The Intel 80386, part 8: Block operations

devblogs.microsoft.com/oldnewthing/20190129-00

January 30, 2019

Raymond Chen

Most of the special-purpose operations that the 80386 inherited from the 8086 are largely obsolete. Although processors still support them, the implementations are not optimized, and compilers don't generate them.

Except for the block operations. Those are still important.

The block operations (formally known as "string" instructions) operate on blocks of memory. They are another class of the unusual instructions that operate on two pieces of memory in a single instruction.

The implied source memory is pointed to by the *esi* register, and the implied destination memory is pointed to by the *edi* register. You are not required to specify the implied operands in assembly language, but the Windows disassembler always shows them. I'll show them as they are disassembled, since the focus of this series is on reading disassembly of compiler-generated code, not on writing assembly.

Remember this table?

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

We saw this table when we studied multiplication and division. Well, we're going to use the `lo` column again.

Let's also define this operation:

```
advance reg {
   if (direction flag is clear) reg += sizeof(size)
   if (direction flag is set  ) reg -= sizeof(size)
}
```

The `advance` operation performs a post-increment if the direction flag is clear, aka *up*, or a post-decrement if the direction flag is set, aka *dn* (down).

The *DF* flag is required to be *up* at function call boundaries. A function is permitted to set it to *dn* temporarily, but it needs to set it back to *up* before allowing control to leave the function.[1]

In practice, the direction flag is always *up*, except possibly for brief moments inside the `memmove` function when moving between overlapped memory blocks.

```
    MOVS    size PTR [edi], size PTR [esi] ; d = s
                                           ; advance edi
                                           ; advance esi

    CMPS    size PTR [edi], size PTR [esi] ; set flags per d - s
                                           ; advance edi
                                           ; advance esi

    SCAS    size PTR [edi]                 ; set flags per lo - d
                                           ; advance edi

    LODS    size PTR [esi]                 ; lo = s
                                           ; advance esi

    STOS    size PTR [edi]                 ; s = lo
                                           ; advance edi
```

The “move string” instruction copies the specified unit of memory from the source address to the destination address, and then post-increments or post-decrements the *edi* and *esi* registers. For example,

```
    MOVS    DWORD PTR [edi], DWORD PTR [esi]
                ; *(int32_t*)edi = *(int32_t*)esi
                ; if up, then edi += 4, esi += 4
                ; if dn, then edi -= 4, esi -= 4
```

The “compare string” instruction sets flags according to the calculation of `d - s`, the same as the `CMP` instruction, and then post-increments/post-decrements the *edi* and *esi* registers.

The “scan string” instruction compares the destination with the *lo* register and then post-increments/post-decrements the *edi* register.

The “load string” instruction loads *lo* from the source and then post-increments/post-decrements the *esi* register.

The “store string” instruction stores *lo* to the destination and then post-increments/post-decrements the *edi* register.

These instructions are known as “string” operations because they can include a “repeat” prefix that indicates that the operation should be repeated for a number of times specified by the *ecx* register, which is the length of the string.

Prefixed opcode	Meaning
`REP MOVS`	Move *ecx* units
`REPE CMPS`	Compare *ecx* units as long as they are equal
`REPNE CMPS`	Compare *ecx* units as long as they are different
`REPE SCAS`	Compare *ecx* units as long as they are equal to *lo*
`REPNE SCAS`	Compare *ecx* units as long as they are different from *lo*
`REP LODS`	Load *ecx* units into *lo*
`REP STOS`	Store *ecx* units from *lo*

The `REP` prefix causes the operation to repeat for *ecx* iterations.

The `REPE` prefix causes the operation to repeat for *ecx* iterations, provided that the result of the comparison was “equal”.

The `REPNE` prefix causes the operation to repeat for *ecx* iterations, provided that the result of the comparison was “not equal”.

In all cases, if *ecx* is zero, then the instruction is a nop.

The assembler accepts `REPZ` and `REPNZ` as synonyms for `REPE` and `REPNE`, respectively.

Although `REP LODS` is technically legal, it is of dubious utility because each iteration will overwrite *lo*, and only the last iteration’s result will remain.

At the end of the instruction, the *ecx* register has been decremented by the number of elements operated upon, and the *esi* and/or *edi* registers have been incremented or decremented by the number of bytes operated upon.

These instructions are typically used only in the following idioms:

```
; copy ecx units from esi to edi
REP MOVS size PTR [edi], size PTR [esi]

; look for lo in a buffer with ecx elements starting at edi
REPNE SCAS size PTR [edi]

; store ecx copies of lo into the buffer starting at edi
REP STOS size PTR [edi]
```

For the cases where there are multiple termination conditions, you can inspect the flags and the *ecx* register to determine which condition terminated the loop and consequently how many iterations of the loop were performed.

```
mov ecx, 100                ; search up to 100 characters
xor eax, eax                ; search for 0
mov edi, offset string      ; search this string
repne scas byte ptr [edi]   ; scan bytes looking for 0 (find end of string)
jnz toolong                 ; not found
sub edi, (offset string) + 1 ; calculate length
```

AFter preparing the preconditions for the `REPNE SCAS` instruction, we kick off the search. At the completion of the instruction, we know the following:

- If the zero byte was not found:
  - The loop ran for 100 iterations.
  - *ZF* will be clear (*nz*).
  - *ecx* was decremented 100 times. Its value is now zero.
  - *edi* was incremented 100 times. It now points one past the end of the buffer.
- If the zero byte was found, at offset *n*:
  - The loop ran for *n*+1 iterations.
  - *ZF* will be set (*zr*).
  - *ecx* was decremented *n*+1 times. Its value is the number of characters not scanned.
  - *edi* was incremented *n*+1 times. It now points one past the zero byte.

After the `REPNE SCAS` instruction, we check the *ZF* flag to see whether the zero byte was found. If not, then we declare the string too long.

Otherwise, the zero byte was found and we want to calculate the length. We have two choices: We could try to infer it from *ecx*, whose final value is 100 – (*n* + 1), or we could try to infer it from *edi*, whose final value is `offset string` + *n* + 1.

To infer it from *ecx*, we solve for *n* and get *n* = 99 – *ecx*. However, the 80386 does not have a way to subtract a register from a constant in a single instruction, so this would require us to use two instructions, say `sub ecx, 99` followed by `neg ecx`.

To infer it from *edi*, we solve for *n* and get *n* = *edi* – `offset string` – 1 = *edi* – (`offset string` + 1).

The second calculation is easier in this case, so we go with that.

These instructions are usually used with a repeat prefix, but for small numbers of iterations, they might be unrolled, to avoid the overhead of having to set up the `ecx` register. The `MOVS` instruction encodes in only one byte, so you can do four of them in fewer bytes than it takes to load a constant into a 32-bit register.

```
; move 16 bytes from esi to edi
MOVS    DWORD PTR [edi], DWORD PTR [esi]
MOVS    DWORD PTR [edi], DWORD PTR [esi]
MOVS    DWORD PTR [edi], DWORD PTR [esi]
MOVS    DWORD PTR [edi], DWORD PTR [esi]
```

The repeating instructions do not operate atomically. Rather, a single iteration is run, the registers are updated, and then the instruction pointer either advances to the next instruction if the loop termination condition is met, or it returns to the instruction if the loop should continue. This means that at each step, the *ecx* register decrements by one, the *edi* and/or *esi* registers advance by one unit, the flags are set as necessary, and then the instruction pointer either moves to the next instruction or stays put. (This design permits interrupts to be serviced during long block operations.)

You'll notice this behavior if you try to single-step through a repeated block operation in the debugger. Each single-step will run one iteration, and it will look like nothing happened because the instruction pointer didn't move. But something did happen: The *ecx* register was decremented, the *edi* and/or *esi* registers advanced, and flags may have been updated.

Next time, we'll look at the stack frame instructions.

[1] Back in the days when assembly language was still commonly used, a frustrating source of bugs was forgetting to set the direction flag back to *up* when you were finished. This caused future string operations to walk backward through memory rather than forward, and the result of the error was often not manifested until much, much later, at which point the culprit was long gone.

Raymond Chen

Follow