What is the historical reason for MulDiv(1, -0x80000000, -0x80000000) returning 2?

devblogs.microsoft.com/oldnewthing/20120514-00

May 14, 2012

Raymond Chen

Commenter rs asks, "Why does Windows (historically) return 2 for `MulDiv(1, -0x80000000, -0x80000000)` while Wine returns zero?"

The `MulDiv` function multiplies the first two parameters and divides by the third. Therefore, the mathematically correct answer for `MulDiv(1, -0x80000000, -0x80000000)` is 1, because $a \times b \div b = a$ for all nonzero b.

So both Windows and Wine get it wrong. I don't know why Wine gets it wrong, but I dug through the archives to figure out what happened to Windows.

First, some background. What's the point of the `MulDiv` function anyway?

Back in the days of 16-bit Windows, floating point was very expensive. Most people did not have math coprocessors, so floating point was performed via software emulation. And the software emulation was slow. First, you issued a floating point operation on the assumption that you had a float point coprocessor. If you didn't, then a *coprocessor not available* exception was raised. This exception handler had a lot of work to do.

It decoded the instruction that caused the exception and then emulated the operation. For example, if the bytes at the point of the exception were `d9 45 08`, the exception handler would have to figure out that the instruction was `fld dword ptr ds:[di][8]`. It then had to simulate the operation of that instruction. In this case, it would retrieve the caller's `di` register, add 8 to that value, load four bytes from that address (relative to the caller's `ds` register), expand them from 32-bit floating point to 80-bit floating point, and push them onto a pretend floating point stack. Then it advanced the instruction pointer three bytes and resumed execution.

This took an instruction that with a coprocessor would take around 40 cycles (already slow) and ballooned its total execution time to a few hundred, probably thousand cycles. (I didn't bother counting. Those who are offended by this horrific laziness on my part can apply for a refund.)

It was in this sort of floating point-hostile environment that Windows was originally developed. As a result, Windows has historically avoided using floating point and preferred to use integers. And one of the things you often have to do with integers is scale them by some ratio. For example, a horizontal dialog unit is ¼ of the average character width, and a vertical dialog unit is 1/8 of the average character height. If you have a value of, say, 15 horizontal dlu, the corresponding number of pixels is 15 × average character width ÷ 4. This multiply-then-divide operation is quite common, and that's the model that the `MulDiv` function is designed to help out with.

In particular, `MulDiv` took care of three things that a simple $a \times b \div c$ didn't. (And remember, we're in 16-bit Windows, so a, b and c are all 16-bit signed values.)

- The intermediate product $a \times b$ was computed as a 32-bit value, thereby avoiding overflow.
- The result was *rounded* to the nearest integer instead of truncated toward zero
- If $c = 0$ or if the result did not fit in a signed 16-bit integer, it returned *INT_MAX* or *INT_MIN* as appropriate.

The `MulDiv` function was written in assembly language, as was most of GDI at the time. Oh right, the `MulDiv` function was exported by GDI in 16-bit Windows. Why? Probably because they were the people who needed the function first, so they ended up writing it.

Anyway, after I studied the assembly language for the function, I found the bug. A `shr` instruction was accidentally coded as `sar`. The problem manifests itself only for the denominator `-0x8000`, because that's the only one whose absolute value has the high bit set.

The purpose of the `sar` instruction was to divide the denominator by two, so it can get the appropriate rounding behavior when there is a remainder. Reverse-compiling back into C, the function goes like this:

```
int16 MulDiv(int16 a, int16 b, int16 c)
{
 int16 sign = a ^ b ^ c; // sign of result
 // make everything positive; we will apply sign at the end
 if (a < 0) a = -a;
 if (b < 0) b = -b;
 if (c < 0) c = -c;
 //  add half the denominator to get rounding behavior
 uint32 prod = UInt16x16To32(a, b) + c / 2;
 if (HIWORD(prod) >= c) goto overflow;
 int16 result = UInt32Div16To16(prod, c);
 if (result < 0) goto overflow;
 if (sign < 0) result = -result;
 return result;
overflow:
 return sign < 0 ? INT_MIN : INT_MAX;
}
```

Given that I've already told you where the bug is, it should be pretty easy to spot in the code above.

Anyway, when this assembly language function was ported to Win32, it was ported as, well, an assembly language function. And the port was so successful, it even preserved (probably by accident) the sign extension bug.

Mind you, it's a bug with amazing seniority.

Raymond Chen

Follow