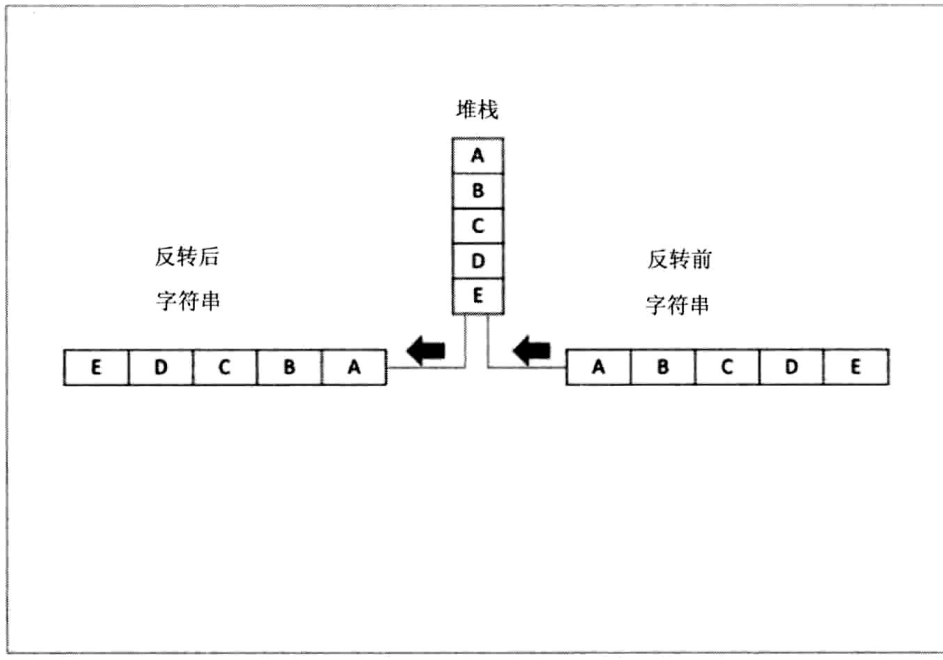

图 10-2　反转一个字符串的模式

然后，根据“后进先出”原则，你必须按如下方式弹出：

```
pop rcx
pop rbx
pop rax
```

除了寄存器之外，你还可以压入内存和即时值。可以弹出到寄存器或内存位置，但不能弹出到一个即时值，这非常明显。

可以使用 pushf 和 popf 指令将标志寄存器压入或弹出堆栈。

10.2　跟踪堆栈

跟踪堆栈很重要，而我们的老朋友 DDD 具有一些简单的功能可以做到这一点。首先，在编辑器中打开源代码，并删除 SASM 添加的调试行。然后保存文件并退出。在命令行上编译程序，然后输入以下内容：

```
ddd stack
```

在菜单栏中选择 Data | Status Displays，然后向下滚动，直至找到 Backtrace of the stack 并启用它。例如，在 main:处设置一个断点，然后在浮动面板中单击 Run。现在开始调试，并使用 step 按钮单步执行程序(相信你不希望通过 printf 函数逐行执

行)。在上方的窗口中查看堆栈的显示和更新方式。不必担心显示的初始内容。当你到达 push 指令之后的指令时，会看到字符以 ASCII 十进制表示(41、42 等)被压入堆栈。观察在第二个循环中堆栈如何减少。这是查看堆栈上的内容和顺序的简单方法。

图 10-3 展示了大致外观。

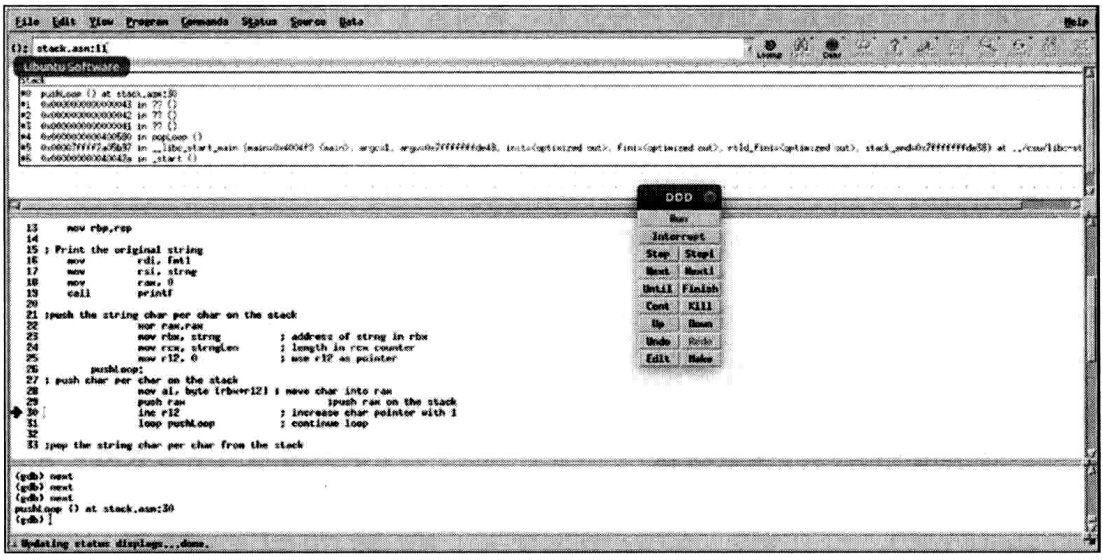

图 10-3 DDD 中显示的堆栈

如前所述，DDD 是开源的且已过时。无法保证它在将来继续按预期工作，但目前而言，它虽然不是很优雅，但基本可以满足使用需求。

你也可以强制 SASM 显示堆栈，但这需要更多的手工操作。它的工作原理是这样的：你可在 SASM 中进行调试时显示内存变量，堆栈只是内存位置的列表，而 rsp 指向最低位置。因此，必须让 SASM 展示地址 rsp 和内存位置之上的内容。图 10-4 展示了 SASM 中显示堆栈的示例内存窗口。

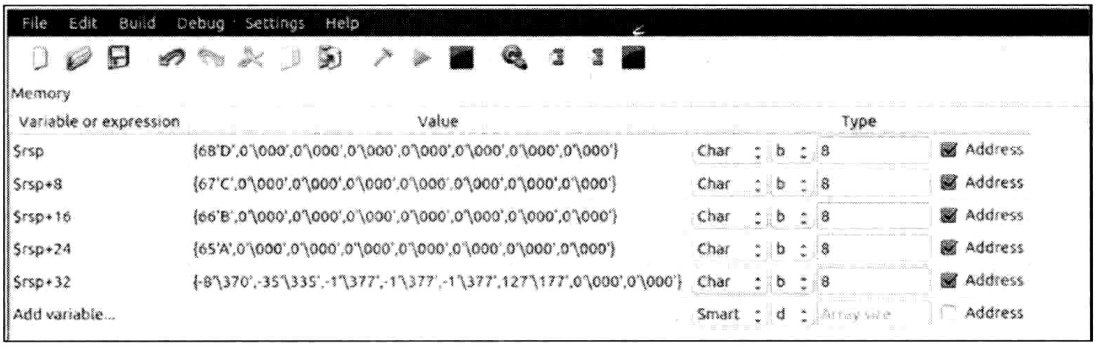

图 10-4 SASM 中显示的堆栈

我们将 rsp 称为$rsp。每次会将堆栈地址增加 8($rsp + 8)，因为每次 push 都将 8

个字节压入堆栈。对于类型，我们指定字符、字节、8 个字节和地址。选择字符是因为我们要压入一个字符串，这样易于阅读；选择字节是因为我们对字节值感兴趣(al 每次包含 1 个字节)，因此每次都压入 8 个字节。rsp 包含一个地址。单步执行该程序，然后查看堆栈如何改变。

它可以正常工作，但你必须手动详细说明每个堆栈的内存位置，如果你使用的是大型堆栈和/或要跟踪其他很多内存变量，这可能十分麻烦。

10.3 小结

本章内容：

- 堆栈从高位内存中的一个地址开始，然后增长到较低的地址。
- 递减栈指针(rsp)。
- 弹出栈指针(rsp)。
- 以相反的顺序进行压入和弹出。
- 如何使用 DDD 查看堆栈。
- 如何使用 SASM 查看堆栈。

第 11 章

浮点运算

你已经了解整数算术，现在介绍一些浮点运算。没有什么难理解的内容，浮点值中有一个小数点和零个或多个小数。有两种浮点数：单精度和双精度。双精度更精确，因为它可以处理更多有效数字。有了这些信息，你现在已经足够了解如何运行和分析本章中的示例程序了。

11.1 单精度与双精度

对于那些更好奇的人来说，故事是这样的。
单精度数字存储在 32 位中：1 个符号位，8 个指数位和 23 个小数位。

```
S    EEEEEEEE    FFFFFFFFFFFFFFFFFFFFFFF
0    1       8    9                      31
```

双精度数字存储在 64 位中：1 个符号位，11 个指数位和 52 个小数位。

```
S    EEEEEEEEEEE     FFFFFFFFFFFFF.....FFFFFFFFFF
0    1          11    12                          63
```

符号位很简单。当数字为正数时，符号位是 0。当数字为负数时，符号位为 1。指数位更复杂。下面看一个十进制的例子。

$200 = 2.0 \times 10^2$
$5000.30 = 5.0003 \times 10^3$

这是一个二进制示例：

$1101010.01011 = 1.0101001011 \times 2^6$ （将小数点向左移动六个位置）

指数可以是正、负或零。为了明确这一区别，在单精度情况下，在存储正指数之前将 127 加到正指数上。这意味着零指数将存储为 127。127 被称为偏差。使用双精度值时，偏差为 1023。

在上例中，1.0101001011 被称为有效位或尾数。根据假设，有效位的第一位是 1(是标准化的)，因此不会存储。

下面是一个简单例子，用来说明它是如何工作的。可使用 https://babbage.cs.qc.cuny.edu/IEEE-754/进行验证和实验。

单精度，十进制数 10：

- 十进制数 10 的二进制形式是 1010。
- 符号位为 0，因为数字是正数。
- 获取格式为 b.bbbb 的数字。1.010 是有效位数，根据需要以 1 开头。前导 1 将不被存储。
- 因此，指数是 3，因为我们将小数点移动了三个位置。因为指数为正，所以加了 127，得到 130，二进制值为 10000010。
- 因此，十进制单精度数字 10 将存储为：

```
0  10000010   01000000000000000000000
S  EEEEEEEE   FFFFFFFFFFFFFFFFFFFFFFF
```

或 41200000(十六进制)。

注意，相同值的十六进制表示形式在单精度和双精度方面有所不同。为什么不总是使用双精度并从高精度中受益呢？ 双精度计算比单精度计算要慢，并且操作数占用更多内存。

如果你认为这很复杂，你是对的。在互联网上找一个合适的工具来完成或至少对转换进行验证。

在较旧的程序中，你可能遇到 80 位浮点数，这些数字具有自己的指令，称为 FPU 指令。这是过去的遗留功能，不应在新的开发中使用。但是你会时不时地在互联网上的文章中看到 FPU 指令。

下面来做一些有趣的事情。

11.2 浮点数编程

代码清单 11-1 展示了示例程序。

代码清单 11-1：fcalc.asm

```
; fcalc.asm
```