Winsen Jiansbomber

Jimbowhy的杂碎

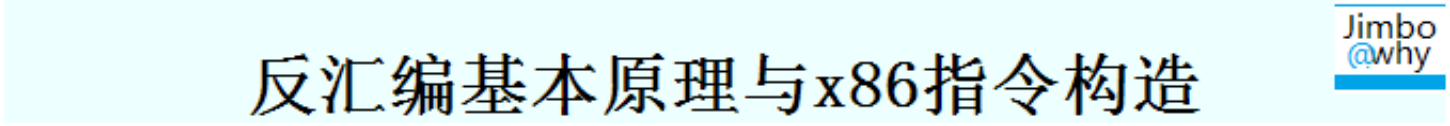
Jimbo

反汇编基本原理与x86指令构造

标签：汇编 反汇编原理 x86

2014-05-28 00:18 1635人阅读 评论(5) 收藏 举报

以上黑带为文章的文字内容。以下为对应正文图片版式。



等级： BLOG 4
排名： 千里之外

原创： 64篇
转载： 0篇
译文： 1篇
评论： 34条

文章搜索

文章分类

C++ (23)
JavaScript (7)
ActionScript (6)
DOC88下载破解 (2)
PDF目录标签生成脚本 (1)
随机数 (1)
反汇编 (2)
Assembly (4)
C (9)
Forth (2)
Make (1)
电路 (1)
Mathematic (2)
数字图像处理 (3)
SQL (1)
mywife-cc (1)
DEBUG (1)

概要：旨在讲述程序的二进制代码转换 到汇编，即反汇编的基本原理。以及 x86 架构的 CPU 的指令构造，有这个基础后就可以自己编写汇编程序了，也可以将二进制代码数据转换成汇编助记指令。当然，把本文当作手册的阅读指导也是可以的。本文还讲述 了 DEBUG 工具的部分功能，32位平台下有一个 DEBUG32 版本可以配合 DOSBOX 工具运行在 Windos 7 这些 NT 系统上，DEBUG 要使用 MSDOS 5.0 版本中的。这是一个十分有用的工具，它同时是 x86 的汇编程序，又是反汇编程序，同时又可以作为交互命令使用，如读写磁盘扇区等系统功能。本文需要汇编与计算机原理等基础知识，如果阅读时发现无法理解的内 容就表明需要补充基础知识，此时请停下来，或者跳过部分内容也是不错的阅读方法。再次，祝贺你，当你看到这篇文章时，你已经开始学会如何把握计算机软件领 域的核心所在了！

“不管现在流行什么语言，你都可以肯定十年二十年之后它不再风光。我总是在自己的书中写些不时髦的东西，但这些东西却值得后代子孙记取。” -- Donald E. Knuth

问题由反汇编开始

对已经开始接触反汇编深层的读者，可以已经使用过甚至自己编写过反汇编引擎了，如 x86 Disassembler Librarys 。所谓反汇编即通过 CPU 的指令构造原理将指令的二进制代码转换成**助记符 Mnemonic** 的过程，而二进制表达的指令就称为**操作码 OpCode**， 这是 CPU 可以理解的指令形式。那么先来看看一条简单的代码片断，操作码为数据：eb 00 eb fe 90。使用 DEBUG 工具，通过 e cs:ip 命令在当前代码段的入口来输入以下指令代码，输入完一个字节按空格，完成时按回车结束，然后通过 u cs:ip 得到类似以下反汇编代码：

```
1C8D:0100 eb00    jmp short 0102
1C8D:0102 ebfe    jmp short 0102
1C8D:0103 90      NOP
```

通 过 DEBUG 的汇编命令也可以直接输入以上的汇编程序，现在通过 t 命令来执行单步调试，看看程序如何运行。在 DEBUG 会高亮显示补指令改动过的内容，通过 r 命令来显示当前寄存器的值。其中就有 IP 寄存器，调试时第一条指令执行后，显示 IP 改变为 0102，这是因为 jmp 跳转指令起作用了。第二条指令也是一条 jmp 指令，可留意到其操作码极为相似。可以联想，操作码的第二个字节其实就是跳转的地址，即指令的**操作数 Operand**。 因为第一条指令操作码为 2-Byte，跳转的偏移量为 0x00+2 即跳转到 0x0100+2=0x0102；因此第二条指令跳转的偏移地址应为 0xFE+2=0x100，但是由于此指令的操作数为一个字节，因此截留结果的低 8-bit 即最终偏移量为 0，正因此，程序在继续执行时 IP 始终停留在 0x0102。

这里就出现了一个疑问，eb00 这样的操作码是如何得到的呢？这个问题关系到，DEBUG 如何反汇编 eb00 得到 jmp 这样的汇编指令，也关系到 DEBUG 如何将汇编指令转译为 CPU 认识的操作码。

正则 (1)
LeetCode (1)
GDI (2)
i18n (1)
DOS (1)
水果编曲 (1)
PS (1)

文章存档

2016年04月 (7)
2016年03月 (12)
2016年02月 (15)
2016年01月 (6)
2015年08月 (5)

展开

阅读排行

mywife.cc 神一样的存在 (11194)
EBT 道客巴巴的加密与破 (2677)
BWAI星际争霸与AI开发 (2164)
Mathematica字符串处理 (2161)
Script Control 组件Win7 (2006)
SWF代码分析与破解之路 (1916)
EBT 道客巴巴的加密与破 (1871)
反汇编基本原理与x86指 (1634)
EBT 道客巴巴的加密与破 (1586)
Shopex模板机制总览(摘 (1530)

特的汇编指令，也关系到 DEBUG 如何将汇编指令转译为 CPU 认识的操作码。

指令操作码构造

先来看一段代码：

```
1C8D:0100 B80000    mov ax, 0
1C8D:0103 B80100    mov ax, 1
1C8D:0106 BB0000    mov bx, 0
1C8D:0109 BB0100    mov bx, 1
```

以 上程序可以通过推断了解到，操作码 B8 可能代表了 mov ax，BB 代表了 mov bx，寄存器的信息已经包含在操作码，CPU 能够将其与特定的寄存器关联，0、1 这些操作数也包含在指令操作码内。有一个重要的环节，x86 CPU 最初是从 16 位机发展起来的，这种机器的运行时内存通过地址总线直接存取，这种模式就称作**实地址模式 Real Mode**，与之后来研发的 32 位机 80386 的引入内存分页机制及保护机制，内存可以通过分页的形式来管理并在保护机制下运行，这种模式称为**保护模式 Protect Mode**， 同时为运行 16 位的程序提供了一个虚拟 x86 模式 Virtual 8086 Mode 简称 V86，这种模式通过虚拟机技术实现，能有效地利用 80386 的多任务特性来运行多个实模式程序。在这之前是没有 eax 这样的 32 位的寄存器，但是由向下兼容的要求，以上指令的操作码在 16 位机上具有相同的作用。尽管如此，因为后来的 CPU 指令集在增长，在不同的模式下，相同的操作码会出现的不同的功能，例如：

```
1C8D:0100 66B800000000    mov eax, 00
```

在 16 位机上，将会解释为：

```
1C8D:0100 66        DB 66
1C8D:0101 B80000    mov ax, 0
1C8D:0104 0000      mov [bx+si], al
```

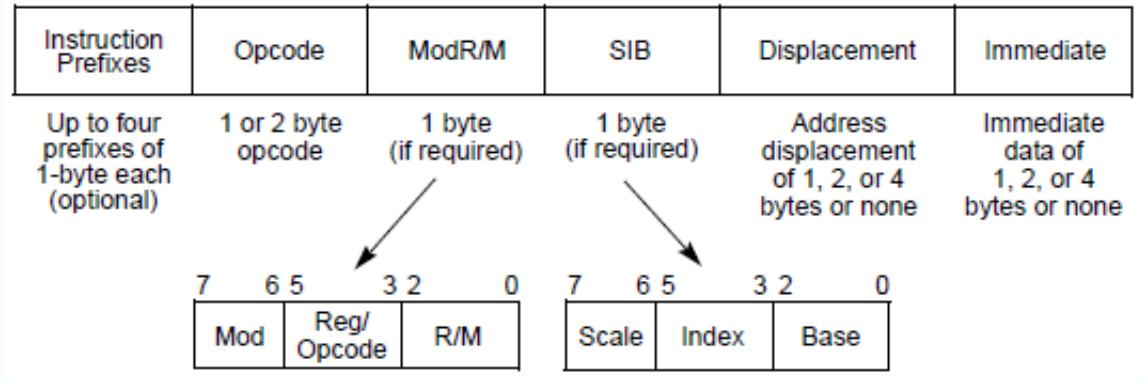
可见，CPU 的运行模式对操作的解码很重要。为了深入操作码的内部，那么只有找 CPU 厂商了，x86 当然就是找 Intel 了，它共享发布有开发者手册共为3卷7个PDF。到 Intel® 64 架构发布后，其对应的手册已经整理成了全集组成一个 PDF 文档。这套手册可以说是低层学习的必需文档，它详尽地记录了开发人员需要了解的 CPU 信息，第二卷全三册指令系统作为开发人员最基本的要求，自然也就是重点。

前面说了这些就是为了热身，在 x86 架构上，一个指令即操作码由6个部分构造而成：

1. Prefixes 指令代码的**前缀**，每指令最多可以有4个/种前缀修饰。有**操作数大小前缀**，如前面提到的 66，它指定 32-bit 操作数大小，即 66 前缀则表示 8-bit 操作数，按 16 位机上的约定，默认的操作为 16 位，不使用前缀；有重

32-bit 操作数大小，FE 前缀则表示 8-bit 操作数，按 16 位机上的传统，默认的操作为 16 位，不使用前缀。有**重复类型前缀**，如最常见的 F3 表示 REP、REPE、REPZ 重复前缀，还有 F2 表示 REPNE、REPNZ 前缀。有**段超越前缀**，2E 对应 CS、36 对应 SS、3E 对应 DS、26 对应 ES，64、65 则对应 FS、GS，段超越是可内存寻址有关的内容。还有**寻址地址大小前缀**，67 表示 32-bit 内存寻址。以及官方手册中提及的一些特别功能的前缀，这些前缀可以按任意的顺序与指令码组合；

2. Code 指令本体，且称**指令码**，是唯一必需的部分，如前面 B80000 中的 B8 就是一个 Code。然而，指令码还可能有一个额外的 3-bit 存放在 ModR/M 处；
3. ModR/M，尽管这部分可能包含指令码的一额外字节，但它主要功能是用来标识操作数的内存寻址信息，就此称作**寻址模式**， 显得恰当。它被细分为三个区域，高 2-bit 为 mod 区、低 3-bit 为 R/M 区。其中 Mod 和 R/M 共 5-bit 有32 种可能值，刚好用来表示 8 个通用寄存器和 24 种寻址模式。R/M 还可以用来指定寄存器作为操作数，它是 Register 和 Memory 的缩写。中间 3-bit 为 Reg/Opcode 区，中间斜杠表示或的意思，它表明此区域可表示寄存器操作数或作为指令码的额外的 3-bit 来使用，具体用途要应指令来决定。对于特定的比例变址寻址模式，才需要用到操作码的 SIB 部分。详细内容见下面的 16-bit 或 32-bit 寻址模式图表。
4. SIB，它是 Scale Index Base 的缩写，是寻址模式的内容补充，是对比例变址寻址方式的详细说明。它也被细分为三个区域，高 2-bit 作为比例因子 Scale 使用，在《深入x86的内存寻址》也提到这种寻址方式的比例因此只能是 1、2、4、8 就和 Scale 所占的 2-bit 有关，它只能表示这四种可能值 。中间 3-bit 为 Index，用来指定一个变址寄存器。正如《深入x86的内存寻址》所描述，除 ESP 外的七个通用寄存器都可以用作比例变址地址，在下面的 SIB 寻址图表中可以看到第一栏的 none 取代了 ESP。低 3-bit 为 Base，用来指定基址寄存器。
5. Displacement **偏移量**，可选，可为 1、2、4 字节。在《深入x86的内存寻址》中提到。
6. Immediate **立即数**，可选，可为 1、2、4 字节。例子 B80000 指令中的 0000 就是**立即数**，即包含在操作码的数据。



总之，一条指令最长也不会超过 16 个字节。

深入指令构造

有 了上面的基础后，就可以理解以下图表了。这些图表可以理解为三维图表，分别使用 ModR/M 或者 SIB 的三个区域来查找用于构造指令操作码的二进制数值。16-bit 或 32-bit 寻址模式图表的区别就是前者的是 16-bit 的地址，后者则为 32-bit 地址。要进行反汇编前，需要先了解手册中的指令集内容如何阅读。以 add 指令为例，手册给出定义表格：

```
Opcode    Instruction         Clocks    Description

04 ib     ADD AL,imm8          2        Add immediate byte to AL
05 iw     ADD AX,imm16         2        Add immediate word to AX
05 id     ADD EAX,imm32        2        Add immediate dword to EAX
80 /0 ib  ADD r/m8,imm8        2/7      Add immediate byte to r/m byte
81 /0 iw  ADD r/m16,imm16      2/7      Add immediate word to r/m word
81 /0 id  ADD r/m32,imm32      2/7      Add immediate dword to r/m dword
00 /r     ADD r/m8,r8          2/7      Add byte register to r/m byte
......
```

第一列指明了指令码的取值，可以看到同一条指令它具有不同的表达形式。同时，对于不同指令还可含有其它特定信息：

- /digit 这里的 digit 指 0－7 的数值，在寻址模式表中有对应的值，表示指令操作数只使用 R/M 部分。这里的数值同时也作为指令码的额外 3-bit 来使用。
- /r 表明指令的寻址模式中指定了寄存器间接寻址操作数和 R/M 操作数。
- cb cw cd cp 分别表示代码偏移值即指令长度为 1-Byte、2-Byte、4-Byte、6Byte，也可能是新的段寄存值。
- ib iw id 分别表示立即数为 1-Byte、2-Byte、4-Byte，指令码可以确认它的符号。
- +rb +rw +rd 分别表示寄存器代码，+号表示将寄存器代码与前面的 16 进制指令码相相加形成一个字节的指令码。寄存器代码如下所示：

```
   rb        rw        rd
AL = 0    AX = 0    EAX = 0
CL = 1    CX = 1    ECX = 1
DL = 2    DX = 2    EDX = 2
BL = 3    BX = 3    EBX = 3
AH = 4    SP = 4    ESP = 4
```

```
CH = 5     BP = 5     EBP = 5
DH = 6     SI = 6     ESI = 6
BH = 7     DI = 7     EDI = 7
```

第二列指明了指令的伪汇编代码的解释，这部分和汇编语言有相似的地方，比较容易理解，主要是在操作数的解释上有些需要罗列的内容：

- rel8: 表示操作数是一个 8-bit 相对寻址，范围在相同的代码段，当前指令末端的前 128-Byte 到后 127-Byte。rel16 和 rel32 作用类似，只是操作数的大小不同，范围也加大了。
- ptr16:16, ptr16:32: 远代码指针，操作数包含了代码段地址和偏移，用于代码寻址，CPU 会按指针的代码段地址来设置 CS 寄存器以实现远跳转。
- r8: 表示一个 8-bit 通用寄存器 AL, CL, DL, BL, AH, CH, DH, BH。
- r16: 表示一个 16-bit 通用寄存器 AX, CX, DX, BX, SP, BP, SI, DI。
- r32: 表示一个 32-bit 通用寄存器 EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI。
- imm8: 表示一个 8-bit 立即数，带符号，取值为 [-128~+127]。imm16 和 imm32 也类似，只是取值范围增大。
- r/m8: 表示操作数可以为 r8 或者 8-bit 的内存数据。r/m16 则对应一个 r16 或者 16-bit 的内存数据，r/m32 同理。
- m8: 表示8-bit 的内存数据，由 DS:SI 或 ES:DI 寻址得到，专用于字符串指令。m16 和 m32 同理。
- m16:16, m16:32: 表示操作数为远指针寻址，冒号前的值为段基址，后面的值为相应偏移。
- m16 & 32, m16 & 16, m32 & 32:a memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. m16 & 16 and m32 & 32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. m16 & 32 is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.
- moffs8, moffs16, moffs32: 表示一个内存偏移值，对于不含 ModR/M 的指令，将在指令码中包含内存偏移信息，它同时决定了地址方式。
- Sreg: 表示一个段寄存器，ES=0, CS=1, SS=2, DS=3, FS=4, GS=5。

第 三列、第四列则分别说明了指令所消耗的时钟周期和指令的功能。文档中还有其它详细的内容，如 Operation 使用

伪代码描述了 CPU 内部执行指令的过程，Flags Affected 描述了标志寄存器受指令影响的标志位，以及指令是否触发 CPU 异常等等信息。

以 dec 指令为例，dec 基本的指令码为 48+rw 或 FE /1，后者表明指令附加了 ModR/M 部分并且 Reg/Opcode 区域指定为指令码的额外的 3-bit，取值为 1。当 dec 结合不同的寄存器时，它的指令码就会产生微小的变化：

```
1C8D:0100 48      dec ax
1C8D:0101 FEC8    dec al
1C8D:0103 4B      dec bx
1C8D:0104 FECB    dec bl
1C8D:0106 49      dec cx
1C8D:0107 FEC9    dec cl
```

又以汇编代码为例：

```
1C8D:0100 67668B00    mov eax,[eax]
1C8D:0104 678B00      mov ax, [eax]
1C8D:0107 678A00      mov al, [eax]
1C8D:010A 67668900    mov [eax], eax
1C8D:010E 678900      mov [eax], ax
1C8D:0111 678800      mov [eax], al
1C8D:0114 66A3        mov eax, eax
```

先来分解第一条指令，首 先，通过 [eax] 这样的寄存器间接寻址方式可以确定 CPU 是 32-bit 寻址的，当然了反汇编是不会事先得知指令的细节的。但是从操作码中还是可以找到线索，67 表明它是指令前缀，指定了内存地址为 32-bit 模式；其次 66 是也表明是一个指令前缀，指定操作数为 32-bit；再次，根据手册定义得到 8B 是 mov 的指令码，这就可以确定它的操作数只为寄存器，而且源操作数是内存或寄存器寻址；接下来的 00 就为寻址模式，而不是其它什么立即数什么的啦。那么就将 00 按 ModR/M 的三个区域展开，得到 Mod=0，Reg/Opcode=0，R/M=0。在对应的 32-bit 寻址模式表中找到 Mod=0的行，这里指令确定了这个区域指定一个寄存器而不是指令码的 3-bit，因此找到 REG=0 的例，再找到 R/M=0 的行，即红、绿、蓝线框依次圈到的内容，最会找到的是 00 这个值，就是前面分解的操作码。再次说明，Mod 和 R/M 确定了源操作数是通过 [EAX] 的寄存器间接寻址，然后前缀 66、指令码 8B 和 Reg/Opcode 确定了目标操作数是 EAX 寄存器。接下来两条代码也如此一番分析，即可以得到反汇编指令，这就是反汇编的原理所在。

第四条指令和第一条指令只在操作数的位置上调换了一下，按前的方法推演，可以

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP[1] EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	EQ	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added

2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD

ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 Base = Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF

none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTE:

1. The [*] nomenclature means a disp32 with no base if MOD is 00, [EBP] otherwise. This provides the following addressing modes:

   disp32[index] (MOD=00).
   disp8[EBP][index](MOD=01).
   disp32[EBP][index](MOD=10).

汇编实践

有了以上信息，就可以汇编及反汇编指令了，这些内容是个基础。例如，汇编最简单的一条指令：

```
mov eax, [edx]
```

首先，可以确定的信息就包含 32-bit 的内存寻址模式，因此代码前缀有 67；使用 32-bit 寄存寻址，因此代码前缀有 66；接下来需要根据手册定义得到到 mov 的指令码，8B，即以下这条指令形式：

```
Opcode   Instruction       Clocks        Description
8B  /r   MOV r32,r/m32      2/4           Move r/m dword to dword register
```

再根据源操作数的[edx]间接寄存器寻址定位到寻址模式表格的数据：

```
+-Effective Address-+ +Mod R/M+ +---------ModR/M Values in Hexadecimal-------+
[EDX]                  00 010   02    0A    12    1A    22    2A    32    3A
```

然后，再根据目标操作数 eax 定位到 REG=000 这列得到 02。最终得到的操作码就为 67 66 8B 02，通过 DEBUG32 可以验证结果。

将上面的汇编指令的操作数的寻址方式反转来试试：

```
mov [eax], edx
```

同样，前缀 67 66 已经可以从汇编指令中得出，根据手册定义的 mov 的对应寻址模式的指令码则为 89，

再根据目标操作数的[eax]间接寄存器寻址定位到寻址模式表格的数据：

```
+-Effective Address-+ +Mod R/M+ +---------ModR/M Values in Hexadecimal-------+
[EAX]                  00 000   00    08    10    18    20    28    30    38
```

再找到源操作数 edx 对应的 REG=002 这一列，得到 10 这个值，最终操作码为 67 66 89 10。

注意，这个过程有点古怪，为了更全面理解这个过程，再来尝试汇编不含内存寻址的指令：

```
mov edx, eax   // 668BD0
mov eax, edx   // 668BC2
```

这条指令，比较有争议，因此手册中有种形式是符合这些指令表达的：

```
Opcode   Instruction       Clocks        Description
89  /r   MOV r/m32,r32      2/2           Move dword register to r/m dword
8B  /r   MOV r32,r/m32      2/4           Move r/m dword to dword register
```

那么究竟哪条指令适用呢？不是直接来看注解后所指出的指令码，假设指令系统通过第一条指令的源操作数 eax 定位到所在行：

```
+-Effective Address-+ +Mod R/M+ +---------ModR/M Values in Hexadecimal-------+
EAX/AX/AL              11 000   C0    C8    D0    D8    E0    E8    F0    F8
```

然后通过目标操作数 edx 定位到了 D0 这个值。对于第二条指令，也如此定位到了 C2 这个值，最终得到相应的指令码。为了验证这一规则，通过 DEBUG32 工具反汇编两条指令码得到以下内容，发现指令系统并不是按这样的规则构造指令码的：

```
mov eax, edx   // 6689D0
mov edx, eax   // 6689C2
```

而是通过 **r/m32** 指定的操作数来定位所在行，通过 **r32** 定位所在的列，这样才可以解释这里8条指令的构造。

```
1C8D:0100 6689D0    mov eax, edx
1C8D:0103 6689C2    mov edx, eax
1C8D:0106 668BD0    mov edx, eax
1C8D:0109 668BC2    mov eax, edx
1C8D:010C 67668B02  mov eax, [edx]
1C8D:0110 67668910  mov [eax],edx
1C8D:0114 67668902  mov [edx],eax
1C8D:0118 67668B10  mov edx, [eax]
```

最后通过一条 div ecx 指令来验证一下这个过程。div 指令有三种形式，其中 EAX/AX/AL 是隐含的，作为被除数的低

位数据，不需要在汇编指令中指出它。

Opcode	Instruction	Clocks	Description
F6 /6	DIV AL,r/m8	14/17	Unsigned divide AX by r/m byte(AL=Quo, AH=Rem)
F7 /6	DIV AX,r/m16	22/25	Unsigned divide DX:AX by r/m word (AX=Quo, DX=Rem)
F7 /6	DIV EAX,r/m32	38/41	Unsigned divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem)

指令中显式给的信息就有 ecx 这个 32-bit 的寄存器，因此可以得到 32-bit 操作数前缀 66 和指令码 F7，通过 /6 这条信息又可以优先定位到 Opcode=6 所在的列。再由 **r/m32** 指定的 ecx 所在的行得到，F1。将各部结构起来就得到了完整的操作码 66 F7 F1，这个过程就是完整的汇编。

第一次手工反汇编

有了汇编的基础后，再来反汇编显得更容易。例如，在 AutoNeoGrub.mbr 引导程序中提取的部分数据：

```
00000000 EB 5E 80 00 20 39 FF FF 00 00 00 00 00 00 00 00 .^.. 9.........
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
~ Duplicate Lines
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000060 FA 31 DB 8E D3 BC 80 05 E8 00 00 5B 81 EB 6B 00 .1.........[..k.
00000070 C1 EB 04 8C C8 01 C3 8E DB 53 6A 7D CB 68 00 20 .........Sj}.h.
```

CPU 在执行程序代码时，会先读入 EB，通过定义得到 EB 为一条短跳转 jmp 指令，通过手册也可以查找到这条指令的定义，它的操作数为 cb 占一个字节。因此可以知道拉下来的 5E 这个字节是个地址偏移，整条指令占 2 个字节，短跳转的实际偏移里为 5E+2＝60。为了方便查找，Sang Cho 制作了一份非常有用的 32-bit 寻址的 Pentium 指令集数据表供参考，也可以在 Intel 80386 Reference Programmer's Manual 的 CHM文档中方便地查找到，链接见附录。

接下来，程序跳转到 00000060 这里开始执行。同样读入一个字节 FA，它是一条 CLI 指令。

接着读入 31，它是一条 XOR r/m32,r32 或者 XOR r/m16, r16 指令，由于 COM 程序是在 DOS 平台下定义的 16 位实模式程序，最重要的是这条指令没有操作数大小前缀。因此它特指后者，表示运行在实模式下的指令。但是 x86 架构的 CPU 是向下兼容的，所有实模式指令也可以在保护模式下运行。能紧接着的 ModR/M 这个字节 DB，二进制值为 11011011，可以得到各个区域的取值与寻址模式表的映射关系，其中 Mod=11，表示了操作数为寄存器寻址，Reg/Opcode=011 表示 BX 寄存器所在的列，R/M=011 表示 BX 所在的行，因此最终得到汇编指令为 xor bx, bx。

跟着的一个字节 8E 为一条 MOV Sreg,r/m16 指令，由于可知接下来一字节 D3 为 ModR/M 数据，分区域得到 Mod=11，R/M=011，可以定位到寄存器操作数 bx，Reg/Opcode=010 表示了段寄存器 SS。因此这条占两个字节的指令为 mov ss, bx。

再跟着 的 BC 这里比较麻烦，事实上它是一条 mov 指令，但是不能在手册上直接查找到。回到前面讲过的内容，指令码可以像 B8+cw 这样的表达，正是因为指令码中增长了操作数信息，而使用得指令基本的编码发生了改变，因此手册上不能直接查找到，只能通过对指令集的整理来实现。因为是 16-bit 数据模式，这里 BC-B8=4 可以推导出这个隐含的寄存器为 SP，那么跟着指令码的两个字节就是 16-bit 的立即数。因此最终的汇编指令为 mov sp, 580h。

然后一个字节是 E8，即 CALL rel16 指令，由此可知整个指令占 3 个字节，后面的两个字节为跳转偏移值，最终构造得到汇编指令 call $+3。这里的 CALL 指令只是将程序的一条指令的地址入栈，程序还是继续执行下一条指令。

经过上一条假 CALL 跳转后，程序执行到 5B 这里，这条指令也不能在手册中直接搜索到，但可以变通地搜索 58 这条指令，即 58 + rw POP r16。从 5B-58=3 这时可以得隐含的寄存器为 BX，即汇编指令为 pop bx。

然后是 81，这个字节十分具有迷惑性，通过搜索手册，竟有 16 条相似的指令：

```
81 /6 iw  XOR r/m16,imm16     81 /7 iw  CMP r/m16,imm16
81 /6 id  XOR r/m32,imm32     81 /7 id  CMP r/m32,imm32
81 /2 iw  ADC r/m16,imm16     81 /1 iw  OR r/m16,imm16
81 /2 id  ADC r/m32,imm32     81 /1 id  OR r/m32,imm32
81 /0 iw  ADD r/m16,imm16     81 /3 iw  SBB r/m16,imm16
81 /0 id  ADD r/m32,imm32     81 /3 id  SBB r/m32,imm32
81 /4 iw  AND r/m16,imm16     81 /5 iw  SUB r/m16,imm16
81 /4 id  AND r/m32,imm32     81 /5 id  SUB r/m32,imm32
```

那 么会是那一条呢？当然 CPU 是肯定知道了。首先，通过 16-bit 操作数就可以排除掉了一半，有 32 字样的指令都不在目标内。然而余下的 8 条指令形式除了在指令码的额外 3-bit 数值上不同外，其它内容形式都是相同的，对我来说，这更像个奇观！通过分析接下来的 ModR/M 这个字节 EB，Mod=11，Reg/Opcode=101，R/M=011，这下就清晰了，满足 Reg/Opcode=101 即 /5 的指令只有一条，那就是... SUB！而且，目标寄存器操作数为 BX，源操作数 imm16 为后而紧跟的两个字节 0x006B。最终得到的汇编指令为 sub bx, 6bh。

至于 AutoNeoGrub.mbr 引导程序余下的代码还有很多，不一而足。读者可以自行应用前面的理论进行手工反汇编工作。到此代码片断的汇编形式就可以按以下 COM 程序的形式来组织：

```
cs:0100 ; +------------------------------------------------------------------------+
```

```
cs:0100 ; File Name   : AutoNeoGrubA.com
cs:0100 ; Format      : MS-DOS COM-file
cs:0100 ; Base Address: 1000h
cs:0100 ; +-------------------------------------------------------------------------+
cs:0100
cs:0100                 .686p
cs:0100                 .mmx
cs:0100                 .model tiny
cs:0100
cs:0100 ; =========================================================================
cs:0100
cs:0100 seg000          segment byte public 'CODE' use16
cs:0100                 assume cs:seg000
cs:0100                 org 100h
cs:0100                 assume es:nothing, ss:nothing, ds:seg000
cs:0100
cs:0100                 jmp     short start
cs:0100 ; -------------------------------------------------------------------------
cs:0102                 db 80h, 0, 20h, 39h, 2 dup(0FFh), 58h dup(0)
cs:0160 ; -------------------------------------------------------------------------
cs:0160
cs:0160 start:
cs:0160                 cli
cs:0161                 xor     bx, bx
cs:0163                 mov     ss, bx
cs:0165                 mov     sp, 580h
cs:0168                 call    $+3
cs:016B                 pop     bx
cs:016C                 sub     bx, 6Bh
```

总结

汇 编，它是个底层基础，是高级语言的基本结构。它们不同点在于，汇编更贴近了 CPU 的硬件运行机理。而高级语言则更注重在算法、程序便利性和性能上的设计。如基本的流程语句的设计，它需要怎么走，如何能让开发人员更有效率地编写代码，如 何能以更快的速度得到最终的程序，如何能让程序的运行效率更接近汇编的层次，等等都是高语言关注的核心问题。

几天前，在构思这篇文章时，觉 得会特别耗时。那里，刚完成了《深入x86的内在寻址》《深入BIOS与中断》等文章的写作，消耗了不少时间与精力。但这篇文章所涉及的内容之重要，又使 我不得不紧接着前两篇的脚步走下去。写到这时，汇

编与反汇编的知识点也基本都讲透了，又觉得是不是还要添加一些内容进来！这几篇文章所涉及的内容都是按进 阶来进行的，同时又起到相互说明的作用，因此我觉得，如果时机成熟，可以将这系列文章组织好再发表成册，这样就满足我小小的虚荣吧。

参考

1. IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
3. Intel 80386 Reference Programmer's Manual http://download.csdn.net/detail/winsenjiansbomber/7281997
4. The art of disassembly http://bbs.pediy.com/showthread.php?t=75094
5. X86 Opcode and Instruction Reference http://ref.x86asm.net/index.html
6. CALL指令有多少种写法（有些 bit 误写成字节） http://blog.ftofficer.com/2010/04/n-forms-of-call-instructions/
7. x86 Disassembler Librarys http://bastard.sourceforge.net/libdisasm.html
8. The Complete Pentium Instruction Set Table (32 Bit Addressing Mode Only) http://bbs.pediy.com/showthread.php?t=54706

- 建档时间：Tue, 27 May 2014 16:13:30 GMT

顶 2　　踩 0

上一篇 重大发现：MSDN for DOS - Microsoft Library 1.03

下一篇 Shopex模板机制总览（摘要版）

猜你在找

查看评论

3楼 A3630623 2014-05-28 12:46发表

很不错的反汇编科普，再结合一款反汇编器源码连载剖析就非常完美了

Re: Jimbo 2014-05-28 20:52发表

回复A3630623:有想过做x86 或 51 之类的汇编工具，再有机会和时间就水到渠成了。

2楼 岁月小龙 2014-05-28 08:55发表

真心看不懂 啊

Re: Jimbo 2014-05-28 20:53发表

回复岁月小龙:唔，不巧被我言中！如果阅读时发现无法理解的内容就表明需要补充基础知识，此时请停下来，或者跳过部分内容也是不错的阅读方法。

1楼 epluguo 2014-05-28 08:55发表

Good job,很早之前了解到过个大概，不过很难在网上找到相关资料

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap