中断调用打印字符串信息：

ebx寄存器存放文件描述符，如stdout是1

ecx寄存器存放字符串

edx寄存器存放字符串长度

eax寄存器存放调用函数。

使用变址的内存地址：

values：

.int 10,15,20,25,30,35

完成这种操作的方式称为变址内存模式，内存位置由下列因素决定：

1. 基址
2. 添加到基址上的偏移地址
3. 数据元素的长度
4. 确定选择哪个数据元素的变址。

表达格式是：

base\_address(offset\_address, index, size)

获取的数据值位于：

base\_address + offset\_address + index \* size

例如上面的20，可以这样：

movl $2, %edi

movl values(, %edi, 4), %eax

.equ作用：

定义静态符号：

虽然数据段主要用于定义变量数据，但是也可以在这里声明静态数据符号。

.equ命令用于把常量值设置为可以在文本段中是用的符号。

实例：

.equ factor, 3

.equ LINUX\_SYS\_CALL, 0x80

经过设置后，数据符号值不可在程序中改动。

为了应用静态数据元素，必须在标签名称前面使用美元符号（$）

如：把赋值给LINUX\_SYS\_CALL符号的值传送给EAX寄存器。

movl $LINUX\_SYS\_CALL, %eax

.align伪指令：

leal指令：

实例：

leal 7(%edx, %edx, 7), %eax

将%edx + %edx \* 7 +7所指的内存地址放入%eax，并不取该地址中的值。

**test指令：逻辑运算指令**

功能：执行BIT与BIT之间的逻辑运算，test指令对两个参数执行AND逻辑操作，并根据结果设置标志寄存器，结果本身并不会被保存。

影响标志：C,O,P,Z,S（其中C与O两个标志会被设为0）

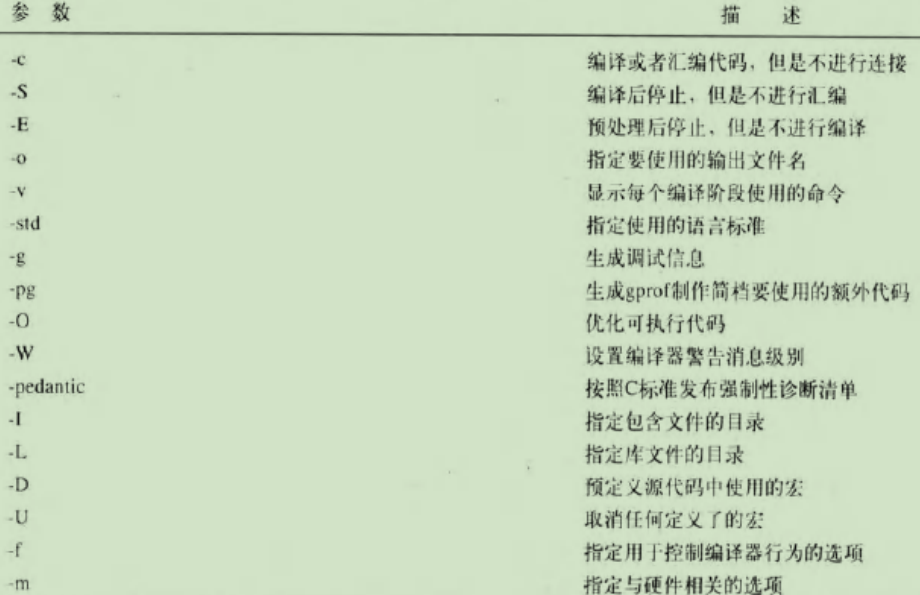
比如测试返回值：test %eax, %eax //如果EAX寄存器

测试某一位是否为零：test %eax, 100b //b后缀表示为二进制

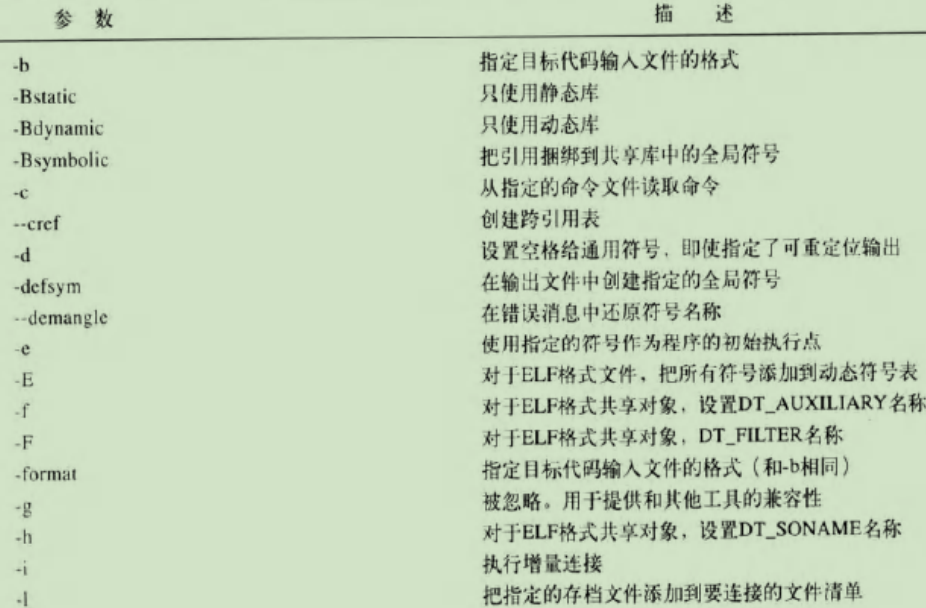
**CMP指令：算术运算指令**

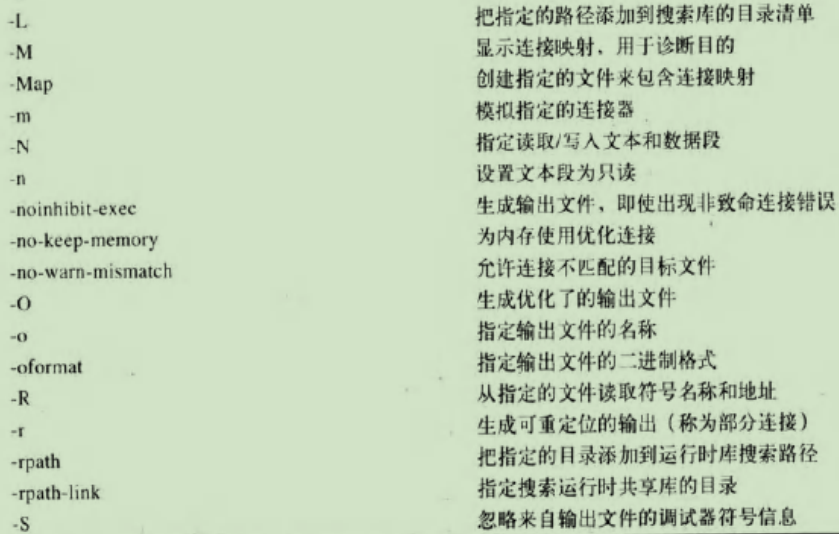
cmp两个操作数作减法，仅修改标志位，不回送结果

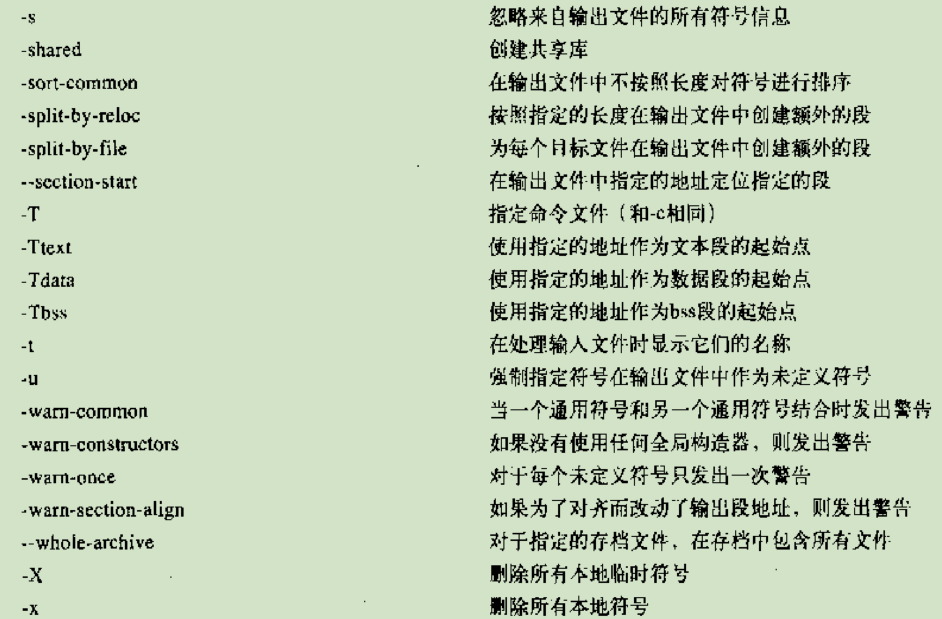
gcc参数：



ld连接器参数：

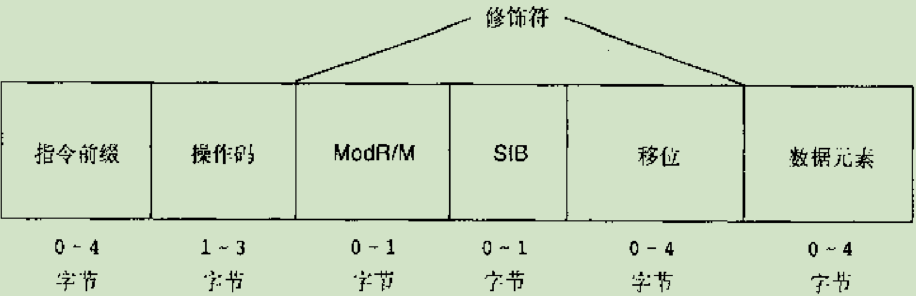






IA-32 指令码格式由四个主要部分构成：

1. 可选的指令前缀
2. 操作码（opcode）
3. 可选的修饰符
4. 可选的数据元素



IA-32指令码格式中唯一必须的部分是操作码。操作码的长度在1-3个字节，他唯一定义要执行的功能。

指令前缀可以包含1到4个修改操作码行为的1字节前缀。按照前缀的功能，这些前缀被分为4个组。修改操作码时，每个组的前缀一次只能使用一个，这四个前缀组如下：

1. 锁定前缀和重复前缀。
2. 段覆盖前缀和分支提示前缀
3. 操作数长度覆盖前缀
4. 地址长度覆盖前缀

锁定前缀表示指令将独占地使用共享内存区域。这对于多处理器和超线程系统非常重要。

重复前缀用于表示重复的功能。

段覆盖前缀定义可以覆盖定义了的段寄存器值得指令。分支前缀尝试向处理器提供程序在条件跳转语句中最可能的路径线索。

操作数长度覆盖前缀通知处理器，程序将在这个操作码之内切换16位和32位的操作数长度。这使程序可以在使用大长度的操作数时警告处理器，帮助加快对寄存器的数据赋值。

地址长度覆盖前缀通知处理器，程序将切换16位和32位的内存地址。这两种长度都可以被声明为程序的默认长度，这个前缀通知处理器程序将切换到另一种长度。

修饰符：

一些操作码需要另外的修饰符来定义执行的功能中涉及到什么寄存器和内存位置。修饰符包含在3个单独的值中：

1. 寻址方式说明符（ModR/M）字节
2. 比例-索引-基址（SIB）字节
3. 1、2或4个的地址移位字节

存储数据和索引数据的方法：

汇编语言允许声明指向内存中特定位置的变量，定义变量包括两个部分：

1. 指向一个内存位置的标记
2. 内存字节的数据类型和默认值

实例：

testvale:

.long 150

message:

.ascii “This is a test message”

pi:

.float 3.14159

汇编语言至少具有3个必须声明的段落：

1. 数据段
2. bss段
3. 文本段

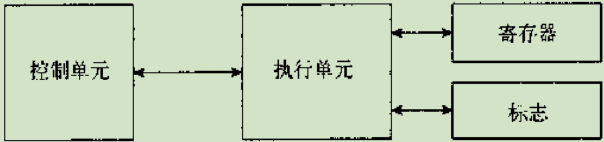
处理器：

1. 总线：控制总线，地址总线，数据总线

处理器的主要组件：

1. 控制单元
2. 执行单元
3. 寄存器
4. 标志

组件以及他们如何在处理器内进行交互

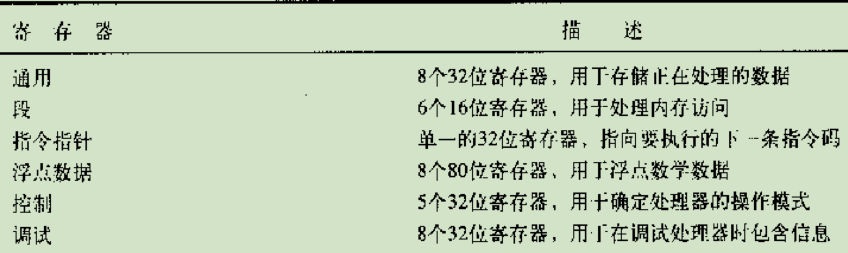


寄存器：

处理器的大多操作都必须处理数据，但是处理器可能采取的最慢的操作时试图读取内存中的数据或者把数据存储到内存中，当处理器访问数据元素时，请求必须被发送到处理器外部、通过控制总线，然后进入内存存储单元。这一过程不近复杂，而且在执行内存访问时迫使处理器处于等待状态，这一停机时间可以用于处理其他指令。

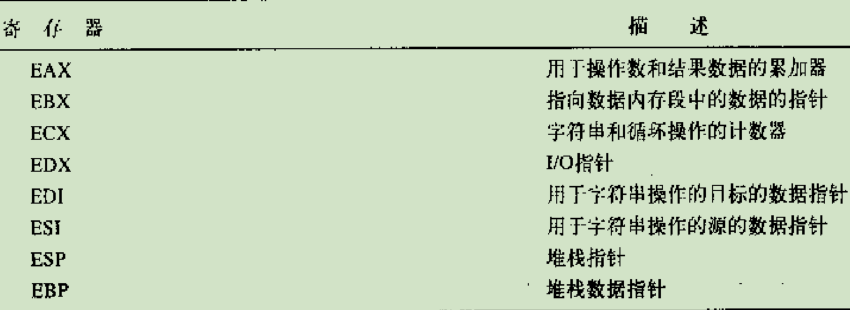
为了解决这个问题，处理器包含内部的存储位置，称为寄存器。寄存器能够存储要处理的数据元素，而无需访问内存存储单元。

**IA-32系统寄存器分组：**



1. 通用寄存器

当处理器处理数据时，通用寄存器用于临时地存储数据。通用寄存器索然大多数都可以用于存储任何类型的数据，但是其中一些有专门的用途，他们在汇编语言程序中以一致的方式使用。



1. 段寄存器

段寄存器专门用于引用内存位置。IA-32处理器平台允许3种不同的访问系统内存的方法：

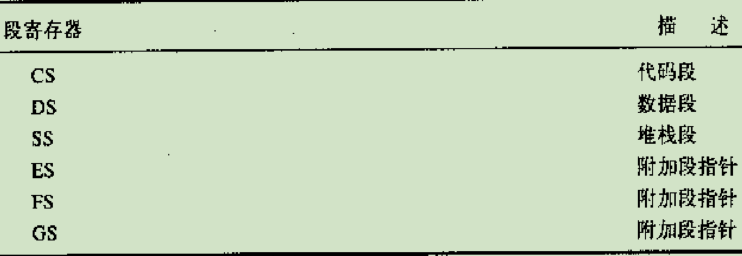
平坦内存模式：把全部系统内存表示为连续的地址空间。所有的指令、数据和堆栈都包含在相同的地址空间中。通过称为线性地址的特定地址访问每个内存位置。

分段内存模式：把系统内存划分为独立段的组，通过位于段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。

段中的内存位置是通过逻辑地址定义的。逻辑地址由段地址和偏移地址构成。处理器把逻辑地址转换为相应的线性地址位置以便访问内存的字节。

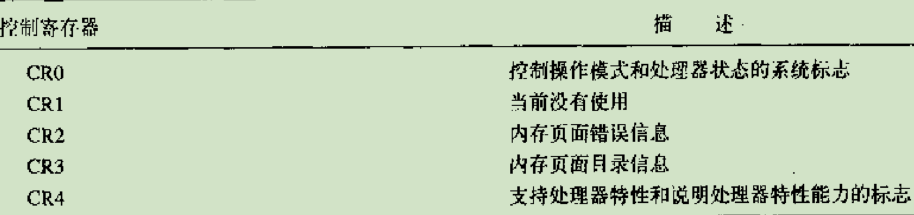
实地址模式

段寄存器用于包含特定数据访问的段地址。可用的段寄存器：



控制寄存器：

5个控制寄存器用于确定处理器的操作模式，还有当前正在执行的任务的特性。



**定义段：**

GNU汇编器使用.section命令语句声明段。.section语句只使用一个参数--他声明的段的类型。

汇编语言程序的安排段的一般方式，bss段总是应该安排在文本段之前。但是数据段可以移动到文本段之后，虽然是不怎么标准。

**定义起始点：**

\_start标签。连接器的-e参数定义新的起始点名称。

.globl命令声明外部程序可以访问的程序标签。

数据类型定义：

|  |  |
| --- | --- |
| 命令 | 数据类型 |
| .ascii | 文本字符串 |
| .asciz | 以空字符结尾的文本字符串 |
| .byte | 字节值 |
| .double | 双精度浮点数 |
| .float | 单精度浮点数 |
| .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） |

.fill n 使汇编器自动创建n个数据元素，默认为每个字段创建一个字节，并且是用0填充他。

**定义静态符号：**

.equ 命令用于把常量值设置为可以在文本段中使用的符号，设置方法如下：

.equ factor, 3

.equ LINUX\_SYS\_CALL, 0x80

经过设置之后，数据符号值是不能在程序中改动的。.equ可以出现在数据段中任何位置。

为了引用静态数据元素，必须在标签名称前面使用美元符号$。

movl $LINUX\_SYS\_CALL, %eax

**bss段：**

bss段无需声明数据类型，只需要声明为所需目的保留的原始内存部分即可。

GNU汇编器使用两个命令声明缓冲区：

|  |  |
| --- | --- |
| 命令 | 描述 |
| .comm | 声明未初始化的数据的通用内存区域 |
| .lcomm | 声明未初始化的数据的本地通用内存区域 |

虽然这两种区域的工作情况类似，但是本地通用内存区域是为不会从本地汇编代码之外进行访问的数据保留的。这两个命令格式是：

.comm symbol, length

其中symbol是赋给内存区域的标签，length是内存区域中的字节数量。

实例：

.lcomm buffer, 10000

把10000字节的内存区域赋值给buffer标签，在声明本地通用内存区域的程序之外的函数是不能访问它们的（不能在.globl命令中使用他们）

**传送数据元素:**

mov指令格式：

movx source, destination

source和destination的值可以是内存地址、存储在内存中的数据值、指令语句中定义的数据值，或者是寄存器。

GNU汇编器为mov指令添加了另一维度，在其中必须声明要传送的数据元素的长度，通过把一个附加字符添加到MOV助记符来声明这个长度，因此，指令就变成了：movx

其中x可以是下面的字符：

l用于32位的长字值

w用于16位的字值

b用于8位的字节值

movs指令：把字符串值从一个内存位置传送给另一个内存位置。

使用mov指令有非常特殊的规则，只有某些位置可以传送给其他位置：

1. 把立即数据元素传送给通用寄存器
2. 把立即数据元素传送给内存位置
3. 把通用寄存器传送给另一个通用寄存器
4. 把通用寄存器传送个段寄存器
5. 把段寄存器传送给通用寄存器
6. 把通用寄存器传送个控制寄存器
7. 把控制寄存器传送给通用寄存器
8. 把通用寄存器传送给传送给调试寄存器
9. 把调试寄存器传递给通用寄存器
10. 把内存位置传递给通用寄存器
11. 把内存位置传递给段寄存器
12. 把通用寄存器传送给内存位置
13. 把段寄存器传送给内存位置

**条件传送：**

旧式汇编：

dec %ecx

jnc continue

movl $0, %ecx

continue:

……

这个代码片段首先使ECX寄存器的值递增1，如果ECX寄存器没有溢出(进位标识没有被设置为1)，JNC命令就跳到continue标签。如果溢出，JNC就会捕捉这个溢出，并且ECX寄存器会被设置回0。

**CMOV指令：**

条件传送指令集中包括了许多指令。格式如下：

cmovx source, destination

其中x是一个或者两个字母的代码，表示将触发传送操作的条件。条件取决于EFLAGS寄存器的当前值。条件传送指令使用特定位在下表介绍：

|  |  |  |
| --- | --- | --- |
| EFLAGS位 | 名称 | 描述 |
| CF | 进位（Carry）标志 | 数学表达式产生进位或者借位 |
| OF | 溢出（Overflow）标志 | 整数值过大或者过小 |
| PF | 奇偶校验（Parity）标志 | 寄存器包含数学操作造成的错误数据 |
| SF | 符号（Sign）标志 | 指出结果为正还是负 |
| ZF | 零（Zero）标志 | 数学操作的结果为零 |

条件传送指令成对的分组再一起，两个指令具有相同的含义。例如，一个值可以大于另一个值，但是也可以说他是不小于或等于另一个值，这两个条件是等同的，但是二者具有各自的条件传送指令。

条件传送指令分为用与于带符号操作的指令和用于无符号操作的指令。带符号操作涉及使用符号标志的比较，而无符号操作涉及忽略符号标志的比较。

**无符号条件传送指令：**

|  |  |  |
| --- | --- | --- |
| 指令对 | 描述 | EFLAGS状态 |
| CMOVA/CMOVNBE | 大于/不小于或等于 | （CF或ZF）=0 |
| CMOVAE/COMVNB | 大于或等于/不小于 | CF=0 |
| CMOVNC | 无进位 | CF=0 |
| CMOVB/CMOVNAE | 小于/不大于或者等于 | CF=1 |
| CMOVC | 进位 | CF=1 |
| CMOVBE/CMOVNA | 小于或者等于/不大于 | （CF或ZF）=1 |
| CMOVE/CMOVZ | 等于/零 | ZF=1 |
| CMOVNE/CMOVNZ | 不等于/不为零 | ZF=0 |
| CMOVP/CMOVPE | 奇偶校验/偶校验 | PF=1 |
| CMOVNP/CMOVPO | 非奇偶校验/奇校验 | PF=0 |

从上表可以看出，无符号传送指令依靠进位、零和奇偶校验标志来确定两个操作数之间的区别。如果操作数是带符号值，就必须使用不同的条件传送指令集，如下：

**有符号条件传送指令：**

|  |  |  |
| --- | --- | --- |
| 指令对 | 描述 | EFLAGS状态 |
| CMOVGE/CMOVNL | 大于或等于/不小于 | （SF异或OF）=0 |
| CMOVL/CMOVNGE | 小于/不大于或等于 | （SF异或OF）=1 |
| CMOVLE/CMOVNG | 小于或等于/不大于 | （（SF异或OF）或ZF）=1 |
| CMOVO | 溢出 | OF=1 |
| CMOVNO | 未溢出 | OF=0 |
| CMOVS | 带符号（负） | SF=1 |
| CMOVNS | 无符号（非负） | SF=0 |

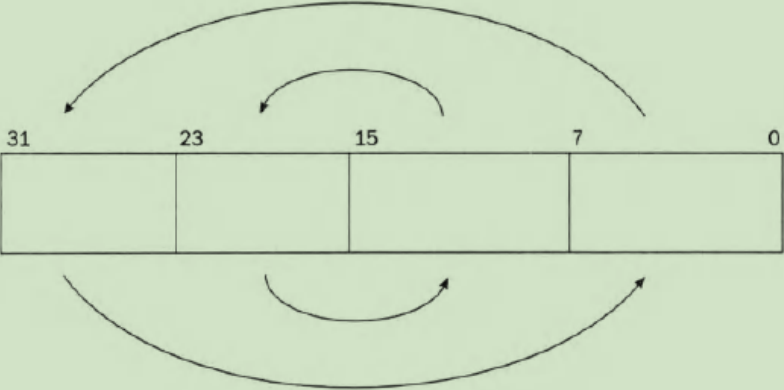
带符号条件传送指令使用符号和溢出标志表示操作数之间比较的状态。

**数据交换指令：**

|  |  |
| --- | --- |
| 指令 | 描述 |
| XCHG | 在两个寄存器之间或者寄存器和内存位置之间交换值 |
| BSWAP | 反转一个32位寄存器中的字节顺序 |
| XADD | 交换两个值并且把总和存储在目标操作数中 |
| CMPXCHG | 把一个值和一个外部值进行比较，并且交换它和另一个值 |
| CMPXCHG8B | 比较两个64位值并且交换它们 |

XCHG在两个通用寄存器之间或者寄存器和内存之间交换数据。(二者不可同时为内存位置)，两个操作数的长度必须相同。当一个操作数是内存位置时，处理器的LOCK会被自动标明，防止在交换过程中被其他处理器访问这个内存位置。LOCK处理非常耗费时间。

BSWAp指令反转寄存器中字节的顺序，32位而言，第0-7位和24-31进行交换，第8-15位和16-23位交换。位的顺序没有被反转，被反转的是寄存器中包含的各个字节。这样就从小端编程了大端。如图：



XADD指令用于交换两个寄存器或者内存位置和寄存器的值。把两个值想加，然后把结果存储在目标位置（寄存器或者内存位置）。格式：

xadd source， destination

其中source必须是寄存器，destination可以是寄存器也可以是内存位置，并且destination包含想加的结果。

CMPXCHG指令比较目标操作数和EAX、AX或者AL寄存器中的值，如果两个值相等，就把源操作数的值加载到目标操作数中，如果两个值不想等，就把目标操作数加载到EAX、AX或者AL寄存器中。格式：

cmpxchg source, destination

**栈：栈指针 SP**

压栈：PUSH

pushx source

其中x是一个字符的代码，表示数据的长度，source是要放入栈中的数据元素。

可进行push操作的数据元素如下：

1. 16位寄存器值
2. 32位寄存器值
3. 16位内存值
4. 32位内存值
5. 16位段寄存器
6. 8位立即数值
7. 16位立即数值
8. 32位立即数值

用于表示数据长度的字符和MOV指令中是一样的格式，但是只能对16位和32位数据值进行PUSH操作：

l 用于长字（32位）

w 用于字（16位）

出栈：

popx destination

destination是接收数据的位置。通过POP指令可以使用下面的数据元素接收数据

1. 16位寄存器
2. 16位段寄存器
3. 32位寄存器
4. 16位内存位置
5. 32位内存位置

压入和弹出所有寄存器：

对于同时快速的设置和获得所有通用寄存器的当前状态，PUSHA和POPA指令非常有用，PUSHA指令压入16位寄存器，使他们按照DI、SI、BP、BX、DX、CX最后是AX的顺序出现在堆栈中。PUSHAD指令按照相同的顺序把这些寄存器对应的32位寄存器压入堆栈。POPA和POPAD指令按照压入寄存器的相反顺序获得寄存器状态。

|  |  |
| --- | --- |
| 指令 | 描述 |
| PUSHA/POPA | 压入或弹出所有16位通用寄存器 |
| PUSHAD/POPAD | 压入或弹出所有32位通用寄存器 |
| PUSHF/POPF | 压入或弹出EFLAGS寄存器的低16位 |
| PUSHFD/POPFD | 压入或弹出EFLAGS寄存器的全部32位 |

**指令指针：**

无条件分支：程序中遇到无条件分支时，指令指针自动转到另一个位置。可以使用的无条件分支有3种：

1. 跳转
2. 调用
3. 终端

跳转：

jmp location

location是要跳转到的内存地址。这个位置被声明为程序代码中的标签。

调用：

调用保存发生跳转的位置，并且具有在需要的时候返回这个位置的能力。在实现函数时使用它。

call address

address操作数引用程序中的标签，它被转换为函数中的第一条指令的内存地址。

调用指令的第二部分是返回指令。它使函数可以返回代码的原始部分，就是紧跟在call指令后面的位置。返回指令没有操作数，只有助记符RET，通过查看堆栈，他知道应该返回什么位置。

中断：

中断时处理器“终端”当前指令码路径并且切换到不同路径的方式，中断有两种：

1. 软件中断
2. 硬件中断

硬件设备生成硬件中断。使用硬件中断发出信号，表示硬件层发生的事情。程序生成软件中断。他们是把控制交给另一个程序的信号。

当一个程序被中断调用时，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程序，并且从被调用的程序内继续运行。被调用的程序完成时，他可以巴控制返回给发出调用的程序（使用中断返回指令）。

软件中断是操作系统提供的，使应用程序可以使用操作系统内的函数。

**条件分支：**

条件分支的结果取决于执行分支时EFLAGS寄存器的状态。

EFLAGS寄存器中有很多位，但是条件分支只和其中的5位有关：

1. 进位（Carry）标志（CF）--第0位（借位有效位）
2. 溢出（Overflow）标志（OF）--第11位
3. 奇偶校验（Parity）标志（PF）--第2位
4. 符号（Sign）标志（SF）--第7位
5. 零（Zero）标志（ZF）--第6位

条件跳转：

条件跳转按照EFLAGS寄存器的当前值来确定是否进行跳转。几种不同的条件跳转指令使用EFLAGS寄存器的不同位。条件跳转指令的格式如下：

jxx address

其中xx是1个到3个字符的条件代码，add是程序要跳转到的位置（通常以标签表示），下表介绍所有可用的条件跳转指令：

|  |  |  |
| --- | --- | --- |
| 指令 | 描述 | EFLAGS |
| JA | 如果大于（above），则跳转 | CF=0与ZF=0 |
| JAE | 如果大于（above）或等于，则跳转 | CF=0 |
| JB | 如果小于（below），则跳转 | CF=1 |
| JBE | 如果小于或等于，则跳转 | CF=1或ZF=1 |
| JC | 如果进位，则跳转 | CF=1 |
| JCXZ | 如果CX寄存器为0，则跳转 |  |
| JECXZ | 如果ECX寄存器为0，则跳转 |  |
| JE | 如果相等，则跳转 | ZF=1 |
| JG | 如果大于（greater），则跳转 | ZF=0与SF=OF |
| JGE | 如果大于或等于，则跳转 | SF=OF |
| JL | 如果小于（less），则跳转 | SF<>OF |
| JLE | 如果小于或等于，则跳转 | ZF=1或SF<>OF |
| JNA | 如果不大于（above），则跳转 | CF=1或ZF=1 |
| JNAE | 如果不大于或等于，则跳转 | CF=1 |
| JNB | 如果不小于（blow），则跳转 | CF=0 |
| JNBE | 如果不小于或等于，则跳转 | CF=0与ZF=0 |
| JNC | 如果无进位，则跳转 | CF=0 |
| JNE | 如果不等于，则跳转 | ZF=0 |
| JNG | 如果不大于，则跳转 | ZF=1或SF<>OF |
| JNGE | 如果不大于或等于，则跳转 | SF<>OF |
| JNL | 如果不小于（less），则跳转 | SF=OF |
| JNLE | 如果不小于或等于，则跳转 | ZF=0与SF=OF |
| JNO | 如果不溢出，则跳转 | OF=0 |
| JNP | 如果不奇偶校验，则跳转 | PF=0 |
| JNS | 如果无符号，则跳转 | SF=0 |
| JNZ | 如果非零，则跳转 | ZF=0 |
| JO | 如果溢出，则跳转 | OF=1 |
| JP | 如果奇偶校验，则跳转 | PF=1 |
| JPE | 如果偶校验，则跳转 | PF=1 |
| JPO | 如果奇校验，则跳转 | PF=0 |
| JS | 如果带符号，则跳转 | SF=1 |
| JZ | 如果为零，则跳转 | ZF=1 |

比较指令：

比较指令比较两个值并且相应的设置EFLAGS寄存器。

cmp operand1, operand2 #opreand2 - operand1

**修改进位标志的指令：**

|  |  |
| --- | --- |
| 指令 | 描述 |
| CLC | 清空进位标志（设置它为零） |
| CMC | 对进位标志求反（把它改变为相反的值） |
| STC | 设置进位标志（设置它为1） |

# 循环：

循环指令：

循环指令使用ECX寄存器作为计数器并且随着指令的执行自动递减他的值。下表介绍循环系列中的指令：

|  |  |
| --- | --- |
| 指令 | 描述 |
| LOOP | 循环直到ECX寄存器为零 |
| LOOPE/LOOPZ | 循环直到ECX寄存器为零，或者没有设置ZF标志 |
| LOOPNE/LOOPNZ | 循环直到ECX寄存器为零，或者设置了ZF标志 |

**整数：**

标准整数长度：

IA-32支持4种不同的整数长度

1. 字节（Byte）：8位
2. 字（Word）：16位
3. 双字（Doubleword）：32位
4. 四字（Quadword）:64位

存储在内存中的超过1个字节的整数被存储为小端格式，但把整数数值传送给寄存器时，值按照大端格式存储在寄存器中。

无符号数：

带符号数：

计算机中描述负数有3种方法：

1. 带符号数值
2. 反码
3. 补码

带符号数值的方法把组成带符号整数的位分为两部分：符号位和数值位。字节的最大有效位（最左侧的一位）用于表示值的符号，正数的最大有效位包含0，而负数的这个位置为1。值中的其余位使用一般的二进制值表示数字的数值。

反码方法采用无符号整数的相反代码生成相应的负值。求反把所有为0的位改变为1，把所有的1改变为0。

补码通过使用简单的数学技巧，解决了带符号数值和反码方法的数学运算问题。对于负数数值，值得反码加上1就是他的补码。

movzx指令把长度小的无符号整数值（可以放在寄存器，也可以放在内存中）传送给长度大的无符号无符号整数值（只能在寄存器中）。

movzx source, destination

source可以是8位或者16位寄存器或者内存地址，destination可以是16位或者32位寄存器。

movsx指令允许扩展带符号证书并且保留符号。

**加法运算：**

add source, destination

其中source可以是立即值、内存位置或者寄存器。destination参数可以是寄存器或者内存位置中存储的值（但是不能同时使用内存位置作为源和目标），加法结果存放在目标位置。

ADD指令可以将8位、16位或者32位值相加，必须通过在ADD助记符的结尾添加b（用于字节）、w（用于字）或者l（用于双字）来指定操作数的长度。

ADC指令执行两个无符号或者带符号整数值的加法，并且把前一个ADD指令产生的进位标志的值包含在其中。

adc source, destination

其中source可以是立即值或者8位、16位或者32位寄存器或者内存位置值，destination可以是8位、16位或者32位寄存器或内存位置值。ADC也需要附加的字符来表名操作数的长度，b、w、l

**减法运算：**

**SUB**

sub source,destination

其中从destination的值中减去source的值，结果存储在destination操作数的位置。

SUB指令也需要在助记符末尾添加长度字符。

**SBB指令在多字节减法操作中利用进位和溢出标志实现跨越数据边界的借位特性。**

SBB指令格式如下：

sbb source, destination

**递增递减：**

INC和DEC指令用于对无符号整数值进行递增和递减操作。

**乘法：**

MUL进行无符号整数乘法。格式：

mul source

source可以是8位、16位、或者32位寄存器或内存值。

乘法的另一个操作数是隐含的。目标位置总是使用EAX寄存器的某种形式，这取决于源操作数的长度。因此，根据操作数的值得长度，乘法操作中使用的另一个操作数必须存放在AL、AX或EAX寄存器中。

由于乘法可能产生很大的值，所以MUL指令的目标位置必须是源操作数的两倍长度。如果源值是8位，那个目标操作数就是AX寄存器，因为结果是16位。

当和16位源操作数相乘时，EAX寄存器不被用于保存32位结果，为了向下兼容老式处理器，Intel使用DX:AX寄存器对保存32位乘法结果值。结果的高位字存储在DX寄存器中，低位字存储在AX寄存器中。

对于32位源值，目标位置使用64位EDX:EAX寄存器对，高位双字存储在EDX及存储器中，低位双字在EAX寄存器中。

|  |  |  |
| --- | --- | --- |
| 源操作数长度 | 目标操作数 | 目标位置 |
| 8位 | AL | AX |
| 16位 | AX | DX:AX |
| 32位 | EAX | EDX:AX |

需要在MUL助记符结尾加上正确的长度字符。

MUL指令只能用于无符号整数

IMUL指令可以用于带符号和无符号整数，但是结果不使用目标的最高有效位。对于较大的值，IMUL指令只对带符号整数是合法的。

IMUL指令有三种不同的指令格式：

第一种与MUL指令完全一样。

imul source

IMUL指令的第二种格式允许指定EAX寄存器之外的目标操作数：

imul source, destination

其中source可以是16位或者32位寄存器或内存中的值。destination必须是16位或者32位通用寄存器，这种格式允许指定把乘法操作的结果存放到哪个位置。

第三种格式：

imul multiplier, source, destination

其中multiplier是一个立即值，source是16位或者32位寄存器或者内存中的值，destination必须是通用寄存器。这种格式允许执行一个值和一个带符号整数的快速乘法操作。IMUL助记符也需要明确指明操作数的长度。

**除法：**

无符号除法：

DIV指令用于无符号整数的除法操作，格式：

div divisor

divisor（除数）是隐含的被除数要除以的值，它可以使8位、16位或32位寄存器或内存中的值。在执行DIV指令之前，被除数必须已经存储到了AX寄存器（对于16位值）、DX:AX寄存器对（对于32位值）或者EDX:EAX寄存器对（对于64位值）。

允许的除数的最大值取决于被除数的长度。对于16位被除数，除数只能是8位；对于32位被除数，除数只能是16位；对于64位被除数，除数只能是32位。

除法操作的结果是两个单独的数字：商和余数。这两个值都被存储在被除数使用的相同寄存器中。如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

也就是说，当除法操作完成后，会丢失被除数，所以要确保被除数是否需要备份。

带符号的除法：

IDIV也使用隐含的被除数，被除数位于AX寄存器、DX:AX寄存器对或者EDX:EAX寄存器中。格式只有一种：

idiv divisor

**移位指令：**

移位乘法：

使整数乘以2的乘方，需要把值向左移位。可以使用两个指令使整数值向左移位：SAL（向左算术移位）和SHL（向左逻辑移位），这两个指令执行相同的操作，并且是可以互换的。他们有3中不同的格式：  
 sal destination

sal %cl, destination

sal shifter, destination

第一种格式把destination的值向左移动1位，相当于乘以2

第二种格式把destination的值向左移动CL寄存器中指定的位数。

第三种格式把destination的值向左移动shifter指定的位数。

需要在助记符结尾附加一个字符，用于指出目标值的长度。

移位除法：

移位除法分为有符号和无符号。

对于无符号整数，向右移位产生的空位可以被填充为0，不会产生影响。对于带符号整数，使用零填充高位部分会对负数产生有害的影响。

SHR指令清空移位造成的空位，所以他只能用于对无符号整数进行移位操作。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，但是对于证书，他们被清空为0。

循环移位：

循环移位指令执行的功能和移位指令一样，只不过溢出位被存放回值得另一端，而不是被丢弃。指令如下：

|  |  |
| --- | --- |
| 指令 | 描述 |
| 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 |

指令格式如下：

1. 单一操作数：按照指定的方向把它移动1位
2. 2个操作数：指定循环次数的%cl寄存器和目标操作数
3. 2个操作数：指定循环次数的立即值和目标操作数

**逻辑操作：**

**布尔逻辑：**

1. AND
2. NOT
3. OR
4. XOR

AND、OR和XOR指令使用相同的格式：

and source, destination

source可以是8位、16位或者32为立即值、寄存器或内存中的值，destination可以是8位、16位或者32为寄存器或内存中的值（不可同时是内存值）

Not指令使用单一操作数，它既是源值，也是目标结果的位置。

布尔逻辑功能和目标执行按位操作，按照顺序对数据元素每个位进行单独比较。

**字符串操作：**

传送字符串：movs指令

movs指令有3种格式：

1. MOVSB：传送单一字节
2. MOVSW：传送一个字（2字节）
3. MOVSL：传送一个双字（4字节）

Intel使用MOVSD传送双字，GNU汇编器使用MOVSL

movs指令使用隐含源和目标操作数。隐含的源操作数是ESI寄存器。它指向源字符串的内存地址。它指向字符串要被复制到的目标内存位置。

LEA指令加载一个对象的有效地址，因为Linux使用32位值引用内存位置，所以对象的内存地址必必须存储在32位的目标值中，源操作数必须指向一个内存位置，比如.data段中使用的标签，指令格式：

leal output, %edi

把output标签的32位内存位置加载到EDI寄存器中。

每次执行MOVS指令时，数据传送后，ESI和EDI寄存器就会自动改变，为另一次传送做准备，ESI和EDI寄存器可能自动地递增，也可能自动的递减，这取决于EFLAGS寄存器中DF的标志。

DF标志被清零，那么每条MOVS指令执行之后EFI和EDI寄存器就会递增，如果DF标志被设置，那么每条MOVS指令执行后ESI和EDI寄存器就会递减。为了确保DF标志被设置为正确的方向，可以使用下面的命令：

1. CLD用于将DF标志清零
2. STD用于设置DF标志

使用STD指令时，ESI和EDI寄存器在每条MOVS指令执行之后递减，所以它们应该指向字符串的末尾，而不是开头。

如果使用STD指令向后处理字符串，MOVSW和MOVSL指令仍然向前获取内存位置。也就是说尽管ESI和EDI寄存器倒着向字符串开始的位置，但是MOVSL和MOVSW依旧在ESI和EDI指向的地方向字符串的尾部走去，每次都将包含之前已经mov过的字符串。

REP前缀：

REP指令的特殊之处在于它自己不执行什么操作，这条指令用于按照特定次数重复执行字符串指令，由ECX寄存器中的值进行控制，和loop循环类似，但是不需要额外的loop指令。REP指令重复地执行紧跟在它后面的字符串指令，直到ECX寄存器中的值为零。

逐块的传送字符串：

如果使用MOVSW和MOVSL指令，ECX寄存器就应该包含遍历字符串所需的迭代次数。例如，如果要传送8字节的字符串，如果是用MOVSB指令，则ECX设置为8，使用MOVSW指令就设置为4，使用MOVSL指令就设置为2。如果超出字符串的边界，则可能会出现问题。

其他REP指令：

除了监视ECX寄存器的值之外，还有监视零标志（ZF）的状态的REP指令。格式：

|  |  |
| --- | --- |
| 指令 | 描述 |
| 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 |

REPE和REPZ指令是相同指令的同义词，REPNE和REPNZ指令是同义词。

LODS指令：用于把内存中的字符串值传送到EAX寄存器中，有3中格式：

1. LODSB：把一个字节加载到AL寄存器中
2. LODSW：把一个字（2字节）加载到AX寄存器中
3. LODSL：把一个双字（4字节）加载到EAX寄存器

Intel文档使用LODSD加载双字，GNU汇编使用LODSL

LODS指令使用ESI寄存器作为隐含的源操作数，ESI寄存器必须包含要加载的字符串所在的内存地址数据传送完成之后，LODS指令按照加载的数据的数量递增或者递减（取决于DF标志状态）ESI寄存器。

STOS和SCAS指令都利用存储在EAX寄存器中的数据，LODS指令把字符串值存放到EAX寄存器中便于这些指令的操作。因为能够加载到EAX寄存器中的数据最多为4个字节，这可以通过单一LODSL指令完成。

STOS指令：

使用LODS指令把字符串放入EAX寄存器之后，可以使用STOS指令把它存放到另一个内存位置中，格式如下：

1. STOSB：存储AL寄存器中的一个字节数据
2. STOSW：存储AX寄存器中的一个字（2字节）的数据
3. STOSL：存储EAX寄存器中一个双字（4个字节）的数据

STOS指令使用EDI寄存器作为隐含的目标操作数。执行STOS指令时，它按照使用的数据长度递增或者递减EDI寄存器的值。

比较字符串：

CMPS指令用于比较字符串的值，格式如下：

1. CMPSB：比较字节值
2. CMPSW：比较字（2字节）值
3. CMPSL：比较双字（4字节）值

隐含的源和目标操作数存储在ESI和EDI寄存器中，根据DF标志的设置，ESI和EDI寄存器按照被比较的数据的长度递增或者递减。

CMPS指令从源字符串中减去目标字符串，并适当的设置EFLAGS寄存器的进位、符号、溢出、零、奇偶校验和辅助进位标志。

CMPS和REP一起使用：

REP指令不在两个重复的过程之间检查标志的状态，它只关心ECX寄存器中的计数值。

可以使用REPE、REPNE、REPZ和REPNZ。这些指令在每次重复过程中检查零标志，如果零标志被设置，就停止重复，可以逐字节的检查字符串以便确定他们是否匹配，只要匹配，REP指令就会停止重复。

扫描字符串：

SCAS指令：用于扫描字符串搜索一个或者多个字符。

1. SCASB：比较内存中的一个字节和AL寄存器的值
2. SCASB：比较内存中的一个字和AX寄存器的值
3. SCASL：比较内存中的一个双字和EAX寄存器的值

SCAS指令使用EDI寄存器作为隐含的目标操作数，EDI寄存器必须包含要扫描的字符串的内存地址。

进行比较时，会相应地设置EFLAGS的辅助进位、进位、奇偶校验、溢出、符号和零标志。可以使用标准的和条件分支指令检查扫面的结果。

REPE和PRPNE指令常常用于在找到搜索字符时停止扫描。

1. REPE：扫描字符串的字符，查找不匹配搜索字符的字符。
2. REPNE：扫描字符串的字符，查找匹配搜索字符的字符。

对于大多数字符串扫描，使用REPNE指令，当找到字符串之后，EDI寄存器包含紧跟在定位到的字符后面的内存地址。这是因为REPNE指令在执行SCAS指令之后递增EDI寄存器。ECX寄存器包含搜索字符距离字符串末尾的位置。使用这个值需谨慎，因为它是从字符串的末尾开始计数的，为了得到距离字符串开头的位置，要从这个值减去字符串长度并且反转符号。

搜索多个字符：

SCASW和SCASL指令可以用于搜索2个或者4个字符序列，查找AX或EAX寄存器中的字符序列。每次比较完成后，EDI寄存器要么递增2（对于SCASW）要么递增4（对于SCASL）

**函数：**

程序堆栈

程序启动时，Linux把4种类型的信息存放到程序堆栈中：

1. 命令行参数（包括程序名称）的数目
2. 从shell提示符执行的程序的名称
3. 命令行中包含的任何命令行参数
4. 在程序启动时的所有当前Linux环境变量

程序名称、命令行参数和环境变量是一空结尾的长度可变的字符串。Linux还把指向每个这些元素的指针加载到堆栈。在命令行参数指针之后，4个字节的空值被存放到堆栈中，用于把参数和指向环境变量的指针的开始位置分隔开来。

分布如下：

ESP

|  |
| --- |
| 环境变量  命令行参数 |
| 指向环境变量的指针 |
| 0x0000000 |
| 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 |
| 参数数目 |

|  |
| --- |
| 函数参数3 |
| 函数参数2 |
| 函数参数1 |
| 返回地址 |
| 旧的EBP值 |
| 局部变量1 |
| 局部变量2 |

8(%ebp)

4(%ebp)

(%ebp)

不指定命令行参数时，环境变量段从ESP寄存器偏移12的位置开始，环境变量段的结束为止被定义为NULL字符串。