The Intel 80386, part 17: Future developments

devblogs.microsoft.com/oldnewthing/20190212-00

February 12, 2019

Raymond Chen

Although this series focused on the Intel 80386, I did promise to discuss future extensions, so here we go.

The Intel 80486 introduced pipelining and on-chip caching. The floating point coprocessor became integrated on most versions of the 80486, rather than existing as a separate chip.

There are a handful of new instructions.

```
    XADD    r/m, r      ; { d, s } = { d + s, d }
```

The *exchange and add* instruction does two things:

- Exchanges the source and destination.
- Adds the source and destination and puts the result in the destination.

Another way of thinking about it is that the `XADD` instruction adds the source to the destination, and then returns the original value of the destination in the source register.

In practice, this instruction is always used with a `LOCK` prefix in order to make it atomic. This instruction makes it possible for the `InterlockedIncrement` and `Interlocked-Decrement` functions to return the incremented or decremented result, rather than merely the sign of the result.

```
    ; Atomically increment a value and return the new value

    mov     eax, 1          ; amount to add
    lock xadd [value], eax  ; atomically add, return previous value
    inc     eax             ; increment previous value to get final value
```

The second new instruction is also used for atomic operations:

```
    CMPXCHG r/m, r      ; if d == lo, then ZF=1, d = s
                        ;             else ZF=0, s = d
```

The term *lo* comes from this table, which we've seen a few times before:

Operand size	Hi	Lo
byte	`AH`	`AL`
word	`DX`	`AX`
dword	`EDX`	`EAX`

The *compare and exchange* instruction compares the destination against *lo* (the correspondingly-sized subset of the *eax* register). If they are equal, then the zero flag is set and the destination is updated. If they are not equal, then the zero flag is clear and the source receives the current value of the destination.

Again, in practice, this instruction is always used with a `LOCK` prefix in order to make it atomic. This instruction makes it possible to implement the `InterlockedCompare-Exchange` function.

```
    ; Atomically set value to edx if the current value is eax

    lock cmpxchg [value], edx ; compare value with eax
                              ; and update to edx if equal
    jz      success           ; Jump if successfully updated

    ; update failed - the edx register contains the value we saw
```

The memory controller always observes a write operation, even if the comparison failed. In the case of a failed comparison, the original value is written back. (This step is necessary so that the memory controller knows when the interlocked operation is finished.)

For atomic operations, the x86 does not follow the "load locked / store conditional" pattern used by pretty much every other processor, so you cannot build things like "atomic multiply by 3" or "atomic take the next step in the Collatz conjecture". This makes it susceptible to the ABA problem unless special countermeasures are taken. As I've said before, the x86 architecture is the weirdo.

There is also a new instruction specifically designed for interop with big-endian systems:

```
    BSWAP   r32             ; reverse order of bytes
```

The *byte swap* instruction reverses the order of bytes in a 32-bit register: Bits 0 through 7 are exchanged with bits 24 through 31, and bits 8 through 15 are exchanged with bits 16 through 23. This is handy for converting between little-endian and big-endian data formats. You don't see this instruction in compiler-generated code, though.

Exercise: `BSWAP` operates only on bytes, and only on 32-bit registers. What if you needed to reverse the order of words in a 32-bit register? Or reverse the order of bytes in a 16-bit or 8-bit register?

There are a few new instructions for cache management, but they are available only to kernel-mode code, so you won't see them in user-mode code.

The next CPU in the 80386 series is the Intel Pentium. The Pentium is dual-issue (if you play your cards right), the floating point unit is now built-in (although it had its issues), it performs branch prediction, and various operations execute in fewer clocks.

The Pentium introduced the MMX instruction set, the first SIMD instructions for the architecture. I haven't covered SIMD instructions in any of these processor retrospectives so far, and I'm not going to start now. Aside from the SIMD instructions, a small number of new instructions for user-mode were introduced, none of which you'll see in compiler-generated code.

```
    CPUID                  ; retrieve CPU identification
```

Up until this point, there was no instruction for identifying which processor you were running on. There were various tricks, usually involving trying to set manipulate flags marked as *reserved*) and seeing what happens. This clearly doesn't scale, because you'll eventually run out of flag bits, And each such little trick becomes a compatibility constraint, so Intel decided to create an instruction whose primary purpose is to identify the processor.

Before issuing the `CPUID` instruction, you put an information code in the *eax* register. After the instruction executes, the *ebx*, *ecx*, and *edx* register contain the results, the meaning of which depends on the information code.

The `CPUID` instruction is a serializing instruction: All modifications to flags, registers, and memory are guaranteed to be completed before the `CPUID` executes, and no instruction after the `CPUID` will be fetched until after the `CPUID` completes. Clever people have used the `CPUID` instruction for this side effect.

```
    CMPXCHG8B m64      ; if d == edx:eax, then ZF=1, d = ecx:ebx
                       ;                  else ZF=0, edx:eax = d
```

The *compare and exchange 8 bytes* instruction is the 8-byte version of the `CMPXCHG` instruction, except that it operates on 8 bytes instead of 4. This lets you attach a counter to your pointer and avoid the ABA problem, though it costs you an additional four bytes of memory. Like `CMPXCHG`, the `CMPXCHG8B` instruction is in practice always combined with a `LOCK` to make the operation atomic.

Finally, we have this guy:

```
    RDTSC              ; edx:eax = processor timestamp counter
```

The *read timestamp counter* stores a 64-bit value into the *edx:eax* register pair, representing the current value of the processor's timestamp counter. In the Pentium, this returned the number of CPU cycles executed by the processor. Translating this into wall-clock time is complicated because the CPU does not execute all cycles in the same amount of time. If the CPU is in a low-power state, then cycles will take longer to execute.

This behavior of the timestamp counter changed with the Pentium 4. Starting with the Pentium 4, the timestamp counter increases at a constant rate, independent of CPU clock speed.

Future versions of this processor series improved performance, but did not change the programming model significantly. A new SIMD instruction set was introduced, called *SSE*, and a handful of new instructions were introduced, but they tended to be special-purpose and not used by compilers. Here are some of the ones you might see:

```
    CMOVcc  r32, r/m32  ; if condition cc is satisfied, then d = s
```

The *conditional move* instruction moves the source to the destination if the corresponding condition code is satisfied. Note that a read is issued to the source even if the condition is false, so the source must be readable.

```
    UD2                 ; undefined opcode
```

The *undefined opcode* is an instruction guaranteed to raise an *invalid instruction* exception. Some compilers emit this into code paths that should never execute, so that programs will crash immediately when there is a programming error, rather than executing random code.

Okay, that ends our whirlwind tour of the Intel 80386. I believe this covers all of the "processors Windows once supported but no longer does", at least for the Windows NT series. If you dig into the now-forgotten Windows CE series, you'll find a number of low-power processors. From that list, I've selected the SuperH-3 (also known as SH-3) for the next series.

I chose SuperH-3 because I found the source code to a version of Windows CE that still supported it! The others, not so much. Sorry, fans of the Philips DR 31500: I'll probably never get around to covering your processor, at least not in the context of processors that Windows once supported but no longer does.

Raymond Chen

Follow