Jumping into the middle of an instruction is not as strange as it sounds

devblogs.microsoft.com/oldnewthing/20220111-00

January 11, 2022

Raymond Chen

Reuben Harris and Monte Davidoff spent time disassembling Bill Gates's original Altair BASIC. In an interview with *The Register*, Harris was impressed with the code, noting with some admiration, "I found a jump instruction that jumped to the middle of another instruction."[1]

You can find the targets of those jumps in the error handling code: Search for "Three common errors."

The trick here is that the 8080 uses variable-length instructions. The instruction sequence in question goes like this:

```
01CD    1E0C    OutOfMemory:    MVI E,0C
01CF    01                      LXI B,....

01D0    1E02    SyntaxError:    MVI E,02
01D2    01                      LXI B,....

01D3    1E14    DivideByZero:   MVI E,14
```

The 8080 processor has 8-bit registers named `A`, `B`, `C`, `D`, `E`, `H`, and `L`. Six of these registers can be paired up to create 16-bit pseudo-registers: `BC`, `DE` and `HL`.

The *load extended immediate* `LXI` instruction is a three-byte instruction which loads a 16-bit immediate value into a register pair. The first byte specifies the opcode and the destination register pair (in the above example, the `BC` register pair), and the second and third bytes form the 16-bit immediate.

The *move immediate* `MVI` instruction is a two-byte instruction which loads an 8-bit immediate value into a single 8-bit register. The first byte specifies the opcode and the destination register (in the above example, the `E` register), and the second byte is the 8-bit immediate.

Let's write out the byte stream that results from jumping to the three labels:

<table>
<tr><th>Address</th><th>Code byte</th><th><code>JMP OutOfMemory</code></th><th><code>JMP SyntaxError</code></th><th><code>JMP DivideByZero</code></th></tr>
<tr><td><code>01CD</code></td><td><code>1E</code></td><td rowspan="2"><code>MVI E,0C</code></td><td rowspan="3"></td><td rowspan="6"></td></tr>
<tr><td><code>01CE</code></td><td><code>0C</code></td></tr>
<tr><td><code>01CF</code></td><td><code>01</code></td><td rowspan="3"><code>LXI B,021E</code></td></tr>
<tr><td><code>01D0</code></td><td><code>1E</code></td><td rowspan="2"><code>MVI E,02</code></td></tr>
<tr><td><code>01D1</code></td><td><code>02</code></td></tr>
<tr><td><code>01D2</code></td><td><code>01</code></td><td rowspan="3"><code>LXI B,141E</code></td><td rowspan="3"><code>LXI B,141E</code></td></tr>
<tr><td><code>01D3</code></td><td><code>1E</code></td><td rowspan="2"><code>MVI E,14</code></td></tr>
<tr><td><code>01D4</code></td><td><code>14</code></td></tr>
</table>

If you jump to `01CD`, then the CPU performs a `MVI E,0C`, and then it interprets the `01` as the start of an `LXI B` instruction, and the next two bytes are treated as the 16-bit immediate operand. On the other hand, if you jump to `01D0`, then the bytes that used to be the 16-bit immediate operand of the `LXI B` instruction are now treated as an `MVI E,02` instruction.

You see the same thing happen at `01D3`, which hides a two-byte instruction inside the 16-bit immediate operand of another `LXI B` instruction. If instruction falls through from above, then the CPU executes an `LXI B,141E`, but if you jump directly to `1D3`, then the CPU executes a `MVI E,14`.

In both cases, the `LXI B` is just a garbage instruction. It loads some nonsense value into the `BC` register pair. The code doesn't care; that register wasn't holding anything useful anyway. The purpose of the instruction is to soak up the next two bytes and prevent them from being treated as another instruction.

Harris expressed some surprise at finding this, but really, it is a pretty common trick when hand-writing assembly for processors with variable-length instructions: If you want to hide a 1-byte instruction, look for another instruction with a 1-byte immediate, and hide the instruction in the immediate. If you want to hide a 2-byte instruction, hide it inside an instruction with a 2-byte immediate.

The "cloaking" instruction should do something harmless. Instructions like "compare with immediate" work great, since they typically affect only flags, and most of the time, there's nothing interesting in the flags anyway. However, the 8080 does not have a "compare with 16-bit immediate" instruction, so we have to make do with "load 16-bit immediate" into a register we don't care about.

On the 6502, the typical instruction for soaking up one or two bytes is the *bit test* `BIT` instruction. The argument is the address of the memory to test (either a 1-byte zero page address or a 2-byte absolute address), and the rest of the test goes into the flags register. Executing a garbage `BIT` instruction therefore reads a byte from some garbage memory location and then sets flags according to the value read. If the flags are subsequently ignored, then this is basically a three-byte `NOP`.

Microsoft 6502 BASIC had a special macro `SKIP2` for generating the first byte of the `BIT` instruction.

This hacky usage of the `BIT` instruction is arguably more popular than its designed purpose as a bit-testing instruction![2] (Related: The hunt for a faster syscall trap.)

One thing to watch out for is that the CPU does perform a load from the memory address that is the argument to the `BIT` instruction, so make sure that the two bytes, when reinterpreted as an address, don't produce an address in an I/O-mapped region. Otherwise, you'll be issuing inadvertent hardware commands. (The 6502 has no memory manager, so you don't have to worry about access violations.)

The trick of "soaking up" bytes to generate multiple entry points to a function was employed in 16-bit Windows. For example, you had this sequence:

```
DelAtom:
    mov     cl, 2
    db      0BBh        ; mov bx, imm16
AddAtom:
    mov     cl, 1
    db      0BBh        ; mov bx, imm16
FindAtom:
    mov     cl, 0
    db      0BBh        ; mov bx, imm16
```

The three functions all have the same parameters, and they share a lot of code, so the entry points merely set up a function code in the `cl` register and all fall through to a common implementation.

So, yeah, jumping into the middle of an instruction. It's a cool trick, but it's not novel. It was rather commonly employed in the early days of personal computing.

[1] For some reason, that quotation has made its way into online dictionaries as a citation for *jump instruction*.

[2] If you've done significant work on the 6502, the machine code for this instruction (`2C`) is probably burned into your brain.

Raymond Chen

Follow