学习 GCC 内联汇编

2015/8/20
renyl

1 介绍

1) GCC 支持在 C/C++代码中嵌入汇编代码，这些代码被称作是“GCC Inline ASM”（GCC 内联汇编）。
2) 内联汇编可以使得 C/C++语法无法表达的指令直接嵌入代码中。
3) 在 C/C++代码中使用内联汇编可以提高程序运行效率。

接下来就介绍下 GCC 内联汇编的用法。

2 基本形式

GCC 中基本的内联汇编语句格式如下：
__asm__ [__volatile__] (“instruction list”); //注意是两个下划线
关于各部分的说明如下：

2.1 __asm__

1) 它是 GCC 定义的关键字 asm 的宏定义（#define __asm__ asm），它是用来声明一个内联汇编表达式，任何一个内联汇编表达式都必须以它开头。
2) 如果要编写符合 ANIS C 标准的代码（即与 ANSI C 兼容），需要使用__asm__。

2.2 __volatile__

1) 它是 GCC 定义的关键字 volatile 的宏定义（#define __volatile__ volatile），这个选项是可选的，用来向 GCC 声明“不要优化 instruction list 中的指令”。
2) 如果不使用__volatile__，当使用优化选项-O 进行优化编译时，GCC 将会根据自己的判断来决定是否将这个内联汇编表达式中的指令优化掉。
3) 如果要编写符合 ANIS C 标准的代码（即与 ANSI C 兼容），需要使用__volatile__。

2.3 instruction list

1) 它是汇编指令列表。它可以是空列表，比如：__asm__ __volatile__(“”);就是合法的内联汇编表达式，只不过这条语句什么都不做，没有意义。
2) Instruction list 的编写规则：

a）当指令列表里面有多条指令时，可以在一对双引号中全部写出，也可以将一条或多条指令放在一对双引号中，所有指令放在多对双引号中。

b） 如果多条指令在一对双引号中，相邻两条指令之间必须使用分号（;）或换行号（\n）隔开。

c）建议每条指令都用“instruction \n\t”这种形式，这样便于使用 gcc -S 编译后查看汇编代码。

3 带有 C/C++表达式的内联汇编

带有 C/C++表达式的内联汇编语句格式如下：

__asm__ [__volatile__] (“instruction list”:Output :Input :Clobber/Modify);

先给出个例子，通过这个例子来对各个参数进行说明：

test1.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
     int a=10,b;
     __asm__ __volatile__(
                        "movl %1 , %%eax \n\t"    //注意有个逗号
                        "movl %%eax , %0 \n\t"
                        :"=c"(b)              /* Output */
                        :"d"(a)               /* Input */
                        :"%eax"               /* Clobber Register */
                        );

        printf("b=%d\n",b);
        return 0;
}
```

关于各部分的说明如下：

3.1Output

1) Output 部分用来指定当前内联汇编语句的输出，称为输出操作表达式。格式为：“操作约束”（C+/C++表达式），如上例中的（“=c”（b））。
2) 用双引号括起来的部分为“操作约束”（Operation Constraint），上列中的“操作约束”（“=c”）包含两个组成部分：等号（=）和字母 c。其中，等号（=）说明圆括号中的表达式 b 是一个只写的表达式，只能被用作输出，不能用作输入；字母 c 表示寄存器 ecx。
3) 用圆括号括起来的部分为 C/C++表达式，它用于保存当前内联汇编语句的一个输出值，其操作是 C/C++赋值语句“=”的左值部分。
4) 那么（“=c”（b））表达的意思就是，把寄存器 ecx 中的值放入变量 b 中。
5) 在 Output 部分可以出现多个输出表达式，多个输出表达式之间必须用逗号（,）隔开。

3.2Input

1) Input 部分用来指定当前内联汇编语句的输入，称为输入操作表达式，格式为：“操作约束”（C/C++表达式）。
2) 上列中的“操作约束”（“d”）只由字母 d 组成，由于没有等号（=）,则表达式 a 只能被用作输入，不能用作输出。
3) 那么（“d”（a））表达的意思就是，把变量 a 中的值放入寄存器 edx 中。
4) 在 Input 部分可以出现多个输出表达式，多个输出表达式之间必须用逗号（,）隔开。

注：等号（=）只能出现在 Output 中，不能出现在 Input 中。

3.3 Clobber/Modify

1) 由于内联汇编语句可能会对某些寄存器或内存进行修改，需要通知 GCC 在编译时将这一点考虑进去，这时候就需要在 Clobber/Modify 中指出。
2) 如上例中的指令语句（movl %1 , %%eax）就对 eax 寄存器进行了修改，那么就需要在 Clobber/Modify 中指出从而通知 GCC 这段代码对 eax 寄存器进行了修改，这样 GCC 在编译会考虑到这点，就不会使用 eax 寄存器保存某些值，否则它将被覆盖掉从而导致数据丢失。
3) 在 Clobber/Modify 中不需要列出在 Output 和 Input 操作表达式中指定的寄存器，因为 GCC 在为给你分配一个寄存器时，对这些寄存器的状态是清楚的，它知道这些寄存器是要被修改的，因此，就不需要在 Clobber/Modify 部分声明它们。
4) 在 Clobber/Modify 中部分由多个寄存器需要声明时，每个寄存器之间用逗号(,)隔开。

注：关于（movl %1，%%eax）中（%1）表示什么意思，暂时不解释，后面会详细说明

3.4 小节

关于 test1.c 中内联汇编的执行过程如下所示：
1) 把变量 a 中的值放入 edx 寄存器 ->（"d"(a)）
2) 把 edx 寄存器中的值放入 eax 寄存器 ->（movl %1 , %%eax）
3) 把 eax 寄存器中的值放入到 ecx 寄存器 ->（movl %%eax , %0）
4) 把 exc 寄存器中的值放入到变量 b 中 ->（"=c"(b)）

把这段 C 代码进行编译，通过查看汇编代码来验证这段内联汇编的执行过程：

```
[root@localhost assemble]#gcc -S test1.c -O test1.s
[root@localhost assemble]#cat test1.s
.file   "test1.c"
.section        .rodata
.LC0:
        .string "b=%d\n"
        .text
```

```
.globl main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -8(%rbp)   // int a=10;
        movl    -8(%rbp), %edx  //把变量 a 中的值放入 edx 寄存器
#APP
# 7 "test.c" 1
        movl %edx, %eax         //把 edx 寄存器中的值放入 eax 寄存器
        movl %eax, %ecx         //把 eax 寄存器中的值放入到 ecx 寄存器

# 0 "" 2
#NO_APP
        movl    %ecx, %edx
        movl    %edx, -4(%rbp)  //这两句实现把 exc 寄存器中的值放入到变量 b 中
        movl    $.LC0, %eax
        movl    -4(%rbp), %edx
        movl    %edx, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-3)"
        .section        .note.GNU-stack,"",@progbits
```

注：汇编输出结果中，#APP 与#NO_APP 之间为“内联汇编指令”。

需要注意的一点是，内联汇编格式
__asm__ [__volatile__] (“instruction list” :Output :Input :Clobber/Modify);的某个部分若为空的话，冒号（:）不要省略，否则无法说明不为空的部分究竟是第几部分。如下所示：

test2.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int a=10;
__asm__ __volatile__(
                    "movl %0,%%ecx"
                    :"r"(b)     //由于 Output 部分为空，就把冒号(:)省略了，这样
                    :"%ecx"     //会让 GCC 以为"r"(b)为 Output 部分，从而出错
                    );
return 0;
}
```

到目前为止，简单了介绍了 C/C++表达式的内联汇编形式，接下将对“占位符”、“操作约束”以及“Cloberr/Modify”进行详细介绍。

4 占位符

占位符是在指令语句中使用的，上例中的指令语句（movl %1，%%eax）中（%1）就是占位符。每一个占位符对应一个 Output/Input 操作表达式。

关于占位符的详细说明如下：

1) GCC 规定一个内联汇编语句中最多只能有 10 个 Output/Input 操作表达式，这些表达式按照它们被列出来的顺序依次赋予标号 0 到 9。对于占位符中的数据而言，与这些编号是对应的。比如：占位符%0 对应编号为 0 的操作表达式，依次类推。

2) GCC 在对占位符进行编译的时候，会将每一个占位符替换为对应的 Output/Input 表达式所指定的寄存器/内存/立即数。因此，test1.c 中的（movl %1，%%eax）在编译后就变成了（movl %edx，%eax）。

3) 由于占位符前面要一个百分号%，为了区别占位符与寄存器，在带有 C/C++表达式的内联汇编语句的指令列表里列出的寄存器名称前面必须使用两个百分号（%%）。

4) 在基本形式的内联汇编中，寄存器前面只能使用一个百分号（%），如下所示：

test3.c

```
int main ()
{
    __asm__("movl %%ebx, %%eax\n\t");
    return 0;
}
```

注：

1：这段代码没有意义，只是为了说明。

2：Instruction list 后面若没有冒号（:）存在，则为基本形式的内联汇编。

在使用 GCC 编译这段代码时，会提示错误“Error: bad register name `%%ebx’”，当去掉一个百分号时，编译即可通过。

5 操作约束

1） 每一个 Input 和 Output 表达式都必须指定自己的操作约束,约束的类型有:寄存器约束，通用约束，立即数约束，内存约束，修饰符。
2）操作表达式的格式为:“操作约束”（C/C++操作表达式）。

接下来就对每种操作约束进行说明：

5.1 寄存器约束

1) 当内联汇编的输入或输出需要借助于一个寄存器时，可以为其指定一个寄存器约束。如上例中的"=c"(b)指定了一个输出寄存器 ecx。

2) 当不需要为某个表达式指定特定的寄存器时,可以使用字母 r 来指定通用寄存器(eax、ebx、exc、edx、esi 和 edi）中的某一个。GCC 会自动分配通用寄存器中的一个给你，如把 test1.c 中的"=c"(b)改为"=r"(b)，"d"(a)改为"r"(b),然后编译查看汇编代码，如下所示：

```
…
    movl    $10, -20(%rbp)
    movl    -20(%rbp), %edx  //为输入表达式分配了 edx 寄存器
#APP
# 7 "test.c" 1
    movl %edx, %eax
    movl %eax, %ebx         //为输出表达式分配了 ebx 寄存器

# 0 "" 2
#NO_APP
    movl    %ebx, -24(%rbp)
    movl    -24(%rbp), %eax
    movl    %eax, %esi
…
```

当使用 r 来指定通用寄存器时，占位符就起到了重要的作用。因为你无法在 Intrusion list 中指明表达式所使用的寄存器，表达式所使用的寄存器是在 GCC 编译时分配的，这个时候只有使用占位符才能表达，这也是占位符存在的意义。

3） GCC 会根据当前操作表达式中 C/C++表达式的宽度来决定使用 ecx、cx 还是 cl。把 test1.c 中的(int a=10,b;)改为（short a=10,b;）然后编译查看汇编代码：

```
…
```

```
movzwl  -18(%rbp), %edx
#APP
# 7 "test.c" 1
    movl %dx, %eax          //由于 a 占 short 类型，已经变为 dx 寄存器了
    movl %eax, %cx          //由于 b 占 short 类型，已经变为 cx 寄存器了

# 0 "" 2
#NO_APP
    movl    %ecx, %ebx
    movw    %bx, -20(%rbp)
    movswl  -20(%rbp), %eax
…
```

5.2 通用约束

1) 约束名“g”可以用于输入和输出表达式中，表示可以使用通用寄存器、内存、立即数等任何一种处理方式。使用约束名“g”可以让 GCC 可以根据不同的 C/C++表达式生成不同的访问方式。如下所示：

test4.c

```
#define XCH(foo)  __asm__ __volatile__ (\
                "movl %0, %%eax\n\t" \
                :\
                :"g"(foo)\
                :"%eax"\
                );
int main()
{
        XCH(100)
        int a=10;
        XCH(a)
        return 0;
}
```

使用 GCC 编译后查看汇编代码：

```
…
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
#APP
# 11 "test4.c" 1
    movl $100, %eax             //立即数处理方式

# 0 "" 2
#NO_APP
```

```
    movl    $10, -4(%rbp)
#APP
# 14 "test4.c" 1
    movl -4(%rbp), %eax     //访问内存处理方式

# 0 "" 2
#NO_APP
    movl    $0, %eax
    popq    %rbp
…
```

2） 约束名“0,1,2,3,4,5,6,7,8”只能用于输入表达式，表示与第 n 个操作表达使使用相同的寄存器/内存。如下所示：

test5.c

```
int main(int argc, char *argv[])
{
    int a=10,b=20;
    __asm__ __volatile__(
                "movl %1, %%eax\n\t"
                "movl %2, %%ebx\n\t"
                "addl %%eax, %%ebx\n\t"
                "movl %%ebx,%0 \n\t"
                :"=c"(a)
                :"0"(a),"r"(b) //表示与第一个操作表达式使用相同的寄存器
                :"%eax","%ebx"
                );
    return 0;
}
```

使用 GCC 编译后查看汇编代码：

```
…
    movl    -24(%rbp), %edx
    movl    %eax, %r12d
#APP
# 8 "tmp.c" 1
    movl %ecx, %eax       //使用了与第一个表达式相同的寄存器 ecx
    movl %edx, %ebx       //GCC 为输入表达式 b 分别了 edx 寄存器
    addl %eax, %ebx
    movl %ebx, %ecx

# 0 "" 2
#NO_APP
…
```

5.3 立即数约束

1） 如果一个 Input 中的 C/C++操作表达式是一个常数，那么可以不借助任何寄存器或内存，直接使用立即数约束。如下所示：

test6.c

```
int main(int argc, char *argv[])
{
    int a=10;
    __asm__ __volatile__(
                    "movl %1,%0\n\t"
                    :"=r"(a)
                    :"i"(100)
                    :
                    );
    return 0;
}
```

使用 GCC 编译后查看汇编代码：

```
…
    movq    %rsi, -40(%rbp)
    movl    $10, -12(%rbp)
#APP
# 6 "tmp.c" 1
    movl $100,%ebx          //立即数处理方式

# 0 "" 2
#NO_APP
    movl    %ebx, -12(%rbp)
    movl    $0, %eax
…
```

2） 约束名“i”（“f”）表示输入表达式是一个整数（浮点数）类型的立即数，不需要借助任何寄存器或内存，只能用于 Input 部分，不能用于 Output 部分。

5.4 内存约束

1） 如果一个 Output/Input 操作表达式想要直接使用内存地址，不想借助任何寄存器，则可以使用内存约束。

2） 使用内存方式进行输入/输出时，由于不借助任何寄存器，GCC 不会按照你的声明对其作任何的输入/输出处理，GCC 只会直接拿来用，究竟对这个 C/C++表达式而言，是输入还是输出完全依赖于你写在“Instruction list”中的指令对其操作。因此，对于内存约束类型的操作表达式而言，放在 Output 部分还是放在 Input 部分，对编译结果是没有影响的。如下所示：

test7.c	test8.c	test9.c

#include <stdio.h>
int main()
{
int count=10;
__asm__ __volatile__(
"decl %0\n\t"
:
:"m"(count)
:
);
 	
printf("count=%d\n",coun
t); return 0;
}
	#include <stdio.h>
	int main()
	{
	int count=10;
	__asm__ __volatile__(
	"decl %0\n\t"
	: "=m"(count)
	:
	:
	);
	
	printf("count=%d\n",coun
	t); return 0;
	}
		#include <stdio.h>
		int main()
		{
		int count=10;
		__asm__ __volatile__(
		"decl %0\n\t"
		: "=m"(count)
		: "m"(count)
		:
		);
		
		printf("count=%d\n",cou
		nt); return 0;
		}

使用 GCC 编译对上面 3 个源代码进行编译，然后查看汇编代码，发现它们的汇编代码是一样的，如下所示：

```
…
    subq    $16, %rsp
    movl    $10, -4(%rbp)
#APP
# 7 "tmp2.c" 1
    decl -4(%rbp)

# 0 "" 2
#NO_APP
    movl    -4(%rbp), %eax
    movl    %eax, %esi
…
```

5.5 修饰符

关于修饰符主要有以下几种：

修饰符	I/O	说明
=	O	表示此 Output 操作表达式是只写的
+	O	表示此 Output 操作表达是可读可写的
&	O	表示此 Output 操作表达式独占为其指定的寄存器
%	I	表示此 Input 与 Output 部分的 C/C++表达式可以互换

注：I 表示输入操作表达式，O 表示输出操作表达式。

5.5.1 修饰符（=）与（+）

关于修饰符（=）与（+）的说明，如下所示：

test10.c	test11.c

#include <stdio.h>
int main(int argc, char *argv[])
{
int a=10,b;
__asm__ __volatile__(
"movl %1 , %%eax \n\t"
"movl %%eax , %0 \n\t"
:"=c"(b) /* Output */
:"d"(a) /* Input */
:"%eax" /* Clobber Register */
);
printf("b=%d\n",b);
return 0;
}
	#include <stdio.h>
	int main(int argc, char *argv[])
	{
	int a=10,b;
	__asm__ __volatile__(
	"movl %1 , %%eax \n\t"
	"movl %%eax , %0 \n\t"
	:"+c"(b) /* Output */
	:"d"(a) /* Input */
	:"%eax" /* Clobber Register */
	);
	printf("b=%d\n",b);
	return 0;
	}

使用 GCC 对上面 2 个源代码进行编译，然后查看汇编代码：

test10.s	test11.s

…
movl $10, -20(%rbp)
movl -20(%rbp), %edx
//把变量 a 放入 edx 寄存器
//没有把变量 b 放入 ecx 寄存器
#APP
# 7 "inline.c" 1 	
movl %edx, %eax
movl %eax, %ecx
 	
# 0 "" 2 	
#NO_APP
movl %ecx, %ebx
movl %ebx, -24(%rbp)
movl -24(%rbp), %eax
movl %eax, %esi
…
	…
	movl $10, -20(%rbp)
	movl -20(%rbp), %edx
	movl -24(%rbp), %eax
	movl %eax, %ebx
	movl %ebx, %ecx
	//这 3 个指令实现了把变量 b 放入 ecx 寄存器
	#APP
	# 7 "inline.c" 1
	movl %edx, %eax
	movl %eax, %ecx
	
	# 0 "" 2
	#NO_APP
	movl %ecx, %ebx
	movl %ebx, -24(%rbp)
	movl -24(%rbp), %eax
	movl %eax, %esi
	…

从汇编代码可以出来，修饰符（+）表示 Output 操作表达式是可读可写的。对于上面这个例子使用修饰符（+）没有多大意义，因为使用修饰符（=）也行。但是对下面的例子来说，修饰符（+）才真正发挥它的作用，如下所示：

test12.c

```
#include <stdio.h>
int main()
{
int a=1,b=2;
__asm__ __volatile__(
  "addl %1,%0 \n\t"
   :"=r"(b)
   :"r"(a)
   :
   );
printf("b=%d\n\t",b);
return 0;
}
```

test13.c

```
#include <stdio.h>
int main()
{
int a=1,b=2;
__asm__ __volatile__(
   "addl %1,%0 \n\t"
   :"=r"(b)
   :"r"(a), "r"(b)
   :
   );
printf("b=%d\n\t",b);
return 0;
}
```

test14.c

```
#include <stdio.h>
int main()
{
int a=1,b=2;
__asm__ __volatile__(
   "addl %1,%0 \n\t"
   :"+r"(b)
   :"r"(a)
   :
   );
printf("b=%d\n\t",b);
return 0;
}
```

使用 GCC 对上面 3 个源代码进行编译，然后查看汇编代码：

test12.s

```
…
    movl   $1, -20(%rbp)
    movl   $2, -24(%rbp)
    movl -20(%rbp), %eax
#APP
# 24 "inline.c" 1
    addl %eax,%ebx

# 0 "" 2
#NO_APP
    movl %ebx, -24(%rbp)
    movl
-24(%rbp), %eax
…
```

test13.s

```
…
    movl   $1, -20(%rbp)
    movl   $2, -24(%rbp)
    movl  -24(%rbp), %eax
    movl  -20(%rbp), %edx
#APP
# 24 "inline.c" 1
    addl %eax,%ebx

# 0 "" 2
#NO_APP
    movl  %ebx, -24(%rbp)
    movl  -24(%rbp), %eax
    movl   %eax, %esi
…
```

test14.s

```
…
    movl  $1, -20(%rbp)
    movl  $2, -24(%rbp)
    movl  -20(%rbp), %eax
    movl  -24(%rbp), %edx
    movl  %edx, %ebx
#APP
# 24 "inline.c" 1
    addl %eax,%ebx

# 0 "" 2
#NO_APP
    movl  %ebx, -24(%rbp)
    movl  -24(%rbp), %eax
    movl   %eax, %esi
…
```

对上面 3 个汇编代码简单分析如下所示：

1）test12.s:

a)：把变量 a 放入到寄存器 eax

b)：指令 addl 指令，寄存器 eax 和 ebx 值相加并把结果放入 ebx 寄存器（由于此时变量 b 并没有放入寄存器 ebx 中，所以结果是错误的）

c)：把 ebx 寄存器中的值写到变量 b 的内存

2）test13.s

a)：把变量 a 放入到寄存器 eax，把变量 b 放入到寄存器 edx

b)：指令 addl 指令，寄存器 eax 和 ebx 值相加并把结果放入 ebx 寄存器（由于此时变

量 b 并没有放入寄存器 ebx 中，而是放入到 edx 寄存器，所以结果是错误的）

c）：把 ebx 寄存器中的值写到变量 b 的内存

3）test14.s

a）：把变量 a 放入到寄存器 eax，把变量 b 放入到寄存器 edx

b）：指令 addl 指令，寄存器 eax 和 ebx 值相加并把结果放入 ebx 寄存器（由于此时变量 b 放入寄存器 ebx 中，所以结果是正确的）

c）：把 ebx 寄存器中的值写到变量 b 的内存

针对 test13.c 需要说明的是：

输出表达式（“=r（b)”）与输入表达式（“r”(b)）（使用了约束符“r”表示从通用寄存器中选择一个寄存器）它们应该使用同一个寄存器，但是 GCC 并不去判断，致使它们使用了不同的寄存器，从而导致最终的结果不正确。

回想下“通用约束”，问题便可解决，如下所示：

test15.c

```
#include <stdio.h>
int main()
{
int a=1,b=2;
__asm__ __volatile__(
   "addl %1,%0 \n\t"
   :"=r"(b)
   :"r"(a), "0"(b)    //这里指定了输入变量 b 和输出变量 b 使用同一个寄存器
   :
   );
printf("b=%d\n\t",b);
return 0;
}
```

使用 GCC 对上面代码进行编译，然后查看汇编代码：

```
…
    movl    $1, -20(%rbp)
    movl    $2, -24(%rbp)
    movl    -20(%rbp), %eax
    movl    -24(%rbp), %edx     //输入变量使用 edx 寄存器
    movl    %edx, %ebx          //输入变量改为 ebx 寄存器
#APP
# 24 "inline.c" 1
    addl %eax,%ebx             //输出变量使用 ebx 寄存器

# 0 "" 2
#NO_APP
…
```

5.5.2 修饰符（&）

关于修饰符（&）的说明，如下所示：

1） 修饰符（&）只能用于对 Output 操作表达式的修饰，它的作用就是要求 GCC 不得为任何 Input 操作表达式分配与 Output 表达式（用修饰符（&）修饰的）相同的寄存器。如下所示：

test16.c

```
int main()
{
int a=10,b=20;
 __asm__ __volatile__(
      "pop %0 \n\t"
      "mov %1, %%ecx \n\t"
      :"=a"(a)
      :"r" (b)
      :"%ecx"
      );
return 0;
}
```

test17.c

```
int main()
{
int a=10,b=20;
 __asm__ __volatile__(
      "pop %0 \n\t"
      "mov %1, %%ecx \n\t"
      :"=&a"(a)
      :"r" (b)
      :"%ecx"
      );
return 0;
}
```

使用 GCC 对上面代码进行编译，然后查看汇编代码：

test16.s

```
…
    movl    $10, -12(%rbp)
    movl    $20, -16(%rbp)
    movl    -16(%rbp), %eax
    movl    %eax, -28(%rbp)
    movl    -28(%rbp), %eax
#APP
# 7 "test.c" 1
    pop %eax
    mov %eax, %ecx

# 0 "" 2
#NO_APP
…
```

test17.s

```
…
    movl    $10, -12(%rbp)
    movl    $20, -16(%rbp)
    movl    -16(%rbp), %eax
    movl    %eax, -28(%rbp)
    movl    -28(%rbp), %edx
#APP
# 7 "test.c" 1
    pop %eax
    mov %edx, %ecx

# 0 "" 2
#NO_APP
…
```

说明：

a） test16.c：

这段代码的本意是想把变量 b 值传给寄存器 ecx，但是由于 GCC 在编译时给输入表达式分配了寄存器 eax，而之前的指令“pop %eax”的寄存器 eax 的值被修改了，这样当执行指令“mov %eax, %ecx”时，eax 寄存器里保存的并不是变量 b 的值，违背了本段代码的本意。

b） test15.c

由于给输出表达式指定了符号（“&”）修饰，这样当输出表达式使用 eax 寄存器时，输入表达式就使用别的寄存器，从而符合本段代码的本意。

2） 给 Output 表达式使用符号（“&”）进行修饰后，Output 和 Input 表达式将使用不相同的寄存器，但这并不是绝对的，如下所示：

test18.c	tets19.c

#include <stdio.h>
int main()
{
int a=1,b;
__asm__ __volatile__(
"movl %1,%0 \n\t"
:"=&c"(b)
:"c"(a)
:
);
printf("b=%d\n\t",b);
return 0;
}
	#include <stdio.h>
	int main()
	{
	int a=1,b;
	__asm__ __volatile__(
	"movl %1,%0 \n\t"
	:"=&c"(b)
	:"0"(a)
	:
	);
	printf("b=%d\n\t",b);
	return 0;
	}

使用 GCC 对上面代码进行编译，然后查看汇编代码：

a） test18.c 编译时会出错，因为 Output 表达式使用（&）修饰符指定了 exc 寄存器，那么 Input 表达式就不能再使用 ecx 寄存器了。

b） test19.c 汇编代码如下：

```
…
    movl    $1, -20(%rbp)
    movl    -20(%rbp), %eax
    movl    %eax, %ebx
    movl    %ebx, %ecx
#APP
# 5 "tmp2.c" 1
    movl %ecx,%ecx    //Input 和 Output 表达式使用了相同的寄存器 exc

# 0 "" 2
#NO_APP
    movl    %ecx, %ebx
…
```

通过汇编代码 test19.s 可以知道，即使 Output 表达式使用修饰符（“&”）指定固定寄存器，仍然可以在 Input 部分使用约束名“0,1,2,3,4,5,6,7,8”使得该输入表达式可以与 Output 表达式使用相同的寄存器。

3） 如果一个 Output 操作表达式的寄存器约束被指定为某个寄存器时，只有当至少存在一个 Input 操作表达式使用了可选约束（如“r”和“g”）时，Output 操作表达式使用符

号(“&”)修饰才有意义。如下所示：

test20.c	test21.c

#include <stdio.h>
int main()
{
int a=1,b;
__asm__ __volatile__(
"movl %1,%0 \n\t"
:"=&c"(b)
:"b"(a)
:
);
printf("b=%d\n\t",b);
return 0;
}
	#include <stdio.h>
	int main()
	{
	int a=1,b;
	__asm__ __volatile__(
	"movl %1,%0 \n\t"
	:"=&c"(b)
	:"r"(a)
	:
	);
	printf("b=%d\n\t",b);
	return 0;
	}

说明：

a） test20.c：

由于 Input 表达式指定了寄存器 ebx，那么此时 Output 表达式在指定了寄存器时使用符号（“&”）修饰就没有任何意义了。如果 Output 表达式没有指定固定寄存器，那么使用符号（“&”）修饰就有意义，如 Output 部分为（“=&r”(b)），Input 部分为（“b”(a)）则表明：“GCC 在编译时给 Output 表达式分配寄存器时不要分配 ebx 寄存器，因为 Input 表达式已经在使用 ebx 寄存器了”。

b） test21.c：

由于 Input 表达式没有指定某个固定寄存器，而是指定了通用寄存器。这个时候 Output 表达式使用符号(“&”)修饰表明：“GCC 在编译时不要给 Input 表达式分配 ecx 寄存器，因为 Output 表达式需要使用 ecx 寄存器”。

注：Output 部分中不能指定相同的寄存器，Input 部分也不能指定相同的寄存器。

5.5.3 修饰符（%）

关于修饰符（“%”）的说明，如下所示

1） 修饰符(“%”)只能用于 Input 操作表达式中。

2） 修饰符(“%”)用于向 GCC 声明：“当前 Input 操作表达式可以与下一个 Input 操作表达互换“这个修饰符一般用于符合交换规律的运算，比如“加法运算”，如下所示：

test22.c	test23.c

int main()
{
int a=10,b=20,c;
__asm__ __volatile__(
"addl %1,%0\n\t"
	int main()
	{
	int a=10,b=20,c;
	__asm__ __volatile__(
	"addl %1,%0\n\t"

```
        :"=r"(c)
        :"r"(a),"0"(b)
        );
 return 0;
}
```

```
        :"=r"(c)
        :"r"(b),"0"(a)
        );
 return 0;
}
```

说明：对 Input 表达式使用百分号（“%”）修饰，让 GCC 知道变量 a 和变量 b 可以互换，那么 GCC 就可以自动将 test22.c 中的内联汇编改变为 test23.c 中的内联汇编。

6 Clobber/Modify

Clobber/Modify 部分主要是标识寄存器/内存做了改变，这样使得 GCC 在编译的时候考虑到这一点。

6.1 Clobber

关于 Clobber 部分的说明如下：

1） 如果你在一个内联汇编语句的 Clobber 部分向 GCC 声明了某个寄存器内容发生了改变，GCC 在编译时发现这个被声明的寄存器的内容在此内联汇编之后还要继续使用，那么 GCC 会首先将此寄存器的内容保存起来，然后在此内联汇编语句的相关代码生成之后，再将其内容恢复。

2） GCC 对指令列表中的寄存器使用情况是不清楚的，在 Clobber 部分声明某些寄存器，是为了告诉 GCC 关于这些寄存器的信息，GCC 得知这些寄存器的使用情况，在进行编译时才会正常地分配和利用寄存器。

3） 在 Clobber 部分声明了一个寄存器，那么这个寄存器将不能再被用作当前内联汇编语句的 Output/Input 操作表达式的寄存器约束。

6.2 Modify

关于 Modify 部分的说明如下：

1） 除了寄存器的内容会被修改之外，内存的内容也会被修改。

2） 如果一个内联汇编语句中的指令对内存进行了修改，或者在此内联汇编出现的地方，内存内容可能发生改变，而被改变的内存地址没有在 Output 操作表达式中使用“m”约束，在这种情况下，需要在 Modify 部分使用“memory”向 GCC 声明：“在这里，内存发生了或可能发生了改变”。

3） 如果一个内联汇编语句的 Modify 部分存在“memory”，那么 GCC 会保证在此内联汇编之前，如果某个内存的内容被装入了寄存器，那么在这个内联汇编之后，如果需要使用这个内存的内容，会直接从这个内存重新读取，而不是使用被存放在寄存器中的拷贝，因为这个时候寄存器中的拷贝很可能已经和内存中的内容不一致了。

附录

本文使用到的“操作约束符”含义如下所示：

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 eax、ebx、ecx、edx、esi、edi 中的一个
	g	将输入变量放入 eax、ebx、ecx、edx、esi、edi 或内存或立即数中
内存	m	内存变量
立即数	i	整型立即数
	f	浮点型立即数
操作数类型	=	输出操作数在指令中是只写的
	+	输出操作数在指令中是可读可写的
	&	该输出操作数不能使用和输入操作数相同的寄存器
	%	该输入操作数可以和下一个输入操作数交换位置（不能是立即数）

参考文献

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html