Multi-bitness x86 code

Tomasz Grysztar
tgrysztar@flatassembler.net

I first needed a trick of this kind nearly 20 years ago, when I was working on a variant of unreal mode that allowed for 32-bit code segments. I wanted a common interrupt table that would not need rewriting back and forth, so my interrupt handlers had to detect if CS was a 32-bit segment and in that case switch back to regular real mode before calling the original vector (a 16-bit BIOS or DOS service). The snippet I used looked like:

```
hex             use16                   use32
3D 77 77        cmp     ax,7777h        cmp     eax,0??EB7777h
EB ??           jmp     already16bit
```

When an interrupt happens, the flags get stored on the stack, so it was not a big deal that CMP altered a few of them.

I thought this was a fun solution, but then I found out that it was actually possible to move IDT base in real mode, what made this trick obsolete.

Not long after, I was learning of the then-upcoming x86-64 architecture (an official name at the time) and I noticed that 32-bit instructions were mostly encoded the same in the new mode. Only things like addresses and stack elements were promoted to 64-bit automatically, other sizes stayed as they were unless a REX prefix was used. It stirred my imagination and made me believe that perhaps some of existing machine code could run correctly in 64-bit segment without any alterations.

Later I realized that many small obstacles made it not really viable in general, though certainly possible for some small snippets, like this one that converts slashes to backslashes in a UCS-2/UTF-16 string at ESI/RSI:

```
hex                     use32                   use64
56                      push    esi             push    rsi
                    convert:                convert:
66 AD                   lods    word [esi]      lods    word [rsi]
66 85 C0                test    ax,ax           test    ax,ax
74 0E                   jz      done            jz      done
66 83 F8 2F             cmp     ax,'/'          cmp     ax,'/'
75 F3                   jne     convert         jne     convert
66 C7 46 FE 5C 00       mov     word [esi-2],'\'mov     word [rsi-2],'\'
EB EB                   jmp     convert         jmp     convert
                    done:                   done:
5E                      pop     esi             pop     rsi
```

I never really needed a variant of my trick that would distinguish 64-bit mode from 32-bit one. But years later I learned that there are sightings of similar contrivances in the wild, even if made for purposes much different than mine. It made me think about upgrading my own snippet.

In 64-bit mode a CMP instruction is still 32-bit by default, so I could not simply reuse the old method.

I came up with this one:

```
hex             use64                   use32
67 8D 06        lea     eax,[esi]       lea     eax,[word 0??EBh]
EB ??           jmp     is64bit
```

This one does not touch any flags, but trashes EAX. It can be changed to use another register, but it always needs to sacrifice one.

Modern processors also got a new opcode that enables a completely transparent variant:

```
hex             use64                   use32
67 0F 1F 06     nop     [esi]           nop     [word 0??EBh]
EB ??           jmp     is64bit
```

The operand of this instruction is decoded but nothing more is done with it. The address does not have to be valid.

It might even be made into a three-way switch, for an unlikely occurrence that the code might get executed in 16-bit mode:

```
hex                 use64
67 0F 1F 06         nop     [esi]
EB ??               jmp     short not32bit
                    ; 32-bit mode detected...
                not32bit:
0F 1F 06            nop     [rsi]
EB ??               jmp     short is64bit
                    ; 16-bit mode detected...
                is64bit:
                    ; 64-bit mode detected...
```

Why would you ever need to tell 16-bit mode from 64-bit one? I don't know!

As a bonus, here comes another three-way detector that simply does not care about preserving flags or registers.

What makes it interesting is perhaps that its intent is obscured when only looking at 64-bit disassembly. But after reading this you might not get fooled anymore!

```
hex             use16                   use32                   use64
48              dec     ax              dec     eax
B8 EB ??        mov     ax,0??EBh       mov     eax,0??EB??EBh  mov     rax,0??EB??EB??EB??EBh
EB ??           jmp     is16bit
EB ??           jmp     is32bit         jmp     is32bit
EB ??           jmp     unreachable     jmp     unreachable
```

The extra copies of 0EBh that never get executed serve no real purpose, they just complete a nice pattern.