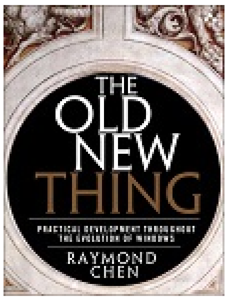
The case of the invalid instruction exception on an instruction that should never have executed

devblogs.microsoft.com/oldnewthing/20250718-00/?p=111390

July 18, 2025



The image processing folks added specialized AVX2 versions of their code, but found that it was crashing with an illegal instruction exception. The code went something like this:

```
void SwizzleAVX2(uint32_t* source, uint32_t* destination, uint32_t count)
{
    ⟦ do stuff using AVX-only instructions ⟧
    ⟦ such as _mm256_cvtepu8_epi16 ⟧
}

void SwizzleSSE4(uint32_t* source, uint32_t* destination, uint32_t count)
{
    ⟦ do stuff using SSE4 instructions ⟧
    ⟦ such as _mm_cvtepu8_epi16 ⟧
}

bool hasAVX2; // initialized elsewhere

void Swizzle(uint32_t* source, uint32_t* destination, uint32_t count)
{
    if (hasAVX2) {
        SwizzleAVX2(source, destination, count);
    } else {
        SwizzleSSE4(source, destination, count);
    }
}
```

This looks good, doesn't it? We check whether AVX2 instructions are available, and if so, we use the AVX2 version; otherwise we use the SSE4 version.

But in fact, this code crashes with an invalid instruction exception on systems that do not have AVX2. How can that be?

Compiler optimization.

According to the "as-if" rule, the compiler is permitted to perform any optimization that a program cannot legitimately detect, where "legitimately" means "within the rules of the language".

What happened is that the compiler first inlined the `SwizzleAVX2` and `SwizzleSSE4` functions into the `Swizzle` function, and then it reordered the instructions so that some of the AVX2 instructions from `SwizzleAVX2` were moved in front of the test of the `hasAVX2` variable. For example, maybe `SwizzleAVX2` started by setting some registers to zero. The compiler might have decided to do this because profiling revealed that `hasAVX2` is usually true, so it wants to get the registers ready in anticipation of using them for the rest of the `SwizzleAVX2` function.

Unfortunately, the compiler doesn't realize that our test of `hasAVX2` was specifically intended to prevent any AVX2 instructions from running. The concept of "instructions that might not be available" does not arise in the C or C++ language specifications, so there is nothing in the language itself that addresses the matter.

There are some directives you can use to tell the compiler that certain memory operations must occur in a specific order. For example, you can use interlocked operations with acquire or release semantics, or you can use `std::atomic_thread_fence`, or you can use explicit memory barriers.

However, none of them are of use here because the offending instruction isn't a memory instruction, so memory ordering directives have no effect.

The (somewhat unsatisfying) solution was to mark the AVX version as noinline so that the compiler cannot reorder instructions out of it.

```
__declspec(noinline)
void SwizzleAVX2(uint32_t* source, uint32_t* destination, uint32_t count)
{
    ⟦ do stuff using AVX-only instructions ⟧
    ⟦ such as _mm256_cvtepu8_epi16 ⟧
}
```