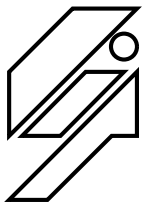


Laboratoire de l'Informatique du Parallélisme

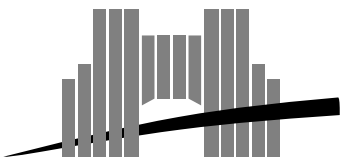
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Reciprocation, Square root, Inverse Square Root, and some Elementary Functions using Small Multipliers

Miloš D. Ercegovac
Tomas Lang
Jean-Michel Muller
Arnaud Tisserand

November 1997

Research Report N^{O} 97-47



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Reciprocation, Square root, Inverse Square Root, and some Elementary Functions using Small Multipliers

Miloš D. Ercegovac
Tomas Lang
Jean-Michel Muller
Arnaud Tisserand

November 1997

Abstract

This paper deals with the computation of reciprocals, square roots, inverse square roots, and some elementary functions using small tables, small multipliers, and for some functions, a final "large" (almost full-length) multiplication. We propose a method that allows fast evaluation of these functions in double precision arithmetic. The strength of this method is that the same scheme allows the computation of all these functions.

Our method is mainly interesting for designing special purpose circuits, since it does not allow a simple implementation of the four rounding modes required by the IEEE-754 standard for floating-point arithmetic.

Keywords: Division, Reciprocation, Square-root, Function evaluation, Computer arithmetic

Résumé

Ce rapport traite de l'évaluation d'inverses, de racines carrées, d'inverses de racines carrées, et de quelques fonctions élémentaires en utilisant de petites tables, de petits multiplieurs, et, pour quelques fonctions, une "grande" (i.e., portant sur des nombres dont la taille est celle des opérandes) multiplication finale. Nous proposons une méthode qui permet l'évaluation de ces fonctions en arithmétique double précision. L'avantage de cette méthode réside dans le fait que le même schéma de calcul (et donc, en pratique, la même implantation) permet le calcul de toutes ces fonctions.

Notre méthode est essentiellement intéressante pour construire des circuits dédiés à une application, car elle ne permet pas d'implanter de manière simple les quatre modes d'arrondis exigés par le standard IEEE pour l'arithmétique flottante.

Mots-clés: Division, Inverse, Racine Carrée, Évaluation de fonctions, Arithmétique des ordinateurs

1 Introduction

For many years, only two classes of methods have been considered when implementing division and square root: digit-recurrence methods [9], and quadratically converging methods, such as Newton's method and Goldschmidt's iteration [1]. Concerning elementary functions, the methods that have mainly been used are shift-and-add, Cordic-like methods [14, 15], and polynomial or rational approximations [5, 2]. A noticeable exception is a method suggested by Farmwald [4], that already uses tables. The progress in VLSI technology now allows the use of large tables, that can be accessed quickly. As a consequence, many table-based methods have emerged during the last decade: high-radix digit-recurrence methods for division and square root [3], mix-up of table-lookup and polynomial approximation for the elementary functions [11, 12, 13], or even (for single precision) use of table-lookups and addition only [17, 6, 10]. Recent overviews on these issues can be found in references [8, 7].

For instance, Wong and Goto [17] recently suggested to evaluate the elementary functions, the reciprocals and the square-root using tables, without multiplications. Assume that the input number is $A_1 z + A_2 z^2 + A_3 z^3 + A_4 z^4$, where $z = 2^{-k}$ and the $A_i's$ are integers less than 2^k. They compute $f(A)$ using the following formula

$$\begin{aligned} f(A) \approx\ & f(A_1 z + A_2 z^2) \\ & + \frac{z^2}{2}\left(f(A_1 z + (A_2 + A_3) z^2) - f(A_1 z + (A_2 - A_3) z^2)\right) \\ & + \frac{z^3}{2}\left(f(A_1 z + (A_2 + A_4) z^2) - f(A_1 z + (A_2 - A_4) z^2)\right) \\ & + z^4 \left(\tfrac{A_3^2}{2} f^{(2)}(A_1 z) - \tfrac{A_3^3}{6} f^{(3)}(A_1 z)\right). \end{aligned}$$

Roughly speaking, their method allows to evaluate a function with approximately n bits of accuracy by just performing look-ups in $\frac{n}{2}$-bit address tables, and additions. This makes their method attractive for single-precision calculations. In [16], Wong and Goto also suggest, for double precision calculations, the use of several look-ups in 10-bit address tables and some multiplications that require rectangular multipliers (typically, 16×56-bit multipliers) only.

The methods suggested by Farmwald [4], and Das Sarma and Matula [10] require the use of tables with approximately $n/3$ address bits to get an n-

bit approximation of the function being evaluated. With current technology, this makes these methods impossible to implement for double-precision arithmetic.

In this paper, we propose a new class of algorithms that allows the evaluation of reciprocals, square roots, inverse square roots and some elementary functions, using one table access, a few "small" multiplications, and at most one "large" multiplication. To approximate a function with n-bit accuracy, we need tables with approximately $n/4$ address bits.

2 Reciprocal, Square Root, and Inverse Square Root

We want to evaluate reciprocals, square roots and inverse square roots for operands and results represented by an n-bit significand. We do not consider the computation of the exponent since this is straightforward. Let us call the generic computation $g(Y)$, where Y is the significand and, as in the IEEE standard, $1 \leq Y < 2$.

The method is based on the Taylor expansion of the function to compute, which converges with few terms if the argument is close to 1. Consequently, the method consists of the following three steps:

1. **Reduction.** From Y we deduce a number A such that $-2^{-k} < A < +2^{-k}$. To produce a simple implementation that achieves the required precision, we use $k = n/4$. For the functions considered, we obtain A as

$$A = Y \times \hat{Y} - 1$$

where $\hat{Y}$ is a $(k+1)$-bit approximation of $1/Y$. Specifically, define $Y^{(k)}$ as Y truncated to the kth bit. Then

$$Y^{(k)} \leq Y < Y^{(k)} + 2^{-k}$$

Hence

$$1 \leq \frac{Y}{Y^{(k)}} < 1 + 2^{-k} \tag{1}$$

Using one lookup in a k-bit address table, one can find the number $\hat{Y}$ defined as $1/Y^{(k)}$ rounded *down* (i.e., truncated) to $k+1$ bits. Then,

$$-2^{-k-1} < \hat{Y} - \frac{1}{Y^{(k)}} \leq 0$$

Therefore, since $1 \leq Y^{(k)} < 2$,

$$1 - 2^{-k} < \hat{Y} Y^{(k)} \leq 1. \tag{2}$$

Using (1) and (2), we get

$$1 - 2^{-k} < \hat{Y} Y < 1 + 2^{-k} \tag{3}$$

The *reduced argument* A is such that $g(Y)$ can be easily obtained from a value $f(A)$, that is computed during the next step;

2. **Evaluation.** We compute an approximation of $B = f(A)$ using the series expansion of f, as described below.

3. **Post-processing.** This is required because of the reduction step. Since reduction is performed by multiplication by $\hat{Y}$, we obtain $g(Y)$ from $B = f(A)$ as

$$g(Y) = M \times B$$

where $M = h(\hat{Y})$. The value of M depends on the function and is obtained by a similar method as $\hat{Y}$. Specifically,

- For reciprocal $M = \hat{Y}$
- For square root $M = 1/\sqrt{\hat{Y}}$
- For inverse square root $M = \sqrt{\hat{Y}}$

Let us now consider the evaluation step.

2.1 Evaluation step

In the following, we assume that we want to evaluate $B = f(A)$, with $|A| < 2^{-k}$. The Taylor series expansion of F is

$$f(A) = C_0 + C_1 A + C_2 A^2 + C_3 A^3 + C_4 A^4 + \ldots, \tag{4}$$

where the $C_i's$ are bounded.

Since $-2^{-k} < A < 2^{-k}$, A has the form

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 \tag{5}$$

where $z = 2^{-k}$, $k = n/4$ and $|A_i| \leq 2^k - 1$.

Our goal is to compute an approximation of $f(A)$, correct to approximately $n = 4k$ bits, using small multiplications. From the series (4) and the decomposition (5) we deduce

$$\begin{array}{c} f(A) = C_0 + C_1 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right) + C_2 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^2 \\ + C_3 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^3 + C_4 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^4 + \ldots \end{array} \tag{6}$$

After having expanded this series and dropped out all the terms of the form $W \times z^j$ that are less than or equal to 2^{-4k}, we get (see Appendix)

$$f(A) \approx C_0 + C_1 A + C_2 A_2^2 z^4 + 2 C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6 . \tag{7}$$

We use this last expression to approximate reciprocals, square roots and inverse square roots. In practice, when computing (7), we make another approximation: after having computed A_2^2, obtaining A_2^3 would require a $2k \times k$ multiplication. Instead of this, we take only the k most-significant bits of A_2^2 and multiply them by A_2.

In the Appendix, we prove the following result:

Theorem 1 *f(A) can be approximated by*

$$C_0 + C_1 A + C_2 A_2^2 z^4 + 2 C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6 ,$$

(where we use the most k significants bits[1] of A_2^2 only when computing A_2^3), with an error less than

$$2^{-4k} \left(\frac{C_{max}}{1 - 2^{-k}} + 4|C_2| + 4|C_3| + 9 \max\{C_2, C_3\} \times 2^{-k} \right)$$

with $C_{max} = \max_{i \geq 4} |C_i|$.

In particular, for $k \geq 7$, and assuming $|C_i| \leq 1$ for any i (which is satisfied for the functions considered in this paper), this error is less than

$$\epsilon = 2^{-4k} \left(C_{max} + 4|C_2| + 4|C_3| + \frac{1}{10} \right) .$$

[1] It would be more accurate to say *digits*, since it is likely that in a practical implementation, A_2^2 will be represented in a redundant (e.g., carry-save or borrow-save) representation.

Now we determine the coefficients and the error bound for the three functions. Since

$$\begin{aligned}
\frac{1}{1+x} &= 1 - x + x^2 - x^3 + x^4 - \ldots \\
\sqrt{1+x} &= 1 + \tfrac{1}{2}x - \tfrac{1}{8}x^2 + \tfrac{1}{16}x^3 - \tfrac{5}{128}x^4 + \ldots \\
\frac{1}{\sqrt{1+x}} &= 1 - \tfrac{1}{8}x + \tfrac{3}{8}x^2 - \tfrac{5}{16}x^3 + \tfrac{35}{128}x^4 - \ldots
\end{aligned}$$

we get

- For reciprocal

$$\begin{aligned}
\frac{1}{1+A} &\approx 1 - A_2 z^2 - A_3 z^3 + \left(-A_4 + A_2^2\right) z^4 + 2A_2 A_3 z^5 - A_2^3 z^6 \\
&= (1 - A) + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6
\end{aligned} \tag{8}$$

  and the bound on the error is

$$\epsilon = 9.1 \times 2^{-4k}$$

- For square root

$$\sqrt{1+A} \approx 1 + \frac{A}{2} + \frac{1}{8}A_2^2 z^4 - \frac{1}{4}A_2 A_3 z^5 + \frac{1}{16}A_2^3 z^6 \tag{9}$$

  and the error bound is

$$\epsilon = 0.9 \times 2^{-4k}$$

- For inverse square root

$$1/\sqrt{1+A} \approx 1 - \frac{A}{2} + \frac{3}{8}A_2^2 z^4 + \frac{3}{4}A_2 A_3 z^5 - \frac{5}{16}A_2^3 z^6 \tag{10}$$

  and the error bound

$$\epsilon = 3.12 \times 2^{-4k}$$

3 Implementation

Now, let us suggest some ways of implementing our method. Fig. 1 presents a functional representation of the general architecture.

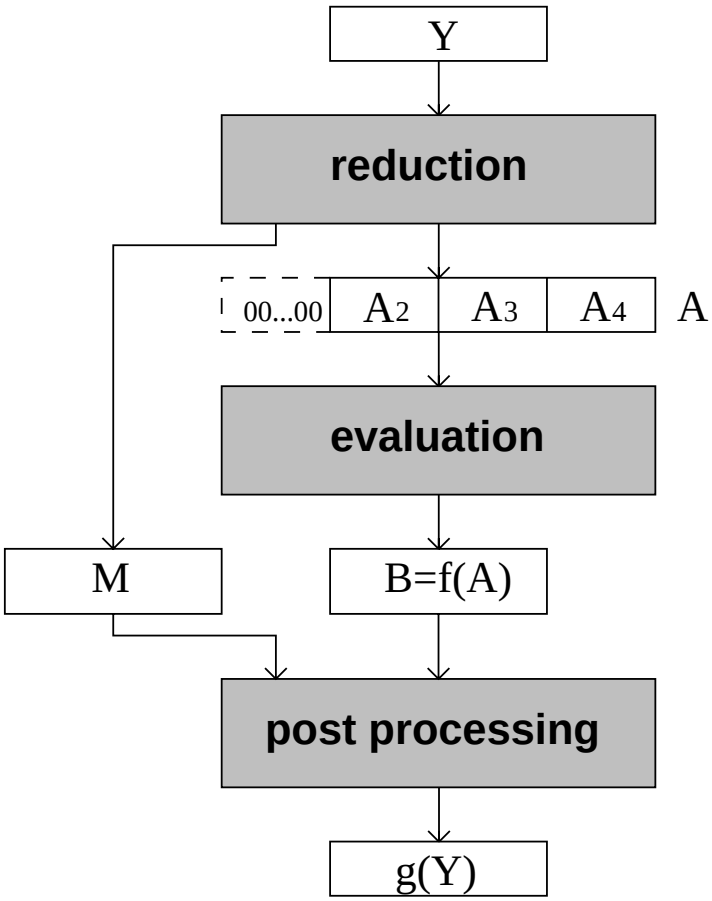


Figure 1: Functional representation of the general architecture

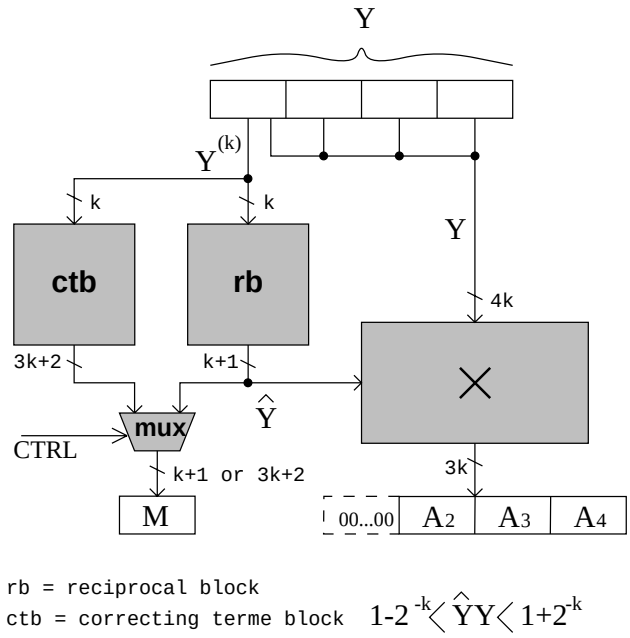


Figure 2: Functional representation of the reduction module (the value of M depends on the function being computed).

3.1 Reduction

Fig. 2 shows a functional representation of the reduction module. From Y, this module computes A and M. Different methods can be used to perform this computation, from direct table look-up to linear interpolation ...

3.2 Evaluation

The evaluation step computes expressions (8), (9), and (10). All three require the computation of A_2^2, $A_2 A_3$, and A_2^3. As indicated before, for A_2^3 we use the approximation

$$A_2^3 \approx (A_2^2)_{high} \times A_2$$

Consequently, these terms can be computed by three k by k multiplications. Moreover, the first two can be performed in parallel.

Alternatively, it is possible to compute the terms by two multiplications as follows:

- For reciprocal and for inverse square root
  1. $$B_1 = A_2^2 + 2A_2A_3z = A_2 \times (A_2 + 2A_3z)$$
  2. $$A_2^3 \approx (B_1)_{high} \times A_2$$
- For square root
  1. $$B_1 = A_2^2 - 2A_2A_3z = A_2 \times (A_2 - 2A_2A_3)$$
  2. $$A_2^3 \approx (B_1)_{high} \times A_2$$

The first of the two multiplications is of k by $2k$ bits and the second is of k by k.

Then the terms (either the output of the three multiplications or of the two multiplications) are multiplied by the corresponding factors, which depend on the function as shown in Table 1. Note that for division and square root these factors correspond just to alignments, whereas for inverse square root multiplications by 3 and 5 are required[2]. Finally the resulting terms are added to produce B.

Fig. 3 shows the weights of these terms (in the case of the reciprocal function). After this addition, the result is rounded to the nearest multiple of 2^{-4k}. As shown[3] in Figure 3, this gives a $3k+1$ number $\hat{B}$. Then B is equal to $\hat{B} + 1$.

[2]Obviously, these "multiplications" will be implemented as – possibly redundant – additions, since $3 = 2 + 1$ and $5 = 4 + 1$.

[3]In the case of the reciprocal function, but this is similar for the other ones.

Table 1: Factors to multiply terms

Function	B_1	A_2^3
Reciprocal	1	1
Square root	1/8	1/16
Inverse Square root	3/8	5/16

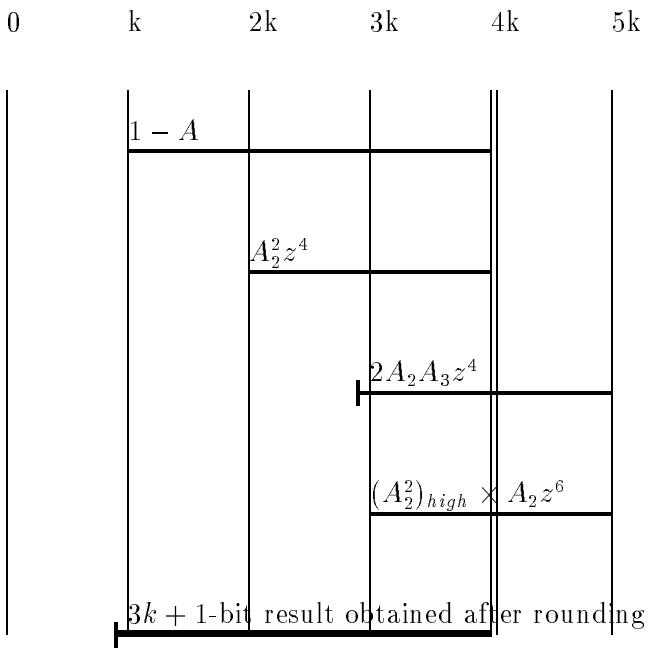


Figure 3: Weights of the various terms that are added during the evaluation step for reciprocal. After the summation, the result is rounded to nearest, so that there are no longer terms of weights less than 2^{-4k}.

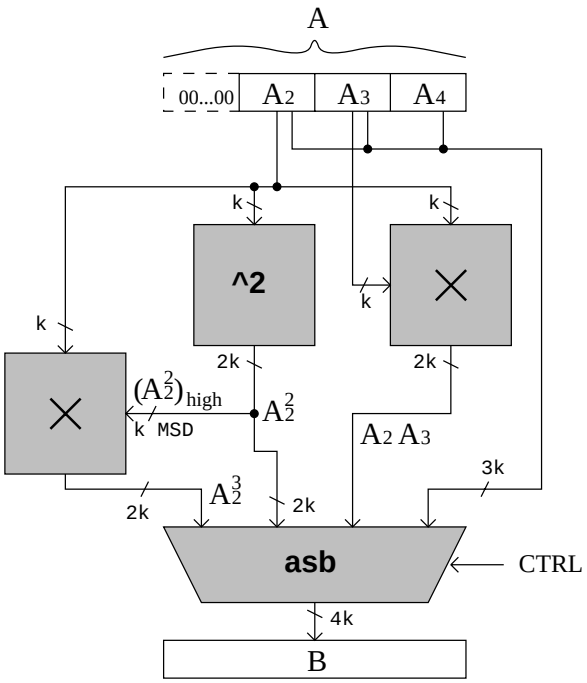


Figure 4: Functional representation of the evaluation module (As told above, the computations of A_2^2 and A_2A_3 can be regrouped in one rectangular multiplication).

function	value of M	size of M	operation
$1/Y$	$\hat{Y}$	$k+1$ bits	$M \times B$
$\sqrt{Y}$	$1/\hat{Y}$	n but $3k+g$ bits only for mult.	$M \times B$
$1/\sqrt{Y}$	$\hat{Y}$	n but $3k+g$ bits only for mult.	$M \times B$

Table 2: Operation being performed during the post-processing step

3.3 Post-Processing

The post-processing consists in multiplying B by M, where $M = g(\hat{Y})$ depends on the function and is computed during the reduction step. Since $B = 1+\hat{B}$ and $\hat{B} < 2^{-k+1}$, to use a smaller multiplier it is better to compute

$$g(Y) = M \times B = M + M \times \hat{B}$$

Note also that in the multiplication it suffices to use the bits of M of weight larger than or equal to 2^{-3k-g}, where g is a small integer. Since the error due to this truncation is smaller than or equal to $2^{-4k+1-g}$, choosing $g = 2$ makes the error bounded by 0.5×2^{-4k} and allows the use of a $(3k+1)\times(3k+2)$-bit multiplier. From the error bounds given in Section 2 and taking into account the additional 0.5×2^{-4k} error due to the use of a $(3k+1) \times (3k+2)$ multiplier for the post-processing step, we suggest to choose $n = 56$ and $k = 14$ for a double-precision implementation.

Table 2 shows the operation that must be performed during the post-processing step, and the value of M that must be used.

4 Comparison with other methods

4.1 High-radix digit-recurrence division

Assume we wish to compute X/Y. Radix-r digit-recurrence division [9] consists in performing the recurrence:

$$X^{(j+1)} = r \times X^{(j)} - q_{j+1}Y, \tag{11}$$

where $X^{(0)} = X$, $X^{(j)}$ is the j-th residual and q_{j+1} is the $j+1$-st radix-r digit of the quotient. When performing a *high-radix* digit-recurrence division [3], Y is first *normalized* (i.e., multiplied[4] by a factor $\hat{Y}$ so that $\hat{Y}$ is

[4]This is exactly the same step as our reduction step.

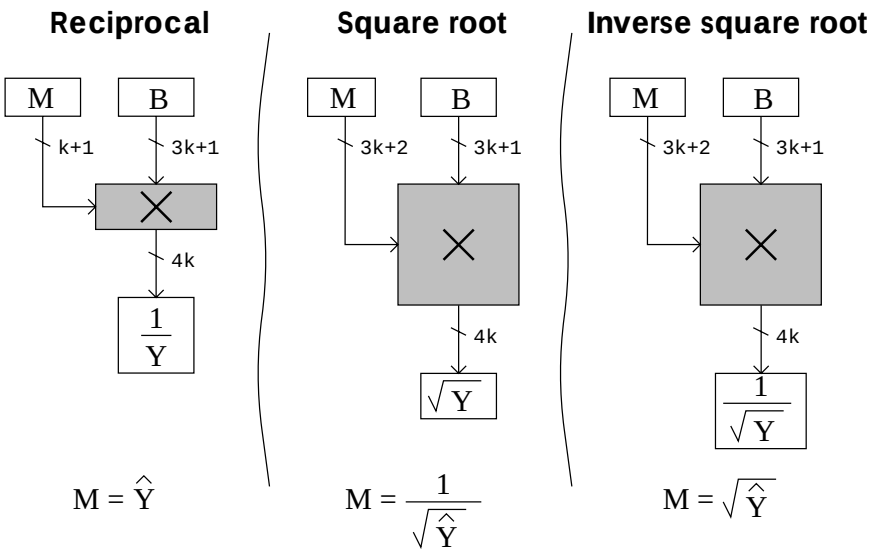


Figure 5: Functional representation of the post processing module

very close to 1). This allows a simple selection of q_{j+1}. Assume here that we perform a radix-2^k digit-recurrence division, where $k = n/4$. To get n-bit accuracy, we will need to perform 4 iterations. If we assume that our goal is double-precision arithmetic, we will therefore need, after the normalization, four consecutive 14×52-bit multiplications. This is very similar to what is required by our method. Therefore, if the issue at stake is *reciprocation*, our method and the high-radix recurrence method have close performances. On the other hand, if we want to perform *divisions*, the high-radix recurrence method is preferable, since it does not require a final "large" multiplication.

4.2 Newton-Raphson and Goldschmidt iterations

The well known Newton-Raphson (NR) iteration for reciprocal

$$x_{n+1} = x_n \times (2 - Y x_n) \tag{12}$$

converges quadratically[5] to $1/Y$ provided that x_0 is close enough to $1/Y$. The usual way to implement this iteration is to first look x_0 up in a table. Assume that we use a k-bit address table, and that we perform the intermediate calculations using an n-bit arithmetic. To compare with our method, we assume $n \approx 4k$. The first approximation x_0 of $1/Y$ is the number $\hat{Y}$ of section 2. It is a k-bit approximation of $1/Y$. To get x_1, one need to perform two $k \times n$-bit multiplications. Since x_1 is a $2k$-bit approximation of $1/Y$, it suffices to use its most $2k$ significant bits to perform the next iteration[6]. After this, one needs to perform two $2k \times n$-bit multiplications to get x_2, which is an n-bit approximation[7] of $1/Y$. Assuming $k = 14$ and $n = 56$, the NR method requires:

- one lookup in a 14-bit address table;
- two 14×56-bit multiplications;
- two 28×56-bit multiplications.

The multiplications that occur cannot be performed in parallel.

The NR iteration for reciprocal square-root[8]

$$x_{n+1} = \frac{1}{2} x_n \left(3 - Y x_n^2\right) \tag{13}$$

has convergence properties very similar to those of the NR iteration for division. Assuming (as previously) that we use a k-bit address table, and that we perform the intermediate calculations using an n-bit arithmetic, with $k = 14$ and $n = 56$, computing an inverse square-root using the NR iteration requires:

- one lookup in a 14-bit address table;
- three 14×56-bit multiplications;

[5] That is, the number of common digits between x_{n+1} and $1/Y$ is approximately twice that between x_n and $1/Y$.

[6] To our knowledge, this property is only used in multiple-precision implementations of the division. Using it in FPU's would require the design of several different size multipliers.

[7] Getting a correctly rounded result would require another iteration.

[8] The well known iteration for division

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{Y}{x_n}\right)$$

cannot be used here, since it requires a division at each step. To compute $\sqrt{Y}$, it is much better to first compute $1/\sqrt{Y}$ using (13), and to multiply the result by Y.

- three 28×56-bit multiplications.

Computing a square-root requires the same number of operations, and a final "large" (56×56-bit) multiplication. This shows that, although only slightly more interesting than the NR iteration for computing reciprocals, our method becomes much more interesting than the NR method when we need to compute square roots and inverse square roots.

Goldschmidt's iteration [8] for computing X/Y:

$$\begin{aligned} N_{i+1} &= N_i \times R_i \\ D_{i+1} &= D_i \times R_i \\ R_{i+1} &= 2 - D_{i+1} \end{aligned} \tag{14}$$

with $N_0 = X$, $D_0 = Y$ and $R_0 = 1 - Y$, has a convergence rate similar to that of the NR iteration[9]. To make the iteration faster, one must replace Y by a value $Y\hat{Y}$ very close to 1, and therefore multiply the final value by $\hat{Y}$. This is exactly what we do during the normalization and post-processing steps of our method. Assuming that $\hat{Y}$ is read from a k-bit address table, Goldschmidt iteration requires the same number of steps as the NR iteration, but the two multiplications required at each step can be performed in parallel (or, merely, in pipeline). Thus, assuming $k = 14$ and $n = 56$, the iteration requires:

- one look-up in a 14-bit address table to get $\hat{Y}$;
- one 14×56-bit multiplication to get $Y\hat{Y}$;
- four 56×56-bit multiplications to perform the iterations (but if two multipliers are available, the time required is that of two multiplications);
- one 14×56-bit multiplication to get X/Y.

4.3 Wong and Goto's method

The method presented by Wong and Goto in [17] requires tables with $m/2$ address bits, where m is the number of bits of the mantissa of the floating-point arithmetic being used. This makes that method un-convenient for

[9] This is not surprising:the same iteration is hidden behind both methods [8].

double-precision calculations. In [16], they suggest another method, that requires table-lookups and rectangular[10] multipliers.

Their method for computing reciprocals is as follows. Let us start from the input value $Y = 1.y_2y_2 \ldots y_{10}$. The first 10 bits of Y are used address bits to get from a table

$$r_0 = \left\lfloor \frac{1}{1.y_1y_2 \ldots y_{10}} \right\rfloor .$$

Then, they compute $r_0 \times Y$. This gives a number A of the form:

$$A = 1 - 0.000 \ldots 0a_9a_{10} \ldots a_{18} \ldots a_{56}$$

Then, using a rectangular multiplier, they compute:

$$\begin{aligned} B &= A \times (1 + 0.000 \ldots 0a_9a_{10} \ldots a_{18}) \\ &= 1 - 0.000000 \ldots 00b_{17}b_{18} \ldots b_{26} \ldots b_{56} \end{aligned}$$

Again, using a rectangular multiplier, they compute:

$$\begin{aligned} C &= B \times (1 + 0.000000 \ldots 00b_{17}b_{18} \ldots b_{26}) \\ &= 1 - 0.00000000000 \ldots 0000c_{25}c_{26} \ldots c_{56} \end{aligned}$$

After this (or, merely, during this), the bits $b_{27}b_{18} \ldots b_{35}$ are used as address bits to get from a table the number β constituted by the most 9 significant bits of $(0.0000 \ldots b_{27}b_{18} \ldots b_{35})^2$. The final result is:

$$\begin{aligned} \frac{1}{Y} \approx r_0 \quad &\times \quad 1.00000 \ldots a_9a_{10} \ldots a_{18} \\ &\times \quad 1.000000 \ldots 00b_{17}b_{18} \ldots b_{26} \\ &\times \quad (1.000000000 \ldots 000c_{25}c_{26} \ldots c_{56} + \beta) \end{aligned} \tag{15}$$

Wong and Goto's method for reciprocation therefore requires one look-up in a 10-bit address table, one look-up in a 9-bit address table, and six rectangular 10×56 multiplications. One can reasonably assume that their rectangular multiplication have approximately the same cost as our $k \times n = 14 \times 56$ multiplications. Therefore their method requires more time than ours. To compute reciprocal square-roots, they need one look-up in a 11-bit address table, one look-up in a 9-bit address table, and nine rectangular multiplications, which is much more than what is needed with our method.

[10] Depending on the function being computed, their rectangular multipliers are between 10×56 and 16×56-bit multipliers.

5 Elementary functions

Using the same basic scheme, our method also allows computation of some of the elementary functions. We briefly describe this below. Implementation is not discussed: it is very similar to what we have previously described for reciprocal, square root and inverse square root.

5.1 Computation of logarithms

In a similar fashion, we get:

$$\ln(1+A) \approx A - A_2^2 z^4 - A_2 A_3 z^5 + \frac{1}{3} A_2^3$$

Again, we only need to compute A_2^2, $A_2 A_3$ and A_2^3. And yet, the 1/3 coefficient in front of A_2^3 may make this last approximation less interesting. The post-processing step reduces to an addition.

5.2 Computation of exponentials

Now, let us assume that we want to evaluate the exponential of an n-bit number $Y = 1 + A_1 z + A_2 z^2 + A_3 z^3 + A_4 z^4$, where $z = 2^{-k}$ $(k = n/4)$, and the A_i's are k-bit integers. We suggest to first computing the exponential of

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4$$

using a Taylor expansion, and then to multiply it by the number

$$M = \exp(1 + A_1 z).$$

M will be obtained by looking up in a k-bit address table.

The exponential of A can be approximated by:

$$1 + A + \frac{1}{2} A_2 z^4 + A_2 A_3 z^5 + A_2^3 z^6 \tag{16}$$

5.3 Sine and Cosine functions

Using the same number A as for the exponential function, we use the approximations:

$$\begin{cases} \cos(A) & \approx & 1 - A_2^2 z^4 - A_2 A_3 z^5 \\ \sin(A) & \approx & A - \frac{1}{6} A_2^3 z^6 \end{cases} \tag{17}$$

Function	table size (bits)	"small" mult.	"large" mult.
reciprocal	$(k+1) \times 2^k$	5 (2 are done in parallel)	0
square-root	$(k+1+n) \times 2^k$	4 (2 are done in parallel)	1
inv. sqrt	$(k+1+n) \times 2^k$	4 (2 are done in parallel)	1
logarithm	$(k+1+n) \times 2^k$	4 (2 are done in parallel)	0
exponential	$n \times 2^k$	3 (2 are done in parallel)	0
sin/cos	$2n \times 2^k$	3 (2 are done in parallel)	4

Table 3: Table sizes and number of various operations required by our method, depending on the function being computed. Here, we call "small" multiplication a $k \times n$ or $k \times k$ multiplication, and "large" multiplication a $(3k+1) \times (3k+2)$ multiplication.

After this, if $M_1 = \sin(1 + A_1 z)$ and $M_2 = \cos(1 + A_1 z)$ are read from a k-bit address table, we get

$$\begin{aligned} \sin(Y) &= M_2 \sin(A) + M_1 \cos(A) \\ \cos(Y) &= M_2 \cos(A) - M_1 \sin(A) \end{aligned}$$

Therefore, for the sine and cosine functions, the post-processing step is more complex.

6 Conclusion

We have proposed a new method for computation of reciprocals, square-roots, inverse square-roots, logarithms, exponentials, sines and cosines. The strength of our method is that the same basic computations are performed for all these various functions. For reciprocation, our method will require a computational delay quite close to that of high-radix digit-recurrence or Newton-Raphson iteration. To get $52 + 1$ bits (double precision), the proposed method requires the working precision of $n = 56$; for single precision result $(23 + 1)$, $n = 28$. Table 3 gives the table sizes and number of various operations required by our method, depending on the function being computed, and Table 4 give the required table sizes assuming either $n = 56$ and $k = 14$ (double-precision), or $n = 28$ and $k = 7$ (single precision).

function	$n = 56$ and $k = 14$ (double-precision)	$n = 28$ and $k = 7$ (single-precision)
reciprocal	30 Kbytes	128 bytes
square-root	145 Kbytes	576 bytes
inv. sqrt	145 Kbytes	576 bytes
logarithm	145 Kbytes	576 bytes
exponential	114 Kbytes	448 bytes
sin/cos	229 Kbytes	896 bytes

Table 4: Required table sizes assuming either $n = 56$ and $k = 14$ (double-precision), or $n = 28$ and $k = 7$ (single precision).

7 Appendix: proof of Theorem 1

Let us start from the series (6):

$$\begin{array}{c} f(A) = C_0 + C_1 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right) + C_2 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^2 \\ + C_3 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^3 + C_4 \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^4 + \ldots \end{array} \tag{18}$$

Let us keep in mind that $A = A_2 z^2 + A_3 z^3 + A_4 z^4$ is obviously less than 2^{-k}. If we drop out from the previous series the terms with coefficients C_4, C_5, C_6, C_7, ..., the error will be:

$$\left| \sum_{i=4}^{\infty} C_i \left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^i \right|,$$

which is bounded by

$$\epsilon_1 = C_{max} \sum_{i=4}^{\infty} \left(2^{-k}\right)^i = C_{max} \frac{2^{-4k}}{1 - 2^{-k}} \tag{19}$$

where $C_{max} = \max_{i \geq 4} |C_i|$.

Now, let us expand the expression obtained from (6) after having dis-

carded the terms of rank ≥ 4. We get:

$$\begin{aligned} f(A) \approx\ & C_0 + C_1 A + C_2 A_2^2 z^4 + 2C_2 A_2 A_3 z^5 \\ & + \left(2C_2 A_2 A_4 + C_2 A_3^2 + C_3 A_2^3\right) z^6 + \left(2C_2 A_3 A_4 + 3C_3 A_2^2 A_3\right) z^7 \\ & + \left(C_2 A_4^2 + 3C_3 A_2^2 A_4 + 3C_3 A_2 A_3^2\right) z^8 + \left(6C_3 A_2 A_3 A_4 + C_3 A_3^3\right) z^9 \\ & + \left(3C_3 A_2 A_4^2 + 3C_3 A_3^2\right) z^{10} + 3C_3 A_3 A_4^2 z^{11} + C_3 A_4^3 z^{12}. \end{aligned} \tag{20}$$

In this rather complicated expression, let us discard all the terms of the form $W \times z^j$ such that the maximum possible value of W multiplied by $z^j = 2^{-kj}$ is less than or equal to z^4. We then get (7), that is:

$$f(A) \approx C_0 + C_1 A + C_2 A_2^2 z^4 + 2C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6 .$$

To get a bound on the error ϵ obtained when approximating (20) by (7), we replace the A_i's by their maximum value 2^k, and we replace the C_i's by their absolute value. This gives:

$$\begin{aligned} \epsilon_2 \quad \leq \quad & \left(4|C_2| + 3|C_3|\right) 2^{-4k} + \left(2|C_2| + 6|C_3|\right) 2^{-5k} + 7|C_3| 2^{-6k} \\ & + 6|C_3| 2^{-8k} + |C_3| 2^{-9k} \\ & \leq \left(4|C_2| + 3|C_3| + 9 \max\{C_2, C_3\} \times 2^{-k}\right) 2^{-4k}. \end{aligned}$$

If we assume that $7 \times 2^{-k} + 6 \times 2^{-2k} + 2^{-3k} < 1$, which is true as soon as $k \geq 4$. Therefore, ϵ_2 is bounded by a value that is very close to $\left(4|C_2| + 3|C_3|\right) 2^{-4k}$.

As explained in section 2, when computing (7), we will make another approximation: after having computed A_2^2, the computation of A_2^3 would require a $2k \times k$ multiplication. Instead of this, we will take the most k significant bits of A_2^2 only, and multiply them by A_2. If we write:

$$A_2^2 = \left(A_2^2\right)_{low} + 2^k \left(A_2^2\right)_{high} ,$$

where $\left(A_2^2\right)_{low}$ and $\left(A_2^2\right)_{high}$ are k-bit numbers, the error committed is

$$C_3 \left(A_2^2\right)_{low} A_2 z^6 ,$$

whose absolute value is bounded by $\epsilon_3 = |C_3| 2^{-4k}$.

By adding the three errors due to our having discarded terms, we get the bound given in Theorem 1.

References

[1] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM 360/370 model 91: floating-point execution unit. *IBM Journal of Research and Development*, January 1967. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[2] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

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

[4] P. M. Farmwald. High Bandwidth Evaluation of Elementary Functions. In K. S. Trivedi and D. E. Atkins, editors, *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*, pages 139–142, Ann Arbor, Michigan, May 1981. IEEE Computer Society Press, Los Alamitos, CA.

[5] 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*. Wiley, New York, 1968.

[6] H. Hassler and N. Takagi. Function evaluation by Table Look-up and Addition. In S. Knowles and W. H. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, July 1995. IEEE Computer Society Press, Los Alamitos, CA.

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

[8] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8), Aug. 1997.

[9] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[10] D. Das Sarma and D.W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. H. McAllister, editors, *Proceedings of the*

12th IEEE Symposium on Computer Arithmetic, pages 10–16, Bath, UK, July 1995. IEEE Computer Society Press, Los Alamitos, CA.

[11] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[12] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, December 1990.

[13] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[14] J. Volder. The CORDIC computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[15] J. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA,1990.

[16] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[17] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3):453–457, March 1995.