The Intel 80386, part 12: The stuff you don’t need to know

devblogs.microsoft.com/oldnewthing/20190204-00

February 5, 2019

Raymond Chen

There are quite a few extra instructions that are technically legal in user-mode code, but which you won’t see in compiler-generated code because they are simply too weird.

```
    PUSHAD              ; push all general-purpose registers
    POPAD               ; pop (almost) all general-purpose registers
    PUSHFD              ; push flags register
    POPFD               ; pop flags register
    LAHF                ; AH = flags
    SAHF                ; flags = AH
```

The `PUSHAD` (push all doubleword) and `POPAD` (pop all doubleword) instructions push and pop the eight general-purpose registers onto the stack. This includes the stack pointer register *esp*! The `PUSHAD` instruction pushes the *esp* register onto the stack, and the `POPAD` instruction pops the value, but doesn’t store it into the *esp* register. The value that would normally go into the *esp* register is simply discarded.

The `PUSHFD` (push flags doubleword) and `POPFD` (pop flags doubleword) instructions push and pop the flags register to/from the stack. When popped, some flag bits are discarded rather than being stored into the flags register.

The `LAHF` (load *ah* from flags) and the `SAHF` (store *ah* to flags) instructions transfer the *sf*,*zf*, *af*, *pf*, and *cf* flags to and from the *ah* register.

The next group of instructions are for binary-coded decimal. Packed binary coded decimal (packed BCD) uses a single byte to represent values from 0 to 99, putting the tens digit in the upper nibble and the units digit in the lower nibble. Each subsequent byte represents another power of 100. For example, the decimal number 764 is represented by the byte `0x64` followed by the byte `0x07`. In the mnenonic, this is known as “decimal” BCD.

Unpacked binary coded decimal (unpacked BCD) uses a single byte to represent a single digit from 0 to 9. The value simply represents itself. Each subsequent byte represents another power of ten. For example, the decimal number 764 is represented by the bytes `0x04`, `0x06` and `0x07`. In the mnenonic, this is known as “ASCII” BCD.

```
    DAA                     ; decimal (packed BCD) adjust after addition
    DAS                     ; decimal (packed BCD) adjust after subtraction
    AAA                     ; ASCII (unpacked BCD) adjust after addition
    AAS                     ; ASCII (unpacked BCD) adjust after subtraction
    AAM                     ; ASCII (unpacked BCD) adjust after multiplication
    AAD                     ; ASCII (unpacked BCD) adjust after division
```

All of the BCD adjustment instructions are expected to be executed immediately after the corresponding arithmetic operation, and the destination of the arithmetic operation is expected to be the *al* register. I mean, there's nothing preventing you from executing the instructions even if you didn't meet the prerequisites, but the results are not likely to be very useful.

The `DAA` instruction (decimal adjust after addition) assumes that you added two bytes in packed BCD format, and it converts the result back into packed BCD format, setting the carry flag according to whether the result was 100 or greater. For example, if you added `0x23` and `0x59`, the initial result is `0x7C` (which is the sum as normal integers), and the the `DAA` instruction adjusts the value to `0x82`, to represent the packed BCD sum.

The `DAS` (decimal adjust after subtraction) instruction operates similarly.

The `AAA` (ASCII adjust after addition) assumes that the operation you performed was on an unpacked BCD value. It adjusts the value in the *al*, and if a carry occured, it increments the *ah* register. The `AAS` (ASCII adjust after subtraction) operates similarly.

The `AAM` (ASCII adjust after multiplication) instruction assumes that the most recent operation was a multiply of two 8-bit values in unpacked BCD format, producing a result in *ax*.

The `AAD` (ASCII adjust before division) instruction is unusual in that you execute it *before* the corresponding instruction. It takes an unpacked BCD two-digit value in *ax* and prepares it so that the upcoming 16-by-8 division will produce correct decimal values.

The next instructions are for bit-scanning.

```
    BSF     r, r/m      ; d = index of first set bit in s
    BSR     r, r/m      ; d = index of last set bit in s
```

The `BSF` (bit scan forward) instruction searches for the least significant set bit in the source value and sets the destination register to the index of that bit. The `BSR` (bit scan reverse) instruction does the same, but it looks for the most significant set bit. If the source is zero, then the destination is undefined and the *zf* flag is set.

The next group is the rotation instructions.

```
    ROL     r/m, CL/i   ; d = d rotate left by s, set flags
    ROR     r/m, CL/i   ; d = d rotate left by s, set flags
    RCL     r/m, CL/i   ; d = d|CF rotate left by s, set flags
    RCR     r/m, CL/i   ; d = d|CF rotate left by s, set flags
```

The `ROL` instruction rotates the bits of the destination left (towards higher significance) by the amount specified by the source, which is taken mod 32. The `ROR` instruction rotates right. The carry flag contains the last bit rotated out, and if the shift amount is the immediate 1, then the overflow flag is set if the sign bit changed. (If the shift amount is not the immediate 1, then the overflow flag is undefined.) The zero, sign, and parity flags are set based on the result.

The `RCL` and `RCR` instructions are similar, except that rotation is through an *n*+1 bit value, where the carry flag is the extra bit.

And then there are the counted loop instructions.

```
    LOOP    dest        ; decrement ecx, jump if result is nonzero
    LOOPE   dest        ; decrement ecx, jump if result is nonzero
                        ; and ZF is set (alternate opcode: LOOPZ)
    LOOPNE  dest        ; decrement ecx, jump if result is nonzero
                        ; and ZF is clear (alternate opcode: LOOPNZ)
    JECXZ   dest        ; jump if ecx is zero
```

The counted loop instructions require the loop counter to be stored in the *ecx* register. The usual pattern is

```
    MOV     ecx, number_of_iterations
    JECXZ   done        ; no iterations at all
again:
    ... do something ...
    LOOP    again       ; do it number_of_iterations times
done:
```

You can also make the loop conditional upon the *zf* flag. The `LOOPE` (loop while equal) instruction loops provided the result of the most recent flags-setting operation was zero. The `LOOPNE` (loop while not equal) requires that the most recent flags-setting result be nonzero.

```
    MOV     ecx, number_of_iterations
    JECXZ   done        ; no iterations at all
again:
    ... do something ...
    CMP     eax, 90
    LOOPZ   again       ; do it number_of_iterations times
                        ; provided eax is 90
done:
    ; loop ends when we have executed all iterations or eax is not 90
```

And then some random instructions I couldn’t categorize easily.

```
    XLAT                ; al = byte at ebx+al
    BOUND   r, m        ; check that d is in range [s]..[s+4]
    INTO                ; check if overflow is set
```

The `XLAT` instruction treats the value in the *al* register as an index into in a table of 256 bytes starting at *ebx*, putting the result back into the *al* register. My guess, given the opcode name, is that this was for character set translation where the characters in the source and destination character sets are both single-byte. (Think ASCII and EBCDIC.)

The `BOUND` instruction performs a bounds check of the destination register. The source refers to two 32-bit values in memory, the first being the smallest legal value and the second being the largest legal value. If the destination value is not in range, then interrupt 5 is raised. The values are treated as unsigned because the intended purpose of this instruction is to perform an array bounds check.

The `INTO` instruction checks whether the overflow bit is set. If so, then it raises interrupt 4.

Finally, there are instructions so weird I won't even go into them. They are technically legal instructions but are not useful in practice because 32-bit Windows uses a flat address space.

```
    ARPL    r/m16, r16  ; adjust requested privilege level
    LAR     r32, r/m32  ; load access rights
    LSL     r32, r/m32  ; load selector limit
```

These instructions operate on selectors, but since there are no interesting selectors in 32-bit Windows (aside from the TEB, which we discussed earlier), these instructions don't accomplish anything interesting.

Next time, we'll look at the Windows calling conventions.

Raymond Chen

Follow