Improving Goldschmidt Division, Square Root, and Square Root Reciprocal

Milos D. Ercegovac, *Member*, *IEEE*, Laurent Imbert, David W. Matula, *Member*, *IEEE Computer Society*, Jean-Michel Muller, *Member*, *IEEE Computer Society*, and Guoheng Wei

Abstract—The aim of this paper is to accelerate division, square root, and square root reciprocal computations when the 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. If we call G_m the Goldschmidt algorithm with m iterations, our variants allow us to reach an accuracy that is between that of G_3 and that of G_4, with a number of cycle equal to that of G_3.

Index Terms—Division, square root, square root reciprocal, convergence division, computer arithmetic, Goldschmidt iteration.

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 1) redundant additions, 2) multiplications with a single digit, and 3) a quotient-digit selection function. The method produces both the quotient, which can be easily rounded, and the remainder. The *iterative*, quadratically convergent, methods, such as the Newton-Raphson and Goldschmidt methods [5], [6], [11] use multiplications and take advantage of fast multipliers implemented in modern processors. These methods do not directly produce the remainder and correct rounding (as required by the IEEE-754 standard [8]) 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 [1]. For more details on correct rounding of division and square-root, see [1], [9], [12], [14].

- *M.D. Ercegovac is with the Computer Science Department, 3732 Boelter Hall, University of California at Los Angeles, Los Angeles, CA 90024. E-mail: milos@cs.ucla.edu.*
- *L. Imbert is with L.I.M., CMI, Université de Provence, 39 rue Joliot Curie, 13453 Marseille cedex 13, France. E-mail: Laurent.Imbert@cmi.univ-mrs.fr.*
- *D.W. Matual and G. Wei are with the Department of Computer Science, Southern Methodist University, Dallas, TX 75275. E-mail: {matula, gwei}@seas.smu.edu.*
- *J.-M. Muller is with CNRS-LIP, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. E-mail: jmmuller@ens-lyon.fr.*

Manuscript received 1 Sept. 1999; revised 1 Feb. 2000; accepted 10 Mar. 2000. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111796.

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 [13]. 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.

Our methods require an initial reciprocal table lookup. We will use the following result:

Theorem 1 (DasSarma and Matula [2]). *The maximum relative error of an optimal reciprocal table with k-bits-in and $k+g$-bits-out is bounded above by*

$$2^{-(k+1)}\left(1+\frac{1}{2^{g+1}}\right).$$

In the following, we assume that we use an optimal reciprocal table, with p-bits-in and $p+2$-bits-out. Let x be a real number. Using such a table, addressed by the first p bits of x, one can get an approximation K to $1/x$ that satisfies

$$1-2^{-p-0.83} < Kx < 1+2^{-p-0.83}. \quad (1)$$

Typical currently feasible values for p are around 10.

An alternative is to use bipartite tables (see [3]). In such a case, $K \approx 1/x$ is obtained by looking up two tables with p address bits. One can show

$$1-2^{-\frac{3p}{2}+1} < Kx < 1+2^{-\frac{3p}{2}+1}. \quad (2)$$

2 Division

2.1 Background and G-Iteration

Assume two n-bit inputs N and D, that satisfy $1 \le N, D < 2$ (i.e., normalized significands of floating-point numbers). We aim at computing $Q = N/D$. The Goldschmidt algorithm consists of finding a sequence $K_1, K_2, K_3, \ldots$ such that the product $r_i = DK_1K_2 \ldots K_i$ approaches 1 as i goes to infinity. Hence,

$$q_i = NK_1K_2 \ldots K_i \to Q.$$

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

1. **Step 1.** Let $D = 1.d_1d_2 \ldots d_{n-1}$ and define $\hat{D} = 1.d_1d_2 \ldots d_p$, where $p << n$. Typical values are $n = 53$ and $p \approx 10$. From an optimal reciprocal table with p-bits-in and $p+2$-bits-out that uses $\hat{D}$ as entry, obtain a $p+2$-bit approximation K_1 to $1/D$. Define $\epsilon = 1 - K_1D$. From (1), $|\epsilon| < 2^{-p-0.83}$. 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**).
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**).
4. **Step 4.** By 2's complementing r_3, we get $K_4 = 1+\epsilon^4$. We then compute $q_4 = q_3K_4$ (**mult. 7**). This gives

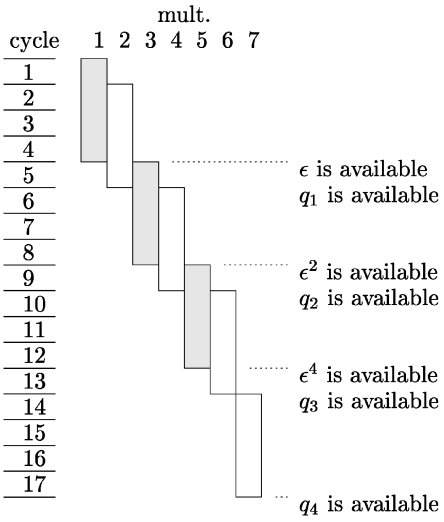

Fig. 1. Schedule of the original G-iteration. It requires 17 cycles to get the final result. It allows interlacing of two independent divisions: It suffices to start multiplication **mult. 1** of the second division at cycle 3, **mult. 2** at cycle 4, **mult. 3** at cycle 7, **mult. 4** at cycle 8, **mult.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

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

This process gives an approximation q_4 to N/D that satisfies

$$q_4 < \frac{N}{D} \le q_4\left(1 + 2^{-8p-6.64}\right).$$

This method has a quadratic convergence: At each step, the number of significant bits of the approximation to the quotient roughly doubles.

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.* This implementation requires 17 cycles. The scheduling of the multiplications in the multiplier is shown in Fig. 1. We can use the "holes" in the pipeline to interlace independent divisions.

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 $p-1$ address bits, where $\hat{\epsilon}$, with the same sign as ϵ, is 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 $|\epsilon| = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\epsilon_{p+3}\epsilon_{p+4}\ldots$, then $|\hat{\epsilon}| = 0.000\ldots0\epsilon_{p+1}\epsilon_{p+2}\ldots\epsilon_{2p}1$. This gives $|\epsilon - \hat{\epsilon}| < 2^{-2p}$. Then,

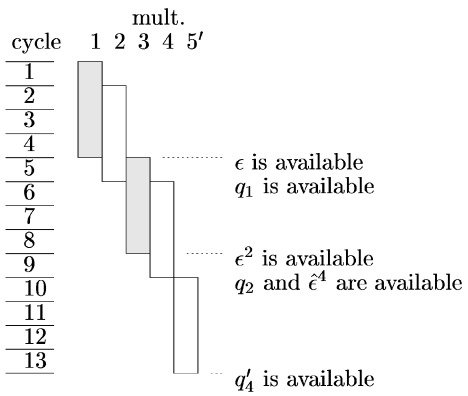

Fig. 2. Schedule of variant A. Requires 13 cycles, with an accuracy lower than that of the direct implementation. Two interlaced divisions are performed in 15 cycles.

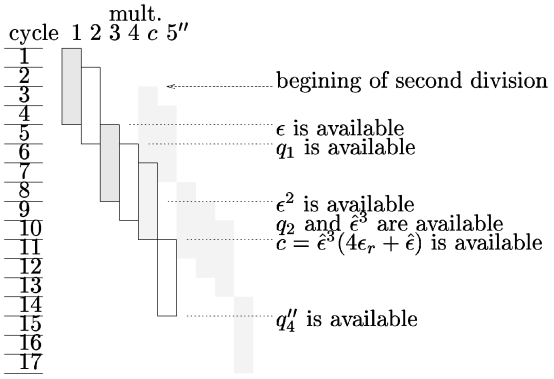

Fig. 3. Variant B. Mult. c is the computation of $\hat{\epsilon}^3(4\epsilon_r + \hat{\epsilon})$. Mult. 5" is the final multiplication. It has one more cycle than Variant A, but the accuracy is much better. Two interlaced divisions need 17 cycles.

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. First, neglecting the term in ϵ^6 leads to an error less than $2^{-6p-4.98}$. This shows how many bits of the values $\hat{\epsilon}^4$ we need to store. These values are less than $2^{-4p-3.32}$, hence it suffices to store $2p+1$ bits of each of these values to get an error similar to the error due to our having neglected ϵ^6. Moreover, from the expansion

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

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 $4\epsilon_r\hat{\epsilon}^3 \le 2^{-5p-0.49}$. We save four cycles compared to the direct implementation, but at the cost of poorer accuracy. This variant is shown in Fig. 2.

2.4 Variant B

To get better accuracy than with variant A, we compute the first error term in (4), that is, $c = 4\epsilon_r\hat{\epsilon}^3$. This is done by tabulating $\hat{\epsilon}^3$ and computing a sharper approximation to q_4:

$$\begin{aligned} q''_4 &= q_2(1+\epsilon^2+\hat{\epsilon}^4+4\epsilon_r\hat{\epsilon}^3) \\ &= q_2(1+\epsilon^2+\hat{\epsilon}^3(4\epsilon_r+\hat{\epsilon})). \end{aligned}$$

We need one table for $\hat{\epsilon}^3$ and one more cycle for the computation of $c = \hat{\epsilon}^3(4\epsilon_r + \hat{\epsilon})$. The error is about $2^{-6p+0.95}$. The corresponding schedule is shown Fig. 3. On a 4-cycle multiplier, it requires 13 cycles. A better performance can be obtained when performing two or three consecutive divisions by the same denominator. This happens, for example, in normalizing 2D (3D) vectors. The improvement comes from the fact that the r_is are the same. Computing a_1/d and a_2/d requires 15 cycles, whereas first computing $1/d$ and then multiplying this intermediate result by a_1 and a_2 would take 20 cycles.

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

2.5 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 four cycles. With a 2-cycle multiplier, the direct iteration (assuming we still wish to compute q_4) is implemented in nine cycles. Variant A is implemented in seven cycles and variant B is implemented in eight cycles. On a 3-cycle multiplier, the direct

TABLE 1
Main Properties of the Proposed Variants

Method	♯ of cycles	accuracy (bits)	table size (bits)
Direct	17	8p+6	$(p+2)\times 2^p$
A	13	5p	$(2p+3/2)\times 2^p$
B	14	6p-1	$(2p+3/2)\times 2^p$

The third column gives the amount of memory required, including the table used for K_1.

iteration is implemented in 13 cycles, whereas variant A is implemented in 10 cycles, and variant B in 11 cycles.

2.6 Implementations with More than Four Iterations

The same approach is applicable if we want to perform more iterations of the Goldschmidt algorithm. Assume that we add one more step to the algorithm presented in Section 2.1. The final result q_5 is obtained as

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

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 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, for instance:

- **First variant:** We compute

  $$q_5'' = q_3\big(1+\epsilon^4+\hat{\epsilon}^8+8\hat{\epsilon}^7\epsilon_r\big)$$

  on a 4-cycle multiplier; this requires 17 cycles and the error is less than $2^{-10p-0.17}$.
- **Second variant:** We compute

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

  on a 4-cycle multiplier; this requires 18 cycles and the error is less than $2^{-11p+1.66}$.

3 Square Root and Square Root Reciprocal

3.1 Conventional Iteration

In this section, we focus on the computation of $1/\sqrt{x}$ and $\sqrt{x}$ for some real variable x. We 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 [10] for $\sqrt{x}$:

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

that can be conveniently implemented (as suggested by Schulte and Wires [15]) as:

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

This approach requires three dependent multiplies per iteration, similar to the Goldschmidt method used here. Assume we wish to compute $\sqrt{x}$ or $1/\sqrt{x}$ for $1\le x<2$. We shall consider the extension to the binade $2\le x<4$ in Section 3.3. Starting from $x = 1.d_1d_2\ldots d_n$, we define

$$\hat{x} = 1.d_1d_2\ldots d_p + 2^{-p-1},$$

so then $|x-\hat{x}|\le 2^{-p-1}$, where $p << n$. From $\hat{x}$ we look up the number G equal to $1/\sqrt{\hat{x}}$ rounded to the nearest $p+2$-bit number, in a table with p-bits in and $p+1$-bits out.[1] We also look up $K_1 = G^2$ in a table with p-bits in and $2p+3$-bits out. It is important that the relation $K_1 = G^2$ be exact. One can show that, as soon as $p\ge 5$:

$$|K_1x-1| < 2^{-p-0.226}. \tag{5}$$

The conventional method (assuming we perform four iterations) consists of performing the following calculations itemized by groups of independent multiplications:

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 Fig. 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 perform the independent multiplications:
   - $\big(1+\frac{\epsilon_1}{2}\big)^2 = \big(1+\frac{\epsilon_1}{2}\big)\times\big(1+\frac{\epsilon_1}{2}\big)$,
   - $r_2 = \big(1+\frac{\epsilon_1}{2}\big)\times r_1$.
3. **Third group:** We compute
   - $x_2 = \big(1+\frac{\epsilon_1}{2}\big)^2\times x_1$

   and we define ϵ_2 by $\epsilon_2 = 1-x_2$.
4. **Fourth group:** We perform the independent multiplications:
   - $\big(1+\frac{\epsilon_2}{2}\big)^2 = \big(1+\frac{\epsilon_2}{2}\big)\times\big(1+\frac{\epsilon_2}{2}\big)$,
   - $r_3 = \big(1+\frac{\epsilon_2}{2}\big)\times r_2$.
5. **Fifth group:** We compute
   - $x_3 = \big(1+\frac{\epsilon_2}{2}\big)^2\times x_2$

   and we define ϵ_3 by $\epsilon_3 = 1-x_3$.
6. **Sixth group:** We compute
   - $r_4 = \big(1+\frac{\epsilon_3}{2}\big)\times r_3$.

The error committed is easily found. Let us define $\epsilon = \epsilon_1$. From (5), we have $|\epsilon| < 2^{-p-0.226}$. From $x_{i+1} = \big(1+\frac{\epsilon_i}{2}\big)^2x_i = \big(1+\epsilon_i+\frac{1}{4}\epsilon_i^2\big)x_i$ and $\epsilon_{i+1} = 1-x_{i+1}$, we find

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

hence,

$$\epsilon_4 \approx \left(\frac{3}{4}\right)^7\epsilon^8 < 2^{-8p-4.713}. \tag{7}$$

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 deduce $r_4 = \sqrt{x_4/x_1}\times r_1$. Hence,

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

where $|\alpha| < 2^{-8p-5.713}$. This gives the final result:

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

1. The "$p+1$" instead of "$p+2$" comes from the fact that the first fractional bit of G is always 1. Therefore, there is no need to store it.

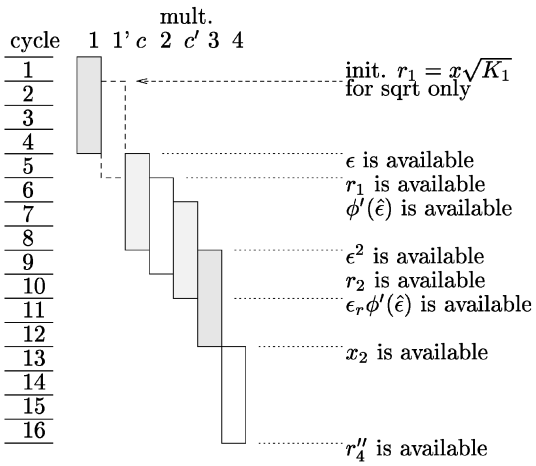

Fig. 4. Implementation of $\sqrt{x}$ and $1/\sqrt{x}$ on a 4-cycle pipelined multiplier. Mult. 1' is performed only for $\sqrt{x}$; c and c' correspond to the computations of ϵ^2 and $\epsilon_r\phi'(\hat{\epsilon})$.

$$r_4 = \frac{1}{\sqrt{x}}(1+\alpha), \text{ with } |\alpha| < 2^{-8p-5.713}.$$

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

$$r_4 = \sqrt{x}(1+\alpha), \text{ with } |\alpha| < 2^{-8p-5.713}.$$

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_is as polynomial identities in ϵ using (6). We obtain $\epsilon_2 = \frac{3}{4}\epsilon^2 + \frac{1}{4}\epsilon^3$ and

$$\epsilon_3 = \frac{27}{64}\epsilon^4 + \frac{9}{32}\epsilon^5 + \frac{39}{256}\epsilon^6 + \frac{27}{256}\epsilon^7 + \frac{9}{256}\epsilon^8 + \frac{1}{256}\epsilon^9.$$

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

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

where

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

Therefore, once ϵ is known, we can look up in a table with $p-1$-bits-in and $2p+4$-bits-out the value $\phi(\hat{\epsilon})$, where $\hat{\epsilon}$ 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,

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

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

TABLE 2
Main Properties of the Proposed Variants

Scheme	♯ of cycles	accuracy (bits)	table size (bits)
Direct	24	8p+5	$(3p+4)\times 2^p$
First	16	5p	$(4p+6)\times 2^p$
Second	16	6p	$(5p+8)\times 2^p$

The third column gives the amount of memory required, including the table used for K_1.

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

The error of this scheme is around $\epsilon_r\phi'(\epsilon)$ (where $\epsilon_r = \epsilon - \hat{\epsilon}$), which is less than $2^{-5p-0.923}$. 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 second scheme is more accurate and implementable in the same number of cycles.

Second scheme: We now assume that $\phi'(\hat{\epsilon})$ is tabulated in a table with $p-1$-bits-in and $p+4$-bits-out and we use the following approximation to r_4:

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

with $\epsilon_r = \epsilon - \hat{\epsilon}$. The error of the second scheme is around $\frac{\epsilon_r^2}{2}\phi''(\epsilon)$, which is less than $2^{-6p-0.112}$. Fig. 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 $\sqrt{x}$. 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.

We present a summary of the properties of these schemes in Table 2.

3.3 Expanding Domain

Depending on the exponent of the input value, 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 somewhere in the pipeline. If we compute $1/\sqrt{x}$, it suffices to insert an optional multiplication by $1/\sqrt{2}$ from cycle 2 to cycle 5 of Fig. 4 (instead of mult 1′). If we compute $\sqrt{x}$, since r_2 is available at cycle 10, we can perform an optional multiplication $r_2\times\sqrt{2}$ from cycle 10 to cycle 13. This delays the final multiplication and the whole calculation is performed in 17 cycles instead of 16. Another solution is to store tables for both $\sqrt{K_1}$ and $\sqrt{2K_1}$, but avoiding duplication of storage is probably desirable in most cases.

4 Conclusion

We have presented several variants for implementing division, square roots, and square root reciprocals on a 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. More details can be found on a report available through http://www.ens-lyon.fr/~jmmuller/RR-3753.pdf.

Acknowledgments

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* and the US National Science Foundation Grant *Effect of Redundancy in Arithmetic Operations on Processor Cycle Time, Architecture, and Implementation*.

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," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 86-105, Apr. 1999.

[2] D. Das Sarma and D.W. Matula, "Measuring the Accuracy of ROM Reciprocal Tables," *IEEE Trans. Computers,* vol. 43, no. 8, Aug. 1994.

[3] D. Das Sarma and D.W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 17-28, July 1995.

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

[5] M.J. Flynn, "On Division by Functional Iteration," *IEEE Trans. Computers,* vol. 19, no. 8, pp. 702-706, Aug. 1970. Reprinted in *Computer Arithmetic,* E.E. Swartzlander, vol. 1. Los Alamitos, Calif.: IEEE CS Press, 1990.

[6] R.E. Goldschmidt, "Applications of Division by Convergence," MSc dissertation, Massachusetts Inst. of Technology, June 1964.

[7] S. Oberman and M.J. Flynn, "Implementing Division and Other Floating-Point Operations: A System Perspective," *Scientific Computing and Validated Numerics,* Alefeld, Fromer, and Lang, eds., pp. 18-24, Akademie Verlag, 1996.

[8] Am. Nat'l Standards Inst. and IEEE, "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," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 233-240, Apr. 1999.

[10] W. Kahan, "Square Root without Division," pdf file accessible electronically, 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 Trans. Computers,* vol. 21, pp. 837-847, 1972.

[12] P.W. Markstein, "Computation of Elementary Functions on the IBM RISC System/6000 Processor," *IBM J. Research and Development,* vol. 34, no. 1, pp. 111-119, Jan. 1990.

[13] S. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 106-115, Apr. 1999.

[14] M. Parks, "Number-Theoretic Test Generation for Directed Roundings," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 241-248, Apr. 1999.

[15] M.J. Schulte and K.E. Wires, "High-Speed Inverse Square Roots," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 124-131, Apr. 1999.

The Montgomery Modular Inverse—Revisited

E. Savas, *Student Member*, *IEEE*, and
Ç.K. Koç, *Senior Member*, *IEEE*

Abstract—We modify an algorithm given by Kaliski to compute the Montgomery inverse of an integer modulo a prime number. We also give a new definition of the Montgomery inverse, and introduce efficient algorithms for computing the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse. The proposed algorithms are suitable for software implementations on general-purpose microprocessors.

Index Terms—Modular arithmetic, modular inverse, almost inverse, Montgomery multiplication, cryptography.

1 INTRODUCTION

THE basic arithmetic operations (i.e., addition, multiplication, and inversion) modulo a prime number p have several applications in cryptography, for example, the deciphering operation in the RSA algorithm [9], the Diffie-Hellman key exchange algorithm [1], the US Government Digital Signature Standard [8], and also, recently, elliptic curve cryptography [5], [6]. The modular inversion operation plays an important role in public-key cryptography, particularly, to accelerate the exponentiation operation using the so-called addition-subtraction chains [2], [4] and also in computing point operations on an elliptic curve defined over the finite field $GF(p)$ [5], [6].

The modular inverse of an integer $a \in [1, p-1]$ modulo the prime p is defined as the integer $x \in [1, p-1]$ such that $ax = 1 (\mathrm{mod}\ p)$. It is often written as $x = a^{-1} (\mathrm{mod}\ p)$. This is the classical definition of the modular inverse [4]. In the sequel, we will use the notation

$$x := \mathrm{ModInv}(a) = a^{-1} (\mathrm{mod}\ p) \tag{1}$$

to denote the inverse of a modulo p. The definition of the modular inverse was recently extended by Kaliski to include the so-called Montgomery inverse [3] based on the Montgomery multiplication algorithm [7]. In this paper, we introduce a new definition of the Montgomery inverse, and also give efficient algorithms to compute the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse of an integer a modulo the prime number p.

2 THE MONTGOMERY INVERSE

The Montgomery multiplication [7] of two integers $a, b \in [0, p-1]$ is defined as $c = ab2^{-n} (\mathrm{mod}\ p)$, where $n = \lceil \log_2 p \rceil$. We denote this multiplication operation using the notation

$$c := \mathrm{MonPro}(a, b) = ab2^{-n} (\mathrm{mod}\ p), \tag{2}$$

where p is the prime number and n is its bit-length. The Montgomery inverse of an integer $a \in [1, p-1]$ is defined by Kaliski [3] as the integer $x = a^{-1}2^n (\mathrm{mod}\ p)$. Similarly, we will use the notation

• *The authors are with the Electrical and Computer Engineering Department, Oregon State University, Corvallis, OR 97331.*
E-mail: {savas, koc}@ece.orst.edu.

Manuscript received 1 Sept. 1999; revised 1 Feb. 2000; accepted 10 Mar. 2000. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111797.

$$x := \text{MonInv}(a) = a^{-1}2^n (\text{mod } p) \quad (3)$$

to denote the Montgomery inversion as defined by Kaliski. The algorithm introduced in [3] computes the Montgomery inverse of a. We give this algorithm below. The output of Phase I is the integer r such that $r = a^{-1}2^k (\text{mod } p)$, where $n \leq k \leq 2n$. This result is then corrected using Phase II to obtain the Montgomery inverse $x = a^{-1}2^n (\text{mod } p)$.

Phase I

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$ and k, where $r = a^{-1}2^k (\text{mod } p)$ and $n \leq k \leq 2n$

1: $u := p$, $v := a$, $r := 0$, and $s := 1$
2: $k := 0$
3: while $(v > 0)$
4: if u is even then $u := u/2$, $s := 2s$
5: else if v is even then $v := v/2$, $r := 2r$
6: else if $u > v$ then $u := (u - v)/2$, $r := r + s$, $s := 2s$
7: else if $v \geq u$ then $v := (v - u)/2$, $s := s + r$, $r := 2r$
8: $k := k + 1$
9: if $r \geq p$ then $r := r - p$
10: return $r := p - r$ and k

Phase II

Input: $r \in [1, p-1]$, p, and k from Phase I
Output: $x \in [1, p-1]$, where $x = a^{-1}2^n (\text{mod } p)$

11: for $i = 1$ to $k - n$ do
12: if r is even then $r := r/2$
13: else then $r := (r + p)/2$
13: return $x := r$

3 The Almost Montgomery Inverse

As shown above, Phase I computes an integer $r = a^{-1}2^k (\text{mod } p)$, where $n \leq k \leq 2n$. The Montgomery inverse of a is defined as $x = a^{-1}2^n (\text{mod } p)$, where $n = \lceil \log_2 p \rceil$. We will call the output of Phase I the *almost Montgomery inverse* of a, and denote it as

$$(r, k) := \text{AlmMonInv}(a) = a^{-1}2^k (\text{mod } p), \quad (4)$$

where $n \leq k \leq 2n$, in the sequel. We note that a similar concept, the almost inverse of elements in the Galois field $GF(2^m)$, was introduced in [11] and some implementation issues were addressed in [10].

Since k is an output of Phase I, we will include it in the definition of the AlmMonInv function as an output value. We also propose to make an additional change in the way the almost Montgomery inverse algorithm is being used. Instead of selecting the Montgomery radix as $R = 2^n$, where $n = \lceil \log_2 p \rceil$, we will select it as $R = 2^m$, where m is an integer multiple of the wordsize of the computer w, i.e., $m = iw$ for some positive integer i. The Montgomery product algorithm would work with any m as long as $m \geq n$, where n is the bit-length of the prime number p. For efficiency reasons, we select the smallest i which makes m larger than n, in other words, $iw = m \geq n$, but $(i-1)w < n$. It turns out that the almost Montgomery inverse algorithm (Phase I) works for this case as well. Furthermore, it even works for an input a which may be larger than p as long as it is less than 2^m, as proven below in Theorem 1. The second issue is the value of k after the almost Montgomery inverse algorithm terminates. We show in Theorem 2 below that $n \leq k \leq m + n$.

TABLE 1
Reduction of uv and $u + v$ at Each Iteration

	uv	$u + v$
Step 4	$(uv)/2$	$(u + 2v)/2$
Step 5	$(uv)/2$	$(2u + v)/2$
Step 6	$u^2/2 - (uv)/2$	$(u + v)/2$
Step 7	$(uv)/2 - v^2/2$	$(u + v)/2$

Theorem 1. *If $p > 2$ is a prime and $a \geq 1$ (a might be larger than p), then the intermediate values r, s, and u in the almost Montgomery inverse algorithm are always in the interval $[0, 2p - 1]$.*

Proof. If $a < p$, then the proof given in [3] is applicable here. If $a > p$ and a is not an integer multiple of p, then only Step 5 and Step 7 are executed in the while loop until v becomes smaller than u. Until then, the variables u, r, and s keep their initial values. They start changing when $v < u$ and, after this point, the algorithm proceeds as in the case $a < p$. Thus, the intermediate values remain in the interval $[0, 2p - 1]$ for $a > p$ as well. □

Theorem 2. *If $p > 2$ is a prime and $a \geq 1$, then the index k produced at the end of the almost Montgomery inverse algorithm takes a value between n and $m + n$, where $n = \lceil \log_2 p \rceil$ and $m = sw$ with $sw \geq n$ with $(s-1)w < n$.*

Proof. The reduction of uv and $u + v$ at each iteration (at Steps 4-7) is illustrated in Table 1. Note that these steps are mutually exclusive, i.e., at an iteration only one of the four cases occurs. At each iteration, the value uv is at least halved while the value $u + v$ is at most halved and, furthermore, both u and v are equal to 1 before the last iteration. Since the initial values of the product uv and the sum $u + v$ are ap and $a + p$, respectively, the index value k (i.e., the number of iterations) satisfies

$$(a + p)/2 \leq 2^{k-1} \leq ap.$$

Since $2^{n-1} < p < 2^n$ and $0 < a < 2^m$, we have

$$\begin{aligned} 2^{n-2} < \quad 2^{k-1} \quad &< 2^m \cdot 2^n, \\ 2^{n-1} \leq \quad 2^{k-1} \quad &\leq 2^{m+n-1}. \end{aligned}$$

Thus, we obtain the result: $n \leq k \leq m + n$. Furthermore, we note that $m - n \leq w - 1$, where w is the word size of the machine. This implies that $m - w + 1 \leq k \leq m + n$. □

4 Using the Almost Montgomery Inverse

The Montgomery inverse algorithm computes $x = a^{-1}2^n (\text{mod } p)$. The Kaliski algorithm [3] uses the bit-level operations in Phase II in order to achieve its goal. It uses $k - n$ steps in Phase II, where, at each step, a bit-level right shift operation is performed. Additionally, if r is odd, an addition operation $r + p$ needs to be performed.

As suggested earlier, we will use the definition $x = a^{-1}2^m (\text{mod } p)$. Furthermore, it is possible to eliminate the bit-level operations completely and use the Montgomery product algorithm to obtain the same result. In our approach, we replace these bit-level operations by word-level Montgomery product operations which are intrinsically faster on microprocessors, particularly when the wordsize of the computer is large (i.e., 16, 32, or 64).

The new Phase II is based on the precomputed Montgomery radix $R = 2^m (\text{mod } p)$, however, we only need $R^2 (\text{mod } p)$. This value can be precomputed and saved and used as necessary. Another issue is the range of input variables to the AlmMonInv and MonPro functions. For both of these functions, any input cannot exceed $2^m - 1$.

4.1 The Modified Kaliski-Montgomery Inverse

This algorithm computes $x = \text{MonInv}(a) = a^{-1}2^m \pmod p$ given the integer a. Thus, it finds the inverse of the integer a modulo p and also converts it to the Montgomery domain. The modified Kaliski-Montgomery inverse algorithm is given below.

Input: a, p, n, and m, where $a \in [1, 2^m - 1]$.
Output: $x = a^{-1}2^m \pmod p$, where $x \in [1, p-1]$.
1: $(r, k) := \text{AlmMonInv}(a)$ where $r = a^{-1}2^k \pmod p$ and $n \le k \le m + n$.
2: If $n \le k \le m$ then
2.1: $r := \text{MonPro}(r, R^2) = (a^{-1}2^k)(2^{2m})(2^{-m}) = a^{-1}2^{m+k} \pmod p$
2.2: $k := k + m > m$
3: $r := \text{MonPro}(r, 2^{2m-k}) = a^{-1} \cdot 2^k \cdot 2^{2m-k} \cdot 2^{-m} = a^{-1}2^m \pmod p$
4: Return $x = r$, where $x = a^{-1}2^m \pmod p$

The inputs to the MonPro function in Step 2.1 are r and R^2, which are both in the correct range. The input 2^{2m-k} to MonPro in Step 3 is also in the correct range since k is adjusted to be larger than m in Step 2.2 when $k \le m$, thus, $0 < 2^{2m-k} < 2^m$.

4.2 The Classical Modular Inverse

In some cases, we are only interested in computing $x = \text{ModInv}(a) = a^{-1} \pmod p$ without converting to the Montgomery domain. One way to achieve this is to first compute the Kaliski-Montgomery inverse of a to obtain $b = a^{-1}2^m \pmod p$ and, then, reconvert the result back to the residue (non-Montgomery) domain using the Montgomery product as

$$b := \text{MonInv}(a) = a^{-1}2^m \pmod p,$$
$$x := \text{MonPro}(b, 1) = (a^{-1}2^m)(1)2^{-m} = a^{-1} \pmod p.$$

Another way of computing the classical inverse is by reversing the order of MonInv and MonPro operations, and using the constant $R^2 = 2^{2m} \pmod p$ as follows:

$$b := \text{MonPro}(a, R^2) = (a)(2^{2m})2^{-m} = a2^m \pmod p,$$
$$x := \text{MonInv}(b) = (a2^m)^{-1}2^m = a^{-1} \pmod p.$$

However, either one of these approaches requires two or three Montgomery product operations in addition to the AlmMonInv function. Instead, we can modify the Kaliski-Montgomery inverse algorithm so that it directly computes the classical modular inverse after the AlmMonInv function with one or two Montgomery product operations.

Input: a, p, n, and m, where $a \in [1, 2^m - 1]$
Output: $x = a^{-1} \pmod p$, where $x \in [1, p-1]$
1: $(r, k) := \text{AlmMonInv}(a)$ where $r = a^{-1}2^k \pmod p$ and $n \le k \le m + n$.
2: If $k > m$ then
2.1: $r := \text{MonPro}(r, 1) = (a^{-1}2^k)(2^{-m}) = a^{-1}2^{k-m} \pmod p$
2.2: $k := k - m < m$
3: $r := \text{MonPro}(r, 2^{m-k}) = (a^{-1})(2^k)(2^{m-k})(2^{-m}) = a^{-1} \pmod p$
4: Return $x = r$, where $x = a^{-1} \pmod p$

4.3 The New Montgomery Inverse

We propose the following new definition of the Montgomery inverse: $x = a^{-1}2^{2m} \pmod p$ given the input $a \pmod p$. According to this new definition, we compute the Montgomery inverse of an integer which is already in the Montgomery domain, producing the output x which is also in the Montgomery domain. We will denote the new Montgomery inverse computation by

$$x := \text{NewMonInv}(a2^m) = (a2^m)^{-1}2^{2m} = a^{-1}2^m \pmod p.$$

The Kaliski-Montgomery inverse of a is defined as $\text{MonInv}(a) = a^{-1}2^m \pmod p$, which has the following property

$$\text{MonPro}(a, \text{MonInv}(a)) = \text{MonPro}(a, a^{-1}2^m) = a(a^{-1}2^m)2^{-m} = 1 \pmod p.$$

In other words, according to the Kaliski-Montgomery inverse, the multiplicative identity is equal to 1, which is an incorrect assumption if we are operating in the Montgomery domain where the image of 1 is $2^m \pmod p$. On the other hand, the new Montgomery inverse has the following property:

$$\text{MonPro}(a2^m, \text{NewMonInv}(a2^m)) = a2^m(a^{-1}2^m)2^{-m} = 2^m \pmod p.$$

This new definition of the inverse is more suitable for computing expressions using the Montgomery multiplication since it computes the result in the Montgomery domain.

The new Montgomery inverse cannot be directly computed using the MonInv algorithm by giving the input as $a2^m \pmod p$ since we would obtain

$$\text{MonInv}(a2^m) = (a2^m)^{-1}2^m = a^{-1} \pmod p.$$

However, this can be converted back to the Montgomery domain using a single Montgomery product with $R^2 \pmod p$. Thus, we obtain a method of computing the new Montgomery inverse as

$$b := \text{MonInv}(a2^m) = (a2^m)^{-1}2^m = a^{-1} \pmod p,$$
$$x := \text{MonPro}(b, R^2) = a^{-1}2^{2m}2^{-m} = a^{-1}2^m \pmod p.$$

Similarly, another method to obtain the same result is by reversing the order of the operations:

$$b := \text{MonPro}(a2^m, 1) = (a2^m)(1)(2^{-m}) = a \pmod p,$$
$$x := \text{MonInv}(a) = a^{-1}2^m \pmod p.$$

The new algorithm uses the precomputed value $R^2 \pmod p$ and it is more efficient: It uses only two or three Montgomery product operations after the AlmMonInv function.

Input: $a2^m \pmod p$, p, n, and m
Output: $x = a^{-1}2^m \pmod p$, where $x \in [1, p-1]$
1: $(r, k) := \text{AlmMonInv}(a2^m)$ where $r = a^{-1}2^{-m}2^k \pmod p$ and $n \le k \le m + n$
2: If $n \le k \le m$ then
2.1: $r := \text{MonPro}(r, R^2) = (a^{-1}2^{-m}2^k)(2^{2m})(2^{-m}) = a^{-1}2^k \pmod p$
2.2: $k := k + m > m$
3: $r := \text{MonPro}(r, R^2) = (a^{-1}2^{-m}2^k)(2^{2m})(2^{-m}) = a^{-1}2^k \pmod p$
4: $r := \text{MonPro}(r, 2^{2m-k}) = (a^{-1}2^k)(2^{2m-k})(2^{-m}) = a^{-1}2^m \pmod p$
5: Return $x = r$, where $x = a^{-1}2^m \pmod p$

5 Conclusions and Applications

We have proposed a new definition of the Montgomery inverse and have given efficient algorithms to compute the classical modular inverse, the Kaliski-Montgomery inverse, and the new Montgomery inverse. The new algorithms are based on the almost

TABLE 2
Implementation Results

Bit	PhI	Algorithm	Old PhII	New PhII	PhII Spd	All Spd
160	138 μs	MonInv	30 μs	9 μs	3.34	1.14
		ModInv	85 μs	10 μs	8.50	1.51
		NewMonInv	57 μs	10 μs	5.70	1.32
192	192 μs	MonInv	39 μs	11 μs	3.54	1.14
		ModInv	128 μs	13 μs	9.85	1.56
		NewMonInv	87 μs	13 μs	6.69	1.36

Montgomery inverse function and require two or three Montgomery product operations thereafter, instead of using the bit-level operations as in [3].

We have performed some experiments by implementing all three inversion algorithms using both classical (shift and add) and newly proposed Montgomery product-based Phase II steps. These algorithms were coded using the Microsoft Visual C++ 5.0 development system. The timing results are obtained on a 450-MHz Pentium II processor running the Windows NT 4.0 operating system. In Table 2, we summarize the timing results. The table contains the old and new Phase II timings (Old PhII and New PhII) in microseconds for operands of length 160 and 192 bits. The last two columns (PhII Spd and All Spd) give the speedup in Phase II only and the overall speedup, which illustrates the efficiency of the algorithms introduced.

An application of the new Montgomery inverse is found in computing eP, where e is an integer and P is a point on an elliptic curve defined over the finite field $GF(p)$. This computation requires that we perform elliptic curve point addition $P+Q$ and doubling $P+P=2P$ operations, where each point operation requires a few modular additions and multiplications and a modular inversion. The inverse operation is used to compute the variable $\lambda := (y_2 - y_1)(x_2 - x_1)^{-1} (\mathrm{mod}\ p)$, which is required in computing elliptic curve point addition of $P=(x_1, y_1)$ and $Q=(x_2, y_2)$ in order to obtain $P+Q=(x_3, y_3)$. Assuming the input variables are given in the Montgomery domain, we would like to obtain the result in the Montgomery domain. If the Kaliski-Montgomery inverse is used, it will compute the classical inverse, which is in the residue (non-Montgomery) domain and cannot be readily used in subsequent operations. We need to perform a Montgomery product with $R^2 (\mathrm{mod}\ p)$ in order to convert back to the Montgomery domain. However, with the help of the new Montgomery inverse, we can perform the above computation in a single step. Since these operations are performed for every bit of the exponent e, the new Montgomery inverse is more efficient and highly useful in this context.

ACKNOWLEDGMENTS

This work is supported by Secured Information Technology, Inc.

REFERENCES

[1] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, no. 11, pp. 644-654, Nov. 1976.
[2] Ö. Egecioglu and Ç.K. Koç, "Exponentiation Using Canonical Recoding," *Theoretical Computer Science,* vol. 129, no. 2, pp. 407-417, 1994.
[3] B.S. Kaliski Jr., "The Montgomery Inverse and Its Applications," *IEEE Trans. Computers,* vol. 44, no. 8, pp. 1,064-1,065, Aug. 1995.
[4] D.E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms,* third ed. Reading, Mass.: Addison-Wesley, 1998.
[5] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. of Computation,* vol. 48, no. 177, pp. 203-209, Jan. 1987.
[6] A.J. Menezes, *Elliptic Curve Public Key Cryptosystems.* Boston: Kluwer Academic, 1993.
[7] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. of Computation,* vol. 44, no. 170, pp. 519-521, Apr. 1985.
[8] Nat'l Inst. for Standards and Technology, *Digital Signature Standard (DSS).* Federal Register, 56:169, Aug. 1991.
[9] J.-J. Quisquater and C. Couvreur, "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem," *Electronics Letters,* vol. 18, no. 21, pp. 905-907, Oct. 1982.
[10] M. Rosing, *Implementing Elliptic Curve Cryptography.* Greenwich, Conn.: Manning Publications, 1999.
[11] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Advances in Cryptology—CRYPTO 95,* D. Coppersmith, ed., pp. 43-56, 1995.