The Intel 80386, part 4: Arithmetic

devblogs.microsoft.com/oldnewthing/20190123-00

January 24, 2019

Raymond Chen

Okay, we've laid enough groundwork that we can start looking at instructions.

Whereas arithmetic operations on most modern processors are three-operand (two sources and a destination), on the 80386, the arithmetic operations have only a single source and a single destination. The operations are performed in place, with the destination providing one of the source values, which is then overwritten by the result.

The general pattern for computation instructions is

```
    OP      r/m, r/m/i      ; d op= s,     set flags
```

As is generally the case, the two operands must be the same size, and they cannot both be memory. Unary operations do not take a second operand.

Note that these computation instructions destroy one of their inputs. If you need that value later, you'll have to remember to save it somewhere before you destroy it with the computation instruction.

The number of bits involved in the operation is implied by the operand sizes. For example:

```
    OP     DWORD PTR [eax], ebx ; *(int32_t)eax op= ebx
```

This is a 32-bit operation because the source and destination operands are both 32-bit values. The destination is a 32-bit memory value, and the source is a 32-bit register value.

You don't have to know the legal combinations of source and destination in order to read disassembly. You can assume the compiler generated valid code.

Okay, let's start doing some math.

```
    ADD     r/m, r/m/i      ; d += s,      set flags
    ADC     r/m, r/m/i      ; d += s + CF, set flags

    SUB     r/m, r/m/i      ; d -= s,      set flags
    SBB     r/m, r/m/i      ; d -= s + CF, set flags

    CMP     r/m, r/m/i      ; set flags for d - s, but do not update d

    NEG     r/m             ; d = 0 - d,   set flags
```

The `ADD` instruction adds the source to the destination. The `ADC` instruction also adds in the carry flag.

The `SUB` instruction subtracts the source from the destination. The `SBB` instruction also subtracts the carry flag.

The `CMP` instruction is the same as `SUB`, except that the result is thrown away rather than being stored back into the destination.

The `NEG` instruction negates its argument in place by subtracting it from zero. (Therefore, the carry flag is set if and only if the original value was nozero.)

```
    INC     r/m             ; d += 1, set flags except leave CF unchanged
    DEC     r/m             ; d -= 1, set flags except leave CF unchanged
```

The `INC` and `DEC` instructions increment and decrement the destination, respectively. They set flags based on the result, except that the carry flag is left unchanged.[1]

The multiplication and division instructions require some more groundwork to understand.

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

```
    MUL     r/m         ; hi:lo = lo * s (unsigned)
    IMUL    r/m         ; hi:lo = lo * s (signed)
```

The `MUL` instruction performs an unsigned multiplication. The `IMUL` instruction performs a signed multiplication. The sole operand is a source because the destination is implied: The source is multiplied by the *lo* register in the table above, and the double-precision result is stored in the *hi:lo* register pair, with the *hi* register receiving the most significant bits of the result, and the *lo* register receiving the least significant bits of the result.

Division is similar, but going from large to small.

```
    DIV     r/m         ; lo = hi:lo / s (unsigned)
                        ; hi = hi:lo % s (unsigned)
    IDIV    r/m         ; lo = hi:lo / s (signed)
                        ; hi = hi:lo % s (signed)
```

The double-width value in the *hi:lo* register pair is divided by the source, with the quotient going into *lo*, and the remainder going in *hi*.

These forms of the the multiplication and division operations do not permit an immediate as a source parameter. The source must be memory or a register.

In practice, you will pretty much always see 32-bit operands (thanks to the C language integer promotion rules), so all you really need to remember for these instructions are

```
    MUL     r/m32       ; edx:eax = eax * s (unsigned)
    IMUL    r/m32       ; edx:eax = eax * s (signed)
    DIV     r/m32       ; eax = edx:eax / source32 (unsigned)
                        ; edx = edx:eax % source32 (unsigned)
    IDIV    r/m32       ; eax = edx:eax / source32 (signed)
                        ; edx = edx:eax % source32 (signed)
```

The multiplication instructions set carry and overflow if the result is larger than the small value register. The division instructions trap to kernel mode if you try to divide by zero, or if the result does not fit in the output registers.

There are two additional forms for the `IMUL` instruction which do not fit the above pattern. The first is a two-operand version that follows the pattern for `ADD` :

```
    IMUL    r, r/m      ; d *= s (signed)
```

This is a more traditional-looking two-operand instruction[2] that updates the destination register in place.

There is even a (gasp) three-operand version similar to what you see in other processors.

```
    IMUL  r, r/m, i     ; d = s * t (signed)
```

This three-operand version accepts an immediate as the third operand, and it's the one the compiler typically generates. For example,

```
    IMUL  EAX, ECX, 212 ; EAX = ECX * 212 (signed)
```

These additional forms produce only single-precision results, but that's what the C and C++ languages produce, so it fits well with those languages. If you need a double-precision result, then you can use the single-operand `MUL` and `IMUL` instructions.

Note that there is no unsigned version of these additional forms. Fortunately, you can use the signed version for unsigned multiplication because the single-precision result is the same for both signed and unsigned multiplication. However, the flags are always set according to the signed result, so you cannot use them to detect unsigned overflow.

In practice, this is not a problem because the C language doesn't give you access to the overflow flags anyway.

Okay, that's arithmetic. Next time, we'll look at the bitwise logical operations.

[1] This quirk of the `INC` and `DEC` instructions later came back to haunt the architecture. Although the 80386 does not perform out-of-order execution, later revisions of the processor do, Leaving the carry flag unchanged creates a register dependency: You cannot execute an `INC` out of order with respect to another arithmetic instruction because the result of the `INC` and `DEC` instruction is dependent on the incoming carry flag from the arithmetic instruction. Compilers will sometimes replace `INC dest` with `ADD dest, 1` (and similarly `DEC` with `SUB`) to avoid the dependency. Even though it seems to do more work (it also has to compute carry), it actually has the potential to run faster because the dependency is removed.

[2] It looks more traditional, but it's actually the old `IMUL` that came first and therefore is more properly the traditional instruction.

Raymond Chen

Follow