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

Marius Cornea, Cristina Anderson, John Harrison,
Ping Tak Peter Tang, Eric Schneider, Charles Tsen[1]
Intel Corporation, [1]University of Wisconsin

Abstract

The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [1] was revised [2], and an important addition is the definition of decimal floating-point arithmetic. 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. 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 for the more important or interesting operations of addition, multiplication, division, and conversions between binary and decimal floating-point formats are also outlined. 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.

1. Introduction

There is increased interest in decimal floating-point arithmetic in both industry and academia as the IEEE 754R [2]. Draft approaches the stage where it may soon become the new standard for floating-point arithmetic. The P754 Draft describes two different possibilities for encoding decimal floating-point values: the binary encoding, based on using a Binary Integer [3] to represent the significand (BID, or Binary Integer Decimal), and the decimal encoding, which uses the Densely Packed Decimal (DPD) [4] 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.

1.1 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 is required by certain applications which need results identical to those calculated by hand [5]. This is true for currency conversion [6], banking, billing, and other financial applications. Sometimes these requirements are mandated by law [6], other times they are necessary to avoid large accounting discrepancies [7].

Because of the importance of this problem a number of decimal solutions exist, both hardware and software. Software solutions include C# [8], COBOL [9], and XML [10], which provide decimal operations and datatypes. Also, Java and C/C++ both have packages, called BigDecimal [11] and decNumber [12], respectively. Hardware solutions were more prominent earlier in the computer age with the ENIAC [13] and UNIVAC [14]. However, more recent examples include the CADAC [15], IBM's z900 [16] and z9 [17] architectures, and numerous other proposed hardware implementations [18] [19] [20]. More hardware examples can be found in [21], and a more in-depth discussion is found

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

in Wang's Processor Support for Decimal Floating-Point Arithmetic [22].

Estimations have been made that hardware approaches to decimal floating-point will have average speedups of 100-1000 times over software [7]. However, results from our implementation show that this is unlikely, as 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 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 1000, to calculate and subtract the remainder, and possibly to add 1000 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: 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 next 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, $\lfloor C \cdot k_3 \rfloor = 1234567890123456$, 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 $\lfloor (C \cdot h_3) \cdot 2^{-3} \rfloor = 1234567890123456$, 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 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).

A third method is obtained if instead of multiplying by 2^{-3} at the end, C is shifted 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.

A 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

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

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 $q \in N, q > 0$, $C \in N, 10^{q-1} \leq C < 10^q - 1, x \in \{1, 2, 3, \ldots, q-1\}$, and $\rho = log_2 10$. If $y \in N$ 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, $C = d_0 \cdot 10^{q-1} + d_1 \cdot 10^{q-2} + \ldots + d_{q-2} \cdot 10^1 + d_{q-1}$, 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 $L = C \ \% \ 10^x = d_{q-x} \cdot 10^{x-1} + d_{q-x+1} \cdot 10^{x-2} + \ldots + d_{q-2} \cdot 10^1 + d_{q-1} \in [0, 10^x - 1]$. Then the minimum value of y can be determined such that $\lfloor C \cdot k_x \rfloor = H \Leftrightarrow H \leq (H + 10^{-x} \cdot L) \cdot (1+\varepsilon) < H + 1 \Leftrightarrow 10^{-x} \cdot \varepsilon < 10^{-2x}/(H + 1 - 10^{-x})$ (1)

But $10^{-x} \cdot \varepsilon$ is the absolute error in $k_x = 10^{-x} \cdot (1+\varepsilon)$, and it is less than one 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 2 to separate 10^{-x} and $k_x = 10^{-x} \cdot (1+\varepsilon)$ (that 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 2 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 ulp(k_x) = $2^{-\lfloor \rho \cdot x \rfloor - y}$. In order to satisfy inequality (1), it is sufficient to have (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 $y \geq \lceil \{\rho \cdot x\} + \rho \cdot q \rceil$, q.e.d. 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 pre-calculated and are stored as pairs (K_x, e_x), with K_x and e_x positive integers, and $k_x = K_x \cdot 2^{-ex}$. This allows for efficient implementations, exclusively in the integer domain, of several decimal floating-point operations, in particular 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 $C \ \% \ 10^x$ and to compare it with 0 and with 50. . .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 0). 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 $q \in N, q > 0$, $x \in \{1, 2, 3, \ldots, q-1\}$, $C \in N, 10^{q-1} \leq C < 10^q - 1$, $C = 10^x \cdot H + L$ where $H \in [10^{q-x-1}, 10^{q-x} - 1]$, $L \in [0, 10^x - 1], H, L \in N$, $f = C \cdot k_x - \lfloor C \cdot k_x \rfloor$, $\rho = log_2 10$, $y \in N, y \geq 1 + \lceil \rho \cdot q \rceil$, $k_x = 10^{-x} \cdot (1+\varepsilon), 0 < \varepsilon < 2^{-y+1}$. 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 $\frac{1}{2} + 10^{-x} < f < 1$

(Proof not included, but 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 one 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 $H = 10^{q-x} - 1 = 99\ldots9$ ($q - x$ 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 $y = 62$ (important, 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 Property 3.

Property 3: Let $q \in N, q > 0$, $C \in N, 10^{q-1} \leq C < 10^q - 1$, $x \in \{1, 2, 3, \ldots, q-1\}$, and $\rho = log_2 10$. If $y \in N$, $y \geq \lceil\{\rho \cdot x\} + \rho \cdot q\rceil - x$ 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 0 (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 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 0, 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 pseudo-code 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 ($C1, C2 \in Z$, $0 < C1 < 10^p$, and $0 < C2 < 10^p$). For simplicity it is assumed that $n1 \geq 0$ and $e1 \geq e2$, 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:

```
$q1, q2$ = nr. of decimal digits needed to represent
        $C1, C2$ // table lookup
if $|q1 + e1 - q2 - e2| \geq p$
    $n = C1 \cdot 10^{e1}$ or $n = C1 \cdot 10^{e1} \pm 10^{e1+q1-p}$
    (inexact)
else // if $|q1 + e1 - q2 - e2| \leq p - 1$
    $C' = C1 \cdot 10^{e1-e2} + C2$ // binary integer
    $q$ = number of decimal digits needed to
        represent $C'$ // table lookup
    if $q \leq p$
        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 $C'$, $x \in [1, p]$
    $C'' = C' + 1/2 \cdot 10^x$
    $k_x = 10^{-x} \cdot (1 + \epsilon), 0 < \epsilon < 2^{-\lceil 2 \cdot \rho \cdot p\rceil}$
    $C^* = C'' \cdot k_x = C'' \cdot K_x \cdot 2^{-Ex}$
    $f^*$ = the fractional part of $C^*$ // lower $Ex$ 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
$q$ = number of decimal digits needed to represent
      $C'$ // 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 exact 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. The notation $digits(X)$ is used for the minimum number of decimal digits needed to represent the integer value X, and p is the maximum digit size of the decimal coefficient, as specified by the format: $p = 16$ for the 64-bit decimal and $p = 34$ for the 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 $C1 < C2$
    $nd = digits(C2) - digits(C1)$
    $C1' = C1 \cdot 10^{nd}$
    $scale = p - 1$
    if($C1' < C2$)
        $scale = scale + 1$
    endif
    $C1^* = C1' \cdot 10^{scale}$
    $Q0 = 0$
    $e = e1 - e2 - scale - nd$ // expected
                                // exponent
else
    $Q0 = \lfloor C1/C2 \rfloor$, $R = C1 - Q0 \cdot C2$ // long
                    // integer divide and remainder
    if ($R == 0$)
        return $Q0 \cdot 10^{e1-e2}$ // result is exact
    endif
    $scale = p - digits(Q0)$
    $C1^* = R \cdot 10^{scale}$
    $Q0 = Q0 \cdot 10^{scale}$
    $e = e1 - e2 - scale$ // expected exponent
endif
$Q1 = C1^* \;/\; C2$, $R = C1^* - Q1 \cdot C2$
    // multiprecision integer  divide
$Q = Q0 + Q1$
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 is somewhat similar, in the sense that it requires computing an integer square root of $2 \cdot p$ digits (while division required an integer divide of $2 \cdot p$ by p digits). Due to lack of space we cannot cover the implementation of these integer operations. Rounding of the square root result is based on a test involving long integer

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

multiplication.

5. Conversions Between Binary and Decimal Formats

Conversions between IEEE 754R binary formats and the new decimal formats may sometimes be needed in user applications, and the revised standard draft specifies that these should be correctly rounded. We first sketch the general implementation for two typical cases, and then consider in more detail the intermediate accuracy required to ensure correct rounding.

5.1. Conversion from Double Precision to 64-bit Decimal

The double precision argument encodes the value
$a = (-1)^{sa} \cdot 2^{ea} \cdot m_a = (-1)^{sa} \cdot 2^k \cdot C_a$,
with C_a an integer in the range $[0, 2^{53})$ and k an integer in the range $[-1074, 971]$. We first check for and handle zeros, infinities and NaNs, and normalize denormal inputs, so we can thereafter assume $2^{52} \leq C_a < 2^{53}$ and $-1126 \leq k \leq 971$ (the lower exponent range expands because we force normalization).

The standard specifies that when a binary-decimal conversion is exact, the preferred exponent is zero, and otherwise is as small as possible. The main path of the algorithm is for the inexact case, where the preferred exponent choice implies that the output should be normalized. However, first we intercept the exact cases where the main path might result in a suboptimal choice of exponent. These cases can be checked for based only on the input exponent and the number of trailing zeros in the significand, using a small table of upper limits; no divisibility testing is needed.

For the main path, normalization constrains the input in some range $2^{\alpha} \leq a < 2^{\alpha+1}$, so there are at most two possible choices for the decimal exponent, say f and $f + 1$. For each input exponent k we tabulate the corresponding f as well as the significand breakpoint m so that if $C_a \leq m$ the provisional exponent selected is f and otherwise is $f + 1$. (Once we get down to the minimal output exponent we need to set f to that value and choose a dummy value of m so that the test will always select f. In the present example, this cannot occur anyway, but it can for other format combinations.) So we test if $C_a \leq m$ and select the provisional output exponent, say f, and then obtain from another table an approximation to $2^k/10^f$, scaled to an integer. The accuracy of these tables was selected (see the discussion below) so that from the product of this multiplier r with C_a we can determine correct rounding in all modes. Rounding may result in an overspill of the output to exactly 10^d where d is the output precision; in this case we use 10^{d-1} instead and increment the provisional exponent (this is why we called it "provisional").

5.2. Conversion from 64-bit Decimal to Double Precision

This conversion is very similar in overall structure to the conversion in the other direction. In some respects it is simpler, because we need no special treatment of exact results and always aim for a normalized output, though we *do* need to handle overflow and underflow. We force normalization of the input in the same 'binary' sense, constraining the input coefficient to a range $2^{\alpha} \leq C_a < 2^{\alpha+1}$ by shifting it appropriately; this is easier than testing for divisibility by 10 and also gives a tighter range. We cannot compensate for this shift l by modifying the *input* exponent (which is decimal), but rather need to incorporate the difference into the *output* exponent by a simple subtraction. The only complication is that now we cannot build the minimal exponent into the tables, because it may be modified by this subtraction. Thus, in the case of underflow we need to manually shift the intermediate result of the product $r \cdot C_a$ right by l bits before deducing the rounded result. (Just shifting the rounded output afterwards might result in incorrect double rounding in rare cases.)

5.3. Required accuracy

Here we explain how bounds were obtained on the required accuracy in the table entries approximating $2^k/10^f$ and $10^f/2^k$, as well as in the calculations involving them. We seek to determine how "hard to round" a conversion can be, i.e. the minimum nonzero relative difference between a floating-point number in the input binary or decimal format and a rounding boundary (floating-point number or midpoint) in the output decimal or binary format. In each case we need to find how small an expression of the following form can become:

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

where the possible e and d are determined by the exponent ranges of the formats, and the possible sizes of the integers m and n by their precision. (When we want to consider midpoints, we simply permit twice the range in that significand.) If we seek a purely analytical answer, we can observe that either the numbers are exactly equal or else:

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

because all quantities are integers. However, this bound is somewhat discouraging, because the exponent may be very large. For example, in the `decimal128` format, we could conceivably have $d = 6111$, meaning that we would need to perform a computation accurate to *thousands* of bits to be sure of rounding correctly. We can get a sharper bound by systematically examining the possible exponent combinations. The relative difference can be written as

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

By slightly expanding the exponent ranges, we can assume without loss of generality that both significands m and n are normalized. This means that there is a tight correlation between the two exponents, so for a given value of e, there are only a few non-trivial cases for d, or vice versa. With $2^e/10^d$ fixed it suffices to find the minimal absolute difference:

$$\left|2^e/10^d - n/m\right|$$

and for each e and d we 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. As we might expect from a naive statistical argument, the bounds determined are much sharper, typically a few bits extra beyond the *sum* of the precisions of the input and output formats. For example, one of the hardest cases for converting from `binary64` to `decimal64` is the following, with a relative distance of $2^{-115.53}$:

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

and a difficult case for conversion from `decimal64` to `binary64` is a relative difference of $2^{-114.62}$ for:

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

6. Performance Data

In this section we present performance data in terms of clock cycle counts needed to execute several library functions. Median and maximum values are shown. 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 as well as 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 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.

Function	EM64t	EM64t	IA-64
Name	Xeon 5100	Xeon	Itanium 2
	3.0 GHz	3.2 GHz	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

Table 1. Clock cycle counts for a subset of *arithmetic* **functions** $\{$**Max / Median Values**$\}$

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 EM64t platform, where the numbers are not too far 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 pre-beta development stage. A small number of optimizations were applied, and significant improvements are still possible.

Function	EM64t	EM64t	IA-64
Name	Xeon 5100	Xeon	Itanium 2
	3.0 GHz	3.2 GHz	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 2. Clock cycle counts for a subset of *conversion* **functions** $\{$**Max / Median Values**$\}$

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.

Function Name	EM64t Xeon 5100 3.0 GHz	EM64t 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
quiet_cmp_eq128	111 / 67	175 / 111	83 / 68
quiet_cmp_eq64	74 / 27	145 / 42	77 / 33
quiet_cmp_gt128	117 / 75	180 / 120	87 / 73
quiet_cmp_gt_eq64	54 / 41	93 / 62	53 / 47
qt_cmp_lt_unord128	118 / 74	182 / 120	88 / 74
qt_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

Table 3. Clock cycle counts for a subset of *miscellaneous* **functions {Max / Median Values}**

Table 3 contains clock cycle counts for *other miscellaneous* IEEE 754R functions. Details on these functions can be found in the latest draft of the revised IEEE standard for floating-point arithmetic [2].

It is interesting to compare these latencies with those for other approaches. For example the decNumber package [12] 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 3.0GHz Xeon 5100 are 133 and 71 clocks cycles respectively. Another example is that based on its median latency, the 64-bit BID add is less than four times slower than of 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.

7. 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 Binary Integer Decimal encoding. We concentrated on the problem or 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 which demonstrate that the possible speedup of hardware implementations over software may not be as dramatic as previously estimated.

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.

References

[1] Institute of Electrical and Electronics Engineers. Standard for Binary Floating-Point Arithmetic, *IEEE Std 754-1985.*

[2] Institute of Electrical and Electronics Engineers. Draft Standard for Floating-Point Arithmetic P754. *http://754r.ucbtest.org/drafts/754r.pdf.* September 2006.

[3] P. Tang, "Binary-Integer Decimal Encoding for Decimal Floating Point" Technical Report. Intel Corporation. Available at *http://754r.ucbtest.org/issues/decimal/bid_rationale.pdf*

[4] M. F. Cowlishaw, "Densely packed decimal encoding." *IEEE Proceedings - Computers and Digital Techniques*, vol. 149, pp. 102-104. May 2002

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

[6] European Commission. "The introduction of the euro and the rounding of currency amounts." Available at *http://europa.eu.int/comm/economy_finance /publications/euro_papers/2001/eup22en.pdf*, March 1998.

[7] IBM Corporation. "The telco bench." Available at *http://www2.hursley.ibm.com/decimal/telco.html*, March 1998.

[8] Standard ECMA-334, "C# language specification." Available at http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf, 2005

[9] "ISO 1989:2002 Programming Languages - COBOL," *ISO Standards, JTC 1/SC 22*, 2002.

[10] W3C, "XML Scheme Part 2: Datatypes Second Edition." Available at *http://www.w3.org/Tr/2004/REC-xmlschema-2-20041028/*, October 2004.

[11] Sun Corp. on-line documentation, “Class BigDecimal” Available at http://java.sun.com/ j2se/1.4.2/docs/api/java/math/BigDecimal.html

[12] M. F. Cowlishaw, “The decNumber library.” *http://www2.hursley.ibm.com/decimal/decnumber/pdf* 2006.

[13] M. H. Weik, “The ENIAC Story.” Available at *http://ftp:.arl.mil/ mike/comphist/eniac-story.html*

[14] G. Gray, “UNIVAC I Instruction Set,” *Unisys History Newsletter*, vol. 2, no. 3, 2001.

[15] M. S. Cohen, T. E. Hull, and V. C. Hamacher, “CADAC: A Controlled-Precision Decimal Arithmetic Unit,” *IEEE Transactions on Computers*, vol. 32, pp. 370-377, April 1983.

[16] F.Y. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz, S.R. Carlough, “The IBM z900 Decimal Arithmetic Unit,” in *Proceedings of the 35th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp 1335, IEEE Computer Society, November 2001.

[17] A.Y. Duale, M.H. Decker, H.-G. Zipperer, M. Aharoni, T.J. Bohizic, “Decimal floating-point in z9: An implementation and testing perspective” in *IBM Journal of Research and Development*, at *http://www.research.ibm.com/journal/rd/511/duale.html*, 2007.

[18] M. A. Erle, J. M. Linebarger, and M. J. Schulte, “Potential Speedup Using Decimal Floating-Point Hardware.” *Proceedings of the Thirty Sixth Asilomar Conference on Signals, Systems, and Computers* Pacific Grove, California. IEEE Press, pp. 1073-1077, November, 2002.

[19] M. A. Erle, M. J. Schulte, “Decimal Multiplication Via Carry-Save Addition.” *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors* The Hague, Netherlands. IEEE Computer Society Press, pp. 348-358, June, 2003.

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

[21] M. F. Cowlishaw. “Decimal Floating-Point : Algorism for Computers,” in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 104-111, June 2003.

[22] L Wang. “Processor Support for Decimal Floating-Point Arithmetic.” Technical Report. Electrical and Computer Engineering Department. University of Wisconsin-Madison. Available upon request.