Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms

Marius A. Cornea-Hasegan, Roger A. Golliver
marius.cornea@intel.com, roger.a.golliver@intel.com
Intel Corporation, Hillsboro, OR

Peter Markstein
markstein@hpl.hp.com
Hewlett-Packard Co, Palo Alto, CA

Abstract

This paper describes a study of a class of algorithms for the floating-point divide and square root operations, based on the Newton-Raphson iterative method. The two main goals were:

(1) Proving the IEEE correctness of these iterative floating-point algorithms, i.e. compliance with the IEEE-754 standard for binary floating-point operations [1]. The focus was on software driven iterative algorithms, instead of the hardware based implementations that dominated until now.

(2) Identifying the special cases of operands that require software assistance due to possible overflow, underflow, or loss of precision of intermediate results.

This study was initiated in an attempt to prove the IEEE correctness for a class of divide and square root algorithms based on the Newton-Rapshson iterative methods. As more insight into the inner workings of these algorithms was gained, it became obvious that a formal study and proof were necessary in order to achieve the desired objectives. The result is a complete and rigorous proof of IEEE correctness for floating-point divide and square root algorithms based on the Newton-Raphson iterative method. Even more, the method used in proving the IEEE correctness of the square root algorithm is applicable in principle to any iterative algorithm, not only based on the Newton-Raphson method. Conditions requiring Software Assistance (SWA) were also determined, and were used to identify cases when alternate algorithms are needed to generate correct results. Overall, this is one important step toward flawless implementation of these floating-point operations based on software implementations.

Introduction

The floating-point algorithms for the floating-point divide and square root operations discussed in this paper differ fundamentally from those in the Intel or Hewlett-Packard processors, in that they can be implemented in software and are based on Newton-Raphson iterations. This approach allows for potentially higher throughput, as the divide, square root, and remainder operations are in this case pipelineable. It also allows for easier modification of the algorithms, should this be needed, and as a side benefit reduces the size of the chip that would adopt them. The decision to study such an approach was also influenced by the availability of relatively new algorithms [2] that were expected to generate the correct result and to set correctly the IEEE status flags in any IEEE rounding mode with only one computational sequence, and without the necessity of performing a correction step at the end. Performance and IEEE correctness remain the two main characteristics that have to be ensured. The expected levels of performance are possible in modern processors due to advanced features in their floating-point architecture (notably the fused multiply-add operation). Several variations of these algorithms can be developed, depending on whether the goal is to maximize throughput or to minimize latency, and also on the particular precision of the result. For example, different variants of floating-point divide and of floating-point square root were designed, verified, and implemented for single precision, double precision, and for double-extended precision floating-point computations. Algorithms for floating-point remainder are directly derived from those for the divide operation. All or part of these algorithms could be used by compilers that might inline the appropriate sequence for each situation, for non-native instruction emulation, for floating-point emulation libraries used by operating systems, in mathematical libraries, and in binary translators. The object of the work presented in this paper was to prove the IEEE correctness of the results generated by the divide, square root and remainder operations, of utmost importance for modern processors, often designed to excel also in floating-point. This study also identified special cases of operands that require software assistance (SWA) due to possible overflow, underflow, or loss of precision of intermediate

results. These are situations that require alternate algorithms to generate correct results. A mix of mathematical proofs [3] [4], sustained by Mathematica [5], C and assembly language programs reached the desired goal.

The result of this work is a complete and rigorous proof of IEEE correctness for floating-point divide and square root algorithms based on the Newton-Raphson iterative method. Correctness, as specified by the IEEE-754 standard for binary floating-point operations, was proven for all the proposed algorithms. This constitutes an important step toward flawless implementation of floating-point operations in present day processors.

Floating-Point Algorithms Correctness Proofs

The following sections detail the issues of IEEE correctness for floating-point divide and square root, and outline correctness proofs for the algorithms implementing them. Since software solutions are considered, it will be assumed that the processor using them does not have single floating–point instructions for these operations. It will be assumed that it has instead instructions that can be used to start the iterative process of calculating the result. For divide, this instruction would have to provide an initial approximation for 1/b, if a/b has to be calculated (in certain special situations, as will be seen, it should provide the result for a/b). For square root, the corresponding instruction should return an approximation for $1/\sqrt{a}$, if $\sqrt{a}$ has to be calculated. The known precision of the initial approximation is in both cases the starting point in proving that the proposed algorithms yield the IEEE correct results after a prescribed number of iteration steps.

General Properties of Floating-Point Computations. 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 (it will be assumed throughout this paper that $N \in \mathbf{Z}$, $N \geq 3$). Mathematically:

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

where $\sigma = \pm 1$, $s \in [1,2)$, $e \in [e_{min}, e_{max}] \cap \mathbf{Z}$, $s = 1 + k / 2^{N-1}$, $k \in \{0, 1, 2, \ldots, 2^{N-1}-1\}$, $e_{min} = -2^{M-1} + 2$, 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 computations allows a limited number of formats, but internally several more computation models may be supported. Even though there is only a finite number of real values that can be represented as floating-point numbers (which are a finite subset of the rational numbers), the total of order O ($2 \times 2^M \times 2^{N-1}$) 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 shows that it is not practical to try and set a limit on the absolute error when approximating real values by floating-point numbers. Instead, as will be seen in our case, algorithms try 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, denoted also as u (f,N), has the magnitude:

$$1 \text{ ulp} = u(f,N) = 2^{e-N+1}$$

The unit in the last place is defined also for real numbers not in $\mathbf{F}_N$, but in the context of operating with floating-point numbers in $\mathbf{F}_N$. For $x \notin \mathbf{F}_N$, if $x \in [2^e, 2^{e+1})$, then one ulp is:

$$1 \text{ ulp} = u(x,N) = 2^{e-N+1}$$

When making a statement about x being within 1 *ulp* of y, the smaller of u (x,N) and u (y,N) is meant.

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

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

The non-linear relationship between ulps and the corresponding relative error is illustrated in Figure 1, and also by Properties 1 and 2 below. (Note that as a marker, we will use a small triangle at the end of the line concluding a property or theorem.)

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

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

Some properties used in the correctness proofs that follow are formulated in terms of errors expressed in ulps. On the other hand, 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.

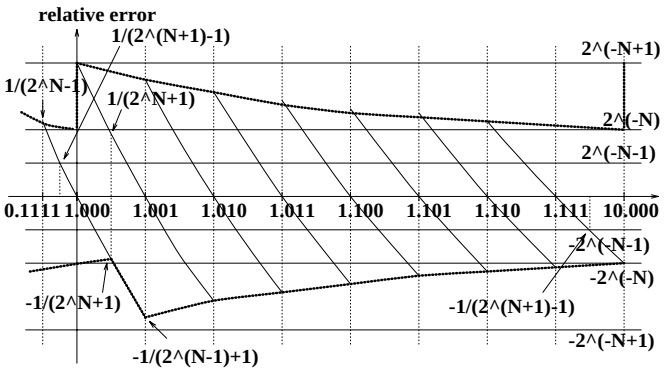

Fig. 1. Relative error when approximating, within 1 ulp, positive real numbers by floating-point numbers in $\mathbf{F}_4$ (N = 4). The envelopes lead to properties 1 and 2.

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 when (a) only rounding to nearest *(rn)*, and (b) any IEEE rounding mode *(rnd)* is acceptable. In addition, one has to prove that the overflow, underflow, and inexact IEEE status flags are set correctly as a result of the computation. It is assumed that the invalid and divide-by-zero status flags are set correctly when examining the input operands in the first computation step, and that the rest of the iterative computation will not take place in such cases, but rather the IEEE mandated result will be provided.

Floating-Point Divide Correctness Proof

The floating-point divide algorithms considered are based on the Newton-Raphson iterative method. If a/b needs to

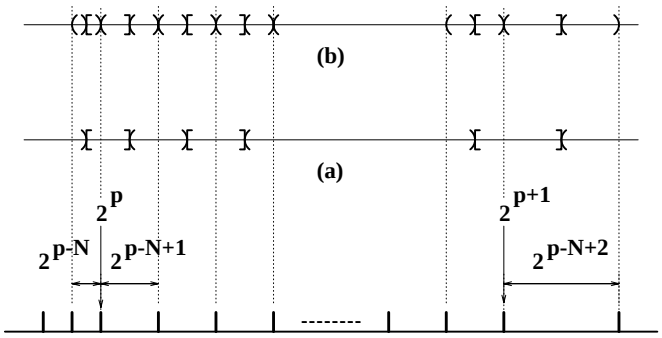

Fig. 2. Intervals on which conditions (a) $x_{rn} = y_{rn}$, and (b) $x_{rnd} = y_{rnd}$ are verified

be computed, a number of Newton-Raphson iterations first calculate an approximation of 1/b, using the function

$$f(y) = b - 1/y$$

The iteration step is:

$$e_n = (1 - b \cdot y_n)_{rn}$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn}$$

where the subscript *rn* denotes the IEEE rounding to nearest mode.

Once a sufficiently good approximation y of 1/b is determined, $q = a \cdot y$ approximates a/b. From case to case, this might need further refinement, which requires only a few more computational steps. The Newton-Raphson method has the advantages of converging at an almost quadratic rate and of being self-correcting. This latter property allows modification of certain computational steps to reduce latency, without affecting the correctness of the final result. For example, the computational step that calculates e_n may sometimes be replaced by

$$e_n = e_{n-1}^2$$

In order to show that the final result generated by the floating-point divide algorithm represents the correctly rounded value of the infinitely precise result a/b in any rounding mode, we have to prove that the exact value of a/b and the final result q* of the algorithm before rounding belong to the same interval among those shown in Figure 2 (b). Then

$$(a/b)_{rnd} = (q^*)_{rnd}$$

where *rnd* is any IEEE rounding mode.

The algorithms proposed for floating-point divide (as well as for square root), are designed as hinted by the Newton-Raphson iteration step shown above, assuming the availability of a floating-point fused multiply-add

operation, FMA, that performs both the multiply and add operations with only one rounding error.

Several variations of floating-point divide algorithms were proven IEEE correct. There are different variants for different precision calculations, and different variants depending on whether the purpose is to minimize latency, or to maximize throughput. The cases whose correctness is the most difficult to prove are those in which the precision of the final result is identical to that of the intermediate computational steps. This might be the case of the double-extended precision calculations, but it will be implementation dependent. After presenting a few important properties, we will illustrate a double-extended precision case.

As several implementations of Newton-Raphson algorithms were used, there are also several variations of iteration steps. A number of properties, not shown here, allow determination of the relative error of the new approximation resulting from the iteration step, based on the relative error of the previous approximation. Properties 1 and 2, shown previously, allow translation of the relative error into ulps (or vice versa). This is useful in facilitating application of the theorems discussed below.

Lemma 1. Let a, b $\in$ $\mathbf{F}_N$. The following hold:

(1) The exact quotient a/b represented in binary is either representable as a number in $\mathbf{F}_N$, or it is periodic.

(2) If a/b is a periodic number represented in binary, then its infinite representation cannot contain more than N - 1 consecutive binary ones, or more than N - 1 consecutive binary zeroes past the first binary one. Δ

(by periodic numbers, we mean e.g. 0.(011) = 0.011011… but also numbers that have a non-repeating part preceding the period, as in 1.01(1011) = 1.0110111011…).

This lemma is used in the proof of Theorem 1 that follows. It is also useful in determining how close a/b can be to a floating-point number or to a midpoint between two consecutive floating-point numbers, which helps identify difficult cases for rounding [4].

Theorem 1. Let a, b $\in$ $\mathbf{F}_N$, such that a/b $\notin$ $\mathbf{F}_N$, q* $\in$ $\mathbf{R}$, $N_1 \in \mathbf{N}$, $N_1 \geq 2 \cdot N + 1$, and *rnd* is any of the four IEEE rounding modes for rounding to N bits.

If q* is within 1 ulp of a/b in $\mathbf{F}_{N1}$ (on N_1 bits), then $(q^*)_{rnd} = (a/b)_{rnd}$ for any IEEE rounding mode *rnd* . Δ

This property can be applied when the precision of the intermediate computational steps is more than twice that of the final result. If the final result of the iterative algorithm, before rounding, satisfies the condition specified above, then after rounding it will represent the correctly rounded value of a/b, in any IEEE rounding mode. This is the case of one of the algorithms for single precision computations. The reason behind this property is that if a/b is not representable as a floating-point number, then there is an exclusion zone of known minimal width around any floating-point number within which a/b cannot exist. There is also an exclusion zone of known minimal width around any midpoint between two consecutive floating-point numbers [4]. If the result q* before rounding and the exact a/b are closer to each other than half of the minimum width of any exclusion zone, then $(q^*)_{rnd} = (a/b)_{rnd}$ for any IEEE rounding mode *rnd* (refer also to Figure 2 (b) above).

The exclusion zones around floating-point numbers in $\mathbf{F}_N$, and around midpoints between consecutive floating-point numbers are specified by the following two properties, derived from Theorems 4 (b) and 5 (b) of [4]:

Theorem A. Let a,b$\in$$\mathbf{F}_N$. If a/b$\notin$$\mathbf{F}_N$, then for any f$\in$$\mathbf{F}_N$, the distance between a/b and f satisfies

$$| a/b - f | > 2^{-N} \cdot u (a/b, N) \qquad \Delta$$

Theorem B. Let a,b$\in$$\mathbf{F}_N$. For any m$\in$$\mathbf{F}_{N+1}$-$\mathbf{F}_N$ (midpoint between two consecutive floating-point numbers in $\mathbf{F}_N$), the distance between a/b and m satisfies

$$| a/b - m | > 2^{-N-1} \cdot u (a/b, N) \qquad \Delta$$

The following theorems are expressed in terms of values of a and b scaled by powers of 2, i.e. in terms of integers A and B, such that B $\in$ $[2^{N-1}, 2^N)$, and A/B $\in$ $[2^{N-1}, 2^N)$, where A/B does not have to be an integer. A property not shown here links the statements in terms of A and B, to those in terms of a and b. Note that for values in $[2^{N-1}, 2^N) \cap \mathbf{F}_N$, 1 ulp = 1, which simplifies somewhat the proofs. Properties very close or identical to Theorems 2, 3, and 4 have proofs outlined in [2].

Theorem 2. Let B $\in$ $\mathbf{Z}$, $2^{N-1} \leq B \leq 2^N - 2$. Let y $\in$ $\mathbf{F}_N$, that approximates 1/B within 1 ulp. Then, one application of the relations:

$$e = (1 - B \cdot y)_{rn}$$

$$y' = (y + e \cdot y)_{rn}$$

yields y' that is the correctly rounded-to-nearest value (in the IEEE sense) of 1/B, $y' = (1/B)_{rn}$. Δ

Note that the computation must be performed by an FMA operation for floating-point numbers.

This theorem states that once the Newton-Raphson iteration has brought the approximation y of 1/B within 1 ulp of 1/B, then one more iteration yields $y' = (1/B)_{rn}$. This is a condition that is useful in applying Theorem 3 below. Note though that Theorem 2 excludes the case when $B = 2^N - 1$ (i.e. when B, represented in binary,

consists only of ones). This constitutes a "difficult" case for the rounding to nearest, and is covered by Theorem 4.

Theorem 3. Let $A \in \mathbf{Z} \cap \mathbf{F}_N$, $A \equiv 0 \pmod{2^{N-1}}$, $B \in \mathbf{Z}$, $2^{N-1} \le B < 2^N$, and *rnd* one of the IEEE rounding modes, such that $2^{N-1} \le (|A / B|)_{rnd} < 2^N$.

If $y \in \mathbf{R^*}$ is a real number within 1/2 ulp of 1/B (non-strict relation), and $q \in \mathbf{F}_N$, $q \cong A / B$, within 1 ulp of A / B, then one application of:

$$r = (A - B \cdot q)_{rn}$$

$$q' = (q + r \cdot y)_{rnd}$$

yields $q' = (A / B)_{rnd}$. Δ

Note: for $y \in \mathbf{R^*}$ to be within 1/2 ulp of 1/B (non-strict relation), it is sufficient to have $y \in \mathbf{F}_N$, $y = (1/B)_{rn}$.

Theorem 3 shows that once we have $y = (1/B)_{rn}$ (which can be verified using Theorem 2) and q within 1 ulp of A/B, one application of the two relations above generates the correctly rounded result of the divide operation A/B, in any IEEE rounding mode *rnd*. Note that when calculating r, rounding to nearest, *rn*, is used.

The next theorem covers the difficult case of $B = 2^N - 1$.

Theorem 4. Let $A \in \mathbf{Z} \cap \mathbf{F}_N$, $A \equiv 0 \pmod{2^{N-1}}$, $B \in \mathbf{Z}$, $B = 2^N - 1$, $y \in \mathbf{F}_N$, y within 1 ulp of 1/B, *rn* the IEEE rounding to nearest mode, and *rnd* any of the four IEEE rounding modes, such that $2^{N-1} \le (|A/B|)_{rnd} < 2^N$.

Let $q \in \mathbf{F}_N$, q within 1 ulp of A/B. If *rnd* = *rn* and $|q| \ge (|A/B|)_{rn}$, or if *rnd* $\in$ {*rz*, *rm*, *rp*}, then one application of the relations:

$$r = (A - B \cdot q)_{rn}$$

$$q' = (q + r \cdot y)_{rnd}$$

leads to $q' = (A/B)_{rnd}$. Δ

This last theorem gives the necessary conditions for the result of the floating-point divide operation to be correct

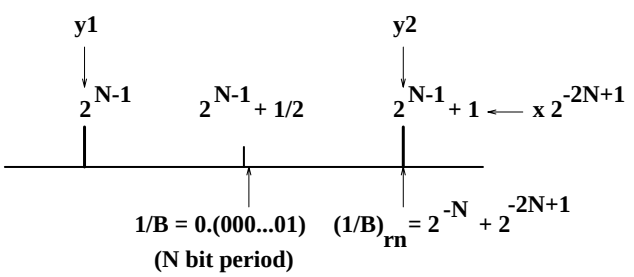

Fig. 3. The value of 1/B for $B = 2^N - 1$.

when the significand of the denominator consists only of binary ones. Figure 3 shows how close 1/B is to a midpoint between two consecutive floating-point numbers, making this a difficult case for the rounding to nearest mode.

Note that according to Lemma 1, this is the closest A/B, and in particular 1/B can get to a midpoint between two consecutive floating-point numbers.

Double-Extended Precision Iterative Floating-Point Divide Algorithm

Of all the algorithms implemented, one will be presented as an example. The following algorithm receives the double-extended precision operands a and b as inputs, and calculates a/b. All the computational steps are performed in double-extended precision. The algorithm is proven correct, in that the final result q_2 equals $(a/b)_{rnd}$ in any IEEE rounding mode *rnd*.

(1) $y_0 = 1/b \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, $m \in \mathbf{R}$, $m \ge 8.886$

(2) $e_0 = (1 - b \cdot y_0)_{rn}$

(3) $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$

(4) $e_1 = (e_0^2)_{rn}$

(5) $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$

(6) $e_2 = (1 - b \cdot y_2)_{rn}$

(7) $y_3 = (y_2 + e_2 \cdot y_2)_{rn}$

(8) $e_3 = (1 - b \cdot y_3)_{rn}$

(9) $y_4 = (y_3 + e_3 \cdot y_3)_{rn}$

(10) $q_0 = (a \cdot y_0)_{rn}$

(11) $r_0 = (a - b \cdot q_0)_{rn}$

(12) $q_1 = (q_0 + r_0 \cdot y_3)_{rn}$

(13) $r_1 = (a - b \cdot q_1)_{rn}$

(14) $q_2 = (q_1 + r_1 \cdot y_4)_{rnd}$

The first step is a table lookup performed by the instruction that gives an initial approximation y_0 of 1/b, with known relative error determined by m. Steps (2) and (3), (4) and (5), (6) and (7) and (8) and (9) represent four iterations that generate increasingly better approximations of 1/b in y_1, y_2, y_3, and y_4. Applying Theorem 2, it is shown that $y_4 = (1/b)_{rn}$, except when the significand of b is all binary ones. Steps (10) through (14) calculate three increasingly better approximations q_0, q_1, and q_2 of a/b. Evaluating their relative errors and applying Property 2, it is shown that q_1 is within 1 ulp of a/b. This allows application of Theorem 3, which means that $q_2 = (a/b)_{rnd}$

in any IEEE rounding mode *rnd*. The case when the significand of b consists only of binary ones is handled by applying Theorem 4. For this, y_4 is shown first to be within 1 ulp of 1/b. We still have q_1 within 1 ulp of a/b. Starting from the known value of y_0, we prove that $|q_1| \geq (|A/B|)_{rn}$. This allows application of Theorem 4, which proves that $q_2 = (a/b)_{rnd}$ in this special case.

Proofs for other variants of floating-point divide algorithms are derived in a similar manner. Mathematica programs are used in evaluating the relative errors, and the theoretical properties shown above are used to translate the relative error information into ulps, and to prove the IEEE correctness of each algorithm.

The floating-point divide algorithms are the basis for implementing floating-point remainder operations. Their correctness is a direct consequence of the correctness of the corresponding floating-point divide algorithms.

Software Assistance Conditions

The Software Assistance conditions for divide specify cases of input operands for a/b that can cause overflow, underflow, or loss of precision of an intermediate result, even though the final result of the computation calculated with unlimited exponent range is valid. Such cases require alternate algorithms to generate correct results. They are identified by the following conditions, which apply to the cases when the precision of the intermediate computation steps is equal to the precision of the final result:

(a) $e_b \leq e_{min} - 2$ (y_i might become huge)

(b) $e_b \geq e_{max} - 2$ (y_i might become tiny)

(c) $e_a - e_b \geq e_{max}$ (q_i might become huge)

(d) $e_a - e_b \leq e_{min} + 1$ (q_i might become tiny)

(e) $e_a \leq e_{min} + N - 1$ (r_i might lose precision)

where e_a is the (unbiased) exponent of a, e_b is the unbiased exponent of b, and N is the number of bits in the significand.

Translating the conditions above into a two-dimensional representation, Figure 4 is obtained.

The two dimensional space containing pairs (e_a, e_b) in Figure 4 is partitioned by 18 straight lines, determining several polygon shaped regions. Alternate software algorithms have to be devised to compute the IEEE correct quotient for numbers whose exponents fall in regions satisfying any of the five conditions above. Some of these may be coalesced having similar mathematical properties, resulting in only 13 different cases that have to be considered. Points in the different areas requiring Software Assistance are scaled by the alternate algorithms

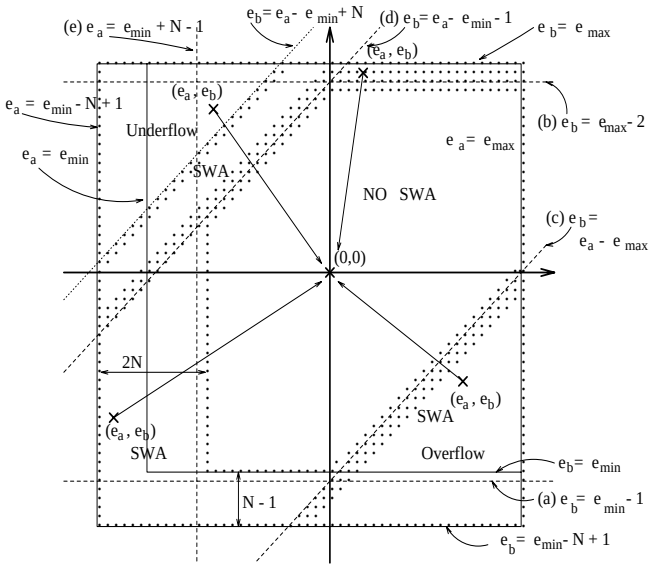

Fig. 4. Software Assistance conditions for the floating-point divide operation

(as shown in the figure by arrows) to the origin (0,0). Figure 4 allows also estimating the frequency of occurrence for pairs (a,b) requiring Software Assistance for divide. This increases for smaller precisions: for double-extended precision floating-point numbers, it is 25%, while for single-precision, the estimated frequency is 31.8%. An obvious way to avoid the negative effects of the need for Software Assistance is to use a wider internal exponent range for calculating the result of the divide operation.

Status Flags Settings

We will assume that the invalid and divide-by-zero status flags are set by the instruction implementing the first step of the algorithm – the reciprocal approximation instruction. The last thing to prove about the iterative algorithms for divide is that the overflow, underflow, and inexact status flags are set correctly in the last step of the computation (we will assume that the intermediate steps do not affect these flags). If the intermediate computation steps use an exponent field larger than that of the final result by at least two bits, then overflow, underflow, or precision loss will not occur, and the last computation step will set correctly the overflow and underflow flags. Otherwise, the overflow, underflow, or precision loss situations will be taken care of by the Software Assistance mechanism (which will also set correctly the status flags).

In order to show that the inexact status flag is set correctly, it is sufficient to consider the last step of the computation. For the double-extended case, this is:

(14) $q_2 = (q_1 + r_1 \cdot y_4)_{rnd}$

If $q_2^* = q_1 + r_1 \cdot y_4$ is the final result before rounding, in order to show that the inexact flag is always set correctly in the last computation step, we needed to prove that

$$a/b \in \mathbf{F}_N \Leftrightarrow q_2^* \in \mathbf{F}_N$$

If $a/b \in \mathbf{F}_N$, it was easily shown that q_1 already represents $(a/b)_{rnd} = a/b \in \mathbf{F}_N$, and $r_1 = 0$, which means that $q_2^* = q_1 \in \mathbf{F}_N$.

If $q_2^* \in \mathbf{F}_N$, we can use the fact that $(q_2^*)_{rnd} = (a/b)_{rnd}$, in any rounding mode *rnd*. This implies that $a/b = q_2^* \in \mathbf{F}_N$, which concludes the proof that the inexact flag is always set correctly.

Floating-Point Square Root Correctness Proof

The floating-point square root algorithms considered are also based on a Newton-Raphson iterative computation. If $\sqrt{a}$ needs to be computed, a number of Newton-Raphson iterations calculate first an approximation of $1/\sqrt{a}$, using the function

$$f(y) = 1/y^2 - a$$

The general iteration step is:

$$e_n = (1/2 - 1/2 \cdot a \cdot y_n^2)_{rn}$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn}$$

where the subscript *rn* denotes the IEEE rounding to nearest mode. The first computation above is rearranged in the real algorithm in order to take advantage of the FMA instruction capability.

Once a sufficiently good approximation y of $1/\sqrt{a}$ is determined, $S = a \cdot y$ approximates $\sqrt{a}$. From case to case, this too might need further refinement.

Two main properties, Theorems 5 and 6 are used in proving the IEEE correctness of the floating-point square root algorithms (but the number of special points singled out in each of them will be different, depending on how tight a relative error we can determine for the final result of the algorithm calculating the square root). They are expressed in terms of a scaled value A of a: if $a \in \mathbf{F}_N$, $a > 0$, $a = \sigma \cdot s \cdot 2^e$, then a can be written as:

$$a = A \cdot 2^{e - 2 \cdot N + 2} \quad \text{if } e = 2 \cdot u,\ u \in \mathbf{Z}, \text{ or}$$

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

where

$$A \in [2^{2 \cdot N - 2}, 2^{2 \cdot N - 1}),\ A \equiv 0 \pmod{2^{N-1}}$$

if $e = 2 \cdot u,\ u \in Z$, and

$$A \in [2^{2 \cdot N - 1}, 2^{2 \cdot N}),\ A \equiv 0 \pmod{2^N}$$

if $e = 2 \cdot u + 1,\ u \in Z$

Theorem 5. 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 above, then for any $f \in \mathbf{F}_N$, and for any integer $F \in [2^{N-1}, 2^N)$:

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

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

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

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

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

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

(Reference [4] provides details on determining A_i and a_i).

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 floating-point number, within which $\sqrt{a}$ cannot exist. The minimum distance between $\sqrt{a}$ and f, or equivalently, between $\sqrt{A}$ and F, was determined by examining instead the distance between A and F^2, as shown in Figure 5 (the identity $|A - F^2| = |\sqrt{A} - F| \cdot |\sqrt{A} + F|$ was used). This property is used in conjunction with that in Theorem 6.

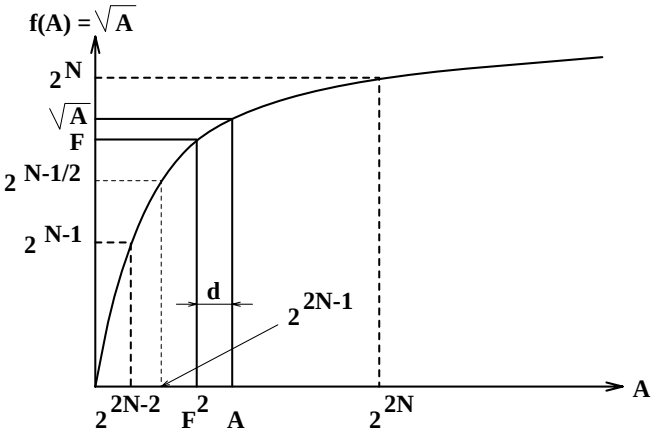

Fig. 5. The distance of A to the square of an integer $F \in [2^{N-1}, 2^N)$: $d \geq 4$, except for two known values, A_1 and A_2, when $d = 1$.

Theorem 6. Let $a \in \mathbf{F}_N$, $a > 0$, $a = \sigma \cdot s \cdot 2^e$. If A is determined by scaling a as specified above, 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) $w_{\sqrt{A}}' = | \sqrt{A} - M | > 17 / 2^{N+3}$

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

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

$A_2' = 2^{2 \cdot N - 2} + 3 \cdot 2^{N-1}$ if $e = 2 \cdot u + 1$, $u \in \mathbf{Z}$

A_3' = … (depends on N) if $e = 2 \cdot u$, $u \in \mathbf{Z}$,

e.g. $A_3' = 7e08a5000000_H$ for N = 24

…

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

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

(corresponding to A_1' through A_7' above):

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

$a_2' = (1 + 3 \cdot 2^{-N+1}) \cdot 2^e$ if $e = 2 \cdot u$, $u \in \mathbf{Z}$,

a_3' = … (depends on N) if $e = 2 \cdot u$, $u \in \mathbf{Z}$,

e.g. $a_3' = 1.f82294_H \cdot 2^e$ for N = 24

… Δ

(Reference [4] provides details on determining A_i' and a_i').

This theorem states that if √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 √a cannot exist. The minimum distance between √a and m, or equivalently, between √A and M, could be determined by examining instead the distance between A and M^2, as shown in Figure 6. Some of the values of a_i' and of the corresponding A_i' (not all shown here) were determined directly, while others were determined by C programs, using recursion.

Note that the widths of the exclusion zones can be increased as desired [4], by eliminating additional points besides those specified in Theorems 5 and 6 (these points are those whose square roots are the closest to floating-point numbers, or to mid-points between consecutive floating-point numbers). For the double-extended precision algorithm presented in the next subsection, the exclusion zones from Theorem 5 had to be increased by a factor of 2, which meant eliminating 9 additional points that are difficult for directed rounding (for a total of 11 special points). The exclusion zones from Theorem 6 had to be increased also by a factor of 32/17, which meant eliminating 6 additional points that are difficult for rounding to nearest (for a total of 13 special points). If no special points are singled out, then the exclusion zones around floating-point numbers in $\mathbf{F}_N$, and around midpoints between consecutive floating-point numbers are specified by the following two properties, derived from Theorems 2 (b) and 3 (b) of [4]:

Theorem C. Let $a \in \mathbf{F}_N$. If $\sqrt{a} \notin \mathbf{F}_N$, then for any $f \in \mathbf{F}_N$, the distance between a/b and f satisfies

$$| \sqrt{a} - f | > 2^{-N-1} \cdot u(\sqrt{a}, N) \qquad \Delta$$

Theorem D. Let $a \in \mathbf{F}_N$. For any $m \in \mathbf{F}_{N+1} - \mathbf{F}_N$ (midpoint between two consecutive floating-point numbers in $\mathbf{F}_N$), the distance between √a and m satisfies

$$| \sqrt{a} - m | > 2^{-N-3} \cdot u(\sqrt{a}, N) \qquad \Delta$$

This property in Theorem 5 is used, as explained below, in conjunction with that in Theorem 6. If the result R* before rounding, of an iterative algorithm for calculating the square root of a, and the exact √a are closer to each other than half of the minimum width of any exclusion zone, then $(R^*)_{rnd} = (\sqrt{a})_{rnd}$ for any IEEE rounding mode *rnd*. See Figure 2(b) above, and Figure 7, which illustrate the idea.

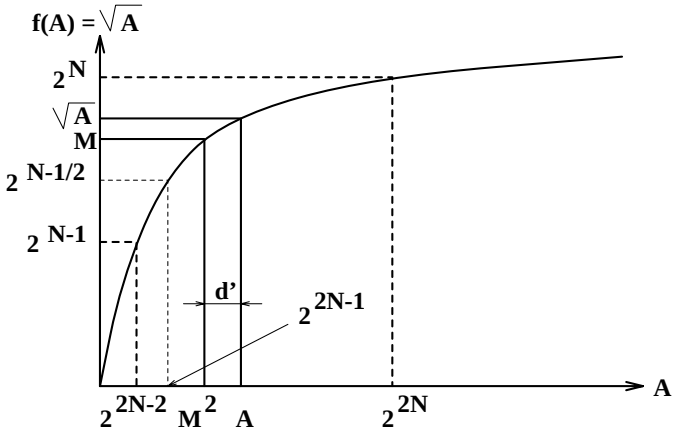

Fig. 6. The distance of A to the square of M, integer + 1/2 ∈ $[2^{N-1}, 2^N)$: d' ≥ 17/4, except for seven known values, A_1' through A_7', when $1/4 \le d' \le 15/4$.

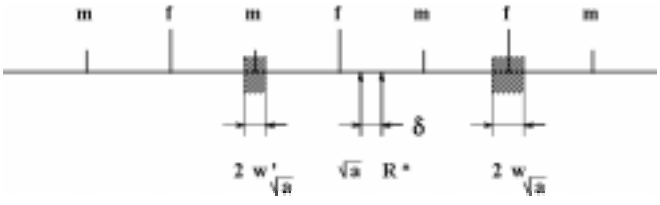

Fig. 7. Exclusion zones around floating-point numbers and around midpoints between consecutive floating-point numbers

Double-Extended Precision Iterative Floating-Point Square Root Algorithm

Just as for divide, of all the algorithms implemented one will be presented as an example. The algorithm that follows receives the value a as a double-extended precision input operand, and calculates $\sqrt{a}$. All the computational steps are performed in double-extended precision. The algorithm is proven correct, in that the final result R equals $(\sqrt{a})_{rnd}$ for any IEEE rounding mode *rnd*.

(1) $y_0 = 1/\sqrt{a} \cdot (1 + \varepsilon_0), |\varepsilon_0| \le 2^{-m}, m \in \mathbf{R}, m \ge 8.831$

(2) $h = (1/2 \cdot a)_{rn}$

(3) $t_1 = (y_0 \cdot y_0)_{rn}$

(4) $t_2 = (1/2 - t_1 \cdot h)_{rn}$

(5) $y_1 = (y_0 + t_2 \cdot y_0)_{rn}$

(6) $t_3 = (y_1 \cdot h)_{rn}$

(7) $t_4 = (1/2 - t_3 \cdot y_1)_{rn}$

(8) $y_2 = (y_1 + t_4 \cdot y_1)_{rn}$

(9) $S = (a \cdot y_2)_{rn}$

(10) $t_5 = (y_2 \cdot h)_{rn}$

(11) $H = (1/2 \cdot y_2)_{rn}$

(12) $d = (a - S \cdot S)_{rn}$

(13) $t_6 = (1/2 - t_5 \cdot y_2)_{rn}$

(14) $S_1 = (S + d \cdot H)_{rn}$

(15) $H_1 = (H + t_6 \cdot H)_{rn}$

(16) $d_1 = (a - S_1 \cdot S_1)_{rn}$

(17) $R = (S_1 + d_1 \cdot H_1)_{rnd}$

The first step is a table lookup performed by the instruction that gives an initial approximation y_0 of $1/\sqrt{a}$, with known relative error determined by m. The following steps implement a Newton-Raphson iterative algorithm. Steps (5) and (8) improve on the approximation of $1/\sqrt{a}$. Steps (9), (14) and (17) calculate increasingly better approximations of $\sqrt{a}$. Mathematical proofs, backed by Mathematica programs, evaluate the relative error of the final result before rounding, R*. The distance between R* and the exact value of $\sqrt{a}$ is shown to be less than half of the minimum width of any exclusion interval given by theorems similar to Theorems 5 and 6 (with wider exclusion zones, as explained). This means that $R = (\sqrt{a})_{rnd}$, for any rounding mode *rnd*, except possibly for the special values identified by the two theorems. C programs and assembly language implementations of the algorithms were used to generate the results for these cases, while Mathematica programs verified their correctness for each of the four IEEE rounding modes. The other variants of floating-point square root algorithms were proven to be IEEE correct in a similar manner.

Software Assistance Conditions

The Software Assistance conditions for square root specify cases of input operands a that can cause underflow or loss of precision of an intermediate result, even though the final result of the computation calculated with unlimited exponent range is valid. Such cases require alternate algorithms to generate correct results, and are identified by the following condition, which applies when the precision of the intermediate computation steps is equal to the precision of the final result:

$$e_a \le e_{min} + N - 1 \quad (d_i \text{ might lose precision})$$

In this condition, e_a is the (unbiased) exponent of a, and N is the number of bits in the significand.

Figure 8 represents the Software Assistance condition on an axis containing the values of the exponent of a.

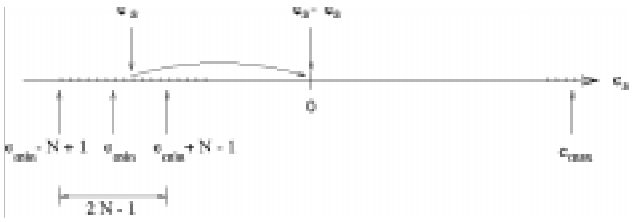

Fig. 8. Software Assistance conditions for the floating-point square root operation

Points requiring Software Assistance are scaled to the origin, as shown by the arrow in the figure. Alternate software algorithms have to be devised to compute the IEEE correct square root for numbers whose exponents fall in the region satisfying the condition above. Figure 8 allows also estimating the frequency of occurrence of argument values requiring Software Assistance for square root. Just as for divide, this increases for smaller precisions. For double-extended precision floating-point numbers, it is of only 0.05%. For single-precision, the estimated frequency becomes 9.6%. In this case too, an obvious way to avoid the negative effects of the need for Software Assistance is to use a wider internal exponent range for calculating the result of the square root operation.

Status Flags Settings

Just as we did for divide, we will assume that the invalid and divide-by-zero status flags are set by the instruction implementing the first step of the algorithm – the

reciprocal square root approximation instruction. All that is left to prove about the iterative algorithm for square root is that the inexact status flag is set correctly in the last step of the computation (we will assume again that the intermediate steps do not affect these flags). We still might have accuracy loss in the intermediate computation steps. If these steps use an exponent field larger than that of the final result by at least one bit, then precision loss will not occur. Otherwise, the precision loss situations will be taken care of by the Software Assistance mechanism (which will also set correctly the status flags).

In order to show that the inexact status flag is set correctly, it is sufficient to consider the last step of the computation. For the double-extended case, this is:

$$(17)\ R = (S_1 + d_1 \cdot H_1)_{rnd}$$

If $R^* = S_1 + d_1 \cdot H_1$ is the final result before rounding, in order to show that the inexact flag is always set correctly in the last computation step, we needed to prove that

$$\sqrt{a} \in \mathbf{F}_N \Leftrightarrow R^* \in \mathbf{F}_N$$

(which is the same idea as for divide).

If $\sqrt{a} \in \mathbf{F}_N$, it was easily shown that S_1 already represents $(\sqrt{a})_{rnd} = \sqrt{a} \in \mathbf{F}_N$, and $d_1 = 0$, which means that $R^* = S_1 \in \mathbf{F}_N$.

If $R^* \in \mathbf{F}_N$, we can use the fact that $(R^*)_{rnd} = (\sqrt{a})_{rnd}$, in any rounding mode *rnd*. This implies that $\sqrt{a} = R^* \in \mathbf{F}_N$, which concludes the proof that the inexact flag is always set correctly.

Results

The main result of the work described in this paper is the proof that the proposed algorithms for divide and square root, in all their variations, are IEEE correct. This also implies correctness for the algorithms implementing the floating-point remainder operations. A byproduct of this study was the determination of conditions leading to the necessity for Software Assistance (SWA). An unlimited exponent range was considered first, and then the effect of the limited exponent range was examined. The SWA conditions are specifying cases when, even though the final result (as calculated with unlimited exponent range) is valid and within the acceptable domain, the result of an intermediate calculation with limited exponent range overflows, underflows, or loses precision, possibly preventing the final result from being correct. Alternate algorithms, though less efficient, are devised for these relatively few cases. These may become part of a floating-point emulation library, and their correctness is also a consequence of the proofs outlined herein. In addition, the current study allowed determination of difficult cases for rounding for both divide (and hence remainder) and square root. Operands that lead to such cases can be systematically determined [4], and can be included in test suites, increasing their probability of success.

Conclusion

The IEEE correctness proofs outlined above are a summary of the proofs described in an internal document [3] that contains lemmas, theorems, and other properties, plus all the Mathematica, C, and assembly language programs used. This constitutes a complete and rigorous proof of correctness for such iterative algorithms. While Theorems 2, 3, and 4 in the proof for the divide algorithm are following properties presented in [2], the work described here is possibly the first to cover all the details following the method from that paper. By contrast, the correctness proof for the square root operation is original [4]. In addition, this study has facilitated determination of SWA conditions for the divide and square root operations based on the Newton-Raphson iterative method, and the determination of difficult cases for rounding.

Proving IEEE correctness for this class of floating-point algorithms is an important step in the implementation of the corresponding floating-point operations. Alternate approaches, careful coding, and thorough testing have to complement the work described in this paper, should such an approach be adopted. Remembering past problems related to floating-point operations makes it clear why such work is important to the scientific community in general.

Acknowledgements

The authors would like to thank John Harrison of Intel Corporation for verifying part of the proofs outlined in the paper, and for his valuable observations.

References

[1] ANSI / IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic , *IEEE 1985*

[2] Markstein P., Computation of Elementary functions on the IBM RISC System/6000 Processor, *IBM Journal*, 1990

[3] Cornea-Hasegan M., Golliver R., Iterative Floating-Point Algorithms Correctness Proofs, *Intel Corporation, 1997*

[4] Cornea-Hasegan M., Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms, Intel Technology Journal, April 1998, *Intel Corporation*

[5] Wolfram S., Mathematica, A System for Doing Mathematics by Computer, *Addison-Wesley Publishing Co., 1993*