Software Division and Square Root Using Goldschmidt's Algorithms

Peter Markstein

Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, U.S.A.

Abstract

Goldschmidt's Algorithms for division and square root are often characterized as being useful for hardware implementation, and lacking self-correction. A reexamination of these algorithms show that there are good software counterparts that retain the speed advantage of Goldschmidt's Algorithm over the Newton-Raphson iteration. A final step is needed, however, to get the last bit rounded correctly.

Key words: division, square root, Goldschmidt, floating-point

1 Introduction

Multiplicative division and square root implementations frequently employ iterative methods such as the Newton-Raphson iterations[1] and Goldschmidt's algorithms[2]. The latter are often characterized as being non-self-correcting and suitable for hardware, while the former are self-correcting and more suitable for software implementation. The usual description of Goldschmidt's algorithms reflect a hardware orientation, but the performance advantages are difficult to realize in software. In this paper, these algorithms are reformulated to make them more "software friendly" while maintaining their latency-reducing properties. Several authors have used the "software friendly" forms without ever mentioning the relationship to Goldschmidt's algorithms.

Email address: `peter.markstein@hp.com` (Peter Markstein).

2 Division

The hardware and software versions of Goldschmidt's algorithm and the Newton-Raphson iteration for division are developed below. It will be seen that after the same number of iterations, both methods have the same relative error, barring the effects of rounding.

2.1 Goldschmidt's Algorithm

Goldschmidt's algorithm for computing a_0/b_0 can be described as follows. Let Y_0 be a suitably good estimate to $1/b_0$. A suitably good approximation is one such that $3/4 \leq Y_0 b_0 < 3/2$. For the integer n such that $3/4 \leq |2^n b_0| \leq 3/2$, setting

$$Y_0 = \mathrm{sgn}(b_0)2^n(2 - \mathrm{sgn}(b_0)2^n b_0)$$

would meet this criterion. However, the initial reciprocal guess is often determined by table lookup [3]. Then, multiplying the dividend and divisor by Y_0 transforms the problem into

$$\frac{a_0}{b_0} = \frac{a_0 Y_0}{b_0 Y_0} = \frac{a_1}{b_1} \tag{1}$$

The objective in Goldschmidt's algorithm is to find quantities Y_i by which to multiply the dividend and divisor of Equation 1 to drive the divisor towards unity and as a consequence the dividend toward the quotient. The relative error of the initial guess is given by

$$e = \frac{1/b_0 - Y_0}{1/b_0} \tag{2}$$

$$= 1 - b_0 Y_0 \tag{3}$$

Then $b_1 = b_0 Y_0 = 1 - e$. By the choice of Y_0, $|e| < 1$, and taking $Y_1 = 1 + e$ as the next multiplier of the dividend and divisor would drive the new divisor to $1 - e^2$. From Equation 2, one can also write $Y_1 = 2 - b_1$. Continuing in this manner yields

$$\frac{a_0}{b_0} = \frac{a_0 Y_0 ... Y_{i-1}}{b_0 Y_0 ... Y_{i-1}} = \frac{a_i}{b_i} = \frac{a_i Y_i}{b_i Y_i} = \frac{a_{i+1}}{b_{i+1}} \qquad i = 1, 2, ... \tag{4}$$

The attractiveness of this technique for hardware embodiment is that $Y_i = 2 - b_i$ can be derived from b_i merely by bit complementation. The quantities b_i and Y_i are essentially available concurrently. As soon as a_i and b_i become available, a_i, b_i and Y_i are recycled to the multiplier's inputs. This process continues until b_i becomes (sufficiently close to) 1, or, in practice, for some fixed number of iterations determined by the quality of Y_0. Corresponding to the final b_i, a_i is taken to be the quotient. Here, the arithmetic has been assumed to be exact; in practice each multiplication is rounded at some precision, and the subtraction $2 - b_i$ itself may also be subjected to rounding. With a pipelined multiplier, the computations of the a_i are interleaved with the b_i, and the entire iteration can take place in one multiplier cycle. If the reciprocal of b_0 is required, then it can be approximated by the product $y_i = \prod Y_i$.

In a software implementation of this description of Goldschmidt's algorithm, the subtraction $2 - b_i$ cannot be hidden, so that each iteration requires two floating point cycles. But there are simple relationships between the relative error e, b_i, and Y_i. By induction, we will show that $b_i = 1 - e^{2^{i-1}}$, and $Y_i = 1 + e^{2^{i-1}}$. Equation 3 shows that $b_1 = 1 - e$, and $Y_1 = 1 + e$ (or $2 - b_1$), satisfying the inductive hypothesis for $i = 1$. Using the inductive hypothesis, $b_{i+1} = b_i Y_i = \left(1 - e^{2^{i-1}}\right)\left(1 + e^{2^{i-1}}\right) = 1 - e^{2^i}$, and $Y_{i+1} = 2 - b_{i+1} = 1 + e^{2^i}$. Thus, Goldschmidt's algorithm can be rewritten as follows:

$$e^{2^{i+1}} = (e^{2^i}) \cdot (e^{2^i}) \tag{5}$$

$$Y_{i+1} = Y_i + Y_i \cdot e^{2^i} \tag{6}$$

where the two operations can take place concurrently; the first to prepare for the subsequent iteration and the second to complete the current iteration. If it is desired to compute a_i or b_i, those computations can be done concurrently with the operations shown in Equation 5 as

$$\begin{aligned} a_{i+1} &= a_i + a_i \cdot e^{2^i} \\ b_{i+1} &= b_i + b_i \cdot e^{2^i} \end{aligned} \tag{7}$$

For software implementation, this formulation of Goldschmidt's algorithm has the advantage that the quantity $2-b_i$ does not appear. The quantities in Equation 7 can each be computed by one fused multiply-add instruction and can execute concurrently with the multiplication needed to evaluate Equation 5 for the next iteration. The software version thus also has a latency of one floating point operation, on a computer with a fused multiply-add instruction and a pipelined floating point unit. In software it is not necessary to compute the b_i, nor the Y_i (other than Y_0) if they are not needed. Finally, $2 - b_i$ suffers from precision loss as the Goldschmidt algorithm progresses, whereas the quantities e^{2^i} are computed to almost full precision. From Equations 6 Goldschmidt's algorithm is seen to be equivalent to

$$\frac{1}{b_0} = Y_0(1 + e)(1 + e^2)(1 + e^4)... \tag{8}$$

From Equation 2 it follows that

$$\begin{aligned} b_0Y_0 &= 1 - e, \qquad\qquad |e| < 1 \\ \frac{1}{b_0} &= \frac{Y_0}{1-e} = Y_0(1 + e + e^2 + e^3 + ...) \end{aligned} \tag{9}$$

which can be seen to be equivalent to Equation 8.

From these developments, taking a$_i$ as a quotient approximation will have a relative error of $e^{2^{i-1}}$. In software, each application of Equation 7 commits an error as large as 1/2 ulp (the errors due to the rounding of the quantities e^{2^i} may contribute a small fraction of an ulp to the final result.) After i iterations, the error due to roundoff may be slightly over i/2 ulps, in addition to the relative error of $e^{2^{i-1}}$ inherent in the Goldschmidt algorithm. Using the original hardware scheme may propagate a full 1 ulp error in each iteration, since $2 - b_i$ is subject to rounding, as are all the multiplications in Equation 4

Goldschmidt[2] and other authors (e.g. [5]) also showed that the algorithm is equivalent to Equation 8 but did not use it directly, whereas [4] does use Equation 8. An unusual implementation of Goldschmidt's algorithm can be found in [6]. That implementation only uses a subset of the bits of $2 - b_i$ to compute $b_i(2 - b_i)$ in order to reduce the depth of the multiplier. Hence each iteration corrects for the errors of the previous iteration which did not use all the available bits of b_i, but it does not correct for accumulated rounding errors.

2.2 Newton-Raphson Division

The Newton-Raphson iteration differs from Goldschmidt's algorithm by referring to the initial divisor during each iteration. As before, let y_0 be an approximation to $1/b_0$, and let $e_0 = 1 - b_0y_0$. If the initial approximation satisfies the same requirements as had been required to start Goldschmidt's algorithm, then $|e_0| < 1$, and Equation 9 holds (with y_0 replacing Y_0.) Taking only the linear approximation to $1/b_0$ leads to the reciprocal improvement:

$$\begin{aligned} e_0 &= 1 - b_0y_0 \\ y_1 &= y_0 + y_0e_0 \end{aligned} \tag{10}$$

As in the case of Goldschmidt's algorithm, the iteration can also be represented by

$$\begin{aligned} p &= b_0y_0 \\ y_1 &= y_0(2 - p) \end{aligned} \tag{11}$$

which gives a hardware implementation the opportunity to compute $2-p$ by means of bit complementation. Either Equation 10 or Equation 11 is repeated either until p becomes sufficiently close to 1, or e_i becomes sufficiently small, or a fixed number of times depending on the quality of y_0. The ith iteration is

$$\left.\begin{aligned} e_i &= 1 - b_0 y_i \\ y_{i+1} &= y_i + y_i e_i \end{aligned}\right\} i \geq 0 \tag{12}$$

The quotient may be developed concurrently as shown in the Goldschmidt algorithm, or the final reciprocal y_i can be multiplied by the dividend a_0 to get an approximation to the quotient. Here again, in practice the arithmetic in Equations 10, 11 and 12 is not exact, but subject to rounding. While e_0 is the relative error of the initial reciprocal approximation y_0, the approximation y_1 has a relative error of e_0^2. It can easily be shown by induction that $y_i = \prod_{j=0}^{i} Y_j$, where the Y_i are the successive multipliers of Goldschmidt's method.

The Newton-Raphson iteration takes the most recent approximation to compute the next error e_i, whereas Goldschmidt never refers back to the original data. Both methods inherently have a relative error bounded by $e^{2^{i-1}}$ after i iterations, but each Newton-Raphson iteration is based on the previous reciprocal approximation, thereby also taking into account rounding errors accumulated during the previous iteration. After any iteration, the software Newton-Raphson method is in error by at most 1/2 ulp in addition to the inherent error of the method. After i iterations, the Newton-Raphson method may have an advantage of $(i-1)/2$ ulps accuracy over Goldschmidt's method. The methods which hardware implementations use to control errors [4,5] are outside the scope of this paper.

Implementing Equation 12 in software or Equation 11 in hardware requires two dependent operations. Thus the latency of the Newton-Raphson method is twice that of Goldschmidt's.

2.3 Division Examples

In practice, Goldschmidt's algorithm and Newton-Raphson iteration are used together in software division algorithms. The Newton-Raphson iteration is used for the final (and sometimes penultimate) iteration to remove the effects of accumulated rounding errors. However, there are several division algorithms that use only the Goldschmidt algorithm to compute correctly rounded quotients. In the following examples, $uv \oplus w$ denotes that w is added to the exact product uv before rounding takes place.

Assume that the initial reciprocal approximation is obtained by the operation frcpa(b) as described for the Itanium[3], returning an approximation to $1/b$ with relative error less than $2^{-8.886}$, and that the approximation has a length of at most 11 bits (i.e., only the eleven leading bits of the significand of frcpa(b) can be non-zero). Ignoring issues of zero divisors, overflows, underflows, infinities, or NaNs, (see [1] for how these special cases are accommodated in Itanium) the following algorithm computes the single precision quotient of the single precision quantities a and b, with all operations in double-extended precision and rounded-to-nearest, except when cast to lower precisions, or when using dynamic rounding to control the rounding of a result according to any of the standard rounding modes.

$$\begin{array}{ll}
y = \mathrm{frcpa}(b) & \\
q_0 = ay; & e = 1 \ominus by \\
q_1 = q_0 e \oplus q_0; & e_1 = e \otimes e \\
q_2 = q_1 e_1 \oplus q_1; & e_2 = e_1 \otimes e_1 \\
q_3 = \mathrm{double}(q_2 e_2 \oplus q_2) & \\
Q = \mathrm{single, dynamicrounding}(q_3) &
\end{array}$$

Fig. 1. Single Precision Division

In Figure 1, calculations appearing on the same line can be carried out concurrently [1]. The quotient approximations q_i contain at least 8.886×2^i valid bits, but q_3 may only have 62 valid bits due to accumulated rounding errors, as discussed in Section 2.1. Notice that q_3 is the rounded-to-nearest approximation of the quotient to 53 bits of precision. Therefore all quotients which can be represented exactly in single precision will be correct at this stage. It is well known that quotients good to $2n + 1$ bits will round correctly to n bits [1], guaranteeing the final result is rounded correctly to single precision (requiring 24 bits of precision). The computations of e and q_1, e_1 and q_2, and e_2 and q_3 are instances of Goldschmidt's algorithm. Cornea et. al. discuss this algorithm in [7]

Using underlying double-extended precision arithmetic, Figure 2 shows an algorithm for double precision division. Note that q_0 is an exact computation from the assumptions about the length of frcpa and that b is double precision. q_4 will be correct to over 70 bits before rounding, so that exact double precision quotients will be correct, and cases difficult to round to nearest will lie halfway between two representable double precision numbers. As in the single

[1] Operations of the form $ab \oplus c$ would be implemented by fma(a, b, c) on Itanium or PowerPC; operations of the form $c \ominus ab$ would be implemented by fnma(a, b, c) on Itanium or nfms (a, b, c) on PowerPC.

```
$y = \text{frcpa}(b)$
$q_0 = ay;$                              $e = 1 \ominus by$
$q_1 = q_0 e \oplus q_0;$                $e_1 = e \otimes e$
$q_2 = q_1 e_1 \oplus q_1;$              $e_2 = e_1 \otimes e_1$
$q_3 = q_2 e_2 \oplus q_2$
$q_4 = q_3 e \oplus q_0$
$r = a - bq_4$
$Q = \text{double, dynamicrounding}(ry \oplus q_4)$
```

Fig. 2. Double Precision Division with Goldschmidt's Algorithm

precision example, there are three instances of Goldschmidt's algorithm. The computation of q_4 is not a Goldschmidt iteration; it exploits the exactness of q_0 to employ Equation 9 to extend the 62 bit accuracy of q_3 to a result accurate to over 70 bits before rounding. The computation of the final result Q uses only the initial approximation to $1/b$, and corrects q_4 by less than 1 ulp of the 64-bit approximation to the quotient. The code uses 11 floating point operations, which complete in 8 floating point latencies.

Assume that the hardware has, in addition to the ability to round to double and double extended (64-bit significand) precision, the ability to round to k-precision, for some k satisfying $55 \leq k \leq 61$. Then the double precision algorithm could be shortened, beginning with the computation of q_3, as shown in Figure 3. By rounding q_3 to $k-$precision, exact results will be exactly represented by q_3. In round-to-nearest, difficult to round quotients are represented by a quantity midway between two double precision numbers. In difficult directed rounding divisions, the $k-$precision quantity q_3 would be representable as a double precision number. The correction will adjust the result in the proper direction to insure correct rounding, but only by a fraction of a

```
$y = \text{frcpa}(b)$
$q_0 = ay;$                              $e = 1 \ominus by$
$q_1 = q_0 e \oplus q_0;$                $e_1 = e \otimes e$
$q_2 = q_1 e_1 \oplus q_1;$              $e_2 = e_1 \otimes e_1$
$q_3 = (\text{k-precision})(q_2 e_2 \oplus q_2)$
$r = a - bq_3$
$Q = \text{double, dynamicrounding}(ry \oplus q_3)$
```

Fig. 3. The shortest double precision division using Goldschmidt's Algorithm

$k-$precision ulp. This algorithm saves one instruction and one floating point latency from the first double precision division example.

In practice, software algorithms have the option to use both Goldschmidt and Newton-Raphson iterations, and may use these algorithms to first develop a high precision reciprocal instead of working directly on the quotient. Figure 4 shows the shortest double precision algorithm in terms of number of operations, using only commonly available rounding precisions (in contrast to the special rounding mode used in Figure 3.) This algorithm can be found in [1] and [7], but no mention is made of Goldschmidt's Algorithm. In this version,

$$
\begin{array}{ll}
y = \text{frcpa}(b) & \\
e = 1 \ominus by & \\
y_1 = y_0 e \oplus y_0; & e_1 = e \otimes e \\
y_2 = y_1 e_1 \oplus y_1; & e_2 = e_1 \otimes e_1 \\
y_3 = y_2 e_2 \oplus y_2 & \\
q = \text{double}(a \otimes y_3) & \\
r = a - bq & \\
Q = \text{double, dynamicrounding}(r y_3 \oplus q) &
\end{array}
$$

Fig. 4. Standard Double Precision Division using Goldschmidt's Algorithm

Goldschmidt's algorithm is used three times to develop a reciprocal approximation y_3 to over 62 bits of precision, from which an initial double precision quotient approximation q is computed. The final computation contains over 114 valid bits before rounding, and so must round correctly to 53 bits.

3 Square Root

Goldschmidt's algorithm to compute $\sqrt{b_0}$ and $1/\sqrt{b_0}$ starts with an initial approximation Y_0 to $1/\sqrt{b_0}$. The objective is to find a series of Y_i which drive the product $b_n = b_0 Y_0^2 Y_1^2 ... Y_{n-1}^2$ to 1. Then the products $y_n = Y_0 Y_1 ... Y_n$ will approach $1/\sqrt{b_0}$ and $g_n = b_0 Y_0 Y_1 ... Y_n$ will approach $\sqrt{b_0}$.

Let Y_0 be a suitably good approximation to $1/\sqrt{b_0}$, such that $1/2 < b_0 Y_0^2 < 3/2$. Such a Y_0 is usually obtained by table lookup or a special hardware instruction [3]. Set $\text{y}_0 = Y_0$, and $g_0 = b_0 y_0$ (note that g_0 is an approximation to $\sqrt{b_0}$ with the same relative error that y_0 has to $1/\sqrt{b_0}$). For $i > 0$, compute $b_i = b_{i-1} Y_{i-1}^2$. Then if $e = 1 - b_i$, $1/b_i = 1/(1-e) = 1 + e + e^2 + ...$, and $Y_i = \sqrt{1/b_i} = 1 + e/2 + \frac{3}{8} e^2 ...$ or $Y_i \approx (3 - b_i)/2$, with quadratic convergence.

Each Goldschmidt square root iteration consists of computing

$$b_i = b_{i-i}Y_{i-1}^2 \tag{13}$$
$$Y_i = (3 - b_i)/2 \tag{14}$$

Update the square root and/or the reciprocal square root approximations by

$$g_i = g_{i-1}Y_i \tag{15}$$
$$y_i = y_{i-1}Y_i \tag{16}$$

Equation 13 involves two dependent multiplications in the computation of b_i. In a hardware implementation, Y_i can be computed from b_i by bit inversion and shifting, and is available almost at the same time as b_i. In software, Equation 14 requires a fused multiply-add instruction. The evaluations of Equations 15 and 16 can occur concurrently with the next Goldschmidt iteration. Thus, the Goldschmidt algorithm requires at least three multiplications in each iteration, two of which are dependent.

Rewriting Equations 13, 15, and 16 in terms of b_0 gives

$$\begin{aligned} b_i &= b_0Y_0^2Y_1^2...Y_{i-1}^2 \\ g_i &= b_0Y_0Y_1...Y_i \qquad (17) \\ y_i &= Y_0Y_1...Y_i \qquad (18) \end{aligned}$$

Iteration stops when $b_i = b_0y_i^2$ is sufficiently close to 1, or a fixed number of iterations depending on the quality of the initial approximation y_0. Observe from Equations 17 and 18 that $g_{i-1}y_{i-1} = b_i$. This leads to an alternative to Goldschmidt's formulation, in which the b_i's are never computed. From the discussion at the beginning of this section

$$\begin{aligned} Y_i = (3 - b_i)/2 &= 1 + (1 - b_i)/2 \\ &= 1 + (1 - g_{i-1}y_{i-1})/2 \\ &= 1 + r_{i-1} \end{aligned}$$

using our observation that $g_{i-1}y_{i-1} = b_i$, and setting $r_{i-1} = 1/2 - g_{i-1}y_{i-1}/2$. With this expression for Y_i, the computations for g_i and y_i can be rewritten as

$$\begin{aligned} g_i &= g_{i-1} + g_{i-1}r_{i-1} \\ y_i &= y_{i-1} + y_{i-1}r_{i-1} \end{aligned}$$

The computations of g_i and y_i can take place concurrently, and both take the form of fused multiply-add operations. Only the computation of r_{i-1} is not quite in proper form for a fused multiply-add. To remove the division by two, let us track $h_i = y_i/2$ instead of y_i. We state the full software alternative to Goldschmidt's algorithm:

Let y_0 be a suitably good approximation to $1/\sqrt{b_0}$, such that $1/2 \leq b_0 y_0^2 \leq 3/2$. Set $g_0 = b_0 y_0$, and $h_0 = y_0/2$. Each iteration then computes

$$\left.\begin{aligned} r_{i-1} &= 0.5 - g_{i-1}h_{i-1} \\ g_i &= g_{i-1} + g_{i-1}r_{i-1} \\ h_i &= h_{i-1} + h_{i-1}r_{i-1} \end{aligned}\right\} \text{ for } i > 0 \tag{19}$$

In the modified iteration, there are three fused multiply-adds, of which the second two are independent. They can be computed concurrently on machines with two fused multiply-add units, or can be started one cycle apart on a pipelined floating point unit. If the reciprocal square root is required, the final h_i can be doubled. As with the division algorithm, when using hardware floating point arithmetic, there are accumulated rounding errors which prevent the final few bits of g_i and h_i from being trustworthy. A correction to the final iteration of Equation 19 is required.

Goldschmidt's square root algorithm can only be used to come within 3 or 4 ulps of the machine precision. Ideally, if there were no rounding errors, the ratio $g_i/h_i = 2b_0$ would be maintained, as can be seen from Equations 17 and 18. As round errors creep into the g_i and h_i, the algorithm drifts towards producing $g_i \approx \sqrt{g_{i-1}/(2h_{i-1})}$. The g's and h's may independently drift by as much as an ulp in each iteration, and the target of convergence by as much as an ulp. In a double-extended precision implementation, it is not reasonable to expect more than 62 correct bits using only Goldschmidt's algorithm.

3.1 Square Root Example

Figure 5 demonstrates a double precision square root program. Assume that the operation `frsqrta(b)` returns an approximation to $\sqrt{1/b}$ with a relative error less than $2^{-8.831}$, as in [3]. We again ignore treatment of arguments which are negative, zeros, infinities, or NaNs. All operations are assumed to be in double-extended precision and round-to-nearest mode, except when cast to lower precision or using dynamic rounding. A description of this algorithm appears in [1], but the author was unaware at the time that he was applying Goldschmidt's algorithm.

Figure 5 contains two instances of Equation 19, the alternative Goldschmidt algorithm, each written in two lines (computing r, g_1, h_1 and r_1, g_2, h_2) to show that the computations on the same line are independent and can be computed concurrently. The quantities g_1 and h_1 contain over 17 valid bits, and g_2 and h_2 over 33 valid bits. A third application of Goldschmidt's algorithm to reach 66 bits is not possible, since the computations of g_2 and h_2 contain

```
$y = \text{frsqrta}(b)$
$g = by;$                                    $h = 1/2 \cdot y$
$r = 1/2 \ominus hg$
$g_1 = gr \oplus g;$                         $h_1 = hr \oplus h$
$r_1 = 1/2 \ominus h_1 g_1$
$g_2 = g_1 r_1 \oplus g_1;$                  $h_2 = h_1 r_1 \oplus h_1$
$d = b \ominus g_2 g_2$
$g_3 = h_2 d \oplus g_2$
$d_1 = b \ominus g_3 g_3$
$S = \text{double, dynamicrounding}(h_2 d_1 \oplus g_3)$
```

Fig. 5. Double Precision Square Root using Goldschmidt's Algorithm

accumulated rounding errors affecting the last two bits of their 64-bit results. Therefore a traditional self-correcting Newton-Raphson iteration is used to compute g_3, exploiting the fact that h_2 contains a 33+ bit representation of $1/(2g_2)$. As in the case of the division algorithms, when g_3 is produced (and rounded to double-extended precision), exact results are already correct, and difficult to round results will be represented by values midway between representable double precision values, or by double precision values in the directed rounding modes. The final calculation computes S, the correctly rounded double precision square root of b, by modifying g_3 by less than a double-extended precision ulp. This adjustment serves to move inexact results a tiny fraction in the appropriate direction before rounding. The algorithm runs in 10 floating point latencies and requires 13 floating point instructions.

4 Conclusions

For both division and square root, Goldschmidt's algorithms have different preferred representations for hardware and software implementations. Hardware tends to exploit the ability to compute $2 - x$ or $(3 - x)/2$ by bit complementation. The algorithms have been restated in terms of fused multiply-add operations. In their restated form, the algorithms are suitable for software implementation, and the latencies of the modified algorithms are the same as that of the original hardware oriented forms of Goldschmidt's algorithms.

We have shown that, except for propagation of errors due to rounding, Goldschmidt's division algorithm and Newton-Raphson division produce identical results. Both versions of Goldschmidt's algorithms retain the property that the first operation for the subsequent iteration can overlap the last operation of the current iteration. The modified versions also involve intermediate quantities that retain almost full precision. The reformulated square root algorithm may be useful for hardware implementation. Nevertheless, neither Goldschmidt method can compensate for the buildup of rounding errors. While the algorithms shown here tend to only use the (modified) Goldschmidt algorithms, in practice the best results are usually achieved by starting with Goldschmidt's algorithm and then switching to the self-correcting Newton-Raphson iteration, as in Figures 4 and 5, when the desired precision is about to be achieved and correct rounding is required.

5 Acknowledgement

The author thanks Eric Schwarz for informative discussions about hardware use of Goldschmidt's algorithms.

References

[1] Peter Markstein, *IA-64 and Elementary Functions: Speed and Precision*, Prentice-Hall PTR, New Jersey, 2000.

[2] Robert E. Goldschmidt, *Applications of Division by Convergence*, MSc dissertation, M.I.T., 1964

[3] *Intel®IA-64 Architecture Software Developer's Manual*, Intel Document Number 245317 (2000).

[4] Milos D. Ercegovac, David W. Matula, Jean-Michel Muller, Guoheng Wei, *Improving Goldschmidt Division, Square Root, and Square Root Reciprocal*, IEEE Transactions on Computers, vol 49, pp. 759-763 (2000).

[5] Guy Even, Peter-Michael Seidel, and Warren Ferguson, *A Parametric Error Analysis of Goldschmidt's Division Algorithm*, Proceedings of the 16th IEEE Symposium on Computer Arithmetic, 2003, 165-171

[6] S.F. Anderson, J.G. Earle, R.E. Goldschmidt, D.M. Powers, *The IBM System/360 Model 91: Floating Point Execution Unit*, IBM Journal of Research and Development, vol 11, pp. 34-53, 1967.

[7] Marius Cornea, John Harrison, Ping Tak Peter Tang, *Scientific Computing on Itanium-based Systems*, Intel Press, Hillsboro, Oregon 2002