This is intended for a forthcoming text on computer architecture by David A. Patterson and John L. Hennessy aimed at college sophomores or juniors. It would be an appendix to a chapter about computer arithmetic.
NOT FOR FURTHER DISTRIBUTION! Patterson & Hennessy may change this.

4.11 Floating-Point: 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

At first it may be hard to imagine a subject much less interesting than computer arithmetic carried out correctly, and harder again to understand why a subject so old and mathematical should be so controversial. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical re-use of registers during serial multiplication and division, still command respect to-day. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest American stored program machine 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 principle 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 man's *negligible* must be another's *everything*.

The First Dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann refused to include it in the machine at Princeton. In an influential report co-authored in 1946 with H. H. Goldstine and A. W. Burks he explained his reasons thus:

> *" Several of the digital computers being planned or built in this country or in England are to contain a so-called 'floating decimal point.' ... There appear to be two major purposes in a 'floating' decimal point system both of which arise from the fact that the number of digits in a word is constant, fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is 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."*

" 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 proved otherwise. A programmer had to choose scale factors to accommodate the largest magnitudes he could not be sure his variables would not take; therefore he chose his scale factors pessimistically. Therefore, variables tended to take values almost always much smaller than could fit in the fields allocated for them; most leading bits went to waste.

This wastage became obvious to practitioners on early machines that displayed all their memory bits as dots on cathode ray tubes (like TV screens) . A partial remedy was called " floating vectors "; the idea was to compute at run-time 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 on the *Pilot ACE* (an English computer built to specifications considerably less ambitious than had first been put forward by Alan M. Turing, among whose previous designs were the still very secret computers the British used to decipher German radio messages during World War II). Floating vectors continued both to waste leading bits and to require careful prescaling lest rightmost bits be lost from some tiny values of high subsequent significance. Where floating-point deserved to be used, no practical alternative existed.

By 1952 every electronic computer solving engineering problems had software floating-point available for programmers to use if they wished, despite its awesome speed penalty, von Neumann's contrary views notwithstanding. Were his opinions just wrong?

In retrospect, we surmise that he was trying to make a virtue of necessity. Had floating-point been part of the original design, it might well have been too complex to build successfully from the hardware components (vacuum tubes) available at the time. In Germany in 1939, K. Zuse had proposed to build a computer with floating-point hardware, and the German Air Ministry had denied authorization for its construction on the grounds that so complex a machine could not be built within a year or two, by which time they expected to have won the war. (The rest of the world must be grateful now for that decision.) Whether von Neumann knew about that decision or not in 1946, his instincts probably averted a similar fate for the Princeton project.

By 1957, magnetic cores made larger faster memories commonplace; semi-conductors were beginning to replace bulky unreliable vacuum tubes; and floating-point hardware was almost ubiquitous. David H. Wheeler had microprogrammed binary floating-point into M. V. Wilkes' *EDSAC II* in Cambridge, England; in the U. S., 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-precison as well as single. Everybody had it, but it was everywhere different.

Diversity vs. Portability

Since roundoff introduces a bit of error into almost all floating-point operations, to complain about another bit of error seems picayune. So, for twenty years nobody complained much that those operations behaved a little differently on different machines. If software required clever tricks to circumvent these idiosyncracies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a while programmers succeeded magnificently, especially at matrix computations which (though this was not appreciated at first) turned out to be sensitive to the details of computer arithmetic in only a very few places. 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 (Algol or Fortran at that time) to be compiled and executed on practically any commercially significant machine, and will then perform its task at least almost as well as can any other program perform that task on that machine. In so far as numerical software has often been thought to consist entirely of machine-independent mathematical formulas, its portability has often been taken for granted; the mistake in that presumption will become clearer later. What should be clear now is that software is far more valuable if portable than if not.

Packages like LINPACK and EISPACK had 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 was costing us all.

A Backward Step

Early evidence that somewhat different arithmetics could engender grossly different software development costs had been put before us in 1964 and overlooked. 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 was Hirondo Kuki, a mathematician working at the University of Chicago as a programmer, who had considerably improved the run-time library (SQRT, EXP, COS, ...) for the 7094 and then, under contract to IBM, had produced the first run-time library for system /360. He described the tricks he had been forced to devise to achieve for the /360 library a level of quality not quite so fine as he had previously achieved for the 7094.

Members of the audience who did not know Kuki well thought he was boasting about his cleverness; he was not. He was trying to warn us that the /360's floating-point differed from the 7094's in disagreeable ways that would oblige us all to entertain tricks like his unless the /360's arithmetic were changed. Many months passed and many /360s were delivered before SHARE's membership came to appreciate how troublesome that obligation would become.

Part of the trouble could have been foretold by von Neumann had he been still alive. In 1948 he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it; however, it did predict correctly that computations with larger arrays of data would probably fall prey more often to roundoff. IBM /360s had bigger memories than 7094s, so data arrays could grow bigger and did. To make matters worse, industrial advances had come to depend upon calculations more accurate than before; but new /360s had narrower single-precision words (32 bits vs. 36) and used a cruder arithmetic (hexadecimal vs. binary) with consequently poorer worst-case precision (21 sig. bits vs. 27) than old 7094s. Consequently software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on /360s. The quickest way to recover adequate accuracy was to replace old codes' SINGLE PRECISION declarations by DOUBLE PRECISION before recompilation for the /360. This practice exercised /360 double-precision far more than had been expected.

Also unexpected was the frequency with which double-precision versions of all software, old and new, failed mysteriously on /360s in ways that we had not seen happen in single-precision. That was what Kuki had tried to warn us would happen.

The early /360s' 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. 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 last aberration in double-precision undermined a precious theorem that single-precision (and most other machines' arithmetics then and now) honored:

> If $1/2 \le X/Y \le 2$ then no rounding error can occur when $X - Y$ is computed; it must be computed exactly.
> (However, when X and Y are so tiny that exponent underflow afflicts $X - Y$, machines that do not conform to IEEE 754/854 may flush it to 0.0 .)

Innumerable computations had benefitted from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof; see pp. 38-40 of Goldberg [1991], or pp. 138-153 of Sterbenz (1974) cited therein, for proofs and some applications. We had been taking all this stuff for granted.

Until we lost them, we had not appreciated how important were the identities and theorems about exact relationships that persisted, despite roundoff, when approximate arithmetic was implemented in reasonable ways. Previously, all that had been thought to matter were precision (how many significant were digits carried) and range (the spread between over/underflow thresholds). Since /360's double-precision had more precision and wider range than 7094's, software was expected to continue to work at least about as well as before. That was a mistake comparable to ignoring the nutritional importance of vitamins and traces of minerals.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on /360s. In 1966 at a SHARE tutorial on the subject they vented their feelings:
" My old 704 never used to do things like that to me."
IBM heard them and agreed to meet a small subcommittee of SHARE, Kuki included, that had been trying vainly for two years to get IBM to alter /360 floating-point. This committee was surprised and grateful to get a fair part of what they asked for, including all-important guard digits. By 1968 these had been retrofitted to /360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility. Those few IBM employees who can remember the incident still shudder.

> But every army large enough includes somebody who does not get the message, or forgets it.

The Boys who Built Bombs
Seymour Cray has been associated for decades with computers that were, when he built them, the world's biggest and fastest. He designed the arithmetic for the Univac 1107 in the late 1950s, the CDC 6600 in the mid 1960s, and his namesake CRAYs in the late 1970s. He has 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 of the Atomic Energy Commission (now the Department of Energy) like those at Livermore and Los Alamos where nuclear weapons were designed. The boys who built bombs had to overcome Mother Nature's challenges, next to which the challenges of interesting arithmetics must have seemed pretty tame.

Perhaps all of us could learn to live with arithmetic idiosyncracy if only one computer's idiosyncracies had to be endured. Instead, when different computers' different anomalies, each one a petty nuisance by itself, accumulate, then software dies the *Death of a Thousand Cuts*. Here is an example from just Cray's machines:

```
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 X existed which 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 by

```
if ( 1.0*X == 0.0 )  Y = 17.0  else  Y = Z/X .
```

Outrages like this had to be perpetrated in all software intended to be portable to the 6600 and its descendants (7600 and Cyber 17x). Then CRAYs appeared; they can terminate the foregoing statement's execution on an OVERFLOW ERROR in two ways. One is to produce numbers X so huge that 1.0*X overflows although 0.9999*X would be safe; this can happen because CRAYs test for exponent overflow during multiplication before the final one-bit left shift for normalization, if needed, occurs. The second way an undeserved OVERFLOW ERROR can happen is when both Z and X are very tiny, and hence Z/X is not big at all. Trouble can arise here because CRAYs compute not Z/X but Z*(1/X) , and 1/X can overflow before the multiplication if X is too tiny.

Of course these calamities must be rare since a CRAY's overflow threshold is gargantuan, near 10^{2466} ; that is why the boys who build bombs are willing to take their chances that such calamities will not occur. The risk of calamity weighs most heavily upon the conscientious would-be portable programmer, the kind we wish had programmed the last piece of software that let us down. How might she revise the foregoing statement so that it will always work? It cannot be simple since it depends upon the provenance of Z .

. . . . . .

" For most men (till by losing rendered sager)
Will back their own opinions by a wager." Lord Byron

Risk management challenges good taste, and excites pleasure among those so confident of their own good taste that they can attribute their occasional lapses to inevitable but excusable bad luck. It must have been bad luck that befell James Sethian in 1989 when he tried to compute on a CRAY Y-MP the Fortran expression

```
ACOS( X / SQRT( X*X + Y*Y ) )
```

and received rude messages instead. He knew that X and Y were never both extremely tiny, nor could either of them get terribly

big; and this knowledge was confirmed after he and CRAY's local engineers had spent a week or so to uncover his troubles' cause:

Roundoff had led to a computed value X/SQRT(X*X + Y*Y) > 1.0 , and ACOS had quite rightly declined to compute its arccos.

The boys who built bombs were not surprised; five rounding errors contaminated the computation of X/SQRT(X*X + Y*Y) which, in case $|Y| < X/10^8$, would come so close to 1 that two rounding errors might easily push it beyond. An elementary programming oversight.

That's not quite true. Despite five rounding errors, the values computed for X/SQRT(X*X + Y*Y) *always* lie between -1.0 and +1.0 inclusive on *every* commercially significant computer and calculator except CRAYs (and perhaps the Intel RISC i860 chip if it is used with grubby divide and square root software). This assertion is provable mathematically, though the proof is easier for binary floating-point on IEEE 754/854 conforming machines or DEC VAXs than for hexadecimal on IBM /360s or /370s or their imitators, and much harder for decimal calculators. On CRAYs, however, if X is chosen at random and $|Y| < X/10^8$ then the computed values in question will exceed 1 by a rounding error at least about 5% of the time, according to recent experiments.

Why is a CRAY so unlucky? Its multiplier is economized; about 1/3 of the bits that normal multiplier arrays generate have been left out of CRAY multipliers because they would contribute less than a unit to the last place of the final CRAY-rounded product. Consequently a CRAY's 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. On a CRAY Sethian's formula entails rather more than five rounding errors, so it must be replaced by something more complicated:

ACOS(MIN(ONE, MAX(-ONE, X/SQRT(X*X + Y*Y)))) .

The tests and branches implicit in MIN and MAX are unnecessary nuisances when this formula is compiled to machines other than CRAYs, and doubly annoying because they replicate tests already incorporated into ACOS.

Sethian was unlucky in his choice of a CRAY to run his program, but lucky because its failings were comparatively easy to uncover. Failures due to roundoff are generally harder to find and harder again to remedy. Far worse are the situations when idiosyncratic arithmetic leads to miscalculations that are accepted unwittingly as correct, so subsequently a feasible project is misclassified as impossible and not attempted; such errors are rarely uncovered until far too late if ever. Nobody knows how often they occur. A real-life occurrence of that kind will be described later.

Computing has laws like those of Mathematics and of Physics. We cannot know all those laws nor may we break one with impunity; those laws predict that anything bad that can happen will happen.

To-day's CRAYs lack a guard digit in subtraction partly because Cray's customers had become accustomed to getting along without it on his earlier machines. The CDC 6600, arguably the world's first real RISC computer, had not so much lacked a guard digit as it had been made to look that way by almost all its compilers; it needed fully five machine instructions to subtract with a guard digit, only two to subtract without. The fast way drove out the slow even though the fast was occasionally slightly wrong.

The boys who built bombs rationalized the lack of a guard digit in several ways:

1. *Lack of a guard digit does not invalidate error-analyses like those for matrix computation published in Wilkinson's books.* The irony here is that Wilkinson had devised his analyses to *explain* arithmetic without a guard digit, not to *excuse* it.

2. *If a program fails for lack of a guard digit, it can usually be repaired, as Sethian's was, at a tolerable cost.* Can that cost be appraised realistically by employees of an organization whose motto, according to one wag, used to be
Why use Lead *when* Gold *will do*?

3. *If a program fails for lack of a guard digit and cannot be repaired or replaced at a tolerable cost, it was probably contrived for that purpose and need not concern practical men.*

The third rationalization used to rankle programmers who, to make software portable to old CDC's and CRAY's as well as ordinary arithmetics, had been compelled to elaborate clever programs into devious ones. Thanks to their efforts, the third rationalization was being turned into a self-fulfilling prophecy. But no longer.

A decade ago, a new algorithm was proposed for the computation on parallel machines of eigenvectors and eigenvalues of big symmetric matrices. This computation interests engineers, scientists and statisticians intensely. The new algorithm ran rather faster than its predecessors on all known computer architectures, serial or parallel, vectorized or distributed. Alas, roundoff rendered it numerically unstable; from time to time it produced plausible but incorrect results. The bigger the dimensions of the matrix, the greater the likelihood that some part of the output were wrong.

Ten years of diligent error analysis has developed that algorithm into one portable program which, without compromising its speed, achieves full numerical stability on all commercially significant computers except CRAYs. No other program nearly so fast has been found to work on CRAYs, and not for lack of trying. Peter Tang and Danny Sorensen, the authors of that program (they call the critical part Acc_Sec) conclude thus [1991]:

" Finally, it is unfortunate that neither Reform nor Acc_Sec would work on machines whose arithmetic subtraction lacks a guard digit (or bit); notable examples are Crays and CDCs. ... Since, despite the many anomalies Cray's arithmetic offers, merely adding a guard digit (or bit) would allow Acc_Sec to work, it is high time to implement that crucial bit — especially since the implementation cost with today's technology is negligible."

A Financial Calculation

The following experiment is intended to be performed upon a shirt-pocket calculator that displays at least 10 sig. dec. Enter a 10-digit telephone number, including area code; take its natural logarithm; and then take its exponential. Since exp(ln(x)) = x , the expected result must be the original telephone number, yet many telephone numbers change in their last digit or two. Look:

LN(9185551212) = 22.94089757 , EXP(22.94089757) = 9185551249 .

In 1974 this experiment was part of a full-page advertisement for a new 10-digit shirt-pocket calculator which, the ad said, would always return to the original telephone number. That could not have been said for the reigning calculators at that time, the Hewlett-Packard HP-45 and HP-65. The ad neglected to mention a far more bizarre phenomenon revealed only by several repetitions of the exp(ln(...)) operation: the new calculator's telephone number would diminish by 1 after every seven or so repetitions, whereas the HP calculators would stick with the second telephone number they displayed.

How can the calculators' different behavior be explained? Which behavior is preferable?

The behavior of the HP calculators is easy to explain. Let LN be obtained from ln by rounding it correctly to 10 sig. dec.;

ln(9185551212) = 22.94089756 60066347 ... ,
LN(9185551212) = 22.94089757 .

Obtain EXP from exp similarly;

exp(22.94089757) = 9185551248.68126156 ... ,
EXP(22.94089757) = 9185551249. .

Although the errors committed in rounding to 10 sig. dec. are as small as they can be, each under half a unit in the last place retained, two such errors suffice to force the telephone number to change by 37 . On the other hand, repeated EXP(LN(...)) operations leave the second telephone number unchanged; in fact, the identity EXP(LN(EXP(LN(x)))) = EXP(LN(x)) is easy to prove for every telephone number x . Therefore, the HP calculators behaved as well as could be expected from 10-digit machines.

Explaining the behavior of the new calculator was not so easy. It actually carried 13 sig. dec. in all its internal registers but rounded the displayed value to 10 sig. dec. and accepted at most 10 sig. dec. from the keyboard. This strategy brings to mind a mother's advice to a modest maiden: *You need not show Everything.*

Unfortunately, the calculator's arithmetic lacked guard digits in multiplication and subtraction, and discarded rightmost digits by chopping instead of rounding. For ln(9185551212) it would get something like 22.94089756 599 but display only 22.94089757 ; after that exp(22.94089756 599) would be computed as something like 9185551211.846 , low by about 0.0012... , but displayed as 9185551212 . What you saw on this new calculator was not quite what you got.

Repeating the exp(ln(...)) operation would cause the 13-digit value held in a register to drift slowly downward by a few units in the eleventh or twelfth digit, leaving the 10-digit display unaffected until after several downward drifts had taken place. Since the calculator provided no straightforward way to see all 13 digits, the display's spasmodic decreases seemed whimsically unpredictable.

That whimsy had not yet become generally known when a meeting was convened at Hewlett-Packard to consider the company's reaction to the advertisement. This was not the first time (nor will it be the last) for a company to face published benchmark results that cast aspersions upon its product's integrity and prowess. A few test calculations performed upon the new competitor and the old HP machines to compare their utility for typical engineering work had not turned up anything much to choose between them other than the telephone number test. Consequently the advertisement's significance as a benchmark was unclear. Opinions varied, from a conviction that quibbling about digits beyond the 6th (never mind the 10th or 13th) was a waste of time, to a fear that that benchmark was a harbinger of something calamitously worse.

Also present at the meeting as a consultant was a mathematics professor from a nearby university. His first contribution to the meeting was an exhibition of the new calculator's whimsy described above; this was amusing. Next he contributed a demonstration that each of the three calculators under consideration had its own inexplicable idiosyncracies, any one of which could yield results unexpectedly far from reasonable expectations; this was alarming. His third contribution was a proposal to change the way Hewlett-Packard calculators carried out their arithmetic; this proposal was confusing, but what else could be expected from a professor?

His proposal was to continue displaying 10 sig. dec., accepting it from the keyboard, and storing it in those registers that held the calculator user's data and intermediate results, but to make a few wider registers with, say, 13 sig. dec. and 3 exponent digits available to whoever had to microprogram the calculator's algorithms. For every built-in operation invoked by a keystroke, say LN(x) for example, the data x would first have three trailing zeros appended, then the logarithm microprogram would be executed carrying about 13 sig. dec. and its result rounded to produce LN(x) very nearly correctly rounded to 10 sig. dec.

It sounded bizarre. First build extra digits into the central processing unit; then throw them away.

Dennis Harms, a mathematician working as a microprogrammer at the time, attended that meeting too. He decided he could implement this proposal and test it sooner than he could check the intricate error-analysis put forward to motivate it. He confirmed what the consultant had predicted, that carrying 13 sig. dec. internally permitted almost every function in the calculator to be performed beautifully rounded to 10 sig. dec. with a comparatively simple algorithm, and the anomalies that afflicted the older calculators went away. All HP calculators now carry those extra digits.

For several years the architectural changes Harms had introduced demonstrated their value by simplifying the introduction of novel and mathematically sophisticated numerical algorithms into service among new scientific and financial calculators. Let's look at a typical financial task, the computation of the interest rate x for a transaction with an initial cash flow A , a final cash flow F , and N-1 constant and regularly spaced cash flows P between them. The signs of the cash flows are positive for income and negative for outflow. The periodic interest rate satisfies

$$A(1+x)^N + P((1+x)^N - 1)/x + F = 0 \ .$$

N can be huge, over 100000 if electronic funds transfers take place hourly; and if N is huge then x can be so tiny that the middle term degenerates into roundoff/x unless something special is done to cope with this *removable singularity*. Also the graph of the equation is not a gentle curve but almost has a corner or spike if N is huge. That is why solving the equation for x poses three interesting challenges:

1: Devise a program that copes with all data that determine only one root $x > -1$. Any data sequence {A, P, F} with just one sign-reversal is admissible and so is any positive integer N though any $N > 1000000$ has at best speculative significance.
2: That program must finish in fewer than 250 floating-point operations lest customers lose patience waiting for it.
3: The result x must be accurate to within a unit in its last digit displayed, or the ninth digit after the decimal point.

Of course the challenges were overcome, and with very gratifying results; one of the calculators that use that program, the HP-12C , is still on the market ten years after its introduction, serving real estate brokers and financial advisors etc. as part of their uniform. More important, two valuable lessons were learned from several years of continuous experience designing calculators:

First, every detail of the arithmetic and its architecture seemed to matter; none could be changed without the risk of disagreeable consequences. For instance, a microprogrammer working from the consultant's notes found a place where an intermediate result was rounded to 10 sig. dec. and, thinking that to be an oversight, changed it to 13 to achieve higher accuracy. The program broke; it had not been an oversight. Somehow the algorithm depended upon the relationship between the two precisions, 10 and 13 sig. dec.

Second, we learned that careful attention to the details of one's computer arithmetic could confer considerable commercial advantage over less meticulous competitors. This is how *Quality Pays Off*. It is a lesson learned at great cost by the automobile, appliance and semiconductor industries, among others. Unfortunately, the payoff comes too slowly to be appreciated by workers who change jobs after less than five years, as all too many have done for a few decades in the fast-changing computer industry. And a payoff need never come unless the marketplace informs itself about aspects of quality that can at times be difficult to quantify.

> Never under-estimate the awesome harm an ill-informed marketplace can inflict upon itself.

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 Mk. 1 computer. (This precocious machine, with an instruction set attributed to Turing, had been built for the engineering market but, after a handful were sold, it flopped ignominiously partly because its mean free time between errors was fifteen minutes on a good day.) By starting so 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, desk-top mechanical calculators, table-top analog differential analyzers, plug-board programmed accounting machines, and all but the very earliest electronic computers and calculators mentioned in this history.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two post-doctoral years in England, where he became acquainted with Wilkes, Wheeler and especially Wilkinson. In 1960 he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free reign to tinker with its operating system, Fortran compiler, and run-time library, but not its hardware. (He denies he ever came near the 7090 with a soldering iron in his hand, but admits to asking to do so.) One of his stories from that time illuminates how misconceptions and numerical anomalies built unwittingly into a computer system can incur awesome hidden costs.

In 1962 a graduate student of aeronautical engineering had been using the 7090 to simulate wings he was designing for very short take-offs 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 had given up that line of inquiry and was looking for some other problem on which to write a Ph. D. thesis.

Meanwhile Kahan had decided to replace IBM's logarithm program (ALOG) in the run-time library by one of his own he hoped would provide better accuracy. While testing it, and to see how much difference it would make to other 7090 users, he re-ran those of their jobs that had used the old ALOG using the new instead. (If that were feasible nowadays it would be a deplorable invasion of privacy.) Two results had changed significantly; one was the student's, and Kahan approached him to find out what happened.

The student was flattered that his work interested anyone else, but puzzled. Much as the student preferred results with the new ALOG — they predicted a gradual stall — he knew they must be wrong because they disagreed with his DOUBLE PRECISION results. And his program did not call ALOG. It did include something like

```
C = EXP(-G(1.0))
...
X = ...
Y = C
IF ( X ≠ 1.0 )  Y = X**(G(X)/(1.0 - X))  .
```

This last line explained how ALOG entered the picture; at that time Fortran compilers defined X**Z as EXP(Z*ALOG(X)) .

The discrepancy between SINGLE and DOUBLE PRECISION results went away a few days later when a new release of IBM's DOUBLE PRECISION arithmetic software for the 7090 arrived. Now the student's wing stalled gradually in both precisions. He went on to write a thesis about it and to build it; it performed as he had predicted. But that is not the story's happy ending.

In 1963 the 7090 was replaced by a faster 7094 with double-precision floating-point hardware but otherwise practically the same instruction-set as the 7090. Only in DOUBLE PRECISION and only 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, to lack a guard bit in DOUBLE PRECISION. Now that he knew what was missing, the student figured out how to overcome this deficiency; he simply replaced (1.0 - X) by ((0.5 - X) + 0.5) . With this trick, the program worked perfectly. But that is not the story's happy ending.

The student's program worked well on every computer he met until he went to work for an aircraft manufacturer. The program failed on his new UNIVAC 1107; it lacked a guard bit in subtraction too but, inexplicably, the trick seemed not to compensate for that. The now ex-student had to replace X**(G(X)/((0.5 - X) + 0.5)) by an entirely different program, based upon power series, that ran slower and yielded poorer results barely adequate for the purpose. Only several years later did he discover that the trouble on the UNIVAC 1107 had been caused by a compiler that had " optimized " ((0.5 - X) + 0.5) back to (1.0 - X) in order to save one add.

. . . . . .

" God is in the details." ... ????? Some cleric ?????
" The Devil is in the details." ... ???? Some architect ????
" Our life is frittered away by detail." ... Henry David Thoreau

By 1963, Kahan had become active in SHARE working with Kuki and others to improve the run-time library and, after /360 was announced, to correct its hardware. In early 1966 he visited Stanford University's pioneering Computer Science Department, still being run by its founder George Forsythe. There, besides teaching about error-analysis, Kahan sampled the dubious joys of trying to run programs on two utterly different computers sitting in adjacent rooms; one was an IBM 7090, the other a BURROUGHS B 5500. He found that he could get about four times as much work through the B 5500 as through the 7090 despite that the 7090 ran long floating-point intensive computations four times faster, and despite his thorough familiarity with the 7090's system.

Kahan traced the difference in throughput to fanatical attention to detail lavished upon the B 5500's design by its architect, Robert S. Barton, and perhaps also by a summer hire from Cal. Tech., Donald E. Knuth, who had worked on the run-time library. A programmer could easily predict what the B 5500 would do *for* her (in Algol), but had to imagine what the 7090 might do *to* her (in Fortran). Unfortunately, attention to detail retards the designer's progress and is, therefore, a losing strategy for a marketplace that values megaflops/sec. over all else. Stanford replaced its B 5500 by a " faster " IBM /360-67 in 1967.

On returning to Toronto, Kahan combined his experience with the 7094 and B 5500 into a scheme for handling exceptions humanely on the /360 and, with the aid of friends at the University of Waterloo, implemented it on an early /360 there. However, IBM software developers in New York had committed themselves to an elaborate scheme of their own which persists to-day though nobody uses it. In general, the handling of floating-point exceptions like over/underflow and division by zero is in roughly the same state it was in 25 years ago, but getting worse because of a trend towards concurrent computation and imprecise interrupts.

Meanwhile the lure of California was working on Kahan and his family. At the request of his younger son, then aged 3, they came to Berkeley and he to the University of California there.

Berkeley's computer was a CDC 6400, a cheaper 6600. Attempts to convert Kahan's Fortran library from the 7094 to the 6400 progressed glacially. The last straw was an inscrutable failure on the 6400 of a program that, on the 7094, had automatically constructed regular solutions for singular differential equations, something regarded as beyond the state of numerical analysis then and, in some quarters, now. Convinced that bugs infested the 6400's hardware, and aided by logic circuit diagrams in out-of-date maintenance manuals tossed by some well-wisher into Kahan's wastebasket, he and a student, David Lindsay, pored over core dumps. There was the evidence that convicted both compiler and hardware of arithmetical perversity, but nobody at CDC with the power to change things felt able to act on it in 1970.

Lindsay's report reached another 6400 user, Niklaus Wirth in Zurich, who had encountered similarly inscrutable anomalies while

testing a compiler for his new language *Pascal*. Armed now with incontrovertible evidence, Wirth sent a letter of remonstrance to CDC's office in Zurich. The reply said (in German)

" 1. CDC's arithmetic does not do what you allege, and
2. If it did, it would not matter. "

Wirth concluded that the best Pascal could do with the REAL data-type was leave it " implementation-defined;" the less said about it, the better. Kahan concluded that a diverting paper [1972] was all he could salvage from the affair. Several years later at the University of Minnesota in Minneapolis a Fortran compiler was written that compensated for some of the 6600's strange ways, but by then the end of the 6600 - 7600 - Cyber 17x line was in sight.

. . . . . .

A top-down approach to high-quality floating-point seemed still a fair approach in 1972-3 when, with Fred Gustavson and Fred Ris at IBM Research in Yorktown Heights, Kahan helped develop an ideal floating-point specification for a *Future System* being contemplated to supplant the /360 (now /370) architecture. It was a paper study; IBM knew that the mid 1970s was not the right time (will it ever come?) to kill the goose that laid golden eggs for it. Similar reasons would keep every other major mainframe maker from changing its floating-point in any way not compatible with prior practice. A top-down approach led nowhere.

The bottom-up approach presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and the consequences have been described above. Fruitful collaboration with congenial coworkers restored his spirits and prepared 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. The 8086 was imminent, and an 8087 floating-point coprocessor for the 8086 was being contemplated. 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 would have been obviously 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 anarchic 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 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 onto one chip some awesome floating-point well beyond anything its competitors had in mind. They 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 a submission from Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based upon the Intel design, with Intel's permission of course, as simplified by Coonen. Their harmonious combination of features, almost none 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 endorsement, and that took time.

In 1980 Intel became tired of waiting and released the 8087 for use in the IBM PC. In 1982 Motorola announced its 68881, which found a place in Sun IIIs and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Zilog, with Coonen's help, produced its Z 8070 coprocessor for the Z 8000 but then withdrew from a market that was getting crowded. Another Berkeley graduate student, George S. Taylor, had soon designed his second high-speed implementation of the proposed standard for the ELXSI 6400, an early super-minicomputer. The standard was becoming *de-facto* before its final draft's ink was dry.

Among the attractions of IEEE 754 was a *Majorization Property*:

> Numerical software, designed to be portable to pre-existing computers or capable of running after recompilation on at least two different arithmetics, would almost surely run at least about as well on a standard-conforming machine as on any other of nearly the same memory capacity, speed and precision.

This property was a design goal for the standard, as it had been for the 8087. It evidenced a kind of mathematical cleanliness in a design that introduced the minimum possible irregularity. For example, ...

```
for  k = 1, 2, 3, .., 8000000  do
  x := float(k) ;
  for  j = 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20, ..,  do
    y := float(j) ;
    q := x/y ;                ...  rounded in floating-point.
    p := q*y ;                ...  rounded in floating-point.
    if ( p ≠ x )  print "Oops!" ;
    next j ;                  ...  any sum of two powers of  2 .
  next k ;
print "End" .
```

In the absence of roundoff, " Oops! " could never be printed; and IEEE 754 floating-point arithmetic never prints " Oops! " either; but every other kind of floating-point arithmetic does print " Oops! " for some integers j and k that vary from one machine to another. This is not much of a test, but it does show that IEEE 754 arithmetic preserves something all others lose. 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 in design. Had the industry's engineering managers realized this fact, they might not have been so quick to affirm that, as a matter of policy,

" We conform to all applicable standards."

To-day the computing industry is enmeshed in a host of standards that evolve continuously as technology changes. The floating-point standards IEEE 754/854 (they are practically the same) stand in somewhat splendid isolation only because nobody wishes to repeat the protracted wrangling that surrounded their birth when, with unprecedented generosity, the representatives of hardware interests acceded to the demands of those few who represented the interests of mathematical and numerical software. Unfortunately, the compiler writing community was not represented adequately in the wrangling, and now that community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such feature; directed rounding another. Without compiler support those features could atrophy;

" Use it or lose it."

The relentless appetite for speed also tends to erode the quality of floating-point arithmetic by cutting corners that might easily go unnoticed, for instance by rounding in an idiosyncratic way. Without performance specification that computer users can check by running benchmarks, the customer is obliged to accept assurances from vendors about conformity to a standard that, even with the best intentions, is difficult to corroborate. Unfortunately the computing industry has gotten mixed signals from benchmarks for speed; to introduce benchmarks for accuracy too could throw open *Pandora's Box* because of the *Stopped Watch Paradox*:

> *A stopped watch is more accurate than any working watch because a working watch almost never tells the correct time exactly, but a stopped watch must be exactly right twice a day.*

Similarly, software that is usually quite wrong can give exactly correct results for a few problems for which high-quality software yields merely extremely good approximations. Similar phenomena can occur with floating-point hardware; recall the telephone number experiment. Guess which kinds of problems might well turn up as benchmarks! This is an area that needs further work.

At present, 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 except Cray Research Inc., and they have announced recently that they too will conform *to some degree* by the mid 1990s to ease the transfer of data files and portable software between CRAYs and the workstations through which CRAY users have come to access their machines nowadays.

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 gave thanks to his many associates for their diligent support, and to his adversaries for their blunders. So, not all errors are bad.

References and Further Reading

Readers interested in learning more about floating-point will find two publications by David Goldberg [1990, 1991] good starting points; they abound with pointers to further reading. Several of the stories told above come from Kahan [1972, 1983]. The latest word on the state of the art of computer arithmetic is often found in the *Proceedings* of the latest IEEE sponsored *Symposium on Computer Arithmetic*, held every two or three years; the tenth was held in 1991.

A. W. Burks, 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* ed. by W. Aspray and A. Burks, The MIT Press, Cambridge Mass., and Tomash Publishers, Los Angeles Calif. (1987) pp. 97-146.

D. Goldberg [1990] "Computer Arithmetic" Appendix A of *Computer Architecture: A Quantitative Approach*, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Mateo Calif.

D. Goldberg [1991] "What Every Computer Scientist Should Know About Floating-Point Arithmetic" *ACM Computing Surveys* **23** #1, 5-48.

W. Kahan [1972] "A Survey of Error-Analysis" in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana) vol. 2, pp. 1214-39, North Holland Publishing, Amsterdam.

W. Kahan [1983] "Mathematics Written in Sand" *Proc. Amer. Stat. Assoc.* Joint Summer Meetings of 1983, *Statistical Computing Section*, pp. 12 - 26.

D. C. Sorenson and P. T. P. Tang [1991] "On the Orthogonality of Eigenvectors Computed by Divide-and-Conquer Techniques" *SIAM J. Numer. Anal.* **28**, 1752-75.

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