

Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms

Marius Cornea-Hasegan, Microprocessor Products Group, Hillsboro, OR, Intel Corp.

Index words: floating-point, IEEE correctness, divide, square root, remainder

Abstract

The work presented in this paper was initiated as part of a study on software alternatives to the hardware implementations of floating-point operations such as divide and square root. The results of the study proved the viability of software implementations, and showed that certain proposed algorithms are comparable in performance to current hardware implementations. This paper discusses two components of that study:

- (1) A methodology for proving the IEEE correctness of the result of iterative algorithms that implement the floating-point square root, divide, or remainder operation.
- (2) Identification of operands for the floating-point divide and square root operations that lead to results representing difficult cases for IEEE rounding.

Some general properties of floating-point computations are presented first. The IEEE correctness of the floating-point square root operation is discussed next. We show how operands for the floating-point square root that lead to difficult cases for rounding can be generated, and how to use this knowledge in proving the IEEE correctness of the result of iterative algorithms that calculate the square root of a floating-point number. Similar aspects are analyzed for the floating-point divide operation, and we present a method for generating difficult cases for rounding. In the case of the floating-point divide operation, however, it is more difficult to use this information in proving the IEEE correctness of the result of an iterative algorithm than it is for the floating-point square root operation. We examine the restrictions on the method used for square root. Finally, we present possible limitations due to the finite exponent range.

Introduction

Floating-point divide, remainder, and square root are three important operations performed by computing systems today. The IEEE-754 Standard for Binary Floating-Point Arithmetic [1] requires that the result of a divide or square root operation be calculated as if in infinite precision, and then rounded to one of the two nearest floating-point numbers of the specified precision that surround the infinitely precise result (the remainder will always be exact).

Most processor implementations to date have used hardware-based implementations for divide, remainder, and square root. Recent research has led to software-based iterative algorithms for these operations that are expected to always generate the IEEE-correct result. Several advantages can be envisioned. Firstly, software-based algorithms lead to a possibly higher throughput than before because they are capable of being pipelined; secondly, they are easier to modify; and finally, they reduce the actual chip size because they do not have to be implemented in hardware.

Different algorithms are used depending on the target precision of the result, or on the particular architecture of the processor. In either case, performance and IEEE correctness have to be ensured. The expected levels of performance are possible due to improvements in the floating-point architecture of most modern processors. Proving the IEEE correctness of the results generated by the divide, remainder, and square root operations, which is of utmost importance for modern processors, is the object of this paper. A new methodology for achieving this goal is presented. Operands that lead to "difficult" cases for rounding are also identified, allowing better testing of the implementations for these operations. The method presented was verified through a mix of mathematical proofs, sustained by Mathematica [2], C language, and assembly language programs.

Correctness proofs following methods similar to those presented in this paper, in conjunction with careful and thorough verification and testing of the actual implementation, should ensure flawless floating-point divide, remainder, and square root operations on modern processors that adopt iterative, software-based algorithms.

Some of the mathematical notations used in the following sections are explained at the end of this paper.

General Properties of Floating-Point Computations and IEEE Correctness

Floating-point numbers are represented as a concatenation of a sign bit, an M-bit exponent field, and an N-bit significand field. Mathematically

$$f = \sigma \cdot s \cdot 2^e$$

where $\sigma = \pm 1$, $s \in [1, 2)$, $e \in [e_{\min}, e_{\max}] \cap \mathbf{Z}$, $s = 1 + k / 2^{N-1}$, $k \in \{0, 1, 2, \dots, 2^{N-1} - 1\}$, $e_{\min} = -2^{M-1} + 2$, and $e_{\max} = 2^{M-1} - 1$. Let \mathbf{F}_N be the set of floating-point numbers with N-bit significands and unlimited exponent range, and let $\mathbf{F}_{M,N}$ be the set of floating-point numbers with M-bit exponents and N-bit significands (no special values included). Note that $\mathbf{F}_{M,N} \subset \mathbf{F}_N \subset \mathbf{Q}^*$.

The IEEE-754 Standard for Binary Floating-Point Arithmetic allows several formats, but the most common are single precision (M=8, N=24), double precision (M=11, N=53), and double-extended precision (M=15, N=64). Even though there is only a finite number of real values that can be represented as floating-point numbers (which constitute a finite subset of the rational numbers), the total of $2 \cdot 2^M \cdot 2^N$ floating-point numbers or special values for a given precision can be huge. Verifying correctness of a binary floating-point operation for double-extended precision by exhaustive testing on a state-of-the-art workstation could easily take 2^{50} years. The only operation that lends itself to such testing, from among those considered herein, is the single precision square root.

The variable density of the floating-point numbers on the real axis implies that it would be difficult to have a requirement regarding the absolute error when approximating real values by floating-point numbers. Instead, most iterative algorithms are analyzed by trying to limit the relative error of the result being generated.

The IEEE-754 standard requires that the result of a divide or square root operation be calculated as if in infinite precision, and then rounded to one of the two nearest

floating-point numbers of the specified precision that surround the infinitely precise result. Special cases occur when this is outside the supported range. The IEEE standard specifies four rounding modes: to nearest (*rn*), to negative infinity (*rm*), to positive infinity (*rp*), and toward zero (*rz*).

In order to determine whether an iterative algorithm for an IEEE operation yields the correctly rounded result (in any rounding mode), we have to evaluate the error that occurs due to rounding in each computational step. Two measures are commonly used for this purpose. The first is the error of an approximation with respect to the exact result, expressed in fractions of an *ulp*, or unit in the last place. For the floating-point number $f = \sigma \cdot s \cdot 2^e \in \mathbf{F}_N$ given above, one *ulp* has the magnitude

$$1 \text{ ulp} = 2^{e-N+1}$$

An alternative is to use the relative error. If the real number x is approximated by the floating-point number a , then the relative error ϵ is determined by

$$a = x \cdot (1 + \epsilon)$$

The non-linear relationship between *ulps* and the corresponding relative error is illustrated in Figure 1 (where the outer envelopes mark the minimum and maximum values of the relative error), by Theorem 1, and also by Corollaries 1a and 1b of this theorem. For example, if $x > 1.000$ in Figure 4, but x is much closer to 1.000 than to 1.001, then if we approximate x by $a=1.001$, the relative error is close to the largest possible (when staying within 1 *ulp* of the approximation), i.e., close to $\epsilon = 1/8$ ($1/8 = 2^{-N+1}$ for $N = 4$).

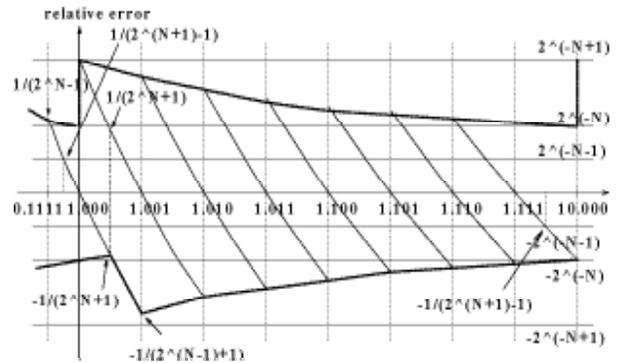


Figure 1: Relative error when approximating within 1 *ulp*, positive real numbers by floating-point numbers in \mathbf{F}_4

Theorem 1 (*ulp*-s versus relative error) Let $x \in \mathbf{R}^*$ be a real number, and $a \in \mathbf{F}_N$, $a = \sigma \cdot s \cdot 2^e$, $|\sigma| = 1$, $s = 1 + k/2^{N-1}$, $k \in \mathbf{Z}$, $0 \leq k \leq 2^{N-1} - 1$, $e \in \mathbf{Z}$ and $m \in \mathbf{R}$, $0 < m \leq 1$. Then

(a) For $k \neq 0$: $a = \sigma \cdot (1+k/2^{N-1}) \cdot 2^e \cong x$ within m ulp of $x \Leftrightarrow a = x \cdot (1+\epsilon)$, $\epsilon \in (-m/(2^{N-1}+k+m), m/(2^{N-1}+k-m))$

(b) For $k=0$: $a = \sigma \cdot 2^e \cong x$ within m ulp of $x \Leftrightarrow a = x \cdot (1+\epsilon)$, $\epsilon \in (-m/(2^{N-1}+m), m/(2^N-m))$

(For the sake of brevity, no proofs are included in this paper.)

Several corollaries can be derived from this property, but the following two are often useful.

Corollary 1a. Let $x \in \mathbb{R}^*$, and $a \in \mathbb{F}_N^*$. If $a \cong x$, within 1 ulp of x , then $a = x \cdot (1+\epsilon)$, with $|\epsilon| < 1/2^{N-1}$.

Corollary 1b. Let $x \in \mathbb{R}^*$, and $a \in \mathbb{F}_N^*$. If $a = x \cdot (1+\epsilon)$, with $|\epsilon| < 1/2^N$, then $a \cong x$, within 1 ulp of x .

Some properties used in IEEE correctness proofs are formulated in terms of errors expressed in ulp-s. However, it is easier to evaluate the errors introduced by each computational step in an iterative algorithm as relative errors. The two properties above, together with other similar ones, are used in going back and forth from one form to the other, as needed.

In order to show that the final result generated by a floating-point algorithm represents the correctly rounded value of the infinitely precise result x , one has to show that the exact result and the final result of the algorithm *before rounding*, y , lie within such an interval that, through rounding, they end up at the same floating-point number. Figure 2 illustrates these intervals over a binade (an interval delimited by consecutive integer powers of the base 2) when (a) only rounding to nearest, and (b) any IEEE rounding mode is acceptable. In this figure, the floating-point numbers are marked on the bottom real axis.

First we will discuss the square root, and then we will examine the divide and remainder operations.

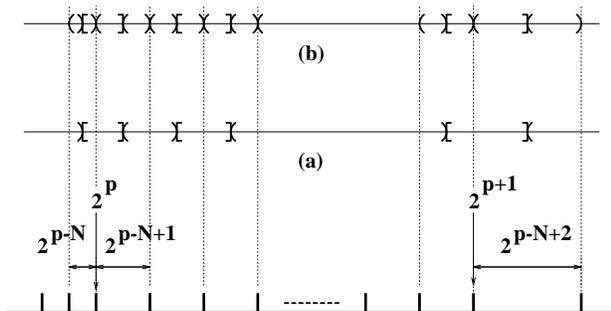


Figure 2: Intervals that condition (a) $x_m = y_m$, and (b) $x_{md} = y_{md}$

IEEE Correctness Proofs for Iterative Floating-Point Square Root Algorithms

We will first present the means to determine difficult cases for rounding in floating-point square root operations, and then we will show how to use this knowledge in proving the IEEE correctness of the result of iterative algorithms implementing the floating-point square root operation.

Iterative Algorithms for Floating-Point Square Root

Among the most common iterative algorithms for calculating the value of the square root of a floating-point number are those based on the Newton-Raphson method. Such algorithms are suggested in [3]. Starting with an initial guess of $1/\sqrt{a}$, a very good approximation of $1/\sqrt{a}$ is derived first in a number of iterations. From this, the value of \sqrt{a} can be calculated. For example, an algorithm that starts by determining the value of the root of $f(x) = 1/x^2 - a$, could have the following iteration:

$$e_i = (1/2 - 1/2 \cdot a \cdot y_i^2)_m$$

$$y_{i+1} = (y_i + e_i \cdot y_i)_m$$

where e_i is the error term, y_{i+1} is the next better approximation, and the subscript m denotes the IEEE rounding to nearest mode.

No matter what iterative algorithm is being used, one can easily evaluate the relative errors introduced in each computational step. We will show that if the relative error that we can derive is small enough, then the IEEE correctness of the generated result is proven. In practice though, the relative error of the final result is more often than not larger than required. We can compensate for the difference between what we can determine in terms of relative error and what is needed, by knowing the values of input arguments that lead to the most difficult cases for rounding. The following subsections will show how to achieve this.

Difficult Cases for Rounding

The lemma shown below allows difficult cases for rounding to be determined. For rounding to nearest, these are represented by values of the argument a of the square root function such that \sqrt{a} is different from, but very close to, a midpoint between two consecutive floating-point numbers. For rounding toward negative infinity, positive infinity, or zero, \sqrt{a} has to be different from, but very

shown above) just for $\delta = 1, -1, 2, -2, 3$, and -3 , two solutions are found for any N , and statements (a) and (b) in Theorem 2 are modified to reflect exclusion zones that are four times wider (in (b), the two inequalities were replaced by the more restrictive one)

(a) The distance $w_{\sqrt{A}}$ between \sqrt{A} and F satisfies

$$w_{\sqrt{A}} = |\sqrt{A} - F| > 1/2^{N-1}, \text{ except for}$$

$$A_1 = 2^{2N-2} + 2 \cdot 2^{N-1} \text{ if } e = 2 \cdot u, u \in \mathbf{Z}, \text{ and}$$

$$A_2 = 2^{2N} - 2 \cdot 2^N \text{ if } e = 2 \cdot u + 1, u \in \mathbf{Z}$$

(b) The distance $w_{\sqrt{a}}$ between \sqrt{a} and f satisfies

$$w_{\sqrt{a}} = |\sqrt{a} - f| > 2^{e/2-2N+3/2}, \text{ except for}$$

$$a_1 = (1+2^{-N+2}) \cdot 2^e \text{ if } e = 2 \cdot u, u \in \mathbf{Z}, \text{ and}$$

$$a_2 = (2-2^{-N+2}) \cdot 2^e \text{ if } e = 2 \cdot u + 1, u \in \mathbf{Z}$$

This property is used in conjunction with the one from Theorem 3.

Theorem 3. Let $a \in \mathbf{F}_N$, $a > 0$, $a = \sigma \cdot s \cdot 2^e$. If $\sqrt{a} \notin \mathbf{F}_N$, and A is determined by scaling a as specified in Lemma 1, then for any midpoint m between two consecutive numbers in \mathbf{F}_N , and for any midpoint M between two consecutive integers in $[2^{N-1}, 2^N)$, $M = k+1/2 \in [2^{N-1}, 2^N)$, $k \in \mathbf{Z}$

(a) The distance $w_{\sqrt{A}}$ between \sqrt{A} and M satisfies

$$w_{\sqrt{A}} = |\sqrt{A} - M| > 1/2^{N+3}$$

(b) The distance $w_{\sqrt{a}}$ between \sqrt{a} and m satisfies

$$w_{\sqrt{a}} = |\sqrt{a} - m| > 2^{e/2-2N-2}, \text{ if } e = 2 \cdot u, \text{ and}$$

$$w_{\sqrt{a}} = |\sqrt{a} - m| > 2^{e/2-2N-5/2}, \text{ if } e = 2 \cdot u + 1, u \in \mathbf{Z}$$

This theorem states that if \sqrt{a} is not representable as a floating-point number with an N -bit significand, then there are exclusion zones of known minimal width around any midpoint between two consecutive floating-point numbers, within which \sqrt{a} cannot exist. The minimum distance between \sqrt{a} and m , or equivalently, between \sqrt{A} and M , could be determined by examining instead the distance between A and M^2 , as shown in Figure 4. Just as in the case of Theorem 2, the minimal width of the exclusion zones can be extended by gradually eliminating the difficult cases for rounding toward nearest (starting with the most difficult cases). For example, if we solve the diophantine equations that determine these cases (as shown above) just for $\delta = 0, 1, -1, 2, -2, 3, -3$, and -4 , seven solutions are found for $N=24$, $N=53$, and $N=64$ (but not always from the same equation for all the three values of N), and statements (a) and (b) in Theorem 3 are modified to reflect exclusion zones that are 17 times wider (in (b), the two inequalities were replaced by the more restrictive one):

(a) The distance $w_{\sqrt{A}}$ between \sqrt{A} and M satisfies

$$w_{\sqrt{A}} = |\sqrt{A} - M| > 17/2^{N+3}$$

except for a set of known values A_1' through A_7' .

(b) The distance $w_{\sqrt{a}}$ between \sqrt{a} and m satisfies

$$w_{\sqrt{a}} = |\sqrt{a} - m| > 17 \cdot 2^{e/2-2N-5/2}$$

except for a set of known values a_1' through a_7' .

Some of the actual values of a_i' and of the corresponding A_i' (not shown here) were determined directly from mathematical expressions, but others were determined by C programs, using recursion. Note that it is not necessary to have the same number of solutions for different values of N .

This property is used, as explained below, in conjunction with that in the preceding Theorem 2.

IEEE Correctness of Iterative Floating-Point Square Root Algorithms

In order to prove that an iterative algorithm for calculating the square root of a floating-point number a generates a result that is IEEE correct, we have to show that the result R^* before rounding and the exact value of \sqrt{a} lead, through rounding, to the same floating-point number. If the result R^* before rounding and the exact \sqrt{a} are closer to each other than half of the minimum width of any exclusion zone determined as shown in Theorems 2 and 3, then $(R^*)_{rnd} = (\sqrt{a})_{rnd}$ for any IEEE rounding mode rnd . This is illustrated in Figure 2 (b) above and in Figure 5.

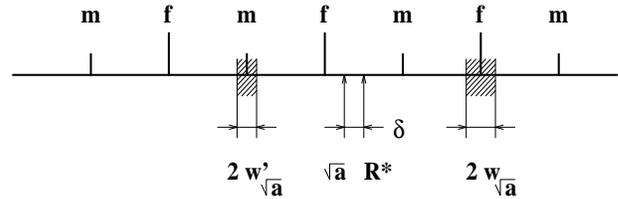


Figure 5: Exclusion zones around floating-point numbers f and around midpoints m between consecutive floating-point numbers

As in Theorem 2 (a), $1/2^{N+1} = 1/2^{N+1} \text{ ulp}$ in $\mathbf{F}_N = 1 \text{ ulp}$ in \mathbf{F}_{2N+1} , and in Theorem 3 (a), $1/2^{N+3} = 1/2^{N+3} \text{ ulp}$ in $\mathbf{F}_N = 1 \text{ ulp}$ in \mathbf{F}_{2N+3} , the above can be expressed in the following corollaries of these two theorems:

Corollary 2. If $a \in \mathbf{F}_N$, $a > 0$, $R^* \in \mathbf{R}$ and $|R^* - \sqrt{a}| \leq 1 \text{ ulp}$ in \mathbf{F}_{2N+1} , then $(R^*)_{rnd} = (\sqrt{a})_{rnd}$ for any rounding mode $rnd \in \{rm, rp, rz\}$

Corollary 3. If $a \in \mathbf{F}_N$, $a > 0$, $R^* \in \mathbf{R}$ and $|R^* - \sqrt{a}| \leq 1 \text{ ulp}$ in \mathbf{F}_{2N+3} , then $(R^*)_m = (\sqrt{a})_m$

In practice, the inequalities to be verified are

$$|R^* - \sqrt{a}| \leq (w_{\sqrt{a}})_{\min}$$

$$|R^* - \sqrt{a}| \leq (w_{\sqrt{a}})_{\min}$$

The method of proving the IEEE correctness of the result of an iterative algorithm that calculates the square root of a floating-point number can then be summarized as a sequence of steps:

Step 1. Evaluate the relative error ϵ of the final result before rounding

$$R^* = \sqrt{a} \cdot (1 + \epsilon)$$

where an upper bound for $|\epsilon|$ is known as $|\epsilon| \leq 2^{-p}$, for some given $p \in \mathbf{R}$, $p > 0$.

Step 2. Evaluate $|R^* - \sqrt{a}| = |\sqrt{a} \cdot \epsilon| \leq \sqrt{a} \cdot 2^{-p}$ and determine the minimum widths $(w_{\sqrt{a}})_{\min}$ and $(w_{\sqrt{a}})_{\min}$ for which the following hold

$$|R^* - \sqrt{a}| \leq (w_{\sqrt{a}})_{\min}$$

$$|R^* - \sqrt{a}| \leq (w_{\sqrt{a}})_{\min}$$

Step 3. For Theorems 2 and 3, determine a sufficient number of difficult cases for rounding, to allow augmenting the widths of the exclusion zones to the values determined in Step 2 (the smaller this new minimum width, the fewer the points we have to determine). Except possibly for these values, the iterative algorithm for calculating \sqrt{a} generates a result that is IEEE correct. Alternatively, instead of steps 1, 2, 3 above, we could use Theorem 1 to verify that the conditions in Corollaries 2 and 3 are satisfied.

Step 4. Verify directly that the algorithm generates IEEE correct results for the special points determined in Step 3. Once this is done, the algorithm is proven to generate IEEE correct results for all the possible input values of the argument.

The method presented above was applied to several software-based iterative algorithms for calculating the square root of a floating-point number. It proved to be of good practical value, as the number of special points to be verified directly was never larger than 20. It therefore represents a viable option for verifying correctness of this class of algorithms.

IEEE Correctness Proofs for Iterative Floating-Point Divide and Remainder Algorithms

As we did for square root, we will first present a method for determining difficult cases for rounding in floating-

point divide operations. We will show, however, that a perfect parallel with the square root is not possible.

The IEEE correctness of the floating-point remainder operation (which is always exact, and therefore not affected by the rounding mode) is a direct consequence of the IEEE correctness of the result of the floating-point divide operation, as

$$\text{rem}(a,b) = a - b \cdot \text{near}(a/b)$$

where $\text{near}(x)$ is the nearest integer to the real number x . Two problems arise. First, we must verify that $\text{near}(a/b)$ fits in an integer ([4] explains this). Second, we must counter the possibility of a double rounding error because what we calculate is actually

$$\text{rem}(a,b) = a - b \cdot \text{near}((a/b)_m)$$

To do this, we just need to compare $(a/b)_m$ and $(a/b)_r$, and apply a correction if needed. Note that this might occur only in the tie cases for the rounding to nearest performed in $\text{near}((a/b)_m)$.

It is thus straightforward to prove the IEEE correctness of the floating-point remainder operation, once we have proved it for the floating-point divide. Therefore, from this point on, we will focus on the floating-point divide operation.

Iterative Algorithms for Floating-Point Divide

Just as for square root, among the most common iterative algorithms for calculating the value of the quotient of two floating-point numbers, a/b , are those based on the Newton-Raphson method. Such algorithms are suggested also in [3]. Starting with an initial guess of $1/b$, a very good approximation of $1/b$ is derived first in a number of iterations. From this, the value of a/b can be calculated. For example, an algorithm that starts by determining the value of the root of $f(x) = b - 1/x$, could have the following iteration:

$$e_i = (1 - b \cdot y_i)_m$$

$$y_{i+1} = (y_i + e_i \cdot y_i)_m$$

where e_i is the error term, and y_{i+1} is the next better approximation.

Regardless of the particular iterative algorithm being used, one can evaluate the relative errors introduced in each computational step. We will show that if the relative error that we can derive is small enough, then the IEEE correctness of the generated result is easily proven. In practice though, the relative error of the final result is

larger than desired. In the case of the divide (unlike for the square root), we cannot easily compensate for the difference between what we can determine in terms of relative error and what is needed. In such cases, alternative methods have to be used, such as those presented in [3].

Difficult Cases for Rounding

For rounding to nearest, the difficult cases are represented by values of the arguments a and b of the divide operation such that a/b is different from, but very close to, a midpoint between two consecutive floating-point numbers. For rounding toward negative infinity, positive infinity, or zero, a/b has to be different from, but very close to, a floating-point number. We will show next how to determine the most difficult cases in each category.

The following lemma shows how to scale a and b to A and B respectively in order to make reasoning easier (a and b are assumed to be positive here).

Lemma 2. Let $N \in \mathbf{Z}$, $N \geq 3$, $a, b \in \mathbf{F}_N^*$, and rnd any IEEE rounding mode. Then

(a) a/b can be expressed as $a/b = A/B \cdot 2^E$, where $A \in \mathbf{Z} \cap \mathbf{F}_N$, $A \equiv 0 \pmod{2^{N-1}}$, $B \in [2^{N-1}, 2^N) \cap \mathbf{Z}$ and $(A/B)_{rnd} \in [2^{N-1}, 2^N) \cap \mathbf{Z}$. Let $A = 2^k \cdot A_1$, such that $A_1 \in [2^{N-1}, 2^N) \cap \mathbf{Z}$. If $A_1 < B$ then $k = N$, and if $A_1 > B$, then $k = N-1$.

(b) $a/b \in \mathbf{F}_N$ if and only if $A/B \in [2^{N-1}, 2^N) \cap \mathbf{Z}$.

(c) a/b is a midpoint between two consecutive floating-point numbers in \mathbf{F}_N if and only if A/B is an integer $+ 1/2$ in $[2^{N-1}, 2^N)$.

(d) If $a/b \notin \mathbf{F}_N$, then $a/b \notin \mathbf{F}_{N+1}$ (this is to say that a/b cannot be a midpoint between two consecutive floating-point numbers in \mathbf{F}_N ; the observation is necessary, as $\mathbf{F}_N \subset \mathbf{F}_{N+1}$).

Note that above, $A_1/B = s_a/s_b$ (ratio of the significands).

We can now determine the most difficult cases for rounding.

Consider first the cases of rounding toward negative infinity, positive infinity, or zero. If $a/b \notin \mathbf{F}_N$ (it is not representable as a floating-point number with N bits in the significand), then how close can a/b be to a floating-point number $f \in \mathbf{F}_N$? Because the values of A/B fall in $[2^{N-1}, 2^N)$, where floating-point numbers with N -bit significands are integers, and $1 \text{ ulp} = 1$, this can be re-formulated as “how close can A/B be to an N -bit integer number F ?” To answer this, we have to solve the equation

$$A/B = q + \delta/B$$

for increasing values of $\delta \in \mathbf{Z}^*$, and for $q \in [2^{N-1}, 2^N) \cap \mathbf{Z}$. This leads to two cases, depending on whether $A_1 < B$, or $A_1 > B$ (see Lemma 2 (a) above).

If $A_1 < B$, according to Lemma 2, the equation above becomes

$$2^N \cdot A_1 = B \cdot q + \delta$$

If $A_1 > B$, according also to Lemma 2, the equation becomes

$$2^{N-1} \cdot A_1 = B \cdot q + \delta$$

In both cases, $q \in [2^{N-1}, 2^N) \cap \mathbf{Z}$.

We can solve the diophantine equations above for $\delta = 1, -1, 2, -2, 3, -3, \dots$ and B fixed. The method we applied was to use Euclid’s algorithm to determine the greatest common divisor of 2^k and B , $\gcd(2^k, B)$ (where $k=N$ or $k=N-1$), and to express it as a linear combination of B and 2^k . This yields a base solution, from which all the admissible solutions can be derived (see [5] for using Euclid’s algorithm to determine above solutions). The most difficult cases are obtained for $\delta=1$ and $\delta=-1$. The number of solutions is very large (unlike for the square root), which makes it impractical to verify them all for a given divide algorithm. For example, for $N=24$, $A_1 < B$, and $\delta=1$, the number of solutions is 1289234. One example is

$$0.a49d25_H / 0.ff7e75_H = .a49e2300000100018b\dots_H$$

which is very close to, but greater than, $.a49e23_H$.

Consider next the case of rounding to nearest. If $a/b \notin \mathbf{F}_N$ (it is not representable as a floating-point number with N bits in the significand), then how close can it be to a midpoint between two consecutive floating-point numbers in \mathbf{F}_N ? This can be re-formulated as “how close can A/B be to an N -bit integer number plus $1/2$?” To find out, we have to solve the equation

$$A/B = q + 1/2 + \delta/B$$

for increasing values (in absolute value) of δ , $2 \cdot \delta \in \mathbf{Z}^*$, and for $q \in [2^{N-1}, 2^N) \cap \mathbf{Z}$. We can solve the diophantine equations above for $\delta = 1/2, -1/2, 1, -1, 3/2, -3/2, \dots$ and B fixed. The method applied was similar to that described for the other three rounding modes. The most difficult cases for rounding are obtained for $\delta = 1/2$ and $\delta = -1/2$, when B is odd. The number of solutions is again very large. For example, for $N = 24$, $A_1 < B$, and $\delta = -1/2$, the number of solutions is 1285649. One example is

$$0.c8227b_H / 0.e73317_H = .dd9a537ffff724506a\dots_H$$

which is very close to, but less than, $.dd9a53_H + 1/2 \text{ ulp}$.

For $N=24$, $A_1 < B$, and $\delta=1/2$, the number of solutions is 1287219. An example is

$$0.\text{ac}1228_{\text{H}} / 0.\text{b}461\text{d}1_{\text{H}} = .\text{f}43467800000\text{b}5\text{a}8\dots_{\text{H}}$$

which is very close to, but greater than, $.\text{f}43467_{\text{H}}+1/2 \text{ ulp}$.

The method of determining values of the arguments a and b that generate values of a/b that are difficult to round is also useful in generating good quality test vectors for any implementation of the floating-point divide operation (not just for iterative algorithms), that can be added to others chosen by different criteria. In addition, this method can help in proving IEEE correctness of certain iterative algorithms for calculating the result of the floating-point divide operation, as shown below.

General Properties

Two main properties, Theorems 4 and 5, are used in the proposed method of proving the IEEE correctness of the result generated by iterative floating-point divide algorithms. They are expressed in terms of the scaled values A and B of the arguments a and b , as explained by Lemma 2 above.

Theorem 4. Let $a, b \in \mathbf{F}_N$, $a = \sigma_a \cdot s_a \cdot 2^{e-a}$, $b = \sigma_b \cdot s_b \cdot 2^{e-b}$. If $a/b \notin \mathbf{F}_N$, and A and B are determined by scaling a and b respectively as specified in Lemma 2, then for any $f \in \mathbf{F}_N$, and for any integer $F \in [2^{N-1}, 2^N)$

(a) The distance $w_{A/B}$ between A/B and F satisfies

$$w_{A/B} = |A/B - F| > 1/2^N$$

(b) The distance $w_{a/b}$ between a/b and f satisfies

$$w_{a/b} = |a/b - f| > 2^{e-a-e-b-2N}, \text{ if } s_a < s_b, \text{ and}$$

$$w_{a/b} = |a/b - f| > 2^{e-a-e-b-2N+1}, \text{ if } s_a > s_b$$

This theorem states that if a/b is not representable as a floating-point number with an N -bit significand, then there are exclusion zones of known minimal width around any floating-point number, within which a/b cannot exist. The minimal width of the exclusion zones cannot be easily extended in this case. If we try to gradually eliminate the difficult cases for rounding toward negative infinity, positive infinity, or zero (starting with the most difficult cases), we find too many cases to make it practical to verify directly.

Theorem 5. Let $a, b \in \mathbf{F}_N$, $a = \sigma_a \cdot s_a \cdot 2^{e-a}$, $b = \sigma_b \cdot s_b \cdot 2^{e-b}$. If $a/b \notin \mathbf{F}_N$, and A and B are determined by scaling a and b respectively as specified in Lemma 2, then for any $m \in \mathbf{F}_{N+1} - \mathbf{F}_N$ (midpoint between two consecutive floating-point numbers in \mathbf{F}_N), and for any M , integer $+1/2 \in [2^{N-1}, 2^N)$

(a) The distance $w_{A/B}$ between A/B and M satisfies

$$w_{A/B} = |A/B - M| > 1/2^{N+1}$$

(b) The distance $w_{a/b}$ between a/b and m satisfies

$$w_{a/b} = |a/b - m| > 2^{e-a-e-b-2N-1}, \text{ if } s_a < s_b, \text{ and}$$

$$w_{a/b} = |a/b - m| > 2^{e-a-e-b-2N}, \text{ if } s_a > s_b$$

This theorem states that if a/b is not representable as a floating-point number with an N -bit significand, then there are exclusion zones of known minimal width around any midpoint between two consecutive floating-point numbers, within which a/b cannot exist. Just as in the case of Theorem 4, the minimal width of the exclusion zones cannot be easily extended.

This property is used, as explained below, in conjunction with that in the preceding Theorem 4.

IEEE Correctness of Iterative Floating-Point Divide Algorithms

In order to prove that the result of an iterative algorithm for calculating the quotient a/b is IEEE correct, we have to show that the result R^* before rounding and the exact value of a/b lead, through rounding, to the same floating-point number. If the result R^* before rounding and the exact a/b are closer to each other than half of the minimum width of any exclusion zone determined as shown in Theorems 4 and 5, then $(R^*)_{\text{rnd}} = (a/b)_{\text{rnd}}$ for any IEEE rounding mode rnd . Figure 5 above, which illustrates this idea for the square root, is applicable in this case too.

As in Theorem 4 (a), $1/2^N = 1/2^N \text{ ulp}$ in $\mathbf{F}_N = 1 \text{ ulp}$ in \mathbf{F}_{2N} , and in Theorem 5 (a), $1/2^{N+1} = 1/2^{N+1} \text{ ulp}$ in $\mathbf{F}_N = 1 \text{ ulp}$ in \mathbf{F}_{2N+1} , the above can be expressed in the following corollaries of these two theorems:

Corollary 4. If $a, b \in \mathbf{F}_N$, $R^* \in \mathbf{R}$ and $|R^* - a/b| \leq 1 \text{ ulp}$ in \mathbf{F}_{2N} , then $(R^*)_{\text{rnd}} = (a/b)_{\text{rnd}}$ for any rounding mode $\text{rnd} \in \{rm, rp, rz\}$

Corollary 5. If $a, b \in \mathbf{F}_N$, $R^* \in \mathbf{R}$ and $|R^* - a/b| \leq 1 \text{ ulp}$ in \mathbf{F}_{2N+1} , then $(R^*)_m = (a/b)_m$

In practice, the inequalities to be verified are

$$|R^* - a/b| \leq (w_{a/b})_{\min}$$

$$|R^* - a/b| \leq (w_{a/b})'_{\min}$$

The method of proving the IEEE correctness of the result of an iterative algorithm that calculates the quotient of two floating-point numbers can then be summarized as a sequence of steps:

Step 1. Evaluate the relative error ε of the final result before rounding:

$$R^* = a/b \cdot (1 + \varepsilon)$$

where an upper bound for $|\varepsilon|$ is known as $|\varepsilon| \leq 2^{-p}$, for some given $p \in \mathbf{R}$, $p > 0$.

Step 2. Evaluate $|R^* - a/b| = |(a/b) \cdot \varepsilon| \leq |a/b| \cdot 2^{-p}$. If the following hold

$$|a/b| \cdot 2^{-p} \leq (w_{a/b})_{\min}$$

$$|a/b| \cdot 2^{-p} \leq (w_{a/b})_{\min}$$

then the algorithm generates IEEE-correct results for any input values a and b .

The method presented above was applied to software-based iterative algorithms for calculating the quotient of two floating-point numbers, when sufficient extra precision existed for the intermediate computational steps to allow the conditions above to be satisfied. Whenever this is not possible, alternative methods, such as the one presented in [3], have to be used.

It is worth mentioning that Corollaries 4 and 5 above were also obtained independently, starting from the following Lemma:

Lemma 3. Let $a, b \in \mathbf{F}_N$, $b \neq 0$. Then

- (a) Either $a/b \in \mathbf{F}_N$ (the exact a/b is representable with N bits in the significand), or a/b is periodic.
- (b) If $a/b \notin \mathbf{F}_N$ (it is periodic), then its infinite representation in binary cannot contain more than $N-1$ consecutive zeroes past the first binary one, and it cannot contain more than $N-1$ consecutive ones.

The Effect of the Limited Exponent Range

A note should be made here regarding the effect of the limited exponent range in iterative algorithms. If we analyze the algorithm by assuming valid inputs in a given floating-point format, and a certain precision (significand size) is ensured for all the computational steps, we may draw the conclusion (possibly using methods such as those presented above) that the final result is IEEE correct. It is possible though for an intermediate computational step to overflow, underflow (due to the limited exponent range), or to lose precision (due to the limited exponent range and to the limited significand size). Such cases have to be identified by carefully examining every computation step and, if they exist, the input operands that can lead to them have to be singled out. For such operands, alternate software algorithms are expected to be used in order to generate the IEEE-correct results, most likely by

appropriate scaling of the input values and corresponding scaling back of the results.

Results

There are two main results of the work described in this paper.

First, a new method of proving IEEE correctness of the result of iterative algorithms that calculate the square root of a floating-point number was devised. This included establishing theoretical properties that also allowed determination of difficult cases for any of the four IEEE rounding modes, and for any desired precision. The method proposed for proving the IEEE correctness was successfully used for several algorithms that calculate the square root of single precision, double precision, and double-extended precision floating-point numbers.

Second, a parallel was drawn for iterative algorithms that calculate the quotient of two floating-point numbers. Again, the theoretical study allowed determination of difficult cases for rounding for any precision (single, double, or double-extended). The method proposed for proving the IEEE correctness of the result was successfully used this time only for algorithms that calculate the quotient of two single precision floating-point numbers. The reason for this limitation was clearly identified.

Overall, the study helped to demonstrate that efficient, IEEE-correct, software-based algorithms can compete with hardware-based implementations for floating-point square root and divide.

Discussion

The methods proposed are general and can be applied to any iterative algorithm that implements the floating-point square root or divide operation. Their usefulness was verified for software-based algorithms.

The method proposed for proving the IEEE correctness of the result for the floating-point square root operation should work for any iterative algorithm. Based on this method, it is possible to completely automate the verification process, and to build an IEEE correctness checker that would take as input the algorithm to be verified, together with the precision and rounding mode for every step. The method proposed for proving the IEEE correctness of the result for the floating-point divide operation will only work if there is sufficient extra precision in the intermediate calculations with respect to the precision of the final result. This will hold mostly for single-precision computations, but might also be true for higher precisions, depending on the particular implementation.

Methods to generate cases of difficult operands for rounding were also described, which are independent of any particular algorithm. They can be used to generate test vectors for any desired precision.

Conclusion

The two goals stated in the beginning of this paper were reached: a methodology for proving the IEEE correctness of the results for iterative algorithms that implement the square root, divide, and remainder operations was established, and operands that lead to results that constitute difficult cases for rounding were identified. This work is important to Intel Corporation because it proves the existence and viability of alternatives to the hardware implementations of these operations.

Notation

Some of the mathematical notations used in this paper are given here for reference:

N	the set of natural numbers
Z	the set of integer numbers
Q	the set of rational numbers
R	the set of real numbers
F_N	the set of floating-point numbers with N-bit significands and unlimited exponent range
F_{M,N}	the set of floating-point numbers with M-bit exponents and N-bit significands
$a \in S$	a is a member of the set S
$S \cap T$	the intersection of sets S and T
$S \cup T$	the union of sets S and T
$S \subset T$	the set S is a subset of set T
$S - T$	the difference of sets S and T
[l,h]	the set of real values x, $l \leq x \leq h$
(l,h)	the set of real values x, $l < x < h$
$x \approx y$	the real number x is approximately equal to the real number y
$p \Leftrightarrow q$	statement p is equivalent to statement q
$m \equiv n \pmod{p}$	integers m and n yield the same remainder when divided by the positive integer p

In all the cases of sets, an asterisk (*) indicates the absence of the value 0, e.g., $\mathbf{R}^* = \mathbf{R} \setminus \{0\}$.

Acknowledgments

The author thanks Roger Golliver from Intel Corporation, Peter Markstein from Hewlett-Packard Company, and Marius Dadarlat from Purdue University for their support, ideas, and/or feedback regarding parts of the work presented in this paper.

References

- [1] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985.
- [2] Wolfram, S., Mathematica – A System for Doing Mathematics by Computer, Addison-Wesley Publishing Company, 1993.
- [3] Markstein, P., Computation of Elementary Functions on the IBM RISC System/6000 Processor, *IBM Journal*, 1990.
- [4] *Pentium® Pro Family Developer's Manual*, Intel Corporation, 1996, pp. 11-152 to 11-154.
- [5] Stark, H. M., *An Introduction to Number Theory*, The MIT Press, Cambridge MA, 1995, pp. 16-50.

Author's Biography

Marius Cornea-Hasegan is a staff software engineer with Microcomputer Software Labs at Intel Corporation in Hillsboro, OR. He holds an M.Sc. in Electrical Engineering from the Polytechnic Institute of Cluj, Romania, and a Ph.D. in Computer Sciences from Purdue University, in West Lafayette, IN. His interests include mainly floating-point architecture and algorithms, individually or as parts of a computing system, and formal verification. His e-mail address is marius.cornea@intel.com.