

计算机系统第一次作业

1. (P. 33) 在本章所设计的原型系统中，实现两个正整数相乘（假设乘积不超过 127），并在虚拟程序中运行。

```
shared > ≡ c-inst.txt
1  8
2  #两个大于1的正数相乘
3  in R1      #乘数a
4  movb R0,R1 #乘数a存放到内存0000 0000
5  in R1      #被乘数b
6  movi 1
7  movb R0,R1 #被乘数b存放在内存0000 0001
8  |          #结果存放在内存 0000 0010
9  #开始循环
10 movi 1     #R0中的值为1
11 movc R1,R0 #从内存中取出值b
12 movi 1     #设置R0中的值为1
13 sub R1,R0  #R1即b值减1，此时设置G值
14 movi 1
15 movb R0,R1 #b值需要保存回去
16 movi 0     #R0中设置为0，即内存地址0
17 movc R2,R0 #取出a值
18 movi 2     #R0中设置为2，即内存地址0000 0010
19 movc R1,R0 #取出结果
20 add R1,R2  #做加法
21 movb R0,R1 #将结果存回去
22 movd      #保存当前的PC值到R3
23 movi -12   #R0的值设置为-12
24 add R3,R0  #R3的值加-12
25 jg         #如果第12行的减法设置G为1,就跳转
26 #循环结束
27 movi 2     #R0中设置为2，即内存地址0000 0010
28 movc R1,R0 #取出结果
29 out R1     #打印结果
30 halt
```

运行结果：

将两个正整数分别存在两块内存（0x00，0x01），将最终的结果保存在内存（0x10）。利用累加来实现两个正整数相乘。

```

xiao@长乐:/mnt/d/myVirtualbox/shared$ q
Command 'q' not found, but can be installed with:
sudo apt install python3-q-text-as-data
xiao@长乐:/mnt/d/myVirtualbox/shared$ ./vspm c-inst.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: 8个字节,0000 0000 ~ 0000 1000
装载指令..... OK!
      共25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 1000          in R1          #乘数a
VM> si
5
      0000 1001          movb R0,R1      #乘数a存放到内存0000 0000
VM> si
      0000 1010          in R1          #被乘数b
VM> si
3
      0000 1011          movi 1
VM> si
      0000 1100          movb R0,R1      #被乘数b存放在内存0000 0001
VM> si
      0000 1101          movi 1          #R0中的值为1
VM> si
      0000 1110          movc R1,R0      #从内存中取出值b
VM> si
      0000 1111          movi 1          #设置R0中的值为1
VM> si
      0001 0000          sub R1,R0       #R1即b值减1,此时设置G值

```

sub R1, R0, 设置状态寄存器 G, 用来控制循环。假如计算 $a*b$, 不妨将 b 看作被乘数。每次循环 $b-1$ 。当 $b \leq 1$ 时, $G=0$, 此时跳出循环; 当 $b > 1$ 时, $G=1$, 此时循环继续。

```

VM> si 0001 0001      movi 1
VM> si 0001 0010      movb R0,R1      #b值需要保存回去
VM> si 0001 0011      movi 0          #R0中设置为0, 即内存地址0
VM> si 0001 0100      movc R2,R0      #取出a值
VM> si 0001 0101      movi 2          #R0中设置为2, 即内存地址0000 0010
VM> si 0001 0110      movc R1,R0      #取出结果
VM> si 0001 0111      add R1,R2       #做加法
VM> si 0001 1000      movb R0,R1      #将结果存回去
VM> si 0001 1001      movd           #保存当前的PC值到R3
VM> si 0001 1010      movi -12        #R0的值设置为-12
VM> si 0001 1011      add R3,R0       #R3的值加-12
VM> si 0001 1100      jg              #如果第12行的减法设置G为1,就跳转
VM> si 0000 1101      movi 1          #R0中的值为1
VM> si 0000 1110      movc R1,R0      #从内存中取出值b
VM> si 0000 1111      movi 1          #设置R0中的值为1
VM> si 0001 0000      sub R1,R0       #R1即b值减1, 此时设置G值
VM> si 0001 0001      movi 1
VM> si 0001 0010      movb R0,R1      #b值需要保存回去
VM> si 0001 0011      movi 0          #R0中设置为0, 即内存地址0

```

将内存中的乘数和每次累加存放的中间结果取出,做加法,更新结果。

以（5*3）为例，此过程会循环 3 次。

```

VM> si      0001 0100      movc R2,R0      #取出a值
VM> si      0001 0101      movi 2          #R0中设置为2, 即内存地址0000 0010
VM> si      0001 0110      movc R1,R0      #取出结果
VM> si      0001 0111      add R1,R2       #做加法
VM> si      0001 1000      movb R0,R1      #将结果存回去
VM> si      0001 1001      movd          #保存当前的PC值到R3
VM> si      0001 1010      movi -12        #R0的值设置为-12
VM> si      0001 1011      add R3,R0       #R3的值加-12
VM> si      0001 1100      jg             #如果第12行的减法设置G为1,就跳转
VM> si      0000 1101      movi 1          #R0中的值为1
VM> si      0000 1110      movc R1,R0      #从内存中取出值b
VM> si      0000 1111      movi 1          #设置R0中的值为1
VM> si      0001 0000      sub R1,R0      #R1即b值减1, 此时设置G值
VM> si      0001 0001      movi 1
VM> si      0001 0010      movb R0,R1      #b值需要保存回去
VM> si      0001 0011      movi 0          #R0中设置为0, 即内存地址0
VM> si      0001 0100      movc R2,R0      #取出a值
VM> si      0001 0101      movi 2          #R0中设置为2, 即内存地址0000 0010
VM> si      0001 0110      movc R1,R0      #取出结果

VM> si      0001 0111      add R1,R2       #做加法
VM> si      0001 1000      movb R0,R1      #将结果存回去
VM> si      0001 1001      movd          #保存当前的PC值到R3
VM> si      0001 1010      movi -12        #R0的值设置为-12
VM> si      0001 1011      add R3,R0       #R3的值加-12
VM> si      0001 1100      jg             #如果第12行的减法设置G为1,就跳转
VM> si      0001 1101      movi 2          #R0中设置为2, 即内存地址0000 0010
VM> si      0001 1110      movc R1,R0      #取出结果
VM> si      0001 1111      out R1          #打印结果
VM> si      15          halt
      0010 0000

```

循环结束，累加结果就是相乘的结果。从内存（0x10）取出结果到寄存器 R1，输出 R1。

2. (P. 47) 当我们调用汇编器的时候，下面代码中的每一行都会产生一个错误信息，请在机器上运行，阅读错误提示信息，并解释每一行是哪里出了错，如何修改。

```
movb $0xF, (%ebx)
```

```
movw %eax, (%esp)
```

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

```
movb %al, %sl
```

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

```
movw %eax, %bx
```

```
movb %si, 8(%esp)
```

```
shared > code1 > ASM 4.3.s
1  .section .text
2  .global _start
3  _start:
4      nop
5      movb $0xF, (%ebx)
6      movw %eax, (%esp)
7      movl (%eax), 4(%esp)
8      movb %al, %sl
9      movl %eax, $0xFFFFFFFF
10     movw %eax, %bx
11     movb %si, 8(%esp)
12     int $0x80
```



```
xiaoye > shared > code1 > ASM 4.3.s
1  .section .text
2  .global _start
3  _start:
4      nop
5      movb $0xF, (%ebx)
6      movl %eax, (%esp)
7      movl %eax, 4(%esp)
8      movw %ax, %si
9      movl $0xFFFFFFFF, %eax
10     movw %ax, %bx
11     movw %si, 8(%esp)
12
13     movl $1, %eax
14     int $0x80
15

问题  输出  调试控制台  终端  端口
● xiaoye@长乐:~/shared/code1$ as -g 4.3.s -o 4.3.o --32
● xiaoye@长乐:~/shared/code1$ ld -o 4.3 4.3.o -m elf_i386
⊗ xiaoye@长乐:~/shared/code1$ ./4.3
Segmentation fault
○ xiaoye@长乐:~/shared/code1$

⊗ xiaoye@长乐:~/shared/code1$ as -g 4.3.s -o 4.3.o --32
4.3.s: Assembler messages:
4.3.s: Warning: end of file not at end of a line; newline inserted
4.3.s:6: Error: incorrect register `%eax' used with `w' suffix
4.3.s:7: Error: operand size mismatch for `mov'
4.3.s:8: Error: bad register name `%sl'
4.3.s:9: Error: operand type mismatch for `mov'
4.3.s:10: Error: operand type mismatch for `mov'
4.3.s:11: Error: `%si' not allowed with `movb'
```

4.3.s:5:并没有报错：它的作用是将 8 位立即数 0xF 存储到由寄存器 %ebx 指向的内存地址中。movb 指令是操作 8 位的寄存器/数据。但最后执行时会报段错误。因为 ebx 存储的地址为 0x0，而在现代操作系统中，地址 0x0 是被保护的，不能用于普通程序的读写操作。

4.3.s:5: 改正：改变 ebx 中的值；或者将立即数直接加入 b1 中。

4.3.s:6: 错误：使用了错误的寄存器 %eax'，它不能与 w' 后缀一起使用。

`movw` 表示指令操作的是 16 位寄存器/数据。`%eax` 是 32 位寄存器。

4.3.s:6: 改正: `movl %eax, (%esp)`。

这条指令的作用是将寄存器`%eax` 中的值移动到栈顶指针`%esp` 所指向的地址处。`%esp` 中的值不变，栈顶指针指向的地址中的值为`%eax` 中的值。

4.3.s:7: 错误: `mov` 指令的操作数大小不匹配。

在 x86 汇编语言中，`movl (%eax), 4(%esp)` 是一个无效的指令。`mov` 指令不能直接在两个内存地址之间移动数据。要实现从一个内存地址到另一个内存地址的数据传输，需要通过一个寄存器中转。

4.3.s:7: 改正: `movl %eax, 4(%esp)`

这条指令的作用是将寄存器`%eax` 中的值存储到栈顶指针`%esp` 所指向的地址偏移 4 字节的位置。

4.3.s:8: 错误: 寄存器名称 ``%sl'` 无效。

在 x86 汇编语言中，寄存器名称是固定的，`%sl` 并不是一个有效的寄存器名称。`%si` 是 x86 架构中的一个 16 位寄存器，全称为 **Source Index**（源变址寄存器）。同时，我们注意到，`movb %al, %si`，`movb`

只能操作 8 位寄存器/数据，`%al` 也是 8 位寄存器。为了避免出现“使用了错误的寄存器”和“`mov` 指令的操作数不匹配”的错误。最好使得指令和寄存器相匹配。

4.3.s:8: 改正: `movw %ax, %si/movb %al, %sil`

第一条指令的作用是将 16 位寄存器`%ax` 的值移动到 16 位寄存器 `%si` 中；第二题指令的作用是将 16 位寄存器`%ax` 的值移动到 16 位寄存

器 %si 的低 8 位中。【注意：%sil 只在 64 位模式下有效，在 32 位模式下无效。】

4.3.s:9: 错误： mov 指令的操作数类型不匹配。

\$0xFFFFFFFF 表示一个立即数 (Immediate Value)，是直接写在指令中的常量值。不可以作为 mov 指令的目的操作数。

4.3.s:9: 改正： movl \$0xFFFFFFFF, %eax

这条指令的作用是将 32 位的立即数移动到 32 位寄存器 eax 中。如果将 0xFFFFFFFF 作为内存地址，由于它是一个非常高的内存地址，可能超出程序的合法地址范围。

4.3.s:10: 错误： mov 指令的操作数类型不匹配。

movw 指令操作的数据大小为 16 位。movw 不能直接将 32 位寄存器 %eax 的值移动到 16 位寄存器 %bx 中。

4.3.s:10: 改正： movw %ax, %bx

这条指令的作用是将 16 位寄存器 %ax 的值移动到 16 位寄存器 %bx 中。

4.3.s:11: 错误： movb 不允许使用 %si。

movb 是 8 位操作，而 %si 是 16 位寄存器。因此，movb 不能直接操作 %si，因为大小不匹配。

4.3.s:11: 改正： movw %si, 8(%esp)

这条指令的作用是将 16 位寄存器 %si 的值移动到内存地址 8(%esp) 处。


```

xiaoYe > shared > code1 > ASM 4.3.s
1  .section .text
2  .global _start
3  _start:
4      nop
5      movb $0xF,%bl
6      movl %eax, (%esp)
7      movl %eax, 4(%esp)
8      movw %ax,%si
9      movl $0xFFFFFFFF,%eax
10     movw %ax,%bx
11     movw %si, 8(%esp)
12
13     movl $1,%eax
14     int $0x80
15

```

问题
输出
调试控制台
终端
端口

```

● xiaoYe@长乐:~/shared/code1$ as -g 4.3.s -o 4.3.o --32
● xiaoYe@长乐:~/shared/code1$ ld -o 4.3 4.3.o -m elf_i386
⊗ xiaoYe@长乐:~/shared/code1$ ./4.3
○ xiaoYe@长乐:~/shared/code1$ █

```

3. (P. 48) 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	寄存器	值
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x15	%edx	0x5
0x10c	0x11		

填写下表，给出下面的指令的效果，说明被更新的寄存器或存储器的位置，以及得到的值。

指令	目的	值
addl %ecx, (%eax)	将32位寄存器 %ecx 中的值与由寄存器 %eax 指向的内存地址中的值相加，并将结果存回内存。	0x0100
subl %edx, 4(%eax)	将从内存地址 4(%eax) 中的值减去 %edx 寄存器中的值，并将结果存回内存地址。	0x01A8
imull \$16, (%eax, %edx, 4)	将立即数 16 与内存地址 (%eax, %edx, 4) 中的值相乘，并将结果存回该内存地址。	0x010C
incl 8(%eax)	将内存地址 8(%eax) 中的值加 1，并将结果存回内存地址。	0x0108
decl %ecx	将32位寄存器 %ecx 中的值减 1，结果保存在32位寄存器 %ecx 中。	0x0
subl %edx, %eax	从32位寄存器中减去32位寄存器 %edx 的值，并将结果存回 %eax 中。	0x01FD

①

(%eax) → 0x100 → 0xFF

%ecx → 0x1

0000 1111 1111

+ 0000 0000 0001

0001 0000 0000

→ 0x0100

②

(%eax) → 0x100 → 4(%eax) → 0xAB

%edx → 0x3

减法 = 补码加法

0xAB - 0x3 = 0xAB + (0x3)补

0x3 = 00000011 补 → 11111101

10101011

+ 11111101

11010100

→ 0x01A8

③

(%eax, %edx, 4) → (%eax + %edx · 4) →

0000 0001 0000 0000

+ 0000 0000 0000 1100

0000 0001 0000 1100

→ 0x010C

④

8(%eax) → 0x100 + 8 →

0001 0000 0000

+ 0000 0000 1000

0001 0000 1000

→ 0x108

⑤

%ecx → 0x1 → -1 → 0x0

⑥

%eax → 0x100

%edx → 0x3

减法 = 补码加法

0x3 = 00000011 补 → 11111101

0001 0000 0000

+ 0000 1111 1101

0001 1111 1101

→ 0x1FD

上述计算过程中，主要注意：

(1) 不妨以 `eax` 为例，`(%eax)` 是间接寻址的方式，表示从寄存器 `eax` 所指向的内存地址中读取或写入数据；`%eax` 是寄存器寻址的方式，

直接使用寄存器的值作为操作数，读取或者写入寄存器。

(2) 立即数寻址，直接将立即数作为操作数。如表格中第 3 条 `imull $16, (%eax, %edx, 4)`。

(3) 基址偏移量寻址，将寄存器指向的内存地址作为基地址，在此基础上，加上偏移量。如表格中第 2 条 `subl %edx, 4(%eax)`。目的操作数的地址=`%eax` 指向地址+4，0x104。再如表格中第 4 条 `incl 8(%eax)`。目的操作数的地址=`%eax` 指向地址+4，0x108。

(4) 变址寻址，其中变址寻址又包括变址基址寻址、比例变址寻址。变址寻址是指操作数的地址=基址寄存器的值+索引寄存器的值的和所指向的地址。在此基础上，加上偏移量，就是变址基址寻址；在此基础上，加上比例因子（即操作数的地址=基址寄存器的值+索引寄存器的值*比例因子的和所指向的地址），就是比例变址寻址。比如表格中第 3 条 `imull $16, (%eax, %edx, 4)`。目的操作数的地址 = $(\%eax + \%edx * 4)$ 。

(5) 在计算机中，减法的计算通过补码来实现。比如， $A-B=A+B$ 的补码。

4. (P. 48) 我们经常可以看见以下形式的汇编代码行：

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

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

(1) 解释这条指令实现了什么操作？

答：这条指令用于对同一个寄存器%eax 进行异或操作。它的作用是将%eax 的值与自身进行异或运算，并将结果存储回%eax。根据异或操作的定义，相同的数异或结果为 0，反之为 1。故本质上实现对%eax 的清零操作。

(2) 更直接表达这个操作的汇编代码是什么？

答：movl \$0,%eax

(3) 比较同一个操作的两种不同实现的编码字节长度。

答：

指令	功能	编码长度	机器码
xorl %eax,%eax	将%eax 清零	2 字节	31 C0
movl \$0,%eax	将%eax 清零	5 字节	B8 00 00 00 00

```
• xiaoye@长乐:~/shared/code1$ objdump -d 4.4

4.4:      file format elf32-i386

Disassembly of section .text:

08049000 <_start>:
 8049000:      90                nop
 8049001:      31 c0             xor     %eax,%eax
 8049003:      b8 00 00 00 00    mov     $0x0,%eax
 8049008:      b8 01 00 00 00    mov     $0x1,%eax
 804900d:      cd 80             int     $0x80
• xiaoye@长乐:~/shared/code1$ S
```

总结：

- xorl %eax, %eax:
 - 编码长度为 2 字节，更短。

- 执行效率高，因为它不需要加载立即数。
- 在性能敏感的代码中，这种实现方式更优。
- **`movl $0, %eax:`**
 - 编码长度为 5 字节，更长。
 - 更直观，易于理解。
 - 在代码可读性更重要的场景中，这种实现方式更优。