arm学习之汇编跳转指令总结

[**GCC 内联汇编**](http://www.cppblog.com/jb8164/archive/2008/02/26/43260.html)

有时为了高效，有时为了直接控制硬件，有些模块我们不得不直接用汇编语言来编写，并且对外提供调用的接口，隐藏细节，这其实就是内联汇编。如何使用内联汇编？我们就以 GCC 为例，一窥其中奥秘！

一、关键字
    如何让 GCC 知道代码中内嵌的汇编呢？ 借助关键字！来看下面的例子：

     \_\_asm\_\_ \_\_volatile\_\_("hlt");

  \_\_asm\_\_ 表示后面的代码为内嵌汇编，asm 是 \_\_asm\_\_ 的别名。\_\_volatile\_\_ 表示编译器不要优化代码，后面的指令保留原样，volatile 是它的别名。括号里面是汇编指令。

二、示例分析
  使用内嵌汇编，要先编写汇编指令模板，然后将 C 语言表达式与指令的操作数相关联，并告诉 GCC 对这些操作有哪些限制条件。示例如下：

   \_\_asm\_\_ \_\_violate\_\_ ("movl %1,%0" : "=r" (result) : "m" (input));

    movl %1,%0  是指令模板；%0  和 %1 代表指令的操作数，称为占位符，内嵌汇编靠它们将C 语言表达式与指令操作数相对应。

    指令模板后面用小括号括起来的是 C 语言表达式，本例中只有两个：result 和 input ，他们按照出现的顺序分别与指令操作数 %0 、%1 对应；注意对应顺序：第一个 C 表达式对应 %0 ；第二个表达式对应 %1 ，依次类推，操作数至多有10 个，分别用 %0, %1 …. %9 表示。

    在每个操作数前面有一个用引号括起来的字符串，字符串的内容是对该操作数的限制或者说要求。result 前面的限制字符串是 =r ，其中 = 表示 result 是输出操作数， r  表示需要将 result 与某个通用寄存器相关联，先将操作数的值读入寄存器，然后在指令中使用相应寄存器，而不是 result 本身，当然指令执行完后需要将寄存器中的值存入变量 result ，从表面上看好像是指令直接对 result 进行操作，实际上 GCC 做了隐式处理，这样我们可以少写一些指令。 input 前面的 r 表示该表达式需要先放入某个寄存器，然后在指令中使用该寄存器参加运算。

  C 表达式或者变量与寄存器的关系由 GCC 自动处理，我们只需使用限制字符串指导 GCC 如何处理即可。限制字符必须与指令对操作数的要求相匹配，否则产生的汇编代码将会有错，读者可以将上例中的两个 r，都改为 m (m表示操作数放在内存，而不是寄存器中)，编译后得到的结果是：

           movl input, result

很明显这是一条非法指令，因此限制字符串必须与指令对操作数的要求匹配。例如指令 movl 允许寄存器到寄存器，立即数到寄存器等，但是不允许内存到内存的操作，因此两个操作数不能同时使用 m 作为限定字符。
内嵌汇编语法如下：

           \_\_asm\_\_(汇编语句模板: 输出部分: 输入部分: 破坏描述部分)

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。例如：

           \_\_asm\_\_ \_\_volatile\_\_("cli": : :"memory")

具体这几部分都有什么限制呢？这得从细处着手！

三、语法细节
1、汇编语句模板
    汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n” 或 “\n\t” 分开。指令中的操作数可以使用占位符引用 C 语言变量，操作数占位符最多10 个，名称如下：%0，%1，…，%9。指令中使用占位符表示的操作数，总被视为 long 型（4个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在 % 和序号之间插入一个字母，b 代表低字节，h 代表高字节，例如：%h1。

2、输出部分
    输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和 C 语言变量组成。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。 例如：

         \_\_asm\_\_ \_\_volatile\_\_("pushfl ; popl %0 ; cli":"=g" (x) )

描述符字符串表示对该变量的限制条件，这样 GCC 就可以根据这些条件决定如何分配寄存器，如何产生必要的代码处理指令操作数与 C 表达式或 C 变量之间的联系。

3、输入部分
    输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符由限定字符串和 C 语言表达式或者 C 语言变量组成。 示例如下:

例 1 ：
 \_\_asm\_\_ \_\_volatile\_\_ ("lidt %0" : : "m" (real\_mode\_idt));

例 2：
Static \_\_inline\_\_ void \_\_set\_bit(int nr, volatile void \* addr)
{

       \_\_asm\_\_(
                       "btsl %1,%0"
                       :"=m" (ADDR)
                       :"Ir" (nr));
}

后例功能是将 (\*addr) 的第 nr 位设为 1。第一个占位符 %0 与 C  语言变量 ADDR 对应，第二个占位符 %1 与 C 语言变量 nr 对应。因此上面的汇编语句代码与下面的伪代码等价：btsl nr, ADDR，该指令的两个操作数不能全是内存变量，因此将 nr 的限定字符串指定为“Ir”，将 nr 与立即数或者寄存器相关联，这样两个操作数中只有 ADDR 为内存变量。

4、限制字符
    限制字符有很多种，有些是与特定体系结构相关，此处仅列出常用的限定字符和i386中可能用到的一些常用的限定符。它们的作用是指示编译器如何处理其后的 C 语言变量与指令操作数之间的关系。

|  |  |  |
| --- | --- | --- |
| **分类** | **限定符** | **描述** |
| 通用寄存器 | “a” | 将输入变量放入eax |
| “b” | 将输入变量放入ebx |
| “c” | 将输入变量放入ecx |
| “d” | 将输入变量放入edx |
| “s” | 将输入变量放入esi |
| “d” | 将输入变量放入edi |
| “q” | 将输入变量放入eax，ebx，ecx，edx中的一个 |
| “r” | 将输入变量放入通用寄存器,即eax,ebx,ecx,edx,esi,edi之一 |
| “A” | 把eax和edx合成一个64 位的寄存器(use long longs) |
| 内存 | “m” | 内存变量 |
| “o” | 操作数为内存变量，但其寻址方式是偏移量类型, 也即基址寻址 |
| “V” | 操作数为内存变量，但寻址方式不是偏移量类型 |
| “ ” | 操作数为内存变量，但寻址方式为自动增量 |
| “p” | 操作数是一个合法的内存地址（指针） |
| 寄存器或内存 | “g” | 将输入变量放入eax，ebx，ecx，edx之一,或作为内存变量 |
| “X” | 操作数可以是任何类型 |
| 立即数 | “I” | 0-31之间的立即数（用于32位移位指令） |
| “J” | 0-63之间的立即数（用于64位移位指令） |
| “N” | 0-255之间的立即数（用于out指令） |
| “i” | 立即数 |
| “n” | 立即数,有些系统不支持除字以外的立即数,则应使用“n”而非 “i” |
| 匹配 | “ 0 ” | 表示用它限制的操作数与某个指定的操作数匹配 |
| “1” ... | 也即该操作数就是指定的那个操作数，例如“0” |
| “9” | 去描述“％1”操作数，那么“%1”引用的其实就是“%0”操作数，注意作为限定符字母的0－9 与指令中的“％0”－“％9”的区别，前者描述操作数, 后者代表操作数。 |
| & | 该输出操作数不能使用过和输入操作数相同的寄存器 |
| 操作数类型 | “=” | 操作数在指令中是只写的（输出操作数）    |
| “+” | 操作数在指令中是读写类型的（输入输出操作数） |
| 浮点数 | “f” | 浮点寄存器 |
| “t” | 第一个浮点寄存器 |
| “u” | 第二个浮点寄存器 |
| “G” | 标准的80387浮点常数 |
| % | 该操作数可以和下一个操作数交换位置,例如addl的两个操作数可以交换顺序（当然两个操作数都不能是立即数） |
| # | 部分注释，从该字符到其后的逗号之间所有字母被忽略 |
| \* | 表示如果选用寄存器，则其后的字母被忽略 |

 5、破坏描述部分
    破坏描述符用于通知编译器我们使用了哪些寄存器或内存，由逗号格开的字符串组成，每个字符串描述一种情况，一般是寄存器名；除寄存器外还有 “memory”。例如：“%eax”，“%ebx”，“memory” 等。