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

by Prof. W. Kahan, EE&CS Dept.,
University of California, Berkeley CA 94720

July 8, 1989
Retypeset by David Bindel, Apr. 21, 2001

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.

1 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.

2 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. This 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.

3 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

$$s = w_1 * z_1 + w_2 * z_2 + \ldots + w_n * z_n \ ,$$

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 fragment above into

$q := 0$; $r := 0$;
for $k = 1$ to n do
{ $q := (u_k * x_k - v_k * y_k) + q$;
$r := (v_k * x_k + u_k * y_k) + r$ } ;

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. (See the Appendix to see how.) 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, no demand for the Weitek 1167 or 3167 would have arisen; 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.

4 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.

5 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 $0 \leq t < 8$, cell number t in the 80X87 associates with cell 8B + TOP + ((t - (TOP mod 8)) mod 8) in the extension. The figure shows associated cells when TOP = 2:

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

```
KKKKK          #1 kkkkk ST(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 above will help to minimize memory traffic. Note that the cost of stack over/underflow will never be negligible, so an optimizing compiler must not push onto the stack indiscriminately things better left in memory or in a cache, lest stack underflow afflict every reference to anything far from the stack's top.

6 Is All This Worth The Bother Now?

Compatibility with old software is how 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, competitors' faster versions of the 80x87 will convey too little 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. Intel's competitors face a paradox; they have to promote the spread of software that will run Intel's chips faster in order to create an environment in which

their own chips can run even faster. That software will come into existence only if compilers evolve along with 80x86/7 hardware towards ever better performance.

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?

To buttress that strength, I propose that a package of drivers be created to hide from users not only stack over/underflows but all discrepancies among different members of the 80x87 family. For example, the 80387's handling of Denormals should be imitated among earlier 80x87s as much as possible. A library to handle other exceptions like Overflow and 0/0 in reasonable ways can be part of the package too. While enhancing the 8087s and 80287s in older equipment, this package would create a milieu for new software that runs correctly with 80x87s and fast with 80X87s. Individuals interested in aiding the creation and dissemination of such a package may write to the author at the address above.

7 Appendix

The examples below are offered as a challenge to compiler writers.

The loop that accumulates the complex scalar product $s := w * z + s$ for $s = q + ir$, $w = u + iv$ and $z = x + iy$ performs in each pass the following 13 operations upon the floating-point stack:

```
... start with     q, r                 in the stack, and then
    load           x, q, r
    push           x, x, q, r       ... the only extra operation
    load           u, x, x,   q, r
    mult           u, x, u*x, q, r
    load           v, u, x,   u*x, q, r
    mult           v, u, v*x, u*x, q, r
    load           y, v, u,   v*x, u*x, q, r    ... 7 items deep
    mult           y, v, u*y, v*x, u*x, q, r
    mult           v*y, u*y,  v*x, u*x, q, r
    subt & pop     u*y, v*x,  u*x-v*y, q, r
    add            v*x+u*y,   u*x-v*y, q, r
    add & pop      u*x-v*y,   q, r := (v*x+u*y)+r
    add            q := (u*x-v*y)+q, r .
```

The next example uses recurrences to compute a polynomial

$$p = a_n + a_{n-1}x + a_{n-2}x^2 + \ldots + a_0x^n$$

and its first two derivatives $q = p'$ and $r = p''$ and a bound b for the accumulation of roundoff in p :

$q := r := 0$; $p := a_0$; $b := |p|/2$;
for $k = 1$ to n do
{ $r := x^*r + q$;
$q := x^*q + p$;
$p := x^*p + a_k$;
$b := |x|^*b + |p|$ } ;
$b := (b - |p| + b)/2^{63}$ ... for 64 sig. bits.

These values figure in Laguerre's iteration to find a real zero of p. Unless $|p| \leq 2b$, in which case x is about as close to a zero as can be expected, iteration replaces x by a new

$$x := x - n * p/(q + \mathrm{CopySign}(((n-1)^2 * q^2 - n * (n-1) * p * r)^{1/2}, q)),$$

which should be better. Each pass of the inner loop performs 13 operations, of which 3 quick *push*es are the cost of using a stack instead of flat registers:

```
... start with      p, b, q, r, |x|, x     in the stack, and then
    push            x, p, b, q, r, |x|, x
    mult            x, p, b, q, x*r, |x|, x
    push            q, x, p, b, q, x*r, |x|, x    ... 8 items deep
    add & pop       x, p, b, q, r := x*r+q, |x|, x
    mult & pop      p, b, x*q, r, |x|, x
    add             p, b, q := x*q+p, r, |x|, x
    mult            x*p, b, q, r, |x|, x
    load & add      p := x*p+ak, b, q, r, |x|, x
    push            |x|, p, b, q, r, |x|, x
    mult & pop p,   |x|*b, q, r, |x|, x
    push p, p,      |x|*b, q, r, |x|, x
    abs             |p|, p, |x|*b, q, r, |x|, x
    add & pop       p, b := |x|*b+|p|, q, r, |x|, x .
```