The Svin 的

OpCode 教程

说明:

1. 这篇教程原文分为 8 帖,它们的地址分别为:

Opcode #1 http://www.asmcommunity.net/board/index.php?topic=8963

Opcode #2 http://www.asmcommunity.net/board/index.php?topic=8967

Opcode #3 http://www.asmcommunity.net/board/index.php?topic=8982

Opcode #4 http://www.asmcommunity.net/board/index.php?topic=9062

Opcode #5 http://www.asmcommunity.net/board/index.php?topic=9063

Opcode #6 http://www.asmcommunity.net/board/index.php?topic=9741

Opcode #7 http://www.asmcommunity.net/board/index.php?topic=10554

Opcode #8 http://www.asmcommunity.net/board/index.php?topic=14153(无实质内容,未翻译)

2. 原帖并没有标题,标题是翻译的时候加上去的.
3. 由于原作者是俄罗斯人,英语不是太好(有语法错误),有些语句我是根据自己的理解翻译过来的.
4. 不知道有前辈翻译过这个没有?至少我在 google 上没有找到.
5. 希望这个对你有用.
6. 感谢朋友 Gyf 所做的校对.
7. 错误之处在所难免，敬请指正。bottlexx@163.com

Xiep

2008.08.19

目录

一 概览

这本书为那些像本人一样渴求知识而不得的人而写.

这套教程的风格是通过练习得出结论,而不是依据结论来做练习 .换句话说,这篇教程包含很多的例子和练习.深入学习本教程理论最好的方法是做所有的练习,以及试验所有的例子.

这本教程并不讲汇编语言编程,我们讲的是"OpCode".

What "OpCode" is?

这是课程要解决的主要问题,现在我给一个简短的回答.

当我们在源代码中写入"lodsb",在编译阶段汇编器(如 ml.exe)遇到"lodsb",它会在可执行文件(或.obj 文件)中用一个字节 ACh 来替代它.

ACh 就是所谓的 OpCode,"lodsb"则是助记符(mnemonic).

助记符"lodsb"对汇编器 ml.exe 说,"给我用字节 ACh 替换 lodsb".处理器并不知道 lodsb 是什么.当寄存器 EIP 指向 ACh 字节时,处理器解码器对字节 ACh 解码,通过这个字节处理器就知道程序要将寄存器 ESI 指向的一个字节的内容送入 AL 寄存器.

OpCode 就是如此简单.

你可能会说:一个助记符是某个特定的操作码的别称.

但是,事实并没有如此简单.对于"lodsb"这个特定的助记符来说,它确实对应 ACh,并且操作码 ACh 也唯一确定助记符"lodsb".然而并非所有的情形都是这样.我们很快会见到"为什么不"的例子.

就像现实生活中一个名称往往并不唯一对应一个事物一样,OpCode 和助记符的关系如下:**不同的 OpCode 可能有同样的助记符; 一个 OpCode 可能有几个助记符对应.有些事情可能很简单但却会吓倒初学者,在不久的将来我将会做详细的解释.**

查看 OpCode 或 mnemonics 的方法有很多种,我认为最快且最好的方法是使用调试器.我将在 OllyDbg 中试验所有的例子.

Exersize1. 在调试器中输入助记符和 OpCode

1. 用 OD 打开任意一个 Win32 可执行文件,加载完成后在代码窗口中双击某一行,在弹出的指令输入对话框中输入 lodsb 回车,你将看到(类似下面的内容):

   0040108C> AC LODS [BYTE DS:ESI]

   第一列(0040108c)是指令在内存中的地址,第二列(AC)是 OpCode,第三列(LODS [BYTE DS:ESI])是 OpCode AC 的助记符.

2. 你可以试着输入一些你其他的助记符,并观察他们的 OpCode.
3. 按 CTRL + e,你将看到一个对话框,在 HEX + 00 对应的文本框中输入 AC 回车,可以看到第二列和第三列如下:

   AC LODS [BYTE DS:ESI]

4. 试着输入一些其他的 OpCode,并观察对应的助记符.

Exersize2. 一个助记符对应一个 OpCode 吗?

1. 按 CTRL+ e,输入 OpCode 90,回车,可以看到 OD 将 90 认为是 NOP:

0040108E 90 NOP

2. 输入助记符 "NOP",同样看到:

0040108E 90 NOP

3. 输入助记符 XCHG EAX, EAX

糟糕! OllyDbg 并没有插入我们的指令,而是用 NOP 代替!!!

不用着急,这是 OllyDbg 的问题,当看到 OpCode 90 的时候它总是显示 NOP,而不是 xchg eax,eax.对我们来说这两个助记符"xchg eax,eax"和"nop"都显示 90h.

由此可见,一个事物(OpCode)不只对应一个名字(助记符).

我再重复一次:**在计算机世界里有的只是 OpCode,助记符只是这些 OpCode 的名字,这些助记符组成的系统就是汇编语言,助记符并不完美,因为没有完美的语言----任何语言的描述和现实总是有或多或少的区别的.**

在我们得出某些结论之前,我们先来做第二部分的练习.

Exersize3.

1. 输入助记符 add eax,1 ,你将看到

0040108E 83C0 01 add eax,1

2. 输入 OpCode 05 01000000 ,我们将惊奇的发现

0040108E 05 01000000 add eax,1

同样的助记符但是不一样的 OpCode!

事实上它们不仅大小不同,结构也不一样.第一个(83C0 01)3 字节长,包含 3 个域;第二个(05 01000000)5 字节长,但只有两个域.

在我们学习这些域之前,我们先来想一想:当你看着这两个 OpCOde:

83C0 01

05 01000000

你是否想到:

第一个 OpCode 中 01 和第二个 OpCode 中 01000000 是加到 EAX 的立即数?

既然助记符向 OpCode 可以有多种转换的方式,那么到底什么 OpCode 将会被插入 exe 文件中?这是由什么决定的呢?

答案是:这是由汇编器决定的.

当我们在数据段中写入一些值时,我们往往会使用:somevar db 0AH,0DH;而在代码段中我们不需要输入名字而只需定义值:db ACH.所以你可以用 16 进制写代码段,比如用

Mov esi,offset somedata Mov esi,offset comedata

Lodsb 代替 db ACH

好了,又到了我断开网络的时候了,我必须把我所写发送出去,否则我什么也不能提交.下次我们将从 OpCode 的结构开始,只是一个介绍而已,还涉及 OpCOde 的一些重要而一般的特性.你将发现这是多么的易学和实用.

二 OpCode 结构

机器指令回答处理器 3 个问题:
1. What to do ?
2. With what to do ?
3. How to do ?
其中第三个问题是可选的.
要回答这些问题,机器指令可能有多个逻辑块.在列举这些逻辑块之前我们先来做一些练习.

为了更好的试验,在开始这些练习之前我们先来定制一个特别的程序:
让程序变得小一点,这样 OllyDbg 可以更快的加载我们的程序;
用一个字节的助记符填充它,这样就可以把代码段当作白纸一样使用;
我们可以利用批处理文件或者设定快捷键等方式,更方便地加载我们的程序.

下面是这个程序的源代码:

```
        .code
start:
        rept 256
        nop
        endm
        call ExitProcess
end start
```

它将在程序中插入 256 个 nop 指令.

既然现在你已经知道在 call ExitProcess 之前有 256 个 90h ,我们是否可以用 16 进制来编码?

```
        .code
start:
        rept 256
        db     90h
        endm
        call ExitProcess
end start
```

或者我们检验一下看看

```
db     90h
nop
xchg eax,eax
```

这三个是不是一样的.我们可以这样改写程序:

```
        .code
start:
        rept 20
        nop
        endm
```

```
        rept 118
        xchg eax,eax
        endm
        rept 118
        db    90h
        endm
        call ExitProcess
end   start
```

当我们用 OllyDbg 加载编译好的可执行文件后,就可以看到在 call ExitProcess 之前全部是 90 nop.

OpCode 域简介

有 6 个域是 OpCode 可能会用到的,它们的名字是什么这并不重要,重要的是它们的排列顺序.

它们是:

1. Prefixes

2. Code

3. byte Modr/m

4. Byte SIB

5. Offset in command

6. imm. Operand

并不是这所有的 6 个域都会被用到,但是有一项却是一定会有的,那就是第 2 项 Code,有些指令甚至只会用到这一项.

在试验程序中输入

C3

2F

90

AD

所有输入的这些 OpCode 都只有 Code 块.

(提示:它们对应的助记符依次为:retn, das,nop,lod**.)

现在我们知道 lod**的 OpCode 是 ACh.我们来看看是不是可以增加额外的域来扩展它的功能呢.我们输入

F3 AC

可以看到

rep lod**.

我们可以确定 AC 是 Code 域,F3 AC 则是[Prefixes] [Code],所以 F3 是 Prifixes 域.F3 表示的是 Rep Prefixes,它也能与 mov**, sto**等指令连用.

现在请试着输入一些可以使用 rep prefixes 的助记符,并观察对应的 OpCode;然后再在 Ctrl+e 窗口中输入这些以 F3 开头的 OpCode,观察对应的助记符.

我们首先应该记住:**OpCode 由 6 个域组成,它们不必都用上,但是 Code 是一定会有的.**

我们再来看六个 BCD 码运算指令,输入

DAA
DAS
AAS
AAA
AAM
AAD

在 Intel 的文档中,这些都是只包含 1 个域的指令,这意味着它们都只有 Code 域.然而最后两条指令却是 2 字节的.我们可以通过观察这些指令,然后断定这些指令的格式和附加的域.

先不要看下面的答案,试着自己想一想 AAM 和 AAD 和别的指令有什么不同……

27 DAA
2F DAS
3F AAS
37 AAA
D4 0A AAM
D5 0A AAD

很明显我们可以看到:

1. AAM 和 AAD 都是 2 字节的,然而其余的 4 条指令都是 1 字节的.
2. AAM 和 AAD 的 OpCode 的第 2 个字节都是 0Ah

不知你是否想起这两条指令的描述:

AAM: divide AL by 10
商 放在 AH 里
余数 放在 AL 里
AAD: AL = AL * 10 + AL

两者的操作都与 10 有关,而且它们的第二个字节都是 10(0Ah).那么 0Ah 会不会是偶然的呢?会不会是操作数的一部分呢?那 AAM 和 AAD 的指令格式会不会不是

AAM: D4 0A
AAD: D5 0A

而是:

AAM: D4 imm8
AAD: D5 imm8

以及 imm8 可以是任何别的值呢?

我们可以验证一下,输入:

Mov al,8
D4 07 ;作用和 D40A 相同,只是除以 7 而不是 10

如果我们是正确的,那么按 F8 单步执行刚输入的指令后 AH(商) = 1, AL(余数) = 1.

现在,我们又知道了一种新的指令格式:

[CODE] [imm] (域 2 和 域 6)

同时我们也发现某些 OpCode 没有对应的助记符.在这里我得说一下,Oleh (OllyDbg 的作者)允许使用立即数作为 AAM 和 AAD 的操作数.

理解了 OpCode 的规则,将有助于底层程序员明白一些鲜为人知的事情----一些未在文档上列出的 OpCode,这也是掌握 OpCode 的另一个好处.

现在我们来看一些基本的规则:**虽然并不是 6 个域都是必要的,但是,它们的排列顺序绝对不能乱,必须严格按照上面的顺序进行.有些域或许不会出现,但只要出现了,编号小的域就绝对不允许出现在编号大的域的后面,反之亦然.**

例如,[Prefixes] [code] 的顺序绝不允许变为[code] [Prefixes].

输入

0110

和

1001

你将看到它们的不同.

(提示:0110 ----add dword ptr [eax], edx 1001----adc byte ptr [ecx], al)

下次我们将讨论处理器是如何确认 OpCode 的开始和结束的,我们也会看到 2 + 2 是怎样等于 3 的.

三 开始和结束

虽然这部分基于一些非常简单的事实,但并不是每一个程序员都能看到这些事实背后的本质.所以我们最好不要太过于轻视它.

用 OllyDbg 加载上次编写好的"nop"程序.看着左列的指令地址,起始的指令地址是多少,在我的程序里是:

00401000> 90 nop

再看看右边的寄存器窗口,EIP 的值又是什么,在我的程序里是

EIP: 00401000

EIP 寄存器的值是内存中某个字节的地址,这个地址将被认为是某条指令的开始地址.当内存中的字节未被处理器加载并解码,还不知道这条指令到何处结束.

现在我们按 F8 执行,可以看到

00401001 90 nop

EIP: 00401001

EIP 指向了内存中下一个字节并把它当作是指令.这里的下一个字节是指紧接上一条指令的结束地址的那个字节.如果你认为我没有写"EIP 指向下一条指令"是因为我糟糕的英语,那你就错了.尽管你并不是错在我糟糕的英语^_^(原作者是俄罗斯人).

如果一切正常的话,这些字节将被解码然后被当作下一条指令执行.事实上这些字节可能仅仅是一些垃圾,而处理器则不能正确的对其解码而产生异常.

在程序的当前代码行(高亮显示的那一行)输入 OpCode

FFFF

OllyDbg 不会把它当作是一条指令,你将在助记符那一栏看到 "???" .同时 EIP 指向我们刚刚输入的 FFFFh.所以如果不管这里的垃圾----对于处理器来说是"非法指令"----的话,处理器将解码并执行它.我们按下 F8 键,可以看到提示的错误信息(在 OllyDbg 下方的状态栏),处理器产生了异常.相应的,OllyDbg 中断并询问如何继续.

我们将 FFFFh 替换为 9090h,然后输入不同的助记符,比如长指令或者使用 lea 指令,单步执行并仔细观察 EIP 和指令长度的变化.你将看到处理器识别出指令的尺寸,且 EIP 现在指向紧接着刚输入的指令即 nop.

处理器是如何做到这个的呢?

事实上处理器是通过解码来正确识别指令的.通过分析指令格式可以知道组成指令的域以及域的大小,不同的位指示了某个大小的域被使用.

例如:短跳转指令 EB: imm8 为两个字节,EB 是 CODE 域,当 EIP 指向的内容是 EB 09,处理器通过解码第一个字节将得知 EB 的指令长度是 2 个字节.

所以问题的一个初步回答是:

1. 开始:处理器认为当前 EIP 指向的内存单元中的第一个字节就是指令的开始.
2. 结束:处理器通过对 OpCode 进行解码(大多数情况是根据[code]域),从而得知哪里是结束
3. 如果指令不是"控制类型"的,那么下一条指令的开始将紧接当前指令的末尾.

注意:除非被处理器加载,处理器是没有办法确定这些内容是否是真正的合法指令.

现在回顾一下第三点,什么是"控制类型"指令?一般来说它们是ret,call,jmp,jcc,int 等.它们之间的共同点是什么呢?不要回答我:"它们都跳转到某个地址".或者更加糟糕的答案:"它们跳转到另外某条指令开始的地方".

我们在程序的当前行输入 EB 00,并按 F8 执行.它将跳到哪里?我们看到处理器停在下一条指令开始的地方.我们再输入 EB FE,运行,但是并没有跳转到任何地方,就像所有的事情都停止响应一样.就算是普通的指令执行后,也会步向下一条指令,而现在却既不跳转也不步进到下一条指令.

真正的底层程序员应该理解指令的本质,而不单单从指令的字面意义上去理解它.真正的底层程序员不会说 cmp 指令比较操作数;他会说 cmp 指令是用第一个操作数减去第二个操作数,由此来设置相应的标志位.同时,我们关心的只是标志位,并不关心键操作后的结果,所以不需要把减操作的结果储存到第一个操作数中.这才是 cmp 指令的本质.

如果知道了它是如何工作的,你就可以用 cmp 指令来做任何它擅长的事情,而不仅仅是比较.其实并没有什么 cmp 指令,有的只是简单的逻辑的,数学和数据转换\写入\读取的 OpCode.

让我们回到"控制指令".它们真正做了什么样的操作呢?答案是:它们改变 EIP 寄存器.

我们知道 EIP 指向的内存数据被处理器当作是指令的开始.

通过这些控制指令我们又可以将什么值赋给 EIP 呢?

答案是:如果你非常了解 OpCode 和这些指令的算法,那么可以是任何 32 位的值.所以我们可以强迫处理器把内存中任意某个字节当作指令的开始.

输入 OpCode
04 AC
你将看到
00401008　04 AC　ADD AL,0AC

我们也知道 ACh 是 lod**的 OpCode,地址 00401008 是 OpCode 04 AC 的起始地址,但是被地址 00401009 指向的指令是什么呢?如果我们使 EIP 指向 00401009,那么处理器将不会把 ACh 当作是 04:imm8 的操作数,而是把它看做一个 OpCode　ACh->lod**?

如果现在我们需要实现一个算法

```
if   ZF = 0
       add    al,ACh
else
       lod**
```

请试着用助记符写下这个算法.

请看下面的汇编代码:

```
jne    $+1
add    al,ACh
```

这就是我们所需要的.请不要试着运行它,除非你清楚的知道[esi]指向的地方是什么内容.

如果 ZF = 1,EIP 将指向指令 add al,ACh 的第二个字节,因为我们知道第二个字节的值是 ACh,而操作码是 ACh ->lod**,我们实现这个算法仅用了 4 个字节!!!

```
00401005    75 01          JNZ          SHORT
00401007    04 AC          ADD          AL,0AC
```

我们一起来看看各个字节都表示什么意思.

75:imm 是 75 01 的域格式,75 是 JNZ 的 OpCode,imm 在这里是 01,会被加到 EIP 里面去,整个 7501 表示:如果这条指令被执行了,则 EIP 会指向下一条指令的第二个字节.04 AC 的域格式:04:imm,04->OpCode,AC->imm.两条指令都有两个域,格式为:[code] [imm].

如果 ZF = 0,7501 这条指令就会把 EIP 中下一条指令的起始地址 + 1,使得 EIP 指向下一条指令的第二个字节,处理器将认为 AC 所在的地址才是下一条指令的开始,这时 AC 将被认为是一个新的 OpCode;

否则 EIP 将指向 04 AC 所在的地址,开始的字节是 04h,处理器将认出格式域:04:imm8(add al,imm8),这时 AC 将被当成操作数,而不是操作码.

现在我们看看谁是最快最聪明的程序员.

1. 实现一个 4 字节的算法:

```
if      ZF = 1
        inc    eax
else
        mov    al,40h
```

2. 另一个测试(5 字节):

```
if      PF = 1
        set all bits in    ax to 0
else
        set all    bits in    eax to 0
```

3. 再一个测试(5 字节):

```
if      PF = 1
        set all bits in    eax to 1
else
        set all    bits in    ax to 1
```

(1 提示:

```
    je    $ + 1
    Mov    al,40h
    74h,01h,B0h,40h    )
```

(2 提示:

```
    jpe    $ + 3
    Xor    eax,eax
    74h,01h,66h,31h,C0h)
```

(3 自己动手啦~~)

四 Prefixes 66h

用 OllyDbg 加载试验用程序,输入

```
or          eax,-1
mov         ecx,edx
and         ebx,ebp
```

接着对上面每一个 OpCode 再按这种方式输入一次,如第一个 OpCode

第一个字节: 66

然后是 OpCode: 83F8FF

对剩余的两条指令用同样的方式输入.可以看到我们用这种方式产生的指令对 3 条指令都适用,除了 16 位寄存器指令,如

```
or          ax,-1
mov          cx,dx
and         bx,bp
```

66h 就是所谓的 Prefix.

Prefixes 是产生 OpCode 的第一个域.

回忆一下 OpCode 的 6 个域:

1. prefixes
2. Code
3. byte modr/m
4. byte sib
5. offset in command
6. imm. Operand

记住:

在实际的应用中,并不是所有的这 6 个域都会被用到,但是有一项是一定会有的,即第二项 Code,这 6 个域的顺序绝对不能乱,必须严格按照上面的顺序进行.

Prefixes 是所有域中最容易理解的一个,请先明了它的一些特性:

1. 它是唯一的一个可能出现在 Code 之前的域

2. 所有的 Prefixes 都只有一个字节

3. 在一个 OpCode 中可能会有多个 Prefixes

Prefix 66 的意思是“切换默认的操作数的大小”.例如在有的系统中有两种默认的操作数大小:16 位和 32 位.在 Win32 编程环境中默认的操作数的大小是 32 位,但操作数有可能会被写成 16 位或者 32 位,唯一的区分方法是看它有没有 Prefix 66.

我们来看一些例子,输入下面这些单字节指令:

```
LOD**
LODSW
LODSD
```

啊!LODSW 和 LODSD 使用同样的 code 域 AD!

其实,LODSW 和 LODSD 这两条指令是同一个指令,只不过它们的操作数的大小不一样而已—LODSW 使用 WORD(不是 Win32 默认操作数大小)作为操作数,而 LODSD 则使用 DWORD(是 Win32 默认操作数大小)作为操作数.

所以对于 LODSW 指令需要用 prefix 66 切换操作数大小.请注意我们并没有说“指定”而是说“切换”,反映到这个例子中,就是“切换默认的 32 位操作数到 16 位”,而不是“指定操作数的大小为 16 位“.如果默认的操作数大小是 WORD,那么切换后就是 DWORD；反之,如果默认的操作数大小是 DWORD,那么切换后就是 WORD.那些没有使用 WORD 或 DWORD 的指令(BYTES,QWORDS 等),不会对操作数的大小做任何改变.

输入:

```
mov         al,0ff
mov         al,cl
```

接着在这两条指令之前加上 66h,可以看到

```
66:B0 FF            MOV         AL,0FF
66:8A C1            MOV         AL,CL
```

什么都没有被改变.

记住:Prefix 66 仅对操作数为 WORD 和 DWORD 的指令起作用.

我们在使用 BYTE 操作数的指令前加入 prefix 66 ,是不是产生了一条非法指令呢?嗯,事实并不是这样子的.

运行指令

```
66:B0 FF            MOV         AL,0FF
66:8A C1            MOV         AL,CL
```

没有任何问题,它们和

```
MOV         AL,0FF
MOV         AL,CL
```

的功能是一样的.

如果 Prefixes 不能对随它之后的 OpCode 起作用,那么处理器将忽略它.

另外的一个 Prefix rep 的作用是让处理器对随后的指令循环执行 ecx(cx)次,指令 inc eax 的 OpCode 是 40,我们来看看如何使用 prefix F3(rep)使 inc eax 连续执行 3 次.

在我们的试验程序里输入

```
xor         eax,eax
mov         ecx,3
rep         inc eax
```

然后运行.你将看到两点:

1. 最后 eax = 1,这意味着 prefix F3 并没有起作用----它被忽略了；
2. 没有任何异常(execption)产生.

在 OllyDbg 里面我们可能会遇到以下两个问题:

1. 如果你输入 rep inc eax 它将提示“无法识别的指令”,我们试下看,输入 F3 40 行不行.奥!OllyDbg 还是没有正确的识别!~~

如果按 F8 单步执行我们刚刚输入的指令,这样运行的结果是:

1) Eax = 1

2) 没有任何异常发生,处理器忽略了 prefix F3.

如果 Prefixes 不能对随它之后的 OpCode 起作用,那么处理器将忽略它.

输入包含两个 prefix 的指令

rep lodsw

66: F3: AD (REP LODS [WORD DS:ESI])

在这条指令中我们看到两个 prefix,66 和 F3.

所以 Prefixes 域的另一个重要的特性是:一条指令可能只有一个 CODE 域,一个 mod r/m 域,或者一个 offset 域等,但是可以有多个 Prefixes.

有关 prefix 66h 的最后的“新闻”是:

你可能错误的认为在实模式下默认的操作数大小是 WORD,而在保护模式下是 DWORD,事实并不完全这样.我们唯一可以确定的是----当在 Win32 环境下默认操作数 DWORD.因此,你**在 Win32 下使用的任何操作字(WORD)的指令都会变长一个字节(prefix 66),并且需要多花一个时钟周期去执行.**这个并不是一点好处也没有,当你习惯于计算数学问题,你可以通过取舍指令大小和运行速度来确定是不是用 WORD 更好一些.任何使用字的指令将多花你一个字节和一个用来解码的时钟周期.

那么默认操作数大小是如何确定的呢----这是被段描述符的 D 位确定的.在实模式下它被赋值位 0,所以在实模式下默认的操作数往往是 WORD.在保护模式下可能是 0 或者 1(在 Win32 应用程序中是 1).

所以:

```
If (PROTECTED MODE    &&    BIT D = = 1)
      AD = LODSD
      66 AD = LODSW
Else
      AD = LODSW
      66 AD = LODSD
```

这就是我们之前所说的:不同的 OpCode 可以有同样的助记符,一个 OpCode 可能有多个不同的助记符.

我们也看到某些 OpCode 可能有两种不同的功能,而这依赖一些条件.如果你认为指令可以和 prefix 66 配合使用,你将明白这里的条件和功能分别指的什么.

下次我们继续学习 OpCode 的第一个域 Prefixes.同时我推荐读者查找一下 x86 指令集查询手册,上面有所有的 OpCode 对应的助记符和指令构成

五 Prefixes 67h,F2h,F3h

在上一篇教程里我们学习了有关 Prefixes 的一般特性:

1. 所有的 Prefixes 都只有 1 个字节.

2. 在一个 OpCode 中可能会有多个 Prefixes.

3. 如果 Prefixes 不能对随它之后的 OpCode 起作用,那么它会被处理器忽略.

我们举例学习了 Prefixes 66h 的作用和用法----切换默认操作数大小.

现在我们学习另外的 Prefixes,看看它们是否有什么好玩的地方可以影响我们的应用程序的执行.

Prefixes 可以被划分为 5 个集合:

1. 切换默认操作数大小(change DEFAULT operand size)(66h)

2. 切换默认地址大小(change DEFAULT address size / segment override prefix)(67h)

3. 重复(Rep) (F3,F2)

4. 切换默认段(change DEFAULT segment)(2e,36,3e,26,64,65)

5. 总线锁定(Bus lock) (F0)

Prefixes 66H

我们已经在之前的教程里面学习过.

一个测试题:

下面哪一个指令可以在 32 位应用程序中拥有 66H Prefix:

sca**

scasw

scasd

你可以在调试器里输入这些操作码来检验你的答案.

(提示:答案是 scasw)

Prefixes 67H ---- 改变默认地址大小.

输入助记符

mov al,[eax]

可以看到

8A00 MOV AL,[BYTE DS:EAX]

现在再输入以 67H 开头的 OpCode.

67:8A00 MOV AL,[BYTE DS:BX+SI]

1.可以看到相应的字节已经被 16 位的寄存器 bx, si 寻址.在 32 位寻址模式中所有的数据都被 32 位的地址确定,包括 32 位基址,32 位索引,32 位偏移.在 16 位模式中这些都是 16 位的.

2.可能你还注意到地址部分并不是变为 mov al,[ax],而是变成了[bx][si].为什么呢?我们将在学习[modr/m]和[sib]的时候给出答案.

现在我们可以暂时认为,在 16 位地址模式中无法像 32 位地址模式一样使用所有的基址寄存器和索引寄存器,OpCode 用来指定寻址的寄存器的位域也是不同的.

不知道你是否需要常在 32 位环境下使用 16 位的寻址模式,不用担心,我们会深入的学习这个.在 32 位环境下使用 16 位的寻址模式可能会非常高效,不过这个需要你确定控制的总的地址范围不能超过 0FFFF(一个内存页面的大小 4kb).

Rep. Prefixes F2, F3

如果你了解串操作指令(如 movs, scas, lods 等),你一定知道什么时候使用

rep\repe\repne 前缀.

一些串操作指令只能使用前缀 rep,如 mov**, lod**----在计数器 e(cx)=0 的时候终止;

另一些则可以使用 repe 或 repne----在计数器变为 0 并且 ZF 标志位不满足前缀指定的条件的时候结束.换句话说 repne 在 ZF=1 而 repe 在 ZF=0 的时候终止.当然它们都会在满足 ecx=0 而当 ZF=0 或 1 的时候各有不同.

有两点规则:

1. 你可以看到有 3 种 Rep prefixes 助记符:rep,repe,repne, 但是只有 2 个 OpCode:F2,F3

2. 如果某些指令只使用前缀 rep,那么这里的 rep 可以用 repe 或者 repne 来代替.这种情况下

```
Rep         Lod**
Repe        Lod**
Repne       Lod**
```

按照同样的方式运行:它们会重复运行指令 LOD**共 ecx(cx)次,而不管 prefix 是 F2 还是 F3.

我们可以验证一下:

```
xor         eax,eax       ;使 ZF = 0
mov         esi,esp       ;esi 指向栈顶
mov         ecx,10
repe        lod**         ;opcode with repe and rep indentical - F3

mov         esi,esp
mov         ecx,10
repne       lod**
```

按 F7 或 F8 运行后可以看到它们的作用是一样的.

只有在可能会改变某些标志位的重复串指令时,F2 和 F3 才会表现出不同,如 sca**.在这种情况下,有些指令与重复前缀操作搭配使用,F2 和 F3 会把最后一位与标志位 ZF 进行比较,如果它们不相同,则重复串指令的操作将会结束.而有些指令不用进行这个比较的操作,因此标志位 ZF 对这些指令的运行结果无影响.只需将 F2(1111 0010)和 F3(1111 0011)用二进制表示你就会明白我的意思.

(提示:F2(1111 0010)的最后一位是 0,指令执行的时候 CPU 会将 F2 的最后一位 0 与 ZF 标志位比较,如果此时 e(cx)为 0,并且 ZF 为 1 与 F2 最后一位不同,则指令不再重复; 否则重复.F3 同理)

下次我们将完成对 prefixes 的讨论并且我们将进一步学习改变 EIP 的 OpCode.

六 Prefixes Segment override and LOCK

这次我们将结束对 prefixes 的学习.

剩下的两个 Prefixes 是"Segment override"和 "LOCK" prefixes.术语"Segment override"可能会让某些初学者感到困惑,但它却像是为 Intel 指令系统定制的.

打开我们之前的试验用程序.输入:

```
mov         eax,[ebx]
```

可以看到

```
8B03        MOV         EAX,[DWORD DS:EBX]
```

输入

```
mov         eax,GS:[ebx]
```

可以看到

```
65:8B03     MOV         EAX,[DWORD GS:EBX]
```

65 就是一个"segment override prefix",用来改变默认的段为 GS.事实上,在使用内存中的数据时,处理器必须首先知道它的段地址和偏移量,但是如果在每个地方都要显式的直接指出段地址,那么在 OpCode 格式中就必须额外增加一个域,这将会比现有的 OpCode 体系多占用大量的字节,而且要处理器必须多花费额外的时钟周期来进行解码----无论在空间上还是时间上,都不值得!

因此,为了解决这个问题,一个方案诞生了.指令按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: EIP 寄存器

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.

有了这个规则,处理器识别当前应该用哪个段将会变得非常简单而直接:如果有“Segment override prefix”,那么就使用这个 prefix 所指定的段;否则就使用默认的段.

输入

```
AC
3E AC
```

可以看到

```
AC              LODS [BYTE DS:ESI]
3E:AC           LODS [BYTE DS:ESI]
```

3E 是表示段 DS,但是实际上在这里即使不直接指明 3E,处理器也是会使用 DS 的,因为 DS 是指令 LODS 的默认段.

对于程序员来说,使用非默认的段将多占用 1 个字节,并且多花费 1 个时钟周期去解码.对于其他的那些改变默认内容的指令也如此(如 66,67).

编写 Win32 用户态汇编程序往往不需要去改变段寄存器,但底层程序员可能会用到.现在我们总结一下 Win32 用户态模式段寄存器的内容:

CS:对于所有的用户态程序这个是一样的,在 NT 系列操作系统中是 1Bh,而在 9x 中则为 227h.如果你记得关于绝对地址跳转的宏,我可以给出一个简单的绝对地址跳转指令格式,但这个在 9x 中不同.

我们一起看看远跳转的 OpCode.EA -- byte "code",这个字节告诉处理器这是一个直接远跳转的 OpCode,当处理器遇到 EA,他将得知后面会跟着一个 48 位地址,低

32 位是偏移量而高 16 位确定段寄存器.

既然现在我们知道对于 NT 代码段寄存器的值是 1Bh,所以为了跳转到地址 12345678h,我们可以这样编写代码:

```
db      EA                  ;long jump
dd      12345678h           ;offset where to jump
dw      1Bh                 ;segment selector for code in NT
```

对于 9x 段来说只是代码段选择子的值不同而已

```
db      EA                  ;long jump
dd      12345678h           ;offset where to jump
dw      227h                ;segment selector for code in 9x
```

如果限制应用程序运行在特定的操作系统(NT 或者 9x),这样做很有好处:

```
;NT=1
absjmp macro addr
    ifdef     NT
        db      0EAh
        dd      addr
        dw      01Bh
    else
        db      0EAh
        dd      addr
        dw      227h
    endif
endm
```

用法:

```
absjump     401000h
```

注释行 NT = 1 依赖与你想使用这个绝对地址跳转的系统.

可以试着在 OllyDbg 中写入一些跳转指令,比如你想跳转到一行,比如

```
00401013    |. 90           NOP
```

你可以输入

```
EA 13 10 40 00
```

剩下的两个字节在 NT 中是 1B 00 ,而在 9x 中是 27 02.

重要的一点是 DS,SS,CS 段都只是用户态段的别名,也就是说它们的数值是一致的.(Important thing to remember is that DS SS CS segment are alias segments in Win32 user mode app.)**这意味着我们可以通过任一段寄存器寻址数据.**

如果你正试着做"String 到 Dwords 的转换",你可以使用栈操作,当栈顶指针直接指向.data 节,使用 push 可以直接将常量字符串放置在数据节.

同样,使用.code 节的数据可以这样子:

```
    .code
msg db 'Some text',0
start:
invoke MessageBox,0,offset sometext,.....
```

另外,下面的代码完成的功能是一致的:

```
        .data
somedw dd 12343
......
        .code
......
mov eax,somedw
mov eax,dword ptr offset somedw
mov eax,dword ptr DS:offset somedw
mov eax,dword ptr CS:offset somedw
mov eax,dword ptr SS:offset somedw
......
```

我们还须记住的是某些节上的数据可能受到保护,(可以在链接的时候使用/section:[sectionname] option 选项改变).然而如果你可以用"ptr somedata"的方式寻址数据,那么你也一定可以任意使用 DS,CS,SS 作为段选择子.我们将在系统编程方面详细讨论这个.

那么剩下的寄存器呢?

我们先看 ES,我们在串操作指令经常要关心它.

在 DOS 时代我们经常要比较 ES 是否等于 DS,来确认我们的源和目的是不是在同一个段.

在 Win32 用户态编程中我们不需要这样,但这并不是说 ES 不再被串操作指令使用.

我们开始试验之前先做点笔记:ES 在包含源和目的两个内存操作数的串操作指令中仍然被使用,例如:mov** = move dword ptr DS:ESI to ES:EDI.

那么如果对于上例包含两个内存操作数的指令运用 Segment override,那么是哪个段选择子被改变呢?

输入:

```
A5 65 A5
A5              movs    [dword es:edi], [dword ds:esi]
65:A5           movs    [dword es:edi], [dword gs:esi]
```

我们看到 Segment override 改变了源.**改变目的选择子是不被允许的!如果你想改变目的,你可以改变 ES 的值,而不是改变作为选择子的寄存器.**所有的包含两个内存操作数的串操作指令都是如此.

顺便说一下,如果你开始试着去查看 OpCode,那么你肯定会看到所有的串操作指令都是单字节的,假设你不使用 66 prefix 去改变操作数大小,那么在速度允许的情况下,可以写出非常简洁的代码.

我们先前说到,ES 仍然在被使用,但是我们不需要特意的用 DS 去初始化它,当系统加载程序的时候就已经为我们做好这个了.然而你可能会轻易的破坏它的值然后糟糕的后果立马而至.

我们看看选择子是如何影响串操作指令的.栈从高址往低址生长,这意味这我们可以使用比当前栈顶指针小的地址,而不会破坏栈内的数据(返回地址,参数,局部变量等).例如,在 OllyDbg 里当前栈顶的内容为:

0012FFC4 77F1B9EA RETURN to KERNEL32.77F1B9EA
0012FFC8 0012E2D4
0012FFCC 77F92CD4 ntdll.77F92CD4
0012FFD0 7FFDF000

所以我们可以使用 0012FFC0 以下的地址空间,需要注意的一点是默认的栈大小是 1MB,当然可以在链接的时候使用相应的选项去改变这个值;并且不要使用开头的 1kb,那是用来捕捉错误的.

你可以使用一些 push 和 pop 指令观察栈的变化情况.

现在我们给局部变量分配一些空间来检测一下 movs 对于不同的选择子是怎么样工作的.我们要做的是:

1.用短的字符串填充栈空间

2.以字节为单位将字符串拷贝到接近栈的区域,并且每一步我们指定不同的段寄存器

- 验证一下 DS,SS,CS 都只是用户太短的别名
- 最后一步改变 ES 看程序是否受影响

输入:

lea edi,[esp-4](该指令可以理解为取 esp-4 处双字的地址,即 esp-4,之所以这样写是因为指令 mov esi,esp-4 是非法的)

现在 edi 指向局部变量所在的空间,所以我们不用担心破坏栈中的返回值和参数等等.我们将使用小于等于 esp-4 的地址空间.

输入助记符

std

我们设置方向位为 1,串操作指令将使用递减的地址.顺便说一下,当你在窗口回调函数中置方向位,请务必记得在返回前清方向位,因为回调函数是通过置 DF=0 为系统返回值的.(When you change DF to 1 in your window callback procedure - remember to set it to 0 before your proc returns - your proc returns to system code and the code assumes that DF=0)

输入另一个助记符

```
mov al,5
sto**
dec al
```

输入 OpCode

75 FB

这些指令类似于下面的汇编源码:

```
lea edi,[esp-4]
std
mov al,5
@@:
sto**
dec al
```

jne @B

这里面有一个问题,短转指令的二进制格式是

0111tttn:imm8

前四位0111确定了这是一个短跳转指令,在调试器里它表现为第一个十六进制位是7,接着的四位是用来确定条件的”tttn”位域,这和许多指令中用来测试标志位的格式是一样的.第二个字节是在跳转指令被解码后将被加到EIP的有符号数.

我们试着解码OpCode 75 FB:

7 -- 短条件跳转的标志

5 -- tttn 0101,0100 代表‘e’,zf=0;0101 代表‘ne’,zf=1;可以看到改变最后一位就相当于把条件改变位‘非’,这就是为什么我们叫它“tttn“,注意这里的“n”.

FB -- 即-5,我们需要EIP回跳5个字节

1字节 AA STOS [BYTE ES:EDI]

2字节 FEC8 DEC AL

2字节 75 FB JNZ SHORT

5字节

好,我们回到"register override".

切换到数据窗口,在OllyDbg的Command文本框里输入命令 d esp-4 ,并往上面滚动一行,这样好看到当前栈顶的内容,因为我们将用字符串来填充esp-4往下(地址变小)的空间.我们按F8单步执行,可以清楚的看到字节 01 02 03 04 05 是如何存放的.

我们首先来验证一下DS,CS,SS是否是段选择子的别名.(First we check if DS CS SS are alias semgent selectors)

输入

lea esi,[edi+5]

这条指令执行后esi将指向之前我们构造的字符串(01 02 03 04 05).并且edi已经指向该字符串的末尾,这样我们就可以开始拷贝了.

mov**的OpCode是 A4

A4 = COPY FROM [BYTE DS:ESI] TO [BYTE ES:EDI]

在A4前使用segment override prefix 我们就能改变源选择子,这样的prefix有CS-2E,SS-36,ES-26.

我们先看看默认的情况:

输入

A4

然后使用F8单步执行,并观察数据窗口内容的变化.

现在我们看看CS的情形:

输入

2E A4

然后使用F8单步执行,并观察数据窗口内容的变化.

同样测试一下 36 和 26.

如果操作正确,你将可以看到字符串的头四个字节被成功拷贝,这表明 SS DS CS ES 指向同样的别名段.(That shows that all SS DS CS ES pointed to the same alias segment through different selectors.).

你可以试一下别的 segment override prefix,你得到的肯定是一个错误提示.

我们用刚才字符串剩下的一个字节来测试一下 ES.

输入

66 6A 00 即 push word 0

输入

pop es

A4

同样的,按 F8 键,当到达 lod**指令行时,OD 的状态栏会提示异常.

这表明 ES 仍然在双内存操作数串操作指令中被使用.我们可以输入

push ds

pop es

来解决刚才的问题.

执行这两条指令,然后输入 A4,然后一切正常.

值得提到的另一个段寄存器是 FS,FS 被用作异常处理.然而现在你记住这句话就好了,每当你使用 FS 操作数据的时候,将多消耗你一个字节以及一个时钟.

当然使用这个寄存器要看你的意图是什么.

下一个 Prefix 是 LOCK.

如果在 Pentium 和 Pentium MMX 下使用如 F00FC7C8 这样的指令,可以冻结处理器,在这里我仅仅引用其中的一些描述.如果你想了解这个可以去看 Dr.Dobbs 的文章.

对于 LOCK prefix 我没有什么要说的,在 Intel 的手册里有很好的解释.在结束这部分之前我想提一些关于 prefixes 的特殊的用途,但是在 P III 下处理器会忽略掉这些,Intel 提出在新的模型中那些特殊的用法可能有新的特殊意义,不正当的用法可能导致一些不可预料的结果.关于新一代 Prefixes 应该提到 3E “hint” prefix.

只有在多处理器系统才需要 LOCK.当对内存操作数执行如下操作过程:

1.读取内存

2.在算术逻辑单元操作读取到的内存变量

3.写回内存

这时才需要.

可以看到在第 1 和第 3 阶段有一段间隔,当目前的处理器正在处理读取到的内存数据的时候,其他处理器也可以读取同一个内存数据,最后可能导致其中的一个被覆盖.而 LOCK 信号可以阻止任何其他处理器访问共享内存,直到指令将数据写回内存.大多在同步系统中被使用.

七 Bit filed

在我们继续学习 OpCode 块中的域之前,记住一个非常重要且常被使用的域----

"reg field"----是非常必要的(请将这里的位域与之前六个的域分辨清楚,这里的位域是被包含在之前的域之中的,如下段的第一句).

reg field 在域 modr/m,sib,以及某些单字节 OpCode 内部被使用.Reg bit field 有 3 位,所以可以包含 2^3 = 8 个值,对应 8 个通用寄存器.

000 = EAX
001 = ECX
010 = EDX
011 = EBX
100 = ESP
101 = EBP
110 = ESI
111 = EDI

Reg field 也可以对应一种不同集合的寄存器--"局部"寄存器(partial registers),我们将在后面碰到的时候再详细讨论这个.现在我们只

需要记住:它的前四个值对应前四个全寄存器(full registers)的低局部寄存器,

000 = AL
001 = CL
010 = DL
011 = BL

后四个值则对应前四个全寄存器的高局部寄存器,

100 = AH
101 = CH
110 = DH
111 = BH

我们先看看只有全寄存器被使用的情况(在带 reg field 的单字节 OpCode 中).在这种指令中,高 5 位是 CODE bitfield,低 3 位是 REG field.

我写了一个简单的用来解码带 reg field 单字节指令的程序(在附件中,bitfield.zip),希望你们可以充分使用它,并记住这所有的 bit fields.

在之前的教程里,我们用 OllyDbg 就足以实践和讨论 OpCode.

现在我们来看关于指令的至关重要的一部分----地址.但这个问题关系到 bit field.

两个最重要的域 modr/m,sib 都有 bit field,你可以用大脑联想一下如何用一个或多个十六进制数字表示 bit field.

bit field 的格式是: 2 : 3 : 3.这意味着高 2 位代表一个东西,低 3 位代表一个东西,中间的 3 位代表另一个东西.但我们将它用 16 进制表示,它又是 2 个十六进制数.因为每个十六进制位由四位组成,所以对于位格式 2 : 3 : 3,高十六进制位由第一个域的两位和第二个域的前两位组成,剩余的位则组成低十六进制位.

例如,在 modr/m 中,如果要表示两个操作数都是寄存器,且寄存器分别是 edx 和 edi,那么 modr/m 的值是 F9.如

mov edi,ecx 的 OpCode 是 8B F9,
sub edi,ecx 是 2B F9,

可以看到两个 OpCode 的 modr/m 相同都是 F9.F9 的二进制位格式是 11 111 001,所

以

11 -- mod(11 表示操作数都是寄存器)

111 -- reg(111 代表 edi)

001 -- 寄存器或者内存数(001 代表 ecx)

关于 modr/m 的另外一点是,上述指令也可以这样编码 11 001 111,也就是 1100 1111 即 CF(可以看到只是 111 和 001 的位置互换),希望这没有使你感到困惑.

如果想通过 modr/m 和 sib 创建任意地址,我们只要知道

1. reg field 8 个可能的值
2. modr/m 4 个可能的值
3. sib 4 个可能的值

我们还需要一些简单通用的规律,和少数的几个 Exeption.

当我们学完这个,在输入指令的时候你就能够很容易的确定任意 OpCode 的大小或尺寸.OpCode 中的地址部分是非常耗空间的.我们经常使用操作数,这意味着我们总是指定操作数或者说是它们的地址(包括寄存器).而 Exeption 是操作数预先定义的助记符,例如串操作指令的操作码都非常简短,这是因为它们没有地址部分.

假如我们不了解 OpCode 的地址部分的话,我们无法断定指令长短或者尺寸.

举个例子,下面两条指令实现同样的功能:

```
1. mov     edx,[ecx*4]
2. xor     eax,eax
   mov     edx,[ecx*4][eax]
```

从表面看似乎第二个比第一个多一行,它的第二条指令甚至要多一个域,还多一些字符等等.好像似乎一定是第二种情况要长一些 0_0.然而,这只是我们的主观臆断而已!

第二个版本比第一个要少两个字节:

```
8B 14 8D 00 00 00 00    MOV     EDX,[DWORD DS:ECX*4]

33 C0                   XOR     EAX,EAX
8B 14 88                MOV     EDX,[DWORD DS:EAX+ECX*4]
```

通过一些不算艰苦的工作,我们也可以在很短的时间内学会这个.

关于 bit field 的编码:虽然使用十六进制对位格式 5 : 3 或者 2 : 3 : 3 进行编码不是那么简单,但是在汇编源码中使用二进制是非常容易的.例如指令"inc reg"的 Code 域是 01000,剩下的 3 位是寄存器.因为 eax = 000,所以 inc eax 就是 01000 000 即 40h.

这里唯一的问题是,在位之间加入空格是不会被汇编器接受的,所以我们应该写作 01000000b.但这种方式导致我们不易区分不同的 bit fields.我们可以编写一个简单的宏(macro),用它来帮助我们轻易的分开这些 bit fields.我们给这个宏取名 bcr(a Byte OpCode with Code and
Reg fields)

```
Bcr     macro   _code,_reg
        db  _code & _reg & b
endm
```

现在我们可以用下面的写法来代替 db 01000000b

```
Bcr   01000,000      ;inc      eax
Bcr   01000,001      ;inc      ecx 等等

Bcr   01010,000      ;xchg     eax,eax  或者  nop
Bcr   01010,001      ;xchg     eax,ecx
.....
Bcr   01010,111      ;xchg     eax,edi 等等.
```

同样的,我们可以编写一个宏帮助我们编码 modr/m 以及 sib,它们的格式是 2 : 3 : 3.

```
Modrm      macro      _mod, _regcode, _rm
      Db     _mod & _regcode & _rm & b
Endm
```

由于 sib 有同样的格式所以我们可以定义

```
bsib       EQU       modrm.
```

现在我们可以很容易的看出各个 field:

```
Modrm      11,000,101
```

比如

```
mov        reg1,reg2
```

的 CODE 是 8Bh,跟在它后面的是 modr/m 字节,其中 mod field 是 11.

例如:

```
Db         8bh
Modrm      11,000,001          ;mov      eax,ecx(eax=000, ecx=001)
Db         8bh
Modrm      11,111,001          ;mov      edi, ecx(edi=111)
Db         8bh
Modrm      11,001,111          ;mov      ecx,edi
```

这里我举的例子只是说明使用二进制编码 bit fields 是没有问题的.

现在我们切实的学习这些 fields 的作用.

我们从使用 REG field 的单字节指令开始,格式是 5:3,高 5 位是 CODE field,低 3 位是 register field.

这里有几个问题:

1.为什么我们从 REG field 开始呢?

REG field 在寻址中使用最多,学习它我们可以更深入的学习指令的地址部分.

Reg field 在 modr/m 和 sib 中被用到.Modr/m 和 sib 在绝大多数 OpCode 中是用来确定指令的地址部分的,它们总共有 6 个域,其中的 4 个用来存放寄存器的值.

2.为什么从 CODE:REG 5 : 3 格式的单字节 OpCode 开始?

因为这是使用了 bit field 的最简单的格式,如果习惯了使用二进制和十六进制,我们可以更容易处理 2 : 3 : 3 的格式.另外,在 5 : 3 格式中,只有一个操作数,这使得编码和解码更加容易.

现在我们运行 regfield.exe(在附件中).它一共有三个 Tab 页.

第一个“Reference”.你可以通过它生成单字节 OpCode,通过二进制和十六进制显示.还有一些用来选择指令和操作数的按钮.

这里有一些小技巧:依次单击第一列按钮,你可以看到他们产生的 CODE 域:

INC - 01000
DEC - 01001
PUSH - 01010
POP - 01011

XCHG - 10010

除了 XCHG - 10010,其他的是从 01000 至 01011 按 1 递增的.

而寄存器操作数也是同样的.

EAX - 000
ECX - 001
EDX - 010
EBX - 011
ESP - 100
EBP - 101
ESI - 110
EDI - 111

第一个 Tab 页实际上是一个最简易的汇编器,虽然它只能翻译 5 种类型的指令,且只能使用全寄存器.而剩下的两个 Tab 也是类似的,"tell mnemonic"页是一个饭汇编器,而"tell opcode"则是汇编器.

在 tell mnemonic 页中可以看到一些类似“Reference”页中的按钮.你可以按动这些按钮,如果你所按按钮代表的 OpCode 和页面上以二进制或十六进制显示的是一致的,那么你可以继续下一个,否则重试.

一些小提示:如果第一个十六进制数是

4 - inc/dec , reg
5 - push/pop, reg
9 - xchg eax, reg

如果同一个数字对应两个 OpCode,那么就要看第二位十六进制数大于等于 8.

最后一个 Tab 页则是要我们解码或者说是汇编助记符(mnemonics).依照界面的提示:

第二行就是需要被汇编的汇编指令,如 XCHG EAX,EDX;

你可以按下第三行的相应按钮使其正好是 XCHG EAX,EDX 的 OpCode 10010 010;

或者是在第四行的黑色文本框中输入十六进制数 92;

然后单击“TEST”按钮,如果你的输入是正确的就可以继续下一个,否则重试.(但是第三行的按钮似乎不怎么管用)

多多练习直到你可以在 5 分钟之类得到 100 个正确的回答并且没有任何错误提示^_^.

inc dec push pop xchg eax,reg 这些都是单字节指令.但并不是只有这些指令才是单字节指令.例如

00001111:11001 reg(bswap reg)

第一个字节只是告诉处理器接下来的 OpCode 属于“新”的指令集.在这条指令中有用的部分也只用一个字节,并且属于 5 : 3 的格式,那么最后的三位就是 Reg

field 了.如

bswap eax = 0FC8 (11001 000)
bswap ecx = 0FC9 (11001 001)
....
bswap edi = 0FCF (11001 111)

你可以试着汇编/反汇编一些包含只使用一个寄存器操作数的 modr/m 的指令.如 mul reg,iul reg 等.

在你开始之前我们先说说 16 位的寄存器.你一定还记得 prefix 66h 吧.在 32 位模式中,我们只需在 OpCode 前加上 66 就可以了.如:

INC EAX = 40h
INC AX = 66h 40h
DEC ECX = 49h
DEC CX = 66h 49h
...
BSWAP EDI = 0F CF
BSWAP DI = 66 0F CF

我们一起看看局部寄存器.

以 MUL 和 IMUL 指令为例,他们是:

MUL 1111011w:11 100 reg
IMUL 1111011w:11 101 reg

我们首先看到的是,两条指令首字节相同,并且最低位都是同样的标记"w"----F7(w=1),F6(w=0).

讨论局部寄存器,这是不得不讲的一位(bit).

若 w = 1,reg field 为全寄存器,否则 reg field 为局部寄存器.

所以:

w = 1		w = 0
reg	value	reg
EAX	000	AL
ECX	001	CL
EDX	010	DL
EBX	011	BL
ESP	100	AH
EBP	101	CH
ESI	110	DH
EDI	111	BH

一共有 8 个通用寄存器,其中的 4 个可以作为两个局部寄存器使用.

你可能会问,reg field 有 3bit,加上 w bit 一共 4 位,2^4=16,刚好足够表示这 16 个寄存器,既然这样,那么可以给每一个寄存器(包括全寄存器和局部寄存器)分配一个数字代号,为什么不采用这种方式呢?

这个问题问得非常好!

这是因为"w"位只影响寄存器而不影响指针(pointer,即地址),而指针只能是全寄存器. 像 reg,[reg][reg*4][displacement]这样的地址,所有寄存器可以使用的域都被使用了,但是这里只有一个操作数(即 reg)可以是局部寄存器.Code 域指示了寄

存器是指针与否.

另外的一个原因是,在现在的编码系统中,这 4 个域一共是 4*3=12 位,被两个字节的[modrm][sib]包含,也包括其中的 scale 和 mod.如果给每一个 reg 域 4 bit,那么仅仅 reg 域就占用了 16bit,在加上 scale 和 mod,那么每条指令都会相应的增大.

现在你可能明白了汇编器是如何编码/汇编的了.

看看下面的语句:

byte ptr ...
word ptr ...
dword ptr ...

大多数情况下(操作数默认为 32 位):

byte ptr - 汇编器置 w 位为 0
word ptr - 置 w 位为 1 且增加 prefix 66h
dword ptr - 置 w 位为 1

对于 16 位的操作数,一般都需要加 prefix 66h

我们回到 MUL/IMUL reg 指令:

MUL 1111011w:11 100 reg
IMUL 1111011w:11 101 reg

你一定发现它们仅仅是第二个字节中间的三位不同(以列分隔).第二个字节就是所谓的 ModR/M,它的格式是 2 : 3 : 3.

bits 7,6 :mod
bits 5,4,3 :code or reg
bits 2,1,0 :mem or reg

在 OllyDbg 中输入以下几条指令:

mov reg,reg
sub reg,reg
add reg,reg
and reg,reg

它们都是双字节 OpCode,且第二个字节都是 ModR/M.

在输入一些使用同样寄存器操作数的指令如:

mov ecx,edi
sub ecx,edi
add ecx,edi
and ecx,edi

你可能会看到他们的第二个字节是一致的.之所以说是可能,是因为这种指令可以有两种编码的方式,呆会你就会明白.

(在 OD 中结果如下:

8BCF mov ecx, edi
2BCF sub ecx, edi
03CF add ecx, edi
21F9 and ecx, edi

8BF9 mov edi, ecx
2BF9 sub edi, ecx

03F9 add edi, ecx
21CF and edi, ecx).

将其中的第二个字节以二进制表示,并且以 2 : 3 : 3 的格式分隔.通过最后的两个 3 bit 我们可以看出使用的寄存器.以 mov ecx,edi 为例,

8BCF mov ecx, edi
第二个字节即 11 001(ecx) 111(edi).
在这个例子中,第二个字节用于指定寄存器,但并不总是这样,对于某些 OpCode,它被用于附加的 Code 位.MUL/iMUL reg 就是如此:

MUL 1111011w:11 [100] reg
IMUL 1111011w:11 [101] reg
对于 1111011w,100 意味着 MUL,而 101 则为 IMUL.

你应该记住的是----如果你不练习,那我这些话对于你没有任何鸟用.
在 OD 中输入一些 MUL/IMUL 指令的 OpCode:
1) 使第一个字节是 F7(bit w = 1),使用全寄存器；F6,局部寄存器.
2) 使第二字节的第一个十六进制数为 E,第二个小于 8,code/reg 的最低位为 0,即 code/reg 为 100,即 MUl；>=8,code/reg 的最低位为 1,即 code/reg 为 101,即 IMUl.
试着练习下.

现在你肯定对 MUL/IMUL 指令比较顺手了,那么你可以试试 DIV/IDIV 指令:

DIV 1111011w:11 110 reg
IDIV 1111011w:11 111 reg
可以看到只有 code/reg 域和 MUL/IMUL 的有些不一样,实际上对于这 4 条指令:

code/reg instruction
100 MUL
101 IMUL
110 DIV
111 IDIV
code/reg 域决定了这是什么指令(MUL,IMUL,DIV,IDIV).

不仅仅是 MUL,IMUL,DIV,IDIV reg,
MUL,IMUL,DIV,IDIV [mem]也是如此:
在 OD 中输入如下指令:

MUL EBX
IMUL EBX
DIV EBX
IDIV EBX
MUL [EBX]
IMUL [EBX]
DIV [EBX]
IDIV [EBX]
将每条指令的第二个字节转化为 2 : 3 : 3 的格式,然后比较它们的异同.如:

Second byte instruction
11 100 011 MUL EBX
11 101 011 IMUL EBX

.....

(OD 的结果如下:

F7E3 mul ebx

F7EB imul ebx

F7F3 div ebx

F7FB idiv ebx

F723 mul dword ptr [ebx]

F72B imul dword ptr [ebx]

F733 div dword ptr [ebx]

F73B idiv dword ptr [ebx]).

我们来看看 modr/m 和 sib 域,modr/m 和 sib 用于指定操作数.

关于 modr/m 和 sib,我们应当记住的第一点是:

可能只有 modr/m,也可能既有 modr/m 又有 sib;不可能只有 sib 而没有 modr/m.或者你可以这样的认为 sib 是 modr/m 的扩展.

我们应当记住的第二件事是:

modr/m 和 sib 都拥有同样的格式 2 : 3 : 3.

modr/m 的高两位被称为 mod(mode 即模式),可以有 2^2=4 种值:

11 - 没有内存操作数,即所有的操作数都是寄存器

10,01,00 - 其中一个操作数是内存操作数,我们等下再讨论详细的情形.

sib 的高两位可以有 2^2=4 种值,对应着被索引寄存器(index register)相应的倍数 1,2,4,8.

modr/m 和 sib 剩下的两个 3 bit 通常被寄存器用于寻址操作数,但我们稍后也会看到一些别的用途.

第三就是:

modr/m 和 sib 能指定什么样的操作数,以及不能指定什么样的操作数.

它们不能指定预定义的操作数.例如在串操作指令中是没有 modr/m 和 sib 字节的,以及 MUL/DIV 等的结果是预先定义的.

它们不能指定立即数,因为在处理器看来并不是立即数而是 OpCode 的一部分.

它们只能自定非预定义的寄存器操作数,以及带 displacement 的内存操作数.displacement 紧随 ModR/M 和 sib,它的大小是由 ModR/M 域指定的:

00 - 无 displacement

01 - 8 位的 displacement

10 - 32 位的 displacement

我们先来解答三个简单的问题.处理器是如何知道:

1. OpCode 有 1 个还是 2 个操作数?

2. 什么样的寄存器集被 ModR/M 和 sib 使用,全寄存器还是局部寄存器?

3. 若有两个操作数,那么哪一个是源,哪一个是目的?

答案是:处理器是根据 Code 域而不是 modr/m 和 sib 域得知这些信息的.(原文如此,似乎与第一个问题的答案矛盾)

第一个问题.

若指令只使用了寄存器操作数,即不使用内存操作数,modr/m 的高两位是 11,也就是这样:

[8 位 code 域]:11 *** ***

例如 mov reg,reg:

100010dw:11 reg reg

所以第一个问题的答案是:

处理器通过 modr/m 的高两位得知是 1 个还是 2 个操作数,若只有一个操作数,那么 code/reg 的低三位即寄存器操作数,中间的三位则是 code 域的扩展. 例:

MUL EBX

1111 011 1: 11 100 011

看着第二个字节

11 100 011

11 - mod 域,只包含寄存器操作数

100 - code/reg field,在这种情况下意味着只有一个操作数,即该域是 code 的扩展

011 - EBX 寄存器.在只有一个操作数的情况下,操作数一般放在 modr/m 的低三位.

练习:

在 OD 中输入一些 OpCode,仅改变:

1. code/reg
2. mem/reg

一般的,如果用到两个操作数,那么 code/reg 和 mem/reg 都被用来放置操作数.如:

MOV EAX,EBX

1000 1011: 11 000 011

第二个字节(modr/m)

11 - 仅使用寄存器

000 - EAX

011 - EBX

第二个问题:这是由 code 域的 w 位决定的.而且我们应该记住的是局部寄存器不能作为指针使用而只能作为寄存器.

if w = 1

全寄存器

else if w = 0

局部寄存器

例:

Mul reg

1111 011w:11 100 reg

w = 1

1111 0111:11 100 001 = MUL ECX

w = 0

1111 0110:11 100 001 = MUL CL

你也可以试试 div,idiv,mul,imul,改变它们 code 域的最低位,看看使用的是什么

寄存器集.

又如:

MOV reg,reg

1000 101w: 11 reg reg

w=1

1000 1011: 11 000 001 MOV EAX,ECX

w=0

1000 1010: 11 000 001 MOV AL,CL

它们的 modr/m 字节也是一样的,不同的仅仅是 code 域的 w 位.

试试别的使用寄存器操作数的指令,如

sub reg,reg

add reg,reg

改变它们的 w 位,并观察寄存器集合的变化.

第三个问题:这是由 code 域的 d 位(code 字节的第 1 位,设从第 0 位开始计数)决定的.如:

mov reg,reg

1000 10dw:11 reg reg

d = 1

1000 1011:11 000 001 = MOV EAX,ECX

d = 0

1000 1001:11 000 001 = MOV ECX,EAX

可以看到它们的 modr/m 字节也是一致的.正是由于 d 位,导致 MOV EAX,ECX 有两种编码方式:

1000 1011:11 000 001

1000 1001:11 001 000

这两个 OpCode 的功能是一致的.

现在试着在 OD 中输入

instr reg,reg

然后更换编码的方式,例如

add eax,ecx

OD 生成的 OpCode 是 03 C1,即

0000 0011:11 000 001

可以改成如下的编码方式:

1. 将第一个字节的第 2 位(code 域的 d 位)置 0

0000 0001

2. 然后交换第二个字节的 bit 0-2 和 bit 3-5(modr/m 的 code/reg 和 mem/reg)

11 001 000

完整的 OpCode 即

0000 0001:11 001 000

十六进制为 01C8

现在按 Ctrl+e,然后在对话框中输入 03 C1

OD 显示的助记符也是 add eax,ecx,即 03C1 和 01C8 对应同样的指令.

再看看他们的二进制格式:

ADD EAX,ECX

0000 0011:11 000 001 ;EAX=000 ECX=001

0000 0001:11 001 000

内存操作数由 modr/m 和 sib 字节指定.下面做一点笔记:**不管是有一个还是两个操作数,都只能有一个内存操作数.**

我们回到 modr/m 字节,它紧跟在 Code 块之后,是指定操作数的主要字节,格式是 2:3:3,

高 2 位 -- 模式(mode)
中间 3 位 -- 寄存器或 Code 域的扩展(reg / code extention)
低 3 位 -- 寄存器或内存操作数(reg / mem)

我们先看看低 3 位.这 3 位到底是指什么? 它能拥有什么样的有意义的值?
其实这 3 位在不同的情况下有不同的释义:

1. if mod = 11
   寄存器
2. if mod = 00 or mod = 01 or mod = 10
   寄存器作为指针被使用
   或 标志着内存操作数在后面的 sib 字节被指定
   或 标志着内存操作数直接由地址指定("flag" that memory operand is specifyed by direct value address)

例:(在所有的示例中 modr/m 都是 OpCode 的第二个字节)

---

mov eax,[ebx]
OpCode 是 8B03
modr/m 字节是 03,
二进制格式是 00 000 011
00 -- 使用无偏移量的内存操作数(using mem oprand without displacement)
000 -- eax
011 -- ebx

看看下面的差异:
mov eax,ebx modr/m: 11 000 011
mov eax,[ebx] modr/m: 00 000 011
他们的不同仅在 mod 部分,
mov eax,ebx 是 11
mov eax,[ebx] 是 00

---

mov ebx, dword ptr [400000h]
OpCode 是 8B 1D 00004000
modr/m 字节是 1D,
二进制格式是 00 011 101
00 -- codr 011 (ebx) memr 101
011 -- ebx
101 -- 当 mod = 00 时意指没有寄存器参与地址计算,该地址在后面的

字节中被指定.需要注意的是 101 在这里指的不是[ebp]

Mov ebx, [eax*4][ecx][3]
OpCode 是 8B 5C 81 03
modr/m 字节是 5C(sib: 81, displacement:03)
二进制格式是 01 011 100
01 -- 使用一字节偏移量的内存操作数(1 byte size displacement used)
000 -- ebx
100 -- 当 mod = 00 或 01,10,指 sib 字节将被需要,且紧随 modr/m 字节.这里
的 100 也不是指[esp].

我们来详细讨论一下.

我们以 INSTR reg, [reg] 或 INSTR [reg], reg 为例.之所以选取这两条指令是因为它拥有两个操作数,其中一个指向内存.你一定还记得,被当作非指针操作数的寄存器可以是任意的通用寄存器,包括全寄存器和局部寄存器,但像[reg]之类的被当作指针使用寄存器则只能是全寄存器.

instr reg1,reg2
instr reg1,[reg2]
它们的不同应该只是表现在 mod:
instr reg1,reg2 mod = 11
instr reg1,[reg2] mod = 00
例如:
mov eax, ebx 8BC3 1000 1011:11 000 011
mov eax, [ebx] 8BC3 1000 1011:00 000 011

实际上,对于指令 instr reg, [reg]内存操作数只能在 memr 比特域中被指定,那么
instr [reg], reg 呢?

其实啊,他们的 modr/m 是一样的,不同仅在之前我们说的 Code 域的 d 位(direction bit 即方向位):
mov eax,[ebx] 8B03 1000 1011:00 000 011
mov [ebx],eax 8903 1000 1001:00 000 011

我们再看看操作数的尺寸.下面 3 个的不同在于什么?
dword ptr [ebx]
word ptr [ebx]
byte ptr [ebx]
我们得记住两点:

1. OpCode 中的内存操作数是一个地址,更精确的说,寄存器和立即数(如果有的话)只是用来计算地址的.
2. 操作数的地址总是操作数的最低位,也就是说,对于
   byte ptr addrx
   word ptr addrx
   dword ptr addrx
   以及任意的尺寸,他们的地址是一致的.在 32 位寻址模式中,地址偏移量总

是 32 位值(没有 67h 前缀),用来寻址的寄存器也总是 32 位的寄存器.

还有就是我们通常使用 Code 域中的 w 位,以及 66h 前缀来指定操作数的尺寸.

bit d = 1; mem -> reg

mov eax, dword ptr [ebx] 8B03 1000 1011: 00 000 011

mov ax, word ptr [ebx] 668B03 同上,仅多有前缀 66h

mov al, byte ptr [ebx] 8A03 1000 1010: 00 000 011

那么现在就请你编码当 d = 0 时的 OpCode.

通晓 OpCode 格式对于仅通过汇编指令或助记符快速得出相应 OpCode 大小是非常有用的.大多数 OpCode 的尺寸可以通过将 Code 域的大小(大多数情况下是 1 字节)加上操作数的尺寸.

对于 instr reg, reg 类型的指令,instr = 1 字节,操作数部分也是 1 字节,一共就是两字节.如果操作数是 16 位的,那么就会多一字节的前缀.

而对于 instr reg, [reg] 和 instr [reg], reg,如果[reg]不是[ebp]和[esp]的话,那地址部分同样是 1 字节.因此,如果你使用[ebp]和[esp]将多耗费一些额外的字节(没有立即数 without displacement).这样的话 OpCode 也是两字节.同样的,如果操作数是 16 位的,那么就会多一字节的前缀.

偏移 和 mod 00,01,10

我们知道,如果 OpCode 有两个寄存器操作数,我们仅需在 modr/m 字节中指定,此时 mod = 11,而 reg 和 mem 比特域分别用来放置两个操作数.如字节 11 000 001 指示操作数为 eax 和 ecx; 而 11 011 111 则是 ebx 和 edi 等等.....

你应该还记得,如果需要指定两个操作数,其中一个是指针,另一个是寄存器,也仅仅需要 modr/m 字节(不能有[ebp],[esp]).

当 mod = 00,code/reg 域是寄存器操作数,而 mem/reg 则是指针操作数.如字节 00 000 001 指示 eax 是寄存器操作数,而[ecx]则是内存操作数; 而 00 011 111 则是,ebx 是寄存器操作数,[edi]是内存操作数.

最后我们来看看为什么仅通过 modr/m 地址字节不能指定[ebp]和[esp].

这是因为当 mem/reg 为[ebp],且 mod = 00 时,意味着没有寄存器被作为指针使用,地址被接下来的一个双字指定; 而 mem/reg 为[esp],且 mod 为 00,01,或 10 时,则是指 sib 字节将会出现,地址部分在 sib 中被指定.

当然 mov reg,[esp]和 mov reg,[ebp]肯定是合法的指令,但是它们的编码方式还有点特别,我们等下就会讲到这个.

我们仍然回到 mod 00,01,10 的主题.

我们将进行一些有趣的优化:在许多 OpCode 包含有双字的立即数,而这其实仅仅需要一个字节.我们将对有符号字节和有符号双字分别进行.

如果立即数是负数且大于等于-128,有符号双字的低位与有符号字节一致,且都是数值位,而高位全部是 1.如-2 双字是 FFFF FFFE, 字节是 FE.

(译者注:计算机内部编码分为,原码,反码,补码,移码.

移码仅用于表示浮点数;

原码是直接将数字翻译成二进制,最高位为符号位,0 表示正,1 表示负,如 1 的原码是 0000 0001,-1 的原码则是 1000 0001;

正数的原码,反码和补码是一致的;

负数反码的编码方式是,符号位不变,其余位取反,即 1 变为 0,0 变为 1,如-1 的反码是 1111 1110；

负数补码的编码方式是在反码的基础上加 1,-1 的补码是 1111 1111；

一般的计算机内部通常都是按补码方式存储的,这样的原因是便于实现计算机内部数值计算；

对于 Intel 来说,是按照 4 字节逆序补码存储的)

如果为正且小于等于 127,有符号双字的低位与有符号字节一致,且都是数值位,而高位全部是 0.如 2　双字是 0000 0002, 字节是 02.

这样一来就允许处理器将有符号字节扩展为有符号双字,换句话说----使用字节代替双字,这样将使带双字立即数的 OpCode 的尺寸大大减小.

但是我们应该如何编码呢？

我们先来区分一下立即数的类别:

1. 立即操作数,如　　and　eax, 03　　；　03 即立即操作数

2. 偏移(displacement),如 and　eax, [ecx][3]　；　3 为偏移量,用来计算地址

这两中类型的立即数编码方式是不一样的.

我们先看偏移是如何编码的.

mod 域指示是否有偏移:

mod = 00 : 没有偏移,但当 mem/r 为 ebp(101)时,没有寄存器作为指针使用,仅有 32 为的偏移；

mod = 01 有 1 字节的偏移

mod = 10 有 1 双字的偏移

如:

```
mov   eax, [ebx]
8B03
03 是 modr/m 字节,二进制格式 00 000 011
00(mod)   ---  没有偏移
000       --- eax
011       --- [ebx]
-----------------------------------------------------
mov   eax, [ebx][-2]
8B43 FE
43 是 modr/m 字节,二进制格式 01 000 011
01(mod)   ---  有 1 字节偏移
000       --- eax
011       --- [ebx]
FE        ---  偏移,将被处理器扩展为 FFFF FFFE
-----------------------------------------------------
mov   eax, [ebx][410000h]
8B 83 00004100
83 是 modr/m 字节,二进制格式 10 000 011
10(mod)   ---  有 1 双节偏移
000       --- eax
011       --- [ebx]
00004100 ---  偏移
```

现在我们就可以对[ebp]进行编码了.[ebp]的编码方式不是 instr reg, [ebp](mod = 00)而是 instr reg, [ebp][0](mod = 01).如果你在 OD 中输入一下两条指令

```
mov   eax, [ebp]
mov   eax, [ebp][0]
```

可以看到它们的 OpCode 是一致的,都是

```
8B45 00          mov        eax, dword ptr [ebp]                ;   mov eax,[ebp].
```

顺便说一下:

instr reg, [ebp]比 instr reg, [除 ebp,esp 外的寄存器]要多 1 字节;

instr reg, [ebp][不等于 0 的偏移]和 instr reg, [任意寄存器][不等于 0 的偏移]的尺寸是一致的.

当然,在示例中第一个操作数都是寄存器操作数,第二个是内存操作数,其实交换一下他们的位置对我们的结论没有任何影响,不同仅在 Code 域的 d 位而已.

SIB

接下来我们将讨论:

1. 何处指示着有 sib 出现?
2. sib 的格式
3. Esp 不能为索引寄存器(index register),如果在 index 位域放置 esp 会产生什么问题?
4. Ebp 寄存器---对于[ebp][reg*scale]只有包含双字偏移的 OpCode.
5. instr reg,[esp]如何编码?
6. 地址公式

第一个问题是,何处指示着有 sib 出现? 我们之前说过,sib 是跟随 modr/m 字节出现的,没有 modr/m 就没有 sib 字节.实际上 sib 字节仅在需要计算内存地址时才需要.

处理器通过两点得知 sib 的存在:

1. mod 位域指示有内存操作数(mod = 00, 01, 10);
2. mem/reg 域为 100(esp),所以像如下格式的 modr/m 字节:

```
00 *** 100
01 *** 100
10 *** 100
```

都指示这 sib 字节紧随着 modr/m.

第二个问题,sib 的格式.

sib 的格式是:SS:III:BBB,

SS: 高 2 位,为索引寄存器的倍率(scale):

```
00 = 1
01 = 2
10 = 4
11 = 8
```

如 [reg][reg] --- 00 *** ***

[reg*2][reg] --- 01 *** ***

[reg*4][reg] --- 10 *** ***
[reg*8][reg] --- 11 *** ***

III:中间的 3 位,为索引寄存器(index register),除 esp 外的任意通用寄存器.

如 [eax][reg] --- 00 000 ***
[ecx*8][reg] --- 11 001 ***
[edx][reg] --- 00 010 ***
[ebx*2][reg] --- 01 011 *** (***为基寄存器 base reg)

BBB:低 3 位,基寄存器(base register)

如 [ecx*4][eax] --- 10 001 000

第三个问题:Esp 不能为索引寄存器.

如果如果在 index 位域放置 esp 将会被忽略,而不管 scale 是什么值,此时只有基寄存器参与地址计算,这样的结果就像是没有 sib 字节,而仅在 modr/m 的 mem/reg 位域指定基寄存器.即:

modr/m sib
** *** 100 ** 100 reg
** *** reg

是一样的.

对于[eax]就有两种编码方式:

modr/m sib
** *** 100 ** 100 000
** *** 000

所以对于 mov eax, dword ptr [eax]就有 5 种编码(8B 是 mov 的 Code 域):

8B00 mov eax, dword ptr [eax]
8B0420 mov eax, dword ptr [eax]
8B0460 mov eax, dword ptr [eax]
8B04A0 mov eax, dword ptr [eax]
8B04E0 mov eax, dword ptr [eax]

第五个问题.instr reg,[esp]如何编码? 回答这个问题也就是回答[esp][偏移量]是如何编码的.

reg,[esp]的编码方式是:

modr/m sib
00 reg 100 ** 100 100

若 mod 为 01 或 10,上面的格式就变成了

mod 01 reg, [esp][单字节偏移]
mod 01 reg, [esp][一双字偏移]

这也就是[esp][偏移量]的编码方式.

也就是说在 sib 中不指定索引寄存器往往是为了编码[esp].

是否还有别的方案呢? 如果有索引寄存器而没有基寄存器可行吗? 要回答这个问题首先得回答上面提到的第 4 问.

EBP -对于[ebp][reg*scale]只有包含双字偏移的 OpCode.

实际上 base 为 101(ebp)且 mod = 00 和 modr/m 的 mem/reg 域为 ebp 且 mod = 00 是一致的,在这种情况下:

1. 没有基寄存器

2. 有 32 位的偏移量作为基(base)

这样就导致了:

对[index*scale][base]编码仅需一个 sib 字节,而对于[index*scale]则是一个 sib 字节加上 4 字节的偏移.

```
mov   eax, [ebx*4][ecx]
OpCode: 8B0499, Code:8B, modr/m: 04, sib: 99
----------------------------------------
mov   eax, [ebx*4]
OpCode: 8B049D 00000000, Code: 8B, modr/m: 04, sib: 99,
displacement:00000000
```

对 OpCode 地址部分尺寸的总结

要计算 OpCode 的尺寸,我们往往先计算 Code 域的尺寸和地址部分的尺寸.对于“老的”通用的指令(如 mov, push/pop, inc/dec, add/sub, and, xor 等),是 1 字节,也许在 modr/m 中有 Code 域的扩展,但这属于 modr/m 字节,可以被忽略,因为我们把它当作是地址部分了.

而“新的”指令集则多有一个 0Fh 的前缀用来指示该指令属于新的指令集(你也可以将该字节理解为转移字符),所以它的尺寸就是 2 字节.

关于地址部分的第一点是:

对于一个带有一个[mem]的操作数的地址部分和两个操作数其中一个是[mem]和另外的寄存器的地址部分,它们的大小是一样的.(Size of addr part with one [mem] operand,and 2 operands where one is [mem] and other reg are the same)如:

```
push   [eax][ebx*4][12]
mov    ecx, [eax][ebx*4][12]
```

它们的 OpCode 大小和地址部分的大小都是一样的.

(在 OD 中是这样的:

```
FF7498 12      push    dword ptr [eax+ebx*4+12]
8B4C98 12      mov     ecx, dword ptr [eax+ebx*4+12] )
```

在我们分析地址部分的格式和尺寸之前,我们先来回忆一下之前的 5 条指令,它们是:inc/dec, push/pop, xchg.它们都是单字节指令,仅有 Code 域,格式是 5:3,其中高 5 位是 Code,低 3 位用于指定寄存器,这里的寄存器是通用的全寄存器.我们之前将他们称之为 exeption,并且讨论过不使用通用全寄存器的情形,它们的 OpCode 的尺寸可以通过通用的规律----code + 地址部分 计算.

下面我们列举一下所有的地址格式和它们的格式:

操作数	地址部分尺寸(字节)	格式
reg	1	modr/m=11 code reg
[reg] 且[reg] != [esp] / [ebp]	1	modr/m=00 code reg
[esp]	2	modr/m=00 code 100 sib=00 100 100
[ebp]	2	modr/m=01 code 101

		1 字节全 0(偏移)
reg,reg	1	modr/m=11 reg1 reg2
[reg],reg 或 reg,[reg] 且[reg]!=[esp]/[ebp]	1	modr/m=00 reg [reg]
reg,[ebp] 或[ebp],reg	2	modr/m=01 reg 101 1 字节全 0(偏移)
reg,[esp] 或[esp],reg	2	modr/m=00 reg 100 sib=00 100 100
reg,[imm32] 或[imm32],reg	5	modr/m=00 reg 101 4 字节 imm32
reg,[reg][imm8] 或[reg][imm8],reg 且[reg]!= [esp]	2	modr/m = 01 reg [reg] 1 字节 imm8
reg,[esp][imm8] 或[esp][imm8],reg	3	modr/m = 01 reg 100 sib = 00 100 100 1 字节 imm8
reg,[reg][imm32] 或[reg][imm32],reg 且[reg]!=[esp]	5	modr/m = 10 reg [reg] 4 字节 imm32
reg,[esp][imm32] 或[esp][imm32],reg	6	modr/m = 10 reg 100 sib = 00 100 100 4 字节 imm32
reg,[indexreg*scale][basereg]或 [indexreg*scale][basereg],reg 且 index!=esp base!= ebp	2	modr/m = 00 reg 100 sib 字节
reg,[indexreg*scale][ebp] 或 [indexreg*scale][ebp],reg	3	modr/m = 01 reg 100 sib = ss iii 101 1 字节全 0(偏移)
reg,[indexreg*scale] 或 [indexreg*scale],reg	6	modr/m = 00 reg 100 sib = ss iii 101 4 字节全 0(偏移)
reg,[indexreg*scale][basereg][imm8] 或 [indexreg*scale][basereg][imm8],reg	3	modr/m = 01 reg 100 sib 字节 1 字节 imm8 偏移
reg,[indexreg*scale][basereg][imm32] 或 [indexreg*scale][basereg][imm32],reg	6	modr/m = 10 reg 100 sib 字节 4 字节 imm32 偏移
注:交换两操作数位置,仅是 Code 域的 d 位不同.		