将是 0021h，不是 0020h，而事实上 0020h 才是下一条指令的存储地址。

要是保留 000Ch 地址处的 Halt 指令，我们能寻求到更好的解决办法吗？

不过，我们可以用一个称为 Jump（跳转）的新指令来替换 Halt 指令。现在把它加入到指令表。

| 操作码 | 代码 |
|---|---|
| Load | 10h |
| Store | 11h |
| Add | 20h |
| Subtract | 21h |
| Add with Carry | 22h |
| Subtract with Borrow | 23h |
| Jump（跳转） | 30h |
| Halt | FFh |

通常情况下自动加法器是以顺序方式对 RAM 阵列寻址的。Jump 指令改变了机器的这种寻址方式，取而代之的是从某个指定的地址开始寻址。这种指令有时也被称作分支（branch）指令或者 Goto 指令，即“转到另一个位置”。

在上面的例子中，我们可以用一个 Jump 指令来替换 000Ch 地址处的 Halt 指令。

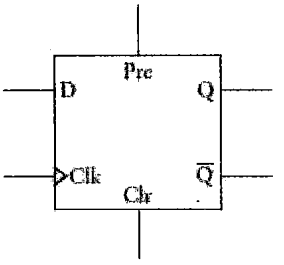
| | | |
|---|---|---|
| 000Ch: | 30h | 跳转到位于0020h地址处的指令 |
| | 00h | |
| | 20h | |

30h 即 Jump 指令的代码。其后的两个字节中存放的 16 位地址就是自动加法器要执行的下一条指令的地址。

因此在上面的例子中，自动加法器仍然从 0000h 地址开始，依次执行一条 Load 指令，一条 Add 指令，一条 Subtract 指令和一条 Store 指令。之后执行一条 Jump 指令，跳转至地址 0020h 继续依次执行一条 Load 指令，两条 Add 指令，一条 Store 指令，最后执行一条 Halt 指令。

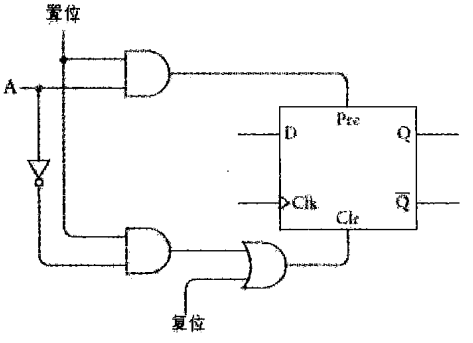
Jump 指令通过作用于 16 位计数器实现其功能。无论何时，只要自动加法器遇到 Jump

指令，计数器就会被强制输出该 Jump 指令后的 16 位地址。这可以通过 16 位计数器的 D 型边沿触发器的预置（Pre）和清零（Clr）输入来实现：




这里要再次声明，在正常的操作下，Pre 和 Clr 端的输入都应该是 0。但是，当 Pre=1，Q=1；当 Clr=1，则 Q=0。

如果你希望向一个触发器加载一个新的值（用 A 表示，代表地址），可以像下图所示这样连接。

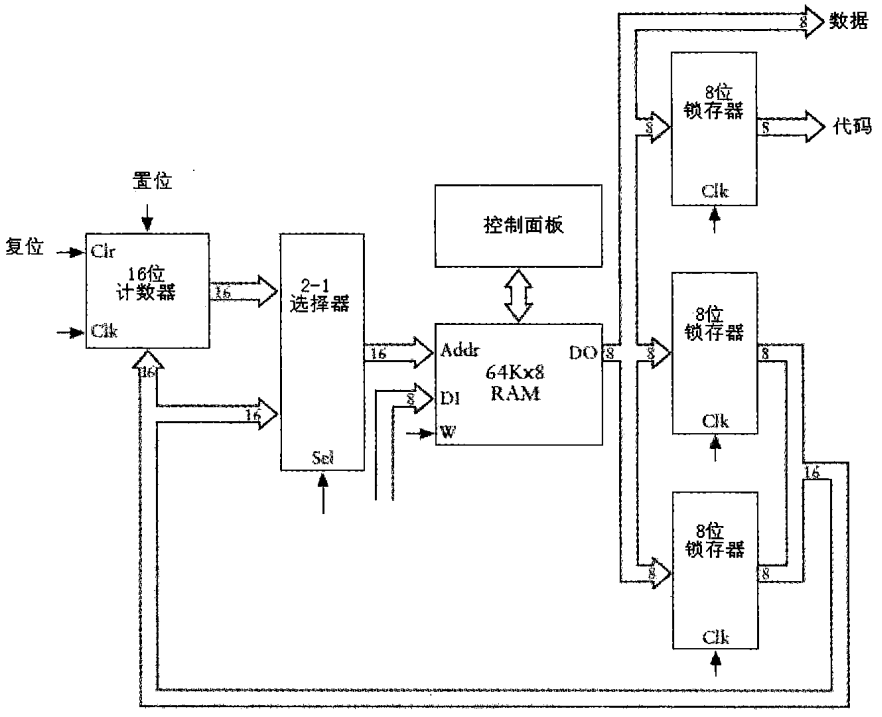



通常，置位信号为 0。此时，触发器的预置端输入为 0，在复位信号不为 1 的情况下，清零信号也为 0。在这种情况下，触发器就可以独立清零，而不受置位信号的影响。当置位信号为 1 时，如果 A 为 1，则清零输入为 0，预置输入为 1；如果 A 为 0，则预置输入 0，清零输入为 1。这就意味着 Q 端将被设置为与 A 端相同的值。

我们需要为 16 位计数器的每一位设置一个这样的触发器。一旦加载了某个特定的值，计数器就会从该值开始计数。

然而，这对电路的改动并不是很大。从 RAM 阵列锁存得到的 16 位地址既可以作为 2-1 选择器（它允许该地址作为 RAM 阵列的地址输入）的输入，也可以作为 16 位计数器

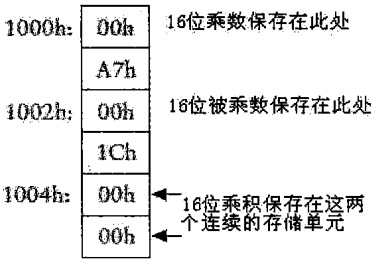
置位信号的输入。



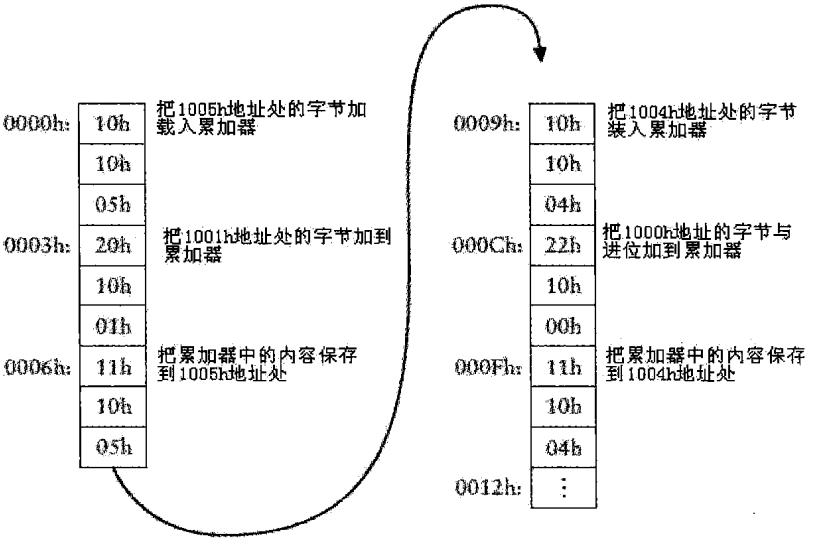

显然，只有当指令代码为 30h 并且其后的 16 位地址被锁存时，我们才必须确保置位信号为 1。

毋庸置疑，Jump 指令的确很有用。但与之相比，一个在我们想要的情况下跳转的指令更加有用，这种指令称做条件跳转（Conditional Jump）。也许说明该命令重要性的最好方法是这样一个问题：怎样让自动加法器进行两个 8 位数的乘法运算？例如，我们如何利用自动加法器得到像 A7h 与 1Ch 相乘这种简单运算的结果呢？

这其实很简单。两个 8 位数相乘得到一个 16 位数，为了方便起见，把该乘法运算中涉及的 3 个数均表示为 16 位数。第一步确定要把乘数和乘积保存到什么位置。



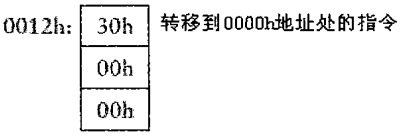

大家都知道 A7h 和 1Ch 相乘的结果（即十进制的 28）和把 28 个 A7h 累加的结果相同。因此保存在地址 1004h 和 1005h 的字节实际上是累加的结果。下图演示了如何把 A7h 加到该地址。




当这 6 条指令执行完毕之后，存储器 1004h 和 1005h 地址保存的 16 位数与 A7h 乘以 1 的结果相同。因此，为了使存放于该地址的值等于 A7h 与 1Ch 相乘的结果，要把这 6 条指令再反复执行 27 次。为了达到这个目的，可以在 0012h 地址开始把这 6 条指令连续输入 27 次；也可以在 0012h 处保存一个 Halt 指令，然后将复位键连续按 28 次得到最终结果。

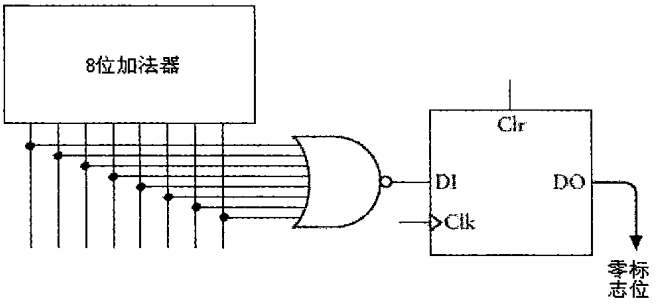
当然，这两种方式都不是很理想。它们都要求你做重复做许多遍烦琐的事情：输入一系列指令或者反复按复位键，而重复的次数就等于乘数。你当然是不会愿意用这种方式来进行 16 位数乘法运算的。

但如果在地址 0012h 处放置一条 Jump 指令会怎样呢？这个指令使得计数器再次从 0000h 处开始计数。




这的确是一个巧妙的方法。第一次执行完指令之后，位于存储器的 1004h 和 1005h 地址的 16 位数等于 A7h 乘 1，然后 Jump 指令使下一条指令从存储器顶部开始执行。第二次执行指令后，该 16 位数等于 A7h 乘 2，最后其结果可以等于 A7h 乘 1Ch。但是，这个过程不会停止下来，它会一直反复执行下去。

我们需要的是这样一种 Jump 指令，它只让这个过程重复执行所需要的次数，这种指令就是条件跳转指令，它并不难实现。要实现它，要做的第一步是增加一个与进位锁存器类似的 1 位锁存器。该锁存器被称为零锁存器（Zero latch），这是因为只有当 8 位加法器的输出全部为 0 时，它锁存的值才是 1。



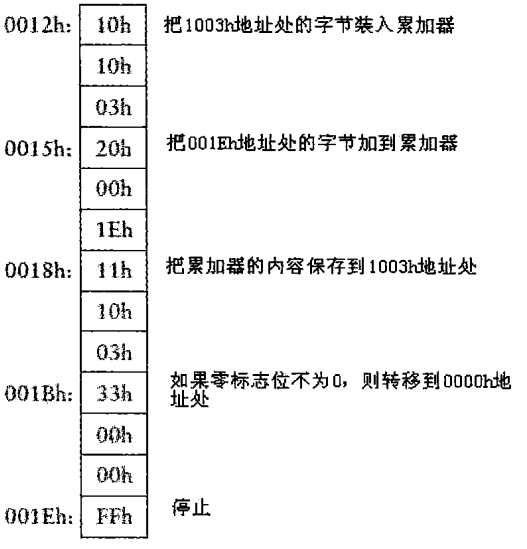

使或非门的输出为 1 的唯一方法是其所有的输入全为 0。与进位锁存器的时钟输入一样，只有当 Add、Subtract、Add with Carry、Subtract with Borrow 这些指令执行时，零锁存器才锁存 1 个数，该数称做零标志位（Zero flag）。注意，它是以一种似乎是相反的方式工作的：当加法器的输出全为 0 时，零标志位等于 1；当加法器的输出不全为 0 时，零标志位等于 0。

有了进位锁存器和零锁存器以后，我们可以为指令表新增 4 条指令。

| 操作码 | 代码 |
|---|---|
| Load | 10h |
| Store | 11h |
| Add | 20h |
| Subtract | 21h |
| Add with Carry | 22h |
| Subtract with Borrow | 23h |
| Jump | 30h |
| Jump If Zero（零转移） | 31h |
| Jump If Carry（进位转移） | 32h |
| Jump If Not Zero（非零转移） | 33h |
| Jump If Not Carry（无进位转移） | 34h |
| Halt | FFh |

例如，非零转移指令（Jump If Not Zero）只有在零锁存器的输出为 0 时才会跳转到指定的地址。换言之，如果上一步的加法、减法、进位加法、或者借位减法等运算的结果为 0 时，将不会发生跳转。为了实现这个设计，只需要在常规跳转命令的控制信号之上再加一个控制信号：如果指令是 Jump If Not Zero，那么只有当零标志位是 0 时，16 位计数器才被触发。

下图中 0012h 地址之后的指令即两个数相乘所用到的上表中的所有指令。



第一次循环之后，位于地址 0004h 和 0005h 处的 16 位数等于 A7h 与 1 的乘积，这和我们的设计相符。在上图中，地址 1003h 处的字节通过 Load 指令载入到累加器，该字节是 1Ch。把这个数和 001Eh 地址的字节相加后，恰好遇到的是 Halt 指令，当然这是一个合法的数。FFh 与 1Ch 相加的结果与从 1Ch 中减去 1 的结果相同，都是 1Bh，因为这个数不等于 0，所以零标志位是 0，1Bh 这个结果会存回到 1003h 地址。下一条要执行的指令是 Jump If Not Zero，零标志位没有置为 1，因此发生跳转。接下来要执行的一条指令位于 0000h 地址。

需要记住的是，Store 指令不会影响零标志位的值。只有 Add、Subtract、Add with Carry、Subtract with Borrow 这些指令才能影响零标志位的值，因此它的值与最近执行上述某个指令时所设置的值相同。

经过两次循环后，1004h 和 1005h 地址所保存的 16 位数等于 A7h 与 2 的乘积。1Bh 与 FFh 的和等于 1Ah，不为 0，因此仍然返回到顶部执行。

当执行到第 28 次循环时，1004h 和 1005h 地址保存的 16 位数等于 A7h 和 1Ch 的乘积。1003h 地址保存的值是 1，它和 FFh 相加的结果是 0，因此零标志位被置位！Jump If Not Zero 指令不会再跳转到 0000h 地址，相反，下一条要执行的指令即 Halt 指令。这样，我们就完成了全部的工作。

现在可以断言，我们一直不断完善的这组硬件构成的机器确实可以被称为计算机（computer）。当然，它还很原始，但毕竟是一台真正的计算机。条件跳转指令将它与我们以往设计的加法器区别开来，能否控制重复操作或者循环（looping）是计算机（computer）和计算器（calculator）的区别。这里演示了该机器如何用条件跳转实现两个数的乘法运算，用类似的方法还可以进行两个数的除法运算。而且，这不仅仅局限于 8 位数，它可以对 16 位、24 位、32 位，甚至更高位的数进行加、减、乘、除运算。而且，既然它能完成这些运算，那么对于开平方根、取对数、三角函数等运算也完全可以胜任。

既然我们已经装配了一台计算机，因此可以使用计算机相关词汇了。

我们装配的计算机属于数字计算机（digital computer），因为它只处理离散数据。曾经还有一种模拟信号计算机（analog computer），但现在已经非常少见了。（数字数据就是离散数据，即这些数据是一些确定的离散值；模拟数据是连续的，并且在整个取值区间

变化。）

一台数字计算机主要由 4 部分构成：处理器（processor）、存储器（memory），至少一个输入（input）设备和一个输出（output）设备。我们装配的计算机中，存储器是 64 KB 的 RAM 阵列，输入和输出设备分别是 RAM 阵列控制面板上的开关和灯泡。这些开关和灯泡可以让我们向存储器中输入数据，并可以检查运算结果。

除了上述 3 种设备之外的其他设备都归类于处理器。处理器也被称作中央处理单元（central processing unit）或者 CPU。更通俗的说法是将其称作计算机的大脑，但本文将避免使用这样的词，因为我们所设计的处理器称不上是大脑（今天，微处理器这个词使用得非常普遍，它是一种非常小的处理器，可以通过本书第 18 章讲到的技术来构造它。但本章中通过继电器构造的机器无论如何也称不上“微小”的）。

我们设计的处理器为 8 位处理器。累加器的宽度为 8 位，而且大部分数据通路都是 8 位的宽度。唯一的 16 位数据通路是 RAM 阵列的地址通路。如果该通路也采用 8 位宽度的话，存储器容量最多就只有 256 字节，而不再是 65536 字节，这样处理器的功能会受到很大的限制。

处理器包括若干组件。毫无疑问累加器就是其中一个，它只是一个简单锁存器，用来保存处理器内部的部分数据。在我们所设计的计算机中，8 位反相器和 8 位加法器一起构成了算术逻辑单元（Arithmetic Logic Unit），即 ALU。该 ALU 只能进行算术运算，最主要的是加法和减法运算。在更加复杂的计算机中（我们会在后面的章节看到），ALU 还可以进行逻辑运算，比如“与”（AND），“或”（OR），“异或”（XOR）等。16 位的计数器被称做程序计数器（PC，Program Counter）。

我们的计算机是由继电器、电线、开关，以及灯泡构造而成的，这些东西都叫做硬件（hardware）。与之对应，输入到存储器中的指令和数值被称做软件（software）。之所以把“硬”改成了“软”，是因为相对于硬件而言，指令和数据更容易修改。

当我们在计算机领域进行讨论时，“软件”这个词几乎与“计算机程序”（computer program），或“程序”（program）等术语是同义的，编写软件也称为计算机程序设计（computer programming）。我们确定用一些指令让计算机实现两个数相乘的过程就是在进行计算机程序设计。

通常，在计算机程序中，我们能够把代码（即指令本身）和数据（即代码要处理的数）区别开。但有时它们之间的界限也不是很明显，比如 Halt 指令（FFh）就可以有两种功能，除了作为代码时表示停止执行外还能代表数值–1。

计算机程序设计有时也被称做编写代码（writing code），或编码（coding），也许你经常会听到："我整个假期都在编码"，"我一直干到今天早上，敲出了很多行代码"。计算机程序设计人员有时也被称做编码员（coders），尽管有些人可能认为这是一个贬义词。程序员更喜欢被别人称做"软件工程师"（software engineers）。

能够被处理器响应的操作码（比如 Load 指令和 Store 指令的代码 10h 和 11h），称做机器码（machine codes），或机器语言（machine language）。计算机能够理解和响应机器码，其原理和人类能够读写语言是类似的，因此这里使用了"语言"来描述它。

一直以来，我们都在使用很长的短语来引用机器所执行的指令，比如 Add with Carry 指令。通常而言，机器码都分配了对应的简短助记符，这些助记符都用大写字母表示，包括 2 个或 3 个字符。下面是一系列上述计算机大致能够识别的机器码的助记符。

| 操作码 | 代码 | 助记符 |
|---|---|---|
| Load（加载） | 10h | LOD |
| Store（保存） | 11h | STO |
| Add（加法） | 20h | ADD |
| Subtract（减法） | 21h | SUB |
| Add with Carry（进位加法） | 22h | ADC |
| Subtract with Borrow（借位减法） | 23h | SBB |
| Jump（转移） | 30h | JMP |
| Jump If Zero（零转移） | 31h | JZ |
| Jump If Carry（进位转移） | 32h | JC |
| Jump If Not Zero（非零转移） | 33h | NC |
| Jump If Not Carry（无进位转移） | 34h | JNC |
| Halt | FFh | HLT |

当这些助记符与另外一对短语结合使用时，其作用更加突出。例如，对于这样一条长语句"把 1003h 地址处的字节加载到累加器"，我们可以用如下简洁的句子替代：

```
LOD  A, [1003h]
```

位于助记符右侧的 A 和[1003h]称为参数（argument），它们是这个 Load 指令的操作对象。参数由两部分组成，左边的操作数称为目标（destination）操作数（A 代表累加器），右边的操作数称为源（source）操作数。方括号“[]”表明要加载到累加器的不是 1003h 这个数值，而是位于存储器地址 1003h 的数值。

类似的，指令“把 001Eh 地址的字节加到累加器”，可以简写为：

```
ADD A, [0010Eh]
```

指令“把累加器中的内容保存到 1003h 地址”，可简写为：

```
STO [1003h], A
```

注意，在上面的语句中，目标操作数（Store 指令在存储器中的位置）仍然在左边，源操作数在右边。这就决定了累加器中的内容必须要保存到存储器的 1003h 地址。“如果零标志位不是 1 则跳转到 0000h 地址处”这个冗长的语句可以简明地表示为：

```
JNZ  0000h
```

注意，这里没有使用方括号，这是因为跳转指令要转移到的地址是 0000h，而不是保存于 0000h 地址的值，即 0000h 地址就是跳转指令的操作数。

用缩写的形式表示指令是很方便的，因为在这种形式下指令以可读的方式顺序列出而不必画出存储器的空间分配情况。通过在一个十六进制地址后面加一个冒号，可以表示某个指令保存在某个特定地址空间，例如：

```
0000h:       LOD A, [1005h]
```

下面的语句表示了数据在特定地址空间的存储情况。

```
1000h:       00h, A7h
1002h:       00h, 1Ch
1004h:       00h, 00h
```

你可能已经注意到了，上面的两个字节都是以逗号分开的，它表示第一个字节保存在左侧的地址空间中，第二个字节保存在该地址后的下一个地址空间中。上面的三条语句等价于下面的这条语句：

```
1000h:       00h, A7h, 00h, 1Ch, 00h, 00h
```

因此上面讨论的乘法程序可以用如下一系列语句来表示：

```
0000h:      LOD A, [1005h]
            ADD A, [1001h]
            STO [1005h], A

            LOD A, [1004h]
            ADC A, [1000h]
            STO [1004h], A

            LOD A, [1003h]
            ADD A, [001Eh]
            STO [1003h], A

            JNC 0000h

001Eh:      HLT

1000h:      00h, A7h
1002h:      00h, 1Ch
1004h:      00h, 00h
```

使用空格和空行的目的仅仅是为了人们更方便地阅读程序。

在编码时最好不要使用实际的数字地址，因为它们是可变的。例如，如果要把数值保存在存储器的 2000h～2005h 地址空间中，你需要在程序中重复多次写这些语句。用标号（label）来指代存储器中的地址空间是个较好的办法。这些标号是一些简单的单词，或是类似单词的字符串。上面的代码可以改写为：

```
BEGIN:      LOD A, [RESULT + 1]
            ADD A, [NUM1 + 1]
            STO [RESULT + 1], A

            LOD A, [RESULT]
            ADC A, [NUM1]
            STO [RESULT], A

            LOD A, [NUM2 + 1]
            ADD A, [NEG1]
            STO [NUM2 + 1], A

            JNZ BEGIN
```

```
NEG1:       HLT

NUM1:       00h, A7h
NUM2:       00h, 1Ch
RESULT:     00h, 00h
```

注意，NUM1，NUM2，RESULT 这些标号都是指存储器中保存两个字节的地址单元。在这些语句中，NUM1+1，NUM2+1 和 RESULT+1 分别指标号 NUM1，NUM2，RESULT 后的第二个字节。注意，NEG1（negative one）用来标记 HLT 指令。

最后，如果你可能忘记这些语句所表示的意思，那么可以在该语句后面加注释（comment），这些注释可以用我们人类的自然语言表述，然后通过分号与程序语句分隔开。

```
BEGIN:      LOD A, [RESULT + 1]
            ADD A, [NUM1 + 1]       ; 低字节相加
            STO [RESULT + 1], A

            LOD A, [RESULT]
            ADC A, [NUM1]           ; 高字节相加
            STO [RESULT], A

            LOD A, [NUM2 + 1]
            ADD A, [NEG1]       ; 第二个数减 1
            STO [NUM2 + 1], A

            JNZ BEGIN

            NEG1:       HLT

            NUM1:       00h, A7h
            NUM2:       00h, 1Ch
            RESULT:     00h, 00h
```

这里给出的是一种计算机程序设计语言，称为汇编语言（assembly language）。它是全数字的机器语言和指令的文字描述的一种结合体。同时它用标号表示存储器地址。人们有时候会混淆机器语言和汇编语言，这是因为它们是对同一种事物的不同描述方式。每一条汇编语句都对应着机器语言中的某些特定字节。

如果你想为本章所设计的计算机编写程序，那么可能首先想到的是用汇编语言来编

写（在纸上）。在你确定程序无误并准备验证其运行结果的时候，你需要手工对其汇编：这就意味着要把每一条汇编语句转换成与之对应的机器语言，这仍然要在纸上操作。完成之后，你需要通过开关把这些机器码输入到 RAM 阵列中并运行该程序，也就是让计算机执行这些指令。

对于学习计算机程序设计的人来说，应该尽早了解“错误”（bug）这个术语。当你编写代码时——特别是采用机器语言——是非常容易出错的。输入一个错误的操作数已经很不妙了，如果输错的是一个指令代码的话，情况会怎样呢？当你准备输入 10h（Load 指令）的时候，却输入了 11h（Store 指令），造成的后果是：期望的数据不会被机器加载，而该地址的数据还会被累加器中的内容替换掉。

一些错误可能导致意想不到的结果。假设你使用 Jump 指令跳转到一个地址，而该地址没有存放任何合法的指令，或者你偶然误用 Store 指令覆盖了其他指令，类似的情况都有可能发生（而且会经常发生）。

甚至上述乘法程序中就存在着一个错误。如果你把程序执行两次，第二次得到的将会是 A7h 与 256 相乘的结果，并且程序会把这个结果与第一次运算的结果相加。这是因为程序执行一次之后，1003h 地址保存的数值是 0。当第二次执行时，FFh 与这个 0 相加的结果不是 0，因此程序会继续执行直到它变为 0。

我们已经利用该机器完成了乘法运算，用类似的方法也可以进行除法运算。同时，前面也讲过，利用这些基本功能还可以进行平方根、对数、三角函数等运算。机器所需要的仅仅是能够做加、减法的硬件以及利用条件跳转指令执行代码的方法。正如程序员经常挂在嘴边的一句话：“我可以用软件完成其他工作”，这些工作我们都可以编程实现。

当然，软件可能是很复杂的。有很多专门讲授程序员如何用算法（algorithm）解决特殊问题的书，本书不打算讲这些内容。目前我们一直讨论的都是自然数，并没有考虑如何在计算机中表示十进制小数，本书将在第 23 章讨论这个问题。

前面不止一次强调过，这些硬件部件早在 100 年前就发明出来了。但是，本章所设计的计算机在当时却并没有被创造出来。当继电器计算机在 20 世纪 30 年代中期被设计出来的时候，很多包含在其中的概念还并不为人所知，直到 1945 年左右世人才开始慢慢了解它们。例如，在当时，人们仍然尝试在计算机中使用十进制数而不是二进制数；而

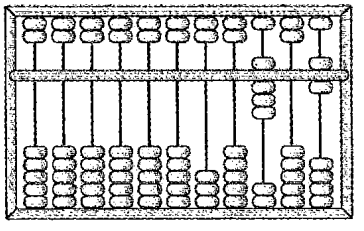
且计算机程序也不是完全存储在存储器中的，有时候它们会被保存在纸带上。特别是在早期的计算机中，存储器非常昂贵并且体积庞大，不论在 100 年前还是在今天，用 500 万个电报继电器构建一个 64 KB 的 RAM 阵列都是令人感到荒唐的事。

当回顾完计算器和计算机这一段历史，让我们展望一下未来。或许有一天我们会发现：没有必要建造一个如此复杂的继电器计算机。正像在第 12 章所讨论过的那样，继电器最终会被真空管和晶体管这类电子器件所替代。或许我们还会发现，已经有人创造出一种全新的设备，它的处理器及存储器的能力与我们所设计出的不相上下，但是，这种机器精致小巧，甚至可以放在我们的掌心。

# 18 从算盘到芯片

自古以来，人们为了尽量简化数学计算，绞尽脑汁发明了很多精巧的工具和机器。虽然人类的计数能力与生俱来，但需要帮助是在所难免的。每个人都是各有所长、各有所短，所以经常会遇到一些自身无法解决的问题。

在人类社会早期，人们借助数字这种工具来帮助自己记录物品和财产。包括古希腊以及美洲土著在内的很多文化中，人们借助小卵石或者谷粒进行计数。在欧洲，这些古老的计数方式演变成了计数板，而在中东，则演变成为我们较为熟悉的由骨架和算珠组成的算盘（abacus），算盘的样子就像下面这幅图所示。



尽管人们通常将算盘与亚洲文化，尤其是中国文化联系到一起，但它似乎是在公元 1200 年左右由商人带到中国的。

很多人不喜欢做乘法和除法，但仍然有一小部分人对此进行了一些研究。苏格兰数