Improving Goldschmidt Division, Square Root and Square Root Reciprocal

Milos Ercegovac, Laurent Imbert, David Matula, Jean-Michel Muller, Guoheng Wei

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Improving Goldschmidt Division, Square Root and Square Root Reciprocal

Milos Ercegovac , Laurent Imbert , David Matula , Jean-Michel Muller , Guoheng Wei

No 3753

Septembre 1999

THÈME 2

ISSN 0249-6399

Improving Goldschmidt Division, Square Root and Square Root Reciprocal

Milos Ercegovac* , Laurent Imbert† , David Matula‡ , Jean-Michel Muller§ , Guoheng Wei¶

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arenaire

Rapport de recherche n°3753 — Septembre 1999 — 19 pages

Abstract: The aim of this paper is to accelerate division, square root and square root reciprocal computations, when Goldschmidt method is used on a pipelined multiplier. This is done by replacing the last iteration by the addition of a correcting term that can be looked up during the early iterations. We describe several variants of the Goldschmidt algorithm assuming 4-cycle pipelined multiplier and discuss obtained number of cycles and error achieved. Extensions to other than 4-cycle multipliers are given.

Key-words: Division, Square root, Square root reciprocal, Convergence division, Computer Arithmetic, Goldschmidt iteration.

(Résumé : tsvp)

This work has been partially supported by a French CNRS and *Ministère des Affaires étrangères* grant PICS-479, *Vers des arithmétiques plus souples et plus précises.*

* University of California at Los Angeles, Los Angeles, USA
† Laboratoire d'Informatique de Marseille, Marseille, France
‡ Southern Methodist University, Dallas, USA
§ CNRS, projet CNRS-ENSL-INRIA Arenaire, ENS Lyon, France
¶ Southern Methodist University, Dallas, USA

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Une amélioration de l'algorithme de Goldschmidt pour la division, la racine carrée et l'inverse de la racine carrée

Résumé : Le but de cet article est l'accélération de la division, et du calcul de racines carrées et d'inverses de racines carrées lorsque la méthode de Goldschmidt est utilisée sur un multiplieur pipe-line. Nous faisons ceci en remplaçant la dernière itération par l'addition d'un terme de correction qui peut être déduit d'une lecture de table effectuée lors des premières itérations. Nous décrivons plusieurs variantes de l'algorithme obtenu en supposant un multiplieur à 4 étages de pipe-line, et donnons pour chaque variante l'erreur obtenue et le nombre de cycles de calcul. Des extensions de ce travail à des multiplieurs dont le nombre d'étages est différent sont présentées.

Mots-clé : Division, Racine carrée, Inverse de la racine carrée, Arithmétique des ordinateurs, Algorithme de Goldschmidt.

1 Introduction

Although division is less frequent among the four basic arithmetic operations, a recent study by Oberman and Flynn [7] shows that in a typical numerical program, the time spent performing divisions is approximately the same as the time spent performing additions or multiplications.

This is due to the fact that in most current processors, division is significantly slower than the other operations. Hence, faster implementations of division are desirable.

There are two principal classes of division algorithms. The *digit-recurrence methods* [4] produce one quotient digit per cycle using residual recurrence which involves (i) redundant additions, (ii) multiplications with a single digit, and (iii) a quotient-digit selection function. The latency and complexity of implementation depends on the radix. The method produces both the quotient which can be easily rounded and the remainder. The *iterative*, quadratically convergent, methods, such as the Newton-Raphson, the Goldschmidt and series expansion methods (see for instance [5, 6, 11]) use multiplications and take advantage of fast multipliers implemented in modern processors. These methods, however, do not produce directly the remainder and rounding requires extra quotient digits. According to [7], roughly twice as many digits of intermediate result are needed as in the final result, unless the iterations are performed using a fused multiply-accumulate operator, that performs computations of the form $ax + b$ with one final rounding only [1].

In this paper, we focus on the latter class of methods. Such methods have been implemented in various microprocessors such as the IBM RS/6000 [12] or the more recent AMD K7 processor [14]. Our goal is to find a way of accelerating the Goldschmidt iteration (G-iteration in the sequel) when implementing it on a pipelined computer. We then extend our work to square root and square root reciprocal calculations.

2 Division

2.1 Background and G-iteration

Assume two n-bit inputs N and D, that satisfy $1 \leq N, D < 2$ (i.e., normalized significands of floating-point numbers). We aim at computing $Q = N/D$. The Goldschmidt algorithm consists in finding a sequence $K_1, K_2, K_3, \ldots$ such that $r_i =$

$DK_1K_2 \dots K_i$ approaches 1 as i goes to infinity. Hence,

$$q_i = NK_1K_2 \dots K_i \rightarrow Q.$$

This is done as follows. The first factor K_1 may be obtained by table lookup. After that, if $r_i = 1 - \alpha$, we choose $K_i = 1 + \alpha$, which gives $r_{i+1} = 1 - \alpha^2$. To be able to discuss possible alternatives to the basic algorithm, we give in detail the steps used in computing q_4.

1. **Step 1.** Let $D = 1.d_1d_2 \dots d_{n-1}$, and define $\hat{D} = 1.d_1d_2 \dots d_p$, where $p << n$. Typical values are $n = 53$ and $p \approx 10$. Obtain $K_1 = 1/\hat{D}$ from a $2^p \times p$ table such that
$$1 - 2^{-p} < K_1D < 1 + 2^{-p} \tag{1}$$
Define $\epsilon = 1 - K_1D$. From (1), $|\epsilon| < 2^{-p}$. Another solution is to add enough guard bits in the table [2] to get
$$1 - 2^{-p-0.5} < K_1D < 1 + 2^{-p-0.5} \tag{2}$$
In such a case, $|\epsilon| < 2^{-p-0.5}$. We successively compute
   - $r_1 = DK_1 = 1 - \epsilon$ (this multiplication will be called **mult. 1**);
   - $q_1 = NK_1$ (**mult. 2**).
2. **Step 2.** By 2's complementing r_1, we get $K_2 = 1 + \epsilon$. We then compute
   - $r_2 = r_1K_2 = 1 - \epsilon^2$ (**mult. 3**);
   - $q_2 = q_1K_2$ (**mult. 4**).

   Note that
$$\frac{N}{D} = \frac{q_2}{1 - \epsilon^2}.$$
3. **Step 3.** By 2's complementing r_2, we get $K_3 = 1 + \epsilon^2$. We then compute
   - $r_3 = r_2K_3 = 1 - \epsilon^4$ (**mult. 5**);
   - $q_3 = q_2K_3$ (**mult. 6**)

   such that
$$\frac{N}{D} = \frac{q_3}{1 - \epsilon^4}.$$

4. **Step 4.** By 2's complementing r_3, we get $K_4 = 1 + \epsilon^4$. We then compute $q_4 = q_3 K_4$ (**mult. 7**) such that

$$\frac{N}{D} = \frac{q_4}{1 - \epsilon^8}. \tag{3}$$

For instance, if $p = 12$, and if we do not have guard bits in the table that gives K_1, the previous process gives an approximation q_4 to N/D that satisfies

$$q_4 < \frac{N}{D} \leq q_4 \left(1 + 2^{-96}\right).$$

If we have guard bits in the table that gives K_1, so that (2) is satisfied, we have

$$q_4 < \frac{N}{D} \leq q_4(1 + 2^{-8p-4}).$$

For instance, if $p = 9$, the error is bounded by 2^{-76}. This method has a quadratic convergence: at each step, the number of significant bits of the approximation to the quotient roughly doubles.

Getting correctly rounded results, as required by the IEEE-754 standard [8], may seem less straightforward than with the digit-recurrence methods. And yet, many studies performed during the past recent years [1, 9, 15] show that with some care this can be done easily, for division as well as for square root. See [1, 12] for more details.

2.2 Basic implementation on a pipelined multiplier

In this section, we assume that we use a 4-cycle $n \times n$ pipelined multiplier. We start counting the cycles when K_1 becomes available.

The error committed using this first method is easily obtained from (3): it is around 2^{-8p} (e.g., for $p = 12$, it produces around 96 bits of accuracy). This implementation requires 17 cycles. The scheduling of the multiplications in the pipelined multiplier is shown Figure 1. It is worth noticing that we can use the "holes" in the pipeline to interlace independent divisions. By doing that, performing two interlaced divisions requires only 19 cycles (see Figure 1).We can use this method with bipartite tables (see [3]). In such a case, $K_1 \approx 1/D$ is obtained by looking up two tables with p address bits. One can show

$$1 - 2^{-\frac{3p}{2}+1} < K_1 D < 1 + 2^{-\frac{3p}{2}+1} \tag{4}$$

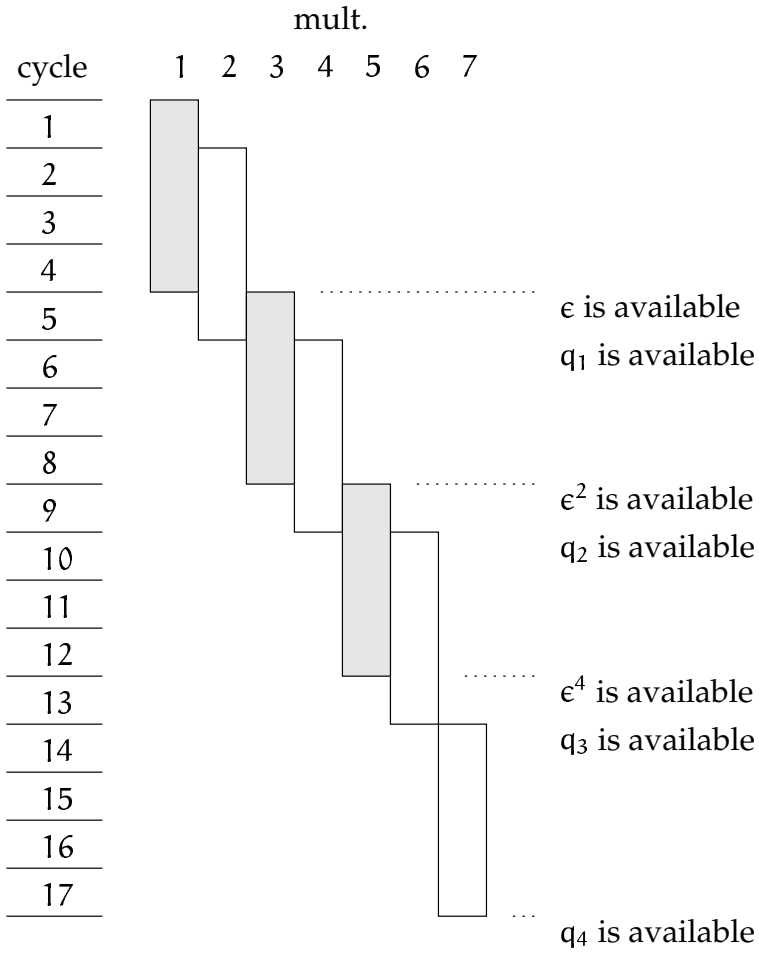

Figure 1: *Schedule of the original G-iteration. It requires* 17 *cycles to get the final result. It allows interlacing of two independent divisions: it suffices to start multiplication* **mult. 1** *of the second division at cycle 3,* **mult. 2** *at cycle 4,* **mult. 3** *at cycle 7,* **mult. 4** *at cycle 8,* **mult. 5** *at cycle 11,* **mult. 6** *at cycle 12, and* **mult. 7** *at cycle 16. Two interlaced divisions require 19 cycles.*

After four conventional iterations, one can get

$$q_4 < \frac{N}{D} \leq q_4(1 + 2^{-12p+8}).$$

For instance, if $p = 9$, this process gives an approximation with error less than 2^{-100}.

2.3 Variant A

As soon as ϵ becomes available (i.e., in cycle 5), we look-up $\hat{\epsilon}^4$ in a table with p address bits, where $\hat{\epsilon}$ is a p-bit number, constituted by the bits of $|\epsilon|$ of weight 2^{-p-1}, 2^{-p-2}, ..., 2^{-2p}. That is, if $\epsilon = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\epsilon_{p+3}\epsilon_{p+4}\ldots$, then $\hat{\epsilon} = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\ldots\epsilon_{2p}$. Then, instead of successively computing $q_3 = q_2(1+\epsilon^2)$ and $q_4 = q_3(1 + \epsilon^4) = q_2(1 + \epsilon^2 + \epsilon^4 + \epsilon^6)$, we compute directly from q_2 an approximation q_4' to q_4:

$$q_4' = q_2(1 + \epsilon^2 + \hat{\epsilon}^4).$$

We now discuss the error in the result produced by this variant. First, neglecting the term in ϵ^6 leads to an error around 2^{-6p}. Moreover from the expansion

$$\epsilon^4 = (\hat{\epsilon} + \epsilon_r)^4 = \hat{\epsilon}^4 + 4\epsilon_r\hat{\epsilon}^3 + 6\epsilon_r^2\hat{\epsilon}^2 + 4\epsilon_r^3\hat{\epsilon} + \epsilon_r^4 \tag{5}$$

where $\epsilon_r = \epsilon - \hat{\epsilon}$ (which gives $|\epsilon_r| < 2^{-2p}$), we find that the error committed when replacing ϵ^4 by $\hat{\epsilon}^4$ is around 2^{-5p+2}. For instance, if $p = 12$ this variant allows to perform the division in 13 cycles (see Figure 2), with an error around 2^{-58}. Hence, we save 4 cycles compared to the direct implementation, but at the cost of a poorer accuracy. If we use a bipartite table lookup, the same error 2^{-58} is achieved, with p equal to 9 instead of 12 (i.e., with much smaller tables).

2.4 Variant B.

To get a better accuracy than with variant A, compute the first error term in (5), that is, $c = 4\epsilon_r\hat{\epsilon}^3$. This is done by tabulating $\hat{\epsilon}^3$ and performing the multiplication $\epsilon_r \times \hat{\epsilon}^3$ in the pipelined multiplier. Hence, in this variant we compute a better approximation to q_4, that is,

$$q_4'' = q_2(1 + \epsilon^2 + \hat{\epsilon}^4 + 4\epsilon_r\hat{\epsilon}^3).$$

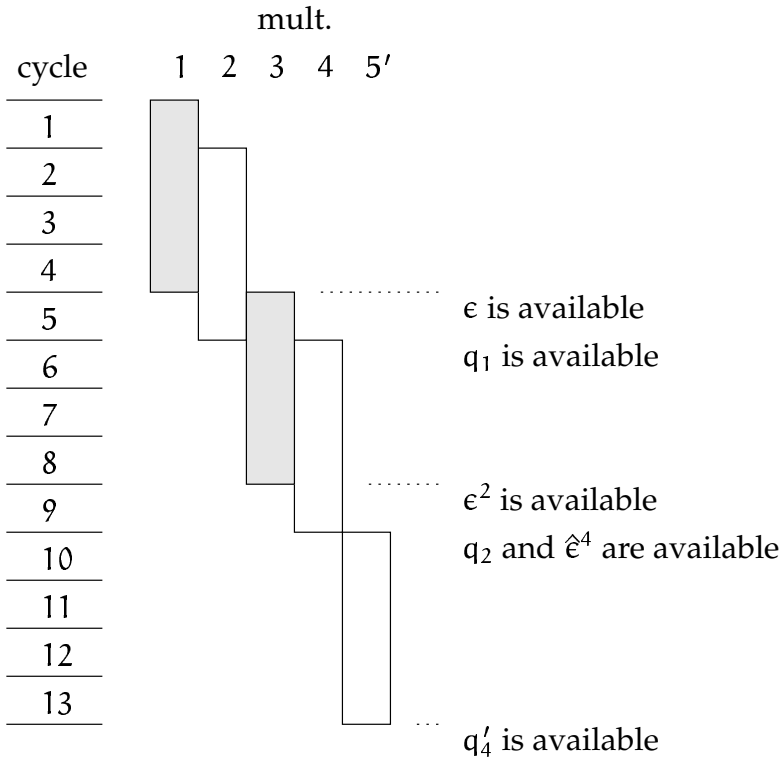

Figure 2: *Schedule of variant A. Requires 13 cycles, with an accuracy lower than that of the direct implementation. Two interlaced divisions are performed in 15 cycles.*

We need one more cycle (for the computation of c) and one more table (for $\hat{\epsilon}^3$) than in variant A. However, it is possible to improve this using the following expression for q_4'':

$$q_4'' = q_2(1 + \epsilon^2 + \hat{\epsilon}^3(4\epsilon_r + \hat{\epsilon})).$$

We now only need one table for $\hat{\epsilon}^3$, and one cycle for the computation of $c' = \hat{\epsilon}^3(4\epsilon_r + \hat{\epsilon})$. The error is about 6×2^{-6p}: for $p = 12$, this is less than 2^{-69}. If we use a bipartite lookup, we get an error 2^{-63} with $p = 9$. The corresponding schedule is shown Figure 3. On a 4-cycle multiplier, it requires 13 cycles. If we interlace two divisions, the whole calculation requires 17 cycles only. A better performance can be obtained when performing two or three consecutive divisions by the same denominator. This happens, for example, in normalizing 2-D (3-D) vectors. The improvement comes from the fact that the r_i's are the same. Computing a_1/d and a_2/d (a_1/d, a_2/d and a_3/d for 3-D) requires 15 cycles (resp. 16 cycles), whereas first computing $1/d$ and then multiplying this intermediate result by a_1 and a_2 (a_1, a_2 and a_3) would take 20 cycles (resp. 21 cycles).

2.5 Variant C

In Variant B, the values of $\hat{\epsilon}^3$ are precomputed and stored in a table with p address bits. If we consider the following formula:$(1 + \epsilon^2 + \epsilon^4) = (1 + \epsilon^2 + \epsilon)(1 + \epsilon^2 - \epsilon)$, it is possible to compute $(1 + \epsilon^2 + \epsilon^4)$ as soon as ϵ^2 is known. This technique requires 16 cycles but no table (except the one for K_1) and the error is around 2^{-6p}. This variant is probably less interesting than the direct implementation or Variant B. We mention it since it reduces the table requirements.

We present a summary of the properties of these different variants in Table 1.

Method	number of cycles	bits of accuracy	table size
Direct	17	96	$n \times 2^p$ bits
Variant A	13	58	$2n \times 2^p$ bits
Variant B	14	69	$2n \times 2^p$ bits
Variant C	16	71	$n \times 2^p$ bits

Table 1: *Main properties of the proposed variants. The third column gives the amount of memory required including the table used for* K_1.

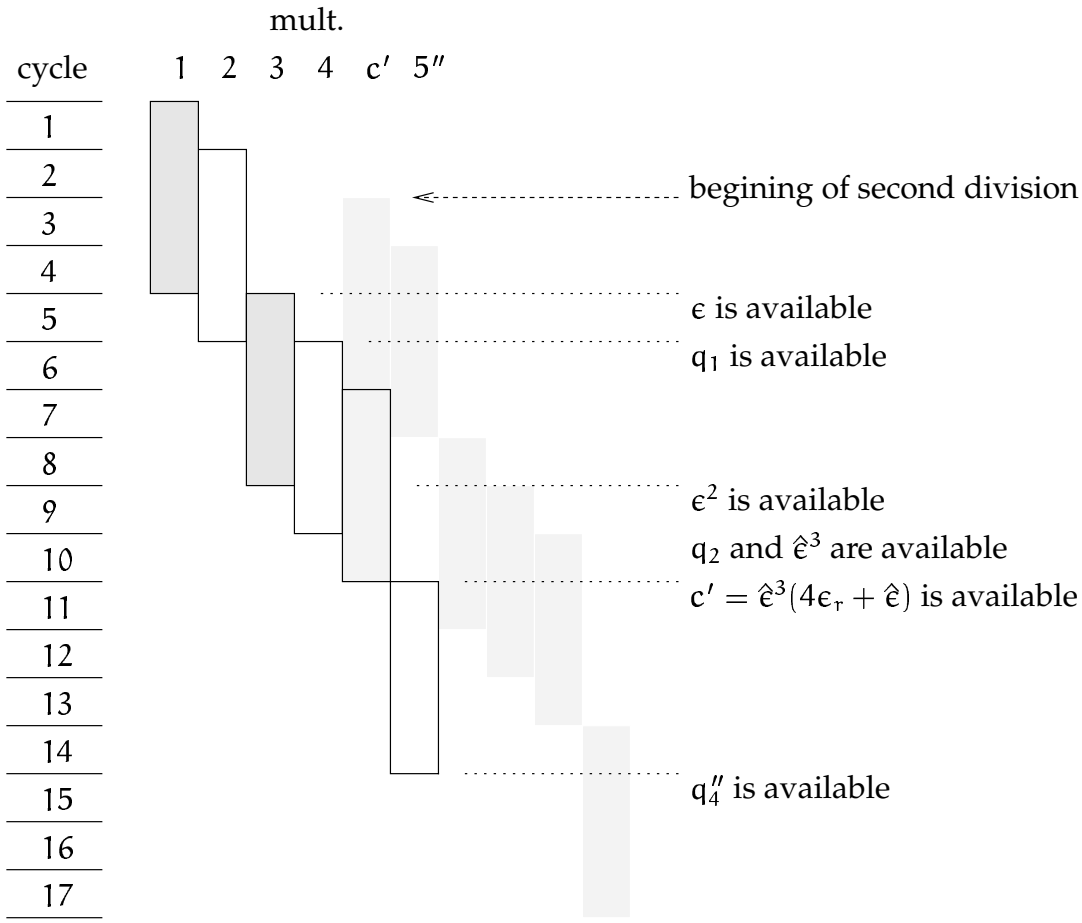

Figure 3: *Variant B. Mult. c′ is the computation of* $\hat{\epsilon}^3(4\epsilon_r + \hat{\epsilon})$. *Mult. 5″ is the final multiplication. It has one more cycle than Variant A, but the accuracy is much better. Two interlaced divisions need 17 cycles.*

2.6 Implementations on multipliers with a different number of cycles

The variants presented so far were illustrated assuming a 4-cycle pipelined multiplier. They can be used on multipliers with less or more than 4 cycles. For instance, let us assume a 2-cycle multiplier. With such a multiplier, the direct iteration (assuming we still wish to compute q_4) is implemented in 9 cycles. Variant A is implemented in 7 cycles, and variant B is implemented in 8 cycles.

On a 3-cycle multiplier, the direct iteration is implemented in 13 cycles, whereas variant A is implemented in 10 cycles, and variant B in 11 cycles.

Nevertheless, performing two consecutive divisions with multipliers with less than 4 cycles seems less interesting than previously. On a 2-cycle (3-cycle) multiplier, the direct iteration is implemented in 16 (18) cycles, variant A in 12 (14) cycles, and variant B in 14 (16) cycles.

2.7 Implementations with more than four iterations

The same approach is applicable if we want to perform one more iteration of the Goldschmidt algorithm. For example, assume that we add one more step to the algorithm presented in section 2.1. The final result q_5 is obtained as

$$q_5 = q_4 \times K_5 = q_4 \left(1 + \epsilon^8\right).$$

A direct implementation on a 4-cycle pipelined multiplier requires 21 cycles. However, once ϵ is known, we can look-up in a table the value $\hat{\epsilon}^8$, where $\hat{\epsilon}$ is the same p-bit number as in the previous sections. That value will be used to directly estimate an approximation to q_5 from q_3. Hence, we can build several variants of the algorithm:

- **First variant** we compute

  $$q'_5 = q_3 \left(1 + \epsilon^4 + \hat{\epsilon}^8\right)$$

  on a 4-cycle multiplier, this requires 17 cycles. The error is less than 2^{-9p+4}.

- **Second variant** we compute

  $$q''_5 = q_3 \left(1 + \epsilon^4 + \hat{\epsilon}^8 + 8\hat{\epsilon}^7\epsilon_r\right)$$

  on a 4-cycle multiplier, this also requires 17 cycles, and the error is less than 2^{-10p+6}. Therefore this variant is more interesting than the previous one.

- **Third variant** we compute

$$q_5''' = q_3\left(1 + \epsilon^4 + \hat{\epsilon}^8 + 8\hat{\epsilon}^7\epsilon_r + 28\hat{\epsilon}^6\epsilon_r^2\right)$$

on a 4-cycle multiplier, this requires 18 cycles, and the error is less than 2^{-11p+7}.

3 Square root and square root reciprocal

3.1 Conventional iteration

In this section we will focus on the computation of $1/\sqrt{x}$ and $\sqrt{x}$ for some real variable x. We will start from the generalization of the Goldschmidt method for square-root and square-root reciprocal that was introduced in [11]. An alternative would have been to use Newton-Raphson iteration for $\sqrt{x}$:

$$r_{i+1} = \frac{1}{2}r_i\left(3 - xr_i^2\right),$$

that can be conveniently implemented (as suggested by Schulte and Wires [16]) as:

$$\begin{array}{lcl} w_i & = & r_i^2 \\ d_i & = & 1 - w_i x \\ r_{i+1} & = & r_i + r_i d_i/2. \end{array}$$

This approach requires three dependent multiplies per iteration, similar to the Goldschmidt method introduced in [11] and cited here.

An interesting discussion on the computation of square roots and square root reciprocals using Newton's method can be found in [10].

Assume we wish to compute $\sqrt{x}$ or $1/\sqrt{x}$ for $1 \leq x < 2$. We shall consider the extension to the binade $2 \leq x < 4$ in section 3.3. The first step is similar to what we have done for the division method. Starting from

$$x = 1.d_1 d_2 \ldots d_n$$

we define

$$\hat{x} = 1.d_1 d_2 \ldots d_{p-1} + 2^{-p}$$

so then

$$|x - \hat{x}| \leq 2^{-p}$$

where $p << n$. From $\hat{x}$ we look-up $n+1$ bit round-to-nearest values of $K_1 = RN(1/\hat{x}) = .1d'_2d'_3\dots d'_{n+1}$ and $\sqrt{K_1} = RN(1/\sqrt{\hat{x}}) = .1d''_2d''_3\dots d''_{n+1}$ in a table with $p-1$ address bits. The trade off of using table lookup of K_1 rather than computing the square of the table lookup of $\sqrt{K_1}$, saves the latency cost of a dependent multiply while increasing the total table size. Importantly, the tables must be designed so that each table lookup value of K_1 corresponds at full target accuracy to the square of the table lookup value of $\sqrt{K_1}$. This requirement precludes the use of bipartite or linear interpolation for constructing both tables to greater initial accuracy as considered for division. Then, the conventional method (assuming we perform 4 iterations) consists in performing the following calculations itemized by groups of independent multiplications depending on results of the previous groups.

1. **First group** We define the variable x_1 and a variable r_1 by the independent multiplications

   - $x_1 = x \times K_1$ (called **mult. 1** in figure 4)
   - $r_1 = \sqrt{K_1}$ if we aim at computing $1/\sqrt{x}$;
   - $r_1 = x \times \sqrt{K_1}$ if we aim at computing $\sqrt{x}$. (**mult. 1'**)

   These choices are due to the fact that the next iterations compute $r_1/\sqrt{xK_1}$.

2. **Second group** We define $\epsilon_1 = 1 - x_1$ and compute the independent multiplications:

   - $(1+\frac{\epsilon_1}{2})^2 = (1+\frac{\epsilon_1}{2}) \times (1+\frac{\epsilon_1}{2})$
   - $r_2 = (1+\frac{\epsilon_1}{2}) \times r_1$

3 **Third group** We compute

   - $x_2 = (1+\frac{\epsilon_1}{2})^2 \times x_1$

   and we define ϵ_2 by $\epsilon_2 = 1 - x_2$.

4. **Fourth group** We compute the independent multiplications:

   - $(1+\frac{\epsilon_2}{2})^2 = (1+\frac{\epsilon_2}{2}) \times (1+\frac{\epsilon_2}{2})$
   - $r_3 = (1+\frac{\epsilon_2}{2}) \times r_2$

5. **Fifth group** We compute

   - $x_3 = (1+\frac{\epsilon_2}{2})^2 \times x_2$

and we define ϵ_3 by $\epsilon_3 = 1 - x_3$.

6. **Sixth group** We compute
   - $r_4 = (1 + \frac{\epsilon_2}{2}) \times r_3$

Note that these computations would require a number of cycles equal to at least 6 times the multiplier latency in a pipelined multiplier implementation.

The error committed by approximating $\sqrt{x}$ (or $1/\sqrt{x}$, depending on the initial value we have chosen for r_1) is easily found. Let us define $\epsilon = \epsilon_1$. Then recalling K_1 is rounded to nearest to $n + 1$ places,

$$|\epsilon| \leq |1 - xK_1| \leq \frac{|\hat{x} - x|}{\hat{x}} + x2^{-n-2} < 2^{-p}$$

for $n \geq 2p$. From $x_{i+1} = (1 + \frac{\epsilon_i}{2})^2 x_i = (1 + \epsilon_i + \frac{1}{4}\epsilon_i^2)x_i$ and $\epsilon_{i+1} = 1 - x_{i+1}$ we easily find

$$\epsilon_{i+1} = \frac{3}{4}\epsilon_i^2 + \frac{1}{4}\epsilon_i^3 \tag{6}$$

hence

$$\epsilon_4 \approx \left(\frac{3}{4}\right)^7 \epsilon^8 < 2^{-8p-2} \tag{7}$$

Now, since each time we multiply x_i by some factor to get x_{i+1} we multiply r_i by the square root of the same factor, we can easily deduce

$$r_4 = \sqrt{\frac{x_4}{x_1}} \times r_1$$

Hence,

$$r_4 = \frac{1 + \alpha}{\sqrt{x_1}} r_1$$

where $|\alpha| < 2^{-8p-3}$. This gives the final result:

- if we compute $1/\sqrt{x}$ (that is, we have chosen $r_1 = \sqrt{K_1}$) then

$$r_4 = \frac{1}{\sqrt{x}}(1 + \alpha), \text{ with } |\alpha| < 2^{-8p-3}$$

- if we compute $\sqrt{x}$ (that is, we have chosen $r_1 = x\sqrt{K_1}$) then

$$r_4 = \sqrt{x}(1 + \alpha), \text{ with } |\alpha| < 2^{-8p-3}.$$

3.2 Accelerating square root (inverse square root) method

Now, let us try to accelerate the computation by directly deducing an approximation to r_4 from r_2 . To do that, we first deduce the values of the ϵ_i's as polynomial identities in ϵ using (6). We obtain

$$\begin{aligned}\epsilon_2 &= \frac{3}{4}\epsilon^2 + \frac{1}{4}\epsilon^3\\ \epsilon_3 &= \frac{27}{64}\epsilon^4 + \frac{9}{32}\epsilon^5 + \frac{39}{256}\epsilon^6 + \frac{27}{256}\epsilon^7\\ &\quad + \frac{9}{256}\epsilon^8 + \frac{1}{256}\epsilon^9.\end{aligned}$$

Using this result, since $r_4 = (1 + \frac{\epsilon_2}{2})\,(1 + \frac{\epsilon_3}{2})\,r_2$, we can deduce

$$r_4 = \left(1 + \frac{\epsilon_2}{2} + \phi(\epsilon)\right) r_2 \tag{8}$$

where

$$\begin{aligned}\phi(y) &= \frac{27}{128}y^4 + \frac{9}{64}y^5 + \frac{159}{1024}y^6 + \frac{135}{1024}y^7\\ &\quad + \frac{261}{4096}y^8 + \frac{1}{32}y^9 + \frac{27}{2048}y^{10} + \frac{3}{1024}y^{11}\\ &\quad + \frac{1}{4096}y^{12}.\end{aligned}$$

This leads to the following solution: once ϵ is known, we can look-up in a table with $p-1$ address bits the value $\phi(\hat{\epsilon})$, where $\hat{\epsilon}$ (as for the division algorithm) is a p-bit number, constituted by the bits of $|\epsilon|$ of weight $2^{-p-1}, 2^{-p-2}, \ldots, 2^{-2p+1}$ and a terminal unit. That is, if

$$|\epsilon| = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\epsilon_{p+3}\epsilon_{p+4}\ldots, |\epsilon| < 2^{-p},$$

then truncating to a midpoint in the $2p$'th place,

$$|\hat{\epsilon}| = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\ldots\epsilon_{2p-1} + 2^{-2p}, |\hat{\epsilon}| < 2^{-p},$$

where with $\hat{\epsilon}$ defined to have the same sign as ϵ,

$$|\epsilon - \hat{\epsilon}| \leq 2^{-2p}.$$

Then we get the **First scheme**: We compute

$$r'_4 = \left(1 + \frac{\epsilon_2}{2} + \phi(\hat{\epsilon})\right).$$

The error of this first scheme is around $\epsilon_r\phi'(\epsilon)$ (where $\epsilon_r = \epsilon - \hat{\epsilon}$, so $|\epsilon_r| \leq 2^{-2p}$), which is less than $\frac{27}{32}2^{-5p}$. This scheme avoids several of the previous dependent multiplies. With a 4-cycle pipelined multiplier the procedure can be implemented in 16 cycles. We do not discuss this implementation in detail, since the following 2nd scheme is more accurate and implementable in the same number of cycles.

Second scheme: We now assume that $\phi'(\hat{\epsilon})$ is tabulated, and we use the following approximation to r_4:

$$r''_4 = \left(1 + \frac{\epsilon_2}{2} + \phi(\hat{\epsilon}) + \epsilon_r\phi'(\hat{\epsilon})\right) r_2.$$

The error of the second scheme is around $\frac{\epsilon_r^2}{2}\phi''(\epsilon)$, which is less than $\frac{81}{64}2^{-6p}$. For instance, if $p = 12$, this gives an error less than 2^{-71}.

Figure 4 depicts the implementation of the computation of either $1/\sqrt{x}$ or $\sqrt{x}$ using a 4-cycle multiplier. Operations 1, 2, 3 and 4 correspond to the computations of x_1, r_2, x_2 and r''_4. Mult. 1′ is performed only for the computation of $\sqrt{x}$. One can notice that the number of cycles required for both computations is the same since the initialization multiplication $r_1 = x\sqrt{K_1}$ is inserted in the pipeline when computing the square root function.

This method requires 4 tables with $p-1$ address bits each for K_1, $\sqrt{K_1}$, $\phi(\hat{\epsilon})$ and $\phi'(\hat{\epsilon})$ with total size of $(2n + 2(n-4p)) \times 2^{p-1}$ bits.

3.3 Expanding domain and contracting tables

Depending on whether the exponent of the input value is even or odd, we need to compute the square root and/or square root reciprocal of x or $2 \times x$ to span the domain $[1, 4)$. This can be implemented by optionally inserting a multiplication by $\sqrt{2}$ somewhere in the pipeline. This can be done without increasing the latency in view of the one cycle allocated to the addition needed to form $(1 + \frac{\epsilon}{2} + \phi(\hat{\epsilon}) + \epsilon_r\phi'(\hat{\epsilon}))$. In Figure 4, r_2 is available at cycle 10. Thus we can perform an optional multiplication $r_2 \times \sqrt{2}$ from cycle 10 to cycle 13. Another solution is to store tables for both $\sqrt{K_1}$ and $\sqrt{2K_1}$. But these tables can be large and avoiding duplication of storage is desirable.

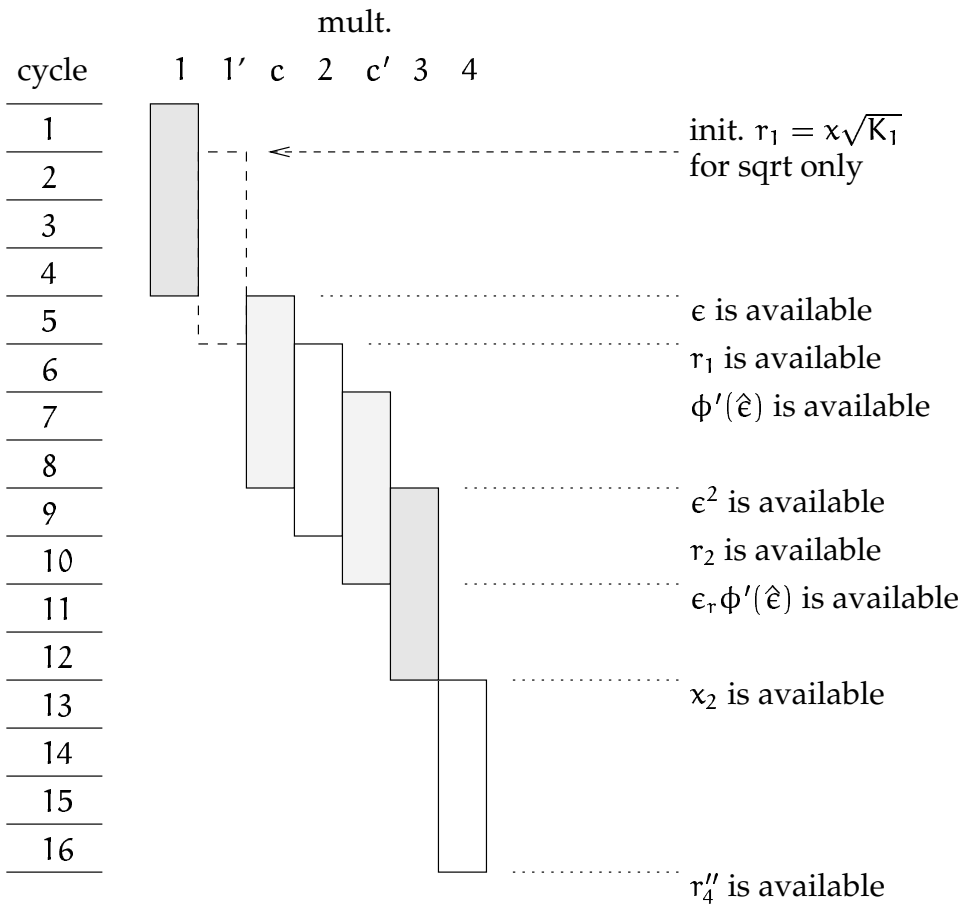

Figure 4: Implementation of $\sqrt{x}$ and $1/\sqrt{x}$ on a 4-cycle pipelined multiplier. Mult. $1'$ is performed only for $\sqrt{x}$; c and c' correspond to the computations of ϵ^2 and $\epsilon_r\phi'(\hat{e})$.

For the first lookup for K_1 and $\sqrt{K_1}$, we can relax the bound for $|\epsilon_1|$ by storing only a rounding to $p+4$ bits in the output table for $\frac{1}{\sqrt{\hat{x}}}$. We still obtain $|\epsilon_1| < 2^{-p+0.25}$. Importantly, we than should store the exact $2p+8$ bit square of the output of the first table as the result for K_1 to make sure that the table values for K_1 and $\sqrt{K_1}$ are correctly related.

The **Second scheme** requires the computation of $\phi(\hat{e}) + \epsilon_r\phi'(\hat{e})$. If we expand $\phi(\hat{e}) + \epsilon_r\phi'(\hat{e})$, we may replace the product-sum employing two tables by a single

product employing only one table similar to the procedure in [13].

$$\phi(\hat{\epsilon}) + \epsilon_r\phi'(\hat{\epsilon}) = (\hat{\epsilon} + 4\epsilon_r)(\frac{27}{128}\hat{\epsilon}^3 + \frac{9}{64}\hat{\epsilon}^4 + \frac{159}{1024}\hat{\epsilon}^5) + \delta\hat{\epsilon}^4\epsilon_r \tag{9}$$

where $|\delta| < 1$.

$\hat{\epsilon} + 4\epsilon_r$ can be obtained as follows: we have a 2-bit overlap between $\hat{\epsilon}$ and $4\epsilon_r$ at the $p-1$th and pth positions. These two bits can be added into $\hat{\epsilon}$ while we do the table lookup for $(\frac{27}{128}\hat{\epsilon}^3 + \frac{9}{64}\hat{\epsilon}^4 + \frac{159}{1024}\hat{\epsilon}^5)$ using $\hat{\epsilon}$. $\delta\hat{\epsilon}^4\epsilon_r$ provides the error term. The error is roughly of the same order of magnitude, i.e., $2^{-6(p-0.25)}$. This avoids one lookup table with no extra latency. Note that reducing the precision of the table output of K_1 and $\sqrt{K_1}$ may increase $\hat{\epsilon}$ to slightly more than 2^{-p}. This could require a p bit index lookup for at least a part of the range of x, a small increase compared to the major reduction in table size for K_1 and $\sqrt{K_1}$.

Conclusion

We have presented several variants for implementing division, square roots and square root reciprocals on a 4-cycle pipelined multiplier. The proposed schemes are based on the Goldschmidt iteration, and require fewer cycles than the original scheme. They also exhibit various trade-offs between computational delay, accuracy, and table size.

References

[1] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 96–105, Adelaide, Australia, April 1999. IEEE Computer Society Press, Los Alamitos, CA.

[2] D. Das Sarma and D. W. Matula. Measuring the Accuracy of ROM Reciprocal Tables. *IEEE Transactions on Computers*,43(8), August 1994.

[3] D. Das Sarma and D. W. Matula. Faithful Bipartite ROM Reciprocal Tables. *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, July 1995. IEEE Computer Society Press, Los Alamitos, CA.

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

[5] M. J. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8):702–706, August 1970. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[6] R. E. Goldschmidt *Applications of division by convergence*. M.Sc. Dissertation Massachusetts Institute of Technology, June 1964.

[7] S. Oberman and M. J. Flynn. Implementing division and other floating-point operations: A system perspective. In Alefeld, Fromer, and Lang, editors, *Scientific Computing and Validated Numerics (Proceedings of SCAN'95)*, pages 18–24. Akademie Verlag, 1996.

[8] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[9] C. Iordache and D. W. Matula. On Infinitely Precise Rounding for Division, Square Root, Reciprocal and Square Root Reciprocal. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240, Adelaide, Australia, April 1999. IEEE Computer Society Press, Los Alamitos, CA.

[10] W. Kahan. Square root without division. PDF file accessible electronically through the Internet at the address http://www.cs.berkeley.edu/~wkahan/ieee754status/reciprt.pdf, 1999.

[11] C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim. Some properties of iterative square-rooting methods using high-speed multiplication. *IEEE Transactions on Computers*, C-21:837–847, 1972. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[12] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[13] N. Takagi. Generating a Power of an Operand by a Table Look-up and a Multiplication. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 126–131, Asilomar, California, July 1997. IEEE Computer Society Press, Los Alamitos, CA.

[14] S. Oberman. Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Arith-14, Adelaide, Australia, April 1999)*, pages 106-115. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[15] M. Parks. Number-theoretic Test Generation for Directed Roundings. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 241–248, Adelaide, Australia, April 1999. IEEE Computer Society Press, Los Alamitos, CA.

[16] M. J. Schulte and K. E. Wires. High-Speed Inverse Square Roots. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 124–131, Adelaide, Australia, April 1999. IEEE Computer Society Press, Los Alamitos, CA.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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