Creating double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result

devblogs.microsoft.com/oldnewthing/20141208-00

December 8, 2014

Raymond Chen

Suppose you want to multiply two double-word values producing a quad-word result, but your processor supports only single-word multiplication with a double-word result. For concreteness, let's say that your processor supports 32 × 32 → 64 multiplication and you want to implement 64 × 64 → 128 multiplication. (Sound like any processor you know?)

Oh boy, let's do some high school algebra. Let's start with unsigned multiplication.

Let $x = A \times 2^{32} + B$ and $y = C \times 2^{32} + D$, where A, B, C, and D are all in the range 0 … $2^{32} - 1$.

$$x \times y = AC \times 2^{64} + (AD + BC) \times 2^{32} + BD$$

$$= \underbrace{AC \times 2^{64} + BD}_{\text{provisional result}} + \underbrace{(AD + BC) \times 2^{32}}_{\text{cross-terms}}$$

Each of the multiplications (not counting the power-of-two multiplications) is a 32 × 32 → 64 multiplication, so they are something we have as a building block. And the basic implementation is simply to perform the four multiplications and add the pieces together. But if you have SSE, you can perform two multiplies in a single instruction.

```
// Prepare our source registers
movq xmm0, x         // xmm0 = { 0, 0, A, B } = { *, *, A, B }
movq xmm1, y         // xmm1 = { 0, 0, C, D } = { *, *, C, D }
punpckldq xmm0, xmm0 // xmm0 = { A, A, B, B } = { *, A, *, B }
punpckldq xmm1, xmm1 // xmm1 = { C, C, D, D } = { *, C, *, D }
pshufd xmm2, xmm1, _MM_SHUFFLE(2, 0, 3, 1)
                     // xmm2 = { D, D, C, C } = { *, D, *, C }
```

The `PMULUDQ` instruction multiplies 32-bit lanes 0 and 2 of its source and destination registers, producing 64-bit results. The values in lanes 1 and 3 do not participate in the multiplication, so it doesn't matter what we put there. It so happens that the `PUNPCKLDQ` instruction duplicates the value, but we really don't care. I used `*` to represent a don't-care value.

```
pmuludq xmm1, xmm0 // xmm1 = { AC, BD } // provisional result
pmuludq xmm2, xmm0 // xmm2 = { AD, BC } // cross-terms
```

In two `PMULUDQ` instructions, we created the provisional result and the cross-terms. Now we just need to add the cross-terms to the provisional result. Unfortunately, SSE does not have a 128-bit addition (or at least SSE2 doesn't; who knows what they'll add in the future), so we need to do that the old-fashioned way.

```
movdqa result, xmm1
movdqa crossterms, xmm2
mov    eax, crossterms[0]
mov    edx, crossterms[4] // edx:eax = BC
add    result[4], eax
adc    result[8], edx
adc    result[12], 0      // add the first cross-term
mov    eax, crossterms[8]
mov    edx, crossterms[12] // edx:eax = AD
add    result[4], eax
adc    result[8], edx
adc    result[12], 0      // add the second cross-term
```

There we go, a 64 × 64 → 128 multiply constructed from 32 × 32 → 64 multiplies.

Exercise: Why didn't I use the `rax` register to perform the 64-bit addition? (This is sort of a trick question.)

That calculates an unsigned multiplication, but how do we do a signed multiplication? Let's work modulo 2^{128} so that signed and unsigned multiplication are equivalent. This means that we need to expand x and y to 128-bit values X and Y.

Let s = the sign bit of x expanded to a 64-bit value, and similarly t = the sign bit of y expanded to a 64-bit value. In other words, s is `0xFFFFFFFF`FFFFFFFF` if $x < 0$ and zero if $x \geq 0$.

The 128-bit values being multiplied are

$$X = s \times 2^{64} + x$$
$$Y = t \times 2^{64} + y$$

The product is therefore

$$X \times Y = st \times 2^{128} + (sy + tx) \times 2^{64} + xy$$

zero adjustment unsigned product

The first term is zero because it overflows the 128-bit result. That leaves the second term as the adjustment, which simplifies to “If $x < 0$ then subtract y from the high 64 bits; if $y < 0$ then subtract x from the high 64 bits.”

```
if (x < 0) result.m128i_u64[1] -= y;
if (y < 0) result.m128i_u64[1] -= x;
```

If we were still playing with SSE, we could compute this as follows:

```
movdqa xmm0, result   // xmm0 = { high, low }
movq   xmm1, x        // xmm1 = { 0, x }
movq   xmm2, y        // xmm2 = { 0, y }
pshufd xmm3, xmm1, _MM_SHUFFLE(1, 1, 3, 2) // xmm3 = { xhi, xhi, 0, 0 }
pshufd xmm1, xmm1, _MM_SHUFFLE(1, 0, 3, 2) // xmm1 = { x, 0 }
pshufd xmm4, xmm2, _MM_SHUFFLE(1, 1, 3, 2) // xmm4 = { yhi, yhi, 0, 0 }
pshufd xmm2, xmm2, _MM_SHUFFLE(1, 0, 3, 2) // xmm2 = { y, 0 }
psrad  xmm3, 31       // xmm3 = { s, s, 0, 0 } = { s, 0 }
psrad  xmm4, 31       // xmm4 = { t, t, 0, 0 } = { t, 0 }
pand   xmm3, xmm2     // xmm3 = { x < 0 ? y : 0, 0 }
pand   xmm4, xmm1     // xmm4 = { y < 0 ? x : 0, 0 }
psubq  xmm0, xmm3     // first adjustment
psubq  xmm0, xmm4     // second adjustment
movdqa result, xmm0   // update result
```

The code is a bit strange because SSE2 doesn’t have a full set of 64-bit integer opcodes. We would have liked to have used a `psraq` instruction to fill a 64-bit field with a sign bit. But there is no such instruction, so instead we duplicate the 64-bit sign bit into two 32-bit sign bits (one in lane 2 and one in lane 3) and then fill the lanes with that bit using `psrad`.

Raymond Chen

Follow