内联汇编 2007-12-06 14:32:27

分类： LINUX

AT&T 的汇编格式

一　基本语法

语法上主要有以下几个不同.

★ 寄存器命名原则

AT&T: %eax Intel: eax

★源/目的操作数顺序

AT&T: movl %eax,%ebx Intel: mov ebx,eax

★常数/立即数的格式

AT&T: movl $_value,%ebx Intel: mov eax,_value

把_value 的地址放入 eax 寄存器

AT&T: movl $0xd00d,%ebx Intel: mov ebx,0xd00d

★ 操作数长度标识

AT&T: movw %ax,%bx Intel: mov bx,ax

★寻址方式

AT&T: immed32(basepointer,indexpointer,indexscale)

Intel: [basepointer + indexpointer*indexscale + imm32)

Linux 工作于保护模式下，用的是 3 2 位线性地址，所以在计算地址时不用考虑 segment:offset

的问题．上式中的地址应为：

imm32 + basepointer + indexpointer*indexscale

下面是一些例子：

★直接寻址

AT&T: _booga　; _booga 是一个全局的 C 变量

注意加上$是表示地址引用，不加是表示值引用．

注：对于局部变量，可以通过堆栈指针引用．

Intel: [_booga]

★寄存器间接寻址

AT&T: (%eax)

Intel: [eax]

★变址寻址

AT&T: _variable(%eax)

Intel: [eax + _variable]

AT&T: _array(,%eax,4)

Intel: [eax*4 + _array]

AT&T: _array(%ebx,%eax,8)

Intel: [ebx + eax*8 + _array]

二　基本的行内汇编

·基本的行内汇编很简单，一般是按照下面的格式:

asm("statements");

例如：asm("nop"); asm("cli");

·asm　和　__asm__是完全一样的．

·如果有多行汇编，则每一行都要加上　"\n\t"

例如：

```
asm( "pushl %eax\n\t"
"movl $0,%eax\n\t"
"popl %eax");
```

实际上 gcc 在处理汇编时，是要把 asm(...)的内容"打印"到汇编文件中，所以格式控制字符是必要的．

再例如：

```
asm("movl %eax,%ebx");
asm("xorl %ebx,%edx");
asm("movl $0,_booga);
```

关于修饰（被改变的）寄存器列表的理解

在上面的例子中，由于我们在行内汇编中改变了 edx 和 ebx 的值，但是由于 gcc 的特殊的处理方法，即先形成汇编文件，再交给 GAS 去汇编，所以 GAS 并 不知道我们已经改变了 edx 和 ebx 的值，如果程序的上下文需要 edx 或 ebx 作暂存，这样就会引起严重的后果．对于变量 _booga 也存在一样的问题． 为了解决这个问题，就要用到扩展的行内汇编语法．

三　扩展的行内汇编

扩展的行内汇编类似于 Watcom.

基本的格式是：

```
asm ( "statements" : output_regs : input_regs : clobbered_regs);
```

clobbered_regs 指的是被改变的寄存器．

下面是一个例子(为方便起见，我使用全局变量）：

```
int count=1;
int value=1;
int buf[10];
void main()
{
asm(
"cld \n\t"
"rep \n\t"
"stosl"
:
: "c" (count), "a" (value) , "D" (buf[0])
: "%ecx","%edi" );
}
```

得到的主要汇编代码为：

```
movl count,%ecx
movl value,%eax
movl buf,%edi
#APP
cld
rep
stosl
#NO_APP
```

cld,rep,stos 就不用多解释了．这几条语句的功能是向 buf 中写上 count 个 value 值．冒号后

的语句指明输入，输出和被改变的寄存器．通 过冒号以后的语句，编译器就知道你的指令需要和改变哪些寄存器，从而可以优化寄存器的分配．

其中符号"c"(count)指示要把 count 的值放入 ecx 寄存器

类似的还有：

a eax

b ebx

c ecx

d edx

S esi

D edi

I 常数值，(0 - 31)

q,r 动态分配的寄存器

g eax,ebx,ecx,edx 或内存变量

A 把 eax 和 edx 合成一个 64 位的寄存器(use long longs)

我们也可以让 gcc 自己选择合适的寄存器．

如下面的例子：

```
asm("leal (%1,%1,4),%0"
: "=r" (x)
: "0" (x) );
```

这段代码实现 5*x 的快速乘法．

得到的主要汇编代码为：

```
movl x,%eax
#APP
leal (%eax,%eax,4),%eax
```

#NO_APP

movl %eax,x

几点说明：

1.使用 q 指示编译器从 eax,ebx,ecx,edx 分配寄存器．使用 r 指示编译器从 eax,ebx,ecx,edx,esi,edi 分配寄存器．

使用自动自动分配方式（“ r ”\“q”）分配的寄存器

2.我们不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器已经记住了它们．

3."="是标示输出寄存器，必须这样用．

4.数字%n 的用法：

数字表示的寄存器是按照出现和从左到右的顺序映射到用"r"或"q"请求的寄存器．如果我们要重用"r"或"q"请求的寄存器的话，就可以使用它们．

5.如果强制使用固定的寄存器的话，如不用%1,而用 ebx,则 asm("leal (%%ebx,%%ebx,4),%0"

: "=r" (x)

: "0" (x));

注意要使用两个%,因为一个%的语法已经被%n 用掉了．

下面可以来解释 letter 4854-4855 的问题：

1、变量加下划线和双下划线有什么特殊含义吗？

加下划线是指全局变量，但我的 gcc 中加不加都无所谓．

2、以上定义用如下调用时展开会是什么意思？

#define _syscall1(type,name,type1,arg1) \

type name(type1 arg1) \

```
{ \
long __res; \
/* __res 应该是一个全局变量    */
__asm__ volatile ("int $0x80" \
/* volatile 的意思是不允许优化，使编译器严格按照你的汇编代码汇编*/
: "=a" (__res) \
/* 产生代码    movl %eax, __res */
: "0" (__NR_##name),"b" ((long)(arg1)); \
/* 如果我没记错的话，这里##指的是两次宏展开．
        即用实际的系统调用名字代替"name",然后再把__NR_...展开．
        接着把展开的常数放入 eax，把 arg1 放入 ebx */
if (__res >= 0) \
return (type) __res; \
errno = -__res; \
return -1; \
}
```