Number-theoretic Test Generation for Directed Rounding

Michael Parks
Sun Microsystems
Michael.Parks@sun.com

Abstract

We present methods to generate systematically the hardest test cases for multiplication, division, and square root subject to directed rounding, essentially extending previous work on number-theoretic floating-point testing to rounding modes other than to-nearest. The algorithms focus upon the rounding boundaries of the modes truncate, to-minus-infinity, and to-infinity, and programs based on them require little beyond exact arithmetic in the working precision to create billions of edge cases. We will show that the amount of work required to calculate trial multiplicands pays off in the form of free extra tests due to an interconnection among the operations considered herein. Although these tests do not replace proofs of correctness, they can be used to gain a high degree of confidence that the accuracy requirements as mandated by IEEE Standard 754 have been satisfied.

Index terms: *arithmetic testing, IEEE Standard 754, rounding functions, Hensel lifting, hyperSPARC*

1 Introduction

In this note we derive procedures to generate difficult cases for multiplication, division, and square root only for the directed rounding modes from IEEE 754. Similar tests for mode to-nearest appear already in the U.C. Berkeley test suite [8]. The algorithms produce rounding boundary cases for the directed modes via a number theory technique called Hensel lifting. Other good test methods exist, based on test vectors (see Coonen [1]) or factorization (Kahan [4]), but herein we shall rely on one kind of construction to locate the data most likely to expose anomalies.

Our tests are intended for algorithms or hardware designs which are presumed reasonably correct. They are most appropriate for iterative schemes, such as Newton's iteration or Goldschmidt's algorithm for quotients and square roots (as discussed in Koren [6]), which typically compute a close approximation to the exact function value followed by a corrective step to get the last bit or two right. An erroneous routine based on such an algorithm is most likely to fail whenever the mathematical result lies close to a rounding boundary, which for any directed rounding mode is simply a floating-point number. We seek, then, test arguments x and y such that the function values xy, x/y, or $\sqrt{x}$ have several consecutive binary zeros or ones past the least significant bit: the longer that string, the stronger the test. But to find those test arguments, we need a trick to solve a sequence of congruence equations. A simple version of Hensel lifting and a few basic notes on IEEE floating-point arithmetic are stated in the next section.

Besides numerical malice, the strength of the material comes from a relationship that furnishes free extra tests (usually for division) which offsets the expense of a long recurrence to generate trial data for multiplication. Careful deduction of inequalities ensures that all operations are exact except those under examination. Furthermore, no multiprecision arithmetic is needed because every correctly rounded result falls out effortlessly as an upshot of the computation. After deriving conditions which determine judicious test data, we will analyze and explain the yields for the square root test and display a handful of examples of boundary cases, as well as the existence of a square root bug which our tests have uncovered.

Two terms will be used often in the remaining sections. First, for a fixed precision n, the n-bit integers are those in the interval $[2^{n-1}, 2^n - 1]$. The other is congruence: We write $a \equiv b \bmod c$ if there is an integer k such that $a - ck = b$. The particular b of this form such that $0 \le b \le c-1$ is the remainder $a \bmod c$. The case where c is a positive power of two deserves special mention in this note: If $a \equiv b \bmod 2^j$, then the rightmost j bits in the binary representations of a and b are identical, and are given precisely by $a \bmod 2^j$.

2 Hensel Lifting

In order to find multiplicands whose product has a partly predetermined bit pattern, we need a technique from number theory to solve a certain type of congruence equation. In more advanced settings Hensel lifting refers to a construction from the field of "p-adic analysis" (see Koblitz [5]), but for our purposes the following simple version is sufficient.

THEOREM: Suppose f is a polynomial with integer coefficients, a is an integer such that $f(a) \equiv 0 \bmod 2^{j-1}$, and $f'(a)$ is odd. Then $f(a - f(a)) \equiv 0 \bmod 2^j$.

PROOF: Write $b = a - f(a)$, and consider $f(b) = \sum_0^N c_i b^i = \sum_0^N c_i (a - f(a))^i$. Working modulo 2^j and using the Binomial Theorem, simplify $(a - f(a))^i$ to $a^i - i a^{i-1} f(a)$ since $f(a)^i \equiv 0 \bmod 2^j$ for $i \geq 2$. Thus $f(b) \equiv \sum_0^N c_i a^i - \sum_1^N i c_i a^{i-1} f(a) \equiv f(a)(1 - f'(a))$. Since $1 - f'(a)$ is even, $f(b) \equiv 0 \bmod 2^j$.

COROLLARY: Moreover, $f(b \bmod 2^j) \equiv 0 \bmod 2^j$.

PROOF: For any positive integer j, $f(b \bmod 2^j) \equiv f(b) \bmod 2^j$, hence the claim.

The Hensel Theorem says that under mild conditions on f, any solution a_{j-1} to the congruence $f(a) \equiv 0 \bmod 2^{j-1}$ can be lifted to a solution a_j of $f(a) \equiv 0 \bmod 2^j$. In plainer language, if the last $j-1$ bits of $f(a_{j-1})$ are zeros and $a_j = a_{j-1} - f(a_{j-1})$, then the last j bits of $f(a_j)$ also are zeros, and in fact by the Corollary the last j bits of a_j will do. In the next section we recast the problem of finding critical test data for directed rounding of a product into the form of a congruence equation, and construct solutions to it using the Hensel Theorem. To ensure that the test is independent of the operation in question, we will determine conditions to restrict all intermediate terms in the algorithm to n bits.

Besides test data, the algorithms also yield the correctly rounded results. To make them transparent, we list explicit formulas for the directed rounding modes prescribed by IEEE Standard 754 [2] using the floor and ceiling functions. For positive x in the normalized range the rounding functions truncate, to-minus-infinity, and to-infinity are

$$\begin{aligned} \mathrm{trunc}(x) &= 2^{e(x)-n+1} \lfloor 2^{n-1-e(x)} x \rfloor \\ \mathrm{minf}(x) &= 2^{e(x)-n+1} \lfloor 2^{n-1-e(x)} x \rfloor \\ \mathrm{inf}(x) &= 2^{e(x)-n+1} \lceil 2^{n-1-e(x)} x \rceil \end{aligned}$$

where the precision n is understood and $e(x) = \lfloor \log_2(x) \rfloor$ is the exponent of x. We shall consider positive arguments since other tests follow from the symmetry relations

$$\begin{aligned} \mathrm{trunc}(-x) &= -\mathrm{trunc}(x) \\ \mathrm{minf}(-x) &= -\mathrm{inf}(x) \end{aligned}$$

and, as truncate and to-minus-infinity are identical from here on, the former needs no further mention. Our test arguments are scaled to locate all target products, quotients, and square roots inside one or two binades since checks near the edges of the exponent range can be deduced from the scaling properties

$$\begin{aligned} \mathrm{minf}(2^m x) &= 2^m \, \mathrm{minf}(x) \\ \mathrm{inf}(2^m x) &= 2^m \, \mathrm{inf}(x) \end{aligned}$$

Finally, the derivations are tailored to preclude overflow, underflow, and denormal numbers.

3 Multiplication

In this section we seek n-bit integers x and y in the range $[2^{n-1}+1, 2^n-1]$ whose product is as close as possible to a directed rounding boundary. Since for such a case $e(xy) = 2n-2$ or $2n-1$, the product can be written as

$$xy = 2^{n-i} p \pm k$$

where $i = 0$ or 1, k is a small integer, and p is an n-bit integer. The bigger k is, the farther xy is from a rounding boundary. We shall consider an odd y (the other case will be solved later) and the minus sign, as the solution for the plus sign will spring from the derivation. Let k be a small positive integer, and use Hensel lifting to solve

$$f(x) = xy + k \equiv 0 \bmod 2^{n-i}$$

for x by solving the sequence of congruence equations

$$x_j y + k \equiv 0 \bmod 2^j$$

for $\{x_j\}_1^n$. To solve the first congruence is easy: take $x_1 = 0$ and $p_1 = \frac{1}{2}k$ if k is even, and $x_1 = 1$ and $p_1 = \frac{1}{2}(y + k)$ if k is odd. Proceed by mathematical induction: Given $f(x_{j-1}) = x_{j-1} y + k \equiv 0 \bmod 2^{j-1}$, define the integer p_{j-1} by $x_{j-1} y + k = 2^{j-1} p_{j-1}$. Since $f'(x_{j-1}) = y$ is odd, the Hensel Theorem and Corollary imply $f(x_j) = x_j y + k \equiv 0 \bmod 2^j$, where

$$\begin{aligned} x_j &= (x_{j-1} - f(x_{j-1})) \bmod 2^j \\ &= (x_{j-1} - 2^{j-1} p_{j-1}) \bmod 2^j \\ &= (x_{j-1} + 2^{j-1} p_{j-1}) \bmod 2^j \\ &= \begin{cases} x_{j-1} & \text{if } p_{j-1} \text{ is even} \\ x_{j-1} + 2^{j-1} & \text{if } p_{j-1} \text{ is odd} \end{cases} \end{aligned}$$

(Add $2^j p_{j-1}$ to the second equation to get the third). We have established that

$$x_j y + k = 2^j p_j$$

for an integer p_j, and so by construction every solution pair (x_{j-1}, p_{j-1}) provides a successor, building x_{n-1} and x_n one bit at a time from right to left. Thus, given an odd y in $[2^{n-1}+1, 2^n-1]$ and small $k > 0$, the full algorithm is

If k is even, set $x_1 = 0$ and $p_1 = \frac{1}{2}k$,
else set $x_1 = 1$ and $p_1 = \frac{1}{2}(y + k)$

For $j = 2, \ldots, n$,
if p_{j-1} is even set $x_j = x_{j-1}$ and $p_j = \frac{1}{2}p_{j-1}$,
else set $x_j = 2^{j-1} + x_{j-1}$ and $p_j = \frac{1}{2}(y + p_{j-1})$

It is useful to bound the terms:

$$x_j \leq 2^j - 1 \quad \text{and} \quad 1 \leq p_j \leq y - 1$$

PROOFS: Since $x_1 \le 1$ and $x_j \le 2^{j-1} + x_{j-1}$ the first claim holds by induction. The lower bound for p_j is trivial, and the upper bound is also proved by induction, viz.: p_1 is either $\frac{1}{2}k$ or $\frac{1}{2}(y+k)$, so $p_1 \le \frac{1}{2}(y+k) \le y-1$ because k is a small integer. Given $p_{j-1} \le y-1$, the recurrence ensures that $p_j \le \frac{1}{2}(y+p_{j-1}) \le y - \frac{1}{2}$, but that bound can be sharpened to $p_j \le y-1$ since p_j and y are integers, hence the second claim.

Also, note that the possibility $x_{n-1} = 0$ can be eliminated, since it implies that k is an integer multiple of 2^{n-1}, which does not correspond to any rounding boundary case. It remains to infer x and p such that $xy = 2^{n-i}p \pm k$ from x_{n-1}, x_n, p_{n-1}, and p_n. Armed with $x_{n-1}y + k = 2^{n-1}p_{n-1}$ and bounds $1 \le x_{n-1} \le 2^{n-1}-1$ and $1 \le p_{n-1} \le y-1$, two derived equations

$$(2^{n-1} + x_{n-1})y = 2^{n-1}(y + p_{n-1}) - k$$
$$(2^n - x_{n-1})y = 2^{n-1}(2y - p_{n-1}) + k$$

provide five situations such that $xy = 2^{n-i}p \pm k$, but we must discard any for which p does not fit into n bits. Consider the first equation. If $p_{n-1} \le 2^n - 1 - y$, then both $x = 2^{n-1} + x_{n-1}$ and $p = y + p_{n-1}$ are n-bit integers since $2^{n-1} \le x \le 2^n - 1$ and the same for p. (Note that $2^n - 1 - y$ is computed exactly since $y \ge 2^{n-1} + 1$). Since $xy = 2^{n-1}p - k$, one pair of directed multiplication tests for correctly rounded products is

$$\mathrm{minf}(xy) = 2^{n-1}(p-1), \quad \mathrm{inf}(xy) = 2^{n-1}p$$

On the other hand if $p_{n-1} \ge 2^n - y$, then there are no worthwhile cases to pursue if p_{n-1} is even, for then $y + p_{n-1}$ is odd and exceeds 2^n, and at least $n+1$ bits are needed to represent it. As long as p_{n-1} is odd, put $x = 2^{n-1} + x_{n-1}$ and $p = \frac{1}{2}(y + p_{n-1})$ to get $xy = 2^n p - k$. (Note that the computation of p is immune to roundoff error because $2^n + 2 \le y + p_{n-1} \le 2y - 1 \le 2^{n+1} - 2$. Similar remarks apply henceforth). If $y + p_{n-1} > 2^n$ check

$$\mathrm{minf}(xy) = 2^n(p-1), \quad \mathrm{inf}(xy) = 2^n p$$

but if instead $y + p_{n-1} = 2^n$ check

$$\mathrm{minf}(xy) = (2^n - 1)p, \quad \mathrm{inf}(xy) = 2^n p$$

The latter looks unusual, but the tests are stated correctly. Infrequently, the product xy barely undercuts a power of two, and so $\mathrm{minf}(xy)$ and $\mathrm{inf}(xy)$ lie in adjacent binades. The binary representation of $\mathrm{minf}(xy)$ consists of n ones [$2^n - 1$, scaled by $p = 2^{n-1}$], and that of $\mathrm{inf}(xy)$ consists of a one followed by $n-1$ zeros [$p = 2^{n-1}$, scaled by 2^n].

The analysis for the second derived equation is similar. Suppose $p_{n-1} \ge 2y - 2^n + 1$. From the bounds upon x_{n-1} and p_{n-1}, both $x = 2^n - x_{n-1}$ and $p = 2y - p_{n-1}$ are n-bit integers in the range $[2^{n-1} + 1, 2^n - 1]$, and since $xy = 2^{n-1}p + k$, the tests are

$$\mathrm{minf}(xy) = 2^{n-1}p, \quad \mathrm{inf}(xy) = 2^{n-1}(p+1)$$

Otherwise reject the case if p_{n-1} is odd, for then $2y - p_{n-1}$ is odd and exceeds 2^n, and so does not fit into n bits. Provided p_{n-1} is even and at most $2y - 2^n$, put $x = 2^n - x_n$ and $p = y - p_n$ so that $xy = 2^n p + k$, and verify

$$\mathrm{minf}(xy) = 2^n p, \quad \mathrm{inf}(xy) = 2^n(p+1)$$

Regardless of the parity of p_{n-1}, at least one test case is found in return for the cost of the recurrence. And the smaller y is, the more likely are $y + p_{n-1} \le 2^n - 1$ and $2y - p_{n-1} \le 2^n - 1$, whereas the conditions $y + p_{n-1} \ge 2^n$ and $2y - p_{n-1} \ge 2^n$ are more likely to apply if y lies closer to $2^n - 1$ than 2^{n-1}.

The preceding test derivation can be generalized neatly to produce test cases if y is an even integer in the range $[2^{n-1} + 2, 2^n - 2]$. Write $y = 2^t y'$ where t is the greatest power of 2 that divides y; then y' must be odd. Select an integer $k = 2^t k'$ where k' is an integer, and solve

$$xy' + k' = 2^{n-i-t}p$$

by computing the solutions to

$$x_j y' + k' = 2^j p_j$$

using the full algorithm up to step $n - t$, but first replace its k by k' and y by y'. With $x_{n-1-t}, x_{n-t}, p_{n-1-t}$ and p_{n-t} at hand, consider the equations

$$(2^{n-1} + x_{n-1-t})y = 2^{n-1}(y + p_{n-1-t}) - k$$
$$(2^n - x_{n-1-t})y = 2^{n-1}(2y - p_{n-1-t}) + k$$

Analogous to the case where y is odd, there are five situations worth investigating. To distinguish them, the admissibility criteria for odd y must be slightly recast, but the statements for verification are notationally unchanged and so are not repeated.

For the first derived equation, if $p_{n-1-t} \le 2^n - 1 - y$, then set $x = 2^{n-1} + x_{n-1-t}$ and $p = y + p_{n-1-t}$ so that $xy = 2^{n-1}p - k$. Next if $y + p_{n-1-t} \ge 2^n$ and also p_{n-1-t} and y have the same parity, take $x = 2^{n-1} + x_{n-1-t}$ and $p = \frac{1}{2}(y + p_{n-1-t})$ to get $xy = 2^n p - k$.

For the second equation, if $p_{n-1-t} \ge 2y - 2^n + 1$, then set $x = 2^n - x_{n-1-t}$ and $p = 2y - p_{n-1-t}$, hence $xy = 2^{n-1}p + k$. Last, if p_{n-1-t} is even set $x = 2^n - x_{n-t}$ and $p = y - p_{n-t}$ so that $xy = 2^n p + k$ as wanted.

The preceding paragraphs actually generalize the case where y is odd, for then $t = 0$. And since it is plausible to test IEEE single precision exhaustively in a reasonable time it does no harm to require $x \ge y$ too.

Examples are shown in Table 1, using decimal form atop the conventional hexadecimal format, itself with an underscore character for the binary point and several extra digits.

x	y	$2^{n-i}xy$
8388609 4b000001	8388609 4b000001	$2^{23} \times 8388610.0000001...$ 56800002_000002...
16777213 4b7fffd	8388609 4b000001	$2^{23} \times 16777214.9999996...$ 56ffffe_fffffa...
8388611 4b000003	8388609 4b000001	$2^{23} \times 8388612.0000003...$ 56800004_000006...
12582911 4b3fffff	8388610 4b000002	$2^{23} \times 12582913.9999997...$ 56c00001_fffffc...
12582914 4b400002	8388610 4b000002	$2^{23} \times 12582917.0000004...$ 56c00005_000008...
13981013 4b555555	8388611 4b000003	$2^{23} \times 13981017.9999998...$ 56d55559_fffffe...
11184811 4b2aaaab	8388611 4b000003	$2^{23} \times 11184815.0000001...$ 56aaaaaf_000002...
11184810 4b2aaaaa	8388611 4b000003	$2^{23} \times 11184813.9999997...$ 56aaaaad_fffffc...

Table 1. Boundary cases for multiplication

4 Division

Each valid pair of unequal multiplicands provides further tests for directed division tests at no cost. Given $xy = 2^{n-i}p \pm k$, it is immediate that

$$\frac{2^{n-i}p}{x} = y \pm \epsilon_x \quad \text{and} \quad \frac{2^{n-i}p}{y} = x \pm \epsilon_y$$

for tiny $\epsilon_x = k/x$ and $\epsilon_y = k/y$. Since $xy = 2^{n-i}p - k$ is a test case for multiplication, the corresponding tests for directed division are

$$2^{n-i}\,\mathrm{minf}(p/x) = y, \quad 2^{n-i}\,\mathrm{inf}(p/x) = y + 1,$$

$$2^{n-i}\,\mathrm{minf}(p/y) = x, \quad 2^{n-i}\,\mathrm{inf}(p/y) = x + 1$$

With $xy = 2^{n-i}p + k$, check for

$$2^{n-i}\,\mathrm{minf}(p/x) = y - 1, \quad 2^{n-i}\,\mathrm{inf}(p/x) = y,$$

$$2^{n-i}\,\mathrm{minf}(p/y) = x - 1, \quad 2^{n-i}\,\mathrm{inf}(p/y) = x$$

The examples shown in Table 2 correspond to various rows in Table 1. Each has at least 21 consecutive like bits after the binary point; the first has 23 ones, which is provably maximal. These test data are far more likely to uncover a defective iterative division algorithm than random tests, and for very wide precisions no practical amount of random testing will stumble upon these patterns.

A complete derivation of test generation for square root is given in the next section. For the moment, note that if $xy \pm k = 2^{n-i}p$ and $x = y$, then that multiplication test provides one test for directed division, and another for directed square root as follows. The derivation depends on the indicated sign and whether the exponent $n - i$ is even or odd. If the plus sign is used and $n - i$ is even, say $2m$, then $x^2 + k = 2^{2m}p$, and two square root tests are

$$2^m\,\mathrm{minf}(\sqrt{p}) = x, \quad 2^m\,\mathrm{inf}(\sqrt{p}) = x + 1$$

But if $n - i$ is odd, say $2m + 1$, then $x^2 + k = 2^{2m+1}p$. Certainly $2p$ is representable if p is, and so

$$2^m\,\mathrm{minf}(\sqrt{2p}) = x, \quad 2^m\,\mathrm{inf}(\sqrt{2p}) = x + 1$$

Other tests associated with the equation $x^2 - k = 2^{n-i}p$ are obtained in a similar manner. For an even exponent, check

$$2^m\,\mathrm{minf}(\sqrt{p}) = x - 1, \quad 2^m\,\mathrm{inf}(\sqrt{p}) = x,$$

and for odd exponent

$$2^m\,\mathrm{minf}(\sqrt{2p}) = x - 1, \quad 2^m\,\mathrm{inf}(\sqrt{2p}) = x$$

For example, from Table 1, we know that $8388609^2 - 1 = 2^{23} \times 8388610$, and so

$$2^{11}\sqrt{2 \times 8388610} = 8388608.99999994...$$

Another case is $16777215^2 - 1 = 2^{24} \times 16777214$, with

$$2^{12}\sqrt{16777214} = 16777214.99999997...$$

This trick can not be exploited to find division or square root boundary cases for mode to-nearest, because for that problem xy (or x^2) must nearly equal the average of consecutive n-bit floating-point numbers, and exactly $n+1$ bits are required to represent that midpoint.

$2^{n-i}p$	x	$2^{n-i}p/x$
$2^{23} \times 8388610$	8388609	8388608.9999998...
56800002	4b000001	4b000000_fffffe...
$2^{23} \times 16777215$	16777213	8388609.0000001...
56ffffff	4b7ffffd	4b000001_000003...
$2^{23} \times 8388612$	8388611	8388608.9999996...
56800004	4b000003	4b000000_fffffa...
$2^{23} \times 12582914$	12582911	8388610.0000001...
56c00002	4b3fffff	4b000002_000002...

$2^{n-i}p$	y	$2^{n-i}p/y$
$2^{23} \times 16777215$	8388609	16777213.0000003...
56ffffff	4b000001	4b7ffffd_000005...
$2^{23} \times 8388612$	8388609	8388610.9999996...
56800004	4b000001	4b000002_fffffa...
$2^{23} \times 12582914$	8388610	12582911.0000002...
56c00002	4b000002	4b3fffff_000003...
$2^{23} \times 12582913$	8388610	12582910.0000004...
4b400001	4b000002	4b3ffffe_000007...

Table 2. Boundary cases for division

5 Square Root

This section establishes tests for square root under directed rounding, normalizing the problem so that the arguments to fall into two adjacent binades. We seek large integers x so that $\sqrt{x} = z \pm \epsilon$ where z is an n-bit integer and $\epsilon > 0$ is tiny, so that $x = z^2 - k$ for a small integer k of either sign. Since $e(x) = 2n - 2$ or $2n - 1$ and x must be expressible using at most n bits, we must have $x \equiv 0 \bmod 2^{n-i}$ where $i = 0$ or 1. Therefore

$$z^2 \equiv k \bmod 2^{n-i}$$

and the smaller $|k|$ is, the closer $\sqrt{x}$ is to a directed rounding boundary. Suppose k is odd (generalized later), then z must be odd also, and consequently $k \equiv 1 \bmod 8$. To generate test integers, choose k in $\{..., -15, -7, 1, 9, 17, ...\}$ and produce corresponding z by solving the congruences

$$z_j^2 \equiv k \bmod 2^j \qquad \text{for } j = 3, ..., n$$

Strictly speaking the Hensel Theorem in Section 2 is too specialized to bring the next recurrence to light, but it can be inferred from the discussion of "p-adic arithmetic" in Koblitz [5], and is easy to verify.

Set $z_3 = 1$ and let $z_j^2 - k = 2^j R_j$

For $j = 4, ..., n$,
  if R_{j-1} is even, set $z_j = z_{j-1}$ and $R_j = \frac{1}{2} R_{j-1}$
  else set $z_j = 2^{j-2} - z_{j-1}$ and
    $R_j = 2^{j-4} + \frac{1}{2}(R_{j-1} - z_{j-1})$

A few relationships among the solutions to the recurrence borrowed from Kahan [3] will assist in the derivation of test cases and indicate the correctly rounded results. For any $k \equiv 1 \bmod 8$ the congruence $z^2 \equiv k \bmod 2^j$ has infinitely many positive solutions. If z_j is the smallest, then the four smallest solutions are

$$\{z_j, 2^{j-1} - z_j, 2^{j-1} + z_j, 2^j - z_j\}$$

lying in isometric integer ranges

$$[1, 2^{j-2} - 1], \quad [2^{j-2} + 1, 2^{j-1} - 1],$$

$$[2^{j-1} + 1, 3 \times 2^{j-2} - 1], \quad [3 \times 2^{j-2} + 1, 2^j - 1]$$

respectively. For $j = n - i$, whichever solutions z satisfy $2^{2n-1-i} \le z^2 - k \le 2^{n-i}(2^n - 1)$ provide difficult test integers $x = z^2 - k$. Infer that the tests are

$$\begin{array}{llll} \text{minf}(\sqrt{x}) = z, & \text{inf}(\sqrt{x}) = z + 1 & \text{if } k < 0 \\ \text{minf}(\sqrt{x}) = z - 1, & \text{inf}(\sqrt{x}) = z & \text{if } k > 0 \end{array}$$

The analysis of whether $z^2 - k$ lies in an acceptable range splits into two cases. First, for $i = 1$, use the recurrence to solve the congruence $z_{n-1}^2 \equiv k \bmod 2^{n-1}$ to get four solutions $\{r_1, r_2, r_3, r_4\} = \{z_{n-1}, 2^{n-2} - z_{n-1}, 2^{n-2} + z_{n-1}, 2^{n-1} - z_{n-1}\}$. It turns out that no such solution is large enough for x to lie in $[2^{2n-2}, 2^{n-1}(2^n - 1)]$, but there is a remedy. We will show that $s_1 = 2^{n-1} + r_1$ leads to a valid test integer, as does $s_2 = 2^{n-1} + r_2$ if it is small enough. Note $s_1^2 \equiv s_2^2 \equiv k \bmod 2^{n-1}$ as required.

LEMMA: s_1 is admissible, and s_2 is admissible if $s_2 < c_1 = \sqrt{2^{2n-1} - 2^{n-1}}$

PROOFS: Assume that $0 < \epsilon \leq 1/2^5$ so that $|k| \leq 2^{n-4}$ and $\sqrt{x}$ has at least four consecutive zeros or ones past the binary point. For all k of interest, $s_1^2 - k \geq (2^{n-1}+1)^2 - 2^{n-4} > 2^{2n-2}$, and $s_1 < 2^{n-1}+2^{n-3}$, so $s_1^2 - k < (2^{n-1}+2^{n-3})^2 + 2^{n-4} < 2^{n-1}(2^n - 1)$. Since $s_2 > s_1$ we have $s_2^2 - k > 2^{2n-2}$. We must also decide whether $s_2^2 - k \leq 2^{2n-1} - 2^{n-1}$, or equivalently whether $s_2^2 - 2^{2n-1} + 2^{n-1} \leq k$. The parabola $k(r) = r^2 - 2^{2n-1} + 2^{n-1}$ is extremely steep on the domain of those r for which $|k| \leq 2^{n-4}$, so steep that comparing s_2 to the positive zero c_1 of $k(r)$ is enough to determine whether $s_2^2 - k \leq 2^{n-1}(2^n - 1)$.

In brief, if k is not too large then the solutions of the congruence can be checked quickly for validity. The same is true for the second case, $i = 0$, for which we use the recurrence to solve $z_n^2 \equiv k \bmod 2^n$ for $\{r_1, r_2, r_3, r_4\} = \{z_n, 2^{n-1} - z_n, 2^{n-1} + z_n, 2^n - z_n\}$. We will show that r_4 is in $[2^{2n-1}, 2^n(2^n - 1)]$, as is r_3 if large enough.

LEMMA: r_4 is always admissible, and r_3 is admissible if $r_3 \geq c_0 = 2^{n-1/2}$

PROOFS: For all k of interest, $r_4^2 - k \geq (3 \times 2^{n-2} + 1)^2 - 2^{n-4} > 2^{2n-1}$, and $r_4^2 - k \leq (2^n - 1)^2 + 2^{n-4} < 2^n(2^n - 1)$. Next, $r_3^2 - k < r_4^2 - k < 2^n(2^n - 1)$, so r_3 is never too large. It remains to determine whether $r_3^2 - k \geq 2^{2n-1}$, or equivalently $k \leq r_3^2 - 2^{2n-1}$. But the r-coordinates of the intersection of the parabola $k(r) = r^2 - 2^{2n-1}$ with lines $k = \pm 2^{n-4}$ hardly differ, so it suffices to compare r_3 to the zero c_0 of $k(r)$.

From here on, let r_3 and r_4 refer to the case $i = 0$ only. The recurrence always provides two integers, one for $i = 1$ and another for $i = 0$. But more than three is impossible:

THEOREM: No k produces four test integers

PROOF: If R_{n-1} is even, then $z_n = z_{n-1}$, so $r_3 = 2^{n-1} + z_n = 2^{n-1} + z_{n-1} = s_1$. Hence $r_3^2 - k = s_1^2 - k \leq 2^{n-1}(2^n - 1) < 2^{2n-1}$, and therefore r_3 is too small to produce a test integer. If R_{n-1} is odd, then $z_n = 2^{n-2} - z_{n-1}$, so $r_3 = 2^{n-1} + 2^{n-2} - z_{n-1} = s_2$. If also s_2 is admissible, then $r_3^2 - k \leq 2^{n-1}(2^n - 1) < 2^{2n-1}$.

Either two or three integers are produced for each k. If distributed evenly within its range, solution s_2 would be valid about $(c_1 - 5 \times 2^{n-3})/2^{n-3} \approx 4\sqrt{2} - 5 = 65.7\%$ of the time, and r_3 would be valid about $(3 \times 2^{n-2} - c_0)/2^{n-2} = 3 - 2\sqrt{2} = 17.2\%$ of the time. For odd k, our implementation corroborates those figures, and averages 2.828 (looks suspiciously like $2\sqrt{2}$) integers per k when testing double precision. A similar analysis ought to explain the percentages in the last section of Kahan [3].

At any rate, each trial case can be computed economically. In the chart below, each parenthesized expression fits into n bits, since $|R_j| < 2^{j-4}$ and $z_j < 2^{j-2}$ for all j large enough, again, as proved in Kahan's note.

$2^{n-i}x$	$\sqrt{2^{n-i}x}$
$2^{23} \times 8388610$ `56800002`	8388608.99999994... `4b000000_ffffff0`...
$2^{24} \times 16777214$ `577ffffe`	16777214.99999997... `4b7ffffe_ffffff7`...
$2^{23} \times 8388612$ `56800004`	8388609.9999997... `4b000001_fffffc`...
$2^{24} \times 16777212$ `577ffffc`	16777213.9999998... `4b7ffffd_fffffd`...
$2^{23} \times 11946704$ `56b64ad0`	10010805.0000003... `4b18c0b5_000005`...
$2^{23} \times 14321479$ `56da8747`	10960715.0000003... `4b273f4b_000005`...
$2^{24} \times 13689673$ `5750e349`	15155019.0000002... `4b673f4b_000003`...
$2^{23} \times 8388614$ `56800006`	8388610.9999994... `4b000002_fffff7`...
$2^{24} \times 16777210$ `577ffffa`	16777212.9999997... `4b7ffffc_fffffb`...
$2^{23} \times 10873622$ `56a5eb16`	9550631.0000007... `4b11bb27_00000d`...

Table 3. Boundary cases for square root

For solution z ...	compute $x = z^2 - k$ via ...
$2^{n-1} + z_{n-1}$	$2^{n-1}(2^{n-1} + (R_{n-1} + 2z_{n-1}))$
$3 \times 2^{n-2} - z_{n-1}$	$2^{n-1}(9 \times 2^{n-3} + (R_{n-1} - 3z_{n-1}))$
$2^{n-1} + z_n$	$2^n(2^{n-2} + (R_n + z_n))$
$2^n - z_n$	$2^n(2^n + (R_n - 2z_n))$

As for multiplication, there is a generalization to find more square root test cases if k is an even integer. If there are any test integers to find for even k, then the corresponding z must be even as well, and so k must be divisible by 4 to begin with. Write $k = 4^t k'$ where t is the greatest power of 4 which divides k, and let $z = 2^t z'$. Both k' and z' are odd, so we must have $k' \equiv 1 \bmod 8$. To obtain trial cases, solve the congruences $z_j'^2 \equiv k' \bmod 2^{n-i-2t}$ using the lifting algorithm up to step $n - i - 2t$, but first replace its z_j by z_j' and k by k'. Then set $z_{n-1} = 2^t z'_{n-1-2t}$ and $z_n = 2^t z'_{n-2t}$ to solve the congruences $z^2 \equiv k \bmod 2^{n-i}$.

Admissible solutions are $s_1 = 2^{n-1} + z_{n-1}$ and possibly $s_2 = 2^{n-1} + 2^{n-2} - z_{n-1}$ (for $i = 1$), along with $r_4 = 2^n - z_n$ and possibly $r_3 = 2^{n-1} + z_n$ (for $i = 0$). At most three test integers are computed using a modified version of the chart; just replace each instance of R_{n-1} by R_{n-1-2t} and R_n by R_{n-2t}.

$2^{n-i}x$	Expected minf$(\sqrt{2^{n-i}x})$	Computed minf$(\sqrt{2^{n-i}x})$
$2^{53} \times 8732221479794286$	8868644699447394	8868644699447395
468f05e8bf67366e	433f81fc40f32062	433f81fc40f32063
$2^{52} \times 8550954388695124$	6205648636410980	6205648636410981
467e610c36d42854	43360c012a92fc64	43360c012a92fc65
$2^{52} \times 7842344481681754$	5942960515215712	5942960515215713
467bdc921d09715a	43351d17526c7160	43351d17526c7161
$2^{52} \times 5935035262218600$	5170011856404048	5170011856404049
467515e21488b168	43325e19302f7e50	43325e19302f7e51
$2^{52} \times 5039650445085418$	4764091504847932	4764091504847933
4671e7890e924aea	4330ecea7dd2ec3c	4330ecea7dd2ec3d
$2^{52} \times 5039721545366078$	4764125111052576	4764125111052577
4671e7999c7a7a3e	4330ecf250e8e920	4330ecf250e8e921
$2^{52} \times 8005963117781324$	6004635918520114	6004635918520115
467c71618bb8614c	4335552f3eedcf32	4335552f3eedcf33
$2^{52} \times 6703494707970582$	5494529667669144	5494529667669145
4677d0cafcd28216	4333853ee10c9c98	4333853ee10c9c99
$2^{52} \times 8010323124937260$	6006270743197038	6006270743197039
467c7558b065e22c	433556abe212b56e	433556abe212b56f
$2^{53} \times 8010776873384260$	8494390118415981	8494390118415982
468c75c255e9b944	433e2d9a51977e6d	433e2d9a51977e6e

Table 4. Erroneous double precision square roots in an early hyperSPARC

Finally, note that if $t = 0$ then the previous two paragraphs handle the case where k is odd, so the generalized algorithm runs through the set

$$\{..., -28, -23, -15, -7, 1, 4, 9, 16, 17, 25, 33, 36, ...\}$$

of integers $k = 4^t k'$ with $t \geq 0$ and $k' \equiv 1 \mod 8$. About one-sixth of all integers in any large symmetric range qualify, empirically yielding 2.67 test integers per k on average.

Some examples are shown in Table 3. The first two have 24 consecutive ones past the binary point, and each square root differs from the nearest 24-bit integer by less than 2^{-20}.

6 A Square Root Bug

An application of the tests in the previous section exposed in six seconds a flawed double precision square root operation in a hyperSPARC microprocessor.[1] A handful of erroneously computed square roots are listed in Table 4.

In this section we will estimate the frequency and total number of failures, and speculate on the implementation of this particular hyperSPARC's square root algorithm. To shorten the analysis, we note that if the computed value of minf$(\sqrt{x})$ is incorrect, always too large by one unit in its last position, then the computed value of inf$(\sqrt{x})$ is likewise too large. Henceforth we will focus solely on minf$(\sqrt{x})$, since the details for inf$(\sqrt{x})$ are analogous.

The square root error spans all binades in the normalized range, but we will study a fundamental domain of scaled 53-bit integers within the interval $[2^{104}, 2^{53}(2^{53} - 1)]$, whose square roots lie near 53-bit integers in $[2^{52}, 2^{53}]$. We have found failures only when $\sqrt{x}$ lies just below a rounding boundary z, the integer nearest $\sqrt{x}$. That is, for some test integers x such that $\sqrt{x} = z - \epsilon$ for tiny $\epsilon > 0$, the hyperSPARC algorithm returns z for minf$(\sqrt{x})$, in violation of IEEE 754 since minf$(\sqrt{x}) = z - 1$.

How far can $\sqrt{x}$ land from an integer with minf$(\sqrt{x})$ nonetheless miscalculated by the hyperSPARC chip? We have observed that, over all arguments x where the binary form of $\sqrt{x}$ has exactly m consecutive ones past bit 53:

If $m \leq 6$, it appears that this algorithm always computes minf$(\sqrt{x})$ correctly.

If $7 \leq m \leq 34$, the algorithm computes at least one minf$(\sqrt{x})$ wrong.

If $m \geq 35$, the algorithm computes minf$(\sqrt{x})$ right!

Certainly if the binary representation of $\sqrt{x}$ has exactly m consecutive ones past bit 53, then

$$\frac{1}{2^{m+1}} < z - \sqrt{x} \leq \frac{1}{2^m}$$

[1] It is not a hyperSPARC made by Sun Microsystems.

Therefore, for this particular square root algorithm, if

$$\sqrt{x} < z - \frac{1}{2^7} \quad \text{or} \quad \sqrt{x} \geq z - \frac{1}{2^{35}}$$

then $\text{minf}(\sqrt{x})$ is computed correctly, but if

$$z - \frac{1}{2^7} \leq \sqrt{x} < z - \frac{1}{2^{35}}$$

then the computed $\text{minf}(\sqrt{x})$ might be wrong.

For the argument $x = 2^{53} \times 8218582589514112$, with

$$\sqrt{x} = 8603860236852920.995909...$$

the hyperSPARC algorithm incorrectly rounds $\sqrt{x}$ down to $z = 8603860236852921$, which exceeds $\sqrt{x}$ by about 1.047×2^{-8}. Of all the stray square roots we have found, this one lands farthest from an integer.

To investigate the error on a larger scale, we have used the square root test for $\text{minf}(\sqrt{x})$ from the previous section using 2^{37} positive k in increasing order through the set $\{1, 9, 17, ..., 2^{40} - 7\}$, to produce a sequence $\{x_i\}$ of over 388 billion candidates to target the rounding boundaries of $\text{minf}(\sqrt{x})$ in four months of computing time. With k increasing as shown in Table 5, the trail candidates were produced in roughly descending order of difficulty, in the sense that the corresponding sequence $\{m_i\}$ to count the number of consecutive ones past bit 53 of $\{\sqrt{x_i}\}$ is (nearly) monotonically decreasing. Note that in the long run, the number of test arguments produced for each time through the Hensel lifting recurrence is

$$\frac{388735554607}{2^{37}} = 2.82842341...$$

which agrees with $2\sqrt{2} = 2.82842712...$ (the figure suspected in the previous section) to six decimals.

range of k	test arguments	failures	hit rate
$1 - 2^{18}$	184925	0	none
$2^{19} - 2^{20}$	370163	3	1 in 123387
$2^{20} - 2^{25}$	11860787	81	1 in 146429
$2^{25} - 2^{30}$	379616108	2596	1 in 146231
$2^{30} - 2^{35}$	12147900900	83659	1 in 145207
$2^{35} - 2^{40}$	388735554607	2461844	1 in 157904

Table 5. Number of errors found

To estimate very roughly the total number of failing cases of $\text{minf}(\sqrt{x})$ for x in the fundamental domain, an alternate chart has been constructed in Table 6. Each row counts the number of arguments x for which the binary representation of $\sqrt{x}$ has exactly m consecutive ones past bit 53, but for which $\text{minf}(\sqrt{x})$ was computed erroneously.

A complete list for $7 \leq m \leq 13$ required more computing time than available, but for smaller m we will guess the number of arguments x with incorrectly computed $\text{minf}(\sqrt{x})$ by assuming for the sake of discussion that for $7 \leq m \leq 13$ the ratio of counts from one m to the next smaller m is constant at 1.97. Assume further that there are no failures for test arguments with $m \leq 6$, since no such case has been found. These conjectures provide the figures in the last half-dozen rows of Table 6.

m	failures
34	2
33	2
32	4
31	3
30	13
29	35
28	52
27	93
26	246
25	412
24	908
23	1777
22	3621
21	7024
20	14220
19	28507
18	56411
17	112802
16	222368
15	435481
14	836914
13	1648720
Projected	
12	3247978
11	6398516
10	12605076
9	24831999
8	48919038
7	96370504

Table 6. Classification of erroneous roots

Under these suppositions, there are around 195 million integer arguments x within the fundamental domain for which the computed value of $\text{minf}(\sqrt{x})$ is incorrect. Since there are 2^{54} arguments in that domain, the chances of finding a failure purely at random is estimated at 1 in 92 million.

We can speculate on structure of the flawed algorithm based on the clues just derived, again considering only the fundamental domain. The hyperSPARC algorithm computes a number r near $\sqrt{x}$, an under-approximation which is

perhaps stored internally as a sum, and is accurate to about 59 bits. A test for correctness is applied; if in particular $\sqrt{x}$ happens to have 35 or more consecutive ones past bit 53, then the correctly rounded results are returned. But if the test fails, the algorithm attempts to compute a correction term s such that the exact sum $r + s$ approximates $\sqrt{x}$ even more accurately that r does. For some (evidently around 1 in 150000) arguments x such that

$$\frac{1}{2^{35}} < z - \sqrt{x} \leq \frac{1}{2^{7}}$$

the computed value s' of s is botched in such a way that $r + s' > z$. Hence the computed values of $\operatorname{minf}(\sqrt{x})$ and $\inf(\sqrt{x})$ wind up too large by one unit in the last position.

The estimates of failure rates show conclusively that focused tests are far and away more efficient and effectual than random tests, and so the cost of the recurrence associated with Hensel lifting is well worth paying. Guesswork aside, the unexpected discovery of the flaw underscores further the importance of searching thoroughly for errors at and near the singularities of the function under examination.

7 Conclusion

We developed methods to generate in descending order of difficulty test data for three fundamental operations by using lifting techniques from number theory. They should be used to reinforce suspicion that an algorithm or hardware design conforms to the expectations for directed rounding imposed by the IEEE Standard for binary arithmetic.

We rigorously established that no intermediate operations depend upon rounded sums, differences, or products, and therefore programs to implement these algorithms should require very little beyond exact addition and subtraction, along with multiplication by powers of two and small constants. The correctly rounded results for validation are apparent without extra work, and further tests can be finessed from each multiplication case at no cost. Moreover, we have demonstrated that extreme cases on both sides of at least a few rounding boundaries always exist; any accuracy claims for directed rounding algorithms to the effect that "the worst case error is 0.998 units in the last position of the computed result" (or some smaller figure) are invalid.

The discussion herein generalizes previous work on deriving these kinds of trials (in Kahan [3] and Tang [7]) which did not tackle the problem for even multiplicands, divisors, and radicands. Sufficiently adequate properties of the rounding modes have been put forth to construct tests for sign combinations and function values at the edges of the exponent range. To conclude, these methods give rise to the rarest of situations, rendering errors in these particular directed rounding operations practically inexcusable.

8 Acknowledgments

Doug Priest and Alex Liu of Sun Microsystems offered many valuable remarks as these notes and programs were in progress. I am indebted to Professor William Kahan, K. C. Ng of Sun Microsystems, and P. T. Peter Tang, whose previous work led to this note, and also to my thesis advisor, Professor Beresford N. Parlett, for insightful comments.

The reviews of the referees were appreciated.

References

[1] Jerome T. Coonen, Contributions to a Proposed Standard for Binary Floating-Point Arithmetic, Ph. D. thesis, University of California at Berkeley, 1984.

[2] IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute for Electrical and Electronics Engineers, New York, 1985.

[3] William Kahan, A Test for Correctly Rounded SQRT, `http://www.cs.berkeley.edu/~wkahan/SQRTest.ps`.

[4] William Kahan, Checking Whether Floating-Point Division is Correctly Rounded, manuscript, April 11, 1987.

[5] Neal Koblitz, p-adic Numbers, p-adic Analysis, and Zeta-Functions, Graduate Texts in Mathematics, volume 58, Springer-Verlag, 1984.

[6] Israel Koren, Computer Arithmetic Algorithms, Prentice Hall, 1993.

[7] Ping Tak Peter Tang, Testing Computer Arithmetic by Elementary Number Theory, Preprint MCS—P84—0889, Mathematics and Computer Science Division, Argonne National Laboratory, August 1989.

[8] The U.C. Berkeley test suite, available from Netlib, `http://www.netlib.org/fp/ucbtest.tgz`.