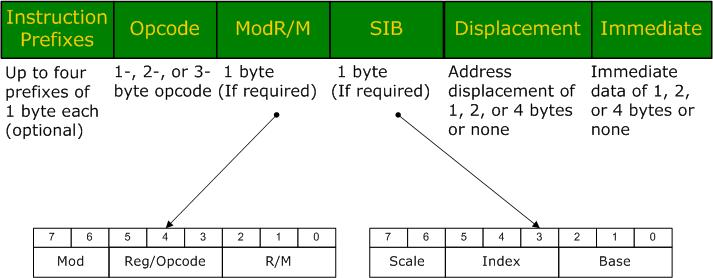
intel的确切的指令格式:

****

opcode 编码了进行什么样的操作，跟汇编格式里面的mnemonic 对应，CPU 知道了什么操作之后就会寻找操作的对象，是寄存器还是内存？ModR/M 部分就给出了操作的对象，R 是register，M 是memory，而Mod 指示了到底是寄存器还是内存。如果ModR/M的字节数足够大的话，那么或许就不需要后面的两个部分了，实际上ModR/M 只有一个字节，能编码所有的寄存器，却不能编码所有的内存寻址模式，intel 使用后面两个部分来辅助ModR/M 完成确切的内存定位——SIB 和displacement.

一. Prefixes

Prefixes的几个特性

1. 它是唯一的一个可能出现在code之前的域。

2. 所有的Prefixes都只有1个字节。

3. 在一个OpCode中可能会有多个Prefixes。如果有多个Prefixes，那么它们的顺序可以打乱，不会有任何问题。

4. 如果Prefixes不能对随它之后的OpCode起作用，那么它就会被忽略。

分类：

1. Change DEFAULT operand size. (66)

prefix 66的作用是切换默认的操作数大小。如果默认的操作数大小是WORD（16位），那么切换后就是DWORD（32位）；反之，如果默认的操作数大小是DWORD（32位），那么切换后就是WORD（16位）。

2. Change DEFAULT address size. (67)

现在把它的OpCode改成以67开头的：

67 8A 00 MOV AL, [BX+SI]

3. Repeat prefixes. (F2, F3)

F2: REPNE

F3: REP / REPE

Repeat Prefix 结束条件1 结束条件2

REP ECX=0 None

REPE ECX=0 ZF=0

REPNE ECX=0 ZF=1

4. Segment override prefixes(change DEFAULT segment). (2E, 36, 3E, 26, 64, 65)

Prefix Explanation

2E CS segment override prefix

36 SS segment override prefix

3E DS segment override prefix

26 ES segment override prefix

64 FS segment override prefix

65 GS segment override prefix

再来看一个例子：

8B 03 MOV EAX, [DWORD DS:EBX]

65 8B 03 MOV EAX, [DWORD GS:EBX]

指令由不同的定义被划分为不同的组，每个组各自有一个默认的段：

CS: for EIP pointer

ES: 目的操作数是内存单元的串指令（movs, cmps等），在这里源操作数是储存在段DS里面。

SS: 堆栈操作（push, pop等）

DS: 剩下的数据操作指令。

65就是一个Segment override prefix，用来改变默认的段，从上表中我们可以看出：65代表的是段GS。

5. LOCK prefix. (F0)

F0表示的是助记符LOCK就足够了

二. OpCode

Intel 通过操作码大小指令改写前缀来对操作对象在16 位和32 位之间进行转换，而使用这里要说的w 位进行8 位和上述两种长度的区分。

16/32 位： 89 1000 100[1]

8 位： 88 1000 100[0]

d 位值的涵义是很明显的，当d = 1 时，源操作数为R/M 目的操作数为Reg/Opcode；当d = 0时，源操作数为Reg/Opcode 而目的操作数为R/M。可以用d 位来进行区分的指令范围也很明显——指令的操作对象中必须有ModR/M 编码的两个操作对象

00401019 89D9 mov ecx, ebx

0040101B 8BD9 mov ebx, ecx

0040101D 890B mov dword ptr [ebx], ecx

0040101F 8B0B mov ecx, dword ptr [ebx]

s位

(1)初始状态 eax=00,00,00,00

          输入机器码:66,81,C0,F8  (debug32默认是16位模式，所以要用66前缀）

      对应的汇编: add eax,f8

         opcode = 81 = 100000[0]1   (opcode里的s位等于0,按0扩展)

         执行后   eax=00,00,00,F8

     (2)初始状态 eax = 00,00,00,00

         输入机器码:66,83,C0,F8

         对应的汇编:add eax,f8

        opcode = 83 = 100000[1]1 (opcode里的s位等于1)

        执行后   eax = FF,FF,FF,F8  (opcode里的s位等于1，按符号扩展)

一切似乎都很顺利的验证了s位的功能。但当我想用nasm来做实验的时候，问题出现了，nasm为add所生成的机器码好像根本都不采用带符号扩展的add指令，而生成的机器码都是不带符号扩展的add指令。后来我才想明白，对于cpu来说，opcode=83的add指令是你传入一个单字节的数据让cpu自动为你扩展到32位，而对nasm来说完全没有必要让cpu去自动扩展，它的扩展是在编译期就实现了的。所以对汇编编译器来说，完全可以无视opcode=83的指令，甚至是所有s位等于1的指令格式都可以无视？值得注意的是opcode=81的add指令后面跟的是32位的立即数，opcode=83的add指令后面跟的是8位立即数。

三. ModR/M和SIB

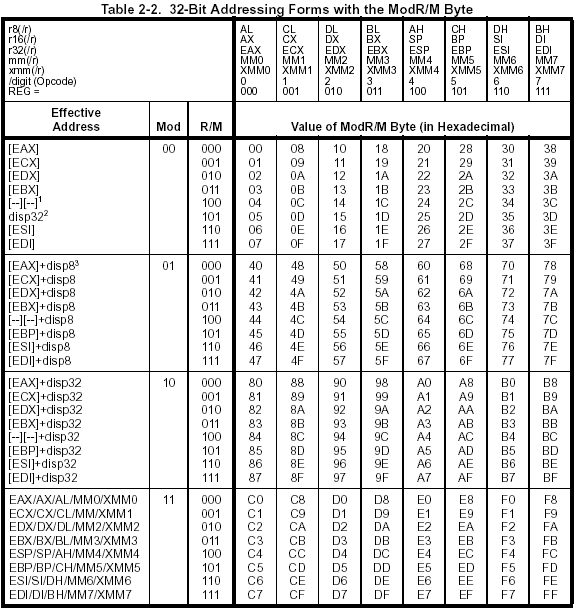
ModR是由Mod、Reg/Opcode和R/M三个部分组成的。每个部分所占的bit大小为：

****

Mod: 占最高位的6～7共2个bit

Reg/Opcode: 占中间位的3～5共3个bit

R/M: 占最低位的0～2共3个bit

****

**寻址方式：**

1:寄存器寻址:

　　例子：movl %eax, %ebx #mod为11b 2个寄存器分别存放于modr/m里的reg/opcode和r/m,存放顺序由上面的d位决定

2.立即数寻址

例子：movl $2, %ebx #$2存放于immediate，寄存器按照具体指令定

内存寻址方式：segreg:disp(base,index,scale)：base+index\*scale+dsp scale = 2, 4, 8，其中index/scale/disp/segreg全部是可选的，完全可以简化掉。

segreg:段寄存器

disp：偏移，为上述的displacement

index：索引，为上述sib里的index

scale：级数，为上述sib里的scale，缺省为1

base：基地址，下列方式3为modr/m里的r/m,4,5为sib的base里

　　3:寄存器间接寻址方式:

　　例子: movl (%ebx), %eax #mod 为00b %eax也可以是常数(当r/m为101时候) movl (2000h), %eax

　　把地址为ebx的变量赋值给eax

　　4:基址变址寻址方式:

　　例子: movl offset(%eax), %ebx #mod为10或者01的时候 偏移32为10b

　　把地址为ea+offset便宜的变量赋值给ebx

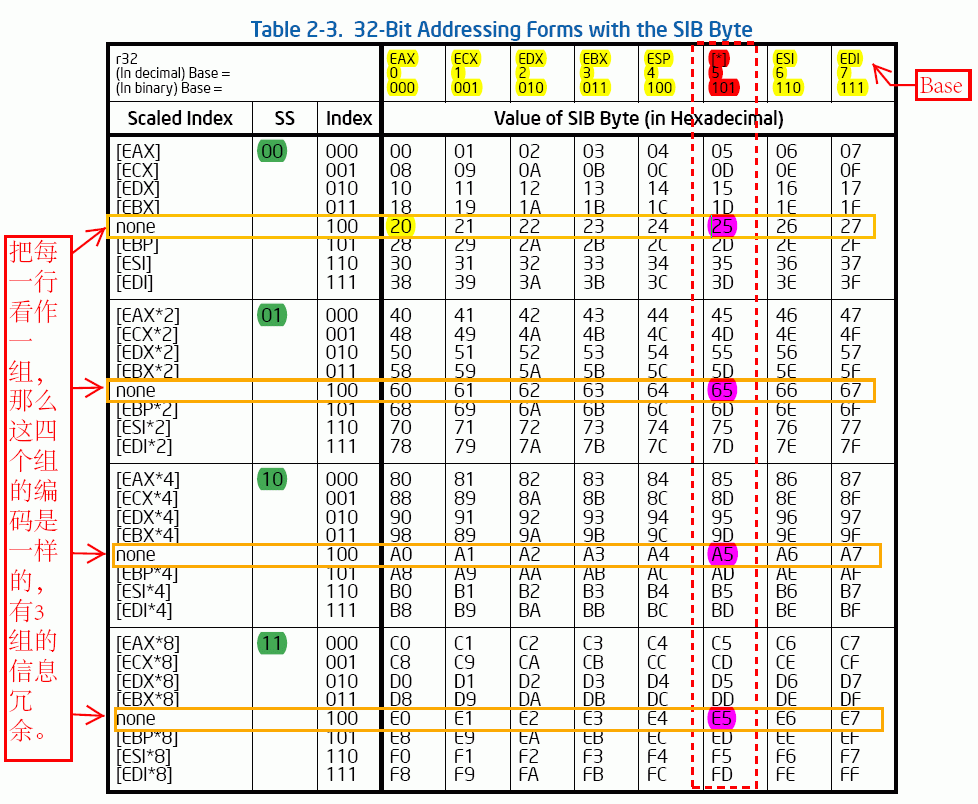
SIB寻址方式：

[base + index \* n + displacement]，n = 2, 4, 8，displacement是后续字节

base和index分别为8个通用寄存器的一种，则各需要3个bit来进行编码，剩余的只有2个bit，显然用来编码n的值。2个bit也只能编码4个值，这恐怕就是为什么n只能取(1), 2, 4, 8的原因了。剩下的问题就是这些bit是如何在SIB这个字节中划分的，看看下图：

****

我们可以分析一下，8个base编码 \* 8个Index编码 \* 4个n = 256SIB寻址方式，Intel完完全全地利用SIB中的每一个bit进行编码。Intel把这256中寻址方式编码到一张表格中，看看这张表格：

****

针对sib方式的特殊情况：为了能实现 寄存器间接寻址方式的2种：寄存器和常数

a. 在sib下 index=100b时 表示index为0所以可以实现 movl (%eax), %ebx包括含有offset的是由displacement决定

b. 在sib下 mod = 00b， base = 101b base为0所以可以实现 movl (2000h), %ebx

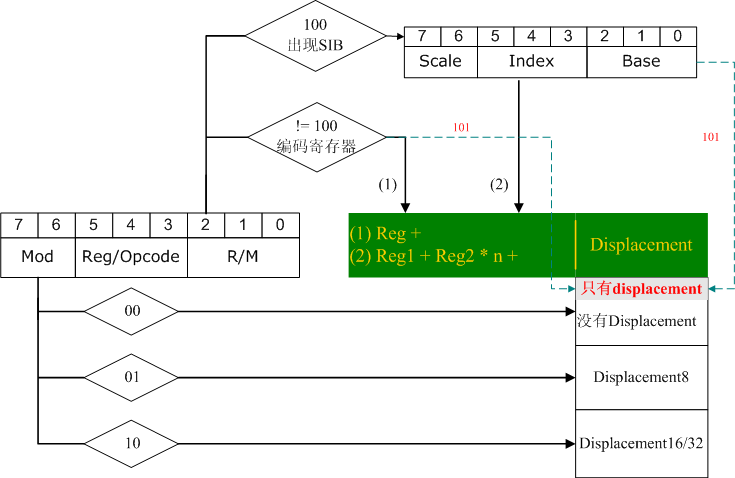
5. 基址索引寻址方式（sib）

例子：movl offset(%eax, %ebx, n), %ecx #mod不为11b且r/m为100b的时候

把地址为eax+ebx\*n+offset的变量传递给ecx

**ModR/M、SIB、Displacement之间的关系**

ModR/M和SIB的每个bit的确切涵义已经总结过了。但是这里还是把它们之间的关系整理一下（只考虑内存寻址）：

****

（1）Mod决定了结果中displacement是否出现。Mod为00的时候，是没有displacement的（或者整条指令中只有一个displacement），Mod为01时dispacement为8位，通过符号扩展的方式加到前面的寄存器上，Mod为10时，偏移地址为16位/32位，取决于CPU当前工作模式。

    （2）R/M部分的编码决定了是直接编码寄存器，还是通过SIB的方式编码寄存器。

    （3）特殊情况1，编码寄存器（无论是直接编码还是通过SIB的方式编码）的时候有个特殊情况，那就是，当Mod = 00并且用来编码基址寄存器的值为101(R/M = 101或者SIB.Base = 101)。整条指令中只有一个16/32位的偏移地址。

（4）特殊情况2，在SIB下index=100b时 表示index为0所以可以实现 movl (%eax), %ebx包括含有offset的是由displacement决定