第二章．AT&T 汇编语言与 GCC 内嵌汇编 (v0.1)

说到 AT&T 汇编语言，我们就需要从 Unix 开始说起。Unix 最初是为 PDP－11 开发的，曾先后被移植到 VAX 及 68000 系列的处理器上，这些处理器上的汇编语言都采用的是 AT&T 的指令格式（那个时候，Intel 公司还未成立，而 AT&T 公司尚未被拆分）。作为 Unix 家族的一员，我们今天使用的 Linux 在设计时也是采用的 AT&T 格式的汇编语言。Intel 公司在成立并推出了面向个人用户的微处理器后，也同时推出了在个人微处理器上运行的汇编语言，即 Intel 汇编语言，以及相应的语法。我们以前学习的 8086 宏汇编语言，由于是基于 Intel 公司的处理器，所以自然是 Intel 汇编语言。

由于在本书其后所提及的一系列实验中，我们设计的是一个类 Unix 的操作系统，开发环境也是在 Linux 系统中运行的一些列 GNU 的开发工具(如 GCC 和 GAS)，所以无论从习惯还是从程序设计的角度，都必须选择 AT&T 汇编语言。

这里，我们对 AT&T 汇编语言以及在 GCC 中内嵌这种类型的汇编语言的语法进行讲解，以期帮助熟悉 Intel 汇编语言的读者在最短时间内掌握这种类型的汇编语言，并将所学知识应用到以后的一系列实验中。需要指出的是，AT&T 汇编语言并不是一门全新的语言，只是在格式上与 Intel 汇编语言有一些差异，对于熟悉 Intel 汇编语言的读者来说，在了解了这些差异后是非常容易掌握的。同时，了解和熟悉 AT&T 汇编语言对于阅读和理解 Linux 内核，以及内核级的一些程序模块时，将带来莫大的帮助。所以，对于 Linux 系统级程序员来说，这几乎是必不可少的知识积累。

2.1 AT&T 汇编语言的相关知识

在 Linux 源代码中，以.S(或.s)为扩展名的文件是包含汇编语言代码的文件。这里，我们结合具体的例子再介绍一些 AT&T 汇编语言的相关知识。

1．GNU 汇编程序 GAS(GNU Assembly)和连接程序

我们编写了一个程序后，就需要对其进行汇编和连接。在 Linux 下有两种方式，一种是使用汇编程序 GAS 和连接程序 LD，一种是使用 GCC，我们先来看一下 GAS 和 LD 的使用。

GAS 把汇编语言源文件(.S 或.s)转换为目标文件(.o)，其基本语法如下：

```
as sourcecode.s -o objfile.o
```

一旦创建了一个目标文件，就需要把它连接并执行，连接一个目标文件的基本语法为：

```
ld objfile.o -o execode
```

这里 objfile.o 是目标文件名，而 execode 是输出(可执行) 文件。

如果要使用 GNC 的 C 编译器 gcc，就可以一步完成汇编和连接，例如：

```
gcc -o execode sourcecode.S
```

这里，sourcecode.S 是你的汇编程序，输出文件(可执行文件)名为 execode。其中，扩展名必须为大写的 S，这是因为，大写的 S 可以使 gcc 自动识别汇编程序中的 C 预处理命令，像 #include、#define、#ifdef、 #endif 等，也就是说，使用 gcc 进行编译，你可以在汇编程序中使用 C 的预处理命令。下面给出一个源程序的例子。

```
.data
output: .ascii "hello world\n"
```

```
.text
.globl  _start
_start:
        movl $4, %eax
        movl $1, %ebx
        movl $output, %ecx
        movl $12, %edx
        int $0x80
        movl $1, %eax
        int $0x80
```

例 1-1

这个程序的结果是在屏幕上打印“hello world”。

2. AT&T 中的节(Section)

在 AT&T 的语法中，一个节由.section 关键词来标识，当你编写汇编语言程序时，至少需要有以下三种节：

（1）.data

这种节包含程序已初始化的数据，也就是说，包含具有初值的那些变量，例如：

```
hello : .string "Hello world!\n"
hello_len : .long 16
```

(2) .bss

这个节包含程序还未初始化的数据，也就是说，包含没有初值的那些变量。当操作系统装入这个程序时将把这些变量都置为 0，例如：

```
name : .fill 30 # 用来请求用户输入名字
name_len : .long 0 # 名字的长度(尚未定义)
```

当这个程序被装入时，name 和 name_len 都被置为 0。如果你在.bss 节不小心给一个变量赋了初值，这个值也会丢失，并且变量的值仍为 0。

使用.bss 比使用.data 的优势在于，.bss 节在编译后不占用磁盘的空间，这样编译、连接生成的代码的尺寸会比较小。例如，在磁盘上，通常一个长整数(4 个字节)所占用的空间，就足以存放一个.bss 节。

需要注意的是，编译程序往往把.data 和.bss 放在 4 字节上对齐，因此，这两个节的起始地址会是 4 的倍数。同时，4 字节对齐的要求也会导致.data 节所占用的空间往往会大于它实际所需的空间，且大小为 4 的倍数。例如，假设.data 总共有 30 字节，在生成代码时，由于它和它之后的.bss 节都需要 4 字节对齐，于是.data 节之后的两个字节都不会被用到，也就是说实际给了.data 节 32 字节的空间。

（3）.text

这个节包含程序的代码。需要指出的是，该节是只读节，而.data 和.bss 是可读写的节。下面给出一个例子。

```
.data
output: .ascii "hello world\n"
.text
.globl  _start
_start:
```

```
        movl $4, %eax
        movl $1, %ebx
        movl $output, %ecx
        movl $12, %edx
        int $0x80
        movl $3, %eax
        movl $1, %ebx
        movl $sentence, %ecx
        movl $30, %edx
        int $0x80
        movl $4, %eax
        movl $30, %edx
        int $0x80
        movl $1, %eax
        int $0x80
.bss
    sentence: .fill 30
```

例 1-2

例 2-2 的源程序中存在.data、.text、.bss 三个节。.bss 节中的变量会被初始化为 0，在程序的运行中会先打印"hello world"然后再打印 sentence 中后来所输入的字符串。

3．汇编程序指令(Assembler Directive)

上面介绍的.section 就是汇编程序指令的一种，GNU 汇编程序提供了很多这样的指令(directive)，这种指令都是以句点“.”为开头，后跟指令名(小写字母)，在此，我们只介绍在内核源代码中出现的几个指令。

- .ascii

语法：.ascii "string"...

.ascii 表示零个或多个(用逗号隔开)字符串，并把每个字符串(结尾不自动加“\0”字符)中的字符放在连续的地址单元。例如例 2-2 中的：

```
output: .ascii "hello world\n"
```

在这里，字符串 output 不会被自动添零，于是在之后的输出过程中程序需要通过 edx 寄存器来告知系统输出字符串的长度。

还有一个与.ascii 类似的.asciz，z 代表“\0”，即每个字符串结尾自动加一个“\0”字符，例如定义字符串 err_int_msg：

```
    err_int_msg:   .asciz      "Unknown interrupt\n"
```

- .fill

语法：.fill repeat , size , value

其中，repeat，size 和 value 都是常量表达式。.fill 的含义是反复拷贝 size 个字节，重复 repeat 次。repeat 可以大于或者等于 0。size 也可以大于等于 0，但不能超过 8，如果超过 8，也只取 8。size 个字节的内容将被填充为 value 的值，如果 size 的大小大于 value 的存储所需要的容量，则将高位用 0 来填充。例如，size 为 8，则最高 4 个字节内容为 0，最低 4 字节内容置为 value。

size 和 value 为可选项。如果第二个逗号和 value 值不存在，则假定 value 为 0。如果第一个逗号和 size 不存在，则假定 size 为 1。例如：

.fill 30, 8 , 0　　即表示反复 30 次，每次向 8 个字节中拷贝 0 值。

● .globl

语法：.globl symbol

.globl 使得连接程序(ld)能够看到 symbol。如果你的局部程序中定义了 symbol，那么，与这个局部程序连接的其他局部程序也能存取 symbol，例如：

某个.S 文件的源程序中某一段为如下

……..

.data

.globl　　number

.set number 10

……..

而与该文件属于同一个文件夹的另一.S 文件的某一段代码为

……..

.text

movl　$number　%eax

……..

可以看到在前一个文件中定义的 globl 变量 number 在后一个文件中可以被引用。

● . rept　.endr

语法：. rept count

…….

.endr

把.rept 指令与.endr 指令之间的行重复 count 次，例如：

.rept 3

.long 0

.endr

相当于：

.long 0

.long 0

.long 0

在这里，.rept 指令比较容易和.fill 指令混淆，它们的区别是.rept 是将.rept 与.endr 之间的指令重复 3 次，而.fill 则是单纯的重复填充数据。

● . space

语法：. space size , fill

这个指令保留 size 个字节的空间，每个字节的值为 fill。size 和 fill 都是常量表达式。如果逗号和 fill 被省略，则假定 fill 为 0。例如：

Label: .space 10, 1

● .byte

语法：.byte expressions

预留 1 个字节，并将这个字节的内容赋值为 expression，如果是用逗号隔开的多个 expression，则为预留多个这样的字节，并将它们的内容依次赋值。例如：

Mark: .byte 100

● .word

语法：.word expressions

预留 2 个字节，并将该 2 个字节的内容赋值为 expression，如果是用逗号隔开的多个

expression，则为预留多个这样的 2 字节，并将它们的内容依次赋值。例如：

num: .word 0x100

- .long

  这与.word 类似，只是这里是给双字(4 个字节)赋值。例如：

  number: .long 0x100

- .set

  设定常数，就好像 C 程序中的#define 的作用一样。例如：

  .set mark, 0x10

  这样在接下来的程序中就可以用诸如 movl $mark, eax 这样的指令来引用 mark。

  下面给出一个源程序的示例可以让读者可以对这些指令有更为深刻的印象。

```
.data
.globl length
.set length, 13
label: .fill 10 , 1 , 65
hello:    .rept 3
.ascii "\nhello"
.endr
number: .byte 100
.word 0x100
.long 0x100
output: .ascii "\nhello world\n"
.text
.globl    _start
_start:
          movl $4, %eax
          movl $1, %ebx
          movl $label, %ecx
          movl $10, %edx
          int $0x80
          movl $4, %eax
          movl $hello, %ecx
          movl $18, %edx
          int $0x80
          movl $number, %edi
          movb (%edi), %al
          movw 0x1(%edi), %ax
          movl 0x3(%edi), %eax
          movl $4, %eax
          movl $output, %ecx
          movl $length, %edx
          int $0x80
          movl $1, %eax
          int $0x80
```

例 1-3

以上这段代码涉及了之前所讲的一些指令，这段程序首先将label所标识的30个字符打印出来，然后再依次将 number 所标识的一个字节、字以及双字复制到 eax 寄存器，最后在屏幕上打印出"hello world"。

2.2 AT&T 与 Intel 的汇编语言语法区别

AT&T 和 Intel 汇编语言的语法区别主要体现在操作数前缀、赋值方向、间接寻址语法、操作码的后缀上，而就具体的指令而言，在同一平台（如本书所涉及的 IA32 平台）上，两种汇编语言是一致的。下面仅列出这两种汇编语言在语法上的不同点。

1. 操作数前缀

在 Intel 的汇编语言语法中，寄存器和和立即数都没有前缀。但是在 AT&T 中，寄存器前冠以“%”，而立即数前冠以“$”。在 Intel 的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在 AT&T 中，十六进制立即数前冠以“0x”，表 2-1 给出几个相应的例子。

表 2-1 Intel 与 AT&T 汇编语言的指令操作数前缀

Intel 语法	AT&T 语法
Mov eax,8	movl $8,%eax
Mov ebx,0ffffh	movl $0xffff,%ebx
int 80h	int $0x80

从表中可以看到在 AT%T 汇编中诸如"%eax"、"%ebx"之类的寄存器名字前都要加上"%"；"$8"、"$0xffff"这样的立即数之前都要加上"$"。

2. 源/目的操作数顺序

细心的读者可能已经发现，在表 2-2 所列出的例子中，Intel 汇编语言的指令与 AT&T 的指令操作数的方向上正好相反：在 Intel 语法中，第一个操作数是目的操作数，第二个操作数源操作数。而在 AT&T 中，第一个数是源操作数，第二个数是目的操作数。

AT&T 汇编语言这样设计，主要是为了接近人们通常的阅读习惯。例如：

表 2-2 Intel 与 AT&T 汇编语言的指令操作数赋值方向

Intel 语法	AT&T 语法
MOV EAX,8	movl $8,%eax

上表中所举的将立即数复制到寄存器 eax 的例子中，我们可以看到 Intel 语法是规定 eax 在前，立即数在后，而 AT&T 则是 eax 在后。

3. 寻址方式

与 Intel 的语法比较，AT&T 间接寻址方式可能更晦涩难懂一些。Intel 的指令格式是 segreg：[base+index*scale+disp]，而 AT&T 的格式是%segreg：disp(base,index,scale)。其中 index/scale/disp/segreg 全部是可选的，完全可以简化掉。如果没有指定 scale 而指定了 index，则 scale 的缺省值为 1。segreg 段寄存器依赖于指令以及应用程序是运行在实模式还是保护模式下，在实模式下，它依赖于指令，而在保护模式下，segreg 是多余的。在 AT&T 中，当立即数用在 scale/disp 中时，不应当在其前冠以“$”前缀，而且 scale,disp 不需要加前缀“&”。另外在 Intel 中基地址使用“[”、“]”，而在 AT&T 中则使用“(”、“)”。

表 2-3Intel 与 AT&T 汇编语言的寻址指令格式

Intel 语法	AT&T 语法
Instr foo,segreg：[base+index*scale+disp]	instr %segreg：disp(base,index,scale),foo

下面是 Intel 和 AT&T 汇编寻址的一些具体的例子：

表 2-4Intel 与 AT&T 汇编语言的寻址指令示例

Intel 语法	AT&T 语法
[eax]	(%eax)
[eax + _variable]	_variable(%eax)
[eax*4 + _array]	_array(,%eax,4)
[ebx + eax*8 + _array]	_array(%ebx,%eax,8)

4. 标识长度的操作码前缀和后缀

在 AT&T 汇编中远程跳转指令和子过程调用指令的操作码使用前缀“l”，分别为 ljmp，lcall，与之相应的返回指令伪 lret。例如：

表 2-5 Intel 与 AT&T 汇编语言的操作码前缀区别

Intel 语法	AT&T 语法
CALL SECTION:OFFSET	lcall $secion:$offset
JMP FAR SECTION:OFFSET	ljmp $secion:$offset
RET FAR STACK_ADJUST	lret $stack_adjust

在 AT&T 的操作码后面有时还会有一个后缀，其含义就是指出操作码的大小。“l”表示长整数（32 位），“w”表示字（16 位），“b”表示字节（8 位）。而在 Intel 的语法中，则要在内存单元操作数的前面加上 byte ptr、 word ptr,和 dword ptr，“dword”对应“long”。表 2-6 给出几个相应的例子。

表 2-6 Intel 与 AT&T 汇编语言的操作码后缀区别

Intel 语法	AT&T 语法
Mov al,bl	movb %bl,%al
Mov ax,bx	movw %bx,%ax
Mov eax,ebx	movl %ebx,%eax
Mov eax, dword ptr [ebx]	movl (%ebx),%eax

下面给出一个 32 位下 Intel 与 AT&T 汇编对比的源程序示例：

首先是 Intel 的汇编

```
.DATA
NUM    DB     100
DW    100h
DD    100h
OUTPUT    DB    "hello world\n"
;
.CODE
BEGIN：MOV    AX, @DATA
          MOV      DS, AX
           MOV EDI, OFFSET NUM
          MOV AL, BYTE PTR [EDI]
           MOV AX, WORD PTR [EDI]
           MOV EAX, DWORD PTR [EDI]
           MOV ECX, OFFSET OUTPUT
           MOV EDX, 12
          MOV EAX, 4
```

```
            MOV EBX, 1
            INT    80H
             MOV EAX, 1
             INT    80H
END         BEGIN
```

然后是 AT&T 的汇编

```
.data
number: .byte 100
.word 0x100
.long 0x100
output: .ascii "hello world\n"
.text
.globl    _start
_start:
             movl $number, %edi
             movb (%edi), %al
             movw 0x1(%edi), %ax
             movl 0x3(%edi), %eax
             movl $output, %ecx
             movl $12, %edx
             movl $4, %eax
             movl $1, %ebx
             int $0x80
             movl $1, %eax
             int $0x80
```

例 1-4

上面程序的功能是将先分别将一个字节、字、双字复制到 al、ax、eax 中，最后打印出"hello world"，在这里，我们依次给出了 Intel 汇编和 AT&T 汇编的程序，读者可以对比以下两者的不同之处。

读者应该还记得我们在 1.1 节中讲述了一个分别在实模式与保护模式下在屏幕上打印字符串的 Intel 汇编语言的例子，现在我们会将这个例子改成用 AT&T 汇编写得程序：

```
   .set PROT_MODE_CSEG, 0x8              # 代码段选择子
   .set PROT_MODE_DSEG, 0x10             # 数据段选择子 r
   .set CR0_PE_ON,          0x1          # 保护模式启动标识

.globl start
start:
.code16                                  # 16 位模式
   cli                                   # 关中断
   cld                                   # 关字符串操作自动增加

# 设置重要数据段寄存器(DS, ES, SS).
```

```
    xorw    %ax,%ax                 # 将 ax 清零
    movw    %ax,%ds                 # 初始化数据段寄存器
    movw    %ax,%es                 # 初始化附加段寄存器
    movw    %ax,%ss                 # 初始化堆栈段寄存器

# 在实模式下通过向显存中写字节流在屏幕上打印"hello world"
    movw    $0xb800,%ax
    movw    %ax,%es
    movw    $msg1,%si
    movw    $0xc82,%di
    movw    $8,%cx
    rep     movsb

    movw    $str,%si
    movw    $0xc94,%di
    movw    $26,%cx
    rep     movsb

# 打开 A20
seta20.1:
    inb     $0x64,%al               # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.1

    movb    $0xd1,%al               # 0xd1 -> port 0x64
    outb    %al,$0x64

seta20.2:
    inb     $0x64,%al               # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.2

    movb    $0xdf,%al               # 0xdf -> port 0x60
    outb    %al,$0x60

# 从实模式切换到保护模式
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl     $CR0_PE_ON, %eax
    movl    %eax, %cr0
# 通过长跳转使得程序在切换到保护模式的同时切换到 protcseg 处执行
    ljmp    $PROT_MODE_CSEG, $protcseg

.code32                             # 32 模式
```

```
protcseg:
   # 在保护模式下设置数据段寄存器
   movw      $PROT_MODE_DSEG, %ax
   movw      %ax, %ds
   movw      %ax, %es
   movw      %ax, %fs
   movw      %ax, %gs
   movw      %ax, %ss

# 在保护模式下打印"hello world"
   movl      $msg2,%esi
   movl      $0xb8d22,%edi
   movl      $44,%ecx
   rep       movsb

spin:
   jmp spin

# GDT 表
.p2align 2                                  # GDT 表 4 字节对齐
gdt:
   SEG_NULL                                 # 空表项
   SEG(STA_X|STA_R, 0x0, 0xffffffff)        # 代码段表项
   SEG(STA_W, 0x0, 0xffffffff)              # 数据段表项

gdtdesc:
   .word     0x17                            # gdt 表长度 - 1
   .long     gdt                             # gdt 表物理地址

#字符串
msg1:
   .byte 'r',0xc,'e',0xc,'a',0xc,'l',0xc
msg2:
   .byte 'p',0xc,'r',0xc,'o',0xc,'t',0xc,'e',0xc,'c',0xc,'t',0xc,'e',0xc,'d',0xc
str:
   .byte ':',0xc,' ',0xc,'h',0xc,'e',0xc,'l',0xc,'l',0xc,'o',0xc,' ',0xc,'w',0xc,'o',0xc,'r',0xc,'l',0xc,'d',0xc
```

例 1-5

在例 2-5 中，可以看到我们分别在实模式下和保护模式下往显存的 0xb8c82 和 0xb8d22 处写字节流。其中保护模式下是段寄存器 es 的值为 0xb800，di 寄存器为 0xc82，这样移位加后得到 0xb8c82 物理地址，而在保护模式下则是段选择子 es 对应的基地址为 0，而 edi 寄存器中的偏移地址为 0xb8d22，这样加起来后便得到需要寻找的物理地址。

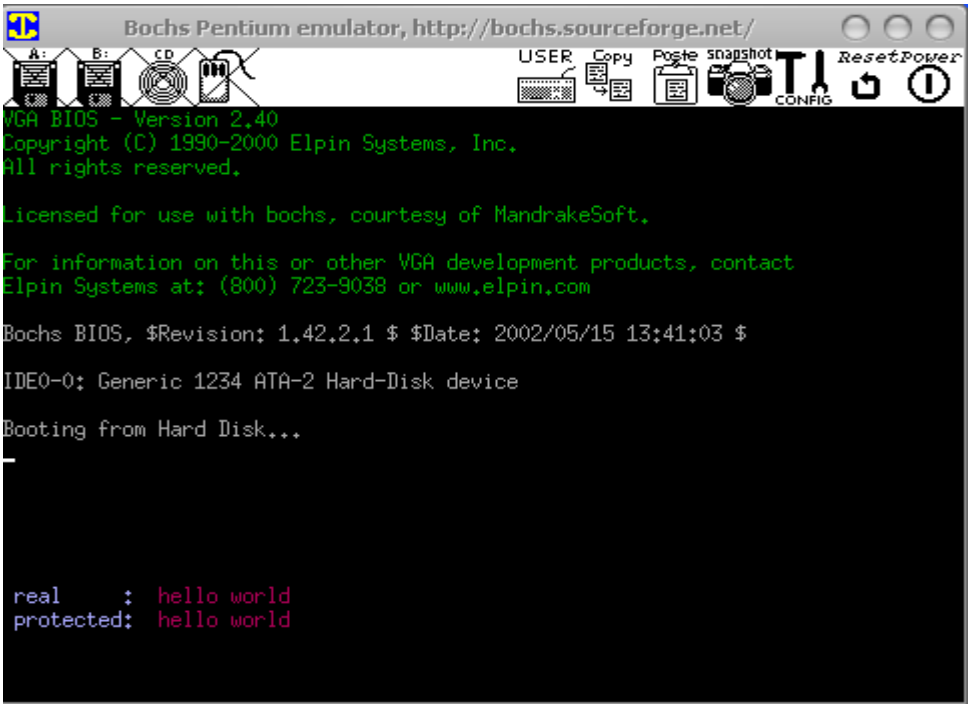

图 1-1 示例程序运行结果

在上图中我们看到在实模式和保护模式下我们都打印出了"hello world"。

2.3 GCC 内嵌汇编

Linux 操作系统内核代码绝大部分使用 C 语言编写，只有一小部分使用汇编语言编写，例如与特定体系结构相关的代码和对性能影响很大的代码。GCC 提供了内嵌汇编的功能，可以在 C 代码中直接内嵌汇编语言语句，大大方便了程序设计。

1. 基本行内汇编

基本行内汇编很容易理解，一般是按照下面的格式：

asm(“statements”);

同时“asm”也可以由“__asm__”来代替，“asm”是“__asm__”的别名。在“asm”后面有时也会加上“__volatile__”表示编译器不要优化代码，后面的指令保留原样，“volatile”是它的别名，在这里值得注意的是无论“__asm__”还是“__volatile__”中的每个下划线都不是一个单独的下划线，而是两个短的下划线拼成的。再后面括号里面的便是汇编指令。

例如：

```
__asm__   __volatile__("hlt");
```

如果有很多行汇编，则每一行后要加上“\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);
```

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.

2. 扩展的行内汇编

在扩展的行内汇编中，可以将 C 语言表达式指定为汇编指令的操作数，而且不用去管如何将 C 语言表达式的值读入寄存器，以及如何将计算结果写回 C 变量，你只要告诉程序中 C 语言表达式与汇编指令操作数之间的对应关系即可，GCC 会自动插入代码完成必要的操作。

使用内嵌汇编，要先编写汇编指令模板，然后将 C 语言表达式与指令的操作数相关联，并告诉 GCC 对这些操作有哪些限制条件。例如在下面的汇编语句：

__asm__ __volatile__ ("movl %1,%0" : "=r" (result) : "r" (input));

“movl %1,%0”是指令模板；“%0”和“%1”代表指令的操作数，称为占位符，内嵌汇编靠它们将 C 语言表达式与指令操作数相对应。指令模板后面用小括号括起来的是 C 语言表达式，本例中只有两个：“result”和“input”，他们按照出现的顺序分别与指令操作数“%0”,“%1，”对应；注意对应顺序：第一个 C 表达式对应“%0”；第二个表达式对应“%1”，依次类推，操作数至多有 10 个，分别用“%0”，“%1”…．“%9，”表示。

在每个操作数前面有一个用引号括起来的字符串，字符串的内容是对该操作数的限制或者说要求。“result”前面的限制字符串是“=r”，其中“=”表示“result”是输出操作数，“r”表示需要将“result”与某个通用寄存器相关联，先将操作数的值读入寄存器，然后在指令中使用相应寄存器，而不是“result”本身，当然指令执行完后需要将寄存器中的值存入变量“result”，从表面上看好像是指令直接对“result”进行操作，实际上 GCC 做了隐式处理，这样我们可以少写一些指令。“input”前面的“r”表示该表达式需要先放入某个寄存器，然后在指令中使用该寄存器参加运算。

限制字符必须与指令对操作数的要求相匹配，否则产生的汇编代码将会有错，读者可以将上例中的两个“r”，都改为“m”(m，表示操作数放在内存，而不是寄存器中)，编译后得到的结果是：

movl input, result

很明显这是一条非法指令，因此限制字符串必须与指令对操作数的要求匹配。例如指令 movl 允许寄存器到寄存器，立即数到寄存器等，但是不允许内存到内存的操作，因此两个操作数不能同时使用“m”作为限定字符。

3. 扩展的行内汇编的语法

内嵌汇编语法如下：

__asm__(

汇编语句模板:

输出部分:

输入部分:

破坏描述部分);

即格式为 asm ("statements" : output_regs : input_regs : clobbered_regs);

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”

格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。下面是一个简单的例子

```
int main(void)
{
    int dest;
    int value=1;
    asm(
    "movl   %1, %0"
    : "=a"(dest)
    : "c" (value)
    : "%ebx");
    printf("%d\n", dest);
    return 0;
}
```

例 1-6

在这段内嵌汇编的意思是将 value 变量的值复制到变量 dest 中，并指定在汇编中使用 eax 与 ecx 寄存器，同时在最后标识这两个寄存器的值有被改变。

1) 汇编语句模板

汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n”或“\n\t”分开。指令中的操作数可以使用占位符引用 C 语言变量，操作数占位符最多 10 个，名称如下：%0，%1…，%9。指令中使用占位符表示的操作数，总被视为 long 型（4，个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在%和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h1。

2) 输出部分

输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和 C 语言变量组成。每个输出操作数的限定字符串必须包含“=”表示它是一个输出操作数。例：

```
__asm__   __volatile__ ("pushfl ; popl %0 ; cli":"=g" (x) )
```

在这里“x”便是最终存放输出结果的 C 程序变量，而“=g”则是限定字符串，限定字符串表示了对它之后的变量的限制条件，这样 GCC 就可以根据这些条件决定如何分配寄存器，如何产生必要的代码处理指令，以及如何处理操作数与 C 表达式或 C 变量之间的关系。在下一部分我们将详细介绍各类的限制字符。

3) 输入部分

输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符同样也由限定字符串和 C 语言表达式或者 C 语言变量组成。例：

```
__asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));
```

4) 限定字符

可以看到，限制字符有很多种，有些是与特定体系结构相关。此处仅列出一些常用的限

定字符和 i386 中可能用到的一些常用的限定符。它们的作用是指示编译器如何处理其后的 C 语言变量与指令操作数之间的关系，例如是将变量放在寄存器中还是放在内存中等，下表列出了常用的限定字母。

表 2-7 内嵌汇编常用限定符

	限定符	描述
具体的一个寄存器	“a”	将输入变量放入 eax
	“b”	将输入变量放入 ebx
	“c”	将输入变量放入 ecx
	“d”	将输入变量放入 edx
	“s”	将输入变量放入 esi
	“D”	将输入变量放入 edi
	“q”	将输入变量放入 eax、ebx、ecx、edx 中的一个
	“r”	将输入变量放入通用寄存器，也就是 eax, ebx, ecx, edx, esi, edi 中的一个
	“A”	放入 eax 和 edx，把 eax 和 edx，合成一个 64 位的寄存器 (uselong longs)
内存	“m”	内存变量
	“o”	操作数为内存变量，但是其寻址方式是偏移量类型
	“V”	操作数为内存变量，但寻址方式不是偏移量类型
	“,”	操作数为内存变量，但寻址方式为自动增量
	“p”	操作数是一个合法的内存地址（指针）
寄存器或内存	“g”	将输入变量放入 eax，ebx，ecx ，edx 中的一个或者作为内存变量
	“X”	操作数可以是任何类型
立即数	“I”	0-31 之间的立即数（用于 32 位移位指令）
	“J”	0-63 之间的立即数（用于 64 位移位指令）
	“N”	0-255 ，之间的立即数（用于 out 指令）
	“i”	立即数
	“n”	立即数，有些系统不支持除字以外的立即数，这些系统应该使用“n”
操作数类型	“=”	操作数在指令中是只写的（输出操作数）
	“+”	操作数在指令中是读写类型的（输入输出操作数）
浮点数	“f”	浮点数
	“t”	第一个浮点寄存器
	“u”	第二个浮点寄存器
	“G”	标准的 80387
	%	该操作数可以和下一个操作数交换位置
	#	部分注释，从该字符到其后的逗号之间所有字母被忽略
	*	表示如果选用寄存器，则其后的字母被忽略

另外“0”，“1”，...， “9”表示用它限制的操作数与某个指定的操作数匹配，也即该操作数就是指定的那个操作数，例如用“0”去描述“%1”操作数，那么“%1”引用的其

实就是“%0”操作数，注意作为限定符字母的 0－9 ，与指令中的“％0”－“％9”的区别，前者描述操作数，后者代表操作数。

5) 破坏描述部分

修改描述部分可以防止内嵌汇编在使用某些寄存器时导致错误。修改描述符是由逗号隔开的字符串组成的，每个字符串描述一种情况，一般是寄存器，有时也会有“memory”。例如：“%eax”、“%ebx”、“memory”等。具体的意思就是告诉编译器在编译内嵌汇编的时候不能使用某个寄存器或者不能使用内存的空间。

下面用一些具体的示例来讲述 GCC 如何把内嵌汇编转换成标准的 AT&T 汇编的。

首先看一个简单的例子，这个例子：

```
int main(void)
{
    int result = 2;
    int input = 1;
    __asm__ __volatile__ ("addl %1, %0": "=r"(result): "r"(input));
    printf("%d\n", result);
    return 0;
}
```

这段内嵌汇编原本的目的是输出 1+2=3 的结果，也就是将 input 变量的值与 result 变量的值相加之后再存入result中。可以看到在汇编语句模板中的%1 与%0 分别代表 input 与 result 变量，而“=r”与“r”则表示两个变量在汇编中应该对应两个寄存器，“=”表示 result 是输出变量。然而实际运行后发现结果实际上是 2。这是为什么呢？我们用(objdump -j .text –S 可执行文件名)这样的命令来查看编译生成后的代码发现这段内嵌汇编经 GCC 翻译后所对应的 AT&T 汇编是：

```
movl    $0x2,0xfffffffc(%ebp)
movl    $0x1,0xfffffff8(%ebp)
movl     0xfffffff8(%ebp),%eax
addl     %eax,%eax
movl     %eax,0xfffffffc(%ebp)
```

例 1-7-1

前两句汇编分别是为 result 和 input 变量赋值。input 为输入型变量，而且需要放在寄存器中，GCC 给它分配的寄存器是%eax，在执行 addl 之前%eax 的内容已经是 input 的值。可见对于使用“r”限制的输入型变量或者表达式，在使用之前 GCC 会插入必要的代码将他们的值读到寄存器；“m”型变量则不需要这一步。读入 input 后执行 addl，显然 addl %eax,%eax 的值不对。再往后看：movl %eax,0xfffffffc(%ebp)的作用是将结果存回 result，分配给 result 的寄存器与分配给 input 的一样，都是%eax。

综上可以总结出如下几点：

1. 使用“r”限制的输入变量，GCC 先分配一个寄存器，然后将值读入寄存器，最后用该寄存器替换占位符；
2. 使用“r”限制的输出变量，GCC 会分配一个寄存器，然后用该寄存器替换占位符，但是在使用该寄存器之前并不将变量值先读入寄存器，GCC 认为所有输出变量以前的值都没有用处，不读入寄存器，最后 GCC 插入代码，将寄存器的值写回变量；

因为第二条，上面的内嵌汇编指令不能奏效，因此需要在执行 addl 之前把 result 的值读

入寄存器，也许再将 result 放入输入部分就可以了（因为第一条会保证将 result 先读入寄存器）。修改后的指令如下（为了更容易说明问题将 input 限制符由“r，”改为“m”）：

```
int main(void)
{
    int result    = 2;
    int input    = 1;
    __asm__ __volatile__ ("addl    %2,%0":"=r"(result):"r"(result),"m"(input));
    printf("%d\n", result);
    return 0;
}
```

这段内嵌汇编所对应的 AT&T 汇编如下：

```
movl     $0x2,0xfffffffc(%ebp)
movl     $0x1,0xfffffff8(%ebp)
movl      0xfffffffc(%ebp),%eax
addl      0xfffffff8(%ebp),%eax
movl      %eax,0xfffffffc(%ebp)
```

例 1-7-2

看上去上面的代码可以正常工作，因为我们知道%0 和%1 都和 result 相关，应该使用同一个寄存器，而且事实上在实际结果中 GCC 也确实是使用了同一个寄存器 eax，所以可以得到正确的结果 3。但是为了更保险起见，为了确保%0 与%1 与同一个寄存器关联我们可以使用如下的方法：

```
int main(void)
{
    int result    = 2;
    int input    = 1;
    __asm__ __volatile__ ("addl    %2,%0":"=r"(result):"0"(result),"m"(input));
    printf("%d\n", result);
    return 0;
}
```

它所对应的 AT&T 汇编为：

```
movl     $0x2,0xfffffffc(%ebp)
movl     $0x1,0xfffffff8(%ebp)
movl      0xfffffffc(%ebp),%eax
addl      0xfffffff8(%ebp),%eax
movl      %eax,0xfffffffc(%ebp)
```

例 1-7-3

输入部分中的 result 用匹配限制符“0”限制，表示%1 与％0，代表同一个变量，输入部分说明该变量的输入功能，输出部分说明该变量的输出功能，两者结合表示 result 是读写型。因为%0 和%1，表示同一个 C 变量，所以放在相同的位置，无论是寄存器还是内存。

至此读者应该明白了匹配限制符的意义和用法。在新版本的 GCC 中增加了一个限制字符“+”，它表示操作数是读写型的，GCC 知道应将变量值先读入寄存器，然后计算，最后写回变量，而无需在输入部分再去描述该变量。

```
int main(void)
{
```

```
    int result = 2;
    int input = 1;
    __asm__ __volatile__ ("addl %1, %0": "+r"(result): "r"(input));
    printf("%d\n", result);
    return 0;
}
```

这段内嵌汇编所对应的 AT&T 汇编为：

```
    movl    $0x2,0xfffffffc(%ebp)
    movl    $0x1,0xfffffff8(%ebp)
    mov     0xfffffffc(%ebp),%eax
    mov     0xfffffff8(%ebp),%edx
    add     %edx,%eax
    mov     %eax,0xfffffffc(%ebp)
```

例 1-7-4

通过这段内嵌汇编所对应的 AT&T 汇编我们可以看出系统首先将 result 变量的值读入了 eax 寄存器，并为 input 变量分配了 edx 寄存器，然后将 eax 与 edx 的值相加后将结果写入内存。

接下来的一个示例要较为复杂一些：

```
int main(void)
{
    int count=3;
    int value=1;
    int buf[10];
    asm(
    "cld \n\t"
    "rep \n\t"
    "stosl"
    :
    : "c" (count), "a" (value) , "D" (buf) );
    printf("%d %d %d\n", buf[0],buf[1],buf[2]);
}
```

经 GCC 翻译后所对应的 AT&T 汇编是：

```
    movl    0xfffffff4(%ebp),%ecx
    movl    0xfffffff0(%ebp),%eax
    lea     0xffffffb8(%ebp),%edi
    cld
    repz stos %eax,%es:(%edi)
```

例 1-8

在这里 count、value 和 buf 是三个输入变量，它们都是 C 程序中的变量，“c”、“a”和“D”表示这三个输入值分别被存放入寄存器 ECX、EAX 与 EDI；“cld　rep　stosl”是需要执行的汇编指令；而“%ecx、%edi”表示这两个寄存器在指令中被改变了。这段内嵌汇编要做的就是向 buf 中写 count 个 value 值。

最后我们给出一个比较综合一点的例子：

```
int main(void)
{
    int input, output, temp;

    input = 1;
    __asm__ __volatile__    ("movl $0, %%eax;\n\t"
                "movl %%eax, %1;\n\t"
                "movl %2, %%eax;\n\t"
                "movl %%eax, %0;\n\t"
                : "=m" (output), "=m"(temp)
                : "r" (input)
                :"eax");
    printf("%d %d\n", temp,output);
    return 0;
}
```

这段内嵌汇编经由 GCC 转化成的汇编代码如下：

```
    movl    $0x1,0xfffffffc(%ebp)
    mov     0xfffffffc(%ebp),%edx
    mov     $0x0,%eax
    mov     %eax,0xfffffff4(%ebp)
    mov     %edx,%eax
    mov     %eax,0xfffffff8(%ebp)
```

例 1-9

可以看到，由于 input、output、temp 都是程序局部整型数变量，于是它们实际上是存放在堆栈中的，也就是内存中的某个部分。其中 output 和 temp 是输出变量，而且“=m”表明它们应该在内存中，input 是输入变量，“r”表明它应存放在寄存器中，于是首先把 1 存入 input 变量，然后将变量的值复制给了 edx 寄存器，在这里我们可以看到内嵌汇编中使用了破坏描述符“eax”，这是告诉编译器在程序中 eax 寄存器已被使用，这样编译器为了避免冲突会将输入变量存放在除 eax 以外别的寄存器中，如像我们最后看到的 edx 寄存器。看看内嵌汇编的代码编译生成的 AT&T 汇编，我们可以发现第二句内嵌汇编中的%1 转化成了 0xfffffff4 (%ebp)，对应的就是 temp 变量；第三句中的%2 则对应到了%edx，即 input 变量所存放的寄存器；而%0 就对应到 output 变量所存放的内存位置 0xfffffff8 (%ebp)。