My, what strange NOPs you have!

devblogs.microsoft.com/oldnewthing/20110112-00

January 12, 2011

Raymond Chen

While cleaning up my office, I ran across some old documents which reminded me that there are a lot of weird NOP instructions in Windows 95.

Certain early versions of the 80386 processor (manufactured prior to 1987) are known as *B1 stepping* chips. These early versions of the 80386 had some obscure bugs that affected Windows. For example, if the instruction following a string operation (such as `movs`) uses opposite-sized addresses from that in the string instruction (for example, if you performed a `movs es:[edi], ds:[esi]` followed by a `mov ax, [bx]`) or if the following instruction accessed an opposite-sized stack (for example, if you performed a `movs es:[edi], ds:[esi]` on a 16-bit stack, and the next instruction was a `push`), then the `movs` instruction would not operate correctly. There were quite a few of these tiny little "if all the stars line up exactly right" chip bugs.

Most of the chip bugs only affected mixed 32-bit and 16-bit operations, so if you were running pure 16-bit code or pure 32-bit code, you were unlikely to encounter any of them. And since Windows 3.1 did very little mixed-bitness programming (user-mode code was all-16-bit and kernel-mode code was all-32-bit), these defects didn't really affect Windows 3.1.

Windows 95, on the other hand, contained a lot of mixed-bitness code since it was the transitional operating system that brought people using Windows out of the 16-bit world into the 32-bit world. As a result, code sequences that tripped over these little chip bugs turned up not infrequently.

An executive decision had to be made whether to continue supporting these old chips or whether to abandon them. A preliminary market analysis of potential customers showed that there were enough computers running old 80386 chips to be worth making the extra effort to support them.

Everybody who wrote assembly language code was alerted to the various code sequences that would cause problems on a B1 stepping, so that they wouldn't generate those code sequences themselves, and so they could be on the lookout for existing code that might have problems. To supplement the manual scan, I wrote a program that studied all the Windows 95 binaries

trying to find these troublesome code sequences. When it brought one to my attention, I studied the offending code, and if I agreed with the program's assessment, I notified the developer who was responsible for the component in question.

In nearly all cases, the troublesome code sequences could be fixed by judicious insertion of `NOP` statements. If the problem was caused by "instruction of type X followed by instruction of type Y", then you can just insert a `NOP` between the two instructions to "break up the party" and sidestep the problem. Sometimes, the standard `NOP` would end up classified as an instruction of type Y, so you had to insert a *special kind of* `NOP`, one that was not of type Y.

For example, here's one code sequence from a function which does color format conversion:

```
push    si          ; borrow si temporarily
; build second 4 pixels
movzx   si, bl
mov     ax, redTable[si]
movzx   si, cl
or      ax, blueTable[si]
movzx   si, dl
or      ax, greenTable[si]
shl     eax, 16     ; move pixels to high word
; build first 4 pixels
movzx   si, bh
mov     ax, redTable[si]
movzx   si, ch
or      ax, blueTable[si]
movzx   si, dh
or      ax, greenTable[si]
pop     si
stosd   es:[edi]    ; store 8 pixels
db      67h, 90h    ; 32-bit NOP fixes stos (B1 stepping)
dec     wXE
```

Note that we couldn't use just any old `NOP`; we had to use a `NOP` with a 32-bit address override prefix. That's right, this isn't just a regular `NOP`; this is a *32-bit* `NOP`.

From a B1 stepping-readiness standpoint, the folks who wrote in C had a little of the good news/bad news thing going. The good news is that the compiler did the code generation and you didn't need to worry about it. The bad news is that you also were dependent on the compiler writers to have taught their code generator how to avoid these B1 stepping pitfalls, and some of them were quite subtle. (For example, there was one bug that manifested itself in incorrect instruction decoding if a conditional branch instruction had just the right sequence of taken/not-taken history, and the branch instruction was followed immediately by a selector load, *and* one of the first two instructions at the destination of the branch was itself a jump, call, or return. The easy workaround: Insert a `NOP` between the branch and the selector load.)

On the other hand, some quirks of the B1 stepping were easy to sidestep. For example, the B1 stepping did not support virtual memory in the first 64KB of memory. Fine, don't use virtual memory there. If virtual memory was enabled, if a certain race condition was encountered inside the hardware prefetch, and if you executed a floating point coprocessor instruction that accessed memory at an address in the range 0x800000F8 through 0x800000FF, then the CPU would end up reading from addresses 0x000000F8 through 0x000000FF instead. This one was easy to work around: Never allocate valid memory at 0x80000xxx. Another reason for the no man's land in the address space near the 2GB boundary.

I happened to have an old computer with a B1 stepping in my office. It ran slowly, but it did run. I think the test team "re-appropriated" the computer for their labs so they could verify that Windows 95 still ran correctly on a computer with a B1 stepping CPU.

Late in the product cycle (after Final Beta), upper management reversed their earlier decision and decide not to support the B1 chip after all. Maybe the testers were finding too many bugs where other subtle B1 stepping bugs were being triggered. Maybe the cost of having to keep an eye on all the source code (and training/retraining all the developers to be aware of B1 issues) exceeded the benefit of supporting a shrinking customer base. For whatever reason, B1 stepping support was pulled, and customers with one of these older chips got an error message when they tried to install Windows 95. And just to make it easier for the product support people to recognize this failure, the error code for the error message was Error B1.

Raymond Chen

Follow