

**SEAL (Simple Educational Assembly Language) 使用手册**

***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 1](#_Toc530341143)

[2. 模拟器使用 3](#_Toc530341144)

[2.1 SEAL内置指令 3](#_Toc530341145)

[2.2 模拟器使用步骤 3](#_Toc530341146)

[3. SEAL汇编语言指令 8](#_Toc530341147)

[3.1 load指令 8](#_Toc530341148)

[3.2 mov指令 9](#_Toc530341149)

[3.3 add指令 9](#_Toc530341150)

[3.4 sub指令 10](#_Toc530341151)

[3.5 shiftl指令 11](#_Toc530341152)

[3.6 shiftr指令 12](#_Toc530341153)

[3.7 store指令 13](#_Toc530341154)

[3.8 slt指令 14](#_Toc530341155)

[3.9 sle指令 15](#_Toc530341156)

[3.10 beqz指令 16](#_Toc530341157)

[3.11 goto指令 16](#_Toc530341158)

[3.12 and指令 17](#_Toc530341159)

[3.13 or指令 18](#_Toc530341160)

[3.14 xor指令 19](#_Toc530341161)

[3.15 clz指令 19](#_Toc530341162)

[3.16 call指令 20](#_Toc530341163)

[3.17 ret指令 20](#_Toc530341164)

[3.18 push指令 21](#_Toc530341165)

[3.19 pop指令 21](#_Toc530341166)

[3.20 \_data指令 21](#_Toc530341167)

[3.21 \_pr指令 22](#_Toc530341168)

[3.22 \_pause指令 26](#_Toc530341169)

[3.23 div指令 26](#_Toc530341170)

[3.24 mul指令 27](#_Toc530341171)

[4. 函数调用的介绍 28](#_Toc530341172)

[5. Debug的使用介绍 28](#_Toc530341173)

[5.1 调试模式的选择与退出 28](#_Toc530341174)

[5.2 执行debug指令 28](#_Toc530341175)

[6. 程序示例 34](#_Toc530341176)

[附录A. 环境准备 37](#_Toc530341177)

## 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全称是Simple Educational Assembly Language，它是精简版汇编语言模拟器。SEAL的主要功能是模拟程序在计算机里的执行过程，并忽略与真实计算机相关的复杂细节。一个计算机系统至少包含了CPU和主存（Main Memory，或称为内存）。CPU是做计算的。主存是存储程序和变量的。SEAL将精简地模拟这些功能。SEAL首先将一行行的程序读入主存，接着控制CPU的运算并指示CPU去读写主存中的变量，在SEAL汇编语言模拟器中，我们还解释了函数调用的执行过程，如返回地址以及栈的管理等内容。SEAL全部用Python语言写成，充分利用了Python中的数据类型实现SEAL的功能，采用这种设计的主要原因是想让读者了解计算机程序的执行过程，同时忽略计算机指令集等设计与实现的复杂性。

SEAL实现了24条“高级”汇编语言指令，这种高级一方面体现在：每条汇编指令与真实机器上实现的汇编语言指令有所区别，真实汇编语言被翻译成机器语言后会直接操作真实机器上的寄存器、内存单元等，而我们的“高级”汇编语言会操作通过Python来模拟的物理硬件，如内存、寄存器等。另一方面体现在：我们的设计使得SEAL突破了真实汇编语言存在的种种限制，比如操作数等必须要放在一个字中，再比如真实的汇编语言专用于某种计算机系统结构，可移植性差等的限制，SEAL的设计突破了这些限制。这使得SEAL能够精简的模拟计算机程序的执行过程，又避免读者陷入困境。

表1-1给出了SEAL设计中最为重要的几种结构。我们知道内存单元具有随机访问特性，在SEAL模拟器中，我们用Python内置数据类型List模拟了真实的内存，使用List的索引（或称为下标）实现地址的随机访问，SEAL提供的内存大小阈值为10000个单元，由于我们开放源代码，这使得读者能够根据需求任意增加或者减少SEAL所能模拟的内存大小，读者只需要修改Memory\_size的大小即可实现。SEAL还模拟了18个寄存器，其中16个64位普通寄存器，即结构Reg，其命名为“R0,R1,R2…R15” ，可以存储数据的大小在范围内且支持十进制和十六进制（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）。一个堆栈指针寄存器，命名为“sp”，一个指令寄存器，命名为“pc”。我们在设计SEAL将底层的操作与功能作更为高级的抽象，忽略底层细节，充分地将计算机指令的执行过程展现出来。这样高级的抽象使得同学能快速掌握计算机在程序执行过程的精髓部分。这和我的教材《计算机科学导论-以Python为舟》相得益彰。

表1-1 SEAL中重要结构

|  |  |  |
| --- | --- | --- |
| 结构 | 含义 | Python类型 |
| Reg | 寄存器 | List |
| Memory | 内存 | List |
| pc | 指令寄存器 | int |
| sp | 堆栈指针寄存器 | int |

SEAL使用手册后续部分的组织结构如下：

第2节 模拟器使用。本节我们介绍了SEAL的内置指令以及SEAL的使用步骤，读者应该认真阅读本节内容，这对于你理解计算机程序的执行过程至关重要。

第3节 SEAL汇编语言指令。本节给出了SEAL支持的“高级”汇编语言指令以及指令的功能与格式。

第4节 对函数调用的介绍以及举例说明

第5节 对SEAL的debug模式具体介绍

第6节 程序示例。本节给出9个程序示例，并给出了设计这些示例的目的。

附录A 简要的介绍了SEAL的运行环境，即安装Python过程。

注：本模拟器是使用python3开发的，请安装python3.0以上的版本，以便正常运行，详细的安装步骤见附录A

## 模拟器使用

### SEAL内置指令

#### 2.1.1 normal指令

功能：同学输入normal可以进入正常模式运行汇编程序，执行完成之后，如果同学的汇编程序中有“\_pr”指令，同学将会看到汇编语言程序的执行结果，并看到“完成！！！”字样，如果没有“\_pr”指令，将只打印“完成！！！”字样，提示同学汇编程序执行结束。

#### 2.1.2 debug指令

功能：同学输入debug可以进入调试模式，此时需要选择设置断点后逐条进行调试还是按照断点进行调试（即设置多个断点）。

（针对debug指令此处不做详细介绍，详细介绍可见第5章。）

#### 2.1.3 exit指令

功能：退出模拟器。在SEAL中，exit指令有多种功能，在这里特指退出模拟器。

#### 2.1.4 again指令

功能：重新执行前述过程。当一次“normal”或者“debug”执行完成之后，此时同学可以选择exit指令退出模拟器，也可以选择again继续运行或调试其它汇编程序。

### 模拟器使用步骤

#### 2.2.1运行模拟器

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：以eg开头、以.txt结尾的9个示例文件（eg3\_for\_1\_debug.txt功能同eg3\_for\_1.txt，此例主要用于讲解debug过程），SEAL教学说明文档(1.1版本).docx，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。

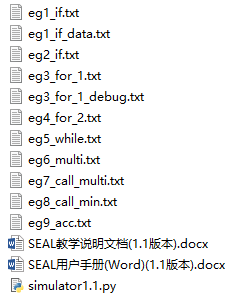


图2-1 SEAL汇编语言模拟器文件夹内容

本节的目的是启动simulator.py，它就是该手册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。

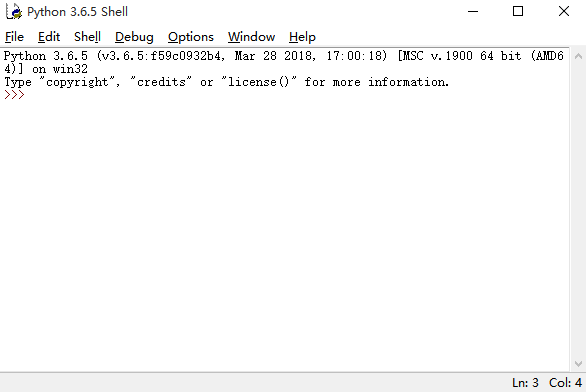


图2-2 IDLE启动界面

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3-3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，请同学接着阅读手册后续部分。



图2-4 SEAL成功启动

#### 2.2.2 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”）：”。为了后续方便解释模拟器的使用过程，这里输入eg3\_for\_1.txt。文件输入成功后，请同学继续阅读手册后续部分。

#### 2.2.3 执行SEAL指令

同学按照2.2.2节成功输入汇编程序文档名后，模拟器会给出提示“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”）：”，此时同学按照提示可以输入“normal”、“debug”或者“exit”，在SEAL内置指令一节中已经知道当输入“exit”时，退出模拟器。“normal”和“debug”指令不存在执行上的先后关系，二者选其一执行即可，不能同时执行。本节将重点说明输入“normal”指令后的情形，对于输入“debug”指令后的情形详见第5章。在解压后的“SEAL汇编语言模拟器”文件夹中，我们看到有eg3\_for\_1.txt文件，本节将使用这个文件来讲解“normal”指令的执行过程。

* **执行normal指令**

模拟器在运行汇编程序时，分为两个模式：normal模式和debug模式。输入normal进入正常模式运行汇编程序，并且打印最终结果（如果在汇编程序中有打印指令便打印最终结果以及“完成！！！”字样；否则只打印“完成！！！”字样）。图2-5给出了eg3\_for\_1.txt汇编文件的执行结果，看到输出有“55”和“完成！！！”。

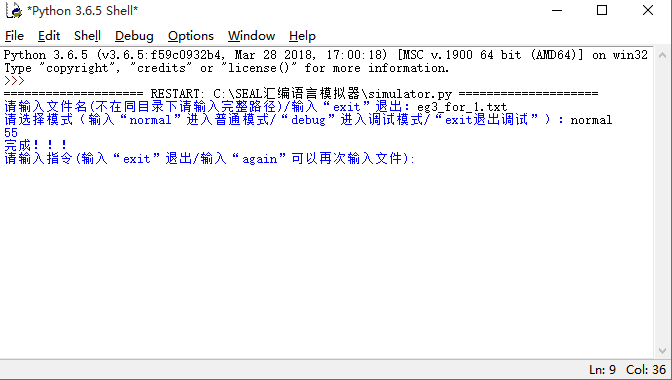


图2-5 对eg3\_for\_1.txt执行normal指令

在使用SEAL过程中，同学对汇编语言中出现的指令含义不甚了解，接下来我们将详细讲解每条指令的功能。

## SEAL汇编语言指令

### load指令

**功能：读取数值操作，即CPU将变量a读取到寄存器R中。**

格式1：load R1,(address)

load指令有两个操作数，前一个是寄存器，后一个为内存地址，其中address是内存地址，(address)表示这个地址存储的值，可以使用十进制表示，也可以使用十六进制表示（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），表示将主存地址为address处的变量值读取到寄存器R1中,即R1←(address)。

【**示例1.1**】load R1,(1000)

该指令表示，将主存地址1000处的值读取到寄存器R1中，即R1←(1000)。

【**示例1.2**】load R1,(3E8H)

该指令与示例1中的指令的功能一致，区别是将主存地址1000使用十六进制3E8H表示，其中E和H也可以写为e和h，即3e8h。均是表示将主存地址为1000（十六进制为3e8h）处的数据读取到寄存器R1中。

格式2：load R1,offset(R2)

load指令有两个操作数，前一个是寄存器，后一个为寄存器加偏移量，表示将R2中的值加上偏移量offset所表示的地址处所存储的值读取到寄存器R1中，即R1←offset(R2)。其中offset可以是十进制也可以是十六进制（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）。

【**示例2.1**】load R1,11(R2)

该指令表示，将R2中的值加11，并将结果作为一个地址，去找该地址处所存储的值，再将该值放入寄存器R1中。如果寄存器R2中最初存储的值是100，则11(R2)表示111，再将地址为111处所存储的数读取到寄存器R1中，所以R1←11(R2)表达的效果等同于R1←(111)。

【**示例2.2**】load R1,bh(R2)

该指令中偏移量使用的是十六进制，bh的十进制为11。该指令表示将R2中的值加11（十六进制为bh），并将结果作为一个地址，去找该地址处所存储的值，再将该值放入寄存器R1中。如果寄存器R2中最初存储的值是100，则bh(R2)表示111，再将地址为111处所存储的数读取到寄存器R1中，即R1←bh(R2)。

### mov指令

**功能：赋值操作，即给寄存器R赋一个值。**

格式1：mov R1,constant

mov指令有两个操作数，前一个是寄存器，后一个是十进制或十六进制的常数，表示将一个常数值赋给寄存器R1,即R1←constant。

【**示例**】mov R1,10

该指令表示，将一个常数值10赋给寄存器R1,即R1←10。

【**示例1.2**】mov R1,Ah

该指令表示，将一个十六进制数Ah（即十进制的数字10）赋给寄存器R1,即表达的效果等同于R1←10。

格式2：mov R2,R1

mov指令有两个操作数，两个都是寄存器，表示将后一个寄存器R1中的值赋给寄存器R2中,即R2←R1。

【**示例**】mov R2,R1

如果寄存器R1中最初存储的值为5，则执行该条指令后，寄存器R2中的值也为5，即R2←R1。

### add指令

**功能：加法操作，即对数值进行加法操作。**

格式1：add R2,R1,constant

add指令有三个操作数，其中前两个是寄存器，最后一个是常数，第一个寄存器是用来存储运算结果，第二个寄存器存储的是进行运算的值。即R2 ← R1+constant。这里的常数可以是十进制也可以是十六进制。

【**示例1.1**】add R1,R1,11

该指令表示将寄存器R1中的数值加11，并将结果存回寄存器R1。如果寄存器R1中的值最初是6，执行该指令后，寄存器R1中的值就为17，即R1← R1+11。

【**示例1.2**】add R1,R1,BH

该指令表示将寄存器R1中的数值加上十六进制数BH（即十进制的11），并将结果存回寄存器R1。如果寄存器R1中的值最初是6，执行该指令后，寄存器R1中的值就为17，即R1← R1+11。

格式2：add R1,R1,R2

add指令有三个寄存器操作数，表示将后两个寄存器R1、R2中的值相加，结果赋给寄存器R1，也就是R1← R1+R2。

【**示例**】add R1,R1,R2

如果最初寄存器R1中存储的值为6，寄存器R2中存储的值为1，则执行该指令后寄存器R1中的值就为7，即为R1← R1+R2。

### sub指令

**功能：减法指令，即对数值进行减法操作。**

格式1：sub R2,R1,constant

sub指令有三个操作数，其中前两个是寄存器，最后一个是常数，第一个寄存器是用来存储运算结果，第二个寄存器存储的是进行运算的值。即R2←R1-constant。这里的常数可以是十进制也可以是十六进制。

【**示例1.1**】sub R2,R1,10

该指令表示将寄存器R1中的数值减5，并将得到的结果存入寄存器R2。如果寄存器R1中的值最初是11，执行该指令后，寄存器R2中的值就为5，即R2← R1-10。

【**示例1.2**】sub R2,R1,Ah

该指令表示将寄存器R1中的数值减去十六进制Ah（即十进制数10），并将得到的结果存入寄存器R2。如果寄存器R1中的值最初是11，执行该指令后，寄存器R2中的值就为5，即R2← R1-10。

格式2：sub R3,R2,R1

sub指令有三个寄存器操作数，表示将后两个寄存器R1、R2中的值相减，结果赋给寄存器R3，也就是R3← R2-R1。

【**示例**】sub R3,R2,R1

如果最初寄存器R2中存储的值为10，寄存器R1中存储的值为5，则执行该指令后寄存器R3中的值为5，即R3←R2-R1。

### shiftl指令

**功能：左移位操作，即将二进制数向左移位，并将移出的位填0。**

格式1：shiftl R3,R1,constant

shiftl指令有三个操作数，其中前两个是寄存器，最后一个是常数（可以用十进制表示，也可以用十六进制表示）， 表示寄存器R1中的数向左移constant位，并将移出的constant位填0，将最终值存入R3。

【**示例1.1**】shiftl R3,R1,5

该指令表示将寄存器R1中的值左移5位，并将移出的5位填0，再将最终值存入寄存器R3中。如果R1中最初的值为3（二进制表示：11），执行该指令后寄存器R3中的值为96（二进制表示：1100000）。

【**示例1.2**】shiftl R3,R1,5h

该指令表示将寄存器R1中的值左移5h（即十进制5）位，并将移出的5位填0，再将最终值存入寄存器R3中。如果R1中最初的值为3（二进制表示：11），执行该指令后寄存器R3中的值为96（二进制表示：1100000）。

格式2：shiftl R3,R1,R2

shiftl指令有三个操作数，均为寄存器操作数，表示将寄存器R1中的二进制值左移(R2)位数（“(R2)”表示R2中存储的数值），将移出的(R2)位填0，存入寄存器R3。

【**示例**】shiftl R1,R1,R2

如果最初寄存器R1中存的值是5（二进制表示：101），寄存器R2中存的值是3，则执行该条指令后，寄存器R1中的值为40（二进制表示：101000）。

### shiftr指令

**功能：右移位操作，即将二进制数向右移位，并将移出的高位填0。**

格式1：shiftr R3,R1,constant

shiftr指令有三个操作数，其中前两个是寄存器，最后一个是常数（可以用十进制表示，也可以用十六进制表示），表示寄存器R1中的数向右移constant位，并将移出的constant位填0，将最终值存入R3。

【**示例1.1**】shiftr R3,R1,5

该指令表示将寄存器R1中的值右移5位，并将移出的5位填0，再将最终值存入寄存器R3中。如果R1中最初的值为53（二进制表示：110101），执行该指令后寄存器R3中的值为1（二进制表示：000001）。

【**示例1.2**】shiftr R3,R1,5h

该指令表示将寄存器R1中的值右移5h（即十进制5）位，并将移出的5位填0，再将最终值存入寄存器R3中。如果R1中最初的值为53（二进制表示：110101），执行该指令后寄存器R3中的值为1（二进制表示：000001）。

格式2：shiftr R3,R1,R2

shiftr指令有三个操作数，均为寄存器操作数，表示将寄存器R1记录的值的二进制右移寄存器R2记录的位数，将移出的位填0，存入寄存器R3。

【**示例**】shiftr R1,R1,R2

如果最初寄存器R1中存的值是53（二进制表示：110101），寄存器R2中存的值是3，则执行该条指令后，寄存器R1中的值为6（二进制表示：000110）。

### store指令

**功能：存回操作，表示CPU将寄存器R中的值存回到主存中。**

格式1：store (address),R1

address是内存地址，(address)表示要存回的地址的值，其中地址可以使用十进制表示，也可以使用十六进制表示（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），R1是寄存器。也就是(address)←R1。

【**示例1.1**】store (500),R1

该指令表示将寄存器R1中的值存回主存地址500处，即(address)←R1。

【**示例1.2**】store (1f4h),R1

该指令表示将寄存器R1中的值存回主存地址1f4h（即十进制500）处，即(address)←R1。

格式2：store offset(R2),R1

offset(R2)是内存地址，是将寄存器R2中的值加上偏移量表示的地址，R1是寄存器。也就是offset(R2)←R1。其中offset可以是十进制也可以是十六进制（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）。

【**示例2.1**】store 5(R2),R1

如果寄存器R2中最初存的值是200，则“5(R2)”表示的地址为205，该条指令则是将寄存器R1中的值存回到主存中地址为205的地方,即5(R2)←R1等同于(205) ←R1。

【**示例2.2**】store 5h(R2),R1

如果寄存器R2中最初存的值是200，则“5h(R2)”表示的地址为205，其中5h是十进制5的十六进制形式，该条指令则是将寄存器R1中的值存回到主存中地址为205的地方,即5h(R2)←R1等同于(205) ←R1。

### slt指令

**功能：“比较x是否小于y”操作。**

格式1：slt R3,R1,R2

该指令执行的操作，即比较寄存器R1中的数值是否小于R2中的数值，如果小于，则将寄存器R3置1，否则置0。

伪代码表示：

if R1＜R2

R3← 1

else

R3←0

【**示例**】slt R3,R1,R2

如果最初寄存器R1中存的值为2，寄存器R2中存的值为5，则比较大小后，寄存器R1中的数值较小，所以寄存器R3中的值为1；如果最初寄存器R1中存的值为10，寄存器R2中存的值为5，则比较大小后，寄存器R1中的数值较大，所以寄存器R3中的值为0。

格式2：slt R2,R1,constant

slt指令有三个操作数，其中前两个是寄存器，最后一个是常数，将寄存器R1中的数值与常数值constant比较，如果R1中的数值小于constant，则寄存器R4置1，否则置0。这里的常数可以是十进制也可以是十六进制。

伪代码表示：

if R1＜constant R2 ←1

else

R2 ←0

【**示例2.1**】slt R2,R1,10

如果寄存器R1中的数值为5，将5和10进行比较，5比较小，所以寄存器R2置1；如果寄存器R1中的数值为13，将13和10进行比较，13比较大，所以寄存器R2置0。

【**示例2.2**】slt R2,R1,Ah

如果寄存器R1中的数值为5，将5和Ah（即十进制10）进行比较，5比较小，所以寄存器R2置1；如果寄存器R1中的数值为13，将13和Ah（即十进制10）进行比较，13比较大，所以寄存器R2置0。

### sle指令

**功能：“判断小于等于”操作。**

格式1：sle R3,R1,R2

该指令执行的操作，即比较寄存器R1中的数值是否小于或者等于R2中的数值，如果小于或者等于，则将寄存器R3置1，否则置0。

伪代码表示：

if R1 ≤ R2

R3 ← 1

else

R3←0

【**示例**】sle R3,R1,R2

如果最初寄存器R1中存的值为2，寄存器R2中存的值为5，则比较大小后，寄存器R1中的数值较小，所以寄存器R3中的值为1；如果最初寄存器R1中存的值为5，寄存器R2中存的值为5，则比较大小后，两个寄存器中的数值相等，所以寄存器R3中的值为1；如果最初寄存器R1中存的值为10，寄存器R2中存的值为5，则比较大小后，寄存器R1中的数值较大，所以寄存器R3中的值为0。

格式2：sle R2,R1,constant

此sle指令有三个操作数，其中前两个是寄存器，最后一个是常数，将寄存器R1中的数值与常数值constant比较，如果R1中的数值小于或者等于constant，则寄存器R4置1，否则置0。这里的常数可以是十进制也可以是十六进制。

伪代码表示：

if R1≤constant R2←1

else

R2←0

【**示例2.1**】sle R2,R1,10

如果寄存器R1中的数值为5，将5和10进行比较，5比较小，所以寄存器R2置1；如果寄存器R1中的数值为10，将10和10进行比较，两个数相等，所以寄存器R2置1；如果寄存器R1中的数值为13，将13和10进行比较，13比较大，所以寄存器R2置0。

【**示例2.2**】sle R2,R1,Ah

如果寄存器R1中的数值为5，将5和Ah（即十进制10）进行比较，5比较小，所以寄存器R2置1；如果寄存器R1中的数值为10，将10和Ah（即十进制10）进行比较，两个数相等，所以寄存器R2置1；如果寄存器R1中的数值为13，将13和Ah（即十进制10）进行比较，13比较大，所以寄存器R2置0。

### beqz指令

**功能：“选择跳转到语句块”操作，表示CPU根据寄存器中的值（0或1）来判断执行哪一个语句块。**

格式：beqz R4,label

该指令有两个操作数，一个是存储比较结果的寄存器，一个是标签。指令beqz来查看寄存器中的值是否为0，如果为0，CPU将不再按照顺序执行下一条语句，而是跳转到另一个语句块。其中将要跳转的语句块，可以用一个“标签（Label）”来标记，指令块可以用L1、L2等标记。用来标记语句块的的标签格式为“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：。

【**示例**】beqz R4,L2

该指令表示，如果寄存器R4中的数值为0，则跳转到标签L2标记的指令块处，否则便顺序执行。

### goto指令

**功能：“直接跳转到语句块”指令，表示CPU直接跳转到其它语句块执行。**

格式：goto Label

该指令只有一个操作数，即“标签”label。即该指令表示调到label标签笔记的指令处执行。用来标记语句块的标签格式为“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：。

【**示例**】goto L3

该指令表示跳到标签L3标记的指令处执行。

**beqz指令和goto指令的比较：beqz指令是根据条件来选择是否跳转，goto指令是告诉CPU进行直接跳转的指令。**

### and指令

**功能：“按位与”操作。**

格式1： and R3,R2,R1

该指令需要三个操作数，将后两个寄存器中二进制数值的对应位进行与操作，即0与0为0,0与1为0,1与0为0,1与1为1，将按位与之后的数值存储在寄存器R3中。

【**示例**】and R3,R2,R1

如果寄存器R2中的值为5（二进制表示：101），寄存器R1中的值为6（二进制表示：110），则按位与的结果为4（二进制表示：100），则最后将结果4存储在寄存器R3中。

格式2：and R3,R2,constant

该指令有三个操作数，其中前两个是寄存器，最后一个是常数，将第二个寄存器中的二进制数值与常数量constant的二进制数值进行对应位做与操作，即0与0为0,0与1为0,1与0为0,1与1为0，将按位与之后的数值存储在寄存器R3中。这里的常数可以是十进制也可以是十六进制。

【**示例2.1**】and R3,R2,6

如果寄存器R2中的值为5（二进制表示：101），和常数6（二进制表示：110）进行按位与，结果为4（二进制表示：100），最后将结果4存储在寄存器R3中。

【**示例2.2**】and R3,R2,6h

如果寄存器R2中的值为5（二进制表示：101），和常数6h（二进制表示：110）进行按位与，结果为4（二进制表示：100），最后将结果4存储在寄存器R3中。

### or指令

**功能：“按位或”操作。**

格式1： or R3,R2,R1

该指令需要三个操作数，将后两个寄存器中的二进制数值的对应位进行或操作，即0或0为0,0或1为1,1或0为1,1或1为1，将按位或之后的数值存储在寄存器R3中。

【**示例**】or R3,R2,R1

如果寄存器R2中的值为5（二进制表示：101），寄存器R1中的值为6（二进制表示：110），则按位与的结果为7（二进制表示：111），则最后将结果7存储在寄存器R3中。

格式2：or R3,R2,constant

该指令需要三个操作数，其中前两个是寄存器，最后一个是常数，将第二个个寄存器中的二进制数值与常数量constant的二进制数值进行对应位做或操作，即0或0为0,0或1为1,1或0为1,1或1为1，将按位或之后的数值存储在寄存器R3中。这里的常数可以是十进制也可以是十六进制。

【**示例2.1**】or R3,R2,6

如果寄存器R2中的值为5（二进制表示：101），和常数6（二进制表示：110）进行按位或，结果为7（二进制表示：111），最后将结果7存储在寄存器R3中。

【**示例2.2**】or R3,R2,6h

如果寄存器R2中的值为5（二进制表示：101），和常数6h（二进制表示：110）进行按位或，结果为7（二进制表示：111），最后将结果7存储在寄存器R3中。

### xor指令

**功能：“按位异或”操作。**

格式1： xor R3,R2,R1

该指令需要三个操作数，将后两个寄存器中的二进制数值的对应位进行异或操作，即0 xor 0为0,0 xor 1为1,1 xor 0为1,1 xor 1为0，将按位异或之后的数值存储在寄存器R3中。

【**示例**】xor R3,R2,R1

如果寄存器R2中的值为5（二进制表示：101），寄存器R1中的值为6（二进制表示：110），则按位与的结果为3（二进制表示：011），则最后将结果3存储在寄存器R3中。

格式2：xor R3,R2,constant

该指令需要三个操作数，其中前两个是寄存器，最后一个是常数，将后第二个寄存器中的二进制数值与常数量constant的二进制数值进行对应位的异或操作，即0 xor 0为0,0 xor 1为1,1 xor 0为1,1 xor 1为0，将按位异或之后的数值存储在寄存器R3中。这里的常数可以是十进制也可以是十六进制。

【**示例2.1**】xor R3,R2,6

如果寄存器R2中的值为5（二进制表示：101），和常数6（二进制表示：110）进行按位或，结果为3（二进制表示：011），最后将结果3存储在寄存器R3中。

【**示例2.2**】xor R3,R2,6h

如果寄存器R2中的值为5（二进制表示：101），和常数6h（二进制表示：110）进行按位或，结果为3（二进制表示：011），最后将结果3存储在寄存器R3中。

### clz指令

**功能：“零计数”操作，统计最高位与第一个1之间的0的个数。**

格式：clz R2,R1

该指令需要两个操作数，均为寄存器操作数，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。

【**示例**】clz R2,R1

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。

### call指令

**功能：“调用函数”操作。**

格式：call Label

该指令只有一个“标签（label）”作为操作数，执行该指令后，CPU便跳到label标记的语句块处执行。用来标记语句块的标签格式为“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：。

【**示例**】call L2

执行该指令后，CPU便会跳到L2标记的语句块处执行。

### ret指令

**功能：“函数返回”操作。**

格式：ret

ret指令是无操作数指令，当函数执行完时，需要返回至原来执行的指令处，便需要ret指令跳回去。

【**示例**】ret

结合call指令，CPU跳到L2标记的语句块处执行，当遇到ret指令，会又跳回到call指令的下一条指令继续顺序执行。

**为了使大家在编写汇编代码时，能够更方便书写，给出以下巨指令（巨指令是指一条指令需要两条及以上的CPU真实指令才可以完成的操作，即使用一条巨指令可以完成几步操作）：**

### push指令

**功能：“压入”操作，将寄存器中的数值存入栈顶。**

格式：push R1

该指令为单操作数指令，将寄存器R1中的数值入栈，存入堆栈的栈顶，其内部的操作步骤为：①sub sp,sp,1 ②store 0(sp),R1。

【**示例**】push R1

如果寄存器R1中最初存储的数值为5，执行该指令后，栈顶所存储的数据为5。

### pop指令

**功能：“弹出”操作，将栈顶的数值取出赋给寄存器。**

格式：pop R1

该指令为单操作数指令，将堆栈的栈顶的值弹出，并放入寄存器R1中，其内部的操作步骤为：①load R1,0(sp)②add sp,sp,1。

【**示例**】pop R1

如果栈顶存储的值为6，栈顶加1位置处所存储的值为8，则将栈顶的6弹出，放入寄存器R1中，此时栈顶的值便不再是的6，而是8，并且R1中的值为6。

**以下给出的三条自定义指令以便大家输入、输出与调试：**

### \_data指令

**功能：一次存储多个数。**

格式：\_data first\_address,[a0,a1,a2,…,an]

该指令有两个操作数，第一个操作数为要存储一组数据的首地址，第二个操作数为所要存储的一组数，依次将该组数存储在以首地址开始递增的内存中，一般首地址都是比较小的地址。其中首地址以及存储的数据均可以使用十进制或者十六进制表示。

【**示例1.1**】\_data 11,[50,40,80,85,42,96,3,45,15,20,1,8,89]

所示的指令表示将[50,40,80,85,42,96,3,45,15,20,1,8,89]这组数存放在从内存地址11处开始递增的位置存储，即50存储在内存地址11处，40存储在内存地址12处，80存储在内存地址13处，依次进行存储。这里的首地址可以根据自己需要进行设置，例如也可以设置为0、100、210等没有被使用的内存地址。

【**示例1.2**】\_data Bh,[32h,28h,50h,55h,2ah,60h,3h,2dh,fh,14h,1f,8f,59h]

所示的指令表示将[32h,28h,50h,55h,2ah,60h,3h,2dh,fh,14h,1f,8f,59h]这组十六进制数存放在从内存地址Bh（即十进制11）处开始递增的位置存储，即32h（即十进制50）存储在内存地址Bh（即十进制11）处，28h（即十进制40）存储在内存地址Ch（即十进制12）处，50h存储在内存地址Dh（即十进制13）处，依次进行存储。这里的首地址可以根据自己需要进行设置，例如也可以设置为0h、64h（即十进制100）、d2h（即十进制210）等没有被使用的内存地址。

### \_pr指令

**功能：“打印”操作，将\_pr后的参数打印。**

【**示例1**】\_pr R1

R1为寄存器，此条指令表示，打印出R1寄存器存储的值。若R1存储的值为3，则打印出的结果为 3 。

【示例2】\_pr “R1:”,R1

此条指令中，第一个“R1：”是用双引号括起来的，表示这是一个字符串，也可以使用单引号，也同样表示这是一个字符串，指令会将这个字符串打印出来。第二个R1，跟示例1相同，表示的是寄存器R1，若R1存储的值为3，所以我们最终打印出的结果是 R1: 3 。

【**示例3.1**】\_pr “(100) : ”,(100)

此条指令中，第一项同示例2，表示一个字符串，第二项 (100) ,表示内存地址，如果在地址100处存储的值为2，所以我们最终打印的结果为 (100) : 2 。

【**示例3.2**】\_pr “(64h) : ”,(64h)

此条指令中，第一项同示例2，表示一个字符串，第二项 (64h) ,表示内存地址，如果在地址64h（即十进制100）处存储的值为2，所以我们最终打印的结果为 (64h) : 2 。

【**示例3.3**】\_pr (R15),1(R15)

此条指令中，第一项表示打印出寄存器R15记录的地址的数据，第二项比第一项在括号外多了个常数，表示偏移量，可以为负数。此条指令表示打印出寄存器R15记录的地址加上偏移量1处记录的数据，假设R15记录的信息是100，地址100存储的值为2，地址101存储的值为1，则最终打印结果为 2, 1 。

【**示例4.1**】\_pr (100)%4

此条指令表示，打印从地址100开始的4个地址的数据，也就是分别打印出100,101,102,103处存储的值。假如地址100到103存的值为分别为1,2,3,4，则我们最终打印出的结果为 100:1,101:2,102:3,103:4 。

【**示例4.2**】\_pr (64h)%4h

此条指令表示，打印从地址64h（即十进制100）开始的4个地址的数据，也就是分别打印出64h （即十进制100）,65h（即十进制101）,66h（即十进制102）,67h（即十进制103）处存储的值。假如地址64h （即十进制100）到67h（即十进制103）存的值为分别为1,2,3,4，则我们最终打印出的结果为 100:1,101:2,102:3,103:4 。

【**示例4.3**】\_pr 10(100)%3,-10(100)%3

此条指令和示例4.1相比，在括号前多了一个十进制数，它表示偏移量，可以为正，也可以为负。第一项里面偏移量为10，则表示打印出从地址100加上偏移量10，即地址110开始的3个地址存储的值，假如地址110到113存的值为分别为1,2,3,4。所以我们打印出的结果为111:2,112:3,113:4。第二项偏移量为-10，则表示打印出从地址100加上偏移量-10，即地址90开始的3个地址存储的值，假如地址90到94存的值为分别为0,1,2,3,4。所以我们打印出的结果为90:0,91:1,92:2。

【**示例4.4**】\_pr Ah(64h)%3h,-Ah(64h)%3h

此条指令和示例4.2相比，在括号前多了一个十六进制数，它表示偏移量，可以为正，也可以为负。第一项里面偏移量为Ah（即十进制10），则表示打印出从地址64h（即十进制100）加上偏移量A h（即十进制10），即地址6Eh（即十进制110开始的3个地址存储的值，假如地址6Eh（即十进制的110）到71h（即十进制113）存的值为分别为1,2,3,4。所以我们打印出的结果为111:2,112:3,113:4。第二项偏移量为-Ah（即十进制-10），则表示打印出从地址64h（即十进制100）加上偏移量-Ah（即十进制-10），即地址5Ah（即十进制90）开始的3个地址存储的值，假如地址5Ah（即十进制90）到5Eh（即十进制94）存的值为分别为0,1,2,3,4。所以我们打印出的结果为90:0,91:1,92:2。

【**示例5.1**】\_pr (R15)%4

此条指令表示，打印从寄存器R15记录的数据开始的4个地址的数据，假如R15存储的值为100，那么就会分别打印出100,101,102,103处存储的值。假如地址100到103存的值为分别为1,2,3,4，则我们最终打印出的结果为 100:1,101:2,102:3,103:4 。

【**示例5.2**】\_pr (R15)%4h

此条指令表示，打印从寄存器R15记录的数据开始的4h（即十进制4）个地址的数据，假如R15存储的值为100，那么就会分别打印出100,101,102,103处存储的值。假如地址100到103存的值为分别为1,2,3,4，则我们最终打印出的结果为 100:1,101:2,102:3,103:4 。

【**示例5.3**】\_pr 1(R15)%3,-1(R15)%3

此条指令和示例5.1相比，在括号前多了一个十进制数，它表示偏移量，可以为正，也可以为负，可以参照示例4.2。第一项里面偏移量为1，则表示打印出以寄存器R15记录的数据加上偏移量1为起始地址的3个连续地址的数据，假如R15存储的值为100，那么起始地址为101，即打印出从地址101开始的3个地址存储的值，假如地址100到103存的值为分别为1,2,3,4。所以我们打印出的结果为101:2,102:3,103:4。第二项偏移量为-1，则表示打印出以寄存器R15记录的数据加上偏移量-1为起始地址的3个连续地址的数据，假如R15存储的值为100，那么起始地址为99，即打印出从地址99开始的3个地址存储的值，假如地址99到103存的值为分别为0,1,2,3,4。所以我们打印出的结果为99:0,100:1,101:2。

【**示例5.4**】\_pr 1h(R15)%3h,-1h(R15)%3h

此条指令和示例5.2相比，在括号前多了一个十六进制数，它表示偏移量，可以为正，也可以为负，可以参照示例4.4。第一项里面偏移量为1h（即十进制1），则表示打印出以寄存器R15记录的数据加上偏移量1h（即十进制1）为起始地址的3h（即十进制3）个连续地址的数据，假如R15存储的值为100，那么起始地址为101，即打印出从地址101开始的3个地址存储的值，假如地址100到103存的值为分别为1,2,3,4。所以我们打印出的结果为101:2,102:3,103:4。第二项偏移量为-1h（即十进制-1），则表示打印出以寄存器R15记录的数据加上偏移量-1 h（即十进制-1）为起始地址的3h（即十进制3）个连续地址的数据，假如R15存储的值为100，那么起始地址为99，即打印出从地址99开始的3个地址存储的值，假如地址99到103存的值为分别为0,1,2,3,4。所以我们打印出的结果为99:0,100:1,101:2。

【**示例6.1**】\_pr 1(sp)%3,-1(sp)%3

此条指令和示例4和示例5类似，不同之处在于括号里面表示的是sp寄存器，可以参照示例4.3和示例5.3。第一项里面偏移量为1，则表示打印出以寄存器sp记录的数据加上偏移量1为起始地址的3个连续地址的数据，假如sp存储的值为100，那么起始地址为101，即打印出从地址101开始的3个地址存储的值，假如地址100到103存的值为分别为1,2,3,4。所以我们打印出的结果为101:2,102:3,103:4。第二项偏移量为-1，则表示打印出以寄存器sp记录的数据加上偏移量-1为起始地址的3个连续地址的数据，假如sp存储的值为100，那么起始地址为99，即打印出从地址99开始的3个地址存储的值，假如地址99到103存的值为分别为0,1,2,3,4。所以我们打印出的结果为99:0,100:1,101:2。

【**示例6.2**】\_pr 1h(sp)%3h,-1h(sp)%3h

此条指令和示例4和示例5类似，不同之处在于括号里面表示的是sp寄存器，可以参照示例4.3和示例5.3。第一项里面偏移量为1h（即十进制1），则表示打印出以寄存器sp记录的数据加上偏移量1 h（即十进制1）为起始地址的3 h（即十进制3）个连续地址的数据，假如sp存储的值为100，那么起始地址为101，即打印出从地址101开始的3个地址存储的值，假如地址100到103存的值为分别为1,2,3,4。所以我们打印出的结果为101:2,102:3,103:4。第二项偏移量为-1 h（即十进制-1），则表示打印出以寄存器sp记录的数据加上偏移量-1 h（即十进制-1）为起始地址的3 h（即十进制3）个连续地址的数据，假如sp存储的值为100，那么起始地址为99，即打印出从地址99开始的3个地址存储的值，假如地址99到103存的值为分别为0,1,2,3,4。所以我们打印出的结果为99:0,100:1,101:2。

### \_pause指令

**功能：“暂停”操作。**

格式：\_pause

该指令为无操作数指令，当汇编程序执行该指令后便会暂停，以便查看此时希望查看的寄存器的值，即设置断点。

【**示例**】\_pause

当在汇编程序的某条指令后加\_pause指令，在debug模式下运行汇编程序，遇到\_pause这条指令会暂停运行。

以下给出两个扩展指令，div和mul指令，这两个指令均是做整数运算的指令：

### div指令

**功能：除法操作，即对数值进行除法操作，求得商。**

格式1：div R2,R1,constant

div指令有三个操作数，前两个为寄存器的，最后一个为常数，这两个寄存器中后一个是进行运算的寄存器，前一个是存储运算结果的寄存器。表示R2 ← R1//constant。这里的常数可以是十进制也可以是十六进制。

【**示例1.1**】div R2,R1,2

该指令表示将寄存器R1中的数值除以2，并将结果存回寄存器R2。如果寄存器R1中的值最初是7，执行该指令后，寄存器R2中的值就为3，即R2← R1//3。

【**示例1.2**】div R2,R1,2h

该指令表示将寄存器R1中的数值除以2h（即十进制2），并将结果存回寄存器R2。如果寄存器R1中的值最初是7，执行该指令后，寄存器R2中的值就为3，即R2← R1//2。

格式2：div R3,R1,R2

div指令有三个寄存器操作数，表示将后两个寄存器R1、R2中的数做除法，结果赋给寄存器R3，也就是R3← R1//R2。

【**示例**】div R3,R1,R2

如果最初寄存器R1中存储的值为6，寄存器R2中存储的值为4，则执行该指令后寄存器R3中的值就为1，即为R3← R1//R2。

### mul指令

**功能：乘法操作，即对数值进行乘法操作。**

格式1：mul R2,R1,constant

mul指令有三个操作数，前两个为寄存器的，最后一个为常数，这两个寄存器中后一个是进行运算的寄存器，前一个是存储运算结果的寄存器。表示R2 ← R1×constant。这里的常数可以是十进制也可以是十六进制。

【**示例1.1**】mul R1,R1,2

该指令表示将寄存器R1中的数值乘2，并将结果存回寄存器R1。如果寄存器R1中的值最初是6，执行该指令后，寄存器R1中的值就为12，即R1← R1×2。

【**示例1.2**】mul R1,R1,2h

该指令表示将寄存器R1中的数值乘2h（即十进制2），并将结果存回寄存器R1。如果寄存器R1中的值最初是6，执行该指令后，寄存器R1中的值就为12，即R1← R1×2。

格式2：mul R1,R1,R2

mul指令有三个寄存器操作数，表示将后两个寄存器R1、R2中的值相乘，结果赋给寄存器R1，也就是R1← R1×R2。

【**示例**】mul R1,R1,R2

如果最初寄存器R1中存储的值为6，寄存器R2中存储的值为3，则执行该指令后寄存器R1中的值就为18，即为R1← R1×R2。

## 函数调用的介绍

**(请见SEAL教学说明文档的函数调用部分。)**

## 5. Debug的使用介绍

同学需要使用debug功能时，可以按照本章内容进行调试。

### 5.1 调试模式的选择与退出

* 1：同学输入数字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，如果同学在进入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）。

### 5.2 执行debug指令

我们在文件夹中看到有eg3\_for\_1\_debug.txt文件，本节将使用该文件来讲解 “debug”指令的执行过程。其中eg3\_for\_1\_debug.txt比在第3章中用到的eg3\_for\_1.txt文件多了三条“\_pause”指令，请同学打开文件对比，“\_pause”指令使得“debug”指令的执行过程有别于“normal”指令。如果同学想调试eg3\_for\_1.txt之外的其它汇编语言代码，请按照需求在汇编语言程序中加适量的“\_pause”指令。

当同学们按照3.3.2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，同学们得到的结果应该与图5-1相同，同时debug执行之后会弹出如图5-2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5-2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5-1 对eg3\_for\_1\_debug.txt执行debug命令

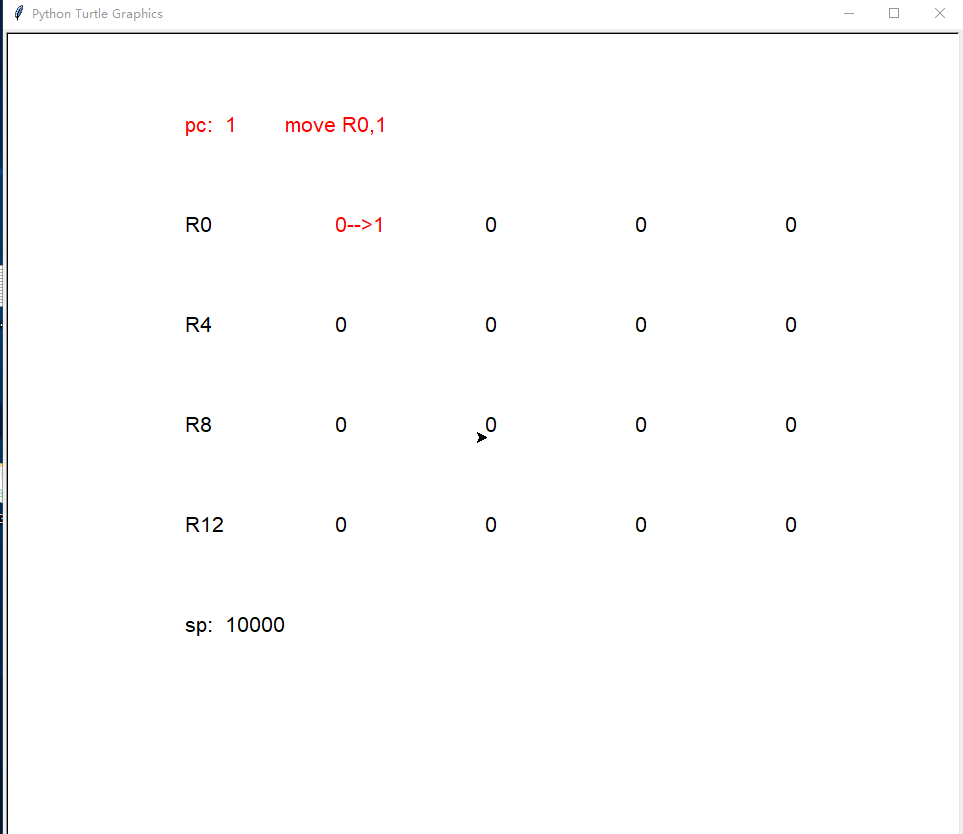


图5-2 调试结果显示窗口

从图5-2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5-2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，R1寄存器的内容将会变成1。

从图5-1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5-3所示内容。此时寄存器的值如图5-4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。

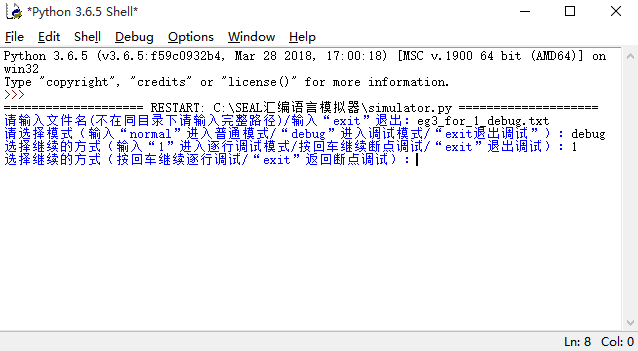


图5-3 逐行调试

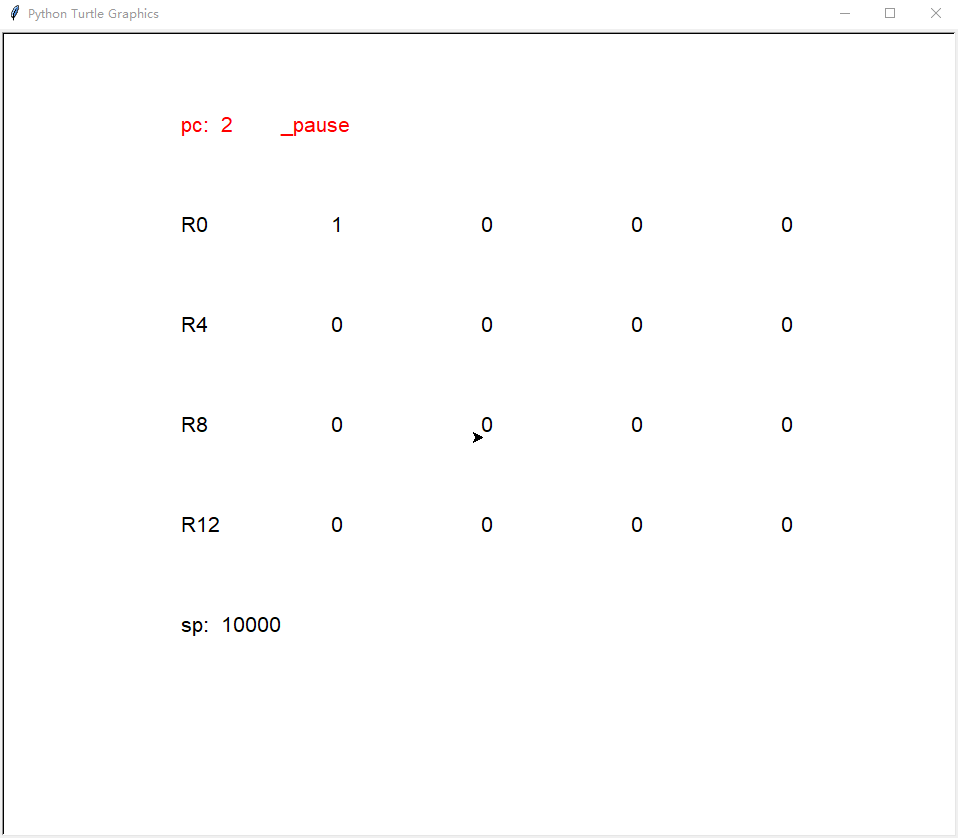


图5-4 逐行调试

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5-5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。

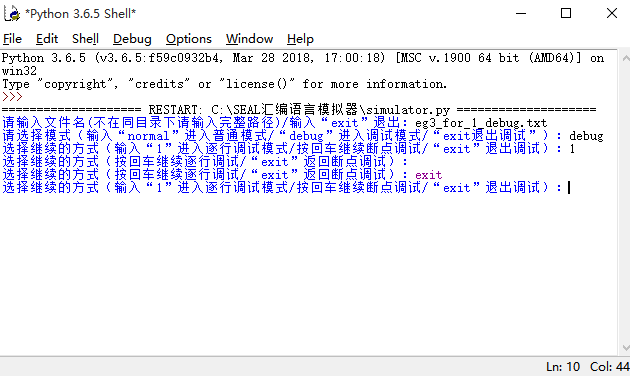


图5-5 退出逐行调试模式

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5-6所示，此时看到图5-6所示窗口顶部“pc: 6 sle R2,R0,10”,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5-6 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5-7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。

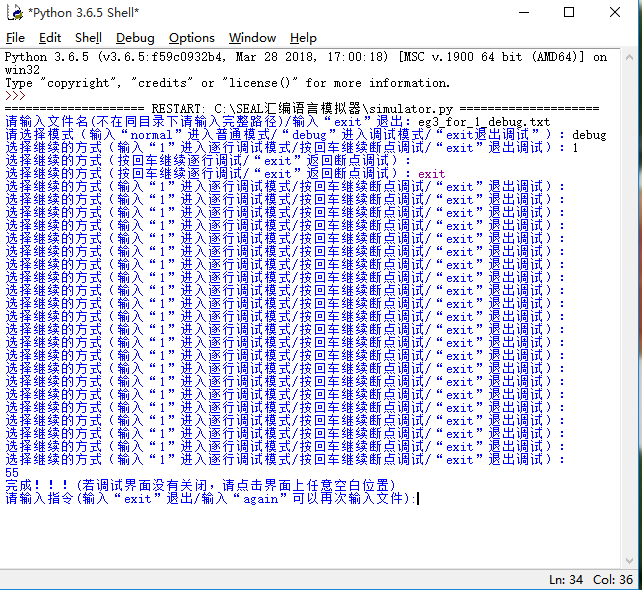


图5-7 调试结束

调试结束之后，如图5-7，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5-8所示。

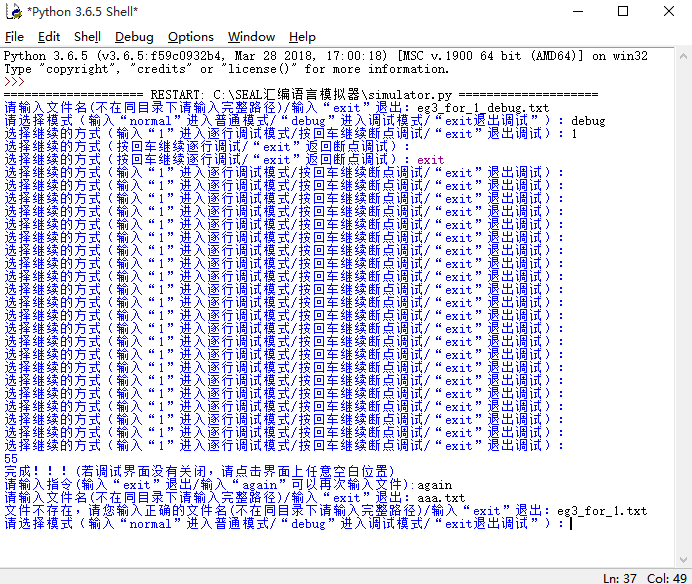


图5-8 继续再一次运行或调试

同时，对于eg3\_for\_1\_debug.txt文件，同学也可以执行“normal”指令，在SEAL的实现中，当执行“normal”指令时，会忽略文件中所有的“\_pause”指令，这使得“normal”指令的行为在这两类文件中一致，其执行结果如图5-9所示，同学可以看到执行结果同样输出了“55”和“完成！！！”字样。至此，同学们已经知道“normal”指令的使用方法，并且能够执行任何符合该手册所有汇编指令组成的程序。

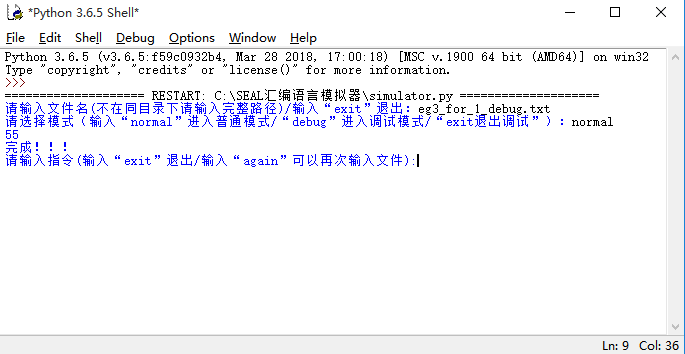


图5-9 对eg3\_for\_1\_debug.txt执行normal指令

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。

## 6. 程序示例

本节给出了9个示例，9个示例均被放在“SEAL汇编语言模拟器”文件夹内，参见图6-1。其中eg1\_if.txt和eg2\_if.txt是两个if语句的使用示例，其中eg1\_if\_data.txt和eg1\_if.txt只是存储的方式不同，eg2\_if.txt比eg1\_if.txt使用的条件更为复杂。eg3\_for\_1.txt、eg3\_for\_1\_debug.txt和eg4\_for\_2.txt给出两个使用for循环的例子，eg3\_for\_1\_debug.txt在eg3\_for\_1.txt基础之上增加了“\_pause”指令，用于debug模式时使用。eg5\_while.txt是while循环的使用示例。eg6\_multi.txt实现两个数的乘法。eg7\_call\_multi.txt使用函数调用实现乘法。eg8\_call\_min.txt通过多次函数调用求三个数的最小值。eg9\_acc.txt递归调用累加函数实现等差数列求和。接下来将分别说明每个示例的设计意图。

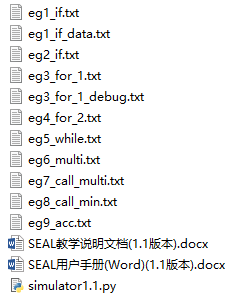


图6-1 SEAL汇编语言模拟器文件夹内容

1. 数字a与数字b比较大小，得到较小的数，存储在寄存器R1中。（eg1\_if.txt，eg1\_if\_data.txt）其中eg1\_if.txt是将两个数字放在寄存器中，eg1\_if\_data.txt使用\_data存储。

2. 判断一个点是否在矩形内。给出两个点的坐标，确定一个矩形，再给出一个点的坐标，判断该点是否在矩形内。（eg2\_if.txt）

示例1和示例2通过if条件语句，使得同学明白if语句的执行过程，不同条件可以与and、or等指令连接成更为复杂的条件。通过if条件分析语句了解计算机程序的执行过程。

3. 使用for循环计算1+2+3+4+5+6+7+8+9+10（eg3\_for\_1.txt）

4. 使用for循环计算a0+a1+a2+…+a9（eg4\_for\_2.txt）

示例3与示例4给出了for循环的示例，为了能够实现任意a0-a9数字的加法，SEAL的指令集中给出了\_data指令，它能够处理用户任意的输入。同学能够根据for循环的例子更加深入的了解程序的执行过程。

1. 使用while求a+d大于等于100时，a与i的值。设a与d的初始值都为1，a = a+d，d = d +1，求出a大于等于100时候是第几项，并且该值是多少。例如：1+1 = 2；2+2 =4；4+3 =7；7+4 =11…以此类推。

示例5给出了while循环的例子，对比while循环与for循环的例子，一方面能够让同学清楚Python中两种不同循环的执行机理，最重要的是通过这两种循环了解程序的执行过程，这也是我的教材《计算机科学导论-以Python为舟》初衷。

1. 对数字a与数字b做乘法，得到两个数的乘积

示例6通过两个数的乘法让同学了解真实的计算机中需要通过加法和移位操作来实现乘法。

**在以下的函数调用的示例中，均用R15表示fp。**

1. 调用乘法函数，计算3×5的结果
2. 三个数比较大小，求得最小值
3. 递归计算1+2+3+4+5的结果

示例7、示例8、示例9是三个函数调用的例子，通过这些例子，让同学们了解计算机如何处理函数调用，比如调用结束后如何返回主调函数、参数如何传递等问题，在第5节，通过get\_min函数讲解了函数调用的具体过程，请同学们仔细阅读。

## 附录A. 环境准备

Python具有跨平台特性，它能够运行在Windows、Linux/Unix和Mac等各种系统上。需要指出的是：该手册针对Windows操作系统。

当前Python主要分为两个版本，一个是2.x版本，另一个是3.x版本，这两个版本不兼容。由于3.x被在大型项目中广泛使用，因此，我们建议您的电脑上安装的是Python 3.x版本。

在安装Python之前，请明确您的Windows操作系统版本，Python官方指出3.5+版本不能在Windows XP甚至更早期的版本中使用，对于64位操作系统，请下载对应的Windows x86-64版本，对于32位操作系统，请下载Windows x86版本。链接https://www.python.org/downloads/release/python-365/ 给出Python 3.6.5的下载地址（请参见图2-1）。下载安装时，请勾选Add Python 3.6 to PATH。

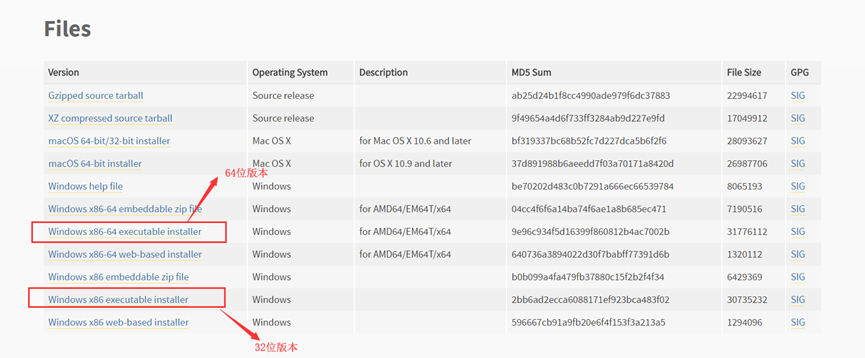


图 附录A-1 下载Python 3.6.5。对于Windows 64位操作系统用户，请下载Windows x86\_64 executable installer；对于Windows 32位操作系统用户，请下载Windows x86 executable installer。

安装成功之后，打开命令提示符窗口，敲入python后若出现图附录A-2所示内容，说明Python已经成功安装，恭喜您，可以继续阅读SEAL使用手册后续部分，并实践手册中的内容。

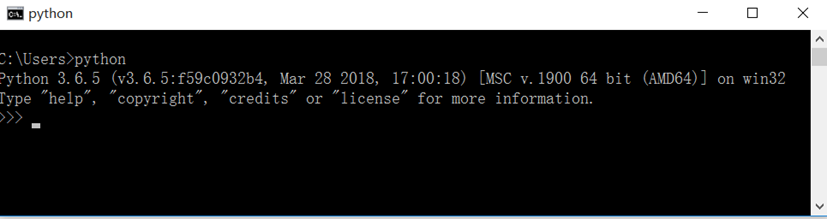


图 附录A-2 成功安装Python。

若出现提示：’python’不是内部或外部命令，也不是可执行的程序或批处理文件，说明系统中没有设置Python环境变量，此时需要手动设置python.exe的路径，如果不会设置环境变量，建议重新运行Python的安装程序，并务必勾选Add Python 3.6 to PATH。

至此环境搭建完成。