实验目标/前置知识

AT&T汇编语法

操作数顺序：源操作数-目的操作数

寄存器命名：%eax

立即数：$0x常量

操作数大小：在 AT&T 语法中，存储器操作数的大小取决于操作码名字的最后一个字符。操作码后缀 'b' 、'w'、'l' 分别指明了字节（8位）、字（16位）、长型（32位）存储器引用。

```
1.直接寻址
movl 0x8000, %eax 将地址0x8000的值放到eax中
2.寄存器寻址
movl $2, %ebx
3.基址寻址
movl $0x8000,%eax
movl 4(%eax),%ebx 把地址0x8004(0x8000+4)的值放到ebx中
4.变址寻址
movl $0x8000,%eax
movl $0x4,%ebx
movl (%eax,%ebx),%ecx 把地址0x8004(0x8000+0x4)的值放到ecx中
5.比例变址寻址
movl $0x2000,%eax
movl (,%eax,4),%ebx 把地址0x8000(0x2000*4)的值放到ebx中
```

内联汇编

基本格式：`asm("汇编代码");`

```
asm("movl %ecx %eax"); /* 将 ecx 寄存器的内容移至 eax  */
__asm__("movb %bh (%eax)"); /* 将 bh 的一个字节数据 移至 eax 寄存器指向的内存 */
```

```
__asm__ ("movl %eax, %ebx/n/t"
         "movl $56, %esi/n/t"
         "movl %ecx, $label(%edx,%ebx,$4)/n/t"
         "movb %ah, (%ebx)");
```

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

在上面的代码中，`ebx`,`edx`存储的值已经发生了改变，但是gcc是通过生成汇编文件，再交给GAS去进行汇编，这就导致GAS并不能知道代码中已经更改了`edx`的值，如果程序的上下文中需要`edx`或`ebx`作为其它变量的暂存区，就会导致赋值覆盖等问题，

基本格式

```
asm [volatile] ( Assembler Template
   : Output Operands #输出操作数
   [
            : Input Operands #输入操作数
            [
                : Clobbers #修饰寄存器列表
            ]
   ]#可选项
)
```

volatile（这是可选项），其含义是避免 “asm” 指令被删除、移动或组合，在执行代码时，如果不希望汇编语句被 gcc 优化而改变位置，就需要在 asm 符号后添加 volatile 关键词。

每个操作数由一个操作数约束字符串描述，其后紧接一个括弧括起的 C 表达式。

冒号用于将汇编程序模板和第一个输出操作数分开，另一个（冒号）用于将最后一个输出操作数和第一个输入操作数分开（如果存在的话）。

```
#功能：将b的值设为a的值
int a=10, b;
asm ("movl %1, %%eax;
      movl %%eax, %0;"
     :"=r"(b)        /* 输出 */
     :"r"(a)         /* 输入 */
     :"%eax"         /* 修饰寄存器 */
     );
```

解析：

- b为输出操作数，用%0引用，a为输入操作数，用%引用
- `"=r/r"`为操作数约束，`"r"`表示可以使用任意一个寄存器，`"="`表示操作数是只写的。
- 寄存器使用%%前缀，用于区分操作数
- 修饰寄存器 %eax 用于告诉 GCC %eax 的值将会在 "asm" 内部被修改，所以 GCC 将不会使用此寄存器存储任何其他值

约束字符串主要用于决定操作数的寻址方式，同时也用于指定使用的寄存器。

约束和内联汇编有很大的关联。但以上对约束的介绍还不多。约束用于表明

- 操作数是否可以位于寄存器和位于哪种寄存器
- 操作数是否可以是一个内存引用和哪种地址
- 操作数是否可以是一个立即数和可能的取值范围，等等

内存操作数约束 m

`m` 约束允许一个内存操作数，可以使用机器普遍支持的任一种地址。 当操作数位于内存时，任何对它们的操作将直接发生在内存位置，这与寄存器约束相反，后者首先将值存储在要修改的寄存器中，然后将它写回到内存位置。 但寄存器约束通常用于一个指令必须使用它们或者它们可以大大提高处理速度的地方

```
asm ("sidt %0\n" : :"m"(loc)); #将IDTR寄存器的值存储与loc处

IDT:中断描述符表，CPU用来处理中断和程序异常
CPU执行中断指令时，会去IDT表中查找对应的中断服务程序（interrupt service routine ISR）。
中断处理过程是由CPU直接调用的，CPU有专门的寄存器IDTR来保存IDT在内存中的位置。程序可以使用LIDT和SIDT指令来读写IDTR。
```

```
asm ("incl %0" :"=a"(var):"0"(var));
这个匹配约束的示例中，寄存器 %eax 既用作输入变量，也用作输出变量。 var 输入被读进 %eax，并且等递增后更新的 %eax 再次被存储进 var。 这里的 0 用于指定与第 0 个输出变量相同的约束
```

一些其它通用约束

1. `o` 约束：允许一个内存操作数，但只有当地址是可偏移的时。即，该地址加上一个小的偏移量可以得到一个有效地址
2. `v` 约束：一个不允许偏移的内存操作数。换言之，任何适合 “m” 约束而不适合 “o” 约束的操作数
3. `i` 约束：允许一个（带有常量）的立即整形操作数，这包括其值仅在汇编时期知道的符号常量
4. `n` 约束：允许一个带有已知数字的立即整形操作数。许多系统不支持汇编时期的常量，因为操作数少于一个字宽。对于此种操作数，约束应该使用 `n` 而不是 `i`
5. `g` 约束：允许任一寄存器、内存或者立即整形操作数，不包括通用寄存器之外的寄存器

约束修饰符

- `=` 约束修饰符：意味着对于这条指令，操作数为只写的，旧值会被忽略并被输出数据所替换
- `&` 约束修饰符：意味着这个操作数为一个早期改动的操作数，其在该指令完成前通过使用输入操作数被修改了. ?不懂

  在汇编程序模板中，每个操作数用数字引用。编号方式如下。如果总共有 n 个操作数（包括输入和输出操作数），那么第一个输出操作数编号为 0，逐项递增，并且最后一个输入操作数编号为 n - 1 。操作数的最大数目有限制

输出操作数的执行表达式必须作为左值

```
asm ("leal (%1,%1,4), %0"
    :"=r"(five_times_x)
    :"r"(x)
    );
#计算一个数的5倍
```

```
__asm__ __volatile__("lock       ;\n"
                     "addl %1,%0 ;\n"
                    :"=m"(my_var) #位于内存的输出
                    :"ir"(my_int), "m"(my_var)   #my_int是一个整形数
                    : /* 无修饰寄存器列表 */
                    );
#原子加法
```

```
__asm__ __volatile__("decl %0; sete %1"
                     :"=m"(my_var), "=q"(cond)
                     :"m"(my_var)
                     :"memory"
                     );
```

这里将 my_var 的值减 1，并且如果结果的值为 0，则变量 cond 置 1。
my_var 是一个存储于内存的变量
cond 位于寄存器 eax、ebx、ecx、edx 中的任何一个，约束 =q 保证了这一点
memory 位于修饰寄存器列表中，也就是说，代码将改变内存中的内容

```
testq %rax, %rax  ; 两个寄存器中的值按位与，结果影响ZF
sete  %r12        ; 若%rax和%rbx相等，即ZF=1，则将%r12设为1，反之设为0
```

```
        .file   "hello.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text /*声明代码的开始*/
LC0:
        .ascii "Hello, world!\12\0" /*声明一个标签，将原始ascii文本放入程序中*/
.globl _main /*告诉汇编器_main是全局标签，允许程序的其它部分看到他，连接器需要看他_main,因为这是程序的入口。*/
        .def    _main;  .scl    2;      .type   32;     .endef
_main: /*定义了_main标签，*/
        pushl   %ebp #将ebp的值保存到堆栈中
        movl    %esp, %ebp #将esp的值移入ebp中
        subl    $8, %esp #从esp减去8
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .def    _printf;        .scl    2;      .type   32;     .endef

.file .def .ascii是汇编指令 ——告诉汇编器如何汇编文件的命令
以文本开头并跟冒号的行是代码中的标签或命名位置
```

```
.section .data
```

汇编程序中以.开头的名称并不是指令的助记符，不会被翻译成机器指令，而是给汇编器一些特殊指示，称为汇编指示（Assembler Directive）或伪操作（Pseudo-operation），由于它不是真正的指令所以加个“伪”字。.section指示把代码划分成若干个段（Section），程序被操作系统加载执行时，每个段被加载到不同的地址，操作系统对不同的页面设置不同的读、写、执行权限。.data段保存程序的数据，是可读可写的，相当于C程序的全局变量

指示符	作用
.text	代码段，之后跟的符号都在.text内
.data	数据段，之后跟的符号都在.data内
.bss	未初始化数据段，之后跟的符号都在.bss中
.section .foo	自定义段，之后跟的符号都在.foo段中，.foo段名可以做修改
.align n	按2的n次幂字节对齐
.balign n	按n字节对齐
.globl sym	声明sym未全局符号，其它文件可以访问
.string “str”	将字符串str放入内存
.byte b1,...,bn	在内存中连续存储n个单字节
.half w1,...,wn	在内存中连续存储n个半字(2字节)
.word w1,...,wn	在内存中连续存储n个字(4字节)
.dword w1,...,wn	在内存中连续存储n个双字(8字节)
.float f1,...,fn	在内存中连续存储n个单精度浮点数
.double d1,...,dn	在内存中连续存储n个双精度浮点数
.option rvc	使用压缩指令(risc-v c)
.option norvc	不压缩指令