2 = (1 + \frac{V_s}{2})z_1$. For the square root computation, this becomes

$$Z = G_s(1 + V_s/2) = G_s + G_s V_s/2,$$

while, for the inverse square root, we have

$$Z = R_s(1 + V_s/2) = R_s + R_s V_s/2.$$

The worst case in the error computation corresponds to the inverse square root operation. Since the output range for this function is $0.5 < Z \leq 1$, the maximum absolute error ϵ_Z is bounded in this case by:

$$\epsilon_Z = \frac{3\sqrt{X}\epsilon_{R_s}^2}{2} + \frac{X\epsilon_{R_s}^3}{2} + \frac{\epsilon_{Gt}}{2X} + \frac{\epsilon_{Vt}}{2\sqrt{X}} + \frac{\epsilon_{R_s}\epsilon_{Vt}}{2} + \frac{\epsilon_{Gt}\epsilon_{R_s}^2}{2} + \frac{\epsilon_{R_s}\epsilon_{Gt}}{\sqrt{X}} + \epsilon_{Zt} < 2^{-54}. \quad (5)$$

In this expression, the effect of employing a finite wordlength datapath has already been taken into account. Since the values of $\epsilon_{Zt} \leq 2^{-55}$, $\epsilon_{Gt} \leq 2^{-57}$, and $\epsilon_{Vt} \leq 2^{-57}$ have been set by the error bounds in the division and reciprocal computation, the only bound to be determined here is ϵ_{R_s}. According to (5), a maximum absolute error in the initial approximation of $\epsilon_{R_s} < 2^{-29}$ suffices for guarantees the achievement of double-precision results with an error of less than 1 ulp for the four functions to be computed.

$$\begin{aligned}\epsilon_Z &= \frac{3\sqrt{X}\epsilon_{R_s}^2}{2} + \frac{X\epsilon_{R_s}^3}{2} + \frac{\epsilon_{Gt}}{2X} + \frac{\epsilon_{Vt}}{2\sqrt{X}} + \frac{\epsilon_{R_s}\epsilon_{Vt}}{2} + \frac{\epsilon_{Gt}\epsilon_{R_s}^2}{2} \\ &\quad + \frac{\epsilon_{R_s}\epsilon_{Gt}}{\sqrt{X}} + \epsilon_{Zt} \\ &< 2^{-57} + 2^{-87} + 2^{-58} + 2^{-58} + 2^{-87} + 2^{-116} + 2^{-86} + 2^{-55} \\ &< 2^{-54}.\end{aligned}$$

3 Architecture

In this section, we propose an unfolded architecture for the computation of reciprocal and division operations and another unfolded architecture for the computation of all four functions: reciprocal, division, square root, and inverse square root. The design of both architectures is based on the method and error analysis presented in the previous section.

Pipelining could be applied to both methods, resulting in the corresponding pipelined architectures. Furthermore, both methods could also be implemented in a pipelined multiplier-based processor, such as the AMD-K7 or similar [17].

The final results produced are guaranteed to be accurate to 1 ulp. As happens to other multiplicative-based methods, it is not possible to directly obtain exactly rounded results without an important overhead (an error less than 2^{-2n} or 2^{-3n} should be provided, for n-bit operands, depending on the function to be computed). Another alternative is to produce a result accurate to 1 ulp (as we guarantee) and determine the exactly rounded value by computing the corresponding remainder. This strategy is employed by the AMD-K7 [17], computing $rem = X \times Z - Y$ for the division operation and $rem = Z \times Z - X$ for the square root. The sign of rem, together with the information on the rounding mode to be employed, determines the rounding direction and guarantees the computation of exactly rounded IEEE double-precision results.

3.1 Reciprocal/Division Unit

The steps to be performed for computing division and reciprocal operations can be summarized as:

$$\begin{aligned}R_d &= 2nd\ degree\ approx(1/X) \\ G_d &= R_d Y \\ V_d &= 1 - R_d X \\ Z &= G_f + G_f V,\end{aligned}$$

where G_d and V_d can be computed in parallel and $G_f = G_d$ when computing division operation, while $G_f = R_d$ when computing the reciprocal $1/X$.

Fig. 2 shows the block diagram of the unit implementing this algorithm. The $m_1 = 9$ most significant bits of the input operand X are employed to address the second-degree approximation look-up tables. The squaring operation is performed in parallel with the table look-up and all partial products of polynomial (1) are accumulated with a multi-operand adder, whose output R_d is in CS representation.

The 30-bit operand R_d is employed as a multiplier in the computation of both G_d and V_d. Therefore, a CS to SD-4 recoding unit is required. When computing the reciprocal operation, R_d is also employed as an addend and multiplicand in the computation of Z, so its two's-complement representation is needed. This assimilation to nonredundant representation is carried out by a CPA in parallel with the CS to SD recoding operation and the computation of G_d and V_d.

The computation of G_d is carried out by a 30×53 bit multiplier unit with SD representation for the multiplier operand and two's-complement rounded to nearest output. Nonredundant representation is used because G_d is going to be employed as addend and multiplicand in the multiply-add unit which computes Z. The computation of V_d is carried out by a 30×53 bit multiply-complement unit, also with an SD representation for the multiplier and a two's-complement output. Since $|V_d| \leq 2^{-29}$, it has at least 29 leading zeros (or ones), only one of them needed as sign bit. Therefore, V_d has only $56 - 28 = 28$ significant bits and an important amount of area can be reduced by omitting the 29 most significant columns of the multiplication matrix. Besides, the short wordlength of V_d allows the employment of a smaller multiply-add unit for the computation of Z.

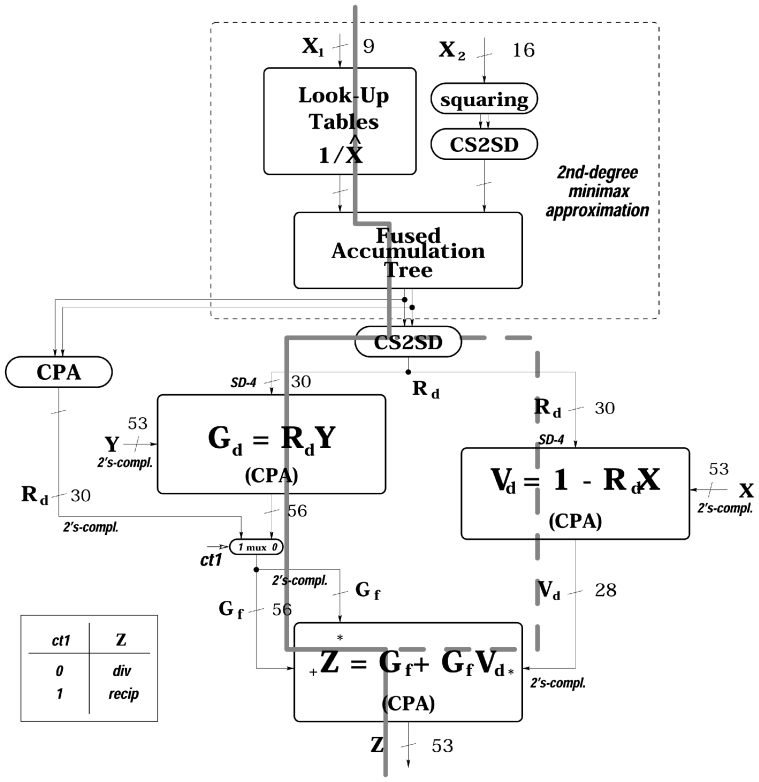

Fig. 2. Reciprocal/division unit architecture.

A 56-bit 2:1 multiplexer is employed to select between G_d and R_d. When computing division, G_f should be chosen as G_d, so the control signal $ct1$ has to be set to the appropriate value ($ct1 = 0$), while $ct1 = 1$ sets $G_f = R_d$ and the computed operation is $Z = 1/X$.

The computation of Z is carried out by a $28 \times 56 + 56$ bit multiply-add unit with two's complement inputs and output. The recoding from nonredundant representation to SD-4 is performed internally in this unit. The output of this unit is the 53-bit final result $Z = Y/X$, when computing division, or $Z = 1/X$, when computing the reciprocal operation.

There are two main paths in this architecture since G_d and V_d are computed in parallel. Anyway, as both units are 30×53 bit multipliers, the critical path consists of the second-degree approximation look-up tables, the fused accumulation tree with CS result, the CS to SD recoding unit, a 30×53 bit multiplier (G_d computation), a 2:1 multiplexer, and a $28 \times 56 + 56$ bit multiply-add unit (Z computation).

3.2 Reciprocal/Division/Square Root/Inverse Square Root Unit

The steps to be performed for computing the square root or the inverse square root operation with our method can be summarized as:

$$
\begin{aligned}
R_s &= 2nd\ degree\ approx(1/\sqrt{X}) \\
G_s &= R_s X \\
V_s &= 1 - R_s G_s \\
Z &= G_f + G_f V_s/2,
\end{aligned}
$$

where G_s and V_s obviously cannot be computed in parallel and $G_f = G_s$ when computing the square root operation, while $G_f = R_s$ when computing the inverse square root $1/\sqrt{X}$.

The modified Goldschmidt iteration for square root and inverse square root computation has been designed so that it involves similar steps as the ones required in the reciprocal and division computation. The logic blocks employed in the previously proposed architecture can thus be reutilized in a complete architecture computing the four operations. A diagram block of this architecture (i.e., the architecture that deals with the computation of reciprocal, division, square root, and inverse square root operations) is shown in Fig. 3, together with a summary of the control signals encoding for an appropriate behavior of the unit. Table 1 shows the operations performed in each logic block, depending on the function to be computed.

The second-degree approximation of the inverse square root ($R_s = 1/\sqrt{\hat{X}}$) can be computed with the same hardware as that employed for the reciprocal approximation ($R_d = 1/\hat{X}$) by only inserting a new set of look-up tables for the coefficients, due to the fact that the squaring operation and the accumulation tree do not depend on the function to compute and, therefore, can be shared for both computations. A set of multiplexers must also be inserted at the inputs of the multioperand adder to select between the reciprocal and inverse square root approximation coefficients. The output of the accumulation tree is the

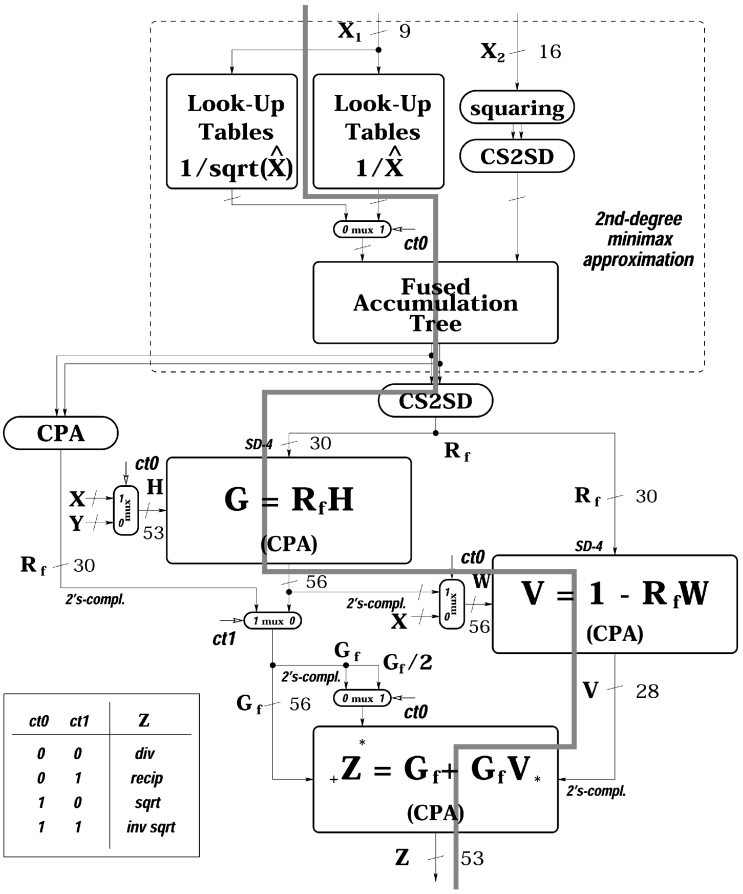

Fig. 3. Reciprocal/division/aquare root/inverse aquare root unit.

SD representation of $R_f = R_d$ or $R_f = R_s$ depending on the control signal $ct0$.

The computation of G_s requires a 30×53 bit multiplication, as the one in the reciprocal/division architecture. Since R_f is selected by $ct0$ and $G = R_f H$, another multiplexer controlled by $ct0$ is needed to select between $H = Y$ and $H = X$ as multiplicand operand. The output result is a 56-bit rounded to the nearest value in nonredundant representation.

A 30×56 bit multiply-complement unit is required for the computation of V. For the computation of square root/inverse square root, $V_s = 1 - R_s G_s$, while, for reciprocal/division computation, we had $V_d = 1 - R_d X$. Since $R_f = R_d$ or $R_f = R_s$ is already selected at the output of the accumulation tree, it is possible to use the same control signal $ct0$ to select between $W = X$ and $W = G_s$ as multiplicand operand for this unit, as shown in Table 1.

TABLE 1
Computations Performed in the Complete Unit Depending on the Operation

Operation	$G = R_f H$	$V = 1 - R_f W$	$Z = G_f + G_f V$
division	$G_d = R_d Y$	$V_d = 1 - R_d X$	$G_d + G_d V_d$
reciprocal	$[R_d Y]$	$V_d = 1 - R_d X$	$R_d + R_d V_d$
sqrt	$G_s = R_s X$	$V_s = 1 - R_s G_s$	$G_s + G_s V_s/2$
inv. sqrt	$G_s = R_s X$	$V_s = 1 - R_s G_s$	$R_s + R_s V_s/2$

The final $28 \times 56 + 56$ bit multiply-add operation required for the computation of Z is $Z = G_f + G_f V_s/2$ for the square root/inverse square root computation, while we had $Z = G_f + G_f V_d$ for reciprocal/division computation. Since the computation of V is on the critical path of the new architecture, the best solution is inserting multiplexers which select between G_f and $G_f/2$ as the multiplicand and addend operand. G_f takes the value of R_f or G, depending on the control signal $ct1$, and the input operand to the unit is selected between G_f or $G_f/2$, depending on the control signal $ct0$. Therefore, as shown in Fig. 3 and Table 1, the combination of the control signals $ct0$ and $ct1$ allows the computation of $Z = 1/X$, $Z = Y/X$, $Z = \sqrt{X}$, or $Z = 1/\sqrt{X}$.

The critical path of this architecture consists of the second-degree approximation look-up tables, a multiplexer to select between reciprocal and inverse square root approximation coefficients, the fused accumulation tree with CS result, the CS to SD recoding unit, a 30×53 bit multiplier (G computation), a 30×56 bit multiplier-complement unit (V computation), and a $28 \times 56 + 56$ bit multiplier-add unit (Z computation). The execution time of this unit is slower than the one for reciprocal/division architecture since the computation of V is now included in the critical path, but the hardware requirements remain similar due to the reutilization of the main logic blocks for the new computations.

TABLE 2
Area and Delay Estimates for the Main Logic Blocks Employed in the Compared Architectures (Explained in [19])

Logic Block	Area (fa)	Delay (τ)
8x56 mult.	305	2 + 1.5 + 4= 7.5
10x56 mult.	400	2 + 2.5 + 4= 8.5
15x15 mult. (SD result)	165	2 + 3 = 5
15x30 mult.	360	2 + 3 + 4 = 9
16x46 mult.	565	2 + 3 + 4 = 9
16x56 mult.	680	2 + 3 + 4 = 9
28(30)x56 mult.	1350	2 + 4.5 + 4 = 10.5
43x44 mult.	1815	2 + 5.5 + 4 = 11.5
56x56 mult.	2600	2 + 6 + 4 = 12
Accum. Tree (CS result)	530	1.5 + 5.5 = 7
Specialized Squaring unit	70	3
7-8 input-bit tables	35 fa/Kb	4
9-11 input-bit tables	35 fa/Kb	5
14-15 input-bit tables	25 fa/Kb	8

4 Execution Time and Area Cost Estimates and Comparison

In this section, estimates of the execution time and the area costs of the proposed architectures are presented and a comparison with some of the most efficient previous multiplicative-based methods for the computation of reciprocal, division, square root, and inverse square root operations is outlined. These estimates are based on a rough model for the cost and delay of the main logic blocks employed, taken from [5], [26]. In this model, the unit employed for the delay estimates is τ, the delay of a complex gate, such as a full-adder, while the area cost estimates are expressed as a multiple of fa, the area of one full-adder. The interconnections between logic blocks are not included in this model. A detailed explanation of the assumptions made can be found in [19].

Table 2 shows the delay and area estimates for the main blocks employed in our design and the architectures to be compared. These logic blocks are conventional multipliers, look-up tables, the multioperand adder, and the specialized squaring unit employed in our minimax approximation, recoding units, multiplexers, and SD adders and multipliers. The conventional multipliers are supposed to employ SD-4 recoding [14] of the multiplier operand, a Wallace accumulation tree, and a final CPA assimilation stage, when needed. The delay estimates for the look-up tables are taken from a previous model [5] and have been validated by synthesis results obtained from implementation using a family of standard gates from the AMS 0.35μm CMOS library. These logic blocks are usually the most difficult ones to model since they cannot be easily described in terms of full-adders.

The actual execution times and hardware requirements of the compared methods depend both on the technology employed and on the actual implementation, but, since all the schemes employ similar logic blocks,[3] the relative values may express a general trend among the different designs and make a good first-order approximation to the actual execution times and area cost values.

4.1 Execution Time Estimates

Recalling that the critical path of the unit for reciprocal/division computation consists of the initial 9-input bit look-up table, the fused accumulation tree with CS output, a CS to SD recoding unit, a 30×53 bit multiplier (G_d computation), a 2:1 multiplexer, and a $28 \times 56 + 56$ bit multiply-add unit (Z computation), the execution time can be estimated as 34.5τ, as shown in Fig. 4.

The critical path for the complete unit includes an extra multiplexer to select between the reciprocal and inverse square root initial approximations and another 30×56 bit multiplication (V computation). Summarizing, this critical path consists of the initial look-up table, a 2:1 multiplexer, the fused accumulation tree with CS output, a CS to SD recoding unit, two 30×56 bit multipliers, another 2:1 multiplexer, and a $28 \times 56 + 56$ bit multiply-add unit and, therefore, its execution time can be estimated as 45τ, as shown in Fig. 4.

4.2 Area Cost Estimates

Recalling the size of the look-up tables and the combinational logic blocks employed in the architecture implementing our method for the computation of all four functions (reciprocal/division/square root/inverse square root), we can summarize its hardware requirements as two 9-input bit sets of look-up tables (initial approximation), a specialized squaring unit, a fused accumulation tree, two 30×56 bit multipliers, and a $28 \times 56 + 56$ bit multiply-add unit. One of the 30×56 bit multipliers (the multiply-complement unit employed for V computation) can have an important area reduction (about 30 percent) by taking advantage of the bound $|V| \leq 2^{-28}$. This property allows the elimination of the 28 most significant colums of the multiplication matrix.

According to the estimates given in Table 2, the total area required by our method can be estimated as around $7,345fa$, with a contribution of $3,080fa$ from the $11KB$ look-up tables and of $4,265fa$ from the combinational logic blocks employed, as shown in Fig. 5.

4.3 Comparison with Other Methods

Table 3 shows the execution time and area cost estimates for the following methods: Newton-Raphson traditional algorithm (NR), Wong and Goto (WG) [26], Ito, Takagi, and Yajima (ITY) [9], Ercegovac, Muller, Lang, and Tisserand (EMLT) [5], and AMD-K7 implementation (AMD) [17], together with the estimates for our architectures. All these methods provide results accurate to 1 ulp. For obtaining exactly rounded results, a remainder should be computed.[4] Fig. 4 details the execution time

3. All compared methods employ a similar number of tables and multipliers of similar sizes. Therefore, comparison results are relatively technology-independent, and the impact of considering interconnections between blocks can be neglected.

4. The AMD-K7 actual implementation includes this remainder computation and final rounding stage. For comparison reasons, the estimates shown here do not include both features and a nonpipelined (unfolded) architecture is considered.

Newton-Raphson

reciprocal: 5τ (table) + 3τ (CPA) + 2x9τ (16x53 mult.) + 2x10.5τ (30x56 mult.) = 47τ

division: **reciprocal** + 12τ (56x56 mult.) = 59τ

inv. sqrt: 5τ (table) + 3τ (CPA) + 3x9τ (16x53 mult.) + 3x10.5τ (30x56 mult.) = 66.5τ

sqrt: **inv. sqrt** + 12τ (56x56 mult.) = 78.5τ

Wong & Goto

reciprocal and **division:** 5τ (table) + 3x8.5τ (10x56 mult.) + 4τ (CPA) + 12τ (56x56 mult.) = 46.5τ

inv. sqrt and **sqrt:** 5τ (table) + 5x8.5τ (11x56 mult.) + 4τ (CPA) + 12τ (56x56 mult.) = 63.5τ

Ito, Takagi & Yajima

reciprocal: 4τ (table) + 7.5τ (8x56 mult.) + 2τ (CPA) + 3x12τ (56x56 mult.) = 49.5τ

division: **reciprocal** + 12τ (56x56 mult.) = 61.5τ

inv. sqrt: not available

sqrt: 5τ (table) + 4τ (table) + 4x12τ (56x56 mult.) = 57τ

Ercegovac, Lang, Muller & Tisserand

reciprocal: 8τ (R_0 comput.) + 9τ (16x56 mult.) + 2x5τ (15x15 mult., SD result) + 4τ (SDA) + 4τ (CPA) + 9τ (16x46 mult.) = 44τ

division: **reciprocal** + 12τ (56x56 mult.) = 56τ

inv. sqrt and **sqrt:** 8τ (R_0 comput.) + 9τ (16x56 mult.) + 2x5τ (15x15 mult., SD result) + 4τ (SDA) + 1.5τ (recode) + 11.5τ (43x44 mult.) = 44τ

AMD-K7

reciprocal: 5τ (table) + 3τ (CPA) + 4x12τ (56x56 mult.) = 56τ

division: 5τ (table) + 3τ (CPA) + 5x12τ (56x56 mult.) = 68τ

inv. sqrt: 5τ (table) + 3τ (CPA) + 6x12τ (56x56 mult.) = 80τ

sqrt: 5τ (table) + 3τ (CPA) + 7x12τ (56x56 mult.) = 92τ

Our method

reciprocal and **division:** 5τ (table) + 7τ (fused accum. tree, CS result) + 1.5τ (recode) + 10τ (30x56 mult.) + 0.5τ (mux) + 10.5τ (28x56 mult.) = 34.5τ

inv. sqrt and **sqrt:** 5τ (table) + 0.5τ (mux) + 7τ (fused accum. tree, CS result) + 1.5τ (recode) + 10τ (30x56 mult.) + 0.5τ (mux) + 10τ (30x56 mult.) + 10.5τ (28x56 mult.) = 45τ

Fig. 4. Execution time estimates for the compared methods.

estimates for these methods, while Fig. 5 gives an explanation of the area cost ones.

In the NR algorithm that we are considering for comparison, bipartite tables [2] are employed to obtain a seed value and then two NR iterations are performed. The WG method employs a polynomial approximation combined with table look-up to obtain double-precision results. The ITY method employs a linear approximation to obtain the initial value and a single iteration of a third-degree convergent algorithm is performed, involving three 56×56 bit multiplications. In ELMT, *small* multipliers (15×15 and 15×45 bit multipliers) are employed, after an initial reduction stage, to compute the functions. The main drawback of this method is the size of the required look-up tables, which makes its hardware requirements prohibitive (its area cost is about three times bigger than the other ones). The AMD-K7 method is an implementation of the traditional Goldschmidt algorithm: It employs bipartite tables to obtain the initial approximation and then computes two Goldschmidt iterations to calculate the double-precision results. A detailed description of any of these methods can be found in [19].

Hardware reutilization is a strategy employed for some of the previous methods (NR, WG, ITY, and AMD), and can also be employed for our algorithm, allowing a significant reduction in the total area at the expense of increasing the

Newton-Raphson

Tables $(2^{10} \times (16+7) + 2^{11} \times (16+7)) +$
16x53 mult. + 30x56 mult. + 56x56 mult. =
Tables $(69Kb) + 600fa + 1350fa + 2600fa =$
$2415fa + 4550fa =$ **6965** fa

Wong & Goto

Tables $(2^{10} \times 10 + 2^9 \times 9 + 2^{11} \times 22 + 2^{11} \times 11 + 2^9 \times 9) +$
$(2\times)$ 10x56 mult. + 56x56 mult.=
Tables $(85Kb) + (2\times)\, 400fa + 2600fa =$
$2975fa + 3400fa =$ **6375** fa

Ito, Takagi & Yajima

Tables $(2^7 \times 14 + 2^{10} \times 119 + 2^8 \times 8)$ + 8x56 mult. + 56x56 mult. =
Tables $(123Kb) + 305fa + 2600fa =$
$4305fa + 2905fa =$ **7210** fa

Ercegovac, Lang, Muller & Tisserand

Tables $(2^{10} \times (16+7) + 2^{11} \times (56+27))$ + 16x56 mult. +
$(3\times)$ 15x15 SD-mult. + SDA3 + SDA5 + 43x46 mult. + 56x56 mult.=
Tables $(355Kb) + 680fa + (3\times)\, 165fa + 225fa + 165fa + 1815fa + 2600fa =$
$12425fa + 5980fa =$ **18405** fa

AMD-K7

Tables $(2^{10} \times (16+7) + 2^{11} \times (16+7))$ + 56x56 mult. =
Tables $(69Kb) + 2600fa =$
$2415fa + 2600fa =$ **5015** fa

Our method

Tables $(2^9 \times 62 + 2^{10} \times 57)$ + squaring unit + accum. tree +
30x53 reduced mult. + $(2\times)$ 30x56 mult. =
Tables $(88Kb) + 70fa + 530fa + (2\times)\, 1350fa + 965fa =$
$3080fa + 4265fa =$ **7345**fa

Our method: single multiply–add unit

Tables $(88Kb)$ + squaring unit + accum. tree + 30x56 mult. =
Tables $(11KB) + 70fa + 530fa + 1350fa =$
$3080fa + 1950fa =$ **5030**fa

Fig. 5. Area estimates for the compared methods.

control logic. G, V, and Z computations can be carried out by a single $30 \times 56 + 56$ bit multiply-add unit if the inputs of this unit are correctly selected by a set of extra control signals. Thus, if this multiplier computes the generic operation $A \times B + C$:

$$\begin{aligned} G &= R_f H, \\ V &= 1 - R_f W, \\ Z &= G_f + G_f V \end{aligned}$$

can be computed, making $A = R_f$, $B = H$, and $C = 0$ for G computation, $A = -R_f$, $B = W$, and $C = 1$ for V computation, and $A = V$, $B = G_f$, and $C = G_f$ for Z computation. This strategy leads to a reduced total area of about $5,030fa$ ($3,080fa$ for the look-up tables, $600fa$ for the accumulation tree and the squaring unit of the minimax approximation, and $1,350fa$ for the reused $30 \times 56 + 56$ multiply-add unit), as shown in Fig. 5.

TABLE 3
Execution Times and Area Cost Estimates for the Compared Methods

Method	Reciprocal	Division	Inv. Sqrt	Sqrt	Total Area (fa)
NR	47	59	66.5	78.5	6965
WG	46.5	46.5	63.5	63.5	6375
ITY	49.5	61.5	-	57	7210
ELMT	44	56	**44**	**44**	18405
AMD	56	68	80	92	**5015**
Our method	**34.5**	**34.5**	**45**	**45**	7345 (**5030**)*

$*$ when employing a single multiplier

Our method presents a speed-up of about 25 percent over the previous methods when computing the reciprocal function. The speed-up increases to about 40 percent when computing division, due to the fact that most methods require an extra multiplication for division computation, while our method and WG provide the division result with the same execution time as for the reciprocal computation. Our execution times for square root and inverse square root

computation are also faster than the previous ones (with the only exception of ELMT, but the area requirements of this method are prohibitive). AMD is the method with lower hardware requirements (taking into account that it employs an existing multiplier, as NR and ITY). However, our method has a similar area cost, especially if we employ a single multiply-add unit to compute G, V, and Z.

As has been said, the actual speed-ups and area ratios depend on the technology employed and the implementation, but the estimated values presented here are reliable approximations since all compared methods employ similar logic blocks and look-up tables of similar sizes (except for ELMT). This means that the speed-ups obtained come directly from the reduction on the latency achieved by performing a single Goldschmidt iteration.

5 Conclusion

A new method for the efficient computation of double-precision floating-point reciprocal, division, square root, and inverse square root functions has been presented in this paper. This method employs a second-degree minimax polynomial approximation, with table look-up, a specialized squaring unit, and a multioperand adder, to obtain an accurate (around 30 bits) initial approximation R_f of the reciprocal and the inverse square root values. The specialized squaring unit and the accumulation tree are shared for the computations of the reciprocal (R_d) and inverse square root (R_s) initial approximations. The size of the employed look-up tables is $11KB$.

After obtaining R_f, a modified Goldschmidt iteration is performed to obtain the double-precision result. A single iteration is required, due to the high accuracy of the initial approximation, which significantly reduces the latency of the algorithm. The modified Goldschmidt iterations for computing reciprocal/division and for computing square root/inverse square root have been combined to allow the design of a single architecture efficiently computing the four operations. A faster architecture computing only reciprocal and division has also been proposed.

A rough model for the delay and area cost of the main logic blocks employed has been outlined and the execution time and area costs for both architectures have been estimated: 34.5τ for reciprocal/division computation and 45τ for square root/inverse square root computation. The total amount of area required for the complete architecture is about $5,030fa$ when employing hardware reutilization strategies.

For comparison purposes, other efficient methods for the computation of these functions have been described and their execution times and area costs have also been estimated. Comparison results show that our method achieves an important speed-up over previous methods, with similar hardware requirements.

Acknowledgments

The authors wish to thank Dr. Jean-Michel Muller for his fundamental contribution to the development of the second-degree minimax approximation method. This work was supported by the Secretaría Xeral de Investigación e Desenvolvemento de Galicia (Spain) under contract PGIDT99PXI20602B.

References

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

[2] D. DasSarma and D.W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th Symp. Computer Arithmetic,* pp. 17-28, 1995.

[3] M.D. Ercegovac, L. Imbert, D.W. Matula, J.M. Muller, and G. Wei, "Improving Goldschmidt Division, Square Root and SquareRoot Reciprocal," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 759-763, July 2000.

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

[5] M.D. Ercegovac, T. Lang, J.M. Muller, and A. Tisserand, "Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 628-637, July 2000.

[6] P.M. Farmwald, "High Bandwidth Evaluation of Elementary Functions," *Proc. Fifth IEEE Symp. Computer Arithmetic,* pp. 139-142, 1981.

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

[8] D. Harris, S. Oberman, and M. Horowitz, "SRT Division Architectures and Implementations," *Proc. IEEE 13th Int'l Symp. Computer Arithmetic (ARITH13),* pp. 18-25, 1997.

[9] M. Ito, N. Takagi, and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root," *Proc. 12th Symp. Computer Arithmetic (ARITH12),* pp. 2-9, 1995.

[10] 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.

[11] I. Koren, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Trans. Computers,* vol. 39, pp. 1030-1037, 1990.

[12] H. Kwan, R.L. Nelson, and E.E. Swartzlander Jr., "Cascaded Implementation of an Iterative Inverse-Square Root Algorithm with Overflow Lookahead," *Proc. 12th Symp. Computer Arithmetic,* pp. 114-123, 1995.

[13] T. Lang and P. Montuschi, "Very-High Radix Square Root with Prescaling and Rounding and a Combined Division/Square Root Unit," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 827-841, Aug. 1999.

[14] C.N. Lyu and D.W. Matula, "Redundant Binary Booth Recoding," *Proc. 12th Symp. Computer Arithmetic,* pp. 50-57, 1995.

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

[16] S. Oberman and M.J. Flynn, "Implementing Division and Other Floating Point Operations: A System Perspective," *Scientific Computing and Validated Numerics,* pp. 18-24, 1996.

[17] 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.

[18] S.F. Oberman and M.J. Flynn, "Design Issues in Division and Other Floating Point Operations," *IEEE Trans. Computers,* vol. 46, no. 2, pp. 154-161, Feb. 1997.

[19] J.A. Piñeiro and J.D. Bruguera, "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root," technical report, http://www.ac.usc.es, 2001.

[20] J.A. Piñeiro, J.D. Bruguera, and J.M. Muller, "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree," *Proc. IEEE 15th Int'l Symp. Computer Arithmetic (ARITH15),* pp. 40-47, 2001.

[21] M.J. Schulte and J.E. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation" *Proc. 13th Symp. Computer Arithmetic (ARITH13),* pp. 175-183, 1997.

[22] P. Soderquist and M. Leeser, "Area and Performance Tradeoffs in Floating Point Divide and Square Root Implementations," *ACM Computer Surveys,* pp. 518-564, 1996.

[23] 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.

[24] P.T.P. Tang, "Table Look-Up Algorithms for Elementary Functions and their Error Analysis," Argonne Nat'l Laboratory Report, MCS-P194-1190, Jan. 1991.
[25] Waterloo Maple Inc., *Maple V Programming Guide,* 1998.
[26] W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations," *IEEE Trans. Computers,* vol. 43, no. 3, pp. 278-294, Mar. 1994.

José-Alejandro Piñeiro received the BSc degree in 1998 and the MSc degree in 1999, both in physics (electronics) from the University of Santiago de Compostela, Spain. He is currently a PhD candidate in the Department of Electronic and Computer Engineering at the University of Santiago de Compostela. His research interests are in the areas of computer arithmetic, VLSI design, and computer graphics. He is a student member of the IEEE and the IEEE Computer Society.

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