EXACT VERSUS INEXACT DECIMAL FLOATING-POINT NUMBERS AND ARITHMETIC

Sadia Noor[1], Ms Nilofer[2]
[1]PG Scholar, Department of DSCE, Shadan Women's College of Engineering and Technology, Hyderabad, sadianoor367@gmail.com
[2]Assistant Professor, Department of ECE, Shadan Women's College of Engineering and Technology, nilofernilofer2022@gmail.com

ABSTRACT

There is no distinction made between accurate and inexact floating-point numbers in the IEEE 754 standard. The binary encoding lacks a bit or field that determines whether a floating-point integer is precise. For both binary and decimal floats, this is true. A floating-point status register's inexact flag is raised by an inexact operation. The result is rounded to make it appear correct when utilized in a subsequent procedure. Because the floating-point arithmetic unit evaluates every input operand as if it were accurate, the computed results may contain significant mistakes. This essay explains arithmetic operations on both kinds of numbers and concentrates on differentiating between precise and inexact decimal values. The user may be certain that every decimal place in the computed result is accurate if the outcome of a series of operations is precise. On the other hand, a loss of significant digits happens if certain input operands are not correct or if the output cannot be calculated exactly. For the approximate calculated value, a separate representation is employed. The imprecise calculated result includes an estimate of the absolute inaccuracy as well. The arithmetic operations and decimal numbers presented in this work yield results that are more accurate than those calculated using the IEEE 754 standard. In the latter portion of this work, a basic assessment is presented.

INTRODUCTION

The original 1985 binary standard [2] was expanded by the IEEE 754-2008 standard [1] for floating-point arithmetic, which added decimal (radix-10) floating-point integers. Because they prevent the rounding mistakes that usually happen when translating a decimal fraction in data entered by humans into a binary fraction, decimal numbers are required. As an illustration, the decimal fraction 0.7 becomes 0.699999988 in a 32-bit binary representation. The binary fraction has to be precisely rounded. When there is an error in a calculated result, decimal values are also rounded. The Radix-10 rounding rules, however, are more focused on people. Financial computations, business databases, banking, taxation, and currency conversions all often involve decimal numbers [3]. Although binary numbers may also be helpful in scientific and technical applications, their widespread hardware support makes them widely employed. The difference between accurate and imprecise decimal floating-point values is discussed in this work. The infinite continuum of real numbers has one discrete value that corresponds to an exact number. Zero errors can be made in its representation. There is no rounding. Decimal floating-point numbers can only accurately represent a finite subset of real numbers due to the restricted precision p of the significand.

Conversely, a decimal number that is not accurate cannot be expressed precisely to a finite degree of accuracy. Since some real values, like π, cannot be represented accurately, they must be rounded to the floating-point representation's level of accuracy.

Even in cases when the operands are accurate, an inexact decimal number may arise from an inexact operation that necessitates rounding. Every imprecise floating-point operation and outcome has a corresponding mistake. An imprecise decimal number corresponds to a range of actual numbers because of the representation's poor accuracy. An inexact decimal number and an infinite set of real numbers are defined as a one-to-infinite relation.

A decimal number is represented numerically as ▪C × 10q, where q is a signed exponent and C is an integer coefficient made up of p decimal digits. The number is subnormal if C's first digit is zero. It becomes normalized otherwise. The same decimal value may have more than one representation, and the IEEE 754 standard does not mandate that decimal values be normalized. Regretfully, the same decimal representation may be rounded or accurate. The bit or field that indicates whether a decimal number is rounded does not exist. Whether implemented in software or hardware, floating-point operations consider all operands as if they were accurate, which might lead to significant inaccuracies in the computed output.

Take the addition of four decimal32 integers, for instance, in the following particular order: ((W + X) + Y) + Z). A signed exponent, an integer coefficient with a maximum of 7 decimal digits—the precision of decimal32—and a sign bit are used to represent each decimal input. The following inputs are all accurate and were selected to increase the error: Z = −1001 × 10−4, Y = −2,124,578 × 10−1, X = 8,900,123 × 10−2, and W = 1,234,567 × 10−1.

(W + X) is first calculated. The coefficient of X with the lower exponent needs to be moved to the right because of the difference in exponents: X = 8,900,123 × 10−2 = 890,012.3×10−1. Next, the coefficients are rounded to the nearest whole number (W + X) = 2,124,579 × 10−1. While the relative error is minor, the result is not exact: (0.3/2,124,579.3) ≈ 1.412 × 10−7.

Then, (W + X) + Y = 2,124,579 × 10−1 + −2,124,578 × 10−1 = 1 × 10−1. The first input operand and the total are not accurate, but the operation is. [4] refers to this subtraction as catastrophic as it has obliterated six meaningful digits. The current relative error is at 0.3 × 10−1/1.3 × 10−1, or 23%. It should be mentioned that even with digit cancellation, the output would have been accurate and error-free if both input operands had been exact. On the other hand, the IEEE 754 standard offers no information on the accuracy of the input operands.

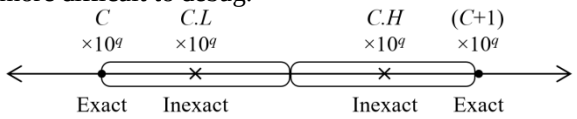
At last, Z = 1 × 10−1 + −1001 × 10−4 ((W + X) + Y) + Z. IEEE 754 states that min(EA, EB) is the favored exponent for decimal addition. When adding a decimal with a bigger exponent to a decimal with a smaller exponent, the coefficient of the larger decimal number has to be left-shifted if it contains leading zeros. Consequently, 1 × 10−1 turns into 1000 × 10−4 and 1000 × 10−4 + −1001 × 10−4 = −1 × 10−4. The actual outcome is, however, 1300 × 10−4 + −1001 × 10−4 = 299 × 10−4. Because the negative sign was computed incorrectly, the overall relative error has exceeded 100%. Once more, the operation is precise, but the operand used as the initial input and the outcome are not. But an exact operation and an exact outcome are not distinguished in the IEEE 754 standard. In conclusion, a single imprecise operation usually has a minor relative error. However, the relative error might increase significantly during a series of procedures. The example above demonstrates the necessity of differentiating between accurate and imprecise decimal values. Even when following procedures are perfect, an imprecise number transmitted in the computation might lead to a significant total inaccuracy. Doing arithmetic on non-integer decimal numbers requires a different approach.

A. INEXACT DECIMAL NUMBERS

This study suggests a novel way to encode and compute inexact floating-point arithmetic for decimal floating-point integers. Figure 1 illustrates how two inexact decimal numbers, C.L × 10q and C.H × 10q, are defined in between two successive exact decimal numbers, C × 10q and (C + 1) × 10q, with the same exponent q. In the range [0, 0.5], the L is a low fraction and the H is a high fraction [0.5, 1). Inaccurate arithmetic calculations yield an approximation of the L and H values, which are unknown.

Think about adding the same four digits from the previous decimal place. In order to get (W + X), first multiply 1,234,567 × 10−1 by 8 and then add 900, 123 × 10−2 to get 1,234,567 × 10−1 plus 890,012.3 × 10−1 ≈ 2,124,579.L × 10−1. It is not exact what (W + X) yields. A low fraction is indicated by the.L notation, where 0.L < 0.5. The outcome is not squared. Rather, the inexact result representation now includes the.L notation. (W + X) + Y = 2,124,579 follows.1.L × 10−1 = L×10−1+−2,124,578× 10−1 = 1. The operation is accurate, but the outcome is not.

Ultimately, 1.L × 10−1 + −1001 × 10−4 = ((W + X) + Y) + Z. Since 1.L × 10−1 is not accurate, it cannot be left shifted. It is not equivalent to 1000 that L × 10−1.L × 10−4. Z must thus be moved to the right: Z = -1.001 × 10−1. Thus, 1.L × 10−1 + −1.001 × 10−1 ≈ 0.L × 10−1 ((W + X) + Y) + Z. This outcome is compatible with the correct total of 0.299 × 10−1, however it is an absolute mistake with zero significant digits. It informs the programmer that there is a better way to perform the computation. On the other hand, the IEEE 754 result (−1 × 10−4) is not trustworthy. To boost instruction-level parallelism, computing (W + X) + (Y + Z) in a different sequence yields a different result. IEEE 754 states that, after rounding, (Y + Z) = −2,124,578 × 10−1 + −1001 × 10−4 ≈ −2,124,579 × 10−1. With a relative error of 100%, the result R is 2,124,579 × 10−1 + −2,124,579 × 10−1 = 0. It gets more difficult to debug.



Define two imprecise decimals in between each pair of consecutive accurate ones in Figure 1. By using the imprecise depiction proposed in Figure 1, (Y + Z) ≈ −2,124,579.Since L × 10−1 is not accurate, the total equals 2,124,579.L = 10^-1 + -2.1245779.L 10^-1 ≈ 0.L 10^-1. This result is congruent with the genuine result and is similar to the one calculated using serial addition. This correction fixes floating-point arithmetic in decimals. It should be mentioned that decimal32 is defined as non-basic in IEEE 754. It serves as a point of reference in the example. Any size and precision of floating-point values can have computational mistakes.

LITERATURE SURVEY

Computer algebra in decimal floating-point format.

Human-centric applications need to employ decimal floating-point arithmetic in order to obtain the same results as decimal arithmetic, which is the standard in human computations. Early measurements show that because software decimal arithmetic has a 100× to 1000× performance penalty over hardware, some programs appear to spend 50% to 90% of their time processing decimals. Decimal floating-point is desperately needed in hardware. However, current designs either don't meet current requirements or don't work with the accepted principles of decimal arithmetic. This study presents a novel method for decimal floating-point that satisfies the limitations and specifications of the IEEE 854 standard while still producing the rigorous outcomes required for commercial applications. This arithmetic is being implemented in hardware, and it is anticipated that this will greatly speed many different applications.

What is important to know about floating point arithmetic for computer scientists

Many people view floating-point math as an arcane subject. Given how commonplace floating-point is in computer systems, this is somewhat unexpected. There is a floating-point datatype in almost every language; computers, ranging from personal computers to supercomputers, have floating-point accelerators; most compilers will occasionally need to assemble floating-point algorithms; and almost all operating systems have to handle floating-point exceptions, like overflow. This article provides an overview of the characteristics of floating-point that directly affect computer system designers. Background information on rounding error and floating-point representation is provided at the outset, followed by a description of the IEEE floating-point standard and a plethora of examples showing how computer manufacturers might improve support for floating-point.

A software program that uses the binary encoding format to perform IEEE 754R decimal floating-point arithmetic

A significant contribution is the definition of decimal floating-point arithmetic [8], [24], which was added to the IEEE Standard 754-1985 for binary floating-point arithmetic [19], which was amended [20]. Because binary floating-point arithmetic may produce slight but unacceptable mistakes, the primary goal of this is to offer a strong and dependable framework for financial applications that are frequently subject to regulatory constraints for rounding and accuracy of the results. To address this problem, binary floating-point calculations were used to simulate decimal calculations. This approach has resulted in the development of many proprietary software packages, each with unique features and functionalities. The implementation of IEEE 754R decimal arithmetic ought to standardize decimal floating-point computations across platforms. This work presents new methods and features that are applied to a software implementation of IEEE 754R decimal floating-point arithmetic with a focus on effective usage of binary operations. Although algorithms for the more significant or fascinating operations of addition, multiplication, and division—including the case of nonhomogeneous operands—as well as conversions between binary and decimal floating-point formats are outlined, the main focus is on rounding techniques for decimal values stored in binary format. Performance results are provided for a greater variety of operations, indicating potential use of our method for decimal floating-point computations applications. This work builds upon a previous publication [6].

From the standpoint of testing and implementation, decimal floating-point in Z9

Despite the widespread usage of decimal arithmetic in financial and commercial applications, the associated calculations are managed by software. Applications that employ decimal data may thus see a decrease in performance. Performance gains for such applications are anticipated when the newly specified decimal floating-point (DFP) format is used in place of binary floating-point. The first IBM computer to support DFP instructions is the System z9TM. We give a summary of this approach and offer some metrics for the performance improvements made possible by hardware assistance. A detailed presentation of the many instruments and methods used for the DFP verification at the unit, element, and system levels is made available. Using a shared reference model to forecast DFP findings, many IBM groups worked together to verify the new DFP facility.

The IBM System Z10 CPU has support for decimal floating points.

The decimal floating-point (DFP) feature, first introduced on the IBM System z9® processor, is now supported by hardware on the newest IBM zSeries® CPU, the IBM System z10TM processor. The z9® processor uses a combination of hardware and low-level software to implement the capability. The System pTM 570 server, which runs on the IBM POWER6TM CPU, has unveiled a hardware implementation of the DFP function. The most recent zSeries processor contains an improved decimal floating-point unit that supports the standard decimal fixed-point instruction set of the zSeries and is based on the POWER6 processor DFP unit. This document describes the new software support for the DFP facility, which includes support in IBM DB2® and middleware, as well as compilers for IBM z/OS®, JavaTM JIT, and C/C++. The hardware architecture to support dual decimal fixed point and DFP is explained as well.

The decimal floating-point accelerator, IBM zEnterprise-196

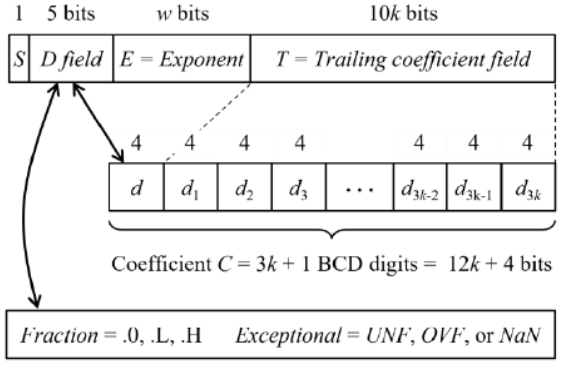
When rounding problems make it impossible to employ binary floating-point operations, decimal floating-point arithmetic is frequently used in commercial computing applications, such as financial transactions. Standardized decimal floating-point (DFP) formats were established by the updated IEEE Standard for Floating-Point Arithmetic (IEEE-754-2008). Hardware accelerators supporting IEEE decimal floating-point are becoming increasingly common as more software programs implement it. The IBM zEnterprise-196 processor's second-generation decimal floating-point accelerator is described in this publication. The 4-cycle deep pipeline was created with the goal of greatly increasing DFP operation bandwidth while optimizing fixed-point decimal operation latency. The unit is described in great length, and a comparison with other implementations available in the literature is given.

Fujistsu's next-generation 16-core CPU for Unix servers, the Sparc64 X

The Sparc64 X from Fujitsu is a 3 GHz processor designed for Unix servers. It has 16 cores, a shared level-2 (L2) cache of 24 Mbyte, memory controllers,

I/O controllers, and system controllers for interconnecting multiple chips. The authors of this paper improved the microarchitecture and included the High-Performance Computing Arithmetic Computational Extensions (HPC-ACE) extended instruction set, which was previously utilized in the K computer. There has a peak memory bandwidth of 102 Gbps. These characteristics provide incredibly high throughput capabilities. Furthermore, the designers enhanced the processor core pipelines with additional features that speed up software operations like processing cryptography. These features are referred to as "software on chip" (SWoC). In addition, they make advantage of mainframe-derived high-reliability technologies to guarantee mission-critical systems run reliably. This page provides an introduction of the Sparc64 X processor family, details the microarchitecture's historical and present developments, and displays the findings of tests conducted to evaluate the power and performance of SWoC.

PROPOSED METHODOLOGY



Coefficient, fraction, and exceptional values.

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.

MODULE EXPLANATION:

A. UNIVERSAL NUMBERS, IEEE 754, AND IEEE 1788

2008 saw the introduction of the IEEE 754 decimal standard, which was updated in 2019 [5]. There are four levels to it. The mathematical structure is defined at the first level as an extended collection of real numbers that includes both positive and negative infinity. An extended real number is mapped to a floating-point value by rounding. There is a many-to-one connection. An algebraic closed system on floating-point data is defined at the second level. Signed zeros, finite non-zero numbers, signed infinities, and not-a-numbers (NaN) are examples of floating-point datums. The representation of floating-point data is defined at level three, while its binary encoding is defined at level four.

The decimal standard designates the interchange formats with widths of 32, 64, and 128 bits, respectively, as decimal32, decimal64, and decimal128.

As seen in Figure 2, the format consists of three fields: a combination field, a trailing coefficient field, and a sign bit S. The biased exponent E and the leading digit of the coefficient are encoded in the combination field's (5 + w) bits. This definition was made in order to maximize the exponent range and optimize the encoding of the leading digit.

3k decimal digits are encoded in 10k bits (k declets) in the trailing coefficient field. With p = 3k + 1 decimal digits, the integer coefficient C has decimal precisions of 7, 16, and 34 for decimal32, decimal64, and decimal128 correspondingly. A decimal float has the following numerical value: (−1)S × C × 10E−Bias.

There are two ways to encode the decimal coefficient according to the decimal standard. Three decimal digits can be effectively encoded using 10-bit declets in the first encoding technique, called Densely Packed Decimal (DPD). As explained in [6], DPD needs basic logic to unpack and pack the BCD digits at the start and finish of each operation. BCD digits are used internally by decimal floating-point units for arithmetic operations. The decimal coefficient is encoded using a binary integer in the second encoding strategy. This is referred to as BID encoding, or Binary Integer Decimal. It is easier to use this encoding system than DPD. Nevertheless, hardware implementation is the biggest challenge when utilizing the BID encoding. For example, left and right shifters are used to align the decimal coefficients in order to create a decimal floating-point adder in hardware. For the DPD encoding that packs BCD digits, this is effective. Nevertheless, the BID encoding raises the cost and complicates the hardware alignment of the two components. It is recommended to implement left and right shifting as hardware multipliers by positive and negative powers of ten. Negative powers of 10, such 10−1 and 10−2, however, cannot be precisely expressed in binary. Because of this, software implementations that utilize binary hardware are the primary users of BID encoding. There are several software options available.

The decNumber C library [8], the Java BigDecimal [10], the C# decimal [9], the Intel Decimal FP library [7], and SQL decimal [11]. The speed of software libraries is a downside. It has been observed that software-implemented operations operate 100–1000 times slower than hardware-implemented operations [3].

A decimal number, in contrast to a binary floating-point number, can have more than one representation. The cohort of a floating-point number is the collection

of representations to which a decimal number translates [1]. There exist (p − n + 1) representations of a non-zero integer with n significant decimal digits (beginning at the most significant non-zero digit and terminating at the least significant non-zero digit), where p is the precision. In decimal32, for instance, the value 0.2 may be represented in seven different ways: $0.2 = 2 \times 10{-}1 = 20 \times 10{-}2 = \ldots = 2000000 \times 10{-}7$. Specifically, zero has a huge cohort: each exponent is represented in the cohort of 0 [1]. Several popular processors, including the IBM Power [12], IBM System Z [13], [14], and [15], as well as the Fujitsu Sparc64 processors [16], were the first to use decimal floating-point units. In addition to decimal adders with injection-based rounding and associated operations [19], [20], and [21], Wang and Schulte also presented the implementation of decimal floating-point square root and division using Newton-Raphson iteration [17], [18]. The construction of parallel decimal multipliers was demonstrated by Vasquez et al. [22], [23]. A hybrid binary/decimal floating-point fused multiply add unit was demonstrated by Wahba and Fahmy [24]. The inability of the IEEE 754 decimal floating-point standard to discriminate between accurate and imprecise decimal values (which also applies to binary integers) is a significant source of worry. When rounding occurs, the standard specifies an inexact operation. On the other hand, neither the binary encoding nor the representation provide information about whether a computed result is inaccurate. Specifically, the error of an operation is increased when the coefficient of an imprecise input operand is left-shifted, inserting erroneous trailing zeros. Improving floating-point arithmetic quality requires being able to distinguish between accurate and inexact integers. An infinitely precise result indicates that a series of floating-point operations produced an exact result. Conversely, an imprecise outcome will alert the user to the extent of the mistake in a particular calculation.

I. INTERVAL ARITHMETIC AND IEEE 1788

Since the 1960s, mathematicians have been using interval arithmetic [25] to set limitations on rounding mistakes and create techniques that provide accurate results. Intervals are the inputs used in all computations, and the outputs are also intervals. It is ensured that the calculated intervals contain the precise computation values. The Fundamental Theorem of Interval Arithmetic (FTIA) [26] is the most valuable aspect of interval arithmetic. Four layers make up the IEEE 1788 standard [27] for interval arithmetic. The definition of an interval on real numbers and the functions of operations on intervals are laid forth in the first level of mathematics. The discretization of intervals, which establishes the interval endpoints and *types*. The fourth level deals with binary encoding, whereas the third level deals with representing intervals using floating-point integers.

As long as they agree on common definitions, the standard is also intended to support several flavors, or models, of intervals. The set-based taste has gained traction thus far. In order to manage exceptions, the standard affixes tags, also known as decorations, to every interval. Compared to IEEE 754, the IEEE 1788 standard includes additional relational operators. Relational operators for intervals include subset, precedes, precedes or touches, and interior to, in addition to checking for equality, less than, and less than or equal. Additionally, several predicates—like before, meets, overlaps, and contains—do not exist in IEEE 754. In 2017, the IEEE 1788 standard was updated and streamlined to incorporate the functions and features that are most frequently utilized in real-world scenarios [28]. A pair of IEEE 754 binary64 floating-point integers, a decorating system for exception-free calculations, and a propagation of the calculated results' attributes are what constitute an interval. Compared to IEEE 754, the IEEE 1788 standard is more complicated. There isn't any hardware integration. As a proof of concept, only software-compliant libraries have been created. Octave library [29] and libieee1788 [30] are two examples of them. The wrapping problem and the dependence problem, which result in huge expansions of the generated intervals and offer no information on the solution, are two main shortcomings of interval arithmetic and IEEE 1788.

II. UNIVERSAL NUMBERS

Gustafson suggested universal numbers, or Unums, as an alternative to the IEEE 754 standard. The significand and exponent are stored in a variable-width format in the initial version, referred to as Type 1 Unum [31]. The size of the exponent and fraction are specified in the esize and fsize variables. The 64-bit binary float is an example of a "one size fits all" option that a programmer is relieved of due to its variable dynamic range and accuracy. But the author acknowledges that Type 1 unums have a lot of disadvantages, especially when it comes to hardware implementation [32]. They need to be emptied into a designated storage space. To reference the other fields, one must first read the additional level of indirection added by the esize and fsize fields. Certain bits patterns are not utilized, and certain values can be stated in more than one manner.

Signed two's complement integers are directly mapped to the projective real number line as type 2 unums [32]. On a circle, real numbers are mapped so that positive and negative infinity converge at the top. The u-lattice is the user-defined collection of precise reals between 1 and infinity that has been chosen. But since it doesn't make use of positional representation or the conventional radix, table lookup is necessary. Requiring the two's complement of the remaining bits and preserving the sign bit, one may effortlessly determine the reciprocal of a Type 2 unum. Division approaches the speed of multiplication. There is no

need for an exception because infinity is the reciprocal of zero and vice versa. But for simple arithmetic operations like addition, subtraction, and multiplication, Type 2 unums are hard to extend to high precision and require huge database lookups (that increase exponentially with precision). On the other hand, algorithmic techniques that are simple to implement in hardware make working with floating point numbers easier.

Type 3, or Posits, is the most recent iteration of unums [33]. Similar to binary floats, positivism provides more accuracy in the vicinity of one. They have four fields in their tapered floating-point format (sign bit, regime, exponent, and fraction), making it significantly more difficult than IEEE 754. Similar to binary floats, posesits are rounded. There is no indication of precise decimal fractions (0.1 + 0.2≠ 0.3) or difference between rounded and exact possibilities. Small 8-bit or 16-bit posits are utilized in machine learning, where they have shown to be beneficial. Positivity is becoming more and more popular. A high-performance IEEE 754-Posit conversion hardware was constructed by Mathis and Stine [41], and a unified Posit/IEEE 754 vector MAC unit was implemented by Crespo et al. [40].

Nonetheless, there are instances in which propositions do worse than floating-point, such as in simulations of particle physics [34]. The rounding error in the product of two posits is not necessarily a posit, multiplying a posit by a power of two is not always correct, and posits can become ugly in multiplicative cancellation, as discussed in [34].

B. EXACT VERSUS INEXACT DECIMAL NUMBERS

This work explains arithmetic on both sorts of numbers (something not done in IEEE 754) and concentrates on differentiating between precise and inexact decimal floating-point integers. Due of their capacity to accurately represent decimal fractions, decimal floats will be the subject of this discussion.

Within the infinite continuum of real numbers, a single discrete value is represented by an accurate decimal floating-point number. Zero errors can be made in its representation. For a precise decimal float to have a distinct representation, it must be normalized. Not in IEEE 754, but in my job, this is a need. Except in cases when the number is 0, the first digit (d) of the coefficient cannot be zero. For instance, 2,000,000×10−7 with p=7 decimal digits is the unique representation of the precise decimal value 0.2. The idea is to have every precise decimal float have a unique representation. The idea of cohorts is dropped. To convert a decimal32 number exactly into a decimal64 number, add trailing zeros to the significand and modify the exponent. For instance, when 0.2 = 2,000,000 × 10−7 is translated to decimal64, it becomes 2,000,000,000,000,000 × 10−16 with 16 decimal digits. If one of the nine trailing decimal digits that are displaced away from the significand is nonzero, the conversion from an exact decimal64 number to an exact decimal32 number might result in an inexact value. Finite precision cannot accurately represent an imprecise decimal number. For instance, 3,141,592.H × 10−6 (with p = 7) is an imprecise decimal32 representation of π, where 0.H denotes a large fraction (0.5 ≤ 0.H < 1). A 0.H × 10−6 absolute inaccuracy is present. π may be represented indecisively in decimal64 as 3,141,592,653,589,793.L is × 10−15 (p = 16), where a low proportion is denoted by 0.L (0 ≤ 0.L < 0.5). A 0.L × 10−15 absolute inaccuracy is present. An imprecise integer, like π, does not get more precise when converted from decimal32 to decimal64. The following has leading zeros added: π = 0,000,000,003,141,592.H = 10^-6. Unknown trailing digits prevent an inexact number from being left-shifted and normalized if it hasn't been normalized. In conclusion, it's possible or not to normalize imprecise decimal values using a distinct representation. As seen in Figure1, they have .L or .H representations that denote low or high fraction intervals: 0.L = [0, 0.5) and 0.H = [0.5, 1). An operation with precise or inexact operands might produce an inexact number. Rounding is not utilized, though. An interval inside the infinite real number continuum is an imprecise number. Interval arithmetic is not utilized, though. This work defines, instead, inexact arithmetic on inexact decimal values.

A. EXACT VERSUS INEXACT ZEROS

As per the IEEE 754 decimal standard, zero is defined as a big cohort with a zero significand and an arbitrary value in the exponent field. For every exponent value q, zero equals 70 × 10q. Exact zero has no special symbol, while inexact zero has no defined term.

This work, however, makes a distinction between accurate and inexact zeros. Since all bits are zeros, exact zero has a unique representation. It has no sign bit and is expressed as 0. Inexact zeros, on the other hand, are many, signed, and denoted as 0.L×10q or 0.H× 10q. They stand for calculation errors. It might be either 0.L or 0.H. Nonetheless, the error's magnitude is indicated by the exponent q.

B. FORMAT

As seen in Figure 3, this work proposes a new format for both accurate and inexact decimal values. The four fields in it are the sign bit (S), the biased exponent field (E), the 5-bit digit field D (which encodes the leading decimal digit of the coefficient and indicates if the number is exact or not), and the trailing coefficient field T (10 × n bits, or n declets) that contains the trailing 3 × n decimal digits of the coefficient.

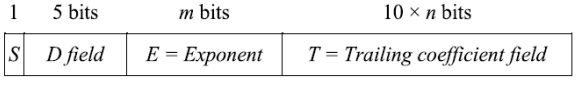


FIGURE 3. A new floating-point format for decimal exchange.

The fields, bit-length, and significant values of the recently proposed decimal floats, DFP32, DFP64, and DFP128—with lengths of 32, 64, and 128 bits, respectively—are defined in Table 1. There are m bits in the exponent field. A finite decimal number has a biased exponent range of E = 0 to 2m −1, and a biased exponent of 2m −1 + p −1, where p is the precision (p = 7, 16, and 34 for DFP32, DFP64, and PDF128 respectively).

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	$\pm C\times10^{E-38}$	$\pm C\times10^{E-143}$	$\pm C\times10^{E-2081}$
Inexact values	$\pm C.L\times10^{E-38}$ $\pm C.H\times10^{E-38}$	$\pm C.L\times10^{E-143}$ $\pm C.H\times10^{E-143}$	$\pm C.L\times10^{E-2081}$ $\pm C.H\times10^{E-2081}$

TABLE 1. Decimal values, both precise and inexact, and field lengths.

An integer coefficient C, which is the product of the leading decimal digit in the D field and the third decimal digits in T, is the significand of a decimal number.

An precise decimal number has the value ⁷C × 10q. An imprecise decimal value can have the value ◂C.L × 10q or ◂C.H × 10q, where q = E − Bias. Although it is not as broad as that specified in the IEEE 754 standard, the exponent range for DFP32, DFP64, and DFP128 is described in Table 1 and is still enough for applications. The precision p is the same, though.

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 = ----

TABLE 2. The D field encoded in five bits.

C. LEADING DIGIT AND TRAILING COEFFICIENT

The 4-bit leading digit d of the integer coefficient C is encoded in the 5-bit D field, which also shows whether a decimal value is precise or not. Table 2 displays the encoding. The integer is either an exact zero or an extraordinary value if D = E = 0. The decimal value is precise, the leading digit d = 1 to 9, and the integer coefficient C is normalized if D = 1 to 7, 24, or 25. The decimal value is imprecise and its coefficient is expanded with a small fraction if D falls between 8 and 15, 26, or 27. If D falls between 16 and 23, 28, or 29, the decimal value is equally imprecise, but it is stretched out using a large fraction. An inexact decimal number's leading digit (d) ranges from 0 to 9, thus normalizing its integer coefficient shouldn't be done. It's easy to decode the 5-bit D field = abcde into a 4-bit leading decimal digit, d = wxyz: d = 0cde for D values between 0 and 23, and d = 100e for D values between 24 and 29. The following are the logic expressions: z = e, y = d&~w, x = c&~w, and w = a&b.

As seen in Figure 4, the integer coefficient C is the result of concatenating the trailing coefficient T with the leading digit D. Densely packed decimal is used to encode the T field (DPD). It is inexpensive and requires only a few basic formulae to unpack and load the 10-bit declets into three BCD digits [6]. Table 2 does not specify the leading digit d if D = 0 and E /= 0, or if D = 30 or 31. D might be set to decimal 10 as one possibility. 1099... 9 is the maximum coefficient C. If D = 0, the decimal number is accurate; if D = 30, or 31, it is not exact. Using an expanded exponent range with d set to 0 is an alternative. With the loss of one significant digit (d), the coefficient C becomes the trailing field T, while the exponent range is expanded.

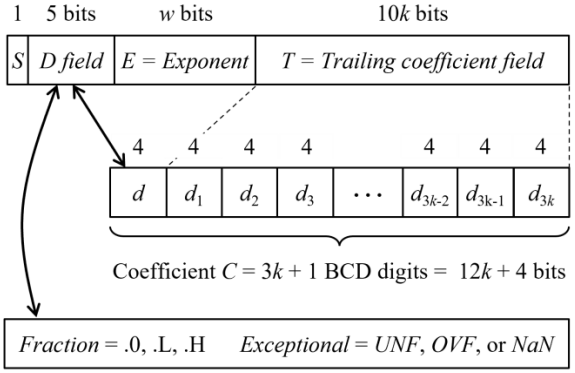


FIGURE 4. COefficient, proportion, and exceptional values

D. EXCEPTIONAL VALUES

There are five exceptions that can be caught in a particular calculation according to the IEEE 754 standard. These include division by zero, overflow, underflow, inexact, and incorrect operation. Either capturing a trap or putting a flag in a floating-point status register indicates an exception. Floating-point results from computational processes may indicate floating-point exceptions. An imprecise operation yields a rounded result, following IEEE 754. In my work, an imprecise operation yields an imprecise outcome, which may be either ◂C.H × 10q or ∗C.L × 10q. There is no need for a hardware flag, and there is no rounding.

It is possible to substitute exceptional values recorded in the binary representation for the hardware flags included in the floating-point status register. This paper defines three extraordinary values. Overflow is a

signed decimal float that is unrepresentable due to its huge exponent. −OVF < x < +OVF holds true for each finite decimal float x that can be expressed.

Underflow (UNF) is the reciprocal of Overflow, and vice versa. In order to give a steady underflow to precise zero, IEEE 754 employs denormalized integers; nonetheless, this work defines UNF as an extraordinary value, which is distinct from zero. −OVF < −x < −UNF < 0 < +UNF < x < +OVF holds for each positive finite decimal integer x. The result OVF × UNF is uncertain, or NaN, as should be mentioned. The third value that is uncommon is Not-a-Number (NaN). It can be Not-a-Real, like the square root of a negative integer, or ambiguous, like dividing zero by zero. NaN values are unordered and have an uncertain sign.

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*)

TABLE 3. Coding the extraordinary values.

When both D and E are zero, exceptional values are encoded using the T field, as Table 3 illustrates. The value is exactly zero and the sign bit is ignored if D = E = T = 0. The exceptional values are NaN, OVF, and UNF, respectively, if D = E = 0 and T /= 0. The sign bit is ignored by the NaN and Zero values. Different NaN exceptional values, including indeterminate and Not-a-Real, can also be encoded.

C. DECIMAL ADDITION AND SUBTRACTION

Exact decimal floating-point number addition and subtraction are clearly stated. The coefficient of the number with the smaller exponent must be moved to the right in order to raise its exponent if the exponents disagree. In my job, we normalize precise decimal values, thus we don't count the leading zeros or shift a source operand to the left in order to reduce its exponent. Next, the difference or total is normalized. Even with correct operands, addition and subtraction can provide imprecise results. This happens when the normalized significand's fractional component isn't zero. There isn't any rounding, though. The L or H fraction is used to denote an approximate result if the fraction that occurs after the decimal point indicates that the result is not accurate.

However, it is more complicated to add and subtract imprecise decimal floating-point values. How to define arithmetic on 0.L and 0.H is the question. Utilizing interval arithmetic is one option. For instance, 0.L + 0.L may equal either 0.L or 0.H, 0.L + 0.H may equal either 0.H or 1.L, and 0.H + 0.H may equal either 1.L or 1.H. Similarly, 0.L − 0.L, 0.H − 0.H, and 0.H − 0.L can all be ∗0.L, 0.H − 0.L can be either 0.H or 0.L, and 0.L − 0.H can be either −0.H or −0.L. The disadvantage of interval arithmetic is that the result can only be represented with two ends. Over a series of operations, the intervals get larger and more complicated, which makes implementation more difficult. This work offers a straightforward method for dealing with arithmetic on imprecise decimal floating-point Although the arithmetic is imprecise, the results are more trustworthy than those produced, per IEEE 754. It makes it abundantly evident that the outcome is imprecise and does not add complexity to the decimal floating-point unit's hardware implementation.

The single-digit approximations of 0.L and 0.H are used in inexact arithmetic. 0.L ≈ 0.2 and 0.H ≈ 0.7 are the options. The reasoning behind this is that the median for 0 to 4 is 2, while the median for 5 to 9 is 7. There is a 0.5 difference between 0.L and 0.H. Likewise, there is a 0.5 difference between 0.H and 1.L.

+	+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

TABLE 4. Addition to 0.L and 0.H that is not accurate.

Table 4 defines inexact addition to ^{7}0.L and ^{7}0.H. In this case, 0.L + 0.H ≈ 0.9 ≈ 0.H, 0.L + 0.H ≈ 0.2 + 0.2 ≈ 0.L, and 0.H + 0.H ≈ 1.4 ≈ 1.L (not 1.H). Likewise, the definitions of (0.L − 0.L) and (0.H − 0.H) are + 0.L. But −0.L + 0.L turns into −(0.L − 0.L) ≈ −0.L instead of + 0.L. All the other items in Table 4 come from consistent sources.

A shifted coefficient can be added or subtracted using the same digit approximation. For instance, 0.L + 0.8 ≈ 1.L, 0.H + 0.8 ≈ 1.5 ≈ 1.H, and 0.L + 0.3 means 0.2 + 0.3 ≈ 0.H. Similarly, 0.L − 0.8 equals −0.H, 0.H − 0.8 ≈ −0.L, and 0.L − 0.3 = 0.2 − 0.3 ≈ −0.L. An procedure for adding and subtracting two decimal values, x and y, is shown in Figure 5. Software or hardware can be used to implement the algorithm.

In the first stage, all of the x and y fields are extracted using the format shown in Figure 4. A digit (.0,.2, or.7) is then injected for the fraction F, and the exceptional values are decoded. To prevent a negative zero outcome, the sign bit Sx or Sy of an input x or y is cleared if it is zero.

In step two, the biased exponents Ex and Ey are compared, and their maximum Eu and absolute difference d are calculated.

Step 3 generates the switched significands {Su, Cu, Fu} and {Cv, Fv} by swapping the input operands if Ex < Ey.

Step 4 finds the input operation Op, where ADD is 0 and SUB is 1, as well as the effective operation EOP based on the sign bits Sx and Sy.

In the event where the first swapped operand is exact (Fu == 0) and there is a difference in the exponents (d!= 0), Step 5 stores the guard digit for subtraction, if necessary. To preserve the guard digit, the new coefficient Cu is moved one decimal place to the left.

Additionally decremented are the maximum exponent Eu and the exponent difference d.

Furthermore, Step 5 moves the significand {Cv, Fv} to the right in accordance with the exponent difference d to get {Cw, Fw, Inx}. By doing this, the significands are aligned to share one exponent, Eu. If any fraction digit is pushed out or deleted, the Inx (inexact) flag is raised.

Step 6 processes the aligned significands {Cu, Fu} and {Cw, Fw} by adding or subtracting them using BCD. It computes the magnitude of the result significand {Carry, Csum, Fsum} using a BCD adder, converts subtraction into addition to the BCD (10's) complement, then computes LT to determine whether {Cu, Fu} is smaller than {Cw, Fw}. This step computes its BCD counterpart in order to post-correct the magnitude of the result {Cr, Fr} if LT == 1 for subtraction. This is basically required for the implementation of hardware. Nevertheless, the Binary Integer Decimal (BID) encoding, which utilizes the binary hardware to calculate the total or difference, may be used in software implementation. In Step 6, the outcome sign Sr = Su ∧ LT is also computed, complementing the sign bit Su if LT.

In step 7, the common exponent Eu is adjusted and the result significand {Cr, Fr} obtained in step 6 is normalized. The exponent Eu is lowered and the result significand {Cr, Fr} is shifted-left in accordance with the count LZ of leading zeros in Cr if it is accurate and contains leading zero digits. To provide a distinctive depiction of the outcome, this is required. In the event that the result contains an additional Carry digit, the exponent Eu is increased and {Carry, Cr, Fr} is moved one BCD digit to the right. The output {En, Cn, Fn} is normalized after step 7. En is lowered to 0 if the outcome is precisely zero.

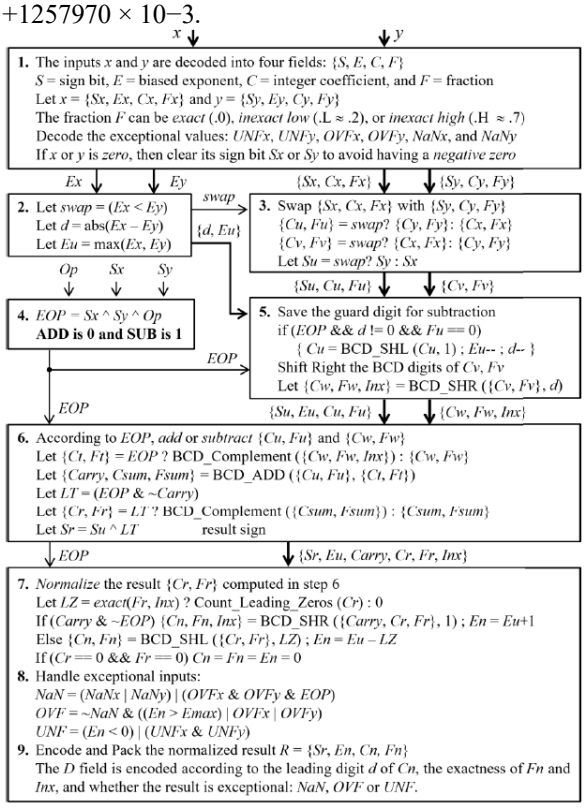
Note that the seventh normalization step is adaptive and may identify precise and imprecise results. It differs from significance arithmetic, which does not distinguish between the two, in this way. The lack of rounding options and rounding steps makes implementation easier.

Step 8 identifies overflow and underflow, processes the exceptional inputs NaN, OVF, and UNF, and generates an extraordinary output.

In step nine, the final normalized result is encoded and packed. For instance, let us add two precise DFP32 input operands: x = −6254763 × 10−5 and y = −9877012 × 10−4. Ex < Ex, hence the operands in the input must be switched. {Cv, Fv} = {6254763, 0}, {Cu, Fu} = {9877012, 0}, and the sign Su = Sy = 1. The effective operation EOP = 0, which is addition, has a maximum exponent of Eu = −4 + bias. {Cv, Fv} needs to be shifted one BCD digit to the right in order to become {Cw, Fw} = {0625476, 3} and Inx = 1 due to the difference in exponents. The result of adding the significands is {Carry, Csum, Fsum} = {1, 0502488, 3}. Sr = Su = 1 is the outcome sign. The exponent is increased to En = Eu + 1 = −3 + bias and the significand is normalized to {Cn, Fn} = {1050248, 8} due to the Carry. R = −1050248.H × 10−3, the approximate result, is encoded in accordance with Table 2 and Figure 4.

Now remove x = +1000234 × 10−1 and y = +9876543 × 10−2 from each other. Since {Cv, Fv} = {9876540, 0} and {Cu, Fu} = {1000234, 0}, Ex > Ey. Subtraction is represented by the sign Su = Sx = 0, the exponent Eu = −1 + bias, and the EOP = 1. The exponent Eu is decremented to become Eu = −2 + bias, the exponent difference is decremented to become d = 0, and the Cu coefficient is shifted-left one decimal digit to become Cu = 10002340 in order to preserve the guard digit.

This is only applied to subtraction when there is an exponent discrepancy and the precise number is the one with the greater exponent. The two significands {Cw, Fw, Inx} = {9876543, 0, 0} are already aligned. After that, subtraction is changed to addition to the complement of ten, and {Ct, Ft} = {90123457, 0} is the result. In order to extend the coefficient and acquire the correct sign of the result, the leftmost digit, 9, is entered into Ct. {Carry, Csum, Fsum} = {10002340, 0} + {90123457, 0} = {0, 0125797, 0} is the result of adding the significands. Zero is the carry digit. It shows that (LT = 0) the outcome is positive. The result would have been negative (LT = 1) if the Carry digit had been 9, necessitating the post-correction of the result significand using the 10's complement of the {Csum, Fsum}. As the fraction is precise and the computed Csum has a leading zero, it is shifted-left and normalized to become {Cn, Fn} = {1257970, 0}. After decrementing the exponent, the answer is R = +1257970 × 10−3.



D. DECIMAL COMPARISON

With the exception of NaN, all floating-point numbers are sorted according to IEEE 754. There are four mutually incompatible relations given two floating-point numbers: equality (EQ), less than (LT), greater than (GT), or unordered (UN). Even if two rounded numbers reflect distinct real numbers, they can nevertheless be equal.
Equality has two connotations in my work. It may be precise or imprecise. If two finite decimal values, x and y, are accurate and have the same binary encoding due to normalization with a distinct representation, then they are equal (EQ). Should x and y be equal, then the difference between them (x - y) should be 0. Conversely, two inexact decimal values, or an exact with an inexact decimal value, can be compared using inexact or approximation equality (AE).

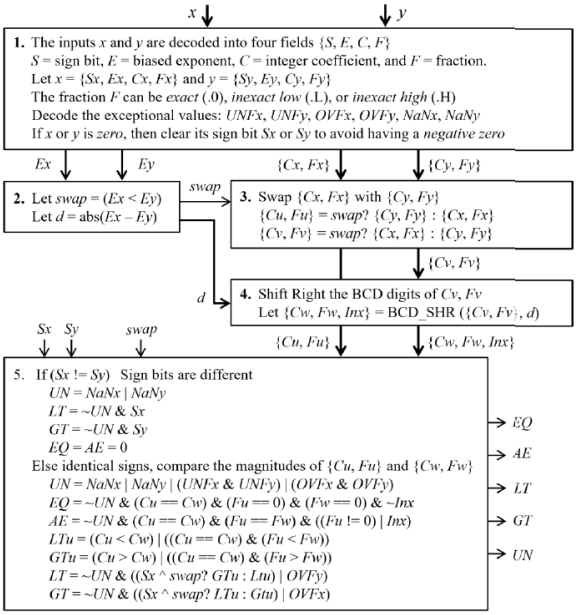


FIGURE 6. comparing x and y, two decimal values.

An algorithm for comparing two decimal values, x and y, is shown in Figure 6. The relations equality (EQ), approximate equality (AE), less than (LT), larger than (GT), and unordered (UN) are the five that are mutually incompatible. Strict ordering applies to exact decimal floats. −OVF < −x < −UNF < 0 < +UNF < +x < +OVF holds for each positive precise decimal value x. When two finite decimal integers x and y have the same sign and at least one of them is imprecise, their aligned significands are nearly identical to one decimal place: x ≈ y ≈ C.L × 10q, or x ≈ y ≈ C.H × 10q. C × 10q, however, < C.L × 10q < C.H × 10q. Exact zero is smaller than inexact zero in a similar manner.
Consider the case where x = 314.y = 31415 and L × 10−2.If there are two estimates of π, H × 10−4, with distinct exponents, then x and y have to line up. With a smaller exponent, y's significand is pushed to the right, becoming y = 31415.H × 10−4 ≈ 314.L × 10−2, which represents x and y's approaching equivalence. Similarly, z = 314.1 × 10−2 ≈ 314 if z = 3141000 × 10−6 is an exact decimal value.Approximately, x equals L × 10−2. But z equals 31410.y = 31415 is less than 0×10−4.H × 10−4. This illustration demonstrates that while perfect equality is transitive, approximate equality is not. Since NaN values are unordered, no decimal number x can be compared to them. In the same way, two OVF (or UNF) values with the same sign cannot be sorted. Nevertheless, there is a finite decimal number x that determines the order of the UNF and OVF values.
The Boolean functions isUNF, isOVF, isNaN, isExact, isZero, and isInexactZero can also be used to determine the value of a decimal number. For instance, if the operand x is exact, the function isExact(x) returns true. Specifically, UNF, OVF, and NaN are imprecise.
The == operator is widely used in computer languages to test equality. An operator to test approximate equality does not exist. In my work, approximate equality is tested using the Boolean function AE(x, y), whereas precise equality is tested using the statement (x == y).

DECIMAL MULTIPLICATION

Decimal multiplication does not require the significands to line up when the exponents are different, in contrast to addition and subtraction. It's easy to multiply two accurate decimal values. The exponents are added and the decimal coefficients are multiplied. Next, the coefficient of result is normalized. The outcome becomes imprecise if one of the numbers that has been pushed out is not zero. Whether the fraction is 0.L or 0.H is shown by the final shifted-out digit.
It is more complex to multiply two inexact decimal numbers (or an exact number multiplied by an inexact decimal number) because multiplying an integer coefficient by 0.L or 0.H increases the mistake. To approach 0.L and 0.H, digit injection is employed, much like in addition and subtraction. If alternative approximations of 0.L and 0.H are employed, different outcomes are obtained, as seen in

$$987.\mathrm{L} \times 10^{-3} \times 6543.\mathrm{H} \times 10^{+2}$$

$$987.0 \times 10^{-3} \times 6543.5 \times 10^{+2} = 6,458,434.5 \times 10^{-1} \approx 645.\mathrm{H} \times 10^{+3}$$

$$987.2 \times 10^{-3} \times 6543.7 \times 10^{+2} = 6,459,940.64 \times 10^{-1} \approx 645.\mathrm{H} \times 10^{+3}$$

$$987.4 \times 10^{-3} \times 6543.9 \times 10^{+2} = 6,461,446.86 \times 10^{-1} \approx 646.\mathrm{L} \times 10^{+3}$$

The coefficients of 987.L × 10−3 and 6543.H × 10+2 in the example above contain three and four significant digits, respectively. The product coefficient, with various approximations of 0.L and 0.H, reaches more than seven digits. The product's remaining numbers are incorrect and ought to be thrown away, but the three most important ones are relevant. As a result, the exponent has to be increased and the product coefficient needs to be moved to the right. Generally speaking, the product coefficient should only have r = min(m, n) digits given two decimal values, x and y, having coefficients with m and n significant digits. The

remaining numbers ought to be changed because they are useless.

An technique for multiplying two decimal values, x and y, is shown in Figure 7. This algorithm can be implemented in hardware or software. The first step extracts each of the x and y fields. In Step 2, Cu and Cv are created by injecting Fx and Fy into Cx and Cy. In order to assess the precision of the outcome, it further counts the maximum leading zeros (LZ) in the coefficients Cx and Cy when an input is imprecise. In Step 3, the product sign Sr and the biased exponent Ep of Cp = Cu × Cv are calculated. The product Cp is computed in step 4. Step 5 is the computation of LZp, or the number of leading zeros in Cp. Additionally, it moves the product Cp to the right in order to get a result coefficient Cr that is limited to the fewest significant digits in Cx and Cy. As a result, there is an inexact flag (Inx) that shows if every shifted-out digit is non-zero, together with a shifted significand ({Cr, Fr}). Step 6 manages extraordinary inputs and generates extraordinary outcomes. The result R is packed and encoded in step 7. Assuming x = -0017652.y = +0145678 and H × 10−2.Cu = {0017652, 7} and Cv = {0145678, 2} for L × 10−3. The leading zero maximum is LZ = 2. The exponent of the result is Ep = −7+ bias, the product is Cp = 0,000,257,161,356,114, and the result sign is Sr = 1 (negative). LZp = 4 is the number of leading zeros in Cp, while SA = 7 is the shift amount. {Cr, Fr, Inx} = {0025716, 1, 1} is the result of shifting Cp seven digits to the right. The exponent is then increased to become Er = 0+ bias. Finally, we get R = -0025716.L × 100.

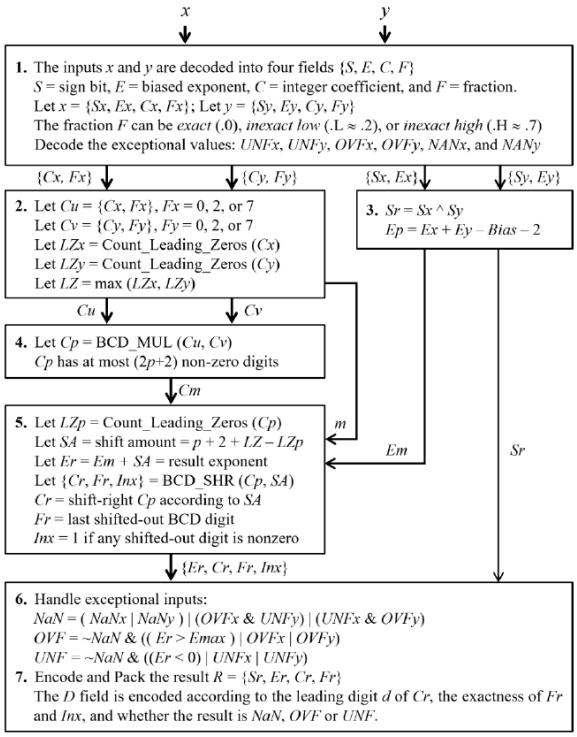


FIGURE 7. multiplying x and y, two decimal integers.

F. DECIMAL DIVISION

The significands of two finite decimal floating-point values, x and y, are divided by their respective significands, and the exponents are then subtracted. Next, the outcome is normalized to the necessary level of accuracy. The result coefficient should be limited to r = min(m, n) significant digits, just like in multiplication, where m and n represent the number of significant digits in Cx and Cy. After injecting Fy, the dividend becomes {Cx, Fx, 0... 0}, while the divisor becomes {Cy, Fy}. Fx is followed by n + 1 decimal zeros. A quotient with either (m + 1) or (m + 2) digits is obtained by dividing an integer with (m + n + 2) decimal digits by a divisor with (n + 1) digits.

The division of two imprecise decimal integers using various estimates of 0.L and 0.H is demonstrated in the example below:

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

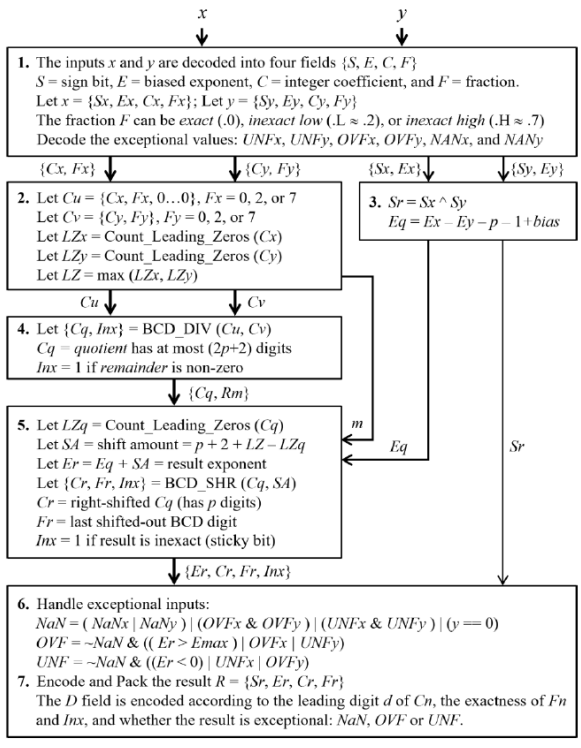


FIGURE 8. dividing x and y, two decimal integers.

An procedure for splitting two decimal values, x and y, is shown in Figure 8. Step 2 creates a coefficient Cu with (2p + 2) decimal digits by injecting Fx (0, 2, or 7) and (p + 1) decimal zeros into Cx, where p is the precision. Fy (0, 2, or 7) is also injected into Cy to create a coefficient Cv with (p + 1) decimal digits. In addition, the maximum number of leading zeros in coefficients Cx and Cy is counted in this step: LZ =

max(LZx, LZy). This is required to ascertain the accuracy of the outcome when an input operand is imprecise.

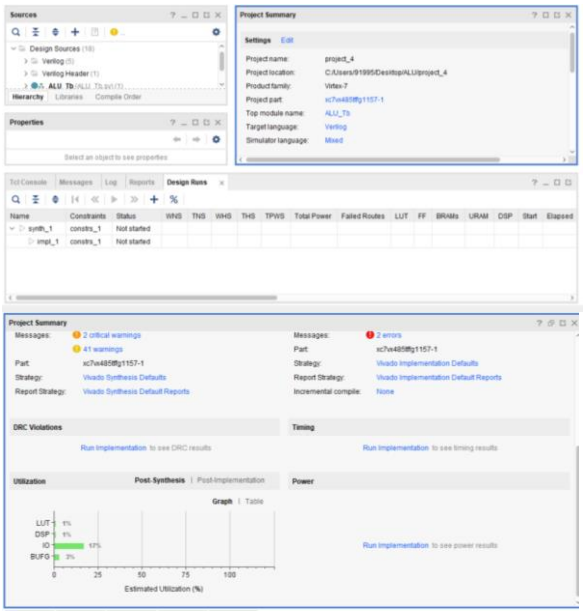
The sign of the result Sr=Sx ∧ Sy is calculated in step three, where ∧ denotes the XOR operator. Additionally, it calculates the quotient's biased exponent: Ex − Ey − p − 1 + Bias = Equation. The decimal coefficients are divided in step 4: Cu/Cv equals Cq. In this phase, a quotient Cq with a maximum of 2p + 2 decimal digits and an Inx flag indicating an imprecise division (Inx might be 0 or 1) are produced.

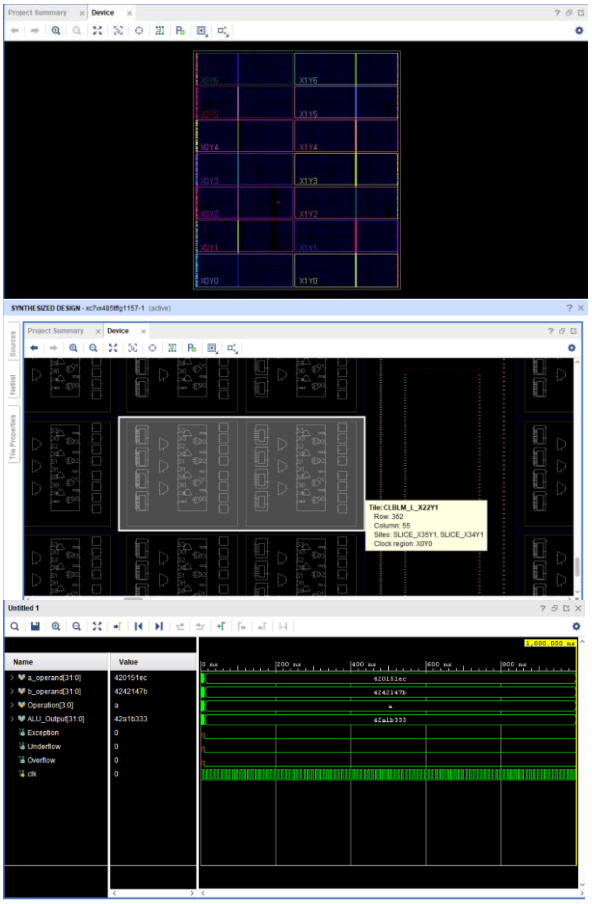
In Step 5, the number of leading zeros in Cq is calculated, or LZq. The shift amount is calculated using the precision p, LZ, and LZq as follows: SA = (p + 2 + LZ − LZq). Er = Eq + SA is the biased exponent that is computed, and the Cq quotient is shifted to the right. The resultant output consists of a sticky inexact flag Inx that specifies whether every shifted-out digit is nonzero, and a shifted significand {Cr, Fr} = BCD_SHR (Cq, SA). There are p decimal digits in the result coefficient Cr. Fr, the resultant fraction, is one decimal place. Step 6 identifies overflow and underflow and deals with unusual inputs.

In step seven, the result R is encoded and packed along with the significand {Cr, Fr}, exponent Er, and sign bit Sr.

For instance, think about dividing x = -6257652.y × 10−2 = + 9815678= H.L × 10−5. Next, Cv = {9815678, 2} and Cu = {6257652, 7, 00000000}. There are no leading zeros in Cx and Cy, as shown by LZx = LZy = 0 and LZ = 0. With Sr=1, Eq = −2 + 5−8=−5+Bias. Inx = 1 and Cq = 0000000063751608. LZq = 8, Er = −4 + Bias, and SA = 7 + 2 + 0 − 8 = 1 are the shift amounts. After that, Cq is moved to the right to create {Cr, Fr} = {6375160, 8}. The calculated outcome is R = -63752060.H × 10−4.

SIMULATION RESULTS:





CONCLUSION

The input domain of floating-point expressions can have a significant impact on them. In order to magnify the inaccuracy and highlight the flaws in the IEEE 754 standard, the inputs listed in Table 5 were chosen. The correct result, calculated with 128-bit decimal arithmetic, is displayed in the first column. Afterwards, the integer coefficients are decreased to a maximum of 16 decimal places, which corresponds to the 64-bit binary and decimal floating-point values' precision. This results in an adjustment of the exponent. If the genuine result has more than 16 decimal places, a fraction is utilized.

The second and third columns display the rounded float64 and decimal64 values together with their corresponding mistakes. The final column displays the DFP64 inexact result. The approximations for the .L and .H are 0.2 and 0.7, respectively. The calculated result is displayed below the relative and ULP errors. The calculations for decimal64 and float64 are rounded to the nearest. These examples do not include the rounding tie scenario. Conclusions can be made based on Table 5's data. Arithmetic and floating-point numbers in binary are often less precise than those in decimal. This is explained by the decimal input fractions' imprecise binary representation. The second conclusion is that incorrect bits and digits are propagated during calculation by the 16-digit decimal64 coefficient and the 53-bit float64 significand. This is demonstrated by the ULP error,

which increases exponentially with the amount of incorrect digits. This results from injecting incorrect zero bits or digits during the normalization of significands with leading zero bits or digits to optimize accuracy in the IEEE 754 floating-point arithmetic procedures. In my job, however, this is not permitted when the outcomes or operands are not exact. It should only be acceptable to normalize significands with leading zero digits if the outcome is precise. The ULP error is therefore minimized, as Table 5 for DFP64 illustrates.

The final conclusion, which is illustrated in Table 5, is that inexact arithmetic warns users when important digits are lost in real-time computations. If inexact floating-point numbers have an explicit representation, this may be easily identified by the programmer. The IEEE 754 numbers and arithmetic operations, which have been directly implemented in hardware and accepted by programming languages and numeric analytic tools for decades [42], nonetheless lack this imprecise representation, which necessitates the real-time identification of significant calculation mistakes as detailed in this work.

This paper presents only a portion of the work that has been done. Further developments include a more thorough error analysis for imprecise computing in real time. Work is being done on a hardware implementation of arithmetic operations with both accurate and inexact floating-point integers.

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/languagereference/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/decimaland-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 multifunction 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.