# Linux 中的汇编语言

在阅读 Linux 源代码时，你可能碰到一些汇编语言片段，有些汇编语言出现在以.S 为扩展名的汇编文件中，在这种文件中，整个程序全部由汇编语言组成。有些汇编命令出现在以.c 为扩展名的 C 文件中，在这种文件中，既有 C 语言，也有汇编语言，我们把出现在 C 代码中的汇编语言叫所“嵌入式”汇编。不管这些汇编代码出现在哪里，它在一定程度上都成为阅读源代码的拦路虎。

尽管 C 语言已经成为编写操作系统的主要语言，但是，在操作系统与硬件打交道的过程中，在需要频繁调用的函数中以及某些特殊的场合中，C 语言显得力不从心，这时，繁琐但又高效的汇编语言必须粉墨登场。因此，在了解一些硬件的基础上，必须对相关的汇编语言知识也所有了解。

读者可能有过在 DOS 操作系统下编写汇编程序的经历,也具备一定的汇编知识。但是，在 Linux 的源代码中，你可能看到了与 Intel 的汇编语言格式不一样的形式，这就是 AT&T 的 386 汇编语言。

一、AT&T 与 Intel 汇编语言的比较

我们知道，Linux 是 Unix 家族的一员，尽管 Linux 的历史不长，但与其相关的很多事情都发源于 Unix。就 Linux 所使用的 386 汇编语言而言，它也是起源于 Unix。Unix 最初是为 PDP－11 开发的，曾先后被移植到 VAX 及 68000 系列的处理器上，这些处理器上的汇编语言都采用的是 AT&T 的指令格式。当 Unix 被移植到 i386 时，自然也就采用了 AT&T 的汇编语言格式，而不是 Intel 的格式。尽管这两种汇编语言在语法上有一定的差异，但所基于的硬件知识是相同的，因此，如果你非常熟悉 Intel 的语法格式，那么你也可以很容易地把它“移植“到 AT&T 来。下面我们通过对照 Intel 与 AT&T 的语法格式，以便于你把过去的知识能很快地“移植”过来。

1．前缀

在Intel的语法中，寄存器和和立即数都没有前缀。但是在AT&T中，寄存器前冠以“%”，而立即数前冠以“$”。在Intel的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在AT&T中，十六进制立即数前冠以“0x”，表2.2给出几个相应的例子。

表2.2 Intel与AT&T前缀的区别

| Intel语法 | AT&T语法 |
|---|---|
| mov eax,8 | movl $8,%eax |
| mov ebx,0ffffh | movl $0xffff,%ebx |
| int 80h | int $0x80 |

2. 操作数的方向

Intel与AT&T操作数的方向正好相反。在Intel语法中，第一个操作数是目的操作数，第二个操作数源操作数。而在AT&T中，第一个数是源操作数，第二个数是目的操作数。由此可以看出，AT&T 的语法符合人们通常的阅读习惯。

例如：在Intel中， mov eax,[ecx]

在AT&T中，movl (%ecx),%eax

3．内存单元操作数

从上面的例子可以看出，内存操作数也有所不同。在Intel的语法中，基寄存器用“[]”括起来，而在AT&T中，用“()”括起来。

例如： 在Intel中，mov eax,[ebx+5]

在AT&T，movl 5(%ebx),%eax

4．间接寻址方式

与Intel的语法比较，AT&T间接寻址方式可能更晦涩难懂一些。Intel的指令格式是

segreg:[base+index*scale+disp]，而 AT&T 的格式是%segreg:disp(base,index,scale)。其中 index/scale/disp/segreg 全部是可选的，完全可以简化掉。如果没有指定 scale 而指定了 index，则 scale 的缺省值为 1。segreg 段寄存器依赖于指令以及应用程序是运行在实模式还是保护模式下，在实模式下，它依赖于指令，而在保护模式下，segreg 是多余的。在 AT&T 中，当立即数用在 scale/disp 中时，不应当在其前冠以“$”前缀，表 2.3 给出其语法及几个相应的例子。

表 2.3 内存操作数的语法及举例

| Intel 语法 | AT&T 语法 |
|---|---|
| 指令 foo,segreg:[base+index*scale+disp] | 指令 %segreg:disp(base,index,scale),foo |
| mov eax,[ebx+20h] | Mov10x20(%ebx),%eax |
| add eax,[ebx+ecx*2h | Add1 (%ebx,%ecx,0x2),%eax |
| lea eax,[ebx+ecx] | Lea1 (%ebx,%ecx),%eax |
| sub eax,[ebx+ecx*4h-20h] | Sub1 -0x20(%ebx,%ecx,0x4),%eax |

从表中可以看出，AT&T 的语法比较晦涩难懂，因为[base+index*scale+disp]一眼就可以看出其含义，而 disp(base,index,scale)则不可能做到这点。

这种寻址方式常常用在访问数据结构数组中某个特定元素内的一个字段，其中，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素还是一个结构，则 disp 为具体字段在结构中的位移。

5．操作码的后缀

在上面的例子中你可能已注意到，在 AT&T 的操作码后面有一个后缀，其含义就是指出操作码的大小。“l”表示长整数（32 位），“w”表示字（16 位），“b”表示字节（8 位）。而在 Intel 的语法中，则要在内存单元操作数的前面加上 byte ptr、 word ptr, 和 dword ptr，“dword”对应“long”。表 2.4 给出几个相应的例子。

表 2.4 操作码的后缀举例

| Intel 语法 | AT&T 语法 |
| --- | --- |
| Mov al,bl | movb %bl,%al |
| Mov ax,bx | movw %bx,%ax |
| Mov eax,ebx | movl %ebx,%eax |
| Mov eax, dword ptr [ebx] | movl (%ebx),%eax |

二、 AT&T 汇编语言的相关知识

在 Linux 源代码中，以.S 为扩展名的文件是“纯”汇编语言的文件。这里，我们结合具体的例子再介绍一些 AT&T 汇编语言的相关知识。

1. GNU 汇编程序 GAS（GNU Assembly 和连接程序

当你编写了一个程序后，就需要对其进行汇编（assembly）和连接。在 Linux 下有两种方式，一种是使用汇编程序 GAS 和连接程序 ld，一种是使用 gcc。我们先来看一下 GAS 和 ld:

GAS 把汇编语言源文件（.o）转换为目标文件（.o），其基本语法如下:

```
as filename.s -o filename.o
```

一旦创建了一个目标文件，就需要把它连接并执行，连接一个目标文件的基本语法为:

```
ld filename.o -o filename
```

这里 filename.o 是目标文件名，而 filename 是输出(可执行) 文件。

GAS 使用的是 AT&T 的语法而不是 Intel 的语法，这就再次说明了 AT&T 语法是 Unix 世界的标准，你必须熟悉它。

如果要使用 GNC 的 C 编译器 gcc，就可以一步完成汇编和连接，例如:

```
gcc -o example example.S
```

这里，example.S 是你的汇编程序，输出文件（可执行文件）名为 example。其中，扩展名必须为大写的 S，这是因为，大写的 S 可以使 gcc 自动识别汇编程序中的 C 预处理命令，像#include、#define、#ifdef、 #endif 等，也就是说，使用 gcc 进行编译，你可以在汇编程序中使用 C 的预处理命令。

2. AT&T 中的节（Section）

在 AT&T 的语法中，一个节由.section 关键词来标识，当你编写汇编语言程序时，至少需要有以下三种节：

.section .data：这种节包含程序已初始化的数据，也就是说，包含具有初值的那些变量，例如：

```
hello      : .string "Hello world!\n"
hello_len : .long 13
```

.section .bss：这个节包含程序还未初始化的数据，也就是说，包含没有初值的那些变量。当操作

系统装入这个程序时将把这些变量都置为 0，例如：

```
name       : .fill 30    # 用来请求用户输入名字
name_len   : .long  0    # 名字的长度（尚未定义）
```

当这个程序被装入时，name 和 name_len 都被置为 0。如果你在.bss 节不小心给一个变量赋了初值，这个值也会丢失，并且变量的值仍为 0。

使用.bss 比使用.data 的优势在于，.bss 节不占用磁盘的空间。在磁盘上，一个长整数就足以存放.bss 节。当程序被装入到内存时，操作系统也只分配给这个节 4 个字节的内存大小。

注意：编译程序把.data 和.bss 在 4 字节上对齐（align），例如，.data 总共有 34 字节，那么编译程序把它对其在 36 字节上，也就是说，实际给它 36 字节的空间。

.section .text ： 这个节包含程序的代码，它是只读节，而.data 和.bss 是读／写节。

3．汇编程序指令（Assembler Directive）

上面介绍的.section 就是汇编程序指令的一种，GNU 汇编程序提供了很多这样的指令（directiv），这种指令都是以句点（.）为开头，后跟指令名（小写字母），在此，我们只介绍在内核源代码中出现的几个指令（以 arch/i386/kernel/head.S 中的代码为例）。

（1）ascii "string"...

.ascii 表示零个或多个（用逗号隔开）字符串，并把每个字符串（结尾不自动加“0“字节）中的字符放在连续的地址单元。

还有一个与.ascii 类似的.asciz，z 代表“0“，即每个字符串结尾自动加一个”0“字节，例如：

```
int_msg:
        .asciz "Unknown interrupt\n"
```

（2）.byte 表达式

.byte 表示零或多个表达式（用逗号隔开），每个表达式被放在下一个字节单元。

（3）.fill 表达式

形式：.fill repeat , size , value

其中，repeat、size 和 value 都是常量表达式。Fill 的含义是反复拷贝 size 个字节。Repeat 可以大于等于 0。size 也可以大于等于 0，但不能超过 8，如果超过 8，也只取 8。把 repeat 个字节以 8 个为一组，每组的最高 4 个字节内容为 0，最低 4 字节内容置为 value。

Size 和 value 为可选项。如果第二个逗号和 value 值不存在，则假定 value 为 0。如果第一个逗号和 size 不存在，则假定 size 为 1。

例如，在 Linux 初始化的过程中，对全局描述符表 GDT 进行设置的最后一句为：

```
.fill NR_CPUS*4,8,0             /* space for TSS's and LDT's */
```

因为每个描述符正好占 8 个字节，因此，.fill 给每个 CPU 留有存放 4 个描述符的位置。

（4）.globl symbol

.globl 使得连接程序（ld）能够看到 symbl。如果你的局部程序中定义了 symbl，那么，与这个局部程序连接的其他局部程序也能存取 symbl，例如：

```
.globl SYMBOL_NAME(idt)

.globl SYMBOL_NAME(gdt)
```

定义 idt 和 gdt 为全局符号。

（5）quad bignums

.quad 表示零个或多个 bignums（用逗号分隔），对于每个 bignum，其缺省值是 8 字节整数。如果 bignum 超过 8 字节，则打印一个警告信息；并只取 bignum 最低 8 字节。

例如，对全局描述符表的填充就用到这个指令：

```
.quad 0x00cf9a000000ffff        /* 0x10 kernel 4GB code at 0x00000000 */
.quad 0x00cf92000000ffff        /* 0x18 kernel 4GB data at 0x00000000 */
.quad 0x00cffa000000ffff        /* 0x23 user   4GB code at 0x00000000 */
.quad 0x00cff2000000ffff        /* 0x2b user   4GB data at 0x00000000 */
```

（6）rept count

把.rept 指令与.endr 指令之间的行重复 count 次，例如

```
    .rept   3
```

```
        .long    0
        .endr
```

相当于

```
        .long    0
        .long    0
        .long    0
```

（7）space size , fill

这个指令保留 size 个字节的空间，每个字节的值为 fill。size 和 fill 都是常量表达式。如果逗号和 fill 被省略，则假定 fill 为 0，例如在 arch/i386/boot1/setup.S 中有一句：

```
.space   1024
```

表示保留 1024 字节的空间，并且每个字节的值为 0。

（8）.word expressions

这个表达式表示任意一节中的一个或多个表达式（用逗号分开），表达式的值占两个字节，例如：

```
gdt_descr:
        .word GDT_ENTRIES*8-1
```

表示变量 gdt_descr 的置为 GDT_ENTRIES*8-1

（9）.long expressions

这与.word 类似

（10）.org new-lc , fill

把当前节的位置计数器提前到 new-lc（new location counter）。new-lc 或者是一个常量表达式，或者是一个与当前子节处于同一节的表达式。也就是说，你不能用.org 横跨

节：如果 new-1c 是个错误的值，则.org 被忽略。.org 只能增加位置计数器的值，或者让其保持不变；但绝不能用.org 来让位置计数器倒退。

注意，位置计数器的起始值是相对于一个节的开始的，而不是子节的开始。当位置计数器被提升后，中间位置的字节被填充值 fill（这也是一个常量表达式）。如果逗号和 fill 都省略，则 fill 的缺省值为 0。

例如：.org 0x2000

ENTRY(pg0)

表示把位置计数器置为 0x2000，这个位置存放的就是临时页表 pg0。

三、 Gcc 嵌入式汇编

在 Linux 的源代码中，有很多 C 语言的函数中嵌入一段汇编语言程序段，这就是 gcc 提供的“asm”功能，例如在 include/asm-i386/system.h 中定义的，读控制寄存器 CR0 的一个宏 read_cr0()：

```
#define read_cr0() ({ \
        unsigned int __dummy; \
        __asm__( \
                "movl %%cr0,%0\n\t" \
                :"=r" (__dummy)); \
        __dummy; \
 })
```

这种形式看起来比较陌生，这是因为这不是标准 C 所定义的形式，而是 gcc 对 C 语言的扩充。其中__dummy 为 C 函数所定义的变量；关键词__asm__表示汇编代码的开始。括

弧中第一个引号中为汇编指令 mov1，紧接着有一个冒号，这种形式阅读起来比较复杂。

一般而言，嵌入式汇编语言片段比单纯的汇编语言代码要复杂得多，因为这里存在怎样分配和使用寄存器，以及把 C 代码中的变量应该存放在哪个寄存器中。为了达到这个目的，就必须对一般的 C 语言进行扩充，增加对编译器的指导作用，因此，嵌入式汇编看起来晦涩而难以读懂。

1. 嵌入式汇编的一般形式：

```
__asm__ __volatile__ ("<asm routine>" : output : input : modify);
```

其中，__asm__表示汇编代码的开始，其后可以跟__volatile__（这是可选项），其含义是避免“asm”指令被删除、移动或组合；然后就是小括弧，括弧中的内容是我们介绍的重点：

- "<asm routine>"为汇编指令部分，例如，"mov1 %%cr0,%0\n\t"。数字前加前缀“％“，如％1，％2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数量，如 Intel 可以有 8 个。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的处理。由于这些样板操作数的前缀使用了”％“，因此，在用到具体的寄存器时就在前面加两个“％”，如%%cr0。
- 输出部分（output），用以规定对输出变量（目标操作数）如何与寄存器结合的约束（constraint）,输出部分可以有多个约束，互相以逗号分开。每个约束以“＝”开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：

  :"=r"（__dummy）

“＝r”表示相应的目标操作数（指令部分的%0）可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

：“＝m”（__dummy）

“＝m”就表示相应的目标操作数是存放在内存单元__dummy 中。

表示约束条件的字母很多，表 2－5 给出几个主要的约束字母及其含义：

表 2.5　主要的约束字母及其含义

| 字母 | 含义 |
| --- | --- |
| m，v,o | 表示内存单元 |
| R | 表示任何通用寄存器 |
| Q | 表示寄存器 eax，ebx，ecx,edx 之一 |
| I，h | 表示直接操作数 |
| E，F | 表示浮点数 |
| G | 表示“任意” |
| a，b.c d | 表示要求使用寄存器 eax/ax/a1，ebx/bx/b1，ecx/cx/c1 或 edx/dx/d1 |
| S，D | 表示要求使用寄存器 esi 或 edi |
| I | 表示常数（0～31） |

· 输入部分（Input）：输入部分与输出部分相似，但没有“＝”。如果输入部分一个操作数所要求使用的寄存器，与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号（如“1”，“2”等）放在约束条件中，在后面的例子中我们会看到这种情况。

· 修改部分（modify）:这部分常常以“memory”为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。

注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，分号“：“要保留，当“memory”存在时，三个分号都要保留，例如 system.h 中的宏定义__c1i()：

```
#define __cli()                    __asm__ __volatile__("cli": : :"memory")
```

2. Linux 源代码中嵌入式汇编举例

Linux 源代码中，在 arch 目录下的.h 和.c 文件中，很多文件都涉及嵌入式汇编，下面以 system.h 中的 C 函数为例，说明嵌入式汇编的应用。

（1）简单应用

```
#define __save_flags(x)          __asm__ __volatile__("pushfl ; popl %0":"=g" (x): /* no input */)
#define __restore_flags(x)       __asm__ __volatile__("pushl %0 ; popfl": /* no output */
                                                      :"g" (x):"memory", "cc")
```

第一个宏是保存标志寄存器的值，第二个宏是恢复标志寄存器的值。第一个宏中的 pushfl 指令是把标志寄存器的值压栈。而 popl 是把栈顶的值（刚压入栈的 flags）弹出到 x 变量中，这个变量可以存放在一个寄存器或内存中。这样，你可以很容易地读懂第二个宏。

(2) 较复杂应用

```
static inline unsigned long get_limit(unsigned long segment)
{
        unsigned long __limit;
        __asm__("lsll %1,%0"
                :"=r" (__limit):"r" (segment));
       return __limit+1;
}
```

这是一个设置段界限的函数，汇编代码段中的输出参数为__limit（即%0），输入参数为 segment（即%1）。Lsll 是加载段界限的指令，即把 segment 段描述符中的段界限字段装入某个寄存器（这个寄存器与__limit 结合），函数返回__limit 加 1，即段长。

（3）复杂应用

在 Linux 内核代码中，有关字符串操作的函数都是通过嵌入式汇编完成的，因为内核及用户程序对字符串函数的调用非常频繁，因此，用汇编代码实现主要是为了提高效率（当然是以牺牲可读性和可维护性为代价的）。在此，我们仅列举一个字符串比较函数 strcmp，其代码在 arch/i386／string.h 中。

```
static inline int strcmp(const char * cs,const char * ct)
{
int d0, d1;
register int __res;
__asm__ __volatile__(
        "1:\tlodsb\n\t"
        "scasb\n\t"
        "jne 2f\n\t"
        "testb %%al,%%al\n\t"
        "jne 1b\n\t"
        "xorl %%eax,%%eax\n\t"
        "jmp 3f\n"
        "2:\tsbbl %%eax,%%eax\n\t"
        "orb $1,%%al\n"
        "3:"
        :"=a" (__res), "=&S" (d0), "=&D" (d1)
                :"1" (cs),"2" (ct));
return __res;
}
```

其中的“\n”是换行符，“\t”是 tab 符，在每条命令的结束加这两个符号，是为

了让 gcc 把嵌入式汇编代码翻译成一般的汇编代码时能够保证换行和留有一定的空格。例如，上面的嵌入式汇编会被翻译成：

```
1:      lodsb           //装入串操作数,即从[esi]传送到 al 寄存器，然后 esi 指向串中下一个元素
        scasb           //扫描串操作数，即从 al 中减去 es:[edi]，不保留结果，只改变标志
        jne2f           //如果两个字符不相等，则转到标号 2
        testb %al  %al
        jne 1b
        xorl %eax %eax
        jmp 3f
2:      sbbl %eax %eax
        orb $1 %al
3:
```

这段代码看起来非常熟悉，读起来也不困难。其中 1f 表示往前（forword）找到第一个标号为 1 的那一行，相应地，1b 表示往后找。其中嵌入式汇编代码中输出和输入部分的结合情况为：

- 返回值__res，放在 al 寄存器中，与%0 相结合；
- 局部变量 d0，与％1 相结合，也与输入部分的 cs 参数相对应，也存放在寄存器 ESI 中，即 ESI 中存放源字符串的起始地址。
- 局部变量 dl，与％2 相结合，也与输入部分的 ct 参数相对应，也存放在寄存器 EDI 中，即 EDI 中存放目的字符串的起始地址。

通过对这段代码的分析我们应当体会到，万变不利其本，嵌入式汇编与一般汇编的区别

仅仅是形式，本质依然不变。因此，全面掌握 Intel 386 汇编指令乃突破阅读低层代码之根本。