On the Advantages of the 8087's Stack

W. Kahan

Nov. 2, 1990
Retypeset by David Bindel, Apr. 21, 2001

Compared with what? The other options available at the time the 8087 co-processor was designed were these:

Classical Stack Architecture (like Burroughs B5500)
Flat Register-set with Two operands/instruction.

Three operands/instruction could not be accomodated; the host 8086/8088 was designed for Two, and more would not fit into the space available for instruction fields. And even Two operands/ instruction would have been an inconvenient choice to implement for lack of available (uncommitted) op-codes for the shortest and most frequent instructions; see the comment about too few "unique encodings for ESC" on page 60 of *The 8087 Primer* by J. F. Palmer and S. P. Morse (1984, Wiley, New York).

Therefore, the 8087 was destined to have preponderantly One-operand instructions; the designer's task was to make a Virtue of this Necessity.

1 How a Classical Stack would have worked: Badly.

The classical stack architecture's instructions refer to only the topmost two or three cells of an indefinitely long stack, so only the topmost several cells have to be implemented as fast registers in hardware; the rest of the stack's cells can reside in memory. To access cells deeper in the stack, one pushes onto the stack's top a Descriptor that points to the desired cell deep in memory, and then executes an instruction either to overwrite a copy of the deep cell onto its descriptor, or to copy the second stack cell onto the deep cell and then *Pop* the two topmost cells off the stack. This procedure seems simple, but it has two drawbacks.

First, the classical stack mixes descriptors with data. That does no harm when the hardware to manipulate memory addresses is the same as to manipulate data. But the 8087 data is floating-point, utterly different from descriptors (memory addresses) with which only the host 8086/8088 has to deal. Why should we have to duplicate the host's descriptor-handling in the 8087?

Second, the classical stack engages in too much intercourse with memory. Each descriptor has to travel from memory to stack-top, and then data has to travel between the stack-top and whatever memory cell the descriptor points to. Expression evaluation works well on a stack when each subexpression evaluated on the stack's top is consumed immediately, but if a subexpression's use or reuse is much deferred then it must travel to and from memory; and if that subexpression is a "common subexpression" that has to be stored and retrieved later, then its reuse may cost two extra memory references for its descriptor.

Memory management is the bottle-neck in computer architecture. This is especially the case for floating-point data because they are so much wider than addresses, and the 8087's data is very wide indeed, 10 bytes, compared with the paths to memory: 1 byte for the 8088, 2 bytes for the 8086 and 80286, 4 for the 80386 and 80486. Moreover, the speed of arithmetic hardware tends to grow faster with the passage of time than does the speed of memory hardware; over the past decade floating-point hardware has become 10 to 100 times faster, while memory has become only 2 to 4 times faster (but 10 to 100 times bigger). These developments were forseen. Their implications were clear.

The designers of the 8087 had to reject the classical stack architecture because ultimately it would have wasted too much time on intercourse with memory.

2 How Flat Registers would have worked: Not so well.

If a Two-operand instruction set for 8 registers had been implementable for the 8087, it would have been done that way if only to provide compiler writers with something they were used to. They were even more accustomed to fewer registers, say 4 as on the NS 32081 and WE 32106; but when we looked ahead to matrix computations with interleaved memory and a cache, and with their consequent loop-unrolling, we could forsee that 4 registers would be too few. That judgment appears to be shared by others, whose subsequent designs for the Motorola 68081/2, SPARC, MIPS and IBM RS/6000 all have at least 8 floating-point registers.

But a flat register set has its problems too. They arise during calls to function subprograms that pass their arguments by value in registers and return their values in a register. Which other registers must be saved before and restored after the function has done its work? Only the calling program knows which registers it needs and might therefore have to save. Only the called function knows which registers it uses and might therefore have to restore. If the programs are compiled separately, and if the program loader is not allowed to insert and delete instructions in them, then one or the other is going to save and restore some registers unnecessarily from time to time. This means unnecessary memory traffic. The extra cost is significant just when there are many calls to short function subprograms, as is increasingly the case in modern programming languages.

3 How the 8087 Stack could have worked well.

This is the subject of a note I wrote in 1989 (attached). The following commments address these points:

> What can be done with a stack of 8 registers that cannot
> be done with 8 flat registers, and vice-versa?
>
> What did we know that others did not (and still do not)
> that made us think the stack was advantageous?

Of course, the 8087's 8 stack registers can be (and often are) used as a flat register file; the extra cost involved in so doing is a number of FXCH instructions to swap registers with the top of stack where most of the action takes place. Each FXCH costs barely more than a FNOP that does nothing, so the cost of a few of them is minimal. Common subexpressions, whose values must be used more than once during expression evaluation, pose no more nor less of a problem for a stack than for a flat file of registers, though the algorithms for solving that problem differ a little. Fortunately, optimal ways to handle a few common sub-expressions are as easy to find for a stack as for a flat file.

Conversely, software can make a flat register file look like the topmost few cells of a stack, for a price. Within *one* compiled module a compiler, in the course of translating the programmer's source code into the machine's object code, could keep track of register usage and put data and results into the same positions as if they were in the 8087's stack, and could do so at no extra cost. But this cannot be done for *several* separately compiled modules; they have no way to know which registers are in use by other modules unless this knowledge is conveyed through some other registers dedicated to the purpose. In current architectures, which do not index register references the way they index memory references, using such knowledge costs far more memory traffic than could be saved by not having to make the usual arrangements for saving a flat file of registers. In short, ...

> Stack registers can simulate a flat file so much easier than a flat file can simulate a stack that building a stack is tantamount to offering users a choice that building a flat file into the 8087 would have denied them.

A stack architecture must traffic with memory whenever stack depth grows or shrinks by more cells than can reside in the registers on the chip. These stack over/underflows entail saving and restoring the contents of the last few registers. This traffic resembles superficially the above-mentioned saving and restoring of flat registers that the stack architecture was designed to avoid. The difference resides in the frequency of stack over/underflows; a floating-point stack's depth fluctuates relatively shallowly, so a floating-point stack

over/underflows very infrequently compared with a classical stack. Consequently, the cost of handling stack over/underflow for the 8087, though horrible when it occurs, is incurred so infrequently that it rarely matters.

When the 8087 was designed, I knew that stack over/underflow was an issue of more aesthetic than practical importance. I still regret that the 8087's stack implementation was not quite so neat as my original intention described in the accompanying note. If the original design had been realized, compilers to-day would use the 8087 and its descendants more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations. Back in 1979 we did not appreciate how difficult it would become, both technicaly and commercially, to disseminate appropriate stack over/underflow handling software; far less did we appreciate how clumsily the early compilers would handle stack over/underflow, nor how long they would take to do better.

Nowadays stack over/underflow is practically a dead issue. Most compilers avoid it by refraining from using all 8 of the stack registers except in bursts of code with no function calls, and by conventions that always save a few registers on calls so that a functon that (like the vast majority) uses only a few stack registers need save no more. However, there are still situations such conventions handle inefficiently though the original design would fare well. Here is an example defined recursively:

Real Function P(Unsigned Integer Value n , Real Value x) :=
if $n = 0$ then (if $(0 < x)$ and $(x \leq 1)$ then 1 else 0)
else $(P(n-1, 2*x-2) + P(n-1, 2*x)) * 0.5 + P(n-1, 2*x-1)$.

The cost of evaluating $P(n, x)$ grows like 3^n, proportional to mostly the cost of overhead (register saving and restoring during function calls). The definition of $P(0, x)$ has been chosen here to permit easy and fast nonrecursive calculation of $P(n, x)$ for checking purposes:

$P(n, x) :=$ if $x < 0$ or $2 < x$ then 0 else
if $x > 1$ then $1 - \text{Ceil}(2^n(x-1))/2^n$
else $\text{Ceil}(2^n x)/2^n$.

Were $P(0, x)$ defined a little differently, the recursive definition might be the only reasonable way to compute $P(n, x)$.

This simple example illustrates a style of computation, with very many calls to very short procedures, that is promoted by current trends in programming languages. The example is typical of the new "Wavelet" techniques that are becoming justifiably popular for computational fluid dynamics, signal processing, and quite possibly for simulations of light and shadow in visually complex scenes. For those reasons, ...

> I recommend urgently that the stack architecture of the 8087's descendants be brought into line with the original intent despite that most computer architects, most compiler writers, most marketing specialists, and most customers have shown no interest in the matter.

Why should they care about something that almost never happens? Because we could easily change "almost never" into "never".

If stack over/underflow were handled well by apt hardware and/or by appropriate software "drivers" loaded automatically by the operating system, stack over/underflow would become invisible to compiler writers as well as to applications programmers and their clients. The operating system would be involved only because it would be responsible for configuring the software to match the hardware, for managing a small 80x87 stack overflow area in memory, and for task-switching it. The rest of us could ignore the issue of stack over/underflow confident that the available floating-point registers were being exploited about as well as possible regardless of instruction-set architecture.