Beastly Numbers

Prof. W. Kahan
Departments of Mathematics, and of
Elect. Eng. & Computer Science #1776
University of California
Berkeley CA 94720-1776

Abstract:

It seems unlikely that two computers, designed by different people 1800 miles apart, would be upset in the same way by the same two floating-point numbers 65535.⋯ and 4294967295.⋯ , but it has happened.

Introduction:

Do you have a favorite number? Some folks like 7 . Many avoid 13 . A few are averse to 666 :

" Let him that hath understanding count the number of the beast:
for it is the number of a man; and his number is
Six hundred threescore and six." *Revelations* 13:17

Such prejudices would be considered unseemly among Numerical Analysts, whose "Equal Opportunity Employer" attitude towards numbers may be all that distinguishes us from Numerologists. Among computers, a predilection towards one number is considered a fault, caused perhaps by a burnt-out transistor. An aversion to one number occurs so rarely among computers that no plausible explanation for it comes to mind. But it has happened.

My story begins with an idea that turned out not to be good enough to deserve explanation in detail. It required an estimate for an integer, an Order of Convergence, which, to be useful, had to be a small positive integer best not over-estimated. To generate that estimate, my program used *Directed Rounding*, a feature available in all hardware that conforms to IEEE Standard 754 for Binary Floating-Point Arithmetic, albeit unsupported by almost all compilers. My program computed a floating-point number H(x) , guaranteed to under-estimate the desired integer despite roundoff, and then rounded it up to the nearest 2-byte integer j thus:

```
j  :=  RoundUpToInteger( H(x) ) .
```

Then j was tested; when it was negative or too big, it was discarded; otherwise it served in an indexed branch (a *Computed* GO TO ⋯ in Fortran) to a subprogram intended to accelerate the convergence of a presumably (j+1)th-order calculation involving x .

My program did not accelerate convergence much and sometimes retarded it, so the idea explored by the program would have been deemed worthless except for one puzzling instance among its results: my program ran inordinately slowly in one instance when convergence should have been extremely fast. There appeared to be a bug in my program. Had that bug degraded acceleration? If so, the idea being explored had not been given a fair trial, and might be worthwhile after all.

I could not find the bug. I distrusted the compiler but, poring over assembly-language listings, could find no fault with it. Days and nights turned into weeks of scrutiny of redundant calculation, auxiliary calculation, exploratory calculation, divide-and-fail-to-conquer, ⋯ .

It had to be a bug in my hardware, a PC based upon the Intel 386.

Two Computers Better than One:

An Intel Pentium® became available. I have written elsewhere about its notorious FDIV bug; this turns out not to be germane here except that, while exploring its peccadillos, I ran my program described above and got practically the same results as before. The discrepancies were compatible with end-figure differences in the transcendental functions computed differently on Pentium than on prior Intel processors. Besides, the computer on which my program had first been run used a Cyrix 83D87 in place of an Intel 387 numeric coprocessor. Cyrix and Intel employ sufficiently different algorithms for their floating-point operations that their close agreement was tantamount to a jury verdict against my program: it was guilty of harboring a bug.

I could not find that bug in my program. I gave up looking for it.

Meanwhile, Intel's Jon Marshall found an occasion to mention that Pentium's FDIV bug might not be its only defect. Its Store-to-Integer instruction FIST deserved scrutiny too. To test it, I wrote a program FISTest that explored all its *singularities* (boundary cases). (*)

Conversion to Integer:

FIST converts a floating-point number to a 2's-complement binary integer 2 or 4 or 8 bytes wide, as selected by the programmer, in accordance with IEEE 754. That standard requires the conversion operation to take account of the prevailing direction of rounding — *to_nearest*, *to_zero*, *upward* or *downward* — unless it is overridden by the programmer or by language conventions. Most languages have such conventions; for example, Fortran rounds integer conversions *to_zero*, so the compiler has to either change the prevailing direction of rounding before conversion and restore it afterward, or else choose a special conversion instruction if the hardware provides one for that purpose. I had learned for previous programs how to override the compiler's override, so it posed no impediment to writing FISTest programs for 2- and 4-byte integer conversions. (My compilers lack support for 8-byte 2's-complement integers.)

What happens when conversion produces an integer too big to fit into the selected 2's-complement destination? IEEE 754 is a little vague about that. The most common outcome, if computation is not interrupted, is that the *Invalid Operation Flag* gets raised and the oversize integer gets supplanted by the biggest in magnitude that will fit into the intended destination and has either the same sign (Motorola 68xxx) or a negative sign (Intel x86/87 and Cyrix 83D87). Such an outcome serves the purposes of the assignment to j in my buggy program, so integer overflow could not cause that bug I could not find. Or so I thought.

FISTest found otherwise. Whenever $H(x) = 65535.\cdots$ rounded up to $65536.0 = 2^{16}$, Pentium's FIST set j to 0 instead of -2^{15} as Intel's and Cyrix's specifications require. When H(x) rounded to any other integer bigger than $2^{15} - 1$, or when $H(x) = 2^{16}$ exactly, FIST set j to -2^{15} correctly. The number 2^{16} was disliked only when FIST stored to 2-byte integers; for storing to 4-byte integers, $4294967296 = 2^{32}$ replaced 2^{16}. The 83D87 did likewise and more; its FIST stored 0 also when rounding down to -2^{16} into 2 bytes, to -2^{32} into 4 bytes.

That explains the bug in my program; it was not in my program. It was two bugs in two different hardware designs but with the same effect. How two competing manufacturers could admit such similar yet different bugs into their hardware is still unexplained.

Where are those bugs now?

Imagine the temptation to do nothing about them: each manufacturer could have excused inaction by citing a need to maintain compatibility with the other. Almost. Actually, remedial actions were under way before either manufacturer learned about the other's infestation.

According to Bob McGhee of Cyrix, only the earliest versions of the 83D87 suffered from those bugs. FISTest has found nothing to complain about in a current version.

According to Intel's *Pentium Processor Specification Update* (Order no. 242480-001) dated February, 1995, Pentium's FIST bugs had been removed from 75-, 90- and 100-MHz Pentiums dated 1994 or later, but had not yet been removed from 60- and 66-MHz Pentiums. Moreover, that document reveals another bug that my first versions of FISTest overlooked:

> Pentium's FIST rounded the fractions ±0.0625 , ±0.125 and ±0.1875 to integer 0 regardless of the rounding direction requested by the program, so rounding away from 0 failed to produce ±1 ; and the Inexact flag required (perversely, I think) by IEEE 754 was set incorrectly for those six numbers too. Intel's recommended work-around is to use the FRNDINT instruction (RouND to INTeger-valued Float) first.

These bugs appear to have arisen when Pentium's FIST algorithm was "optimized" to save time wasted on redundant shifting by the algorithm used for FIST on earlier Intel 80x87 and 486 chips, which did something like FRNDINT before storing. New circuitry must have been introduced to round off fractional bits independently of the circuitry that rounds other floating-point operations. What (il)logical design could create an aversion to six innocuous fractions? What testing strategy could be expected to expose such an aversion without foreknowledge?

Conclusions:

Notwithstanding the foregoing misadventure, the best way to distinguish a bug in software from a system bug is still to run suspect software on two appropriately different computer systems. And, for all that a reader's confidence in tests must be shaken by the foregoing account, testing remains the only way we know to corroborate both the designs believed correct and the alleged proofs of their correctness. Seeking out singularities remains the best strategy for economical tests, though it does depend upon foreknowledge of singularities both intrinsic, in the function being tested, and accidental, in the function's implementation. Without foreknowledge about how FIST was implemented, FISTest could not reasonably be expected to discover that three isolated fractions 1/16 , 1/8 and 3/16 are mishandled. (FISTest 's current version finds them.) And without some foreknowledge about the implementation under test, a test program can too easily use an algorithm that coincides accidentally with the tested implementation's, thereby sharing its defects.

On the other hand, a test program can be corrupted by too much foreknowledge if thereby the tester and the implementor share a misconception. I see no easy way around the dilemma here.

Testing works best on designs created with testing in mind. That kind of design is still more of an art than a science. To cultivate this art among designers, they should occasionally be assigned the task of testing other designers' creations. To rotate creative design engineers through diverse assignments – design, testing, manufacture, marketing support – is neither common nor popular as a personnel policy, but it does produce better products and better engineers.

(*) FISTest is available to anyone who sends me a floppy diskette for it and a stamped self-addressed envelope.