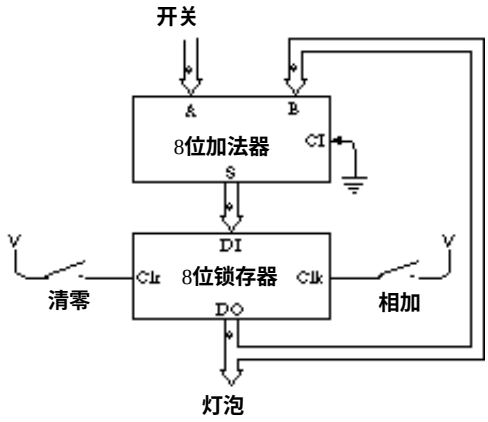
第17章 自动操作

人类是非常富于创造性而且是十分勤勉的，但是，人类在本质上也是十分懒惰的。非常明显，人类并不愿意去工作，这种对工作的反感导致人们用大量的时间来设计和制造可以把工作日缩短到几分钟的设备。幻想使人感到兴奋，甚至远比我们所看到新奇的事物更令人兴奋得多。

当然不会在这里介绍自动割草机的设计。本章将通过设计更精密的机器，使加减法运算更加自动化，这听起来也许有些不可思议。本章最后设计出的机器将具有广泛的用途，它实际上可以解决任何利用加减法的问题，这些问题的范围太大了。

当然，由于精密机器越来越复杂，因此有些部分可能会很粗糙。不过如果你略过了某些困难的细节，没有人会责备你。有时，你可能会不耐烦并且发誓再也不会因为一个数学问题而去寻求电或机械的帮助。不过请耐心坚持到底，因为本章最后将发明一个叫作计算机的机器。

我们曾在第14章见过一个加法器。它有一个8位锁存器，累加由8个开关输入的数的和：



前面曾讲过，8位锁存器用触发器来保存8位数据。使用这个设备时，必须首先按下清零开关使锁存器的存储内容清零，然后用开关来输入第一个数字。加法器简单地把这个数字与锁存器输出的零相加，因此其结果就是你刚输入的数字。按下相加开关可在锁存器中保存该数并且通过灯泡显示出来。现在从开关上输入第二个数，加法器把这个数与存储在锁存器中的数相加，再按下相加开关把总和存储在锁存器中并通过灯泡显示出来。通过这种方法，你可以加上一串数字并显示出运算总和。当然，其中存在的一个局限是 8个灯泡不能显示总和超过255的数。

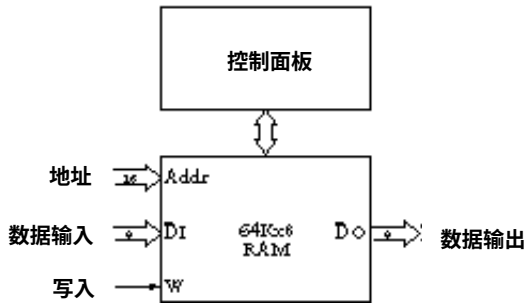
第14章介绍该电路的时候，只讲到一种锁存器，它是电平触发的。在电平触发的锁存器中，时钟输入端必须先置1然后回到0，才能使锁存器保存数据。当时钟信号等于 1时，锁存器的数据输入可以改变，这种改变将会影响所保存的数据输出。第 14章的后面又介绍了边沿触发的锁存器，这种锁存器在时钟输入从 0变化到1的瞬间保存数据。由于边沿触发的锁存器易

于使用，所以假定本章用到的锁存器为边沿触发的锁存器。

用于累加数字的锁存器叫作累加器，本章后面将会看到累加器并非仅仅进行简单的累加。累加器通常是一个锁存器，保存第一个数字，然后该数字又加上或减去另一个数字。

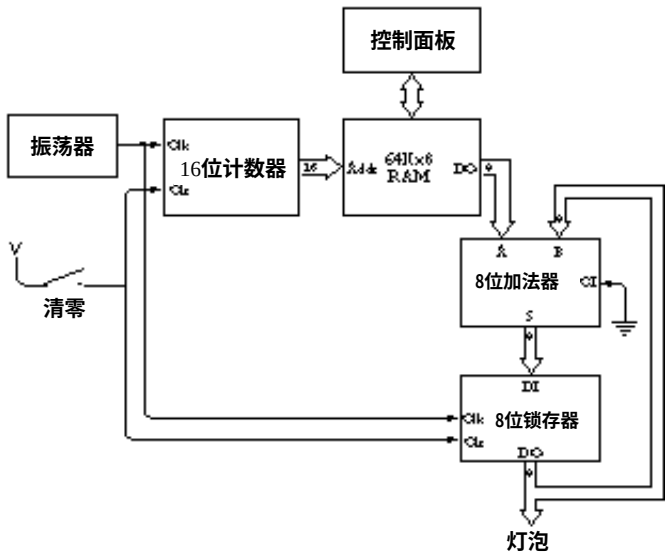
上面这个加法机存在的最大问题已经相当明显：如果想把 100个二进制数加起来，你就得坐在加法机前耐着性子输入每一个数字并累加起来。当你完成时，却发现有两个数字是错误的，你只好又重复全部的工作。

不过，也可能并非如此。上一章用了差不多 500万个继电器来构造一个64KB的RAM阵列。另外，我们还连接了一个控制面板，用来闭合接管开关接通线路，并使用开关进行 RAM阵列的写入和读出。



如果你向RAM阵列中输入100个二进制数字，而不是直接输入到加法机中，那么进行数据修改会容易得多。

现在我们面临着一个挑战，即如何将RAM阵列连到累加器上。显而易见，RAM的数据输出信号应该代替累加器的开关组。但是，用一个 16位的计数器（正如在第14章构造的）就可以控制RAM阵列的地址信号。在下面这个电路中，连到RAM 的数据输入信号和写入信号可以不要：



当然这并非已经发明的最容易操作的计算装置。在使用之前，必须先闭合清零开关，以清除锁存器的内容并把 16位计数器的输出置为 0000h，接着闭合RAM控制面板上的接管开关。你可以从RAM地址的0000h处开始输入一组想要加的 8位数，如果有 100个数，则它们保存在从0000h～0063h的地址中（也可以把 RAM阵列中没有用到的单元都设置为 00h）。然后断开

RAM控制面板上的接管开关（这样控制面板不会再对 RAM阵列起控制作用了），并断开清零开关。这时，你就只需坐着看灯泡的亮灭变化了。

其工作情况为：当清零开关第一次断开时， RAM阵列的地址输入为0000h，保存在RAM阵列当前地址的8位数是加法器的输入。由于锁存器也清零，所以加法器的另 8位输入为00h。

振荡器提供时钟信号 —— 一个在0和1之间迅速交替变化的信号。在清零开关断开后，当时钟由0变为1时，将同时发生两个事件：锁存器保存来自加法器的结果；同时， 16位计数器加1，指向RAM阵列的下一个地址。在清零开关断开后，当时钟第一次由 0变为1时，锁存器保存第一个数，同时，计数器增加到 0001h；当时钟第二次由0变为1时，锁存器保存第一个数与第二个数之和，同时计数器增加到0002h；依此类推。

当然，这里先做了一些假设，首要一点，振荡器需慢到允许电路的其余部分可以工作。对于每次时钟振荡，在加法器输出端显示有效和之前，许多继电器必须触发其他继电器。

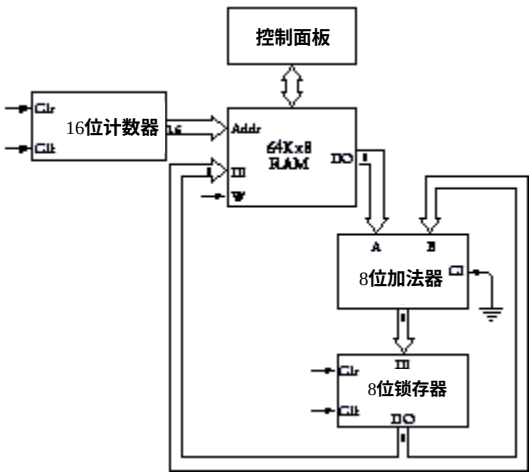
这种电路有一个问题，即没有办法让它停止。到一定时候，灯泡会停止闪动，因为 RAM阵列中的其余数都为00h。这时，你可以看到二进制和。但当计数器最终到达 FFFFh时，它又会翻到0000h（就像汽车里程表），这时自动加法器又会开始把这些数加到已经计算过的和中。

这种加法机还有一个问题：它只能用于加法，并且只能加 8位数。不仅在RAM阵列中的每个数不能超过255，而且其总和也不能超过 255。这种加法器也没有办法进行减法运算。虽然可以用2的补码表示负数，但是在这种情况下，加法器只能处理－ 128～127之间的数字。让它处理更大数字（例如， 16位数）的一种显而易见的方法就是使 RAM阵列、加法器和锁存器的宽度加倍，同时再提供8个灯泡。不过你可能不太愿意做这种投资。

当然，要不是我们最终要去解决这些问题，这儿是不会提到这些问题的。不过我们首先想谈的却是另外一个问题。如果不是要把 100个数加成一个数，会怎么样？如果只想用自动加法器把50对数字加成50个不同的结果又会怎么样？也许你希望有一个万能的机器来累加多对数字、10个数字或100个数字，并且希望所有的结果都可方便地使用。

前面提到的自动加法器在与锁存器相连接的一组灯泡上显示出其相加结果。对于把 50对数字加成50个不同的和来说，这种方法并不好。你可能希望把结果存回 RAM阵列中，然后，在方便的时候用RAM控制面板来检查结果。控制面板上有专门为此目的而设计的灯泡。

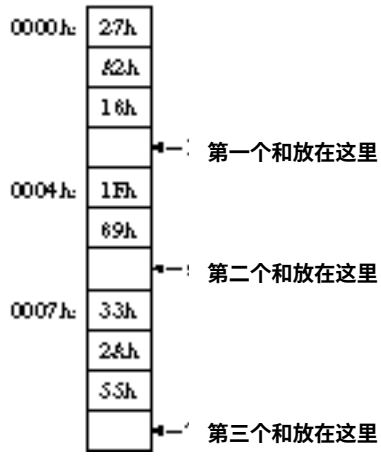
这意味着连接在锁存器上的灯泡可以去掉。不过，锁存器的输出端必须连接到 RAM阵列的数据输入端上，以便于和可以写入到RAM中：



上图中省略了自动加法器的其余部分，特别是振荡器和清零开关，因为不再需要显著标出计数器和锁存器的清零和时钟输入来源。此外，既然我们已经充分利用了 RAM的数据输入端，就需要有一种方法来控制RAM的写入信号。

我们不去考虑这个电路能否工作，而把重点放在需要解决的问题上。当前需要解决的问题是能配置一个自动加法器，它不会仅用来累加一串数字。我们希望能随心所欲地确定累加多少数字、在RAM中存储多少不同的结果以供日后检查。

例如，假设我们希望先把三个数字加在一起，然后把另两个数字加在一起，最后再把另外三个数加在一起。我们可能会将这些数字存储在从地址 0000h开始的RAM阵列中，存储器的内容如下所示：



这是本书第16章所说明的内容。方格里是存储单元中的内容，存储器的每一个字节在一个方格中。方格的地址在方格左面，并非每一个地址都要表示出来，存储器的地址是连续的，因而可以算出某个方格的地址。方格的右侧是关于这个存储单元的注释，它们表示出我们希望自动加法器在这些空格中存储三个结果。（虽然这些方格是空的，但存储单元并非空的。存储单元中总有一些东西，即使只是随机数，但此时它不是有用的数。）

现在可以试一下十六进制算术运算并且把结果存到方格中，但这并不是此项试验的要点，我们想让自动加法器来做一些额外的工作。

不是让自动加法器只做一件事情 —— 在最初的加法器中，只是把 RAM地址中的内容加到称为累加器的8位锁存器中 —— 实际上是让它做四件不同的事。要做加法，需先从存储器中传送一个字节到累加器中，这个操作叫作 Load（装载）。第二项所要执行的操作是把存储器中的一个字节加(Add)到累加器中。第三项是从累加器中取出结果，保存(Store)到存储器中。最后，需要有一些方法使自动加法器停止(Halt)工作。

详细说来，让自动加法器所做的工作如下所示：

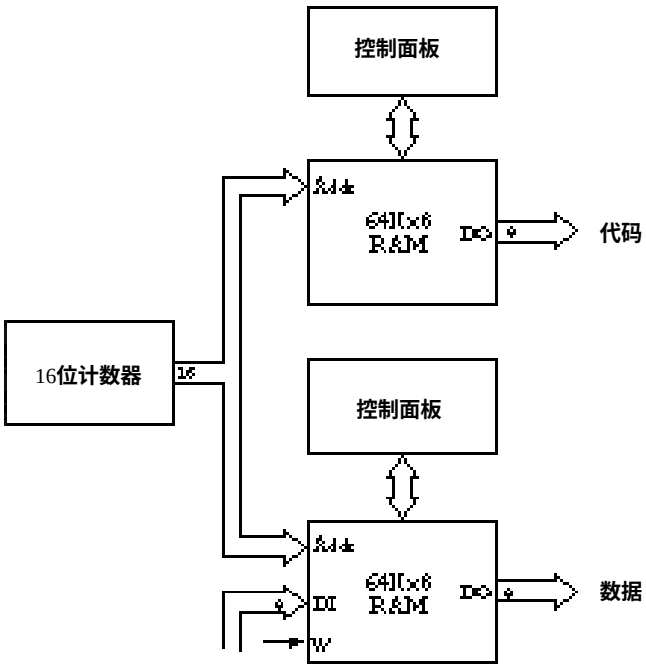
- 把地址0000h中的数装载到累加器中
- 把地址0001h中的数加到累加器中
- 把地址0002h中的数加到累加器中
- 把累加器中的数保存到地址0003h中
- 把地址0004h中的数装载到累加器中

- 把地址0005h中的数加到累加器中
- 把累加器中的数保存到地址0006h中
- 把地址0007h中的数装载到累加器中
- 把地址0008h中的数加到累加器中
- 把地址0009h中的数加到累加器中
- 把累加器中的数保存到地址000Ah中
- 停止自动加法器的工作

注意，同最初的自动加法器一样，存储器的每个字节的地址是连续的，开始处为 0000h。以前自动加法器只是简单地把存储器中相应地址的数加到累加器中。某些情况下，现在仍然需要这样做，但有时我们也想直接把存储器中的数装载到累加器中或者把累加器中的数保存到存储器中。在所有事情都完成以后，我们还想让自动加法器停下来以便检查 RAM阵列中的内容。

怎样完成这些工作呢？只是简单地键入一组数到 RAM中并期望自动加法器来正确操作是不可能的。对于RAM中的每个数字，我们还需要一个数字代码来表示自动加法器所要做的工作：装载，加，保存或停止。

也许最容易（但肯定不是最便宜）的方法是把这些代码存储在一个完全独立的 RAM阵列中。这第二个RAM 阵列与最初的RAM阵列同时被访问，但它存放的不是要加的数，而是用来表明自动加法器将要对最初的 RAM阵列的相应地址进行某种操作的代码。这两个 RAM阵列可以分别标为数据（最初的RAM阵列）和代码（新的RAM阵列）：



已经确认新的自动加法器能够把“ 和 ”写入到最初的 RAM阵列（标为数据），而要写入新的RAM阵列（标为代码）则只能通过控制面板来进行。

我们用4个代码来表示自动加法器希望能实现的 4个操作。4个代码可任意指定，下面为可能的一组代码：

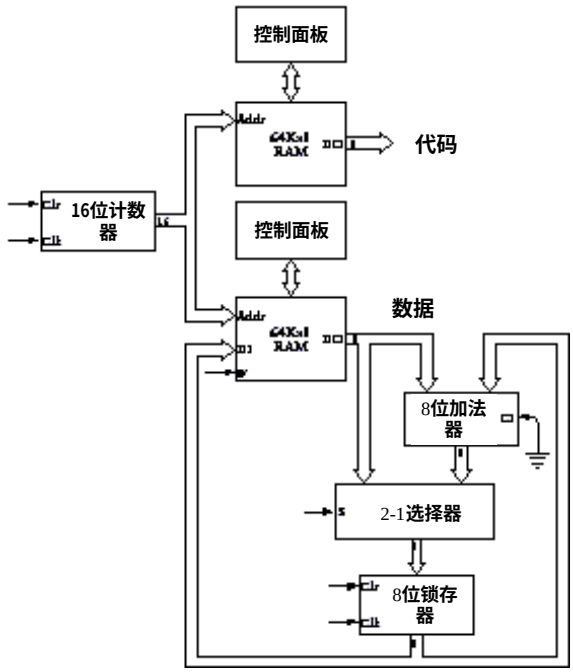
操作码	代码
Load（装载）	10h
Store (保存)	11h
Add (加)	20h
Halt (停止)	FFh

为了执行以上例子中提到的三组加法，需要用控制面板把下面这些数保存到代码RAM阵列中：

0000h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
0004h:	10h	Load
	20h	Add
	11h	Store
0007h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
000Bh:	FFh	Halt

你可能想把这个RAM阵列中的内容与存放累加数据的 RAM阵列的内容作一比较。你会发现代码RAM中的每个代码或者对应于数据 RAM中一个要装入或加到累加器的数，或者表示一个要存回到存储器中的数。这样的数字代码通常称作指令码或操作码，它们“指示”电路执行某种“操作”。

前面谈到过，早期自动加法器的 8位锁存器的输出需要作为数据 RAM阵列的输入，这就是“保存”指令的功能。另外还需要一个改变：以前，8位加法器的输出是作为锁存器的输入，但现在为了实现“装载”指令，数据 RAM的输出有时候也要作为 8位锁存器的输入，这种改变需要2-1数据选择器。改进的自动加法器如下图：



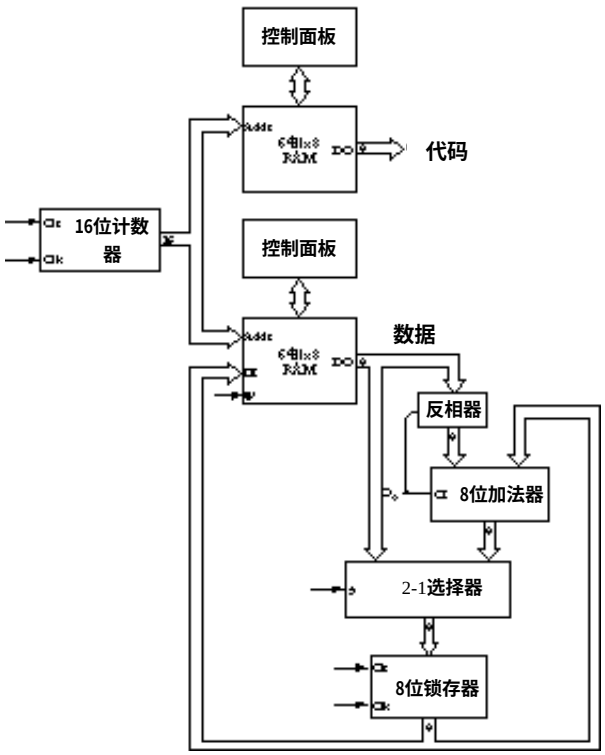
上图中少了一些东西，但它显示了各种组件间的 8位数据通路，一个 16位计数器为 2个RAM阵列提供地址。通常，数据 RAM 阵列输出到8位加法器上执行加法指令。 8位锁存器的输入可能是数据 RAM阵列的输出也可能是加法器的输出，这需要 2-1选择器来选择。通常，锁存器的输出又流回到加法器，但对“保存”指令而言，它又作为数据 RAM阵列的输入。

上图中缺少的是控制这些组件的信号，统称为控制信号。它们包括16位计数器的时钟（Clk）和清零(Clr)输入，8位锁存器的Clk和Clr输入，数据 RAM阵列的写入(W)输入以及2-1选择器的选择(S)输入。其中有一些信号明显基于代码 RAM 阵列的输出，例如，若代码 RAM 阵列的输出表示装载指令，则2-1选择器的S输入必须为0（选择数据RAM阵列的输出）。仅当操作码为保存指令时，数据RAM阵列的W输入才为1。这些控制信号可以由逻辑门的各种组合来产生。

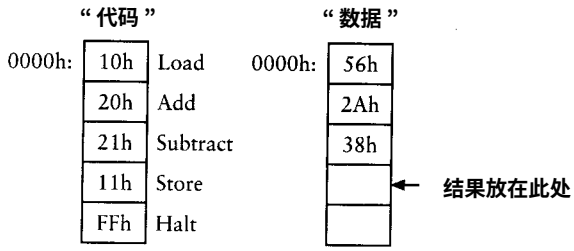
利用最小数量的附加硬件和新增的操作码，也能让这个电路从累加器中减去一个数。第 1步是扩充操作码表：

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract(减)	21h
Halt	FFh

加法和减法只通过操作码的最低有效位来区分。若操作码为 21h，除了在数据 RAM阵列的输出数据输入到加法器之前取反并且加法器的进位输入置 1外，电路所做的几乎与电路执行加法指令所做的完全相同。在下面这个改进的有一个反相器的自动加法器里， C_0信号可以完成这两项任务：



现在假设把56h和2Ah相加再减去38h，可以按下图所显示的存储在两个 RAM阵列中的操作码和数据进行计算：



装载操作完成后，累加器中的数为 56h。加法操作完成后，累加器中的数为 56h加上2Ah的和，即80h。减法操作使数据 RAM阵列的下一个数（38h）按位取反，变为C7h。加法器的进位输入置为1时，取反的数C7h与80h相加：

```
  C7h
+ 80h
+  1h
-----
  48h
```

其结果为48h。（按十进制，86加42减56等于72。）

还有一个未找到适当解决方法的问题是加法器及连到其上的所有部件的宽度只有 8位。以往唯一的解决方法就是连接两个8位加法器（或其他的两个部件），形成16位的设备。

但也有更便宜的解决方法。假设要加两个 16位数，例如：

```
 76ABh
+232Ch
------
```

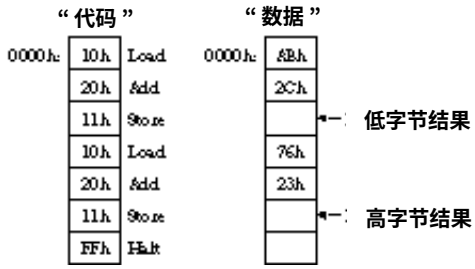
这种16位加法同单独加最右边的字节（通常称作低字节）：

```
  ABh
+ 2Ch
-----
  D7h
```

然后再加最左边的字节，即高字节

```
  76h
+ 23h
-----
  99h
```

得到的结果一样，为99D7h。因此，如果像这样把两个16位数保存在存储器中：



结果中的D7h存在地址0002h中，99h存在地址0005h中。

当然，并非所有的数都这样计算，对上例中的数是这样计算。若两个16位数76ABh和236Ch相加会怎么样呢？在这种情况下，2个低字节数相加的结果将产生一个进位：

```
  ABh
+ 6Ch
-----
 117h
```

这个进位必须加到2个高字节数的和中：

```
   1h
+ 76h
+ 23h
-----
 9Ah
```

最后的结果为9A17h。

可以增强自动加法器的电路功能以正确进行16位数的加法吗？当然可以。需要做的就是保存低字节数相加结果的进位，然后把该进位作为高字节数相加的进位输入。如何存储1位呢？当然是用1位锁存器。这时，该锁存器称为进位锁存器。

为了使用进位锁存器，需要有另一个操作码，称作“进位加”(Add with Carry)。在进行8位数加法运算时，使用的是常规“加法”指令。加法器的进位输入为0，加法器的进位输出锁存在进位锁存器中（尽管根本不必用到）。

在进行16位数加法运算时，仍然使用常规“加法”指令来进行低字节加法运算。加法器的进位输入为0，其进位输出锁存到进位锁存器中。要进行高字节加法运算就要使用新的“进位加”指令。这时，两个数相加使用进位锁存器的输出作为加法器的进位输入。如果低字节加法有进位，则其进位可用在第二次运算中；如果无进位，则进位锁存器的输出为0。

如果进行16位数减法运算，还需要一个新指令，称为“借位减”(Subtract with Borrow)。通常，减法操作需要使减数取反且把加法器的进位输入置为1。因为进位通常不是1，所以往往被忽略。在进行16位数减法运算时，进位输出应保存在进位锁存器中。高字节相减时，进位锁存器的结果应作为加法器的进位输入。

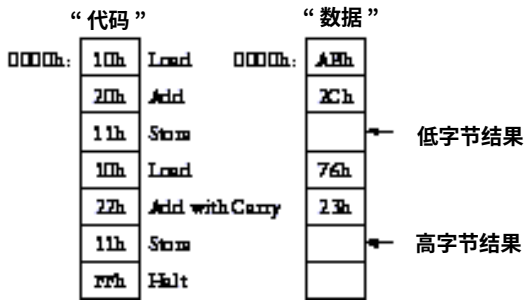
加上新的“进位加”和“借位减”操作，共有7个操作码：

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry(进位加)	22h
Subtract with Borrow(借位减)	23h
Halt	FFh

在减法和借位减法运算中，需要把送往加法器的数取反。加法器的进位输出作为进位锁存器的输入。无论何时执行加法、减法、进位加法和借位减法操作，进位锁存器都被同步。当进行减法操作，或进位锁存器的数据输出为1并且执行进位加法或者借位减法指令时，8位加法器的进位输入被置为1。

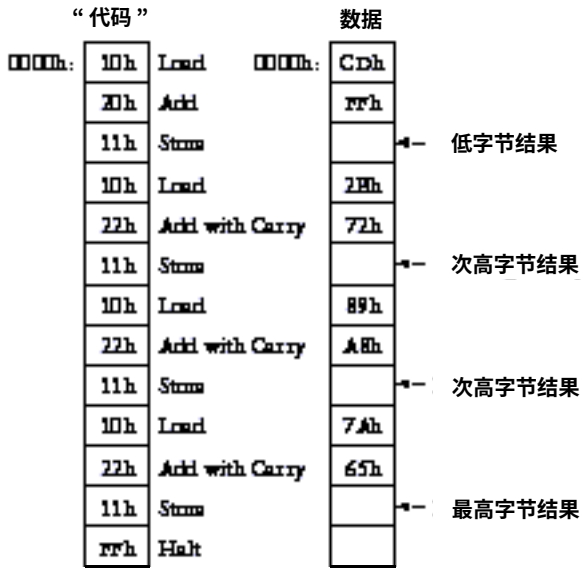
记住，只有上一次的加法或者进位加法指令产生进位输出时，进位加法操作才会使8位加

法器的进位输入为1。任何时候进行多字节数加法运算时，不管是否必要，都应该用进位加法指令计算。为正确编码前面列出的16位加法，可用如下所示方法：



不管是什么样的数，该方法都能正确工作。

有了这两个新的操作码，极大地扩展了机器处理的范围，使其不再只局限于进行8位数加法。重复使用进位加法指令，能进行16位数、24位数、32位数、40位数等更多位数的加法运算。假设要把32位数7A892BCDh与65A872FFh相加，则需要一个加法指令及三个进位加法指令：



当然，把这些数存放到存储器中并非真的很好。这不仅要用开关来表示二进制数，而且数在存储器中的地址也并不连续。例如，7A892BCDh从最低有效字节开始，每个字节分别存入存储器地址0000h、0003h、0006h及0009h中。为了得到最终结果，还必须检查地址0002h、0005h、0008h及000Bh中的数。

此外，当前设计的自动加法器不允许在随后的计算中重复利用计算结果。假设要把3个8位数加起来，然后再在和中减去一个8位数，并且存储结果。这需要一次装载操作、两次加法操作、一次减法和一次保存操作。但如果想从原先的和中减去另外一个数会怎么样呢？那个和是不能访问的，每次用到它时都要重新计算。

原因在于我们已经建造了一个自动加法器，其中的代码RAM和数据RAM阵列同时、顺序地从0000h开始寻址。代码RAM中的每条指令对应于数据RAM中相同地址的存储单元。一

旦“保存”指令使某个数据保存在数据 RAM中，这个数就不能再被装载到累加器中。

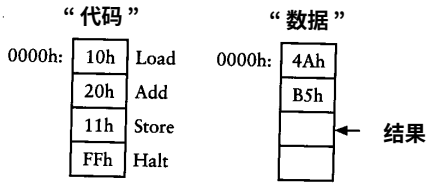
为了解决这个问题，要对自动加法器做一个基本的及大的改变。虽说刚开始看上去会异常复杂，但很快你就会看到一扇通向灵活性的大门打开了。

让我们开始吧，目前我们已经有了7个操作码：

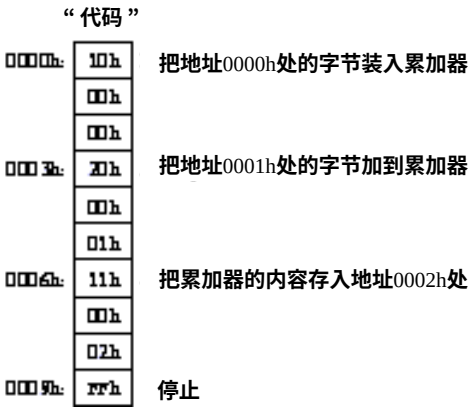
操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry	22h
Subtract with Borrow	23h
Halt	FFh

每个操作码在存储器中占1个字节。除了“停止”代码外，现在希望每条指令在存储器中占3个字节，其中第一个字节为代码本身，后两个字节存放一个 16位的存储器单元地址。对于装载指令来说，其地址指明数据在数据 RAM阵列中的存储单元，该存储单元存放要装载到累加器中的字节；对于加法、减法、进位加法和借位减法指令来说，地址指明要从累加器中加上或者减去的字节的存储单元；对于保存指令来说，地址指明累加器中的内容将要保存的存储单元。

例如，当前自动加法器所能做的最简单的工作就是加两个数。要完成这项工作，可以按照下面的方法来设置代码RAM阵列 和数据 RAM阵列：



在改进的自动加法器中，每条指令（除了“停止”）需要3个字节：

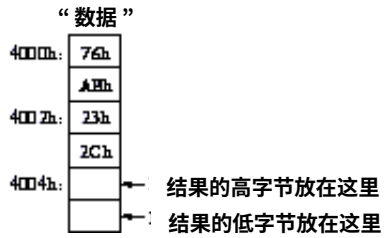


每条指令（除了“停止”）后跟2个字节，用来表示在数据 RAM阵列中的16位地址。这三个地址碰巧为0000h、0001h和0002h，它们可以是任何其他地址。

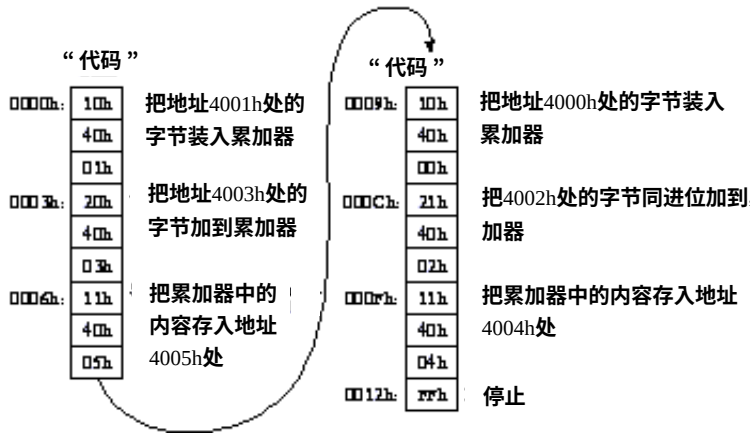
前面说明了如何使用加法和进位加法指令来相加一对 16位数——比如76ABh和232Ch。必须把2个数的低字节保存在存储器单元0000h和0001h中，把2个高字节保存在0003h和0004h中，

其相加结果保存在0002h和0005h中。

这样，我们可以用更合理的方式来保存两个加数及其结果，这可能会保存在以前从未用过的存储区域：

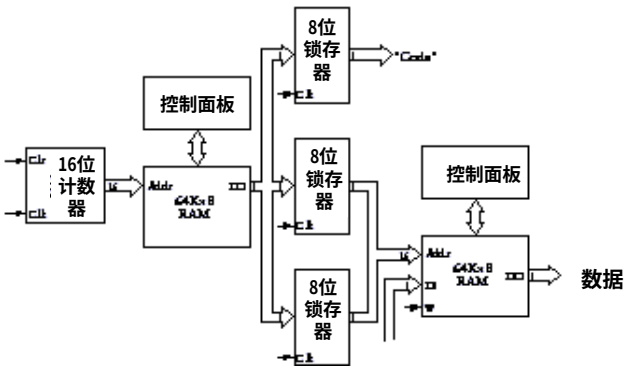


这6个存储单元不必像图中这样连在一起，它们可分散在整个64KB数据RAM阵列中的任何地方。为了把这些地址中的数相加，必须在代码RAM阵列中按如下所示设置指令：



可以看到保存在地址4001h和4003h中的两个低字节首先相加，并把结果保存在地址4005h中。两个高字节（在地址4000h和4002h中）利用进位加法进行相加，其结果保存在地址4004h中。如果去掉“停止”指令并向代码RAM中加入更多指令，随后的计算就可以简单地通过存储器地址来利用原先的数及它们的和。

实现这种设计的关键就是把代码RAM阵列中的数据输出到3个8位锁存器中，每个锁存器保存3字节指令的一个字节。第一个锁存器保存指令代码，第二个锁存器保存地址的高字节，第三个锁存器保存地址的低字节。第二和第三个锁存器的输出组成了数据RAM阵列的16位地址：



从存储器中取出指令的过程叫作取指令。在上述加法机中，每个指令长 3个字节。因每次只能从存储器中取出一个字节，因此每次取指令需要 3个时钟周期。此外，一个完整的指令周期需要四个时钟周期。所有这些变化使得控制信号变得更为复杂。

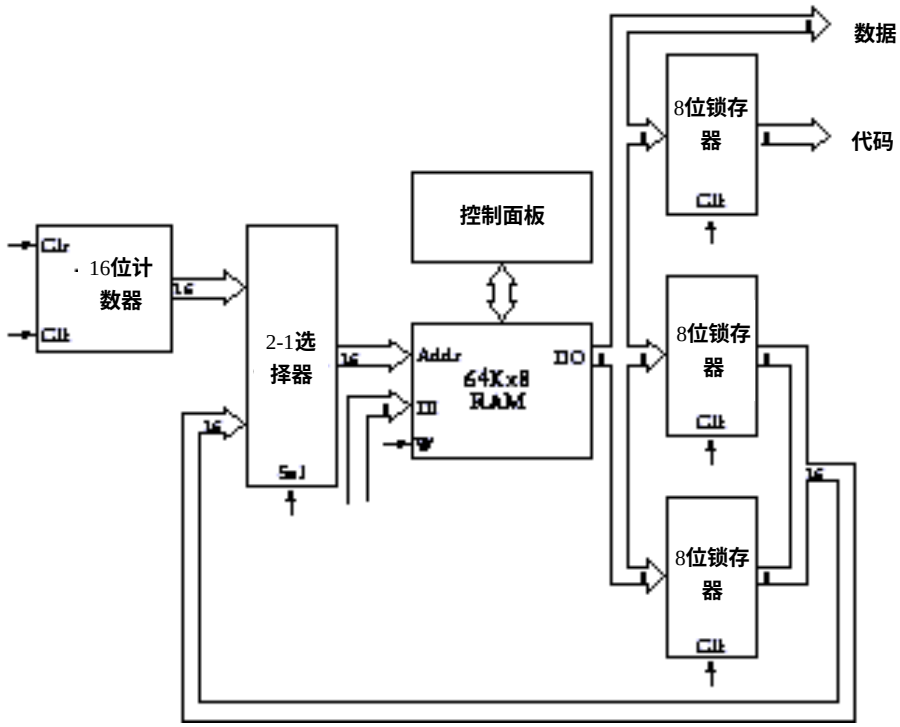
机器响应指令代码执行一系列操作称为执行指令，但这并不是说机器是有生命的东西，它也不是通过分析机器码来决定做什么。每一个机器码用唯一的方式触发各种控制信号，使机器产生各种操作。

注意，为了使上述加法机更为有用，我们已经放慢了它的速度。利用同样的振荡器，它进行数字加法运算的速度只是本章列出的第一个自动加法器的 1/4。这符合一个叫作TANSTAAFL的工程原理，TANSTAAFL的意思是“世界上没有免费的午餐”。通常，机器在某一方面好一点儿，在另一些方面必然会差一些。

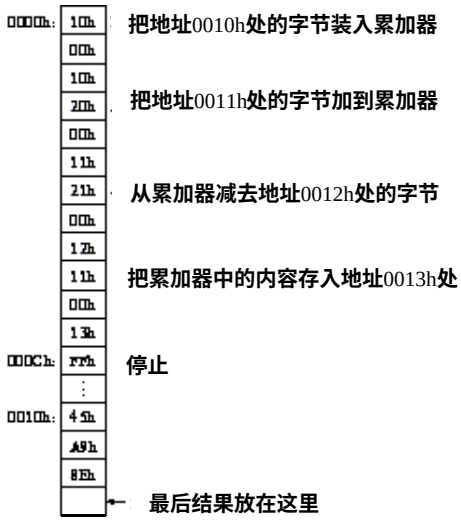
如果不用继电器来建造这样一个机器，电路的大部分显然只是两个 64KB RAM阵列。确实，早就该省去这些组件，并且一开始就决定只用 1KB的存储器。如果能保证存储的所有东西都在地址0000h～03FFh之间，那么用少于64kB的存储器也能很好地解决问题。

然而，你可能也不会太在意用到了两个 RAM阵列。事实上，也确实不用。前面介绍过的两个RAM阵列 —— 一个存储代码，一个存储数据 —— 使得自动加法器的体系结构变得尽可能清晰、简单。但既然已经决定每条指令占 3个字节 —— 用第二和第三个字节来表示数据的存储地址 —— 就不再需要有两个独立的RAM阵列，代码和数据可存储在同一个 RAM阵列中。

为了实现这个目标，需要一个 2-1选择器来确定如何寻址 RAM阵列。通常，像前面一样，其地址来自16位计数器。 RAM数据输出仍然连接到用来锁存指令代码及其 2字节地址的三个锁存器上，但它们的 16位地址是2-1选择器的第二个输入。在地址被锁存后，选择器允许被锁存的地址作为RAM阵列的地址输入：



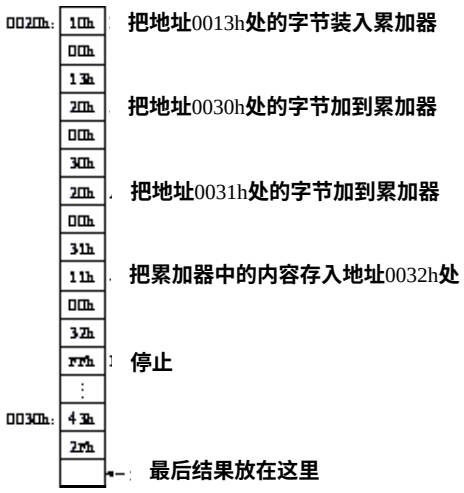
我们已经取得了很大的进步。现在把指令和数据输入到一个 RAM阵列中已成为可能。例如，下图显示出怎样把两个8位数相加再减去第三个数：



通常，指令开始于0000h，这是因为复位后计数器从0000h处开始访问RAM阵列。最后的停止指令存储在地址000Ch处。可以把这3个数及其运算结果保存在RAM阵列中的任何位置（除了开始的13个字节，因为这些存储单元已经被指令占用），因而我们选择从0010h处开始存储数据。

现在假设你需要再加两个数到结果中，你可以输入一些新的指令来替换你刚输入的所有指令，不过可能你并不想这样做。你也许更愿意在那些已有的指令末尾接着新的指令，但首先得用一个新的装载指令来替换地址 000Ch 中的停止指令。此外，还需要两个新的加法指令、一个保存指令和一个新的停止指令。唯一的问题在于有一些数据保存在地址 0010h 中，必须把这些数据移到更高的存储地址中，并且修改那些涉及到这些存储器地址的指令。

想一想，把代码和数据混放在一个RAM阵列中也许并不是一个迫切的问题，但可以肯定，这样的问题迟早会到来，因此必须解决它。在这种情况下，可能你更愿意做的就是从地址0020h处开始输入新指令，从地址0030h处开始输入新数据：



注意第一条装载指令指向存储单元0013h，即第一次运算结果存储的位置。

因此现在有开始于0000h的一些指令、开始于0010h的一些数据、开始于0020h的另外一些指令以及开始于0030h的另外一些数据。我们想让自动加法器从0000h处开始并执行所有的指令。

我们必须从000Ch处去掉停止指令，并用其他一些东西来替换它，但这样就足够了吗？问题在于无论用什么来替换停止指令都会被解释为一个指令字节，并且至此后存储器中每隔3个字节——在000Fh、0012h、0015h、0018h、001Bh和001Eh处，字节也会被解释为一个指令字节。如果其中一个正好是11h会怎样呢？这是一个保存指令。如果保存指令后的两个字节刚好指向地址0023h又会怎样呢？机器会把累加器的内容写入该地址中，但是该地址中已经包含有一些重要的东西。即使没有诸如此类的事情发生，加法器从存储器地址001Eh的下一个地址中取得的指令字节将在地址0021h中，而不是0020h中，而0020h却正好是下一个指令的真实所在。

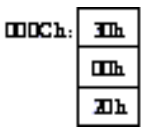
我们是否都同意不把停止指令从000Ch处移走，而期待最佳方案呢？

不过，我们可用一个叫作Jump（转移）的新指令替换它。现在把它加入到指令表中。

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

通常，自动加法器顺序寻址RAM阵列。转移指令改变其寻址模式，而从RAM阵列的某个特定地址开始寻址。这样的命令有时也叫分支（branch）指令或者goto指令，即“转到另外一个地方”的意思。

在前面的例子中，可用转移指令来替换000Ch中的停止指令：

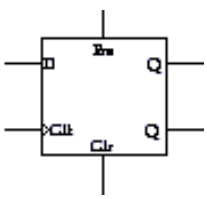


转移到地址0020h处的指令

30h就是转移指令的代码，其下的16位地址表示自动加法器要读的下条指令的地址。

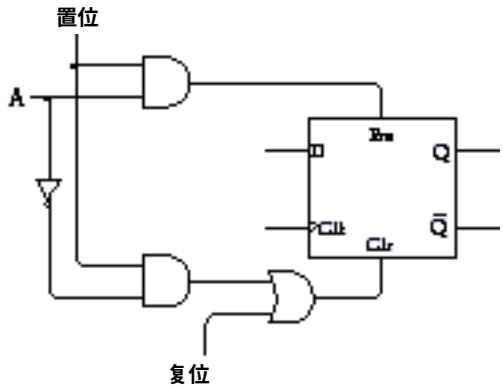
因此，在前面的例子中，自动加法器仍从地址0000h处开始，执行一条装载、一条加法、一条减法和一条保存指令，然后执行转移指令，接着继续从0020h处执行一条装载、两条加法和一条保存指令，最后执行停止指令。

转移指令影响16位计数器。当自动加法器遇到转移指令时，计数器被强制输入紧随转移指令代码的新地址，这可以通过组成16位计数器的边沿触发的D型触发器的预置（Pre）和清零(Clr)输入来实现：



前面曾讲过，正常操作下，预置和清零输入都应该为 0。但如果Pre = 1，则Q = 1；如果Clr = 1，则Q = 0。

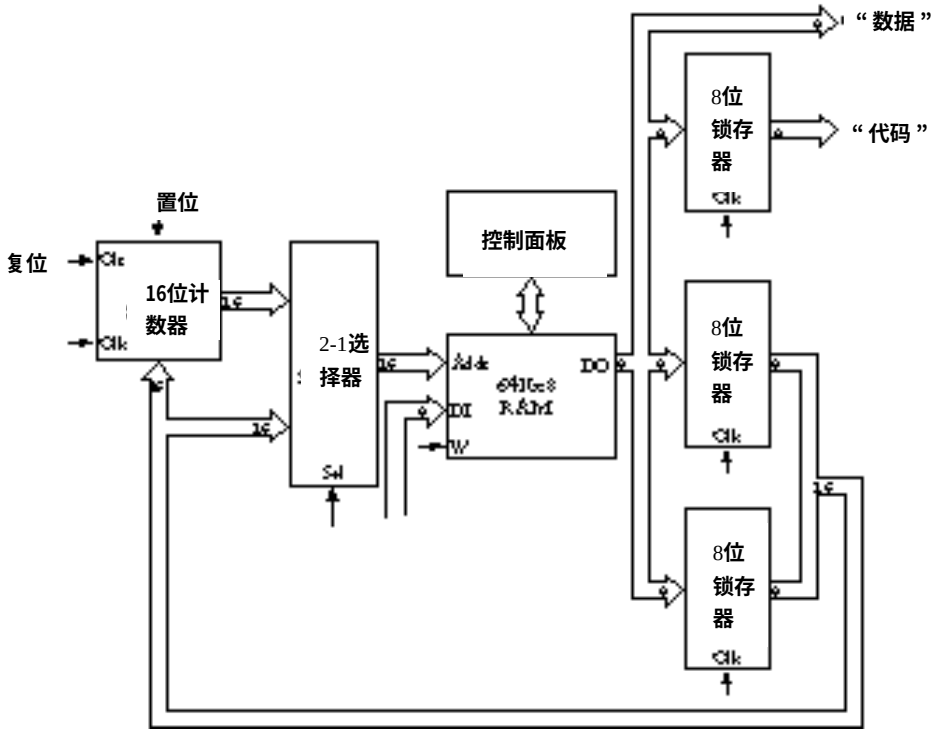
如果想装载一个新值（称作A，代表地址）到单个触发器中，可这样连线：



通常，置位信号为0。这时，触发器的预置端为0。除非复位信号为1，否则清零端也为0。这样触发器可以不通过置位信号就可以清零。当置位信号为 1时，若A = 1，则Pre = 1且Clr = 0；若A = 0，则Pre = 0且Clr = 1。这意味着Q端的值设置为A端的值。

16位计数器的每一位都需要一个这样的触发器。一旦装载一个特定的值，计数器将从那个值开始继续计数。

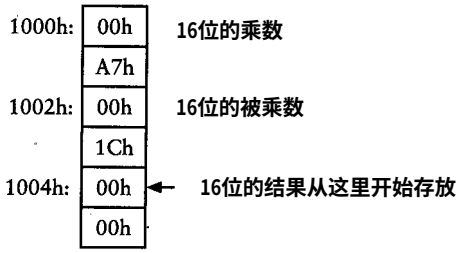
然而，这些变化并不大。从RAM阵列中锁存的16位地址既可作为2-1选择器（它允许该地址作为RAM阵列的地址输入）的输入也可作为16位计数器的输入并由置位信号设置：



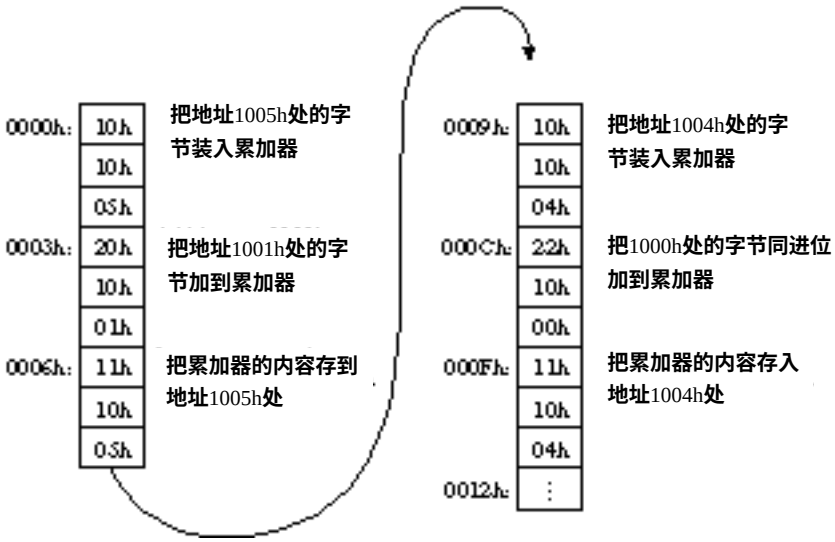
显而易见，只有当指令代码为30h且其后面的地址被锁存，我们才必须保证置位信号为 1。

转移指令当然很有用，但它并非和一条只有时跳转而并非时刻跳转的指令一样有用，这样的一个指令叫作条件转移。为了显示该命令如何有用，可提出这样一个问题：怎样才能让自动加法器完成两个8位数的相乘？例如，怎样才能得到像A7h乘以1Ch这样简单运算的结果？

很容易，不是吗？两个8位数相乘的结果是一个16位数。为了方便起见，乘法中的3个数都用16位数来表示。首要的工作是决定把乘数和乘积放在何处：



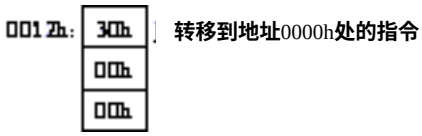
每个人都知道A7h和1Ch（即十进制的28）相乘的结果与A7h相加28次的结果相同。因此，在1004h和1005h处的16位数就是累加结果。下图显示的是把A7h加一次到那个位置的代码：



在这6条指令执行完后，存储单元1004h和1005h处的16位数等于A7h乘以1。因此，为了使这16位数等于A7h乘以1Ch，这6个指令必须重复执行27次。可以通过在地址0012h处接着输入27次这6个指令来实现；也可以在0012h处输入停止指令，然后按28次复位键来得到最终结果。

当然，这两个方案都不理想。它们需要你做某些事情 (输入大批指令或者按复位键)的次数和乘数相当。当然你不愿意这样去进行16位数的乘法运算。

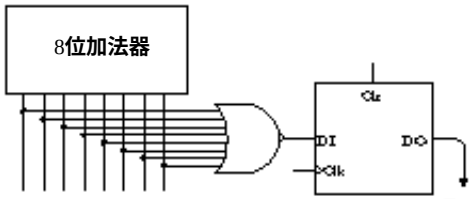
但是如果在0012h处输入转移指令会怎么样呢？这个指令使计数器从0000h重新开始计数：



这当然是一个技巧。第一次执行指令后，存储单元1004h和1005h处的16位数等于A7h乘1，然后转移指令使其返回到存储器顶部。第二次执行指令后，此16位数等于A7h乘2。终于，其

结果将等于A7h乘1Ch。不过这样的过程并不会停止，它将不断地运行、运行、运行。

我们想让转移指令做的是使循环过程只重复所需的次数，这就是条件转移，它实施起来并不困难。我们要做的第一件事情就是增加一个与进位锁存器类似的 1位锁存器。因为只有8位加法器的输出全为0时它才锁存1，所以叫它零锁存器：



零标志位

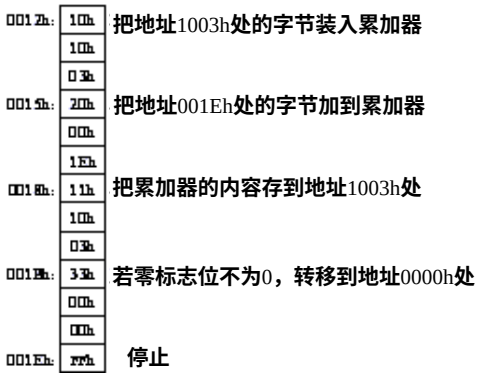
只有当或非门的8个输入全为0时，其输出才为1。同进位锁存器的时钟输入一样，只有当加法、减法、进位加法或借位减法指令运行时，零锁存器的时钟输入才锁存一个数，这个被锁存的数值叫作零标志位。注意它，是因为它似乎行为相反：如果加法器输出全为 0，则零标志位为1；若加法器输出不全为0，则零标志位为0。

利用进位锁存器和零锁存器，可以在指令表中再添加四条指令：

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，只有当零锁存器输出为0时，非零转移指令才转移到指定地址。换句话说，如果上一次加法、减法、进位加法和进位减法指令计算结果为 0，则没有转移发生。实现这个设计只需在实现常规转移命令的控制信号上再加上一个控制信号：如果为非零转移指令，则只有当零标志位为0时，16位计数器的置位信号才被触发。

利用上述代码实现两个数的乘法所需的操作可由如下开始于地址 0012h处的指令完成：



正如我们所设计的，循环一次后，位于1004h和1005h处的16位数等于A7h乘以1。上图中的这些指令把字节从1003h处装载到加法器中，此字节为1Ch。再把这个字节与001Eh处的数据相加，此处数据正好是停止指令，但当然也是有效数字。把FFh同1Ch相加与从1Ch减去1的结果相同，都等于1Bh。这个值不为0，所以零标志位为0，字节1Bh存回到地址1003h处。接下来是一条非零转移指令，零标志位没有置为1，所以转移发生。下一条指令位于地址0000h处。

记住，存储指令不会影响零标志位。零标志位只能被加法、减法、进位加法、借位减法指令所影响，因此它同这些指令中最近一个执行时所设置的值相同。

循环两次后，位于1004h和1005h处的16位数将等于A7h乘以2。而1Bh加上FFh等于1Ah，不是0，因此又返回到存储器顶部。

循环到第28次时，位于1004h和1005h处的16位数等于A7h乘以1Ch。位于1003h处的值等于1，它将加上FFh结果等于0，因此零标志位被置位。非零转移指令不再转移到存储器地址0000h处，相反，下一条指令为停止指令。至此，我们完成了全部工作。

现在可以肯定，很长一段时间以来我们已经装配了一组硬件，同时可以把它叫作计算机。当然，它只是一台原始的计算机，但它毕竟是一台计算机。它与我们以前设计的计算器的不同之处在于条件转移指令，控制重复或循环是计算机和计算器的区别。这里已经演示了条件转移指令是如何使得这台机器进行两个数的乘法运算的。用类似的方法，它也能计算两个数的除法。而且，还不局限于8位数。它能加、减、乘、除16位、24位、32位甚至更多位的数，而且如果它能实现这些操作，也就能计算平方根，对数和三角函数。

既然已装配了一台计算机，就可以开始使用一些计算机方面的词汇。

我们装配的计算机归类为数字计算机，因为它采用的是离散值。曾经有过模拟计算机，但它们正逐渐消失。（数字数据是离散数据，是具体的确定的值；而模拟信息是连续的、在整个范围内变化的值。）

数字计算机有4个主要部分：处理器、存储器、至少一个输入设备和一个输出设备。上述机器中，存储器是一个64KB的RAM阵列。输入和输出设备分别是RAM阵列控制面板上的几行开关和灯泡。这些开关和灯泡使人们可以输入数据到存储器并检查结果。

处理器是计算机中除存储器、输入/输出设备以外的一切东西。处理器也叫中央处理器单元或CPU。再通俗一点儿，处理器有时也称作计算机的大脑。但尽量避免用这样的术语，这是因为在本章中我们所设计的东西根本不像大脑。（今天，微处理器这个词用得非常普及。微处理器只是一个很小的处理器，通过采用第18章将要讲到的技术而实现。但此刻我们用继电器所建造的东西则很难用“微”来定义。）

我们所建造的处理器是一个8位处理器。累加器宽度为8位，并且许多数据通路的宽度都是8位，只有RAM阵列的地址的数据通路是16位的。如果用8位的地址通路，则存储器容量只能限于256字节而非65 536字节，那样处理器则有太大的局限性。

处理器有一些组件。已经确定的一个是累加器，它是一个简单的锁存器，用来在处理器内部保存数据。我们所设计的计算机中，8位反向器和8位加法器一起称作算术逻辑单元或ALU。ALU只能进行算术运算，主要是加法和减法。在稍微复杂一点儿的计算机中（我们将会看到），ALU也可进行逻辑运算，如“与”、“或”、“异或”。16位计数器叫作程序计数器PC。

我们的计算机是用继电器、电线、开关和灯泡建造的，所有这些都是硬件。与之对应，

指令和输入存储器中的其他数据叫作软件，之所以叫“软件”是因为它们比硬件更容易改变。

当谈论计算机时，“软件”和“计算机程序”，更简单地讲“程序”是同义的，编写软件也称作计算机程序设计。当采用一系列计算机指令使计算机进行两个数的乘法时，我们所做的工作就是计算机程序设计。

通常，在计算机程序中，可以区分代码（即指令）和供代码使用的数据。有时这种区分并不明显，如停止指令还可作为数－1执行双重功能。

计算机程序设计有时也叫编写代码或编码。有时计算机程序员也叫编码员，尽管一些人可能认为这是一个贬义的名词。程序员更愿意被称作“软件工程师”。

处理器可以响应的操作码（如指装载和存储的10h和11h）叫作机器码，或机器语言。之所以用“语言”这个术语是因为机器码类似于可读/写的人类语言可被机器理解和响应。

我们要用很长的短语表示机器所执行的指令，如：进位加法（Add with Carry）。通常，机器码都分配指定了用大写字母表示的短的助记符，这些助记符有2或3个字符。下面是一系列可能的上述计算机所能识别的机器码的助记符：

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

这些助记符特别适于和另外一对简洁短语结合使用。例如，不说像“把1003h处的值装载到累加器中”这样罗嗦的话，而是用下面语句来代替：

```
LOD   A，[1003h]
```

位于助记符LOD右边的A和[1003]叫作操作数，它们是特定的装载（Load）指令的操作对象。左边的操作数为目的操作数（A代表累加器），右边的为源操作数，方括号表示要装载到累加器中的值不是1003h，而是存储在存储器地址1003h中的值。

同样，指令“把001Eh处的字节加到累加器中”可简写为：

```
ADD   A，[001Eh]
```

而“把累加器中的内容保存到地址1003h处”记作：

```
STO   [1003h]，A
```

注意，目的操作数（存储指令的存储单元）仍然在左边，源操作数在右边。累加器的内容存储在地址1003h处。指令“若零标志位不为1则转移到0000h处”可简洁地记作：

```
JNZ 0000h
```

该指令中没有使用方括号，这是因为该指令是转移到地址0000h处而不是转移到地址

0000h中保存的值所表示的位置处。

用缩写指令的形式来表示很方便，因为指令能以可读的方式连续列出来而不需画出存储器的分配图。为了表示某一指令存储在某一地址，可以用一个十六进制地址后加冒号来表示，如下所示：

```
0000h：LOD A，[1005h]
```

下面表示了一些存储在某一地址的数据：

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

用逗号隔开的两个字节表示第一个字节保存在左边的地址中，第二个字节保存在紧接着该地址的下一个地址中。上述三行相当于：

```
1000h：00h，A7h，00h，1Ch，00h，00h
```

因此，整个乘法程序可写成如下一系列语句：

```
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

            JNZ 0000h
001Eh:  HLT
1000h:  00h，A7h
1002h:  00h，1Ch
1004h:  00h，00h
```

使用空格和空行只是为了使程序具有更好的可读性，以方便人们阅读程序。

写代码时最好不要用真实的数字地址，因为它们是会变的。例如，如果要把数字存储到地址2000h～2005h处，需要重写许多语句。较好的方法是使用标号来指定存储单元，这些标号是简单的单词，或类似于单词的东西，如：

```
          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，标号NUM1、NUM2和RESULT都表示保存两个字节的存储单元。在这些语句中，标号NUM1+1、NUM2+1和RESULT+1都指向特定标号后的第二个字节。注意，*NEG1*（negative one）用来标记HLT指令。

此外，为了不忘记这些语句的意思，可以加上一些注释，它们与语句之间用分号隔开：

```
BEGIN:  LOD A, [RESULT+1]
        ADD A, [NUM1+1]         ;Add low-order byte (加低字节)
        STO [RESULT+1], A
        LOD A, [RESULT]
        ADC A, [NUM1]           ;Add high-order byte (加高字节)
        STO [RESULT], A
        LOD A, [NUM2+1]
        ADD A, [NEG1]           ;Decrement second number (第二个数减1)
        STO [NUM2+1], A
        JNZ BEGIN
NEG1:   HLT
NUM1:   00h, A7h
NUM2:   00h, 1Ch
RESULT: 00h, 00h
```

以上表示的是一种计算机程序设计语言，称作汇编语言。它是全数字的机器代码和指令描述性语言的综合，且存储器地址用符号表示。人们有时会把机器语言和汇编语言弄混淆，因为它们是表示同种事情的两种不同的方法。汇编语言的每条语句都对应于机器代码的特定字节。

如果你想为本章所创建的计算机编写程序，你可能首先想用汇编语言写出来（在纸上）。然后，在认为它正确并准备测试它时，可以对它进行手工汇编：这意味着用手工的方法把每一个汇编语句转换成机器代码，仍然写在纸上。接着，你可以用开关把机器码输入到RAM阵列并运行该程序，即让机器执行指令。

学习计算机程序设计的概念时，不可能很快就能正确知道程序的毛病所在。编写代码时——特别是用机器代码——很容易产生错误。输入一个错误的数字已经很不好了，但如果输入一条错误的指令会怎么样呢？本想输入10h(装载指令)，但却输入了11h（保存指令），不但机器不会把期望的数据装载，而且该处的数据还会被累加器中的内容覆盖。

一些错误可以导致难以预料的结果。假设使用无条件转移指令转移到没有有效指令代码的位置，或者偶然使用存储指令覆盖了一些指令，任何事情都可能发生（经常如此）。

上述乘法程序中也有一些毛病。如果你执行它两次，则第二次将会是A7h乘以256，并且结果将加到原来计算的结果中。这是因为程序执行一次后，地址1003h处的值为0。当程序第二次执行时，FFh将加到那个值中，其结果不为0，程序将继续执行直到它为0。

我们已看到上述机器可以进行乘法运算，同样，它也可以进行除法运算。此外，它可利用这些基本功能进行平方根、对数和三角函数的计算。机器所需要的只是用来进行加法、减法的硬件及利用条件转移指令来执行适当代码的一些方法。正如一个程序员所说：“我可以用软件完成其余功能”

当然，软件可能相当复杂。许多书中都描述了一些算法供程序员解决专门的问题，本书还没准备这样做。我们一直在考虑自然数而没有考虑如何在计算机中表示十进制小数，我们将在第23章介绍它。

前面已说过几次，建造这些设备的所有硬件在100多年前就有了。但本章中出现的计算机在那时却没有建造出来。在20世纪30年代中期，最早的继电器计算机制造出来时，包含在设计中的许多概念还未形成，直到1945年左右人们才开始意识到。例如，直到那时候，人们仍然设法在计算机内部使用十进制数而不是二进制数；计算机程序也并非总是存储在存储器中，而是有时把它存在纸带上。特别是早期计算机的存储器非常昂贵且体积庞大。不管是在100年前还是在现在，用500万个电报继电器来建造64KB的RAM阵列都是荒唐的。

当我们展望和回顾计算器和计算装置的历史时，可能会发现根本没必要建造这样精致的继电器计算机。就像在第12章提到的，继电器最终会被真空管和晶体管这样的电子设备所取代。或许我们也会发现他人制造的相当于我们设计的处理器和存储器的东西能小到放在手掌中。