Creating double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result using intrinsics (part 1)

devblogs.microsoft.com/oldnewthing/20171213-00

December 13, 2017

Raymond Chen

Some time ago, I derived how to create double-precision integer multiplication with a quad-precision result from single-precision multiplication with a double-precision result, but I expressed it in assembly language. This time, I'll express it in C using intrinsics.

Using intrinsics rather than inline assembly language is slightly more portable, since all the compiler toolchains that implement the intrinsics agree on what the intrinsics mean. They disagree on the members of a `__m128i`, but at least they agree on the intrinsics.

First, a straightforward translation of the assembly language code to intrinsics:

```
__m128i Multiply64x64To128(uint64_t x, uint64_t y)
{
    // movq xmm0, x         // xmm0 = { 0, 0, A, B } = { *, *, A, B }
    auto x00AB = _mm_loadl_epi64((__m128i*) &x);

    // movq xmm1, y         // xmm1 = { 0, 0, C, D } = { *, *, C, D }
    auto x00CD = _mm_loadl_epi64((__m128i*) &y);

    // punpckldq xmm0, xmm0 // xmm0 = { A, A, B, B } = { *, A, *, B }
    auto xAABB = _mm_unpacklo_epi32(x00AB, x00AB);

    // punpckldq xmm1, xmm1 // xmm1 = { C, C, D, D } = { *, C, *, D }
    auto xCCDD = _mm_unpacklo_epi32(x00CD, x00CD);

    // pshufd xmm2, xmm1, _MM_SHUFFLE(1, 0, 3, 2)
    //                      // xmm2 = { D, D, C, C } = { *, D, *, C }
    auto xDDCC = _mm_shuffle_epi32(xCCDD, _MM_SHUFFLE(1, 0, 3, 2));

    // pmuludq xmm1, xmm0 // xmm1 = { AC, BD } // provisional result
    auto result = _mm_mul_epu32(xAABB, xCCDD);

    // pmuludq xmm2, xmm0 // xmm2 = { AD, BC } // cross-terms
    auto crossterms = _mm_mul_epu32(xAABB, xDDCC);

    // mov    eax, crossterms[0]
    // add    result[4], eax
    auto carry = _addcarry_u32(0,
                               result.m128i_u32[1],
                               crossterms.m128i_u32[0],
                               &result.m128i_u32[1]);

    // mov    edx, crossterms[4] // edx:eax = BC
    // adc    result[8], edx
    carry = _addcarry_u32(carry,
                          result.m128i_u32[2],
                          crossterms.m128i_u32[1],
                          &result.m128i_u32[2]);

    // adc    result[12], 0      // add the first cross-term
    _addcarry_u32(carry,
                  result.m128i_u32[3],
                  0,
                  &result.m128i_u32[3]);

    // mov    eax, crossterms[8]
    // add    result[4], eax
    carry = _addcarry_u32(0,
                          result.m128i_u32[1],
                          crossterms.m128i_u32[2],
                          &result.m128i_u32[1]);

    // mov    edx, crossterms[12] // edx:eax = AD
    // adc    result[8], edx
    carry = _addcarry_u32(carry,
                          result.m128i_u32[2],
                          crossterms.m128i_u32[3],
                          &result.m128i_u32[2]);

    // adc    result[12], 0      // add the second cross-term
    _addcarry_u32(carry,
                  result.m128i_u32[3],
                  0,
                  &result.m128i_u32[3]);

    return result;
}
```

The Microsoft Visual Studio compiler produces the following:

```
; standard function prologue
    push     ebp
    mov      ebp, esp
    and      esp, -16   ; room for _result on the stack
    sub      esp, 24

; load up x and y
    movq     xmm1, QWORD PTR _y$[ebp]
    movq     xmm2, QWORD PTR _x$[ebp]

; duplicate x
    punpckldq xmm1, xmm1

; make another copy of the duplicated x
    movaps   xmm0, xmm1

; duplicate y
    punpckldq xmm2, xmm2

; multiply main terms, result in xmm0
    pmuludq xmm0, xmm2

; shuffle and multiply cross terms, cross-terms in xmm1
    pshufd   xmm1, xmm1, 141
    pmuludq xmm1, xmm2

; Now the adjustments for cross-terms

; save a register
    push     esi

; save result to memory
    movaps   XMMWORD PTR _result$[esp+32], xmm0

; load up result[2] into esi and result[3] into ecx
    mov      esi, DWORD PTR _result$[esp+40]
    mov      ecx, DWORD PTR _result$[esp+44]

; load result[1] into edx by shifting
    psrldq   xmm0, 4
    movd     edx, xmm0

; xmm0 holds cross-terms
    movaps   xmm0, xmm1

; load crossterms[0] into eax
    movd     eax, xmm1

; prepare to load crossterms[1] into eax by shifting
    psrldq   xmm0, 4

; add crossterms[0] to result[1]
```

```
    add      edx, eax

; load crossterms[1] into eax
    movd     eax, xmm0

; xmm0 holds cross-terms (again)
    movaps   xmm0, xmm1

; prepare to load crossterms[3] into eax by shifting
    psrldq   xmm1, 12

; prepare to load crossterms[2] into eax by shifting
    psrldq   xmm0, 8

; add crossterms[1] to result[2], with carry
    adc      esi, eax

; load crossterms[2] into eax
    movd     eax, xmm0

; propagate carry into result[3]
    adc      ecx, 0

; add crossterms[2] to result[1]
    add      edx, eax

; load crossterms[3] into eax
    movd     eax, xmm1

; save final result[1]
    mov      DWORD PTR _result$[esp+36], edx

; add crossterms[3] to result[2]
    adc      esi, eax

; save final result[2]
    mov      DWORD PTR _result$[esp+40], esi

; propagate carry into result[3]
    adc      ecx, 0

; save final result[3]
    mov      DWORD PTR _result$[esp+44], ecx

; load final result
    movaps   xmm0, XMMWORD PTR _result$[esp+32]

; clean up stack and return
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret
```

I was impressed that the compiler was able to convert our direct accesses to the internal members of the `__m128i` into corresponding shifts and extractions. (Since this code was compiled with only SSE2 support, the compiler could not use the `pextrd` instruction to do the extraction.)

Even with this very lame conversion, the compiler does quite a good job of optimiznig the code. The opening instructions match our handwritten assembly almost exactly; the second half doesn't match as well, but that's because the compiler was able to replace many of our memory accesses with register accesses.

The compiler was able to optimize our inline assembly!

We'll take this as inspiration for trying to get rid of all the memory accesses. The story continues next time.

Bonus chatter: In the three years since I wrote the original article, nobody picked up on the fact that I got the parameters to `_MM_SHUFFLE` wrong.

Bonus bonus chatter: I could have reduced the dependency chain a bit by tweaking the calculation of `xDDCC` :

```
// pshufd xmm2, xmm1, _MM_SHUFFLE(0, 0, 1, 1);
//                      // xmm2 = { D, D, C, C } = { *, D, *, C }
auto xDDCC = _mm_shuffle_epi32(x00CD, _MM_SHUFFLE(0, 0, 1, 1));

// punpckldq xmm1, xmm1 // xmm1 = { C, C, D, D } = { *, C, *, D }
auto xCCDD = _mm_unpacklo_epi32(x00CD, x00CD);
```

Basing the calculation of `xDDCC` on `x00CD` rather than `0xCCDD` removes one instruction from the dependency chain.

Bonus bonus bonus chatter: I chose to use `punpckldq` instead of `pshufd` to calculate `xCCDD` because `punpckldq` encodes one byte shorter.

Raymond Chen

Follow