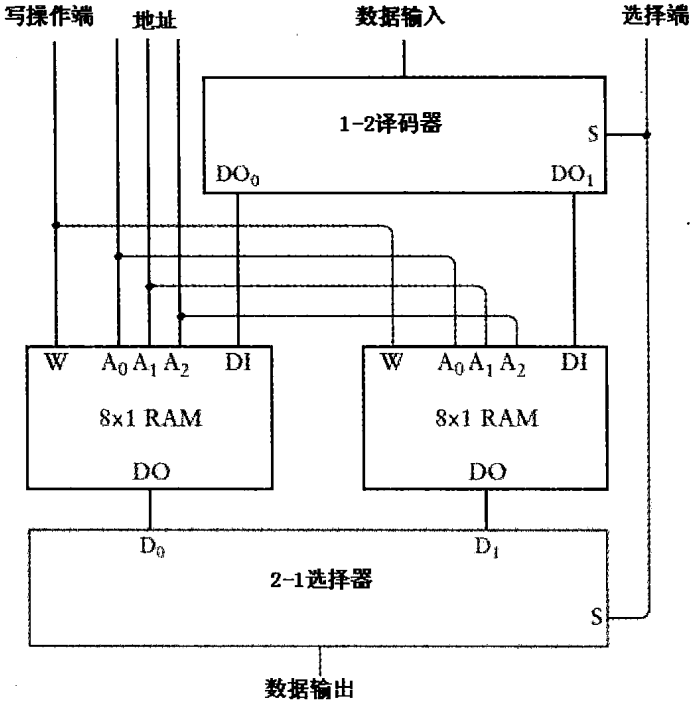
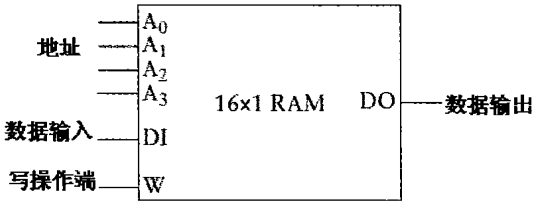


“选择”端之所以连接到译码器和选择器，主要作用是在两个 8×1 RAM 阵列中选择一个，本质上它扮演了第 4 根地址线的角色。因此这种结构实质上是一种 16×1 的 RAM 阵列，如下图所示。

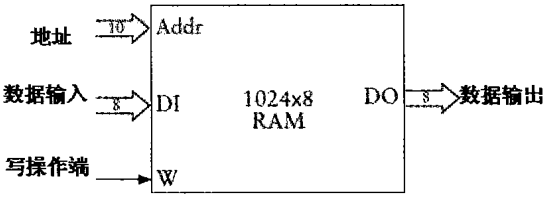


上图所示的 RAM 阵列存储容量为 16 个单位，每个单位占 1 位。

RAM 阵列的存储容量与其地址输入端的数量有直接的联系。在没有地址输入端的情况下（只有 1 位锁存器和 8 位锁存器的情况），只能存储 1 个单位的数据；当存在 1 个地址输入端时，可以存储 2 个单位的数据；有两个地址输入端时，可以存储 4 个单位的数据；有 3 个地址输入端时，可以存储 8 个单位的数据；有 4 个地址输入端时，可以存储 16 个单位的数据。我们可以把它们之间的关系归纳成如下等式：

$$\text{RAM 阵列的存储容量} = 2^{\text{地址输入端的个数}}$$

前面已经向大家演示了怎么搭建小型 RAM 阵列，你可能会问：为什么不搭建一个大规模的 RAM 阵列呢？就像下面这样。



上图所示的 RAM 阵列可存储 8192 个比特的信息，每 8 个比特为一组，共分为 1024 个组。因为 2 的 10 次方恰好是 1024，所以地址端共有 10 个输入端口。电路还包括 8 位的数据输入端和 8 位的数据输出端。

从专业的角度来讲，这个 RAM 阵列的存储容量为 1024 个字节。就好比一个邮局放置了 1024 个邮箱，而每个邮箱里面都可以存放 1 字节大小的邮件（希望不是垃圾邮件）。

1024 字节通常简称为 1 千字节（kilobyte），1K 这种称呼不可避免地要引起许多混淆。其中它的前缀 kilo（源于希腊文 khilioi，意思为 1000），经常在公制系统中用到。比如 1 千克（ kilogram）代表着 1000 克（grams）；1 千米（kilometer）代表着 1000 米（meter）。有所不同的是，这里所说的 1 千字节却代表着 1024 个字节——并非 1000 个字节。

它们之间不同的根本原因在于公制系统是基于 10 的幂的计数系统，而计算机采用的是基于 2 的幂的计数系统，它们之间没有交集。比如 10 的幂为 10、100、1000、10000、100000 等，而 2 的幂为 2、4、8、16、32、64 等。我们可以证明不存在一对整数 a 和 b 使得 10 的 a 次幂与 2 的 b 次幂相等。

但是偶尔也会碰见非常接近的数字。事实的确如此，1000 十分接近 1024，用数学化的描述方法可以称这种关系为“约等于”，这样我们可以得到相应的数学表达式：

$$2^{10} \approx 10^3$$

这个表达式并非空穴来风，它真正的意义在于表明 2 的某次幂和 10 的某次幂几乎相等。我们利用这一巧合可以很方便地把 1024 个字节的存储空间用 1 千字节来表示。

千字节可以简写为 KB。这样我们可以说前面所讲过的那个 RAM 阵列存储能力为 1024 个字节，也可以说成是 1KB。

绝不能认为 1KB 的 RAM 阵列的存储能力为 1000 字节，它实际上是大于 1000 字节，是 1024 个字节，为了准确而清晰地表达你脑海中的数据，我们可以使用“1 KB””或“1 千字节”这两种通用的表述方式。

存储容量为 1KB 的存储系统由 8 个数据输入端、8 个数据输出端和 10 个地址输入端所组成。由于这些字节是由 10 个地址输入端来标识和访问的，所以这种 RAM 阵列存储容量为 2^{10} 个字节。如果我们再加上一条地址线，它的存储容量将变成原来的两倍。下面的公式表示了存储容量的翻倍的过程。

$$1\ \text{KB} = 1024\ \text{B} = 2^{10}\ \text{B} \approx 10^{3}\ \text{B}$$
$$2\ \text{KB} = 2048\ \text{B} = 2^{11}\ \text{B}$$
$$4\ \text{KB} = 4096\ \text{B} = 2^{12}\ \text{B}$$
$$8\ \text{KB} = 8192\ \text{B} = 2^{13}\ \text{B}$$
$$16\ \text{KB} = 16{,}384\ \text{B} = 2^{14}\ \text{B}$$
$$32\ \text{KB} = 32{,}768\ \text{B} = 2^{15}\ \text{B}$$
$$64\ \text{KB} = 65{,}536\ \text{B} = 2^{16}\ \text{B}$$
$$128\ \text{KB} = 131{,}072\ \text{B} = 2^{17}\ \text{B}$$
$$256\ \text{KB} = 262{,}144\ \text{B} = 2^{18}\ \text{B}$$
$$512\ \text{KB} = 524{,}288\ \text{B} = 2^{19}\ \text{B}$$
$$1{,}024\ \text{KB} = 1{,}048{,}576\ \text{B} = 2^{20}\ \text{B} \approx 10^{6}\ \text{B}$$

请注意最左侧一排的数字也以 2 的幂的顺序逐步递增。

我们把 1024 个字节简化成为了 1 KB，相同的逻辑，我们把 1024 KB 统称为 1 兆字节（megabyte，希腊文中的 mega 意味着宏大），兆字节通常缩写为 MB。下面这个例子表示了兆字节为单位的存储容量翻倍的过程。

$$1\ \text{MB} = 1{,}048{,}576\ \text{B} = 2^{20}\ \text{B} \approx 10^{6}\ \text{B}$$
$$2\ \text{MB} = 2{,}097{,}152\ \text{B} = 2^{21}\ \text{B}$$
$$4\ \text{MB} = 4{,}194{,}304\ \text{B} = 2^{22}\ \text{B}$$

8 MB = 8,388,608 B = 2^{23} B

16 MB = 16,777,216 B = 2^{24} B

32 MB = 33,554,432 B = 2^{25} B

64 MB = 67,108,864 B = 2^{26} B

128 MB = 134,217,728 B = 2^{27} B

256 MB = 268,435,456 B = 2^{28} B

512 MB = 536,870,912 B = 2^{29} B

1,024 MB = 1,073,741,824 B = 2^{30} B ≈ 10^{9} B

希腊文中的 giga 意味着巨大，1024 MB 也就被顺其自然地称为 1 吉（gigabyte）字节，缩写为 GB。

同理，1 太字节（terabyte，teras 希腊语意思为巨人）表示 2^{40} 个字节（约为 10^{12}），也就是 1,099,511,627, 776 个字节，太字节的缩写为 TB。

1 KB 近似为 1000 个字节，1 MB 近似为 100 万个字节，1 GB 近似为 10 亿个字节，1 TB 近似为 1 万亿个字节。

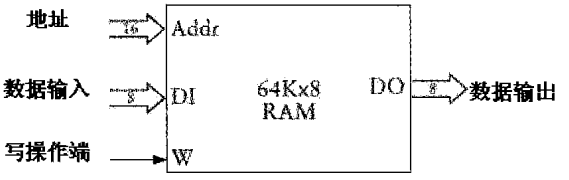
比 TB 还高数量级平时一般很少使用，比如 2^{50} 个字节表示为 1 批字节（petabyte），计算出来就是 1,125,899,906,842,624 个字节，约等于一千万亿，即 10^{15} 字节。1 安字节（exabyte）代表 2^{60} 个字节，也就是 1,152,921,504,606,846,976 个字节，约为 100 万的 3 次方，即 10^{18} 。

我们来补充一些生活中的基本常识。在写本书时（1999 年），家用电脑的随机存储器的容量一般为 32 MB、64 MB 或 128 MB（为了避免混淆，这里的任何描述都不涉及硬盘驱动器的任何内容，范围仅仅限定为 RAM），通过计算可以得到它们的存储大小分别为 33,554,432 个字节、67,108,864 个字节和 134,217,728 个字节。

简洁明了是我们人类交流方式的一大特色。比如，家中电脑内存大小如果为 65,536 字节，我们会说“我的是 64 K（这句话很可能是在 1980 年听到的）”；家中电脑内存大小如果为 33,554 ,432 字节，我们会说“我的是 32 M”。有少数电脑配备了 1,073,741,824 字节的内存，他们或许会说“我的可是上了 G 的（有时这句话的英文表述会让人误以为你在谈论音乐——I've got a gig）”。

有时人们可能会用到千比特或兆比特（注意是比特而不是字节），这只是极少数情况。当我们讨论涉及存储器的相关问题时，通常使用的是字节数而非比特（需要的时候可以通过把字节数乘以 8 将其转换成比特）。有一种情况下我们会经常用到千比特和兆比特，那就是在描述在线路中流动的数据时，很多句子中经常会出现千比特每秒（kbps）或兆比特每秒（mbps）这些用语。例如，一台 56K 的调制解调器指的是其数据处理速度为 56 千比特每秒，而不是 56 千字节每秒。

既然我们已经学会如何构造任意大小的 RAM 阵列，接下来继续对这个问题深究下去。假设现在已经构造好了一个容量为 65,536 字节的存储器组织，如下图所示。



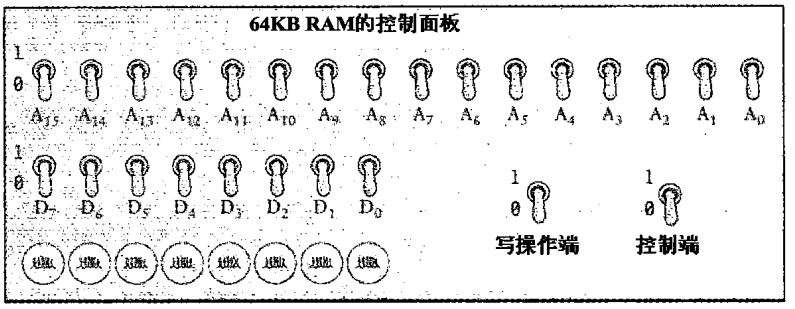
为什么选择大小为 64 KB 的 RAM 阵列？而非 32 KB 或 128 KB？因为 65,536 是一个约整数，转换为幂的形式就是 2^{16}，这个 RAM 阵列需要配备 16 位的寻址端。换句话说，该地址恰好可以用 2 个字节表示。将地址范围转化为十六进制就是 0000h～FFFFh。

我前面也提到过，64 KB 的内存是 1980 年的个人电脑的主流配置，但它的确不是用电报继电器组成的。我们可以用继电器来组成一块内存吗？我也相信你不想这么做。在我们先前的讨论中，存储每个比特需要 9 个继电器，推算一下 64K×8 的 RAM 阵列就需要至少 500 万个继电器！

如果用一种控制面板来辅助我们管理对这块 64KB 存储器的操作——包括写数据和读数据，一切将会直观明了。在这款控制面板上，有 16 个开关用于控制地址位，还有 8 个开关用来控制要输入的 8 比特数据。写操作端也用一个开关来表示，8 个灯泡用来显示 8 位数据，这个控制面板如下图所示。

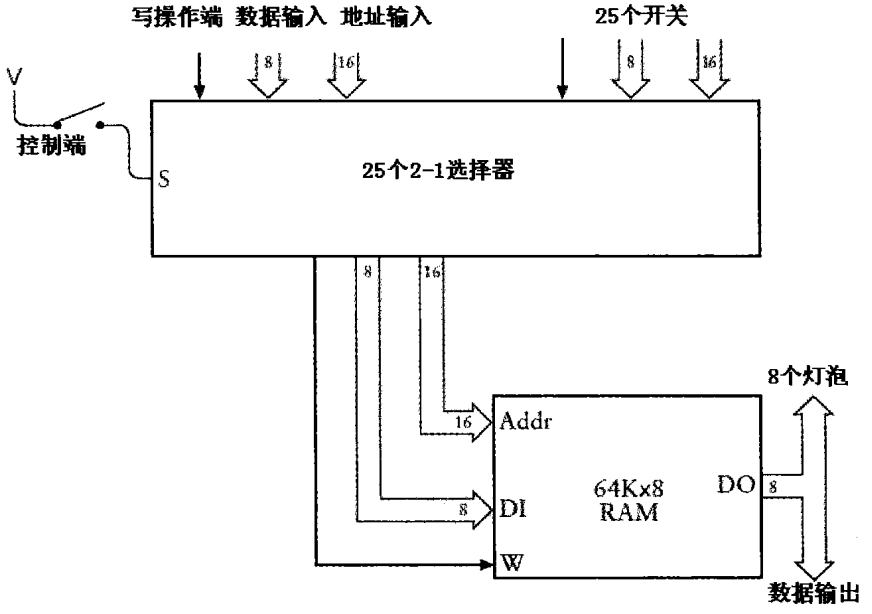
初始状态下所有的开关均置为 0。其中右下角有一个标识为控制端（takeover）的开关，这个开关的作用是确定由控制面板还是由外部所连接的其他电路来控制存储器。如果其他电路连接到与控制面板相连的存储器，这时控制端置 0（如图所示），此时存储器由其他电路系统接管，控制面板上的其他开关将不起任何作用；当控制端置 1 时，控制

面板将重新获得对存储器的控制能力。

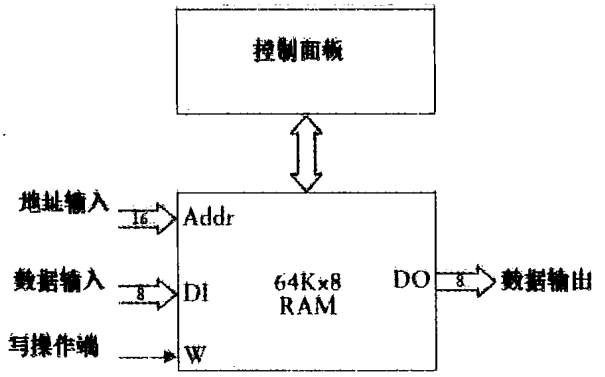


这种功能可以用一些 2-1 选择器来实现。仔细数一下会发现，我们需要 25 个 2-1 选择器——其中包括 16 个地址输入端、8 个数据输入端，以及 1 个写操作端。电路如下图所示。

当控制端开关断开时，RAM 阵列的地址端、数据输入和写操作端的数据全部来源于外部信号，也就是在 2-1 选择器的左上角的输入信号；当控制端开关闭合，RAM 阵列的地址端、数据输入端和写操作端的数据来源于控制面板开关发出的信号。但最终 RAM 阵列的输出信号都会传输到 8 个灯泡上或其他可能的地方。



下面这幅是控制面板与 64K×8 RAM 阵列的逻辑结构框图。



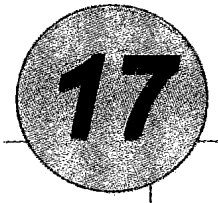
当控制端开关闭合时，通过操作 16 个地址开关，可以选择 65,536 个地址中的任何一个，灯泡的状态将表示该地址中所保存的 8 位数据。我们可以使用 8 个数据开关表示出一个新数，然后把写操作端置 1，从而将数据写入存储器。

64K×8 的 RAM 阵列和控制面板这一组合的确很实用，它可以帮助我们存储 65,536 个 8 位数据并且读取其中的任意一个。与此同时，我们也给其他部件提供了接入系统的机会——需要接入系统的通常是一些电路部件——这些部件可以轻易地读取并利用存储器中存放的数据，还可以把数据写入存储器。

关于存储器有一个问题尤其值得我们注意，而且需要特别注意。在学习第 11 章的时候，我们曾介绍过逻辑门的概念及原理，但是没有画出组成逻辑门的单个继电器的结构图。特别是，当时没有指明每个继电器都与某个电源连接在一起。只要继电器连通，电流就会流过电磁线圈并产生磁场，继而吸下金属片。

一个辛辛苦苦装满 65,536 字节珍贵数据的 64 K×8 RAM 阵列，如果断掉电源，会发生什么事情呢？首先所有的电磁铁都将因为没有电流而失去磁性，随着“梆”的一声，金属片将弹回原位，所有继电器将还原到未触发状态。RAM 中存储的数据呢？它们将如风中残烛般消失在黑暗中。

正因为如此，随机访问存储器也被称为易失性（volatile）存储器。为了保证存储的数据不丢失，易失性存储器需要恒定的电流。



自动操作

我们人类的创造能力与勤奋精神常常令我感叹不已，但人类的本性却是相当懒惰的。举个简单而又常见的例子，我们总是不情愿工作。我们对工作的反感是如此的强烈——当然人类也很聪明——以至于情愿花费大量的时间去设计并制造一些设备，哪怕这些设备只能将工作时间缩减几分钟。悠闲地躺在吊床上，看着自己刚发明的新奇工具自动修剪草坪，没有什么事情能比这更让我们快乐的神经为之一动了。

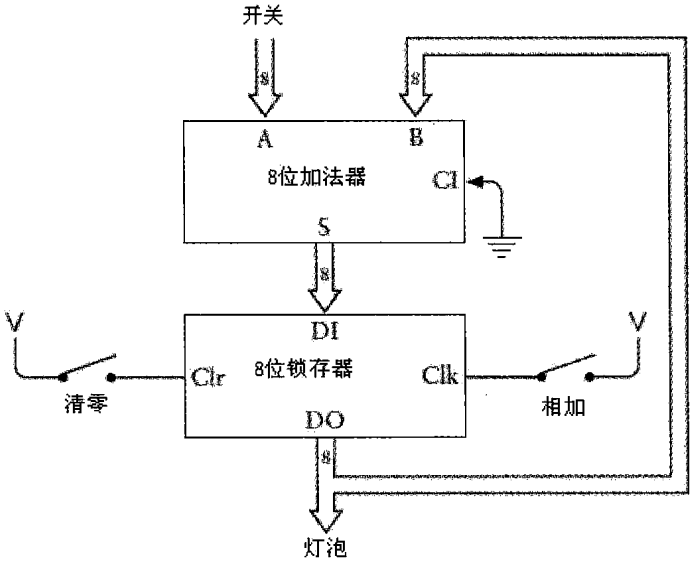
很可惜，我并不打算在本章中介绍自动割草机的设计。在这里我们将学会设计更加先进的机器，目标就是要使加减法的过程自动化，这听起来有些难以置信。但是，本章最终设计出的机器用途将十分广泛，因为它可以解决所有能用加、减法处理的问题，而事实上现实中的很多问题都是可以用加法与减法来解决的。

当然，随着机器变得越来越精密，其复杂程度也越来越高，因此对其构造的某些部分理解起来比较困难。因此如果你略去了某些复杂的细节，这也是在情理之中，没有人会为此责怪你。有时，我们会感到焦躁不安，甚至厌倦，发誓再也不会为了解决一个复杂的数学难题而去求助于某些电子或机械设备。但是请保持耐心，在本章的末尾我们将会设计出一种机器，我们可以称它为计算机（Computer）。

回忆我们曾在第 14 章讨论过的一个加法器。这个版本的加法器包括一个 8 位的锁存

器，用于对 8 个开关的输入数据进行迭代求和。下面是其结构图。

从图中可以看出，8 位锁存器利用触发器来保存 8 位数据。使用这个设备时，首先需要按下清零开关使锁存器中的内容全部都变为 0，然后通过开关输入第一个数。加法器只是简单地将这个数字和锁存器输出的 0 进行求和，因此相加的结果与原先输入的数字是一样的。按下相加开关可以把这个数保存在锁存器中，最后会点亮某些灯泡以显示它。现在通过开关输入第二个数，加法器把它与已经存放在锁存器中的第一个数相加。再次按下相加开关，就可以把相加的结果存入锁存器中，并通过灯泡显示这个结果。通过这种方式，可以把一串数相加并显示运行结果。显然，这种设计方案存在一个缺陷：8 个灯泡无法显示大于 255 的数。



对于第 14 章所介绍的这种电路，目前为止只讲到了一种锁存器，它是电平触发（level triggered）的。在电平触发的锁存器中，为了保存数据必须将时钟输入端首先置 1，然后回置为 0。当时钟输入端为 1 时，锁存器的数据输入端可以改变，而这些变化将会影响到数据输出。在第 14 章的后半部分还介绍了边沿触发（edge-triggered）的锁存器，这种锁存器在时钟输入从 0 跳变为 1 的瞬间保存数据。边沿触发器在很多方面更加易于使用，因此假定本章用到的所有触发器都是边沿触发的。

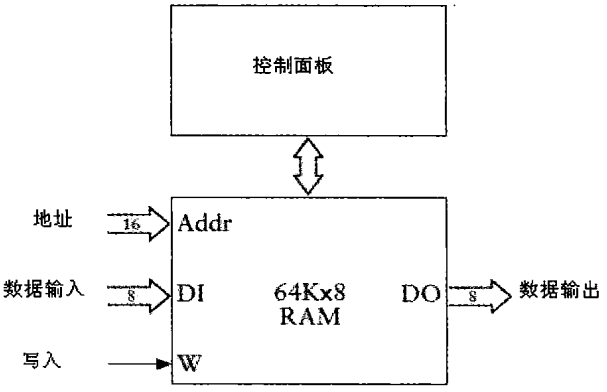
用来累加多个数的锁存器称做累加器（accumulator）。在本章的后面将会看到累加器

不仅仅做简单的累加，它还充当着锁存器的角色，保存第一个数，并且和下一个数做加法或减法运算。

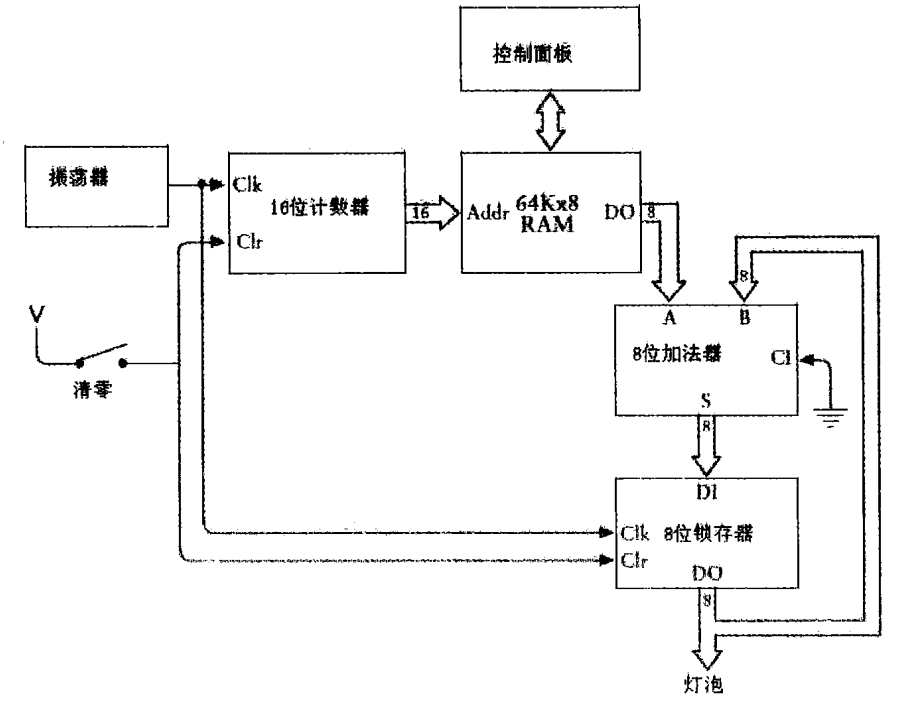
很显然，上面的加法器存在着一个很大的缺陷：假如要把 100 个二进制数加起来，你必须端坐于加法器前，并且耐心地输入所有的数并累加起来。但是当你终于完成时，却发现其中有两个数输错了，而你只能重复一遍所有的工作。

但是，也许并非如此。在前一章我们使用了大约 500 万个继电器构造了一个 64 KB 的 RAM 阵列。除此之外，我们还把一个控制面板连接到电路帮助我们工作，闭合它的控制（Takeover，有些书中也称“接管”）端开关后，就可以使用其他开关来控制 RAM 阵列的读写。下面是 64 KB RAM 阵列结构图。

如果把这 100 个二进制数输入到 RAM 阵列中而不是直接输入到加法器中，一旦需要修改一些数据，我们的工作将会变得容易得多。



因此我们所现在面临的挑战就是如何把 RAM 阵列和累加器连接起来。很显然，RAM 阵列的输出信号可以替代加法器的开关。而你也许想不到，用一个 16 位的计数器（比如我们在 14 章构造的那种）就可以控制 RAM 阵列的地址信号。在这个电路中，RAM 阵列的数据输入信号和写操作端信号可以省去。修改后的电路结构如下图所示。



当然，这并不是迄今发明的最易于使用的计算设备。要使用它，首先要闭合清零开关，这样做的目的是，清除锁存器中的内容并把 16 位计数器的输出置为 0000h。然后闭合 RAM 控制面板的控制端开关。现在你可以从地址 0000h 开始输入一组你想要相加的 8 位数。如果有 100 个数，那么它们将被存放在 0000h～0063h 的地址空间中（也应该把 RAM 阵列中未使用的单元设置为 00h）。然后闭合 RAM 控制面板的控制端开关（这样控制面板就不再控制 RAM 阵列了），同时断开清零开关。做完了这些，我们可以静静地坐下来，观察灯泡显示运算结果。

让我们来看一下它是怎样工作的：当清零开关第一次断开时，RAM 阵列的地址输入是 0000h。RAM 阵列的该地址中存放的 8 位数值是加法器的输入数据。加法器的另一个输入数据为 00h，因为此时锁存器也已经清零了振荡器提供的时钟信号——一个可以在 0，1 之间快速切换的信号。清零开关断开后，当时钟信号由 0 跳变为 1 时，将有两件事同时发生：锁存器保存加法器的计算结果，同时 16 位计数器增 1，指向 RAM 阵列的下一个地址单元。清零开关断开之后，时钟信号第一次从 0 跳变为 1 时，锁存器就将第一个数值保存下来，同时计数器增加为 0001h；当时钟发生第二次跳变时，锁存器保存之前两个数的求和结果，同时计数器增加

为0002h；按这种方式往复操作。

要注意的是，这里首先做了一些假设。最主要的一点就是，振荡器要足够慢以使电路的其他部分可以工作。每次时钟振荡的过程中，在加法器输出有效的结果之前，一些继电器必须去触发其他继电器。

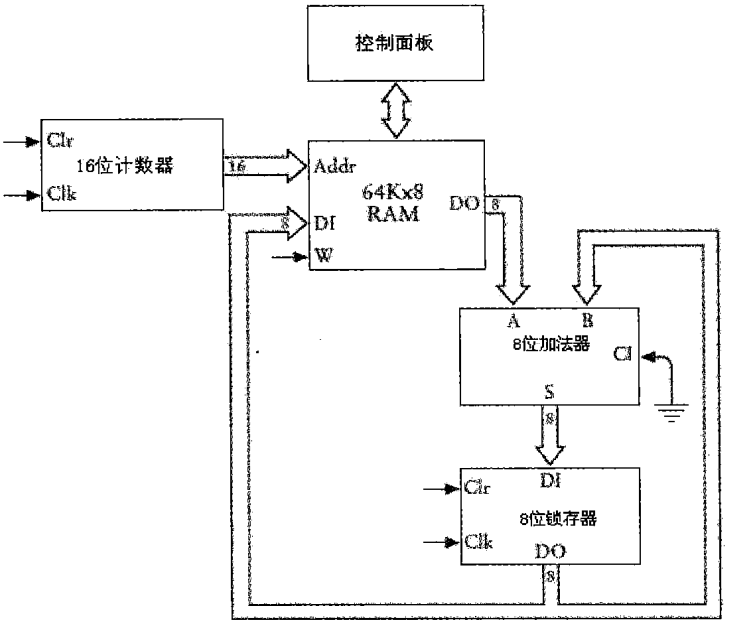
这个电路存在的一个缺陷是：我们没有办法使它停下来。在某一个时刻，所有灯泡会停止发光，因为RAM阵列的剩余部分存放的数都是00h。这时，你可以读取二进制的运算结果。但是当计数器达到FFFFh时，它会重新回滚（roll over）到0000h（这就好像汽车的里程表一样），这时自动加法器会再一次把所有的数累加到已经计算出来的结果中去。

这个加法器还存在另一个问题：它只能做加法运算，并且只能做8位数的加法。在这个RAM阵列中，不但每一个数要小于255，而且任意个数相加的结果也要小于255。此外，该加法器也不能处理减法运算，尽管可以用2的补数表示负数，但在这种情况下加法器能处理的数字的范围被限制在-128到127之间。要处理更大的数（例如，16位数）的话，一个简单的方法是：把RAM阵列、加法器、锁存器的位宽全都加倍，同时增加8个灯泡。但这些投资在我们看来是不合算的。

当然，这里提到这个问题的原因是最终我们要解决它。但首先来关注另一个问题，如果你不需要把100个数加在一起呢？如果你想做的是用自动加法器把50对数分别相加，得出50个不同的结果呢？或者你需要一种万能机，它可以方便地对两个数，10个数甚至100个数求和，并且所有的计算结果都可以很方便地使用。

先前提到的自动加法器都是用连接在锁存器上的灯泡来显示运行结果的，但是如果你想对50对数分别求和的时候，这就不是一个好的方法了。你可能会想到把运算结果存回到RAM阵列中去，这样的话，就可以在适当的时候用RAM阵列的控制面板来检查运算结果。为了实现这个目的，控制面板上专门设计了灯泡。

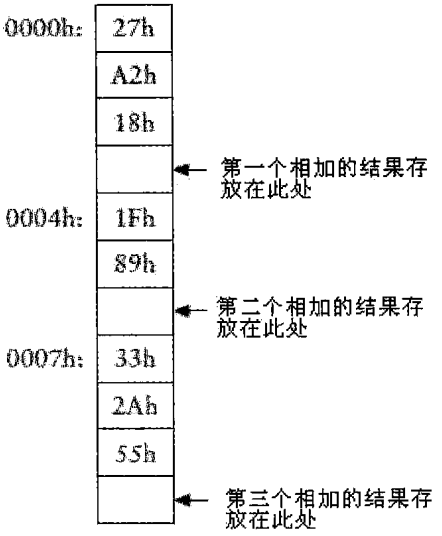
这意味着我们可以去掉与锁存器连接的灯泡，取而代之的是把锁存器的输出端连接到RAM阵列的数据输入端，这样就可以把计算结果写回到RAM阵列中去，如下图所示。



上图中略去了自动加法器的其他部分，其中包括振荡器和清零开关，这是因为我们不再需要特别标注计数器和锁存器的清零及时钟输入。此外，既然我们现在已经开始利用 RAM 的数据输入，因此需要一种用来控制 RAM 写入信号的方法。

现在我们不需要担心电路能否工作，而要把注意力集中到急需解决的问题上来。目前的当务之急是如何配置一个自动加法器，使它不仅仅可以对一组数字做累加运算，还希望它能够自主地确定要累加多少个数字，而且还能记住在 RAM 中存放了多少个计算结果，这样就可以简化查询工作。

例如，假设我们先要对三个数进行求和，然后对两个数进行求和，最后再对三个数进行求和。想象一下，我们可以把这些数保存在 RAM 阵列中以 0000h 开始的一组空间中，这些数存储在 RAM 阵列中的具体形式如下图所示。



本书中将用这样的形式表示一小段存储器。方格表示的是存储器的内容。存储器的每一个字节写在一个方格里。地址标记在方格的左边，并不是每一个地址都需要标记，因为地址是线性的，所以总是可以通过计算确定某个方格对应的地址。方格右边是关于该存储单元内容的注释，这些标记的单元就是我们想要自动加法器保存三个计算结果的位置（尽管这些方格画出来是空的，但是存储单元内并不是空的，它们总是保存着一些东西，就算只是一些随机数，但此时存放的是一些没有用的数）。

或许大家都有一种冲动，想亲自去做十六进制计算，并把结果填到那些小格子中去，但这并不是实验的目的，我们想要自动加法器为我们做这些加法。

我们并不希望自动加法器成为单任务系统——在它的第一个版本中，只是把 RAM 地址中的内容加到称为累加器的 8 位锁存器中——实际上我们希望它能做四件事：进行加法操作，首先它要把一个字节从存储器中传送到累加器中，这个操作称为加载（Load）。第二个操作把存储器中的一个字节加（Add）到累加器的内容中去。第三个操作把累加器中的计算结果取出并存放到存储器中。另外我们需要用一个方法令自动加法器停（Halt）下来。

我们借助具体的例子详细介绍这一过程，以上文提到的自动加法器所做的运算为例来说明。

（1）把 0000h 地址处的内容加载到累加器。

（2）把 0001h 地址处的内容加到累加器中。

（3）把 0002h 地址处的内容加到累加器中。

（4）把累加器中的内容存储到 0003h 地址处。

（5）把 0004h 地址处的内容加载到累加器。

（6）把 0005h 地址处的内容加到累加器。

（7）把累加器中的内容存储到 0006h 地址处。

（8）把 0007h 地址处的内容加载到累加器。

（9）把 0008h 地址处的内容加到累加器。

（10）把 0009h 地址处的内容加到累加器。

（11）把累加器中的内容存储到 000Ah 地址处。

（12）令自动加法器停止工作。

这里要注意，同最初的自动加法器一样，存储器中的每一个字节的地址仍然是以 0000h 为起点线性排列的。最初的加法器只是简单地把存储器指定地址的内容和累加器中的内容相加，在某些情况下需要这样做。但是有时我们需要把存储器中的某个值直接加载到累加器，或者把累加器中的值直接保存到存储器。做完了这些工作，算得上万事俱备只欠东风了，我们还希望自动加法器能方便地停下来，以便于查看 RAM 阵列中存放的值。

该如何来完成这些工作呢？能不能仅仅简单地向 RAM 阵列中输入一组数，然后期待自动加法器正确地完成所有工作呢？答案是否定的。对于 RAM 阵列中的每一个数，我们还需要用一些数字代码来标识加法器要做的每一项工作：加载、相加、保存和终止。

也许存放这些代码的最简单的方法（但肯定不是代价最小的）是把它们存放在一个独立的 RAM 阵列中。这个 RAM 应该和第一个 RAM 同时被访问。但是这个 RAM 中存放的是不需要求和的数，而是一些数字代码，用来标记自动加法器对第一个 RAM 中指定地址要做的一种操作。这两个 RAM 可以分别被标记为“数据”（第一个 RAM 阵列）和