Weird things I learned while writing an x86 emulator

timdbg.com/posts/useless-x86-trivia

February 1, 2023

Feb 1, 2023
11 min read

If you've read my first post about assembly language, you might expect that this is another post on how to understand assembly language. I will write more about that at some point, but this post is not that. Instead, this post is going to talk about some of the weird things and random trivia I learned while writing an x86 and amd64 emulator. The emulator I wrote was for Time Travel Debugging. One piece of TTD is a CPU emulator, which is used to record the entire execution of a process at an instruction level. The first version of TTD was called iDNA, and the emulator for iDNA was written almost completely in assembly code, which was fast but difficult to maintain and extend.

I wasn't involved in the first version of TTD, but I was involved in the second version where we rewrote the emulation portion (and eventually most other parts). The new one was written in C++, and we aimed to achieve most of the performance of the assembly language version while having a more maintainable code base. Writing a CPU emulator is, in my opinion, the best way to *REALLY* understand how a CPU works. You need to pay attention to every detail. This post is a somewhat random collection of the things I learned. If you have a lot of experience with x86 these might be old news, but maybe a few are things you haven't seen before.

Useless x86 encoding trivia

The x86 encoding scheme is a bit funny in that there are often multiple ways to encode the exact same instruction. The `int 3` instruction can be encoded as `CD 03`, but can also be encoded in a single byte of `CC`. This is a very useful encoding because `int 3` is used as a software breakpoint. That way, it's always possible to set a breakpoint at any point in a function, even if it's an instruction that lands at the end of a memory page (with no page mapped after it). Many of these alternate encodings are designed to be shorter for some common case. For instance, adding an immediate value to `EAX` or `RAX` with the `ADD` instruction can be expressed in a compact form that's shorter than the more general instruction. An `add eax, ...` instruction can be encoded like this:

```
05cccccccc      add     eax,0CCCCCCCCh
```

Doing the same for ECX takes an extra byte:

```
81c1cccccccc    add     ecx,0CCCCCCCCh
```

So while both registers are general purpose, the fact that `EAX` is called the "Accumulator register" is not just a convention, it actually makes a difference to the encoding (and potentially the performance, as a result). Compilers tend to be very smart, and can take advantage of these shorter encodings whenever possible. Shorter instructions mean less data transferred from main memory, and more instructions that can fit in the instruction cache, which is good for performance.

There are other ways to have equivalent instructions besides just alternate encodings. Instructions in x86 can take "prefix bytes" which modify the behavior of the instruction. The "REX" set of prefixes are commonly used in 64 bit code to access a wider range of registers compared to 32-bit code (and make code sequences easier to recognize). An x86 CPU will happily take one of these prefixes, even if it doesn't have any effect. Put a "REX" byte on an 8-bit add, and it does nothing:

```
4004cc          add     al,0CCh
```

In fact, you can put TWO of these prefixes on. Many disassemblers (including the one in WinDbg) will get confused, but the CPU will execute it just fine:

```
0:000> rrax=0
0:000> u . L2
00007ff8`22160950 40              ???
00007ff8`22160951 4004cc          add     al,0CCh
0:000> t
00007ff8`22160954 cc              int     3
0:000> rrax
rax=00000000000000cc
```

Can we add as many prefixes as we want? You can add quite a few until you get to 15 bytes[1]. This length is a hard limit on current x86-compatible CPUs. Any instruction longer than 15 bytes is considered invalid and will generate an exception.

There are also some other weird prefixes that you might not know about. The "Address override" prefix can be used to reference a 32-bit address when running in 64-bit mode.

```
488d0424        lea     rax,[rsp]
67488d0424      lea     rax,[esp] # Note the extra '0x67' prefix
```

And when running 32-bit code, it will switch the address mode to 16-bit addresses! I don't know that I've ever seen a compiler generate code like that, but it's there. Something interesting to note, however, is that disassembling or interpreting these bytes correctly relies on knowing the default operand size and address size of a code segment (which can be configured by the kernel). If we disassemble the same bytes in 32-bit mode and 64-bit mode, we can get different results.

In 32-bit mode:

```
8b0424          mov     eax,dword ptr [esp]
```

In 64-bit mode:

```
8b0424          mov     eax,dword ptr [rsp]
```

It's not just address sizes either. The entire range that was previous dedicated to `INC reg` and `DEC reg` on x86 (40-4F) is now used for the REX prefix bytes on x64!

In 32-bit mode, we have this:

```
48              dec     eax
030424          add     eax,dword ptr [esp]
```

But in 64-bit mode, those two instructions have merged to form a single instruction!

```
48030424        add     rax,qword ptr [rsp]
```

The encoding space used by `INC` and `DEC` was quite large, so it's understandable why the AMD64 designers decided to use these bytes for the new prefixes to expand the register set in 64 bit mode. There was already a different encoding for these instructions that supported both registers and memory locations, so nothing was really lost (except a slightly larger encoding for a somewhat infrequent instruction).

I'll also note that due to this peculiarity and due to the fact that WinDbg will always assemble instructions as if they are in 32-bit mode, trying to assemble an "INC reg" instruction will always fail, because it will generate the single byte form which is actually a REX prefix.

```
0:000> a.
00007ff8`4bfc0950 inc eax
00007ff8`4bfc0951
0:000> u .
ntdll!LdrpDoDebuggerBreak+0x30:
00007ff8`4bfc0950 40eb00          jmp     ntdll!LdrpDoDebuggerBreak+0x33
(00007ff8`4bfc0953)
```

The `inc eax` here got turned into a useless REX prefix modifying the next instruction, instead of actually being an `inc` instruction.

Odd flag quirks

Speaking of `INC` and `DEC`, there's a slightly odd aspect of these instructions that's worth noting. You might thing that `INC EAX` does the same thing as `ADD EAX, 1`, but they are slightly different. An `ADD` instruction will update the carry flag but the `INC` instruction does not! This is an easy thing to miss, and when writing the TTD emulator I got this wrong initially, until I caught it with some unit tests.

There are a few other outliers worth mentioning. Most arithmetic and logical operations will set the overflow, sign, zero, auxiliary carry, parity, and carry flags. That includes the `CMPXCHG` (compare and exchange) instruction. Interestingly, it does not include the `CMPXCHG8B` / `CMPXCHG16B` instructions, which will only modify the zero flag. Weird, right?

Other instructions leave some of the flags in an undefined state, although in some cases they are *conditionally* undefined. The shift and rotate instructions leave the overflow flag undefined if the shift amount was more than one. It seems that the actual behavior of the undefined flags is related to the internal implementation of the shift operation, and is different between different architectures. I've heard that the Atom line of CPUs used a cheaper/slower way of doing bit shifts in the ALU, which results in the undefined flags having different values, although I haven't tested this myself.

More surprises with shift instructions

Consider this instruction:

```
66c1e810        shr     ax,10h
```

This instruction is to shift the `ax` register by 16 bits. Since this is a 16 bit register, it will clear the register to zero. Now consider this instruction:

```
c1e820          shr     eax,20h
```

This instruction says that it will shift the eax register 0x20 (32 decimal) bits to the right. Since this is a 32 bit register, you might think that it will clear the `eax` register. In reality, the value of eax is unchanged! If you read the details of this instruction in the Intel SDM, you'll see that the count is masked against 1FH, essentially using only the lowest five bits of the rotation.[2] If the `REX.W` prefix is used, the mask will be 3FH, meaning the maximum shift is 63 bits.

Segment overrides

While segmented memory might make you think we are back in the days of 16-bit code, it turns out that segments are alive and well in 32-bit and 64-bit code, and they can have real effects. We tend not to think about them very much because for the most part every OS uses a mostly-flat memory model and all of the segments have a base address of 0.[3] The exception to this tends to be for thread local storage, where one of the "extra segment registers" is used, either `FS` or `GS` (or both).

~~What can complicate things is the fact that usermode code doesn't have access to the CPU configuration that determines the base address of the FS or GS segments. So if you want to know what flat address corresponds to `GS:0x12345678`, there's no way to determine that directly unless the OS has a way of querying this information.~~

CORRECTION: Sixtyvividtails points out that you can read the base of fs/gs segs using the rdfsbase/wrfsbase/rdgsbase/wrgsbase instructions, which are available in unprivileged code. These instructions are available starting on Ivy Bridge (from 2012). Thanks for the correction, sixtyvividtails!

On Windows, these registers are used for referring to the TEB (Thread Execution Block), and these structures conveniently have a "self" pointer with a flat address to the start of the structure, which also happens to be the base of the segment.

In 32-bit processes, the TEB is located using `FS`. We can see how the OS does this by looking at the disassembly of the GetLastError function, which simply accesses a field out of the TEB.

```
0:000> u kernelbase!GetLastError
KERNELBASE!GetLastError:
773e4080 64a118000000    mov     eax,dword ptr fs:[00000018h]   # Grab the
TEB.NtTib.Self
773e4086 8b4034          mov     eax,dword ptr [eax+34h]        # Grab the
"LastErrorValue" field out of the TEB
773e4089 c3              ret
```

Looking at this using public symbols, you can see what those offsets correspond to:

```
0:000> dt -r1 _TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
      +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
...
      +0x018 Self             : Ptr32 _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
...
   +0x034 LastErrorValue   : Uint4B
```

In 64-bit processes, the TEB is located using `GS`:

```
0:000> u kernelbase!GetLastError
KERNELBASE!GetLastError:
00007ff8`4960cd60 65488b042530000000 mov   rax,qword ptr gs:[30h]
00007ff8`4960cd69 8b4068             mov     eax,dword ptr [rax+68h]
00007ff8`4960cd6c c3                 ret
```

It might seem odd that that 64-bit processes use a different segment register for the TEB than a 32-bit process, but there is a very good reason for this. A 32-bit process on a 64-bit OS will have both a 32-bit TEB *and* a 64-bit TEB, and in some contexts it can be useful to have access to both TEBs (such as the 64-bit WOW code that runs in a 32-bit process).

Segment overrides: More trivia

I mentioned earlier that the base address of the FS segment and GS segment is determined by CPU configuration. You might be wondering "what CPU configuration?" And the answer is "it depends". Specifically, it depends on whether you're in 32-bit mode or 64-bit mode. In 32-bit mode, the actual value of the segment register is used to reference a segment descriptor (defined by the Global Descriptor Table and Local Descriptor Table). But in 64-bit mode, the base is controlled by two MSRs, the FS Base (IA32_FS_BASE in the Intel SDM) and GS Base (IA32_GS_BASE). A side effect of this scheme is that the actual value of FS and GS don't matter at all in 64-bit mode. You can see the effect of this in WinDbg by trying to directly read from something in one of those segments. When debugging a 32-bit process, you can dump the contents of the "FS segment" by using the value of the FS register:

```
0:000> rfs
fs=00000053
0:000> db 53:0
0053:00000000  60 f2 2f 03 00 00 30 03-00 30 20 03 00 00 00 00 `./...0..0 .....
```

If you try the same thing on a 64-bit process, it doesn't work because the segment values don't matter, only the segment override prefixes!

```
0:000> rfs
fs=0053
0:000> db 53:0
0053:00000000`00000000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
0:000> rgs
gs=002b
0:000> db 2b:0
002b:00000000`00000000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
```

Roll credits

This turned out to be a pretty random list of x86 trivia. Most of it totally useless unless you want to write an emulator (which I *highly* recommend if you ever get the chance). I still think it's sort of interesting, and gives a little bit of insight into how things "really work". Some of this I learned through trial and error, but I had some great mentors while writing an x86 emulator, one of whom was Darek Mihocka, who has been doing emulators long enough that he owns emulators.com. I'd never claim to be an expert myself, but if this sort of thing is interesting to you make sure to check out the fantastic resources on Agner Fog's website. As usual, if I made any mistakes or if you have any questions, let me know on Twitter or Mastodon!

Footnotes

[1] There are some other limits on prefixes on older CPUs, but the 15 byte limit applies even on newer CPUs. Besides the length limitation, some prefixes are also more strict about when they can be used, such as the `LOCK` prefix (F0)

[2] This actually came up in an interview question at Microsoft. My interviewer was asking me all the ways you could clear a 32-bit register in a single instruction. He was convinced that using a shift would work and didn't believe me when I said it wasn't possible.

[3] When executing in 64-bit mode, the segment base of CS, DS, ES, and SS are always treated as 0 by the CPU.