Decimal Arithmetic FAQ
Part 1 – General Questions 2024.04.24

Contents [back to FAQ contents]

Why the increasing emphasis on decimal arithmetic?

Most people in the world use decimal (base 10) arithmetic. When large or small values are needed, exponents which are powers of ten are used. However, most computers have only binary (base two) arithmetic, and when exponents are used (in floating-point numbers) they are powers of two.

Binary floating-point numbers can only approximate common decimal numbers. The value **0.1**, for example, would need an infinitely recurring binary fraction. In contrast, a decimal number system can represent 0.1 exactly, as one tenth (that is, 10^{-1}). Consequently, binary floating-point cannot be used for financial calculations, or indeed for any calculations where the results achieved are required to match those which might be calculated by hand. See below for examples.

For these reasons, most commercial data are held in a decimal form. Calculations on decimal data are carried out using decimal arithmetic, almost invariably operating on numbers held as an integer and a scaling power of ten.

Until recently, the IBM z900 (mainframe) computer range was the only widely-used computer architecture with built-in decimal arithmetic instructions. However, those early instructions work with decimal integers only, which then require manually applied scaling. This is error-prone, difficult to use, and hard to maintain, and requires unnecessarily large precisions when both large and small values are used in a calculation.

The same problems existed in the original Java decimal libraries; the problems were so severe that IBM raised a Java Specification Request (JSR), formally endorsed by a wide range of other companies, detailing the problems. This has now been implemented (see the Final Draft for details) and was shipped with Java 1.5 in 2004.

Both problems (the artifacts of binary floating-point, and the limitations of decimal fixed-point) can be solved by using a decimal floating-point arithmetic. This is now available in both hardware and software, as described on the **General Decimal Arithmetic** page, and is standardized in the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

What problems are caused by using binary floating-point?

Binary floating-point cannot exactly represent decimal fractions, so if binary floating-point is used it is not possible to guarantee that results will be the same as those using decimal arithmetic. This makes it extremely difficult to develop and test applications that use exact real-world data, such as commercial and financial values.

Here are some specific examples:

1. Taking the number 9 and repeatedly dividing by ten yields the following results:

Decimal	Binary
`0.9`	`0.9`
`0.09`	`0.089999996`
`0.009`	`0.0090`
`0.0009`	`9.0E-4`
`0.00009`	`9.0E-5`
`0.000009`	`9.0E-6`
`9E-7`	`9.0000003E-7`
`9E-8`	`9.0E-8`
`9E-9`	`9.0E-9`
`9E-10`	`8.9999996E-10`

   Here, the left hand column shows the results delivered by decimal floating-point arithmetic (such as the BigDecimal class for Java or the decNumber C package), and the right hand column shows the results obtained by using the Java `float` data type. The results from using the `double` data type are similar to the latter (with more repeated 9s or 0s).

   Some problems like this can be partly hidden by rounding (as in the C printf function), but this confuses users. Errors accumulate unseen and then surface after repeated operations.

   For example, using the Java or C `double` datatype, 0.1 × 8 (a binary multiple) gives the result 0.8000000000000000444089209850062616169452667236328125 but 0.1 added to itself 8 times gives the different answer 0.79999999999999993338661852249060757458209991455078125. The two results would not compare equal, and further, if these values are multiplied by ten and rounded to the nearest integer below (the 'floor' function), the result will be 8 in one case and 7 in the other.

   Similarly, the Java or C expression `(0.1+0.2==0.3)` evaluates to *false*.

2. Even a *single* operation can give very unexpected results. For example:

   - Consider the calculation of a 5% sales tax on an item (such as a $0.70 telephone call), which is then rounded to the nearest cent.

     Using double binary floating-point, the result of 0.70 x 1.05 is 0.73499999999999998667732370449812151491641998291015625; the result should have been 0.735 (which would be rounded up to $0.74) but instead the rounded result would be $0.73.

     Similarly, the result of 1.30 x 1.05 using binary is 1.3650000000000002131628207280300557613372802734375; this would be rounded up to $1.37. However, the result should have been 1.365 – which would be rounded *down* to $1.36 (using "banker's rounding").

     Taken over a million transactions of this kind, as in the 'telco' benchmark, these systematic errors add up to an overcharge of more than $20. For a large company, the million calls might be two-minutes-worth; over a whole year the error then exceeds $5 million.

   - Using binary floating-point, calculating the remainder when 1.00 is divided by 0.10 will give a result of exactly 0.0999999999999999500399638918679556809365749359130859375 (here's a test program showing this). Even if rounded this will still give a result of 0.1, instead of 0, the result obtained if decimal encoding and arithmetic are used.

3. Binary calculations can make apparently predictable decisions unsafe. For example, the Java loop:

```
for (double d = 0.1; d <= 0.5; d += 0.1) System.out.println(d);
```

   displays five numbers, whereas the similar loop:

```
for (double d = 1.1; d <= 1.5; d += 0.1) System.out.println(d);
```

   displays only four numbers. (If *d* had a decimal type then five numbers would be displayed in both cases.)

4. Rounding can take place at unexpected places (power-of-two boundaries rather than power-of-ten boundaries). For example, the C program:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  double x = 0.49999999999999994;
  int    i = (int)(x+0.5);
  printf("%0.17f  %d\n", x, i);
  return 0;
  }
```

   uses a decades-old idiom for rounding a floating-point number to an integer. Unfortunately in this particular case the value of *x* is rounded up during the addition, so *i* ends up with the value 1 instead of the expected zero. (Thanks to Steve Witham for this example.)

5. In engineering, exact measurements are often kept in a decimal form; processing such values in binary can lead to inaccuracies. This was the cause of the Patriot missile failure in 1991, when a missile failed to track and intercept an incoming Scud missile. The error was caused by multiplying a time (measured in tenths of a second) by 0.1 (approximated in binary floating-point) to calculate seconds. See Douglas Arnold's page for more details.

6. Many programming languages do not specify whether literal constants are represented by binary or decimal floating-point values (even when written using decimal notation). Hence simple constant expressions can give different results depending on the implementation. For example, the C test program:

```
#include
int main(int argc, char *argv[]) {
  if (argc>0) printf("Running: %s\n", argv[0]);

  if (0.1+0.2==0.3) printf("Literals are decimal\n");
             else printf("Literals are binary\n");
  return 0;
  }
```

   will display a different result depending on the implementation (usually '`Literals are binary`').

7. Finally, there are legal and other requirements (for example, in Euro regulations) which dictate the working precision (in decimal digits) and rounding method (to decimal digits) to be used for calculations. These requirements can only be met by working in base 10, using an arithmetic which preserves precision.

Do applications actually use decimal data?

Yes. Data collected for a survey of commercial databases (the survey reported in IBM Technical Report TR 03.413 by A. Tsang & M. Olschanowsky) analyzed the column datatypes of databases owned by 51 major organizations. These databases covered a wide range of applications, including Airline systems, Banking, Financial Analysis, Insurance, Inventory control, Management reporting, Marketing services, Order entry, Order processing, Pharmaceutical applications, and Retail sales.

In all, 1,091,916 columns were analysed. Of these columns, 41.8% contained identifiably numeric data (53.7% contained 'char' data, with an average length of 8.58 characters, some of which will have contained numeric data in character form).

Of the numeric columns, the breakdown by datatype was:

Type	Columns	percent
Decimal	251,038	55.0
SmallInt	120,464	26.4
Integer	78,842	17.2

Integer	78,842	17.3
Float	6,180	1.4

These figures indicate that almost all (98.6%) of the numbers in commercial databases have a decimal or integer representation, and the majority are decimal (scaled by a power of ten). The integer types are often held as decimal numbers, and in this case almost all numeric data are decimal.

Why are trailing fractional zeros important?

Trailing fractional zeros (for example, in the number 2.40) are ignored in binary floating-point arithmetic, because the common (IEEE 754) representation of such numbers cannot distinguish numbers of equal value.

However, decimal numbers can preserve the zero in the example because they are almost always represented by an integer which is multiplied by an exponent (sometimes called a scale). The number 2.40 is represented as 240 times 10^{-2} (`240E-2`), which does encode the final zero.

Similarly, the decimal addition 1.23 + 1.27 (for example) typically gives the result 2.50, as most people would expect.

In some programing languages and decimal libraries, however, trailing fractional zeros are either removed or hidden. While sometimes harmless, serious and material problems are caused by this:

- End users expect trailing zeros to be preserved in addition, subtraction, and multiplication (as in the 1.23 + 1.27 example). This is especially true for financial and commercial applications. If the zeros are not preserved, users are surprised, frustrated, and lose confidence in the application.
- Currency calculations are often defined in terms of a given precision (for instance, regulations dictate that Euro exchange rates must be quoted to 6 digits). All the digits must be present, even if some trailing fractional digits are zero. For example, 1 Euro = 340.750 Greek drachmas.
- The original unit of a measurement is often indicated by means of the number of digits recorded. If trailing fractional zeros are removed, measurements and specifications may appear to be more vague (less precise) than intended, and information (the datatype of the number) is lost. For example, the length of a steel beam might be specified as 1.200 meters; if this value is altered to 1.2 meters then a contractor might be entitled to provide a beam that is within 5 centimeters of that length, rather than measured to the nearest millimeter.

  As another example, consider driving directions. “Turn left after 7 miles” and “Turn left after 7.0 miles” lead to different driver behavior.
- Similarly, geographical survey and mapping records indicate the precision of measurements using fractional trailing zeros as necessary. Loop closure software makes use of this information for distributing errors. If fractional zeros are lost then precisely measured segments will appear imprecise; they will be over-adjusted and the final result will be corrupt.
- Most numeric data in databases have a decimal data type (see above), which is characterized by a given scale (number of fractional digits). This scale needs to be preserved when data are moved in or out of the database so that ‘round trips’ are possible without loss of information.
- In medical practice, trailing zeros are used where required to demonstrate the level of precision of a value being reported, such as for laboratory results, imaging studies that report size of lesions, or catheter/tube sizes (however, a single trailing zero immediately following a decimal point must not be used in medication orders or other medication-related documentation).
- When decimal numbers are being used simply as labels, and are not expected to be subject to arithmetic, programmers often (perhaps incorrectly) use a decimal datatype to store them. Here, trailing fractional zeros are often significant – section 3.20 of a book, for example, is distinct from section 3.2.
- Requiring that fractional trailing zeros be removed can cause significant extra processing, especially if the coefficient of a number is represented by a binary integer. In this case an division by some power of ten, or the equivalent, is required to remove the trailing zeros. Similarly, this may cause numbers to become unaligned (for example, prices in dollars and cents) which slows addition or subtraction considerably.

For these reasons, decimal libraries must not assume that only the numerical value of a number is significant; they must also take care to preserve the full precision of the number where appropriate.

See also “Why are the encodings unnormalized?” and “Why is the arithmetic unnormalized?”.

How much precision and range is needed for decimal arithmetic?

This depends, of course, on the application. Let’s consider financial applications. For these applications, values similar to the Gross National Product (GNP) of large countries give an idea of the largest values needed to represent a good range of currency values exactly.

Two examples of these are the USA (in cents) and Japan (in Yen). Both of these need values up to about 10^{15} (a recent GDP number for the USA was $8,848,200,000,000.00, and for Japan ¥554,820,000,000,000). If stored in a fixed-point representation, these would require 15 digits.

When carrying out calculations, more precision and range is needed because currency values are often multiplied by very precise numbers. For example, if a daily interest rate multiplier, *R*, is (say) 1.000171 (0.0171%, or roughly 6.4% per annum) then the exact calculation of the yearly rate in a non-leap year is *R* raised to the power of 365. To calculate this to give an exact (unrounded) result needs 2191 digits (both the Java BigDecimal class and the C decNumber package easily handle this kind of length).

It’s useful to be able to do this kind of calculation exactly when checking algorithms, generating testcases, and comparing the theoretical exact result with the result obtained with daily rounding.

In production algorithms, it is rarely, if ever, necessary to calculate to thousands of digits. Typically a precision of 25-30 digits is used, though the ISO COBOL 2002 standard requires 32-digit decimal floating-point for intermediate results.

As a more concrete example of why 20 digits or more are needed, consider a telephone company’s billing program. For this, calls are priced and taxed individually (that is, tax is calculated on each call). Taxes (state, federal, local, excise, *etc.*) are often apportioned and are typically calculated to six places (*e.g.*, $0.000347) before rounding. Data are typically stored with 18 digits of precision with 6 digits after the decimal point (but even with this precision, a nickel call immediately has a rounding error).

(See elsewhere for a definition of *precision*.)

What rounding modes are needed for decimal arithmetic?

The three most important are

1. *round-half-even*, where a number is rounded to the nearest digit, and if it is exactly half-way between two values then it is rounded to the nearest even digit. This method ensures that rounding errors cancel out (on average), and is sometimes called “banker’s rounding”. For example, 12.345 rounded to four digits is 12.34 and 12.355 rounded to four digits is 12.36. This is also called *round to nearest, ties to even*.
2. *round-half-up*, where a number is rounded to the nearest digit, and if it is exactly half-way between two values then it is rounded to the digit above. Here, 12.345 rounded to four digits is 12.35 and 12.355 rounded to four digits is 12.36. This is also called *round to nearest, ties away from zero*.
3. *round-down*, where a number is truncated (rounded towards zero).

The first of these is commonly used for mathematical applications and for financial applications (other than tax calculations) in most states of the USA. The second is used for general arithmetic, for financial applications in the UK and Europe, and for tax applications in the USA. The third is used for both tax and other calculations, world-wide.

Decimal arithmetic packages often provide more rounding modes than these. See, for example, the decNumber context settings, which include eight rounding modes.

Which programming languages support decimal arithmetic?

Decimal data are extremely common in commerce (55% of numeric data columns in databases have the decimal datatype, see above), so almost all major programming languages used for commercial applications support decimal arithmetic either directly or through libraries. (A notable and regrettable exception is JavaScript/JScript.)

Almost all major IBM language products support decimal arithmetic, including C (as a native data type, on System z machines), C++ (via libraries), COBOL, Java (through the Sun or IBM BigDecimal class), OS/400 CL, PL/I, PL/X (on AS/400), NetRexx, PSM (in DB2), Rexx, Object Rexx, RPG, and VisualAge Generator. Many other languages also support decimal arithmetic directly, including Microsoft’s Visual Basic and C# languages (both of which provide a floating-point decimal arithmetic using the .Net runtime Decimal class). Decimal arithmetic packages are available for most other languages, including Eiffel and Python.

It is certain that in the future more languages will add decimal datatypes; the ISO JTC1/SC22/WG14 C and WG21 C++ committees are preparing Technical Reports which add the decimal types to those languages, and this will make it much simpler for other languages to add similar support. The GCC and IBM XL C compilers have already implemented the support described in the C technical report.

For a list of implementations of the new decimal types, see: `http://speleotrove.com/decimal/#links`

What disadvantages are there in using decimal arithmetic?

Decimal numbers are traditionally held in a binary coded decimal form which uses about 20% more storage than a purely binary representation.

More compact representations can be used (such as the Densely Packed Decimal and Chen-Ho encodings) but these do complicate conversions slightly (in hardware, compressing or uncompressing costs about two gate delays). (Pure binary integers can also be used, as in the Java BigDecimal class, but this makes rounding, scaling, conversions, and many other decimal operations very expensive.) Even with these more efficient representations, decimal arithmetic requires a few extra bits (an extra digit) to achieve the same accuracy as binary arithmetic.

Calculations in decimal can therefore require about 15% more circuitry than pure binary calculations, and will typically be a little slower. However, if conversions would be needed to use a binary representation because the data are in a decimal base then it can be considerably more efficient to do the calculations in decimal.

Some properties that hold for binary do not hold for any other base. For example, $(d \div 2) \times 2$ gives d in binary (unless there is an underflow), but with base 10 it might not if d is full precision and $d \div 2$ is Inexact.

Currently, binary floating-point is usually implemented by the hardware in a computer, whereas decimal floating-point is implemented in software. This means that decimal computations are slower than binary operations (typically between 100 and 1000 times slower).

Even using the BCD decimal integer support in the IBM System z, the simulation of scaled or floating-point arithmetic has a significant overhead (mostly in keeping track of scale, and also partly because z900 decimal arithmetic uses Store-to-Store, not Register-to-Register, instructions).

The new hardware support in the Power6 processor (and in the expected z6 processor) is much faster than software, but is still somewhat slower than binary floating-point (for details of the hardware implementations, see the General Decimal Arithmetic page). Binary floating-point is therefore widely used for ‘numerically intensive’ work where performance is the main concern. This is likely to continue to be the case for some time.

For more details, see the papers in the Choice of base section of the Bibliography.

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`

Decimal Arithmetic FAQ
Part 2 – Definitions 2024.04.24

Contents [back to FAQ contents]

What does *Algorism* mean?

Algorism is the name for the Indo-Arabic decimal system of writing and working with numbers, in which symbols (the ten digits 0 through 9) are used to describe values using a place value system, where each symbol has ten times the weight of the one to its right.

This system was originally invented in India in the 6th century AD (or earlier), and was soon adopted in Persia and in the Arab world. Persian and Arabian mathematicians made many contributions (including the concept of the decimal fractions as an extension of the notation), and the written European form of the digits is derived from the *ghubar* (sand-table or dust-table) numerals used in north-west Africa and Spain.

The word *algorism* comes from the Arabic *al-Kowarizmi* ("the one from Kowarizm"), the cognomen (nickname) of an early-9th-century mathematician, possibly from what is now Khiva in western Uzbekistan. (From whose name also comes the word *algorithm*.)

See also *Wikipedia: Algorism*.

What does *precision* mean?

In English, the word *precision* generally means the state of being precise, or defines the repeatability of a measurement *etc.*

In computing and arithmetic, it has some more specific (and different) meanings (the first is the most common):

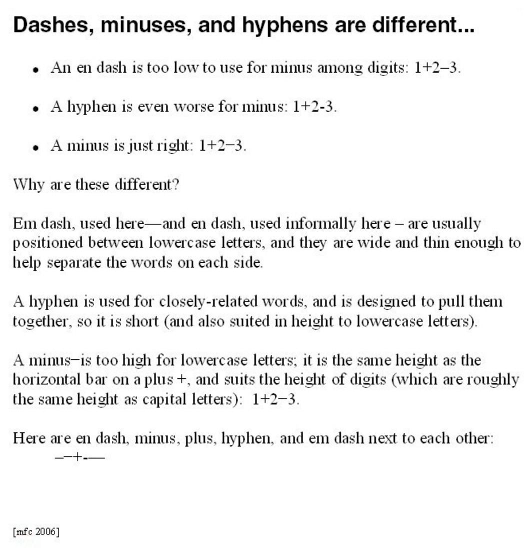
1. The number of significant digits in a number (leading zeros are not considered significant). For example, the following numbers all have three significant digits:
   ```
   123   1.23   1.50   0.00123   1.23E+22
   ```
   and are said to have a precision of 3. The number zero is a special case; because it requires one digit to indicate its presence it is usually considered to have a precision of 1.

   A calculation which rounds to three digits is said to have a *working precision* or *rounding precision* of 3.
2. The units of the least significant digit of a measurement. For example, if a measurement is 17.130 meters then its precision is millimeters (one unit in the last place, or *ulp*, is 1mm).
3. (In some programming languages and databases.) The number of decimal places after the decimal point in a fixed-point number. To avoid confusion, this usage is best avoided.

See also *Wikipedia: Precision*.

What is the difference between a *minus* and an *en dash*?

A dash or a hyphen is vertically positioned lower than a minus because the latter is designed to be used alongside digits (which are typically the same height as capital letters) whereas dashes usually appear between lowercase letters.



Dashes, minuses, and hyphens are different...

- An en dash is too low to use for minus among digits: 1+2–3.
- A hyphen is even worse for minus: 1+2-3.
- A minus is just right: 1+2−3.

Why are these different?

Em dash, used here—and en dash, used informally here – are usually positioned between lowercase letters, and they are wide and thin enough to help separate the words on each side.

A hyphen is used for closely-related words, and is designed to pull them together, so it is short (and also suited in height to lowercase letters).

A minus−is too high for lowercase letters; it is the same height as the horizontal bar on a plus +, and suits the height of digits (which are roughly the same height as capital letters): 1+2−3.

Here are en dash, minus, plus, hyphen, and em dash next to each other:
–−+-—

[mfc 2006]

"A picture is worth a thousand words" (and in this case is the only way to guarantee the differences are shown correctly):

What are Normal numbers, Subnormal numbers, E_{max}, *etc.*?

These terms are derived or extrapolated from the IEEE 754 and 854 standards, and describe the various kinds of numbers that can be represented in a given computer encoding: The answers in Part 5 of the FAQ explain in more detail the meaning of some of the terms described here:

Overflow threshold
: If a calculation results in a number whose magnitude is greater than or equal to the *overflow threshold* it is considered to have overflowed. Under IEEE 854 rules the result will then be either infinity or the largest representable number (depending on the rounding mode).

Underflow threshold
: If a calculation results in a non-zero number whose magnitude would be less than the *underflow threshold* it is considered to have underflowed. The result will then sometimes be a subnormal number (see below), but it may be rounded down to zero or up to the threshold value.

Normal numbers
: Any representable number which is greater than or equal to the underflow threshold and less than the overflow threshold is considered to be a *normal number*. The normal numbers form a balanced, or close to balanced, range (the underflow threshold × the overflow threshold equals either 10 or 100 for decimal encodings).

Largest normal number
: The magnitude of the largest normal number.

Smallest normal number
: The magnitude of the smallest normal number; this is also the *underflow threshold*.

Subnormal numbers
: Non-zero numbers whose magnitude is less than the underflow threshold. These allow for gradual underflow, and are required by IEEE 854.

Supernormal numbers
: Numbers whose magnitude is greater than or equal to the overflow threshold. Encodings do not necessarily support supernormal numbers, and they are not required by IEEE 854.

Maximum representable number
: The magnitude of the largest finite number that an encoding can distinguish.

Minimum representable number
: The magnitude of the smallest (tiniest) non-zero number that an encoding can distinguish. This is the smallest *subnormal number*.

Here's a table illustrating the terms above, just showing positive numbers. The symbols on the left are sometimes used to refer to certain values. Example values are shown on the right are for the 32-bit decimal encoding with 7 digits of precision (*decimal32*).

Symbol	Name	Range	Example
	Overflow threshold	Supernormal number	10E+96
N_{max}	Largest normal number (Maximum representable number)	Normal numbers	9.999999E+96
Unity	One		1 (1E+0)
N_{min}	Smallest normal number (Underflow threshold)		1E-95
	Largest subnormal number	Subnormal numbers	0.999999E-95
N_{tiny}	Smallest subnormal number (Minimum representable number)		1E-101

Note that the example value of N_{tiny} could be written 0.000001E-95. The exponent of N_{max} when written in scientific notation (+96 in the example) characterizes an encoding, and is called E_{max}. The exponent of the smallest normal number, $-E_{max}+1$, is called E_{min}, and the smallest possible exponent seen when a number is written in scientific notation (-101 in the example) is called E_{tiny}. E_{tiny} is $E_{min}-(p-1)$, where *p* is the precision of the encoding (7 in these examples).

For more on the ordering of decimal numbers, see Which is larger? 7.5 or 7.500?

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`

This page was created on 2024.04.24.

Decimal Arithmetic FAQ
Part 3 – Hardware Questions 2024.04.24

Contents [back to FAQ contents]

Why doesn't hardware support decimal arithmetic directly?

Most computer architectures other than 'pure RISC' machines do, in fact, provide some form of decimal arithmetic instructions or support. These include the Intel x86 architecture (used in PCs), the Motorola 68x architecture and its derivatives (used in Apple's earlier machines and the Palm Pilot), the IBM System z (the descendents of the IBM/360 family), the HP PA-RISC architecture, and most general-purpose microprocessors.

However, for all of these machines, only integer decimal arithmetic is supported, and for most the support is limited to decimal adjustment or conversion instructions which simplify decimal operations. These instructions are only accessible through assembly-language programming, and lead to only small performance improvements. In all cases, any scaling, rounding, or exponent has to be handled explicitly by the applications or middleware programmer; a complex and very error-prone task.

The native (hardware) decimal floating-point arithmetic now available in the IBM Power6 processor and expected in the z6 microprocessor makes programming far simpler and more robust, and with much better performance than software (for details of the new hardware implementations, see the General Decimal Arithmetic page).

Why did computers use binary in the first place?

Many early computers (such as the ENIAC, or the IBM 650) were in fact decimal machines. In the 1950s, however, most computers turned to binary representations of numbers as this made useful reductions in the complexity of arithmetic units (for example, a binary adder requires about 15% less circuitry than a decimal adder). This reduction in turn led to greater reliability and lower costs.

Storing decimal integers in a simple binary coded decimal (BCD) form, rather than a pure binary form, also uses up to 20% more storage than the binary form, depending on the coding used.

Later, it was also shown that binary floating-point allows simpler error analysis, and for a given number of bits gives more accurate results for certain mathematical operations.

Decimal arithmetic, therefore, is inherently less efficient than binary arithmetic, and at the time this justified the switch to binary floating-point arithmetic (just as a two-digit representation for the year in a date was justifiable at the time). However, the programming and conversion overheads and other costs of using binary arithmetic suggest that hardware decimal arithmetic is now the more economical option for most applications.

Surely software emulation of decimal arithmetic is fast enough?

No, it is not. The performance of existing software libraries for decimal arithmetic is very poor compared to hardware speeds. In some applications, the cost of decimal calculations can exceed even the costs of input and output and can form as much as 90% of the workload. See "The 'telco' benchmark" for an example and measurements on several implementations.

Binary floating-point emulation in software was unacceptable for many applications until hardware implementations became available; the same is true for decimal floating-point (or even fixed-point) emulation today. Even using the decimal integer instructions on an IBM System z machine only improves fixed-point performance by about a factor of 10; rounding and scaling in software adds significant overhead.

Complaints about the performance of decimal arithmetic are extremely common. Software emulation is 100 to 1000 times slower than a hardware implementation could be. For example, a JIT-compiled 9-digit BigDecimal division in Java^{TM} 1.4 takes over 13,000 clock cycles on an Intel Pentium. Even a simple 9-digit decimal addition takes at least 1,100 clock cycles. In contrast, a native hardware decimal type could reduce this to a speed comparable to binary arithmetic (which is 41 cycles for integer division on the Pentium, or 3 for an addition).

The benefits are even larger for multiplications and divisions at the higher precisions often used in decimal arithmetic; a 31-digit division can take 20–110 times as long as a 9-digit addition.

Do 'hand-held' calculators use decimal arithmetic?

Yes. The first microprocessor-based electronic calculator, the Busicom (actually a desk-top machine), used its Intel 4004 to implement decimal arithmetic in 1970.

Later, the Hewlett Packard HP-71B calculator used a 12-digit internal decimal floating-point format (expanded to 15 digits for intermediate calculations), to implement the IEEE 854 standard.

Today, the Texas Instruments TI-89 and similar calculators use a 14-digit or 16-digit Binary Coded Decimal internal floating-point format with a three digit exponent. HP calculators continue to use a 12-digit decimal format; Casio calculators have a 15-digit decimal internal format.

These all use software to implement the arithmetic, as single-calculation performance is not usually an issue. Oddly, most calculators discard trailing fractional zeros.

Has any company formally announced hardware decimal floating-point support?

Yes. IBM announced on 18 April 2007 hardware decimal floating point facilities for IBM z9 EC and z9 BC:

> "IBM is implementing hardware decimal floating point facilities in System z9. The facilities include 4-, 8-, and 16-byte data formats, an encoded decimal (base-10) representation for data, instructions for performing decimal floating point computations, and an instruction which performs data conversions to and from the decimal floating point representation.
>
> Base 10 arithmetic is used for most business and financial computation. To date, floating point computation used for work typically done in decimal arithmetic has involved frequent necessary data conversions and approximation to represent decimal numbers. This has made floating point arithmetic complex and error prone for programmers using it in applications where the data is typically decimal data.
>
> Initial software support for hardware decimal floating point is limited to High Level Assembler support running in z/OS and z/OS.e on System z9. z/OS V1.9 will provide support for hardware decimal floating point instructions and decimal floating point data types in the C and C++ compilers as a programmer-specified option. Support is also provided in the C Run Time Library and the dbx debugger. This function is also supported by z/VM V5.2 and later for guest operating system use. There is no support available for machines earlier than System z9. Refer to the Software requirements section."

Since then, IBM has also announced support for decimal floating-point in the Power6 processors, and has released details of the decimal floating-point unit in the z6 microprocessor. For details of these hardware implementations, see the General Decimal Arithmetic page.

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`

Decimal Arithmetic FAQ
Part 4 – Arithmetic Specification Questions 2024.04.24

Contents [back to FAQ contents]

Why is decimal arithmetic unnormalized?

It is quite possible for decimal arithmetic to be normalized, even if the encoding (representation) of numbers is not. A floating-point unit can easily offer both normalized and unnormalized arithmetic, either under the control of a context flag or by a specific 'normalize' instruction (the latter being especially appropriate in a RISC design).

However, for decimal arithmetic, intended as a tool for human use, the choice of unnormalized arithmetic is dictated by the need to mirror manual calculations and other human conventions. A normalized (non-redundant) arithmetic is suitable for purely mathematical calculations, but is inadequate for many other applications.

Note that the unnormalized arithmetic gives *exactly* the same mathematical value for every result as normalized arithmetic would, but has the following additional advantages.

1. **Unnormalized arithmetic is compatible with existing languages and applications.**
   Decimal arithmetic in computing almost invariably uses scaled integer calculations. For example, the languages COBOL, PL/I, Java, C#, Rexx, Visual Basic and the databases DB2, Oracle, MS SQL Server, and Informix all use this form of computation, as do decimal arithmetic libraries, including decNumber for C, bignum for Perl 6, Decimal in Python 2.4, EDA for Eiffel, ArciMath and IBM's BigDecimal classes for Java, ADAR for Ada, and the X/Open ISAM decimal type.

   A normalized arithmetic cannot duplicate the results of unnormalized arithmetic and so these existing software decimal calculations cannot be replaced by hardware which provided only normalized arithmetic. In up to 27% of cases the resulting coefficient and exponent will be different. This would require that all applications and their testcases be rewritten; an effort comparable to but significantly larger than the 'Year 2000' problem.

   Normalized arithmetic is close to useless for most applications which use decimal data, because so many operations would have to be be recalculated in software in order to give the expected results.

2. **The arithmetic of all existing decimal datatypes can be derived by constraining the unnormalized arithmetic.**
   Decimal arithmetic has a number of flavors:
   - Integer arithmetic: Addition and multiplication are the standard arithmetic, unchanged. Integer division is the general division followed by a truncating rescale(0) (in practice, an early-finish division).
   - Fixed-point arithmetic: addition is unchanged; multiplication and divison are the general operations followed by rescale(n).
   - Variable precision floating-point: all operations are the general arithmetic, either with precision control or with a following re-round operation.
   - Normalized floating-point: all operations are the general operations, followed by a normalize operation (this simply removes trailing zeros while increasing the exponent appropriately; no errors are possible from this operation).
   - Unnormalized floating point: all operations are the general arithmetic.

   All of these types are a subset of the standard type; in a RISC-like implementation it is especially appropriate for the rescale, re-round, and normalize operations to be independent of the general arithmetic.

   There is therefore no need for fixed-point and integer decimal datatypes in addition to the floating-point type, and no conversions are needed when mixed integer, fixed-point, and floating-point types are used in a calculation. (For example, when calculating the product 17 × 19.99, as in 17 items at $19.99.)

   If a normalized arithmetic were used, a separate unnormalized floating-point processing unit would still be needed, and conversions between the two units would be necessary.

3. **Unnormalized arithmetic often permits performance improvements.**
   Common addition and subtraction tasks (*e.g.*, summing a column of prices or costs) need no alignment shift or normalization step. Compare the following sums (unnormalized on the left, normalized on the right):

   ```
     1.23       1.23
     2.50        2.5
   + 1.27     + 1.27
   -------    -------
     5.00          5
   ```

   With a normalized arithmetic, alignment of the operands is needed more often and an extra step is required at the end of every calculation to determine whether normalization is required and then effect it if it is. These unnecessary steps require extra code (in software) or extra circuitry (in hardware). These both increase the costs of calculations, testing, and validation.

   Further, in the unnormalized arithmetic, the calculations of a result coefficient and exponent are independent (except when the result has to be rounded). This independence reduces the complexity of the calculation and permits the two parts of the calculation to be done entirely in parallel.

4. **Gradual underflow is 'free'.**
   Subnormal numbers are simply the low end of the range of unnormalized numbers; they arise naturally out of calculations when required and require no special treatment. This helps performance, testing, *etc.*

   For example, $10000\times10^{E_{tiny}}$ (the smallest normal number if *precision*=5) ÷2 gives $5000\times10^{E_{tiny}}$, a subnormal number, with no special processing.

   In contrast, normalized arithmetic necessarily must treat subnormal values as special cases, which adds complexity and complicates implementation and testing.

5. **Zeros are not special cases.**
   Similarly, results and operands which are zero are treated in exactly the same manner as any other values.

   In a normalized arithmetic, zero must be encoded differently from all other values, so every result or use of zero requires special-case treatment. This requires extra code or circuitry, with the associated testing and validation burden.

6. **Results are easier to predict and to test.**
   Compared to a normalized arithmetic with fractional coefficients, the integer-coefficient arithmetic is simpler. Fractional arithmetic has rules which are different from integer arithmetic (even though the values represented are equivalent). With integer coefficients one set of rules suffices.

   Also, although theoretically equivalent, integer arithmetic has a longer history and is more familiar than left-aligned fractional arithmetic. This may make it easier to find existing proofs for results, *etc.* In general, testing and validation are simplified: all computers already have integer arithmetic, so testing methods can be made consistent with the testing of the integer arithmetic unit (and are well-known and understood).

7. **Rounded numbers are full precision**
   A number which has been rounded will always have the full working precision (except in the case of a subnormal result).

   In particular, this means that if the result of an operation has less than full precision then it was not rounded by that operation; this is particularly useful for checking that adequate precision has been allowed for calculations which must not be rounded.

8. **Conversions to and from existing decimal datatypes are faster.**
   Existing decimal data are invariably held in an unnormalized two-integer format. The standard decimal type is also an unnormalized two-integer format, so conversions are simple and fast, with no exponent adjustment or shifting needed.

9. **Flexibility.**
   An optional normalization step can be omitted or added to suit the application. This could be a normalize instruction or control register bit, depending on the architecture (or, in software, an extra subroutine or method call). When the normalization step is separated in this way, the penalties of normalization can be avoided when desired.

10. **Application design is simpler.**
    If the exponent is implicit in the data or calculations then presentation decisions are often simpler or unnecessary. Verification and testing are often easier, too. For example, currency regulations state that certain rates must be expressed to exactly 6 digits. This is easier to verify if at all times 6 digits are actually stored.

11. **Unnormalized arithmetic results match human expectations.**
    The results from the unnormalized arithmetic defined in the specification are exactly those of Algorism. These are the results which humans are taught to compute, and therefore expect. For example, 1.23 + 1.27 gives 2.50 (2.5 is a surprise to calculator users).

    In general exponents must be preserved; they form part of the type of a number. If two numbers which have the same exponent are added or subtracted, the result is expected to have that same exponent. If normalization is applied, however, the result exponent varies depending on the values of the operands.

    When results are different from those expected, users are surprised, frustrated, and lose confidence in the application.

    Further, when results match the results computed by hand exactly, application test case generation and validation are easier. Testing is often simpler, too, because the operations are essentially integer operations.

What are the advantages of normalization?

1. **Normalization guarantees that the minimum number of digits are involved in a calculation.**
   This can speed multiplies and divides, especially in software.

   However, if operands are random, then only one in ten will have trailing zeros, and only one in a hundred dyadic operations will have both operands with trailing zeros. Any performance advantage of normalizing these is quite possibly negated by the extra alignments and shifts required for normalization. (The result of 27% of multiplications would need an otherwise unnecessary normalization shift, for example.)

2. **Normalization allows more values to be encoded.**
   Potentially 11% more values can be encoded, or a wider exponent range supported, because one digit of the coefficient would only need to take the values 1 through 9 (instead of 0 through 9).

   With the current layouts being discussed, it is not practical to increase the coefficient length, so the benefit is limited to increasing the exponent range. In a 64-bit layout, for example, using a normalized representation would allow increasing the exponent range from ±384 (already larger than the range available in a binary double) to ±456.

   This slight increase in exponent range is unlikely to enable a new class of applications or otherwise significantly improve the usefulness of the arithmetic.

3. **Testing for equality can be a byte-array compare.**
   This can speed up localized processing in databases, *etc.*

   This advantage also applies to unnormalized numbers, provided that they are normalized before storing. For example, this can be achieved using a normalize (or 'store normalized') instruction.

How are zeros with exponents handled?

In scaled integer arithmetic, zero need not be treated as a special case; just like some other numbers, redundant encodings are allowed. All numbers with a coefficient of zero and with any exponent are valid, and (of course) they all have the value zero. (0×10^5 is zero.)

These permitted redundant encodings of zeros mean that, very importantly, the exponent is independent of the coefficient in all calculations which are not rounded.

For instance, consider subtraction. The rule here is simply that the exponent of the result is the lesser of the exponents of the operands and the coefficient of the result is the result of subtracting the coefficients of the operands after (if necessary) aligning one coefficient so its exponent would have the result exponent.

For example:

```
123 - 122 => 1
```

(123 - 122 gives the result 1), and this can also be written (showing the integer coefficient before the E and the exponent after it) as:

```
123E+0 - 122E+0 => 1E+0
```

Now consider the similar calculation, but with exponent `-2` on the two operands instead. The coefficients are used in the same way and give exactly the same result coefficient, and again the exponent of the result is the same as the exponents of the operands:

```
1.23 - 1.22  => 0.01   or:  123E-2 - 122E-2 => 1E-2
```

We follow exactly the same process of calculation even if the result happens to be zero:

```
 123 - 123   => 0      or:  123E+0 - 123E+0 => 0E+0
```

And with exponent `-2` on the two operands, again the process is the same:

```
1.23 - 1.23  => 0.00   or:  123E-2 - 123E-2 => 0E-2
```

Note that we do not have to inspect the result coefficient to determine the exponent. The exponent can be calculated entirely in parallel with, or even in advance of, the calculation of the coefficient. This simplifies a hardware design or speeds up a software implementation.

Similarly, we don't have to inspect the exponent in order to determine if a value of a number is zero; the coefficient determines this.

If a zero result were to be treated as a special case (perhaps forcing the exponent to zero), there would be different paths for addition and subtraction depending on the value of the result.

Further, the exponent of the result might then have to depend on the coefficient of the operands, too. For example, in the sum:

```
1E+3 + yE+0
```

one could argue that if y=0 then its exponent should be ignored, and the answer should therefore be 1E+3. However, if y=1 then the answer would be 1001E+0. With the consistent rule for zero, the exponent of the result would be +0 in both cases (1000E+0 and 1001E+0), and again the same process is used whatever the operands.

The only complication is when the exponent of a zero result would be too large or small for a given representation (after a multiplication or division). In this case it is set (clamped) to the largest or smallest exponent available, respectively (a result of 0 cannot cause an overflow or underflow).

The preservation of exponents on zeros, therefore, simplifies the basic arithmetic operations. It also gives a helpful indication that an underflow may have occurred (after an underflow which rounds to zero, the exponent of the zero will always be the smallest possible exponent, whereas zeros which are the results of normal calculations will have exponents appropriately related to the operands).

Is the decimal arithmetic ‘significance’ arithmetic?

Absolutely not. Every operation is carried out as though an infinitely precise result were possible, and is only rounded if the destination for the result does not have enough space for the coefficient. This means that results are exact where possible, and can be used for converging algorithms (such as Newton-Raphson approximation), too.

To see how this differs from significance arithmetic, and *why* it is different, here’s a mini-tutorial...

First, it is quite legitimate (and common practice) for people to record measurements in a form in which, by convention, the number of digits used – and, in particular, the units of the least-significant-digit – indicate in some way the likely error in a measurement.

With this scheme, **1.23m** might indicate a measurement which is believed to be 1.230000000... meters, plus or minus 0.005m (or “to the nearest 0.01m”), and **1.234m** might indicate one that is ±0.0005m (“to the nearest millimeter”), and so on.

(Of course, the error will sometimes be outside those bounds, due to human error, random events, and so on, but let us assume that is rare.)

This is one of the reasons why a representation of a number needs to be able to ‘preserve trailing zeros’ (that is, record the original power of ten associated with the number). When a number is a record of a measurement, the two numbers: **1.000** and **1** are in some sense different. The first perhaps records a measurement within ±0.0005, and the other could be ±0.5 (exactly what they mean depends on the context, of course).

This is the distinction that is lost in a normalized representation (which would record both of these as 1).

Now let us consider arithmetic on these two numbers, in a scientific or other context, and in particular let’s consider adding 0.001 to each.

For the first case (1.000+0.001) we would expect the answer 1.001.

For the second case (1 + 0.001, where the 1 is understood to be ±0.5) it can be argued that the error bounds associated with the 1 means that the 0.001 is irrelevant (‘swamped by the noise’). So what happens when we attempt this addition?

- we could raise an error, because we are trying to do an unreasonable calculation
- we could ‘ignore’ the 0.001 and return 1
- we could carry out the calculation exactly – giving 1.001 – and then round to a working number of digits (say 2) which would also result in the 0.001 being ‘ignored’.
- we could carry out the calculation exactly – giving 1.001 – and let the recipient of the result determine what to do with it (one of the first three, possibly).

Note that the first three of these actions only make sense if we *know* we are dealing with measurements (with their implied error bounds). We must not take these actions in the analogous case of adding one thousand dollars ($1E+3) to a bank account with a balance of one dollar and five cents (which must give the result $1001.05).

Given that an addition operator does *not* know whether a number is a measurement or an exact quantity then it has no option other than to return the exact answer (the fourth case). How this result is interpreted is up to the application, and that application is at liberty to apply some rounding rule or to treat the result as exact, as appropriate.

So, default operations must treat calculations as exact. But could we not have a context setting of some kind and do another kind of arithmetic where the precision (significance) of the result depends on the precision of the operands?

Let us consider the most popular of such arithmetics. It’s called *significance arithmetic* and is sometimes advocated in experimental science (particularly in the field of Chemistry).

Significance arithmetic is essentially a set of ‘rules of thumb’. The primary rule is that a result of a calculation must not be given to more digits than the less precise of its operands. This and its other rules ‘work’ in the sense that they alert a student to a situation in which he or she must take care, but they are *not* a valid arithmetic on measurements.

Consider a simple example in which two measurements are added together, such as 2.3 + 7.2 (and let’s assume the true values really are in the ranges 2.25 through 2.35 and 7.15 through 7.25). From the rules of ’significance arithmetic’, a result should have the same precision as the least precise of the operands, so in this case the result should be 9.5.

But, if we continue to assume the rule that the last digit indicates the error bounds, this would suggest that the result is 9.5 ±0.05. However, both the original measurements could err in the same direction, so it is clear that in fact the final total could be anywhere in the range 9.4 through 9.6 (that is, 9.5 ±0.1, not ±0.05). So this rule is over-optimistic after the very first calculation (and compounds with each subsequent calculation), and so we cannot apply the rule for more than the first calculation in a sequence.

But the problems are worse than this, because with measurements the likely error is not uniformly distributed between two bounds. As Delury pointed out (In *Computation with Approximate Numbers*, Daniel B. Delury, The Mathematics Teacher 51, pp521-530, November 1958):

> The statement that a length of something or other is 11.3 inches, to the nearest tenth of an inch, is either trivial or not true. The statement that a length of something or other is 11.3 inches, with a standard deviation of 0.2 inch, *is* a meaningful statement.

(This is because the distribution of errors is not a rectangle but a normal curve – consider a measurement of something about half way between 11.3 and 11.4.)

He then goes on to quote a practical example, of a real experiment where 10 measurements are made of the breaking strain of a wire; the 10 measurements vary from 568 through 596, all three figures. The sum of the 10 measurements is 5752, giving an average (to three figures) of 575 which, according to the rules of significance arithmetic means that the result is 575 ±0.5.

But he also calculates the standard deviation of the average of the measurements (charitably assuming each individual measurement is infinitely accurate), which is 8.26, meaning that “with near certainty” the true mean lies in the range 575 ±9. In other words, the significance arithmetic has given a grossly false estimate of the bounds in which the true mean will lie while at the same time losing a digit which indicates the center point of any result distribution.

The underlying problem here is that a convention for recording measurements does not follow through to an arithmetic on measurements, however much one might wish it to. One has to apply the theory of errors, or use some other technique (such as interval arithmetic).

Delury gives some specific advice:

> Two questions require answers. “How shall [students] carry out their arithmetic and how shall they present the results of their calculations?”
> The answers are easily given. In their arithmetic, all numbers are to be treated as exact, with the proviso that if the number of decimal places becomes unduly large, some of them may be eliminated by rounding off in the course of the calculation. The final answer should be rounded off to a reasonable number of decimals.
> I am sure we would, all of us, like to have something more definite than this, but the fact is that there are no grounds for definiteness.

So, in summary:

- there are many reasons why people need or want to preserve both the exponent (scale) of a number (which defines the units of the last place) and the precision of the integral number of those units which represents a particular value
- an arithmetic processor cannot determine which, if any, of those reasons is in effect (is the number a currency, a measurement, a theoretical constant, an integer, or what?), and so the precision of operands must not have an effect on the result
- therefore the only ‘reasonable’ course is to preserve the precision of numbers as far as possible (that is, unless restricted by some physical limit), even though that *might* be ‘over accurate’ for some applications.

And that is what IEEE 754-1985, IEEE 854, and the arithmetic now described in the revised IEEE 754, provide.

Which is larger? 7.5 or 7.500?

Numerically, the decimal numbers 7.5 or 7.500 compare equal, but sometimes it is useful to have a defined *sorting order* for all decimal numbers. The IEEE 754-2008 standard defines a suitable total ordering for decimal numbers (including the special values). In brief, when two finite numbers have the same numeric value but different exponents, the one with the larger exponent is treated as though it were ‘larger’ than the other.

This following table lists a sequence of sample decimal64 values, in the order defined in the IEEE 754-2008 standard. Here, *Nmax* is the largest positive normal number, *Nmin* is the smallest positive normal number, *Ntiny* is the smallest positive subnormal number (the tiniest non-zero). NaNs and signalling NaNs (sNaN) are sorted by ‘payload’ (*e.g.*, NaN10 – a NaN with a payload of 10 – sorts higher than a NaN with a payload of 0 (shown as simply ‘NaN’ in the table).

decimal64 value
NaN10
NaN
sNaN10
sNaN
Infinity
Nmax = 9.999999999999999E+384

75
7.50
7.500
Nmin as 1E−383
Nmin as 1.000000000000000E−383 which is 1000000000000000E−398
Ntiny = 1E−398
0E+4
0
0.0000
−0.0000
−0
−0E+4
−Ntiny = −1E−398
−Nmin as −1.000000000000000E−383 which is −1000000000000000E−398
−Nmin as −1E−383
−7.500
−7.50
−75
−Nmax = −9.999999999999999E+384
−Infinity
−sNaN
−sNaN10
−NaN
−NaN10

When (and why) is exponential notation used in strings?

The specification describes conversions from decimal numbers (which may be in some internal format) to strings which preserve the sign, coefficient, and exponent of finite numbers. Since the exponents of number can be very large, *exponential notation* is used to keep the lengths of the result strings reasonably short. For example, the value of 1 multiplied by 10^{-20} is converted to `1E-20` rather than `0.00000000000000000001`.

If a negative exponent is small enough, however, a number is converted to a string without using exponential notation. The switch point is defined to be such that at most five zeros will appear between the decimal point and the first digit of the coefficient. This definition is an arbitrary choice, in a sense, but the choice of five means that measurements down to microns (for example) avoid using exponential notation.

Measurements are often quoted using a power of ten that is a multiple of three, so other possible choices included two leading zeros and eight; however, experiments carried out when the arithmetic in the Rexx programming language was designed in 1981 showed that eight zeros was too many to count 'at a glance', and two was too few for many applications.

Exponential notation is also used whenever the exponent is positive, to preserve the distinction between (say) `12300` and `123E+2`.

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`

Decimal Arithmetic FAQ
Part 5 – Encoding Questions 2024.04.24

Contents [back to FAQ contents]

How are decimal numbers encoded?

There are many ways of encoding numbers, but here we'll only discuss decimal numbers which are encoded in a series of contiguous bytes (like binary floats and doubles) and which are described by a pair of parameters: a *coefficient* which is multiplied by ten raised to the power of an *exponent*. For information on other forms of decimal encodings, see "How are the parts of decimal numbers encoded?".

The value of a number encoded with these two parameters is *coefficient* × $10^{exponent}$. For example, if the coefficient is 9 and the exponent is 3 then the value of the number is 9000 (9×10^{3}), and if the exponent were -2 then the value would be 0.09 (9×10^{-2}).

(For simplicity, only positive numbers and zero will be described in this answer. Assume that for any of these numbers there is a corresponding negative number, indicated by a separate sign bit.)

In a given encoding of decimal numbers, each of these parameters will have a 'hard limit':

- P_{limit}, the maximum precision of the coefficient. This is the maximum length of the coefficient, in digits. Any result from an operation which needs more digits than this will be rounded to fit. If this rounding caused non-zero digits to be removed, the result is *Inexact*.
- E_{limit}, the maximum *encoded exponent*. The encoded exponent is a non-negative number, in the range 0 through E_{limit}, from which the exponent parameter is calculated by subtracting a *bias*. (This use of a bias makes it easier to compare exponents in a hardware implementation.)

These limits are usually determined by some external factor (often the size of a hardware register). For this discussion, suppose P_{limit}=7 and E_{limit}=191 (we'll use these limits for all the examples below – they are conveniently small and correspond to the actual limits in the IEEE 754-2008 decimal 32-bit format).

Within these limits, there is some flexibility in the way numbers can be encoded. We can choose to treat the digits of the coefficient as an integer (in the range 0 through 9999999) or we can apply a *scale*, which is a constant power of ten by which such a coefficient is divided. (For example, if the scale were 6, the value of the coefficient would be in the range 0 through 9.999999.) Similarly, the *bias* can be varied to change the range of the exponent (for example, if the bias were 90 then the exponent could take the values -90 through +101).

These two parameters (scale and bias) are related. A given encoding (for example a coefficient encoded as 9999999 and an exponent encoded as 90) will have the same value when the scale is 6 and the bias is 90 as when the scale is 0 and the bias is 84. In the first case, the value of the coefficient is 9.999999 and the exponent is 0, and in the second case, the value of the coefficient is 9999999 and the exponent is -6.

In fact, the choice of scale is arbitrary: for a given scale, we can adjust the bias by the scale so the value of any particular bit pattern (encoding) is unchanged. We can therefore simplify this discussion by choosing a particular scale and then just consider the bias.

For many reasons, decimal numbers are usually described by a pair of integers, and therefore it proves convenient to consider the coefficient of decimal numbers as having a scale of 0 (so the coefficient is an integer). We'll use this value for the other questions in this FAQ. (Bear in mind that for describing the encoding we could equally well have chosen a scale of 6, similar to the traditional way of describing binary floating-point numbers, without affecting the remainder of this discussion other than the need to subtract 6 from the bias.)

How is the exponent bias chosen?

(This answer assumes you have already read the answer to "How are decimal numbers encoded?" where the terminology and examples are explained.)

The choice of bias for a given encoding is largely constrained by the rules of IEEE 854 and the revised IEEE 754-2008. These rules place two requirements on the set of values which must be representable:

1. A balanced range of exponents is defined by the parameters, E_{max} and E_{min}, which determine the overflow threshold ($10\times10^{E_{max}}$) and the underflow threshold ($1\times10^{E_{min}}$) respectively.

   An encoding must be able to represent all possible values with precision up to P_{limit} whose value is lower than the overflow threshold and greater than or equal to the underflow threshold. These values are called the *normal numbers*. In our example format, the range of normal numbers is $1\times10^{E_{min}}$ through $9.999999\times10^{E_{max}}$.
2. In addition to the normal numbers, it must also be possible to encode a further range of numbers of lower precision which are smaller than the underflow precision. These numbers are called *subnormal numbers*. The smallest subnormal number must be $1\times10^{E_{tiny}}$, where E_{tiny} is given by E_{min} - (P_{limit} - 1). In our example format, the range of subnormal numbers is $1\times10^{E_{tiny}}$ through $999999\times10^{E_{tiny}}$ (which is $0.000001\times10^{E_{min}}$ through $0.999999\times10^{E_{min}}$).

Given these requirements, it would seem that we can now determine E_{max} and E_{min}, and hence the bias, given P_{limit} and E_{limit}. However, IEEE 854 allows a choice to be made which affects this calculation: the encoding may be redundant if desired:

- In a *redundant* encoding, more than one coefficient (with an appropriate exponent) can be used to represent a given numerical value (for example, the underflow threshold could be represented as either $1000000\times10^{E_{tiny}}$ or $1\times10^{E_{min}}$).
- In a *non-redundant* encoding, only one coefficient is used for a given numerical value. (The coefficient chosen usually depends on the scale – when the scale is 0 the smallest coefficient is preferred).

For decimal arithmetic, intended as a tool for human use, the choice here is dictated by the need to mirror manual calculations and other human conventions (see "Why are the encodings unnormalized?"). A non-redundant encoding is inadequate for many applications, so a redundant (dual-integer) encoding is the norm.

This makes a difference at the top of the range of numbers (at the bottom of the range, the subnormal numbers cover the values for which redundant encodings can occur). For example, in the sample format, the largest normal number is $9.999999\times10^{E_{max}}$, which is represented by a coefficient of 9999999 and an exponent of E_{max}-6. However, the number $9.999990\times10^{E_{max}}$ is in the range of normal numbers, and this could be represented by either a coefficient of 9999990 and an exponent of E_{max}-6 or a coefficient of 999999 and an exponent of E_{max}-5.

When multiplying numbers using this form of representation, the result coefficient is simply the product of the operand coefficients and the result exponent is the sum of the operand exponents (2E+5 × 3E+7 gives 6E+12). Hence, either encoding for $9.999990\times10^{E_{max}}$ can arise by multiplying an appropriate pair of smaller normal numbers together.

Note that the second encoding shown for $9.999990\times10^{E_{max}}$ has a larger exponent than the exponent of the largest normal number (E_{max}-6), and in fact if all the redundant encodings which use up to P_{limit} digits are allowed, the largest exponent used in a representation will be E_{max}. (For example, the number $9.000000\times10^{E_{max}}$ encoded with a coefficient of 9 and an exponent of E_{max}.)

It might seem that we can avoid using these larger exponents by converting any such result to a value with a larger coefficient (for example, encoding the number $9.000000\times10^{E_{max}}$ with a coefficient of 9000000 and an exponent of E_{max}-6). This process is called *clamping*.

However, if we do this the result of a multiplication could differ depending on the encoding used, even though the value of the result is in the range of normal numbers and there is no overflow. If the same calculation were carried out in a format which had a greater exponent range, those results which had their exponent reduced (normalized) in the restricted format would not be normalized in the larger format: they would have a different coefficient and exponent.

Similarly, if the same calculation were carried out by hand, or by using existing computer decimal arithmetic (such as in Java, C#, or Rexx), we would not get the normalized result. This normalization would be an artifact which only appeared near a physical format (encoding) boundary.

This disadvantage, it was decided in committee, is outweighed by the the wider exponent range achieved (and the avoidance of invalid 'supernormal' numbers), and hence clamping is assumed.

Therefore, the largest exponent needed is E_{max}-6, and the smallest exponent needed is E_{min}-6. Providing a range of exponents bounded by these values allows us to meet all the requirements of IEEE 854 and decimal arithmetic.

From these two figures, we can easily calculate the E_{max} for a given E_{limit}. In our example format there must be 2 × E_{max} exponent values (the -E_{min}+6 negative values, the E_{max}-6 positive values, and 0). As E_{limit} is 191, there are 192 values available and so E_{max} must be +96. (In general, E_{max} = (E_{limit}+1) ÷ 2, and E_{min} = -(E_{max} - 1).)

The value of the bias follows directly from the value of E_{min}. The smallest exponent (E_{tiny}) must be -101, and this will be encoded as 0. The bias is therefore 101. (In general: bias = -Emin + P_{limit} - 1.)

How are the parts of decimal numbers encoded?

In the early days of electronic computers, many computers were decimal (some even used decimal numbers for addressing storage), and a great variety of both fixed-point and floating-point decimal encodings were used.

Over the years, most of these encodings were abandoned, but the form of decimal encoding that has endured (because of its practicality and usefulness) is the *dual-integer* encoding. Dual-integer encodings describe a decimal number using two integers: a *coefficient* and an *exponent* (often called a scale, which is a negative exponent). The value of such an number is

coefficient × $10^{exponent}$

(For example, if the coefficient were 123 and the exponent -2, the value of the number is 1.23.)

These two integers can be encoded in various ways. The exponent is almost always encoded as a small binary integer (up to 32 bits). The coefficient is generally one of three forms:

- **Binary Coded Decimal (BCD).** Here the coefficient is encoded as a series of decimal digits, each encoded in four bits with weights 8-4-2-1. The sign is either held separately or as a 4-bit code outside the range 0-9 which follows the digits of the coefficient.

  BCD is less efficient in space than a binary integer encoding, but it is much easier to convert a decimal number in this form to and from a character string representation. Rounding after any operation, and alignment before an addition or subtraction, are simplest in this form.
- **Binary.** Here the coefficient is encoded as a 'pure binary' number. This is more efficient in space than BCD, and faster for multiplications, but alignment, rounding, conversions, and any operations that deal with the digits in a number (such as calculating the check digit for a credit card or bar code) are slower. In the case of rounding, two multiplies and several other operations are usually needed, which is a punitive overhead.
- **Base 100 derivations.** Some database storage schemes store two decimal digits in a byte, using the values 0-99 in each byte, with various conventions for handling negative numbers. These schemes provide most of the advantages of BCD, but tend to be more complex.

The IEEE 754 decimal encodings for decimal numbers are also dual-integer in form, but use a compressed form of BCD (Densely Packed Decimal) which allows a higher precision decimal number in a given size. For example, a 64-bit encoded number can hold a 16-digit coefficient with a maximum normal exponent (E_{max}) of +384. In contrast, if BCD were used for the exponent the coefficient would be 13 digits, with a reduced maximum exponent of +64.

The Densely Packed Decimal encoding can be expanded to (or compressed from) BCD very rapidly (using table lookup in software or very simple logic with 2–3 gate delays in hardware). This means that the advantages of having a BCD encoding are preserved while allowing more precision and range in calculations.

How are zeros encoded?

Any number whose coefficient is zero has the value zero. There are therefore many redundant encodings of zero, and the exponents of the operands of a calculation are preserved when the answer is zero in the same way as they are when the result is non-zero.

For details, see "How are zeros with exponents handled?".

Why are the encodings unnormalized?

For decimal arithmetic, intended as a tool for human use, the choice of an unnormalized encoding is dictated by the need to mirror manual calculations and other human conventions. A normalized (non-redundant) encoding is ideal for purely mathematical calculations, but is inadequate for many other applications. Notably:

1. **Unnormalized encodings are used in existing languages, databases, and applications.**
   Decimal arithmetic in computing almost invariably uses a scaled integer representation. For example, the languages COBOL, PL/I, Java, C#, Rexx, Visual Basic and the databases DB2, Oracle, MS SQL Server, and Informix all use this form of encoding, as do decimal arithmetic libraries, including decNumber for C, bignum for Perl 6, Decimal in Python 2.4, EDA for Eiffel, ArciMath and IBM's BigDecimal classes for Java, ADAR for Ada, and the X/Open ISAM decimal type. In engineering, the same representation is often used – for example, in resistor color codes.

A normalized encoding would mean that the specification could not support these uses, and these existing software decimal calculations cannot be replaced by hardware which used a normalized encoding, because in up to 27% of cases the resulting coefficient and exponent will be different. This would require that all applications and their testcases be rewritten; an effort comparable to but significantly larger than the ‘Year 2000’ problem.

A normalized format could therefore only be used to store the integer coefficient of the numbers, with the decimal exponent being calculated and held separately (as in software today). Although this would give some performance improvement over a purely software implementation, all the calculation of exponents, calculating the length of the result, testing for rounding and overflow, *etc.*, will still have to be done serially and in software, hence largely obliterating the potential performance advantages – while requiring the programmer to provide the rules of arithmetic instead of building them into the hardware.

2. **All decimal forms can be derived by constraining the unnormalized encoding.**
   Three classes of decimal data are in common use:
   - Integers *(for example, numbers of items, call times in seconds)*: these are the numbers whose exponents are constrained to be 0. ‘Large integers’, such as 14 billion, are those whose exponents are greater than 0.
   - Fixed-point *(for example, $1.50, 1.200m)*: these are the numbers whose exponents are constrained to be a fixed value which is less than zero (-2 and -3 for the two examples).
   - Floating-point *(for example, exchange rates, tax rates, intermediate values in calculations)*: these are the numbers whose exponent is unconstrained (except for some maximum and minimum value). The coefficient may be constrained to a given precision (as for exchange rates or COBOL intermediate values), and it may be normalized or unnormalized, depending on the application.

   These types are all a subset of the standard type.

3. **An unnormalized encoding allows more efficient handling of integer quantities.**
   Decimal arithmetic often involves mixed integer and fractional arithmetic (for example: total 144 items at $17.99). Conversions to and from integers, whether decimal or binary, are simpler and faster with a unnormalized representation.

   When the coefficient is itself an integer, as in the specification, this becomes effectively a copy, with the exponent being set to 0. Integer arithmetic on these values is then a trivial subset of the floating-point arithmetic.

   In contrast, with a normalized representation the integers 10 and 11 *must* be stored with different exponents, and the coefficients will be shifted differently.

4. **Zero is not a special case.**
   With an unnormalized encoding, the value zero is simply a number with a coefficient of zero (for details, see “How are zeros encoded?”). Zeros are treated in just the same way as any other number and need no special processing.

   For example, the sums 1 + 1 and 1 + 0 can be handled identically, whether in software or in hardware.

   In contrast, the coefficient in a normalized representation must always be non-zero, and so zero must have a special coding. This, in turn, means that there has to be a separate pathway for zero operands of every instruction, and a separate pathway to handle zero results. Further, every conversion has to test for zero (in both directions) instead of treating all numbers in the same manner.

5. **Conversions from existing types require only integer conversions.**
   All existing decimal datatypes are encoded as a scaled integer (see “How are the parts of decimal numbers encoded?”), where the scale is a binary number and the coefficient is an integer (which might be binary, BCD, or some other format).

   All of these types need only the appropriate integer conversion to be converted to or from the standard format.

6. **Numbers can be recorded with their precision as written.**
   An unnormalized encoding allows store-and-forward handling of decimal numbers while preserving their implied exponent, as is required when the model for their use is an integral number of quanta (for example, an integer multiple of millimeters). This is independent of any arithmetic on the numbers.

   For example, a number retrieved from a database will have a coefficient (perhaps 250) and an exponent (perhaps -2), having the value 2.50. With an unnormalized layout, this number can be stored and then later retrieved without loss of information. With a normalized layout, it would have to be stored as 25 with an exponent of -1 (or a fraction-coefficient equivalent) and the original values of coefficient and exponent cannot be reconstructed.

   Similarly, when the characteristic exponent of a number is preserved by using an unnormalized layout, it is possible to separate the display of a number from its calculation. The display component of an application can safely display a number knowing that the intent of the logic that produced it is preserved; this means that the process of displaying can be guaranteed to not alter the value of a number and instead be only concerned with locale-dependent aspects of display.

   For example, if numbers are normalized then the display component may be forced to choose a display exponent for the number (perhaps rounding to two digits after the decimal point). This will hide information present in the number if its exponent were less than -2, and could well introduce rounding errors – or obscure serious errors of calculation.

7. **An unnormalized encoding preserves conventional precision indication in engineering and human-centric applications.**
   If trailing fractional zeros are removed, measurements and contracts may appear to be more vague (less precise) than intended, and information is lost. For example:

   - The length of a steel beam might be specified as 1.200 meters; if this value is altered to 1.2 meters then a contractor might be entitled to provide a beam that is within 5 centimeters of that length, rather than measured to the nearest millimeter.
   - Geographical survey and mapping records indicate the precision of measurements using fractional trailing zeros as necessary. Loop closure software makes use of this information for distributing errors. If fractional zeros are lost then precisely measured segments will appear imprecise; they will be over-adjusted and the final result will be corrupt.
   - The driving directions “Turn left after 14 miles” and “Turn left after 14.0 miles” lead to different driver behavior.

8. **Human-oriented applications require unnormalized encodings.**
   The results of human calculations and measurements are written in an unnormalized form (1.23 + 1.27 gives 2.50). To preserve these in their familiar form, they must be recorded as an unnormalized number.

   If hardware forced the software to record data in a normalized form, the end user has to adapt to the unusual and unexpected format of the computer. Applications which intrude in this way are unacceptable to many people.

9. **An unnormalized encoding allows either normalized or unnormalized arithmetic.**
   Unnormalized arithmetic is required or expected in many applications (see “Why is the arithmetic unnormalized?”).

   A normalized layout would not allow unnormalized arithmetic, whereas an unnormalized layout can support both normalized and unnormalized arithmetic.

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`

Decimal Arithmetic FAQ
Part 6 – Miscellaneous Questions 2024.04.24

Contents [back to FAQ contents]

How many decimal digits are needed to represent a binary floating-point number as an exact fraction?

When the exponent is zero, you need as many decimal digits as there are binary bits in the significand of the binary floating-point number (you get one power of two in each digit).

For example, the 64-bit IEEE 754 binary floating-point format (often called a '`double`') has a 53-bit significand (52 bits explicit in the format and one implied). To represent this exactly needs 53 decimal digits.

Here's a brief proof-by-example:

Consider the `double` which has all ones and an exponent of zero; that is, the binary fraction 1.1111111.... (53 one bits in all). That has the value $2^0 + 2^{-1} + 2^{-2} + ... + 2^{-52}$.

Now consider the first and last terms of that expansion:

$2^0 = 1$

$2^{-52} = 0.0000000000000002220446049250313080847263336181640625$

The total must be less than 2, so the first term (1) provides the leftmost digit of the total. Also, no term is smaller than 2^{-52}, so there will be no digits further to the right than those of that term. The sum of these two terms is:

```
1.0000000000000002220446049250313080847263336181640625
```

which has 53 digits.

Using a power of two

Note, however, that if we can assume a multiplier of 2^{-52} then we can express the same number exactly instead of as a decimal fraction by using the expression

$9007199254740991 \times 2^{-52}$

which needs only 16 digits. But, if the value2 $^{-52}$ is written out in decimal the expression becomes

```
9007199254740991 × 0.0000000000000002220446049250313080847263336181640625
```

There are 37 significant digits in the right-hand term, again giving a total of 53 significant digits.

See also see item 2 in an earlier answer for a couple more examples where you need over 50 digits for the *exact* conversion of a binary `double` to a decimal fraction.

Non-zero exponents

Additional digits may be needed when the exponent is non-zero; probably up to 63 or 64 (again, one digit for each bit of information in the original number). Leading zeros (depending on the chosen format for the result) may also be needed.

How many decimal digits are needed to represent a binary floating-point number reversibly?

As described in the last answer, representing a binary floating-point (BFP) number as an exact decimal number needs as many decimal digits as there are bits in the binary number.

However, far fewer are needed for a safe reversible conversion (that is, converting a BFP number to decimal and then back to BFP giving the identical BFP to that which one started with). For the 64-bit IEEE 754 binary floating-point format ('`double`'), it's 17.

Why 17? Think of the significand as a 53-bit integer. This can have (simplifying slightly, but erring on the safe side) 2^{53} different values. All we need to do is convert each of those values to a different decimal digit sequence (and ensure that converting back gives us the original binary integer). To do that we need a decimal number that can encode 2^{53} or more values.

2^{53} is 9007199254740992, which has 16 digits. To accomodate edge cases when taking into account the exponent it turns out one more digit than that is needed, which is 17 digits.

For more detail on this, see the *Conversions* section of links at the General Decimal Arithmetic page, especially Steele & White's paper and others which cite it.

Please send any comments or corrections to Mike Cowlishaw, `mfc@speleotrove.com`