第十一章

数字逻辑

11.1布尔代数

11.2门

11.3组合电路

布尔函数的实现

多路复用器

  译码器

只读存储器

加法器

11.4时序电路

Flip-Flops

寄存器

  计数器

11.5可编程逻辑器件

可编程逻辑阵列

现场可编程门阵列

11.6关键术语和习题

**学习目标**

学习本章后，你应该能够：

* 理解**布尔代数**的基本运算。
* 区分不同类型的flip-flops触发器。
* 使用卡诺图来简化布尔表达式。
* 概述**可编程逻辑器件**。

数字计算机的操作是基于二进制数据的存储和处理的。贯穿本书，我们假设存储元件可以同时以两种稳定状态之一存在，同时电路在控制信号控制下，可以操作二进制数据来实现各种计算机功能。在本章中，我们将讨论如何实现在数字逻辑中实现这些存储元件和电路，特别是组合电路和时序电路。本章首先简要回顾布尔代数，它是数字逻辑的数学基础。其次，将介绍门的概念。最后，将描述由**门**构成的组合电路和时序电路。

## 11.1布尔代数

  数字计算机和其他数字系统中的数字电路的设计以及行为分析，是通过一种称为**布尔代数**的数学学科完成的。这个名字是为了纪念英国数学家乔治·布尔（George Boole），他在1854年的论文《逻辑与概率数学理论的思想法则研究》*（An Investigation of the Laws of Thought on Which to Found the thematical Theories of Logic and Probabilities）*中提出了这门代数的基本原理。1938年，麻省理工学院电气工程系研究助理克劳德·香农提出，布尔代数可以用来解决继电器开关电路设计中的问题[SHAN38]1。香农的理论后续被用在分析和设计数字电子电路上。布尔代数在以下两个方面应用起来十分方便：

* **分析**：它在描述数字电路功能时十分经济实用。
* **设计**：对于给定的函数，布尔代数可以简化该函数的实现。

  与任何其它的代数一样，布尔代数中有变量和操作。只是在布尔代数中，变量和操作是逻辑变量和逻辑操作。其中，变量可以取值1（真）或0（假）。基本逻辑操作是AND, OR和NOT，它们一般用点、加号和上划线2表示。

A AND B ＝ A · B

A OR B ＝ A + B

NOT A =

  当且仅当两个操作数都为真时，AND操作后值才为true（二进制值1）。如果操作数中有一个为真或两个都为真，OR操作结果都为true。一元运算NOT会反转它的操作数的值。例如，考虑方程

D = A + ( · C)

1这篇论文可在box.com/COA10e中查阅。

2逻辑NOT通常用一个撇号来表示：NOT A = A′。

如果A = 1，或者B＝0同时C＝1，则D等于1。否则D等于0。

关于符号有几个要点需要注意。在没有括号的情况下，AND操作的优先级可以直接简单连接，而不用点运算符来表示。因此：

A + B · C = A + ( B · C ) ＝ A + BC

等式两边都表示：B与C求与运算，然后将结果和A求或运算。

  表11.1a以真值表的形式定义了基本逻辑操作，真值表列出了操作数的每个可能的取值组合，并给出操作结果。该表还列出了三个其他的有用的操作符：XOR、**NAND**和**NOR**。当且仅当两个操作数中一个操作数为1而另一个为0，其异或（XOR）值才为1。NAND函数是AND函数的补函数(NOT)，NOR是OR的补函数：

A NAND B = NOT (A AND B) ＝ AB

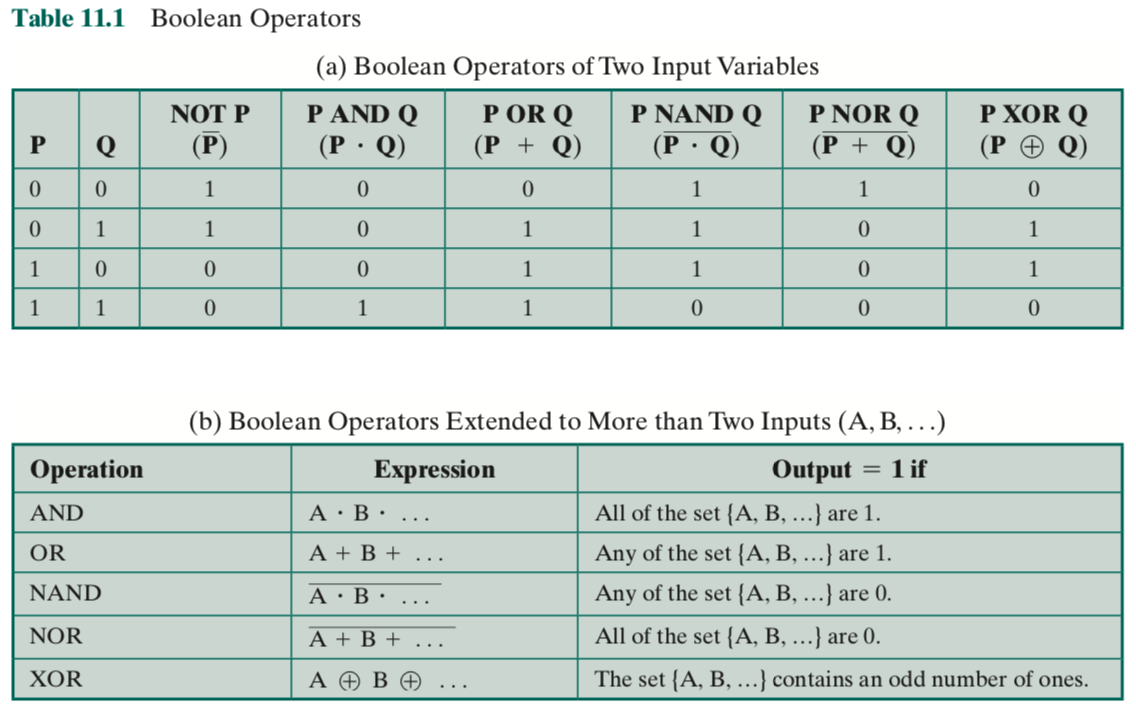
A NOR B = NOT (A OR B) ＝ A + B

 正如我们将看到的，这三个新的操作符在实现某些数字电路时是很有用的。

  除了NOT之外，逻辑操作可以推广到两个以上的变量，如表11.1b所示。

  表11.2总结了布尔代数的主要运算法则。等式被分为两列，用来显示AND和OR操作的互补性或对称性。表中有两类恒等式：基本公式（或公理），它们并没有证明，而其他恒等式可以通过基本公式导出。这些公式定义了解释布尔表达式的方式。这两个分配律中的一个值得注意，因为它不同于我们在普通代数中的法则：

A + (B · C) = (A + B) · (A + C)

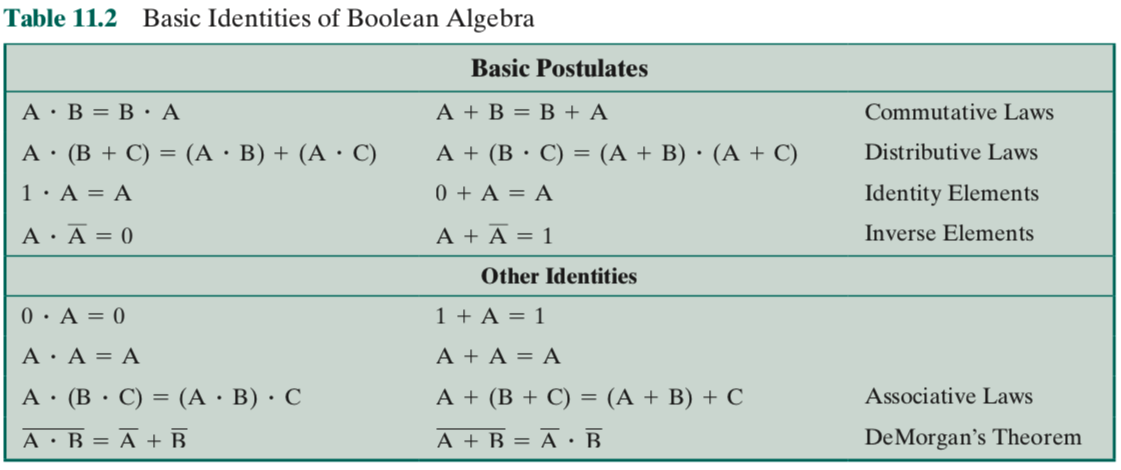


  这两个最底部的表达式被称为德摩根定律。我们可以将它们重述如下：

A NOR B = AND

A NAND B =  OR

  请读者通过将真值（1和0）替换为变量A、B和C来验证表11.2中的表达式。

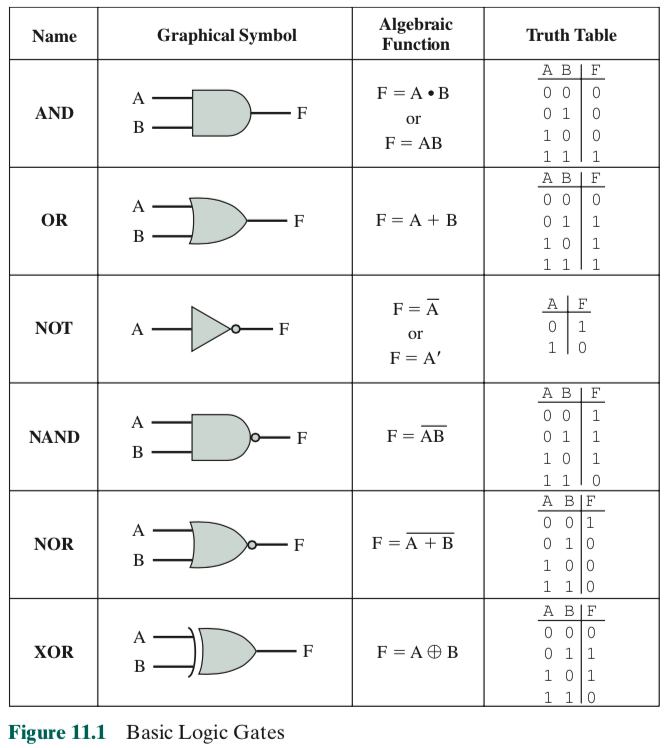


## 11.2 门

所有数字逻辑电路的基本组成部分是门。逻辑功能是通过门的互连实现的。

门是一种电子电路，它会对其输入信号进行简单的布尔运算，从而产生一个输出信号。数字逻辑中使用的基本门有AND、OR、NOT、NAND、NOR和XOR。图11.1展示了这六个门电路。每个门以三种方式定义：图形符号、代数符号和真值表。本章使用的符号来自IEEE标准IEEE Std 91。注意，反转（NOT）操作用圆圈表示。

  图11.1所示的每个门具有一个或两个输入和一个输出。然而，正如表11.1b所示，除了NOT之外的所有门都可以有多于两个输入。因此，(X + Y + Z)可以通过有三个输入的单个**OR门**来实现。当改变输入端一个或多个值时，正确的输出信号几乎在瞬时出现，仅被信号通过门的传播时间(称为门延迟)延迟。这种延迟的重要性在第11.3节中讨论。在某些情况下，一个门可以有两个输出，一个输出是另一个输出的反转。



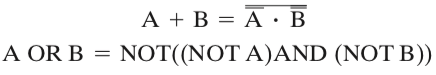
  这里我们介绍一个通用术语：我们说**断言**一个信号就是使信号线从逻辑假（0）状态转换到逻辑真

 （1）状态。逻辑真（1）状态要么是一个高电平状态，要么是一个低电平状态，这取决于电子电路的类型。

通常，并非所有门类型都被用于实现。如果只使用一种或两种门，设计和制造比较简单。因此，识别功能完整的门的集合非常重要。这意味着任何布尔函数都可以只使用集合中的门来实现。以下是功能完整的门的集合：

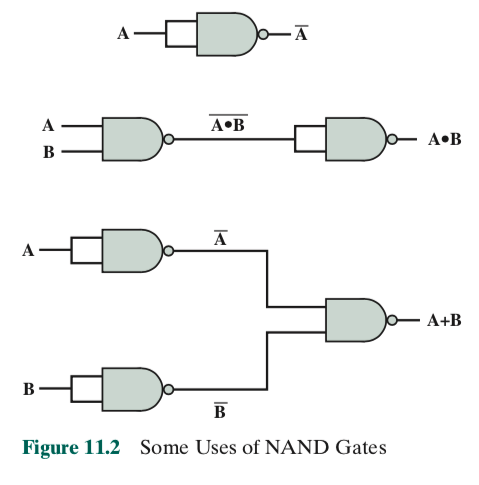
* AND, OR, NOT
* AND, NOT
* OR, NOT
* NAND
* NOR

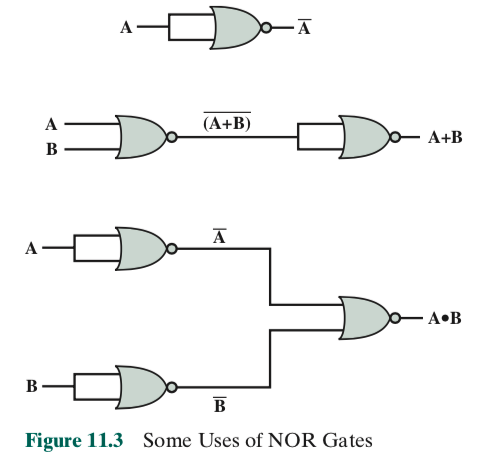
应该清楚的是，AND、OR和NOT门构成一个功能完整的集合，是因为它们代表布尔代数的三个运算。为了让AND和NOT门形成一个功能完整的集合，必须有一种方法来从AND和NOT操作合成OR操作。这可以通过运用德摩根定理来完成：



类似地，OR和NOT操作在功能上是完整的，因为它们可以用来合成AND操作。

图11.2显示了如何仅用NAND门实现AND、OR和NOT函数，图11.3显示了用NOR门实现的相同的情况。由于这个原因，数字电路可以也经常仅用NAND门或仅用NOR门实现。





有了门，我们已经达到了计算机硬件的最原始的电路级别。对用于构造门的晶体管组合的研究则脱离了该领域，而进入了电气工程领域。然而，对于我们的要求，我们只需要描述清如何使用门作为构建块，从而实现数字计算机的基本逻辑电路即可。

## 11.3组合电路

**组合电路**是一组相互连接的门电路，其输出在任何时候都只是当前输入的函数。与单个门一样，输出几乎紧接着输入而出现，只有门延迟。

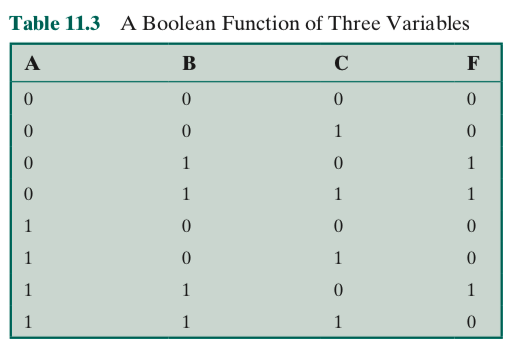
  一般来说，组合电路由n个二进制输入和m个二进制输出组成。与门一样，组合电路可以通过三种方式定义：

* **真值表**：对于输入信号的2n个可能的组合中的每一个，列出m个输出信号的二进制值。
* **图形符号**：描述了互连的门的陈列布局。
* **布尔方程**：每个输出信号被表示为输入信号的布尔函数。

### 布尔函数的实现

  任何布尔函数都可以以电子形式实现为门的网络。对于任何给定的功能，都有多种可相互替代的实现方式。考虑由表11.3中的真值表表示的布尔函数。我们可以通过简单地逐项列出使得F为1的A, B, C组合的值:

F = B + BC +AB (11.1)



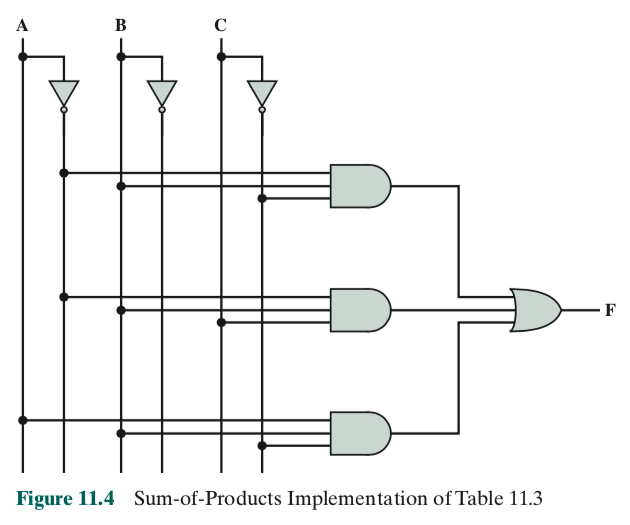
有三种输入值的组合可以使F为1，如果这些组合中的任何一个出现，结果都是1。由于显而易见的原因，这种表达形式被称为**结果的和**（sum of products, 或SOP）形式。图11.4给出了AND、OR和NOT门的直接实现方式。

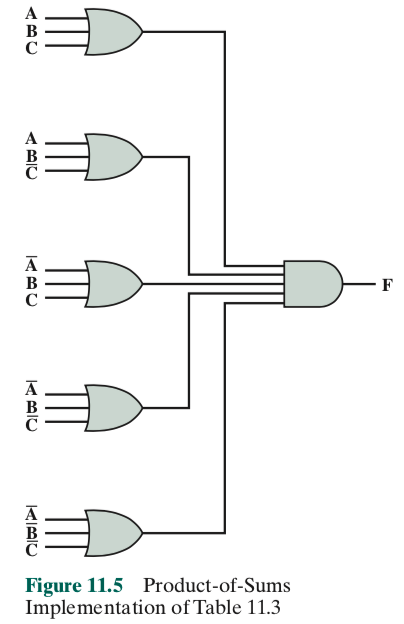
  另一种形式也可以从真值表中导出。如果产生1的任何输入组合为真，则SOP形式表示输出为1。如果产生0的输入组合都不为真，那么也可以说输出为1。因此，



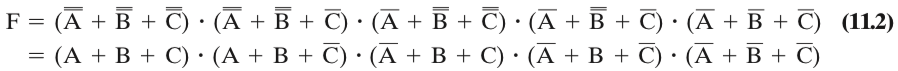
  该式可以使用德摩根定理的推广来重写：







因此，



  这是**结果的和（POS）**的形式，如图11.5所示。为了清楚起见，没有显示非门。而是假设每个输入信号及其补信号可用。这简化了逻辑图，并使得对门电路的输入更加明显。

  因此，布尔函数可以通过SOP或POS的形式实现。在这一点的选择上，似乎取决于真值表输出函数是否包含更多的1或0：SOP对于每个1有一个项，而POS对于每个0有一个项。然而，还有其他需要考虑的因素：

* 通常，从真值表推导出的布尔表达式可能比通过SOP或POS推导出的要简单。
* 最好使用单个门类型(NAND或NOR)来实现功能。

  第一点的意义在于，使用更简单的布尔表达式，可以使得实现该函数需要的门更少。有三种可以用来简化表达式的方法：

* 代数简化
* 卡诺图
* 奎因-麦克鲁斯基表

代数简化 代数简化指应用表11.2的恒等式来将布尔表达式简化为元素较少的表达式。例如，再次考虑等式（11.1）。一个等价的表达式是：

F ＝ B + B （11.3）

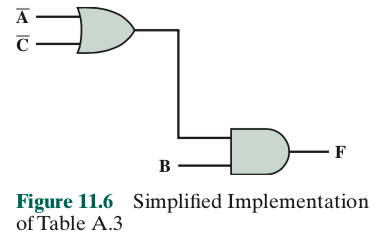
 或者，甚至更简单：

F = B ( + )

  此表达式可以用图11.6所示方式实现。方程（11.1）的简化主要是通过观察完成的。对于更复杂的表达式，需要一些更系统的方法。

卡诺图 为了简化，**卡诺图**是表示少量（最多4个）变量的布尔函数的一种方便的方法。该映射是一个由2n个正方形组成的阵列，表示n个二进制变量值的所有可能的组合。图11.7a显示了一个由两个变量组成的函数的四个正方形的卡诺图。为了以后使用，我们按照00、01、11、10的顺序列出组合结果。因为与组合相对应的方块被用于记录信息，所以组合结果通常写在方块之上。在三个变量的情况下，表示的是八个正方形的排列（图11.7b），其中一个变量的值写在左边，其他两个变量的值在正方形的上面。对于四个变量，需要16个正方形，其排列如图11.7c所示。

  卡诺图可以用以下方式表示任何布尔函数。每个正方形对应于乘积和形式的唯一结果，其中1值对应于变量，0值对应于该变量的非。因此，结果A对应于图11.7a中的第四个正方形。对于函数中的每个这样的结果，1被放在相对应的正方形中。因此，对于两个变量的例子，图对应于A+B。给定布尔函数的真值表，构造卡诺图是一件容易的事情：对于真值表中产生1结果的变量值的每个组合，将其在卡诺图中对应的正方向中填充1。图11.7b显示了表11.3的真值表的卡诺图结果。要将布尔表达式转换为卡诺图，首先必须将表达式转换为范式：表达式中的每一项必须包含每个变量。例如，对于等式（11.3），我们必须首先把它展开成等式（11.1）的完整形式，然后再把它转换成卡诺图。



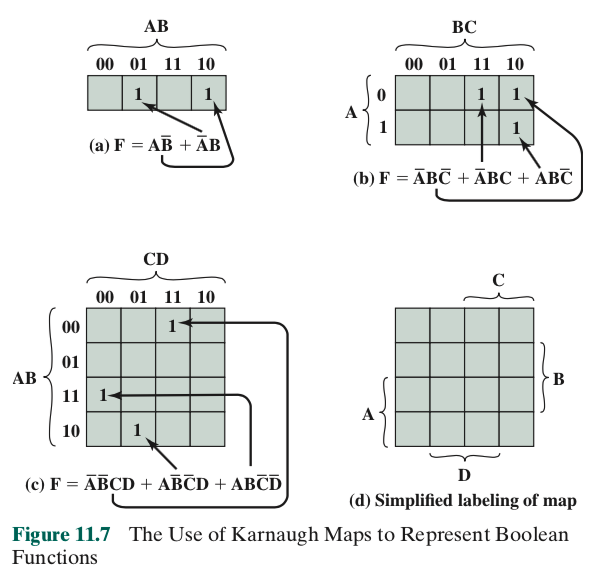


图11.7d中使用的标签强调了变量与卡诺图的行列之间的关系。在这里，符号A包含的两行是变量A具有值1的行；符号A不包含的行是变量A值为0的行；B、C和D的情况与A类似。

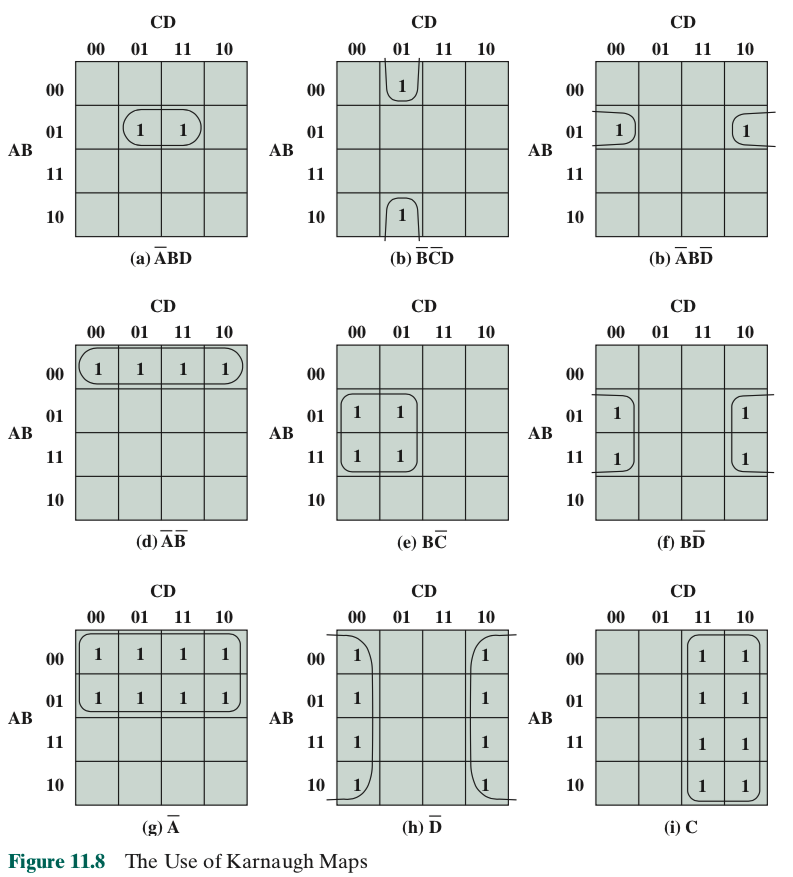
  一旦创建了函数的卡诺图，我们通常可以通过记录卡诺图上1的排列来为它编写简单的代数表达式。其原理如下。相邻的两个正方形只有一个变量不同。如果两个相邻的正方形都有一个1，则相应的结果项仅在一个变量上不同。在这种情况下，可以通过消除该变量来合并两个项。例如，在图11.8a中，与ABCD和ABCD两项对应的两个相邻的正方形。因此，该式可以合并为：

BD + BCD ＝ BD

这个过程可以通过几种方式来扩展。首先，邻接的概念可以扩展到卡诺图的边缘。因此，同一列顶部的正方形与底部正方形是相邻的，同时同一行的最左边正方形与最右边的正方形也相邻。这些情况显示在图11.8b和c中。其次，我们不仅可以分组2个正方形，而且可以分组2n个相邻正方形（即2, 4, 8个等等）。图11.8中的接下来的三个例子显示了4个正方形的分组。注意，在这种情况下，可以消除两个变量。最后三个例子表示了8个正方形分组的情况，这里可以消除三个变量。

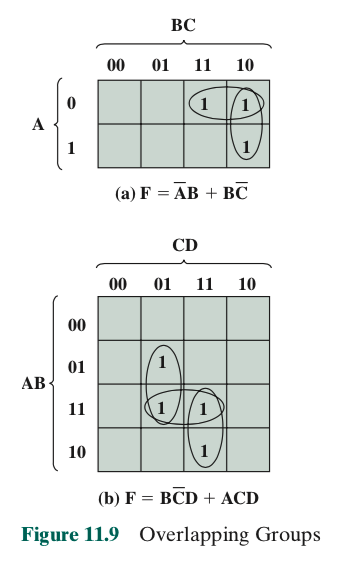
  我们可以将简化规则归纳如下：

1. 在被标记的正方形（写有1的正方形）中，找到那些由1、2、4或8个小正方组成的唯一的最大块的正方形，并将这些块圈起来。



1. 选择尽可能大、数量尽可能少的由被标记正方形组成的其它块，但至少包括每个标记方形一次。在某些情况下，结果可能不是唯一的。例如，如果一个有标记的方块正好与其他两个方块结合，并且没有第四个有标记的方块来完成一个更大的组，那么就需要选择这两组中的一个。在圈出分组的时候，允许多次使用相同的1值。
2. 继续圈出单个被标记的正方形，或相邻的被标记的正方形对，或四个、八个构成的组等等，使得每个被标记的正方形都属于至少一个圈；然后使用尽可能少的这些块来包含所有标记的正方形。

  基于表11.3的图11.9a说明了简化过程。如果在分组之后仍然存在任何孤立的1，那么这些孤立的1将被圈为1的组。。



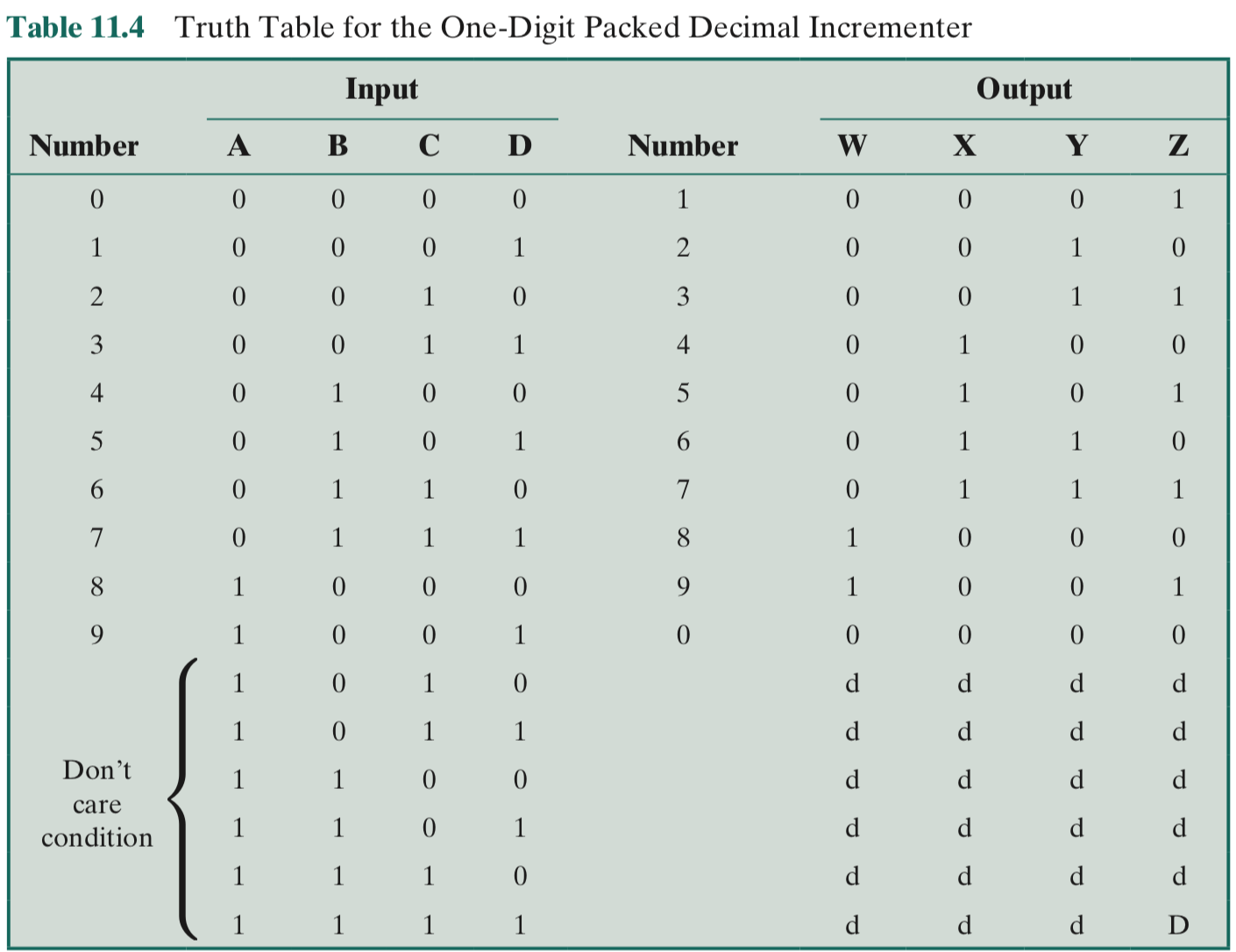
最后，在从卡诺图到简化的布尔表达式之前，可以消除所有已经被其他组完全重叠的由1组成的组。如图11.9b所示。在这种情况下，水平组是冗余的，在创建布尔表达式时可以忽略它。

  卡诺图还有一个重要特点。在某些情况下，变量值的某些组合不会出现，因此也不会出现相应的输出。这些被称为“不关心”的情况。对于每个这样的情况，字母“d”被输入到卡诺图的相应正方形中。在进行分组和简化时，每个“d”都可以被当作1或0，哪一个都可以得到最简表达式。

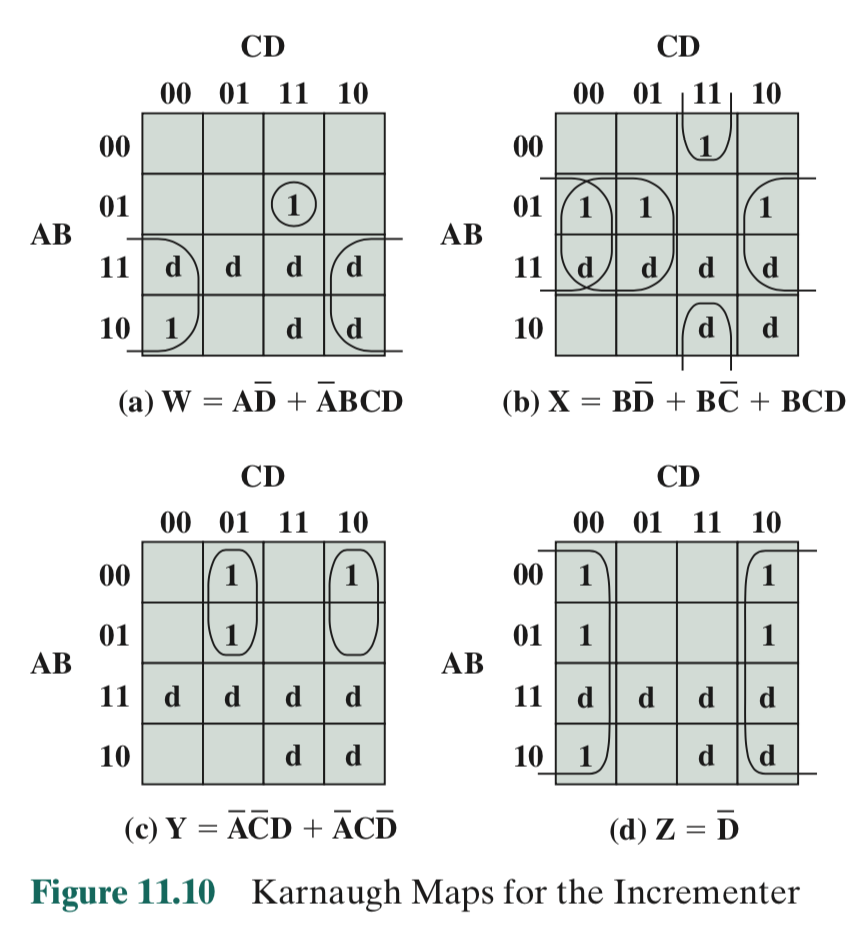
  在[HAYE98]中给出的例子说明了我们讨论的要点。我们想要得到一个电路的布尔表达式，它把1加到一组十进制数上。对于这组十进制数，每个十进制数都用简单的4位代码表示。从而，0＝0000, 1＝0001，… ，8 ＝1000, 9＝1001。剩下的的4位值，从1010到1111，并没有使用。此代码也被称为**二进制编码十进制码（BCD码）**。

  表11.4显示了用于产生比4位BCD码的输入要多一的4位结果的真值表。这里加法的结果需要模10。因此，9 + 1 ＝ 0。此外，请注意，输入代码中有6个会产生“不关心”结果，因为这些不是有效的BCD输入。图11.10显示了对应于每个输出变量的卡诺图结果。写有d的方块被用来达到最佳的可能分组。

*QUINE-MCCLUSKEY*方法 对于四个以上的变量，卡诺图方法变得越来越麻烦。对于五个变量，需要两个16×16的卡诺图，其中一个图是在三维空间上位于另一个图之上，从而实现临接关系。六个变量则



需要在四位空间上使用四个16 × 16的表！另一种方法是一种表格技术，称为Quine–McCluskey方法。该方法适合于在计算机上编程，自动给出自动产生最简布尔表达式。



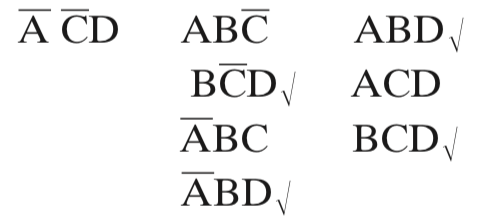
该方法最好通过一个例子来说明。请考虑以下表达式：

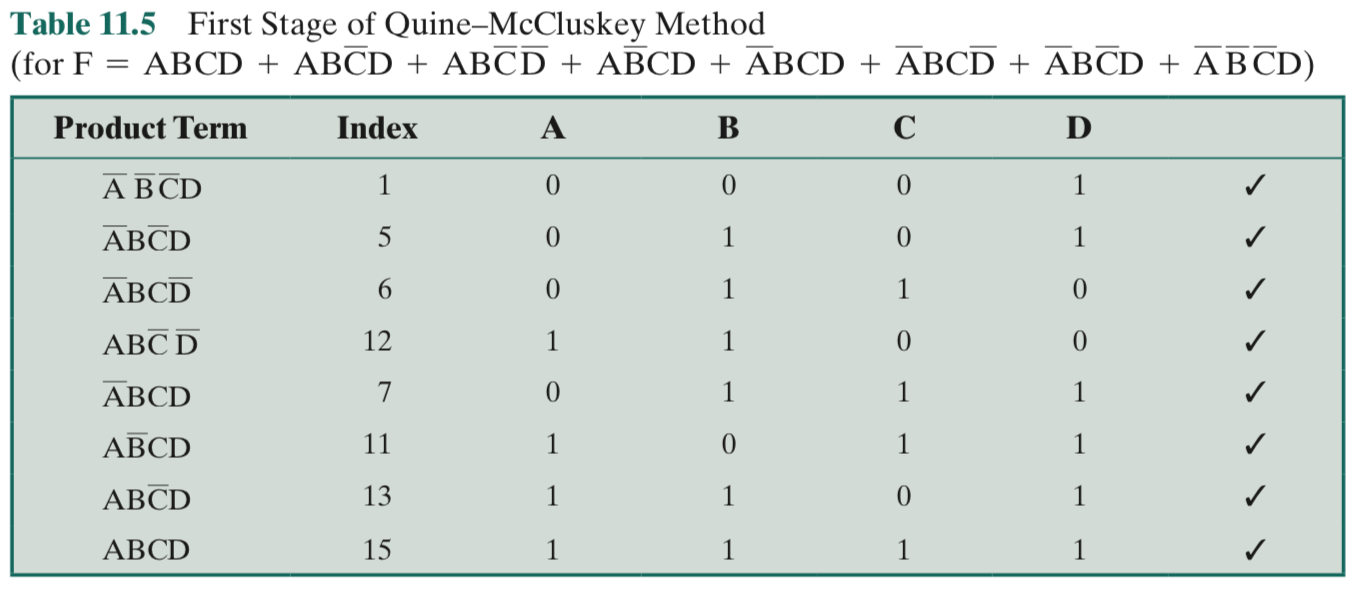


我们假设这个表达式是从真值表导出的。我们希望生成一个适合于用门电路实现的最简表达式。

第一步是构造一个表，其中每一行对应于表达式的乘积项之一。这些项按照补充变量的数量进行分组。也就是说，我们从没有补充变量的项开始，如果存在，那么再处理有一个补充变量的项，以此类推。表11.5展示了该实例表达式的列表，其中的每一行表示一个分组。为了清楚起见，每个项由每个非补充变量的1和每个补充变量的0来表示。因此，我们将每一项按照它们包含的1的个数进行分组。索引列表示该行对应的十进制数值，我们在后面的内容中会用到。

  下一步是查找仅有一个变量不同的所有项对，也就是说，项对中除了有一个变量的值不同（在两项中一个是0，一个是1）项对中其它变量的值都一样。根据每一项的分组方式，我们可以从第一组开始，将第一组的每一项与第二组的每一项进行比较。然后将第二组的每一项与第三组的每一项进行比较，以此类推。每当找到匹配项时，在每个项旁边画一个对号，通过消除两个项中不同的变量来组合该项对，并将其添加到新的列表中。因此，比如说，项BC和BCD结合起来产生BC。这个过程一直持续到遍历完整个原始表。得到一个新表，有以下项：





正如所指出的，新表以与第一个表相同的方式被组织成组。然后也以与第一个表相同的方式处理第二个表。也就是说，只对照找出只有一个变量中不同的项，并为第三个表产生一个新的项。在这个示例中，生成的第三个表只包含一个项：BD。

  一般来说，该过程将遍历连续的表，直到生成没有匹配的表。在这种情况下，需要用到三个表。

  一旦刚刚描述的过程完成，我们就消除了表达式中许多可能的项。如表11.6所示，那些尚未消除的项被用于构造一个矩阵。矩阵的每一行对应于迄今为止使用过的任何表中尚未被消除（没有画对号）的一项。每一列对应于原始表达式中的一项。在行和列的每个交叉点都写上一个X，这样使得行元素与列元素“一致”。也就是说，行元素中的变量与列元素中的变量具有相同的值。接下来，圈出一列中单独的X。然后，在任何一个包含有被圈X的行中的每一个X的四周放一个正方形。如果现在每个列都有一个正方形或一个被圈的X，那么我们就完成了任务，并且那些X已经被标记的行元素构成了最简表达式。因此，在我们的例子中，最终的表达式是

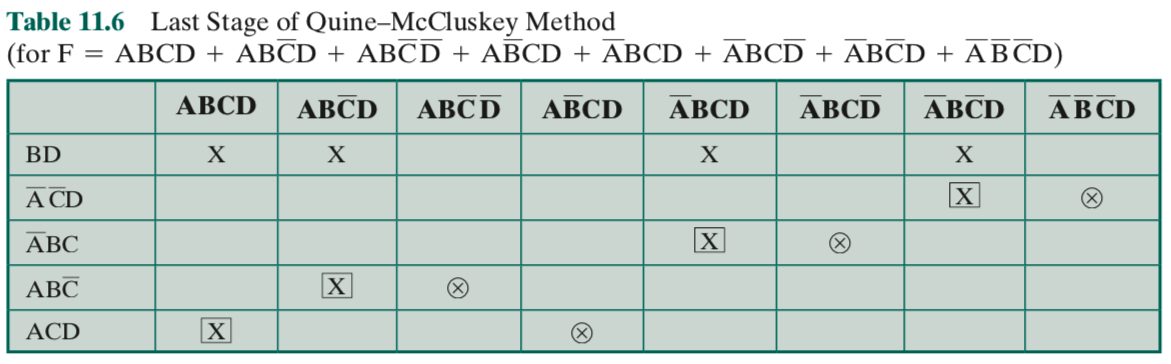
AB + ACD + BC + D

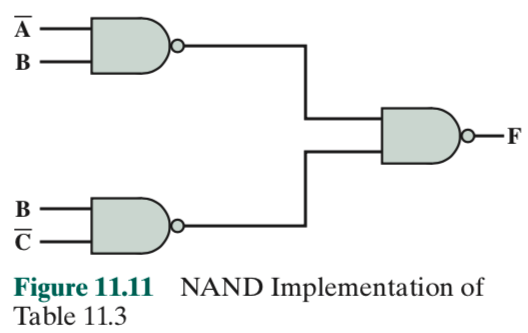
  在某些列既没有画圈也没有画正方形的情况下，需要额外的处理。基本上，我们一直添加行元素，直到覆盖所有列。

  让我们总结一下Quine-McCluskey方法，试图直观地证明它的工作原理。操作的第一阶段相当简单。该过程消除了结果项中不需要的变量。因此，表达式ABC+AB相当于AB，因为：

ABC + AB ＝ AB(C + ) ＝ AB

  在消除变量之后，剩下的表达式显然与原始的表达式等价。然而，这个表达式中可能存在冗余项，正如我们在卡诺图中发现的冗余分组一样。矩阵的布局确保了原始表达式中的每个项都被覆盖，并且是以一种最小化最终表达式中项的数量的方式被覆盖。





NAND和NOR实现 实现布尔函数时需要考虑的另一个问题是所使用的门的类型。有时希望只用NAND门或只用NOR门来实现布尔函数。虽然这可能不是最小门实现，但优点是实现起来很规律，从而可以简化制造过程。再考虑等式（11.3）：

F = B( + )

因为值的非的非值就是原始值，



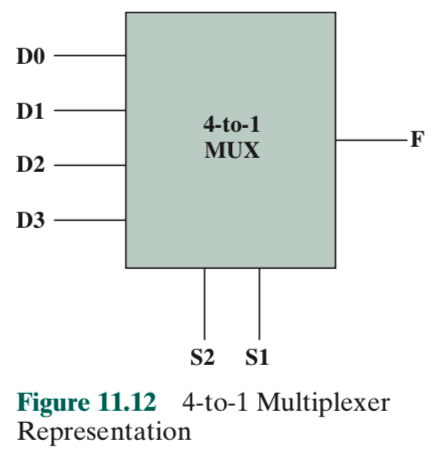
  应用德摩根定律：

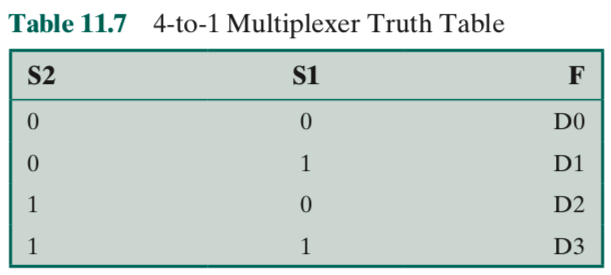


  它有三个NAND形式，如图11.11所示。

### 多路复用器

**多路复用器**将多个输入连接到单个输出。在任何时候，输入信号中的一个会被选择传递到输出端。图11.12显示了一个通用的框图表示。这代表一个4到1多路复用器。有四个输入行，标记为D0、D1、D2和D3。其中一条线被选择到输出信号F。为了选择四个可能的输入中的一个，需要2位的选择

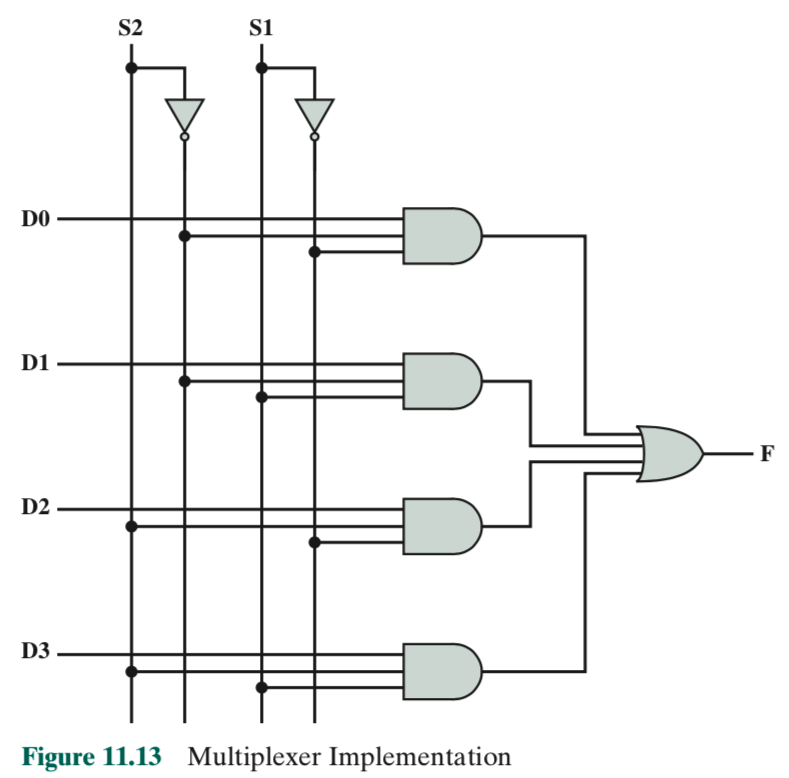


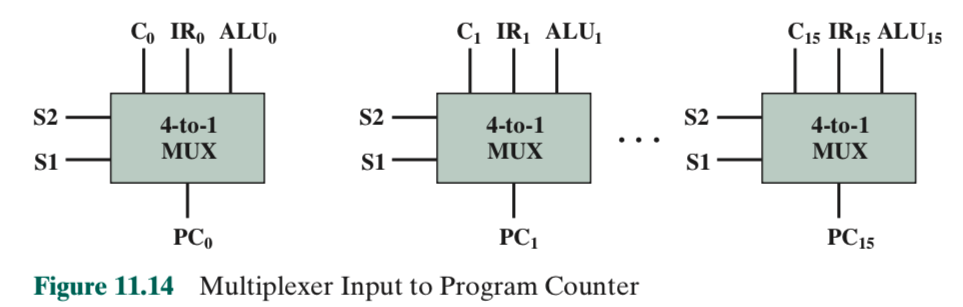


码，并且通过标记为S1和S2的两条选择线来实现。

  示例4到1多路复用器可由表11.7中的真值表定义。这是真值表的简化形式。它没有显示输入变量的所有可能的组合，而是将输出显示为来自行D0、D1、D2或D3的数据。图11.13显示了一种使用与门、或门和非门的实现方式。S1和S2以这样的方式连接到与门：对于S1和S2的任意组合，三个与门输出0。第四个**与门**将输出所选行的值，该值是0或1。因此，到或门的三个输入总是0，并且或门的输出将等于所选输入门的值。使用这种规则的组织，很容易构造大小为8到1、16到1等等的多路复用器。

  在数字电路中，多路复用器被用来控制信号和数据路由。一个例子是**程序计数器**（PC）的加载。要加载到程序计数器中的值可以来自几个不同资源中的一个：





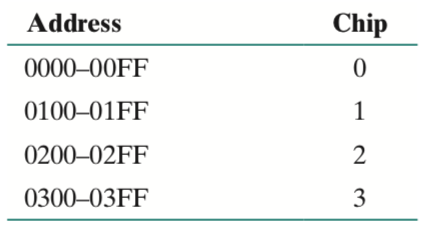
* 一个二进制计数器，如果PC需要递增用于下一个指令。
* 指令**寄存器**，如果使用直接地址的分支指令刚刚被执行。
* ALU的输出结果，如果分支指令使用位移模式指定地址。

  这些不同的输入可以连接到多路复用器的输入线上，PC连接到输出线上。选择线决定哪个值要被加载到PC中。因为PC包含多个位，所以需要使用多个多路复用器，每比特一个。图11.14显示了用于16位地址的情况。

### 译码器

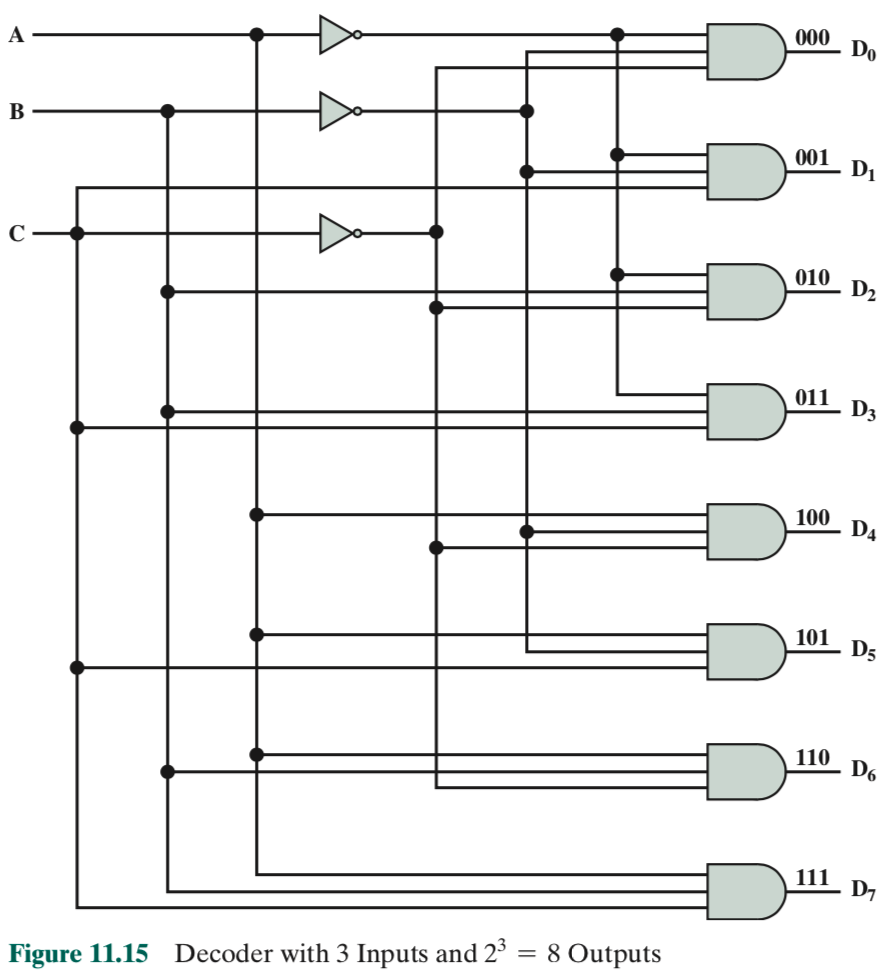
**译码器**是具有多个输出线的组合电路，其中在任何时候，只有一个输出线被断言。具体哪个输出线被断言取决于输入线的模式。通常，解码器具有n个输入和2n个输出。图11.15显示了一个具有三个输入和八个输出的译码器。

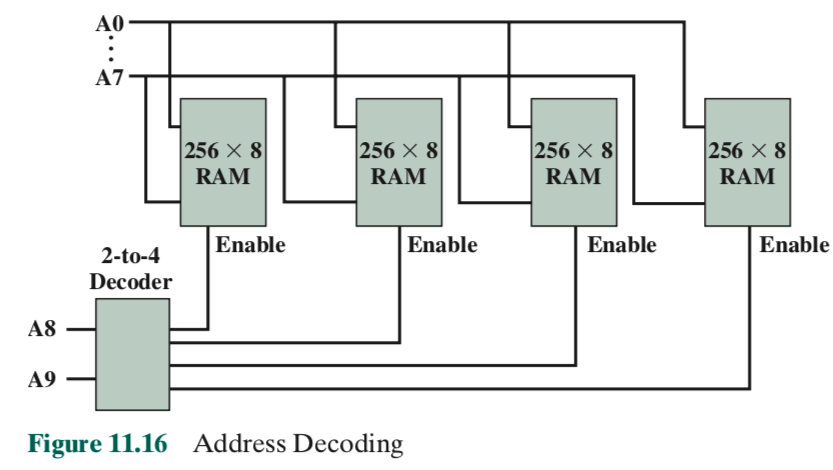
译码器在数字计算机中有许多用途。一个例子是地址译码。假设我们希望用四个256 × 8位RAM芯片构造出一个1K字节的存储器。我们想要一个统一的地址空间，可以分解如下：

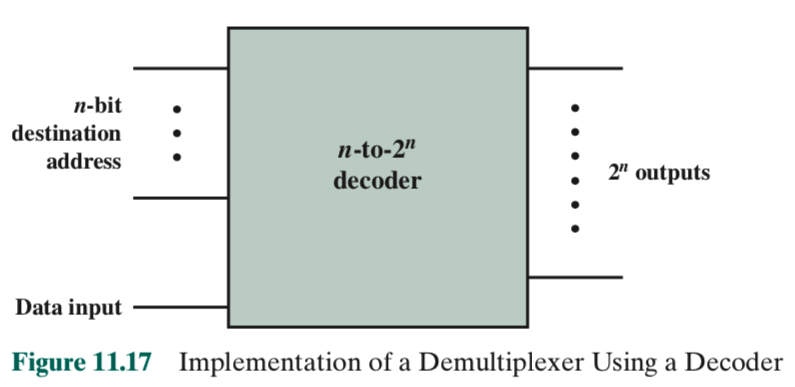


每个芯片需要8个地址线，这些地址是由地址的低8位提供的。10位地址的高2位用于选择四个RAM芯片中的一个。为此目的，使用2到4译码器器，其输出使能四个芯片中的一个，如图11.16所示。

通过附加的输入线，译码器可用作解复用器。解复用器执行多路复用器的逆函数，它将单个输入连接到多个输出之一。如图11.17所示。如前所述，n个输入被译码来产生2n个输出中的一个。所有2n个







输出线均与一个数据输入线相与。因此，n个输入作为地址来选择特定的输出线，并且数据输入线(0或1)上的值被路由到该输出线。

  图11.17中的配置可以用另一种方式来理解。将新行上的标签从数据输入更改为使能。这使得可以对解码器的定时进行控制。只有当存在编码输入并且使能线具有值1时，才会出现译码输出。

### 只读存储器

组合电路通常被称为“无记忆”电路，因为它们的输出仅取决于它们的当前输入，并且不保留先前输入的历史。然而，存在一种用组合电路实现的存储器，即**只读存储器（ROM）**。

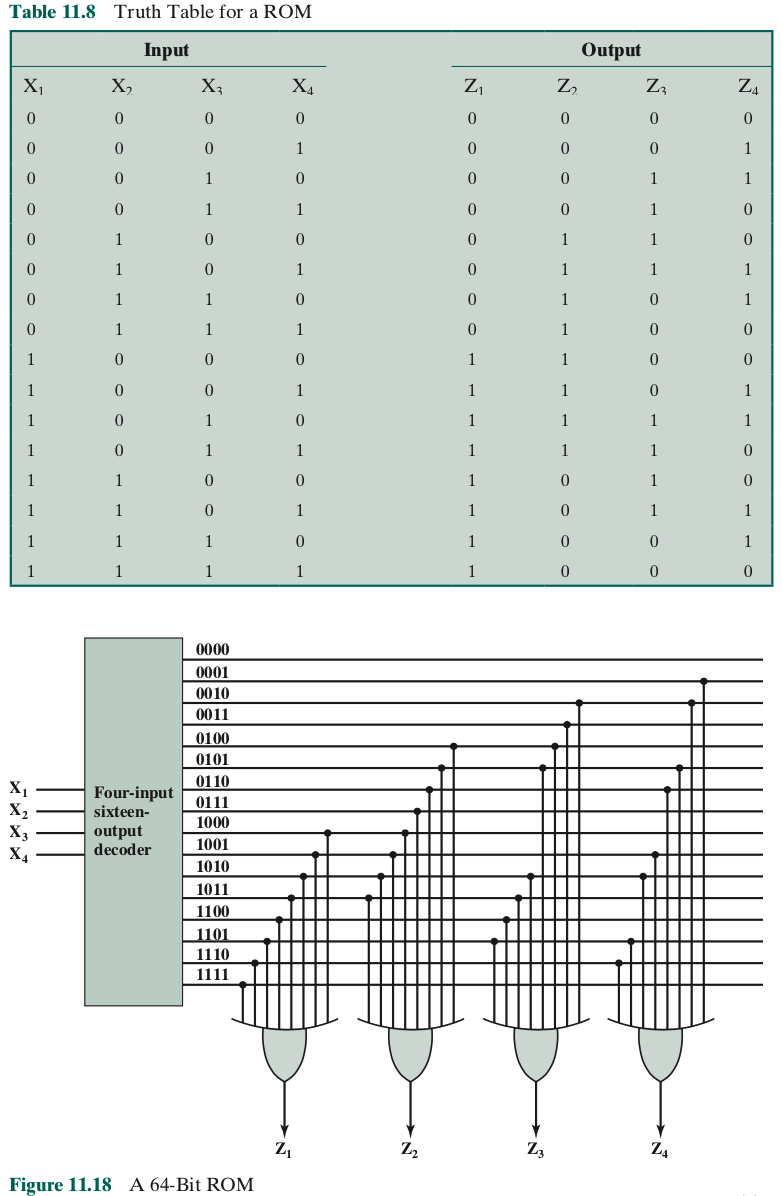
  回想一下，ROM是只执行读取操作的存储单元。这意味着存储在ROM中的二进制信息是永久性的，并且在制造过程中被写入。因此，ROM的给定输入（地址线）总是产生相同的输出（数据线）。因为输出只是当前输入的函数，所以ROM实际上是一个组合电路。

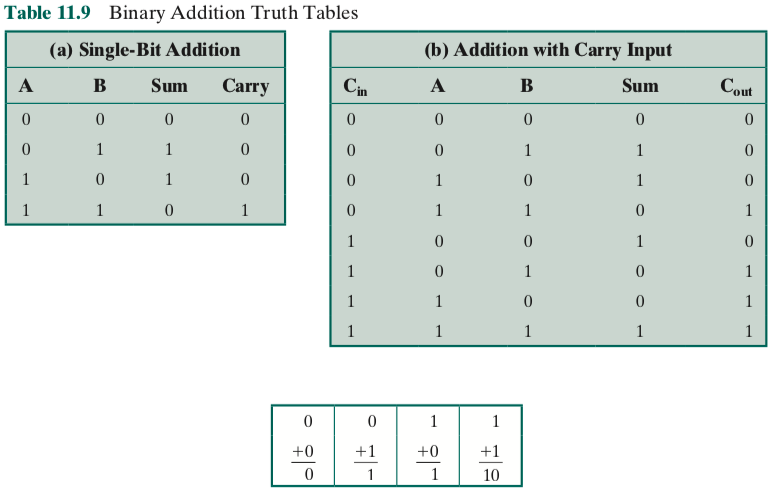
  ROM可以通过译码器和一组或门来实现。举个例子，考虑表11.8。这可以看作是一个具有四个输入和四个输出的真值表。对于16个可能的输入值中的每一个，都显示了相应的输出值的集合。它也可以被看作是定义一个64位ROM的内容，它由16个字组成，每个字长4位。四个输入指定地址，四个输出将指定地址所指位置上的内容。图11.18显示了如何使用4到16译码器和4个或门来实现这个存储器。与PLA一样，使用常规的组织，并且进行互连来给出期望的结果。

### 加法器

到目前为止，我们已经看到如何利用互连门电路来实现诸如信号路由、译码和ROM等功能。一个至今还没有提到的重要部分是运算。在本小节的概述中，我们将介绍加法功能。

  二进制加法与布尔代数的不同之处在于其结果中包含进位项。如下：





  然而，加法仍然可以用布尔项来处理。在表11.9a中，我们显示了相加两个输入位来产生1位求和结果和一个进位的逻辑。这个真值表可以很容易地用数字逻辑实现。然而，我们对只对单个比特执行的加法不感兴趣。我们更希望对两个n位数字求和。这可以通过将一组加法器放在一起来完成，这样来自一个加法器的进位可以作为下一个加法器的输入。图11.19中描述了一个4位加法器。

  为了使多位加法器工作，每个单比特加法器必须有三个输入，包括来自下一低阶加法器的进位。修改后的真值表见表11.9b。两个输出可表示如下：

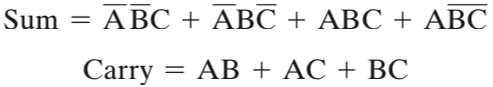
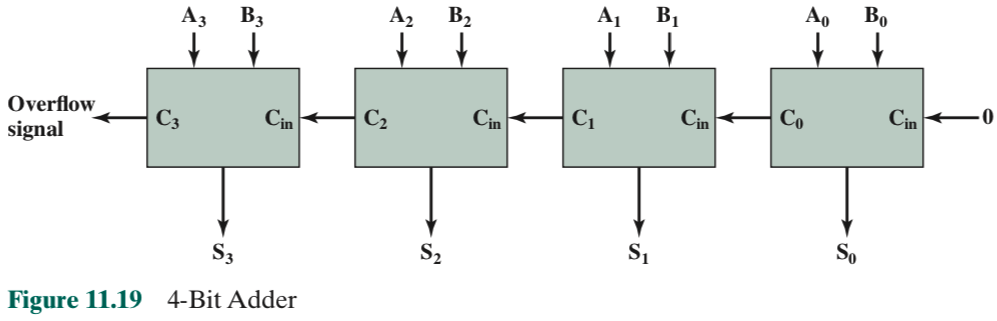
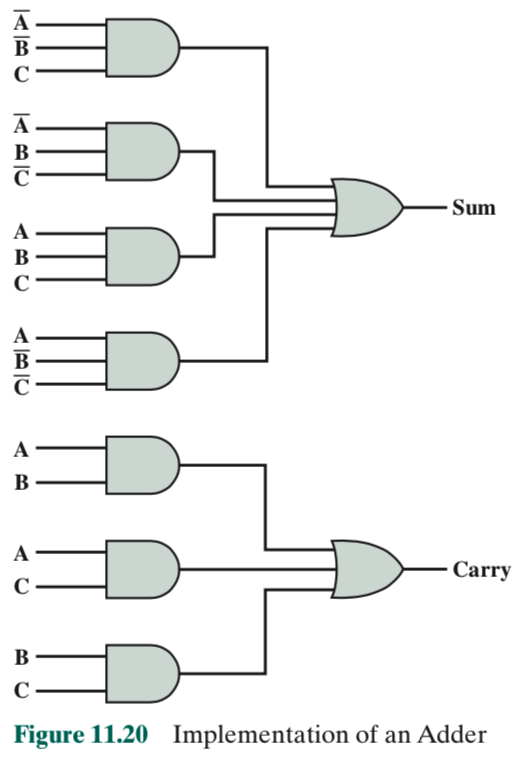


  图11.20显示了使用与、或、非门的一种实现：

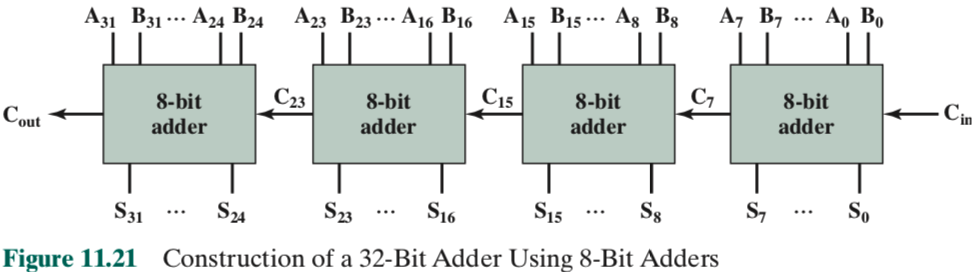




  因此，我们有了实现多位加法器的必要逻辑，如图11.21所示。注意，因为每个加法器的输出取决于前一个加法器的进位，所以会增加从最小有效位到最高有效位的延迟。每个单比特加法器都会有一定量的门延迟，并且这个门延迟会累积。对于较大的加法器，累积延迟可能高的不可接受。

  如果不通过等待前面阶段按序的进位输出，就能确定进位值，那么每个单比特加法器就可以独立地工作，并且延迟不会累积。这可以通过被称为进位前瞻的方法来实现。让我们再次使用4位加法器来解释这种方法。

我们想得到一个表达式，该表达式可以在不参考以前的进位值的情况下，指定到加法器的任何阶段的进位输入。我们有



  C0 = A0B0 （11.4）

C1 = A1B1 + A1A0B0 + B1A0B0 （11.5）

按照同样的程序，我们得到

C2 = A2B2 + A2A1B1 + A2A1A0B0 + A2B1A0B0 + B2A1B1

+ B2A1A0B0 + B2B1A0B0

  这个过程可以在任意长加法器上重复。每个进位项可以表示为SOP形式，仅作为原始输入的函数，而不依赖于进位。因此，无论加法器的长度如何，仅发生两级门延迟。

  对于长数据，这种方法会变得过于复杂。计算n位加法器的最高有效位的表达式需要一个具有2n - 1个输入的或门和2n-1个具有2到n+1输入的与门。因此，通常一次仅执行4至8位的全进位前瞻。图11.21显示了如何使用四个8位加法器构造32位加法器。在这种情况下，进位必须按序通过四个8位加法器，但这将比按序通过32个1比特加法器快得多。

## 11.4时序电路

  组合电路实现了数字计算机的基本功能。然而，除ROM这一特殊情况之外，它们不提供存储器或状态信息，这些元素对于数字计算机的操作来说也是必不可少的。为了达到这些目的，人们采用了一种更复杂的数字逻辑电路形式：**时序电路**。时序电路的电流输出不仅取决于电流输入，还取决于输入过去的历史。另一种通常更有用的观点是，时序电路的电流输出取决于该电路的电流输入和当前状态。

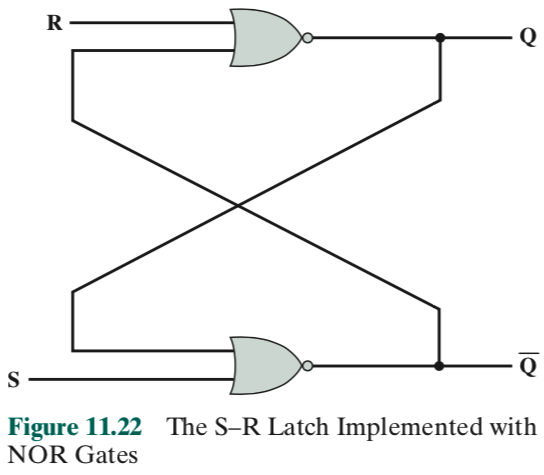
  在本节中，我们将研究一些简单但有用的时序电路实例。我们将会看出，时序电路利用了组合电路。

### Flip-Flops

时序电路最简单的形式是flip-flops触发器。存在多种flip-flops触发器触发器，它们都具有两个特性：

* flip-flops触发器是双稳态器件。它存在于两种状态之一，并且在没有输入的情况下，仍然处于当前状态。因此，flip-flops触发器可以用作1位存储器。
* flip-flops触发器有两个输出，它们总是互补的。他们通常被标记为Q和。

*S-R*锁存器 图11.22所示的是一种S-R flip-flops触发器或**S-R锁存器**常见的结构。该电路有两个输入S(Set)和R(Reset)以及两个输出Q和，并且由两个以反馈方式连接的或非门组成。



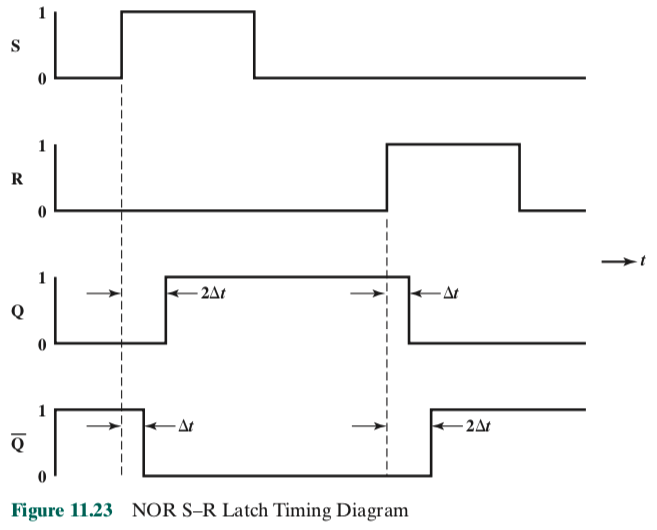
首先，让我们证明电路是双稳态的。假设S和R都是0，Q是0。到低或非门的输入是Q ＝ 0和S＝ 0。因此，输出 = 1表示到高或非门的输入是 = 1和R = 0，其输出Q = 0。因此，电路的状态在内部是一致的，并且只要S = R = 0，它就能保持稳定。类似的推理过程表明状态Q = 1， = 0对于R = S = 0也是稳定的。

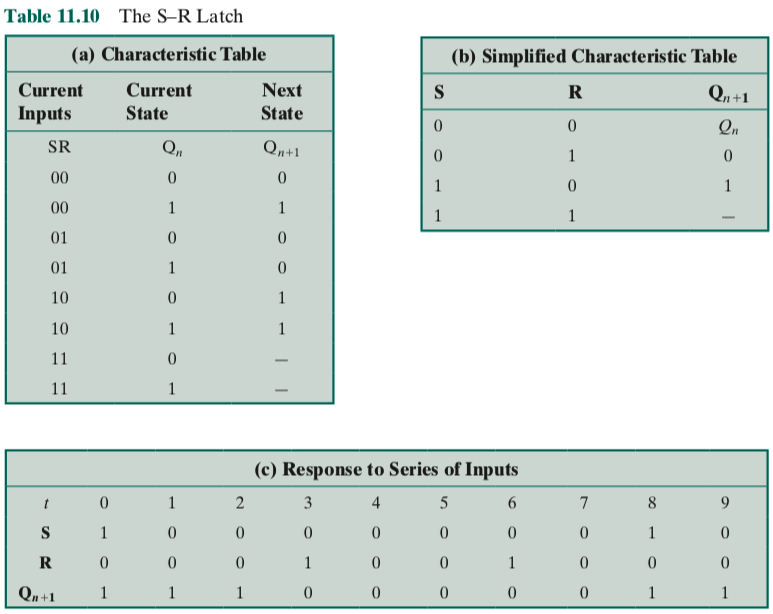
因此，该电路可以用作1比特存储器。我们可以把输出Q看作该比特的“值”。输入S和R用于将值1和0分别写入存储器。为了证明这一点，考虑状态Q ＝ 0， ＝ 1，S ＝ 0，R ＝ 0。假设S的值变为1。现在输入到低或非门是S ＝ 1，Q ＝ 0。经过一段时间延迟Δt之后，低或非门的输出将变为 = 0（参见图11.23）。因此，此时，到上或非门的输入变为R = 0， = 0。在另一个Δt长度的门延迟之后，输出Q变为1。这又是一个稳定的状态。到较低门的输入现在是S = 1，Q = 1，它保持输出 = 0。只要S = 1和R = 0，输出将保持Q = 1， = 0。此外，如果S返回到0，则输出将保持不变。

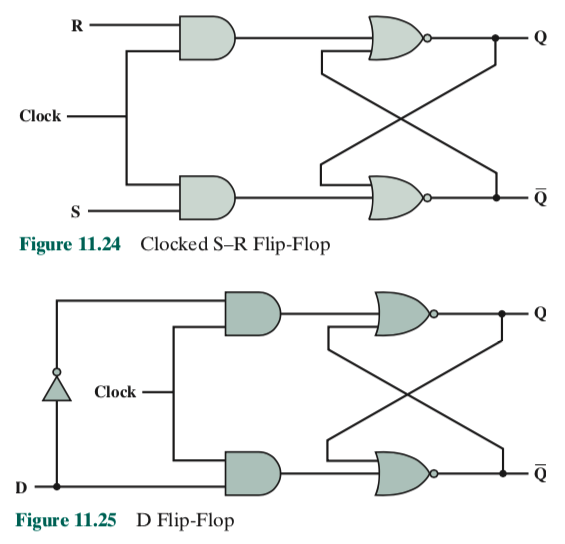
R输出执行相反的功能。当R变为1时，不管Q和 的前一个状态如何，它都强制Q = 0， = 1。同样，在最终状态建立之前会产生2Δt的时间延迟（图11.23）。

S–R锁存器可以用类似于真值表的表来定义，称为特征表，它显示时序电路的下一个或后续多个状态，作为一个当前状态和输入的函数。在S–R锁存器中，状态可以由Q值定义。表11.10a显示了所得到的特征表。注意，输入S = 1，R = 1是不允许的，因为这些将产生不一致的输出（Q和都等于0）。该表可以更简洁地表达为表11.10b所示的样子。表11.10c解释了S–R锁存器的行为。

计时*S-R FLIP-FLOPS*触发器S–R锁存器的输出，在短暂的时间延迟之后，随着输入的变化而变化。这被称为异步操作。更典型的是，数字计算机中的事件与时钟脉冲同步，因此只有



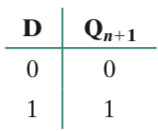




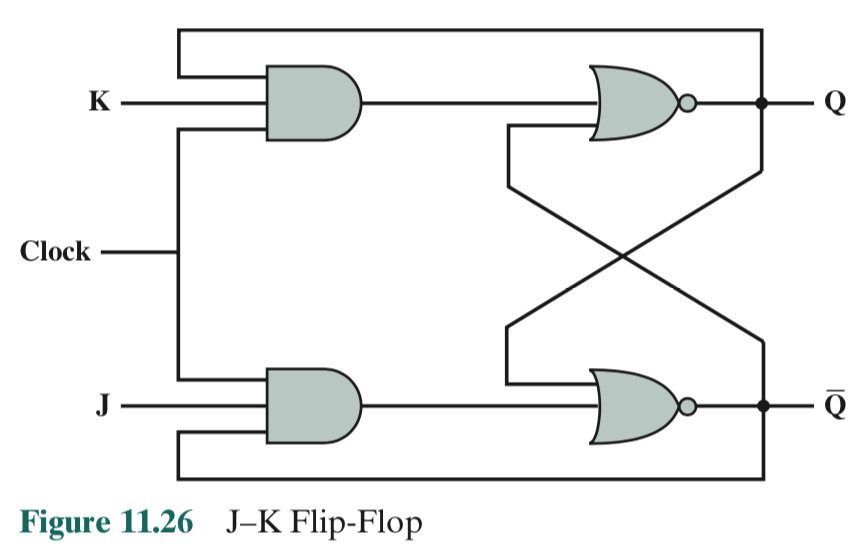
当时钟脉冲到来时才会发生变化。图11.24显示了这一情况。这个装置被称为**时钟S-R flip-flops触发器**。需要注意，R和S输入仅在时钟脉冲到来时才会传递到或非门。

*D flip-flops*触发器 S-R flip-flops触发器的一个问题是必须避免R = 1，S = 1的状态。这样做的一种方法是只允许一个输入。**D flip-flops触发器**实现了这一点。图11.25显示了D flip-flops触发器的门实现。通过使用反相器，保证两个与门的非时钟输入彼此相反。

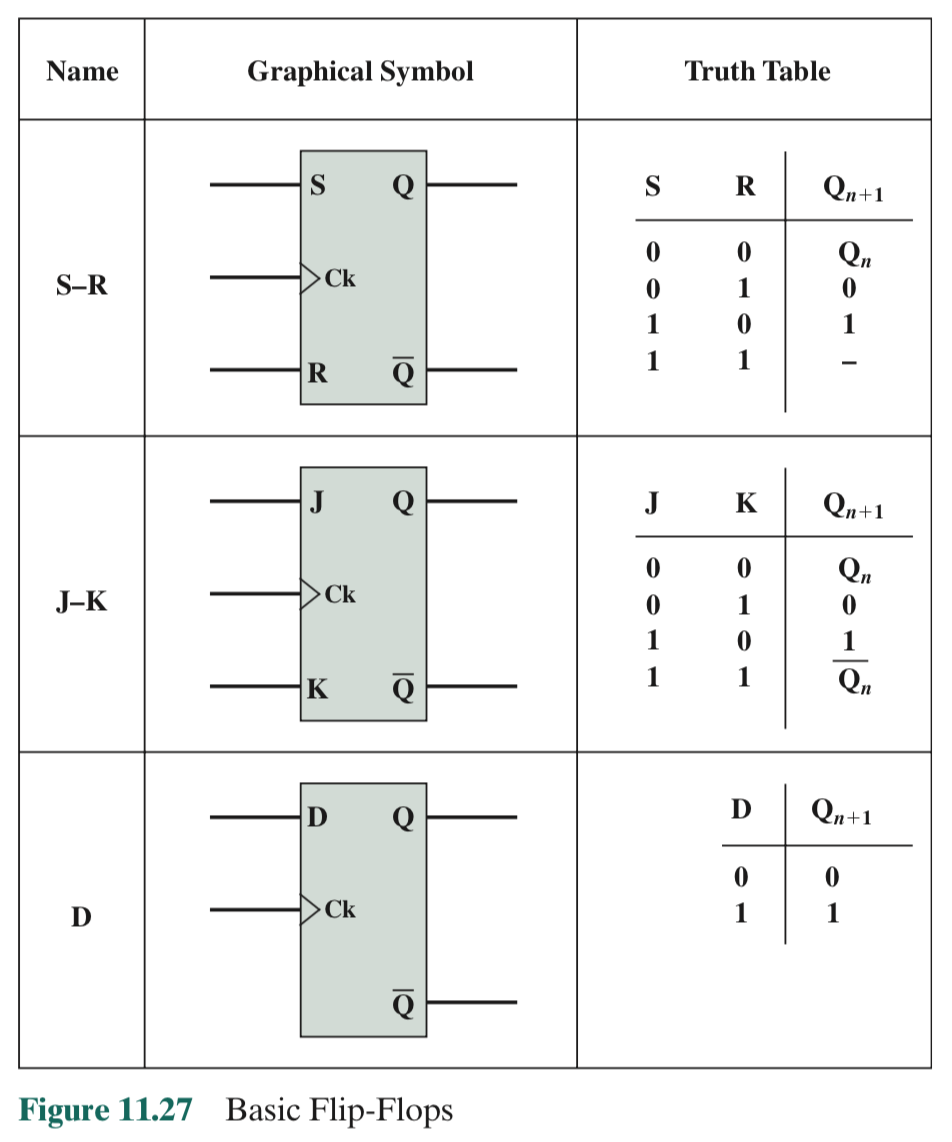
D flip-flops触发器有时被称为数据flip-flops，因为它实际上是一位数据存储。D flip-flops触发器的输出总是等于应用于输入的最新值。因此，它记住并产生最后的输入。它也被称为延迟 flip-flops，因为它延迟一个时钟脉冲输入一个0或1。我们可以从下面的真值表中捕获到D flip-flops触发器的逻辑：

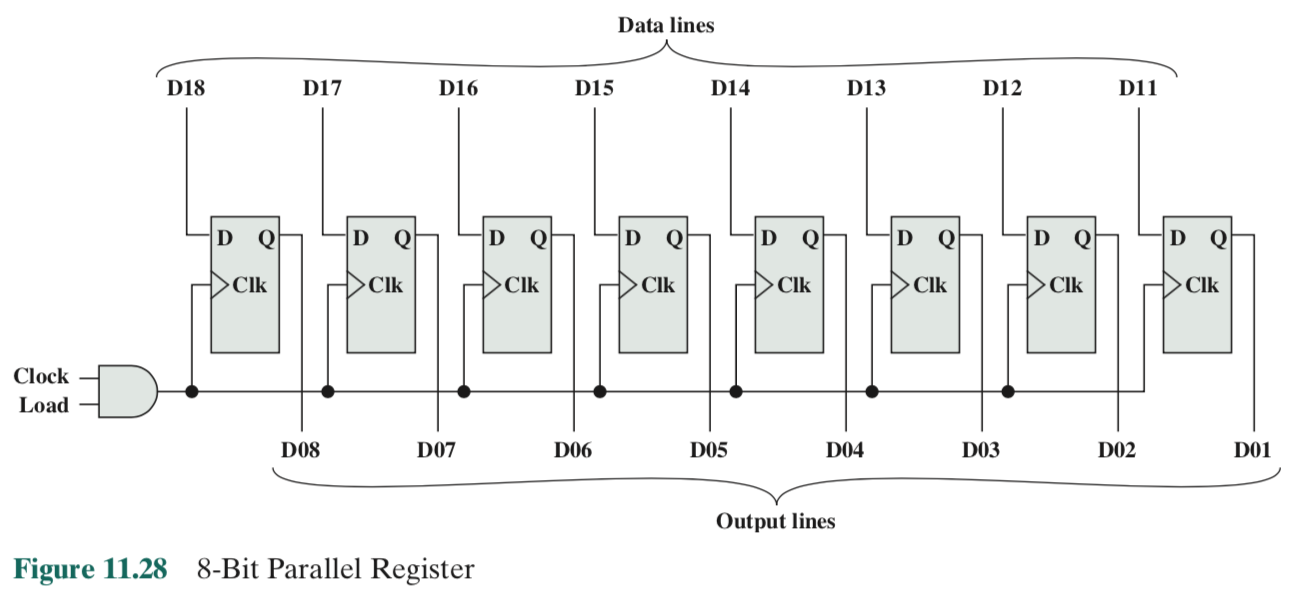


*J-K FLIP-FLOPS* 另一个有用的触发器是J-K触发器。像S-R触发器一样，它有两个输入端。然而，在这种情况下，所有输入值的所有可能组合都是有效的。图11.26显示了J-K触发器的门电路实现，图11.27显示了它的特征表（同时显示了S-R和D触发器的特征表）。注意，前三个组合和S-R flip-flop触发器相同。由于没有输入断言，其输出是稳定的。如果只有输入J被断言，则其结果是一个集合函数，其会导致输出为1；如果只有输入K被断言，则结果为



复位函数，其会使输出为0。当输入J和K同时为1时，所执行的函数称为切换函数(toggle function)：输出是相反的。因此，如果Q是1，并且1被幅值给J和K，那么Q将变为0。读者应该可以验证图11.26的实现将产生这个特征函数。





### 寄存器

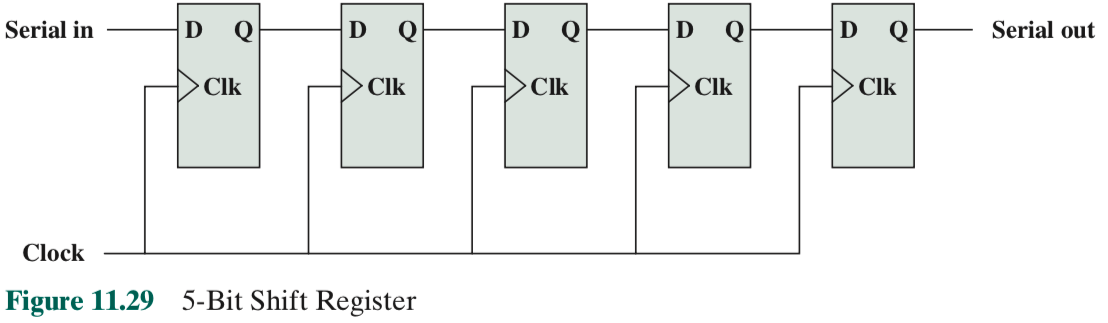
  作为使用flip-flops触发器的一个例子，让我们首先研究一下CPU的基本组成元素之一：寄存器。众所周知，寄存器是CPU内部来存储一个或多个数据位的数字电路。常用的两种寄存器是：并行寄存器和移位寄存器。

**并行寄存器** **并行寄存器**由一组可同时读写的1位存储器组成。它用于存储数据。我们在本书中讨论的寄存器都是并行寄存器。

图11.28中的8位寄存器说明了使用D触发器的并行寄存器的操作。标记为负载的控制信号控制从信号线D11到D18写入寄存器。这些行可能是多路复用器的输出，以便将来自各种源的数据加载到寄存器中。

**移位寄存器** **移位寄存器**串行地接受和/或传输信息。例如，考虑图11.29，它显示了由时钟D触发器构成的5位移位寄存器。数据只被输入到最左边的flip-flop触发器。每到一个时钟脉冲，数据右移一个位置，最右侧的一位会被传送出去。

移位寄存器可以用作接口连接串行I/O设备。此外，它们可以在ALU内用于执行逻辑移位



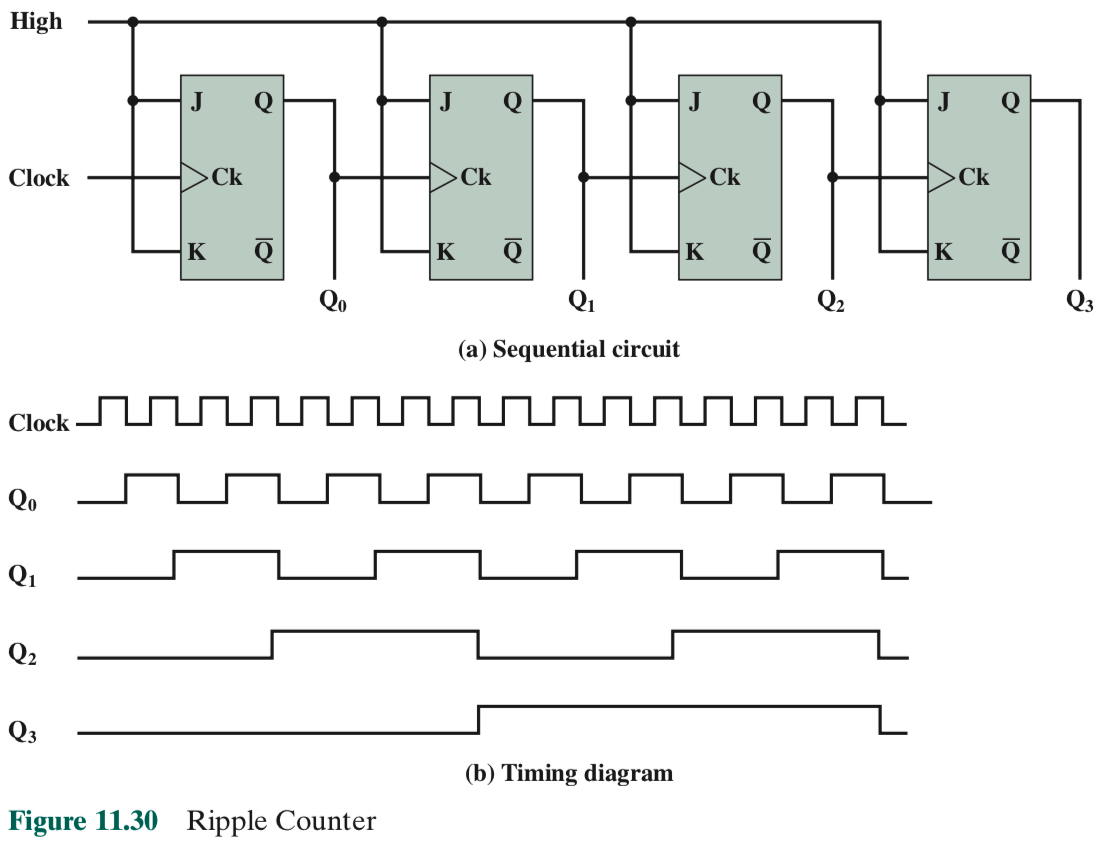
和旋转功能。为了实现ALU内部的这些功能，寄存器需要同时配备串行和并行读/写电路。

### 计数器

时序电路中另一个很有用的分类是**计数器**。计数器是一个寄存器，其值很容易自加1，并将值模上寄存器的容量。也就是说，在达到最大值之后，再自加一的时候，计数器值将被设置为0。因此，由n个flip-flops触发器构成的寄存器可以计数到2n - 1。CPU中计数器的一个实例是程序计数器。

计数器可以被指定为异步的或同步的，这取决于它们操作的方式。异步计数器相对较慢，因为一个flip-flop触发器的输出，会触发下一个flip-flop触发器状态的改变。在**同步计数器**中，所有的flip-flop触发器同时改变状态。因为同步计数器要快得多，它是CPU中使用的类型。但是，先描述清楚异步计数器，对于我们的讨论来说很有用。

纹波计数器 异步计数器也被称为纹波计数器，因为促使计数器自加的变化从一端开始，并像“波纹”一样传播到另一端。图11.30显示了使用J–K触发器的4位计数器的一种实现，同时显示了说明其行为的时序图。时序图是理想化的，因为它没有显示当信号沿着一系列flip-flop触发器向下移动时发生的传播延迟。最左边的flip-flop触发器（Q0）的输出是最低有效位。显然，通过级联更多的flip-flop触发器，该设计可以扩展到任意数量的位。

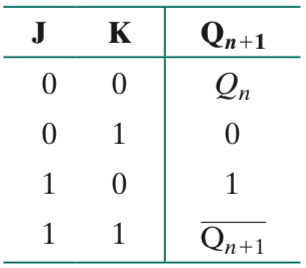


在图示的实现中，计数器随着每个时钟脉冲的到来而递增。到每个触发器的J和K输入保持在一个常数1。这意味着，当时钟脉冲到来时，Q处的输出将被取反（1到0；0到1）。注意，状态的改变发生在时钟脉冲的下降沿；这被称为边缘触发器。在复杂电路中，使用对时钟脉冲的改变作出响应的触发器，比起使用对脉冲本身改变做出相应的触发器，可以提供更好的定时控制。如果查看这个计数器的输出模式，可以看到它在0000,0001、...、1110、1111,0000等状态间循环。

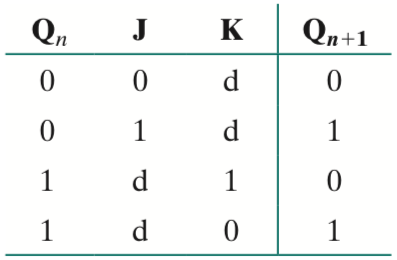
**同步计数器** 纹波计数器的缺点是在改变值时产生的延迟，延迟与计数器的长度成正比。为了克服这个问题，CPU使用同步计数器，其中计数器的所有触发器同时改变。在本小节中，我们提出了一个3位同步计数器的设计。这样做的同时，我们会说明同步电路设计中的一些基本概念。

对于一个3位计数器，需要三个flip-flop触发器。让我们使用J-K flip-flop触发器。将三个flip-flop触发器的未取补的输出依次标记为C, B和A，其中C表示最高有效位。第一步是构建一个与J-K输入和输出相关的真值表，从而帮助我们设计整个电路。这样的真值表如图11.31a所示。前三列显示了输出C、B和A的可能组合。它们按照计数器递增时出现的顺序列出。每一行都列出C、B和A的当前值以及三个触发器的输入，得到下一组C、B和A值需要这三个输入。

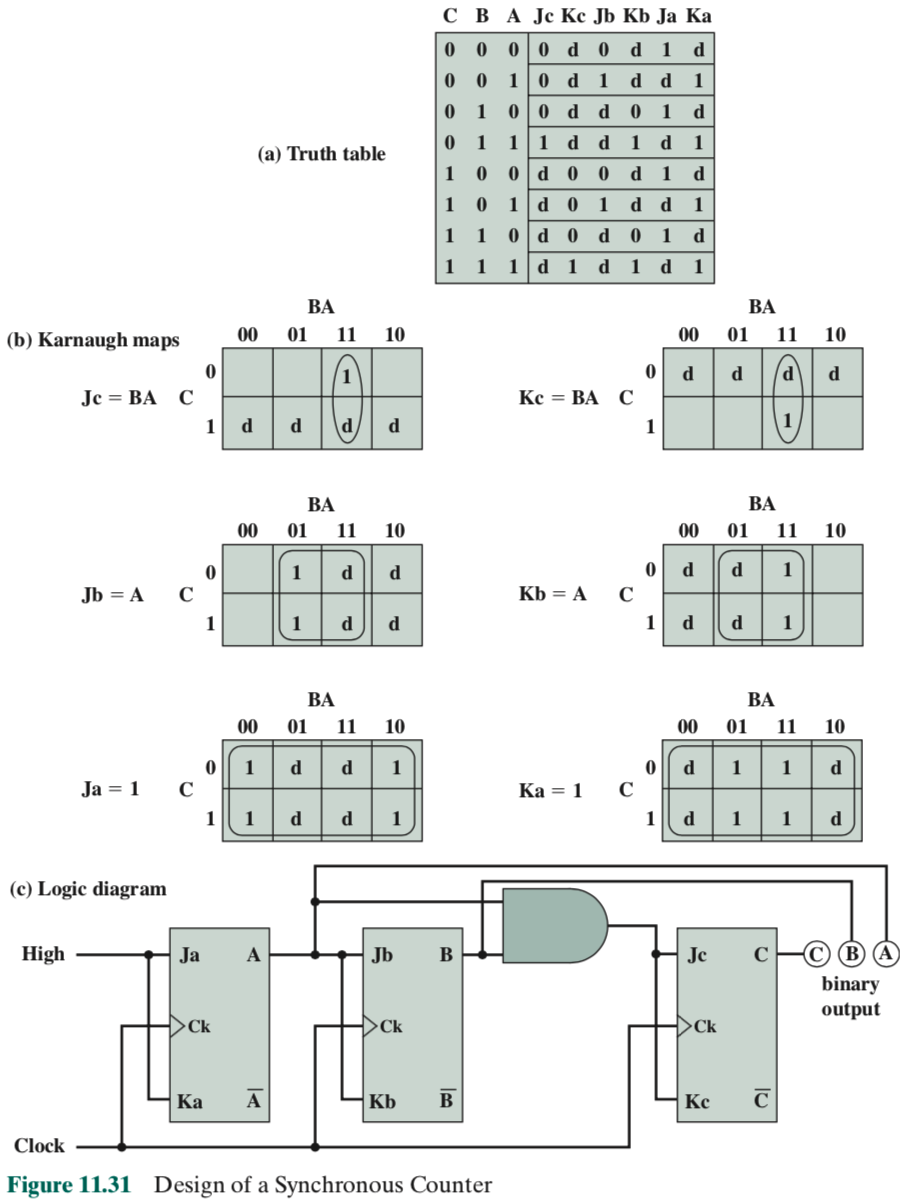
为了理解图11.31a中真值表的构造方法，重新构造J-K触发器的特征表可能会有帮助。回想一下，本表如下所示：



该表显示了J和K输入对输出的影响。现在考虑对相同数据的下面的组织方式：



在这个表中，当输入和当前输出已知时，表会给出下一个输出值。这正是设计计数器或任何时序电路所需的信息。这种表被称为**激励表**。



让我们回到图11.31a。考虑第一行。我们希望C值保持为0，B值保持为0，A值随着时钟脉冲的下一次到来从0变为1。激励表显示，为了保持输出为0，我们必须有J = 0的输入，并且不关心K。为了实现从0到1的变化，输入必须是J = 1和K = d。这些值显示在表的第一行。通过类似的思路，可以填充表的其余部分。

在构造了图11.31a的真值表之后，我们看到该表显示了所有需要的J和K的输入，每个输入都可以作为C、B和A的当前值的函数。借助卡诺图，我们可以给出这六个函数的布尔表达式。这在图中的b部分中显示。例如，变量Ja（触发器从J输入，产生A输出）的卡诺图导出表达式Ja = BC。当推导出所有六个表达式时，设计实际电路就很简单了，如图中c部分所示。

## 11.5可编程逻辑器件

到目前为止，我们已经将各个单独的门作为构建块，从中可以实现任意功能。设计者可以通过操纵相应的布尔表达式来达到使用最少数量门的策略。

  随着集成电路提供的集成级别的增加，增加了它的考虑。早期的集成电路，使用小规模集成（SSI），在一个芯片上部署一到十个门。在至今所描述的构造块方法中，每个门是独立处理的。为了构造逻辑功能，在印刷电路板上布置了许多这样的芯片，并且进行适当的引脚互连。

  集成度的提高使得在芯片上部署更多的门、同时在芯片上进行门互连成为可能。这样降低了成本、减小了尺寸并提高了速度（因为片上延迟比片外延迟更短）。然而，出现了一个设计问题。为了实现每个特定的逻辑功能或功能集，都必须设计芯片上的门和互连的布局。设计这种定制芯片所需要的的成本很高同时需要花费很多时间。因此，开发出可以适应特定功能的通用芯片变得很有吸引力。这是可编程逻辑器件（PLD）产生的动机。

在市场上有许多不同类型的PLD。表11.11列出了一些专业术语并定义了一些最重要的类型。在本节中，我们首先介绍这类设备中最简单的一种，可编程逻辑阵列*（PLA）*，然后介绍可能是最重要同时应用最广泛的一类PLD，现场可编程门阵列*（FPGA）*。

### 可编程逻辑阵列

PLA基于这样的一个事实，任何布尔函数（真值表）都可以用乘积和（sum-of-products, SOP）的形式表示，正如我们所看到的。PLA由芯片上的非、与以及或门的常规布置组成。每个芯片输入会通过非门，一遍每个输入及其反可用于每个与门。每个与门的输出对于每个或门可用，并且每个或门的输出是芯片输出。通过建立适当的连接，可以实现任意的SOP表达式。

  图11.32a显示了一个具有三个输入、八个门和两个输出的PLA。左边是一个可编程的AND阵列。AND阵列是通过建立一个在PLA输入或其否定和AND门输入之间的连接来编程的，通过连接它们交点的对应线来实现。

 表 11.11 PLD 术语

**可编程逻辑设备（PLD）**

通用术语，指用于实现数字硬件的任何类型的集成电路，其中芯片可由终端用户配置来实现不同的设计。对这种设备的编程通常需要将芯片放入专门的编程单元，但有些芯片也可以“在系统中”配置。同时也被称为现场可编程设备（FPD）

**可编程逻辑阵列（PLA）**

一种相对较小的PLD，包含两个逻辑层，一个AND平面和一个OR平面，其中两个层都是可编程的。

**可编程阵列逻辑（PAL）**

一种相对较小的PLD，具有可编程的AND平面，后面跟着一个写定的OR平面。

**简单PLD（SPLD）**

一种PLA或PAL.

**复杂PLD（CPLD）**

一种更复杂的PLD，由单个芯片上的多个SPLD样块组成。

**现场可编程门阵列（FPGA）**

PLD具有通用结构，允许非常高的逻辑容量。而CPLD具有具有大量输入（AND平面）的逻辑资源，但是FPGA提供更窄的逻辑资源。FPGAs还提供了比CPLD更高的触发器到逻辑资源的比率。

**逻辑块**

在FPD中的阵列中复制的相对较小的电路块。当一个电路在FPD中实现时，它首先被分解成较小的子电路，每个子电路可以被映射到一个逻辑块中。术语逻辑块主要用于FPGA的相关描述中，但它也可以指CPLD中的电路块。

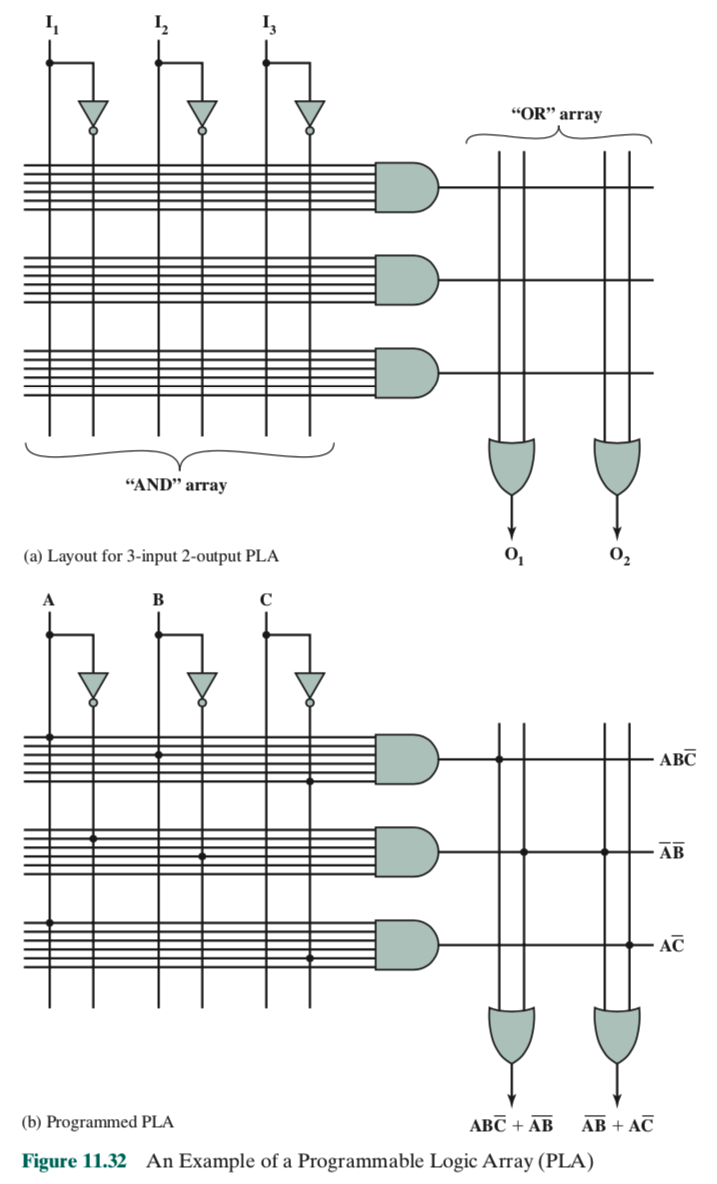
右边是可编程的OR阵列，它将与门输出连接到或门输入。大多数较大的PLA包含几百个门、15到25个输入以及5到15个输出。从输入到与门以及从与门到或门的连接，直到编程的时候才被指定。

PLA以两种不同的方式制造，以便于编程（制造连接）。首先，每个可能的连接都是通过每个交点处的引线进行的。然后，可以通过去掉引线的方式来去除不需要的连接。这种类型的PLA被称为现场可编程逻辑阵列*（FPLA）*。或者，在芯片制造期间，通过使用为特定互连模式设计的适当的掩模，可以建立适当的连接。无论哪种情况，PLA都提供一种灵活、廉价的实现数字逻辑功能的方法。

  图11.32b展示了一个实现两个布尔表达式的已编程PLA。

### 现场可编程门阵列

  PLA是一个简单的PLD（SPLD）的例子。提高严格SPLD架构的容量的难点在于，随着输入数量的增加，可编程逻辑平面的结构在大小上增长得太快。提供基于SPLD架构的大容量器件的唯一可行方法是，将多个SPLD集成到单个芯片上，并提供互连，以便可编程地将SPLD块连



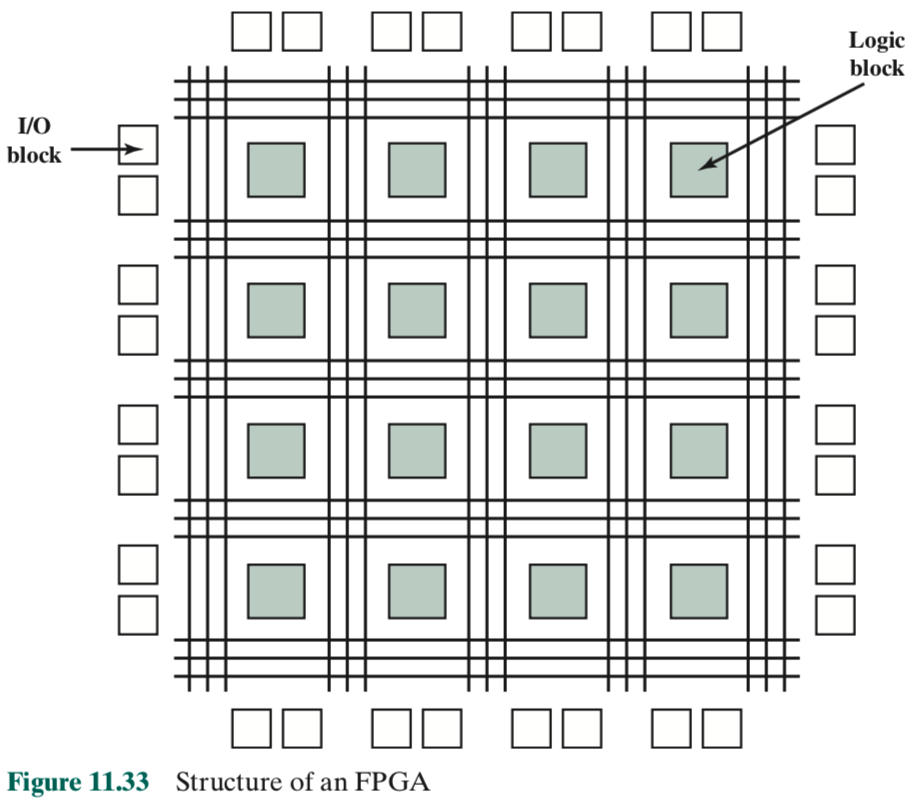
接在一起。现今市场上的许多商用PLD产品使用的是这种基本的结构，被统称为复杂PLD（CPLD）。最重要的CPLD类型是FPGA。

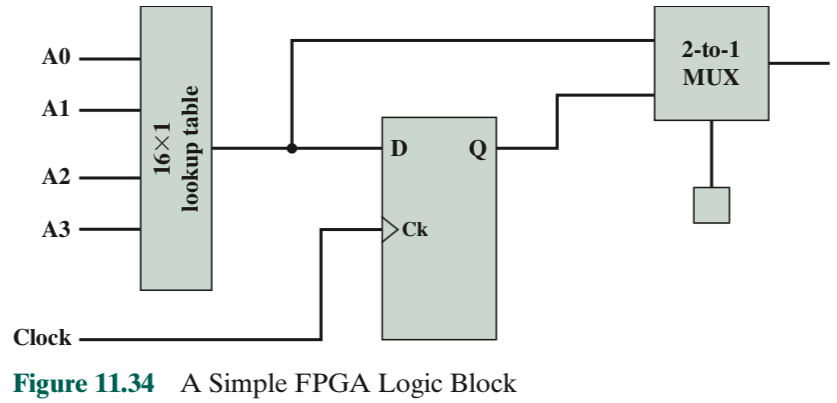
FPGA由未定用途的电路元件阵列（称为**逻辑块**）和互连资源组成。图11.33显示了典型的FPGA体系结构。FPGA的主要部件是：

* **逻辑块**：可配置逻辑块是计算用户电路的地方。
* **I/O模块**：I/O模块将I/O引脚连接到芯片的电路上。
* **互连**：这些是可用于在I/O模块和逻辑块之间建立连接的信号路径。

逻辑块可以是组合电路或时序电路。实质上，逻辑块的编程是通过下载逻辑函数的真值表的内容来完成的。图11.34显示了一个简单的逻辑块的示例，该逻辑块由一个D触发器、一个2到1的多路复用器和一个16位**查找表（lookup table）**组成。查找表是由16个1比特元素组成的存储器，因此需要4个输入线来选择16位中的一个。更大的逻辑块具有更大的查找表和多个互连查找表。由查找表实现的组合逻辑可以直接输出，也可以存储在D触发器中并同步输出。一个单独的一位存储器控制多路复用器，用以确定输出是直接来自查找表还是来自触发器。

通过互连多个逻辑块，可以很容易地实现非常复杂的逻辑功能。





## 11.6 关键术语和习题

**关键术语**

|  |  |  |
| --- | --- | --- |
| adder 加法器 | OR gate 与门 | field-programmable gate array  (FPGA) |
| AND gate 与门 | parallel register 并行寄存器 | flip-flop 触发器 |
| assert 断言 | combination circuit 组合电路 | ripple counter 纹波计数器 |
| Boolean algebra 布尔代数 | complex PLD(CPLD) 复杂PLD | sequential circuit 时序电路 |
| clocked S-R flip-flop 时钟S-R触发器 | counter 计数器 | shift register 移位寄存器 |
| D flip-flop D触发器 | decoder 译码器 | simple PLD (SPLD) 简单PLD |
| gates 门 | product of sums(POS) 和的乘积 | sum of products (SOP) 乘积的和 |
| graphical symbol 图像符号 | programmable array logic  (PAL) 可编程阵列逻辑 | synchronous counter 同步计数器 |
| J-K flip-flop J-K触发器 | programmable logic array  (PLA) 可编程逻辑阵列 | S-R Latch S-R锁存器 |
| Karnaugh map 卡诺图 | programmable logic device  (PLD) 可编程逻辑设备 | truth table 真值表 |
| logic block 逻辑块 | Quine–McCluskey method | XOR gate 或非门 |
| lookup table 查找表 | read-only memory (ROM) |  |
| NAND gate 与非门 | register 寄存器 |  |
| NOR 或非 | excitation table 激励表 |  |

**习题**

11.1 为一下布尔表达式构建真值表：

1. (A + + C)( + B + )
2. (A + B + )(A + B + C)( + B + )
3. + ((B + C)( + ))
4. (A B) + ( ) + (A )

11.2根据交换法简化下列表达方式：

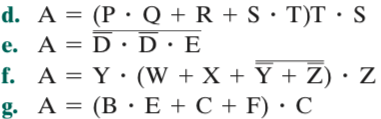
1. ( + D)(D + )(B + E + )( + E + B)(A + B + C)
2. (X + Y)(Y + Z)(Y + X)
3. (A + B + C)(D + E + F)(X + Y)(C + A + B)
4. (ABC) + () + () + A(B + C) + (CBA) + ()

11.3将德摩根定律应用到下列等式中：

1. A =
2. B =

11.4简化下列表达式：

1. A = S·T + V·W + R·S·T
2. A = T·U·V + X·Y + Y
3. A = F·(E + F + G)



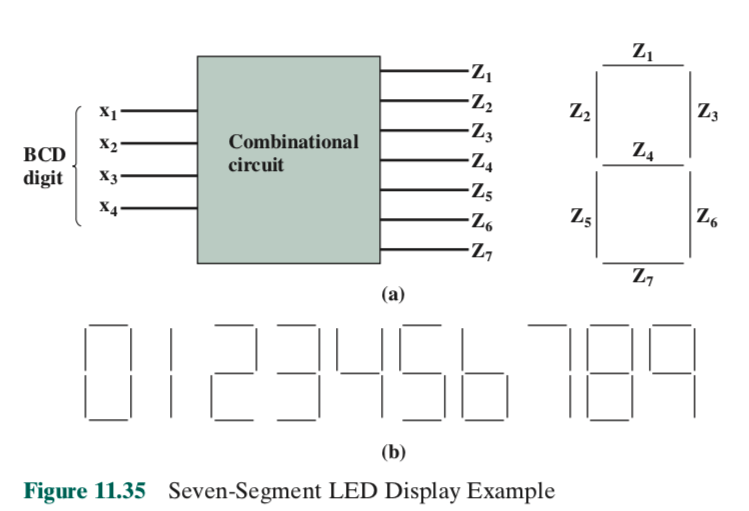
11.5 使用基本布尔操作或和与构造或非操作。

11.6 使用或非门和非门，画出可以实现三输入与功能的逻辑图。

11.7 写出三输入或非门的布尔表达式。

11.8 图11.35所示的是一个用来控制十进制数的七段显示的组合电路。该电路具有四个输入，提供用于表示十进制数的四位代码（010 = 0000,...,910 = 1001）。七个输出决定了哪些段将被激活点亮来显示给定的十进制数字。注意，有一些输入和输出的组合是用不到的。

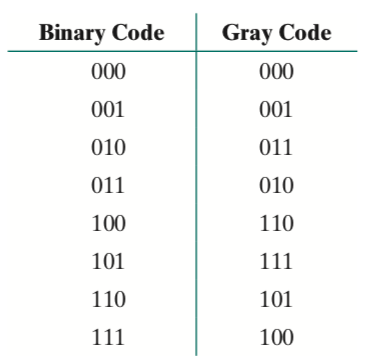
1. 写出这个电路的一个真值表。
2. 写出真值表的SOP形式的表达。
3. 写出真值表的POS形式的表达。
4. 写出简化的表达式。



11.9 设计一个4到1多路复用器。

11.10 在图11.15中添加一行，使其起到解复用器的作用。

11.11 格雷码是一种整数的二进制编码。它不同于普通的二进制表示，在格雷码中，任何两个相邻数字的代码表示中只有一位是不同的。这对于实现计数器或产生数字序列的模数转换器等应用非常有用。因为一次只改变一个比特，所以不会由于微小的计时差异而造成任何模糊。代码的前八项是：



设计一个从二进制码转换为格雷码的电路。

11.12 使用四个3×8译码器（带有使能输入）和一个2×4译码器设计一个5×32译码器。

11.13 只使用五个门，实现图11.20所示的完整加法器。（提示：有些门是**异或门**。）

11.14 考虑图11.20。假设每个门会产生15 ns的延迟。因此，输出的和在45ns之后有效，进位输出在0ns之后有效。对于以下两种16位加法器，其总加法时间分别是多少？

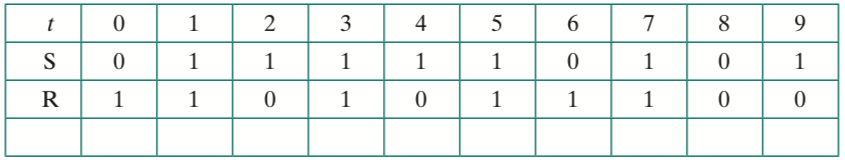
a. 不使用进位前瞻技术，如图11.19所示。

b. 使用进位前瞻技术，同时使用8位加法器，如图11.21所示。

11.15 另一种S-R锁存器的实现形式具有与图11.22相同的结构，但是使用与非门而不是或非门。

a. 对于使用与非门实现的S-R锁存器，重做表11.10a和11.10b。

b. 完成下表，类似于表11.10c。



11.16考虑图11.27中S-R触发器的图形符号。添加额外的线来表示从S–R触发器接线的D触发器。

11.17表达具有三个输入(C、B、A)和四个输出(O0、O1、O2、O3)的PLA的结构，其输出定义如下：

O0 = C+A+AB

O1 = C+AB

O2 ＝ C

O3 = A+AB

11.18 PLA的一个有趣的应用是从旧的、过时的穿孔卡片字符码转换为ASCII码。过去在计算机上非常流行的标准穿孔卡片字符码有12行和80列，可以在其中打孔。每列对应一个字符，因此每个字符都有一个12位的代码。然而，实际上只使用了96个字符。考虑一个读取穿孔卡并将字符代码转换为ASCII码的应用。

1. 描述此应用的PLA实现。
2. 这个问题能用ROM解决吗？给出解释。