A Parametric Error Analysis of Goldschmidt's Division Algorithm

- EXTENDED ABSTRACT -

Guy Even
Dept. of Electrical Engineering Systems
Tel-Aviv University
Tel-Aviv 69978, Israel
guy@eng.tau.ac.il

Peter-Michael Seidel
Computer Science & Engineering Dept.
Southern Methodist University
Dallas, TX, 75275
seidel@engr.smu.edu

Warren E. Ferguson
warren_e_ferguson@hotmail.com

Abstract

Back in the 60's Goldschmidt presented a variation of Newton-Raphson iterations for division that is well suited for pipelining. The problem in using Goldschmidt's division algorithm is to present an error analysis that enables one to save hardware by using just the right amount of precision for intermediate calculations while still providing correct rounding. Previous implementations relied on combining formal proof methods (that span thousands of lines) with millions of test vectors. These techniques yield correct designs but the analysis is hard to follow and is not quite tight.

We present a simple parametric error analysis of Goldschmidt's division algorithm. This analysis sheds more light on the effect of the different parameters on the error. In addition, we derive closed error formulae that allow to determine optimal parameter choices in four practical settings.

We apply our analysis to show that a few bits of precision can be saved in the floating-point division (FP-DIV) micro-architecture of the AMD-K7™ microprocessor. These reductions in precision apply to the initial approximation and to the lengths of the multiplicands in the multiplier. When translated to cost, the reductions reflect a savings of 10.6% *in the overall cost of the FP-DIV micro-architecture.*

1. Introduction and Summary

Asymptotically optimal division algorithms are based on multiplicative division methods [13, 17, 21]. Current commercial processor designs employ a parallel multiplier for performing division and square-root operations for floating-point [1, 4, 12, 15]. The parallel multiplier is used for additional operations, such as: multiplication and fused multiply-add. Since meeting the precision requirements of division operations requires more precision than other operations, the dimensions of the parallel multiplier are often determined by precision requirements of division operations. It follows that tighter analysis of the required multiplier dimensions for division operations can lead to improvements in cost, delay, and even power consumption.

The main two methods used for multiplicative division are a variation of Newton's method [7, 8] and a method introduced by Goldschmidt [9] that is based on an approximation of a series expansion. Division based on Newton's method has a quadratic convergence rate (i.e., the number of accurate bits doubles in each iteration) and is self-correcting (i.e., inaccuracies of intermediate computations do not accumulate). A rigorous error analysis of Newton's method appears in [3, 10, 14] and for various exceptional cases in [4]. The analysis in [3, 10] considers the smallest precision required per iteration. Our error analysis follows this spirit by defining separate error parameters for every intermediate computation. In addition, the analysis in [3, 10] relies on directed roundings, a method that we use as well.

Each iteration of Newton's method involves two *dependent* multiplications; namely, the product of the first multiplication is one of the operands of the second multiplication. The implication of having to compute two dependent multiplications per iteration is that these multiplications cannot be parallelized or pipelined.

Goldschmidt's division algorithm also requires two multiplications per iteration and the convergence rate is the same as for Newton's method. However, the most important feature of Goldschmidt's algorithm is that the two multiplications per iteration are *independent* and can be pipelined or computed in parallel. On the other hand, Goldschmidt's algorithm is not self-correcting; namely, inaccuracies of intermediate computations accumulate and cause

the computed result to drift away from the accurate quotient. Goldschmidt's division algorithm was used in the IBM System/360 model 91 [2] and even more recently in the IBM S/390 [19] and in the AMD-K7^{TM} microprocessor [15]. However, lack of a general and simple error analysis of Goldschmidt's division algorithm has averted most designers from considering implementing Goldschmidt's algorithm. Thus most implementations of multiplicative division methods have been based on Newton's method in spite of the longer latency due to dependent multiplications in each iteration [4, 12] (see also [22] for more references).

Goldschmidt's method is not self-correcting as explained in [11] (there is a wrong comment on this in [23]). This makes it particularly important and difficult to keep track of accumulated and propagated error terms during intermediate computations. We were not able to locate a general analysis of error bounds of Goldschmidt's algorithm in the literature. Goldschmidt's error analysis in [9] is with respect to a design that uses a serial radix-4 Booth multiplier with 61-bits. Goldschmidt's design computes the quotient of two binary numbers in the range $[1/2, 1)$, and his analysis shows that the absolute error is in the range $[-2^{56}, 0]$. Krishnamurthy [11] analyzes the error only for the case that only one multiplicand is allowed to be imprecise in intermediate computations (the second multiplicand must be precise); such an analysis is only useful for determining lower bounds for delay. Recent implementations of Goldschmidt's division algorithm still rely on an error analysis that over-estimates the accumulated error [15]. Such over-estimates lead to correct designs but waste hardware and cause unnecessary delay (since the multiplier and the initial lookup table are too large). These over-estimations were based on informal arguments that were confirmed by a mechanically verified proof that spans over 250 definitions and 3000 lemmas [18].

Agarwal *et al.* [1] presented a multiplicative division algorithm that is based on an approximate series expansion. This algorithm was implemented in IBM's Power3^{TM}. Their algorithm provides no advantages over Goldschmidt's algorithm. In double precision, their algorithm requires 8 multiplications and the longest chain of dependent multiplications consists of 4 multiplications.

We present a version of Goldschmidt's division algorithm that uses directed roundings. We develop a simple general parametric analysis of tight error bounds for our version of Goldschmidt's division algorithm. Our analysis is parametric in the sense that it allows arbitrary one-sided errors in each intermediate computation and it allows an arbitrary number of iterations. In addition, we suggest four practical simplified settings in which errors in intermediate computations are not arbitrary. For each of these four settings, we present a closed error formula. The advantage of closed formulae is in simplifying the task of finding optimal parameter combinations in implementations of Goldschmidt's division method for a given target precision.

We demonstrate the advantages of our error analysis by showing how it could lead to savings in cost and delay. For this purpose we consider Oberman's [15] floating-point micro-architecture used in the AMD-K7^{TM} design. We present a micro-architecture that implements our version of Goldschmidt's algorithm and follows the micro-architecture described in [15]. The modules building our micro-architecture were made as similar as possible to the modules in [15]. This was done so that the issue of the precisions of the lookup table and multiplier could be isolated from other issues. Based on our analysis, we use a smaller multiplier (70×74 bits compared to 76×76 in [15]) and we allow a slightly larger initial error ($2^{-13.51}$ compared to $2^{-13.75}$ in [15]). Based on the cost models of Paul & Seidel [16] and Mueller & Paul [14], we estimate that our parameter choices for multiplier widths and initial approximation accuracy reduce the cost of the micro-architecture by 10.6% compared to the parameter choices in [15].

The paper is organized as follows. In Section 2 we present Newton's method for division and then proceed by presenting Goldschmidt's algorithm as a variation of Newton's method. In Section 3, a version of Goldschmidt's algorithm with imprecise intermediate computations is presented as well as an error analysis. In Section 4 we develop closed form error bounds for Goldschmidt's method with respect to specific settings. In Section 5 we present an alternative micro-architecture to [15] and compare costs. Due to space limitations, all proofs are omitted and can be found in the full version [6].

2. Goldschmidt's division algorithm

In this section we present Newton's method for computing the reciprocal of a given number. We then continue by describing a version of Goldschmidt's division algorithm [9] that uses precise intermediate computations. We show how Goldschmidt's algorithm is derived from Newton's method. The error analysis of Newton's method is used to analyze the errors in Goldschmidt's algorithm.

2.1. Newton's method.

Newton's method can be applied to compute the reciprocal of a given number. To compute the reciprocal of $B > 0$, apply Newton's method to the function $f(x) = B - 1/x$. Note that: (a) the root of $f(x)$ is $1/B$, which is the reciprocal we want to compute, and (b) the function $f(x)$ has a derivative $f'(x) = x^{-2}$ in the interval $(0, \infty)$. In particular, the derivative $f'(x)$ is positive.

Newton iterations are defined by the following recurrence: Let x_0 denote an initial estimate $x_0 \neq 0$ and define

x_{i+1} by

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{B - \frac{1}{x_i}}{x_i^{-2}} \\ &= x_i \cdot (2 - B \cdot x_i). \end{aligned} \tag{1}$$

Consider the *relative error* term e_i defined by

$$\begin{aligned} e_i &\triangleq \frac{\frac{1}{B} - x_i}{\frac{1}{B}} \\ &= 1 - B \cdot x_i. \end{aligned}$$

It follows that

$$\begin{aligned} e_{i+1} &= 1 - B \cdot x_{i+1} \\ &= 1 - B \cdot x_i \cdot (2 - B \cdot x_i) \\ &= (1 - B \cdot x_i)^2 \\ &= e_i^2. \end{aligned} \tag{2}$$

Equation 2 has three implications:

1. Convergence of x_i to $1/B$ at a quadratic rate is guaranteed provided that the initial relative error is less than 1. Equivalently, convergence holds if $x_0 \in (0, \frac{2}{B})$.

2. For $i \geq 1$, the relative error e_i is non-negative, hence, $x_i \leq 1/B$. This property is referred to as *one-sided convergence*.

3. If $B \in [1, 2)$, then also the *absolute error* decreases at a quadratic rate. Hence, the number of "accurate" bits doubles every iteration, and the number of iterations required to obtain p bits of accuracy is logarithmic in p.

The disadvantage of Newton's iterations, with respect to a pipelined multiplier, is that each iteration consists of 2 *dependent* multiplications: $\alpha_i = B \cdot x_i$ and $\beta_i = x_i \cdot (2 - \alpha_i)$. Namely, the product β_i cannot be computed before the product α_i is computed.

2.2. Goldschmidt's Algorithm

In this section we describe how Goldschmidt's algorithm can be derived from Newton's method. Here, our goal is to compute the quotient A/B. Goldschmidt's algorithm uses three values N_i, D_i and F_i defined as follows:

$$\begin{aligned} N_i &\triangleq A \cdot x_i \\ D_i &\triangleq B \cdot x_i \\ F_i &\triangleq 2 - D_i. \end{aligned}$$

Algorithm 1 Goldschmidt-Divide(A, B) - Goldschmidt's iterative algorithm for computing A/B

Require: $|e_0| < 1$.
1: Initialize: $N_{-1} \leftarrow A$, $D_{-1} \leftarrow B$, $F_{-1} \leftarrow \frac{1-e_0}{B}$.
2: **for** $i = 0$ to k **do**
3: $\quad N_i \leftarrow N_{i-1} \cdot F_{i-1}$.
4: $\quad D_i \leftarrow D_{i-1} \cdot F_{i-1}$.
5: $\quad F_i \leftarrow 2 - D_i$.
6: **end for**
7: Return(N_i)

Consider Newton's iteration: $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$. We may rewrite an iteration by

$$x_{i+1} = x_i \cdot F_i,$$

which when multiplied by A and B, respectively, becomes

$$\begin{aligned} N_{i+1} &= N_i \cdot F_i \ \stackrel{i \to \infty}{\longrightarrow} A/B \\ D_{i+1} &= D_i \cdot F_i \ \stackrel{i \to \infty}{\longrightarrow} 1. \end{aligned}$$

Since x_i converges to $1/B$, it follows that N_i converges to A/B and D_i converges to 1.

Note that: (a) $N_i/D_i = A/B$, for every i; (b) N_i converges to A/B at the same rate that x_i converges to $1/B$; and (c) Let $A, B > 0$. Since the relative error e_i in Newton's iterations is non-negative, for $i \geq 1$, it follows that $N_i \leq A/B$, $D_i \leq 1$ and $F_i \geq 1$ for $i \geq 1$.

As in Newton's iterations, the algorithm converges if $|e_0| < 1$ and the relative error decreases quadratically. One could use a fixed initial approximation of the quotient. Usually a more accurate initial approximation of $1/B$ is computed by a lookup table or even a more elaborate functional unit (c.f. [5, 20]).

Algorithm 1 lists Goldschmidt's division algorithm. Given A and B the algorithm computes the quotient A/B. The listing uses the same notation used above, and iterates k times.

Observe that the two multiplications that take place in every iteration (in Lines 3-4) are independent, and therefore, Goldschmidt's division algorithm is more amenable to pipelined implementations. The initial approximation is assumed to depend on the value of B. Note that k iterations of either Newton's method or Goldschmidt's algorithm require $2k + 1$ multiplications. These $2k + 1$ multiplication must be linearly ordered in Newton's method implying a critical path of $2k + 1$ dependent multiplications. In Goldschmidt's algorithm the two multiplications that take place in every iterations are independent, hence the critical path consists only of $k + 1$ multiplications.

An error analysis of Goldschmidt's algorithm with precise arithmetic is based on the following claim.

Claim 1 *The following equalities hold for $i \geq 0$:*

$$D_i = 1 - e_0^{2^i}$$
$$F_i = 1 + e_0^{2^i}$$

The key difficulty in analyzing the error in imprecise implementations of Goldschmidt's algorithm is due to the violation of the invariant $N_i/D_i = A/B$. Consider the equality

$$\begin{aligned} N_k &= A/B \cdot D_0 \cdot F_0 \cdot F_1 \cdot \ \ldots \ \cdot F_{k-1} \\ &= A/B \cdot (1 - e_0) \cdot \prod_{i=0}^{k-1} (1 + e_0^{2^i}). \end{aligned}$$

Imprecise $D_0, F_0, \ldots, F_{k-1}$ accumulate to an imprecise approximation of A/B.

3. Imprecise Intermediate Computations

This section contains the core contribution of our paper. We present a version of Goldschmidt's algorithm with imprecise intermediate computations. In this algorithm the invariant $N_i/D_i = A/B$ of Goldschmidt's algorithm with precise arithmetic does not hold anymore. We then develop a simple parametric analysis for error bounds in this algorithm. The error analysis is based on relative errors of intermediate computations. The setting is quite general and allows for different relative errors for each computation.

We define the relative error as follows.

Definition 1 *The* relative error *of x with respect to y is defined by $\frac{y-x}{y}$.*

Note that one usually uses the negative definition (i.e., $(x - y)/y$). We prefer this definition since is helps clarify the direction of the directed roundings that we use.

The analysis begins by using the exact values of all the relative errors. The values of the relative errors depend on the actual values of the inputs and on the hardware used for the intermediate computations. However, one can usually easily derive upper bounds on the absolute errors of each intermediate computation. E.g., such bounds on the absolute errors are simply derived from the precision of the multipliers. Our analysis continues by translating the upper bounds on the absolute errors to upper bounds on the relative errors. Hence we are able to analyze the accuracy of an implementation of the proposed algorithm based on upper bounds on the absolute errors.

An interesting feature of our analysis is that directed roundings are used for all intermediate calculations. Surprisingly, directed roundings play a crucial role in this analysis and enable a simpler and tighter error analysis than round-to-nearest rounding (c.f. [15]).

3.1. Goldschmidt's division algorithm using approximate arithmetic

A listing of Goldschmidt's division algorithm using approximate arithmetic appears in Algorithm 2. The values corresponding to N_i, D_i, and F_i using the imprecise computations are denoted by N_i', D_i' and F_i', respectively.

Algorithm 2 Goldschmidt-Approx-Divide(A, B) - Goldschmidt's division algorithm using approximate arithmetic

1: Initialize: $N_{-1}' \leftarrow A$, $D_{-1}' \leftarrow B$, $F_{-1}' \leftarrow \frac{1-e_0}{B}$.
2: **for** $i = 0$ to k **do**
3: $\quad N_i' \leftarrow (1 - n_i) \cdot N_{i-1}' \cdot F_{i-1}'$.
4: $\quad D_i' \leftarrow (1 + d_i) \cdot D_{i-1}' \cdot F_{i-1}'$.
5: $\quad F_i' \leftarrow (1 - f_i) \cdot (2 - D_i')$.
6: **end for**
7: Return(N_i')

Directed roundings are used for all intermediate calculations. For example, N_i' is obtained by rounding down the product $N_{i-1}' \cdot F_{i-1}'$. We denote by n_i the relative error of N_i' with respect to $N_{i-1}' \cdot F_{i-1}'$. Since $N_{i-1}' \cdot F_{i-1}'$ is rounded down, we assume that $n_i \geq 0$. Similarly, rounding down is used for computing F_i' (with the relative error f_i) and rounding up is used for computing D_i' (with the relative error d_i).

The initial approximation of the reciprocal $1/B$ is denoted by F_{-1}'. The relative error of F_{-1}' with respect to $1/B$ is denoted by e_0. We do not make any assumption about the sign of e_0.

Our error analysis is based on the following assumptions:

1. The operands are in the range $A, B \in [1, 2)$.
2. All the relative errors incurred by directed rounding are at most $1/4$. This assumption is easily met by multipliers with more than 4 bits of precision.
3. We require that $|e_0| + 3d_0/2 + f_0 < 1/2$. Again, this assumption is easily met if the multiplications and the initial reciprocal approximation are precise enough.
4. The initial approximation F_{-1}' of $1/B$ is in the range $[1/2, 1]$. This assumption is easily met if lookup tables are used.

Definition 2 *The* relative error *of N_i' with respect to A/B is*

$$\rho(N_i') \triangleq \frac{A/B - N_i'}{A/B}.$$

3.2. A simplifying assumption: strict directed roundings

The following assumption about directed rounding used in Algorithm 2 helps simplify the analysis.

Definition 3 (Strict Directed (SD) rounding) *Rounding down is* strict *if $x \geq 1$ implies that $rnd(x) \geq 1$. Similarly, rounding up is* strict *if $x \leq 1$ implies that $rnd(x) \leq 1$.*

Observe that, in general, rounding down means that $rnd(x) \leq x$, for all x. Often the absolute error introduced by rounding is bounded by $\varepsilon > 0$, namely $x - \varepsilon \leq rnd(x) \leq x$. Strict rounding down requires that if $x \geq 1$, then $rnd(x) \geq 1$ no matter how close x is to 1. In non-redundant binary representation strict rounding is easily implemented as follows. Strict rounding down can be implemented by truncating. Strict rounding up can be obtained by (i) an increment by a unit in each position below the rounding position and (ii) truncation of the bit string in positions less significant than the rounding position.

Assumption 2 (SD rounding) *All directed roundings used in Algorithm 2 are strict.*

3.3. Parametric Error Analysis

Lemma 3 bounds the ranges of N'_i and F'_i in Algorithm 2 under Assumption 2. This lemma is an extension of the properties $D_i \leq 1$ and $F_i \geq 1$ (for $i \geq 1$) of Algorithm 1.

Goldschmidt already pointed out that since F_i tends to 1 from above, one could save hardware since the binary representation of F_i begins with the string $1.000\ldots$. An analogous remark holds for D_i. However, Lemma 3 refers to the inaccurate intermediate results (i.e., D'_i and F'_i) rather than the precise intermediate results (i.e., D_i and F_i). Parts 2-3 of lemma 3 show that the same hardware reduction strategy applies to Algorithm 2, even though intermediate calculations are imprecise.

Definition 4 *Define δ_i, for $i \geq 0$, as follows:*

$$\delta_i \quad := \quad \begin{cases} |e_0| + 3d_0/2 & \text{for } i = 0 \\ \delta_{i-1}^2 + f_{i-1} & \text{otherwise.} \end{cases}$$

Lemma 3 *(ranges of D'_i and F'_i)*
The following bounds hold:

1. $D'_0 \in [1 - \delta_0, 1 + \delta_0] \subseteq (1/2, 3/2)$.
2. $D'_i \in [1 - \delta_i, 1]$, *for every* $i \geq 1$.
3. $F'_i \in [1, 1 + \delta_i]$, *for every* $i \geq 1$.
4. $D'_i \leq D'_{i+1}$, *for every* $i \geq 1$.

The following claim summarizes the relative error of Goldschmidt's division algorithm using approximate arithmetic.

Theorem 4 *For every $i > 0$, the relative error $\rho(N'_i) = \frac{A/B - N'_i}{A/B}$ satisfies $\pi_i \leq \rho(N'_i) \leq \pi_i + \delta_i$, where π_i is defined by $\pi_i \triangleq 1 - (1 - n_i) \cdot \prod_{j=0}^{i-1} \frac{1-n_j}{1+d_j} \geq 0$.*

For $i = 0$ it can be verified that $|\rho(N'_0)| \leq \pi_0 + \delta_0$.

A somewhat looser (yet easier to evaluate) bound on the relative error follows from

$$\pi_i \quad \leq \quad \sum_{j=0}^{i} n_j + \sum_{j=0}^{i-1} d_j. \tag{3}$$

3.4. Deriving bounds on relative errors from absolute errors

In this subsection we obtain bounds on the relative errors $\rho(N'_i)$ from the absolute errors of the intermediate computations. The reason for doing so is that in an implementation one is able to easily bound the absolute errors of intermediate computations; these follow directly from the precision of the operation, the rounding used (e.g., floor or ceiling), and the representation of the results (binary, carry-save, etc.).

Consider the computation of N'_i. The relative error introduced by this computation is n_i, and N'_i equals $(1 - n_i) \cdot N'_{i-1} \cdot F'_{i-1}$. An accurate computation would produce the product $N'_{i-1} \cdot F'_{i-1}$. Hence, the absolute error is $n_i \cdot N'_{i-1} \cdot F'_{i-1}$.

Definition 5 *The absolute errors of intermediate computations are defined as follows:*

$$\begin{aligned} neps_i &\triangleq n_i \cdot N'_{i-1} \cdot F'_{i-1} \\ deps_i &\triangleq d_i \cdot D'_{i-1} \cdot F'_{i-1} \\ feps_i &\triangleq f_i \cdot (2 - D'_i). \end{aligned}$$

In an implementation, the exact absolute errors are unknown. Instead, we use upper bounds on the absolute errors. We denote these upper bounds as follows: $\widehat{neps}_i \geq neps_i$, $\widehat{deps}_i \geq deps_i$ and $\widehat{feps}_i \geq feps_i$.

The following claim shows how one can derive upper bounds on the relative errors from upper bounds on the absolute errors.

Claim 5 *(from absolute to relative errors) If $A, B \in [1, 2)$, then for $i \geq 2$ the relative errors are bounded by:*

$$\begin{aligned} 0 \leq n_i &\leq 2\widehat{neps}_i/(1 - \pi_{i-1} - \delta_{i-1}) \\ 0 \leq d_i &\leq \widehat{deps}_i/(1 - \delta_{i-1}) \\ 0 \leq f_i &\leq \widehat{feps}_i \end{aligned}$$

A careful reader might be concerned by the fact that δ_{i-1} and π_{i-1} appear in the above bounds on the relative errors n_i and d_i. When analyzing the errors, one computes upper bounds for all relative errors from the first iteration to the last. These bounds are used to compute upper bounds on δ_{i-1} and π_{i-1}, which in turn are used to bound n_i and d_i. In the full version [6] bounds are given for the relative errors in the first and second iteration.

4. Closed Form Error Bounds in Specific Settings

In this section we describe specific settings of the relative errors that enable us to derive closed form error bounds. The advantage of having closed form error bounds is that such bounds simplify the task of minimizing an objective function (modeling cost or delay) subject to the required precision. Closed form error bounds also enable one to easily evaluate the effect of design choices (e.g., initial error, precision of intermediate computations, and number of iterations) on the final error. We have derived closed form error bounds in four specific settings. We are describing Setting I and Setting IV in the following. Settings II ($n_i, d_i \leq \hat{n}$ and f_i/δ_i^2 is exponential in $-k$) and Setting III ($n_i, d_i \leq \hat{n}$ and f_i/δ_i^2 is constant) are presented in the full version [6].

4.1. Setting I: $n_i, d_i \leq \hat{n}$ and $f_i = 0$.

Setting I deals with the situation that all the relative errors n_i, d_i are bounded by the same value $\hat{n}$. In addition it is assumed in this setting that $f_i = 0$, for every i. The justification for Setting I is that if all internal operands are represented by binary strings of equal length, then it is possible to bound all the relative errors n_i, d_i by the same value. The relative errors f_i can be assumed to be 0, if the computations $F_i' = (2 - D_i')$ are precise.

Using Theorem 4 and Eq. 3, the relative approximation error $\rho(N_i') = \frac{A/B - N_i'}{A/B}$ in Setting I can be bounded by:

$$0 \;\leq\; \rho(N_i') = \frac{A/B - N_i'}{A/B} \;\leq\; (2i+1)\hat{n} + (|e_0| + \frac{3}{2}\hat{n})^{2^i}.$$

4.2. Setting IV: $n_i, d_i \leq \hat{n}$ and $f_i \leq \hat{f}$ for every i.

In setting IV the assumptions are: (i) $n_i, d_i \leq \hat{n}$, for every i, and (ii) $f_i \leq \hat{f} \leq 1/8$, for every i. Hence, $\delta_i \leq \delta_{i-1}^2 + \hat{f}$ for all $i \geq 0$.

The following claim bounds the error term δ_k corresponding to the kth iteration of Algorithm 2.

Claim 6 *Let $\alpha = (1 + \sqrt{\hat{f}})$. For every $k > 0$ the following holds:*

$$\delta_k \;\leq\; \hat{f} + \max\{\alpha^{2^{k+1}-2} \cdot \delta_0^{2^k}, (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2, 9\hat{f}^2\}.$$

Note that a slightly looser bound that does not involve a max function can be written as:

$$\delta_k \;\leq\; \hat{f} \;+\; \alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} \;+\; 7\hat{f}^{3/2}.$$

Based on Theorem 4 and Equation 3 the error bound in setting IV satisfies:

$$\rho(N_k') \;<\; (2k+1) \cdot \hat{n} \;+\; \hat{f} \;+ \max\{\alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} \;,\; (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2 \;,\; 9\hat{f}^2\}.$$

One can easily see that, due to the first term, there is a threshold above which increasing the number of iterations (while maintaining all other parameters fixed) increases the bound on the relative error. Moreover, the contribution of the error term $\hat{f}$ to $\rho(N_k')$ does not increase with the number of iterations k (as opposed to $\hat{n}$). This implies that in a cost effective choice one would use $\hat{f} > \hat{n}$.

5. Application: An Alternative FP-DIV Micro-architecture for AMD-K7™

In this section we propose an alternative FP-DIV micro-architecture for the AMD-K7 microprocessor [15]. This alternative micro-architecture is a design that implements Algorithm 2. Our micro-architecture uses design choices that are similar to those of [15] to facilitate isolating the effect of precisions on cost. Our error analysis allows us to accurately determine the required multiplier precision and thus both save cost and reduce delay.

Overview micro-architecture. The FP-DIV micro-architecture of the AMD-K7 microprocessor is described in [15]. The micro-architecture is based on Goldschmidt's algorithm. We briefly outline this micro-architecture: (i) Round-to-nearest rounding is done in intermediate computations (as opposed to directed rounding suggested in Algorithm 2). (ii) The design contains a single 76×76-bits multiplier. This means that the absolute errors $\widehat{neps}_i$ and $\widehat{deps}_i$ are identical during all the iterations (i.e., since round-to-nearest is used, $\widehat{neps}_i = \widehat{deps}_i = 2^{-76}$). However, our alternative micro-architecture may use smaller multipliers (even multipliers in which the multiplicands do not have equal lengths) provided that the error analysis proves that the final result is accurate enough. (iii) Intermediate results are compressed and represented using non-redundant binary representation. This means that Assumption 2 on strict directed rounding is easy to implement in our alternative micro-architecture. (Recall that directed rounding is used in Algorithm 2.) (iv) The computation of F_i' is done using one's complement computation. This means that the absolute error $\widehat{feps}_i$ is identical during all the iterations, and that the error analysis of Setting IV is applicable for our alternative architecture. (v) Final rounding of the quotient is done by back multiplication. Our alternative micro-architecture uses the same final rounding simply by meeting the same error bounds needed in the final rounding of [15].

Required final precisions. The micro-architecture in [15] supports multiple precisions: single precision $(24, 8)$ in one iteration, double precision $(53, 11)$ in two iterations, an extended precision $(64, 15)$ and an internal extended precision $(68, 18)$ in three iterations. Final rounding is based on back-multiplication: namely, comparing

$N'_k \cdot B$ with A. In general, correct IEEE rounding based on back-multiplication requires that $\rho(N'_k) < 2^{-(p+1)}$, where p denotes the precision. (The description of the required precision for correct rounding in [15] is somewhat confusing since it is stated in terms of a two sided absolute error. For example, the absolute error in the 68-bit precision is bounded by 2^{-70}.)

To summarize, the upper bounds on the relative errors are as follows: (i) for single precision: $\rho(N'_1) < 2^{-25}$, (ii) for double precision: $\rho(N'_2) < 2^{-54}$, (iii) for extended double precision: $\rho(N'_3) < 2^{-65}$, and (iv) for the 68-bit precision: $\rho(N'_3) < 2^{-69}$.

Note that the bound for the 64-bit precision is weaker than the bound for the 68-bit precision. The bound for single precision is easily satisfied by the parameter choices needed to satisfy the 53-bit precision. Hence we focus below on two iterations for double precision and on three iterations for the 68-bit precision.

From relative errors to multiplier dimensions. In the full version [6], we analyze the lengths of the multiplicands in Algorithm 2. We obtain the following results. The length of the first multiplicand (used for N'_i and D'_i) should be slightly larger than $\log_2(1/\hat{n}) + 2$. The length of the second multiplicand should be greater than or equal to $log_2(1/\hat{f}) + 1 + \log_2(1/(1-\delta_0))$.

Optimizing the error parameters. In the full version [6], we present a search for combinations of relative errors that minimize the sizes of the multiplier and lookup table. We used a cost function that is based on the cost model of Paul & Seidel [16] for Booth Radix-8 multipliers and the cost model of Paul & Mueller [14] for lookup tables, adders, etc. (Formulas for hardware costs appear in the full version [6].) The optimal parameter combination for double precision is $-\log_2(\hat{f}) = 55.67$, $-\log_2(\hat{n}) = 57.74$, and $-\log_2(e_0) = 13.92$. For the 68-bit internal precision we found the following combination: $e_0 = 2^{-13.51}$, $-\log_2 \hat{n} = 71.91$, and $-\log_2 \hat{f} = 68.9$. We conclude that multiplier dimensions 70×74 combined with a relative error bound $e_0 \leq 2^{-13.51}$ are a feasible choice of error parameters. These parameters lead to a savings in cost of 10.6% compared to the micro-architecture described in [15].

References

[1] R. Agarwal, F. Gustavson, and M. Schmookler. Series approximation methods for divide and square root in the power3 processor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 116–123, 1999.

[2] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM 360/370 model 91: floating-point execution unit. *IBM J. of Research and Development*, Jan. 1967.

[3] P. Beame, S. Cook, and H. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[4] 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 *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 96–105, 1999.

[5] D. DasSarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In *Proc. 12th IEEE Symposium on Computer Arithmetic*, pages 17–28, 1995.

[6] G. Even, P.-M. Seidel, and W. E. Ferguson. A parametric error analysis of Goldschmidt's division algorithm. full version, submitted for Journal publication, available at request, 2003.

[7] D. Ferrari. A division method using a parallel multiplier. *IEEE Trans. on Computers*, EC-16:224–226, Apr. 1967.

[8] M. J. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8):702–706, Aug. 1970.

[9] R. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, June 1964.

[10] D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3nd edition, 1998.

[11] E. V. Krishnamurthy. On optimal iterative schemes for high-speed division. *IEEE Transactions on Computers*, C-19(3):227–231, Mar. 1970.

[12] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Books. Prentice Hall, 2000.

[13] K. Mehlhorn and F. Preparata. Area-time optimal division for $t = \omega((logn)^{1+\epsilon})$. *Information and Computation*, 72(3):270–282, 1987.

[14] S. M. Mueller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[15] S. F. Oberman. Floating-point division and square root algorithms and implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 106–115, 1999.

[16] W. Paul and P.-M. Seidel. To Booth or Not to Booth? *INTEGRATION, the VLSI Journal*, 33:1–36, Jan. 2003.

[17] J. Reif and S. Tate. Optimal size integer division circuits. *SIAM Journal on Computing*, 19(5):912–924, Oct. 1990.

[18] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the amd-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[19] E. Schwarz, L. Sigal, and T. McPherson. CMOS floating point unit for the S/390 parallel enterpise server G4. *IBM J. of Research and Development*, 41(4/5):475–488, 1997.

[20] P.-M. Seidel. *On the Design of IEEE Compliant Floating-Point Units and their Quantitative Analysis*. PhD thesis, University of Saarland, Computer Science Department, Germany, 1999.

[21] N. Shankar and V. Ramachandran. Efficient parallel circuits and algorithms for division. *Information Processing Letters*, 29(6):307–313, 1988.

[22] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys*, 28(3):518–564, 1996.

[23] O. Spaniol. *Computer Arithmetic - Logic and Design*. Wiley, 1981.