A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format

Marius Cornea, *Member*, *IEEE*, John Harrison, *Member*, *IEEE*, Cristina Anderson, Ping Tak Peter Tang, *Member*, *IEEE*, Eric Schneider, and Evgeny Gvozdev

Abstract—The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [19] was revised [20], and an important addition is the definition of decimal floating-point arithmetic [8], [24]. This is intended mainly to provide a robust reliable framework for financial applications that are often subject to legal requirements concerning rounding and precision of the results, because the binary floating-point arithmetic may introduce small but unacceptable errors. Using binary floating-point calculations to emulate decimal calculations in order to correct this issue has led to the existence of numerous proprietary software packages, each with its own characteristics and capabilities. The IEEE 754R decimal arithmetic should unify the ways decimal floating-point calculations are carried out on various platforms. New algorithms and properties are presented in this paper, which are used in a software implementation of the IEEE 754R decimal floating-point arithmetic, with emphasis on using binary operations efficiently. The focus is on rounding techniques for decimal values stored in binary format, but algorithms are outlined for the more important or interesting operations of addition, multiplication, and division, including the case of nonhomogeneous operands, as well as conversions between binary and decimal floating-point formats. Performance results are included for a wider range of operations, showing promise that our approach is viable for applications that require decimal floating-point calculations. This paper extends an earlier publication [6].

Index Terms—Computer arithmetic, multiple-precision arithmetic, floating-point arithmetic, decimal floating-point, computer arithmetic, correct rounding, binary-decimal conversion.

1 Introduction

There is increased interest in decimal floating-point arithmetic in both industry and academia as the IEEE 754R[1] [20] revision becomes the new standard for floating-point arithmetic. The 754R Standard describes two different possibilities for encoding decimal floating-point values: the binary encoding, based on using a Binary Integer [24] to represent the significand (BID, or Binary Integer Decimal), and the decimal encoding, which uses the Densely Packed Decimal (DPD) [7] method to represent groups of up to three decimal digits from the significand as 10-bit *declets*. In this paper, we present results from our work toward a 754R decimal floating-point software implementation based on the BID encoding. We include a discussion of our motivation, selected algorithms, performance results, and future work. The most important or typical operations will be discussed: primarily decimal rounding but also addition, multiplication, division, and conversions between binary and decimal floating-point formats.

1. Since the time of writing of this paper, the IEEE 754R Standard has become the IEEE Standard 754™-2008 for Floating-Point Arithmetic.

- *M. Cornea, J. Harrison, and C. Anderson are with Intel Corp., JF1-13, 2111 NE 25th Avenue, Hillsboro, OR 97124. E-mail: {Marius.Cornea, cristina.s.anderson}@intel.com, johnh@ichips.intel.com.*
- *P.T.P. Tang is with D.E. Shaw Research, L.L.C., 120 West 45th Street, 39th Floor, New York, NY 10036. E-mail: Peter.Tang@DEShawResearch.com.*
- *E. Schneider is with Intel Corp., RA2-406, 2501 NW 229th Avenue, Hillsboro, OR 97124. E-mail: eric.schneider@intel.com.*
- *E. Gvozdev is with Intel Corp., SRV2-G, 2 Parkovaya Street, Satis, Diveevo District, Nizhny Novgorod Region 607328, Russia. E-mail: evgeny.gvozdev@intel.com.*

Manuscript received 23 July 2007; revised 19 June 2008; accepted 10 Sept. 2008; published online 7 Nov. 2008.
Recommended for acceptance by P. Kornerup, P. Montuschi, J.-M. Muller, and E. Schwarz.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-07-0378.
Digital Object Identifier no. 10.1109/TC.2008.209.

1.1 Motivation and Previous Work

An inherent problem of binary floating-point arithmetic used in financial calculations is that most decimal floating-point numbers cannot be represented exactly in binary floating-point formats, and errors that are not acceptable may occur in the course of the computation. Decimal floating-point arithmetic addresses this problem, but a degradation in performance will occur compared to binary floating-point operations implemented in hardware. Despite its performance disadvantage, decimal floating-point arithmetic is required by certain applications that need results identical to those calculated by hand [3]. This is true for currency conversion [13], banking, billing, and other financial applications. Sometimes, these requirements are mandated by law [13]; other times, they are necessary to avoid large accounting discrepancies [18].

Because of the importance of this problem a number of decimal solutions exist, both hardware and software. Software solutions include C# [22], COBOL [21], and XML [25], which provide decimal operations and datatypes. Also, Java and C/C++ both have packages, called BigDecimal [23] and decNumber [9], respectively. Hardware solutions

were more prominent earlier in the computer age with the ENIAC [27] and UNIVAC [16]. However, more recent examples include the CADAC [5], IBM's z900 [4] and z9 [10] architectures, and numerous other proposed hardware implementations [11], [12], [1]. More hardware examples can be found in [8], and a more in-depth discussion is found in Wang's work [26].

The implementation of nonhomogeneous decimal floating-point operations, whose operands and results have mixed formats, is also discussed below. The method used is based on replacing certain nonhomogeneous operations by a similar homogeneous operation and then doing a floating-point conversion to the destination format. Double rounding errors that may occur in such cases are corrected using simple logical equations. Double rounding errors have been discussed in other sources. A case of double rounding error in conversions from binary to decimal (for printing binary values) is presented in [15] (in the proof of Theorem 15; a solution of how to avoid this is given based on status flags and only for this particular case). A paper by Figueroa [14] discusses cases when double rounding errors will not occur but does not offer a solution for cases when they do occur. A paper by Boldo and Melquiond [2] offers a solution for correcting double rounding errors when rounding to nearest but only if a non-IEEE special rounding mode is implemented first, named in the paper as rounding to odd (which is not related to rounding to nearest-even). However, the solution presented here for correcting double rounding errors seems to be new.

Estimations have been made that hardware approaches to decimal floating-point arithmetic will have average speedups of 100-1,000 times over software [18]. However, the results from our implementation show that this is unlikely, as the maximum clock cycle counts for decimal operations implemented in software are in the range of tens or hundreds on a variety of platforms. Hardware implementations would undoubtedly yield a significant speedup but not as dramatic, and that will make a difference only if applications spend a large percentage of their time in decimal floating-point computations.

2 Decimal Rounding

A decimal floating-point number n is encoded using three fields: sign s, exponent e, and significand σ with at most p decimal digits, where p is the precision (p is 7, 16, or 34 in IEEE 754R, but $p = 7$ for the 32-bit format is for storage only). The significand can be scaled up to an integer C, referred to as the *coefficient* (and the exponent is decreased accordingly): $n = (-1)^s \cdot 10^e \cdot \sigma = (-1)^s \cdot 10^{e'} \cdot C$.

The need to round an exact result to the precision p of the destination format occurs frequently for the most common decimal floating-point operations: addition, subtraction, multiplication, fused multiply-add, and several conversion operations. For division and square root, this happens only in certain corner cases. If the decimal floating-point operands are encoded using the IEEE 754R binary format, the rounding operation can be reduced to the rounding of an integer binary value C to p decimal digits. Performing this operation efficiently on decimal numbers stored in binary format is very important, as it enables good software implementations of decimal floating-point arithmetic on machines with binary hardware. For example, assume that the exact result of a decimal floating-point operation has a coefficient $C = 1234567890123456789$ with $q = 19$ decimal digits that is too large to fit in the destination format and needs to be rounded to the destination precision of $p = 16$ digits.

As mentioned, C is available in binary format. To round C to 16 decimal digits, one has to remove the lower $x = 3$ decimal digits $(x = q - p = 19 - 16)$ and possibly to add one unit to the next decimal place, depending on the rounding mode and on the value of the quantity that has been removed. If C is rounded to nearest, the result will be $C = 1234567890123457 \cdot 10^3$. If C is rounded toward zero, the result will be $C = 1234567890123456 \cdot 10^3$.

The straightforward method to carry out this operation is to divide C by 1,000, to calculate and subtract the remainder, and possibly to add 1,000 to the result at the end, depending on the rounding mode and on the value of the remainder.

A better method is to multiply C by 10^{-3} and to truncate the result so as to obtain the value 1234567890123456. However, negative powers of 10 cannot be represented exactly in binary, so an approximation will have to be used. Let $k_3 \approx 10^{-3}$ be an approximation of 10^{-3}, and thus, $C \cdot k_3 \approx 1234567890123456$. If k_3 overestimates the value of 10^{-3}, then $C \cdot k_3 > 1234567890123456$. Actually, if k_3 is calculated with sufficient accuracy, it will be shown that $\lfloor C \cdot k_3 \rfloor = 1234567890123456$ with certainty.

(Note that the $floor(x)$, $ceiling(x)$, and $fraction(x)$ functions are denoted here by $\lfloor x \rfloor$, $\lceil x \rceil$, and $\{x\}$, respectively.)

Let us separate the rounding operation of a value C to fewer decimal digits into two steps: first calculate the result rounded to zero and then apply a correction if needed, by adding one unit to its least significant digit. Rounding overflow (when after applying the correction, the result requires $p + 1$ decimal digits) is not considered here.

The *first step* in which the result rounded toward zero is calculated can be carried out by any of the following four methods.

Method 1. *Calculate $k_3 \approx 10^{-3}$ as a y-bit approximation of 10^{-3} rounded up (where y will be determined later). Then, $\lfloor C \cdot k_3 \rfloor = 1234567890123456$, which is exactly the same value as $\lfloor C/10^3 \rfloor$.*

It can be noticed, however, that $10^{-3} = 5^{-3} \cdot 2^{-3}$ and that multiplication by 2^{-3} is simply a shift to the right by 3 bits. Three more methods to calculate the result rounded toward zero are then possible:

Method 1a. *Calculate $h_3 \approx 5^{-3}$ as a y-bit approximation of 5^{-3} rounded up (where y will be determined later). Then, $\lfloor (C \cdot h_3) \cdot 2^{-3} \rfloor = 1234567890123456$, which is the same as $\lfloor C/10^3 \rfloor$. However, this method is no different from Method 1, so the approximation of 5^{-3} has to be identical in the number of significant bits to the approximation of 10^{-3} from Method 1 (but it will be scaled up by a factor of 2^3).*

The third method is obtained if instead of multiplying by 2^{-3} at the end, C is shifted to the right and truncated before multiplication by h_3.

Method 2. *Calculate $h_3 \approx 5^{-3}$ as a y-bit approximation of 5^{-3} rounded up (where y will be determined later). Then, $\lfloor \lfloor C \cdot 2^{-3} \rfloor \cdot h_3 \rfloor = 1234567890123456$, which is identical to $\lfloor C/10^3 \rfloor$. It will be shown that h_3 can be calculated to fewer bits than k_3 from Method 1.*

The final and fourth method is obtained by multiplying C by h_3 and then truncating, followed by a multiplication by 2^{-3} and a second truncation.

Method 2a. *Calculate $h_3 \approx 5^{-3}$ as a y-bit approximation of 5^{-3} rounded up (where y can be determined later). Then, $\lfloor \lfloor C \cdot h_3 \rfloor \cdot 2^{-3} \rfloor = 1234567890123456$. However, it can be shown that in this last case, h_3 has to be calculated to the same number of bits as k_3 in Method 1, which does not represent an improvement over Method 1.*

Only Method 1 and Method 2 shall be examined next.

We will solve next the problem of rounding correctly a number with q digits to $p = q - x$ digits, using approximations to negative powers of 10 or 5. We will also solve the problem of determining the shortest such approximations, i.e., with the least number of bits.

2.1 Method 1 for Rounding a Decimal Coefficient

For rounding a decimal coefficient represented in binary using Method 1, the following property specifies the minimum accuracy y for the approximation $k_x \approx 10^{-x}$, which ensures that a value C with q decimal digits represented in binary can be truncated without error to $p = q - x$ digits by multiplying C by k_x and truncating. Property 1 states that if $y \geq \lceil \{log_2 10 \cdot x\} + log_2 10 \cdot q \rceil$ and k_x is a y-bit approximation of 10^{-x} rounded up, then we can calculate $\lfloor C \cdot k_x \rfloor$ in the binary domain, and we will obtain the same result as for $\lfloor C/10^x \rfloor$ calculated in decimal.

Property 1. *Let $q \in \mathbb{N}$, $q > 0$, $C \in \mathbb{N}$, $10^{q-1} \leq C \leq 10^q - 1$, $x \in \{1, 2, 3, \ldots, q-1\}$, and $\rho = log_2 10$. If $y \in \mathbb{N}$ satisfies $y \geq \lceil \{\rho \cdot x\} + \rho \cdot q \rceil$ and k_x is a y-bit approximation of 10^{-x} rounded up (the subscript RP, y indicates rounding up to y bits), i.e., $k_x = (10^{-x})_{RP,y} = 10^{-x} \cdot (1 + \varepsilon)$, where $0 < \varepsilon < 2^{-y+1}$, then $\lfloor C \cdot k_x \rfloor = \lfloor C/10^x \rfloor$.*

Proof outline. The proof can be carried out starting with the representation of C in decimal, $C = d_0 \cdot 10^{q-1} + d_1 \cdot 10^{q-2} + \cdots + d_{q-2} \cdot 10^1 + d_{q-1}$, where $d_0, d_1, \ldots d_{q-1} \in \{0, 1, \ldots, 9\}$, and $d_0 \neq 0$. The value of C can be expressed as $C = 10^x \cdot H + L$, where $H = \lfloor C/10^x \rfloor = d_0 \cdot 10^{q-x-1} + d_1 \cdot 10^{q-x-2} + d_2 \cdot 10^{q-x-3} + \ldots + d_{q-x-2} \cdot 10^1 + d_{q-x-1} \in [10^{q-x-1}, 10^{q-x} - 1]$, and $L = C \text{ percent } 10^x = d_{q-x} \cdot 10^{x-1} + d_{q-x-1} \cdot 10^{x-2} + \ldots + d_{q-2} \cdot 10^1 + d_{q-1} \in [0, 10^x - 1]$. Then, the minimum value of y can be determined such that $\lfloor C \cdot k_x \rfloor = H \Leftrightarrow H \leq (H + 10^{-x} \cdot L) \cdot (1 + \varepsilon) < H + 1 \Leftrightarrow 10^{-x} \cdot \varepsilon < 10^{-2x}/(H + 1 - 10^{-x})$.

But $10^{-x} \cdot \varepsilon$ is the absolute error in $k_x = 10^{-x} \cdot (1 + \varepsilon)$, and it is less than 1 unit in the last place (ulp) of k_x. It can be shown that for all x of interest in the context of the IEEE 754R floating-point formats, $x \in \{1, 2, 3, \ldots, 34\}$, k_x is in the same binade as 10^{-x}, because one cannot find a power of two to separate 10^{-x} and $k_x = 10^{-x} \cdot (1 + \varepsilon)$ (which would place the two values in different binades) with k_x represented with a reasonable number of bits y. This can be checked exhaustively for $x \in \{1, 2, 3, \ldots, 34\}$. For this, assume that such a power of two exists: $10^{-x} < 2^{-s} \leq 10^{-x} \cdot (1 + \varepsilon) \Leftrightarrow \frac{10^x}{(1+\varepsilon)} \leq 2^s < 10^x \Leftrightarrow \frac{2^{\rho \cdot x}}{(1+\varepsilon)} \leq 2^s < 2^{\rho \cdot x}$.

Even for those values of x where $\rho \cdot x$ is slightly larger than an integer ($x = 22$, $x = 25$, and $x = 28$), ε would have to be too large in order to satisfy $\frac{2^{\rho \cdot x}}{(1+\varepsilon)} \leq 2^s$. The conclusion is that k_x is in the same binade as 10^{-x}, which will let us determine $\text{ulp}(k_x) = 2^{-\lfloor \rho \cdot x \rfloor - y}$. In order to satisfy inequality (1), it is sufficient to have the following (increasing the left-hand side and decreasing the right-hand side of the inequality): $ulp(k_x) \leq \frac{10^{-2x}}{(10^{q-x} - 10^{-x})}$. This leads eventually to $y \geq \lceil \{\rho \cdot x\} + \rho \cdot q \rceil$, as required. □

In our example from Method 1 above, $y = \lceil \{\rho \cdot 3\} + \rho \cdot 19 \rceil = 65$ (so 10^{-3} needs to be rounded up to 65 bits).

The values k_x for all x of interest are precalculated and are stored as pairs (K_x, e_x), with K_x and e_x positive integers, and $k_x = K_x \cdot 2^{-e_x}$. This allows for efficient implementations, exclusively in the integer domain, of several decimal floating-point operations, particularly addition, subtraction, multiplication, fused multiply-add, and certain conversions.

The *second step* in rounding correctly a value C to fewer decimal digits consists of applying a correction to the value rounded to zero, if necessary. The inexact status flag has to be set correctly as well. For this, we have to know whether the original value that we rounded was exact (did it have x trailing zeros in decimal?) or if it was a midpoint when rounding to nearest (were its x least significant decimal digits a 5 followed by $x - 1$ zeros?).

Once we have the result truncated to zero, the straightforward method to check for exactness and for midpoints could be to calculate the remainder $C \text{ percent } 10^x$ and to compare it with 0 and with $50 \ldots 0$ (x decimal digits), which is very costly. However, this is not necessary because the information about exact values and midpoints is contained in the fractional part f of the product $C \cdot k_x$.

The following property states that if $C \cdot 10^{-x}$ covers an interval [H, H + 1) of length 1, where H is an integer, then $C \cdot k_x$ belongs to [H, H + 1) as well, and the fraction f discarded by the truncation operation belongs to (0, 1) (so f cannot be zero). More importantly, we can identify the exact cases $C \cdot 10^{-x} = H$ (then, rounding C to $p = q - x$ digits is an exact operation) and the midpoint cases $C \cdot 10^{-x} = H + \frac{1}{2}$.

Property 2. *Let $q \in \mathbb{N}$, $q > 0$, $x \in \{1, 2, 3, \ldots, q-1\}$, $C \in \mathbb{N}$, $10^{q-1} \leq C \leq 10^q - 1$, $C = 10^x \cdot H + L$, where $H \in [10^{q-x-1}, 10^{q-x} - 1]$, $L \in [0, 10^x - 1]$, and H, $L \in \mathbb{N}$, $f = C \cdot k_x - \lfloor C \cdot k_x \rfloor$, $\rho = log_2 10$, $y \in \mathbb{N}$, $y \geq 1 + \lceil \rho \cdot q \rceil$, and $k_x = 10^{-x} \cdot (1 + \varepsilon)$, $0 < \varepsilon < 2^{-y+1}$. Then, the following are true:*

a. *$C = H \cdot 10^x$ iff $0 < f < 10^{-x}$.*
b. *$H \cdot 10^x < C < (H + \frac{1}{2}) \cdot 10^x$ iff $10^{-x} < f < \frac{1}{2}$.*
c. *$C = (H + \frac{1}{2}) \cdot 10^x$ iff $\frac{1}{2} < f < \frac{1}{2} + 10^{-x}$.*
d. *$(H + \frac{1}{2}) \cdot 10^x < C < (H + 1) \cdot 10^x$ iff $\frac{1}{2} + 10^{-x} < f < 1$.*

(The proof is not included but is straightforward to obtain.) This property is useful for determining the exact and the midpoint cases. For example if $0 < f < 10^{-x}$, then we can be sure that the result of the rounding from q decimal digits to $p = q - x$ digits is exact.

Another important result of Property 2 was that it helped refine the results of Property 1. Although the accuracy y of k_x determined in Property 1 is very good, it is not the best possible. For a given pair q and $x = q - p$, starting with the value $y = \lceil \{\rho \cdot x\} + \rho \cdot q \rceil$, one can try to reduce it 1 bit at a time, while checking that the corresponding k_x will still yield the correct results for rounding in all cases, and it will also allow for exactness and midpoint detection, as shown above. To verify that a new (and smaller) value of y still works, just four inequalities have to be verified for boundary conditions related to exact cases and midpoints: $H \cdot 10^x \cdot k_x < H + 10^{-x}$, $(H + 1/2 - 10^{-x}) \cdot 10^x \cdot k_x < H + 1/2$, $(H + 1/2) \cdot 10^x \cdot k_x < H + 1/2 + 10^{-x}$, and $(H + 1 - 10^{-x}) \cdot 10^x \cdot k_x < H + 1$. Because the functions of H from these inequalities can be reduced to monotonic and increasing ones, it is sufficient to verify the inequalities just for the maximum value $H = 10^{q-x} - 1 = 99 \ldots 9$ ($q - x$ decimal digits). For example, this method applied to the constant k_3 used to illustrate Method 1 (for truncation of a 19-digit number to 16 digits) allowed for a reduction of the number of bits from $y = 65$ to $y = 62$ (importantly, because k_3 fits now in a 64-bit integer).

2.2 Method 2 for Rounding a Decimal Coefficient

This method has the advantage that the number of bits in the approximation of 5^{-x} is less by x than that for 10^{-x} in Method 1, as stated in the following property.

Property 3. *Let $q \in \mathbb{N}$, $q > 0$, $C \in \mathbb{N}$, $10^{q-1} \le C \le 10^q - 1$, $x \in \{1, 2, 3, \ldots, q-1\}$, and $\rho = log_2 10$. If $y \in \mathbb{N}$, $y \ge \lceil \{\rho \cdot x\} + \rho \cdot q \rceil - x$, and h_x is a y-bit approximation of 5^{-x} rounded up, i.e., $h_x = (5^{-x})_{RP,y} = 5^{-x} \cdot (1 + \varepsilon)$, $0 < \varepsilon < 2^{-y+1}$, then $\lfloor \lfloor C \cdot 2^{-x} \rfloor \cdot h_x \rfloor = \lfloor C / 10^x \rfloor$.*

However, determining exact cases and midpoints is slightly more complicated than with Method 1.

Two fractions are removed by truncation: the first in $\lfloor C \cdot 2^{-x} \rfloor$ and the second in $\lfloor \lfloor C \cdot 2^{-x} \rfloor \cdot h_x \rfloor$. To determine whether the rounding was exact, test first whether the lower x bits of C are zero (if they are not, C could not have been divisible by 10^x). If C was divisible by 2^x, we just need to test $\lfloor C \cdot 2^{-x} \rfloor$ for divisibility by 5^x. This is done by examining the fraction removed by truncation in $\lfloor \lfloor C \cdot 2^{-x} \rfloor \cdot h_x \rfloor$, using a property similar to Property 2 from Method 1. Actually, exact and midpoint cases can be determined together if the first shift is by $x - 1$ bits only (and not by x bits). If C had the x lower bits equal to zero, the rounding might be exact. If it had only $x - 1$ bits equal to zero, the rounding might be for a midpoint. In both cases, the answer is positive if the fraction f removed by the last truncation in $\lfloor \lfloor C \cdot 2^{-x+1} \rfloor \cdot h_x \rfloor$ satisfies conditions similar to those from Property 2. The least significant bit in $\lfloor \lfloor C \cdot 2^{-x+1} \rfloor \cdot h_x \rfloor$ and the values of the fractions removed in the two truncations will tell whether the rounding was exact, for a midpoint, or for a value to the left or to the right of a midpoint.

3 Addition and Multiplication

Addition and multiplication were implemented using straightforward algorithms, with the rounding step carried out as explained in the previous section. The following pseudocode fragment describes the algorithm for adding two decimal floating-point numbers encoded using the binary format, $n1 = C1 \cdot 10^{e1}$ and $n2 = C2 \cdot 10^{e2}$, whose significands can be represented with p decimal digits ($C1$, $C2 \in \mathbb{Z}$, $0 < C1 < 10^p$, and $0 < C2 < 10^p$). For simplicity, it is assumed that $n1 \ge 0$ and $e1 \ge e2$ and that the rounding mode is set to rounding to nearest: $n = (n1 + n2)_{RN,p} = C \cdot 10^e$. In order to make rounding easier when removing x decimal digits from the lower part of the exact sum, $1/2 \cdot 10^x$ is added to it, and rounding to nearest is reduced to truncation except possibly when the exact sum is a midpoint. The notation $digits(X)$ is used for the minimum number of decimal digits needed to represent the integer value X.

$q1 = digits(C1)$, $q2 = digits(C2)$ // table lookup
if $|q1 + e1 - q2 - e2| \ge p$
  $n = C1 \cdot 10^{e1}$ or $n = C1 \cdot 10^{e1} \pm 10^{e1+q1-p}$ (inexact)
else // if $|q1 + e1 - q2 - e2| \le p - 1$
  $C' = C1 \cdot 10^{e1-e2} + C2$ // binary integer
  $q = digits(C')$ // table lookup
  if $q \le p$
    return $n = C' \cdot 10^{e2}$ (exact)
  else // if $q \in [p + 1, 2 \cdot p]$
    continue
  $x = q - p$, decimal digits to be removed from lower part of C', $x \in [1, p]$
  $C'' = C' + 1/2 \cdot 10^x$
  $k_x = 10^{-x} \cdot (1 + \epsilon)$, $0 < \epsilon < 2^{-\lceil 2 \cdot \rho \cdot p \rceil}$
  $C^* = C'' \cdot k_x = C'' \cdot K_x \cdot 2^{-E_x}$
  f^* = the fractional part of C^* // lower E_x bits of product $C'' \cdot K_x$
  if $0 < f^* < 10^{-p}$
    if $\lfloor C^* \rfloor$ is even
      $C = \lfloor C^* \rfloor$
    else
      $C = \lfloor C^* \rfloor - 1$
  else
    $C = \lfloor C^* \rfloor$
  $n = C \cdot 10^{e2+x}$
  if $C = 10^p$
    $n = 10^{p-1} \cdot 10^{e2+x+1}$ // rounding overflow
  if $0 < f^* - 1/2 < 10^{-p}$
    the result is exact
  else
    the result is inexact
endif

For multiplication, the algorithm to calculate $n = (n1 \cdot n2)_{RN,p} = C \cdot 10^e$ for rounding to nearest is very similar. There are only two differences with respect to addition: the multiplication algorithm begins with

$C' = C1 \cdot C2$ // binary integer
$q = digits(C')$ // table lookup

and at the end, before checking for rounding overflow and inexactness, the final result is $n = C \cdot 10^{e1+e2+x}$ instead of $n = C \cdot 10^{e2+x}$.

The most interesting aspects are related to the rounding step. For addition and multiplication, the length of the exact result is at most $2 \cdot p$ decimal digits (if this length is larger

for addition, then the smaller operand is just a rounding error compared to the larger one, and rounding is trivial). This also means that the number x of decimal digits to be removed from a result of length $q \in [p+1, 2 \cdot p]$ is between 1 and p. It is not difficult to prove that the comparisons for midpoint and exact case detection can use the constant 10^{-p} instead of 10^{-x}, thus saving a table read.

Note that 10^{-p} (or 10^{-x}) cannot be represented exactly in binary format, so approximations of these values have to be used. It is sufficient to compare f^* or $f^* - \frac{1}{2}$ (both have a finite number of bits when represented in binary) with a truncation t^* of 10^{-p} whose unit in the last place is no larger than that of f^* or $f^* - \frac{1}{2}$. The values t^* are calculated such that they always align perfectly with the lower bits of the fraction f^*, which makes the tests for midpoints and exactness relatively simple in practice.

The IEEE status flags are set correctly in all rounding modes. The results in rounding modes other than to nearest are obtained by applying a correction (if needed) to the result rounded to nearest, using information on whether the precise result was an exact result in the IEEE sense, a midpoint less than an even floating-point number, a midpoint greater than an even number, an inexact result less than a midpoint, or an inexact result greater than a midpoint.

4 Division and Square Root

The sign and exponent fields are easily computed for these two operations. The approach used for calculating the correctly rounded significand of the result was to scale the significands of the operands to the integer domain and to use a combination of integer and floating-point operations.

The division algorithm is summarized below, where p is the maximum digit size of the decimal coefficient, as specified by the following format: $p = 16$ for a 64-bit decimal and $p = 34$ for a 128-bit decimal format. $EMIN$ is the minimum decimal exponent allowed by the format. Overflow situations are not explicitly treated in the algorithm description below; they can be handled in the return sequence, when the result is encoded in BID format.

if $C1 < C2$
  $nd = digits(C2) - digits(C1)$ // table lookup
  $C1' = C1 \cdot 10^{nd}$
  $scale = p - 1$
  if$(C1' < C2)$
    $scale = scale + 1$
  endif
  $C1^* = C1' \cdot 10^{scale}$
  $Q0 = 0$
  $e = e1 - e2 - scale - nd$ // expected exponent
else
  $Q0 = \lfloor C1/C2 \rfloor$, $R = C1 - Q \cdot C2$ // long integer divide and remainder
  if $(R == 0)$
    return $Q \cdot 10^{e1-e2}$ // result is exact
  endif
  $scale = p - digits(Q)$
  $C1^* = R \cdot 10^{scale}$
  $Q0 = Q0 \cdot 10^{scale}$
  $e = e1 - e2 - scale$ // expected exponent
endif
$Q1 = \lfloor C1^*/C2 \rfloor$, $R = C1^* - Q1 \cdot C2$ // long integer divide and remainder
$Q = Q0 + Q1$
if $(R == 0)$
  eliminate trailing zeros from Q:
  find largest integer d s.t. $Q/10^d$ is exact
  $Q = Q/10^d$
  $e = e + d$ // adjust expected exponent
  if $(e \geq EMIN)$
    return $Q \cdot 10^e$
endif
if $(e \geq EMIN)$
  round $Q \cdot 10^e$ according to current rounding mode
  // rounding to nearest based on comparing $C2$ and $2 \cdot R$
else
  compute correct result based on Property 1
  // underflow
endif

The square root algorithm (not presented here) is somewhat similar, in the sense that it requires computing an integer square root of $2 \cdot p$ digits (while division required an integer division of $2 \cdot p$ by p digits). Rounding of the square root result is based on a test involving long integer multiplication.

5 Conversions between Binary and Decimal Formats

Conversions between IEEE 754 binary formats and the new decimal formats may sometimes be needed in user applications, and the revised standard specifies that these should be correctly rounded. We will first consider the theoretical analysis determining the accuracy required in intermediate calculations and then describe more details of the algorithms for some typical source and target formats.

5.1 Required Accuracy

We analyze here how close, in relative terms, a number x in one format can be to a rounding boundary y (either a floating-point number or the midpoint between two floating-point numbers) in another. If we can establish a lower bound ϵ for all cases such that either $x = y$ or $|x/y - 1| > \epsilon$, then we will be able to make a rounding decision based on an approximation good to a relative error of order ϵ, as spelled out in more detail below.

Consider converting binary floating-point numbers to a decimal format; essentially, similar reasoning works the other way round. We are interested in how close, in relative terms, the answer can be to a decimal floating-point number (for directed rounding modes) or a midpoint between them (for rounding to nearest), i.e., either of

$$\left| \frac{2^e m}{10^d n} - 1 \right| \quad \textbf{or} \quad \left| \frac{2^e m}{10^d (n + 1/2)} - 1 \right|,$$

where integers m and n are constrained by the precision of the formats, and d and e are constrained by their exponent ranges. We can encompass both together by considering

problems of the form $|\frac{2^{e-1}m}{10^d n} - 1|$, where n is bounded by twice the usual bound on the decimal coefficient.

5.1.1 Naive Analysis

Based on the most straightforward analytical reasoning, we can deduce that either the two numbers are exactly equal or else

$$\left|\frac{2^e m}{10^d n} - 1\right| = \frac{|2^e m - 10^d n|}{|10^d n|} \geq \frac{1}{|10^d n|}$$

because all quantities are integers. However, this bound is somewhat discouraging, because the exponent d may be very large, e.g., $d = 6{,}111$ for Decimal128.

On the other hand, it seems plausible on statistical grounds that the actual worst case is much less bad. For example, imagine distributing N input numbers (say, $N = 2^{64}$ for the two 64-bit formats) randomly over a format with M possible significands. The fractional part of the resulting significand will probably be distributed more or less randomly in the range $0 \leq x < 1$, so the probability of its being closer than δ to a floating-point number will be about δ. Therefore, the probability that *none* of them will be closer than δ is about $(1-\delta)^N = (1-\delta)^{\frac{1}{\delta}N\delta} \approx e^{-N\delta}$. A *relative* difference of ϵ typically corresponds to an absolute error in the integer significand of about $\delta = \epsilon M$, and so, the probability of relative closeness ϵ is about $1 - e^{-MN\epsilon}$. Thus, we expect the minimum to be of order $\epsilon \approx 1/(MN)$.

5.1.2 Diophantine Approximation Analysis

The above argument is really just statistical hand-waving and may fail for particular cases. However, we can establish such bounds rigorously by rearranging our problem. We are interested in

$$\left|\frac{2^e m}{10^d n} - 1\right| = \left|\frac{2^e/10^d - n/m}{n/m}\right|.$$

We can therefore consider all the permissible pairs of exponent values e and d, finding the n/m with numerator and denominator within appropriate bounds that gives the best (unequal) approximation. In fact, if we expand the exponent range a little, it is easy to see that we can assume both m and n to be normalized. This gives strong correlations between the exponents, meaning that we only have to examine a few possible e for each d and vice versa.

For each e and d, we can find lower bounds on this quantity subject to the size constraints on m and n, using a variant of the usual algorithms in Diophantine approximation based on computing a sequence of convergents using mediants or continued fractions. (For those interested, we spell out the exact Diophantine approximation algorithm that we use in the Appendix.) As expected from the naive statistical argument, the bounds determined are much better, typically a few bits extra beyond the sum of the precisions of the input and output formats. For example, the most difficult case for converting from Binary64 to Decimal64 is the following, with a relative distance of $2^{-115.53}$:

$$5789867926332032 \cdot 2^{479} \approx 9037255902774040\frac{1}{2} \cdot 10^{144},$$

and the most difficult for conversion from Decimal64 to Binary64 is a relative difference of $2^{-114.62}$ for

$$3743626360493413 \cdot 10^{-165} \approx 6898586531774200\frac{1}{2} \cdot 2^{-549}.$$

5.2 Implementations

We will now show the general pattern of the actual algorithms for conversions in both directions; in fact, both directions have a lot of stylistic similarities. Generally, we consider the two problems "parametrically," using a general p for the precision of the binary format and d for the number of decimal digits in the decimal format. However, some structural differences arise depending on the parameters, two of which we mention now.

First, for conversions between certain pairs of formats (e.g., from any binary format considered into the Decimal128 format), no overflow or underflow can occur, so certain tests or special methods noted below can be omitted and are not present in the actual implementations.

Second, many of the algorithms involve scaled prestored approximations to values such as $10^e/2^f$. In the presentation below, we assume that such values are stored for all possible input exponents (the output exponent is determined consequentially). For several of our implementations, this is indeed the case; doing so ensures high speed and allows us to build appropriate underflow of the output into the tables without requiring special program logic. However, for some conversions, in particular those into Decimal128, the storage requirement seems excessive, so the necessary values are instead derived internally from a bipartite table.

5.2.1 Decimal-Binary Conversion Algorithm

Consider converting from a decimal format with d decimal digits to a binary format with precision p. The basic steps of the algorithm are the following:

- Handle NaN, infinity and zero, and extremely large and small inputs.
- Strip off the sign, so the input is now $10^e m$, where $0 < m < 10^d$.
- "Normalize" the input coefficient to the range $10^d/2 \leq m < 10^d$ by multiplying by a suitable 2^k.
- Pick provisional binary exponent f based on m and e.
- Form scaled product $\frac{10^e}{2^f} m$ to approximate the binary significand.
- Round depending on the sign and rounding mode to give binary significand n. If $n = 2^p$, set $n = 2^{p-1}$ and add 1 to the provisional exponent.
- Subtract k from the provisional exponent, check for overflow, underflow, and inexactness, and combine with the sign.

We now consider some of the central steps in more detail, starting with the selection of the provisional binary exponent. If we set $f = \lceil (e+d)\log_2 10 \rceil - p$, then we have $2^{f+p-1} < 10^{e+d} \leq 2^{f+p}$. Therefore, for any floating-point number $10^e m$ with exponent e and significand $10^d/2 \leq m < 10^d$, we have

$$2^{f-1}2^{p-1} < 10^e m < 2^f 2^p.$$

That is, x lies strictly between the smallest normalized binary floating-point number with exponent $f-1$ and the next beyond the largest with exponent f. We therefore have only two possible choices for the "provisional exponent" f. (We say provisional because rounding may still bump it up to the next binade, but this is easy to check at the end.) And that is without any knowledge of m, except the assumption that it is normalized. Moreover, we can pick the exact m_{min} that acts as the breakpoint between these possibilities:

$$m_{min} = \lceil 2^{f+p-1}/10^e \rceil - 1,$$

so $10^e m_{min} < 2^{f+p-1} \leq 10^e(m_{min}+1)$. Thus, we can pick the provisional exponent to be $f-1$ for all $m \leq m_{min}$ and f for all $m > m_{min}$, and we are sure that $2^f 2^{p-1} \leq x < 2^f 2^p$. In the future, we will just use f for this correctly chosen provisional exponent.

Now, we aim to compute n with $2^f n \approx 10^e m$, i.e., the left-hand side should be a correctly rounded approximation of the right-hand side. Staying with the provisional exponent f, this entails finding a correctly rounded integer $n \approx \frac{10^e}{2^f} m$. We do this by multiplying by an approximation to $10^e/2^f$ scaled by 2^q for some suitably chosen q and rounded up, i.e.,

$$r = \lceil 2^q 10^e / 2^f \rceil,$$

and truncating by shifting to the right by q bits. How do we choose q? From the results above, given the input and output formats, we know an $\epsilon > 0$ such that if $n \neq \frac{10^e}{2^f} m$ or $n + \frac{1}{2} \neq \frac{10^e}{2^f}$, then their relative difference is at least ϵ. We have

$$\frac{2^q 10^e}{2^f} \leq r < \frac{2^q 10^e}{2^f} + 1,$$

and therefore (since $m < 10^d$)

$$\frac{10^e}{2^f} m \leq \frac{mr}{2^q} < \frac{10^e}{2^f} m + \frac{10^d}{2^q}.$$

We want to make a rounding decision for $\frac{10^e}{2^f} m$ based on $\frac{mr}{2^q}$. First, since the latter is an overestimate, it is immediate that if the true result is $\geq$ a rounding boundary, so is the estimate. So we just need to check that if the true answer is $<$ a rounding boundary, so is the estimate. We know that if the true answer is $<$ a rounding boundary, then so is $\frac{10^e}{2^f} m(1+\epsilon)$, so it suffices to show that

$$\frac{10^e}{2^f} m + \frac{10^d}{2^q} \leq \frac{10^e}{2^f} m(1+\epsilon),$$

i.e., that $\frac{10^d}{2^q} \leq \frac{10^e}{2^f} m\epsilon$. By the choice of f made, we have $2^f 2^{p-1} \leq 10^e m$, so it suffices to establish that $\frac{10^d}{2^q} \leq 2^{p-1}\epsilon$.

We also want to be able to test whether a conversion is exact. If it is, then the fractional part of $\frac{mr}{2^q}$ is between 0 and $10^d/2^q$ by the bound noted above. If it is not, then the fractional part is at least $2^{p-1}\epsilon - 10^d/2^q$. To discriminate cleanly, we want this to be at least $10^d/2^q$, so we actually want to pick a q such that $\frac{10^d}{2^q} \leq 2^{p-2}\epsilon$ or, in other words, $2^q \geq \frac{10^d}{2^{p-2}\epsilon}$.

For example, to convert from Decimal64 to Binary64, we need $2^q \geq \frac{10^{16}}{2^{51}.2^{-114.62}}$ (based on the above best approximation results), i.e., $q \geq 117$. To convert from Decimal32 to Binary32, we need $2^q \geq \frac{10^7}{2^{22}2^{-54.81}}$, i.e., $q \geq 57$. In fact, we almost always make q slightly larger than necessary so that it is a multiple of 32 or 64. This makes the shift by q more efficient since it just amounts to picking words of a multiword result.

When q is chosen satisfying these constraints, we just need to consider the product mr to make our rounding decisions. The product with the lowest q bits removed is the correctly truncated significand. Bit $q-1$ is the rounding bit, and the bottom $q-1$ bits (0 to $q-2$) can be considered as a sticky bit: test if it is at least $10^d/2^q$. Using this information, we can correctly round in all rounding modes. Where the input is smaller in magnitude than the smallest possible number in the output format, we artificially limit the effective shift so that the simple interpretation of "sticky" bits does not need changing.

The mechanics of rounding uses a table of "boundaries" against which the lowest q bits of the reciprocal product (i.e., the "round" and "sticky" data) is compared. If this value is strictly greater than the boundary value, the truncated product is incremented. (In the case where it overflows the available significands, the output exponent is one greater than the provisional exponent.) To avoid lengthy tests, these boundaries are indexed by a composition of bitfields $4r + 2s + l$, where r is the rounding mode, s is the sign of the input, and l is the least significant bit of the truncated exponent.

5.2.2 Binary-Decimal Conversion Algorithm

Consider converting from a binary format with precision p to a decimal format with d decimal digits. An important difference from decimal-binary conversion is that the exponent in the output is supposed to be chosen as close to zero as possible for exact results and as small as possible for inexact results. The core algorithm is given as follows:

- Handle NaN, infinity and zero, and extremely large and small inputs.
- Strip off the sign, so the input is $2^e m$ with $0 < m < 2^p$.
- Normalize the input so that $2^{p-1} \leq m < 2^p$ (this will usually be the case already).
- Filter out and handle exact results where the main path would give the wrong choice of output exponent (albeit numerically the same).
- Pick provisional decimal exponent f based on m and e.
- Form scaled product $\frac{2^e}{10^f} m$ to approximate the decimal coefficient.
- Round depending on the sign and rounding mode to give decimal coefficient n. If $n = 10^d$, set $n = 10^{d-1}$ and add 1 to the provisional exponent.
- Check for overflow, underflow, and inexactness and combine with the sign.

We consider some of the main steps in more detail, starting with the filtering out of exact cases where we cannot rely on the main path of the algorithm to correctly force the exponent toward zero. We let through either inexact cases or those where the main path will behave correctly, e.g., large integers beyond the output coefficient

range such as 2^{200}. To maximize code sharing in this part of the algorithm, we always treat the input as a quad precision number, so $2^{112} \le m < 2^{113}$, and the exponent is adjusted accordingly. We can pass through any inputs with $e > 0$ since in this case, the input is $\ge 2^{113}$, which is too large for the significand of any of the decimal formats. So we can assume that $e \le 0$; let us write t for the number of trailing zeros in m. Now, there are two cases:

- If $e + t \ge 0$, the input is an integer, so we treat it specially iff it fits in the coefficient range. We just need to test if the shifted coefficient $m' = m/2^{-e}$ is within range for the coefficient range of the output format. If so, that is the output coefficient, and the output exponent is zero. Otherwise, we can pass through to the main path.
- If $a = -(e + t) > 0$, then we have a noninteger input. If special treatment is necessary, the result will have a coefficient of $m' = 5^a(m/2^t)$ and an exponent of $-a$. If $a > 48$, this can be dismissed since $5^{49} > 10^{34}$, which is too big for any decimal format's coefficient range. Otherwise, we determine whether we are indeed in range by a table based on a and, if so, get the multiplier from another table indexed by a.

Now, we consider the main path. If we set $f = \lceil (e + p) \log_{10} 2 \rceil - d$, we have $10^{f-1+d} < 2^{e+p} \le 10^{f+d}$. This means that for *any* floating-point number x with exponent e, we have

$$10^{f-1}10^{d-1} = 10^{f-1+d}/10 < 2^{e+p}/10 < 2^{e+p-1} \le x < 2^{e+p} \le 10^{f+d}.$$

That is, x lies strictly between the smallest normalized decimal number with exponent $f - 1$ and the next beyond the largest with exponent f. We therefore have only two possible choices for the "provisional exponent" f. (We say provisional because rounding may still bump it up to the next decade, but this is easy to check at the end.) And that is without any knowledge of m, except the assumption that it is normalized. Moreover, we can pick the exact m_{min} that acts as the breakpoint between these possibilities:

$$m_{min} = \lceil 10^{f+d-1}/2^e \rceil - 1,$$

so $2^e m_{min} < 10^{f+d-1} \le 2^e(m_{min} + 1)$. Thus, we can pick the provisional exponent to be $f - 1$ for all $m \le m_{min}$ and f for all $m > m_{min}$, and we are sure that $10^f 10^{d-1} \le x < 10^f 10^d$. In the future, we will just use f for this correctly chosen provisional exponent.

Now, we aim to compute n with $10^f n \approx 2^e m$, i.e., the left-hand side should be a correctly rounded approximation of the right-hand side. Staying with the provisional exponent f, this entails finding a correctly rounded $n \approx \frac{2^e}{10^f} m$. We do this multiplying by an approximation to $2^e/10^f$ scaled by 2^q for some suitably chosen q and rounded up, i.e.,

$$r = \lceil 2^{q+e}/10^f \rceil,$$

and truncating by shifting to the right by q bits. How do we choose q? From the results above, we know an ϵ such that if $n \ne \frac{2^e}{10^f} m$ or $n + \frac{1}{2} \ne \frac{2^e}{10^f}$, then their relative difference is at least ϵ. We have

$$\frac{2^{q+e}}{10^f} \le r < \frac{2^{q+e}}{10^f} + 1,$$

and therefore (since $m < 2^p$)

$$\frac{2^e}{10^f} m \le \frac{mr}{2^q} < \frac{2^e}{10^f} m + 2^{p-q}.$$

We want to make a rounding decision for $\frac{2^e}{10^f} m$ based on $\frac{mr}{2^q}$. First, since the latter is an overestimate, it is immediate that if the true result is $\ge$ a rounding boundary, so is the estimate. So we just need to check that if the true answer is $<$ a rounding boundary, so is the estimate. We know that if the true answer is $<$ a rounding boundary, then so is $\frac{2^e}{10^f} m(1 + \epsilon)$, so it suffices to show that

$$\frac{2^e}{10^f} m + 2^{p-q} \le \frac{2^e}{10^f} m(1 + \epsilon),$$

i.e., that $2^{p-q} \le \frac{2^e}{10^f} m\epsilon$. By the choice of f made, we have $10^f 10^{d-1} \le 2^e m$, so it suffices to establish that $2^{p-q} \le 10^{d-1}\epsilon$.

We also want to be able to test whether a conversion is exact. If it is, then the fractional part of $\frac{mr}{2^q}$ is between 0 and 2^{p-q} by the bound noted above. If it is not, then the fractional part is at least $10^{d-1}\epsilon - 2^{p-q}$. To discriminate cleanly, we want this to be at least 2^{p-q}, so we actually want to pick a q such that $2^{p-q+1} \le 10^{d-1}\epsilon$ or, in other words, $2^q \ge \frac{2^{p+1}}{10^{d-1}\epsilon}$.

For example, to convert from Binary64 to Decimal64, we need $2^q \ge \frac{2^{54}}{10^{15}2^{-115.53}}$ (based on the above best approximation results), i.e., $q \ge 120$. To convert from Binary32 to Decimal32, we need $2^q \ge \frac{2^{25}}{10^6 2^{-51.32}}$, i.e., $q \ge 57$.

When q is chosen satisfying these constraints, we just need to consider the product mr to make our rounding decisions. The product with the lowest q bits removed is the correctly truncated significand. Bit $q - 1$ is the rounding bit, and the bottom $q - 1$ bits (0 to $q - 2$) can be considered as a sticky bit: zero out the lowest p bits and test whether the result is zero or nonzero or, in other words, test whether the bitfield from bit p to bit $q - 2$ is nonzero. Using this information, we can correctly round in all rounding modes. The approach uses exactly the same boundary tables as in the decimal-binary case discussed previously.

6 Nonhomogeneous Decimal Floating-Point Operations

The IEEE 754R Standard mandates that the binary and the decimal floating-point operations of addition, subtraction, multiplication, division, square root, and fused multiply-add have to be capable of rounding results correctly to any supported floating-point format, for operands in any supported floating-point format in the same base (2 or 10).

The more interesting cases are those where at least some of the operands are represented in a wider precision than that of the result (such a requirement did not exist in IEEE 754-1985). There are thus a relatively large number of operations that need to be implemented in order to comply with this requirement. Part of these are redundant (because of the commutative property), and others are trivial, but several

where at least some of the operands have a larger precision N_0 than the precision N of the result require a more careful implementation. One solution is to implement each operation separately, for example, for $\mathrm{Decimal128} + \mathrm{Decimal64} = \mathrm{Decimal64}, \mathrm{Decimal128} + \mathrm{Decimal128} = \mathrm{Decimal64}$, etc. However, since these nonhomogeneous operations will occur quite rarely in practice, an alternate solution was chosen. If a floating-point operation has to be performed on operands of precision N_0 and the result has to be rounded to precision $N < N_0$, the implementation is based on

1. an existing operation of the same type that accepts operands of precision N_0 and rounds the result to precision N_0 and
2. an existing floating-point conversion operation from precision N_0 to precision N.

This is a more economical solution than implementing an independent operation with operands of precision N_0 and result of precision N, especially since the number of pairs (N_0, N) can be quite large for practical purposes when several floating-point formats are supported (the same method is applicable to binary floating-point operations, where the precisions may include formats with 24, 53, 64, and 113 bits in the significand). However, this two-step method works correctly only with directed IEEE rounding modes: rounding down, rounding up, and rounding toward zero. If the rounding method is to nearest-even or to nearest-away, then in rare cases, double rounding errors (of 1 ulp) may occur (rounding to nearest-away is similar to the rounding to nearest-even, with one difference: midpoints between two consecutive floating-point numbers are rounded to the floating point of larger magnitude of the two by the former and to the floating-point with an even significand by the latter). To correct against the possibility of such errors, a third component is added to the implementation:

3. correction logic against double rounding errors.

The conditions that cause double rounding errors and the logical equations for their correction will be presented next—first for rounding to nearest-even and then for rounding to nearest-away.

In rounding-to-nearest-even mode, double rounding errors may occur when the exact result res_0^* of a floating-point operation with operands of precision N_0 is rounded correctly (in the IEEE 754R sense) first to a result res_0 of precision N_0, and then, res_0 is rounded again to a narrower precision N. Sometimes, the result res does not represent the IEEE-correct result res' that would have been obtained were the original exact result res_0^* rounded directly to precision N. In such cases, res differs from res' by 1 ulp, and the error that occurs is called a double rounding error. Only positive results will be considered in this context, as the treatment of negative results is similar because rounding to nearest is symmetric with respect to zero. A double rounding error for rounding to nearest-even can be upward (when the result res is too large by 1 ulp) or downward (when the result res is too small by 1 ulp).

For example, consider the addition operation of two decimal floating-point values in 128-bit format $(N_0 = 34)$, with the result rounded correctly to 128-bit format $(N_0 = 34)$, as specified by the IEEE 754R Standard:

$$\begin{aligned} a &= 10000000000000015 * 10^{-16}, \\ b &= -1 * 10^{-34}, \\ res_0 &= (a+b)_{RN,34} \\ &= 1000000000000001500000000000000000 * 10^{-33}. \end{aligned}$$

(The notation $x_{RN,34}$ indicates the rounding of x to nearest-even to 34 decimal digits.) Next, the result res_0 is rounded to 64-bit format $(N = 16)$:

$$res = (res_0)_{RN,16} = 1000000000000002 * 10^{-15}.$$

This result is different from what would have been obtained by rounding a + b directly to precision $N = 16$; a double rounding error of 1 ulp, upward, has occurred:

$$\begin{aligned} &res' \\ &= (1000000000000001499999999999999999 * 10^{-34})_{RN,16} \\ &= 1000000000000001 * 10^{-15}. \end{aligned}$$

However, not all nonhomogeneous operations can lead to double rounding errors. For example, of the eight possible variations of addition (each argument and the result can be in any of the three formats), only three required the presence of correction logic, where some operands are in Decimal128 format, and the result is in Decimal64. In general, correction of double rounding errors can be achieved by using minimal additional information from the floating-point operation applied to operands of precision N_0 with the result rounded to precision N_0 and the conversion of the result of precision N_0 to precision N, where $N_0 > N$. This extra information consists of four logical indicators for each of the two rounding operations (N_0 to N_0 and then N_0 to N), which can be used for correcting the result when a double rounding occurs (C notation is used; MP denotes a midpoint between two consecutive floating-point numbers):

`inexact_lt_MP`, `inexact_gt_MP`, `MP_lt_even`, `MP_gt_even`.

A fifth indicator can be derived from the first four:

`exact` = `!inexact_lt_MP` && `!inexact_gt_MP` && `!MP_lt_even` && `!MP_gt_even`.

This information is used to either add 1 or subtract 1 to/from the significand of the second result, in case a double rounding error occurred. This can be an efficient solution for implementing correctly rounded operations with operands of precision N_0 and results of precision N $(N_0 > N)$, instead of having complete implementations for these less frequently used but mandatory operations defined in IEEE 754R. (It can be noted, however, that for certain floating-point operations, under special circumstances, double rounding errors will not occur; in most cases, however, when performing two consecutive rounding operations as explained above, double rounding errors can occur. For example, if a binary floating-point division operation or square root operation is performed on an operand(s) of precision N_0 and the result of precision N_0 is

then rounded a second time to a smaller precision N, then double rounding errors are not possible if $N_0 > 2 \cdot N + 1$.)

The logic for correcting double rounding errors when rounding results to nearest-even is based on the observation that such errors occur when

1. the first rounding pulls up the result to a value that is a midpoint situated to the left of an even floating-point number (as seen on the real axis) for the second rounding (double rounding error upward) or
2. the first rounding pulls down the result to a value that is a midpoint situated to the right of an even floating-point number (on the real axis) for the second rounding (double rounding error downward).

In C-like pseudocode, the correction method can be expressed as follows (0 identifies the first rounding):

```
// avoid double rounding for rounding to nearest-even
sub1 = MP_lt_even && (inexact_gt_MP0||MP_lt_even0)
add1 = MP_gt_even && (inexact_lt_MP0||MP_gt_even0)
if sub1 // double rounding error upward
   significand- - // significand becomes odd
   if significand == b^(N - 1) - 1
      // falls below the smallest N-digit significand
      significand = b^N - 1 // largest significand in base b
      unbiased_exp- - // decrease exponent by 1
   endif
else if add1 // double rounding error downward
   significand++
   // significand becomes odd (so it cannot be b^N for b = 2
   or b = 10)
endif
// Otherwise no double rounding error; result is already correct
```

If l, r, and s represent the least significant decimal digit before rounding, the rounding digit, and the sticky bit (which is 0 if all digits to the right of r are 0 and 1 otherwise), then

$$\begin{aligned}
&\texttt{inexact_lt_MP} = ((r == 0 \;\&\&\; \texttt{s} != 0) \| \; (1 <= r \&\& r <= 4)),\\
&\texttt{inexact_gt_MP} = ((r == 5 \;\&\&\; \texttt{s} != 0) \| \; (r > 5)),\\
&\texttt{MP_lt_even} = (((l \text{ percent } 2) == 1) \,\&\&\, (r == 5) \,\&\&\, (\texttt{s} == 0)),\\
&\texttt{MP_gt_even} = (((l \text{ percent } 2) == 0) \,\&\&\, (r == 5) \,\&\&\, (\texttt{s} == 0)),\\
&\texttt{exact} = ((r == 0) \;\&\&\; (\texttt{s} == 0)).
\end{aligned}$$

The correction logic modifies the result res of the second rounding only if a double rounding error has occurred; otherwise, $res' = res$ is already correct. Note that after correction, the significand of res' can be smaller by 1 ulp than the smallest N-digit significand in base b. The correction logic has to take care of this corner case too. For the example considered above, $l_0 = 9$, $r_0 = 9$, $s_0 = 0$, $l = 1$, $r = 5$, and $s = 0$. It follows that $add1 = 0$ and $sub1 = 1$, and so, the double rounding error upward will be corrected by subtracting 1 from the significand of the result. (But in practice, for the BID library implementation, it was not necessary to determine l, r, and s in order to calculate the values of these logical indicators.)

The logic shown above remedies a double rounding error that might occur when rounding to nearest-even and generates the correct value of the result in all cases. However, in certain situations, it is useful to know also the correct rounding indicators from the second rounding. For example, they can be used for correct denormalization of the result res' if its exponent falls below the minimum exponent in the destination format. In this case, the complete logic is expressed by the following pseudocode, where in addition to correcting the result, the four rounding indicators from the second rounding are also corrected if double rounding errors occur:

```
// avoid a double rounding error and adjust rounding indicators
// for rounding to nearest-even
if MP_lt_even && (inexact_gt_MP0||MP_lt_even0)
  // double rounding error upward
  significand- - // significand becomes odd
  if significand == b^(N - 1) - 1
    // falls below the smallest N-digit significand
    significand = b^N - 1 // largest significand in base b
    unbiased_exp- - // decrease exponent by 1
  endif
  MP_lt_even = 0
  inexact_lt_MP = 1
else if MP_gt_even && (inexact_lt_MP0||MP_gt_even0)
  // double rounding error downward
  significand++
  // significand becomes odd (so it cannot be b^N for b = 2 or
  b = 10)
  MP_gt_even = 0
  inexact_gt_MP = 1
else if !MP_lt_even && !MP_gt_even && !inexact_lt_MP &
  & !inexact_gt_MP
  // if this second rounding was exact the result may still be
  // inexact because of the previous rounding
  if (inexact_gt_MP0||MP_lt_even0)
    inexact_gt_MP = 1
  endif
  if (inexact_lt_MP0||MP_gt_even0)
    inexact_lt_MP = 1
  endif
else if (MP_gt_even && (inexact_gt_MP0||MP_lt_even0))
  // pulled up to a midpoint greater than an even floating-point
  number
  MP_gt_even = 0
  inexact_lt_MP = 1
else if MP_lt_even && (inexact_lt_MP0||MP_gt_even0)
  // pulled down to a midpoint less than an even floating-point
  number
  MP_lt_even = 0
  inexact_gt_MP = 1
else
  // the result and the rounding indicators are already correct
endif
```

In rounding-to-nearest-away mode, double rounding errors may also occur just as they do in rounding-to-nearest-even mode: when the exact result res_0^* of a floating-point operation with operands of precision N_0 is rounded correctly (in the IEEE 754R sense) first to a result res_0 of precision N_0, and then, res_0 is rounded again to a narrower precision N. Sometimes, the result res does not represent

the IEEE-correct result res' that would have been obtained were the original exact result res_0^* rounded directly to precision N. In this case too, only positive results will be considered, because rounding to nearest-away is symmetric with respect to zero. A double rounding error for rounding to nearest-away can only be upward.

The same example considered above for rounding to nearest-even can be used for rounding to nearest-away. In this case, we have

$$res_0 = (a+b)_{RA,34} = 10000000000000015000000000000000000 * 10^{-34}.$$

(The notation $x_{RA,34}$ indicates the rounding of x to nearest-away to 34 decimal digits.) The result res_0 rounded to 64-bit format $(N = 16)$ is

$$res = (res_0)_{RA,16} = 1000000000000002 * 10^{-16}.$$

This result is different from what would have been obtained by rounding a + b directly to precision $N = 16$; a double rounding error of 1 ulp, upward, has occurred, because

$$res' = (10000000000000014999999999999999999 * 10^{-34})_{RA,16} = 1000000000000001 * 10^{-16}.$$

Again, correction of double rounding errors can be achieved by using minimal additional information from the floating-point operation applied to operands of precision N_0 and result rounded to precision N_0 and the conversion of the result of precision N_0 to precision N, where $N_0 > N$.

The additional information consists in this case of just three logical indicators for each of the two rounding operations (N_0 to N_0 and then N_0 to N):

`inexact_lt_MP`, `inexact_gt_MP`, `MP`.

A fourth indicator can be derived from the first three:

`exact` = !`inexact_lt_MP` && !`inexact_gt_MP` && !`MP`.

The method for correction of double rounding errors when rounding results to nearest-away is based on a similar observation that such errors occur when the first rounding pulls up the result to a value that is a midpoint between two consecutive floating-point numbers, and therefore, the second rounding causes an error of 1 ulp upward.

The correction method can be expressed as follows (0 identifies again the first rounding):

```
// avoid a double rounding error for rounding to nearest-away
sub1 = MP && (MP0||inexact_gt_MP0)
if sub1 // double rounding error upward
  significand- -
  if significand == b^(N − 1) − 1
  // falls below the smallest N-digit significand
    significand = b^N − 1
    unbiased_exp- - // decrease exponent by 1
  endif
// Otherwise no double rounding error; the result is already
correct
endif
```

Just as for rounding to nearest-even, the values of the rounding digit r and of the sticky bit s can be used to calculate

$$\texttt{inexact\_lt\_MP} = ((\texttt{r}==0\ \&\&\ \texttt{s}\ !=0)\ ||\ (1<=\texttt{r}\ \&\&\ \texttt{r}<=4)),$$
$$\texttt{inexact\_gt\_MP} = ((\texttt{r}==5\ \&\&\ \texttt{s}\ !=0)\ ||\ (\texttt{r}>5)),$$
$$\texttt{MP} = ((\texttt{r}==5)\ \&\&\ (\texttt{s}==0)),$$
$$\texttt{exact} = ((\texttt{r}==0)\ \&\&\ (\texttt{s}==0)).$$

The correction is applied to the result res of the second rounding based on the value of sub1 but only if a double rounding error has occurred. Otherwise, $res' = res$ is already correct. Note that after correction, the significand of res' can be smaller by 1 ulp than the smallest N-digit significand in base b. Again, the correction logic has to take care of this corner case too. For the example shown above, the correction signal $sub1 = 1$ is easily calculated given $r_0 = 9$, $s_0 = 0$, $r = 5$, and $s = 0$. The double rounding error is corrected by subtracting 1 from the significand of the result.

Again, it may be useful to know also the correct rounding indicators from the second rounding. The complete correction logic is similar to that obtained for rounding to nearest-even.

This method was successfully applied for implementing various nonhomogeneous decimal floating-point operations, with minimal overhead. It is worth mentioning again that the same method can be applied for binary floating-point operations in hardware, software, or a combination of the two.

7 Performance Data

In this section, we present performance data in terms of clock cycle counts needed to execute several library functions. The median and maximum values are shown. The minimum values were not considered very interesting (usually, a few clock cycles) because they reflect just a quick exit from the function call for some special-case operands. To obtain the results shown here, each function was run on a set of tests covering corner cases and ordinary cases. The mix of data has been chosen to exercise the library (including special cases such as treatment of NaNs and infinities) rather than to be a representative of a specific decimal floating-point workload.

These preliminary results give a reasonable estimate of worst case behavior, with the median information being a good measure of the performance of the library.

Test runs were carried out on four different platforms to get a wide sample of performance data. Each system was running Linux. The best performance was on the Xeon 5100 Intel®64 platform, where the numbers are within one to two orders of magnitude from those for possible hardware solutions.

Table 1 contains clock cycle counts for a sample of *arithmetic* functions. These are preliminary results as the library is in the pre-beta development stage. A small number of optimizations were applied, but significant improvements may still be possible through exploitation

TABLE 1
Clock Cycle Counts for a Subset of *Arithmetic* Functions {Max/Median Values}

Function Name	Intel®64 Xeon 5100 3.0 GHz	Intel®64 Xeon 3.2 GHz	IA-64 Itanium 2 1.4 GHz
abs128	19 / 1	2 / 2	44 / 44
abs64	15 / 6	6 / 5	12 / 12
add128	205 / 94	337 / 178	242 / 149
add64	133 / 71	249 / 132	219 / 118
div128	808 / 559	1369 / 1020	679 / 454
div64	266 / 171	484 / 312	294 / 180
fma64	283 / 211	487 / 365	284 / 228
maxnum128	108 / 69	187 / 130	120 / 85
minnum128	113 / 75	182 / 126	117 / 82
mul128	449 / 307	750 / 543	306 / 280
mul64	132 / 69	227 / 116	149 / 102
quantize128	97 / 92	188 / 172	100 / 98
quantize64	45 / 27	78 / 64	76 / 62
sqrt128	544 / 519	1001 / 911	458 / 431
sqrt64	194 / 188	292 / 287	223 / 213

of specific hardware features or careful analysis and reorganization of the code.

Table 2 contains clock cycle counts for *conversion* functions. It is worth noting that the BID library performs well even for conversion routines to and from string format.

Table 3 contains clock cycle counts for *other miscellaneous* IEEE 754R functions, which can be found in [20] under slightly different names. For example, `rnd_integral_away128` and `q_cmp_lt_unord128` are our implementations of, respectively, the IEEE 754R operations `roundToIntegralTiesAway` and `compareQuietLessUnordered` on the Decimal128 type.

It is interesting to compare these latencies with those for other approaches. For example, the decNumber package [9] run on the same Xeon 5100 system has a maximum latency of 684 clock cycles and a median latency of 486 clock cycles for the add64 function. As a comparison, the maximum and median latencies of a 64-bit addition on the 3.0-GHz Xeon 5100 are 133 and 71 clocks cycles, respectively. Another example is that based on its median latency (71 clock cycles for add64 in Table 1), the 64-bit BID add is less than four times slower than a four-clock-cycle single-precision binary floating-point add operation in hardware on a 600-MHz Ultra Sparc III CPU of just a few years ago.

TABLE 2
Clock Cycle Counts for a Subset of *Conversion* Functions {Max/Median Values}

Function Name	Intel®64 Xeon 5100 3.0 GHz	Intel®64 Xeon 3.2 GHz	IA-64 Itanium 2 1.4 GHz
cvt_bid128_to_int32	127 / 51	240 / 107	143 / 92
cvt_int32_to_bid64	98 / 9	167 / 13	181 / 13
cvt_int32_to_bid128	97 / 46	169 / 84	182 / 93
cvt_string_to_bid128	336 / 54	321 / 133	391 / 95
cvt_string_to_bid64	215 / 82	553 / 81	332 / 133
cvt_bid128_to_string	345 / 103	812 / 201	509 / 198
cvt bid64_to_string	130 / 84	249 / 152	281 / 155

TABLE 3
Clock Cycle Counts for a Subset of *Miscellaneous* Functions {Max/Median Values}

Function Name	Intel®64 Xeon 5100 3.0 GHz	Intel®64 Xeon 3.2 GHz	IA-64 Itanium 2 1.4 GHz
class128	116 / 22	181 / 31	81 / 5
class64	30 / 17	60 / 23	31 / 5
q_cmp_eq128	111 / 67	175 / 111	83 / 68
q_cmp_eq64	74 / 27	145 / 42	77 / 33
q_cmp_gt128	117 / 75	180 / 120	87 / 73
q_cmp_gt_eq64	54 / 41	93 / 62	53 / 47
q_cmp_lt_unord128	118 / 74	182 / 120	88 / 74
q_cmp_lt_unord64	56 / 42	93 / 63	53 / 48
rnd_integral_away128	95 / 84	190 / 164	113 / 99
rnd_integral_away64	43 / 40	71 / 61	78 / 62
rnd_integral_zero128	91 / 78	180 / 150	117 / 100
rnd_integral_zero64	41 / 13	71 / 40	78 / 51
total_order128	122 / 78	194 / 128	98 / 85
total_order_mag128	109 / 72	176 / 117	96 / 81

8 Conclusion

In this paper, we presented some mathematical properties and algorithms used in the first implementation in software of the IEEE 754R decimal floating-point arithmetic, based on the BID encoding. We concentrated on the problem of rounding correctly decimal values that are stored in binary format while using binary operations efficiently and also presented briefly other important or interesting algorithms used in the library implementation. Finally, we provided a wide sample of performance numbers that demonstrate that the possible speedup of hardware implementations over software may not be as dramatic as previously estimated. The implementation was validated through testing against reference functions implemented independently, which used, in some cases, existing multiprecision arithmetic functions.

As we look toward the future, we expect further improvements in performance through algorithm and code optimizations, as well as enhanced functionality, for example, through addition of transcendental function support. Furthermore, we believe that hardware support can be added incrementally to improve decimal floating-point performance as demand for it grows.

Appendix

Algorithms for Best Rational Approximations

The standard problem in Diophantine approximation is to investigate how well a number x can be approximated by rationals subject to an upper bound on the size of the denominator. That is, what is the minimal value of $|x - p/q|$

for $|q| \leq N$? The usual approach to finding such approximations is to generate sequences of rationals $p_1/q_1 < x < p_2/q_2$ straddling the number of interest such that $p_2q_1 = p_1q_2 + 1$, so-called *convergents*.

Usually, in the standard texts, one assumes that x is itself irrational, so the case $p/q = x$ does not need to be considered. In our context, however, the x we will be concerned with are rational, so we need to make a few small modifications to the standard approach. Moreover, it will be useful for us to be able to restrict both the denominator *and* numerator of the rational approximations, and it is simple to extend the usual techniques to this case. We will distinguish between left and right approximants and seek to solve the two problems:

- Best left approximation to x: what is the rational p/q with $|p| \leq M$, $|q| \leq N$, and $p/q < x$ that minimizes $x - p/q$?
- Best right approximation to x: what is the rational p/q with $|p| \leq M$, $|q| \leq N$, and $x < p/q$ that minimizes $p/q - x$?

For simplicity, we will consider nonnegative numbers in what follows, as well as fractions where both the numerator and the denominator are positive. This is no real loss of generality since a best left approximation to $-x$ is the negation of the best right approximation to x, and so on. We will focus on finding best approximations subject only to a denominator bound, but incorporating a numerator bound is analogous. Moreover, we will always assume that $M \geq 1$ and $N \geq 1$, since otherwise, the problem is of little interest. But note that even with this assumption, there may be *no* best right approximation within some bounds. For example, if $x = 2$, there is no rational $x < p/q$ such that $|p| \leq 1$.

Straightforward methods for solving the one-sided approximation problems can be based on a slightly modified notion of convergent. A pair of rationals $(p_1/q_1, p_2/q_2)$ is said to be a pair of *left convergents* if $p_2q_1 = p_1q_2 + 1$ and

$$p_1/q_1 < x \leq p_2/q_2$$

and a pair of *right convergents* if, again, $p_2q_1 = p_1q_2 + 1$ and also

$$p_1/q_1 \leq x < p_2/q_2.$$

Note that a pair $(p_1/q_1, p_2/q_2)$ is a convergent in the usual sense if it is both a pair of left convergents and a pair of right convergents, since in this case, proper inequalities hold at both sides. If we consider best left and right approximations, then left and right convergent pairs, respectively, retain the key property of two-sided convergent pairs.

Theorem 1. *Suppose $p_1/q_1 < x \leq p_2/q_2$ with $p_2q_1 = p_1q_2 + 1$. Any other rational a/b that is a better left approximation to x than p_1/q_1 must have $a \geq p_1 + p_2$ and $b \geq q_1 + q_2$.*

Proof. Observe that such an a/b must satisfy $p_1/q_1 < a/b < x \leq p_2/q_2$. This implies that

$$\begin{aligned} p_1b &< aq_1, \\ aq_2 &< p_2b, \end{aligned}$$

and so, since we are dealing with integers,

$$\begin{aligned} p_1b + 1 &\leq aq_1, \\ aq_2 + 1 &\leq p_2b, \end{aligned}$$

and, therefore,

$$\begin{aligned} p_1bq_2 + q_2 &\leq aq_1q_2, \\ aq_1q_2 + q_1 &\leq p_2bq_1, \end{aligned}$$

yielding $p_1bq_2 + q_1 + q_2 \leq p_2bq_1$. By the convergent property, $p_2q_1 = p_1q_2 + 1$, so $p_1bq_2 + q_1 + q_2 \leq p_1bq_2 + b$, i.e., $q_1 + q_2 \leq b$. The proof of $p_1 + p_2 \leq a$ is exactly analogous. □

Theorem 2. *Suppose $p_1/q_1 \leq x < p_2/q_2$ with $p_2q_1 = p_1q_2 + 1$. Any other rational a/b that is a better right approximation to x than p_2/q_2 must have $a \geq p_1 + p_2$ and $b \geq q_1 + q_2$.*

This theorem is the basis for the approximation algorithms: we generate a sequence of left (respectively, right) convergent pairs with increasing denominators as far as possible until we exceed the bound. To start off, we use a pair of approximants with denominator 1, which we know is within the bounds (if $N < 1$, the whole question makes little sense). We can get a starting pair of left and right convergents by, respectively

- $p_1/q_1 = \lceil x \rceil - 1$ and $p_2/q_2 = \lceil x \rceil$ for left convergents and
- $p_1/q_1 = \lfloor x \rfloor$ and $p_2/q_2 = \lfloor x \rfloor + 1$ for right convergents.

From the preceding theorems, we know if we ever reach a state where $q_1 + q_2 > N$, we know, respectively, that p_1/q_1 is the best left approximation with denominator bound N and p_2/q_2 is the best right approximation with denominator bound N. If on the other hand, $q_1 + q_2 < N$, define

$$\begin{aligned} p &= p_1 + p_2, \\ q &= q_1 + q_2. \end{aligned}$$

The fraction p/q is called the *mediant* of the two fractions p_1/q_1 and p_2/q_2. Note that the pairs $(p_1/q_1, p/q)$ and $(p/q, p_2/q_2)$ both have the convergent property:

$$\begin{aligned} (p_1 + p_2)q_1 &= p_1(q_1 + q_2) + 1, \\ p_2(q_1 + q_2) &= (p_1 + p_2)q_2 + 1, \end{aligned}$$

since these both reduce to the original property after canceling common terms from both sides:

$$p_2q_1 = p_1q_2 + 1.$$

Note that provided p_1/q_1 and p_2/q_2 are both in their lowest terms, so is p/q, because of the convergent property $p_2q_1 = p_1q_2 + 1$. Moreover, this property implies that the fraction p/q is "automatically" in its lowest terms. So now, we have two possibilities:

- If $p/q < x$ (for left approximations) or $p/q \leq x$ (for right approximations), then the next pair of one-sided convergents will be $(p/q, p_2/q_2)$.
- If $x \leq p/q$ (for left approximations) or $x < p/q$ (for right approximations), then the next pair of left convergents will be $(p_1/q_1, p/q)$.

We keep iterating this procedure; since the denominators properly increase, we will eventually terminate when the

next mediant exceeds the denominator limit. In practice, one can sometimes reach pathological situations where an infeasible number of mediant steps are to be taken. So rather than make these steps one at a time, we can condense multiple "go left" and "go right" operations into one. (This is effectively like a traditional continued fraction algorithm, except that we are more delicate about the limits on the numerator and the denominator.) The following defines the number of left and right steps we can take when finding left approximations before we either need a step of the other kind or exceed the denominator bound:

$$k_{left} = \begin{cases} \left\lfloor \frac{N-q_2}{q_1} \right\rfloor, & \text{if } q_1 x = p_1, \\ \min\left(\left\lfloor \frac{N-q_2}{q_1} \right\rfloor, \left\lfloor \frac{p_2 - q_2 x}{q_1 x - p_1} \right\rfloor \right), & \text{otherwise,} \end{cases}$$

and

$$k_{right} = \begin{cases} \left\lfloor \frac{N-q_1}{q_2} \right\rfloor, & \text{if } q_2 x = p_1, \\ \min\left(\left\lfloor \frac{N-q_1}{q_2} \right\rfloor, \left\lceil \frac{q_1 x - p_1}{p_2 - q_2 x} \right\rceil - 1 \right), & \text{otherwise.} \end{cases}$$

Now, if $k_{left} > 0$, our next left convergent pair will be $(p_1/q_1, p/q)$, where $p = p_1 k_{left} + p_2$, and $q = q_1 k_{left} + q_2$. Otherwise, if $k_{right} > 0$, our next left convergent pair will be $(p/q, p_2/q_2)$, where $p = p_1 + p_2 k_{right}$, and $q = q_1 + q_2 k_{right}$. Otherwise, we have reached the denominator limit, and the best left approximation is p_1/q_1. When finding right approximations, the corresponding k_{left} and k_{right} are just a little different:

$$k_{left} = \begin{cases} \left\lfloor \frac{N-q_2}{q_1} \right\rfloor, & \text{if } q_1 x = p_1, \\ \min\left(\left\lfloor \frac{N-q_2}{q_1} \right\rfloor, \left\lceil \frac{p_2 - q_2 x}{q_1 x - p_1} \right\rceil - 1 \right), & \text{otherwise,} \end{cases}$$

and

$$k_{right} = \begin{cases} \left\lfloor \frac{N-q_1}{q_2} \right\rfloor, & \text{if } q_2 x = p_1, \\ \min\left(\left\lfloor \frac{N-q_1}{q_2} \right\rfloor, \left\lfloor \frac{q_1 x - p_1}{p_2 - q_2 x} \right\rfloor \right), & \text{otherwise.} \end{cases}$$

Of course, to find the best approximation to a number, we see which of its best left and right approximations is closer and pick either if they are equally close. (If the number is itself rational, we may, depending on context, also include the number if its denominator is within range.)

Another advantage of distinguishing left and right approximations is that we can very easily generate the closest k left or right approximations. For example, let p_1/q_1 be the best left approximation to x with denominator bound N. The next-best left approximation to x with denominator bound N must in fact be the best left approximation to p_1/q_1 (since if it were $> p_1/q_1$, it would contradict the best approximation status of p_1/q_1), and so forth. In this way, we can also enumerate all rationals subject to the denominator bound that are within some given $\epsilon > 0$ of x. Of course, depending on x, N, and ϵ, the number of such approximations may mean that they are not *feasibly* enumerable.

REFERENCES

[1] G. Bohlender and T. Teufel, "A Decimal Floating-Point Processor for Optimal Arithmetic," *Computer Arithmetic: Scientific Computation and Programming Languages,* pp. 31-58, Teubner Stuttgart, 1987.

[2] S. Boldo and G. Melquiond, "When Double Rounding Is Odd," *Proc. 17th IMACS World Congress, Scientific Computation, Applied Math. and Simulation,* 2005.

[3] W. Buchholz, "Fingers or Fists? (The Choice of Decimal or Binary Representation)," *Comm. ACM,* vol. 2, no. 12, pp. 3-11, 1959.

[4] F.Y. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz, and S.R. Carlough, "The IBM z900 Decimal Arithmetic Unit," *Proc. 35th Asilomar Conf. Signals, Systems and Computers,* vol. 2, p. 1335, Nov. 2001.

[5] M.S. Cohen, T.E. Hull, and V.C. Hamacher, "CADAC: A Controlled-Precision Decimal Arithmetic Unit," *IEEE Trans. Computers,* vol. 32, no. 4, pp. 370-377, Apr. 1983.

[6] M. Cornea, C. Anderson, J. Harrison, P.T.P. Tang, E. Schneider, and C. Tsen, "An Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07),* pp. 29-37, 2007.

[7] M.F. Cowlishaw, "Densely Packed Decimal Encoding," *IEE Proc.—Computers and Digital Techniques,* vol. 149, pp. 102-104, May 2002.

[8] M.F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH '03),* pp. 104-111, June 2003.

[9] M.F. Cowlishaw, *The decNumber Library,* http://www2.hursley.ibm.com/decimal/decnumber.pdf, 2006.

[10] A.Y. Duale, M.H. Decker, H.-G. Zipperer, M. Aharoni, and T.J. Bohizic, "Decimal Floating-Point in z9: An Implementation and Testing Perspective," *IBM J. Research and Development,* http://www.research.ibm.com/journal/rd/511/duale.html, 2007.

[11] M.A. Erle, J.M. Linebarger, and M.J. Schulte, "Potential Speedup Using Decimal Floating-Point Hardware," *Proc. 36th Asilomar Conf. Signals, Systems, and Computers,* pp. 1073-1077, Nov. 2002.

[12] M.A. Erle and M.J. Schulte, "Decimal Multiplication via Carry-Save Addition," *Proc. IEEE 14th Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP '03),* pp. 348-358, June 2003.

[13] European Commission, *The Introduction of the Euro and the Rounding of Currency Amounts,* http://europa.eu.int/comm/economy_finance/publications/euro_papers/2001/eup22en.pdf, Mar. 1998.

[14] S.A. Figueroa, "When Is Double Rounding Innocuous?" *ACM SIGNUM Newsletter,* vol. 20, no. 3, pp. 21-26, 1995.

[15] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys,* vol. 23, pp. 5-48, 1991.

[16] G. Gray, "UNIVAC I Instruction Set," *Unisys History Newsletter,* vol. 2, no. 3, 2001.

[17] T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-Bit Performance," *IEEE Micro,* vol. 19, no. 3, pp. 73-85, May/June 1999.

[18] IBM Corp., *The Telco Benchmark,* http://www2.hursley.ibm.com/decimal/telco.html, Mar. 1998.

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

[20] *Draft Standard for Floating-Point Arithmetic P754, Draft 1.2.5.,* IEEE, http://www.validlab.com/754R/drafts/archive/2006-10-04.pdf, Oct. 2006.

[21] *ISO 1989:2002 Programming Languages—COBOL,* ISO Standards, JTC 1/SC 22, 2002.

[22] *C# Language Specification,* Standard ECMA-334, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf, 2005.

[23] Sun Corp., *Class BigDecimal,* http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html, 2008.

[24] P. Tang, "Binary-Integer Decimal Encoding for Decimal Floating Point," technical report, Intel Corp., http://754r.ucbtest.org/issues/decimal/bid_rationale.pdf.

[25] *XML Scheme Part 2: Datatypes Second Edition,* World Wide Web Consortium (W3C) recommendation, http://www.w3.org/Tr/2004/REC-xmlschema-2-20041028/, Oct. 2004.

[26] L. Wang, "Processor Support for Decimal Floating-Point Arithmetic," technical report, Electrical and Computer Eng. Dept., Univ. of Wisconsin, Madison, year?
[27] M.H. Weik, *The ENIAC Story*, http://ftp:.arl.mil/~mike/comphist/eniac-story.html, 2007.

Marius Cornea received the MSc degree in electrical engineering from the Polytechnic Institute of Cluj, Romania, and the PhD degree in computer science from Purdue University, West Lafayette, Indiana. He joined Intel Corp., Hillsboro, Oregon, in 1994 and is now a principal engineer and the manager of the Numerics Team. His work has involved many aspects of scientific computation, design and development of numerical algorithms, floating-point emulation, exception handling, and new floating-point instruction definition and analysis. He is a member of the IEEE.

John Harrison received the bachelor's degree in electronic engineering, the diploma in computer science, and the PhD degree in computer science from the University of Cambridge, England. He joined Intel Corp., Hillsboro, Oregon, in 1998 and is now a principal engineer. He specializes in formal verification, automated theorem proving, floating-point arithmetic, and mathematical algorithms. He is a member of the IEEE.

Cristina Anderson received the MS degree in computer science from the Polytechnic University of Bucharest, Romania, in 1995 and the PhD degree in computer engineering, from Southern Methodist University, Dallas, in 1999. She has been with Intel Corp., Hillsboro, Oregon, since 1999. Her topics of interest include numerical algorithms and computer architecture, with a focus on arithmetic unit design.

Ping Tak Peter Tang received the PhD degree in mathematics from the University of California, Berkeley. He was, until recently, a senior principal engineer with Intel Corp., Hillsboro, Oregon. He is currently with D.E. Shaw Reaserch, L.L.C., New York. His main interest is in numerical algorithms, as well as error analysis of such algorithms. He is a member of the IEEE.

Eric Schneider received the MS degree in computer science from Washington State University. He has been with Intel Corp., Hillsboro, Oregon, since 1995. He is interested in all aspects of CPU validation, specializing in ALU validation along with random instruction test generation.

Evgeny Gvozdev graduated from Moscow State University. He formerly worked as a software engineer in the Russian Federal Nuclear Center, Sarov. He joined Intel Corp., Nizhny Novgorod Region, Russia, in 1996 and currently works for the Intel MKL team. His special interest is in testing methodologies for numerical functions, which he has applied to support the quality assurance of Intel mathematical libraries.