1汇编语法格式

绝大多数 Linux 程序员以前只接触过DOS/Windows 下的汇编语言，这些汇编代码都是 Intel 风格的。但在 Unix 和 Linux 系统中，更多采用的还是 AT&T 格式，两者在语法格式上有着很大的不同：

1. 在 AT&T 汇编格式中，寄存器名要加上 '%' 作为前缀；而在 Intel 汇编格式中，寄存器名不需要加前缀。例如：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| pushl %eax | push eax |

1. 在 AT&T 汇编格式中，用 '$' 前缀表示一个立即操作数；而在 Intel 汇编格式中，立即数的表示不用带任何前缀。例如：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| pushl $1 | push 1 |

1. AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；而在 AT&T 汇编格式中，目标操作数在源操作数的右边。例如：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| addl $1, %eax | add eax, 1 |

1. 在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l'分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32比特）；而在 Intel 汇编格式中，操作数的字长是用 "byte ptr" 和 "word ptr" 等前缀来表示的。例如：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| movb val, %al | mov al, byte ptr val |

1. 在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上'\*'作为前缀，而在 Intel 格式中则不需要。
2. 远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为 "ljump" 和 "lcall"，而在 Intel 汇编格式中则为 "jmp far" 和 "call far"，即：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| ljump $section, $offset | jmp far section:offset |
| lcall $section, $offset | call far section:offset |

1. 与之相应的远程返回指令则为：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| lret $stack\_adjust | ret far stack\_adjust |

1. 在 AT&T 汇编格式中，内存操作数的寻址方式是

section:disp(base, index, scale)

而在 Intel 汇编格式中，内存操作数的寻址方式为：

section:[base + index\*scale + disp]

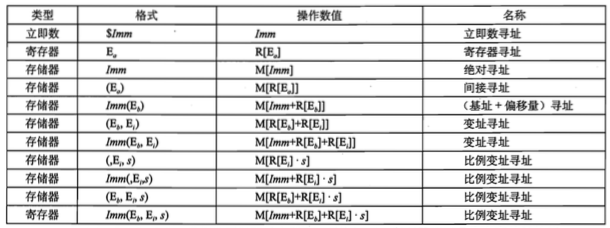
由于 Linux 工作在保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑段基址和偏移量，而是采用如下的地址计算方法：

disp + base + index \* scale

下面是一些内存操作数的例子：

| **AT&T 格式** | **Intel 格式** |
| --- | --- |
| movl -4(%ebp), %eax | mov eax, [ebp - 4] |
| movl array(, %eax, 4), %eax | mov eax, [eax\*4 + array] |
| movw array(%ebx, %eax, 4), %cx | mov cx, [ebx + 4\*eax + array] |
| movb $4, %fs:(%eax) | mov fs:eax, 4 |

## 操作数指示符



**寻址模式**

ADDRESS\_OR\_OFFSET(%BASE\_OR\_OFFSET,%INDEX,MULTIPLIER)

它所表示的地址可以这样计算出来：

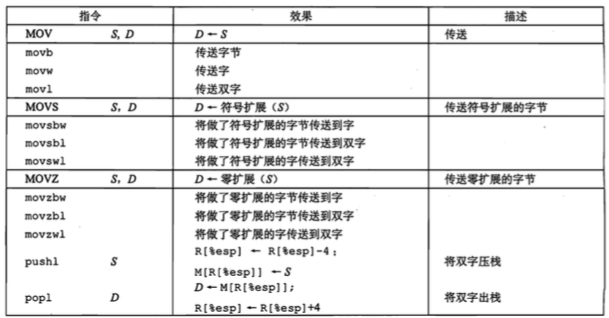
FINAL ADDRESS = ADDRESS\_OR\_OFFSET + BASE\_OR\_OFFSET + MULTIPLIER \* INDEX

其中ADDRESS\_OR\_OFFSET和MULTIPLIER必须是常数，BASE\_OR\_OFFSET和INDEX必须是寄存器。在有些寻址方式中会省略这4项中的某些项，相当于这些项是0。

* 直接寻址（Direct Addressing Mode）。只使用ADDRESS\_OR\_OFFSET寻址，例如movl ADDRESS, %eax 把ADDRESS地址处的32位数传送到eax 寄存器。
* 变址寻址（Indexed Addressing Mode） 。上一节的movl data\_items(,%edi,4), %eax 就属于这种寻址方式，用于访问数组元素比较方便。
* 间接寻址（Indirect Addressing Mode）。只使用BASE\_OR\_OFFSET寻址，例如movl (%eax), %ebx，把eax寄存器的值看作地址，把这个地址处的32位数传送到ebx寄存器。注意和movl %eax, %ebx区分开。
* 基址寻址（Base Pointer Addressing Mode）。只使用ADDRESS\_OR\_OFFSET和BASE\_OR\_OFFSET寻址，例如movl 4(%eax), %ebx ，用于访问结构体成员比较方便，例如一个结构体的基地址保存在eax 寄存器中，其中一个成员在结构体内的偏移量是4字节，要把这个成员读上来就可以用这条指令。
* 立即数寻址（Immediate Mode）。就是指令中有一个操作数是立即数，例如movl $12, %eax 中的$12 ，这其实跟寻址没什么关系，但也算作一种寻址方式。
* 寄存器寻址（Register Addressing Mode）。就是指令中有一个操作数是寄存器，例如movl $12, %eax 中的%eax ，这跟内存寻址没什么关系，但也算作一种寻址方式。在汇编程序中寄存器用助记符来表示，在机器指令中则要用几个Bit表示寄存器的编号，这几个Bit也可以看作寄存器的地址，但是和内存地址不在一个地址空间。

## 数据传送指令

数据传送指令:



常用的三条指令：

* mov S,D
* push S
* pop D

## 算术和逻辑运算指令

算术和逻辑运算指令：

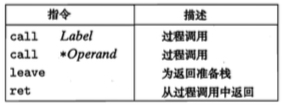


常用的三条指令：

* inc D
* add S,D
* sub S,D

## 过程调用指令

过程调用指令：



常用的三条指令：

* call Lable
* leave
* ret

上面这三条指令每一条都相当于几条指令。

call指令的效果是将返回地址入栈，并跳转到被调用过程的起始处。

leave指令使栈做好返回的准备，等价于下面的代码序列：

mov %rbp,%rsp

pop %rbp

ret指令从栈中弹出地址，并跳转到这个位置。