To Test Whether Binary Floating-Point Multiplication is Correctly Rounded.

W. Kahan
July 13, 1988

A floating-point arithmetic operation is *Correctly Rounded* when its rounding error does not exceed half an *ulp* (*u*nit in its *l*ast *p*lace). Something more must be said to define correct rounding in the ambiguous cases when the error is exactly half an ulp. The DEC VAX™ rounds these cases away from zero. IEEE standard 754 for Binary Floating-Point Arithmetic requires that these cases be rounded "to nearest (even)" by default; this implies that the rounded value, call it $[X \cdot Y]$, of a product $X \cdot Y$ must be the representable number nearest $X \cdot Y$ and, if there are two of those, the one whose last bit is zero. Whichever specification may be in force, we wish to test whether multiplication conforms to that specification. Our tests will be *intrinsic* in the sense that they may be executed entirely upon the very computer whose multiplication is being tested, assuming only that addition and subtraction are correctly rounded, and that multiplication and division are in error by less than one ulp.

To be specific, assume π significant bits are being carried;

$\pi = 24$ for VAX F and for IEEE 754 Single Precision,
$\pi = 53$ for VAX G and for IEEE 754 Double Precision,
$\pi = 56$ for VAX D,
$\pi = 64$ for IEEE 754 Double Extended on Intel, Motorola, and Western Electric chips,
$\pi = 113$ for VAX H and for Hewlett-Packard *Precision* architecture
IEEE 754 Double-Extended.

Unless over/underflow is at issue, scale operands X and Y so that $2^{\pi-1} \le X \cdot Y < 2^{\pi} - 1$; then the product $X \cdot Y$ must round to the integer $[X \cdot Y]$ nearest $X \cdot Y$ in the interval $2^{\pi-1} \le [X \cdot Y] \le 2^{\pi} - 1$. The critical case that arises when $X \cdot Y$ is halfway between two consecutive integers is resolved by choosing then the nearest *even* integer for $[X \cdot Y]$ to conform to IEEE 754, or the next *larger* integer on a VAX.

To test whether $X \cdot Y$ has been rounded correctly, one might generate hosts of examples at random and see what happens. However, if multiplication is nearly correctly rounded, so nearly that the error in $[X \cdot Y]$ can barely exceed half a unit in its last place, then almost all such test cases will lie far enough from halfway cases to be rounded correctly, and no incorrect cases will be observed unless the number of examples tested is gargantuan. Testing more efficiently is possible if we can systematically generate test cases that all lie on or very near the critical halfway points. The schemes outlined herein do that, and at a moderate cost far lower than undisciplined random testing. The schemes presented here assume that floating-point operations are all correct when they do not have to be rounded (when they are exact), and that addition and subtraction are already correctly rounded, and that division is in error by less than one ulp. Then these operations can be used to generate the operands that will test whether multiplication is correctly rounded when it is inexact.

Generating Halfway Cases at Random

The following scheme generates products $X \cdot Y$ that are all half-odd-integers in the binade $2^{\pi-1} < X \cdot Y < 2^{\pi}$, for which $[X \cdot Y]$ should round to the nearest even integer for IEEE 754, to the next larger integer for the VAX. (A conscientious tester will also vary randomly the exponents of X and Y to check on the correctness of rounding in other binades, using for this purpose the function *scalb* for IEEE 754, *ldexp* in C, or a table of powers of 2.0. Sign bits should be varied randomly too.)

Choose at random any *odd* integer X in the interval $2 < X < 2^{\pi}$. Next compute in floating-point two integers

$$J_L := \text{ceil}((2^{\pi} - (X-1))/(2X)) \text{ and}$$
$$J_U := \text{floor}((2^{\pi+1} - (X+1))/(2X)).$$

Each quotient can be computed with only one rounding error which, ideally, should be directed upward for J_L, downward for J_U; in the absence of directed roundings, inaccurate quotients can be remedied by adjusting these two integers so that they barely satisfy $2^{\pi-1} - (X-1)/2 \le X \cdot J_L$ and $X \cdot J_U \le 2^{\pi} - (X+1)/2$. All the terms in these inequalities can be computed exactly in floating-point. Then choose at random any integers J between J_L and J_U, as well as $J = J_L$ and $J = J_U$, from which to construct test arguments $Y := J + 1/2$ representable exactly in floating-point. (Including the extreme values J_L and J_U among the otherwise random values of J enhances the power of the tests by generating products with long chains of carries and borrows.) Now every product $X \cdot Y$ turns out to be half an odd integer in a binade where it must round to the nearest even integer for IEEE 754 , the next larger integer for a VAX. The rounded product $[X \cdot Y]$ should match the rounded sum $[X \cdot J + X/2]$ of two terms each of which is computable exactly; otherwise multiplication and addition do not round *consistently.* To test both operations for *correct* rounding, calculate $U := (X-1) \cdot J + (X-1)/2 + J$ exactly and expect to find $[X \cdot Y] = U$ if U is even and IEEE 754 is in force, otherwise $[X \cdot Y] = U + 1$, or else conclude that rounding has failed the test.

Generating Nearly Halfway Cases at Random

We shall generate odd integers X and Y at random, in the binade between $2^{\pi-1}$ and 2^{π}, of which many will satisfy either $2^{2\pi-1} < X \cdot Y < 2^{2\pi}$ and $X \cdot Y = [X \cdot Y] \pm (2^{\pi-1} - 1)$, or $2^{2\pi-2} < X \cdot Y < 2^{2\pi-1}$ and $X \cdot Y = [X \cdot Y] \pm (2^{\pi-2} - 1)$. These products come as close as possible to half-way cases without hitting one. IEEE 754 and VAX round them the same way.

Let us abbreviate $t := 2^{\pi-3}$, so that all integers between $\pm 8t$ are representable exactly in floating-point. The first step is to choose at random an odd integer i in the interval $0 < i < t$; $i := 1$, $i := 3$, $i := t - 1$ and $i := t - 3$ are good choices too. Then compute the greatest common divisor of i and $4t$ using the Euclidean algorithm provided in the appendix to get j and k too:

$$1 = \text{GCD}(i, 4t) = i \cdot j - 4t \cdot k \text{ with } 0 < j < 4t \text{ and } 0 \le k < i.$$

From this trio (i, j, k) derive a collection of quantities all computed exactly in floating-point thus:

First let $\delta := \text{sign}(2t - j) = \pm 1$ according as $2t \gtrless j$. Then compute in turn

$$\begin{array}{llll}
i_0 := i & ; \quad X_0 := 4t + i_0 & ; \quad j_0 := j & ; \quad Y_0 := 4t + j_0 \quad ; \\
i_1 := 2t + i & ; \quad X_1 := 4t + i_1 & ; \quad j_1 := 2\delta t + j & ; \quad Y_1 := 4t + j_1 \quad ; \\
i_2 := 4t - i_0 & ; \quad X_2 := 4t + i_2 & ; \quad j_2 := 4t - j_0 & ; \quad Y_2 := 4t + j_2 \quad ; \\
i_3 := 4t - i_1 & ; \quad X_3 := 4t + i_3 & ; \quad j_3 := 4t - j_1 & ; \quad Y_3 := 4t + j_3 \quad ; \\
k_{00} := k & & ; \quad k_{01} := k + i\delta/2 & & ; \\
k_{10} := k + j/2 & & ; \quad k_{11} := (k_{01} + k_{10}) - k + \delta t & & ;
\end{array}$$

for $L = 0$ to 1 do for $M = 0$ to 1 do

$$\{ \quad k_{L(M+2)} := i_L - k_{LM}; \quad k_{(L+2)M} := j_M - k_{LM};$$
$$k_{(L+2)(M+2)} := 4t - (i_L + j_M) + k_{LM} \quad \};$$

for $L = 0$ to 3 do for $M = 0$ to 3 do

$$\{ \quad l_{LM} := \text{sign}((L - 1.5)(M - 1.5)) \quad \}; \ldots = \pm 1.$$

At this point each of the intervals $(0, t)$, $(t, 2t)$, $(2t, 3t)$ and $(3t, 4t)$ contains just one i_L and just one j_M, whereupon each interval $(4t, 5t)$, $(5t, 6t)$, $(6t, 7t)$ and $(7t, 8t)$ must contain just one X_L and just one Y_M, all of them odd integers that can be represented exactly in floating-point. Moreover, all 32 products $i_L \cdot j_M$ and $X_L \cdot Y_M$ differ from multiples of $2t$ by $l_{LM} = \pm 1$. In fact,

$$i_L \cdot j_M = 4t \cdot k_{LM} + l_{LM} \text{ and}$$
$$X_L \cdot Y_M = 4t \cdot (4t + (i_L + j_M) + k_{LM}) + l_{LM};$$

but note that k_{LM} is a half-integer if $L - M$ is odd, an integer if $L - M$ is even.

Therefore the 16 products $X_L \cdot Y_M$ are all 2π or $2\pi - 1$ bits wide with $\pi - 2$ trailing bits ...00001 or ...11111; about half of the products will have the property needed to test multiplication for correct rounding, namely that they come as close as possible to half-way cases without hitting one. All that remains is to show how to compute $[X_L \cdot Y_M]$ in another way that tests whether multiplication is rounded correctly, consistently with addition.

We need formulas that express each product as a sum of two terms,

$$X_L \cdot Y_M = H_{LM} + T_{LM},$$

each of which is computable exactly. Here they are:

For $L = 0$ to 3 do for $M = 0$ to 3 do

$$\begin{array}{lll}
\{ \ \lambda_{LM} := k_{LM} - 2 \cdot \text{Floor}(k_{LM}/2); & \ldots = (2k_{LM} \bmod 4)/2 \geq 0 \\
\quad \Lambda_{LM} := k_{LM} - \lambda_{LM}; & \ldots \text{ an even integer} \leq k_{LM}. \\
\quad S_{LM} := 4t \cdot (4t + (i_L + j_M) + \Lambda_{LM}); & \\
\quad D_{LM} := 4t \cdot \lambda_{LM} + l_{LM}; & \\
\quad H_{LM} := [S_{LM} + D_{LM}]; & \ldots \text{ rounded to } \pi \text{ dignificant bits.} \\
\quad T_{LM} := (S_{LM} - H_{LM}) + D_{LM}; & \ldots \text{ exactly } \}.
\end{array}$$

Now to test whether multiplication and addition are rounded consistently, test whether

every $[X_L \cdot Y_M] = H_{LM}$. And to test whether rounding is correct, test whether

$$\begin{aligned} 2|T_{LM}| &< \text{1 ulp of } H_{LM} \\ &= |\text{NextAfter}(H_{LM}, \text{sign}(T_{LM}) \cdot \infty) - H_{LM}|. \end{aligned}$$

Still to come:

Testing directed roundings.
Testing gradual underflow.
Examples.
Complete program.

Appendix :

Program to Compute GCD$(i, 4t) = i \cdot j - 4t \cdot k$:
Based upon the Euclidean algorithm for a Greatest Common Divisor, this program starts from a given $t = 2^{\pi-3}$ and a given randomly chosen positive integer $i < t$, and yields a sequence of triples $\{ g_n, j_n, k_n \}$ that all satisfy $g_n = i \cdot j_n - 4t \cdot k_n$ while each new $|g_n| \leq |g_{n-1}|/2$ until at last $|g_{n-1}| = \text{GCD}(i, 4t) = 1$.

Initialization:
$g := 4t$; $g_1 := i$; $j := k_1 := 0$; $j_1 := 1$; $k := -1$;
Repeat {
$g_0 := g$; $g := g_1$; $j_0 := j$; $j := j_1$; $k_0 := k$; $k := k_1$;
$m :=$ integer nearest g_0/g; ... or very nearly nearest
$g_1 := g_0 - m \cdot g$; $j_1 := j_0 - m \cdot j$; $k_1 := k_0 - m \cdot k$
} until $g_1 = 0$;
If $g < 0$ then { $g := -g$; $j := -j$; $k := -k$ };
If $j < 0$ then { $j := t + j$; $k := k + i$ };

Now GCD$(i, 4t) = g = i \cdot j - t \cdot k$ with $0 < j < t$ and $0 \leq k < i$, and $g = 1$, because of the way i and t were chosen. Since all integers no bigger than $8t = 2^{\pi}$ can be represented exactly, all computations in this program except g_0/g can be carried out exactly in floating-point; and g_0/g accurate to within an ulp is accurate enough.