Understanding Intel Instruction Sizes

In certain types of programming, such as 256 byte intros, space is severely limited. As a result, size-optimizing code in assembly language it is often necessary. This article discusses the machine-code sizes of the common Intel architecture instructions, from the perspective of code optimization. Understanding the size of the machine code produced by an assembler is necessary to make effective optimization decisions. Without this information, it is impossible to chose between different coding options other than by trial-and-error, which is time-consuming and not highly effective.

This article contains two sections. The first section gives a general overview of the Intel instruction format, while the second part gives the encoding details of each common Intel instruction. The first section contains the background information necessary to understand the second part, while the second part is meant to be more of a reference.

An important distinction between DOS and Windows applications is the size of a machine word. Intel designed the original 8086 opcodes with 16-bit computing in mind. As a result, they use a single bit to distinguish between 16-bit operands and 8-bit operands. Because one bit has two possible values, this forces the newer 32-bit processor mode to have only two operand sizes as well. 32-bit mode changes the meaning of the size bit to distinguish between 32-bit operands and 8-bit operands. As a result, the size of a "small" operand is 8 bits in both modes, but the size of a "large" operand depends on the mode. Under DOS, the CPU runs in legacy 16-bit mode. This means that the default size of a "large" operand is 16 bits. Windows, however, runs in 32-bit mode, making the default size of a "large" operand 32 bits. To keep things simple, this article uses the term "word" to mean the size of a large operand. If your application runs under DOS, a "word" is 16 bits, but if your application runs under Windows, a "word" is 32 bits.

Intel Instruction Format

Although Intel instructions vary in size from one byte up to fourteen bytes, all Intel instructions have the same six-part structure. Understanding the purpose of each part is the first step to learning the sizes of the different Intel instructions. The parts of an Intel-format instruction are listed below, in the order that they appear in the instruction:

- Prefixes: 0-4 bytes
- Opcode: 1-2 bytes
- ModR/M: 1 byte
- SIB: 1 byte
- Displacement: 1 byte or word
- Immediate: 1 byte or word

Except for the opcode, all of these parts are optional. They are only present when the particular instruction requires them. Simple instructions such as NOP require just the opcode. Complicated instructions, such as ADD [ES: my_data+EBX+ESI*8], WORD 1003H, require all of the fields. The following paragraphs explain how and when each instruction field is used.

Prefixes

The optional prefixes are the first part of an Intel instruction. These prefixes modify the instruction's behavior in several different ways. Prefixes can change the default segment of an instruction, override the default size of the machine-word, control looping in string instructions, and control the processor's bus usage. Each prefix adds one byte to the instruction. An instruction can have one prefix from each of the four prefix groups, for a maximum of four prefix bytes:

- Group 1: LOCK, REPE/REPZ, REP, REPNE/REPNZ
- Group 2: CS, DS, ES, FS, GS, SS, Branch hints
- Group 3: Operand-size override (16 bit vs. 32 bit)
- Group 4: Address-size override (16 bit vs. 32 bit)

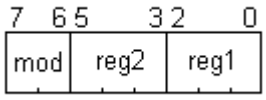
Opcode

The operation code, or opcode, comes after any optional prefixes. The opcode tells the processor which instruction to execute. In addition, opcodes contain bit fields describing the size and type of operands to expect. The NOT instruction, for example, has the opcode 1111011w. In this opcode, the w bit determines whether the operand is a byte or a word. The OR instruction has the opcode 000010dw. In this opcode, the d bit determines which operands are the source and destination, and the w bit determines the size again. Some instructions have several different opcodes. For example, when OR is used with the accumulator register (AX or EAX) and a constant, it has the special space-saving opcode 0000110w, which eliminates the need for a separate ModR/M byte. From a size-coding perspective, memorizing exact opcode bits is not necessary. Having a general idea of what type of opcodes are available for a particular instruction is more important.

Not all opcodes are the same size. The original instructions from the 8088 have one-byte opcodes, while new instructions since the 386 generally have two-byte opcodes. Some SSE instructions even have three-byte opcodes. This is because the size of a byte limits the number of possible opcodes. As Intel runs out of unused opcodes, the only way to add more instructions is to give them opcodes larger than one byte.

ModR/M

If the instruction requires it, the ModR/M byte comes after the opcode. This byte tells the processor which registers or memory locations to use as the instruction's operands. The byte has the following structure:



Both the reg1 and reg2 fields take three-bit register codes, indicating which registers to use as the instruction's operands. By default, reg1 is the source operand and reg2 is the destination. Some opcodes, such as the OR opcode mentioned above, contain a direction bit which overrides this default. Other instructions require a single operand. If an instruction requires only one operand, the unused reg2 field holds extra opcode bits rather than a register code. This is especially true for floating-point instructions, which use ST(0) as their implied destination.

The mod field determines the meaning of the reg1 field. It can have the following possible values:

Code	Assembly Syntax	Meaning
00	[reg1]	The operand's memory address is in reg1.
01	[reg1 + byte]	The operand's memory address is reg1 + a byte-sized displacement.
10	[reg1 + word]	The operand's memory address is reg1 + a word-sized displacement.
11	reg1	The operand is reg1 itself.

The meaning of reg1 field becomes more complicated in 16-bit mode. When mod specifes a memory address (mod = 00, 01, or 10), reg1 does not contain a simple register code. Instead, it specifies one of the following register combinations:

Code	Register Combination
000	BX + SI
001	BX + DI
010	BP + SI
011	BP + DI
100	SI
101	DI
110	BP
111	BX

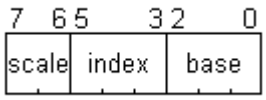
Both 16-bit and 32-bit modes have an additional complication. In the system above, ModR/M provides no obvious way to specify a fixed memory location as an operand. All of the combinations for mod and reg1 include a register as part of the memory address. To fix this problem, Intel arbitrarily defines the combination mod = 00, reg = BP / EBP to mean that the address of the operand is a simple [word] displacement. Because the codes for [BP] and [EPB] have this new meaning, there is no simple way to access memory given by the base pointer register. When the assembler sees one of these operands, it automatically creates the form [BP+00] or [EBP+00], which requires an additional displacement byte.

Finally, 32-bit mode has its own complication. When mod indicates a memory address (mod = 00, 01, or 10) and when reg1 indicates the ESP register, an additional byte follows the ModR/M byte. This byte, called the SIB byte, is used instead of reg1 to determine the operand's memory address. The structure of the SIB byte is discussed later.

Not all opcodes require the ModR/M byte. Some instructions, such as AAM, have fixed sources and destinations. Other instructions, such as PUSH and POP, encode the source or destination directly into the opcode. Knowing which instructions need a ModR/M byte and which instructions do not is the hardest part of learning Intel instruction sizes.

SIB

When ModR/M contains the correct mod and reg1 combination, a SIB byte follows the ModR/M byte. SIB is an acronym which stands for Scale*Index+Base. It is a powerful addressing format available only in 32-bit mode. In SIB, the combination of two registers and a scaling factor replaces reg1 in the operand's address. The SIB byte's format is shown below:



In the SIB byte, both index and base are three-bit register codes, and scale is a two-bit number. To compute the SIB value, the processor uses the following formula: (index * 2^scale) + base. (Obviously, the processor uses a bit shift to perform the power-of-two multiplication.) Once the processor finds the SIB value, it uses it in place of the ModR/M byte's reg1 value in the memory address computation.

The SIB byte enables complicated addresses such as [ebx*4 + esi + my_table]. For this example, the ModR/M and SIB bytes' fields have the following values:

- ModR/M.mod = 10 (In other words, the mode is [reg1 + word].)
- ModR/M.reg2 = Whatever (Usually the destination register, but depends on the opcode.)
- ModR/M.reg1 = ESP (Intel redefines ESP's code to mean SIB in 32-bit memory addresses.)
- SIB.scale = 2 (Because 2^2 = 4)
- SIB.index = EBX
- SIB.base = ESI

The SIB byte is ordinarily not present. It is only needed when an instruction uses the Scale*Index+Base addressing format.

Displacement → it is a part of a operand

When the mod is either 01 or 10, a displacement is part of the operand's address. This displacement comes immediately after the ModR/M and optional SIB byte. Dending on the mod field, the displacement is either a byte or a word.

For example, here is the full machine code for the 32-bit instruction OR EAX, [ECX + EDX*2 + 406080A0h]:

Opcode	ModR/M	SIB	Displacement
00001011	10 000 100	01 010 001	10100000 10000000 01100000 01000000

In 32-bit mode, a word-sized displacement takes four bytes. This is an enormous amount of space. When an instruction contains a four byte displacement, it is usually a good idea to look at other forms of addressing the may be smaller, such as using the stack, or a register plus a smaller displacement.

Immediate → it is a operand

If an instruction uses an immediate value as an operand, such as ADD AX, 0xF00F, the immediate value is the last part of the instruction. Like addressing displacements, immediates can be either a byte or a machine word.

To illustrate, here is the machine code for the 16-bit instruction, AND SI, 0420h:

Opcode	ModR/M	Immediate
10000001	11 100 110	00100000 00000100

Just with addressing displacements, a 32-bit word-sized immediate requires a huge amount of space. Big immediates usually compress better than displacements, however, because immediates usually contain more zero bytes.

Detailed Instruction Encodings

Directly memorizing Intel instruction sizes is not really possible, because an instruction's size depends on its operands. Instead, it is better to memorize which fields an instruction contains. By adding the sizes of the different fields, finding the instruction's size is easy. This section lists the opcode sizes, ModR/M requirements, and literal sizes of the common Intel instructions.

Integer Instructions

For simplicity, this section is organized as a table. The first column of the table lists the instructions in alphabetical order. The second column shows the different combinations of operands each instruction can take, while the third column shows the fields required to encode each combination. The table uses the following abbreviations:

- m - memory
- r - register
- * - memory or register
- i - immediate
- disp - displacement
- ac - accumulator (AL, AX, or EAX)
- cc - condition code
- op - one opcode byte
- mod - ModR/M [+ optional SIB] [+ optional disp]

To show the size of each operand, the following suffixes are used:

- b - byte
- w - machine word
- 1, 2, 3, 4, 6, 8 - number of bytes

If the table does not show the size of some operands, the operands can be either a byte or a word, as long as they are the same size. This is because opcodes use a size bit to determine operand sizes.

Instruction	Operands	Encoding
AAA	none	op
AAD	none	op i.b
AAM	none	op i.b
AAS	none	op
ADC	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
AND	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
ADD	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
BOUND	r.w, m.w	op mod
BSF	r.w, *.w	op op mod
BSR	r.w, *.w	op op mod
BSWAP	r.w	op op mod
BT	*.w, r.w *.w, i.b	op op mod op op mod i.b
BTC	*.w, r.w	op op mod

	*.w, i.b	op op mod i.b
BTR	*.w, r.w *.w, i.b	op op mod op op mod i.b
BTS	*.w, r.w *.w, i.b	op op mod op op mod i.b
CALL	disp.w *.w	op disp.w op mod
CBW	none	op
CDQ	none	op
CLC	none	op
CLD	none	op
CLI	none	op
CMC	none	op
CMOVcc	*.w, *.w	op op mod
CMP	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
CMPS	none	op
CMPXCHG	*, r	op op mod
CMPXCHG8B	m.8	op op mod
CPUID	none	op op
CWD	none	op
CWDE	none	op
DAA	none	op
DAS	none	op
DEC	* r.w	op mod op
DIV	*	op mod
ENTER	i.16, i.8	op i.3
HLT	none	op
IDIV	*	op mod
IMUL	* r.w, *.w r.w, i r.w, *.w, i	op mod op op mod op mod i op mod i
IN	ac, i.b ac, DX	op i.b op
INC	* r.w	op mod op
INS	none	op

INT	i.b 3	op i.b op
INTO	none	op
IRET	none	op
Jcc	disp.b disp.w	op disp.b op op disp.w
JCXZ	disp.b	op disp.b
JMP	disp *.w	op disp op mod
LAHF	none	op
LDS	r.w, m.w	op mod
LEA	r.w, m	op mod
LEAVE	none	op
LES	r.w, m.w	op mod
LFS	r.w, m.w	op mod
LGS	r.w, m.w	op mod
LSS	r.w, m.w	op mod
LODS	none	op
LOOP	disp.b	op disp.b
LOOPZ	disp.b	op disp.b
LOOPNZ	disp.b	op disp.b
MOV	*, * *, i r, i ac, [disp.w]	op mod op mod i op i op disp.w
MOVS	none	op
MOVSX	r.w, *.b	op op mod
MOVZX	r.w, *.b	op op mod
MUL	*	op mod
NEG	*	op mod
NOP	none	op
NOT	*	op mod
OR	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
OUT	ac, i.b OUT ac, DX	op i.b op
OUTS	none	op

POP	* r FS GS	op mod op op op op op
POPA	none	op
POPF	none	op
PUSH	* r i FS GS	op mod op op i op op op op
PUSHA	none	op
PUSHF	none	op
RCR	*, 1 *, CL *, i.b	op mod op mod op mod i.b
RCL	*, 1 *, CL *, i.b	op mod op mod op mod i.b
RET	none i.2	op op i.2
ROL	*, 1 *, CL *, i.b	op mod op mod op mod i.b
ROR	*, 1 *, CL *, i.b	op mod op mod op mod i.b
SAHF	none	op
SAL	*, 1 *, CL *, i.b	op mod op mod op mod i.b
SAR	*, 1 *, CL *, i.b	op mod op mod op mod i.b
SBB	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
SCAS	none	op
SETcc	*.b	op op mod
SHL	*, 1 *, CL *, i.b	op mod op mod op mod i.b
SHLD	*.w, r.w, CL *.w, r.w, i.b	op op mod op op mod i.b
SHR	*, 1 *, CL *, i.b	op mod op mod op mod i.b

SHRD	*.w, r.w, CL *.w, r.w, i.b	op op mod op op mod i.b
STC	none	op
STD	none	op
STI	none	op
STOS	none	op
SUB	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i
TEST	*, r *, i ac, i	op mod op mod i op i
WAIT	none	op
XADD	*, r	op op mod
XCHG	*, r ac, r	op mod op
XLAT	none	op
XOR	*, * *, i *.w, i.b ac, i	op mod op mod i op mod i.b op i

Many instructions in the above list have special space-saving opcodes that do not require an additional ModR/M byte. These instructions are:

- DEC, INC, POP, or PUSH used with a word-sized general register.
- ADC, ADD, AND, CMP, OR, SBB, SUB, TEST, or XOR used with the accumulator and an immediate.
- MOV used with any general register and an immediate.
- MOV used with the accumulator and a simple word displacement.
- XCHG used with the accumulator and a word register.

To save space, the binary arithmetic instructions ADC, ADD, AND, CMP, OR, SBB, SUB, and XOR can use a byte-sized immediate with a word-sized destination. To do this, these instructions first sign-extend the literal to the destination's size before using it in the operation. This is especially valuable for 32-bit code, since it saves three bytes per instruction. Unfortunately, NASM, a fairly popular assembler, does not use the sign-extension encoding by default. To use this encoding, prefix the immediate with the BYTE keyword.

Two notable instructions in the above list are AAD and AAM. In the old Intel manuals, these instructions have two-byte opcodes. New Intel manuals now show these instructions with one-byte opcodes followed by an immediate equal to 0x0A. The AAD instruction multiplies AH by the immediate and adds the product to AL. AAM divides AL by the immediate, and stores the remainder in AL and the quotient in AH. It is possible to change the value of the immediate byte by coding the instructions in machine language, creating two new, nameless instructions for quickly dividing and multiplying a byte by a constant. The opcode for AAD is 0xD5, and the opcode for AAM is 0xD4.

Note also that ENTER, CALL FAR, and JMP FAR, and RET's immediate form are exceptions to the rule that an instruction's displacement and immediate literals must be either a byte or a word. ENTER takes a three-byte immediate, while CALL FAR and JMP FAR take either four-byte or six-byte displacements, depending on whether the processor is in 16 or 32-bit mode. The immediate form of RET requires a two-byte literal, regardless of the machine word's size.

Floating Point Instructions

For historical reasons, all floating-point instructions have a one-byte opcode followed by a ModR/M byte. If a floating point instruction does not access memory, the entire ModR/M byte holds opcode bits, so the encoding is effectively a two-byte opcode.

The original PC processor, the 8088, did not contain floating-point instructions. An optional math coprocessor, the 8087, provided floating point support for the 8088. To communicate with the math coprocessor, the 8088 contained eight escape instructions with ModR/M bytes. When the main processor received an escape instruction, it read the memory identified by the ModR/M byte and then performed a no-operation. Meanwhile, the math coprocessor recorded the contents of the escape opcode, the ModR/M byte, and the address of the memory read. The math coprocessor used the escape opcode and the ModR/M byte to determine the operation to perform, and used the memory address as the operation's target.

All Intel processors since the 486 have integrated floating-point units, so they no longer use the escape mechanism. Nevertheless, the instruction format of the 8087 remains.

Other Instructions

MMX instructions all have two-byte opcodes plus a ModR/M byte, except for shift-by-constant instructions, which include a one-byte immediate as well.

People who plan to use SSE or 3DNow! are probably code gurus already, so they can look up the operation sizes themselves.

There are many issues related to size-optimizing code; writing an article on all of them is impossible. Hopefully, understanding the sizes of Intel instructions provides a useful basis for discovering and better understanding these optimizing techniques.

May the Source be with you.