##
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

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

##
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

##
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。