Notes prepared for a presentation

BUMPS ON THE PATH TO FLOATING-POINT PROGRESS

for the IEEE-sponsored

HOT CHIPS SYMPOSIUM

at Stanford University

June 26-7, 1989

by Prof. W. Kahan

Elect. Eng. & Computer Science,
University of California
Berkeley CA 94720

... the race goes not always to the swift,
nor the battle to the strong,
neither security to the prudent
nor wealth to the well-connected,
nor yet success to the skilful,
but Time and Chance happens to them all.

- a version of Ecclesiastes IX-11.

The RISC philosophy can tempt engineers to take risks that deserve more thought. As we attempt not just to produce faster computers, but also to produce them faster, we impose upon the fabric of the computer industry strains that could undo our efforts:

By concentrating too hard on optimizing designs for the most common situations, do we undermine performance by requiring defensive programming to cope with exceptions too rare to figure largely in our thoughts, yet not rare enough that they can be ignored?

Can compilers for new architectures really exploit their supposed architectural advantages without conflicting with long-established habits among programmers? Do enough good compiler writers, knowledgeable not only about computer architectures but also applications, exist?

We all want software to be reusable, portable or at least transportable; but excessively diverse computer architectures pose unresolvable dilemmas for the would-be portable programmer.

How do we strike a balance between innovation and stability, between healthy diversity and fatal dispersion of effort?

How Intel 80x87 Stack Over/Underflow Should Have Been Handled.

by W. Kahan

The Intel 80x87 family of numeric co-processors keep their eight floating-point operands in a Stack . Trying to push or generate a ninth operand on the stack precipitates instead a stack overflow exception; trying to reference an empty cell on the stack causes a stack underflow exception. These exceptions are expensive to handle in software because the handler has too much work to do:
- Discriminate between stack over/underflow and other INVALID operations (easier on the 80387 than its predecessors).
- Decide what to copy between stack and its extension in memory.
- Retry the operation that was thwarted by stack over/underflow.

This expense could be reduced substantially by slightly revising what the 80x87 hardware does. Such a revision would bring the chip into line with the original intention for its design, which was frustrated by misunderstandings between the specifiers and the implementors of the 8087 ; see p. 93 of *The 8087 Primer* by John Palmer and Stephen P. Morse (1984), Wiley, N. Y.

Frustration continues. All attempts to persuade Intel's chip implementors that 80x87 stack over/underflow handling has to be fixed by hardware modifications have failed. Intel's attitude seems to be " it's just a matter of software." But software to cope with the problem has yet to appear in Intel's own CEL run-time library for the 80x87 family, and is elsewhere almost nonexistent. Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chips' performance by typically 50% with the spurious stores and loads necessary simply to preclude stack over/underflow.

Compared with architectural changes that have already occurred in the course of evolution from the 8087 to the 80387, changes advocated below to eliminate the stack over/underflow problem are few, simple, upward compatible, and more likely than previous changes to promote improved performance. Curing the stack over/underflow problem will change what is perceived as a disadvantage of 80x87 architectures into an advantage compared with the flat register architectures of the Motorola 68881/2 and WE 321/206 .

How the 80x87 stack should work

Think of the eight registers in the 80x87 as the topmost eight cells of an indefinitely long stack. Floating-point operands and results can travel between memory and stack only via the cell on top of the stack, as is customary for stacks. Every arithmetic operation combines a source operand with a destination operand and writes the result over the latter; one of these two operands must be the cell on top of the stack. The other operand (and possibly destination) can be any of the eight cells in the 80x87 ; this peculiarity of the 80x87 permits subexpressions to remain in the stack for subsequent re-use, and permits more than one floating-point stack to reside ephemerally in the 80x87 during tight loops. The top cell can also be popped, duplicated, or swapped with any other of the eight cells, so anything that can be done with eight registers, as exist on the Motorola 68881/2, can be imitated on the 80x87 at the cost of some swaps.

Why is the 80x87 organized as a stack instead of a flat set of registers like the 68881 ? On p. 60 of their book, Palmer and Morse attribute this choice to limits on available op-code space, but acknowledge that the stack architecture's advantages go beyond making a virtue of necessity. Foremost among them is the freedom from having to save and restore registers when functions that pass their floating-point arguments by value are invoked. These values (normally just one or two) need merely be pushed onto the stack to be consumed and replaced by the function's result. And if an interrupt requests some small floating-point service, such as scaling or transforming the numbers received from or sent to a transducer, that service can be performed quickly on top of the stack without first saving and later restoring its contents provided nothing extra is left on the stack afterwards. Since a floating-point stack's depth fluctuates very little compared with other stacks -- a floating-point stack is often empty and hardly ever has as many as four cells active -- memory traffic during those function calls and interrupts tends to be much lower with a stack than with flat registers. That is an important advantage for machines whose memory bus is much narrower than an operand.

Of course, the foregoing assumes that stack over/underflows will occur very rarely, and that when they occur very little time will be spent unloading or reloading the bottom few cells of the 80x87 to or from an area in memory devoted to the rest of the stack. If the 80x87 had been provided with slightly better facilities to handle stack over/underflow, the second assumption would be true.

Provision for stack over/underflow on the 80x87

The 80x87 has a two-bit tag associated with each stack cell. This tag takes the value 11_2 (in binary) if the cell is EMPTY ; otherwise its value is used by the 8087 and 80287, but not the 80387, to indicate what the cell contains:

00	Finite nonzero number
01	± Zero
10	± Infinity, or NaN (Not-a-Number)
11	EMPTY cell

(The 80387 sets the tag bits the way the other chips do, but ignores distinctions among nonEMPTY cell-tags.)

The 80x87 has a three-bit pointer called TOP that points to the cell that is currently on top of the stack. Pushing another item onto the stack decrements TOP by 1 ; popping an item off the stack increments TOP by 1 . References to cells are always relative to the stack top; a reference to ST(i) is to the cell pointed to by TOP + i . The addition and increments/decrements are all performed modulo 1000_2 in binary; decrementing TOP from 000_2 puts it to 111_2 .

Stack overflow occurs when an attempt to push (FLDx, FILD, FBLD) or create (FPTAN) another item on the stack would decrement TOP to point to a nonEMPTY cell. For that stack overflow the intended remedy is to copy a few cells from the bottom of the 80x87's stack into a downward extension of that stack in memory, and then tag those cells EMPTY to permit the top of stack to grow into them, and then retry the operation that was thwarted.

Stack underflow occurs when an attempt to read a stack cell finds it EMPTY. The intended remedy for stack underflow is to refill that EMPTY cell and perhaps some others from the stack's extension in memory, and retry the operation that was thwarted.

The area in memory devoted to the extension of the 80x87's stack can be tiny; 1280 bytes is almost always ample. Far larger areas might be needed to cope with Recursive (self-calling) programs; but *Recursion* (as distinct from *Recurrence* or *Iteration*) so seldom involves floating-point values that reallocating a larger area, whether on demand at run-time or only after recompilation, should be relegated to the category of remote possibilities.

Software to handle stack over/underflow (especially underflow) turns out to be extremely intricate. Part of the problem springs from differences within the 80x87 family. For instance, format and operating system differences make it necessary for an 80x87 trap handler either to be configured differently for each chip or to recognize at run-time which chip is in the machine; for the 80287 there are two variant configurations to be recognized, one Intel's standard and the other IBM's in PC-ATs. Instruction retry is complicated by differences tantamount to bugs in the ways the chips record an offending operation's op-code:
    80287 and 80387 sense *segment over-ride*, 8087 does not.
    80386 forgets to tell 80387 about FILD (word) .
    80387 treats FXCH's stack underflow anomalously.
The 80387 distinguishes stack over/underflow from other invalid arithmetic operations like 0.0/0.0 by providing (except for the FXCH instruction!) information that the other chips do not; but this information is hard to exploit in codes that have to be portable in binary (.EXE) form to different PC hardware.

Intel has abdicated its responsibility to supply its 80x87 chips with standardized device-driver software that would have hidden their differences and difficulties. Instead PCs are cursed with intractable diversity that renders exception handling uneconomical at every level -- operating system, compiler, application code.

The Blight

For lack of software to handle stack over/underflow, compilers have to preclude it altogether. The simplest and most common way to do that is to leave no intermediate result on the stack unless it is to be used immediately as an operand. Doing so can double the incidence of loads and stores in loops. For example, the inner loop of

```
        s := 0 ;  for  k = 1 to n  do  s := w[k]*z[k] + s ;
```

which computes a scalar product

```
        s  =  w[k]*z[k] + w[k]*z[k] + ... + w[k]*z[k] ,
```

should contain one floating-point multiply, one add, and two loads (of w[k] and z[k]); but the simplest policy to avoid stack overflow would generate three loads (of w[k], z[k] and s) and a store (of s) . Until fairly recently, almost every compiler for IBM PC's used to do that, almost halving the speed of the loop.

Better policies have begun to appear in compilers. Some of them reserve (say) four registers as scratch registers, so they can

retain as many as four values in the stack without having to save and restore any of them when a function is called. Such a policy brings the simplest loops, like the one above, up to speed, but does not do much for others that are common but more complicated. For instance, suppose
$s = q + ir$, $w[k] = u[k] + iv[k]$ and $z[k] = x[k] + iy[k]$
are complex variables, expanding the program segment above into

```
q := 0 ;  r := 0 ;
for  k = 1 to n  do
   {  q := q + u[k]*x[k] - v[k]*y[k] ;
      r := r + v[k]*x[k] + u[k]*y[k]  } ;
```

whose inner loop can be effected on an 80x87 using at most 7 stack cells and four floating-point multiplies, four adds, four loads and one DUPLICATE of the stack's top. But no compiler I know to be governed by a policy that restricts register residency can get by with fewer than six loads, and some take eight.

A number of ugly consequences can be traced to the lack of proper stack over/underflow handling. Expression evaluation should be simpler to compile to an 80x87-like stack architecture than to a flat set of registers whose allocation has to be optimized, but the threat of stack overflow has instead complicated compilers and delayed their dissemination. As arithmetic gets faster relative to memory management, superfluous loads and stores detract ever more severely from performance. Had their deleterious effect upon benchmark runs of the 8087 and 80287 been appreciated sooner, the Weitek 1167 and 3167 might not have been developed; the latter chip was expected to outperform the 80387 by a factor of four but it barely achieves a factor of two with newer compilers that generate fewer superfluous loads and stores. So meager an improvement in speed hardly compensates for the tragic dilution of software development and fragmentation of the market brought about by arithmetic incompatibilities between the two families; Weitek chips lack the Double-Extended (80 bit) format that is the most efficient medium for expression evaluation on the 80x87 family.

The prevalence of superfluous loads and stores among current compilers and applications for the 80x87 cripples the market for a chip identical to the 80x87 but faster. Even if such a chip performed arithmetic twice as fast as the 80x87 it could not run existing software more than about 4/3 as fast since the time now wasted on spurious loads and stores would continue to be wasted.

What Should We Do?

We need a family of standardized device drivers, one for each hardware configuration that includes an 80x87 chip, that hide all exception-handling differences from compilers and applications codes. These drivers must hide stack over/underflow as well as certain arithmetic differences between the 80387 and its two predecessors; the latter differences can be hidden well enough by supplying driver software that makes the 8087 and 80287 conform more nearly to IEEE Standard 754, as does the 80387. Intel's CEL library might have served as such a driver but for its neglect of exception handling and its outrageous price; it still provides a model worth copying in other respects.

The drivers have to be extremely inexpensive if they are to become ubiquitous; otherwise software developers will not use them. And we need some expectation that hardware will evolve to support our driver software efficiently, promising future higher performance as an incentive to convert software to use the drivers now. The support needed from hardware is small, as the rest of this report will attempt to show. The reason that hardware has to evolve is twofold: first, the trap-handlers needed for the present 80x87 hardware are unnecessarily complicated and slow; second, they face a dilemma that can never be resolved perfectly.

The dilemma arises first when the stack overflows; how many cells should be copied from the bottom of the 80x87's stack into its extension in memory? And then when the stack underflows, how many of the 80x87's stack cells should be refilled from memory? An adequate answer to "How many?" is probably three or four, but the best answer may well vary from one program to another.

The dilemma would not arise if the 80x87 had been implemented according to the original intentions. No description of those intentions has been published yet; what follows is the first.

What Should Have Been Done

What follows is the description of a hypothetical 80X87 that differs from current 80x87s only in the way the stack behaves. The same instruction set and the same eight registers are assumed, though Complex arithmetic and Interval arithmetic would fare better with sixteen registers even if only the eight on top of the stack were accessible directly. The hypothetical 80X87 differs from the 80x87 also in the interpretation of the two-bit tags, and in the use of a five-bit two's complement integer to hold TOP even though only its last three bits (TOP mod 8) point to the top of the 80X87's stack. The role played during stack over/underflow by TOP's two leading bits and some other minor changes will be described later.

For the sake of definiteness, suppose the stack extension area in memory is allocated 1280 bytes; since each stack cell occupies 10 bytes, this allows for 128 stack cells all told, addressed from 0 to 127 . These cells can be grouped in 16 blocks of eight, numbered from 0 to 15. The current top of the stack is at address $T = TOP + 8B$, where B is the current block number though 8B is kept in memory. Initially $8B = 128$ and $TOP = 0$ but there is some ambiguity about the representation of T since adding +8 or -8 to 8B and doing the opposite to TOP changes neither T nor the 80X87 cell (TOP mod 8). Pushing an item onto the stack decrements TOP and T ; popping increments them.

Every cell in the 80X87's stack is associated with a cell in the stack's extension in memory although the contents of these two cells may differ. For $0 \le t < 8$, cell number t in the 80X87 associates with cell $8B + TOP - (((TOP \bmod 8) - t) \bmod 8)$ in the extension. The figure shows associated cells when TOP = 2 :

PRINCIPAL INDICATORS of the COST of DEVELOPING and MAINTAINING NUMERICAL SOFTWARE

Indicator	
SUBROUTINE CALLS	Depend upon how complex the program's task is relative to the resources available.
DECLARATIONS	Depend upon the adequacy of Type-Support in the chosen programming language.
INPUT/OUTPUT	Formats, windows, graphics, interaction in real time, ... are big issues in few numerical codes except during debugging.
TESTS & BRANCHES	What to test? Compared with what threshold? Where to go? What to do there?

EXCEPTIONS INTRODUCE COSTS INTO SOFTWARE BY MULTIPLYING TESTS AND BRANCHES, INTRODUCING (SOMETIMES INVISIBLE) SPAGHETTI THAT OBSCURES A PROGRAM'S PATHS OF CONTROL.

WE WISH TO HANDLE EXCEPTIONS WITH A MINIMUM OF TESTS AND BRANCHES, AS FEW OF THEM AS POSSIBLE IN INNER LOOPS, AND WITHOUT ANY NEW OR EXOTIC CONTROL-STRUCTURES.

WHAT EXCEPTIONS?

INXCT:	INEXACT RESULT (only for IEEE 754/854)
UNFLO:	UNDERFLOW (not for CRAY, ...)
DIVBZ:	DIVIDE-BY-ZERO, actually means an INFINITE RESULT COMPUTED EXACTLY FROM FINITE OPERAND(S).
OVFLO:	OVERFLOW (only Floating-Point)
INTXR:	INTEGER EXCEPTION OR ERROR WITH DUBIOUS RESULT ... some overflows, and 1/0 .
INVLD:	INVALID OPERATION, such as ...

ZOVRZ:	0.0/0.0
IOVRI:	INFINITY/INFINITY
INVDV:	either of the above
ZTMSI:	0.0*INFINITY
IMINI:	INFINITY - INFINITY
FODOM:	FUNCTION OUTSIDE ITS DOMAIN, such as ... SQRT(-3) , LOG(-3) , ARCSIN(3) , ...
UNDTA:	UNINITIALIZED DATA or VARIABLE
DTSTR:	ATTEMPTED ACCESS OUTSIDE A DATA STRUCTURE, ... like ARRAY REFERENCE OUT OF BOUNDS
NLPTR:	DEREFERENCING A NIL POINTER

ALLXS: ALL OF THE ABOVE ... for treatment *en masse*.

IN MEMORY		IN AN 80X87			
.....					
(BBBBB)	8B				
(CCCCC)					
DDDDD	T = 8B+2	#2	ddddd	SP(0)	TOP = 2
EEEEE		#3	eeeee	SP(1)	
FFFFF		#4	fffff	SP(2)	
GGGGG		#5	ggggg	SP(3)	
HHHHH		#6	hhhhh	SP(4)	
IIIII		#7	iiiii	SP(5)	
JJJJJ	8B+8	#0	jjjjj	SP(6)	
KKKKK		#1	kkkkk	SP(7)	
LLLLL					
.....					

Every cell in the 80X87 is tagged with two bits to tell first whether that cell is EMPTY, and if nonEMPTY then whether its contents have been COPIED into its associate in the extension area in memory. That copying occurs only when the 80X87 stack over/underflows. Initially all tags are EMPTY.

The only legitimate way to fill an EMPTY cell is to push an item into it, either by loading the item from memory or from another nonEMPTY stack cell, or by creating it during FPTAN ; after that the cell is tagged nonEMPTY and unCOPIED. The same thing happens when an item is pushed onto a cell previously tagged nonEMPTY but COPIED; this dispels the aforementioned dilemma and substantially reduces the incidence of stack overflow on 80X87s.

Stack overflow occurs when an attempt to push or create another item on the 80X87's stack would decrement TOP to point to a nonEMPTY unCOPIED cell. The remedy is to copy all such nonEMPTY unCOPIED cells from the 80X87 into their associates in memory and flag those 80X87 cells COPIED. But first the leading two bits of TOP have to be cleaned up; add or subtract 8 to put TOP strictly between -8 and +8 , and do the opposite to 8B , and then do the copying. Finally retry the operation that caused the overflow; this would be facilitated if the operation and its memory operand had been saved so that retry and return from the overflow trap handler could occur simultaneously.

The 80X87 is supplied with a new instruction that simultaneously retries the saved operation that precipitated stack over/underflow (which was detected as soon as that offending instruction was issued) and returns the host processor from the trap handler; this eliminates a need to decode or copy the offending instruction and prevents unwanted interactions with other kinds of exceptions. There is ample room on the chip to save the offending operation, and either an operand from memory or the address of a destination in memory, in registers not yet used to carry out the offending instruction.

Stack underflow occurs, as before, when an attempt to obtain an operand from a cell finds it tagged EMPTY. The remedy is to first clean up the first two bits of TOP as described before, then copy their associates' contents into all EMPTY cells and tag them nonEMPTY and COPIED, and then return-and-retry.

Finally, the stack over/underflow trap handler must always check for over/underflow of the stack's extension in memory. Overflow entails reallocating the extension to a bigger area. Underflow is probably a blunder. As long as normal stack discipline prevails, whereby only pushes and pops are allowed to lengthen or shorten the stack, EMPTY and nonEMPTY cells will never interlace, so the scheme described above cannot malfunction but must nearly minimize memory traffic.

Is All This Worth The Bother Now?
Compatibility with old software is the way the computer industry plays God ...

" ..., visiting the iniquity of the fathers upon the children unto the third and fourth generation ... ."
Exodus XX-5

No easy way exists to correct a mistake after innumerable sources of software have wound their expedients around it.

The incentive for correcting Intel's mistaken treatment of stack over/underflow must arise among Intel's competitors. Unless new compilers supplant old applications programs by new ones free from most of the superfluous loads and stores that now afflict users of 80x87s, faster versions of the 80x87 will not convey enough of their speed to existing software to justify their higher price, especially since Intel can so easily lower its price for 80x87s when competition looms. A paradoxical aspect of the situation now is that competitors have to promote the dissemination of software that will enhance the performance of Intel's chips in order to create opportunities for them to compete by enhancing performance again. Compiler writers have to see a path along which successive versions of their compilers will enjoy ever better performance as they evolve together with the hardware.

Compiler writers must be sorely tempted by half-measures like the policy mentioned above that sets aside some of the 80x87's stack in order to use the rest efficiently. Half measures can solve a technical problem satisfactorily for so large a fraction of the market as to put satisfaction for the rest beyond the purview of profitable commerce. Only the urge to do things right, and the strength of character to resist temptation, will put the right solution for the 80x87's stack problems into circulation. We know what the right thing to do is; who has strength to do it?

Benjamin Franklin's advice (1757) :

Three removes are as bad as a fire.

A change of address for one's family and possessions used to be called a "remove". B. F. reckoned three of them as risky to one's possessions as a fire. Moving data too often, as required by some computer architectures, also entails risks; branching burns performance severely.

I have come to believe that Vector Architectures are destined for decline in almost all markets because they fly in the face of Benjamin Franklin's advice.

FOUR IMPEDIMENTS to PORTABILITY of NUMERICAL SOFTWARE for SCIENTIFIC and ENGINEERING COMPUTATION:

1. DISPARATE ROUNDOFF PROCESSES AMONG DIVERSE COMPUTER ARITHMETICS.
(Abated by IEEE STANDARDS 754 and 854)

```
*******************************************************************
*                                                                 *
*  2.      DISPARATE RESPONSES TO  EXCEPTIONS                      *
*         ( like   .../0 ,  SQRT(-...) ,   ... )                   *
*          AMONG DIVERSE ...     MACHINES,                         *
*                                OPERATING SYSTEMS,                *
*                                PROGRAMMING LANGUAGES.            *
*                                                                 *
*******************************************************************
```

3. SHORTAGE OF ADEQUATE TESTS and DIAGNOSTIC BENCHMARKS.

4. FEDERAL FUNDING FOR NUMERICAL SOFTWARE DEVELOPMENT

EXCLUSIVELY

FOR MAINFRAME/SUPER-COMPUTERS ARITHMETICALLY WORSE THAN ALMOST ALL WORK-STATIONS.

Gresham's Law:
Bad money drives out Good.

- attributed to Sir Thomas Gresham (1519-79)

(Coins debased by paring or adulteration of precious metal remain in circulation while undebased coins are hoarded, going out of circulation. Have you any real silver coins left in your pocket? Gresham's law is not so much a law of Nature or legislature as a remark about human nature.)

Gresham's Law adapted to Computers:

The Faster drive out the Slower,
even if the Faster are wrong.

The thought that CPU speed is all that really matters is a misperception based upon over-simplification, as if other aspects like accuracy, robustness and intellectual economy had no commercial value.

We should by now have learned from the Japanese that qualities like Integrity are worth more than money, and worth a lot of money in the long run.

SOLVING LINEAR SYSTEMS A LITTLE BIT BETTER
by W. Kahan

We wish to solve
$$AX = B$$
for the N-by-M matrix X .

GAUSSIAN ELIMINATION by Crout-Doolittle Factorization:

$$PA = LU$$

To compute	Permutation matrix	P ,
	Lower triangle	L ,
	Upper triangle	U ;

costs about N^3 operations, mostly in Scalar Products.

Now ($PAX =$) $LUX = PB$ can be solved in steps:

Solve $LC = PB$ for C , and then
$UX = C$ for X ;

costs about N^2M operations, mostly in Scalar Products.

Despite contrary received wisdom, Cache Misses and Page Faults can be mostly avoided.

ITERATIVE REFINEMENT improves approximately calculated X thus:

Residual $R = B - AX$ .... MAIN LIMIT TO ACCURACY IS HERE

Correction Y solved from ($PAY =$) $LUY = PR$.

Improved approximation is $Z = X + Y$.

HOW MUCH BETTER THAN X IS Z ?

All arrays and computation in Double (53 sig. bits) .....

Z is often worse than X !

All arrays in Double, but all scalar products accumulated in Double-Extended (64 sig. bits) .....

Z is always better than X ,
and about 10 bits better than X before,
and even better when N is in the hundreds or more,
AND FASTER TOO WHEN PERFORMED ON i80x87, MC68881/2, WE321/206.

But no compiler support on SUN III, little on IBM PC, ..., !

WHAT MAKES AN EXCEPTION EXCEPTIONAL ?

NOT ITS RARITY

NOT ITS WRONG (?) RESULT

e.g.: X = 3.0/7.0 IS RARE.

Y = (8.0*X - 3.0) - X IS EXACTLY NONZERO;

IS Y WRONG ?

****

AN EXCEPTION IS A COMPUTATIONAL EVENT FOR WHICH ANY POLICY, CHOSEN IN ADVANCE TO DEFINE ITS RESULT IN ALL CIRCUMSTANCES, WILL OCCASIONALLY PRODUCE A RESULT TO WHICH SOMEBODY MIGHT REASONABLY TAKE EXCEPTION.

****

e.g.: 0.0/0.0 , SQRT(-3) , READ PAST END OF FILE , ...

. . . .

SOME EXCEPTIONS SHOULD NOT BE EXCEPTIONAL AT ALL:

e.g.: $0.0^{0.0} = 1.0$ and

$\cos(1\ 00000\ 00000\ 00000\ 00000.0) = 0.7639...$

have been declared EXCEPTIONAL ARBITRARILY !

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

EXCEPTIONS ARE NOT ERRORS

...

UNLESS THEY ARE HANDLED BADLY.

WHAT FOLLOWS ARE THREE EXAMPLES INTENDED TO ILLUSTRATE WHY EXCEPTIONS SHOULD NOT ALWAYS BE TREATED AS ERRORS.

John Milton's observation (1608-74) :

They also serve who only stand and wait.

- from Sonnet XIX "On His Blindness"

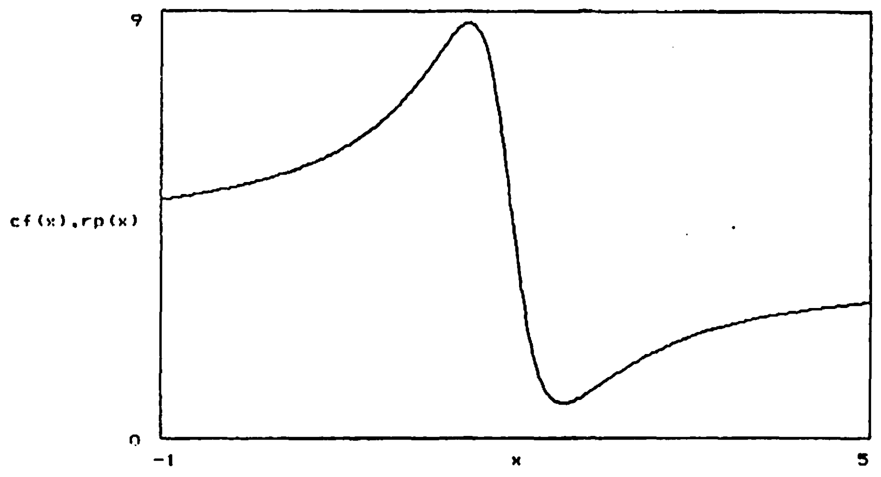
Despite low duty-cycles, some components deserve a place in computers so that they may respond quickly to rare but valuable opportunities. A divider is such a component.

$x := -1, -.98 \,..\, 5$

Here are two ways to express the same rational function:

$$cf(x) := 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}$$

$$rp(x) := \frac{622 - x\cdot(751 - x\cdot(324 - x\cdot(59 - 4\cdot x)))}{112 - x\cdot(151 - x\cdot(72 - x\cdot(14 - x)))}$$



The coincidence of the graphs obtained by plotting both expressions confirms that they represent the same function, though they treat Roundoff, Overflow and Division-by-Zero differently.

For example, ...

$cf(1) = \infty$	$cf(2) = \infty$	$cf(3) = \infty$	$cf(4) = \infty$
singularity	singularity	singularity	singularity
$rp(1) = 7$	$rp(2) = 4$	$rp(3) = 1.6$	$rp(4) = 2.5$

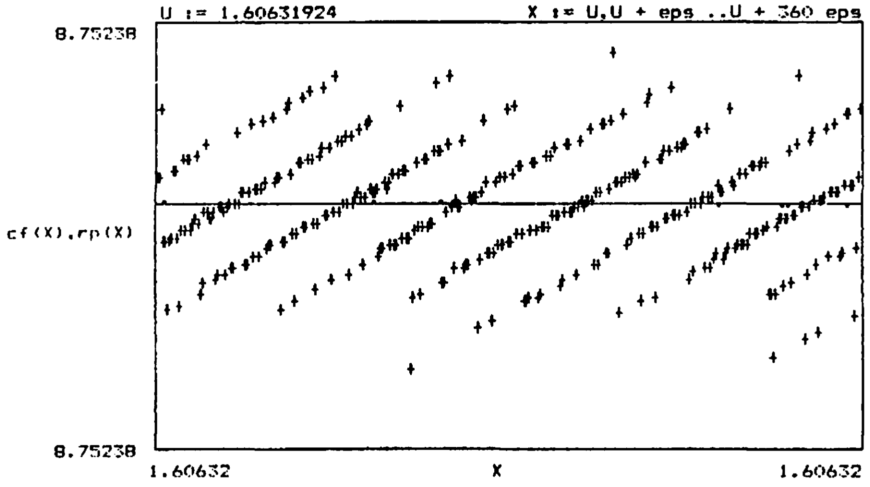
Division-by-Zero cannot happen to rp(x) ; and it would be harmless in cf(x) too if the ∞ supplied by the hardware (it has an INTEL 80x87 that conforms to IEEE standard 754) were used as its designers intended. For instance, computing cf(3) would then produce correctly

$2/(x-3) = \infty$, $x-2-\infty = -\infty$, $10/\infty = 0$, $x - 7 - 0 = -4$, ...

$eps := 0.5^{52}$

$$cf(x) := 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}$$

$$rp(x) := \frac{622 - x\cdot(751 - x\cdot(324 - x\cdot(59 - 4\cdot x)))}{112 - x\cdot(151 - x\cdot(72 - x\cdot(14 - x)))}$$



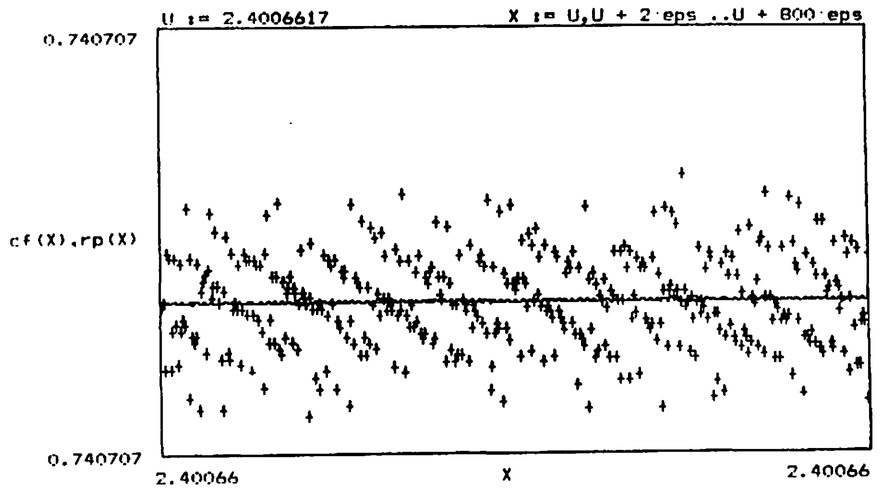
The nearly smooth graph ---- belongs to cf(x) ; the ragged graph of + 's belongs to rp(x) . Every point on each graph has been plotted to show not only how much worse roundoff affects rp(x) than cf(x) but also that roundoff is not nearly so random as some people think.

The next example illustrates that cf(x) is invulnerable to Overflow but rp(x) is not :

$cf\left[10^{77}\right] = 4$ correctly.

$rp\left[10^{77}\right] = \bullet\bullet\bullet$ $\boxed{\text{overflow}}$

The next graphs are included just to show that the previous one was not a fluke. They use different ranges of values for x .

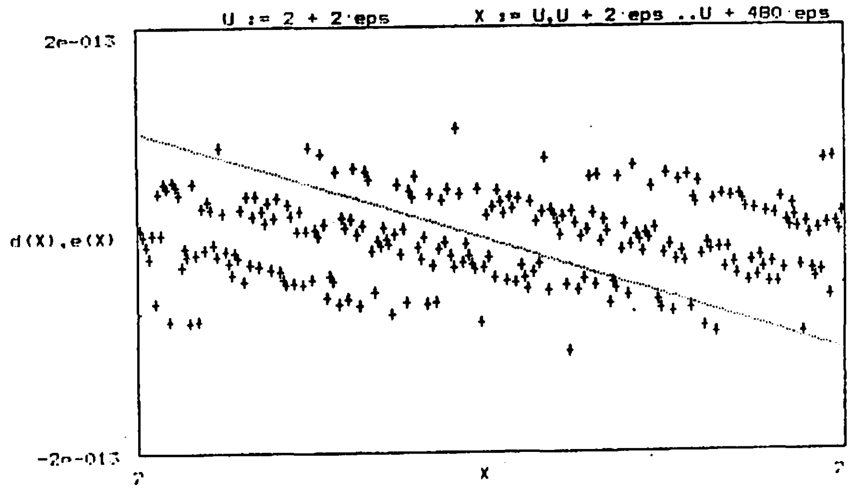


V := 2 + 240 eps

$e(x) := cf(x) - rp(x)$

$$d(x) := \frac{cf(x) - cf(V)}{10}$$

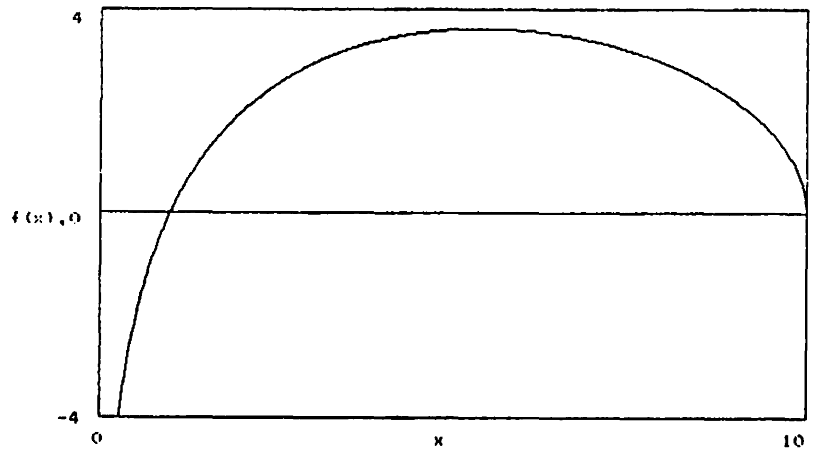
The next graph shows how roundoff obscures rp(x) , but not cf(x) , by about 24 times as much as that function changes when x changes by one unit in its last place for values x slightly bigger than 2 ; for most other values of x roundoff in rp is much worse than this.



$x := \frac{1}{64}, \frac{1}{32} .. 10$ $\quad$ $TOL := 10^{-9}$

Here is an example of a simple equation $f(x) = 0$ that can be hard for conventional zero-finding software to solve for x because the search may well escape from the domain on which $f(x)$ is defined, and then the underlying computer system may abort the program.

$f(x) := \ln(x) \cdot \sqrt{10 - x}$



Starting guess: $X := 2$ $\quad$ root(f(X),X) = 1

$X := 4$ $\quad$ root(f(X),X) = ■ ■ $\boxed{\text{not converging}}$

$X := 9.999$ $\quad$ root(f(X),X) = ■ $\boxed{\text{not converging}}$

$X := 10$ $\quad$ root(f(X),X) = 10

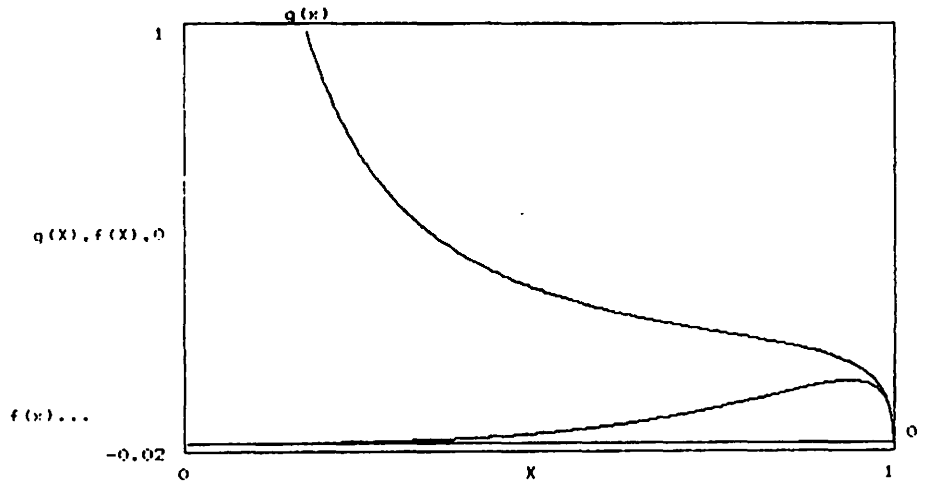
This computer system (MathCAD on an IBM PC) resorts to complex arithmetic whenever it has to compute square roots or logarithms of negative numbers, but that does not solve the problem either because discontinuities in those functions' principal values upset the logic of the zero-finder.

$X := \frac{1}{256}, \frac{1}{128} .. 1$ $\quad$ $TOL := 10^{-14}$

LOOKING FOR AN ELUSIVE POSITIVE ZERO

We seek a positive zero of $f(x) := \tan(x) - \operatorname{asin}(x)$ if it exists.

Since $f(x)$ has an unwanted triple zero at $x = 0$, we remove it by constructing $g(x) := x^{-4} \cdot f(x)$ and looking for its positive zero.



First guess: $X := .1$ $\quad$ root(f(X),X) = ■ ■ $\boxed{\text{singularity}}$

root(g(X),X) = ■ ■ $\boxed{\text{underflow}}$

Another guess: $X := .9$ $\quad$ root(f(X),X) = 0.000034452027 ... = 0 very nearly

root(g(X),X) = ■ $\boxed{\text{underflow}}$

Another guess: $X := .995$ $\quad$ root(f(X),X) = ■ ■ $\boxed{\text{not converging}}$

root(g(X),X) = ■ ■ $\boxed{\text{not converging}}$

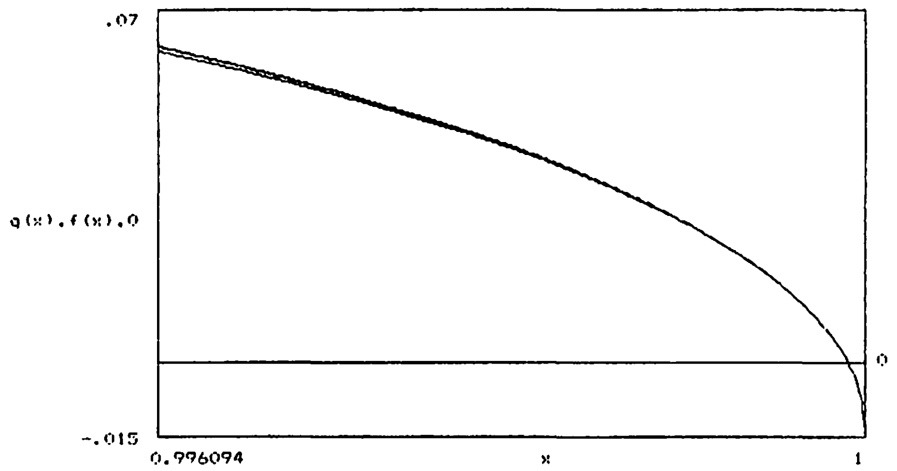
MathCAD's built-in zero-finder seems unable to find any strictly positive zero x near 1 .

One might plausibly conclude that $f(x)$ has no positive zero, but ...

$x := \frac{255}{256}, \frac{255}{256} + \frac{1}{65536} .. 1$

... let's expand the lower right corner of the previous graph:

Upper curve: $g(x) := x^{-4} \cdot f(x)$ Lower curve: $f(x) := \tan(x) - \operatorname{asin}(x)$



First guess:

```
    X := .9999
                 root(f(X),X) = 0.999906012413      A POSITIVE ZERO
                                                    DOES EXIST AFTER
                 root(g(X),X) = 0.999906012413      ALL.
```

Another:

```
    X := 1
                 root(f(X),X) = 0.999906012413 - 1.257087904161·10^-18 i

                 root(g(X),X) = 0.999906012413 - 6.77260495771·10^-19 i
```

Imaginary parts are spurious artifacts of the unnecessary use of complex arithmetic.

To find the positive zero, we had to give the zero-finder first guesses that matched that zero to four sig. dec.! This is not so much a flaw in the zero-finder as a defect in the way exceptions like asin(±1) are handled. If all such exceptions could be handled (at the programmer's request) in the spirit of IEEE standards 754/854, zero finders and other search programs would work more smoothly, as do those on recent HP calculators, without imposing a nuisance upon software users.

IF YOU WOULD HANDLE EXCEPTIONS WELL, YOU MUST DEAL WITH ...

NAMES: WE SHOULD ALL USE THE SAME NAMES FOR THE SAME THINGS IF WE WISH TO SHARE EACH OTHER'S PROGRAMS.

FLAGS: THESE TELL A PROGRAM WHICH EXCEPTIONS HAVE OCCURRED. They cannot be numerous lest we forget their names.

MODES: A PROGRAM CHANGES THEM TO CHANGE THE WAY ITS EXCEPTIONS WILL BE HANDLED. Very few are useful.

SCOPE: CONCERNED WITH LINGUISTIC CONVENTIONS FOR *HIDING* THE SETTING, SENSING, CLEARING, SAVING AND RESTORING OF FLAGS AND MODES .

RETROSPECTIVE DIAGNOSTICS: THESE PROVIDE A WAY FOR THE USER OF A PROGRAM TO COPE WITH THE POSSIBLY MISHANDLED EXCEPTIONS, AND ARE PROVABLY INDISPENSABLE FOR CORRECT EXCEPTION HANDLING.

We have been working on good ways to deal with these issues that are, as nearly as possible,

INDEPENDENT OF HARDWARE
INDEPENDENT OF LANGUAGE
INDEPENDENT OF COMPILERS.

BREAKING THE VICIOUS CIRCLE

PORTABILITY:
NO RESPONSIBLE APPLICATIONS PROGRAMMER WILL EXPLOIT UNFAMILIAR MECHANISMS TO HANDLE EXCEPTIONS UNLESS REASONABLY ASSURED OF THEIR (IMPENDING) UNIVERSALITY.

INDIFFERENCE:
NO RESPONSIBLE COMPILER-WRITER WILL IMPLEMENT UNFAMILIAR MECHANISMS TO HANDLE EXCEPTIONS UNLESS ASSURED THAT CUSTOMERS DESIRE THEM ENOUGH TO PAY FOR THEM.

MARKETING:
HARDWARE DESIGNERS TEND TO OMIT FEATURES THAT COMPILER WRITERS TEND NOT TO USE.

" WHEN HIP-DEEP IN ALLIGATORS, WHO CAN ENTERTAIN PROPOSALS TO JOIN IN DRAINING THE SWAMP ? "

A consortium of interested parties in the computer industry is being formed to agree upon exception-handling well enough to implement would-be universal capabilities. W. K.

USE IT OR LOSE IT

IEEE 754 DIRECTED ROUNDING:
Round to +INFINITY
Round to -INFINITY
... at the programmer's option.

Major application ... INTERVAL ARITHMETIC

... for Roundoff Error Bounds ...

... for ABSOLUTE SENSITIVITY ANALYSIS

... for SEARCHES for ROOTS of systems of equations
for OPTIMA

NO SUPPORT IN COMPILERS ===> NO USE TO APPLICATIONS PROGRAMMERS

LANGUAGE DESIGNERS AND IMPLEMENTORS ARE TOO OFTEN

PETTY TYRANTS

EXERCIZING ENORMOUS INFLUENCE, IF NOT CONTROL, OVER

HARDWARE ARCHITECTS AND APPLICATIONS PROGRAMMERS,

YET SELDOM MORE KNOWLEDGEABLE THAN EITHER.

W. Kahan