Semantics for Exact Floating Point Operations

G. Bohlender, W. Walter
Institut für Angewandte Mathematik
Universität Karlsruhe
Kaiserstr. 12, D-7500 Karlsruhe
W. Germany

P. Kornerup
Dept. of Math and Computer Sci.
Odense University
DK - 5230 Odense
Denmark

D.W. Matula
Dept. of Computer Science and Eng.
Southern Methodist University
Dallas, TX 75275
USA

Abstract

Semantics are given for the four elementary arithmetic operations and the square root, to characterize what we term exact floating point operations. The operands of the arithmetic operations and the argument of the square root are all floating point numbers in one format. In every case, the result is a pair of floating point numbers in the same format with no accuracy lost in the computation. These semantics allow us to realize the following principle: it shall be a user option to discard any information in the result of a floating point arithmetic operation. The reliability and portability previously associated only with mathematical software implementations in integer arithmetic can thus be attained exploiting the generally higher efficiency of floating point hardware. Currently, these operations are partially supported in hardware on many existing architectures. This proposal is intended to influence the design of future floating point processors to include the full functionality with hardware efficiency.

1 Introduction and Summary

Hardware implementations of floating point arithmetic operations have been prone to discard supplemental information necessarily generated in the process of computing and returning a rounded result. Most notably the low order bits of floating point addition, subtraction and multiplication are often available at intermediate stages of the hardware algorithm, as also is the remainder for divide and square root, yet these results are generally unavailable as a secondary supplemental result. It is our contention that for many purposes the user would be well served to have this supplemental information provided in a convenient form, particularly in view of its availability at very modest additional hardware cost.

In this paper we first state a goal for hardware floating point design in the appealing sense of "exact floating point operations," designed to dispel the popular misconception that floating point operations are inherently inexact. We support our goal with prescribed semantics characterizing the notion of exact floating point operations, where the prescribed results can be readily obtained and provided to the user by traditional hardware procedures. Of the utmost significance in this approach is then that any compromise in floating point computation by truncation of either the accuracy of a number representation, or the limiting value of a process, is shifted to *user control*. With floating point accuracy compromises then an artifact of a mathematical software system, any ill-contrived and/or ill-defined approach may be appropriately corrected at that level.

Our motivation is expoused in the following principle.

Exact Floating Point Operation Principle:
It shall be a user option to discard any information in the result of a floating point arithmetic operation.

This dictate is effectively directed to hardware arithmetic unit designers with the term "user" denoting either a person or program having access to and controlling the operations. By "floating point arithmetic operation" we mean the five operations: addition, subtraction, multiplication, division, and square root; explicitly described in the IEEE/ANSI floating point arithmetic standards[1,2]. We shall herein describe semantics for exact floating point arithmetic operations satisfying this principle.

Our proposed semantics may briefly be summarized as follows. The "exact" finite precision result of each of these floating point operations shall be expressed by two component numbers, the first of which is a rounded value of the infinitely precise result rounded by an appropriate one of the IEEE defined controlled roundings: to-nearest, to-positive-infinity, to-negative-infinity; and designated the *high order part*. Let x and y be floating point numbers of

CH3015-5/91/0000/0022$01.00 © 1991 IEEE

the same format. Letting h denote the high order part for addition, subtraction or multiplication of x, y, we define the second component number to be the *low order part* ℓ given, respectively, by $\ell = x + y - h$, $\ell = x - y - h$, and $\ell = x \times y - h$. Letting q denote the high order part for division of x by y or the square root of x, we define the second component number to be the *remainder* r given by $r = x - q \times y$ for division and $r = x - q \times q$ for square root. We note as a principal result that if the high order part is chosen by round to-nearest, the second component number can itself always be represented exactly by another floating point number of the same format (in the absence of overflow/underflow or invalid operation exceptions). It is possible for some, but not all, of the five operations that the high order part may be chosen by an alternative rounding with the second component number still representable by a floating point number of the same format.

Our semantics is simply to prescribe as allowed roundings for the high order part those roundings which guarantee that the second component number will be exactly representable as a second floating point number of the same format. The exact floating point result is then the floating point number pair in each instance, in the absence of any exception condition.

The fact that the low order part and/or remainder so defined can be represented by a floating point number of the same format in each operation instance is straightforward to verify and is not claimed to be a significant result in and of itself. The importance we ascribe here is to the consistent package of results, namely, that barring exceptions, the result of addition, subtraction, multiplication, division, and square root can each be expressed by a pair of floating point numbers in the same floating point format. Furthermore, these value pairs satisfy well known mathematical identities, allowing the user total flexibility to build more complex expressions of arbitrarily high precision by well known techniques [3,4,5,6].

In section 2 we give a brief formal account of the semantics and prove their attainability. In section 3 we consider alternative roundings avilable to prescribe the high order part in conformance with the low order part/remainder guaranteed to be representable in a second floating point word. In section 4 we note exception conditions that can arise, and refer to a recommended known approach to the handling of underflow/overflow.

2 Semantics for Exact Floating Point Operations

We consider a floating point system $F = F(b, p, emin, emax)$ characterized by its base b, the length p of a mantissa, and the minimal and maximal exponents, $emin$ and $emax$. A floating point number x in this system consists of a sign, a fixed-length mantissa with digits $d_1, d_2, \ldots, d_p$, where $0 \leq d_i \leq b - 1$ for all i, and an exponent e with $emin \leq e \leq emax$, both given to the base b:

$$x = \pm\, b^e \sum_{i=1}^{p} d_i\, b^{-i}.$$

For simplicity, we will restrict our discussion to normalized floating point numbers, that is, numbers with $d_1 \neq 0$, and we do not define the representation of 0.

For floating point numbers, we define $e(x)$ to mean the exponent of x in the preceding sense. Also, we define the notation

$$ulp(x) := b^{e(x)-p}$$

to mean one unit in the last place relative to the floating point number x. Note that for all floating point numbers x, $x + ulp(x)$ and $x - ulp(x)$ are the adjacent floating point numbers above and below x, unless one of them overflows or underflows, or unless $x = \pm b^n$ for some integer n. If $x = b^n$ for some integer n, then there are $b - 1$ floating point numbers between $x - ulp(x)$ and x. By symmetry to zero, an analogous statement holds for $x = -b^n$.

For the remainder of this paper, we will assume the above definitions. In particular, we will assume a floating point system F with an arbitrary base $b \geq 2$ and with a number of mantissa digits $p \geq 2$. All floating point numbers are assumed to be members of this floating point system F.

The mathematical specification of the proposed exact floating point operations is summarized in the following table. Note that all arguments are of the same floating point format.

operation	arguments		mathematical specification
	in	out	
exact_add	(x, y,	h, l)	$x + y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
exact_sub	(x, y,	h, l)	$x - y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
exact_mul	(x, y,	h, l)	$x * y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
exact_div	(x, y,	q, r)	$x = qy + r$ with $\|r\| < \|y\| ulp(q)$ unless $q = 0$
exact_sqrt	(x,	q, r)	$x = q^2 + r$ with $-2q \cdot ulp(q) + ulp^2(q) < r$ $r \leq 2q \cdot ulp(q)$

We will demonstrate that the above specifications are reasonable: they can be fulfilled, and they require the first part of the result to be of one ulp accuracy.

Lemma: For every pair of floating point operands x, y in the four arithmetic operations and for every floating point argument x in the square root operation, it is possible to find a pair of floating point numbers h, l or q, r, respectively, such that the above specifications are fulfilled — unless an exception occurs.

Proof:

Addition, Subtraction:

Without loss of generality, let us assume that $e(x) \geq e(y)$. Two cases have to be distinguished. If $e(x) - e(y) < p$, their sum (difference) can be represented with $2p$ digits even if a carry occurs. Taking the first p digits for h and the last p digits for l makes $e(h) - e(l) \geq p$. Note that l may have to be normalized, making $e(h) - e(l) > p$, or l may be 0. On the other hand, if $e(x) - e(y) \geq p$, then $h = x$ and $l = y$ (or $l = -y$ for subtraction) obviously satisfies the specification.

Multiplication:

The exact product of two floating point numbers can be represented with $2p$ digits. From the leading non zero digit choose the leading p digits as the high order part h and the balance of at most p digits as the low order part l.

Division:

For division of x by y, we term a partial quotient q to be the result of rounding the exact result x/y towards positive or negative infinity to a certain length. The corresponding remainder is defined by $r = x - qy$. The traditional long-hand division of x by y can be used to determine at each step a partial quotient q and the corresponding remainder r.

In the trivial case, the exact quotient x/y is exactly representable as a floating point number, so choose q to be the exact quotient and $r = 0$. For the general case, let us assume that $q > 0$ without loss of generality. Choose q to be the round-to-zero (truncated) result, that is $q < x/y < q + ulp(q)$. Then $0 < x/y - q < ulp(q)$ and thus $0 < |r| < |y| \cdot ulp(q)$.

Any floating point number $z \neq 0$ is an integral multiple of $ulp(z)$, that is $z = n \cdot ulp(z)$ for some integer n with $|n| < b^p$. Thus qy is an integral multiple of $ulp(q) \cdot ulp(y)$. Since $|x| > q \cdot |y|$ by assumption, x is also an integral multiple of $ulp(q) \cdot ulp(y)$, and so is $r = x - qy$. Since $|r| < |y| \cdot ulp(q) = i \cdot ulp(y) \cdot ulp(q)$ where $|y| = i \cdot ulp(y)$, r can be represented by a floating-point number.

Square Root:

For the square root of $x \geq 0$, we term a partial root q of x to be the result of rounding the exact result $\sqrt{x}$ towards positive or negative infinity to a certain length. The corresponding remainder is defined by $r = x - q^2$. The traditional long-hand square root process can be employed to visualize the selection of the partial root q and the corresponding remainder r.

Let q be the round-to-nearest value of $\sqrt{x}$, and assume that $r = x - q^2$ does not underflow. It is readily noted that $e(q)$ cannot be of opposite sign as $e(x)$ and can be of no greater magnitude unless $e(x) = 0$, $e(q) = 1$. q can never overflow or underflow and r can never overflow, but can underflow.

By the same argument as for division, x as well as q^2 must be integral multiples of $ulp^2(q)$. Hence $r = x - q^2$ is also an integral multiple of $ulp^2(q)$, say $r = i * ulp^2(q)$. We can also write $q = j * ulp(q)$ for some integer j. Our proof will be completed by showing $|i| \leq |j|$.

Since q is the round-to-nearest value of $\sqrt{x}$,

$$q - \frac{\text{ulp(q)}}{2} \leq \sqrt{x} \leq q + \frac{\text{ulp(q)}}{2}.$$

So now

$$\begin{aligned} r &= x - q^2 \\ &= (\sqrt{x} + q) * (\sqrt{x} - q), \end{aligned}$$

and then

$$\begin{aligned} |r| &\leq (2q + \frac{\text{ulp(q)}}{2}) * \frac{\text{ulp(q)}}{2} \\ &= q * \text{ulp(q)} + \frac{\text{ulp}^2(q)}{4}. \end{aligned}$$

Since $r = i * \text{ulp}^2(q)$,

$$|i| * \text{ulp}^2(q) \leq |j| * \text{ulp}^2(q) + \frac{\text{ulp}^2(q)}{4},$$

so then $|i| \leq |j|$ since i and j are integers, and thus r must be representable in the same format as q.

Given that a result pair q, r for the exact square root of x exists, we wish to specify that r must correspond to less than $ulp(q)$. Note that if the remainder r is positive, x must fall between q^2 and $(q+ulp(q))^2 = q^2+2q \cdot ulp(q)+ulp^2(q)$. Therefore, $r < 2q \cdot ulp(q) + ulp^2(q)$, and since r is an integral multiple of $ulp^2(q)$, we obtain $r \leq 2q \cdot ulp(q)$. Similarly, a negative remainder must satisfy $r > -2q \cdot ulp(q) + ulp^2(q)$.

3 High Order Part Rounding

For all five exact floating point operations $+, -, *, /$, and $\sqrt{\ }$, the preceding specifications require that the exact answer lie in the interval $[h - ulp(h), h + ulp(h)]$ or $[q - ulp(q), q + ulp(q)]$. A consistant way of implementing these operations is to always choose the round to-nearest result as the first floating point number of the returned result, that is, the second component (l or r) can then be shown to always be exactly representable as a floating point number.

Note that if $|e(x) - e(y)| > p + 1$ for addition, the only way to exactly represent the sum is to return the original operands x and y as result, and this is obtained only by round to-nearest. The situation is analogous for subtraction. For multiplication, on the other hand, any one ulp rounding will work since the product can always be represented with $2p$ contiguous digits.

In $+, -, *$, note that the condition $e(h) - e(l) \geq p$ is equivalent to the requirement $|l| < ulp(h)$. This further guarantees that for $l \neq 0$, h will be one of the two floating-point numbers bracketing the exact result of the add, subtract or multiply operation unless h is a power of the base. In a two digit decimal floating-point system, we may obtain for $x = 0.98$ and $y = 0.005$ the result $x + y = 0.985 = h + l$ with $h = 1.0$ and $l = -0.015$. Rather than disallow such results, we have chosen the criterion corresponding to $|l| < ulp(h)$ as sufficient in view of its simplicity and reasonable extension to the divide and square root operations.

In $+, -, *$, if the round to-nearest result is chosen for h, the stricter requirement $|l| \leq \frac{1}{2}ulp(h)$ is always satisfied. Similarly, in division, if the round to-nearest result is chosen for q, the stricter requirement $|r| \leq \frac{1}{2}|y|ulp(q)$ is satisfied. In this and the following examples, we will use a two digit decimal floating point system. Consider the following problem. The division 99/120 may produce $q = 0.82$ and $r = 0.60$ or $q = 0.83$ and $r = -0.60$, where the rounding is of 1/2 ulp accuracy in both cases, and where the limit for r is attained in both cases.

Note that in division, the difference $e(x) - e(r)$ must sometimes be allowed to be $p - 1$. At other times it is required to be at least p. Essentially, this means that the required relationship must be $|r| < |y| \cdot ulp(q)$ and not $|r| < ulp(x)$, as seen below.

For 99 divided by 6.0 we should obtain either $q = 16$ or 17 with $r = 3.0$ or $r = -3.0$, respectively. Thus the exponent difference $p - 1 = 1$ must be allowed between x and r in this case. For 110 divided by 6.0, we should obtain either $q = 18$ or 19 with $r = 2.0$ or -4.0, respectively. In this case an allowed exponent difference between x and r of $p - 1 = 1$ would not rule out the result $q = 16$, $r = 14$ and many other such undesirable results.

For the square root, simply choosing q as a partial root of the floating-point number x does not always lead to a corresponding remainder r representable in the same floating point format. Consider that $\sqrt{9200} = 95.91...$, and the partial root $q = 95$ has a corresponding remainder $r = 9200 - 95^2 = 175$. Here though, the partial root $q = 96$ would yield a corresponding remainder $r = 9200 - 96^2 = -16$.

4 Exception Handling

The following table gives a brief overview of the exceptions that can occur in each of the five exact floating point operations. The appearance of an argument name in the table indicates that this output argument can overflow or underflow, respectively. 'X' indicates an invalid operation can occur.

operation	overflow		underflow			invalid operation
$+, -$	h		l		h & l	
$*$	h	h & l	l		h & l	
$/$	q		r	q	q & r	X
$\sqrt{\ }$			r			X

There are different ways of treating an exception. Without requiring any specific actions, we propose that overflowed and underflowed arguments are scaled appropriately. Additionally, we propose that for $+, -, *$, both output arguments are scaled simultaneously in such a case so that their sum also appears scaled. The exception handling defined in the IEEE standard 854 may be followed [2].

5 Applications and Conclusion

The described exact floating point operations can be used as elementary tools to control and increase the accuracy of numerical computations. For example, they can be used to implement multiple-precision arithmetic.

Exact addition and subtraction operations are extremely useful in applications employing repeated add/subtract operations. In particular, the ill effects of leading digit cancellation can thus be avoided. This may be of great advantage in iterative refinement methods. Together with the exact floating point multiplication as presented, this allows the implementation of an exact dot product which will not be rounded until the final exact result has been computed, thus incurring only a single rounding error. Such a dot product is an invaluable tool in matrix and vector computations.

Certain exact floating point operations were previously available in hardware in the late 60's, e.g. on UNIVAC 1100 series computers for single precision. The operations described here are (at least partially) supported by hardware on many existing architectures including IEEE 754 [1] and IBM /370 arithmetic. The multiply and add instruction of the IBM RISC 6000 allows recovery of the low order term for multiply and the remainder for divide and square root. On processors which do not support these operations in hardware, they can be simulated in software using methods from [3, 4, 5, 6].

The inclusion of these operations in new programming languages such as Fortran 90 would serve to encourage manufacturers of floating point processors to fully incorporate these operations in future hardware designs. By placing such exact floating point arithmetic in the floating point unit, hardware designers could relegate integer multiply and divide to the floating point unit, and then for the rest of the processor adopt a simplistic RISC oriented design.

References

[1] ANSI/IEEE Standard 754-1985: IEEE Standard for Binary Floating- Point Arithmetic, ANSI/IEEE, New York (1985)

[2] ANSI/IEEE Standard 854-1987: IEEE Standard for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE, New York (1987)

[3] Dekker, T. J.: A floating-point technique for extending the available precision, Numerical Mathematics 18, 224-242 (1971).

[4] Kahan, W.: Further Remarks on Reducing Truncation Errors, Commun. ACM 8, 40 (1965).

[5] Linnainmaa, S.: Analysis of some known methods of improving the accuracy of floating-point sums, BIT 14, 167-202 (1974).

[6] Møller, O.: Quasi double-precision in floating-point addition, BIT 5, 37-50 (1965).