The Intel 80386, part 5: Logical operations

devblogs.microsoft.com/oldnewthing/20190124-00

January 25, 2019

Raymond Chen

The next group of instructions we'll look are the bitwise logical operation.

```
    AND     r/m, r/m/i  ; d &= s, set flags
    OR      r/m, r/m/i  ; d |= s, set flags
    XOR     r/m, r/m/i  ; d ^= s, set flags

    TEST    r/m, r/m/i  ; calculate d & s, set flags

    NOT     r/m         ; d = ~d, do not set flags
```

The `AND`, `OR`, and `XOR` instructions set flags based on the numeric value of the result; carry and overflow are always clear.

The `TEST` instruction is the same as `AND`, except that the result is thrown away rather than being stored back into the destination. You can say that `AND` is to `TEST` as `SUB` is to `CMP`.

A quirk of the `TEST` instruction is that it does not support an 8-bit immediate with sign extension. The immediate must be the same size as the other operand. This means that you can save instruction encoding space by using a smaller data size:

```
    TEST    DWORD PTR [rax+10h], 40000000h  ; 7-byte instruction
    TEST    BYTE PTR [rax+13h], 40h         ; 4-byte instruction
```

If you do this, you will run afoul of the store-to-load forwarder. Fortunately, the 80386 doesn't have one.

We will learn later that moving constants into registers requires a large instruction encoding. To avoid this, you may see two idioms for setting a register to zero: You can subtract it from itself, or you can exclusive-or it with itself.

```
    SUB     eax, eax        ; set eax = 0, set flags
    XOR     eax, eax        ; set eax = 0, set flags
```

The 80386 doesn't really care either way, but later versions of the processor recognize the "`XOR` a register with itself" idiom and special-case it to avoid the dependency on the previous value of the register. Therefore, you'll see the `XOR` version in compiler-generated code.

The next group of instructions is the bit-testing group.

```
    BT      r/m, r/i        ; copy bit s of d to CF
    BTS     r/m, r/i        ; copy bit s of d to CF and set
    BTR     r/m, r/i        ; copy bit s of d to CF and reset
    BTC     r/m, r/i        ; copy bit s of d to CF and complement
```

The `BT` instruction tests a bit (lowest-order bit is bit zero) of the destination operand to the carry flag. If the destination is a register, then the bit number is taken mod *n*, where *n* is the register size. If the destination is memory, then the memory is considered a packed bit array, and bit *s* % 8 of byte *m* + (*s* / 8) is copied.[1] For example,

```
    BT      eax, 17     ; copy bit 17 of eax to carry
    SBB     ecx, -1     ; ecx -= -1 + CF
```

The effect of this sequence of operations is to increment the *ecx* register if bit 17 of *eax* is clear: If the bit is not set, then the `BT` results in carry clear, so the `SBB` instruction subtracts −1 from *ecx*, which has the effect of adding 1. If the bit is set, then the `BT` results in carry set, so the `SBB` instruction subtracts −1 from *ecx*, and then subtracts one more. Some algebra shows that *ecx* − (−1) −1 = *ecx* + 1 −1 = *ecx*, so there is no net change to the *ecx* register.

The `BTS`, `BTR`, and `BTC` instructions copy the bit to the carry flag, and then set, reset, or toggle the bit that was tested. I haven't seen the compiler generate these instructions, so you probably don't need to know them.

Next are the shift instructions.

```
    SHL     r/m, CL/i       ; d = d << s,               set flags
    SHR     r/m, CL/i       ; d = d >> s (zero-fill), set flags
    SAR     r/m, CL/i       ; d = d >> s (sign-fill), set flags
```

The `SHL` instructions shifts left, The `SHR` instructions shifts right with zero fill (unsigned shift), and the The `SAR` instructions shifts right with sign fill (signed shift).

The shift amount can be a constant (the encoding with 1 is more compact than the encoding with other constants), or it can be a variable in the *cl* register. No other register can be used to specify the shift amount. The shift amount is taken mod 32.

The last bit shifted out is placed in the carry flag. If the shift amount is the immediate 1, then the overflow flag is set if the sign bit changed. (If the shift amount is not the immediate 1, then the overflow flag is undefined.) The zero, sign, and parity flags are set based on the result.

Next come the double shift instructions.

```
    SHLD    r/m, r, CL/i        ; d = d << t, fill from s, set flags
                                ; n = 16, 32
    SHRD    r/m, r, CL/i        ; d = d >> t, fill from s, set flags
                                ; n = 16, 32
```

The shift left double and shift right double instruction shift the destination by the amount specified by the third operand (which must be a constant or the *cl* register) and fills in the bits from the second operand. The `SHLD` instruction fills with the high-order bits of *s*, and the `SHRD` instruction fills with the low-order bits of *s*. The last bit shifted out is copied to the carry flag. The shift amount is taken mod 32.

Although *n* can be 16, you won't see it in practice, so there's no point mentioning that the behavior is undefined if the shift amount (mod 32) is greater than 16.

Okay, so those were the logical operations. Next time, we'll look at data transfer instructions.

[1] Technically, it is bit *s* % *n* of *n*-bit unit *m* + (*s* / *n*). This means that

```
    MOV     ecx, 32
    BT      DWORD PTR [eax], ecx
```

will read four bytes from `[eax+4]` to `[eax+7]` and then test bit 0 of the value. Note that the bytes from `[eax+5]` to `[eax+7]` do not participate in the bit test, but they must still be accessible, or you will take an access violation.

Raymond Chen

Follow