Feature

Composite Arithmetic: Proposal for a New Standard

A general-purpose arithmetic standard could give general computation the kind of reliability and stability that the floating-point standard brought to scientific computing. The author describes composite arithmetic as a possible starting point.

W. Neville Holmes
University of Tasmania

Ever since their early days, digital computers and their arithmetics have made different kinds of complex computation possible. Frustrated, however, by the limitations of integer arithmetic (originally intended for counting loops and calculating addresses), scientists and engineers developed a floating-point number representation.

Floating-point arithmetic, or more properly, arithmetic using scaled numbers, was variously implemented, first in software and later in hardware. Scientific computing was greatly enhanced by the worldwide adoption in 1985 of the ANSI/IEEE standard for binary floating-point arithmetic.[1]

Significant growth in general (or popular) computing has led to a widespread use of spreadsheets, for which floating-point arithmetic is too specialized. Programs such as computer graphics, on the other hand, push floating-point arithmetic too far.[2] And programmers have long been bedeviled by the need to choose, often in ignorance, between fixed- and floating-point representations for values.[3]

Although floating-point arithmetic, commonly used in pocket calculators and electronic diaries, does not generally yield exact results, ordinary PC users naturally expect computer calculations to be exact, and some, like accountants, insist on at least the appearance of it. However, one very popular calculator gives the answer –0.000000001 to the calculation $((1 \div 3) \times 3) - 1$![4] Scientists and engineers, on the other hand, know that their measurements are approximate in the first place, so expect any results the computer gives them, based on those measurements, to be approximate.

These circumstances, coupled with developments in circuit technology and computation, have encouraged and made possible a more general arithmetic that saves the programmer from having to choose between number representations and gives the user more informative results. In this article I propose what I call *composite arithmetic*. Composite arithmetic combines aspects of traditional integer and floating-point arithmetics with less familiar aspects of rational and logarithmic arithmetics to complement the binary floating-point standard and satisfy more diverse computational needs. I describe a formatting scheme for storage and display of exact and inexact numbers and an extended arithmetic with a number format as its basis. I also introduce possibilities for implementing the arithmetic and discuss the interface between the representations and the arithmetic.

NUMBER REPRESENTATION

No matter how varied, computer arithmetics, and the digital forms of the numbers they use, follow a traditional pattern. Fixed-point arithmetic, more correctly called integer arithmetic outside the computing industry, is meant to represent and compute with integers exactly. In integer arithmetic, calculation with fractions is not done directly and must be carried out via subterfuges, such as scaling, which may deliver inexact results. Instead, fractions can be handled by an arithmetic called *floating slash*, which has been proposed[5] but not widely adopted. A similar scheme is built into the composite arithmetic I propose.

Fixed-point arithmetic can handle a relatively limited range of numbers. More than one length of representation is often provided so that the programmer can choose a length to cope with the expected range of numbers. But very soon the numbers become too large to store, a condition called arithmetic overflow, which a program must deal with specially to prevent wrong results.

Floating-point arithmetic was designed to circumvent this overflow problem but at the cost of exactness. Floating-point arithmetic can cope with a large range of numbers, but it does so only by approximating. More than one length of representation lets the programmer choose a length that produces suitable precision, yet overflow can still occur,

0018-9162/97/$10.00 © 1997 IEEE

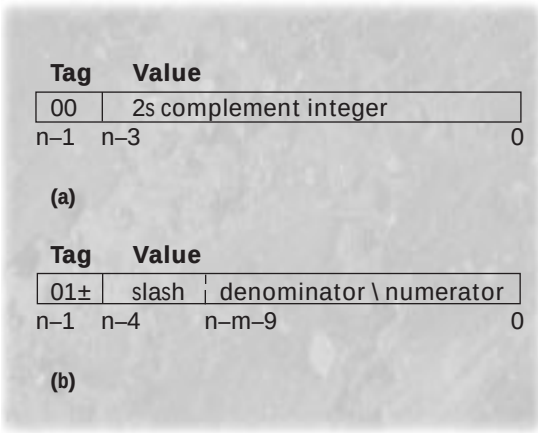

Figure 1. Proposed exact storage forms include (a) primary exact form (integer), and (b) secondary exact form (rational). In the bit numbering as shown, n *stands for the number of bits in the form and can be 32, 64, 128, or 256, while* m *numbers the different forms in increasing size as 0, 1, 2, or 3.*

Table 1. Integer storage form sizes.*

Number		Magnitude		
Bits	Bytes	Bits	Digits	Length
32	4	29	8.7	Short
64	8	61	18.4	Normal
128	16	125	37.6	Long
256	32	253	76.2	Extended

*Three bits are needed for the tag and the arithmetic sign.

as can underflow when numbers are too small to represent. An ideal general-purpose arithmetic would deal exactly with commonly encountered fractions like 1/3 and $19.99. It would neither overflow nor underflow.

To attain this ideal arithmetic, we must first prescribe a storage form for numbers that combines the advantages of fixed- and floating-point forms and also better represents fractions and extremely large and small numbers. The programmer should not have to choose between fixed- and floating-point forms—the arithmetic should dynamically determine which is needed. Moreover, users should not have to key in numbers in any specific format—the arithmetic should cope with whatever numbers it gets.

PROPOSED STORAGE FORMS

Composite arithmetic will provide a single binary form, for storing numbers in programs and files, that combines several different formats by a method traditionally known as tagging.[6]

Formatting numbers compositely in any binary storage has two main aspects. *How many bits to use* bears mainly on the precision with which a number can be stored. *How the bits are used* bears mainly on the kind and the value of the number being stored.

The number of bits for a storage form is usually specified in multiples of 16, as in the IEEE standard for binary floating-point arithmetic that provides 32- and 64-bit formats. Altogether, I recommend four lengths: short (32 bits), normal (64 bits), long (128 bits), and extended (256 bits). For composite arithmetic, 32- and 64-bit storage sizes would allow a choice in file and array design that would also provide adequate precision for most commercial calculations on exact numbers and for most scientific computations on inexact numbers. For financial and number theory calculations where very long exact results can occur, and for those occasional technical computations where very precise results can be required beyond the equivalent of 20 decimal digits, 128- and 256-bit storage sizes should be provided. Such extra precision is warranted by the proliferation of multiple-precision subroutine libraries.

EXACT FORMS

To free the programmer from having to choose between fixed- and floating-point representation, composite arithmetic merges both exact and inexact representations through a *tag bit* that signifies whether the number is stored exactly or not. If the tag bit is 0, the number is exact; otherwise it is inexact. Generally, only computation performed entirely with exact numbers will produce an exact result. The participation of only a single inexact number will normally produce an inexact result.

In composite arithmetic, a single tag bit isn't enough: In exact computations both integers and fractional numbers must be available. A second tag bit is therefore needed to signify whether a *primary* or a *secondary* form is being used. For exact values, primary is integer form, secondary is rational.

Integer storage forms

For integer storage forms, all but the tag bits can be used for storing the value, as shown in Figure 1a. One of the value bits is needed to store the arithmetic sign of the number, but the rest of the bits can store the magnitude, as shown in Table 1.

In the integer storage form, negative numbers should be stored as the 2s complement of their magnitude (one plus the bitwise complement of the magnitude) so that a negative zero cannot be stored. Zero is always exact.

Rational storage forms

Arithmetic with rational numbers is as exact as integer arithmetic. At least it can be, if appropriate representation and arithmetic are provided. In exact arithmetic, values may change from integer to rational and back again, depending on the computation.

A rational number typically springs from division of integers and to be exact must have both its denominator and its numerator stored. Rational numbers are stored in the secondary exact form, for which the tag bits are 0 and 1. This form can store both very large numbers, in which the numerator is much larger than the denominator, and very small numbers, in which the numerator is much smaller than the denominator.

Table 2. Approximate ranges of numbers in rational form.

Number		Value		Range		
Bits	Bytes	Bits	Digits	Small	Large	Length
32	4	24	7.2	6×10^{-8}	4×10^{6}	Short
64	8	55	16.6	3×10^{-17}	9×10^{15}	Normal
128	16	118	35.5	3×10^{-36}	8×10^{34}	Long
256	32	245	73.8	2×10^{-74}	1×10^{73}	Extended

For efficiency, a rational number's storage form must provide for sharing the value bits between numerator and denominator,[5] as shown in Figure 1b. So that the numerator can be normalized, this sharing must be resolvable to the bit. For a 32-bit number, five bits are needed to position the floating slash. For a 256-bit number, eight bits are needed. Because 0 is given an integer storage form, a normalizing high-order 1-bit can be implied for numerators. However, a bit is needed for the arithmetic sign and is shown in Figure 1b following the tag bits.

If the numerator is 1, then no bits are needed for the numerator. All value bits are available for the denominator. A denominator of 1 is not required in rational storage form, such a value being an integer, so that the smallest denominator is 2. The largest numerator is effectively one bit less than the largest denominator, because the two bits lost to the smallest denominator are partly compensated for by the numerator's normalizing bit.

Table 2 lists the approximate ranges encompassed by rational numbers. However, the distribution of representable numbers in a range is far from uniform. For example, rational numbers with relatively large denominators are restricted to having relatively small numerators.

The rational storage form in composite arithmetic is extremely versatile, though partly redundant because factors common to the numerator and denominator should be eliminated in any final arithmetic result. Redundancy can be exploited in several ways to increase the range of representable numbers. The bit address of the floating slash, which should point to the denominator's low-order bit, can range over the entire length of the rational storage form. This address, however, may point to bits that cannot be part of the denominator. Such redundancy can imply denominators that are powers of 10, which lets decimal fractions otherwise beyond the range of the rational storage form be represented exactly.

Infinity and indeterminacy

The denominator of a value in rational storage form can be 0. What is represented then is not a number, but infinity: the pseudonumber that is greater than any proper number.

Infinity is a notional value that needs storing, because it can result from division by zero. Zero and infinity are close counterparts. Multiplying them by any proper number can have no effect, apart from changing their arithmetic sign. But what should the result be when zero is multiplied by infinity? The rule that both are unchanged by multiplication can no longer hold for this case. The answer is usually said to be indeterminate and if a result can be indeterminate, then indeterminacy must have a representation. The denominator of a value in rational storage form can be 1. Ordinarily this shouldn't happen, because if the result of an exact calculation produces 1 as its denominator, then that result is an integer and should be stored as an integer. Such numbers are redundant to the scheme of representation and can be used to store an indeterminate value, as shown in Table 3.

Table 3. Representing exceptional values.

Value	Numerator	Denominator	Sign	Tag
Zero	0	None	No	00
Infinity	Any	0	Yes	01
Indeterminacy	Any	1	Yes	01

INEXACT STORAGE FORMS

A value must be stored inexactly when its magnitude or other aspect of its value prevents it from being stored exactly. In composite arithmetic, an inexact number is one that can be stored neither as an integer nor as a rational number. The exceptional values of Table 3 are here treated as exact.

Inexact numbers can be represented in two ways. Double-number forms separate the value into two arithmetically distinct parts—one multiplicative, the other exponential—as in traditional floating-point formats. Single-number forms have only one distinct part, solely exponential, as in the recently developed signed logarithmic form,[7] which represents the value by using its logarithm. (See the "Inexact Forms for Double and Single Numbers" sidebar for more information.)

Two levels of inexact storage forms are required. A *primary inexact* storage form stores the inexact values most commonly encountered during computation. A *secondary inexact* storage form lets values be represented that fall outside the capabilities of the primary form. Ideally, providing the secondary form will avoid the need for exception handling of overflow or underflow. Forms such as the traditional floating point are subject to overflow and underflow.

Double-number forms

Double-number forms are shown in Figures 2a and 2b and discussed in the "Inexact Forms for Double

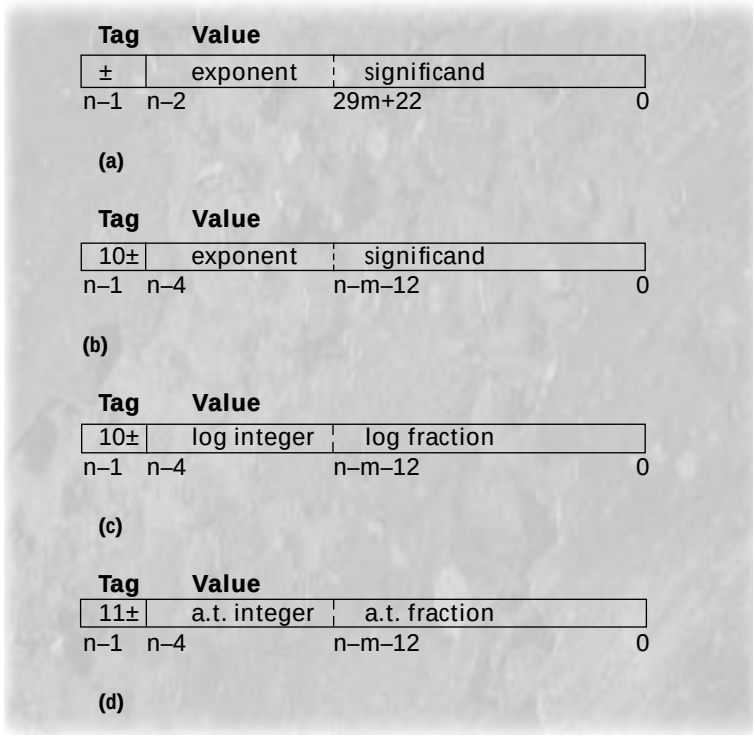

Figure 2. Inexact storage forms include (a) IEEE 754 (for comparison), (b) double-number form (semilogarithmic), (c) proposed single-number primary form (logarithmic), and (d) proposed single-number secondary form (antitetrational). In the bit numbering,* n *stands for the number of bits in the form and can be 32, 64, 128, or 256, while* m *numbers the different forms in increasing size as 0, 1, 2, or 3. Exception: (a), where* n *can be 32 or 64 and* m *can be 0 or 1.

Table 4. Primary inexact storage form sizes.

Number		Significand		Exponent		
Bits	Bytes	Bits	Digits	Bits	Largest	Length
32	4	21	6.3	8	3×10^{38}	Short
64	8	52	15.7	9	1×10^{77}	Normal
128	16	115	34.6	10	1×10^{154}	Long
256	32	242	72.8	11	2×10^{308}	Extended

and Single Numbers" sidebar.

Primary inexact. IEEE Std 754, shown in Figure 2a, can readily be adapted for use in composite arithmetic, although the two tag bits must be provided, and representations for special values like infinity and overflow are not required. Figure 2b shows an appropriate adaptation. The IEEE characteristics would be transferred as far as possible, and the floating-point storage forms would have the properties listed in Table 4.

The exponent field is lengthened one bit for each step up in form length just as the slash field of the secondary exact storage form is lengthened. Thus, the longer primary inexact forms not only represent all the values the shorter forms do, but the range of representable values is greatly increased.

Secondary inexact. A secondary inexact storage form is needed in composite arithmetic to deal with numbers too large or too small to fit within the primary inexact form.[8] One format possibility would resemble the primary but with, say, double the exponent size. This would greatly enlarge the range of representable numbers but not enough to completely avoid overflows and underflows.

Another possibility is to copy the floating-slash scheme, using a *floating bump field* to divide the value bits between significand and exponent ("bump" meaning a broadish point). Integers of 18 billion decimal digits could thus be inexactly stored in normal form. The floating-bump method may provide enough range for current needs; however, this extended range of storage forms may well lead to new kinds of computation that are then prone to underflows and overflows.

Ideally, overflow and underflow are to be avoided altogether. The *floating-root method* tries for this using a storage form similar to floating point but with value subfields of different significance. The sign bit still conveys the number's arithmetic sign, but the field corresponding to the exponent indicates how many times the square root of the magnitude must be taken to attain a value below two. The magnitude of a small number is increased instead by squaring until it exceeds one half. The field corresponding to the significand conveys the final magnitude after squaring or square rooting.

With the similar, symmetric level indexing scheme,[2] in which logarithms rather than roots are repeatedly taken, the representation system closes at a surprisingly low level for the four basic arithmetic functions,[9] which prevents overflow and underflow.

Single-number forms

The storage forms of composite arithmetic are intended for representing, as effectively as possible, numbers kept within programs and data files, and for holding temporarily intermediate values of complex calculations. These forms, not intended as the basis for arithmetic, are relatively compact and must use their length as efficiently as possible. The values they store are converted to and from the register form, explained later as the basis for the arithmetic.

Single-number inexact storage forms are more storage-efficient than double-number forms and less prone to cumulative loss of precision during conversions to and from register form. For these reasons I recommend single-number forms for use as the inexact forms of composite arithmetic. As a primary inexact form, Figure 2c shows a signed pure logarithmic system of representation[10] adapted to the composite arithmetic format for the primary form, which gives roughly the same characteristics as those shown in Table 4.

Symmetric level indexing as a secondary inexact storage form is a double-number form with precision much more discontinuous than that of floating-point form. Just as a pure logarithmic single-number form

can avoid discontinuity of precision in a primary inexact storage form, so can a pure antitetrational single-number form avoid it for secondary.

Tetration relates to exponentiation as exponentiation relates to multiplication.[11] By analogy, antitetration bears the same relation to tetration as the logarithm bears to exponentiation. The base 2 logarithm of 65536 is 16, but the base 2 antitetration of 65536 is 4. The base 2 logarithm of 2^{65536} is 65536, but its base 2 antitetration is only 5, which illustrates the compressive capability of antitetration that avoids overflow and underflow even under extreme conditions.

A pure antitetrational system is a practical secondary inexact storage form, with the same closure properties as symmetric level indexing, as the two systems are closely related.

Composite arithmetic would, ideally, apply pure logarithmic and antitetrational representations for the inexact storage forms, as shown in Figures 2c and 2d respectively, with negative logarithms given in 2s complement form, confining the more familiar semilogarithmic representations to the display forms.

PROPOSED DISPLAY FORMS

Storage forms, as described, are for binary digital computers, not for the users of those computers. The storage form seeks to formally preserve as much data as possible about the number being stored, yet it is pointless to preserve that data unless it informs the user.

With traditional programming languages, the programmer must decide both how to store numbers in the computer and how to display them to users. A general-purpose arithmetic standard should therefore specify both storage and display forms.

Programming languages

Digital computers based on composite arithmetic instead of floating point will use it for all numeric computation, though not for address calculations and loop control. Today's programming languages can simulate their usual number forms and their mixture of fixed- and floating-point arithmetic with composite arithmetic.

But problems arise with display forms, which traditional programming languages require the programmer to specify in intricate detail. Because numbers are presented to and by the user as character strings, their values must be converted from and to whatever storage form is used. With composite arithmetic, the present conversions can be simulated, though not without losing some of the extra data that the arithmetic keeps for the user.

For users and programmers to appreciate the full advantages of composite arithmetic, programming languages must eventually provide composite data types to make the extra data inherent in the storage form available to users or programmers. The language must also provide display forms that clearly and usefully convey the storage form information.

With compiled languages, the programmer need choose only the length of the composite data type for either storage or display. For interpreted languages and related utilities such as spreadsheet processors (for which composite arithmetic is most apt), users will be primarily concerned with display field length.

The numbers used in composite arithmetic include rational numbers, as well as the infinities and indeterminacies given in this format, plus formally inexact values with huge ranges. Apart from the need to present different kinds of numbers in the same display or input field, there is also the need to present rationality, neither of which is met by present programming languages.

Displaying rationality

Display form problems typically result from ASCII character set deficiencies. This character set generally has 12 usable characters for displaying exact numbers decimally—10 decimal digits, a dot, and a hyphen. The solidus(/) is not available because it serves as a

Inexact Forms for Double and Single Numbers

Double-number inexact forms, such as the traditional floating-point number representations, give their values in two distinct components—the *exponent* and the *significand*. These forms also use an implied base, popularly 2, 10, or 16. The value represented is recovered from its representation by multiplying its significand by its base as many times as specified by its exponent, which is always an integer. The exponent is the integer part of the logarithm of the value being represented, where the logarithm is taken to the implied base.

Single-number inexact forms, such as the signed logarithmic representation, give their values in a single distinct numerical component: a single number, though it has an integer part and a fractional part. The representation is the logarithm of the value being represented, and the value is recovered from its representation by raising its implied base to the power specified by the representation.

Both kinds of inexact forms keep two arithmetic signs. The arithmetic sign of the value being represented is usually kept explicitly and is the sign bit shown in Figure 2 of the main text. The other arithmetic sign is the sign of the exponent or the logarithm, which signifies whether the magnitude of the value being stored is greater than one or not, and which is usually implied by the complement representation of the exponent in floating-point forms.

As a storage form, the main drawback of the double-number forms is discontinuous precision. Representable values in a floating-point system are uniformly spaced for any particular exponent value. But across an exponent boundary, the spacing changes by a factor that is the implied base or its reciprocal. For an implied base of two, the imprecision doubles or halves when crossing an exponent boundary. The precision of single-number forms is continuous, which makes these forms more reliable.

A general-purpose composite arithmetic requires comprehensive storage and display forms for exact and inexact numbers.

division symbol in the absence of ÷ from the ASCII character set. The dot serves as a decimal point, which is a shame because it is the second most significant value signifier in a number yet second only to the blank in visual insignificance.

Decimal nonintegers are shown with a decimal point between the fractional part, which forms the numerator of the implied decimal denominator, and the integral part. The denominator is implied by the number of digits in the fractional part. Only an explicit denominator need be added to let ordinary fractions be displayed. One unambiguous way to do this in composite arithmetic is to place it to the right of the display form, separated from the numerator by a second dot called, say, the fraction point.

With the dot as a fraction point, a number such as $456\,^{7}/_{8}$ can be shown or entered as 456·7·8 conveniently, if unfamiliarly at first. Expressing $^{2}/_{3}$ as 0·2·3 is not quite so convenient but more convenient and accurate than as 0.66667.

The hyphen serves as a prefix to designate negativity. If it were used as a fraction point, then $456\,^{7}/_{8}$ would more conveniently be shown as 456·7–8 and $^{2}/_{3}$ as 2–3. But this is ambiguous because the computing industry, encouraged by the world's arithmeticians, who were the original sinners here, uses the hyphen to symbolize the subtraction function as well as the property of negativity.

Using the dot both for the decimal point and the fraction point in rational numbers therefore seems the only practical approach. The double-dot convention is suited to fixed-field formatting for rational numbers in commercial applications, because both the numerator and denominator subfields can be padded to a fixed length with leading zeroes. It is also suited to presenting exceptional values, using 0·1·0 for infinity and 0·0·0 for indeterminacy. The dot could be keyed in as an ordinary period, but the display could automatically enlarge and raise it when its context requires this.

Displaying inexactness

By and large, integers and common fractions are intended to be exact. Numbers in scientific or engineering notation with explicit scaling, as in 2.995×10^{10} or 2.995e10, are intended as measurements and thus are inexact. A measurement like 7.89 is typically intended to be exact up to its rightmost digit, which is soft in the sense that 7.89 usually implies a value in the range 7.885 to 7.895. (Decimal fractions might be exact for accountants but are frequently only approximate measurements for others.)

Since composite arithmetic lets inexactness be shown in the storage form, this inexactness must be expressed to and by the user. This means adopting new display conventions, which should follow existing conventions insofar as possible and should be the same whether for storage or display.

For expressing inexact numbers, the present display form uses the letter *e* to separate the significand from the exponent. The exponent uses a scaling base of 10, but the international standard for SI (Système Internationale) Metrics specifies a scaling base of 1,000. This base lets us adopt a scheme in composite arithmetic to show standard measurements in a way that resembles current practice.

A *k*-notation (kilo-notation) would be like the *e*-notation (engineering-notation) described above, but would use 1,000 rather than 10 as its scaling base. Thus 100k1 (popularly 100k) would stand for 1e5, and 100k2 would stand for 1e8. The *k*-notation could be supported by an *m*-notation (milli-notation) to avoid squeezing a negative sign into the exponent. For example, while 6.023×10^{23} could be written as 602·3k7, 0.67×10^{-11} could be written as 6·7m4. Secondary inexactness could be displayed by a second *k* or *m*.

Display form usage

Values presented in display form by a program would be converted from register form and appear as exact or inexact, primary or secondary, depending on the properties of the value being displayed and the length of the field provided for the display. Exact and inexact displayed values would be distinguished, so that 0·1·3 and 0·3333 would show different exact values, while 333·3m would show an inexact value. An inexact value would never be shown as exact, but an exact value would be shown as inexact if the display field were too short to let it be shown exactly.

Users would key in values in display form according to the same conventions. All exact values keyed in would be stored exactly in register form, and all inexact values would be stored as inexact in register form.

INTERFACE ISSUES

A general-purpose composite arithmetic requires comprehensive storage and display forms for exact and inexact numbers. The arithmetic itself will be carried out by instruction on arithmetic registers, the design of which will largely determine the detailed nature of that arithmetic.

Register design must consider the different instructions required to interface the various storage and display forms with the registers. Many varied instructions will be needed to move numbers to and from the registers and to test register values. This strongly implies

Draft Standard

Three documents suggesting wordings for a draft standard are available at ftp://ftp.comp. utas.edu.au/pub/nholmes/ca/{dssf.ps,dsdf.ps,dsrf.ps}. I will reflect comments in these documents.

only one kind of register, to minimize the number of different instructions. Nonetheless, the variety of storage and display forms, and the need to specify different kinds of rounding, require many different instructions.

One obvious but unsatisfactory approach would provide a few registers in the 256-bit extended storage form. The load and store instructions would then convert, if necessary, to and from that form, and the arithmetic would be carried out on that form. Should the arithmetic require that the nature of a value be changed between exact and inexact or between primary and secondary forms, the execution of the instruction that caused the change would include the conversion.

This approach presents several problems, such as fitting many new instructions into existing machine-instruction repertoires. Because executing these instructions would be complex, at least partly because of their need to convert number forms occasionally, composite arithmetic instructions would not, without heavy use of pipelining, fit very well into RISC architectures, which need to execute instructions in a single machine cycle.

A better approach is to logically decouple the composite arithmetic instruction repertoire from that of the host computer. What I envision here is a new kind of subprogram engine, a numerical computer with its own local registers into which storage and display forms can be loaded without loss of information. All elementary arithmetic operations defined for the engine would be executed between and within local registers using register form.

The subprogram engine could be implemented as library subroutines, as a physically distinct coprocessor, or as an intrinsic processor manipulated by one or two general main processor instructions. The operands of these instructions would transmit either arithmetic instructions to the subprogram engine, or main storage addresses of storage or display form numbers to be loaded or stored.

COMPOSITE ARITHMETIC

An as-yet-undefined standard would specify three aspects of composite arithmetic:

- Number forms.
- Subprogram engine capabilities, regarding instruction format and repertoire (basic arithmetic operations first, others added in an evolutionary manner).
- Subprogram engine and the host computer interface. Specified machine instructions for the interfacing should not be detailed, as the appropriate digital computer's instruction architecture will do the actual interfacing.

The composite arithmetic itself will depend on register design. For some computations, precision greater than 256 bits is needed to store intermediate results. A very large register, elsewhere called a long accumulator (see the "Long Accumulator Design" sidebar), would provide this precision and could even simplify calculation by representing primary inexact storage form values in a fixed-point form.

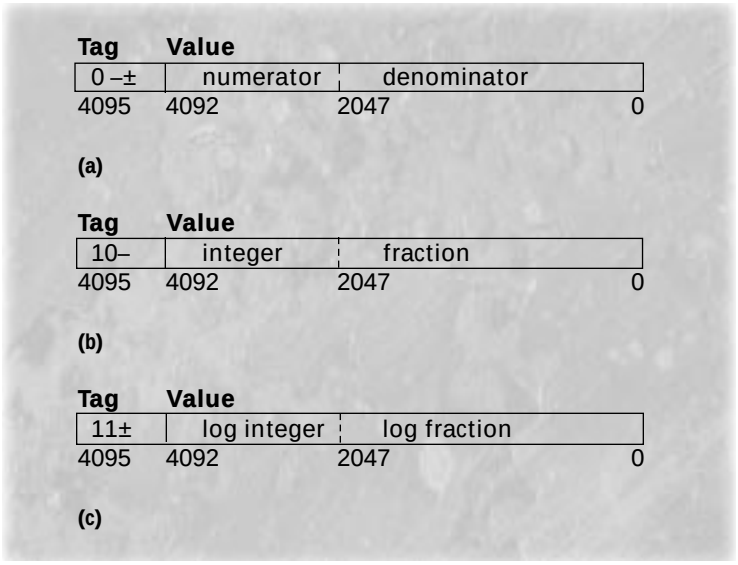

Figure 3. Proposed register forms include (a) a single exact form (rational), (b) a primary inexact form (fractional), and (c) a secondary inexact form (logarithmic). The leftmost two bits show the tags; the third bit, an arithmetic sign.

Register form

The composite arithmetic register would be basically a fixed-point register large enough to store in fixed-point form any number representable in extended primary inexact storage form. Because that storage form has an effective 10-bit log integer and a 242-bit log fraction, a register form fraction field of at least 1,266 bits is needed. If the register is to be symmetrical between integer and fraction, as shown in Figure 3a, this means a register size of at least 2,532 bits, or 317 8-bit bytes. To provide spare capacity and a round binary size, 4,096 bits or 512 bytes makes a good choice for the register.

The 4,096 bits of main register data must accommodate certain auxiliary data. Two tag bits would signal what kind of value (exact or inexact, primary or secondary) is being stored. The inexact forms of Figures 3b and 3c keep their values in 2s complement with implied arithmetic sign, but the secondary inexact form needs an overall arithmetic sign as shown in Figure 3c alongside the 11 tag bits.

For inexact numbers loaded from secondary inexact storage form, or which arise from overflow or underflow of a primary inexact result, the register stores the logarithm of the value rather than the value proper. Arithmetic with logarithms is well understood and presents no fundamental problems.

For exact numbers, as shown in Figure 3a, the integer part of the inexact register form could be used

Long Accumulator Design

Adding and subtracting floating-point values of somewhat different magnitudes can give strange results when carried out directly. For example, $(1 + 1 \times 10^{20}) - 1 \times 10^{20}$ may result in zero because the 1 vanishes off the end of the floating-point representation of 1×10^{20} when it is added before the canceling 1×10^{20} is subtracted, while $1+(1 \times 10^{20}–1 \times 10^{20})$ will yield one because the two 1×10^{20} values cancel before the one is added.

To avoid these and similar unwanted effects, Ulrich Kulisch and Willard Miranker included a *long accumulator* in their scheme for accurate arithmetic.[1] The long accumulator is a fixed-point register long enough to allow any floating-point value to be loaded into it in a fixed-point representation. There is therefore no loss of digits when values are added or subtracted into the long accumulator because there is always a place for them to go. This has been implemented in microprogram on the IBM 4361 and is cleanly adaptable to serve the composite arithmetic's more general purposes.[2]

References

1. U.W. Kulisch and W.L. Miranker, "The Arithmetic of the Digital Computer: A New Approach," *SIAM Review*, Mar. 1986, pp. 1-40.
2. E. Lange, "Implementation and Test of the ACRITH Facility in a System/370," *IEEE Trans. Computers*, Sept. 1987, pp. 1,088-1,096.

for the numerator, and the fraction part for the denominator. No distinction need be made between primary and secondary exact forms when they are held in the register. Overflow or underflow of an exact result would cause a transition to primary inexact form.

Such a large register may at first seem fated to significantly slow down the arithmetic. However, in hardware at least, data paths could be sufficiently wide to convey a complete register value entirely in parallel. Other techniques are also possible to speed up the arithmetic, such as pipelining and balanced ternary representation.

Intermediate results

When a composite arithmetic register is used as described, its real nature would be invisible to users, that nature being evidenced only in the arithmetic results: for example, yielding exact results when less straightforward arithmetic would yield inexact results. Some of these improved results would be available only when sequences of basic arithmetic operations could be carried out entirely within the register set or, for a software implementation, in a library function.

To consistently achieve improved results, intermediate results should be accessible in register form such that register contents are directly loadable or storable. To accommodate these results, a composite arithmetic should comprise

- a *register form*, in which values can be stored and used repeatedly and exactly, and if exactness is not possible, with recorded accuracy;
- a *storage form*, in which values can be stored efficiently for use within programs and files, and for transmission;
- a *display form*, in which values can be displayed distinctively and effectively, as appropriate to the length of field provided; and
- an *arithmetic processor* that works on register form values to produce results exactly if possible, otherwise as accurately as possible.

The general-purpose composite arithmetic I've outlined is the basis I propose for a new standard complementing the IEEE binary floating-point standard. Successful development of a composite arithmetic standard would be most timely, given the burgeoning ability to manufacture complex processors and the interest in extended forms of arithmetic being shown in the research literature. It would also be highly beneficial in support of better electronic calculator arithmetic and standard operation of generic software packages such as those including spreadsheet capabilities. Three areas would be the target of such a standard.

- *Number form*. One storage form should address the needs of all commonly used numbers, commercial and scientific, and one display form should address the needs of as many users as possible, private and professional. The aim and major benefit are that the specific storage and display forms depend on the value to be stored or displayed, and that the form is automatically and consistently determined.
- *Arithmetic*. Arithmetic should be defined that gives the best possible result allowed by the values used as starting points to a calculation. Arithmetic with a long accumulator will yield better-than-usual results, and it implies also that a register form should be specified by a standard using such arithmetic.
- *Interface between the forms and the arithmetic*. Arithmetic must be exploited practically by programs and operating systems.

The manufacturing technology exists that could provide cheap processors for a general-purpose arithmetic. Applications that would be enhanced by a thorough-going general-purpose arithmetic are already very widely used. Fringe computations proliferate that need the extra accuracy composite arithmetic would provide.

Given the need for a new standard of this kind, work toward its adoption must start from acceptance of the general approach I've proposed. Details to define each of the three areas can be negotiated later. With care, a basic capability could be defined that could be adopted relatively promptly, allowing functions beyond the most basic and capabilities like accuracy control to be added later. ❖

Acknowledgments

This work was supported by my family, colleagues, and the anonymous reviewers. The attitudes on which this work is based were engendered by Jacques Grenot and Ken Iverson, who have always insisted that computation can be graceful, and by Willard Miranker, who once patiently explained to me how computation can be made trustworthy. This article was provoked by discussions at a 1994 workshop in Toowoomba, led by Iverson and run by Mike McFarlane and Walter Spunde of the University of Southern Queensland.

References

1. *ANSI/IEEE Std. 754-1985, Binary Floating-Point Arithmetic*, IEEE Press, Piscataway, N.J., 1985 (also called ISO/IEC 559).
2. D.W. Lozier, "An Underflow-Induced Graphics Failure Solved by SLI Arithmetic," in *Proc. 11th Symp. Computer Arithmetic*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 10-17.
3. L. Hatton and A. Roberts, "How Accurate Is Scientific Software?" *IEEE Trans. Software Eng.*, Oct. 1994, pp. 785-797.
4. H.W. Thimbleby, "A New Calculator and Why It Is Necessary," *Computer J.*, Issue 6, 1995, pp. 418-433.
5. D.W. Matula and P. Kornerup, "Foundations of Finite Precision Rational Arithmetic," *Computing*, Suppl. 2, 1980, pp. 85-111.
6. J.K. Iliffe, *Basic Machine Principles*, Macdonald, London, 1968.
7. D.M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Computers*, Aug. 1994, pp. 974-982.
8. H. Yokoo, "Overflow/Underflow-Free Floating-Point Number Representations with Self-Delimiting Variable-Length Exponent Field," *IEEE Trans. Computers*, Aug. 1992, pp. 1,033-1,039.
9. D.W. Lozier and F.W.J. Olver, "Closure and Precision in Level-Index Arithmetic," *SIAM J. Numerical Analysis*, Oct. 1990, pp. 1,295-1,304.
10. M.G. Arnold et al., "Applying Features of IEEE 754 to Sign/Logarithm Arithmetic," *IEEE Trans. Computers*, Aug. 1992, pp. 1,040-1,050.
11. R.L. Goodstein, *Fundamental Concepts of Mathematics*, Pergamon Press, Oxford, UK, second edition, 1979.

Neville Holmes is a senior lecturer in the Department of Computing at the University of Tasmania, Launceston, Australia. He held various technical positions during 30 years in the computing industry. Holmes received a BEE from the University of Melbourne and a master's in cognitive science from the University of New South Wales. Contact him at Neville.Holmes@utas.edu.au.