IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 January 2023, accepted 8 February 2023, date of publication 14 February 2023, date of current version 23 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3244891

RESEARCH ARTICLE

Exact Versus Inexact Decimal Floating-Point Numbers and Arithmetic

MUHAMED F. MUDAWAR
Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
e-mail: mudawar@kfupm.edu.sa

This work was supported by the King Fahd University of Petroleum and Minerals.

ABSTRACT The IEEE 754 standard does not distinguish between exact and inexact floating-point numbers. There is no bit or field in the binary encoding that indicates whether a floating-point number is exact or not. This is the case for binary and decimal floats. An inexact operation raises an inexact flag in a floating-point status register. The inexact result is rounded and used in a later operation as if it were exact. The floating-point arithmetic unit treats all the input operands as if there were exact, and hence might produce substantial errors in the final computed results. This paper focuses on making the distinction between exact and inexact decimal numbers and defines arithmetic operations on both types of numbers. If the result of a sequence of operations is exact, the user can trust that every decimal digit in the computed result is correct. On the other hand, if some input operands are inexact or the result cannot be computed exactly, a loss of significant digits occurs. A different representation is used for the inexact computed value. An estimate of the absolute error is also part of the inexact computed result. The decimal numbers and arithmetic operations introduced in this paper produce more accurate results that those computed by the IEEE 754 standard. A simple evaluation is shown in the last section of this paper.

INDEX TERMS IEEE 754 standard, exact versus inexact decimal numbers, exact versus inexact zeros, exact versus inexact decimal arithmetic.

I. INTRODUCTION

The IEEE 754-2008 standard [1] for floating-point arithmetic extended the original 1985 binary standard [2] by adding decimal (radix-10) floating-point numbers. Decimal numbers are needed because they avoid the rounding errors that typically occur when converting a decimal fraction in human entered data into a binary fraction. For example, the decimal fraction 0.7 becomes 0.699999988, when represented as a 32-bit binary float. The binary fraction must be rounded to the required precision. Decimal numbers are also rounded when a computed result is inexact. However, the Radix-10 rounding rules are more human centric. Decimal numbers are typically used in financial calculations, commercial databases, banking, taxes, and currency conversions [3]. They can also be useful in scientific and engineering applications, but binary numbers are commonly used because of their ubiquitous hardware support.

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

This paper makes the distinction between exact and inexact decimal floating-point numbers. An exact number maps to one discrete value in the infinite continuum of real numbers. It can be represented with zero error. No rounding is done. Given the limited precision p of the significand, only a finite subset of real numbers can be represented exactly as decimal floating-point numbers.

On the other hand, an inexact decimal number cannot be represented exactly with finite precision. Some real numbers, such as π, cannot be represented exactly, and hence must be rounded to the precision of the floating-point representation. An inexact decimal number can also be the result of an inexact operation that requires rounding, even when the operands are exact. There is an error associated with each inexact floating-point operation and result. Given the limited precision of the representation, an inexact decimal number maps to an interval of real numbers. A one-to-infinite relation is defined between an inexact decimal number and the infinite set of real numbers.

A decimal number has a numeric value equal to $\pm C \times 10^q$, where C is an integer coefficient consisting of p decimal

digits and q is a signed exponent. If the leading digit of C is zero then the number is subnormal. Otherwise, it is normalized. The IEEE 754 standard does not require the normalization of decimal numbers and the same decimal value can have multiple representations. Unfortunately, the same decimal representation can be exact or rounded. There is no bit or field that specifies whether a decimal number is rounded. Floating-point operations, whether implemented in hardware or software, treat all operands as if there were exact and hence might produce substantial errors in the final computed results.

As an example, consider adding four *decimal32* numbers in this specific order: $(((W + X) + Y) + Z)$. Each decimal input is represented with a sign bit, an integer coefficient having at most 7 decimal digits, which is the precision of *decimal32*, and a signed exponent. All inputs are exact and chosen to magnify the error: $W = 1{,}234{,}567 \times 10^{-1}$, $X = 8{,}900{,}123 \times 10^{-2}$, $Y = -2{,}124{,}578 \times 10^{-1}$, and $Z = -1001 \times 10^{-4}$.

First, $(W + X)$ is computed. Because of the difference in exponents, the coefficient of X with smaller exponent must be right shifted: $X = 8{,}900{,}123 \times 10^{-2} = 890{,}012.3 \times 10^{-1}$. Then, the coefficients are added and rounded: $(W + X) \approx 2{,}124{,}579 \times 10^{-1}$. The result is inexact, but the relative error is small: $(0.3/2{,}124{,}579.3) \approx 1.412 \times 10^{-7}$.

Next, $(W + X) + Y = 2{,}124{,}579 \times 10^{-1} + -2{,}124{,}578 \times 10^{-1} = 1 \times 10^{-1}$. The operation is exact, but the first input operand and the sum are inexact. This subtraction is described as catastrophic in [4] because it has destroyed six significant digits. The relative error has increased to: $0.3 \times 10^{-1}/1.3 \times 10^{-1} = 23\%$. It should be noted that if both input operands were exact, then the result would have been exact, with zero error, even under digit cancellation. However, the IEEE 754 standard provides no clue about the exactness of the input operands.

Finally, $((W + X) + Y) + Z = 1 \times 10^{-1} + -1001 \times 10^{-4}$. According to IEEE 754, the preferred exponent for decimal addition is $min(EA, EB)$. The coefficient of the decimal number with larger exponent must be left-shifted if it has leading zeros when adding it to a decimal of a lesser exponent. Therefore, 1×10^{-1} becomes 1000×10^{-4} and $1000 \times 10^{-4} + -1001 \times 10^{-4} = -1 \times 10^{-4}$. However, the true result is $1300 \times 10^{-4} + -1001 \times 10^{-4} = 299 \times 10^{-4}$. The overall relative error has exceeded 100% because of the wrongly computed negative sign. Again, the operation itself is exact, but the first input operand and the result are inexact. However, the IEEE 754 standard does not distinguish between an exact operation and an exact result.

To conclude, the relative error of a single inexact operation is typically small. However, for a sequence of operations, the relative error can grow substantially. The above example shows the need to distinguish exact decimal numbers from inexact ones. When an inexact number is propagated in a computation, subsequent operations can result in a large overall error, even when the later operations are exact. Arithmetic on inexact decimal numbers should be done differently.

FIGURE 1. Defining two inexact decimals between every two consecutive exact ones.

A. INEXACT DECIMAL NUMBERS

This paper proposes a new representation for inexact decimal floating-point numbers and a new method for inexact floating-point arithmetic. Given two *consecutive exact* decimal numbers $C \times 10^q$ and $(C + 1) \times 10^q$ with the same exponent q, two *inexact* decimal numbers are defined in between: $C.L \times 10^q$ and $C.H \times 10^q$, as shown in Figure 1. The $.L$ is a *low* fraction in the interval [0, 0.5), while $.H$ is a *high* fraction in [0.5, 1). The $.L$ and $.H$ values are *unknown* but *approximated* in inexact arithmetic computations.

Consider adding the same four decimal numbers given above. First, $(W + X)$ is computed as $1{,}234{,}567 \times 10^{-1} + 8,900,123 \times 10^{-2} = 1{,}234{,}567 \times 10^{-1} + 890{,}012.3 \times 10^{-1} \approx 2{,}124{,}579.\text{L} \times 10^{-1}$. The result of $(W + X)$ is *inexact*. The.L notation represents a *low* fraction, where 0.L < 0.5. The result is *not rounded*. Instead, the .L notation is now part of the inexact result representation.

Next, $(W+X)+Y = 2{,}124{,}579.\text{L} \times 10^{-1} + -2{,}124{,}578 \times 10^{-1} = 1.\text{L} \times 10^{-1}$. The result is *inexact*, even though the operation is exact.

Finally, $((W + X) + Y) + Z = 1.\text{L} \times 10^{-1} + -1001 \times 10^{-4}$. The number $1.\text{L} \times 10^{-1}$ cannot be left shifted because it is inexact. $1.\text{L} \times 10^{-1}$ is *not equal* to $1000.\text{L} \times 10^{-4}$. Therefore, Z must be right-shifted: $Z = -1.001 \times 10^{-1}$. Therefore, $((W + X) + Y) + Z = 1.\text{L} \times 10^{-1} + -1.001 \times 10^{-1} \approx 0.\text{L} \times 10^{-1}$. This result is an *absolute error* with zero significant digits, but consistent with the true sum $= 0.299 \times 10^{-1}$. It provides a feedback to the programmer that the computation should be done differently. In contrast, the IEEE 754 result (-1×10^{-4}) is *unreliable*.

Computing $(W + X) + (Y + Z)$ in a different order to increase instruction-level parallelism produces a different result. According to IEEE 754, $(Y + Z) = -2{,}124{,}578 \times 10^{-1} + -1001 \times 10^{-4} \approx -2{,}124{,}579 \times 10^{-1}$ with rounding. The result R becomes $2{,}124{,}579 \times 10^{-1} + -2{,}124{,}579 \times 10^{-1} = 0$ with a relative error $= 100\%$. Debugging becomes harder.

Using the inexact representation suggested in Figure 1, $(Y + Z) \approx -2{,}124{,}579.\text{L} \times 10^{-1}$ is *inexact* and the sum becomes $2{,}124{,}579.\text{L} \times 10^{-1} + -2{,}124{,}579.\text{L} \times 10^{-1} \approx 0.\text{L} \times 10^{-1}$. This result is identical to the one computed with serial addition, and also consistent with the true result. This improvement rectifies decimal floating-point arithmetic.

It should be noted that IEEE 754 defines *decimal32* as *not basic*. It is used in the example for comparison purposes. Computational errors can occur in floating-point numbers of any size and precision.

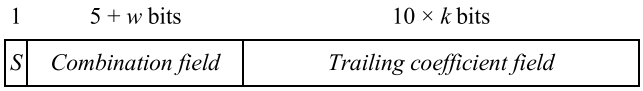

FIGURE 2. IEEE decimal interchange floating-point format.

This paper presents new ideas and contributions to decimal floating-point numbers. It introduces a new representation and encoding of exact and inexact decimal numbers. Inexact decimal numbers cannot be normalized if there is a loss of significant digits. They propagate in computations. This paper also distinguishes between exact zero and inexact ones that represent absolute errors in computations. It defines inexact equality and introduces inexact arithmetic on inexact decimal numbers.

The next section provides an overview and critique of the IEEE 754 decimal floating-point standard, the IEEE 1788 standard for interval arithmetic, and *universal numbers*.

II. IEEE 754, IEEE 1788, AND UNIVERSAL NUMBERS

The IEEE 754 decimal standard was introduced in 2008 and revised in 2019 [5]. It is structured into four levels. The first level defines the mathematical structure as an extended set of real numbers together with positive and negative infinity. Rounding maps an extended real number to a floating-point number. The relationship is many-to-one. The second level defines an algebraic closed system on floating-point data. A floating-point datum can be a signed zero, finite non-zero number, signed infinity, or not-a-number (NaN). The third level defines the representation of floating-point data, and the fourth level defines its binary encoding.

The decimal standard defines interchange formats, called *decimal32*, *decimal64*, and *decimal128* of widths 32, 64, and 128 bits, respectively. The format has three fields: a sign bit S, a *combination* field, and a *trailing coefficient* field, as shown in Figure 2. The combination field has $(5 + w)$ bits that encode the leading digit of the coefficient and the biased exponent E. It was defined this way to optimize the encoding of the leading digit and increase the exponent range. The trailing coefficient field has $10k$ bits (k declets) that encode $3k$ decimal digits. The integer coefficient C has $p = 3k + 1$ decimal digits, where the precision $p = 7$, 16, and 34 for *decimal32*, *decimal64*, and *decimal128*, respectively. The numeric value of a decimal float is: $(-1)^S \times C \times 10^{E-Bias}$.

The decimal standard defines two ways to encode the decimal coefficient. The first encoding scheme, known as Densely Packed Decimal (DPD), uses 10-bit declets to encode three decimal digits efficiently. DPD requires simple logic to unpack/pack the BCD digits at the beginning/end of each operation, as detailed in [6]. Internally, a decimal floating-point unit uses BCD digits in arithmetic operations.

The second encoding scheme uses a binary integer to encode the decimal coefficient. This is known as Binary Integer Decimal or BID encoding. This encoding scheme is simpler than DPD. However, the major difficulty of using the BID encoding is in hardware implementation. For example, to implement a decimal floating-point adder in hardware, left and right shifters are used to align the decimal coefficients. This works well for the DPD encoding that packs BCD digits. However, the BID encoding complicates hardware alignment of the two coefficients and increases its cost. Left and right shifting should be implemented as hardware multipliers by positive and negative powers of 10. However, negative powers of 10, such as 10^{-1} and 10^{-2}, cannot be represented exactly in binary. For this reason, the BID encoding is used mainly for software implementations that take advantage of the binary hardware. A number of software solutions exist. These include Intel Decimal FP library [7], the decNumber C library [8], C# decimal [9], Java BigDecimal [10], and SQL decimal [11]. The drawback of software libraries is speed. Operations implemented in software are reported to run 100× to 1000× slower than those implemented directly in hardware [3].

Unlike a binary floating-point number, a decimal number can have multiple representations. The set of representations a decimal numbers maps to is called the floating-point number's *cohort* [1]. If a non-zero number has n significant decimal digits (starting at its most significant non-zero digit and ending at its least significant non-zero digit) then there are $(p - n + 1)$ representations of the same number, where p is the precision. For example, the number 0.2 has 7 possible representations in *decimal32*: $0.2 = 2 \times 10^{-1} = 20 \times 10^{-2} = \ldots = 2000000 \times 10^{-7}$. In particular, zero has a large cohort: the cohort of ± 0 contains a representation for each exponent [1].

Decimal floating-point units appeared in some major processors, such as IBM Power [12], IBM system z [13], [14], [15], and Fujitsu Sparc64 processors [16]. Wang and Schulte demonstrated the implementation of decimal floating-point square root and divider using Newton-Raphson iteration [17], [18], and decimal adders with injection-based rounding and related operations [19], [20], [21]. Vasquez et al. showed the design of parallel decimal multipliers [22], [23]. Wahba and Fahmy showed the implementation of a combined binary/decimal floating-point fused multiply add unit [24].

A major concern of the IEEE 754 decimal floating-point standard is its inability to distinguish exact decimal numbers from inexact ones (also applicable to binary floating-point). The standard defines an *inexact operation*, when rounding takes place. However, there is nothing in the representation and binary encoding that indicates whether a computed result is inexact. In particular, left-shifting the coefficient of an inexact input operand inserts *incorrect trailing zeros* that increase the error of an operation.

Distinguishing between exact and inexact numbers is essential to improving the quality of floating-point arithmetic. If the result of a sequence of floating-point operations is

exact, then the result is correct with infinite precision. On the other hand, an inexact result will alarm the user about the magnitude of the error in a given computation.

A. INTERVAL ARITHMETIC AND IEEE 1788

Interval arithmetic was developed by mathematicians since the 1960s [25], as an approach to putting bounds on rounding errors and thus developing methods that yield reliable results. Every computation is performed using intervals as inputs and produces intervals as outputs. The computed intervals are *guaranteed* to enclose the exact values of the computation. This is the most precious feature of interval arithmetic, called the *Fundamental Theorem of Interval Arithmetic* (*FTIA*) [26].

The IEEE 1788 standard for interval arithmetic [27] is structured into four levels. The first level is the *mathematical* level that specifies what an interval on real numbers is, and what operations on intervals do. The second level is the *discretization* of intervals that defines the interval *endpoints* and *types*. The third level is about the *representation* of intervals in terms of floating-point numbers, and the fourth level is about the binary encoding.

The standard is also designed to accommodate different models of intervals, called *flavors*, as long as they agree on common definitions. So far, the *set-based flavor* has been adopted. The standard handles exceptions by attaching tags, called *decorations*, to each interval.

The IEEE 1788 standard defines more relational operators than IEEE 754. In addition to testing for *equality*, *less than*, and *less than or equal*, there are also relational operators for intervals: *subset*, *precedes*, *precedes or touches*, and *interior to*. There are also predicates that do not exist in IEEE 754, such as: *before*, *meets*, *overlaps*, and *contains*.

The IEEE 1788 standard was revised in 2017 and simplified to include those operations and features that are most commonly used in practice [28]. An interval is defined as a pair of IEEE 754 *binary64* floating-point numbers and a decoration system for exception-free computations and a propagation of properties of the computed results.

The IEEE 1788 standard is more complex than IEEE 754. There is no hardware implementation. Only software compliant libraries have been developed as a proof of concept. These include the *Octave library* [29] and *libieee1788* [30].

A major drawback of interval arithmetic and IEEE 1788 is the wrapping problem and the dependency problem, that produce large expansions of the resulting intervals, and provide no information about the answer. Another concern is the complexity, cost, and inefficiency of implementation.

B. UNIVERSAL NUMBERS

Universal numbers, called *Unums*, were proposed by Gustafson as an alternative to the IEEE 754 standard. The first version, known as Type 1 Unum [31] is a variable-width storage format for both the significand and exponent. The *esize* and *fsize* fields specify the size of the exponent and fraction. The flexible dynamic range and precision eliminates the pressure for a programmer to choose a ''one size fits all,'' such as the 64-bit binary float. However, the author admits that Type 1 *unums* have many drawbacks, particularly for hardware implementation [32]. They must be unpacked into a fixed storage size. The *esize* and *fsize* fields add a level of indirection that must be read first to reference the other fields. Some values can be expressed in more than one way and some bit patterns are not used.

Type 2 *unums* are a direct map of signed two's complement integers to the projective real number line [32]. It maps reals onto a circle, such that positive and negative infinity meet at the top. The selected set of exact reals between 1 and infinity is user-defined and called the *u*-lattice. However, it requires table look-up because it does not use the traditional radix or positional representation. The reciprocal of a Type 2 *unum* can be obtained easily by keeping the sign bit and obtaining the two's complement of the remaining bits. Division becomes as fast as multiplication. The reciprocal of *zero* is *infinity* and vice versa, so no exception is needed. However, Type 2 *unums* are difficult to extend to high precision and require large table lookup (that grow exponentially according to precision) for basic arithmetic operations, such as addition, subtraction, and multiplication. On the other hand, floating-point numbers are easy to work with by algorithmic methods that can be implemented easily in hardware.

The latest version of *unums* is Type 3, called *Posits* [33]. Posits are similar to binary floats but offer better precision in the range near one. They have a tapered floating-point format, which is slightly more complex than IEEE 754, and consists of four fields: sign bit, regime, exponent, and fraction. Posits are rounded like binary floats. There is no distinction between exact and rounded posits and no representation of exact decimal fractions: $0.1 + 0.2 \neq 0.3$. Posits were proven useful in machine learning, where small 8-bit or 16-bit posits are used. There is a growing interest in posits. Crespo et al. implemented a unified Posit/IEEE 754 vector MAC unit [40] and Mathis and Stine implemented a high performance IEEE 754-Posit conversion hardware [41].

However, there are also situations where posits are worse than floating-point, such as particle physics simulations [34]. Multiplying a posit by a power of two is not always exact as in binary floats, the rounding error in the product of two posits is not always a posit, and posits can get ugly in multiplicative cancellation, as explained in [34].

III. EXACT VERSUS INEXACT DECIMAL NUMBERS

This paper focuses on making the distinction between exact and inexact decimal floating-point numbers, and describes arithmetic on both types of numbers, something not done in IEEE 754. The focus will be on decimal floats because of their ability to represent exact decimal fractions.

An *exact* decimal floating-point number represents a single discrete value in the infinite continuum of real numbers. It can be represented with *zero error*. An *exact* decimal float *should be normalized* to have a unique representation. This is a requirement in my work, not in IEEE 754. The leading

digit d of the coefficient cannot be zero, except when the number is zero. For example, the exact decimal number 0.2 is represented uniquely as $2{,}000{,}000 \times 10^{-7}$ with $p = 7$ decimal digits. The rationale is to force a unique representation on all exact decimal floats. The concept of cohorts is eliminated.

Converting an exact *decimal32* number into *decimal64* is done by appending *trailing* zeros into the significand and adjusting the exponent accordingly. For example, $0.2 = 2{,}000{,}000 \times 10^{-7}$ becomes $2{,}000{,}000{,}000{,}000{,}000 \times 10^{-16}$ with 16 decimal digits when converted into *decimal64*. However, converting an exact *decimal64* number into *decimal32* might produce an *inexact* number, if one of the nine trailing decimal digits that are shifted-out from the significand is non-zero.

An *inexact* decimal number cannot be represented exactly with finite precision. For example, an *inexact decimal32* representation of π is $3{,}141{,}592.\text{H} \times 10^{-6}$ (with $p = 7$), where 0.H represents a *high* fraction ($0.5 \leq 0.\text{H} < 1$). The absolute error is $0.\text{H} \times 10^{-6}$. An *inexact decimal64* representation of π is $3{,}141{,}592{,}653{,}589{,}793.\text{L} \times 10^{-15}$ (with $p = 16$), where 0.L represents a *low* fraction ($0 \leq 0.\text{L} < 0.5$). The absolute error is $0.\text{L} \times 10^{-15}$.

Converting an inexact number, such as π, from *decimal32 to decimal64* does not increase its precision. *Leading zeros* are inserted: $\pi = 0{,}000{,}000{,}003{,}141{,}592.\text{H} \times 10^{-6}$. If an inexact number is not normalized, it *cannot* be left-shifted and normalized because the trailing digits are *unknown*. In summary, *inexact* decimal numbers may or may not be normalized with a unique representation. They have .L or.H representations that indicate *low* or *high* fraction intervals: $0.\text{L} = [0, 0.5)$ and $0.\text{H} = [0.5, 1)$, as shown in Figure 1.

An inexact number can be the result of an operation with exact or inexact operands. However, rounding is not used. An inexact number is an *interval* in the infinite continuum of real numbers. However, interval arithmetic is not used. Instead, *inexact* arithmetic on inexact decimal numbers is defined in this paper.

A. EXACT VERSUS INEXACT ZEROS

The IEEE 754 decimal standard defines only exact zero as a large cohort that has a zero significand and an arbitrary value of the exponent field: $zero = \pm 0 \times 10^{q}$ for any exponent value q. There is no unique representation of exact zero and no definition of inexact zero.

In contrast, this paper distinguishes between exact zero and inexact ones. Exact zero has a unique representation (all bits are zeros). It is written as 0 with *no sign bit*. However, *inexact zeros* are many, signed, and written as: $\pm 0.\text{L} \times 10^{q}$ or $\pm 0.\text{H} \times 10^{q}$. They represent errors in computations. The significand is 0.L or 0.H. However, the exponent q indicates the scale of the error.

B. FORMAT

This paper suggests a new format for exact and inexact decimal numbers, as shown in Figure 3. It consists of four fields: a *sign bit* S, a 5-bit *digit field* D that encodes the

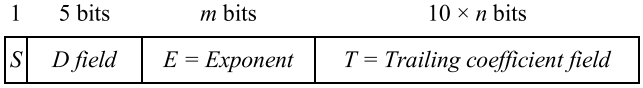

FIGURE 3. New decimal interchange floating-point format.

TABLE 1. Field lengths, exact, and inexact decimal values.

Name	DFP32	DFP64	DFP128
Format length	32 bits	64 bits	128 bits
Sign	1 bit	1 bit	1 bit
Digit field	5 bits	5 bits	5 bits
Exponent field	6 bits	8 bits	12 bits
Trailing field	20 bits	50 bits	110 bits
Precision	7 digits	16 digits	34 digits
Biased exponent	$E = 0$ to 63	$E = 0$ to 255	$E = 0$ to 4095
Bias	38	143	2081
Exact value	$\pm C \times 10^{E-38}$	$\pm C \times 10^{E-143}$	$\pm C \times 10^{E-2081}$
Inexact values	$\pm C.L \times 10^{E-38}$ $\pm C.H \times 10^{E-38}$	$\pm C.L \times 10^{E-143}$ $\pm C.H \times 10^{E-143}$	$\pm C.L \times 10^{E-2081}$ $\pm C.H \times 10^{E-2081}$

leading decimal digit of the coefficient and specifies whether the number is exact or not, a *biased exponent field* E, and a *trailing coefficient field* T having $10 \times n$ bits (n declets) that encode the trailing $3 \times n$ decimal digits of the coefficient.

Table 1 defines the fields, bit-length, and important values of the newly proposed decimal floats, named *DFP32*, *DFP64*, and *DFP128*, of length 32, 64, and 128 bits, respectively. The exponent field consists of m bits. The biased exponent range for a finite decimal number is $E = 0$ to $2^m - 1$ and the *Bias* is $2^{m-1} + p - 1$, where p is the precision ($p = 7$, 16, and 34 for *DFP32*, *DFP64*, and *PDF128* respectively).

The significand of a decimal number is an integer coefficient C, which is the concatenation of the leading decimal digit in the D field and the $3n$ decimal digits in T. The value of an *exact* decimal number is $\pm C \times 10^{q}$. The value of an *inexact* decimal number can be either: $\pm C.L \times 10^{q}$ or $\pm C.H \times 10^{q}$, where $q = E - Bias$. The exponent range defined in Table 1 for *DFP32*, *DFP64*, and *DFP128* is smaller than that defined in the IEEE 754 standard, but sufficiently large for applications. However, the precision p is the same.

C. LEADING DIGIT AND TRAILING COEFFICIENT

The 5-bit D field indicates whether a decimal number is exact or not and encodes the 4-bit leading digit d of the integer coefficient C. The encoding is shown in Table 2. If $D = E = 0$ then the number is either an *exact zero* or *exceptional* value. If D is 1 to 7, 24, or 25 then the decimal number is *exact*, the leading digit $d = 1$ to 9, and the integer coefficient C is normalized. If D is 8 to 15, 26, or 27 then the decimal number is *inexact*, and its coefficient is extended with a *low*

TABLE 2. 5-bit encoding of the *D* field.

5-bit *D* field $D = a\,b\,c\,d\,e$	Type	4-bit leading digit $d = w\,x\,y\,z$
$D = 0$ and $E = 0$	*Zero* or *Special*	$d = 0$
$D = 0$ and $E \neq 0$	*Reserved*	$d =$ ----
$D = 00cde = 1$ to 7	*Exact*	$d = 0cde = 1$ to 7
$D = 01cde = 8$ to 15	*Inexact* **.L**	$d = 0cde = 0$ to 7
$D = 10cde = 16$ to 23	*Inexact* **.H**	$d = 0cde = 0$ to 7
$D = 1100e = 24$ or 25	*Exact*	$d = 100e = 8$ or 9
$D = 1101e = 26$ or 27	*Inexact* **.L**	$d = 100e = 8$ or 9
$D = 1110e = 28$ or 29	*Inexact* **.H**	$d = 100e = 8$ or 9
$D = 1111e = 30$ or 31	*Reserved*	$d =$ ----

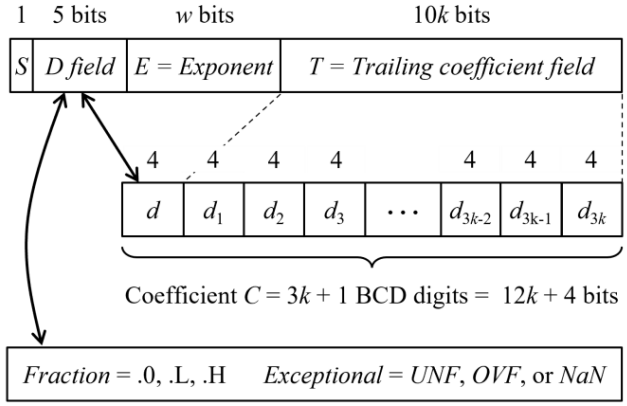

FIGURE 4. Coefficient, fraction, and exceptional values.

fraction. If D is 16 to 23, 28, or 29 then the decimal number is also *inexact*, but extended with a *high fraction*. The leading digit d for an inexact decimal number is 0 to 9 and its integer coefficient should not be normalized.

Decoding the 5-bit D field $= abcde$ into a 4-bit leading decimal digit $d = wxyz$ is simple: $d = 0cde$ if D is 0 to 23, and $d = 100e$ if D is 24 to 29. The logic expressions are: $w = a\&b$, $x = c\&\sim w$, $y = d\&\sim w$, and $z = e$.

The integer coefficient C is the concatenation of the leading digit d and the trailing coefficient T, as shown in Figure 4. The T field is encoded using densely packed decimal (DPD). Unpacking and packing the 10-bit declets into three BCD digits has a low cost and uses simple equations [6].

If $D = 0$ and $E \neq 0$, or $D = 30$ or 31, then the leading digit d is not defined in Table 2. One option is to have d equal to decimal *ten*. The maximum coefficient C becomes $1099\ldots9$. The decimal number is exact if $D = 0$ and inexact if $D = 30$ or 31. A second option is to have d equal to 0 and use an extended exponent range. The coefficient C becomes the trailing field T with a loss of one significant digit d, but the exponent range is increased.

TABLE 3. Encoding the exceptional values.

S bit	*D*, *E* fields	*T* field	Value
-	$D = E = 0$	$T = 0$	*Exact Zero*
0 or 1	$D = E = 0$	$T = 1$	*Underflow* (±*UNF*)
0 or 1	$D = E = 0$	$T = 2$	*Overflow* (±*OVF*)
-	$D = E = 0$	$T = 4$	*Not-a-Number* (*NaN*)

D. EXCEPTIONAL VALUES

The IEEE 754 standard defines five exceptions that can be caught in a given computation. These are: *invalid operation*, *division by zero*, *overflow*, *underflow*, and *inexact*. An exception is signaled by either setting a flag in a floating-point status register or taking a trap. Computational operations produce floating-point results that might signal the floating-point exceptions. According to IEEE 754, an inexact operation produces a *rounded* result. In my work, an inexact operation produces an *inexact* result, which can be $\pm C.L \times 10^q$ or $\pm C.H \times 10^q$. No rounding is done, and no hardware flag is needed.

The hardware flags in the floating-point status register can be replaced with exceptional values encoded in the binary representation. Three exceptional values are defined in this paper. *Overflow* is a *signed* decimal float with a large exponent that cannot be represented. For any finite decimal float x that can be represented, $-OVF < x < +OVF$.

The reciprocal of *Overflow* is *Underflow* (*UNF*) and vice-versa. IEEE 754 uses denormalized numbers to provide gradual underflow to exact zero, but this paper defines *UNF* as an exceptional value, which is different from zero. For any positive finite decimal number x, $-OVF < -x < -UNF < 0 < +UNF < x < +OVF$. It should be noted that the product $OVF \times UNF$ is *indeterminate*, or *NaN*.

Not-a-Number (*NaN*) is the third exceptional value. It can be *indeterminate* such as dividing *zero* by *zero*, or *Not-a-Real* such as the square-root of a negative number. The *sign* of *NaN* is unknown and *NaN* values are *unordered*.

Exceptional values are encoded using the T field when D and E are both 0, as shown in Table 3. If $D = E = T = 0$ then the value is *exact zero* and the sign bit is ignored. If $D = E = 0$ and $T \neq 0$, the exceptional values are $\pm UNF$, $\pm OVF$, and *NaN*, respectively. The *Zero* and *NaN* values ignore the sign bit. It is also possible to encode different *NaN* exceptional values, such as *indeterminate* and *Not-a-Real*.

IV. DECIMAL ADDITION AND SUBTRACTION

The addition and subtraction of *exact* decimal floating-point numbers is well-defined. If the exponents are different, the coefficient of the number with *lesser exponent* must be *shifted-right* to increase its exponent. Since exact decimal numbers are *normalized* in my work, there is no counting of the leading zeros and no left-shifting of a source operand to decrease its exponent. The sum or difference is then

TABLE 4. Inexact addition to ±0.L and ±0.H.

+	+0.L	+0.H	−0.L	−0.H
+0.L	+0.L	+0.H	+0.L	−0.H
+0.H	+0.H	+1.L	+0.H	+0.L
−0.L	−0.L	+0.H	−0.L	−0.H
−0.H	−0.H	−0.L	−0.H	−1.L

normalized. The result of addition and subtraction can be inexact, even when the operands are exact. This occurs when the fractional part of the normalized significand is *not zero*. However, there is no rounding. If the result significand is inexact, it is represented with the.L or .H fraction according to the fraction that appears after the decimal point.

On the other hand, the addition and subtraction of *inexact* decimal floating-point numbers is more intricate. The question is how to define arithmetic on 0.L and 0.H. One choice is to use interval arithmetic. For example, (0.L + 0.L) can be 0.L or 0.H, (0.L + 0.H) can be 0.H or 1.L, and (0.H + 0.H) can be 1.L or 1.H. Similarly, (0.L − 0.L) and (0.H − 0.H) can be ±0.L, (0.H − 0.L) can be 0.H or 0.L, and (0.L − 0.H) can be −0.H or −0.L. The drawback of interval arithmetic is that it requires two endpoints to represent the result. The intervals become larger and more complex over a sequence of operations, which complicates implementation.

This paper suggests a simple approach to handle arithmetic on inexact decimal floating-point numbers. The arithmetic is *inexact* but produces more reliable results than those obtained according to IEEE 754. It clearly indicates that the result is inexact and does not complicate the hardware implementation of the decimal floating-point unit.

Inexact arithmetic uses a *single-digit approximation* of 0.L and 0.H. The choice is to have 0.L ≈ 0.2 and 0.H ≈ 0.7. The rationale is that 2 is the median of 0 to 4, and 7 is the median of 5 to 9. The difference between 0.L and 0.H is 0.5. Similarly, the difference between 0.H and 1.L is also 0.5.

Inexact addition to ±0.L and ±0.H is defined in Table 4. For example, 0.L + 0.L ≈ 0.2 + 0.2 ≈ 0.L, 0.L + 0.H ≈ 0.9 ≈ 0.H, and 0.H + 0.H ≈ 1.4 ≈ 1.L (not 1.H). Similarly, (0.L − 0.L) and (0.H − 0.H) are defined to be + 0.L. However, −0.L + 0.L becomes −(0.L − 0.L) ≈ −0.L (not + 0.L). The remaining entries in Table 4 are derived consistently.

Adding and subtracting a shifted coefficient uses the same digit approximation. For example, 0.L + 0.3 ≈ 0.2 + 0.3 ≈ 0.H, 0.L + 0.8 ≈ 1.L, and 0.H + 0.8 ≈ 1.5 ≈ 1.H. Similarly, 0.L − 0.3 ≈ 0.2 − 0.3 ≈ −0.L, 0.L − 0.8 ≈ −0.H, and 0.H − 0.8 ≈ −0.L.

Figure 5 defines an algorithm for adding and subtracting two decimal numbers x and y. The algorithm can be implemented in hardware or software. The first step extracts all the fields of x and y according to the format of Figure 4, injects a digit for the fraction F (.0, .2, or .7) and decodes the exceptional values. If an input x or y is *zero* then its sign bit Sx or Sy is cleared to avoid having a negative zero result.

Step 2 compares the biased exponents Ex and Ey, computes their absolute difference d and their maximum Eu. Step 3 swaps the input operands if $Ex < Ey$, and produces the swapped significands $\{Su, Cu, Fu\}$ and $\{Cv, Fv\}$.

Step 4 determines the effective operation EOP according to the sign bits Sx and Sy, and the input operation Op, where ADD is 0 and SUB is 1.

Step 5 saves the guard digit (if needed) for subtraction if there is a difference in the exponents (d != 0) and the first swapped operand is exact ($Fu == 0$). The new coefficient Cu is *shifted-left* one decimal digit to save the guard digit. The maximum exponent Eu and the exponent difference d are also decremented. In addition, Step 5 *shifts-right* the significand $\{Cv, Fv\}$ to produce $\{Cw, Fw, Inx\}$ according to the exponent difference d. It aligns the significands to have a common exponent Eu. The Inx (inexact) flag is set if any shifted-out (discarded) fraction digit is non-zero.

Step 6 does the BCD addition or subtraction of the aligned significands $\{Cu, Fu\}$ and $\{Cw, Fw\}$. It convert subtraction into addition to the BCD (10's) complement, uses a BCD adder to compute the *magnitude* of the result significand $\{Carry, Csum, Fsum\}$, and computes LT that indicates whether $\{Cu, Fu\}$ is less than $\{Cw, Fw\}$. If $LT == 1$ for subtraction, then this step post-corrects the *magnitude* of the result $\{Cr, Fr\}$ by computing its BCD complement. This is essentially needed for hardware implementation. However, software implementation can use the Binary Integer Decimal (BID) encoding, which takes advantage of the binary hardware to compute the sum or difference. Step 6 also computes the result sign $Sr = Su \wedge LT$, which complements the sign bit Su if LT is 1 for subtraction.

Step 7 normalizes the result significand $\{Cr, Fr\}$ computed in step 6 and adjusts the common exponent Eu. If the result significand $\{Cr, Fr\}$ is *exact* and with leading zero digits, it is *shifted-left* according to the count LZ of leading zeros in Cr and the exponent Eu is reduced. This is necessary to produce a unique representation of the result. If the result has an extra $Carry$ digit, then $\{Carry, Cr, Fr\}$ is *shifted-right* one BCD digit, and the exponent Eu is incremented. Step 7 produces a normalized output $\{En, Cn, Fn\}$. If the result is *exact zero* then En is reduced to 0.

It should be noted that the normalization Step 7 is adaptive and distinguishes between exact and inexact results. This makes it different from *significance arithmetic* that does not make this distinction. There are no rounding modes and no rounding step, which simplifies implementation.

Step 8 handles the exceptional inputs NaN, OVF, and UNF, detects overflow and underflow, and produces an exceptional output. Step 9 encodes and packs the final normalized result.

As an example, consider adding $x = -6254763 \times 10^{-5}$ and $y = -9877012 \times 10^{-4}$, which are exact DFP32 input operands. Because $Ex < Ex$ then the input operands must be *swapped*. $\{Cu, Fu\} = \{9877012, 0\}$, $\{Cv, Fv\} = \{6254763, 0\}$, and the sign $Su = Sy = 1$. The maximum

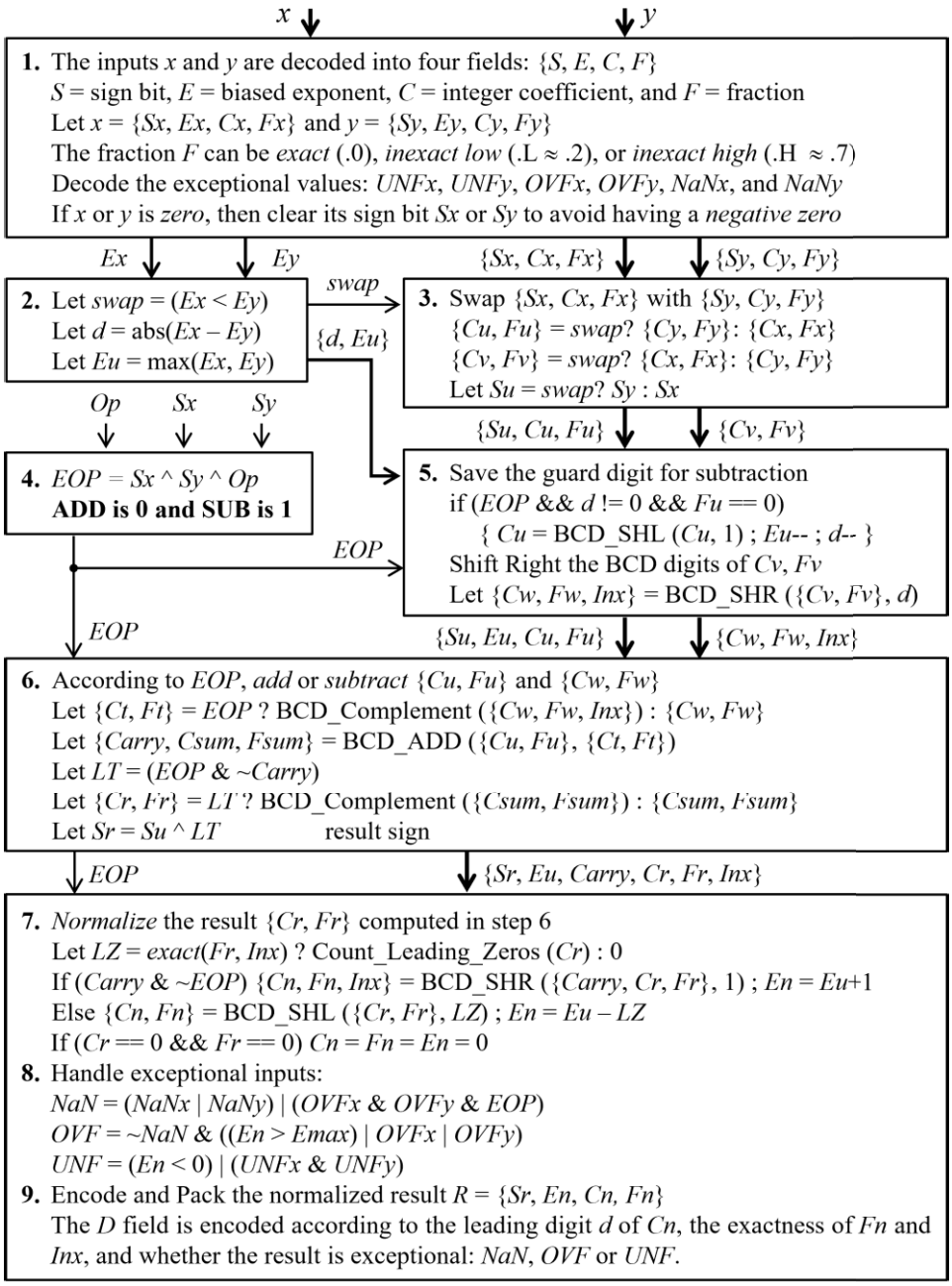

FIGURE 5. Adding and subtracting two decimal numbers x and y.

exponent is $Eu = -4 +$ bias, and the effective operation $EOP = 0$, which is addition. Because of the difference in exponents, $\{Cv, Fv\}$ must be shifted-right one BCD digit to become: $\{Cw, Fw\} = \{0625476, 3\}$ and $Inx = 1$. The significands are then added to become $\{Carry, Csum, Fsum\} = \{1, 0502488, 3\}$. The result sign is $Sr = Su = 1$. Because there is a *Carry*, the significand is normalized to become $\{Cn, Fn\} = \{1050248, 8\}$ and the exponent is incremented to become $En = Eu + 1 = -3 +$ bias. The inexact result is $R = -1050248.\text{H} \times 10^{-3}$, which is encoded according to Table 2 and Figure 4.

Consider now subtracting $x = +1000234 \times 10^{-1}$ and $y = +9876543 \times 10^{-2}$. Because $Ex > Ey$, $\{Cu, Fu\} = \{1000234, 0\}$ and $\{Cv, Fv\} = \{9876540, 0\}$. The sign $Su = Sx = 0$, the exponent $Eu = -1 +$ bias, and $EOP = 1$, which is subtraction. To save the *guard* digit, the Cu coefficient is *shifted-left* one decimal digit to become $Cu = 10002340$, the Eu exponent is decremented to become $Eu = -2 +$ bias, and

the exponent difference is decremented to become $d = 0$. This is done only for subtraction if there is a difference in exponents and the number with larger exponent is exact. The two significands are already aligned and $\{Cw, Fw, Inx\} = \{9876543, 0, 0\}$. Subtraction is then converted into addition to the 10's complement and $\{Ct, Ft\} = \{90123457, 0\}$. The leftmost digit 9 is inserted into Ct to extend the coefficient and obtain the correct sign of the result. The significands are then added to become $\{Carry, Csum, Fsum\} = \{10002340, 0\} + \{90123457, 0\} = \{0, 0125797, 0\}$. The $Carry$ digit is 0. It indicates that the result is positive ($LT = 0$). If the $Carry$ digit were 9, the result would have been negative ($LT = 1$), and the 10's complement of the $\{Csum, Fsum\}$ would have been required to post-correct the result significand. Because the computed $Csum$ has a leading zero and the fraction is exact, it is shifted-left and normalized to become $\{Cn, Fn\} = \{1257970, 0\}$. The exponent is decremented, and the final result becomes $R = +1257970 \times 10^{-3}$.

V. DECIMAL COMPARISON

According to IEEE 754, all floating-point numbers are ordered, except for *NaN*. Given two floating-point numbers, there are four mutually exclusive relations: *equality* (EQ), *less than* (LT), *greater than* (GT), or *unordered* (UN). Two rounded numbers can be equal even when they represent different real numbers.

In my work, equality has two meanings. It can be exact or inexact. Two finite decimal numbers x and y are *equal* (EQ) if they are *both exact* and have *identical binary encoding* because they are normalized with a unique representation. If x and y are equal, then their difference $(x - y)$ must be *zero*. On the other hand, *inexact* or *approximate equality* (AE) is used to compare two inexact, or an exact with an inexact decimal value.

Figure 6 defines an algorithm for comparing two decimal numbers x and y. There are five mutually exclusive relations: *equality* (EQ), *approximate equality* (AE), *less than* (LT), *greater than* (GT), or *unordered* (UN). Exact decimal floats are strictly ordered. For any positive exact decimal number x, $-OVF < -x < -UNF < 0 < +UNF < +x < +OVF$.

Two finite decimal numbers x and y of the same sign are approximately equal, when at least one of them is inexact and their *aligned* significands are equal up to a single digit approximation: $x \approx y \approx C.L \times 10^q$, or $x \approx y \approx C.H \times 10^q$. However, $C \times 10^q < C.L \times 10^q < C.H \times 10^q$. Similarly, exact zero is *less than* an inexact zero.

For example, given $x = 314.\text{L} \times 10^{-2}$ and $y = 31415.\text{H} \times 10^{-4}$ are two approximations of π with different exponents then x and y must be aligned. The significand of y with lesser exponent is shifted right: $y = 31415.\text{H} \times 10^{-4} \approx 314.\text{L} \times 10^{-2}$, indicating approximate equality of x and y.

Similarly, if $z = 3141000 \times 10^{-6}$ is an exact decimal number, then $z = 314.1 \times 10^{-2} \approx 314.\text{L} \times 10^{-2}$ is approximately equal to x. However, $z = 31410.0 \times 10^{-4}$ is *less than* $y = 31415.\text{H} \times 10^{-4}$. This example shows that *approximate equality is not transitive*, while *exact equality is transitive*.

NaN values are *unordered* and cannot be compared with any decimal number x. Similarly, two *OVF* (or two *UNF*) values of the same sign are *unordered*. However, the *UNF* and *OVF* values are ordered with respect to a finite decimal number x.

There are also Boolean functions that test the value of a decimal number: *isUNF*, *isOVF*, *isNaN*, *isExact*, *isZero*, and *isInexactZero*. For example, the function *isExact*(x) returns *true* if the operand x is exact. In particular, *UNF*, *OVF*, and *NaN* are inexact.

Many programming languages use the == operator for testing equality. There is no operator for testing approximate equality. In my work, the expression (x == y) is used to test exact equality, while the Boolean function $AE(x, y)$ tests approximate equality.

VI. DECIMAL MULTIPLICATION

Unlike addition and subtraction, decimal multiplication does not require the alignment of significands when the exponents are different. Multiplying two exact decimal numbers is simple. The decimal coefficients are multiplied, and the exponents are added. The result coefficient is then normalized. If one of the shifted-out digits is non-zero, the result becomes inexact. The last shifted-out digit indicates whether the fraction is 0.L or 0.H.

Multiplying two inexact decimal numbers (or an exact with an inexact decimal number) is more intricate, because the error gets amplified when multiplying an integer coefficient by 0.L or 0.H. As in addition and subtraction, digit injection is used to approximate 0.L and 0.H. Different results are computed if different approximations of 0.L and 0.H are used, as shown in the following example:

$$
\begin{aligned}
&987.\text{L} \times 10^{-3} \times 6543.\text{H} \times 10^{+2} \\
&987.0 \times 10^{-3} \times 6543.5 \times 10^{+2} = 6,458,434.5 \times 10^{-1} \\
&\quad \approx 645.\text{H} \times 10^{+3} \\
&987.2 \times 10^{-3} \times 6543.7 \times 10^{+2} = 6,459,940.64 \times 10^{-1} \\
&\quad \approx 645.\text{H} \times 10^{+3} \\
&987.4 \times 10^{-3} \times 6543.9 \times 10^{+2} = 6,461,446.86 \times 10^{-1} \\
&\quad \approx 646.\text{L} \times 10^{+3}
\end{aligned}
$$

In the above example, $987.\text{L} \times 10^{-3}$ and $6543.\text{H} \times 10^{+2}$ have coefficients with 3 and 4 significant digits, respectively. The product coefficient has more than 7 digits using different approximations of 0.L and 0.H. The most-significant 3 digits of the product are meaningful, while the remaining digits are wrong and should be discarded. Therefore, the product coefficient must be shifted-right and the exponent must be incremented accordingly. In general, given two decimal numbers x and y having coefficients with m and n significant digits, the product coefficient should be restricted to have only $r = \min(m, n)$ digits. The remaining digits are meaningless and should be shifted-out.

Figure 7 defines an algorithm for multiplying two decimal numbers x and y, which can be implemented in hardware

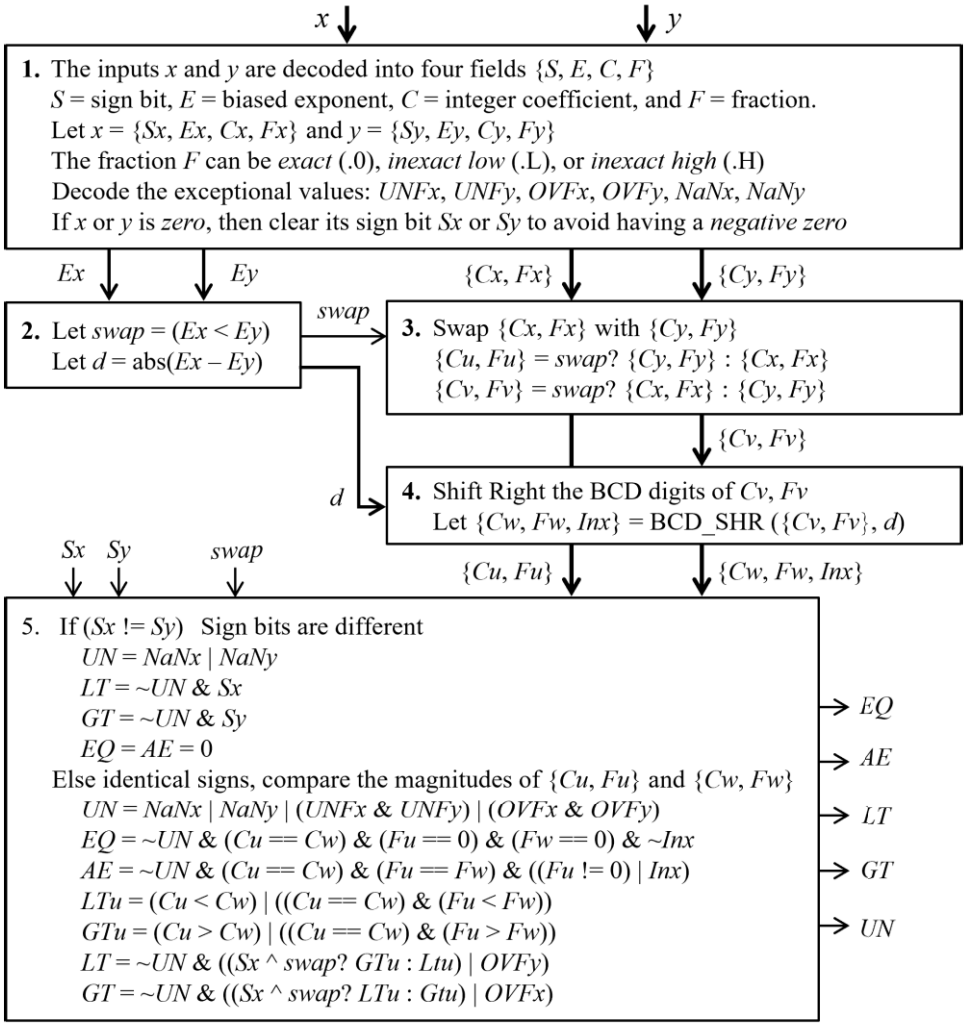

FIGURE 6. Comparing two decimal numbers x and y.

or software. Step 1 extracts all the fields of x and y. Step 2 injects Fx and Fy into Cx and Cy to produce Cu and Cv. It also counts the maximum leading zeros LZ in the coefficients Cx and Cy when an input is inexact to determine the precision of the result. Step 3 computes the product sign Sr and the biased exponent Ep of the product $Cp = Cu \times Cv$. Step 4 computes the product Cp. Step 5 computes LZp, which is the count of leading zeros in Cp. It also shifts-right the product Cp to produce a result coefficient Cr restricted to the minimum number of significant digits in Cx and Cy. The result is a shifted significand $\{Cr, Fr\}$ and an inexact flag Inx that indicates whether any shifted-out digit is non-zero. Step 6 handles exceptional inputs and produces exceptional results. Step 7 encodes and packs the result R.

Given $x = -0017652.\text{H} \times 10^{-2}$ and $y = +0145678.\text{L} \times 10^{-3}$ then $Cu = \{0017652, 7\}$ and $Cv = \{0145678, 2\}$. The maximum leading zeros is $LZ = 2$. The result sign is $Sr = 1$ (negative), the product $Cp = 0{,}000{,}257{,}161{,}356{,}114$, and its exponent is $Ep = -7+$ bias. The count of leading zeros in Cp is $LZp = 4$ and the shift amount $SA = 7$. Cp is shifted-right 7 digits to produce $\{Cr, Fr, Inx\} = \{0025716, 1, 1\}$ and the result exponent is incremented to become $Er = 0+$ bias. The final result is $R = -0025716.\text{L} \times 10^{0}$.

VII. DECIMAL DIVISION

Given two finite decimal floating-point numbers x and y, the significand of x is divided by the significand of y, and the exponents are subtracted. The result is then normalized to the required precision. Similar to multiplication, the result coefficient should be restricted to $r = \min(m, n)$ significant digits, where m and n are the number of significant digits in Cx and Cy. The divisor becomes $\{Cy, Fy\}$ after injecting Fy, whereas the dividend becomes $\{Cx, Fx, 0 \ldots 0\}$ after injecting Fx followed by $n + 1$ decimal zeros. Dividing an integer having $(m + n + 2)$ decimal digits by a divisor having $(n + 1)$ digits produces a quotient having either $(m + 1)$ or $(m + 2)$ digits.

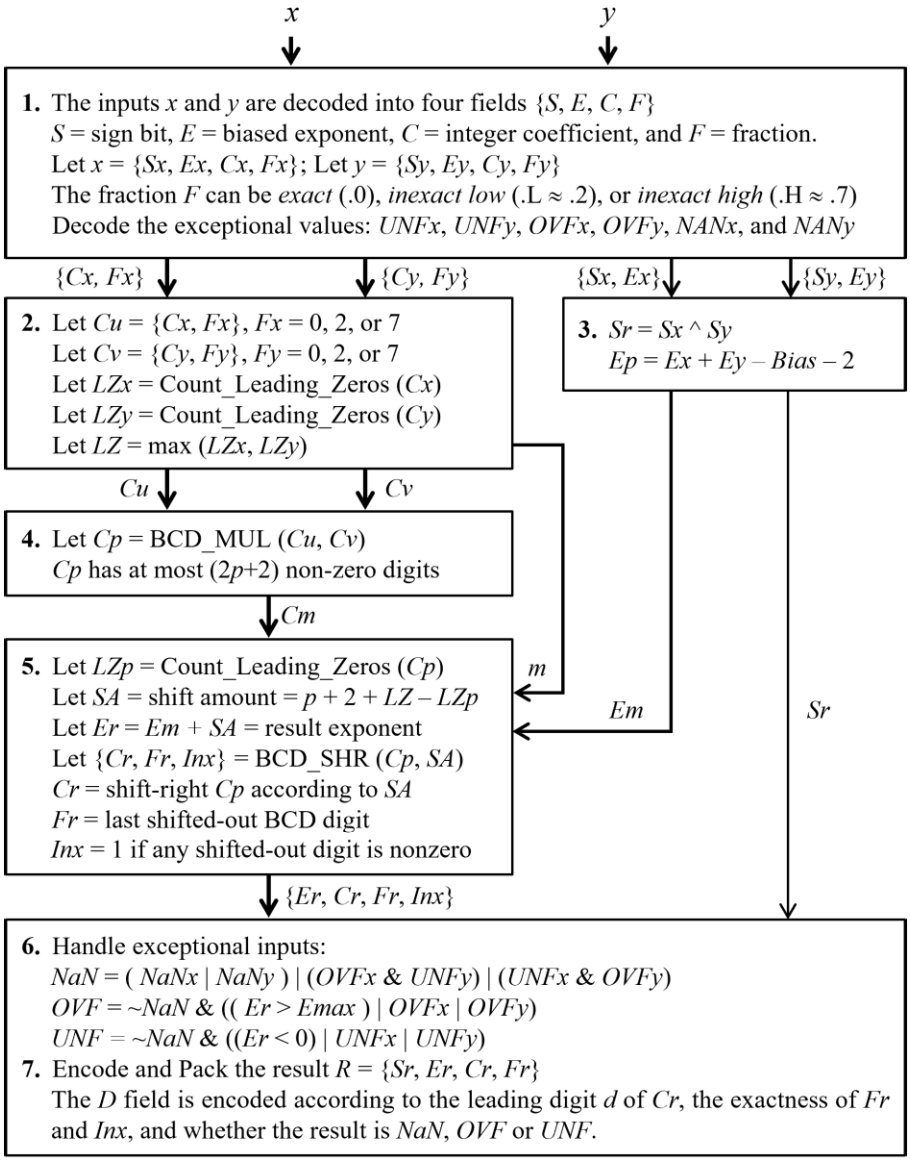

FIGURE 7. Multiplying two decimal numbers x and y.

The following example shows the division of two inexact decimal numbers that use different approximations of 0.L and 0.H:

$$123.\text{L} \times 10^{-1} / 45678.\text{H} \times 10^{-2}$$
$$1230000000 \times 10^{-8} / 456785 \times 10^{-3} \approx 2692 \times 10^{-5} \approx 269.\text{L} \times 10^{-4}$$
$$1232000000 \times 10^{-8} / 456787 \times 10^{-3} \approx 2697 \times 10^{-5} \approx 269.\text{H} \times 10^{-4}$$
$$1234000000 \times 10^{-8} / 456789 \times 10^{-3} \approx 2701 \times 10^{-5} \approx 270.\text{L} \times 10^{-4}$$

Figure 8 defines an algorithm for dividing two decimal numbers x and y. Step 2 injects Fx (0, 2, or 7) and $(p + 1)$ decimal zeros into Cx to produce a coefficient Cu having $(2p + 2)$ decimal digits, where p is the precision. It also injects Fy (0, 2, or 7) into Cy to produce a coefficient Cv having $(p + 1)$ decimal digits. This step also counts the maximum number of leading zeros in both coefficients Cx and Cy: $LZ = \max(LZx, LZy)$. This is needed when an input operand is inexact to determine the precision of the result.

Step 3 computes the sign of the result $Sr = Sx \wedge Sy$, where $\wedge$ is the XOR operator. It also computes the biased exponent of the quotient: $Eq = Ex - Ey - p - 1 + Bias$.

Step 4 divides the decimal coefficients: $Cq = Cu/Cv$. This step produces a quotient Cq having at most $(2p + 2)$ decimal digits, and an Inx flag that indicates whether division is *inexact* (Inx can be 0 or 1).

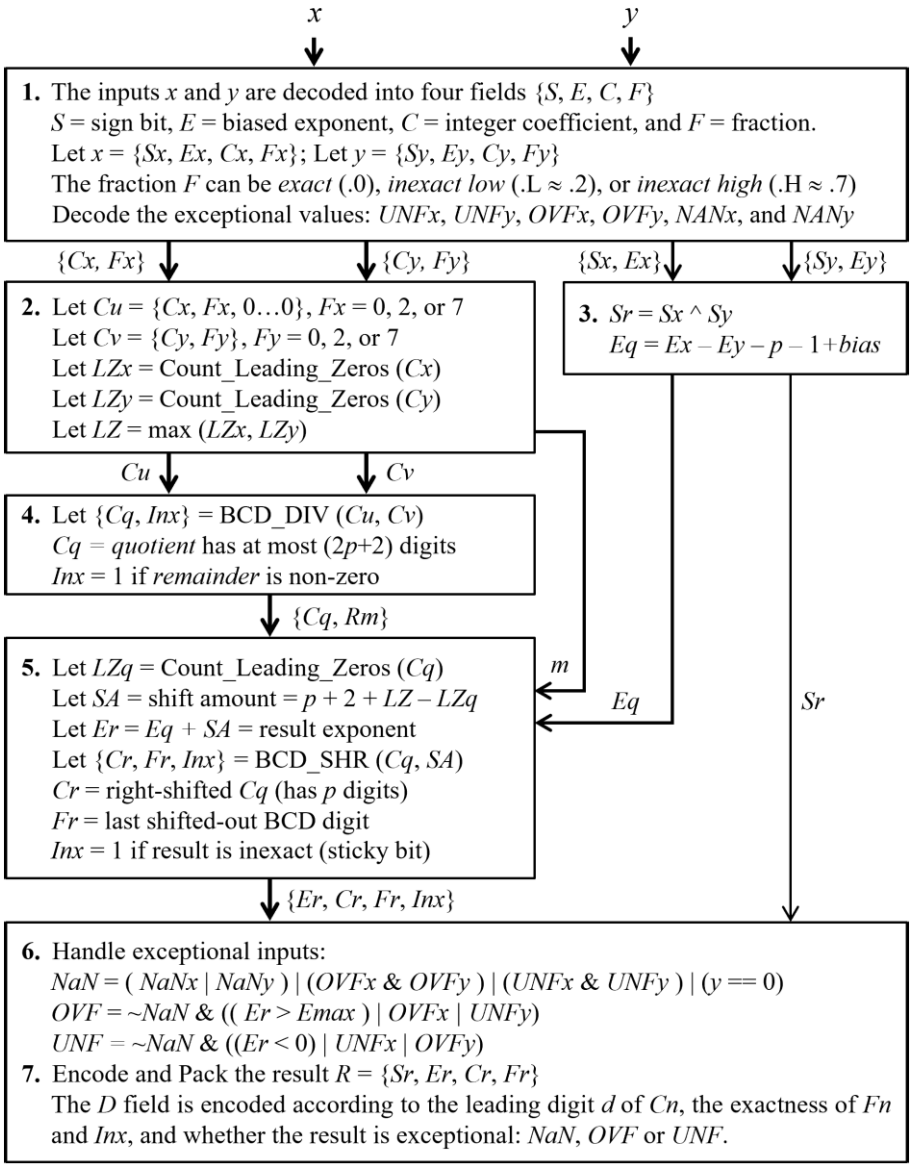

FIGURE 8. Dividing two decimal numbers x and y.

Step 5 computes LZq, which is the count of leading zeros in Cq. It determines the shift amount according to the precision p, LZ and LZq: $SA = (p + 2 + LZ - LZq)$. It computes the result biased exponent $Er = Eq + SA$ and shifts right the Cq quotient. The output is a shifted significand $\{Cr, Fr\} =$ BCD_SHR (Cq, SA) and a sticky inexact flag Inx that indicates whether any shifted-out digit is nonzero. The result coefficient Cr has p decimal digits. The result fraction Fr is a single decimal digit.

Step 6 handles exceptional inputs and detects overflow and underflow. Step 7 encodes and packs the result R, with sign bit Sr, exponent Er, and significand $\{Cr, Fr\}$.

For example, consider dividing $x = -6257652.\text{H} \times 10^{-2}$ by $y = +\ 9815678.\text{L} \times 10^{-5}$. Then, $Cu = \{6257652, 7, 00000000\}$ and $Cv = \{9815678, 2\}$. $LZx = LZy = 0$ and $LZ = 0$ indicating no leading zeros in Cx and Cy. $Eq = -2 + 5 - 8 = -5 + Bias$ and $Sr = 1$. $Cq = 0000000063751608$ and $Inx = 1$. $LZq = 8$, the shift amount $SA = 7 + 2 + 0 - 8 = 1$, and $Er = -4 + Bias$. Cq is then shifted-right to produce $\{Cr, Fr\} = \{6375160, 8\}$. The computed result is $R = -6375160.\text{H} \times 10^{-4}$.

VIII. EVALUATION

I wrote a C library to evaluate the ideas and algorithms presented in this paper. A new scientific notation is introduced for exact and inexact decimal numbers that can be easily understood by users. The traditional scientific notation is used for exact decimal numbers. For example, 1.23E-1 is

TABLE 5. Evaluation of FPBench expressions using *float64*, *decimal64*, and *DFP64*.

Expression / Inputs	float64	decimal64	DFP64
Intro sum: $w+x+y+z$ w = 9.87654E+11, x = 2.3456E-1 y = -5.4000000234E+7, z = -9.876E+11 True Result = 56E-5	Rounded Result = 6103515625000000E-19 Relative Error = 8.99% ULP Error = 5.035…	Rounded Result = 6E-4 Relative Error = 7.14% ULP Error = 0.4	Inexact Result = 5.H-4 Relative Error = 1.79% ULP Error = 0.1 **Warning:** 1 computable digit
Nonlin1: $z / (z+1)$ z = -1.000000000000123 True Result = 8130081300814008.130…E-3	Rounded Result = 8129241204640885E-3 Relative Error = 1.03×10^{-4} ULP Error = 840×10^{9}	Rounded Result = 8130081300814008E-3 Relative Error = 1.60×10^{-17} ULP Error = 0.130…	Inexact Result = 8130081300814008.L-3 Relative Error = 8.60×10^{-18} ULP Error = 0.07…
Nonlin2: $(x*y-1) / ((x*y)*(x*y)-1)$ x = 2.345 and y = 0.42644 True Result = 4999995500004049.996…E-16	Rounded Result = 4999995499900057E-16 Relative Error = 2.08×10^{-11} ULP Error = 103993	Rounded Result = 4999995500004050E-16 Relative Error = 7.29×10^{-19} ULP Error = 0.004…	Inexact Result = 4999995500004049.H-16 Relative Error = 5.93×10^{-17} ULP Error = 0.296…
Turbine1: $(6*v - (0.5*v) * (((w*w)*r)*r) / (1-v)) - 2.5$ v=1.000000000000123, w=1.23, r=3.45 = 7320037500001250.3646…E-2	Rounded Result = 7319281107135523E-2 Relative Error = 1.033×10^{-4} ULP Error = 756 $\times10^{9}$	Rounded Result = 7320037500001250E-2 Relative Error = 4.98×10^{-17} ULP Error = 0.3646…	Inexact Result = 7320037500001250.L-2 Relative Error = 2.25×10^{-17} ULP Error = 0.1646…
Turbine3: $(3 - 2/(r*r)) - 0.125*(1+2*v)*w*w*r*r / (1-v) - 0.5$ v=1.000000000000123, w=1.23, r=3.45 = 5490028125000683.379…E-2	Rounded Result = 5489460830351650E-2 Relative Error = 1.03×10^{-4} ULP Error = 567 $\times10^{9}$	Rounded Result = 5490028125000684E-2 Relative Error = 1.13×10^{-16} ULP Error = 0.621…	Inexact Result = 5490028125000683H-2 Relative Error = 5.85×10^{-17} ULP Error = 0.321…
Doppler1: $-(331.4+0.6*T)*v / ((331.4+0.6*T)+u)^2$ T=1.23E+16, u= -7.38E+15, v = 5.67E-8 = -3810082789169507.74…E-12	Rounded Result = -3819297012623276E-12 Relative Error = 2.42×10^{-3} ULP Error = 9214 $\times10^{9}$	Rounded Result = -3819297012623277E-12 Relative Error = 2.42×10^{-3} ULP Error = 9214 $\times10^{9}$	Inexact Result = -381L+1 Relative Error = 5.01×10^{-4} ULP Error = 0.1917… **Warning:** 3 computable digits

exact. It becomes 1,230,000E-7 when encoded as *DFP32*, and 1,230,000,000,000,000E-16 when encoded as *DFP64*. Trailing zeros appear in the coefficient. On the other hand, 1.23L-1 and 1.234H + 2 are *inexact* and cannot be normalized. 1.23L-1 becomes 123.L-3 when encoded as *DFP32* or *DFP64*. Similarly, 1.234H + 2 becomes 1234.H-1. Leading zeros are introduced in the coefficient according to the precision p. The L and H notations indicate that the numbers are inexact, while E means exact. The decimal exponent appears after E, L, or H. The library also includes functions that convert user input from a string into a 32-bit and 64-bit decimal number and functions that convert decimal results into formatted output strings.

There are several ways to measure the error of the elementary arithmetic operations defined by the algorithms in this paper against the IEEE 754 standard. Given that R' is an inaccurate computation of the true result R, two common mathematical definitions are the absolute and relative error:

$$Abs\_Err = |R - R'| \quad \text{and } Rel\_Err = |(R - R')/R|$$

Relative error is useful for measuring both large and small numbers because it scales with the value being measured.

Another notion of error closely tied to the floating-point representation is the Units in the Last Place (or ULPs). Since the floating-point numbers are distanced exponentially according to the exponent value, the ULP error measurement scales similarly to the relative error. Unfortunately, there is no unique definition of the ULP error in the literature. The following definition is adopted in this paper for decimal numbers. Given that R' is an inaccurate computation of the true result $R = \pm C.F \times 10^{q}$ then:

$$Ulp\_Err = |R - R'|/10^{qmax}, \quad \text{where } qmax = \max(q, q')$$

For example, given $R = 1234567.8 \times 10^{-5}$ is the true value of a computation and $R' = 1234567.\text{L} \times 10^{-5}$ is the inaccurate result then $Ulp\_Err = 0.6$, where .L is approximated as 0.2 in R'. However, if $R' = 123.\text{L} \times 10^{-1}$ then Ulp_Err becomes 0.25678, but with a loss of 4 significant digits. The loss of significant digits in an inexact result R' is equal to the number of leading zeros LZ in the integer coefficient C'. It is an indicator of the instability of a floating-point computation. The programmer can always switch to higher precision for better results but at the cost of increasing the size of data in memory.

If the result R' of a computation is exact, with zero fraction, then it must be identical to the true value R with zero error. This is guaranteed by the elementary arithmetic operations defined in Figures 5 to 8.

I used the FPBench suite [35] to evaluate the inexact decimal arithmetic introduced in this paper against the IEEE 754 binary and decimal arithmetic. The FPBench suite contains examples from a variety of domains, including the Herbie test suite [36], the Salsa test suite [37], the Rosa test suite [38], and the FPTaylor test suite [39]. I used these benchmarks to evaluate the accuracy of my *DFP64* numbers against the IEEE 754 *float64* and *decimal64*.

Table 5 shows the 64-bit direct execution of six expressions. No transformation is done to any expression. The first one is the *sum* example used in the introduction (not FPBench).

Floating-point expressions can be very sensitive to their input domain. The inputs were selected in Table 5 to amplify the error and expose the weakness of the IEEE 754 standard. The first column shows the true result obtained using 128-bit decimal arithmetic. The integer coefficient are then reduced to at most 16 decimal digits, which is the precision of 64-bit binary and decimal floating-point numbers. The exponent is adjusted accordingly. A fraction is used if the true result has more than 16 decimal digits.

The rounded *float64* and *decimal64* results, and their relative errors are shown in the second and third columns. The *DFP64* inexact result is shown in the last column. The .L and .H are approximated as 0.2 and 0.7, respectively. The *relative* and ULP *errors* are shown below the computed result. The *float64* and *decimal64* computations are rounded to nearest. The rounding tie case does not occur in these examples.

One can draw conclusions from the results in Table 5. In general, binary floating-point numbers and arithmetic are less accurate than their decimal counterpart. This is attributed to the inexact binary representation of the decimal input fractions.

The second conclusion is that the 53-bit *float64* significand and the 16-digit *decimal64* coefficient propagate erroneous bits and digits in computation. This is evident in the ULP error that grows exponentially according to the number of wrong digits. This is caused when normalizing significands with leading zero bits or digits to maximize precision in the IEEE 754 floating-point arithmetic operations injecting erroneous zero bits or digits. However, this is not allowed in my work when the input operands or results are inexact. Normalizing significands with leading zero digits should be permissible only if the result is exact. The ULP error is then reduced to a minimum as shown in Table 5 for DFP64.

The third conclusion is that inexact arithmetic provides warnings about the loss of significant digits in real-time computations, as shown in Table 5. These are detected easily by the programmer if there is an explicit representation of inexact floating-point numbers. The IEEE 754 numbers and arithmetic operations, adopted by programming languages and numeric analysis tools for decades [42], and implemented directly in hardware, still lack this inexact representation, which entails real time detection of serious errors in computation as described in this paper.

The work presented in this paper is still incomplete. Future directions include a more comprehensive error analysis for real-time inexact computation. A hardware implementation of exact and inexact floating-point numbers and arithmetic operations is work in progress.

REFERENCES

[1] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, IEEE Computer Society, Aug. 2008.
[2] *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, New York, NY, USA, 1985.
[3] M. F. Cowlishaw, "Decimal floating-point: Algorism for computers," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Jun. 2003, pp. 104–111.
[4] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
[5] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2019, Microprocessor Standards Committee, Jun. 2019.
[6] M. Cowlishaw, "Densely packed decimal encoding," *IEE Proc., Comput. Digit. Techn.*, vol. 149, pp. 102–104, May 2002.
[7] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev, "A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format," *IEEE Trans. Comput.*, vol. 58, no. 2, pp. 148–162, Feb. 2009.
[8] IBM Corporation. (2010). *The DecNumber C Library, Version 3.68*. [Online]. Available: http://speleotrove.com/decimal/decnumber.html
[9] *C# Decimal (C# Reference)*. Accessed: Sep. 29, 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/decimal
[10] (2020). *BigDecimal, Java Platform SE 7*. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html
[11] (2023). *SQL Decimal and Numeric Data Types, Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql
[12] E. M. Schwarz and S. R. Carlough, "Power6 decimal divide," in *Proc. IEEE Int. Conf. ASAP*, Montreal, QC, Canada, Jul. 2007, pp. 128–133.
[13] 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. Res. Develop.*, vol. 51, nos. 1–2, pp. 217–227, Jan. 2007.
[14] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw, "Decimal floating-point support on the IBM system z10 processor," *IBM J. Res. Develop.*, vol. 53, no. 1, pp. 4:1–4:10, Jan. 2009.
[15] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The IBM zEnterprise-196 decimal floating-point accelerator," in *Proc. IEEE 20th Symp. Comput. Arithmetic*, Tuebingen, Germany, Jul. 2011, pp. 139–146.
[16] T. Yoshida, T. Maruyama, Y. Akizuki, R. Kan, N. Kiyota, K. Ikenishi, S. Itou, T. Watahiki, and H. Okano, "Sparc64 X: Fujistsu's new-generation 16-core processor for unix servers," *IEEE Micro*, vol. 33, no. 6, pp. 16–24, Nov./Dec. 2013.
[17] L. K. Wang and M. J. Schulte, "Decimal floating-point square root using Newton–Raphson iteration," in *Proc. 16th IEEE Int. Conf. ASAP*, Samos, Greece, Jul. 2005, pp. 309–315.
[18] L. K. Wang and M. J. Schulte, "A decimal floating-point divider using Newton–Raphson iteration," *J. VLSI Signal Process.*, vol. 49, no. 1, pp. 3–18, Oct. 2007.
[19] L.-K. Wang and M. J. Schulte, "Decimal floating-point adder and multi-function unit with injection-based rounding," in *Proc. 18th IEEE Symp. Comput. Arithmetic (ARITH)*, Montpellier, France, Jun. 2007, pp. 56–68.
[20] L.-K. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam, "Hardware designs for decimal floating-point addition and related operations," *IEEE Trans. Comput.*, vol. 58, no. 3, pp. 322–335, Mar. 2009.
[21] L.-K. Wang and M. J. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator," in *Proc. 19th IEEE Symp. Comput. Arithmetic*, Jun. 2009, pp. 125–134.
[22] A. Vazquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *IEEE Trans. Comput.*, vol. 59, no. 5, pp. 679–693, May 2010.

[23] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high. Performance parallel decimal multipliers," in *Proc. 18th IEEE Symp. Comput. Arithmetic (ARITH)*, Montpellier, France, Jun. 2007, pp. 195–204.
[24] A. Wahba and H. Fahmy, "Area efficient and fast combined binary/decimal floating point fused multiply add unit," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 226–239, Feb. 2017.
[25] R. E. Moore, *Interval Analysis*. Englewood Cliffs, NJ, USA: Prentice-Hall, Englewood Cliffs, 1966.
[26] N. Revol, "Introduction to the IEEE 1788–2015 standard for interval arithmetic," in *Numerical Software Verification*, in Lecture Notes in Computer Science, vol. 10381. New York, NY, USA: Springer, 2017, pp. 14–21.
[27] (2023). *IEEE 1788-2015 Standard for Interval Arithmetic*. [Online]. Available: https://standards.ieee.org/standard/1788-2015.html
[28] (2023). *IEEE 1788.1-2017 Standard for Interval Arithmetic*. [Online]. Available: https://standards.ieee.org/standard/1788_1-2017.html
[29] O. Heimlich, "Interval arithmetic in GNU octave," in *Proc. Summer Workshop Interval Methods (SWIM)*, 2016.
[30] M. Nehmeier, "Libieeep1788: A C++ implementation of the IEEE interval standard P1788," in *Proc. IEEE Conf. Norbert Wiener 21st Century (CW)*, Jun. 2014, pp. 1–6.
[31] J. Gustafson, *The End of Error: Unum Computing*, 1st ed. Boca Raton, FL, USA: CRC Press, 2015.
[32] J. L. Gustafson, "A radical approach to computation with real numbers," *Supercomputing Frontiers Innov.*, vol. 3, no. 2, pp. 38–53, Jun. 2016.
[33] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers Innov.*, vol. 4, no. 2, pp. 71–86, Jun. 2017.
[34] F. de Dinechin, J. Müller, L. Forget, and Y. Uguen, "Posits: The good, the bad and the ugly," in *Proc. Conf. Next Gener. Arithmetic (CoNGA)*, Singapore, Mar. 2019, pp. 1–10.
[35] *FPBench Benchmarks*. [Online]. Available: https://fpbench.org/benchmarks.html
[36] J. Panchekha, A. Sanchez, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proc. PLDI*, vol. 15, 2015, pp. 1–11.
[37] N. Damouche, M. Martel, and A. Chapoutot, "Intra-procedural optimization of the numerical accuracy of programs," in *Proc. FMICS*, in Lecture Notes in Computer Science, vol. 9128. New York, NY, USA: Springer, 2015, pp. 31–46.
[38] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proc. POPL*, vol. 14, Jan. 2014, pp. 235–248.
[39] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," in *Proc. 20th Int. Symp. Formal Methods (FM)*, Oslo, Norway, Jun. 2015, pp. 532–550.
[40] L. Crespo, P. Tomas, N. Roma, and N. Neves, "Unified posit/IEEE-754 vector MAC unit for transprecision computing," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 5, pp. 2478–2482, May 2022.
[41] B. Mathis and J. E. Stine, "Implementation of high performance IEEE 754-posit conversion hardware," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May/Jun. 2022, pp. 934–937.
[42] J. Rivera, F. Franchetti, and M. Puschel, "A compiler for sound floating-point computations using affine arithmetic," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2022, pp. 66–78.

MUHAMED F. MUDAWAR received the B.Sc. degree in electrical engineering from the American University of Beirut, in 1986, and the Ph.D. degree in computer engineering from Syracuse University, Syracuse, NY, USA, in 1993.

From 1993 to 2004, he was with the Computer Science Department, The American University in Cairo. In 2004, he joined the Computer Engineering Department, King Fahd University of Petroleum and Minerals, Saudi Arabia. He is the author of numerous journals and conference publications and holds three patents. His current research interests include accurate floating-point arithmetic and hardware units, instruction set architectures, near-memory computing, parallel computing, and interconnection networks.