Creating double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result using intrinsics (part 3)

devblogs.microsoft.com/oldnewthing/20171215-00

December 15, 2017

Raymond Chen

Last time, we converted our original assembly language code for creating double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result to C++ code with intrinsics working entirely in registers, thereby making the function eligible for leaf function optimizations.

Our last step is adding support for signed multiplication. This is a straightforward translation of the original assembly language into intrinsics.

```
__m128i Multiply64x64To128(int64_t x, int64_t y)
{
    auto x128 = _mm_loadl_epi64((__m128i*) &x);
    auto term1 = _mm_unpacklo_epi32(x128, x128);

    auto y128 = _mm_loadl_epi64((__m128i*) &y);
    auto term2 = _mm_unpacklo_epi32(y128, y128);

    auto flip2 = _mm_shuffle_epi32(term2, _MM_SHUFFLE(1, 0, 3, 2));

    auto result = _mm_mul_epu32(term1, term2);
    auto crossterms = _mm_mul_epu32(term1, flip2);

    // Now apply the cross-terms to the provisional result
    unsigned temp;

    auto result1 = _mm_srli_si128(result, 4);
    auto carry = _addcarry_u32(0,
                               _mm_cvtsi128_si32(result1),
                               _mm_cvtsi128_si32(crossterms),
                               &temp);
    result1 = _mm_cvtsi32_si128(temp);

    auto result2 = _mm_srli_si128(result, 8);
    crossterms = _mm_srli_si128(crossterms, 4);
    carry = _addcarry_u32(carry,
                          _mm_cvtsi128_si32(result2),
                          _mm_cvtsi128_si32(crossterms),
                          &temp);
    result2 = _mm_cvtsi32_si128(temp);

    auto result3 = _mm_srli_si128(result, 12);
    _addcarry_u32(carry,
                  _mm_cvtsi128_si32(result3),
                  0,
                  &temp);
    result3 = _mm_cvtsi32_si128(temp);

    crossterms = _mm_srli_si128(crossterms, 4);
    carry = _addcarry_u32(0,
                          _mm_cvtsi128_si32(result1),
                          _mm_cvtsi128_si32(crossterms),
                          &temp);
    result1 = _mm_cvtsi32_si128(temp);

    crossterms = _mm_srli_si128(crossterms, 4);
    carry = _addcarry_u32(carry,
                          _mm_cvtsi128_si32(result2),
                          _mm_cvtsi128_si32(crossterms),
                          &temp);
    result2 = _mm_cvtsi32_si128(temp);
    _addcarry_u32(carry,
                  _mm_cvtsi128_si32(result3),
                  0,
                  &temp);
    result3 = _mm_cvtsi32_si128(temp);

    result = _mm_unpacklo_epi64(
       _mm_unpacklo_epi32(result, result1),
       _mm_unpacklo_epi32(result2, result3));

    // Apply sign adjustment.
    __m128i xsign = _mm_shuffle_epi32(x128, _MM_SHUFFLE(1, 1, 3, 2));
    xsign = _mm_srai_epi32(xsign, 31);
    __m128i ysign = _mm_shuffle_epi32(y128, _MM_SHUFFLE(1, 1, 3, 2));
    ysign = _mm_srai_epi32(ysign, 31);
    __m128i xshift64 = _mm_shuffle_epi32(x128, _MM_SHUFFLE(1, 0, 3, 2));
    __m128i yshift64 = _mm_shuffle_epi32(x128, _MM_SHUFFLE(1, 0, 3, 2));
    __m128i xadjust = _mm_and_si128(xsign, xshift64);
    __m128i yadjust = _mm_and_si128(ysign, yshift64);
    result = _mm_sub_epi64(result, xadjust);
    result = _mm_sub_epi64(result, yadjust);

    return result;
}
```

Each of the new statements translates into a single instruction. There are still enough of XMM registers available so that we can still do all the work in registers. (And if you look at the resulting assembly, you'll see that the compiler reordered the operations, presumably for optimization purposes.)

So there you have it, creating 64-bit by 64-bit multiplication with a 128-bit result (either signed or unsigned) from 32-bit code without any inline assembly. Intrinsics let you express what you want in C++, and give the variables meaningful names, and you can let the compiler do the tedious work of of register assignment, something compiler are generally better at than humans anyway.

Raymond Chen

Follow