Creating double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result using intrinsics (part 2)

devblogs.microsoft.com/oldnewthing/20171214-00

December 14, 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. We observed that the compiler was able to optimize out some memory accesses by extracting the values using shifts.

Let's see if we can tweak the code to remove the last of the memory accesses. Although the Windows calling convention for x86 does not have many general purpose registers available (only `eax` `ecx`, and `edx`), it does have eight `xmm` registers available, so we can use those as temporary holding places.

```
__m128i Multiply64x64To128(uint64_t x, uint64_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));

    return result;
}
```

We keep each of the four pieces of the result in a separate MMX register and convert it to an integer for the purpose of the `_addcarry_u32`, then convert it back to an MMX register once the arithmetic is complete. At the end, recombine the four pieces into a single value.

The convert-on-demand-and-then-back pattern is

```
    carry = _addcarry_u32(carry,
                          _mm_cvtsi128_si32(blah),
                          _mm_cvtsi128_si32(crossterms),
                          &temp);
    blah = _mm_cvtsi32_si128(temp);
```

where we take the low-order 32-bit value in `blah`, perform an add-with-carry with the low-order 32-bit vlaue in `crossterms`, then save the result back into `blah` while retaining the carry.

The other trick is that the lanes of the cross-terms are consumed only once each, and in order, so we can shift them into position and use `_mm_cvtsi128_si32` to pull them out one at a time.

The resulting compiler-generated assembly goes like this:

```
; xmm0 = y = { 0, 0, C, D }
    movq     xmm0, QWORD PTR _y$[esp-4]
; xmm1 = x = { 0, 0, A, B }
    movq     xmm1, QWORD PTR _x$[esp-4]

; xmm0 = { C, C, D, D }
    punpckldq xmm0, xmm0

; xmm4 = { C, C, D, D }
    movaps   xmm4, xmm0

; xmm1 = { A, A, B, B }
    punpckldq xmm1, xmm1

; xmm4 = { A * C, B * D } ; "result"
    pmuludq xmm4, xmm1

; xmm3 = { D, D, C, C }
    pshufd   xmm3, xmm0, 78

; xmm3 = { A * D, B * C } ; "crossterms"
    pmuludq xmm3, xmm1

; ecx = result[1]
    movaps   xmm0, xmm4
    psrldq   xmm0, 4
    movd     ecx, xmm0

; prepare to load result[2] from xmm0
    movaps   xmm0, xmm4

; eax = crossterms[0]
    movd     eax, xmm3

; prepare to load result[2] from xmm0
    psrldq   xmm0, 8

; shift crossterms[1] into position
    psrldq   xmm3, 4

; result[1] += crossterms[0], carry set appropriately
    add      ecx, eax

; eax = crossterms[1]
    movd     eax, xmm3

; shift crossterms[2] into position
    psrldq   xmm3, 4

; xmm1 = result[1]
    movd     xmm1, ecx
```

```
; ecx = result[2]
    movd     ecx, xmm0

; prepare to load result[3] from xmm0
    movaps   xmm0, xmm4
    psrldq   xmm0, 12

; result[2] += crossterms[1] + carry, carry set appropriate
    adc      ecx, eax

; eax = result[3]
    movd     eax, xmm0

; result[3] += carry
    adc      eax, 0

; xmm2 = result[2]
    movd     xmm2, ecx

; ecx = result[1]
    movd     ecx, xmm1

; xmm0 = result[3]
    movd     xmm0, eax

; eax = crossterms[2]
    movd     eax, xmm3

; shift crossterms[3] into position
    psrldq   xmm3, 4

; result[1] += crossterms[2], carry set appropriately
    add      ecx, eax

; eax = crossterms[3]
    movd     eax, xmm3

; xmm1 = result[1]
    movd     xmm1, ecx

; ecx = result[2]
    movd     ecx, xmm2

; xmm4 = { *, *, result[1], result[0] }
    punpckldq xmm4, xmm1

; result[2] += crossterms[3]
    adc      ecx, eax

; eax = result[3]
    movd     eax, xmm0
```

```
    ; result[3] += carry
        adc      eax, 0

    ; xmm2 = result[2]
        movd     xmm2, ecx

    ; xmm1 = result[3]
        movd     xmm1, eax

    ; xmm2 = { *, *, result[3], result[2] }
        punpckldq xmm2, xmm1

    ; xmm4 = { result[3], result[2], result[1], result[0] }
        punpcklqdq xmm4, xmm2

    ; set as return value
        movaps   xmm0, xmm4
        ret
```

I could go even further and realize that one of the `result#` variables could be left in a general-purpose register, since we need only two registers to perform the integer add. I also could have shifted the result a little bit at a time the same way I shifted the cross-terms a little bit at a time.

This version can perform all its work in registers, which means that there's no need for stack variables, which means that it becomes a lightweight leaf function. That means it doesn't need to create a stack frame.

Next time, we'll move on to signed multiplication.

Raymond Chen

Follow