33_3 实现两个正整数相乘。

```
#---输入指令:1:  (I,      ,      ,      a      )
in R1               #输入一个值
movi 0                    #将变量的内存地址值赋值到R0
movb R0,R1                #将R1的值赋值给变量所在的内存地址
#---输入指令:2:  (I,      ,      ,      b      )
in R1               #输入一个值
movi 1                    #将变量的内存地址值赋值到R0
movb R0,R1                #将R1的值赋值给变量所在的内存地址
#---乘法指令:3:  (*,     a,     b,     T1     )
movi 0                    #乘法指令，变量a的内存地址放入R0
movc R1,R0                #乘法指令，变量a从内存地址R0拷贝至R1
movi 7                    #乘法指令，临时内存空间地址放R0
movb R0,R1                #乘法指令，变量a放入临时内存空间地址
movi 1                    #乘法指令，变量b的内存地址放入R0
movc R1,R0                #乘法指令，变量b从内存地址R0拷贝至R1
movi 8                    #乘法指令，临时内存空间地址放R0
movb R0,R1                #乘法指令，变量b放入临时内存空间地址
#开始做乘法循环
movi 8                    #b的内存地址存至R0
movc R1,R0                      #从内存中取出值b
movi 1                    #设置R0中的值为1
sub R1,R0                 #R1即b值减1，此时设置G值
movi 8                    #b的内存地址存至R0
movb R0,R1                      #b值需要保存回去
movi 7                    #a的内存地址存至R0
movc R2,R0                      #取出a值
movi 9                    #a+b的结果内存地址存至R0
movc R1,R0                      #取出a+b的结果值
```

```
movc R1,R0              #取出a+b的结果值
add R1,R2           #做加法
movb R0,R1              #将结果存回去
movd                #保存当前的PC值到R3
movi -12            #R0的值设置为-12
add R3,R0               #R3的值加-12
jg              #如果第12行的减法设置G为1,就跳转
#循环结束
movi 9              # 做乘法，将临时结果的内存地址值赋值到R0
movc R2,R0              #取出a+b的结果值
movi 3              # 做乘法，将结果的内存地址值赋值到R0
movb R0,R2              #将结果值写回内存
#---赋值指令:4: (=,  T1,   ,    c    )
movi 3              #将变量b的内存地址值赋值到R0
movc R1,R0          #从内存中取出b的值到R1
movi 2              #将变量a的内存地址值赋值到R0
movb R0,R1          #将R1的值（即b）赋值给a所在的内存地址
#---输出指令:5: (O,   ,    ,    c    )
movi 2              #将变量的内存地址赋值给R0
movc R1,R0          #将变量的值从内存送到寄存器R1
out R1              #输出变量的值
mova R0,R0          #空指令
halt
```

在系统中执行该文件:

```
PS D:\Desktop\CS资料\tool> ./vspm 33.txt
VSPM-湖南大学非常简单原型机      1.0
        作者: 杨科华
VSPM  start ...
VSPM  info:
        地址位数:  8  bit, 共  256  字节
         6  个寄存器: R0~R3,G,PC
初始化内存......        OK!
初始化寄存器......      OK!
分配数据段......        OK!
        数据段大小为: 10个字节,0000 0000 ~ 0000 1010
装载指令.....  OK!
        共43条指令
准备执行指令,第一条指令所在地址及指令内容为:
        0000 1010               in R1                   #输入一个值
```

文件在原型机中打开的样子。

```
        0000 1010               in R1                   #输入一个值
VM> si
5
        0000 1011               movi 0                  #将变量的内存地址值赋值到
VM> si
        0000 1100               movb R0,R1              #将R1的值赋值给变量所在的
VM> si
        0000 1101               in R1                   #输入一个值
VM> si
3
```

输入两个乘数：5 和 3

```
VM> si
3
        0000 1110               movi 1                  #将变量的内存地址值赋值到R0
VM> si
        0000 1111               movb R0,R1              #将R1的值赋值给变量所在的内存地址
VM> si
        0001 0000               movi 0                  #乘法指令，变量a的内存地址放入R0
VM> si
        0001 0001               movc R1,R0              #乘法指令，变量a从内存地址R0拷贝至R1
VM> si
        0001 0010               movi 7                  #乘法指令，临时内存空间地址放R0
VM> si
        0001 0011               movb R0,R1              #乘法指令，变量a放入临时内存空间地址
VM> si
        0001 0100               movi 1                  #乘法指令，变量b的内存地址放入R0
VM> si
        0001 0101               movc R1,R0              #乘法指令，变量b从内存地址R0拷贝至R1
VM> si
        0001 0110               movi 8                  #乘法指令，临时内存空间地址放R0
VM> si
        0001 0111               movb R0,R1              #乘法指令，变量b放入临时内存空间地址
VM> si
```

执行乘法指令

```
VM> si
        0010 1001           movc R2,R0                      #取出a+b的结果值
VM> si
        0010 1010           movi 3                  # 做乘法，将结果的内存地址值赋值到R0
VM> si
        0010 1011           movb R0,R2                      #将结果值写回内存
VM> si
        0010 1100           movi 3                  #将变量b的内存地址值赋值到R0
VM> si
        0010 1101           movc R1,R0              #从内存中取出b的值到R1
VM> si
        0010 1110           movi 2                  #将变量a的内存地址值赋值到R0
VM> c
15
```

最后我们得到结果输出：15

47_3

```
PS D:\Desktop\CS资料\tool> gcc -g -o 47_3 47_3.s
47_3.s: Assembler messages:
47_3.s: Warning: end of file not at end of a line; newline inserted
47_3.s:2: Warning: using `%ax' instead of `%eax' due to `w' suffix
47_3.s:3: Error: too many memory references for `mov'
47_3.s:4: Error: bad register name `%sl'
47_3.s:5: Error: operand type mismatch for `mov'
47_3.s:6: Error: invalid instruction suffix for `mov'
47_3.s:7: Error: `%si' not allowed with `movb'
47_3.s:1: Error: invalid operands (*UND* and *UND* sections) for `%'
```

第一行是使用 % 符号的方式不符合语法要求。

可以看到第二行的错误是对于 movw，我们应该使用%ax 而不是使用%eax，因为“w”是对于 16 位的操作而%eax 是三十二位寄存器。

第三行的错误是对于 mov，使用了两个内存引用。

```
movl (%eax), 4(%esp)
```

我们看到两个寄存器都带有括号，我们需要将（%eax）改成%eax:

```
movl %eax, 4(%esp)
```

第四行的错误是%sl 是一个名字错误的寄存器，参照寄存器表，这里的 sl 正确拼写应该为 si，同时注意到%si 为十六位而%al 是八位，我们需要改%al 为%ax，:

```
movw %ax, %si
```

第五行的错误是 mov 指令的源和目的操作数类型不匹配
“movl %eax, $0xFFFFFFFF”,指令中不能将一个寄存器中的数给一个立即数，我们需要将两者位置互换:

```
movl $0xFFFFFFFF,%eax
```

第六行的错误是 mov 指令的源和目的操作数类型不匹配:
movw %eax, %bx，%eax 是三十二位寄存器而 movw 和%bx 都是十六位，所以要把%eax 改成%ax:

```
movw %ax, %bx
```

第七行的错误是%si 和 movb 不匹配，因为%si 是 16 位寄存器但是 movb 的操作数应该是八位数据。我们将 movb 改成 movw。
至此，所有的错误都被改正:

```
.section .text
.global  main
 main:
    movb $0xF, (%ebx)
    movw %ax, (%esp)
    movl %eax, 4(%esp)
    movw %ax, %si
    movl $0xFFFFFFFF, %eax
    movw %ax, %bx
    movw %si, 8(%esp)
```

```
PS D:\Desktop\CS资料\tool> gcc -g -o 47_3 47_3.s
PS D:\Desktop\CS资料\tool> gcc -g -o 47_3 47_3.s
PS D:\Desktop\CS资料\tool>
```

再进行操作不会报错。

48_6

指令 1:

addl %ecx,(%eax) 将%ecx 的值加到%eax 值对应的地址中的值上面，我们看题目上面的表知道，此时%eax=0x100，而地址 0x100 对应的值为 0xFF，%ecx 的值为 0x1，两者相加得到更新后的地址 0x100 的值为 0x100.

指令 2:

Subl %edx,4(%eax) 指令的目的是将%edx 从 %eax 值对应寄存器中的值加上偏移量 4 后对应的地址中的值减去。%edx 此时的值为 0x3，%eax 值对应地址中的值为 0x100，得到的地址为 0x100+4 等于 0x104,查表得到该地址对应的值为 0xAB。故更新后的地址 0x104 的值为 0xA8.

指令 3:

Imull $16,(%eax,%edx,4)，指令的目的是从(%eax+%edx*4)得到地址，将地址对应的值乘以 16 之后写回原地址中。%eax=0x100,%edx=0x3,计算得到的地址为 0x10C，查表得到该地址对应的值为 0x11，计算后得到更新后的地址 0x10C 对应值为 0x176.

指令 4:

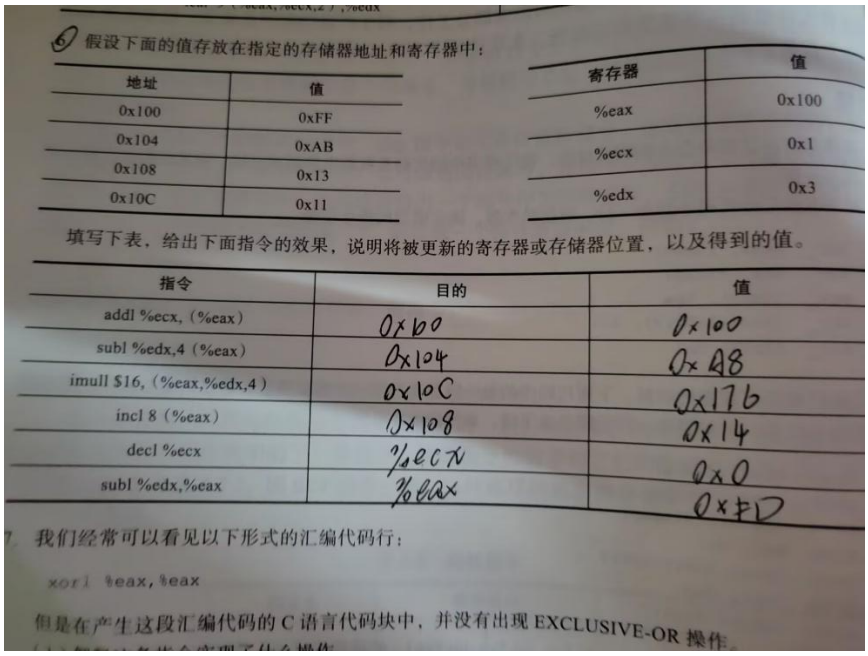
Incl 8 (%eax)，指令目的是将%eax 值加 8 后对应地址中的值加 1，地址为 0x100+8=0x108，查表得到该值为 0x13，加一后更新地址 0x108 的值为 0x14.

指令 5:

Decl %ecx，目的是将%ecx 的值减一，更新后的%ecx 的值为 0x0.

指令 6:

Subl %edx，%eax 指令目的是将%edx 从%eax 中减去，得到更新后的%eax 的值为 0x100-0x3=0XFD。



假设下面的值存放在指定的存储器地址和寄存器中：

地址	值
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

寄存器	值
%eax	0x100
%ecx	0x1
%edx	0x3

填写下表，给出下面指令的效果，说明将被更新的寄存器或存储器位置，以及得到的值。

指令	目的	值
addl %ecx, (%eax)	0x100	0x100
subl %edx,4 (%eax)	0x104	0xA8
imull $16, (%eax,%edx,4)	0x10C	0x176
incl 8 (%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx,%eax	%eax	0xFD

我们经常可以看见以下形式的汇编代码行：

```
xorl %eax,%eax
```

但是在产生这段汇编代码的 C 语言代码块中，并没有出现 EXCLUSIVE-OR 操作。

47_7:

1. 这条指令实现了操作将%eax 和自己做异或，每一位和自己异或后都变为了 0，从而将其

值变成 0.

2. 更直接地表达这个操作的汇编代码为将%eax 直接设置成 0：movl $0,%eax。

```
PS D:\Desktop\临时文件> as -o syb.o syb.s
PS D:\Desktop\临时文件> objdump -d syb.o

syb.o:     file format pe-i386


Disassembly of section .text:

00000000 <_main>:
   0:   31 c0                   xor    %eax,%eax
   2:   b8 00 00 00 00          mov    $0x0,%eax
   7:   90                      nop
```

3. 我们可以看到使用异或只需要 2 个字节而使用直接赋值需要 5 个字节。