SEAL (Simple Educational Assembly Language) 教学说明文档

*计算机科学导论 – 以 **Python** 为舟*（沙行勉，清华大学出版社）第三章的辅助工具。一种为了教学而用的汇编语言的模拟器。欢迎使用。

沙行勉

目录

1. 概述

由沙行勉教授主编的《计算机科学导论-以 Python 为舟》第三章指出读者应该理解程序如何在计算机里执行。在真实计算机上，全部了解这些内容可能需要读者掌握操作系统、计算机组成原理、计算机体系结构及指令集，这会使读者陷入复杂的计算机世界中无法自拔。因此，我们设计并用高级程序设计语言 Python 实现了一个汇编语言模拟器 SEAL。SEAL 实现了 24 条“高级”汇编语言指令，同时模拟了容量为 10000 的内存以及 18 个寄存器，其中有 16 个 64 位通用寄存器 R0-R15，可以存储数据的大小在-2^{63}~$2^{63}-1$范围内支持十进制整型数和十六进制数（注意十六进制表示时数字尾部需要加上“H”或“h”，其中十六进制中使用到的 A~F，不区分大小写），1 个 pc 指令寄存器，1 个 sp 堆栈指针寄存器。

2. 模拟器的使用

同学从清华大学的官网上下载“SEAL 汇编语言模拟器.zip”文件解压后会看到“SEAL 汇编语言模拟器”文件夹，进入该文件夹将会看到：一个“教学示例”文件夹，以 eg 开头、以.txt 结尾的 9 个示例文件（本节主要使用 eg1_if.txt 进行讲解），SEAL 教学说明文档(1.1 版本).docx 文件，SEAL 用户手册(Word)(1.1 版本).docx 文件，simulator1.1.py 文件，同学解压文件夹中的内容如图 2-1 所示。

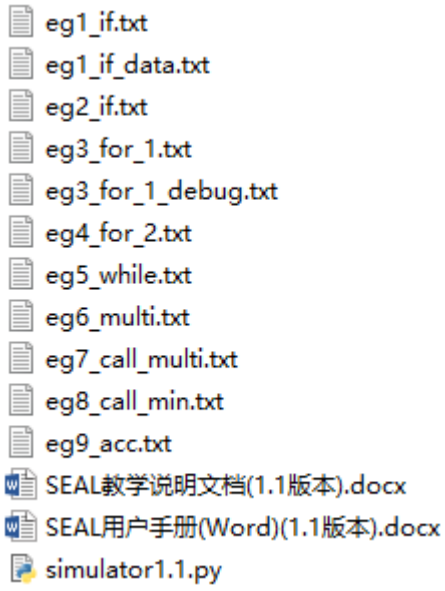


图 2-1 SEAL 汇编语言模拟器文件夹内容

- **启动 simulator.py**

simulator.py 即该使说明书所说的汇编语言模拟器 SEAL。假设同学使用 IDLE 启动模拟器，IDLE 是 Python 软件包自带的一个集成开发环境，同学可以利用它方便地创建、运行、测试和调试 Python 程序。在“开始”—>“所有应用”—>“Python 2.x/Python 3.x”—>“IDLE(Python)”即可启动 IDLE，启动 IDLE 之后，其图形用户界面如图 2-2 所示。

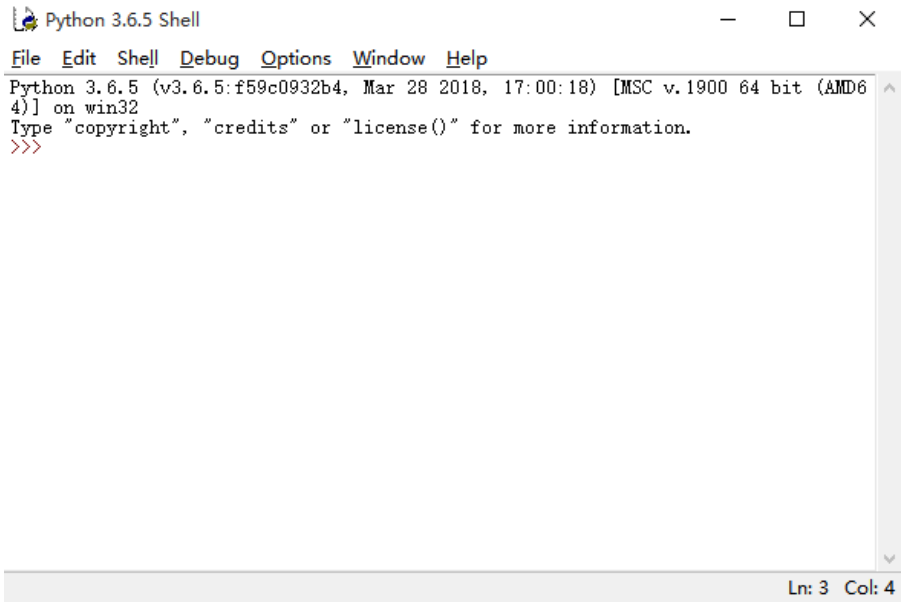


图 2-2 IDLE 启动界面

接着，在图 2-2 所示的界面菜单栏点击“File”—>“Open”，在弹出框中选择“simulator.py”文件，接着会弹出图 2-3 所示界面。

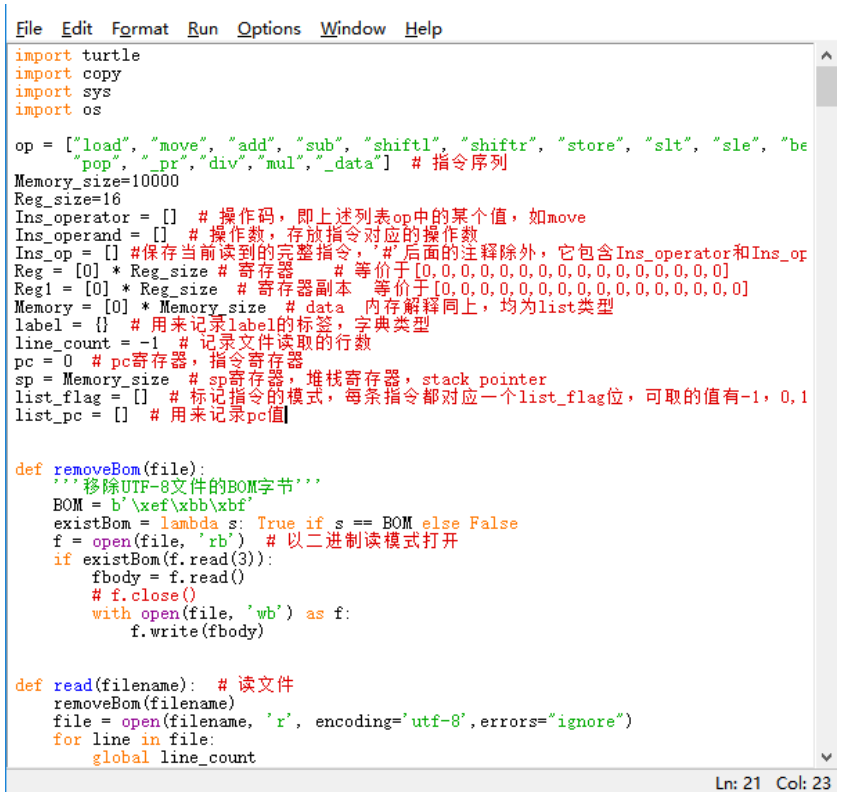


图 2-3 打开 simulator.py 文件

图 2-3 中显示内容为 simulator.py 文件的源代码（即 SEAL 汇编语言模拟器源代码），我们开放源码，供同学学习以及修改。接着在图 2-3 的菜单栏中看到“Run 按钮”，点击它会看到“Run Module F5”,继续点击“Run Module F5”，同学会

看到图 2-4 所示内容。其中我们看到“请输入文件名(不在同目录下请输入完整路径)/输入“exit”退出：”字样，此时表示 SEAL 模拟器已经成功启动。

图 2-4 SEAL 成功启动

- **示例展示**

本节使用 eg1_if.txt 示例进行模拟器运行的讲解，先给出该示例的 python 程序以及汇编程序，以便同学对汇编程序有一定的了解和直观感受，对于本示例的具体指令介绍以及指令执行过程将会在第 3 节讲解。

```
#<python 代码：两个数字比较大小，并将二者中较小的数字输出>
a = 5
b = 7
if a <=b:
    print(a)
else:
    print(b)
```

```
#<汇编代码：两个数字比较大小>
#输入两个数字到 R0，R2,输出到 R1
mov R0,5        #a = 5
mov R2,7        #b = 7
sle R3,R0,R2
beqz R3,L1
mov R1,R0
goto L2
L1:
mov R1,R2
L2:
_pr R1
```

该示例是对两个数字比较大小，输出较小的数字，其中两个数字是 5 和 7，最后希望输出的结果是 5。要对该汇编程序进行运行，需要通过以下几个步骤：

- **输入汇编程序的文档名**

输入汇编程序的文档名（例如 acc.txt），当然这里的文档名可以自己根据所写功能以及自己的喜好进行命名，此处的文档名的后缀一定要是.txt（**注意，请同学在自己的 Windows 系统中的“文件扩展名”选项前打勾**），并且文档的编码格式最好是 utf-8 无 bom 格式，其它编码格式也可以，只是会有乱码的可能，不过不影响使用。注意这里的文档要和模拟器的程序放在一个文件夹下才可以直接输入文档名，否则需要加上汇编程序所在文件的路径（例如 E:\python\test.txt，亦即 test.txt 文件所在的完整路径），此时也可以选择输入 exit 退出模拟器或者选择运行模式。

如果输入的文档名有错，模拟器会提示“文件不存在，请您输入正确的文件名(不在同目录下请输入完整路径)/输入“exit”退出:”，此时请同学再次输入文件名，并确保文件的确存在，同学亦可选择“exit”退出模拟器。实验中所有指令全部忽略大小写，即当输入 Exit 时，仍然会退出模拟器。如果同学输入正确，会提示“请选择模式（输入“normal”进入普通模式/“debug”进入调试模式/“exit 退出调试”）：”。

根据以上输入文档名的介绍，所以我们应该在图 2-4 的界面输入 eg1_if.txt，输入文档名之后显示如图 2-5：

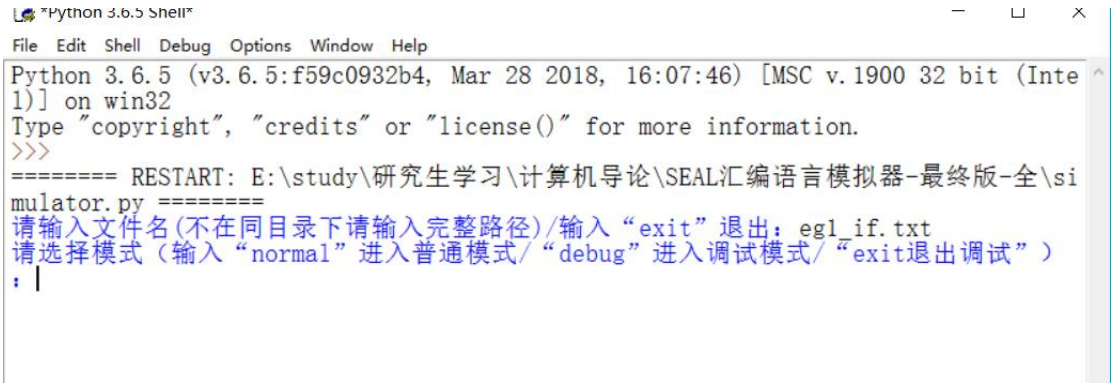


图 2-5 输入汇编程序文件名

由于只需要得到该汇编程序的计算结果，所以直接输入“normal”便可以输出计算结果，输出的结果如图 2-6 所示，正是我们所希望输出的结果。

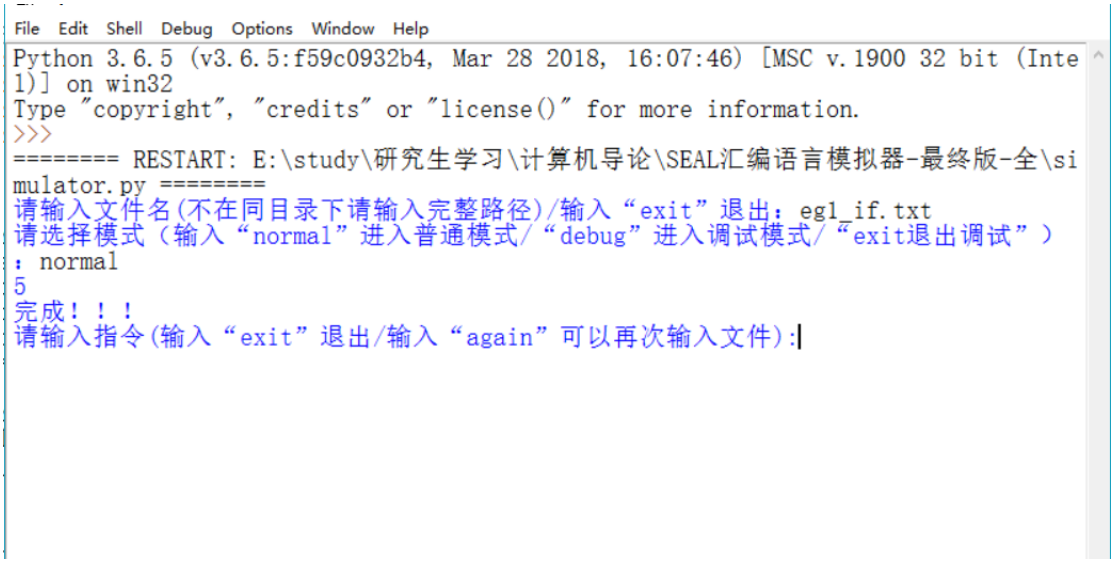


图 2-6 执行 normal 后的结果

图中的“debug”进入调试模式，具体会在第 4 部分进行讲解。

在得到计算结果之后，便可以输入“exit”退出 SEAL。

3. 汇编程序及指令的介绍

3.1 程序例一：两个数字比较大小，并将二者中较小的数字输出

```
#<python 代码：两个数字比较大小，并将二者中较小的数字输出>
a = 5
b = 7
if a <=b:
    print(a)
else:
    print(b)
```

```
#<汇编代码：两个数字比较大小>
#输入两个数字到 R0，R2,输出到 R1
mov R0,5        #a = 5
mov R2,7        #b = 7
sle R3,R0,R2
beqz R3,L1
mov R1,R0
goto L2
L1:
mov R1,R2
L2:
_pr R1
```

该示例是对两个数比较大小，并将较小的数输出。大家会疑惑，这些汇编指令究竟是什么意思？为什么就可以实现两个数的大小比较呢？

要实现比较两个数字的大小，整体的思路应该为：

先将需要比较的两个数字分别赋值给两个变量，这两个变量的值是存储在寄存器 R0、R2 中的，所以要分别对两个寄存器赋值。这里使用 mov 指令，该指令的功能是赋值操作，给寄存器 R 赋一个值，可以有两种格式：①mov R0,constant ②mov R0,R1。这两种格式均是将第二个操作数赋值给第一个操作数，本示例中使用的是格式①，直接用一个常数给寄存器赋值，而格式②是将后一个寄存器中的数值赋值给前一个寄存器。该示例中两条 mov 指令实现了对两个变量 a，b（即寄存器 R0，R2）的赋值。

然后对两个寄存器中的值进行大小判断，并将判断结果存储到寄存器 R3 中，根据 R3 的值决定接下来的操作指令（即要将哪个数输出）。使用指令“sle

R3,R0,R2”判断寄存器 R0 中的数值是否小于等于寄存器 R2 中的数值，并当 R0 中的数值小于 R2 中的数值时给寄存器 R3 赋值 1，否则赋值 0。sle 指令可以有两种格式：①sle R3,R2,R1 ②sle R3,R2,constant。格式①是将后两个寄存器中的数值进行小于等于的比较，而格式②是将后一个寄存器中的数值与常数进行小于等于的比较。该示例中 sle 指令实现了对两个数的大小比较，并将寄存器 R3 的值赋为了 1（因为 5<7）。

另：相应的还会有一个判断是否小于的 slt 指令，slt 指令与 sle 的用法大同(书写格式同)小异(在第二个操作数小于第三个操作时对第一个操作数赋值 1，否则赋值 0)。

接下来根据寄存器 R3 的值决定后续需要执行的指令。如果 R3 为 0，则说明 R0 小于等于 R2，需将 R0 存储在 R1 中；如果 R3 为 1，则说明 R0 大于 R2，需将 R2 存储在 R1 中。使用指令“beqz R3,L1”，它的功能是第一个操作数等于 0 则跳转到标签（即第二个操作数）所标记的指令块处执行，否则顺序执行。该示例中的 R3 中的数值是 1，所以不进行跳转，将顺序执行。

其中 L1 是个标签，用来标记语句块,标签格式为“L:”，其中 L 与冒号之间可以用其它任意的字符串来区分，例如使用数字 1，即 L1:，例如求最小值的函数，标签可以定义为 Lmin：。

该示例中，我们不希望将 R2 中的数值输出，所以我们需要跳过将 R2 中数值给 R1 赋值的指令，所以使用“goto L2”指令进行跳转。指令“goto L2”是无条件直接跳转到 L2 标记的指令块处执行。

最后，执行指令_pr 将结果打印输出（即 L2 标记的指令块）。该示例中执行“_pr R1”指令后，打印出较小数值：5。对于_pr 指令，后边可以有多个操作数，依次将它们打印输出。

明白了该示例中的每条指令的含义及程序的执行过程之后，我们给出该程序的执行流程图如图 3-1 所示：

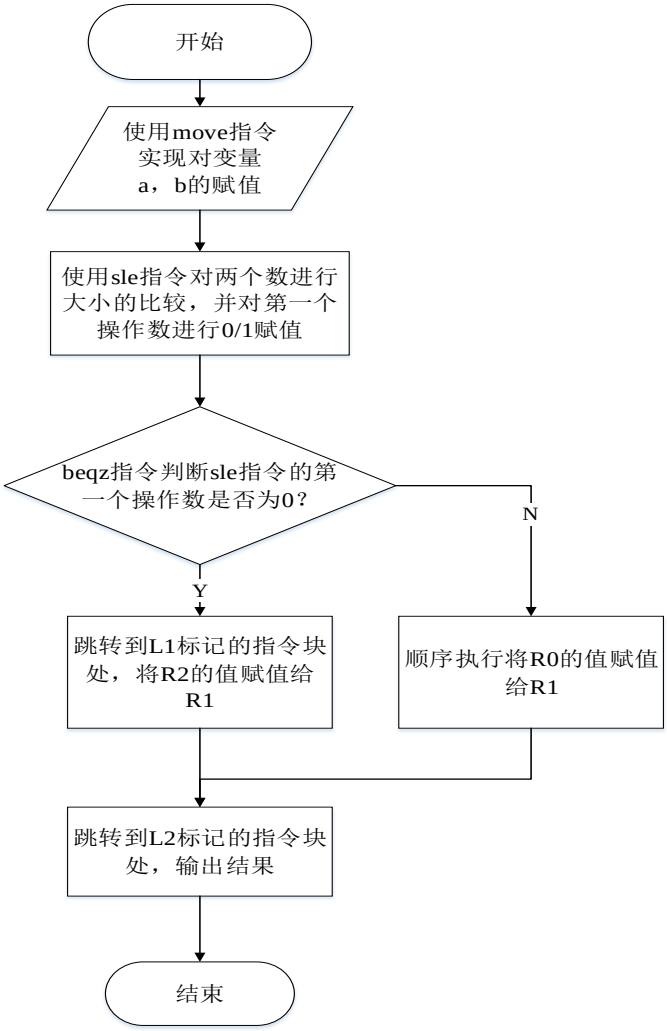


图 3-1 两个数字比较大小执行顺序

3.2 程序例二：for 循环求 a0+a1+a2+…+a9

```
#<python 代码：for 循环求 a0+a1+a2+…+a9>
a = [84,65,21,74,58,46,37,88,97,10]
s = 0
for i in range(len(a)):
        s = s+a[i]
print(s)
```

```
#<汇编代码：for 循环求 a0+a1+a2+…+a9>
_data 1,[84,65,21,74,58,46,37,88,97,10]   #(1)
mov R1,0 #存结果      (2)
mov R2,0 #计数        (3)
mov R5,1 #存储首地址     (4)
L1:                   # (5)
slt R3,R2,10          # (6)
```

```
beqz R3,L2                  #(7)
load R4,00(R5)              #(8)
add R5,R5,1     #每次所取数据的地址都会加 1          (9)
add R1,R1,R4          #(10)
add R2,R2,1           #(11)
goto L1               #(12)
L2:                   #(13)
_pr R1                #(14)
```

该示例是使用 for 循环对 a0，a1…a9 十个数进行求和运算，该示例中有些指令已经在示例一中进行详细介绍，在本示例中不再进行详细介绍，只对新出现的指令进行介绍。

首先需要将 a0，a1…a9 这 10 个数字存储，所以使用_data 指令进行存储，“_data 1,[84,65,21,74,58,46,37,88,97,10]”指令是将[]中的数据依次存储在从首地址为 1 开始递增的内存中，其中首地址可以根据自己需要进行设定，例如从 0 开始、从 100 开始。本示例中该指令实现了将[84,65,21,74,58,46,37,88,97,10]依次存储在地址为 1，2，3，…，10 的内存处。

同时，还要有存储结果、计数、存储首地址的三个变量，所以使用三条 mov 指令分别对寄存器 R1、R2、R5 赋初始值 0、0、1。

然后需要有一个 for 循环依次将十个数字加起来以及判断循环结束的条件，在汇编中可以用 sle 或者 slt 指令配合 beqz 指令实现。该示例中是使用指令“slt R3,R2,10”和指令“beqz R3,L2”设置的循环终止条件，由于一共有 10 个数进行求和且计数从 0 开始，所以将计数的寄存器 R2 与 10 进行小于比较，当计数小于 10 的时候，R3 的值是 1，执行 beqz 指令后将不会跳转；当计数等于 10 的时候，说明数据求和结束，执行 beqz 指令将会跳转到 L2 标记的指令块处。

在执行 beqz 指令不跳转的情况下，需要将当前数据拿出，加到前边数据求和的结果上。这里使用指令“load R4,00(R5)”将内存中的数据加载到寄存器中。该指令有两种格式：①load R1,offset(R2)②load R1,(address)。格式①是将后一个操作数的寄存器中的数加上偏移量所得到的内存地址处的数加载到前一个操作数的寄存器中，格式②是第二个操作数直接就是内存地址，将该地址处的数据加载到寄存器中，例如 load R1,(500)表示从内存地址为 500 处取出数据存储在寄存器 R1 中。该示例中的 load 指令是格式①，将寄存器 R5 中的数据加上偏移量 0

得到内存地址，取出该地址处的数据加载到寄存器 R4 中。

另：加载指令对应的还会有存储指令，store 指令便是将数据存储到内存处的指令，也是有两种格式：①store (address),R1 ②store offset(R2),R1。表示将后一个操作数的寄存器中的数据存储到前一个操作数所表示的内存地址处。

在取出当前要进行求和的数据之后，下一次希望取到当前数据的下一个数据，所以需要将地址加 1；然后将当前数据加回到结果中并且计数加 1，表示又进行了一次数据的加法。该示例中进行加法操作的是 add 指令。add 指令有两种格式：①add R1,R2,R3②add R1,R2,constant。其中第(9)和(11)条是使用的格式②，第(10)条使用的是格式①。将后两个操作数的进行求和赋值给第一个操作数的寄存器。

另：除 add 指令进行加法运算之外，还有 sub、mul、div 三条指令，分别是减法、乘法、除法运算，格式与用法与 add 指令完全相同。需要注意的是在 div 除法指令中，结果得到是整数商，例如 7/3=2。

在对当前数据求和之后，需要重新判断是否将所有数据都已计算完毕，所以需要跳转到 for 循环的判断语句处，这里将循环语句块标记为 L1，使用 goto L1 指令跳转。

最后在循环结束时，要将结果输出。beqz 指令会跳转到 L2 标记的打印输出指令处执行指令“_pr R1”将求和结果输出。

至此，本示例中新的指令以及执行过程已经介绍完毕，接下来给出该示例的执行过程流程图，如图 3-2 所示：

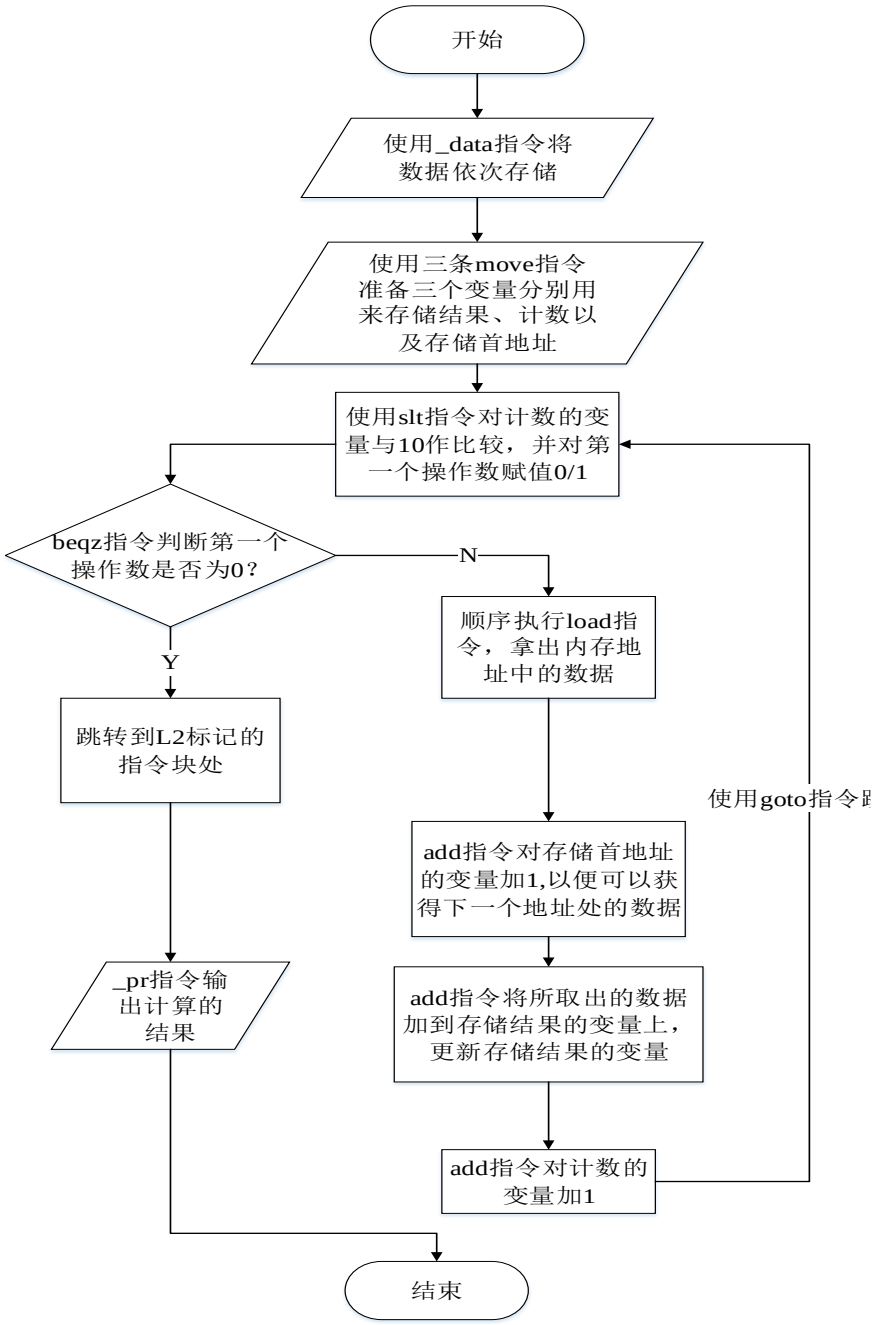


图 3-2 for 循环求 a0+a1+a2+…+a9 执行顺序

3.3 程序例三：while 求 a+d 大于等于 100 时，a 与 i 的值

```
#<python 代码：while 求 a+d 大于等于 100 时，a 与 i 的值>
a = 1
d = 1
i = 0     #计数
while a<100:
      a = a+d
      d = d+1
      i = i+1
```

```
print(i,a)
```

```
#<汇编代码：while 求 a+d 大于等于 100 时，a 与 i 的值>
mov R1,1     #a
mov R2,1     #d
mov R3,0     #i

L1:
slt R4,R1,100
beqz R4,L2
add R1,R1,R2
add R2,R2,1
add R3,R3,1
goto L1
L2:
_pr R3,R1
```

该示例中的所有指令在程序例一与程序例二中均已经介绍过，所以相信大家是可以看懂这汇编程序。该示例是在给出第一个数 a=1，每次以 d=d+1（d 的初始值为 1）的长度增加，还有一个用来计数的变量 i（初始值为 0），统计 a 增加到大于 100 时候需要加多少次 d。那现在我们给出该程序的执行顺序，以便大家能够判断自己所理解的程序执行顺序是否正确。

①该程序需要有三个变量 a、d、i。使用 mov 指令进行三个变量的赋值，也就是将初始值 1、1、0 分别存储在寄存器 R1、R2、R3 中。

②其中 a = a+d，所以判断 a 是否大于等于 100，便是判断 a+d 是否大于等于 100，这里的 while 循环也是将 slt 或 sle 配合 beqz 使用，作为循环结束的判断条件。该示例使用 slt 指令进行 a 与 100 的大小比较，beqz 对比较结果进行判断是否需要跳转进行结果的输出。第一次执行时 a 是 1，所以是小于 100，R4 赋值为 1，不进行跳转。

③三条 add 指令做 a = a+d、d = d+1、i = i+1 操作。

④goto 指令跳转到 L1 处，依次执行步骤②③。

⑤重复②③④步，直至 a 大于 100，通过判断循环结束的 slt 和 beqz 指令跳转到 L2 处，输出结果 i 和 a。

为了对执行过程更加明了，我们给出程序的执行流程，如图 3-3 所示：

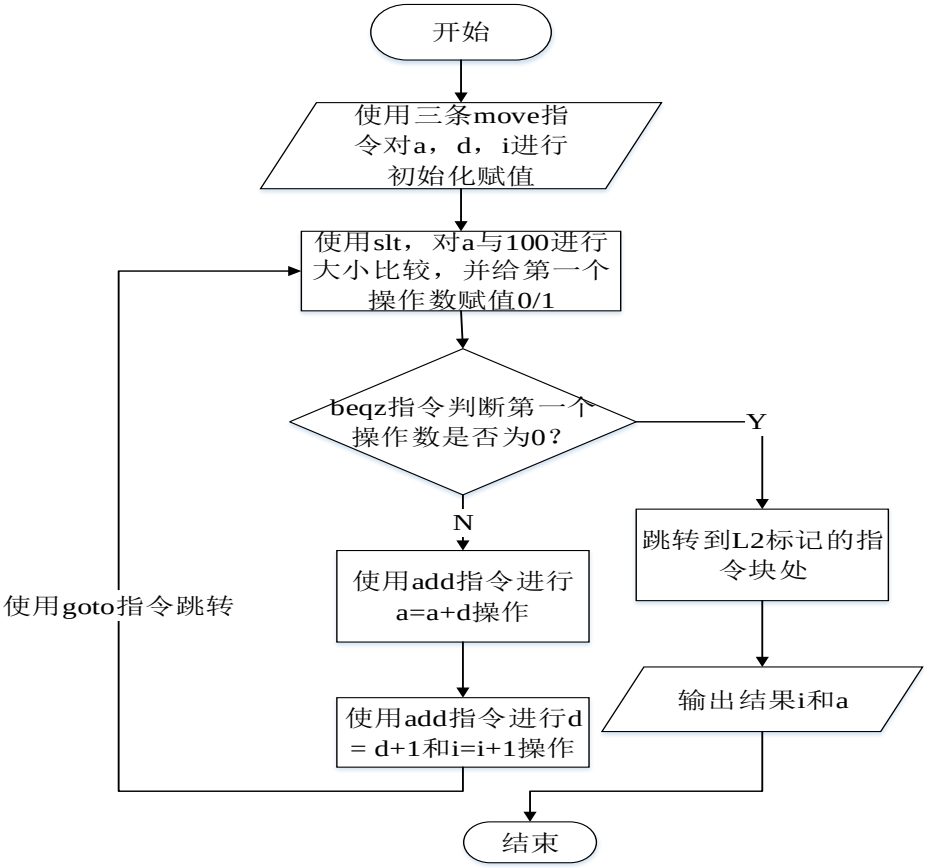


图 3-3 while 求 a+d 大于等于 100 时，a 与 i 的值的执行过程

3.4 程序例四：两个数做乘法运算

```
#<python 代码：两个数的乘法>
#python 可以直接使用*进行两个数的乘法，例如 c=a*b
#我们也可以使用移位的方法来计算正数乘法，如下
a = 4
b = 8
c = 0
while b > 0:
    if (b & 1 == 0):
        b = b >> 1
        a = a << 1
    else:
        c = c + a
        b = b >> 1
        a = a << 1
print(c)
```

```
#<汇编代码：两个数的乘法>
mov R1,4        #a = 4
```

```
mov R2,8          #b = 8
mov R4,0          #设置最终结果的初值为 0，c
L1:
and R3,R2,1
beqz R3,L2
add R4,R4,R1      #使用寄存器 R4 存储每次做加法后的值，计算结束时为最终结果
L2:
shiftr R2,R2,1
shiftl R1,R1,1
sle R5,R2,0
beqz R5,L1
_pr R4
```

该示例是对两个数做乘法运算，其中有三条新的指令（and、shiftr、shiftl）我们在前边的例子中没有遇到过，它们均是对位操作的指令，我们将会对这两个新的指令进行介绍。

(1) 计算两个数的乘法，首先需要有被乘数和乘数，同时还需要有变量来存储乘积，所以这里使用三条 mov 指令分别对被乘数、乘数和乘积进行赋值，乘积的初始值为 0。

(2) 计算乘法是由基本的二进制加法和移位操作进行的，对乘数的从低到高的每一位进行判断是 0 还是 1。判断前需要将乘数和 1 进行按位与操作获得此时乘数的低位的值，这里使用指令“and R3,R2,1”，它是将后两个操作数进行按位与操作，并将结果赋值给第一个操作数，该指令有两种格式：①and R3,R2,R1 ②and R2,R1,constant。该示例中使用的是格式①。

另：对应的还有 or、xor 指令，分别是按位或、按位异或操作，与 and 指令的用法完全相同。

(3) 得到乘数低位的数值之后，需要判断是否为 0，于是用 beqz R3,L2（这里使用 L2 标记移位的一系列操作）指令进行判断。当 beqz 指令判断乘数的最低位为 1 的时候，便需要将被乘数加一次到存储结果的变量上，这里使用 add 指令将乘数加回乘积的寄存器中。再对被乘数进行左移位操作，乘数进行右移位操作以便获取低第 2 位的值。如果为 0，则不做加法操作，直接进行移位的操作。对于移位操作，使用“shiftr R2,R2,1”对乘数进行右移位和“shiftl R1,R1,1”对被乘数进行左移位。这两条指令都是有两种格式，先给出 shiftr 的两种格式：①shiftr R1,R2,R3 ②shiftr R1,R2,constant。shifitl 的指令格式与其完全相同。表示对第二个操作数进行向右或向左移位第三个操作数所给出的位数，将移位结果存储在第

一个操作数中。该示例使用的是格式②。

(4) 在对乘数移位之后，需要判断此时的乘数是否为 0，为 0 则表示乘数已经移位结束，将要结束计算，否则要重复步骤(2)(3)(4)直至乘数为 0。这里使用 sle R5,R2,0 和 beqz R5,L1（L1 标记的是对乘数的低位获取的操作）指令进行判断。当 R5 为 0 的时候说明乘数不为 0，则跳转到 L1 处；否则说明乘数为 0，执行_pr R1 输出计算结果。以上的几条新指令均是对位的操作，对于位的操作是在二进制的情况下进行的，所以我们给寄存器赋值的虽然是十进制数，但是在实际执行该指令的时候是将其转化成二进制进行操作的，将操作的结果再转化为十进制。

新指令介绍完毕，我们给出该程序示例的执行过程流程图如 3-4 所示：

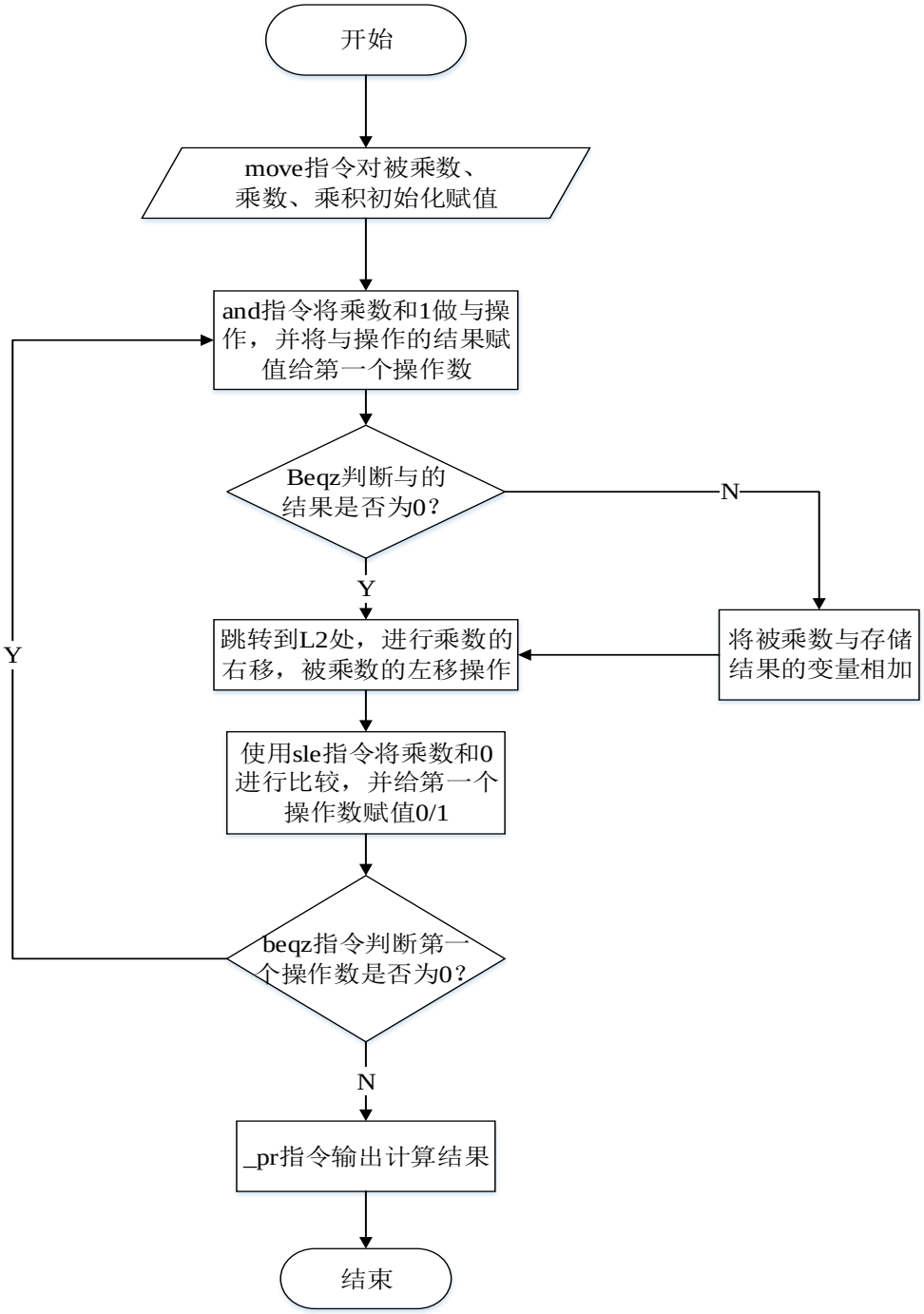


图 3-4 两个数的乘法执行过程

3.5 程序例五：函数调用对两个数求和

```
#<python 代码：函数调用两个数求和>
def add(a,b):
    c = a+ b
    return c

if __name__=="__main__":
```

```
    x = 5
    y = 6
    print(add(x,y))
```

```
#<汇编代码：函数调用两个数求和>
mov R15,10000     #R15 表示 fp，fp = 10000
mov sp,R15        #sp = fp
sub sp,sp,2       #sp 从 10000 往下开辟 3 个空间给局部变量 x，y，sp = sp-2

mov R2,5          #x = 5
mov R3,6          #y = 6
store -1(R15),R3
store -2(R15),R2

push R3   #传参数 b
push R2   #传参数 a

call Ladd         #调用函数 add(a,b),返回值存在 R1 中
goto Lprint

#add 函数有两个参数 a，b，将和放到 R1 中返回
Ladd:   #add(a,b)
push R15          #将旧的 fp 值压入栈内
mov R15,sp        #新的 fp = sp
sub sp,sp,1       #留一个空间，存放局部变量 c
push R2           #在函数中被更改，所以先存入栈内，在 return 之前会 pop 出来这个值
push R3           #在函数中被更改，所以先存入栈内，在 return 之前会 pop 出来这个值
load R2,2(R15)    #R2 = a
load R3,3(R15)    #R3 = b
add R1,R2,R3
store -1(R15),R1      #存放 c

Lreturn:
pop R3            #返回初始的 R3
pop R2            #返回初始的 R2
mov sp,R15        #sp = fp
pop R15           #重置 fp，成为旧的 fp
ret

Lprint:
_pr R1
```

该示例是使用函数调用对两个数求和，我们假设调用函数称作主函数，被调用函数称作子函数，主函数里会调用 add 子函数对两数求和。在函数调用中，需

要用到对栈操作的指令 push，pop 以及函数调用的指令 call 和返回的指令 ret。所以在该示例中，除了这四条指令是新指令，其余指令我们在前边的示例中已经使用过。在函数调用中，在子函数开始执行前，需要在栈的顶部建立一个栈帧(frame)，基本上，sp 指针指向栈帧的顶部，fp 指针指向栈帧的底部(sp 和 fp 是两个寄存器)。在一个 frame 中所存的包含主函数所传的参数值、函数内的所有局部变量、函数返回的 pc 值(也就是主函数调用子函数后的下一条指令的位置)，以及主函数的 fp 值。总而言之，一个函数在执行前必须要将现在函数的 fp 及 sp 建立起来。这个过程叫做函数的连接(linkage)。我们现在所使用的每一个 CPU 都有它的 linkage 规则，我们在 SEAL 中所使用的是类似于 x86 的标准 linkage 规则。

在 SEAL 中我们假设把 R15 用作 fp，sp 是一个特殊的寄存器。你也可以将其他的寄存器设为 fp，只要在编译函数时有一个统一的规则就好了。sp 的值可以用汇编指令 ADD 和 SUB 来更改，也可以用 push 和 pop 指令来做加 1 或减 1 的更新。当 push R 时，该指令将 sp 减 1，然后将寄存器 R 的值存入 sp 所指的地址；pop R 时此指令会将 sp 所指地址的值 load 进寄存器 R，然后将 sp 加 1。我们先假设主函数已经有一个 frame 了，栈底是 fp，栈顶是 sp，如图 3-5(a)所示。在主函数中有两个局部变量 x 和 y，x 的地址-2(R15)，y 的地址是-1(R15)，大家可以参看主程序中关于 x=5 和 y=6 的汇编代码。

接下去主函数要调用子函数 add(x,y)，我们要在这里详细描述函数调用是新的栈帧建立的过程。

一、参数的传递

将参数以反向的顺序 push 进 stack，所以先 push y 的值，再 push x 的值，如图 3-5(b)所示。

二、执行 call 指令

call 指令会将 pc 值 push 进入 stack，如图 3-6(a)所示，这个 pc 值指向 call Ladd 的下一条指令，也就是函数执行完后返回的地址，然后 goto Ladd(就是把 pc 值设成 Ladd 的地址)。

三、函数起始的三条指令

这三条指令是所有函数开始时都会有的三条类似的指令：①push R15, 将主函数的 fp 值存入栈中；②mov R15, sp，将新的 fp 指向 sp 的位置，也就是 fp 指向此时栈的顶端，如图 3-6(b)所示；③sub sp,sp,1，将 sp 再往上移，留出局部变

量的空间，此 add 函数只有一个局部变量，所以只要留一个位置，假如有 n 个局部变量，sp 就要-n。经过这三个指令之后，栈的状态如图 3-6(c)所示。

四、add 函数中的计算

一个函数的栈帧建立后，fp 会固定住，sp 会随着函数中的 push、pop 指令而更改，所以我们通常是用 fp 做基准位置来得到参数或函数内的局部变量的地址。在 add 函数中的参数 a 的地址是 2(R15)，参数 b 的地址是 3(R15)，局部变量 c 的地址是-1(R15)，请参考汇编语言代码的相关 load, store 语句。函数的结果用 R1 传回主函数。

五、函数结束的三条指令

这三条指令会将栈帧返回主函数的栈帧状态。①mov sp,R15，将 sp 下拉到 fp 所指的位置；②pop R15，返回主函数的 fp 值；③ret，相当于 pop pc，也就是返回到主函数调用子函数的下一条指令。

经验谈：

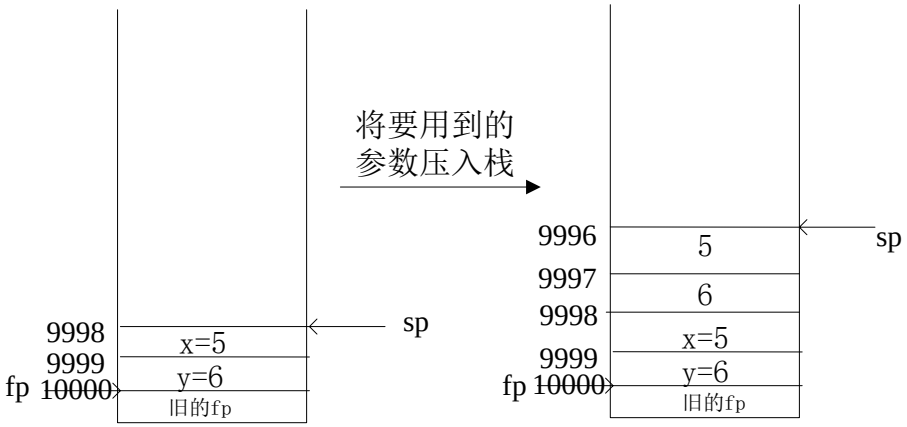
(1) 主函数将参数值压入栈后，这些参数的位置就会被子函数当作变量来使用，如例中子函数的 a 和 b，其地址分别是 2(R15)和 3(R15)。

(2) 主函数是将参数的**“值”**传递给子函数，这种方式叫做“call by value”，是一种较为通用的参数传递方式，比如 C 语言就是用这种方式。当然，这个值也可以用来传递参数的地址，只不过子函数要做相应的更改，就如同在 C 语言中传递一个指针，那么子函数中就要对指针做运算了，在此我们就不做额外的解释了。

(3) 函数的返回值普通用寄存器 R1 返回，假如有多个返回值的话可以用多个寄存器返回，但是要事先约定好。

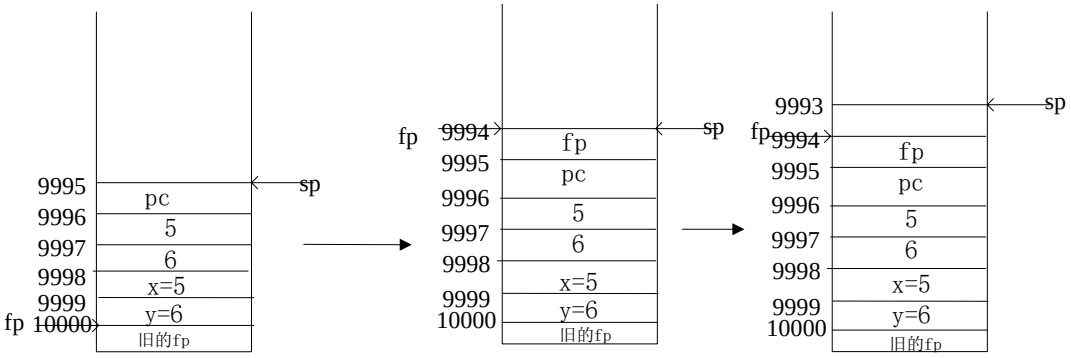
(4) 除了返回的寄存器 R1 之外，其他的寄存器应该在函数调用后保持与函数调用之前相同的值，所以在函数计算开始前，需要将函数中会被更改的寄存器值 push 到 stack 中保存，在 return 前再一一 pop 回来，例如函数 add 更改了寄存器 R2 和 R3，所以在更改之前先 push 进 stack，在 return 前再 pop 返回原来的 R2 与 R3 的值，请见汇编代码。

(5) 返回后，参数仍然留在 stack 中，主函数可以将参数 pop 出，但是需要消耗 pop 指令的代价，所以主函数常常就坐视不管，让其留在 stack 中。



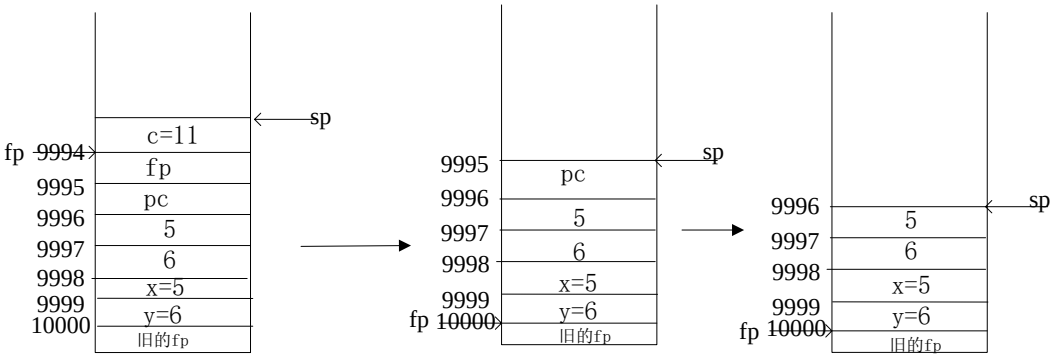
(a)函数调用前栈的状态　　(b)传递参数后栈的状态

图 3-5 执行 call 指令之前栈的状态



(a)执行call指令之后　　(b)将旧的fp入栈　　(c)最终栈的状态

图 3-6 执行 call 指令之后栈的状态



(a)函数结尾三条指令之前的状态　　(b)ret之前的状态　　(c)ret之后的状态

图 3-7 函数返回时栈的状态

3.6 程序例六：函数调用求三个数最小值

```
#<python 代码：三个数求最小值>
def get_min(x,y):
    if x<=y:
        return x
```

```
    else:
        return y
if __name__=="__main__":
    a = 7
    b = 18
    c = 9
    print(get_min(get_min(a,b),c))
```

```
#主函数有三个变量 a = 7，b = 18,c = 5
#  调用函数 min(a,b)2 次
#  找出 a，b，c 中的最小值

#<汇编代码：三个数求最小值>
mov R15,300        #R15 表示 fp，将基地址设置为 300
mov sp,R15       # sp = fp
sub sp,sp,3     # sp 从 300 往下开辟 3 个空间给局部变量 a,b,c    sp = sp-3

mov R2,7     # a = 7
mov R3,18    # b = 18
mov R4,9    # c = 9
store -1(R15),R4
store -2(R15),R3
store -3(R15),R2

push R3    #传参数 b
push R2    #传参数 a

call Lmin #调用函数 min(a,b),返回值存在 R1 中

push R4     #传参数 c
push R1     #传 a 和 b 中的最小值

call Lmin     #再次调用 min(R1,c),返回值为保存在 R1 中
goto Lprint     #

# min 函数有两个参数 a，b，将最小值放到 R1 中返回
Lmin:    # min(a,b)
push R15       #将旧的 fp 值存入栈内
mov R15,sp #  新的 fp 等于 sp
#sub sp,sp,0     #  因为此函数没有局部变量，所以可以去掉
push R2     #在函数中被更改，所以先存入栈内，在 return 之前会 pop 出来
push R3     #在函数中被更改，所以先存入栈内，在 return 之前会 pop 出来
push R4     #在函数中被更改，所以先存入栈内，在 return 之前会 pop 出来
load R2, 2(R15)     # R2 存放 x 的值
```

```
load R3, 3(R15)   # R3 存放 y 的值
sle R4, R2, R3    # R4 =(R2<=R3)
beqz R4, L100 # if (R4 == 0) goto L100
mov R1,R2     # R1=R2，结果存在 R1 中
goto Lreturn

L100:
mov R1,R3     #R1=R3，结果存在 R1 中

Lreturn:
pop R4     #返回旧的 R4
pop R3
pop R2
mov sp,R15     # sp = R15
pop R15       #重设 R15 的值，成为旧的 R15
ret       # pop pc

Lprint:
_pr R1     #打印最后的结果
```

在经过程序例五的介绍，相信大家对函数调用有了一定的了解，现在我们给出一个相对于程序例五比较复杂的函数调用示例。在本例中，是使用函数调用求的三个数的最小值，而求最小值的函数只是对两个数进行比较，所以需要调用两次求最小值的函数才可以求得三个数中的最小值。

首先需要设置主函数的 fp、sp 值且初始 sp 值等于 fp 值，在这里我们使用 mov 指令将 fp 的值赋给 sp。同时，还需要为主函数的局部变量开辟对应的空间，本例的主函数有三个局部变量，所以需要开辟三个空间，使用 sub 指令对 sp 进行减三操作，sp 指向开辟空间后的栈顶，使用三条 mov 指令对三个变量赋值，即我们希望在这三个数中获得最小值，并将三个数存储在所开辟的空间中。此时栈的状态如图 3-8(a)所示。

接下来第一次调用求最小值 min 函数时，需要将前两个数作为参数传给 min 函数的形参，使用两条 push 指令进行参数的传递。此处传的是 a，b 两个参数，所以将 a 和 b 分别入栈，此时的栈的状态如图 3-8(b)。

传完参数之后开始对 min 函数进行调用，执行指令 call Lmin。

在执行 call 指令之后，依然是会完成两步操作，一步是将返回地址值压入栈中，第二步是跳转到被调函数进行执行，此时栈的状态如图 3-9(a)所示；在被调函数开始依然需要进行三步操作：①将旧的 fp 存储②将 sp 的值赋给 fp，作为被

调函数的 fp③假设被调函数的局部变量需要 n 个空间，则 sp 上移 n 个位置。

先将旧的 fp 压入栈，此时栈的状态如图 3-9(b)；再将 sp 的值赋给 fp，即将 fp 上移到 sp 所指的位置，此时栈的状态如图 3-9(c)。在例子中的被调函数中没有局部变量，所以 sp 不需要移动，因此栈的状态保持不变。

同样，将 R2，R3，R4 压入栈中，以确保数据的干净与安全，此时栈的状态如图 3-10。参数 x 的地址是 2(R15)，参数 y 的地址是 3(R15)，比较后较小的值由 R1 返回。

接下来要返回时要执行三条指令，指令 mov sp, R15 把 fp 值赋给 sp，即把 sp 下移，如图 3-11(a)所示；指令 pop R15 返回主函数的 fp 值，如图 3-11(b)所示；指令 ret，相当于 pop pc，也就是返回到主函数调用子函数的下一条指令，如图 3-11(c)所示。

在此我们假设主函数不将原来的两个参数 a, b 弹出，这样并不会影响程序的正确性，接下来我们将两个新的参数 c 和 R1 依次压入栈中，如图 3-12 所示，然后再 call Lmin，栈帧的建立方式如前所述，在此就不再重复了。最后的最小值由 R1 返回。

经验谈：

此例显示我们依循函数连接的标准规则后，一个函数可以被多次调用而依旧能正确地执行，各位同学也可以试着看递归函数的栈帧如何能依次建立和返回。但是要注意，在程序执行前，栈中要保留足够的空间，使得每个函数调用时能够有足够的空间来建立它的栈帧，尤其对于递归函数而言，栈空间的大小更是重要。

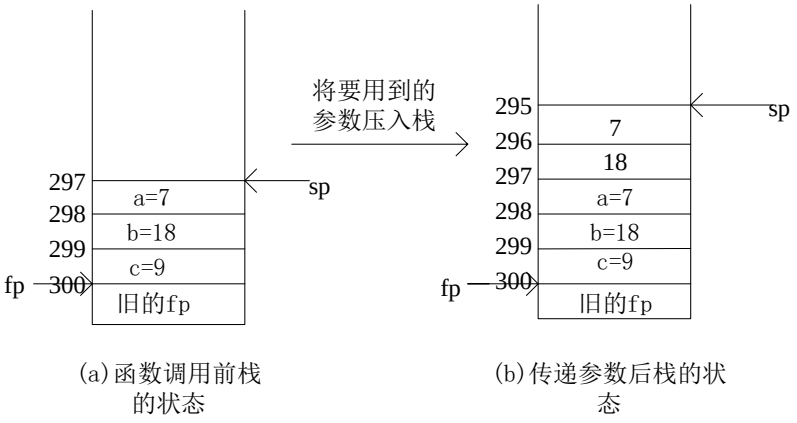
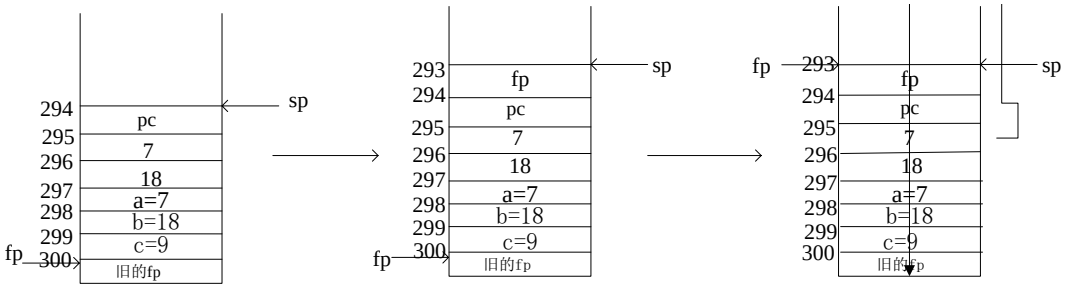


图 3-8 执行 call 指令之前栈的状态



(a)执行call指令之后　　(b)将主函数的fp压入栈　　(c)将fp上移到sp所指的位置

图 3-9 执行 call 指令之后栈的状态

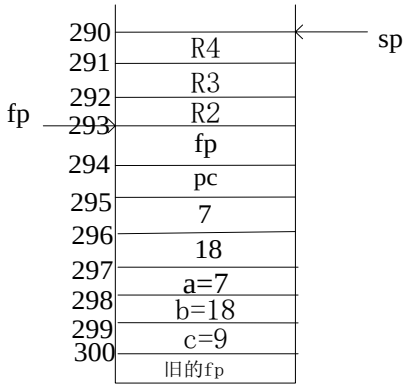
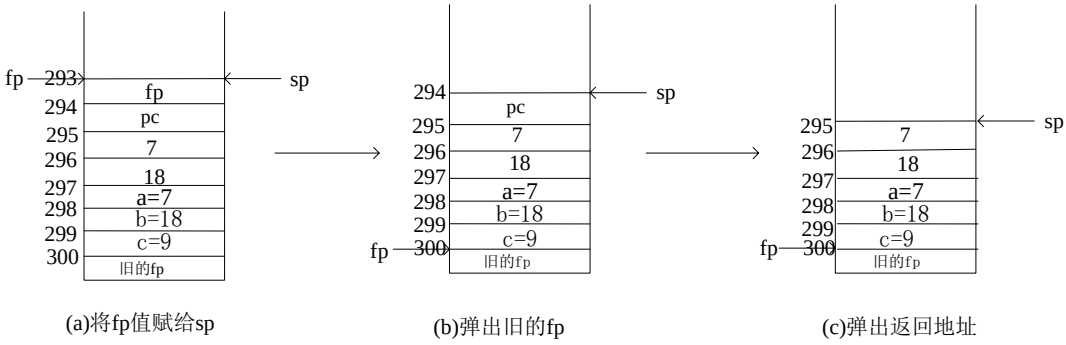


图3-10 push在子函数中更改的寄存器后栈的状态



(a)将fp值赋给sp　　(b)弹出旧的fp　　(c)弹出返回地址

图 3-11 函数返回时栈的状态

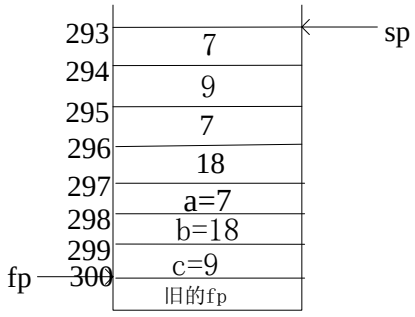


图3-12 第二次函数调用传递参数

根据返回的地址，跳转到第一次调用 min 函数的下一条指令处，是第二次调

用 min 函数的参数传递指令。第二次传参是将第三个数和第一次调用返回的较小值传递。传参之后的函数调用以及函数返回过程与第一次函数调用一样，大家可根据第一次函数调用时栈的状态来思考第二次函数调用的栈的状态。

同学们要参看《计算机科学导论——以 Python 为舟》一书的第 3.5 章节的基本概念，在 SEAL 中与书上的细节有以下几点差异：①在 SEAL 中，是先将参数压入栈中，再进行返回地址压入栈中，书中与此相反；②书中没有介绍将 fp 压入栈中，SEAL 中介绍了 fp 的压入，并且使用 R15 表示 fp；③书中讲的是基本概念，SEAL 是参照 x86 的通用 c 语言的模式，是真实的栈的管理方式，所以在进入函数之后，编译器会计算出函数的局部变量所需要的空间，一次性将 sp 上移足够的空间，为函数的局部变量保留足够的空间。

至此，在以上六个程序例中已经介绍了 22 条汇编指令，还有两条指令是 clz 指令和_pause 指令，对于_pause 指令，将会在第 4 节进行具体介绍，现在给出 clz 指令的使用，该指令有一种格式：clz R2,R1。执行该指令后将会把寄存器 R1 中存储的值从最高位到遇到第一个 1 之间的 0 统计，再将统计的值存入寄存器 R2。如果最初寄存器 R1 中所存储的值为 7（二进制：111），由于寄存器是 64 位，所以统计第一个 1 之前的 0（最高位为 0）应该有 61 个，将统计出来的 61 存入寄存器 R2。clz 指令在设计除法计算的时候使用到，大家可以自己设计一个计算除法的程序，用一用 clz 指令。

4. Debug 的使用介绍

SEAL 除了 normal 模式，还有 debug 模式，可以设置断点进行调试，调试分为两种调试方式：①从断点处开始逐行调试；②按照断点进行调试。

本节中使用示例 eg3_for_1 和 eg3_for_1_debug 进行介绍，由于两个示例的计算功能相同，区别是 eg3_for_1_debug.txt 文件比 eg3_for_1.txt 文件多了三条“_pause”指令，所以此处只展示出 eg3_for_1_debug.txt 中的程序：

```
#<python 代码：for 循环求 1+2+…+10>
s = 0
for i in range(1,11):
    s = s + i
print(s)
```

```
#需要加的数字放在 R0 上
#最终的结果放在 R1 上
#<汇编代码：for 循环求 1+2+…+10>
mov R0,1    #第一个数
_pause
mov R1,0    #存结果
_pause
L0:
sle R2,R0,10    #循环终止条件
_pause
beqz R2,L1      #if R2>=10，跳转到 L1
add R1,R1,R0
_pause
add R0,R0,1
goto L0    #跳到 L0 循环执行
L1:
_pr R1
```

两个示例均可在 normal 模式下运行，且得到的值是一样的。它们是对 1+2+…+10 进行计算。

①首先使用指令“mov R0,1”将第一个数赋值给寄存器 R0；使用指令“mov R1,0”对寄存器 R1 进行赋值，用来存结果，初始值为 0。

②在每次开始计算的时候，要先判断计算是否结束，所以使用指令“sle R2,R0,10”进行判断，如果 R0 中的数值小于 10，则将 R2 的值赋为 1，否则赋为 0。因为是从 1 加到 10，所以需要 R0 大于 10 之后，计算结束，故将用寄存器 R0

与 10 进行小于等于判断。

③使用指令“beqz R2,L1”对 R2 进行是否等于 0 判断，来确定是顺序执行还是跳转到 L1 标记的指令块处执行。如果 R2 的值为 1（加法计算未完成），则顺序执行下一条指令；否则（加法计算完成）跳转到 L1 标记的指令块，执行指令“_pr R1”将计算结果输出。在前 10 次的比较中，均是顺序执行下一条指令的。

④在第一次循环中，将是顺序执行指令“add R1,R1,R0”，将寄存器 R0 中的数与存储结果的寄存器 R1 中的数 0 进行求和，将求和的结果存回寄存器 R1 中。在计算一次加法之后，需要使用指令“add R0,R0,1”对计数的寄存器 R0 进行加 1 以便 R0 作为下一个数进行求和。

⑤在执行完一次加法所需要执行的一系列指令之后，要进行计算是否结束的判断，所以需要执行指令“goto L0”，跳转到 L0 标记的指令块处进行判断，即执行步骤②。

⑥之后的加法重复执行②③④⑤步，直至步骤③中的寄存器 R2 的值为 0，跳转到 L1 标记的指令块，输出计算结果，示例的计算结果应该为 55。

先对该示例按照第 2 节中对 eg1_if.txt 示例的运行过程对 eg3_for_1.txt 文件和 eg3_for_1_debug.txt 文件分别在 normal 模式下进行运行，可以得到两个文件的执行结果是一致的，且均是 55，执行结果如图 4-1：

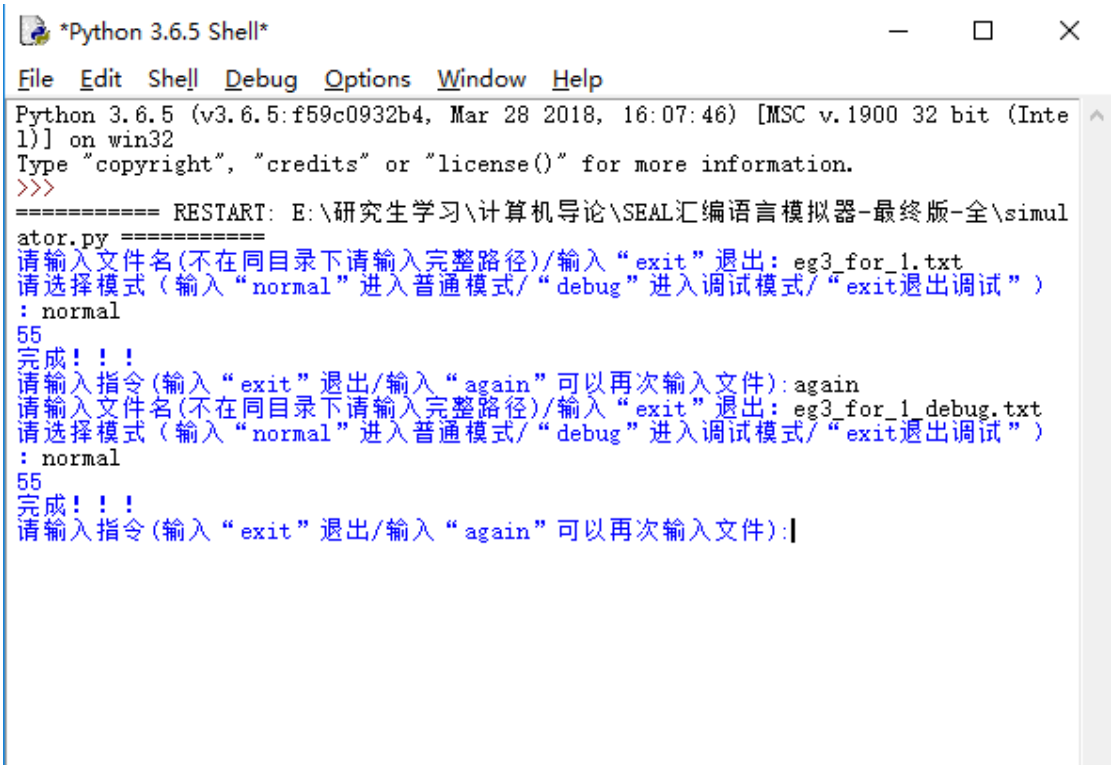


图 4-1 for 循环 normal 模式下计算结果

- **调试模式的选择与退出**

➢ 1：输入数字 1 进入逐条调试模式（即从断点处开始逐条调试）。

➢ 回车：回车指令有两种含义，如果在进入 debug 模式之后，首先输入数字 1，紧接着按“回车”，此时“回车”表示逐行调试。当进入 debug 模式后，此时直接按“回车”，意味着在按回车之前没有输入数字 1，模拟器将会启动断点调试功能，并且这取决于同学的汇编程序是否设置多个断点，如果在汇编程序中仅设置了一个断点，那么程序会在一次回车后执行完全部程序，并输出结果。如果在汇编程序中设置了多个断点，按“回车”后会跳转到下一个断点继续执行。

➢ exit： 同学输入 exit 退出 debug 模式或者返回 debug 模式（根据目前所处情况会有其对应的退出）。

- **执行 debug 指令**

使用 eg3_for_1_debug.txt 示例进行 debug 模式的介绍，当同学们按照第 2 节输入正确的汇编程序文档名之后，紧接着输入 debug 进入调试模式，得到的结果应该与图 4-2 相同，同时 debug 执行之后会弹出如图 4-3 所示的调试结果显示窗口，该窗口的顶部显示的是当前执行的指令的 pc 值以及指令内容，在图

4-3 中显示“pc: 1 mov R0,1”， 查看 eg3_for_1_debug.txt 源码同学们能够知道，第一个“_pause”指令被设置在“mov R0,1”指令之后。

图 4-2 对 eg3_for_1_debug.txt 执行 debug 命令

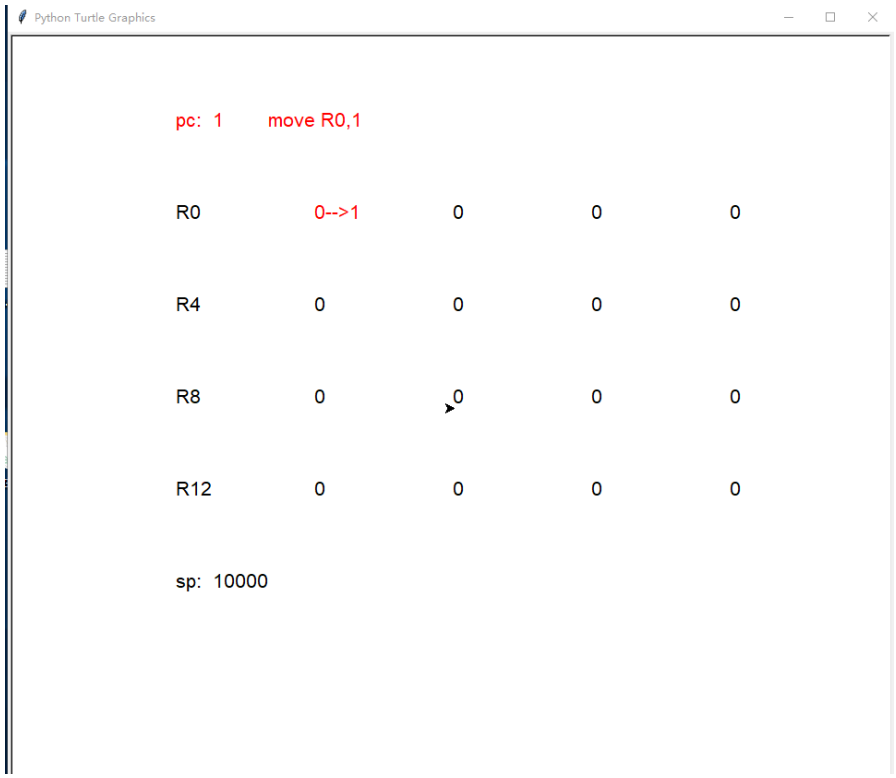


图 4-3 调试结果显示窗口

从图 4-3 中我们看到，中间区域显示了 17 个寄存器的值，第一行即 R0 行分别表示寄存器 R0、R1、R2、R3 的值，第 2 行即 R4 行分别表示寄存器 R4、R5、R6、R7 的值，第三行即 R8 行，分别表示 R8、R9、R10、R11 的值，第四行即 R12 行分别表示 R12、R13、R14、R15 的值，最后一行即 sp 行，表示 sp 指针寄存器的值，主要用于函数调用示例中。总共 17 个寄存器，在调试汇编代码的过

程中，同学们能够动态的看到这些寄存器内容的变化。从图 4-3 中我们看到 R0 寄存器“0->1”，这表示指令“mov R0,1”执行完成之后，R0 寄存器的内容将会变成 1。

从图 4-2 中，同学们看到“选择继续的方式（输入“1”进入逐行调试模式/按回车继续断点调试/“exit”退出调试）：”，即 SEAL 汇编语言模拟器给出两种调试模式，输入数字 1 进入逐行调试模式和按“回车”进入断点调试模式，下面我们分别讲解这两种调试模式。

执行完上述步骤后，紧接着在 IDLE 中输入数字 1（即开始逐行调试模式），将出现图 4-4 所示内容。此时寄存器的值如图 4-5 所示，我们看到 R1 寄存的值变为 1，这是由于第一条指令已经执行完毕。

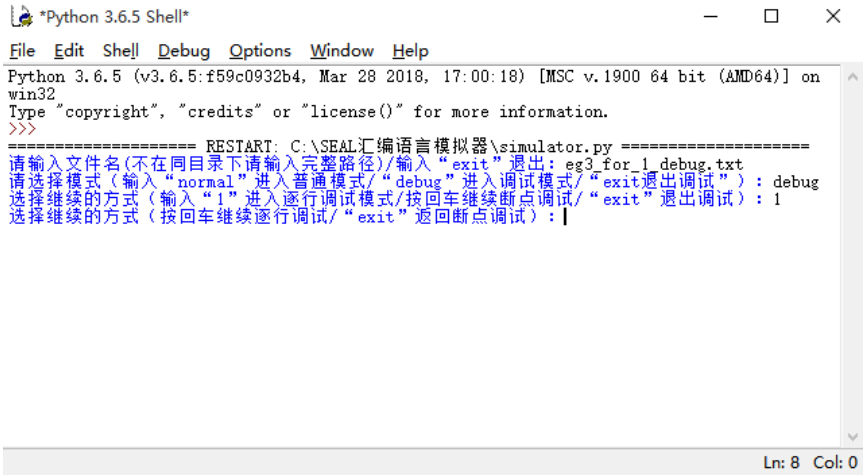


图 4-4 逐行调试

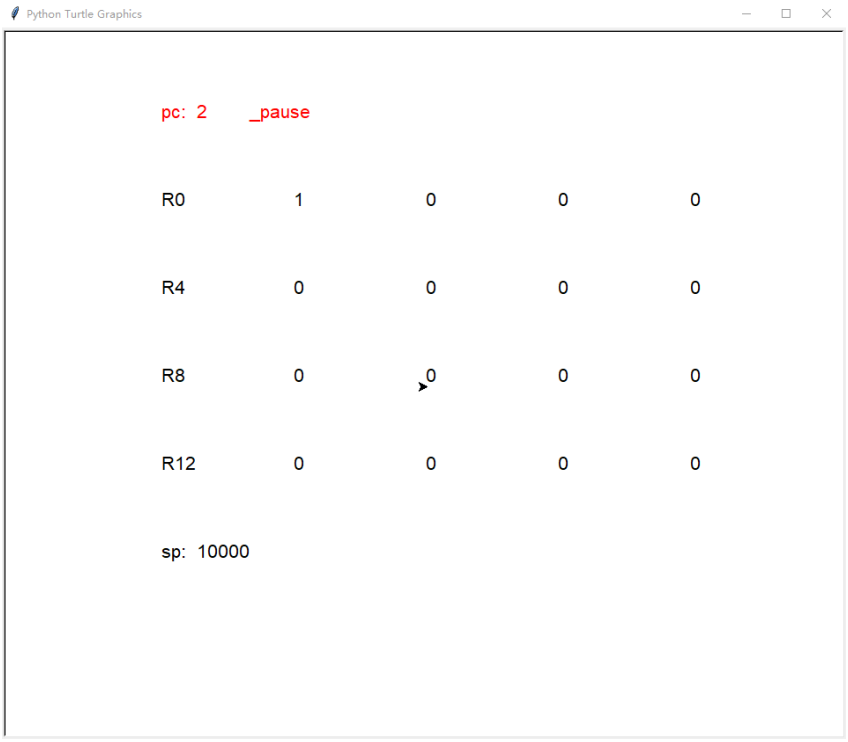


图 4-5 逐行调试

接着同学们能够看到“选择继续的方式（按回车继续逐行调试/“exit”返回断点调试）:”，此时，如果按“回车”将继续逐行调试，每次按“回车”都会询问同学是否要继续选择逐行调试，如果不想继续逐行调试，输入“exit”会出现图 4-6 所示内容，此时会显示“选择继续的方式（输入“1”进入逐行调试模式/按回车继续断点调试/“exit”退出调试）:”，同学们能够重新选择调试模式。或者退出调试。

图 4-6 退出逐行调试模式

紧接着，我们输入“回车”，进行断点间调试过程的讲解。输入“回车”后，调试界面如图 4-7 所示，此时看到图 4-7 所示窗口顶部“pc: 6 sle R2,R0,10”,且

R2 的值从 0 变为了 1：“0-->1”细心的同学查看 eg3_for_1_debug.txt 源代码后发现，该行之后插入了“_pause”指令。之后同学可按照给出的提示继续执行。

图 4-7 断点调试

当调试完成之后，如果汇编代码中有“_pr”指令，将会输出结果，否则会输出“完成!!!”字样，并且提示“若调试界面没有关闭，请点击界面上任意空白位置”，同学点击调试窗口，该窗口将会消失。如图 4-8 所示，调试结束，并且输出“55”和“完成!!!”字样，并且提示同学们点击“界面上任意空白处”关闭调试窗口。

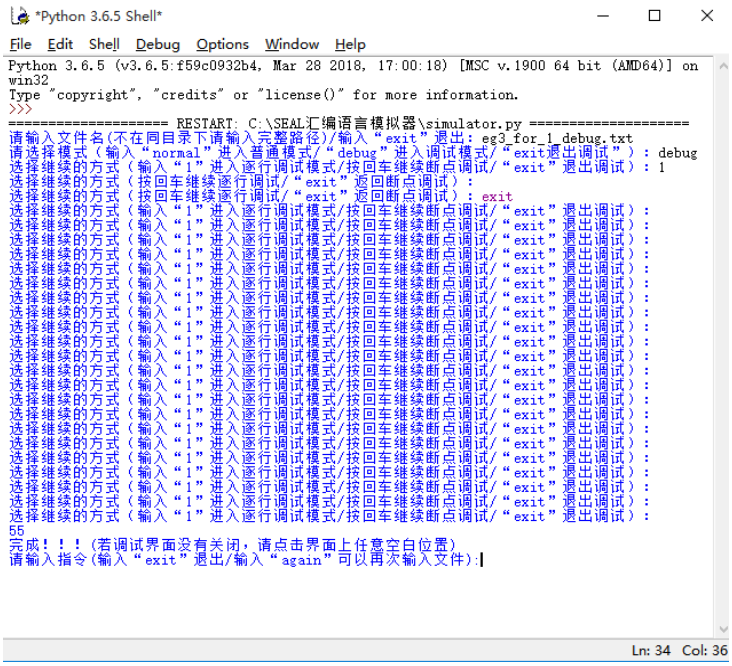


图 4-8 调试结束

调试结束之后，如图 4-8，会提示“请输入指令(输入“exit”退出/输入“again”可以再次输入文件):”，此时同学输入“exit”将退出模拟器，如果输入“again”将继续提示“请输入文件名(不在同目录下请输入完整路径)/输入“exit”退出：”，循环前面的所有操作，如图 4-9 所示。

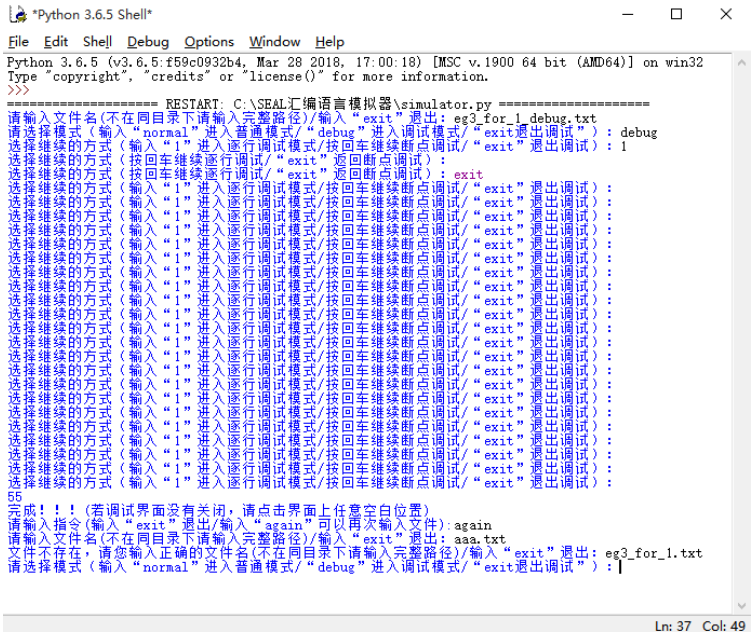


图 4-9 继续再一次运行或调试

Debug 模式下的调试方法已经详细介绍，同学可根据以上介绍过程进行汇编程序的调试。

5. 习题

习题1: 假设寄存器中的 R1 的值为 5，执行完下面给出的指令后，寄存器 R2 中存储的值是多少，存储的内存地址是多少。

```
mov R2,R1
store 04(R1),R2
```

习题2: 假设寄存器 R1、R2 中的值分别为 10 和 15，执行完下面这段汇编指令后，R2 中的值是多少，R1 的值存储的内存地址是多少？

```
L0:
sle R3,R2,R1
beqz R3,L1
sub R2,R2,2
goto L0
L1:
store 05(R2),R1
```

习题3: 假设寄存器 R1 中的值为 5，执行下面汇编指令后，寄存器 R1 中的值是多少？

```
shiftl R1,R1,3
add R1,R1,3
shiftr R1,R1,4
```

习题4: 输入一个数放在 R0 上，计算出这个数的二进制有多少个 1，计算结果放在 R1 上。例如：R0 = 10，二进制是 1010，它有 2 个 1，所以 R1 = 2。

习题5: 使用汇编指令，将列表 L[0,1,2,3,4,5,6,7,8,9,10] 中为偶数的数值加起来。假设这个列表的存储使用_data，放在首地址 0 处，结果放在 R1 上。

习题6: 输入一个任意整数，例如 725，放在 R0 中，结果为最高位的整数值并且放在 R1 中，例如 725 的最高位是 7。

习题7: 输入一个字符串，使用_data 输入，首地址设为 0，第 0 个位置放字符串的长度，从 1 位置开始放字符串，其中字符为 0 或 1 的 ASCII 码值，即 48 或 49，计算出这个字符串的十进制的整数值是多少，放在 R1 上。例如：_data 0,[4,49,49,49,48]，结果为 14 放在 R1 上。

请试验如下数据：_data 0,[8,48,49,48,49,48,49,49,48]

习题8: 输入一个字符串，使用_data 输入，首地址设为 0，第 0 个位置放字符串的长度，从 1 位置开始放字符串，其中字符为 0 或 1 的 ASCII 码值，即 48 或 49，计算出该字符串中最长连续段 1 是多少个，并将结果放在 R1 上。例如：

_data 0,[16,49,48,49,49,49,48,48,49,49,49,49,49,48,49,49,48]，结果为 5 放在 R1 上。

习题9: 使用_data 输入一个序列，将首地址设为 0，对该序列进行一趟遍历之后，可以同时找到该序列的最大值与最小值，最后将该序列的最大值放在 R0 中，最小值放在 R1 中。

习题10: 使用_data 输入一个排序好的序列，将首地址设为 0，同时给定一个元素，放在 R0 上，将该元素插入到正确的位置上，将新的序列存储在与原来的序列首地址相同的位置处，并且使用_pr (0)%n 将新的序列打印出来。

例如序列为_data 0,[2,5,6,9,10,54]，插入新的数字 8 之后新的序列为：[2,5,6,8,9,10,54]，此时打印出的结果为：0：2，1：5，2：6，3：8，4：9，5：10，6:54 。

习题11: 输入一个无符号的正整数，最大为 65535（二进制数为 16 位），放在 R0 上，对 16 取余，得到的余数放在 R1 上。注意：不能使用 mul 指令和 div 指令。

例如：一个整数为 118，放在 R0 上，其二进制为 0111 0110，那么对 16 进行取余，只需要将 118 的二进制的低四位（0110）取出作为余数，所以 118 对 16 取余的结果的二进制为 0110，即十进制数 6，所以 R1=6。

习题12: 输入一个无符号的正整数，最大为 65535（二进制数为 16 位），放在 R0 上，对 15 取余，得到的余数放在 R1 上。注意：不能使用 mul 指令和 div 指令。

假设一个 16 位的二进制数，将其四位四位分割，第一个四位取为 a，第二个四位取为 b，第三个四位取为 c，第四个四位取为 c，则十进制整数表示为$a*16^3+b*16^2+c*16+d$，那么这个整数对 15 取余可以表示为：

$(a*16^3+b*16^2+c*16+d)\%15$

$=(a\%15*(16\%15)^3)+(b\%15*(16\%15)^2)+(c\%15*(16\%15)^1)+d\%15$

$=a\%15+b\%15+c\%15+d\%15$ ，

因此该整数对 15 取余可以转化为(a+b+c+d)%15，则当(a+b+c+d)小于 15 时，余数便为(a+b+c+d)，否则(a+b+c+d)减去 15，直至小于 15 得到余数。

例如：一个整数 38351，它的二进制表示为 1001 0101 1100 1111，将其四位四

位分割，a=9，b=5，c=12，d=15，可以表示为$9*16^3+5*16^2+12*16^1+15$，故对 15 取余，可以转化为(9+5+12+15)%15，(9+5+12+15)大于 15，需要减两次 15，才会比 15 小，得到此时的余数为 11。

习题13: 使用汇编指令，写出计算机做除法的运算过程。其中用将被除数放在 R0 上，除数放在 R1 上，商放在 R2 上，余数放在 R3 上。（提示：在做除法的时候，需要将余数进行左移位与被除数的最高位对齐，这里可以先使用 clz 指令将被除数和除数从最高位到第一个 1 之间的 0 统计出来，以确定除数需要移几位可以与被除数最高位对齐。）

习题14: 讲一讲 push 和 pop 分别是如何做的，通过哪几条汇编指令可以完成操作？

习题15: 讲一讲函数调用的时候，是如何建立栈帧的。

习题16: 假设 R15 表示 fp，fp 的值为 300，通过执行下面这段汇编指令后，fp 和 sp 的值分别是多少，在函数调用过程中局部变量 R0 和 R2 的值所存储的内存地址是多少，传递参数时两个参数存储的内存地址分别是多少？

```
mov R15,300
mov sp,R15
sub sp,sp,2
mov R0,3
mov R2,5
store -1(R15),R2
store -2(R15),R0

push R2
push R0
call L0
goto L1
L0:
push R15
mov R15,sp
load R0,2(R15)
load R2,3(R15)
add R1,R0,R2
mov sp,R15
pop R15
ret
L1:
```

习题17: 使用汇编指令，将习题 13 写好的除法封装在一个函数里，并且调用除法

函数计算 81÷7 的值。将 81 作为被除数放在 R2 上，除数放在 R3 上，所得的商放在 R1 上，余数放在 R0 上。

习题18: 使用_data 输入一个序列，首地址为 0，存储的第一个数据为序列的长度，后续依次是该序列的数值，使用选择排序算法对该序列排序，并将排序好的序列存储在原始序列所存储的内存地址处，同习题 13，使用_pr (0)%n 将排好序的序列打印出来。

例如：_data 0,[10,8,10,7,16,10,3,9,7,20,2]中，[]中的第一个数 10 为序列的长度，存储在内存地址 0 处，从内存地址为 1 开始的十个数字为序列的数字。排好序后的序列[2,3,7,7,8,9,10,10,16,20]依然是存储在内存地址为 1 开始的地方。使用_pr (R0)%10 打印，打印的结果为：1：2，2：3，3：7，4：7，5：8，6：9，7：10，8：10，9：16，10：20，

习题19: 使用_data 输入一个序列，首地址为 0，存储的第一个数据为序列的长度，后续依次是该序列的数值，使用插入排序算法对该序列排序，并将排序好的序列存储在原始序列所存储的内存地址处，同习题 18，使用_pr (0)%n 将排好序的序列打印出来。

例如：_data 0,[10,8,10,7,16,10,3,9,7,20,2]中，[]中的第一个数 10 为序列的长度，存储在内存地址 0 处，从内存地址为 1 开始的十个数字为序列的数字。排好序后的序列[2,3,7,7,8,9,10,10,16,20]依然是存储在内存地址为 1 开始的地方。使用_pr (R0)%10 打印，打印的结果为：1：2，2：3，3：7，4：7，5：8，6：9，7：10，8：10，9：16，10：20，