Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, 1992

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, the rationale for the 80-bit stack architecture for floating point in the IA-32, and an update on the next round of the standard.

At first it may be hard to imagine a subject of less excitement than the correctness of computer arithmetic or its accuracy, and harder still to understand why a subject so old and mathematical should be so contentious. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical reuse of registers during serial multiplication and division, still command respect today. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest stored-program computer had been built:

> *… a project under von Neumann was to be set up at the Institute of Advanced Studies in Princeton.… Goldstine explained to me the principal features of the design, including the device whereby the digits of the multiplier were put into the tail of the accumulator and shifted out as the least significant part of the product was shifted in. I expressed some admiration at the way registers and shifting circuits were arranged … and Goldstine remarked that things of that nature came very easily to von Neumann.*

There is no controversy here; it can hardly arise in the context of exact integer arithmetic, so long as there is general agreement on what integer the correct result should be. However, as soon as approximate arithmetic enters the picture, so does controversy, as if one person's "negligible" must be another's "everything."

The First Dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann was aware of Konrad Zuse's proposal for a computer in Germany in 1939 that was never built, probably because the floating point made it appear too complicated to finish before the Germans expected World War II to end. Hence, von Neumann refused to include it in the computer he built at Princeton. In an influential report coauthored in 1946 with H. H. Goldstine and A. W. Burks, he gave the arguments for and against floating point. In favor:

> *… to retain in a sum or product as many significant digits as possible and … to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplication constants which serve to keep numbers within the limits of the machine.*

Floating point was excluded for several reasons:

> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

The argument seems to be that most bits devoted to exponent fields would be bits wasted. Experience has proven otherwise.

One software approach to accommodate reals without floating-point hardware was called *floating vectors*; the idea was to compute at runtime one-scale factor for a whole array of numbers, choosing the scale factor so that the array's biggest number would barely fill its field. By 1951, James H. Wilkinson had used this scheme extensively for matrix computations. The problem proved to be that a program might encounter a very large value, and hence the scale factor must accommodate these rare sizeable numbers. The common numbers would thus have many leading 0s, since all numbers had to use a single-scale factor. Accuracy was sacrificed, because the least significant bits had to be lost on the right to accommodate leading 0s. This wastage became obvious to practitioners on early computers that displayed all their memory bits as dots on cathode ray tubes (like TV screens) because the loss of precision was visible. Where floating point deserved to be used, no practical alternative existed.

Thus, true floating-point hardware became popular because it was useful. By 1957, floating-point hardware was almost ubiquitous. A decimal floating-point unit was available for the IBM 650, and soon the IBM 704, 709, 7090, 7094 … series would offer binary floating-point hardware for double, as well as single, precision.

As a result, everybody had floating point, but every implementation was different.

Diversity versus Portability

Since roundoff introduces some error into almost all floating-point operations, to complain about another bit of error seems picayune. So for 20 years, nobody complained much that those operations behaved a little differently on different computers. If software required clever tricks to circumvent those idiosyncrasies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a long time, matrix computations mystified most people who had no notion of error analysis; perhaps this continues to be true. That

may be why people are still surprised that numerically stable matrix computations depend upon the quality of arithmetic in so few places, far fewer than are generally supposed. Books by Wilkinson and widely used software packages like Linpack and Eispack sustained a false impression, widespread in the early 1970s, that a modicum of skill sufficed to produce *portable* numerical software.

"Portable" here means that the software is distributed as source code in some standard language to be compiled and executed on practically any commercially significant computer, and that it will then perform its task as well as any other program performs that task on that computer. Insofar as numerical software has often been thought to consist entirely of computer-independent mathematical formulas, its portability has commonly been taken for granted; the mistake in that presumption will become clear shortly.

Packages like Linpack and Eispack cost so much to develop—over a hundred dollars per line of Fortran delivered—that they could not have been developed without U.S. government subsidy; their portability was a precondition for that subsidy. But nobody thought to distinguish how various components contributed to their cost. One component was algorithmic—devise an algorithm that deserves to work on at least one computer despite its roundoff and over-/underflow limitations. Another component was the software engineering effort required to achieve and confirm portability to the diverse computers commercially significant at the time; this component grew more onerous as ever more diverse floating-point arithmetics blossomed in the 1970s. And yet scarcely anybody realized how much that diversity inflated the cost of such software packages.

A Backward Step

Early evidence that somewhat different arithmetics could engender grossly different software development costs was presented in 1964. It happened at a meeting of SHARE, the IBM mainframe users' group, at which IBM announced System/360, the successor to the 7094 series. One of the speakers described the tricks he had been forced to devise to achieve a level of quality for the S/360 library that was not quite so high as he had previously achieved for the 7094.

Von Neumann could have foretold part of the trouble, had he still been alive. In 1948, he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it. It did predict correctly, however, that computations with larger arrays of data would probably fall prey to roundoff more often. IBM S/360s had bigger memories than 7094s, so data arrays could grow larger, and they did. To make matters worse, the S/360s had narrower single-precision words (32 bits versus 36) and used a cruder arithmetic (hexadecimal or base 16 versus binary or base 2) with consequently poorer worst-case precision (21 significant bits versus 27) than the old 7094s. Consequently,

software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on S/360s. The quickest way to recover adequate accuracy was to replace old codes' single precision declarations with double precision before recompilation for the S/360. This practice exercised S/360 double precision far more than had been expected.

The early S/360's worst troubles were caused by lack of a guard digit in double precision. This lack showed up in multiplication as a failure of identities like $1.0 * x = x$ because multiplying x by 1.0 dropped x's last hexadecimal digit (4 bits). Similarly, if x and y were very close but had different exponents, subtraction dropped off the last digit of the smaller operand before computing $x - y$. This final aberration in double precision undermined a precious theorem that single precision then (and now) honored: If $1/2 \leq x/y \leq 2$, then no rounding error can occur when $x - y$ is computed; it must be computed exactly.

Innumerable computations had benefited from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof. We had been taking all this stuff for granted.

The identities and theorems about exact relationships that persisted, despite roundoff, with reasonable implementations of approximate arithmetic were not appreciated until they were lost. Previously, it had been thought that the things to matter were precision (how many significant digits were carried) and range (the spread between over-/underflow thresholds). Since the S/360's double precision had more precision and wider range than the 7094's, software was expected to continue to work at least as well as before. But it didn't.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on S/360s. A small subcommittee of SHARE proposed improvements to the S/360 floating point. This committee was surprised and grateful to get a fair part of what they asked for from IBM, including all-important guard digits. By 1968, these had been retrofitted to S/360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility (a lesson learned by Intel 30 years later; see Figure 3.25). IBM employees who can remember the incident still shudder.

The People Who Built the Bombs

Seymour Cray was associated for decades with the CDC and Cray computers that were, when he built them, the world's biggest and fastest. He always understood what his customers wanted most: *speed*. And he gave it to them even if, in so doing, he also gave them arithmetics more "interesting" than anyone else's. Among his customers have been the great government laboratories like those at Livermore and Los Alamos, where nuclear weapons were designed. The challenges of "interesting" arithmetics were pretty tame to people who had to overcome Mother Nature's challenges.

Perhaps all of us could learn to live with arithmetic idiosyncrasy if only one computer's idiosyncrasies had to be endured. Instead, when accumulating different computers' different anomalies, software dies the Death of a Thousand Cuts. Here is an example from Cray's computers:

```
if (x == 0.0)    y = 17.0 else y = z/x
```

Could this statement be stopped by a divide-by-zero error? On a CDC 6600 it could. The reason was a conflict between the 6600's adder, where `x` was compared with 0.0, and the multiplier and divider. The adder's comparison examined `x`'s leading 13 bits, which sufficed to distinguish zero from normal nonzero floating-point numbers `x`. The multiplier and divider examined only 12 leading bits. Consequently, tiny numbers existed that were nonzero to the adder but zero to the multiplier and divider! To avoid disasters with these tiny numbers, programmers learned to replace statements like the one above with

```
if (1.0 * x == 0.0)    y = 17.0 else y = z/x
```

But this statement is unsafe to use in would-be portable software because it malfunctions obscurely on other computers designed by Cray, the ones marketed by Cray Research, Inc. If `x` was so huge that 2.0 * `x` would overflow, then 1.0 * `x` might overflow too! Overflow happens because Cray computers check the product's exponent before the product's exponent has been normalized, just to save the delay of a single AND gate.

Rounding error anomalies that are far worse than the over-/underflow anomaly just discussed also affect Cray computers. The worst error came from the lack of a guard digit in add/subtract, an affliction of IBM S/360s. Further bad luck for software is occasioned by the way Cray economized his multiplier; about one-third of the bits that normal multiplier arrays generate have been left out of his multipliers, because they would contribute less than a unit to the last place of the final Cray-rounded product. Consequently, a Cray multiplier errs by almost a bit more than might have been expected. This error is compounded when division takes three multiplications to improve an approximate reciprocal of the divisor and then multiply the numerator by it. Square root compounds a few more multiplication errors.

The fast way drove out the slow, even though the fast was occasionally slightly wrong.

Making the World Safe for Floating Point, or Vice Versa

William Kahan was an undergraduate at the University of Toronto in 1953 when he learned to program its Ferranti-Manchester Mark-I computer. Because he entered the field early, Kahan became acquainted with a wide range of devices and a large proportion of the personalities active in computing; the numbers of both were small at that time. He has performed computations on slide rules, desktop

mechanical calculators, tabletop analog differential analyzers, and so on; he has used all but the earliest electronic computers and calculators mentioned in this book.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two years of postdoctoral study in England, where he became acquainted with Wilkinson. In 1960, he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free rein to tinker with its operating system, Fortran compiler, and runtime library. (He denies that he ever came near the 7090 hardware with a soldering iron but admits asking to do so.) One story from that time illuminates how misconceptions and numerical anomalies in computer systems can incur awesome hidden costs.

A graduate student in aeronautical engineering used the 7090 to simulate the wings he was designing for short takeoffs and landings. He knew such a wing would be difficult to control if its characteristics included an abrupt onset of stall, but he thought he could avoid that. His simulations were telling him otherwise. Just to be sure that roundoff was not interfering, he had repeated many of his calculations in double precision and gotten results much like those in single; his wings had stalled abruptly in both precisions. Disheartened, the student gave up.

Meanwhile Kahan replaced IBM's logarithm program (ALOG) with one of his own, which he hoped would provide better accuracy. While testing it, Kahan reran programs using the new version of ALOG. The student's results changed significantly; Kahan approached him to find out what had happened.

The student was puzzled. Much as the student preferred the results produced with the new ALOG—they predicted a gradual stall—he knew they must be wrong because they disagreed with his double precision results. The discrepancy between single and double precision results disappeared a few days later when a new release of IBM's double-precision arithmetic software for the 7090 arrived. (The 7090 had no double-precision hardware.) He went on to write a thesis about it and to build the wings; they performed as predicted. But that is not the end of the story.

In 1963, the 7090 was replaced by a faster 7094 with double precision floating-point hardware but with otherwise practically the same instruction set as the 7090. Only in double precision and only when using the new hardware did the wing stall abruptly again. A lot of time was spent to find out why. The 7094 hardware turned out, like the superseded 7090 software and the subsequent early S/360s, to lack a guard bit in double precision. Like so many programmers on those computers and on Cray's, the student discovered a trick to compensate for the lack of a guard digit; he wrote the expression `(0.5 - x) + 0.5` in place of `1.0 - x`. Nowadays we would blush if we had to explain why such a trick might be necessary, but it solved the student's problem.

Meanwhile the lure of California was working on Kahan and his family; they came to Berkeley and he to the University of California. An opportunity presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and his work

dramatically improved the accuracy of HP calculators, but that is another story. Fruitful collaboration with congenial coworkers, however, fortified him for the next and crucial opportunity.

It came in 1976, when John F. Palmer at Intel was empowered to specify the "best possible" floating-point arithmetic for all of Intel's product line, as Moore's Law made it now possible to create a whole floating-point unit on a single chip. The floating-point standard was originally started for the iAPX-432, but when it was late, Intel started the 8086 as a short-term emergency stand-in until the iAPX-432 was ready. The iAPX-432 never became popular, so the emergency stand-in became the standard-bearer for Intel. The 8087 floating-point coprocessor for the 8086 was contemplated. (A *coprocessor* is simply an additional chip that accelerates a portion of the work of a processor; in this case, it accelerated floating-point computation.)

Palmer had obtained his Ph.D. at Stanford a few years before and knew whom to call for counsel of perfection—Kahan. They put together a design that obviously would have been impossible only a few years earlier and looked not quite possible at the time. But a new Israeli team of Intel employees led by Rafi Navé felt challenged to prove their prowess to Americans and leaped at an opportunity to put something impossible on a chip—the 8087.

By now, floating-point arithmetics that had been merely diverse among mainframes had become chaotic among microprocessors, one of which might be host to a dozen varieties of arithmetic in ROM firmware or software. Robert G. Stewart, an engineer prominent in IEEE activities, got fed up with this anarchy and proposed that the IEEE draft a decent floating-point standard. Simultaneously, word leaked out in Silicon Valley that Intel was going to put on one chip some awesome floating point well beyond anything its competitors had in mind. The competition had to find a way to slow Intel down, so they formed a committee to do what Stewart requested.

Meetings of this committee began in late 1977 with a plethora of competing drafts from innumerable sources and dragged on into 1985, when IEEE Standard 754 for Binary Floating Point was made official. The winning draft was very close to one submitted by Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based on the Intel design, with Intel's permission, of course, as simplified by Coonen. Their harmonious combination of features, almost none of them new, had at the outset attracted more support within the committee and from outside experts like Wilkinson than any other draft, but they had to win nearly unanimous support within the committee to win official IEEE endorsement, and that took time.

The First IEEE 754 Chips

In 1980, Intel became tired of waiting and released the 8087 for use in the IBM PC. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per

instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." (Kahan's [1990] history of the stack architecture selection for the 8087 is entertaining reading.)

Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was that the restriction of the top of stack as one operand was not so bad since it only required the execution of an FXCH instruction (which swapped registers) to get the same result as a two-operand instruction, and FXCH was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence, he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data.

The Intel 8087 was implemented in Israel, and 7500 miles and 10 time zones made communication from California difficult. According to Palmer and Morse (*The 8087 Primer*, J. Wiley, New York, 1984, p. 93):

> *Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid operation is indeed due to a stack overflow.… Also lacking is the ability to restart the instruction that caused the stack overflow …*

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> *Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/under-flow.…*
>
> *I still regret that the 8087's stack implementation was not quite so neat as my original intention.… If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.*

In 1982, Motorola announced its 68881, which found a place in Sun 3s and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Another Berkeley graduate student, George S. Taylor, had soon designed a high-speed implementation of the proposed standard for an early superminicomputer (ELXSI 6400). The standard was becoming de facto before its final draft's ink was dry.

An early rush of adoptions gave the computing industry the false impression that IEEE 754, like so many other standards, could be implemented easily by following a standard recipe. Not true. Only the enthusiasm and ingenuity of its early implementors made it look easy.

In fact, to implement IEEE 754 correctly demands extraordinarily diligent attention to detail; to make it run fast demands extraordinarily competent ingenuity of design. Had the industry's engineering managers realized this, they might not have been so quick to affirm that, as a matter of policy, "We conform to all applicable standards."

IEEE 754 Today

Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied.

The successful parts of IEEE 754 are that it is a widely implemented standard with a common floating-point format, that it requires minimum accuracy to one-half ulp in the least significant bit, and that operations must be commutative.

The IEEE 754/854 has been implemented to a considerable degree of fidelity in at least part of the product line of every North American computer manufacturer. The only significant exceptions were the DEC VAX, IBM S/370 descendants, and Cray Research vector supercomputers, and all three have been replaced by compliant computers.

IEEE rules ask that a standard be revisited periodically for updating. A committee started in 2000, and drafts of the revised standards were circulated for voting, and these were approved in 2008. The revised standard, IEEE Std 754-2008 [2008], includes several new types: 16-bit floating point, called *half precision*; 128-bit floating point, called *quad precision*; and three decimal types, matching the length of the 32-bit, 64-bit, and 128-bit binary formats. IEEE Std 754-2019 made minor changes to the standard. The plan is to revisit it every 10 years. In 1989, the Association for Computing Machinery, acknowledging the benefits conferred upon the computing industry by IEEE 754, honored Kahan with the Turing Award. On accepting it, he thanked his many associates for their diligent support, and his adversaries for their blunders. So . . . not all errors are bad.

Further Reading

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 2002] are good starting points; they abound with pointers to further reading. Several of the stories told in this section come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the most recent IEEE-sponsored Symposium on Computer Arithmetic, held every two years; the 27th was held in 2020.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [2002]. "Computer arithmetic". Appendix A of *Computer Architecture: A Quantitative Approach*, third edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4–3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* 23(1), 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana), Vol. 2, North-Holland Publishing, Amsterdam, 1214–1239.

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California, Berkeley.

What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available at *http://www.cims.nyu.edu/~dbindel/class/cs279/87stack.pdf*.

A collection of memos related to floating point, including "Beastly numbers" (another less famous Pentium bug), "Notes on the IEEE floating point arithmetic" (including comments on how some features are atrophying), and "The baleful effects of computing benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, …).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.