INTEL INSTRUCTION FORMAT

The average person will not know or even care about the Intel opcode format. However, if you're going to be writting an assembler or compiler, which is often needed when writting your own operating system, then this knowledge is essential. I'll try to present this information as best as I can, however, it is a very in depth topic and difficult to explain at times. You may be confused at times, but I suggest you either read on and things may make more sence as you see them used more, or try reading from the beginning again. Some things I will only explain briefly (mainly the register/memory byte) because it is such an in depth topic. See Intel's developer's site for the most advanced and up to date information.

History

To understand certain things, it is often useful to research its history. Intel instruction formatting is no exception. When Intel introduced the 386 it added 32-bit versions of all the registers. You know these as EAX, EBX, ECX and so on. However, Intel did not add new opcodes to reflect these new registers. Infact, both the 16-bit registers and 32-bit registers look exactly the same from an opcode point of view. How then, do you distinguish them?

The answer is simple; If the code is executing in a 32-bit segment, the 32-bit registers are used. If the code is executing in a 16-bit segment, the 16-bit registers are used. However, how do you access 16-bit registers in a 32-bit segment and visa-versa? You use opcode prefixes.

There are two prefixes relevant here. The operand-size prefix (66h) and the address-size prefix (67h). When the operand-size prefix is used, then the non-default register set is used. For example, if the intruction is executed in a 16-bit segment, but has the operand-size prefix in the opcode, then the 32-bit registers are used. Inversly, if the intruction is executed in a 32-bit segment, but has the operand-size prefix in the opcode, then the 16-bit registers are used. The address-size prefix works analagously to the operand-size prefix, except that it effects wether 16-bit memory addresses and offsets or 32-bit addresses and offsets are used.

Alright then, how does the processor determine wether the segment is a 16-bit or a 32-bit segment? Well, in protected mode the descriptors hold this information. The 'D' bit of the executable segment descriptor determines what size the segment is. A value of 0 sets the default address and operand size to 16-bits. Conversly a value of 1 selects 32-bit default address and operand sizes. In real mode the default address and operand size is always 16-bits. This is also true for programs executing in Intel's 'Virtual-8086' mode.

What about the stack? You may ask. Well, the size of the stack is controlled by the 'B' bit of the data segment descriptor in the SS register. Again, a value of 0 selects a 16-bit stack, while a value of 1 selects a 32-bit stack.

Ofcourse, there are more prefixes to come, however, they'll have to wait for a while.

The Format

Okay, now with the history out of the way we can begin taking about the actual physical format of an opcode. Here's what a general Intel intruction looks like:

Intruction Prefix	Address-size prefix	Operand-size prefix	Segment override	Opcode	Register / Memory	Scale, index , base	displacement	immediate
Optional: 1 byte	Optional: 1 byte	Optional: 1 byte	Optional: 1 byte	1 or 2 bytes	Optional: 1 byte	Optional: 1 byte	Optional: 1, 2, or 4 bytes	Optional: 1, 2 or 4 bytes

That should be pretty self explainatory. The byte values at the bottom are the length of each of the units. Simply put, an Intel intruction is an opcode, with four possible prefixes and four optional postfixes. Simple, eh?

Alright, lets go from left to right and explain what needs to be. First off, you'll notice the intruction prefix. Where did that come from? Well, remember your good friend 'rep stosd'? Contrary to popular belief, rep is not an actual opcode. It's a modifier, and it belongs in the intruction prefix byte.

Here is a list of all possible intruction prefixes on current Intel processors, and also some limitations of their use:

F3h : REP, REPE, and REPZ	Use only with string operations (stosb, movsb, etc)
F2h : REPNE and REPNZ	Again, use only with string operations
F0h : LOCK	Use only with: BT, BTS, BTR, BTC ('mem, reg/imm' operand forms) XCHG ('reg, mem' and 'mem, reg' operand forms) ADD, OR, ADC, SBB ('mem, reg/imm' operand forms) AND, SUB, XOR (all operand forms) NOT, NEG, INC, DEC (on 'mem' operands)

The next thing you'll notice will be the segment override byte. You can probably guess where this comes into play. It's needed in structions like 'mov ax, fs:[eax]' where the default segment is overrideded. It's as simple as that. Here are the available segment override bytes:

2Eh	CS segment override
36h	SS segment override
3Eh	DS segment override
26h	ES segment override
64h	FS segment override
65h	GS segment override

The opcode is, as you probably already know, a byte which represents the intruction that is to be executed. Unfortunately, I can't list all the opcodes in this text. There are hundreds of opcodes on Intel processors. Intel themselves will probably provide this information on their web site

(developer.intel.com) however.

Next comes with register/memory byte. This byte is a bit more complicated then all the others, and is rather opcode dependent. It's basic purpose is to tell the opcode which operands to use. It looks like this:

Mod	Reg/Opcode	R/M
Bits 7 and 6	Bits 5 to 3	Bits 0 to 2

The Mod field basically tells which type of addressing the opcode should use. The Reg/Opcode section either holds a value for a register, or an extra three bits of the opcode. The R/M field either contains another register, or a memory operand. Here's what each of the registers look like:

000	AL, AX, EAX
001	CL, CX, ECX
010	DL, DX, EDX
011	BL, BX, EBX
100	AH, SP, ESP
101	CH, BP, EBP
110	DH, SI, ESI
111	BH, DI, EDI

Now, you already know that you use the operand-size prefix to select either the 16-bit or 32-bit registers. But how do you select the 8-bit registers? Well, there are special opcodes for that purpose which work on only 8-bit registers.

Realise, ofcourse, than my description of the Register/Operand byte is very basic. I have opted to leave out some of the specifics because they would fill pages and are already freely available at developer.intel.com.

Alright, next we find the SIB, or scale index base byte. This is the byte that allows you to do such things as '[EAX*8]' to address memory. The format is simple:

Scale	Index	Base
Bits 7 and 6	Bits 5 to 3	Bits 2 to 0

The scale is a 2 bit value which represent a scale of either 1 (00),2 (01),4 (10) or 8 (11). The index is a register, which is denoted by the table above. The base is also a register.

Lastly, I'll cover both the displacement and immediate fields together. They are exactly what you'd think they are. The displacment can be a 1, 2 or 4 byte displacement for the particular intruction. The immediate field holds any immediate data to be processed by the intruction.