汇编语言程序设计

陶治江

四川大学电气信息学院

gcc 编译器:
C 语言：
vi test.c
gcc -o test test.c
./test
（不用进行连接，而且不用改变生成的可执行代码的执行位）
gcc –S test.c 会生成 C 语言的汇编代码，默认生成的文件名是：test.s
使用目标文件生成汇编代码的方法：
gcc –c test.c //默认生成 test.o；-c 表示编译或者汇编代码而不进行连接，生成目标文件
objdump –d test.o //生成 test.s 代码
汇编：

```
.section .data
output:
    .asciz "Now my age is %d \n"
age:
    .int 23
.section .text
.global _start
_start:
nop
pushl age
pushl $output
call printf
add $8,%esp
pushl $0
call exit
```

编译：as -o test.o test.s
连接：ld -dynamic-linker /lib/ld-linux.so.2 -o test -lc test.o
//使用了标准 C 语言库函数
执行：./test
汇编语言的调试：
gdb 工具，若要调试，在编译的时候就需要添加-gstabs 选项，生成调试的信息，这个编译的结果要大的多：
as –gstabs –o testo test.c
ld -dynamic-linker /lib/ld-linux.so.2 -o test -lc test.o
gdb test
关于 gdb 的命令：
run 运行程序
break *_start+1 设置断点
next 单步执行
cont continue 程序正常执行
info registers 查看所有寄存器的信息
print 查看具体的某个寄存器的信息 print/x $eax x 十六进制 t 二进制 d 十进制
x 查看具体内寸处的信息 x/42cb &output 数字是要显示的字段数 &内存地址

c 字符 d 十进制 x 十六进制 字段的长度：b 字节 h16 为半字节 w32 位字
数据段：
.data .rodata 定义只读数据段，修改后会发生段错误
常用数据类型：.ascii .asciz(末尾有空字节) .byte .double .float .int .octa（八字整数，16 个字节） .quad(四字整数，八个字节) .short
定义数组：
number:
.int 23,34,45
movl number+4,%eax //然后 number+4 引用的就是第二个元素，以字节为偏移量
对于其他的数据类型：
number:
.octa 23,34,45
movl number+4,%eax
此时 number+16 引用的就是 34，因为数据本身是比较小的，所以只引用了四个自己也可以读出数据，其实是不允许的。
定义静态符号：
.equ num,0xa
movl $num,%eax
一般的使用都是作为 4 字节的整数使用的，可以是不同的记数进制
bss 段：.comm buffer,1000 (字节为单位)
.comm 申明未初始化的数据的内存区域
.lcomm 申明未初始化的数据的本地内存区域，不允许本地汇编代码之外进行访问
由于不需要初始化，这部分内存在汇编连接后不用关注，可执行的代码块很小
数据的传送：movx source,destination 传送的目标和地址不能同时是内存
x 为 b 字节 w 字 l 双字
movb %al,%bl movw %ax,%bx movl %eax,%ebx
基于数组的求值，除了像上面的使用地址相加的办法，还可以的用法是：
base_address（offset_address,index,size）
base_address 一般是数组名 offset_address 一般是零，但是只能是寄存器，所以就空着 index 就是元素个数）size（每个元素的字节数）
关于 offset_address，是可以用来访问跨数组名的数组元素的：
data1:
.int 1,2,3
data2:
.int 4,5,6
那么 data1(12,1,4)访问的就是 5，当然，前两个参数必须要放在寄存器中
一个数组的循环访问：
.section .rodata
output:
.asciz "The number now is:%d\n"
num:
.int 12,23,4324,35

.section .text

```
.global _start
_start:
nop
movl $0,%edi
loop:
pushl num(,%edi,4)
pushl $output
call printf
addl $8,%esp
inc %edi
cmpl $4,%edi
jne loop

pushl $0
call exit
```

寄存器间接寻址（使用指针）：
对于一个普通变量 value 他的地址是$value ，当一个寄存器%edi 拥有这个地址时，（%edi）就是这个地址所指向的内存的值，而地址的偏移表示为 4(%edi),-4(%edi)等，而（$value）就是 value，前者是不允许使用的。
条件传送指令：cmovx source,destination
条件取决于 EFLAGS 寄存器的值
对于无符号数，检查的是进位标志，零标志和奇偶校验标志来判断两个数的关系的
对于有符号数，检查符号标志和溢出标志来判断两个数的关系的
无符号数：above below equal(同 zero)carry
有符号数：greater less equal sign(有符号，是负数) overflow
注意：比较指令是使用第二个数来减去第一个数后来设置标记寄存器的，所以比较的时候后一个数不能使立即数：（cmpl %edi,$4 就是不合法的）
应用：比较出数组的最大值

```
.section .data
output:
    .asciz "The max number now is:%d\n"
num:
    .int 12,23,35346,43,2425,346

.section .text
.global _start
_start:
nop
movl $1,%edi
movl num,%eax

loop:
movl num(,%edi,4),%ebx
```

```
cmpl %eax,%ebx
cmovgl %ebx,%eax
inc %edi
cmpl $6,%edi
jnz loop

pushl %eax
pushl $output
call printf
addl $8,%esp
pushl $0
call exit
```

数据交换：

可以不使用第三个临时交换寄存器条件下在寄存器之间或者寄存器和内存之间交换数据

xchg：xchg operand1 operand2; 操作数的长度必须相等，当使用在寄存器和内存之间交换数据的时候，处理器 LOCK 信号被标识，防止其他过程的改动，这个过程比较耗时

bswap 用于反转寄存器中的字节顺序（非比特顺序），使小尾数与大尾数之间的数进行转换，操作数只能在寄存器中（似乎只用于 32 位的寄存器）。

xadd source,destination ，将 source destination 的值相交换，然后将两者相加后将结果保存在 destination 中，其 source 必须是寄存器，xadd 可以添加后缀，也可以没有后缀，寄存器可以是 16，32 位

冒泡法排序的源代码：

```
.section .data
array:
        .int 23,2345,12,5,36,457,453,3252,423,54
.section .text
.global _start
_start:
nop
movl $9,%ecx
movl $9,%ebx
movl $array,%esi
loop:
movl (%esi),%eax
cmpl 4(%esi),%eax
jl skip
xchg %eax,4(%esi)
movl %eax,(%esi)
skip:
addl $4,%esi
dec %ebx
jnz loop
dec %ecx
jz end
```

```
movl %ecx,%ebx
movl $array,%esi
jmp loop
end:
pushl $0                //一定要有，否则程序不能正常结束
call exit
```

堆栈：在堆栈中可以使用 pushl popl 进行四个字节的数据的压入和弹出，使用 pushw popw 可以对两个字节的数据进行操作，但是要注意的是，在压入堆栈进行 prinf 操作的时候，默认是使用一个双字进行输出的，即两个两个字节的元素是一次输出的，当然使用 ESP 指针也可以进行同样的效果的操作：

```
subl $8,%esp
movl $10,4(%esp)
movl $output,(%esp)
```

当然操作完成之后还要还原堆栈：addl $8,%esp

在内存中：数据元素是按照从低内存位置开始，依次向高内存的位置存放的；而堆栈却相反，堆栈被保存在内存的末尾的位置，当在里面存放数据的时候，向下增长，地址不断减少。

pusha/popa 将八个 16 位寄存器压入弹出堆栈　　pushad/popad 将八个 32 位寄存器压入弹出堆栈　　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

跳转：短跳转：使用一个字节作为偏移地址，所以跳转的距离最多就是 128 个字节，远跳转是在分段内存模式下从一个段跳转到另外一个断使用的，近跳转是其他的跳转情况，在汇编指令中，只需要使用 jmp 就可以了，而不用顾及跳转的距离。

调用与跳转的区别是保留了返回地址以便返回，使用 ret 指令返回，在执行 call 指令的时候，它把 EIP 的值放到堆栈中，然后修改 EIP 的值使它对到要调用的函数的地址，调用结束后从堆栈中获得 EIP 原先的值以便返回到原先的地址。在实际的函数的编写的过程中是将 ESP 的值复制到 EIP 寄存器中的，然后使用 EIP 寄存器的值获得在 CALL 指令之前传递给堆栈的信息的，并且可以将本地变量保存在堆栈中。

函数编写的模式：

```
function_label:
pushl %ebp
movl %esp,%ebp
---------
movl %ebp ,%esp
popl %ebp
ret
```

在函数中要注意对堆栈的清理：

但是从函数的调用模式中可以发现，最后总是要使用 movl %ebp ,%esp 进行恢复的，可能不用手工恢复：

```
.section .data
output:
        .asciz "The result is : %d \n"
.section .text
.global _start
_start:
```

```
nop
call function
pushl $0
call exit

function:
pushl %ebp
movl %esp,%ebp
pushl $10
pushl $output
call printf
addl $8,%esp          //用不用？？
movl %ebp,%esp
popl %ebp
ret
```

中断：中断分为软中断和硬中断，硬中断时由硬件发出的信号（如 IO 操作），软中断是由操作系统提供的，可以调用操作系统的核心函数，甚至可以调用底层的 BIOS 层次，在 Linux 中是使用 0x80 调用的。

条件跳转指令：

无符号数：above below equal(同 zero)carry

有符号数：greater less equal sign(有符号，是负数) overflow

多了两个指令：JCXZ JCXNZ 是专门针对 EXC 寄存器的状态进行判断的，看它是否是零

注意的是条件跳转指令在分段的内存模式下是不支持远跳转的，这时就只能使用条件判断后使用无条件跳转指令进行实现了。

常使用的标志位：

1.零标志：可以是比较结果相等，或者是计算的结果是零从而进行置位　jz jnz je jne

2.溢出标志：处理带符号数据 jo jno

3.奇偶校验：用于计数数据中的二进制的 1 的个数，当 1 的个数是奇数时，不置位；当 1 的个数是偶数的时候进行置位。jp　jnp

4.符号标志：当时有符号数时，如果是负数，符号位就被置位：js　jns

符号标志的一个用处就是在处理数组的时候，当使用一般的 jnz jne 指令的时候，当计数到达零的时候就停了，而使用 jns 指令在到达零的时候还可以进行一次的循环，就可以引用下标为 0 的首元素了。

```
.section .data
output:
        .asciz "The result is : %d \n"
data:
        .int 12,24,2543,3425,2645
.section .text
.global _start
_start:
nop
movl $4,%edi
loop1:
```

```
pushl data(,%edi,4)
pushl $output
call printf
addl $8,%esp
dec %edi
jns loop1

pushl $0
call exit
```

5.进位标志：当无符号数发生溢出的时候被置位，但是和溢出标志不同的是：当无符号的数小于零时，进位标志被置位；当使用 DEC INC 自增自减的指令进行操作数据时，即使数据发生溢出或者小于零的时候也不会对进位标志进行置位，即此时:

subl $1,%eax 同 dec %eax 语句是不等价的

关于进位标志可以使用指令进行操作：

clc 清空进位标志　　cmc 对进位标志取反　　stc 置位进位标志

循环：使用 ECX 寄存器进行计数

loop 当 exc 的值为零时停止循环　loopz/loope 循环直到 exc 的值为零或者没有设置 zp 标志 loopnz/loopne 循环直到 exc 的值为零或者设置了 zp 标志

循环指令只支持 8 位的地址偏移量，只能进行 128 字节内的循环，这里的循环的一个好处是可以递减 EXC 的值而不影响 EFLAGS 寄存器的标志位，当 ecx 的值为零时 ZP 也不会因为它而置位。

loop 灾难：由于 loop 只察看 exc 寄存器的值（作为无符号的值进行看待），如果 exc 的值一开始就是零，就会无限递增下去直到寄存器溢出，可以在循环之前进行使用 jcxz 察看。

```
.section .data
output:
        .asciz "The result is : %d \n"
.section .text
.global _start
_start:
nop
movl $0,%ecx
movl $0,%eax
jcxz end
loop1:
addl %ecx,%eax
loop loop1

pushl %eax
pushl $output
call printf
addl $8,%esp
end:
pushl $0
call exit
```

高级语言 for 语句的伪指令：

```
.section .data
value:
        .int 23
output:
        .asciz "The value of ecx register now is:%d\n"
.section .text
.global _start
_start:
nop
movl $0,%ecx
for:
        cmpl $22, %ecx
        jle forcode
        jmp end
forcode:
        pushl %ecx          //一定要局部保存，否则会被破坏
        pushl %ecx
        pushl $output
        call printf
        addl $8,%esp
        popl %ecx
        inc %ecx
        jmp for
end:
        pushl $0
        call exit
```

使用数字：

在内存中数据是按照小尾数的顺序进行排列的，就是低字节的数据放在内存位置较低的位置，其余的字节依次向高内存的位置进行排放，当数据传输到寄存器的时候，数据会自动地转换为大尾数的顺序放在寄存器中的。

扩展数据：当将数据转换为到较大的数据位置时，会有数据扩展，分为有符号扩展和无符号扩展，用于有符号数和无符号数的扩展：movzx ； movsx 如 movsx %cl,%eax

源操作数可以使内存数或者寄存器数，而目标操作数必须是寄存器数，可以依据目标操作数的长度进行正确的扩展。

在 gdb 中察看 8 个字节的数据的时候可使用：x/1gd &data

在输出数据的时候，压入堆栈的时候应该记住内存的数据是小尾数的：

```
pushl data+4        pushl data          pushl $output
.asciz "The value now is:%qd\n"         //如果没有 q 参数的时候就认为是两个数据
```

SIMD 指令：

打包的整数是能够表示多个整数的一系列的字节，可以把这些字节当作一个整体，对它进行数学操作，并行的处理多个整数值。

MMX（多煤体扩展技术）整数

使用 8 个 64 位的 MMX 寄存器（标号 mm0-mm7），在处理器内部被影射为 FPU 寄存器，可

以将 8 个字节数，4 个字整数或者 2 个双字整数放入寄存器中：
movq source,destination
目标和源可以是 MMX ，SSE 或者 64 位的内存位置，但是不能同是内存位置，
使用 print $mm0 查看
SSE（流化 SIMD 扩展技术）整数
使用 8 个 128 位的 XMM(xmm0-xmm7)寄存器，可以将 16 个字节，8 个字，4 个双字，2 个四字的数据传入其中，使用 movdqu 传送：
movdqu source ,destination
目标和源可以是 128 位的 SSE 寄存器或者内存位置（不能同是内存）。
data: .int 1,23,34,2 data2: .quad 2,33 data3: .octa 23
movdqu data,%xmm0 movdqu data2,%xmm1 movdqu data3,%xmm2（但是 otca 不能正常显示，只显示其 16 进制的数据）
BCD 数据：分为打包的和不打包的数据类型，打包的使用一个字节表示两个位数。
FPU 可以用于在 FPU 的内部进行 BCD 的数学操作，FPU 有 8 个 80 位的寄存器，使用低位的 9 个字节储存 BCD 值，格式是打包的 BCD 值，包含 1 8 个数据位，大多数的情况下不使用最到的字节，而将最高字节的最高一个比特用作符号位；为了将打包的数据加载到寄存器中，必须首先在内存中创建打包的数据，然后将数据传送到 FPU 寄存器中之后，它就会被自动转换为扩展双精度的浮点格式，当要从寄存器中获得结果的时候，扩展的浮点数会自动转换为 80 位打包的 BCD 格式。
FPU 寄存器的使用类似于堆栈的操作，使用 fbld 压入寄存器， fbstp 弹出寄存器

```
data:
        .byte 0x89,0x67,0x56,0x34,0x12,0x11,0x22,0x33,0x44
(gdb) print $st0
$1 = 443322111234566789
```

使用 16 进制的数据进行创建（只能 18 位），然后 FPU 自动转换，st0-st7 显示。

```
data:
        .int 0x89675634,0x12112233
```

这样也能显示相应的 BCD 值，但是由于小尾数的转换问题，不能正确显示：

```
(gdb) print $st0
$1 = 1211223389675634
.section .data
data:
        .byte 0x56,0x34,0x12,0x00,0x00,0x00,0x00,0x00,0x00
data2:
        .int 2
.section .text
.global _start
_start:
nop
fbld data
fimul data2
fbstp data
pushl $0
call exit
```

浮点数：
单精度：1+8+23 双精度：1+11+52 扩展双精度：1+15+64
浮点数是在 FPU 寄存器中进行操作的，操作方式类似于堆栈：
单精度：flds fsts 双精度：fldl fstl
查看使用 print $st0-7 命令；对于内存中的浮点数：x/f 单精度 x/gf 双精度
对浮点数据进行压入与弹出的时候，是不能对立即数进行操作的，否则就会发生错误，即使压入的是整数也不可以。
压入常用的预置的数据：
fld1 正 1 fldl2t 以 2 为底的 10 的对数 fldl2e 以 2 为底的 e 的对数 fldpi 圆周率
fldlg2 以 10 为底的 2 的对数 fldln2 以自然数为底的 2 的对数 fldz 正零
SSE 的浮点数据类型：拥有 8 个 128 位的 XMM 寄存器，可以容纳 4 个单精度浮点值或者 2 个双精度的浮点值。
单精度浮点数
movups 将四个单精度的浮点值传送到 XMM 寄存器或者内存中
movss 把一个单精度的浮点值传送到寄存器的低 4 个字节或内存中
movlps 把 2 个单精度的浮点值传送到寄存器的低 8 个字节中或者内存中
movhps 把 2 个单精度的浮点值传送到寄存器的高 8 字节
movlhps 把 2 个单精度的浮点值从低 8 字节传送到高 8 字节
movhlps 把 2 个单精度的浮点值从高 8 字节传送到低 8 字节
双精度浮点数
movupd 把两个双精度的浮点值传送到 XXM 寄存器或者内存中
movsd 把 1 个双精度的浮点值传送到寄存器的底 8 字节或者内存
movhpd 把 1 个双精度的浮点值传送到寄存器的高 8 字节或者内存
movlpd 把 1 个双精度的浮点值传送到寄存器的低 8 字节或者内存
SSE3 技术：
movshdup 传送 128 位，复制了 2，4 元素 DCBA > DDBB
movsldup 传送 128 位，复制 1，2 元素 DCBA > CCAA
movddup 传送 64 位复制 128 位 A > AA
在实际的寄存器的查看中可能显示的不同
寄存器显示的是 ABCD BBDD ABCD AACC

```
data1:
        .float 23.24,123.123
data2:
        .float 11.11,22.22
movss data1,%xmm0
movhps data2,%xmm0
```

v4_float = {23.2399998, 0, 11.1099997, 22.2199993}
可见这里的寄存器的后面是高位，前面是低位。
数学计算：
加法与减法：add sub 他们通用于有符号的和无符号的数据。
加法的操作记住检查 CF OF 标志，对于减法，当计算的结果为负数的时候，如果当作是武无符号的操作时，CF 是被置位的。其实计算机本身的操作是不区分带符号与不带符号的数据的，区分是由程序员本身进行区分的。例如：sub 3,2 虽然结果是-1 这个正确的数据，但是使用 jc 仍然可以进行跳转。neg 对寄存器的数据进行取反操作，速度比用零减要快多

特殊的长数据的加法与减法运算：

adc sbb 可以用于长度很大的数据的加减操作,它把上次计算的借位或者进位标志自动算在其中了。

```
.section .data
data1:
        .quad 7252051615
data2:
        .quad 5732348928
outputsum:
        .asciz "The sum of the number is:%qd\n"
outputdif:
        .asciz "The dif of the number is:%qd\n"
.section .text
.global _start
_start:
nop
movl data1,%ebx
movl data1+4,%eax
movl data2,%edx
movl data2+4,%ecx
addl %edx,%ebx
adcl %ecx,%eax
pushl %eax
pushl %ebx
pushl $outputsum
call printf
movl data1,%ebx
movl data1+4,%eax
movl data2,%edx
movl data2+4,%ecx
subl %ebx,%edx
sbbl %eax,%ecx
pushl %ecx
pushl %edx
pushl $outputdif
call printf
addl $24,%esp
pushl $0
call exit
```

```
The sum of the number is:12984400543
The dif of the number is:-1519702687
```

自增与自减：inc dec

用于无符号的数据的自增与自减操作，他们不会影响进位标志，当程序输入的是从 0xffffffff 进行递增还是递减时，是被当作是一个很大的无符号的数进行操作的。

乘法与除法：他们是进行有符号与无符号计算的区分的，而且都要在指令的结尾标明长度。
无符号的乘法：mul source
目标是寄存器或者内存的位置，隐藏的操作数是：AL AX EAX 的寄存器中，结果储存的位置是原来的两个操作数的两倍：AX DX:AX EDX:EAX
有符号的乘法：imul
1,imul source 同无符号的乘法 mul
2,imul source,destination 目标可以是 16，32 位的通用寄存器，就是结果可以指定存放在某个寄存器中，但是要注意不要溢出
3,imul multiplier,source,destination
multiplier 必须是一个立即数，这个指令方式用于将一个带符号的立即值同一个数进行快速的相乘并将结果存放在特定的通用寄存器中。
除法：div divisor 有符号：idiv
divisor 可以是寄存器或者一个内存值，被除数在 AX DX:AX EDX:EAX 中，除数的长度只能是被除数的一半，不满足使用 movsx 进行扩充。
计算的结果：商 AL AX EAX 余数 AH DX EDX
对于有符号的除法，余数的符号总是同被除数的符号相同。

```
.section .data
output:
        .asciz "The quotient is:%d\n"
output1:
        .asciz "The remainder is %d\n"
dividend:
        .quad -2352347
divisor:
        .int 23
quotient:
        .int 0
remainder:
        .int 0
.section .text
.global _start
_start:
nop
movl dividend,%eax
movl dividend+4,%edx
idivl divisor
movl %edx,remainder        //这里的两步不是多余的，在除法结束后要尽快保护计算的结果
movl %eax,quotient         //否则就会被破坏而发生错误
pushl quotient
pushl $output1
call printf
pushl remainder
pushl $output
call printf
```

addl $16,%esp
pushl $0
call exit
移位指令：
sal shl 向左移位，由于右端全部都是添零，所以这两个指令是等价的，指令需要添加后缀。
sal destination
sal %cl,destination //只能使用这个寄存器
sal shifter,destination //shifter 是一个立即值
其中的 destination 可以是寄存器或者内存，移位操作从左边出去的数据都被放到了 CF 中了
右移指令：
sar 算数右移，用于有符号的友谊操作 shr 逻辑右移，用于无符号的右移操作
语法同左移，右边溢出的数据放于 CF 中
由于处理器的乘法与除法的操作时很费时的，使用移位操作可以进行一些快速的乘除计算
循环移位：
rol 向左循环移位 ror 向右循环移位 rcl rcr 向左向右循环移位，包含 CF 标志
操作的语法同前面的移位操作
不打包的 BCD 运算：这种数据在内存中创建小尾数的数据后，在正常计算后（一般是计进位的计算方法）后每步对操作的结果进行指令自动的调整
aaa aas aam aad
这些指令能够一般用在 add adc sub sbb mul 等指令之后 aad 指令不同，它是对被除数进行先操作后再进行除法计算。这些调整指令都用到了一个隐含的操作数据：AL 寄存器，假设前面计算的结果都被保留在了 AL 寄存器中，并把值调整为不打包的 BCD 格式，ADD 指令假设被除数以不打包的格式放在 AX 寄存器中的，结果是 AL 中的商和 AH 中的余数，他们都是不打包 BCD 的形式。

```
.section .data
data:
        .byte 0x04,0x03,0x02
data2:
        .byte 0x02,0x03,0x01
.section .bss
.lcomm sum,4
.section .text
.global _start
_start:
nop
xor %edi,%edi
movl $3,%ecx
loop1:
movb data(,%edi,1),%al
adcb data2(,%edi,1),%al
aaa
movb %al,sum(,%edi,1)
inc %edi
loop loop1
```

adcb $0,sum(,%edi,1)　　//捕捉最后的进位标志
pushl $0
call exit
打包的 BCD 运算：DAA　DAS　只用于加减的调整，具体的使用方法同不打包的计算方法
布尔操作：　AND OR XOR NOT 只有 NOT 使用单一的操作数
TEST source destination　相当于 AND 操作但是不改变操作数的本身。
FPU 高级数学功能：
FPU 寄存器堆栈：FPU 是一个自持单元，用与标准寄存器独立的一组寄存器操作数据，有 8 个 80 位的数据寄存器和 3 个 16 位的控制，状态，标志寄存器。FPU 寄存器命令为 R0-R7，和内存的堆栈不同的是这个堆栈是循环的，堆栈的最后一个寄存器连接最先的一个寄存器，当堆栈满时，如果把第九个数据加载到堆栈寄存器中，堆栈指针会绕回到第一个寄存器中，使用新值替换这个值，并引发一个 FPU 异常错误（后来加入的值被显示为 nan 无效值）。
①状态寄存器：fstat 寄存器
使用 fstsw 指令可以将 fstat 寄存器的值加载到内存或者 AX 寄存器中：fstsw %ax fstsw status
默认情况下 fstat 寄存器的所有位都是被置零的。
②控制寄存器：fctrl 寄存器
前 6 位用于设置错误的掩码的，默认的情况下所有的掩码都被置位，即屏蔽所有的异常。
8-9 控制确定浮点的计算精度，00 单精度 01 未使用 10 双精度 11 扩展双精度
默认情况下是使用扩展双精度，但最为耗时
10-11 控制舍入的方法：00 舍入到最近值 01 向下舍入 10 向上舍入 11 向零舍入
默认是向最近值舍入
整个控制寄存器默认的设置值是 0x037f 将它设置为 0x7f 使用单精度的浮点计算，加快浮点的计算速度， 使用 fstcw 将寄存器的内容弹出到内存中
fldcw 将内存中的值加载到控制寄存器中
③标志寄存器:ftag 寄存器
顺序是 R7----R1 最右端的和最左端的零往往省略。
使用 16 个字节表示 8 个数据寄存器的状态，每个寄存器两个字节
00 一个合法的扩展双精度值 01 零值 10 特殊的浮点数（nan 值）11 空，无内容
数据寄存器的堆栈操作：
首先使用 finit 初始化控制寄存器和状态寄存器和标志寄存器为默认的值，但不改变数据寄存器的数据操作的结果是将数据放入栈低，然后将 ftag 标志为 0xffff，数据是不可以使用的，等于是全部清除了，但是会遗留一些无效的数据。
整数 filds fists　　单精度 flds fsts　　双精度 fldl　fstl
其他指令：fst %st(4)将 st(0)的数据复制到 st(4)中　fxch %st(4)交换寄存器 st(1)和 st(4)
fstp 同 fst 一样复制数据，但是复制完 st(0)的数据之后就把 st(0)的数据弹出
fstp %st(4)是将 st(0)的数据复制到 st(4)中的，然后将 st(0)的数值弹出，实际数据在 st(3)中了
弹出的数据正如是循环的堆栈，放在寄存器的高位，但是标志寄存器标志为无效的数据。
浮点的基本运算：
fadd　fdiv　fdivr　fmul　fsub　fsubr　带有 r 标志的是进行反序操作
另外在指令中添加一个 i 用于整数操作，结尾添加一个 p 字将结果弹出堆栈
关于每个指令的扩充：
fadd source 将内存中的值与 st(0)中的数据进行相加(fadd source ,%st(0))
fadd %st(x),%st(0)
fadd %st(0),%st(x)

faddp %st(0),%st(x)相加的结果储存在 st(x) 中，并弹出 st(0)
faddp （faddp %st(1)%,st(0)）
fiadd source (source 中是 16 或者 32 位的内存整数值)
在使用到内存值得时候要使用 sl 后缀
高级浮点运算指令：
fabs 计算 st0 绝对值 fchs 改变 st0 中值得符号 fcos 计算 st0 的余弦值
fpatan 计算 st0 的部分反正切 fprem 计算 st0/st1 的部分余数
fprem1 计算 st0/st1 的部分余数（IEEE 格式的） fptan 计算 st0 的部分正切
frndint 对 st0 最近取整（取整的方向由 fctrl 控制） fscale 计算 st0 乘以 2 的 st1 次方 fsin 计算 st0 的正弦 fsincos 计算 st0 的正弦和余弦，正弦在 st1，余弦在 st0 fsqrt fyl2x 计算 st1*logst0 （以 2 为底数） fyl2xp1 计算 st1*log（st0+1）（以 2 为底数）
fprem 是 Intel 开发出来的指令，默认是向零舍入的方法 fprem1 使用的是 IEEE 的格式，是向上舍入的方法 这两个指令使用的都是 FPU 状态寄存器的 C2 位来决定迭带的次数，可以将状态寄存器的值加载到 ax 寄存器中使用 test 来检测 C2 的第 10 位)

```
flds f1
fldl d1
loop:
fprem1
fstsw %ax
testb $4,%ah
jnz loop
fsts result
```

三角函数：默认使用 st0 操作数，结果存储在 st0 中，数据使用的是弧度值，如果是角度值就需要自己进行换算：

```
finit
filds val180
fldpi
fdivp
fmull data
fsin
```

fptan 计算 st0 的正切值，并把值放在 st0 后，再压入 1，原来的值就在 st1 中了，这样滞后就可以使用 fdivp 就可以计算正切的倒数了。
fpatan 计算 st1/st0 的反正切，结果在 st1 中，然后弹出 st0，结果保存在 st0 中，返回的结果是弧度值
对数操作：
fyl2x 计算 st1*logst0 （以 2 为底数） fyl2xp1 计算 st1*log（st0+1）（以 2 为底数）
公式是以 2 位底数的，实际可以用公式将一般的底数转为以 2 为底的底数

```
fld1
fldl data1
fyl2x
fld1
fdivp
fldl data
```

fyl2x
浮点条件分支：
fcom 比较 st0 st1 寄存器　　fcom st(x)比较 st0 同其他的寄存器
fcom source 比较 st0 与内存内的一个 32 位或者 64 位值
fcomp　fcomp st(x)　fcomp source　　fcompp 比较 st0 st1 并两次弹出堆栈
ftst 比较 st0 同 0.0
fcom st1 st0　默认的格式
比较之后可以进行 fstat 状态寄存器的设置，通常将其映射为普通的 EFLAGS 寄存器，使用普通的跳转指令等进行跳转
fldl data
fldl data1
fcom
fstsw %ax
sahf
ja end
记住：这里使用的是 a b e 等比较的方法
sahf 指令很关键，因为它将状态寄存器的值（此时在 AX 寄存器中）映射到 EFLAGS 中，只设置 CF PF ZF SF 和对准标志而不影响其他的标志
另外，由于浮点数表示是由一定的误差的，所以不建议用这个方法比较两个浮点数的相等
fcomi fcomip 是将 fcom sahf 合并的指令，比较之后就可以直接使用 abe 等指令了
但是他们的缺陷是只能用于两个寄存器之间的比较
条件传送指令：
fcmov(n)b(e) source ,%st(0)　　fcmov(n)a(e) source ,%st(0)用于将另外一个寄存器的值传送到 st0 寄存器，经常使用在 fcomi 指令之后
保护和恢复 FPU 状态
由于 MMX 技术使用的是 FPU 寄存器的映射，所以在同时使用打包的数据计算和浮点计算的时候可能会破坏ＦＰＵ寄存器的状态和数据
保护和恢复 FPU 环境：
fstenv　　fldenv
这些指令的操作数是一个２８字节的用于保存状态的内存块，主要保存的是：三个特殊的寄存器，FPU 指令指针的偏移量，ＦＰＵ数据指针，ＦＰＵ最后执行的操作码
保存和恢复ＦＰＵ状态：
fsave　　frstor
他们的操作数是一个１０８字节的内存块，主要保存三个特殊的寄存器，８个数据寄存器。
字符串的处理：
movs 指令，用于将字符串从内存的一个位置传到另外的一个内存位置，他们具体使用
movsb movsw movsl　分别传送１，２，４个字节，不用指明操作数，默认的操作数分别是％ＥＳＩ　％ＥＤＩ寄存器指向的内存地址
加载内存地址：
movl $output,%edi 或者 leal output,%edi，在调试的时候使用 x/s &output 查看字符串，在使用.bss 段时，所有的字节都被默认的初始化为 0，自动的作为字符串结尾的\0
在每次进行转移操作的时候，％ＥＳＩ　％ＥＤＩ的值都会自动变更，具体的方向取决与 EFLAGS 寄存器的 DF 标志位：如果这个位被清零，寄存器的值递增，正向操作；如果 DF 被置位，寄存器的值递减，字符串反向操作，可以使用指令 cld std 进行 DF 的清除与置位，

注意的是当 DF 被置位的时候，如果每次移动的数据是不同的，那每次都会从字符串的结尾记数，就是每次从头开始，而每次转移的字符串大小一样或者进行正向操作没有这个问题。
x/12bc &des
rep 前缀：用于反复前缀之后的一条指令，反复的次数是 ECX 寄存器的次数，单步调试的时候，rep 的指令时当作一步进行处理的，如果是进行反向的传送数据，建议使用 movsb，其他的指令同 rep 前缀似乎不能搭配使用

```
.section .data
src:
        .asciz "Tao Zhijang\n"
.section .bss
        .lcomm des,13
.section .text
.global _start
_start:
nop
leal des,%edi
leal src,%esi
cld
mov $13,%ecx
shrl $2,%ecx
rep movsl
movl $13,%ecx
and $3,%ecx
rep movsb

pushl $des
call printf
addl $4,%esp
pushl $0
call exit
```

其他 rep 前缀指令：repe　repne　repz　repnz
单独的 rep 指令前缀是只关心 ecx 寄存器的次数的，而这些指令不仅关心 ecx 寄存器的数目，而且每次都检查 ZF 标记，用于查找等十分方便
存储和加载字符串：
lods 把内存中的字符串加载到 eax 寄存器中，有（bwl）三个后缀，隐含的操作数是%ESI
stos 将 exc 中的字符串加载到另外的一个内存位置中去，有(bwl)三个后缀，隐含的操作数是%EDI，stos 同 rep 一起使用可以将 eax 中的内容多倍复制到内存中。
字符串函数：用于文本的转换为大写

```
.section .data
src:
        .asciz "Tao Zhijiang is studying in sichuan university!\n"
length:
        .equ len,length-src
.section .bss
```

```
        .lcomm des,len
.section .text
.global _start
_start:
nop
leal des,%edi
leal src,%esi
cld
movl $len,%ecx
loop1:
lodsb
cmpb $'a',%al
jb ok
cmpb $'z',%al
ja ok
subb $0x20,%al
ok:
stosb
loop loop1
pushl $des
call printf
addl $4,%esp
pushl $0
call exit
```

字符串的比较（非字符的比较）

cmps (bwl)进行比较后设置 EFLAGS 寄存器，可以使用一般的跳转指令进行跳转，可以将 cmps 同 repe repne repz repnz 指令进行配合使用，repz repe（相等时继续跳转），repnz repne(不相等时进行跳转)，操作数默认是%ESI,%EDI 寄存器

比较的原则：ASCII 的原则 z>a>Z>A,长的>短的

扫描字符串：scas(bwl) 默认的操作数是%ESI 和%EAX 中的数据，常同 repne 等使用，

完整的字符串比较函数的代码：

```
.section .data
src1:
        .asciz "Tao zhijiang is studying in sichuan university!\n"
length1:
        .equ len1,length1-src1
src2:
        .asciz "Tao Zhijiang is studying in sichuan university!\n"
length2:
        .equ len2,length2-src2
a:
        .asciz "source is greater than destination\n"
b:
        .asciz "source is less than destination\n"
```

```
e:
        .asciz "source is equal to destination\n"
.section .text
.global _start
_start:
nop
leal src1,%esi
leal src2,%edi
movl $len1,%eax
movl $len2,%ecx
cmpl %eax,%ecx
jge skip
xchgl %eax,%ecx
skip:
cld
rep cmpsb
ja greater
jb less
je equal

greater:
pushl $a
jmp end
less:
pushl $b
jmp end
equal:
movl $len2,%eax
cmpl $len1,%eax
ja less
jb greater
pushl $e
end:
call printf
addl $4,%esp
pushl $0
call exit
```

查找字符串全代码：

```
.section .data
src:
        .asciz "Tao zhijiang is studying in sichuan university!\n"
length:
        .equ len,length-src
des:
```

```
        .ascii "s"
notfound:
        .asciz "string is not found!\n"
found:
        .asciz "string is found,the address is:%d!\n"
.section .text
.global _start
_start:
nop
leal src,%edi
leal des,%esi
movl $len,%ecx
cld
lodsb
repnz scasb
subl $len,%ecx
neg %ecx
cmpl $0,%ecx
jnz findit
pushl $notfound
call printf
addl $4,%esp
jmp end
findit:
pushl %ecx
pushl $found
call printf
addl $8,%esp
end:
pushl $0
call exit
```

这个代码用于”\0”查找来确定字符串的长度

函数：（可以使用 XMM FPU 等寄存器的操作）

输入：寄存器，全局变量，堆栈　　输出：寄存器，全局变量的内存中

汇编函数同一般的高等语言函数不同，他们不需要在函数定义之前进行定义或者声明，在函数调用中，函数对寄存器的操作是不确定的，所以要进行 pusha popa 进行保存和恢复

C 语言进行数据的传递约束：

参数使用堆栈进行传递，返回值在 %EAX　%EDX:%EAX　ST(0) 寄存器中

程序堆栈：

函数参数 3	16(%ebp)
函数参数 2	12(%ebp)
函数参数 1	8(%ebp)
返回地址	4(%ebp)
旧的 EBP 值	(%ebp)

局部变量 1	-4(%ebp)
局部变量 2	-8(%ebp)

通常使用 subl $8,%esp 可以为局部变量保存一定的内存空间，这样就可以在函数内部进行 pushl popl 进行堆栈操作而不会破坏局部变量了

使用 call 指令进行函数调用完成之后，要使用 addl $-,%esp 将传入的参数的堆栈进行恢复

使用独立的函数文件：

这样的函数文件同独立的汇编代码文件，不用使用_start 和数据段，而把函数名声明为全局的，这样所有的其他的文件就可以访问这个函数了，函数写完后单独汇编，只要在连接的时候将函数文件添加就可以了，调用的过程没有区别

但是在调试的时候好象在主函数的输出都发生段错误

Linux 系统的程序堆栈分析：

环境变量，命令行参数
指向环境变量的指针
0x00000000
指向命令行参数 3 的指针
指向命令行参数 2 的指针
指向命令行参数 1 的指针
程序名称
参数个数（<ESP）

注意：所有的命令行参数都是字符串，即使看起来象数字，程序名称是第一个命令行参数

转换函数都需要将要转换的字符串的指针放在%EAX 中，结果：

atoi EAX atol()EDX:EAX atof() st(0)

显示命令行参数：

```
.section .data
output1:
        .asciz "There are %d arguments of the program.\n"
output2:
        .asciz "%s\n"
.section .text
.global _start
_start:
movl %esp,%ebp
pushl (%ebp)
pushl $output1
call printf
addl $8,%esp
movl (%ebp),%ecx

loop1:
pushl %ecx
addl $4,%ebp
pushl (%ebp)
pushl $output2
call printf
```

```
addl $8,%esp
popl %ecx
loop loop1

pushl $0
call exit
```

查看系统的环境变量：（环境变量的指针数组是以 NULL 结尾的，当没有其他的命令行参数时，命令行参数指针地址同 ESP 相差 12 个地址），代码如下：

```
.section .data
output:
        .asciz "%s\n"
.section .text
.global _start
_start:
movl %esp,%ebp
addl $12,%ebp

loop1:
cmpl $0,(%ebp)
je end
pushl (%ebp)
pushl $output
call printf
addl $8,%esp
addl $4,%ebp
jmp loop1

end:
pushl $0
call exit
```

命令行参数使用范例：

```
.section .data
output:
        .asciz "The sum is:%d\n"
temp:
        .int 0
.section .text
.global _start
_start:
pushl 8(%esp)
call atoi
addl $4,%esp
movl %eax,temp
pushl 12(%esp)
```

```
call atoi
addl $4,%esp
addl temp,%eax
pushl %eax
pushl $output
call printf
addl $8,%esp
pushl $0
call exit
```

注意：函数调用之后需要自己进行堆栈的清理工作，这是 C 调用的规定

Linux 系统调用：

查看系统调用的编号：cat /usr/include/asm/unistd.h

查看具体的调用函数：man 2 exit　　man 2 brk 等

具体的调用格式：

EAX 具体的调用编号

EBX，ECX，EDX，ESI，EDI 按照函数原形顺序的参数，超过 6 个参数就要使用 EBX 传递保存参数的具体内存块的位置

系统调用的值一般都返回在 EAX 寄存器中

例如对于 write 系统调用：

函数原形：ssize_t write(int fd, const void *buf, size_t count);

调用代码：

```
.section .data
output:
        .asciz "The chars to be write.\n"
length:
        .equ len,length-output
.section .text
.global _start
_start:
movl $4,%eax
movl $1,%ebx
movl $output,%ecx
movl $len,%edx
int $0x80

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

汇编语言中的结构定义：

```
result:
name:
    .asciz “taozhijiang”
age:
    .int 12
```

这样既可以使用 result 引用整个标签，也可以使用 name age 引用单个的元素
strace 工具跟踪系统调用：
重要的参数选项：
strace –c（以时间排序生成报表） –o filename（指定到输出文件中） –e trace=getuid (指定跟踪的系统调用，多个使用逗号分割) ./test（可执行的文件，或者命令，如：id，或者正在执行的任务，使用-p PID）
关于 C 语言的调用在第 3 页 ：man 3 exit
汇编语言和 C 语言交替
汇编库，函数文件；内联汇编技术；直接汇编代码优化
内联汇编代码：
格式：asm(“assembly code”);为了汇编之后能生成格式规范的汇编代码，每句代码之后要\n\t
内联函数能够使用 C 语言的全局变量，变量必须是全局的，内联汇编对寄存器的使用是不确定的，所以在代码的开头一般都有 pusha popa 指令，在编译的时候可能会由于编译器的优化作用而生成不好的优化代码，可以使用 volatile 关键字强制不进行优化， asm volatile(“”);
由于可能 C 的关键字和汇编的关键字冲突，关键字可以写为__asm__ __volatile__
例如：

```
#include<stdio.h>
        int a = 10;
        int b = 20;
        int result;
int main()
{
        __asm__ __volatile__ ("pusha \n\t"
        "movl a,%eax\n\t"
        "movl b,%ebx\n\t"
        "addl %ebx,%eax\n\t"
        "movl %eax,result\n\t"
        "popa"
        );
        printf("The result is %d\n",result);
        return 0;
}
```

对于基本的内联汇编，可以使用全局变量，变量是在 main 函数之外定义的，对于扩展的其它格式的汇编，可以使用函数内的局部变量，但是局部变量必须同寄存器相互关联，无法单独进行引用
扩展 asm 格式：asm(“assembly code”:output locations:input locations:changed registers);
当其中的某项没有时，可以为空，但是两边的冒号必须保留，当最后一项没有时，可以省略冒号
输入与输入：“constraint”(variable)
约束可以是：
abcd 表示是%eax %ax %al 这四类寄存器
SD 表示是：%ESI %SI %EDI %DI 寄存器
r 表示的是任意可用的普通寄存器之一
q 表示是%eax%ebx%ecx%edx 这四个寄存器之一
A 表示的是 64 位的%EDA:%EAX

浮点寄存器：t st0　　u st1　　f 任何可用的浮点寄存器
输出列表中不能使用 f 寄存器，必须使用 t u 进行限制

```
#include<stdio.h>
int main()
{
        float a = 10.2;
        double b = 20.222;
        double result;
        __asm__ __volatile__ (
        "faddp\n\t"
        :"=t"(result)
        :"0"(a),"u"(b)
        );
        printf("The result is %f\n",result);
        return 0;
}
```

在调用的时候，必须保持浮点寄存器堆栈平衡，如果输出值在 st0 ，而 st1 中有数据，就必须在改动的寄存器列表中指明这些寄存器。
m 使用变量的内存地址
对于输入还有特殊的限制：+ 可以读取和写入操作数　　=只能写入操作数
举例：asm (“nop”:”=a”(result):”d”(data1):”c”(data2));
在 asm 的汇编代码中，寄存器的使用要使用%%，为了同占位符进行区别
一些程序已经假定有输入值了，比如字符的操作，在输入中指定了%EDI 后就不用在输出中再指定了，输入列表可以为空，但是，这样的汇编代码必须使用 volatile 关键字，否则程序会认为这个 asm 段没有输入而不需要，从而不进行代码的生成

```
#include<stdio.h>
int main()
{
        int len = 19;
        char *src="Sichuan University";
        char des[30];
        __asm__ __volatile__(
        "rep movsb\n\t"
        :
        :"S"(src),"D"(des),"c"(len)
        );
        printf("%s\n",des);
        return 0;
}
```

占位符：根据内联汇编代码中列出的每个输入值和输入值在列表中的位置，每个值被赋予一个从 0 开始的数字，在汇编代码中可以使用%数字 来代替相应的寄存器，有时候在输入和输出的列表中可能有相同的变量，在输出列表中的占位符可以使用在输入列表中：
“=r”(data1)　“0”(data1)　这样可以减少寄存器的使用数量，但是要注意标号：

```
"fadd %2,%0\n\t"
```

```
        :"=t"(result)
        :"0"(a),"u"(b)
```

关于内联汇编的调试：在 main 函数设置断点，使用 stepi 命令进入 asm 段的单步执行

改变的寄存器列表：在输入输出的积存器列表中如果有了特定的寄存器，就不用在改动列表中进行声明了，当在汇编代码中使用了没有出现在输入输出列表中的寄存器时，必须在改动的寄存器列表中进行声明，而且要使用全称：(:::"%eax","%ebx");这是因为在输入输出寄存器列表中使用了 r 等不确定寄存器时，在声明了改变的寄存器列表后，编译器会避免使用这些声明了的寄存器的。

另外，如果在内联汇编代码中使用了没有作为输入和输出的内存变量时，那它必须被声明为是被破坏的，在改动寄存器列表中使用"memory"告诉编译器这个内存位置是在内联汇编中被改变的。

约束"m"用于标识为内存位置，在汇编代码中照样可以使用占位符引用，不过如果汇编代码中的操作数必须使用寄存器时，这个变量就必须使用寄存器进行限制

内联汇编的跳转问题：使用局部跳转避免标签的重复问题：

使用数字作为标签 0：1： 2：…… 跳转时使用 bf 后缀：jmp 1b

例如：

```
#include<stdio.h>
int main()
{
        int a = 10;
        int b = 20;
        int result;
        __asm__ __volatile__ (
        "cmpl %2,%1\n\t"
        "jg 0f\n\t"
        "xchg %2,%1\n"
        "0:\n\t"
        "movl %1,%0"
        :"=a"(result)
        :"b"(a),"c"(b)
        );
        printf("The larger one is %d\n",result);
        return 0;
}
```

宏函数：C #define NAME(input values,output value) (function)

例如：

```
#include<stdio.h>
#define SUM(a,b,result) \
        ((result) = (a) + (b))
#定义使用括号防止参数被错误地展开
int main()
{
        int a = 1,b = 2;
        int result;
        SUM(a+b,b,result);  //同宏参数在不同作用域，而且在定义宏的时候没有参数检查
```

```
        printf("The sum of a and b is:%d\n",result);
        return 0;
}
```

宏函数是没有数据类型检查的，另外，宏的变量独立于函数中的变量，可以在宏定义中使用任何变量名，要想在编译中查看宏展开，使用-E 选项

汇编：

```
#include<stdio.h>

#define SUM(a,b,result) ({ \                    //每行后面都要有连接符号
        asm("addl %1,%2\n\t" \
        "movl %2,%0" \
        :"=c"(result) \
        :"a"(a),"b"(b)); })

int main()
{
        int a = 1,b = 2;
        int result;
        SUM(a+b,b,result);
        printf("The sum of a and b is:%d\n",result);
        return 0;
}
```

汇编库：汇编函数必须遵守 C 语言的调用格式，就是输入变量的传递在堆栈中，输出的结果一般在%EAX 寄存器中

在汇编函数中必须保留的寄存器：

%EBX 指向全局偏移表　　%EBP 在 C 中用于保存堆栈基址指针

%ESP 在 C 中用于指向新堆栈位置　　%EDI 和%ESI：在 C 程序中被用作局部寄存器

所以 C 调用的汇编函数基本格式为：

```
.section .text
.type func, @function
func:
pushl %ebp
movl %esp,%ebp
subl $12,%esp
pushl %edi
pushl %esi
pushl %ebx
<函数体>
popl %ebx
popl %esi
popl %esi
movl %ebp,%esp
popl %ebp
ret
```

编译文件：

gcc –o main main.c asm..s
gcc –o main mian.c asm.o　　接源代码文件或者目标文件都可以
注意：在 c 中调用的汇编函数要有括号
函数文件：

```
.section .data
.type fun,@function
.global fun
fun:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%eax
addl 12(%ebp),%eax
popl %ebx
movl %ebp,%esp
popl %ebp
ret
```

调用的主函数：

```
#include<stdio.h>

int main()
{
        int a = 1,b = 2;
        int result = fun(a,b);
        printf("The sum of a and b is:%d\n",result);
        return 0;
}
```

处理返回值：
一般的函数使用%eax 作为返回值，这样就可以直接在调用函数的时候将返回值赋值给一个变量，对与字符串的返回，一般是返回目标字符串的地址在%EAX 中，但是由于 c 程序默认的%eax 返回的是一个数值，所以在调用返回地址的函数时，要在调用的 main 函数调用之前进行声明，告诉 c 程序函数返回的是地址，当然返回的地址可以是字符串的地址，也可以是任意返回值的地址，不过类型一定要对应起来。
字符串函数：

```
.section .data
output:
        .asciz "This is just a string function.\n"
.section .text
.type fun2,@function
.global fun2
fun2:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

```
movl $output,%eax
popl %esi
movl %ebp,%esp
popl %ebp
ret
```

调用主函数：

```
#include<stdio.h>
char *fun2();
int main()
{
        char *result=fun2();
        printf("%s",result);
        return 0;
}
```

浮点返回值：

默认的返回值在 st0 中，调用程序会自动的进行 fpu 堆栈的弹出工作，而且由于在 fpu 中的数据都是按照扩展双精度的格式进行存放的，所以无论输出的是什么格式的浮点格式都能进行正确的转换，在 c 调用中，所有的返回浮点的函数都必须进行声明：

```
float/double function_name(int ,float);
```

浮点函数：

```
.section .data
.type fun,@function
.global fun
fun:
pushl %ebp
movl %esp,%ebp
pushl %ebx
fldl 8(%ebp)
filds 16(%ebp)      //第一个双精度参数是 8 个字节
faddp
popl %ebx
movl %ebp,%esp
popl %ebp
ret
```

调用的主函数：

```
#include<stdio.h>
double fun(double,int);         //必须声明
int main()
{
        int a =1;
        double b = 3.24;
        double result = fun(b,a); //注意传参的顺序
        printf("The sum of a and b is:%f\n",result);
        return 0;
```

}
多个输入值：result = function(i,j)　8(%ebp) =i　12(%ebp)=j
当遇到混合参数时，一定要注意参数的顺序和参数的所占的空间打小，例如：
int function (double,int) 这样的参数，在进行 c 调用的时候，是将参数压入堆栈的，而双精度的浮点是占用 8 个字节的，注意下一个参数引用时的内存位置
静态库：库名的约定： libx.a　x 是所起的名字，而 a 是后缀
创建和添加：ar r 库名 目标文件列表
ar r lib1.a fun2.o 第一次使用创建，后来就是追加
察看　　：ar tv 库名
创建索引，加快调用速度： ranlib 库名
察看索引　：nm –s 库名　显示目标文件和函数的对应关系
使用： gcc –o main main.c libx.a　　//有的函数具体的调用时候有问题
共享库：库名的约定：libx.so
创建：gcc –shared –o libx.so --.o　　如果有多行可以使用\ 进行连接
使用：gcc –o main –L 路径 –l 库名 --.c
gcc -o asm2 asm2.c -L. -l1 参数紧挨着
其中的库名是去掉 lib .so 的 x；路径可以使用.代表当前路径
ldd 可执行文件　察看文件依赖的共享库
加载共享库：
① 设置 LD_LIBRARY_PATH 环境变量，这个只对当前的对话有用，不需要特殊的权限
　export LD_LIBRARY_PATH=” LD_LIBRARY_PATH:.” 使用.分割多个路径
②配置/etc/ld.so.conf 文件，设置之后使用 ldconfig 命令进行重新加载
gcc –gstabs
使用文件：
打开 5　　关闭 6　　读取 3　　写入 4
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
打开文件：
int open(const char *pathname, int flags, mode_t mode);
flags ： 00 01 02 只读，只写，读写　　0100 0200 如果文件不存在就创建文件，如果文件存在就不打开他　　01000 02000 重新覆盖，追加
使用以上的位组合为一个八进制的数
mode ：同 Linux 中的权限
以上都是八进制的数，必须使用 0 开头 $02002　　$0664
文件的操作：

```
.section .data
filename:
        .asciz "output.txt"
string:
        .asciz "I am a student of Sichuan University!\n"
len:
        .equ length,len-string
```

```
.section .bss
        .lcomm filehandle,4
.section .text
.global _start
_start:

#open the file
movl $5,%eax
movl $filename,%ebx
movl $0102,%ecx
movl $0664,%edx
int $0x80
movl %eax,filehandle

#write the file
movl $4,%eax
movl filehandle,%ebx
movl $string,%ecx
movl $length,%edx
int $0x80
#close the file
movl $6,%eax
movl filehandle,%ebx
int $0x80

movl $0,%eax
movl $0,%ebx
int $0x80
```