

# Code: The Hidden Language of Computer Hardware and Software

---

- 透过现象进而探索本质可以发现新事物；将复杂的事物简单化，可以发现解决问题的新方法。
- 计算机体体系结构、汇编语言及数字电子技术
- 明喻与暗喻是文学描述中精妙的辅助手段，但它们常常掩盖了科学技术的真正光芒。计算机是二十世纪技术领域的“登峰造极之作”，它是一种值得欣赏、具有“美”学文化底蕴的人类伟大成果，这种“美”不需要明喻与暗喻的额外修饰。
- 存储器（storage）与内存（memory）有何区别？
  - 市场上考察个人计算机的存储性能，最主要的就是这两个概念。即便对于最初级的计算机用户来说，他们也一定需要了解到底多少“兆字节”或多少“吉字节”的存储器才能应对运行在其上的程序。如果进一步去思考，这些初级用户或许更加想了解计算机中的“文件”是什么概念，甚至连带这些文件如何从存储器加载进内存，又如何从内存存储到存储器，他们也非常期望学习这些知识。
  - 内存与存储器的区别其实是在逻辑层面上的，它体现着计算机体体系结构的实际需求与存储器客观性能之间的矛盾，简单地说就是我们找不到一种同时具备这两种存储器所有优点的存储媒介，这些优点就包括存储速度块、存储容量大、非易失性等等。今天的计算机都采用“冯·诺依曼体系结构”——五十年来它一直是计算机体体系结构的主导，而内存与存储器的区分也正是由于这种体系结构的不足所导致的。
- 计算机拥有与生俱来的层次化体系结构，这种结构的底层是晶体管，其顶层则是计算机显示器上所呈现的信息。自底向上分析该结构的每一层——这也是本书的编写结构——其实这一切并没有人们想象中那么难。当然，现代计算机的内部结构不断推陈出新，但其本质上仍然是一些常见且简捷的操作集合。
- 尽管今天的计算机比起25年前，以及50年前的都复杂许多，但它们在本质上是完全一致的。学习技术发展史的重要意义正在于此：追溯的历史越久远，技术的脉络就变得越清晰。因此，我们需要做的就是确定某些关键的历史阶段，在这些阶段，技术最天然、最本质的一面将清晰可见。

## 1 至亲密友

---

- 虽然莫尔斯电码和计算机毫无关系，但是，熟悉编码的本质对于深入理解计算机软硬件内部结构以及隐匿在其后的语言将大有裨益。
- 在这本书里，编码这个词的意思是指一种用来在机器和人之间传递信息的方式。换句话说，编码就是交流。
  - 大部分编码必须易于理解，因为它们是人类交流的基础。你可以说英语词汇就是一类编码。对任何能听见我们的声音并理解我们所说的语言的人来说，我们发出的声音所形成的词语是一种可识别的编码。我们将这个编码称为“口头话语（the spoken word）”或“言辞（speech）”。对于写在纸上（或刻在石头上、木头上，或者在空气中比划）的词，我们还有其他的编码方式。这种编码以手写字符或打印在报纸、杂志以及书本上的字符形式出现。我们叫它“书面语言（the written word）”或“文本（text）”。
  - 我们使用各种不同的编码来为我们自己的交流服务，因为有些编码有时比其他编码更便捷。如果一种编码可以用在其他编码无法取代的地方，那么它就是一种有用的编码。

- 各种类型的编码也用在计算机里来存储和传递数字、声音、音乐、图片和电影。计算机不能直接处理人类的编码，因为计算机无法通过与人类的眼睛、耳朵、嘴巴和手指完全相同的方式来接收人类发出的信息。然而，计算机技术的一个最新趋势，已使得我们的个人计算机能够获取、存储、处理和呈现一切用于与人类沟通的信息，无论视觉信息（文字和图片），还是听觉信息（口语、声音和音乐），或两者的相结合（动画和电影）。所有这些类型的信息都需要它们各自的编码，就像人类说话需要一套器官（嘴和耳朵）而写作和阅读需要另一套（手和眼）一样。
- 莫尔斯电码（Morse Code）使用“点（dot）”和“划（dash）”
  - 三个点、三个划，再加三个点就表示SOS，即国际求救信号。SOS不是一个缩写，这只是一个易于记忆的莫尔斯编码序列。
- 事实上，两个不同的事物，只要经过适当的组合，就可以表示所有类型的信息，这的确是千真万确的。

## 2 编码与组合

---

- 莫尔斯码其实是伴随着电报机的问世而被发明的，关于电报机，我们在后面也将做详细的探讨。正如通过研究莫尔斯码我们可以很方便地理解编码的本质一样，通过电报机来了解计算机硬件也是个不错的途径。
- 如果知道了码字中“点”和“划”的数目，那么以这个数目为指数的2的幂运算结果就是其总共可以表示的码字数。
  - 码字的数目 =  $2^{\text{“点”和“划”的数目}}$
- 码字的数目 =  $2^{\text{编码的位数}}$
- 莫尔斯码也被称作二进制码（Binary Code），因为这种编码的组成元素只有两个——“点”和“划”。

## 3 布莱叶盲文与二进制码

---

- 我们将6位二进制码（其实是6个点）所能表示的全部64种可能的编码都罗列了一遍。而且这64组编码中有很大一部分，根据上下文的不同将有着双重身份。
  - 尤其值得注意的是数字标识符和取消“数字标识状态”的字母标识符。它们改变了后面编码的意义——从表示字母到表示数字，又从表示数字回到表示字母。像这样的编码通常被称作“优先码”（precedence codes）或者“换档码”（shift codes）。它们改变着作用域内编码的含义，直到作用域结束。
  - 大写字母标识符表示紧随它的字母（而且仅仅是紧随它的字母）应该被译为大写。类似这样的编码被称为“逃逸码”（escape codes）。逃逸码让你“逃离”对编码串单调的、一成不变的解析，而转入一种新的解析方式中。在以后的章节中我们将看到，在使用二进制码对书面语言进行编码时，换档码和逃逸码是相当常见的。

## 4 手电筒的剖析

---

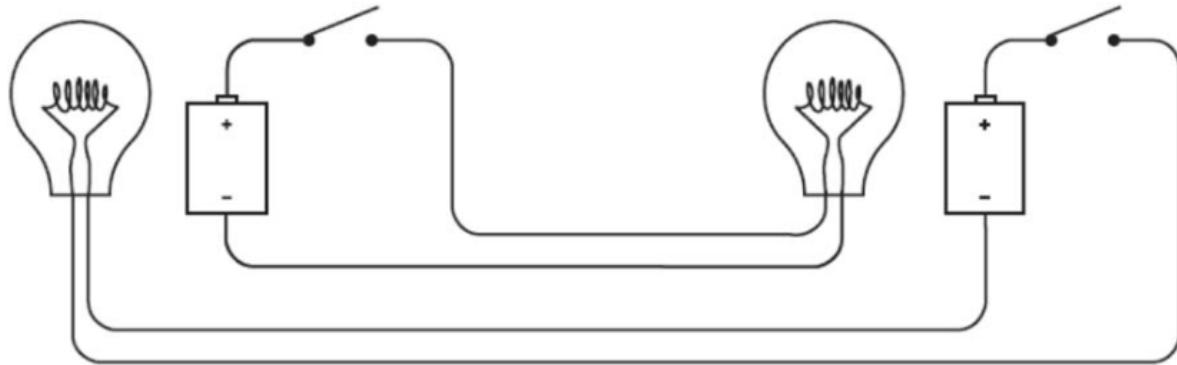
- 一个原子中电子的数目一般情况下与质子数目相同。但是在某些情况下，电子可能从原子中脱离。这就是电流产生的原因。
  - 当质子和电子在相同数目的条件下共存时，它们都处于最和谐、最稳定的状态。如果质子和电子之间出现失衡现象，它们就会试图进行自我修复。
  - 电路中，某原子所含有的一电子逃逸到它相邻的下一个原子中，与此同时，这个原子又从相邻的上一个原子中获取一个电子，而失去电子的原子又会从与其相邻的一个原子获得电子，如此循环。电路中的电子不断地从一个原子移动到下一个原子，就形成了电流。

- 所有电池的内部都会发生化学反应，也就是说一些分子被分裂形成其他的分子，或者分子间互相结合形成了新分子。电池内的化学物质是经过研究精心选择的，它们之间的化学反应能够使多余的自由电子聚集到标负号“-”的那端（称为负极或者阴极），而在标有正号“+”的那端（称为正极或者阳极）则变得急需额外的电子。于是，化学能就被转化成了电能。
- 从电池的负极到正极，电子流经了导线和灯泡。但是为什么我们需要导线呢？电流不可以直接被空气传导吗？哦，可以说能，也可以说不能。电流可以通过空气传导（特别是潮湿的空气），不然的话我们就看不到闪电了。但是电流也不能轻易地穿过空气。
  - 如果原子在最外电子层中只含有1个电子，那么这个电子很容易逃逸，这就是易导电物质所需具备的特性。这些物质对于电流来说是“导通”的，因而被称做导体（conductor）。最好的导体是铜、银和金。这三种元素位于元素周期表的同一列不是巧合。其中铜是用来制作导线的最常见的原料。
  - 与导电性相反的是阻抗性。有一些物质与其他的物质相比更容易让电流通过，我们称其为电阻。如果一种物质有着很强的阻抗性——也就是说它几乎不能传导任何电流——它就被称为绝缘体（insulator）。橡胶和塑料都是很好的绝缘体，因而它们经常被用来包裹金属导线。布料和木头在干燥的空气中也是很好的绝缘体。不过事实上只要有足够高的电压，任何的物质都是可以导电的。
  - 铜具有很低的阻抗，但实际上或多或少仍然会存在一点。导线越长，它的阻抗就越高。导线越粗，它的阻抗就越低。粗一些的导线可以使更多的电子顺畅地通过线路。
- 电压表征了电流做功的“势”（potential），也就是电势能的大小。不管电池是否被连接到电路中，电压都是存在的。电流与流经电路的电子数有关。它的计量单位是安培。
- 在电学中，如果你知道了电压和电阻，就可以计算出电路中的电流是多少。电阻——一般来说物质都倾向于阻拦电子的通过——的单位是欧姆。欧姆定律： $I = E / R$ 。其中， $I$ 用来表示电路中的电流， $E$ 用来表示电压（它代表电动势）， $R$ 表示电阻。
- 如果导线电阻较低的话，它将变热并且发光。这就是白炽灯发光的原理。如果暴露在空气中，钨丝将达到燃点并开始燃烧，但是在灯泡的真空泡室内，钨丝就会发出光亮。
  - 瓦特是功率（P）的计量单位，它的计算公式如下： $P = E \times I$
- 当开关被设置成允许电流通过时，我们称它的状态为“开”，或者“闭合”。当处在“关”或者“断开”状态时，开关不允许电流通过。（“闭合”和“断开”这两个词，在用来描述开关时，其含义与用来描述房门时恰好是相反的。一扇闭合的门会阻止所有试图穿过它的东西；一个闭合的开关却可以使电流导通。）
- 开关只能是闭合状态或断开状态。电流只能是有或者无。灯泡也只能是发光或不发光。就像莫尔斯和布莱叶发明的二进制码一样，这个简单的手电筒要么是开着的，要么是关着的。没有介于二者之间的状态。在后面的章节中，二进制码与电气电路之间的这种相似性将起到很大作用。

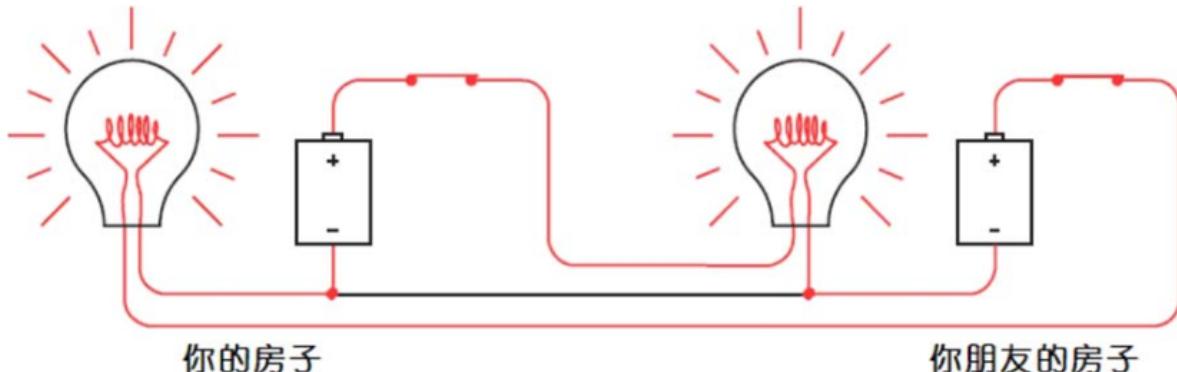
## 5 绕过拐角的通信

---

- 一套双向电报系统



- 采用公用连接 (common) , 节省25%的导线。



- 如果整个系统必须铺设很长的线路, 可能还得绞尽脑汁再削减一根导线以进一步节约开支。

- 你不必非得用导线来完成电路的共用部分, 你可以用其他的东西来代替导线。恰巧我们这儿有一个现成的大球, 你可以用它来代替。这个大球直径近7900英里, 由金属、岩石、水, 以及有机质(其中大部分是没有生命的)组成。我们称这个巨大球体为“地球”。

在讲导体的时候提到了银、铜和金, 但没有提过岩石层和覆盖层。实际上, 泥土并不是一个很好的导体, 尽管有一些泥土(例如沼泽)的导电性比其他的(例如干沙)要好一些。但是我们知道导体有一条性质: 截面越大导电性越好。一条很粗的导线, 其导电性要远远好于一条细导线。这就是地球所拥有的优势。它实在是太大了。

要想用地球充当导体, 可不是随便在西红柿地里插根线那么简单。你必须使用跟地球有充分接触的物体, 也就是有很大表面积的导体。这儿有个不错的解决方案, 即使用一个至少8英尺长、 $1/2$ 英寸粗的铜柱电极。它提供了与地球150平方英寸的接触面积。你可以用一个大锤子把这个电极砸进地里, 然后在上面接上一根导线。或者, 如果你家的水管是用铜做的, 并且是从屋外的地下接过来的, 那么你可以在水管上接上导线。

- 如果你使用的是高压电池和大灯泡, 你只需要在你和你朋友的房子之间接一根导线, 因为可以把地球当成一条导线。



你的房子

你朋友的房子

电子从你朋友房子的地下出发，经过灯泡、导线和你房间的开关，最后回到电池的正极。而电子最初是从电池的负极传入地下的。电子怎么知道它要去什么地方？其实它们不知道。换一种描述地球的模式或许更恰当些。地球是一个巨大的导体，但是我们也可以把它看做是电子的来源和储藏库。地球之于电子就恰如海洋之于水滴。地球是一个近乎无尽的电子之源，同时也是一个无比庞大的电子池。

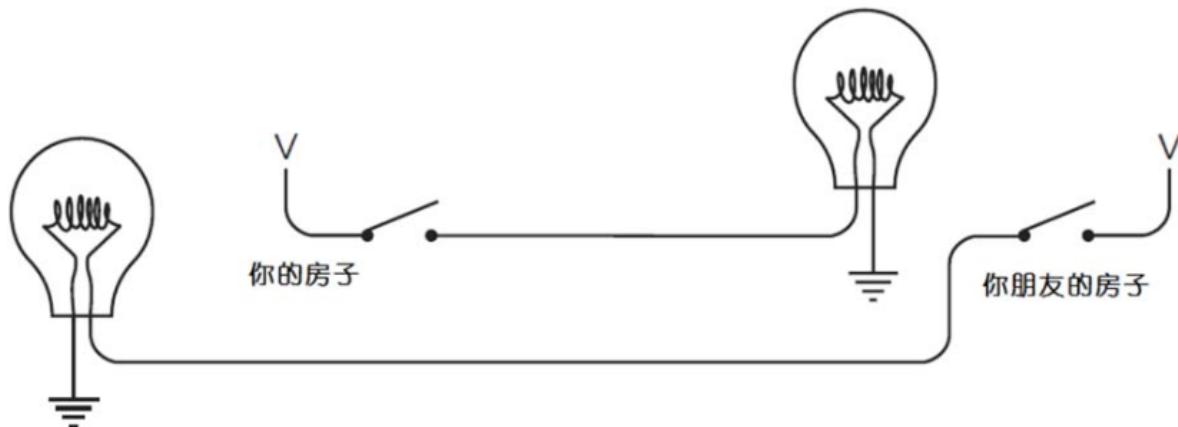
- 不过地球还是有一定的电阻的。所以当我们使用1.5伏干电池和小灯泡时，不能通过接地来节约我们所需的线路开支。对于低电压电池而言，地球的电阻实在是太高了。
- 字母V是电压的意思，但是它也有吸尘器的意思。我们把V想象成一个电子吸尘器，然后把地面想象成电子的海洋。电子吸尘器通过电路把电子从地下拉出来，让它们沿设计好的线路开始工作（例如点亮灯泡）。



你的房子

你朋友的房子

- 大地有时也被认为是零电势 (zero potential) 点。
- 只用两根导线构建一个双向的莫尔斯码发送系统。



- 之前我们使用莫尔斯码交流时，必须要在视线直视的范围里，并且要保证在手电筒光线可以传播的距离之内。

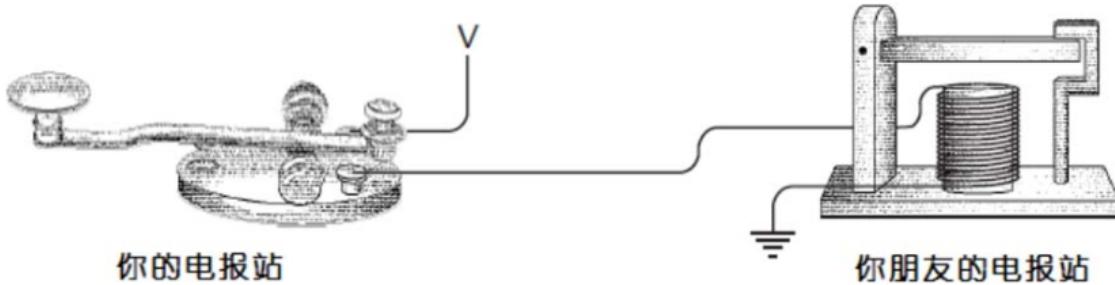
但是使用导线，我们不仅可以构建出一个可以绕过拐角的、能够在视线之外的发报系统，而且我们自己再也无需受距离的限制。我们可以跨越成百上千英里来进行通信，只需要铺设足够长的线路即可。

- 尽管铜是很好的导电体，但是它也不是完美的。线路越长，它们的电阻也就越大。电阻越大，线路中的电流就越少。电流越少，灯泡就越暗。使用粗一点的导线是一个不错的解决方案，但是那会比较昂贵。另一种解决方案是增加电压，使用更大电阻的灯泡，减小导线的电阻对整个电路的影响。

- 在150年前，人们在铺设第一个跨越美洲和欧洲的电报系统时，这些都是面临的问题。根据设计方案，系统距离跨度的极限是200英里。这个长度与纽约和加利福尼亚间数千英里的距离相比，还是有很大差距的。
- 这个问题的解决办法存在于近代电报系统中。尽管近代电报系统只不过是个很简陋的装置，但正是基于这个装置，整个计算机系统才被构建出来。

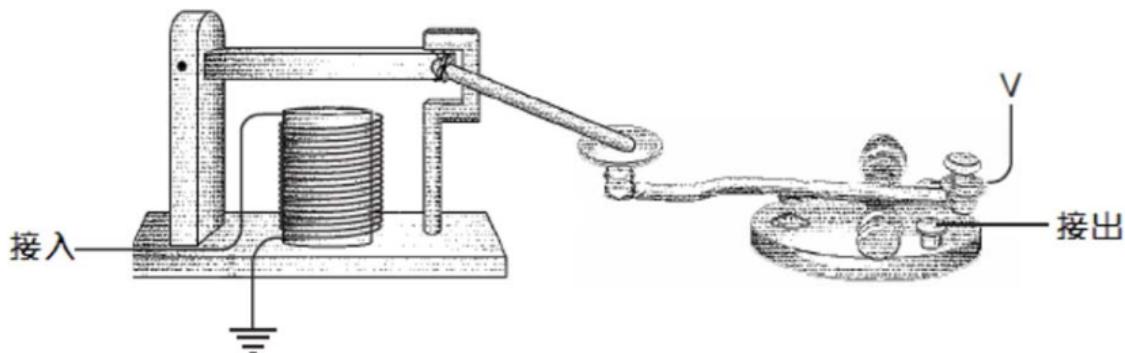
## 6 电报机与继电器

- 全球性即时通信对于我们来说已经司空见惯了，这项技术其实是在近代才得到发展的。回溯到19世纪早期，你也可以进行即时通信或者远距离通信，但是不能同时做到这两点。即时通信受声音传播距离的限制（没有扩音器可以用），或者受视野的限制（或许可以使用望远镜作为辅助工具）。使用信件可以进行更远距离的通信，但是寄信耗时的时间太多，并且需要马匹、火车或轮船。
- 电报机的原理是很简单的：在线路的这一端采取一些措施，使线路的另一端发生某种变化。
  - 电磁铁是电报机的基础。在线路的一端闭合或断开开关，可以使线路另一端的电磁铁有所动作。

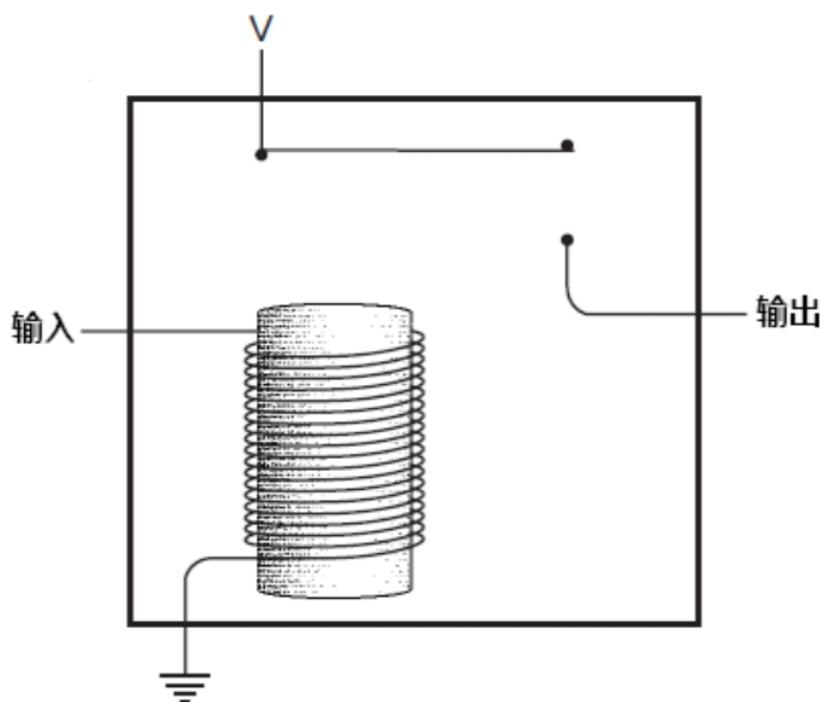


- 当电报机的电键被按下时，发声器中的电磁铁拉动上面的活动横杠下降，它会发出“滴”的声音。当松开电键的时候，横杠弹回到原来的位置，发出“嗒”的声音。一次快速的“滴-嗒”声代表点；一次慢速的“滴——嗒”声则代表划。
- 电报机的发明标志着现代通信的开始。人们第一次能够在视线或者听力之外的距离范围进行实时交流了，而且信息传递的速度比骏马疾驰还要快。而更加耐人寻味的是，这个发明使用了二进制码。但是在后来的电子和无线通信（包括电话、无线电、电视）所使用的通信模式中，二进制码被废弃了，直到后来它又被应用在了电脑、光盘、数字影碟、数字卫星电视广播和高清电视上。
- 这种电报机最大的问题就是长导线所带来的电阻。尽管一些电报线路使用高达300伏的电压，而使有效距离能够超过300英里，但是线路还是不能无限延长。
  - 显然，设置一个中继系统是解决该问题的一个方案。每隔200英里左右，为一个工作人员装配好发声器和电键，他就可以接收信息，然后再把它转发出去。
  - 开始时，他喜欢接收完一条完整的信息后再把它转发。首先，根据发声器发出的滴答声，将字母记下来；当信息接收完毕时，再开始用他的电键来发送。最后，他终于掌握了诀窍，在听到滴答声的同时就可以发送信息，不需要再把信息记录下来了。这节约了不少时间。

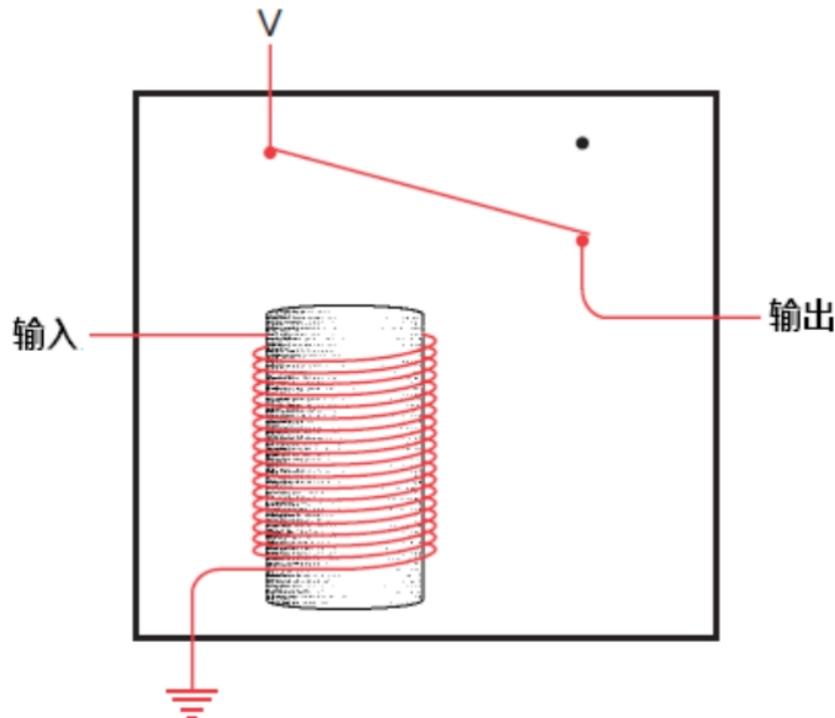
- 有一天，他恍然发现发声器上下跳跃的节奏与电键是一致的。因此他就去外面找了根小木棍，然后用木棍和一些细绳把发声器和电键连接到了一起，如下图所示。现在，设备可以自己工作了。



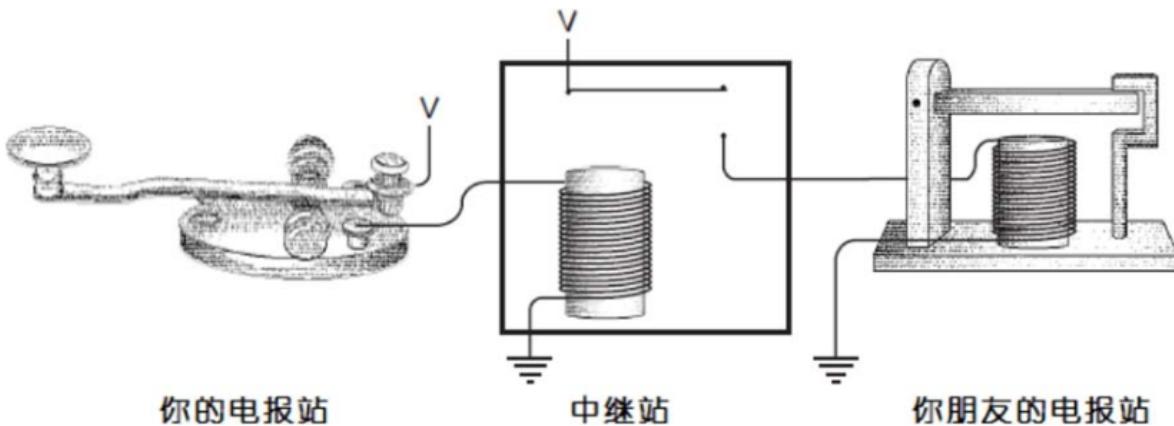
- 刚刚我们发明的这个设备称做“继电器”。继电器与发声器很像，传进来的电流驱动电磁铁拉动金属杠，金属杠同时又作为一个开关的组成部分，而这个开关连接着电池和输出线路。通过这种方法，输入的比较弱的电流就被“放大”成了较强的输出电流。



- 当输入的电流触发了电磁铁，电磁铁把一个弹性金属条吸附下来，就像闭合了开关一样，使电流可以从接口输出。



- 因此，把一个电报机电键、一个继电器，还有一个发声器连接好，差不多就是下图的这个样子。



- 继电器是一个意义非凡的设备。当然，它是一个开关，但是这个开关的闭合和断开并不是由人来操纵的，而是由电流控制的。
  - 你可以通过它来完成一些不可思议的事情。实际上，使用它，你甚至可以装配好一台近乎完整的计算机来！
  - 不过我们在使用它之前，还要学会如何计数。

## 7 我们的十个数字

- 数字当然是我们平常所能接触到的一种最抽象的编码。
- 大多数文明都是建立在以10为基数的数字系统上的（有的时候是以5为基数），这种情况并不奇怪。最开始，人们用自己的手指来计数。如果我们人类有8个或12个手指，那么我们的计数方式就会和现在有所不同。英语中Digit（数字）这个词同时也有手指、脚趾的意思，并且还有数字的意思，这并不是巧合。而five（五）和fist（拳头）这两个单词的拥有相同的词根也是同样的道理。
  - 在这个意义上，以10为基数或使用十进制数字系统完全是随意的。
- 大多数历史学家认为数字最初起源于对事物的计数，例如：人数、财产或商业交易的计数等。所有早期的数字系统中，只有罗马数字沿用到了今天。

- 沿用到今天的罗马数字符号有: I V X L C D M
- 字母I表示1, 可以看做是一个划线或者一根伸出的手指。字母V像一只手, 表示5。两个V是一个X, 代表数字10。L是50。C来自单词centum, 表示100。D是500。最后一个, M来自于拉丁文mille, 意为1000。
- 在很长一段时间内, 罗马数字被人们看做是易于加减的, 这也是为什么罗马数字在欧洲作记账之用一直沿用到今天。但是用罗马数字进行乘法和除法却很复杂。
  - 很多其他早期数字系统(像古希腊数字系统)和罗马数字系统相似, 它们在用于复杂运算方面同样也存在一定的不足。尽管古希腊人发明的非凡的几何学至今仍然是高中生的一门课程, 但古希腊人并不是以代数而著称的。
  - 如今我们所用的数字系统通常被称为阿拉伯数字, 也可以称为印度-阿拉伯数字系统。它起源于印度, 被阿拉伯数学家带入欧洲。其中最著名的就是波斯数学家穆罕默德·伊本穆萨·奥瑞兹穆(根据这个人的名字衍生出英文单词“algorithm”, 算法)
- 阿拉伯数字系统不同于先前的数字系统, 体现在以下三点。
  - 阿拉伯数字系统是和位置相关的。也就是说, 一个数字的位置不同, 其代表数量也不同。
    - 位置计数系统的好处并不在于它有多么好用, 而在于对非十进制的系统而言, 它仍然是易于实现计数的。
  - 实际上在早期的数字系统中也有一点是阿拉伯数字系统所没有的, 那就是用来表示数字10的专门的符号。而在我们现在使用的数字系统中是没有代表10的专门符号的。
  - 另一方面, 实际上阿拉伯数字也有一点是几乎所有早期数字系统所没有的, 而这恰恰是一个比代表数字10的符号还重要得多的符号, 那就是0。
    - 是的, 就是0。小小的一个零无疑是数字和数学史上最重要的发明之一。它支持位置计数法, 因此可以将25、205和250区分开来。0也简化了与位置无关的数字系统中的一些非常复杂的运算, 尤其是乘法和除法。

## 8 十的替代品

---

- 当我们认识到, 10可以表示两只鸭子, 我们就可以解释开关、导线、灯泡和继电器(进一步推广到计算机)是如何表示数字的了。
- 八进制数字系统与十进制数字系统并没有什么不同。它们只是在细节上存在一些差异。例如, 八进制数中的每个位所代表的值是该位数字乘以8的整数次幂的结果。
- 在十进制中, 好整数通常是指结尾有若干个零的数。在结尾有两个零的十进制数代表的是100TEN, 而100TEN表示10TEN乘以10TEN。在八进制数中, 结尾有两个零代表是100EIGHT, 而100EIGHT表示10EIGHT乘以10EIGHT(或8TEN乘以8TEN, 即64TEN)。
  - 这些好整数100EIGHT、200EIGHT和400EIGHT在十进制中分别与64TEN、128TEN和256TEN相等, 它们全是2的整数次幂。这是非常有意义的。例如, 400EIGHT等于4EIGHT乘以10EIGHT乘以10EIGHT, 而这里所有的数都是2的整数次幂。任何2的整数次幂乘以2的整数次幂的结果依然是2的整数次幂。
  - 八进制好整数暗示我们, 十进制以外的数字系统可能对使用二进制码有所帮助。
- 一个八进制数3725EIGHT可以分解成如下形式:
  - $3725_{\text{EIGHT}} = 3000_{\text{EIGHT}} + 700_{\text{EIGHT}} + 20_{\text{EIGHT}} + 5_{\text{EIGHT}}$
  - $3725_{\text{EIGHT}} = 3 \times 1000_{\text{EIGHT}} + 7 \times 100_{\text{EIGHT}} + 2 \times 10_{\text{EIGHT}} + 5 \times 1$

- $3725_{EIGHT} = 3 \times 512_{TEN} + 7 \times 64_{TEN} + 2 \times 8_{TEN} + 5 \times 1$
- $3725_{EIGHT} = 3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 5 \times 8^0$
- 二进制数的长度增长得很快而非数值增大得快
  - 任何一个以1开头而后面全是0的二进制数一定都是2的整数次幂。幂指数就等于这个二进制数中0的个数。

2的整数次幂	十进制数	八进制数	四进制数	二进制数
$2^0$	1	1	1	1
$2^1$	2	2	2	10
$2^2$	4	4	10	100
$2^3$	8	10	20	1000
$2^4$	16	20	100	10000
$2^5$	32	40	200	100000
$2^6$	64	100	1000	1000000
$2^7$	128	200	2000	10000000
$2^8$	256	400	10000	100000000
$2^9$	512	1000	20000	1000000000
$2^{10}$	1024	2000	100000	10000000000
$2^{11}$	2048	4000	200000	100000000000
$2^{12}$	4096	10000	1000000	1000000000000

- 二进制数转换为十进制数

$$\begin{array}{cccccccc}
 \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{0} \\
 \times 128 & \times 64 & \times 32 & \times 16 & \times 8 & \times 4 & \times 2 & \times 1 \\
 \boxed{128} + \boxed{0} + \boxed{0} + \boxed{16} + \boxed{0} + \boxed{4} + \boxed{2} + \boxed{0} = \boxed{150}
 \end{array}$$

- 十进制数转换为二进制数

$$\begin{array}{cccccccc}
 \boxed{150} & \boxed{22} & \boxed{22} & \boxed{22} & \boxed{6} & \boxed{6} & \boxed{2} & \boxed{0} \\
 \div 128 & \div 64 & \div 32 & \div 16 & \div 8 & \div 4 & \div 2 & \div 1 \\
 \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{0}
 \end{array}$$

- 将整个十进制数（小于或等于255）填入到上面一行最左端的格子中。用第一个除数（128）去除这个数。所得的商填入正下方的格子（左下角的格子），余数填入右边的格子（上面一行左数第二个格子）。用第一个余数再除以下一个除数64。依照模板的顺序用同样的方法继续进行下去。每次求得的商只能是0或者1。如果被除数小于除数，商为0，余数就是被除数。如果被除数大于或等于除数，那么商为1，余数就是被除数减去除数所得之差。
- 人们在使用二进制数的时候通常将它们写成带有前导零的形式（即第一个1的左边有零）。这样写不会改变数字的大小，仅仅是为了美观。

二进制数	十进制数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

- 仔细考虑这4个垂直列中每一列的1和0，注意它们在一列中自上而下是以怎样的规律交替的。
  - 最右边的一列一直在0和1之间交替。
  - 右数第二列是在每两个0和两个1之间相互交替。
  - 下一列是在每四个0和每四个1之间相互交替。
  - 再下一列是在每八个0和每八个1之间相互交替。
- 每次只要有一个二进制数位的值由1变到0，紧挨着的高位数字也会发生变化，而且其变化不是由0到1就是由1到0。
- 为了让二进制数更易读，通常在每四个数字之间用一个连字符来分开，例如，1011-0111-0001-1011-0000-0000，或者每四位空出一个空格：1011 0111 0001 1011 0000 0000。

- 不可能找到比只有0和1两个数字的二进制数字系统更简单的数字系统了。
- 二进制数字系统在算术与电子技术之间架起了一座桥梁。开关、电线、灯泡，还有继电器等物体，都可以用来表示二进制数0和1。二进制数与计算机之间有着紧密的联系。

## 9 二进制数

---

- 十进制与其他数字系统相比并没有什么不同，只是我们通常使用它来计数。很明显，我们使用十进制数字系统是因为我们有十个手指。如果我们将数字系统建成八进制数字系统（如果我们是卡通人物）或四进制数字系统（如果我们是龙虾），甚至二进制数字系统（如果我们是海豚），也是合乎情理的。
- 但是二进制数字系统存在一点特殊性。这个特殊性就在于它是人们所能得到的最简单的数字系统。
- 单词“bit（比特）”被创造出来代表英文“binary digit”
  - 这个词也有它一般的意义：“一小部分，程度很低或数量很少”。这个意义很贴切，因为1比特——一个二进制数位——确实是一个非常小的量。
- 在计算机时代，比特被看做是组成信息块的基本单位。
  - 二进制数不是传达信息的唯一方法。字母、单词、莫尔斯码、布莱叶盲文和十进制数都可以来表达信息。关键在于，比特所传递的信息量极少。1比特是可能存在的最小的信息量。任何小于1比特的内容都根本算不上是信息。
- 由于1比特表示的是可能存在的最小信息量，那么复杂一些的信息就可以用多位二进制数来表达（比特传递的信息量“小”，并不是说它传送的信息不重要）
- 两盏提灯在一起可以表达四种可能的信息：
  - 00 = 英军今晚不会入侵
  - 01 \ 10 = 英军正由陆路入侵
  - 11 = 英军正由海路入侵
  - 保罗·里威尔将三种可能性用两盏提灯来传送
    - 通信理论中，噪声（noise）是指影响通信效果的所有事物。电话线上的静电就是一个影响电话通信的鲜明例子。然而，电话通信通常都能成功，因为即使存在噪声，语音中仍存在大量的冗余。我们要理解对方的意思，并不需要听清对方所说的每个词的每个音节。
    - 在上述例子中，噪声是指黑夜中暗淡的光线以及保罗·里威尔距钟楼的距离，这两个因素都能让他分辨不出点亮的是哪盏提灯。
  - 用通信理论的术语来说，他运用了“冗余”（redundancy）来抵消噪声的影响。
- 信息是指多个可能性中的一种。例如，当我们与一个人交谈的时候，我们所说的每个字都是对字典中所有字的一个选择。
- 换句话说，所有可以被转换成对两种或多种可能性的选择的信息，都可以用比特来表示。
  - 不用说，人类有很多形式的交流是不能用对非此即彼的可能性的选择来表示的，但这些交流形式对我们人类的生存又是至关重要的。这就是人类为什么没有与计算机之间建立起浪漫关系的原因（无论如何，我们都希望这种情况不会发生）。
  - 如果你无法用语言、图画或声音来表达某种事物的时候，你就也无法将这个信息用比特的形式来编码。当然，你也不会想去这么做。
- 某一位或一连串比特位所表示的意义通常是和上下文有关的。
- 利用二进制表示信息的一个额外的好处就是我们可以清楚地知道我们是否已经想到了所有的可能性。

- 无论何时我们谈到比特，通常所指的都是一定数目的比特位。我们拥有的比特位数越多，所能表示的不同可能性就越多。

- 在十进制中也是同样的道理。
- 在二进制中，可能有的编码数等于2的整数次幂，其幂指数就是比特位的位数。

比特位数	编码数量
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

- 每增加一个比特位就会将编码的数量增加一倍。
- 最常见的二进制数的表现形式就是无所不在的通用产品代码（UPC，Universal Product Code）
  - 在UPC中，黑色条纹有四种不同的宽度，宽条纹的宽度分别是最细条纹宽度的两倍、三倍或四倍。同样，宽间隔的宽度分别是最窄间隔宽度的两倍、三倍或四倍。
  - 扫描仪只识别整个条形码的一条窄带，条形码做得很大是为了便于结算台的操作人员用扫描仪对准。
  - 有部分冗余对于检错来讲是必要的。
- 如果一组比特位中含有偶数个1，它就称为偶校验；如果含有奇数个1，那么它就称为奇校验。
- 比特可以表示莫尔斯码、布莱叶盲文
- 比特可以表示文字、图片、声音、音乐、电影，也可以表示产品编码、胶片速度、影评结果、英国军队的入侵，以及心爱之人的意图。但是，从根本上说，比特是数字。在用比特表示其他信息的时候我们所要做的就是计算有多少种可能性。这决定了我们需要的比特位数，以便每种可能性都可以分配到一个编号。

- 比特在逻辑学中也很重要。逻辑学是哲学和数学的奇特融合，其主要目的就是确定某个陈述是真还是假。真和假同样可以表示为1和0。

## 10 逻辑与开关

- 什么是真理？亚里士多德认为真理是与逻辑相关的某种事物。对于古希腊人来说，逻辑是在追求真理的过程中的一种分析方法，因此它被人们认为是一种哲学形式。
- 布尔发明了一种代数，这种代数看上去与传统代数非常相似，而且运算规则也非常类似。
  - 在传统的代数中，操作数（通常为字母）代表数字，算子（通常为“+”和“×”）则用来指示这些数字之间如何运算。
  - 传统代数的一个特点就是，它是处理数字的，例如，豆腐的重量、鸭子的数量、火车行驶的距离或家庭成员的年龄。
  - 布尔的天才之处就在于他把代数从数的概念中抽离出来而使其更加抽象。在布尔代数（Boole's algebra）中，操作数不是数字而是类（class）。简单说，一个类就是一个事物的群体，它后来也被称为集合（set）。
  - 在布尔代数中，也有“+”和“×”这样的符号，这很可能造成混淆。所有人都知道在传统代数中如何将数字相加或相乘，但是我们如何将类相加或相乘呢？
    - 在布尔代数中，符号“+”表示两个集合的并集。
    - 在布尔代数中，符号“×”表示两个集合的交集。
    - 为了避免在传统代数与布尔代数间混淆，有时用符号“U”和“∩”来代替“+”和“×”。布尔对数学方面的革命性影响是让人们熟知的符号更加抽象，因此，我决定坚持他的做法，而不是引入新的符号到他的代数中。
- 布尔代数中还有另外两个符号是非常重要的。这两个符号看起来像数字，但是它们并不是真正意义上的数字，相对于数字而言，它们有不同的意义。在布尔代数中，符号1表示“全集”——也就是我们所提到的所有事物。在布尔代数中，符号0表示空集——不包含任何元素的集合。
- 你大概会去抛弃并集和交集的概念，并用OR和AND取而代之。这里将字母大写是因为，它们表示的不仅是字面意义，而表示布尔代数中的运算。
  - 符号“+”（之前作为并集的符号）现在可以用OR来表示。
  - 符号“×”（之前作为交集的符号）现在可以用AND来表示。
  - 符号“1-”（之前意思是从全集中去掉某些元素）现在用NOT来表示。
- 你想要的猫是在以下这样的集合里： $(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$ 
  - 为了避免麻烦，我采用了一种略微有点不同的布尔代数形式——字母不仅仅代表集合。这里，字母可以用数字来赋值。我们只用数字0和1。数字1代表YES, True，即这只猫是符合这样的标准的。数字0表示NO, False，即这只猫不符合这种特定标准。
  - 现在我们要做的就是化简这个表达式。如果简化结果为1，则这只猫就符合你的标准；如果简化结果为0，那么这只猫就不符合。在化简的时候要切记，我们并不是真的在进行加和乘的运算，尽管通常我们可以当做是。符号“+”代表OR，符号“×”代表AND，在大多同类的规则中适用（有时在现代课本中也使用符号“^”和“v”来表示AND和OR，而非用符号“×”和“+”。但是在里符号“+”和“×”是具有特殊意义的）。
- 可以通过连通开关和灯泡的方法来确定某类猫咪是否符合你的标准
  - 两个开关串联表示逻辑AND（用符号“×”表示）

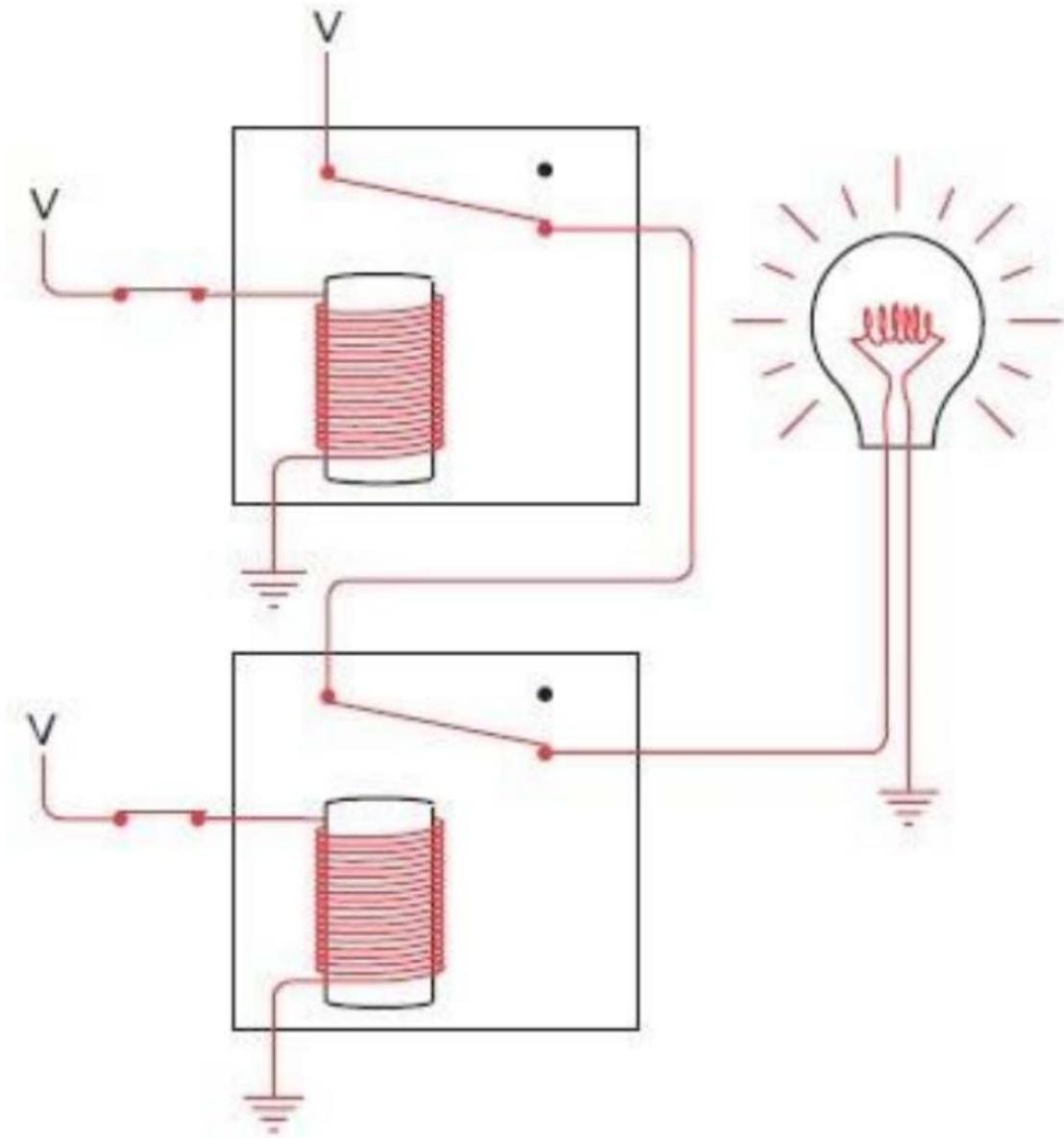
- 两个开关并联表示逻辑OR（用符号“+”表示）
- 将布尔代数与电路相融合，设计和制造利用二进制进行计算的计算机。
- 塞缪尔·莫尔斯在1844年论证了他的电报机是可行的——早于布尔发表《思维规律的研究》10年——将电报发声器替换成上述电路中的灯泡本应该是非常简单的。然而，在19世纪，没有人将布尔代数中的AND和OR同线路中的开关串联及并联关联到一起。没有这样的数学家，没有这样的电学家，没有这样的电报员，没有这样的人。

## 11 门

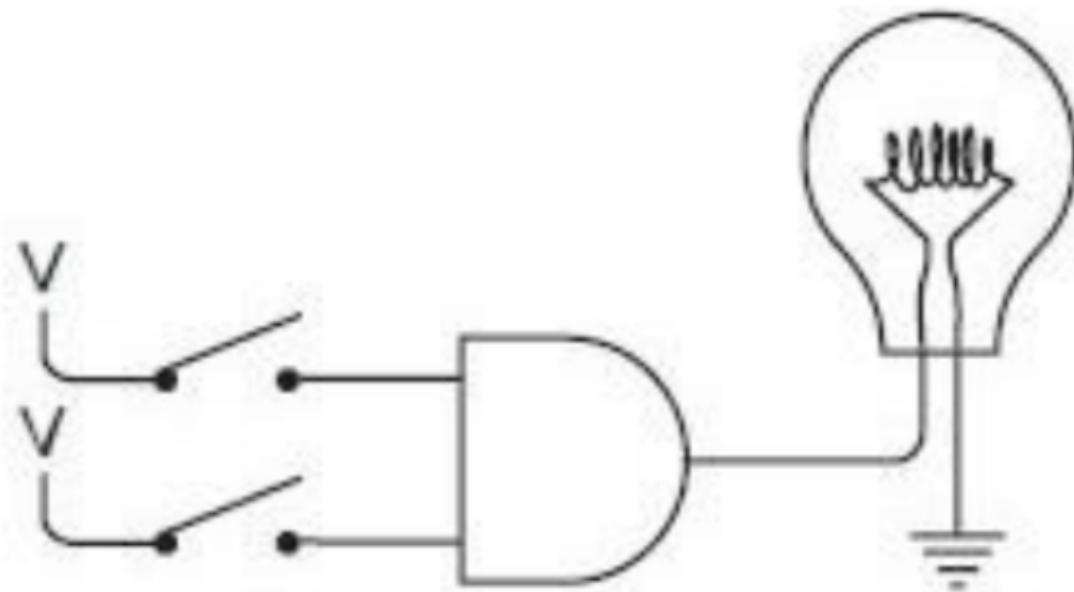
---

- 在逻辑学中，逻辑门的工作方式非常简单——让电流通过或阻止电流通过。
- 布尔表达式可以用一个由开关和灯泡组成的电路来表示，这样的电路有时被称为网络（network），而如今这个词更多地被用来描述计算机之间的连接，而不仅指多个开关的集合。
- 尽管这个电路中的所有元件早在19世纪就都已经被发明出来了，但在那个时代，没有人意识到布尔表达式可以在电路中实现。
  - 这个等价关系直到20世纪30年代才被发现。主要贡献人是克洛德·艾尔伍德·香农（生于1916）。1938年，香农在麻省理工学院完成了那篇题为《继电器和开关电路的符号分析》（A Symbolic Analysis of Relay and Switching Circuits）的著名硕士论文，在文中阐述了这个问题（10年之后，他又发表了论文“通信的数学原理”，即The Mathematical Theory of Communication，这是第一篇使用“bit”这个词来表示二进制数字的文章）。
- 电子工程师可以运用布尔代数的所有工具去设计开关电路。此外，如果你简化了一个描述网络的布尔表达式，那么你也可以简化相应的网络。
- 在计算机术语中，开关是一种输入设备（input device），输入是控制电路如何工作的信息。
  - 在本例中，输入开关对应于4个二进制数信息，这些信息用来描述一只猫。输出设备（output device）就是灯泡。
- 继电器像开关一样，可以串联或并联在电路中执行简单的逻辑任务。这种继电器的组合叫做逻辑门（logic gates）。
  - 这里提到的逻辑门执行“简单”逻辑任务是指逻辑门只完成最基本的功能。
  - 继电器优于开关之处就在于，继电器可以被其他继电器所控制，而不必由人工控制。这就意味着，这些简单的逻辑门组合起来可以实现更复杂的功能，例如一些简单的算术操作。
  - 事实上，下一章中，我们就将介绍如何利用电线、开关、灯泡、电池和电报继电器来制作一个加法器（尽管它只能用于二进制数计算）。
- 继电器对于电报系统的工作而言是至关重要的。
  - 在长距离情况下，连接电报站的电线具有很高的电阻。这就需要采取一些措施来接收微弱信号并把它增强后再发射出去。继电器就是通过电磁铁控制开关来实现这一目的。实际上，继电器是通过放大微弱信号来生成强信号的。
  - 就我们的目的而言，我们对于继电器放大微弱信号的功能并不感兴趣。我们真正感兴趣的是继电器可以作为一个电流控制而非人工控制的开关。
- 当电流流经输入时（例如，用一个开关把输入连到“V”端），电磁铁就会被触发，输出就得到一个电压。
  - 继电器的输入不一定只能是开关，其输出也未必只限于灯泡。一个继电器的输出可以连到另一个继电器的输入

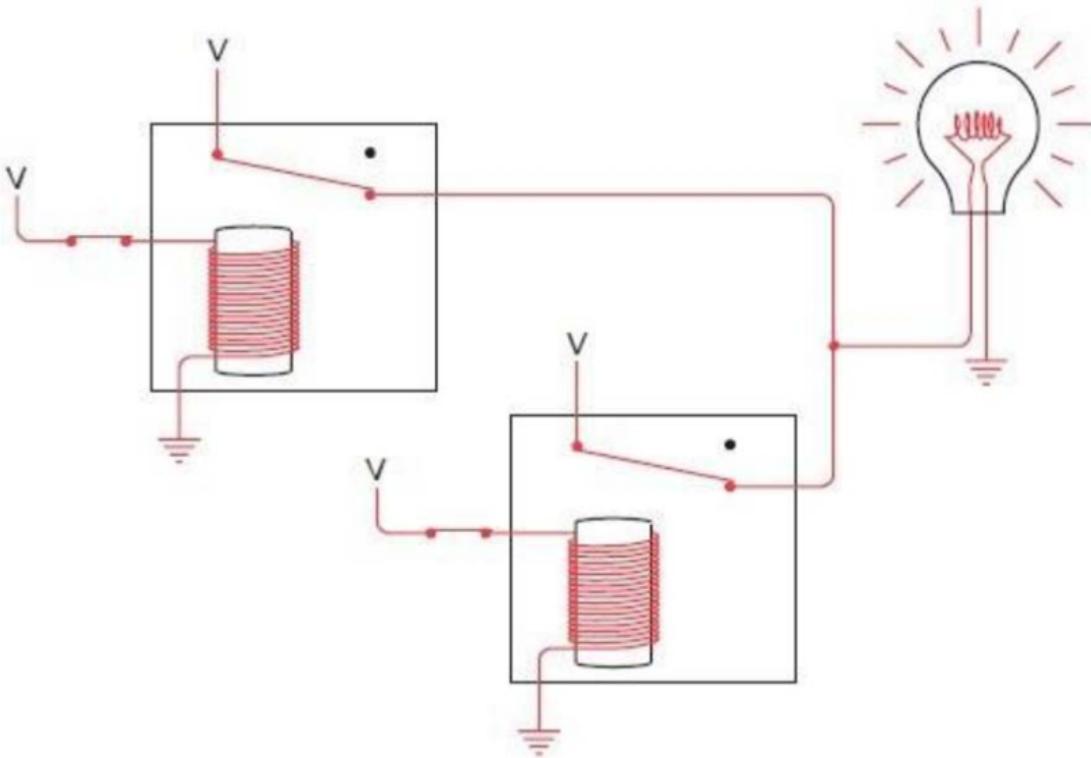
- 当电磁铁将金属簧片拉下来时，我们称继电器被“触发”（triggered）。
- 连接继电器是建立逻辑门的关键。
  - 两个继电器都被触发，电流从V流经灯泡后流入到地中。只有当两个继电器都被触发的时候灯泡才会亮。这样两个继电器的串联被称为一个“与门”。



- 为了避免复杂的图示，电气工程师用如下专门的符号表示一个与门。与门的符号不仅仅代替了两个串联的继电器，而且还暗示着上面的继电器与电源相连，两个继电器都接地。只有当上面的开关与下面的开关都闭合的时候，灯泡才会发光。



- 当上面的开关或下面的开关闭合，灯泡都会发光，这里的关键词是“或”，因此这样的门被称为“或门”。它和与门的符号稍微有点相似，但是输入端的一边是弧线，像英文单词“OR”中字母“O”一样（这样可以帮你分清它们）。



- 开关闭合，灯泡就会熄灭。以这种方式连接的继电器叫做反向器（inverter）。反向器不是逻辑门（一个逻辑门通常有两个或多个输入），尽管如此，它的用处还是很广。
- 或非门，简称NOR，只有两个输入都为0，输出才为1。
- 与非门，简称NAND，只有两个输入都为1，输出才为0。
- 缓冲器（buffer）。很明显，缓冲器“没有什么作用”，它的输入与输出是相同的。在输入信号很微弱的时候，缓冲器就可以派上用场。之前提到过，这也就是很多年前在电报机中使用继电器的原因。另外，缓冲器还可以用于延迟一个信号。这是因为继电器需要一点时间——几分之一秒——才会被触发。

- 以后的电路会由缓冲器、反向器、四种基本逻辑门和其他由逻辑门组成的复杂电路（如2-4译码器）组成。当然，所有这些器件也是由继电器构成的，但我们用不着直接使用它了。

<b>AND</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	0
<b>1</b>	0	1

<b>OR</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	1

<b>NAND</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	1
<b>1</b>	1	0

<b>NOR</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	0
<b>1</b>	0	0

- 按照惯例，黑实心点表示交叉线之间是连接的，没有黑实心点的交叉线则表示仅仅是穿过，没有连接。
- 一个门（或反向器）的输出可以作为一个或多个其他门（或反向器）的输入。但是两个或多个门（或反向器）的输出是不可以相互连接的。
- 摩根定律在电路中的实现。
  - 带有两个反向输入的与门和或非门是等价的。
  - 带有两个反向输入的或门和与非门也是等价的。
- 摩根定律可以简单地表示为如下形式：

$$\overline{\overline{A} \times \overline{B}} = \overline{\overline{A} + \overline{B}}$$

$$\overline{\overline{A} + \overline{B}} = \overline{\overline{A} \times \overline{B}}$$

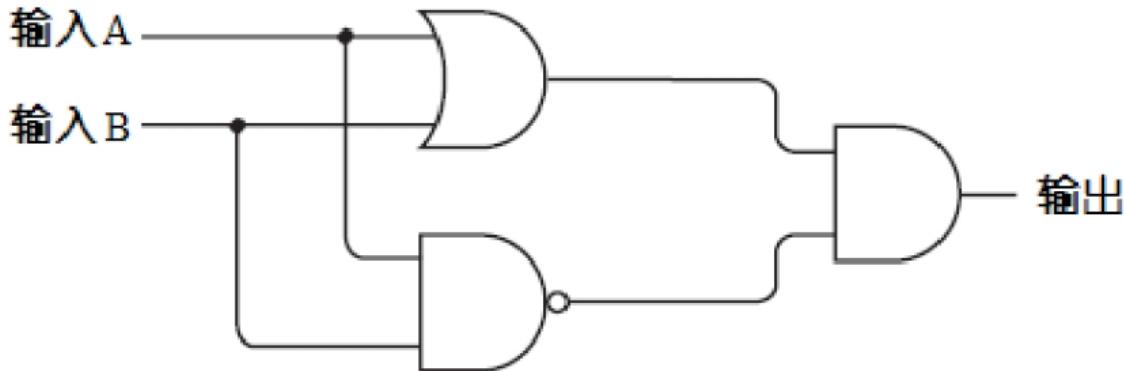
A和B是两个布尔操作数。在第一个表达式中，两个操作数先被取反，再进行与运算。这与两个操作数进行或运算后再取反（即或非）的结果是一样的。在第二个表达式中，两个操作数先取反，再进行或运算，这和两个操作数先做与运算后再取反（即与非）的结果是等价的。

- 摩根定律是简化布尔表达式的一种重要手段，因此也可以用来简化电路。从历史的角度来说，这正是香农的论文带给电气工程师们的真正意义。
  - 但在本书中简化电路不是重点，我们关注的是让一切有效地运转而不是以最简的形式运转。我们下面要做的是一个加法器。

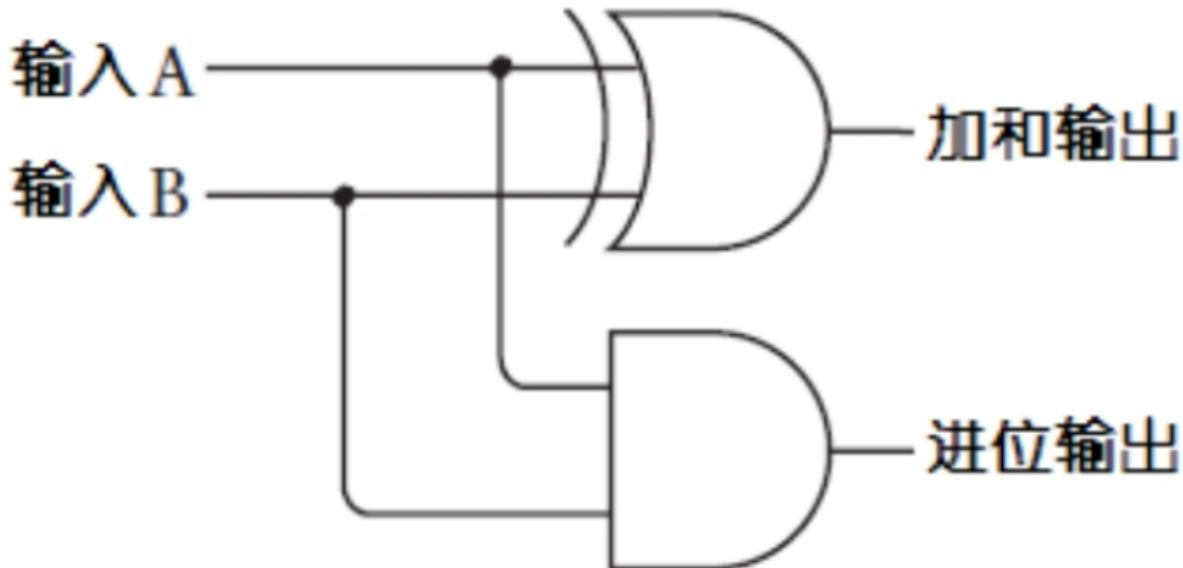
## 12 二进制加法器

- 加法是算术运算中最基本的运算，因此如果想搭建一台计算机（这也正是本书所隐含的内容），那么首先就要造出可以计算两个数的和的器件。当你真正面对它时，就会发现，**原来加法计算就是计算机要做的唯一工作**。如果我们可以造出加法器，同样地，就可以利用加法来实现减法、乘法和除法，计算按揭付款，引导火箭飞到火星、下棋，以及填写我们的话费账单。

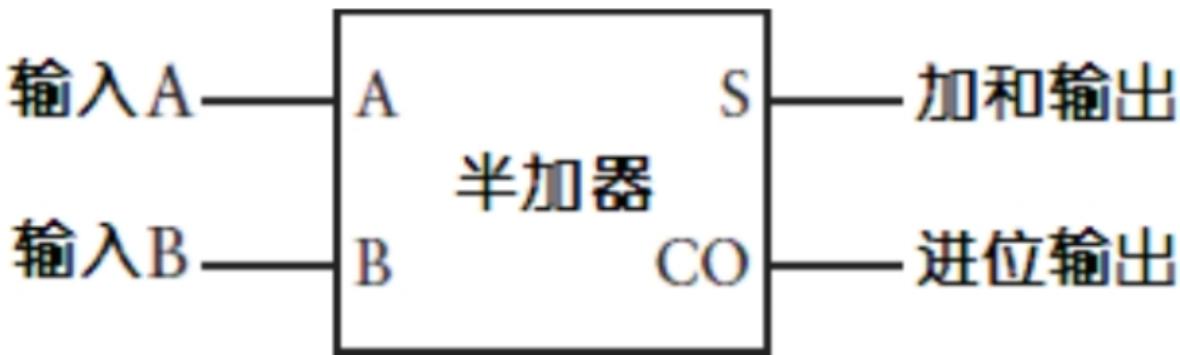
- 一对二进制数相加的结果中具有两个数位，其中一位叫做加法位（sum bit），另一位则叫做进位位（carry bit，例如，1加1等于0，进位为1）。用这种方法来看二进制加法非常方便，因为在我们的加法器中加法与进位是分别进行的。
- 加法器构成：开关、灯泡、以各种形式连接起来的逻辑门。开关将触发逻辑门中的继电器来点亮相应的灯泡。
- 没有人能看懂以各种方式连接起来的上百个继电器所表达的意义。相反地，我们要利用逻辑门来分阶段地处理这个问题。
  - 利用与门可以计算两个二进制数加法的进位。
  - 加法位：两个输入同时作为或门和与非门的输入。或门和与非门的输出又分别作为一个与门的输入，最后得出了我们想要的结果。



- 实际上这个电路有个专门的名称，叫做异或门，简写为XOR。之所以称为异或门是因为若想其输出结果为1，要么仅让输入A为1，要么仅让输入B为1。
- 异或门在输入端比或门多出了一条曲线，除此之外它看上去和或门非常相像。
- 同或门
- 两个二进制数相加的结果是由异或门的输出给出的，而进位位是由与门的输出给出的。因此我们可以将与门和异或门连在一起计算两个二进制数（即A和B）的和。

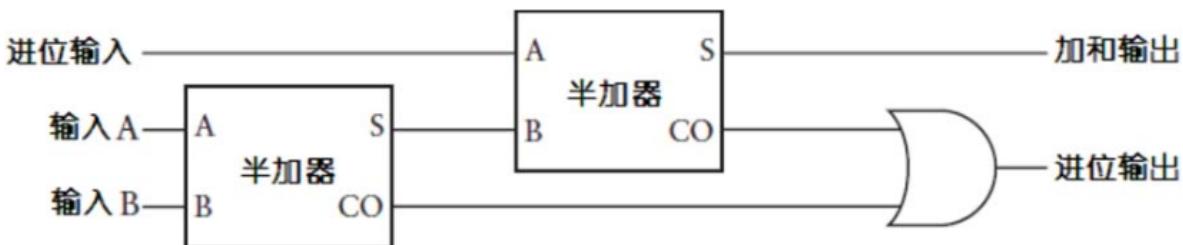


- 为了避免重复画与门和异或门，你可以采用如下这种简单的表示方式。



这个符号被称为半加器 (Half Adder)。绝大多数二进制数是多于1位的，半加器没有做到的是将之前一次的加法可能产生的进位位纳入下一次运算。

- 为了对三个二进制数进行加法运算，我们需要将两个半加器和一个或门做如下连接。



首先从最左边第一个半加器的输入A和输入B开始，其输出是一个加和及相应的进位。这个和必须与前一列的进位输入相加，然后再把它们输入到第二个半加器中。第二个半加器的输出和是最后的结果。两个半加器的进位输出又被输入到一个或门中。你可能会觉得，这里还需要一个半加法器，这当然是可行的。但是如果你了解了所有的可能性之后，你会发现，两个半加法器的进位输出是不会同时为1的。或门在这里已经足够，因为或门除了在输入都为1的时候以外，其他情况下结果和异或门结果相同。

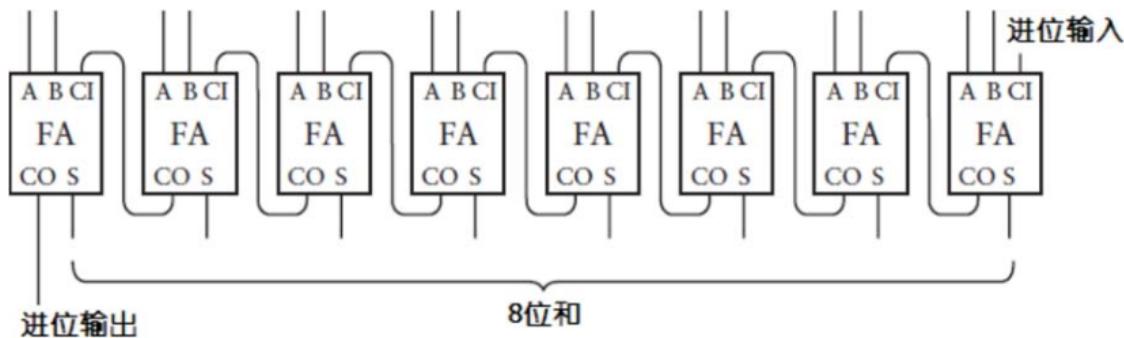
若有进位，则 $S=0, CO=1$ ，因此进位输入+0不可能再产生进位输出。

若无进位，则 $S=1, CO=0$ ，若进位输入为1，可以产生进位输出。

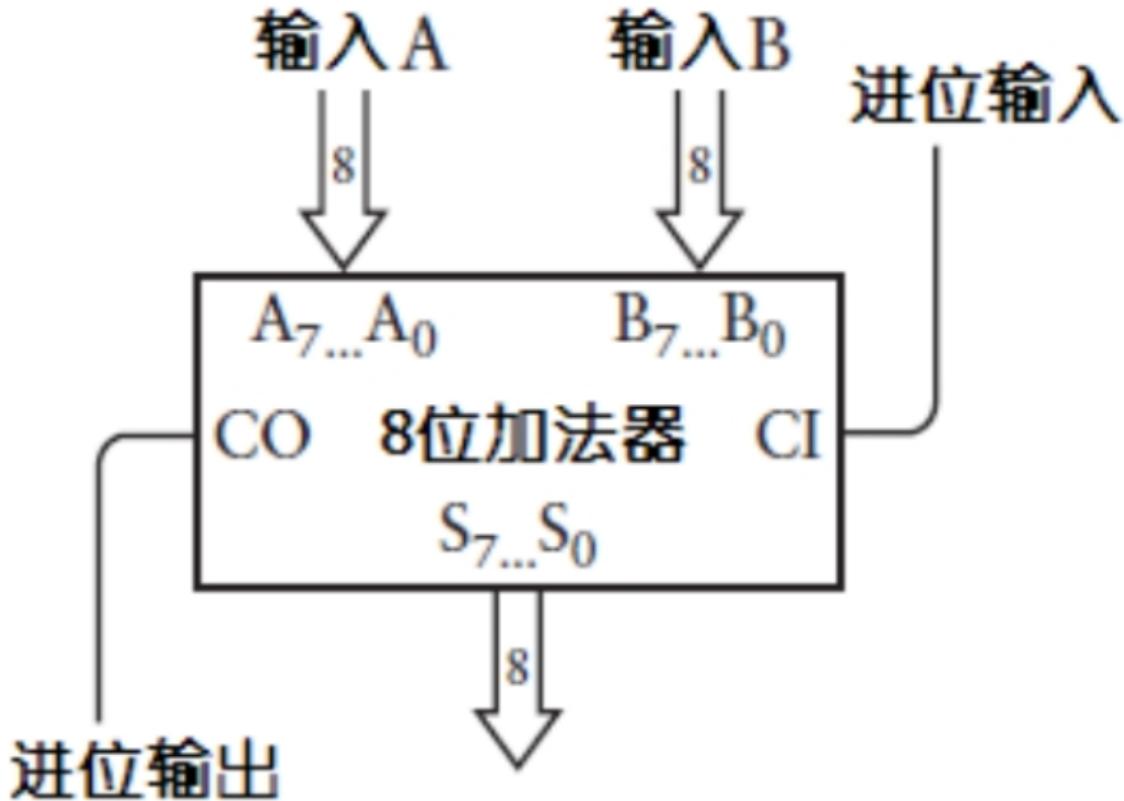
为了避免重复地画上面的那个图，我们用以下形式来替代上图中的一堆符号，它称为全加器 (Full Adder)。



- 8位二进制加法器
  - 每个全加器的进位输出都作为下一个全加器的进位输入。



- 输入标记为A0 ~ A7和B0 ~ B7，输出标记为S0 ~ S7。



- A0、B0和S0是最有效位，或者说是最右边的一位。A7、B7和S7是最高有效位，或者说是最左边的一位。
- 双线箭头包含了8个输入端，代表一组8个独立的信号。

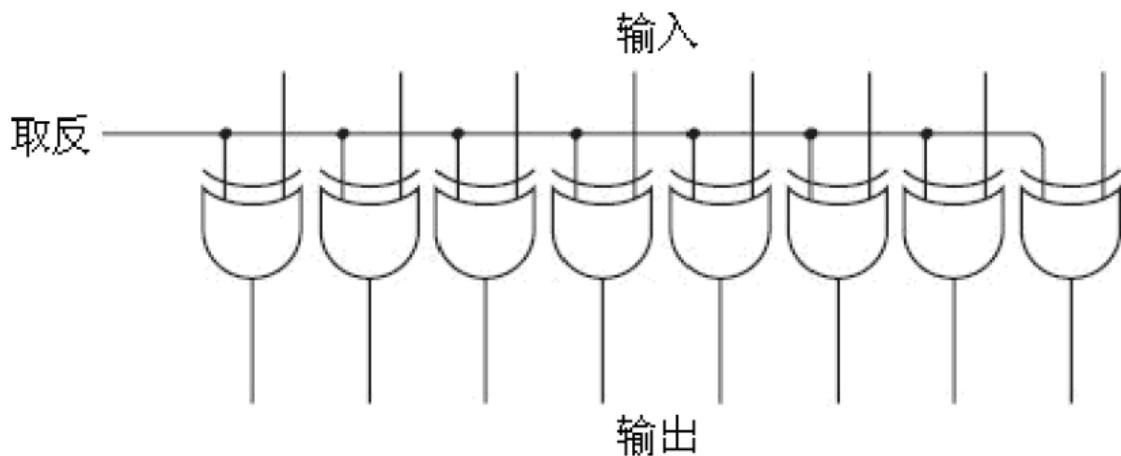
- 计算机进行加法运算时

- 可以制作一个比这个算得更快的加法器。如果你看一下这个电路是如何工作的，最低有效位的一对数字相加所得出的一个进位输出，将要参与接下来的一对数字的加法运算，由此得到的一个进位输出又要参与再下一对数字的加法运算，依此类推。加法器的总体速度等于数字的位数乘以全加器器件的速度，这被称做行波进位（ripplecarry，或脉冲进位）。更快的加法器运用了一个被称为“前置进位”的电路来提高运算的速度。
- 计算机已经不再使用继电器了！尽管它曾经被使用过。第一台数字计算机在20世纪30年代被建造完成，当时所使用的就是继电器，后来也使用过真空管。今天的计算机使用的是晶体管。在被用到计算机中时，晶体管的工作方式与继电器基本相同，但是晶体管要比继电器计算速度更快、体积更小，而且噪声更弱、耗能也更低，而且更便宜。搭建一个8位加法器依然需要144个晶体管（如果你用前置进位法代替行波进位，将会用到更多的晶体管），但是电路却是极小的。

## 13 如何实现减法

- 加法和减法在某些方面相互补充，但在机制方面这两个运算则是不同的。加法是始终从两个加数的最右列向最左列进行计算的。每一列的进位加到下一列中。在减法中没有进位，而是有借位——一种与加法存在本质区别的麻烦机制。
- 由于减法与计算机中以二进制编码的存储有关，详细地了解减法也是很重要的。
- 将进行减法的两个数分别用被减数 (minuend) 和减数 (subtrahend) 表示。从被减数中减去减数，得到的结果叫做差 (difference) 。
- $253-176=77$ 
  - 为了避免借位，首先要从999中减去减数，而不是从原来的被减数中减去减数。
  - 由于操作数是三位数，所以这里使用999。如果操作数是4位，则用9999。从一串9中减去一个数叫做对9求补数。176对9的补数是823。这样的好处就是无论减数是多少，计算对9的补数都不需要借位。
  - 计算出减数对9的补数后，将补数与原来的被减数相加：
    - $253+823=1076$
    - 最后再将结果加1，并减去1000。
      - $1076+1-1000=77$
    - 到此，我们就得到了结果。答案与先前的相同，而且没有用到借位。
- 原理：
  - $253-176$ 
    - 在这个式子中加上一个数再减去这个数，结果是相同的。
  - 因此先加上1000，再减去1000：
    - $253-176+1000-1000$
  - 这个式子与下式等价：
    - $253-176+999+1-1000$
  - 然后用以下方式将数字重新组合：
    - $253+(999-176)+1-1000$
  - 这个式子与刚才描述过的用9的补数进行的计算是相同的。
  - 我们用两个减法和两个加法来替代一个减法，而在这个过程中避免了烦琐的借位。
- 减数大于被减数
  - $176-253$ 
    - 首先要像前面一样，用999减去减数253，计算出对9的补数746
    - 把该数对9的补数与被减数相加
      - $176+746=922$
    - 在前面的例子中，下一步应该加1，并减去1000来得到最终结果。但是在这里，这种方法并不适用。因为你会遇到923减去1000的情况，这又导致了借位。
    - 由于我们之前已经加了999，这里再减去999。到这里，我们会意识到这个问题的结果是负数，因此需要将减数与被减数交换，用999减去922。这里没有用到借位，结果与我们期望的相同
      - $922-999=-77$

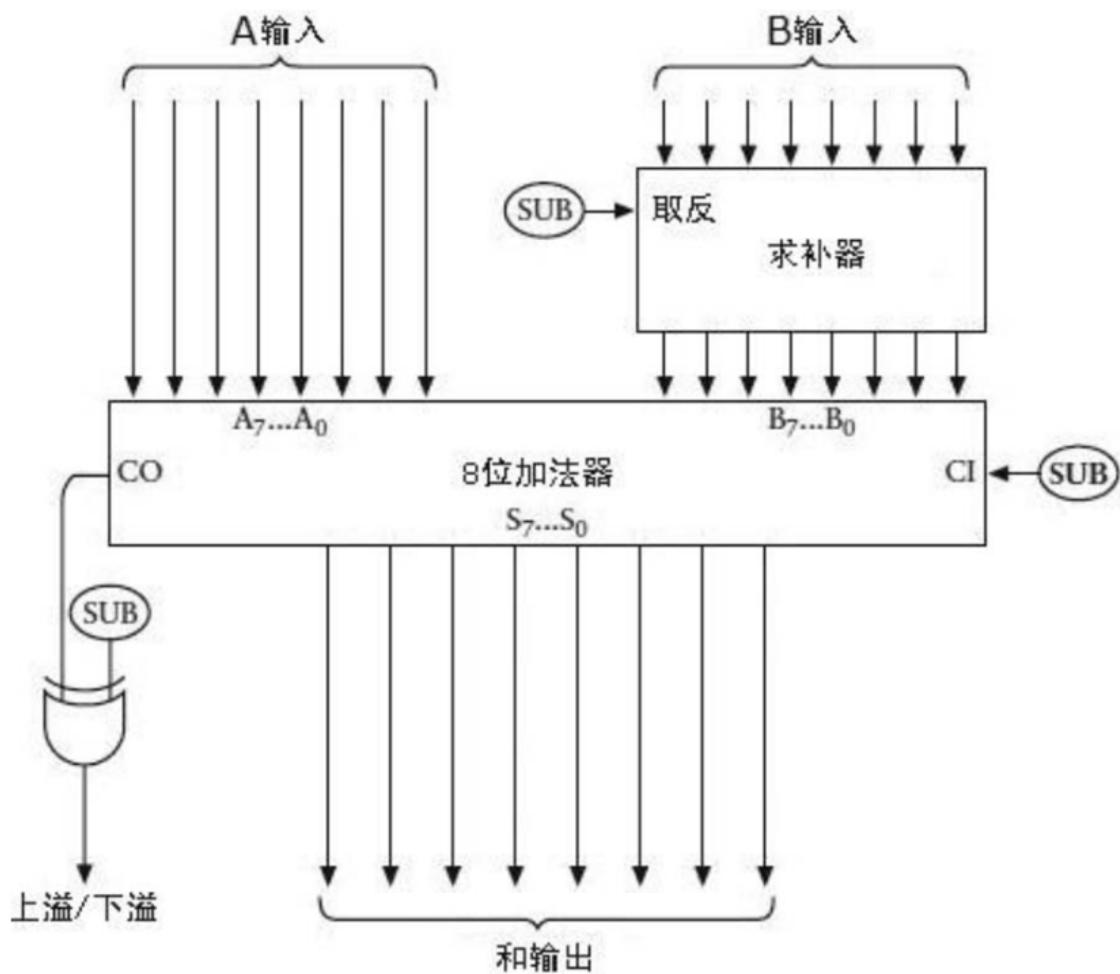
- 同样的技巧可以用于二进制数中，而且实际上这要比十进制数简单。
  - 253-176 (11111101-10110000)
    - 第一步，用11111111 (即255) 减去减数**
      - 当计算十进制数减法的时候，减数是从一串9中减去的，结果称为9的补数。在二进制数减法中，减数是从一串1中减去的，结果称为1的补数。
      - 但是请注意，我们在求对1的补数时并不需要用到减法。在求对1的补数时，只需将原来的二进制数中的1变为0，将0变为1即可。因此对1求补数有时也会称为相反数 (negation) 或反码 (inverse)。
    - 第二步，将减数对1的补数与被减数相加**
      - $11111101 + 01001111 = 101001100$
    - 第三步，将上式所得结果加1**
      - $101001100 + 1 = 101001101$
    - 第四步，减去100000000 (即256)**
      - $101001101 - 100000000 = 1001101$  (77TEN)
  - 176-253 (10110000-11111101)
    - 第一步，用11111111减去减数，得到对1的补数**
      - $11111111 - 11111101 = 00000010$
    - 第二步，将减数对1的补数与被减数相加**
      - $10110000 + 00000010 = 10110010$
    - 现在我们要用某种方法在结果中减去11111111。当减数小于被减数的时候，我们将结果加1再减100000000来完成计算。但是你无法在不借位的情况下做到这一点。所以，我们先用11111111减去所得结果：
      - $11111111 - 10110010 = 01001101$
      - 这里又一次用到了将各位取反来求得结果的方法，但是这个结果是77，而真正的答案应该是-77。
  - 改造搭建的加法器，并让它像实现加法一样来实现减法运算。为了不让问题太复杂，这个新的加/减法器只执行在减数小于被减数的减法操作，即结果为正数的操作。
    - 8位加/减法器所用的新面板较从前做了些许的改动。它增设了一个开关，用以选择做加法还是做减法。
    - 此外，右侧的8个灯泡用于表示计算结果。这里，第9个灯泡表示“上溢/下溢”。这个灯泡表明了正在计算的数字是一个不能用8个灯泡来表示的数字。
      - 如果在加法中得到了大于255 (上溢， overflow) 或在减法中得到了负数 (下溢， underflow) 这个灯泡就会发光。当减数大于被减数的时候，就会得到一个负数。
    - 加法器中新增的主要部分就是一个用来求8位二进制数对1补数的电路。之前提到，二进制数对1求补数相当于对其每位取反，因此我们计算8位二进制数补数的时候可以简单地应用8个反向器。问题是，该电路只会对输入求反，而我们要的是一台既能做加法又能做减法的机器，因此就要求该电路当且仅当进行减法运算时才实现反转。电路可以改造为如下图所示。



如果“取反”信号是0，则8个异或门输出与输入相同。如果“取反”信号为1，则输出信号反置。

将8个异或门合并起来画成一个器件，称为求补器（One's Complement）

- 将一个求补器，一个8位二进制加法器和一个异或门做如下连接。



这里三个信号都标识为“SUB”，这就是加/减法转换开关。当该信号为0的时候，其进行的是加法运算，为1时进行的则是减法运算。

在减法中，输入B（第二排开关）在送入加法器之前，需先通过求补电路进行取反。

此外，在做减法时，我们通过设定CI（进位输入）为1来使得结果加1。而在加法中，求补电路将不起作用，且输入CI为0。

加法器的SUB信号和CO（进位输出）输出作为异或门的输入来控制表示上溢/下溢的灯泡。如果SUB信号为0（表示进行加法运算），则当加法器CO输出为1时灯亮，意思是加法计算结果大于255。

当进行减法运算的时候，如果减数（输入B）小于被减数（输入A），这时加法器的CO输出为1。这表示减法的最后一步要减去100000000。上溢/下溢指示灯仅在加法器的CO输出为0，也就是当减数大于被减数，结果为负时才会亮起。但上面所示器件现在还不能表示负数。

- 负数在二进制中是如何表示的

- 当然，你可以简单地用一个二进制位来表示负号，当这一位为1的时候就表示负数，为0则表示正数，尽管这样也是可行的，但还远远不够。
  - 还有另一种方法可以解决负数的表示问题，而且它还可以很轻松地让负数和正数相加。这种方法的缺点是必须提前算一下可能遇到的所有数字的位数。
    - 如果我们并不需要无限大或无限小的数，而且在开始的时候我们就可以预知所使用的数字的范围，那情况就有所不同了。
    - -500到499之间，总共1000个数。这个约束说明只用三位十进制数，且不用负号就可以表示所有需要的数字。我们并不需要用到从500到999之间的正数，因为我们所需要的数的最大值为499。因此从500到999的三位数可以用来表示负数。
      - -500, -499, -498 ... -4, -3, -2, -1, 0, 1, 2, 3, 4 ... 497, 498, 499
      - 用这种表示法，我们可以将它们写成：500, 501, 502 ... 996, 997, 998, 999, 000, 001, 002, 003, 004 ... 497, 498, 499
      - 这就形成了一个循环排序。最小的负数（500）看起来像是最大正数（499）的延续。而数字999（实际上是-1）是比0小1的第一个负数。如果我们在999上加1，通常会得到1000。由于我们处理的是三位数，这个结果实际上就是000。
    - 这种标记方法称为10的补数（ten's complement）。为了将三位负数转化为10的补数，我们用999减去它再加1。也就是说，对10求补数就是对9的补数再加1。
      - 例如想要得到-255对10的补数，用999减去255得到744，然后再加1，得到745。
    - 利用10的补数，我们将不会再用到减法。所有的步骤都用加法来进行。
      - 143-78
        - -78对10的补数为999-078+1，即922
        - 143+922，相当于65（忽略溢出）
        - 如果我们又-150，-150对10求补数为850。因此65加上850等于915，就是新的账户余额。而这个余额实际上是-85美元。
  - 这样的机制在二进制中被称为2的补数。以8位二进制数为例。范围为00000000 ~ 11111111，对应十进制中的0 ~ 255。但是如果你还想表示负数的话，则以1开头的每个8位数都表示一个负数，如下表所示。

二进制数	十进制数
10000000	-128
10000001	-127
10000010	-126
10000011	-125
...	...
11111101	-3
11111110	-2
11111111	-1
00000000	0
00000001	1
10000010	2
...	...
01111100	124
01111101	125
01111110	126
01111111	127

现在所表示的数的范围是-128 ~ +127。最高有效位（最左位）作为符号位（sign bit）。符号位中，1表示负数，0表示正数。

为了计算2的补数，则首先要计算1的补数，然后再加1。这等价于将每位取反再加1。例如，十进制数125写为二进制为01111101。为了表示-125的对2的补数，首先将01111101的每位取反，得到10000010，再加1，得到10000011。用同样的步骤，每位取反再加1，可以将数值还原。

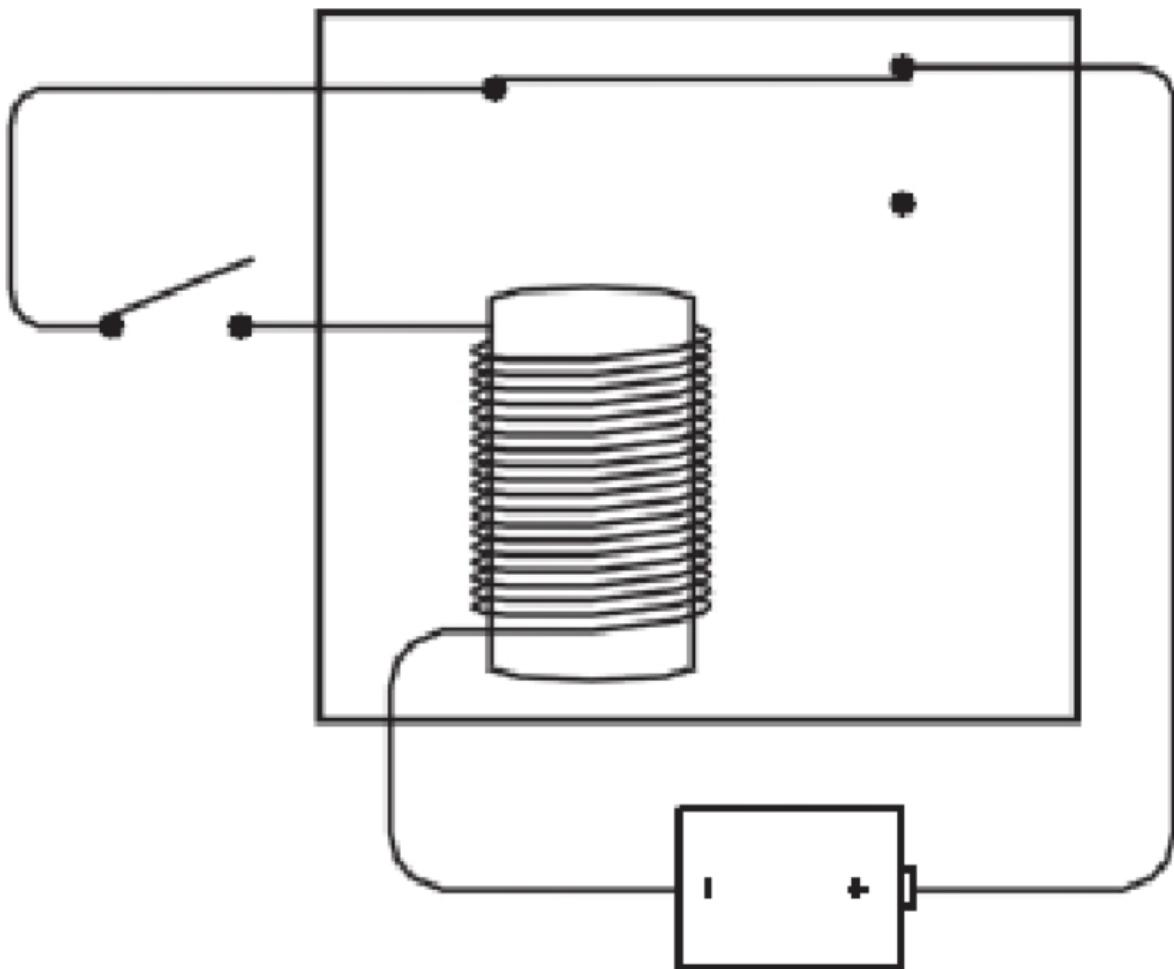
- 这个系统为我们提供了一种不用负号就能表示正、负数的方法。同样也让我们自由地将正数和负数用加法法则相加。
  - 例如，将与-127和124等价的两个二进制数相加。利用上面的表格，可以简单地写为
  - $10000001 + 01111100 = 11111101$
  - 结果等于十进制的-3。
  - 要注意的是，这里涉及了上溢和下溢情况，即结果大于127或小于-128。

- 例如，将125与它自身相加
    - $01111\ 1101 + 0111\ 1101 = 1111\ 1010$  (-6)
  - 将-125与它本身相加也会出现同样的情况
    - $1000\ 0011 + 1000\ 0011 = 1\ 0000\ 0110$  (只看右边八位, +6)
  - 一般来说，如果两个操作数的符号相同，而结果的符号与操作数的符号不相同，这样的加法就是无效的。
- 二进制数可以有两种不同的使用方法。二进制数可以是有符号的，也可以是无符号的。
    - 无符号的8位二进制数所表示的范围是0 ~ 255。有符号的8位二进制表示的范围是-128 ~ 127。
    - 无论是有符号的还是无符号的，数字本身是无法显示的。例如，如果有一个人问：“有一个8位二进制数，值为10110110。它相当于十进制的多少？”你必须先问：“它是有符号数还是无符号数？它可能为-74或者182。”
    - 这就是二进制数的麻烦之处，它们只是一些0和1，本身并没有任何含义。

## 14 反馈与触发器

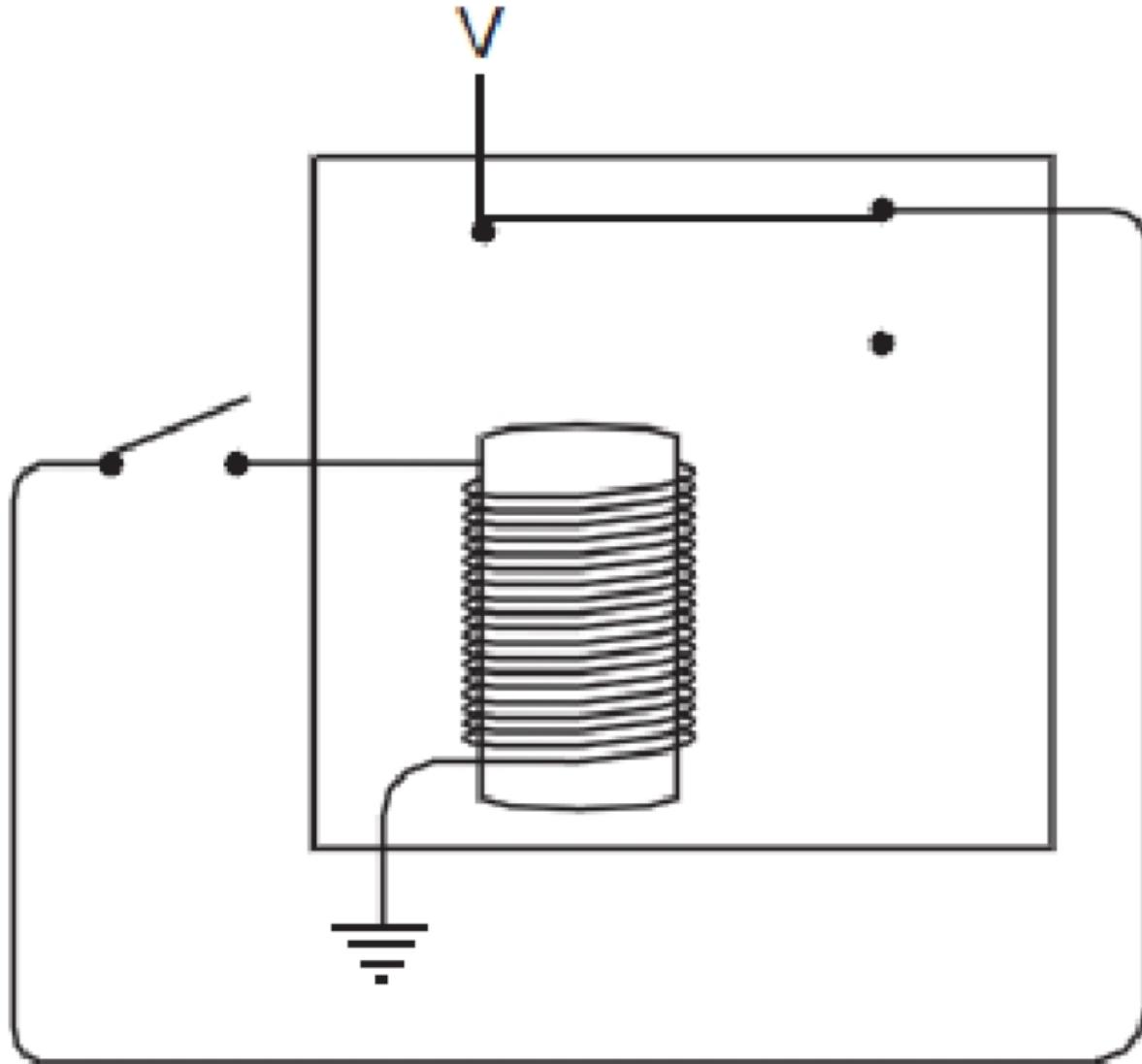
---

- 电可以让物体运动，这个道理人人都懂。只要稍微扫视一下我们的房间就会发现：很多电器中都装了电动机，比如钟表、电扇、食品加工器，以及CD播放器等。电同样可以使扩音器中的磁芯振动，正因为如此我们的音响设备、电视机才能够产生声音，播放语言和音乐。
- 有一类设备或许能很清晰地阐释电能驱使物体运动的最简单也最具代表性的方式，然而由于这类设备正在被能够实现同样功能的电子器件逐步取代，它们正在迅速地消失。在我看来，最令人赞叹的例子应该算是电子蜂鸣器和电铃了。
- 将继电器、电池、开关按如下形式连接。

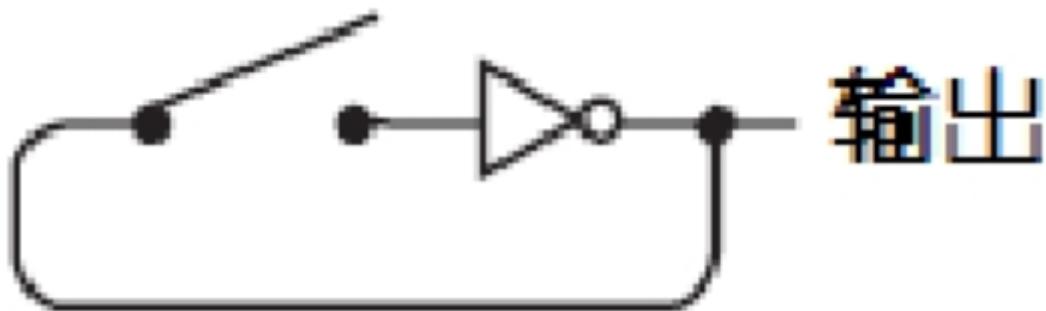


开关一旦闭合，金属簧片就会上下跳动，电路也会随之连通或断开。如果金属簧片发出了一种刺耳的声音，这套系统就成为了一个蜂鸣器。如果金属簧片前端是一把小锤子，旁边只要放上一个锣，就构成了一个电铃。

- 蜂鸣器的另一种形式



其实中间部分是一个反向器，因此电路可以简化为如下图所示。你也可以将电路中的开关省去。

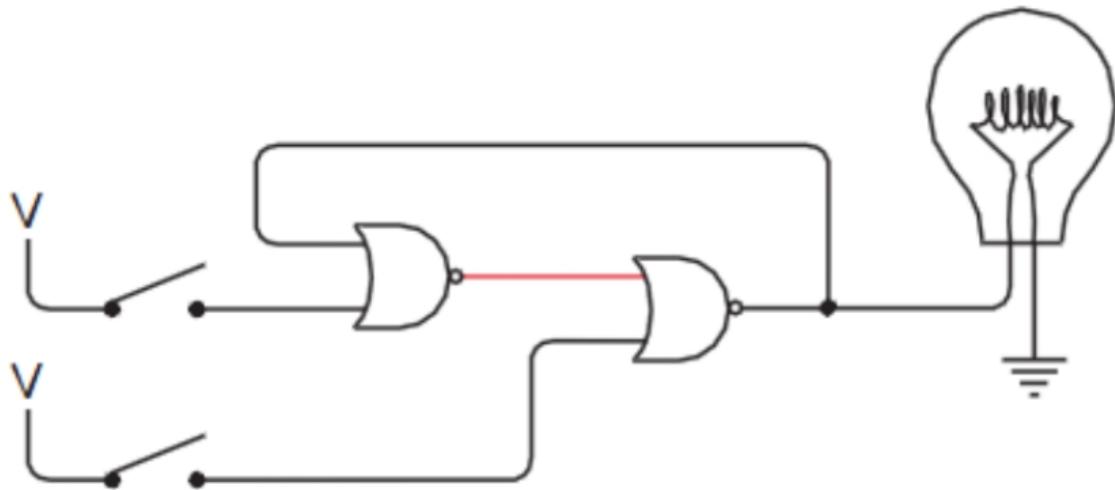


反向器的输出与其输入是相反的，但是在这里，输出同时又是输入。反向器在本质上就是一个继电器，而继电器将状态取反以得到另一个状态是需要一点点时间的。所以，即使输入和输出是相同的，输出也会很快地改变，成为输入的相反状态（当然，输出随即也会很快改变输入，如此反复）。

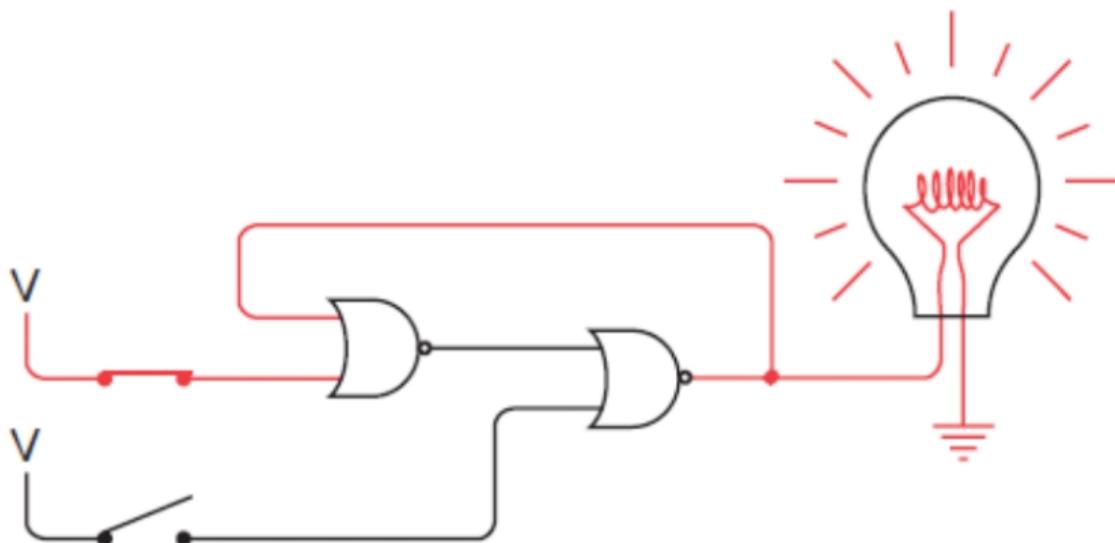
电路的输出是什么呢？其实就是要么提供电压，要么不提供电压，在两者之间切换。我们也可以换种方式来表达——输出结果要么是0，要么是1。

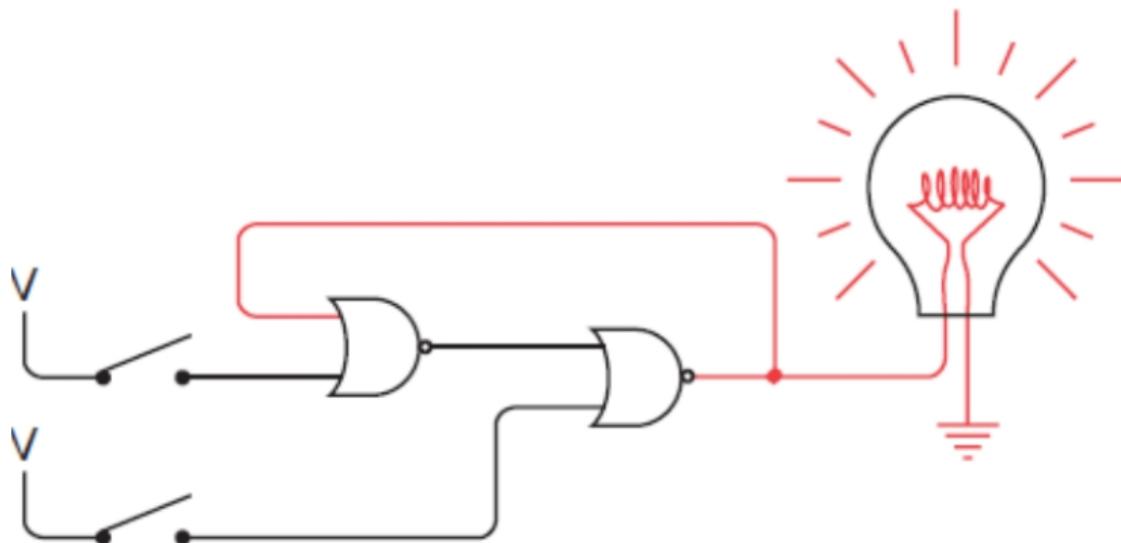
- 我们把这种电路称为振荡器（oscillator），它和我们先前学到的所有东西存在本质上的区别。在此之前我们讲过的所有的电路，其状态的改变都依靠人为的干预，通常是通过改变开关状态来实现的。但是振荡器却在不需要人干涉的情况下，可以完全自发地工作。

- 单独的一个振荡器用处并不大，在与其他电路连接后所组成的自动控制系统中，振荡器有着举足轻重的作用。为了使不同组件同步工作，所有计算机都配备着某种振荡器。
- 振荡器的输出在0和1之间按照固有的规律交替变化。正因为这一点，振荡器又经常被称为时钟（clock），通过振荡进行计数也是一种计时方式。
- 经过一段时间又回到先前初始状态的这一段间隔定义为振荡器的一个循环（cycle），或者称为一个周期。
- 一个循环所占用的时间就是该振荡器的周期（period）。
- 周期的倒数就是振荡器的频率（frequency）。代表每秒钟的循环次数，单位Hz。
- 反馈（feedback），系统的输出返回给输入
- 下面是一个包含两个或非门、两个开关和一个灯泡的电路。左边或非门的输出是右边或非门的输入，而右边或非门的输出是左边或非门的输入。
- 假设初始态，电路中只有左边的或非门输出电流，灯泡不亮。

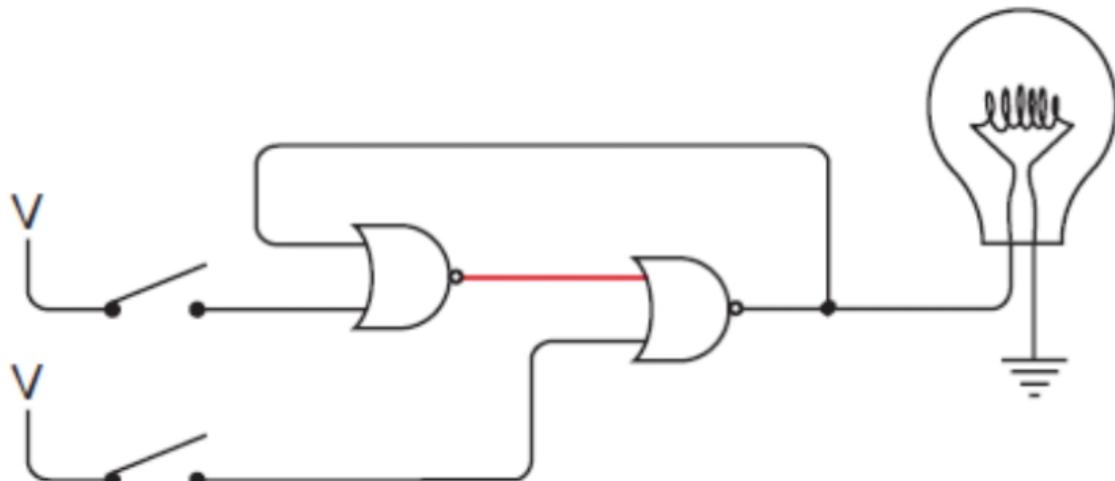
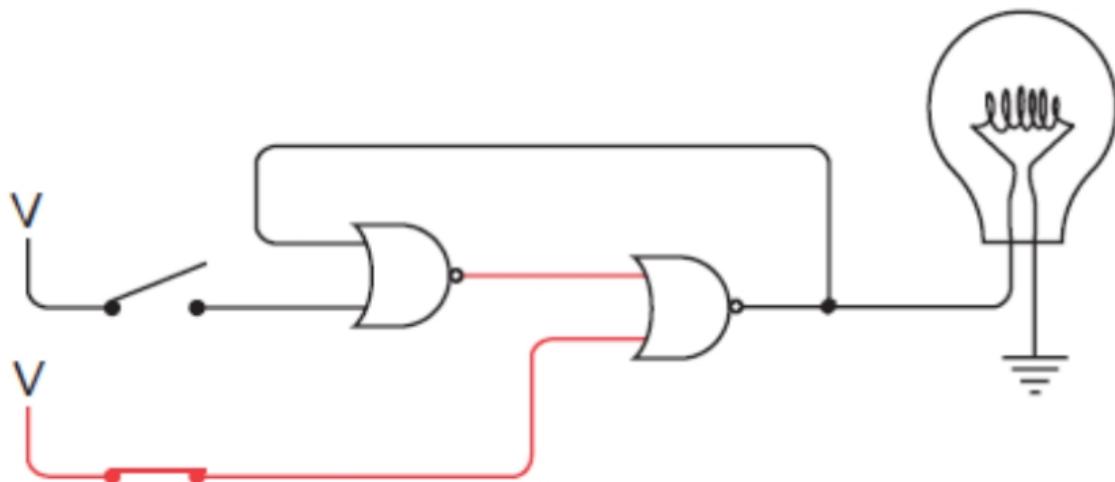


- 接通上面的开关，灯泡被点亮，断开此开关灯泡仍然亮着。



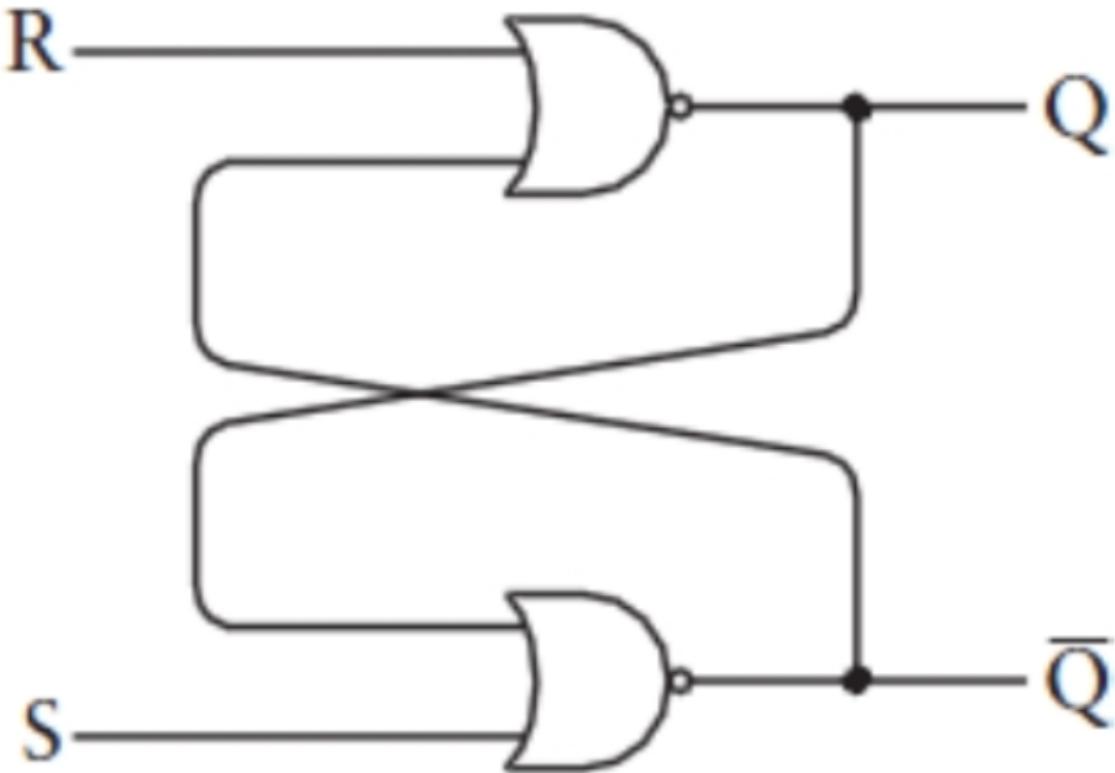


- 接通下面的开关，灯泡被熄灭，断开此开关灯泡仍然不亮。



- 电路的奇怪之处是：同样是在两个开关都断开的状态下，灯泡有时亮着，有时却不亮。
- 当两个开关都断开时，电路有两个稳定态，这类电路统称为**触发器**（Flip-Flop），触发器是在1918年被发明的。
- 触发器电路可以保持信息，它可以“记住”某些信息。
  - 特别地，对于本章先前所讲述的触发器，它可以记住最近一次是哪个开关先闭合。如果如果它的灯泡是亮着的，你就可以推测出最后一次连通的是上面的开关；而如果灯泡不亮则可推测出最后一次连通的是下面的开关。

- 触发器和跷跷板有着很强的相似性。跷跷板也有两个稳定状态，它不会长期停留在不稳定的中间位置。通过观察跷跷板，我们很容易推测出哪边最后一次被压下来。
- 尽管你现在可能还没感受到这一点，但触发器的确是一种必不可少的工具。它们可以让电路“记住”之前发生了什么事情。
- 想象一下，如果你没有了记忆力，该如何去数数，我们不记得刚刚数过的数，当然也就无法确定下一个数是什么！同理，一个能计数的电路（本章后面要讲到）必定需要触发器。
- 触发器种类繁多，先前所讲述的是最简单的一种R-S（Reset-Set，复位/置位）触发器。
  - 我们通常把两个或非门绘制成另一种形式，加上标识符就得到了下面这幅图。



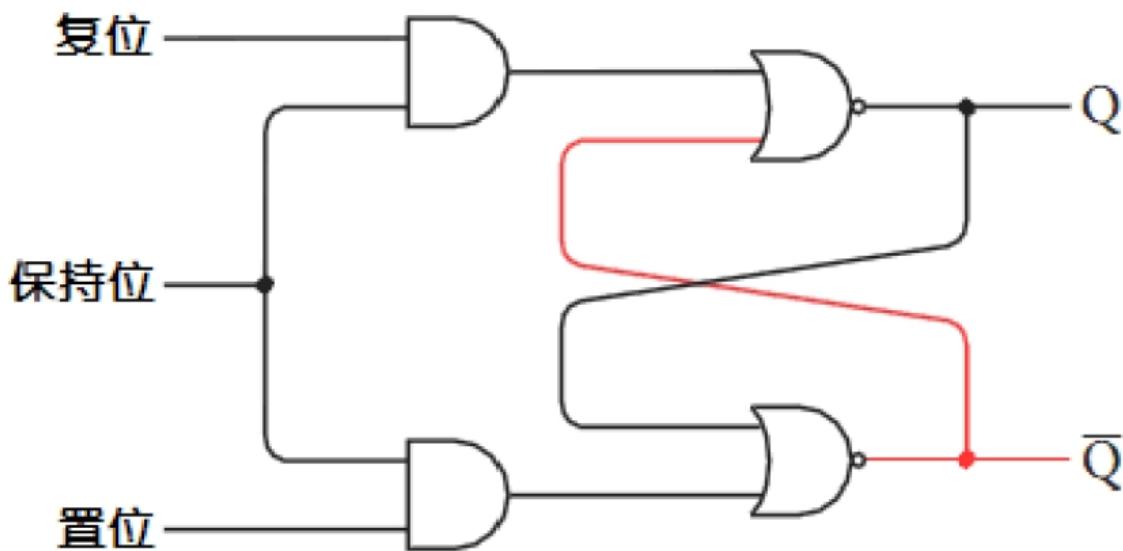
- 我们通常用Q来表示用于点亮灯泡的输出的状态。另一个输出（读做Q反）是对Q的取反。
- 输入端S（Set）用来置位，R（Reset）用来复位。你可以把“置位”理解为把Q设为1，而“复位”是把Q设为0。当S和R输入均为0时，输出保持为S、R同时被设为0以前的输出值。

输入		输出	
S	R	Q	$\bar{Q}$
1	0	1	0
0	1	0	1
0	0	Q	$\bar{Q}$
1	1	禁止	

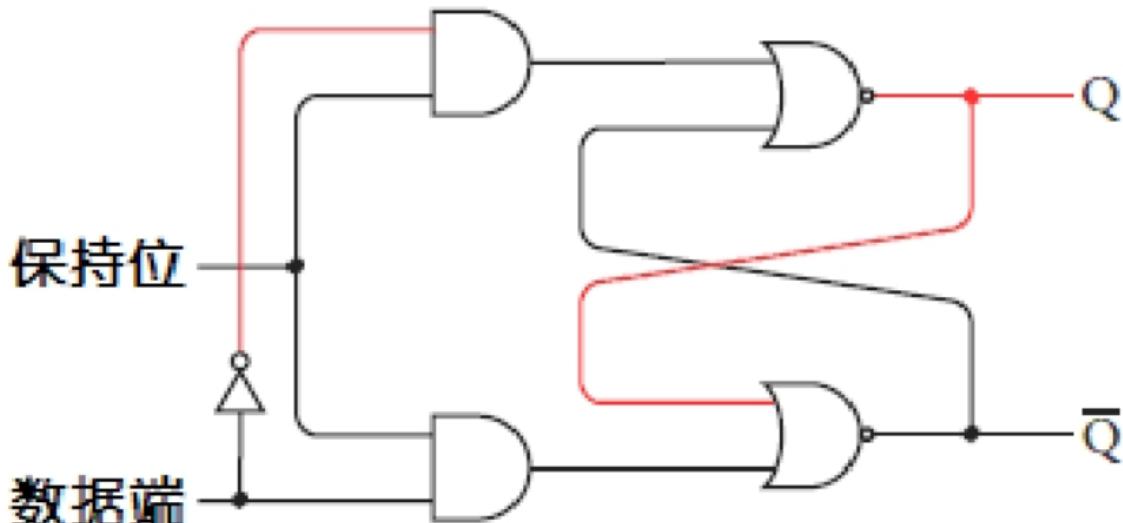
这类表称为功能表 (function table) 、逻辑表 (logic table) 或真值表 (truth table) 。它表达了不同输入组合所对应的不同输出结果。因为R-S触发器仅有两个输入端，所以不同的输入组合共有4种，分别对应于表中的4行。

表中最后一行表示S和R均为1的输入组合是不合法的。如果S、R状态同时为1时，Q和 $\bar{Q}$ 均会为零，这与Q和 $\bar{Q}$ 互反的假设关系相矛盾。所以当使用R-S触发器进行电路设计时，R、S输入同时为1的情况一定要避免。

- R-S触发器最突出的特点在于，它可以记住哪个输入端的最终状态为1。但是有时候我们需要一种记忆能力更加强大的电路，例如能记住在某个特定时间点上的一个信号是0还是1。
  - 在构造具备这种功能的电路之前，让我们先来思考一下它的具体行为。
    - 这个电路存在两个输入。其中一个我们称之为数据端 (Data) 。与所有数字信号一样，数据端取值为0或1；另一个输入被称为保持位 (Hold That Bit) ，保持位的作用就是使当前的状态被“记住”，通常情况下保持位被设置为0，在这种情况下数据端对电路不产生影响。当保持位置1时，数据端的值就会在电路系统中被“记住”。随后保持位又置为0，这时电路已经“记住”了数据端的最后一次输入，而之后数据端的输入无论如何变化都不会对电路产生影响。
  - 使用先前学过的R-S触发器来实现这种具有保持位的功能系统。当保持位信号为1时，这套电路系统就和先前讲过的R-S触发器功能一致。

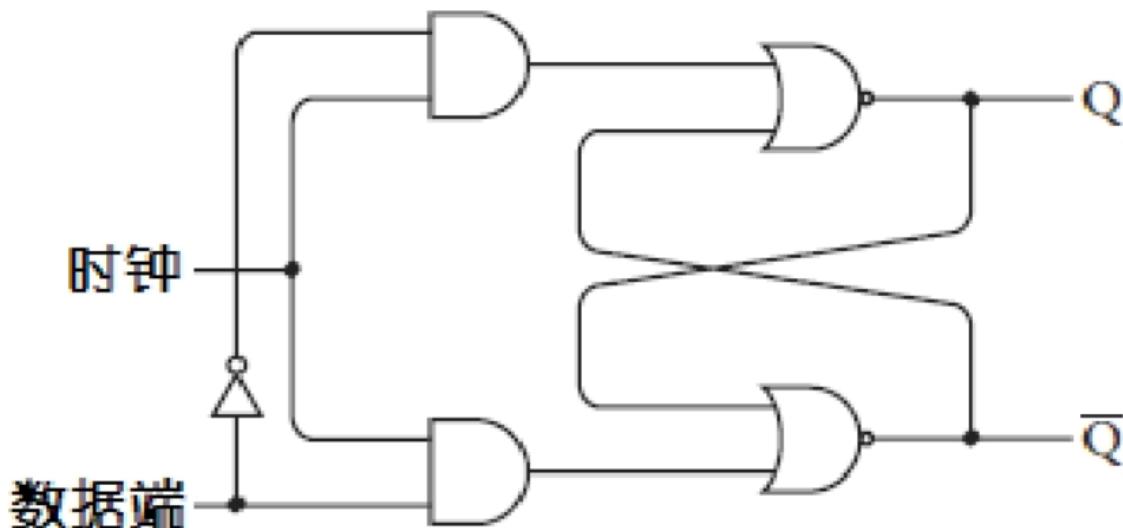


真正有意义的输入可以是S为0, R为1或者是R为0, S为1的情形。



这个电路称为**电平触发的D型触发器**, D (Data) 表示数据端输入。所谓电平触发是指当保持位输入为某一特定电平 (本例中为“1”) 时, 触发器才保存数据端的输入值。

通常情况下, 当这种电路出现在书中的时候, 输入端是不会被标记为保持位的, 而是被标记为**时钟 (clock)**。当然, 这种信号并不是真正的时钟, 但是在某些情况下它却具有类似时钟的属性, 即它可以在0和1之间有规律地来回变化。但是现在时钟仅仅用来指示什么时候保存数据。



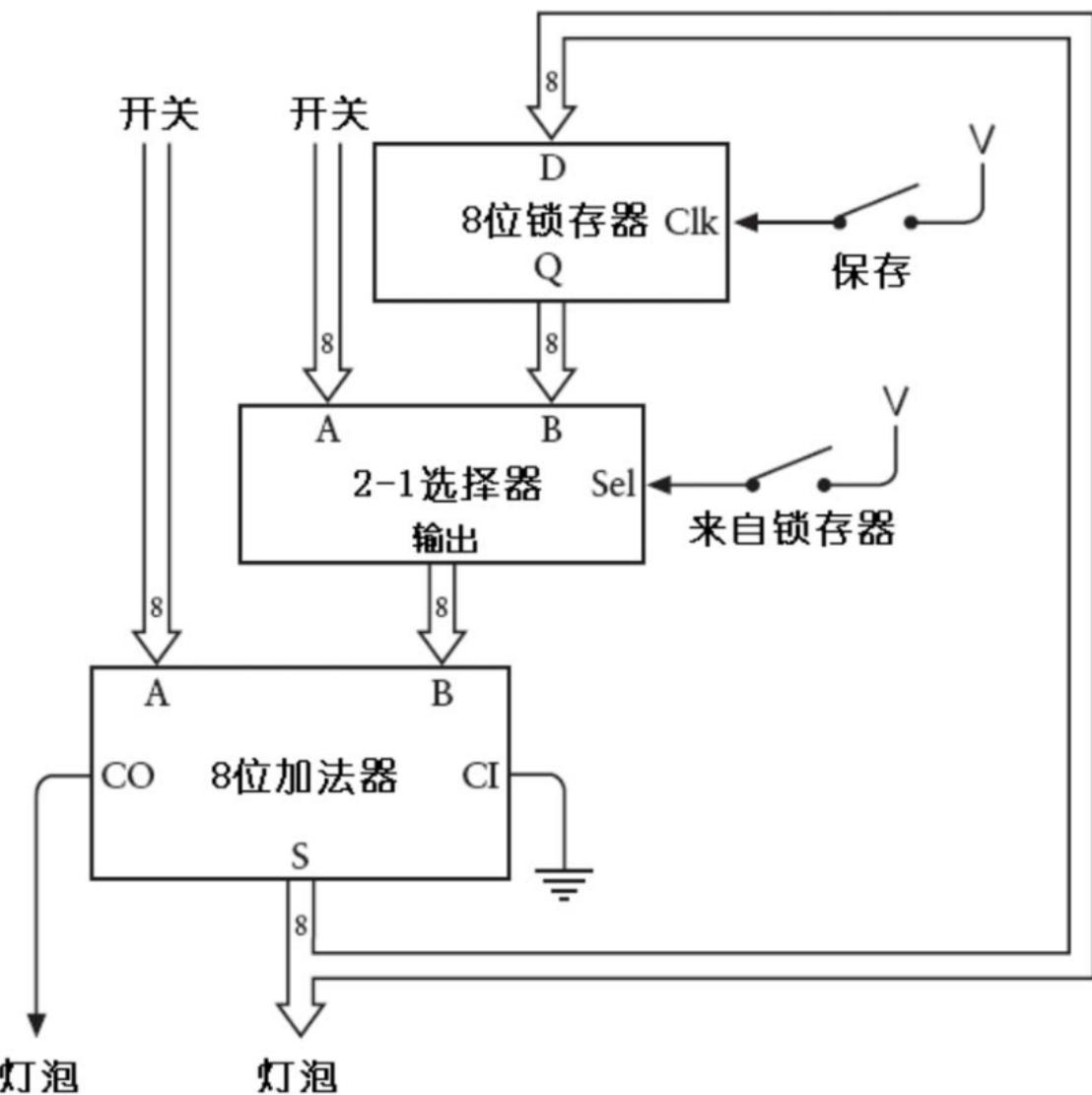
通常把数据端简写为D, 时钟端简写为Clk, 其功能表如下所示。

X表示“其取值情况与结果无关”，只要保持位的值为0，那么数据位对电路的输出没有影响，电路的输出和其前一个状态相同。

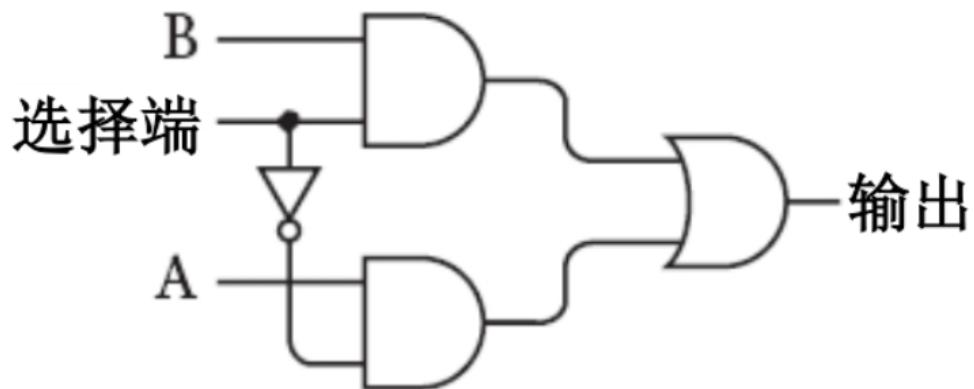
输入		输出	
D	Clk	Q	$\bar{Q}$
0	1	0	1
1	1	1	0
X	0	Q	$\bar{Q}$

这个电路也就是所谓的电平触发的D型锁存器，它表示电路锁存住一位数据并保持它，以便将来使用。这个电路也可以被称为1位存储器。

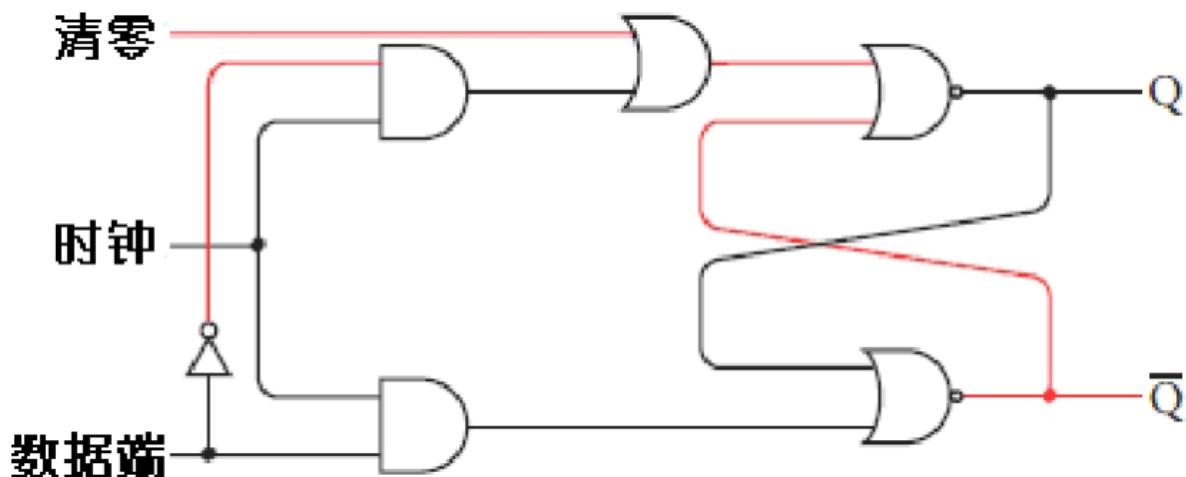
- 我们在一个小盒子里布置8个锁存器，如前所述，每个锁存器包括两个或非门、两个与门以及一个反相器。所有的时钟输入端都互相连在一起。
  - 当时钟信号为1时，D端输入的8位值被送到Q端输出。当时钟信号为0时，这8位值将保持不变，直到时钟信号再次被置1。
  - 经过改进，8位加法器的8个S输出端既与灯泡相连，又连接到8位锁存器的数据（D）输入端。标记为“保存”（Save）的开关是锁存器的时钟输入，用来存放加法器的运算结果。



- 标识为2-1选择器的方块是让你用一个开关来选择加法器的B端输入是取自开关还是取自锁存器的Q端输出。当开关闭合时，就选择了用8位锁存器的输出作为B端输入。改进后的加法器中包含了8个这样的1位选择器。所有的选择端输入信号都是连在一起的。



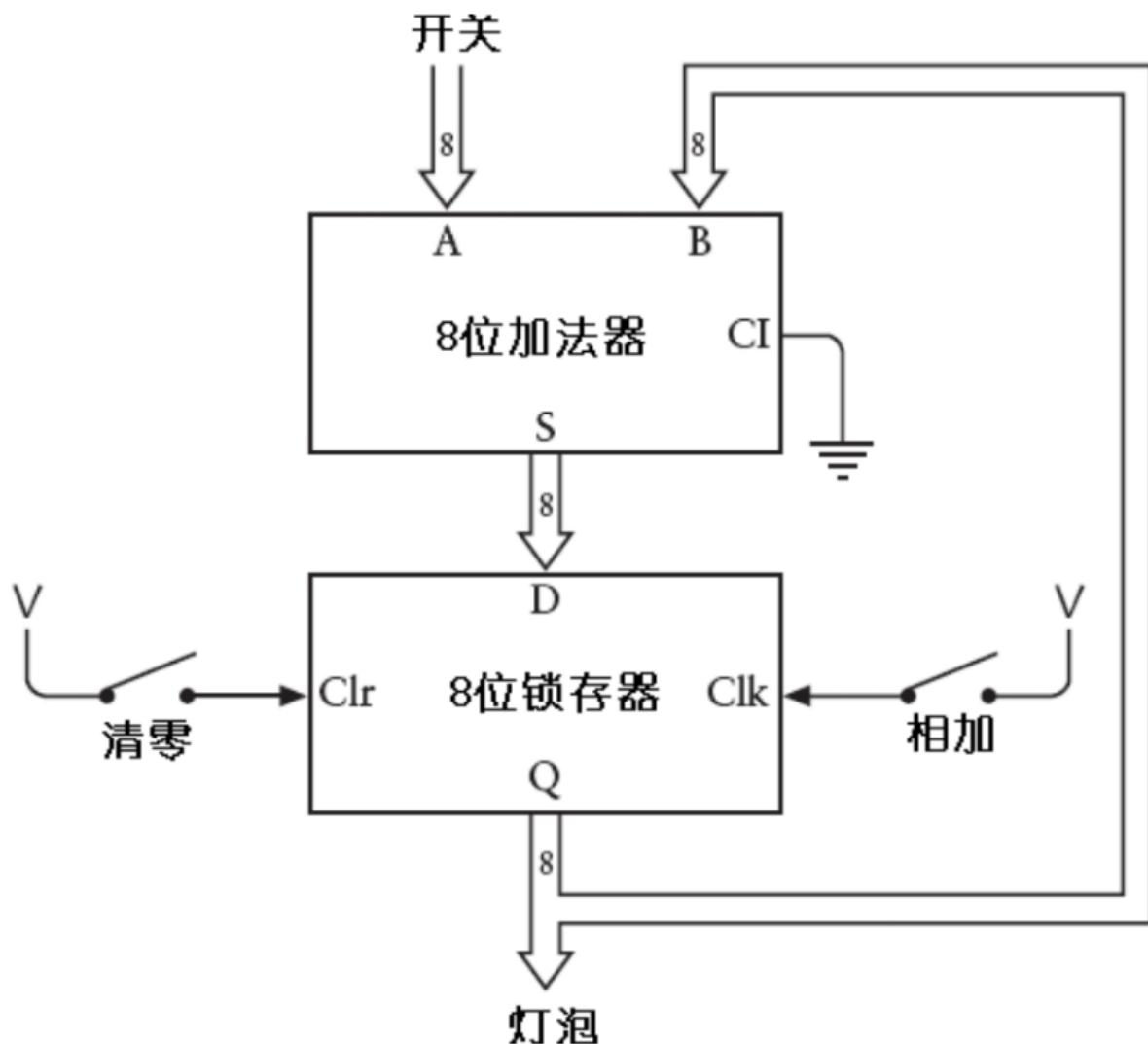
- 改进后的加法器不能很好地处理进位输出 (CO) 信号。如果两个数的相加使得进位输出信号为1，那么当下个数被加进来的时候，这个信号将被忽略掉。一个可能的解决方案是将加法器、锁存器、选择器均设置为16位宽，或者至少应该比你可能遇到的最大的和的位数多一位。这个问题留到第17章具体讲述。
- 对于加法器来说，一个更好的改进方法是去掉一整排8个开关。但是首先要对D触发器做一些修改，为它加一个或门和一个称为清零 (Clear) 的输入信号。清零信号通常为0，但当它为1时，Q输出为0，如下图所示。



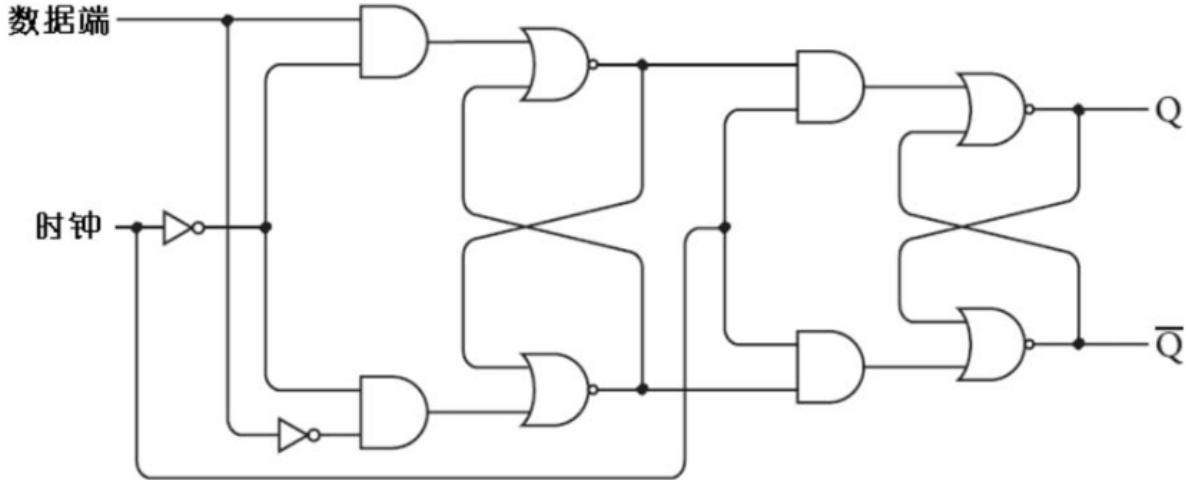
也许你还不明白为什么要设置这个信号，为什么不能通过把数据输入端置0和把时钟输入端置1来使触发器清零呢？这也许是因为我们无法精确控制数据端的输入信息的缘故。

标识为“相加”（Add）的开关现在控制着锁存器的时钟输入。

你可能会发现这个加法器比前面的那个好用，特别是当你需要加上一长串数字时。首先按下清零开关，这个操作会使锁存器的输出为0，并且熄灭了所有的灯泡，同时使8位加法器的第2行输入全为0。然后，通过开关输入第一个加数，并且闭合“相加”开关，这个加数的值就反映在灯泡上。再输入第二个加数并再次闭合“相加”开关。由开关输入的8位操作数加到前面的结果上，所得的和体现到灯泡上。反复如此操作，可以连续进行很多次加运算。



- 前面提到过，我们所设计的D触发器是电平触发的，也就是说为了使数据端的值保存在锁存器中，必须把时钟端的输入从0变为1（即高电平）。但是，当时钟端输入为1时，数据端的输入是可以改变的，这时数据端输入的任何改变都会反映在Q的输出值中。
  - 对某些应用而言，电平触发时钟输入已经足够用了；但是对另外一些应用来说，**边沿触发（edge-triggered）** 时钟输入则更有效。对于边沿触发器而言，只有当时钟从0跳变到1时，才会引起输出的改变。它们的区别在于，在电平触发器中，当时钟输入为0时，数据端输入的任何改变都不会影响输出；而在边沿触发器中，当时钟输入为1时，数据端输入的改变也不会影响输出。只有在时钟输入从0变到1的瞬间，数据端的输入才会影响边沿触发器的输出。
- 边沿触发的D型触发器是由两级R-S触发器按如下方式连接而成的。

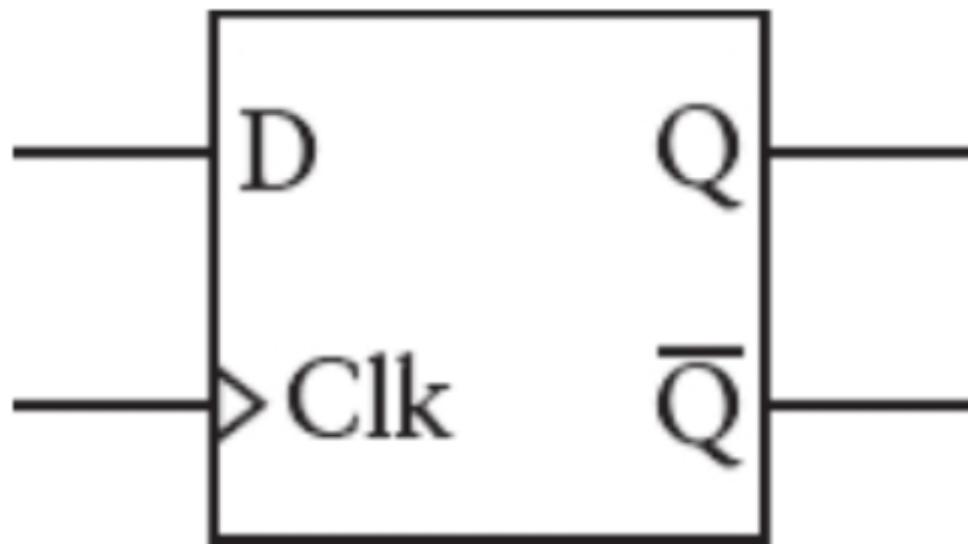


- 时钟端的输入既控制着第一级R-S触发器，也控制着第二级，但是要注意的是时钟信号在第一级中进行了取反操作，这意味着除了当时钟信号为0时保存数据外，第一级R-S触发器和D型触发器工作原理完全一致。第二级R-S触发器的输入是第一级的输出，当时钟信号为1时，它们都被保存。一言概之，只有当时钟信号由0变为1时，数据端输入才被保存下来。
- 只有在时钟输入从0变化到1的瞬间Q输出才发生变化。
- 边沿触发的D型触发器的功能表需要一个新的符号来表示从0到1的瞬时变化，即用一个向上的箭头（↑）表示，如下表所示。

输入		输出	
D	Clk	Q	$\bar{Q}$
0	↑	0	1
1	↑	1	0
X	0	Q	$\bar{Q}$

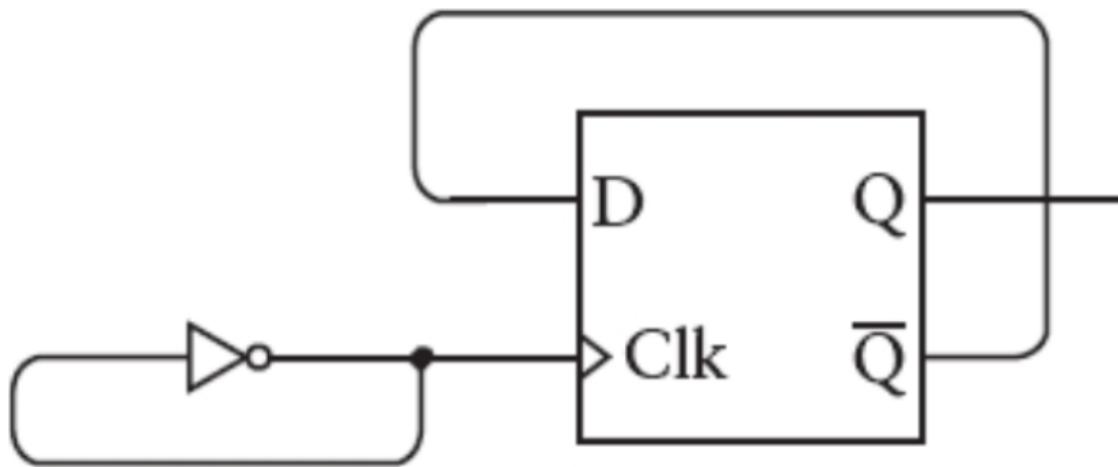
表中箭头表示当时钟端由0变为1时（称为时钟信号的“正跳变”，“负跳变”是指从1变为0），Q端输出与数据端输入是相同的。

触发器的符号如下图所示。



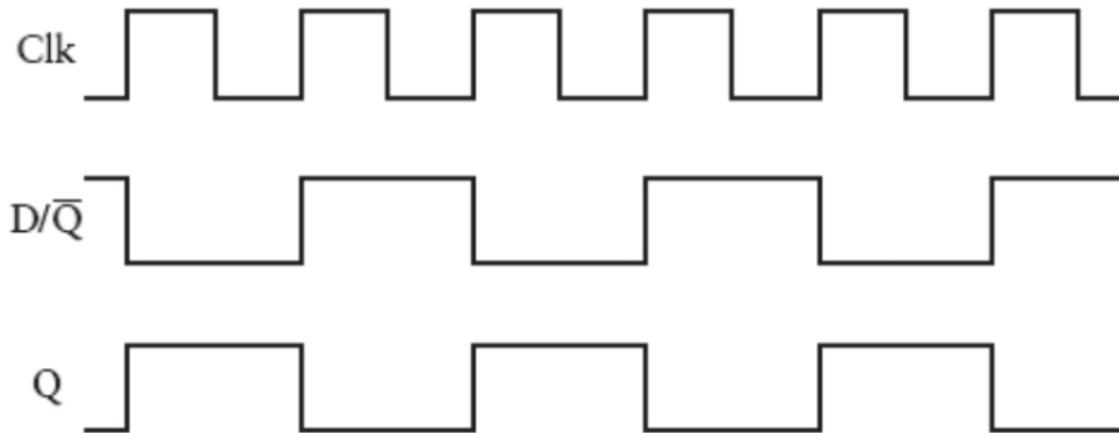
图中的小三角符号表示触发器是边沿触发的。

- 展示一个使用边沿D型触发器的电路，这个电路是不能用电平触发形式复制出来的。
  - 把振荡器的输出与边沿触发的D型触发器的时钟端输入连接，同时把/Q端输出连接到本身的D输入端。



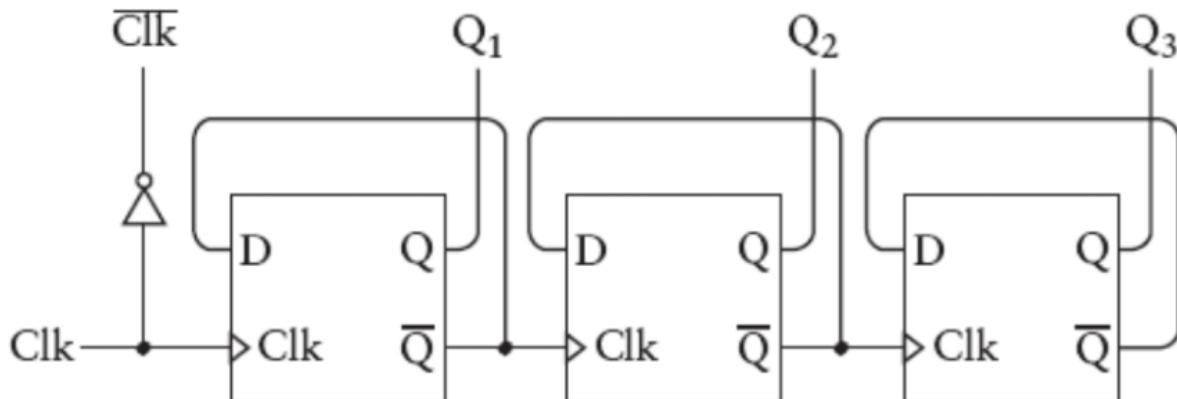
这个触发器的输出同时又是它自己的输入。实际上，这种构造可能是有问题的，振荡器是由状态来回迅速改变的继电器构成的，其输出与构成触发器的继电器相连，而这些其他的继电器不一定能跟得上振荡器的速度。为了避免这些问题，这里假设振荡器中的继电器比电路中其他地方的继电器速度要慢得多。

- 每当时钟输入由0变为1时，Q端输出就发生变化，或者从0到1，或者由1到0。下面的时序图可以更加清楚地说明这个问题。

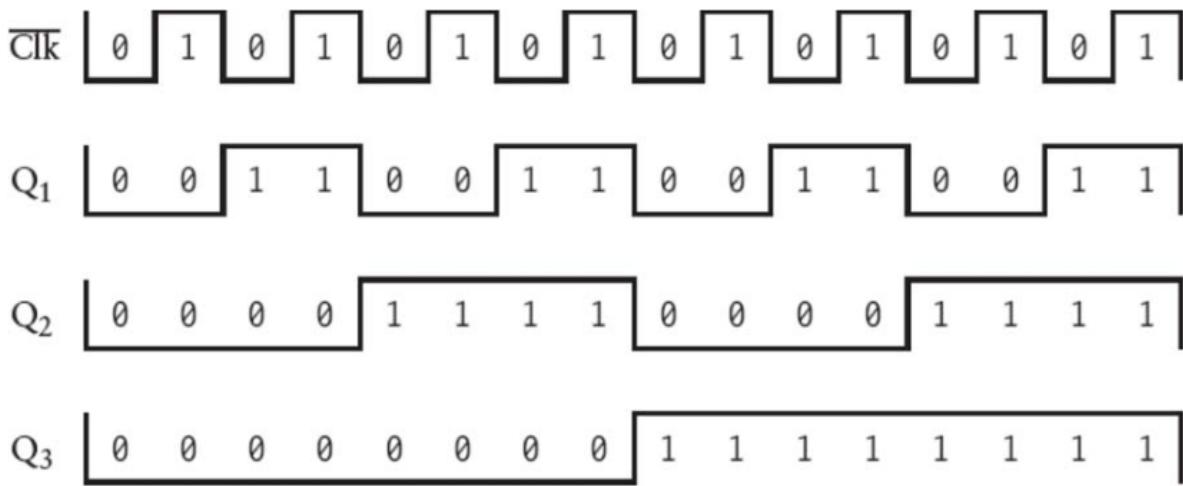


如果这个振荡器的频率是20Hz（即20个周期的时间为1s），那么Q的输出频率是它的一半，即10Hz，由于这个原因，这种电路称为分频器（frequency divider）

- 当然，分频器的输出可以作为另一个分频器的Clk输入，并再一次进行分频。下面是三个分频器连接在一起的示意图。



上图顶部的4个信号变化规律如下图所示。

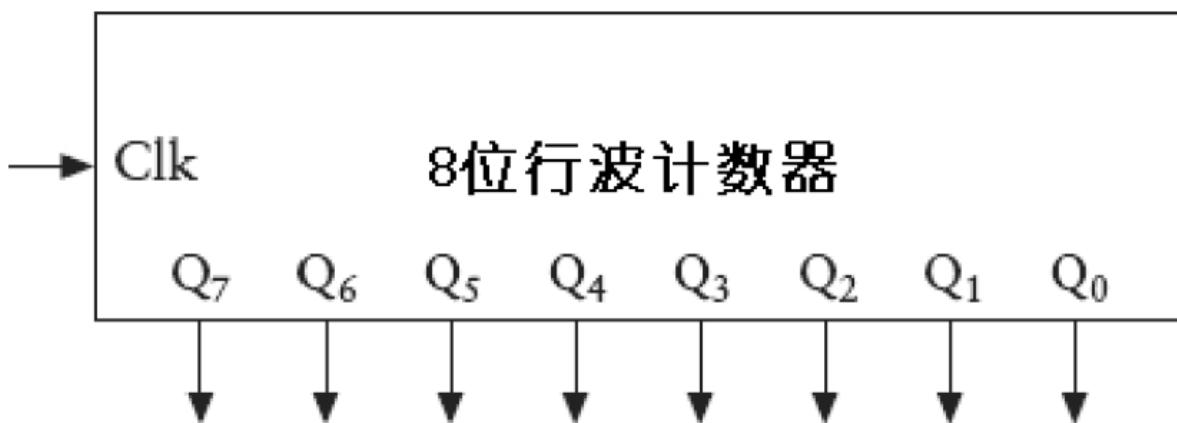


把这些信号标上0和1，顺时针旋转90°，然后读一读每一行的4位数字，它们分别对应了十进制中的0~15中的一个数。

如果在这个电路中添加更多的触发器，其计数范围就会更大。

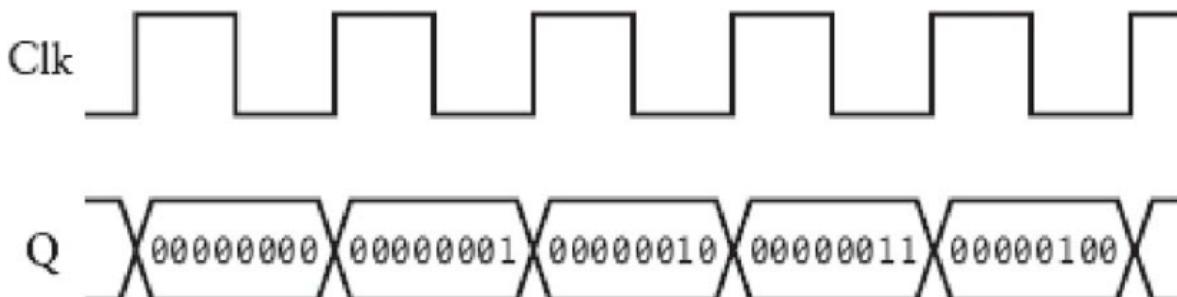
在第8章中提到一个顺序递增的二进制序列，每一列数字在0和1之间的变化频率是其右边那一列数字变化频率的一半，这个计数器就是模仿了这一点。在每一次时钟信号的正跳变时，计数器的输出是增加的，即递增1。

- 把8个触发器连接在一起，然后放入一个盒子中，构成了一个8位计数器。



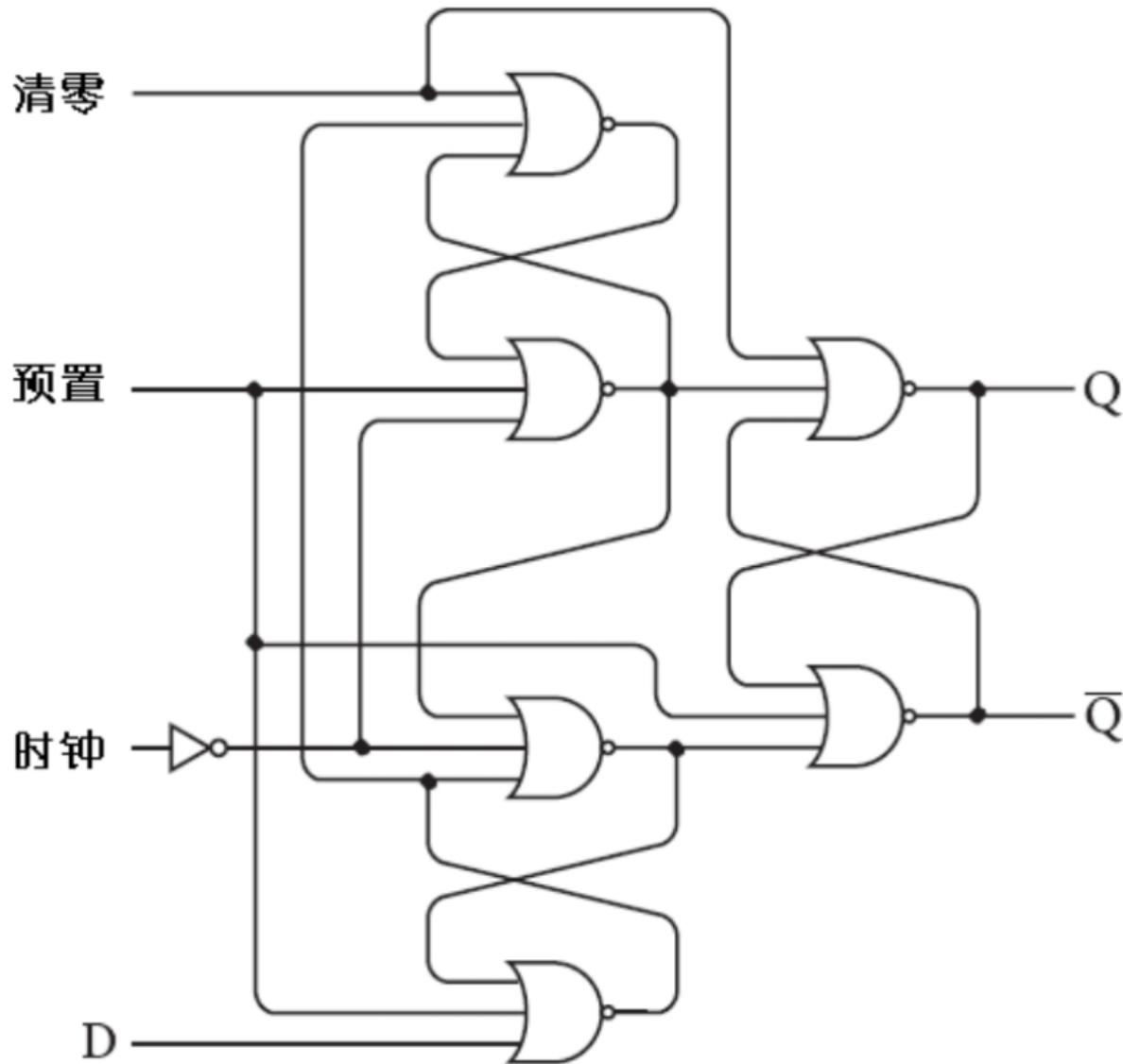
这个计数器称为“8位行波计数器”，因为每一个触发器的输出都是下一个触发器的时钟输入。变化是在触发器中一级一级地顺序传递的，最后一级触发器的变化必定会有一些延迟，更先进的计数器是“并行（同步）计数器”，这种计数器的所有输出是在同一时刻改变的。

在计数器中输出端用Q0~Q7标记，在最右边的Q0是第一个触发器的输出。如果将灯泡连到这些输出端上，就可以将8位数字读出来。这样一个计数器的时序图可以将8个输出分别表示出来，也可以将它们作为整体一起表示出来，如下图所示。

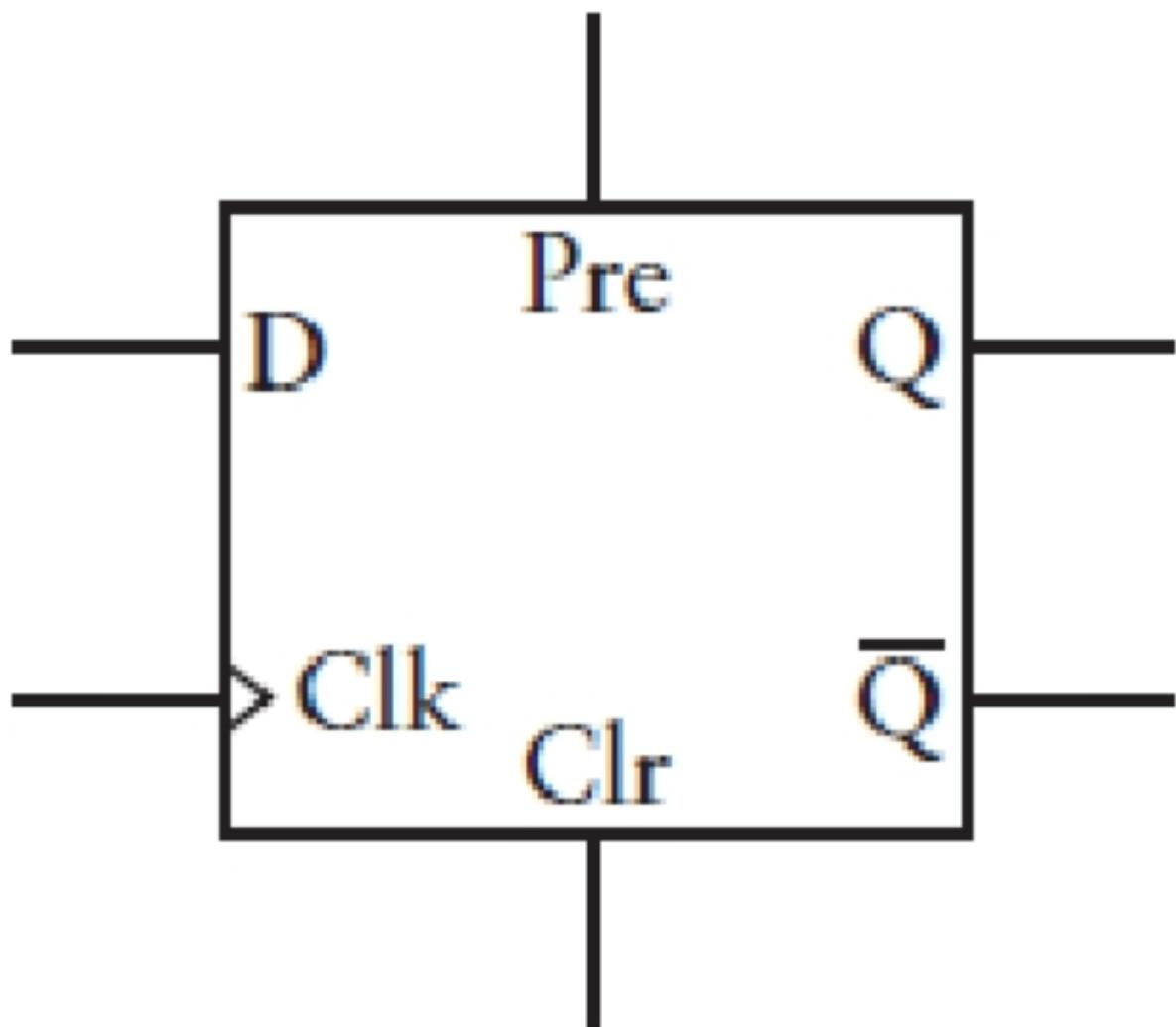


- 本章前面提到过可以找到某种方法来确定振荡器频率，现在已经找到这种方法了。

- 如果把一个振荡器连接到8位计数器的时钟输入端上，那么这个计数器会显示出振荡器经过的循环次数。当计数器总数达到11111111（十进制的255），它又返回为00000000。使用计数器确定振荡器频率的最简单的方法就是把计数器的8个输出端分别接到8只灯泡上。当所有的输出都是0时（即所有灯泡都是熄灭的），启动一个秒表计时；当所有灯泡都点亮时，停止秒表计时。这就是振荡器循环256次所需要的时间。假设这个时间为10s，则振荡器的频率是 $256 \div 10$ ，即25.6 Hz。
- 随着触发器功能的增加，它的结构也变得更加复杂，下面给出了一个带预置和清零功能的边沿型D触发器。



输入				输出	
Pre	Clr	D	Clk	Q	$\bar{Q}$
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	$\bar{Q}$



- 现在，我们已经懂得如何使用继电器来做加法、减法和计数了，这是一件很有成就感的事情，因为我们使用的硬件是100多年前就存在的东西。

## 15 字节与十六进制

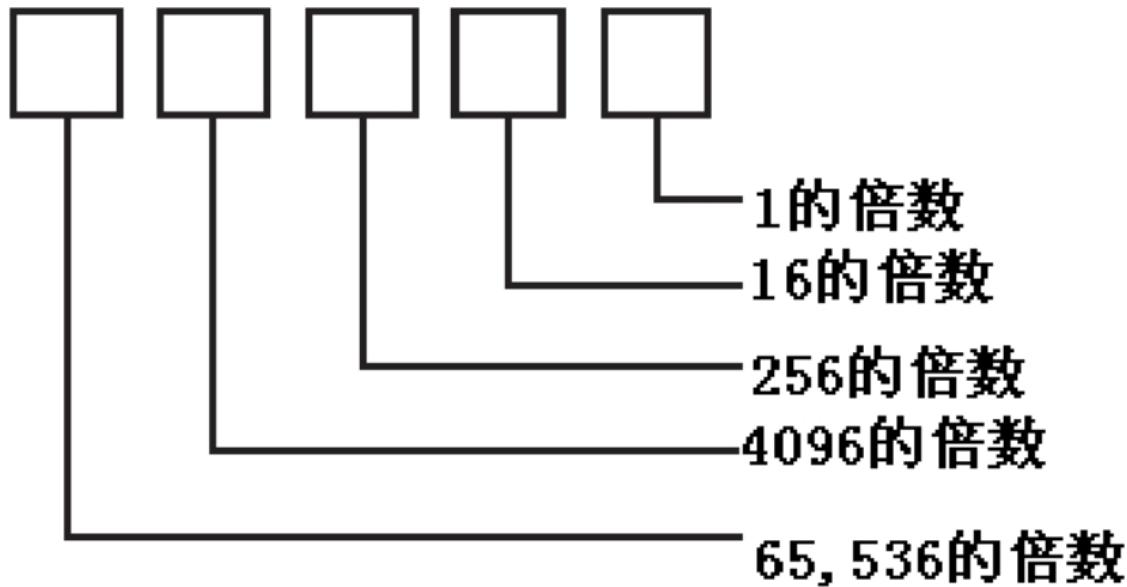
- 通过对上一章中两类改进加法机的剖析，我们学习了**数据路径（Data Path）**这个概念。
  - 纵观整个电路的脉络，每8个比特流为一组，如潺潺溪流般在器件与器件之间流动。
  - 其实，这8位比特流就是加法器、锁存器以及数据选择器的输入形式，同时它也是这些器件单元的输出形式。这8位的比特流可以用开关的不同状态组合所定义，而且可以用灯泡的亮灭来显示。这样一来，这些电路中数据路径的**位宽（bits wide）**就是8。
  - 为什么我们要把它定义为“8”位呢？为什么没有定义为6位、7位、9位或10位呢？
    - 追根究底，原始加法机偏偏是8位其实没有什么特别的原因。只不过在每次使用8位位宽时，一切工作都显得非常方便——一种优雅的比特化（bitful）的比特流。
    - 或许大家已经感觉到我在极力隐瞒一些东西，无法否认，其实我心中一直都很清楚（或许你也知道）：8比特代表一个字节（byte）。
- 字节这个词最早起源于1956年前后，由IBM公司提出。最早的拼写方式是bite，但为了避免与bit混淆用y代替了i。曾几何时，字节仅表示某一数据路径上的位数，直到20世纪60年代中叶，在IBM的360系统的发展下（一种大规模复杂的商用计算机），字节这个词逐渐开始用来表示一组8比特数据。
  - 由于有8位，一个字节的取值范围为00000000到11111111。相应地，它还可以表示成为0~255之间的正整数，如果将一个数的补码作为其相对应的负数，那么一个字节可以表示在-128~127范围内的正、负整数。一个给定的字节可以代表 $2^8$ ，即256种不同事物中的一个。
- 当8作为比特流的一种尺度，它的确表现出了非常完美的特质。字节在很多方面都比单独的比特更胜一筹。IBM采用字节有一个很重要的原因，就是这样一来数字就可以按照BCD形式（第23章中将会讲述）方便地保存。在本章随后的讲述中，我们会发现凑巧的是：全世界大部分书面语言（除了中文、日文以及韩文中使用的象形文字体系）的基本字符数都少于256，所以字节是一种理想的保存文本的手段。字节同样适合表示黑白图像中的灰度值，这是由于肉眼能区分的灰度约为256种。当一个字节无法表示所有信息（如刚提到的中文、日文以及韩文中使用的象形文字体系等），我们只需采用两个字节——就可以表示216也就是65536个不同的物体——这也是一种很好的解决方案。
- 字节的一半——即4比特——我们称之为半字节（nibble，也可拼写成nybble），在计算机这个领域，它并不像字节那样经常使用。
  - 假如一个8位二进制数表示为10110110，这种表达方式自然而又直观，但它还不够简洁。
  - 10110110这个字节很容易就表示为八进制数266。这种方法简洁明了，八进制用来表示字节不失为一个好方法。但还是有那么一点美中不足。
    - 字节可以表示的二进制数的范围为00000000~11111111，如果采用八进制表示，那么相对应的范围也随之变成了000~377。仔细分析下先前的例子，我们从右到左把3位二进制数对应于中间以及最靠右的八进制数，而最靠左的八进制数却是由2位二进制数对应的。
    - 如果我们把这个16位二进制数平分为两个字节并将其分别表示为八进制数会得到不同结果。
- 为了使多字节值能和分开表示的单字节值取得一致，我们需要一种可以等分单个字节的系统，按照这种思想，我们可以把每个字节等分成4组，每组2比特（基于4的计数系统）；还可以等分为2组，每组4比特（基于16的计数系统）。

- 基于16的计数系统 (Base 16)，对于我们而言是一种全新的系统。基于16的计数系统也被称为十六进制 (hexadecimal)
  - 这个单词很容易让我们产生混淆。这是因为很多以hexa-为前缀的单词（比如Hexagon, Hexapod以及Hexameter）都会与6或多或少有些联系。而hexadecimal这个词却偏偏代表了16 (Sixteen) 这个含义。虽然在《微软出版物风格与技术手册》 (The Microsoft Manual of Style for Technical Publications) 中明确说明“请勿将十六进制缩写为hex”，但包括我在内的绝大多数人还总是在不经意间使用这种缩写。

二进制	十六进制	十进制
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

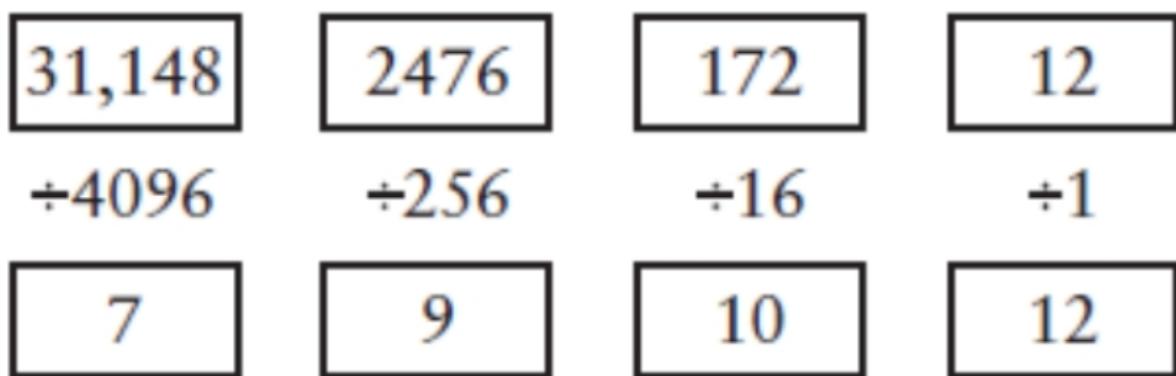
- 这样一来A字节10110110就可以表示为十六进制的B6
  - B6SIXTEEN (B6<sub>16</sub>) 还可以用如下方法表示：B6<sub>HEX</sub>
  - 在本书中将使用一种更加简洁实用的方法，那就是用一个小写的h紧跟在数字后边表示这个数是以十六进制表示的，就像这样：B6h

- 通过分析可以得到十六进制数的每一位代表16的不同整数幂的倍数，如下图所示。



- $9A48Ch = 9 \times 10000h + A \times 1000h + 4 \times 100h + 8 \times 10h + C \times 1h$
- $9A48Ch = 9 \times 16^4 + A \times 16^3 + 4 \times 16^2 + 8 \times 16^1 + C \times 16^0$
- $9A48Ch = 9 \times 65536 + A \times 4096 + 4 \times 256 + 8 \times 16 + C \times 1$

- 十进制数转换为十六进制数通常涉及除法运算。



还有一种转换小于65,535的十六进制数的方法，首先我们把原数通过除以256的方式将其分为两个字节。接下来对于每个字节，再分别除以16。

51,966

÷256

202

254

÷16

÷16

12

10

15

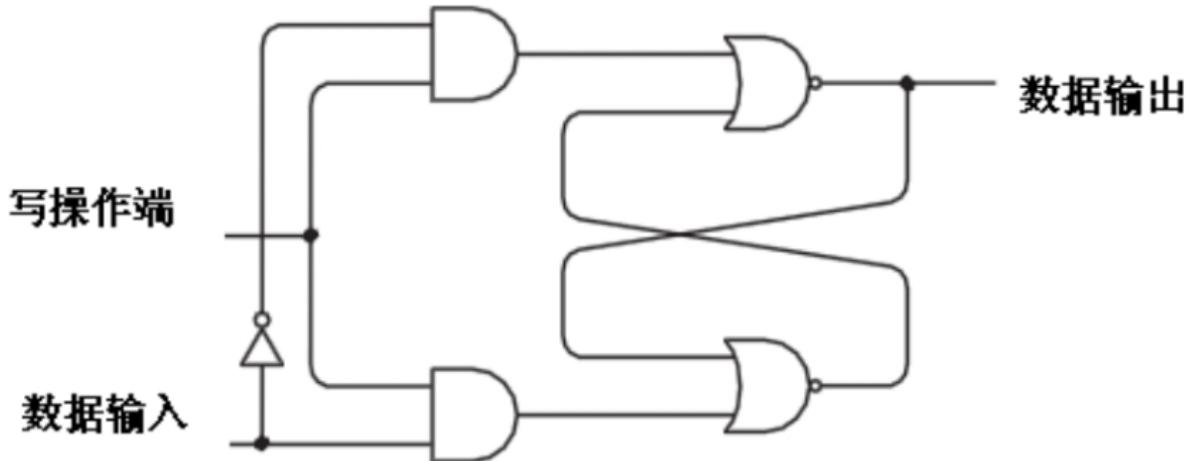
14

- 我们讨论过可以用一个2的补数来表示与其相对应的负数。如果我们处理的是带符号的8位二进制数，那么所有负数的最高位都为1。在十六进制系统中，最高位为8、9、A、B、C、D、E或F的两位带符号数都是负数，因为这些十六进制数对应的二进制数的最高位为1。例如99h可以表示无符号的十进制数153（你必须清楚它是单字节的无符号数），也可以表示十进制的-103（这时它被看做有符号数）。

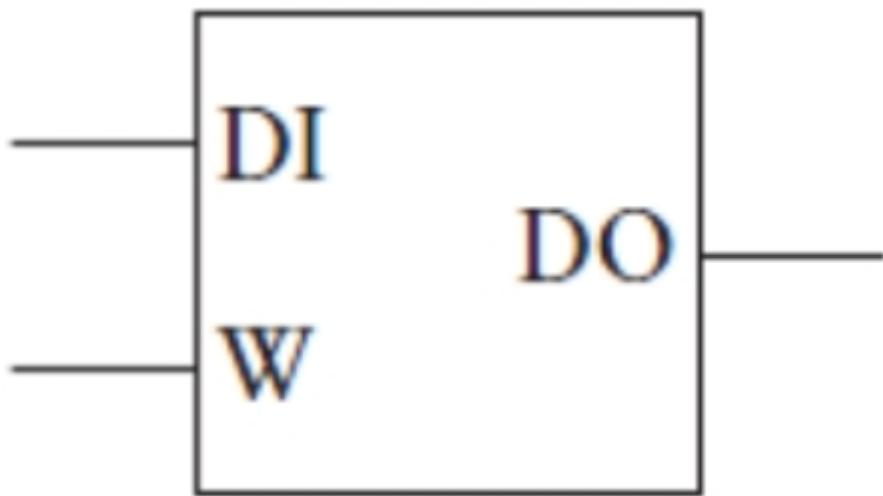
## 16 存储器组织

- 每天清晨，我们将自己从沉睡中唤醒，这时大脑的空白会很快被记忆填充。我们立刻会意识到自己身在何方，最近做了些什么事情，有什么计划和打算。有的事情我们很快就能想起来，但有时，我们大脑处于失忆状态，有那么几分钟发现自己什么都想不起来，但总的来说，我们总是能够与自己的过去保持足够的连续性，继续新的生活，展开人生新的一页。人类的记忆也并非能面面俱到。书面记录这种技术的引入，从某种层面来讲，就是为了弥补人类记忆容易遗漏这一缺陷。
- 我们总是将需要记住的内容事先记下来，在需要时拿出来阅读；习惯于将可能用到的事物先存起来，在需要时将它们取出。从技术角度来讲，这个过程称为先存储后访问。存储器的职责和作用就在于此，它负责保障这两个过程之间信息完好无损。我们每次存储信息都要利用不同种类的存储器。比如，保存文本信息的不二之选就是纸张，而磁带则更适于存储音乐和电影。
- 电报继电器（Telegraph Relays）——以一定形式组织起来构成逻辑门，然后再形成触发器——同样具备保存信息的能力。
  - 在前面章节中我们讨论过，一个触发器可以对1位信息进行存储。这样的存储能力要存储一大堆的信息还远远不够，但它却为我们达到目标迈出了坚实的一步。其实知道了如何存储1位信息，很容易就可以想象出如何存储2位、3位或更多位信息。

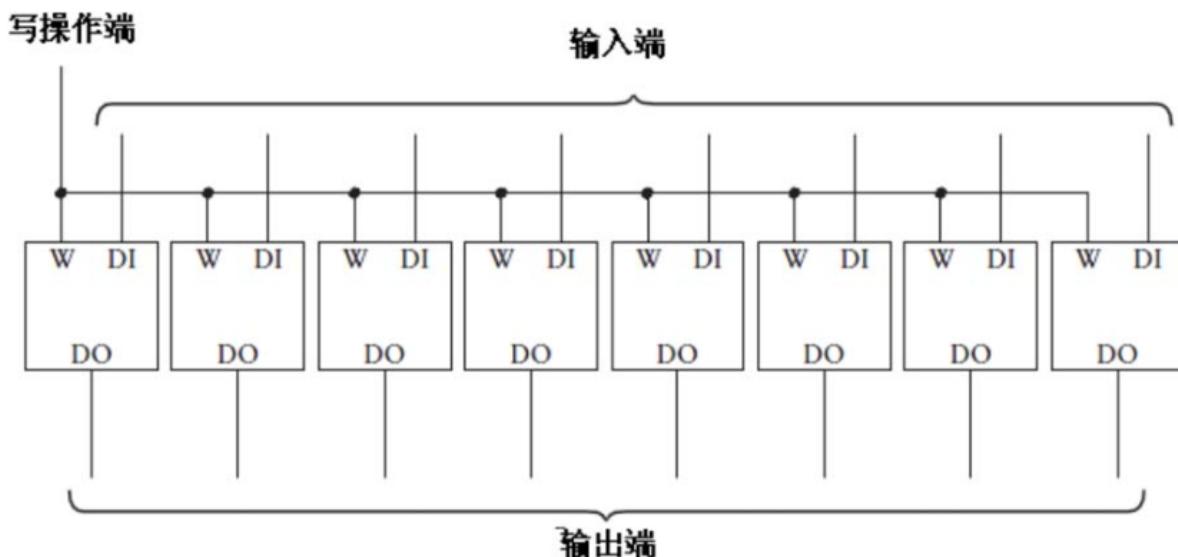
- 从结构上来讲，这套电路与先前所学到的是同一种触发器，只是命名的方式不尽相同，现在Q输出端被称为数据输出端（Data Out），时钟输入端（在第14章叫做保持位）命名为写操作端（Write）。



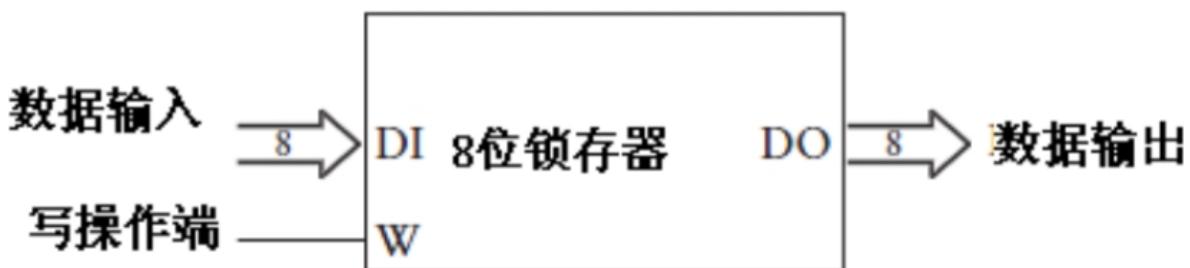
- 就像信息可以被记录在纸上一样，写操作端的信号同样使得数据输入（Data In）信号被写入（Written Into），也可以称之为被存储（stored）到电路中。
- 一般情况下，如果写操作端为0，则数据输入信号的状态对输出无影响。而当我们想把数据输入信号存储在触发器中时，可以把写入信号应先置1后置为0。
- 这种类型的电路也被称为锁存器，因为存储进去的数据就好像被锁住了一样。下面给出了1位锁存器简化框图，框图未画出其内部结构中的部件。



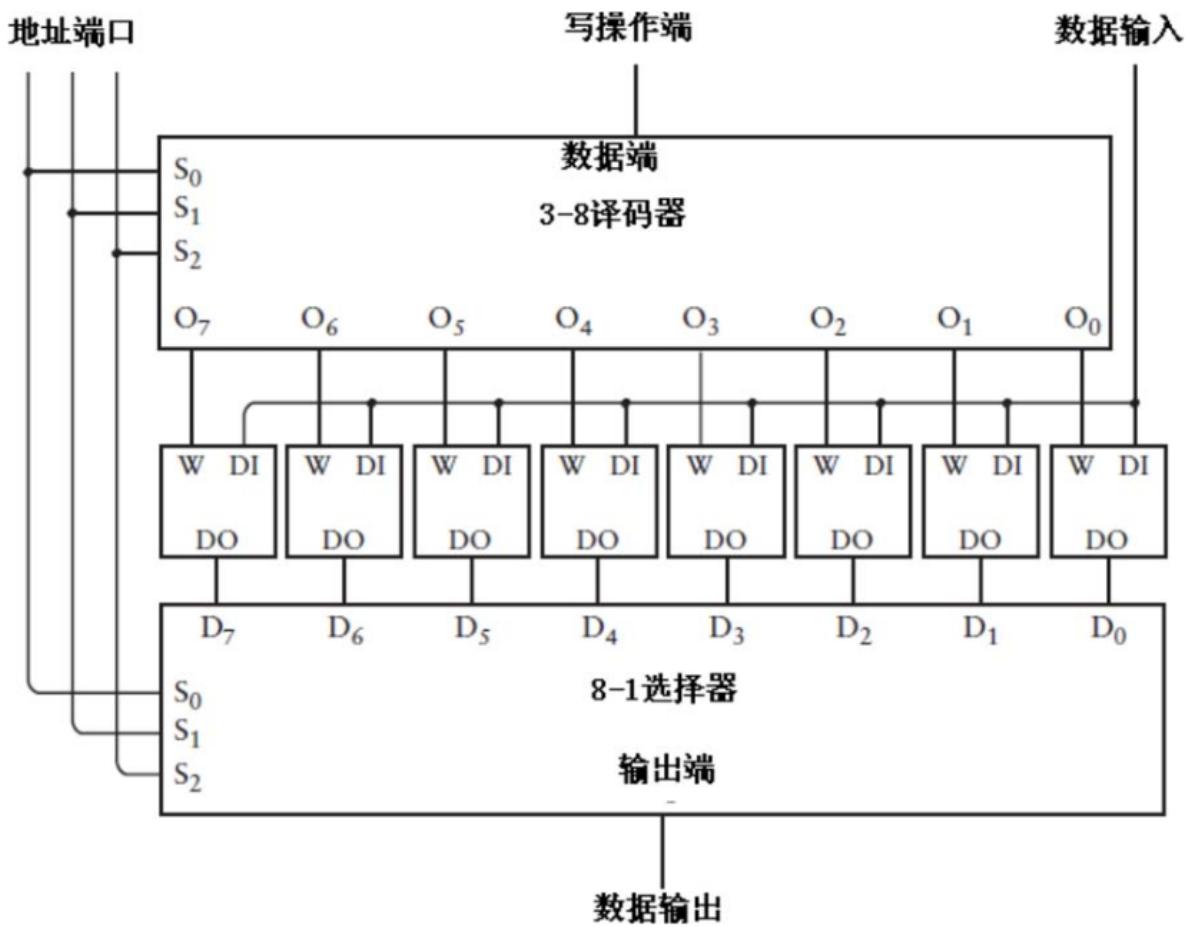
- 很容易想到如何把多个1位锁存器组织成为多位锁存器，所要做的就是把写操作端的信号连接到系统中，就像下面这样。



图中显示的8位锁存器其输入和输出端各有8个。另外还包括一个写操作端，在非工作状态下一般为0。如果要把一个8位二进制数存储在锁存器中，首先要把写操作端置1，然后置0。



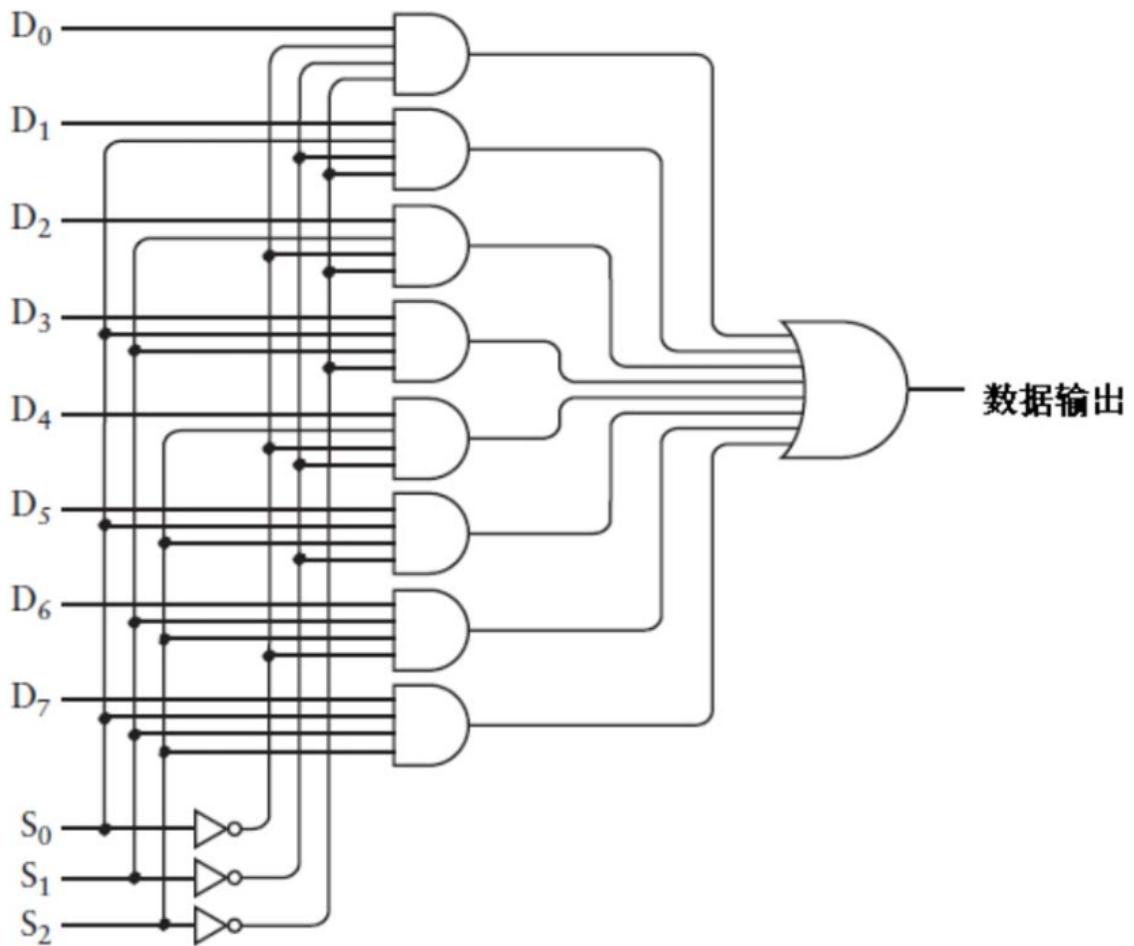
- 还有另一种方法集成8个1位锁存器。假设我们只想用一个数据输入和输出信号端，而且希望锁存器能将输入信号数据分8次独立存储。换句话说，在这种锁存器中我们只想存储8个单独的比特，而不是存储1个8位二进制数。



- 值得注意的是，译码器和选择器具有相同的选择信号，在图中这三个信号一起被称为地址端口 (Address)。
- 在3-8译码器的输入端，地址起到了决定哪些锁存器可以被写操作端的信号触发来保存数据的作用。在输出端（图的下半部分），8-1选择器通过地址来选择8个锁存器中的一个，最后将其输出。
- 8-1数据选择器 (8-Line-to-1-Line Data Selector)

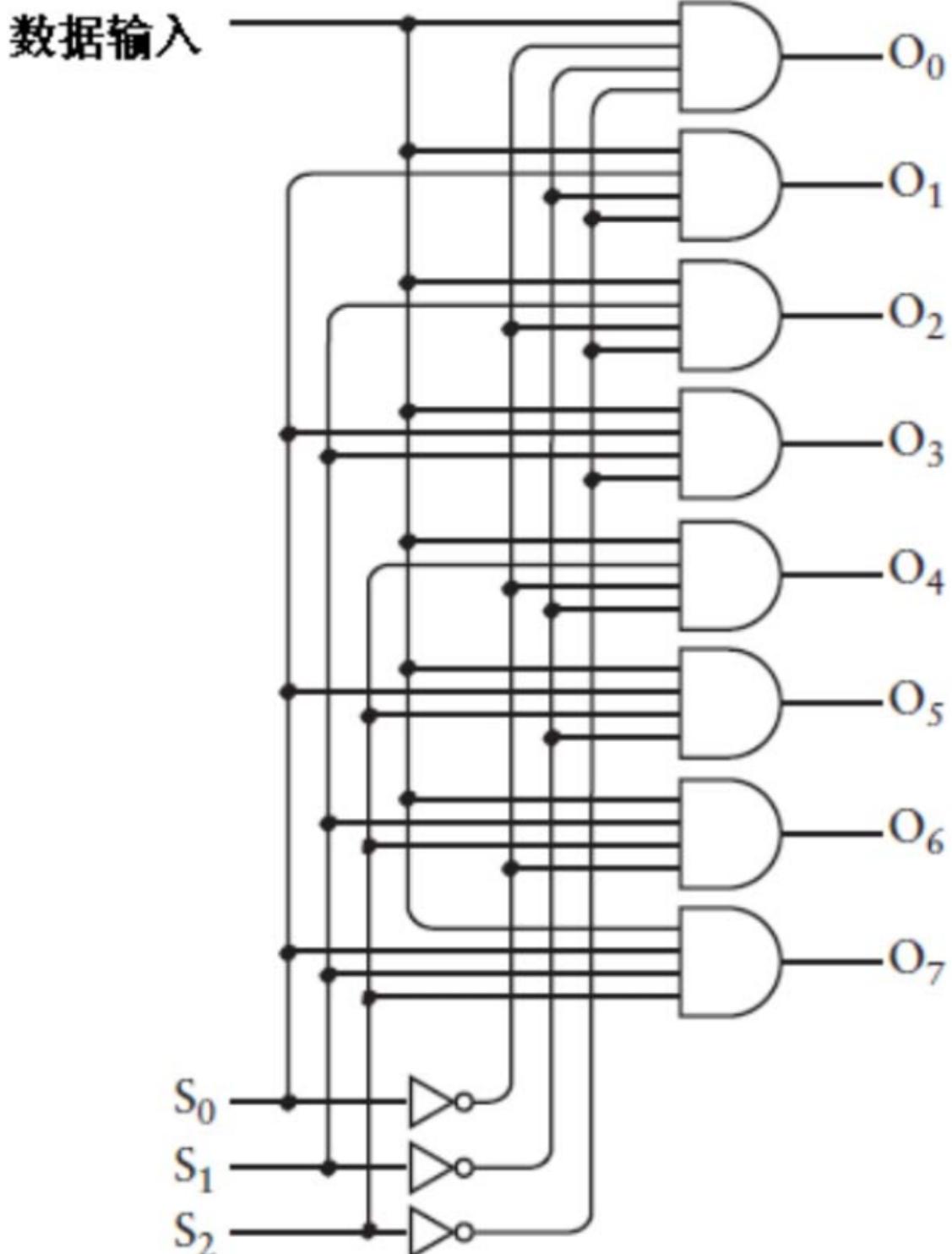
输入			输出
$S_2$	$S_1$	$S_0$	$Q$
0	0	0	$D_0$
0	0	1	$D_1$
0	1	0	$D_2$
0	1	1	$D_3$
1	0	0	$D_4$
1	0	1	$D_5$
1	1	0	$D_6$
1	1	1	$D_7$

8-1选择器主要组成部件为：三个反向器、八个4端口输入与门、一个8端口输入或门，系统的组织结构如下图所示。



这个电路看上去线路密布，要理解它是如何工作的，最好方式就是一起来看一个例子。假设 $S_2$ 初始化为1， $S_1$ 初始化为0， $S_0$ 初始化为1。从顶部开始的第6个与门的输入由 $S_0$ 、 $/S_1$ 、 $S_2$ 组成，初始状态下它们全为1。其余与门的这三项输入数据都与第6个与门不尽相同，这使得其余与门输出全部为0。若 $D_5$ 变为0意味着第6个与门输出为0；反之第6个与门输出则为1。对最右边的或门也可以按照同样的方式理解。我们可以总结出下面这个结论：若选择端为101，则数据输出端与 $D_5$ 的输出保持一致。

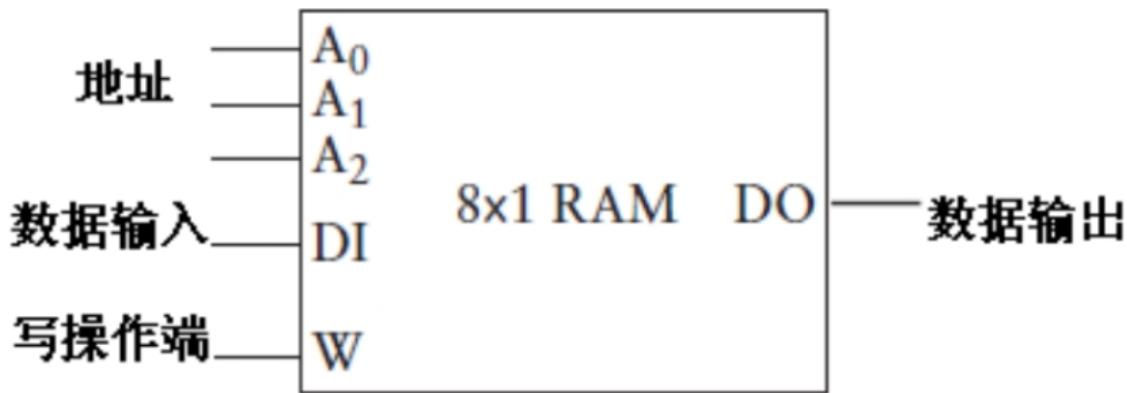
- 可以把所有数据输入信号在锁存器的输入端连接在一起。但8个写入信号是不可以连在一起的，因为我们很可能要向每个锁存器依次写入数据。除此之外还需要一个独立的写入信号，它能被路由到任意（且唯一）的锁存器上。我们需要另外一款电路元件，3-8译码器（3-to-8 Decoder）。
  - 前面的章节中我们曾学习过一个简易的数据译码器（Data Decoder）——在第11章中为了选择喜欢的猫咪，我们把开关以一定方式进行连接使其具有选择功能。



注意从上往下数的第6个与门，它的输入包括 $S_0$ 、 $\neg S_1$ 、 $S_2$ 。没有任何一个与门具有和它相同的三个输入。在这种情况下，如果选择输入端为101，则除了 $O_5$ 要根据情况进行判定外，其余与门输出都为0。这个时候，若数据端输入为0，则 $O_5$ 随之输出为0；相应的，若数据端输入为1，则 $O_5$ 输出为1。译码器的逻辑表可以如下表所示。

输入			输出							
$S_2$	$S_1$	$S_0$	$O_7$	$O_6$	$O_5$	$O_4$	$O_4$	$O_2$	$O_1$	$O_0$
0	0	0	0	0	0	0	0	0	0	Data
0	0	1	0	0	0	0	0	0	Data	0
0	1	0	0	0	0	0	0	Data	0	0
0	1	1	0	0	0	0	Data	0	0	0
1	0	0	0	0	0	Data	0	0	0	0
1	0	1	0	0	Data	0	0	0	0	0
1	1	0	0	Data	0	0	0	0	0	0
1	1	1	Data	0	0	0	0	0	0	0

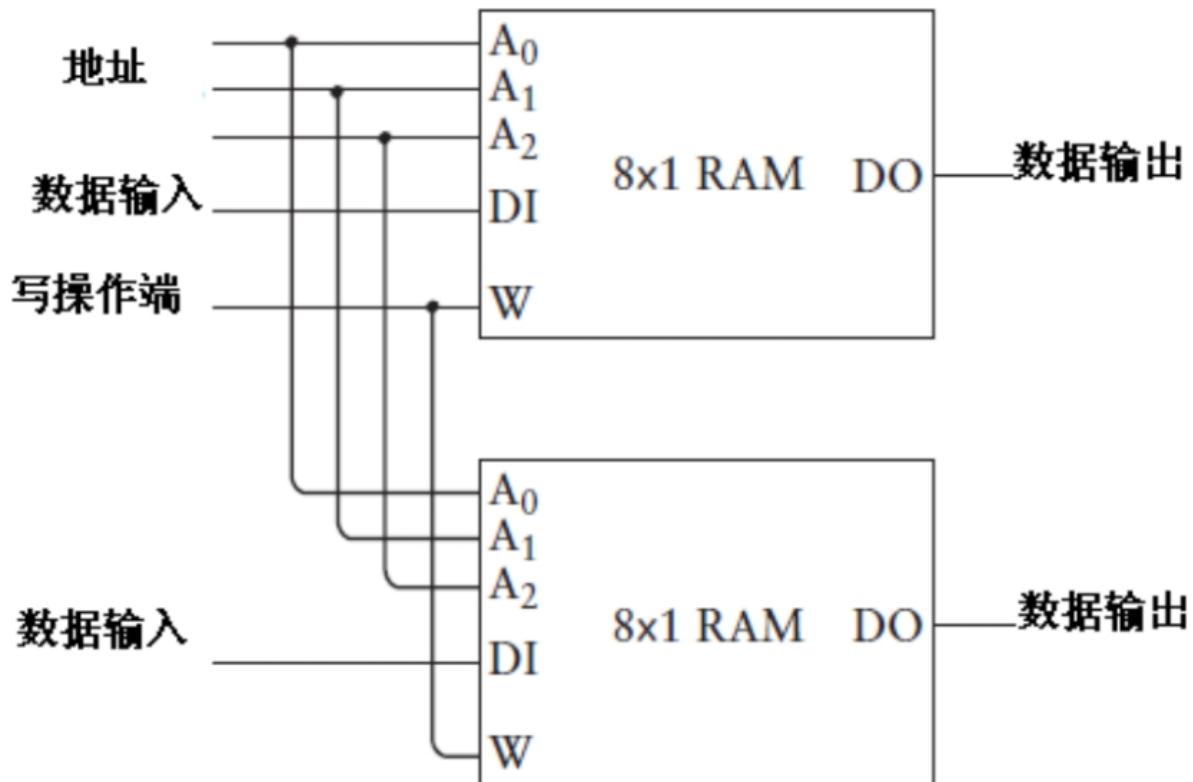
- 这种配置下的锁存器在有的资料中也被称为读/写存储器 (read/write memory)，但更普遍的叫法是**随机访问存储器 (Random Access Memory)**，或RAM。可以认为我们讨论的这种存储器是可存储8个独立比特的RAM，它的简化结构图如下所示。



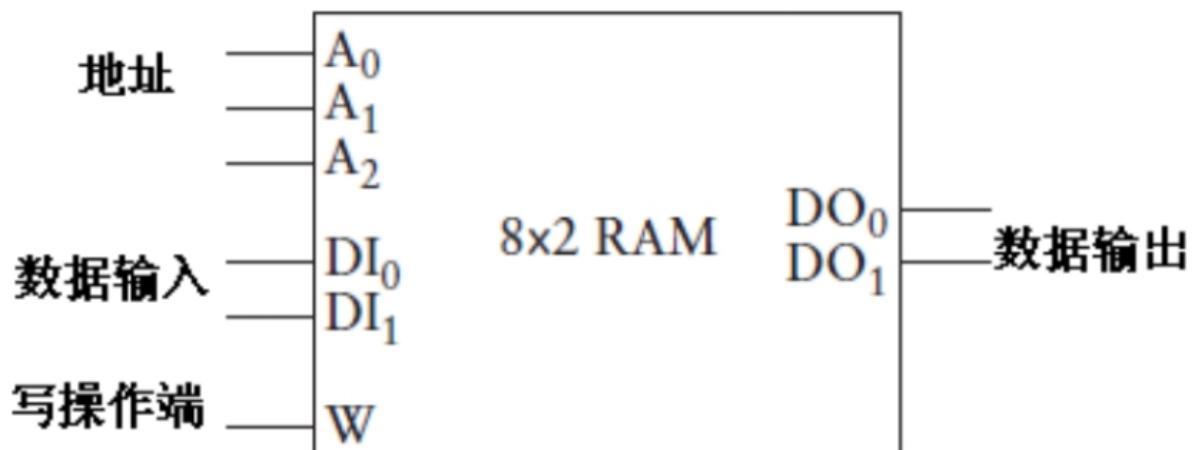
上图所示的电路之所以能够被称为存储器是因为它可以保存信息。而能够被称为读/写存储器是因为它不仅可以在每个锁存器中存储新的数据（可以把这种功能称为写数据），而且我们还可以检查每个锁存器都保存了什么数据（可以把这种功能称为读数据）。

之所以可以被称为随机访问存储器，是因为读写操作很自由，我们只需要改变地址及相关的输入，就可以从8个锁存器中读出或写入需要的数据。相反，一些其他类型的存储器必须按顺序读取——也就是说，如果想要读取地址为101的数据，你将不得不先把地址为100的数据读出来。

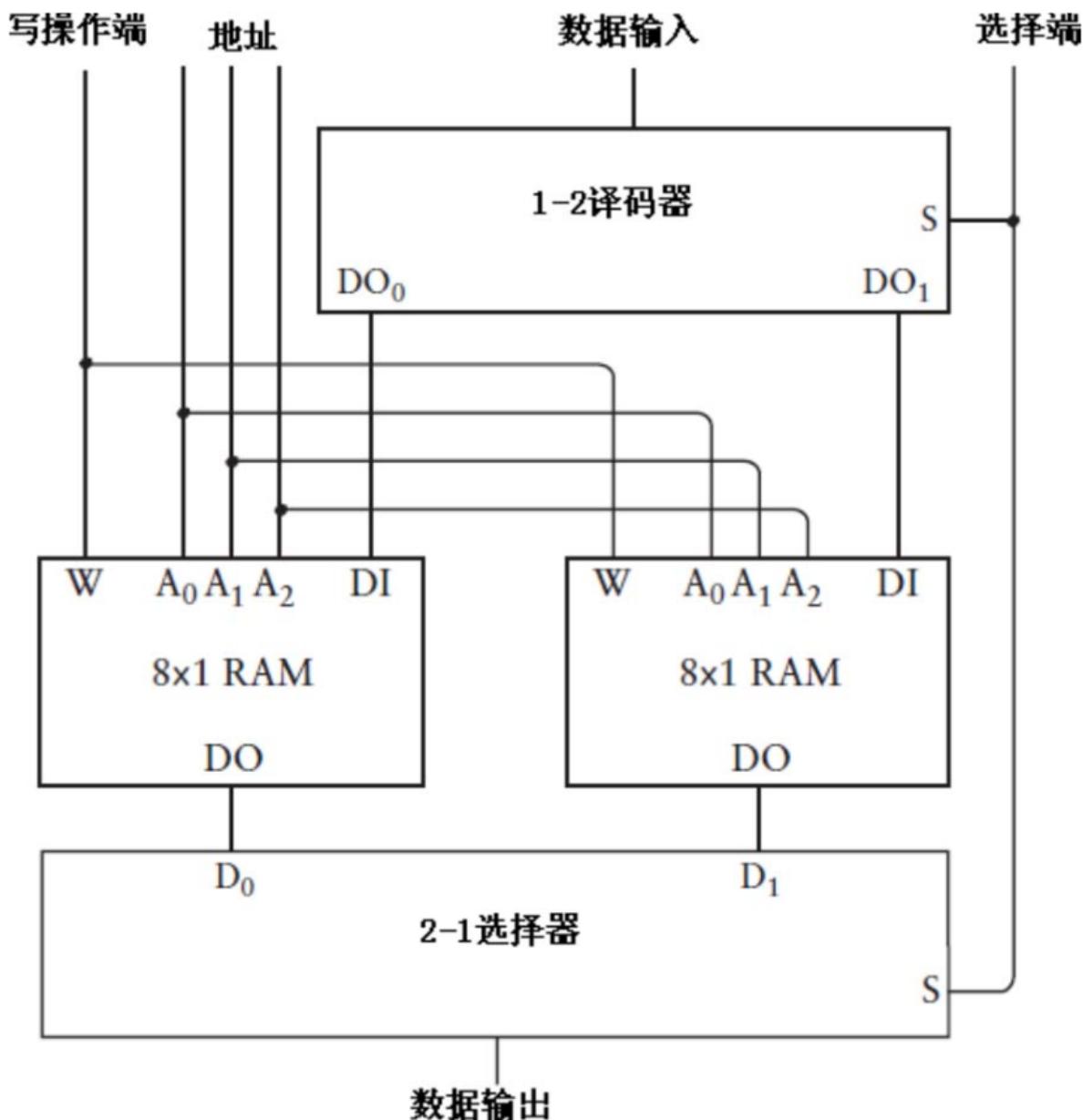
- 将RAM进行特殊的配置可形成RAM阵列 (Array)，我们所讨论的这种RAM阵列以8×1 (读做8乘1) 的方式组织起来。阵列以1比特作为存储单位，共存储8个单位的数据。所以这个RAM阵列中能存储的位数等于8与1的乘积。
- RAM阵列的组合形式多种多样。比如我们可以通过共享地址的方式可以把两个8×1的RAM阵列连接起来，如下图所示。



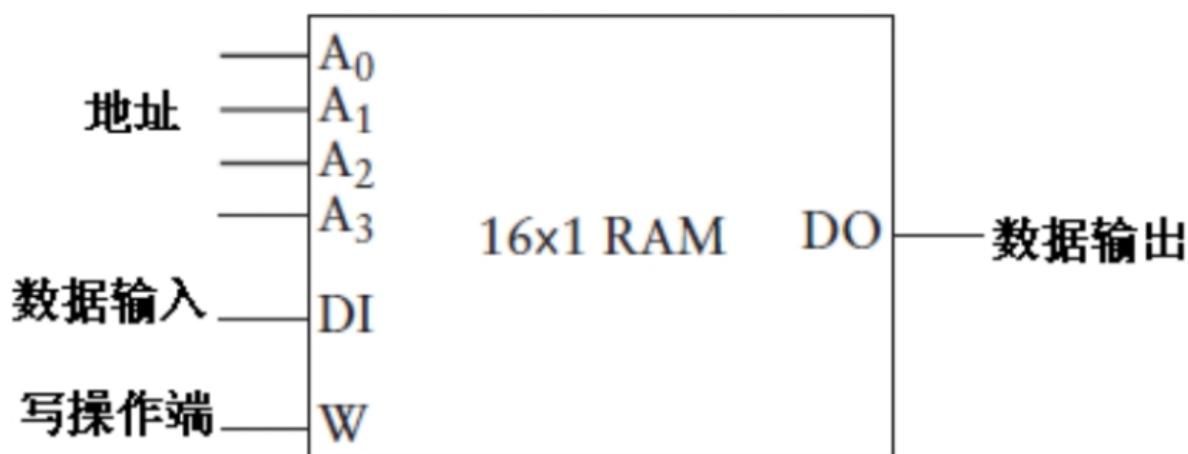
我们把这两个 $8 \times 1$ 的RAM阵列的地址和输出都分别看成一个整体，这样就得到了一个 $8 \times 2$ 的RAM阵列，如下图所示。



- 我们还可以把两个 $8 \times 1$ 的RAM阵列看做是两个锁存器，使用一个2-1选择器和一个1-2译码器就可以把它们按照单个锁存器连接方式进行集成，下面给出了这种方案的电路图。



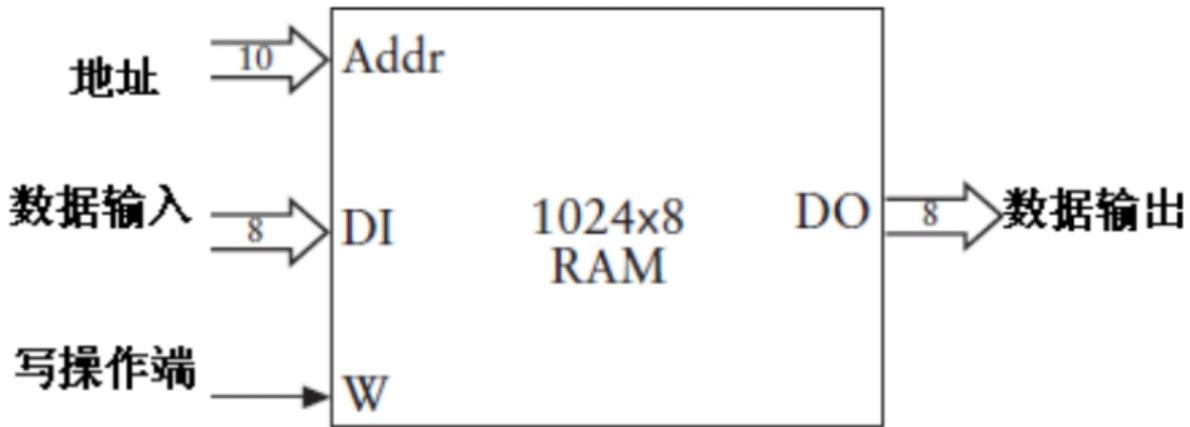
“选择”端之所以连接到译码器和选择器，主要作用是在两个 $8 \times 1$ RAM阵列中选择一个，本质上它扮演了第4根地址线的角色。因此这种结构实质上是一种 $16 \times 1$ 的RAM阵列，如下图所示。



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：

- RAM阵列的存储容量 =  $2^{\text{地址输入端的个数}}$
- 下图所示的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 $2^{10}$ 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。



- 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。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： $2^{10} \approx 10^3$
  - 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。

$$1 \text{ MB} = 1,048,576 \text{ B} = 2^{20} \text{ B} \approx 10^6 \text{ B}$$

$$2 \text{ MB} = 2,097,152 \text{ B} = 2^{21} \text{ B}$$

$$4 \text{ MB} = 4,194,304 \text{ B} = 2^{22} \text{ B}$$

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

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

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

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

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

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

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

$$1,024 \text{ MB} = 1,073,741,824 \text{ B} = 2^{30} \text{ B} \approx 10^9 \text{ B}$$

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。

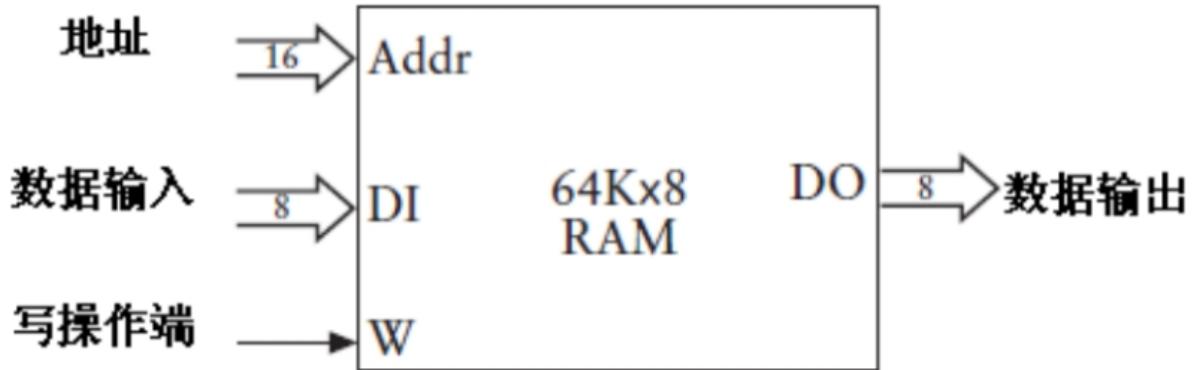
同理，1太字节（terabyte，teras希腊语意思为巨人）表示240个字节（约为 $10^{12}$ ），也就是1,099,511,627,776个字节，太字节的缩写为TB。

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 $10^{18}$ 。

- 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。
  - 100M宽带指的是其数据处理速度为100M比特每秒，等于100/8M字节每秒

- 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。



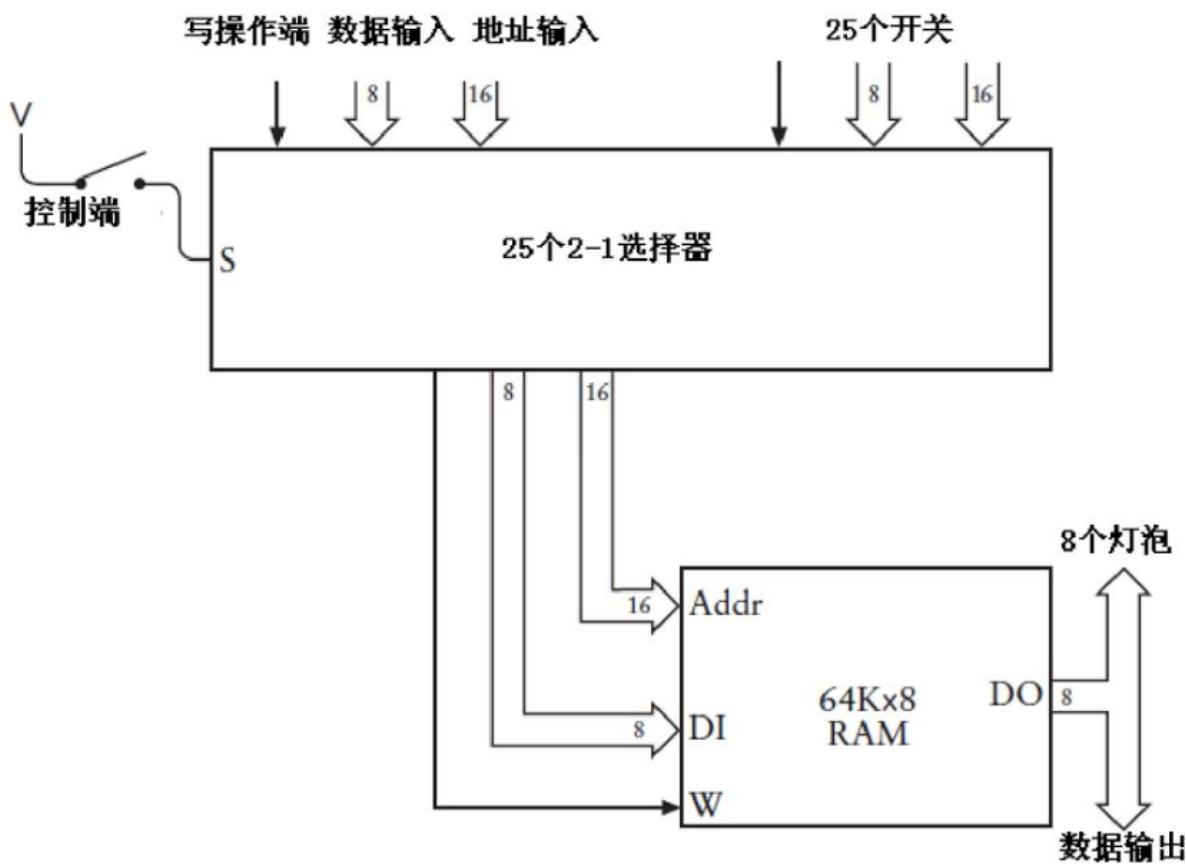
为什么选择大小为64 KB的RAM阵列？而非32 KB或128 KB？因为65,536是一个约整数，转换为幂的形式就是 $2^{16}$ ，这个RAM阵列需要配备16位的寻址端。换句话说，该地址恰好可以用2个字节表示。将地址范围转化为十六进制就是 0000h ~ FFFFh。

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。

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。

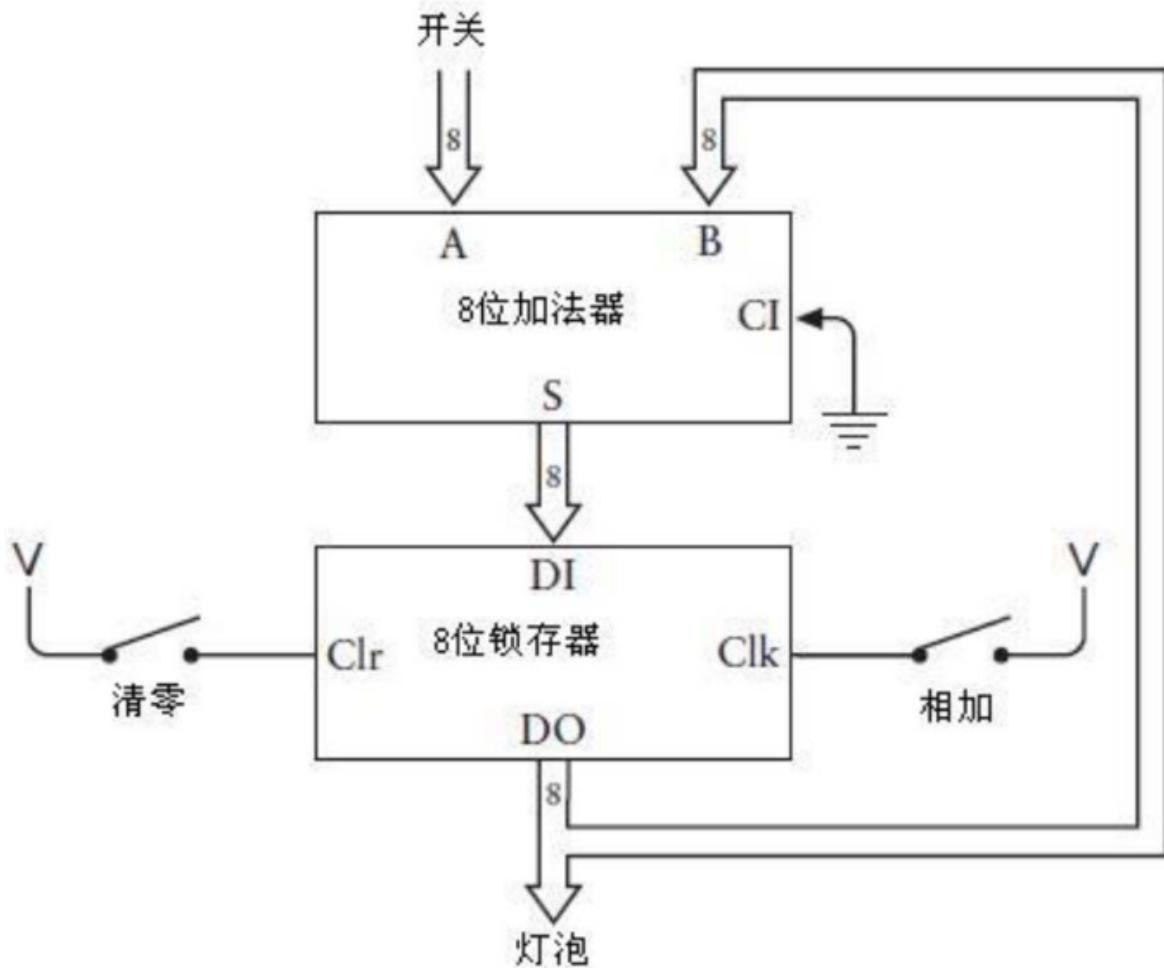
64K×8的RAM阵列和控制面板这一组合的确很实用，它可以帮助我们存储65,536个8位数据并且读取其中的任意一个。与此同时，我们也给其他部件提供了接入系统的机会——需要接入系统的通常是一些电路部件——这些部件可以轻易地读取并利用存储器中存放的数据，还可以把数据写入存储器。

- 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。

## 17 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——以至于愿意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。

- 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。
- 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。

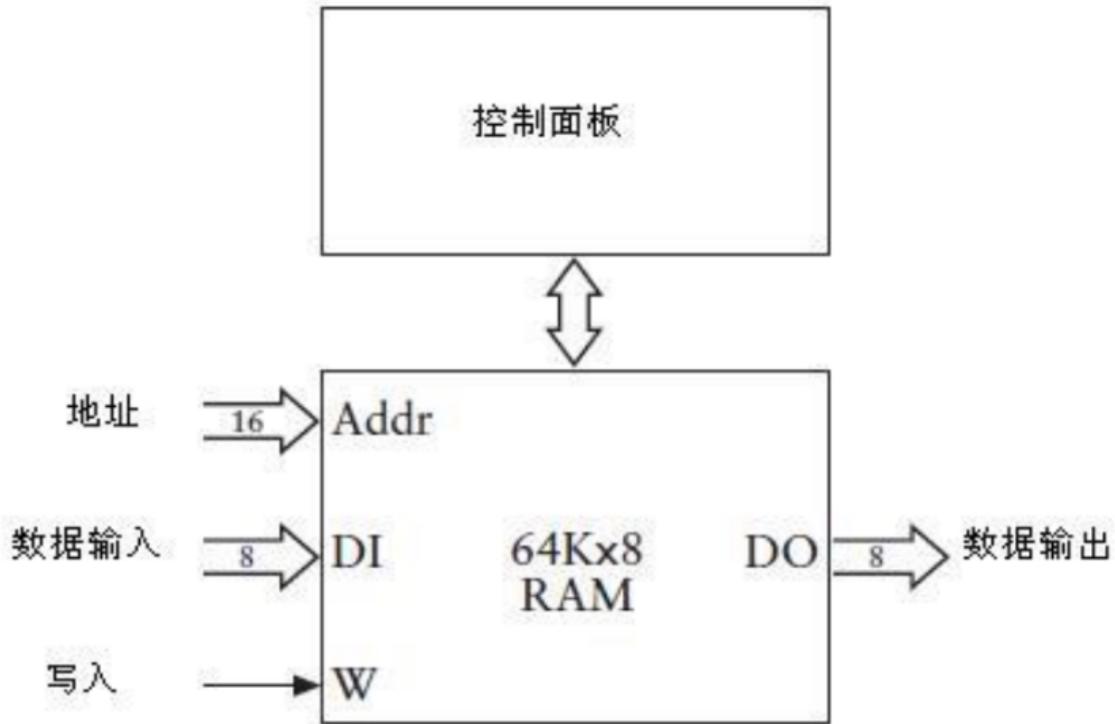


8位锁存器利用触发器来保存8位数据。使用这个设备时，首先需要按下清零开关使锁存器中的内容全部都变为0，然后通过开关输入第一个数。加法器只是简单地将这个数字和锁存器输出的0进行求和，因此相加的结果与原先输入的数字是一样的。按下相加开关可以把这个数保存在锁存器中，最后会点亮某些灯泡以显示它。现在通过开关输入第二个数，加法器把它与已经存放在锁存器中的第一个数相加。再次按下相加开关，就可以把相加的结果存入锁存器中，并通过灯泡显示这个结果。

通过这种方式，可以把一串数相加并显示运行结果。

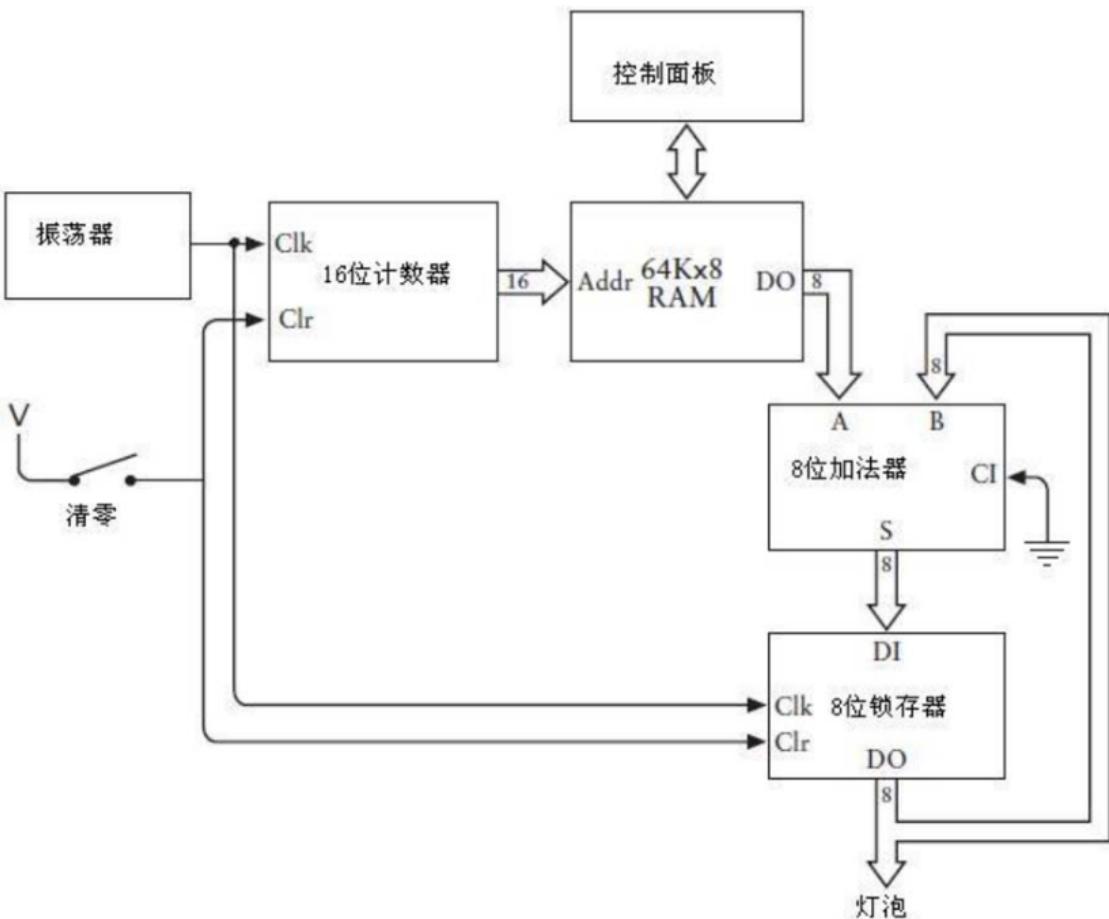
显然，这种设计方案存在一个缺陷：8个灯泡无法显示大于255的数。

- 相比于电平触发 (level triggered)，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。
  - 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。



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。

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。

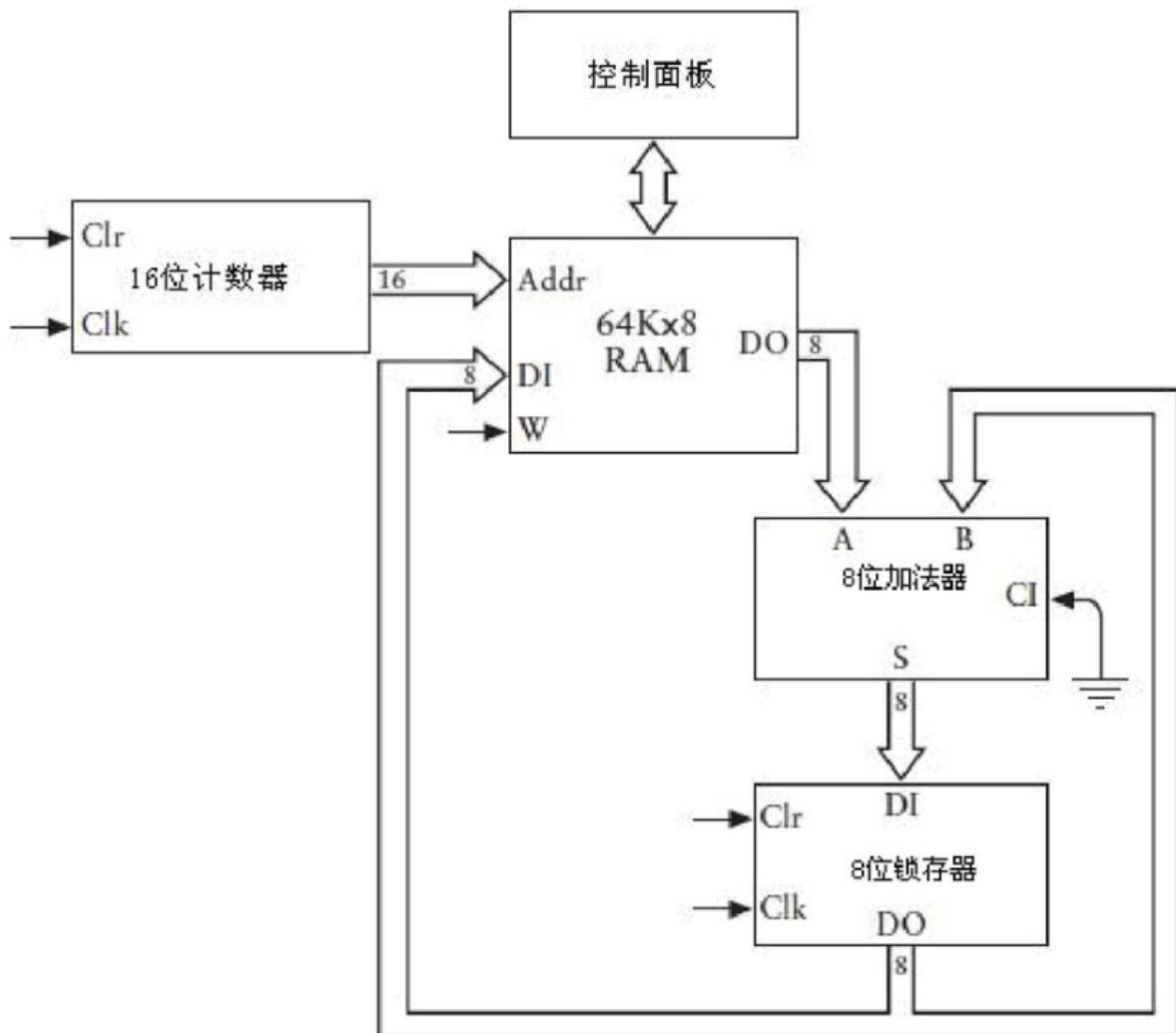
这个电路存在的一个缺陷是：我们没有办法使它停下来。在某一个时刻，所有灯泡会停止变化，因为RAM阵列的剩余部分存放的数都是00h。这时，你可以读取二进制的运算结果。但是当计数器达到FFFFh时，它会重新回滚（roll over）到0000h，这时自动加法器会再一次把所有的数累加到已经计算出来的结果中去。

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。

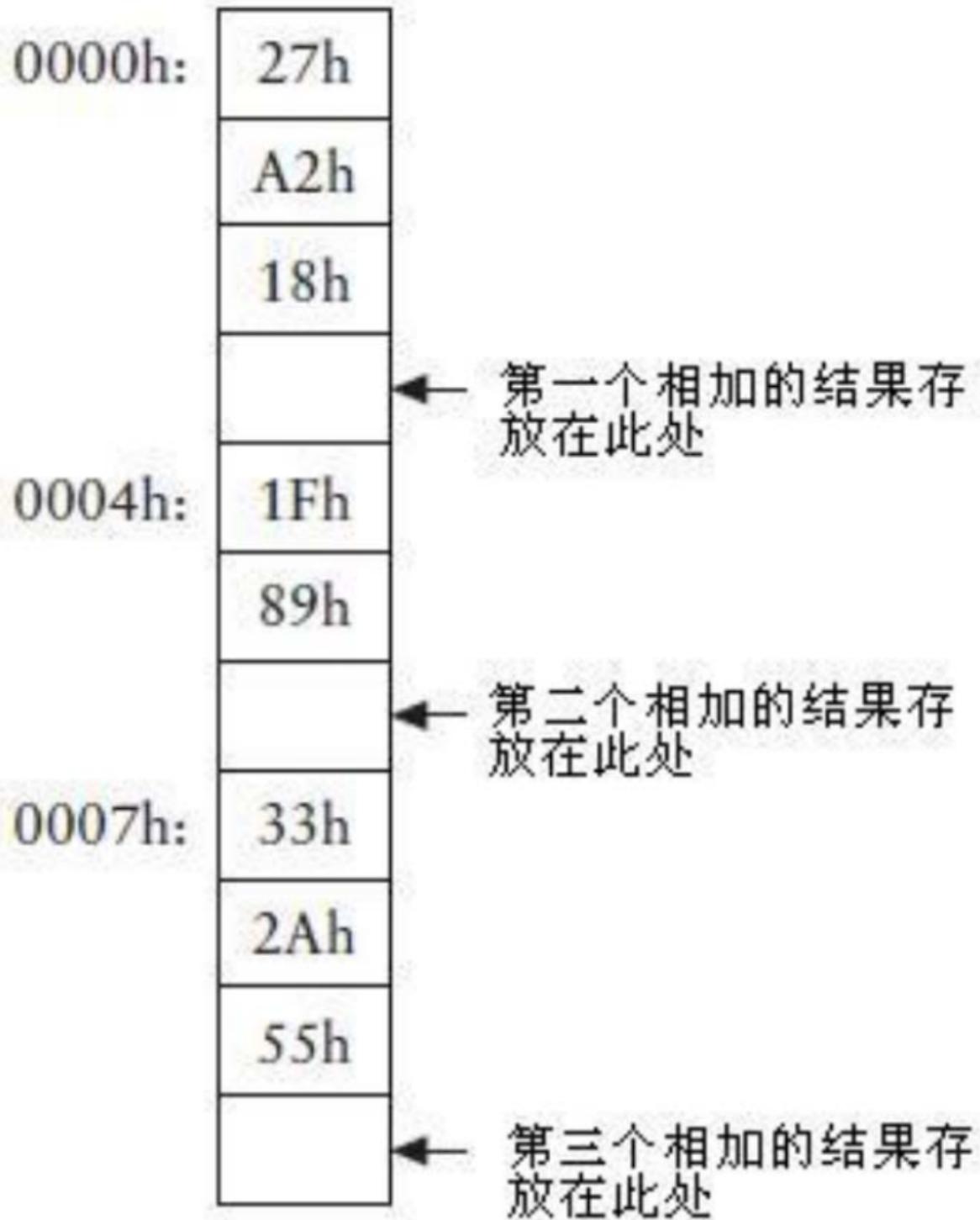
这意味着我们可以去掉与锁存器连接的灯泡，取而代之的是把锁存器的输出端连接到RAM阵列的数据输入端，这样就可以把计算结果写回到RAM阵列中去，如下图所示。



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。

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。

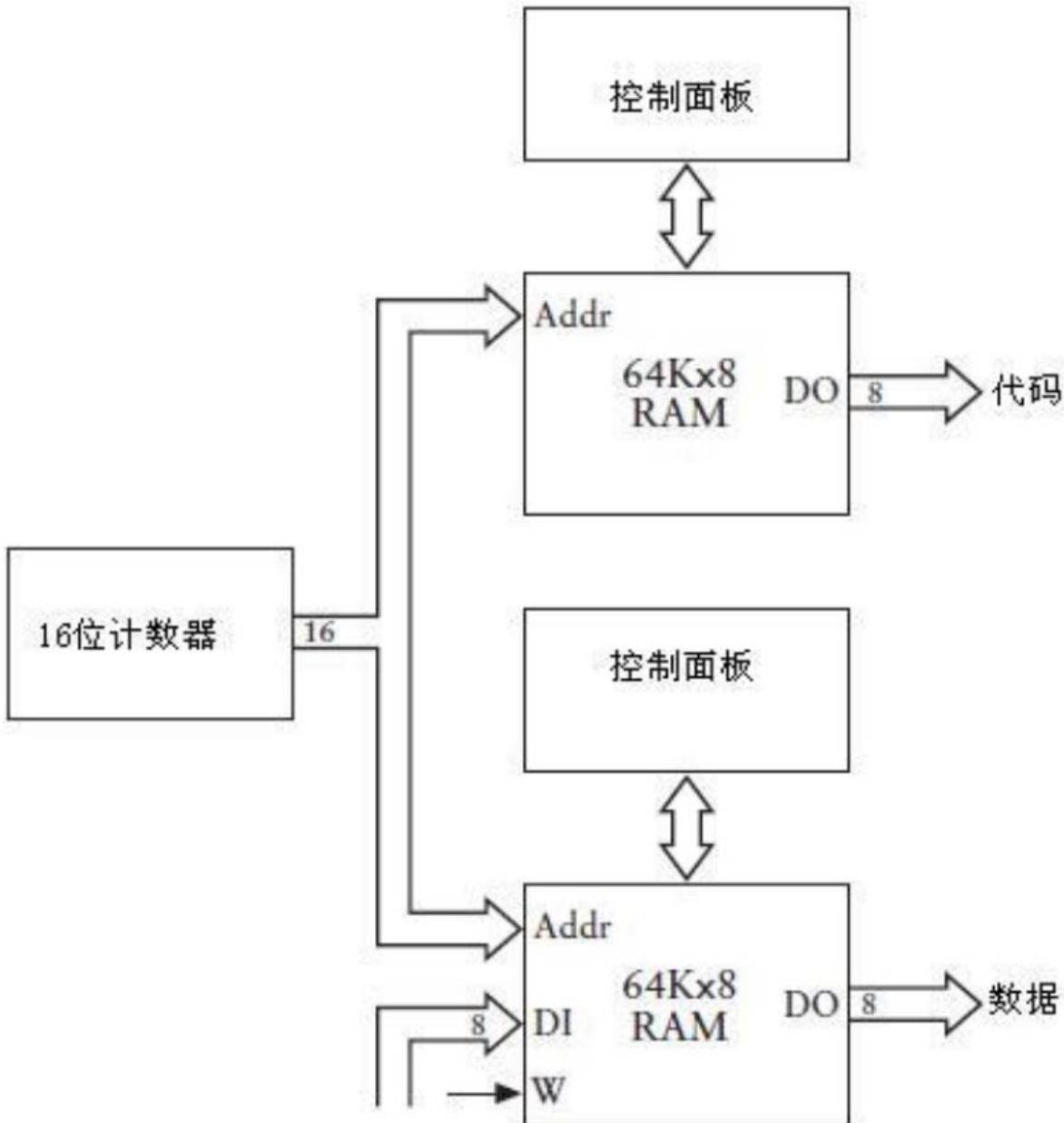
方格表示的是存储器的内容。存储器的每一个字节写在一个方格里。

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）。

- 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。
- 该如何来完成这些工作呢？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“代码”（第二个RAM阵列）。其结构如下图所示。



我们需要四个代码来标记新的自动加法器需要做的四个操作，这些代码可以任意指定。如下所示的是一种方案。

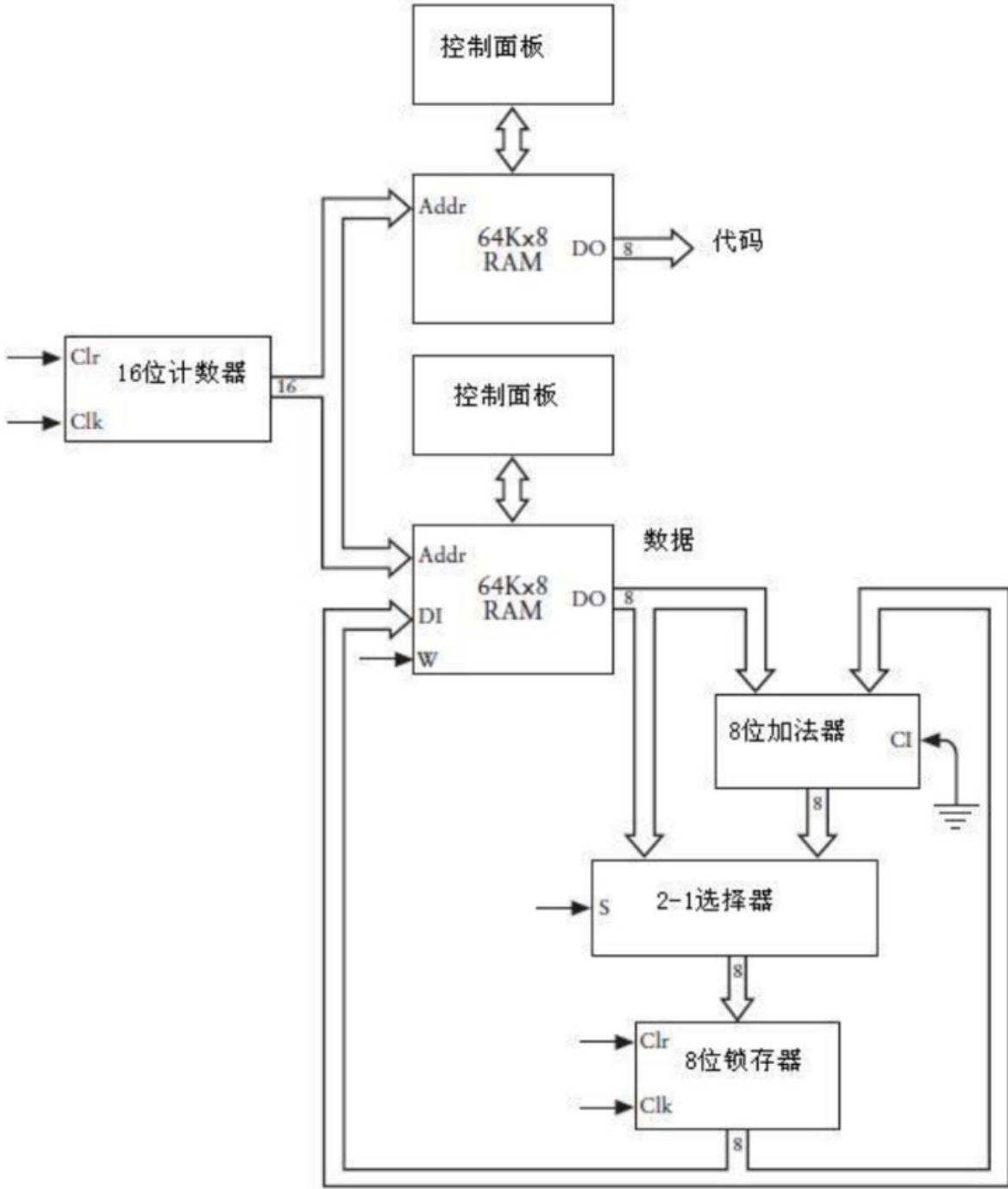
操作码	代码
Load ( 加载 )	10h
Store ( 保存 )	11h
Add ( 加法 )	20h
Halt ( 停止 )	FFh

为了使上面讨论的三组加法得以正常执行，你需要通过控制面板把如下值存入代码RAM阵列。

- 比较一下该RAM阵列与存放累加数据的RAM阵列中的内容，你会发现，代码RAM 阵列中存放的每一个代码都对应着数据RAM中要被加载或者加到累加器中的数，或者对应需要存回到数据RAM中的某个数。以这种方式使用的数字代码常常被称为指令码 (instruction code) 或操作码 (operation code, opcode) 。它们指示电路要执行的某种操作。

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

- 如前所述，最初的自动加法器的8位锁存器的输出要作为数据RAM阵列的输入，这就是Save指令的功能。还需要做另一个改变：以前8位加法器的输出是8位锁存器的输入，但现在为了执行Load指令，数据RAM阵列的输出有时也要作为8位锁存器的输入，这种新的变化需要一个2-1选择器来实现。改进后的自动加法器如下图所示。



图中略去了一些组件，但是仍然清晰地描述了各个组件之间的8位数据通路。16位的计数器为两个RAM阵列提供地址输入。

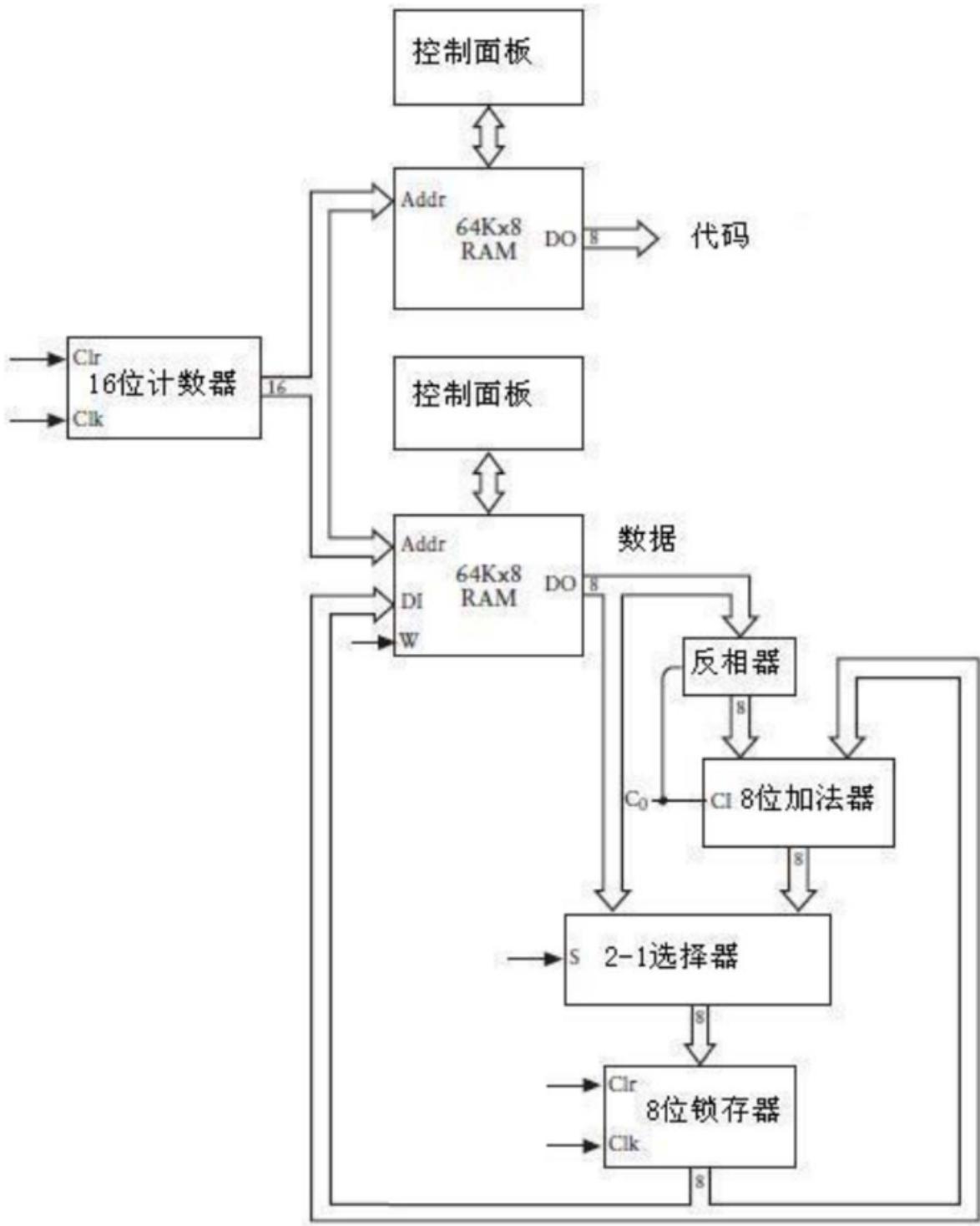
- 通常，数据RAM阵列的输出传入到8位加法器执行加操作。
- 8位锁存器的输入可以是数据RAM阵列的输出（当执行Load指令时），也可以是加法器的输出（当执行Add指令时），这种情况下就需要2-1选择器。
- 通常，锁存器电路的输出又流回到加法器中，但是当执行Save指令时，它就成为了数据RAM阵列的输入数据。

上图缺少的是控制所有这些组件的信号，它们统称为控制信号，包括16位计数器的“时钟”输入和“清零”输入，8位锁存器的“时钟”输入和“清零”输入，数据RAM阵列的“写”(W)输入，2-1选择器的“选择”(S)输入。其中的一些信号很明显是基于代码RAM阵列的输出，例如，如果代码RAM阵列输出是Load指令，那么2-1选择器的“选择”输入必须是0（即选择数据RAM的输出）。只有当操作码是指令Store时，数据RAM阵列的“写”(W)输入必须是1。这些控制信号可以通过逻辑门的各种组合来实现。

- 利用最少的附加硬件和一些新增的操作码，可以让这个电路从累加器中减去一个数。第1步是向操作码表增加一些代码。

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

对于Add和Subtract的代码，其区别仅在于最低有效位，我们称该位为C0。如果操作码为21h，除了数据RAM阵列的数据传入加法器之前要取反，并且加法器进位输入置1之外，电路所做的操作与执行Add指令所做的操作相同。在这个增加了一个反相器的改进电路中，C0信号可以完成这两项任务。改进后的电路结构图如下。



- 还有一个一直没有找到合适的解决办法的问题：加法器及连接到它的所有设备的宽度只有8位。以前提出过的一个解决办法是把两个8位加法器（其他的大部分设备也用两个）连在一起，构成一个16位的设备。
  - 但还有代价更小的解决办法，假如你想把两个16位的数相加，先单独处理最右边的字节（通常称之为低字节），然后再计算最左边的字节，即高字节的和，将低字节运算结果与高字节运算结果分别保存到两个地址。
  - 若对两个数的低字节求和时会产生一个进位，产生的这个进位必须与两个数的高字节的和再相加。

- 为了改进自动加法器的电路，使它可以正确地进行16位数的加法操作。我们需要做的仅仅是在第一步运算时保存低字节数运算的进位输出，并把它作为下一步高字节数运算的进位输入。如何保存1位呢？1位锁存器就是最好的选择了，该锁存器应该被称为**进位锁存器（Carry latch）**。

为了使用进位锁存器，还需要另一个操作码，我们称之为“进位加法”（Add with Carry）。当进行8位数加法时，使用的是常规的Add指令。加法器的进位输入是0，它的进位输出将会保存到进位锁存器（尽管它根本不会被用到）。

如果要对两个16位的数进行加法运算，我们仍然使用常规的Add指令对两个低字节数进行加法运算。加法器的进位输入是0，而其进位输出被锁存到进位锁存器中。当把两个高字节数相加时，要使用新的Add with Carry指令。在这种情况下，两个数相加时要用进位锁存器的输出作为加法器的进位输入。因此，如果第一步低字节数的加法运算有进位，则该进位将用于第二步高字节数的加法运算；如果没有进位，则进位锁存器的输出是0。

- 如果要进行16位数的减法运算，则还需要一个新的指令，称为“借位减法”（Subtract and Borrow）。通常，Subtract指令需要将减数取反并且把加法器的进位输入置1。进位输出通常不是1，因此应该被忽略。但对16位数进行减法运算时，进位输出应该保存在进位锁存器中。在进行第二步的高字节减法运算时，锁存器保存的结果应该作为加法器的进位输入。
- 在加入了Add with Carry和Subtract and Borrow之后，目前我们已经有了7个操作码，如下表所示。

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

需要记住的是，只有当前一次的加法或者进位加法操作使加法器产生进位输出时，Add with Carry指令才会使8位加法器的进位输入置1。因此，只要进行多字节数加法运算，不管实际是否需要，都应该使用Add with Carry指令。为了保证编码的正确，使前面提到16位加法正常进行，可用如下方法。

代码		数据
0000h: 10h	Load	0000h: ABh
20h	Add	2Ch
11h	Store	
10h	Load	76h
22h	Add with Carry	23h
11h	Store	
FFh	Halt	

← 低字节运算结果保存在此处  
← 高字节运算结果保存在此处

- 当然，把这些数依次输入存储器并不是最好的做法。因为你不但要使用开关来输入这些数，而且保存这些数的存储单元的地址也不是连续的。例如，AB76h从最低字节开始，每个字节依次保存在0000h, 0003h中。而为了得到最后的结果，还需要检查0002h, 0005h中的数。
- 除此之外，当前设计的自动加法器不允许在随后的计算中重复使用前面的计算结果。
  - 假设我们要对三个8位数求和，然后再从中减去一个8位数并保存结果。这可能需要一条Load指令，两条Add指令，一条Subtract指令以及一条Store指令。但如果想从原来的求和结果（3个8位数的和）中减去另一个数该怎么做呢？这个求和结果已经不能被访问了，每次我们使用它的时候都必须重新计算。
- 产生上述情况的原因就在于我们构造的自动加法器具有如下的特性：它的代码存储器和数据存储器是同步的、顺序的，并且都从0000h开始寻址。代码存储器中的每一条指令对应数据存储器中相同地址的存储单元。
  - 一旦执行了一条Store指令，相应的，就会有一个数被保存到数据存储器中，而这个数将不能重新加载到累加器。
- 要解决这个难题，需要对自动加法器的设计做一个根本性的且程度极大的修改。这个想法实现起来似乎非常困难，但是很快你就会发现改进后的加法器具有更高的灵活性。

目前已经有了7个操作码。每一个操作码在存储器中占1个字节。现在除了Halt操作码外，我希望每一个指令在存储器中仅占据3个字节的空间，其中第一个字节为代码本身，另外的两个字节用来存放1个16位存储器单元地址。

对于Load指令来说，后两个字节保存的地址用来指明数据RAM阵列的一个存储单元，该单元存放的是需要被加载到累加器中的字节。

对于Add, Subtract, Add with Carry, Subtract with Borrow指令来说，该地址指明的存储单元所保存的是要从累加器中加上或减去的字节。

对于Store指令来说，该地址指明的是累加器中的内容将要保存到的存储单元地址。

- 例如，当前加法器所能进行的最简单的运算就是对两个数求和。为了执行这个操作，需要按下面的方式设置代码RAM阵列和数据RAM阵列。

代码		数据
0000h:	10h	Load
	20h	Add
	11h	Store
	FFh	Halt
0000h:	4Ah	
	B5h	

← 结果保存在此处

在改进的自动加法器中，每条指令（除了Halt指令）需要3个字节。

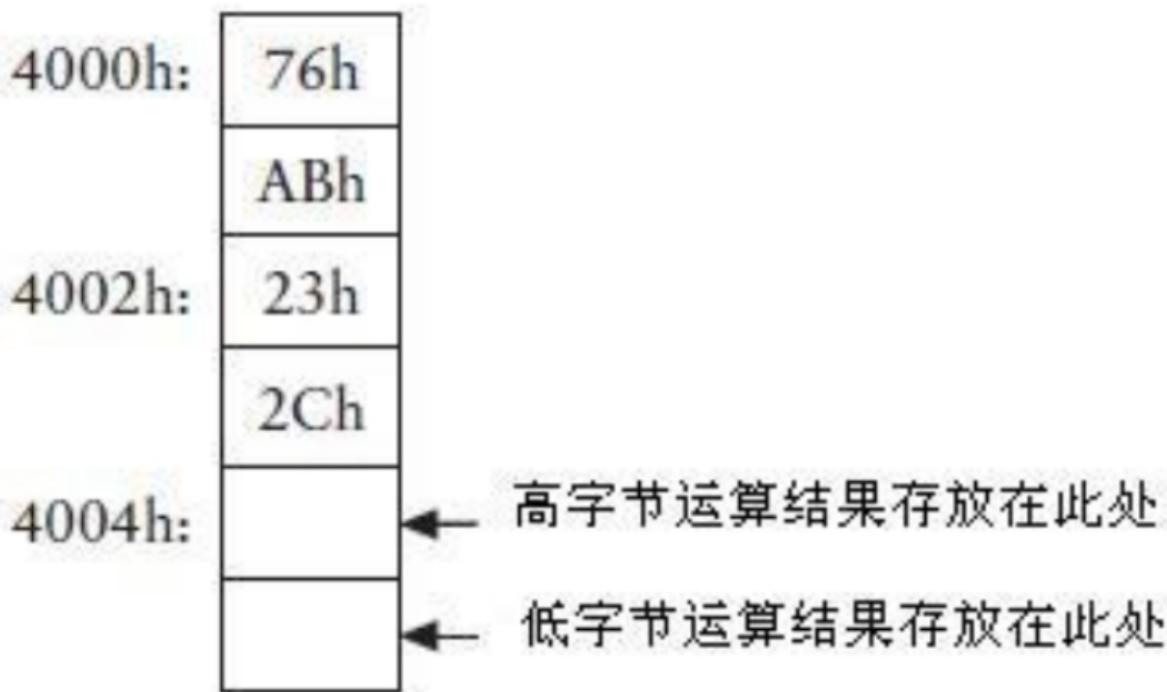
代码		
0000h:	10h	把0000h地址处的字节 装入累加器
	00h	
	00h	
0003h:	20h	把00001h地址处的字 节加到累加器
	00h	
	01h	
0006h:	11h	把累加器的内容保存到 0002地址处
	00h	
	02h	
0009h:	FFh	Halt

每一条指令的代码（除了Halt指令）后跟两个字节，用来指明数据RAM阵列中16位的存储地址。这三个地址恰巧是0000h, 0001h和0002h, 但它们可以是任何其他可用的地址。

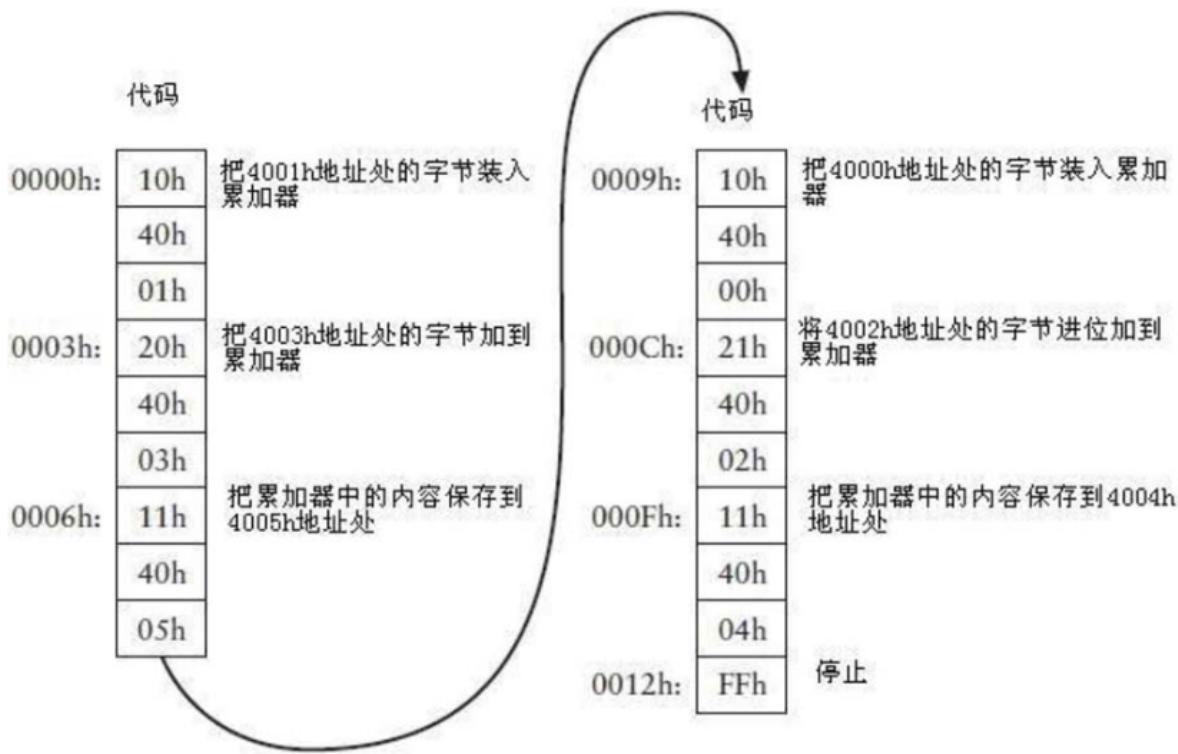
- 前面讲到了如何用Add或Add with Carry指令来对两个16位数求和。必须把两个数的低字节保存到存储器的0000h和0001h地址，把其高字节保存到0003h和0004h地址，运算的结果分别保存在0002h和0005h。

通过这种变化，我们可以用一种更合理的方式来保存这两个操作数及其运算结果，可能会把它们保存到我们从未用到过的存储区域。

## 数据



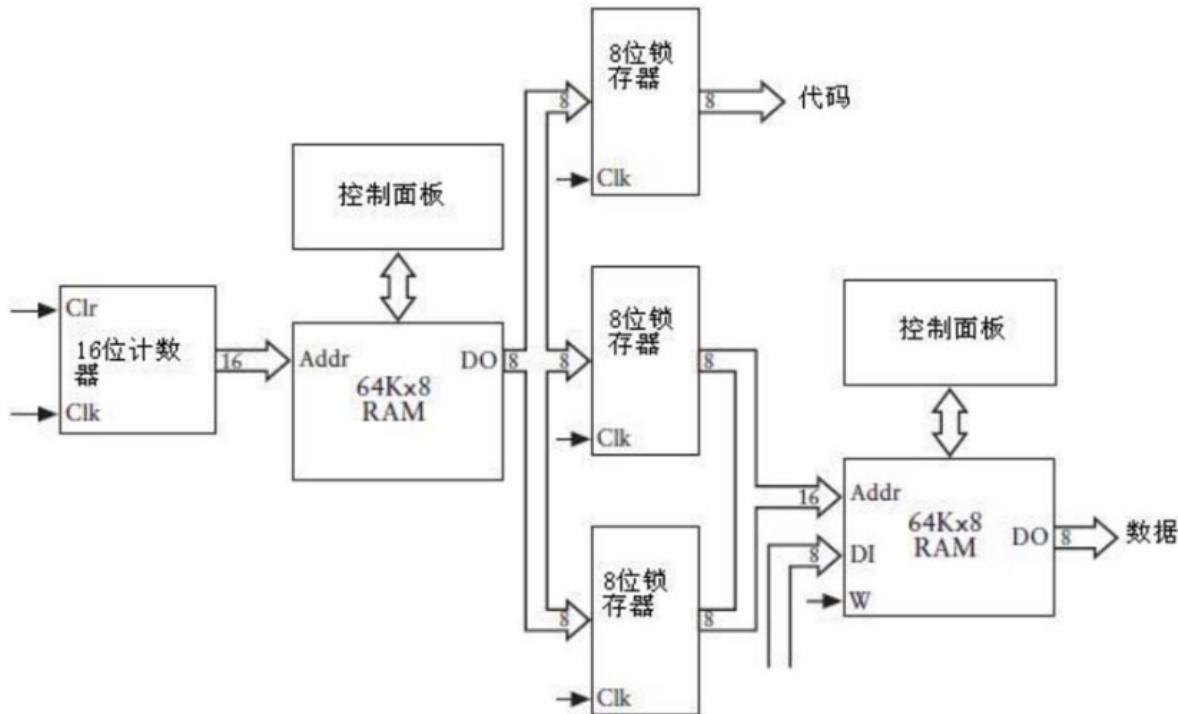
这6个存储单元不必像上图中这样全都连在一起，它们可以分散在整个64 KB数据RAM阵列的任意位置。为了把这些地址中的数相加，代码RAM阵列中的指令必须用以下方式设置。



可以看到，保存在地址4001h和4003h处的两个低字节数先执行加法，其结果保存在4005h地址处。两个高字节数（分别保存在4000h和4002h处）通过Add with Carry指令相加，其结果保存在地址4004h处。

如果去掉Halt指令并向代码RAM中加入更多指令，随后的计算可以通过引用地址很方便地使用原来的那些操作数及其结果。

- 实现该设计的关键是把代码RAM阵列的数据输出到3个8位锁存器中。每个锁存器保存该3字节指令的一个字节。第一个锁存器保存指令代码本身，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。第二个和第三个锁存器的输出构成了数据RAM阵列的16位地址。



从存储器中取出指令的过程称为**取指令 (instruction fetch)**。在我们设计的加法器中，每一条指令的长度是3个字节。因为每次从存储器取回一个字节，所以取每条指令需要的时间为3个时钟周期。此外，一个完整的指令周期需要4个时钟周期。这些变化必然使得控制信号更加复杂。

机器响应指令码做一系列操作的过程称为**执行 (execute) 指令**，但这并不能表明机器是一种有生命的东西，因为它不能自行分析机器代码并决定该做什么。每一种机器码用其唯一的方式触发多种控制信号，从而引发机器执行各种操作。

- 注意，为了让上面的加法器更加有用，我们牺牲了运算速度。使用同样的振荡器，它的运算速度只有本章提到的第一个加法器的1/4。这验证了一个称为TANSTAAFL (There Ain't No Such Thing As A Free Lunch) 的工程准则，它的意思是“天下没有免费的午餐”。通常上帝总是很公平的，你改进了机器的某个方面，则其他方面就会受到损失，有得就有失。

前面介绍了两种RAM阵列，一个用来存放指令码，另一个用来存放操作数据——这种设计使得自动加法器的结构非常清晰和易于使用。但现在我们使用3字节长的指令格式，第二个和第三个字节用来指明操作数的存储地址，因此就没有必要再使用两个独立的RAM阵列。操作码和操作数可以存放在同一个RAM阵列。

- 为了实现这个设计，我们需要一个2-1选择器来确定如何对RAM阵列寻址。通常，和前面的方式相同，我们用一个16位的计数器来计算地址。数据RAM阵列的输出仍然连接到3个锁存器，分别用来保存指令代码及其对应操作数的16位地址，其16位的地址输出是2-1选择器的第二种输入。地址被锁存后，可以通过选择器将其作为RAM阵列的地址输入。

现在可以把操作指令和操作数据保存在同一个RAM阵列中。

例如，下图演示了如何把两个8位数相加，然后从结果中再减去一个8位数。

0000h:	10h 00h 10h 20h 00h 11h 21h 00h 12h 11h 00h 13h FFh : 0010h:	把0010h地址处的字节装入累加器  把0011h地址处的字节加到累加器  从累加器中减去0012h地址处的字节  把累加器中的内容保存到0013h地址处  停止  最终的结果保存在此处
--------	--	---

通常，指令从0000h开始存放，这是因为当计数器复位后从该位置访问RAM阵列。最后的Halt指令存放 在000Ch地址。我们可以把这3个操作数及它们的运算结果保存在RAM阵列的任何地址（当然这不包括最开始的13个字节，因为它们已经用来存放操作指令），所以我们选择在从0010h地址开始保存操作数。

假设现在你发现需要在原来的结果中再加两个数，你可以向存储器中输入一些新的指令以替换原来所有的指令，但是你可能不愿意这么做。

或许你更倾向于在原指令的地址后增加一些新的指令。第一步要做的就是把000Ch地址处的Halt指令替换为一个Load指令。但你仍然需要增加两条Add指令，一条Store指令，以及一条新的Halt指令。

唯一的问题是，现在0010h地址已经保存了一些数据，因此需要把这些数据转移到较高的地址空间中，然后还需要修改那些指向这些地址空间的指令。

试想一下，把操作码和操作数存放在同一个RAM阵列并不是一个急于解决的问题。但可以肯定的是，这是一个迟早要解决的问题，不如现在就找一个解决办法吧。

在当前的例子中，也许你更愿意从0020h地址开始存放新的指令，并从0030h处开始存放新的操作数。

0020h:	10h	把0013h地址处的字节装入累加器
	00h	
	13h	
	20h	把0030h地址处的字节加到累加器
	00h	
	30h	
	20h	把0031h地址处的字节加到累加器
	00h	
	31h	
	11h	把累加器中的内容保存到0032h地址处
	00h	
	32h	
	FFh	停止
	:	
0030h:	43h	
	2Fh	
		← 最终结果保存在此处

现在，两部分指令的位置分别起始于地址0000h和0020h，而两部分操作数据的地址分别起始于0010h和0030h。我们希望自动加法器从0000h开始执行所有指令完成计算任务。

我们必须移除000Ch处的Halt指令，这里的移除是指用其他代码替换它。

但仅仅如此是不够的，问题在于不论我们用什么来替换Halt指令，保存在地址000Ch的字节都会被当做指令代码。更糟糕的是，从这个位置开始，存储器中每隔3个字节的地址：000Fh, 0012h, 0015h, 0018h, 001Bh以及001Eh，这些地址保存的字节也会被当做指令代码来处理。

如果其中的一个字节恰好为11h（这是Store指令的代码）将会发生什么？如果Store指令后面的2个字节的地址所指向的位置刚好为0023h，那又会发生什么呢？它导致的结果是，加法器将累加器中的内容写入这个地址，而这个地址中已经保存了重要的数据。

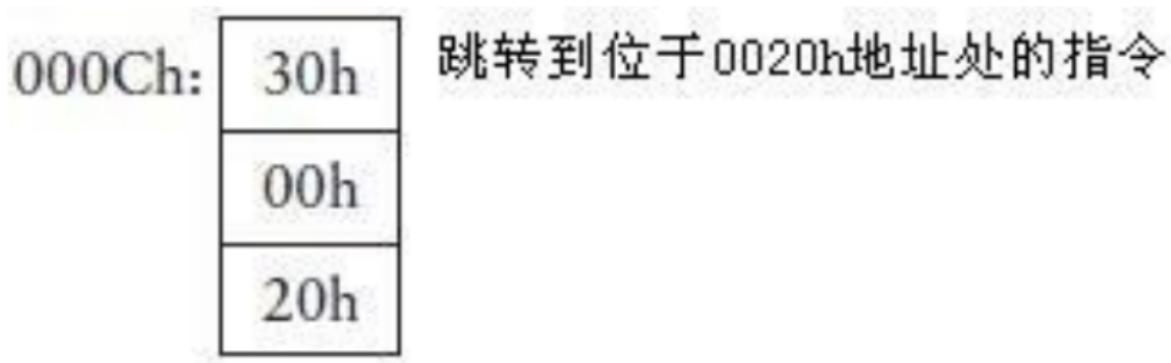
即使这些情况都没有发生，加法器从存储器的001Eh地址之后取到的下一条指令的位置将是0021h，不是0020h，而事实上0020h才是下一条指令的存储地址。

- 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”。

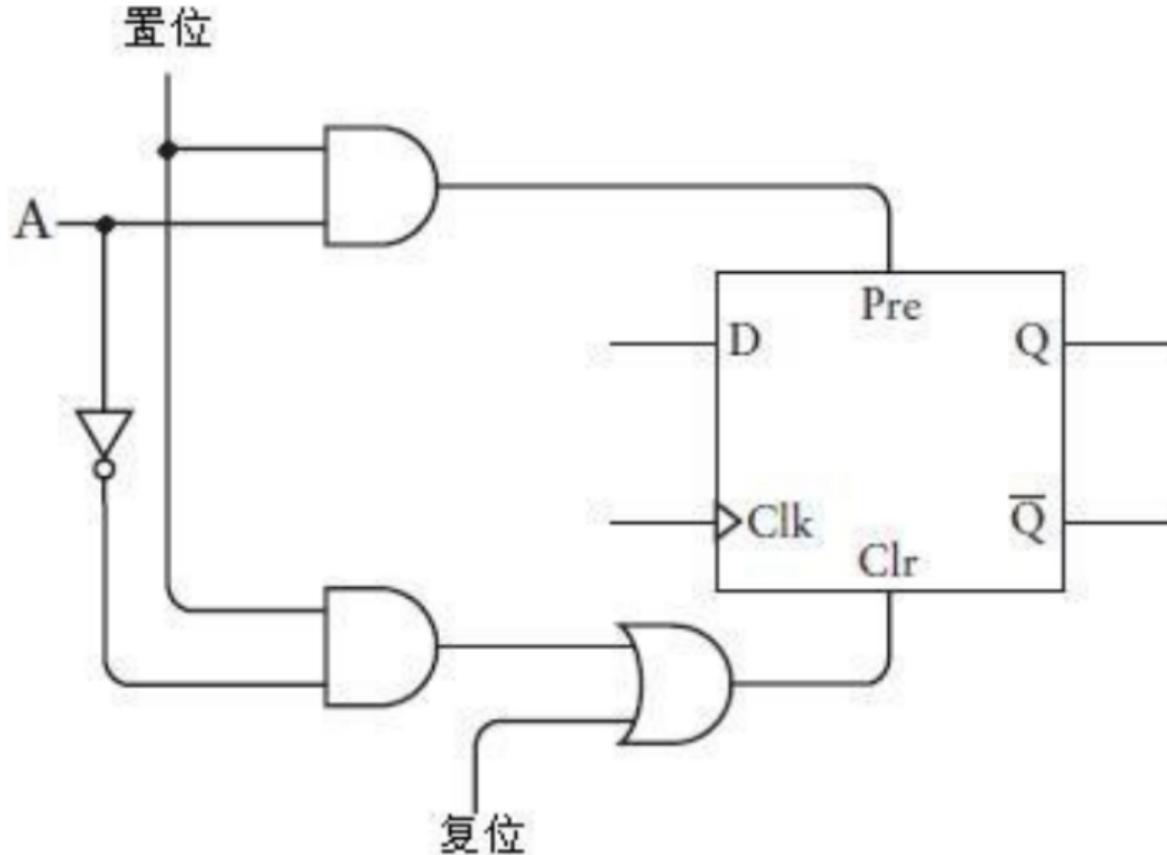
在上面的例子中，我们可以用一个Jump指令来替换000Ch地址处的Halt指令。



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：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。



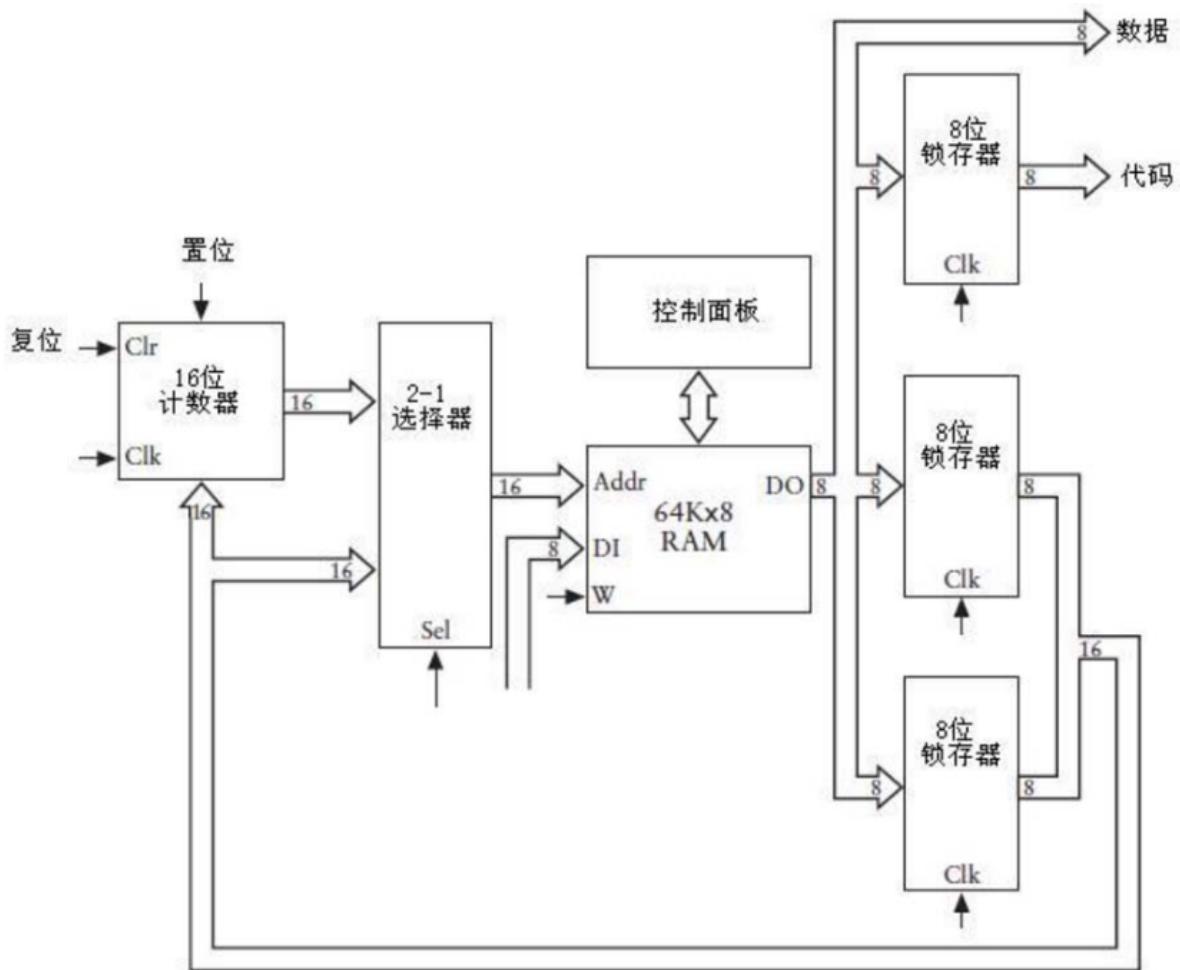
通常，置位信号为0。此时，触发器的预置端输入为0，在复位信号不为1的情况下，清零信号也为0。在这种情况下，触发器就可以独立清零，而不受置位信号的影响。

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。



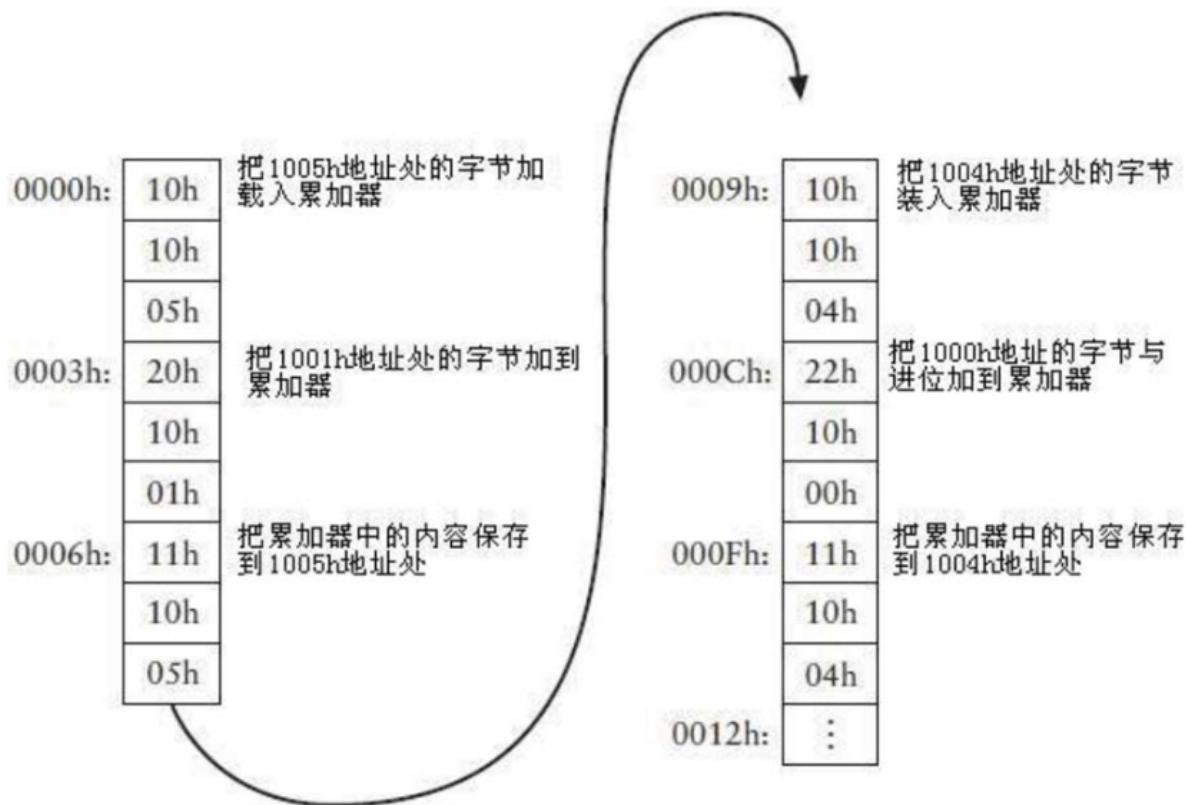
显然，只有当指令代码为30h并且其后的16位地址被锁存时，我们才必须确保置位信号为1。

- 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（ConditionalJump）。

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？

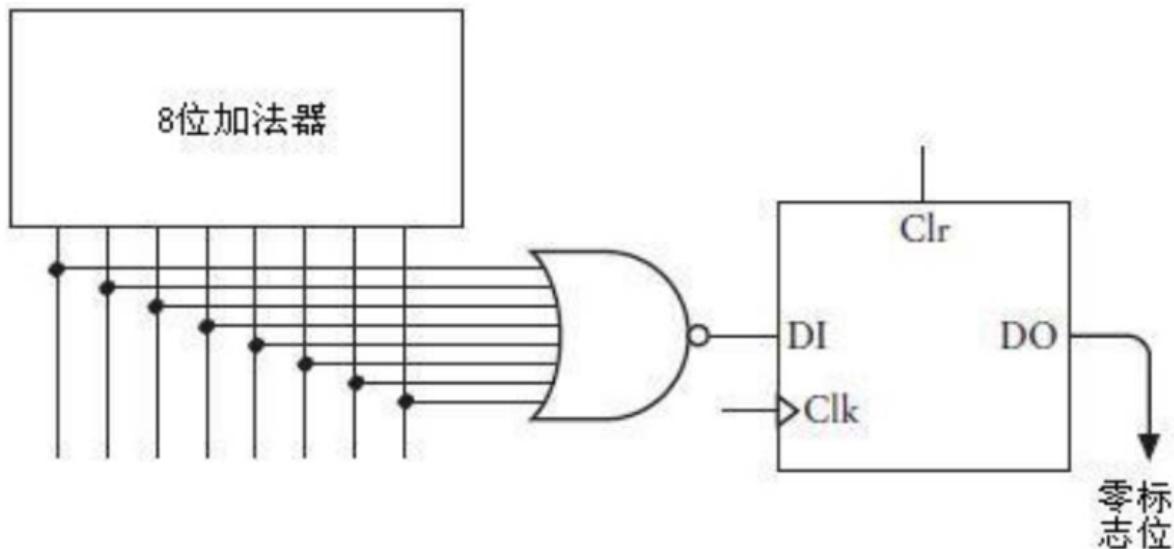


A7h和1Ch相乘的结果（即十进制的28）和把28个A7h累加的结果相同。



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。

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指令。

0012h:	10h 10h 03h	把1003h地址处的字节装入累加器
0015h:	20h 00h 1Eh	把001Eh地址处的字节加到累加器
0018h:	11h 10h 03h	把累加器的内容保存到1003h地址处
001Bh:	33h 00h 00h	如果零标志位不为0，则转移到0000h地址处
001Eh:	FFh	停止

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，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。
- 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。

“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]      ; 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

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。

- 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，这是因为他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，你需要手工对其进行汇编：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。
- 在18世纪（直到20世纪40年代），计算机就好比一个以计算维持生计的人，而计算能力就好比计算机的生命线。在这个用天上星辰进行航海导航的时期，经常要用到对数表，而三角函数表对航海导航也非常重要。此外，如果你想要发表新数学表，就需要使用许多台计算机，让它们一起工作，最后将所有的结果汇总成一张表。当然，从初始计算到设置打印最终结果，每一个阶段都可能会出现意想不到的错误。为了消除数学表中错误，英国数学家和经济学家查尔斯·巴贝奇（Charles Babbage，1791-1871）勤奋工作，他和塞缪尔·莫尔斯（Samuel Morse）差不多是同时代的人。在那个时期，数学表（例如对数表）并没有计算表中每一项的确切对数值，因为那样做将耗费太多时间。取而代之的方法是选择性的对数计算，即选取一些数字进行对数计算，而对于介于这些数字之间的数的对数则采用插补法进行填充，即差分法（differences），通过相对简单的计算求得结果。

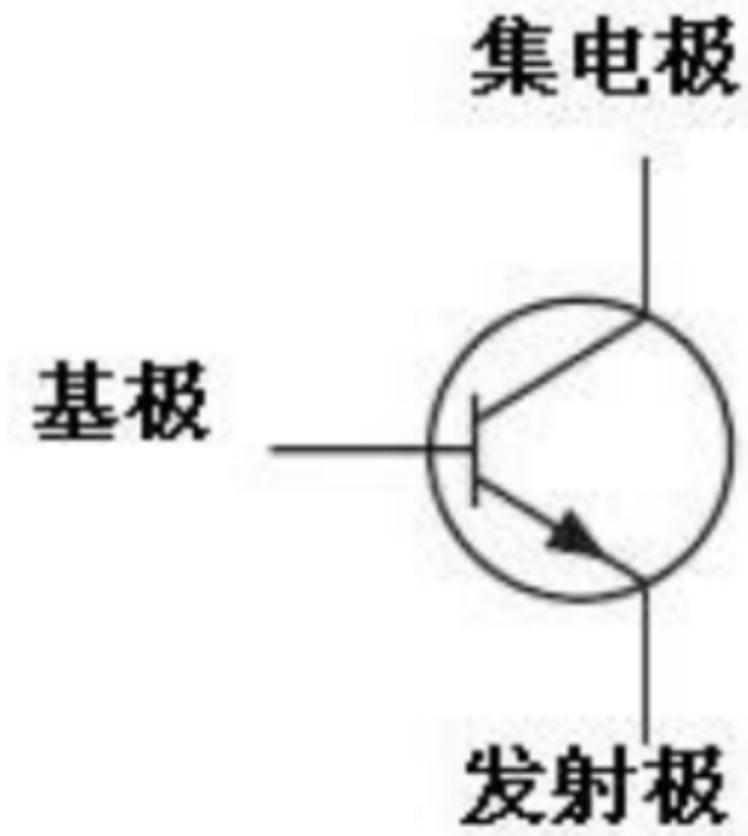
- 大约在1820年，巴贝芝认为他可以设计制造一台可以自动建表，甚至可以自动设置打印类型的机器，这种机器可以完全消除上述错误。因此他设想出了差分机（Difference Engine），从本质上讲差分机是一个大型机械加法器。在差分机中，多位的十进制数通过可以啮合在10个不同位置的轮子表示，而负数用10的补数来表示。尽管早期一些模型表明巴贝芝的设计是完全可行的，而且他也获得了英国政府的一些资金支持（当然不是很多），但差分机却从来没有完成过，在1833年，巴贝芝放弃了这项工作。
- 可正在那个时候，巴贝芝有了一个更好的想法，那就是解析机（Analytical Engine），他的后半生一直在不断重复地设计与修改（其间还制作过几个小模型以及部分构件）这个机器，直到其生命的尽头。
  - 解析机是19世纪最接近于计算机的器件，在巴贝芝的设计中，解析机包含一个存储部件（类似于现在存储器的概念）和一个运算部件（类似于算术逻辑单元）。乘法可以通过重复的加法运算求解，同样的，除法可以通过重复的减法求解。
- 巴贝芝大概是第一个意识到条件跳转在计算机中重要性的人，操作循环（cycle），应该这样去理解：它意味着某个操作集（set of operations）重复执行的次数不止一次。它的次数可以是仅仅两次或者是无限次，但它们实际上都是组成操作集的操作被重复执行了。在很多实例的分析中，我们经常会看到由一个或多个循环构成的重复组群（recurring group），也就是循环中包含的一个循环或者多个循环。
- 尽管在1853年差分机最终由一对父子——乔治（George）和爱德华·舒尔茨（Edvard Scheutz）制造出来，但已经被人们遗忘了好多年的巴贝芝设计的差分机，直到20世纪30年代才因为人们开始探索20世纪计算机的起源时而重新被提起。那时，巴贝芝所做的一些工作已经都被后来的技术超越，除了超前的自动化观念，他所做的工作对20世纪计算机工程来说几乎没有可以利用的。
- 1896年，赫尔曼·霍尔瑞斯创办制表机公司（Tabulating Machine Company），租借并出售其穿孔卡片设备。到1911年，由于公司的合并，制表机公司更名为计算制表记录公司（Computing-Tabulating-Recording Company），或者叫做C-T-R公司。再到1915年，托马斯J·华盛顿（Thomas J. Watson，1874-1956）成为C-T-R公司的总裁，他在1924年将公司的名字更改为国际商业机器公司（International Business Machines Corporation），即IBM。到1928年，在1890年人口普查中最初使用的卡片逐渐演变为著名的IBM卡片，“do not spindle, fold, or mutilate”，这种卡片有80列12行，使用了将近50年，甚至在其后期，还有人把它们叫做霍尔瑞斯卡片。关于这些卡片的遗留问题将在第20、21和24章中进一步的讲述。
- 进入到20世纪之前，让我们重新审视一下19世纪这一百年。因为主题所限，本书更多的关注的是数字性质的发明，其中包括电报、盲人用点字法、巴贝芝机器，以及霍尔瑞斯卡片。而在与数字概念以及相关设备打交道时，你会发现整个世界皆为数字。但是，19世纪的发现和发明确切的来讲不是数字的。实际上，通过感官所认识的大自然中只有很少一部分是数字的，更多的时候表现为不可分割的整体。
- 尽管霍尔瑞斯在他的制表机以及分类机中使用了继电器（relays），但是人们直到20世纪30年代中期才开始用继电器来构建计算机——它们最终被叫做机电化（electromechanical）计算机。在这些机器中使用的继电器不同于一般的电报继电器，后者的主要作用是为了完善电话系统的路由控制。
  - 早期的继电式计算机与上一章中的继电式计算机不是同一个概念（我们随后会学到，从20世纪70年代开始，这种继电式计算机依靠微处理器进行计算）。需要特别说明的一点，尽管现代计算机内部使用二进制数，但早期的继电式计算机并非如此。
  - 我们的继电式计算机与早期的继电式计算机存在另外一个不同点，那就是在20世纪30年代，没有人能够疯狂到用继电器制造出524,288位的存储器！资金的花费、空间的占用和能源的耗费使得制造如此大的存储器变得不大可能。可得到的极少存储器也只用来存储中间结果，而程序本身则存储在一些物理媒介上面，例如带穿孔的纸带。实际上，将代码和程序放入到存储器进行处理是后来发明的做法。
- 下面按时间顺序进行介绍

- 第一台继电式计算机由康拉德·楚泽 (Conrad Zuse, 1910-1995) 制造，1935年还是工科学生的他在其父母位于柏林的家中制造了这台机器。这台机器中使用了二进制数，但其早期的版本中使用的是机械存储器而非继电器。楚泽使用老式35毫米电影胶片进行穿孔，然后在上面编制程序。
- 1937年，贝尔电话实验室 (Bell Telephone Laboratories) 的乔治·史提必兹 (George Stibitz, 1904-1995) 将一对电话继电器带回了家中，并在他厨房的桌子上连接了一个1位加法器，后来他妻子将其称之为K机器 (K是厨房“kitchen”的头一个字母)，这个实验促使1939年贝尔实验室中复数计算机的诞生。
- 同一时期，哈佛大学研究生霍华德·艾肯 (Howard Aiken, 1900-1973) 要寻找做大量的重复计算的方法，而正是他的这一需求促使哈佛大学与IBM合作，并最终在1943年创造出一台自动连续可控计算机 (Automated Sequence Controlled Calculator, ASCC)，也就是闻名于世的Harvard Mark I。这是第一台可以打印表格的数字计算机，它最终将查尔斯·巴贝芝的梦想付诸于现实。Mark II是最大的继电式计算机，使用了13,000个继电器。
- 对于构造计算机来说，继电器不是最完美的设备，因为它们是机械性的，利用金属片的弯曲和伸直状态进行工作，而频繁的工作可能导致其断裂，另外如果接触点之间有污垢或者卡住纸屑，也会导致继电器失效。1947年发生了一件著名的事故，人们从Mark II计算机的一个继电器中发现了一只飞蛾。格蕾丝·莫瑞·赫柏 (Grace Murray Hopper, 1906-1992) 于1944加入了艾肯的团队，日后成为了计算机编程语言领域非常著名的人物。他将上面提到的那只飞蛾用带子绑在计算机日志 (logbook) 上，并在其边上注明“第一个被发现的有生命的bug”。
- 真空管 (vacuum tube) 是一种可以替代继电器的元件，它是由约翰·安布罗斯·弗莱明 (John Ambrose Fleming, 1849-1945) 和李·德·福雷斯特 (Lee de Forest, 1873-1961) 在进行无线电通信连接研究时开发出来的。到20世纪40年代，真空管已经被广泛应用于放大电话信号，实际上，那时几乎每一个家庭都拥有一台带有发光二极管可控收音机，它们能放大无线信号，并且把它们变成还原为人们能听见的声音。真空管同样可以通过连接成与门、或门、与非门，以及与或门——这一点很像继电器。
  - 究竟是由继电器还是由真空管组成这些逻辑门并不重要，重要的是这些逻辑门可以被装配组合成加法器、选择器、解码器、触发器，以及计数器。不论真空管何时取代继电器，前面章节中讲述的关于基于继电器部件的一切同样是有有效的。
  - 真空管同样存在自身的问题，比如，价格昂贵、耗电量大，以及产生的热量太多。可是，其最大的问题是真空管最终会被烧坏，如同人活一世一样，是无法改变的事实。那时，拥有真空管收音机的人们习惯于定期更换真空管，而电话系统设计时有很多冗余的真空管，所以一个真空管的报废有时并不是什么大事 (不管怎样，人们不会期待电话系统是完美无瑕的)。可是在计算机中，当一个真空管烧坏时并不可能立刻被检测到，此外，一台计算机拥有数量巨大的真空管，按统计学来分析，每隔几分钟就会烧坏一个。
  - 用真空管取代继电器的最大好处在于真空管的状态可以在百万分之一秒 ( $\mu\text{s}$ ) 内发生转变。真空管状态转变 (开关的打开与关闭) 的速度比继电器要快1000倍，继电器在其最好状态下状态的转变也需要1ms，即千分之一秒。十分有趣的是在计算机的早期发展中，计算速度并不是主要考虑的问题，而这个时期的计算速度与从纸张或者电影胶片中读取程序的速度有关。由于当时的计算机都按照这种方式构建，因此采用真空管比继电器到底计算速度提升了多少，并不重要。
  - 但是在20世纪40年代初期，新设计的计算机中真空管开始取代继电器。到1945年，真空管已经完全取代了继电器。虽然继电器计算机被称为电动机械计算机，但真空管是第一台电子计算机的基础。

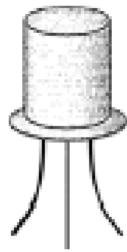
- 在英国，巨像（Colossus）计算机（1943年首次投入使用）用来破译德国名为“Enigma”代码生成器产生的代码，艾伦·M·图灵（Alan M.Turing, 1912-1954）为这个项目（以及英国后来几个的计算机项目）做出了巨大贡献，图灵撰写了两篇非常有影响的论文，这使他如今成为计算机领域的鼎鼎大名的人物。
  - 第一篇论文发表于1937年，首次提出了“可计算性”（computability）这个概念，用来分析哪些事情计算机可以做到，哪些做不到。他为计算机构想了一个抽象模型，这就是现在为人所熟知的图灵机（Turing Machine）。
  - 图灵第二篇非常有名的论文是关于人工智能的，在这篇论文中他介绍了一种测试机器智能的方法，即现在为人熟知的图灵测试法（Turing Test）。
- 在摩尔电子工程学院（宾夕法尼亚大学），J·普利斯普·埃克特（J.Presper Eckert, 1919-1995）和约翰·莫克利（John Mauchly, 1907-1980）设计了ENIAC（Electronic Numerical Integrator and Computer，电子数字积分计算机），使用了18,000个真空管并最终在1945年底完成。按全部吨位算（大约30吨），ENIAC是曾经（或许以后也是）制造出来的最大的计算机。到1977年，人们可以在Radio Shack买到速度更快的计算机。埃克特和莫克利想为计算机申请专利，可是却被竞争者约翰·V·安塔纳索夫（John V.Atanasoff, 1903-1995）阻扰了，他更早一步设计了一台电子计算机，但它运行得并不顺畅。
- ENIAC吸引了数学家约翰·冯·诺依曼（John von Neumann, 1903-1957）的眼球。从1930年开始，出生在匈牙利的冯·诺依曼就定居美国。作为一名令公众瞩目的人物，因其仅凭自己的大脑就能进行复杂的数学计算而闻名，冯·诺依曼当时是普林斯顿高级研究院的一名数学教授，研究范围很广，从量子力学到游戏应用，甚至到经济理论。
- 约翰·冯·诺依曼协助设计的ENIAC的后续产品EDVAC（Electronic Discrete Variable Automatic Computer）。特别是在1946年与亚瑟·W·伯克斯（Arthur W.Burks）和荷曼·哥斯廷（Herman H.Goldstine）共同执笔的题为“电子计算器件逻辑设计的初步分析及讨论（Preliminary Discussion of the Logical Design of an Electronic Computing Instrument）”的论文中，他描述了几个EDVAC比ENIAC更加先进的特点。EDVAC的设计者们感觉到计算机内部中应当使用二进制数，而ENIAC使用的是十进制数。同时他们认为计算机中应当拥有尽可能大容量的存储器，这些存储器应该用来存储程序代码和程序执行中产生的数据（再说明一下，这些在ENIAC中都是不能实现的，对于ENIAC来说，编程不过是扳动开关和插拔电线的事情）。这些指令在存储器中是顺序存放的，而且可以由程序计数器进行寻址，但允许条件跳转。这就是著名的“存储程序概念”（stored-program concept）。
- 这些设计上的决策是计算机历史中非常重要的一个进化阶段，现在我们称之为“冯·诺依曼结构”。上一章中设计的计算机就是一个经典冯·诺依曼计算机。但是伴随着冯·诺依曼结构，又出现**冯·诺依曼瓶颈（von Neumann bottleneck）**。在冯·诺依曼计算机中，为了执行指令通常需要花费大量的时间先将这些指令从存储器中取出来。我们仔细回忆一下，第17章中最后设计的计算机需要花费3/4的时间用来取指令。
- 在EDVAC的那个时期，考虑到成本效益，用真空管制造大容量存储器是不可行的，因此那时提出了一些临时的替代方案。其中一个成功的方案是“水银延迟线路存储器”（mercury delay linememory），它使用5英尺水银真空管，在管子的一端每隔1μs向水银发送一个短脉冲，这些短脉冲大约需要1ms到达管子的另一端（可以如同检测声波一样检测到这些短脉冲，并折回开始端），因此一个水银管可以存储大约1024位的信息。
- 直到20世纪50年代中期人们才开发出了“磁芯存储器”（magnetic core memory）。众多的被磁化的小金属环由电线串起来组成了磁芯存储器。每一个小金属环可以存储1位信息。磁芯存储器沿用了很长一段时间才被别的技术取代，所以常常会听见老一辈程序员们把存储器的访问过程叫做“访问磁芯”。

- 20世纪40年代，对计算机本质进行概念化设想的并非只有约翰·冯·诺依曼一人。
  - 克劳德·香农 (Claude Shannon, 生于1916年) 是另外一个非常有影响力的思想家。在第11章中讨论了他1938年的硕士论文，正是这篇文章确定了开关、继电器以及布尔代数之间的关系。在1948年为贝尔电话实验室工作期间，香农在Bell System Technical Journal上发表了一篇题为“通信过程中的代数理论” (A Mathematical Theory of Communication) 的文章，在这篇文章中他不仅将“位”的概念介绍给了世界，更开创了一个新的研究领域，即著名的“信息论” (information theory)。信息论研究的是数字信息在有噪声（这些噪声通常阻止信息的通过）的情况下传输，以及如何弥补因噪声产生的损失。1949年，他撰写了第一篇关于如何编程可以让计算机下棋的文章，1952年他设计了一个通过继电器控制的机械鼠，它可以在一个迷宫中记住路径。骑单车、变戏法这些耍宝也使得香农在贝尔实验室声名鹊起。
  - 诺博尔特·韦纳 (Norbert Wiener, 1894-1964) 从哈佛大学获得数学博士学位时只有18岁，其撰写的Cybernetics, or Control and Communication in the Animal and Machine (1948年) 一书使他闻名于世。他使用新创词汇“控制论” (cybernetics: 源于希腊语舵手) 表示人类和动物的生物过程同计算机和机器人的机械原理之间的关系。大众文化中，人们普遍使用cyber作为前缀表示与计算机相关的一切，最著名的一个词，数百万台计算机通过因特网相连被称做“cyberspace”（网络空间），这个词源于计算机科幻小说作家威廉·吉布森 (William Gibson) 在1984年发表的小说Neuromancer中“cyberpunk”一词。
  - 1948年，埃克特与莫奇利 (Eckert-Mauchly) 计算机公司（后来成为雷明顿兰德公司一部分）开始开发第一台商用计算机——通用自动计算机，或者称为UNIVAC。这台机器于1951年完成，此后就被送到了人口普查局。UNIVAC在网络应用方面的首次亮相在哥伦比亚广播公司，它被用来预测1952年的总统选举结果。沃尔特·克朗凯特 (Walter Cronkite) 将UNIVAC称做“electronic brain”（电脑）。同样是在1952年，IBM发布了其第一个商用计算机系统，代号701。
  - 自此开始了漫长的公司和政府的计算历史。尽管这段历史很有趣，但我们要追踪另一段历史——如何缩减计算机成本和体积以及让其走入寻常百姓家，这开始于1947年一个鲜为人知的电子技术突破。
- 许多年以来，贝尔电话实验室是一个让天才们对他们感兴趣的一切事物进行研究的地方。非常幸运的是，他们其中的一些人对计算机有浓厚的兴趣。上面提到的乔治和香农就是在贝尔实验室工作时对早期的计算机发展做出了重大贡献。后来，在20世纪70年代，贝尔实验室成为很有影响的计算机操作系统UNIX和编程语言C的诞生地，我们将在下面章节中介绍。
  - 当AT&T (美国电话电报公司) 正式将科学与技术的研究同其他的业务分割时，贝尔实验室内部结构发生了改变，在1925年1月1日成立了子公司。贝尔公司的最初目的是发展改良电话系统的相关技术，幸运的是在这种非常模糊的目标下可以做很多技术研究，但是对于电话系统而言，一个显而易见的长期目标是通过电线不失真的传播语音信号。
  - 从1912年开始，贝尔系统致力于真空管放大器的研究，为了能让电话系统使用真空管，对其进行相当数量的研究和设计方面的改进。尽管做了大量的工作，真空管仍然有许多必须改进的地方。真空管体积太大、耗能大，并且最终会烧毁，但是，它们在当时是唯一的选择。
  - 1947年12月16日，当贝尔实验室的两个物理学家约翰·巴丁 (John Bardeen, 1908-1991) 和沃尔特·布兰坦 (Walter Brattain, 1902-1987) 制作出另一种放大器时，所有的一切都发生了改变。这种新型的放大器用一块锗（一种半导体元素）平板和一条黄金薄片制成。一周之后，他们将这个东西演示给他们的老板威廉·肖克利 (William Shockley, 1910-1989)。这就是第一个“晶体管” (transistor)，它被一些人称为20世纪最重要的发明之一。

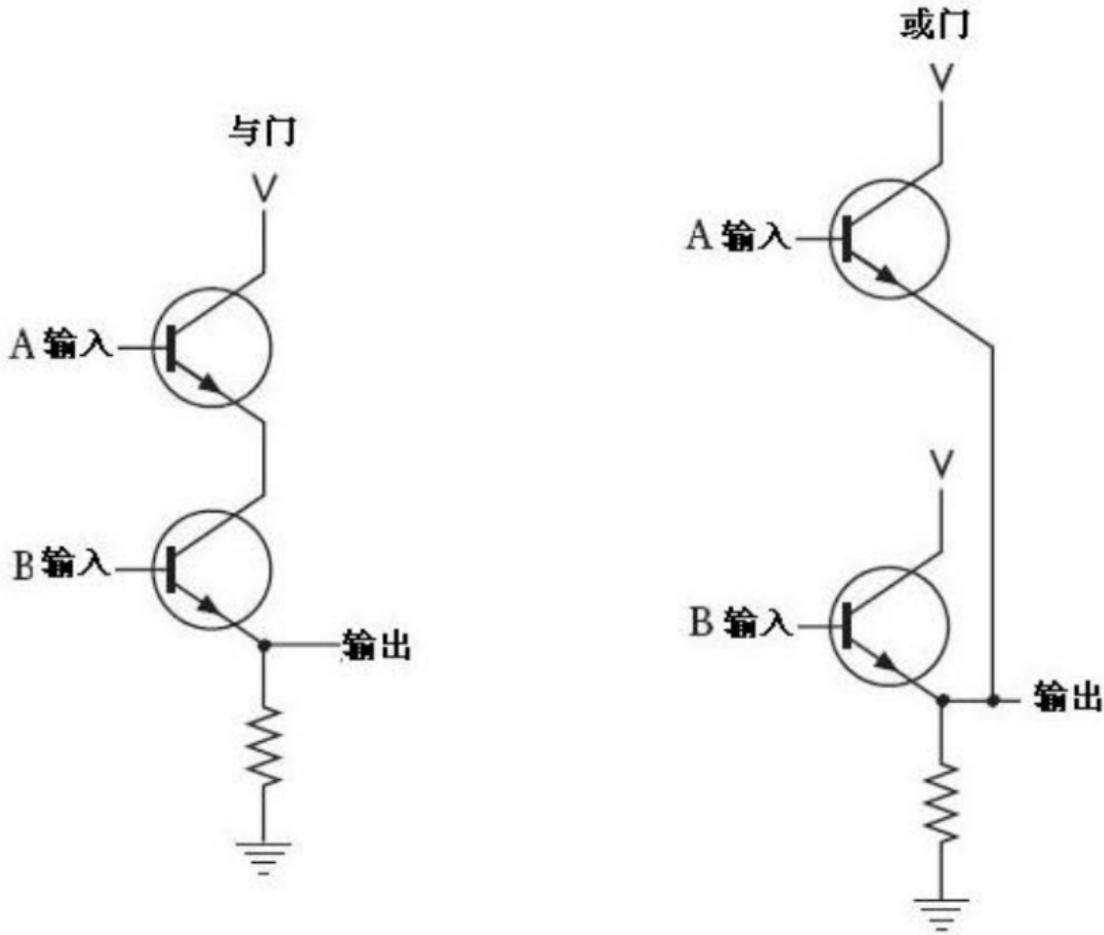
- 晶体管并不是凭空产生的。因为早在8年前，即1939年12月29日，肖克利在他的笔记本上写道“今天我突然想到，使用半导体来制作放大器从原理上讲比使用真空管更为可能。”在晶体管诞生后的几十年里，人们不断地完善它。1956年肖克利、巴丁和布兰坦“因为他们在半导体上的研究以及晶体管效应的发现”获得了当年的诺贝尔物理学奖。
- 在本书的开始部分探讨了导体和绝缘体。导体因为它们可以有利于电流的通过而得名。铜、银以及金都是很好的导体，元素周期表中这三种元素同属一列并非巧合。
  - 前面谈到过，原子中的电子分布在原子核外，并围绕原子核运动。这三种导体的共同特征是在原子核最外层都有一个单独的电子，而这个电子可以很容易地与原子中的其他电子剥离，因此可以自由移动形成电流。与导体对应的是绝缘体——比如橡胶和塑料——几乎不能导电。
  - 铋元素和硅元素（以及一些化合物）被称为“半导体”（semiconductor），之所以称为“半导体”并不是因为它们的导电性能是导体的一半，而是因为它们的导电系数可以通过多种方式操控。半导体的原子核在最外层有4个电子，是外层所能拥有的最大电子数目的一半。纯半导体中，原子之间形成稳定的化学键以及类似金刚石的结构。这种半导体不是良好的导体。
  - 但是，半导体可以掺入一些杂质，即与某些杂质组合。一种类型的杂质称做N型（N表示negative）半导体，它们为原子之间的结合提供额外的电子。另外一种类型的杂质被称做P型半导体。
  - 把一个P型半导体夹在两个N型半导体之间可以使之成为一个放大器。这就是著名的NPN晶体管，其三部分分别为集电极（collector）、基极（base），以及发射极（emitter）。下面是NPN晶体管原理示意图。



在基极施加微小的电压就可以控制非常大的电压从集电极到发射极。如果在基极没有施加电压，那么晶体管将不起作用。晶体管通常封装在直径为四分之一英寸的小金属罐中，并伸出三根金属线，外形如下图所示。

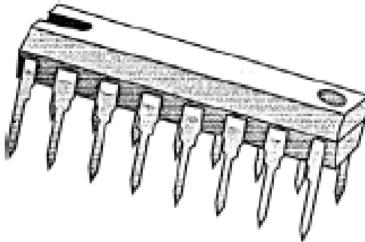


- 晶体管开创了固态电子器件的时代，即指晶体管不再需要真空而是使用固体制造，尤其是使用半导体以及当今最为常见的硅来制造。除了体积比真空管更小，晶体管需要的电量更小，产生的热量更少，而且持久耐用。随身携带一个真空管收音机是无法想象的一件事情。但晶体管收音机不同，它可以由一节电池供电，而且不会发烫。1954年，对于一些幸运的人来说，或许在圣诞节早上打开礼物盒时能获得一件可以随身携带的晶体管收音机。德州仪器公司，半导体革命中一个非常重要的公司，制造了第一批可以随身携带的晶体管收音机。
- 可是，晶体管真正的商业应用却始于助听器。为了纪念亚历山大·格雷厄姆·贝尔 (Alexander Graham Bell) 为聋人奉献毕生精力，AT&T公司允许助听器制造商无偿使用晶体管技术。晶体管电视机诞生于1960年，到现在电子管的应用几乎已经消失了（可是，并非完全消失，一些高保真音响爱好者以及电子吉他弹奏者较热衷于电子管设备，他们更喜欢真空管放大器产生的音质）。
- 1956年，肖克利离开了贝尔实验室成立了肖克利半导体实验室 (Shockley Semiconductor Laboratories)。他回到了自己出生的地方，加利福尼亚帕罗奥图市。他的公司是第一个落户于该地区的大公司。其他的半导体和计算机公司立刻也在该地区建立基业，旧金山南部的这个地区现在被人称为硅谷 (Silicon Valley)。
- 开发真空管的最初目的是为了放大电信号，但是它们同样可以应用在逻辑门的开关上，作用与晶体管一样。下面你将看到非常类似于继电器形式的由晶体管构造的与门。只有当A和B输入同时为1时晶体管才可以导通电流，从而输出为1。电阻的作用是预防短路。
  - 按照下图右边的方式连接两个晶体管可以组成一个或门。在与门中，上端晶体管的发射极连接下端晶体管的集电极。在或门中，两个晶体管的集电极都与电压源连接，两个发射极相互连接。



- 使用继电器构造逻辑门以及其他部件的方法对于晶体管同样是有效的。继电器、真空管以及晶体管最初都是为了开发放大器设计的，但是通过相似方式连接可以组成逻辑门，而计算机则是由这些部件构成的。1956年诞生了第一台晶体管计算机，随后的几年里，在新型计算机设计中电子管就被淘汰了。
- 有一个疑问：晶体管肯定可以使计算机更加可靠、体积更小以及需要的电量更少，但晶体管可以使计算机的结构变得更简单么？
  - 答案是否定的。晶体管允许在更小的空间里安装更多的逻辑门，但是你还是要考虑这些组件之间的互连问题。把晶体管连接起来构造逻辑门，与把继电器和真空管连接起来构造逻辑门一样困难。在某些方面来看，这更加困难，因为晶体管更加小而且不容易被控制。如果你想用晶体管制造第17章中的计算机和64 KB的RAM，设计工作的重要部分应当是构造某种可以放置所有部件的结构。而你的大部分体力劳动是在数百万只晶体管之中连接数百万根线，这是很乏味的。
- 可是，我们已经发现晶体管的某些组合具有特定功能，可以重复利用。一对晶体管可以连接成门，而门常常可以连接成振荡器、加法器、选择器，以及解码器。振荡器可以组成多位锁存器或者RAM阵列。如果把晶体管预先连接成常见的构件，再用其来组装计算机将更加容易。
  - 这种设想由英国物理学家杰弗里 (Geoffrey Dummer, 生于1909年) 在1952年5月的一次演讲中提出，他说：“我希望展望未来，”，接下来他提出了以下观点：“随着晶体管的出现以及半导体研究的广泛开展，现在也许可以设想将来会出现不采用连线而是由固体块组成的电子设备。这种固体块可能由绝缘层、导体层、整流层以及放大层四个层次组成，将不同层次的隔离区连接起来即可实现电子功能。”
  - 然而，真正可以使用的产品还需要再等上几年。

- 1958年7月，德州仪器公司的杰克·基尔比（Jack Kilby，生于1923年）想到了一个可以在一块硅片制造出多个晶体管、电阻和其他电子元件的方法，而他并不知道杰弗里预言。6个月过后，也就是1959年1月，罗伯特·诺依斯（Robert Noyce，1927-1990）也想到了类似的方法。诺依斯起初是为肖克利半导体实验室工作，但在1957年，他与其他7位科学家离开了肖克利半导体实验室创办了仙童（Fairchild）半导体公司。
  - 在技术的发展史中，同时产生一项发明是较常见的，这可能超出了人们的想象。尽管基尔比比诺依斯早6个月发明了这种设备，而且德州仪器公司先于仙童公司申请专利，但却是诺依斯首先获得了专利。因此产生了法律上的纠纷，但过了10年后，问题才得到令双方都满意的解决。尽管基尔比和诺依斯并没有在一起共事，但今天他们俩被称为集成电路，或者叫做IC（更通俗的说法是芯片）的共同发明者。
- 集成电路需要经过非常复杂的工艺流程才可以制造出来，包括将硅片分层，然后非常精确地掺入杂质以及蚀刻不同的区域形成微小组件。开发一种新的集成电路尽管很昂贵，但可以大量生产中获得效益——产量越大，价格就越便宜。
- 实际上，硅片是薄而且易碎的，因此它必须被安全地封装起来，这样不仅可以起到保护作用，还可以为芯片内部的部件与其他芯片之间的连接提供某种便利。集成电路有几种不同的封装方式，但最为常见的是采用矩形塑料双排直插式（或称为DIP），提供14、16或者40个管脚。



上图是一个有16个管脚的芯片。将芯片上的凹槽朝左放置（如图），用1到16对管脚进行编号，从左下角开始，环绕到右端，依次为1~16，16号管脚位于左边最上端。管脚之间的距离正好是1/10英寸。

- 纵观20世纪60年代，太空项目以及军备竞赛推动了早期的集成电路市场的发展。在民用方面，第一台用集成电路构造的商品是极点公司（Zenith）在1964年出售的助听器。1971年，德州仪器公司开始出售第一批便携计算器，同年，脉冲星公司（Pulsar）出售了第一块电子手表。随后其他利用了集成电路的产品陆续出现。
- 1965年，戈登·E·摩尔（Gordon E. Moore，当时在仙童公司工作，后来成为英特尔公司的合伙创办人）发现从1959年以后，技术在以这样一种方式发展：同一块芯片上可以集成的晶体管的数目每年翻一倍。他预测这种趋势将会持续。真实的发展速度比摩尔的发现稍慢一些，因此摩尔定律（最终命名）被修正为：每18个月同一块芯片上集成晶体管数目就会翻一倍。这仍是一个令人吃惊的速度，它解释了为什么刚刚过了几年就家用计算机好像已经过时了。一些人相信直到2015年摩尔定律仍然有效。
- 发展的早期，人们常常谈论小规模集成电路（small-scale integration），即SSI，指那些逻辑门少于10个的芯片；中规模集成电路（medium-scale integration），即MSI（包含10到100个逻辑门）；大规模集成电路（large-scale integration），即LSI（包含100到5000个逻辑门）。随后的术语为特大规模集成电路（very-large-scale integration），即VLSI（包含5000到50,000个逻辑门）；超大规模集成电路（super-large-scale integration），即SLSI（包含50,000到100,000个逻辑门）；超特大规模集成电路（ultra-large-scale integration，超过100,000个逻辑门）。
- 本章的剩余部分以及下一章，我想将时间停留在20世纪70年代中期，此时正是第一部《星球大战》电影发行前的时代，而VLSI处于萌芽阶段。那时，人们使用几种不同的技术来制造集成电路的组件。有时每一种技术被称之为一个IC家族，到20世纪70年代中期，有两个“家族”盛行开来：TTL和CMOS。

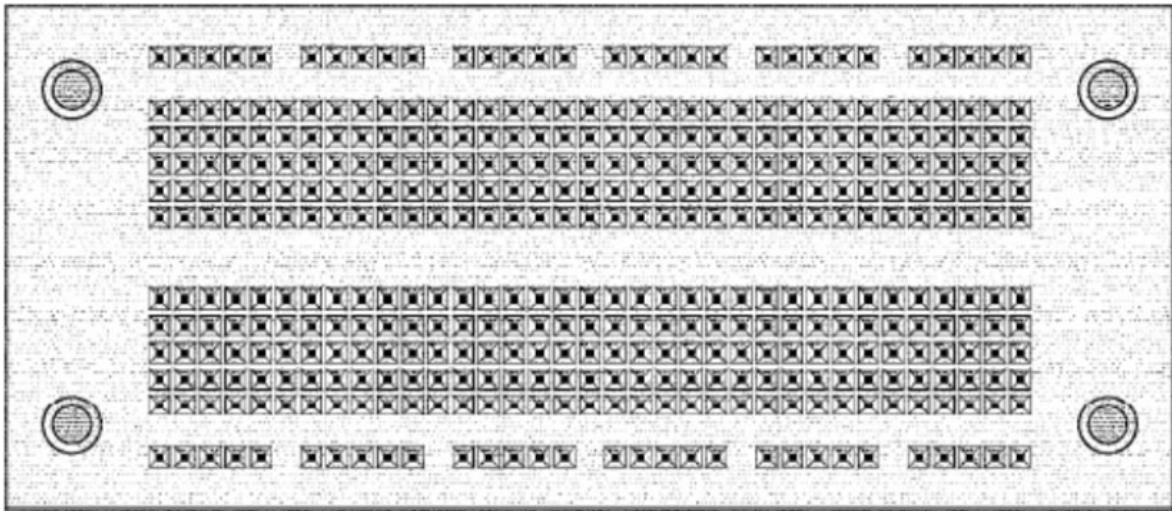
- TTL代表transistor-transistor logic（晶体管-晶体管逻辑）。20世纪70年代中期，如果你身为一名数字电路设计师（用IC设计大规模电路），那么一本1.25英寸厚、由德州仪器公司在1973年出版的名为The TTL Data Book for Design Engineers（《TTL工程师设计数据手册》，以下简称《TTL数据手册》）的书将会是你书桌上的常客。这是一本德州仪器和其他几个公司出售的TTL集成电路7400系列完整的参考书，由之所以这样称呼是因为这个IC“家族”的每一名“成员”都是以数字74开头。

- 7400系列中的每一个集成电路都是由以特定方式连接的预留逻辑门组成。一些芯片提供简单的预留的逻辑门，设计者可以用它们来组成更大规模的组件；另外一些芯片则提供通用组件，例如：触发器、加法器、选择器以及解码器。
- V<sub>cc</sub>与符号V一样，用来代表电压（顺便说一下，大写字母V的双下标代表电压源。下标的字母C指晶体管的电压输入端，即集电极，collector）。管脚标注的GND代表接地（ground）。在特定电路中使用的所有集成电路都必须有接电源端与接地端，缩写Nc表示无连接（no connection）。
- TTL中，当电压介于0~0.8V任一值则可以认为是逻辑“0”，而介于2~5V则可以认为是逻辑“1”。0.8~2V范围的电压输入则应当尽量避免。
- TTL的典型输出是以0.2V表示逻辑“0”，以3.4V表示逻辑“1”。考虑到电压值不稳定，有时会有一些波动，集成电路的输入和输出端有时不用“0”和“1”表示，而是用“低”和“高”表示。此外，有时候低电压可以表示逻辑“1”，而高电压则可以表示逻辑“0”，这种配置称为“负逻辑”。7400芯片被称为“四个双输入正与非门”，而这里的“正”则代表了上述所讲的正逻辑。
- 影响一个集成电路性能的最重要因素可以认为是传播时间（propagation time），也就是输入端发生变化引起输出端发生相应变化所需要的时间。
  - 通常以纳秒来衡量芯片的传播时间，缩写为nsec，即ns。1纳秒是非常短的时间。千分之一秒称为毫秒，百万分之一秒称之为微秒，那么十亿分之一秒则称为纳秒。7400芯片中与非门的传播时间应该保证小于22 ns，即0.000000022s。
  - 纳秒使计算机成为可能。正如在第17章中看到的，计算机处理器迟钝地做着简单的事情——从存储器中取出一个字节放到寄存器中，再将两个字节相加，然后将结果存放回存储器。迅速地完成这些操作是计算机（并非第17章中的计算机，而是当今使用的计算机）能完成任何实际工作的唯一原因。诺依斯说过：“当更好地认识了纳秒后，从概念上来讲计算机操作是相当简单的”。
- 作为一名数字电路设计工程师，你应当多花点时间看完《TTL数据手册》这本书，使自己熟悉用得到的TTL芯片类型。一旦你熟知了使用的工具，那么就可以使用TTL芯片构造一台第17章中的计算机。将芯片连接起来比将一个个的晶体管连接起来容易得多，但你可能不会考虑使用TTL来做64KB RAM阵列。1973年出版的《TTL数据手册》中，列出的容量最大的RAM芯片仅仅只有256×1位，制造64 KB需要2048个芯片！对制造存储器来讲，TTL绝非最好的技术。关于存储器将在第21章中详细讨论。
- 或许你想使用更好一点的振荡器。只要将TTL反相器的输出连接到输入，就会获得一个振荡器，而且其振荡频率更容易计算。这种振荡器使用石英晶体制造相当简单，石英晶体放在带有两个引线的密封小扁罐中。这些石英晶体的振荡频率在一个特定的值，通常情况下是每秒至少振荡一百万个周期，称1兆赫兹，缩写为MHz。如果要使用TTL制造第17章中的计算机，那么需要时钟频率为10 MHz才可使其运行良好，每条指令执行时间为400 ns。当然，这比使用继电器来做任何我们所构想的事情都要快。

◦ 芯片家族中另一位明星（至今仍是）是CMOS，CMOS表示互补金属氧化物半导体（complementary metal-oxide semiconductor）。如果你是一名20世纪70年代中期的使用CMOS集成电路进行电路设计的爱好者，那么可能会使用到一本名为《CMOS数据手册》的书作为参考源，该书由美国国家半导体公司出版，可以在当地的Radio Shack商店买到，书中涵盖了CMOS家族中4000系列的IC的信息。

- TTL供电电压要求在4.75 ~ 5.25V范围内。而对于CMOS来说，范围在3 ~ 18V内的电压均可，这是非常灵活的。此外，CMOS相比TTL需要更少的能量，这使得用电池运行小型CMOS电路变得可行。
- CMOS的缺点是速度慢，比如，在供电电压为5伏的情况下，CMOS4008 4位全加器可以保证的传播时间只有750 ns。CMOS芯片的传播速度会随着电压提高而提高——10V时为250ns，15伏时为190ns，但仍不能与TTL4位加法器的24 ns的传播时间相媲美（25年前，TTL的速度与CMOS低功率之间的权衡是非常清晰的，斗转星移，今天TTL已经拥有了低功率版本而CMOS也有了高速版本）。

- 在实际应用中，芯片的连接是在一个塑料“面包板”（如下图所示）完成的。

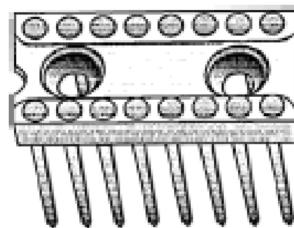


每一短行有5个孔，在塑料板背面这5个孔通过电线相连。

把芯片插入到面包板中，芯片将横跨在中间长槽的两侧，芯片的管脚则分别插到槽两侧的孔里，这样芯片的每一个管脚都会与其他四个孔里的管脚相连接，通过在孔之间连接电线可以实现芯片之间的连接。

在同一列中的5个插孔是互相连通的；列和列之间以及凹槽上下部分是不连通的。

使用一种叫做“钢丝包装”（wire-wrapping）的技术可以使芯片之间连接更加牢固，芯片插入到带有几个长长的方柱的插槽中，如下图所示。



每一个柱体对应芯片的一个管脚，而插口本身则插入到事先穿孔的薄板中。在板子的另一面，使用特殊的钢丝包装枪将每一个柱体周围紧紧包上绝缘线。柱体的直角边缘则从绝缘线中破出，与导线相连。

- 如果实际应用中使用集成电路制造一个特定的电路，那可能会用到“印刷电路板”（printed circuit board）。很久以前，这是集成电路爱好者做的事情。这种电路板上面布满了洞，并且被一层薄铜片覆盖。基本上，可以让防酸物质覆盖铜片上所有你想保护的地方，而使用酸蚀刻剩余的部分，接着就可以将集成电路的插口（或者是集成电路本身）直接焊接到电路板上的铜片上，但由于集成电路中存在很多互相连接，仅覆盖一层铜片通常情况下无法完成电路，所以商业制造的印刷电路板有多层互连。
- 到20世纪70年代早期，使用集成电路在一块电路板上制造一个完整的计算机处理器变得可能，实际上这距离将整个处理器放入一块芯片中，只是一个时间问题。虽然德州仪器公司在1971年为一块单芯片计算机提交了专利申请，但真正制造出一块这种单片机芯片的荣誉却属于英特尔公司（英特尔公司成立于1968年，由仙童公司以前的雇员罗伯特·诺伊斯和戈登·摩尔合伙创办）。1970年，英特尔发售了第一款产品，一个可以存储1024位数据的芯片，在当时这是单一芯片中可以存储的最大位数。
- 英特尔在为日本吉康（Busicom）公司生产的可编程计算器设计芯片的过程中，决定采取一种不同的方法。正如英特尔工程师特德·霍夫（Ted Hoff）说的：“我想让它成为一个具有通用功能的计算机，进而可以通过编程成为一个计算器，而不是使这个设备成为一个只有一些编程能力的计算器，”这导致了Intel 4004的产生，它是第一块“计算机芯片”，或者叫做“微处理器”。1971年11月，4040芯片已经可以得到使用，它拥有2300个晶体管（依照摩尔定律，18年后微处理器包含的晶体管数将是这个数字的4000倍，或者说是1000万。这是一个相当精确的预测）。
- 我们已经知道4004芯片包含的晶体管数目，下面是4004芯片另外三种重要的特征。自4004芯片开始，在比较微处理器性能时，通常采用三个衡量标准。
  - 第一个标准：4004是一个4位微处理器，这意味着处理器中数据通路宽度只有4位。每次做加、减运算时，它只能处理4位的数字。对比来看，第17章中的计算机数据通路是8位，因此被称为8位处理器。我们即将看到8位处理器很快就超越了4位处理器。但技术并没有停止于此，20世纪70年代末期，16位微处理器已经得到了应用。回想一下第17章中的内容，以及在8位处理器中进行两个16位数加法所必需的指令码，你就会欣喜地发现16位处理器带来的优势。到20世纪80年代中期，32位微处理器诞生了，并自此一直作为家用计算机的主要处理器。
  - 第二个标准：4004每秒最大时钟频率为108,000周期，即108KHz。时钟频率是指连接到微处理器并驱动它运行的振荡器的最大频率，超过此时钟频率，微处理器将不能正常工作。到1999年，家用计算机的微处理器已经达到了500 MHz——比4004要快5000倍。
  - 第三个标准：4004的可寻址存储器只有640字节，现在来看这个数字小得有点荒唐，但这与当时可得的存储芯片的容量是一致的。下一章中你将会看到，两年之中微处理器的寻址能力就达到了64 KB，与第17章中计算机的能力比肩。1999年英特尔生产的芯片可以寻址64TB的空间，尽管当时多数人的家用电脑RAM容量还不到256 MB。
- 上述三个数字指标并不能影响一台计算机的计算能力。比如，4位处理器同样可以实现32位数字加法，只不过是将其简单拆分为4位的数来进行。某种意义上讲，所有的数字计算机都是相同的，如果一台处理器从硬件上无法做到另外一台可以做到的事情，那么它可以通过软件途径做到，最终它们可以完成相同的事情，这是1937年图灵在论文里面关于可计算性的一种定义。
  - 然而，速度是处理器之间的根本不同点，同时速度也是我们使用计算机的一大原因。
- 最大时钟频率（maximum clock speed），也称为主频，是影响处理器速度的决定性因素之一。
- 时钟频率决定了执行一条指令所需要的时间，处理器的数据位宽也影响处理器的速度。尽管4位处理器可以完成32位数字的加法，但速度是不能与32位处理器相媲美的。
- 然而，令人感到迷惑的是，处理器可寻址存储器的最大空间对处理器速度也是有影响的，首先，可寻址存储器看上去只反映了处理器的某些能力，尤其是在需要大容量存储器的前提下进行数据处理的能力，而与处理器速度无关。

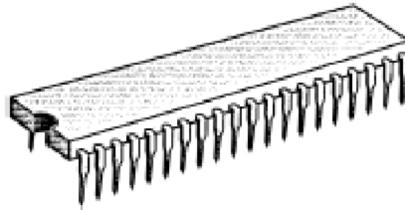
- 但其实，处理器可以利用某些存储器地址去控制其他的介质来存取信息，这样就绕开了存储器容量的限制（比如，假设在特定的存储器地址写入一个字节就在一个纸带上穿一个孔，而从存储器中读出一个字节等于从纸带上读取一个孔一样）。可是这种处理办法会降低整个计算机的速度——这就又回到了速度问题！
- 当然，这三个数字只能粗略地反映微处理器操作的速度，它们并不能体现出微处理器的内部构造以及机器指令代码的效率和能力。随着处理器变得越来越复杂，以前通过软件完成的任务都可以在处理器上完成，我们将会在后面的章节中看到这方面的例子。
- 即使所有的计算机具有相同的计算能力，即使它们只能做和图灵设计的早期计算机一样简单的事情，但有一点是无法回避的，那就是处理器的速度最终决定了其用途。
  - 比如那些表现比人脑还慢的计算机是毫无用处的，跟进一步来说，如果处理器需要用一分钟来画一帧图像，那么对于现代计算机而言，要想在其屏幕上播放电影也是不可能实现的。
- 回到20世纪70年代中期，虽然4004有很多局限性，但毕竟只是个开始。到1972年4月，英特尔发布了8008芯片——一个时钟频率为200KHz、可寻址空间为16 KB的8位微处理器（瞧，用三个数字来总结一个处理器是多么简单的事）。后来在1974年5月，英特尔公司和摩托罗拉公司同时发布了8008微处理器的改进版，正是两款芯片改变了整个世界。

## 19 两种典型的微处理器

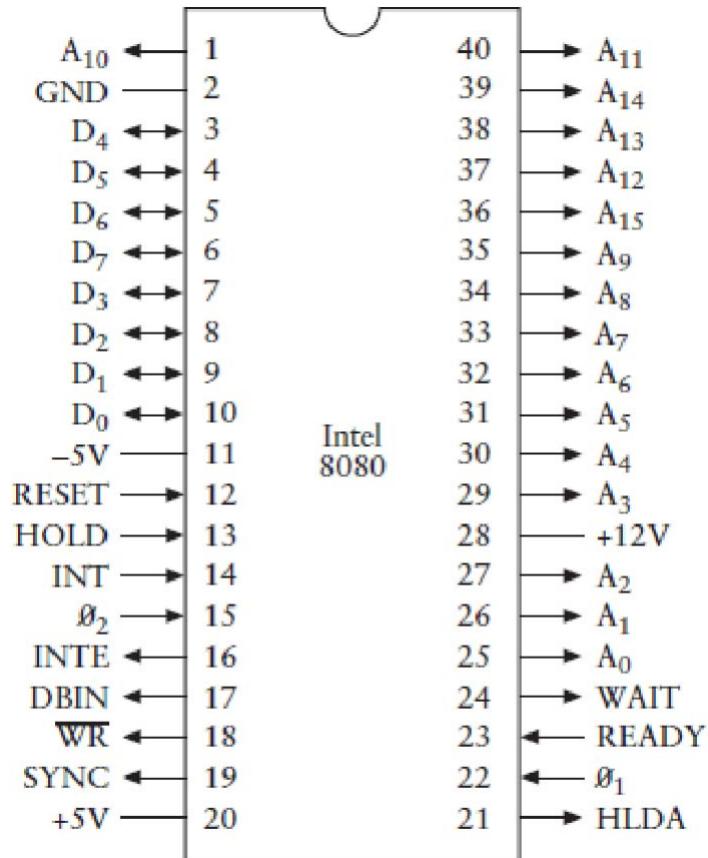
---

- 微处理器——正是它，将计算机中央处理器的所有构成组件整合在一起，集成在一个硅芯片上——诞生于1971年。它的诞生有着很好的开端：第一个微处理器，即Intel 4004系列，包括了2300个晶体管。到现在，大约三十年过去了，家用计算机的微处理器中的晶体管数量也逐步逼近10,000,000个。从本质上说，微处理器实际上所做的工作一直没有变。在现在的芯片上，新增的几百万个晶体管所做的很多事情令我们眼前一亮，但我们正处于微处理器探索的初期，过多的关心这些当代的芯片并不合适，因为它们只会分散我们的注意力而无法帮助我们去学习与理解它。为了更清晰地认识微处理器是如何工作的，让我们首先来看一下最原始的微处理器。
- 我们要讨论的微处理器出现于1974年。在这一年，英特尔公司在4月推出了8080处理器，摩托罗拉公司——从20世纪50年代生产半导体和晶体管——在8月推出了6800处理器。不仅如此，当年还有其他的一些微处理器面世。同年，德克萨斯仪器设备公司 (Texas Instruments) 推出了4位的处理器TMS 1000，它用于多种计算器、玩具和设备；国家半导体公司 (National Semiconductor) 推出了PACE——首个16位微处理器。但当我们回顾历史的时候就会发现，8080和6800是两个最具有重大历史意义的芯片。
  - 8080是一个8位的微处理器，它包括6000个晶体管，运行的时钟频率为2 MHz，寻址空间为64 KB。摩托罗拉的6800（今天的售价也是1.95美元）包括4000个晶体管，其寻址空间也是64 KB。第一个版本的6800的运行速度为1 MHz，但摩托罗拉于1977年推出了运行速度分别为1.5 MHz和2 MHz的版本。
- 这些芯片被称为“单芯片微处理器” (single-chip microprocessors)，不太准确的说法是“单芯片的计算机”。处理器只是计算机的一部分。除了处理器之外，计算机还需要其他一些设备，至少要包括一些随机访问的存储器 (RAM)，一些方便用户把信息输入计算机的设备（输入设备），一些使用户能够把信息从计算机中读取出来的设备（输出设备），以及其他一些能把所有构件连接在一块的芯片。
- 当描述微处理器的时候，我们总是习惯用一些框图来阐明其内部的构件及其连接情况。然而，在第17章我们已经使用了数不清的框图来描述它，现在我们将观察微处理器和外部设备的交互过程，以此来认识其内部的结构和工作原理。换句话说，为了弄清微处理器的工作原理，我们把它视做一个不需要详细研究其内部操作的黑盒。取而代之的方法是通过观测芯片的输入、输出信号，特别是芯片的指令集来理解微处理器的工作原理。

- 8080和6800都是40个管脚的集成电路。这些芯片最常见的IC封装大约为2英寸长，1/2英寸宽，1/8英寸厚。



当然，你所看到的只是外部的封装。其内部的硅晶片是非常小的，例如在早期的8位微处理器中，硅晶片还不到1/4平方英寸。外包装可以保护内部的硅晶片，并且通过管脚提供了处理器的输入和输出访问接入点。下面给出了8080的40个管脚的功能说明图。



本书中我们所创建的所有电气或电子设备都需要某种电源来供电。8080的一个特殊的地方就是它需要三种电源电压：管脚20必须接到5V的电压；管脚11需要接到-5V的电压；管脚28需接12V的电压；管脚2接地。（英特尔在1976年发布了8085芯片，目的就是简化对这些电源的要求）

其他的管脚都标有箭头。从芯片引出的箭头表明这是一个输出（output）信号，这种信号由微处理器控制，计算机的其他芯片对该信号响应。指向芯片的箭头表明该信号是一个输入（input）信号，该信号由其他芯片发出，并由8080芯片对其响应。还有一些管脚既是输入又是输出。

第17章所设计的处理器需要一个振荡器来使其工作。8080需要两个不同的同步时钟输入，它们的频率都是2 MHz，分别标记为 $\Phi_1$ 和 $\Phi_2$ ，位于管脚22和15上。这些信号可以很方便地由英特尔生产的8224时钟信号发生器产生。为8224连接一个18 MHz的石英晶体后，它基本上就可以完成其余工作了。

- 一个微处理器通常有多个用来寻址存储器的输出信号。用于寻址的输出信号的数目与微处理器的可寻址空间大小直接相关。
  - 8080有16个用于寻址的输出信号，标记为 $A_0 \sim A_{15}$ ，因此它的可寻址空间大小为 $2^{16}$ ，即65,536字节。

- 8080是一个8位的微处理器，可以一次从存储器读取或向存储器写入8位数据。该芯片还包括标记为D0 ~ D7的8个信号，这些信号是芯片仅有的几个既可以用做输入又可以用做输出的信号。当微处理器从存储器中读取一个字节时，这些管脚的功能是输入；当微处理器向存储器写入一个字节时，其功能又变成了输出。
- 芯片的其余10个管脚是控制信号（control signals）。例如，RESET（复位）输入用于控制微处理器的复位。输出信号/WR的功能是指明微处理器需要向RAM中写入数据（/WR信号对应于RAM阵列的写输入）。此外，当芯片读取指令时，在某些时刻一些控制信号会出现在D0 ~ D7管脚处。使用8080芯片构建的计算机系统通常使用8228系统控制芯片来锁存附加的控制信号。本章在后面将会讲述一些控制信号。但8080的控制信号是极其复杂的，因此，除非你准备用该芯片搭建一台计算机，否则最好不要在这些控制信号上过多花费时间。
- 假设8080微处理器连接了一个64KB的存储器，这样我们就能独立地读写数据而不依赖于微处理器。

8080芯片复位后，它把锁存在存储器0000h地址处的字节读入微处理器，通过在地址信号端A0 ~ A15输出16个0实现该过程。它读取的字节必须是8080指令，读取该字节的过程被称为取指令（instruction fetch）。

在第17章设计的计算机中，所有的指令（除了HLT指令）都是3个字节长，包括1字节的操作码和2字节的地址。在8080中，指令的长度可以是1字节、2字节，或者3字节。

有些指令使8080从存储器的一个特定地址读取字节到微处理器，有些指令使8080将一个字节从微处理器写入存储器的特定地址；还有些指令使8080在其内部执行而不需要访问RAM。

8080执行完第一条指令后，接着从存储器读取第二条指令，并依此类推。这些指令组合在一起构成了计算机程序，可以用来做一些很有趣的事情。

当8080以最高速度2 MHz运行时，每个时钟周期是500ns ( $1 \div 2,000,000 = 0.000000500s$ )。第17章中的计算机的所有指令都需要4个时钟周期，8080的每条指令需要4 ~ 18个时钟周期，这就意味着每条指令的执行时间为2 ~ 9μs。

- 也许了解某个特定微处理器的功能的最好办法就是全面地测试其完整的指令集。

第17章最后完成的计算机仅包括12条指令。一个8位处理器的指令数很容易达到256，每一条指令的操作码就是一个特定的8位数（如果某些指令包含2字节的操作码，其指令集会更大）。8080虽然没有这么多指令，但是其指令数也已经达到了244。这看起来似乎是很多，但从总体上说，其功能并不比第17章的计算机强大。例如，如果想利用8080进行乘法或除法运算，你仍然需要自己写一小段代码。

在第17章曾经讲到过，为了方便地引用指令，我们为处理器的每一条指令的操作码都指派了一个特殊的助记符，而且其中的一些助记符是可以带有参数的。这种助记符只是在我们使用操作码时提供方便，它对于处理器是没有帮助的，处理器只能读取字节，对于助记符组成的文本的含义一无所知（为了讲解清楚，本书选用了Intel 8080说明文档中用到的部分助记符为例来说明）。

同第17章的累加器一样，8080的8位累加器也记做A。8080也有与第17章的计算机的Load指令和Store指令功能相同的两条指令，它们也称做加载（Load）和保存（Store）。在8080中，加载指令和保存指令的操作码分别是32h和3Ah，每个操作后面也同样跟着一个16位的地址。在8080中，它们的助记符分别是STA（Store Accumulator，表示加载到累加器）和LDA（Load Accumulator，表示保存到累加器）：操作码32 指令STA [aaaa]，A，操作码3A 指令LDA A，[aaaa]

8080芯片的微处理器的内部除累加器外还设置了6个寄存器（register），每个寄存器可以存放一个8位的数。这些寄存器和累加器非常相似，事实上累加器被视为一种特殊的寄存器。这6个寄存器和累加器一样，本质上都是锁存器。处理器既可以将数据从存储器读入寄存器，也可以将数据从寄存器存回存储器。当然，其他的寄存器没有累加器所具有的丰富的功能，例如，当把两个8位数相加时，其结果总是保存到累加器而不会保存到其他寄存器。

在8080中用B, C, D, E, H和L来表示新增的6个寄存器。人们通常会问以下两个问题：“为什么不使用F和G来表示？”，以及“I, J和K用来代表什么？”

答案是，使用H和L来命名寄存器是因为它们具有特殊的含义，H可以代表高（High）而L可以代表低（Low）。通常把两个8位的寄存器H和L结合起来构成一个16位的寄存器对（register pair），称做HL，H用来保存高字节而L用来保存低字节。这个16位的值通常用来对存储器寻址，我们将在下面看到它是怎样以简单的方式工作的。

- 寄存器是计算机必不可少的部件吗？为什么在第17章搭建的计算机中并没有寄存器的踪迹？从理论上讲，这些寄存器不是必需的，在第17章也没有用到它们，但在实际应用中使用它们将带来很大的方便。很多计算机程序都同时用到多个数据，将这些数据存放在寄存器比存放在存储器更便于访问，因为程序访问内存的次数越少其执行速度就越快。
- 在8080中有一条指令至少用到了63个操作码，这条指令就是MOV，即Move的缩写。其实该指令是一条单字节指令，它主要用来把一个寄存器中的内容转移到另一个寄存器（也可能就是原来的寄存器）。因为8080微处理器设计了7个寄存器（包括累加器在内），因此应用中使用大量的MOV指令是很正常的。

下面列出了前32条MOV指令。再一次提醒你，两个参数中左侧的是目标操作数，右侧的是源操作数。

操作码	指 令	操作码	指 令
40	MOV B, B	50	MOV D, B
41	MOV B, C	51	MOV D, C
42	MOV B, D	52	MOV D, D
43	MOV B, E	53	MOV D, E
44	MOV B, H	54	MOV D, H
45	MOV B, L	55	MOV D, L
46	MOV B, [HL]	56	MOV D, [HL]
47	MOV B, A	57	MOV D, A
48	MOV C, B	58	MOV E, B
49	MOV C, C	59	MOV E, C
4A	MOV C, D	5A	MOV E, D
4B	MOV C, E	5B	MOV E, E
4C	MOV C, H	5C	MOV E, H
4D	MOV C, L	5D	MOV E, L
4E	MOV C, [HL]	5E	MOV E, [HL]
4F	MOV C, A	5F	MOV E, A

这些指令使用起来非常方便。利用上面的指令可以方便地把一个寄存器存放的数据转移到另一个寄存器。下面让我们研究一下以HL寄存器对作为操作数的4条指令。

MOV B, [HL]

前面讲过LDA指令，它可以把单字节的操作数从存储器转移到累加器；LDA操作码后面直接跟着该操作数的16位地址。在上面列出的指令中，MOV指令把字节从存储器转移到B寄存器，但该字节的16位地址却存放在HL寄存器对中。HL是怎样得到16位存储器地址的呢？这并不难解决，有很多方法可以做到，比如通过某种计算实现。

`LDA A, [aaaa]` 和 `MOV B, [HL]` 的功能都是把一个字节从内存读入微处理器，但它们寻址存储器的方式并不相同。第一种方式称做直接寻址 (direct addressing)；第二种方式称做间接寻址 (index addressing)。

下面列出了其余32条MOV指令，我们看到HL保存的16位存储器地址也可以作为目标操作数。

操作码	指令	操作码	指令
40	<code>MOV B, B</code>	50	<code>MOV D, B</code>
60	<code>MOV H, B</code>	70	<code>MOV [HL], B</code>
61	<code>MOV H, C</code>	71	<code>MOV [HL], C</code>
62	<code>MOV H, D</code>	72	<code>MOV [HL], D</code>
63	<code>MOV H, E</code>	73	<code>MOV [HL], E</code>
64	<code>MOV H, H</code>	74	<code>MOV [HL], H</code>
65	<code>MOV H, L</code>	75	<code>MOV [HL], L</code>
66	<code>MOV H, [HL]</code>	76	<code>HLT</code>
67	<code>MOV H, A</code>	77	<code>MOV [HL], A</code>
68	<code>MOV L, B</code>	78	<code>MOV A, B</code>
69	<code>MOV L, C</code>	79	<code>MOV A, C</code>
6A	<code>MOV L, D</code>	7A	<code>MOV A, D</code>
6B	<code>MOV L, E</code>	7B	<code>MOV A, E</code>
6C	<code>MOV L, H</code>	7C	<code>MOV A, H</code>
6D	<code>MOV L, L</code>	7D	<code>MOV A, L</code>
6E	<code>MOV L, [HL]</code>	7E	<code>MOV A, [HL]</code>
6F	<code>MOV L, A</code>	7F	<code>MOV A, A</code>

其中的一些指令如：`MOV A, A` 并不会执行有意义的操作。而指令：`MOV [HL], [HL]` 是不存在的，事实上，与之对应的指令是HLT (Halt) 即停止指令，也就是说该指令的意义是停止。

- 研究MOV操作码的位模式能更好地了解它，MOV操作码由8位组成：01dddsss。其中ddd这3位是目标操作数的代码，sss这3位是源操作数的代码。它们所表示的意义如下：

000 = 寄存器 B

001 = 寄存器 C

010 = 寄存器 D

011 = 寄存器 E

100 = 寄存器 H

101 = 寄存器 L

110 = 寄存器 HL 保存的存储器地址中的内容

111 = 累加器 A

例如，指令 `MOV L, E` 对应的操作码为：01101011，用十六进制数可表示为6Bh。这与前面列出的表格是一致的。

可以设想一下，在8080的内部可能是这样的：标记为sss的3位用于8-1数据选择器，标记为ddd的3位用来控制3-8译码器以此确定哪一个寄存器锁存了值。

寄存器B和C也可以组合成16位的寄存器对BC，同样我们还可以用D和E组成寄存器对DE。如果这些寄存器对也包含要读取或保存的字节的存储器地址，则可以用下面的指令实现：

操作码	指 令	操作码	指 令
02	<code>STAX [BC], A</code>	0A	<code>LDAX A, [BC]</code>
12	<code>STAX [DE], A</code>	1A	<code>LDAX A, [DE]</code>

另一种类型的传送（Move）指令称做传送立即数（Move Immediate），它的助记符写做MVI。传送立即数指令是一个双字节指令，第一个字节为操作码，第二个是数据。这个单字节数据从存储器转移到某个寄存器，或者转移到存储器中的某个存储单元，该存储单元由HL寄存器对寻址。

操作码	指 令
06	MVI B, xx
0E	MVI C, xx
16	MVI D, xx
1E	MVI E, xx
26	MVI H, xx
2E	MVI L, xx
36	MVI [HL], xx
3E	MVI A, xx

例如，当指令：MVI E, 37h 执行后，寄存器E存放的字节是37h。这就是我们要介绍的第三种寻址方式——**立即数寻址 (immediate addressing)**。

下面将列出一个操作码集，包括32个操作码，它们能完成4种基本的算术运算，这些运算在第17章设计处理器时我们已经熟悉了，它们是加法 (ADD)、进位加法 (ADC)、减法 (SUB) 和借位减法 (SBB)。可以看到，在所有的例子中，累加器始终用于存放其中的一个操作数，同时用来保存计算结果。这些指令如下。

操作码	指 令	操作码	指 令
80	ADD A, B	90	SUB A, B
81	ADD A, C	91	SUB A, C
82	ADD A, D	92	SUB A, D
83	ADD A, E	93	SUB A, E
84	ADD A, H	94	SUB A, H
85	ADD A, L	95	SUB A, L
86	ADD A, [HL]	96	SUB A, [HL]
87	ADD A, A	97	SUB A, A
88	ADC A, B	98	SBB A, B
89	ADC A, C	99	SBB A, C
8A	ADC A, D	9A	SBB A, D
8B	ADC A, E	9B	SBB A, E
8C	ADC A, H	9C	SBB A, H
8D	ADC A, L	9D	SBB A, L
8E	ADC A, [HL]	9E	SBB A, [HL]
8F	ADC A, A	9F	SBB A, A

假如累加器A存放的字节是35h，累加器B存放的字节是22h，经过减法运算：SUB A, B，累加器中的值变为22h，即两个字节的差。

如果累加器A中的值为35h，寄存器H和L中的值分别是10h和7Ch，而存储器地址107Ch处的字节为4Ah，指令：ADD A, [HL] 把累加器中的值（35h）与寄存器对HL寻址（107Ch）存储器得到的数值（4Ah）相加，并把计算结果（7Fh）保存到累加器。

在8080中，使用ADC指令和SBB指令可以对16位数、24位数、32位数甚至更高位的数进行加法、减法运算。例如，假设现在寄存器对BC和DE各自保存了一个16位的数，我们要把这两个数相加，并且把结果保存在寄存器对BC中。具体做法如下：

MOV A, C ; 低字节操作

ADD A, E

MOV C, A

MOV A, B ; 高字节操作

ADC A, D

MOV B, A

在上面的计算中，用ADD指令对低字节相加，用ADC指令对高字节相加。低字节相加产生的进位会进入高字节的运算中。在这段简短的代码中，我们用到了4个MOV指令，这是因为在8080中只能利用累加器进行加法运算，操作数在累加器和寄存器之间来回地传送，因此在8080的代码中会大量使用MOV指令。

- 现在我们来讨论8080的标志位 (flag)。第17章设计的处理器已经有了CF (进位标志位Carry Flag) 和ZF (零标志位) 两个标志位，在8080中又新增了3个标志位，包括符号标志位SF (Sign Flag)，奇偶标志位PF (Parity Flag) 和辅助进位标志位AF (Auxiliary Carry Flag)。在8080中，用一个专门的8位寄存器来存放所有标志位，该寄存器称做程序状态字 (Program Status Word, PSW)。不同的指令对标志位有不同的影响，LDA、STA或MOV指令始终都不会影响标志位，而ADD、SUB、ADC以及SBB指令会影响标志位的状态，具体情况如下。

- 如果运算结果的最高位是1，那么符号标志位SF标志位置1，表示该计算结果是负数。
- 如果运算结果为0，则零标志位ZF置0。
- 如果运算结果中“1”的位数是偶数，即具有偶数性 (evenparity)，则奇偶标志位PF置1；反之，如果“1”的位数是奇数，即运算结果具有奇数性 (odd parity)，则PF置0。由于PF的这个特点，有时会被用来进行简单的错误检查。PF在8080程序中并不常用。
- 进位标志位CF的情况和第17章描述的稍有不同，当ADD和ADC运算产生进位或者SUB和SBB运算不发生借位时，CF都置1。

下面的两条指令会直接影响进位标志位CF。

操作码	指 令	含 义
37	STC	令 CF 置 1
3F	CMC	令 CF 取反

- 第17章设计的计算机可以执行ADD、ADC、SUB和SBB指令（虽然缺乏灵活性），而8080功能更为强大，它还可以执行AND (与)、OR (或)、XOR (异或) 等逻辑运算。不论是算术运算还是逻辑运算，都是由8080处理器的算术逻辑单元 (ALU) 来完成的。

以下是8080的算术运算和逻辑运算指令。

操作码	指 令	操作码	指 令
A0	AND A, B	B0	OR A, B
A1	AND A, C	B1	OR A, C
A2	AND A, D	B2	OR A, D
A3	AND A, E	B3	OR A, E
A4	AND A, H	B4	OR A, H
A5	AND A, L	B5	OR A, L
A6	AND A, [HL]	B6	OR A, [HL]
A7	AND A, A	B7	OR A, A
A8	XOR A, B	B8	CMP A, B
A9	XOR A, C	B9	CMP A, C
AA	XOR A, D	BA	CMP A, D
AB	XOR A, E	BB	CMP A, E
AC	XOR A, H	BC	CMP A, H
AD	XOR A, L	BD	CMP A, L
AE	XOR A, [HL]	BE	CMP A, [HL]
AF	XOR A, A	BF	CMP A, A

AND、XOR和OR都是按位运算 (bitwise operations) 指令，也就是说对于这些逻辑运算指令，其操作数的每一个对应位都是独立运算的，例如：

```
MVI A, 0Fh  
MVI B, 55h  
AND A, B
```

保存到累加器的结果将会是05h。假如我们把3条指令换作OR，则最终的结果将会是5Fh；如果换作XOR，则结果又变成了5Ah。

- CMP (Compare, 比较) 指令同SUB指令类似，也是把两个数相减，不同之处在于它并不在累加器中保存计算结果，计算的目的是为了设置标志位。这个标志位的值可以告诉我们两个操作数之间的大小关系。例如，我们考虑下面的指令：

```
MVI B, 25h  
CMP A, B
```

指令执行后，累加器A中的值并没有变化。改变的是标志位的值，如果A中的值等于25h，则零标志位ZF置1；如果A中的值小于25h，则进位标志位CF置1。

同样的，也可以对立即数进行这8种算术逻辑操作。

操作码	指 令	操作码	指 令
C6	ADI A, xx	E6	ANI A, xx
CE	ACI A, xx	EE	XRI A, xx
D6	SUI A, xx	F6	ORI A, xx
DE	SBI A, xx	FE	CPI A, xx

例如，可以用下面的这条指令来替代上面列出的两条指令：CPI A, 25h

- 下面是两种特别的8080指令。

操作码	指 令
27	DAA
2F	CMA

CMA是Complement Accumulator的简写。它对累加器中的数按位取反，即把0变为1，1变为0。例如，累加器中的数如果是01100101，使用CMA命令后，累加器中的数按位取反，得到10011010。

我们还可以使用如下指令对累加器中的数取反：XRI A, FFh

前面提到过，DAA是Decimal Adjust Accumulator的缩写，即十进制调整累加器，它可能是8080中最复杂的一条指令。在8080微处理器中专门设计了一个完整的小部件用来执行该指令。

DAA指令提供了一种用二进制码表示十进制数的方法，称为**BCD码 (binary-coded decimal)**，程序员可以在该指令的帮助下实现十进制数的算术运算。BCD码采用的表示方式为，每4位为一段，每段所能表示数的范围是：0000 ~ 1001，对应十进制数的0 ~ 9。因为1字节有8位故可分割为2个段，因此在BCD码格式下，一个字节可以表示两位十进制数。

假设累加器A存放的是BCD码表示的27h，显然它就对应十进制数的27（通常，十六进制的27h对应的十进制数是39）。同时假设寄存器B中存放着BCD码表示的94h。假如执行如下指令：

```

MVI A, 27h
MVI B, 94h
ADD A, B

```

累加器中存放的最终结果是BBh，当然，这肯定不是BCD码。因为BCD码中每4位组成的段所能表示的十进制数不会超过9。然而，当我们继续执行指令：DAA

那么累加器最后所保存的值是21h，而且进位标志位CF置1。因为十进制的27与94相加的结果为121。由此可以看到，使用BCD码进行十进制的算术运算是很方便的。

- 在8080程序中，经常会对一个数进行加1或减1运算。在第17章的乘法程序中，为了实现对一个数减1，我们把该数与FFh相加，它是-1的补码。8080提供了专门的指令用来对寄存器或存储器中的数进行加1（称作增量）或减1（称作减量）操作。

操作码	指 令	操作码	指 令
04	INR B	05	DCR B
0C	INR C	0D	DCR C
14	INR D	15	DCR D
1C	INR E	1D	DCR E
24	INR H	25	DCR H
2C	INR L	2D	DCR L
34	INR [HL]	35	DCR [HL]
3C	INR A	3D	DCR A

INR和DCR都是单字节指令，它们可以影响除CF (Carry Flag) 之外的所有标志位。

- 8080还包括4个循环移位 (Rotate) 指令，这些指令可以把累加器中的内容向左或向右移动1位，它们的具体功能如下。

操作码	指 令	意 义
07	RLC	使累加器循环左移
0F	RRC	使累加器循环右移
17	RAL	带进位的累加器循环左移
1F	RAR	带进位的累加器循环右移

这些指令只对进位标志位CF有影响。

假设累加器中存放的数是A7h，即二进制的10100111。RLC指令使其每一位都向左移一位。最终的结果是，最低位（左端为低位，右端为高位）移出顶端移至尾部成为最高位，这条指令也会影响CF的状态。在这个例子中CF置1，最后的结果为01001111。RRC指令以同样的方式进行右移位操作。如果移位之前的数是10100111，执行RRC之后将变为11010011，同时CF置1。

较之RLC和RRC，RAL和RAR指令的工作方式稍有不同。执行RAL指令时，累加器中的数仍然按位左移，把CF中原来的值移至累加器中数值的最后一位，同时把累加器中数据的原最高位移至CF。例如，假设累加器中移位之前的数是10100111且CF为0，执行RAL指令后，累加器中的数变为01001110而CF变为1。类似的，如果执行的是RAR指令，累加器中的数变为01010011而CF变为1。

当我们在程序中需要对某个数进行乘2（左移）或除2（右移）运算时，使用移位操作会使运算变得非常简单。

- 我们通常把微处理器可以寻址访问的存储器称为**随机访问存储器 (random access memory, RAM)**，主要的原因是：只要提供了存储器地址（有多种方式），微处理器可以用非常简便的方式访问存储器的任意存储单元。RAM就像一本书，我们可以翻到它的任意一页。这种方式很方便，它不像存储在一个微缩胶片上的整个星期的报纸，为了阅读星期六的内容，我们需要扫描几乎整个星期的内容。

同样，它也比读取磁带快得多，因为当我们要听最后一首歌时，需要快进磁带的一整面。微缩胶片和磁带都不是随机访问的，它们是顺序访问（sequential access）的。

- 显然，随机访问存储器是非常好的一种寻址方式，对于经常访问存储器的微处理器来说更是如此。然而，在某些情况下使用不同的寻址方式访问存储器也是有好处的。例如，下面例子中的这种存储方式既不是随机的也不是顺序的：
  - 假设你在办公室工作，有人会到你办公桌前为你分配任务，每一项工作都用到某种文件夹。这些工作通常有这样的特点，在你完成某项工作之前首先要做另一项相关工作，并用到另一个文件夹。因此你只能放下第一个文件夹，并在它上面打开一个第二个文件夹继续工作。现在又有一个人给你分配了一个比前一项优先级更高的工作，于是你打开第三个文件夹放在前面两个上，继续工作。而这项工作也需要先做一项相关工作，于是你只好再打开第四个文件夹，现在你的办公桌上已经堆叠了四个文件夹了。
  - 你可能已经注意到了，事实上，这些堆叠的文件夹很有秩序地保存了你干活的顺序轨迹。最上面的文件夹总是代表优先级最高的工作，完成该工作之后就可以做接下来的工作了，依此类推。最后当你处理完办公桌上最后一个文件夹（即接受的第一个任务）后，就可以回家了。
- 这种形式的存储器称作**堆栈**（stack）。使用堆栈时，我们以从底部到顶部的顺序把数据存入堆栈，并以相反的顺序把数据从堆栈中取出，因此该技术也称作**后进先出存储器**（last-in-first-out, LIFO）。堆栈的特点是，最先保存到堆栈中的数据最后被取出，而最后保存的数据则被最先取出。

本章自此把stack翻译成立了“堆栈”，为了把堆和栈区别开来（而不是统一来看），最好统一改成“栈”。

堆(heap)是堆，栈(stack)是栈。只有堆(Heap)和栈(Stack)，没有“堆栈”。

同样，在计算机中也可以使用堆栈，当然计算机中的堆栈保存的是数据而不是工作。大量的实践证明，在计算机中使用堆栈技术是十分方便的。通常把将数据存入堆栈的过程称作压入（push），把从堆栈取出数据的过程称作弹出（pop）。

- 假设你正在编写一个汇编语言程序，需要用到寄存器A、B、C来存储数据。在编写程序的过程中，你注意到程序需要做一个小的计算，并且该计算也需要用到寄存器A、B、C。你希望在完成该计算之后仍然回到原来的地方，并且仍然使用寄存器A、B、C中原先存放的数据。

为了保存寄存器A、B、C原先存放的数据，可以简单地把这些数据保存到存储器中不同的地址中，需要进行的计算完成之后，再把这些数据从存储器转移到寄存器。但这种方式需要记录数据存放的地址。有了堆栈的概念之后，我们可以用一种更清晰的方式来处理这个问题，即把这些寄存器中的数据依次存放到底栈中。

```
PUSH A  
PUSH B  
PUSH C
```

这些指令以某种方式把寄存器中的内容保存到底栈中。这些指令执行之后，寄存器中原有的数据将妥善地保存下来，你就可以放心地使用这些寄存器进行别的工作了。为了取回原来的数据，可以使用POP指令把它们从堆栈中弹出，当然弹出的顺序和原来压入的顺序是相反的。相应的POP指令如下所示。

```
POP C  
POP B  
POP A
```

再次谨记：后进先出。如果POP指令的顺序弄错了，将会引起严重的错误。

我们可以在程序中多次用到堆栈而不会引起混乱，这是堆栈机制的一个特殊优势。例如，在我们要编写的程序中已经把寄存器A、B、C中的数保存到了堆栈，在程序的另一段又需要把寄存器C、D、E中的数保存到堆栈，可以使用下面的指令：

```
PUSH C  
PUSH D  
PUSH E
```

当然，在该段程序中还需要使用一些指令将保存到堆栈中的数据取回至寄存器，这些指令是：

```
POP E  
POP D  
POP C
```

显然，由于先进后出的原则，这些数据在先前存放的C、B、A中的数据之前弹出堆栈。

- 堆栈的功能是怎样实现的呢？首先，堆栈其实就是一段普通的RAM存储空间，只是这段空间相对独立不另作他用。8080微处理器设置了一个专门的16位寄存器对这段存储空间寻址，这个特殊的寄存器称为**堆栈指针 (SP, Stack Pointer)**。

在我们所举的例子中，对于8080来说使用PUSH和POP对寄存器操作实际上是不准确的。在8080中，执行PUSH指令实际上是把16位的数据保存到堆栈，执行POP指令是把这些数据从堆栈中取回至寄存器。因此，对于上面的如PUSH C，POP C等指令，我们对其进行如下的修改。

操作码	指令	操作码	指令
C5	PUSH BC	C1	POP BC
D5	PUSH DE	D1	POP DE
E5	PUSH HL	E1	POP HL
F5	PUSH PSW	F1	POP PSW

PUSH BC 指令将寄存器B和C中的数据保存到堆栈，而 POP BC 则将这些数据从堆栈取回到寄存器B和C中，并且保持原来的顺序。最后一行指令中的PSW代表程序状态字，如前所述，这是一个8位的寄存器，用于保存标志位。最后一行的PUSH和POP指令的操作对象实际上是累加器和PSW，即压入和弹出堆栈的数据由累加器和PSW中的内容组成。如果你想把所有寄存器中的数据及全部标志位都保存到堆栈，可以使用下面的指令：

```
PUSH PSW  
PUSH BC  
PUSH DE  
PUSH HL
```

- 堆栈是怎样工作的呢？我们假设堆栈指针是8000h，当执行PUSH BC指令时将会引发以下操作。
  - 堆栈指针减1，变为7FFFh。
  - 寄存器B中的内容被保存到堆栈指针指向的地址，即存储器地址7FFFh处。
  - 堆栈指针减1，变为7FFEH。

- 寄存器C中的内容被保存到堆栈指针指向的地址，即存储器地址7FFEh处。
  - 类似的，在堆栈指针仍为7FFEh的情况下，执行POP BC指令时会将上面的步骤反过来执行一遍：
    - 堆栈指针指向的地址 (7FFEh) 的内容加载到累加器C。
    - 堆栈指针加1，变为7FFFh。
    - 堆栈指针指向的地址 (7FFFh) 的内容加载到累加器B。
    - 堆栈指针加1，变为8000h。
  - 每执行一条PUSH指令，堆栈都会增加两个字节，这可能会导致程序出现一些小错误——堆栈可能会不断增大，最终覆盖掉存储器中保存的程序所必需的代码或数据。这种错误被称作**堆栈上溢 (stack overflow)**。类似的，如果在程序中过多地使用了POP指令，则会过早地取完堆栈中的数据从而导致类似错误，这种情况称为**堆栈下溢 (stack underflow)**。
  - 如果8080连接的是一个64 KB的存储器，你可能会把堆栈指针初始化为0000h。当执行第一条PUSH指令时，堆栈指针会减1变为FFFFh，即存储器的最后一个存储单元。这时，堆栈的初始位置将会是存储器的最高地址，因为程序的代码通常从0000h开始存放，因此两者将保持非常远的距离。
- 8080使用LXI指令为堆栈寄存器赋值，LXI是Load Extended Immediate的缩写，即加载扩展的立即数。下面的这些指令将把操作码后的两个字节保存到16位寄存器对中。

操作码	指 令
01	LXI BC, xxxx
11	LXI DE, xxxx
21	LXI HL, xxxx
31	LXI SP, xxxx

LXI指令保存一个字节。而且上表中最后一条LXI指令为堆栈指针赋了一个特殊的值。通常下面的指令作为微处理器复位之后首先执行的指令之一。

0000h: LXI SP, 0000h

类似的，还可以对寄存器对和堆栈指针进行加1或减1操作，即把它们看做16位寄存器。

操作码	指 令	操作码	指 令
03	INX BC	0B	DCX BC
13	INX DE	1B	DCX DE
23	INX HL	2B	DCX HL
33	INX SP	3B	DCX SP

下面的指令把由任意2个寄存器组成的16位寄存器对的内容加到寄存器对HL中。

操作码	指 令
09	DAD HL, BC
19	DAD HL, DE
29	DAD HL, HL
39	DAD HL, SP

这些指令可以减少操作的字节数。例如，上面的第一条指令一般情况下需要6个字节。

MOV A, L

ADD A, C

MOV L, A

MOV A, H

ADC A, B

MOV H, A

DAD指令一般用来计算存储器地址，只对进位标志位CF有影响。

- 接下来我们来认识一下各种各样的指令。下面两条指令的特点是操作码后面都跟着2字节的地址，第一条指令把HL寄存器对的内容保存到该地址，第二条指令把该地址的内容加载到HL寄存器对。

操作码	指 令	意 义
E9h	PCHL PC, HL	将 HL 保存的数据加载到程序计数器
F9h	SPHL SP, HL	将 HL 保存的数据加载到堆栈指针

PCHL指令本质上是一种Jump指令，它把HL保存的存储器地址加载到程序计数器，而8080处理器要执行的下一条指令就是程序计数器所指明的存储器地址中存放的指令。SPHL指令可以作为另一种为堆栈指针赋值的指令。

- 下面的两条指令中，第一条将HL保存的数据与堆栈顶部的两个字节进行交换；第二条指令将HL保存的数据和寄存器对DE保存的数据进行交换。

操作码	指 令	意 义
E3h	XTHL HL, [SP]	把 HL 中的内容和堆栈顶部 2 个字节进行交换
EBh	XCHG HL, DE	把 DE 中的内容和 HL 中的内容进行交换

- 除了刚讲过的PCHL指令外，目前为止还没有介绍过8080的跳转指令。如第17章所述，在处理器中专门设置了一个称为程序计数器PC的寄存器，它用来保存处理器将要取出并执行的指令的存储地址。通常，处理器在PC的指引下顺序执行存储器中存放的指令，但有一些指令，如Jump（跳转）、Branch（分支）或Goto（无条件转移）——使处理器脱离原来的执行顺序。这些指令使PC重新加载另外的值，处理器要执行的下一条指令存放于存储器的其他位置，而不在按原来的顺序寻址。
- 当然，原始的普通跳转指令的确有一定的作用，但从第17章得来的经验可以知道，条件跳转（conditional jump）指令的作用更大。条件跳转指令使处理器根据某些标志位的值转移到特定的地址，这些标志位可以是进位标志位CF、零标志位ZF等。正是由于条件跳转指令的引入，第17章所设计的自动加法器才成为一般意义上的数字计算机。
- 8080有5个标志位，其中有4个可用于条件跳转指令。8080支持9种不同的跳转指令，包括了非条件跳转指令，还包含根据ZF（Zero Flag）、CF（Carry Flag）、PF（Parity Flag）以及SF（Sign Flag）是否为1而跳转的条件跳转指令。
- 在介绍这些指令之前，首先来介绍与Jump指令相关的另外两种指令。
  - 第一种是Call（调用）指令，它和Jump指令类似，但是有所不同的是：执行Call指令后，程序计数器（Program Counter，在这部分讲解中简称PC）加载一个新的地址，而处理器会把原来的地址保存起来，保存到何处呢？最好的选择自然是堆栈了。这种策略使Call指令有效地记录了“从何处跳转”（where it jumped from），即保存了跳转之前的相关信息。堆栈中保存的地址可以使处理器最后返回到转移之前的位置。
  - 用于返回的指令称为Return（返回）。Return指令从堆栈中弹出两个字节，并把它们加载到PC中，这样就完成了返回到跳转点的工作。
  - 对于任何处理器来说，Call和Return指令都非常重要。在它们的帮助下，程序员可以在程序中使用子程序（subroutine），子程序是一段频繁使用的完成特定功能的代码（这里的“频繁”意味着“不止一次”）。对于汇编语言来说，子程序是其基本的组成部分。
- 让我们来看一个使用子程序的例子。假设你在编写一个汇编语言程序，在程序的某个位置你需要把两个字节相乘，因此你编写了一段用于两个数相乘的代码，然后继续向下写，在程序的另一个位置你发现需要再一次对两个字节相乘。因为你已经写过把两个字节相乘的代码，所以只需要重复使用这些代码就可以了。但是怎么做呢？只是简单地把这些代码重复输入到存储器吗？我们希望不是，因为这样做不仅耽误时间而且浪费存储空间，一个更好的方法是跳转到先前写的那段乘法代码所在的位置。但是普通的Jump指令不能完成这个操作，因为在执行乘法之后不能准确地返回程序的当前位置。因此你需要使用Call指令和Return指令来帮助你实现这个功能。
- 用来实现两个数相乘的一组指令可以作为一个子程序。下面我们将看到这个子程序。在第17章的乘法程序中，被乘数（还有乘积）被保存在存储器的特定位置；而在8080的子程序中，乘数和被乘数分别存放在寄存器B和寄存器C中，乘积保存到16位寄存器对HL中。8080中的乘法子程序如下：

Multiply:	PUSH	PSW	； 将要修改的寄存器的原内容保存至堆栈
	PUSH	BC	
	SUB	H, H	； 将 HL (即乘积) 置为 0000h
	SUB	L, L	
	MOV	A, B	； 乘数送至累加器 A
	CPI	A, 00h	； 如果累加器中的值是 0，则结束
	JZ	AllDone	
	MVI	B, 00h	； 将 BC 的高字节置为 0
MultLoop:	DAD	HL, BC	； 将 BC 的内容加到 HL
	DEC	A	； 乘数减 1
	JNZ	MultLoop	； 如果不为 0，则跳转
AllDone:	POP	BC	； 将堆栈中保存的数据恢复至寄存器
	POP	PSW	
	RET		； 返回

注意，上述子程序的第一行有一个标志Multiply。当然，实际上这个标志对应着子程序在存储器中的起始地址。该子程序在开始处使用了两个PUSH指令，这是因为通常在子程序的起始处要保存程序用到的寄存器。

保存寄存器后，子程序下面要做的是把寄存器H和L置0。尽管可以使用MVI（转移立即数）指令代替SUB指令来实现该操作，但这样会用到4个字节而不是2个字节的指令。子程序执行成功后，运算结果会保存到寄存器对HL中。

接下来子程序把寄存器B中的数（即乘数）转移到累加器A，并判断该数是否为0。如果为0，则乘法子程序结束，因为乘数为0。由于寄存器H和L已经为0，所以子程序可以使用JZ（Jump If Zero）指令跳转到程序最后的两条POP指令。

如果乘数不是0，子程序会把寄存器B置为0。现在寄存器对BC存放的是16位的被乘数，而累加器中存放的是乘数。接下来DAD指令会把BC（被乘数）加到HL（运算结果）中。A中的乘数减1，如果结果不为0，则执行JNZ（非零跳转）指令，该指令会使BC再次加到HL。这个循环会继续执行，直到循环的次数等于乘数为止（当然也可以利用8080的移位指令编写一个更有效率的乘法子程序）。

在程序中使用如下指令来调用这个乘法子程序，例如，把25h和12h相乘：

MVI	B, 25h
MVI	C, 12h
CALL	Multiply

CALL指令把PC的值保存到堆栈中，被保存的这个值是CALL指令的下一条指令的地址，然后CALL指令将使程序跳转到标志为Multiply的指令，即子程序的起始处。当子程序得到计算结果后，执行RET（返回）指令，该指令使保存在堆栈的PC的值弹出，并重新设置到PC，之后程序将继续执行CALL指令后面的指令。

- 8080指令集包括条件CALL指令和条件Return指令，但它们使用的频率比条件跳转指令小得多。下面的表格完整地列出了这些指令。

条件	操作码	指令	操作码	指令	操作码	指令
<b>None</b>	C9	RET	C3	JMP aaaa	CD	CALL aaaa
<b>Z not set</b>	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
<b>Z set</b>	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
<b>C not set</b>	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
<b>C set</b>	D8	RC	DA	JC aaaa	DC	CC aaaa
<b>Odd parity</b>	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
<b>Even parity</b>	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
<b>S not set</b>	F0	RP	F2	JP aaaa	F4	CP aaaa
<b>S set</b>	F8	RM	FA	JM aaaa	FC	CM aaaa

- 正如你大概所了解的，存储器并不是连接在微处理器上的唯一设备。一个完整的计算机系统通常需要输入/输出设备（I/O）以实现人机交互。输入/输出设备通常包括键盘和显示器等。
- 微处理器是如何与外围设备（peripheral，除存储器外，与微处理器连接的所有设备都可以称为外围设备）互相通信的呢？外围设备配备了与存储器类似的接口，微处理器通过与某种外围设备对应的特定地址（即接口）对其进行读写操作。在某些微处理器中，外围设备实际上占用了一些通常用来寻址存储器的地址，这种结构称作内存映像I/O（memory-mapped I/O）。但在8080中，除了常规的65536个地址外，另外增加了256个地址专门用来访问输入/输出设备，它们被称作I/O端口（I/O ports）。I/O地址信号标记为A0~A7，但I/O的访问方式与存储器的访问方式不同，两者的区分由8228系统控制芯片的锁存信号来标识。
- OUT（输出）指令把累加器中的内容写入到紧跟该指令后的字节所寻址的端口（port）。IN（输入）指令把一个字节从端口读入到累加器。它们的格式如下所示。

操作码	指令
D3	OUT PP
DB	IN PP

- 外围设备有时候需要获得处理器的注意。例如，当你按下键盘的某个键时，处理器应该马上注意到这个事件。这个过程由一个称为中断（interrupt）的机制实现，这是一个由外围设备产生的信号，连接至8080的INT输入端。

但是，当8080复位后，就不再响应中断。程序必须执行EI (Enable Interrupt) 指令来允许中断，然后执行DI (Disable Interrupts) 禁止中断。这两条指令如下所示。

操作码	指令
F3	DI
FB	EI

8080的INTE输出信号用来指明何时允许中断。当外围设备需要中断微处理器的当前工作时，它需要把8080的INT输入信号置为1。8080通过从存储器中取出指令来响应该中断，同时控制信号指明有中断发生。外围设备通常提供下列指令来响应8080微处理器。

操作码	指令	操作码	指令
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
D7	RST 2	F7	RST 6
DF	RST 3	EF	RST 7

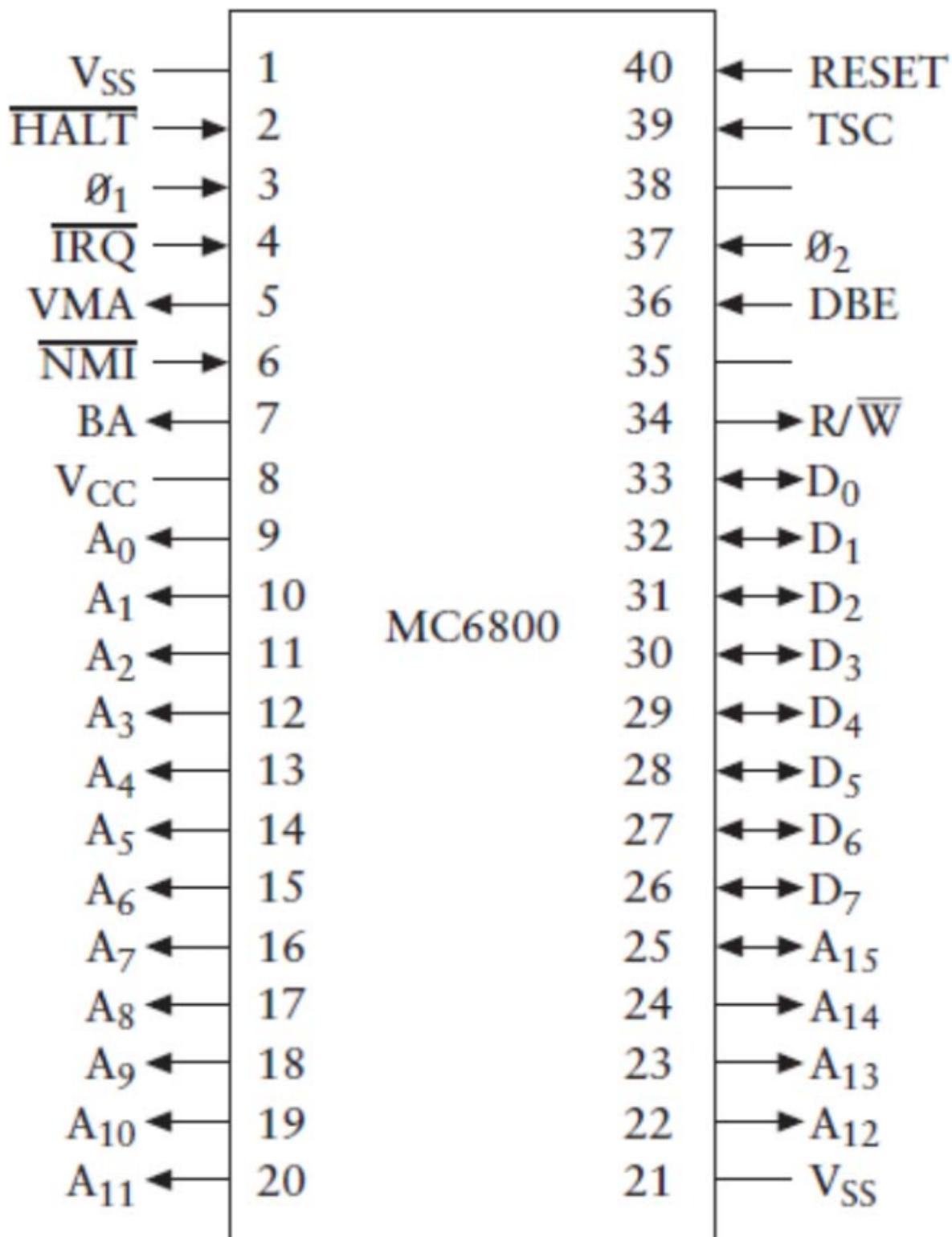
上面列出的这些指令都称作Restart (重新启动) 指令，在其执行的过程中也会把当前PC中的数据保存到堆栈，这一点与CALL指令类似。但Restart指令在保存PC数据之后会立刻跳转到特定的地址，而且是根据参数的不同将跳转到不同的地址：比如RST 0将跳转到地址0000h处，RST 1将跳转到地址00008h处，依此类推，最后的RST7将跳转到地址0038h处。这些地址存放的代码都是用来处理中断的。例如，由键盘引起的中断将执行RST 4指令，程序将跳转到地址0020h处，该地址存放的代码将负责从键盘读入数据（完整的过程将在第21章讲述）。

- 目前为止，我们已经介绍了243个操作码。在前255个数中，有12个没有作为操作码使用，它们是：08h, 10h, 18h, 20h, 28h, 30h, 38h, CBh, D9h, DDh, EDh和FDh。下面还需要讲到一个操作码。

操作码	指令
00	NOP

NOP代表（即声明）no op（no operation，无操作）。NOP指令使处理器什么操作也不执行。这样做有什么好处呢？填空，即保持处理器的运行状态而不做任何事情。8080可以执行一批NOP指令而不会引起任何错误事件的发生。

- 本章不准备详细介绍Motorola 6800微处理器，因为在构造和功能方面它与8080非常相似。下面是6800的40个管脚的功能描述图。



在上图中，V<sub>ss</sub>表示接地，V<sub>cc</sub>代表5V的电源。同8080一样，6800也有16个地址输出信号端和8个数据信号端，其中数据信号端既可以用于输入信号也可以用于输出信号。它还有一个RESET信号端和一个R/W（read/write，读/写）信号。IRQ信号代表中断请求。与8080相比，6800的时钟信号较为简单，6800没有设计独立的I/O端口，所有的输入/输出设备的地址都是存储器地址空间的一部分。

- 6800有一个16位的程序计数器PC、一个16位的堆栈指针SP、一个8位的状态寄存器（用来保存标志位），以及两个8位的累加器A、B。A和B都可以用做累加器（而不是把B作为普通的寄存器），因为A和B的功能完全相同，任何用A做的工作都可以用B实现。与8080不同，6800没有设置其他的8位寄存器。
- 6800设置了一个16位的索引寄存器（index register），它可以用来自保存16位的地址，其功能与8080的HL寄存器对相似。对于6800的大部分指令来说，它们的地址都可以由索引寄存器与紧跟在操作码后的字节相加得到。
- 尽管6800实现的操作与8080大致相同——加载、保存、加法、减法、移位、跳转、调用等，但对应的操作码和助记符是完全不同的。例如，下面列出了6800的转移（Branch）指令集。

操作码	指 令	意 义
20h	BRA	转移
22h	BHI	大于则转移
23h	BLS	相等或小于则转移
24h	BCC	进位为 0 则转移
25h	BCS	进位为 1 则转移
26h	BNE	不相等则转移
27h	BEQ	相等则转移
28h	BVC	溢出置 0 则转移
29h	BVS	溢出置 1 则转移
2Ah	BPL	为正数则转移
2Bh	BMI	为负数则转移
2Ch	BGE	大于或等于 0 则转移
2Dh	BLT	小于 0 则转移
2Eh	BGT	大于 0 则转移
2Fh	BLE	小于或等于 0 则转移

与8080不同，6800没有设置奇偶标志位，而是设置了一个溢出标志位（Overflow flag）。上面的转移指令中有一些依赖于标志位的组合（combinations of flags）。

- 当然，8080和6800的指令集是不同的，虽然这两款芯片于同一年发布，但它们是由属于不同公司的两组不同的工程师设计的。这就造成了它们之间的不兼容，因此它们不能执行对方的机器码，为一种芯片编写的汇编语言程序也不能在另一种芯片上执行。如何编写能在不同类型处理器上执行的计算机程序是第24章的主题。

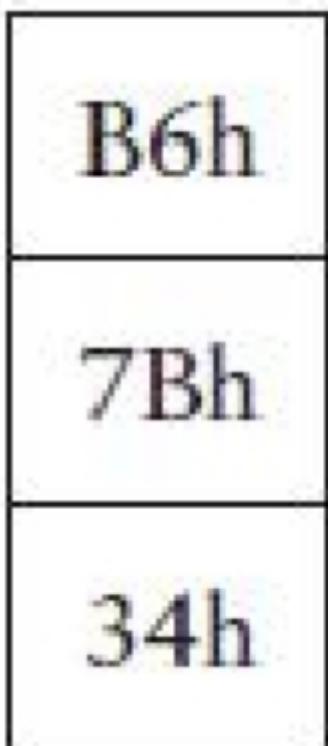
- 8080和6800的另一个有趣的区别是：在两个处理器中，LDA指令都从存储器的特定地址将数据加载到累加器。8080的操作码是3Ah，而6800的操作码是B6h。两种微处理器对紧跟在操作码后的地址的处理方式是不同的，8080假设低字节在前，高字节在后；而6800假设高字节在前，低字节在后。

例如，在8080中，下面的字节序列将把存储器地址347Bh处的字节加载到累加器。



8080 LDA 指令

现在对比一下6800的LDA指令，它使用6800扩展寻址模式（6800 extended addressing mode）。这组字节序列将把存储器7B34h地址处的字节加载到累加器。



6800 LDA 指令

- Intel和Motorola的微处理器在保存多字节数据问题上的根本区别从未得到解决。直到今天，英特尔的微处理器在保存多字节数据时，仍然把最低有效字节放在最前面（也就是说，在最低地址处），而Motorola的微处理器在保存多字节数据时，仍然把最高有效字节放在最前面。

这两种不同的方式分别称为**little-endian** (Intel方式) 和**big-endian** (Motorola方式)。争论两者之间哪一种方式更好是件有趣的事，但在这么做之前，先要知道big-endian这个术语出自乔纳森·斯威夫特 (Jonathan Swift) 的Gulliver's Travels，指的是Lilliput和Belfuscu之间关于在吃鸡蛋之前应该把鸡蛋的哪一头敲碎的争论。因此，这种争论可能是没有意义的（另一方面，坦白地说，在本书第17章设计的计算机所采用的方式我个人并不喜欢）。尽管不能确定那一种方式本质上是“对的”，这种差别确实造成了附加的兼容性问题，这种问题通常会在采用little-endian和big-endian系统的机器共享信息时出现。

- 8080被应用在一些人所谓的第一台个人电脑 (personal computer) 上，更准确地说应该是用于第一台家用电脑 (home computer) 上。下图是Altair 8800，它曾登上了1975年1月的Popular Electronics杂志的封面。

HOW TO "READ" FM TUNER SPECIFICATIONS

TECHNOLOGY

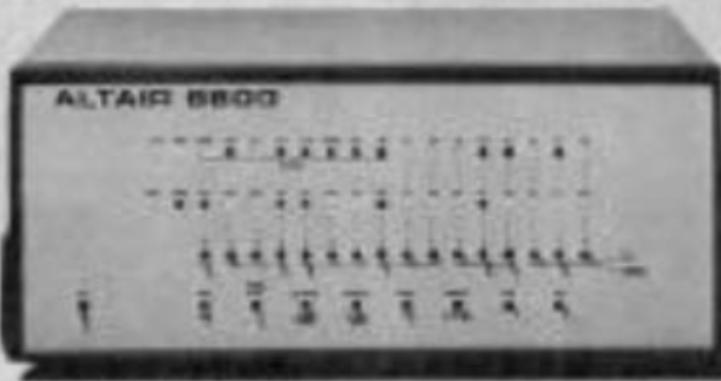
# Popular Electronics

WORLD'S LARGEST-SELLING ELECTRONICS MAGAZINE JANUARY 1978 / 75¢

## PROJECT BREAKTHROUGH!

### World's First Minicomputer Kit to Rival Commercial Models...

"ALTAIR 8800"      SAVE OVER \$1000



## ALSO IN THIS ISSUE:

- An Under-\$90 Scientific Calculator Project
- CCD's—TV Camera Tube Successor?
- Thyristor-Controlled Photoflashers



## TEST REPORTS:

- Technics 200 Speaker System
- Pioneer RT-1011 Open-Reel Recorder
- Tram Diamond-40 CB AM Transceiver
- Edmund Scientific "Kirlian" Photo Kit
- Hewlett-Packard 5381 Frequency Counter

当你看到Altair 8800时，前面板上的灯泡和开关会让你感到似曾相识。这个界面和第16章介绍64 KB RAM阵列时的初始“控制面板”的界面是类似的。

- 在8080之后，Intel又推出了8085芯片，而具有更重大意义的是Z-80芯片的出现。Z-80是由Zilog公司制造的，该公司是英特尔公司的竞争对手，由英特尔的前雇员费德瑞克·菲戈金（Federico Faggin）创立，费德瑞克·菲戈金曾在4004芯片的研制过程中做出重要贡献。Z-80和8080完全兼容，并且增加了许多非常有用的指令。1977年，Z-80曾被应用在Radio Shack TRS-80 Model 1上。

- 同样是在1977年，由斯蒂夫·乔布斯（Steven Jobs）和史蒂芬·沃兹内卡（Stephen Wozniak）创立的苹果计算机公司推出了新一代产品Apple II。Apple II既没有使用8080也没有使用6800，而是使用了基于MOS技术的更加便宜的6502芯片，它是6800的改进加强版本。
- 1978年6月，英特尔公司推出了8086芯片，这是一个16位的微处理器，可以寻址1MB的地址空间。8086的操作码与8080不兼容，但它包含了乘法指令和除法指令。一年后，英特尔推出了8088芯片，其内部结构与8086完全相同，但在外部仍以字节为单位（即外部接口为8位）访问存储器，所以该芯片能使用为8080设计的较为流行的8位的外围芯片（8-bit support chips）。IBM在5150个人计算机中使用了8088芯片，这种计算机通常称为IBM PC，于1981年秋季推出。
- IBM大举进军个人计算机（Personal Computer，有时也简称为PC）市场对业界产生了重大影响，许多公司都推出了与个人计算机兼容的机器（兼容的含义将在随后的几章里详细讨论）。多年以来，“IBM PC兼容”也暗示了“Intel inside”（即内部使用了Intel微处理器），这里特指Intel x86系列微处理器。x86系列微处理器包括1982年发布的186芯片和286芯片，1985年发布的32位386芯片，1989年发布的486芯片。从1993年开始，英特尔公司推出Intel奔腾（Intel Pentium）系列微处理器，而这个系列如今被广泛地应用于PC兼容机。虽然这些处理器的指令集都在不断扩展，但是它们仍然支持始于8086的所有早期处理器的操作码。
- 苹果公司的Macintosh于1984年首次发布，它采用摩托罗拉的68000微处理器，68000是16位微处理器，是6800的下一代产品。68000及其后续产品（通常称为68K系列）是已发布的处理器中最受欢迎的一类。
- 从1994年开始，Macintosh计算机开始使用PowerPC微处理器，该处理器是由摩托罗拉，IBM以及苹果公司联合开发的。PowerPC是采用RISC（Reduced Instruction Set Computing，精简指令集计算机）微处理器体系结构来设计的，其目的是通过某些方面的简化来提高处理器的速度。在RISC计算机中，通常指令都是等长的（PowerPC中是32位），只有加载和保存两种指令能访问存储器，并且尽量简化指令的操作。RISC处理器设置了大量的寄存器，这样就能避免频繁访问存储器以提高运行速度。
- PowerPC拥有完全不同的指令集，因此不能执行68K系列微处理器的代码。然而，目前Macintosh计算机使用的PowerPC微处理器可以仿真（emulate）68K系列微处理器。运行在PowerPC上的仿真程序逐一检查68K程序的操作码，并执行相应的操作。它执行的速度没有PowerPC本身的代码那么快，但可以正常工作。
- 根据摩尔定律（Moore's Law），微处理器中的晶体管数量每18个月翻一倍，人们不禁要问：增加的这些大量的晶体管用来做什么呢？一些晶体管用来适应处理器不断增加的数据宽度——从4位、8位、16位到32位；另一些新增的晶体管用来应对新的指令。例如，现在大部分微处理器都支持用于浮点数的指令（将在第23章详细介绍）；还有一些新增的指令用来执行重复计算，以便在计算机屏幕上呈现图片和电影。
- 现代处理器使用多种技术来提高其运行速度。其中一种就是流水线技术（pipelining），即处理器在执行一条指令的同时读取下一条指令，尽管Jump指令在一定程度上会改变这种流程。现代处理器还包括一个Cache（高速缓冲存储器），它是一个设置在处理器内部，访问速度非常快的RAM阵列，用来存放处理器最近要执行的指令。由于计算机程序经常执行一些小的指令循环，使用Cache可以避免反复加载这些指令。上面提到的这些提高运行速度的策略都需要在处理器内部增加更多的逻辑组件和晶体管。
- 正如前面所提到的，微处理器只是整个计算机系统的一部分（尽管是最重要的一部分）。我们会在第21章构造这样一个系统，但首先要学习如何处理存储器中的数据，包括操作码和数字，我们要对这些数据进行编码。

## 20 ASCII码和字符转换

---

- 数字计算机中的存储器唯一可以存储的是比特，因此如果要想在计算机上处理信息，就必须把它们按位存储。通过先前的学习，我们已经掌握了如何用比特来表示数字和机器码。现在我们面临的一大挑战就是如何用它来存储文本。毕竟，人类所积累的大部分信息，都是以各种文本形式保存的。文本信息聚集最多的地方之一就是图书馆，数不清的书、杂志和报纸所提供的都是文本信息。当然，我们现在已经开始使用计算机来存放图像和影音信息了，不过为了易于理解，我们还是先从如何使用计算机存放文本开始讲解。
- 为了将文本表示为数字形式，我们需要构建一种系统来为每一个字母赋予一个唯一的编码。数字和标点符号也算做文本的一种形式，所以它们也必须拥有自己的编码。简而言之，所有由符号所表示的字母和数字（Alphanumeric）都需要编码。具有这种功能的系统被称为字符编码集（Coded Character Set），系统内的每个独立编码称为字符编码（Character Codes）。
- 许多疑问也随之而来，而要解决的第一个问题是：构成这些编码究竟需要多少比特？要想回答这个问题就需要我们从长计议了。
- 文本与其印刷在纸上时采用的排版格式是两码事。充分发挥想象力，将文本看成是一维的由字母、数字和标点符号组成的数据流吧。当然，有时为了标明一句话的开始和结尾，还需要一些额外的编码。
- 在先前对莫尔斯码和布莱叶盲文的学习中，我们了解了如何将字母表中的字符以二进制的形式表现出来。这些系统在适合的场合很好用，但要想用到计算机中却是难上加难。就拿莫尔斯码来说，它是变量自适应长度（Variable-Width）编码：常用字符的编码较短，而不常用字符的编码较长。这样的编码非常适合电报系统，但并不适用于计算机。另外，莫尔斯码并不区分字母的大小写。
- 布莱叶盲文编码使用固定宽度，非常适合计算机使用。每一个字符对应着6比特的编码，并且用到了转义（Escape）码对大小写进行了区分。转义码用来表明下一个字符为大写。这也就是说，每个大写字母都需要两组编码来表示。布莱叶盲文中用移位（Shift）码表示数字：移位码后紧跟的编码都被看做数字，直到遇到下一个移位码，此时系统又将后面的内容当做字母。
- 我们的目标是开发一个字符编码集，使用这个编码集，系统可以将如下的句子转换成为一系列的编码：  
I have 27 sisters.
  - 每一个字符的编码都会占据一定的比特。有的编码用来表示字母，有的用来表示标点符号，还有一些用来表示数字。甚至于单词间的空格也需要单独的编码。对一个句子进行编码后得到的连续字符串通常被称为文本字符串（string）。
- 我们需要对字符串中的数字进行编码，例如上面的句子中的27。或许大家会感到疑惑，因为之前我们都是用比特来表示数字的。最简单的，也是最容易想到的做法就是使用二进制数10和111作为2和7的编码。但是这里却不适用。在这个句子中，可以像处理其他的字符一样来处理2和7。它们的编码可以和本身表示的含义无关。
- 1874年由法国电报服务公司（French Telegraph Service）职员埃米尔·波多（Emile Baudot）发明了可以打印的电报机，划时代的波多电传码也应运而生。即使在今天来看，这种编码十分“经济划算”，每一个文本字符都采用5位编码。这种编码1877年被法国电报服务公司采纳，后来经唐纳德·默里（Donald Murray）修改，最终在1931年被当年的CCITT组织（Comité Consultatif International Télégraphique et Téléphonique），即现在的国际电信联盟（ITU）定为标准。该编码的正式名称是国际电报字母表第二号（International Telegraph Alphabet No.2）或ITA - 2，在美国常常被称为波多印字电报制（Baudot），不过更准确地说，叫做默里（Murray）编码。
  - 随着20世纪的到来，Baudot被广泛应用于电传打字机（teletypewriters）。Baudot电传打字机配备了一个输入键盘，这款键盘有些像打字机，但只有30个键和一个空格键。电传打字机键盘上的每一个键实质上都起到了转换器的作用，它负责产生二进制编码并且通过输出电缆逐位传输出去。电传打字机也具备打印功能，通过输入电缆读取编码，触发电磁铁，从而将字符打印在纸上。

- 由于Baudot对每个字符采用5位编码，整个系统由32个编码所组成，这些编码的十六进制取值范围从00h到1Fh。下表给出了32个不同编码的十六进制形式及其所对应的字母表中的字符。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	E
01	T	11	Z
02	回车	12	D
03	O	13	B
04	空格	14	S
05	H	15	Y
06	N	16	F
07	M	17	X
08	换行	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	数字转义符号
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K
0F	V	1F	字符转义符号

编码00h被保留了下来，没有指派给任何值。剩下的31个编码中，字母表中的字符占了26个，其余5个用来调整格式，如上表中的语句所示。编码04h用来表示空格，通常用于分隔单词。

编码02h和08h表示的是回车和换行。这些都是电传打字机中的专用术语。

当使用电传打字机上打字，一旦到了一行的末尾时，我们通常会按下一个操作杆或按钮。这个操作其实包括两个动作：第一个动作是，使打印机的滑架回到起始位置，这样打印下一行时可以从纸的最左边开始，这就是**回车**。第二个动作是，将打印机的滑架移至正在使用中的位置的下一行，这就是**换行**。在Baudot编码系统中，这两个编码由专门的按键产生。Baudot电传打字机在打印的时候会响应这两个编码以完成相应的操作。

编码1Bh中暗藏玄机，它的实际作用是数字转义（Figure Shift）。数字转义编码后的所有的编码都会被解释为数字或标点符号，直到遇到字符转义编码（1Fh），一切就又被解释为字符。下表展示了十六进制编码以及所对应的数字和标点符号。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	3
01	5	11	+
02	回车	12	身份不明
03	9	13	?
04	空格	14	'
05	#	15	6
06	,	16	\$
07	.	17	/
08	换行	18	-
09	)	19	2
0A	4	1A	响铃
0B	&	1B	数字转义符号
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	字符转义符号

像莫尔斯码一样，这种5位的编码并没有提供区分大、小写的方法。

请注意三个转义码的使用：1Bh出现在数字之前，1Fh出现在数字之后，而标点之前又出现了1Bh。这一行编码以回车、换行符结尾。问题出来了，如果把相同的数据流再一次输入到电传打印机，情况就大不一样了，如下所示：

I SPENNT \$25 TODAY.

8'03,5 \$25 TODAY.

若在接收到第二行编码之前，打印机接收到的最后一个转义码是数字转义码，遇见第二行开头几个编码时，打印机将它们解释成数字。

- 这种问题产生的根源就是采用了转义码，这的确很让人头痛。尽管Baudot电传码是很简洁实用的编码，但是，我们更加希望采用能唯一表示字符、数字及标点符号的编码方案，如果还能对大、小写进行区分那就更好不过了。
- 如果想知道比Baudot更好用的编码系统中一个编码需要多少比特，我们需要做几个小加法：所有的大小写字母加起来共需52个编码，0~9数字需要10个编码，加起来共有62个，如果算上一些标点符号，数量超过了64个，也就是说，一个编码至少需要6比特。但无论如何字符数应该不超过128个，而且应该远远不够128个，也就是说编码长度不会超过8位。所以，答案就是7。在采用7位编码时，不需要转义字符，而且可以区分字母的大小写。
- 所有人都遵循并使用统一化的编码，计算机的存在才有意义。这样一来，使用不同方法制造出的计算机之间就可以互相兼容，甚至可以互相交流文本信息。这种标准已经存在并且被广泛使用，它被称为美国信息交换标准码（American Standard Code for Information Interchange），简称为ASCII码。从1967年正式公布至今，它一直是计算机产业中最重要的标准。不过还有一个大的例外（后面会讲到），无论何时，当你在计算机上处理文本时，总会在不经意间使用到ASCII码。
- ASCII码是7位编码，它的二进制取值范围为0000000~1111111，对应于十六进制就是00h~7Fh。

- 现在我们一起来讨论下ASCII码，但我不建议从开始学起，因为相对于后面的编码，前32个编码理解起来还有一点难度。所以我们从第2组32个编码开始学习，它包括标点符号和10个数字。下表列出了这32个字符及相应的十六进制编码。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
20	空格	30	0
21	!	31	1
22	“	32	2
23	#	33	3
24	\$	34	4
25	%	35	5
26	&	36	6
27	‘	37	7
28	(	38	8
29	)	39	9
2A	*	3A	:
2B	+	3B	;
2C	,	3C	<
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

值得注意的是20h代表空格符，它的作用是将单词或句子隔开。

接下来的32个编码是大写字母和一些附加的标点符号的编码。除了@符号和下划线之外，其余的符号很难在打字机上找到。它们真正出现的地方是标准计算机键盘，下表列出了这些字符及相应的十六进制编码。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
40	@	50	P
41	A	51	Q

42	B	52	R
43	C	53	S
44	D	54	T
45	E	55	U
46	F	56	V
47	G	57	W
48	H	58	X
49	I	59	Y
4A	J	5A	Z
4B	K	5B	[
4C	L	5C	\
4D	M	5D	]
4E	N	5E	^
4F	O	5F	-

再接下来的32个编码是所有小写字母和一些附加的标点符号及其对应的十六进制编码，这些字符也很少在打字机上出现。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
60	`	70	p
61	a	71	q
62	b	72	r
63	c	73	s
64	d	74	t
65	e	75	u
66	f	76	v
67	g	77	w
68	h	78	x
69	i	79	y
6A	j	7A	z
6B	k	7B	{
6C	l	7C	
6D	m	7D	}
6E	n	7E	~
6F	o		

注意，表的最后不包括7Fh及其对应的字符。如果你统计一下，就会发现这三张表共涵盖了95个字符。由于ASCII码的编码长度为7位，所以最多可以表示128个编码，这样算下来还剩33个编码可用。

- 在ASCII码中，一个大写字母与其对应的小写字母的ASCII码值相差20h。这种规律大大简化了程序代码的编写，例如一段将特定的字符串变成大写的程序。假设有一个字符串存放在内存的某个区域，每个字符占据一个字节。下面是一段8080子程序，初始状态下字符串的首地址存放在寄存器HL中；寄存器C存放字符串的长度，也就是字符的个数。

```

Capitalize: MOV A, C          ; c 表示剩余的字符数
             CPI A, 00h      ; 与 0 进行比较
             JZ AllDone      ; 如果剩余的字符数为 0, 程序结束
             MOV A, [HL]       ; 取得下一个字符
             CPI A, 61h       ; 判断 A 代表的字符的 ASCII 码是否比'a'小
             JC SkipIt        ; 如果比'a'小, 就跳过
             CPI A, 78h       ; 判断是否比'z'大
             JNC SkipIt       ; 如果是, 则跳过
             SBI A, 20h       ; 判断是否是小写, 如果是, 则减 20h
             MOV [HL], A       ; 保存修改过的字符
SkipIt:     INX HL           ; 指向下一个字符
             DCR C            ; 计数器减一
             JMP Capitalize   ; 返回到程序起始处
AllDone:    RET

```

还有另外一种方法也可以将小写字母减去20h而转换成大写字母，如下所示：

**ANI A, DFh**

ANI指令 (AND Immediate) 用来“与”一个立即数。在上面这个例子中，累加器中的数值与DFh执行“按位与”操作，其中DFh转换成二进制数就是11011111。“按位与”操作就是把两个数分别转换成二进制，然后将对应的位进行“与”操作。这个例子中，除了自左向右数的第3位被置成0外，A中的其他位均被保留。通过将这一位设置为0，我们实现了将小写字母的ASCII码转换成大写字母的目的。

- 前面讲到的95个编码也被称为图形文字 (graphic characters)，因为它们可以被显示出来。其实ASCII码还包含33个控制字符 (control characters)，它们用来执行某一特定功能，因而不用显示出来。为了完整地讨论ASCII编码，下面将这33个控制字符也列举了出来，有一些的确很难理解，不过不用在意。其实在ASCII码公布以后，当时人们更多的是想把它用在电传打字机上，所以，如今其中的许多编码已经渐渐离开了人们的视线。

十六进制编码	缩 写	控制字符的含义
00	NUL	空字符
01	SOH	标题开始
02	STX	文本开始
03	ETX	文本结束
04	EOT	传输中止
05	ENQ	询问
06	ACK	应答
07	BEL	响铃
08	BS	回退
09	HT	水平制表
0A	LF	换行
0B	VT	垂直制表
0C	FF	换页
0D	CR	回车
0E	SO	移出
0F	SI	移入
10	DLE	转义
11	DC1	设备控制 1
12	DC2	设备控制 2
13	DC3	设备控制 3
14	DC4	设备控制 4
15	NAK	否定应答
16	SYN	同步

17	ETB	块传输结束
18	CAN	取消
19	EM	媒介取消
1A	SUB	替代字符
1B	ESC	跳出
1C	FS	文件分割或信息分割 4
1D	GS	组分割或信息分割 3
1E	RS	记录分割或信息分割 2
1F	US	单元分割或信息分割 1
7F	DEL	删除

编码09代表水平制表符，简写为Tab。假设打印的过程中，所有水平排列字符的起始位置都为0，Tab的作用是在下一个水平位置即在距前一个字符的间距为字符长度8倍的位置打印下一个字符，这种简单有效方法使得字符可以保持按列对齐。

计算机中的回车和换行与Baudot码中表示的意思相同，它们可以算得上是控制符中最重要的两个符号。在打印机中，回车符使得打印头换行并转移至当前页面的最左端，换行符使打印头转移至当前位置下一行。这两种操作都使得打印头移至新的一行。回车符通常用来另起一行继续打印，换行符通常在不需要移到页面最左端而换行时使用。

- 尽管ASCII码在计算机领域可以说是一统江湖，但许多IBM大型机上却没有采用这种标准。例如，System/360产品内部采用的是IBM自发研制的8位字符编码系统，也被称为扩展的BCD交换码(Extended BCD Interchange Code)，或EBCDIC (英文中的发音为EBB-see-dick)。
  - 第0~9行称做数字行 (digit rows) 或数字穿孔 (digit punches)，第11和12行被称做区域行 (zone rows) 或区域穿孔 (zone punches)。
  - 一个EBCDIC字符码由8位比特组成，进一步可以细分为高半字节 (4比特) 与低半字节。低半字节是BCD码，与字符的数字穿孔保持一致，高半字节与区域穿孔的编码保持一致 (而且与区域穿孔一一对应)。
    - 数字0~9并不需要区域穿孔进行额外表示，它们的EBCDIC编码的高半字节是1111，代表了区域穿孔不起作用，而0~9的EBCDIC编码的低半字节是数字穿孔的BCD码。
    - 大写字母有一些有趣的规律，如果区域穿孔只出现在第12行，则高半字节标识为1100；如果只出现在第11行，则高半字节标识为1101；如果出现在第0行，则高半字节标识1110。
    - 小写与大写字母的数字穿孔是相同的，但它们的区域穿孔不同。在a~i的小写字母，穿孔位于第12行和第0行，高半字节对应的编码为1000；在j~r的小写字母，穿孔位于第12行和第11行，高半字节对应的编码为1001；在s~z的小写字母，穿孔位于第11行和第0行，高半字节对应的编码为1010。
    - 当然，标点符号和控制字符也都有自己的EBCDIC编码，但没有必要去深究。
  - 仔细观察IBM打孔卡，其中每一列细细数下共有12个孔，每个孔代表1位，也就是说可以提供12位的编码信息。我们其实可以用打孔卡上每一列12孔中的7个来表示ASCII码。但是，这种方案有一个非技术方面的缺陷，那就是太多的穿孔将使得卡片变得很脆弱，容易折断。

- 采用8位编码的EBCDIC中其实还有很多编码未定义，这也说明当年ASCII码采用了7位编码也是合乎情理的。在ASCII码刚刚问世的那个年代，存储器的价格贵得令人咋舌，有一些观点认为ASCII码可以用6位编码并配合转义字符来使用，这样既可以区分大小写又节约了存储器。这种方案并没有被采纳，当时还有一些人认为ASCII码应采用8位编码，他们对计算机的体系结构有了一个大胆的推测，即计算机应该按字节存储，7位存储是不合适的。今天来看，8位的字节存储已经作为了一项标准。尽管ASCII码从技术的本质上来看是7位编码，但仍以8位的形式存储。
- 在字节与字符之间建立一种等价关系大大简化了我们的工作，举例来讲，如果要粗略估计一个文本文件所需要的存储空间，只要统计字符数就可以了。这时前面学过的K (kilos) 和M (Megas) 就派上了用场，用它们来表示文本所占据的计算机存储空间更加通俗易懂。
  - 32,550个字符，约等于32 KB。
  - 美国国会图书馆 (The United States Library of Congress) 藏书约为2000万本，大概有20万亿字符，从存储器角度来说，数据总量为20 TB (这还不包括图书馆中的大量珍贵照片和录音资料)。
- 尽管ASCII码是计算机领域最重要的标准，但它并不是十全十美的。它的问题就蕴含在它的全称中——American Standard Code for Information Interchange，它太美国化了！即使那些以英语为主要语言的国家，ASCII码也并不适用。ASCII码中包含美元符号，而英镑符号怎么找不到呢？还有西欧国家语言中用到的重音符号在哪里？更别说使用非拉丁字母的希腊文 (Greek)、阿拉伯文 (Arabic)、希伯来文 (Hebrew) 和西里尔文 (Cyrillic) 等欧洲国家了。此外，印度及东南亚地区用到的婆罗门手记、北印度的Devanagari方言、孟加拉语、泰语、西藏语也并没有在ASCII码中出现。简单的7位编码在面对数以万计的中国、日本、韩国的象形文字，以及奇怪的朝鲜文音节时也显得力不从心。
- 大多数计算机系统采用8位编码来存储字符，我们也自然地想到设计一种扩展的ASCII字符集，这样可以包含256个字符，比原先扩展了一倍。在这种字符集中，编码00h ~ 7Fh与原ASCII码保持一致；编码80h ~ FFh可以用来引入其他字符。这项技术已经被用来定义附加的字符编码，比如前面提到过的重音字母以及非拉丁字母。
  - 编码A0h对应的字符为不中断空格 (No-Break Space)。通常计算机在对文本进行排版时，会将其划分为行和段，行与行之间以空格符号区分（空格所对应的ASCII码为20h）。编码A0h显示为空格，但是并不表示行与行之间被断开。比如在“WWII”这样一段文字中就可以使用不中断空格。
  - 编码ADh被定义为软连字符 (soft hyphen)，它的用途是连接同一单词之间的音节，在一个单词被不得已划分在两行时就会用到它。
- 只可惜问题也随之而来，近几十年来出现了许多不同版本的扩展的ASCII码，多个不同的版本严重影响了编码的一致性，导致了混淆和不兼容。ASCII码被扩展到极致，有的甚至可以对中文、日文和韩文进行编码。在亚洲地区，双字节字符编码系统 (double-byte character sets, DBCS) 很流行。双字节字符集有很多版本，但兼容性并不是它最主要的问题。它的另一个缺陷是，一些字符，特别是通用的ASCII码字符，是用单个字节编码表示的，相比而言，成千上万的象形文字则是双字节编码，这在无形之中增加了使用这种字符集的难度。
- 业界一直有一个目标，那就是建立一个独一无二的字符编码系统，它可以用于世界上所有语言文字，从1988年开始，几大著名计算机公司合作研究出一种用来替代ASCII码的编码系统，取名为Unicode (统一化字符编码标准)。相对于ASCII的7位编码，Unicode采用了16位编码，每一个字符需要2个字节。也就是说Unicode的字符编码范围为0000h ~ FFFFh，总共可以表示65,536个不同字符。全世界所有的人类语言，尤其是经常出现在计算机通信过程中的语言，都可以使用同一个编码系统，而且这种系统还具备很高的扩展性。
  - Unicode编码其实并不是从零开始设计的，前128个字符编码——即0000h ~ 007Fh——与ASCII码是一致的。Unicode编码中的00A0h ~ 00FFh与先前讲到的第1号拉丁字母表是一致的。

- 全世界很多标准也被一同收录在Unicode中。尽管相对于之前讲过的一些字符编码系统，可以说Unicode做出了有效地改进，但这也不能确保它被全世界广泛采纳。ASCII码，包括数不清的有一点小缺陷的扩展ASCII码已经在计算机领域根深蒂固，想一下子就取代它们并不是轻而易举的。
- 对于Unicode来讲，它唯一的问题，就是它改变了字符与存储空间之间“单字符，单字节”的等价对应关系。采用ASCII编码方式存储的著作《怒火之花》，其所占据的存储空间约为1 MB。而如果采用Unicode编码，约占2 MB。为了使编码系统兼容，Unicode在存储空间上付出了相应的代价。

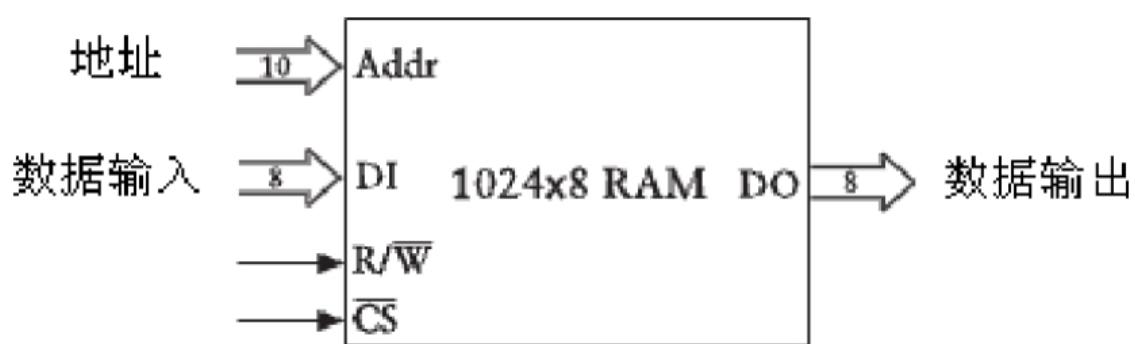
## 21 总线

---

- 在一台计算机中，中央处理器无疑是最重要的部件，但它并不是唯一的部件。随机访问存储器（Random Access Memory, RAM）也是计算机不可或缺的部件，它存放着处理器要执行的机器代码指令。通过怎样的方法才能把指令加载到RAM中？怎样才能把程序的结果变得可见呢？或许你一下子就想到了输入设备（Input Device）和输出设备（Output Device）。回想一下前面讲过的内容，RAM是易失性存储器——换言之，当掉电的时候其中的内容就会丢失。所以，长期存储设备也是一台计算机必不可少的部件，只有这样，代码和数据才能够被永久保存，不会因为掉电而丢失重要的数据。
- 搭建一台完整的计算机还需要很多集成电路，这些集成电路都必须挂载（mounted）到电路板上。在一些小型的机器中，一块电路板足以容纳所有的集成电路，但这种情况并不常见。我们通常所看到的是另一种情况：计算机中各部件按照功能被分别安装在两个或更多的电路板上。这些电路板之间通过总线（bus）通信。如果对总线做一个简单的概括，可以认为总线就是数字信号的集合，而这些信号被提供给计算机上的每块电路板。通常把这些信号划分为如下四类。
  - 地址信号。这些信号是由微处理器产生，通常用来对RAM进行寻址操作，当然也可以用来对连接到计算机的其他设备进行寻址操作。
  - 数据输出信号。这些信号也是由微处理器产生的，用来把数据写入到RAM或其他设备。这里特别要注意区分术语输入（input）和输出（output），来自微处理器的数据输出信号会变成RAM和其他设备的数据输入信号。
  - 数据输入信号。这些信号是由计算机的其他部分提供的，并由微处理器读取。通常情况下，数据输入信号由RAM输出，这就解释了微处理器是怎样从内存中读取内容的。其实，其他部件也可以给微处理器提供数据输入信号。
  - 控制信号。这些信号是多种多样的，通常与计算机内所用的特定的微处理器相对应。控制信号可以产生于微处理器，也可以由与微处理器通信的其他设备产生。比如，当微处理器要把一些数据写入到特定内存单元时，它所使用的信号就是控制信号。
  - 还有一点需要说明：总线还可以为计算机上不同电路板供电。
- 回顾一下总线的发展历程。
  - 在家用计算机领域，早期比较流行的就是S-100总线，1975年第一台家用计算机MITS Altair就率先采用了这种总线。尽管一开始，S-100总线只是基于8080微处理器的，后来经过改进，也开始适用于其他处理器，例如6800。一块S-100电路板的规格是5.3×10英寸，其中有一边是要插到一个插槽上的，这个插槽有100个连接器（这就是名为S-100的原因）。
  - 每台S-100计算机都有一块很大的被称为母板（motherboard或mainboard）的电路板，它有若干相互连接的S-100总线插槽（可能有12个）。有时候，这些插槽也被称为扩展插槽（expansion slots）。S-100电路板（也称为扩展板，expansion boards）就插在这些插槽中。8080微处理器及支持芯片（第19章提到过的其中的一些）分布在一块S-100电路板上，而RAM分布在一块或多块其他电路板上。

- S-100总线是专门为8080芯片而设计的，有16个地址信号，8个数据输入信号及8个数据输出信号（仔细回忆一下，8080本身并不区分数据输入和输出信号，这项工作是由电路板上的其他支持芯片完成的）。
- 总线上也含有8个中断信号，其他设备需要CPU立即做出响应时，便会产生这些信号。下面我们看一个例子（本章的后面也要讲到），当某个按键按下时，键盘可能就会产生一个中断信号。接下来8080会执行一段小程序，检测出是什么按键被按下，并做出响应。通常，在安装了8080的电路板上有一个被称为Intel 8214优先级中断控制单元的芯片，就是专门用来处理中断的。当中断发生时，这个芯片会产生一个中断信号并送给8080。8080识别出这个中断后，此芯片就会提供一个RST (Restart, 重启) 指令，在这条指令的作用下，微处理器会把当前程序计数器的值保存下来，并依据中断类型，跳转到地址0000h、0008h、0010h、0018h、0020h、0028h、0030h或0038h处执行。
- 如果你在设计一个新的计算机系统，而这个系统中采用新的总线类型，你可以选择把总线规范公布于众（也可以通过其他方式发布出去）或者使其保密，决定权在于你。
- 一旦一条指定总线的规范公布开来，其他制造商——称为第三方 (third-party) 制造商——就可以设计并销售采用了这种总线的扩展板了。这些额外扩展板不仅加强了计算机的实用性，还使其更加满足实际需求。计算机的销售情况越好，扩展板的市场前景也就越好。正是由于这个原因，设计者在设计多数小型计算机系统时，都会坚持开放体系结构 (Open Architecture) 的原则，这样一来，其他制造商就可以生产计算机的外设。最终会有一条总线成为工业标准。在今天，“标准”已经成为个人计算机产业的一个重要组成部分。
- 1981年秋，最著名的开放体系结构个人计算机——IBM的PC问世。IBM公布了PC的技术参考资料 (technical reference)，里面包含了整台计算机的全部电路图，IBM为其制造的扩展板的资料也在其中。这个手册可是很重要的资料，它的出现使得很多制造商可以生产自己的PC扩展板，实际上，这创造出了整个PC的克隆体——其实与IBM的PC几乎完全相同，运行的软件也一样。
- 在如今的桌面计算机领域，从起初的IBM的PC发展而来的计算机数量庞大，占据约90%的市场份额。尽管IBM本身只占很小一部分，但事实上，如果起初的PC采用的是封闭体系结构 (closed architecture) 且设计是私有化 (proprietary) 的，其所占的市场份额会更少。苹果公司的麦金托什 (Macintosh) 起初采用的是封闭体系结构，尽管也曾部分考虑过开放的问题，这也就解释了为什么Macintosh在如今的桌面计算机市场上只占有不到10%的份额。（请记住这一点：一个计算机系统可以是在开放体系结构下设计的，也可以是在封闭体系结构下设计的，无论是哪种情况，其他公司都可以为其设计软件。但也有例外的情况，某些视频游戏的开发商会限制其他公司开发其专用系统上的软件）。
- 最初的IBM PC采用的是Intel 8088微处理器，可以寻址1 MB的存储单元。虽然8088内部是一个16位的微处理器，但外部却只能寻址8位的存储器。工业标准体系结构 (Industry Standard Architecture, ISA) 总线，是IBM为最初的PC设计的。扩展板上有62针的连接插头。有20个地址信号，8个复用的数据输入/输出信号，6个中断请求信号及3个**直接存储器访问 (Direct Memory Access, DMA)** 请求信号。DMA可以使存储设备（本章最后我们会讲到）快速地执行存储操作，这比采用其他方法快得多。通常情况下，所有读/写内存的操作都是由微处理器来完成的，但采用了DMA后，其他设备可以不通过微处理器而获得总线的控制权，进而直接对内存进行读写。
- 在S-100系统中，所有的部件都安放在扩展板上。就拿IBM PC来说，微处理器、支持芯片及一些RAM都安装在一块系统板上，系统板 (system board) 是IBM的“内部称呼”，但它常常也被称为母板或主板。

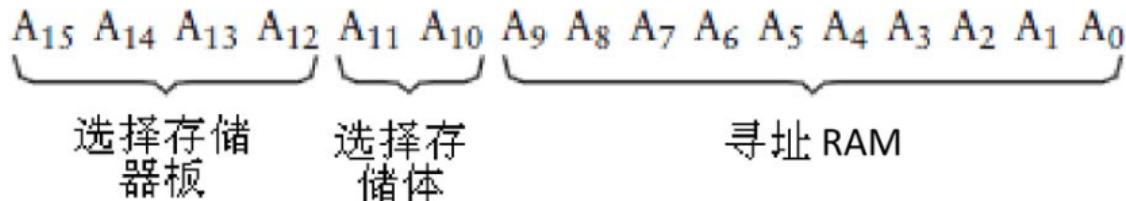
- 微处理器所使用的数据宽度（从8位到16位再到32位）和输出的地址信号的数目在不断增长，当这些超出总线的承受能力时，总线就需要升级换代了。如果微处理器的处理速度很快时，也会出现这种情况。早期的总线是为当时的微处理器而设计的，它们的时钟频率一般是几兆赫兹而不是几百兆赫兹。如果设计出来的总线不适合高速传输的话，就会出现射频干扰（RFI），这会使附近的收音机和电视机产生静电或其他噪声。
- 1987年，IBM推出了微通道体系结构（Micro Channel Architecture，MCA）总线。这种总线的某些部分已经成为IBM的专利，如果其他公司使用这种总线，IBM就会从中收取授权费用。也许就正是由于这个原因，MCA才没能成为一种工业标准。然而就在1988年，9家公司（并不包括IBM）联合推出的32位EISA（Extended Industry Standard Architecture）总线取代了MAC，成为了工业标准。近几年，Intel公司设计的外围部件互连（PCI）总线已普遍使用在PC兼容机上。
- 计算机上的各种不同的部件是如何工作的呢？为了能更好地理解，让我们再次回到20世纪70年代中期去看一看。想象一下，我们正在为Altair设计电路板，或者是在为自己设计的8080或6800计算机做这样的事情。我们不仅要考虑为计算机设计一些存储器，用键盘作为输入，用电视机作为输出；还要考虑关于计算机时，如何把存储器中的内容保存下来。如何把这些部件添加到计算机中呢？下面就来看看能实现这个功能的各种接口（Interface）。
  - RAM阵列有地址输入、数据输入，以及数据输出信号，另外还有一个用来把数据写入存储器的控制信号。RAM阵列能存放的数据的数量是和地址输入信号的个数有关的，它们之间有着如下的关系：RAM阵列中数字的个数=  $2^{\text{地址信号的个数}}$ 。数据输入、输出信号决定着所存储的数值的大小（位数）。
- 20世纪70年代中期，2102是用于家用计算机的一款流行的存储器芯片。它也是MOS（metal-oxide semiconductor）家族中的一员，与8080和6800微处理器所采用的技术相同。MOS半导体管很容易与TTL芯片连接起来；通常情况下，其内部晶体管的密度要比TTL高，但速度却不如TTL快。这个芯片存储容量可以达到1024位，这个数值可以根据地址信号（A<sub>0</sub> ~ A<sub>9</sub>）、数据输出（DO）和数据输入（DI）信号（输入和输出复用一个信号线）的数目计算出来。你所使用2102芯片型号不同，访问时间（read access time，指从芯片接收到地址信息到输出有效数据所需的时间）也是各有差异，从350 ns ~ 1000 ns不等。当需要从存储器中读取数据时，R/W（读/写）信号置1；当向芯片中写入数据的时候，这个信号要置0，而且至少要持续170 ~ 550 ns的时间，也是由所使用的2102芯片的型号决定的。
- 这里我们不得不提到的一个信号就是/CS信号，也称片选信号。该信号置1时，芯片不被选中，意思就是说，不会响应R/W信号。其实，/CS信号的作用不止这些，对芯片还有其他重要的作用，下面我们将简单描述一下。
  - 若让你为8位的微处理器组织存储器的话，你会怎么做呢？是选择按8位存储形式，还是1位存储形式？你肯定会选择前者。如果想存储整个字节，则至少需要8个这样的2102芯片。具体的做法就是，把8个芯片对应的地址信号、R/W及/CS信号连接起来，如下图所示。



实际上，这是一个1024×8位的RAM阵列，或者说是容量为1 KB的RAM。

把存储器芯片安装在一块电路板上，这是很符合实际的做法。那么，到底一块电路板上能安装多少块这样的芯片呢？如果是紧紧排列在一起的话，一块S-100板就能容纳64个。这样一来，就提供了一个8 KB的存储空间。一般我们不这样做，更合适的方法是，用32个芯片组成一个4 KB的存储器。为了存储完整的字节，而连接在一起的芯片的集合，称为**存储体（bank）**。例如，一个4 KB大小的存储器板就由4个存储体组成，而每个存储体又包含8个芯片。

8位微处理器，例如8080、6800，有16位地址，可用来寻址64 KB 的存储空间。如果你制作了一个包含4个存储体、大小为4 KB的存储器板，则存储器板上的16位地址信号就有如下所示的功能。

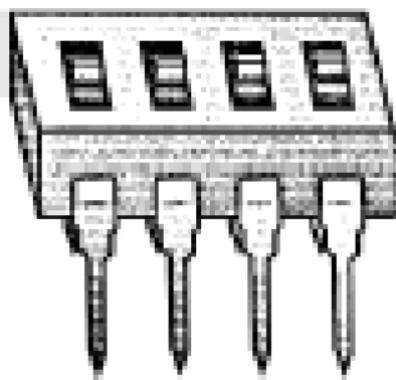


下面详细解释一下这16位地址信号。A<sub>0</sub> ~ A<sub>9</sub>直接与RAM芯片相连接；A<sub>10</sub>和A<sub>11</sub>用来选择4个存储体中要被寻址的那一个；A<sub>12</sub> ~ A<sub>15</sub>确定哪些地址申请用这块存储器板，换言之，就是这块存储器板响应哪些地址。微处理器整个存储空间的大小是64 KB，被划分成16个不同的区域，每个区域的大小是4 KB，我们设计的4KB存储器板占用了其中一个区域。这16个区域划分情况如下。

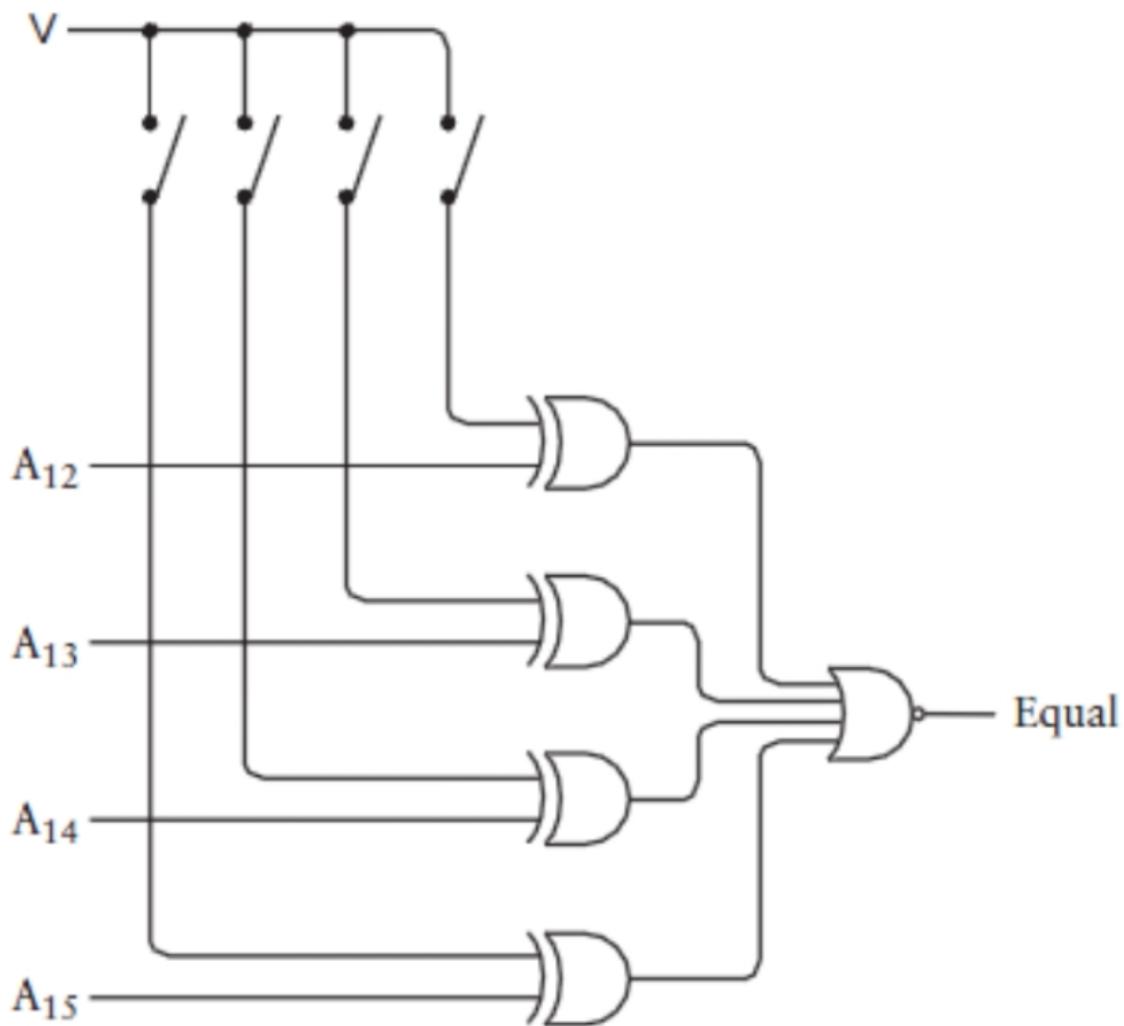
0000h ~ 0FFFh, 1000h ~ 1FFFh, 2000h ~ 2FFFh, ……, F000h ~ FFFFh

举例说明，假定4 KB存储器板使用了A000h ~ AFFFh地址区域。这就意味着，第一个存储体占用了地址A000h ~ A3FFh，第二个占用了地址A400h ~ A7FFh，第三个占用了地址A800h ~ ABFFh，剩下的AC00h ~ AFFFh地址空间分给了第四个存储体。

你完全可以制作一块4 KB存储器板，在用到它的时候再灵活确定其地址范围。要获得这样的灵活性，可以使用一种名为双列直插式封装（dual inline package, DIP）开关的器件。在DIP中，有一系列极小的开关（从2到12个不等）。DIP是可以插在标准的IC插槽中的，如下图所示。



在一种称为比较器（comparator）的电路中，你可以把这个开关和总线上地址信号的高4位连接起来，就像下面这样。

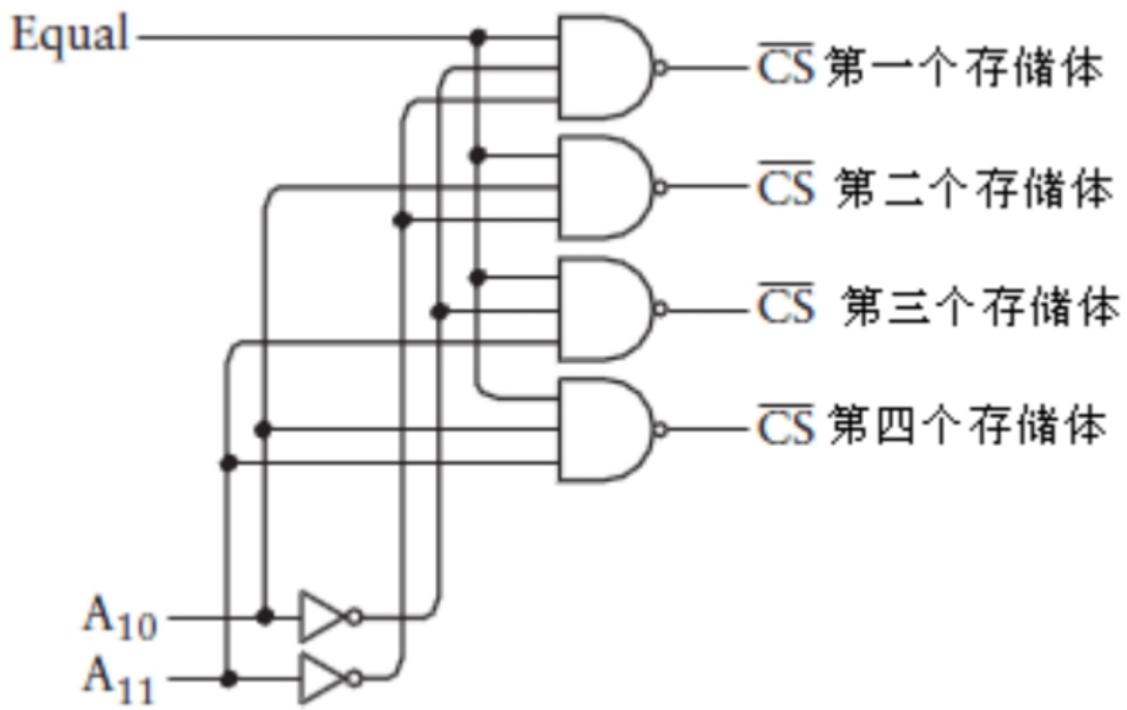


异或 (XOR) 门电路在两个输入端中只有一个高电平时，输出才为高电平；当两个输入端同时为低电平或高电平时，输出是低电平。

例如，如果把A13和A15对应的开关闭合，就意味着让存储器板能响应存储器空间A000h ~ AFFFh (高字节为1010, A)。若总线上的地址信号A12、A13、A14和A15与开关上设置的值相同的话，四个异或 (XOR) 门的输出都是0，或非 (NOR) 门的输出为1，如下图所示。

接下来我们把Equal信号和一个2-4译码器联合起来使用，就能为四个存储体中的每一个都产生一个CS信号，便于对存储体进行选择。具体连接图如下图所示。

例如，若想选择第三个存储体，把A10、A11分别置0和1就可以了。



- 你可能会认为我们还需要8个4-1选择器，用来从4个存储体中选择正确的数据输出信号。但我们并没有这么做，下面来讨论下原因。

通常情况下，TTL兼容集成电路的输出信号要么大于2.2V（逻辑1）要么小于0.4V（逻辑0）。试想一下，如果把输出信号连接起来会发生什么呢？一个集成电路的输出为1，另一个集成电路的输出为0，若把这两个输出连接在一起，结果又是什么呢？恐怕谁也无法回答。就是由于这种不确定性，一般不会把集成电路的输出信号连接在一起。

2102芯片的数据输出信号是**三态 (tri-state)**的，也就是说，除了逻辑0和逻辑1之外，数据输出信号还有第三种状态。我们必须清楚地认识这种状态——它其实是一种“真空”态，就像芯片的引脚上什么也没连一样。当片选信号为1的时，2102芯片的数据输出信号就会进入这种状态。这样一来，我们可以把4个存储体相应的数据输出信号连接在一起，并且可以把8个输出复用作为总线的8个数据输入信号。

之所以强调三态输出的概念，是因为它对总线的操作是至关重要的。几乎所有连接在总线上的器件都使用由总线传递而来的数据输入信号。但不管何时，连接在总线上的电路板中只有一个能确定总线数据输入信号的类型，其他电路板处于三种状态中的无效状态。

- 2102是一款静态随机访问存储器芯片 (Static Random Access Memory, SRAM)，它与动态访问存储器 (Dynamic Random Access Memory, DRAM) 是不同的。通常对于每1位存储空间，SRAM需要用4个晶体管（在第16章中讲过将触发器作为存储器用，其用到的晶体管更多），而DRAM只需要1个晶体管，但DRAM需要较复杂的外围支持电路，这正是它的缺点。

SRAM芯片，例如2102，在电源持续供电的情况下，其内容就能保留下来；一旦掉电，其内容就会丢失。在这方面，DRAM和SRAM很类似。但不同的是，DRAM芯片在使用时需要定期访问其存储器中的内容，尽管有时并不需要这些内容。这一过程称之为更新 (refresh) 周期，每秒钟都必须进行几百次。这种做法就好像为不让某人入睡而每隔一段时间就用手肘轻推他一样。尽管业界在使用DRAM上有些争论，但近年来，DRAM芯片的容量日益增加，使得DRAM最终成为标准。1975年，英特尔公司推出了一款DRAM芯片，容量为16,384位。其实，DRAM芯片在容量上基本每三年翻两番，符合摩尔定律。如今，计算机主板上一般都配备内存插槽，这些内存插槽可以容纳几块小存储器板，分为单列直插内存模块 (single inline memory modules, SIMM) 和双列直插内存模块 (dual inline memory module, DIMM) 两种，里面包含好几个DRAM芯片。如今，花费不到300美元就可以买到128 MB的DIMMs了。

- 应该没有人会把微处理器的整个存储空间都分配给存储器，必须留些空间给输出设备。

- 电子射线管 (cathode-ray tube, CRT) ——20世纪上半个世纪，在家庭中常见的物件，它从外观上看就像电视机一样——已经成为最常见的计算机输出设备了。我们称连接到计算机上的CRT为视频显示器 (video display) 或监视器 (monitor)，而称可以为视频显示器提供信号的电子元件为**视频适配器** (**video display adapter**)。通常在计算机中，视频适配器是独立存在的，它们拥有自己的电路板，也就是我们常说的**显卡** (**video board**)。
- 表面上看来，视频显示器或电视机的二维图像很复杂，但实际上它是由一束连续的光束射线迅速扫描屏幕而形成的。射线从屏幕左上角开始，从左到右进行扫描，到达屏幕边缘后又折回向左，进行第二行扫描。我们称每一个水平行为扫描行 (scan line)，称射线回到每个扫描行的开始位置为水平回归 (horizontal retrace)。当完成了对最后一行的扫描时，射线不会停下来，它会从屏幕的右下角返回到屏幕的左上角 (垂直回归，vertical retrace)，并重复上一过程。就拿美国的电视信号来说，每秒钟要进行60次 (称为场频，field rate) 这样的扫描。由于扫描的速度很快，所以不会看到图像出现闪烁的现象。
- 电视机采用的是隔行 (interlaced) 扫描技术，情况要复杂些。我们先来看一下帧 (frame) 的概念，帧是一个完整的静态视频图像，两个场 (field) 才能形成一个单独的帧。整个帧的扫描线分由两个场来完成——偶数扫描线属于第一个场，奇数扫描线属于第二个场。这里要说明一下水平扫描频率 (horizontal scan rate) 的概念，即扫描每个水平行的速率，例如15,750 Hz。把这个数除以60 Hz，结果是262.5行，这正是每个场所包含的扫描线的数目，整个帧的扫描线的数目是场的两倍，也即525行。
- 不管隔行扫描技术是怎样实现的，组成视频图像的连续射线都是由一个连续的信号所控制。虽然一套电视节目的声音和图像部分是一起播出的，但若想把它们广播出去或者通过有线电视系统传送出去，就不得不分开进行。这里所说的视频信号其实与VCR、录像机、摄像机及一些电视机上的视频输入或输出信号是一样的。
- 黑白电视机的视频信号十分简单且易于理解 (彩色电视机要稍微复杂些)。每秒钟扫描60次，扫描信号包含一个垂直同步脉冲 (vertical sync pulse)，用来指示一个场的开始。这个脉冲为0 V，宽度约为400 ms。相比较而言，水平同步脉冲 (horizontal sync pulse) 则用来指示每个扫描行的开始：视频信号为0 V，宽度为5 ms，每秒钟出现15,750次。在两个水平同步脉冲之间，信号的电压是在0.5 ~ 2.0 V范围内变化的，其中0.5 V表示黑色，2.0 V表示白色，处于两者之间的电压则表示一定的灰度。
- 正是由于上述原因，电视才会出现部分是数字图像、部分却是模拟图像的情况。虽然在垂直方向上，图像被分为525行，但每个扫描行的电压却是连续变化的——用来模拟图像的可视强度。这并不等于说，电压可以随意地变化。事实上，电视机能响应的信号变化频率是有上限的，我们称这一上限为电视机带宽 (bandwidth)。
- 在通信领域中，带宽是极其重要的概念，某个特定的传输媒介能够传输的信息量都是受带宽限制的。以电视机为例，带宽限制了视频信号从黑到白然后又回到黑这一变化的速率。对于美国的广播电视来说，带宽大约为4.2 MHz。
- 一旦我们把视频显示器连接到计算机上，就不该把它作为模拟和数字的混合设备来对待，把它看做是完完全全的数字设备更合适一些。从计算机的角度来说，我们可以很方便地把视频图像想象成由离散点组成的矩形网格，这些离散点称为像素 (这一术语来自picture element)。
- 水平扫描行上像素的个数是受带宽严格限制的。在这里，我把带宽定义为视频信号从黑到白然后又回到黑的变化速率。如果电视机的带宽为4.2 MHz，它就允许2个像素每秒420万次的变化，或者——用 $2 \times 4,200,000$ 除以水平扫描速率15,750——每个水平扫描行有533个像素。但并不是所有的像素都可用，约1/3的像素被隐藏了起来——处于图像的远端或射线的水平回归中。这样算来，水平扫描行上可用的像素约为320个。

- 与水平方向类似，垂直方向上525个像素也不是都可用。原因是，像素在屏幕的顶部、底部以及垂直回归期间都会有所损失。当计算机采用电视机作为显示器时，就不依赖于隔行扫描技术了，垂直方向上有合理的像素数目200。因此我们可以说，早期普通电视机上的视频适配器的分辨率为 $320 \times 200$ ，即水平方向上有320个像素、垂直方向上有200个像素。
- 如何确定网格中像素的总数呢？你可以去数一下，也可以简单地用320乘以200得出结果64,000。每个像素可以是黑色或白色，或者为某一特定的颜色，这要取决于视频适配器的配置。
- 现在我们想在显示器上显示出一些文本字符，那么到底能显示出多少呢？很明显，这取决于每个字符所用的像素数。下面是一种可行的方法，每个字符使用 $8 \times 8$ 的网格（64个像素）。

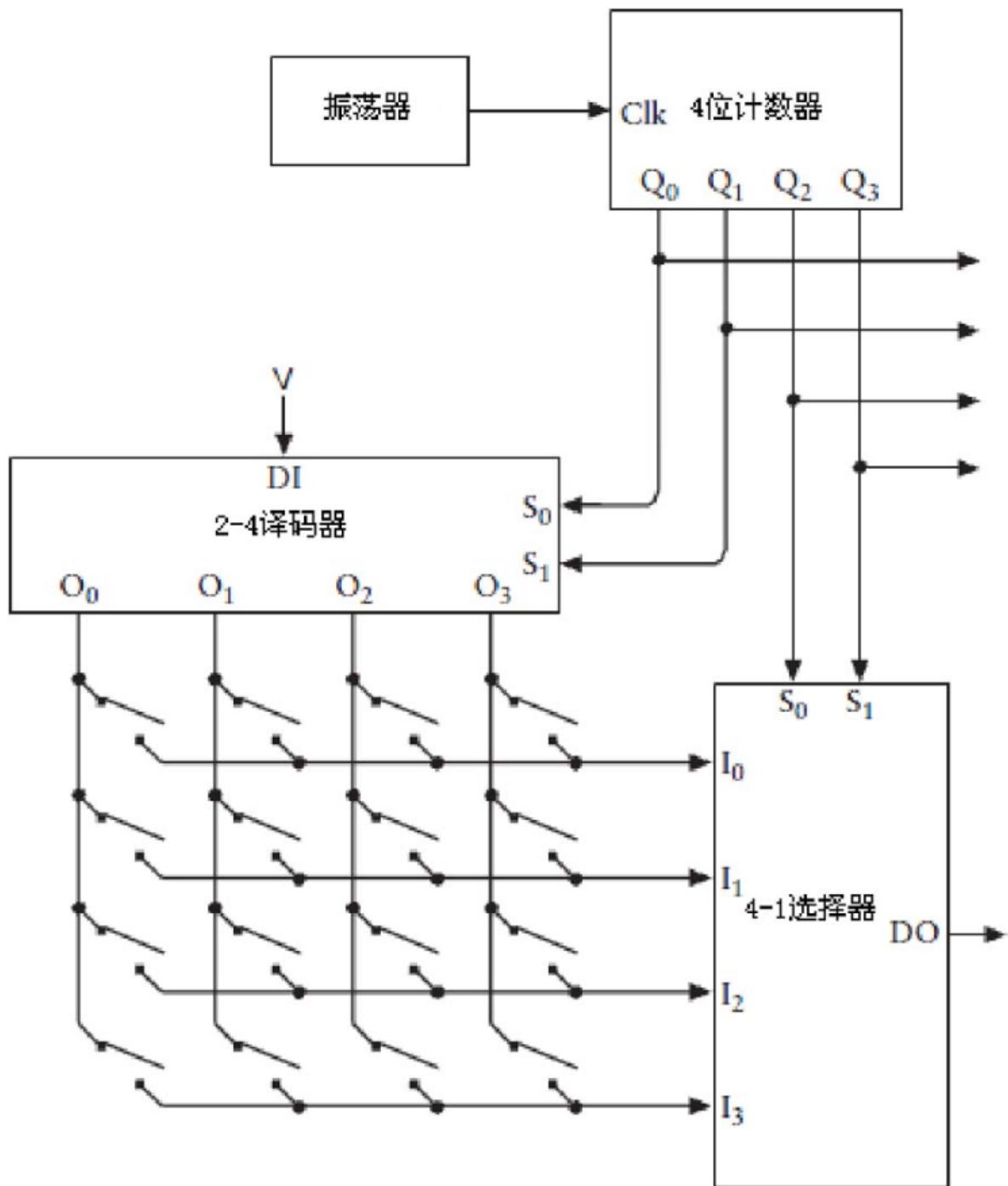


- 视频适配器中必须配置一些RAM，用以存储所显示的内容；微处理器也必须能够向此RAM中写入数据以改变显示器上显示的内容。更方便的是，这个RAM也是微处理器存储空间的一部分。那么，上面描述的显示适配器需要多大的RAM呢？这个问题并不好回答！我们只能说，结果可能处于1 KB ~ 192 KB之间。
- 我们从最简单的情况去考虑。怎样减少显示适配器的内存需求呢？一种方法是限制适配器的功能，让其只显示文本。我们已经明确地知道，视频显示器的每屏幕能显示25行、每行40个字符，也可以说，总共能显示1000个字符。这样一来，视频卡上的RAM只需存储这1000个字符的7位ASCII码。  
1000×7bit，大小约为1024字节，即1 KB。

- 字符生成器 (character generator) 也是视频适配器板上的一部分，包含了所有ASCII码字符的像素图。通常，它是**只读存储器 (read-only memory)**，即ROM。它是一种集成电路，在生产时里面已经填入了数据，固定的地址输出的数据是不变的。ROM中并没有数据输入信号，这点与RAM不同。
- 你可以把ROM看成是可以进行代码转换的电路。每片ROM都有7个地址信号（用来表示ASCII码）及64个数据输出信号，里面存储了128个ASCII码字符的 $8 \times 8$ 像素图。因此，ROM可以实现7位ASCII码到64位码（定义了字符显示的外观）的转换。但是你有没有想过，64个数据输出信号会使芯片变得很大。更合适的做法是，用10个地址信号和8个输出信号。其中7个地址信号是用来确定ASCII码字符的（这7个地址位来自视频板上RAM的数据输出）。其他三个地址信号则用来表示行。举个例子来说，最高行用000表示，最低行用111表示。8个输出位就是每行的8个像素。
- 只显示文本的视频显示适配器还必须支持光标 (Cursor) 功能。光标是一个小小的下画线，用来表明从键盘上输入的下一字符会在屏幕的什么位置显示出来。光标所在的行和列常被存储在两个8位的寄存器中，这两个寄存器也是视频板的一部分，而且微处理器可以对其进行写操作。
- 有的显示适配器不仅仅只显示文本，还可以显示其他数据，我们称这样的显示适配器为图形适配器（图形显卡）。通过向图形显卡上的RAM写入数据，微处理器就可以画出图形了，当然能显示各种大小和样式的文本。相比较而言，图形显卡要比只显示文本的显卡所需的存储空间更大。 $320 \times 200$ 的图形显卡有64,000个像素，如果每个像素需要1位RAM，那么这样的图形显卡就需要64,000位的RAM，即8000字节。然而，这只是最低的要求。1位是和1个像素相对应的，只能用来表示两种颜色——例如黑白两色。0可能对应于黑色像素，1可能对应于白色像素。
- 让我们仔细观察一下黑白电视机，很快会发现，它们不仅仅只显示黑色和白色，还能显示不同灰度的色彩。为了让图形显卡拥有这种功能，通常每个像素对应于RAM中的一整个字节，其中00h表示的是黑色，FFh表示的是白色，介于两者之间的数值对应不同的灰度。一个 $320 \times 200$ 的视频板若能显示256 ( $2^8$ ) 种灰度，就需要64,000字节的RAM。这与一直在讨论的某个8位微处理器的整个地址空间非常接近。
- 如果想显示出丰富多彩的颜色，每个像素就需要至少3个字节。如果现在你手头有放大镜的话，不妨用它观察一下彩色电视机或计算机视频显示器，你会发现，每种颜色都是由红、绿、蓝三原色的不同组合而形成的。为了获取所有的颜色，三原色中每种颜色的强度都需要用一个字节来表示。这么算来，就需要192,000字节的RAM（更多有关彩色图形的内容将在本书最后一章介绍）。
- 图形显卡到底能显示出多少种不同的颜色呢？这与每个像素所赋予的比特数是有关的。对于这种关系，你可能会感到很熟悉，因为本书中讲到的很多编码都与之类似，它们都涉及2的幂，它们之间的关系如下：颜色数量 =  $2^{\text{每个像素赋予的比特数}}$
- 在标准的电视机上， $320 \times 200$ 的分辨率是所能达到的最高分辨率。正是由于这样的原因，我们要为计算机特制显示器，以使其具有比电视机更高的带宽。1981年，第一台显示器随IBM PC一起销售，它可以显示25行，每行80个字符。这正是使用在IBM大型机上的CRT显示器能显示的字符数目。对于IBM来说，80个字符具有特殊的意义，为什么这样说呢？因为它和IBM的打孔卡片 (punch card) 上的字符数目一样。的确，早期连接到主机上的CRT显示器常被用来显示打孔卡片上的内容。偶尔，你会听到有人称只显示字符的视频显示器为卡片，当然这是一种过时的叫法。
- 这么多年以来，视频适配器的分辨率以及能显示的颜色不断增加，这两者也成为了视频显示适配器的重要参数。到了1987年，水平640像素、垂直480像素的视频适配器被IBM的PS/2个人计算机和苹果公司的Macintosh II机采用，这种适配器的出现起到了里程碑的作用，因为从那时起 $640 \times 480$ 就是视频分辨率的最低标准了。
- $640 \times 480$ 的分辨率具有很重要的意义。也许你可能无法相信，它之所以那么重要，是因为它和托马斯·爱迪生 (Thomas Edison, 1847-1931) 有关。大概在1889年，爱迪生及他的工程师威廉·肯尼迪·劳里·迪克生 (William Kennedy Laurie Dickson) 正在进行活动电影摄影机和活动电影放映机的研究，他们决定：让电影图像的宽比高多出1/3。图像的宽和高之比，称为屏幕长宽比 (aspect ratio)。通常，我

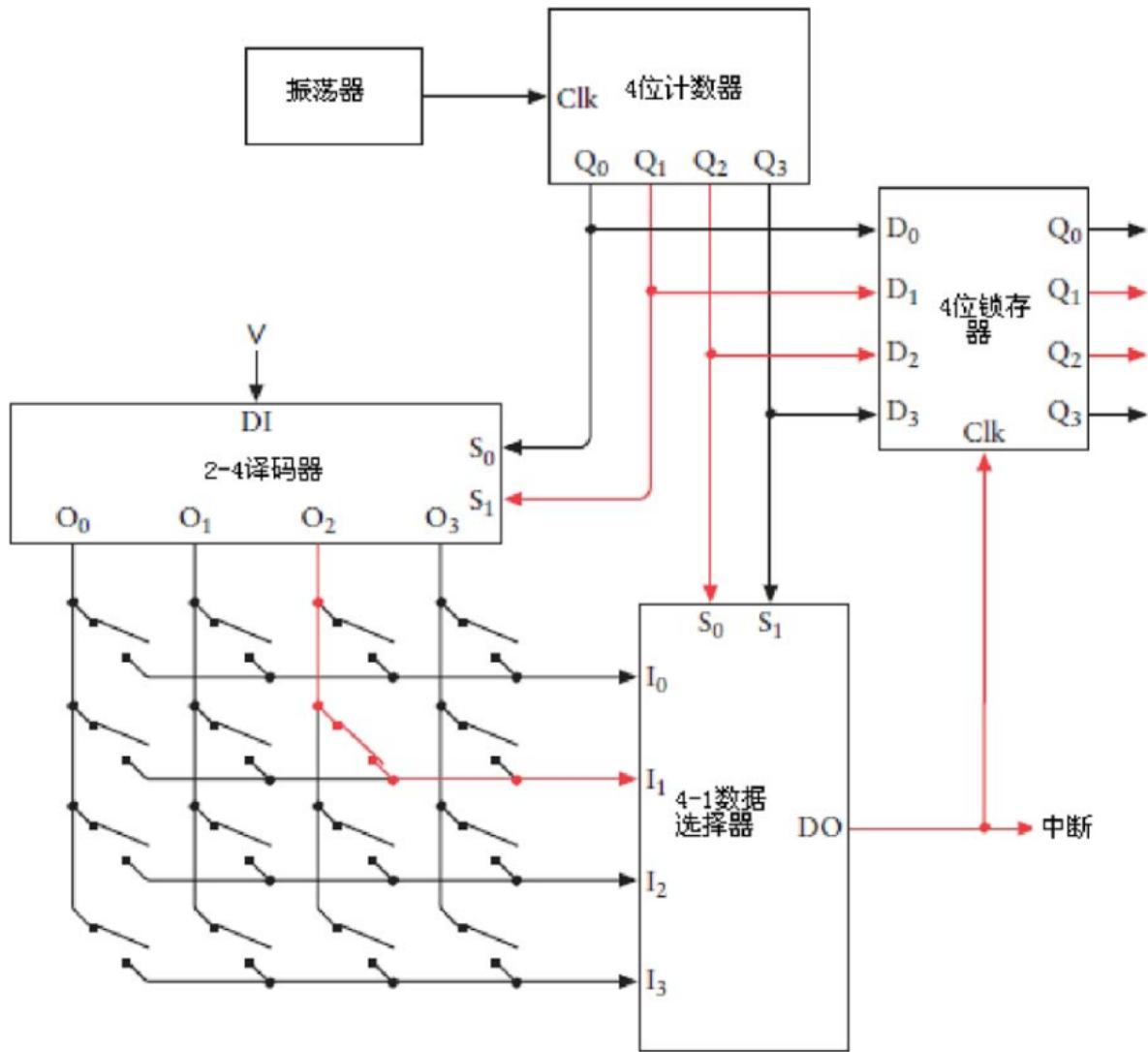
们把爱迪生和迪克生所确定的这个比表示成1.33:1，或者不想使用小数点的话，就表示成4:3。60多年了，大多数电影一直采用这个比例，电视机也是如此。但在20世纪50年代早期，好莱坞引入宽屏(widescreen)技术，与电视展开竞争，并最终打破了这个比例。

- 多数计算机的显示器的长宽比也是4:3，如果你不信的话，可以用尺子实际量一下，就可以证明我所说非虚。640×480分辨率也是这个比例。这就说明（打个比方）100个像素的水平线和100个像素的垂直线有着相同的物理长度。对于计算机图形学来说，这是个非常重要的特性，我们称为正方形像素(square pixel)。
- 如今，视频适配器和显示器都支持640×480的分辨率，但同样也支持多种其他的视频模式，包括800×600、1024×768、1280×960、1600×1200。
- 如果要让连接到计算机上的键盘能正常工作的话，就需要配备一些硬件来为每个按键提供唯一的代码，以便区分哪一个按键被按下了。假定这个代码就是按键的ASCII码，这样可行吗？你的答案或许是肯定的，但要设计出能识别ASCII码的硬件却是不切实际的。举例来说，键盘上的按键A对应的ASCII码可能是41h，也可能是61h，具体是哪个，还取决于用户是否按下了Shift键；另外，现在计算机键盘上有很多的按键并没有ASCII码与之对应。我们称键盘硬件提供的代码为扫描码(scan code)。当按下键盘上的某个按键时，一小段计算机程序就会计算出这个按键对应的ASCII码（如果有的话）。
- 这里为了避免键盘硬件的电路图太复杂，假设键盘上只有16个按键。任何一个按键被按下，键盘硬件就会产生一个4位的代码，二进制数值范围是0000到1111。



上图左下部分所示的是键盘的16个按键，简单地用开关表示。4位的计数器在按键对应的16个编码间快速且重复地循环着，循环的速度必须足够快，以保证在按下并松开一个按键之前循环已经结束。

4位计数器的输出同样也是2-4译码器和4-1数据选择器的输入。在没有按键按下的情况下，选择器的输入全都不为1，因此，其输出也不为1；一旦有某个按键被按下，而且与4位计数器某一特定输出相对应，那么选择器的输出就为1。例如，如果右上角对角线方向的第二个开关被按下，且计数器的输出是0110，选择器就会输出1，如下图所示。



0110就是这个按键的代码。在这个按键按下的情况下，计数器的其他输出都不会使选择器的输出为1，也就是说每个按键都代码都是唯一的。

扫描码的位数是由键盘上按键的数目确定的。如果键盘上有64个键，就需要6位的扫描码，也就需要一个6位的计数器。用一个3-8译码器和一个8-1选择器就可以把这些按键组成一个 $8 \times 8$ 的阵列。如果键盘上的按键数目为65~128个，就需要7位的扫描码。你就可以用一个4-16译码器和一个8-1的选择器（或者一个3-8译码器和一个16-1选择器）把这些按键组成一个 $8 \times 16$ 的阵列。

在这个电路中，接下来将会发生什么事情呢？这取决于键盘接口。每个按键都应该在RAM中拥有1位的存储空间，这是设计键盘硬件时该考虑的事情。而且这些RAM是由计数器寻址的，RAM的内容为0或1，具体是什么值取决于按键按下（RAM为1）与否（RAM为0）。微处理器是可以读取RAM中的内容的，并通过内容判断每个按键的状态。

中断信号是键盘接口一个很有用的信号。回想一下前面讲过的内容，我们知道8080有一个输入信号允许外部设备中断当前微处理器正进行的工作。微处理器是通过从内存中读取一条指令来响应中断的，通常是一条RST指令。这条指令使微处理器跳转到内存中一个特定的区域并执行其中的中断处理程序。

- 最后，我们要介绍一下能够长期存储信息的外围设备。前面曾提到，无论是用继电器、电子管，还是用晶体管作为介质构成随机访问存储器，一旦掉电，它存储的内容就会丢失。正因如此，能够在掉电时长期保存信息的存储器，是一台完整的计算机不可或缺的组成部分。长期以来，人们通过在纸上或卡片上打孔来保存永久信息，IBM打孔卡片是其中典型的代表。在早期小型计算机中，为了能够长久保存程序和数据，通常在滚动的纸带上打孔，而在需要时，这些程序和数据可以从纸带加载到内存。

- 打孔卡片和纸带的使用也不是尽善尽美的，它也存在一些问题。首先是介质的不可重用性，一旦打孔卡片或纸带被打孔后就很难还原为原来的状态。其次是效率很低，假如你有机会看到当时纸带上保留的某一比特信息，就会发现这种做法实在太浪费纸带了。
- 正是由于这些原因，打孔卡片和纸带慢慢退出了历史的舞台。**磁介质存储器 (magnetic storage)** 逐渐发展成目前最为流行的长期存储器。磁介质存储器的起源要追溯到1878年，这一年美国工程师奥柏林·史密斯 (Oberlin Smith, 1840-1926) 描述了它的工作原理。1898年，即工作原理被提出20年后，第一块可用的磁介质存储器问世，它由丹麦发明家巴尔德马尔·波尔森 (Valdemar Poulsen, 1869-1942) 制造。波尔森后来发明了录音电话机，当家里没人时，通过它可以记录收到的电话信息。声音通过电磁铁和可变长度的金属丝来记录，其中电磁铁是电报机里很常见的部件，它根据声音的高低来磁化金属丝。当磁化的金属丝切割电磁线圈运动的时候，产生的电流强度与其磁化程度有关。不论使用何种磁化介质，记录和读取信息都是利用电磁铁的**磁头 (head)** 来完成的。
- 1928年，澳大利亚发明家弗里茨·佛勒玛 (Fritz Phleumer) 发明了一种磁记录设备，并为其申请了专利。此设备采用在生产香烟上的金属带时所用的技术，将铁粒子覆盖在很长的纸带上。不久以后，纸带被强度更高的醋酸盐纤维素取代，而一种更耐久、更知名的记录介质也从此诞生——卷轴式磁带，它被包装在塑料盒里，可以很方便地使用。对于记录和回放音乐及视频来说，卷轴式磁带无疑是很受欢迎的介质。
- 1950年，雷明顿兰德公司 (Remington Rand) 发明了第一个用于记录计算机数字数据的商用磁带系统。当时，磁带的容量有限，一个0.5英寸的卷轴式磁带容量只有几兆字节。在早期家用计算机中，常见的盒式磁带录音机被人们用来保存信息。通过调用一些小的程序，可以将内存块中的内容保存到磁带上，以后再需要时还可以从磁带调入内存中。在第一代IBM PC上，有一个专为连接盒式磁带存储器而设计的接头。至今，磁带仍然是一种很通用的存储介质，对于那些想要长期保存的文档，磁带更是首要的选择。但是，磁带并不是最理想的存储介质，想要快速地移动到磁带的任一位置是不可能的，它只能顺序访问，频繁地快进和倒带会花费很多时间。
- 以几何学角度来看，磁盘是能够实现快速访问的介质。磁盘围绕其中心旋转，连到臂上的一个或多个磁头从磁盘外沿向中间移动，通过磁头可以快速访问磁盘上的任何区域。
- 在记录声音信息方面，磁盘实际上要早于磁带。早在1956年，IBM公司就发明了世界上首款用来存储计算机数据的磁盘驱动器，称为RAMAC (Random AccessMethod of Accounting and Control, 计算与控制过程中的随机访问模式)，它由50个金属盘片组成，直径2英尺，存储量为5 MB。
- 自从磁盘被用作记录信息以来，它的体积越来越小而容量越来越大，习惯上将磁盘分为软盘 (floppy disk) 和硬盘 (hard disk, 或fixed disk)。软盘是由单面覆盖磁性物质的塑料片组成，外面由厚纸板或塑料包装，起到保护作用（塑料包装主要是防止磁盘弯折，虽然现在的磁盘看起来不像以前的软盘那么松软，还有很多的区别，但软盘这个名字一直在用，延续至今）。在使用软盘的时候，必须将其插入到软盘驱动器，软盘驱动器是一个连接到计算机的部件，通过它可以读/写磁盘中的内容。早期的软盘直径为8英寸，第一代IBM PC使用的是5.25英寸的软盘，不过，现在最流行的是3.5英寸的软盘。软盘有很大的灵活性，可以实现在不同计算机之间传递数据。对商用软件来说，磁盘仍然是一个不可或缺的发行媒介。
- 硬盘是由多个金属磁盘构成的，它永久驻留在驱动器里。相对于软盘来说，它的存取速度更快、存储量更大，唯一的缺点是硬盘本身是固定的，不能移动。
- 磁盘的表面被划分成许多同心圆，称为**磁道 (tracks)**，每个磁道又被划分成像圆饼切片形状的**扇区 (sectors)**，每个扇区可以存放一定数量的字节，通常为512字节。第一代IBM PC上使用的软盘只有一面可以存储信息，直径5.25英寸，被划分成40个磁道，每个磁道8个扇区，每个扇区存储512字节数据。通过计算，每个软盘可以存储163,840字节数据，即160 KB。现今PC兼容机使用的软盘通常有两面，每面80个磁道，每个磁道18个扇区，每个扇区512字节，总的磁盘容量是1,474,560字节，即1440 KB。

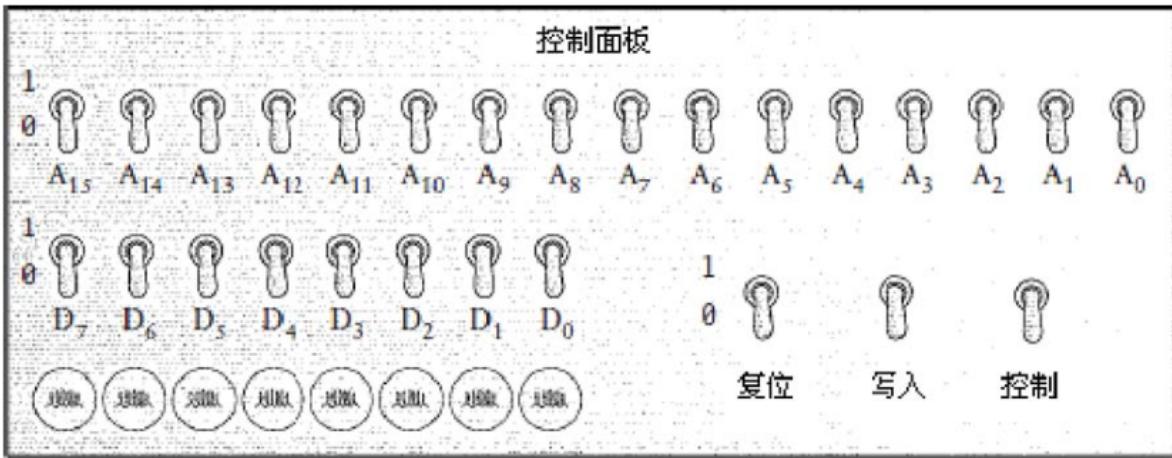
- 1983年，IBM在其PC/XT上率先使用了硬盘驱动器，当时硬盘的容量仅仅只有10MB。自此之后的16年里，硬盘的容量扩大了上百倍，而价格一直在下降。1999年，20 GB（200亿字节）容量的硬盘驱动器诞生了，而售价却不到400美元。
- 软盘和硬盘通常有它们自己的电气接口，除此之外，为了能和微处理器交互数据，这些电气接口与微处理器之间还需要有额外的接口与之相连。现在流行的硬盘驱动器标准接口有：小型计算机系统接口（Small Computer System Interface, SCSI）；增强的小型设备接口（Enhanced Small Device Interface, ESDI）和集成设备电气接口（Integrated Device Electronics, IDE）。这些接口都利用**直接内存访问 (direct memory access, DMA)** 技术来使用总线，DMA可以不经过微处理器，实现数据在随机访问存储器和硬盘之间直接传送。这样的传送是以块为单位进行的，每次传输的块大小是磁盘扇区字节数的倍数，通常是512字节。
- 由于经常听到关于兆字节和吉字节之类的相关术语方面的谈论，让很多家用计算机的初学者感到困惑：到底随机访问存储器和磁盘存储器有什么区别？不过最近几年推出了一条分类规则，这让人们对术语的困惑减少了许多。这条规则规定：**memory (内存)** 仅仅表示半导体随机访问存储器；**storage (存储器)** 用来指任何的存储设备，通常包括软盘、硬盘和磁带。在本书中，尽量遵循这些规则，它的确能够给我们带来许多好处。
- 实际上，存储的信息是否易失，是随机访问存储器与磁介质存储器的主要区别。随机访问存储器是易失性存储设备，一旦掉电，存储内容将会消失；而磁介质存储器是永久性存储设备，像软盘和磁盘，除非故意删除或写覆盖，否则数据将会一直保留不变。如果对微处理器的工作原理很了解，就会观察到随机访问存储器和磁介质存储器之间的显著区别，例如当微处理器发出一个地址信号，通常是寻址随机访问存储器，而非磁介质存储器。
- 微处理器不能直接从磁盘读取数据，需要将所需的数据从磁盘调入内存（随机访问存储器），然后它才能对其访问，当然这需要额外的步骤。微处理器还需要执行一段小程序，这段程序会访问磁盘，并将数据从磁盘调入内存。
- 关于随机访问存储器和磁介质存储器之间的差别，有个形象的比喻可以帮助我们加深理解：随机访问存储器就像办公桌的桌面，上面的任何东西都可以拿来直接使用；而磁介质存储器就像一个文件柜，里面的东西不能直接使用，如果想要使用放在文件柜里的某件东西，你需要站起来，走到文件柜前，查找需要的文件，然后带回桌面。如果桌面太拥挤，没有空间放置需要的文件，还需要把桌面上暂时不用的东西先放回到文件柜中。
- 这个比喻恰到好处，实际上，存储在磁盘上的数据的确是以文件（files）作为实体来存放的。存储和检索文件是操作系统（Operating System）很重要的一个功能。

## 22 操作系统

---

- 一直以来，有一种想法在我们脑子里涌动：亲手去组装——在想象中进行虚拟组装也可以——一台近似完整的计算机。一块微处理器、一些随机访问存储器、一款键盘、一台视频显示器和一个磁盘驱动器是这台计算机所拥有的部件。当所有的硬件各就各位，我们激动地盯着计算机的开关，伸出手来给它上电，将这台计算机从沉睡中唤醒。但我们还是漏了一些东西，运行这台新组装的计算机，请告诉我你眼前出现了什么？
- 当阴极射线管加热之后，一串排列整齐——而又完全随机的——ASCII码字符阵列出现在屏幕上。不出我们所料，掉电的时候，半导体存储器中的内容就会被全部清零；而首次给它上电的时候，它将处于随机且不可预测的状态。同样，我们用来构建微处理器的所有RAM都包含随机的字节。如果可能，开机后微处理器会将这些随机字节解释为机器代码并执行。不用担心会因此发生什么糟糕的事情——计算机不会因此而坏掉——但是，计算机也无法完成任何有意义的工作。

- 这里我们漏掉的就是软件。当一个微处理器首次上电或复位时，它会从特定的内存地址开始执行机器代码。在英特尔的8080系统中，这个地址就是0000h。在一台设计精良的计算机中，通过上电启动，将会有一条机器代码指令被载入到该内存地址中（一般情况下是一段程序的第一条指令）。
- 机器代码指令是怎么加载到那个内存地址中的呢？把软件安装到一台新设计的计算机的过程，可能是整个过程中最令人困惑的部分了，怎样理解它呢？让我们先从第16章所讲的一个控制面板入手吧，这个控制面板的功能是把字节写入随机访问存储器，之后还可将其读出。



与前面介绍过的有所不同，这个控制面板上设计了一个复位开关。复位开关与微处理器的复位输入相连接。一旦复位开关闭合（置1），微处理器就会停止工作；当此开关断开后（置0），微处理器就开始执行机器代码。

使用此控制面板的方法是：打开复位开关，微处理器复位并停止执行机器代码；打开控制开关，就会接收总线上的地址信号和数据信号。在该状态下，你可以通过开关A0 ~ A15来指定一个16位的存储器地址；通过灯泡的明灭组合来显示该存储器地址中的8位数据。那么怎样把一个新的字节写入到此地址中呢？首先通过开关来设置想要写入的字节，然后把写入开关先打开再关闭。当你已经完成向存储器中插入字节的工作后，关上控制及复位开关，微处理器就会执行程序。

上面这个过程展示了向这台我们刚刚打造出来的计算机输入第一条机器代码的步骤。不言而喻，这是一个耗时耗力的过程。在这个过程中一些小错误是在所难免的。

- 但当你开始用视频显示器显示程序运行的结果时，到底是什么使这一切都变得简单、方便呢？上一章中我们讲到只显示字符的视频显示器，它有一个1 KB的随机访问存储器，可以存储25行、每行40个字符的ASCII码。程序将要显示的内容写入到此存储器中，其方法与向计算机中其他存储器中写入数据的方法一样。
- 尽管把程序的输出结果显示在视频显示器上看似简单，实则不然。例如，你编写了一个程序用来完成某个计算任务，如果计算结果是4Bh，不能将这个值直接写入到视频显示器的内存中。如果犯了这样的错误，屏幕上显示的将是字母K，因为此字母的对应ASCII码的值正是4Bh。4Bh由两个字符组成，其中4对应的ASCII码是34h，B对应的ASCII码是42h，应该将这两个ASCII码写到视频显示器存储器上，才能在显示器上看到期望的数值。这里再强调一下，8位二进制数可以表示两位十六进制数字，因此必须将每一位十六进制数字对应的ASCII码，写入到视频显示器的存储器中才能显示这个数。

这种转换可以通过编写小的程序来实现。下面是一段8080汇编程序，功能是把存储在累加器中的十六进制数（假设这个数介于00h与0Fh之间）的每一位转换成对应的ASCII码：

```

NibbleToAscii: CPI A, 0Ah ; 检查是数字还是字母
                JC Number
                ADD A, 37h ; 把 A~F 转换成 41h~46h
                RET
Number:          ADD A, 30h ; 把数字 0~9 转换成 30h~39h
                RET

```

通过两次调用NibbleToAscii，下面的程序实现了把累加器A中的一个字节转换成两个ASCII码对应的数字，分别存放在寄存器B和C中。

```

ByteToAscii:    PUSH PSW           ; 保存累加器 A
                RRC
                RRC
                RRC
                RRC           ; 获取高半字节
                CALL NibbleToAscii ; 转换成 ASCII 码
                MOV B, A         ; 把结果存入寄存器 B 中
                POP PSW          ; 取出原始的 A
                AND A, 0Fh        ; 获取低半字节
                CALL NibbleToAscii ; 转换成 ASCII 码
                MOV C, A         ; 把结果存入寄存器 C 中
                RET

```

通过这些程序，可以把一个用十六进制表示的字节显示在视频显示器上。进一步来讲，如果想把它转换成十进制数，还需要做些别的工作。

记住，到此为止实际上你还没有将汇编语言程序写入到内存。你需要把汇编语言写到纸上，将它们转换成机器代码后才写入到内存中。直到第24章我们一直都会采用这种“手工汇编”的方式。

- 控制面板的确不需要很多硬件支持，但它不便于使用，因为它有着最糟糕的输入/输出形式。我们甚至可以从头开始独立建造一台计算机，但仍然无法改变这样糟糕的输入/输出方式——通过按键输入0和1——这的确让人尴尬。如何把控制面板去掉是要解决的首要问题。
- 实现按键来控制输入/输出的不二之选就是键盘。前面我们已经搭建了一个计算机键盘，每次有按键按下时候，就会产生一个中断信号送至微处理器。计算机内有中断控制芯片，通过执行一条RST指令使得微处理器响应这次中断，例如RST 1，微处理器执行这条指令，把当前程序计数器的值压入到堆栈中，然后跳转到地址0008h处。可以直接在这片地址空间上输入一些代码（使用控制面板），这些代码称为键盘处理程序（keyboard handler）。
- 为了使复位后微处理器能正常工作，微处理器在复位的时候需要执行一些代码，称为初始化代码（initialization code）。堆栈指针在运行初始化代码的时候会被设置，以保证堆栈处在内存的有效区域内。为了不让屏幕上显示随机字符，初始化代码还把视频显示器内存中的每个字节设置成十六进制数20h，在ASCII码中这是一个空格符。此外，初始化代码还要把光标定位在第一行第一列的位置——OUT（Output）指令可以完成这一操作：光标在视频显示器上是以下画线的形式出现的——它可以显示出下一个要输入字符的位置。为了使微处理器能响应键盘中断，必须设置EI指令开中断，而HLT指令可以使微处理器停止工作。
  - 上面讲述的就是初始化代码的作用。执行了HLT指令后，计算机则处于停机状态。为了把计算机从停机状态唤醒，只能通过控制面板的复位信号或者键盘的中断信号来实现。
  - 键盘处理程序的规模要远大于初始化代码，这个程序才是真正响应键盘事件的代码段。

- 任何时候，只要键盘上的一个按键被按下，微处理器就会响应本次中断，并从初始化代码末尾的HLT语句跳转到键盘处理程序。键盘处理程序利用IN (Input) 指令用来检查是哪个按键被按下，并根据这个按键执行相关的操作（就是说，键盘处理程序对每个按键进行相应的处理），然后执行RET (Return) 指令以返回HLT语句，等待另一个键盘中断。
- 当你按下字母、数字或者是标点符号键的时候，键盘扫描程序就会启用键盘扫描码，并根据Shift键是否按下与确定相应的ASCII码。接下来我们要做的，就是把这个ASCII码写到视频显示器的内存中，当然这不是随意的，而是要写在光标所在的位置。这样的一个过程，我们可以很形象地称之为按键到显示器的回显 (echoing)。光标会随着字符的写入而移动，换言之，它总会出现刚显示的字符后面的空格处。通过键盘，可以输入一串字符，然后把它们在屏幕上显示出来。
- 当按下回退键（相应的ASCII码值是08h）时，最后写入视频显示器内存中的字符会被键盘处理程序删除（其实并不复杂——我们只要把空格符对应的ASCII码——20h写入到那个内存位置处就行了）。在这个过程中，光标会移回一格。
- 通常我们在输入一行字符时，错误是难免的，这时就需要用退格键来改正错误，然后按下回车键。回车键并不难找到，键盘上标有“Enter”字样的按键就是。打字员在电动打字机上按下“Return”键表示已经完成一行文字的输入，同时也表明他们已经做好了输入下一行的准备，光标会指向下一行的开始。同样，在计算机中“Enter”键用来实现相同的功能，结束一行的输入并转到下一行。
- 当键盘处理程序对“Return”或“Enter”键（对应的ASCII码是0Dh）进行处理时，它把视频显示器内存中的这一行文本解释为计算机的一条命令 (command)，换言之，键盘处理程序的任务是执行此命令。实际上，在键盘处理程序内含有一个命令处理器 (command processor)，它可以解释如下三条命令：W命令、D命令和R命令。下面我们来深入地理解一下它们。

- 首先是W命令。它是以W开头的文本行，此命令用来把若干字节写入 (Write) 到内存中。

**W 1020 35 4F 78 23 9B AC 67**

- 运行这条命令，命令处理器会从内存地址1020h处开始，把35、4F等十六进制表示的字节写入内存中。要完成这项工作，键盘处理程序需要把ASCII码转换成字节——前面讲过把字节转换成ASCII码，这里其实就是它的逆变换。

- 接下来是D命令。它是以D开头的文本行，此命令用来把内存中的一些字节显示 (Display) 出来。例如输入到屏幕上的一行内容如下所示：**D 1030**

- 接收到命令后，命令处理器会把从地址1030h开始的11个字节的内容显示出来（这里之所以说是11个字节，是因为在一个每行可以容纳40个字符的显示器上，除去显示命令与地址标识，后面能显示的也只有这么多了）。有了这条命令，就可以很方便地查看内存中的内容了。

- 最后是R命令。它是以R开头的文本行，表示运行 (Run)。该命令的形式如下：**R 1000**

- 执行此命令意味着“处理器会运行从地址1000h开始的一段程序”。首先命令处理器把1000h存储在寄存器对HL中，接着执行指令PCHL，这条指令的功能是，把HL所存储的值加载到程序计数器中，然后跳转到程序计数器指向的地址并运行程序。

- 键盘处理程序及命令处理器简化了很多工作，可以说是计算机发展的一个里程碑。一旦使用了它，就无须理会那个令人难以忍受的控制面板了。我们不得不承认，使用键盘输入更简单、更快、更好，这是其他方法无法媲美的。

- 但是，你仍然没有摆脱之前的老问题，一旦关掉电源，你辛辛苦苦所输入的数据会全部消失。当然，你可以把所有新代码存到只读存储器 (ROM) 中。
- 还记得上一章中，我们讲到的那个ROM芯片吗？它就包含了把ASCII码字符显示到视频显示器上所需的全部点阵模式。

- 这里假定，这些数据在厂家制造芯片时已经配置好了，当然你也可以对ROM芯片进行编程。可编程只读存储器（Programmable Read-Only Memory, PROM）只能编程一次；而可擦除可编程只读存储器（Erasable Programmable Read-Only Memory, EPROM）可重复擦除和写入，该芯片的正面开有一个玻璃窗口，让紫外线透过这个孔照射内部芯片就可以擦除其中的内容了。
- 回忆前面曾讲过的内容，你一定会想起，RAM板与一个DIP开关相连，有了这个开关就可以设定RAM板的起始地址了。8080系统在初始化时，其中一个RAM板的起始地址被设置为0000h。但是如果有ROM的话，这个地址就会被其占用，而RAM板则转到更高的地址。
- 命令处理程序的使用极大地推动了计算机的发展，通过它不仅可以更快地向存储器输入数据，更重要的是，计算机变得可交互（interactive）了。当你通过键盘输入一些内容，计算机会立即做出响应，把你所输入的内容显示出来。
- 将命令处理程序存储到ROM后，就可以执行操作了：把内存中的数据写入到磁盘驱动器（可能是按照与磁盘的扇区大小一致的块的形式），然后再把数据读回到内存中。因为掉电的时候，RAM中的内容会丢失，所以与RAM相比，把程序和数据存储到磁盘中会更安全（它们不会因为突然断电而丢失数据），就灵活性来说，也比存储到ROM中要好。
- 仅仅有上面的命令还是不够的，还需要向命令处理程序中添加新命令。例如，S命令表示存储（Store）：

S 2080 2 15 3

运行这条命令后，在磁盘的第2面、第15道、第3扇区中将存放起始地址为2080h的内存块数据（被存放内存块的大小是由磁盘扇区的大小决定的）。

类似地，还可以通过加载（Load）命令，把磁盘上相应扇区的内容写回到内存中，如下所示：

L 2080 2 15 3

当然，还要把存储的位置记录下来，这是必须要做的。你可以用手头上的纸和笔来完成。

需要注意的是：你不能把位于某个地址处的代码又加载到内存的另外一个地址中，如果这样的话，程序将不能正常运行。具体来说，程序代码在内存中改变位置后，其跳转（Jump）和调用（Call）指令标识的依然是原来的地址，所以运行时会报错。也存在这样的情况，程序比磁盘的扇区大，这时就需要多个扇区来存放程序。而磁盘上某些扇区已被其他程序或数据占用了，而另外一些扇区是空闲的，可能在磁盘上找不到一块足够大的、连续的扇区来存放程序。

最后，所有的东西存储在磁盘的什么位置，都需要你手工地记录下来，这个工作挺多，也挺麻烦。出于这个原因，**文件系统（file system）**应运而生。

- 文件系统是磁盘存储的一种方法，就是把数据组织成文件（file）。简单地说，文件是相关数据的集合，占用磁盘上一个或多个扇区。更重要的是，你可以为每个文件命名，这有助于记下文件里存放的内容。想象一下，磁盘是不是类似于一个文件柜，每个文件都有个小标签，标签上有文件的名称。
- 操作系统（operating system）是许多软件构成的庞大程序集合，文件系统就是其中的一部分。前面讲到的键盘处理程序和命令处理程序最终也能经过拓展，演变成为操作系统。那么操作系统到底能够做些什么、又是如何工作的呢？这里撇开操作系统漫长的发展演化过程，把重点放在刚才提出的问题上，目的就是试图让大家了解操作系统的工作机制。
- 如果你对操作系统的发展史有一定了解的话，你就会知道CP/M（Control Program for Micros）是最著名的8位微处理器操作系统，它是20世纪70年代中期由加里·基尔代尔（Gary Kildall，生于1942年）专门为Intel 8080微处理器而开发的，加里后来成了DRI（Digital Research Incorporated）公司的创始人。

- CP/M操作系统是存放在磁盘上的。单面、8英寸的磁盘是早期的CP/M最常用的存储介质，它有77个磁道，每个磁道有26个扇区，每个扇区的大小是128个字节（总共算下来共有256,256个字节），CP/M系统存放在磁盘最开始的两个磁道。在启动计算机时，需要把CP/M从磁盘调入到计算机的内存中，下面将介绍这一过程是如何进行的。
- 存放CP/M本身只占用2个磁道，那么剩下的75个磁道用来存储文件。CP/M的文件系统固然简单，但两个最基本的要求还是可以满足的。首先，每个文件在磁盘中都有属于自己的名字，便于识别，这个名字也是存放在磁盘中的；实际上，文件以及读取这些文件需要的所有信息也是一起存储在磁盘中的。其次，文件存储在磁盘中不一定要占据连续的扇区空间，可以想象，由于大小不同的文件会被经常地创建和删除，磁盘上的可用空间就会很零碎。那么如何将文件存储在零碎的空间里呢？这主要得益于文件系统具有很强大的管理功能，它可以把一个大文件分散存储在不连续的磁盘上。
  - 上面曾提到过剩的75个磁道中的扇区用来存放文件，这些扇区按**分配块 (allocation blocks)** 进行分组。每个分配块中有8个扇区，总计1024个字节（每个扇区大小是128字节）。可以计算，在磁盘上共有243个分配块，编号为0~242。
  - 目录 (directory) 区占用最开始的两个分配块（编号为0和1，总共2048字节）中。目录区是磁盘中的一个非常重要的区域，磁盘文件中每个文件的名字和其他一些重要信息都存储在该区域，根据目录就能够很方便、高效地查找文件。存放目录也需要占用空间，磁盘上每个文件对应的**目录项 (directory entry)** 大小均为32字节，由于目录区大小为2048字节，所以这个磁盘上最多可以存放64（ $2048/32$ ）个文件。
    - 为了能够根据目录找到相应的文件，每一个32字节的目录项包含以下信息。

字 节	含 义
0	通常设为 0
1 ~ 8	文件名
9 ~ 11	文件类型
12	文件扩展
13 ~ 14	保留（设置为 0）
15	最后一块扇区数
16 ~ 31	磁盘存储表

在目录项中，第一个字节用来设置文件的共享属性，只有文件系统被两个或更多人同时共享时才设置此字节为1。在CP/M中，这个字节跟第13、14字节一样，通常设置为0。

CP/M中每个文件的文件名由两部分构成，第一部分称为文件名 (filename)，文件名最多由8个字符构成，目录项的第1~8字节用来存储文件名。第二部分称为文件类型 (file type)，最多由3个字符表示，这些字符存储在目录项的第9~11字节中。我们会经常遇到一些常见的标准文件类型，例如：TXT表示文本文件（此文件只包含ASCII码）；COM (command的简写) 表示这个文件中存放的是8080机器码指令，或者说是一段程序。当命名一个文件时，通过一个点来隔开这两部分，如下所示：MYLETTER.TXT, CALC.COM。人们习惯形象地称这种文件命名方式为8.3，就是说在点号隔开的前半部分最多有8个字母，后半部分最多有3个字母。

- 接下来介绍一下如何利用目录项查找文件。目录项中的第16~31字节是磁盘存储表，它能够标明文件所存放的分配块。假设磁盘存储表的前4项分别为14h、15h、07h、23h，其余项全为0，这表明文件占用了4个分配块的空间，大小为4 KB，而实际上文件可能并没有用完4 KB空间，因为最后一个分配块往往只有部分扇区被使用。目录项的第15字节表明最后一个分配块到底用到了多少个128字节的扇区。
- 磁盘存储表的长度为16字节，最多可以容纳16,384字节（16 KB）的文件，如果文件的长度超过16 KB，就需要用多个目录项来表示，这种方法称为扩展（extents）。如果一个大文件使用目录项扩展来，则将第一个目录项的第12字节设置为0，第二个目录项的第12字节则设置为1，依此类推。
- 文本文件对我们并不陌生，通常也称之为ASCII文件（ASCII file）、纯文本文件（text-only file）或纯ASCII文件（pure-ASCII file），当然还有其他一些类似的名称。ASCII码（包括换行符和回车符）是文本文件中唯一包含的字符，文本文件通俗易懂，便于浏览。除了文本文件外，其余的文件称为**二进制文件**（binary file），在CM/P中，COM文件存放的是二进制的8080机器码，它是二进制文件。
- 假设一个小文件中包含三个16位数，如：5A48h、78BFh和F510h。如果此文件是二进制文件，只要6字节就可以了。

48 5A BF 78 10 F5

这是采用Intel格式来存储的，放在前面的是低位，放在后面的是高位。并不是所有的数据都是按照这种格式来存储的，例如为Motorola处理器编写的程序更倾向于按以下的方式来组织文件：

5A 48 78 BF F5 10

假如上面的三个16位数用ASCII文本文件来存放，则文件中保存的数据如下所示：

35 41 34 38 68 0D 0A 37 38 42 46 68 0D 0A 46 35 31 30 68 0D 0A  
 上面的这些字节是数字和字母的ASCII码表示形式，用回车符（0Dh）和换行符（0Ah）来表示每一个数的结束。因为文本文件可以不通过解释相应的ASCII字节串来显示字符，而是将字符本身直接显示，所以显得更加方便，如下所示：

5A48h

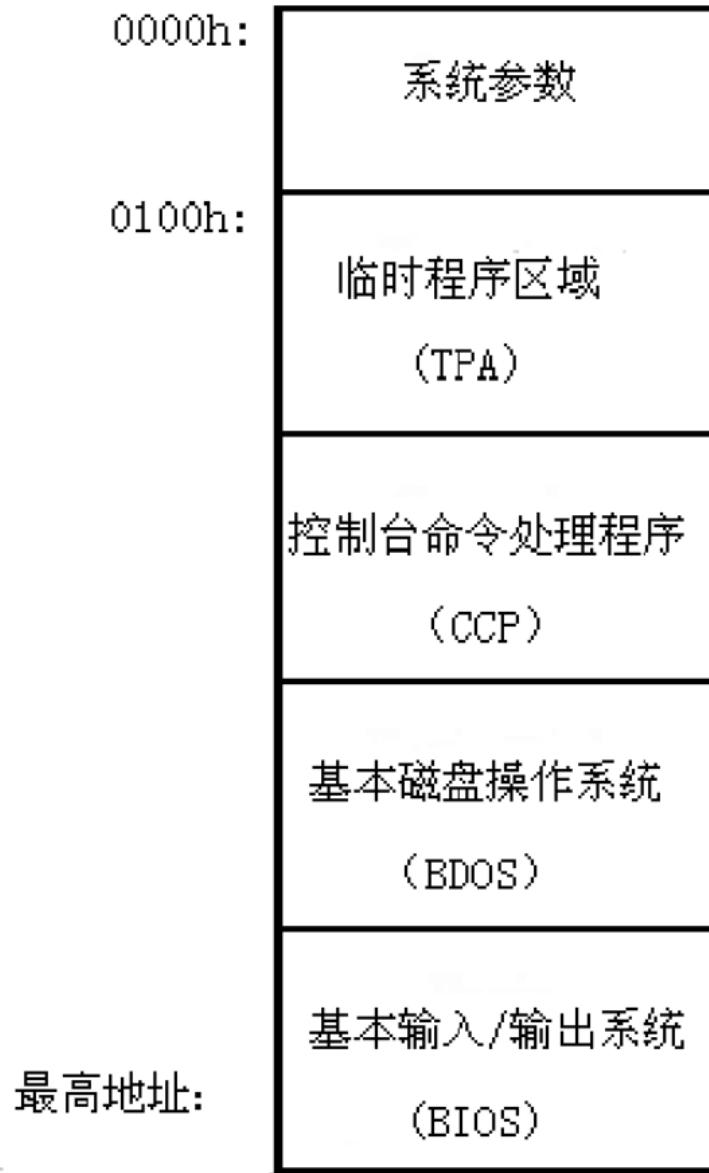
78BFh

F510h

也可以用十进制数的ASCII码形式来表示上述三个数，这两种表示形式是等价的。

- 显然，文本文件更易于人们阅读，同样，与十六进制相比，十进制更符合人们的习惯，没理由使用十六进制而拒绝十进制。
- 磁盘最开始的两个磁道存储CP/M系统本身，而CP/M在磁盘上是无法运行的，必须将其加载到内存里。只读存储器（ROM）在CP/M计算机中使用得并不多，只需要用它来存放一小段称为引导程序（**bootstrap loader**，操作系统的其余部分可以通过这段代码的自举操作被高效地引导）的代码即可。开机启动时，磁盘上最开始的128字节的扇区内容，会首先由引导程序加载到内存并运行，这个扇区包含有特定的代码，可以把CP/M中的其余部分加载到内存中，整个过程称为操作系统的引导（booting）。

- 操作系统的引导过程完成后，随机存储器（RAM）的最高地址区域用来存放CP/M，加载完CP/M后，整个内存空间的组织结构如下所示。



该图仅仅粗略地表示出了内存各构成部分，没有按比例刻画各部分所占内存的大小。**控制台命令处理器 (Console Command Processor, CCP)**、**基本磁盘操作系统 (Basic Disk Operating System, BDOS)** 和**基本输入/输出系统 (Basic Input/Output System, BIOS)** 是CP/M的三个组成部分，这三个部分只占用了6KB大小的内存空间。在拥有64 KB内存空间的计算机中，大约58 KB被**临时程序区 (Transient Program Area, TPA)** 占用，但是这58 KB空间一开始时是空的。

- 控制台命令处理程序的功能和以前讨论过的命令处理程序是一样的。在这里，键盘和显示器组成了**控制台 (console)**。控制台命令处理程序显示如下所示的**命令提示符 (prompt)**：`A>`。在命令提示符后面可以输入一些信息。大多数计算机可能有不止一个磁盘驱动器，第一个磁盘驱动器标为A，CP/M被装载到该驱动器中。在命令提示符后面敲入命令并按回车 (Enter) 键，控制台命令处理程序会执行该命令，然后将执行的结果显示到屏幕上。执行完命令后，命令提示符又会显示在屏幕上，等待下一次输入。
  - 控制台命令处理程序只能识别一部分命令，其中最重要的命令是：`DIR`。该命令用于显示磁盘的目录信息，换句话说，它列出了存储在磁盘上的所有文件。然而有时候需要查看具有特定名字和类型的文件，这时候就可以在命令中使用像“?”和“\*”这样的特殊字符来限定。如果想显示当前目录下所有的文本文件，可以使用如下指令：`DIR *.TXT`, `DIR A???B.*` 则显示所有文件名由5个

字符构成，其中第一个字符是A，最后一个字符是B的文件。

- 如果想删除磁盘中的文件，要用到命令ERA，它是Erase的缩写。运行 ERA \*.TXT 命令，所有的文本文件都被删除。一旦删除文件，此文件的目录项及其所占用的磁盘空间都将被释放。
- REN也是一个常用命令，它是Rename的缩写，此命令可以改变文件名。如果想显示文本文件的内容，可以使用TYPE命令。SAVE命令用来保存文件，它可把临时存储区域中的一个或多个256字节的内存块保存到磁盘中，并且给这个内存块指定一个名字。
- 当然，上述所介绍的都是CP/M可识别的命令，如果你输入一个不能被CP/M识别的命令，CP/M就会默认认为输入的是保存在磁盘上的一个程序名。而程序通常是以文件形式存储的，其文件类型为COM，代表着命令。控制台命令处理程序负责在磁盘上查找此文件，如果找到，此文件会被CP/M从磁盘加载到临时程序区域，该区域的地址从0100h开始，一旦文件被调入内存即可运行。
  - 假如在CP/M命令提示符后面输入：CALC，如果在磁盘中存在名为CALC.COM的文件，则该文件会被控制台命令处理程序调入到以地址0100h开始的内存中，接着控制台命令处理程序会转到该地址并执行这段程序。
- 前面介绍了如何将机器码指令插入到内存空间的任意位置并执行。但是存储在磁盘上的CP/M程序并不能被随意存放到内存中的任意位置，它必须被加载到指定的位置，在这里是以0100h开始的内存空间。
- CP/M由一些实用的程序组成，如PIP（Peripheral Interchange Program），即外设交换程序，通过它可以复制文件。ED是文本编辑器，可以创建和修改文本文件。在CP/M系统中，有很多像PIP和ED一样的程序，它们很小但可以完成简单的事务处理，这类程序称为实用（utility）程序。仅有这些小的简单程序是远远不够的，还有一些大的商业化应用程序（application），比如字处理软件或计算机电子报表软件等，在CP/M系统中，这些软件的使用会给你的工作带来很大方便。当然，如果你是一个程序开发高手，也可以自己动手编写这些软件，这些程序的存储类型都是COM文件类型。
- 在CP/M（跟许多操作系统一样）中，我们了解了很多内容，比如如何利用命令和实用程序对文件进行基本操作，如何将程序加载到内存中并运行等。操作系统的功能远不止如此，下面将介绍它的第三个主要功能。
- 前面提到过，把输出信息写到视频显示器上，或者从键盘读取输入的内容，或者读/写磁盘中的文件，这些都是运行在CP/M下的程序常常要做的操作。这就需要CP/M程序能够直接向视频显示器的内存写入输出内容，也需要CP/M程序能够访问键盘硬件来捕获所输入的内容，还需要CP/M程序能够访问磁盘驱动器来读/写磁盘扇区，然而在通常情况下程序本身是很难直接做到的。
  - 那么有没有别的方法来实现上述的要求呢？答案是肯定的，这些常用事务由CP/M中的子程序集来完成，在CP/M下运行的程序通过调用这些子程序即可完成相应的操作。这些子程序都是专门设计的，计算机中的所有硬件都可以很容易地通过它们来访问，如视频显示器、键盘、磁盘驱动器等，而程序员无须关心这些外设实际是如何连接的。更重要的是，像磁道、扇区这类信息，程序员没有必要知道，这些工作都是由CP/M来完成的，它可以负责读/写磁盘上的文件。
  - 操作系统提供的第三个主要功能是让程序能够方便地访问计算机的硬件，操作系统提供的这种访问操作称为API（Application Programming Interface），即应用程序接口。
- 那么如何设置和使用API呢？在CP/M下运行的程序，可以通过将寄存器C设置为特定的值（称为功能值），并且运行如下指令：CALL 5 来使用API。例如，你从键盘上按下一个键，程序会通过执行下面的指令来获取此键对应的ASCII码：

MVI C, 01h

CALL 5

并且将这个键的ASCII码值保存在累加器A中。

类似的，运行这条命令：

MVI C, 02h

CALL 5

在视频显示器上当前的光标位置将显示累加器A中的ASCII码字符，然后光标移到下一个位置。

如果程序要新建一个文件，它首先将文件名所在区域的地址保存在寄存器对DE中，接着执行如下代码：

MVI C, 16h

CALL 5

执行此命令，CP/M会在磁盘上新建一个空文件。程序可以利用CP/M提供的其他功能来向空文件中写入内容，最后关闭（close）文件，关闭文件意味着文件使用完毕，不需要在对该文件执行任何操作了。当然，该文件可以再次被此程序和其他程序打开（open）并读取内容。

- CALL 5指令到底如何工作呢？CP/M在内存中地址为0005h处设置了一条JMP（Jump）指令，它跳转到CP/M基本磁盘操作系统（Basic Disk Operating System，BDOS）中的某个位置。这个区域包含许多小程序，CP/M的每一项功能都可由它们完成。BDOS，顾名思义，它的主要功能是维护磁盘上的文件系统。文件系统经常要与终端设备打交道，所以BDOS经常要调用CP/M基本输入/输出系统（Basic Input And Output System，BIOS）中的一些子程序。这里顺便提一下，BIOS可以对硬件进行访问，比如键盘、视频显示器和磁盘驱动器等。实际上，BIOS是CP/M中唯一需要了解计算机中硬件的程序，其他一些对硬件的操作都可通过调用BIOS中的子程序来实现。控制台命令处理程序通过调用BDOS的子程序来实现自己所有的功能，而在CP/M中运行的程序也是这样。
- 对于计算机硬件来说，API是一个与设备无关（device-independent）的接口。换言之，对于特定的机器上键盘的工作机制、视频显示器的工作机制以及磁盘扇区的读/写机制，在CP/M下编写的程序不需要知道也没有必要知道。程序使用CP/M提供的功能便可完成对键盘、视频显示器和磁盘驱动器操作，简言之，API屏蔽了硬件之间的差异。有了API，尽管不同计算机硬件差别很大，其访问外设的方式也不尽相同，但CP/M程序都可以在上面运行，从而实现了CP/M程序的跨平台（这里要求所有CP/M程序必须运行在Intel 8080微处理器上，或者运行在能执行Intel 8080指令的处理器上，例如Intel 8085或Zilog Z-80处理器）。所以，在使用CP/M系统的计算机中，程序对硬件访问是通过CP/M来实现的。如果没有标准的API，程序必须根据不同型号的计算机进行相应的修改后，才能访问硬件。
- CP/M是8080中非常流行的操作系统，曾经辉煌一时，至今仍有重要的历史意义。16位操作系统QDOS（Quick and Dirty Operating System）的开发在很多方面都借鉴了CP/M的思想。QDOS当初是专为英特尔公司的16位8086和8088芯片而设计的，它出自西雅图计算机产品公司（Seattle computer product）的提姆·帕特森之手。QDOS最终更名为86-DOS，并被微软公司买断版权。1981年，随着IBM第一代PC诞生，微软公司也将86-DOS更名为MS-DOS（Microsoft Disk Operating System），并以此名授权给IBM公司用作第一代个人计算机的操作系统。这就是著名的MS-DOS系统，在现在的计算机中也会经常看到它的身影。虽然16位版本的CP/M（称为CP/M-86）也可用于IBM PC，但由于MS-DOS的影响力更大，其很快成为了标准。其他生产IBM PC兼容机的厂商也被授权使用MS-DOS（在IBM计算机上称为PC-DOS）系统。

- CP/M的文件系统在MS-DOS没有被继续使用，在MS-DOS中，文件系统是以**文件分配表（FAT, File Allocation Table的简写）**的形式来组织的，Microsoft公司早在1977年就开始使用FAT了。FAT的基本思想是：将磁盘空间分成**簇（cluster）**——簇的大小由磁盘空间的大小来决定——从512字节到16 K字节不等，每个文件占用若干簇。文件的目录项只记录文件起始（starting）簇的位置，而磁盘上每一簇的下簇的位置由FAT来记录。
  - 每个目录项在MS-DOS磁盘上占用32字节，其命名形式跟CP/M上的8.3形式一样，只是使用的术语有些区别：最后的三个字符称做文件扩展名，而非CP/M中的文件类型。MS-DOS目录项中没有包含分配块列表，它主要包含如下所示的有用信息：文件的最后修改的日期、时间和文件的大小等。
  - 实际上，MS-DOS的早期版本与CP/M在结构上很类似，只是IBM PC本身在ROM中已经包含了一套完整的BIOS了，所以在MS-DOS中不再需要BIOS。在MS-DOS中，命令处理器是一个命名为COMMAND.COM的文件。MS-DOS有两种运行程序：一是以COM为扩展名的文件，其大小不能超过64 KB，二是更大一些的程序，以EXE（意思是“可以被执行”）为文件的扩展名，表明文件本身是可执行的。
- 尽管MS-DOS起初支持API函数的CALL 5接口，但不久Microsoft为新的程序重新设计了一款新的接口。这个新的接口使用了8086的一个称为软件中断（software interrupt）的功能，说起软件中断，它与子程序调用很类似，只不过程序不必知道它所调用的子程序的确切地址。通过执行INT 21h这条指令，程序可以调用MS-DOS的API功能。
  - 从理论上讲，应用程序并不能直接访问计算机的硬件，如果它要访问计算机的硬件，可以通过操作系统提供的接口来进行。然而实际上，在20世纪70年代末和80年代初，由于计算机上运行的都是小型操作系统，许多应用程序往往都绕过它们，这种情况在处理有关视频显示器任务的时候显得特别明显。为什么会这样呢？因为如果程序把字节直接写入视频显示存储器，它的执行速度要远快于不直接写入视频显示存储器。事实上，对于一些程序——比如要将图形显示在视频显示器上——操作系统就显得“心有余而力不足”，这就需要在操作系统之外另作处理。也许正是因为MS-DOS系统卓越的“反传统性”，使得大多数程序员都很热衷于它，它可以让程序员编写的程序最大限度地达到硬件的最快速度，更加充分地发挥硬件的性能。
  - 恰恰是这个原因，运行在IBM PC上的流行软件通常依赖于IBM PC这个硬件平台，换句话说，就是这些软件是根据IBM PC的硬件特点编制的。为了和IBM PC竞争，其他机器制造商不得不沿袭这些特点。如果不这样做的话，再流行的软件也将流行不起来，因为它不能在这些机器上运行。所以在软件的显著位置往往会出现“与IBM PC百分之百兼容”的字样，这同时体现了软件对硬件的要求。
- 微软于1983年3月发布了MS-DOS 2.0版本，与最开始的版本相比，它加强了对硬盘驱动器的管理。虽然当时的硬盘容量很小（按今天的标准），但是发展很快，没过多久，硬盘的容量变得越来越大。这带来了不少问题，硬盘容量越大存储的文件也就越多，存储的文件越多，当然，查找某个指定的文件或组织文件也就越困难。
  - 为了解决上述问题，MS-DOS 2.0引入了层次文件系统（hierarchical file system），它只是在原有MS-DOS的文件系统上做了一些小的改动。前面介绍过，目录存储在磁盘中特定区域，它是一个文件列表，包含了文件存储在磁盘位置的信息。在层次文件系统中，有些文件其本身可能就是目录，也就是说这些文件包含其他文件，其中的一些可能还是目录。在磁盘中，目录的称法也是有讲究的，常规的目录称为**根目录（root directory）**，**子目录（subdirectories）**是包含在其他目录里的目录。有了目录（有时称文件夹，folder），就可以很方便地对相关文件进行分组，所以目录在磁盘文件的管理中起着非常重要的作用。

- 讲到操作系统，我们不能不提到著名的UNIX系统，它在操作系统的发展史上有着举足轻重的地位。实际上，层次文件系统和MS-DOS 2.0的其他很多功能都是从UNIX操作系统借鉴而来的。UNIX操作系统是20世纪70年代初在贝尔实验室开发的，肯·汤普森（Ken Thompson，生于1943年）和丹尼斯·里奇（Dennis Ritchie，生于1941年）是此系统的主要开发者。UNIX系统（包括名字本身）的发展历程在这里要提一下，UNIX系统来源于早期的Multics（Multiplex Information and Computing Services，表示多路复用信息和计算业务），Multics是贝尔实验室和MIT（麻省理工大学）和GE（通用电气公司）合作开发的一个项目，开发初期由于Multics没能达到预定的目的而且进度缓慢，加之昂贵的开发代价，1969年贝尔实验室退出了该项目，但肯·汤普森和丹尼斯·里奇继续对此进行了开发，为了区别于Multics，取名为UNIX。
  - 对于计算机核心程序开发的精英们来说，UNIX无疑是最受欢迎的操作系统。以前大部分操作系统是针对特定的计算机硬件平台开发的，但UNIX打破了常规，它不针对具体的计算机硬件平台，具有很好的可移植性（portable），也就意味着它在各种机器上都可以运行。
  - 在开发UNIX的时候，贝尔实验室是电信业巨头AT&T（American Telephone & Telegraph，美国电话电报公司）旗下的一员，为了限制AT&T在电话业务的垄断地位，法院裁定禁止AT&T销售UNIX系统，AT&T被迫将它授权给别人。因此从1973年初，很多大学、公司和政府机构被授权使用和研究UNIX，这在一定程度上促进了UNIX的发展。1983年，AT&T获准重返计算机业务并发布了它自己的UNIX版本。
  - 由于UNIX的广泛授权导致了许多不同版本共存的情况，它们使用着不同的名字，运行在不同的计算机上，并由不同的经销商销售。由于UNIX的影响力和开源性，使得很多人将精力投入到UNIX中并推动着它的不断发展。当人们向UNIX中添加新的组件时，无形之中流行着一种不成文的“UNIX思想”。在这种思想的指导下，人们都使用文本文件作为公用文件形式。许多UNIX实用程序读取文本文件，利用它提供的一些功能做些处理，然后将其写入另一个文本文件。因此可以用链的形式来组织UNIX实用程序，然后对这些文本文件做相应的处理。
  - 在20世纪60-70年代，计算机不仅体积庞大而且价格昂贵，仅仅为一个人服务显然不太现实，为了能够让多个人同时使用计算机，必须有相应操作系统来支持，这也就是开发UNIX系统的最初目的。使用UNIX系统的计算机通过时分复用（time sharing）技术——这种技术允许多个用户同时与计算机进行交互——来达到这个目的。计算机连接多个配备了显示器和键盘的终端（terminals），每个用户通过这些终端访问计算机。通过在所有终端间的快速切换，使用户感觉这台计算机似乎只为自己工作，而其实计算机同时在为多个用户提供服务。
- 如果在一个操作系统上可以同时运行多个程序，则称此系统为多任务（multitasking）操作系统。显然，与CP/M和MS-DOS这样的单任务的系统相比，这种操作系统要复杂得多。正是由于支持多任务这种功能，文件系统变得很复杂，因为同一个文件可能被多个用户同时访问。程序的运行需要占用内存空间，多任务系统就要考虑内存的分配问题，也就是说需要进行内存管理（memory management）。也许你会有疑问，多道程序并行运行需要占用大量内存，如果内存不够怎么办？为此，操作系统引入了虚拟内存（virtual memory）技术。虚拟内存是指，在磁盘上划出部分空间用做保存临时文件，程序把暂时不需要用的内存块放到临时文件里，待需要时再把它调入内存。
- UNIX能够存在并发展到现在是无数人共同努力的结晶，如今FSF（Free Software Foundation，自由软件基金会）和GNU项目为推动UNIX的发展注入了新的活力，它们都是由理查德·斯托曼（Richard Stallman）创建的。GNU意味着：“GNU与UNIX，既要划清界限又相辅相成”。GNU项目的宗旨是：创建一个与UNIX系统兼容，但不受私有权限制的操作系统和开发环境。在这个项目的推动下，涌现出了许多和UNIX兼容的实用程序和工具，其中最著名的要算Linux系统。Linux系统的内核（Core与Kernel都可以表示内核的意思）和UNIX的是完全兼容的，它的大部分程序是由芬兰的李纳斯·托沃兹（Linus Torvalds）完成的。由于Linux系统的开源性，近年来已成为非常流行的操作系统。

- 从20世纪80年代中期开始，开发大型的、复杂度更高的系统成为了操作系统发展的一大趋势，例如苹果公司的Macintosh系统和微软的Windows系统，它们融合了图形和高级可视化视频显示技术，使应用程序变得更容易使用。关于图形和可视化方面的发展趋势，我们将在本书的最后一章进一步介绍。

## 23 定点数和浮点数

- 整数、分数以及百分数等各种类型的数字与我们形影不离，它几乎出现在我们生活的所有角落。例如你加班2.75小时，而公司按正常工作时间的1.5倍支付你工资，你用这些钱买了半盒鸡蛋并交了8.25%的销售税。就算你不是数字研究方面的专家，也可能对这种“数字生活”非常熟悉。我们还经常听到类似这样的统计信息“美国每个家庭的平均人口是2.6人”，但谁也不会为了满足这个数字表达的真实含义而把人拆解掉（这种事情想起来都觉得恐怖）。
- 在计算机存储器中，整数和分数之间的转换并不是这么随意。现在我们应该清楚，计算机中的一切数据都是以位的形式存储的，这就意味着所有的数都表示为二进制形式。但另一个不可否认的事实是，某些类型的数比其他类型更容易用位的形式来表示。
- 我们将从整数的二进制表示开始，这里的整数被数学家称做“自然数”（positive whole numbers），即计算机程序员口中的“正整数”（positive integers），之后将介绍如何利用2的补数来表示“负整数”（negative integers），该方法可以让正数和负数的相加变得非常简单。下面的表格列出了8位、16位、32位二进制数所能表示的正整数及其2的补数的范围。

数的位数	正整数的范围	整数的2的补数范围
8	0 ~ 255	-128 ~ 127
16	0 ~ 65,535	-32,768 ~ 32,767
32	0 ~ 4,294,967,295	-2,147,483,648 ~ 2,147,483,647

我们所要介绍的整数部分就是这些。除此之外，数学家还定义了用两个整数的比值表示的一类数，称做有理数（rational number）或分数（fraction）。例如， $\frac{3}{4}$ 是一个有理数，因为它是整数3和4的比。我们也可以把 $\frac{3}{4}$ 表示成十进制小数的形式，即0.75。尽管我们可以把它写成十进制数的形式，但它实际上代表一个分数，即 $75/100$ 。

- 除此之外，数学家还定义了用两个整数的比值表示的一类数，称做有理数（rational number）或分数（fraction）。有一些有理数很难表示成小数，比如 $\frac{1}{3}$ ,  $\frac{1}{7}$ 。通过在第一个3的上面加一条短横线，可以将这个无限循环的数简单表示。
- 无理数（irrational number）是一些更加奇特的数，如2的平方根等。它们不能表示为两个整数的比，这就意味着其小数部分是无穷的，而且毫无规律，没有循环。
- 如果某个数不是任何以整数为系数的代数方程的解，那么这个数称做超越数（transcendental，所有的超越数都是无理数，但是反之不成立）。 $\pi$ 就是一个典型的超越数，它是圆的周长与其直径的比值。 $e$ 是另一个典型的超越数，它是数学表达式： $(1 + \frac{1}{n})^n$ 当n趋向无穷大时的值。
- 目前我们所讨论过的所有数——有理数和无理数——统称为实数（real numbers）。使用实数定义他们的目的是为了将其与虚数（imaginary numbers）区别开来，虚数是负数的平方根。实数和虚数一起构成了复数（complex numbers）。不管名称如何，它们都有重要的作用，例如，虚数确实存在于现实世界，它在解决电子学的某些高级问题中有重要应用。
- 我们习惯于把数字看做连续（continuous）的，任意给出两个有理数，都可以找出一个位于它们之间的数。实际上，只需要取这两个数的平均值即可。但是，数字计算机对连续数据却无能为力，因为二进制中的每一位非0即1，两者之间没有任何数。这一特点决定了数字计算机只能处理离散（discrete）数据。二进制数的位数直接决定了所能表示的离散数值的个数。例如，如果你选择的二进制位数是32，则

所能表示的自然数的范围是0 ~ 4,294,967,295。如果想要在计算机中存储4.5这个数，则需要选择新的方法并做一些其他方面的改进。

- 小数也可以表示为二进制数吗？当然可以，最简单的方法可能就是使用BCD码（Binary-Coded Decimal）。如第19章所述，BCD码是将十进制数以二进制的形式进行编码。BCD编码在程序处理用美元和美分表示的钱款、账户时特别有用。银行和保险公司是非常典型的两类整日与钱打交道的机构，这些机构所使用的计算机程序中，大多数小数所占用的存储空间仅仅相当于两个十进制数所占用的位数。
- 通常把两个BCD数字存放在一个字节，这种方式称为压缩BCD（packed BCD）。由于2的补数不和BCD数一起使用，因此压缩BCD通常需要增加1位用来标识数的正负，该位被称做符号位（sign bit）。用整个字节保存某个特定的BCD数是很方便的，但要为这个短小的符号位牺牲4位或8位的存储空间。
  - 4,325,120.25可以表示为下面5个字节：00010100 00110010 01010001 00100000 00100101。注意，最左边的半个字节所构成的1用来指明该数是负数，这个1即符号位。如果这半个字节所构成的数是0，则说明该数是正数。组成该数的每一个数字都需要用4位来表示。
- 这种基于二进制的存储和标记方式也被称作定点格式（fixed-point format），所谓的“定点”是指小数点的位置总是在数的某个特定位置——在本例中，它位于两位小数之前。值得注意的是，有关小数点位置的计数信息并没有与整个数字一起存储。所以，使用定点小数的程序必须知道小数点的位置。你可以设计有任意小数位的定点小数，并且可以在程序中混合使用它们，但程序中对这些数进行算术运算的部分都需要知道小数点的位置，这样才能正确地对其做各种运算处理。
- 如果可以确定程序用到的数字不会大到超过预定的存储空间，并且这些数的小数位不会很多，那么使用定点格式的小数将是一个很好的选择。在表示非常大或非常小的数时，使用定点格式数是绝对不合适的。假设需要保留一块内存空间用来存放以英尺为单位的距离数据，可能存在的问题是某些距离的长度可能超出范围。地球与太阳之间的距离是490,000,000,000英尺，而氢原子的半径只有0.000,000,000,26英尺，如果采用定点格式的存储方案，为了存储这些极大或极小的数需要12个字节。
- 科学家和工程师们喜欢使用一种称为“科学计数法”（scientific notation）的方法来记录这类较大或较小的数，利用这种计数系统可以更好地在计算机中存储这些数。科学计数法把每个数表示成有效位与10的幂的乘积的形式，这样就可以避免写一长串的0，因此这种计数方式特别适合表示极大或极小的数。
- 科学计数法第一部分被称做小数部分或者首数（characteristic），有时候也被称作尾数（mantissa，这个词通常与对数运算一起使用）。在计算机术语中这一部分被称做有效数（ significand），因此为了保持一致，这里把科学计数法表示形式中的这一部分也称作有效数。为了便于操作，一般规定有效数的取值范围是大于或等于1而小于10。这种写法有时被称做科学计数法的规范化式（normalized）。
- 采用科学计数法表示的数可以分为两部分，其中指数（exponent）部分用来表示10的几次幂。指数可以表明小数点相对于有效数移动的距离。这里需要说明，指数的正负性只是表明了数的大小，它并不能指明数本身的正负性。
- 在计算机中，对于小数的存储方式，除了定点格式外还有另外一种选择，它被称做浮点格式（floating-point notation）。因为浮点格式是基于科学计数法的，所以它是存储极大或极小数的理想方式。但计算机中的浮点格式是借助二进制数实现的科学计数法形式，因此我们首先要了解如何用二进制表示小数。
- 实际操作起来比预想的要简单。在十进制数中，小数点右边的数字与10的负整数次幂相关联；而在二进制数中，二进制小数点（就是一个简单的句点，看起来同十进制小数点一样）右边的数字和2的负整数次幂相关。例如，下面这个二进制数：101.1101可以用如下方式转换为十进制数：

$$\begin{aligned}
& 1 \times 2^2 + \\
& 0 \times 2^1 + \\
& 1 \times 2^0 + \\
& 1 \times 2^{-1} + \\
& 1 \times 2^{-2} + \\
& 0 \times 2^{-3} + \\
& 1 \times 2^{-4}
\end{aligned}$$

经过这种计算，101.1101与十进制数5.8125是相等的。

- 在十进制的科学计数法中，规范化式的有效数应该大于或等于1且小于10；类似的，在二进制的科学计数法中，规范化式的有效数应该大于或等于1且小于10（即十进制的2）。因此，在二进制的科学计数法中，下面这个数字：101.1101其规范化式应该是：

$$1.011101 \times 2^2$$

这个规则暗示了这样一个有趣的现象：在规范化二进制浮点数中，小数点的左边通常只有一个1，除此之外没有其他数字。

- 当代大部分计算机和计算机程序在处理浮点数时所遵循的标准是由IEEE (Institute of Electrical and Electronics Engineers, 美国电气和电子工程师协会) 于1985年制定的，ANSI (American National Standards Institute, 美国国家标准局) 也认可该标准。ANSI/IEEE Std 754-1985称作IEEE二进制浮点数算术运算标准 (IEEE Standard for Binary Floating-Point Arithmetic) ——它只有18页——相对于其他标准来说是非常简短了，但却奠定了以简便方式编码二进制浮点数的基石。
- IEEE浮点数标准定义了两种基本的格式：以4个字节表示的单精度格式和以8个字节表示的双精度格式。
- 让我们首先来了解一下单精度格式。它的4个字节可以分为三个部分：1位的符号位（0代表正数，1代表负数），8位用做指数，最后的23位用做有效数。下表给出了单精度格式的三部分的划分方式，其中有效数的最低位在最右边。

$s = 1$ 位符号	$e = 8$ 位指数	$f = 23$ 位有效数
-------------	-------------	---------------

三部分共32位，也就是4个字节。我们刚才提到过，对于二进制科学计数法的规范化式，其有效数的小数点左边有且仅有一个1，因此在IEEE浮点数标准中，这一位没有分配存储空间。在该标准中，仅存储有效数的23位小数部分，尽管存储的只有23位，但仍然称其精度为24位。我们将在下面的内容里体会24位精度的含义。

8位指数部分的取值范围是0~255，称为偏移 (biased) 指数，它的意思是：对于有符号指数，为了确定其实际所代表的值必须从指数中减去一个值——称做偏移量 (bias)。对于单精度浮点数，其偏移量为127。

指数0和255用于特殊的目的，稍后将简单介绍。如果指数的取值范围是1~254，那么对于一个特定的数，可以用s (符号位)，e (指数) 以及f (有效数) 来描述它：

$$(-1)^s \times 1.f \times 2^{e-127}$$

-1的s次幂是数学上所采用的一种巧妙的方法，它的含义是：如果s = 0，则该数是正的（因为任何数的0次幂都是1）；如果s = 1，则该数是负的（因为-1的1次幂等于-1）。

1的后面是小数点，小数点后面跟着23位的有效数。与2的幂相乘，其中指数等于内存中的8位的偏移指数减去127。

- 注意，目前为止我们还没有学习如何表达那个经常遇到却又总被遗忘的一个数字：那就是“0”。这是一种特殊的情况，下面我们对其进行说明。

- 如果 $e=0$ 且 $f=0$ ，则该数为0。在这种情况下，通常把32位都设置为0以表示该数为0。但是符号位可以设置为1，这种数可以解释为负0。负0可以用来表示非常小的数，这些数极小以至于不能在单精度格式下用数字和指数来表示，但它们仍然小于0。
- 如果 $e=0$ 且 $f\neq 0$ ，则该数是合法的，但不是规范化的。这类数可以表示为：

$$(-1)^s \times 0.f \times 2^{-127}$$

- 如果 $e=255$ 且 $f=0$ ，则该数被解释为无穷大或无穷小，这取决于符号位s的值。
- 如果 $e=255$ 且 $f\neq 0$ ，则该值被解释为“不是一个数”，通常被缩写为NaN (nota number)。NaN用来表示未知的数或非法操作的结果。

- 单精度浮点格式下，可以表示的规格化的最小正、负二进制数是：

$$1.000000000000000000000000_2 \times 2^{-126}$$

小数点后面跟着23个二进制0。单精度浮点格式下，可以表示的规格化的最大正、负二进制数是：

$$1.111111111111111111111111_2 \times 2^{127}$$

在十进制下，这两个数近似地等于 $1.175494351 * 10^{-38}$ 和 $3.402823466 * 10^{38}$ ，这也就是单精度浮点数的有效表示范围。

10位二进制数可以近似地用3位十进制数来表示。其含义是，如果把10位都置为1，即十六进制的3FFh或十进制的1023，它近似等于把十进制数的3位都置为9，即999，可以表示为下面的约等式：

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

两者之间的这种关系意味着：单精度浮点数格式存放的24位二进制数大体上与7位的十进制数相等。因此，可以说单精度浮点格式提供24位的二进制精度或者7位的十进制精度。其深层的含义是什么呢？

当我们查看定点数时，其精确度是很明显的。例如，当我们表示钱款时，采用两位定点小数就可以精确到美分。但是对于采用浮点格式的数，就不能如此肯定了。其精确度依赖于指数的值，有时候浮点数可以精确到比美分还小的单位，但有时候其精确度甚至达不到美元。

这样说可能更合适：单精度浮点数的精度为 $1/2^{24}$ ，或 $1/16777216$ ，或百万分之六，但其真正的含义是什么呢？

首先，这意味着在单精度浮点格式下，16,777,216和16,777,217将表示成同一个数。不仅如此，处于这两个数之间的所有的数（例如，16,777,216.5）也将被表示成同一个数。上面提到的3个十进制数都按32位单精度浮点数：4B800000h来存放。将该数按符号位、指数位和有效数位划分，可以表示为：

$$0\ 10010111\ 00000000000000000000000000000000$$

也就是说：

$$1.000000000000000000000000000000_2 \times 2^{24}$$

下一个二进制浮点数可表示的最大有效数是16,777,218，即：

$$1.00000000000000000000000000000001_2 \times 2^{24}$$

以同一个浮点数来表示两个不同的十进制数，有时可能成为一个问题，也可能不会。但如果你为银行编写程序，用单精度浮点数来存放以美元、美分为单位的数字时，就会发现262144.00美元和262144.01美元在计算机中存储为同一个数：

$$1.00000000000000000000_2 \times 2^{18}$$

这也是为什么人们在处理以美元、美分表示的钱款数目时更愿意使用定点数的一个原因。当使用浮点数时，你会发现它还存在着一些让人崩溃的小问题。你的程序进行了一系列计算，应该得到的结果为3.50的，但由于使用浮点数，你得到的可能是3.499999999999。这种问题在浮点数运算中经常发生，而且没有一套完整的解决方案。

- 如果想在程序中使用浮点格式数，但使用单精度格式又会出现各种问题，这时你可以考虑使用双精度浮点数（double-precision floating-point format）。这种类型的数需要用8个字节来表示，它的结构如下表所示。

$s = 1$  位符号位     $e = 11$  位指数位     $f = 52$  位有效数

双精度浮点数的指数偏移量是1023，或十六进制的3FFh，因此以该格式存储的数可以表示为：

$$(-1)^s \times 1.f \times 2^{e-1023}$$

上面提到的关于单精度浮点格式下的0, 无穷大(小)和NaN的判断规则同样适用于双精度浮点格式。

双精度浮点格式下可以表示的最小正数或负数为：

注意，小数点的后面共有52个0。同样的，可以表示的最大数为：

其所能表示的范围，用十进制可以近似记为：

$$2.2250738585072014 \times 10^{-308} \approx 1.7976931348623158 \times 10^{308}$$

10的308次幂是一个非常巨大的数，在1的后面跟着308个0。

- 双精度浮点格式的有效数有53位（包括前面没有列出的那一位），大致相当于十进制的16位。与单精度浮点格式相比，这已经有了很大的改进了，但仍然不能避免两个不同的数存储为同一个结果的情况。例如，140,737,488,355,328.00和140,737,488,355,328.01在内存中存放时，会被当做同一个数来处理，它们的双精度浮点格式表示为：42E0000000000000h，即：

- 当然，为浮点数发明一种在内存中的存储方式，这只是在汇编程序使用浮点数所涉及的工作的一小部分。如果你决定闭门造车，完全独立地开发一台计算机，则必须要独立编写用于浮点数加、减、乘、除运算的函数集。幸运的是，有了前面关于整数四则运算的学习，这些关于浮点数的运算就可以分解成许多小的关于整数的加、减、乘、除运算，这样就能将问题大大简化。
    - 例如，浮点数加法中最重要一点的就是如何对有效数相加，为了能使它们的有效位匹配，需要利用指数来确定对其如何移位。

- 两个浮点数的乘法意味着要把有效数当作整数相乘，并且把指数部分相加。为了使结果规范化，一般需要对指数调整一到两次。
- 浮点数运算另一层次的复杂性体现在处理一些较为繁杂的函数运算，例如平方根、指数、对数和三角函数。但所有的这些运算都可以通过加、减、乘、除这四种基本的浮点数运算来实现。
- 例如，三角函数中的sin函数可以通过下面的一系列展开式近似计算：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

参数x的值必须是弧度，360°对应的弧度范围是 $2\pi$ 。

上面的算式中，唯一让人感到棘手的地方是最后的省略部分，这意味着计算会一直继续下去。然而事情并没有想象中的那么糟糕，在实际运算中，如果把弧度的取值限制在 $0\sim\pi/2$ 的范围内（从这个范围就可以推导出所有的正弦值），你根本不需要进行多少运算，因为大约展开12项后，就可以使结果精确到双精度浮点数要求的53位。

- 当然，使用计算机的目的就是帮助人们更加方便地解决问题，而编写程序来进行浮点数运算这一繁杂工作似乎和这个目的背道而驰。但这正是软件的优势所在：一旦某个人为特定的计算机编写了浮点数运算的程序，那么其他的人都可以使用它。浮点数运算在科学和工程类程序中极为重要，因此常常被赋予很高的优先级。在计算机发展的早期，为新制造的计算机做的第一项工作就是为其编写浮点数运算程序。
- 实际上，甚至可以直接利用计算机机器码指令来实现浮点数的运算。当然，实际做起来要比“动动嘴皮子”困难得多，但这也从另一个方面说明了浮点数运算的重要性。如果可以在硬件上实现浮点数算术运算——类似在16位微处理器上进行乘法和除法运算——则该机器上的所有的浮点数运算都会变得更快。
  - IBM公司在1954年发布了IBM 704，它是第一台将浮点数运算硬件作为可选配件的商用计算机，该机器以36位空间来存储所有的数。对于浮点数而言，其36位被分成27位的有效数，8位的指数和1位的符号位。浮点运算硬件可以直接进行加法、减法、乘法和除法运算。其他的浮点运算则必须通过软件来实现。
  - 从1980年开始，浮点运算硬件开始应用于桌面计算机，这起始于英特尔当年发布的8087数字协同处理（Numeric Data Coprocessor）芯片，当时这种集成电路被称做数学协同处理器（math coprocessor）或浮点运算单元（floating-point unit, FPU）。8087不能独立工作，它只能与8086或8088（Intel的首个16位微处理器）芯片一起工作，因此被称做协处理器。
  - 8087拥有40个管脚，它使用的很多信号与8086或8088完全相同。微处理器和数学协处理器通过这些信号相连。当CPU读取到一条特殊指令ESC（Escape）时——协处理器开始接管控制权并执行下一条机器指令，该指令可以是三角函数、指数和对数等68条指令中的任一条。它所处理的数据类型遵循IEEE标准。在当时，8087被认为是市面上最高水平的集成电路。
  - 可以把协处理器当做一个小型的自包含计算机。当响应某个特定的浮点运算机器码指令时（例如FSQRT指令，它用来计算平方根），协处理器会以固有的方式执行存放在ROM中属于自己的指令序列。这些内部指令称做微代码（microcode）。通常，这些指令都是循环的，因此不能立即得到最终的结果。虽然如此，但数学协处理器在运算速度方面仍然表现优异，与软件方法相比，其速度至少是后者的10倍。
  - 在最初版本的IBM PC主板上，位于8080芯片的右边有1个40个管脚的插槽供8087芯片接入。但令人失望的是，这个插槽是空的，用户如果需要进行浮点数运算就必须单独购置一块8087芯片，并将其插入主板后才能使用。安装数学协处理器并不能提高所有应用程序的运行速度，因为有些应用程序——比如文字处理程序——几乎用不到浮点数运算。其他应用程序，比如电子表格处理程

序，对浮点数运算依赖程度很高，在安装了数学协处理器之后，它们的执行速度有很大的提高，但并非所有的程序都是如此。

- 可以看到，在安装了数学协处理器后，程序员必须使用协处理器机器码指令来编写特殊的代码，因为数学协处理器不是标准硬件，因此它只能执行这些特殊的代码。而这些工作让程序员烦不胜烦。尽管他们不愿意，他们仍不得不编写自己的浮点数运算子程序（因为大多数人并没有安装数学协处理器），因此这就多了一个额外的工作——一个并不轻松的工作——在程序中支持8087芯片。最后就出现了这样的局面：如果机器上安装了数学协处理器，程序员就要学会编写相应的应用程序以支持它的运行；如果没有安装，程序员就要通过编程来模拟它进行浮点数的运算。
- 在随后的几年内，英特尔还发布了与286配合工作的287数学协处理器，与386配合工作的387数学协处理器。但是，在1989年发布的486DX芯片中，FPU已经内建在CPU的结构里，它不再作为一个配件供选择安装了。令人失望的是，在1991年发布的一款低端芯片486SX中，英特尔没有为该其内建FPU，而是提供了一块可选的487SX数学协处理芯片。但1993年发布的奔腾芯片中，CPU内置FPU再次成为标准，也许这是永远的标准。在1990年发布的68040芯片中，摩托罗拉首次将FPU集成在CPU中，在此之前，摩托罗拉发布了68881和68882数学协处理器来支持68000家族早期的微处理器。PowerPC芯片同样使用了内置浮点数运算硬件的技术。
- 浮点数运算硬件对于困惑的汇编程序员来说无疑是个惊喜的礼物，但相对于20世纪50年代开始的某些其他工作来言，这只是历史所迈出的一小步。接下来，我们的探索之旅即将到达下一站：计算机语言。

## 24 高级语言与低级语言

---

- 使用机器码编写程序就如同用牙签吃东西，伸出手臂使出较大的力气刺向食物，但每次都只获取到小小的一块，这个过程是辛苦且漫长的。同样的，每个机器码字节所能完成的工作，是你能想象到的最微小且最简单的工作——从内存获取一个数，之后加载到处理器，再把它与另一个数相加，最终将运算结果保存到内存等——正因如此，很难想象如何使用这些机器码构成一个完整的程序。
- 目前为止，至少对于在第22章讨论的原始模型阶段来说，我们已经取得了一定的进步，在那个阶段，我们使用过控制面板上的开关将二进制数据输入到存储器。在第22章中，介绍了如何编写一段简单的程序，让我们可以利用键盘将十六进制机器码输入计算机，以及通过视频显示设备来检查这些代码。这种改进固然可取，但仍不是我们的终极目标。
- 前面的章节介绍过，可以使用某些较短的助记符来关联机器码字节，这些助记符包括MOV, ADD, CALL, HLT等，通过这些类似英语的符号我们可以较方便地引用机器码。通常这些助记符的后面会跟着操作数，这可以进一步指明它所关联的机器码指令的功能。例如8080机器码字节46h，它的功能是令处理器将存储在内存特定地址的字节转移至寄存器B，而该地址由寄存器对HL中的16位数寻址。这个操作可以简单地写做：`MOV B, [HL]`

显然，使用汇编语言编写程序要比使用机器语言简单得多，但微处理器并不能解释汇编语言。在前面的章节中我们已经学习了如何在纸上编写汇编程序，但只有当你确实准备在微处理器上运行汇编程序，才会手工对其汇编，这样就可以将汇编语言程序的语句转换成了机器语言代码，并把它们输入内存。

当然，我们希望最好由计算机能独自完成语言转换的工作。如果你的8080计算机正在运行CP/M操作系统，而且你已经拥有了所有必需的工具，那就再好不过了，因为下面我们将介绍其工作原理。

- 第一步，建立一个文本文件，并将汇编语言程序输入到该文本文件中。这项工作可以使用CP/M的应用程序ED.COM来完成。该程序是一个可以用来创建、修改文本文件的编辑器。假设你把该文本文件命名为PROGRAM1.ASM，其中ASM是文件类型，用来指明该文本文件的内容是由汇编语言程序组成。这个文件的内容如下：

```
ORG 0100h
LXI DE, Text
MVI C, 9
CALL 5
RET
Text: DB 'Hello!$'
END
```

这个文件中有两条语句我们从未接触过。第一条语句是ORG (origin) , 它不与任何8080指令对应，其功能是用来指明下面语句的地址从0100h地址处开始。如前所述，该地址是CP/M将程序装入内存的起始地址。

第二条语句是LXI (Load Extended Immediate) 指令，其功能是将一个16位数加载到寄存器对DE。在本例中，该16位数是由标记Text提供的。该标记在程序底端的附近，位于DB (Data Byte) 语句之前。DB语句我们也是第一次遇到，其后可以跟着一些字节，这些字节以逗号分隔或者用单引号括起来（如本例）。

MVI (Move Immediate) 语句将数值9转移到寄存器C。CALL 5语句实现CP/M的函数调用功能。函数5的作用是：显示以寄存器对DE给出的地址为起始处的字符串，直到遇到\$结束（可以看到，在程序的结尾处使用了美元符号"\$"作为文本的结束标志，这种方式看起来很奇怪，但CP/M就是采取的这种方式）。最后的RET语句用来结束程序，并把控制权交还给CP/M（实际上，这只是结束CP/M程序的方法之一）。END语句用来指明汇编语言文件已经结束。

- 现在我们已经有了一个包含7行语句的文本文件，下一步要做的就是对其进行汇编，即将其转换成机器语言代码。以前这项工作是通过手工完成的，但现在我们的机器运行的是CP/M系统，可以利用CP/M中一个叫做ASM.COM的模块来完成这项工作。该模块是CP/M的汇编器 (assembler) 。可以在CP/M的命令行中使用下面的语句运行ASM.COM文件：`ASM PROGRAM1.ASM`。ASM对PROGRAM1.ASM文件进行汇编，产生一个名为PROGRAM1.COM的新文件，PROGRAM1.COM包含了与我们编写的汇编程序相对应的机器码（实际上，该过程还包含另一个步骤，但在该操作中并不重要）。现在就可以使用CP/M的命令行来运行PROGRAM1.COM文件，程序运行的结果是显示字符串“Hello!”然后结束。

PROGRAM1.COM文件包含以下16个字节：

11 09 01 0E 09 CD 05 00 C9 48 65 6C 6C 6F 21 24

开始的3个字节是LXI指令，其后的两个字节是MVI指令，接下来的三个字节是CALL指令，紧随其后的一个字节是RET指令，最后的7个字节是ASCII码，包括5个字母“Hello”，感叹号“！”以及美元符号“\$”。

- 像ASM.COM这样的汇编器程序所做的工作是：读取一个汇编语言文件（source-code，通常称做源代码文件），将其转换得到一个包含机器码的文件——可执行文件（executable file）。从宏观的角度来看，汇编器是非常简单的，因为构成汇编语言的助记符和机器码之间是一一对应的。汇编器拥有一张包括所有可能助记符及其参数的表，它逐行读取汇编语言程序，把每一行都分解成为助记符和参数，然后把这些短小的单词和字符与表中的内容匹配。通过这种匹配的过程，每一个语句都会找到与其对应的机器码指令。

注意，汇编器如何知道LXI指令必须将寄存器DE的值设置为地址0109h（Text的地址）。如果LXI指令本身被存放在地址0100h处（CP/M将程序加载至内存开始运行时的起始地址），而0109h则是Text字符串的起始地址。一般来说，程序员在使用汇编器时有很多方便之处，其中一点就是不需要关心汇编程序各部分在内存中的存放地址。

- 第一个编写汇编器的人需要手工对程序汇编。如果要为机器写一个新的汇编器（或者对其进行修改），则可以使用汇编语言编写该程序，然后使用原有的汇编器对其进行汇编。一旦新的汇编器通过了汇编，则它也就能够对自身进行汇编。
- 每当一种新的微处理器面世，就需要为其编写新的汇编器。然而，新的汇编器可以在已有的计算机上编写，并利用其汇编器进行汇编。这种方式称为交叉汇编（cross-assembler），即利用计算机A的汇编器对运行在计算机B上的程序汇编。
- 虽然汇编器的引入消除了汇编语言编程中重复性的劳动部分（即手动汇编部分），但汇编语言仍然存在两个主要问题。第一个问题（也许你已经意识到了），使用汇编语言编程非常乏味，因为这是在微处理器芯片级的编程，因此不得不考虑每一个微小的细节。
- 汇编语言存在的第二个问题是不可“移植”（portable）。如果你为Intel 8080写了一个汇编语言程序，则该程序不能在Motorola 6800上运行，你必须在6800上重写一个相同功能的汇编语言程序。编写类似程序的过程也许没有编写第一个程序那么困难，因为你已经解决了程序的组织和算法问题，但仍然还有很多工作要做。
- 上一章介绍了现代微处理器集成浮点运算机器码指令的原理。不可否认，这已经为我们带来了很大的便利，但仍不能令人特别满意。一种更好的方式是：完全放弃那些实现每个基本操作的机器码指令，这些指令与处理器相关，因而导致程序缺乏移植性。我们采用的替代策略是使用一些经典的数学表达式来描述复杂的数学运算。下面是一个表达式的例子：

$$A \times \sin(2 \times \text{PI} + B) / C$$

上式中的A、B、C代表数字，而PI = 3.14159。

假设在某个文本文件中有这样一个表达式，那么我们可以尝试编写一个汇编语言程序来读取该文本文件，并将其中的数学表达式转换为机器码。

如果只需要计算一次该表达式，那么可以手工计算或借助计算器来完成。如果需要对A、B、C取不同的值多次计算该表达式，那么你可能要考虑使用计算机来完成这些计算。因此，代数表达式不会孤立地出现，必须考虑其前后的语句，这些语句使表达式对不同的值进行运算。

- 现在你所创建的东西已经触及所谓的高级程序设计语言（high-level programming language）。我们一直在介绍的汇编语言称做低级语言（low-level programming language），因为它与计算机硬件的关系相当紧密。尽管除了汇编语言以外的其他程序设计语言都可以称为“高级语言”，但它们之间还是有高低之分的，一些语言通常被认为比别的语言更高级。如果你是一家公司的总裁，坐在计算机前输入这些命令（也可能做得更轻松：口头发布这个命令），“计算出本年度的收益和损耗，生成年度报表，最后打印出2000份送至每个股东”，你所使用的才真正是一种非常高级的语言！但在实际工作中，程序设计语言还达不到这种理想化的水平。

- 人类语言通常都是经历了千百年复杂的互相影响、偶然演变以及不断吐故纳新才形成的，就算一些人工语言如世界语（Esperanto），也处处显露出与现实语言的渊源。但高级程序设计语言是经过深思熟虑的设计的，更加概念化的语言。设计程序设计语言所面临的一大挑战就是：如何让语言更具吸引力。因为语言定义了人们向计算机传送指令的方式，只有更易用的方式才能让人们对语言产生兴趣。据1993年的一项估算，从1950年到1993年大约有1000多种高级程序设计语言被发明出来并被应用。
- 然而，仅仅定义（define）高级语言，包括定义语言的语法（syntax）来表达该语言可以描述的一切事物，还远远不够；我们还需要为其编写一个**编译器（compiler）**，编译器可以将高级语言的程序语句转换成机器码指令。同汇编器类似，编译器也是逐字逐句地读取源文件并将其分解成为短语、符号和数字的，但实现过程要比汇编器更加复杂。从某些方面来看，汇编器相对简单，因为汇编语言的语句和机器码是一一对应的。而一般的高级语言却不具备这种对应关系，编译器通常必须把一条语句转换多个机器码指令。编译器的编写非常复杂，许多书都是用全部的篇幅来讲解如何设计和构造编译器。
- 当然，任何事物都是具有两面性的，高级语言也不例外，它有很多优势但也存在不少缺陷。高级语言最基本的优点在于它比汇编语言易于学习并且更容易编写程序，用高级语言编写的程序通常更加清晰简明——与汇编语言不同，高级语言通常不依赖于特定的处理器，因此它们通常具有良好的可移植性。因为这种特点，使用高级语言的程序员不再需要关心最终运行程序的计算机的底层结构。当然，如果要在不同类型的处理器上运行程序，则需要用处理器对应的编译器将程序转换成对应的机器码。因此，最后生成的可执行文件仍然只适用于特定的处理器。
- 另一方面，有一种普遍现象：一个优秀的汇编程序员所编写的程序比编译器所产生的代码更加有效率。也就是说，从高级语言程序生成的可执行程序比相同功能的汇编语言程序更大，并且运行速度更慢（但从近年的发展来看，这种差别已变得不再明显，因为微处理器变得更加复杂，而且编译器在优化代码方面也更加成熟）。
- 此外，虽然高级语言提高了处理器的易用性，但并没有让其变得更强大。微处理器的任何一个功能都可以通过汇编语言实现，因此汇编语言可以高度利用处理器的功能。因为高级语言必须转化成机器码，所以它只会降低微处理器的能力。事实上，如果某种高级语言具有真正意义的可移植性，那么它将不能使用某些处理器的特有功能。
- 例如，许多微处理器都有移位指令。如前所述，这些指令能将累加器中的字节的每一位向左或向右移动。但事实上，几乎没有哪一种高级语言包含这种操作。如果在程序中需要进行移位操作，则必须通过乘2或除2来模拟该过程（这并不是什么坏事：事实上，许多现代编译器都是利用处理器的移位指令来实现乘以或除以2的幂的）。除此之外，许多高级语言也不包括按位逻辑运算。
- 在早期的家用计算机中，大部分应用程序都是用汇编语言写的，而现在除了一些特殊的应用场合之外，汇编语言已经很少使用了。而今处理器引入了一些新的硬件，可以实现流水线技术——同时有若干个指令码渐次执行——这使得汇编语言变得更加复杂且不易处理。与此同时，编译器却变得更加成熟，越来越多的程序开始使用高级语言来编写。现代计算机大容量的存储器也作为一个重要的角色，推动了这种趋势：程序员不再局限于编写运行在小内存和小磁盘上的程序。
- 早期的计算机设计者都曾尝试用数学符号来描述问题，但公认的第一个真正可以工作的编译器是A-0，它是为UNIVAC开发的编译器，于1952年由雷明顿兰德公司（Remington-Rand）的格瑞斯·穆雷·霍珀（Grace Murray Hopper，1906-1992）开发完成。霍珀博士的早期计算机研究工作始于1944年，那时她效力于霍华德·艾肯（Howard Aiken），主要研究Mark I。在她八十多岁的时候，仍然孜孜不倦地在计算机界工作，当时她在DEC（Digital Equipment Corporation）公司从事公关事务。
- FORTRAN语言是目前仍在使用的最古老的高级语言（虽然这些年来人们对其进行了大量修改）。你可能注意到了，很多计算机语言都是以大写字母命名的，这是因为它们的名字大都是由几个单词的首字母组成。FORTRAN这个名字来源于FORmula的前三个字母和TRANslation的前四个字母的组合，它由IBM在20世纪50年代中期开发，主要应用于704系列计算机。自其发布的几十年来，FORTRAN一直被

认为是科学和工程应用程序开发的首选语言。它广泛地支持浮点运算，甚至支持非常复杂的数的运算（即我们上一章讲到的由实数和虚数构成的复数）。

- 任何一种计算机程序设计语言都有其支持者和批评者，而且人们通常只对自己喜欢的语言有热情。本书尽量以一种客观的态度来讨论某种语言，这里选取了一种语言作为原型，通过它来解释那些几乎已经销声匿迹的程序设计概念。我们的选择是ALGOL（即ALGOrithmic的缩写，有趣的是，ALGOL也是仙女座第二亮的恒星的名字）。ALGOL作为过去40年中许多曾经流行一时的通用高级语言的直接鼻祖，也非常适合用来研究高级程序设计语言的本质，该语言可看做是一粒种子，它的成长最终形成了高级语言这棵大树。直到今天，人们仍然在使用“类ALGOL”程序设计语言的概念。
- 通过在命令行运行ALGOL编译器对FIRST.ALG文件进行编译，其格式如下

## ALGOL FIRST.ALG

ALGOL编译器对拼写的检查非常严格，它在这一点上比传统的语文教师更甚。因为输入程序时，误把“end”拼写做“ende”，所以编译器通过提示信息告诉我们程序中有语法错误（syntax error）。当编译器检查到“ende”时，它期待能遇到一个可识别的关键字（keyword），但由于上述错误，编译不能通过。

将程序中的错误改正之后，可以再次执行编译命令。由于系统平台和编译器版本的不同，有时编译器会直接生成一个可执行文件（CP/M平台下此文件名为FIRST.COM，MS-DOS平台下名为FIRST.EXE）；有时还需要再执行一个步骤才可以完成。不论是哪种情况，最后你都可以在命令行执行FIRST程序：

FIRST

里还有一个拼写错误：first被误做fist！编译器没有检查出这个错误，因此它被称为运行时错误（run-time error）——程序被执行时才出现的错误。

很明显，我们的第一个ALGOL程序中，print语句的功能是把一些信息显示到屏幕上，在本程序中是显示一行文本（从功能的角度来看，该程序与本章开始所给出的汇编程序是等价的）。ALGOL语言的正式规范中并不包括print语句，但我们假设所使用的特定ALGOL编译器包括这个便利的工具，它有时候也被称做内部函数（built-in function）。除了begin和end之外的大部分ALGOL语句都要以分号结尾。你可能注意到了print语句使用了向右缩进的格式，这并不是必要的，其作用只是为了让程序的结构更加清晰。

- 假设现在要编写一个用于两个数相乘的程序。每一种程序设计语言都包括变量（variable）的概念。程序中的变量可以是一个字母、一个短的字母序列，也可以是一个单词，由程序员自己决定。变量名实际上对应内存的一个存储单元，但在程序中是通过名字来访问该存储单元的，而不是直接使用存储单元的地址值。

real语句称为声明（declaration）语句，用来指明程序中要定义的变量。在该程序中，变量a，b，c被定义为实数（real）类型或浮点数类型（同时，ALGOL语言也支持使用integer关键字来定义整数型变量）。程序设计语言中的变量名通常以字母开头，变量名也可以包括数字，但前提是第一个字符必须是字母。变量名不能含有空格，也不能包含除字母和数字以外的其他大部分字符。通常编译器会规定变量名的最大长度。

```
begin
    real a, b, c;
    a := 535.43;
    b := 289.771;
    c := a × b;
    print ('The product of', a, ' and ', b, ' is ', c);
end
```

假如我们使用的特定ALGOL编译器支持IEEE浮点数标准，则本程序中所定义的三个变量每一个需要4个字节的存储空间（采用单精度格式）或8个字节的存储空间（采用双精度格式）。

声明语句之后的三个语句是赋值（assignment）语句。在ALGOL语言中，赋值语句很容易被识别，因为它的格式很固定，总是在冒号后面跟着一个等号（在大多数计算机语言中，赋值语句通常只包括等号）。赋值语句的冒号左边是一个变量，而等号右边是一个表达式，表达式的计算结果将被赋值给左边的变量。前两条赋值语句指明，变量a, b将分别被赋予一个特定的值；第三条赋值语句指明，将a和b的乘积赋值给变量c。

时至今日，我们所熟悉的乘法符号“ $\times$ ”已经不允许出现在程序设计语言中了，因为它没有被包括在ASCII和EBDCIC字符集中。大多数程序设计语言使用星号（\*）来替代它作为程序中的乘号标记。尽管ALGOL使用了普遍使用的斜杠（/）作为除法标记，但在该语言仍然可以使用除法标记（ $\div$ ），该标记用于整数除法，用来指明被除数与除数的倍数关系。ALGOL还使用了另一个非ASCII字符“↑”，该箭头符号用来做乘方运算。

最后的print语句用来显示所有变量的值。它包含文本和变量，并以逗号分隔。print语句的主要工作并不是用来显示ASCII码值的，但本程序中却做了更多的工作：将浮点数也转换成了ASCII码并显示：

The product of 535.43 and 289.771 is 155152.08653

接着会执行end语句，程序终止并将控制权交还给操作系统。

- 如果要将另外两个数相乘，则需要做以下工作：修改程序，改变变量的值，重新编译并重新运行程序，这将是一件非常烦琐的工作。为了避免这些重复工作，我们可以借助于另一个内部函数read。修改后的程序如下：

```
begin
    real a, b, c;
    print ('Enter the first number: ');
    read (a);
    print ('Enter the second number: ');
    read (b);
    c := a × b;
    print ('The product of ', a, ' and ', b, ' is ', c);
end
```

read语句的功能是读取从键盘键入的ASCII码值，并将其转换成浮点数。

- 循环（loop）是高级语言的重要组成部分。循环使得程序可以对同一个变量的不同取值反复执行相同的操作。假设我们要写一段程序用来计算3, 5, 7, 9各自的平方，可以这样编写程序：

```
begin
    real a, b;

    for a := 3, 5, 7, 9 do
        begin
            b := a × a × a;
            print ('The cube of ', a, ' is ', b);
        end
    end
```

for语句将变量a的值第一次设为3，然后执行do关键字后面的语句。如果do后面要执行的语句不止一条（如本例），则必须将它们置于begin和end之间，这两个关键字定义了一个语句块（block）。第一次循环之后，for语句会依次为a赋值5, 7, 9并执行相同的语句块。

下面的程序中采用了for语句的另一种使用方式，这段程序用来计算3~99之间所有奇数的立方。

```
begin
    real a, b;
    for a := 3 step 2 until 99 do
        begin
            b := a × a × a;
            print ('The cube of ', a, ' is ', b);
        end
    end
```

for语句将变量a初始化为3，并执行for后面的语句块。第一次循环结束后，变量a与step关键字后面的增量相加，这里是2。新得到的a的值是5，它将用于第二次执行语句块。变量a继续增加2并用于下一次循环，直到a的值超过99，这时for循环结束。

一般而言，程序设计语言对语法都有着非常严格的要求。在ALGOL 60中，就关键字for而言，其语法格式是：for的后面只能跟一个变量名。而英语中的这种限制宽松的多，单词for的后面可以跟所有类型的单词，例如“for example”，“for can”等。尽管编译器是非常复杂的程序，但其所能解释的语言显然要比人类的语言简单得多。

- 大部分程序设计语言的另一个重要特征体现在条件（conditional）语句的使用。条件语句的特点是，只有当某个条件成立时才会执行另一条对应的语句。在下面的例子中，我们使用ALGOL的内部函数sqrt来计算一些数的平方根。sqrt函数的参数不能是负数，因此要在程序中通过条件测试避免这种情况。

```
begin
    real a, b;

    print ('Enter a number: ');
    read (a);
    if a < 0 then
        print ('Sorry, the number was negative.');
    else
        begin
            b = sqrt(a);
            print ('The square root of ', a, ' is ', b);
        end
    end
```

左尖括号（<）是小于号。如果程序的使用者输入的是一个小于0的数，if语句中的判断语句为真，因此第一个print语句将会被执行。反之，如果该数大于或等于0，则else关键字后面的语句块则会被执行。

- 本章目前所用到的变量都是一个变量对应一个值，我们也可以用一个变量对应多个值，数组（array）就是一个很好的选择。在ALGOL程序中可以这样声明一个数组：下图中语句定义一个数组变量a，它可以用来存放100个不同的浮点数，这些数被称做数组元素。可以使用数组名加标号的方式来引用数组元素，例如，第一个数组元素是a[1]，第二个是a[2]，最后一个a[100]。方括号中的数字称做数组下标（index）。

下面的程序用来计算1~100所有数的平方根，将结果保存在一个数组中，然后再通过循环将这些结果显示出来。代码如下：

```
begin
    real array a[1:100];
    integer i;
    for i := 1 step 1 until 100 do
        a[i] := sqrt(i);

    for i := 1 step 1 until 100 do
        print ('The square root of ', i, ' is ', a[i]);
end
```

程序中还定义了一个整型变量i（由于它是integer的首字母，经常被程序员用做整型变量名）。第一个for循环的执行过程中，每个数组元素被赋值为其下标的平方根；第二个for循环执行过程中，数组中的每一个元素被显示出来。

- 变量的类型有很多，除了我们已经介绍过的实型和整型之外，变量还可以被声明为布尔型（Boolean，该名称是为了纪念第10章提到的乔治·布尔）。布尔变量的取值只可能有两种，即true和false。在本章的最后将介绍一个用到布尔数组的例子（这个例子也将用到目前所介绍的大部分内容），来实现一个寻找素数的著名算法——爱拉托逊斯筛法（Sieve of Eratosthenes）。爱拉托逊斯（约公元前276–196年）传说是亚历山大图书馆的管理员，他因准确计算出地球的周长而永载史册。
- 素数是只能被1及其本身整除的一类整数。第一个素数是2（也是唯一的偶数素数），其他的素数还包括3, 5, 7, 11, 13, 17, 等等。
  - 爱拉托逊斯方法以2开始的整数表开始，因为2是素数，因此所有可以被2整除的数都被排除掉（即除了2之外的全部偶数）。接下来是3，因为3是素数，因此所有能被3整除的数也被排除掉。因为4在第一个步骤中已经被排除掉，所以下一个要考虑的数是5，即排除所有5的倍数。按这种方式不断循环，最后剩下的都是素数。
  - 下面的ALGOL程序用来筛选2~10,000之间的所有素数，程序中定义了一个布尔数组，用来对所有的数进行标识。该程序如下：

```
begin
    Boolean array a[2:10000];
    integer i, j;

    for i := 2 step 1 until 10000 do
        a[i] := true;

    for i := 2 step 1 until 100 do
        if a[i] then
            for j := 2 step 1 until 10000 ÷ i do
                a[i × j] := false;

    for i := 2 step 1 until 10000 do
        if a[i] then
            print (i);
end
```

第一个for循环将数组a的每一个元素的初始值设置为布尔值true。这里的true表示该位置的数是素数，因此现在程序默认所有的数都是素数。第二个for循环的范围是1~100（100刚好是10000的平方根）。在第二个for循环中，如果判断条件成立，该数为素数，即a[i]为true，则第三个for循环则会把该数的所有小于或等于10000的倍数（除了其本身）设置为false，因为这些数都不是素数。最后的for循环用来输出所有的素数，这里的判断条件是：若a[i]为true，则i为素数。

- 程序设计到底是一门科学还是一门艺术呢？这的确是一个有趣的问题，一些人甚至还为此争论不休：一方面，你或许在大学里系统地学习了计算机科学（Computer Science）课程；另一方面，你又读过如唐纳德·克努斯（Donald Knuth）的名著《计算机编程艺术系列》（The Art of Computer Programming series）等著作。然而物理学家理查德·费叶曼（Richard Feynman）曾这样写道：“从某种程度上看计算机科学像是一种工程，它的工作范畴是利用一些事物去实现其他事物。”
  - 在程序设计中有一种现象：如果让100个人来编写输出素数的程序，你可能会得到100个不同的解决方法。就算所有的程序员都使用“爱拉托逊斯筛法”来解决这个问题，其最后所写的程序也不一定与本文所写程序完全相同。如果说程序设计是一门科学，那么就不应该出现如此多的解法，而不正确的方法将会非常明显。偶尔，一个程序设计问题会诱发出极富创造性的火花或洞若观火般的觉察力，这就是所谓的程序设计的“艺术”。但是，程序设计的更多的时候是设计和建造，就像修建一座大桥的过程。
- 早期的程序设计对编程人员的要求很高，所以很多早期的程序员都是科学家或工程师，他们通常利用FORTRAN或ALGOL中的数学算法来描述并解决各自领域的问题。回顾程序设计语言发展的整个历程时，我们会发现，人们一直在努力开发一种能为更大范围的人群所使用的语言。
- 第一个成功地为商务系统所使用的程序设计语言是COBOL（Common Business Oriented Language），今天它仍然被广泛使用。COBOL于1959年开始开发，由美国工业界和国防部组成的委员会发起并实施，它的设计思路受到格瑞斯·霍珀早期编译器的影响。从某些方面来看，COBOL的设计中渗透了这种思想：使管理人员——可能并不进行实际的编码工作——但他们至少可以看懂程序代码，而且能够检测程序能否完成预定工作（实际上这种情况非常少见）。
- COBOL语言广泛支持读取记录（record）和生成报表（report）。记录是按照统一方式归类整理的信息的集合。例如，保险公司一般会维护一个包括其所售的所有保险信息的大型文件，每一项保险业务称为一条单独的记录。每一条记录包括客户的姓名、出生日期等信息。早期编写的COBOL程序，大都是为了处理存储在IBM打孔卡片上的80列记录而编写的。为了尽量减少孔洞所占用的卡片空间，年份通常设计成2位而不是4位，随着时间的推移，这个设计的缺陷逐渐显露出来，最终导致在2000年出现了著名的“千年虫问题”（millennium bug）。
- 在20世纪60年代中期，为了配合System/360项目的开发，IBM同时开发了程序设计语言PL/I（I是罗马数字中的1，因此PL/I的含义是：Programming Language Number One）。PL/I的设计者们想要使其融合ALGOL的块结构，FORTRAN语言的数学函数功能以及COBOL处理记录和报表的能力，但该语言却远没有达到FORTRAN和COBOL那样广泛的使用程度。
- 虽然FORTRAN，ALGOL，COBOL以及PL/I都可以应用于家用计算机，但它们对于小型计算机的影响远没有BASIC语言那么深远。
- BASIC（Beginner's All-purpose Symbolic Instruction Code）由达特茅斯（Dartmouth）大学数学系的约翰·克莫尼（John Kemeny）和托马斯·克鲁兹（Thomas Kurtz）在1964年开发，该语言最初是为达特茅斯分时系统而设计的。达特茅斯大学的学生并非数学或工程专业，因此他们不应该为打孔卡片和复杂的程序语法花费太多精力，他们要做的只是端坐于计算机终端前，在数字后面输入一些BASIC语句来完成编程。BASIC语句前的数字用来指明该语句在程序中的次序。前面没有数字的语句是系统命令，如SAVE（将BASIC程序保存至磁盘），LIST（按顺序显示行）以及RUN（编译并运行程序）。BASIC手册的第一版中的第一个程序是这样的：

```
10 LET X = (7 + 8) / 3
20 PRINT X
30 END
```

- 与ALGOL语言不同，BASIC不要求程序员指定变量的存储类型，究竟一个变量是保存为整型还是浮点型并不需要程序员担心，大部分数默认都是以浮点数格式存储的。
- 很多BASIC的后续版本都是解释型（**interpreter**）而不是编译型（compiler）。如前所述，编译器读取源文件并生成一个可执行文件；而解释器却采取边读边执行的方式，不会产生新的文件。解释器比编译器的原理简单一些，因此更容易编写，但其运行程序的速度要比后者要慢。BASIC语言应用于家用计算机的时间较晚，1975年，比尔·盖茨（Bill Gates，生于1955年）和其好友保罗·艾伦（Paul Allen，生于1953年）为Altair 8800编写了BASIC解释器，这一事件可以视为BASIC在此领域的开端，同一年他们创建了微软公司（Microsoft Corporation）。
- Pascal程序设计语言继承了ALGOL的大部分结构，同时还继承了COBOL的记录处理功能，它由瑞士计算机科学教授尼尔莱斯·沃思（Niklaus Wirth，生于1934年）在20世纪60年代末开发完成。IBM PC的程序员对Pascal非常青睐，而备受欢迎Pascal版本却是大名鼎鼎的Turbo Pascal。1983年，宝兰公司（Borland International）发布了Turbo Pascal，当时的售价是49.95美元。Turbo Pascal由一名叫安德斯·海尔斯伯格（Anders Hejlsberg，生于1960年）的丹麦大学生开发，它提供了完整的**集成化开发环境**（**integrated development environment**）。程序的文本编辑器和编译器集成在一起，这样就方便了程序的调试和运行，大大加快了程序开发速度。集成化开发环境以前主要用于大型计算机，Turbo Pascal实现了在小型计算机上的突破。
- Pascal对Ada的影响也非常大。Ada是为美国国防部开发应用的一种语言，它以奥古斯塔·艾达·拜伦（Augusta Ada Byron）命名。在第18章曾提到过，奥古斯塔·艾达·拜伦是查尔斯·巴贝芝的解析机发展历程的记录者。
- 接下来就是C，一种深受喜爱的程序设计语言。C语言主要是由贝尔电话实验室的丹尼斯·M·里奇（Dennis M.Ritchie）开发的，从1969年开始设计并于1973年开发完成。人们常常对为什么以C来命名该语言感兴趣，答案其实很简单，它是一种早期的程序设计语言B的后继者。B是BCPL（Basic CPL）语言的一种精简版本，而BCPL来源于CPL（Combined Programming Language）。
- 如第22章所述，UNIX操作系统在设计的过程中充分考虑到了可移植性。当时的许多操作系统都是基于某种处理器的，并且使用汇编语言编写，基本上没有可移植性可言。1973年，UNIX采用C语言编写（更准确地说，应该是重写）成功，从此以后UNIX操作系统和C语言就变得密不可分了。
- C是一种风格非常简洁的语言。例如，ALGOL和Pascal使用关键字begin和end来界定程序块，而在C中这两个单词被一对大括号“{}”取代。在16位或32位微处理器中，i++这种语句仅需要一条机器码指令就可以执行。
- 在本章的前面曾讲过，很多高级语言都不支持移位操作和按位布尔运算操作，而许多处理器其实支持这类操作，C语言打破了这种局限，它广泛地支持这类运算。除此之外，C语言的另一重要特征是对于指针（pointer）的支持，指针本质是数字化描述的内存地址。C语言中的很多操作与通用处理器的指令非常相似，因此C也被称为高级汇编语言（high-level assembly language）。与类ALGOL语言相比，C的操作集与通用处理器的指令集接近程度更高，或者说远胜过它们。

- 但是，所有的类ALGOL语言——即大多数常用程序设计语言——其设计模式都是基于冯·诺依曼计算机体系的。设计一种非冯·诺依曼体系的程序设计语言并非易事，而让人们接受并使用这种语言则更加困难。LISP (List Processing) 是一种非冯·诺依曼体系程序设计语言，它主要应用于人工智能领域，由约翰·麦卡锡 (John McCarthy) 在20世纪50年代末期开发完成。APL (A Programming Language) 是另一种全新的语言，与LISP完全不同，它同样完成于20世纪50年代末期，由肯尼斯·艾佛森 (Kenneth Iverson) 开发。APL的特殊之处在于，它使用一个特殊的符号集，利用其中的符号可以一次性对整个数组里的数字完成操作。
- 类ALGOL语言一直在程序语言领域占据着重要地位，而且近年来，此类语言在一些方面进行了改进，导致面向对象程序设计语言 (object-oriented language) 的产生。面向对象语言主要应用在图形化操作系统中，我们将会在下一章（也是最后一章）介绍这种操作系统。

## 25 图形化革命

---

- 1945年，万尼瓦尔·布什 (Vannevar Bush 1890-1974) 发表了一篇关于未来科学大胆猜想的文章，这篇文章名为《思维之际》 (As We May Think)，文中描述了一种未来的发明，这项发明可以帮助科学家和研究人员更轻松地处理日益增多的技术期刊及文章。布什提出可以利用微缩胶片作为解决方案，同时他构想出了一种叫做麦克斯储存器 (Memex, 又名记忆扩展器) 的设备，它可以对书籍、文章、录音和图片进行保存。麦克斯储存器还有一项重要的功能，那就是它可以让用户根据某个主题在所有的素材之间建立起关联，这些关联的基本来源就是我们人类的思维。他还大胆预言一种新的职业群体，他们的工作就是在繁杂的信息载体之间提炼并建立起可靠的关联。
- 在20世纪的那个年代，讲述辉煌未来的文章屡见不鲜，但《思维之际》这篇文章却异常耀眼。它所讲述的不是可以替代我们去做家务劳动的设备，也不是关于未来运输方式或智能机器人的故事，这个故事的主角是信息 (Information)，故事的主线是如何利用新技术成功的处理信息。
- 回顾历史，从第一台继电器计算器出现到现在为止，65年过去了，计算机的体积越来越小，处理速度越来越快，价格也越来越便宜。这一趋势极大地改变了计算的原始属性。当计算机价格变得很便宜，可以实现人手一台；当计算机体积越小、处理速度越快，软件就能发挥更大的作用，而机器就可以承担越来越多的工作。
- 要充分利用日益增长的运算和处理能力，较好的一种方法就是不断改进计算机系统中的关键部位，最典型的就是用户界面 (User Interface) ——它可以看作人机交互的轴心。人与计算机是两种完全不同形式的“客观存在”，只可惜在人机交互的这个过程中，与其让计算机去适应人类的特性，远不如劝服人们进行调整以适应计算机的特性来得容易。
- 在计算机发展早期，交互式这个概念并没有它的实际意义。人们编程时更多使用的是开关和电缆，有一部分人使用的是打孔纸带或胶片。到了20世纪50到60年代（有些观点认为这一时间可以延续到70年代），计算机已经可以使用批处理 (batch processing) 进行编程：程序和数据被“分布”在打孔卡上，然后一次性录入到计算机内存。这些工作完成之后，再由程序对数据进行分析，得出结论，最后将结果打印在纸上。
- 最早的交互式计算机运用的是电传打字机。我们回忆一下前面讲过的达特茅斯 (Dartmouth) 时分操作系统（原型出现于20世纪60年代早期），这种系统支持多个电传打字机同时工作，而且互不影响。此类系统中，用户在打字机上输入一行，计算机会相应地输出一行或多行。通常，打字机和计算机之间的信息交流是由一串ASCII码（也有可能是其他字符集）来完成的，这些ASCII码大多由字符编码组成，当然还包括像回车、换行等一系列简单的控制字符编码。随着机器的运行，相应的事务也随着打印纸的旋转逐步推进。

- 阴极射线管（cathode-ray tube, CRT，这是20世纪70年代随处可见的设备）并不受这类限制。使用软件来协调整个屏幕显得更加灵活方便——这可以算得上是一种二维的信息平台。但是为了尽量保持操作系统显示输出的逻辑一致性，早期那些为小型计算机编写的软件都把CRT显示器看做“玻璃屏幕电传打字机”——所有内容都是一行一地显示的，当字符排到底端，屏幕被填满时，屏幕上的内容要整体向上翻滚。除了CP/M（微处理机操作系统）中的所有工具软件之外，大部分MS-DOS下的工具软件都采用这种方法——它们都仿照电传打字机的工作方式来使用视频显示器。使用电传打字机这种工作原理的操作系统有很多，或许UNIX才算是最典型的原型操作系统之一，它还一直保留着这种“传统工艺”。
- 不巧的是，ASCII码字符集不完全适用于阴极射线管的工作方式。在最原始的ASCII码设计中，编码1Bh被标识为Escape，它的主要作用是帮助字符集进行扩充。在1979年，美国国家标准协会（American National Standards Institute, ANSI）发布了一项题为“ASCII码使用的附加控制（Additional Controls for Use with American National Standard Code for Information Interchange）”的标准。该标准发布的初衷是为了“适应二维字符-图像设备输入/输出控制中迫在眉睫的相关需求，其中包括阴极射线管和打印机之间的交互终端……”
- 其实Escape的编码1Bh只占据一个字节，且它的含义是唯一的。Escape如果作为一串序列的前缀字符，那么这串字符序列的含义也随之改变。比如下面这串序列：

**1Bh 5Bh 32h 4Ah**

可以看出Escape 编码随后紧跟的是字符“[”“2”“]”的ASCII码，现在这一串字符的含义为“清屏”然后移动光标至左上角。这种定义在电传打字机上是不可能出现的。下面这串序列：

**1Bh 5Bh 35h 3Bh 32h 39h 48h**

即Escape编码随后紧跟的是字符“[”、“5”、“；”、“2”、“9”、“H”，这串字符的作用是把光标移到第5行的第29列。

键盘和CRT一起对远程计算机传输来的ASCII码（可能还包括Escape字符序列）做出响应，这种设备我们称之为哑终端（dumb terminal）。哑终端相对于电传打字机速度要更快，从某种程度来讲也更灵活，但从速度的提高程度上来讲，并不足以引领用户界面的革新。真正的革新出现在20世纪70年代小型计算机中——它类似于第21章我们构建的假想计算机，配备了“视频显示存储器”，并作为微处理器地址空间的组成部分。

- 第一个预示着家用计算机将与它的孪生兄弟——体积庞大、价格昂贵的大型机划分界限的标志性的事件是VisiCalc的使用。VisiCalc由丹·布莱克林（Dan Bricklin，生于1951年）和鲍勃·弗兰克斯顿（Bob Frankston，生于1949年）设计并编程实现，而这套系统于1979 年引入苹果II型电脑（Apple II）中。VisiCalc通过屏幕将一个二维电子数据表呈现给用户。在VisiCalc出现之前，数据表就是一张划分好了行、列的纸，主要用于一系列计算。VisiCalc用视频显示器将纸质材料取而代之，通过这种方式，用户可以在数据表中随处游走，在相应位置输入数据、公式，并在修改后对结果进行重新计算，为用户提供了更多的自由。

我们惊讶与无奈的是，VisiCalc这款应用程序无法在大型机上运行。因为像VisiCalc这类程序需要以较快的速度不断更新屏幕，所以，它们直接将数据写入Apple II视频显示器所配备的RAM中。该RAM是微处理器地址空间的一部分。大型时分计算机以及哑终端之间的接口速度过慢，以至于电子报表程序无法使用。

- 计算机对键盘的响应速度越快，对视频显示器的更新速度越快，则人机交互就越频繁。在IBM PC刚刚推出的10年里（即20世纪80年代），几乎搭配的所有软件都是直接将输出的数据写入视频显示存储器的。当时IBM建立了一套硬件标准，其他硬件制造商参照这些标准去生产，这样软件制造商就可以绕过操作系统直接操控硬件，统一化的硬件标准确保了程序的正确运行（同时也杜绝了不能运行的情况）。

如果所有同构的PC都拥有异构的视频显示器硬件接口，这种做法无异于将软件厂商推到了火坑里，因为做软件的同时还要关注硬件设计细节是不现实的。

- IBM 早期PC配备的应用程序通常只有字符输出，很少有图形输出。使用文本输出大大加快了应用程序的运行速度。假设PC上配备一台第21章所描述的视频显示器，那么程序所要做的就是把字符相应的ASCII码写入内存，然后屏幕上就会显示出该字符。但是如果使用的是图形视频显示设备，那么相应的程序需要将8个或更多的字节写入到内存中，这样做的目的就是画出字符的外观并以图形的方式显示。
- 在计算机的发展史上，从字符显示到图形显示是一次伟大的变革，计算机在这次变革中迈出了重要的一步。然而，相对于显示文本和数字所采用的软硬件，图形化计算机的软硬件发展十分缓慢。早在1945年，约翰·冯·诺伊曼（John von Neumann）就预见了一种类似示波器的显示器，它的最大特点是可以显示图像化信息。但直到20世纪50年代早期，MIT（当时得到了IBM资助）建立了林肯实验室，实验室的主要任务就是帮助美国空军开发一种适用于防空系统的计算机，这次项目使计算机的图形化成为现实。该项目被称为半自动地面防空系统，简称SAGE（Semi-Automatic Ground Environment），项目的内容包括构建一个显示图形的屏幕，以此来帮助操作员分析海量数据。
- 早期的视频显示器，比如SAGE中使用的这一种显示器，与我们今天所使用的PC配套显示器不尽相同。我们日常所用的PC配套的纯平显示器属于光栅（raster）显示器。它的原理就像电视机，每一幅图像背后都是一行行的光栅线，这些光栅是电子枪（electron gun）发出光束迅速来回移动覆盖整个屏幕而形成的。我们可以把屏幕想象成一个巨大的矩形阵列，阵列的每个元素都是一个点，这些点称为像素（pixels）。在计算机内部，有一块专门供视频显示器使用的内存区域，屏幕上的每一个像素点由1个或多个比特表示。这些二进制数值不仅决定了像素点的亮度，还决定了它的颜色。
- 由于要用到的颜色数目逐渐增加，为了表示这些颜色，每个像素所需要的比特越来越多，显示适配器需要配备的存储器容量也越来越大。比如我们想使像素点具备不同的灰度，那么可以提供一个字节的存储空间。在这种处理方式下，字节00h代表着黑色，FFh代表着白色，两者之间的值代表着不同的灰度。
- CRT上的色彩空间由三个电子枪产生，每一个电子枪分别产生三原色中的一种，包括红色、绿色、蓝色（用放大镜来观察电视机或彩色计算机屏幕，你可以清楚地看到，每一幅图像都是利用许许多多不同的三原色组合显示出来的），红绿组合出黄色，红蓝组合出品红色，蓝绿组合是青色，三原色组合出白色。
- 在最简单的彩色显示适配器中，表示每个像素点需要3个比特。最直观的编码方式就是每一种原色对应编码中的1位。

比特	色彩
000	黑
001	蓝
010	绿
011	青
100	红
101	品红
110	黄
111	白

这种方案可能只适合简单的类似卡通画的图像。真实世界出现的几乎所有颜色都是由红、绿、蓝三原色的不同色阶（levels）组合而成的。如果为每个像素赋予2个字节的存储空间，这样一来，可以给每一个原色分配5位（1位保留）存储空间，这种方法可以表示出红、绿、蓝三种颜色且每种颜色具备32种不同的色阶，这样算下来总共有32,768种不同的颜色。这种模式通常称做高彩色（high color）或数千种颜色（thousands of colors）。

- 我们下面尝试一下用3个字节来表示一个像素，三原色中的每一种各占一个字节。这种编码模式使红、绿、蓝各自呈现出256种不同的色阶，这样算下来共有16,777,216种不同的颜色，这种方案通常叫做全彩色 (full color) 或百万种颜色 (millionsof colors)。如果视频显示器的分辨率为 $640 \times 480$ ，即水平640像素，垂直480像素，将像素点的数量乘以表示每个像素点需要的字节数可以得到，共需要921,600字节的存储容量，即将近1MB。
- 每个像素所赋予的比特数有时也称做色深 (color depth) 或色彩分辨率 (colorresolution)。颜色数与单个像素被赋予的比特数的关系如下：颜色数 =  $2^{\text{每个像素所赋予的比特数}}$
- 如果视频适配卡配备的存储器容量有限，那么它的最大色深或色彩分辨率自然而然也受到约束。假设有一个配备了1 MB存储器的视频适配卡，在每个像素被赋予3个字节的情况下分辨率可以达到 $640 \times 480$ 。如果想把分辨率提高到 $800 \times 600$ ，存储器就不足以每个像素赋予3个字节，必须缩减到用2个字节来表示一个像素。
- 虽然现在来看在显示器上使用光栅技术似乎是很自然的事情，但是在早期，这种做法并不可行，因为在当时看来，这种技术需要的存储器空间太大。在这种情况下SAGE视频显示器应运而生，它是一种矢量 (vector) 显示器，相比电视机，它更像一种示波器。电子枪可以通过电驱动定位到显示器任何一个像素点上，之后可以直接画出直线或曲线。由于屏幕上的图像具有持久性，不会立即消失，这样就可以利用直线和曲线形成最基本的画面。
  - 支持光笔 (light pen) 是SAGE计算机的一大特色，操作者使用光笔可以改变显示器上的图像。光笔这种设备很特殊，从外观上来看是一端连有电线的笔。如果使用与之配套的软件，计算机能够感知到光笔所指的屏幕位置，随即根据光笔的位移相应地改变图像。
  - 光笔的工作原理是什么呢？如果是第一次看到它，即使是相关领域的技术专家，也会感到困惑。理解它的关键在于光笔并不发射 (emit) 光——它所做的是检测 (detect) 光。对于CRT（无论采用的是光栅还是向量显示技术），电子枪移动控制电路有两个最重要的功能，第一个功能是光笔一旦感知到电子枪射出的光，系统需要立即做出反应；第二个功能是系统在对其做出反应的过程中，需要确定出光笔指向的屏幕位置。
- 伊凡·苏泽兰 (Van Sutherland，生于1938年) 是最早预见到了计算机发展的一个全新领域，即交互式计算的人之一。在1963年，他演示了为SAGE计算机专门开发的名为“画板” (sketchpad) 的程序。画板不仅可以将图像信息存放在存储器中，还可以把图像在屏幕上显示出来。你还可以使用光笔在显示器上画出图像并进行修改，与此同时，计算机会对光笔的轨迹一直进行跟踪。
- 还有一位早期交互式计算的预言家，那就是道格拉斯·恩格尔巴特 (DouglasEngelbart，生于1925年)。他曾阅读过1945年万·布什发表的文章《思维之际》，巧合的是，五年之后他开始致力于研究计算机界面显示的新方法，并为之奉献毕生精力。20世纪60年代中期，当恩格尔巴特在斯坦福研究所 (Stanford Research Institute) 工作时，他重新思考并设计了输入设备，提出了用五股 (five-pronged) 键盘作为指令输入设备（这个设备并未普及），另外还提出了一种配备轮子和按钮的设备，它的名字就是鼠标 (mouse)。鼠标现在已经全世界被广泛接受，它可以用来移动屏幕内的指针，还可以选择屏幕上出现的对象。
- 许多在早期热衷于交互式图形计算的科学家（但这里并不包括恩格尔巴特），他们不约而同地聚集在了施乐 (Xerox) 公司，幸运的是，此时的光栅显示器已经变得经济实用。施乐公司在1970年建立了帕洛阿尔托研究中心 (Palo Alto Research Center, PARC)，中心的主要任务之一就是协助产品开发，以此来加快公司迈入计算机产业的步伐。PARC中最著名的预言家应该算是阿伦·凯 (Alan Kay，生于1940年)，14岁那年，阿伦·凯在一篇罗伯特·海因莱因 (Robert Heinlein) 撰写的故事中，读到了万·布什提出的微缩胶片图书馆，阿伦·凯因此而深受启发，不久他构想了一种名为“Dynabook”的便携式计算机。

- PARC着手的第一个大的工程是阿尔托（Alto），它的设计和制造完成于1972-1973年。从那个年代的标准去看，它是一个令人眼前一亮的产品。它采用落地式系统单元，配备16位处理器、2个3 MB的磁盘驱动器、128 KB的内存（最多可扩充到512KB），还包括一个三按钮的鼠标。在Alto开发的时候，16位单芯片微处理器还未面世，所以它的处理器由将近200个集成电路组成。
- Alto有许多与众不同的地方，视频显示器是其中一个方面。屏幕的大小和形状就像一张纸——8英寸宽，10英寸高。它采用光栅成像技术，水平像素值为606，垂直像素值为808，算下来共有489,648个像素。其中每个像素占据1位存储空间，即每个像素取值只有两种：黑色或白色。视频显示的专用存储器容量为64 KB，占用处理器的地址空间。
- 通过直接对视频显示存储器进行写操作，软件可以在屏幕上绘图或将不同字体、不同大小的文本显示在屏幕上。用户可以通过移动鼠标，在屏幕上对指针进行定位，还可以与屏幕上的对象进行交互。视频显示器与电传打字机在很多方面不尽相同，电传打字机顺序响应用户输入并按行将程序输出，而视频显示器的屏幕可以看做二维空间上的高密度的信息阵列，它还可以作为直接的用户输入源。
- 20世纪70年代晚期，Alto所搭配的程序逐渐凸显出很多新奇有趣的特点。比如窗口中可以容纳多个程序并同时显示在屏幕上。Alto的视频图像功能使得软件从文本的束缚中摆脱出来，使其可以更加真实地反映用户的想法。图形对象（Graphical objects，比如按钮、菜单，以及被称做图标的小图片）成为用户接口的一员。鼠标可以在多个窗口中进行选择、触发图形对象来执行程序功能。
- 软件的内涵就在于此，它的意义远不止仅有的用户接口，还包括与用户的亲密耦合。软件使得计算机所涵盖的应用领域变得更广，而不仅仅局限于简单的数字变换。软件之所以被设计出来，其最终目的是——引用道格拉斯·恩格尔巴特在1963发表的一篇著名论文的标题——《为了扩展人类的智慧》（For the Augmentation of Man's Intellect）。
- PARC在Alto这个项目的开发成果预示着**图形用户界面（Graphic User Interface, GUI）**登上了历史的舞台。施乐公司并没有将Alto推向市场（价格定位3万美元以上绰绰有余）。从10年之后的今天来看，当时的Alto应该被包装成一种成功的消费产品并推向市场。
- 1979年，斯蒂夫·乔布斯（Steve Jobs）带领苹果公司代表团对PARC进行了访问，在那里的所见所闻给他们留下了深刻的印象。而他们却花费了三年多的时间才推出具有图形界面的计算机，这就是在1983年1月推出的苹果莉萨（Apple Lisa），可惜这套系统在当时并不被看好。而一年以后推出的麦金托什机（Macintosh）却大获成功。
  - 最原始的Macintosh机配备有Motorola 68000微处理器、64 KB的只读存储器、128 KB的随机访问存储器、一个3.5英寸的磁盘驱动器（存储容量为400 KB）、一个键盘、一个鼠标和一个视频显示器，显示器水平像素为512，垂直像素为342（仅为9英寸的CRT对角线长度），像素总量为175,104个。每个像素赋予1位内存，只能显示黑白两色，这种配置约占22 KB的视频显示存储器。
  - 最原始Macintosh机硬件方面很精巧，但是可更新能力很差。1984年Macintosh操作系统的诞生对于Mac（即Macintosh机）意义非凡，它的出现使得Mac变得与众不同，当时我们把这个操作系统称为系统软件（System Software），它就是现在著名的苹果操作系统（Mac OS）。
  - 基于文本的单用户操作系统，如CP/M或MS-DOS，体积很小但是不支持扩展的应用程序接口（API）。关于这点在第22章进行过解释，在这些基于文本的操作系统中，没有为访问文件系统的应用程序提供一种渠道。Mac OS这种图形化操作系统所占的空间比前面提到的这两种要大得多，其中包含了上百个API函数，每一个函数都用其功能来命名。
  - MS-DOS操作系统是基于文本的，如果要在屏幕上以电传打字机方式将文本显示出来，使用几个简单的API函数即可，但对于Mac OS这种基于图形的操作系统，必须提供一种在屏幕上显示图像的途径，程序通过这条途径对图像进行显示。从理论上来讲，一个API函数完全可以胜任这项任务，函数的功能就是设置某个水平和垂直坐标下的像素的颜色。但在实际应用中，这种方法效率较低以至于严重影响到了图像显示的速度。

- 在这种需求下，如果操作系统可以提供一整套图形编程系统，那么其意义是重大的，这样的操作系统必须包含如下API函数：画线、画矩形、画椭圆（包括圆）以及画出文本。其中，线条可以是实线，可以是虚线，还可以是点线；矩形和椭圆可以具备不同的填充模式；字符可以具备不同字体和大小，还可以具备不同特效，如加粗和下划线等。图形编程系统负责规划如何将各式各样的图形对象以点阵集合的形式表示在显示器上。
- 如果程序在图形操作系统下运行，那么它们在显示器或打印机上画图这一过程中，使用的是一套完全相同的API。正因为如此，字处理程序在屏幕上显示出的文档，与打印出来而得到的纸质文档，看上去非常相似。这种特点称为“所见即所得”（简写为WYSIWYG）。这是喜剧演员弗雷普·威尔森（Flip Wilson）在扮演杰拉尔丁（Geraldine）角色中的一句话，这句话也成了计算机领域的一个经典口号。
- 图形用户界面对用户而言是极具吸引力的，其中一个重要原因就是不同的应用程序使用着大致相同的工作原理，并且影响着用户的使用经验。这样一来操作系统就承担起了支持API函数的重任，而应用程序就可以利用这些API函数去实现用户界面的不同组件，如按钮和菜单等。GUI不仅是一种看上去简洁友好的用户环境，对于程序员而言，它还是一种重要的开发环境。程序员在开发新一代用户界面的时候可以不用从底层开始重新编写。
- 其实早在Macintosh问世之前，一些公司已经开始着手为IBM PC及其兼容机创建图形操作系统。这两种工作有一个显著的不同：苹果公司的硬件和软件都是由苹果公司自己设计的，因此开发人员的工作更加轻松。Macintosh系统软件只支持一种类型的磁盘驱动器、一种视频显示器，以及两种型号的打印机。而IBM PC的图形操作系统开发人员所面对的是许多不同的硬件，操作系统与不同的硬件之间需要同时兼容。
- 还有一点，虽然IBM PC问世的时间（1981年）较早，但MS-DOS应用程序已经在多数人心中根深蒂固，人们不愿意放弃它们。因此PC的图形操作系统必须具备一个重要的特性，那就是新的操作系统应该可以直接兼容MS-DOS应用程序，就好像MS-DOS应用程序是为新的操作系统专门设计的（Macintosh不兼容Apple II系列软件，因为它们的微处理器型号不同）。
- 在1985年，迪吉多科研公司（Digital Research，CP/M的后续公司）推出了图形环境管理器（Graphical Environment Manager，GEM）；VisiCorp（推出VisiCalc软件的公司）推出了VisiOn；与此同时微软公司发布了Windows 1.0版本，作为一匹黑马，当时它也被很多人认为将会成为“视窗争夺战”的胜利者。然而直到1990年3月Windows 3.0发布，Windows才真正受到大众瞩目。星星之火从那时开始燎原。在本书出版的2000年，约90%的小型计算机上使用的都是Windows操作系统。除了外观上的不同，Macintosh和Windows这两种操作系统所包含的API也有着天壤之别。
- 从原理上来分析，除了图形显示器，图形操作系统与文本操作系统相比，对硬件支持的要求并没有太多不同。从理论上来讲甚至硬盘驱动器都可以算是多余的：比如最初的Macintosh没有配备，Windows 1.0也不需要。虽然大家都认为使用鼠标操作会更加方便，但其实Windows 1.0可以不需要鼠标。
- 有一点很容易想到，随着微处理器速度越来越快，内存和外存的容量越来越大，图形用户界面将更加深入人心。图形操作系统将会支持越来越多的特性，它们所占的存储空间也将越来越大。2000年左右的主流图形操作系统通常需要200 MB的硬盘空间和32 MB以上的内存。
- 在图形操作系统中，应用程序几乎都不使用汇编语言来开发。就拿早期的几款操作系统来看，Pascal是Macintosh下的主流开发语言。在Windows操作系统中，C语言一统江湖。还有一个不得不提到的例子，那就是PARC向我们展示的一种全新的方法。大概从1972年开始，PARC的研究员着手开始研发一种名为Smalltalk的语言，这种语言嵌入了面向对象程序设计思想（Object-Oriented Programming），也就是今天的OOP。
- 从传统意义上讲，高级程序设计语言会自然而然地区分出代码（比如以set、for、if这样的关键词开头的语句）和数据，即变量所代表的数字。这种区分毫无疑问来自于冯·诺依曼计算机的体系结构。在这样一种体系结构中，只有两种元素，一种是机器码，一种是机器码所操作的数据。

- 在面向对象的程序设计中，和冯·诺依曼计算机的体系结构所不同的是，对象（object）实际上是代码和数据的组合。在对象内部，与其相关联的代码决定了数据存在的意义，要理解数据的存储方式首先需要理解代码。对象如果需要与其他对象通信，则通过发送或接收消息（message）来实现这一过程，比如一个对象可以通过给另一个对象发送指令来获得相应信息。
- 在图形操作系统的应用程序开发过程中，面向对象语言可以算得上是一种很不错的工具，因为编程人员处理屏幕上的对象（如窗口和按钮等）的过程就是用户感知屏幕元素的过程。举例来讲，假设按钮是面向对象语言中的一个对象。屏幕上的按钮具备一定尺寸和位置，按钮上可以显示文本或小图标，这些都可以抽象成为与对象关联的数据。如果用户通过键盘或鼠标按下按钮，系统就会向按钮对象发送一个表示其被触发的消息，该按钮对象收到消息后就会调用与自身关联的代码进行响应。
- 小型计算机上最流行的面向对象语言是一种对传统的类似于ALGOL语言的扩展，C和Pascal就属于此类。由C扩展的面向对象语言就是赫赫有名的C++（我们可以回忆一下，两个加号放在一起等价于C语言中的自增操作）。C++的核心思想大部分来自于贝尔电话实验室（Bell Telephone Laboratories）的贾尼·斯特劳斯特卢普（Bjarne Stroustrup，生于1950年），最开始C++是作为一种转换程序，它可以把编写的程序转换成C程序（但是转换出的C程序即难看又难以理解）。转换完成之后的C程序可以像普通程序一样编译。
- 其实，面向对象语言能做到的，传统语言也能做到。但是编程终究是人类发明的一种解决问题的活动，面向对象语言使得编程人员多了一种可选的解决方案，这种解决方案具备更加优越的组织结构。如果你想——虽然困难重重——面向对象语言编写的一种程序，并使其在Macintosh和Windows上都可以编译后运行，这是完全可以做到的。此类程序并不直接引用API，使用的是被称为API函数的对象。  
Macintosh 和Windows 使用两种不同的对象定义来编译程序。
- 许多在小型计算机上工作的编程人员已经逐渐不用命令行编译程序，而是使用**集成开发环境**（**Integrated Development Environment, IDE**）。这个环境里集成了所有需要的工具，而环境本身可以像其他图形应用程序一样运行，这样以来大大简化了程序开发任务。还有一种称做可视化编程（Visual Programming）的技术被程序开发人员广泛利用，按钮及其他组件可以通过鼠标拖曳进行“排版”，从而达到在窗口交互设计的目的。
- 在第22章中我们一起讨论过文本文件。为了方便人们阅读，这类文件仅由ASCII字符组成。我们回想一下使用基于文本的操作系统的那个年代，文本文件是应用程序之间进行交流的理想媒介。它的最大优点是可检索性——程序可以检索多个文本文件，然后确定它们中是否有文件包含某一字符串。但如果操作系统中有一种机制用来显示不同字体、大小，以及不同效果比如斜体、黑体和下画线，那么文本文件就不再适用了。很多字处理软件其实都会使用一种自己独有的二进制格式来存储文档。文本文件同样也不适用于图形信息。
- 但我们要清楚的是，与文本相关的信息（比如，字体及段落版式），都可以被编码，而且编码后并不影响其可读性。这种方案的关键是选用一个适当的转换字符来标识出这些信息。在Microsoft设计的**富文本文件格式**（**rich text format, RTF**）中，大括号“{}”以及反斜杠“\”封装了文本的格式信息，RTF也成为了应用程序间传递格式化文本的一种方法。
- PostScript作为一种文本格式，将这种概念发挥到了极致。PostScript的设计者是Adobe系统的创始人之一——约翰·沃诺克（John Warnock，生于1940年）。PostScript是一种通用的图形编程语言，在2000年时主要用在高端计算机的打印机上，用于显示字符或图形。
- 随着硬件性能逐渐提升，价格日渐便宜，图形显示与个人计算环境的融合已是大势所趋。微处理器的处理速度越来越快，存储器价格越来越低廉，视频显示器及打印机分辨率不断增加，而且支持的颜色数目也成千上万，这一切大大推动了计算机图形界发展。
- 计算机图形也逐步产生了两种分支——本章的前面曾提到过这两个词，当时是为了区分图形视频显示器——这两个分支就是矢量和光栅。

- 矢量图形 (vector graphics) 在一些算法的帮助下，利用直线、曲线及填充区域生成图形。这也正是**计算机辅助设计 (Computer-Assisted Drawing, CAD)** 所应用的领域。矢量图形在工程和体系结构设计中有着十分重要的作用。矢量图形一般转化为**图元文件 (metafile)** 格式以存放到文件中。图元文件是由生成矢量图形的一系列绘制命令的集合组成的，这些命令通常已经被编码为二进制形式。
  - 矢量图形的主要工具就是直线、曲线及填充区域。如果你想设计桥梁，使用矢量图形来实现将很简单，但如果要显示桥梁的实际结构，矢量图形就显得无能为力了。对于现实世界里的一副桥梁的整体结构图，用矢量图形来表示将会很复杂，而且困难重重。
  - 光栅图形（也称做位图），就是为了解决这一问题应运而生的。**位图 (bitmap)** 将图像以矩阵阵列的形式进行编码，阵列中的一个单位对应着输出设备上的一个像素点。就像视频显示器一样，位图是一种空间上的概念（可以称其具有分辨率），其图像的宽度和高度都以像素为单位来表示。位图也具备色深（也可叫做颜色分辨率/颜色深度）的概念，色深是指每一个像素被赋予的比特数。位图中每个像素被赋予的比特数相同。
  - 尽管位图从表现形式上看是二维的，但其本身的存储形式却是一串连续的字节——通常从最顶端1行像素开始，紧跟着的是第2行、第3行，等等。
- 有些位图产生于图形操作系统设计的绘制程序，它们都是由某个操作者利用这些程序“手工绘制”出来的，还有一些位图是由计算机代码通过某种算法产生的。如今很多现实中的场景都利用位图来表示（比如数码照片），要把现实世界的图像输入到计算机中，可以借助一些不同的硬件，这类设备一般统称为**电荷耦合器 (charge-coupled device, CCD)**，它是一种在光线下会起电的半导体器件。每个像素都需要一个CCD单元来进行采样。
  - 这些设备中最原始的可以算是扫描仪 (scanner) 了，其原理和影印机类似，都是利用一行CCD扫过需要复印的图像的表面，比如照片。由于光感度不同，不同区域CCD累积的电荷数也不同。扫描仪的配套软件把图像转换成位图存放在文件中。
  - 视频摄像机利用二维CCD单元阵列捕捉图像。通常被捕捉到的图像存放在录像磁带上。但其实视频输出可以直接交给视频帧采集器 (video frame grabber) 去处理，它的工作原理是把模拟信号转换成像素值阵列。帧采集器可以收集通用的视频源，如录像机 (Video Cassette Recorder, VCR) 或激光影碟机 (Laser Disc Player)，甚至可以用于有线电视机机顶盒。
  - 近几年，数码相机的价格已逐渐降低到了家庭用户可以承担的水平，它们从外观上来看和一般相机几乎一样。但数码相机并不使用胶片，而是使用CCD阵列来捕捉图像并直接将其转移到相机的存储器中，之后在适当的时候转储至计算机中。
  - 图形操作系统通常可以将位图文件以某种格式进行存储。Macintosh系统采用Paint格式，之所以叫这个名字是参考了创建这种格式的MacPaint程序（Macintosh的PICT格式同时支持位图和矢量图形，而且是它们的首选格式）。Windows里的默认的格式是BMP，位图文件通常以它作为扩展名。
  - 位图文件可能很大，如果有方法可以让它们变小一些那就再好不过了。这种需求催生了计算机科学中的**数据压缩 (Data Compression)** 这一全新领域。
  - 假设我们正在处理一幅每个像素占3位的图像，这种图像在本章前面曾讲过。这张图片上出现的画面是一片天空、一栋房子和一块草坪。因此，图片中可能有大片的蓝色和绿色。很可能位图的最上面一行出现了72个蓝色像素。如果有一种方法可以表示蓝色像素连续且重复了72次，那么通过这种方法表示的位图文件将会比原先的小很多。这样的压缩方法称为**游程长度编码 (Run-Length Encoding)**，即RLE。
    - 通常办公室的传真机采用的就是RLE压缩方法，压缩过程一般在传真机通过电话线传送图像之前。由于传真机展现出的图片都是黑白两色，没有灰度和彩色，所以通常像素值都会有很长串的白色区域，适合使用RLE压缩。

- 这十多年里风光无限的位图文件格式是图形交换格式 (Graphics InterchangeFormat) 即GIF，由计算服务 (CompuServe) 公司于1987年开发。GIF文件所采用的压缩技术称为LZW，LZW源自其三位创建者的名字：Lemplel、Ziv和Welch。LZW比RLE更加强大，因为它所考虑的是像素值的模式 (patterns)，而RLE针对的是具有相同值的像素串。
- RLE和LZW都属无损 (lossless) 压缩技术范畴，因为可以从压缩数据中重新生成完整的初始文件。专业一点的说法是，压缩过程是可逆的 (reversible)。可逆压缩方法并不适用于所有类型的文件，这点不难证明。在某些情况下，采用这些方法“压缩”后的文件比初始文件还要大！
- 近几年来看，有损 (lossy) 压缩技术大行其道。有损失的压缩是不可逆的，这是由于部分原始数据在压缩过程中被丢弃了。有损压缩技术不应该用于电子报表或文字处理文档，因为在这些重要文档里面少一个数字或者字母都会“失之毫厘，谬以千里”。但对于压缩图像，这些损失还是可以接受的，部分数据的损失不会使图片的整体效果有太大的变化。这就是为什么有损压缩技术的思想起源于心理视觉的原因，心理视觉领域所探究的是人的视觉，并根据心理因素确定人们所看到的景象中哪些比较重要而哪些不重要。
- 在JPEG中，人们使用了一系列具有重大意义的位图有损压缩技术。JPEG代表的是联合图像专家组 (Joint Photography Experts Group)，它涵了几种压缩技术，其中一些是无损的，另一些是有损的。
- 把图元文件转换成位图文件的方法很简单。这是由于视频显示存储器与位图在概念上保持一致。如果程序能把图元文件画在视频显示存储器中，则它也能在位图上画出图元文件。
- 但从位图文件到图元文件的转换却不那么容易，如果位图文件过于复杂甚至会导致无法转换。为了解决此类问题，人们发明了一项技术，那就是**光学字符识别 (Optical Character Recognition)**，或简称为OCR。对于位图上出现的一些字符（来自于传真机，或来自于页面扫描），如果需要转换成ASCII码，那么OCR就会派上用场。OCR软件会对比特流进行模式识别，然后确定其代表的字符。由于算法的复杂性，OCR软件并不能保证百分之百准确。虽然不很准确，但是OCR软件一直尝试将手写的字符也转换成ASCII码字符。
- 位图和图元文件都是数字化的可视信息。同理，音频信息也能转换成比特和字节。
- 随着1983年激光唱片 (compact disc) 的问世，数字化音响掀起了一轮消费狂潮，它同时也成为了最成功的电子消费品案例。CD是由飞利浦和索尼公司联合开发出的产品，一张直径为12 cm的CD可存储74分钟的数字化声音。而这个74分钟的时长因为贝多芬的第九交响曲 (Beethoven's Ninth Symphony) 刚好可以存储在一张CD上。
- CD中声音信息采用的编码技术被称为**脉冲编码调制技术 (Pulse Code Modulation)**，简称为PCM。名字听起来有点故弄玄虚，但从概念上讲PCM其实是一种很简单的过程。
- 振动是声音之源。人们发出的声音来源于声带的振动，大号的声音也来自于振动，森林里的树倒下的声音归根结底也是振动，还有很多例子，它们的振动来源于空气分子的移动。空气在被推拉的过程中，有时压缩有时放松，有时向后有时向前，这个过程每秒钟可以达到成百上千次，最终使耳膜产生振动，从而我们能够听到声音。
- 声波可以被模拟，一个成功的例子就是1877年爱迪生发明的第一台电唱机，它利用锡箔圆桶表面上的隆起和凹陷部位录制和播放背景音乐。直到CD的出现，这种录制技术才发生了稍许变化，圆桶变成了光盘，锡箔变成了塑料。早期的电唱机是机械化的，但是后来，电子放大器被用来放大声音。声音可以通过麦克风上配备的可变电阻转换成电流，喇叭中的电磁铁又可以将电流转换为声音。
- 代表声音的电流与先前所讲过信号不同，本书之前讨论过的在“连通——断开”之间跳变的1/0数字信号。声波的变化是连续的，因而产生电流的电压也是如此。在这里电流产生的目的是模拟 (analog) 声波。为了达到这个目的，我们使用了一种新设备，通常把它叫做**模拟数字转换器 (Analog To Digital Converter, ADC)** ——所有的功能集成在了一个芯片上——将模拟电压转换成二进制数表示。一定长

度的数字信号将会被ADC所输出——通常长度为8、12或16个比特——它们组合在一起表明了电压的相对级别。如果ADC的转化长度为12比特，那么电压值的取值范围为000h ~ FFFh，这样一来就可以区分出4096个不同的电压级别。

- 在一种名为脉冲编码调制（Pulse Code Modulation）的技术中，以电压形式表示的声波将以恒定的频率被转换成数值。而这些数值将以小孔的形式刻在光盘表面，通过这种方式，电压就以数值的形式被存储在CD上。要读取这些信息时，可以通过分析从CD表面反射的激光读取到所存储的数值。在播放声音的时候这些数值又被转换成电流，这一过程利用到了数字模拟转换器（Digital-To-Analog converter，即DAC，DAC还可以用在彩色图形板上，将像素值转换成模拟信号并传输至显示器）。
- 声波电压在恒定的频率下被转换成了数字，该频率被称为采样率（sampling rate）。1928年，贝尔电话实验室的哈里·奈奎斯特（Harry Nyquist）通过证明得到了一个总要结论：采样频率应至少为被采样信号（即被记录和播放的信号）最大频率的两倍。人类可以听到的声音的频率范围通常为20 ~ 20,000Hz。CD使用的采样频率为每秒44,100次，比人类听觉范围最大频率的两倍还要大一些。
- CD中存储的声音的动态变化范围决定了每次采样的比特数，这个范围就是CD存储声音的最高与最低频率之差。这难免给人感觉有些复杂：电流通过不断变化来模拟声波，其达到的最高峰称为声波的振幅（amplitude）。我们感知到的声音强度是振幅的两倍。1贝尔（bel，这个名称来源于Alexander Graham Bell最后一个单词的三个字母），代表着强度的10倍；1分贝（decibel）代表着1贝尔的十分之一。1分贝代表着人类所能感知到的声音最小强度变化。
- 如果每次采样大小为16比特，这样就能够表示96分贝的动态范围，这一范围的下限是刚好能听到的声音的阈值（低于这一值的声音是听不到的），而上限就是人们承受最大负荷声音的极限阈值。CD光盘的每个采样点就是用16比特表示的。
- 综上所述，CD光盘中每秒产生44,100个采样样本，每个样本占据2个字节。有时你还希望享受立体声效果，这样的话采样信息需要翻倍，则每秒总共需要176,400字节，算下来每分钟需要10,584,000个字节（这是个庞大的数字，正因如此，20世纪80年代前数字记录声音的方法并没有普及）。74分钟的立体声CD需要字节数为783,216,000。
- 相对于模拟声音，数字化声音的优点不言而喻。特别是在模拟声音被复制的时候（例如把录音磁带转录成电唱片）难免会有些失真。而对于用数字形式表示的数字化声音而言，都可以实现无失真的转录或复制。细心观察的人会发现，过去在电话通话中，信号传输线路越长则声音越糟。这种情况已经一去不复返了，因为现在大部分电话系统都已经数字化了，即使你打跨国电话，听筒中的声音也像在对街对话一样清晰。
- CD可以存储声音，也可以存储数据。专门用来存储数据的CD统称为CD-ROM（CD只读存储器），通常CD-ROM最大存储容量约为660 MB。如今许多计算机中都配备CD驱动器，而应用程序及游戏软件都可以存储在CD-ROM中。
- 声音、音乐、视频逐渐走入个人计算机是在大约10年前，当时统称为多媒体（multimedia），虽然这几年发展很迅速，但这个名字已经普遍使用开来，所以也就不再需要新起一个特别的名称了。如今的家用计算机都配备声卡，声卡中包含一个ADC，它可以将麦克风传输的模拟声音信号转储成为数字信号，此外还包括一个DAC，它的作用是帮助扩音器播放录制的数字声音。声音还可以按照波形文件（waveform files）方式存放于磁盘。
- 在我们使用家用计算机录制和播放声音的时候，其实对声音质量要求并不高，一般不会苛求达到CD的效果，所以Macintosh和Windows这两种操作系统提供了较低的采样频率，尤其是22,050 Hz、11,025 Hz和8,000 Hz这三种频率，采样信息量也保持在较少的8位，并且使用了频度录制手段。声音录制时所占的存储容量也减少到了每秒8000字节，这样算下来每分钟约占480,000字节。
- 很多人在科幻电影及电视剧集中看到过这样的场景，未来的计算机正在用纯正的英语与用户进行交互。只要计算机配备了录制和播放数字化声音的硬件，那么实现这个目标不过是一个软件问题。

- 如果要使计算机能够用易于理解的单词和句子与人交谈，有很多种方法。一种方法是预先录制句子、短语、单词还有数字，然后将它们以文件的形式存储，之后使用多种形式将其糅合在一起。这种方法常见于电话公司的信息系统中，在这种环境下只需要有限的单词、数字及其组合，因此这种方式能适用。
- 有一种更加通用的声音合成方法，它把ASCII码字符转换成波形数据。例如在英语拼写方面，由于存在很多不一致性，软件系统需要使用一个词典或复杂的算法来确定单词的准确发音。一个完整的单词由基本的音节（也叫做音素，phonemes）组成。一般情况下软件都需要做一些其他方面的调整。如果一个句子后紧跟着的是问号，则最后一个单词的发音频率必须相应提高。
- 语音识别（voice recognition）——把波形数据转换成ASCII码字符——这个是一个极其复杂的过程。其实许多人在理解口语化的方言时也有很多困难。在使用个人计算中的语音处理软件时，通常需要对软件进行样本训练，以便软件能尽量准确地转录某个人的话语。这中间涉及的问题已经大大超出了ASCII码文本转换的范畴，从本质上来讲是一个利用编程技术使得计算机“理解”人类语言的过程。这类问题正是人工智能（artificial Intelligence）所研究的领域。
- 当今计算机中的声卡还配备了小型电子音乐合成器，它能模仿128种不同的乐器和47种不同的打击乐器，这类设备被称做MIDI合成器。MIDI即乐器数字接口（Musical Instrument Digital Interface），这项发明出现于在20世纪80年代早期，由一家电子音乐合成器制造协会推出，主要用于将电子乐器组合起来，并且连到计算机上。
- MIDI合成器的类型不同，它们合成乐器声音的方法也就不同，有一些方法的效果更加逼真。MIDI合成器所展现出来的特性已经超越了MIDI所定义的范畴。但其实它的功能就是以演奏声音的方式来响应短消息序列——长度通常为1、2或3个字节。MIDI消息本身就指明了所需要的乐器、将要演奏的音符，或正在演奏的音乐的休止符。
- MIDI文件不仅包含了一系列MIDI消息集合，还包括了时间信息。通常一个MIDI文件“麻雀虽小，五脏俱全”，包含了整套演奏信息，因此可以由计算机上的MIDI合成器完整地演奏。相比于包含同样一段音乐的波形文件，MIDI文件通常要小得多。就文件的相对大小而言，如果把一个波形文件比作位图文件，则MIDI文件就好像矢量图元文件。MIDI文件的缺点来源于MIDI合成器的异构性：同样一个MIDI文件可能在一个合成器上完美演奏，但在另一个合成器上可能不堪入耳。
- 数字化电影是多媒体的另一片天地。把一系列静止图像快速播放，可以达到电影和电视图像中出现的物体移动效果。我们把这期间出现的单个图像称为帧（frames）。电影的播放速率为24帧/秒，北美的电视节目为30帧/秒，世界上大部分地方的电视为25帧/秒。
- 计算机中的电影文件一般都是由一系列附带声音的位图组合而成。但是如果未经压缩处理，一部电影文件中的数据量将会很大。假如电影中每一帧包含的像素大小是640×480，每个像素为24位真彩色，那么每一帧的大小就为921,600字节。如果播放速度为30帧/秒，则每秒需要的存储空间为27,648,000字节。照这样计算下去，每分钟需要的空间大小为1,658,880,000字节，一部两小时的电影需要199,065,600,000字节，大约200 GB。正因如此，我们的计算机上播放的电影经常很短、解析度很小，而且清晰度很差。
- 就像JPEG压缩技术可以用来减少静态图像所占的数据空间一样，MPEG压缩技术用于处理动态电影文件。MPEG全称是移动图像专家小组（Moving Pictures Expert Group）。动态图像压缩技术基于的是一种客观事实，即每一帧继承了前一帧的大部分信息，也就是说存在冗余信息。
- 针对不同的多媒体产品有不同的MPEG标准。MPEG-2用于高清晰度电视（high-definition television，HDTV）及数字影音光盘（digital video discs，DVD）。DVD也叫数字多用光盘（digital versatile discs），其物理大小与CD一样，但是DVD的两面都可以记录数据而且每一面有两层。在DVD中，视频信息可以压缩为原始大小的五十分之一，因此前面提到的一部两小时的电影将占据4 GB的空间，而且只占据其中一面的一层。所以一张具有两面四层的DVD盘容量可达到16GB，容量达到了CD的25倍。如果不料，在未来的某一天，DVD将替代CD-ROM成为新的软件存储媒介。

- 随着CD-ROM和DVD-ROM的出现，这是否意味着万·布什预言的麦克斯储存器在今天成为了现实？最原始的麦克斯储存器设想中，使用的原材料是缩微胶片，但显然CD-ROM和DVD-ROM更合适担此重任。由于电子媒体易于检索，它们比物理媒体更具有优越性。可惜同时访问多个CD或DVD驱动器并不现实。我们的存储设备有一点与万·布什所提出的概念有所不同，那就是所有信息并不一定要触手可及，真正使它们达到信息共享的做法是计算机互连，这样做还可以更有效地利用存储空间。
- 对计算机进行公开性远程操作的第一人是乔治·史帝比兹（George Stibitz），也正是他，在20世纪30年代设计了贝尔实验室的继电器计算机。对继电器计算机的远程操作的演示地点在达特茅斯，时间是在1940年。
- 电话系统在线路上传输的是声音，而不是比特。在电话线路上传输比特需要先将其转换成声音，传输完之后再转换回比特。单一频率和振幅的连续声波（统称为载波，carrier）无法表达完整清晰的信息。但如果对声波进行一些调整——说得专业一点就是，对声波进行调制（modulate）使其反映两种不同的状态——通过这种方式可以表示出0和1。将比特与声波进行互相转换的设备被称做**调制解调器**（modem，它包括调制和解调两个功能）。调制解调器以串口（serial）形式工作，因为字节中的单个比特是一个接一个传输的，而不是一拥而上（打印机一般通过并行接口与计算机相连：整个字节由8根线并行传输）。
- 早期调制解调器采用了频移键控（frequency-shift keying，FSK）技术。假设调制解调器的处理速度为300 bps，而且二进制数0被调制到1070 Hz，而1被调制到1270 Hz。每个字节被夹在一个起始位和一个停止位中间，所以每个字节其实占据了10位空间。在300 bps的传输速率下，每秒传输的字节数为30个。现在许多采用先进技术的新一代调制解调器速度超过了它的100倍。
- 早期家用计算机的狂热爱好者可以使用计算机和调制解调器建立**电子公告牌系统**（Bulletin Board System，BBS），其他计算机可以接入到这个系统中并下载（download）文件，这意味着文件从远程计算机中传输到了自己的计算机。这些概念发展广泛，甚至于扩展到了大型信息服务领域中，例如线上资料库服务（CompuServe）。在大多数环境下，通信中采用的一般都是ASCII码。
- Internet与这些早期的发明有本质上的差别，因为它是一种非中心化的系统。Internet从本质上讲是一组协议的集合，这些协议是计算机之间相互通信的保证。众多的协议中，最重要的当属TCP/IP协议，它包括了**传输控制协议**（Transmission Control Protocol，TCP）和**网际协议**（Internet Protocol，IP）。这种协议使得传输过程变得规则化，不再是简单地通过线路传输ASCII码字符，而由基于TCP/IP协议的传输系统把大的数据块分割成小的包（packets），之后在传输线上（通常是在电话线上）独立发送，最后在传输线路的另一端将数据重新组装起来。
- Internet中最流行的是万维网（World Wide Web），而这一部分与图形联系最紧密。万维网采用了HTTP协议来支持其工作，HTTP（Hypertext Transfer Protocol）即超文本传输协议。几乎所有在Web页面上看到的数据都遵循一定格式，那就是HTML（Hypertext Markup Language），即超文本标记语言。其中超文本（hypertext）这个单词用来描述相关链接信息，非常类似于万·布什预言的麦克斯储存器。HTML文件可以包含指向其他Web页面的链接，这样可以轻松访问其他页面。
- HTML与本章前面讨论过的富文本格式（RTF）很相似，它们都含有带有格式信息的ASCII码文本。HTML也可包含多种图片格式，例如：GIF文件、PNG（portable network graphics，便携式网络图像格式）文件，以及JFIF（JPEG文件交换格式）等。大部分万维网浏览器都支持浏览HTML文件，这是文本格式的优势所在。易检索性也是HTML被定义成文本文件的另一个优点。虽然名字可能有些让人迷惑，但HTML与我们在第19章和第24章讲到的语言不同，它并不是真正的程序设计语言。Web浏览器首先读取HTML文件，根据读取到的内容显示文本和图形并编排它们的格式。
- 当我们浏览某些Web页面并进行一些操作时，有一些特殊程序需要并发执行，程序中的代码可以运行在服务器端（Server，用来存储原始Web页面的计算机）或客户端（Client），我们自己的计算机就是客户端。服务器端责任重大，通常要完成一些重要的处理工作（例如解释客户端填写的在线表格），服务器端的工作可以通过公共网关接口（Common Gateway Interface，CGI）脚本来处理。而对于客户

端，HTML文件可以包含简单的程序设计语言，例如著名的JavaScript。Web浏览器可以对JavaScript语句进行解释，就像解释HTML文本一样。

- 为什么Web站点不为我们的计算机提供一个可执行程序，如果这样的话问题一下子就解决了。要回答这个问题，我们首先要明确：我们的计算机是什么样的？如果使用的是Macintosh机，那么这个可执行程序需要包含Power PC机器码，而且需要引用Mac OS中的API函数；如果是一台PC兼容机，那么这个可执行程序需要包含Intel Pentium机器码，而且需要使用Windows API函数。但计算机及图形操作系统的种类繁杂，还有许多其他类型的计算机和图形操作系统。进一步来说，我们也不想毫无目的地下载可执行文件，它们很可能来自于非信任站点而且会带有恶意行为。
- 上述问题的答案就在Sun公司开发的Java语言中（请勿与JavaScript混淆）。Java是一款成熟的面向对象程序设计语言，和C++有些类似。前面章节中我们讨论过**编译语言（compiled languages，可产生包含机器码的可执行文件的语言）**和**解释语言（不可产生可执行文件的语言）**之间的区别，Java是一种介于两者之间的语言。它需要经过编译，但编译的结果不是机器码，而是**Java字节码（Java byte codes）**。Java字节码与机器码在结构上很相似，但Java字节码可以在一种虚拟的计算机下被解释，即**Java虚拟机（Java Virtual Machine，JVM）**上。被编译的Java程序产生Java字节码，之后计算机模拟JVM对其进行解释。Java程序的运行可以不受限于机器与图形操作系统的类型，所以它具有平台无关性（platform-independent）。
- 关于如何利用电流在线路上传输信号和信息，在讲述这一点时本书利用了大段章节，但其实有一种更加行之有效的方法，那就是利用光纤——一种由玻璃或聚合体制造的光导纤维，通过从不同角度对光进行反射达到光传输效果。光信号在光纤中的传输速率可以达到千兆赫兹——即每秒十亿个比特。
- 展望未来，在以后的家庭和办公室中，光子似乎要替代电子承担海量信息传输的重任。比起莫尔斯码中的一“点”一“划”，比起为了与一街之隔的好友深夜交流而绞尽脑汁想出的闪光灯，光子的速度无与伦比。