算，控制器根据该运算结果来控制计算机。看到“控制”一词时，大家可能会将事情想象得过于复杂，其实所谓的控制就是指数据运算以外的处理（主要是数据输入输出的时机控制）。比如内存和磁盘等媒介的输入输出、键盘和鼠标的输入、显示器和打印机的输出等，这些都是控制的内容。

1.2　CPU 是寄存器的集合体

CPU 的四个构成部分中，程序员只需要了解寄存器即可，其余三个都不用太过关注。那么，为什么必须要了解寄存器呢？这是因为**程序是把寄存器作为对象来描述的**。

首先我们来看一下代码清单 1-1。这是用汇编语言（assembly）[①]编写的程序的一部分。**汇编语言**采用助记符（memonic）来编写程序，每一个原本是电气信号的机器语言[②]指令都会有一个与其相应的助记符，助记符通常为指令功能的英语单词的简写。例如，mov 和 add 分别是数据的存储（move）和相加（addition）的简写。汇编语言和机器语言基本上是一一对应的。这一点和 C 语言、Java 语言等高级编程语言[③]有很大不同，这也是我们使用汇编语言来说明 CPU 运行的原因。通常我们将汇编语言编写的程序转化成机器语言的过程称为**汇编**；反之，机器

① 把汇编语言转化成机器语言的程序称为汇编器（assembler）。有时汇编语言也称为汇编。详情可参阅第 10 章。

② 机器语言是指 CPU 能直接解释和执行的语言。

③ 高级编程语言是指能够使用类似于人类语言（主要是英语）的语法来记述的编程语言的总称。BASIC、C、C++、Java、Pascal、FORTRAN、COBOL 等语言都是高级编程语言。使用高级编程语言编写的程序，经过编译转换成机器语言后才能运行。与高级编程语言相对，机器语言和汇编语言称为低级编程语言。

语言程序转化成汇编语言程序的过程则称为**反汇编**。

代码清单 1-1　汇编语言编写的程序示例

```
mov  eax, dword ptr [ebp-8]    …把数值从内存复制到 eax
add  eax, dword ptr [ebp-0Ch]  …exa 的数值和内存的数值相加
mov  dword ptr [ebp-4], eax    …把 exa 的数值（上一步的相加结果）存储在内存中
```

通过阅读汇编语言编写的代码，能够了解转化成机器语言的程序的运行情况。从代码清单 1-1 的汇编语言程序示例中也可以看出，机器语言级别的程序是通过寄存器来处理的。也就是说，在程序员看来“CPU 是寄存器的集合体”。至于控制器、运算器和时钟，程序员只需要知道 CPU 中还有这几部分就足够了。

代码清单 1-1 中，eax 和 ebp 表示的都是寄存器。通过阅读刚才的示例代码，想必大家对程序使用寄存器来实现数据的存储和加法运算这一情况应该有所了解了。汇编语言是 80386[①] 以上的 CPU 所使用的语言。eax 和 ebp 是 CPU 内部的寄存器的名称。内存的存储场所通过地址编号来区分，而寄存器的种类则通过名字来区分。

上文可能有些难以理解，不过不用担心，因为我们并不要求大家必须掌握 CPU 的所有寄存器种类和汇编语言，大家只需对 CPU 是怎么处理程序的有一个大致印象即可。也就是说，使用高级语言编写的程序会在编译[②] 后转化成机器语言，然后再通过 CPU 内部的寄存器来处理。例如，a=1+2 这样的高级语言的代码程序在转化成机器语言后，就是利用寄存器来进行相加运算和存储处理的。

① 80386 是美国英特尔公司开发的微处理器的产品名。“80386 以上”是指 80386、80486、奔腾等微处理器。

② 编译是指将使用高级编程语言编写的程序转换为机器语言的过程，其中，用于转换的程序被称为编译器（compiler）。

不同类型的 CPU，其内部寄存器的数量、种类以及寄存器存储的数值范围都是不同的。不过，根据功能的不同，我们可以将寄存器大致划分为八类，如表 1-1 所示。可以看出，寄存器中存储的内容既可以是指令也可以是数据。其中，数据分为“用于运算的数值”和“表示内存地址的数值”两种。数据种类不同，存储该数值的寄存器也不同。CPU 中每个寄存器的功能都是不同的。用于运算的数值放在累加寄存器中存储，表示内存地址的数值则放在基址寄存器和变址寄存器中存储。代码清单 1-1 的程序中用到的 eax 和 ebp 分别是累加寄存器和基址寄存器。

表 1-1　寄存器的主要种类和功能

种　类	功　能
累加寄存器 (accumulator register)	存储执行运算的数据和运算后的数据
标志寄存器 (flag register)	存储运算处理后的 CPU 的状态
程序计数器 (program counter)	存储下一条指令所在内存的地址
基址寄存器 (base register)	存储数据内存的起始地址
变址寄存器 (index register)	存储基址寄存器的相对地址
通用寄存器 (general purpose register)	存储任意数据
指令寄存器 (instruction register)	存储指令。CPU 内部使用，程序员无法通过程序对该寄存器进行读写操作
栈寄存器 (stack register)	存储栈区域的起始地址

对程序员来说，CPU 是什么呢？如图 1-3 所示，CPU 是具有各种功能的寄存器的集合体。其中，程序计数器、累加寄存器、标志寄存器、指令寄存器和栈寄存器都只有一个，其他的寄存器一般有多个。程序计数器和标志寄存器比较特殊，这一点在后面的章节中会详细说明。另外，存储指令的指令寄存器等寄存器，由于不需要程序员做多关注，因此图 1-3 中没有提到。

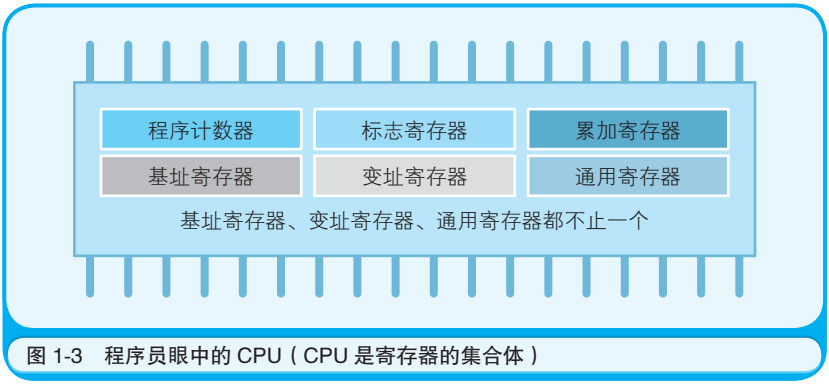

图 1-3 程序员眼中的 CPU（CPU 是寄存器的集合体）

1.3 决定程序流程的程序计数器

只有 1 行的有用程序是很少见的，机器语言的程序也是如此。在对 CPU 有了一个大体印象后，接下来我们看一下程序是如何按照流程来运行的。

图 1-4 是程序起动时内存内容的模型。用户发出启动程序的指示后，Windows 等操作系统[①]会把硬盘中保存的程序复制到内存中。示例中的程序实现的是将 123 和 456 两个数值相加，并将结果输出到显示器上。正如前文所介绍的那样，存储指令和数据的内存，是通过地址来划分的。由于使用机器语言难以清晰地表明各地址存储的内容，因此这里我们对各地址的存储内容添加了注释。实际上，一个命令和数据通常被存储在多个地址上，但为了便于说明，图 1-4 中把指令、数据分配到了一个地址中。

地址 0100 是程序运行的开始位置。Windows 等操作系统把程序从硬盘复制到内存后，会将程序计数器（CPU 寄存器的一种）设定为

① 操作系统（operating system）是指管理和控制计算机硬件与软件资源的计算机程序。关于操作系统的功能，第 9 章有详细说明。

0100，然后程序便开始运行。CPU 每执行一个指令，程序计数器的值就会自动加 1。例如，CPU 执行 0100 地址的指令后，程序计数器的值就变成了 0101（当执行的指令占据多个内存地址时，增加与指令长度相应的数值）。然后，CPU 的控制器就会参照程序计数器的数值，从内存中读取命令并执行。也就是说，程序计数器决定着程序的流程。

图 1-4　内存中配置的程序示例（显示相加的结果）

1.4　条件分支和循环机制

程序的流程分为顺序执行、条件分支和循环三种。**顺序执行**是指按照地址内容的顺序执行指令。**条件分支**是指根据条件执行任意地址的指令。**循环**是指重复执行同一地址的指令。顺序执行的情况比较简单，每执行一个指令程序计数器的值就自动加 1。但若程序中存在条件分支和循环，机器语言的指令就可以将程序计数器的值设定为任意地址（不是 +1）。这样一来，程序便可以返回到上一个地址来重复执行同一个指令，或者跳转到任意地址。接下来，我们就以条件分支为例，来具体说明循环时程序计数器的数值设定机制也是一样的。

图 1-5 表示把内存中存储的数值（示例中是 123）的绝对值输出到显示器的程序的内存状态。程序运行的开始位置是 0100 地址。随着程序计数器数值的增加，当到达 0102 地址时，如果累加寄存器的值是正数，则执行跳转指令（jump 指令）跳转到 0104 地址。此时，由于累加寄存器的值是 123，为正数，因此 0103 地址的指令被跳过，程序的流程直接跳转到了 0104 地址。也就是说，“跳转到 0104 地址”这个指令间接执行了“将程序计数器设定成 0104 地址”这个操作。

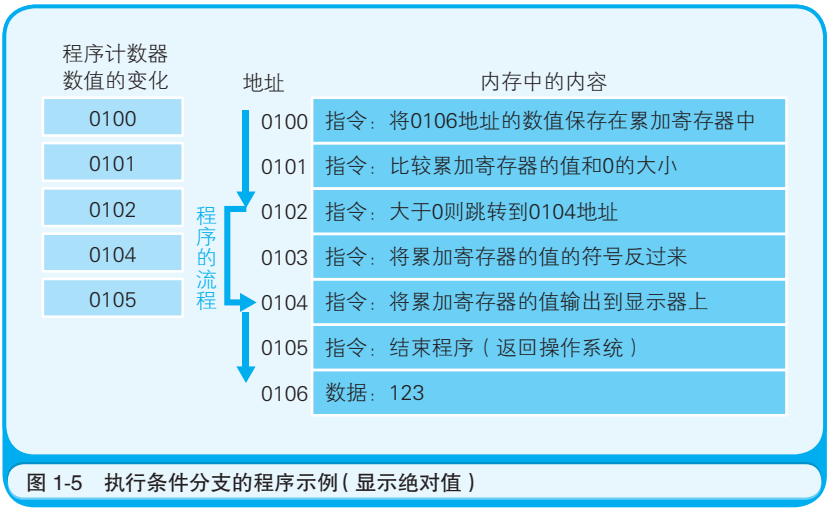

图 1-5 执行条件分支的程序示例（显示绝对值）

条件分支和循环中使用的**跳转指令**，会参照当前执行的运算结果来判断是否跳转。表 1-1 所列出的寄存器中，我们提到了标志寄存器。无论当前累加寄存器的运算结果是负数、零还是正数，**标志寄存器**都会将其保存（也负责存放溢出[①]和奇偶校验[②]的结果）。

① 溢出（overflow）是指运算的结果超出了寄存器的长度范围。

② 奇偶校验（parity check）是指检查运算结果的值是偶数还是奇数。

CPU 在进行运算时，标志寄存器的数值会根据运算结果自动设定。条件分支在跳转指令前会进行比较运算。至于是否执行跳转指令，则由 CPU 在参考标志寄存器的数值后进行判断。运算结果的正、零、负三种状态由标志寄存器的三个位[①]表示。图 1-6 是 32 位 CPU（寄存器的长度是 32 位）的标志寄存器的示例。标志寄存器的第一个字节位、第二个字节位和第三个字节位的值为 1 时，表示运算结果分别为正数、零和负数。

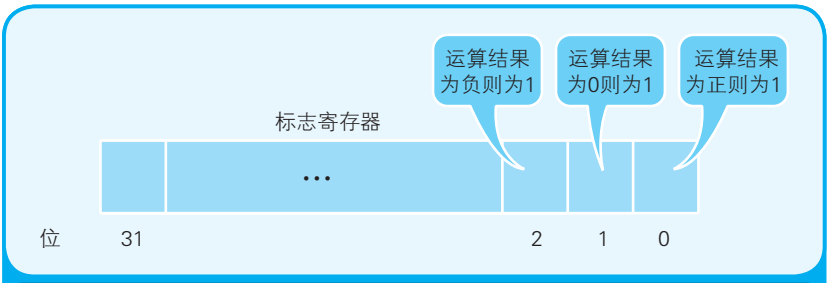

图 1-6　比较运算的结果存储在标志寄存器的三个位中

CPU 执行比较的机制很有意思，因此请大家务必牢记。例如，假设要比较累加寄存器中存储的 XXX 值和通用寄存器中存储的 YYY 值，执行比较的指令后，CPU 的运算装置就会在内部（暗中）进行 XXX－YYY 的减法运算。而无论减法运算的结果是正数、零还是负数，都会保存到标志寄存器中。结果为正表示 XXX 比 YYY 大，零表示 XXX 和 YYY 相等，负表示 XXX 比 YYY 小。程序中的比较指令，就是在 CPU 内部做减法运算。怎么样，是不是挺有意思的？

① 1 位（bit =binary digit）就是一个位数的二进制数，表示 0 或 1 的数值。32 位 CPU 指的就是用 32 位的二进制数来表示数据及地址的数值。关于二进制数的详细内容，请读者参阅第 2 章。

1.5 函数的调用机制

接下来，我们继续介绍程序的流程。哪怕是高级语言编写的程序，函数[①]调用处理也是通过把程序计数器的值设定成函数的存储地址来实现的。不过，这和条件分支、循环的机制有所不同，因为单纯的跳转指令无法实现函数的调用。函数的调用需要在完成函数内部的处理后，处理流程再返回到函数调用点（函数调用指令的下一个地址）。因此，如果只是跳转到函数的入口地址，处理流程就不知道应该返回至哪里了。

图 1-7 是给变量 *a* 和 *b* 分别代入 123 和 456 后，将其赋值给参数（parameter）来调用 MyFunc 函数的 C 语言程序。图中的地址是将 C 语言编译成机器语言后运行时的地址。由于 1 行 C 语言程序在编译后通常会变成多行的机器语言，所以图中的地址是离散的。

此外，通过跳转指令把程序计数器的值设定成 0260 也可实现调用 MyFunc 函数。函数的调用原点（0132 地址）和被调用函数（0260 地址）之间的数据传递，可以通过内存或寄存器来实现。不过，当函数处理进行到最后的 0354 地址时，我们知道应该将程序计数器的值设定成函数调用后要执行的 0154 地址，但实际上这一操作根本无法实现。那么，怎么办才好呢？

机器语言的 call 指令和 return 指令能够解决这个问题。建议大家把二者结合起来来记忆。函数调用使用的是 call 指令，而不是跳转指令。在将函数的入口地址设定到程序计数器之前，call 指令会把调用函数后

① 很多高级编程语言都采用类似于 $y=f(x)$ 这样的数学函数的语法来记述编写处理。我们知道，该数学函数的意思是将 x 这个值通过 f 处理后得到数值 y。如果套用函数的语法，x 就是参数，y 就是返回值，执行函数的功能就是函数调用。

要执行的指令地址存储在名为栈[①]的主存内。函数处理完毕后，再通过函数的出口来执行 return 命令。return 命令的功能是把保存在栈中的地址设定到程序计数器中。如图 1-7 所示，MyFunc 函数被调用之前，0154 地址保存在栈中。MyFunc 函数的处理完毕后，栈中的 0154 地址就会被读取出来，然后再被设定到程序计数器中（图 1-8）。

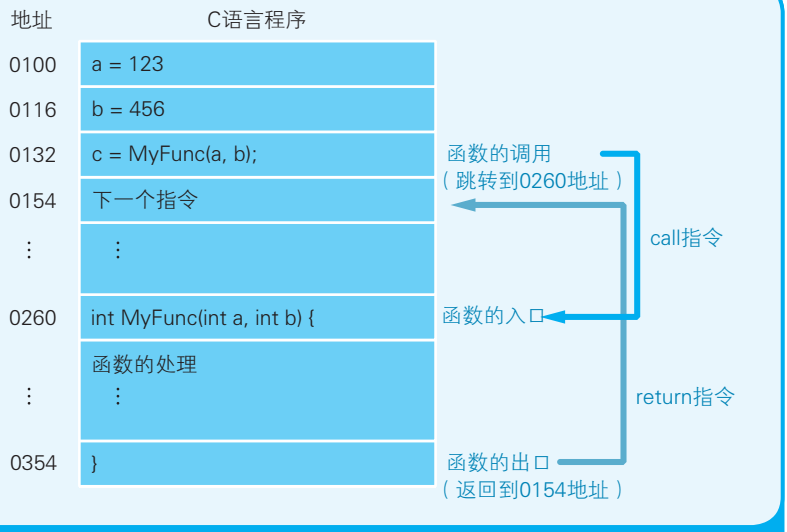

图 1-7　程序调用函数示例（这里直接展示了 C 语言的源代码，实际上各地址存储的应该是变换成机器语言后的程序）

① 栈（stack）本来是“干草等堆积如山”的意思。在程序领域中，通常使用该词来表示不断地存储各种数据的内存区域。函数调用后之所以能正确地返回调用前的地址，就是栈的功劳。关于栈，我们会在第 4 章进行详细说明。

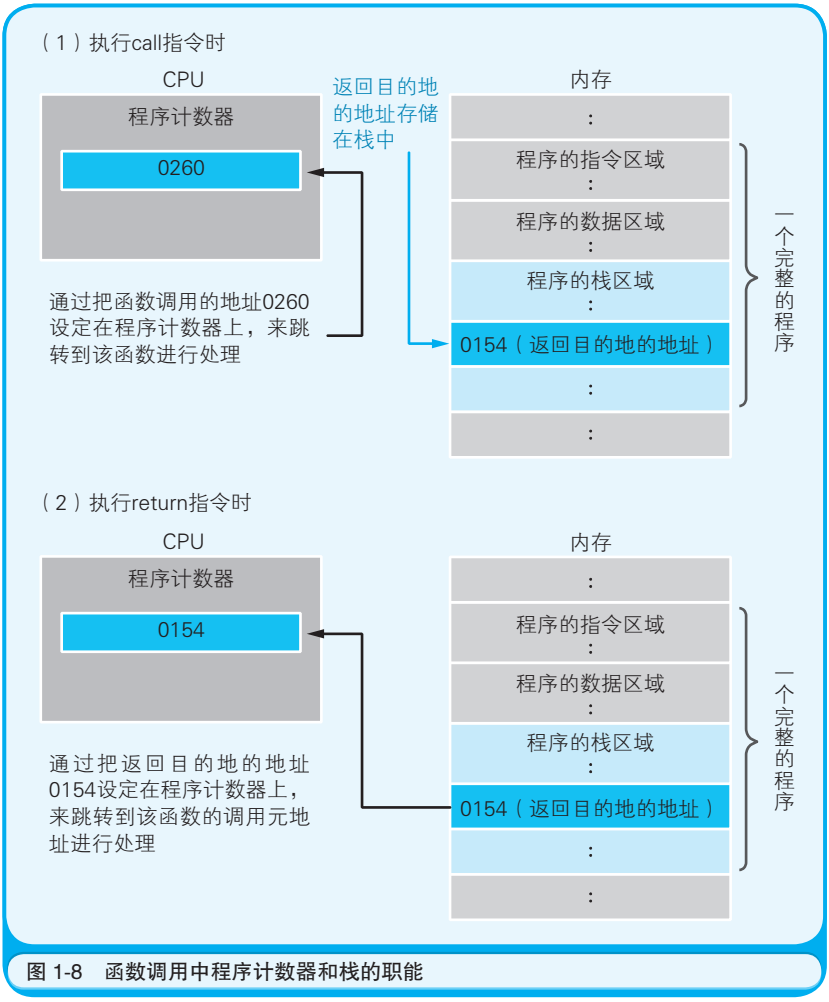

图 1-8　函数调用中程序计数器和栈的职能

在编译高级编程语言的程序后，函数调用的处理会转换成 call 指令，函数结束的处理则会转换成 return 指令。这样一来，程序的运行也就变得非常流畅。