The Intel 80386, part 6: Data transfer instructions

devblogs.microsoft.com/oldnewthing/20190127-00

January 28, 2019

Raymond Chen

The most common data transfer operation is the simple copy of a value from one place to another.

```
MOV     r/m, r/m/i  ; d = s, does not set flags
```

The `MOV` instruction does not support an 8-bit immediate with sign extension. The immediate must be the same size as the other operand. This is annoying if you want to move a small constant into a 32-bit register, because you have to burn four bytes to encode the constant.

Not often seen, but included here for completeness is the exchange of two values:

```
XCHG    r/m, r/m    ; exchange d and s, does not set flags
```

The `XCHG` instruction exchanges two values, and it has the bonus property of being atomic if one of the operands is a memory location. It is the only read-modify-write memory instruction on the 80386 that is automatically atomic. We'll learn more about atomic operations later.

The 80386 has an architectural stack register, which was the style at the time. (Nowadays, RISC-style processors establish the stack register by convention.)

```
PUSH    r/m/i32     ; push d onto stack
                    ; esp = esp - 4
                    ; *esp = d
POP     r/m32       ; pop top of stack into d
                    ; d = *esp
                    ; esp = esp + 4
```

The stack grows downward (toward lower addresses), and the stack pointer points to the top of stack. Therefore, pushing a value onto the stack means decrementing the stack pointer by 4, and then storing the value at the new top-of-stack address. Popping a value consists of loading from the top-of-stack into the destination, and then incrementing the stack pointer by 4.

Note that you are allowed to push or pop a memory operand. This makes `PUSH` and `POP` unusual in that they access two memory locations in a single instruction. One memory location is the operand, and the other is the top of the stack.

There are other types of push and pop operations, but you are not going to see them in compiler-generated code, so I won't bother covering them here.

We saw above that moving small constants into registers requires a large instruction encoding. We saw earlier how the `XOR` instruction could be used to generate the value zero. To generate small signed integers, you might see this:

```
PUSH     1          ; push sign-extended 8-bit constant to stack
POP      EAX        ; pop the value from the stack to the EAX register
```

The next category of data transfer operations are those that change size.

```
CBW                 ; sign-extend al to ah:al
CWD                 ; sign-extend ax to dx:ax
CDQ                 ; sign-extend eax to edx:eax
CWDE                ; sign-extend ax to eax
```

The Convert (Byte|Word|Dword) to (Word|Dword|Qword) instructions perform sign extension by copying the sign bit of the implied source to all bits of the implied destination.

Whereas the above "convert" instructions have implied source and destination, the "move and extend" instructions let you specify which registers you want as the source and destination.

```
MOVSX rm, r/mn      ; sign-extend s to d (m > n)
MOVZX rm, r/mn      ; zero-extend s to d (m > n)
```

These instructions are unusual in that the source and destinations are different sizes. The destination of a "move with sign extension" or "move with zero extension" must be a register larger than the source. (Because if it were the equal or smaller in size, there would be no extension going on.) Note that the source cannot be an immediate, so you can say good-bye to your dreams of using `MOVZX eax, 2` to load a small constant into *eax*.

There's another instruction which is in the same family as the data transfer operations, except that it doesn't actually transfer any data.

```
LEA     r, m        ; d = address of s, does not set flags
```

The "load effective address" instruction goes through the motions of loading a value from memory, except that instead of loading the value from memory, it loads the *address* of the memory that *would have been loaded*. In other words, it lets you take advantate of the arithmetic built into the memory address circuitry.

For example, there is no three-register `ADD` instruction, but you can simulate it with the `LEA` instruction:

```
LEA     eax, DWORD PTR [ecx + ebx] ; eax = ecx + ebx, does not set flags
```

Even though there is no memory access, you still have to specify a memory size in order to keep the memory address circuitry happy.

As we saw in our exploration of addressing modes, the memory address circuitry can perform computations of the form *reg* + *reg* * *scale* + `constant`, so you can use the `LEA` instruction to perform those calculations, provided you don't care about flags.

```
LEA     eax, DWORD PTR [ecx + ecx * 4] ; eax = ecx + ecx * 4 = ecx * 5
```

If you happen to know the value of a particular register, you can use `LEA` to calculate a small constant.

```
XOR     ebx, ebx        ; ebx = 0
LEA     eax, [ebx+1]    ; eax = ebx + 1 = 1
```

This gives you another way to load a small constant without burning a large `MOV eax, 1`.

Okay, so those are the data transfer instructions. Next time, we'll look at conditional instructions and control transfer.

Raymond Chen

Follow