The Intel 80386, part 2: Memory addressing modes

devblogs.microsoft.com/oldnewthing/20190121-00

January 22, 2019

Raymond Chen

All of the memory addressing mode demonstrations will be some form of this instruction:

```
MOV     somewhere, 0
```

which stores a zero *somewhere*.

In practice, the registers used to calculate effective addresses will be 32-bit registers.[1]

All the addressing modes look like

```
size PTR [something]
```

where the *size* specifies the number of bytes being accessed, and the *something* specifies which memory you want to access.

If you are simply reading disassembly, then you don't need to know the rules about which combinations of registers are legal for which types of addressing modes. You can assume the compiler generated valid code. From a disassembly point of view, you can treat all addressing modes as

```
size PTR [expression]
```

Specifically,

```
BYTE PTR [expression]                ; *(int8_t*)(expression)
WORD PTR [expression]                ; *(int16_t*)(expression)
DWORD PTR [expression]               ; *(int32_t*)(expression)
QWORD PTR [expression]               ; *(int64_t*)(expression)
TWORD PTR [expression]               ; *(int80_t*)(expression)
```

Examples:

```
MOV     BYTE PTR ds:[01234567h], 0       ; *(int8_t*)(0x01234567) = 0
MOV     WORD PTR [eax], 0                ; *(int16_t*)(eax) = 0
MOV     DWORD PTR [ecx*2+2Ch], 0         ; *(int32_t*)(ecx*2+0x2c) = 0
MOV     DWORD PTR [eax+ebx*4-12h], 0     ; *(int32_t*)(eax+ebx*4-0x12) = 0
```

Note that there is a `ds:` prefix on the first instruction. For some reason, the Windows disassembler doesn't trust itself when performing access to an absolute memory address and prints a superfluous `ds:` prefix on the instruction. Don't worry about it. For now.

The 80386 permits unaligned memory access, except where noted. Unaligned access may be slower than aligned access, however.

If all you care about is reading disassembly, then that is all you really need to know for now. The rest of today is digging into the various types of expressions you are allowed to put inside the square brackets.

Absolute: The address is a constant.

```
MOV     BYTE PTR ds:[01234567h], 0      ; *(int8_t*)(0x01234567) = 0
```

Register indirect: The address is the value of a register.

```
MOV     WORD PTR [eax], 0               ; *(int16_t*)eax = 0
```

Register indirect with short displacement: The address is the value of a register plus a signed 8-bit immediate.

```
MOV     DWORD PTR [eax-7], 0            ; *(int32_t*)(eax-7) = 0
```

Register indirect with long displacement: The address is the value of a register plus a 32-bit signed immediate.[2]

```
MOV     BYTE PTR [eax+123h], 0         ; *(int8_t*)(eax+0x123) = 0
```

The remaining memory addressing modes are more complicated.

Register indexed: The address is the sum of the values of two registers.

```
MOV     BYTE PTR [eax+ebx], 0           ; *(int8_t*)(eax+ebx) = 0
```

Register indexed with short displacement: The address is the sum of the values of two registers plus a signed 8-bit immediate.

```
MOV     WORD PTR [eax+ebx+12h], 0       ; *(int16_t*)(eax+ebx+0x12) = 0
```

Register indexed with long displacement: The address is the sum of the values of two registers plus a signed 32-bit immediate.

```
MOV     DWORD PTR [eax+ebx+1234h], 0    ; *(int32_t*)(eax+ebx+0x1234) = 0
```

Register scaled: The address is the value of a register multiplied by 2, 4, or 8.

```
MOV     BYTE PTR [eax*2], 0             ; *(int8_t*)(eax*2) = 0
```

Register scaled with short displacement: The address is the value of a register multiplied by 2, 4, or 8, plus a signed 8-bit immediate.

```
MOV     WORD PTR [eax*4+2], 0           ; *(int16_t*)(eax*4+2) = 0
```

Register scaled with long displacement: The address is the value of a register multiplied by 2, 4, or 8, plus a signed 32-bit immediate.

```
MOV     BYTE PTR [eax*4+01234567h], 0   ; *(int8_t*)(eax*4+0x1234567) = 0
```

Register scaled indexed: The address is the value of a register plus the value of a register multiplied by 2, 4, or 8.

```
MOV     WORD PTR [eax+ebx*2], 0         ; *(int16_t*)(eax+ebx*2) = 0
```

Register scaled indexed with short displacement: The address is the value of a register, plus the value of a register multiplied by 2, 4, or 8, plus a signed 8-bit immediate.

```
MOV     BYTE PTR [eax+ecx*2-8], 0       ; *(int8_t*)(eax+ecx*2-8) = 0
```

Register scaled indexed with long displacement: The address is the value of a register, plus the value of a register multiplied by 2, 4, or 8, plus a signed 32-bit immediate.

```
MOV     DWORD PTR [eax+ecx*2+01234567h], 0 ; *(int32_t*)(eax+ecx*2+0x1234567) = 0
```

The *ebp* register cannot be used with register indirect addressing because its encoding pattern is used to indicate that the addressing mode is one of the complicated ones. (These complicated ones use a so-called *SIB*, or *scaled index byte*, to help encode the operands.) If you want to perform a register indirect access through *ebp*, you can get the same effect by using a register indirect with displacement, and specify a displacement of zero.

The Microsoft assembler[3] allows you to specify the terms in any order.

```
MOV     DWORD PTR [eax+ebx*2+1234h], 0  ; *(int32_t*)(eax+ebx*2+0x1234) = 0
MOV     DWORD PTR [ebx*2+eax+1234h], 0  ; *(int32_t*)(eax+ebx*2+0x1234) = 0
MOV     DWORD PTR [1234h+ebx*2+eax], 0  ; *(int32_t*)(eax+ebx*2+0x1234) = 0
```

It also allows you to move a value out of the brackets, or to have multiple sets of brackets, in which case the values are combined via addition.

```
; assume "array" is a global variable

MOV     DWORD PTR array[ebx*2], 0       ; *(int32_t*)(array+ebx*2) = 0
MOV     DWORD PTR array[4], 0           ; *(int32_t*)(array+4) = 0
MOV     DWORD PTR [ebx*2][eax][4], 0    ; *(int32_t*)(eax+ebx*2+4) = 0
```

You can omit the square brackets if the reference is to a global variable. The assembler assumes you want to access the memory at that address and inserts the brackets automatically.

```
; assume "array" is a global variable

    MOV     DWORD PTR [array], 0            ; *(int32_t*)(array) = 0
    MOV     DWORD PTR array, 0              ; *(int32_t*)(array) = 0
```

You can also omit the `size PTR` if the size of the operand can be inferred. For example, most instructions have the rule that the source and destination be the same size. If one of the arguments has an ambiguous size, the assembler may be able to infer its size from the other argument. Examples:

```
    MOV     [eax+ebx*2], ecx                ; *(int32_t*)(eax+ebx*2) = ecx

; assume "array" is a global variable of type DWORD

    MOV     array[eax], 0                   ; *(int32_t*)(array+eax) = 0
```

In the first example, the assembler infers that you meant `DWORD PTR` because the other operand is a 32-bit register. In the second example, the assembler infers that you meant `DWORD PTR` because the `array` variable is of type `DWORD`.

There are some instructions that have implied memory address operands; we'll discuss those as they arise.

The debugger does not use any of the above shorthands. It always specifies the memory size explicitly, and it always uses square brackets to indicate a memory access. These two instructions are quite different:

```
    MOV     DWORD PTR [eax], 0              ; *(int32_t*)eax = 0
    MOV     eax, 0                          ; eax = 0
```

Next time, we'll look at the flags register.

[1] It is technically legal to use 16-bit registers to calculate the effective address, but your options are much more limited. Furthermore, only the least significant 16 bits of the result are used as the effective address, so the exercise is already pointless because the bottom 64KB of address space is left unmapped. You went to all the effort of calculating an address that cannot be used.

[2] You might wonder why we specify that the immediate is signed, since there is no sign extension from a 32-bit value to a 32-bit value. But the disassembler knows that it's signed, because it displays values greater than `7FFFFFFFh` as negative offsets.

[3] Note that other assemblers, most notably NASM, follow different rules from the Microsoft assembler (MASM).

Raymond Chen

Follow