AT&A 汇编

1.Register Reference

引用寄存器要在寄存器号前加百分号%,如“movl %eax, %ebx”。
80386 有如下寄存器：
[1] 8 个 32-bit 寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp；
(8 个 16-bit 寄存器，它们事实上是上面 8 个 32-bit 寄存器的低 16 位：%ax，%bx，%cx，%dx，%di，%si，%bp，%sp；
8 个 8-bit 寄存器：%ah，%al，%bh，%bl，%ch，%cl，%dh，%dl。它们事实上是寄存器%ax，%bx，%cx，%dx 的高 8 位和低 8 位；)
[2] 6 个段寄存器：%cs(code)，%ds(data)，%ss(stack), %es，%fs，%gs；
[3] 3 个控制寄存器：%cr0，%cr2，%cr3；
[4] 6 个 debug 寄存器：%db0，%db1，%db2，%db3，%db6，%db7；
[5] 2 个测试寄存器：%tr6，%tr7；
[6] 8 个浮点寄存器栈：%st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)。

2. Operator Sequence

操作数排列是从源（左）到目的（右），如“movl %eax(源), %ebx(目的）”

3. Immediately Operator

使用立即数，要在数前面加符号$, 如“movl $0x04, %ebx”

或者：

para = 0x04

movl $para, %ebx

指令执行的结果是将立即数 0x04 装入寄存器 ebx。

4. Symbol Constant

符号常数直接引用 如

value: .long 0x12a3f2de

movl value , %ebx

指令执行的结果是将常数 0x12a3f2de 装入寄存器 ebx。

引用符号地址在符号前加符号$, 如“movl $value, % ebx”则是将符号 value 的地址装入寄存器 ebx。

5. Length of Operator

操作数的长度用加在指令后的符号表示 b(byte, 8-bit), w(word, 16-bits), l(long,32-bits)，如“movb %al, %bl”，“movw %ax, %bx”，“movl %eax, %ebx ”。

如 果没有指定操作数长度的话，编译器将按照目标操作数的长度来设置。比如指令“mov %ax, %bx”，由于目标操作数 bx 的长度为 word，那么编译器将把此指令等同于“movw %ax,%bx”。同样道理，指令“mov $4, %ebx”等同于指令“movl $4, %ebx”，“push %al”等同于“pushb %al”。对于没有指定操作数长度，但编译器又无法猜测的指令，编译器将会报错，比如指令“push $4”。

6. Sign and Zero Extension

绝大多数面向 80386 的 AT&T 汇编指令与 Intel 格式的汇编指令都是相同的，但符号扩展指令和零扩展指令有不同格式。符号扩展指令和零扩展指令需要指定源操作数长度和目的操作数长度，即使在某些指令中这些操作数是隐含的。

在 AT&T 语法中，符号扩展和零扩展指令的格式为，基本部分"movs"和"movz"（对应 Intel 语法的 movsx 和 movzx），后面跟上源操作数长度和目的操作数长度。movsbl 意味着 movs （from）byte （to）long；movbw 意味着 movs （from）byte

（to）word；movswl 意味着 movs （from）word （to）long。对于 movz 指令也一样。比如指令“movsbl %al,%edx”意味着将 al 寄存器的内容进行符号扩展后放置到 edx 寄存器中。

其它的 Intel 格式的符号扩展指令还有：
cbw -- sign-extend byte in %al to word in %ax；
cwde -- sign-extend word in %ax to long in %eax；
cwd -- sign-extend word in %ax to long in %dx:%ax；
cdq -- sign-extend dword in %eax to quad in %edx:%eax；
对应的 AT&T 语法的指令为 cbtw，cwtl，cwtd，cltd。

7. Call and Jump

段内调用和跳转指令为"call"，"ret"和"jmp"，段间调用和跳转指令为"lcall"，"lret"和"ljmp"。段间调用和跳转指令的格式为“lcall/ljmp $SECTION, $OFFSET”，而段间返回指令则为“lret $STACK-ADJUST”。

8. Prefix

操作码前缀被用在下列的情况：
[1]字符串重复操作指令(rep,repne)；
[2]指定被操作的段(cs,ds,ss,es,fs,gs)；
[3]进行总线加锁(lock)；
[4]指定地址和操作的大小(data16,addr16)；
在 AT&T 汇编语法中，操作码前缀通常被单独放在一行，后面不跟任何操作数。例如，对于重复 scas 指令，其写法为：
repne
scas
上述操作码前缀的意义和用法如下：
[1]指定被操作的段前缀为 cs,ds,ss,es,fs,和 gs。在 AT&T 语法中，只需要按照
section:memory-operand 的格式就指定了相应的段前缀。比如：
lcall %cs:realmode_swtch
[2]操作数 / 地址大小前缀是“data16”和"addr16"，它们被用来在 32-bit 操作数 / 地址代码中指定 16-bit 的操作数 / 地址。
[3]总线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅仅对 ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG,DEC,
INC, NEG, NOT, OR, SBB, SUB, XOR, XADD,XCHG 指令有效，如果将 Lock 前
缀用在其它指令之前，将会引起异常。
[4]字符串重复操作前缀"rep","repe","repne"用来让字符串操作重复“%ecx”次。

9. Memory Reference

Intel 语法的间接内存引用的格式为：
section:[base+index*scale+displacement]
而在 AT&T 语法中对应的形式为：
section:displacement(base,index,scale)
其中，base 和 index 是任意的 32-bit base 和 index 寄存器。scale 可以取值 1，2，4，8。如果不指定 scale 值，则默认值为 1。section 可以指定任意的段寄存器作为段前缀，默认的段寄存器在不同的情况下不一样。如果在指令中指定了默认的段前缀，则编译器在目标代码中不会产生此段前缀代码。
下面是一些例子：
-4(%ebp)：base=%ebp，displacement=-4，section 没有指定，由于 base＝%ebp，所以默认的 section=%ss，index,scale

没有指定，则 index 为 0。

foo(,%eax,4)：index=%eax，scale=4，displacement=foo。其它域没有指定。这里默认的 section=%ds。

foo(,1)：这个表达式引用的是指针 foo 指向的地址所存放的值。注意这个表达式中没有 base 和 index，并且只有一个逗号，这是一种异常语法，但却合法。

%gs:foo：这个表达式引用的是放置于%gs 段里变量 foo 的值。

如果 call 和 jump 操作在操作数前指定前缀“*”，则表示是一个绝对地址调用/跳转，也就是说 jmp/call 指令指定的是一个绝对地址。如果没有指定"*"，则操作数是一个相对地址。

任何指令如果其操作数是一个内存操作， 则指令必须指定它的操作尺寸(byte,word,long），也就是说必须带有指令后缀(b,w,l)。

Linux 工作在保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑段基址和偏移量，而是采用如下的地址计算方法:

```
disp + base + index * scale
```

下面是一些内存操作数的例子：

AT&T 格式
movl -4(%ebp), %eax
movl array(, %eax, 4), %eax
movw array(%ebx, %eax, 4), %cx
movb $4, %fs:(%eax)

其中下面这些省略了浮点数及 IA-32 如 SSE FPU 等特殊的指令集部分，我觉得重要的是学习 linux 汇编的语法及编译原理和程序控制流程，具体的指令细节就不那么重要了。

```
#########################################################################################################
# 一, IA-32 硬件特性
#########################################################################################################
```

寄存器:
1, 通用寄存器, 用于存放正在处理的数据
EAX 用于操作数和结果数的累加器
EBX 指向数据内存断中的数据的指针
ECX 字符串和循环操作的计数器
EDX IO 指针
EDI 用于字符串操作的目标的数据指针
ESI 用于字符串操作的源的数据指针
ESP 堆栈指针
EBP 堆栈数据指针

其中寄存器 EAX, EBX, ECX, EDX 又可以通过 16 位和 8 位寄存器名称引用如 EAX, AX 引用 EAX 低 16 位, AL 引用 EAX 低 8 位, AH 引用 AL 之后的高 8 位

2, 段寄存器:
IA-32 平台允许使用 3 中内存模型: 平坦内存模式 分段内存模式 实地址模式

平坦内存: 把全部的系统内存表示为连续的地址空间， 通过线性地址的特定地址访问内存位置.

分段内存: 把系统内存划分为独立的段组, 通过位于寄存器中的指针进行引用. 每个段用于包含特定类型的数据。 一个段用于包含指令码, 另一个段包含数据元素, 第三个段包含数据堆栈。
段中的内存位置是通过逻辑地址引用的, 逻辑地址是由段地址加上偏移量构成， 处理器把逻辑地址转换为相应的线性地址以便访问。

段寄存器：
CS 代码段
DS 数据段
SS 堆栈段
ES 附加段指针
FS 附加段指针
GS 附加段指针

每个段寄存器都是 16 位的， 包含指向内存特定段起始位置的指针，程序不能显示加载或改变 CS 寄存器, DS, ES, FS, GS 都用于指向数据段, 通过 4 个独立的段, 程序可以分隔数据元素， 确保他们不会重叠， 程序必须加载带有段的正确指针值的数据段寄存器， 并且使用偏移值引用各个内存的位置。
SS 段寄存器用于指向堆栈段, 堆栈包含传递给函数和过程的数据值。

实地址: 如果实地址模式, 所有段寄存器都指向线性 0 地址， 并且都不会被程序改动， 所有的指令码 数据元素 堆栈元素 都是通过他们的线性地址直接访问的。

3， 指令指针寄存器
是 EIP 寄存器, 它跟踪要执行程序的下一条指令代码, 应用程序不能修改指令指针本身，不能指定内存地址把它拖放 EIP 寄存器中，相反必须通过一般的跳转指令来改变预存取缓存的下一条指令。

在平坦内存模型中， 指令指针包含下一条指令码的线性地址, 在分段模型中指令指针包含逻辑地址指针， 通过 CS 寄存器的内存引用。

4, 控制寄存器
CRO 控制操作模式 和 处理器当前状态的系统标志
CR1 当前没有使用
CR2 内存页面错误信息
CR3 内存页面目录信息
CR4 支持处理器特性和说明处理器特性能力的标志

不能直接访问控制寄存器, 但是能把控制寄存器中的值传递给通用寄存器，如果必须改动控制寄存器的标志， 可以改动通用寄存器的值， 然后把内容传递给控制寄存器。

标志:
IA-32 使用单一的寄存器来包含一组状态控制和系统标志, EFLAGS 寄存器包含 32 位标志信息

1, 状态标志
标志 位 说明
CF 0 进位标志, 如果无符号数的数学操作产生最高有效位的进位或者借位, 此时值为 1
PF 2 奇偶校验标志, 用于表明数学操作的结果寄存器中的是否包含错误数据
AF 4 辅助进位标志, 用于二进制编码的 10 进制(BCD)的数学操作中， 如果用于运算的寄存器的第三位发生进位或借位， 该值为 1
ZF 6 0 标志, 如果操作为 0， 则该值为 1
SF 7 符号标志, 设置为结果的最高有效位, 这一位是符号位表明结果是正值还是负值
OF 11 溢出标志

2, 控制标志
当前只定义了一个控制标志 DF 即方向标志, 用于控制处理器处理字符串的方式如果设置为 1, 字符串指令自动递减内存地址以便到达字符串中的下一字节。
反之。

3, 系统标志
标志 位 说明
TF 8 陷阱标志, 设置为 1 时启用单步模式, 在单步模式下处理器每次只执行一条命令。
IF 9 中断使能标志, 控制处理器如响应从外部源接收到的信号。
IOPL 12 和 13 IO 特权级别标志, 表明当前正在运行任务的 IO 特权级别, 它定义 IO 地址空间的特权访问级别, 该值必须小于或者等于访问 I/O 地址空间的级别; 否则任何访问 IO 空间的请求都会被拒绝!
NT 14 嵌套任务标志控制当前运行的任务是否连接到前一个任务, 它用于连接被中断和被调用的任务.

RF 16 恢复标志用于控制在调试模式中如何响应异常。
VM 17 虚拟 8086 模式, 表明处理器在虚拟 8086 模式中而不是保护模式或者实模式。
AC 18 对准检查标志, 用于启用内存引用的对准检查
VIF 19 虚拟中断标志, 当处理器在虚拟模式中操作时, 该标志起 IF 标志的作用.
VIP 20 虚拟中断挂起标志, 在虚拟模式操作时用于表示一个中断正在被挂起。
ID 21 表示 CPU 是否支持 cpuid 指令, 如果处理器能够设置或者清零这个标志, 表示处理器支持该指令。

##
二，GNU 汇编工具系列
##
1, 二进制工具系列
addr2line 把地址转换成文件名或者行号

ar 创建 修改或者展开文件存档

as 把汇编语言代码汇编成目标代码
常用选项:
-a -> 指定输出中包含那些清单
-D -> 包含它用于向下兼容 但是被忽略
--defsym -> 在汇编代码之前定义符号和值
-f -> 快速汇编跳过注释和空白
--gstabs -> 包含每行源代码的调试信息
--gstats+ -> 包含 gdb 专门的调试信息
-I -> 指定包含文件的目录
-J -> 不警告带符号溢出
-L -> 在符号表中保存本地符号
-o -> 给定输出目标名
-R -> 把数据段合并进文本段
--statistics -> 显示汇编使用的最大空间和总时间
-v -> 显示 as 的版本号
-W -> 不显示警告信息

c++filt 还原 c++符号的过滤器

gprof 显示程序简档信息的程序

ld 把目标代码文件转换成可执行文件的转换器
常用选项:
-d -> 指定目标代码输入文件的格式
-Bstatic -> 只使用静态库
-Bdynamic -> 只使用动态库
-Bsymbolic-> 把引用捆绑到共享库中的全局符号
-c -> 从指定的命令文件读取命令
-cref -> 创建跨引用表
-defsym -> 在输出文件中创建指定的全局符号
-demangle -> 在错误消息中还原符号名称
-e -> 使用指定的符号作为程序的初始执行点
-E -> 对于 elf 文件把所有的符号添加到动态符号表
-share -> 创建共享库
-Ttext -> 使用指定的地址作为文本段的起始点
-Tdata -> 使用指定的地址作为数据段的起始点
-Tbss -> 使用指定的地址作为 bss 段的起始点
-L -> 把指定的路径添加到库搜索清单
-O -> 生成优化的输出文件
-o -> 指定输出名
-oformat -> 指定输出文件的二进制格式
-R -> 从指定的文件读取符号和地址
-rpath -> 把指定的位置添加到运行时库搜索路径
-rpath-link-> 指定搜索运行时共享库的路径

-X -> 删除本地所有临时符号
-x -> 删除本地所有符号

nm 列出目标文件中的符号

objcopy 复制或翻译目标文件

objdump 显示来自目标文件的信息

ranlib 生成存档文件内容的索引

readelf 按照 elf 格式显示目标文件信息

size 列出目标文件或者存档文件的段长度

strings 显示目标文件中可打印字符串

strip 丢弃符号

windres 编译 Microsoft Windows 资源文件

2, GNU 编译器
gcc
常用选项:
-c 编译或者汇编代码但不进行连接
-S 编译后停止但不进行汇编
-E 预处理后停止但不进行编译
-o 指定输出文件名
-v 显示每个编译阶段使用的命令
-std 指定使用的语言标准
-g 生成调试信息
-pg 生成 gprof 制作简档要使用的额外代码
-O 优化可执行代码
-W 设置编译器警告级别
-I 指定包含文件清单
-L 指定库文件目录
-D 预定义源代码中使用的宏
-U 取消任何定义了的宏
-f 指定控制编译器行为的选项
-m 指定与硬件相关的选项

3, GNU 调试程序
gdb
常用选项:
-d 指定远程调试时串行接口的线路速度
-batch 以批处理模式运行
-c 指定要分析的核心转储文件
-cd 指定工作目录
-d 指定搜索源文件的目录
-e 指定要执行的文件
-f 调试时以标准格式输出文件名和行号
-q 安静模式
-s 指定符号的文件名
-se 指定符号和要执行的文件名
-tty 设置标准输出和输入设备
-x 从指定的文件执行 gdb 命令

由于 gnu 调试时忽略开始处断点, 需要在开始标签处执行一个空指令
如:
.globl _start
_start:
nop

此时断点可以设置成 break *_start+1
查看寄存器状态 info registers
使用 print 命令查看特定寄存器或者变量的值, 加上修饰符可以得到不同的输出格式:
print/d 显示十进制数字
print/t 显示二进制数字
print/x 显示 16 进制数字
使用 x 命令可以查看特定内存的值:
x/nyz
其中 n 为要显示的字段数
y 时输出格式, 它可以是:
c 用于字符, d 用于十进制, x 用于 16 进制
z 是要显示的字段长度, 它可以是:
b 用于字节, h 用于 16 字节, w 用于 32 位字
如:
x/42cb 用于显示前 42 字节

##
三, GNU 汇编语言结构
##
主要包括三个常用的段:
data 数据段 声明带有初始值的元素
bss 数据段 声明使用 0 或者 null 初始化的元素
text 正文段 包含的指令, 每个汇编程序都必须包含此段

使用.section 指令定义段, 如:
.section .data
.section .bss
.section .text

起始点:
gnu 汇编器使用_start 标签表示默认的起始点, 此外如果想要汇编内部的标签能够被外部程序访问,需要使用.globl 指令,
如:.globl _start

使用通用库函数时可以使用:
ld -dynamic-linker /lib/ld-linux.so.2

##
四, 数据传递
##
1, 数据段
使用.data 声明数据段, 这个段中声明的任何数据元素都保留在内存中并可以被汇编程序的指令读取,此外还可以使用.rodata 声明只读的数据段, 在声明一个数据元素时, 需要使用标签和命令:

标签:用做引用数据元素所使用的标记， 它和 c 语言的变量很相似, 它对于处理器是没有意义的, 它只是用做汇编器试图访问内存位置时用做引用指针的一个位置。

指令:这个名字指示汇编器为通过标签引用的数据元素保留特定数量的内存， 声明命令之后必须给出一个或多个默认值。

声明指令:
.ascii 文本字符串
.asciz 以空字符结尾的字符串
.byte 字节值
.double 双精度浮点值

.float 单精度浮点值
.int 32 位整数
.long 32 位整数, 和 int 相同
.octa 16 字节整数
.quad 8 字节整数
.short 16 位整数
.single 单精度浮点数(和 float 相同)

例子:
output:
.ascii "hello world."

pi:
.float 2.14

声明可以在一行中定义多个值, 如:
ages:
.int 20, 10, 30, 40

定义静态符号:
使用.equ 命令把常量值定义为可以在文本段中使用的符号，如:
.section .data
.equ LINUX_SYS_CALL, 0x80
.section .text
movl $LINUX_SYS_CALL, %eax

2, bss 段
和 data 段不同, 无需声明特定的数据类型, 只需声明为所需目的保留的原始内存部分即可。
GNU 汇编器使用以下两个命令声明内存区域:
.comm 声明为未初始化的通用内存区域
.lcomm 声明为未初始化的本地内存区域

两种声明很相似， 但.lcomm 是为不会从本地汇编代码之外进行访问的数据保留的， 格式为:
.comm/.lcomm symbol, length

例子:
.section .bss
.lcomm buffer, 1000
该语句把 1000 字节的内存地址赋予标签 buffer, 在声明本地通用内存区域的程序之外的函数是不能访问他们的.(不能在.globl 命令中使用他们)

在 bss 段声明的好处是, 数据不包含在可执行文件中。在数据段中定义数据时, 它必须被包含在可执行程序中, 因为必须使用特定值初始化它。因为不使用数据初始化 bss 段中声明的数据区域,所以内存区域被保留在运行时使用, 并且不必包含在最终的程序中

3， 传送数据
move 指令:
格式 movex 源操作数, 目的操作数。 其中 x 为要传送数据的长度, 取值有:
l 用于 32 位的长字节
w 用于 16 位的字
b 用于 8 位的字节值

立即数前面要加一个$符号, 寄存器前面要加%符号。

8 个通用的寄存器是用于保存数据的最常用的寄存器, 这些寄存器的内容可以传递给其他的任何可用的寄存器。 和通用寄存器不同, 专用寄存器(控制, 调试, 段)的内容只能传送给通用寄存器, 或者接收从通用寄存器传过来的内容。

在对标签进行引用时:
例:
```
.section .data
value:
.int 100
_start:
movl value, %eax
movl $value, %eax
movl %ebx, (%edi)
movl %ebx, 4(%edi)
```

其中:movl value, %eax 只是把标签 value 当前引用的内存值传递给 eax
movl $value, %eax 把标签 value 当前引用的内存地址指针传递给 eax
movl %ebx, (%edi) 如果 edi 外面没有括号那么这个指令只是把 ebx 中的
值加载到 edi 中, 如果有了括号就表示把 ebx 中的内容
传送给 edi 中包含的内存位置。
movl %ebx, 4（%edi) 表示把 edi 中的值放在 edi 指向的位置之后的 4 字节内存位置中
movl %ebx, -4(%edi) 表示把 edi 中的值放在 edi 指向的位置之前的 4 字节内存位置中

cmove 指令(条件转移):
cmovex 源操作数, 目的操作数. x 的取值为:
无符号数:
a/nbe 大于/不小于或者等于
ae/nb 大于或者等于/不小于
nc 无进位
b/nae 小于/不大于等于
c 进位
be/na 小于或等于/不大于
e/z 等于/零
ne/nz 不等于/不为零
p/pe 奇偶校验/偶校验
np/po 非奇偶校验/奇校验

有符号数:
ge/nl 大于或者等于/不小于
l/nge 小于/不大于或者等于
le/ng 小于或者等于/不大于
o 溢出
no 未溢出
s 带符号（负)
ns 无符号(非负)

交换数据:
xchg 在两个寄存器之间或者寄存器和内存间交换值如:
xchg 操作数, 操作数, 要求两个操作数必须长度相同且不能同时都是内存位置其中寄存器可以是 32，16，8 位的 bswap 反转一个 32 位寄存器的字节顺序如: bswap %ebx

xadd 交换两个值 并把两个值只和存储在目标操作数中如: xadd 源操作数,目标操作数
其中源操作数必须是寄存器, 目标操作数可以是内存位置也可以是寄存器其中寄存器可以是 32，16，8 位的

cmpxchg
cmpxchg source, destination
其中 source 必须是寄存器, destination 可以是内存或者寄存器, 用来比较两者的值, 如果相等，就把源操作数的值加载到目标操作数中, 如果不等就把目标操作数加载到源操作数中，其中寄存器可以是 32，16，8 位的, 其中源操作数是 EAX,AX 或者 AL 寄存器中的值

cmpxchg8b 同 cmpxchg， 但是它处理 8 字节值， 同时它只有一个操作数
cmpxchg8b destination 其中 destination 引用一个内存位置, 其中的 8 字节值会与 EDX 和 EAX 寄存器中包含的值(EDX 高位寄存器,EAX 低位寄存器)进行比较, 如果目标值和 EDX:EAX 对中的值相等, 就把 EDX:EAX 对中的 64 位值传递给内存位置, 如果不匹配就把内存地址中的值加载到 EDX:EAX 对中

4, 堆栈
ESP 寄存器保存了当前堆栈的起始位置, 当一个数据压入栈时， 它就会自动递减, 反之其自动递增

压入堆栈操作:
pushx source, x 取值为:
l 32 位长字
w 16 位字

弹出堆栈操作:
popx source
其中 source 必须是 16 或 32 位寄存器或者内存位置, 当 pop 最后一个元素时 ESP 值应该和以前的相等

5，压入和弹出所有寄存器
pusha/popa 压入或者弹出所有 16 位通用寄存器
pushad/popad 压入或者弹出所有 32 位通用寄存器
pushf/popf 压入或者弹出 EFLAGS 寄存器的低 16 位
pushfd/popfd 压入或者弹出 EFLAGS 寄存器的全部 32 位

6，数据地址对齐
gas 汇编器支持.align 命令， 它用于在特定的内存边界对准定义的数据元素， 在数据段中.align 命令紧贴在数据定义的前面

##
五，控制流程
##
无条件跳转:
1, 跳转
jmp location 其中 location 为要跳转到的内存地址, 在汇编中为定义的标签

2，调用
调用指令分为两个部分:
1, 调用 call address 跳转到指定位置
2, 返回指令 ret, 它没有参数紧跟在 call 指令后面的位置

执行 call 指令时，它把 EIP 的值放到堆栈中， 然后修改 EIP 以指向被调用的函数地址， 当被调用函数完成后， 它从堆栈获取过去的 EIP 的值， 并把控制权返还给原始程序。

3，中断
由硬件设备生成中断。 程序生成软件中断当一个程序产生中断调用时, 发出调用的程序暂停, 被调用的程序接替它运行, 指令指针被转移到被调用的函数地址， 当调用完成时使用中断返回指令可以返回调原始程序。

条件跳转：
条件跳转按照 EFLAGS 中的值来判断是否该跳转，格式为：

jxx address，其中 xx 是 1－3 个字符的条件代码，取值如下：
a 大于时跳转
ae 大于等于
b 小于
be 小于等于
c 进位
cxz 如果 CX 寄存器为 0
ecxz 如果 ECS 寄存器为 0
e 相等
na 不大于
nae 不大于或者等于
nb 不小于
nbe 不小于或等于
nc 无进位
ne 不等于
g 大于(有符号)
ge 大于等于(有符号)
l 小于(有符号)
le 小于等于(有符号)
ng 不大于(有符号)
nge 不大于等于(有符号)
nl 不小于
nle 不小于等于
no 不溢出
np 不奇偶校验
ns 无符号
nz 非零
o 溢出
p 奇偶校验
pe 如果偶校验
po 如果奇校验
s 如果带符号
z 如果为零

条件跳转不支持分段内存模型下的远跳转， 如果在该模式下进行程序设计必须使用程序逻辑确定条件是否存在，然后实现无条件跳转，跳转前必须设置 EFLAGS 寄存器

比较：
cmp operend1, operend2

进位标志修改指令：
CLC 清空进位标志(设置为 0)
CMC 对进位标志求反(把它改变为相反的值)
STC 设置进位标志(设置为 1)

循环：
loop 循环直到 ECX 寄存器为 0
loope/loopz 循环直到 ecx 寄存器为 0 或者没有设置 ZF 标志
loopne/loopnz 循环直到 ecx 为 0 或者设置了 ZF 标志

指令格式为：loopxx address 注意循环指令只支持 8 位偏移地址

##
六，数字
##

IA-32 平台中存储超过一字节的数都被存储为小尾数的形式但是把数字传递给寄存器时，寄存器里面保存是按照大尾数的形式存储

把无符号数转换成位数更大的值时， 必须确保所有的高位部分都被设置为零

把有符号数转换成位数更大的数时：
intel 提供了 movsx 指令它允许扩展带符号数并保留符号，它与 movzx 相似，但是它假设要传送的字节是带符号数形式

浮点数：
fld 指令用于把浮点数字传送入和传送出 FPU 寄存器，格式：
fld source
其中 source 可以为 32 64 或者 80 位整数值

IA-32 使用 FLD 指令用于把存储在内存中的单精度和双精度浮点值 FPU 寄存器堆栈中， 为了区分这两种长度 GNU 汇编器使用 FLDS 加载单精度浮点数，FLDL 加载双精度浮点数

类似 FST 用于获取 FPU 寄存器堆栈中顶部的值，并且把这个值放到内存位置中，对于单精度使用 FSTS，对于双精度使用 FSTL

##
七，基本数学运算
##
1，加法
ADD source，destination 把两个整数相加
其中 source 可以是立即数内存或者寄存器，destination 可以是内存或者寄存器，但是两者不能同时都是内存位置

ADC 和 ADD 相似进行加法运算，但是它把前一个 ADD 指令的产生进位标志的值包含在其中，在处理位数大于 32(如 64) 位的整数时，该指令非常有用

2，减法
SUB source，destination 把两个整数相减
NEG 它生成值的补码
SBB 指令，和加法操作一样，可以使用进位情况帮助执行大的无符号数值的减法运算. SBB 在多字节减法操作中利用进位和溢出标志实现跨数据边界的的借位特性

3，递增和递减
dec destination 递减
inc destination 递增

其中 dec 和 inc 指令都不会影响进位标志，所以递增或递减计数器的值都不会影响程序中涉及进位标志的其他任何运算

4，乘法
mul source 进行无符号数相乘
它使用隐含的目标操作数，目标位置总是使用 eax 的某种形式，这取决与源操作数的长度， 因此根据源操作数的长度,目标操作数必须放在 AL，AX，EAX 中。 此外由于乘法可能产生很大的值，目标位置必须是源操作数的两倍位置，源为 8 时，应该是 16，源为 16 时，应该为 32，但是当源为 16 位时 intel 为了向下兼容，目标操作数不是存放在 eax 中，而是分别存放在 DX:AX 中，结果高位存储在 DX 中，地位存储在 AX 中。对于 32 位的源，目标操作数存储在 EDX:EAX 中，其中 EDX 存储的是高 32 位，EAX 存储的是低 32 位

imul source 进行有符号数乘法运算, 其中的目标操作数和 mul 的一样

imul source, destination 也可以执行有符号乘法运算, 但是此时可以把目标放在指定的位置, 使用这种格式的缺陷
在与乘法的操作结果被限制为单一目标寄存器的长度.

imul multiplier, source, destination
其中 multiplier 是一个立即数, 这种方式允许一个值与给定的源操作数进行快速的乘法运算, 然后把结果存储在通用寄存器中

5, 除法
div divisor 执行无符号数除法运算
除数的最大值取决与被除数的长度, 对于 16 位被除数 ,除数只能为 8 位, 32 或 64 位同上
被除数 被除数长度 商 余数
AX 16 位 AL AH
DX:AX 32 位 AX DX
EDX:EAX 64 位 EAX EDX

idiv divisor 执行有符号数的除法运算, 方式和 div 一样

6, 移位
左移位:
sal 向左移位
sal destination 把 destination 向左移动 1 位
sal %cl, destination 把 destination 的值向左移动 CL 寄存器中指定的位数
sal shifter, destination 把 destination 的值向左移动 shifter 值指定的位数
向左移位可以对带符号数和无符号数执行向左移位的操作, 移位造成的空位用零填充, 移位造成的超过数据长度的任何位都被存放在进位标志中, 然后在下一次移位操作中被丢弃

右移位:
shr 向右移位
sar 向右移位
SHR 指令清空移位造成的空位, 所以它只能对无符号数进行移位操作
SAR 指令根据整数的符号位, 要么清空, 要么设置移位造成的空位, 对于负数, 空位被设置为 1

循环移位:
和移位指令类似, 只不过溢出的位被存放回值的另一端, 而不是丢弃
ROL 向左循环移位
ROR 向右循环移位
RCL 向左循环移位, 并且包含进位标志
RCR 向右循环移位, 并且包含进位标志

7, 逻辑运算
AND OR XOR
这些指令使用相同的格式:
and source, destination
其中 source 可以是 8 位 16 位或者 32 位的立即值 寄存器或内存中的值, destination 可以是 8 位 16 位或者 32 位寄存器或内存中的值, 不能同时使用内存值作为源和目标。 布尔逻辑功能对源和目标执行按位操作。
也就是说使用指定的逻辑功能按照顺序对数据的元素的每个位进行单独比较。

NOT 指令使用单一操作数, 它即是源值也是目标结果的位置
清空寄存器的最高效方式是使用 OR 指令对寄存器和它本身进行异或操作.当和本身进行 XOR 操作时, 每个设置为 1 的位就变为 0， 每个设置为 0 的位也变位 0。

位测试可以使用以上的逻辑运算指令, 但这些指令会修改 destination 的值, 因此 intel 提供了 test 指令, 它不会修改目标值而是设置相应的标志

##
八，字符串处理

##
1, 传送字符串
movs 有三种格式
movsb 传送单一字节
movsw 传送一个字
movsl 传送双字

movs 指令使用隐含的源和目的操作数， 隐含的源操作数是 ESI， 隐含的目的操作数是 EDI， 有两种方式加载内存地址到 ESI 和 EDI，第一种是使用标签间接寻址 movl $output, %ESI, 第二种是使用 lea 指令, lea 指令加载对象的地址到指定的目的操作数如 lea output, %esi, 每次执行 movs 指令后， 数据传送后 ESI 和 EDI 寄存器会自动改变，为另一次传送做准备, ESI 和 EDI 可能随着标志 DF 的不同自动递增或者自动递减， 如果 DF 标志为 0 则 movs 指令后 ESI 和 EDI 会递增, 反之会递减， 为了设置 DF 标志, 可以使用一下指令:
CLD 将 DF 标志清零
STD 设置 DF 标志

2，rep 前缀
REP 指令的特殊之处在与它不执行什么操作, 这条指令用于按照特定次数重复执行字符串指令, 有 ECX 寄存器控制,但不需要额外的 loop 指令, 如 rep movsl

rep 的其他格式:
repe 等于时重复
repne 不等于时重复
repnz 不为零时重复
repz 为零时重复

3, 存储和加载字符串
LODS 加载字符串, ESI 为源, 当一次执行完 lods 时会递增或递减 ESI 寄存器， 然后把字符串值存放到 EAX 中

STOS 使用 lods 把字符串值加载到 EAX 后， 可以使用它把 EAX 中的值存储到内存中去:
stos 使用 EDI 作为目的操作数, 执行 stos 指令后, 会根据 DF 的值自动递增或者递减 EDI 中的值

4, 比较字符串
cmps 和其他的操作字符串的指令一样, 隐含的源和目标操作数都为 ESI 和 EDI, 每次执行时都会根据 DF 的值把 ESI 和 EDI 递增或者递减, cmps 指令从目标字符串中减去源字符串, 执行后会设置 EFLAGS 寄存器的状态.

5，扫描字符串
scas 把 EDI 作为目标, 它把 EDI 中的字符串和 EAX 中的字符串进行比较 ,然后根据 DF 的值递增或者递减 EDI

##
九，使用函数
##
GNU 汇编语言定义函数的语法:
.type 标签(也就是函数名), @function
ret 返回到调用处

##
十，linux 系统调用
##
linux 系统调用的中断向量为 0x80

1, 系统调用标识存放在%eax 中
2, 系统调用输入值:
EBX 第一个参数

ECX 第二个参数
EDX 第三个参数
ESI 第四个参数
EDI 第五个参数

需要输入超过 6 个输入参数的系统调用, EBX 指针用于保存指向输入参数内存位置的指针, 输入参数按照连续的的顺序存储, 系统调用的返回值存放在 EAX 中

##
十一，汇编语言的高级功能
##
1，gnu 内联汇编的语法:
asm 或__asm__("汇编代码");
指令必须包含在引号里
如果包含的指令超过一行 必须使用新行分隔符分隔

使用 c 全局变量, 不能在内联汇编中使用局部变量, 注意在汇编语言代码中值被用做内存位置, 而不是立即数值

如果不希望优化内联汇编, 则可以 volatile 修饰符如:__asm__ volatile（"code");

2，GCC 内联汇编的扩展语法
__asm__("assembly code":output locations:input operands:changed registers);
第一部分是汇编代码
第二部分是输出位置, 包含内联汇编代码的输出值的寄存器和内存位置列表
第三部分是输入操作数，包含内联汇编代码输入值的寄存器和内存位置的列表
第四部分是改动的寄存器， 内联汇编改变的任何其他寄存器的列表
这几个部分可以不全有, 但是没有的还必须使用:分隔

1, 指定输入值和输出值， 输入值和输出值的列表格式为:
"constraint"(variable), 其中 variable 是程序中声明的 c 变量, 在扩展 asm 格式中， 局部和全局变量都可以使用,使用 constrant（约束）定义把变量存放到哪(输入)或从哪里传送变量(输出)
约束使用单一的字符, 如下:
约束 描述
a 使用%eax, %ax, %al 寄存器
b 使用%ebx, %bx, %bl 寄存器
c 使用%ecx, %cx, %cl 寄存器
d 使用%edx, %dx, %dl 寄存器
S 使用%esi, %si 寄存器
D 使用%edi, %di 寄存器
r 使用任何可用的通用寄存器
q 使用%eax, %ebx, %ecx,%edx 之一
A 对于 64 位值使用%eax, %edx 寄存器
f 使用浮点寄存器
t 使用第一个(顶部)的浮点寄存器
u 使用第二个浮点寄存器
m 使用变量的内存位置
o 使用偏移内存位置
V 只使用直接内存位置
i 使用立即整数值
n 使用值已知的立即整数值
g 使用任何可用的寄存器和内存位置

除了这些约束之外, 输出值还包含一个约束修饰符:
输出修饰符 描述
+ 可以读取和写入操作数
= 只能写入操作数
% 如果有必要操作数可以和下一个操作数切换
& 在内联函数完成之前, 可以删除和重新使用操作数

如:
__asm__("assembly code": "=a"(result):"d"(data1),"c"(data2));
把 c 变量 data1 存放在 edx 寄存器中, 把 c 变量 data2 存放到 ecx 寄存器中, 内联汇编的结果将存放在 eax 寄存器中, 然后传送给变量 result

在扩展的 asm 语句块中如果要使用寄存器必须使用两个百分号符号

不一定总要在内联汇编代码中指定输出值, 一些汇编指令假定输入值包含输出值, 如 movs 指令

其他扩展内联汇编知识:
1, 使用占位符
输入值存放在内联汇编段中声明的特定寄存器中, 并且在汇编指令中专门使用这些寄存器.虽然这种方式能够很好的处理只有几个输入值的情况, 但对于需要很多输入值的情况, 这中方式显的有点繁琐. 为了帮助解决这个问题, 扩展 asm 格式提供了占位符, 可以在内联汇编代码中使用它引用输入和输出值.

占位符是前面加上百分号的数字, 按照内联汇编中列出的每个输入和输出值在列表中的位置,每个值被赋予从 0 开始的地方. 然后就可以在汇编代码中引用占位符来表示值。

如果内联汇编代码中的输入和输出值共享程序中相同的 c 变量, 则可以指定使用占位符作为约束值, 如:
__asm__("imull %1, %0"
: "=r"(data2)
: "r"(data1), "0"(data2));
如输入输出值中共享相同的变量 data2， 而在输入变量中则可以使用标记 0 作为输入参数的约束

2, 替换占位符
如果处理很多输入和输出值, 数字型的占位符很快就会变的很混乱, 为了使条理清晰 ,GNU 汇编器(从版本 3.1 开始)允许声明替换的名称作为占位符.替换的名称在声明输入值和输出值的段中定义, 格式如下:
%[name]"constraint"(variable)
定义的值 name 成为内联汇编代码中变量的新的占位符号标识, 如下面的例子:
__asm__("imull %[value1], %[value2]"
: [value2] "=r"(data2)
: [value1] "r"(data1), "0"(data2));

3, 改动寄存器列表
编译器假设输入值和输出值使用的寄存器会被改动, 并且相应的作出处理。程序员不需要在改动的寄存器列表中包含这些值, 如果这样做了, 就会产生错误消息. 注意改动的寄存器列表中的寄存器使用完整的寄存器名称, 而不像输入和输出寄存器定义的那样仅仅是单一字母。 在寄存器名称前面使用百分号符号是可选的。

改动寄存器列表的正确使用方法是, 如果内联汇编代码使用了没有被初始化地声明为输入或者输出值的其他任何寄存器 , 则要通知编译器。编译器必须知道这些寄存器, 以避免使用他们。如:
```
int main(void) {
int data1 = 10;
int result = 20;

__asm__("movl %1, %%eax\n\t"
"addl %%eax, %0"
: "=r"(result)
: "r"(data1), "0"(result)
: "%eax");
printf("The result is %d\n", result);
return 0;
}
```

4, 使用内存位置
虽然在内联汇编代码中使用寄存器比较快, 但是也可以直接使用 c 变量的内存位置。约束 m 用于引用输入值和输出值中的内存位置。记住, 对于要求使用寄存器的汇编指令, 仍然必须使用寄存器, 所以不得不定义保存数据的中间寄存器。如:
```
int main(void) {
```

```
int dividentd = 20;
int divisor = 5;
int result;

__asm__("divb %2\n\t"
"movl %%eax, %0"
: "=m"(result)
: "a"(dividend), "m"(divisor));
printf("The result is %d\n", result);
return 0;
}
```

5, 处理跳转
内联汇编语言代码也可以包含定义其中位置的标签。 可以实现一般的汇编条件分支和无条件分支， 如：

```
int main(void) {
int a = 10;
int b = 20;
int result;

__asm__("cmp %1, %2\n\t"
"jge greater\n\t"
"movl %1, %0\n\t"
"jmp end\n"
"greater:\n\t"
"movl %2, %0\n"
"end:"
:"=r"(result)
:"r"(a), "r"(b));
printf("The larger value is %d\n", result);
return 0;
}
```

在内联汇编代码中使用标签时有两个限制。 第一个限制是只能跳转到相同的 asm 段内的标签，不能从一个 asm 段跳转到另一个 asm 段中的标签。第二个限制更加复杂一点。 以上程序使用标签 greater 和 end。 但是, 这样有个潜在的问题, 查看汇编后的代码清单， 可以发现内联汇编标签也被编码到了最终汇编后的代码中。 这意味着如果在 c 代码中还有另一个 asm 段, 就不能再次使用相同的标签, 否则会因为标签重复使用而导致错误消息。还有如果试图整合使用 c 关键字(比如函数名称或者全局变量)的标签也会导致错误。

```
##############################################################################################################
# 十二，优化你的代码
##############################################################################################################
```

GNU 编译器提供-O 选项供程序优化使用:
-O 提供基础级别的优化
-O2 提供更加高级的代码优化
-O3 提供最高级的代码优化
不同的优化级别使用的优化技术也可以单独的应用于代码。 可以使用-f 命令行选项引用每个单独的优化技术。

1， 编译器优化级别 1
在优化的第一个级别执行基础代码的优化。 这个级别试图执行 9 种单独的优化功能：
-fdefer-pop: 这种优化技术与汇编语言代码在函数完成时如何进行操作有关。 一般情况下, 函数的输入值被保存在堆栈种并且被函数访问。函数返回时, 输入值还在堆栈种。 一般情况下, 函数返回之后, 输入值被立即弹出堆栈。这样做会使堆栈种的内容有些杂乱。

-fmerge-constans: 使用这种优化技术, 编译器试图合并相同的常量. 这一特性有时候会导致很长的编译时间, 因为编译器必须分析 c 或者 c++程序中用到的每个常量，并且相互比较他们.

-fthread-jumps: 使用这种优化技术与编译器如果处理汇编代码中的条件和非条件分支有关。 在某些情况下, 一条跳转指令可能转移到另一条分支语句。 通过一连串跳转, 编译器确定多个跳转之间的最终目标并且把第一个跳转重新定向到最终目标。

-floop-optimize: 通过优化如何生成汇编语言中的循环， 编译器可以在很大程序上提高应用程序的性能。 通常, 程序由很多大型且复杂的循

环构成。 通过删除在循环内没有改变值的变量赋值操作，可以减少循环内执行指令的数量，在很大程度上提高性能。 此外优化那些确定何时离开循环的条件分支， 以便减少分支的影响。

-fif-conversion：if-then 语句应该是应用程序中仅次于循环的最消耗时间的部分。简单的 if-then 语句可能在最终的汇编语言代码中产生众多的条件分支。 通过减少或者删除条件分支，以及使用条件传送 设置标志和使用运算技巧来替换他们，编译器可以减少 if-then 语句中花费的时间量。

-fif-conversion2：这种技术结合更加高级的数学特性， 减少实现 if-then 语句所需的条件分支。

-fdelayed-branch：这种技术试图根据指令周期时间重新安排指令。 它还试图把尽可能多的指令移动到条件分支前，以便最充分的利用处理器的治理缓存。

-fguess-branch-probability：就像其名称所暗示的，这种技术试图确定条件分支最可能的结果，并且相应的移动指令，这和延迟分支技术类似。因为在编译时预测代码的安排，所以使用这一选项两次编译相同的 c 或者 c++代码很可能会产生不同的汇编语言代码， 这取决于编译时编译器认为会使用那些分支。 因为这个原因，很多程序员不喜欢采用这个特性，并且专门地使用-fno-guess-branch-probability 选项关闭这个特性

-fcprop-registers：因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性(两个段要求使用相同的寄存器)并且删除不必要的寄存器复制操作。

2，编译器优化级别 2
结合了第一个级别的所有优化技术，再加上一下一些优化：
-fforce-mem：这种优化再任何指令使用变量前，强制把存放再内存位置中的所有变量都复制到寄存器中。 对于只涉及单一指令的变量，这样也许不会有很大的优化效果. 但是对于再很多指令(必须数学操作)中都涉及到的变量来说，这会时很显著的优化，因为和访问内存中的值相比 ，处理器访问寄存器中的值要快的多。

-foptimize-sibling-calls：这种技术处理相关的和/或者递归的函数调用。 通常，递归的函数调用可以被展开为一系列一般的指令， 而不是使用分支。 这样处理器的指令缓存能够加载展开的指令并且处理他们，和指令保持为需要分支操作的单独函数调用相比，这样更快。

-fstrength-reduce：这种优化技术对循环执行优化并且删除迭代变量。 迭代变量是捆绑到循环计数器的变量，比如使用变量，然后使用循环计数器变量执行数学操作的 for-next 循环。

-fgcse： 这种技术对生成的所有汇编语言代码执行全局通用表达式消除历程。 这些优化操作试图分析生成的汇编语言代码并且结合通用片段，消除冗余的代码段。如果代码使用计算性的 goto，gcc 指令推荐使用-fno-gcse 选项。

-fcse-follow-jumps：这种特别的通用子表达式消除技术扫描跳转指令，查找程序中通过任何其他途径都不会到达的目标代码。这种情况最常见的例子就式 if-then-else 语句的 else 部分。

-frerun-cse-after-loop：这种技术在对任何循环已经进行过优化之后重新运行通用子表达式消除例程。这样确保在展开循环代码之后更进一步地优化还编代码。

-fdelete-null-pointer-checks：这种优化技术扫描生成的汇编语言代码，查找检查空指针的代码。 编译器假设间接引用空指针将停止程序。如果在间接引用之后检查指针， 它就不可能为空。

-fextensive-optimizations：这种技术执行从编译时的角度来说代价高昂的各种优化技术，但是它可能对运行时的性能产生负面影响。

-fregmove：编译器试图重新分配 mov 指令中使用的寄存器，并且将其作为其他指令操作数，以便最大化捆绑的寄存器的数量。

-fschedule-insns：编译器将试图重新安排指令，以便消除等待数据的处理器。 对于在进行浮点运算时有延迟的处理器来说， 这使处理器在等待浮点结果时可以加载其他指令。

-fsched-interblock：这种技术使编译器能够跨越指令块调度指令。 这可以非常灵活地移动指令以便等待期间完成的工作最大化。

-fcaller-saves：这个选项指示编译器对函数调用保存和恢复寄存器，使函数能够访问寄存器值，而且不必保存和恢复他们。 如果调用多个函数，这样能够节省时间，因为只进行一次寄存器的保存和恢复操作，而不是在每个函数调用中都进行。

-fpeephole2：这个选项允许进行任何计算机特定的观察孔优化。

-freorder-blocks：这种优化技术允许重新安排指令块以便改进分支操作和代码局部性。

-fstrict-aliasing：这种技术强制实行高级语言的严格变量规则。 对于 c 和 c++程序来说，它确保不在数据类型之间共享变量. 例如，整数变量不和单精度浮点变量使用相同的内存位置。

-funit-at-a-time: 这种优化技术指示编译器在运行优化例程之前读取整个汇编语言代码。 这使编译器可以重新安排不消耗大量时间的代码以便优化指令缓存。 但是, 这会在编译时花费相当多的内存, 对于小型计算机可能是一个问题。

-falign-functions: 这个选项用于使函数对准内存中特定边界的开始位置。 大多数处理器按照页面读取内存，并且确保全部函数代码位于单一内存页面内, 就不需要叫化代码所需的页面。

-fcrossjumping: 这是对跨越跳转的转换代码处理， 以便组合分散在程序各处的相同代码。 这样可以减少代码的长度， 但是也许不会对程序性能有直接影响。

3, 编译器优化级别 3
它整合了第一和第二级别中的左右优化技巧, 还包括一下优化:
-finline-functions: 这种优化技术不为函数创建单独的汇编语言代码， 而是把函数代码包含在调度程序的代码中。 对于多次被调用的函数来说, 为每次函数调用复制函数代码。 虽然这样对于减少代码长度不利, 但是通过最充分的利用指令缓存代码, 而不是在每次函数调用时进行分支操作, 可以提高性能。

-fweb: 构建用于保存变量的伪寄存器网络。 伪寄存器包含数据, 就像他们是寄存器一样, 但是可以使用各种其他优化技术进行优化, 比如 cse 和 loop 优化技术。

-fgcse-after-reload: 这中技术在完全重新加载生成的且优化后的汇编语言代码之后执行第二次 gcse 优化,帮助消除**不同优化方式创建的任何冗余段。**

二、Hello World!

真不知道打破这个传统会带来什么样的后果，但既然所有程序设计语言的第一个例子都是在屏幕上打印一个字符串 "Hello World!"，那我们也以这种方式来开始介绍 Linux 下的汇编语言程序设计。

在 Linux 操作系统中，你有很多办法可以实现在屏幕上显示一个字符串，但最简洁的方式是使用 Linux 内核提供的系统调用。使用这种方法最大的好处是可以直接和操作系统的内核进行通讯，不需要链接诸如 libc 这样的函数库，也不需要使用 ELF 解释器，因而代码尺寸小且执行速度快。

Linux 是一个运行在保护模式下的 32 位操作系统，采用 flat memory 模式，目前最常用到的是 ELF 格式的二进制代码。一个 ELF 格式的可执行程序通常划分为如下几个部分：.text、.data 和 .bss，其中 .text 是只读的代码区，.data 是可读可写的数据区，而 .bss 则是可读可写且没有初始化的数据区。代码区和数据区在 ELF 中统称为 section，根据实际需要你可以使用其它标准的 section，也可以添加自定义 section，但一个 ELF 可执行程序至少应该有一个 .text 部分。 下面给出我们的第一个汇编程序，用的是 AT&T 汇编语言格式：

例 1. AT&T 格式

```
#hello.s
.data                   # 数据段声明
        msg : .string "Hello, world!\n" # 要输出的字符串
        len = . - msg               # 字串长度
.text                   # 代码段声明
.global _start          # 指定入口函数

_start:                 # 在屏幕上显示一个字符串
        movl $len, %edx  # 参数三：字符串长度
        movl $msg, %ecx  # 参数二：要显示的字符串
        movl $1, %ebx    # 参数一：文件描述符(stdout)
        movl $4, %eax    # 系统调用号(sys_write)
        int  $0x80       # 调用内核功能
```

```
                                # 退出程序
        movl $0,%ebx    # 参数一：退出代码
        movl $1,%eax    # 系统调用号(sys_exit)
        int  $0x80      # 调用内核功能
```

初次接触到 AT&T 格式的汇编代码时，很多程序员都认为太晦涩难懂了，没有关系，在 Linux 平台上你同样可以使用 Intel 格式来编写汇编程序：

例 2. Intel 格式

```
; hello.asm
section .data           ; 数据段声明
        msg db "Hello, world!", 0xA     ; 要输出的字符串
        len equ $ - msg                 ; 字串长度
section .text           ; 代码段声明
global _start           ; 指定入口函数
_start:                 ; 在屏幕上显示一个字符串
        mov edx, len    ; 参数三：字符串长度
        mov ecx, msg    ; 参数二：要显示的字符串
        mov ebx, 1      ; 参数一：文件描述符(stdout)
        mov eax, 4      ; 系统调用号(sys_write)
        int 0x80        ; 调用内核功能
                        ; 退出程序
        mov ebx, 0      ; 参数一：退出代码
        mov eax, 1      ; 系统调用号(sys_exit)
        int 0x80        ; 调用内核功能
```

上面两个汇编程序采用的语法虽然完全不同，但功能却都是调用 Linux 内核提供的 sys_write 来显示一个字符串，然后再调用 sys_exit 退出程序。在 Linux 内核源文件 include/asm-i386/unistd.h 中，可以找到所有系统调用的定义。

三、Linux 汇编工具

Linux 平台下的汇编工具虽然种类很多，但同 DOS/Windows 一样，最基本的仍然是汇编器、连接器和调试器。

1.汇编器

汇编器（assembler）的作用是将用汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法，可以用来汇编用 AT&T 格式编写的程序：

```
[xiaowp@gary code]$ as -o hello.o hello.s
```

Linux 平台上另一个经常用到的汇编器是 NASM，它提供了很好的宏指令功能，并能够支持相当多的目标代码格式，包括 bin、a.out、coff、elf、rdf 等。NASM 采用的是人工编写的语法分析器，因而执行速度要比 GAS 快很多，更重要的是它使用的是 Intel 汇编语法，可以用来编译用 Intel 语法格式编写的汇编程序：

```
[xiaowp@gary code]$ nasm -f elf hello.asm
```

2.链接器

由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码。链接器通常用来将多个目标代码连接成一个可执行代码，这样可以先将整个程序分成几个模块来单独开发，然后才将它们组合(链接)成一个应用程序。 Linux 使用 ld 作为标准的链接程序，它同样也包含在 binutils 软件包中。汇编程序在成功通过 GAS 或 NASM 的编译并生成目标代码后，就可以使用 ld 将其链接成可执行程序了：

```
[xiaowp@gary code]$ ld -s -o hello hello.o
```

3.调试器

有人说程序不是编出来而是调出来的，足见调试在软件开发中的重要作用，在用汇编语言编写程序时尤其如此。Linux 下调试汇编代码既可以用 GDB、DDD 这类通用的调试器，也可以使用专门用来调试汇编代码的 ALD(Assembly Language Debugger)。

从调试的角度来看，使用 GAS 的好处是可以在生成的目标代码中包含符号表(symbol table)，这样就可以使用 GDB 和 DDD 来进行源码级的调试了。要在生成的可执行程序中包含符号表，可以采用下面的方式进行编译和链接：

```
[xiaowp@gary code]$ as --gstabs -o hello.o hello.s
[xiaowp@gary code]$ ld -o hello hello.o
```

执行 as 命令时带上参数 --gstabs 可以告诉汇编器在生成的目标代码中加上符号表，同时需要注意的是，在用 ld 命令进行链接时不要加上 -s 参数，否则目标代码中的符号表在链接时将被删去。

在 GDB 和 DDD 中调试汇编代码和调试 C 语言代码是一样的，你可以通过设置断点来中断程序的运行，查看变量和寄存器的当前值，并可以对代码进行单步跟踪。图 1 是在 DDD 中调试汇编代码时的情景：

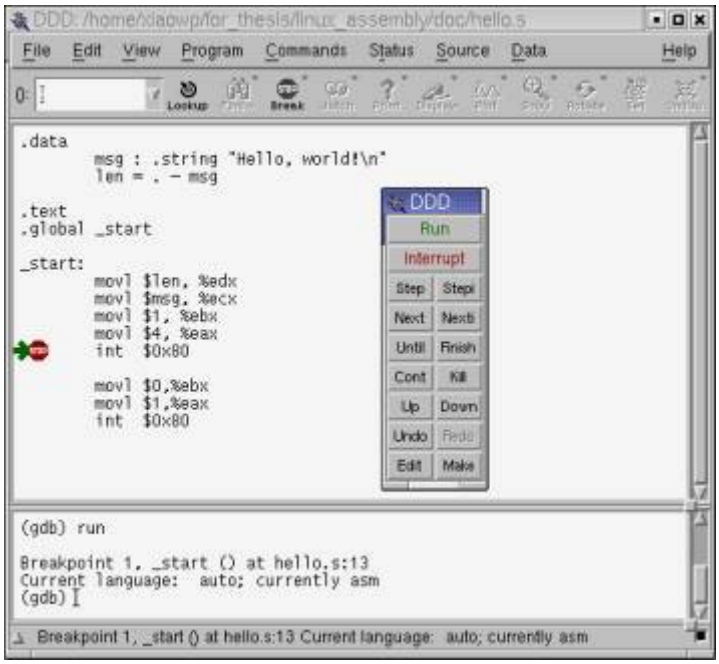

图 1 用 DDD 中调试汇编程序

汇编程序员通常面对的都是一些比较苛刻的软硬件环境，短小精悍的 ALD 可能更能符合实际的需要，因此下面主要介绍一下如何用 ALD 来调试汇编程序。首先在命令行方式下执行 ald 命令来启动调试器，该命令的参数是将要被调试的可执行程序：

```
[xiaowp@gary doc]$ ald hello
Assembly Language Debugger 0.1.3
Copyright (C) 2000-2002 Patrick Alken
hello: ELF Intel 80386 (32 bit), LSB, Executable, Version 1 (current)
```

```
Loading debugging symbols...(15 symbols loaded)
ald>
```

当 ALD 的提示符出现之后，用 disassemble 命令对代码段进行反汇编：

```
ald> disassemble -s .text
Disassembling section .text (0x08048074 - 0x08048096)
08048074  BA0F000000              mov edx, 0xf
08048079  B998900408              mov ecx, 0x8049098
0804807E  BB01000000              mov ebx, 0x1
08048083  B804000000              mov eax, 0x4
08048088  CD80                    int 0x80
0804808A  BB00000000              mov ebx, 0x0
0804808F  B801000000              mov eax, 0x1
08048094  CD80                    int 0x80
```

上述输出信息的第一列是指令对应的地址码，利用它可以设置在程序执行时的断点：

```
ald> break 0x08048088
Breakpoint 1 set for 0x08048088
```

断点设置好后，使用 run 命令开始执行程序。ALD 在遇到断点时将自动暂停程序的运行，同时会显示所有寄存器的当前值：

```
ald> run
Starting program: hello
Breakpoint 1 encountered at 0x08048088
eax = 0x00000004 ebx = 0x00000001 ecx = 0x08049098 edx = 0x0000000F
esp = 0xBFFFF6C0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x08048088 eflags = 0x00000246
Flags: PF ZF IF
08048088  CD80                    int 0x80
```

如果需要对汇编代码进行单步调试，可以使用 next 命令：

```
ald> next
Hello, world!
eax = 0x0000000F ebx = 0x00000000 ecx = 0x08049098 edx = 0x0000000F
esp = 0xBFFFF6C0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x0804808F eflags = 0x00000346
Flags: PF ZF TF IF
0804808F  B801000000              mov eax, 0x1
```

若想获得 ALD 支持的所有调试命令的详细列表，可以使用 help 命令：

```
ald> help
```

```
Commands may be abbreviated.
If a blank command is entered, the last command is repeated.
Type `help <command>' for more specific information on <command>.
General commands
attach      clear       continue    detach      disassemble
enter       examine     file        help        load
next        quit        register    run         set
step        unload      window      write
Breakpoint related commands
break       delete      disable     enable      ignore
lbreak      tbreak
```

使用宏

清单 3 演示本节讨论的概念；它接受用户名作为输入并返回一句问候语。

清单 3. 读取字符串并向用户显示问候语的程序

行号	NASM	GAS
001	section .data	.section .data
002		
003	prompt_str db 'Enter your name: '	prompt_str:
004		.ascii "Enter Your Name: "
005	; $ is the location counter	pstr_end:
006	STR_SIZE equ $ - prompt_str	.set STR_SIZE, pstr_end - prompt_str
007		
008	greet_str db 'Hello '	greet_str:
009		.ascii "Hello "
010		
011	GSTR_SIZE equ $ - greet_str	gstr_end:
012		.set GSTR_SIZE, gstr_end - greet_str
013		
014	section .bss	.section .bss
015		
016	; Reserve 32 bytes of memory	// Reserve 32 bytes of memory
017	buff resb 32	.lcomm buff, 32
018		
019	; A macro with two parameters	// A macro with two parameters
020	; Implements the write system call	// implements the write system call
021	%macro write 2	.macro write str, str_size
022	mov eax, 4	movl $4, %eax
023	mov ebx, 1	movl $1, %ebx
024	mov ecx, %1	movl \str, %ecx

```
025         mov   edx, %2
026         int   80h
027     %endmacro
028
029
030 ; Implements the read system call
031     %macro read 2
032         mov   eax, 3
033         mov   ebx, 0
034         mov   ecx, %1
035         mov   edx, %2
036         int   80h
037     %endmacro
038
039
040 section .text
041
042     global _start
043
044     _start:
045         write prompt_str, STR_SIZE
046         read  buff, 32
047
048 ; Read returns the length in eax
049         push  eax
050
051 ; Print the hello text
052         write greet_str, GSTR_SIZE
053
054         pop   edx
055
056 ; edx  = length returned by read
057         write buff, edx
058
059     _exit:
060         mov   eax, 1
061         mov   ebx, 0
062         int   80h
```

```
        movl  \str_size, %edx
        int   $0x80
    .endm


// Implements the read system call
    .macro read buff, buff_size
        movl  $3, %eax
        movl  $0, %ebx
        movl  \buff, %ecx
        movl  \buff_size, %edx
        int   $0x80
    .endm


.section .text

    .globl _start

    _start:
        write $prompt_str, $STR_SIZE
        read  $buff, $32

// Read returns the length in eax
        pushl %eax

// Print the hello text
        write $greet_str, $GSTR_SIZE

        popl  %edx

// edx = length returned by read
    write $buff, %edx

    _exit:
        movl  $1, %eax
        movl  $0, %ebx
        int   $0x80
```

本节要讨论宏以及 NASM 和 GAS 对它们的支持。但是，在讨论宏之前，先与其他几个特性做一下比较。

清单 3 演示了未初始化内存的概念，这是用 .bss 部分指令（第 14 行）定义的。BSS 代表 “block storage segment”（原来是以一个符号开头的块），BSS 部分中保留的内存在程序启动时初始化为零。BSS 部分中的对象只有一个名称和大小，没有值。与数据部分中不同，BSS 部分中声明的变量并不实际占用空间。

NASM 使用 resb、resw 和 resd 关键字在 BSS 部分中分配字节、字和双字空间。GAS 使用 .lcomm 关键字分配字节级空间。请注意在这个程序的两个版本中声明变量名的方式。在 NASM 中，变量名前面加 resb（或 resw 或 resd）关键字，后面是要保留的空间量；在 GAS 中，变量名放在 .lcomm 关键字的后面，然后是一个逗号和要保留的空间量。

NASM：varname resb size

GAS：.lcomm varname, size

清单 3 还演示了位置计数器的概念（第 6 行）。 NASM 提供特殊的变量（$ 和 $$ 变量）来操作位置计数器。在 GAS 中，无法操作位置计数器，必须使用标签计算下一个存储位置（数据、指令等等）。

例如，为了计算一个字符串的长度，在 NASM 中会使用以下指令：

```
prompt_str db 'Enter your name: '
STR_SIZE equ $ - prompt_str      ; $ is the location counter
```

$ 提供位置计数器的当前值，从这个位置计数器中减去标签的值（所有变量名都是标签），就会得出标签的声明和当前位置之间的字节数。equ 用来将变量 STR_SIZE 的值设置为后面的表达式。GAS 中使用的相似指令如下：

```
prompt_str:
      .ascii "Enter Your Name: "

pstr_end:
      .set STR_SIZE, pstr_end - prompt_str
```

末尾标签（pstr_end）给出下一个位置地址，减去启始标签地址就得出大小。还要注意，这里使用 .set 将变量 STR_SIZE 的值设置为逗号后面的表达式。也可以使用对应的 .equ。在 NASM 中，没有与 GAS 的 set 指令对应的指令。

正如前面提到的，清单 3 使用了宏（第 21 行）。在 NASM 和 GAS 中存在不同的宏技术，包括单行宏和宏重载，但是这里只关注基本类型。宏在汇编程序中的一个常见用途是提高代码的清晰度。通过创建可重用的宏，可以避免重复输入相同的代码段；这不但可以避免重复，而且可以减少代码量，从而提高代码的可读性。

NASM 使用 %beginmacro 指令声明宏，用 %endmacro 指令结束声明。%beginmacro 指令后面是宏的名称。宏名称后面是一个数字，这是这个宏需要的宏参数数量。在 NASM 中，宏参数是从 1 开始连续编号的。也就是说，宏的第一个参数是 %1，第二个是 %2，第三个是 %3，以此类推。例如：

```
%beginmacro macroname 2
      mov eax, %1
      mov ebx, %2
%endmacro
```

这创建一个有两个参数的宏，第一个参数是 %1，第二个参数是 %2。因此，对上面的宏的调用如下所示：

```
macroname 5, 6
```

还可以创建没有参数的宏，在这种情况下不指定任何数字。

现在看看 GAS 如何使用宏。GAS 提供 .macro 和 .endm 指令来创建宏。.macro 指令后面跟着宏名称，后面可以有参数，也可以没有参数。在 GAS 中，宏参数是按名称指定的。例如：

```
.macro macroname arg1, arg2
      movl \arg1, %eax
      movl \arg2, %ebx
.endm
```

当在宏中使用宏参数名称时，在名称前面加上一个反斜线。如果不这么做，链接器会把名称当作标签而不是参数，因此会报告错误。

回页首

函数、外部例程和堆栈

本节的示例程序在一个整数数组上实现选择排序。

清单 4. 在整数数组上实现选择排序

行号	NASM	GAS
001	section .data	.section .data
002		
003	array db	array:
004	89, 10, 67, 1, 4, 27, 12, 34,	.byte 89, 10, 67, 1, 4, 27, 12,
005	86, 3	34, 86, 3
006		
007	ARRAY_SIZE equ $ - array	array_end:
008		.equ ARRAY_SIZE, array_end - array
009		
010	array_fmt db " %d", 0	array_fmt:
011		.asciz " %d"
012		
013	usort_str db "unsorted array:", 0	usort_str:
014		.asciz "unsorted array:"
015		
016	sort_str db "sorted array:", 0	sort_str:
017		.asciz "sorted array:"
018		
019	newline db 10, 0	newline:
020		.asciz "\n"
021		
022		
023	section .text	.section .text
024	extern puts	
025		
026	global _start	.globl _start
027		
028	_start:	_start:
029		
030	push usort_str	pushl $usort_str
031	call puts	call puts
032	add esp, 4	addl $4, %esp
033		
034	push ARRAY_SIZE	pushl $ARRAY_SIZE

```
035         push  array
036         push  array_fmt
037         call  print_array10
038         add   esp, 12
039
040         push  ARRAY_SIZE
041         push  array
042         call  sort_routine20
043
044 ; Adjust the stack pointer
045         add   esp, 8
046
047         push  sort_str
048         call  puts
049         add   esp, 4
050
051         push  ARRAY_SIZE
052         push  array
053         push  array_fmt
054         call  print_array10
055         add   esp, 12
056         jmp   _exit
057
058         extern printf
059
060     print_array10:
061         push  ebp
062         mov   ebp, esp
063         sub   esp, 4
064         mov   edx, [ebp + 8]
065         mov   ebx, [ebp + 12]
066         mov   ecx, [ebp + 16]
067
068         mov   esi, 0
069
070     push_loop:
071         mov   [ebp - 4], ecx
072         mov   edx, [ebp + 8]
073         xor   eax, eax
074         mov   al, byte [ebx + esi]
075         push  eax
076         push  edx
```

```
        pushl $array
        pushl $array_fmt
        call  print_array10
        addl  $12, %esp

        pushl $ARRAY_SIZE
        pushl $array
        call  sort_routine20

# Adjust the stack pointer
        addl  $8, %esp

        pushl $sort_str
        call  puts
        addl  $4, %esp

        pushl $ARRAY_SIZE
        pushl $array
        pushl $array_fmt
        call  print_array10
        addl  $12, %esp
        jmp   _exit


    print_array10:
        pushl %ebp
        movl  %esp, %ebp
        subl  $4, %esp
        movl  8(%ebp), %edx
        movl  12(%ebp), %ebx
        movl  16(%ebp), %ecx

        movl  $0, %esi

    push_loop:
        movl  %ecx, -4(%ebp)
        movl  8(%ebp), %edx
        xorl  %eax, %eax
        movb  (%ebx, %esi, 1), %al
        pushl %eax
        pushl %edx
```

```
077
078             call  printf
079             add   esp, 8
080             mov   ecx, [ebp - 4]
081             inc   esi
082             loop  push_loop
083
084             push  newline
085             call  printf
086             add   esp, 4
087             mov   esp, ebp
088             pop   ebp
089             ret
090
091         sort_routine20:
092             push  ebp
093             mov   ebp, esp
094
095     ; Allocate a word of space in stack
096             sub   esp, 4
097
098     ; Get the address of the array
099             mov   ebx, [ebp + 8]
100
101     ; Store array size
102             mov   ecx, [ebp + 12]
103             dec   ecx
104
105     ; Prepare for outer loop here
106             xor   esi, esi
107
108         outer_loop:
109     ; This stores the min index
110             mov   [ebp - 4], esi
111             mov   edi, esi
112             inc   edi
113
114         inner_loop:
115             cmp   edi, ARRAY_SIZE
116             jge   swap_vars
117             xor   al, al
118             mov   edx, [ebp - 4]
```

```
             call  printf
             addl  $8, %esp
             movl  -4(%ebp), %ecx
             incl  %esi
             loop  push_loop

             pushl $newline
             call  printf
             addl  $4, %esp
             movl  %ebp, %esp
             popl  %ebp
             ret

        sort_routine20:
             pushl %ebp
             movl  %esp, %ebp

    # Allocate a word of space in stack
             subl  $4, %esp

    # Get the address of the array
             movl  8(%ebp), %ebx

    # Store array size
             movl  12(%ebp), %ecx
             decl  %ecx

    # Prepare for outer loop here
             xorl  %esi, %esi

        outer_loop:
    # This stores the min index
             movl  %esi, -4(%ebp)
             movl  %esi, %edi
             incl  %edi

        inner_loop:
             cmpl  $ARRAY_SIZE, %edi
             jge   swap_vars
             xorb  %al, %al
             movl  -4(%ebp), %edx
```

```
119         mov   al, byte [ebx + edx]
120         cmp   byte [ebx + edi], al
121         jge   check_next
122         mov   [ebp - 4], edi
123
124     check_next:
125         inc   edi
126         jmp   inner_loop
127
128     swap_vars:
129         mov   edi, [ebp - 4]
130         mov   dl, byte [ebx + edi]
131         mov   al, byte [ebx + esi]
132         mov   byte [ebx + esi], dl
133         mov   byte [ebx + edi], al
134
135         inc   esi
136         loop  outer_loop
137
138         mov   esp, ebp
139         pop   ebp
140         ret
141
142     _exit:
143         mov   eax, 1
144         mov   ebx, 0
145         int   80h
```

```
        movb  (%ebx, %edx, 1), %al
        cmpb  %al, (%ebx, %edi, 1)
        jge   check_next
        movl  %edi, -4(%ebp)

    check_next:
        incl  %edi
        jmp   inner_loop

    swap_vars:
        movl  -4(%ebp), %edi
        movb  (%ebx, %edi, 1), %dl
        movb  (%ebx, %esi, 1), %al
        movb  %dl, (%ebx, %esi, 1)
        movb  %al, (%ebx,  %edi, 1)

        incl  %esi
        loop  outer_loop

        movl  %ebp, %esp
        popl  %ebp
        ret

    _exit:
        movl  $1, %eax
        movl  0, %ebx
        int   $0x80
```

初看起来清单 4 似乎非常复杂，实际上它是非常简单的。这个清单演示了函数、各种内存寻址方案、堆栈和库函数的使用方法。这个程序对包含 10 个数字的数组进行排序，并使用外部 C 库函数 `puts` 和 `printf` 输出未排序数组和已排序数组的完整内容。为了实现模块化和介绍函数的概念，排序例程本身实现为一个单独的过程，数组输出例程也是这样。我们来逐一分析一下。

在声明数据之后，这个程序首先执行对 `puts` 的调用（第 31 行）。`puts` 函数在控制台上显示一个字符串。它惟一的参数是要显示的字符串的地址，通过将字符串的地址压入堆栈（第 30 行），将这个参数传递给它。

在 NASM 中，任何不属于我们的程序但是需要在链接时解析的标签都必须预先定义，这就是 `extern` 关键字的作用（第 24 行）。GAS 没有这样的要求。在此之后，字符串的地址 `usort_str` 被压入堆栈（第 30 行）。在 NASM 中，内存变量（比如 `usort_str`）代表内存位置本身，所以 `push usort_str` 这样的调用实际上是将地址压入堆栈的顶部。但是在 GAS 中，变量 `usort_str` 必须加上前缀`$`，这样它才会被当作地址。如果不加前缀 `$`，那么会将内存变量代表的实际字节压入堆栈，而不是地址。

因为在堆栈中压入一个变量会让堆栈指针移动一个双字，所以给堆栈指针加 4（双字的大小）（第 32 行）。

现在将三个参数压入堆栈，并调用 `print_array10` 函数（第 37 行）。在 NASM 和 GAS 中声明函数的方法是相同的。它们仅仅是通过 `call` 指令调用的标签。

在调用函数之后，ESP 代表堆栈的顶部。esp + 4 代表返回地址，esp + 8 代表函数的第一个参数。在堆栈指针上加上双字变量的大小（即 esp + 12、esp + 16 等等），就可以访问所有后续参数。
在函数内部，通过将 esp 复制到 ebp （第 62 行）创建一个局部堆栈框架。和程序中的处理一样，还可以为局部变量分配空间（第 63 行）。方法是从 esp 中减去所需的字节数。esp - 4 表示为一个局部变量分配 4 字节的空间，只要堆栈中有足够的空间容纳局部变量，就可以继续分配。
清单 4 演示了基间接寻址模式（第 64 行），也就是首先取得一个基地址，然后在它上面加一个偏移量，从而到达最终的地址。在清单的 NASM 部分中，[ebp + 8] 和 [ebp - 4]（第 71 行）就是基间接寻址模式的示例。在 GAS 中，寻址方法更简单一些：4(%ebp) 和 -4(%ebp)。
在 print_array10 例程中，在 push_loop 标签后面可以看到另一种寻址模式（第 74 行）。在 NASM 和 GAS 中的表示方法如下：
NASM：mov al, byte [ebx + esi]
GAS：movb (%ebx, %esi, 1), %al
这种寻址模式称为基索引寻址模式。这里有三项数据：一个是基地址，第二个是索引寄存器，第三个是乘数。因为不可能决定从一个内存位置开始访问的字节数，所以需要用一个方法计算访问的内存量。NASM 使用字节操作符告诉汇编器要移动一个字节的数据。在 GAS 中，用一个乘数和助记符中的 b、w 或 l 后缀（例如 movb）来解决这个问题。初看上去 GAS 的语法似乎有点儿复杂。
GAS 中基索引寻址模式的一般形式如下：
%segment:ADDRESS (, index, multiplier)
或
%segment:(offset, index, multiplier)
或
%segment:ADDRESS(base, index, multiplier)
使用这个公式计算最终的地址：
ADDRESS or offset + base + index * multiplier.
因此，要想访问一个字节，就使用乘数 1；对于字，乘数是 2；对于双字，乘数是 4。当然，NASM 使用的语法比较简单。上面的公式在 NASM 中表示为：
Segment:[ADDRESS or offset + index * multiplier]
为了访问 1、2 或 4 字节的内存，在这个内存地址前面分别加上 byte、word 或 dword。

其他方面

清单 5 读取命令行参数的列表，将它们存储在内存中，然后输出它们。

清单 5. 读取命令行参数，将它们存储在内存中，然后输出它们

行号	NASM	GAS
001	section .data	.section .data
002		
003	; Command table to store at most	// Command table to store at most
004	; 10 command line arguments	// 10 command line arguments
005	cmd_tbl:	cmd_tbl:
006	%rep 10	.rept 10
007	dd 0	.long 0
008	%endrep	.endr
009		
010	section .text	.section .text

```
011
012     global _start
013
014     _start:
015 ; Set up the stack frame
016         mov   ebp, esp
017 ; Top of stack contains the
018 ;  number of command line arguments.
019 ; The default value is 1
020         mov   ecx, [ebp]
021
022 ; Exit if arguments are more than 10
023         cmp   ecx, 10
024         jg    _exit
025
026         mov   esi, 1
027         mov   edi, 0
028
029 ; Store the command line arguments
030 ;  in the command table
031     store_loop:
032         mov   eax, [ebp + esi * 4]
033         mov   [cmd_tbl + edi * 4], eax
034         inc   esi
035         inc   edi
036         loop  store_loop
037
038         mov   ecx, edi
039         mov   esi, 0
040
041         extern puts
042
043     print_loop:
044 ; Make some local space
045         sub   esp, 4
046 ; puts function corrupts ecx
047         mov   [ebp - 4], ecx
048         mov   eax, [cmd_tbl + esi * 4]
049         push  eax
050         call  puts
051         add   esp, 4
052         mov   ecx, [ebp - 4]
```

```
    .globl _start

    _start:
// Set up the stack frame
        movl  %esp, %ebp
// Top of stack contains the
//  number of command line arguments.
// The default value is 1
        movl  (%ebp), %ecx

// Exit if arguments are more than 10
        cmpl  $10, %ecx
        jg    _exit

        movl  $1, %esi
        movl  $0, %edi

// Store the command line arguments
//  in the command table
    store_loop:
        movl  (%ebp, %esi, 4), %eax
        movl  %eax, cmd_tbl( , %edi, 4)
        incl  %esi
        incl  %edi
        loop  store_loop

        movl  %edi, %ecx
        movl  $0, %esi




    print_loop:
// Make some local space
        subl  $4, %esp
// puts functions corrupts ecx
        movl  %ecx, -4(%ebp)
        movl  cmd_tbl( , %esi, 4), %eax
        pushl %eax
        call  puts
        addl  $4, %esp
        movl  -4(%ebp), %ecx
```

```
053         inc   esi                     incl  %esi
054         loop  print_loop              loop  print_loop
055
056         jmp   _exit                   jmp   _exit
057
058       _exit:                        _exit:
059         mov   eax, 1                  movl  $1, %eax
060         mov   ebx, 0                  movl  $0, %ebx
061         int   80h                     int   $0x80
```

清单 5 演示在汇编程序中重复执行指令的方法。很自然，这种结构称为重复结构。在 GAS 中，重复结构以 .rept 指令开头（第 6 行）。用一个 .endr 指令结束这个指令（第 8 行）。.rept 后面是一个数字，它指定 .rept/.endr 结构中表达式重复执行的次数。这个结构中的任何指令都相当于编写这个指令 count 次，每次重复占据单独的一行。

例如，如果次数是 3：

```
.rept 3
     movl $2, %eax
.endr
```

就相当于：

```
movl $2, %eax
movl $2, %eax
movl $2, %eax
```

在 NASM 中，在预处理器级使用相似的结构。它以 %rep 指令开头，以 %endrep 结尾。%rep 指令后面是一个表达式（在 GAS 中 .rept 指令后面是一个数字）：

```
%rep <expression>
     nop
%endrep
```

在 NASM 中还有另一种结构，times 指令。与 %rep 相似，它也在汇编级起作用，后面也是一个表达式。例如，上面的 %rep 结构相当于：

```
times <expression> nop
```

以下代码：

```
%rep 3
     mov eax, 2
%endrep
```

相当于：

```
times 3 mov eax, 2
```

它们都相当于：

```
mov eax, 2
mov eax, 2
mov eax, 2
```

在清单 5 中，使用 .rept（或 %rep）指令为 10 个双字创建内存数据区。然后，从堆栈一个个地访问命令行参数，并将它们存储在内存区中，直到命令表填满。

在这两种汇编器中，访问命令行参数的方法是相似的。ESP（堆栈顶部）存储传递给程序的命令行参数数量，默认值是 1（表示没有命令行参数）。`esp + 4` 存储第一个命令行参数，这总是从命令行调用的程序的名称。`esp + 8`、`esp + 12` 等存储后续命令行参数。

还要注意清单 5 中从两边访问内存命令表的方法。这里使用内存间接寻址模式（第 31 行）访问命令表，还使用了 ESI（和 EDI）中的偏移量和一个乘数。因此，NASM 中的 `[cmd_tbl + esi * 4]` 相当于 GAS 中的 `cmd_tbl(, %esi, 4)`。

四、系统调用

即便是最简单的汇编程序，也难免要用到诸如输入、输出以及退出等操作，而要进行这些操作则需要调用操作系统所提供的服务，也就是系统调用。除非你的程序只完成加减乘除等数学运算，否则将很难避免使用系统调用，事实上除了系统调用不同之外，各种操作系统的汇编编程往往都是很类似的。

在 Linux 平台下有两种方式来使用系统调用：利用封装后的 C 库（libc）或者通过汇编直接调用。其中通过汇编语言来直接调用系统调用，是最高效地使用 Linux 内核服务的方法，因为最终生成的程序不需要与任何库进行链接，而是直接和内核通信。

和 DOS 一样，Linux 下的系统调用也是通过中断（int 0x80）来实现的。在执行 int 80 指令时，寄存器 eax 中存放的是系统调用的功能号，而传给系统调用的参数则必须按顺序放到寄存器 ebx，ecx，edx，esi，edi 中，当系统调用完成之后，返回值可以在寄存器 eax 中获得。

所有的系统调用功能号都可以在文件 /usr/include/bits/syscall.h 中找到，为了便于使用，它们是用 SYS_<name> 这样的宏来定义的，如 SYS_write、SYS_exit 等。例如，经常用到的 write 函数是如下定义的：

```
ssize_t write(int fd, const void *buf, size_t count);
```

该函数的功能最终是通过 SYS_write 这一系统调用来实现的。根据上面的约定，参数 fb、buf 和 count 分别存在寄存器 ebx、ecx 和 edx 中，而系统调用号 SYS_write 则放在寄存器 eax 中，当 int 0x80 指令执行完毕后，返回值可以从寄存器 eax 中获得。

或许你已经发现，在进行系统调用时至多只有 5 个寄存器能够用来保存参数，难道所有系统调用的参数个数都不超过 5 吗？当然不是，例如 mmap 函数就有 6 个参数，这些参数最后都需要传递给系统调用 SYS_mmap：

```
void  *  mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
```

当一个系统调用所需的参数个数大于 5 时，执行 int 0x80 指令时仍需将系统调用功能号保存在寄存器 eax 中，所不同的只是全部参数应该依次放在一块连续的内存区域里，同时在寄存器 ebx 中保存指向该内存区域的指针。系统调用完成之后，返回值仍将保存在寄存器 eax 中。

由于只是需要一块连续的内存区域来保存系统调用的参数，因此完全可以像普通的函数调用一样使用栈(stack)来传递系统调用所需的参数。但要注意一点，Linux 采用的是 C 语言的调用模式，这就意味着所有参数必须以相反的顺序进栈，即最后一个参数先入栈，而第一个参数则最后入栈。如果采用栈来传递系统调用所需的参数，在执行 int 0x80 指令时还应该将栈指针的当前值复制到寄存器 ebx 中。

五、命令行参数

在 Linux 操作系统中，当一个可执行程序通过命令行启动时，其所需的参数将被保存到栈中：首先是 argc，然后是指向各个命令行参数的指针数组 argv，最后是指向环境变量的指针数据 envp。在编写汇编语言程序时，很多时候需要对这些参数进行处理，下面的代码示范了如何在汇编代码中进行命令行参数的处理：

例 3. 处理命令行参数

```
# args.s
```

```
.text
.globl _start

_start:
        popl    %ecx                    # argc
vnext:
        popl    %ecx                    # argv
        test    %ecx, %ecx      # 空指针表明结束
        jz  exit
        movl    %ecx, %ebx
        xorl    %edx, %edx
strlen:
        movb    (%ebx), %al
        inc %edx
        inc %ebx
        test    %al, %al
        jnz strlen
        movb    $10, -1(%ebx)
        movl    $4, %eax        # 系统调用号(sys_write)
        movl    $1, %ebx        # 文件描述符(stdout)
        int $0x80
        jmp     vnext
exit:
        movl    $1,%eax         # 系统调用号(sys_exit)
        xorl    %ebx, %ebx      # 退出代码
        int     $0x80

        ret
```

六、GCC 内联汇编

用汇编编写的程序虽然运行速度快，但开发速度非常慢，效率也很低。如果只是想对关键代码段进行优化，或许更好的办法是将汇编指令嵌入到 C 语言程序中，从而充分利用高级语言和汇编语言各自的特点。但一般来讲，在 C 代码中嵌入汇编语句要比"纯粹"的汇编语言代码复杂得多，因为需要解决如何分配寄存器，以及如何与 C 代码中的变量相结合等问题。GCC 提供了很好的内联汇编支持，最基本的格式是：

```
__asm__("asm statements");
```

例如：

```
__asm__("nop");
```

如果需要同时执行多条汇编语句，则应该用"\\n\\t"将各个语句分隔开，例如：

```
__asm__( "pushl %%eax \\n\\t"
         "movl $0, %%eax \\n\\t"
```

```
        "popl %eax");
```

通常嵌入到 C 代码中的汇编语句很难做到与其它部分没有任何关系，因此更多时候需要用到完整的内联汇编格式：

```
__asm__("asm statements" : outputs : inputs : registers-modified);
```

插入到 C 代码中的汇编语句是以":"分隔的四个部分，其中第一部分就是汇编代码本身，通常称为指令部，其格式和在汇编语言中使用的格式基本相同。指令部分是必须的，而其它部分则可以根据实际情况而省略。

在将汇编语句嵌入到 C 代码中时，操作数如何与 C 代码中的变量相结合是个很大的问题。GCC 采用如下方法来解决这个问题：程序员提供具体的指令，而对寄存器的使用则只需给出"样板"和约束条件就可以了，具体如何将寄存器与变量结合起来完全由 GCC 和 GAS 来负责。

在 GCC 内联汇编语句的指令部中，加上前缀'%'的数字(如%0，%1)表示的就是需要使用寄存器的"样板"操作数。指令部中使用了几个样板操作数，就表明有几个变量需要与寄存器相结合，这样 GCC 和 GAS 在编译和汇编时会根据后面给定的约束条件进行恰当的处理。由于样板操作数也使用'%'作为前缀，因此在涉及到具体的寄存器时，寄存器名前面应该加上两个'%'，以免产生混淆。

紧跟在指令部后面的是输出部，是规定输出变量如何与样板操作数进行结合的条件，每个条件称为一个"约束"，必要时可以包含多个约束，相互之间用逗号分隔开就可以了。每个输出约束都以'='号开始，然后紧跟一个对操作数类型进行说明的字后，最后是如何与变量相结合的约束。凡是与输出部中说明的操作数相结合的寄存器或操作数本身，在执行完嵌入的汇编代码后均不保留执行之前的内容，这是 GCC 在调度寄存器时所使用的依据。

输出部后面是输入部，输入约束的格式和输出约束相似，但不带'='号。如果一个输入约束要求使用寄存器，则 GCC 在预处理时就会为之分配一个寄存器，并插入必要的指令将操作数装入该寄存器。与输入部中说明的操作数结合的寄存器或操作数本身，在执行完嵌入的汇编代码后也不保留执行之前的内容。

有时在进行某些操作时，除了要用到进行数据输入和输出的寄存器外，还要使用多个寄存器来保存中间计算结果，这样就难免会破坏原有寄存器的内容。在 GCC 内联汇编格式中的最后一个部分中，可以对将产生副作用的寄存器进行说明，以便 GCC 能够采用相应的措施。

下面是一个内联汇编的简单例子：

例 4.内联汇编

```
/* inline.c */
int main()
{
    int a = 10, b = 0;
    __asm__ __volatile__("movl %1, %%eax;\\n\\r"
                         "movl %%eax, %0;"
                         :"=r"(b)      /* 输出 */
                         :"r"(a)       /* 输入 */
                         :"%eax");     /* 不受影响的寄存器 */

    printf("Result: %d, %d\\n", a, b);
}
```

上面的程序完成将变量 a 的值赋予变量 b，有几点需要说明：

- 变量 b 是输出操作数，通过%0 来引用，而变量 a 是输入操作数，通过%1 来引用。
- 输入操作数和输出操作数都使用 r 进行约束，表示将变量 a 和变量 b 存储在寄存器中。输入约束和输出约束的不同点在于输出约束多一个约束修饰符'='。

- 在内联汇编语句中使用寄存器 eax 时，寄存器名前应该加两个'%'，即%%eax。内联汇编中使用%0、%1 等来标识变量，任何只带一个'%'的标识符都看成是操作数，而不是寄存器。
- 内联汇编语句的最后一个部分告诉 GCC 它将改变寄存器 eax 中的值，GCC 在处理时不应使用该寄存器来存储任何其它的值。
- 由于变量 b 被指定成输出操作数，当内联汇编语句执行完毕后，它所保存的值将被更新。

在内联汇编中用到的操作数从输出部的第一个约束开始编号，序号从 0 开始，每个约束记数一次，指令部要引用这些操作数时，只需在序号前加上'%'作为前缀就可以了。需要注意的是，内联汇编语句的指令部在引用一个操作数时总是将其作为 32 位的长字使用，但实际情况可能需要的是字或字节，因此应该在约束中指明正确的限定符：

限定符	意义
"m"、"v"、"o"	内存单元
"r"	任何寄存器
"q"	寄存器 eax、ebx、ecx、edx 之一
"i"、"h"	直接操作数
"E"和"F"	浮点数
"g"	任意
"a"、"b"、"c"、"d"	分别表示寄存器 eax、ebx、ecx 和 edx
"S"和"D"	寄存器 esi、edi
"I"	常数（0 至 31）

Linux 中 x86 的内联汇编

GCC 为内联汇编提供特殊结构，它具有以下格式：

GCG 的 "asm" 结构

```
   asm ( assembler template

: output operands                (optional)

: input operands                 (optional)

: list of clobbered registers
    (optional)

);
```

本例中，汇编程序模板由汇编指令组成。输入操作数是充当指令输入操作数使用的 C 表达式。输出操作数是将对其执行汇编指令输出的 C 表达式。

内联汇编的重要性体现在它能够灵活操作，而且可以使其输出通过 C 变量显示出来。因为它具有这种能力，所以 "asm" 可以用作汇编指令和包含它的 C 程序之间的接口。

一个非常基本但很重要的区别在于 *简单内联汇编*只包括指令，而 *扩展内联汇编*包括操作数。要说明这一点，考虑以下示例：

内联汇编的基本要素

```
{
    int a=10, b;
    asm ("movl %1, %%eax;

movl %%eax, %0;"
        :"=r"(b)  /* output */
        :"r"(a)       /* input */
        :"%eax"); /* clobbered register */
}
```

在上例中，我们使用汇编指令使 "b" 的值等于 "a"。请注意以下几点：

- "b" 是输出操作数，由 %0 引用，"a" 是输入操作数，由 %1 引用。
- "r" 是操作数的约束，它指定将变量 "a" 和 "b" 存储在寄存器中。请注意，输出操作数约束应该带有一个约束修饰符 "="，指定它是输出操作数。
- 要在 "asm" 内使用寄存器 %eax，%eax 的前面应该再加一个 %，换句话说就%%eax，因为 "asm" 使用 %0、%1 等来标识变量。任何带有一个 % 的数都看作是输入 / 输出操作数，而不认为是寄存器。
- 第三个冒号后的修饰寄存器 %eax 告诉将在 "asm" 中修改 GCC %eax 的值，这样 GCC 就不使用该寄存器存储任何其它的值。
- movl %1, %%eax 将 "a" 的值移到 %eax 中， movl %%eax, %0 将 %eax 的内容移到 "b" 中。因为 "b" 被指定成输出操作数，因此当 "asm" 的执行完成后，它将反映出更新的值。换句话说，对 "asm" 内 "b" 所做的更改将在 "asm" 外反映出来。

现在让我们更详细的了解每一项的含义。

汇编程序模板

汇编程序模板是一组插入到 C 程序中的汇编指令（可以是单个指令，也可以是一组指令）。每条指令都应该由双引号括起，或者整组指令应该由双引号括起。每条指令还应该用一个定界符结尾。有效的定界符为新行 (\n) 和分号 (;)。 '\n' 后可以跟一个 tab(\t) 作为格式化符号，增加 GCC 在汇编文件中生成的指令的可读性。 指令通过数 %0、%1 等来引用 C 表达式（指定为操作数）。

如果希望确保编译器不会在 "asm" 内部优化指令，可以在 "asm" 后使用关键字 "volatile"。如果程序必须与 ANSI C 兼容，则应该使用 __asm__ 和 __volatile__，而不是 asm 和 volatile。

操作数

C 表达式用作 "asm" 内的汇编指令操作数。在汇编指令通过对 C 程序的 C 表达式进行操作来执行有意义的作业的情况下，操作数是内联汇编的主要特性。

每个操作数都由操作数约束字符串指定，后面跟用括弧括起的 C 表达式，例如："constraint" (C expression)。操作数约束的主要功能是确定操作数的寻址方式。

可以在输入和输出部分中同时使用多个操作数。每个操作数由逗号分隔开。

在汇编程序模板内部，操作数由数字引用。如果总共有 *n* 个操作数（包括输入和输出），那么第一个输出操作数的编号为 0，逐项递增，最后那个输入操作数的编号为 *n* -1。总操作数的数目限制在 10，如果机器描述中任何指令模式中的最大操作数数目大于 10，则使用后者作为限制。

修饰寄存器列表

如果 "asm" 中的指令指的是硬件寄存器，可以告诉 GCC 我们将自己使用和修改它们。这样，GCC 就不会假设它装入到这些寄存器中的值是有效值。通常不需要将输入和输出寄存器列为 clobbered，因为 GCC 知道 "asm" 使用它们（因为

它们被明确指定为约束）。不过，如果指令使用任何其它的寄存器，无论是明确的还是隐含的（，也不在输出约束列表中出现），寄存器都必须被指定为修饰列表。修饰寄存器列在第三个冒号之后，其名称被指定为字符串。

至于关键字，如果指令以某些不可预知且不明确的方式修改了内存，则可能将 "memory" 关键字添加到修饰寄存器表中。这样就告诉 GCC 不要在不同指令之间将内存值高速缓存在寄存器中。

操作数约束

前面提到过，"asm" 中的每个操作数都应该由操作数约束字符串描述，后面跟用括弧括起的 C 表达式。操作数约束

主要是确定指令中操作数的寻址方式。约束也可以指定：

- 是否允许操作数位于寄存器中，以及它可以包括在哪些种类的寄存器中
- 操作数是否可以是内存引用，以及在这种情况下使用哪些种类的地址
- 操作数是否可以是立即数

约束还要求两个操作数匹配。

常用约束

在可用的操作数约束中，只有一小部分是常用的；下面列出了这些约束以及简要描述。有关操作数约束的完整列表，请参考 GCC 和 GAS 手册。

寄存器操作数约束（r）

使用这种约束指定操作数时，它们存储在通用寄存器中。请看下例：

```
asm ("movl %%cr3, %0\n" :"=r"(cr3val));
```

这里，变量 cr3val 保存在寄存器中，%cr3 的值复制到寄存器上，cr3val 的值从该寄存器更新到内存中。

指定 "r" 约束时，GCC 可以将变量 cr3val 保存在任何可用的 GPR 中。要指定寄存器，必须通过使用特定的寄存器约束直接指定寄存器名。

```
a    %eax
b    %ebx
c    %ecx
d    %edx
S    %esi
D    %edi
```

内存操作数约束 (m)

当操作数位于内存中时，任何对它们执行的操作都将在内存位置中直接发生，这与寄存器约束正好相反，后者先将值存储在要修改的寄存器中，然后将它写回内存位置中。但寄存器约束通常只在对于指令来说它们是绝对必需的，或者它们可以大大提高进程速度时使用。当需要在 "asm" 内部更新 C 变量，而您又确实不希望使用寄存器来保存其值时，使用内存约束最为有效。例如，idtr 的值存储在内存位置 loc 中：

```
("sidt %0\n" : :"m"(loc));
```

匹配（数字）约束

在某些情况下，一个变量既要充当输入操作数，也要充当输出操作数。可以通过使用匹配约束在 "asm" 中指定这种情况。

```
asm ("incl %0" :"=a"(var):"0"(var));
```

在匹配约束的示例中，寄存器 %eax 既用作输入变量，也用作输出变量。将 var 输入读取到 %eax，增加后将更新的 %eax 再次存储在 var 中。这里的 "0" 指定第 0 个输出变量相同的约束。即，它指定 var 的输出实例只应该存储在 %eax 中。该约束可以用于以下情况：

- 输入从变量中读取，或者变量被修改后，修改写回到同一变量中
- 不需要将输入操作数和输出操作数的实例分开

使用匹配约束最重要的意义在于它们可以导致有效地使用可用寄存器。

一般内联汇编用法示例

以下示例通过各种不同的操作数约束说明了用法。有如此多的约束以至于无法将它们一一列出，这里只列出了最经常使用的那些约束类型。

"asm" 和寄存器约束 "r" 让我们先看一下使用寄存器约束 r 的 "asm"。我们的示例显示了 GCC 如

何分配寄存器，以及它如何更新输出变量的值。

```
int main(void)
{
    int x = 10, y;

    asm ("movl %1, %%eax;

 "movl %%eax, %0;"
        :"=r"(y)  /* y is output operand */
        :"r"(x)       /* x is input operand */
        :"%eax"); /* %eax is clobbered register */
}
```

在该例中，x 的值复制为 "asm" 中的 y。x 和 y 都通过存储在寄存器中传递给 "asm"。为该例生成的汇编代码如下：

```
main:
pushl %ebp
```

```
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
movl -4(%ebp),%edx  /* x=10 is stored in %edx */
#APP    /* asm starts here */
movl %edx, %eax     /* x is moved to %eax */
movl %eax, %edx     /* y is allocated in edx and updated */
#NO_APP /* asm ends here */
movl %edx,-8(%ebp)  /* value of y in stack is updated with

 the value in %edx */
```

当使用 "r" 约束时，GCC 在这里可以自由分配任何寄存器。在我们的示例中，它选择 %edx 来存储 x。在读取了 %edx 中 x 的值后，它为 y 也分配了相同的寄存器。

因为 y 是在输出操作数部分中指定的，所以 %edx 中更新的值存储在 -8(%ebp)，堆栈上 y 的位置中。如果 y 是在输入部分中指定的，那么即使它在 y 的临时寄存器存储值 (%edx) 中被更新，堆栈上 y 的值也不会更新。

因为 %eax 是在修饰列表中指定的，GCC 不在任何其它地方使用它来存储数据。

输入 x 和输出 y 都分配在同一个 %edx 寄存器中，假设输入在输出产生之前被消耗。请注意，如果您有许多指令，就不是这种情况了。要确保输入和输出分配到不同的寄存器中，可以指定 & 约束修饰符。

下面是添加了约束修饰符的示例。

```
int main(void)
{
    int x = 10, y;

    asm ("movl %1, %%eax;

 "movl %%eax, %0;"
        :"=&r"(y) /* y is output operand, note the

 & constraint modifier. */
        :"r"(x)       /* x is input operand */
        :"%eax"); /* %eax is clobbered register */
}
```

以下是为该示例生成的汇编代码，从中可以明显地看出 x 和 y 存储在 "asm" 中不同的寄存器中。

```
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
```

```
movl -4(%ebp),%ecx  /* x, the input is in %ecx */
#APP
    movl %ecx, %eax
    movl %eax, %edx     /* y, the output is in %edx */
#NO_APP
movl %edx,-8(%ebp)
```

特定寄存器约束的使用

现在让我们看一下如何将个别寄存器作为操作数的约束指定。在下面的示例中，cpuid 指令采用 %eax 寄存器中的输入，然后在四个寄存器中给出输出：%eax、%ebx、%ecx、%edx。对 cpuid 的输

入（变量 "op"）传递到 "asm" 的 eax 寄存器中，因为 cpuid 希望它这样做。在输出中

使用 a、b、c 和 d 约束，分别收集四个寄存器中的值。

```
asm ("cpuid"
: "=a" (_eax),
"=b" (_ebx),
"=c" (_ecx),
"=d" (_edx)
: "a" (op));
```

在下面可以看到为它生成的汇编代码（假设 _eax、_ebx 等... 变量都存储在堆栈上）：

```
movl -20(%ebp),%eax /* store 'op' in %eax -- input */
#APP
cpuid
#NO_APP
movl %eax,-4(%ebp)  /* store %eax in _eax -- output */
movl %ebx,-8(%ebp)  /* store other registers in
movl %ecx,-12(%ebp)
 respective output variables */
movl %edx,-16(%ebp)
```

strcpy 函数可以通过以下方式使用 "S" 和 "D" 约束来实现：

```
asm ("cld\n

rep\n

movsb"

: /* no input */
```

```
:"S"(src), "D"(dst), "c"(count));
```

通过使用 "S" 约束将源指针 src 放入 %esi 中，使用 "D" 约束将目的指针 dst 放入 %edi 中。

因为 rep 前缀需要 count 值，所以将它放入 %ecx 中。

在下面可以看到另一个约束，它使用两个寄存器 %eax 和 %edx 将两个 32 位的值合并在一起，

然后生成一个 64 位的值：

```
#define rdtscll(val) \
 __asm__ __volatile__ ("rdtsc" : "=A" (val))
The generated assembly looks like this (if val has a 64 bit memory space).
#APP
rdtsc
#NO_APP
movl %eax,-8(%ebp)  /* As a result of A constraint
movl %edx,-4(%ebp)
 %eax and %edx serve as outputs */
Note here that the values in %edx:%eax serve as 64 bit output.
```

使用匹配约束

在下面将看到系统调用的代码，它有四个参数：

```
#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
"d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}
```

在上例中，通过使用 b、c、d 和 S 约束将系统调用的四个自变量放入 %ebx、%ecx、%edx 和 %esi 中。

请注意，在输出中使用了 "=a" 约束，这样，位于 %eax 中的系统调用的返回值就被放入变量 __res 中。通过将匹配约束 "0" 用作输入部分中第一个操作数约束，syscall 号 __NR_##name 被放入 %eax 中，并用作对系统调用的输入。这样，这里的 %eax 既可以用作输入寄存器，又可以用作输出寄存器。没有其它寄存器用于这个目的。另请注意，输入（syscall 号）在产生输出（syscall 的返回值）之前被消耗（使用）。

内存操作数约束的使用

请考虑下面的原子递减操作：

```
__asm__ __volatile__(
"lock; decl %0"
:"=m" (counter)
:"m" (counter));
```

为它生成的汇编类似于：

```
#APP
    lock
    decl -24(%ebp) /* counter is modified on its memory location */
#NO_APP.
```

您可能考虑在这里为 counter 使用寄存器约束。如果这样做，counter 的值必须先复制到寄存器，递减，然后对其内存更新。但这样您会无法理解锁定和原子性的全部意图，这些明确显示了使用内存约束的必要性。

使用修饰寄存器

请考虑内存拷贝的基本实现。

```
   asm ("movl $count, %%ecx;

up: lodsl;

stosl;

loop up;"
        :           /* no output */
        :"S"(src), "D"(dst) /* input */
        :"%ecx", "%eax" );  /* clobbered list */
```

当 lodsl 修改 %eax 时，lodsl 和 stosl 指令隐含地使用它。%ecx 寄存器明确装入 count。但 GCC 在我们通知它以前是不知道这些的，我们是通过将 %eax 和 %ecx 包括在修饰寄存器集中来通知 GCC 的。

在完成这一步之前，GCC 假设 %eax 和 %ecx 是自由的，它可能决定将它们用作存储其它的数据。请注意，%esi 和 %edi 由 "asm" 使用，它们不在修饰列表中。这是因为已经声明 "asm" 将在输入操作数列表中使用它们。这里最低限度是，如果在 "asm" 内部使用寄存器（无论是明确还是隐含地），既不出现在输入操作数列表中，也不出现在输出操作数列表中，必须将它列为修饰寄存器。