3.9 Fallacies and pitfalls

> "Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.
> *Bertrand Russell, Recent Words on the Principles of Mathematics, 1901.*

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.

Recall that a binary number *x*, where *xi* means the *i*th bit, represents the number

$$\cdots + (x^3 \times 2^3) + (x^2 \times 2^2) + (x^1 \times 2^1) + (x^0 \times 2^0)$$

Shifting the bits of *x* right by *n* bits would seem to be the same as dividing by 2*n*. And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}. The two's complement representation of -5_{ten} is

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111011_{two}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$00111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $4{,}611{,}686{,}018{,}427{,}387{,}902_{ten}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

The result is -2_{ten} instead of -1_{ten}; close, but no cigar.

PARTICIPATION ACTIVITY 3.9.1: Unsigned and signed integer division.

1) Shifting the bits of an unsigned integer right by n bits divides the integer by 2^n.
   - ○ True
   - ○ False

2) Shifting the bits of a *signed* integer right by n bits divides the integer by 2^n if the shifter extends the sign bit instead of shifting in 0s.
   - ○ True
   - ○ False

Pitfall: Floating-point addition is not associative.

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number. For example, let's see if *c + (a + b) = (c + a) + b*. Assume c = $-1.5_{ten} \times 10^{38}$, a = $1.5_{ten} \times 10^{38}$, and b = 1.0, and that these are all single precision numbers.

$$\begin{aligned} c + (a + b) &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38} + 1.0) \\ &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38}) \\ &= 0.0 \end{aligned}$$

$$\begin{aligned} (c + a) + b &= (-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}) + 1.0 \\ &= (0.0_{ten}) + 1.0 \\ &= 1.0 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{ten} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{ten} \times 10^{38} + 1.0$ is still $1.5_{ten} \times 10^{38}$. That is why the sum of *c*, *a*, and *b* is 0.0 or 1.0, depending on the order of the floating-point additions, so c + (a + b) ≠ (c + a) + b. Therefore, floating-point addition is *not* associative.

Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, "Do the two versions get the same answer?" If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you were to add a million numbers together, you would get the same results whether you used one processor or 1000 processors. This assumption holds for two's complement integers, since integer addition is associative. Alas, since floating-point addition is not associative, the assumption does not hold.

A more vexing version of this fallacy occurs on a parallel computer where the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer. As the varying number of processors from each run would cause the floating-point sums to be calculated in different orders, getting slightly different answers each time despite running identical code with identical input may flummox unaware parallel programmers.

Given this quandary, programmers who write parallel code with floating-point numbers need to verify whether the results are credible, even if they don't give the exact same answer as the sequential code. The field that deals with such issues is called numerical analysis, which is the subject of textbooks in its own right. Such concerns are one reason for the popularity of numerical libraries such as LAPACK and SCALAPAK, which have been validated in both their sequential and parallel forms.

PARTICIPATION ACTIVITY 3.9.2: Floating-point fallacies and pitfalls.

1) Floating-point addition is _____.
   - O associative
   - O not associative

2) Rewriting programs that contain _____ arithmetic operations to execute in parallel may affect the result.
   - O two's complement integer
   - O floating-point

3) A parallel program containing floating-point arithmetic operations executing on 10 processors may produce a different result than the same program executing on 1,000 processors.
   - O True
   - O False

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see the figure below). The following is the inside story behind the headlines.

Figure 3.9.1: A sampling of newspaper and magazine articles from November 1994, including the New York Times, San Jose Mercury News, San Francisco Chronicle, and Infoworld (COD Figure 3.26).

The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing -2, -1, 0, +1, or +2. The guess is multiplied by the divisor and subtracted from the remainder to generate a new remainder. Like nonrestoring division, if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently, there were five elements of the table from the 80486 that Intel engineers thought could never be accessed, and they optimized the PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

A math professor at Lynchburg College in Virginia, Thomas Nicely, discovered the bug in September 1994. After calling Intel technical support and getting no official reaction, he posted his discovery on the Internet. This post led to a story in a trade magazine, which in turn caused Intel to issue a press release. It called the bug a glitch that would affect only theoretical mathematicians, with the average spreadsheet user seeing an error every 27,000 years. IBM Research soon counterclaimed that the average spreadsheet user would see an error every 24 days. Intel soon threw in the towel by making the following announcement on December 21:

> "We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer.

Analysts estimate that this recall cost Intel $500 million, and Intel engineers did not get a Christmas bonus that year. This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

PARTICIPATION ACTIVITY 3.9.3: Implementation details.

1) An inaccuracy in floating-point division cost Intel an estimated $500 million.
   - ○ True
   - ○ False

Provide feedback on this section