The Intel 80386, part 1: Introduction

devblogs.microsoft.com/oldnewthing/20190120-00

January 21, 2019

Raymond Chen

Windows NT stopped supporting the Intel 80386 processor with Windows 4.0, which raised the minimum requirements to an Intel 80486. Therefore, the Intel 80386 technically falls into the category of "processor that Windows once supported but no longer does." This series focuses on the portion of the x86 instruction set available on an 80386, although I will make notes about future extensions in a special chapter.

The Intel 80386 is the next step in the evolution of the processor series that started with the Intel 8086 (which was itself inspired by the Intel 8080, which was in turn inspired by the Intel 8008). Even at this early stage, it had a long history, which helps to explain many of its strange corners.

As with all the processor retrospective series, I'm going to focus on how Windows NT used the Intel 80386 in user mode because the original audience for all of these discussions was user-mode developers trying to get up to speed debugging their programs. Normally, this means that I omit instructions that you are unlikely to see in compiler-generated code. However, I'll set aside a day to cover some of the legacy instructions that are functional but not used in practice.

The Intel 80386 has eight integer registers, each 32 bits wide.

Register	Meaning	Preserved?
eax	accumulator	No
ebx	base register	Yes
ecx	count register	No
edx	data register	No
esi	source index	Yes
edi	destination index	Yes

ebp	base pointer	Yes
esp	stack pointer	Sort of

The register names are rather unusual due to the history of the processor line. That history also explains why the instruction encoding uses the non-alphabetical-order *eax*, *ecx*, *edx*, *ebx*.

Also for historical reasons, there are also names for selected partial registers.

Register	Meaning
ax	Lower 16 bits of *eax*
bx	Lower 16 bits of *ebx*
cx	Lower 16 bits of *ecx*
dx	Lower 16 bits of *edx*
si	Lower 16 bits of *esi*
di	Lower 16 bits of *edi*
bp	Lower 16 bits of *ebp*
sp	Lower 16 bits of *esp*
ah	Upper 8 bits of *ax*
al	Lower 8 bits of *ax*
bh	Upper 8 bits of *bx*
bl	Lower 8 bits of *bx*
ch	Upper 8 bits of *cx*
cl	Lower 8 bits of *cx*
dh	Upper 8 bits of *dx*
dl	Lower 8 bits of *dx*

Operations on these register fragments affect only the indicated bits; the other bits of the 32-bit register remain unaffected. For example, storing a value into the *ax* register leaves the most-significant 16 bits of the *eax* register unchanged.[1]

Windows NT requires that the stack be kept on an 4-byte boundary. There is no red zone.

The 80386 also has eight 80-bit extended precision floating point registers named *st0* through *st7*. The floating point system is rather unusual: In addition to the fact that the registers are extended precision, the programming model for the floating point registers is as a stack. Values are pushed onto the floating point stack, operations are performed on the stack, and results are popped off.

Floating point support is optional and is provided by the 80387 coprocessor chip, which runs concurrently with the main CPU. If a floating point instruction is executed on a system that lacks a floating point coprocessor, the floating point instruction traps, and the kernel emulates the instruction.

There are also some non-integer registers which are difficult/impossible to get to, but which still participate in user-mode instructions.

Register	Meaning	Notes
eip	instruction pointer	program counter
eflags	flags	
cs	code segment	Don't worry about it
ds	data segment	Don't worry about it
es	extra segment	Don't worry about it
fs	F segment	For TEB access
gs	G segment	Not used

Windows NT uses the 80386 in flat mode, which means that applications see a contiguous 32-bit address space. The segment registers largely don't come into play when in flat mode, with the exception of the *fs* register, which we'll learn about more when we get to the TEB.

The flags register is updated by many instructions. We'll learn more about flags when we study conditionals.

The 80386 is unusual in that it supports multiple calling conventions. Common to all the calling conventions are the register preservation rules and the return value rules: The function return value is placed in *eax*. If the return value is a 64-bit value, then the most significant 32 bits are returned in *edx*. If the return value is a floating point value, it is returned in *st0*, and possibly *st1* (for complex numbers).

Furthermore, link-time code generation is permitted to manufacture ad hoc calling conventions which may not even follow the register preservation rules. *It's crazy free-for-all time.*

The architectural names for data sizes are as follows:

- **byte**: 8-bit value
- **word**: 16-bit value
- **dword** (doubleword): 32-bit value
- **qword** (quadword): 64-bit value
- **tword** (ten-byte word): 80-bit value

Instruction encoding is highly irregular. Instructions are variable-length, and instructions can begin at any byte boundary.

The general pattern for multi-operand opcodes is

```
opcode  destination, source
```

Note that the destination is on the left. Note also that three-operand instructions are rare. This will become interesting when we get to arithmetic.

Here's the notation I will use when introducing instructions:

Notation	Meaning
r*n*	*n*-bit register
m*n*	*n*-bit memory
i*n*	*n*-bit immediate
r/m*n*	*n*-bit register or *n*-bit memory
r/m/i*n*	*n*-bit register, *n*-bit memory, *n*-bit immediate, or 8-bit immediate sign-extended to *n* bits

- If *n* is omitted, then 8, 16, and 32 are permitted. For example, "r/m" means "r/m8, r/m16, or r/m32".
- Immediates are sign-extended as necessary.
- The first operand is called "d" (destination).
- The second operand (if any) is called "s" (source).
- The third operand (if any) is called "t" (second source).
- At most one of the operands can be a memory operand.
- All operands must have the same size.

Exceptions to the above rules will be called out as necessary.

For example:

```
ADD     r/m, r/m/i          ; d += s,       set flags
```

The `ADD` instruction takes two operands. The first is a register or memory, and the second is a register or memory or immediate or single-byte immediate. They cannot both be memory operands. They must be the same size.

Many instructions have a more compact encoding if the destination register is *al*, *ax*, or *eax*.

The assembly language overloads multiple variations of instructions into a single opcode. This is different from most other processors, where each opcode maps to an instruction template, where all that's left to fill in are the registers and immediates. For example, the MIPS R4000 has two different shift opcodes depending on whether the shift amount is specified by an immediate or a register. But the 80386 assembly language uses the same opcode for both, and it's the assembler's job to figure out which variant you intended.

The 80386 does not not perform speculation, does not have an on-chip cache, does not have a branch predictor, and does not reorder memory accesses. Life was simpler then.

Okay, that's enough background. We'll dig in next time by looking at memory addressing modes.

[1] This partial register behavior wasn't a big deal at the time, but it ended up creating register dependencies that made it much harder to add out-of-order execution to later versions of the processor. It even created a register version of the store-to-load forwarding problem.

The x86-64 architecture took a different approach when it extended the 32-bit registers to 64-bit registers: If the destination register is encoded as a 32-bit subset of a 64-bit register, the upper 32 bits of the destination register are zeroed.

Raymond Chen

Follow