C内嵌汇编语法

转载自CSDN 在内嵌汇编中，可以将C语言表达式指定为汇编指令的操作数，而且不用去管如何将C语言表达式的值读入哪个寄存器，以及如何将计算结果写回C 变量，你只要告诉程序中C语言表达式与汇编指令操作数之间的对应关系即可， GCC会自动插入代码完成必要的操作。

1、简单的内嵌汇编

例：

```
__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。
```

2、内嵌汇编举例

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(编译器)对这些操作有哪些限制条件。例如在下面的汇编语句：

```
__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。本列中"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。

3、内嵌汇编语法

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")
```

3.1、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。

3.2、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：

```
__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。

3.3、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。 例1：

```
__asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));
```

例2（bitops.h）：

```
static __inline__ void __set_bit(int nr, volatile void * addr)
{
    __asm__(
            "btsl %1,%0"
            :"=m" (ADDR)
            :"Ir" (nr));
}
```

例2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 "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。

3.4、限制字符

4.1、限制字符列表

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	说明
通用寄存器	"a"	将输入变量放入eax	这里有一个问题：假设eax已经被使用，那怎么办？其实很简单：因为GCC 知道eax 已经被使用，它在这段汇编代码的起始处插入一条语句pushl %eax，将eax 内容保存到堆栈，然后在这段代码结束处再增加一条语句popl %eax，恢复eax的内容
通用寄存器	"b"	将输入变量放入ebx	
通用寄存器	"b"	将输入变量放入ebx	
通用寄存器	"c"	将输入变量放入ecx	
通用寄存器	"d"	将输入变量放入edx	
通用寄存器	"s"	将输入变量放入esi	

分类	限定符	描述	说明
通用寄存器	"d"	将输入变量放入edi	
通用寄存器	"q"	将输入变量放入eax，ebx，ecx，edx中的一个	
通用寄存器	"r"	将输入变量放入通用寄存器	也就是eax，ebx，ecx，edx，esi，edi中的一个
通用寄存器	"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	
内存	"o"	操作数为内存变量	但是其寻址方式是偏移量类型，也即是基址寻址，或者是基址加变址寻址
内存	"V"	操作数为内存变量	但寻址方式不是偏移量类型
内存	" "	操作数为内存变量	但寻址方式为自动增量
内存	"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中的一个或者作为内存变量	

分类	限定符	描述	说明
立即数	"I"	0-31之间的立即数	用于32位移位指令
立即数	"J"	0-63之间的立即数	用于64位移位指令
立即数	"N"	0-255之间的立即数	用于out指令
立即数	"i"	立即数	
立即数	"n"	立即数	有些系统不支持除字以外的立即数，这些系统应该使用"n"而不是"i"
匹配	"0"..."9"	表示用它限制的操作数与某个指定的操作数匹配，也即该操作数就是指定的那个操作数，	例如"0"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。
匹配	&	该输出操作数不能使用过和输入操作数相同的寄存器	
匹配	%	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
匹配	#	部分注释	从该字符到其后的逗号之间所有字母被忽略
匹配	*	表示如果选用寄存器，则其后的字母被忽略	
操作数类型	"="	操作数在指令中是只写的	输出操作数

分类	限定符	描述	说明
操作数类型	"+"	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	
浮点数	"t"	第一个浮点寄存器	
浮点数	"u"	第二个浮点寄存器	
浮点数	"G"	标准的80387浮点常数	

3.5、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。 memory 比较特殊，可能是内嵌汇编中最难懂部分。为解释清楚它，先介绍一下编译器的优化知识，再看C关键字volatile。最后去看该描述符。

4、编译器优化介绍

内存访问速度远不及CPU处理速度，为提高机器整体性能，在硬件上引入硬件高速缓存Cache，加速对内存的访问。另外在现代CPU中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用CPU的指令流水线，提高执行速度。以上是硬件级别的优化。再看软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。编译器优化常用的方法有：将内存变量缓存到寄存器；调整指令顺序充分利用CPU指令流水线，常见的是重新排序读写指令。对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题的解决办法是在从硬件（或者其他处理器）的角度看必须以特定顺序执行的操作之间设置内存屏障（memory barrier），linux 提供了一个宏解决编译器的执行顺序问题。

```
void barrier(void)
```

这个函数通知编译器插入一个内存屏障，但对硬件无效，编译后的代码会把当前CPU寄存器中的所有修改过的数值存入内存，需要这些数据的时候再重新从内存中读出。

5、关键字volatile

C语言关键字volatile（注意它是用来修饰变量而不是上面介绍的__volatile__）表明某个变量的值可能在外部被改变，因此对这些变量的存取不能缓存到寄存器，每次使用时需要重新存取。该关键字在多线程环境下经常使用，因为在编写多线程的程序时，同一个变量可能被多个线程修改，而程序通过该变 量同步各个线程，例如：

```
DWORD __stdcall threadFunc(LPVOID signal)
{
    int* intSignal=reinterpret_cast<int*>(signal);
    *intSignal=2;
    while(*intSignal!=1)
        sleep(1000);
    return 0;
}
```

该线程启动时将intSignal置为2，然后循环等待直到intSignal为1时退出。显然intSignal的值必须在外部被改变，否则该线程不会退出。但是实际运行的时候该线程却不会退出，即使在外部将它的值改为1，看一下对应的伪汇编代码就明白了：

```
mov ax,signal
    label:
    if(ax!=1)
        goto label
```

对于C编译器来说，它并不知道这个值会被其他线程修改。自然就把它cache在寄存器里面。记住，C编译器是没有线程概念的！这时候就需要用到volatile。volatile的本意是指：这个值可能会在当前线程外部被改变。也就是说，我们要在threadFunc中的intSignal前面加上volatile关键字，这时候，编译器知道该变量的值会在外部改变，因此每次访问该变量时会重新读取，所作的循环变为如下面伪码所示：

```
label:
    mov ax,signal
    if(ax!=1)
        goto label
```

6、Memory

有了上面的知识就不难理解Memory修改描述符了，Memory描述符告知GCC：

- 1）不要将该段内嵌汇编指令与前面的指令重新排序；也就是在执行内嵌汇编代码之前，它前面的指令都执行完毕
- 2）不要将变量缓存到寄存器，因为这段代码可能会用到内存变量，而这些内存变量会以不可预知的方式发生改变，因此GCC插入必要的代码先将缓存到寄存器的变量值写回内存，如果后面又访问这些变量，需要重新访问内存。

如果汇编指令修改了内存，但是GCC本身却察觉不到，因为在输出部分没有描述，此时就需要在修改描述部分增加"memory"，告诉GCC内存已经被修改，GCC得知这个信息后，就会在这段指令之前，插入必要的指令将前面因为优化Cache到寄存器中的变量值先写回内存，如果以后又要使用这些变量再重新读取。 虽然使用"volatile"也可以达到这个目的，但是我们在每个变量前增加该关键字，不如使用"memory"方便。

回到顶部