2.6.5. Machine instructions representation

A x86 machine instruction represents a sequence of 1 to 15 bytes, these values specifying an operation to be run, the operands to which it will be applied and also possible supplementary modifiers.

A x86 machine instruction has maximum 2 operands. For most of the instructions, they are called *source* and *destination* respectively. From these two operands, only one may be stored in the RAM memory. The other one must be either one **EU** register, either an integer constant. Therefore, an instruction has the general form:

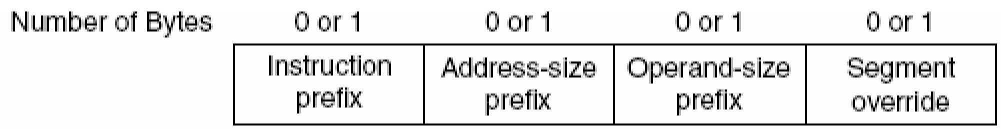
instruction_name *destination**,** source*

The internal format of an instruction varies between 1 and 15 bytes, and has the following general form (*Instructions byte-codes from OllyDbg)*:

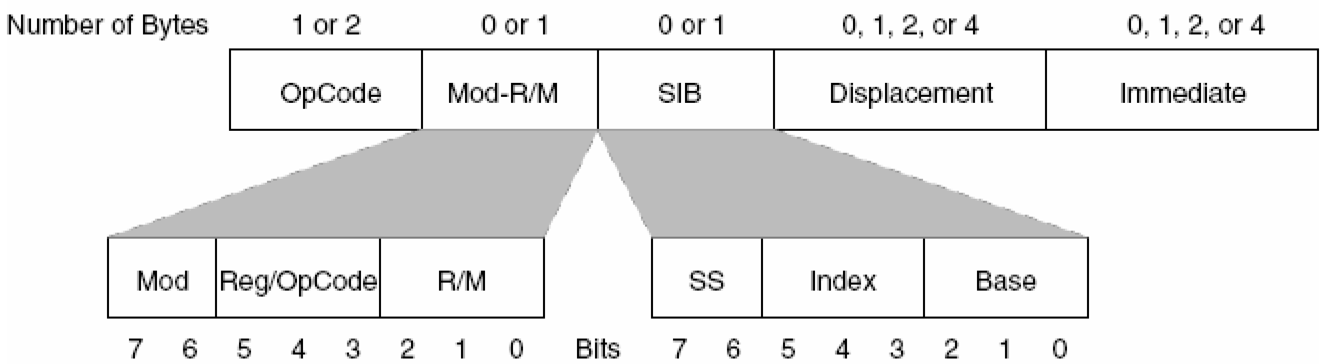
[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

The *prefixes* control how an instruction is executed. These are optional (0 to maxim 4) and occupy one byte each. For example, they may request repetitive execution of the current instruction or may block the address bus during execution to not allow concurrent access to operands and results.

The operation to be run is identified by 1 to 2 bytes of *code* (opcode), which are the only mandatory bytes, no matter of the instruction. The byte *ModeR/M* (register/memory mode) specifies for some instructions the nature and the exact storage of operands (register or memory). This allows the specification of a register or of a memory location described by an offset.

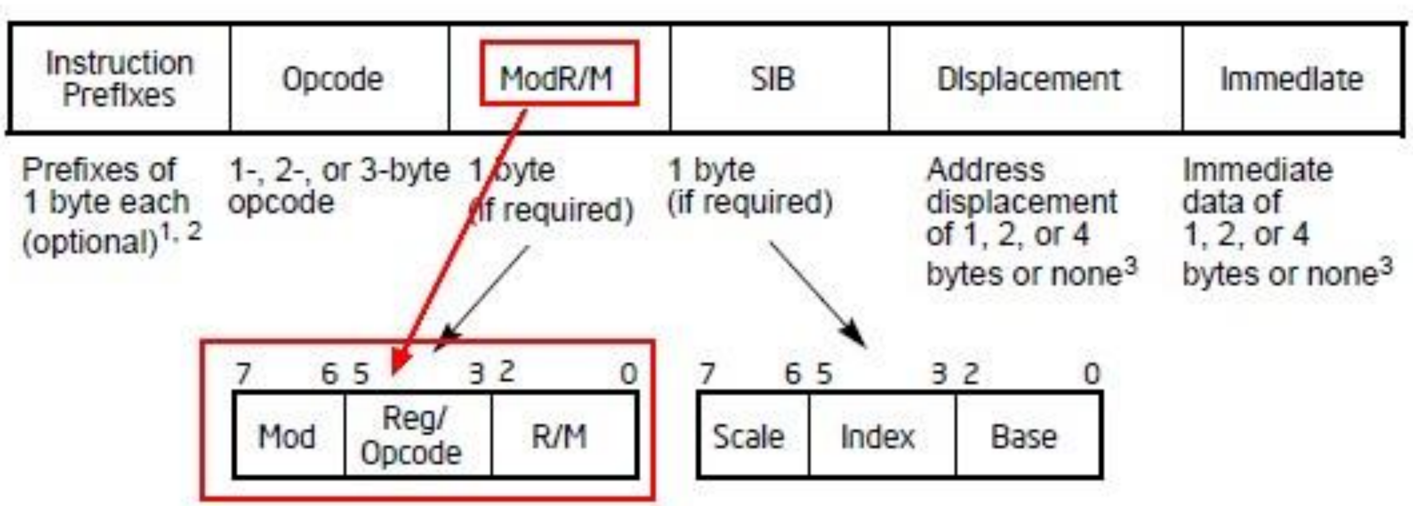


(a) Optional instruction prefixes



(b) General instruction format

Although the diagram seems to imply that instructions can be up to 16 bytes long, in actuality the x86 will not allow instructions greater than 15 bytes in length.



For more complex addressing cases than the one implemented directly by ModeR/M, combining this with SIB byte allows the following formula for an offset (*http://datacadamia.com/intel/modrm*):

[base] + [index × scale] + [constant]

(SIB) (displacement+immediate)

where for base and index the value of two registers will be used and the scale is 1, 2, 4 or 8. The allowed registers as base or/ and as indexes are: EAX, EBX, ECX, EDX, EBP, ESI, EDI. The ESP register is available as base but cannot be used as index (http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0100_sib_byte_layout).

Most of the instructions use for their implementation either only the opcode or the opcode followed by ModeR/M.

The *displacement* is present in some particular addressing forms and it comes immediately after ModeR/M or SIB, if SIB is present. This field can be encoded either on a byte or on a doubleword (32 bits).

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. **The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location**. The displacement-only addressing mode is perfect for accessing simple scalar variables. Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. **On the 80x86 processors, this displacement is an offset** from the beginning of memory (that is, address zero).

Displacement mode, **the operand's offset** is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. **Displacement addressing can be useful for referencing global variables.**

As a consequence of the impossibility of appearing more than one ModeR/M, SIB and displacement fields in one instruction, the x86 architecture doesn't allow encoding of two memory addresses in the same instruction.

With the *immediate value* we can define an operand as a numeric constant on 1, 2 or 4 bytes. When it is present, this field appears always at the end of instruction.

2.6.6. FAR addresses and NEAR addresses.

To address a RAM memory location two values are needed: one to indicate the segment and another one to indicate the offset inside that segment. For simplifying the memory reference, the microprocessor implicitly chooses, in the absence of other specification, the segment's address from one of the segment registers CS, DS, SS or ES. The implicit choice of a segment register is made after some particular rules specific to the used instruction.

An address for which only the offset is specified, the segment address being implicitly taken from a segment register is called a *NEAR address*. A NEAR address is always inside one of the 4 active segments.

An address for which the programmer explicitly specifies a segment selector is called a *FAR address.* So, a FAR address is a COMPLETE ADDRESS SPECIFICATION and it may be specified in one of the following 3 ways:

- $s_3s_2s_1s_0$: *offset_specification* where $s_3s_2s_1s_0$ is a constant;
- segment register: offset_specification, where segment registers are CS, DS, SS, ES, FS or GS;
- FAR [variable], where variable is of type QWORD and contains the 6 bytes representing the FAR address.

The internal format of an FAR address is: at the smallest address is the offset, and at the higher (by 4 bytes) address (the word following the current doubleword) is the word which stores the segment selector.

The address representation follows the little-endian representation presented in Chapter 1, paragraph 1.3.2.3: the less significant part has the smallest address, and the most significant one has the higher address.

2.6.7. Computing the offset of an operand. Addressing modes.

For an instruction there are 3 ways to express a required operand:

- *register mode*, if the required operand is a register;   mov eax, 17
- *immediate mode*, when we use directly the operand's value (not its address and neither a register holding it); mov eax, 17
- *memory addressing mode*, if the operand is located somewhere in memory. In this case, its offset is computed using the following formula:

$$\boldsymbol{\mathit{offset\_address}} = \textbf{[base] + [index} \times \textbf{scale] + [constant]}$$

So *offset_address* is obtained from the following (maximum) four elements:

- the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI, EDI or ESP as base;
- the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI or EDI as index;
- scale to multiply the value of the index register with 1, 2, 4 or 8;
- the value of a numeric constant, on a byte or on a doubleword.

From here results the following modes to address the memory:

- *direct addressing*, when only the *constant* is present;
- *based addressing*, if in the computing one of the base registers is present;
- *scale-indexed addressing*, if in the computing one of the index registers is present.

These three mode of addressing could be combined. For example, it can be present direct based addressing, based addressing and scaled-indexed etc.

A non direct addressing mode is called ***indirect addressing*** (based and/or indexed). So, an indirect addressing is a one for which we have at least one register specified between squared brackets.

In the case of the jump instructions another type of addressing is present called *relative* addressing.

Relative addressing indicates the position of the next instruction to be run relative to the current position. This ”distance“ is expressed as the number of bytes to jump over. The x86 architecture allows relative SHORT addresses, described on a byte and having values between -128 and 127, but also relative NEAR addresses, represented on a doubleword with values between -2147483648 and 2147483647.

Jmp Below2 ; this instruction will be translated into (see OllyDbg) usually in something as **Jmp [0084]↓**

………….

………….

Below2:

Mov eax, ebx