不管程序最初是如何编写的（编写程序的方法有很多种），它最终都需要在CPU上作为一系列机器语言指令来执行。正如你所期望的，CPU指令可以归结为一系列0和1，就像计算机处理的所有其他事情一样。这里值得重复一下：不论程序最初是如何编写的，也不论用的是哪种编程语言，更不论使用的是什么技术，最后程序都会变成一系列的0和1，表示CPU可以执行的指令。

几年前，我有一项工作是关于诊断软件故障的。通常，我要分析发生在其他公司编写的软件中的问题。我没有这类软件的源代码，也没有很多关于它应如何工作的相关信息，但是我的工作就是确定软件故障的原因！我有个同事对此很淡然，他常常提醒我"这就是个代码"。换句话说，故障软件就是一组被CPU解释为指令的1和0。如果CPU可以理解代码，那么你也可以。

8.2 机器指令示例

我认为切入机器码主题的最简单方法就是看示例。我们来看看ARM系列处理器能理解的特定机器指令。你可能还记得，大多数智能手机上都能发现ARM处理器，所以你的手机可能也可以理解这条指令。

我们的示例指令告诉处理器把数字4移动到寄存器r7中，r7是ARM处理器通用寄存器中的一个。回忆一下我们之前讨论的计算机硬件，寄存器是CPU中的一种小容量存储位置。执行这个操作的ARM指令的二进制形式如下：

```
11100011101000000111000000000100
```

我们来看看ARM CPU是如何理解这条指令的，如图8-1所示。注意，我们跳过了无关的一些位。

详细说明：

- 条件 = 1110 = 总是执行（无条件）
- 立即数 = 1 = 指令的最后8位是值
- 操作码 = 1101 = 移动数值，通常表示为“mov”
- 目标寄存器 = 0111 = r7
- 立即数值 = 0000 0100 = 十进制数4

图8-1 ARM指令

条件部分指示了在什么条件下该指令应该执行。1110意味着这条指令是无条件执行的，所以CPU应该总是执行它。虽然本例中不是这样的，但是有些指令只需要在特定条件下执行。接下来的两位（本例中为00）与我们的讨论无关，所以跳过它们。立即数位告诉我们将要访问的是寄存器中的值，还是指令本身指定的值（称为立即数）。本例中，立即数位为1，要访问的是指令中指定的数。如果立即数位是0，那么被访问寄存器将由指令中的其他位来指定。操作码表示CPU要执行的操作。本例中，操作码是mov，意思是CPU要移动数据。目标寄存器0111告诉我们把值移动到寄存器r7（0111是十进制的7）中。最后，立即数值00000100是十进制的4，这是我们想要移动到r7中的数。简单概括一下，这个二进制序列告诉ARM CPU把数字4移动到r7寄存器中。

CPU总是用二进制处理各种事情，但是绝大多数人很难处理这些0和1。现在，我们用十六进制表示同样的指令，以便让它更容易读懂：

```
e3a07004
```

现在是不是好点啦？好吧，也许不是。这种格式比二进制格式更加紧凑，也更容易区分，但它的含义仍然不明显。对我们而言，幸运的是还有另一种方法来表示这条指令：汇编语言。汇编语言是一种编程语言，它的每一条语句都直接表示一条机器语言指令。每种类型的机器语言都有对应的汇编语言——如x86汇编语言、ARM汇编语言等。汇编语言语句由一个表示CPU操作码的助记符和其他必需的操作码（比如寄存器或数值）组成。助记符是人类可读的操作码形式，它允许汇编语言程序员在代码中用mov而不是1101。之前讨论过的相同的ARM指令也可以用如下汇编语言语句表示：

```
mov r7, #4
```

与相应的二进制和十六进制表示相比，这个语句当然是表示"把4移动到r7寄存器中"的更好的方式！最起码对人类来说更容易阅读。也就是说，汇编语言语句只是为了人类方便。CPU从不会执行文本格式的指令，它只处理二进制格式的指令。如果程序员用汇编语言编写程序，那么在计算机执行该程序之前，汇编指令仍然必须转换成机器码。这要用"汇编器"来完成，汇编器是一种把汇编语言语句转换成机器码的程序。汇编语言文本文件被送入汇编器，汇编器输出的就是包含机器码的二进制目标文件，如图8-2所示。

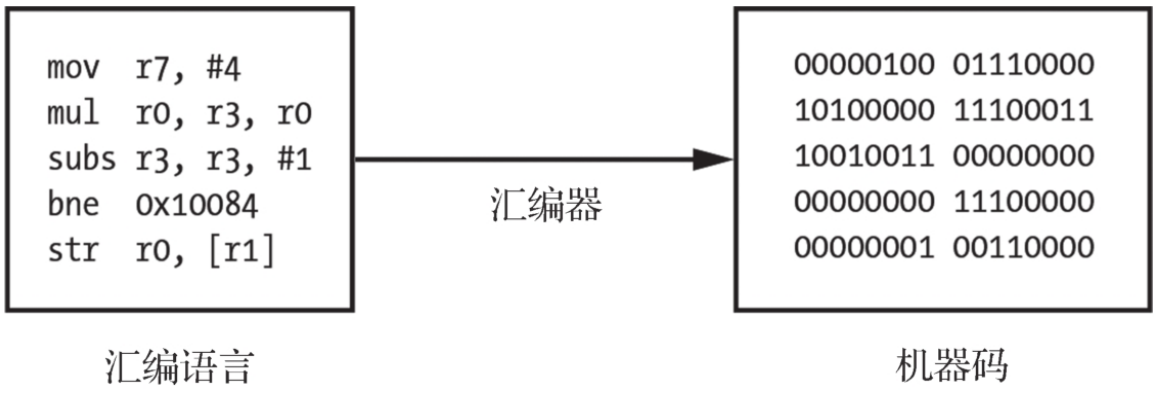

图8-2 汇编器把汇编语言转换成机器码

8.3 用机器码计算阶乘

现在我们已经查看了单条ARM指令，接下来我们看看如何把多条指令组合在一起来执行有用的任务。我们来看一个计算整数阶乘的ARM机器码。你可能还记得在数学课上学过，*n*的阶乘（写作*n*!）是小于或等于*n*的正整数的乘积。例如，4的阶乘为

$$4!=4\times3\times2\times1=24$$

现在我们已经有了阶乘的定义，让我们来看看用ARM机器码如何实现阶乘算法。为了简单起见，我们不查看完整的程序代码，只看执行阶乘算法的部分。我们假设*n*的初始值保存在寄存器r0中，并且当代码完成时，计算结果也保存在r0中。

如同计算机处理的任何其他数据一样，在CPU访问机器码之前，必须把它加载到内存中。下面的是机器码的32位（4字节）十六进制表示，以及每个值的内存地址。

```
地址      数据
0001007c  e2503001
00010080  da000002
00010084  e0000093
00010088  e2533001
0001008c  1affffc
```

当我们的代码加载到内存时，阶乘逻辑从地址0001007c开始。来看看从这个地址开始的内存内容。请注意：0001007c不是一个神奇的地址；它只是恰好是本例中代码加载的地方。还要注意，内存地址值增幅为4，因为每个数据值都需要4个字节来存放。每条ARM指令长度都为4字节，所以这里的数据表示5条ARM指令。

把这些指令看作十六进制的值并不能让我们深入理解它们的含义，所以让我们对这些指令进行译码以便理解这个程序。在下面的清单中，我已经把十六进制的数值转换成相应的汇编语言助记符。如果你好奇的话，把机器语言手动翻译成汇编语言不是你需要了解的！我们有完成这项功能的软件，称为“反汇编器”。现在，这本书就充当了你的反汇编器。下面列出了每条指令，以及与之配对的汇编语句：

```
地址       数据       汇编
0001007c   e2503001   subs r3, r0, #1
00010080   da000002   ble  0x10090
00010084   e0000093   mul  r0, r3, r0
00010088   e2533001   subs r3, r3, #1
0001008c   1afffffc   bne  0x10084
00010090   ---
```

CPU按顺序执行这些指令，直到遇到分支指令（比如ble或bne），分支指令会让它跳转到程序的另一个部分。地址00010090标记着阶乘逻辑的结束。一旦到达这个地址，阶乘运算结果就已经保存到r0了。此时，CPU执行位于地址00010090处的指令。

你可能好奇这些指令是如何表示阶乘的计算的。对大多数人来说，粗略地看一下这些指令不足以理解其所含的意思。采取循序渐进的方式，在每条指令执行时跟踪寄存器的值能帮助你理解这个程序。我将给你提供一些必需的背景资料，然后你可以尝试评估这个程序是如何工作的。

为了理解这个程序，你首先需要对每条指令的说明。在表8-1中，我已经给出了这个程序中每条指令的解释。在这个表中，我使用了寄存器的占位符名称，比如***Rd***和***Rn***。当你查看汇编代码时，你将看到实际使用的寄存器名称，比如r0和r3。代码中列出的操作数顺序对应于表8-1中的操作数顺序。例如，subs r3，r0，#1表示从存储在r0中的值中减去1，并把结果存放到r3中。

表8-1 一些ARM指令的解释

指令	详细信息
`subs` ***Rd***，***Rn***，`#Const`	**减法** 将存储在寄存器***Rn***中的值减去常数值`Const`，并把结果存放到寄存器***Rd***中。换句话说：***Rd***=***Rn***-`Const`
`mul` ***Rd***，***Rn***，***Rm***	**乘法** 把存储在寄存器***Rn***中的值和存储在寄存器***Rm***中的值相乘，结果存放在寄存器***Rd***中。换句话说：***Rd***=***Rn*** × ***Rm***
`ble` ***Addr***	**如果小于或等于，则跳转到分支指令** 如果前面的操作结果小于或等于0，那么跳转到位于地址***Addr***处的指令，否则，继续执行下一条指令
`bne` ***Addr***	**如果不相等，则跳转到分支指令** 如果前面的操作结果不等于0，那么跳转到位于地址***Addr***处的指令，否则，继续执行下一条指令

分支和状态寄存器

分支指令实际并不查看前一条指令的数值结果。和大多数CPU一样，ARM处理器也有一个寄存器专门跟踪状态。这个状态寄存器有32位，每个位对应一个特定的状态标志。例如，位31是N标志，当指令结果为负数时，该标志置1。只有某些指令会影响这些标志状态。例如，subs指令会改变标志的状态。如果某个减法运算结果是负的，那么N标志会置1，否则，它会清零。其他指令（包括分支指令）则查看状态标志位来决定该做什么。这看上去是一种迂回的方式，但实际上，它简化了像bne这样的指令——处理器可以根据单个位的值进行分支（或不进行分支）。

关于这个主题的解释已经讲完了，本章剩余部分包含一个练习和两个设计。在练习8-1中，你将用表8-1中的详细信息来完成阶乘程序，以理解每条指令是如何工作的。

练习8-1：把自己的大脑当作CPU

尝试自己运行如下ARM汇编程序：

```
地址       汇编指令
0001007c  subs r3, r0, #1
00010080  ble  0x10090
00010084  mul  r0, r3, r0
00010088  subs r3, r3, #1
0001008c  bne  0x10084
00010090  ---
```

假设输入值*n*=4最初保存在r0中。当程序运行到00010090处的指令时，就表明已经到达本代码的末尾，且r0中将是预期的输出值24。建议对于每条指令，在其执行前后都跟踪r0和r3的值。遍历这些指令，直到到达地址00010090处的指令，看看是否得到了预期的结果。如果一切正常，你应该多次循环执行相同的指令，这是有意为之的。

在纸上学习汇编语言是很好的开始，但是在计算机上尝试汇编语言会更好。

注意

请参阅设计12编写阶乘运算的汇编代码并在其执行时进行检查。此外，请参阅设计13学习一些检查机器码的其他方法。

8.4 总结

本章讨论了机器码，即特定于CPU的一系列指令，它们在内存中以字节表示。你学习了如何对示例ARM处理器指令进行编码，以及如何用汇编语言表示该指令。你了解到汇编语言是一种源代码，尤其是人类可读形式的机器码。我们还了解了怎样组合多条汇编语言语句来执行有用的操作。

第9章将讨论高级编程语言。这种语言提供了CPU指令集的抽象，允许开发人员编写更容易理解且可以在不同计算机硬件平台之间移植的源代码。

设计12：汇编语言中的阶乘运算

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。建议参考附录B了解如何使用Raspberry Pi操作系统，包括如何处理文件，这在本章设计中会普遍用到。

本设计中，你将用汇编语言构建一个阶乘程序。然后你将查看生成的机器码。除了本章已经给出的代码之外，阶乘程序还包含一些其他的代码。具体说来，程序还要从内存读取*n*的初始值，把结果写回到内存中，并在最后把控制权交还给操作系统。

汇编指令和伪指令

由于要包含其他代码，因此表8-2解释了代码中用到的各种指令。你已经在表8-1中看到了其中的一些指令，但是表8-2中仍包含了全部指令以方便你参考。

表8-2 设计12中使用的ARM指令

指令	详细信息
`ldr` ***Rd***, ***Addr***	**从内存加载到寄存器** 读取地址 ***Addr*** 中的值并将其放入寄存器 ***Rd***
`str` ***Rd***, ***Addr***	**把寄存器的值存入内存** 把寄存器 ***Rd*** 中的值写入地址 ***Addr***
`mov` ***Rd***, #***Const***	**把常数移动到寄存器** 把常数值 ***Const*** 移动到寄存器 ***Rd*** 中
`svc`	**进行系统调用** 向操作系统发出请求
`subs` ***Rd***, ***Rn***, #***Const***	**减法** 将存储在寄存器 ***Rn*** 中的值减去常数值 ***Const***，并把结果存放到寄存器 ***Rd*** 中。换句话说：***Rd*** = ***Rn***−***Const***
`mul` ***Rd***, ***Rn***, ***Rm***	**乘法** 把存储在寄存器 ***Rn*** 中的值和存储在寄存器 ***Rm*** 中的值相乘，结果存放在寄存器 ***Rd*** 中。换句话说：***Rd*** = ***Rn*** × ***Rm***
ble Addr	**如果小于或等于，则跳转到分支指令** 如果前面的操作结果小于或等于 0，那么跳转到位于地址 ***Addr*** 处的指令，否则，继续执行下一条指令
`bne` ***Addr***	**如果不相等，则跳转到分支指令** 如果前面的操作结果不等于 0，那么跳转到位于地址 ***Addr*** 处的指令，否则，继续执行下一条指令

当用汇编语言编写代码时，开发人员还会使用汇编伪指令。它们不是ARM指令，而是给汇编器的命令。这些伪指令以句点开头，所以很容易把它们与指令区分开来。在下面的代码中，你还可以看到文本后面跟着冒号——这些是标签，是给内存地址的名称。由于在编写代码的时候我们不知道指令会位于内存的什么位置，因此我们用标签而不是内存地址来引用内存位置。还有一件事情需要注意：@符号表示跟在其后面（同一行）的文本是注释。我已经用注释来解释程序，但是如果你愿意的话，也可以跳过注释。

输入并检查代码

这些背景知识足够了，毕竟这只是个设计！是时候输入代码了。在你的主文件夹根目录下，用你选择的文本编辑器创建一个名为fac.s的新文件。附录B中有Raspberry Pi文档，其中介绍了Raspberry Pi操作系统中使用文本编辑器的详细步骤。把如下ARM汇编代码输入文本编辑器中（无须保留缩进和空行格式，但一定要保留换行符，虽然额外的换行符不会有问

题）。如果你还未理解这段代码，也不用担心，我会在代码的后面解释你需要理解的内容。

```
.global _start❶

.text❷
_start:❸
  ldr  r1, =n      @ set r1 = address of n❹
  ldr  r0, [r1]    @ set r0 = the value of n
  subs r3, r0, #1  @ set r3 = r0 - 1
  ble  end         @ jump to end if r3 <= 0
loop:
  mul  r0, r3, r0  @ set r0 = r3 x r0
  subs r3, r3, #1  @ decrement r3
  bne  loop        @ jump to loop if r3 > 0
end:
  ldr  r1, =result @ set r1 = address of result❺
  str  r0, [r1]    @ store r0 at result

@ Exit the program
  mov  r0, #0❻
  mov  r7, #1
  svc 0

.data❼
  n: .word 5❽
  result: .word 0
```

输入代码后，在主文件夹根目录下，在文本编辑器中将其保存为fac.s。让我们从伪指令和标签开始，完整地看看这段代码。

如前所述，文本后面跟一个冒号（例如_start：）表示内存位置的标签❸。标签_start标记程序开始执行的点。.global伪指令使得_start标签对于链接器而言是可见的❶（我们马上就会说到链接器），因此它可以被设置为程序的入口点。.text伪指令告诉汇编器，它后面的行是指令❷。

在代码结束的地方，.data伪指令告诉汇编器，它后面的行是数据❼。在数据部分，程序存放了两个32位的值，每个值都用.word伪指令来指示❽。第一个是*n*的值，初值设置为5。第二个是result，初值设置为0。这里的“word”表示4个字节或32位。

现在来看看除了本章内容之外，在代码中添加的功能部分。我们已经有了从内存加载*n*、把阶乘结果保存到内存，以及退出程序的代码。_start中的前两条指令从内存位置加载*n*的值❹。ldr指令把数值加载到寄存器。我

们用=n来引用*n*的地址。下一行中，[r1]带有方括号，这是因为r1中是个地址，而程序要访问存放在这个地址中的值。

end后面的两条指令是把结果保存到内存位置中的指令❺。第一条指令把名为result的内存位置的地址移动到r1寄存器中。之后，代码把r0寄存器中的值（恰好是计算得到的阶乘结果）保存到result内存地址，这个地址由r1引用。

.text部分的最后三条指令用于干净地退出程序❻。这需要操作系统的帮助，所以我将跳过这些指令的细节，直到第10章再讨论操作系统。

汇编、链接和运行

你现在有了一个汇编语言指令的文本文件，但这不是计算机能运行的形式。你需要通过两个步骤把汇编语言指令转换成机器码字节。首先，你需要用汇编器把指令转换（汇编）成机器码字节。这一步的结果是一个目标文件，它包含了字节程序，但仍然不是最终程序运行所需的格式。接下来，你需要使用被称为"链接器"的程序把目标文件转换成操作系统能运行的可执行文件。这个过程如图8-3所示。

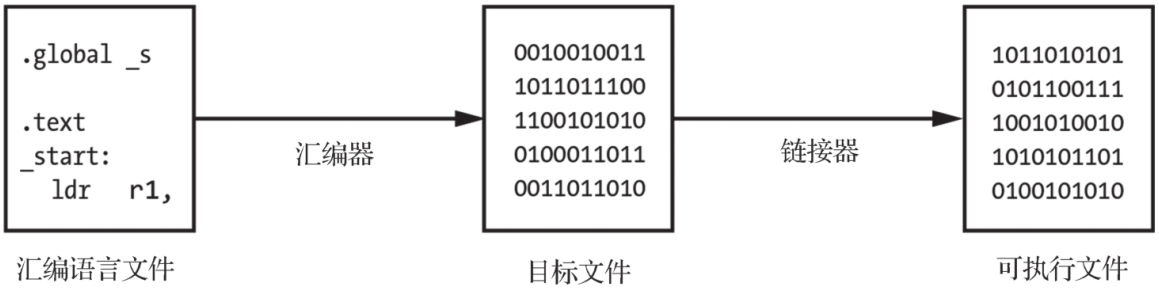

图8-3 经过汇编和链接生成可执行文件

你可能会好奇为什么需要这两步。如果你在汇编作为一个程序而一起工作的多个源文件，那么每个源文件都将汇编成一个目标文件。然后，用链接器把各个目标文件组合成一个可执行文件。这允许之前创建的目标文件在需要的时候链接。本例中，只有一个目标文件，链接器只需简单地把它转换成可执行的格式。

现在，汇编代码：

```
$ as -o fac.o fac.s
```

as工具是GNU汇编器，它把汇编语言语句转变成机器码。这个命令把生成的机器码写到名为fac.o的文件中，fac.o是一个目标文件。如果fac.o文件没有用换行符结束，汇编器可能会发出一个警告——你可以安全地忽略这个警告。

源代码被汇编成目标文件后，你需要用GNU链接器（ld）把目标文件转换成可执行文件：

```
$ ld -o fac fac.o
```

这个命令把fac.o作为输入，然后输出一个名为fac的可执行文件。此时，你可以用如下命令来运行程序：

```
$ ./fac
```

这个命令会立即返回下一行，没有输出。这是因为程序实际上没有在屏幕上显示任何文本，它只计算阶乘，把阶乘结果保存到内存，然后退出。要与用户交互，程序需要向操作系统请求一些帮助。但是，由于我们试图让这个程序尽可能地最小化，因此你不需要做这些。

用调试器加载程序

既然程序没有任何输出，那你怎么知道它在做什么？你可以使用调试器，它是一个可以在进程运行时对其进行检查的程序。调试器可以附加到正在运行的程序上，然后暂停其执行。当程序暂停时，调试器可以查看寄存器和目标进程的内存。在这里，你可以使用GNU调试器gdb，目标程序是fac程序。

首先，执行如下命令：

```
$ gdb fac
```

当运行这个命令时，gdb加载fac文件，但不执行指令。当出现（gdb）提示符时，输入如下命令查看程序的起始地址：

```
(gdb) info files
```

你应该看到这样的一行输出，不过具体地址可能有所不同：

```
Entry point: 0x10074
```

这就告诉你程序的入口点是地址0x10074。请记住，在你写程序时，你并不知道使用哪些内存地址，所以你用标签来代替地址。既然程序已经构建好并加载到内存，便可以查看实际的内存地址了。入口点地址对应于_start标签，因为那是程序开始的地方。现在，你可以使用gdb从程序入口点开始反汇编机器码。反汇编是把机器码字节看作汇编语言指令的过程。下面的命令使用0x10074作为起始地址（如果你的入口点不一样，则使用你自己的地址）：

```
(gdb) disas 0x10074
Dump of assembler code for function _start:
   0x00010074 <+0>:     ldr     r1, [pc, #40]   ; 0x100a4 <end+20>
   0x00010078 <+4>:     ldr     r0, [r1]
   0x0001007c <+8>:     subs    r3, r0, #1
   0x00010080 <+12>:    ble     0x10090 <end>
```

运行这个命令后，你应该看到前四条指令被反汇编。默认情况下，gdb只反汇编少量指令。这是一个很好的开始，但看到整个程序会更好。为此，你需要告诉gdb你想看到的代码的结束地址。如果看一下之前输入fac.s中的代码，你会看到在程序中总共有12条指令。每条指令是4个字节，所以程序的长度应该是48个字节。这意味着程序应该在起始地址之后48个字节结束，所以结束地址应该是0x00010074+48。你可以自己来做这个加法，但由于在gdb中，因此你可以要求它来做这个计算并找到程序的结束地址（如果需要的话，用你的入口点地址来替换0x10074）：

```
(gdb) print/x 0x00010074 + 48
$1 = 0x100a4
```

开始的时候，print命令输出可能会有点令人费解。命令中/x的含义是“打印用十六进制表示的结果”。如果你查看输出，你会看到左边的值

（$1）是一个为了方便而临时设置的变量，它是gdb中的一个暂存位置。把值保存到方便的变量中是gdb让你之后能轻松返回该结果的方式。等号后面的值是要打印的值，即计算的结果，本例为0x100a4。

现在你知道了结束地址（0x100a4），你可以要求gdb反汇编完整的程序。请注意，如果你的起始地址与本例的不同，你需要在下面的命令中替换这两个地址：

```
(gdb) disas 0x10074,0x100a4
Dump of assembler code from 0x10074 to 0x100a4:
   0x00010074 <_start+0>:       ldr     r1, [pc, #40]   ; 0x100a4 <end+20>❶
   0x00010078 <_start+4>:       ldr     r0, [r1]
   0x0001007c <_start+8>:       subs    r3, r0, #1
   0x00010080 <_start+12>:      ble     0x10090 <end>
   0x00010084 <loop+0>: mul     r0, r3, r0
   0x00010088 <loop+4>: subs    r3, r3, #1
   0x0001008c <loop+8>: bne     0x10084 <loop>
   0x00010090 <end+0>:  ldr     r1, [pc, #16]   ; 0x100a8 <end+24>❷
   0x00010094 <end+4>:  str     r0, [r1]
   0x00010098 <end+8>:  mov     r0, #0
   0x0001009c <end+12>: mov     r7, #1
   0x000100a0 <end+16>: svc     0x00000000
```

这看起来非常像你刚开始输入fac.s并汇编的内容，只不过现在每条指令都被分配了地址，而且对*n*和result的引用也被替换成相对于程序计数器的内存偏移量（例如，[pc, #40]❶）。程序计数器或指令指针存放的是当前指令的内存地址。为了简单起见，我不会详细说明为什么这里使用程序计数器偏移量，你只要知道0x10074❶和0x10090❷处的指令分别把*n*和result的内存地址加载到r1中。

运行并用调试器断点检查程序

现在程序已经加载到内存，让我们看看程序是否按预期工作。为此，你需要在某些指令上设置断点，这能让你在断点处查看程序的状态。断点告诉调试器在到达某个地址时暂停执行。在某个地址上设置断点可使程序在执行相应指令之前立即停止执行。在下面的示例命令中，我使用了系统上显示的地址，如果你的内存地址与之不同，请确保使用你自己的地址。

你将设置如下断点：

❑**0x10074** 程序开始。

❑**0x1007c** 阶乘逻辑的开头，第一条指令后面的8个字节。当程序到达这条指令时，寄存器r0应该是输入值*n*，在程序中该值被硬编码为5。

❑**0x10090** 阶乘逻辑的结尾，第一条指令后面的0x1C个字节。当程序到达这条指令时，寄存器r0应该保存计算出来的阶乘值。

❑**0x100a0** 程序的最后一条指令。当程序到达这条指令时，标记为result的内存位置应保存阶乘结果。

按下面所示设置断点（如果你的起始地址不是0x10074，请调整地址）：

```
(gdb) break *0x10074
(gdb) break *0x1007c
(gdb) break *0x10090
(gdb) break *0x100a0
```

现在，开始运行程序：

```
(gdb) run
Starting program: /home/pi/fac

Breakpoint 1, 0x00010074 in _start ()
```

你将看到如上所示的输出，表示程序在第一个断点停止。此时，程序准备执行第一条指令，你可以查看一下当前状态。首先，查看一下寄存器，实际上此时我们唯一关心的是程序计数器（pc），因为我们想要确认当前指令在起始地址0x10074处。现在，让调试器显示pc寄存器的值：

```
 (gdb) info register pc
pc             0x10074  0x10074 <_start>
```

这告诉你，和预期的一样，程序计数器指向起始地址和第一个断点。另一种确认当前指令的方法是简单地反汇编代码，如下所示：

```
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010074 <+0>:     ldr     r1, [pc, #40]   ; 0x100a4 <end+20>
   0x00010078 <+4>:     ldr     r0, [r1]
   0x0001007c <+8>:     subs    r3, r0, #1
   0x00010080 <+12>:    ble     0x10090 <end>
```

注意

=>符号表示当前指令。

现在你确认了程序已经准备好执行其第一条指令，你可以查看两个有标记的内存地址n和result的当前值。它们分别应该是5和0，因为这是你在fac.s中给它们定义的初始值。你可以再次使用print命令来查看这些值。当你这样做的时候，你需要将数据类型定义为int（32位整数），这样print命令就知道如何显示这些值。

```
(gdb) print (int)n
$2 = 5
(gdb) p (int)result
$3 = 0
```

注意在第二个命令中p是如何代替print的。gdb支持命令缩写，这可以节省打字时间。如你所见，print命令使得输出有标记内存位置的值更加容易！

虽然输出已标记内存位置的值很方便，但这确实带来了一个问题：gdb中的print命令如何得知你在原始fac.s文件中给这些内存位置提供的标签？CPU不使用这些标签，它只使用内存地址。机器码也不通过名称引用这些内存位置。调试器之所以可以这样做是因为文件fac既保存了机器码，也保存了符号信息。这些调试符号告诉调试器某些命名的内存位置，比如n和result。通常，在把可执行文件分发给最终用户之前会移除符号信息，但是fac可执行文件中仍存有符号信息。

请记住，n和result只是内存位置的标签，你如何找到这些变量的实际内存地址呢？一种方法是使用&运算符输出地址，在gdb中它表示“……的地址”。因此，&n的意思是“n的地址”。现在，输出n的地址和result的地址。