Keynote Speech, 10th IEEE Symosium on Computer Arithmetic, Grenoble, France, June 25, 1991

Arithmetic Standards: The Long Road

W. J. Cody

This is the tenth IEEE Symposium on Computer Arithmetic. A number of people here tonight were among the 25 attendees of the first symposium, a 1-day workshop organized and hosted by Robert Shively in Minneapolis in June, 1969. Some of the attendees have been involved with all 10 of the symposia, so there is a strong thread of continuity running through these meetings. A lot has changed since 1969. One of the most significant changes for us has been the emergence of official standards for floating-point arithmetic.

When David Matula and Peter Kornerup asked me to speak tonight, they suggested that a discussion of the history and significance of these standards would be appropriate. My views in these matters are undoubtedly biased; I am a numerical analyst concerned with mathematical software, not an electrical engineer. I know nothing about the difficulty of designing and implementing arithmetic engines, but I can attest to the misery that poor arithmetic engines cause. My professional concern has always been with the provision of high-quality numerical programs for scientific computation. Ideally, these programs will be reliable, robust, and portable. Reliability means that the program returns an accurate result anytime that result can be computed from the data. Robustness means that the program is resilient under intentional or unintentional misuse. And portability means that the program can be moved from one machine to another with no degradation in performance. Reliability, robustness, and portability are now standard attributes of good numerical software.

We now routinely expect our software to have all of these attributes. They are often difficult to achieve; some might say they are not worth the effort. Why worry about whether a machine rounds or truncates, for example? How could that possibly affect in any significant way the accuracy of the result, especially with the reasonable wordlength of modern computers? For the answer to that, we turn to the Wall Street Journal of November 8, 1983.

> Things were looking grim at the Vancouver Stock Exchange.
>
> Brokers and investors on the free-wheeling penny-stock market wondered where the North American bull market had gone, as the exchange index kept dropping. The index, which had been established in January 1982 at a level of 1,000, recently has been hitting lows in the 520 range.
>
> Now the exchange says it goofed in calculating the index. Donald Hudson, the exchange's embarrassed president, estimates the true value (of the index) at between 900 and 1000, or maybe even over 1000.

> The index is based on the selling price of all 1,400 or so stocks listed on the exchange. Every time a stock price changes, which happens 2,800 times on an average day, a computer recalculates the index to three decimal places.
>
> The mistake was made in calculating the last decimal place. (The computer simply truncated instead of rounding to three decimal places.)

So here is an example where truncating instead of rounding completely destroyed the validity of a simple computation. Yes, the destruction occurred over a period of time and over many computations, but consider how many computations a modern supercomputer carries out each second. The lesson is there for all of us.

We know of no major disasters that have occurred because of poor arithmetic, but let me also read you an excerpt from an article in Aviation Week and Space Technology for March 31, 1991. The article concerns flight testing of a new, high-tech research aircraft, the X-31, jointly designed by the United States and Germany.

> The Honeywell digital flight control system is derived from one used on the Lockheed C-130 high-technology testbed aircraft. It has three channels with a fourth computer used as a tie breaker. The control system went into a reversionary mode four times in the first nine flights, usually due to disagreement between the two air data sources. The air data computer dates back to the mid-1960s and had a divide-by-zero that occurred briefly. This was not a problem in its previous application, but the X-31 flight control system would not tolerate it. This was fixed with software in January, and the problem has not reoccurred in seven subsequent flights.

Exactly what is a briefly occurring divide-by-zero, and why was it ignored for thirty years? Recent arithmetic systems would call attention to a divide-by-zero either with an interrupt or with a signal and a default infinite result. Either way, the problem would be made known. Can we assume that those using this Honeywell system were also aware of the problem and simply chose to ignore it? That is a frightening possibility. How many such flawed systems are in use today? Is it only a matter of time before we are talking about a major disaster caused by bad computer hardware or software? As computer engineers and programmers, quality must be our first priority.

Some of us working on mathematical software became convinced of that in the early 1960s when subroutine libraries first became popular. Despite being based on solid mathematics, our programs often failed in mysterious ways. Worse yet, routines we thought were working properly suddenly failed on new machines. W. Kahan, then at Toronto, H. Kuki at the University of Chicago, G. Forsythe at Stanford, a small group at Bell Labs, and a group I headed at Argonne all worked at producing decent libraries for our own installations. As we gained experience and began to understand what we were doing, we sought to write programs with the three attributes mentioned earlier: reliability, robustness, and transportability.

This was not easy, because we discovered that even the most fundamental mathematical relations failed to hold on some computers (as they still fail on some of our biggest and fastest computers today). Given machine numbers X and Y, one reasonably expects that

```
     X * 1.0 = X,
     X + X = 2.0 * X,
     X * Y = Y * X,
```

and

```
     X < (X + Y) / 2.0 < Y when X < Y.
```

Yet each of these relations could fail on at least one machine, some on many different machines, and the nature of the failures differed from one arithmetic engine to another. Many arithmetic design parameters contributed to these difficulties. Some machines lacked guard digits for one or more of the fundamental arithmetic operations, some used hexadecimal arithmetic, some truncated, some used bizarre rounding schemes, some had over-length arithmetic registers, and some compilers were flaky. Come to think of it, not much has changed in thirty years, has it?

To illustrate the problems we faced, lets go back 20 or 25 years and consider a simple computational task: write a portable Fortran code to determine the underflow threshold on any machine. This is one of the steps in the MACHAR program for determining machine parameters. First of all, what do we mean by underflow and how do we test for it? Intuitively, underflow occurs when the exponent of a computed result becomes too small for the representation scheme. Such results are often represented as 0.0 (the "flush-to-zero" approach). So the obvious strategy is to construct smaller and smaller numbers until underflow is detected. The following code might be a first try. We assume that the variable B has been initialized to the radix for the machine.

```
        X = 1.0 / B
        Y = X
        I = 0
     10 Z = Y
           Y = Y * X
           I = I + 1
           IF (Y .NE. 0.0) GO TO 10
```

If all goes well, Z is the smallest power of the radix that does not underflow. This scheme worked well on mainframe IBM machines, for example, but went into an infinite loop on another popular machine where underflow was replaced by the smallest positive machine number. That was easily fixed, of course, by modifying the test to read

```
          IF ((Y .NE. 0.0) .AND. (Y .NE. Z)) GO TO 10
```

But on other machines, the underflowed exponent wrapped around to produce a large number, sometimes with a sign change. So the test had to become

```
          IF ((Y .NE. 0.0) .AND. (ABS(Y) .LT. Z)) GO TO 10
```

But on the CDC 6600, the smallest non-vanishing power of the radix was ambiguous. On those machines there existed small positive Y such that

$$Y + Y = 0.0$$

even though other operations with Y were okay, i.e.,

$$Y * 1.0 = Y$$

to within rounding error. Clearly, the test for non-vanishing Y had to be further altered to check whether Y + Y vanished. CDC was justifiably embarrassed by that behavior, so they fixed it on the CDC 7600. On that machine, $Y > 0.0$ implied that $Y + Y > 0.0$. Unfortunately, for some small $Y > 0$, $Y * 1.0$ now vanished. This required yet another modification to the test. With all of this the algorithm still did not detect a useful underflow threshold in double precision on the CDC 7600 because the standard CDC compiler on that machine treated double-precision numbers as an ordered pair of single-precision numbers. You guessed it, the least significant member of the pair could underflow to zero independent of the most significant member, so a computation could lose half of its significance without warning of any kind. To detect this strange underflow, the test in our algorithm had to be modified once more, and by now what started out as a simple computation had turned into a puzzling collection of tests most of which were superfluous on any particular machine, but each of which was necessary on some machine.

Add together all of these schemes for underflow, throw in the various weird rounding schemes, the lack of guard digits, hexadecimal arithmetic, over-length arithmetic registers on some machines, flaky compilers and operating systems, and stir the whole mess up. This was the hardware/software swamp that those of us working on numerical software faced in the late 1960s. We found ourselves spending more and more of our time trying to discover and program around anomalies in the popular machines. With one notable exception, manufacturers paid little attention to our loud and bitter complaints about their arithmetic engines. Hirondo Kuki did manage to convince IBM to make a field change on the early 360 machines that lacked a guard digit in double precision. But that is the only success we had, and IBM people confided to me privately that they would never make such a hardware field change again. We felt that our needs were being ignored.

The IBM 360 family also introduced hexadecimal arithmetic. Unfortunately, Kuki could do nothing about that and we have struggled with the resulting "wobbling precision" ever since. The mystique of the IBM name caused IBM sales to soar despite their poor arithmetic design, and their success prompted other manufacturers to go the hexadecimal route. One nameless manufacturer in particular designed a hexadecimal machine that was extremely flawed. The demonstration team that visited Argonne bragged about a fast multiply operation that produced double-length results from single-precision operands. This operation became the basis for a fast double-precision multiply. Their scheme divided the double-precision operands into pairs of single-precision operands, and then summed three, not four, single-precision multiplies, each producing a double length result. A back-of-the-envelope computation showed that rather than producing a full double-precision product, they could lose almost a fourth of their significance in some cases. The problem was wobbling precision coupled with that ignored fourth product. This was pointed out to them before they left Argonne. I still have the letter the team leader wrote a few days later in which he said "Yes, our engineers see now that they have blundered, but we would not let that stop us from selling you a machine." Perhaps that attitude was unusual, but the truth is probably that the attitude was common; only its expression in writing was unusual.

Clearly, quality of numerical software depends on the quality of the underlying algorithms and the skill of the programmer. But it also depends on the quality of the tools the programmer uses: the arithmetic engines and the algebraic compilers. So our quest for quality in mathematical software necessarily involves us with those providing the tools. And that is where we become professionally concerned with standards. Not necessarily formal written "standards", but informal minimal standards of performance.

Many of our frustrations with the hardware were discussed at the first Mathematical Software meeting, hosted by John Rice at Purdue in 1970. One of the complaints voiced at that meeting was that our message was not getting through to the designers of arithmetic systems. Shortly after that meeting, Kuki and I attempted to substantiate our dissatisfaction with existing arithmetic designs and to justify our preferences by statistically studying the effect of various design parameters on the accuracy of common computations. Unfortunately, Kuki passed away shortly after the manuscript was accepted for publication.

A friend suggested that I summarize the work with Kuki at an arithmetic conference to be held in Maryland in May 1972. That, of course, was ARITH2. It was a follow up to the 1-day workshop on computer arithmetic organized and hosted by Robert Shively in Minneapolis in June 1969. Among the 25 attendees at the Minneapolis meeting were at least two numerical analysts, Dave Matula and Richard Brent, so our participation followed a precedent. At any rate, the paper was accepted for ARITH2, and I attended the meeting with great hopes of finally convincing designers of the seriousness of our concerns. Alas, all of the numerical talks, including some by Brent and Matula, were scheduled for the last morning of the meeting. The audience was small. Professors Avizienis, Ercegovac, and Svoboda were in attendance, but many of the other people we had come to impress had left. Numerical analysts found themselves largely talking to themselves, and left the meeting completely frustrated. Somehow, miraculously, some of the papers were printed in a special issue of the IEEE Transactions on Computers, and we did reach designers after all.

I don't recall hearing the last paper scheduled for ARITH2, but it was printed in the informal proceedings. It was titled "Standards for Computer Arithmetic". The author, a Professor Marcovitz from Florida State University, pointed out all of the obvious disadvantages to standardizing arithmetic, and urged rejection of a proposed standard recently submitted to ANSI. Marcovitz' paper is the only reference I ever saw to that early standardization effort, so the standard must have been still-born.

From 1972 on, these Computer Arithmetic Conferences have been an important forum for designers and users of arithmetic systems to share opinions. The next few meetings included additional discussion of machine parameters and of the implications of machine design for numerical accuracy in addition to the papers on circuit design and p-adic arithmetic. Still, the economic forces driving the mainframe marketplace dictated that each company pursue its own design to the exclusion of all others, and little progress was made towards improved arithmetic design, let alone minimal standards of performance.

The next mention of an arithmetic standard in the proceedings of these meetings is a paper by G. Walker from Motorola at ARITH5, held in Ann Arbor in 1981. The paper discussed the extension of the M68000 to incorporate the proposed IEEE standard for floating-point arithmetic. There was also a minisymposium discussing the proposed IEEE standard at that meeting. What happened between ARITH4 in Santa Monica in 1978 and the Ann Arbor meeting?

The main catalyst was the appearance of a new kid on the block, one who didn't know much about floating-point arithmetic except that he probably needed it. That was the microchip industry.

As best I can reconstruct the important events leading up to the IEEE effort, John Palmer of Intel had taken a class from W. Kahan, now at UC Berkeley, and had absorbed Kahan's preaching about what a floating-point unit should provide. Palmer was also aware of Kahan's influence in the design of HP calculators, and convinced the people at Intel to bring Kahan in as a consultant. Kahan not only saw the opportunity to finally design an arithmetic unit that would meet his expectations, but the opportunity to influence an industry as a whole. He confided to me at a meeting in Pasadena in 1976 or 1977 that while he could not influence the design of mainframes or of minis directly, he could insure that micros had excellent arithmetic. He believed that the market for micros would grow, and that their superior arithmetic would eventually pressure the manufacturers of larger machines to provide better arithmetic engines. In modern terminology, he intended to outflank them.

Thus was born IEEE project P754. Main players in the effort were Kahan and Palmer, of course, together with Harold Stone from the University of Massachusetts, Robert Fraley and Stephen Walther from HP, Mary Payne, Dileep Bhandakar, and William Strecker from DEC, David Stevenson from Zilog, Jerry Coonen from UC Berkeley, David Hough and Jim Thomas from Apple, Fred Ris from IBM, Richard Karpinski from UCSF, Richard Delp from Four Phase Systems, Tom Pittman from Itty Bitty Computers, Bob Stewart from Stewart Research, and others whose names elude me at the moment. I joined the effort after the first year or so. Kahan won the Turing Award last year for his role in the effort. The important thing is that there was wide-spread participation in the effort, and the early deliberations involved several fundamentally different proposals. The outcome of that effort and the follow-on P854 project is familiar to all of you. Today we have floating-point arithmetic standards that are implemented on all sorts of machines from workstations to massively parallel machines, like the Connection Machine and the Intel Delta, that verge on being supercomputers. Kahan has recently told me that Cray is thinking seriously of adding IEEE arithmetic to their machine line, something that most of us would applaud. The lack of a guard digit on Crays is still a source of anomalies, as Kahan has pointed out over and over again for the last decade or so.

Has the binary standard been a bad thing? I think not. From the hardware viewpoint, it has minimized the design effort for startup companies and has minimized if not eliminated the appearance of new anomalies on new machine designs, but it has not stifled creativity of designs. Until recently, at least, the designers of Crays, mainframe IBMs, and other such machines have still been doing their thing. Nor is it the only good design around. The DEC VAX offered a marvelous arithmetic engine about the time the standards effort began. The VAX is still a great machine for numerical computations, especially since they introduced the G-format double-precision and the H-format extended-precision arithmetic.

If there is a drawback to the standards, it is the inclination of those preparing government procurement specs to require IEEE arithmetic simply because the standard is so well known. Just as I objected to buying IBM machines on the basis of the name, I believe that requiring IEEE machines without a strong technical reason is an error. For one thing, it rules out consideration of worthy machines like the VAX in cases where they might be ideal for the intended purpose. Further, and more important, despite the appearance of the standards, few manufacturers offer full implementations. Many take shortcuts by not offering features like graceful underflow, infinities and NaNs that the standard requires. Even if the hardware offers a complete implementation, the operating system and compilers often do not offer support for features like rounding control and user-supplied trap handlers. The danger in specifying full conformance to the IEEE standard is that the choice is suddenly limited to very few machines unless one waves his hands and says, "you know what we mean by conformance". Such vague statements are not only contrary to the very idea of a standard, they also provide a good living for lawyers contesting procurement contracts.

The formal approval of a standard hasn't solved all of our problems -- much remains to be done. From the viewpoint of the producer of numerical software, the standard has made the job easier, but it has also introduced new barriers. The IEEE standards do not address the problems of language interfaces -- how the hardware features are to be made available to programmers. Drafters of the Ada and C standards, at least, are currently wrestling with how to exploit the features of IEEE arithmetic without crippling applications running on other architectures. How does one specify rounding control, or the handling of signed zero for machines that lack such features? For that matter, how does one exploit signed zero on machines that have it? Think a bit about what a signed zero means for a complex argument on a branch cut; a returned function value can jump from one sheet of a Riemann surface to another with a change in the sign of zero. How does Ada, which requires an overflow exception, coexist with the IEEE infinity? These are deep questions that are not easily answered. Fortunately, both the Ada and C efforts involve numerical analysts who understand the hardware and the preparation of numerical software.

In addition to the vital problems of software support, there are problems associated with areas where the binary standard is permissive. What is the proper way to output an infinity or a NaN, for example? How does one indicate such a thing in an input stream? The binary standard says nothing about this, although the 854 standard does. What is the proper ordering of the halves of a double-precision number in storage (the old big-endian, little-endian problem)? The binary standard specifies what a double-precision quantity looks like in a register, but not in storage. The Sun is a big-endian machine, while the Sequent Symmetry is a little-endian, so these questions are important. What determines when an intermediate result should be retained in the extra-precision registers found in many implementations of the standard, and when it should be forced to storage format? This is a real bone of contention between those designing optimizing compilers and those trying to do delicate computations.

The committees that drafted the floating-point standards have been criticized lately for not dealing with these language issues themselves, for not specifying the way a language should control rounding options, for example. I personally believe the criticism to be unjustified. We had enough trouble drafting arithmetic standards for what we thought to be a small market. It would have been presumptuous of us to try to dictate to the language standards people as well. Indeed, if we had tried to do so, our drafts would probably not have passed the ANSI and ISO committees. In hindsight, because of the enormous success and popularity of the arithmetic standards, we might be better off today if language bindings had been included.

One final thought on standards. They are there as guidance. They should never be taken as the final, once and forever way for everyone to do something. Several years ago, someone approached me with his new scheme for computer arithmetic. He informed me that the IEEE standards effort was a waste of time because his scheme was far superior. In effect, he said, "you are either with us or against us." That attitude delayed acceptance of his scheme as a useful tool for years. Beware of such zealots, IEEE supporters or not. There is no best way to do anything.

In summary, I personally believe that the IEEE floating-point standards have been a significant step forward. They are not perfect, but they have greatly simplified work on numerical software. At the same time, the job is not done; much remains to be done to fully exploit what they offer. More importantly, other valid approaches to computer arithmetic exist. Lets hope that the convenience of a pre-designed system does not stifle creative juices. Lets also hope that the standards have established a lower bound on acceptable quality of arithmetic designs. Let us never again have to wrestle with an X that vanishes when added to itself.