HZ BOOKS

电子与电气工程丛书

McGraw Hill Education

数字逻辑基础与Verilog设计

（原书第2版）

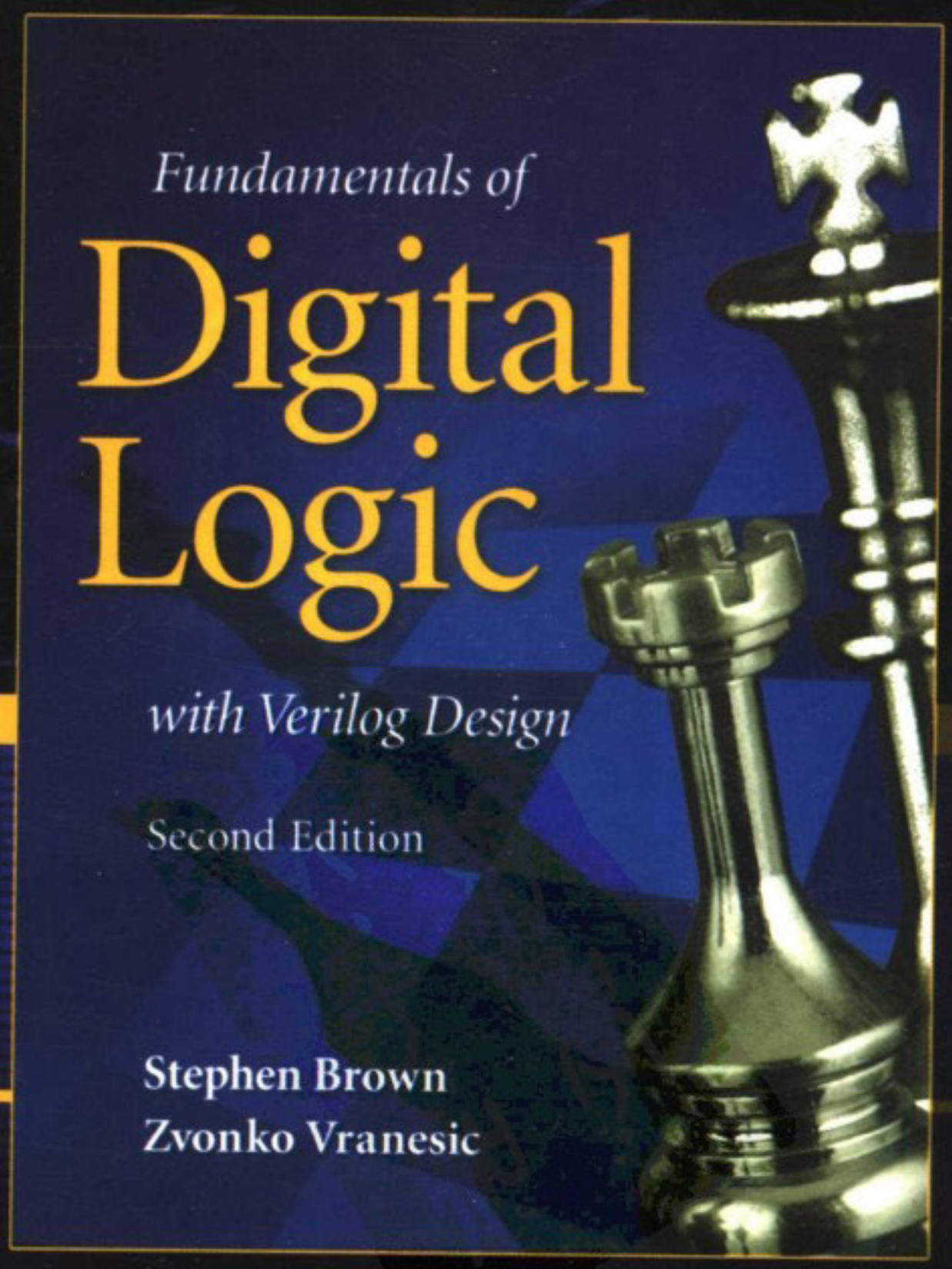
Fundamentals of Digital Logic with Verilog Design

(Second Edition)

Stephen Brown
（加） Zvonko Vranesic 著
多 伦 多 大 学

夏宇闻 等译　须毓孝 审校

附赠光盘



机械工业出版社
China Machine Press

电子与电气工程丛书

数字逻辑基础与Verilog设计

（原书第2版）

Fundamentals of Digital Logic with Verilog Design (Second Edition)

Stephen Brown
（加） Zvonko Vranesic 著
多 伦 多 大 学

夏宇闻 等译
须毓孝 审校

机械工业出版社
China Machine Press

译 者 序

为了等待原书第2版的出版，以便给读者提供最新版的中文翻译书，在译完第1版后又几乎等了1年，才得到完整的原书第2版。又经过几个月的持续努力，终于把机械工业出版社的翻译任务完成了。回想当初接受出版社邀请的时候，我是十分犹豫的。高质量地完成这么多页教材的翻译所需要花费的时间和精力，对于动过脊柱固定手术、开始进入老年的教师，无疑是一个沉重的负担。何况本人还有120多学时本科生和研究生的教学任务，以及正在进行的科研项目的压力。

2003年北航IC设计硕士班第一次开班时，计算机学院教务处要求我负责该班60多名学生的数字电路基础和设计课程的教学工作，并规定必须用英语授课。为此，我搜寻了五六本国外2000年以后出版的数字电路基础和设计方面的经典教材，仔细比较后，发现McGraw-Hill出版社当年出版的这本书（第1版）最适合这些学生的水准。教材的内容全面细致、概念清楚，与最新技术发展结合得很好，而且与Verilog语言的教学和复杂系统的设计也能很好地衔接，因此毅然决定在教学中选用本书作为教材。

几年的教学实践使我深深体会到当初选择的正确。教材的内容我已非常熟悉，关键部分还编写了详细的教案，这些虽然为翻译本书打下了很好的基础，但离全书完整的翻译还有很大的距离。翻译不但必须忠实原文，而且还要符合中国人的阅读习惯，更何况书中有许多内容在国内的教材中从来未曾涉及过。学生们虽然帮助做了一些翻译工作，但修改整理比完全由我自己翻译所花费的时间还要多。为了准确、通顺地表达书中的内容，我花费了大量的精力。在9个多月的时间里，我抓紧平时点点滴滴的时间，并利用周末、寒假和暑假持续地努力来完成翻译任务。

支持我翻译本书的动力是，我认为这是到目前为止我所看到过的最好的一本教科书。特别要提出的是，本书的第2版已经把辅导教材改成最近流行的Altera Quartus II 6.0工具，作者对职业市场的人才需求做出了非常及时的反应。本书的中文版在作者和Altera公司的帮助下，得以和英文版基本同步出版发行，这对于中国的广大学生是莫大的幸运，有助于他们适应职业市场对人才的需求。本书的内容非常准确、细致和全面，而且对复杂数字电路设计中的关键问题讲解得非常透彻，对于想进入数字设计领域的优秀学生，不需要老师指导，完全可以通过自学掌握本书的全部内容。可能部分读者会认为本书的内容过于细致和烦琐，但我认为这些内容对于想真正掌握关键设计技术和原理的工程师都是必需的，而国内目前出版的教材很少能达到这样准确、细致、深入和全面的程度，原书作者作为教师的负责态度令人感动。

参加本书翻译的人员还包括李鹏、邢小地、宋成伟、霍强、邢志成、李琪、路倩、薛静、沈立、石伟、孙丰达、尹庆立和赵彤，他们每人翻译的初稿从5000～20 000个中文字不等，

余下的部分由我自己翻译完成。译稿的修改、整理和最后成文全部由我负责完成。翻译定稿后，我邀请了内蒙古大学计算机学院的须毓孝教授承担本书的最终审阅工作。须教授是我大学时期最好的朋友之一，我们一起在清华大学度过了难忘的青年时代。他极其认真负责地审阅，发现并改正了多处由于我的疏忽而造成的错误。须教授在审阅工作中付出了巨大的努力，最终使本书的翻译工作得以高质量地完成。第2版的翻译修改得到了神州龙芯公司陈岩工程师的帮助，他耐心地帮助我核对了第2版的全部补充翻译，在此表示衷心的感谢。

在本书译稿完成交付印刷之际，谨向在本书翻译过程中所有帮助过我的同学和朋友，以及不断地鼓励我坚持工作的出版社的同志表示由衷的感谢。

夏宇闻

译者简介

夏宇闻 上海市人。20世纪60年代毕业于清华大学自动控制系，改革开放后首届硕士学位获得者。目前为北京航空航天大学教授，主要从事复杂数字逻辑的设计研究和教学。曾获得包括国家发明二等奖在内的多项国家级奖励，发表论文多篇。编写和翻译出版的著作包括：《Verilog数字设计教程》、《Verilog HDL数字设计与综合》和《System Verilog验证方法学》等。

须毓孝 江苏省苏州市人。20世纪60年代毕业于清华大学自动控制系，改革开放后首届硕士学位获得者。目前为内蒙古大学计算机学院教授，主要从事计算机软硬件的开发、应用研究和教学。曾获得省部级科技进步奖多项，发表论文多篇。编写出版的著作包括：《工业控制及PLC的原理和应用》和《软件体系结构及实例分析》等。

序　言

下国际象棋是一种游戏，它提供了无与伦比的载体，可在对弈环境下考验人类的智慧。在过去的30年中，下国际象棋起了检测平台的作用，可以用来确定机器仿真人的智力行为的程度。目前的计算机可以运行许多种下棋程序。下棋专用机就是一种由计算机和下棋程序组成的计算机，目前已有能力击败最强大的人类棋手。

1997年，国际商用机器（IBM）公司的深蓝（Deep Blue）下棋专用机，在一场六轮制的比赛中击败了世界冠军盖瑞·卡斯帕罗夫（Garry Kasparov），这是一场终极的挑战。从本质上来讲，这台机器就是逻辑电路、算法和软件的组合，当然还包括知道如何使用这些资源的人。虽然所有这些因素都非常关键，但从下棋能力的角度看，深蓝下棋专用机最大的优势在于开发了功能极其强大的逻辑电路，其中大部分都是通用计算机常用的逻辑电路，但还有一些是为下棋而专门设计的逻辑电路。深蓝下棋专用机究竟为什么会有如此强大的功能呢？关键在于它能在1秒钟内进行2亿次棋子位置的计算。

本书讲解逻辑电路的原理及设计方法。另外，书中还纳入了1997年“人机大战”第六次比赛中的决定性棋步，让读者了解精心设计的逻辑电路是如何达到难以置信的智能程度的。当时深蓝下棋专用机执白子先走。

前　言

本书是为“数字逻辑设计”课程编写的入门教材，这门课是大多数电子和计算机工程专业的基础课程。成功的逻辑电路设计人员必须深入理解这门课的基本概念，并熟练掌握计算机辅助设计工具的使用。本书的宗旨是希望能在基本概念的教学和计算机辅助设计工具的实际应用之间提供一种恰到好处的平衡。为了帮助读者学习，书中还包括了必要的计算机辅助设计（CAD）工具。

许多有关数字逻辑设计的书覆盖面太广，这是很严重的问题。覆盖面太广的书不适合作为教材，特别是当课本涉及的主题深度不够时，情况更糟。另外，有的书想把大量的实际设计经验传授给学生，却使得还不清楚基本概念的学生很难理解书中的内容。我们的目的是避免出现这两方面的问题。

本书的主要目的是：1）通过教学使学生掌握经典的手工设计数字电路的基本概念；2）清楚地说明当今使用计算机辅助设计工具设计数字电路的方法。虽然现代设计人员（除了少数情况外）已经不再使用手工设计技术，但是手工设计可以让学生直观地了解数字电路是如何工作的。手工技术还可以为计算机辅助设计工具所进行操作的类型提供说明，让学生了解自动化设计技术的优点。本书中穿插了许多例子来引入基本概念，这些例子涉及一些简单的电路设计，我们不但用手工的方法，而且也用现代的基于计算机辅助设计的方法来完成这些电路的设计。在基本概念建立起来之后，用计算机辅助设计工具提供了一些比较复杂的例子。因此，本书的重点放在如何利用现代设计方法来进行数字电路和系统的设计上。

技术和计算机辅助设计支持

本书讲述现代数字电路实现技术，简要地讨论了小规模集成电路（SSI）以及半定制和全定制技术，但重点放在可编程逻辑器件（PLD）上。可编程逻辑器件是最适用于教学的技术，原因如下：首先，在实践中，可编程逻辑器件得到了广泛的应用，几乎所有类型的数字电路设计都可以由PLD来完成。其次，可编程逻辑器件实现的电路是由最终用户通过编程实现的。因此学生有机会在实验室的条件下，把课本上的例子配置到真实的芯片中。学生还可以在自己的计算机上对所设计的电路进行仿真，观察电路行为。本书中，使用两种最常用的可编程逻辑器件作为设计的实现器件：复杂可编程逻辑器件（CPLD）和现场可编程门阵列（FPGA）。

本书中使用的计算机辅助设计工具是Altera公司的Quartus II软件。Quartus II能自动地把设计映射到Altera公司的复杂可编程逻辑器件和现场可编程门阵列中，这两种器件是工业界最常用的可编程逻辑器件。Quartus II软件的特色如下：

- Quartus II是一个商业产品。本书采用的版本支持产品的所有主要特性。学生可以方便地把设计输入到计算机辅助设计工具中，然后把设计编译到所选择的器件中（可以在任何时间改换其他器件，也可以把设计重新映射到其他器件中），进行功能仿真，也可以对最终电路进行详细的时序仿真，如果学校的实验室配备了相关的设备，还可以在真实

的芯片中实现设计。

- Quartus II提供了两种设计输入工具：硬件描述语言（HDL）和电路原理图。在本书中，我们提供了用电路原理图输入设计的例子，但重点放在基于硬件描述语言的设计上，因为这种方法是实际应用中效率最高的。本书详细地描述了IEEE标准的Verilog语言，并在例子中广泛使用了这种语言。本书包括的计算机辅助设计软件中的Verilog编译器可使学生运用Verilog代码来创建电路，并在真实的芯片中实现这些电路。
- Quartus II可以自动地将设计映射到不同类型的器件中。这一特性使我们得知目标器件的体系结构在哪些方面影响了设计者所设计的电路。
- Quartus II可以在大多数普通类型的计算机上运行。建议学生在IBM兼容的安装有任何版本微软Windows操作系统的计算机上运行这个软件。但是，根据Altera公司的大学计划，Quartus II也可以在其他类型的计算机（例如SUN 或者 HP工作站）上运行。

本书附赠光盘中包括Quartus II软件，附录[⊖]中介绍了该软件的使用方法，以便学生操作、运行和测试书中的各个设计范例。

内容概览

第1章对数字系统的设计过程进行总体介绍，讨论设计过程中的关键步骤，并描述计算机辅助设计工具如何把许多必须做的任务自动化。

第2章介绍逻辑电路的基本概念，说明如何用布尔代数来表示逻辑电路。通过硬件描述语言详细描述逻辑电路的范例，使读者对Verilog有一个初步印象。

第3章介绍数字电路在电子学方面的概念，说明基本逻辑门是如何用晶体管构成的，展示影响电路性能的多种因素。该章的重点是讲述最新的技术，并特别关注CMOS技术和可编程逻辑器件。

第4章讲述组合逻辑的综合。该章覆盖了综合过程的各个方面，从开始设计到产生最后的电路所必需的优化步骤。该章也说明如何将计算机辅助设计工具用于这一目的。

第5章的重点放在算术运算电路上，首先讨论数字系统中数的表示，接着说明如何用逻辑电路巧妙地处理这样的数。该章说明如何用Verilog 语言详细描述想得到的电路功能，以及计算机辅助设计工具是如何提供开发所需要的电路的机制的。本书把数的表示安排在本章，而没有安排在前面，这样做是为了使讨论更有意义且使学生更感兴趣，因为此时可以举例说明如何用实际电路来处理数字信息。

第6章介绍用做构造电路模块的组合逻辑电路。这类电路包括编码器、译码器和多路选择器。这些电路用来说明许多Verilog语句结构是非常方便的，为读者提供发掘更多Verilog高级特性的机会。

第7章介绍存储元件，讨论使用触发器来实现规则的结构，如移位寄存器和计数器等。另外，还包括如何编写Verilog代码来生成此类结构的电路。该章还展示了如何设计较大型系统，如简单的处理器。

第8章详细地讨论同步时序电路（有限状态机）。该章解释了这类电路的行为，并介绍如何手工/自动地设计和开发实际电路。

第9章讨论异步时序电路，但不是十分全面，而是向读者介绍了这类电路的主要特点。即

⊖ 附录内容放在光盘中。——编辑注

使目前在实际工作中已很少采用异步电路，也还是有必要研究异步电路，因为异步电路有助于读者深入理解一般数字电路的功能。异步时序电路清楚地说明了传播延迟和冒险竞争造成的后果，而这种延迟和竞争是电路结构所固有的。

第10章讨论设计实际系统时碰到的几个具体问题，突出介绍在实际设计工作中经常遇到的问题，并说明如何解决这类问题。较大型电路的范例说明了在数字系统设计中必须采用层次化的解决办法，该章还介绍了这类电路完整的Verilog代码。

第11章介绍测试。逻辑电路设计人员必须认识到电路测试的重要性，至少应该熟悉有关测试的最基本问题。

第12章介绍用计算机辅助设计工具设计、实现和测试数字电路。

附录A完整地总结了Verilog语言。虽然Verilog的使用贯穿本书，但附录A还是为读者提供了便利，编写Verilog代码时可以随时参考本附录。

附录B、C、D是一系列的自学教程，讲解Quartus II计算机辅助设计工具，介绍如何一步一步地使用这个软件。

附录E给出范例中使用的器件的详细说明。

教学建议

本书中的所有内容可以在两个学期中讲授，其中最重要的部分可以在一个学期内讲授。当然，讲解过程中不能过于详细地讲解计算机辅助设计工具和Verilog中许多错综复杂的特征。为了实现这一教学进度，本书采取了便于自学的以模块为单元的形式来组织Verilog的内容。根据我们在多伦多大学授课的经验，只需花2～3个学时讲解Verilog就可以了，主要讲解时序电路的技术指标。本书中的大部分Verilog范例都有很好的说明，学生很容易理解。此外，不必讲解如何使用计算机辅助设计工具，因为学生完全可以自学附录B、C和D中的Quartus II辅导教材。

本书还适用于一般的逻辑设计课程（不讲解Verilog语言）。然而，有关Verilog的知识，即使是一些入门知识，对学生也是非常有益的，可以为学生成为数字电路设计师打下坚实的基础。

一学期（15~18周）课程安排

正式讲课的自然起点是第2章。第1章是一般性的介绍，目的是告诉学生为什么逻辑电路很重要，以提高他们的学习兴趣，这部分材料对学生而言很容易理解。

建议课程包括：

- 第2章。
- 第3章：3.1～3.7节。如果学生具有一些电路基本知识，添加3.8节和 3.9节会有帮助。
- 第4章：4.1～ 4.7节，还有4.12节。
- 第5章：5.1～5.5节。
- 第6章。
- 第7章。
- 第8章：8.1～8.9节。

如果时间允许，可包括第9章的9.1～9.3节以及9.6节，还可加进第10章一两个例子。

短学期（13周）课程安排

在短学期课程中，建议课程包括：

- 第2章。
- 第3章：3.1～3.3节。
- 第4章：4.1～ 4.5节，还有4.12节。
- 第5章：5.1～5.3节，还有5.5节。
- 第6章。
- 第7章：7.1～7.10节，还有7.13节。
- 第8章：8.1～8.5节。

传统教学方法

第2章和第4章介绍了布尔代数、组合逻辑电路和逻辑最小化的基本技术。第2章首次探讨与门、或门、非门、与非门以及或非门的话题。第3章详细地讨论了技术的实施，然后在第4章中再谈到综合技术和其他类型的门。我们认为，如果学生理解现存的与非门、或非门、异或门以及各种可编程逻辑器件的技术原理，会更好地理解第4章中的内容。

喜欢传统教学方法的教师，最好先讲第2章，接着讲解第4章。为了理解与非门、或非门以及异或门的使用，教师只需讲解这些门的功能性定义。

Verilog

Verilog 是一种复杂的语言，有些教师觉得对于初学者来说，Verilog太难掌握了。我们完全理解这一点，并努力设法解决。在课堂中介绍完整的Verilog语言是没有必要的。在本书中，介绍了对逻辑电路的综合和设计有用的重要的Verilog语句构造。许多其他的语言构造，例如只在仿真时才有意义的语句构造，书中没有介绍。本书中Verilog的内容是逐渐引入的，只有当有关Verilog内容在电路设计的某个关键点上能展示其先进的功能特性时，才将其引入。

本书中包括了150多个Verilog 代码的例子。这些例子说明如何用Verilog来描述各种逻辑电路，从只包含几个门的电路到表示数字系统的电路，例如简单的处理器。

问题求解

许多章都有问题求解举例。这些例子向读者展示了习题求解的典型方法。

家庭作业问题

本书提供了400多道家庭作业题，其中部分习题答案在书后给出。教师可以从McGraw-Hill网站（http://highered.mcgraw-hill.com/sites/0073380334/）上得到与本书配套的《解题手册》（Solution Manual），其中有这些家庭作业的标准答案。

实验室

本书可以用做不需要实验室练习的课程的教材。即使没有实验室，学生也可以使用与本书配套的计算机辅助设计工具，在计算机上对自己设计的电路进行仿真，获得有用的实践经验。如果有一个实验室，本书中的许多例子都适合在实验室实践。

教师可以从McGraw-Hill网站上获得PowerPoint幻灯片（包括书中的所有图）。

致谢

衷心感谢在本书编写期间给予我们帮助的人。Kelly Chan帮助完成了本书手稿的技术准备工作。Dan Vranesic绘制了书中的大量插图，他和Deshanand Singh 还帮助编写了《解题手册》的初稿。Tom Czajkowski检查了一些习题的答案。Jonathan Rose对时序问题的讨论提出了有益的建议。新泽西理工学院的William Barnes、北卡罗来纳州立大学的Thomas Bradicich、McGill大学的James Clark、佐治亚理工学院的Stephen DeWeerth、北卡罗来纳州立大学罗利分校的Clay Gloster,Jr.、皇后大学的Carl Hamacher、科罗拉多大学的Vincent Heuring、威斯康星大学的Yu Hen Hu、得克萨斯大学奥斯汀分校的Wei-Ming Lin、滑铁卢大学的Wayne Loucks、犹他大学的Chris Myers、加州大学戴维斯分校的Vojin Oklobdzija、罗切斯特理工学院的James Palmer、南加州大学的Gandhi Puvvada、Milwaukee工程学院的Teodoro Robles、波士顿大学的Tatyana Roziner、卡内基-梅隆大学的Rob Rutenbar、佛罗里达大学的Eric Schwartz、俄勒冈州立大学的Wen-Tsong Shiue、马里兰大学的Charles Silio, Jr.、密苏里大学罗拉分校的Scott Smith、艾奥瓦州立大学的Arun Somani、得克萨斯大学阿灵顿分校的Bernard Svihel、McGill大学的Zeljko Zilic，认真审阅了本书的初稿，提出了许多宝贵的意见和建设性的建议，对本书的修改和最后定稿起了很大作用。

我们也非常感谢Altera公司提供了Quartus II计算机辅助设计系统，特别是Misha Burich。McGraw-Hill出版公司给了我们很大的帮助和支持，衷心地感谢Michael Hackett、Brenda Rolwes、Darlene Schueller、April Southwood、Kris Tibbetts、Judi David和Michael Weitz的帮助。

Stephen Brown
Zvonko Vranesic

作者简介

Stephen Brown　在加拿大新布鲁斯威克（New Brunswick）大学获得电机工程学士学位，在多伦多大学获得电机工程硕士和博士学位。自1992年起在加拿大多伦多大学任教，目前他是该大学电机与计算机工程系教授，同时也是Altera公司多伦多技术中心的开发工程和大学计划部主任。

他的研究领域包括：现场可编程超大规模集成电路技术和计算机体系结构。他曾获得加拿大国家科学和工程研究会1992年度最佳博士论文奖，并发表了60多篇科研论文。

他曾四次荣获电机工程、计算机工程和计算机科学课程的最佳教学奖。除本书外，他还与人合写了《Fundamentals of Digital Logic with VHDL Design, 2nd ed.》（数字逻辑基础与VHDL设计，第2版）和《Field-Programmable Gate Arrays》（现场可编程门阵列）两本书。

Zvonko Vranesic　先后在加拿大多伦多大学获得电机工程学士、硕士和博士学位。1963~1965年间，他担任安大略布莱玛勒北方电器公司（Northern Electric Co. Ltd.）的设计师，1968年起在多伦多大学任教，目前他是该大学电机与计算机工程系以及计算机科学系教授。1978~1979年间，他在英国剑桥大学做高级访问学者。1984~1985年间，他在法国巴黎第六大学工作。1995~2000年，他担任加拿大多伦多大学工程科学部主任。他还参与Altera公司多伦多技术中心的研究和开发工作。

目前他的研究方向包括：计算机体系结构和现场可编程超大规模集成电路技术。

除本书外，他还参与了另外四本书的编写工作：《Computer Organization, 5th ed.》（计算机组成，第5版）、《Fundamentals of Digital Logic with VHDL Design, 2nd ed.》（数字逻辑基础与VHDL设计，第2版）、《Microcomputer Structures》（微型计算机体系结构）、《Field-Programmable Gate Arrays》（现场可编程门阵列）。1990年，他由于指导大学本科实验的创新和杰出贡献而获得威腾奖金（Wighton Fellowship）。2004年，他获得多伦多大学应用科学与工程系教学奖。

他曾经代表加拿大参加了多次国际象棋比赛，拥有国际象棋大师的头衔。

本书系统介绍数字逻辑基本概念与实际应用。主要内容包括：逻辑电路、组合逻辑、算术运算电路、存储元件、同步时序电路（有限状态机）、异步时序电路、测试等。本书内容全面，概念清楚，结合了逻辑设计最新技术的发展。

本书适合作为电子工程、通信工程、计算机等专业数字逻辑设计课程的教材或教学参考书，也可作为相关技术人员的参考书。

Stephen Brown and Zvonko Vranesic：Fundamentals of Digital Logic with Verilog Design, Second Edition (ISBN 13: 978-0-07-338033-9 ISBN 10: 0-07-338033-4).

Copyright © 2008 by The McGraw-Hill Companies, Inc.

Original English edition published by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press.

本书中文简体字翻译版由机械工业出版社和美国麦格劳-希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2007-4200

图书在版编目（CIP）数据

数字逻辑基础与Verilog设计（原书第2版）/（加）布朗（Brown, S.），（加）弗兰尼斯克（Vranesic, Z.）著；夏宇闻等译. -北京：机械工业出版社，2007. 10

（电子与电气工程丛书）

书名原文：Fundamentals of Digital Logic with Verilog Design, Second Edition

ISBN 978-7-111-22182-1

Ⅰ. 数… Ⅱ. ① 布… ② 弗… ③ 夏… Ⅲ. ① 数字逻辑-逻辑系统 ② 硬件描述语言，Verilog HDL-程序设计 Ⅳ. TP302. 2 TP312

中国版本图书馆CIP数据核字（2007）第136812号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：迟振春

北京诚信伟业印刷有限公司印刷 · 新华书店北京发行所发行

2008年1月第1版第1次印刷

184mm × 260mm · 31.25印张

定价：65.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

目　录

译者序
译者简介
序言
前言
作者简介

第1章　设计概念 ……1
1.1　数字硬件 ……1
1.1.1　标准芯片 ……2
1.1.2　可编程逻辑器件 ……2
1.1.3　定制芯片 ……3
1.2　设计过程 ……3
1.3　数字硬件的设计 ……4
1.3.1　基本设计循环 ……5
1.3.2　计算机的结构 ……5
1.3.3　数字硬件单元的设计 ……6
1.4　本书中的逻辑电路设计 ……8
1.5　理论和实践 ……8
参考文献 ……9
第2章　逻辑电路入门 ……10
2.1　变量和函数 ……10
2.2　反相 ……12
2.3　真值表 ……12
2.4　逻辑门和逻辑网络 ……13
2.5　布尔代数 ……15
2.5.1　维恩图 ……18
2.5.2　符号和术语 ……20
2.5.3　运算的优先级别 ……20
2.6　用与门、或门和非门进行综合 ……21
2.7　与非以及或非逻辑网络 ……26
2.8　设计举例 ……29
2.8.1　三路灯光控制 ……29
2.8.2　多路选择器电路 ……30
2.9　计算机辅助设计工具简介 ……31
2.9.1　设计输入 ……32
2.9.2　综合 ……33
2.9.3　功能仿真 ……33
2.9.4　物理设计 ……33
2.9.5　时序仿真 ……34
2.9.6　芯片配置 ……34
2.10　Verilog 简介 ……35
2.10.1　逻辑电路的结构描述 ……35
2.10.2　逻辑电路的行为描述 ……37
2.10.3　编写Verilog代码必须注意的关键点 ……38
2.11　小结 ……39
2.12　问题求解举例 ……39
练习题 ……42
参考文献 ……45
第3章　实现技术 ……46
3.1　晶体管开关 ……46
3.2　NMOS逻辑门 ……48
3.3　CMOS逻辑门 ……50
3.4　负逻辑系统 ……53
3.5　标准芯片 ……55
3.6　可编程逻辑器件 ……56
3.6.1　可编程逻辑阵列 ……57
3.6.2　可编程阵列逻辑 ……58
3.6.3　PLA和PAL的编程 ……60
3.6.4　复杂可编程逻辑器件 ……61
3.6.5　现场可编程门阵列 ……63
3.6.6　用CAD工具在CPLD 和FPGA上实现逻辑电路 ……66
3.6.7　CPLD和FPGA的应用 ……66
3.7　定制芯片、标准单元和门阵列 ……66
3.8　实际问题 ……68
3.8.1　MOSFET晶体管的制造和行为 ……68
3.8.2　MOSFET晶体管的导通电阻 ……71

3.8.3 逻辑门的电平 ……71
3.8.4 噪声容限 ……72
3.8.5 逻辑门的动态操作 ……73
3.8.6 逻辑门的功率消耗 ……75
3.8.7 通过晶体管开关传递1和0 ……76
3.8.8 逻辑门的扇入和扇出 ……77
3.9 传输门 ……81
3.9.1 异或门 ……81
3.9.2 多路选择器电路 ……82
3.10 SPLD、CPLD和 FPGA 的实现细节 ……82
3.11 小结 ……88
3.12 问题求解举例 ……88
练习题 ……93
参考文献 ……98
第4章 逻辑函数的优化实现 ……99
4.1 卡诺图 ……99
4.2 最小化策略 ……104
4.2.1 专业术语 ……105
4.2.2 最小化步骤 ……106
4.3 和之积形式的化简 ……108
4.4 非完全指定函数 ……109
4.5 多输出电路 ……110
4.6 多级综合 ……112
4.6.1 提取公因子 ……113
4.6.2 函数分解 ……115
4.6.3 多级与非以及或非电路 ……119
4.7 多级电路的分析 ……120
4.8 立方体表示法 ……124
4.9 列表法化简 ……126
4.9.1 质蕴涵项的产生 ……127
4.9.2 最小覆盖的确定 ……128
4.9.3 列表法小结 ……131
4.10 使用立方体表示法最小化函数 ……132
4.10.1 本质蕴涵项的确定 ……133
4.10.2 求解最小覆盖的完整步骤 ……135
4.11 一些实际问题的考虑 ……137
4.12 由Verilog代码综合得到电路举例 ……137
4.13 小结 ……140
4.14 问题求解举例 ……140
练习题 ……144
参考文献 ……148
第5章 数的表示和算术电路 ……150
5.1 数位的表示法 ……150
5.1.1 无符号整数 ……150
5.1.2 十进制数与二进制数之间的转换 ……151
5.1.3 八进制数和十六进制数的表示 ……151
5.2 无符号数的加法 ……153
5.2.1 全加器的分解 ……155
5.2.2 行波进位加法器 ……155
5.2.3 设计举例 ……156
5.3 有符号数 ……157
5.3.1 负数 ……157
5.3.2 加法和减法 ……158
5.3.3 加法器和减法器单元 ……160
5.3.4 基数补码方案 ……161
5.3.5 算术溢出 ……164
5.3.6 电路的性能问题 ……164
5.4 快速加法器 ……165
5.5 使用CAD工具设计算术电路 ……170
5.5.1 使用原理图编辑工具设计算术电路 ……170
5.5.2 使用Verilog设计算术电路 ……172
5.5.3 使用向量信号 ……173
5.5.4 使用自动生成语句 ……174
5.5.5 Verilog中的线网和变量 ……175
5.5.6 算术赋值语句 ……176
5.5.7 Verilog代码中数的表示 ……178
5.6 乘法 ……180
5.6.1 无符号数的阵列乘法器 ……181
5.6.2 有符号数的乘法 ……182
5.7 数的其他表示方法 ……182
5.7.1 定点数 ……183
5.7.2 浮点数 ……183
5.7.3 二-十进制编码 ……184
5.8 ASCII字符码 ……186
5.9 问题求解举例 ……188

练习题 ……191
参考文献 ……193
第6章 组合电路构件块 ……194
6.1 多路选择器 ……194
6.1.1 用多路选择器的逻辑函数的综合 ……197
6.1.2 用香农展开的多路选择器综合 …198
6.2 译码器 ……202
6.3 编码器 ……205
6.3.1 二进制编码器 ……205
6.3.2 优先级编码器 ……206
6.4 码型转换器 ……207
6.5 算术比较电路 ……207
6.6 用Verilog表示组合电路 ……208
6.6.1 条件操作符 ……208
6.6.2 if-else 语句 ……209
6.6.3 case 语句 ……211
6.6.4 for 循环语句 ……215
6.6.5 Verilog操作符 ……216
6.6.6 生成结构 ……220
6.6.7 任务和函数 ……220
6.7 小结 ……222
6.8 问题求解举例 ……223
练习题 ……228
参考文献 ……230
第7章 触发器、寄存器、计数器和简单处理器 ……232
7.1 基本锁存器 ……233
7.2 门控SR锁存器 ……234
7.3 门控D锁存器 ……236
7.4 主从D触发器和沿触发的D触发器 …238
7.4.1 主从D触发器 ……238
7.4.2 沿触发的D触发器 ……239
7.4.3 有清零端和预置信号的D触发器 ……241
7.4.4 触发器的时序参数 ……242
7.5 T触发器 ……243
7.6 JK触发器 ……244
7.7 术语小结 ……244
7.8 寄存器 ……245
7.8.1 移位寄存器 ……245
7.8.2 并行存取的移位寄存器 ……246
7.9 计数器 ……247
7.9.1 异步计数器 ……247
7.9.2 同步计数器 ……248
7.9.3 可并行置数的计数器 ……251
7.10 同步复位 ……252
7.11 其他类型的计数器 ……254
7.11.1 BCD计数器 ……254
7.11.2 环形计数器 ……255
7.11.3 Johnson计数器 ……256
7.11.4 计数器设计要点 ……256
7.12 用CAD工具在设计中加入存储元件 ……256
7.12.1 在电路原理图中添加存储元件 ……256
7.12.2 用Verilog代码实现存储元件 …258
7.12.3 阻塞赋值和非阻塞赋值 ……259
7.12.4 组合逻辑电路的非阻塞赋值 …261
7.12.5 具有清零功能的触发器 ……261
7.13 用CAD工具在设计中加入寄存器和计数器 ……262
7.13.1 在电路原理图中添加寄存器和计数器 ……262
7.13.2 在Verilog代码中使用库模块 …264
7.13.3 在Verilog代码中使用寄存器和计数器结构 ……264
7.14 设计举例 ……267
7.14.1 总线结构 ……267
7.14.2 简单的处理器 ……275
7.14.3 反应计时器 ……283
7.14.4 寄存器传输级代码 ……286
7.15 触发器电路的时序分析 ……286
7.16 小结 ……288
7.17 问题求解举例 ……289
练习题 ……292
参考文献 ……296
第8章 同步时序电路 ……298

8.1 基本设计步骤……299
8.1.1 状态图 ……299
8.1.2 状态表 ……300
8.1.3 状态分配 ……300
8.1.4 触发器的选择以及下一个状态和输出表达式的推导 ……301
8.1.5 时序图 ……302
8.1.6 设计步骤小结 ……303
8.2 状态分配问题……306
8.3 米利型状态模型……308
8.4 用CAD工具设计有限状态机 ……311
8.4.1 摩尔型有限状态机的Verilog代码 ……312
8.4.2 Verilog代码的综合 ……312
8.4.3 电路的仿真和测试 ……314
8.4.4 另一种风格的Verilog代码……314
8.4.5 用CAD工具的设计步骤小结……316
8.4.6 在Verilog代码中进行状态分配…316
8.4.7 用Verilog语句来编写米利型有限状态机 ……317
8.5 串行加法器举例……318
8.5.1 用米利型有限状态机实现的串行加法器 ……318
8.5.2 用摩尔型有限状态机实现的串行加法器 ……320
8.5.3 串行加法器的Verilog 代码 ……321
8.6 状态最小化……324
8.6.1 划分最小化的步骤 ……324
8.6.2 未完全指定的有限状态机……328
8.7 用时序电路方法设计计数器 ……329
8.7.1 模8计数器的状态图和状态表 …330
8.7.2 状态分配 ……330
8.7.3 用D触发器实现的计数器 ……331
8.7.4 用JK触发器实现的计数器……332
8.7.5 举例：一个不同的计数器……335
8.8 用作仲裁器电路的有限状态机 ……336
8.8.1 仲裁器电路的实现 ……338
8.8.2 有限状态机输出延迟的最小化…339
8.8.3 小结 ……340
8.9 同步时序电路的分析 ……340
8.10 算法状态机图 ……343
8.11 时序电路的形式化模型 ……345
8.12 小结 ……346
8.13 问题求解举例……346
练习题 ……352
参考文献 ……355
第9章 异步时序电路 ……356
9.1 异步行为 ……356
9.2 异步电路分析……358
9.3 异步电路综合……363
9.4 状态化简 ……372
9.5 状态分配 ……380
9.5.1 转移图 ……381
9.5.2 未指定的下一个状态项的利用…383
9.5.3 使用附加状态变量进行的状态分配 ……386
9.5.4 独热状态分配 ……388
9.6 冒险 ……389
9.6.1 静态冒险 ……390
9.6.2 动态冒险 ……393
9.6.3 冒险的意义 ……394
9.7 一个完整的设计实例 ……394
9.8 小结 ……398
9.9 问题求解举例 ……399
练习题 ……403
参考文献 ……406
第10章 数字系统设计 ……407
10.1 构件块电路 ……407
10.1.1 有使能输入的触发器和寄存器…407
10.1.2 有使能输入的移位寄存器 ……408
10.1.3 静态随机存取存储器 ……408
10.1.4 可编程逻辑器件中的SRAM块…410
10.2 设计举例 ……410
10.2.1 位计数电路 ……410
10.2.2 算法状态机图蕴涵的时序信息……411
10.2.3 移位相加实现的乘法器 ……413

10.2.4 除法器 ……418
10.2.5 算术平均值 ……425
10.2.6 排序操作 ……428
10.3 时钟同步 ……435
10.3.1 时钟偏差 ……436
10.3.2 触发器的时序参数 ……436
10.3.3 触发器的异步输入 ……438
10.3.4 开关的抖动 ……438
10.4 小结 ……439
练习题 ……439
参考文献 ……442
第11章 逻辑电路测试 ……443
11.1 故障模型 ……443
11.1.1 固滞模型 ……443
11.1.2 单个故障和多个故障 ……443
11.1.3 CMOS电路 ……444
11.2 测试集的复杂度 ……444
11.3 路径的敏感化 ……445
11.4 树状结构的电路 ……448
11.5 随机测试 ……448
11.6 时序电路的测试 ……450
11.7 内建自测试 ……453
11.7.1 内建逻辑块观察器 ……456
11.7.2 签字分析 ……457
11.7.3 边界扫描 ……457
11.8 印制电路板 ……458
11.8.1 印制电路板的测试 ……459
11.8.2 测试仪器 ……460
11.9 小结 ……460
练习题 ……461
参考文献 ……462
第12章 计算机辅助设计工具 ……464
12.1 综合 ……464
12.1.1 网表的生成 ……465
12.1.2 门的优化 ……465
12.1.3 技术映象 ……467
12.2 物理设计 ……469
12.2.1 布局 ……469
12.2.2 布线 ……470
12.2.3 静态时序分析 ……471
12.3 小结 ……472
参考文献 ……472
部分习题答案 ……474

* 附录部分见随书附带光盘。——编辑注

第1章 设计概念

本书讲述逻辑电路，即构造计算机的逻辑电路。深入理解逻辑电路，对今天的电子工程师和计算机工程师至关重要。逻辑电路是构造计算机的要素，也可应用于许多其他场合。逻辑电路不仅在日用产品，例如数字电子表、各种家用电器、CD播放机、电子游戏机等产品中使用，而且在大型系统，例如电话和电视网络的设备中，更是广泛应用。

本书将向读者介绍许多有关逻辑电路设计的问题。作者用简单的例子来解释要点，展示复杂电路是如何由基本电路组成的，并深入探讨有关逻辑电路设计的经典理论，以帮助读者直观地理解逻辑电路的本质。但是，贯穿全书还介绍设计逻辑电路的先进方法，即使用高级计算机辅助设计（CAD）软件，在本书中这种软件工具是基于工业标准设计语言Verilog HDL的。基于Verilog的设计方法从第2章就开始介绍，同时在本书中，Verilog语言和CAD工具的具体应用与每一章的知识紧密结合成为一个整体。

逻辑电路是通过集成电路芯片上的晶体管来实现电子功能的。现代技术已能在芯片上集成数以亿计的晶体管，例如计算机的处理器。构造这些电路的基本模块是很容易理解的，但是对于包含几亿个晶体管的电路来说，就不那么容易理解了。因规模庞大带来的逻辑电路的复杂性必须由高度结构化的设计技术来解决。我们将在本章介绍这些技术，但首先要简要地介绍构造逻辑电路的硬件技术。

1.1 数字硬件

逻辑电路用于构造计算机硬件，以及多种其他产品。所有这类产品广义上都归为数字硬件。使用“数字的”这个形容词的原因将在本书随后的部分中阐明，它源于计算机中表达信息的方式，即将数字信息表示为电信号。

用于构造数字硬件的技术在过去的40年间惊人地发展着。直到20世纪60年代，逻辑电路都是由体积较大的分立元件（例如晶体管和电阻）所构成。集成电路的出现使得我们有可能把一些晶体管甚至整个电路集成到单个芯片上。起初这些电路只有几个晶体管，随着技术的进步，集成的晶体管数量越来越多。集成电路芯片是在硅圆晶片上制造的，如图1-1所示。硅圆晶片被切割开成为许多个片芯，然后再把片芯放置到专门的芯片封装内。到20世纪70年代已可将实现微处理器必须的所有电路置于一块片芯上。尽管以今天的眼光看来，早期的微处理器计算能力有限，但是它们为廉价个人电脑的实现提供了手段，从而打开了信息处理革命的大门。大约在30年前，Intel公司的总裁戈登·摩尔（Gordon Moore）先生就观察到，集成电路技术正在以单个芯片上集成的晶体管数量每一年半或两年就翻一番的惊人速度发展着。这种现象，俗称摩尔定律，直到今天仍在延续。在20世纪90年代初，已经可以制造包含几百万个晶体管的微处理器，而到20世纪90年代末，已有可能制造包含千万个晶体管的芯片。如今的芯片已可集成超过十亿个晶体管。

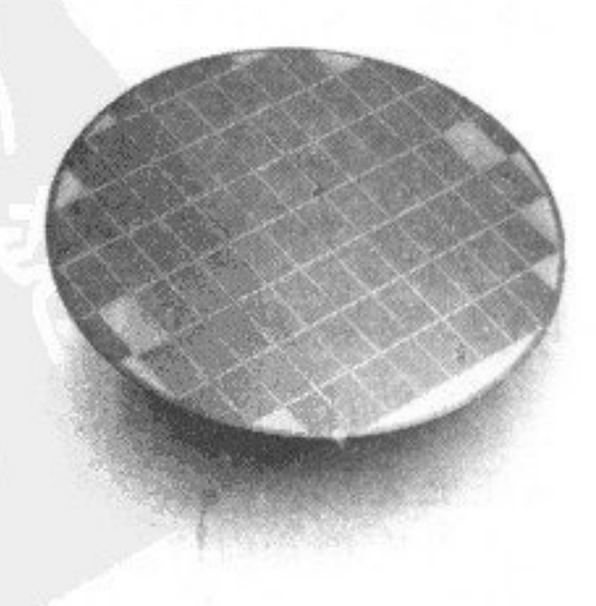


图1-1 一片硅圆晶片
（感谢Altera公司授权）

摩尔定律被认为至少在下一个十年仍能继续适用。一个

集成电路协会联盟提出了一份对技术发展趋势的评估报告，即著名的“半导体国际技术路标”(International Technology Roadmap for Semiconductors, ITRS) [1]，它预测了集成电路芯片上所能安装的晶体管的最小尺寸。晶体管尺寸是由一个称为“门长度”的参数来测量的，这一点我们将在第3章讨论。表1-1举例说明了ITRS。2006年能够可靠生产的最小门长度是78nm。表的第一行指出，到2012年，最小门长度将会稳定地降至约36nm。晶体管的最小门长度决定了在给定的芯片区域上能集成多少个晶体管。从表中可以看出，2006年已经达到每平方厘米可集成28.3亿个。到2012年，这个数字将会增至113.3亿。那时最大的芯片尺寸估计将达858mm^2，也就是说将有可能出现晶体管个数高达100亿的芯片！无疑，这种技术将对人们的生活各方面产生巨大的影响。

表1-1 ITRS示例

	年					
	2006	2007	2008	2009	2010	2012
晶体管门长度	78nm	68nm	59nm	52nm	45nm	36nm
每平方厘米晶体管数	283M	357M	449M	566M	714M	1133M
每芯片晶体管数	2430M	3061M	3857M	4859M	6122M	9718M

数字硬件的设计者也许会面对这样的逻辑电路设计问题，即设计能在单个芯片上实现的电路，或者更有可能的是设计安装在一块印制电路板（PCB）上由多个芯片实现的电路。通常，一些逻辑电路可以用现成的商业芯片实现。这就简化了设计任务，缩短了开发最终产品的时间。在详细讨论设计过程之前，我们应该介绍一下可能会用到的各种集成电路。

商业芯片有很多种，可以用来实现数字硬件设计中有用的功能。从非常简单、功能很少的芯片到极其复杂的芯片一应俱全。例如，某数字硬件产品可能需要微处理器来执行一些算术运算功能，也需要接口芯片很容易地完成与输入/输出设备之间的连接。这样的芯片可以在不同的厂商那里买到。

对于大多数数字硬件产品来说，根据草图来设计和构造逻辑电路也是必要的。实现这样的电路常用到的芯片主要有三种：标准芯片、可编程逻辑器件、定制芯片。这些将在下面讨论。

1.1.1 标准芯片

有许多种现成的芯片可用于常用逻辑电路的实现。我们将这类芯片称为*标准芯片*，因为它们通常在功能和规格上均符合公认的标准。每个标准芯片都包含有少量的电路（通常不到100个晶体管），能执行简单的功能。为了构造逻辑电路，设计者选择能完成所需功能的芯片，然后确定这些芯片应如何连接，以实现更大规模的逻辑电路。

直到20世纪80年代初，设计逻辑电路时选用标准芯片还是很流行的。然而，随着集成电路技术的进步，在PCB上有限的宝贵空间里，使用功能很少的芯片就显得效率低下了。标准芯片的另一缺点就是每个芯片的功能是固定的，无法更改。

1.1.2 可编程逻辑器件

我们有可能构建这样的芯片，它与功能固定的标准芯片不同，内含可由用户配置的电路，可在更大范围内实现不同的逻辑电路。这些芯片具有通用化的结构，包括一个可编程开关集合，允许用户以多种方式修改芯片内部的电路。设计者可以通过适当选择开关的配置来实现在特定应用中所需的功能。开关并不是在制造的时候就进行编程，而是由最终用户编程。这些芯片被称为*可编程逻辑器件*（PLD）。我们将在第3章中介绍它们。

大多数型号的可编程逻辑器件都可以进行多次编程。这个功能非常有用，因为设计者最初设计的是产品的原型，在此后的原型硬件测试运行过程中，可以发现问题，并通过对PLD再编程，进行设计的修正。再编程很可能是必不可少的，例如设计的功能与起初的设计意图不尽相符时，或者需要在原设计中添加原来没有考虑到的新功能时，都需要对PLD再次编程。

我们可以购买到各种不同规格（大小）的可编程逻辑器件。它们可用来实现典型标准芯片无法实现的复杂大型逻辑电路。可编程逻辑器件如今应用非常广泛，这是因为它们可实现的逻辑规模很大，并且可进行裁剪以适应特殊用途的需要。最先进的可编程逻辑器件之一，叫做*现场可编程门阵列*（FPGA）。几亿个晶体管的FPGA现在已经出现[2, 3]。图1-2中所示的为一个FPGA芯片。该芯片由大量微型逻辑电路元件构成，均可用可编程开关进行连接。逻辑电路元件被安排在规则的二维结构中。

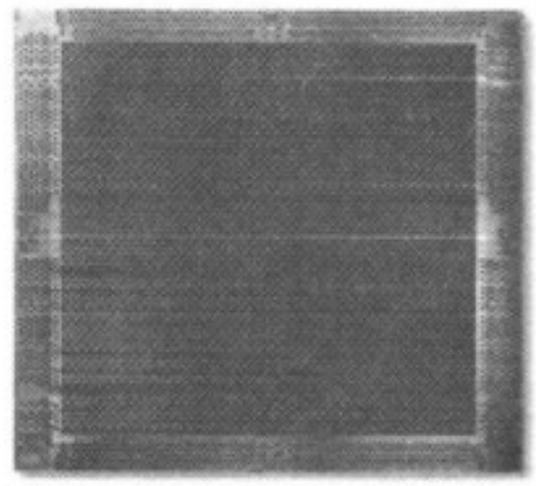


图1-2 现场可编程门阵列（感谢Altera公司授权）

1.1.3 定制芯片

可编程逻辑器件（PLD）作为现货供应的商业元件，可以从各供应厂商采购到。由于可编程逻辑器件具有可编程性，所以可用PLD来实现数字硬件中的大多数逻辑电路。然而可编程逻辑器件也有不足，它不但消耗了许多宝贵的芯片空间，而且电路实际运行的速度也受到限制。因此在某些情况下，PLD也许不能达到预期的性能或成本目标。在这种情况下，可能需要从头开始设计芯片。也就是说，先设计芯片上的逻辑电路，再选择适当的技术来制造芯片。最后，芯片在拥有制造设备的公司进行生产。这个过程即所谓*定制设计*或*半定制设计*，这类芯片也称为*定制芯片*或*半定制芯片*。此类芯片是专为特殊应用所生产的，有时也被称为*专用集成电路*（ASIC）。

定制芯片最大的优点在于：可以针对特定任务做最优化设计。因此常常能够达到更高的性能，而且定制芯片中有可能比其他类型芯片集成更多的逻辑电路。虽然这种芯片的生产成本很高，但是如果用在销售量大的产品中，将成本平均分摊至每个芯片，则每个芯片的成本可能低于功能相同的现货供应芯片。此外，如果可以用单个芯片来替代多个芯片，则最终产品的印制电路板所需的芯片装配空间将可以进一步降低，从而降低了产品成本。

定制设计芯片的解决方案美中不足之处在于：制造定制芯片通常耗时不菲，需要数月才能完成。相比之下，如果使用可编程逻辑器件，芯片可由最终用户编程，无需耽搁制造时间。

1.2 设计过程

计算机辅助设计工具的出现与广泛使用，对多种设计环境下的设计过程都产生了重大的影响。举例来说，设计汽车的基本方法和设计熔炉或电脑的基本方法十分相似。为了确定最终产品是否能达到特定的设计目标，设计周期中的某些步骤是不可缺少的。我们将用最常用的术语，从介绍典型的设计开发周期开始，然后着重讲解与逻辑电路设计有关的特殊流程。

图1-3中的流程图描述了典型的设计过程。我们假定这个设计过程是为了开发一项符合某确定要求的产品。最明显的要求就是，产品必须运作正常，必须达到预期的性能等级，而且成本还不能超过预期的目标。

设计过程从定义产品规范开始。产品的基本特征一旦确定，评估最终产品所实现的功能是否可接受的方法也就建立起来了。产品规范必须足够严格，以确保开发的产品能全面地达

到预期的要求，但是不应有不必要的死板约束（也就是说，规范不应阻止有可能导致意外好处的设计选择）。

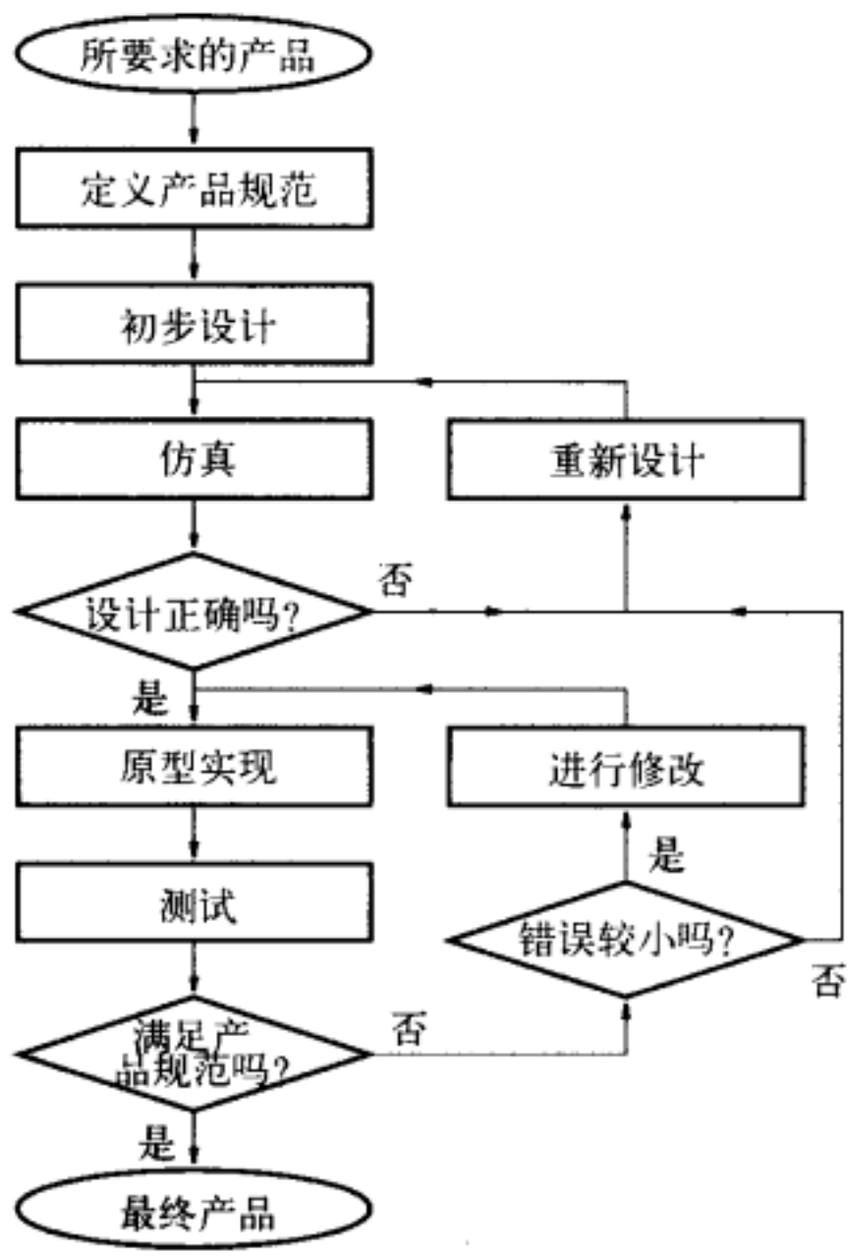


图1-3 设计过程

从一整套完整的技术指标出发，详细地确定产品的初步设计总体框架是很有必要的。这一步很难由自动化工具进行。通常是由经验丰富的设计师亲自思考决定，因为目前还没有清晰的开发产品整体框架的策略，这项工作需要具有相当丰富的设计经验积累和直觉。

在总体框架建立之后，就能用计算机辅助设计（CAD）工具来完成细节的设计。CAD工具有许多种，从辅助设计系统的个别部件到能在计算机中描述整个系统结构的，一应俱全。当最初设计完成后，必须根据最初所设定的技术指标对结果进行验证。在计算机辅助设计工具面世之前，这一步往往涉及建立所设计产品的物理模型，通常只包含关键部件。今天基本上用不着建立物理模型了。CAD工具使得设计者可以对极其复杂产品的行为进行仿真，通过这样的仿真来确定设计是否符合必须的技术指标。如果发现了问题，进行适当调整，通过仿真对新修改的设计再重复进行验证。尽管一些最难发现的问题通过仿真也无法检测到，但几乎所有问题通常都可以用仿真的方法予以解决。

当仿真表明设计正确无误时，需要为产品建造一个完整的物理原型样机。对原型样机要进行彻底的测试，以确保与指标的一致性。测试中发现的任何错误必须加以修正。若发现的错误不大，通常可以通过直接在产品原型样机上进行小的修改来消除错误。若出现大的错误，就有必要重新设计产品，重复上述仿真测试过程。当原型样机通过所有测试后，就可认为设计取得成功，便可以投产了。

1.3 数字硬件的设计

我们前面以最普通的方式讨论了设计过程。图1-3描绘的步骤完全适用于数字硬件的整个

开发流程。在开始讨论整个开发流程之前，我们还应强调一下设计过程的迭代性。

1.3.1 基本设计循环

任何设计过程都由不同情形下按照顺序进行的一系列基本任务所组成，这个顺序系列如图1-4所示。假设我们对设计过程中应该取得的目标已经有了最初的概念，那么设计工作的第一步就是开始初步设计。这一步通常需要大量的人工：因为大多数的设计都有某些特定目标，只能靠设计者的知识、技术和直觉来完成。下一步就是对手头的初步设计进行仿真。这一步可以借助优秀的CAD工具来设计。为了成功地进行仿真，还需要在仿真时施加适当的输入条件，将来测试最终产品时亦然。施加这些输入条件后，仿真器就会依照最初的设计要求验证所设计的产品。如果仿真显示存在故障，那就必须修改设计来克服这些问题。重新设计后的版本还需要仿真，直到仿真成功为止。谨慎的设计者在仿真过程中会致力于修正错误，因为如果在设计过程的后期再发现错误，将会显著增加修改工作的难度。尽管如此，仍难免有一些错误从仿真中漏网，如遇到这种情况，只好留待在开发周期较后面的阶段处理了。

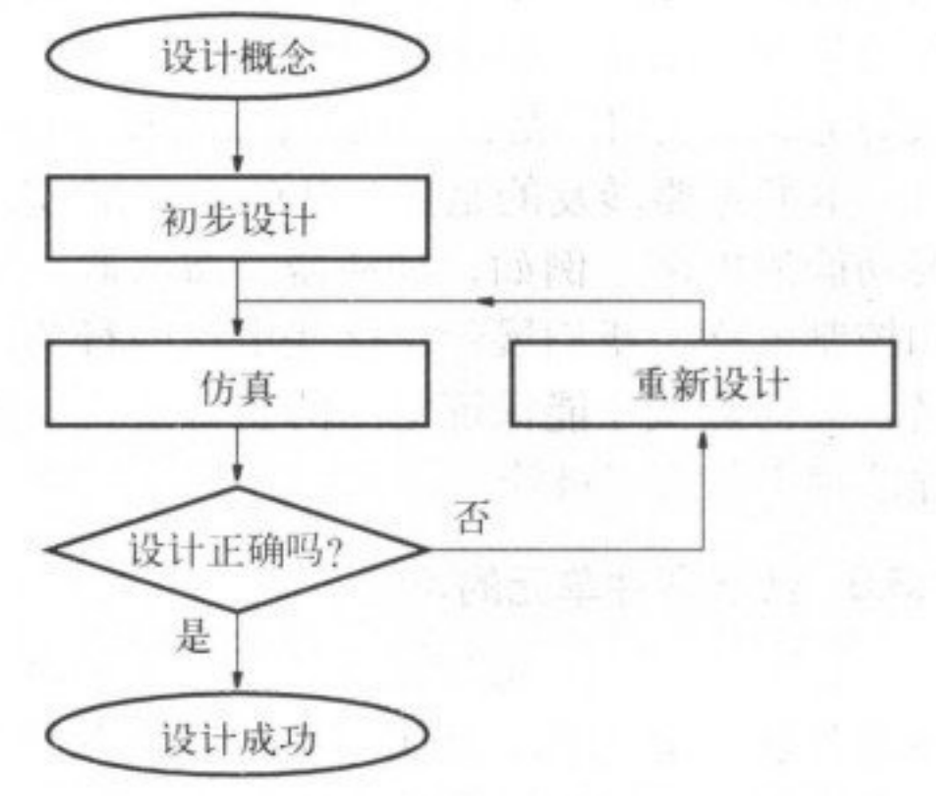


图1-4 基本设计过程

1.3.2 计算机的结构

为了理解逻辑电路在数字系统中的作用，请考虑如图1-5a所示的典型计算机结构。计算

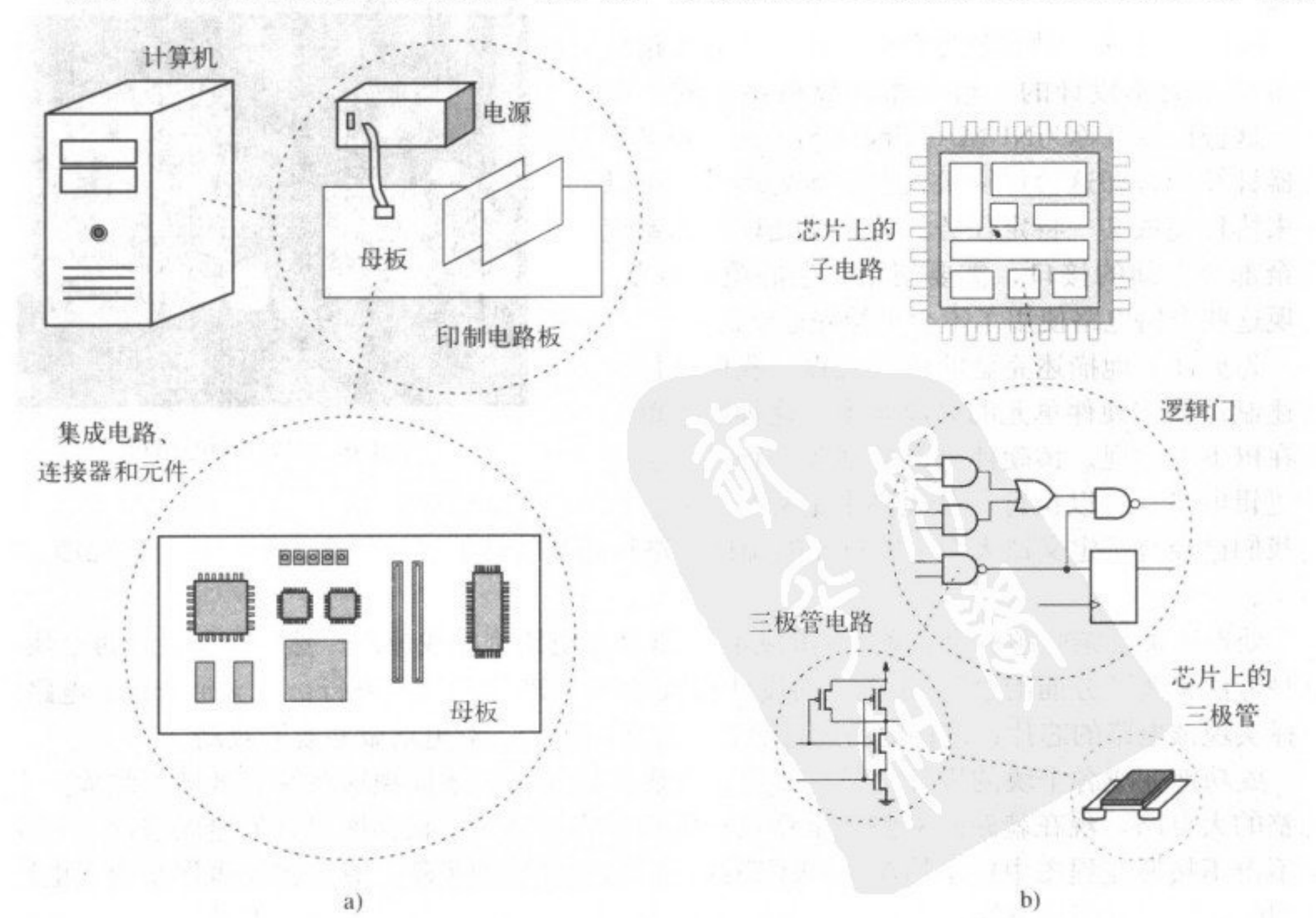


图1-5 数字硬件系统

机的机箱中有许多块印制电路板、一个电源和许多个存储单元（例如硬盘、DVD 或者CD-ROM 驱动器等），我们在图上没有画出这些单元。每个单元均插入到主印制电路板（主板）上，如图1-5a底部的图形所示。母板上还安装有几块集成电路芯片，还有一些插槽用于连接其他印制电路板，例如声卡、显卡和网卡。

图1-5b说明了集成电路芯片的结构。该芯片包含许多个子电路，它们之间互相连接组成了完整的电路。子电路有许多种，例如，进行算术运算的子电路、存储数据的子电路、控制数据流动的子电路。每个子电路都是逻辑电路。如图1-5b中间所示，逻辑电路是由互相连接的逻辑门构成的网络。每个逻辑门完成一个非常简单的功能，许多门连接在一起可以实现比较复杂的运算。逻辑门是由三极管组成的，而三极管是由制造在硅芯片上的许多层材料实现的。

本书主要涉及的是图1-5b的中间部分，即逻辑电路的设计。我们讲解如何设计能完成重要功能的电路，例如，加法器、减法器、乘法器和计数器电路，数据储存电路以及信息处理和控制电路。我们展示了这些电路的行为，特别关注如何设计价格最低、运行速度最快的电路，如何测试才能保证电路的正确运行。我们还简明地阐述了三极管的工作原理以及如何在硅芯片上制造三极管。

1.3.3 数字硬件单元的设计

如图1-5所示，数字硬件产品通常包含一块或多块印制电路板（PCB），每块PCB都包含许多芯片和其他元件。这类产品的开发从定义整体架构开始，然后选择所需的集成电路芯片，并设计用来安装和连接芯片的PCB。如果所选的芯片包括了可编程逻辑器件（PLD）或定制芯片，那么这些芯片的设计必须在PCB级的设计开始之前完成。由于在单个芯片和电路板上实现的电路通常极其复杂，充分运用CAD工具十分必要。

图1-6所示为一块印制电路板。这块印制电路板是多伦多大学设计的一台大型计算机系统的一部分。这台计算机称为NUMAchine[4,5]，是一台多处理器计算机，即这台计算机包含许多处理器，可以用来协作完成某一特定任务。为实现处理器和系统其余部分之间的接口，需要使用复杂的逻辑电路。实现这些逻辑电路使用了不少可编程逻辑器件。

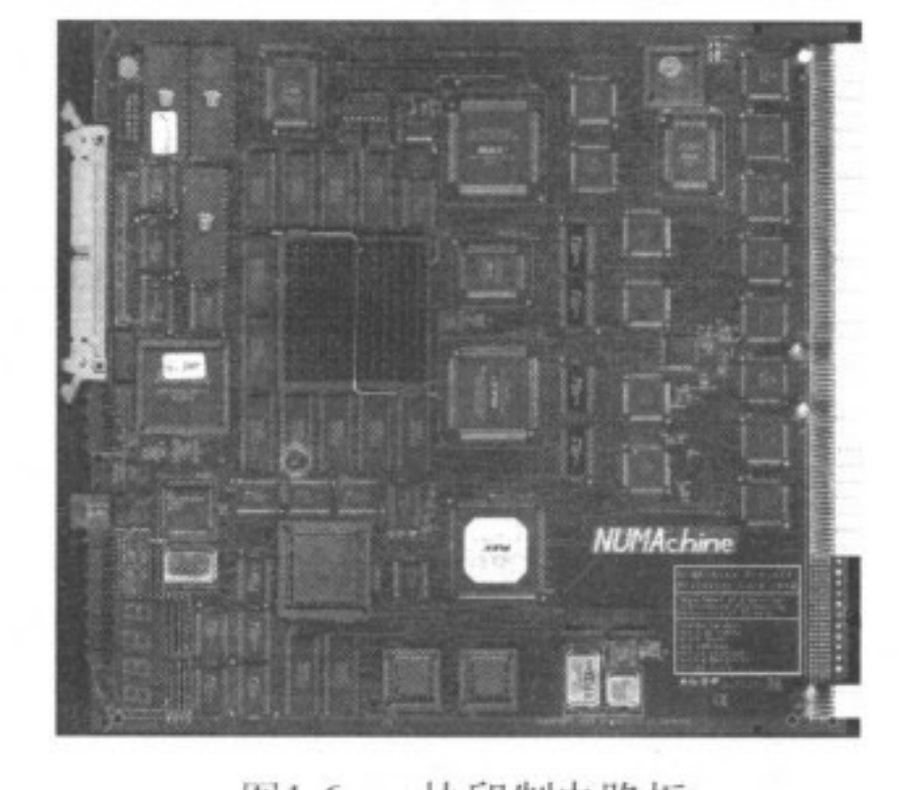


图1-6 一块印制电路板

为更详细地描述完整的设计流程，我们将仔细描述制造数字硬件单元的必要步骤，这种数字单元能在PCB上实现。该硬件可以被视为一种非常复杂的逻辑电路，可以执行产品规格中定义的功能。假定我们已经有了定义该大型电路单元的预期电路行为及特性的设计概念，则设计流程如图1-7所示。

处理这类复杂问题有个有条不紊的办法，就是把电路分割成小块，然后分别设计每个块，这种方法称为“分而治之”。每个块的设计都依照图1-4所示的过程进行：定义每个块的电路，选择实现该电路的芯片；进行电路运行仿真，发现问题后，对电路做必要的改动。

成功地完成各个块的设计之后，必须定义块间的连接，才能把这些块有效地组织成一个完整的大电路。现在就轮到对整个电路进行仿真并修正错误。根据所遇到的错误情况，可能会不得不按照流程图中所示的A、B或C路线，倒退到前面的步骤。有些错误也许是由于电路块间的连接不当而造成的，若如此，则如路径C所示，需要重定义连接。有些块也许设计本身

有误，若如此，则依路径B所示，重新设计有误的块。还有一个可能，就是在把大电路分割成小模块时工作没有做好，若如此，则依路径A所示处理。如果没有一块电路能实现整个电路必须的某些功能，这种情形就有可能发生。

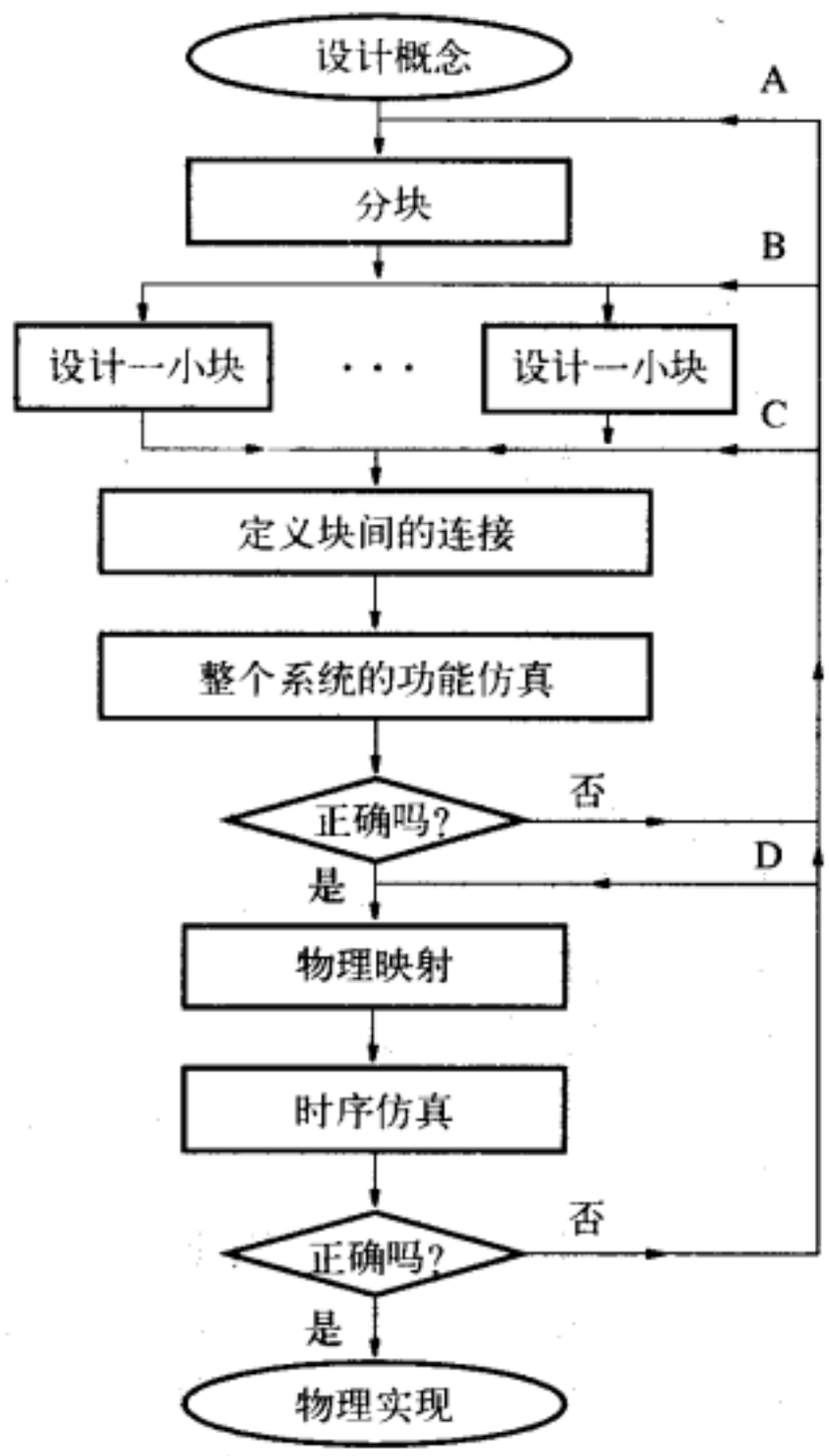


图1-7 逻辑电路的设计流程

功能仿真的成功意味着所设计的电路将会正确地执行其全部功能。下一步就是决定怎么在印制电路板上实现它。每个芯片在板上的物理位置以及芯片间的连线方案都需要预先定好。我们把这一步称为PCB的*物理设计*。这个任务主要依靠CAD工具自动完成。

一旦芯片安装到PCB上，电路板的连接线也得以实现，我们就想了解物理布线是如何影响成品电路板性能的。不妨假设前面的功能仿真表明所有的功能均能正确运行，则在物理设计步骤使用的CAD工具将能保证：必要的功能行为不会由于在电路板上安装芯片，用连线把它们连接起来，最终实现电路而遭到破坏。然而，就算功能行为正确，所实现的电路仍然可能远比预想中的电路运行速度慢，这就会导致性能不能充分发挥。这种情况的发生是由于在PCB上布线的时候必须用到金属箔线路，这对于电信号来说相当于存在电阻和电容，因此可能会对运行速度造成显著的影响。为了把只考虑电路功能的仿真和也考虑到时序的仿真区别开来，通常习惯使用“*功能仿真*”和“*时序仿真*”这两个术语。时序仿真可以揭示潜在的运行性能问题，如果速度性能与设计期望要求还有差距，可以用CAD工具在PCB的物理设计上进行改动，以期解决。

完成设计过程后，设计的电路就等着进行物理实现了。实现物理原型样机必须的步骤见图1-8。制造第一版电路板并进行测试。大多数发现的小错误通常可以通过直接在原型样机板

上作改动来修正。这也许会牵涉到布线的修改，甚至需要重新对某些PLD编程。更大些的问题需要更深层次的重新设计。根据问题的具体情况，设计者也许需要回到设计过程中的A、B、C或者D点（如图1-7所示）。

我们已经描述了使用许多块芯片在PCB上实现最终电路的开发过程。本书中讲述的材料可直接应用于这类设计问题。然而，为了从实际出发，本书中出现的设计示例都相对较小，全部可以在单个集成电路（不管是定制芯片还是PLD芯片）中实现。尽管书中示例的设计电路块的规模更小，图1-7中所有的设计步骤大体上都是针对这类可在单个芯片上实现的设计规模。

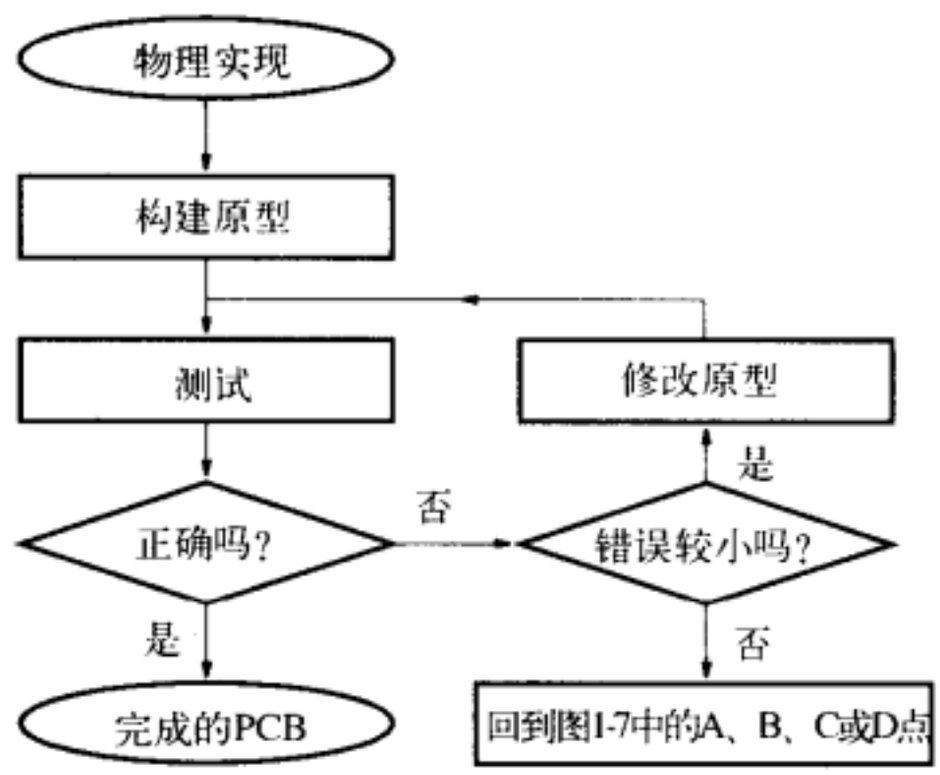


图1-8 完成PCB开发

1.4 本书中的逻辑电路设计

本书中广泛使用了PLD，以描述逻辑电路设计的各个方面。我们之所以选择PLD技术，是由于它在实际数字硬件产品中得到广泛应用，还由于PLD芯片是客户可编程的。PLD技术用于教学尤其合适，这是因为许多读者都能接触到PLD编程工具，这使得他们能够自行实现例子里的电路。为说明实际的设计问题，在本书中，我们使用了两种当今数字硬件产品中广泛应用的PLD芯片：一种称为复杂可编程逻辑器件（CPLD），另一种称为现场可编程门阵列（FPGA）。这两种芯片将在第3章中介绍。

为了加深对逻辑电路的理解并得到实践经验，我们建议读者利用CAD工具自己独立做一遍本书中的例子。大多数CAD系统公司向大学提供用于教学目的CAD工具，这类公司有：Altera、Cadence、Mentor Graphics、Synopsys、Synplicity和Xilinx。以上任何一家公司提供的CAD工具都与本书配合很好。对于尚无CAD工具的用户，本书附带光盘中包括Altera公司的Quartus II CAD系统。这个最新的系统支持设计流程内所有步骤，功能强大且易用。该软件易于在PC机上安装，而且我们也在附录B、C、D中提供了一系列完整的教程，描述了CAD工具的使用。

为课堂教学使用，某些PLD生产厂商提供了实验室开发用的印制电路板，包括了一个或多个PLD芯片以及与个人电脑的接口。一旦逻辑电路用CAD工具设计完毕，该电路便可以下载到印制电路板上的PLD里。然后可以通过简单的开关向PLD中施加输入信号，检查产生的输出。这些实验室电路板在PLD供应商的网页上都有介绍。

1.5 理论和实践

现代逻辑电路设计主要依靠CAD工具，但是逻辑设计这门学科早在CAD工具发明以前就已创立。历史记录清楚地表明，最初的计算机是用逻辑电路构造的，当时显然不会有可用的计算机来设计它们！

人们开发出了无数种手工设计技术来处理逻辑电路设计问题。我们将在第2章中介绍的布尔代数就是表示这种数字电路的数学工具。大量的“理论”被开发出来，向人们展示某些设计问题应该如何处理。设计者必须在实践中应用这些知识才能取得成功。

CAD工具不仅使设计者有可能设计出难以置信的复杂电路，而且使设计工作总体上大为简化。CAD工具能自动执行许多任务。也许有人会想，今天的设计工作者应该不需要理解

CAD工具在完成任务时使用的理论概念吧。于是问题浮上水面：为什么人们非得学习在设计工作中不再需要的理论呢？何不直接学习使用CAD工具呢？

有三个理由让我们来学习相关理论：首先，尽管可以通过CAD工具自动执行优化逻辑电路的任务以适应特定设计目标，但设计者仍然需要给出逻辑电路的最初描述。如果设计者描述的电路先天不足，那么最终产品的质量也好不了。其次，今天的CAD工具直接运用了数学理论和规则来设计和巧妙地处理逻辑电路。如果不掌握基础理论，CAD工具的用户就不可能了解工具究竟在做什么。最后，CAD工具提供了多种可选的处理步骤，供用户在设计过程中调用。设计者通过检查CAD工具产生的最终电路是否满足所需目标，来选择要使用的选项。而设计者决定在某种情况下该不该选用某选项的唯一途径，就是了解选中那个选项后CAD工具将在原来的基础上再做些什么，这就意味着设计者必须熟悉底层基础理论。没有深入理解基本概念，就当不了出色的逻辑电路设计师，因此我们在本书中全面地阐述了经典的逻辑电路理论。

最后要指出的一点是：就算不为了掌握CAD工具，学习逻辑电路理论还有一个好理由。简而言之，逻辑电路的学习是一种智力挑战，令人兴味盎然。虽说在充斥着精密自动化机械的摩登时代，使用工具替代人类思维颇为诱人；然而逻辑电路设计也如其他任何种类的设计过程一样，基于电脑的工具并不能代替人类的灵感和创新思维。只有在一位彻底了解逻辑电路本质的设计者手中，基于计算机的工具才能设计出高质量的数字电路。

参考文献

1. "International Technology Roadmap for Semiconductors," http://www.itrs.net
2. Altera Corporation, "Stratix III Field Programmable Gate Arrays," http://www.altera.com
3. Xilinx Corporation, "Virtex-5 Field Programmable Gate Arrays," http://www.xilinx.com
4. S. Brown, N. Manjikian, Z. Vranesic, S. Caranci, A. Grbic, R. Grindley, M. Gusat, K. Loveless, Z. Zilic, and S. Srbljic, "Experience in Designing a Large-Scale Multiprocessor Using Field-Programmable Devices and Advanced CAD Tools," 33rd IEEE Design Automation Conference, Las Vegas, June 1996.
5. A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic, "The Design and Implementation of the NUMAchine Multiprocessor," IEEE Design Automation Conference, San Francisco, June 1998.

第2章　逻辑电路入门

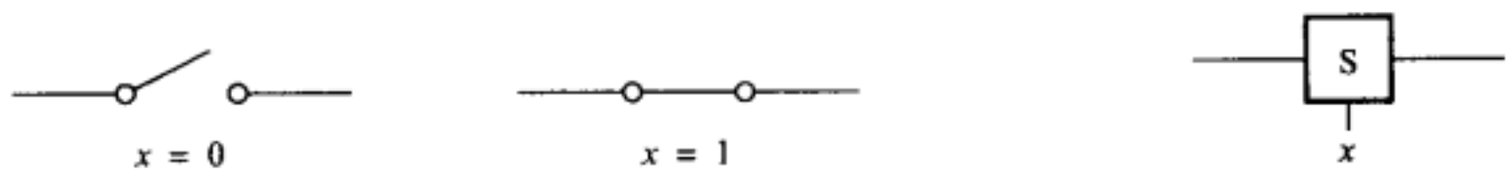
逻辑电路的研究发展主要是由它们在数字计算机中的应用所推动的。这种电路也构成了许多其他数字系统的基础，而这些系统的重点并不是进行数字的算术运算。例如，在许多控制应用中，只需要根据输入的信息做一些简单逻辑操作，而并不需要做大量的数值运算。

逻辑电路进行数字信号的操作，逻辑电路通常由电子电路实现，在电路中只用几个离散的值来表示信号。在二进制逻辑电路中只用两个值，0和1。在十进制的逻辑电路中用十个值，从0到9。因为人们很自然地用数字来表示每个信号值，所以这类逻辑电路也称为数字电路。与数字电路不同，还存在着另外一种模拟电路，在模拟电路中，信号的取值可在最大值和最小值之间连续地变化。

在这本书中，我们研究在数字技术中占统治地位的二进制电路。我们希望帮助读者理解这些电路是怎样进行工作的，怎样表示数学符号，怎样使用现代的自动设计技术进行设计。我们从介绍一些基本的与二进制逻辑电路有关的概念开始讲解。

2.1　变量和函数

二进制电路在数字系统应用中占统治地位的原因是由于其简易性，二进制电路中信号只有两种可能的取值。最简单的二进制元件是具有两个状态的开关。若某给定开关是由输入变量x控制的，则我们说当$x = 0$时开关断开，当$x = 1$时开关闭合，如图2-1a所示。在后面的示意图中我们使用图2-1b所示的图形符号来表示这些开关。请注意，在图形符号中，我们清晰地标出了控制输入变量x。在第3章中我们将会解释怎样用晶体管来实现这些开关。



a) 开关的两个状态　　b) 开关的符号

图2-1　二状态开关

考虑开关的一个简单的应用：小灯泡的开启和关闭。这一动作是由图2-2a所示的电路实现的。电路由一个电池供电，当有足够的电流通过灯丝（电阻）时，灯泡就会发光。当开关闭合（即$x = 1$）时，电路中有电流流动。在本例中，引起电路行为变化的输入变量是控制开关（断/合）的变量x。输出被定义为灯的状态，用L表示灯。若灯开时，则$L = 1$。若灯关时，则$L = 0$。根据这个约定，我们可以将灯L的状态描述为输入变量x的函数。当$x = 1$时，$L = 1$，当$x = 0$时，$L = 0$，则我们说：

$$L(x) = x$$

这个简单的逻辑表达式描述了电路的输出（为输入的函数）。我们说$L(x) = x$是个逻辑函数，而x是输入变量。

普通手电筒所使用的电路如图2-2a所示，其中开关是个简单的机械装置。在电子电路中，开关是由晶体管来实现的，灯则是由发光二极管（LED）担当。电子电路的能源由某给定电

压（可能为5V）的电源提供。电源的另一头接地，如图2-2b所示。接地可以使电流形成回路，为了使电路成为一个回路，可以将灯的另一条引线接地来实现。当然，也可以将灯连接到电源的负极来实现接地，如图2-2a所示。

a) 与电池的简单连接　　b) 用地线连接作为返回通路

图2-2　用开关控制的电灯泡

现在考虑用两个开关来控制灯泡状态的可能性，设x_1和x_2表示这两个开关的控制输入端。这两个开关可以串联或并联，如图2-3所示。当串联时，只有当两个开关同时闭合时，灯才会点亮。如果其中有一个开关断开，灯就会熄灭，这个行为可以用如下表达式表示：

$$L(x_1, x_2) = x_1 \cdot x_2$$

当$x_1=1$且$x_2=1$时，$L=1$；其他情况下，$L=0$

符号“·”称为与操作符（AND operator），图2-3a中的电路体现的就是这种逻辑与关系。

a) 逻辑与函数（串行连接）　　b) 逻辑或函数（并行连接）

图2-3　两种基本的函数

图2-3b所示的是两个开关的并联。在这种情况中，x_1或者x_2中只要有一个闭合，灯就会亮。两个开关都闭合，灯也会亮。只有当这两个开关同时断开时，灯才会熄灭。这个关系可以表示为：

$$L(x_1, x_2) = x_1 + x_2$$

当$x_1=1$或$x_2=1$或$x_1=x_2=1$时，$L=1$；当$x_1=x_2=0$时，$L=0$

符号“+”称为或操作符（OR operator），图2-3b中的电路实现了逻辑或功能。

在上面表示与（AND）和或（OR）关系的表达式中，输出$L(x_1, x_2)$是输入变量x_1和x_2的逻辑函数。与函数和或函数是两种最重要的逻辑函数，它们和其他简单的函数一起，可以作为构件块来实现各种逻辑电路。图2-4说明了怎样用三个开关以更复杂的方式控制一个灯。这个串并联开关连接实现了下面的逻辑表达式：

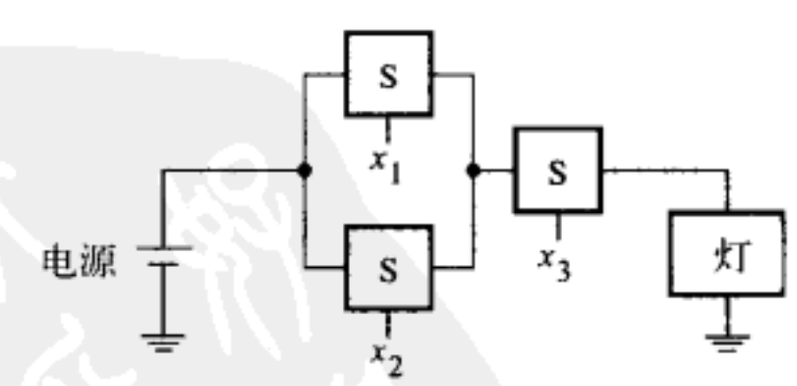


图2-4　串－并联连接

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

当$x_3=1$，同时x_1和x_2中至少有一个为1时，灯才能亮。

2.2 反相

到目前为止，我们已假设当开关闭合时，就会发生某些正面的事情，如电灯点亮。考虑当开关断开时，发生某些正面的事，也同样是有趣和有用的。假设我们将灯连接在如图2-5所示的电路中。在该例子中，开关与灯并联，而不是串联。产生的结果是，当开关闭合时，就把灯短路了，电流不再流过电灯。注意我们在电路中连接了另一个电阻，以防闭合的开关将电源短路。而当开关断开时，电灯反而点亮。这样的电路行为可以正式地用以下的逻辑表达式描述：

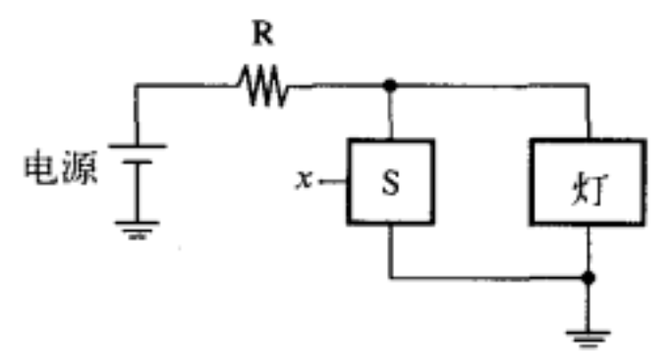


图2-5 反相电路

$$L(x) = \bar{x}$$

当$x=0$时，$L=1$；当$x=1$时，$L=0$

该函数的值正好与输入变量的值相反。通常我们不说求反，而称求补（complement）。所以在这个例子中，我们称函数$L(x)$为x的补。该操作另一个常见的名字是非操作。有几种记号可以用来表示求补。在前面的表达式中我们在x上面加了一个横线。这个记号从视觉的角度来说大概是最好的。然而，当需要由计算机键盘输入求补操作符时，就很不方便，而这件事却是CAD工具经常需要做的，所以使用变量上加横线的方法不可行。我们使用在变量后加一个撇号或者在变量前加惊叹号或是在变量前加波浪符号（~）来表示求补。下面几种表示是等价的：

$$\bar{x} = x' = !x = \sim x$$

求补操作可以加在单个变量前，也可以加在更复杂的操作上。例如，若

$$f(x_1, x_2) = x_1 + x_2$$

则f的补为

$$\bar{f}(x_1, x_2) = \overline{x_1 + x_2}$$

只有当x_1和x_2都不是1时，即$x_1 = x_2 = 0$时，这个表达式的值才为1。而且以下这几个表达式都是等价的：

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2)$$

2.3 真值表

我们已经介绍了三种最基本的逻辑操作：与、或、非，并把它们与简单的开关电路联系起来。这种办法给这些操作赋予了一定的物理意义。用真值表的形式也可以定义逻辑操作，如图2-6所示。表格的前两列（双竖线左侧）列出了x_1和x_2所有四种可能的逻辑值组合，接下来的一列定义了每种逻辑值组合的与运算，最后一列定义了每种逻辑值组合的或运算。因为我们要经常用到“逻辑值组合”，所以我们采用简短的术语取值（valuation）来表示逻辑值的组合。

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

图2-6 与、或操作的真值表

真值表有助于表达逻辑函数所包含的信息，在本书中我们将使用真值表来定义某些特殊的逻辑函数，并验证某些函

数关系的正确性。规模小的真值表很容易处理，然而随着变量的增多，真值表的规模会随着变量的增加呈现指数的增长，变得相当复杂。三个输入变量的真值表总共有8行，因为这些变量总共有8种取值。图2-7所示的就是这样一张真值表，其中定义了三输入的与、或函数。而四输入变量的真值表总共有16行，依此类推。一般来说，n输入变量的真值表总共有2^n行。

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

图2-7 三输入与、或操作的真值表

与、或操作可以扩展到n个变量。在输入变量为$x_1, x_2, \cdots, x_n$的与函数中，只有当这n个变量都是1时，函数值才能为1。在输入变量为$x_1, x_2, \cdots, x_n$的或函数中，至少有一个或多个变量是1时，逻辑函数的值才能为1。

2.4 逻辑门和逻辑网络

前面几节中介绍的三种基本逻辑操作可以用来实现任意复杂的逻辑函数。复杂的逻辑函数可能需要许多这样的基本操作才能实现。每个逻辑操作都能用晶体管来实现，实现逻辑操作的电路元件叫做*逻辑门*。逻辑门有一个或若干个输入，有一个输出，输出表示为输入的函数。用画电路图的方法来描述逻辑电路通常是很方便的，电路图由表示逻辑门的图形符号组成。表示与门、或门和非门的图形符号如图2-8所示。图中左边画的是只有几个输入时，逻辑门的表示方法；右边画的是有很多输入端时，逻辑门的表示方法。在第3章中，我们将介绍逻辑门是怎样用晶体管构成的。

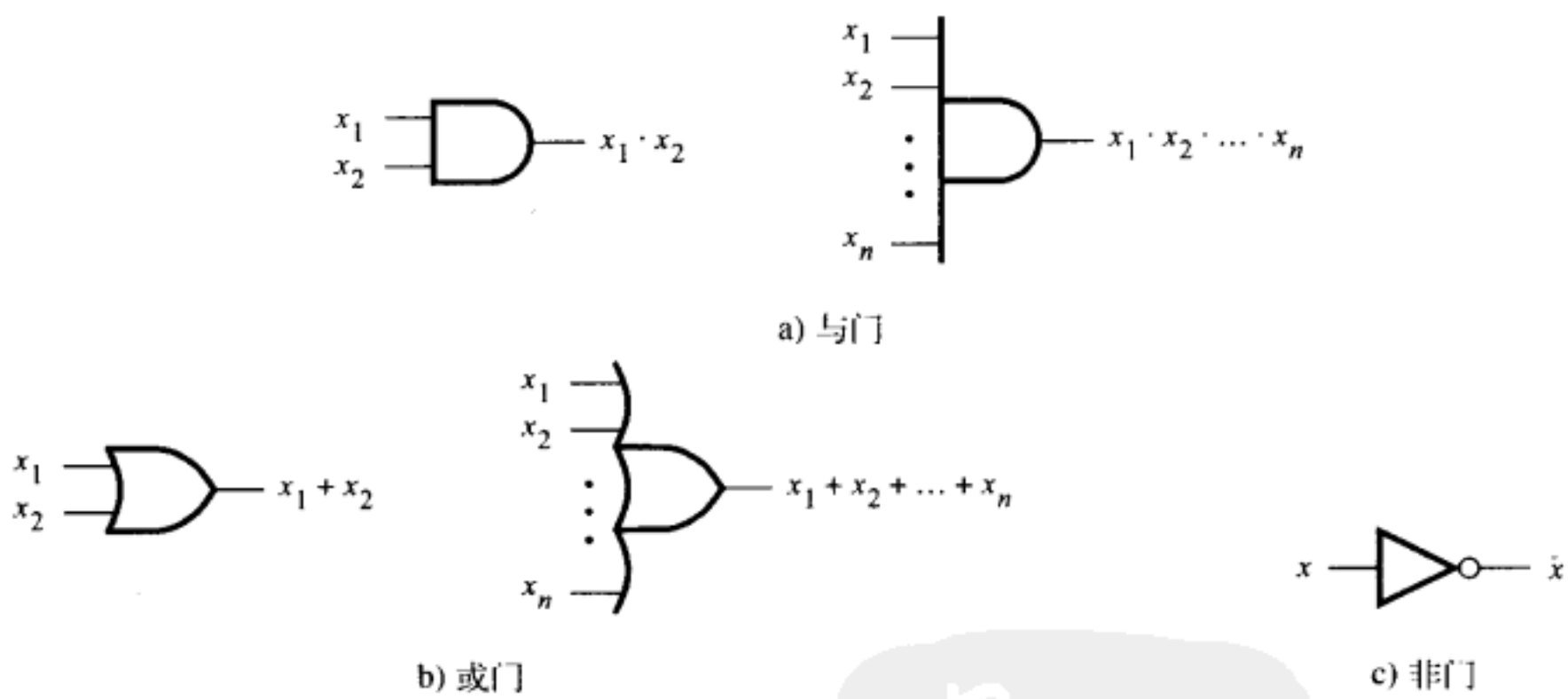


图2-8 基本门

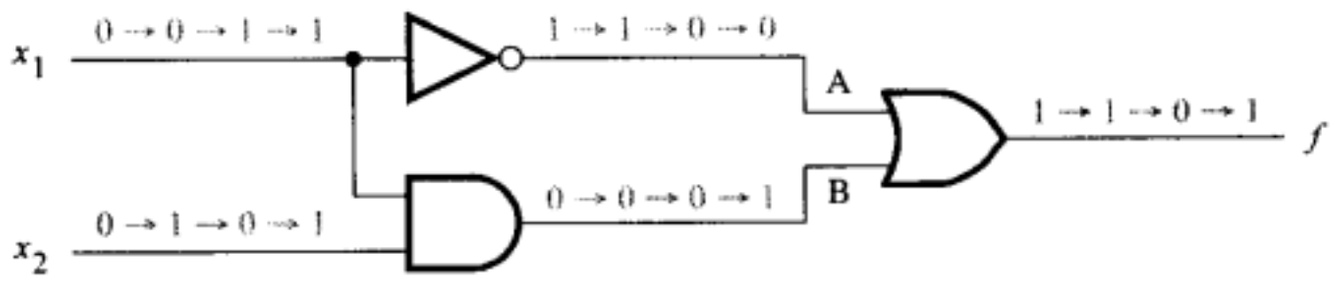
大规模电路由逻辑门组成的网络实现，例如图2-4所示的逻辑函数，可以由图2-9所示的网络来实现。给定网络的复杂程度直接影响电路的制造成本。因为我们总是想要降低制造产品的成本，所以找到成本尽可能低的逻辑电路实现方案就非常重要。后面我们很快将会看到，给定的逻辑函数可以由许多种不同的电路网络方案来实现。某些方案的电路网络比其他的简单些，因此必须仔细地寻找降低成本的方案。

图2-9 图2-4电路的函数

在技术术语中逻辑门网络通常称为逻辑网络，或简单地称为逻辑电路。我们常混用这些术语。

逻辑网络的分析

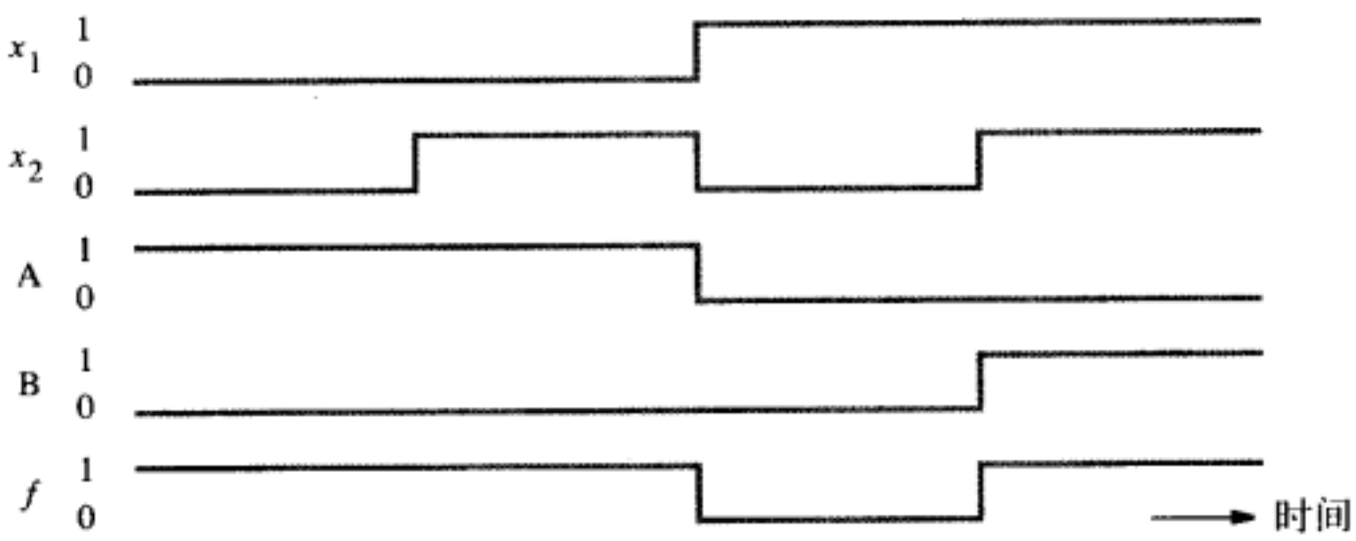
数字系统的设计者会遇到两个基本问题。第一个问题是：对已存在的逻辑网络，必须有办法确认所实现的逻辑功能，这项任务称为分析（analysis）过程。第二个问题是：设计一个新的网络，使该网络实现所要求的逻辑功能行为，这项任务称为综合（synthesis）过程。分析过程相当直观易懂，远比综合过程简单。



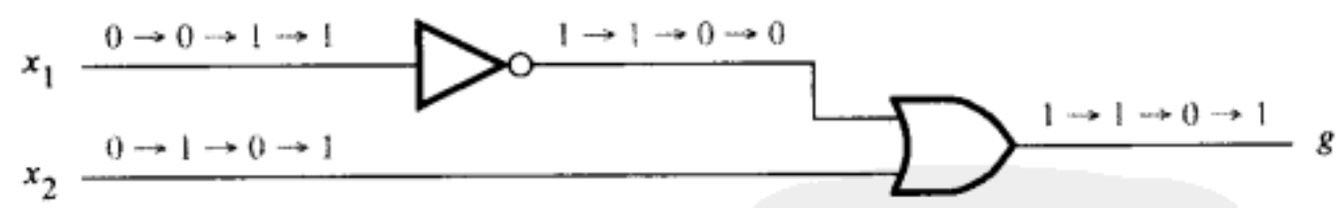
a) 实现 $f = \overline{x}_1 + x_1 \cdot x_2$ 的网络

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

b) f 的真值表



c) 时序图



d) 实现 $g = \overline{x}_1 + x_2$ 的网络

图2-10 逻辑网络举例

图2-10a展示了由3个逻辑门组成的简单的逻辑网络。要分析它的功能，我们要考虑当输入所有可能的输入信号取值时，输出是什么。假设输入为 $x_1 = x_2 = 0$，这使与门输出为0，非门输出为1。因为或门的一个输入是1，所以或门输出是1。所以，令 $x_1 = x_2 = 0$，则 $f = 1$。令 $x_1=0$，$x_2=1$，则 $f = 1$ 输出不变，因为与门和非门的输出仍然分别是1和0。接着，令 $x_1=1$，$x_2=0$，这时非门的输出变为0，同时与门的输出仍然是0。或门的两个输入都是0，则 $f = 0$。最后，令 $x_1 = x_2 = 1$，与门的输出变成1，则 $f=1$。用语言所做的解释，可以用图2-10b所示的真值表来表示。

时序图

通过考虑输入x_1和x_2的四种可能取值之后，我们已经确定了图2-10a所示网络的行为。假设对应这些输入变量取值的信号按照我们讨论的顺序逐一施加到网络的输入端：即（x_1，x_2）=（0，0），（0，1），（1，0），（1，1）。网络中每点信号的变化见图2-10a中的表示，同样的信息也可以用图的形式表示，这种图就称为时序图，如图2-10c所示。时间从左到右变化，每个输入取值都持续一段时间。图形表示了电路的输入和输出的波形，还有标记为A点和B点的两个内部信号的波形。

如图2-10c所示的时序图显示了位于A点、B点和输出点f的波形改变正好发生在x_1和x_2发生改变的同一时刻。由于我们假设逻辑门对于其输入的响应时间为0，所以才有这样理想的波形。这种时序图只可以用来表明逻辑电路的功能行为，而并不反映实际情况。因为实际逻辑门是用电子电路实现的，所以需要一些时间才能改变其状态。因此，在输入值发生改变到门的输出值发生相应的变化之间必定存在一个延迟。在后面的几章中，我们将使用包含了这种延迟的时序图。

时序图有多种用途。时序图用波形的方式描述了逻辑电路的行为。当电路用诸如逻辑分析仪器和示波器测试时，能观察到这种形式的波形。CAD工具可以生成这些波形，在电路具体实现之前，波形可以告诉设计者给定的逻辑电路将如何运行。在本章的后面将介绍CAD工具，CAD工具的使用将贯穿全书。

功能等价的网络

现在让我们考虑一下图2-10d所示的网络，用同样的分析步骤，我们可以发现输出g和图2-10a中的输出f完全相同。因此，$g(x_1, x_2) = f(x_1, x_2)$，也就是说这两个电路等价；它们的输出都可以用图2-10b中的真值表表示。既然这两个网络实现相同的功能，那么选用简单的那一个就很有意义了，因为简单的网络实现成本低。

一般情况下，一个逻辑函数可以用许多种不同的网络来实现，实现电路的成本也可能有所不同。这就产生了一个重要的问题，怎样为给定的逻辑函数寻找最优的实现电路呢？有许多逻辑函数的综合技术可用来生成电路。我们将在第4章中重点讨论这个问题。从目前情况看，我们应该注意到有必要进行一些处理，将如图2-10a所示的复杂网络转化为如图2-10d所示的网络。既然$f(x_1, x_2)=\bar{x}_1+x_1\cdot x_2$，$g(x_1, x_2)=\bar{x}_1+x_2$，则必然存在某些规律使得下面的等式成立：

$$\bar{x}_1+x_1\cdot x_2=\bar{x}_1+x_2$$

我们前面用详细的分析方法和填写真值表的方法得到这两个电路等价的结论，由此，上面的等式成立。下一节中，我们将讨论逻辑函数的数学处理方法，该方法为现代电路设计奠定了技术基础。

2.5 布尔代数

1849年乔治·布尔发表了一篇论文，论文涉及逻辑思想和推理过程的代数描述方法[1]。该方法及其以后进一步的完善形成了著名的布尔代数。等了几乎100年，布尔代数才被发现在工程上有应用价值。20世纪30年代后期，克劳德·香农发现了布尔代数是描述开关电路的有效方法[2]。因此这种代数被用于描述逻辑电路。我们将会展示，布尔代数是设计和分析逻辑电路的功能强大的工具。读者也将意识到布尔代数为现代数字技术奠定了基础，从而感谢乔治·布尔的贡献。

布尔代数公理

布尔代数和其他代数一样，由很少几个基本假设推导得到一套规则，布尔代数就建立在这套规则的基础上。这几个基本假设就称为公理。我们假设二进制布尔代数只包含取两个值（0和1）的元素。假设以下公理正确：

1a. $0 \cdot 0 = 0$

1b. $1 + 1 = 1$

2a. $1 \cdot 1 = 1$

2b. $0 + 0 = 0$

3a. $0 \cdot 1 = 1 \cdot 0 = 0$

3b. $1 + 0 = 0 + 1 = 1$

4a. 若 $x = 0$，则 $\bar{x} = 1$

4b. 若 $x = 1$，则 $\bar{x} = 0$

单变量定理

根据公理我们可以定义一些规则来处理单个变量。这些规则通常称为定理。若x是二进制变量，则有以下定理存在：

5a. $x \cdot 0 = 0$

5b. $x + 1 = 1$

6a. $x \cdot 1 = x$

6b. $x + 0 = x$

7a. $x \cdot x = x$

7b. $x + x = x$

8a. $x \cdot \bar{x} = 0$

8b. $x + \bar{x} = 1$

9. $\bar{\bar{x}} = x$

用完全归纳法（perfect induction）很容易证明这些定理的正确性，即将$x = 1$和$x = 0$代入表达式，运用上面介绍的公理进行推导。例如在定理5a中，若$x = 0$，则有$0 \cdot 0 = 0$，根据公理1a这是正确的。同样，若$x = 1$，则由定理 5a 得到$1 \cdot 0 = 0$，根据公理3a，这也是正确的。读者可以用这种方法验证定理5a到9。

对偶性

注意，前面我们将公理和定理成对列出。这体现了非常重要的对偶规则。对于给定的一个逻辑表达式，将所有的“+”操作符替换为“·”，并将所有的“·”操作符替换为“+”；将所有的0替换为1，并将所有的1替换为0，就得到对偶表达式。布尔代数中任何一个定理或公理的对偶表达式都是正确的。在讨论的这一时刻，读者也许并不理解对偶性为什么是一个有用的概念，然而到了本章的后面部分，这个概念就会变得非常清楚，那时，我们就会知道：对偶性意味着每个逻辑函数至少可以用两种不同的布尔代数表达式表示。其中一种表达式比另一种表达式的物理实现更简单，因而更被认可。

二变量和三变量性质

为了能够处理多个变量，定义2个变量和3个变量的代数恒等式是很有用的。对于每个恒等式，给出了其对偶的形式。这些恒等式通常被称为性质。若x, y, z是二进制的变量，则下列性质成立：

10a. $x \cdot y = y \cdot x$ 交换律

10b. $x + y = y + x$

11a. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ 结合律

11b. $x + (y + z) = (x + y) + z$

12a. $x \cdot (y + z) = x \cdot y + x \cdot z$ 分配律

12b. $x + y \cdot z = (x + y) \cdot (x + z)$

13a. $x + x \cdot y = x$ 吸收律

13b. $x \cdot (x + y) = x$

14a. $x \cdot y + x \cdot \bar{y} = x$ 合并律

14b. $(x + y) \cdot (x + \bar{y}) = x$

15a. $\overline{x \cdot y} = \bar{x} + \bar{y}$ 德摩根（DeMorgan）定律

15b. $\overline{x + y} = \bar{x} \cdot \bar{y}$

16a. $x + \bar{x} \cdot y = x + y$

16b. $x \cdot (\bar{x} + y) = x \cdot y$

17a. $x \cdot y + y \cdot z + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z$ 合意

17b. $(x + y) \cdot (y + z) \cdot (\bar{x} + z) = (x + y) \cdot (\bar{x} + z)$

我们可以再次用完全归纳法或用代数变换证明这些公式的正确性。图2-11表示的是怎样用完全归纳法来证明德摩根定律，这里使用了真值表的形式，性质15a中左边和右边的取值也得出了相同的结果。

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\bar{x}$	$\bar{y}$	$\bar{x} + \bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

图2-11 15a德摩根定律的证明

我们已经列出了一些公理、定理和性质。定义布尔代数并不需要所列出的全部规则，例如，假设定义了“+”和“·”操作，包含定理5和8以及性质10和12就足够了。有时把这些最核心的规则称为亨廷顿（Huntington）基本法则[3]。其他等式都可以从这些基本法则中推出。

前面的公理、定理和性质提供了对更复杂逻辑表达式进行代数变换的必要信息。

例2.1 让我们来证明下面逻辑等式的正确性。

$$(x_1 + x_3) \cdot (\bar{x}_1 + \bar{x}_3) = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

左边可以做如下变换，用分配律12a，得到

$$\text{左式} = (x_1 + x_3) \cdot \bar{x}_1 + (x_1 + x_3) \cdot \bar{x}_3$$

再次应用分配律，得到

$$\text{左式} = x_1 \cdot \bar{x}_1 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + x_3 \cdot \bar{x}_3$$

请注意，分配律允许括号中的各项使用与普通代数类似的方式来处理与操作项。然后通过定理8a，$x_1 \cdot \overline{x_1}$ 和 $x_3 \cdot \overline{x_3}$ 两项都等于0，因此得到

$$\text{左式} = 0 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + 0$$

根据6b，得到

$$\text{左式} = x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3$$

最后，应用交换律10a和10b，上式变为

$$\text{左式} = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

和原式的右式相等。 ■

例2.2 考虑逻辑等式：

$$x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 = \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

左式可以做如下化简：

$$\begin{aligned}
\text{左式} &= x_1 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3 && \text{用10b} \\
&= x_1 \cdot (\bar{x}_3 + x_3) + \bar{x}_2 \cdot (\bar{x}_3 + x_3) && \text{用12a} \\
&= x_1 \cdot 1 + \bar{x}_2 \cdot 1 && \text{用8b} \\
&= x_1 + \bar{x}_2 && \text{用6a}
\end{aligned}$$

右式可以做如下化简：

$$\begin{aligned}
\text{右式} &= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot (x_2 + \bar{x}_2) && \text{用12a} \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot 1 && \text{用8b} \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 && \text{用6a} \\
&= x_1 + \bar{x}_1 \cdot \bar{x}_2 && \text{用10b} \\
&= x_1 + \bar{x}_2 && \text{用16a}
\end{aligned}$$

原式两边化简后的表达式子完全相同，证明了逻辑等式的正确性。请注意，同一个逻辑函数既可以用等式左边的表达式表示，也可以用等式右边的表达式表示，即

$$\begin{aligned}
f(x_1, x_2, x_3) &= x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2
\end{aligned}$$

化简使我们得到了比原来简单得多的表达形式：

$$f(x_1, x_2, x_3) = x_1 + \bar{x}_2$$

上面的表达式也表示同样的逻辑关系。使用这个更简单的表达式，可以制造出也能实现该逻辑功能而成本更低的电路。■

例2.1和例2.2说明了在代数化简处理机制中应用公理、定理和性质的目的。我们通过这些简单的例子可以看到，用这种办法来处理非常复杂的表达式是不实际的，即使如此，这些定理和性质为计算机辅助设计工具提供了逻辑函数自动综合的基础。想要理解使用这些工具能做出什么，设计者必须掌握这些基本概念。

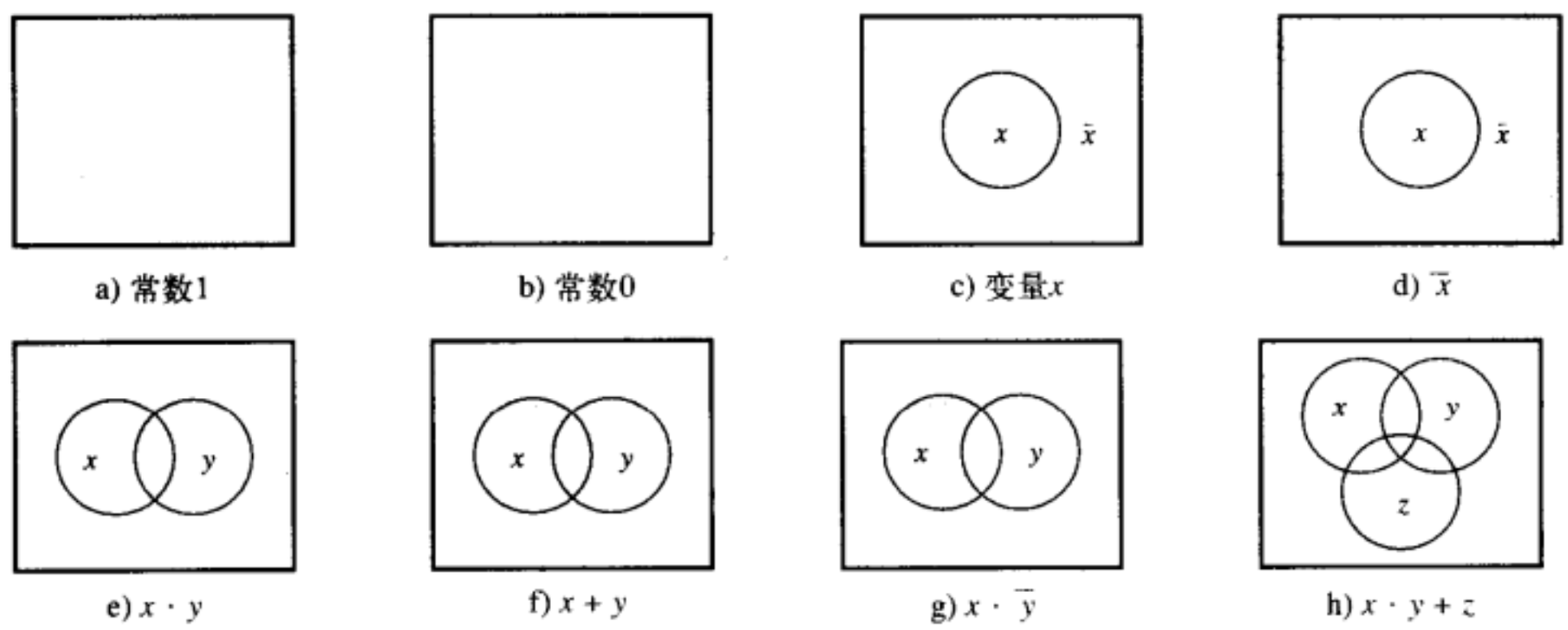
2.5.1 维恩图

我们已经说过，完全归纳法可以用于定理和性质的验证。这一过程是乏味的，从概念的观点而言，也提供不了什么信息。有一种简单的从视觉上就可直观地证明定理和性质的方法，这种方法称为维恩图。读者会发现，维恩图可以提供两个表达式为什么相等的更直观的理解。

在传统上维恩图一直用于数学，可以对集合代数的各种操作与关系提供图示的解释。集合s是由许多元素组成的集合体，其中的每个元素都被称为s的成员。在维恩图中，集合的元素用轮廓线包围的区域图形表示，诸如用矩形、圆形或椭圆形等。例如，在从1到10的整数全域N集合中，N中的偶数集合为：$E = \{2, 4, 6, 8, 10\}$，表示E的轮廓线把这些偶数都圈在内部。而N集合中的所有奇数形成了E的补集，因此轮廓线以外的区域表示的是：$\bar{E} = \{1, 3, 5, 7, 9\}$。

既然在布尔代数的全域B中只有两个取值：$B = \{0, 1\}$，因此我们说，轮廓线包围的区域内对应于集合s，用$s = 1$表示，轮廓线包围的区域以外的部分用$s = 0$表示。在维恩图中，我们将把$s = 1$的区域画上阴影。维恩图的概念如图2-12所示。全域B用矩形表示，常数1和0分别用图

2-12a和图2-12b表示。变量x用圆形表示，圆内的区域对应于$x = 1$，圆外的区域对应于$x = 0$，见图2-12c所示。包含1个或多个变量的表达式取值为1的部分用阴影表示。图2-12d指出如何表示 x的补集。



a) 常数1　b) 常数0　c) 变量x　d) $\overline{x}$

e) $x \cdot y$　f) $x + y$　g) $x \cdot \overline{y}$　h) $x \cdot y + z$

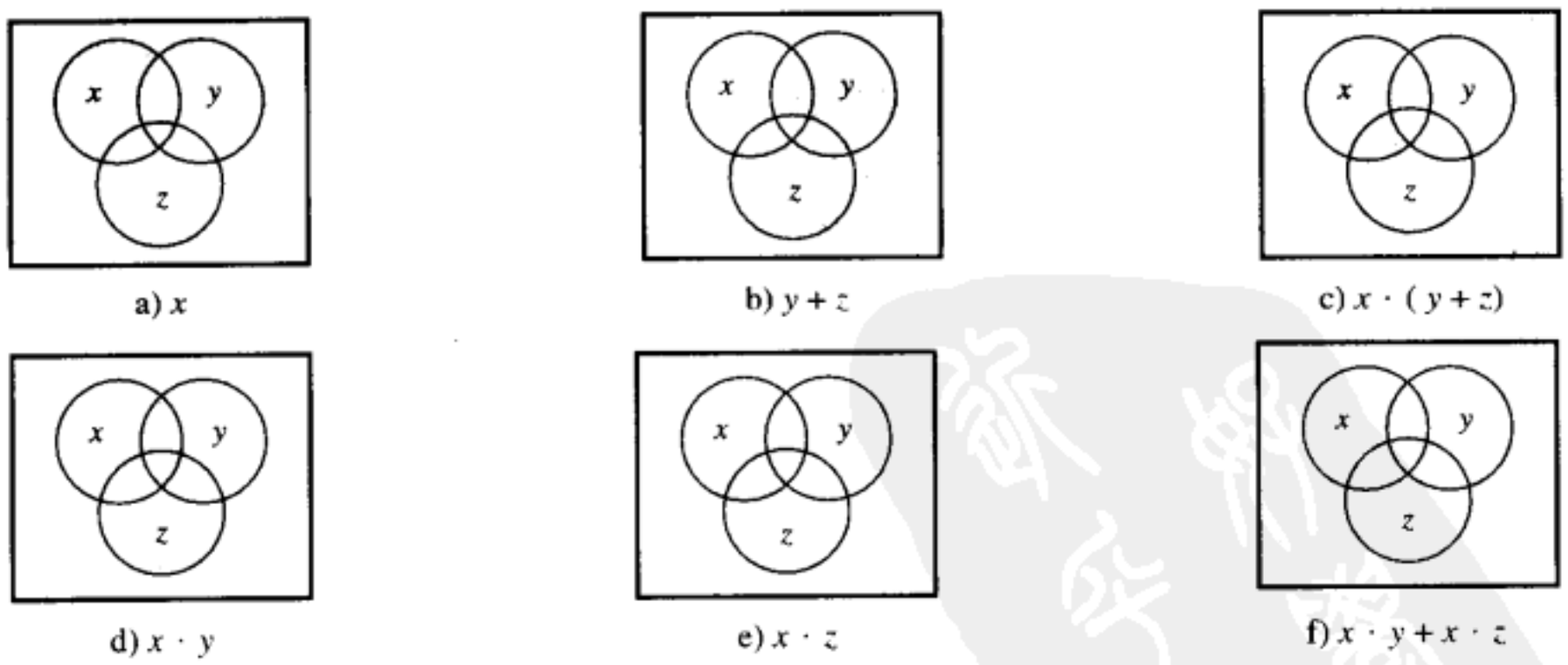
图2-12　维恩图表示法

为了表示两个变量x和 y，我们画两个互相交叠的圆。两个圆交叠的区域表示$x = y = 1$的情形，换言之，表示的是x和 y的与，见图2-12e。因为该公共区域由x与 y 的交叠部分组成，所以在正式场合，与操作通常称为x与 y的交集。图2-12f所示的是或操作，$x + y$ 表示的是两个圆包括的全部域，即x或 y至少有一个为1。因为合并了两个圆包括的域，所以在正式场合，或操作通常称为x和 y 的并集。

图2-12g说明的是乘积项$x \cdot \overline{y}$，在图中用x和$\overline{y}$的交叠区域来表示。图2-12h给出了一个3变量的例子，在图中表达式$x \cdot y + z$是由x和 y 的交叠区和z区合并在一起的区域所表示的。

为了理解怎样用维恩图来验证两个表达式等价，让我们举例验证2.5节中介绍的分配律12a。图2-13给出了定义该性质的左式与右式相等的图形结构。

$$x \cdot (y + z) = x \cdot y + x \cdot z$$



a) x　b) $y + z$　c) $x \cdot (y + z)$

d) $x \cdot y$　e) $x \cdot z$　f) $x \cdot y + x \cdot z$

图2-13　验证分配律：$x \cdot (y + z) = x \cdot y + x \cdot z$

图2-13a阴影区域表示的是$x = 1$。图2-13b阴影区域表示的是 $y + z$，图2-13c阴影区域表示的是$x \cdot (y + z)$，它由图2-13a和图2-13b的阴影区域交叠得到。等式右边由图2-13的d、e、f三部分组成。图2-13d和e描述的是$x \cdot y$和$x \cdot z$，它们的并集对应图2-13f的阴影。由于图2-13c中

阴影区域和图2-13f中阴影区域相同，因此分配律是正确的。

再举另外一个例子，考虑下面的等式：

$$x\cdot y+\bar{x}\cdot z+y\cdot z=x\cdot y+\bar{x}\cdot z$$

验证过程见图2-14。请注意等式中$y\cdot z$项完全可以被$x\cdot y$和$\bar{x}\cdot z$两项覆盖，因而可以省略。

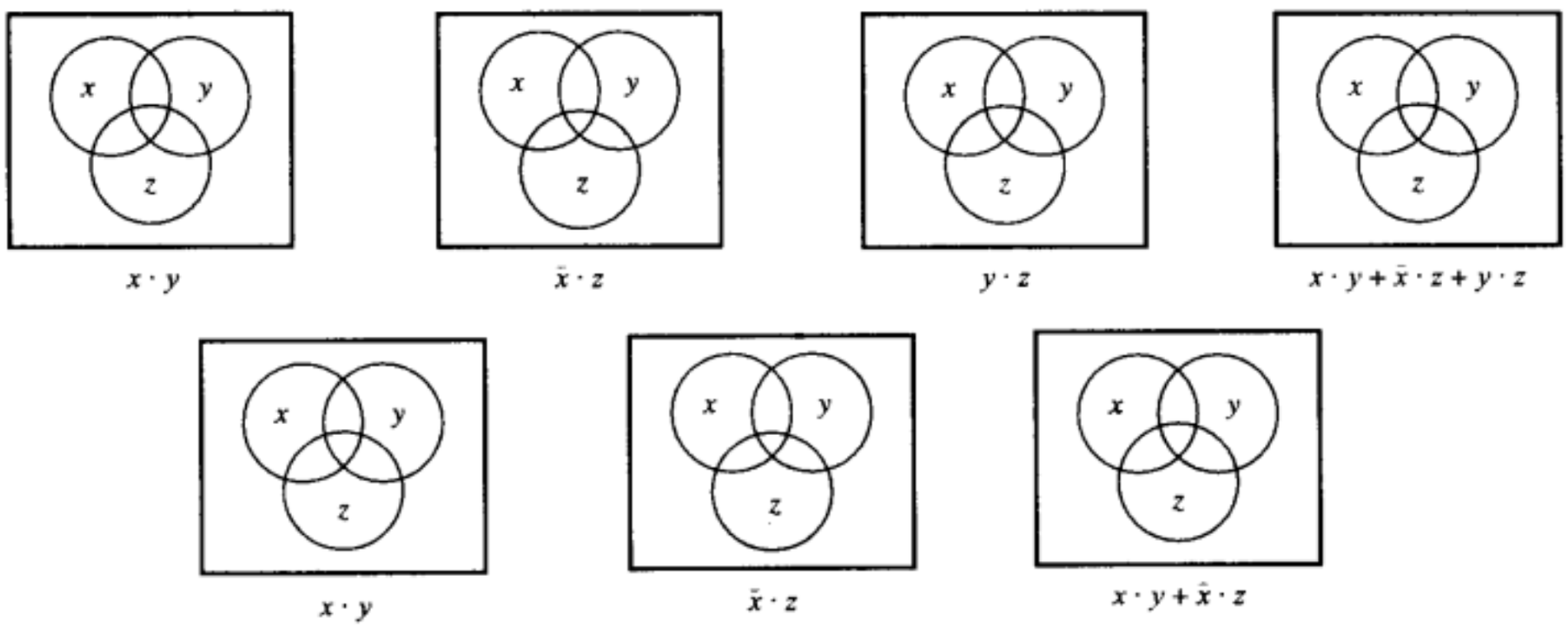


图2-14 验证 $x\cdot y+\bar{x}\cdot z+y\cdot z=x\cdot y+\bar{x}\cdot z$

读者应该使用维恩图证明其他等式。用这种方法证明德摩根定律特别有启发。

2.5.2 符号和术语

布尔函数是建立在“与”（AND）和“或”（OR）操作的基础之上的。我们用“ · ”和“+ ”来代表这两种操作运算。这两个符号也是人们熟悉的表示算术乘法运算和加法运算的标准符号。考虑到布尔代数和算术运算的相似性，我们就能明白采用相同的符号的主要原因。实际上，就单个数字而言，算术运算和布尔代数运算只有一点有明显的不同：在普通的算术运算中1+1=2，而在布尔代数中，由2.5节中的定理7b可以知道1+1=1。

在处理数字电路问题时，在大多数场合，“+”明显地表示或操作。然而，当逻辑电路的设计包含算术运算时，有可能出现“+”符号使用的混乱。为了避免出现这种混淆的情况，我们可以用另外一套符号来表示与操作和或操作。用∧符号表示与操作，用∨表示或操作，是相当常见的方法。所以可以将$x_1\cdot x_2$写成$x_1\wedge x_2$，将x_1+x_2写成$x_1\vee x_2$。

由于算术运算和逻辑运算的相似性，“或”和“与”运算经常被叫做“逻辑和”和“逻辑积”运算。所以x_1+x_2是x_1和x_2的逻辑和，$x_1\cdot x_2$是x_1和x_2的逻辑积。习惯上也通常把“逻辑和”以及“逻辑积”简称为“和”与“积”。因此表达式

$$x_1\cdot\bar{x}_2\cdot x_3+\bar{x}_1\cdot x_4+x_2\cdot x_3\cdot\bar{x}_4$$

叫做三个积的和，而下面的表达式

$$(\bar{x}_1+x_3)\cdot(x_1+\bar{x}_3)\cdot(\bar{x}_2+x_3+x_4)$$

叫做三个和的积。

2.5.3 运算的优先级别

用与、或、非三种基本操作，可以构造无限多个逻辑表达式。括号经常被用来规定运算执行的顺序，然而为了避免过多地使用括号，定义了基本运算的优先级别。换言之，在不加

括号的情况下，逻辑表达式中的运算按以下顺序执行：非，与，最后是或。所以在以下表达式中：

$$x_1 \cdot x_2 + \bar{x}_1 \cdot \bar{x}_2$$

首先应该计算$\bar{x}_1$和$\bar{x}_2$，其次才是$x_1 \cdot x_2$，最后才是它们的或运算。如果没有观察这个优先级的顺序，我们将不得不使用括号来达到同样的效果，见下面用括号表示的操作执行顺序：

$$(x_1 \cdot x_2) + ((\bar{x}_1) \cdot (\bar{x}_2))$$

最后，为了进一步简化逻辑表达式的形式，在不会产生混淆的前提下，一般省略与操作符“·”，因此前面的表达式可以写成：

$$x_1 x_2 + \bar{x}_1 \bar{x}_2$$

本书中将使用这种风格来表示逻辑表达式。

2.6 用与门、或门和非门进行综合

掌握了一些基本概念后，现在我们就能试着利用与门、或门和非门来实现任意的逻辑功能。假设我们想设计具有双输入的逻辑电路，输入信号分别为：x_1和x_2。假定x_1，x_2代表两个开关的状态，每个开关都可以为断开（0）或者闭合（1）两种状态中的某一种状态。电路的功能是连续不断地监控开关的状态，并当开关（x_1, x_2）处于（0, 0）、（0, 1）或（1, 1）时，电路的输出为逻辑值1；当开关状态为（1, 0）时，则电路的输出为0。换言之，电路要求的行为是：若开关x_1闭合且x_2断开时，则电路输出为0，否则输出为1。我们可以用真值表的形式表达电路要求的行为，如图2-15所示。

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

图2-15 待综合的函数

设计实现真值表指定功能的逻辑电路的方法之一是：为每个使函数 f 值为1的输入取值创建值为1的乘积项。然后，再对这些乘积项求逻辑和，就可以实现函数 f。让我们先从真值表的第四行看起，此行对应于 $x_1 = x_2 = 1$。与该输入取值对应的输出为1的乘积项是 $x_1 \cdot x_2$，相当于将x_1和x_2进行逻辑与。接下来考虑真值表的第一行，此行对应于$x_1 = x_2 = 0$。与该输入取值对应的输出为1的乘积项是 $\bar{x}_1 \bar{x}_2$。同理，第二行的乘积项为 $\bar{x}_1 \cdot x_2$。从而，可得到 f 为：

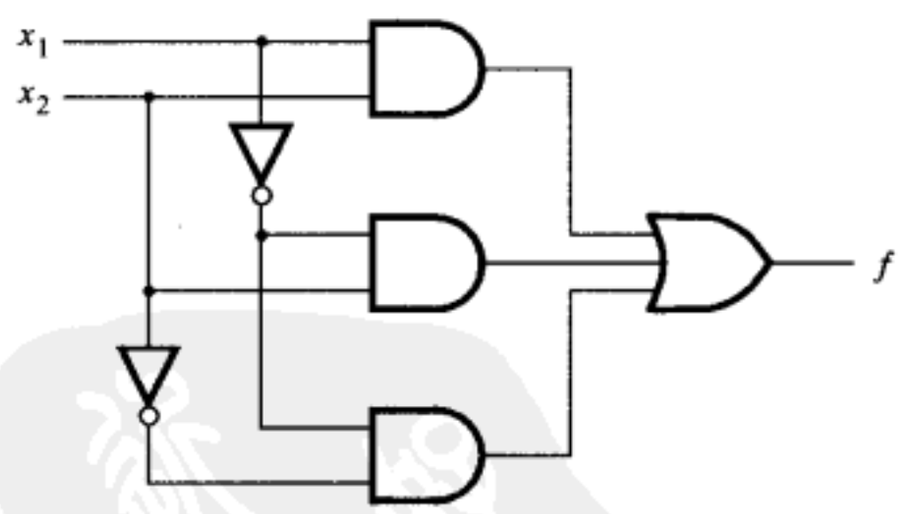
$$f(x_1, x_2) = x_1 x_2 + \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2$$

对应于该逻辑表达式的电路原理图如图2-16a所示。

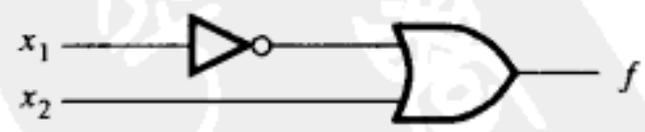
尽管这个电路原理图正确地实现了f，但并非是最简单的电路。为了找到一个更简单的电路，我们可以利用2.5节提供的定理和性质来化简已得到的表达式。根据定理7b，我们可以复制逻辑和表达式中的任一项。复制表达式中的第三项，上式就成为：

$$f(x_1, x_2) = x_1 x_2 + \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2 + \bar{x}_1 x_2$$

根据交换律10b，将第2和第3乘积项的位置



a) 正则积之和的电路



b) 成本最低的电路

图2-16 图2-15函数的两种实现

互换，则得到：

$$f(x_1,\ x_2)=x_1x_2+\bar{x}_1x_2+\bar{x}_1\bar{x}_2+\bar{x}_1x_2$$

由分配律12a，可得到：

$$f(x_1,\ x_2)=(x_1+\bar{x}_1)x_2+\bar{x}_1(\bar{x}_2+x_2)$$

再根据定理8b，可得：

$$f(x_1,\ x_2)=1\cdot x_2+\bar{x}_1\cdot 1$$

最后，由定理6a，得到：

$$f(x_1,\ x_2)=x_2+\bar{x}_1$$

由该表达式描述的电路如图2-16b所示。很明显，该电路的成本远低于图2-16a所示的电路。

这个简单的例子说明了两个问题。首先，利用对应于真值表中使每个函数值等于1的乘积项（与门）直接来实现逻辑函数。每个乘积项包含所有的输入变量，其写法如下：若输入变量x_i在行中为1时，则记为x_i；若$x_i=0$，则记为$\bar{x}_i$。这些乘积项的逻辑和就是所求的函数。其次，实现给定的逻辑函数存在很多种不同的电路方案。有些方案比其他的方案简单。可用代数处理的方法化简逻辑表达式，从而得到成本较低的电路实现方案。

从描述期望电路的功能行为着手，然后形成一个能实现该行为的电路网络，我们将这样一个过程称之为综合。因此，我们可以说由图2-15所示的真值表“综合”成如图2-16所示的电路网络。根据真值表生成与-或表达式只是许多种综合技术中的一种，本书将在后面的章节介绍这些综合技术。

积之和形式以及和之积形式

借助于非常简单的例子介绍了综合过程后，我们现在将用更正式的术语，即用在技术文献中会遇到的术语，更深入地介绍综合。我们也将展示如何将在2.5节中已介绍过的对偶性原理更广泛地应用到综合过程中。

若函数 f 已用真值表的形式确定，则实现 f 的表达式便可以通过考虑真值表中使 $f=1$的那些行得到，这在前面已经讲过，或者也可以通过考虑真值表中使 $f=0$的那些行得到，我们下面马上就讲解。

最小项

对于有n个变量的函数来说，若有一个乘积项，其中n个变量的每一个都出现过一次，则称该乘积项为最小项（minterm）。这些变量或以原变量或以反变量的形式出现在最小项中。对给定真值表的某一行而言，若$x_i=1$，则最小项中包含x_i，若$x_i=0$，则最小项中包含$\bar{x}_i$。

为了阐明这一概念，请考虑如图2-17所示的真值表。为了便于说明，对真值表的各行进行了编号（0到7）。（熟悉二进制数表示法的读者知道，所选择的行号就是变量x_1、x_2和x_3的位代码表示的数，我们将在第5章中讨论数的表示法。）图2-17列出了三变量表中所有的最小项。例如，第一行中变量的值为 $x_1=x_2=x_3=0$，对应的最小项为 $\bar{x}_1\bar{x}_2\bar{x}_3$。第二行中 $x_1=x_2=0$，$x_3=1$，对应的最小项为 $\bar{x}_1\bar{x}_2x_3$，依次类推。为了能够更方便地查阅每个最小项，用图中相应的行号作为索引来标识每个最小项。我们用符号m_i来表示第 i 行的最小项。这样就有$m_0=\bar{x}_1\bar{x}_2\bar{x}_3$，$m_1=\bar{x}_1\bar{x}_2x_3$，依次类推。

积之和形式

函数 f 可以用最小项之和的表达式描述，其中每个最小项是输入变量的相应取值和函数 f

值的逻辑与。举例来说，二变量最小项有$m_0=\bar{x}_1\bar{x}_2$，$m_1=\bar{x}_1x_2$，$m_2=x_1\bar{x}_2$和$m_3=x_1x_2$。如图2-15所示的函数可以被描述为：

$$\begin{aligned} f &= m_0\cdot 1+m_1\cdot 1+m_2\cdot 0+m_3\cdot 1 \\ &= m_0+m_1+m_3 \\ &= \bar{x}_1\bar{x}_2+\bar{x}_1x_2+x_1x_2 \end{aligned}$$

行号	x_1	x_2	x_3	最小项	最大项
0	0	0	0	$m_0=\bar{x}_1\bar{x}_2\bar{x}_3$	$M_0=x_1+x_2+x_3$
1	0	0	1	$m_1=\bar{x}_1\bar{x}_2x_3$	$M_1=x_1+x_2+\bar{x}_3$
2	0	1	0	$m_2=\bar{x}_1x_2\bar{x}_3$	$M_2=x_1+\bar{x}_2+x_3$
3	0	1	1	$m_3=\bar{x}_1x_2x_3$	$M_3=x_1+\bar{x}_2+\bar{x}_3$
4	1	0	0	$m_4=x_1\bar{x}_2\bar{x}_3$	$M_4=\bar{x}_1+x_2+x_3$
5	1	0	1	$m_5=x_1\bar{x}_2x_3$	$M_5=\bar{x}_1+x_2+\bar{x}_3$
6	1	1	0	$m_6=x_1x_2\bar{x}_3$	$M_6=\bar{x}_1+\bar{x}_2+x_3$
7	1	1	1	$m_7=x_1x_2x_3$	$M_7=\bar{x}_1+\bar{x}_2+\bar{x}_3$

图2-17 三变量的最小项和最大项

这一形式与上一节用直观的方法得到的形式相同。结果表达式中只包含使 $f=1$对应行的最小项。

任何函数 f 都可以用真值表中使 $f=1$ 的各行所对应的最小项的逻辑和来表示。这样实现的函数在功能上是正确的，也是唯一的，但并非一定是函数 f 的最低成本的实现。若逻辑表达式由乘积项（与）的逻辑和（或）组成，则该逻辑表达式被称为积之和（SOP）形式。若每个乘积项都为最小项，则该式被称为函数 f 的正则积之和表达式。正如我们在图2-16的例子中所见到的，综合过程中的第一步是由给定逻辑函数得到正则积之和表达式。然后再利用2.5节的定理和性质对该表达式进行进一步的处理，其目的是找到功能等价但是成本较低的积之和表达式。

行号	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

图2-18 三变量函数

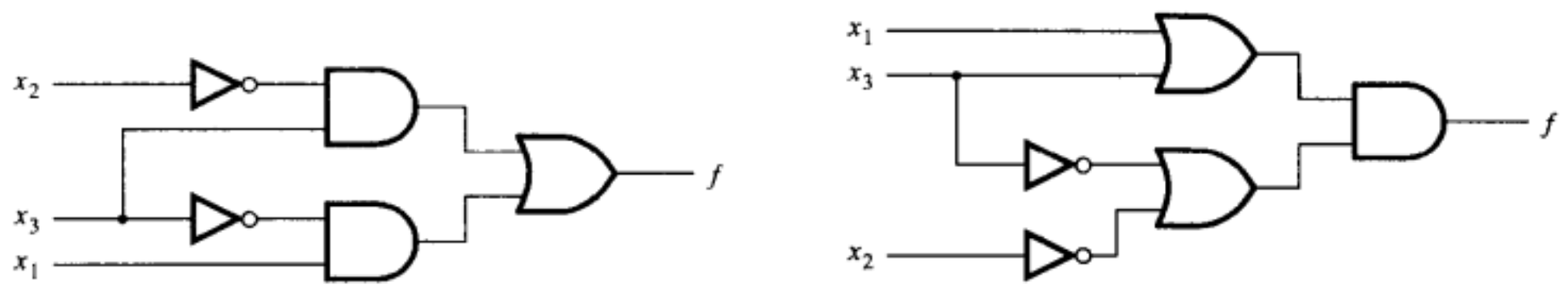
再举一个例子，考虑三变量函数 $f(x_1, x_2, x_3)$，其真值表如图2-18所示。为了对此函数进行综合，必须包括最小项 m_1，m_4，m_5和m_6。根据图2-17，可以得到如下的正则积之和表达式：

$$f(x_1,\ x_2,\ x_3)=\bar{x}_1\bar{x}_2x_3+x_1\bar{x}_2\bar{x}_3+x_1\bar{x}_2x_3+x_1x_2\bar{x}_3$$

对该表达式进行如下化简：

$$\begin{aligned} f(x_1,\ x_2,\ x_3) &= (\bar{x}_1+x_1)\bar{x}_2x_3+x_1(\bar{x}_2+x_2)\bar{x}_3 \\ &= 1\cdot\bar{x}_2x_3+x_1\cdot 1\cdot\bar{x}_3 \\ &= \bar{x}_2x_3+x_1\bar{x}_3 \end{aligned}$$

这是 f 的最低成本的积之和表达式。它描述的电路如图2-19a所示。评价逻辑电路成本高低的一个重要指标是统计电路中逻辑门的总数加上各个门的输入端的总数。根据统计得到的数字，图2-19a所示的电路网络总成本是13，因为共有5个门和8个门输入。通过比较，网络以正则积之和为表达式来实现的话，成本将高达27。根据前面的表达式，可以看到或门有四个输入，四个与门每个有三个输入，三个非门每个有一个输入。



a) 最小的积之和实现　　b) 最小的和之积实现

图2-19　图2-18函数的两种实现

用带有行号下标的最小项可以简明地表示一个给定的逻辑函数。例如，如图2-18所示的函数可以表示为：

$$f(x_1, x_2, x_3) = \sum (m_1, m_4, m_5, m_6)$$

还可以更简单地表示为：

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

求总和号 $\sum$ 表示逻辑和运算。在实际工作中经常应用这种简明的符号。

最大项

对偶原理指出：若考虑真值表中使 $f=1$ 的各行可以综合出一个函数 f，则考虑使 $f=0$ 的各行也可以综合出函数 f。这种方法利用了最小项的反，称为最大项。三变量函数的所有可能的最大项都在图2-17中列出。用行号 j 为最大项 M_j 做标记，与图中的最小项 m_j 对应，如图2-17所示。

和之积形式

若给定函数 f 用真值表来描述，则它的非 $\overline{f}$ 可以表示为使 $\overline{f}=1$ 的最小项的和，即使 $f=0$ 的各行。例如，对图2-15所示函数，存在：

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2 \\ &= x_1\overline{x}_2\end{aligned}$$

若我们用德摩根定理来对上述表达式求反，则结果为：

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x}_2} \\ &= \overline{x}_1 + x_2\end{aligned}$$

请注意，前面我们是通过对函数 f 的积之和代数表达式进行简化才获得这一表达式的。这里的关键点是：

$$f = \overline{m}_2 = M_2$$

上式中，M_2 是真值表第2行对应的最大项。

再举另一个例子，重新考虑图2-18所示函数。该函数的非可以表示为：

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7 \\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

由此，f 可被表示为：

$$
\begin{aligned}
f &= \overline{m_0 + m_2 + m_3 + m_7} \\
&= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7 \\
&= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\
&= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)
\end{aligned}
$$

该表达式用最大项的积来表示f。

一个由各个和项（逻辑或）的逻辑积（逻辑与）组成的逻辑表达式称为和之积（product of sum）形式。若每个和项都是最大项，则表达式称为给定函数的正则和之积。任何函数f可以通过找到它的正则和之积来实现综合，即从真值表中的每一行中取出使$f = 0$的最大项，把这些最大项组成乘积。

回到前面的例子，我们可以试图降低最大项积表达式的复杂性。利用2.5节介绍的交换律10b和结合律11b，该表达式可改写为

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(x_1 + (\overline{x}_2 + \overline{x}_3))(\overline{x}_1 + (\overline{x}_2 + \overline{x}_3))$$

然后，利用合并律14b，将表达式简化为

$$f = (x_1 + x_3)(\overline{x}_2 + \overline{x}_3)$$

对应的电路网络如图2-19b所示。电路成本为13，恰好与图2-19a中的积之和电路的成本相等。请读者注意，不能因此就认定“积之和”电路的成本在一般情况下与“和之积”电路的成本相等。

用简化算符，上面的函数可以表示为

$$f(x_1,\ x_2,\ x_3) = \Pi(M_0,\ M_2,\ M_3,\ M_7)$$

还可以更简单地表示为

$$f(x_1,\ x_2,\ x_3) = \Pi M(0,\ 2,\ 3,\ 7)$$

符号Π表示逻辑乘（与）运算。

前面的讨论表明如何用逻辑电路来实现逻辑函数，即逻辑函数可以由执行基本功能的门组成的网络来实现。给定函数可以由不同结构的电路来实现，这意味着实现电路的成本是不同的。对设计者来说，一个重要的目标就是降低设计电路的成本。我们将在第4章讨论寻找成本最低实现的最重要的技术。

例2.3 考虑函数

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

该逻辑函数的标准积之和（与或）表达式可以用最小项推得：

$$
\begin{aligned}
f &= m_2 + m_3 + m_4 + m_6 + m_7 \\
&= \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3
\end{aligned}
$$

该表达式可以用2.5节中的等式化简为如下表达式：

$$
\begin{aligned}
f &= \overline{x}_1 x_2 (\overline{x}_3 + x_3) + x_1 (\overline{x}_2 + x_2) \overline{x}_3 + x_1 x_2 (\overline{x}_3 + x_3) \\
&= \overline{x}_1 x_2 + x_1 \overline{x}_3 + x_1 x_2 \\
&= (\overline{x}_1 + x_1) x_2 + x_1 \overline{x}_3 \\
&= x_2 + x_1 \overline{x}_3
\end{aligned}
$$

■

例2.4 再次考虑例2.3中的函数。这次我们不用最小项，而用最大项的积来表示该函数。

换言之，这次我们表示的是卡诺图中$f=0$的部分。

$$f(x_1,\ x_2,\ x_3)=\Pi M(0,\ 1,\ 5)$$

因此，标准的和之积（或与）表达式可以表示为

$$\begin{aligned}f&=M_0\cdot M_1\cdot M_5\\&=(x_1+x_2+x_3)(x_1+x_2+\bar{x}_3)(\bar{x}_1+x_2+\bar{x}_3)\end{aligned}$$

简化的和之积表达式可以从如下推导中得到：

$$\begin{aligned}f&=((x_1+x_2)+x_3)((x_1+x_2)+\bar{x}_3)(x_1+(x_2+\bar{x}_3))(\bar{x}_1+(x_2+\bar{x}_3))\\&=((x_1+x_2)+x_3\bar{x}_3)(x_1\bar{x}_1+(x_2+\bar{x}_3))\\&=(x_1+x_2)(x_2+\bar{x}_3)\end{aligned}$$

请注意：通过使用分配律12b，该表达式可以表示为

$$f=x_2+x_1\bar{x}_3$$

上述表达式与例2.3推导得到的表达式相同。 ■

例2.5 假设一个四变量的函数由以下表达式定义：

$$f(x_1,x_2,x_3,x_4)=\sum m\ (3,7,9,12,13,14,15)$$

其标准积之和表达式为

$$f=\bar{x}_1\bar{x}_2x_3x_4+\bar{x}_1x_2x_3x_4+x_1\bar{x}_2\bar{x}_3x_4+x_1x_2\bar{x}_3\bar{x}_4+x_1x_2\bar{x}_3x_4+x_1x_2x_3\bar{x}_4+x_1x_2x_3x_4$$

更简化的积之和表达式可以按照如下步骤推导得到：

$$\begin{aligned}f&=\bar{x}_1(\bar{x}_2+x_2)x_3x_4+x_1(\bar{x}_2+x_2)\bar{x}_3x_4+x_1x_2\bar{x}_3(\bar{x}_4+x_4)+x_1x_2x_3(\bar{x}_4+x_4)\\&=\bar{x}_1x_3x_4+x_1\bar{x}_3x_4+x_1x_2\bar{x}_3+x_1x_2x_3\\&=\bar{x}_1x_3x_4+x_1\bar{x}_3x_4+x_1x_2(\bar{x}_3+x_3)\\&=\bar{x}_1x_3x_4+x_1\bar{x}_3x_4+x_1x_2\end{aligned}$$

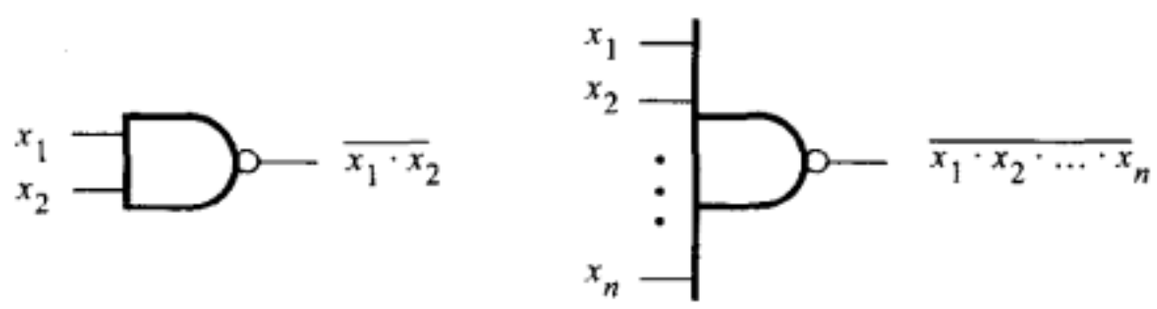
■

2.7 与非以及或非逻辑网络

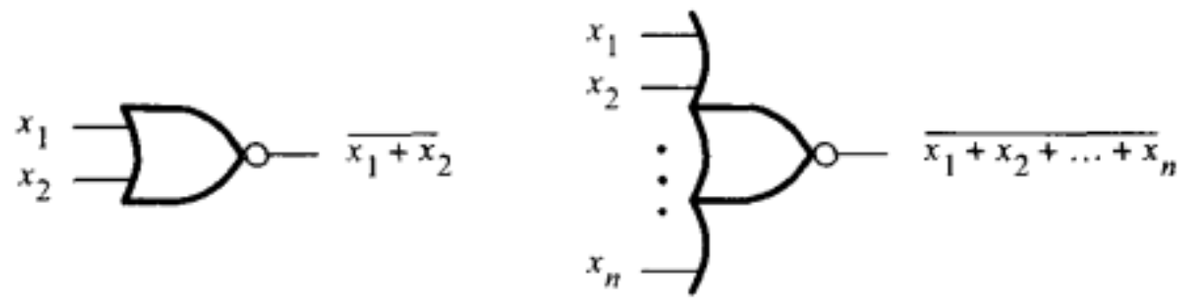
我们已经讨论了把与、或、非门用在逻辑电路的综合上。还有其他的基本逻辑函数也可以用于逻辑电路的综合。特别有用的是与非和或非函数，它们可以分别通过与/或运算的输出求反得到。这些函数之所以要引起注意，是因为它们的电路比与/或函数的电路更加简单，关于这一点，我们将在第3章中介绍。图2-20给出了与非门及或非门的图形符号。对输出信号的取反，可用画在与门/或门输出端的一个小圆圈来表示。

若与非门和或非门的实现电路比与门和或门的电路更简单，则我们就该考虑是否可以在逻辑电路的综合时，直接使用与非门和或非门。在2.5节，我们介绍了德摩根定律。图2-21用逻辑门解释了德摩根定律的原理。恒等式15a的解释见图2-21a所示。 该图详细说明了输入变量为x_1和x_2的与非门等价于先把x_1和x_2求反再输入或门。请注意在最右边的那张图上，那些添加的小圆圈表示的就是非门，表明在该点逻辑值取反。德摩根定律的另一半，恒等式15b，由图2-21b说明，或非门等价于信号取反后输入的与门。

在2.6节中，我们解释了为什么任何逻辑函数既可以用积之和形式的电路又可以用和之积形式的电路来实现，这又导致逻辑函数的实现电路网络可以分为与-或和或-与两种结构。我们现在将展示，这样的电路网络可以只用与非门或者只用或非门来实现。

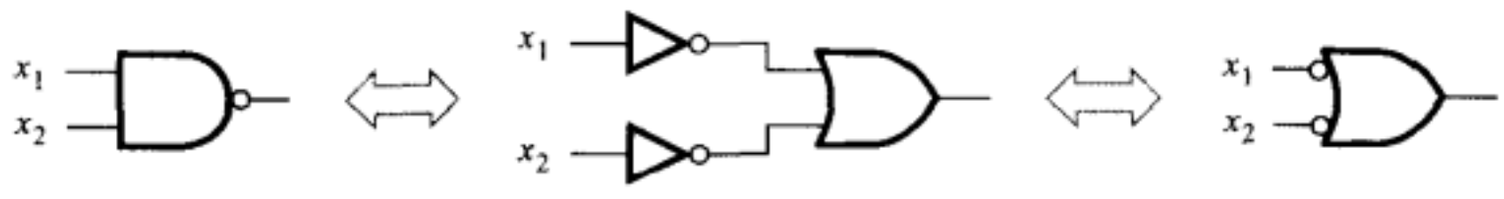


a) 与非门

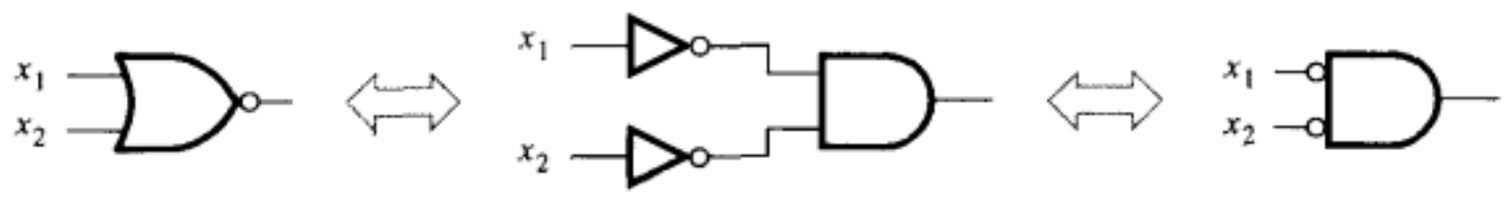


b) 或非门

图2-20 与非门和或非门



a) $\overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$



b) $\overline{x_1 + x_2} = \overline{x}_1 \overline{x}_2$

图2-21 逻辑门项中的德摩根定律

考虑图2-22所示的电路网络，作为常见的与-或网络的代表。该网络可以被转换成只使用与非门的网络，见图所示。首先，每个与门与或门之间的节点被替换成一个包含两个取反的

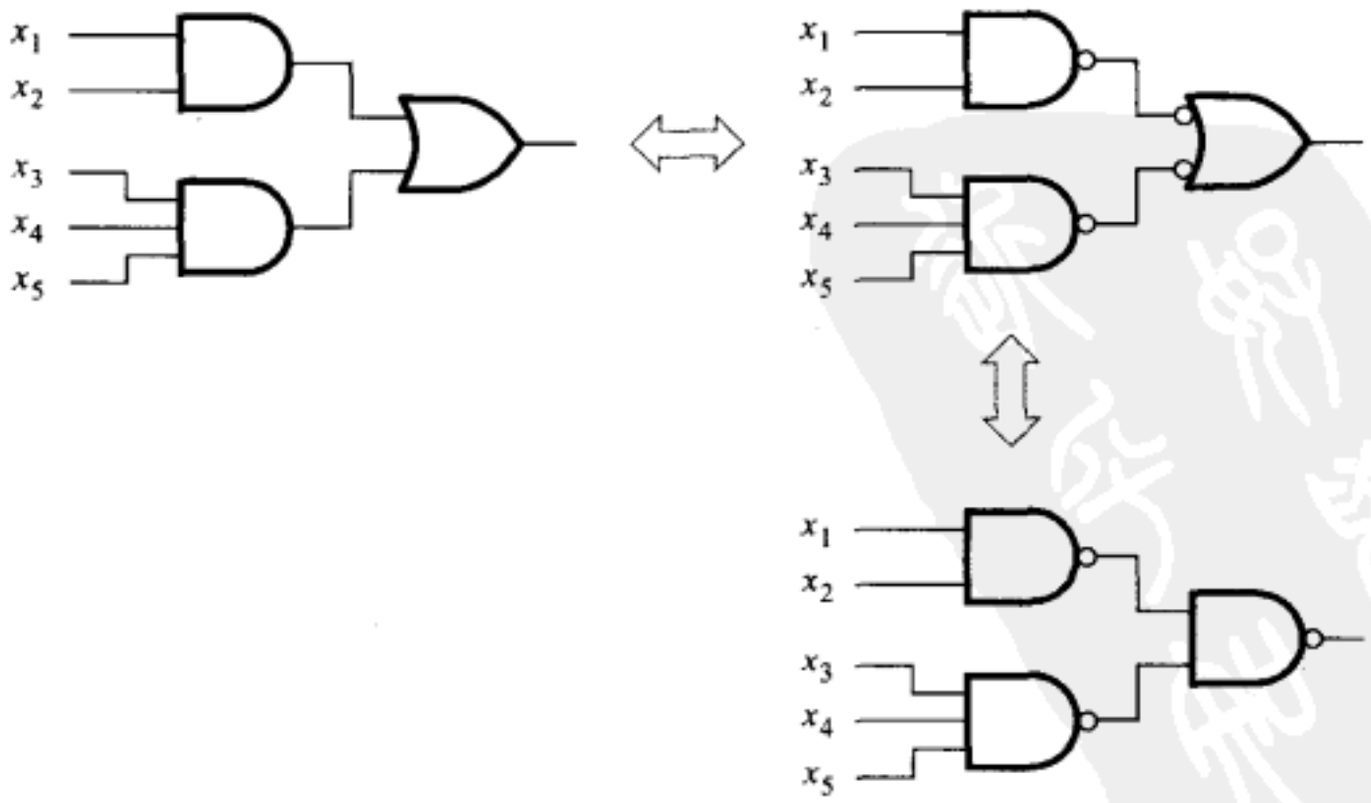


图2-22 使用与非门实现积之和形式

信号的节点：一个是在与门的输出端取反，另一个是在或门的输入端取反。这样的双重取反，根据2.5节的定理9，对该网络的行为没有影响。根据图2-21a，对其输入取反的或门等价于与非门。因此，我们可以只用与非门重画这个网络，见图2-22。这个例子体现了任何与-或网络都可以用同样拓扑结构的与非-与非网络来实现。

图2-23所示的是一个类似的和之积网络结构，该网络可以转化成只使用或非门的网络。这个过程，除了用到图2-21b推导出的恒等式外，和图2-22所做的解释完全一致。得到的结论就是任何或-与网络都可以用同样拓扑结构的或非-或非网络实现。

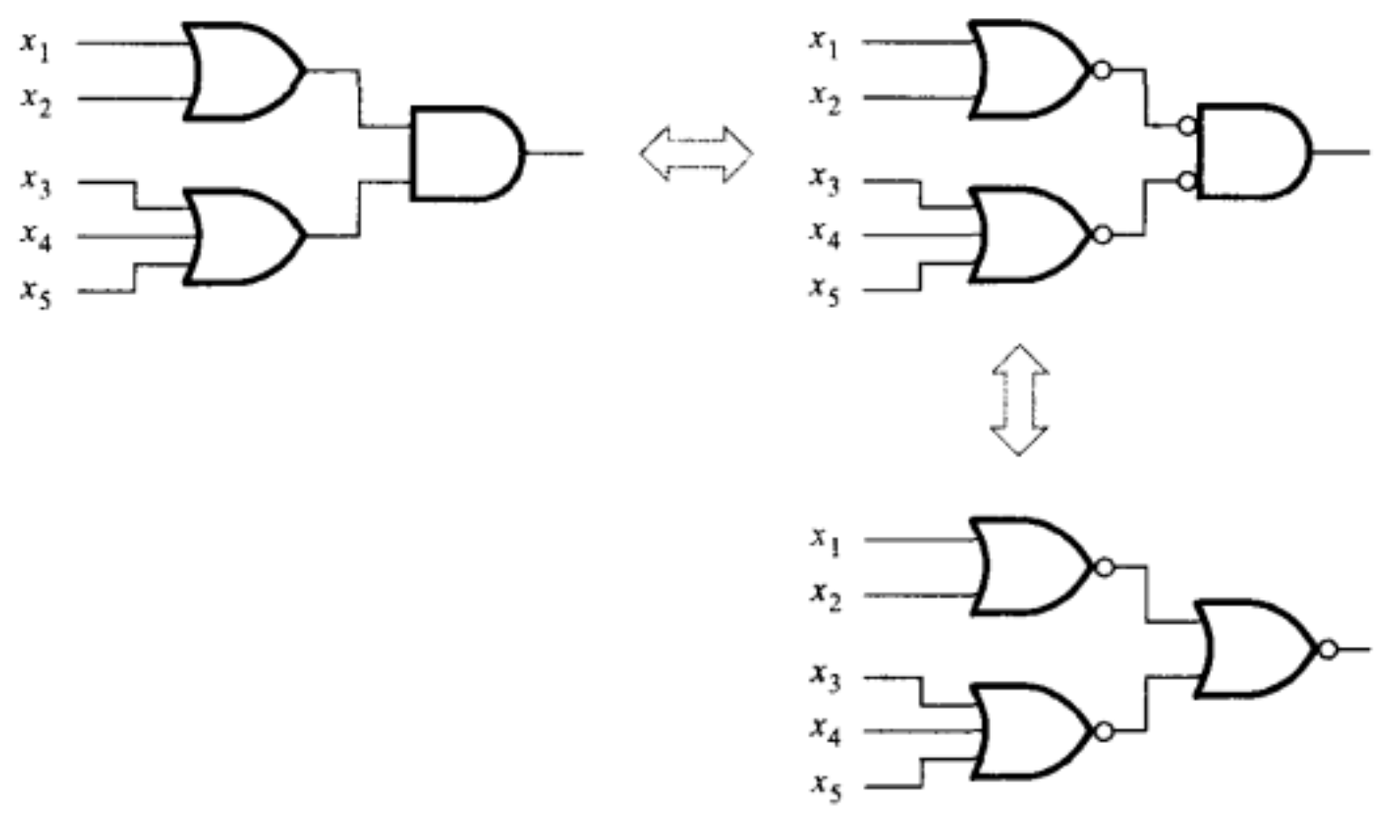


图2-23 使用或非门实现和之积形式

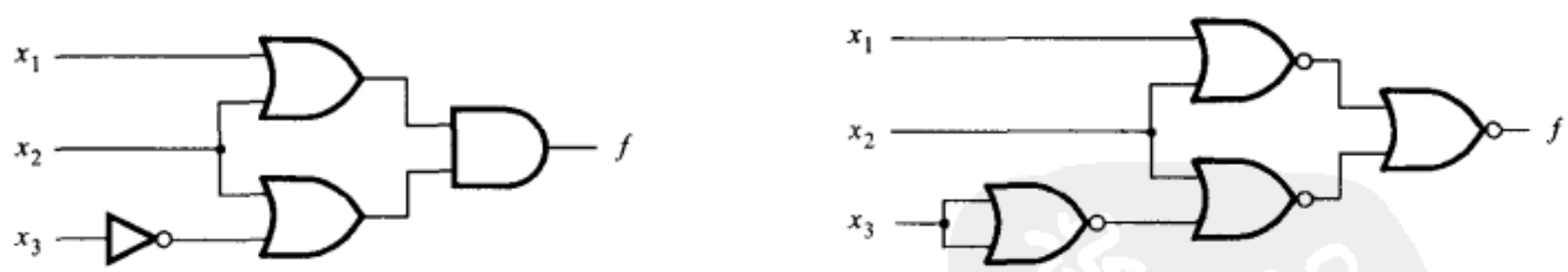
例2.6 只用或非门实现以下函数：

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

在例2.4中我们曾展示该函数可以表示为和之积表达式：

$$f = (x_1 + x_2)(x_2 + \overline{x}_3)$$

对应于该表达式的或与电路如图2-24a所示。该逻辑函数也可以用相同的电路结构，全部采用或非门实现，如图2-24b所示。请注意：我们用一个两输入端连接在一起的或非门将x_3反相。



a) 和之积实现　　b) 或非门实现

图2-24 采用或非门实现例2.4中的函数 ■

例2.7 只用与非门实现以下函数：

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

在例2.3中我们曾推导出如下的积之和表达式：

$$f = x_2 + x_1\overline{x}_3$$

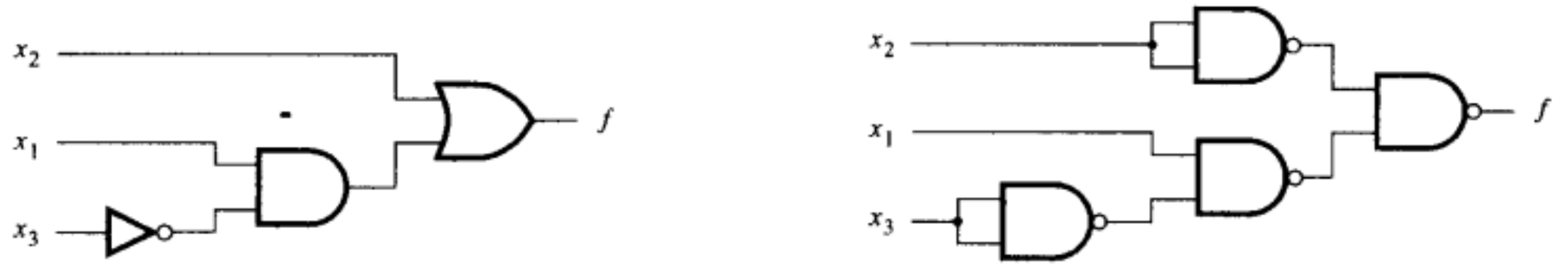
该表达式可以用图2-25a所示的电路实现。我们仍旧可以用同样的结构（除一个差别外）

得到全部用与非门实现的电路。信号x_2只通过一个或门，而不是通过一个与门和一个或门。若我们只是简单地用一个与非门来替换或门，信号x_2就会反相，从而出现错误的输出。既然必须保证x_2不反相，或者反相两次，我们就可以将x_2通过两个与非门达到此目的，如图2-25b所示。请注意观察该电路输出函数f的表达式为

$$f=\overline{\overline{x_2}\cdot\overline{x_1\overline{x_3}}}$$

应用德摩根（DeMorgan）定律，上述表达式可以表示为

$$f=x_2+x_1\overline{x_3}$$



a) 积之和实现　　b) 与非门实现

图2-25　采用与非门实现例2.3中的函数

2.8 设计举例

逻辑电路提供了问题的解。逻辑电路实现执行专门任务必须完成的逻辑功能（函数）。在计算机架构内，逻辑电路提供执行程序和处理数据的能力。这样的电路是十分复杂的，并且很难设计。但是若不考虑给定电路的复杂性，逻辑电路的设计者总是面对相同的基本问题。首先，必须详细描述所需电路的行为；其次，必须把描述综合成电路并实现之；最后，必须对已实现的电路进行测试，以验证其确实符合所要求的行为。这些所要求的行为最初通常是用语言描述的，必须将其转换为一种正规的格式。本节给出了两个简单的设计范例。

2.8.1 三路灯光控制

假定一个大屋子有三个门，每个门边有一个开关去控制大房间的同一个灯。要求改变任何一个开关都能够开灯或者关灯。

第一步，把这一段描述转变成真值表。令x_1、x_2和x_3分别表示各开关的状态，即逻辑函数的输入变量。假定所有开关断开时灯熄灭。闭合其中任一开关，都可以将灯点亮。然后闭合另一个开关，将使灯熄灭。因此若一个开关闭合，则灯亮；若两个（或没有）开关闭合，则灯熄灭；若两个开关闭合时灯熄灭，那么闭合第三个开关，则灯亮。用$f(x_1, x_2, x_3)$表示灯的状态，那么所要求的功能行为可用真值表说明，如图2-26所示。因此，指定逻辑功能的函数其积之和表达式为：

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

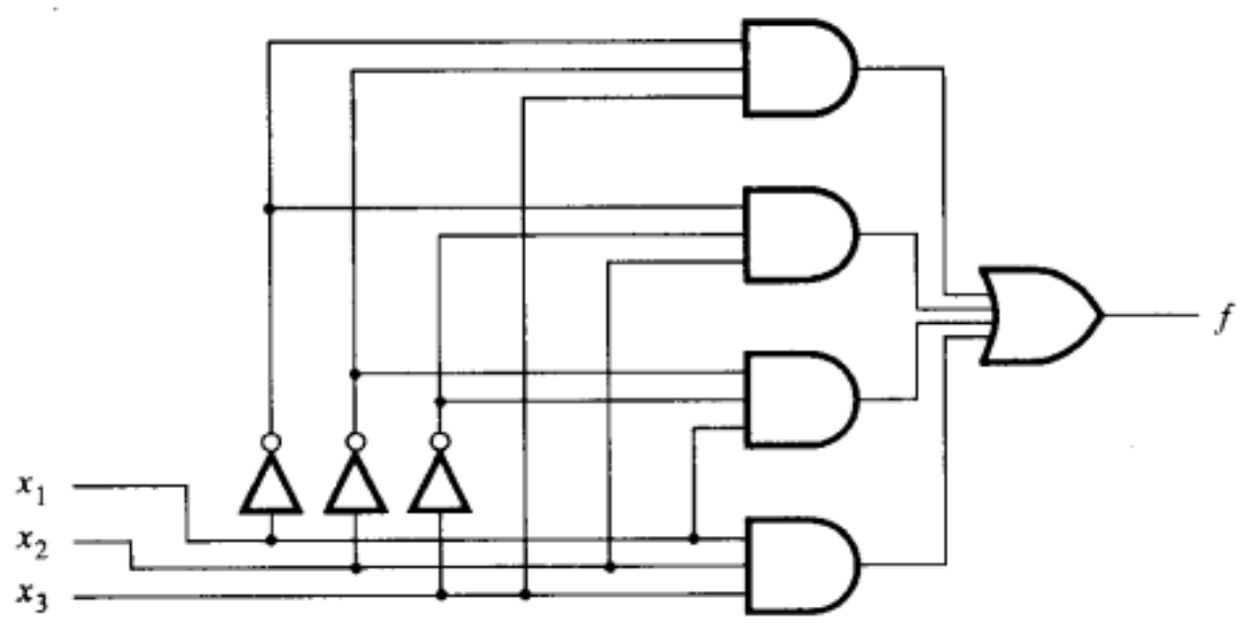
图2-26　三路灯控制的真值表

$$\begin{aligned}f&=m_1+m_2+m_4+m_7\\&=\overline{x}_1\overline{x}_2x_3+\overline{x}_1x_2\overline{x}_3+x_1\overline{x}_2\overline{x}_3+x_1x_2x_3\end{aligned}$$

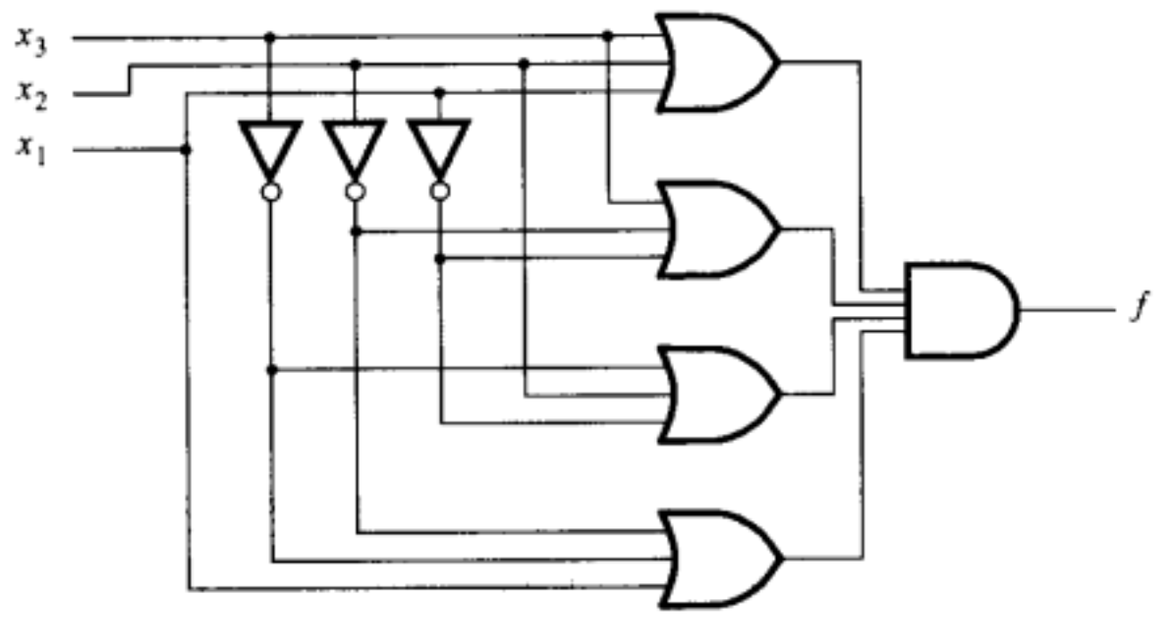
该表达式不能再简化为成本更低的积之和表达式。结果电路如图2-27a所示。

该逻辑函数的另一种实现是用和之积的形式。其表达式为

$$\begin{aligned}f&=M_0\cdot M_3\cdot M_5\cdot M_6\\&=(x_1+x_2+x_3)(x_1+\overline{x}_2+\overline{x}_3)(\overline{x}_1+x_2+\overline{x}_3)(\overline{x}_1+\overline{x}_2+x_3)\end{aligned}$$



a) 积之和的实现



b)和之积的实现

图2-27 图2-26函数功能的实现

结果电路如图2-27b所示。它与图2-27a电路的成本相同。

设计电路实现后，对电路施加各种不同的输入取值，检查输出是否与真值表中描述的相应输出值一致。最直接的方法就是检查8个可能的输入取值是否都产生正确的输出。

2.8.2 多路选择器电路

在计算机系统中，经常需要从几个数据源中挑选出一个信号输出。假如有两个数据源，分别为x_1和x_2，其信号值随时间而改变，假设它们经有规律的时间间隔而改变。于是，就在x_1和x_2每个输入端上施加了由0和1组成的序列。我们希望设计这样的一个电路：使其输出值等于x_1或者x_2中的一个输入序列。究竟选择哪一个输出由控制信号s的值决定。所以，电路应该有三个输入x_1、x_2和s。若$s=0$，则电路的输出值就是x_1；若$s=1$，则电路的输出值就是x_2。

基于这些要求，我们以真值表的形式详细指定了想要设计的电路，见图2-28a。从真值表中我们可以得出积之和表达式：

$$f(s,\ x_1,\ x_2)=\bar{s}x_1\bar{x}_2+\bar{s}x_1x_2+s\bar{x}_1x_2+sx_1x_2$$

利用分配律，该表达式可以写作：

$$f=\bar{s}x_1(\bar{x}_2+x_2)+s(\bar{x}_1+x_1)x_2$$

应用定理8b得：

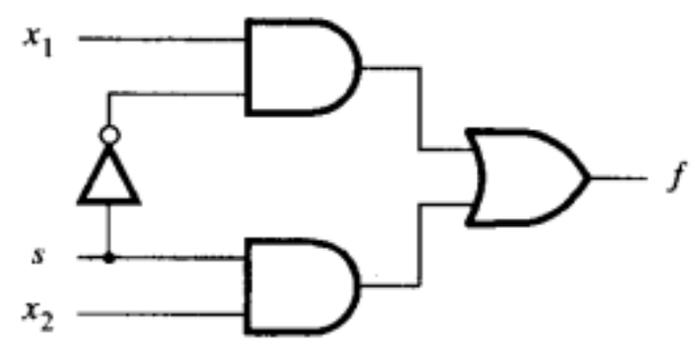
$$f=\bar{s}x_1\cdot 1+s\cdot 1\cdot x_2$$

最后，由定理6a，得到表达式：

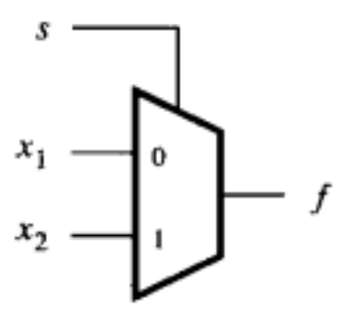
$$f = \bar{s}x_1 + sx_2$$

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

a) 真值表



b) 电路图



c)图形符号

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

d) 更紧凑的真值表

图2-28 多路选择器的实现

图2-28b表示了实现该功能的电路。这种电路由于得到广泛的应用，因此有一个专门的名字。该电路根据一个或多个选择控制输入，从多路输入信号中选取一路输出，这种电路称为*多路选择器*（简称多路器）。我们说多路选择器，是指它能从多路输入中选取一路输出。

在本例中，我们推导出了有两个数据输入的多路器，称为“2选1多路器”的电路。图2-28c展示了常用的2选1多路器的图形符号。同样的概念可以扩展到更大的电路。例如4选1多路器有四个输入一个输出。这样就需要两个输入控制信号来挑选四个输入信号中的一个作为输出。8选1多路器需要8个数据输入和3个控制输入信号，等等。

请注意，“若$s = 0$，则$f = x_1$，且若$s = 1$，则$f = x_2$”可以更简洁的真值表的形式表示，见图2-28d。在后面的一些章节中，我们将使用这种真值表的表示法。

前面介绍了怎样由与门、或门和非门来构建多路器。用相同的电路结构，但只用与非门，也可以构建多路器，具体解释见2.7节的介绍。在第3章中我们将介绍构建多路器的其他方法。在第6章中，我们还将更详细地讨论多路器的使用。

逻辑电路设计者的工作严重地依赖CAD（计算机辅助设计）工具。我们想鼓励读者尽可能地熟悉本书所提供的CAD工具的支持。我们已经认识到了学会使用这些工具的重要性，下一节中将介绍使用这些工具必须掌握的基本概念。在2.10节我们也将介绍一种描述逻辑电路的专门语言，称为Verilog。该语言用于电路的描述，并将其作为CAD工具的输入，然后推导出合适的电路实现。

2.9 计算机辅助设计工具简介

前面几节中，我们介绍了逻辑电路综合的基本途径。设计者能够通过这个途径用手工设计并综合出简单的电路。然而，复杂系统的逻辑电路，例如当今的计算机，根本不可能用手工完成设计。它们是采用复杂的自动实现综合技术的CAD工具完成设计的。

为了设计逻辑电路，需要很多种CAD工具。它们通常打包在一起成为计算机辅助设计系统，一般包括以下功能：设计输入，综合和优化，仿真，以及物理设计。本节将介绍其中的一些工具，在后面的章节里还将继续讨论这个问题。

2.9.1 设计输入

设计逻辑电路过程的起点是搞清楚电路所要完成的功能和它的一般结构形式。这个工作是由设计者手工完成的，因为需要设计者的工作经验和直觉。设计过程的其余部分是借助于CAD工具完成的。这个步骤称为设计输入，即向CAD系统输入想要完成的设计描述。我们将介绍两种不同的设计输入方式：用原理图获取和用硬件描述语言编写的源代码。

原理图获取

通过绘制逻辑门并将这些逻辑门用线路连接起来，我们可以定义一个逻辑电路。支持用这种方法将设计电路输入计算机的CAD工具称为*原理图获取*（Schematic Capture）工具。所谓原理图是指一张绘有许多电路元件的图形符号（如逻辑门等元件图形符号）和电路元件之间连接线的图。

原理图获取工具利用计算机的图形能力，允许设计者通过操作鼠标绘制原理图。为了方便地将逻辑门绘制到电路图中，该工具提供了由许多图形符号组成的集合，这些图形符号表示不同类型的门，其输入端的个数也各不相同。这样的符号集合称为*库*。该库中的门可以添加到设计者绘制的原理图中，原理图获取工具还提供了将这些门互相连接组成原理网络的绘图方法。

任何以前绘制的子电路都能表示为图形符号加以保存，随时可以添加到当前绘制的原理图中。在实际工作中，CAD系统的用户在创建规模较大的电路图时将小电路的图形符号添加到原理图中是很常见的。这种方法学称为*分层次的设计方法*，它为处理复杂的大型电路提供了一种很好的手段。

在附录B中，我们详细地描述了如何用原理图获取方法来绘制电路图。该方法虽然用起来很简单，但是当电路规模很大时，会变得非常麻烦。对规模较大的电路而言，更好的手段是用硬件描述语言编写代码的办法来表示。

硬件描述语言

硬件描述语言（HDL）类似于计算机程序，所不同的是硬件描述语言描述的是硬件，而不是计算机上运行的程序。市场上有许多种商业的硬件描述语言可用。有些硬件描述语言是某个公司的专有财产，这意味着该硬件描述语言只能由该公司提供，并必须使用该公司提供的技术来实现电路。本书不准备讨论这类公司专有的硬件描述语言，而是着重介绍电气和电子工程师协会（IEEE）认可的符合IEEE标准的数字硬件描述语言，该语言得到全世界所有厂商的支持。IEEE是一个世界性的组织，其宗旨是促进全世界的技术进步，其中一项活动是制订技术标准和规范，以使这些技术概念更加适用于更大的人群。

目前世界上有两种硬件描述语言符合 IEEE 标准，它们分别是Verilog HDL和VHDL (Very High Speed Integrated Circuit Hardware Description Language，超高速集成电路硬件描述语言)。这两种语言都在工业界得到广泛的应用。本书中，我们采用Verilog，但本书的VHDL版本可以参见[4]。尽管两种语言在很多方面都有所不同，但是学习逻辑电路时，选择哪一种语言并不特别重要，因为两者的特性很相似。本书中用Verilog阐述的概念也可以直接用于VHDL。

与原理图获取的电路设计方法相比较，Verilog具有很多优点。因为大多数提供数字硬件技术的公司支持Verilog。Verilog提供了设计的*可移植性*。采用Verilog语言描述的电路能够用

不同的芯片实现，并可使用不同公司提供的CAD工具，而不必改写描述该电路的Verilog代码。由于数字电路技术发展得很快，所以设计可移植性是一个重要的优点。通过使用标准化的语言，设计者只需集中精力于电路的功能，而不必过多考虑最终在实现时所用的技术细节。

编写Verilog 源代码就可以完成逻辑电路的设计输入。电路中的信号可以用源代码中的变量来表示，逻辑功能可以表示为对这些变量的赋值。Verilog源代码是纯文本文件，对设计者而言，在代码文件中添加一些注释，解释电路是如何工作的是很容易的。这一优点再加上Verilog的广泛应用，促进了用Verilog描述的电路的共享和重用。若有现成的Verilog代码适用于新产品的电路设计，则将大大加快新产品的开发。

类似于用原理图获取设计大型电路的方法，用Verilog语言也可以编写出支持层次化设计的模块代码。无论小型逻辑电路，还是大型的逻辑电路的设计，都能用Verilog代码进行有效的描述。Verilog已被用于定义含有数百万枚晶体管的微处理器。

Verilog设计输入也可以与其他方法相结合。例如，原理图获取工具可使用由Verilog描述的子电路。我们将在2.10节中介绍Verilog。

2.9.2 综合

综合是从初始的设计描述（可以是原理图的形式或者以硬件描述语言代码形式表示的设计细节）生成逻辑电路的过程。CAD综合工具能根据设计指标的描述自动地生成高效的具体电路。

将Verilog代码转换（或编译）成逻辑门网络的过程是综合的一部分。

无论使用哪种类型的设计输入，由综合工具产生的最初的逻辑表达式一般不可能是最优的形式。因为这些表达式反映的是设计者输入CAD工具的设计要求，对设计者而言，他自己很难用手工推导出最优结果，特别当电路规模很大时更是如此。综合工具的最重要的任务之一是：根据用户的设计自动地产生等价的但是更优化的电路。

某个电路优于另一个电路的度量取决于设计项目的特殊需求和选用的实现技术。在2.6节我们曾提到，成本最低的电路也许就是最好的电路。还存在许多其他的优化目标，这些目标是由实现电路所用的硬件技术的类型所推动的。我们将在第3章中讨论实现技术问题，在第4章再回到优化目标问题。

由综合产生的电路的性能可以通过对实际电路的测试加以评价。而电路的行为可以借助于仿真加以评估。

2.9.3 功能仿真

以逻辑表达式形式表示的电路可以通过仿真来验证该电路的功能是否与预期一致。能完成这个任务的工具称为*功能仿真器*。功能仿真器使用在综合期间生成的逻辑表达式（通常称为逻辑方程），并假设这些表达式用完全理想的没有任何延迟的逻辑门实现其功能。仿真器要求用户指定在仿真期间应施加到电路输入端信号的取值。对每一组取值，仿真器通过逻辑表达式计算出电路的输出。仿真的结果通常以时序图的形式提供，用户通过检查该时序图，可以验证电路的运行是否符合要求。在附录B中，我们将详细地介绍功能仿真。

2.9.4 物理设计

设计流程中逻辑综合后的下一个步骤是确定如何在一个给定的芯片中切切实实地实现该电路。这个步骤通常称为*物理设计*。正如我们在第3章中将会看到的那样，可以采用多种不同的技术来实现该逻辑电路。物理设计工具把用逻辑表达式表示的电路映射成具体的物理电路，并予以实现，在这个实现过程中可利用目标芯片中已存在的可用资源。物理设计工具可以确定特定逻辑元件的布局，这些逻辑元件不一定局限于到目前为止我们所遇到过的简单的逻辑

门。物理设计工具还能确定实现预期电路的元件之间所必需的布线连接。

2.9.5 时序仿真

逻辑门和其他逻辑元件都可以用电子电路来实现，我们将在第3章中讨论这个问题。具体的电子电路不可能实现没有任何延迟的逻辑功能。当电路的输入值发生变化时，必须花费一定的时间才能在输出端观察到相应的变化。这段时间称为电路的传导延迟。传导延迟由两种延迟组成：一种是元件本身产生的延迟，即元件输入端信号值发生变化到输出端观察到相应的信号变化所需的时间。另一种是不同逻辑元件之间的连接线存在的延迟。实际电路所表现的延迟是这两种延迟共同作用的结果，由此对逻辑电路的运行速度产生了显著的影响。

时序仿真器对已设计逻辑电路的预期延迟进行估算。估算的结果可以用来确定生成的电路是否满足了设计所要求的时序技术指标。若时序指标不能达到要求，则设计者可以在标明必须满足的特定时序约束后，用物理设计工具再运行一次。若这样做还没有成功，则设计者必须在综合阶段用不同的优化方法试验一下，否则必须对提交给综合工具的初始设计进行修改。

2.9.6 芯片配置

在确认所设计的电路已经完全达到各项技术指标后，我们就可以在真实的芯片上实现该电路。这一操作步骤称为芯片的配置（configuration）或者编程（programming）。

本节讨论的CAD工具是CAD系统最核心的几个部分。我们已讨论过的完整的设计流程如图2-29所示。这里所介绍的只是简要的入门知识。在第12章，我们将全面地向读者介绍CAD工具。

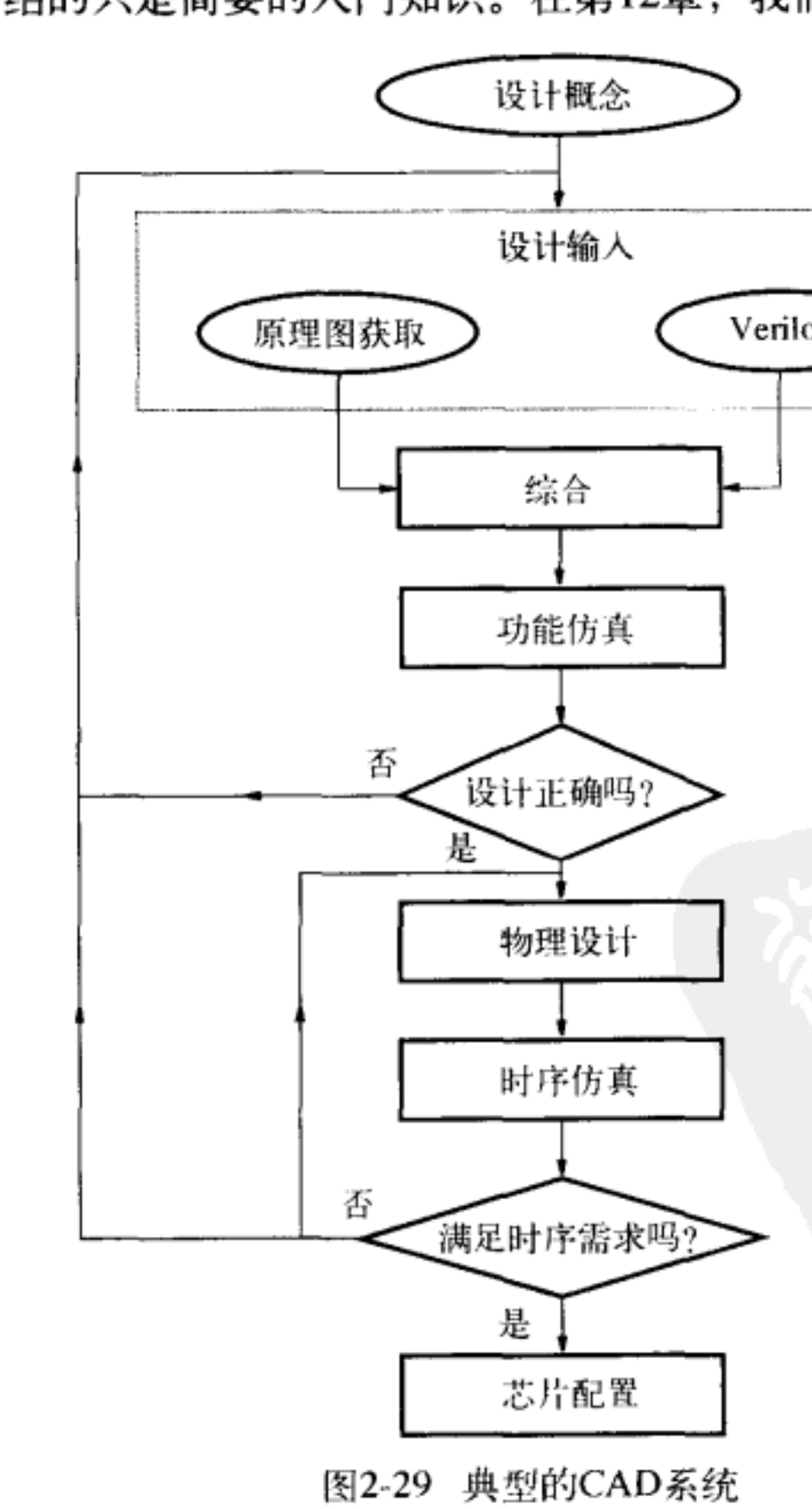


图2-29 典型的CAD系统

到目前为止，读者应该已初步了解使用CAD工具时将涉及的一些知识。但是要真正地全面理解工具，必须亲自动手操作才行。在附录B到D中，我们提供了循序渐进的自学教程，向读者介绍如何使用本书附带的Quartus II CAD系统。我们强力建议读者通过使用这几个附录所介绍的材料进行工作。因为这些辅助材料都使用Verilog作为设计输入，在下一节中，我们将介绍Verilog的入门知识。

2.10 Verilog简介

在20世纪80年代，集成电路的快速发展导致人们努力开发数字电路的标准化设计方法，Verilog 就是这种努力的成果之一。最初版本的Verilog是由Gateway Design Automation公司开发的，随后被 Cadence Design System公司收购。1990年，Verilog被公开推向市场，从那以后，Verilog 就成为最流行的描述数字电路的语言。1995年，Verilog 正式被批准为IEEE的标准，叫做1364—1995。Verilog的增强版本叫做Verilog 2001，于2001年正式被批准为IEEE标准，叫做1364—2001。这一版本引入了一些新特性，它支持原始Verilog版本的所有特性。

Verilog 最初是想用来做数字电路的仿真和验证的。后来添加了逻辑电路综合的能力。在CAD系统中，用Verilog做设计输入变得日益广泛。CAD工具可以用来将Verilog代码转化成描述电路的硬件实现。在本书中我们主要使用 Verilog 进行综合。

Verilog 是一门复杂的全面的语言，学会它所有的功能是一项繁重的任务。然而就电路综合而言，只需要掌握Verilog 的一个子集。为了简化描述，我们将集中讨论本书例子用到的功能。本书所介绍的材料足以使读者设计出很多种类型的数字电路。想要完整地学习Verilog语言的读者，可以参考本章的参考文献[5-11]。Verilog的许多语言结构和C编程语言很类似，懂得C语言的读者将会发现，学会Verilog并不困难。

在本书中，我们分几个阶段来介绍Verilog。本书的总体方针是只介绍书中设计课题所涉及的Verilog特定的语法现象。在附录A中，我们对本书所涉及的Verilog语法条款作了简要的总结。读者将会发现，语法总结为参阅常用的语法条款带来了方便。本章的其余部分讨论了编写简单的Verilog代码必须掌握的最基本的概念。

数字电路的Verilog描述

我们曾在2.9.1节中介绍过，设计者可以用几种不同的方式描述想要设计的电路，然后用CAD工具将其综合成具体的逻辑电路。其中一个有效率的描述电路的方法是编写该电路的Verilog源代码，然后由Verilog编译器将该代码转换成逻辑电路。

Verilog允许设计者用两种不同的基本方式来表示逻辑电路。一种方式是使用表示电路元件的Verilog结构，例如用逻辑门，甚至于用晶体管。可以用编写代码的方式把表示电路元件的Verilog结构连接起来组成较大的电路。这种方式叫做逻辑电路的*结构表示*。第二种描述逻辑电路的方式是使用逻辑表达式和编程结构，它们描述的是电路的行为，而并非真实的用门表示的结构。逻辑电路的这种描述叫做*行为描述*。

2.10.1 逻辑电路的结构描述

Verilog包含了一套对应于常用逻辑门的*门级原语*（gate level primitives）。逻辑门由它的函数（功能）名、输出和输入来描述。例如一个二输入端的与门，输入是x_1，x_2，输出是y，可以表示为AND（y，x_1，x_2）；一个四输入端的或门可以表示为OR（y，x_1，x_2，x_3，x_4）；关键字NAND和NOR与表示与非门和或非门的英文字相同。非门可以表示为NOT（y，x）；也可以表示为 $y = \overline{x}$。门级原语可以用来定义更大的电路。附录A中的表A-2列出所有可用的Verilog门级原语。

逻辑电路可用模块的形式加以定义，模块中包含了定义电路的语句。模块具有输入和输出，通常称为端口。模块端口的命名通常采用与输入或输出端相连接的常用电路的名称。考虑图2-30的电路，描述该电路的Verilog代码如图2-31所示。第一条语句给出模块的名称：example1，并表明该模块有四个信号端口，接下来两句说明x_1，x_2和x_3为输入，f 为输出。电路的具体结构用下面的四条语句说明，NOT（非门）给出$k = \overline{x}_2$；两个AND（与门）分别给出$g = x_1x_2$和$h = \overline{x}_2x_3$。两个与门的输出连接到或门的输入产生模块的输出 f：

$$f = g + h$$
$$= x_1x_2 + \overline{x}_2x_3$$

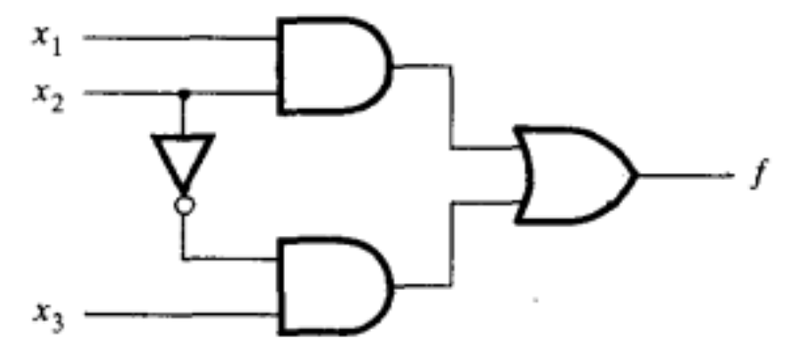


图2-30 简单的逻辑函数

模块用关键字endmodule结束。Verilog关键字在程序中用粗体字表示，以便使程序更容易读懂，本书中我们将遵守这一约定。

第二个Verilog代码举例，如图2-32所示。该代码定义了一个四输入信号（x_1，x_2，x_3和x_4），三个输出信号（f，g和 h）的电路，实现的逻辑函数为

$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \overline{x}_3)(\overline{x}_2 + x_4)$$
$$f = g + h$$

```
module example1 (x1, x2, x3, f);
   input  x1, x2, x3;
   output  f;

   and (g, x1, x2);
   not (k, x2);
   and (h, k, x3);
   or (f, g, h);

endmodule
```

图2-31 图2-30所示电路的Verilog代码

```
module example2 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   and (z1, x1, x3);
   and (z2, x2, x4);
   or (g, z1, z2);
   or (z3, x1, ~x3);
   or (z4, ~x2, x4);
   and (h, z3, z4);
   or (f, g, h);

endmodule
```

图2-32 一个四输入电路的Verilog代码

在代码中，不用显式的NOT（非）门来定义 $\overline{x}_2$ 和 $\overline{x}_3$，而用Verilog操作符号“~”来表示求反。因此，$\overline{x}_2$ 表示为代码中的~x_2，用Verilog编译器生成的电路如图2-33所示。

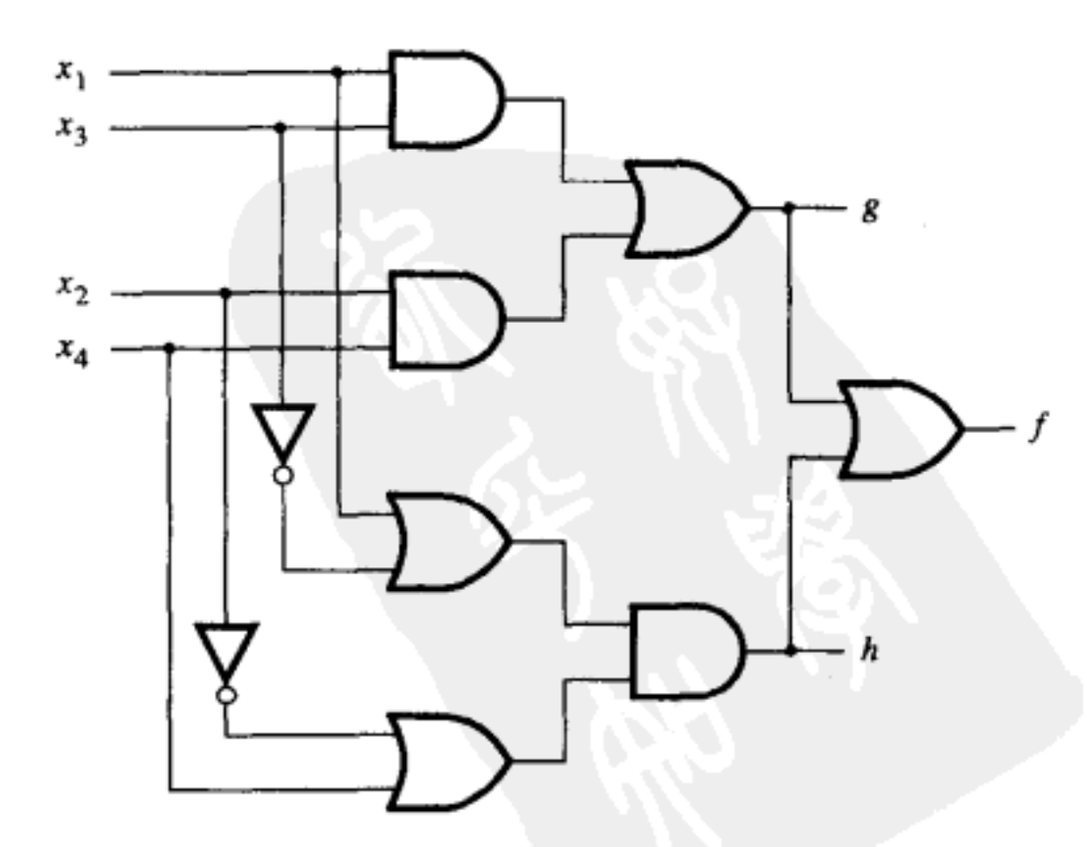


图2-33 图2-32所示代码的逻辑电路图

Verilog语法

Verilog代码的模块和信号的命名遵循两条简单的规则：命名必须以字母起头，其中可以包括任何字母和数字以及下划线符“_”和美元符号$。Verilog语言是区分字符的大小写的，因此，名字k 与K不同，Example1 和example1也不同。Verilog的语

法并不特别强调特定的代码形式，例如，多条语句可以写在同一行中，空格符号（例如SPACE和TAB）和空白行都被忽略。作为好的代码风格，代码应编排得容易阅读，行缩进和空行可以用来隔开代码的不同部分，使代码容易理解，请见图2-31和图2-32。代码中应该包括注释以提高代码的可阅读性。注释语句以双斜杠符“//”起头直至一行的末尾。

2.10.2 逻辑电路的行为描述

当必须设计大型电路时，只使用门级原语是很麻烦的。另外一个办法是使用更抽象的表达式和编程构造来描述数字电路的行为。其中一种方法是使用逻辑表达式来定义电路。图2-34展示了图2-30所示的电路可由以下表达式定义：

$$f = x_1x_2 + \bar{x}_2x_3$$

与操作和或操作分别用符号“&”和“|”表示。assign关键字为信号 f 提供了连续赋值。连续（continuous）这个词来源于应用Verilog进行仿真；只要等式右边的信号状态一旦有所改变，就重新计算 f 的值。其效果等价于使用图2-31所示的门级原语。随后，便可以把图2-33所示的电路编写成如图2-35所示的代码。

逻辑表达式使编写Verilog代码变得更加容易。而用更高层次的抽象通常有更大的优势。再次考虑图2-30所示的电路。这个电路类似于2.8.2节曾经讨论过的2选1多路选择器。x_2作为输入的控制信号，x_1，x_3是数据的输入端。这个电路可以用文字这样描述：若$x_2 = 1$，则 $f = x_1$；若$x_2 = 0$，则 $f = x_3$ 。

```
module example3 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

图2-34 使用连续赋值语句说明图2-30所示的电路

```
module example4 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   assign g = (x1 & x3) | (x2 & x4);
   assign h = (x1 | ~x3) & (~x2 | x4);
   assign f = g | h;

endmodule
```

图2-35 使用连续赋值语句说明图2-33所示的电路

刚才用文字描述的行为可以用**if-else**语句来定义，即

```
if(x2 == 1)
   f = x1;
else
   f = x3;
```

图2-36给出了完整的代码。第一行说明怎样插入注释。而代码中的**if-else**语句是Verilog语言的一种过程性语句。我们还将在第5和第6章介绍Verilog其他类型的过程性语句，例如loop语句。

Verilog语法要求过程性语句必须包含在叫做**always**的结构块中，如图2-36所示。**always**块可以包含一条声明语句，如本例所示，**always**块也可以包含很多条声明语句。典型的Verilog设计模块可以包括几个**always**块，每个**always**块表示电路模型的一个部分。**always**块的一个重要

```
// Behavioral specification
module example5 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;
   reg f;

   always @(x1 or x2 or x3)
      if (x2 == 1)
         f = x1;
      else
         f = x3;

endmodule
```

图2-36 图2-30所示电路的行为说明

特性是它所包含的语句是按代码排列顺序执行的。这与连续赋值语句不同，连续赋值语句是并列执行的，因此没有顺序执行的概念。

always块@符号后面的部分，即圆括号里的部分，称为*敏感列表*。在用Verilog进行仿真的时候，该敏感列表所起的作用是根本性的。**always**块所包含的语句只有在敏感列表中的一个或多个信号值发生改变的时候才会执行。这样的话，仿真过程的复杂性就被简化了，因为没有必要一直执行代码中的每一条语句。当Verilog代码被用来综合成电路时，如本书所讲述的那样，敏感列表只是简单地通知Verilog编译器哪些信号可以直接影响**always**块所产生的输出。

若用过程性语句语言给某个信号赋值，则Verilog语法要求必须先要声明该信号为一个变量；这是通过使用关键字**reg**来实现的，如图2-36所示。这一术语也来自于仿真行话：意思是，一旦某变量在过程性语句中被赋值，则仿真器就“寄存”该值，而且将一直保持不变，直到**always**块再次执行时才有可能改变。我们将在第5章中详细地讨论这个问题。

```
// Behavioral specification
module example6 (x1, x2, x3, f);
   input x1, x2, x3;
   output reg f;

   always @(x1, x2, x3)
      if (x2 == 1)
         f = x1;
      else
         f = x3;

endmodule
```

图2-37 图2-36所示代码的更简单的表示

Verilog 2001标准在指定信号类型和敏感信号列表方面提供了极大的自由度。不必如图2-36的程序所示的那样，用另外一条语句来声明变量 *f* 是**reg**（寄存器）类型的，而可以直接在声明变量 *f*是**output** 的语句中添加**reg**来定义其类型，如图2-37的程序所示。在**always**语句的敏感列表中，我们可以用逗号来代替**or**，如图2-37的程序所示。

而且可以用一个*号来代替敏感列表中的有关信号，所以可以简化为

always @ (*)

或者用更简单的形式：

always @ *

假设编译器能够自己确定哪些信号需要考虑。在模块的端口信号列表中，也可以直接声明信号的类型，这使得代码变得更加紧凑。我们可以将模块端口的定义按照如下方式编写：

module example5 (**input** x1,x2,x3, **output reg** f);

在上述场合，不再需要独立的**input**和**output**语句。然而为了提高可读性，在贯穿全书的举例中，我们使用如图2-37所示的风格。

附录A全面定义了各种Verilog声明语句。

逻辑电路的行为描述语句所定义的只是它的行为。CAD综合工具使用这样的行为描述来构建实际电路。综合所产生的电路的详细结构将取决于所使用的工艺技术。我们将在后面的章节中看到，结构不同的电路可以实现相同的功能行为。

2.10.3 编写Verilog代码必须注意的关键点

在学习怎样使用Verilog或其他硬件描述语言时，初学者往往趋向于采用类似计算机编程的思路来编写代码，在程序中包含了许多变量和循环语句。对这样的程序代码进行综合，CAD工具最终将生成什么样的逻辑电路是很难确定的。本书包含了100多个已经完成了的

Verilog程序代码实例，涵盖范围广泛的各种数字电路。这些示例中的程序代码很容易与其描述的逻辑电路相联系。建议读者采用同样风格的代码。一个很好的一般指导原则是设想：若设计者都不清楚Verilog代码所代表的逻辑电路是什么，则 CAD工具也不可能综合出设计师试图建模的电路。

若为某一特定设计，你已编写了所有的Verilog程序代码，则建议读者分析一下由综合工具所生成的电路。通过这一过程，你将会学到很多有关Verilog、逻辑电路以及逻辑综合的知识。附录A讲述了如何编写Verilog代码。

2.11 小结

本章介绍了逻辑电路的概念。我们展示了这样的电路可以使用逻辑门来实现，也可以用称为布尔代数的数学模型来描述。因为实际逻辑电路经常是大规模的，所以好的计算机辅助设计（CAD）工具对于设计者来说很重要。与本书配套的CAD工具是Quartus II软件，它是由Altera公司提供的一个CAD工具。我们介绍了该工具的一些基本特点，建议读者尽可能早地使用这一软件。

到目前为止，我们的讨论都还相当基本。在后面的章节里，我们将更加深入地讨论逻辑电路和CAD工具等方面的问题。首先，在第3章中，我们将介绍构建逻辑电路的最重要的电子技术。这些材料将帮助读者理解逻辑电路设计师所必须面对的电路实际问题。

2.12 问题求解举例

本节列举了读者可能遇到的某些典型问题，并且展示了这些问题是如何求解的。

例2.8 确定下列方程是否成立。

$$\bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2 = \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3$$

解：若上述方程中左侧表达式与右侧表达式代表同一个函数，则上述方程成立。为了进行比较，我们为等号两侧的表达式各建一个真值表，看一看这两个真值表是否相等。代数求解的途径是分别推导等号两侧表达式的积之和形式的标准逻辑表达式。

应用 $x + \bar{x} = 1$（公理8b）这个事实，可以对左侧的表达式进行如下推导：

$$\begin{aligned}\text{LHS} &= \bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2\\ &= \bar{x}_1(x_2+\bar{x}_2)\bar{x}_3 + (x_1+\bar{x}_1)x_2x_3 + x_1\bar{x}_2(x_3+\bar{x}_3)\\ &= \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3 + \bar{x}_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3\end{aligned}$$

这些乘积项分别代表最小项2，0，7，3，5和4。

对右侧的表达式进行如下推导：

$$\begin{aligned}\text{RHS} &= \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3\\ &= \bar{x}_1x_2(x_3+\bar{x}_3) + x_1(x_2+\bar{x}_2)x_3 + (x_1+\bar{x}_1)\bar{x}_2\bar{x}_3\\ &= \bar{x}_1x_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3\end{aligned}$$

这些乘积项分别代表最小项3，2，7，5，4和0。由于等号左右两侧的表达式所表示的最小项是相同的，因此它们代表的是同一个逻辑函数，于是该方程式成立。该函数的另外一种表示方法是用$\sum m\,(0, 2, 3, 4, 5, 7)$。

例2.9 为下面的逻辑函数设计一个成本最低的和之积表达式：

$$f(x_1, x_2, x_3, x_4) = \sum m\,(0, 2, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

解：上述函数是用最小项定义的。为了找到一个和之积表达式，我们应该从最大项的定义出发，用最大项表示的表达式为 $f=\prod M(1,3,9,11,13)$。

因此

$$\begin{aligned}f&=M_1\cdot M_3\cdot M_9\cdot M_{11}\cdot M_{13}\\&=(x_1+x_2+x_3+\bar{x}_4)(x_1+x_2+\bar{x}_3+\bar{x}_4)(\bar{x}_1+x_2+x_3+\bar{x}_4)\times\\&\quad(\bar{x}_1+x_2+\bar{x}_3+\bar{x}_4)(\bar{x}_1+\bar{x}_2+x_3+\bar{x}_4)\end{aligned}$$

可以将前两个最大项改写为：

$$\begin{aligned}M_1\cdot M_3&=(x_1+x_2+\bar{x}_4+x_3)(x_1+x_2+\bar{x}_4+\bar{x}_3) && \text{用交换律10b}\\&=x_1+x_2+\bar{x}_4+x_3\bar{x}_3 && \text{用分配律12b}\\&=x_1+x_2+\bar{x}_4+0 && \text{用定理8a}\\&=x_1+x_2+\bar{x}_4 && \text{用定理6b}\end{aligned}$$

同样，$M_9\cdot M_{11}=\bar{x}_1+x_2+\bar{x}_4$。现在我们可以再次用$M_{11}$，根据定理7a，推导得到$M_{11}\cdot M_{13}=\bar{x}_1+x_3+\bar{x}_4$。因此

$$f=(x_1+x_2+\bar{x}_4)(\bar{x}_1+x_2+\bar{x}_4)(\bar{x}_1+x_3+\bar{x}_4)$$

再次应用定理12b，我们得到最后的解：

$$f=(x_2+\bar{x}_4)(\bar{x}_1+x_3+\bar{x}_4)$$

例2.10 一个控制给定数字系统的电路具有三个输入信号：x_1，x_2和x_3。该电路必须能够识别三个不同的条件：

1）若x_3为真，而且x_1为真或者x_2为假，则条件A为真。

2）若x_1为真，而且x_1为假或者x_2为假，则条件B为真。

3）若x_2为真，而且x_1为真或者x_3为假，则条件C为真。

若A，B和C三个条件中至少有两个为真，则该控制电路必定产生一个为1的输出。设计能实现以上功能的最简单的电路。

解：用1表示真，0表示假，可把上面三个条件用如下三个表达式表示：

$$\begin{aligned}A&=x_3(x_1+\bar{x}_2)=x_3x_1+x_3\bar{x}_2\\B&=x_1(\bar{x}_2+\bar{x}_3)=x_1\bar{x}_2+x_1\bar{x}_3\\C&=x_2(x_1+\bar{x}_3)=x_2x_1+x_2\bar{x}_3\end{aligned}$$

由此电路的预期输出可以用表达式 $f=AB+AC+BC$ 来表示。这些乘积项可以化简为：

$$\begin{aligned}AB&=(x_3x_1+x_3\bar{x}_2)(x_1\bar{x}_2+x_1\bar{x}_3)\\&=x_3x_1x_1\bar{x}_2+x_3x_1x_1\bar{x}_3+x_3\bar{x}_2x_1\bar{x}_2+x_3\bar{x}_2x_1\bar{x}_3\\&=x_3x_1\bar{x}_2+0+x_3\bar{x}_2x_1+0\\&=x_1\bar{x}_2x_3\\AC&=(x_3x_1+x_3\bar{x}_2)(x_2x_1+x_2\bar{x}_3)\\&=x_3x_1x_2x_1+x_3x_1x_2\bar{x}_3+x_3\bar{x}_2x_2x_1+x_3\bar{x}_2x_2\bar{x}_3\\&=x_3x_1x_2+0+0+0\\&=x_1x_2x_3\end{aligned}$$

$$
\begin{aligned}
BC &= (x_1\bar{x}_2 + x_1\bar{x}_3)(x_2x_1 + x_2\bar{x}_3) \\
&= x_1\bar{x}_2x_2x_1 + x_1\bar{x}_2x_2\bar{x}_3 + x_1\bar{x}_3x_2x_1 + x_1\bar{x}_3x_2\bar{x}_3 \\
&= 0 + 0 + x_1\bar{x}_3x_3 + x_1\bar{x}_3x_2 \\
&= x_1x_2\bar{x}_3
\end{aligned}
$$

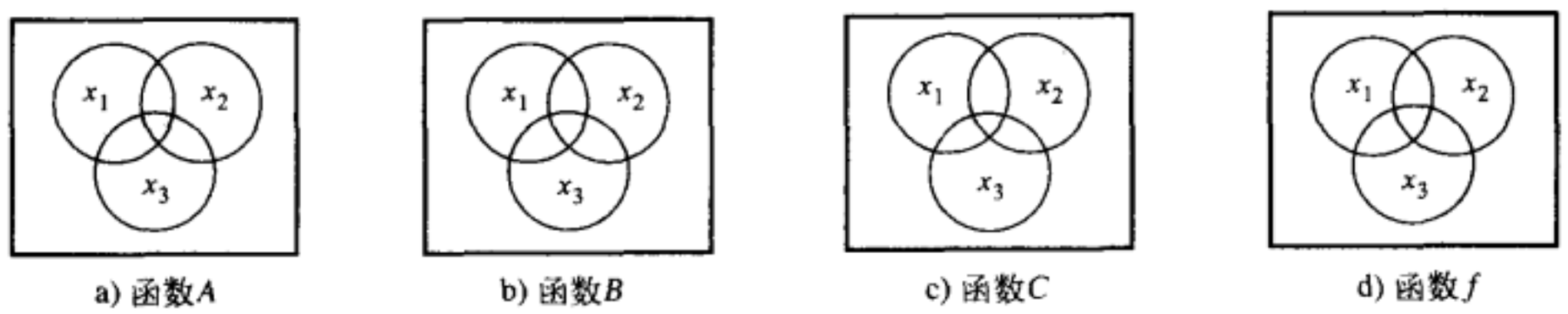
因此，f 可以记为

$$
\begin{aligned}
f &= x_1\bar{x}_2x_3 + x_1x_2x_3 + x_1x_2\bar{x}_3 \\
&= x_1(\bar{x}_2 + x_2)x_3 + x_1x_2(x_3 + \bar{x}_3) \\
&= x_1x_3 + x_1x_2 \\
&= x_1(x_3 + x_2)
\end{aligned}
$$

例2.11 用维恩图（Venn diagram）求解例2.10中的问题。

解：表示例2.10中的三个函数A，B和C的维恩图分别如图2-38a、b和c所示。由于当A，B和C中的两个或者两个以上为真时，函数 f 必须为真，所以表示 f 的维恩图可以通过找出表示A，B和C的维恩图（即各自的阴影区）的公共部分来确定。在这三个表示A，B和C的维恩图中，只要有两个或者两个以上的阴影重合，就是函数 f 的阴影区，如图2-38d所示。该维恩图对应的函数为：

$$f = x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$$



a) 函数A　b) 函数B　c) 函数C　d) 函数f

图2-38 例2.11的维恩图

例2.12 推导函数 $f = x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1\bar{x}_2x_4$的最简积之和表达式。

解：对函数 f 的前两项应用性质17a（见2.5节）得到如下表达式：

$$
\begin{aligned}
f &= x_2\bar{x}_3x_4 + x_1x_3x_4 + x_2x_4x_1x_4 + x_1\bar{x}_2x_4 \\
&= x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1x_2x_4 + x_1\bar{x}_2x_4
\end{aligned}
$$

然后，对最后两项应用性质14a（结合律）得到如下表达式：

$$f = x_2\bar{x}_3x_4 + x_1x_3x_4 + x_1x_4$$

最后，应用性质13a 得到如下表达式：

$$f = x_2\bar{x}_3x_4 + x_1x_4$$

例2.13 推导函数 $f = (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4)$ 的最简和之积表达式。

解：对函数 f 的前两项应用性质17a 得到如下表达式：

$$
\begin{aligned}
f &= (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_1 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4) \\
&= (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_4)(\bar{x}_1 + x_3 + \bar{x}_4)(\bar{x}_1 + x_3 + x_4)
\end{aligned}
$$

然后，对最后两项应用性质14a 得到如下表达式：

$$f=(\overline{x}_1+x_2+x_3)(\overline{x}_1+\overline{x}_2+\overline{x}_4)(\overline{x}_1+x_3)$$

最后，对第一项和最后一项应用性质13a 得到如下表达式：

$$f=(\overline{x}_1+\overline{x}_2+\overline{x}_4)(\overline{x}_1+x_3)$$

练习题

标有*号的习题答案参见书后。

2.1 用代数运算证明：$x+yz=(x+y)\cdot(x+z)$。请注意这是分配律，即前面2.5节定理12b。

2.2 用代数运算证明：$(x+y)\cdot(x+\overline{y})=x$。

2.3 用代数运算证明：$xy+yz+\overline{x}z=xy+\overline{x}z$。注意，这是2.5节的合意性质17a。

2.4 用维恩图证明2.1题的恒等式。

2.5 用维恩图证明德摩根定律，即前面2.5节中的表达式15a和15b。

2.6 用维恩图证明：$(x_1+x_2+x_3)\cdot(x_1+x_2+\overline{x}_3)=x_1+x_2$。

***2.7** 判断下列式子是否正确，即等式左边和右边的式子是否表示相同的函数（功能）。

(a) $\overline{x}_1x_3+x_1x_2\overline{x}_3+\overline{x}_1x_2+x_1\overline{x}_2=\overline{x}_2x_3+x_1\overline{x}_3+x_2\overline{x}_3+\overline{x}_1x_2x_3$

(b) $x_1\overline{x}_3+x_2x_3+\overline{x}_2\overline{x}_3=(x_1+\overline{x}_2+x_3)(x_1+x_2+\overline{x}_3)(\overline{x}_1+x_2+\overline{x}_3)$

(c) $(x_1+x_3)(\overline{x}_1+\overline{x}_2+\overline{x}_3)(\overline{x}_1+x_2)=(x_1+x_2)(x_2+x_3)(\overline{x}_1+\overline{x}_3)$

2.8 为图2-19a绘制时序图，显示电路中所有线上可以观察到的波形。

2.9 为图2-19b绘制时序图，显示电路中所有线上可以观察到的波形。

2.10 用代数运算证明：$\sum m(1,2,3,4,5,6,7)=x_1+x_2+x_3$，其中 x_1、x_2和x_3是三个输入变量。

2.11 用代数运算证明：$\prod M(0,1,2,3,4,5,6)=x_1x_2x_3$，其中$x_1$、$x_2$和$x_3$是三个输入变量。

***2.12** 用代数运算求函数：$f=x_1x_3+x_1\overline{x}_2+\overline{x}_1x_2x_3+\overline{x}_1\overline{x}_2\overline{x}_3$的最小积之和表达式。

2.13 用代数运算求函数：$f=x_1\overline{x}_2\overline{x}_3+x_1x_2x_4+x_1\overline{x}_2x_3\overline{x}_4$ 的最小积之和表达式。

2.14 用代数运算求函数：$f=(x_1+x_3+x_4)\cdot(x_1+\overline{x}_2+x_3)\cdot(x_1+\overline{x}_2+\overline{x}_3+x_4)$ 的最小和之积表达式。

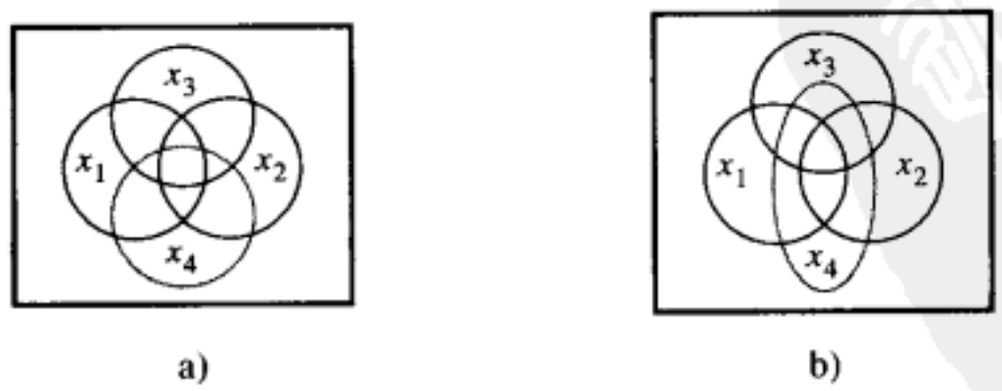
***2.15** 用代数运算求函数：$f=(x_1+x_2+x_3)\cdot(x_1+\overline{x}_2+x_3)\cdot(\overline{x}_1+\overline{x}_2+x_3)\cdot(x_1+x_2+\overline{x}_3)$ 的最小和之积表达式。

2.16 (a) 在一个3变量的维恩图中标出所有最小项的位置。

(b) 为函数 $f=x_1\overline{x}_2x_3+x_1x_2+\overline{x}_1x_3$中的每一个乘积项单独画出维恩图，并用维恩图找出该函数的最小积之和表达式。

2.17 用维恩图表示图2-18中的函数，并找出函数的最小积之和表达式。

2.18 图P2-1展示了两种不同的4变量维恩图的画法。请解释为什么图片的a)、b)两部分的维恩图是不正确的。（提示：维恩图必须能表示出4变量的所有16个最小项。）



图P2-1 两种不同的画4变量维恩图的尝试

2.19 图P2-2给出了4变量维恩图的表示法，并且标出了m_0，m_1，m_2三个最小项的位置，请给出维恩图中其他最小项的位置，并在图中表示函数：$f=\overline{x}_1\overline{x}_2x_3\overline{x}_4+x_1x_2x_3x_4+\overline{x}_1x_2$。

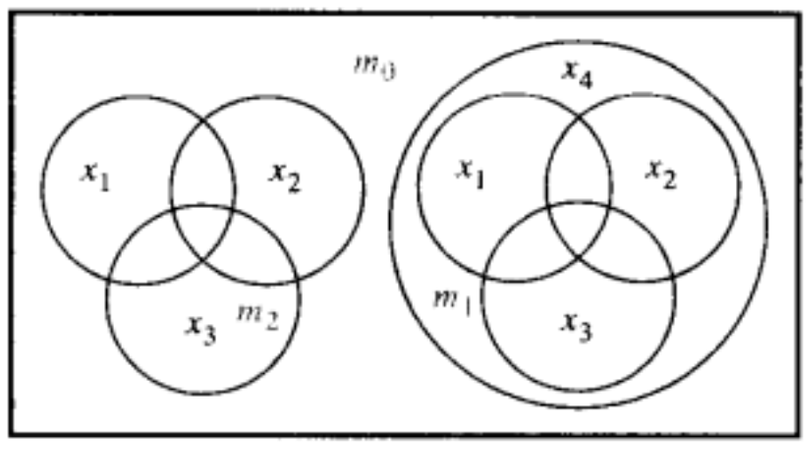
*2.20 设计能实现函数 $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$的最简积之和电路。

2.21 设计能实现函数 $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ 的最简积之和电路。

2.22 设计能实现函数 $f(x_1, x_2, x_3) = \prod M(0, 2, 5)$ 的最简和之积电路。

*2.23 设计能实现函数 $f(x_1, x_2, x_3) = \prod M(0, 1, 5, 7)$的最简和之积表达式。

2.24 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = x_1\overline{x}_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_1\overline{x}_2\overline{x}_3$，推导出它的最简积之和表达式。



图P2-2 4变量的维恩图

2.25 设有逻辑函数 $f(x_1, x_2, x_3, x_4, x_5) = \overline{x}_1\overline{x}_3\overline{x}_5 + \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_1x_4x_5 + x_1\overline{x}_2\overline{x}_3x_5$，推导出它的最简积之和表达式。（提示：利用性质17a。）

2.26 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)(\overline{x}_2 + \overline{x}_3 + x_4)(x_1 + \overline{x}_2 + \overline{x}_3)$，推导出它的最简和之积表达式。（提示：利用性质17b。）

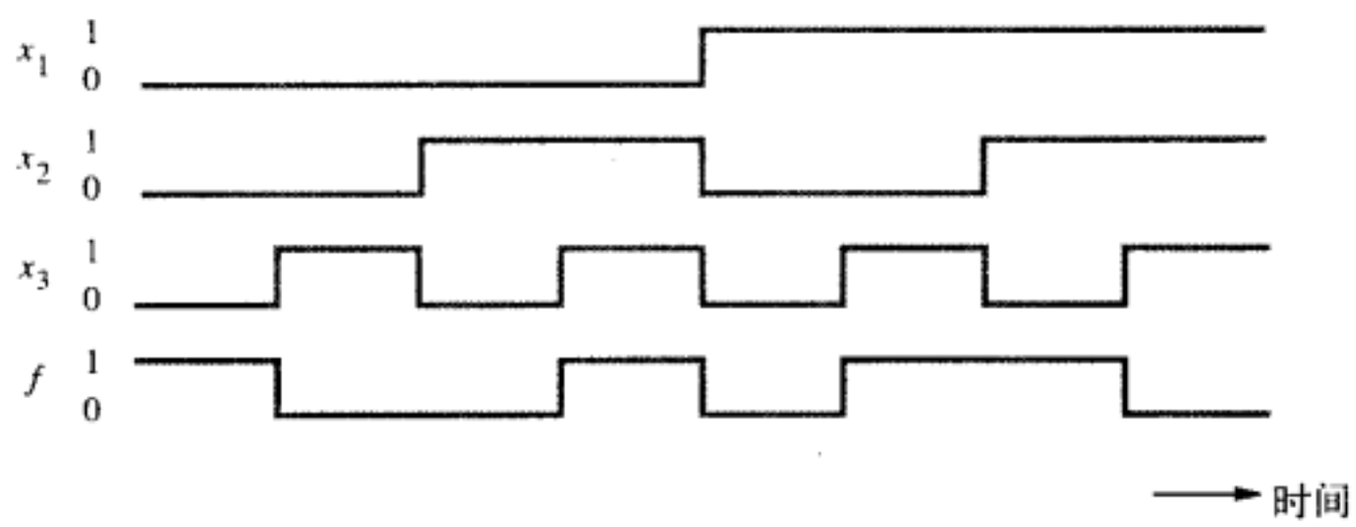
2.27 设有逻辑函数 $f(x_1, x_2, x_3, x_4, x_5) = (\overline{x}_2 + x_3 + x_5)(x_1 + \overline{x}_3 + x_5)(x_1 + x_2 + x_5)(x_1 + \overline{x}_4 + \overline{x}_5)$，推导出它的最简和之积表达式。（提示：利用性质17b。）

*2.28 设计最简单的电路，电路有x_1, x_2, x_3三个输入，若电路中两个或两个以上输入变量值为1，输出值为1；否则输出值为0。

2.29 设计一个最简单的电路，电路有x_1，x_2，x_3三个输入，若电路中有一个或两个输入值为1，输出值为1；否则输出值为0。

2.30 设计一个最简单的电路，电路有x_1，x_2，x_3和x_4四个输入，若电路中有三个或三个以上输入值为1，输出值为1；否则输出值为0。

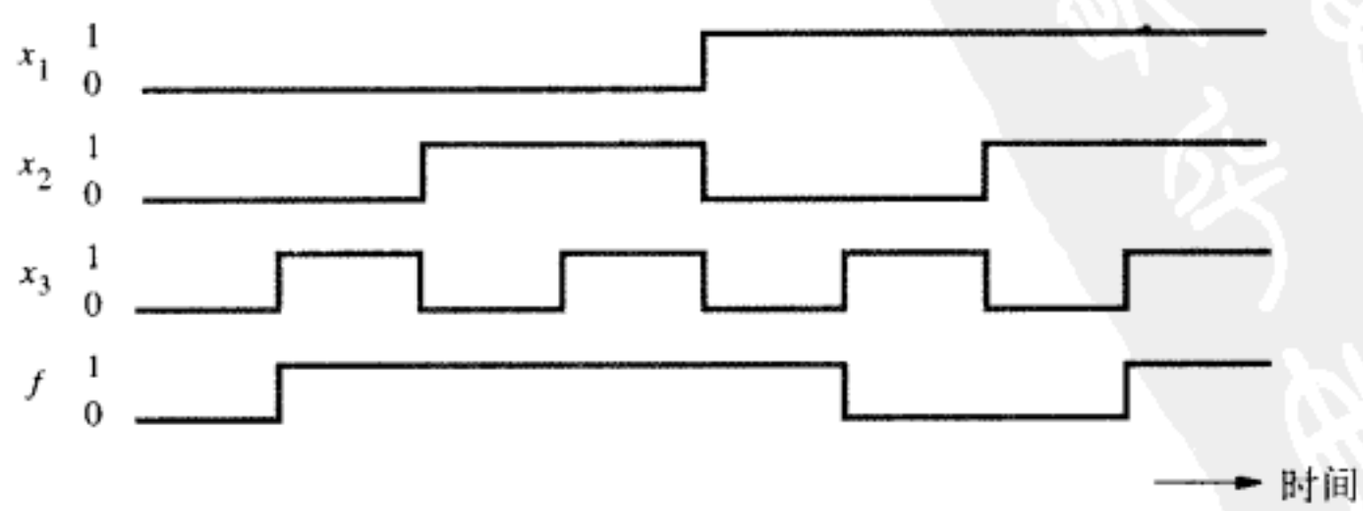
2.31 对于图P2-3的时序图，综合出最简积之和形式表示的函数 $f(x_1, x_2, x_3)$。



图P2-3 用时序图表示的逻辑函数

*2.32 对于图P2-3的时序图，综合出最简和之积形式表示的函数 $f(x_1, x_2, x_3)$。

*2.33 对于图P2-4的时序图，综合出最简积之和形式表示的函数 $f(x_1, x_2, x_3)$。



图P2-4 用时序图表示的逻辑函数

2.34 对于图P2-4的时序图，综合出最简和之积形式表示的函数 $f(x_1, x_2, x_3)$。

2.35 设计一个电路，该电路的输入为x_1，x_0，y_1，y_0，输出为 f。令$X = x_1x_0$ 为一个有4种可能取值的数字，即00，01，10和11，分别表示0，1，2，3四个数（我们将在第5章讨论数字的表示法）。同样，$Y = y_1y_0$ 表示另外一个有4种可能的取值的数字。若X和Y 表示的数字相同，则输出 f 值为1。否则 f 值为0。

(a) 写出 f 的真值表。

(b) 综合出表示 f 的最简单的和之积表达式。

2.36 若条件变为只有当$X \geqslant Y$ 时 f 的值为1，重复做2.35题。

(a) 写出 f 的真值表。

(b) 写出表示 f 的正则积之和表达式。

(c) 写出表示 f 的最简积之和表达式。

2.37 只用与非门实现图2-26所示的函数。

2.38 只用或非门实现图2-26所示的函数。

2.39 用与非门和或非门实现图2-33所示的电路。

***2.40** 设计能实现函数：$f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$的最简电路（要求只用与非门实现）。

2.41 设计能实现函数：$f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$的最简电路（要求只用与非门实现）。

***2.42** 要求只用或非门，重做2.40题。

2.43 要求只用或非门，重做2.41题。

2.44 (a) 用原理图获取工具绘制能实现下列逻辑函数的原理图：

$$f_1 = x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_4 + \bar{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\bar{x}_4 + \bar{x}_1x_2 + x_2x_3$$

(b) 用功能仿真证明：$f_1 = f_2$。

2.45 (a) 用原理图获取工具绘制能实现下列逻辑函数的原理图：

$$f_1 = (x_1 + x_2 + \bar{x}_4)\cdot(\bar{x}_2 + x_3 + \bar{x}_4)\cdot(\bar{x}_1 + x_3 + \bar{x}_4)\cdot(\bar{x}_1 + \bar{x}_3 + \bar{x}_4)$$
$$f_2 = (x_2 + \bar{x}_4)\cdot(x_3 + \bar{x}_4)\cdot(\bar{x}_1 + \bar{x}_4)$$

(b) 用功能仿真证明：$f_1 = f_2$。

2.46 编写能实现图2-27a所示电路的Verilog代码，要求用门级原语言编写。

2.47 编写能实现图2-27b所示电路的Verilog代码，要求用门级原语言编写。

2.48 编写能实现逻辑函数：$f(x_1, x_2, x_3) = \sum m(1, 2, 3, 4, 5, 6)$的 Verilog代码，要求用门级原语言编写，并确保设计的电路尽可能地简单。

2.49 编写能实现逻辑函数：$f(x_1, x_2, x_3) = \sum m(0, 1, 3, 4, 5, 6)$的 Verilog代码，要求用连续赋值语句编写。

2.50 (a) 编写描述下列逻辑函数的Verilog代码：

$$f_1 = x_1\bar{x}_3 + x_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2 + x_1\bar{x}_4$$
$$f_2 = (x_1 + \bar{x}_3)\cdot(x_1 + x_2 + \bar{x}_4)\cdot(x_2 + \bar{x}_3 + \bar{x}_4)$$

(b) 用功能仿真证明：$f_1 = f_2$。

2.51 考虑下列的Verilog语句：

```
f1 = (x1 & x3) | (~x1 &~x3) | (x2 & x4) | (~x2 & ~x4);
f2 = (x1 & x2 & ~x3 & ~x4) | (~x1 &~x2 & x3 & x4) |
     (x1 & ~x2 & ~x3 & x4) | (~x1 & x2 & x3 & ~x4).
```

(a) 分别写出能实现上述函数的完整的Verilog代码。

(b) 用功能仿真证明：$f1 = \overline{f2}$。

参考文献

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by Dover Publications, New York, 1954.
2. C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions of AIEE* 57 (1938), pp. 713–723.
3. E. V. Huntington, "Sets of Independent Postulates for the Algebra of Logic," *Transactions of the American Mathematical Society* 5 (1904), pp. 288–309.
4. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 2nd ed. (McGraw-Hill: New York, 2005).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
6. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
7. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
8. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice Hall: Upper Saddle River, NJ, 2000).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

第3章 实现技术

在1.2节中我们曾经提到过，逻辑电路可以用晶体管实现，而且存在许多种实现技术。在本节中，我们将更详细地探讨逻辑电路的实现技术问题。

首先，让我们思考如何用电子线路中信号的物理量来表示逻辑变量。我们的讨论将限定在二进制变量上，即取值只允许为1和0。在电路中，这些信号既可以用电路中的电平表示，也可以用电流的强度表示，究竟用哪一种表示方法，取决于不同的技术。本书中采用最简单和最流行的表示方法，即用电平来表示逻辑变量的值。

用电平表示逻辑值的最直接的方法是定义阈值，任何低于该阈值的电平代表逻辑0；而高于该阈值的电平代表逻辑1。也可以采用另外一种选择，即任何低于该阈值的电平代表逻辑1；而高于该阈值的电平代表逻辑0。前者称为*正逻辑系统*（positive logic system）；后者称为*负逻辑系统*（negative logic system），可以在这两种系统中任择其一。本书中只使用正逻辑系统，只在3.4节中对负逻辑作简要的介绍。

在正逻辑系统中，可以简单地把逻辑值0看作“低”；而把逻辑值1看作“高”。为了实现阈值的概念，定义了高电平和低电平的范围如图3-1所示。图中，最低的电位叫做Vss，最高的电位叫做V_{DD}，它们是电路中实际存在的电压。我们进一步假设Vss是0 V，即电路中的地，记作Gnd；而V_{DD}代表电源电压。电源电压V_{DD}的值通常在5 V和1.5 V之间，本章中主要使用V_{DD} = 5 V。如图3-1所示，电平值在Gnd和$V_{0,max}$之间时表示逻辑0，符号$V_{0,max}$代表逻辑0的最高电平，小于它的电平被视作“低”。与此相似，在$V_{1,min}$和V_{DD}之间的电平值代表逻辑1，符号$V_{1,min}$代表逻辑1的最低电平，即高于它的电平被视作“高”。$V_{0,max}$和$V_{1,min}$的精确值取决于所采用的技术，其典型值是：$V_{0,max}$是V_{DD}的40%，而$V_{1,min}$是V_{DD}的60%。电平值在$V_{0,max}$和$V_{1,min}$之间的逻辑值没有定义，逻辑电路中的信号一般不处在这个电平范围，只有在信号由一个状态向另一个状态转换时才经过这个范围。我们将在3.8.3节深入讨论逻辑电路中的电平问题。

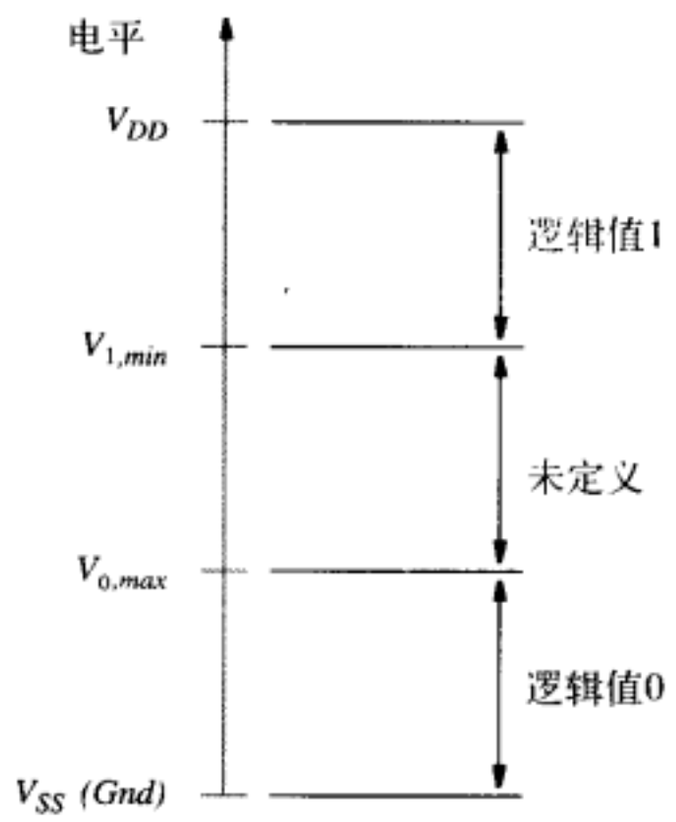
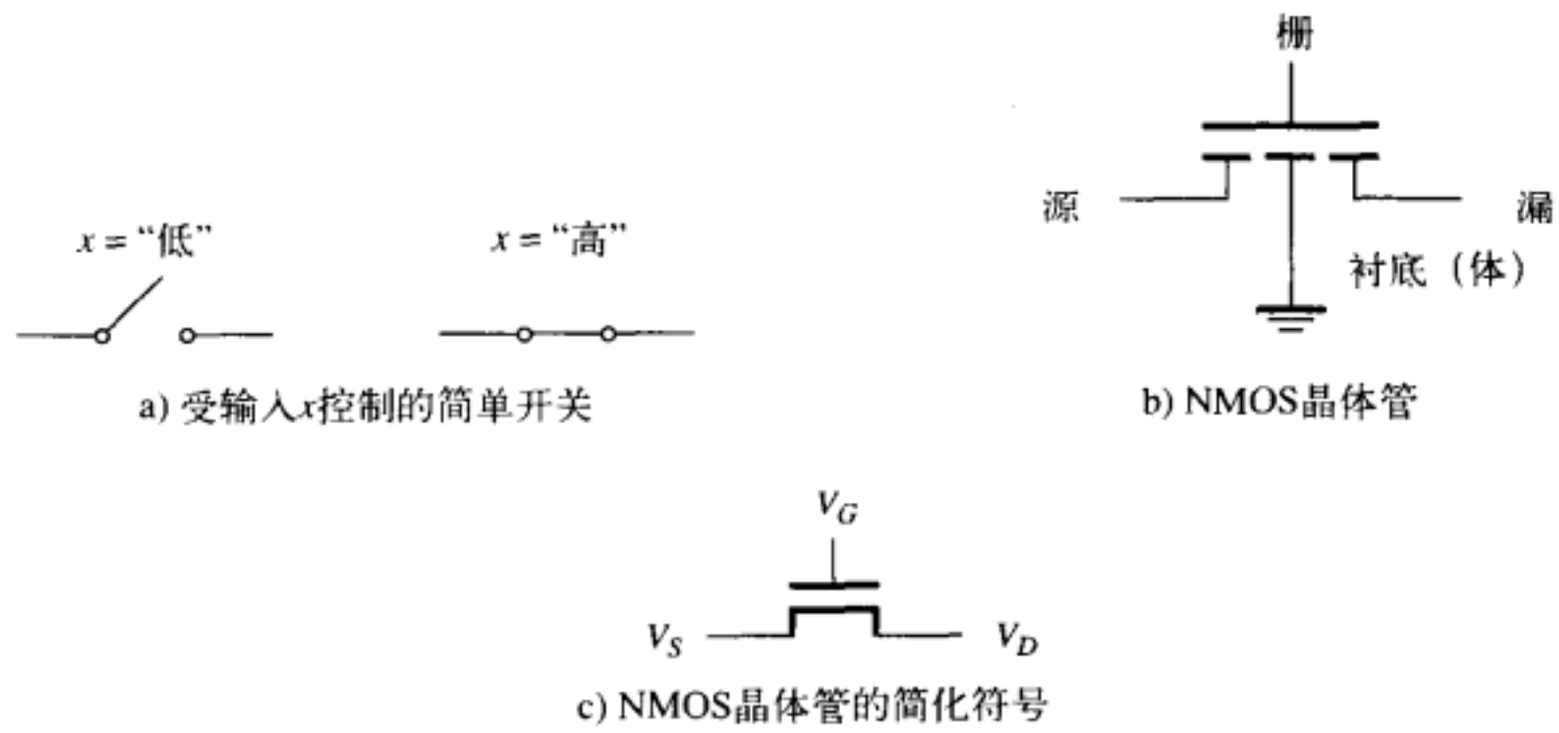


图3-1 用电平表示的逻辑值

3.1 晶体管开关

逻辑电路由晶体管组成。全面讨论晶体管的行为特性超出了本书的范围，感兴趣的读者可参阅电子类教科书，例如参考文献[1]和[2]。为了理解逻辑电路的组成原理，我们可以把晶体管看作一个简单的开关。如图3-2a所示，一个受逻辑信号x控制的开关，当信号x的值为低时开关断开，而x的值为高时开关接通。最流行的用于实现简单开关的晶体管是*金属氧化物半导体场效应晶体管*（MOSFET）。MOSFET有两种类型：*N沟道晶体管*（NMOS）和*P沟道晶体管*（PMOS）。



a) 受输入x控制的简单开关

b) NMOS晶体管

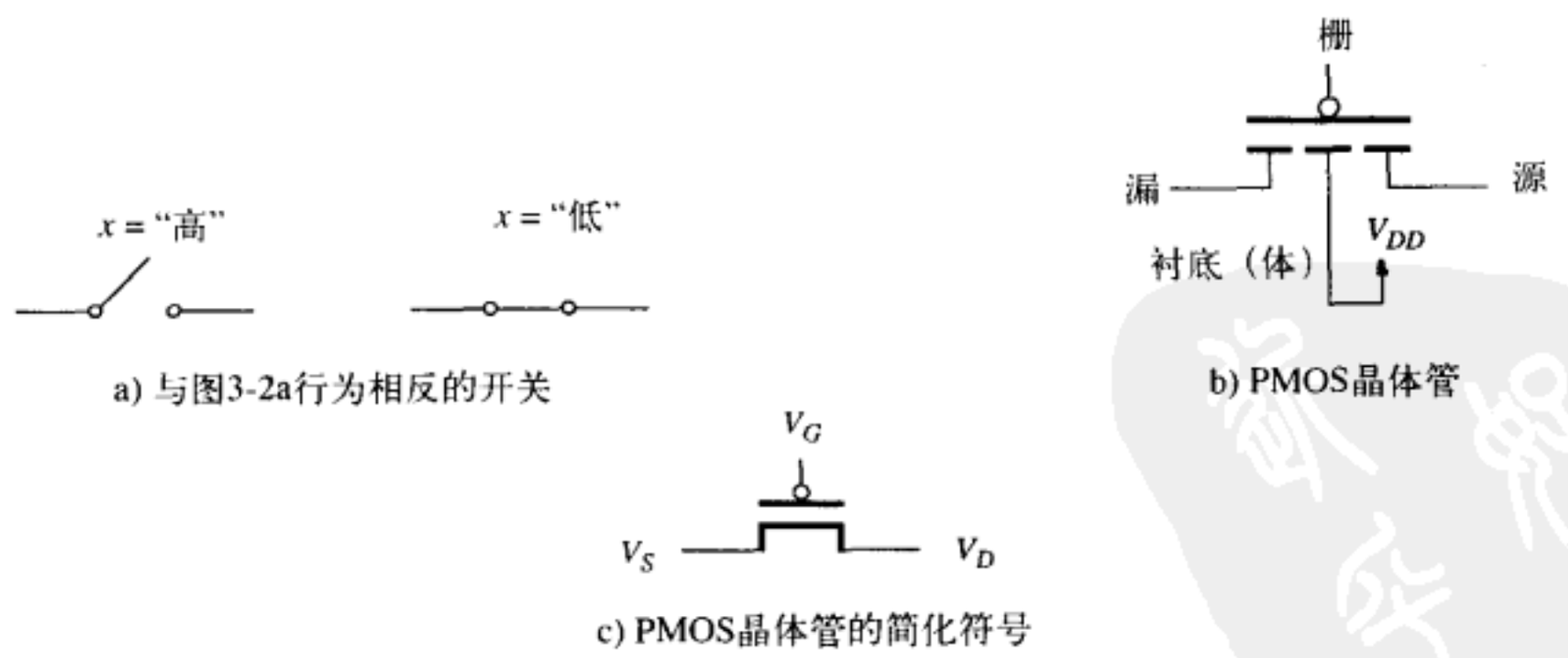
c) NMOS晶体管的简化符号

图3-2 把NMOS晶体管用做开关

图3-2b所示为NMOS晶体管的图形符号，它有4个接线端，分别称为源、漏、栅和衬底。在逻辑电路中，衬底（也称为体）端和地相连。图3-2c显示了NMOS晶体管的简化图形符号，它把衬底端省略了。源极和漏极在物理上没有什么区别，通常根据加在晶体管上的电平来区分源极和漏极，电平较低的一端被认为是源极。

我们将在3.8.1节中详细解释晶体管的工作原理，现在认为晶体管受栅极电平V_G的控制就足够了。如果V_G是低电平，则NMOS晶体管的源极和漏极之间没有连接，称该晶体管处于断开状态；如果V_G是高电平，则NMOS晶体管处于接通状态，相当于一个接通的开关，它的源极和漏极相互连接。我们将在3.8.2节中介绍如何计算晶体管导通时源极与漏极之间的电阻，现在先假设该电阻值为零。

PMOS晶体管的行为特性和NMOS晶体管相反。PMOS晶体管可被用来实现图3-3a所示的开关：当控制信号x低电平时开关导通；而x是高电平时开关断开。PMOS晶体管的图形符号如图3-3b所示。在逻辑电路中，PMOS的衬底永远与V_{DD}相连，由此导出如图3-3c所示的简化符号。当V_G是高电平时，PMOS晶体管不导通，相当于一个断开的开关；当V_G是低电平时，晶体管导通，相当于一个接通的开关，把源极和漏极连接起来。在PMOS晶体管中，源极是电平较高的节点。



a) 与图3-2a行为相反的开关

b) PMOS晶体管

c) PMOS晶体管的简化符号

图3-3 把PMOS晶体管用做开关

图3-4总结了NMOS和PMOS晶体管在逻辑电路中的典型用法。当NMOS晶体管的栅极高电平时，该晶体管导通；而当PMOS晶体管的栅极低电平时，该晶体管导通。当NMOS晶体管导通时，它的漏极被下拉到Gnd；当PMOS晶体管导通时，它的漏极被上拉到V_{DD}。由于晶体

管的工作原理，NMOS不能把晶体管的漏极电平完全上拉到V_{DD}。与此相似，PMOS晶体管的漏极电平也不能完全下拉到*Gnd*。我们将在3.8节中详细讨论金属氧化物半导体场效应晶体管（MOSFET）的工作原理。

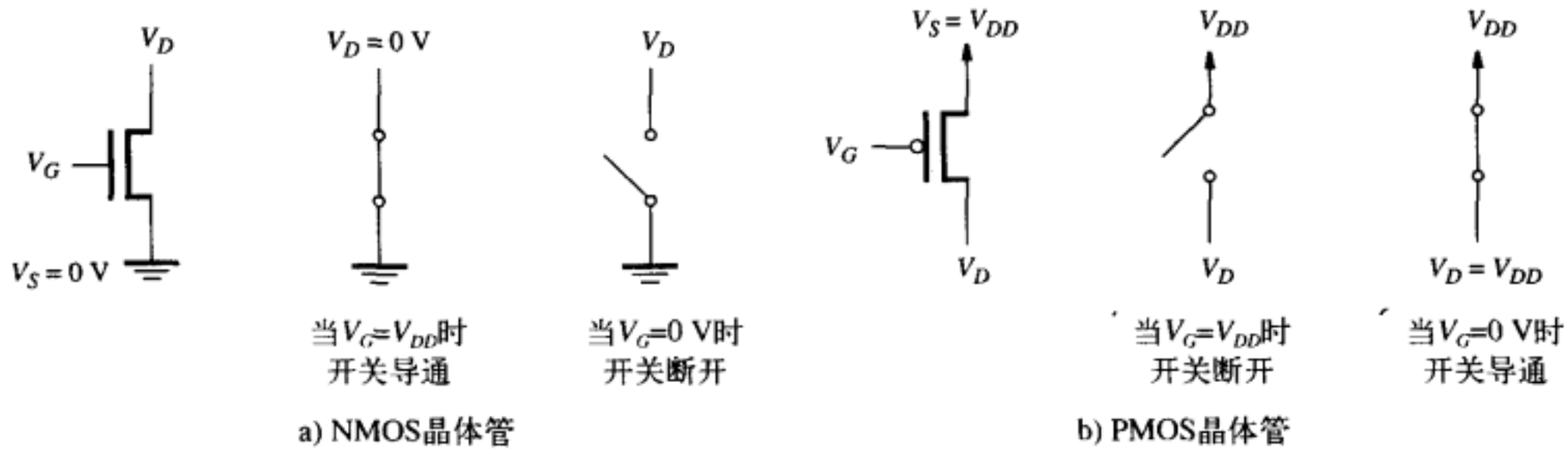
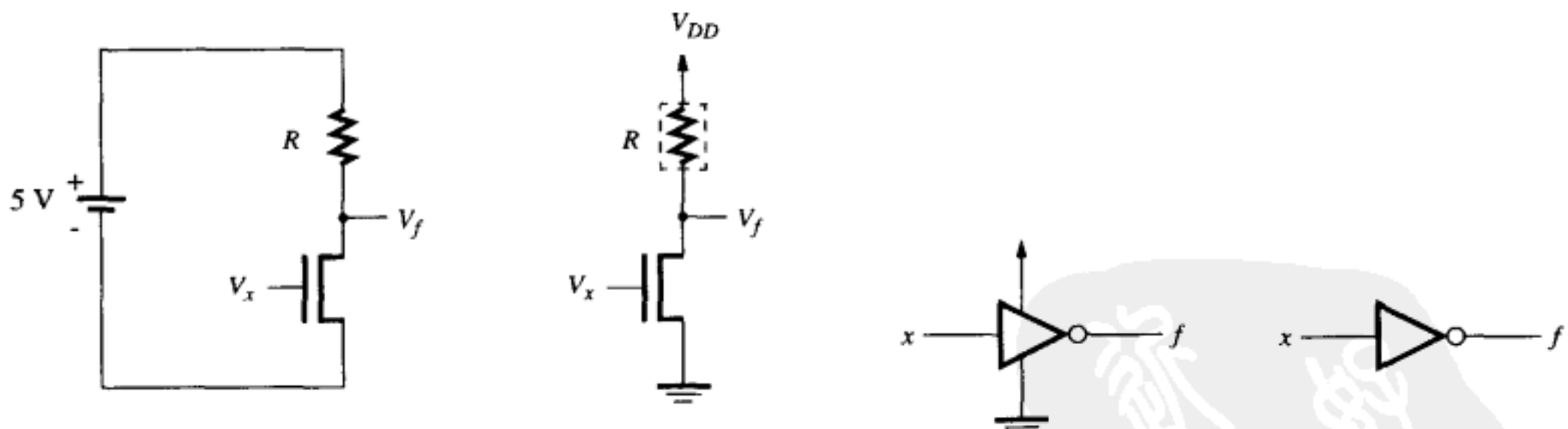


图3-4 逻辑电路中的NMOS 和PMOS晶体管

3.2 NMOS逻辑门

在20世纪70年代，用MOSFET构造逻辑门电路的方法开始流行。其实现方法是：单独用PMOS晶体管或者用NMOS晶体管，而两种晶体管不同时使用。从20世纪80年代初期开始，把PMOS和NMOS晶体管组合起来使用。为了易于理解，我们将先描述怎样用NMOS晶体管组成逻辑电路，这种电路被称为NMOS电路。然后描述怎样把NMOS和PMOS晶体管组合起来构成逻辑电路，就是当前流行的互补MOS或CMOS技术。

在图3-5a所示的电路中，当$V_x = 0$ V时，NMOS晶体管断开，没有电流流过电阻R，于是$V_f = 5$ V；当$V_x = 5$ V时，NMOS晶体管导通，节点V_f被下拉到低电平，电平V_f的精确值取决于流过电阻R和晶体管的电流，其典型值是0.2 V左右（参见3.8.3节）。如果把V_f看作V_x的函数，则该电路可看作非门（NOT）的NMOS实现，对应的布尔表达式为：$f = \overline{x}$。图3-5b是图3-5a电路图的简化形式：箭头上边标的V_{DD}表示连接到电源正极，*Gnd* 符号表示连接到电源的负极。本章后面的部分将采用这种简化形式的电路图。



a) 电路图　　b) 简化的电路图　　c) 图形符号

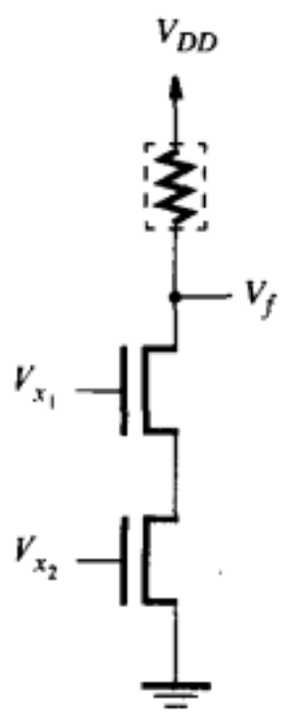
图3-5 由NMOS技术建造的非门

非门中电阻R的作用是：当$V_x = 5$ V时，用来限制流过晶体管的电流。用晶体管代替电阻R也可达到同样的目的，在3.8.3节中我们将会更详细地讨论这个问题。在图3-5b中，电阻R的外面加一个虚线框，表示这个电阻R是用晶体管实现的。

图3-5c表示的是非门的图形符号。左图展示输入、输出、电源和地端，右图只简单地展

示输入和输出端。实际应用中仅使用这种简化的符号。非门又叫做反相器（inverter），书中我们两个名字都使用。

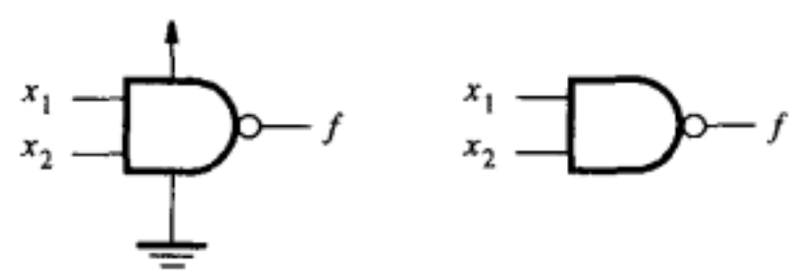
在2.1节中我们看到，串联的开关对应与门的逻辑功能，并联的开关对应或门的逻辑功能。图3-6a显示了两个NMOS晶体管串联形成的电路，当$V_{x1} = V_{x2} = 5$ V时，两个晶体管都导通，V_f将接近0 V；但是如果V_{x1}或V_{x2}是0 V时，则不会有电流流过串联的晶体管，V_f的电平将上拉至5 V。该电路的逻辑关系真值表形式如图3-6b所示。其实现功能是与门逻辑功能的反相，即与非逻辑，记作NAND。此电路实现了与非门的逻辑功能，图3-6c是其图形符号。



a) 电路

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

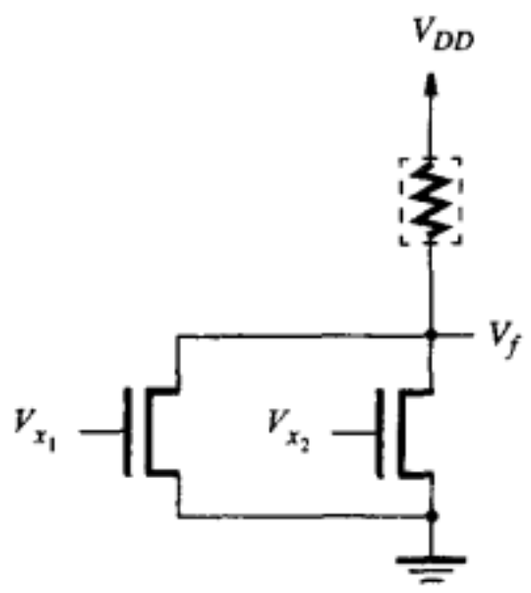
b) 真值表



c) 图形符号

图3-6 由NMOS技术实现的与非门

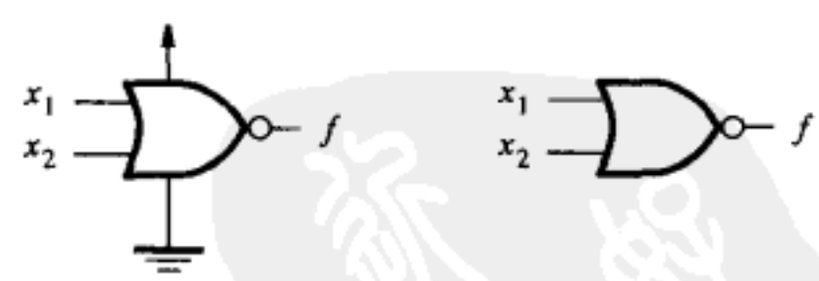
图3-7a展示了由两个NMOS晶体管并联形成的电路。在这个电路中，如果$V_{x1} = 5$ V或$V_{x2} = 5$ V，V_f的电平将接近0 V；仅当$V_{x1} = V_{x2} = 0$ V时，V_f的电平被上拉至5 V。该电路对应的真值表形式如图3-7b所示。其实现功能是或门的非，即或非，记作NOR。此电路实现了或非门的逻辑功能，图3-7c是其图形符号。



a) 电路

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

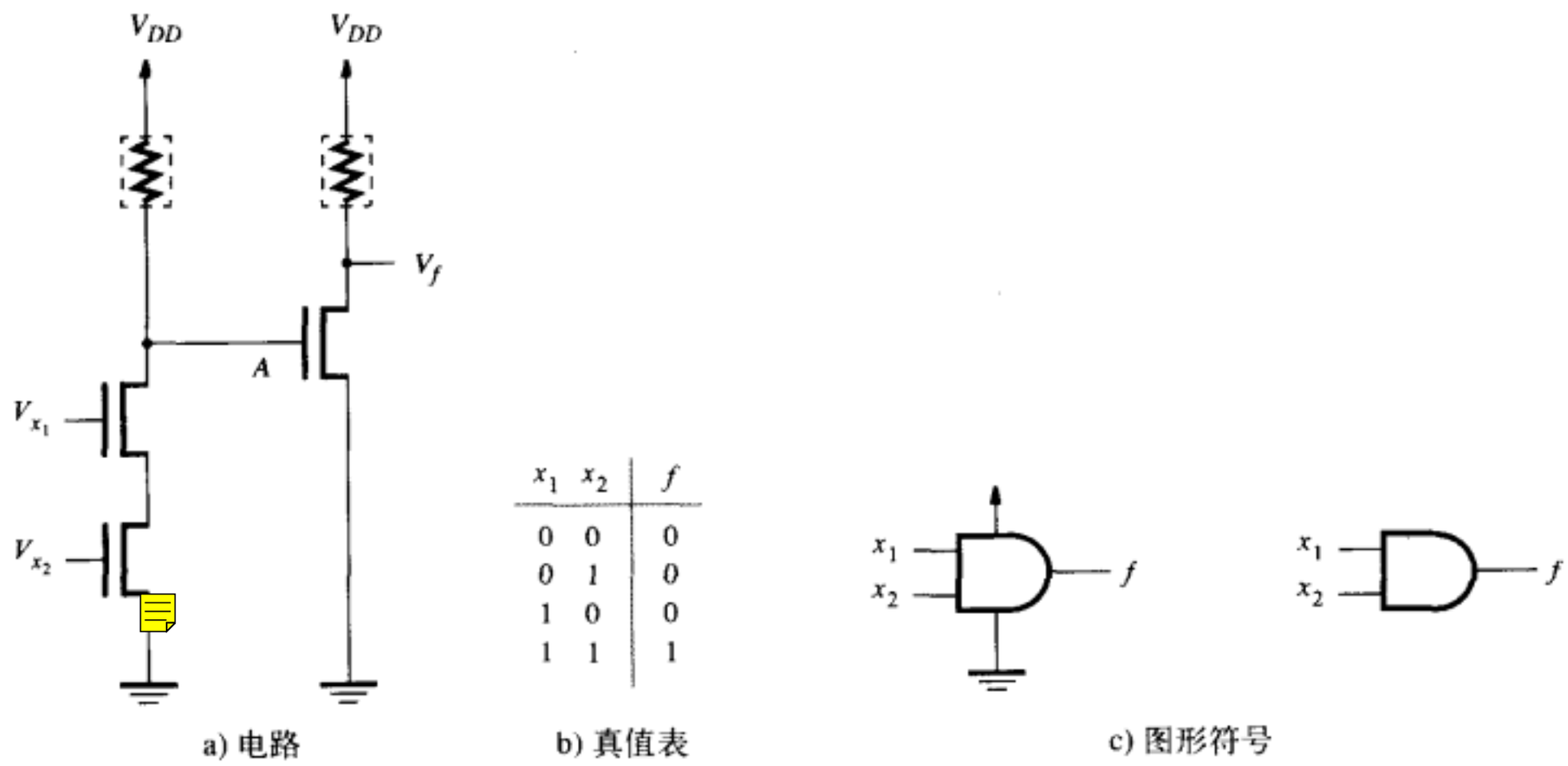
b) 真值表



c) 图形符号

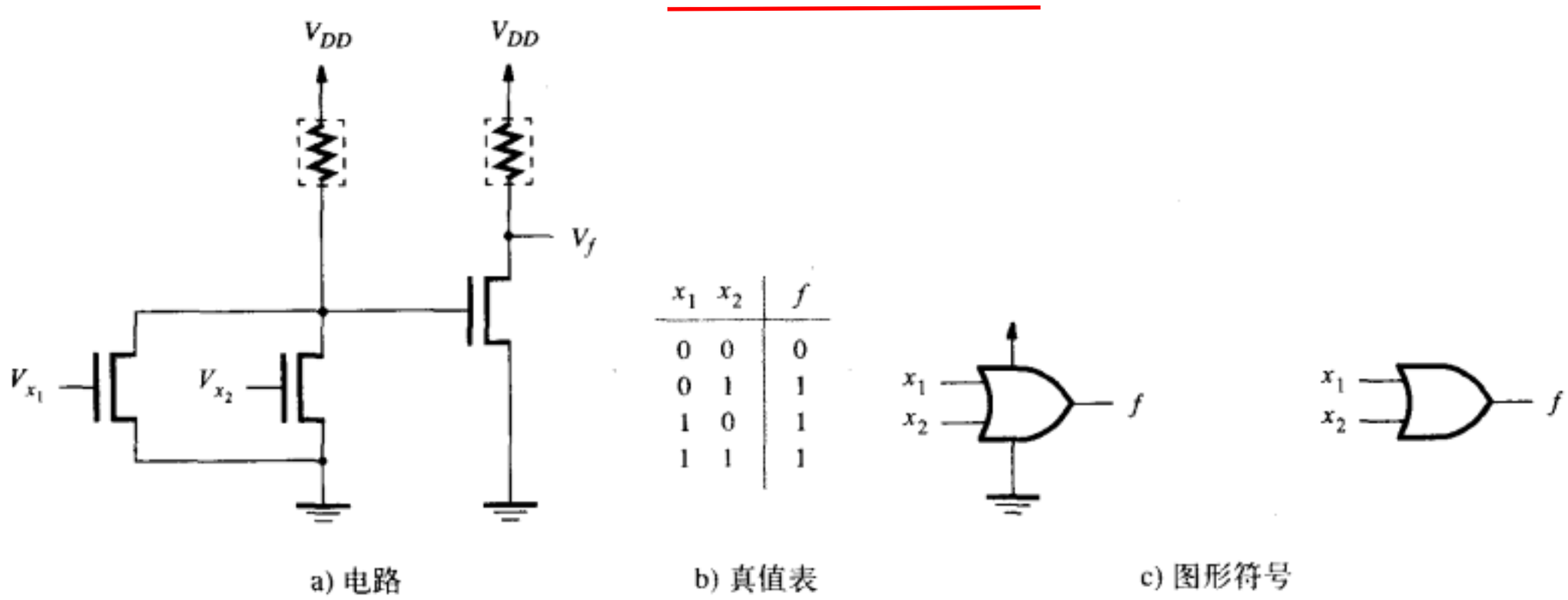
图3-7 由NMOS技术实现的或非门

除了刚介绍过的与非门和或非门外，读者可能更关心在前几章中广泛使用的与门和或门的实现方法。图3-8表明采用NMOS技术的与门是如何实现的：在与非门的后面连接一个反相器就实现了与门。就其逻辑关系而言，节点A实现输入x_1和x_2的与非功能，而f则代表x_1和x_2相与（AND）的功能。同样，在或非门后面连接反相器就实现了或门的逻辑功能，如图3-9所示。



a) 电路　　b) 真值表　　c) 图形符号

图3-8　由NMOS技术实现的与门



a) 电路　　b) 真值表　　c) 图形符号

图3-9　由NMOS技术实现的或门

3.3　CMOS逻辑门

到目前为止，我们已经知道了如何使用NMOS晶体管实现逻辑门。对于前面所介绍的每一种电路，都可以使用PMOS晶体管推导出对应的等价电路。但是，更有趣的是考虑同时使用NMOS和PMOS晶体管所组成的逻辑电路，这就是当前最流行的CMOS技术。在3.8节中我们将介绍CMOS技术与NMOS技术相比较，它有着一些更有吸引力的实际优点。

在NMOS电路中，逻辑功能是依靠NMOS晶体管和一个上拉电阻器件实现的。我们称电路中包含NMOS晶体管的部分为下拉网络（pull-down network，PDN）。这样，图3-5到图3-9所示电路结构都可以用图3-10来表示。CMOS电路的基本概念是：上拉器件用上拉网络（pull-up network，PUN）代替，用PMOS晶体管实现上拉网络PUN。逻辑电路的功能由PDN和PUN互补共同实现。这种逻辑电路，例如典型逻辑门的结构如图3-11所示。对于输入信号的任一给定值，输出节点V_f或者被PDN下拉至Gnd，或者被PUN上拉到V_{DD}。PDN和PUN晶体管数目相同，而且两个网络是成对安排的。如果PDN中NMOS的晶体管是串行连接的，则PUN中PMOS的晶体管是并行连接的，反之亦然。

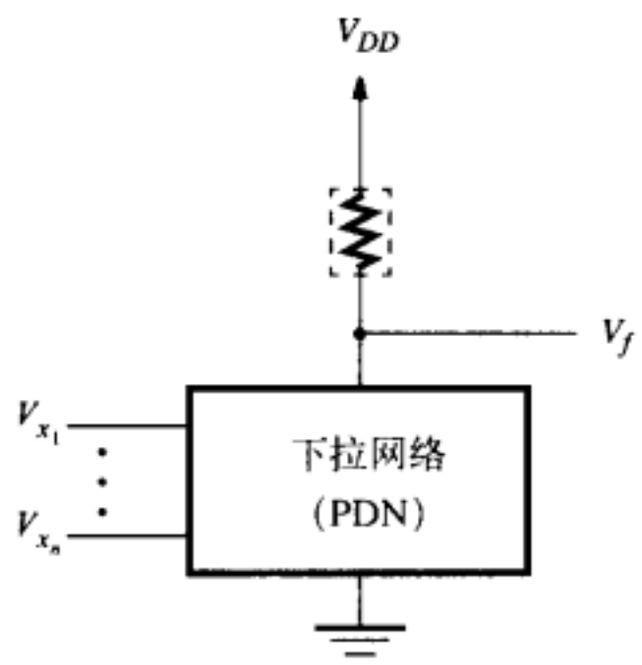


图3-10 NMOS电路的结构

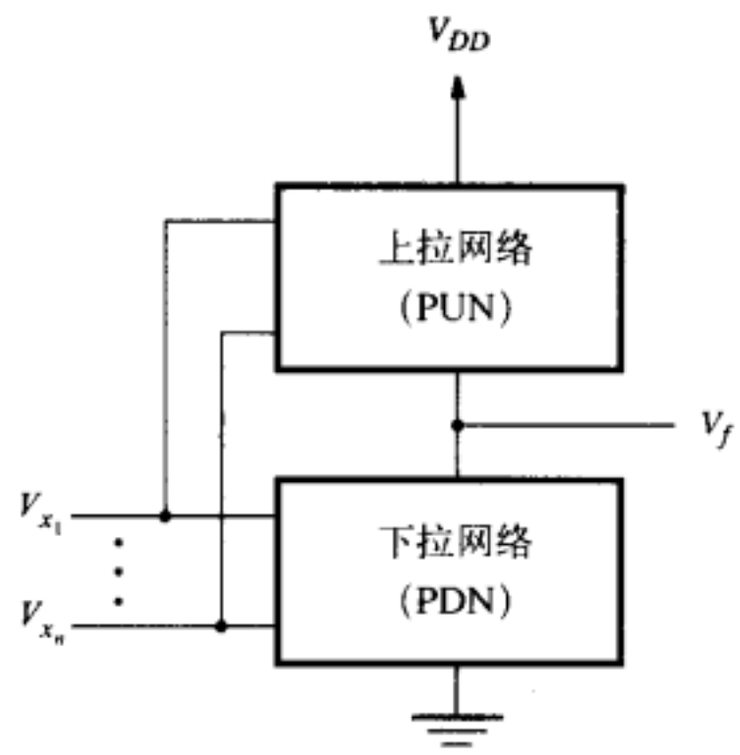
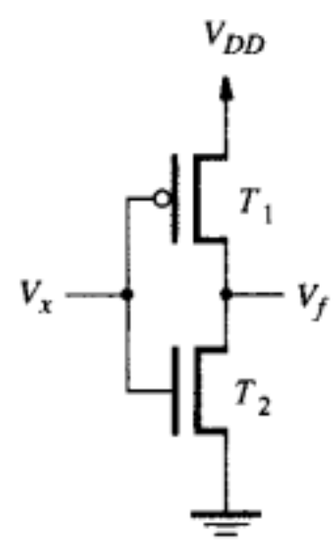


图3-11 CMOS电路的结构

CMOS电路的最简单例子是非门NOT，如图3-12所示。当$V_x = 0$ V时，晶体管T_2断开（off），而晶体管T_1导通（on），于是$V_f = 5$ V。由于晶体管T_2断开，晶体管中没有电流流过。当$V_x = 5$ V时，晶体管T_1断开而晶体管T_2导通，于是$V_f = 0$ V。由于晶体管T_1断开，晶体管中没有电流流过。

CMOS非门最关键的优点是，无论输入电平是低还是高，电路中都没有电流流过。对于所有CMOS电路来说，都存在这一优点，因而CMOS电路在静态下没有功率消耗。这一特性使得CMOS成为当前最流行的实现逻辑电路的技术。在3.8节中将详细地讨论CMOS电路中的电流和功率消耗问题。



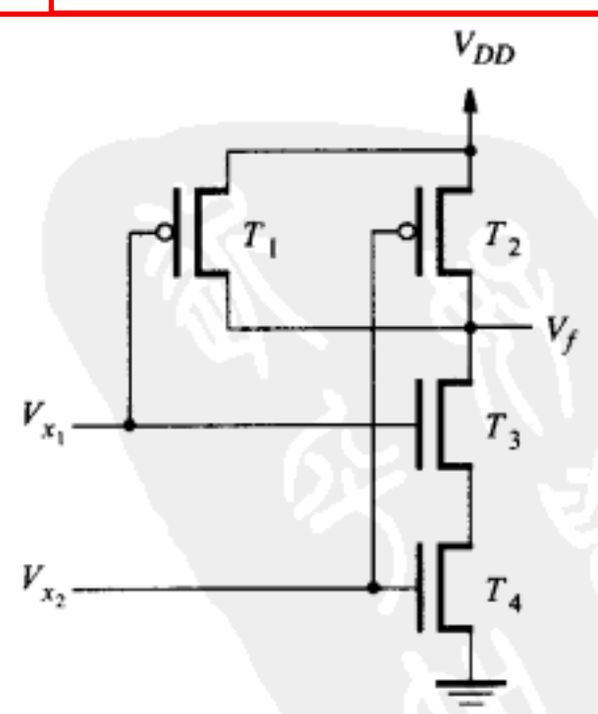
a) 电路

x	T_1	T_2	f
0	on	off	1
1	off	on	0

b) 真值表和晶体管状态

图3-12 由CMOS实现的非门

图3-13提供了一个用CMOS工艺实现的与非门电路图。它和图3-6所示的用NMOS工艺实现的与非门电路类似，只是把上拉器件换成了由两个PMOS晶体管并联而成的上拉网络(PUN)。图3-13b是该与非门的真值表，真值表描述了每个输入的x_1、x_2值下，四个晶体管每个各自的开关状态和该与非门的输出值。图中的真值表规定了该电路的每个状态，恰好实现了与非的逻辑功能。而在静态条件下，不存在从V_{DD}到地的电流路径。



a) 电路

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

b) 真值表和晶体管状态

图3-13 由CMOS实现的与非门

图3-13a所示的电路可由定义与非操作的逻辑表达式导出，该表达式为：$f=\overline{x_1x_2}$。此表达式定义了 $f=1$的条件；因此定义了上拉网络（PUN）。由于上拉网络由PMOS晶体管组成，当控制输入为0时，PMOS晶体管导通，如果一个输入变量$x_i=0$，晶体管导通。根据德摩根定律，有以下逻辑表达式：

$$f=\overline{x_1x_2}=\bar{x}_1+\bar{x}_2$$

只要x_1或x_2中有一个值为0，输出 $f=1$。也就是说，上拉网络（PUN）必须由两个PMOS晶体管并联而成。下拉网络（PDN）实现的逻辑应当是f的非，即：

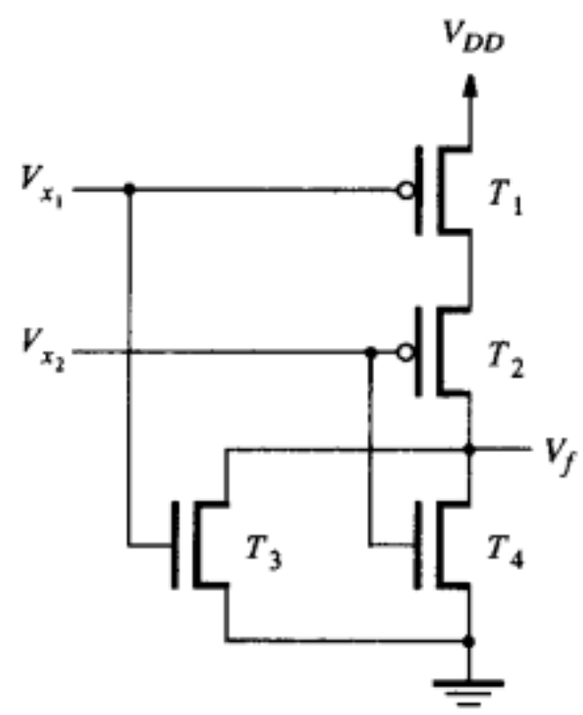
$$\bar{f}=x_1x_2$$

只有当x_1和x_2的值同时为1时，才能使 $\bar{f}=1$，因而下拉网络必须由两个串联的NMOS晶体管组成。

CMOS或非门电路可由以下定义或非的逻辑表达式导出：

$$f=\overline{x_1+x_2}=\bar{x}_1\bar{x}_2$$

只有当$x_1=x_2=0$时，才有输出 $f=1$，所以上拉网络由两个PMOS晶体管串联而成。实现$\bar{f}=x_1+x_2$逻辑功能的下拉网络由两个并联的NMOS晶体管组成，由此而导出的电路如图3-14所示。



a) 电路

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

b) 真值表和晶体管状态

图3-14 由CMOS实现的或非门

CMOS与门由与非门连接一个反向器组成，如图3-15所示。同样，或门由或非门连接一个反向器组成。

上述推导出CMOS逻辑电路的步骤可以用于更一般化的逻辑函数，推导出更复杂的门电路，下面两个例子详细说明了这一过程。

例3.1 设逻辑函数为：

$$f=\bar{x}_1+\bar{x}_2\bar{x}_3$$

逻辑表达式中所有变量都以反变量的形式出现，所以我们可以直接推导出上拉网络（PUN），它包含电路中由x_1控制的一个并联的PMOS晶体管和电路中分别由x_2、x_3控制的两个串联的PMOS晶体管。而对下拉网络（PDN），

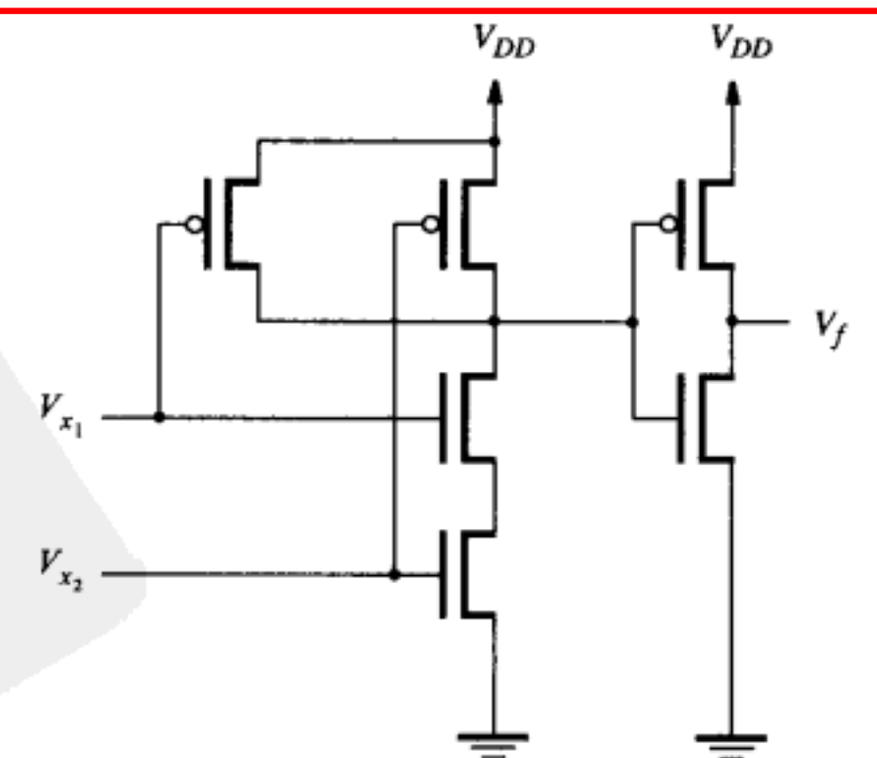


图3-15 由CMOS实现的与门

逻辑表达式为：

$$\overline{f} = \overline{\overline{x}_1 + \overline{x}_2\overline{x}_3} = x_1(x_2 + x_3)$$

该表达式显示了下拉网络，它包含由x_1控制的串联的一个NMOS晶体管和分别由x_2、x_3控制的两个并联NMOS晶体管，如图3-16所示。 ■

例3.2 设逻辑函数为：

$$f = \overline{x}_1 + (\overline{x}_2 + \overline{x}_3)\overline{x}_4$$

因此

$$\overline{f} = x_1(x_2x_3 + x_4)$$

由此表达式可直接推导出如图3-17所示的电路。 ■

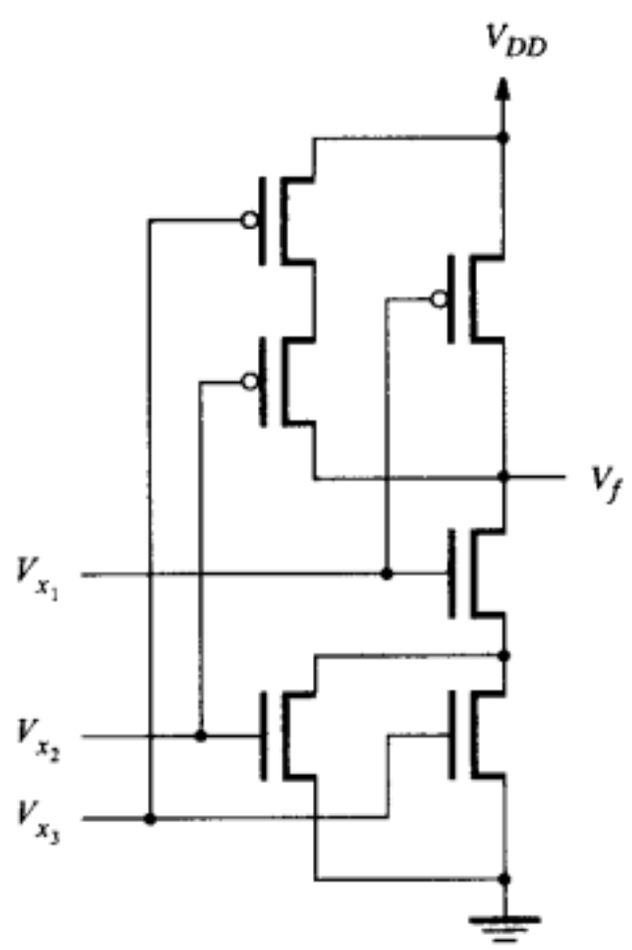


图3-16 例3.1的电路

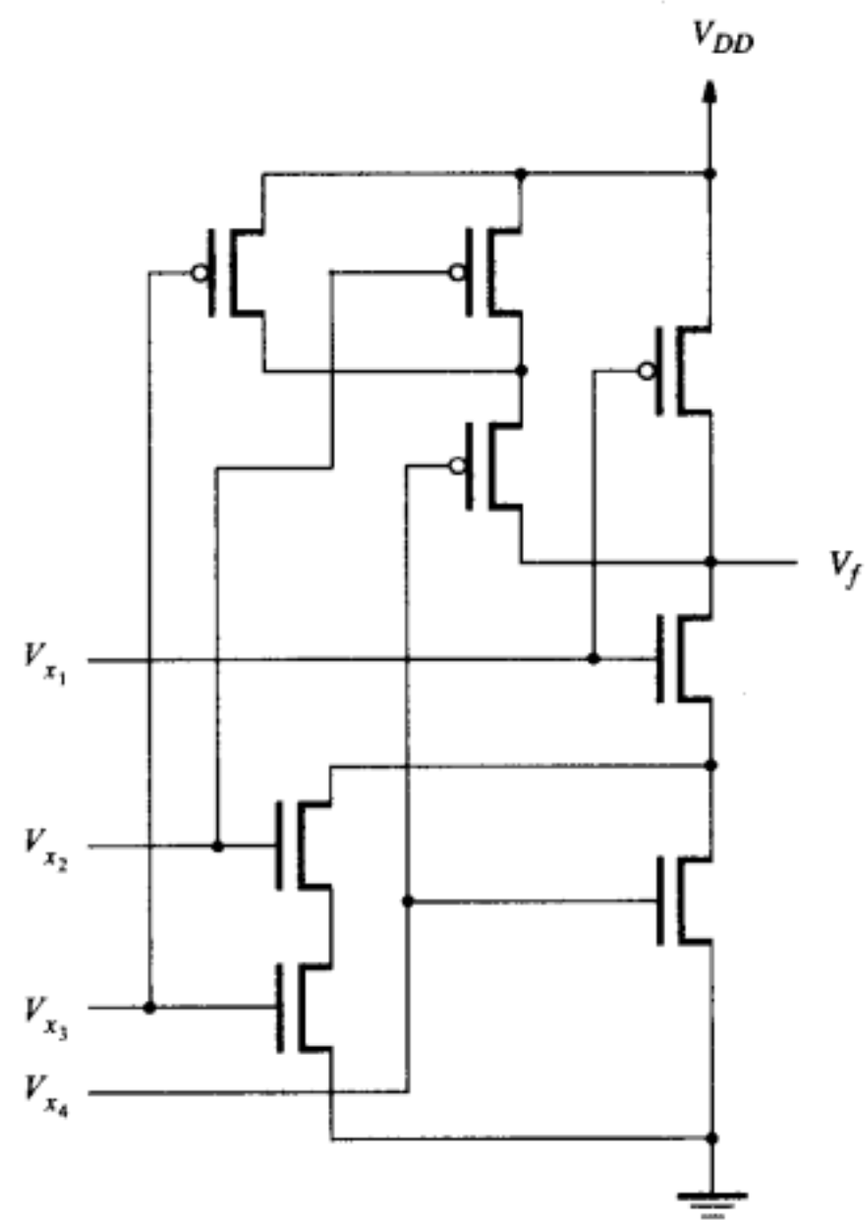


图3-17 例3.2的电路

图3-16和图3-17所示例子表明，可以用（开关）晶体管的串并联来实现一个复杂逻辑功能的电路，而不需要用完整的与门（参见图3-15）、或门的串并联来实现。

逻辑门电路的速度

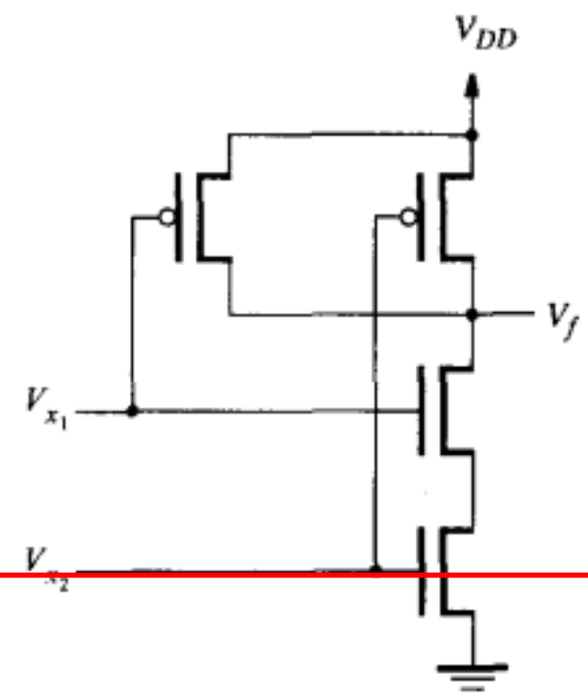
在前面的章节中，我们把晶体管看做理想的开关，即电流流过接通的开关时没有任何电阻。由此推导出的逻辑门电路虽然具有所需的逻辑功能，但是却忽略了一个非常重要的问题，即电路的工作速度。实际上，当晶体管开关接通时，它有一个很明显的电阻。此外，晶体管电路还存在寄生电容，这些因素是由制造过程的副作用引起的。上述因素影响了信号通过逻辑门的传播速度。在3.8节中，我们将详细讨论逻辑门的速度问题，以及其他一些实际问题。

3.4 负逻辑系统

在本章开头，我们曾说过逻辑值用具有显著区别的两个电平值（低电平、高电平）来表示。当时我们用高电平代表逻辑值1，低电平代表逻辑值0，这习惯上称为正逻辑系统。正逻

辑系统在实践中应用得非常广泛。但本节将简要介绍负逻辑系统，其电平和逻辑值的对应关系正好与正逻辑系统相反。

让我们重新考虑图3-13所示的CMOS电路图，并在这里把它重新画一遍，见图3-18a所示。图3-18b是该电路的真值表，不过表中列出的是电平值而不是逻辑值。表中L代表低电平，为0 V；H代表高电平，为电源电压V_{DD}。这种风格的真值表是集成电路生产商的数据表中经常采用的，它用来描述芯片的功能，这样就可以不考虑用户使用哪种逻辑系统，他们可以使用两种逻辑系统中的任何一种，即L = 0而H = 1，或者L = 1而H = 0。



V_{x_1}	V_{x_2}	V_f
L	L	H
L	H	H
H	L	H
H	H	L

a) 电路　　b) 电平

图3-18　图3-13中电路的电平

图3-19a表示的为正逻辑，此时L=0而H=1。由图3-13的讨论，我们早就知道此电路是一个用正逻辑表示的与非门。图3-19b采用的是负逻辑，因此逻辑解释正好相反，此时L = 1而H = 0。而从用负逻辑表示的真值表看恰好是一个或非门。为了与图3-18b用L和H表示的逻辑一致，图3-19b中的真值表正好与常用的正逻辑真值表的取值相反。图3-19b所示的是或非门逻辑符号，同时在或非门的每个接线端各加了一个小三角，用以表示所采用的是负逻辑系统。

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

a) 正逻辑真值表和门符号

x_1	x_2	f
1	1	0
1	0	0
0	1	0
0	0	1

b) 负逻辑真值表和门符号

图3-19　图3-18中电路的两种解释

再举一个例子，重新考虑图3-15所示电路，用电平形式表示其真值表，如图3-20a所示。采用正逻辑系统，这个电路是与门，如图3-20b所示。如果采用负逻辑系统，这个电路是一个或门，如图3-20c所示。

V_{x_1}	V_{x_2}	V_f
L	L	L
L	H	L
H	L	L
H	H	H

a) 电平

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

b) 正逻辑

x_1	x_2	f
1	1	1
1	0	1
0	1	1
0	0	0

c) 负逻辑

图3-20　图3-15电路采用正负逻辑的不同解释

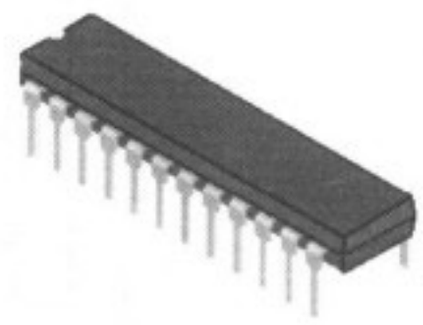
允许在一个电路中混合使用正逻辑和负逻辑系统，称为混合逻辑系统（mixed logic system）。在通常的应用中，我们都采用正逻辑系统，因此本书中将不再考虑负逻辑。

3.5 标准芯片

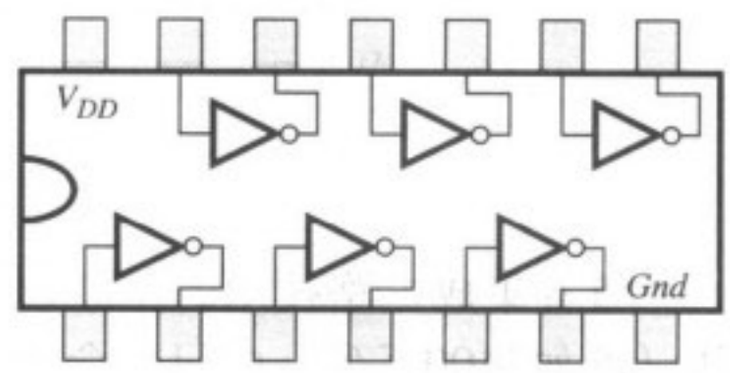
在第1章曾经提到几种不同的可以用来实现逻辑电路的集成电路芯片，现在我们更详细地讨论如何选择不同的芯片。

7400系列标准芯片

在20世纪80年代中期前，广泛采用的电路设计方法是把多个芯片连接起来形成一个逻辑电路，每一个芯片中只包含很少几个逻辑门。为了区分不同类型的逻辑门，产生了芯片系列标准。7400系列的特点是开始的两个数字永远是74，图3-21所示的是74系列芯片的一个实例。图3-21a是一个双列直插（dual-inline package，DIP）封装的芯片，图3-21b所示的为7404芯片的结构，它包含六个非门。芯片的外部连接端称为引脚（pin），有两个引脚分别用于连接到电源V_{DD}和地（*Gnd*），其他引脚用于连接非门的输入和输出端。7400系列包含很多种不同的芯片，芯片制造商提供的数据手册中描述了这些芯片的技术资料[3-7]，有些教课书（见参考文献[8-12]）中也包含某些芯片的图表。



a) 双列直插封装



b) 7404芯片的结构

图3-21 一个7400系列的芯片

很多集成电路生产商使用相同的规范生产标准化的7400系列芯片。制造商之间的竞争使得电路设计者获益，因为竞争降低了芯片价格，并保证了及时供货。对于7400系列中的某特定芯片，由于生产的工艺技术有所不同，型号也有所区别。例如74LS00采用的是TTL技术，见附录E的介绍；而74HC00采用的是CMOS技术。总体上，目前最流行的芯片一般是采用CMOS技术生产的。

现在举例说明如何用7400系列芯片实现一个逻辑电路。假设想要实现的逻辑功能为：$f = x_1x_2 + \overline{x}_2x_3$，其逻辑图如图2-30所示，用一个非门来产生$\overline{x}_2$，还用了两个二输入的与门和一个二输入的或门。该逻辑功能的实现使用了三个7400系列的芯片，如图3-22所示。我们假设三个输入信号x_1，x_2，x_3由其他电路的输出端产生，用导线连接到三个芯片。请注意，这三个芯片的电源和地都已连接，芯片中还剩一部分门没有使用，它们可用来实现其他逻辑功能。

因为标准芯片的逻辑能力低下，现在已经很少使用，但是有一个例外，即缓冲器（buffer）芯片，许多现代产品中仍然在用它。缓冲器是逻辑门的一种，通常用来提高电路的速度。图3-23所示的是缓冲器的一个例子。它是一片74244芯片，内部包含8个三态缓冲器。我们将在3.8.8节中介绍三态缓冲器是怎样工作的。在图3-23中，我们并没有画出芯片封装内部缓冲器具体是如何安排的，而只用引脚号码来表示与缓冲器的输入输出的连接关系，这种表示方法与图3-21画出芯片封装内部非门是如何具体地与引脚连接有所不同。缓冲器封装有20个引脚，引脚的编号方式和图3-21相同，地（*Gnd*）和电源（V_{DD}）分别连接到引脚10和20。还有许多种其他类型的缓冲器芯片，例如162244芯片，其中含有16个三态缓冲器。162244芯片属于另外一个系列，该系列和7400系列类似，但每个芯片中的门数是7400系列的两倍。这种芯片有很多种封装，其中最流行的是SOIC封装（small-outline integrated circuit）。SOIC封装的形状和DIP封装类似，只是几何尺寸小很多。

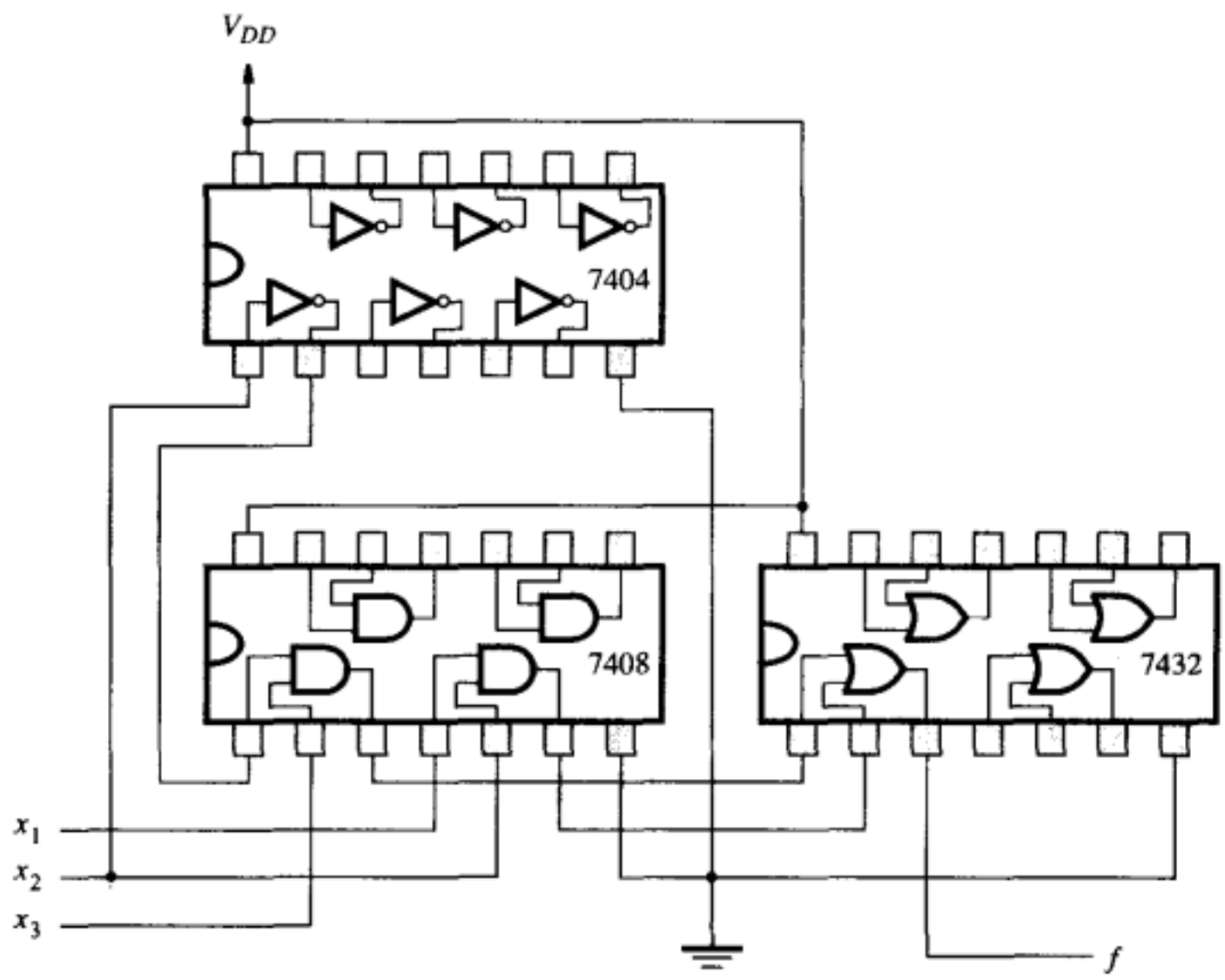


图3-22 逻辑函数 $f = x_1x_2 + \overline{x}_2x_3$ 的实现

随着集成电路技术的不断改进，出现了根据芯片规模的大小进行分类的趋势。最早的芯片（例如7400系列）中只包含少量的门，用这种技术生产的芯片叫做小规模集成电路（SSI）。包含更多些门的芯片，一般包含10～100门的芯片，称为中规模集成电路（MSI）。到了20世纪80年代中期，芯片电路的规模变得更大，用MSI表示已经不相称，于是称为大规模集成电路（LSI）。近年来，用芯片中包含的门数来对集成电路分类已经很少见了，大多数集成电路包含的晶体管数目已达数千或数百万个，所以不管它们的规模究竟有多大，都称为超大规模集成电路（VLSI）。数字硬件产品的趋势是尽可能把更多的电路集成到一个芯片中。所以，当今使用的绝大多数芯片都使用VLSI技术制造，老型号芯片已经很少使用。

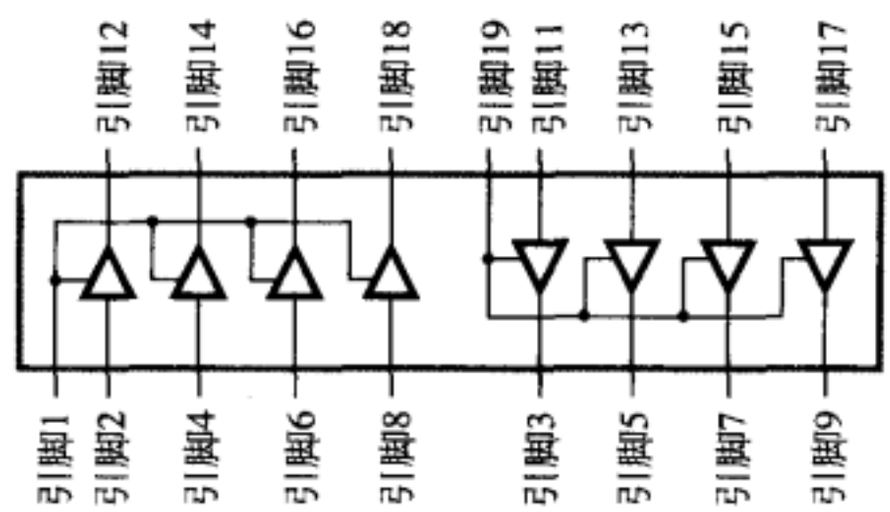


图3-23 74244缓冲器芯片

3.6 可编程逻辑器件

7400系列中每一种器件的逻辑功能是固定的，不能为设计某一特定逻辑电路而改变；而且这些器件内所包含的逻辑门的数量较少，所以构造大型逻辑电路的效果较差。20世纪70年代，出现了包含门数较多并且内部逻辑结构可配置的器件，这种器件被称为可编程逻辑器件（PLD）。

PLD是一种实现逻辑电路的通用器件。其中包含许多个逻辑电路单元，可以根据客户的需要构成不同功能的逻辑电路。可以把PLD看做一个黑盒子，当中包含有很多逻辑门和可编程开关，如图3-24所示。PLD中的逻辑门通过可编程开关连接起来，形成所需要的逻辑电路。

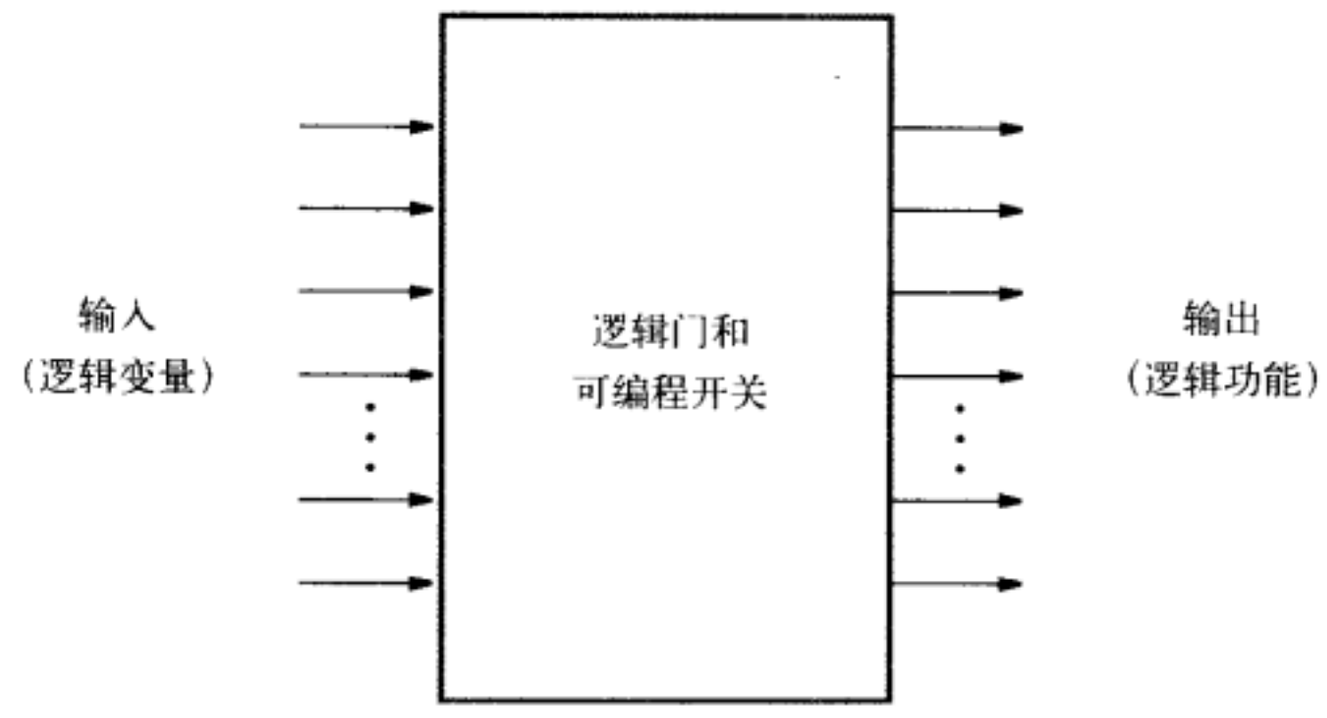


图3-24 作为黑盒子的可编程逻辑器件

3.6.1 可编程逻辑阵列

市面上可以买到许多品种的PLD器件。最先研制出来的叫做可编程逻辑阵列（PLA）。PLA的整体结构如图3-25所示。由于PLA的逻辑功能可以用“积之和”的形式实现，PLA可由一组与门的输出供给一组或门组成。如图3-25所示，PLA的输入x_1，…，x_n经过一组缓冲器（分别提供原变量及反变量）进入与平面，也称与阵列，与阵列产生乘积项P_1，…，P_k。每一个乘积项可以实现输入变量x_1，…，x_n的任何“与”功能。乘积项作为或阵列的输入，或阵列产生输出信号f_1，…，f_m。每一个输出信号可以实现乘积项的任何“或”功能。也就是说，实现了PLA输入的“积之和”功能。

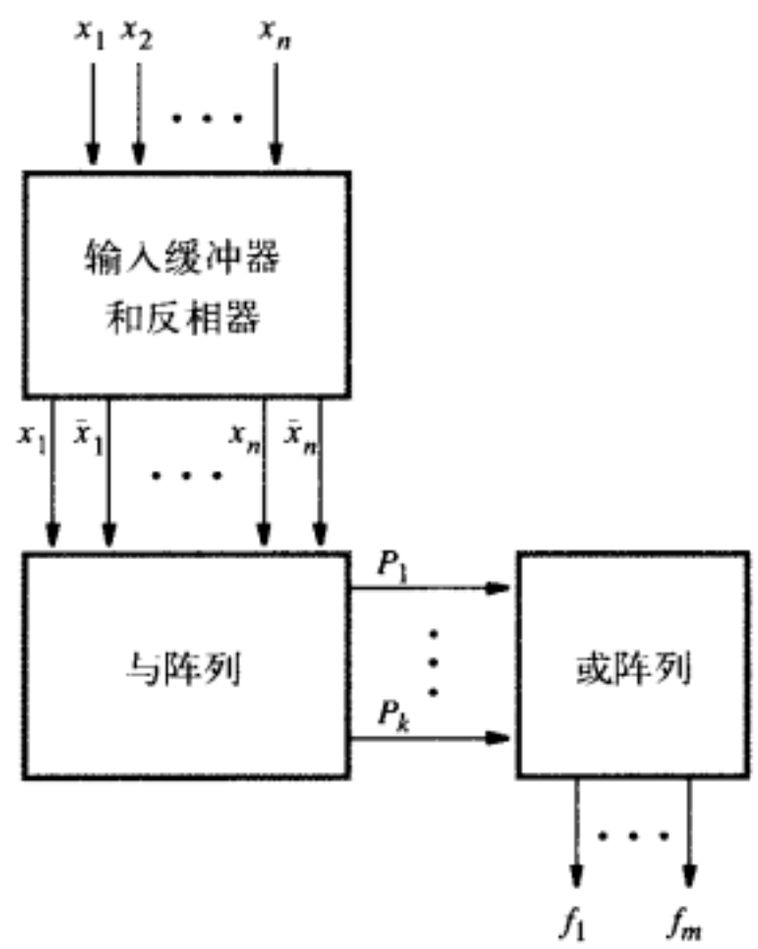


图3-25 可编程逻辑阵列的总体结构

图3-26所示的是一个小规模PLA的详细图示，它有三个输入，四个乘积项，两个输出。与阵列中的每一个与门有六个输入，分别对应于三个输入的原变量及反变量。每一个输入和与门的连接关系是可编程的，输入和与门相连用波浪线表示，输入和与门不相连用断开的线表示。电路的这种设计使得和与门不相连的输入对于此与门的输出没有作用。在市场上销售的PLA中，实现连接线编程的方法有多种。在3.10节中，将详细介绍如何用晶体管来构造一个PLA。

如图3-26所示，输出为P_1的与门的输入端连接x_1和x_2，因此 $P_1 = x_1x_2$。同理，$P_2 = x_1\overline{x}_3$，$P_3 = \overline{x}_1\overline{x}_2x_3$，$P_4 = x_1x_3$。或阵列同样是可编程的，输出为 f_1的或门的输入端连接到乘积项P_1、P_2和P_3。因此实现了 $f_1 = x_1x_2 + x_1\overline{x}_3 + \overline{x}_1\overline{x}_2x_3$。同理，$f_2 = x_1x_2 + \overline{x}_1\overline{x}_2x_3 + x_1x_3$。尽管图3-26画出了实现上述功能的PLA，但实际上对与阵列及或阵列分别编程，输出 f_1 和 f_2 各自都可以实现输入x_1、x_2和x_3的各种函数。对所能实现逻辑功能的唯一的限制是与阵列的规模，因为只有四个乘积项。市场上销售的PLA的规模比上面介绍的大一些，典型的参数是16个输入，32个乘积项，8个输出。

虽然图3-26清晰地表示了PLA的功能结构，但是这种画图风格对于表示大型PLA电路是非常不适用的。图3-27是技术上惯用的另一种风格，每个与门与一条水平横线连接，而有可能成为与门输入的信号线画成与水平线相交的垂直线，水平线和垂直线的交点处打个叉（×）

表明该输入被编程为和与门连接。图3-27所示的编程连接是按图3-26的乘积项要求实现的。或阵列的描述风格与此类似，或门连接一根垂直线，和与门的输出线相交，对交叉点连接进行编程，可以实现所需要的逻辑功能。图3-27与图3-26实现的逻辑功能相同。

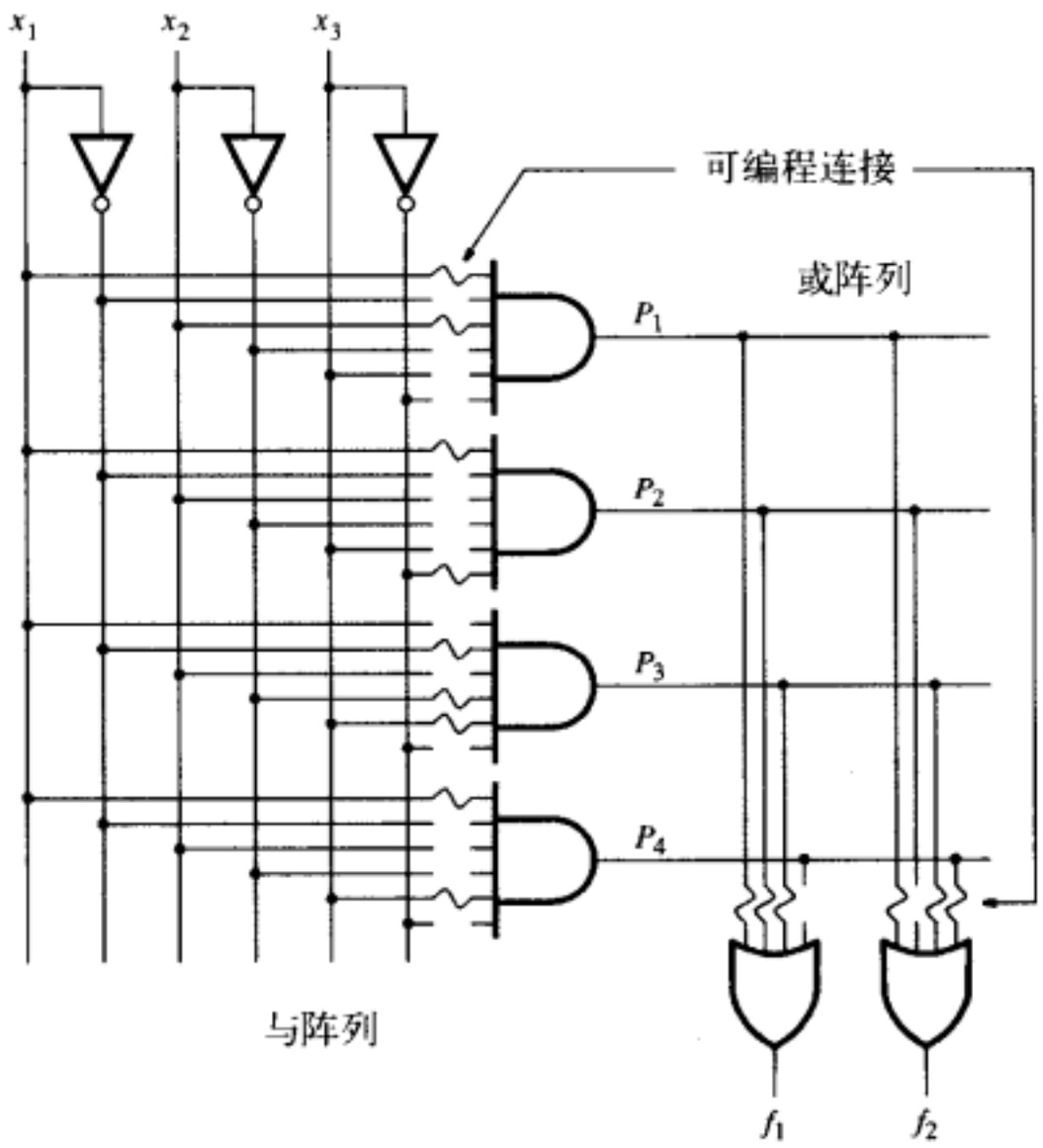


图3-26 PLA的门级图

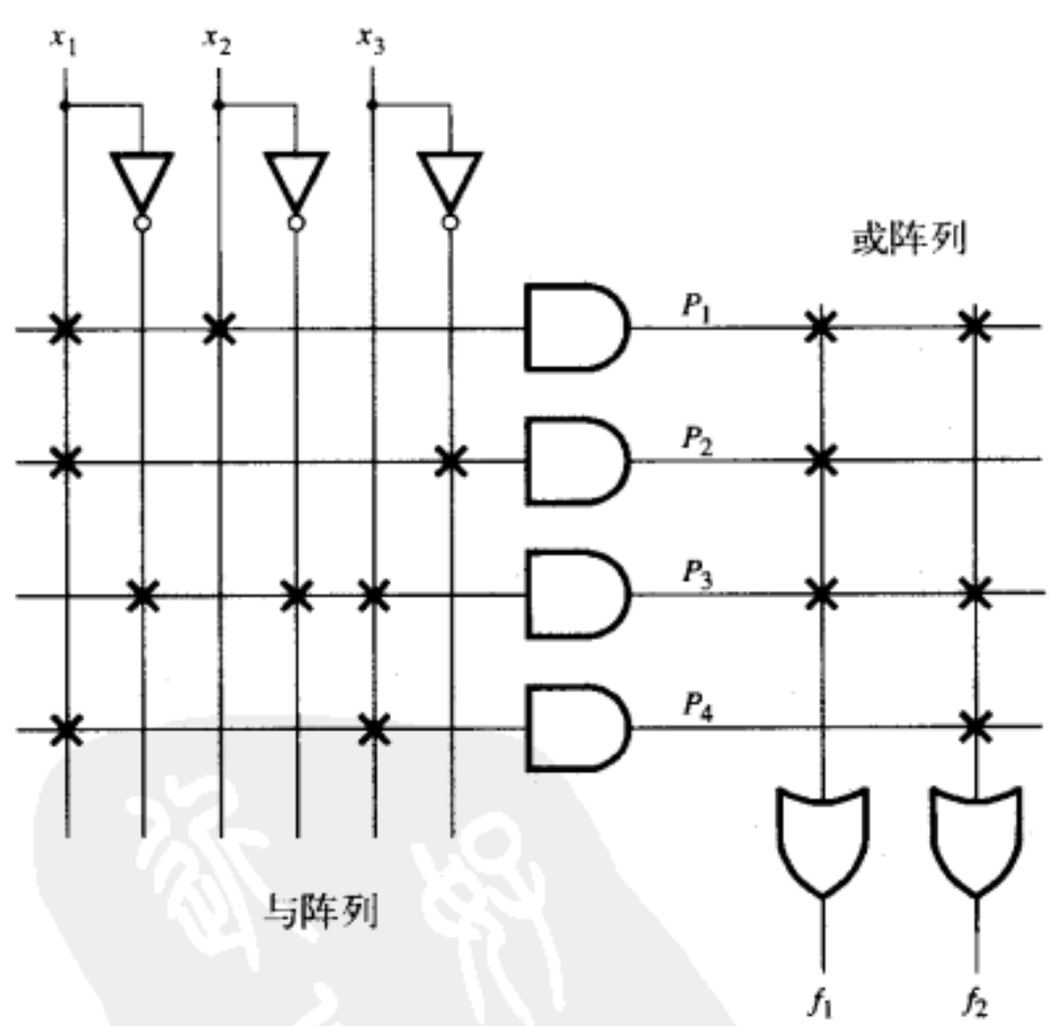


图3-27 图3-26中PLA的惯用线路图

电路集成时，采用PLA结构有利于节约芯片面积。所以，PLA结构常常应用于大型芯片，例如微处理器中。在这种情况下，与门以及或门的连接都是固定的，而不是可编程的。在3.10节中我们将会看到，可以用类似的结构建立可编程的和固定的PLA。

3.6.2 可编程阵列逻辑

PLA中的与阵列和或阵列都是可编程的。从历史的发展来看，可编程开关在制造上曾存

在两大困难：制造无瑕疵很难，用PLA实现的电路速度低。因此，人们开发了一种类似的器件，它的与阵列可编程而或阵列是固定的，这种电路称为可编程阵列逻辑（PAL）。因为PAL制造简单一些，价格低廉一些，性能高一些，因而在实际应用中更为流行。

图3-28所示是PAL的一个例子，它有3个输入，4个乘积项，2个输出。乘积项P_1和P_2硬性连接到一个或门，而P_3和P_4则硬性连接到另一个或门。该PAL被编程实现下述两个逻辑函数：

$$f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$$

$$f_2 = \bar{x}_1\bar{x}_2 + x_1x_2x_3$$

和图3-27所示PLA相比较，PAL的灵活度较小：PLA中每一个或门最多允许4个输入，而PAL中或门的输入却只有2个。作为对PAL灵活性减小的一种补偿，PAL的规模可在一定范围内变化，其输入输出数量皆可变化，或门的输入端数也可以不同。附录E中有一个市售PAL的例子。

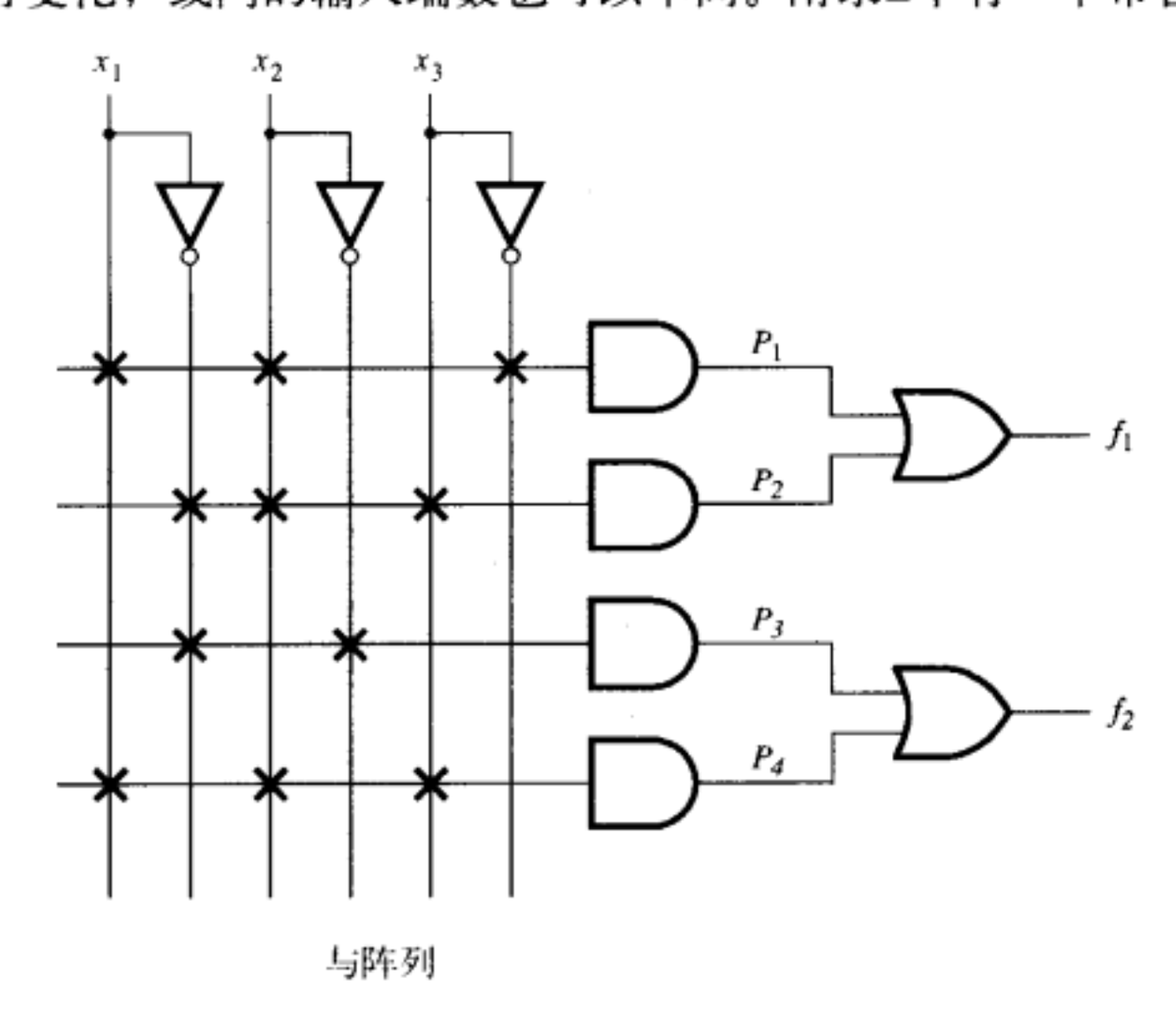


图3-28 PAL实例示意图

到目前为止，和PLA一样，我们只是假定PAL中或门的输出直接连接到芯片的输出。为了增加PAL的灵活性，实际上，许多PAL在或门之后还附加一些额外电路，这种电路通常称为*宏单元*（macrocell）。图3-29所示的是宏单元灵活性的一个实例，上面标有“触发器”的符号表示一个记忆元件，它在特定时刻保存来自或门的信号值，并且一直保持下去。触发器受时钟信号clock的控制，当时钟信号由0跳变到1时，触发器保存D端输入信号当时的值，并把这个值输出到触发器的输出端Q。很多逻辑电路中都会用到触发器，我们将在第7章中介绍触发器。

在2.8.2节中我们曾经讨论过2选1多路复用器，它有2个数据输入，1个选择输入和1个输出。选择输入用来从2个数据输入中选择一个送到多路复用器的输出。图3-29中的2选1多路复用器用来为PAL输出端选择信号：或者把或门的输出作为PAL的输出，或者把触发器的输出作为PAL的输出。多路复用器的选择线可被编程为1或0。图3-29中还另有一个逻辑门，叫做三态缓冲器，连接在多路复用器和PAL输出端之间。我们将在3.8.8节讨论三态缓冲器。最后，我们注意到多路复用器的输出反馈到PAL的与阵列。这根反馈连接线的存在就允许由多路复用器产生的逻辑函数可应用于PAL的内部，从而允许产生由逻辑门组成的多级电路。

有许多厂商可提供PLA或PAL以及类似的简单PLD（Simple PLD，SPLD）产品，附录E

中列出了部分厂商及其生产的SPLD类产品清单。有兴趣的读者可以上网查找这些厂商提供的有关信息。附录E中的表E-1列出了这些厂商的万维网地址。

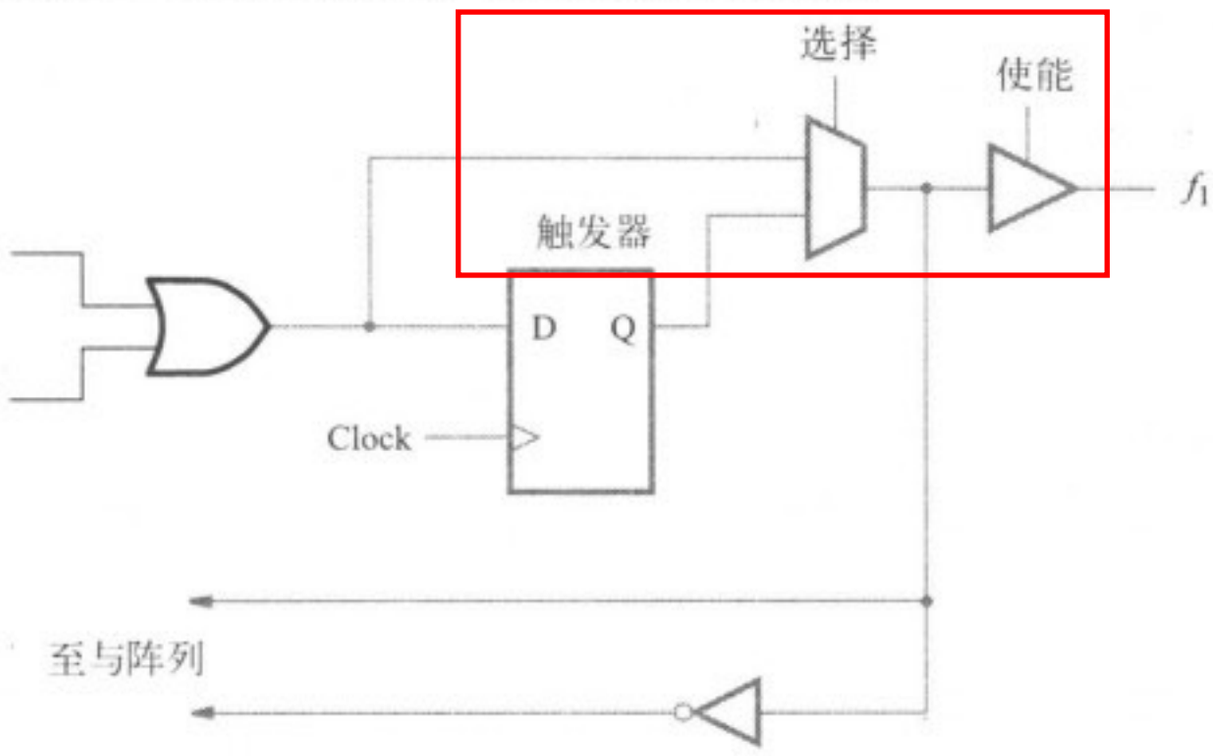


图3-29 图3-28或门输出后的附加电路

3.6.3 PLA和PAL的编程

在图3-27和图3-28中，PLA或PAL中的信号和与门/或门的相连用符号“×”表示，我们将在3.10节中介绍如何用晶体管来实现这些开关。通过对这些开关的适当配置或编程，就能实现用户的电路。市售的PLD芯片含有数千个可编程开关，因此由用户人工地指定每一个开关的状态是不现实的，所以只能使用CAD工具来实现这个目的。第2章中介绍了CAD工具以及如何用CAD工具进行电路描述并仿真。那些支持PLD器件的CAD工具，能够自动产生对PLD器件中每一个开关的编程信息。运行CAD工具的计算机通过电缆与专用的编程器连接。当用户完成电路设计时，CAD工具就会产生一个编程文件（或称为熔丝映射表），它规定了PLD器件中每一个开关的状态，用于正确地实现用户设计的电路。PLD器件被放置在编程器中，计算机系统把编程文件传送到编程器中。接着编程器使芯片进入特定的编程模式中，然后给芯片中每一个开关配置合适的状态（译者注：即转化为烧断熔丝的操作）。编程器实物的照片如图3-30所示。主编程器旁边的是一些适配器，每一种适配器适合于某种特定封装的芯片。

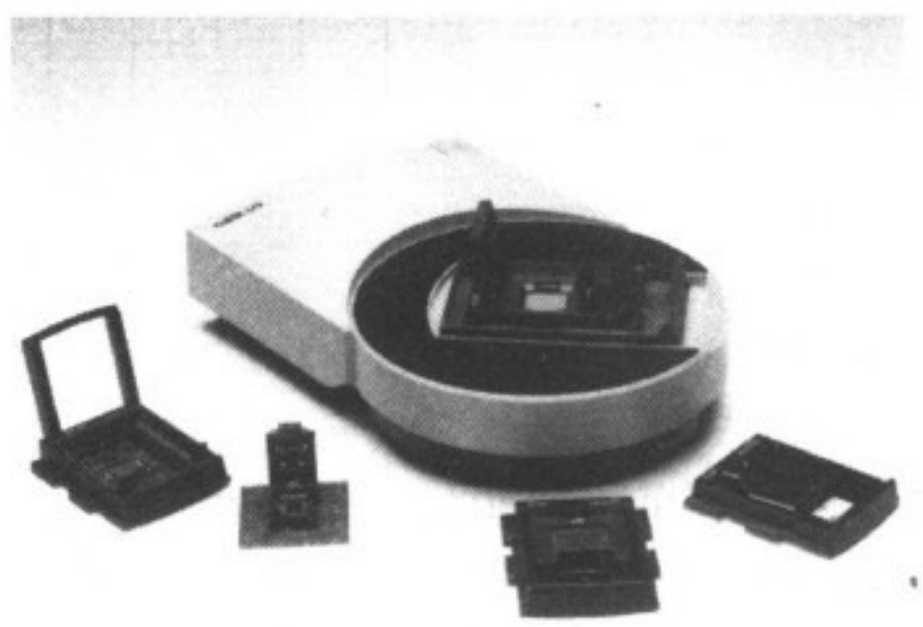


图3-30 PLD编程器（感谢Data IO公司许可）

编程的过程大概需要几分钟才能完成。编程完毕之后，通常编程器会自动读回芯片编程后内部每一个开关的状态，以检验芯片编程的正确性。关于如何借助CAD工具用可编程PLD芯片实现设计电路的过程，请参阅附录B、C和D，其中有详细的介绍。

PLA或PAL芯片通常作为逻辑电路的一部分和其他芯片共同处于一块印制电路板上。上述编程过程是假设该PLD芯片可以从印制电路板上取下来，然后放入编程器中进行编程。芯片能够从印制电路板上取下的原因，是因为印制电路板上装有插座（见图3-31）。插座有多种形式：如图3-21所示的双列直插式（简称DIP），也有图3-31所示的塑料封装（plastic-leaded chip carrier，PLCC）。PLCC封装的四边都有引脚，而DIP封装只在两侧才有引脚。插座用锡焊接在印制电路板上，而PLCC则是靠摩擦固定在插座中。

若使用编程器进行编程，必须把芯片从印制电路板上拔下来，为了避免这种麻烦，允许芯片仍然在印制电路板上的情况下进行编程，出现了在线编程（in-system programming，ISP）技术。不过这种技术一般不用在PLA和PAL编程中，而是用于下面介绍的更复杂的芯片编程。

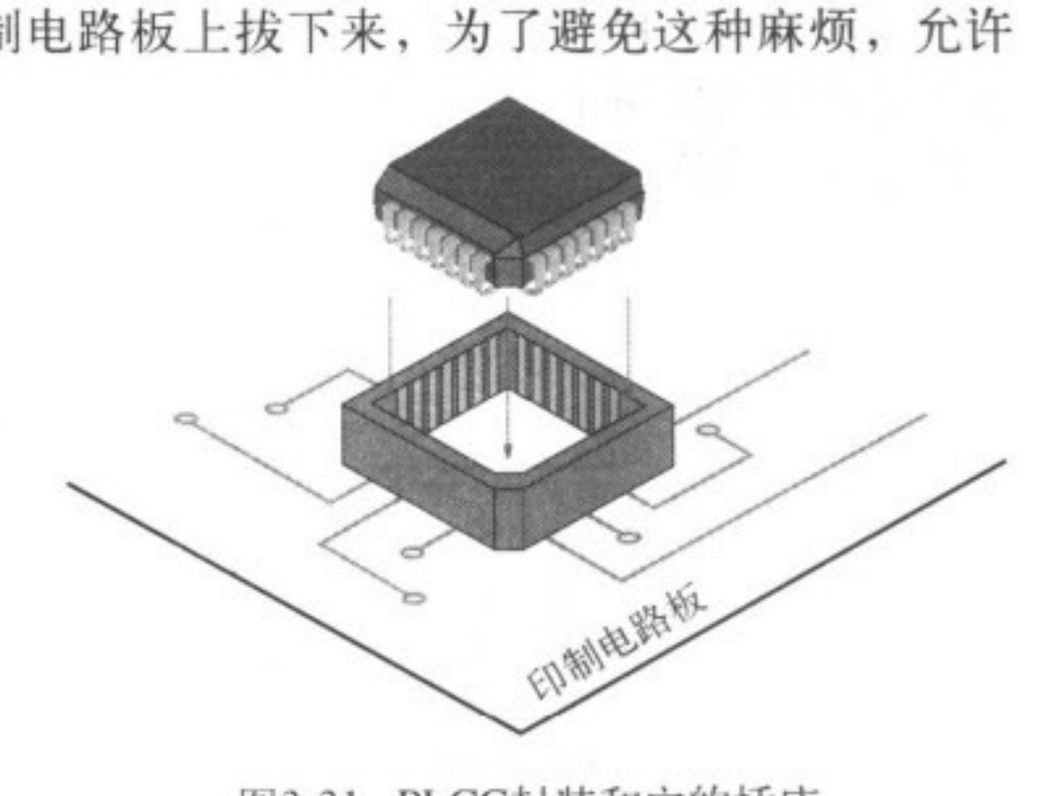


图3-31 PLCC封装和它的插座

3.6.4 复杂可编程逻辑器件

PLA和PAL通常用于实现规模较小的数字电路，专用芯片的输入和输出引脚数以及乘积项的个数都会受到一定的限制。输入引脚数加上输出引脚数一般不超过32。为了实现有更多输入和输出需求的大型电路，就需要采用多个PLA或PAL，或者采用另一种更复杂的称为复杂可编程逻辑器件（简称CPLD）的芯片。

CPLD芯片中包含多个电路块，并且有内部连线资源把这些电路块连接在一起。每个电路块都和PLA或PAL结构相似，我们把这些电路块称为类似PAL的电路块。如图3-32所示是CPLD的一个实例，该芯片包含4个类似PAL的电路块，每个类似PAL的电路块和内部连线资源相连，还连接到标记为I/O块的子电路。I/O块和芯片的输入输出引脚相连。

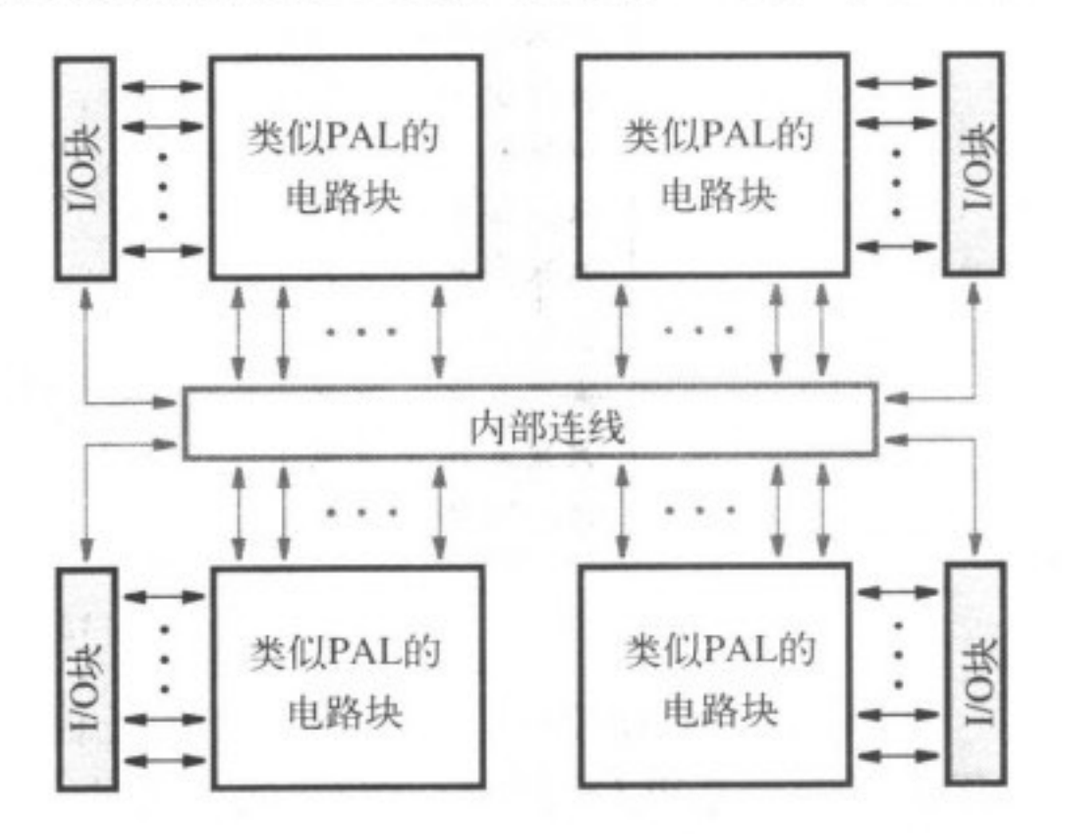


图3-32 CPLD器件的内部结构

图3-33举了一个例子，表示CPLD内部类似PAL电路块的连线结构以及它和电路块的连接情况。这个类似PAL的电路块中包含3个宏单元（实际典型的CPLD的电路块中包含16个宏单元），每个宏单元的或门有4个输入（实际典型的CPLD的或门的输入个数是5～20）。或门的输出连接到另一种我们还没有介绍的典型逻辑门，即异或门（XOR）。我们将在3.9.1节中介绍异或门。异或门的行为和或门相似，只有一点例外，即当两个输入都是1时输出是0。图3-33中的异或门的两个输入中的一个可以被编程，即可以连接到1或0。如果是1，则异或门把或门输出的信号反相后输出；如果是0，则异或门把或门输出的信号按原样传送到自己的输出端。宏单元中还有1个触发器、1个多路复用器和1个三态缓冲器。在我们介绍图3-29时已经提到，触发器用于保存或门产生的输出值。每个三态缓冲器（参见3.8.8节）连接到CPLD封装的一个引脚，三态缓冲器就像开关一样，使得每一个引脚既可以作为CPLD的输出，又可以作为CPLD的输入。若把相应三态门的使能端置为1，则开关接通，该引脚就成为输出引脚。若把三态门的使能端置为0，则开关断开，该引脚就变为输入引脚。在这种情况下，外部信号经过该引脚到达内部连线资源，然后连接到其他宏单元。

内部连线资源包含着许多可编程开关，用于类似PAL的电路块的互相连接。每一根水平线可以和与之相交的垂直线的一部分相连，但不是全部相连。人们在进行许多深入研究之后确定了连线具体需要的开关数量。实际应用中，开关的数目既要足够多，以保证连接的灵活

性，又不要太多而造成浪费。有一个值得注意的细节是：一个引脚用做输入时，与之相连的宏单元就不能使用了，也就浪费了。为了避免浪费，有些CPLD在宏单元和内部连线资源之间又添加了一些连线。

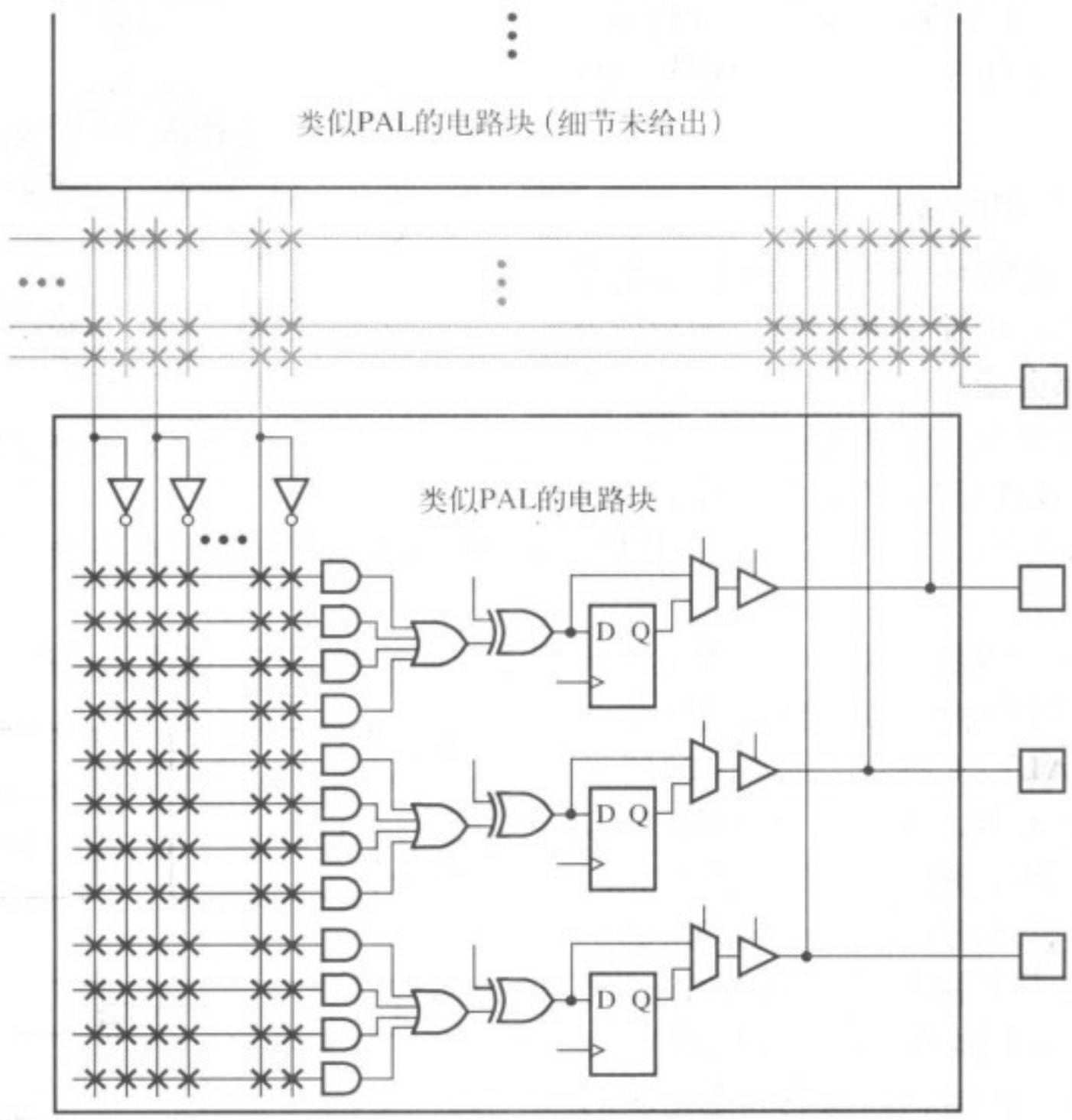


图3-33 图3-32中的部分细节放大图

市售CPLD的规模可以从包含两个到几百个类似PAL的电路块不等。封装形式也多种多样，如图3-31所示是PLCC封装，图3-34a则是另一种叫做方形扁平（quad flat pack，QFP的封装）。QFP和PLCC封装的相同之处是：封装四面都有引脚；不同处在于：QFP的引脚弯曲向外，而PLCC的引脚是直立的。QFP的引脚比PLCC的引脚细很多，所以一个封装能容纳更多的引脚。通常PLCC的引脚总数小于100，QFP的引脚总数却可以大于200。

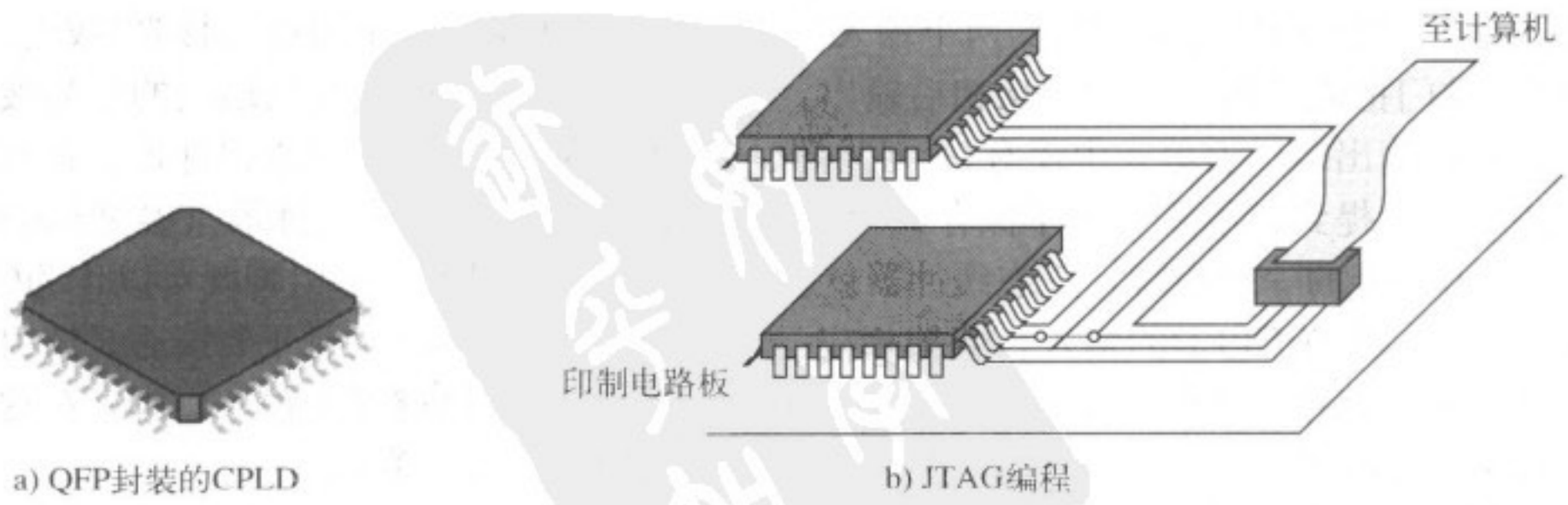


图3-34 CPLD的封装和编程

绝大多数的复杂可编程逻辑器件（CPLD）和将在3.10节中介绍的简单可编程逻辑器件

（SPLD）内部可编程开关的类型是相同的。编程开关可以用3.6.3节中介绍的编程方法完成编程，即把芯片置于特殊的编程器中。但是这种方法对于大型CPLD却很不方便，原因之一是大型CPLD封装的引脚总数超过200，这些引脚又多又细，易折并且易弯；原因之二是需要一个价格昂贵的插座，这种插座的价格甚至高于CPLD本身。由于这些原因，CPLD支持在线可编程（ISP）技术，CPLD芯片所在的印制电路板上焊有一个小的连接器，用一根电缆把连接器和计算机系统连接起来。CAD系统产生的编程信息通过电缆和连接器到达CPLD芯片，从而实现对CPLD的编程。这种技术已经由IEEE的*联合行动测试组*（Joint Test Action Group，JTAG）加以标准化，在被编程器件的内部附加一些电路，在被编程电路的外部使用4根连线（称为JTAG端口），通过这4根连线在计算机和被编程器件之间传送信息。图3-34b所示是用JTAG端口对在同一电路板上的两个CPLD编程的一个实例，两个器件连接在一起，所以可以同时连接到计算机系统，同时被编程。一旦CPLD器件被编程后，它的编程状态就能保持不变，即使电源断电也不会丢失原有的信息，这种性质叫*非挥发性编程*。

CPLD可用于实现很多种数字电路。用可编程逻辑器件实现的工业设计中，常使用CPLD，而SPLD则越来越少见了。有许多家公司可提供CPLD器件，附录E中的表E-2列出了CPLD主要供应厂商的名字和网址，我们鼓励读者到相关公司的网站上查看产品信息。附录E中举了一个例子，详细地介绍了市售的CPLD。

3.6.5 现场可编程门阵列

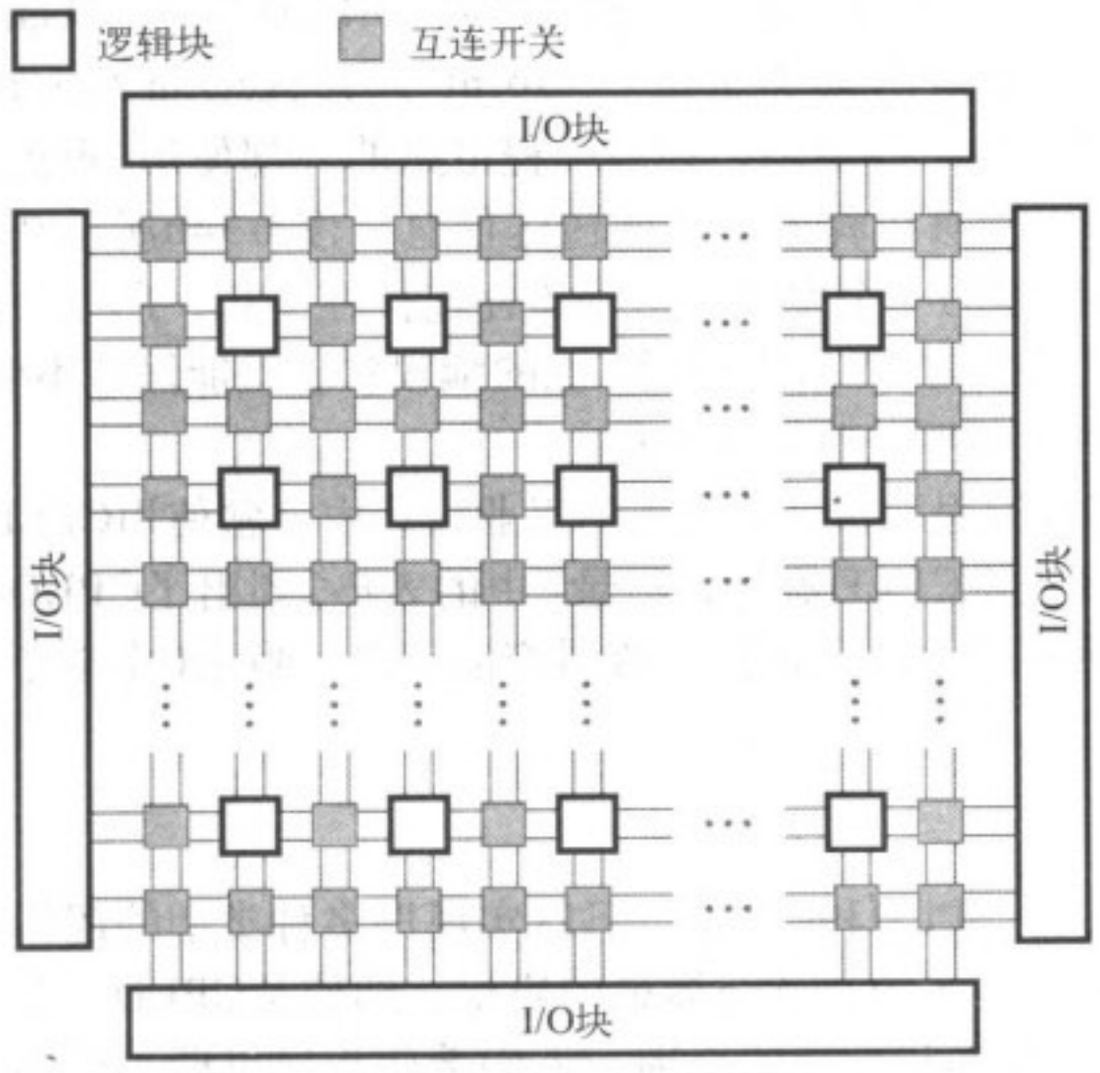
以上介绍过的各种器件，包括7400系列、SPLD和CPLD，可以实现许多种类型的逻辑电路。除了CPLD之外，这些器件的规模都很小，多数只适合简单的应用。即使是CPLD，在一个芯片中也只能容纳中规模逻辑电路。由于成本和性能的原因，人们希望用尽可能少的芯片来实现所要求的电路。所以，给定芯片中逻辑门的数量和逻辑功能就变得十分重要。衡量电路规模的方法之一是假设电路只用简单的逻辑门组成，并计算实现电路所需的逻辑门总数。常用的方法是计算电路所使用的2输入与非门的总数，这个总数又称为*等价门数*。

用等价门数作衡量标准，7400系列芯片的规模很容易计算，因为每个芯片中只包含单纯的逻辑门，数量也很少。而SPLD和CPLD中的基本单元是宏单元，而不单纯是逻辑门，所以典型的方法是把每个宏单元看做20个等价门。由此，包含8个宏单元的PAL器件，就可以容纳大约160个等价门的电路；包含1000个宏单元CPLD器件，就可以容纳大约2万个等价门的电路。

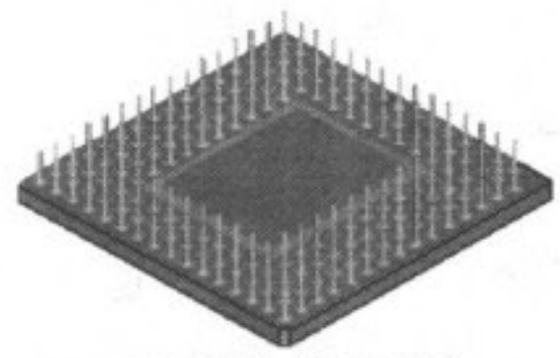
随着技术的进步，按现代标准，由2万个门所组成的电路已经不能算作大型逻辑电路。为了实现更大规模的逻辑电路，使用一种类型不同具有更大逻辑功能的芯片会更方便些。*现场可编程门阵列*（FPGA）是一种可以实现大规模逻辑电路的器件。FPGA与SPLD和CPLD的内部结构迥然不同，因为FPGA内部没有与阵列和或阵列结构。FPGA是用逻辑块来实现其所需的逻辑功能。FPGA的总体结构如图3-35a所示，其中主要包含三种资源：逻辑块（logic block）、I/O块用于连接封装的引脚，还有内部连线和开关。逻辑块排列成二维阵列，连线在逻辑块行和列之间排列成垂直和水平路由通道。路由通道中含有连线和可编程开关，允许逻辑块以多种方式连接。图3-35a中灰色方块表示可编程开关的两种位置：处在和逻辑块相邻的灰色开关用来连接逻辑块的输入输出端口和连线；而处在逻辑块间对角线位置上的开关则用于实现内部各连线之间的连接（例如水平线和垂直线的连接）。可编程开关也存在于I/O块和连线之间，用来实现它们之间的连接。市售FPGA芯片中可编程开关和连线的实际数目各不相同，取决于芯片的规格型号和生产厂商。

用FPGA可以实现很大规模的电路（超过一百万个等价门）。附录E中列举了Altera公司以及Xilinx公司的商用FPGA的几个例子。FPGA的封装有多种选择，如前面提到的PLCC和QFP

封装，如图3-35b所示则是另外一种形式的封装，叫做*引脚网格阵列*（pin grid array，PGA）。PGA封装的引脚数可达数百个，引脚从封装的底部开始以网格形式直接向下延伸。还出现了一种称为*球形网格阵列*（ball grid array，BGA）的封装形式，BGA和PGA很相似，不同点在于它的引脚是小球而不是直的小针。BGA的优点是引脚非常小，所以一个封装上可以提供更多的引脚。



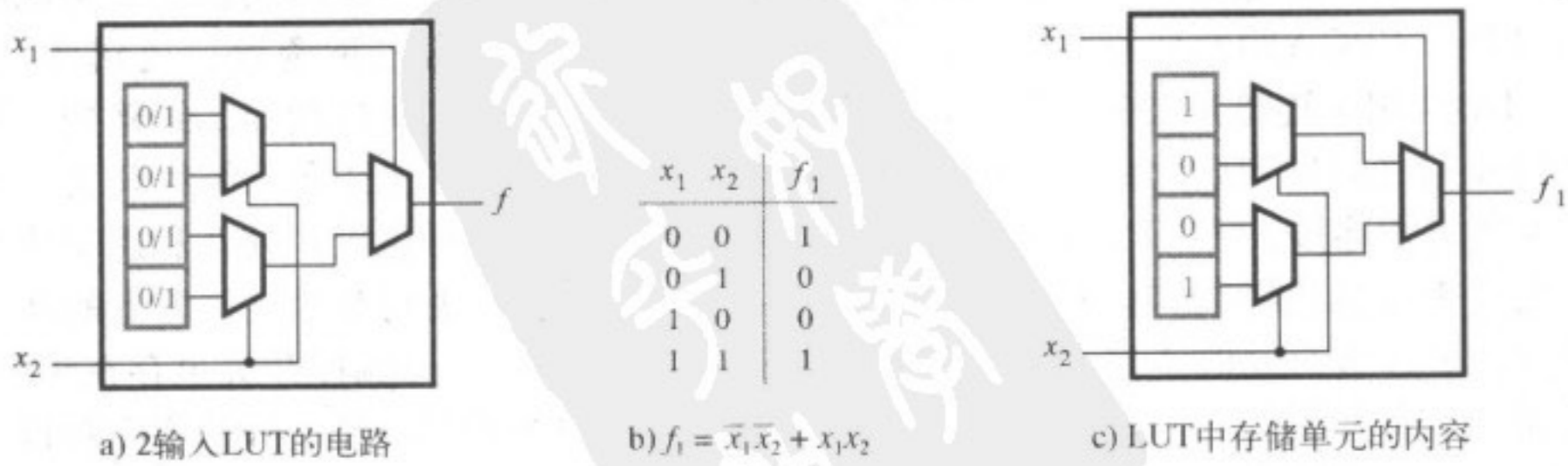
a) FPGA的总体结构



b) PGA封装（底视图）

图3-35 现场可编程门阵列的总体结构和封装

FPGA的每个逻辑块典型情况下有很少几个输入和输出，市售的FPGA逻辑块的特点也各不相同，最常用的逻辑块是*查找表*（LUT）。LUT中包含若干个*存储单元*，用于实现一个小规模的逻辑函数。每个存储单元都可以存储一个逻辑值0或1，作为该单元的输出。可以创建各种规模的LUT，规模由输入个数来决定。如图3-36a所示是一个规模较小的LUT，它只有2个输入x_1和x_2，1个输出f，可以实现2个输入变量的任意逻辑函数。因为2个变量的真值表有4行，所以这个LUT有4个存储单元，一个存储单元对应真值表中一行的输出值。输入变量x_1和x_2用作3个多路器的选择控制，根据x_1和x_2的值从4个存储单元中选择1个作为LUT的输出。我们在2.7.2节中已经介绍过多路器，存储单元将在第10章中介绍。



x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

a) 2输入LUT的电路　　b) $f_1 = \overline{x}_1\overline{x}_2 + x_1x_2$　　c) LUT中存储单元的内容

图3-36 一个2输入查找表

为了深入了解怎样用2输入的LUT实现逻辑函数，观察图3-36b所示的真值表。真值表中

函数 f_1可存储在图3-36c所示的LUT中，LUT中的多路器的安排能够正确地实现函数 f_1。当$x_1 = x_2 = 0$时，LUT的输出被顶层存储单元所驱动，和真值表中输入为$x_1x_2 = 00$的那行相对应。同理，当输入变量x_1x_2取其他值时，LUT的输出值也都和真值表一致。因为只允许1个存储单元通过多路器到达LUT的输出端，所以这种结构能够正确地实现想要的逻辑函数。我们将在第6章中详细讨论多路器的应用。

如图3-37所示是一个3输入的LUT，因为3变量真值表总共有8行，所以它有8个存储单元。在市售FPGA芯片中，LUT的输入个数一般是4个或5个，所以存储单元的个数也就分别是16个或32个。在图3-29中我们曾介绍过，PAL中除了包含与阵列和或阵列之外，还有一些附加的电路。同样，FPGA的逻辑块中除了有LUT之外，也包含一些附加的电路。如图3-38所示，在FPGA中添加了一个触发器。如图3-29所示，触发器的作用是：在时钟信号的控制下，将D输入端的值保存起来。市售FPGA逻辑块的实例见附录E。

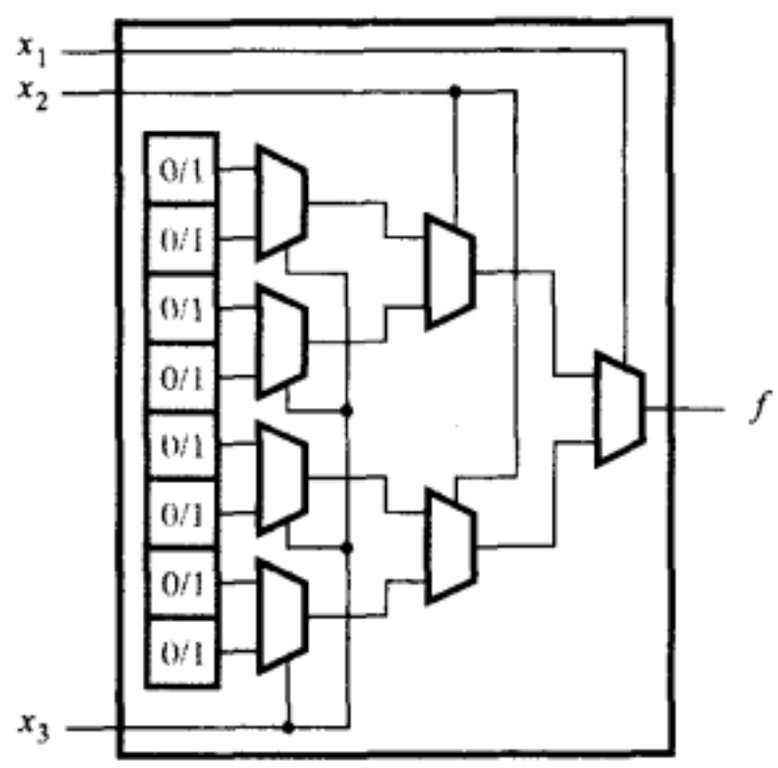


图3-37 一个3输入的查找表

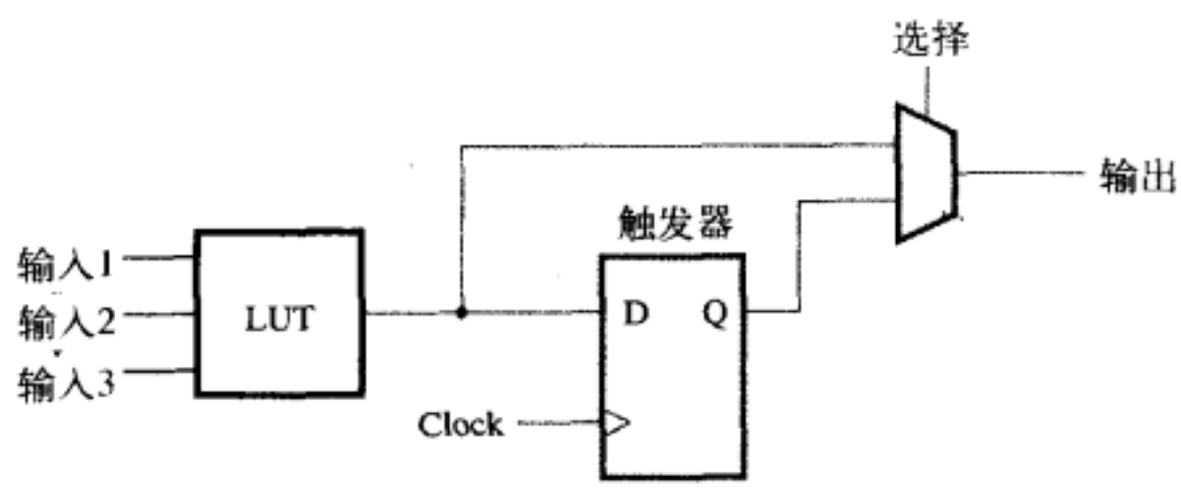


图3-38 在FPGA的逻辑块中添加一个触发器

由于一个逻辑块的规模较小，它只能实现规模足够小的逻辑函数。对于规模很大的逻辑电路而言，为了能用FPGA芯片实现，复杂的逻辑函数必须分解成足够小以适应逻辑块的规模。实际上，用户电路可以用CAD工具自动转化为所需要的形式（见第12章）。若想用FPGA实现一个电路，则FPGA的逻辑块被编程，用来实现必要的逻辑功能，布线通道通过编程实现了逻辑块之间的连接。我们在3.6.4节中介绍过，FPGA的编程采用ISP方法。FPGA中LUT的存储单元是挥发性的，意味着当电源断开时芯片中的内容会全部丢失。所以，每次通电时都要对FPGA进行重新编程。通常可以用一个芯片保存编程数据，这个芯片叫做*可编程只读存储器*（PROM），它与FPGA安装在同一块电路板上。当给电路板通电时，数据就可以自动地从PROM中加载到FPGA的存储单元中。

如图3-39所示是一个已经被编程的小规模的FPGA，它实现了一个小规模的逻辑电路。该FPGA的LUT有2个输入端，每个布线通道有4根线。该图显示了FPGA中一部分逻辑块和布线通道的编程状态，可编程开关用符号×表示。灰色表示开关接通，即把水平线和垂直线相连，黑色表示开关断开。我们将在3.10.1节中介绍如何用晶体管实现开关。根据真值表的内容对FPGA最上边一行的逻辑块进行编程，对应的逻辑函数为：$f_1 = x_1x_2$和$f_2 = \overline{x}_2x_3$。图中右下角的查找表实现的逻辑函数为：

$$f = f_1 + f_2 = x_1x_2 + \overline{x}_2x_3$$

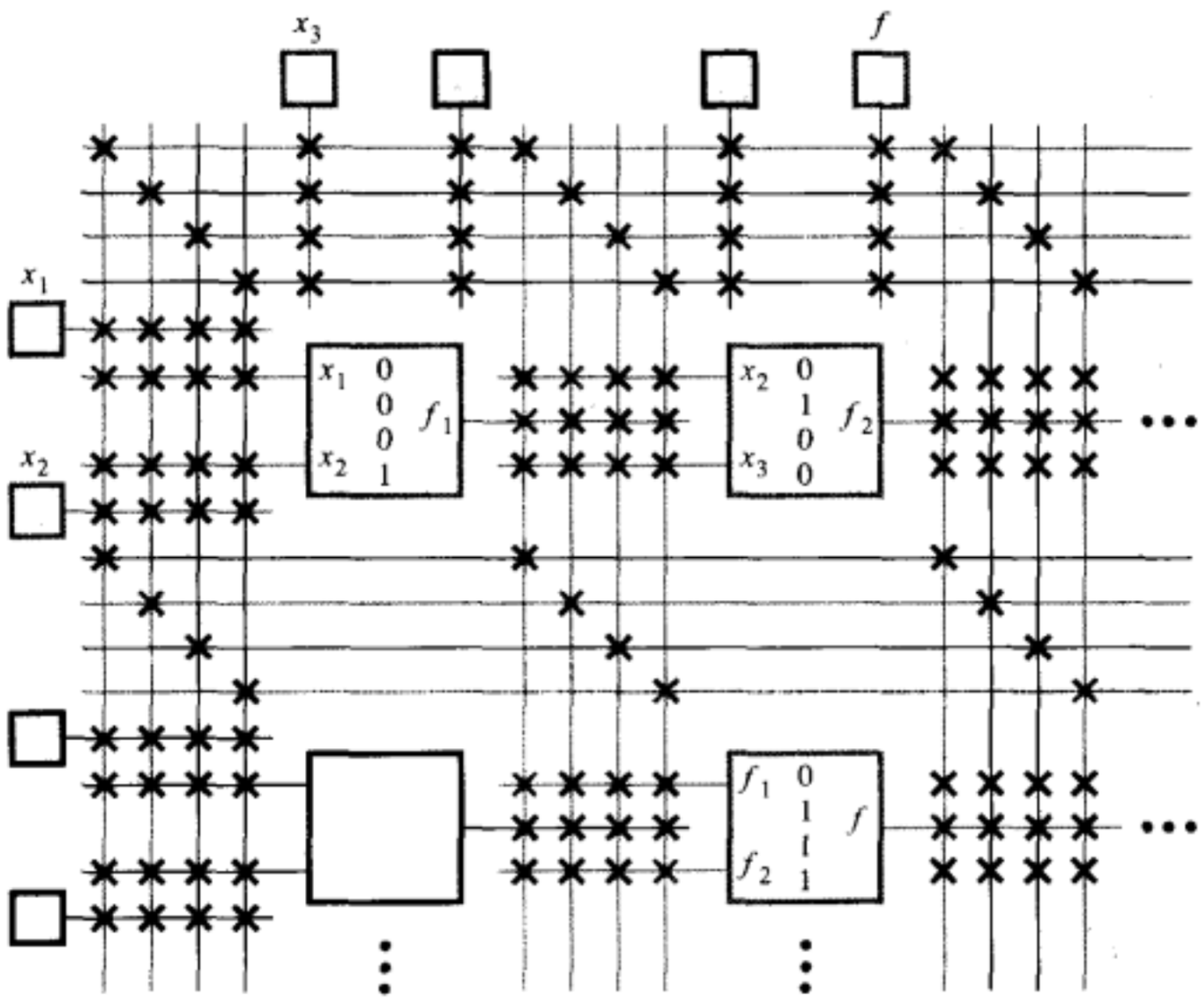


图3-39 已被编程的FPGA的一部分

3.6.6 用CAD工具在CPLD 和FPGA上实现逻辑电路

在2.9节中我们曾建议读者应该学习一下附录B，以获得一些使用CAD工具的实际经验。附录B中详细介绍了设计输入和功能仿真的步骤。到目前为止，我们已经比较详细地介绍了如何在芯片中实现所需的电路，读者可能希望使用CAD工具做更进一步的实验。在附录C和D中，将说明如何把使用CAD工具设计的电路从CPLD和FPGA芯片实现。

3.6.7 CPLD和FPGA的应用

目前CPLD和FPGA被应用在多种多样的电子产品中，例如，消费电子产品（DVD、高端电视机产品等）、汽车公司的控制设备和测试设备、因特网路由器和高速网络开关、大型磁带和硬盘储存系统等计算机设备。

当所需的电路规模不十分庞大时，或者在电源一旦通电，设备必须立刻开始正常工作的场合，选择CPLD是很合适的。而FPGA用在后一种情况不是十分合适，正如我们曾讲过的，FPGA是由可挥发的存储元件所配置的，所以一旦电源掉电，FPGA就可能丢失其储存的内容。由于FPGA的这个特点，所以当电源接通后，电路系统需要一点延迟时间，才能进入正常工作状态。

适合FPGA的电路，其规模有很大的伸缩性，从1000个到几百万个等效逻辑门规模的数字系统都适合用FPGA。除了规模大小之外，设计者还必须考虑对电路的许多其他要求，例如，电路运行速度的快慢，功率消耗的限制，以及芯片价格的高低等要素。当FPGA不能满足其中一个或者几个要求时，用户就可能选择客户定制的芯片来实现其电路，下面我们将讲解这一类芯片。

3.7 定制芯片、标准单元和门阵列

限制PLD电路规模的关键因素是PLD器件中含有许多可编程开关。虽然它带来了用户可编程的优点，但也消耗了芯片的面积（导致成本增加），并降低了电路的运行速度，增加了功

耗。本节将介绍不含可编程开关的集成电路技术。

包含逻辑门个数最多、运行速度最快、功率最低的芯片就是所谓的定制芯片。PLD器件中包含了许多预先制造的逻辑门和可编程开关，通过编程实现用户所需电路，而定制芯片是从草图开始设计的。客户定制芯片的设计者有很大的自由度，由自己决定电路的形式、芯片的规模、晶体管的数量以及晶体管在芯片中的位置及连接方式。定义晶体管在芯片上的位置以及相互连线的过程在技术上称为布局布线（layout）。定制芯片的设计者完全可以自由地创建自己希望得到的任何布局布线结果。由于芯片中所包含的晶体管个数可能超过数百万，因而设计定制芯片需要大量的精力和昂贵的成本。因此，只有在标准部件（如FPGA）满足不了需求的情况下，才使用客户定制芯片。而且，只有在销售量很大的情况下，才能补偿设计成本。最常见的客户定制芯片的例子是：微处理器和存储器芯片。

在定制芯片的设计者不需要自由地创建晶体管的布局布线的情况下，可以采用一种标准单元技术减少定制芯片的设计工作量，用这种技术制成的芯片通常叫做专用集成电路（ASIC）。图3-40所示的是使用标准单元技术制成的芯片的一小部分，图中的逻辑门按行排列，各行间是布线通道，逻辑门之间用布线通道内的线连接。一般说来，ASIC中使用的标准逻辑门有很多种，这些逻辑门事先存储在库中，供设计者随时选用。图3-40所示的连线有两种。图中的金属连线分布在不同的层上，所以它们虽然相互交叉却不会发生短路，灰色线表示金属线的一层，黑色线表示另外一层。每一个灰色线方块表示不同层的金属线的连接，叫做通孔。在目前的技术条件下，金属线的层数可达到8层甚至更多层。有些金属层可以置于逻辑门晶体管的顶层，从而使芯片的布局布线图更为有效。

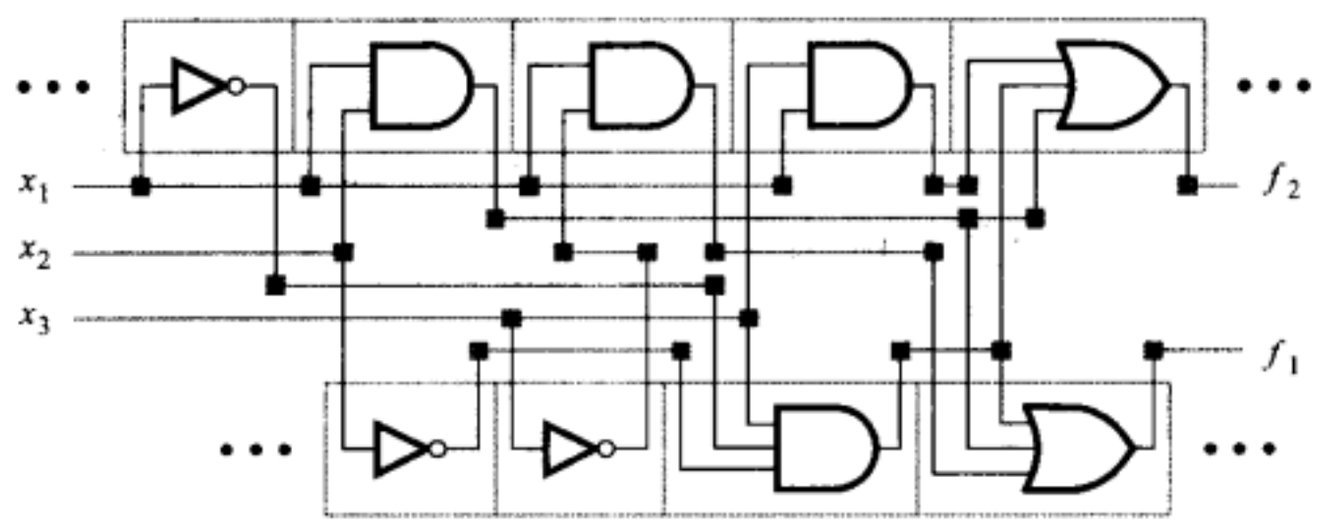


图3-40 标准单元芯片中的一部分

标准单元芯片与客户定制芯片很相似，它们都是根据用户的技术说明勾画的草图开始创建的。图3-40所示电路实现2个逻辑函数，与图3-26中的PLA所实现的逻辑函数相同，即：

$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$$

由于成本的原因，类似图3-40所示的小规模的电路一般不会应用标准单元电路制造，实际上，图3-40只是大规模电路的一小部分。标准单元内部晶体管的布图是预先设计好的并且已经确定，所以芯片的布图可以由CAD工具自动实现。芯片的典型安排是：逻辑门安排在很长的行中，行与行之间用来布线，I/O块分布在芯片的周围，用来连接封装的引脚。芯片的封装一般采用QFP、PGA或BGA。

另一种与标准单元相似的技术是门阵列技术。门阵列的一部分是预先制造的，而另一部分则是按客户要求定制的。提出这一概念的原因是：集成电路制造过程包含一系列步骤，其中有些步骤用来制造晶体管，而另一些步骤用来制造晶体管之间的连线。在门阵列技术中，生产厂商完成的大部分工作主要是晶体管的创建，而不考虑晶体管之间的连线。这部分工作

完成后的硅晶片（见图1-1）还是半成品，这种半成品硅晶片称为门阵列模板。然后对模板进行修改，主要是制造连线来连接晶体管，用来在芯片上完成用户电路。和客户定制芯片相比，门阵列技术的成本较低。生产商可以大量生产的芯片，因为硅晶片模版相同，分摊在每一片上的成本就降低了。门阵列技术的具体实现方法也有些不同，有的门阵列的逻辑单元规模较大，而另外一些规模较小，有的就在单个晶体管级别上配置。

图3-41所示是一个门阵列模板的例子，它由逻辑单元的二维阵列构成。芯片结构和标准单元类似，唯一的不同是门阵列中的逻辑单元都是相同的。虽然门阵列中所用的逻辑单元类型上可以有所变化，但通常使用的只是2或3输入的与非门。某些门阵列的逻辑单元行之间留有空间，以便以后布线连接逻辑单元。某些门阵列的单元行之间未留空间，此时布线区可放在逻辑单元的上层。这样做是可行的，正如在介绍图3-40时所说的那样，金属线可以处在芯片的不同层面上。人们称这种技术为门海（sea-of-gates）技术。图3-42所示的是门阵列的一小部分，实现的逻辑函数为：$f = x_2\overline{x}_3 + x_1x_3$。正如我们在2.7节中介绍的那样，仅使用与非门的电路和与或门电路可以是等价的，这种逻辑等价关系很容易得到验证。

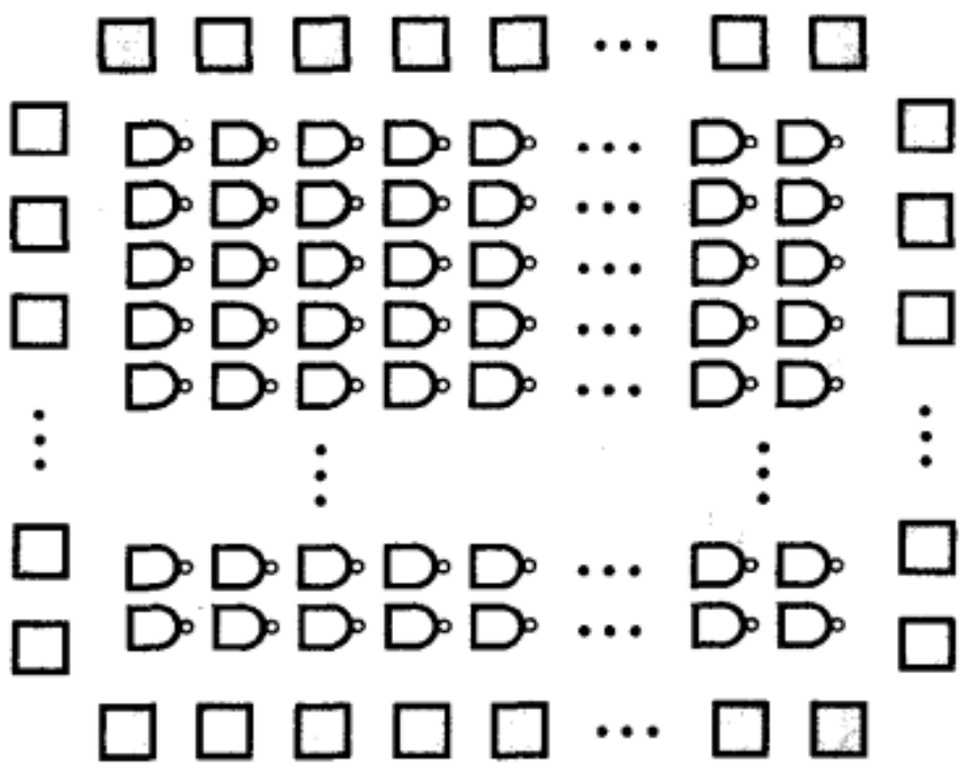


图3-41 采用门海技术的门阵列

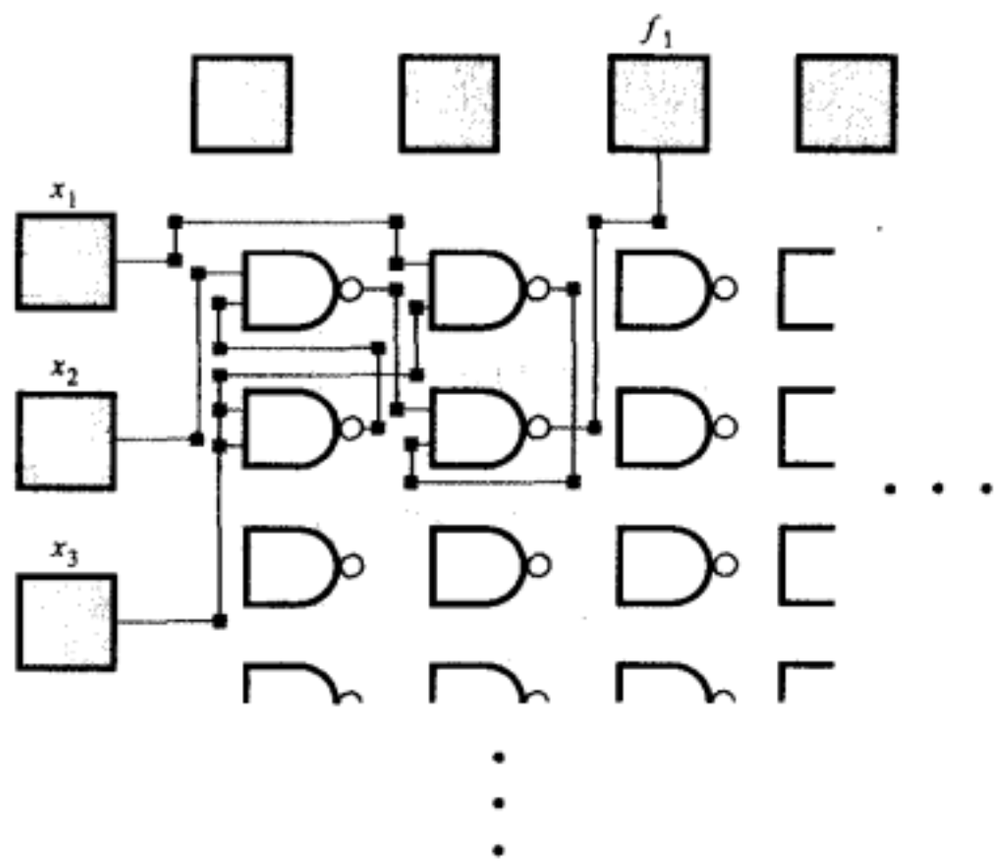


图3-42 图3-41所示的门阵列中实现逻辑函数 $f_1 = x_2\overline{x}_3 + x_1x_3$

3.8 实际问题

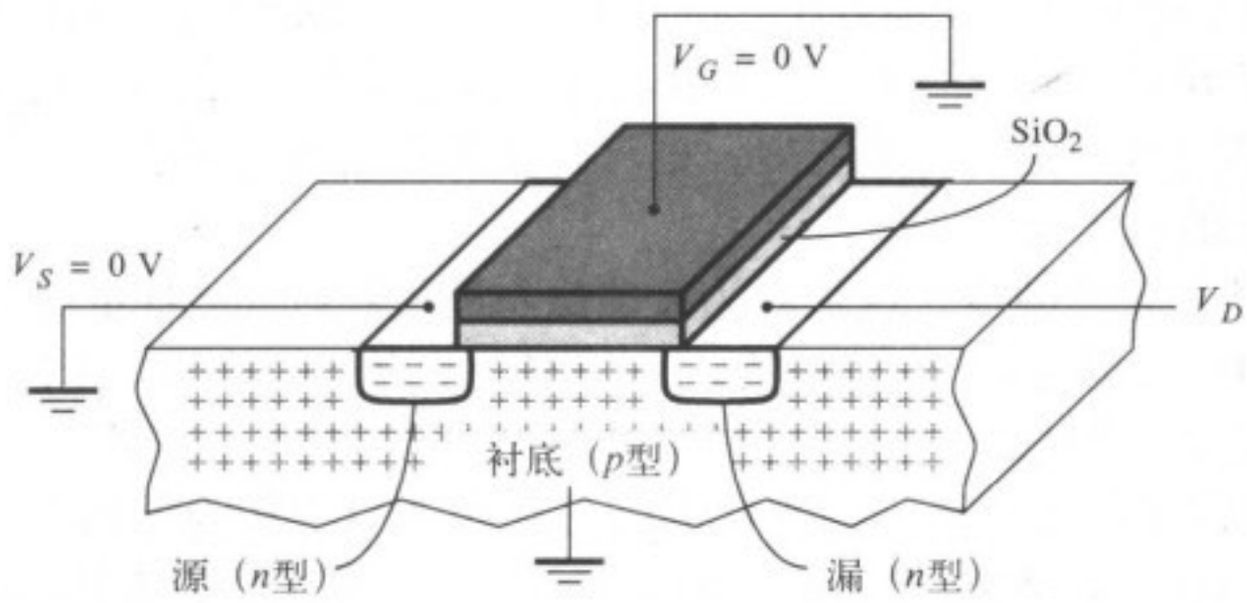
到目前为止，在本章中我们已经介绍了逻辑门电路的若干基本特性，并给出了市场上销售芯片的一些例子。本节将提供有关数字电路几个方面的更详细的资料。我们将描述如何在硅片上制造晶体管，并详细地解释晶体管是如何工作的。我们还将介绍逻辑电路的鲁棒性（robustness）问题，以及探讨逻辑门的信号传播延迟和功率消耗等重要问题。

3.8.1 MOSFET晶体管的制造和行为

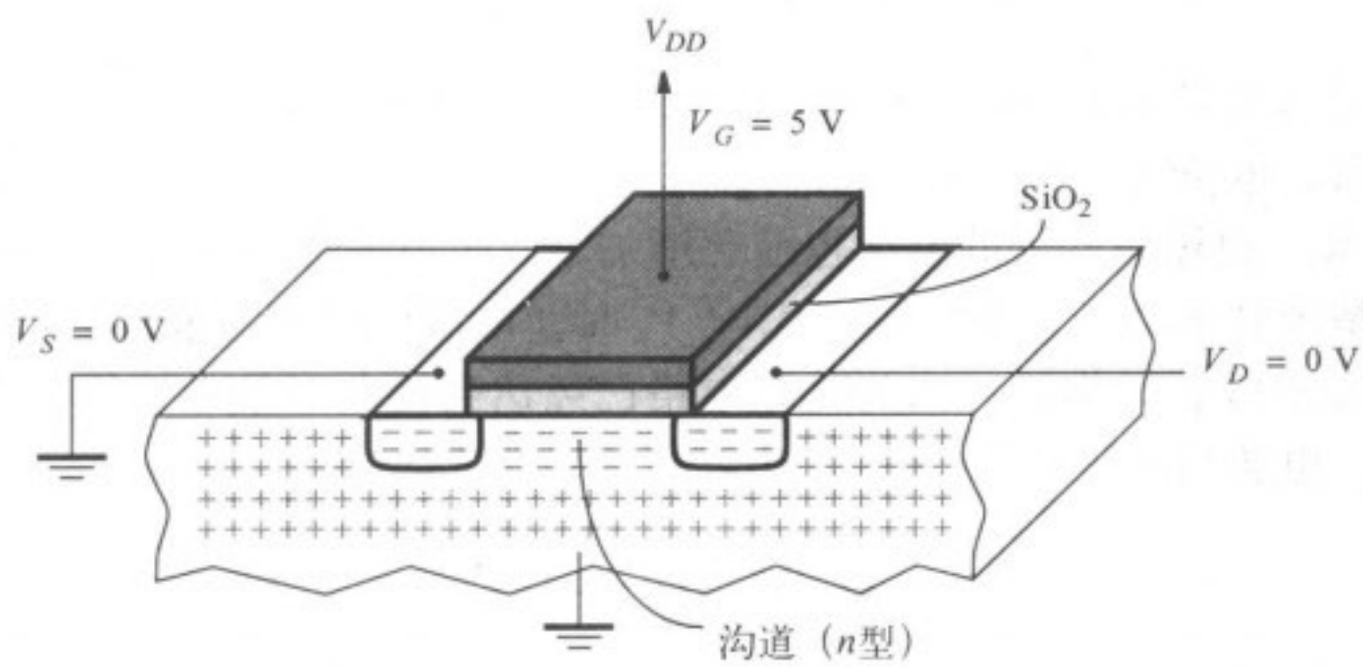
为了理解NMOS和PMOS晶体管的工作原理，我们需要理解如何将晶体管制造在集成电路上。大家都知道，集成电路是制造在硅圆晶片上的。硅圆晶片（见图1-1）的直径通常在6、8或12英寸左右，有点类似于音乐光盘（CD）。在一薄片硅圆晶上可以制造许多个集成电路芯片，然后将其切割成单个芯片。

硅的电气性能属于半导体，这意味着硅有时候表现为电流的导体，而有些时候又表现为电流的非导体。制造晶体管实际上是在硅衬底上制造出一些区域，使这些区域含有过多的正

电荷或者负电荷。含负电荷多的区域叫n型，而含正电荷多的区域叫p型。图3-43说明了NMOS晶体管的结构。该晶体管的源极和漏极都用n型硅材料，而基片衬底用p型硅材料。金属线用于实现与源极和漏极的电气连接。



a) 当$V_{GS} = 0$ V时，晶体管断开



b) 当$V_{GS} = 5$ V时，晶体管接通

图3-43 NMOS晶体管的物理结构

当初金属氧化物半导体场效应晶体管（MOSFET）发明时，栅极是用金属制造的，现在则用一种称为多晶硅的材料制造。多晶硅和金属一样是一种导体，但多晶硅之所以比金属好，是因为在MOSFET的制造过程中，它所占用的面积极小，且制造性能优良。栅极在电气上通过一层二氧化硅（SiO_2）与晶体管的其他部分绝缘，二氧化硅是一种玻璃材料，起到栅极与衬底之间电气绝缘的作用。晶体管的运行状态受到施加到晶体管各极的电压所形成电场的控制，下面我们再详细讨论。

如图3-43所示，加在晶体管源极、栅极和漏极上的电压分别标记为：V_S，V_G 和V_D。考虑如图3-43a所示的第一种情况，源极和栅极全都连接到地（即$V_S = V_G = 0$ V）。n型的源极和n型的栅极互相之间由p型衬底隔开。根据电学原理，在源极和漏极之间形成了两个二极管。第一个二极管是由衬底和源极之间的$p-n$结构成；第二个二极管是由衬底和漏极之间的$p-n$结构成。这两个背对背的二极管在源极和漏极之间形成了非常高的电阻（约10^{12}欧姆[1]），阻止了电流的流动。我们就将这种情况称为晶体管处于断开或截止状态。

接下来，考虑增加栅极和源极之间的电压。令V_{GS}表示栅极和源极之间的电压。若V_{GS}大于某个阈值电压V_T（最小正电压），则晶体管就从截止（断开的开关）转变成导通（闭合的开关），下面再进行解释。阈值电压V_T的准确值取决于许多因素，一般情况下，大约是V_{DD}的0.2倍。

图3-43b展示了当$V_{GS} > V_T$情况下的晶体管状态。栅极连接到V_{DD}，使得$V_{GS} = 5$ V。栅极上的正电压吸引存在于（n型）源极以及晶体管其他区域中的自由电子趋向于栅极。因为这些电子不能穿越栅极下的绝缘层，因而在衬底上位于源极和漏极之间的区域聚集起来，该区域称为沟道（channel）。由于电子聚集在沟道区，使该区域由p型转变为n型，从而将源极和漏极有效地连接起来。沟道的尺寸由栅极的长度和宽度决定。沟道的长度L是位于源极和漏极之间的栅极的长度，沟道的宽度W则是栅极的宽度。也可以认为沟道还有深度，该深度取决于施加在源极、栅极和漏极上的电压大小。

没有电流能流过晶体管的栅极，这是因为玻璃层将栅极和衬底绝缘的缘故。电流I_D可以从漏极流到源极。对于一个固定的V_{GS}电压值，$V_{GS} > V_T$，电流I_D的值取决于沟道两侧，即漏-源之间所施加的电压V_{DS}。若$V_{DS} = 0$ V，则没有电流流过。随着V_{DS}的增大，I_D也随之增大，与所施加的电压V_{DS}大致成线性关系，只要V_D的值足够小，至少在沟道的漏极端提供V_T，即$V_{GD} > V_T$。在这个电压范围里，即$0 < V_{DS} < (V_{GS} - V_T)$，晶体管工作在三极管工作区，又叫做线性工作区。电压和电流的关系可近似地用等式（3.1）表示。

$$I_D = k_n' \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \tag{3.1}$$

符号k_n'叫做过程跨导参数（process transconductance parameter），它是一个常数，取决于所采用的工艺技术，单位为A/V^2。

随着V_D的增大，通过晶体管的电流也随之增加，此时电流值由式（3.1）给定，但电流增加到一定值后，情况就不同了。当$V_{DS} = V_{GS} - V_T$时，电流已经达到最大值。对于更大的V_{DS}，晶体管不再工作在三极管工作区。因为此时电路已经达到饱和（最大）值，我们称晶体管处在饱和区。此时，电流与电压V_{DS}无关，I_D由下面的表达式给定：

$$I_D = \frac{1}{2}k_n' \frac{W}{L}(V_{GS} - V_T)^2 \tag{3.2}$$

图3-44所示的是NMOS晶体管在固定的V_{GS}电压值：$V_{GS} > V_T$的情况下，电流和电压的关系。图3-44指出了晶体管离开三极管工作区开始进入饱和区的点，该点出现在：$V_{DS} = V_{GS} - V_T$。

例 3.3 假设$k_n' = 60\ \mu A/V^2$，$W/L = 2.0\ \mu m/0.5\ \mu m$，$V_S = 0$ V，$V_G = 5$ V，$V_T = 1$ V。若$V_D = 2.5$ V，则由式（3.1）计算出晶体管电流$I_D \approx 1.7$ mA；若$V_D = 5$ V，由式（3.2）计算出晶体管饱和电流$I_D \approx 2$ mA。 ■

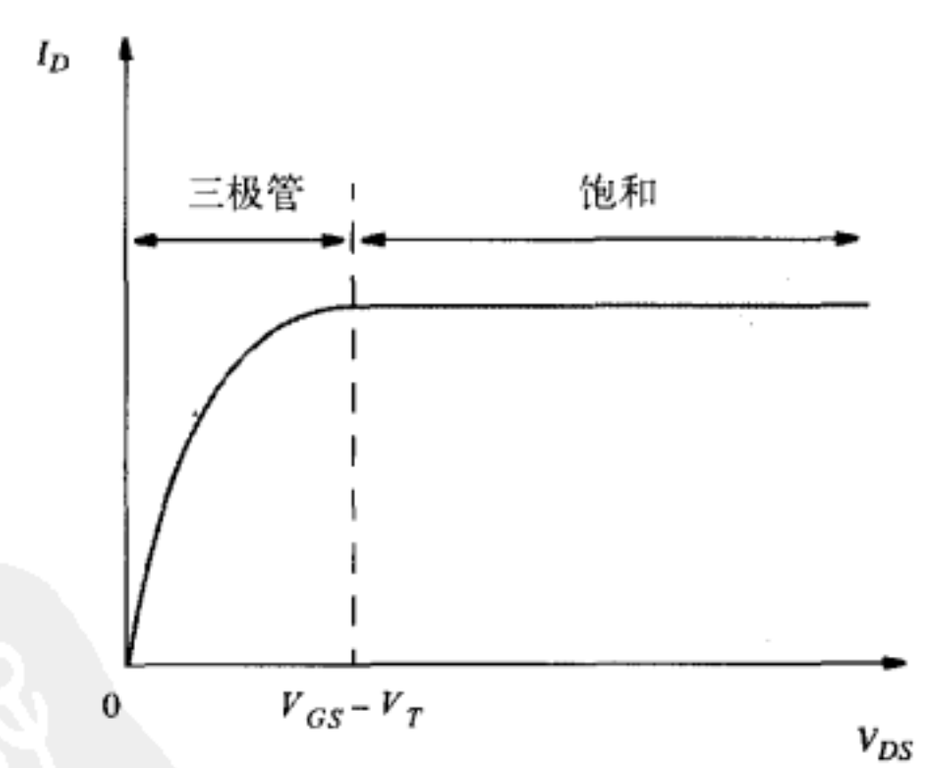


图3-44 NMOS晶体管的电流电压关系

PMOS晶体管

PMOS晶体管与NMOS晶体管的行为相同，区别在于电压和电流的方向都正好相反。PMOS晶体管源极电平较高，而漏极较低（NMOS晶体管源极电平较低，而漏极较高），使PMOS晶体管导通的阈值电压是一个负值。PMOS晶体管的物理结构和NMOS晶体管相似，区别在于NMOS晶体管中的n区对应于PMOS晶体管的p区；NMOS晶体管中的p区对应于PMOS晶体管的n区。若想使PMOS晶体管截止，图3-43a上源极和栅极都应该接到V_{DD}。若想使PMOS晶体管导通，就要把图3-43b栅极接地，使$V_{GS} = -5$ V。

因为沟道采用了p型硅而不是n型硅，所以PMOS晶体管中导电的物理机制和NMOS晶体管

不同。关于这个问题的详细讨论已超出本书的范围，但这里必须指出由此而引起的问题。式（3.1）和式（3.2）都用到参数k_n'，PMOS晶体管中的对应参数是k_p'，但是电流流过n型硅比流过p型硅容易，因此在一般工艺技术下$k_p' \approx 0.4 \times k_n'$。若想使PMOS晶体管的导电能力与NMOS晶体管的相同，则必须使PMOS晶体管的W/L增大到NMOS晶体管的2～3倍。在实现逻辑门时，通常要考虑NMOS和PMOS晶体管的大小因素。

3.8.2 MOSFET晶体管的导通电阻

在3.1节中，我们把MOSFET作为理想开关。当开关断开时电阻为无穷大；而开关导通时电阻为0。但是对于实际的晶体管而言，晶体管导通时沟道的电阻是V_{DS}/I_D，叫做导通电阻。用式（3.1）可以计算出晶体管工作在三极管状态下的导通电阻，见例3.4所示。

例3.4 考虑一个CMOS反相器，其输入电压V_x为5 V，NMOS晶体管导通，输出电压V_f接近于0 V。因此，NMOS晶体管的V_{DS}接近于0 V，晶体管工作于三极管区。见图3-44所示的曲线，此时NMOS晶体管工作在接近于原点的位置。虽然V_{DS}的值非常之小，但并非精确的0值。在下一节中，我们将解释V_{DS}的值一般约为0.1 mV，因此I_D也不正好为0，I_D的值可以由式（3.1）求得。因为V_{DS}的值非常小，所以我们可以忽略V_{DS}^2。此时，导通电阻可以近似地表示为：

$$R_{DS} = V_{DS} / I_D = 1 / \left[k_n' \frac{W}{L} (V_{GS} - V_T) \right] \tag{3.3}$$

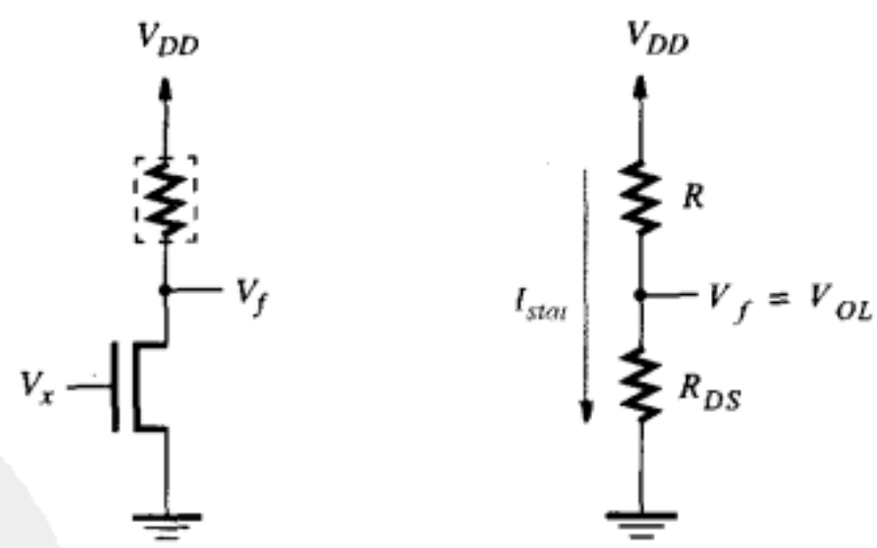
假设$k_n' = 60\mu A/V^2$，W/L=2.0μm/0.5μm，$V_{GS} = 5$ V，V_T =1 V。可以得到$R_{DS} \approx 1$ kΩ。 ■

3.8.3 逻辑门的电平

在图3-1中我们曾展示逻辑值可以由电平的范围来表示。现在让我们更仔细地考虑逻辑电平的问题。

基本反相器的操作可以用来表示逻辑系列中高、低电平的性质。我们在图3-45a中重新绘制了图3-5所示的用NMOS工艺制造的反相器电路。当$V_x = 0$ V时，NMOS晶体管截止，没有电流流过，因此V_f = 5 V。当$V_x = V_{DD}$时，NMOS晶体管导通。为了计算此时的V_f值，我们可以用一个电阻R_{DS}代替 NMOS晶体管，如图3-45b所示。由电阻分压原理得到：

$$V_f = V_{DD} \frac{R_{DS}}{R_{DS} + R}$$



a) NMOS 非门　　b) $V_x = 5$ V

图3-45 NMOS 反相器中的电平

例3.5 假设R = 25 kΩ，用例3.4中的结果，R_{DS}= 1 kΩ，得到$V_f \approx 0.2$ V。

如图3-45b所示，在$V_x = V_{DD}$的静态条件下，流过NMOS反相器的电流I_{stat}的值是：

$$I_{stat} = V_f / R_{DS} = 0.2 \text{ V} / 1 \text{ k}\Omega = 0.2 \text{ mA}$$

这个静态电流有很重要的含义，我们将在3.8.6节中讨论。

在现代NMOS电路中，上拉电阻通常是用PMOS晶体管实现的，这种电路被称为伪NMOS电路。这种电路和CMOS电路完全兼容，由此，在单个芯片中既可以包含CMOS门又可以包含伪NMOS门。例3.13展示了伪NMOS反相器电路，并讨论如何计算该反相器的输出电平。 ■

CMOS反相器

通常用符号V_{OH}和V_{OL}分别表示逻辑电路的高/低电平，V_{OH}表示输出为高电平，同样，V_{OL}表示输出为低电平。对于上面讨论过的NMOS反相器，$V_{OH}=V_{DD}$，而V_{OL}大约是0.2 V。

让我们重新考虑图3-12a所示的CMOS反相器，它的输入—输出电压关系可用图3-46所示的电压转移特性来表示。该曲线给出了每个V_x值下的V_f的稳态值。当$V_x=0$ V时，NMOS晶体管截止，没有电流流过，因而$V_f=V_{OH}=V_{DD}$；当$V_x=V_{DD}$时，PMOS晶体管截止，没有电流流过，此时$V_f=V_{OL}=0$ V。我们应当指出，严格地说，即使在晶体管截止时，也会有称为漏电流的极小电流流过。这个漏电流对于V_{OH}和V_{OL}的影响非常小，举个例子，V_{OL}的典型值是0.1 mV，而不是0 V[1]。

如图3-46所示，在输出电平由高向低（反之亦然）的转变点处作了标记。电压V_{IL}点表示输出电平为高，且曲线在该点的斜率等于−1。V_{IL}表示的是输入电平为低的最大值，此时，反相器的输出电平仍为高。与此类似，V_{IH}是曲线上另一个斜率等于−1的点，V_{IH}表示的是输入电平为高的最小值，反相器在该处的输出电平仍为低。参数V_{OH}，V_{OL}，V_{IL}和V_{IH}是衡量逻辑电路健壮性的重要参数，下面我们将详细讨论。

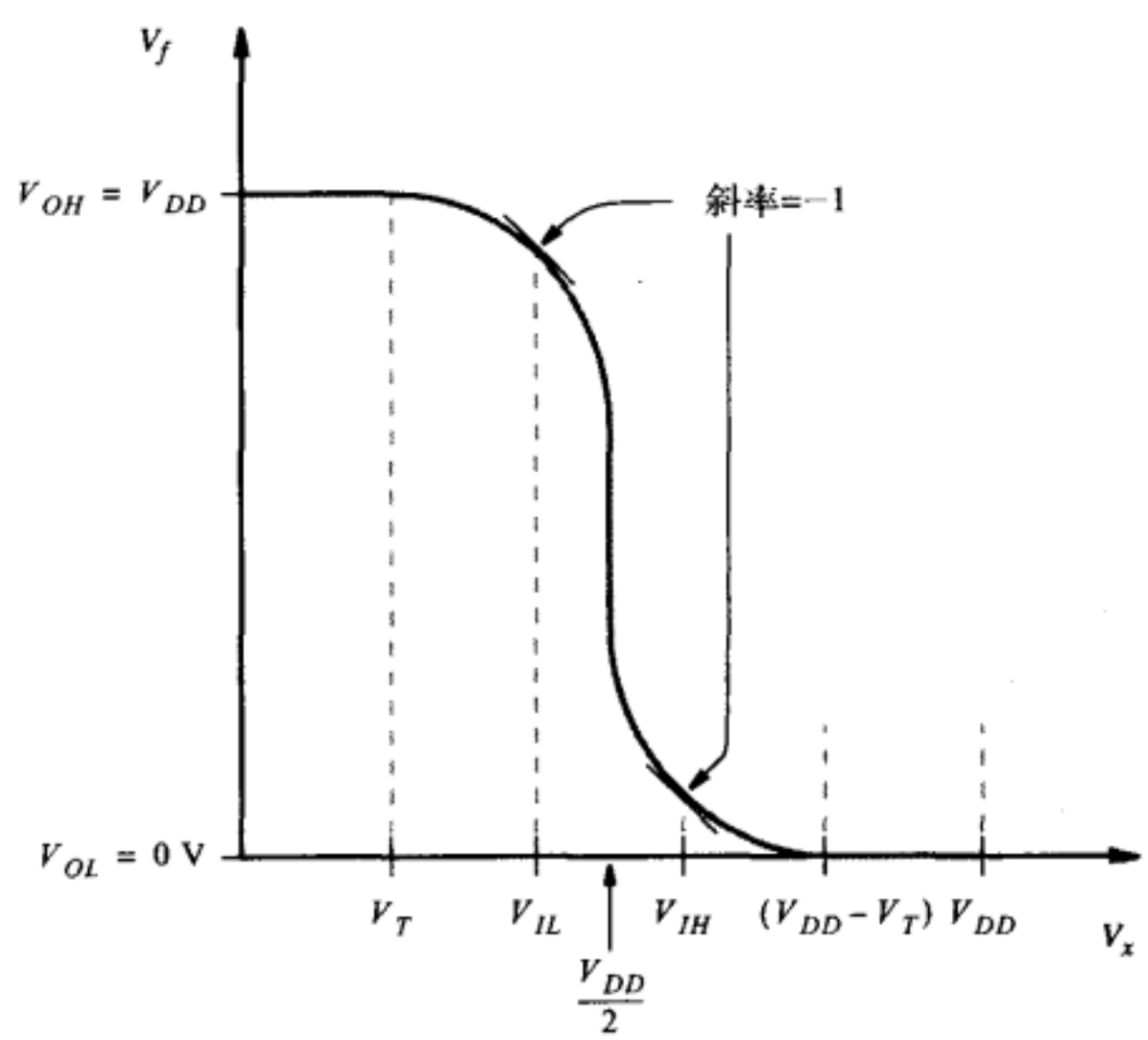


图3-46 CMOS反相器的电压转移特性

3.8.4 噪声容限

考虑如图3-47a所示的两个反相器，令左边和右边的反相器分别记为N_1和N_2。电子线路经常会受到随机的干扰，这种噪声干扰会影响反向器N_1输出电平的改变。最重要的是，这种噪声不要影响N_2把由N_1输入的低电平误当做高电平（或者把高电平误当做低电平）。考虑N_1输出为低电平V_{OL}的场合，噪声的存在可能影响输出电平的实际值，但只要它保持小于V_{IL}，N_2就能解释输入信号，产生正确的输出。允许噪声在一定限度内的存在，而不影响电路正常工作的能力称为噪声容限。对于低电平输出的情况，低电平噪声容限定义为：

$$NM_L=V_{IL}-V_{OL}$$

当N_1产生高电平输出V_{OH}时，也存在类似的情况。电路中任何噪声的存在均可影响该电路

输出电平的实际值，但是只要输出电平保持大于V_{IH}，N_2就可以正确解释。高电平噪声容限定义为：

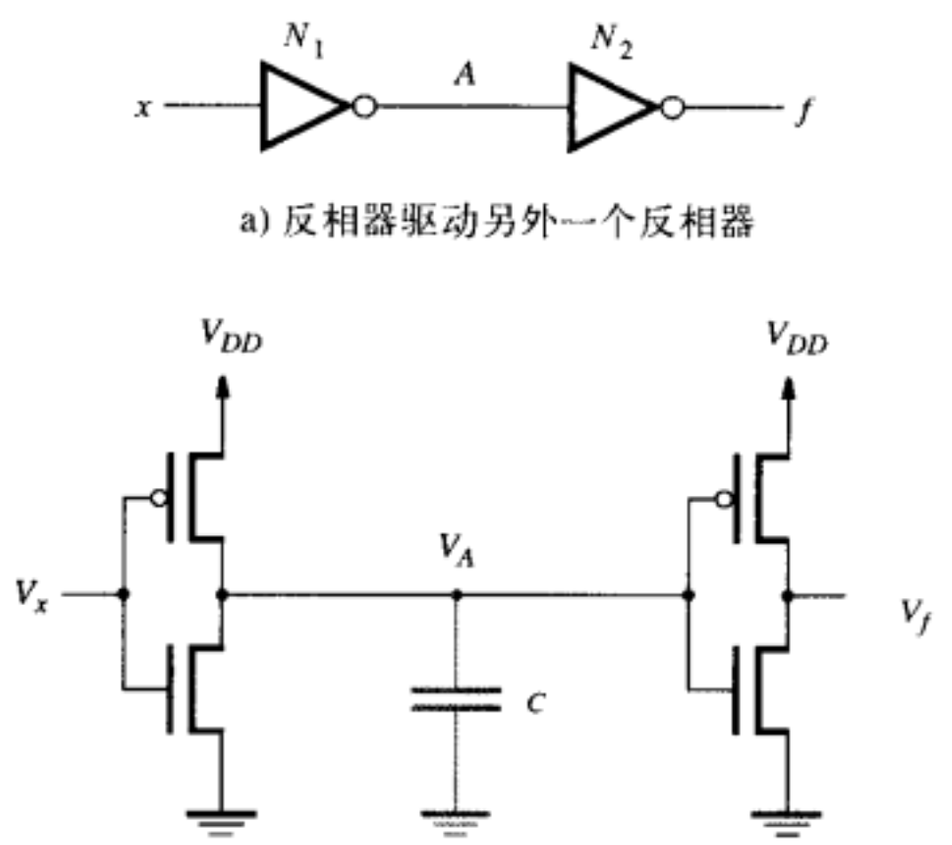
$$NM_H = V_{OH} - V_{IH}$$

例3.6 若给定制造工艺技术，则基本反相器的电压转移特性决定了V_{OH}，V_{OL}，V_{IL}和V_{IH}的电平值。对于图3-46所示的CMOS反相器的电压转移特性，$V_{OH} = V_{DD}$，$V_{OL} = 0$ V。通过找到转移特性上斜率等于−1的两点，可以得出[1]：$V_{IL} \approx \frac{1}{8}(3V_{DD} + 2V_T)$，以及$V_{IH} \approx \frac{1}{8}(5V_{DD} - 2V_T)$。

V_T的典型值是：$V_T = 0.2\ V_{DD}$，因此得到：

$$NM_L = NM_H = 0.425 \times V_{DD}$$

由此可知噪声容限和电源电压有关。当$V_{DD} = 5$ V时，噪声容限是2.1 V；当$V_{DD} = 3.3$ V时，噪声容限是1.4 V。 ■



a) 反相器驱动另外一个反相器

b) 节点A的电容性负载

图3-47 集成电路中的寄生电容

3.8.5 逻辑门的动态操作

在图3-47a中，位于两个反相器之间的节点记作A。因为硅片中晶体管的构造方式，N_2对节点A起一个电容负载的效应。图3-43表明晶体管由不同材料的许多层构成。只要在晶体管内部有两种材料相接或重叠在一起，就会形成电容。这种电容叫做*寄生电容*，因为这种电容是制造过程产生的副作用，不是我们所希望产生的。在图3-47中，我们对节点A处的电容感兴趣，在该节点连接着很多寄生电容，其中一部分由N_1引起，而另一部分是由N_2引起的。其中反相器N_2的输入端和地之间的寄生电容的影响比较显著。寄生电容的值与非门N_2的晶体管尺寸有关。每个晶体管都有一个*栅极电容*，$C_g = W \times L \times C_{ox}$，参数$C_{ox}$称为*氧化层电容*，它是一个和工艺有关的常数，单位是fF/μm²。其他的电容是由晶体管N_1和与节点A相连的金属线引起的。这些寄生电容可以用单独一个在节点A和地之间的等价电容来表示[2]。在图3-47b中，这个等价电容标记为C。

寄生电容的存在给逻辑电路的运行速度带来负面的影响。电容两端的电压不能立即变化，电容充电或放电所需要的时间取决于电容的尺寸和流经电容的电流的大小。在图3-47b所示的电路中，当N_1的PMOS晶体管导通时，电容被充电到V_{DD}；当NMOS晶体管导通时，电容放电。在这两个过程中，充/放电的速度取决于流经相关晶体管的充/放电流I_D和电容C的大小。

在第2章中我们介绍了时序图的概念，图2-10所示的是理想的波形图，其中显示的波形当从低电平转移到高电平（或相反）时是完美的垂直线。而在实际电路中，波形不可能有这样理想的形状，而是图3-48中所示的形状。图3-48所示的是输入到图3-47b电路的V_x端的信号波形图，以及节点A处产生的结果V_A的波形。假定V_x的初始状态处于高电平V_{DD}，然后转到低电平0 V。一旦V_x电平达到足够低时，N_1就开始驱动V_A向V_{DD}转移。由于寄生电容的影响，V_A不能立即变化，而是如图所示地逐渐变化。V_A由低向高变化所需的时间叫做*上升时间*t_r，定义为V_{DD}从10%上升到90%所需要的时间。图3-48还定义了由V_x的变化所引起的V_A变化所需的全部时间，这叫做*传播延迟*，通常用t_p表示。它是从V_x变到V_{DD}的50%算起到V_A达到相同的电平（V_{DD}的50%）所需的时间。

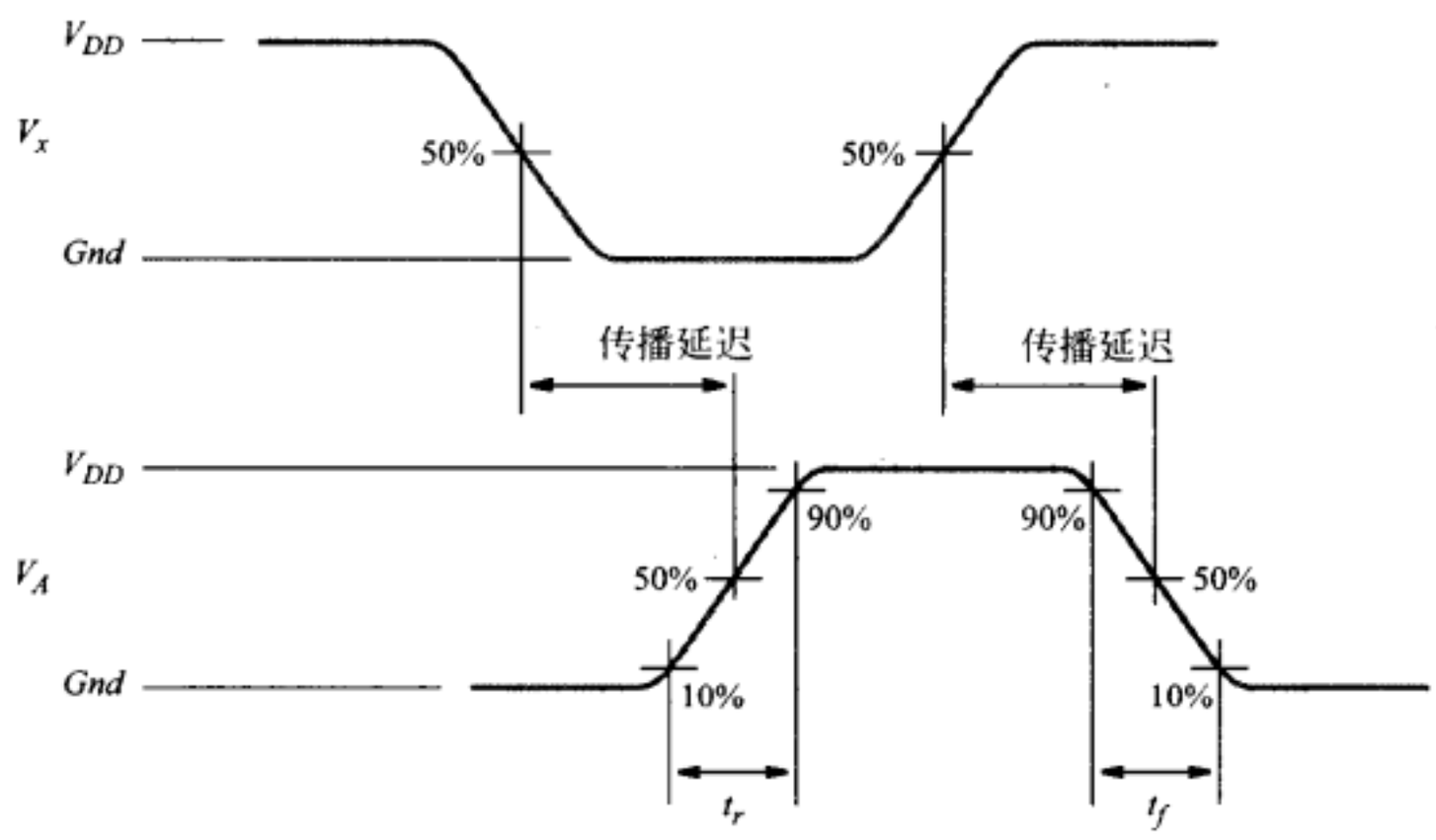


图3-48 逻辑门的电平波形

V_x保持在0 V一段时间后，然后改变为V_{DD}，使得N_1将寄生电容C放电至地。在这种情况下，节点A的电平从高逐渐转变为低，所需要的电平转变时间为下降时间t_f，即从V_{DD}的90%下降到V_{DD}的10%所需要的时间。正如图上所示的，V_x的新变化影响到V_A的变化有一个对应的传播延迟。在给定的逻辑门电路中，PMOS和NMOS晶体管的相对大小通常需要做仔细的调整，使得上升时间t_r和下降时间t_f的值相同。

式（3.1）和式（3.2）指出了流经NMOS晶体管的电流大小。假设图3-47中电容的值给定，则可以计算出V_A由高电平逐渐转为低电平的传播延迟。为简化计算，假定V_x的初始值是0 V，因此PMOS晶体管导通；$V_A = 5$ V。然后V_x在0时刻变到V_{DD}，使PMOS晶体管截止，NMOS晶体管导通。传播延迟就是电容C经NMOS晶体管而放电，电平下降到$V_{DD}/2$所需的时间。当V_x刚刚到达V_{DD}，此时$V_A = 5$ V；因此NMOS晶体管会有$V_{DS} = V_{DD}$，晶体管将进入饱和区，电流I_D由式（3.2）求得。一旦V_A下降到$V_{DD} - V_T$以下，NMOS晶体管将进入三极管工作区，此时电流I_D由式（3.1）给定。我们的目的是近似估算V_A由V_{DD}下降到$V_{DD}/2$时的电流，即算出在$V_{DS} = V_{DD}$情况下用式（3.2）计算出的电流I_D值，和在$V_{DS} = V_{DD}/2$情况下用式（3.1）计算出的电流I_D值，求二者的平均值。使用求电容充电需要时间的基本表达式（见例3.11），我们得到：

$$t_p = \frac{C\Delta V}{I_D} = \frac{CV_{DD}/2}{I_D}$$

在上式中，代入式（3.2）表示的I_D，可以得到：

$$t_p \cong \frac{1.7C}{k_n' \frac{W}{L} V_{DD}} \tag{3.4}$$

上面的表达式说明电路的速度不但取决于寄生电容C，也取决于NMOS晶体管的尺寸。减小C或者增大W/L的值，可以减小电路的传播延迟。此表达式确定了输出由高向低转变时的传播延迟。输出由低向高转变时的传播延迟也可以这样计算，但使用的是PMOS晶体管的参数k_p'和W/L。

在逻辑电路中，晶体管长度L通常选用在所用工艺技术允许前提下的最小值。宽度W的选取通常根据电流的大小以及期望的传播延迟。图3-49说明了不同尺寸的两个晶体管。图3-49a中的晶体管的尺寸较小，可用在电容负载较小或速度要求不高的电路上。而图3-49b所示的是

一个尺寸较大的晶体管，L虽然与图3-49a的晶体管相同，但W变大了。晶体管尺寸的选择必须权衡利弊，因为尺寸大的晶体管比尺寸小的需要更大的芯片面积；增大W虽然可以增大流过晶体管的电流，但也会增大寄生电容的值（栅极对地的电容C_g和W/L成正比），因此，抵消了一部分期望的性能改善。所以，在逻辑电路中，当有很大的电容负载必须驱动且传播延迟必须减小时，才选用尺寸较大的晶体管。

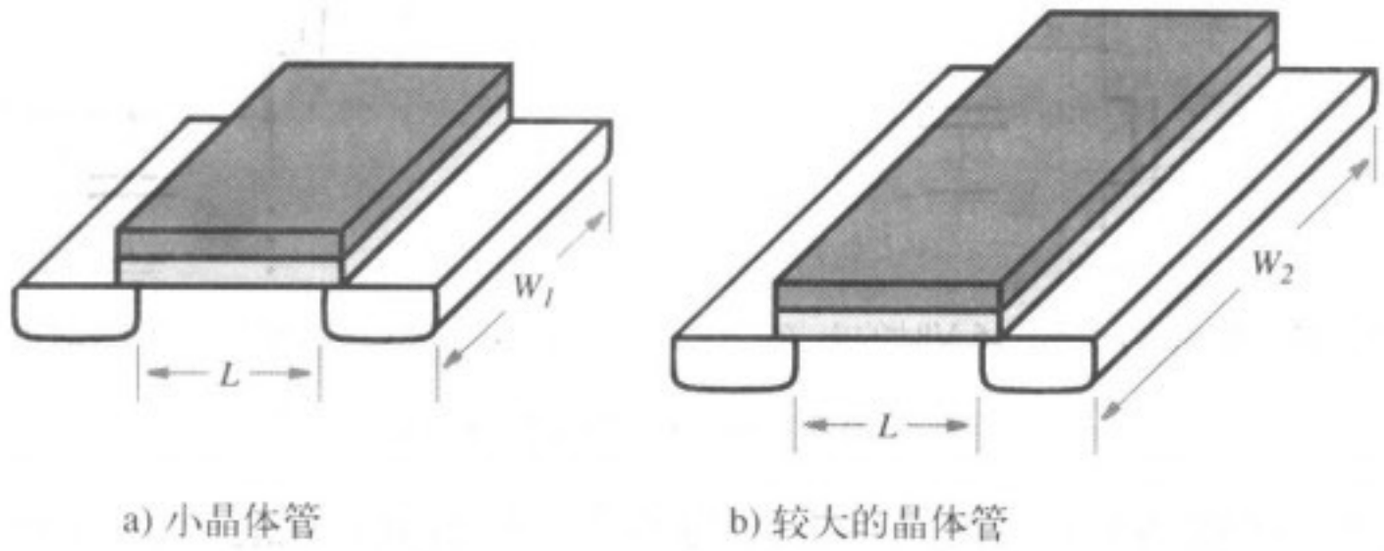


图3-49 晶体管的大小

例3.7 在图3-47所示的电路中，假定C = 70 fF，W/L = 2.0μm/0.5μm。k_n' = 60 μA/V^2，V_{DD} =5 V。应用式（3.4），可求出反相器由高到低的传播延迟为：$t_p \approx 0.1$ ns。 ■

3.8.6 逻辑门的功率消耗

在电子线路设计中，考虑晶体管所消耗的功率是极其重要的。集成电路制造技术已经允许在单个芯片中制造数百万枚晶体管，因此每个晶体管的功耗必须极其小。在任何一种逻辑电路的应用中，功耗都是必须考虑的重要问题，尤其对电池供电的设备更是至关重要，例如便携式计算机等产品。

让我们再一次考虑如图3-45所示的NMOS反相器，当V_x = 0 V时，没有电流流过，因而功耗为0。当V_x = 5 V时，由于电流I_{stat}的存在而产生功耗。在稳态情况下，功耗$P_s = I_{stat}V_{DD}$。在例3.5中，我们计算出I_{stat} = 0.2 mA，此时的功耗P_s = 0.2 mA × 5 V = 1.0 mW。若某芯片包含1万个反相器，则总功耗可高达10 W！正因为功耗如此巨大，NMOS类型的门只能用于某些特殊的场合，关于这一点我们将在3.8.8节中介绍。

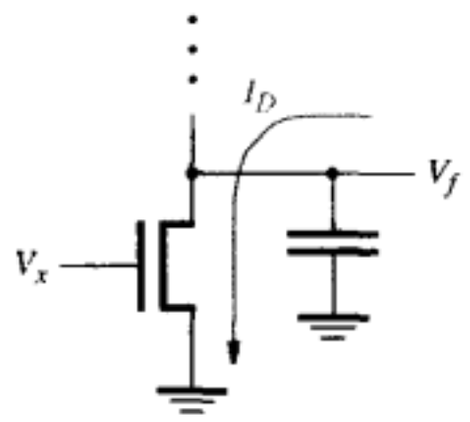
为了区分反相器处于稳态情况下的功耗和信号电平正发生变化时的功耗，习惯上把功耗分成两种类型：*静态功耗*和*动态功耗*，前者由稳态时的电流引起，后者由信号电平发生变化时的电流引起。NMOS晶体管既有静态功耗又有动态功耗，而CMOS晶体管只有动态功耗。

考虑图3-12a所示的CMOS反相器。当输入电平V_x是低电平时，没有电流，因为NMOS晶体管截止；当输入电平V_x是高电平时，也没有电流，因为PMOS晶体管截止。因此，在稳态情况下，CMOS电路没有电流。但是，在信号由一个电平向另一个电平变化的一小段时间里，电路中确实有电流流过。

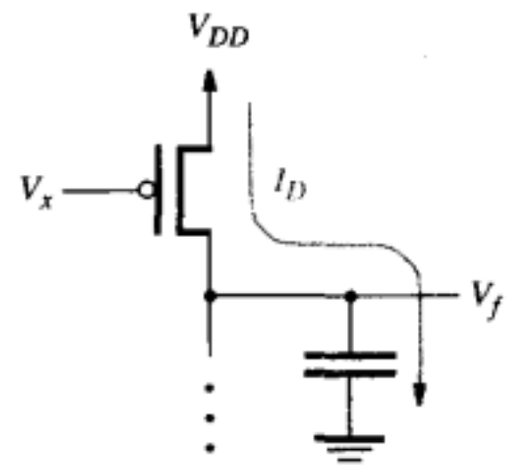
如图3-50a所示描述了下述情况。假定V_x已经保持0 V一段时间，因此V_f = 5 V。现在让V_x变到5 V，于是NMOS晶体管导通，将V_f下拉到0 V。由于节点f处的寄生电容C的存在，电压V_f不会立即发生变化，并且电容C放电的一段时间内电流I_D流经NMOS晶体管。与此类似，当V_x从5 V向0 V变化时如图3-50b所示：电容C两端的电压初始值为0 V，然后逐渐向5 V变化。由于PMOS晶体管的导通，对电容C充电期间，有电流从电源流经PMOS晶体管，直到V_f = 5 V为止。

由图3-46所示的CMOS反相器的电压传输特性可知，输入电压V_x处于某个小范围时两个晶

体管都导通。在这个电压范围里，即$V_T < V_x < (V_{DD} - V_T)$，电流从$V_{DD}$开始经过两个晶体管流向地，一般称这个电流为逻辑门中的短路电流。在大多数情况下，和对电容C的充电（放电）电流相比，这个短路电流极小，可以忽略不计。



a) 当输入V_x从0 V改变到5 V时的电流



b) 当输入V_x从5 V改变到0 V时的电流

图3-50 CMOS电路中的动态电流

单个CMOS反相器的功耗是很小的。让我们再一次看图3-50a，当$V_f = V_{DD}$时，存储在电容里的能量是$CV_{DD}^2/2$（见例3.12）；当电容放电到0 V时，电容所存储的能量消耗在NMOS晶体管中。图3-50b所示的情况与此类似，当给C充电到V_{DD}时，能量$CV_{DD}^2/2$消耗在PMOS晶体管中。在反相器充放电的一个周期中，能量的总消耗为CV_{DD}^2。因为功率的定义是单位时间内的能量消耗，所以功耗就是单位时间内的能量消耗和每秒钟充/放电周期的次数 f 的乘积。因此反相器的动态功耗为：

$$P_D = f\,CV_{DD}^2$$

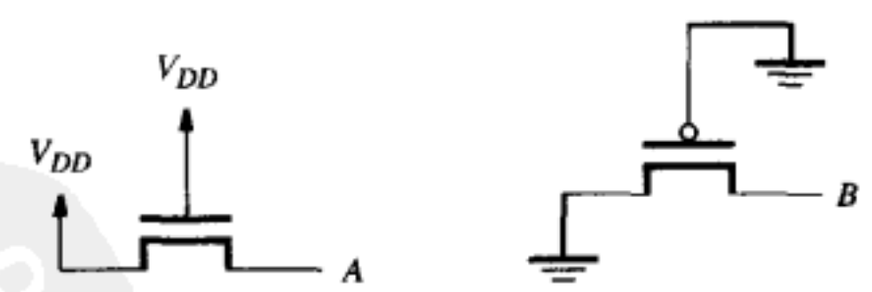
实际上，CMOS电路总的动态功耗显著地小于其他工艺技术电路的总功耗，例如NMOS电路。正因为这个原因，现在的大规模集成电路几乎都采用CMOS技术制造。

例3.8 对于CMOS反相器，假定C =70 fF，f =100 MHz，门的动态功耗P_D = 175 μW。假定某芯片含有相当于1万个反相器的门电路，平均有20%的门电路在任意给定时刻改变值，则该芯片的总动态功耗为$P_D = 0.2 \times 10\,000 \times 175\mu W = 0.35$ W。 ■

3.8.7 通过晶体管开关传递1和0

由图3-4所示的例子可以看出，NMOS晶体管被用作下拉器件，而PMOS晶体管被用作上拉器件。现在我们考虑以相反的方式使用这两种晶体管：用NMOS晶体管驱动输出高；用PMOS晶体管驱动输出低。

如图3-51a所示的是NMOS晶体管的一个例子，晶体管的栅极和开关的一端都连接到V_{DD}。假定栅极的初始电平V_G和节点A的初始电平均为0 V，然后让V_G变到5 V。节点A是晶体管的源极，因为它的电平最低。因为$V_{GS} = V_{DD}$，所以晶体管导通，并驱动节点A的电平向V_{DD}变化。当节点A的电平上升时，V_{GS}的值下降，当V_{GS}的值下降到小于V_T时，晶体管截止。此时，晶体管进入稳定状态$V_A = V_{DD} - V_T$，这意味着NMOS晶体管只能部分地传输高电平信号。



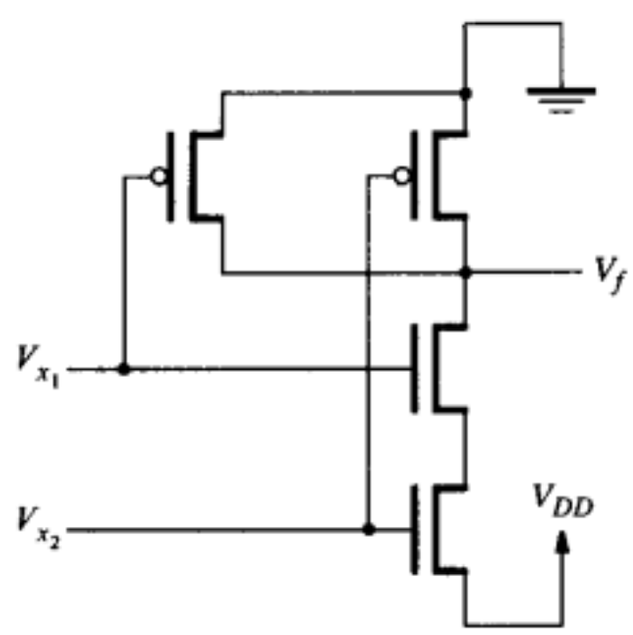
a) NMOS晶体管　　b) PMOS晶体管

图3-51 以与图3-4相反的方式使用NMOS和PMOS晶体管

当用PMOS晶体管传输低电平时出现类似的情况，如图3-51b所示。这里假定初始栅极电平V_G和节点B的电平均为5 V。然后让V_G变到0 V，于是晶体管导通并驱动源节点B的电平转向0 V。当节点B的电平下降到V_T时，晶体管截止，因此稳态电平为V_T。

在3.1节中我们曾提到，NMOS晶体管的衬底端与地相连接，而PMOS晶体管的衬底与V_{DD}相连接。源极和衬底之间的电压V_{SB}称为衬底偏压（substrate bias voltage），在逻辑电路中一般值为0 V。但是在图3-51中，NMOS和PMOS晶体管的衬底偏压V_{SB}都等于V_{DD}（即$V_{SB} = V_{DD}$）。此衬底偏压将产生增大晶体管阈值电压V_T约1.5倍或更高倍数的效应[2, 1]。这个问题称为体效应（body effect）。

我们考虑图3-52所示的逻辑电路。该电路中的Gnd和V_{DD}的连接正好和前面讨论过的电路相反。当V_{x_1}和V_{x_2}都是高电平时，V_f被上拉到输出高电平，$V_{OH} = V_{DD} - 1.5\ V_T$。若$V_{DD} = 5$ V且$V_T = 1$ V，则$V_{OH} = 3.5$ V。无论V_{x_1}或V_{x_2}是低电平时，则V_f被下拉到输出低电平，$V_{OL} = 1.5V_T$，或者约1.5 V。正如图中的真值表所示，该电路表示的是一个与门。图3-52所示的电路与图3-15所示的一般与门相比较，具有的优点是需要的晶体管个数比较少。但该电路有一个缺点：因为V_{OH}和V_{OL}电平较差，所以噪声容限较差。



a) 一种与门电路

逻辑值		电平	逻辑值
x_1	x_2	V_f	f
0	0	1.5 V	0
0	1	1.5 V	0
1	0	1.5 V	0
1	1	3.5 V	1

b) 真值表以及电平

图3-52 一种差的CMOS与门的实现方案

图3-52所示电路的另一个重要的缺点是产生了静态功耗，而一般的CMOS与门是没有静态功耗的。假定该与门的输出连接到CMOS反相器的输入，当与门的输出V_f =3.5 V时，反相器中的NMOS晶体管导通，反相器的输出为低电平。但是此时反相器中PMOS晶体管并没有截止，因为栅极和源极之间的电压是−1.5 V，大于V_T。静态电流从V_{DD}流经反相器到达地。当与门产生低输出电平V_f =1.5 V时的情况与此相似，PMOS晶体管导通但NMOS晶体管并没有截止。所以图3-52所示的与门实际上是不能使用的。

3.8.8 逻辑门的扇入和扇出

逻辑门的扇入（fan-in）被定义为该逻辑门的输入个数。由于逻辑门构造的缘故，逻辑门输入端的个数超过一定限度是不切实际的，通常，逻辑门的扇入被限制在一个较小的数目内。举例说明，考虑如图3-53所示的NMOS与非门，该与非门有k个输入端。我们来看扇入k对该逻辑门传播延迟t_p产生的影响。假设k个NMOS晶体管具有同样的宽度W和长度L，因为晶体管是串联在一起的，所以可以把它们等价于一个长度为$k \times L$而宽度为W的晶体管。利用（3.4）式（该式既适用于CMOS也适用于NMOS晶体管）可以得到传播延迟为：

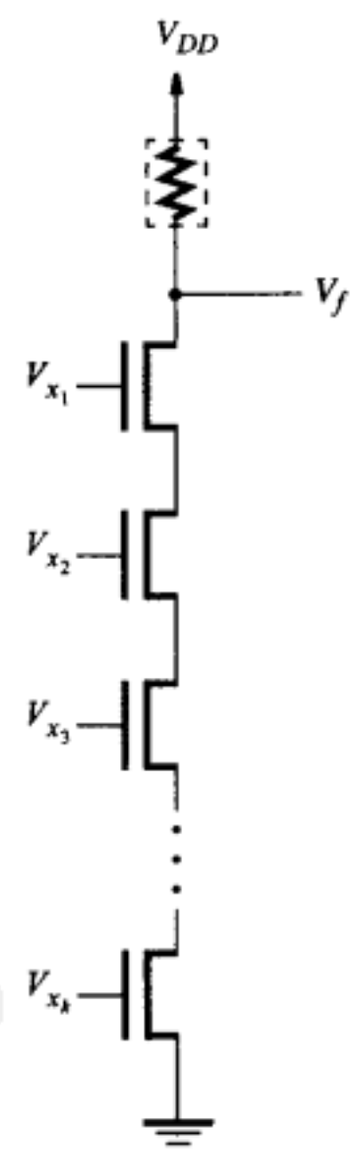


图3-53 高扇入NMOS 与非门

$$t_p \cong \frac{1.7C}{k_n' \frac{W}{L} V_{DD}} \times k$$

上式中C是门输出端的等价电容，包括由k个晶体管中的每个所带的寄生电容。可以通过增加每个NMOS晶体管的宽度W来改善性能，但是这种改变会增加电容C和增大芯片面积。此电路的另一个缺点是每一个NMOS晶体管都会影响V_{OL}的值，使它增大，从而降低了噪声容限。实际上，用这样的方式来建造与非门是不切实际的，只是在扇入很小的情况下，才会采用这种形式的与非门。

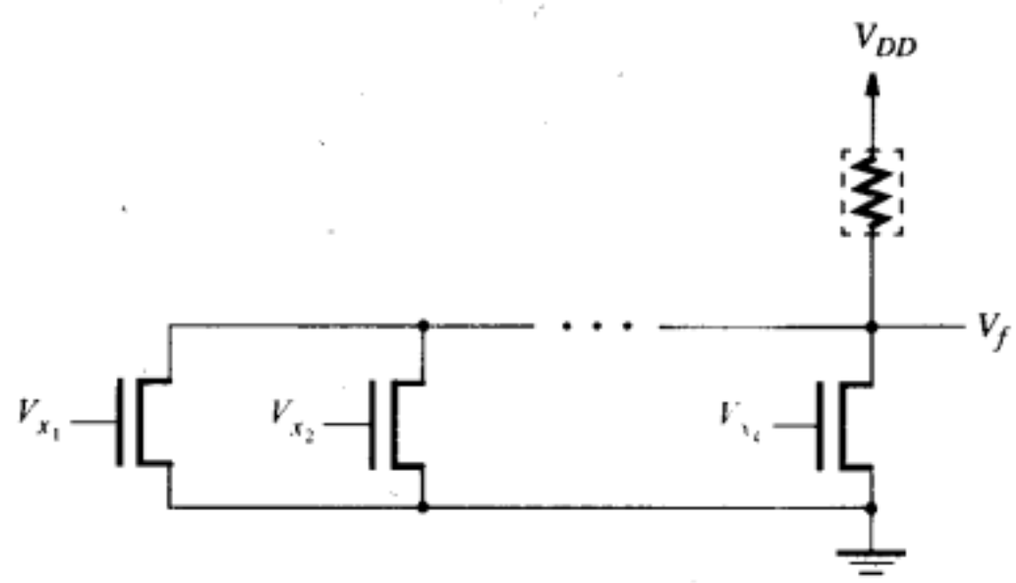


图3-54 高扇入NMOS或非门

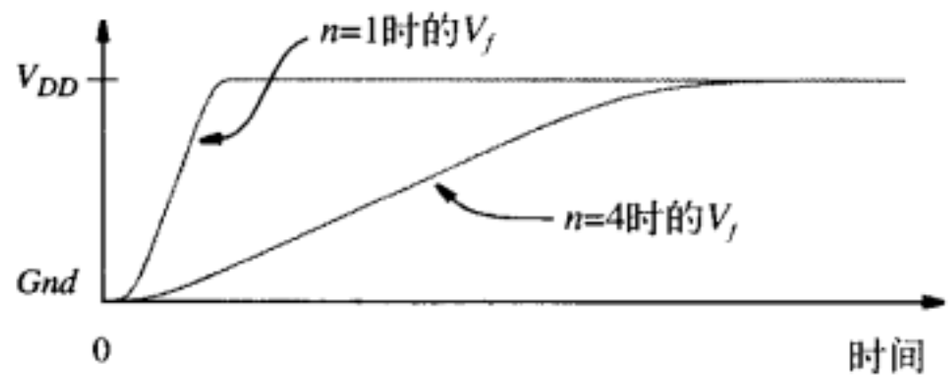
图3-54所示的是另一个扇入的例子，图中展示了一个NMOS的k输入或非门。在这个例子中，并联的k个NMOS晶体管可看做一个宽度为$k \times W$、长度为L的大晶体管。根据(3.4)式，传播延迟应该下降k倍。然而，晶体管的并联将增加门输出端的负载电容C，更重要的一点是：当V_f从高电平变化到低电平时，很少有机会出现所有晶体管一起导通的情况。因此采用NMOS工艺技术来建造高扇入或非门是很实际的。但是，需要提醒读者注意的是，因为NMOS电路中的上拉器件中流经的电流受到限制，所以电路由低到高的传播延迟要比由高到低的传播延迟缓慢（见例3.13和例3.14）。

高扇入的CMOS电路不是要求k个NMOS晶体管串联，就是要求k个PMOS晶体管串联，因此不切合实际。在CMOS电路中，构造高扇入门唯一可行的方法是用2个或多个低扇入门构造高扇入门。例如，如果要构造一个6输入的与门，其实现方法是把2个3输入与门连接到1个2输入与门上。还有比这个方法所用晶体管更少的方法，我们把这个问题留给读者作为练习（见习题3.4）。

扇出

图3-48展示了一个反相器驱动另一个反相器的时间延迟。在实际电路中，每个逻辑门电路都可能需要驱动很多个其他门。某个门驱动的其他门的个数称为扇出（fan-out）。图3-55a所示为扇出的举例。图中展示了反相器N_1驱动其他n个反相器的输入。每个被驱动的反相器都对节点f的电容有一份贡献，如图3-55b所示，用一个总负载电容C_n表示。简单地说，每一个反相器都贡献一个相等电容C，总负载电容$C_n = n \times C$。(3.4)式表明传播延迟直接和n成正比。

图3-55 扇出对传播延迟的影响



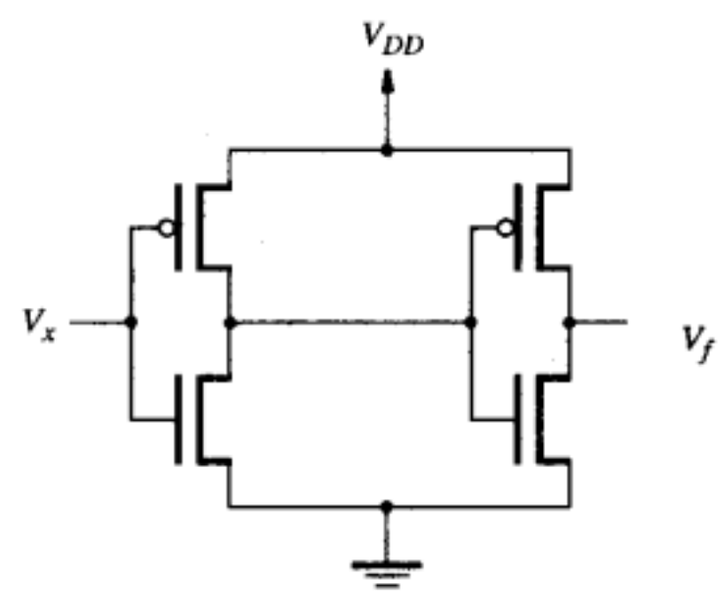
c) 不同n值的延迟时间

图3-55 （续）

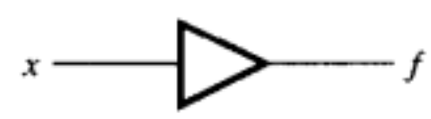
图3-55c用图说明了n如何影响传播延迟。假定在0时刻，信号x由逻辑值1变到0。一条曲线表示$n = 1$时V_f的波形，另一条曲线表示$n = 4$时V_f的波形。采用例3.7中的参数，当$n = 1$时$t_p = 0.1$ ns；则当$n = 4$时$t_p \approx 0.4$ ns。我们可以通过增加晶体管N_1的W/L的比来减小t_p的值。

缓冲器

在必须驱动很大电容性负载的逻辑门电路中，通常采用缓冲器来改善性能。缓冲器（buffer）是输入为x和输出为f，逻辑关系为$f = x$的逻辑门。最简单的缓冲器用2个反相器实现，见图3-56a所示。缓冲器可以建造成具有不同的驱动能力，能力的大小取决于晶体管的尺寸（见图3-49）。总的说来，因为缓冲器用于驱动相对于正常电容性负载而言，较大的电容性负载，所以缓冲器中的晶体管尺寸大于典型逻辑门中的晶体管尺寸。图3-56b所示为非反相的缓冲器的图形符号。



a) 缓冲器的实现



b) 图形符号

图3-56 非反相的缓冲器

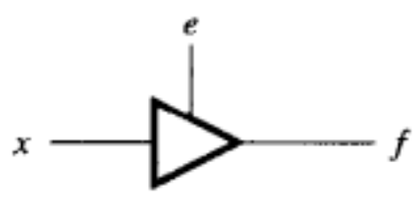
另一种类型的缓冲器是反相缓冲器。它产生的输出信号和反相器相同，即$f = \overline{x}$，但是它是用尺寸相对较大的晶体管构成的。反相缓冲器的图形符号和非门相同，反相缓冲器只是一个能驱动较大电容性负载的非门。如图3-55所示，当扇出n较大时，标记为N_1的非门应该使用反相缓冲器。

除了用来改善电路的速度性能之外，缓冲器还用来驱动需要大电流的外部设备。因为缓冲器由较大的晶体管构成，所以可通过较大的电流。缓冲器应用的最普遍的例子就是控制发光二极管（LED）。我们将在7.14.3节中介绍缓冲器在这方面的应用。

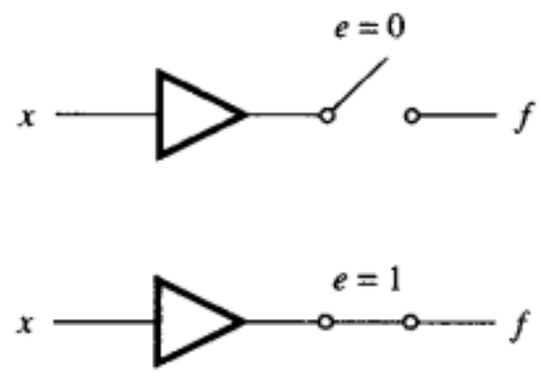
一般说来，扇出、电容性负载和电流都是数字电路设计者必须仔细考虑的重要问题。但在实际工作中，电路中是否需要使用缓冲器的决定是借助于CAD工具完成的。

三态缓冲器

在3.6.2节中，我们曾经提到在某些标准芯片及PLD中包含一种叫做三态缓冲器的元件。三态缓冲器有一个输入x和一个输出f，还有一个叫做使能（enable）的控制输入信号e。图3-57a所示的是三态缓冲器的图形符号。使能输入信号的作用是控制三态缓冲器是否产生输出信号，如图3-57b所示。当$e = 0$时，缓冲器和输出端f完全断开；当$e = 1$时，缓冲器把输入x的值驱动到f，使$f = x$。三态缓冲器的这种行为可用真值表的形式描述，见图3-57c，真值表中$e = 0$的两行的输出为逻辑值Z，Z称为高阻态。高阻态Z称为第三态，三态这个名称就来自于：对逻辑信号而言，有两种正常状态0和1，而Z表示第三种状态，该状态不产生输出信号。图3-57d所示的是三态缓冲器的一种可能的实现方案。



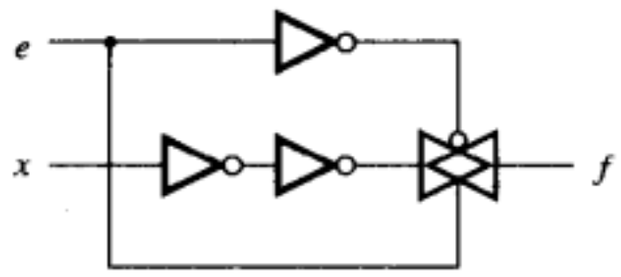
a) 一个三态缓冲器



b) 等价电路

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

c) 真值表



d) 实现

图3-57 三态缓冲器

图3-58所示的是几种不同类型的三态缓冲器。图3-58b所示的缓冲器和图3-58a所示缓冲器其行为基本相同，唯一不同之处是前者在$e=1$时，$f=\overline{x}$。图3-58c所示缓冲器和图3-58a所示缓冲器的不同之处是使能信号的作用恰好相反，即$e=0$时，$f=x$，而当$e=1$时$f=Z$。描述这种行为的技术术语是：该使能信号为低电平有效。图3-58d所示缓冲器也是低电平有效，但是当$e=0$时，该缓冲器的输出$f=\overline{x}$。

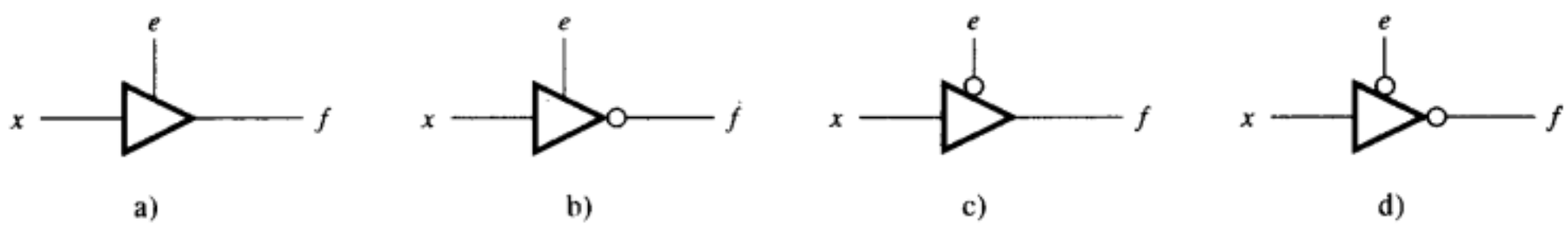


图3-58 四种类型的三态缓冲器

图3-59所示的电路是如何应用三态缓冲器的小例子。该电路的输出f究竟等于x_1还是等于x_2，取决于信号s的值。若$s=0$，则$f=x_1$；若$s=1$，则$f=x_2$。这种电路的特点是从输入信号中选择其中的一个作为输出，称为*多路选择器*（简称多路器）。用与门和或门可以实现的多路器，见图2-26所示。3.9.2节将给出另一种构建多路器的方法，在第6章中我们将详细地对这些方法进行讨论。

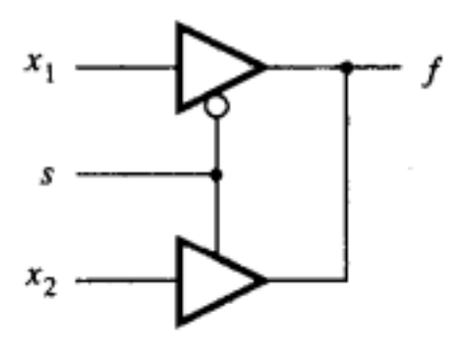


图3-59 三态缓冲门的应用

如图3-59所示，两个三态缓冲器的输出端连接在一起。这样做是允许的，因为控制输入信号s是这样连接的，可确保两个缓冲器中的一个处于高阻状态。只有在$s=0$时，输入为x_1的缓冲器才有效；只有在$s=1$时，输入为x_2的缓冲器才有效，这就保证了两个缓冲器不会同时有效。若两个缓冲器同时有效，则当两个缓冲器的输出值不同时，就会在电源和地之间形成短路电流。例如，假定$x_1=1$且$x_2=0$。缓冲器x_1的输出为V_{DD}，而缓冲器x_2的输出为Gnd，短路电流从V_{DD}经过三态缓冲器的晶体管到达Gnd。一般情况下，这么大的短路电流足以对电路造成破坏。

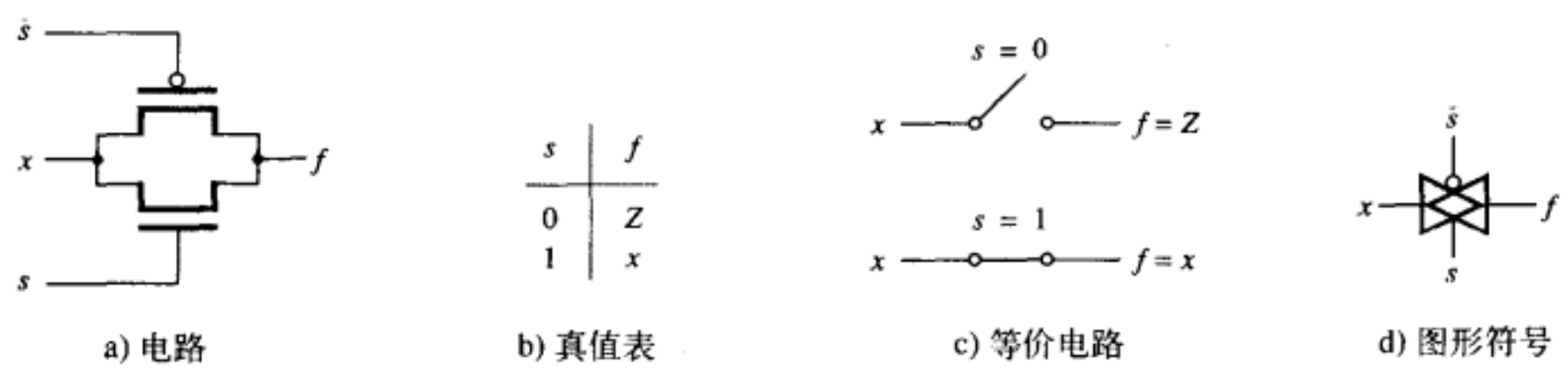
三态缓冲器的连线方式不适用于普通逻辑门，因为普通逻辑门的输出永远有效，从而导致短路电流的产生。正如我们早就知道的那样，若想使用普通逻辑门实现2选1的逻辑关系，

则可以通过积之和形式，用与门、非门以及或门实现从x_1或x_2中选一个输出的多路器。

3.9 传输门

在3.8.7节中已经介绍过，NMOS晶体管传输信号值0的能力较好，传输信号值1的能力较差。PMOS晶体管的情况恰好相反，传输信号值1的能力较好，传输信号值0的能力较差。因此我们可以把NMOS和PMOS晶体管组合起来形成一个开关，使得输出端的电平无论是高还是低都同样地好，图3-60a所示就是这样的一个传输门电路。如图3-60b和图3-60c所示，该传输门的作用如同一个开关，把x连接到 f。开关的控制由选择输入s和它的非$\bar{s}$提供。当V_s = 5 V且 $V_{\bar{s}}$ = 0 V时，开关接通。当V_x = 0 V时，因为$V_{GS} = V_s - V_x$ = 5 V，故NMOS晶体管导通，于是V_f是0 V；反之，当V_x = 5 V时，因为$V_{GS} = V_{\bar{s}} - V_x$ = −5 V，故PMOS晶体管导通，于是V_f是5 V。传输门的图形符号见图3-60d所示。

传输门可以有很多种应用，本节将介绍怎样用传输门有效地实现异或逻辑门（XOR）和多路器电路。



s	f
0	Z
1	x

a) 电路　b) 真值表　c) 等价电路　d) 图形符号

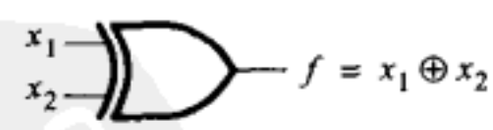
图3-60 传输门

3.9.1 异或门

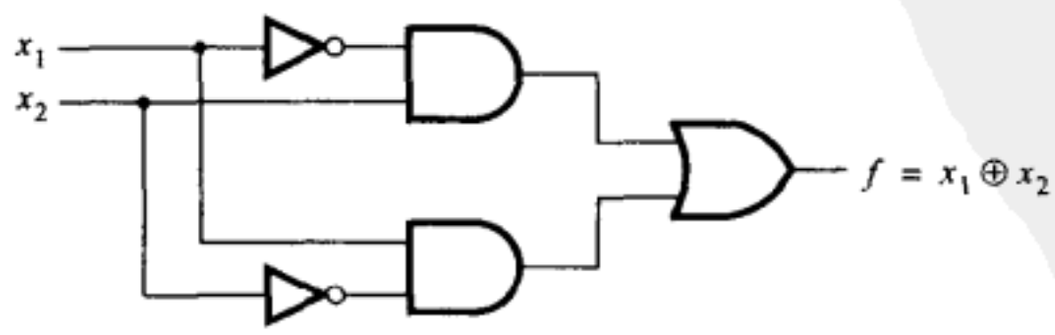
到目前为止，我们已经遇到过与门、或门、非门、与非门以及或非门，它们都是构成逻辑电路的基本元件。在实际工作中，还有一种基本元件非常有用，特别在构造算术运算电路时非常有用（见第5章）。该基本元件实现图3-61a定义的异或函数的功能。该函数的真值表类似于或函数的真值表，除了当两个输入都是1时，该函数输出 f = 0这一点外。因为这两个函数之间的相似性，所以该函数被称为异或函数，异或函数通常缩写为XOR。实现异或门的图形符号见图3-61b所示。

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

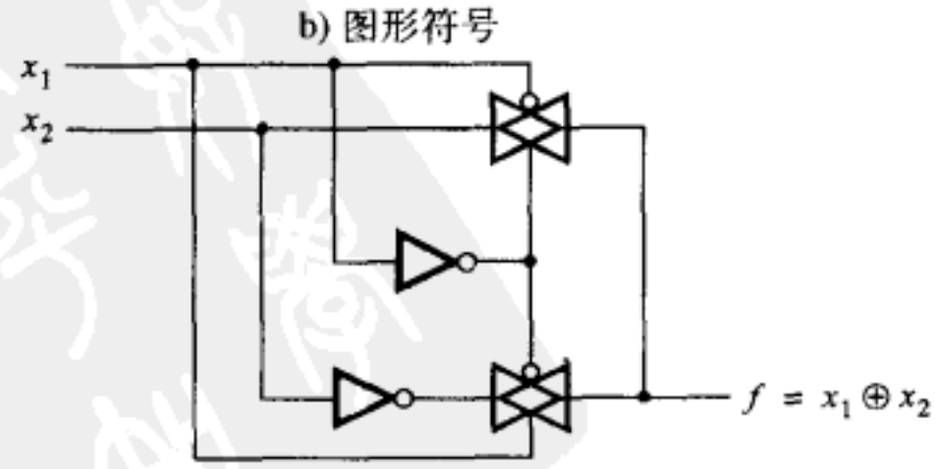
a) 真值表



b) 图形符号



c) 积之和实现



d) CMOS实现

图3-61 异或门

异或运算通常用符号⊕表示，它的积之和表示形式为：

$$x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$$

对应的电路如图3-61c所示。从3.3节中我们知道，采用CMOS技术的每个2输入与门（和或门）需要6个晶体管，每个非门需要2个晶体管，因此实现异或门总共需要22个晶体管。若采用传输门来实现异或门，则所需晶体管数会大大减少。图3-61d所示的电路为用2个传输门和2个反相器实现的异或门。当$x_1 = 0$时上面的传输门导通，使$V_f = x_2$；当$x_1 = 1$时下面的传输门导通，使 $V_f = \bar{x}_2$。读者可以自己验证该电路正确地实现了异或功能，我们将在第6章中介绍该电路是怎么推导出来的。

3.9.2 多路选择器电路

在图3-59中，我们展示了如何用三态缓冲器构造一个多路选择器。现在我们以类似的结构用传输门来实现多路选择器，见图3-62。选择输入信号s用于确定输出的值 f 应该是输入x_1还是x_2。若$s = 0$，则 $f = x_1$；若$s = 1$，则$f = x_2$。

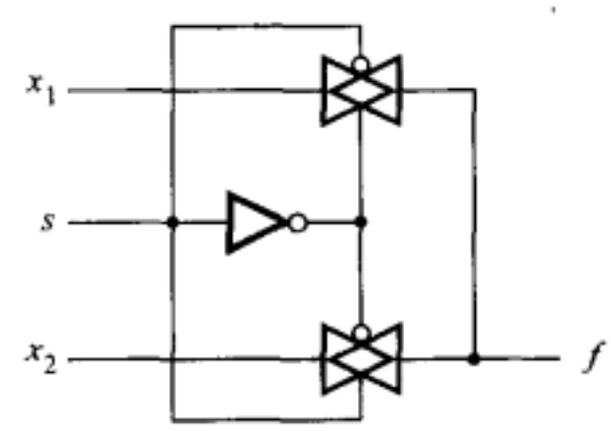


图3-62 用传输门构建的2选1多路选择器

3.10 SPLD、CPLD和 FPGA 的实现细节

我们在3.6节中曾介绍过PLD，在该节的芯片图中，用符号×表示可编程开关，现在我们讨论怎样用晶体管实现这些开关。

市售的SPLD主要用两种技术实现可编程开关。最早的技术是采用金属合金熔丝来做可编程连接。PLA和PAL就是采用这种技术制造的，每一个水平线和垂直线的交叉点通过一个很小的金属熔丝连接起来。在编程的时候，电路中所有不需要实现的点熔丝就被熔化掉。编程过程是不可逆转的，因为熔丝被熔化后就断开了。本节不再详细介绍这种技术，因为该技术已经被更新和更好的方法取代了。

当前生产的PLA和PAL，其可编程开关是用一种特殊的可编程晶体管实现的。因为CPLD是由许多类似PAL的基本模块组成的，所以简单可编程逻辑器件所使用的技术同样可以在复杂可编程逻辑器件（CPLD）中使用。我们首先介绍PLA的主要思想。PLA可以用来实现范围很宽的逻辑函数，既支持变量个数很少的函数又支持变量个数很多的函数。在3.8.8节中曾讨论过逻辑门的扇入问题，我们的结论是当扇入很大时，NMOS工艺的或非门是最适合的门类型，因此PLA通常基于这种类型的门。

考虑如图3-63所示的电路，这是说明PLA实现原理的小例子。标有S_1的水平线是NMOS工艺的或非门的输出，该门的输入分别是x_2和$\bar{x}_3$。因此 $S_1 = \overline{x_2 + \bar{x}_3}$ 。同样，S_2和S_3也是或非门的输出，表达式为 $S_2 = \overline{x_1 + x_3}$ 和 $S_3 = \overline{x_1 + \bar{x}_2 + x_3}$ 。产生信号S_1、S_2和S_3 的三个或非门，被安排在用集成电路能高效实现的规则结构中，该结构称为*或非平面*（NOR plane，也称或非阵列）。只需要在列的方向上增加输入、在行的方向上增加或非门的个数，则或非（NOR）平面的规模便可以扩展得更大。

信号S_1、S_2和S_3，对第二个或非平面而言，所起的作用是输入。第二个或非平面相对于第一个或非平面顺时针方向旋转了90°，这样做是为了容易画图。产生输出信号 f_1的或非门其输入是S_1和S_2。于是：

$$f_1 = \overline{S_1 + S_2} = \overline{\overline{(x_2 + \bar{x}_3)} + \overline{(x_1 + x_3)}}$$

根据德摩根定律，与上述表达式等价的和之积形式为：

$$f_1 = \overline{S}_1\overline{S}_2 = (x_2 + \overline{x}_3)(x_1 + x_3)$$

同样，输出信号为f_2的或非门的输入是S_1和S_3，因此

$$f_2 = \overline{S_1 + S_3} = \overline{\overline{(x_2 + \overline{x}_3)} + \overline{(x_1 + \overline{x}_2 + x_3)}}$$

其等价的和之积形式为：

$$f_2 = \overline{S}_1\overline{S}_3 = (x_2 + \overline{x}_3)(x_1 + \overline{x}_2 + x_3)$$

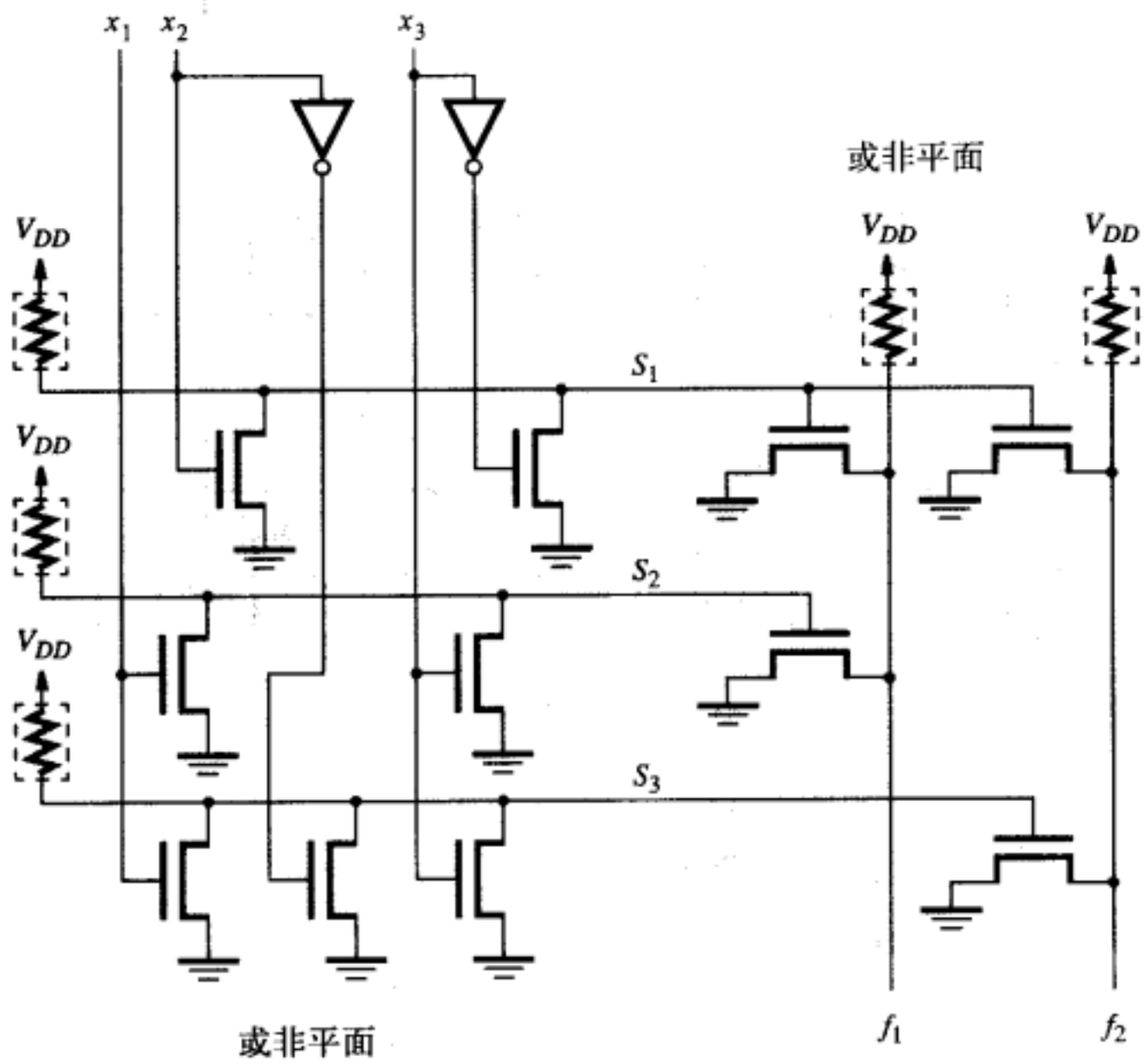


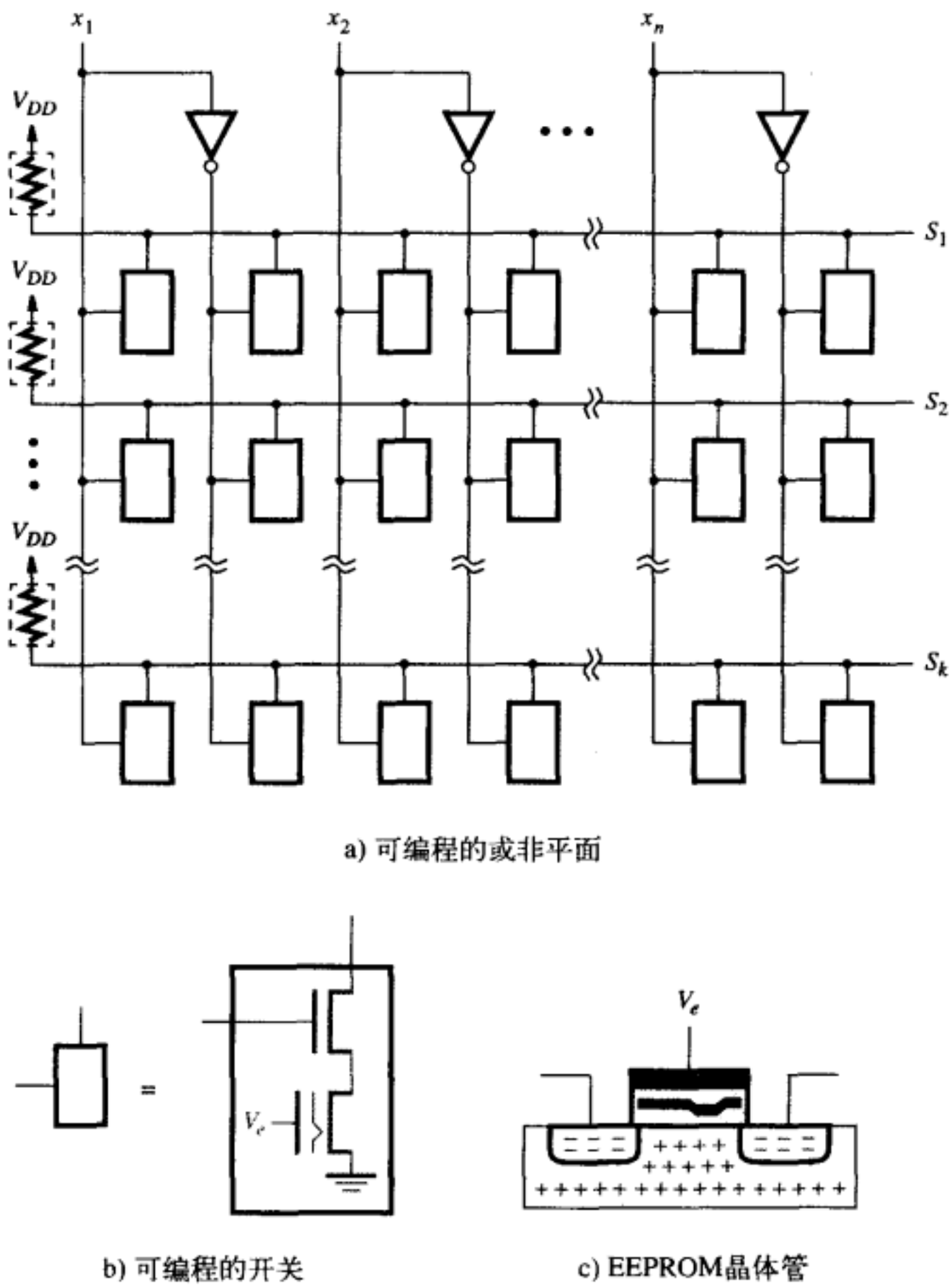
图3-63 或非—或非可编程逻辑阵列（PLA）举例

如图3-63所示的那种PLA叫做NOR-NOR PLA。还有其他的实现方法，但是由于这种形式的简单性，NOR-NOR形式是最流行的。读者应注意，图3-63所示PLA的晶体管连接关系是不可编程的，它只能实现两种特定的逻辑函数 f_1 和 f_2。但是，NOR-NOR结构也可以应用于可编程PLA中，下面我们将详细讨论。

严格地讲，PLA这个术语应该用来描述如图3-63所示的固定类型的PLA。描述可编程类型PLA的技术术语应该是：*现场可编程逻辑阵列*（FPLA）。然而，人们通常省略F。图3-64a展示了一种可编程的或非平面。该或非平面具有n个输入信号：$x_1,\cdots,x_n$；k个输出信号：$S_1,\cdots,S_k$。在每个水平和垂直交叉点上都存在一个可编程开关。该开关由两个串连的晶体管组成：一个NMOS晶体管和一个*电可擦除可编程只读存储器*（EEPROM）晶体管。

可编程开关的基础是EEPROM晶体管的行为。电子学教科书详细地解释了EEPROM晶体管的工作原理，请参阅文献[1，2]。 本节只做简单的介绍。图3-64b绘制了一个可编程开关，图3-64c画出了EEPROM晶体管的内部结构。EEPROM晶体管的外观与一般NMOS晶体管基本相同（见图3-43），只有一个主要的不同点。EEPROM晶体管具有两个门：其中一个为普通的NMOS晶体管门，另外一个为*浮动门*。之所以将第二个门称为浮动门是因为该门由玻璃绝缘层包围，与晶体管的其他部分没有连接。当该晶体管处于未编程的原始状态时，则浮动门对晶体管的运行没有任何影响，因此该晶体管就与一般的NMOS晶体管一样运行。在PLA的正

常使用期间，施加于浮动门上的电压V_e由电路（图上没有标出）设置成V_{DD}，则EEPROM晶体管随之导通。



a) 可编程的或非平面

b) 可编程的开关

c) EEPROM晶体管

图3-64 用EEPROM晶体管来创建可编程的或非平面

对EEPROM晶体管的编程需要用比正常电压高的电平（典型情况下，V_e = 12 V）将晶体管导通，使大量的电流流过晶体管的沟道。浮动门的一部分向下伸展以使其非常接近沟道的顶层，如图3-64c所示。若有大电流流过沟道，就会产生Fowler -Nordheim隧道效应，由于该隧道效应的作用，沟道中有些电子会穿过玻璃绝缘层的最薄处，聚集到浮动门下。编程过程结束后，聚集的电子将会排斥其他电子进入沟道。若施加V_e = 5 V到EEPROM晶体管，一般情况下，应该使其导通，然而聚集的电子使该晶体管保持断开的状态。因此在图3-64a所示的或非平面中，编程是将输入信号与或非平面断开。对于那些应该连接到每个或非门的输入，则相应的EEPROM晶体管应该保留在未编程状态。

一旦EEPROM被编程，它将永远保留其被编程后的状态。但是编程过程也可以反向进行，这个过程叫做擦除，需要使用跟编程极性相反的电压才能完成。在擦除场合，所施加的电压将使聚集在浮动门下的电子再次通过隧道效应返回沟道。于是EEPROM晶体管恢复到原始状态，再次像正常的NMOS晶体管那样动作。

为了完整起见，我们也应该提及另外一种类似EEPROM的技术，这种技术叫做可擦除

PROM（简称EPROM）。这种类型的晶体管实际上是EEPROM的上一代产品，用类似EEPROM的方式编程，但擦除的方法不同。EPROM晶体管擦除时必须将其暴露在指定波长的光能下。为了做到这一点，基于EPROM技术的芯片必须安装在有透明玻璃窗口的封装内。为了擦除芯片中的内容，必须把该芯片暴露在紫外线光源下达几分钟。因为EEPROM晶体管所用的电擦除方法比EPROM晶体管所用的光擦除方法更加方便，所以目前在实际工作中EPROM技术已经被 EEPROM 技术所替代。

图3-65绘制了一个完整的使用EEPROM技术的或非-或非可编程逻辑阵列（PLA），该芯片在第一个或非平面中有四个输入信号、六个或（逻辑和）项，并有两个输出。每个被编程为处于断开状态的可编程开关用黑色的×符号表示，而每个未曾编程的开关用灰色的×符号表示。用图3-65所示的编程状态，该PLA所实现的逻辑函数 f_1和 f_2分别为：$f_1 = (x_1 + x_3)(x_1 + \overline{x}_2)(\overline{x}_1 + x_2 + \overline{x}_3)$和 $f_2 = (x_1 + \overline{x}_3)(\overline{x}_1 + x_2)(x_1 + \overline{x}_2)$。

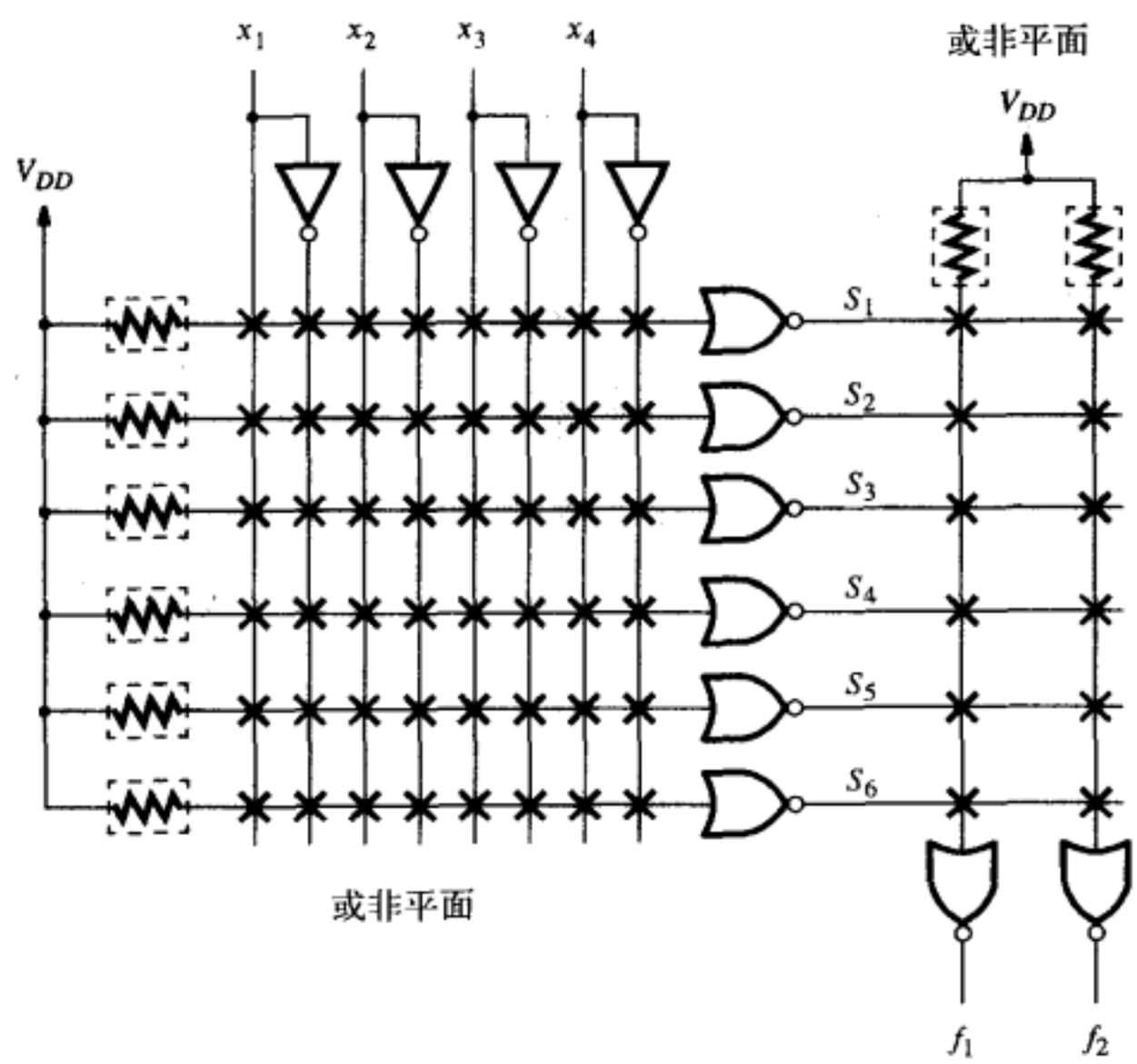


图3-65 或非-或非可编程逻辑阵列的可编程版

在PLA中也可以不用和之积的形式而用积之和形式来实现某逻辑函数。对积之和的形式，在PLA的第一个或非平面中必须要有与门。若我们把输入到或非平面的输入信号先求反，则根据德摩根定律，此举等价于创建一个与平面。在PLA中，求反不需要成本，因为PLA已为每个输入提供了正、反两种形式的输入。图3-66所示的例子说明了积之和形式的实现。从第一个或非平面的输出用P_1，…，P_6标记，反映了我们将其解释为乘积项。信号P_1被编程为实现 $\overline{\overline{x}_1 + \overline{x}_2} = x_1x_2$ 。同样，$P_2 = x_1\overline{x}_3$，$P_3 = \overline{x}_1\overline{x}_2x_3$，$P_4 = \overline{x}_1\overline{x}_2\overline{x}_3$。已经产生了想要的乘积项后，我们现在需要将它们或起来。这一操作可以用第二个或非平面的输出求反来实现。为此目的，图3-66中添加了非门。图中在该或平面上的可编程开关表示的状态产生下列输出：$f_1 = P_1 + P_2 + P_3 = x_1x_2 + x_1\overline{x}_3 + \overline{x}_1\overline{x}_2x_3$和 $f_2 = P_1 + P_4 = x_1x_2 + \overline{x}_1\overline{x}_2\overline{x}_3$。

上面所描述的关于PLA的概念也可用于PAL。图3-67展示了具有四个输入两个输出的PAL。让我们假设第一个或非平面被编程为实现上述乘积项。请注意图中乘积项被用硬线连接成三个一组，再连接到或门，从而产生PAL的输出。正如我们曾在图3-29中说明的那样，该PAL也

许在或门和输出引脚之间还有一些其他的电路，而在图3-67中并没有画出这个电路。该PAL被编程为实现与图3-66所示的PLA生成的f_1和f_2相同的逻辑函数。观察到乘积项x_1x_2在PAL中实现了两次，一次在P_1另外一次在P_4上。重复是必须的，因为PAL的乘积项不能像在PLA中那样被多个输出分享。从图3-67中可观察到的另外一些细节是虽然函数 f_2 只需要两个乘积项，但每个或门还是用硬线连接到三个乘积项。多余的乘积项 P_6 必须设置成逻辑值0，这样就不会产生任何影响。这是通过把 P_6 编程为某个输入和它的非的乘积项实现的，因此 P_6 的结果总是0。在图3-67中 $P_6 = x_1\overline{x}_1 = 0$，但是也可以用任何其他输入来达到此目的。

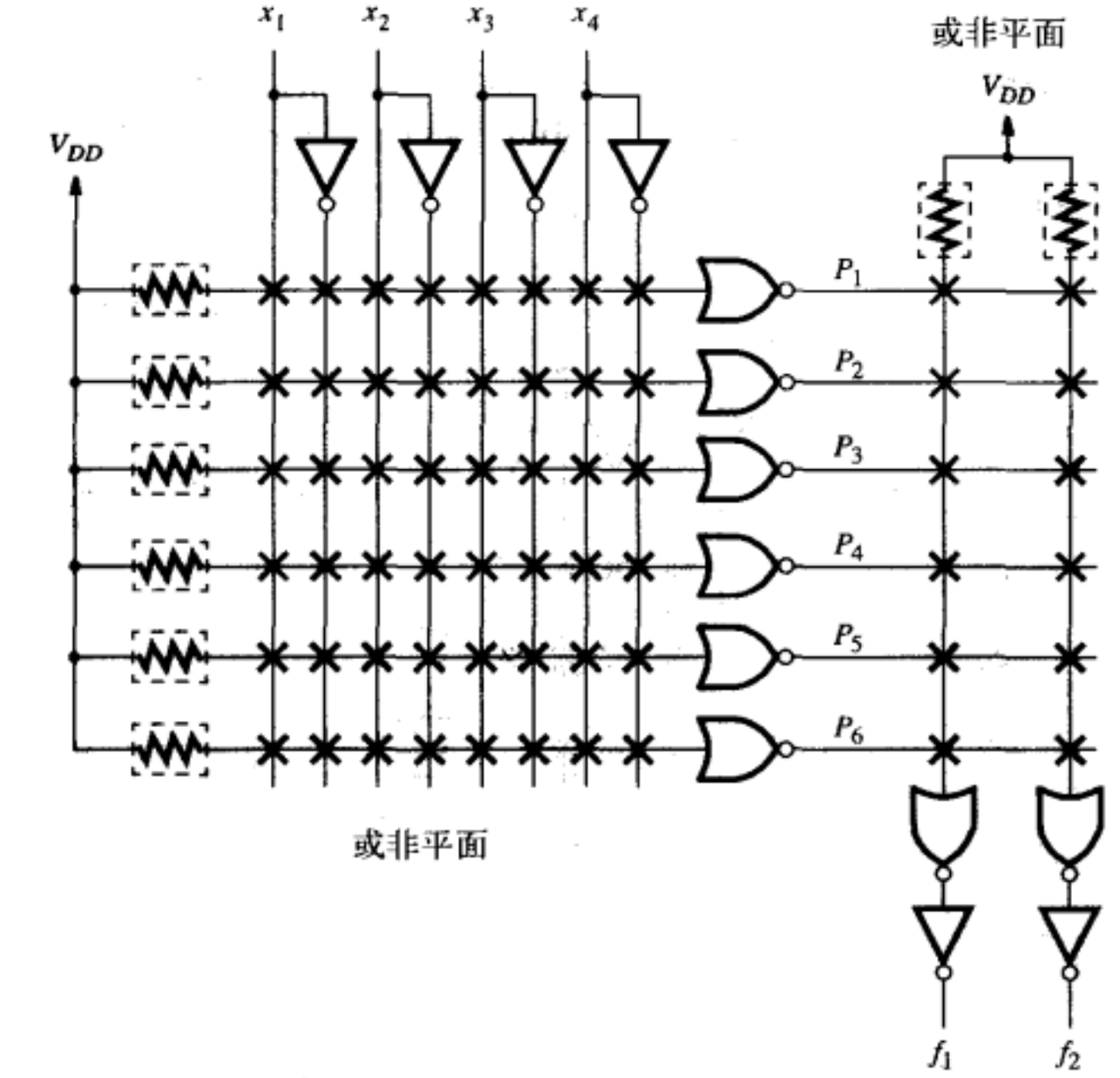


图3-66 用于积之和形式的或非-或非可编程逻辑阵列

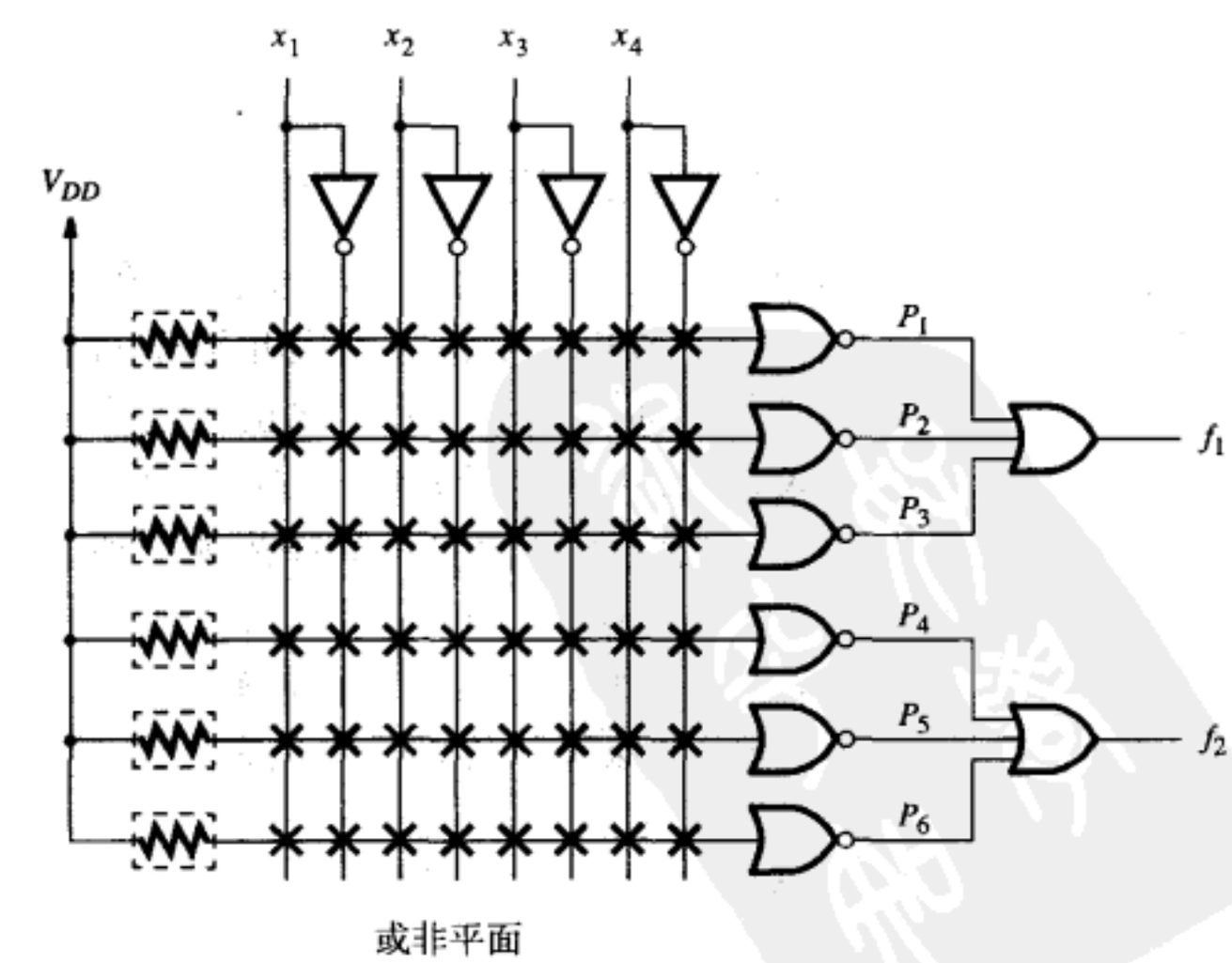


图3-67 实现图3-66所示函数的PAL编程示意

包含在CPLD中的类似于PAL的块通常是用本节介绍的技术实现的。在典型的CPLD中，与（AND）平面是用既有输入信号又有相应求反信号的NMOS或非门所构建的。或（OR）平面与PAL一样是用硬线连接的，而并非像PLA中那样是完全可编程的。然而，连接到或门的许多乘积项之中还存在着某些灵活性。这种灵活性是通过可编程电路实现的，通过编程可以将某乘积项连接到用户想要的或门。附录E中举了一个例子，介绍了市售CPLD的这种灵活性。

FPGA中的实现

FPGA不是使用EEPROM技术来实现可编程开关的。FPGA是将编程信息存储在叫做*静态随机存取存储器*（SRAM）的存储单元之中。关于该类型存储单元的操作将在10.1.3节详细介绍。现在我们只要知道每个存储单元可以存储逻辑0或者1，并可以将存储的值输出，这就已经足够了。一个SRAM单元可以存储查找表（LUT）所表示的真值表中的一个值。SRAM单元可以被用来配置FPGA中的线路连接。

图3-68所示的是从图3-39所示的FPGA中截取的一小部分。所示的逻辑块产生的输出 f_1 驱动灰色的水平线。这条线可以通过可编程开关连接到与之相交叉的某些垂直线。每个开关用一个栅极由SRAM单元控制的NMOS晶体管实现。这样的开关称为*导通晶体管开关*（pass-transistor switch）。若SRAM单元内存储的为0，则相应的NMOS晶体管便断开。若SRAM单元内存储的为1，则相应的NMOS晶体管便接通。该开关形成了两条线路之间的连接，把信号传送到目的地。FPGA所能提供的开关的数目取决于特定芯片的结构。在某些FPGA中，有些开关是用三态缓冲器实现的，而不是用传输晶体管实现的。附录E中列举了一些市售的FPGA芯片。

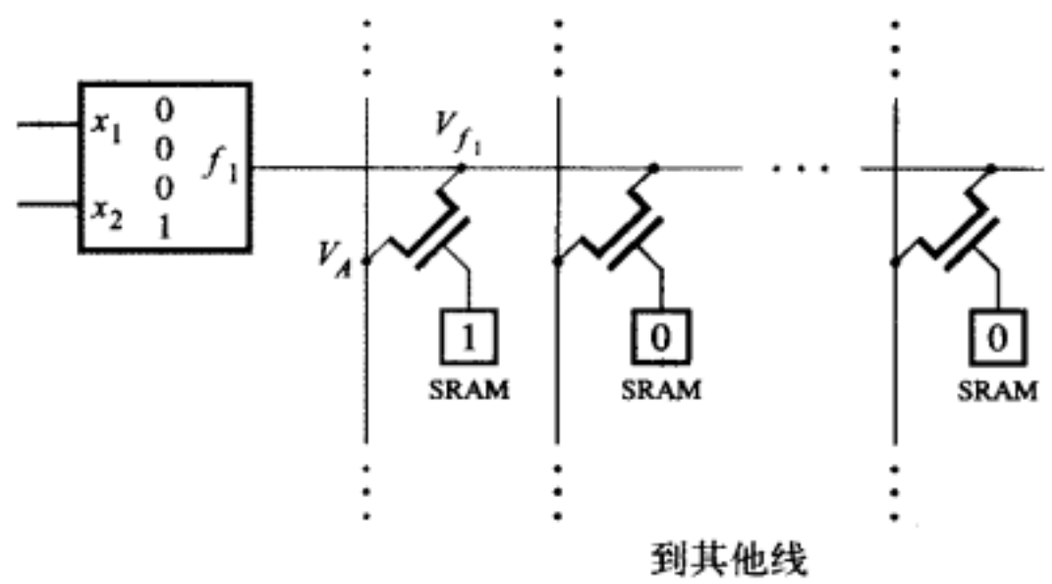


图3-68 FPGA中的导通晶体管开关

在3.8.7节我们曾提到，NMOS晶体管只能传递部分高逻辑值。因此，在图3-68中，若V_{f_1}= 5 V，则V_A = 3.5 V。我们曾在3.8.7节中解释过，电平的降低将造成静态功率的消耗（见练习题3.35）。图3-69解释了解决这个问题的一种方法[1]。我们假设信号V_A必须流过另外一个导通晶体管开关才能到达另外一个逻辑块。信号V_B的值和信号V_A的值是相等的，因为阈值电平的下降只发生在经过第一个导通晶体管时。为了恢复V_B的电平，在电路中添加了一个反相器做缓冲。在反相器的输入和V_{DD}之间连接了一个PMOS晶体管，该晶体管由反相器的输出控制。V_B = 0 V时，PMOS晶体管对反相器的输出电平没有任何影响。而当V_B = 3.5 V时，反相器的输出为低电平，则PMOS晶体管导通。该晶体管很快就把V_B恢复到V_{DD}的固有电平，则阻止了在稳定态下的电流流动。另外一种解决方案是不用上拉晶体管，而是在集成电路的制造过程中改变图3-69中反相器的PMOS晶体管的阈值电平，使得当V_B = 3.5 V时，阈值电平的幅度就足以令晶体管断开。在市售FPGA中，这两种技术分别用在不同的芯片之中。

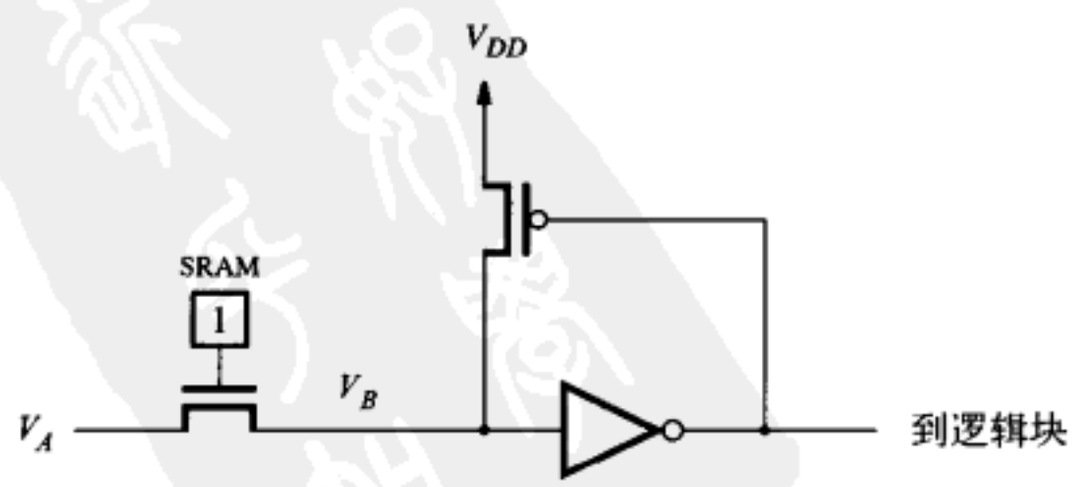


图3-69 恢复高电平

除了使用单个NMOS晶体管作为开关外，还可以使用传输门作为开关，见3.9节的介绍。

虽然这个办法解决了电平问题，但存在着两个缺点。第一点，在开关中既有NMOS又有PMOS晶体管，增加了互连线路的电容性负载，这将增加传播延迟。第二点，传输门比单个晶体管占用更大的芯片面积。因为这些原因，目前市售的FPGA芯片不使用传输门开关。

3.11 小结

我们已经描述了如何用晶体管构建逻辑门的最重要的概念。讨论了晶体管的制造、电平、传播延迟、功耗和其他一些问题，其宗旨是帮助读者理解在设计和使用逻辑电路时必须考虑的实际问题。

我们已经介绍了几种类型的集成电路芯片。每一种类型都有适合自己的特殊应用：标准芯片，诸如7400系列，只包含少量的简单门，现在已经很少使用，只有缓冲器是例外。当数字电路必须以很高的速度驱动大的电容性负载时，还需要用到缓冲器。不同类型的可编程逻辑器件都有着广泛应用。简单的可编程逻辑器件，诸如PLA和PAL，适合实现小规模的逻辑电路。SPLD成本较低而且速度很高。CPLD的应用范围与SPLD相同，但CPLD也非常适合实现1万～2万门以上的大规模电路。很多可以用CPLD实现的电路都可以用FPGA来实现，具体设计中究竟选择这两种类型电路的哪一种取决于许多因素。随着把尽可能多的电路装入单个芯片之中的发展趋势，目前CPLD和FPGA的应用范围远超过SPLD。当今工业界所创建的绝大部分数字设计中都包含某些类型的PLD。

在不适于应用PLD的场合，可以使用门阵列、标准单元以及全定制芯片技术。若需要设计电路的规模非常巨大，要求速度非常高、功耗非常低，预期设计产品的销售量也非常巨大，这样的情况最适合应用全定制技术。

下一章将讨论逻辑函数的优化问题。其中有一部分只与逻辑电路的综合有关，而不考虑逻辑电路是用哪一种技术实现的。还有一部分关于综合技术的讨论需要配合电路考虑，以便能用芯片中特定类型的资源来实现逻辑。我们将展示，当把逻辑函数综合为电路时，所使用的优化方法至少部分地取决于所使用芯片的类型。

3.12 问题求解举例

本节介绍读者可能遇到的一些问题，并展示如何求解这一类典型问题。

例3.9 在3.7节中，我们曾介绍过标准单元技术。在这种技术中，电路是由一些实现简单逻辑功能的构件块（如基本的逻辑门）互相连接组成的。有一种常用的标准单元叫做*与或反相器*（and-or-invert，AOI），可用CMOS技术制造这样复杂的门，其效率是较高的。考虑图3-70所示的AOI单元逻辑图。该单元实现的逻辑函数为 $f=\overline{x_1x_2+x_3x_4+x_5}$。请推导出实现该逻辑函数的复杂门构造。

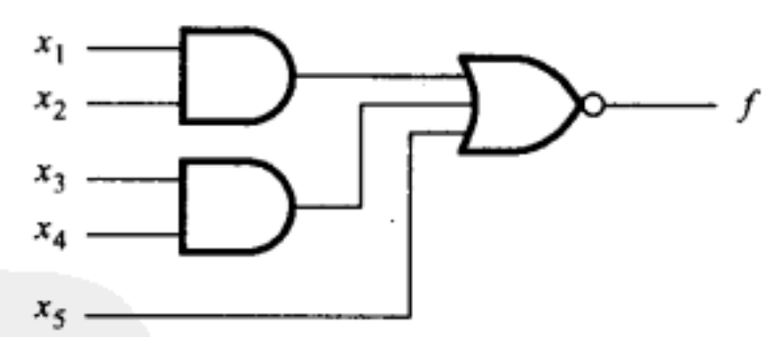


图3-70 例3.9中的AOI（与或反相器）单元

解：分两步应用德摩根定理得到如下表达式：

$$
\begin{aligned}
f &= \overline{x_1x_2}\cdot\overline{x_3x_4}\cdot\overline{x_5} \\
&= (\overline{x}_1+\overline{x}_2)\cdot(\overline{x}_3+\overline{x}_4)\cdot\overline{x}_5
\end{aligned}
$$

因为上述表达式中所有的输入变量都是求反的，所以我们可以直接得到由x_1和x_2控制的并行连接的PMOS三极管组成的上拉网络，此上拉网络与由x_3和x_4控制的并行连接的三极管串联，再与由x_5控制的三极管串联。该电路以及对应的下拉网络如图3-71所示。

例3.10 对图3-71中的复杂CMOS门而言，确定所用三极管的大小以保证该门的速度性能接近反相器的速度性能。

解：回忆一下3.8.5节的内容：长度为L、宽度为W的三极管，其驱动强度与W/L的比值成正比。再回忆一下：当三极管并联时，其宽度相加，因此其驱动强度得到有效的增强。当三极管串联时，其长度相加，因此其驱动强度下降。假设所有的NMOS和PMOS三极管的长度全都相等，即$L_n = L_p = L$。在图3-71中，假设连接到输入V_{x_5}的NMOS三极管与反相器W_n的宽度是相同的。但是该电路在最坏情况下通过下拉网络的路径涉及两个串联的NMOS三极管。对这些与V_{x_1}，…，V_{x_4}连接的NMOS三极管，我们应该考虑其宽度为$2 \times W_n$。对这个上拉网络而言，最坏情况下的路径涉及三个串联的三极管。正如我们在3.8.1节中提到过的那样，由于PMOS三极管的驱动强度大约是NMOS三极管的一半，因此我们应该将PMOS三极管的有效宽度制成：

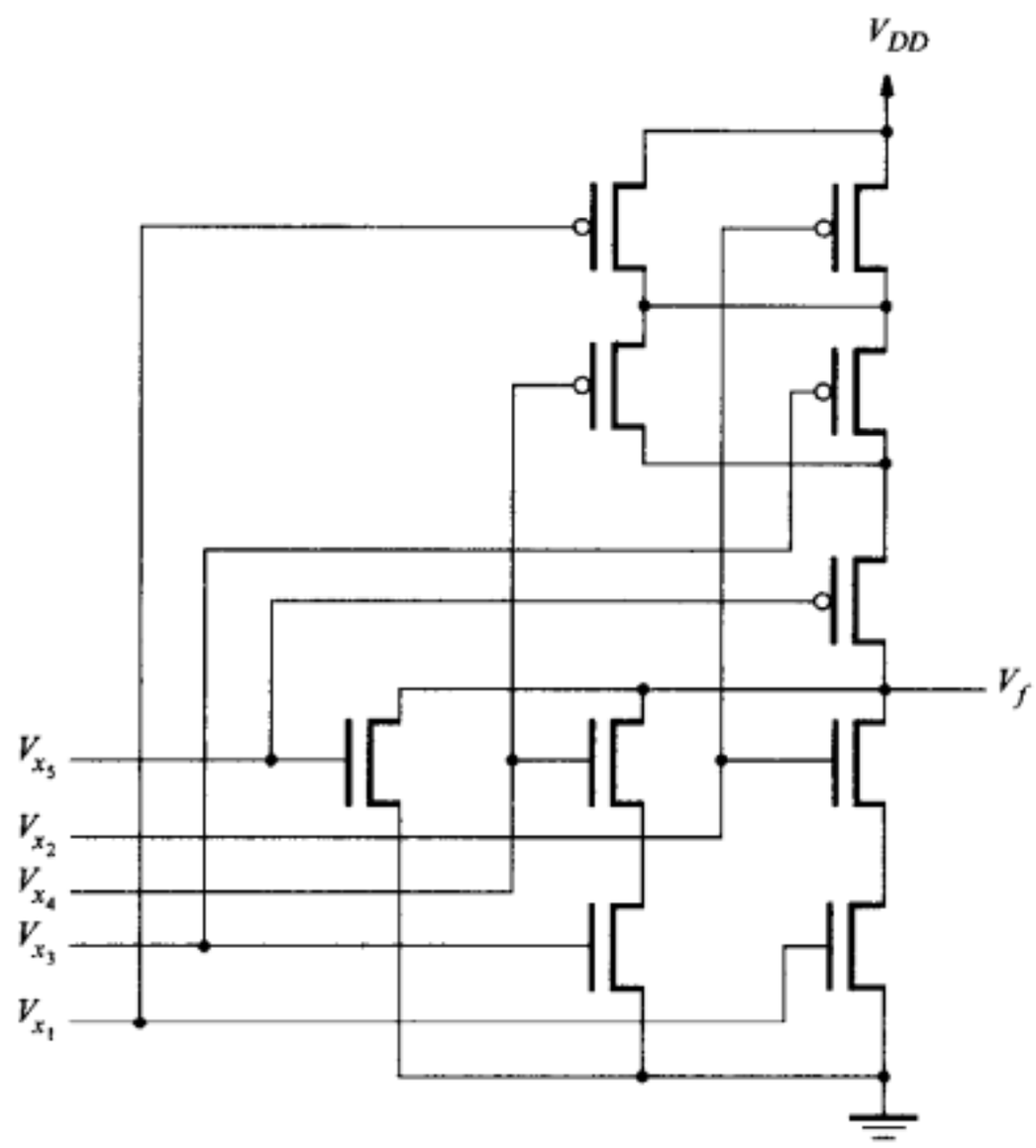


图3-71 例3.9和例3.10的电路图

$$W_p = 3 \times W_n \times 2 = 6W_n$$

例3.11 在3.8.5节中，我们曾讲到对电容充电所需要的时间由下式给定：

$$t_p = \frac{C\Delta V}{I}$$

请推导出上述表达式。

解：正如我们在3.8.5节中讲过的那样，电容器两端的电压不能发生瞬间的跳变。在图3-50a中，V_f是从0伏特变化到V_{DD}的，该电压的变化将按照如下公式进行：

$$V_f = \frac{1}{C}\int_0^{\infty} i(t)\mathrm{d}t$$

在上面的表达式中，变量t是独立时间变量，且$i(t)$表示了在t时刻流过电容的瞬时电流。将上述表达式的两边对时间求微分，并重新组织一下，便可以得到如下表达式：

$$i(t) = C\frac{\mathrm{d}V_f}{\mathrm{d}t}$$

在电流I为常量的情况下，可以得到如下表达式：

$$\frac{I}{C} = \frac{\Delta V}{\Delta t}$$

因此

$$\Delta t = t_p = \frac{C\Delta V}{I}$$

例3.12 在3.8.6节中，我们曾对图3-50a进行过讨论：当电容两端的电压V_f被充电到V_{DD}时，电容C储存的能量等于$CV_{DD}^2/2$。请推导出该表达式。

解：正如例3.11所示，流经正在充电的电容C的电流与该电容两端电压的变化率有关，其关系为：

$$i(t) = C\frac{\mathrm{d}V_f}{\mathrm{d}t}$$

在该电容上所消耗的瞬时功率可以由以下表达式表示：

$$P = i(t) \times V_f$$

由于能量被定义为在一定的时间周期内用掉的功率，所以我们可以通过在时间周期内对瞬间功率进行积分，计算出当V_f从0变化到V_{DD}时，储存在电容中的能量E_C。其计算表达式如下：

$$E_C = \int_0^\infty i(t)V_f \mathrm{d}t$$

在上面的表达式中，将 $i(t)$ 替换成 $C\frac{\mathrm{d}V_f}{\mathrm{d}t}$ 可以得到如下表达式：

$$\begin{aligned} E_C &= \int_0^\infty C\frac{\mathrm{d}V_f}{\mathrm{d}t}V_f \mathrm{d}t \\ &= C\int_0^{V_{DD}} V_f \mathrm{d}V_f \\ &= \frac{1}{2}CV_{DD}^2 \end{aligned}$$

例3.13 在原来的NMOS技术中，上拉器件是n沟道MOSFET三极管。但是当前生产的集成电路都采用CMOS技术。因此用PMOS三极管做上拉电阻是很方便的，如图3-72所示。这样的电路称为伪NMOS电路。其上拉器件称为“弱”PMOS三极管，因为该三极管的W/L比很小。

当$V_x = V_{DD}$时，V_f的值很小。NMOS三极管工作在三极管工作区，而PMOS三极管限制了电流的流动，因为此时PMOS三极管工作在饱和区。流经NMOS和PMOS三极管的电流必须相等，并且由式（3.1）和式（3.2）给定。该低电平的输出电压V_f由下面的表达式给定：

$$V_f = (V_{DD} - V_T)\left(1 - \sqrt{1 - \frac{k_p}{k_n}}\right)$$

其中k_p和k_n叫做增益因素，分别取决于PMOS和NMOS三极管的大小，由$k_p = k_p' W_p/L_p$和$k_n = k_n' W_n/L_n$来定义。

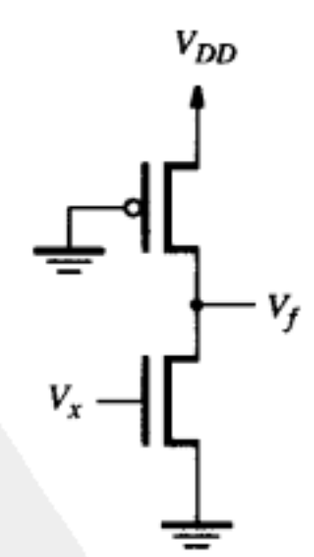


图3-72 伪NMOS反相器

解：为了简单起见，假设NMOS和PMOS三极管的阈值电压的幅度是相等的，因此

$$V_T = V_{TN} = -V_{TP}$$

因为PMOS三极管工作在饱和区，所以流经它的电流由下面的表达式给定：

$$\begin{aligned} I_D &= \frac{1}{2}k_p' \frac{W_p}{L_p}(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(V_{DD} - V_T)^2 \end{aligned}$$

同样，NMOS 三极管工作在三极管工作区，其电流由以下表达式定义：

$$\begin{aligned} I_D &= k_n' \frac{W_n}{L_n}\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right] \end{aligned}$$

因为只有一条电流通路，所以可以将流经NMOS和PMOS三极管的电流列成方程，并求解电压V_f。

$$k_p(V_{DD} - V_T)^2 = 2k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right]$$

$$k_p(V_{DD} - V_T)^2 - 2k_n(V_{DD} - V_T)V_f + k_nV_f^2 = 0$$

用以下参数：

$$a = k_n,\ b = -2\,k_n\,(V_{DD} - V_T),\ c = k_p(V_{DD} - V_T)^2$$

该二次方程式可以用标准公式求解，得到

$$\begin{aligned} V_f &= \frac{-b}{2a} \pm \sqrt{\frac{b^2}{4a^2} - \frac{c}{a}} \\ &= (V_{DD} - V_T) \pm \sqrt{(V_{DD} - V_T)^2 - \frac{k_p}{k_n}(V_{DD} - V_T)^2} \\ &= (V_{DD} - V_T)\left(1 \pm \sqrt{1 - \frac{k_p}{k_n}}\right) \end{aligned}$$

这两个解中只有一个是合理的，因为我们开始就假设NMOS三极管工作在三极管工作区，而PMOS三极管工作在饱和区，因此

$$V_f = (V_{DD} - V_T)\left(1 - \sqrt{1 - \frac{k_p}{k_n}}\right)$$

例3.14 对图3-72所示的电路，假设值$k_n' = 60\mu A/V^2$，$k_p' = 0.4k_n'$，$W_n/L_n = 2.0\ \mu m/0.5\mu m$，$W_p/L_p = 0.5\mu m/0.5\mu m$，$V_{DD} = 5V$，且$V_T = 1V$。当$V_x = V_{DD}$时，请计算下列值：

（a）静态电流I_{stat}。

(b) 该NMOS三极管的导通电阻。

(c) V_{OL}。

(d) 该反相器的静态功率消耗。

(e) 该PMOS三极管的导通电阻。

(f) 假设该反相器的驱动电容负载为70fF。请利用式（3.4）计算从低到高和从高到低的传播延迟。

解：(a) 该PMOS三极管是饱和的，由此得到

$$I_{stat}=\frac{1}{2}k_p'\frac{W_p}{L_p}(V_{DD}-V_T)^2$$
$$=12\frac{\mu A}{V^2}\times 1\times(5\ V-1\ V)^2=192\ \mu A$$

(b) 利用式（3.3），

$$R_{DS}=1\Big/\left[k_n'\frac{W_n}{L_n}(V_{GS}-V_T)\right]$$
$$=1\Big/\left[0.060\frac{mA}{V^2}\times 4\times(5V-1V)\right]=1.04\ k\Omega$$

(c) 利用从例3.13推导得到的表达式，我们得到

$$k_p=k_p'\frac{W_p}{L_p}=24\frac{\mu A}{V^2}$$
$$k_n=k_n'\frac{W_n}{L_n}=240\frac{\mu A}{V^2}$$
$$V_{OL}=V_f=(5\ V-1\ V)\left(1-\sqrt{1-\frac{24}{240}}\right)$$
$$=0.21\ V$$

(d)

$$P_D=I_{stat}\times V_{DD}$$
$$=192\ \mu A\times 5\ V=960\ \mu W\approx 1\ mW$$

(e)

$$R_{SDP}=V_{SD}/I_{SD}$$
$$=(V_{DD}-V_f)/I_{stat}$$
$$=(5\ V-0.21\ V)/0.192\ mA=24.9\ k\Omega$$

(f) 从低电平到高电平的传播延迟是

$$t_{pLH}=\frac{1.7C}{k_p'\frac{W_p}{L_p}V_{DD}}$$
$$=\frac{1.7\times 70\ fF}{24\frac{\mu A}{V^2}\times 1\times 5\ V}=0.99\ ns$$

从高电平到低电平的传播延迟是

$$t_{PHL} = \frac{1.7C}{k_n' \frac{W_n}{L_n} V_{DD}}$$

$$= \frac{1.7 \times 70 \text{ fF}}{60 \frac{\mu A}{V^2} \times 4 \times 5 \text{ V}} = 0.1 \text{ ns}$$

例3.15 在图3-69所示的电路中，我们展示了把NMOS三极管用作电路通道来解决静态功率消耗问题。假设在这个电路中把PMOS上拉三极管拿走。假设参数$k_n' = 60\ \mu A/V^2$，$k_p' = 0.5k_n'$，$W_n/L_n = 2.0\mu m/0.5\mu m$，$W_p/L_p = 4.0\mu m/0.5\mu m$，$V_{DD} = 5V$，且$V_T = 1$ V。当$V_B = 3.5$ V时，请计算下列值：

(a) 静态电流I_{stat}。

(b) 反相器的输出电压V_f。

(c) 反相器的静态功率消耗。

(d) 若芯片内包含以这种方式工作的250 000个反相器，求出总的静态功率消耗。

解：(a) 若假设该PMOS三极管工作在饱和区，则流经反相器的电流可以用下面的表达式定义：

$$I_{stat} = \frac{1}{2} k_p' \frac{W_p}{L_p} (V_{GS} - V_{TP})^2$$

$$= 120 \frac{\mu A}{V^2} \times ((3.5 \text{ V} - 5 \text{ V}) + 1 \text{ V})^2 = 30 \ \mu A$$

(b) 由于流经PMOS三极管的静态电流I_{stat}，也流过NMOS三极管，我们还假设该三极管工作在三极管工作区，因此可以得到下面的表达式：

$$I_{stat} = k_n' \frac{W_n}{L_n} \left[(V_{GS} - V_{Tn}) V_{DS} - \frac{1}{2} V_{DS}^2 \right]$$

$$30 \ \mu A = 240 \frac{\mu A}{V^2} \times \left[2.5 \text{ V} \times V_f - \frac{1}{2} V_f^2 \right]$$

$$1 = 20 V_f - 4 V_f^2$$

求解上面的方程，得到 $V_f = 0.05$V。请注意输出电压V_f满足PMOS三极管工作在饱和区，而NMOS三极管工作在三极管工作区的假设。

(c) 该反相器的静态功率消耗为

$$P_S = I_{stat} \times V_{DD} = 30 \ \mu A \times 5 \text{ V} = 150 \ \mu W$$

(d) 250 000个反相器的总静态功率消耗为

$$250\ 000 \times P_S = 37.5 \text{ W}$$

练习题

标有*号的习题答案参见书后。

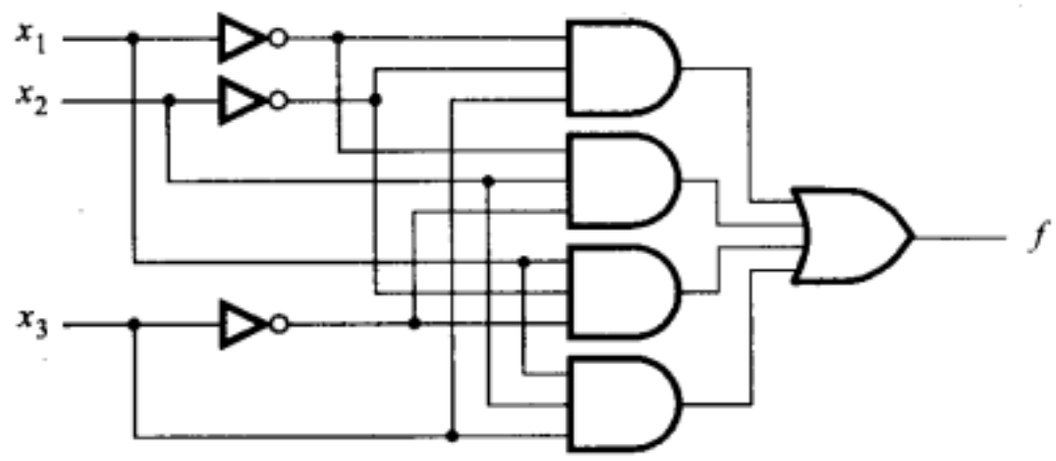
3.1 请考虑图P3-1所示的电路。

(a) 写出逻辑函数 f 的真值表。

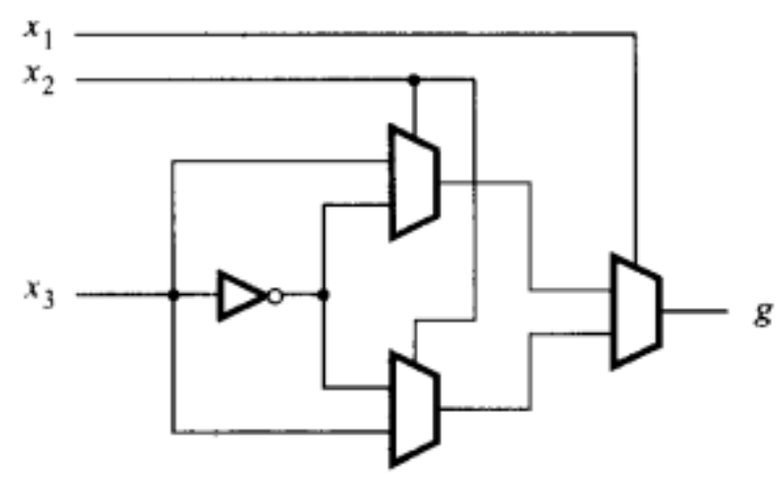
(b) 如果电路中的每一个门都用CMOS门实现，共需要多少个晶体管?

3.2 (a) 对比图P3-2和图P3-1的电路，证明这两个电路的逻辑功能是等价的。

(b) 实现这个CMOS电路共需要多少个晶体管?



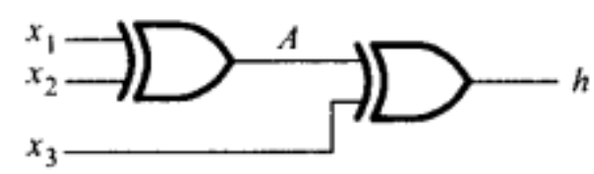
图P3-1 积之和CMOS电路



图P3-2 用多路器构建的CMOS电路

3.3 (a) 对比图P3-3和图P3-2的电路，证明这两个电路的逻辑功能是等价的。

(b) 如果此CMOS电路中的每一个异或门都用图3-61d所示电路实现，需要多少个晶体管?



图P3-3 练习题3.3的电路

* **3.4** 在3.8.8节中我们曾经介绍过，一个6输入的CMOS与门可以用两个3输入与门和一个2输入与门来构建，这样做共需要22个晶体管。只用CMOS与非门和或非门怎样构成6输入与门，并且计算出所需晶体管的数量（提示：使用德摩根定律）。

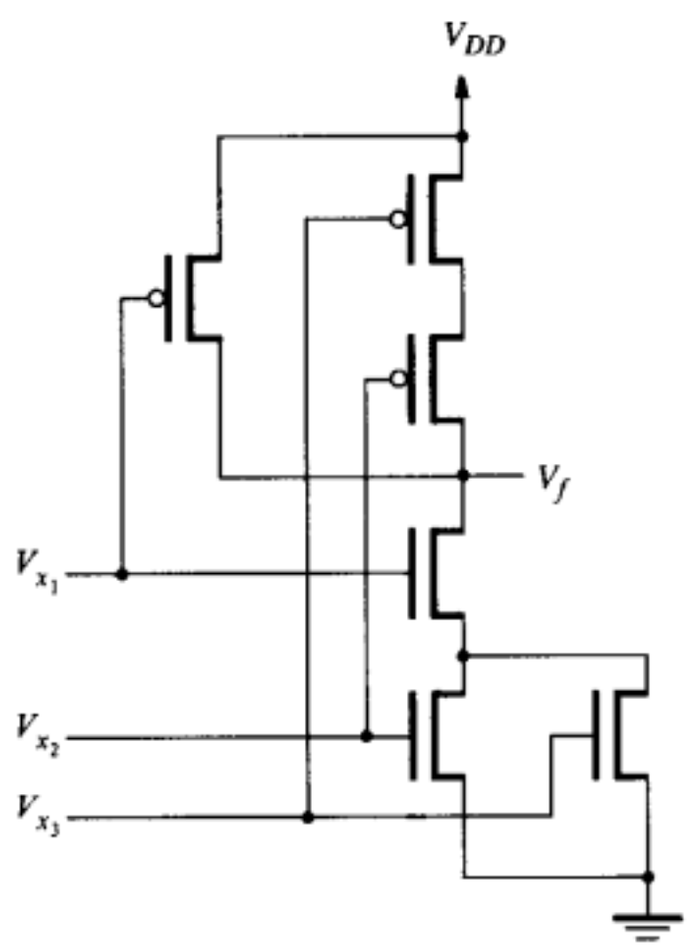
3.5 只用CMOS与非门和或非门怎样构成一个8输入CMOS或门，重复上题。

3.6 (a)写出图P3-4所示CMOS电路的真值表。

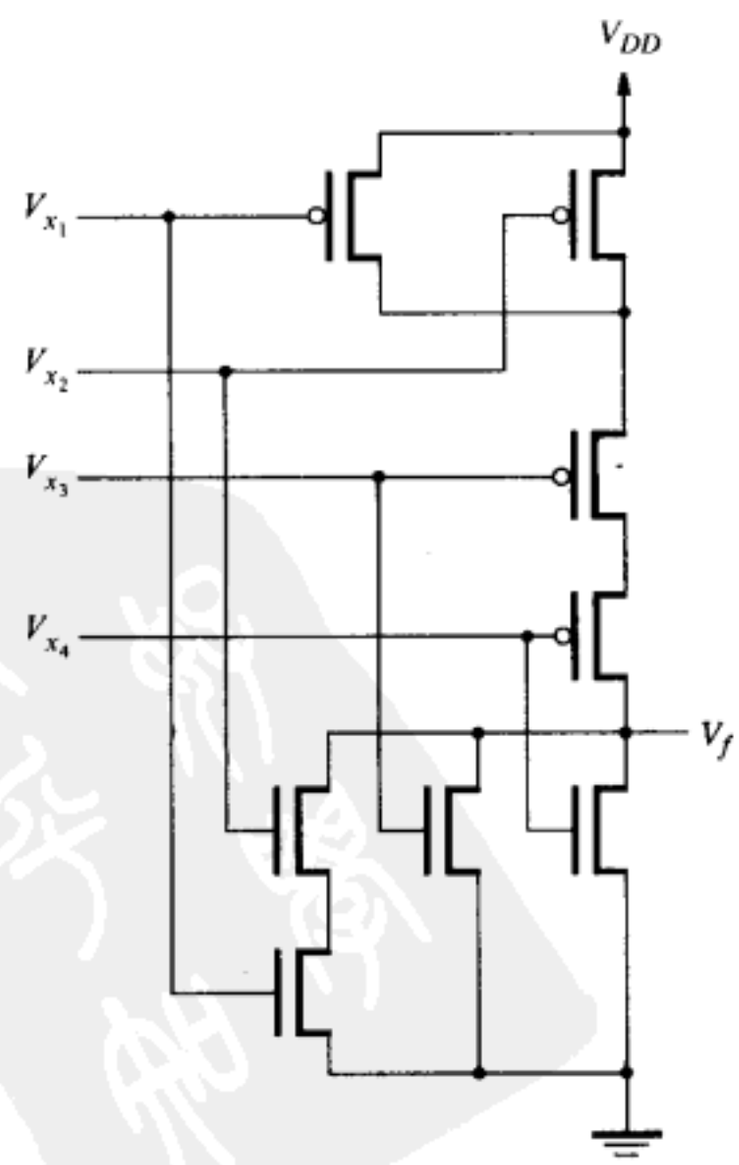
(b) 推导出本题（a）所得真值表的正则积之和形式的表达式。如果仅使用CMOS与门、或门和非门实现一个代表此表达式的电路，总共需要多少个晶体管?

3.7 (a) 请给出图P3-5所示CMOS电路的真值表。

(b) 推导（a）真值表的最简单的积之和形式。如果只用CMOS与门、或门和非门实现此电路，总共需要多少个晶体管?



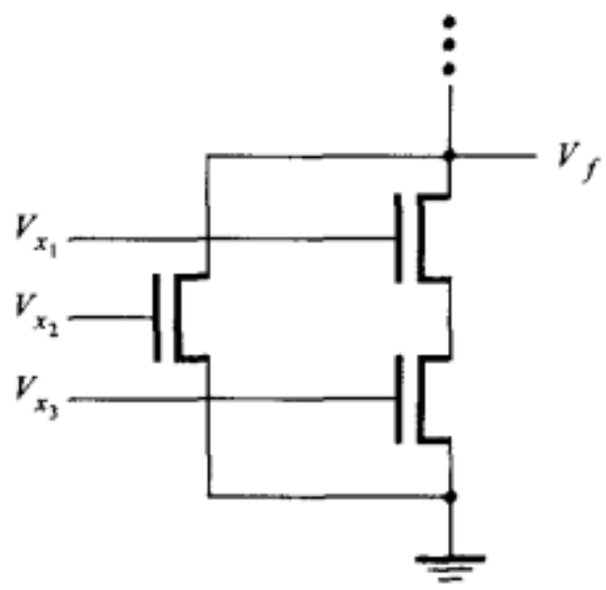
图P3-4 三输入的CMOS电路



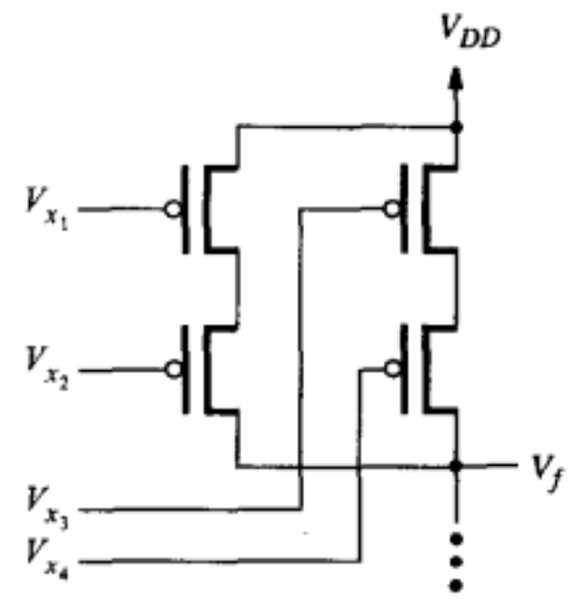
图P3-5 四输入的CMOS电路

*** 3.8** 图P3-6所示的为CMOS电路的一半，画出电路中包含PMOS晶体管的另外一半。

3.9 图P3-7所示为CMOS电路的一半，画出电路中包含NMOS晶体管的另外一半。



图P3-6 CMOS电路中的下拉网络



图P3-7 CMOS电路中的上拉网络

3.10 推导实现逻辑功能 $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 10)$ 的CMOS的复杂门电路。

3.11 推导实现逻辑功能 $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 6, 8, 10, 12, 14)$ 的CMOS的复杂门电路。

*** 3.12** 推导使用尽量少的晶体管实现逻辑函数 $f = xy + xz$的CMOS复杂门电路（提示：考虑 $\overline{f}$）。

3.13 推导使用尽量少的晶体管实现逻辑函数 $f = xy + xz + yz$的CMOS复杂门电路（提示：考虑 $\overline{f}$）。

*** 3.14** 假设NMOS晶体管的参数为：$k_n' = 20\ \mu A/V^2$，$W/L = 2.5\ \mu m/0.5\ \mu m$，$V_{GS} = 5\ V$，$V_T = 1\ V$。计算：

(a) 当$V_{DS} = 5$ V时的I_D。

(b) 当$V_{DS} = 0.2$ V时的I_D。

3.15 假设PMOS晶体管的参数为：$k_p' = 10\mu A/V^2$，$W/L = 2.5\ \mu m/0.5\ \mu m$，$V_{GS} = -5\ V$，$V_T = -1\ V$。计算：

(a) 当$V_{DS} = -5$ V时的I_D。

(b) 当$V_{DS} = -0.2$ V时的I_D。

3.16 假设NMOS晶体管的参数为：$k_n' = 20\ \mu A/V^2$，$W/L = 5.0\ \mu m/0.5\ \mu m$，$V_{GS} = 5\ V$，$V_T = 1\ V$。在$V_{DS}$值很小时，计算$R_{DS}$。

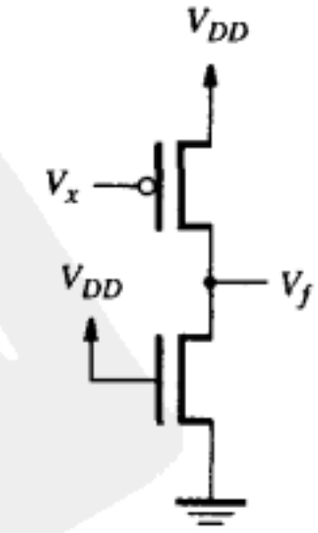
*** 3.17** 假设NMOS晶体管的参数为：$k_n' = 40\ \mu A/V^2$，$W/L = 3.5\ \mu m/0.35\ \mu m$，$V_{GS} = 3.3\ V$，$V_T = 0.66\ V$。在$V_{DS}$值很小时，计算$R_{DS}$。

3.18 假设PMOS晶体管的参数为：$k_p' = 10\ \mu A/V^2$，$W/L = 5.0\ \mu m/0.5\ \mu m$，$V_{GS} = -5\ V$，$V_T = -1\ V$，在$V_{DS} = -4.8$ V的情况下，计算R_{DS}。

3.19 假设PMOS晶体管的参数为：$k_p' = 16\ \mu A/V^2$，$W/L = 3.5\ \mu m/0.35\ \mu m$，$V_{GS} = -3.3\ V$，$V_T = -0.66\ V$，在 $V_{DS} = -3.2$ V的情况下，计算R_{DS}。

3.20 在例3.13中，我们展示了如何计算伪NMOS反相器中的电平。图P3-8给出了一个伪PMOS反相器。在这种技术下，用“弱”NMOS晶体管做下拉电阻。

当$V_x = 0$时，V_f的值很大。PMOS晶体管工作于三极管区域，此时NMOS晶体管正处于饱和区，所以阻止了电流的流过。流过NMOS和PMOS晶体管的电流必须相等，二者的关系由式（3.1）和式（3.2）确定。求高输出电压$V_f = V_{OH}$的表达式，用V_{DD}，V_T，k_p和k_n表示，其中k_p和k_n是增益因素，如例3.13中所定义。



图P3-8 伪NMOS反相器

3.21 对如图P3-8所示的电路，假设$k_n' = 60\ \mu A/V^2$，$k_p' = 0.4\ k_n'$，$W_n/L_n = 0.5\ \mu m/0.5\ \mu m$，$W_p/L_p = 4.0\ \mu m/0.5\ \mu m$，$V_{DD} = 5\ V$，$V_T = 1\ V$。当$V_x = 0$时，计算：

(a) 静态电流I_{stat}。

(b) PMOS晶体管的导通电阻。

(c) V_{OH}。

(d) 反相器的静态功耗。

(e) NMOS晶体管的导通电阻。

(f) 假设反相器的负载电容为70 fF，使用（3.4）式计算反相器的上升传播延迟和下降传播延迟。

3.22 假设习题3.21的NMOS晶体管尺寸改变为$W_n/L_n = 4.0\ \mu m/0.5\ \mu m$，重新计算练习题3.21。

3.23 例3.13（见图3-72）展示了伪NMOS技术，上拉器件用PMOS晶体管来实现。用伪NMOS技术实现一个与非门，重复计算这个问题。假定门电路中两个NMOS晶体管的参数相同，如例3.14中所给出。

3.24 对伪NMOS技术实现的或非门，重复习题3.23的计算。

*** 3.25** (a) 设$V_{IH} = 4\ V$，$V_{OH} = 4.5\ V$，$V_{IL} = 1\ V$，$V_{OL} = 0.3\ V$，$V_{DD} = 5\ V$，计算噪声容限NM_H和NM_L。

(b) 用NMOS技术实现一个8输入与非门，若每经过一个晶体管电压下降0.1 V，求V_{OL}的值。如果参数和（a）中不一样，对应的NM_L是多少？

3.26 稳定状态下n输入的CMOS与非门的V_{OL}和V_{OH}电平是多少？请解释。

3.27 假设CMOS反相器的负载电容$C = 150$ fF，$V_{DD} = 5\ V$，反相器的电平以平均频率为$f = 75$ MHz在高、低电平之间周期性地变化。

(a) 计算反相器的动态功耗。

(b) 若有一芯片，其包含的晶体管个数等价于250 000个反相器，假定在任意时刻有20%的逻辑门正在改变状态，计算该芯片的总动态功耗。

*** 3.28** 假设：$C = 120$ fF，$V_{DD} = 3.3\ V$，$f = 125$ MHz，重新计算习题3.27。

3.29 对于CMOS反相器，假设$k_n' = 20\ \mu A/V^2$，$k_p' = 0.4\ k_n'$，$W_n/L_n = 5.0\ \mu m/0.5\ \mu m$，$W_p/L_p = 5.0\ \mu m/0.5\ \mu m$，$V_{DD} = 5\ V$，负载电容为150 fF。计算：

(a) 由高到低的传播延迟。

(b) 由低到高的传播延迟。

(c) 使反相器的由低电平到高电平和由高电平到低电平的传播延迟相等，PMOS晶体管的尺寸应为何值？在计算中忽略PMOS晶体管尺寸对反相器的负载电容的影响。

3.30 假设$k_n' = 40\ \mu A/V^2$，$k_p' = 0.4k_n'$，$W_n/L_n = W_p/L_p = 3.5\ \mu m/0.35\ \mu m$，$V_{DD} = 3.3\ V$，重新做习题3.29。

3.31 假设CMOS反相器的$W_n/L_n = 2$，$W_p/L_p = 4$。设计一个CMOS与非门，使其输出的驱动电流，无论高电平和低电平时，都等于反相器的电流，试求该与非门中NMOS晶体管和PMOS晶体管的宽长比W/L。

*** 3.32** 设计一个CMOS或非门，重新做习题3.31。

3.33 设计一个如图3-16所示的CMOS复杂门，重新做习题3.31。选择晶体管的尺寸，使得最坏情况下的驱动电流至少和反相器一样大。

3.34 设计一个如图3-17所示的CMOS复杂门，重新做习题3.31。

3.35 图3-69所示为NMOS导通晶体管静态功耗问题的一个解决方案。假定把PMOS上拉晶体管从电路中移走，晶体管的参数如下：$k_n' = 60\ \mu A/V^2$，$k_p' = 0.4\ k_n'$，$W_n/L_n = 1.0\ \mu m/0.25\ \mu m$，$W_p/L_p = 2.0\ \mu m/0.25\ \mu m$，$V_{DD} = 2.5\ V$，$V_T = 0.6\ V$。对$V_B = 1.6\ V$，计算：

(a) 静态电流I_{stat}。

(b) 反相器输出电平V_f。

(c) 反相器的静态功耗。

(d) 如果某个芯片中包含50万个这样的反相器，求出总静态功耗。

3.36 用如图3-66所示的风格画出已编程的PLA的示意图，实现函数为 $f_1(x_1, x_2, x_3) = \sum m(1, 2, 4, 7)$。PLA的输入为$x_1$，$x_2$，$x_3$，乘积项为$P_1$，…，$P_4$，输出为$f_1$和$f_2$。

3.37 用图3-66所示的风格画出已编程的PLA的示意图，实现函数为 $f_1(x_1, x_2, x_3) = \sum m(0, 3, 5, 6)$。PLA的输入为$x_1$，$x_2$，$x_3$，乘积项为$P_1$，…，$P_4$，输出为 f_1和f_2。

3.38 用图3-65所示的风格实现习题3.36的函数f_1，画出该PLA的编程图，该PLA的输入为x_1，x_2，x_3，和项为S_1，…，S_4，输出为f_1和f_2。

3.39 用图3-65所示的风格实现习题3.37的函数f_1，画出该PLA的编程图，该PLA的输入为x_1，x_2，x_3，和项为S_1，…，S_4，输出为f_1和f_2。

3.40 用图3-63所示的PLA编程示意图风格，重做习题3.38。

3.41 用图3-63所示的PLA编程示意图风格，重做习题3.39。

3.42 若f_1已经如习题3.36描述的那样实现了，列出能用该PLA的输出f_2实现的所有逻辑函数。

3.43 若f_1已经如习题3.37 描述的那样实现了，列出能用该PLA的输出f_2实现的所有逻辑函数。

3.44 用五个2输入查找表（LUT）实现函数：$f(x_1, x_2, x_3) = x_1\overline{x}_2 + x_1x_3 + x_2\overline{x}_3$。用图3-39所示的风格标出每个LUT的真值表，不必画出FPGA中的连线。

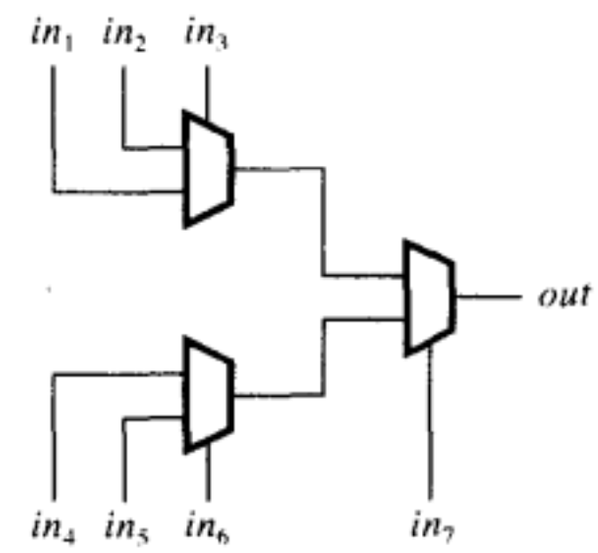
***3.45** 用两个2输入查找表（LUT）实现函数：$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$。用图3-39所示风格标出每个LUT的真值表，不必画出FPGA中的连线。

3.46 若逻辑函数为：$f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$，则可以用四个3输入的LUT在FPGA中直接实现。请用三个3输入的LUT来实现此函数，请以LUT所实现的逻辑函数来标记每个LUT的输出。

3.47 请用2输入LUT实现习题3.46的逻辑函数 f，画出其电路图。该电路的实现正好需要用七个2输入LUT，请以LUT所实现的逻辑函数来标记每个LUT的输出。

3.48 图3-39所示为一个已编程的实现某一函数的FPGA。芯片的一个引脚已用作函数 f 的输出，其他的引脚未被使用。不改变图中已经被编程为接通状态的开关，列出除函数 f 之外，可以在未被使用的引脚上实现的10个逻辑函数。

3.49 假设门阵列中包含如图P3-9所示的逻辑单元，输入in_1，…，in_7可以被连接到逻辑1或者逻辑0，或者其他任何逻辑信号。



图P3-9 门阵列逻辑单元

(a) 如何使用该逻辑单元实现函数 $f = x_1x_2 + x_3$。

(b) 如何使用该逻辑单元实现函数 $f = x_1x_3 + x_2x_3$。

3.50 假设门阵列电路中只有3输入与非门可用，每个与非门的输入可以被连接到逻辑1，或者逻辑0，或者其他任何逻辑信号。在该门阵列中如何实现以下逻辑函数（提示：用德摩根定律）：

(a) $f = x_1x_2 + x_3$。

(b) $f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1$。

3.51 编写VHDL代码实现函数：$f = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_4 + \overline{x}_1x_2x_3 + x_1x_2x_3$。

(a) 使用CAD工具，在某类型的芯片（如CPLD）中实现该逻辑函数。写出由工具产生的表示 f 的逻辑表达式。使用时序仿真，确定输入从x_1，x_2，x_3变化到输出 f 所需要的传播时延。

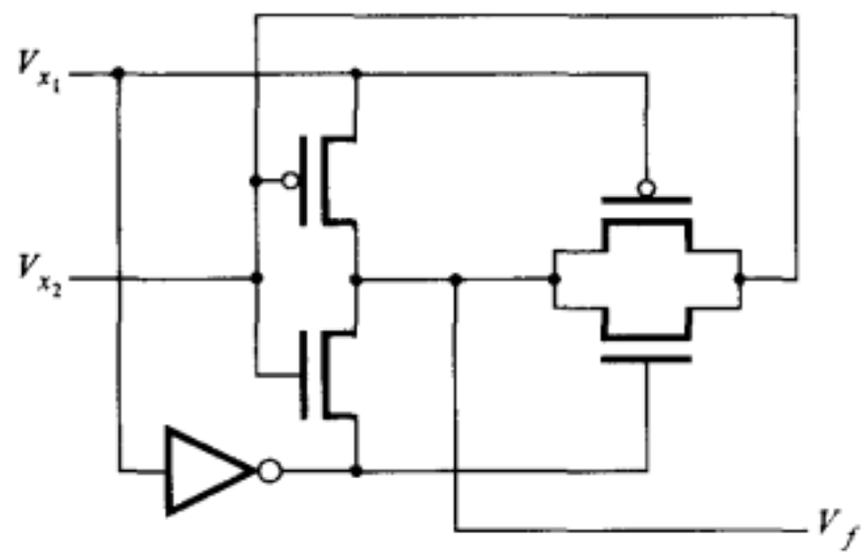
(b) 使用不同的芯片（如用FPGA）重做（a）题，实现该电路。

3.52 设逻辑函数为：$f = (x_1 + x_2 + \overline{x}_4)(\overline{x}_2 + x_3 + \overline{x}_4)(\overline{x}_1 + x_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$，重做习题3.51。

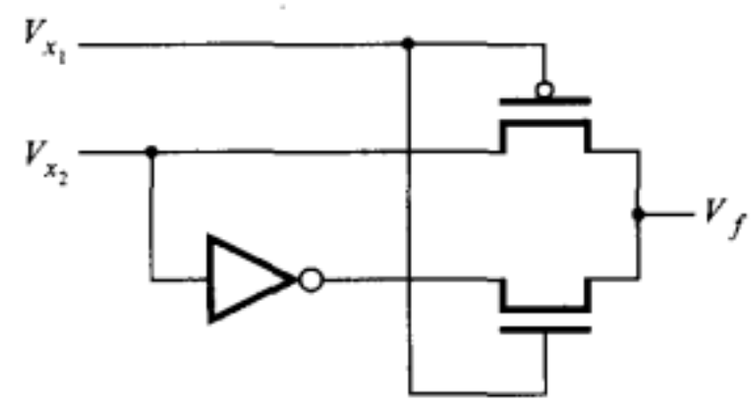
3.53 设逻辑函数为：$f(x_1, \cdots, x_7) = x_1x_3\overline{x}_6 + x_1x_4x_5\overline{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$，重做习题3.51。

3.54 图P3-10所示电路实现的是什么逻辑门？该电路有什么主要缺点？

*** 3.55** 图P3-11所示电路实现的是什么逻辑门？该电路有什么主要缺点？



图P3-10 练习题3.54的电路



图P3-11 练习题3.55的电路

参考文献

1. A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 5th ed. (Oxford University Press: New York, 2003).
2. J. M. Rabaey, *Digital Integrated Circuits*, (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. Texas Instruments, *Logic Products Selection Guide and Databook CD-ROM*, 1997.
4. National Semiconductor, *VHC/VHCT Advanced CMOS Logic Databook*, 1993.
5. Motorola, *CMOS Logic Databook*, 1996.
6. Toshiba America Electronic Components, *TC74VHC/VHCT Series CMOS Logic Databook*, 1994.
7. Integrated Devices Technology, *High Performance Logic Databook*, 1994.
8. J. F. Wakerly, *Digital Design Principles and Practices* 3rd ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
9. M. M. Mano, *Digital Design* 3rd ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
10. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
11. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
12. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

第4章　逻辑函数的优化实现

在第2章中我们曾指出，对逻辑函数进行代数运算（化简）可以用于找到其最低成本的实现。该章的目的是引入综合过程的基本概念。读者也许认为，由逻辑函数的正则形式就能很容易地直接得到其电路的实现，然而事实却并非如此。合理地选择和应用2.5节中的定理及其性质，找到最低成本的电路实现并非是一件轻而易举的任务。实际上，对于多变量的函数而言，运用代数定理进行化简是极其繁冗和不切实际的。

若在逻辑电路的设计工作中应用了计算机辅助设计（CAD）工具，则对逻辑电路进行优化以降低实现电路的成本的工作负担可以减轻。CAD工具能够自动地担当起必要的优化工作。即便如此，对优化过程有所了解仍是很必要的。因为大多数CAD工具都有许多用户可选择的特性和控制选项，为了要知道什么时候用到和怎样应用这些选项，用户必须要理解CAD工具所做的事情。

本章将要介绍CAD工具实现优化时用到的一些技术，并展示这些技术如何应用于自动化的CAD工具。我们首先要讨论的是图表法，即称为卡诺图的方法，该方法为我们提供了方便快捷的途径，可以利用手工的方法得到简单逻辑函数的最低成本的电路实现。尽管这种方法并不适用于CAD工具，但是它说明了许多关键的概念。我们将演示如何设计两级及多级电路，然后讲述适用于CAD工具的逻辑函数的立方体表示法。我们还将进一步讨论Verilog语言。

4.1　卡诺图

在2.6节中我们注意到，找到给定逻辑函数的最低成本表达式的关键是通过尽可能合理地运用合并性质14a（或14b），来减少表达式中所需的乘积项（或者是和项）的数量。卡诺图为实现这种优化提供了一种系统化的方法。为了理解这种方法的工作原理，有必要先复习一下第2章介绍的代数方法。考虑图4-1中的逻辑函数 f 的真值表，其正则积之和表达式由最小项 m_0、m_2、m_4、m_5和m_6组成，因此，

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

合并律（14a）告诉我们：可以将表达式中两个仅有一个变量取值不同的最小项替换为一个不含该变量的乘积项。例如，m_0和m_2都包含$\bar{x}_1$和$\bar{x}_3$，在这两项中仅有x_2取值不相同，这是因为m_0中包含$\bar{x}_2$而m_2中包含x_2，因而得到：

$$\begin{aligned}\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 &= \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3\\ &= \bar{x}_1\cdot 1\cdot\bar{x}_3\\ &= \bar{x}_1\bar{x}_3\end{aligned}$$

因此m_0和m_2可以被单独的乘积项$\bar{x}_1\bar{x}_3$替代。同样，m_4和m_6只在x_2的取值上不同，因而可以合并，得到：

$$\begin{aligned}x_1\bar{x}_2\bar{x}_3 + x_1x_2\bar{x}_3 &= x_1(\bar{x}_2 + x_2)\bar{x}_3\\ &= x_1\cdot 1\cdot\bar{x}_3\\ &= x_1\bar{x}_3\end{aligned}$$

行号	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

图4-1　函数 $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$

现在这两个新生成的乘积项$\overline{x}_1\overline{x}_3$和$x_1\overline{x}_3$，还可以进一步合并为：

$$\begin{aligned}\overline{x}_1\overline{x}_3 + x_1\overline{x}_3 &= (\overline{x}_1 + x_1)\overline{x}_3 \\ &= 1\cdot\overline{x}_3 \\ &= \overline{x}_3\end{aligned}$$

以上优化步骤说明我们可以用单独一个乘积项$\overline{x}_3$来代替四个最小项m_0、m_2、m_4和m_6。换句话说，m_0、m_2、m_4和m_6这几个最小项全部被蕴涵在乘积项$\overline{x}_3$中。f 的表达式中还剩下一个最小项m_5可以与m_4合并如下：

$$x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 = x_1\overline{x}_2$$

根据2.5节定理7b，得到：

$$m_4 = m_4 + m_4$$

这就意味着我们可以两次使用最小项m_4，第一次与最小项m_0、m_2和m_6合并生成乘积项$\overline{x}_3$，第二次与m_5合并生成乘积项$x_1\overline{x}_2$。

现在我们已对 f 中所有的最小项进行了化简；得到下面成本最低的表达式，它覆盖了原先的5个最小项。

$$f = \overline{x}_3 + x_1\overline{x}_2$$

表达式中含有乘积项$\overline{x}_3$，因为当$x_3 = 0$时就有 $f = 1$，而不必考虑x_1和x_2的取值。m_0、m_2、m_4和m_6这四个最小项代表了$x_3 = 0$时的所有可能情况，即x_1和x_2的所有四种取值00、01、10 和11。因而若$x_3 = 0$，便可以保证 $f = 1$。这点可能从图4-1的真值表中不易直接看出，但是若我们从真值表中将有关项取出整理在一起（见下表所示），情况就显而易见了。

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

同样，如果将m_4和m_5两项的取值组合起来，可以看到如下表所示：

	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

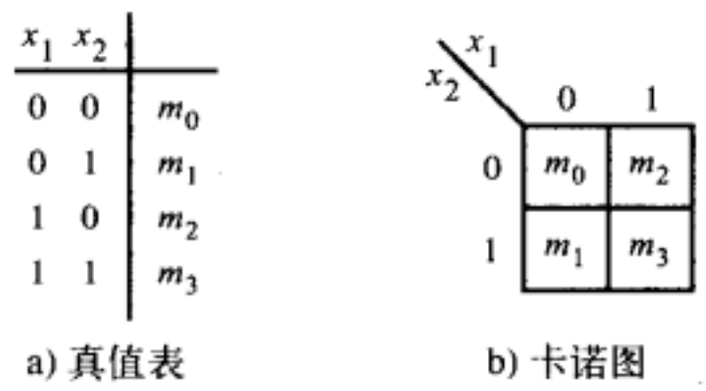
很明显，若$x_1 = 1$并且$x_2 = 0$，则无论x_3取什么值，$f = 1$。

上述讨论提示我们，如果能想出一种方法，使我们能很容易地找出那些令$f = 1$的最小项组合，并且能将那些最小项合并成若干独立的乘积项，将是十分有用的。卡诺图是达到上述目的一种行之有效的工具。

卡诺图[1]和真值表都可以用来表达逻辑函数。卡诺图由对应于真值表各行的单元组成。考虑如图4-2所示的二变量逻辑函数。图4-2a是真值表的形式，共有四行，每行都对应一个最小项。图4-2b是一个四个单元的卡诺图。图中的列标记为 x_1，而行标记为x_2。如图所示，这些标记确定了最小项在卡诺图上的位置。与真值表相比，卡诺图的优点是很容易发现哪些最小项可以合并（应用2.5节中的合并律14a）。无论在同一行还是在同一列中，任意两个相邻单元里的最小项都可以合并。例如，最小项m_2和m_3可以按下式合并：

$$
\begin{aligned}
m_2 + m_3 &= x_1\bar{x}_2 + x_1x_2 \\
&= x_1(\bar{x}_2 + x_2) \\
&= x_1 \cdot 1 \\
&= x_1
\end{aligned}
$$

卡诺图不只对于合并成对的最小项有用。我们还将在几个较大的示例中看到，可以用卡诺图来直接得到逻辑函数的最低成本电路。



a) 真值表　　b) 卡诺图

图4-2　二变量最小项的位置

二变量卡诺图

图4-3所示的是一张二变量逻辑函数的卡诺图。它与图2-15所示的函数 f 相对应。在输入变量x_1和x_2的不同取值情况下，f 的输出值全都标记在图中对应的单元格内。在最下面一行的两个单元中都标记为1，并且是相邻的，因此就可以合并，合并后的乘积项等价于合并前的两个最小项。即当输入变量的值对应于这两个单元中任意一个的时候，该乘积项就会使 $f = 1$。为说明这个情况，我们在图中用一个长圆形把这两个单元圈了起来。现在我们就可以在卡诺图上直接得到这个乘积项，而不必使用代数形式的合并律来合并最小项。这两个单元格对应的都是$x_2 = 1$，而左边的单元对应的输入是$x_1 = 0$，右边的单元则是$x_1 = 1$。于是若$x_2 = 1$，则不论x_1取值为0还是1，f 均等于1。因此这两个单元的乘积项就简化为x_2。

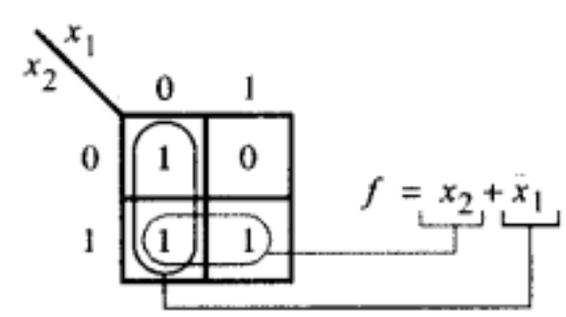


图4-3　图2-15的函数

同样，卡诺图第一列中的两个单元格均被标记为1（即函数输出为 1）。而这两个单元格对应的都是$x_1 = 0$。因此推导出乘积项为$\bar{x}_1$。因为我们只关心 f =1的所有情况，因此可以得到该函数的最低成本实现如下：

$$f = x_2 + \bar{x}_1$$

显然，为了找到给定函数的最低成本实现，必须找出输入变量个数最少的乘积项，而这些乘积项能够产生1值，以满足在所有输入条件下都能使 f =1。并且，这些乘积项的成本应该尽可能的低。我们注意到，覆盖两个相邻单元的乘积项与仅覆盖一个单元的项相比较，其实现成本更低。如在本例中，一旦下面一行的两个单元被乘积项x_2所覆盖，在卡诺图中就只留下一个单元（左上角）格。尽管这一项可以用$\bar{x}_1\bar{x}_2$覆盖，但是更佳的选择是左边一列两个单元合并后生成的乘积项$\bar{x}_1$，因为这样做可以使实现成本更低。

三变量卡诺图

三变量卡诺图是由两张二变量卡诺图并排构成的。图4-4显示了这样的图，图中标明了最小项的位置。在这种情形下，输入变量x_1和x_2的每一个取值组合决定图中的一列，而x_3的取值区分开两行。为确保图中相邻单元中的最小项都可以被合并成一个独立的乘积项，相邻单元间的取值必须只在一个变量上有不同。因此各列是由(x_1, x_2)等于00，01，11，10的取值顺序来确定的，而非常见的00，01，10，11的顺序。这就使得二、三列只在变量x_1上取值不同。同样，一、四列也只在变量x_1上取值不同，这就意味着这两列也可以看成是相邻的。读者可能会发现，将一张卡诺图假想为一个矩形，将其卷成一个使得图4-4b中的左右边缘可以接上的圆筒，这样就不难理解了。（一个代码序列，或者取值序列，它们中相邻的代码之间只在一个变量上取值不同，这种编码称为格雷码（Gray code）。格雷码有多种用途，其中一些将在本书的后面介绍。）

x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

a) 真值表

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

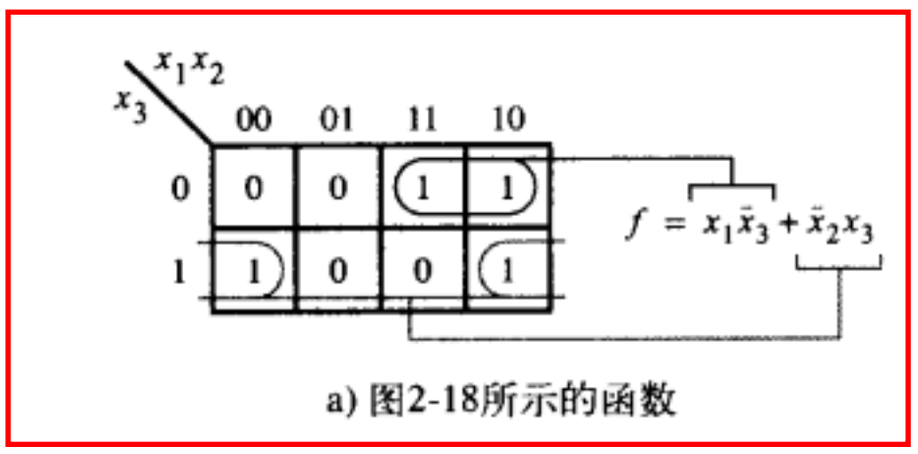
b) 卡诺图

图4-4 三变量卡诺图最小项的位置

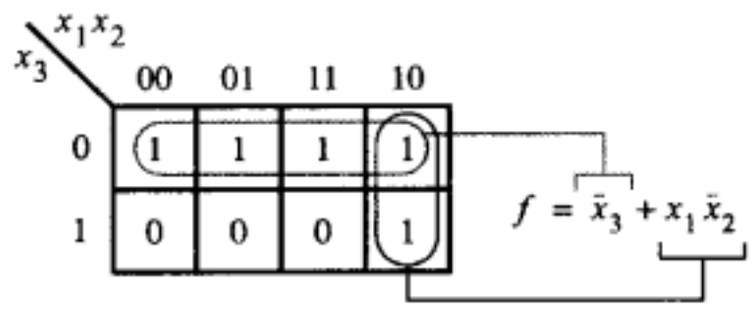
图4-5a以卡诺图的形式展示了图2-18所示的函数。为综合这个函数，必须尽可能有效地将图中的4个标记着1的单元都覆盖。不难看出，用两个乘积项就能满足这个要求，第一个乘积项覆盖了上面一行的1，可以用$x_1\overline{x}_3$来表示。第二个乘积项是$\overline{x}_2x_3$，它覆盖了下面一行的1。因此该函数的实现为：

$$f = x_1\overline{x}_3 + \overline{x}_2x_3$$

上式描述了图2-19a所示的电路。



a) 图2-18所示的函数



b) 图4-1所示的函数

图4-5 三变量卡诺图的示例

在三变量的卡诺图中，可以将多个单元合并为一个乘积项，这种合并可以基于一个单元，或两个相邻的单元，或四个相邻单元组进行。在图4-5b中阐明了如何用一个乘积项来实现四相邻单元的合并，其中用到了图4-1中的函数。上面一行中的四个单元分别对应于（x_1, x_2, x_3）取值为000、010、110和100的四种情况。正如以前所讨论过的那样，这说明若$x_3 = 0$，则对于x_1和x_2的所有四种可能取值组合f都等于1，换言之，令$f = 1$的唯一必要的条件是$x_3 = 0$。因此，乘积项$\overline{x}_3$涵盖了这四个单元的内容。留下的标记为1的单元对应于最小项m_5，合并最右边一列的两个单元，可以被乘积项$x_1\overline{x}_2$很好地覆盖。于是得到函数f的完全实现为：

$$f = \overline{x}_3 + x_1\overline{x}_2$$

在三变量图中8个单元全部标记为1的情况也是有可能发生的。这种情况并不少见，即无论输入变量取什么值，函数f都为1。换言之，f等于常量1。

卡诺图提供了一种简单的机制，可以为给定函数产生优化实现所需的乘积项。一个乘积项包含的变量必须符合下面的规则：该乘积项对应单元组里取值始终一致的变量。若变量在该单元组里取值为1，则它在乘积项中以原变量的形式出现；如果此变量在该单元组里取值为0，则它在乘积项中以反变量的形式出现。单元组里时而取值为1、时而取值为0的变量，不应该出现在乘积项中。

四变量卡诺图

四变量卡诺图由两张三变量卡诺图放置在一起构成，共有四行，正如我们用两张二变量

卡诺图构成四列的三变量卡诺图的方式一样。四变量卡诺图的结构和最小项的位置如图4-6所示。我们在这张图中还引入了另一种经常用到的指定行和列的方法，即只指定给定变量等于1所对应的行和列就足够了，见图上大括号指示的部分。因此最右边两列对应于$x_1 = 1$，中间两列对应于$x_2 = 1$，底部两行对应于$x_3 = 1$，中间两行对应于$x_4 = 1$。

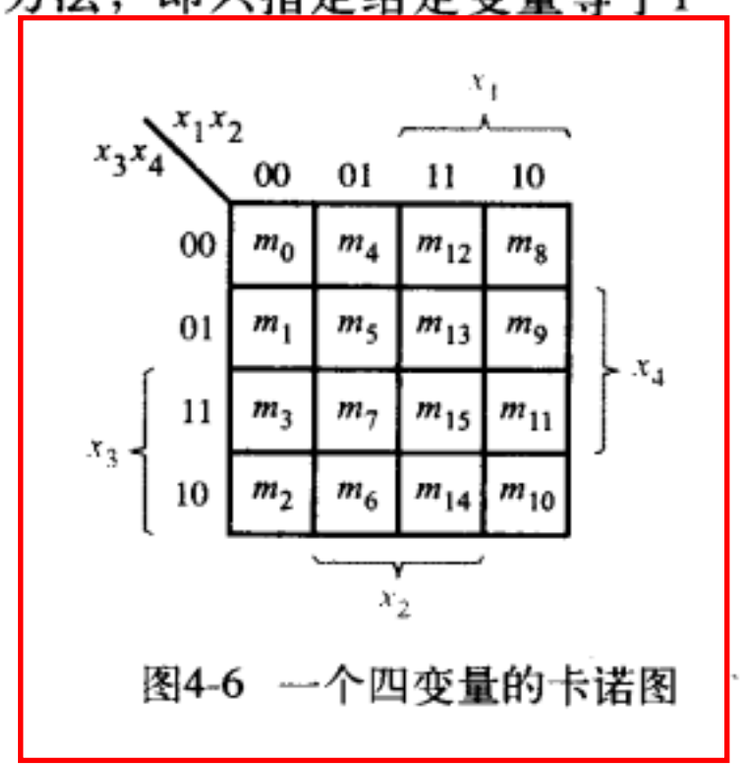


图4-6 一个四变量的卡诺图

图4-7给出了4个四变量函数的例子。函数 f_1在卡诺图的底下两行有标记为1的相邻的四个单元组成的组，对应于$x_2 = 0$ 和$x_3 = 1$，这些单元可以用乘积项$\overline{x}_2x_3$来表示。这样就只剩下第二行中的两个标记为1的单元还未覆盖，它们可以由项$x_1\overline{x}_3x_4$来覆盖。因此该函数的最低成本实现为：

$$f_1 = \overline{x}_2x_3 + x_1\overline{x}_3x_4$$

函数 f_2包含了一个八个1相邻的单元组，可以用一个乘积项x_3实现。读者应该再次注意，若将剩下的两个标记为1的单元单独实现，则得到乘积项$x_1\overline{x}_3x_4$。若如图所示的那样，把这两个1作为相邻四个1的组中的一部分来实现，就会得到成本更低的乘积项x_1x_4。

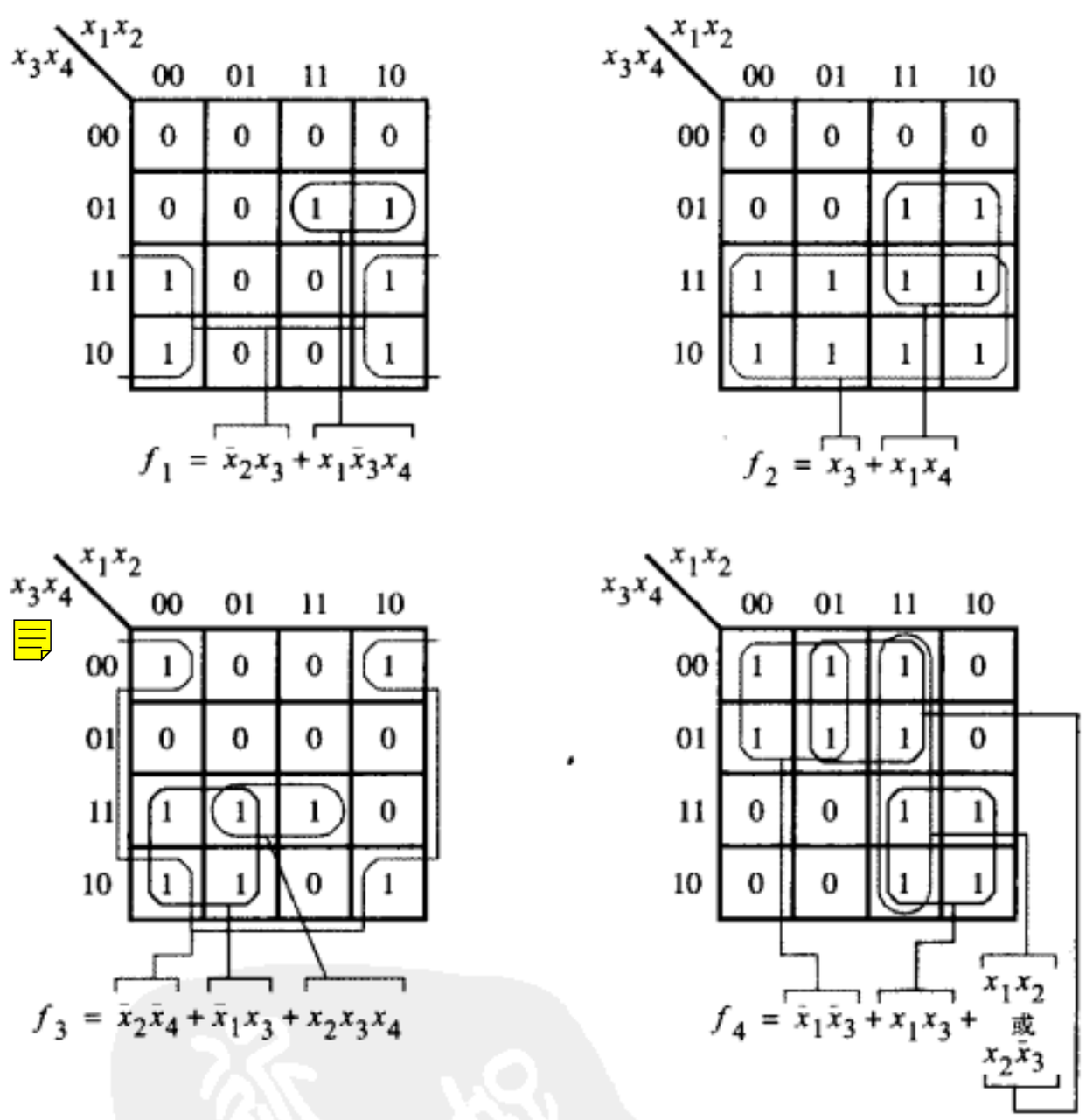


图4-7 四变量卡诺图的例子

根据变量的分配方式，卡诺图左右边界是相邻的，顶部和底部的边界也是相邻的。实际上，图中的四个角也是相邻的，因而可以形成相邻四个1的组，并可用乘积项$\overline{x}_2\overline{x}_4$实现。这正是函数 f_3所描述的情况。除了这组1之外，实现 f_3还需要覆盖另外四个1，实现方式见图4-7所示。

到目前为止，我们所举的例子，都只存在一种方案来实现成本最低的电路，而函数 f_4的最低成本解决方案可以有多种选择。可以分别用乘积项$\overline{x}_1\overline{x}_3$和$x_1x_3$实现图中左上角和右下角由四个1构成的组，还剩下对应于乘积项$x_1x_2\overline{x}_3$的两个1没有覆盖。但是这两个1可以视为由相邻的四个1构成的组的一部分，从而得到成本更低的实现。这两个1可以归于两个不同的四元组，

如图4-7所示。

一种选择导致了乘积项x_1x_2，另一种则导致$x_2\overline{x}_3$。这两个乘积项的成本是一样的；因而从成本的角度来看，最终电路选择哪个乘积项是无所谓的。注意，这里与x_1x_2相比较，$x_2\overline{x}_3$中x_3的求反不会带来更多的成本，其原因是在函数的实现中包含有乘积项$\overline{x}_1\overline{x}_3$，而产生$\overline{x}_1\overline{x}_3$时，无论如何都已经存在$x_3$的反，因此$x_3$求反的成本已经体现在$\overline{x}_1\overline{x}_3$中了。

五变量卡诺图

我们可以用两张四变量卡诺图来构建一张五变量卡诺图。很容易想象出一张四变量卡诺图紧贴在另一张后面的结构，两张图的区别在于：一张对应于$x_5=0$，而另一张则对应于$x_5=1$。由于这样的结构很难画出，故我们只是简单地将两张图并排放在一起，如图4-8所示。对于此例中的函数，两个由相邻四个1构成的组同时出现在两张四变量图中的同样位置，因而它们的实现不依赖于x_5的取值。在第二行中的两个由相邻的两个1构成的组也是一样的。右上角为1的最小项只出现在右边（对应于$x_5=1$）的那张图中，该项由乘积项$x_1\overline{x}_2\overline{x}_3x_5$实现，包含在由相邻两个1构成的组中。

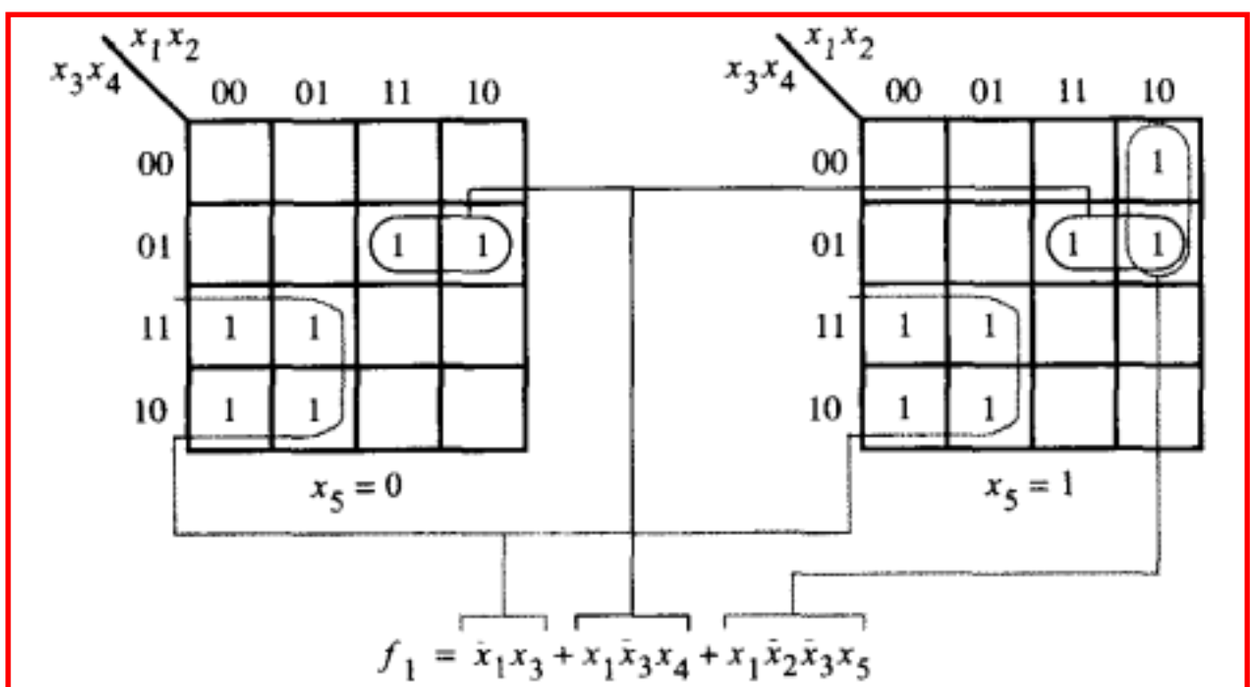


图4-8 五变量卡诺图

请注意在图4-8中，我们让对应于$f=0$的单元格都空着，这使得卡诺图的可读性更好。对下面很多的卡诺图，我们都将这样处理。

五变量卡诺图的应用很明显要比变量少的卡诺图难以掌握。从实用的角度来看，将卡诺图的概念延伸至多变量的情形其实用性并不强。但多变量卡诺图化简并不会给设计者带来很多麻烦，因为在实际工作中，逻辑函数必要的简化和综合都是由CAD工具自动完成的。尽管卡诺图只在设计小型的逻辑电路时会偶尔用到，我们在这里仍然介绍它，主要目的是通过卡诺图这样一个简单的手段可以说明最小化过程中的核心思想。

4.2 最小化策略

在前一节的例子中，我们运用了一种直观的方法来确定卡诺图中的那些标记为1的单元应该怎样划分在不同的组里，从而得到给定函数的最低成本实现。我们直观的策略是找到个数尽可能少并且规模尽可能大的由相邻的1构成的组，这些组应该包含函数取值为1的所有情形。每一个由相邻的1构成的组所包括的单元都能够用一个单独的乘积项表示。1的组规模越大，对应乘积项中的变量数目就越少。这个方法之所以有效，是由于我们所举的例子中卡诺图都不是很大。对于更庞大的逻辑函数，会有很多的变量，这种方法就不再适用了。取而代之的是，我们要用一种系统化的办法来获得最低成本实现。本节将介绍一种可行的办法，它与CAD工具中自动使用的技术类似。为了说明这些思想，我们仍要用到卡诺图。后面在4.8节我们会介绍一种不同的描述逻辑函数的方式，也正是CAD工具中所使用的方式。

4.2.1 专业术语

人们在逻辑函数的综合技术方面已经做了大量的研究工作。根据这些研究成果已经发表了无数的论文。为了便于表述这些成果，出现了一些专业术语以避免冗长的叙述。由于在描述最小化过程时会用到这些术语，因此我们在下面给出了一些定义。

因子

给定的乘积项包含多个变量，每个变量可能会以原变量或者反变量的形式出现。变量（不论原变量还是反变量）的每一次出现都称为一个因子（literal）。例如，乘积项$x_1\overline{x}_2x_3$有3个因子，而$\overline{x}_1x_3\overline{x}_4x_6$就有4个因子。

蕴涵项

若某个乘积项能说明输入变量的取值组合可以使给定函数（输出）为1，则称该乘积项为该函数的蕴涵项（implicant）。最基本的蕴涵项就是我们曾在2.6.1节中提及的最小项。对于一个n变量函数，最小项就是包含n个因子的蕴涵项。

现在考虑图4-9所示的三变量函数，该函数有11个蕴涵项。其中包括了5个最小项：$\overline{x}_1\overline{x}_2\overline{x}_3$，$\overline{x}_1\overline{x}_2x_3$，$\overline{x}_1x_2\overline{x}_3$，$\overline{x}_1x_2x_3$和$x_1x_2x_3$，还有5个由两个可合并的最小项组成的蕴涵项，即$\overline{x}_1\overline{x}_2$（$m_0$和$m_1$），$\overline{x}_1\overline{x}_3$（$m_0$和$m_2$），$\overline{x}_1x_3$($m_1$和$m_3$)，$\overline{x}_1x_2$($m_2$和$m_3$)和$x_2x_3$($m_3$和$m_7$)。最后，还有一个只含一个因子$\overline{x}_1$的蕴涵项，它覆盖了由4个最小项构成的组。

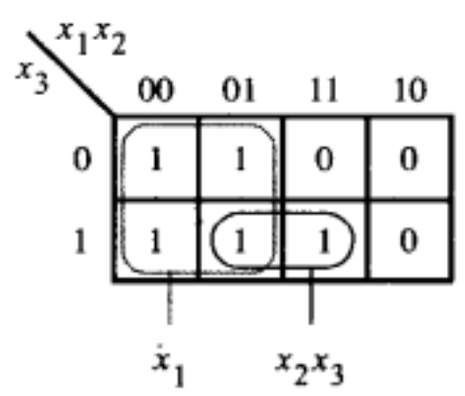


图4-9 三变量函数 $f(x_1, x_2, x_3) = \Sigma m(0, 1, 2, 3, 7)$

质蕴涵项

若某个蕴涵项不能再进一步合并为另外一个因子数更少的蕴涵项，则称该蕴涵项为质蕴涵项（prime implicant）。另一种表述该定义的方式是：质蕴涵项中的任意一个因子都是不能删除的，否则它将不再是有效的蕴涵项。

图4-9所示的逻辑函数有两个质蕴涵项：$\overline{x}_1$和x_2x_3。注意，我们不能从这两个质蕴涵项中的任意一个中删去某个因子。倘若对$\overline{x}_1$这样做会使这项不复存在；而倘若对x_2x_3删去一个因子后，只剩下x_2或x_3。x_2不是一个蕴涵项，原因是：x_2中含有当 $(x_1, x_2, x_3) = 110$时，$f = 0$的情况；同样，x_3不是一个蕴涵项的原因是：x_3中含有当 $(x_1, x_2, x_3) = 101$时，$f = 0$的情况。因此x_2或x_3都将不再是蕴涵项。

覆盖

若蕴涵项的集合能说明给定函数等于1的所有取值情况，则称该蕴涵项集合为该给定函数的覆盖（cover）。对于大多数函数而言，存在很多个不同的覆盖。很明显，能使 $f = 1$的所有最小项的集合就是一个覆盖。显然，所有质蕴涵项的集合也是一个覆盖。

一个覆盖定义了一个函数的特定实现。在图4-9所示的函数中，根据最小项组成的覆盖可以得到如下表达式：

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3$$

另一个有效的覆盖由以下表达式表示：

$$f = \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_2x_3$$

而由质蕴涵项组成的覆盖则是：

$$f = \overline{x}_1 + x_2x_3$$

上面的表达式都能正确地描述函数 f，但是由质蕴涵项所组成的覆盖才是该函数的最低成

本实现。

成本

在第2章中我们曾提出逻辑电路成本的估价定义：统计逻辑电路中门的总数以及所有门的输入端的总数，把这两个数相加即得到电路成本。本书中，我们都将使用这种成本的定义。但是我们要预先假定：原始输入（即输入变量）的原变量形式和反变量形式都存在，即输入信号的求反成本为零。在这种定义下，表达式：

$$f = x_1\overline{x}_2 + x_3\overline{x}_4$$

的成本为9，因为上式可以用2个与门和1个或门以及连在门上的6个输入端来实现。

如果电路中需要用到求反，则对应的非门以及非门输入端都应该计入成本。例如，实现下面的表达式：

$$g = \overline{x_1\overline{x}_2 + x_3}(\overline{x}_4 + x_5)$$

需要用2个与门、2个或门和1个非门，非门用来求（$x_1\overline{x}_2 + x_3$）的反。这些门的输入端总数为9，因而总成本为14。

4.2.2 最小化步骤

我们已经看到可以用多种电路来实现某给定逻辑函数。这些电路的结构和成本各不相同。在进行逻辑电路的设计时，通常必须满足某些准则。其中一个准则可能是电路的成本，这点我们在前面的讨论中已经考虑过了。一般情况下，电路规模越大，成本因素也就变得越重要。本节中，假设主要目标是得到成本最低的电路。

虽然我们曾提及电路的成本是最基本的考虑因素，但是也应当注意到，在某些场合，可能会有另外一些更适用的优化准则。举个例子，在第3章中我们曾介绍几种不同类型的预定义结构的可编程逻辑器件（PLD），这些器件编程后可以实现各种不同的电路。对于这类器件，我们主要的目标是设计某特定的电路加载到该目标器件中实现。我们不必太在意这个电路的成本是否最低，只要该电路可以在器件上顺利地运行就可以了。旨在支持某特定器件设计的CAD工具会自动地进行适合该类型器件的优化。在4.6节中我们将看到，电路优化的方式会由于器件的类型不同而不同。

在4.2.1节中，我们得出结论：当给定函数的覆盖只由质蕴涵项组成时，才能实现成本最低的电路。接下来的问题是怎样在覆盖函数的质蕴涵项集合中确定实现成本最低的子集。某些质蕴涵项必须包括在这个覆盖中，而另外一些则是可选项。若某个质蕴涵项含有其他质蕴涵项中没有的令 $f = 1$的最小项，则该质蕴涵项必须被包含在覆盖之中，且被称为本质蕴涵项(essential prime implicant)。在图4-9的例子中，两个质蕴涵项都是必要的。x_2x_3项是唯一覆盖了最小项m_7的质蕴涵项，并且$\overline{x}_1$是唯一覆盖了最小项m_0、m_1和m_2的质蕴涵项。注意，最小项m_3同时被这两个质蕴涵项所包含。则函数的最低成本实现是：

$$f = \overline{x}_1 + x_2x_3$$

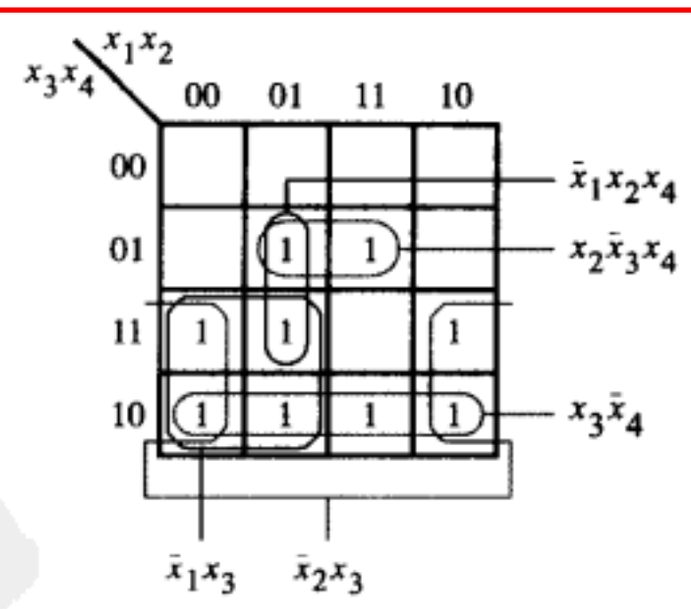


图4-10 四变量函数$f(x_1, \cdots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11\ 13, 14)$

现在我们举几个例子来说明如何为最终（成本最低的）覆盖选择质蕴涵项。思考一下如图4-10所示的四变量函数，它有5个质蕴涵项：$\overline{x}_1x_3$，$\overline{x}_2x_3$，$x_3\overline{x}_4$，$\overline{x}_1x_2x_4$和$x_2\overline{x}_3x_4$。其中本质蕴涵项为：$\overline{x}_2x_3$（因为m_{11}）、$x_3\overline{x}_4$（因为

m_{14}）和$x_2\overline{x}_3x_4$（因为m_{13}），它们必须包含在覆盖中。这三个质蕴涵项覆盖了除m_7外所有的使$f=1$的最小项。很明显m_7可以被$\overline{x}_1x_3$或$\overline{x}_1x_2x_4$覆盖。由于$\overline{x}_1x_3$有更低的成本，所以它被选入覆盖中。因此，最低成本实现为：

$$f=\overline{x}_2x_3+x_3\overline{x}_4+x_2\overline{x}_3x_4+\overline{x}_1x_3$$

根据前面的讨论，我们可以得出求解给定函数最低成本实现的步骤如下：

1）生成给定函数的所有质蕴涵项。

2）找出本质蕴涵项的集合。

3）如果本质蕴涵项的集合覆盖了所有$f=1$的取值情况，则这个集合就是函数f理想的最小成本覆盖。否则，确定应添加的非本质蕴涵项，以实现完整的最小成本覆盖。

从非本质蕴涵项中选取哪些项包含在覆盖中的主要考虑因素是成本。这些可选项通常不太明显。实际上，对于规模较大的函数，确实存在着很多种可选方案，而且必须用一些启发尝试性的方法（例如，只在可能的某个子集中选择，找到一个在大多数情况下令人满意的结果，而并非最佳）。启发尝试性方法的一种是：任意选择一个非本质蕴涵项并把它包含在覆盖中，再确定出覆盖的其他部分。接着，确定另外一个不含有刚才随意选择的质蕴涵项的覆盖。将两种覆盖结果的成本进行比较，成本较低的覆盖作为实现的选择。

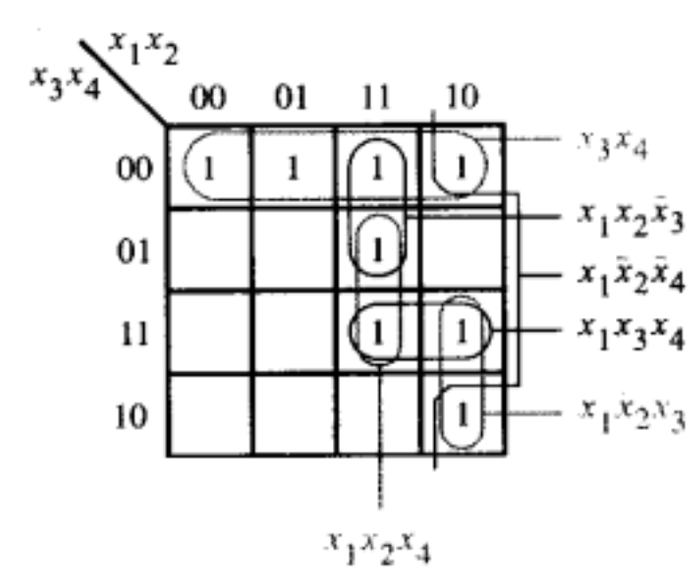


图4-11 函数$f(x_1,\cdots x_4)=\sum m(0,4,8,10,11,12,13,15)$

我们可以利用图4-11的函数来说明这个过程。在该函数的六个质蕴涵项之中，只有$\overline{x}_3\overline{x}_4$是必要的本质蕴涵项。接下来随机选取质蕴涵项$x_1x_2\overline{x}_3$，并且将其包含在第一个覆盖之中。于是留下待覆盖的三个最小项，m_{10}、m_{11}和m_{15}，还需要再加入两个质蕴涵项来覆盖它们。因此第一种实现方案是：

$$f=\overline{x}_3\overline{x}_4+x_1x_2\overline{x}_3+x_1x_3x_4+x_1\overline{x}_2x_3$$

第二种方案是$x_1x_2\overline{x}_3$没有被覆盖包含。则$x_1x_2x_4$成为必要的，这是因为没有其他方式来覆盖m_{13}。因为$x_1x_2x_4$还覆盖了m_{15}，就只剩下m_{10}和m_{11}还没被覆盖，它们可以由$x_1\overline{x}_2x_3$所覆盖。因此，第二种实现方案是：

$$f=\overline{x}_3\overline{x}_4+x_1x_2x_4+x_1\overline{x}_2x_3$$

很明显，后面这种实现是更佳的选择。

某些场合下，可能不存在本质蕴涵项。图4-12所示的就是一个这样的例子，从中任意选出某些质蕴涵项，先把其包含在覆盖内，然后再从覆盖中排除出去，两种选择的实现成本相同。用黑色圈标记的方案，其表达式为：

$$f=\overline{x}_1\overline{x}_3\overline{x}_4+x_2\overline{x}_3x_4+x_1x_3x_4+\overline{x}_2x_3\overline{x}_4$$

用灰色圈标记的方案，其表达式为：

$$f=\overline{x}_1\overline{x}_2\overline{x}_4+\overline{x}_1x_2\overline{x}_3+x_1x_2x_4+x_1\overline{x}_2x_3$$

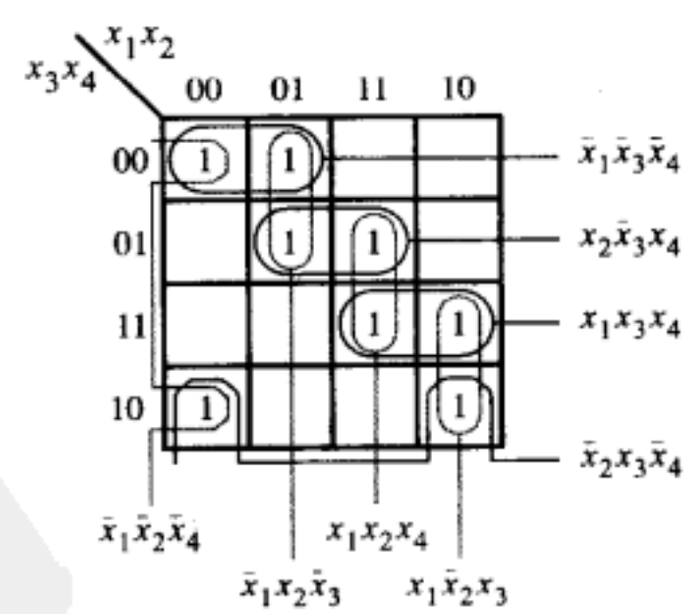


图4-12 函数$f(x_1,\cdots,x_4)=\sum m(0,2,4,5,10,11,13,15)$

无论小规模或者大规模逻辑函数，这一系列步骤对于找到成本最低的实现都是适用的。在我们所举的例子

中，用卡诺图来确定函数的质蕴涵项，然后选取最终的覆盖是很方便的。还有一些技术，虽然其基本原理相同，但更适合于CAD工具，我们将在4.9节和4.10节中介绍。

前面的例子都是用布尔代数积之和的形式来表达的，下面我们将用和之积的形式来说明同样的概念。

4.3 和之积形式的化简

到目前为止，我们已经知道怎样从某给定函数导出其成本最低的积之和（SOP）形式，应用同样的技术和对偶性原理，我们就可以得到函数的成本最低的和之积（POS）实现。在求和之积简化形式时，我们的目标是使最大项（使 $f=0$的并且必须合并入和项的）被尽可能更大的和项合并掉。同样，若一个和项覆盖的最大项越多，则认为该和项越大；若一个和项越大，则实现该和项的成本就越低。

图4-13描述了与图4-9所示相同的函数，其中有三个必须被覆盖的最大项：M_4、M_5和M_6。它们可以用两个和项来覆盖，见图4-13，由此得到该函数的和之积的实现形式：

$$f=(\bar{x}_1+x_2)(\bar{x}_1+x_3)$$

与该表达式相对应的电路有两个或门和一个与门，每个门有两个输入端，该电路的实现成本要比由图4-9推导得到的积之和形式的电路要高，因为尽管两个电路的逻辑功能是等价的，但积之和的实现只需要一个或门和一个与门。

将图4-10所示的函数重新画出，把使 $f=0$的最大项标记在图上，找到覆盖，见图4-14所示，于是得到如下逻辑表达式：

$$f=(x_2+x_3)(x_3+x_4)(\bar{x}_1+\bar{x}_2+\bar{x}_3+\bar{x}_4)$$

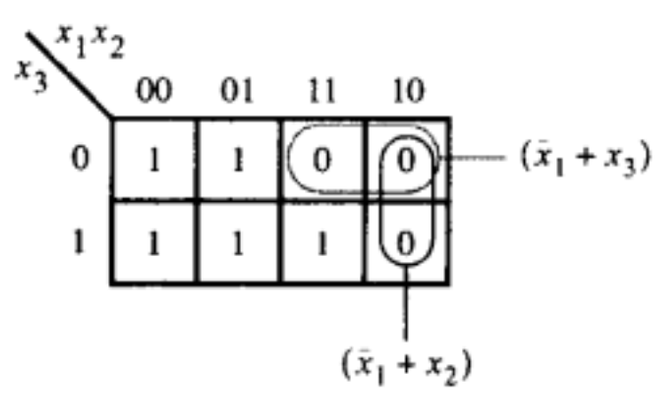


图4-13 函数 $f(x_1,x_2,x_3)=\prod M(4,5,6)$的和之积化简

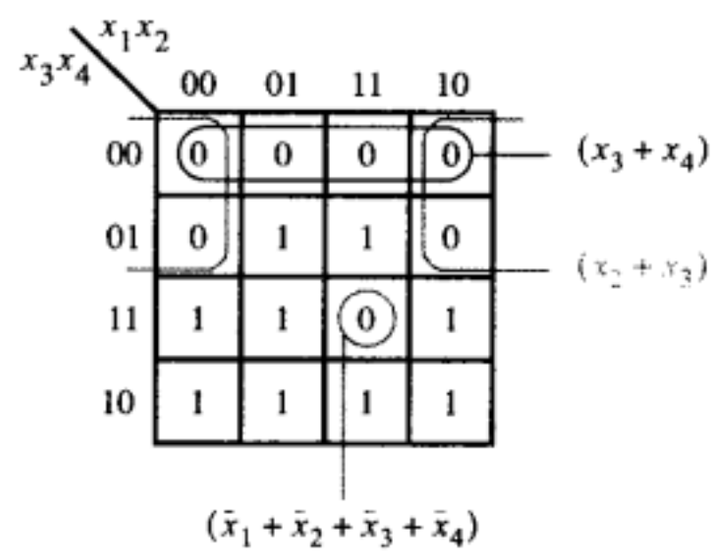


图4-14 函数 $f(x_1,\cdots,x_4)=\prod M(0,1,4,8,9,12,15)$的和之积化简

上面的表达式描述了一个由三个或门和一个与门组成的电路。三个或门中两个或门各有2个输入，第三个或门有4个输入；而与门有3个输入。假定x_1到x_4的原变量和反变量形式都存在，无需添加成本，则该电路的总成本是15。而由图4-10所导得的积之和形式的实现需要5个门和13个输入，总成本为18。相比之下，本实现电路成本较低，因而更为可取。

一般情况下，正如我们从2.6.1节中已经知道的那样，给定函数的积之和形式与和之积形式的实现电路的成本可能相同也可能不同。建议读者在求出图4-11和图4-12中函数的积之和形式之后，再求出它们的和之积形式，对两者实现电路的成本进行比较。

我们已经展示了如何得到函数成本最低的和之积形式的实现方法，即用尽可能少且尽可能最大的和项覆盖所有使 $f=0$的最大项。得到相同解的另一种方式是：求出 f 反形式的成本最低的积之和表达。然后应用德摩根定律，根据 $f=\bar{\bar{f}}$ 就可以得到 f 的和之积的最简形式。例如，如图4-13所示 $\bar{f}$的最简积之和表达式为：

$$\overline{f} = x_1\overline{x}_2 + x_1\overline{x}_3$$

应用德摩根定律对上式求反，得到：

$$\begin{aligned} f &= \overline{\overline{f}} \\ &= \overline{x_1\overline{x}_2 + x_1\overline{x}_3} \\ &= \overline{x_1\overline{x}_2} \cdot \overline{x_1\overline{x}_3} \\ &= (\overline{x}_1 + x_2)(\overline{x}_1 + x_3) \end{aligned}$$

这个结果和前一方法所得结果相同。

把同样的方法应用于图4-14所示的函数，得到：

$$\overline{f} = \overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4$$

对上式求反，得到：

$$\begin{aligned} f = \overline{\overline{f}} &= \overline{\overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\overline{x}_2\overline{x}_3} \cdot \overline{\overline{x}_3\overline{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4) \end{aligned}$$

得到的和之积形式的表达式与以前的一致。

4.4 非完全指定函数

在数字系统中，某些输入条件往往永远也不会出现。例如，假设x_1和x_2控制两个不能同时接通的互锁开关。那么这两个开关只可能出现三种状态：1）两开关均断开；2）甲断开乙闭合；3）乙断开甲闭合。换言之，输入的取值组合只有 $(x_1, x_2) = 00$、01或10，而决不会有11出现。由于输入取值为 11 的情况根本不会出现，所以设计时可以不必考虑该输入取值，我们称 $(x_1, x_2) = 11$ 为无关条件。若函数含有无关条件，则称该函数为非完全指定函数。

无关条件简称无关项，在设计逻辑电路时得加以利用，从而带来一些好处。既然这些输入取值永远不会出现，设计人员就可以将这些取值条件下的函数值任意地假设为1或0，设1还是0要看哪种情况更利于求解函数的最低成本实现。上述思想可以用图4-15的例子说明，该函数中标记为1的最小项为m_2、m_4、m_5，m_6和m_{10}。假设是上面提到的开关互锁的情况，输入x_1和x_2永远不会同时等于1；因而最小项m_{12}、m_{13}、m_{14}和m_{15}都可以设置为无关项。这些无关项在图中由字母 d 标记。运用速记符号将函数 f 表达为：

$$f(x_1, \cdots, x_4) = \sum m\,(2, 4, 5, 6, 10) + D\,(12, 13, 14, 15)$$

式中D表示无关项的集合。

图4-15 函数 $f(x_1, \cdots, x_4) = \sum m\,(2, 4, 5, 6, 10) + D\,(12, 13, 14, 15)$ 的两种实现形式

图4-15a表示该函数的最佳积之和形式的实现。为了构成尽可能大的由相邻的1构成的组，从而得到成本最低的质蕴涵项，需要令无关项D_{12}、D_{13}和D_{14}（对应于最小项m_{12}、m_{13}和m_{14}）

的值为1，而令D_{15}的值为0。这样只需要用两个质蕴涵项就能够提供 f 的完整覆盖。实现结果为：

$$f = x_2\bar{x}_3 + x_3\bar{x}_4$$

图4-15b展示了怎样得到最佳的和之积形式。无关项的值仍做同样的假定。实现结果为：

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)$$

无关项的取值可以由设计者随意指定的这一特性，使得函数的实现得到了极大的简化。若我们只是单纯地将无关项从函数的综合中排除出去，假定它们的值永远是0，则积之和形式的表达式就会是：

$$f = \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_3\bar{x}_4 + \bar{x}_2x_3\bar{x}_4$$

而和之积形式的表达式将会是：

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2)$$

这两个表达式的实现成本都高于对无关项进行适当赋值后化简得到的表达式。

尽管可以随意设定无关项的值，但是随意设定的值不能保证给定函数得到成本最低的实现。若有k个无关项，则可以有2^k种方案来给这些项赋值为1或0。在卡诺图中，通过观察，我们就能知道怎样最佳地进行赋值以便找到函数的最简实现。

在上面的例子中，为了实现积之和以及和之积的简化，将无关项D_{12}，D_{13}和D_{14}设置为1，而将无关项D_{15}设置为0。由此推导得到的表达式表示的是同一个逻辑函数，该逻辑函数仍旧可以用$\sum m$ (2, 4, 5, 6, 10, 12, 13, 14) 来表示。为实现积之和以及和之积的简化，将无关项设置成相同的值并不总是一个好主意。有时为了实现积之和的简化，将某个无关项设置为1有好处；而为了实现和之积的简化，把该无关项设置为0有好处（或者反过来分别设置成0和1）。在这一类场合，最佳的积之和以及和之积表达式所表示的已经不再是原来的函数了，而是包括了这些无关项的逻辑函数。但这些函数只是在对应这些无关项取值时才有所不同。4.14节的例4.24用图解释了这个问题。

本节中使用互锁开关的例子来引入无关项的概念，这似乎有些人为的痕迹。然而，在本书的第6、8和9章里有很多例子表明，在实际数字电路的设计过程中，确实会碰到无关项的情形。

4.5 多输出电路

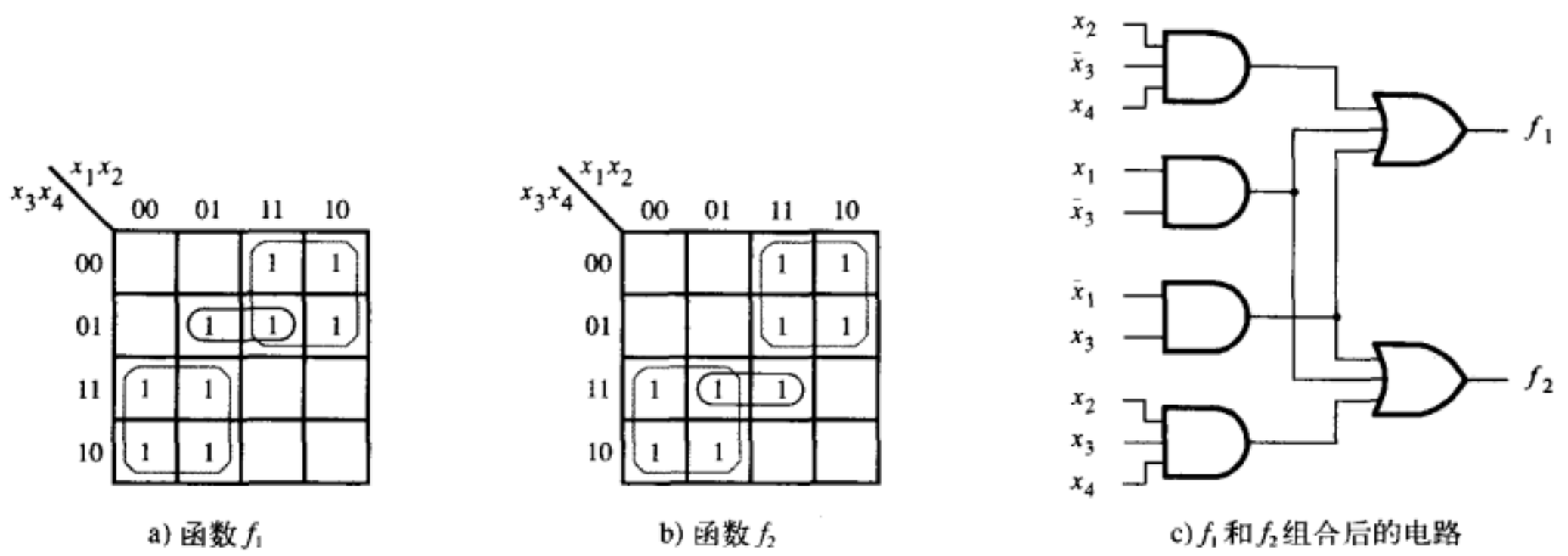
前面所有例子中我们只考虑了单个函数及其实现。在实际的数字系统中，常常遇到许多个逻辑函数在一起的情况，要求我们把这些个逻辑函数作为大规模电路的一部分来实现。若我们不是分别简化每个函数的实现，而是作为整体来考虑，则有可能共享某些在各自函数的实现中需要用到的门，从而显著降低整体电路的总成本。

例4.1 图4-16所示的是共享门的例子。图中有两个输入变量相同的函数 f_1 和 f_2。这两个函数的成本最低的实现分别由图4-16a和图4-16b表示，简化后的表达式如下：

$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$

$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

其中 f_1 的成本是4个门和10个输入端，总成本为14。f_2 与 f_1 的成本相同。若两个函数用各自的电路来实现，则总成本为28。



a) 函数 f_1　　b) 函数 f_2　　c) f_1 和 f_2 组合后的电路

图4-16　多输出综合的例子

若将两个电路合并成一个双输出端的电路，就可能得到成本更低的实现。由于两表达式的头两个乘积项是一致的，因此实现同一个乘积项只需要一个与门，无需使用双份。合并后的电路见图4-16c，其实现成本为6个门和16个输入端，总成本为22。■

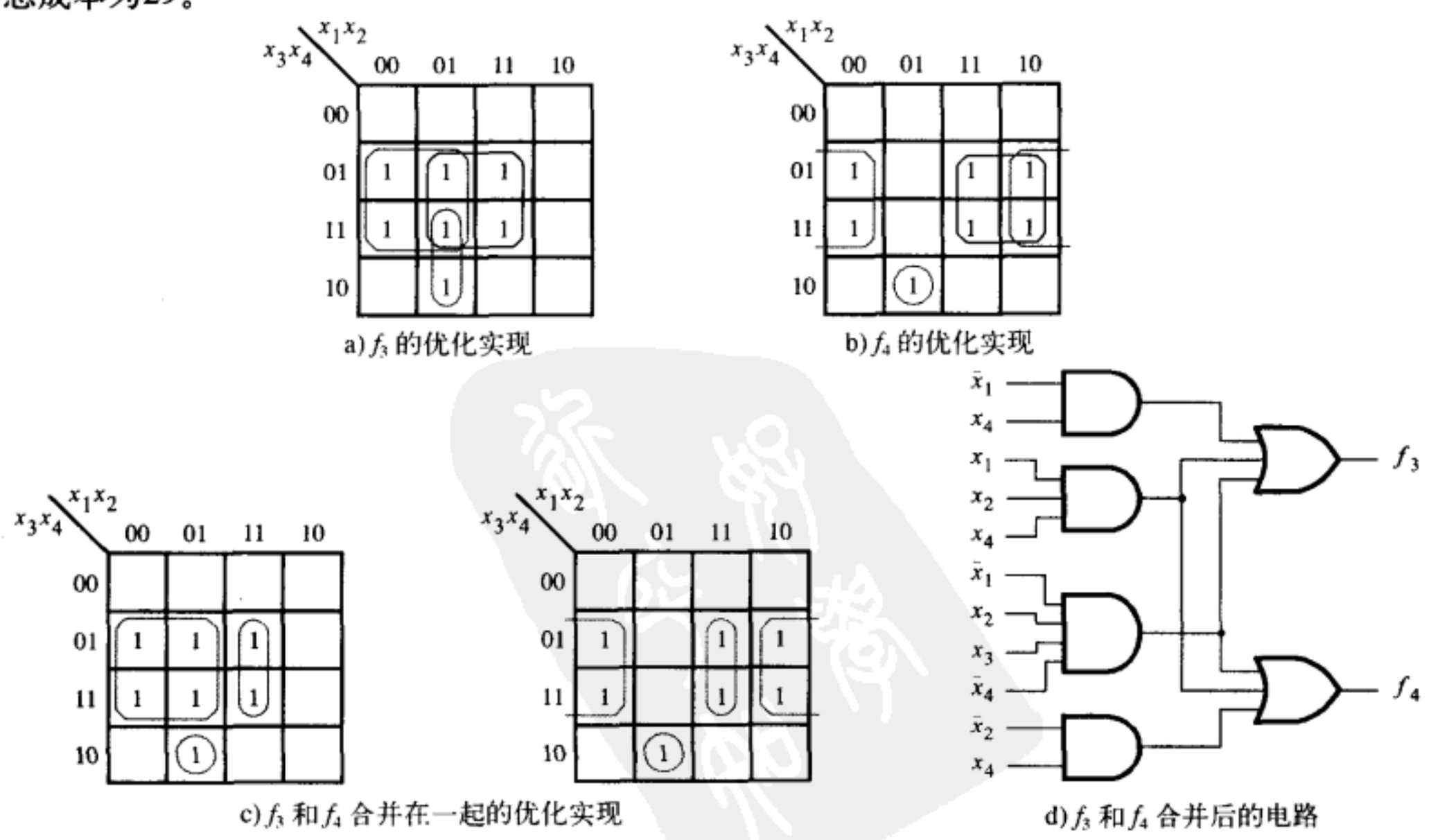
此例中我们分别求出 f_1 和 f_2 的最低成本实现，然后共享实现相同乘积项的门，从而减少了整体电路的总成本。这种策略并不能保证在任何情况下都得到成本最低的电路，见下面的例子。

例4.2　图4-17所示的是想用一个电路实现的两个函数。分别由图4-17a和图4-17b得到函数 f_3 和 f_4 最小成本的表达式如下：

$$f_3 = \bar{x}_1x_4 + x_2x_4 + \bar{x}_1x_2x_3$$

$$f_4 = x_1x_4 + \bar{x}_2x_4 + \bar{x}_1x_2x_3\bar{x}_4$$

这里没有可以共享的与门，因此合并后的电路成本为6个与门、2个或门和21个输入端，总成本为29。



a) f_3 的优化实现　　b) f_4 的优化实现

c) f_3 和 f_4 合并在一起的优化实现　　d) f_3 和 f_4 合并后的电路

图4-17　多输出电路综合的另一个例子

虽然没有可以共享的门，但是还可能有其他可供选择的方案。在推导 f_3 和 f_4 的表达式时，除了只限用质蕴涵项外，还可以寻找其他一些蕴涵项，这些蕴涵项有可能在合并后的电路中实现共享。图4-17c所示的是蕴涵项的最佳选取方案，其对应的逻辑表达式为：

$$f_3 = x_1x_2x_4 + \bar{x}_1x_2x_3\bar{x}_4 + \bar{x}_1x_4$$
$$f_4 = x_1x_2x_4 + \bar{x}_1x_2x_3\bar{x}_4 + \bar{x}_2x_4$$

这两个式子中头两个蕴涵项是相同的。由此推导出的电路，见图4-17d所示。共有6个门和17个输入端，总成本为23。■

例4.3 在例4.1中我们找到了函数 f_1 和 f_2 的最佳的积之和形式的实现，结果如图4-16所示。现在我们考虑该函数的和之积形式的实现。函数 f_1 和 f_2 的成本最低的和之积表达式为：

$$f_1 = (\bar{x}_1 + \bar{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_3 + x_4)$$

上述表达式中没有共同的和项，因此在合并的电路中也没有可以共享的或门。而且，观察图4-16的卡诺图，显然找不到有利于同时实现 f_1 和 f_2 的和项(即覆盖 $f_1 = f_2 = 0$的单元的那些项)。那么按照之前的叙述，最好的选择就是将每个函数各自单独实现。实现每个函数需要3个或门、1个与门和11个输入，因此实现两个函数的总成本为30。这个电路的成本高于例4.1得到的积之和形式的实现成本。■

例4.4 现在考虑图4-17所示函数 f_3 和 f_4 的和之积形式的实现。其成本最低的和之积表达式如下：

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(\bar{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + \bar{x}_4)$$

上述表达式中前三个和项是一样的，因此可以在合并后的电路中共享，这些项需要3个或门和6个输入。另外，函数 f_3 还要一个2输入或门和一个4输入与门，函数 f_4 还要一个3输入或门和一个4输入与门。这样合并后的电路总共包含5个或门、2个与门和19个输入，总成本为26。该成本略微高于例4.2所得电路之成本。■

这些例子表明，给定函数用最佳积之和形式或者最佳和之积形式来实现，其复杂程度可能会很不同。对于图4-16和图4-17中的函数，积之和形式会得出更好的实现结果。但是若我们关心的是实现这些图中四个函数的反函数形式，则和之积的形式也许成本会更低一些。

用于进行逻辑综合的功能全面的CAD工具，将会自动完成上述例子中各种类型的优化过程。

4.6 多级综合

在以前的章节中，我们的目标是求出给定函数的成本最低的积之和或和之积形式的实现。这种类型的逻辑电路包含两个级别的门。在积之和形式中，第一级电路为与门，与门的输出端连接到第二级电路的或门的输入端上。在和之积形式中，由第一级的或门将信号传给第二级的与门。我们已经假定这些输入变量的逻辑真和非的形式都存在，因而无需用非门来求这些变量的反。

两级电路的实现对于变量较少的函数通常是高效的。然而，随着输入端的数量增多，两级电路可能会出现扇入问题，而是否会出现这种问题取决于实现这种电路所采用的技术。例如，我们看下面这个函数：

$$f(x_1, \cdots, x_7) = x_1x_3\bar{x}_6 + x_1x_4x_5\bar{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

这是个成本最低的积之和形式的表达式。现在考虑在两种可编程逻辑器件（PLD）中实现 f：一种是复杂可编程逻辑器件（CPLD），另一种是现场可编程门阵列（FPGA）。图4-18所示的是在CPLD中的实现，其基本单元是图3-33所示的类似于可编程阵列逻辑（PAL）的模块。图中灰色图形表示用于实现函数 f 的电路。很明显，该函数的积之和形式适合于此CPLD的芯片结构。

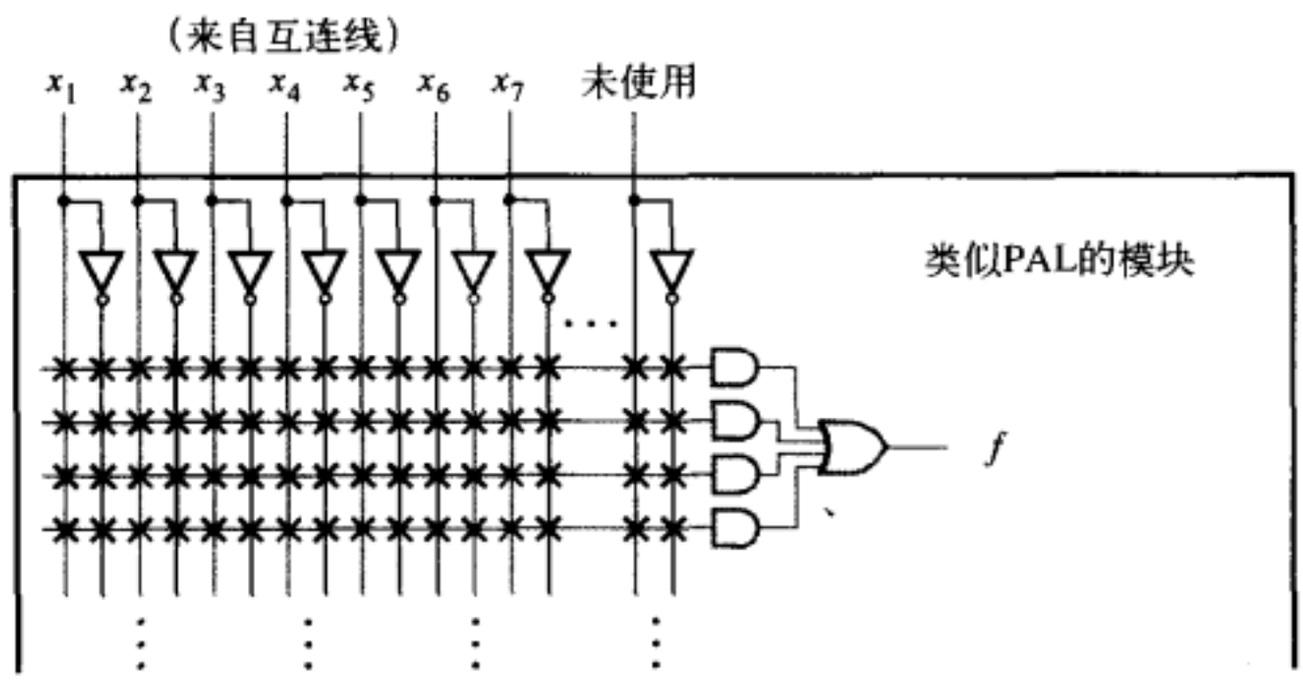


图4-18 在CPLD中的实现

接下来，考虑在FPGA中实现函数 f。本例中，我们所用的FPGA如图3-39所示，其中包含有两输入的查找表（LUT）。由于 f 的积之和表达式需要3输入和4输入的与门以及一个4输入或门的运算符，在这里实现函数所需的扇入值对于目标芯片的结构来说太庞大了，因此不能用这片FPGA中的基本单元直接实现函数 f。

为了解决扇入的问题，f 必须表达成为多于两级的逻辑表达式，这样的形式称为多级逻辑表达式。多级电路的综合有几种不同的途径。我们将要讨论两种重要的技术：提取公因子法（factoring）和功能分解法（functional decomposition）。

4.6.1 提取公因子

用2.5节中介绍的分配律，我们可以对前面的表达式 f 提取公因式，所得的结果如下：

$$f = x_1\bar{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5)$$
$$= (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

相应电路的最大扇入为2，因而可以用两输入的查找表来实现。图4-19所示的是一种可行的实现方案，即用图3-39中的FPGA来实现。请注意必须由查找表实现的二变量函数，在图中用方块代表（每个方块对应一个查找表）。

扇入问题

前面的例子说明，扇入约束是由FPGA的固有结构造成的，因为此FPGA中每个查找表只允许有两个输入。然而，即使目标芯片的结构不是固定的，也仍有可能存在扇入的问题。为用实际例子说明这种情况，请考虑一片全定制芯片电路的实现。以前我们曾经提过，全定制芯片上通常包含有大量的门，假定芯片是用CMOS工艺制造的，仍然会存在扇入约束的问题（见3.8.8节中的讨论）。在CMOS工艺条件下，每个逻辑门输入端的数量通常较少。例如，与门输入端的个数一般限制在5个之内。在这种限制条件下，如果某逻辑表达式包括一个7输入的乘积项，我们就不得不使用两个4输入与门来实现它（如图4-20所示）。

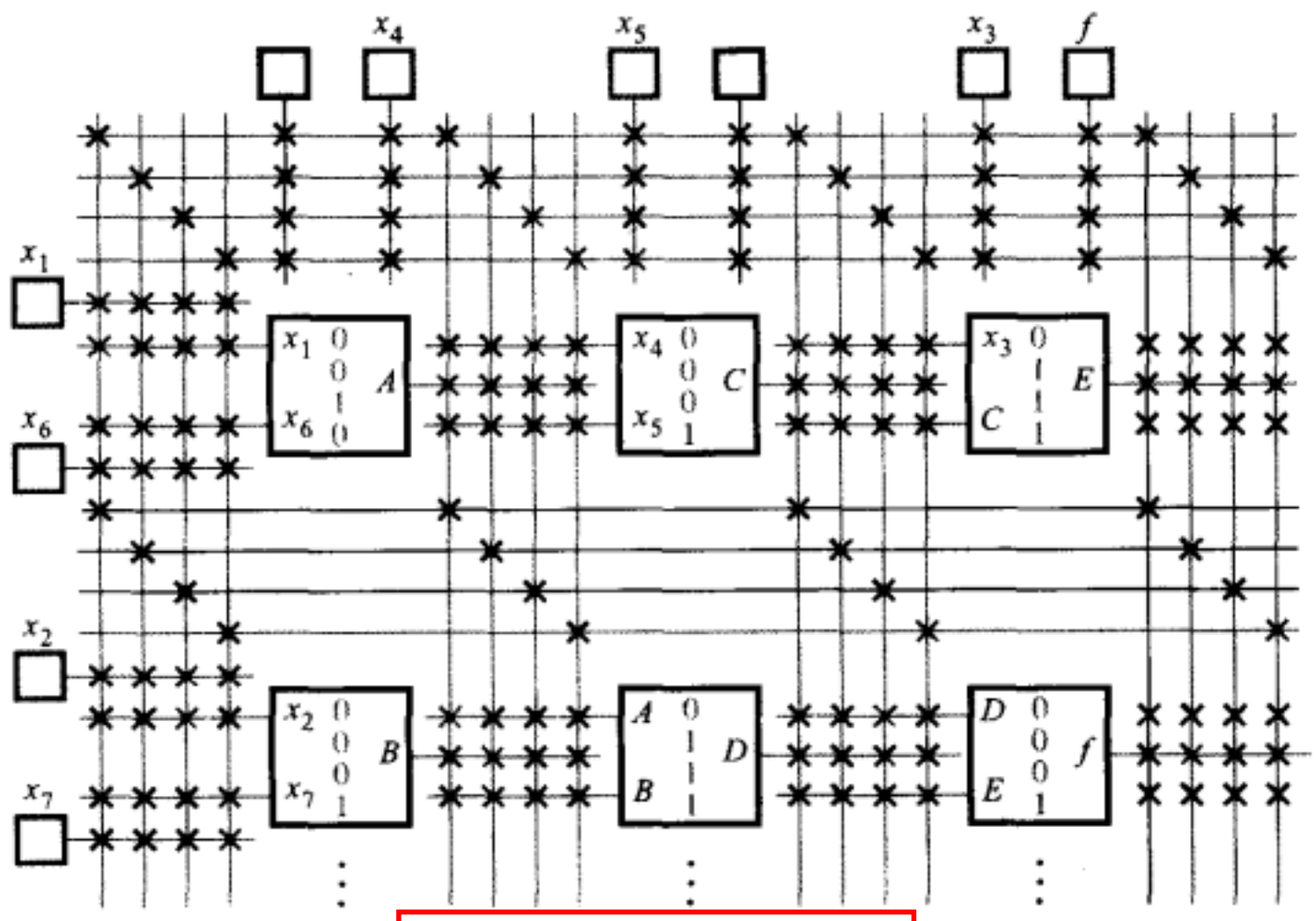


图4-19 在FPGA中的实现

提取公因子法可以用来解决扇入问题。我们假定可用门的最大扇入是4，而想要实现的函数为：

$$f = x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1x_2\bar{x}_3\bar{x}_4\bar{x}_5x_6$$

这是一个最简的积之和形式的表达式。使用图4-20所示的方法，我们需要4个与门和1个或门来实现该逻辑表达式。更好的解决方案是将表达式提取公因子后，得到如下的逻辑表达式：

$$f = x_1\bar{x}_4x_6(\bar{x}_2x_3x_5 + x_2\bar{x}_3\bar{x}_5)$$

这样只需用3个与门和1个或门就足以实现上面的函数，如图4-21所示。

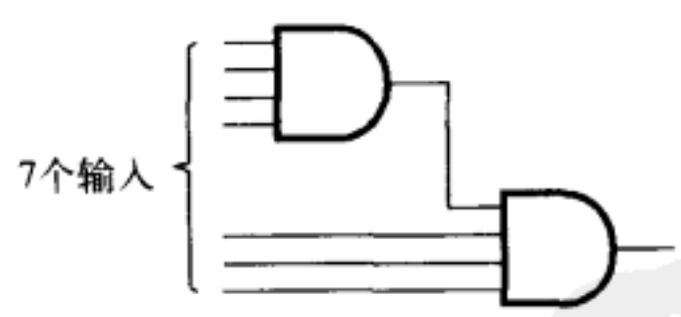


图4-20 用4输入与门实现7输入乘积项

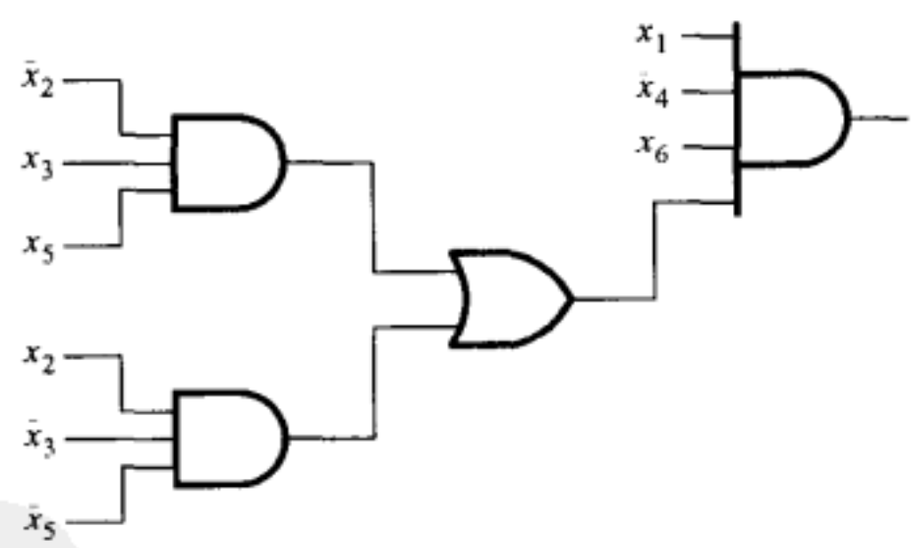


图4-21 一个提取公因子后的电路

例4.5 在实际的情况中，逻辑电路设计人员常常遇到一些条件说明，这些说明会很自然地使逻辑函数在初始设计中以提取公因子的形式出现。假设我们需要设计一个符合下列条件的电路：该电路共有四个输入x_1、x_2、x_3和x_4，一个输出f_1。输出$f_1 = 1$的条件是：1）若x_1和x_2中至少有一个为1，并且x_3和x_4都为1；2）若$x_1 = x_2 = 0$并且x_3或x_4有一个为1。其他所有情况下，$f_1 = 0$。另一个输出f_2，若$x_1 = x_2 = 0$或$x_3 = x_4 = 0$，则$f_2 = 0$，除此之外的所有情况下，$f_2 = 1$。

根据这些条件，函数f_1可以表示为：

$$f_1 = (x_1 + x_2)x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

上面的表达式可以简化为：

$$f_1 = x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

读者可以用卡诺图验证此化简结果。

第二个函数 f_2，以其非的形式更容易来说明，即

$$\bar{f}_2 = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

运用德摩根定律可得

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

这是 f_2 的最低成本形式的表达式；若用积之和的形式，则电路成本将会增加。

因为我们的设计目标是实现函数 f_1 和 f_2 的成本最低的组合电路，从以上分析来看，似乎对两函数都采用提取公因子的形式效果最好。这样函数 f_1 和 f_2 可以共享和项（即或项）$(x_3 + x_4)$；此外，由于 $\bar{x}_1\bar{x}_2 = \overline{x_1 + x_2}$，只要将 f_1 改写成如下形式：

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

则或项$(x_1 + x_2)$同样可以被函数 f_1 和 f_2 共享。因此得到合并在一起的电路，见图4-22所示，共包含3个或门、3个与门、1个非门和13个输入，总成本为20。 ■

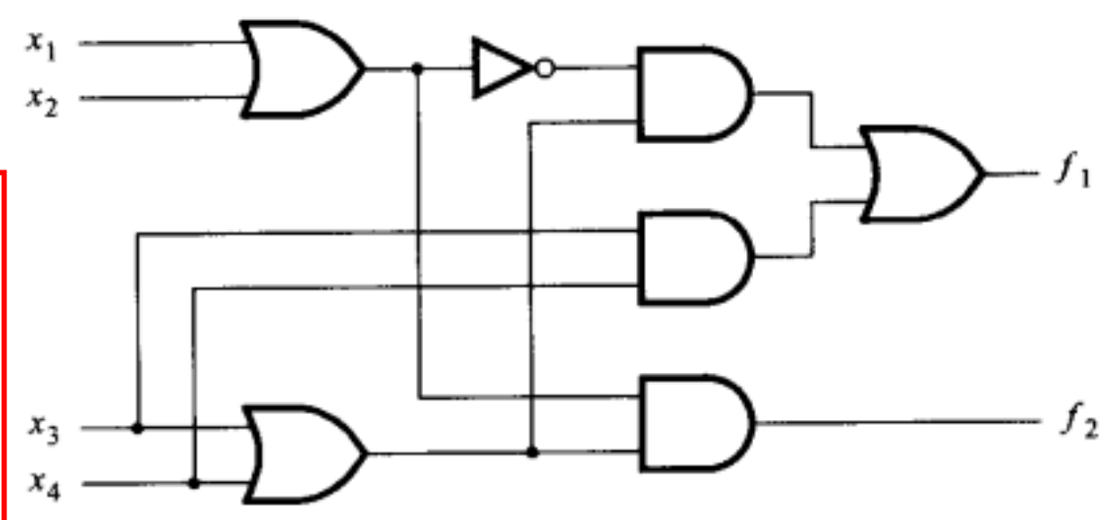


图4-22 例4.5的电路

对布线复杂度的影响

集成电路芯片的面积由两部分组成：1）实现逻辑门的电路；2）各个逻辑门之间的连接线。其中连线所用面积占总面积的很大一部分。因此应该尽量设法降低连线的复杂度。

逻辑表达式中每个因子对应于电路中的一根信号线。提取公因子后一般都能够减少因子的数量，所以提取公因子是减少逻辑电路连线复杂度的有效机制。在综合过程中，CAD工具需要考虑到很多不同的因素，其中包括电路的成本、扇入个数以及连线复杂度等。

4.6.2 函数分解

在前面介绍的提取公因子方法的例子中，使用了多级电路来处理扇入限制的问题。然而，即便不存在扇入问题，设计者也可能更乐意采用多级电路。多级电路比其等价的两级电路更受欢迎的原因是：在某些情况中，多级电路可能降低电路的实现成本。另一方面，多级电路由于使用了多级的逻辑门，因此需要更长的传播延迟。下面我们将借助例子来讨论这些问题。

逻辑电路在连线和逻辑门方面的复杂度，通常能够通过将两级电路分解为子电路而得以降低，在构建最终电路时，一个或多个子电路实现的函数可能会被多次用到。为了达到这个目的，把一个两级逻辑表达式分解成两个或更多的新表达式，然后将这些新表达式合并在一起，构成一个等价的多级电路。我们可以通过下面简单的例子来说明这个思想。

例4.6 考虑如下成本最低的积之和形式的表达式：

$$f = \bar{x}_1x_2x_3 + x_1\bar{x}_2x_3 + x_1x_2x_4 + \bar{x}_1\bar{x}_2x_4$$

并假定输入x_1和x_4只有原变量形式可以用（不提供反变量形式）。则表达式定义了这样一个电路：有4个与门、1个或门、2个非门和18条到所有门的输入连线。与门的扇入是3， 或门的扇入是4。读者应该注意到，在这个例子中我们引入了求x_1和x_2的反所用的非门的成本，而不是

像以前那样，假定所有输入变量的逻辑真和反的形式都是现成的。

在表达式的头两个乘积项中提取公因子x_3，在后两个乘积项中提取x_4后，该表达式改写为：

$$f=(\bar{x}_1x_2+x_1\bar{x}_2)x_3+(x_1x_2+\bar{x}_1\bar{x}_2)x_4$$

现在令 $g(x_1,\ x_2)=\bar{x}_1x_2+x_1\bar{x}_2$，可以看到：

$$\begin{aligned}\bar{g}&=\overline{\bar{x}_1x_2+x_1\bar{x}_2}\\&=\overline{\bar{x}_1x_2}\cdot\overline{x_1\bar{x}_2}\\&=(x_1+\bar{x}_2)(\bar{x}_1+x_2)\\&=x_1\bar{x}_1+x_1x_2+\bar{x}_2\bar{x}_1+\bar{x}_2x_2\\&=0+x_1x_2+\bar{x}_1\bar{x}_2+0\\&=x_1x_2+\bar{x}_1\bar{x}_2\end{aligned}$$

则函数f的表达式可记为：

$$f=gx_3+\bar{g}x_4$$

根据上面的表达式可推导出如图4-23所示的电路。该电路需要添加1个或门和1个使g的值取反的非门，但是该电路只需要15个输入端。而且，最大的扇入也已降至 2。该电路的成本低于等价的两级电路的成本。由于电路逻辑层次增加了3级，因此为降低成本该电路付出的代价是传播延迟的增加。

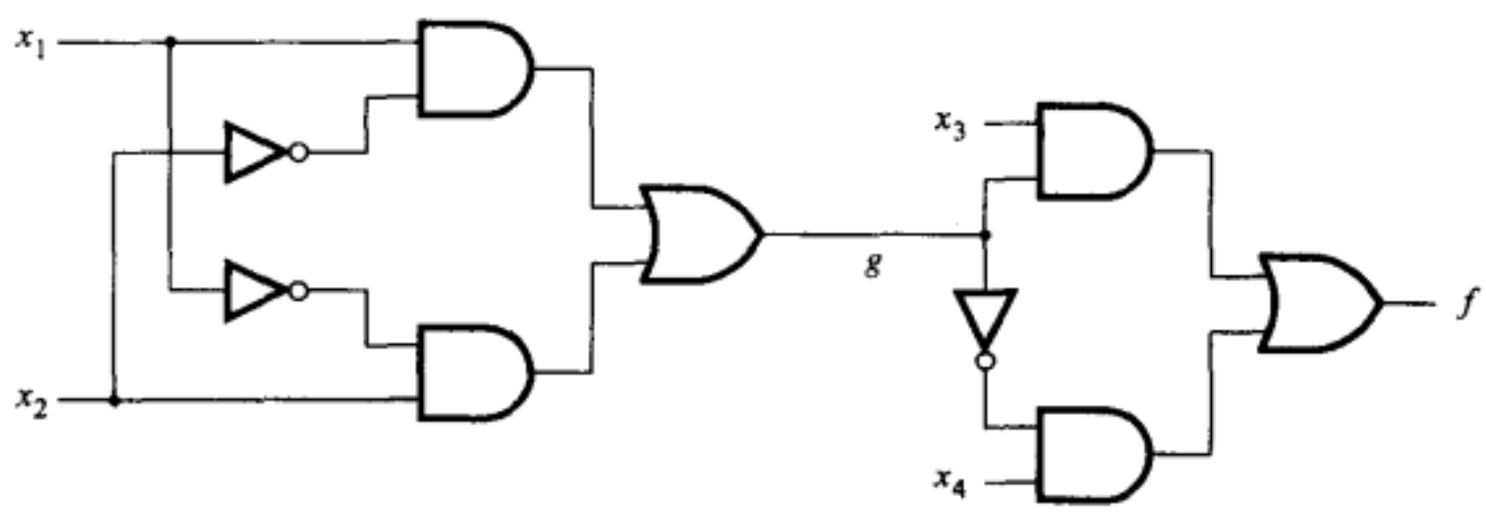


图4-23 例4.6的逻辑电路

在本例中，子函数g是变量x_1和x_2的函数。这个子函数对应于函数 f 所实现的整个电路的一部分，被用作电路其余部分的输入。把电路其余部分对应的函数记作h，该函数只依赖于三个输入：g、x_3和x_4。则函数f的分解实现可以用代数方法表达为：

$$f(x_1,x_2,x_3,x_4)=h\ [g\ (x_1,x_2),x_3,x_4]$$

这种函数分解的结构可以用图4-24所示的方块图的形式表示。 ■

函数分解确实可以使电路的成本和复杂度大幅度地降低。若上面这个例子效果还不明显，则从下面的这个例子，读者可以充分看到这样做的好处。

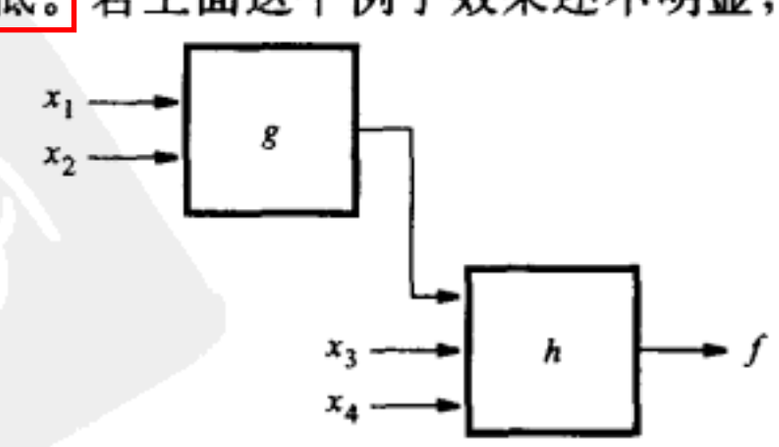


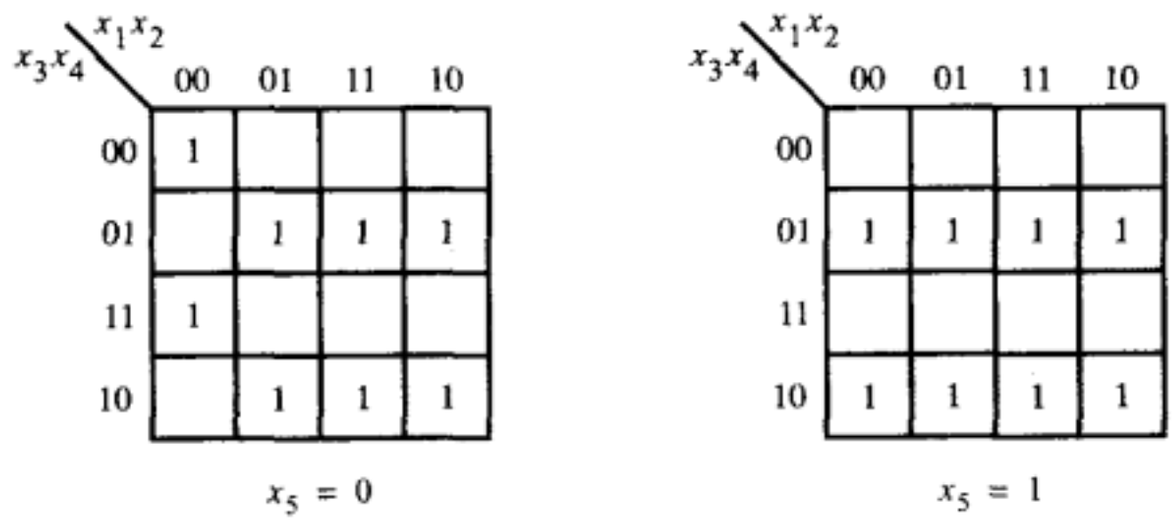
图4-24 例4.6的分解结构

例4.7 图4-25a以卡诺图的形式定义了一个五变量的函数 f。为了找到该函数的一种好的分解结果，我们首先要识别出将用作子函数输入的变量。我们可以从卡诺图标记为1的分布图案得到有用的线索。请注意：以行为单位来观察卡诺图，各行中只有两种明显不同的图

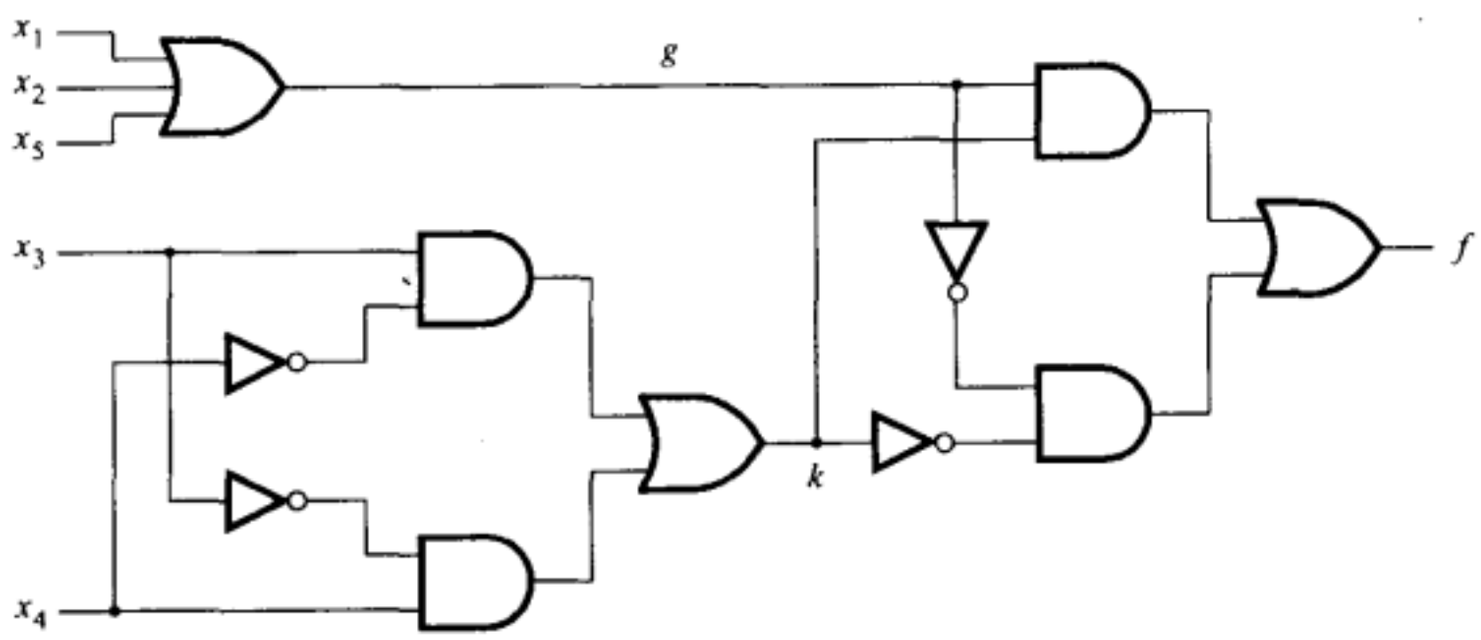
案。第2和第4行的图案相同（以灰色背景标记），而第1和第3行（译者注：原文此处为2）是另一种图案。一旦我们需要详细指出每一行中的图案，那么该行的图案就只取决于该行中定义的列变量，即x_1、x_2和x_5。令子函数$g(x_1, x_2, x_5)$代表第2、4行的图案，则该子函数的表达式为：

$$g = x_1 + x_2 + x_5$$

因为这些变量中的任何一个为1时，都会使该图案的值等于1。为详细说明g的图案在各行中出现的位置，我们用变量x_3和x_4确定行的位置。$\overline{x}_3x_4$和$x_3\overline{x}_4$项各自确定第2和第4行。则表达式$(\overline{x}_3x_4 + x_3\overline{x}_4) \cdot g$描述了$f$中由第2、4行定义的部分。



a) 函数f的卡诺图



b) 利用分解后得到的电路

图4-25 例4.7的函数分解

下一步，我们要找到第1行和第3行中的图案。该图案只在$x_1 = x_2 = x_5 = 0$的单元中取值为1，对应的项是$\overline{x}_1\overline{x}_2\overline{x}_5$。但是我们有一个十分有用的发现，即这一项恰好是$g$的非，$\overline{g} = \overline{x}_1\overline{x}_2\overline{x}_5$。第1行和第3行分别由$\overline{x}_3\overline{x}_4$和$x_3x_4$项确定，则表达式$(\overline{x}_3\overline{x}_4 + x_3x_4)\cdot\overline{g}$表示了$f$的第1行和第3行。

我们又有另一个有用的发现，表达式$(\overline{x}_3x_4 + x_3\overline{x}_4)$和$(\overline{x}_3\overline{x}_4 + x_3x_4)$恰好互为反，如例4.6所示。因此，如果令$k(x_3, x_4) = \overline{x}_3x_4 + x_3\overline{x}_4$，那么函数$f$的完全分解可以记为：

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h\,[\,g\,(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \overline{k}\overline{g} \end{aligned}$$

其中

$$g = x_1 + x_2 + x_5$$
$$k = \overline{x}_3x_4 + x_3\overline{x}_4$$

与上述表达式对应的电路见图4-25b所示，它总共需要11个门和19个输入，最大扇入为3。

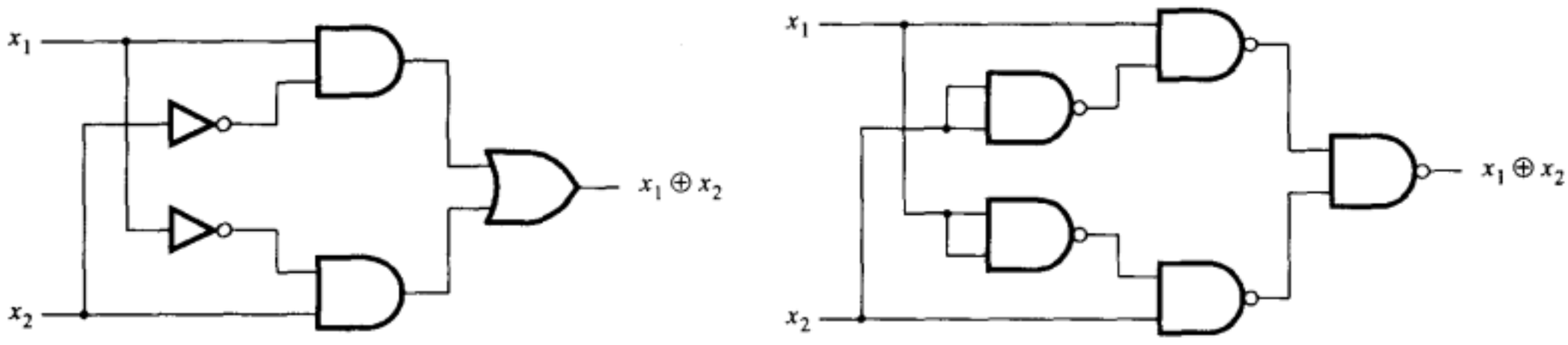
作为比较，列出函数f的成本最低的积之和形式的表达式为：

$$f = x_1\overline{x}_3x_4 + x_1x_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_2x_3\overline{x}_4 + \overline{x}_3x_4x_5 + x_3\overline{x}_4x_5 + \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2x_3x_4\overline{x}_5$$

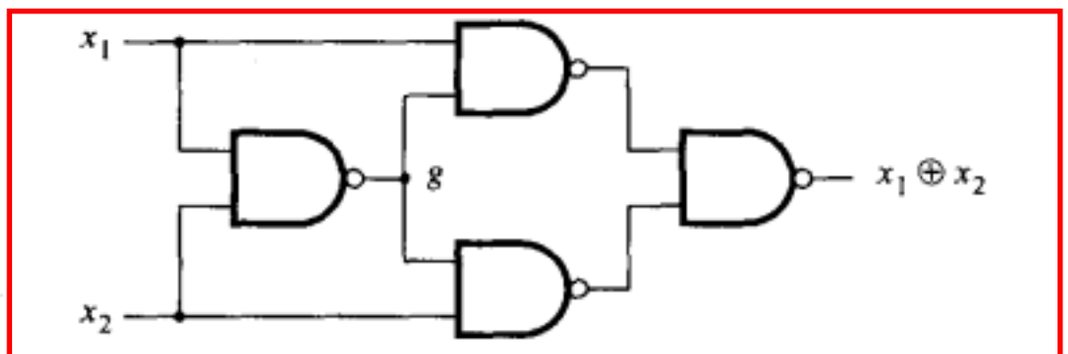
对应的电路总共需要14个门（包括用来生成原始输入的反的5个非门）和41个输入。用作输出的或门，其最大扇入为8。显然，函数分解使该函数的实现得到了极大的简化。 ■

在前两个例子中，函数分解采用的是这样一种方法，分解后得到的子函数依赖于某些原始输入变量，而剩余部分依赖于其他一些变量。这类分解在专业文献中称为无交集分解。当然也有可能采用有交集分解的方法，即子函数中的变量也可以用于剩余部分电路的实现。下面的例子说明了这种情况。

例4.8 异或（XOR）是一种很有用的函数。在3.9.1节中我们说明了这种运算可以用一种特殊的电路来实现。异或函数也可以用与门和或门来实现，见图4-26a所示。在2.7节中我们曾阐述如何将任意的与-或电路转换为相同结构的与非-与非电路。



a) 积之和实现　　b) 与非门实现



c) 最佳的与非门实现

图4-26 异或的实现

现在让我们采用函数分解的方法来找出只用与非门实现异或函数的更好方案。令符号↑表示与非操作符，则 $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$，异或函数的积之和表达式为：

$$x_1 \oplus x_2 = x_1\bar{x}_2 + \bar{x}_1 x_2$$

由2.7节的讨论可知，这个表达式可以用与非运算符的形式改写为：

$$x_1 \oplus x_2 = (x_1 \uparrow \bar{x}_2) \uparrow (\bar{x}_1 \uparrow x_2)$$

实现这个表达式需要5个与非门，见图4-26b所示电路。请注意，该图中用2输入与非门来实现一个反相器（即将两个输入端连接在一起）。

为函数分解，我们对 $(x_1 \uparrow \bar{x}_2)$ 项进行如下的操作：

$$(x_1 \uparrow \bar{x}_2) = (x_1\bar{x}_2) = (x_1(\bar{x}_1 + \bar{x}_2)) = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2))$$

对 $(\bar{x}_1 \uparrow x_2)$ 可以进行类似的操作，并生成：

$$x_1 \oplus x_2 = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2)) \uparrow ((\bar{x}_1 + \bar{x}_2) \uparrow x_2)$$

根据德摩根定律：$\bar{x}_1 + \bar{x}_2 = x_1 \uparrow x_2$，则可以推导出：

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

于是得到：

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$

与上述表达式对应的电路只需要4个与非门，见图4-26c所示。 ■

实际问题

函数分解是降低电路复杂度的有效技术，并且也可以用于在某些具有内部约束的电路中实现一般的逻辑函数。例如，在第3章中介绍过的可编程逻辑器件（PLD）中，需要把所需的逻辑电路装入PLD可用的特定逻辑块中。PLD中可用的特定逻辑块所能实现的逻辑函数正是函数分解的目标子函数，用这些子函数可实现规模较大的函数。

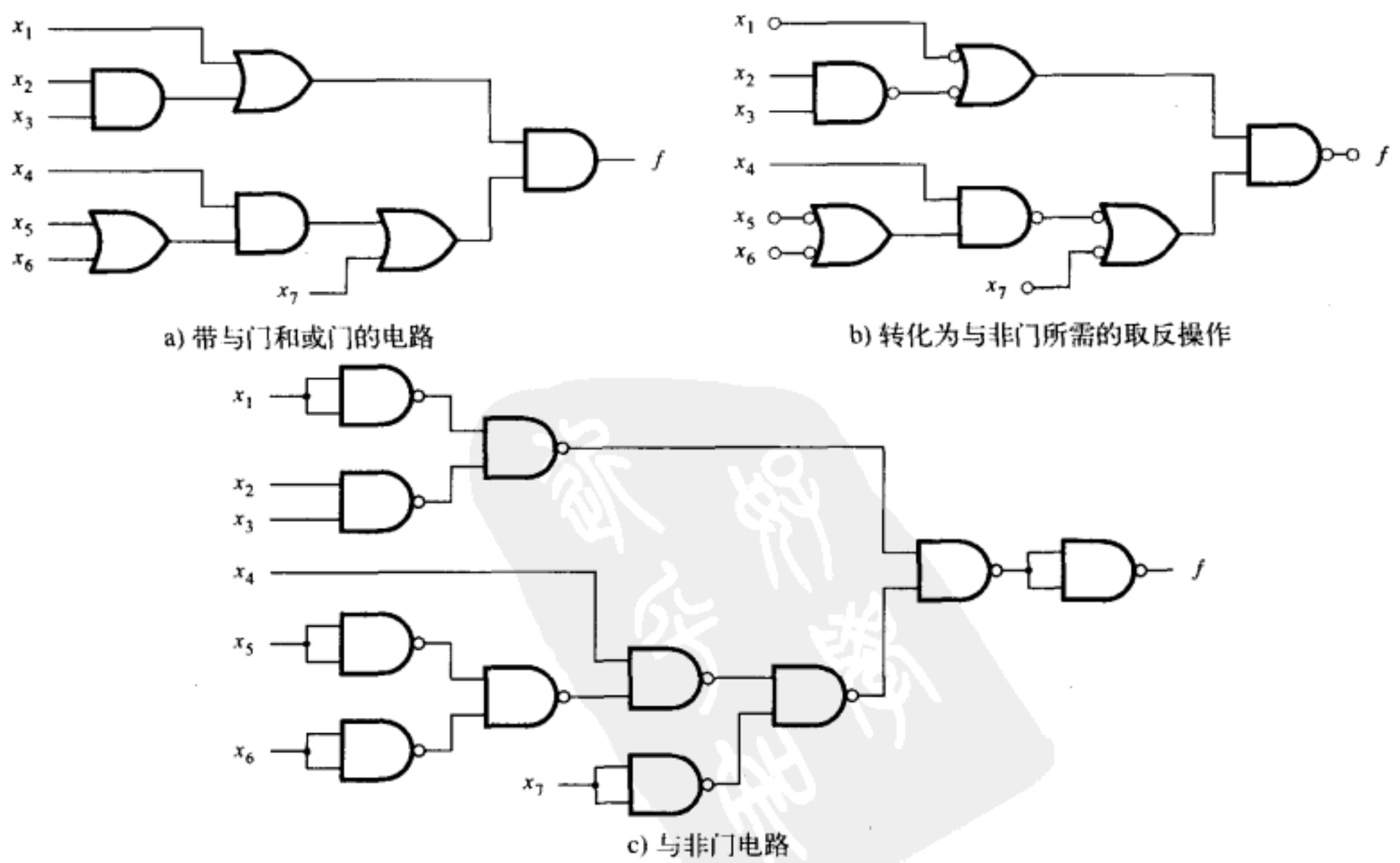
如何找到合适的子函数是函数分解中的大问题。对于多变量函数，存在大量的可能性需要尝试，这使得找到最佳解决方法非常困难。取而代之的方法是，采取试探性的启发式手段，从而求得可接受的较优解。

全面讨论函数分解和公因子提取方法已经超出了本书的范围。有兴趣的读者可以参考有关文献[2-5]。现代CAD工具广泛地使用函数分解的概念。

4.6.3 多级与非以及或非电路

在2.7节中，我们曾展示了由与门和或门组成的两级电路可以方便地转化为用同样结构的与非门和或非门实现的电路。特别是，与-或（积之和）电路可以用与非-与非电路实现，而或-与（和之积）电路也可以转化为或非-或非电路实现。同样的转化方法可以用于多级电路。我们将用例子说明此方法。

例4.9 图4-27a所示的为一个由与门和或门组成的四级电路。让我们首先推导只包含与非门的等价电路，每个与门被转化为与非门需要将其输出取反。每个或门转化为与非门要将其输入取反。上述转换方法正是德摩根定律的应用，见图2-21a所示。图4-27b中以小圆圈表示必



a) 带与门和或门的电路

b) 转化为与非门所需的取反操作

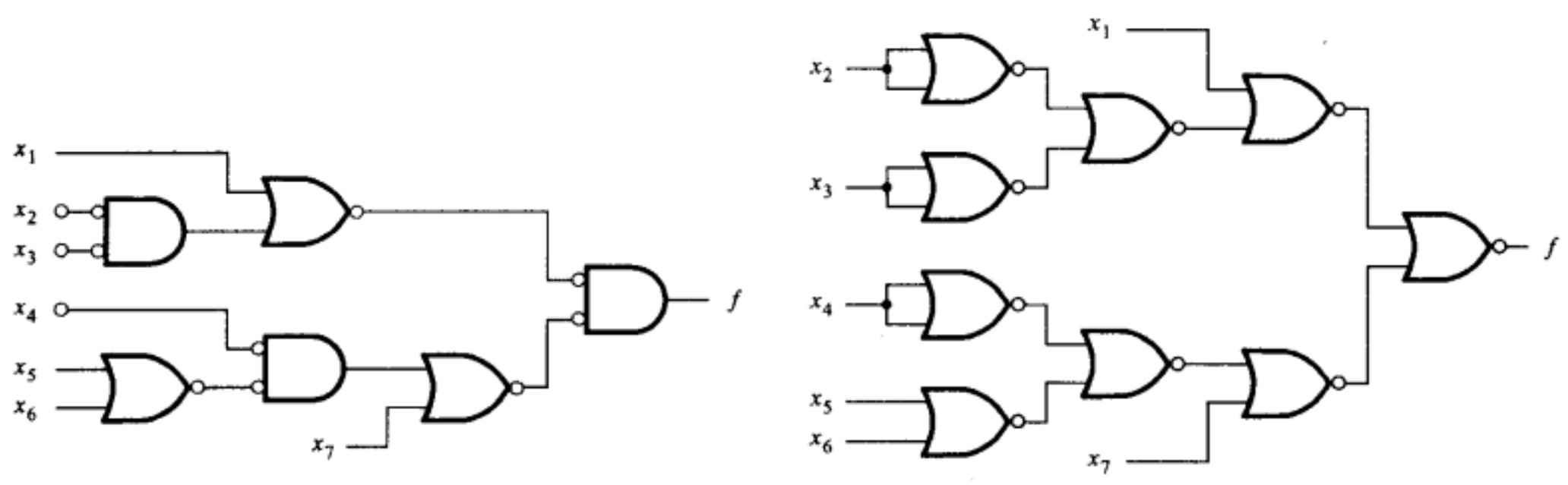
c) 与非门电路

图4-27 将给定电路转化为与非门电路

要的取反操作。请注意取反操作是在每一根线的两端同时进行的。在把每一个门转换为与非门的过程中，绝大多数取反操都被融入与非门之中，只有5个取反操作需要单独实现，这几个取反操作是对输入信号x_1、x_5、x_6和x_7的求反以及输出 f 的求反。求反操作可以由2输入的与非门实现，只需要将与非门的两个输入端连接在一起。转换后的电路见图4-27c所示。

用类似的方法可以将图4-27a所示的电路转化为仅由或非门组成的电路。或门可以转化为或非门，但需要将其输出取反。与门可以转化为或非门，但需要将其输入取反，见图2-21b所示。使用此方法后，需要增加取反操作，这些取反操作在示例电路上以小圈标出，见图4-28a所示。

于是每个门都变为了或非门。输入x_2、x_3和x_4的取反可以用两输入或非门来实现，或非门的两个输入被连接在一起。最后得到的电路见图4-28b所示。



a) 转化成或非门所需的取反操作　　b) 或非门电路

图4-28　将给定电路转化为或非门电路

很明显，电路的基本布局不会随着与门和或门转化为与非门或者或非门而发生实质性的变化。然而，若电路中的取反操作没有全部都融入其他逻辑门成为它们的一部分，则就有可能需要添加非门来实现取反操作。■

4.7　多级电路的分析

前一节中我们曾指出，采用多级电路来实现逻辑函数也许会带来好处，同时指出逻辑综合工具一般也是用这种方法将函数综合成电路的。在本节中我们将考虑如何分析已有的电路，从而确定实现该电路的逻辑函数。

对于两级电路，这个分析过程很简单。若电路具有与-或（与非-与非）结构，则通过观察即可用积之和的形式写出它的输出函数。同样，对于或-与（或非-或非）结构的两级电路，也可以通过观察，很容易地写出其和之积形式的表达式。而对多级电路来说，分析工作会比较复杂，这是由于很难通过观察直接写出函数的表达式。我们不得不跟踪整个电路并确定电路的功能才能得到该电路的表达式。跟踪可以由输入端开始向后进行直至输出端，或者由输出端开始往回进行直至输入端。在电路的中间的某一个点上，必须求出这些逻辑门所实现的子函数。

例4.10　将图4-27a所示的电路拷贝到图4-29。为了确定该电路所实现的函数 f，我们先来考察组成该电路的各个门的输出，即该电路内部各节点的信号。这些点在图中被标记为P_1到P_5。这些点实现的函数为：

$$P_1 = x_2x_3$$
$$P_2 = x_5 + x_6$$
$$P_3 = x_1 + P_1 = x_1 + x_2x_3$$
$$P_4 = x_4P_2 = x_4(x_5 + x_6)$$
$$P_5 = P_4 + x_7 = x_4(x_5 + x_6) + x_7$$

因此，f 可以表示为如下的表达式：

$$\begin{aligned} f &= P_3P_5 \\ &= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7) \end{aligned}$$

应用分配律消去圆括号得到如下表达式：

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

注意，这个表达式描述了一个由6个与门、1个或门和25个输入端组成的电路。这个两级电路的成本高于图4-29所示电路的成本。但是该电路的传播延迟较小。 ■

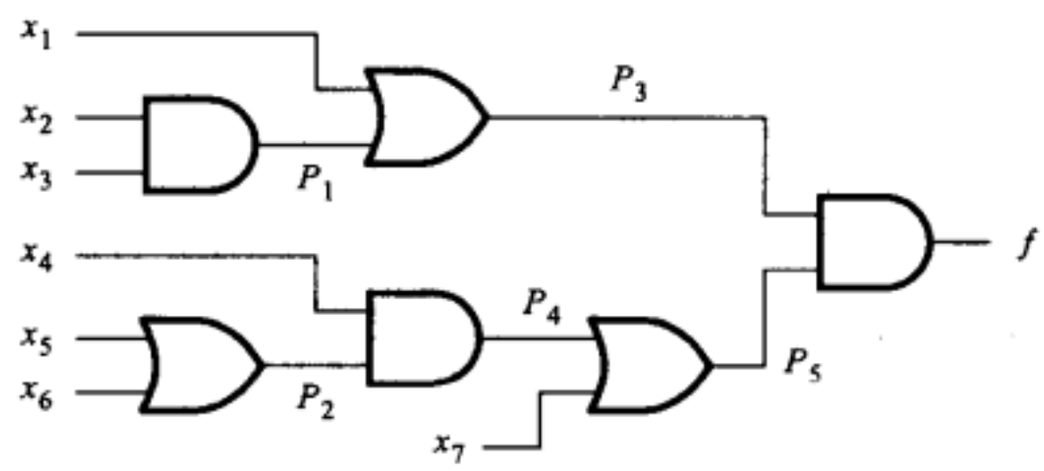


图4-29 例4.10的电路图

例4.11 由例4.7得到图4-25b所示的电路。在该电路中，除了与门和或门外，还有一些非门。我们在这里复制了该电路图，见图4-30所示，将内部节点标记为P_1到P_{10}，可得到如下子函数：

$$P_1 = x_1 + x_2 + x_5$$
$$P_2 = \overline{x}_4$$
$$P_3 = \overline{x}_3$$
$$P_4 = x_3P_2$$
$$P_5 = x_4P_3$$
$$P_6 = P_4 + P_5$$
$$P_7 = \overline{P}_1$$
$$P_8 = \overline{P}_6$$
$$P_9 = P_1P_6$$
$$P_{10} = P_7P_8$$

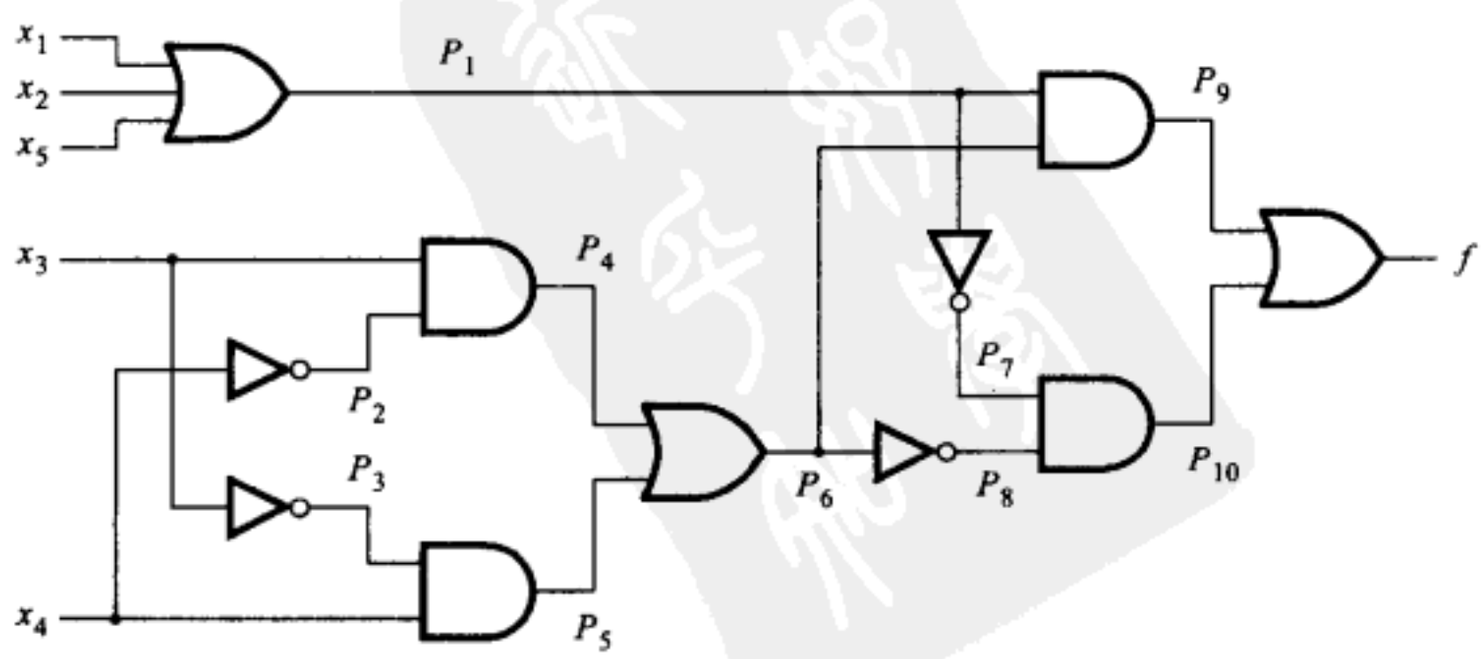


图4-30 例4.11的电路图

我们可以由输出端往回追踪至输入端，得到函数 f 的表达式如下：

$$
\begin{aligned}
f &= P_9 + P_{10} \\
&= P_1P_6 + P_7P_8 \\
&= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1\overline{P}_6 \\
&= (x_1 + x_2 + x_5)(x_3P_2 + x_4P_3) + \bar{x}_1\bar{x}_2\bar{x}_5\overline{P}_4\overline{P}_5 \\
&= (x_1 + x_2 + x_5)(x_3\bar{x}_4 + x_4\bar{x}_3) + \bar{x}_1\bar{x}_2\bar{x}_5(\bar{x}_3 + \overline{P}_2)(\bar{x}_4 + \overline{P}_3) \\
&= (x_1 + x_2 + x_5)(x_3\bar{x}_4 + \bar{x}_3x_4) + \bar{x}_1\bar{x}_2\bar{x}_5(\bar{x}_3 + x_4)(\bar{x}_4 + x_3) \\
&= x_1x_3\bar{x}_4 + x_1\bar{x}_3x_4 + x_2x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_5x_3\bar{x}_4 + x_5\bar{x}_3x_4 + \\
&\quad \bar{x}_1\bar{x}_2\bar{x}_5\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_5x_4x_3
\end{aligned}
$$

上面的表达式和由例4.7求得的表达式相同。 ■

例4.12 由与非门和或非门构成的电路分析起来较为困难，因为这两种门都包含了取反操作。图4-31a描述了由与非门组成的简单电路，下面通过此例子来说明取反的影响。我们可以使用与例4.9中所介绍的相反的方法，将该电路转化为由与门和或门组成的电路。由德摩根定律，若把如图4-31b所示的标记反相的小圆圈去掉，则原电路便可以转化为如图4-31c所示的电路，转变成完全由与门和或门构成。我们注意到在转化后的电路中，输入变量x_3和x_5取反变量的形式。根据图4-31c所示电路，便可以确定函数 f 的表达式为：

$$
\begin{aligned}
f &= (x_1x_2 + \bar{x}_3)x_4 + \bar{x}_5 \\
&= x_1x_2x_4 + \bar{x}_3x_4 + \bar{x}_5
\end{aligned}
$$

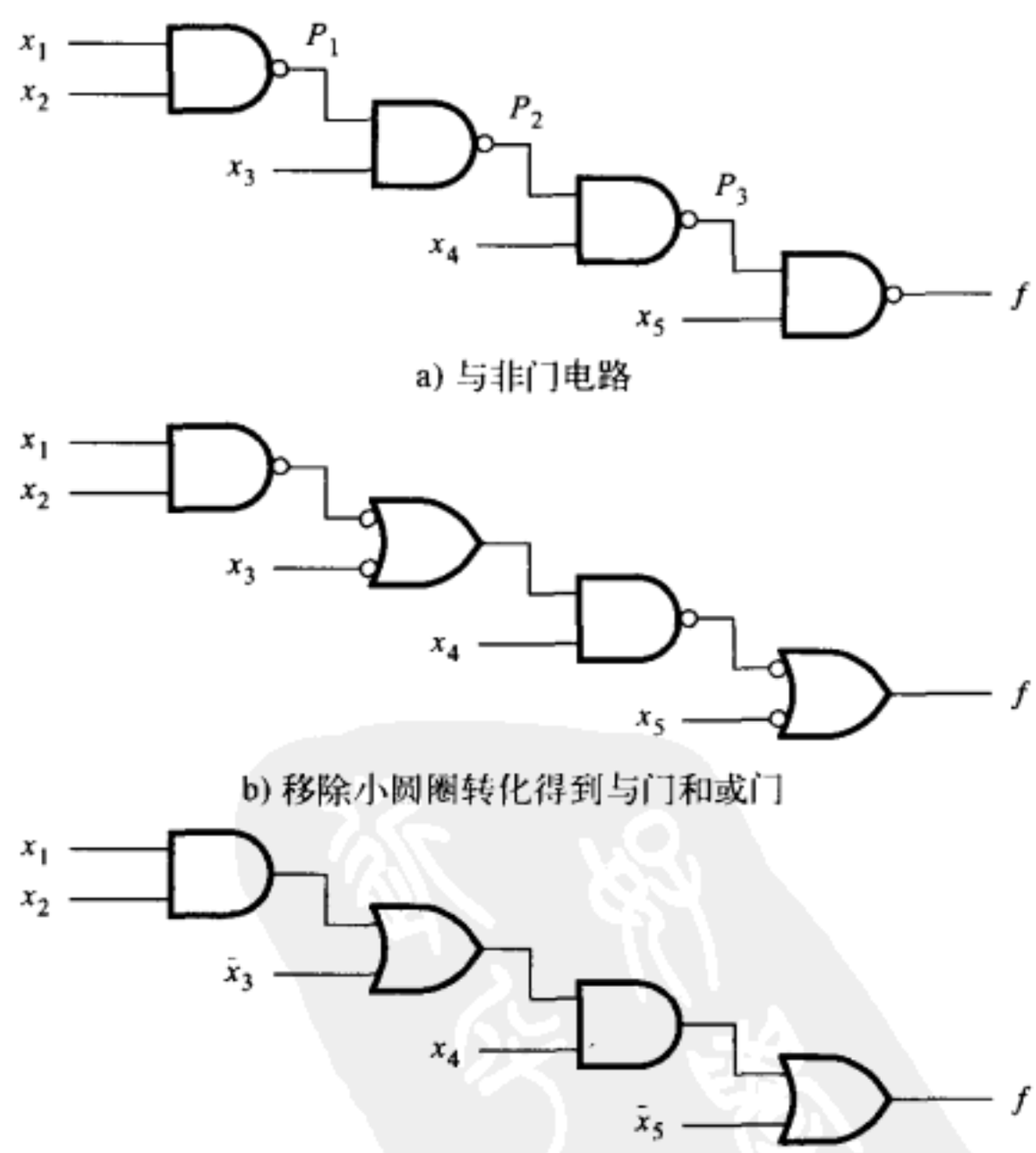


图4-31 例4.12的电路图

并非必须把由与非门组成的电路转化为由与门和或门组成的电路后，才能确定该电路的函数表达式。 我们可以使用来自于例4.10和例4.11的方法，如下所示推导出函数 f。给电路的

内部节点做上标记，见图4-31a所示，分别记为P_1、P_2和P_3。由此得到如下一系列表达式，最后化简得到函数f的表达式：

$$
\begin{aligned}
P_1 &= \overline{x_1x_2} \\
P_2 &= \overline{P_1x_3} \\
P_3 &= \overline{P_2x_4} \\
f &= \overline{P_3x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2x_4}} + \overline{x}_5 = P_2x_4 + \overline{x}_5 \\
&= \overline{P_1x_3}x_4 + \overline{x}_5 = (\overline{P_1} + \overline{x}_3)x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1x_2}} + \overline{x}_3)x_4 + \overline{x}_5 \\
&= (x_1x_2 + \overline{x}_3)x_4 + \overline{x}_5 \\
&= x_1x_2x_4 + \overline{x}_3x_4 + \overline{x}_5
\end{aligned}
$$

■

例4.13 图4-32所示的电路由与非门和或非门组成。对该电路的分析如下：

$$
\begin{aligned}
P_1 &= \overline{x_2x_3} \\
P_2 &= \overline{x_1P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4\overline{x}_5 \\
&= \overline{\overline{P_2 + P_3}} \cdot \overline{x}_5 \\
&= (P_2 + P_3)\overline{x}_5 \\
&= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2x_3 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= \overline{x}_1\overline{x}_5 + x_2\overline{x}_5 + \overline{x}_3\overline{x}_5 + \overline{x}_4\overline{x}_5
\end{aligned}
$$

请注意，上面的表达式在倒数第3行到倒数第2行的推导过程中，应用了2.5节中的性质16a，将（$x_2x_3 + \overline{x}_3$）简化为（$x_2 + \overline{x}_3$）。

电路的分析远比电路的综合简单。经过一小段时间的练习后，读者自己就能容易地解决电路的分析问题，即使相当复杂的电路分析也不成问题。 ■

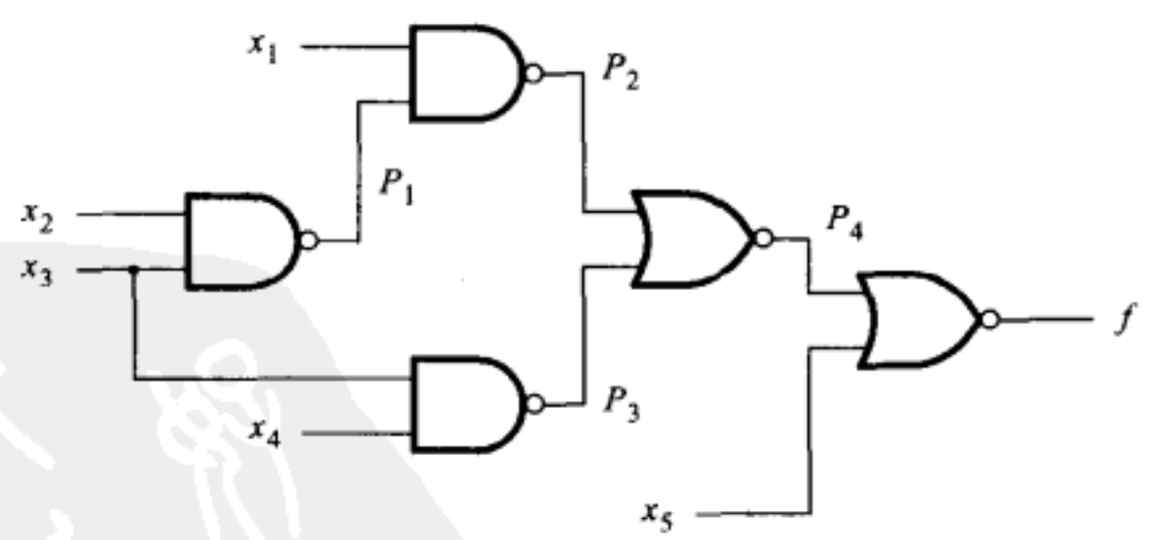


图4-32 例4.13的电路图

到现在为止，我们已经讨论了不少关于逻辑函数综合与分析的内容。我们应用卡诺图作为载体来说明寻求逻辑函数的最优实现时所涉及的概念。我们还指出逻辑函数允许有多种实现的方案，无论是用两级逻辑还是多级逻辑的方案都是可以的。在现代设计环境中，逻辑电路的综合是使用CAD工具完成的，而不是用人工的方法来做的。本章中所讨论的概念都是很一般的；这些概念代表了CAD工具进行电路综合时所用的算法以及策略。正如我们前面曾提到过的，用卡诺图表示逻辑函数的方法并不适用于

CAD工具。在下一节中，我们将讨论另一种适用于CAD工具算法的逻辑函数表示法。

4.8 立方体表示法

卡诺图是说明逻辑化简概念的非常有效的工具，若逻辑函数的变量很少，则用卡诺图进行手工逻辑设计也很方便。但当处理变量个数很多的大型函数时，卡诺图方法就不太适合，需要采用代数的方法，因为代数方法可以处理有任意多个变量的函数。

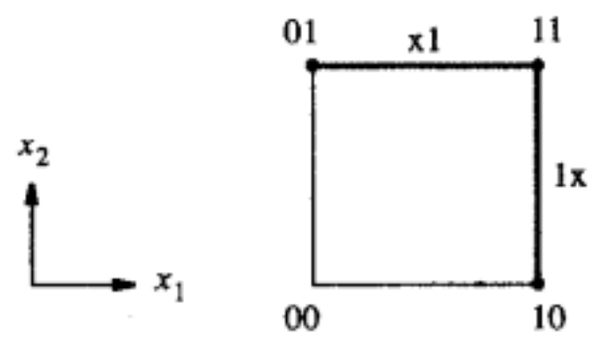
人们已经研究出很多种代数优化技术。我们不想详细地研究这些技术，但我们想让读者对代数优化技术有一个大致的了解。这样就能使我们对CAD工具能完成什么以及期望CAD工具能为我们提供什么结果有一个认识。将要介绍的这种方法利用了逻辑函数的立方体表示法。

立方体和超立方体

迄今为止，我们在本书中遇到了四种表示逻辑函数的方法：真值表，代数表示法，维恩图和卡诺图。另外一种可能的表示法是把n变量的逻辑函数映射为n维立方体。

二维立方体

图4-33所示的是一个二维立方体。立方体的四个角叫做顶点（vertex），它们对应着真值表中的四行。每一个顶点由两个坐标确定。假设横坐标对应着变量x_1，纵坐标对应变量x_2。则顶点00在立方体的左下角，它对应真值表的第0行。顶点01在左上角，此时$x_1=0$，$x_2=1$，它对应真值表中的第1行，另外两个顶点以此类推。



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

图4-33 函数$f(x_1, x_2) = \sum m(1, 2, 3)$的表示

我们在立方体上用黑色的小圆点表示令$f=1$的顶点，以此在立方体上表示一个函数。在图4-33中，令$f=1$的顶点是01、10和11。我们可以用一个顶点的集合表示这个函数，记作$f=\{01,10,11\}$。图4-33还列出了函数f的真值表。

两个相邻的顶点可以连接成一条边，给连接的边做一个标记，用x指出哪一个变量是不同的。因此，若把两个令$f=1$的顶点连接成一条边，则这条边所代表的逻辑项与两个独立顶点所代表的逻辑项是相同的。例如，顶点10和11令$f=1$。我们用标记为1x的边连接这两个顶点。我们习惯上用x表示其所对应的变量可以取0或1中任意一个值。因此1x表示$x_1=1$，而x_2可以为1或者0。同样，边x_1连接01和11两个顶点，它表明x_1可以为1或0，$x_2=1$。请读者们不要把x的这种用法与带下标的x_1和x_2表示变量的用法混淆。

两个相邻顶点可以用一条边表示这正是2.5节中性质14a'（结合律）的具体体现。边1x是顶点10和11的逻辑和。它本质上就是乘积项x_1，x_1是最小项$x_1\overline{x}_2$与x_1x_2的逻辑和。而性质14a恰好表明：

$$x_1\overline{x}_2 + x_1x_2 = x_1$$

因此，寻找$f=1$的边就等于在应用结合律。显然，这与在卡诺图中寻找令$f=1$的相邻的两个单元类似。

边1x和x1完全定义了图4-33中的逻辑函数。因此我们可以记这个函数为$f=\{1x, x1\}$，其对应的逻辑表达式为：

$$f = x_1 + x_2$$

很明显，上面的表达式也与图4-33中的真值表一致。

三维立方体

图4-34展示了一个三维立方体。在图的左边标出了x_1、x_2和x_3的坐标系。每一个顶点都由三个坐标变量的具体取值而确定。在立方体上画出的函数 f 来自图4-1，在图4-5b中我们也用到函数 f 。图4-34有五个顶点使 f 为1，它们是000、010、100、101和110。这些顶点可以连成五条边（用粗线标出），这五条边分别是x00、0x0、x10、1x0和10x。因为顶点000、010、100和110包含了当 x_3=0时，x_1和x_2的所有取值组合，它们能用xx0表示，xx0这一项表示不论 x_1与 x_2的取值如何，只要x_3=0，则 f =1。xx0 表示的是立方体的正前面，用灰色阴影标记。

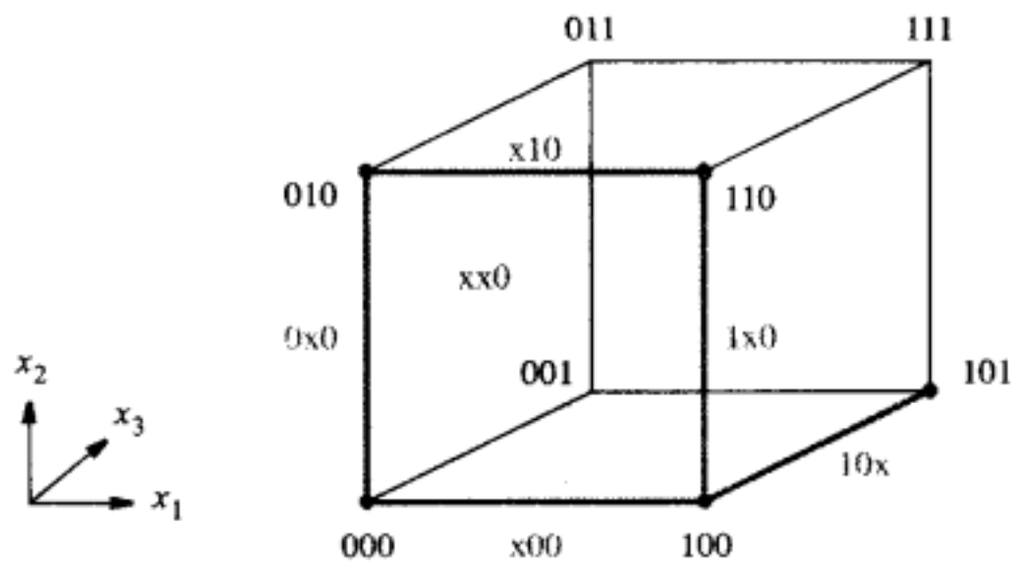


图4-34 函数 $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$

通过前面的讨论，很显然逻辑函数 f 可以有多种表示方法，其中几种可能是：

$$
\begin{aligned}
f &= \{000, 010, 100, 101, 110\} \\
&= \{0x0, 1x0, 101\} \\
&= \{x00, x10, 101\} \\
&= \{x00, x10, 10x\} \\
&= \{xx0, 10x\}
\end{aligned}
$$

物理实现以上表达式时，一个乘积项可以由一个与门实现。很明显，当用 f ={xx0, 10x}表示逻辑函数 f 时我们可以获得成本最低的电路，这种表示等价于逻辑表达式：

$$f = \bar{x}_3 + x_1\bar{x}_2$$

这正是我们在图4-5b中用卡诺图得到的结果。

四维立方体

画出二维和三维立方体的图形都比较容易，而画出四维立方体的图形就比较难。图4-35所示是一个四维立方体的图形，它由顶点相连的两个三维立方体组成。最简单的使四维立方形象化的方法是使一个立方体位于另一个立方体的内部，图4-35用的就是这种方法。我们假设x_1、x_2和x_3三个坐标轴与图4-34所规定的相同，x_4= 0对应外部的立方体，x_4= 1对应内部的立方体。图4-35显示出图4-7中的逻辑函数 f_3 如何在四维立方体中表示。为了避免太多的标号使图像混乱，我们只标出了使 f = 1的顶点。同样，连接这些顶点的边都加粗了。

如果四个相邻顶点都可以使 f =1，那么这四个顶点可以由一个平面表示，图中有两组这样的顶点。由0000、0010、1000和1010组成的这一组顶点可以用平面x0x0表示，由0010、0011、0110和0111组成的这一组顶点可以用平面0x1x表示。这两个面都在图中用阴影标出。逻辑函数 f_3可以有几种表示法，例如：

$$
\begin{aligned}
f_3 &= \{0000,0010,0011,0110,0111,1000,1010,1111\} \\
&= \{00x0,10x0,0x10,0x11,x111\} \\
&= \{x0x0,0x1x,x111\}
\end{aligned}
$$

因为x表示它所对应的变量可以取0或1中的任意一个值，所以它可以略去。则由 f = {x0x0, 0x1x, x111}可以得到最简单的电路，这等价于逻辑表达式：

$$f_3 = \bar{x}_2\bar{x}_4 + \bar{x}_1x_3 + x_2x_3x_4$$

我们从图4-7中可以得到相同的表达式。

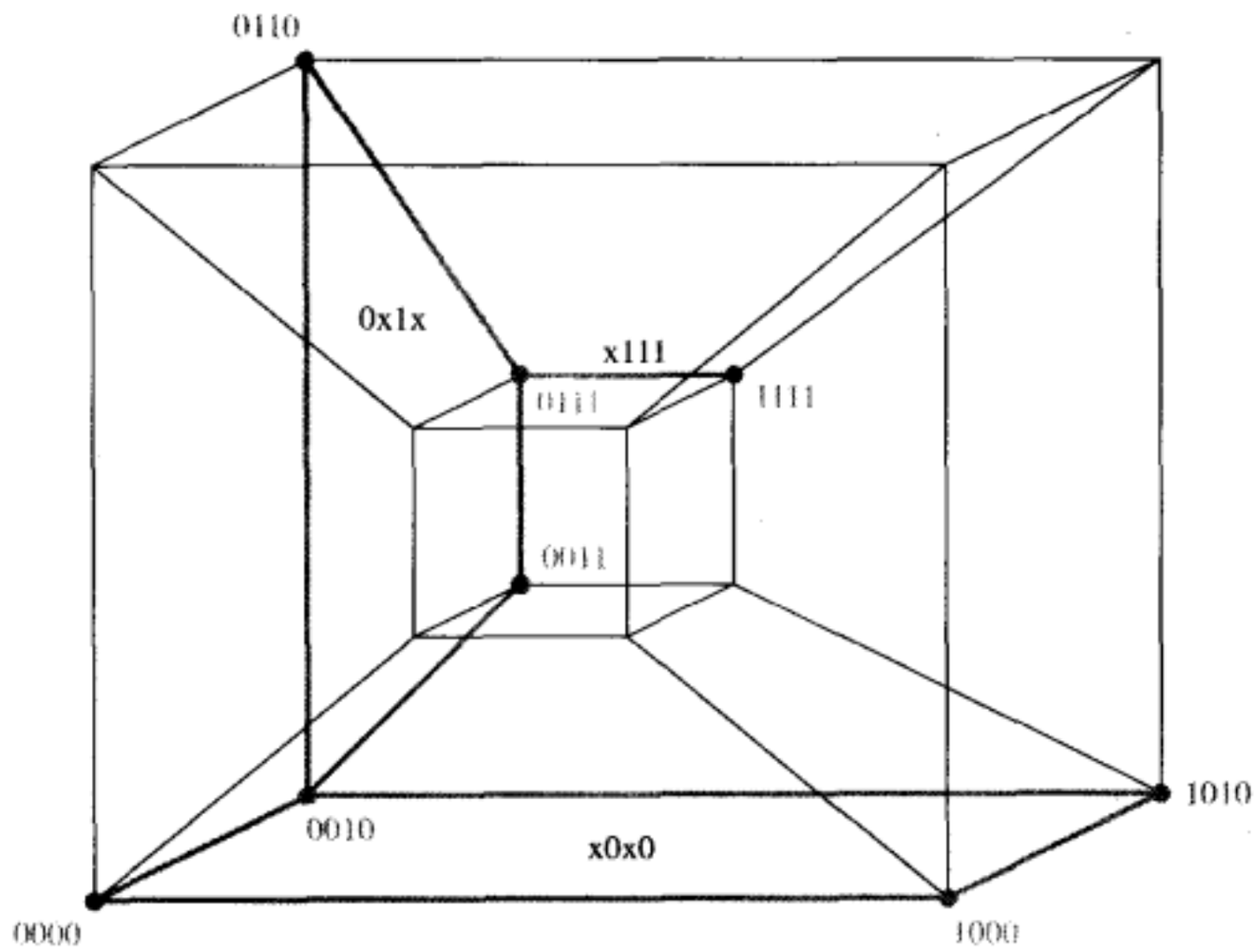


图4-35 图4-7中函数f_3的立方体表示

n维立方体

n变量的逻辑函数可以映射到一个n维立方体上。尽管用图形表示超过四变量的立方体比较困难，我们不难把上面叙述的化简逻辑函数的思路推广到n变量的函数。因为很难用图形表示超过四维的结构，也因为我们通常只用“*立方体*”（cube）这个词描述三维结构，许多人就用“*超立方体*”（hypercube）描述超过三维的结构。在以下的讨论中，我们将继续用“立方体”这个词，而不管其维数是否大于三。

我们用立方体中顶点的数目来描述立方体的大小非常方便。顶点是立方体中最小的单元。一个顶点中的每个变量都有0或1两个取值。若有两个立方体，它们的变量取值除了某一个位置上的变量不同，其余都相同，则在该位置上包含x的立方体比没有x的立方体要大，因为x可以有0和1两个取值。例如，立方体1x01包含1001和1101两个顶点。含有两个x的立方体包含四个顶点，依此类推。具有k个x的立方体含有2^k个顶点。

一个 n 维立方体有2^n个顶点，如果两个顶点只有一个坐标变量的取值不同，那么我们称这两个顶点相邻。因为一个顶点有n个坐标变量（n维立方体有n个坐标轴），所以每一个顶点都与其他n个顶点相邻。在n维立方体中包含有一些低维度空间的立方体。最低维度的立方体是顶点。因为它们的维度是零，我们称它为0维立方体。边是1维立方体。三维立方体中的一个面是2维立方体，整个三维的立方体被称为3维立方体，依此类推。概括地说，我们将把2^k个相邻顶点组成的立方体叫做k维立方体。

从图4-34和图4-35的例子中，我们容易看出一个给定逻辑函数的最大可能立方体对应着逻辑函数的质蕴涵项。在下一节，我们将利用逻辑函数的立方体表示法来讨论逻辑函数的化简。

4.9 列表法化简

逻辑函数的立方体表示法很适合于用计算机来处理逻辑函数的化简，这种化简算法可以通过编程在计算机中高效率地执行。现代CAD工具采用了这种算法。尽管使用CAD工具并不需要知道函数化简的算法究竟是怎样执行的，读者也许会发现了解这方面的知识也是很有趣的。本节将介绍一种相对简单的列表法，该方法说明了最主要的概念，并指出由这种方法引

起的一些问题。

Willard Quine[6] 和 Edward McCluskey[7] 在1950年提出了逻辑函数的列表化简法。这种方法被称为Quine-McCluskey方法，后来它逐渐流行起来。尽管在当代的CAD工具中使用这种方法效率并不高，然而它却是诠释最小化算法关键问题的简单方法。我们将用在4.8节中讨论过的立方体表示来介绍这种方法。

4.9.1 质蕴涵项的产生

正如在4.8节中提到的，给定逻辑函数 f 的质蕴涵项是令逻辑函数 $f=1$ 的最大的 k 维立方体。就非完全指定的函数而言（其中包含一套无关项顶点），其质蕴涵项是最大可能的 k 维立方体，被包含于其中的顶点取值可能是1，也可能还未指定。

我们假设所有使 $f=1$的项都以最小项的形式给出。同样，无关项也以最小项的形式给出。这样我们可以给出一个顶点列表，其中顶点的值可能为1，也可能未指定。我们以成对的方式比较这些顶点，观察是否能由这些顶点组成维度更高的立方体。然后把这些新的立方体继续组合成更大的立方体，持续这个过程直到我们找到质蕴涵项为止。

这种方法的基础就是布尔代数中的结合律：

$$x_i x_j + x_i \overline{x}_j = x_i$$

我们用这个性质导出了立方体表示法。如果有两个立方体，它们只有一个坐标变量取值不同，而其他所有坐标变量的取值都相同，那么这两个立方可以合成一个更高一维的立方体。例如，考虑函数 $f(x_1,\cdots,x_4)$={1000 ,1001,1010,1011}。立方体1000和1001只有坐标变量x_4的取值不同，它们能被合成为一个新的立方体100x。同样，1010和1011能被合成为101x，然后能把100x和101x合成为更高一维的立方体10xx，这意味着逻辑函数 f 可以被简化为：$f=x_1\overline{x}_2$。

图4-36表示了我们如何得出图4-11中逻辑函数 f 的质蕴涵项，逻辑函数的定义如下：

$$f(x_1,\cdots,x_4)=\sum m\,(0,4,8,10,11,12,13,15)$$

这个函数没有无关项。因为高一维的立方体可以由只有一个坐标变量取值不同的最小项合成，我们可以把顶点的坐标变量中具有相同“1”的个数的顶点合成一组，以“1”的个数的多少对这些最小项分类，以此来减少配对时顶点的对数。这样，我们只需要把给定组中的立方体与之前组中的所有立方体配对就行。在图4-36中，表1中的最小项就是这样排列的。（请注意，我们把最小项所对应的十进制数也标出以便于讨论。）这些最小项，在4.8节中称为0维立方体，可以合并为表2中的1维立方体。我们在所有被包含在1维立方体中的0维立方体右边做一个标记。现在我们由表2中的1维立方体组成2维立方体。xx00是唯一能产生的2维立方体，我们把它放入表3。同样，把包含在2维立方体中的1维立方体也做上标记。因为此例只存在一个2维立方体，这个函数中不会含有3维立方体。每一个表中没有被标记的立方体就是该

表1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
10	1 0 1 0	✓
12	1 1 0 0	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

表2

0,4	0 x 0 0	✓
0,8	x 0 0 0	✓
8,10	1 0 x 0	
4,12	x 1 0 0	✓
8,12	1 x 0 0	✓
10,11	1 0 1 x	
12,13	1 1 0 x	
11,15	1 x 1 1	
13,15	1 1 x 1	

表3

0,4,8,12	x x 0 0

图4-36 图4-11中函数质蕴涵项的产生

函数的质蕴涵项。因此，质蕴涵项的集合P是：

$$P = \{10x0, 101x, 110x, 1x11, 11x1, xx00\}$$
$$=\{p_1, p_2, p_3, p_4, p_5, p_6\}$$

4.9.2 最小覆盖的确定

得到函数质蕴涵项的集合以后，我们需要从中找出一个覆盖所有使 $f = 1$的最小项的成本最低的子集。作为简化，我们假设电路成本和门的输入数目成正比，电路中门的输入个数与函数的质蕴涵项所含的因子数相等。

为了找到一个成本最低的覆盖，我们首先创建一个*质蕴涵项覆盖表*，表中每一行代表一个质蕴涵项，每一列表示一个最小项。如果某个最小项被质蕴涵项覆盖，我们就在此质蕴涵项所在行与这个最小项所在列的交叉位置划上勾。图4-36的质蕴涵项列在图4-37a的质蕴涵项覆盖表中。如果覆盖表中某一列只有一个勾，那么就说明只有一个质蕴涵项覆盖这个最小项，所以该质蕴涵项必须包含在最终的覆盖中，我们称其为*本质蕴涵项*。p_6就是一个例子，它是唯一包含最小项0和4的质蕴涵项。第二步，我们把本质蕴涵项所对应的行和本质蕴涵项所覆盖的最小项对应的列去掉。因此我们移去p_6这一行和0, 4, 8, 12这四列，这样就得到了图4-37b中的表。

质蕴涵项	最小项 0	4	8	10	11	12	13	15
p_1 = 1 0 x 0			✓	✓				
p_2 = 1 0 1 x				✓	✓			
p_3 = 1 1 0 x						✓	✓	
p_4 = 1 x 1 1					✓			✓
p_5 = 1 1 x 1							✓	✓
p_6 = x x 0 0	✓	✓	✓			✓		

a) 最初的质蕴涵项覆盖表

质蕴涵项	最小项 10	11	13	15
p_1	✓			
p_2	✓	✓		
p_3			✓	
p_4		✓		✓
p_5			✓	✓

b) 去掉本质蕴涵项后的覆盖表

质蕴涵项	最小项 10	11	13	15
p_2	✓	✓		
p_4		✓		✓
p_5			✓	✓

c) 去掉支配行后的覆盖表

图4-37 图4-11中函数覆盖的选择过程

现在，我们可以用*支配行*的概念来简化覆盖表。在图4-37b中，注意到p_1只覆盖最小项10而p_2覆盖最小项10和11。我们说p_2支配（dominate）p_1，因为p_1和p_2成本相同，所以选择p_2比选择p_1要合理，因此我们把p_1从表中去除。同样，p_5支配 p_3，所以把p_3从表中去除。从而，我们得到图4-37c中的覆盖表。这个覆盖表表明我们必须选择p_2从而覆盖最小项10，选择p_5覆盖最小项13，它们同时覆盖了最小项11和15。因此，最终的覆盖是

$$C =\{p_2, p_5, p_6\}$$
$$=\{101x, 11x1, xx00\}$$

这意味着逻辑函数的最低成本实现是

$$f = x_1\bar{x}_2x_3 + x_1x_2x_4 + \bar{x}_3\bar{x}_4$$

这与我们从4.2.2节得到的表达式一致。

在这个例子中，我们用支配行的概念化简覆盖表。我们把被支配的行去掉的原因是它们比支配行所覆盖的最小项少，而其所对应的质蕴涵项的成本却与支配行对应的质蕴涵项的成本相同。然而，如果被支配行质蕴涵项的成本比支配行质蕴涵项的成本低，则不能把被支配行去除。在习题4.25中我们将遇到关于这种情况的一个例子。

列表法同样能用于函数包含无关项的情况，我们在下面的例子中将做讨论。

例4.14 与使逻辑函数 $f=1$的最小项一样，把无关项也列在第一张表中。考虑以下函数：

$$f(x_1,\cdots,x_4)=\sum m(0,2,5,6,7,8,9,13)+D(1,12,15)$$

我们建议读者画出该函数的卡诺图，以使得下面的过程更形象易懂。图4-38描述了质蕴涵项的产生过程，得到的结果是：

$$P=\{00x0,\ 0x10,\ 011x,\ x00x,\ xx01,\ 1x0x,\ x1x1\}$$
$$=\{p_1,p_2,p_3,p_4,p_5,p_6,p_7\}$$

表1

0	0 0 0 0	✓
1	0 0 0 1	✓
2	0 0 1 0	✓
8	1 0 0 0	✓
5	0 1 0 1	✓
6	0 1 1 0	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

表2

0,1	0 0 0 x	✓
0,2	0 0 x 0	
0,8	x 0 0 0	✓
1,5	0 x 0 1	✓
2,6	0 x 1 0	
1,9	x 0 0 1	✓
8,9	1 0 0 x	✓
8,12	1 x 0 0	✓
5,7	0 1 x 1	✓
6,7	0 1 1 x	
5,13	x 1 0 1	✓
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
7,15	x 1 1 1	✓
13,15	1 1 x 1	✓

表3

0,1,8,9	x 0 0 x
1,5,9,13	x x 0 1
8,9,12,13	1 x 0 x
5,7,13,15	x 1 x 1

图4-38 例4.14中函数质蕴涵项的产生过程

图4-39a中是最初的质蕴涵项的覆盖表。因为不需要覆盖无关项，所以没有把无关项列在覆盖表中。表中没有本质蕴涵项。观察可见，最小项8和最小项9所在的列在同一行都有勾号，可以看出最小项8和最小项9都可由质蕴涵项 p_4 和 p_6 覆盖，而最小项 9 在 p_5 所在的行还有一个勾号。我们把这种情况称为*列支配*。因此，列9支配列8。当一列支配另一列时，我们可以移去支配列，列9就是这种情况。注意，这与行支配中移去被支配行（而不是支配行）正好相反。这样做的原因是当我们选择一个质蕴涵项去覆盖被支配列所对应的最小项时，这个质蕴涵项也将覆盖支配列对应的最小项。在这个例子中，选择 p_4 或 p_6 均可覆盖最小项8和9。类似地，列13支配列5，因此列13也可以移去。

移去列9和列13后，我们得到图4-39b中简化后的覆盖表。在这个表中行p_4支配行p_6，行p_7支配行p_5。所以p_5和p_6可以移去，从而给出图4-39c。现在，p_4和p_7分别是覆盖最小项8和5的本质蕴涵项，所以我们得到图4-39d，从此图中可以明显看出p_2覆盖余下的最小项2和6。注意到p_2同时支配行p_1和p_3。

所以函数的最终的覆盖是：

$$C=\{p_2,p_4,p_7\}$$
$$=\{0x10,\ x00x,\ x1x1\}$$

函数实现如下：

$$f = \bar{x}_1 x_3 \bar{x}_4 + \bar{x}_2 \bar{x}_3 + x_2 x_4$$

质蕴涵项	最小项 0	2	5	6	7	8	9	13
p_1 = 0 0 x 0	✓	✓						
p_2 = 0 x 1 0		✓		✓				
p_3 = 0 1 1 x				✓	✓			
p_4 = x 0 0 x	✓					✓	✓	
p_5 = x x 0 1			✓				✓	✓
p_6 = 1 x 0 x						✓	✓	✓
p_7 = x 1 x 1			✓		✓			✓

a) 最初的质蕴涵项覆盖表

质蕴涵项	最小项 0	2	5	6	7	8
p_1 = 0 0 x 0	✓	✓				
p_2 = 0 x 1 0		✓		✓		
p_3 = 0 1 1 x				✓	✓	
p_4 = x 0 0 x	✓					✓
p_5 = x x 0 1			✓			
p_6 = 1 x 0 x						✓
p_7 = x 1 x 1			✓		✓	

b) 移去列9和列13后的质蕴涵项覆盖表

质蕴涵项	最小项 0	2	5	6	7	8
p_1	✓	✓				
p_2		✓		✓		
p_3				✓	✓	
p_4	✓					✓
p_7			✓		✓	

c) 移去行p_5和p_6后得到的覆盖表

质蕴涵项	最小项 2	6
p_1	✓	
p_2	✓	✓
p_3		✓

d) 确定p_4和p_7为本质蕴涵项后得到的覆盖表

图4-39 例4.14中函数最终覆盖的选择过程 ■

在图4-37和图4-39中，我们用行支配和列支配的概念简化覆盖表。这种方法不是所有情况下都适用，如下例所示。

例4.15 考虑函数

$$f(x_1, \cdots, x_4) = \sum m(0, 3, 10, 15) + D(1, 2, 7, 8, 11, 14)$$

这个函数的质蕴涵项的集合是：

$$P = \{00xx, x0x0, x01x, xx11, 1x1x\}$$
$$= \{p_1, p_2, p_3, p_4, p_5\}$$

初始的质蕴涵项覆盖表如图4-40a所示。表中没有本质蕴涵项。同时，表中也没有支配行或支配列。此外，因为所有质蕴涵项都由两个因子所组成，所以它们的成本都相同。因此，这个覆盖表没有提供任何可用的线索来把其化为最低成本覆盖表。

化简这类函数的方法就是利用4.2.2节介绍的分支的概念。我们可以选择任何一个质蕴涵项，比如说p_3，首先使这个质蕴涵项包含在最终覆盖表中。然后可以用通常的方法来确定必须包含在最终覆盖表中的其他质蕴涵项，从而计算其成本。下一步，我们把p_3排除在最终覆盖表之外，也计算所得结果的成本。我们把两种结果所花费的成本进行比较，从中选出成本低的作为最终简化的结果。

图4-40b给出了p_3包含在最终覆盖表中时，覆盖表中其他的项。表中没有包含最小项3和10的列，因为这两项被p_3覆盖。这个表格表明，完整的覆盖必须包含p_1或p_2其中一项覆盖最小项0，还必须包含p_4或p_5其中一项覆盖最小项15。因此，完整的覆盖表可以是：

$$C = \{p_1, p_3, p_4\}$$

如果把p_3排除在最终覆盖表之外，我们得到图4-40c中的覆盖表。从这里我们看到一个最低成本的覆盖只需要两个质蕴涵项。一种可能是选择p_1和p_5。另一种可能是选择p_2和p_4。因此，

最低成本的覆盖就是

$$C_{min} = \{p_1, p_5\}$$
$$= \{00xx,\ 1x1x\}$$

这个函数的最简实现表达式为：

$$f = \bar{x}_1\bar{x}_2 + x_1x_3$$

质蕴涵项	最小项 0	3	10	15
p_1 = 0 0 x x	✓	✓		
p_2 = x 0 x 0	✓		✓	
p_3 = x 0 1 x		✓	✓	
p_4 = x x 1 1		✓		✓
p_5 = 1 x 1 x			✓	✓

a) 最初的质蕴涵项覆盖表

质蕴涵项	最小项 0	15
p_1	✓	
p_2	✓	
p_4		✓
p_5		✓

b) p_3包含在最终覆盖表中所得的覆盖表

质蕴涵项	最小项 0	3	10	15
p_1	✓	✓		
p_2	✓		✓	
p_4		✓		✓
p_5			✓	✓

c) p_3被排除在最终覆盖表之外时所得的覆盖表

图4-40 例4.15中函数最终覆盖的选择过程 ■

4.9.3 列表法小结

列表法可以总结如下：

1. 首先，列一个表格，使令逻辑函数 f 为1的最小项以及无关项都包含在其中，用立方体的连续配对比较方法，产生函数的质蕴涵项。

2. 列一个初始的质蕴涵项覆盖表，使 f 为1的所有最小项都包含在其中。

3. 使本质蕴涵项包含在最终覆盖表中（如果有的话），通过移去质蕴涵项和被覆盖的最小项的方法化简覆盖表。

4. 用行支配和列支配的概念进一步化简覆盖表。只有在被支配行的质蕴涵项的成本大于或等于支配行的质蕴涵项的成本时，才把被支配行移去。

5. 重复步骤3和4直到覆盖表变空或者不可能进一步化简。

6. 如果化简后的覆盖表不为空，则用分支方法确定余下的质蕴涵项中哪些应该包含在最低成本的覆盖中。

列表法阐明了如何使用代数方法产生质蕴涵项。这种方法也指出了一种解决覆盖问题找到成本最低覆盖的简单方法。不过，这种方法有一些实际限制。实际上，逻辑函数很少以最小项的形式给出。函数通常以代数表达式或立方体集合的形式给出。若我们的优化过程需要从最小项的形式开始，则意味着逻辑表达式或立方体集合必须首先扩展为最小项的形式。这样得到的覆盖表会非常大。随着更大的立方体集合的产生，配对简化的计算量也会相应增大，因而计算过程就会比较慢。 若需要简化的函数规模很大，则用覆盖表来选择最优化的质蕴涵项集合需要经过繁冗的计算。

人们一直都在致力于研究和开发缩短生成优化覆盖表所需时间的代数方法。鉴于大多数这种技术都超出了本书的范围，我们只在下一节简单地讨论其中的一种方法。若读者只想学

习使用CAD工具，而对自动实现最小化技术的细节并不感兴趣，则可以跳过这一节继续阅读，这并不会影响阅读的连续性。

4.10 使用立方体表示法最小化函数

假设函数的初始描述以蕴涵项之和的形式给出，每一个蕴涵项并不一定是最小项或质蕴涵项。那么，定义一种运算，使在初始描述中没有明显地给出的其他蕴涵最终以质蕴涵项之和的形式给出，这是非常有用的。星（*）积运算就是这样一种方法。我们将简单地把这种运算称为“*”运算。

星积运算

两个立方体若只有一个变量取值不同，则星积运算可以将它们合并成为一个新的立方体。设A和B为n变量函数中的两个蕴涵项，用立方体表示法表示，记为：

$$A = A_1A_2\cdots A_n \text{和} B = B_1 B_2\cdots B_n$$

每一个坐标（变量）A_i或B_i的取值可以是0、1或x。星积运算有两个步骤。第一步，根据图4-41，每一组A_i, B_i（$i = 1, 2,\cdots, n$）按坐标进行运算。基于每一个坐标运算得到的结果，由一系列的规则得到星积运算的整体结果。图4-41定义了按坐标的星积运算规则，对于A_i和B_i的所有取值组合，给出了$A_i * B_i$的结果，这就是A与B的取值在表中相交的部分。注意当A_i与B_i取值不同时（一个取值为0，另一个取值为1），$A_i * B_i$的结果为空，以符号∅表示。利用这个表格，A与B的星积运算定义如下：

$$C = A*B$$

规则1：当在多于一个坐标上$A_i * B_i = \varnothing$时，$C = \varnothing$。

规则2：当以上条件不满足时，若$A_i * B_i \neq \varnothing$，则$C_i = A_i * B_i$，若$A_i * B_i = \varnothing$，则$C_i = \text{x}$。

举几个实际例子，能帮助我们理解星积运算的规则：

A_i \ B_i	0	1	x
0	0	∅	0
1	∅	1	1
x	0	1	x

$A_i * B_i$

图4-41 按坐标的星积运算规则

例1：设A={0x0}，B={111}。则$A_1*B_1 = 0*1=\varnothing$，$A_2 *B_2 = \text{x}*1=1$，$A_3 * B_3 = 0*1=\varnothing$，因为在两个坐标上星积运算结果为∅，根据规则1，所以$A*B=\varnothing$。换句话说，这两个立方体不能被合并成一个新的立方体，因为它们有两个坐标的变量取值不同。

例2：设A={11x}和B={10x}。在这种情况下$A_1*B_1=1*1=1$，$A_2*B_2=1*0=\varnothing$，$A_3*B_3= \text{x} * \text{x} = \text{x}$。根据规则2，$C_1=1$，$C_2=\text{x}$，$C_3=\text{x}$，这使得 $C = A*B = \{1\text{xx}\}$。由于两个1维立方体只在一个坐标上取值不同，它们进行星积运算的结果是一个2维立方体。

例3：星积运算也可能产生维数小于参与运算的立方体的立方体。设A={1x1}和B={11x}。则$C=A*B$={111}。注意到C被同时包含在A和B中，则C对于寻找质蕴涵项没有用处。所以，在最小化算法中应当把C删去。

例4：设 A={x10}和B={0x1}。则$C=A*B$={01x}。这三个立方体的维数相同，C 既不被A包含也不被B包含。因此，在找寻质蕴涵项时，C应该保留。画出A、B和C的卡诺图，将有助于读者理解C与A和B之间的联系。

用星积运算寻找质蕴涵项

星积运算的本质是用来在存在的立方体对中寻找新的立方体。如果新产生的立方体不能被原来的立方体包含，则它就是我们感兴趣的立方体。寻找质蕴涵项的过程可以归纳如下。

由于逻辑函数f的蕴涵项可以表示成立方体，所以函数f可以用立方体的集合表示，而且

这个集合就是函数 f 的一个覆盖，我们把它设为C^k。设c^i和c^j为C^k中的任意两个立方体。然后对C^k中的所有立方配对进行星积运算，令G^{k+1}为新产生的立方体集合。因此对所有c^i，$c^j \in C^k$：

$$G^{k+1} = c^i * c^j$$

现在用C^k和G^{k+1}可以形成 f 的一个新覆盖。这个新覆盖中的一些立方体可能是冗余的，应该把它们去掉。设新覆盖为：

$$C^{k+1} = C^k \cup G^{k+1} - \text{冗余立方体}$$

这里∪符号表示两个集合的并集，减号（−）表示去掉集合中的某些元素。如果$C^{k+1} \neq C^k$，我们用同样的过程产生覆盖C^{k+2}。如果$C^{k+1} = C^k$，则覆盖中的立方体都是逻辑函数f的质蕴涵项。对一个n变量的逻辑函数而言，这个过程最多需要进行n次。

经由立方体的配对比较后确定必须去除的冗余项。若立方体 $A = A_1A_2\cdots A_n$由立方体$B = B_1 B_2\cdots B_n$所包含，则应该删去立方体A。这种情况等价于对每一个坐标变量$i \in (1, 2, \cdots, n)$，存在 $A_i = B_i$ 或 $B_i = \text{x}$，则 A 包含于 B 中，故A是冗余立方体。

例4.16 考虑图4-9中的函数 $f(x_1, x_2, x_3)$。假设 f 被表示为与最小项m_0、m_1、m_2、m_3和 m_7所对应的顶点的集合。所以其初始覆盖为C^0 ={ 000, 001, 010, 011, 111 }。使用星积运算产生一个新的立方体集合，我们得到 G^1 = {00x, 0x0, 0x1, 01x, x11 }。然后运算 $C^1 = C^0 \cup G^1$ −冗余立方体。观察后发现C^0中每一个立方体都被包含在G^1中的某个立方体中，因而C^0中所有立方体都是冗余立方体，所以$C^1 = G^1$。

下一步是对C^1中的立方体进行星积运算，从而得到：G^2 ={ 000, 001, 0xx, 0x1, 010, 01x, 011 }。注意G^2中所有的立方体都包含在立方体0xx中；所以除了立方体0xx以外，其他都是冗余项。现在我们容易看出：

$$\begin{aligned} C^2 &= C^1 \cup G^2 - \text{冗余项} \\ &= \{ \text{x11, 0xx} \} \end{aligned}$$

因为C^1中除了立方体x11外，其他所有项都是冗余项，都包含在0xx中，所以把它们删去。

对C^2进行星积运算得到G^3 = {011}，则有：

$$\begin{aligned} C^3 &= C^2 \cup G^3 - \text{冗余项} \\ &= \{ \text{x11, 0xx} \} \end{aligned}$$

由于C^3=C^2，所以立方体集合{ x11, 0xx}就是逻辑函数 f 的质蕴涵项集合，这个集合与乘积项x_2x_3和 $\overline{x}_1$ 等价。这与我们在图4-9中利用卡诺图所得到的函数 f 的质蕴涵项集合完全一致。

请注意，上述例子中求质蕴涵项的方法与4.9节中所用的列表法类似，这是因为函数 f 最初都是以最小项集合的形式给出的。■

例4.17 再举一个例子，考虑图4-10中的四变量函数。假设这个函数最初被指定为覆盖：

$$C^0 = \{0101, 1101, 1110, 011\text{x}, \text{x}01\text{x}\}。$$

然后连续地进行星积运算并去除冗余项，得到：

$$\begin{aligned} C^1 &= \{ \text{x01x, x101, 01x1, x110, 1x10, 0x1x} \} \\ C^2 &= \{ \text{x01x, x101, 01x1, 0x1x, xx10} \} \\ C^3 &= C^2 \end{aligned}$$

因此，质蕴涵项是$\overline{x}_2x_3$、$x_2\overline{x}_3x_4$、$\overline{x}_1x_2x_4$、$\overline{x}_1x_3$和$x_3\overline{x}_4$。■

4.10.1 本质蕴涵项的确定

在求得由函数质蕴涵项组成的覆盖后，需要从中萃取函数的最小覆盖。正如我们在4.2.2节曾见到过的，所有本质蕴涵项都应该包含在这个最小覆盖中。为了找到本质蕴涵项，有必

要定义一种能判定一个立方体是否被另一个立方体包含的运算。该运算被称为 # 运算（译者注：英文发音读为“sharp operation”，下面我们翻译为“飒运算”，取飒的中文读音与sharp接近），本书中我们把这种运算称为飒运算，其定义描述如下。

飒（#）运算

同样，用 $A=A_1A_2\cdots A_n$ 和 $B=B_1B_2\cdots B_n$ 表示n 变量逻辑函数中的两个蕴涵项（即立方体）。飒运算 $A \# B$ 求得的结果是“A中不能被B包含的部分”。与星积运算类似，飒运算分两个步骤进行：首先，对i的每一个取值，进行$A_i \# B_i$的运算；然后，根据按坐标进行的飒运算的结果，由一系列运算规则得到整体的结果。图4-42定义了按坐标进行的飒运算的规则。当对每一对（A_i, B_i）进行运算后，再按以下规则进行整体的飒运算：

$$C = A \# B$$

A_i \ B_i	0	1	x
0	ε	$\varnothing$	ε
1	$\varnothing$	ε	ε
x	1	0	ε

$A_i \# B_i$

图4-42 按坐标进行的飒运算规则

规则1：如果存在某个 i 使$A_i \# B_i = \varnothing$，则 $C = A$。

规则2：如果对所有的 i 都使 $A_i \# B_i = \varepsilon$，则 $C = \varnothing$。

规则3：如果以上两条规则的条件都不满足，则对于满足条件$A_i = \mathrm{x}$，$B_i \neq \mathrm{x}$的 i：

$$C = \cup_i (A_1, A_2, \cdots, \overline{B}_i, \cdots, A_n)$$

规则1与立方体A和立方体B不相交的情况对应，即A与B中至少有一个变量不相同，这意味着B没有包含A的任何部分。例如，令A=0x1，B=11x，由按坐标进行的飒运算求得：$A_1 \# B_1 = \varnothing$，$A_2 \# B_2 = 0$，$A_3 \# B_3 = \varepsilon$。则由规则1得到0x1#11x=0x1。

规则2与A完全被B覆盖的情况对应。例如，0x1 # 0xx = $\varnothing$。规则3与A被B部分覆盖的情况对应。在这种情况下，飒运算将产生一个或多个立方体。具体地，对于每一个满足条件$A_i = \mathrm{x}$，$B_i \neq \mathrm{x}$的坐标i，它将产生一个立方体。该立方体除了在坐标i上 A_i 被$\overline{B}_i$置换之外，其他坐标都与A相同。例如，0xx # 01x = 00x，0xx # 010 ={00x, 0x1}。

我们现在说明如何用飒运算来得到本质蕴涵项。设P为给定函数的所有质蕴涵项的集合。令p^i表示集合P中的一个质蕴涵项，DC表示函数 f 的无关项所代表的顶点的集合。（因为我们用角标表示立方体的不同坐标变量，所以我们在这一节中用上标表示不同的质蕴涵项。）那么，当且仅当

$$p^i \# (P - p^i) \# DC \neq \varnothing \text{时}$$

p^i为函数f的本质蕴涵项。这意味着如果存在至少一个使f=1 的顶点被包含在p^i中，同时这个顶点不被其他任何质蕴涵项包含，则p^i为本质蕴涵项。上面的表达式含有DC的原因是：p^i中包含的无关项不一定要被覆盖。$p^i \# (P - p^i)$ 的意思是飒运算被连续运用于P中的每一个质蕴涵项。例如，考虑$P = \{p^1, p^2, p^3, p^4\}$，$DC = \{d^1, d^2\}$。为了检查$p^3$是否为本质蕴涵项，我们对下式求值：

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

如果上式结果为非$\varnothing$，则p^3是本质蕴涵项。

例4.18 在例4.16中，我们得到立方体x11和0xx是图4-9中函数的质蕴涵项。我们可以用以下的运算求它们是不是函数的本质蕴涵项：

$$\mathrm{x11} \# \mathrm{0xx} = 111 \neq \varnothing$$

$$\mathrm{0xx} \# \mathrm{x11} = \{\mathrm{00x}, \mathrm{0x0}\} \neq \varnothing$$

立方体x11是必需的，因为它是覆盖顶点111的唯一的质蕴涵项。质蕴涵项0xx是必需的，因为它是覆盖顶点000、001和010的唯一的质蕴涵项。从图4-9所示的卡诺图也能看出上述结论。■

例4.19 在例4.17中，我们求出图4-10所示函数的质蕴涵项集合是：P = {x01x, x101, 01x1, 0x1x, xx10}。因为这个函数没有无关项，我们进行以下计算

$$x01x \# (P - x01x) = 1011 \neq \varnothing$$

这个结果是按以下步骤得出的：x01x # x101 = x01x，然后x01x # 01x1 = x01x，然后x01x # 0x1x =101x，然后 101x # xx10 =1011。用同样的方法，我们得到：

$$x101 \# (P - x101) = 1101 \neq \varnothing$$
$$01x1 \# (P - 01x1) = \varnothing$$
$$0x1x \# (P - 0x1x) = \varnothing$$
$$xx10 \# (P - xx10) = 1110 \neq \varnothing$$

因此，本质蕴涵项为x01x、x101和xx10，因为它们分别唯一地包含顶点1011、1101和1110。从图4-10的卡诺图中可以明显地看到这一点。

当我们用飒运算判断一个立方体A是否是本质蕴涵项时，要从$(P - A)$中依次选出一个立方体进行飒运算，而每次飒运算可能产生多个立方体。如果是这种情况，那么所产生的每一个立方都必须与$(P - A)$中剩余的立方体进行飒运算。■

4.10.2 求解最小覆盖的完整步骤

介绍了星积运算和飒运算后，我们可以写出求解 n 变量逻辑函数最小覆盖的完整过程。假设函数 f 最初是用令 $f = 1$的顶点集合表示的；这些顶点称为函数的*开集*，记作ON；同时，无关项被描述为函数的*无关集*，记作DC。则函数的最初覆盖是开集和无关集的并集。

我们在4.10节曾经作过解释，函数 f 的质蕴涵项可以由星积运算产生。如4.10.1节所述，飒运算可以用来求出函数 f 的本质蕴涵项。如果本质蕴涵项包含整个开集，则它们组成逻辑函数 f 的成本最低的覆盖。否则，我们还需要寻找其他质蕴涵项，直到开集中的所有顶点都全部被覆盖为止。

对于一个非本质蕴涵项 p^i，如果存在一个成本更低的质蕴涵项 p^j，且被 p^i 所覆盖的所有在开集中的顶点都被p^j 所覆盖，那么p^i 应该被删掉。如果余下的所有非本质蕴涵项成本相同，那么一种可行的方法是：首先强行把其中一个质蕴涵项选择到最终覆盖中，然后得到覆盖的其余部分；其次，把之前强行选择的质蕴涵项排除在覆盖之外，得到另一个最终覆盖，这两个覆盖中成本较低的一个被用来进行物理实现。我们在4.2.2节和4.9.2节中已经用过这种称为*分支法*的方法。

总结以上讨论，求最小化覆盖可以归纳为如下步骤：

1. 令 $C^0 = ON \cup DC$ 为逻辑函数 f 及其无关项的初始覆盖。
2. 用星积运算找出C^0的所有质蕴涵项，令P为所得的质蕴涵项集合。
3. 用飒运算找出所有本质蕴涵项。对于其中一个质蕴涵项p^i，若 $p^i \# (P - p^i) \# DC \neq \varnothing$，则质蕴涵项 p^i 为本质蕴涵项。若本质蕴涵项包含了开集（ON）中所有的顶点，则该本质蕴涵项集合组成最小化覆盖。
4. 若质蕴涵项 p^i 比质蕴涵项 p^j 成本要高，而且 $p^i \# DC \# p^j = \varnothing$，则将 p^i 删去。
5. 找出覆盖所有剩余顶点的成本最低的质蕴涵项。如果各个质蕴涵项的成本相同，则对剩余的质蕴涵项使用启发式的分支法求得成本最低的覆盖。

例4.20 为了说明上述过程，我们通过对逻辑函数 f 的具体求解，来帮助读者理解最小

覆盖的求解过程。逻辑函数 f 的表达式如下所示：

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

上述函数的卡诺图见图4-43所示 。

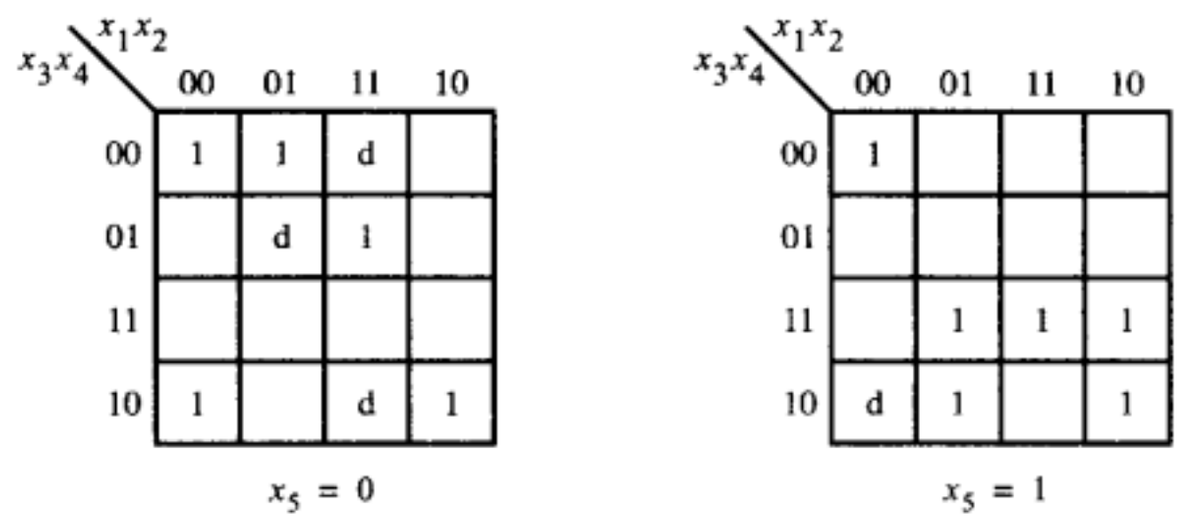


图4-43 例4.20中函数的卡诺图

这里我们以积之和的形式表示逻辑函数 f，而没有用最小项的形式表示：

$$f = \bar{x}_1\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4x_5 + \bar{x}_1x_2x_3x_5 + x_1\bar{x}_2x_3x_5 + x_1x_3x_4x_5 + \bar{x}_2x_3\bar{x}_4\bar{x}_5$$

同样，我们把函数 f 的无关项表示如下：

$$DC = x_1x_2\bar{x}_4\bar{x}_5 + \bar{x}_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4x_5$$

故开集为：

$$ON=\{ 0x000, 11010, 00001, 011x1, 101x1, 1x111, x0100 \}$$

无关项的集合为：

$$DC=\{11x00, 01010, 00101\}$$

初始覆盖 C^0 由开集和无关集组成：

$$C^0=\{0x000, 11010, 00001, 011x1, 101x1, 1x111, x0100, 11x00, 01010, 00101\}$$

使用星积运算所得的一系列的覆盖如下：

C^1 = {0x000, 011x1, 101x1, 1x111, x0100, 11x00, 0000x, 00x00, x1000,
010x0, 110x0, x1010, 00x01, x1111, 0x101, 1010x, x0101, 1x100, 0010x}

C^2 = {0x000, 011x1, 101x1, 1x111, 11x00, x1111, 0x101, 1x100,
x010x, 00x0x, x10x0}

$C^3 = C^2$

所以，$P = C^2$。

使用飒运算，我们找到了两个本质蕴涵项：00x0x（因为它是覆盖顶点00001的唯一质蕴涵项）以及x10x0（因为它是覆盖顶点11010的唯一质蕴涵项）。这两个质蕴涵项覆盖的 f 的最小项是 m (0, 1, 4, 8, 26)。

下一步，我们发现立方体1x100可以被删掉，因为它所覆盖的唯一一个立方体是10100(m_{20})，而这个立方体也被x010x覆盖，并且立方体x010x的成本比1x100的成本低。请注意：删掉1x100后，质蕴涵项x010x成为本质蕴涵项，因为在余下的所有质蕴涵项中只有它覆盖了顶点10100。所以，质蕴涵项x010x必须包含在最终的覆盖中，它覆盖了 m(20,21)。

我们还需要找到质蕴涵项覆盖 m(13, 15, 23, 31)。利用分支法，使用质蕴涵项011x1和1x111可以得到成本最低的覆盖。因此，最小覆盖为：

$$C_{minimum}=\{00x0x,\ x10x0,\ x010x,\ 011x1,\ 1x111\}$$

其所对应的积之和表达式为：

$$f=\bar{x}_1\bar{x}_2\bar{x}_4+x_2\bar{x}_3\bar{x}_5+\bar{x}_2x_3\bar{x}_4+\bar{x}_1x_2x_3x_5+x_1x_3x_4x_5$$

尽管用人工进行计算的过程比较繁杂，但是编写一个程序自动实现这个算法并不太困难。读者可以利用图4-43所示的卡诺图检查求出的最小覆盖解的正确性。 ■

4.11 一些实际问题的考虑

前面的几节是为了给读者一个总体概念，使大家了解逻辑函数的化简是怎样在CAD工具中自动进行的。我们介绍的方法并不难理解，但它却存在着一些缺点，它最大的弊端是化简过程中所需要考虑的立方体的数目非常巨大，以至于难以处理。

如果对化简的结果要求并不严格，以至于不是必须要找到成本最低的实现，那么运用启发式的技术在一定的时间内可以得到较好的化简结果。美国加州大学伯克利分校的Espresso程序就以这种启发性算法为基础。Espresso程序通过万维网传遍了世界，它是一个两级优化程序。程序的输入和输出都以立方体的形式表示。Espresso程序运用了一种蕴涵项扩张的技术（习题4.30有关于蕴涵项扩张的说明）而不是用星积运算来求解质蕴涵项。Espresso程序的详细介绍见文献[19]，它的简要介绍可以在文献[3,12]中找到。

加州大学伯克利分校也提供了两种可以用来设计多级电路的软件，它们分别叫做MIS[20]和SIS[21]。它们允许使用者对逻辑电路应用各种多级优化技术。使用者可以用诸如提取公因子和电路分解等各种方法对整个电路或其中一部分进行不同方案的优化实验。SIS还提供了Espresso的二级逻辑最小化算法以及其他优化技术。

市场上有许多商业用的CAD工具。Cadence Design Systems、Mentor Graphics、Synopsys和Synplicity这四家公司的产品被广泛地使用。在万维网上有关于它们产品的信息。每个公司都提供了基于各种类型芯片的逻辑综合软件，例如可编程逻辑器件、门阵列、标准单元和定制芯片。如我们在前面的章节看到的，因为综合一个给定的电路有很多可能的途径，每一种商用CAD软件都有自己的基于启发性的专利优化算法。

为了描述CAD工具，人们创造了一些术语，其中有两个术语由于在工业上得到广泛应用，需要特别提一下。这两个术语分别为：1）*与技术无关的逻辑综合*；2）*技术映射*。第一个术语指的是在电路优化时，不考虑目标芯片的可用技术资源。本章所介绍的大多数技术都属于这一类。第二个术语技术映射指的是确保逻辑综合产生的电路可以由目标芯片的逻辑资源实现。例如，如果目标芯片中只含有与非门，则技术映射的任务是把所有的逻辑运算都用与非门实现。在电路用只有与非门的门阵列实现函数时就需要用到这项技术。另一个例子是当用FPGA实现设计时，技术映射把所有的逻辑运算都用查找表实现。

第12章将详细地讨论CAD工具，展示一个典型的设计流，设计者可能用它来实现数字系统。

4.12 由Verilog代码综合得到电路举例

2.10节讲述了如何编写简单的Verilog代码来描述逻辑函数。本节将提供更多的用Verilog代码编写的电路示例。我们把综合后的代码用两种目标芯片（CPLD和FPGA）实现。

例4.21 考虑图4-44所示的Verilog代码。其所描述的逻辑函数 f 与图4-1所示的真值表对应。如果我们把这段代码编译后用CPLD来实现，CAD工具得到的表达式是：

$$f = \bar{x}_3 + x_1\bar{x}_2$$

这是我们用图4-5b中的卡诺图得到的最小积之和形式。图4-45展示了如何在CPLD中实现该表达式。被编程为闭合的开关在图上用灰色显示。用来实现 f 的门用灰色显示。请注意观察：逻辑函数 f 的实现只用了最上面两个与门。下面三个与门不产生任何输入效果，因为每个与门的输入端都与未使用的输入信号的正和反相连，所以这三个与门的输出总是为0。

```
module example4_21 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   assign f = (~x1 & ~x2 & ~x3) | (~x1 & x2 & ~x3) |
              (x1 & ~x2 & ~x3) | (x1 & ~x2 & x3) | (x1 & x2 & ~x3);

endmodule
```

图4-44 图4-1所示函数的Verilog代码

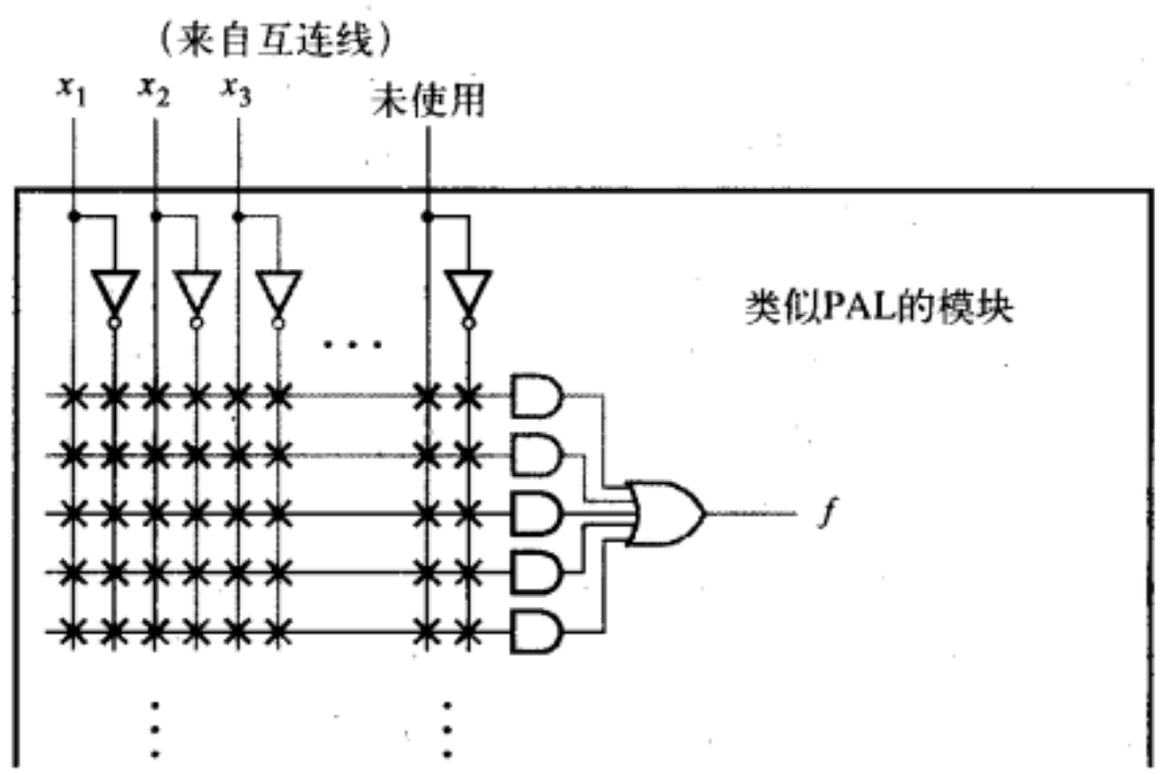
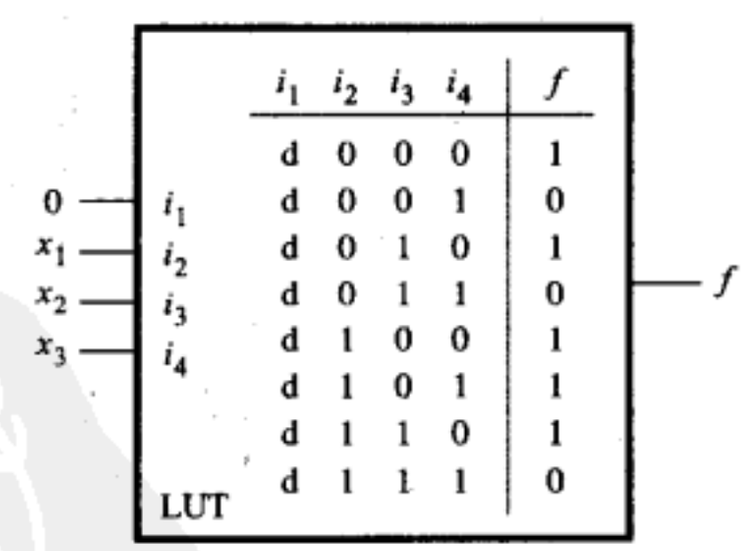


图4-45 图4-44中Verilog代码的实现

图4-46所示为图4-44中的Verilog代码综合到FPGA芯片中的结果。假设综合器产生的积之和的表达式与上面的一样。因为这种芯片的逻辑单元是四输入的查找表，所以函数只需要这样一个单元就能实现函数 f。图表显示变量x_1，x_2和x_3分别与查找表的输入i_2，i_3和i_4相连。因为函数只有三个变量，所以输入i_1没有使用。查找表的真值表显示没有使用的输入被当做无关项对待。因此图中只显示了真值表一半的行（4输入查找表的真值表应该有16行），因为另一半与此完全一样。没有使用的输入在图中与0相连，它同样也可以与1相连。 ■



i_1	i_2	i_3	i_4	f
d	0	0	0	1
d	0	0	1	0
d	0	1	0	1
d	0	1	1	0
d	1	0	0	1
d	1	0	1	1
d	1	1	0	1
d	1	1	1	0

图4-46 用查找表实现图4-44中的逻辑函数

考虑逻辑综合的优化所带来的好处是很有意义的。为了在CPLD上实现函数，把由五个乘积项组成的表达式化简为只有两个乘积项组成的表达式。然而，无论化简的还是未化简的形式都用芯片的一个宏单元实现，所以实现的成本相同（图4-45中的宏单元有五个乘积项）。同样，对于FPGA，因为使用一个查找表实现，函数是否化简对单个查找表没有什么关系。原因是我们举例的电路非常小。而对于大规模电路而言，进行优化是必不可少的。例4.22和例4.23用图说明了逻辑

函数经过优化后将能降低电路的实现成本。

例4.22 图4-47所示的Verilog代码与图4-7所示的逻辑函数 f_1 对应。因为在正则形式的表达式中有六个乘积项，这需要图4-45中的两个宏单元。经由CAD工具综合后，其结果表达式为

$$f = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$$

这与我们在图4-7中推导求得的结果相同。因为优化后的电路只有两个乘积项，所以只需要用一个宏单元就能实现，所以电路的成本就降低了。

```
module example4_22 (x1, x2, x3, x4, f);
   input x1, x2, x3, x4;
   output f;

   assign f = (~x1 & ~x2 & x3 & ~x4) | (~x1 & ~x2 & x3 & x4) |
              (x1 & ~x2 & ~x3 & x4) | (x1 & ~x2 & x3 & ~x4) |
              (x1 & ~x2 & x3 & x4) | (x1 & x2 & ~x3 & x4);

endmodule
```

图4-47 图4-7中函数 f_1 的Verilog代码

当把 f_1 综合到FPGA中实现时，生成的表达式与综合到CPLD中实现的表达式相同。函数 f_1 只有四个输入，因此只需要一个查找表就能实现。■

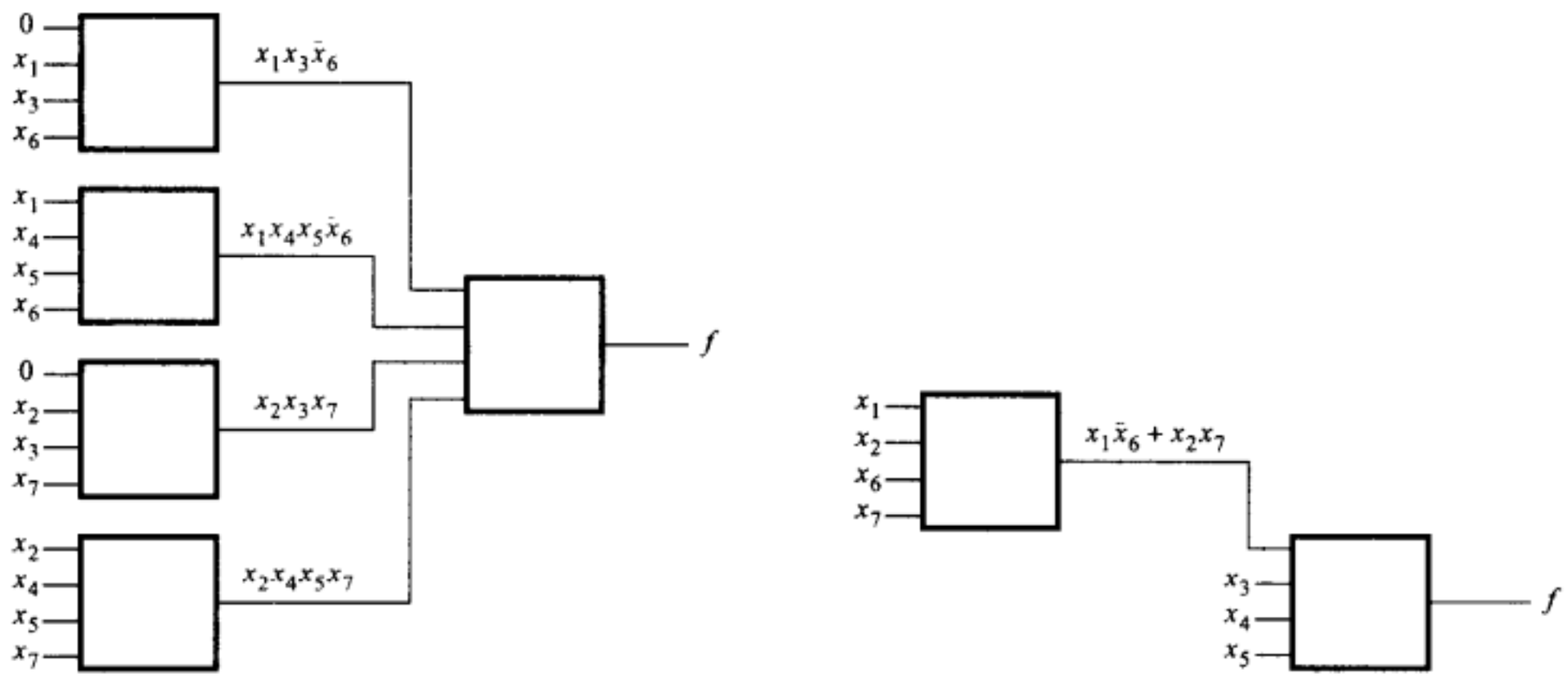
例4.23 在4.6节中，我们曾用一个七变量的逻辑函数引入了多级综合的概念。该函数的Verilog代码如图4-48所示。因为逻辑函数是以最小的积之和形式的表达式给出，所以当用CPLD实现该函数时，CAD工具并不需要对函数进行优化，只需要一个宏单元就能实现函数。若我们考虑用四输入查找表的FPGA实现这个函数，就会发现一件很有意思的事情。因为该函数有七个输入，一个查找表只能处理四个输入，因此该函数的实现需要一个以上的查找表。若直接用所给的Verilog代码实现该函数所对应的电路，则需要五个查找表，见图4-49a所示。我们在图4-49a中的查找表输出处标出由该LUT实现的逻辑函数，而不是展示每个配置后查找表对应的真值表。但是，如果函数用下面的表达式表示：

$$f = (x_1 \bar{x}_6 + x_2 x_7)(x_3 + x_4 x_5)$$

这是利用4.6节的公因子提取法推导出的结果，则如图4-49b中所诠释的，只用两个查找表便可以实现上述表达式。由一个查找表实现 $S = x_1 \bar{x}_6 + x_2 x_7$ 项。另一个查找表实现四输入函数 $f = Sx_3 + Sx_4 x_5$ 项。

```
module example4_23 (x1, x2, x3, x4, x5, x6, x7, f);
   input x1, x2, x3, x4, x5, x6, x7;
   output f;

   assign f = (x1 & x3 & ~x6) | (x1 & x4 & x5 & ~x6) |
              (x2 & x3 & x7) | (x2 & x4 & x5 & x7);

endmodule
```

图4-48 4.6节中函数的Verilog代码



a) 积之和表达式　　b) 因式分解表达式的实现

图4-49　图4-48中的Verilog代码的实现

4.13 小结

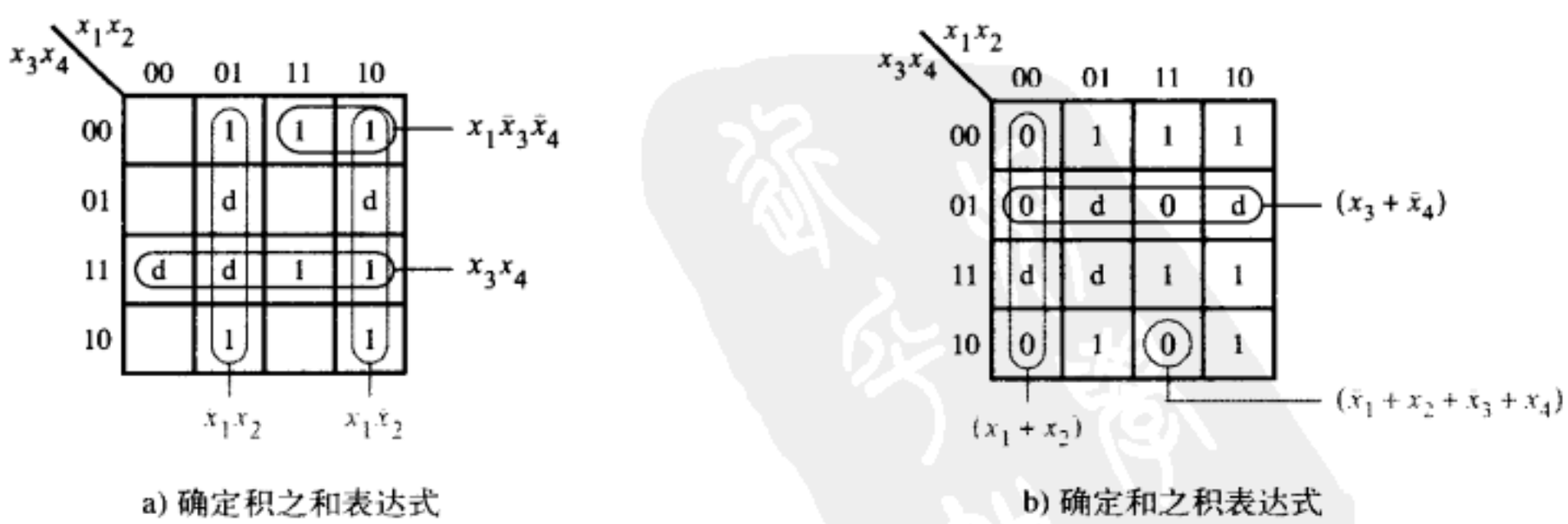
本章的宗旨是提供有关逻辑函数综合的各方面知识。到目前为止，读者已具备了一定的基本概念，可以研究更复杂的数字电路。下一章将讨论怎样用电路进行算术运算，这是计算机的关键部件。

4.14 问题求解举例

本节介绍了一些读者可能会遇到的典型问题，并展示如何解答这类问题。

例4.24　设有逻辑函数 $f(x_1, x_2, x_3, x_4) = \sum m(4, 6, 8, 10, 11, 12, 15) + D(3, 5, 7, 9)$，确定其最低成本的积之和和和之积表达式。

解：该函数可以表示为卡诺图的形式，如图4-50a所示。注意该卡诺图的最小项位置如图4-6所示。为了找到成本最低的积之和表达式，有必要找到覆盖该卡诺图所有1的质蕴涵项(prime implicant)。无关项可以根据需要使用。最小项m_6只被质蕴涵项$\bar{x}_1x_2$覆盖，因此该质蕴涵项是基本的，必须被包含在最后的表达式中。同样，$x_1\bar{x}_2$和x_3x_4也是基本的，因为它们只是分别覆盖m_{10}和m_{15}的质蕴涵项。



a) 确定积之和表达式　　b) 确定和之积表达式

图4-50　例4.24的卡诺图

这三个质蕴涵项覆盖了除m_{12}外所有 $f=1$的最小项。最小项m_{12}可以用两种方法予以覆盖，

即$x_1\overline{x}_3\overline{x}_4$或者$x_2\overline{x}_3\overline{x}_4$。由于这两个质蕴涵项的成本相同，所以随意选择其中一个即可。选择前一个，即$x_1\overline{x}_3\overline{x}_4$，便可得到想要的积之和表达式：

$$f = \overline{x}_1x_2 + x_1\overline{x}_2 + x_3x_4 + x_1\overline{x}_3\overline{x}_4$$

这几个质蕴涵项在卡诺图中都用画圈的方法予以确认。

想要的和之积表达式可以用图4-50b所示的方法予以确定。在这种场合，我们必须找到覆盖此函数所有为0的求和项。注意，我们在卡诺图中清楚地画出了所有为0的求和项，以强调这个事实。为了覆盖卡诺图第0和第2号方格中的0项，即对应于最大项的M_0和M_2，$(x_1 + x_2)$项是基本的。$(x_3 + \overline{x}_4)$ 和 $(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$ 这两项必须分别用来覆盖卡诺图第13和14号方格中的0项。由于这三个求和项覆盖了卡诺图中所有的0项，因此得到其和之积表达式如下：

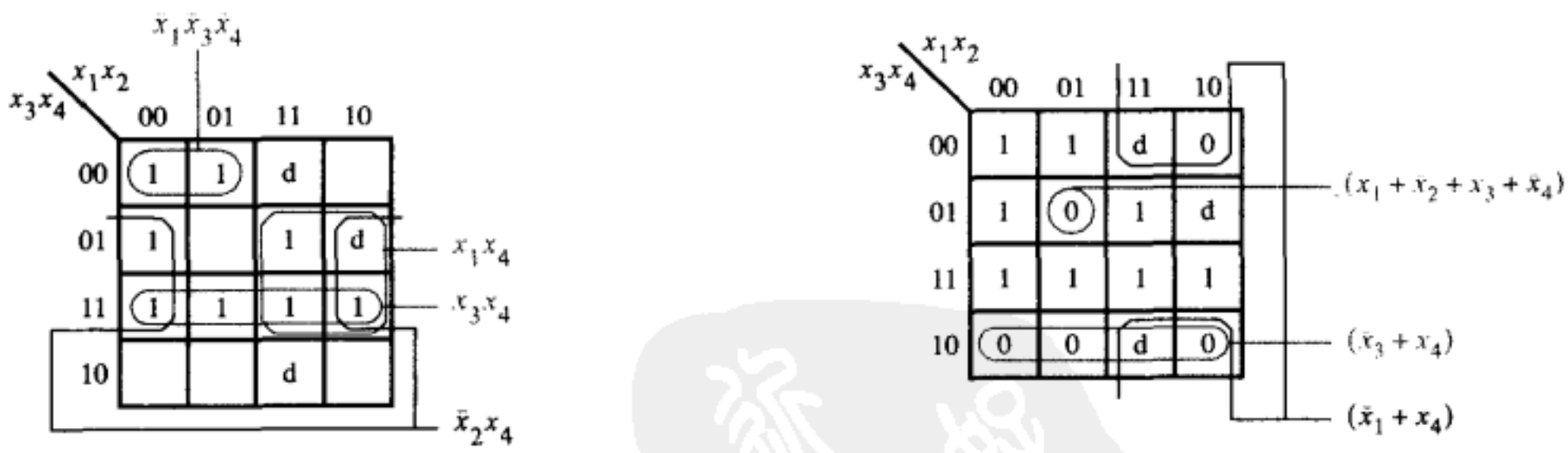
$$f = (x_1 + x_2)(x_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$$

被选中的求和项在该卡诺图中用画圈的方法予以确认。

观察该例子中使用无关项的情况。为了得到成本最低的积之和表达式，我们曾假设将所有四个无关项都设置为1。只有把无关项3，5和9设置为0，无关项7设置为1，我们才有可能得到成本最低的和之积表达式。这就意味着求得的积之和表达式以及和之积表达式就它们所代表的函数项而言，并不完全相同。就 f 被指定为1或0的所有取值而言，积之和表达式与和之积表达式是完全相同的，但是就无关项，即卡诺图第3，5 和 9号的取值而言，逻辑函数 f 是不同的。当然这种不同不会造成任何问题，因为这些无关项的取值从来不会用作实现电路的输入。

例4.25 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$，用卡诺图找出其最低成本的积之和和和之积表达式，假设该逻辑函数的无关项为 $D = \Sigma(9, 12, 14)$。

解：表示该函数的卡诺图如图4-51a所示。将用于定义 f 的表达式的每个乘积项所对应的卡诺图的最小项处放置1。$\overline{x}_1\overline{x}_3\overline{x}_4$项对应于最小项0和4。$x_3x_4$项表示该卡诺图的第3行，包含最小项3，7，11和15。$\overline{x}_1\overline{x}_2x_4$项指定最小项1和3。第四个乘积项表示的是最小项13。该卡诺图还包含三个无关条件。



a) 确定积之和表达式　　b) 确定和之积表达式

图4-51 例4.25的卡诺图

为了找到想要的积之和表达式，我们必须找到一组成本最低的覆盖该卡诺图中所有1的质蕴涵项。x_3x_4项是质蕴涵项，因为它是覆盖最小项7的唯一质蕴涵项，此外它还覆盖了最小项3，11和15。最小项4可以被$\overline{x}_1\overline{x}_3\overline{x}_4$或者$x_2\overline{x}_3\overline{x}_4$所覆盖。这两个项的成本相同，我们选择$\overline{x}_1\overline{x}_3\overline{x}_4$，因为它还覆盖最小项0。最小项1可以被$\overline{x}_1\overline{x}_2\overline{x}_3$或者$\overline{x}_2x_4$覆盖，我们应该选择$\overline{x}_2x_4$，因为它的成本更低。于是只剩下最小项13需要覆盖，可以用x_1x_4或者x_1x_2将其覆盖，它们的成本是相同的。选择x_1x_4

覆盖最小项13，得到成本最低的积之和表达式如下：

$$f = x_3x_4 + \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2x_4 + x_1x_4$$

图4-15b展示了如何找到和之积表达式。求和项（$\overline{x}_3 + x_4$）覆盖了卡诺图最下面一行的0。为了覆盖第8号方格中的0，必须包括（$\overline{x}_1+x_4$）。余下的第5号方格中的0必须由（$x_1 + \overline{x}_2 + x_3 + \overline{x}_4$）覆盖。由此得到成本最低的和之积表达式为

$$f = (\overline{x}_3 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + x_3 + \overline{x}_4)$$

例4.26 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$，用4.9节介绍的列表法推导其最低成本的积之和表达式，假设该逻辑函数的无关项为 $D = \sum(9, 12, 14)$ 。

解：列表法要求一开始就以最小项来定义逻辑函数。正如图4-51所示，函数 f 可以表示为如下表达式：

$$f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 3, 4, 7, 11, 13, 15) + D(9, 12, 14)$$

其对应的11个0-立方体见图4-52的表1。

表1

0	0 0 0 0	✓
1	0 0 0 1	✓
4	0 1 0 0	✓
3	0 0 1 1	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
14	1 1 1 0	✓
15	1 1 1 1	✓

表2

0,1	0 0 0 x	
0,4	0 x 0 0	
1,3	0 0 x 1	✓
1,9	x 0 0 1	✓
4,12	x 1 0 0	
3,7	0 x 1 1	✓
3,11	x 0 1 1	✓
9,11	1 0 x 1	✓
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
12,14	1 1 x 0	✓
7,15	x 1 1 1	✓
11,15	1 x 1 1	✓
13,15	1 1 x 1	✓
14,15	1 1 1 x	✓

表3

1,3,9,11	x 0 x 1
3,7,11,15	x x 1 1
9,11,13,15	1 x x 1
12,13,14,15	1 1 x x

图4-52 例4.26中逻辑函数质蕴涵项的生成

现在成对地比较所有的0-立方体来确定显示于表2的1-立方体，表2中列出的是由0-立方体对组合而得到的。注意：所有的0-立方体包括在1-立方体中，正如表1右侧的勾号所标明的那样。接下去，成对地比较所有的1-立方体来确定显示于表3的2-立方体。某些2-立方体可以用多种方法生成，但是将一个2立方体列出一次以上没有用（例如，表3中的x0x1可以通过组合表2中的立方体1，3和9，11得到或者通过组合立方体1，9和3，11得到）。请注意，除了三个1-立方体外，所有的1-立方体都包括在2-立方体中。不可能产生任何3-立方体，因此没有包括进某些其他项的所有项（即表2中没有打勾标记的项和表3中的所有项）都是 f 函数的质蕴涵项。因此，质蕴涵项的集合是：

$$P = \{000x, 0x00, x100, x0x1, xx11, 1xx1, 11xx\}$$
$$= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$$

为了找到 f 函数成本最低的覆盖，构建图4-53a所示的表，该表展示了所有的质蕴涵项以及必须覆盖的最小项，即那些使 $f = 1$ 的项。表中的勾号标明表头的最小项已被特定的质蕴涵项所覆盖。因为最小项7只被p_5所覆盖，所以该质蕴涵项必须包含在最后的覆盖中。我们观察到p_2行覆盖了p_3行，因此p_3行可以删除。同样p_6行覆盖了p_7行，所以可以删除p_5，p_3和p_7行，也

可以删除3，7，11 和15（该列由p_5覆盖）列。由此得到简化的表，如图4-53b所示。在这个表中p_2和p_6是本质的，它们覆盖了最小项0，4和13。因此只剩下最小项1需要覆盖，选择p_1或者p_4便可以将其覆盖。因为p_4的成本更低一些，所以选择p_4。因此最后的覆盖为

$$C = \{p_2, p_4, p_5, p_6\}$$
$$= \{0x00, x0x1, xx11, 1xx1\}$$

且该函数可以用以下表达式实现：

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_3x_4 + x_1x_4$$

质蕴涵项	最小项 0	1	3	4	7	11	13	15
p_1 = 0 0 0 x	✓	✓						
p_2 = 0 x 0 0	✓			✓				
p_3 = x 1 0 0				✓				
p_4 = x 0 x 1		✓	✓			✓		
p_5 = x x 1 1			✓		✓	✓		✓
p_6 = 1 x x 1						✓	✓	✓
p_7 = 1 1 x x							✓	✓

a) 初始质蕴涵项覆盖表

质蕴涵项	最小项 0	1	4	13
p_1 = 0 0 0 x	✓	✓		
p_2 = 0 x 0 0	✓		✓	
p_4 = x 0 x 1		✓		
p_6 = 1 x x 1				✓

b) 删除行p_3，p_5和p_7以及列3, 7, 11和15后的表

图4-53 例4.26中逻辑函数的覆盖

例4.27 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$，用星积运算找到它的所有质蕴涵项。假设该逻辑函数的无关项为 $D = \sum(9,12,14)$。

解：该逻辑函数的1集合（ON-set）可以用下式表示：

$$ON = \{0x00, xx11, 00x1, 1101\}$$

由1集合（ON-set）和无关项组成的初始覆盖可以用下式表示：

$$C^0 = \{0x00, xx11, 00x1, 1101, 1001, 1100, 1110\}$$

用星积运算，可以得到后续覆盖表达式如下：

$C^1 = \{0x00, xx11, 00x1, 000x, x100, 11x1, 10x1, 111x, x001, 1x01, 110x, 11x0\}$

$C^2 = \{0x00, xx11, 000x, x100, x0x1, 1xx1, 11xx\}$

$C^3 = C^2$

因此，表示所有质蕴涵项集合的表达式如下：

$$P = \{\bar{x}_1\bar{x}_3\bar{x}_4, x_3x_4, \bar{x}_1\bar{x}_2\bar{x}_3, x_2\bar{x}_3\bar{x}_4, \bar{x}_2x_4, x_1x_4, x_1x_2\}$$

例4.28 设有逻辑函数 $f(x_1, x_2, x_3, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$，请用最低成本实现该函数。假设该逻辑函数的无关项为 $D = \sum(9,12,14)$。

解：该函数与例4.25到例4.27所用的函数完全相同。在例4.25中，我们曾找到成本最低的积之和表达式为

$$f = x_3x_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_1x_4$$

实现上述表达式的电路需要4个与门、1个或门以及13个到这些门的输入，总成本是18。成本最低的和之积表达式为

$$f = (\bar{x}_3 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

实现上述表达式的电路需要3个或门、1个与门以及11个到这些门的输入，总成本是15。

我们也可以考虑用多层次的途径实现该函数。对上面的和之积表达式使用因子分解的方法产生如下表达式：

$$f=(x_1+\overline{x}_2+x_3)x_4+\overline{x}_1\overline{x}_3\overline{x}_4$$

实现上述表达式的电路需要2个与门、2个或门以及10个到这些门的输入，总成本是14。对积之和表达式与和之积表达式进行比较后，可以发现从门和输入端总数的角度而言，上述表达式的成本是最低的，但是就电路运行速度而言却是最慢的，因为从信号的输入到函数的输出必须经过三层门的传播延迟。当然，若在FPGA中实现此函数，只需要用一个查找表（LUT）即可。

例4.29 有几种商用FPGA芯片，其逻辑块采用四输入查找表（LUT）。两个这样的查找表如图4-54所示连接起来，通过逻辑函数分解的方法，可以用来实现七变量的逻辑函数：

$$f(x_1,\cdots,x_7)=f[g(x_1,x_2,x_3,x_4),x_5,x_6,x_7]$$

可以看到，诸如$f=x_1x_2x_3x_4x_5x_6x_7$和$f=x_1+x_2+x_3+x_4+x_5+x_6+x_7$的函数能够很容易地用查找表级连的形式实现。请展示确实存在一些其他类型的七变量逻辑函数，它们是不能用2个四输入查找表实现的。

解：七变量的真值表如图4-55所示。该表可以安排$2^7=128$个最小项。变量x_1，x_2，x_3和x_4的每次取值都可以挑选该真值表16列中的某一列。而x_5，x_6和x_7的每次取值可以挑选该真值表8行中的某一行。因为我们必须使用图4-54所示的电路，所以函数f的真值表可以由子函数g来定义。在这种场合，可以由g来确定真值表16列中的哪一列已被选中，而不是由x_1，x_2，x_3和x_4的取值确定的。因为g只可能有两种取值，即0或者1，因此我们只可以在该真值表中取用两列。在图4-54中，16列中只有当两列存在明显的1和0取值差别时才有可能，因此只有相对较小的一个七变量子集能够只用两个查找表实现。

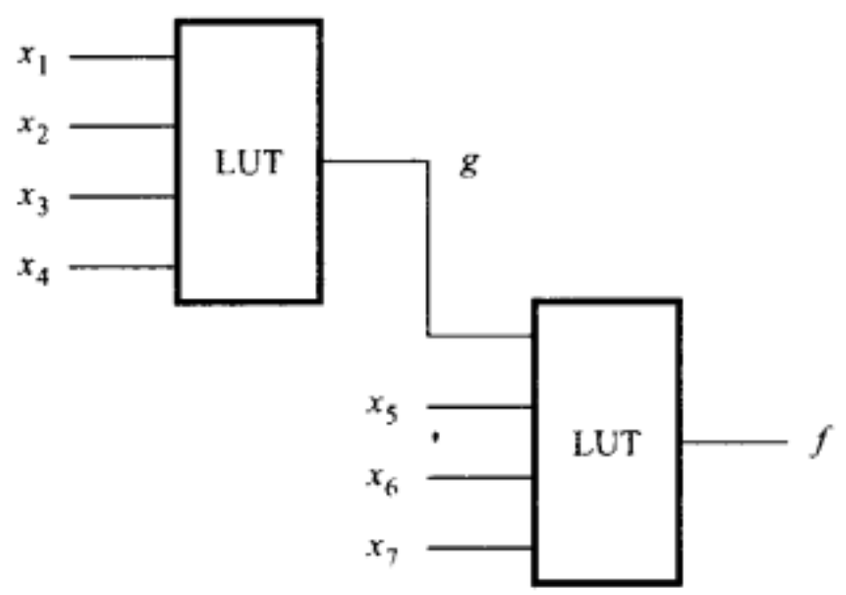


图4-54 例4.29的电路

$x_5x_6x_7$ \ $x_1x_2x_3x_4$	0000	0001	···	1110	1111
000	m_0	m_8		m_{112}	m_{120}
001	m_1	m_9		m_{113}	m_{121}
010	m_2	m_{10}		m_{114}	m_{122}
011	m_3	m_{11}	···	m_{115}	m_{123}
100	m_4	m_{12}		m_{116}	m_{124}
101	m_5	m_{13}		m_{117}	m_{125}
110	m_6	m_{14}		m_{118}	m_{126}
111	m_7	m_{15}		m_{119}	m_{127}

图4-55 七变量函数真值表的可能形式

练习题

标有*号的习题答案参见书后。

* **4.1** 求函数$f(x_1,x_2,x_3)=\sum m(1,2,3,5)$成本最低的积之和与和之积形式的表达式。

* **4.2** 设函数为：$f(x_1,x_2,x_3)=\sum m(1,4,7)+D(2,5)$，同上题的要求。

4.3 设函数为：$f(x_1,\cdots,x_4)=\prod M(0,1,2,4,5,7,8,9,10,12,14,15)$，同上题的要求。

4.4 设函数为：$f(x_1,\cdots,x_4)=\sum m(0,2,8,9,10,15)+D(1,3,6,7)$，同上题的要求。

* **4.5** 设函数为：$f(x_1,\cdots,x_5)=\prod M(1,4,6,7,9,12,15,17,20,21,22,23,28,31)$，同上题的要求。

4.6 设函数为：$f(x_1,\cdots,x_5)=\sum m(0,1,3,4,6,8,9,11,13,14,16,19,20,21,22,24,25)+D(5,7,12,15,17,23)$，同上题的要求。

4.7 设函数为：$f(x_1,\cdots,x_5)=\sum m(1,4,6,7,9,10,12,15,17,19,20,23,25,26,27,28,30,31)+D(8,16,21,22)$，同上题的要求。

4.8 找出五个三变量的函数，其和之积形式必须比其积之和形式的成本低。

* **4.9** 如果一个四变量函数的任意三个变量为1或四个变量全为1时其函数值为1，则称它为多数函数。找出一个实现此函数的成本最低的SOP电路。

4.10 用最低成本实现以下四变量函数：如果函数中两个或三个变量为1其函数值为1，否则函数值为零。

* **4.11** 证明或用反例说明以下命题：若某逻辑函数有唯一的成本最低的积之和表达式，则它也有唯一的成本最低的和之积表达式。

* **4.12** 有一个两输出的函数，其输出为：

$$f(x_1,\cdots,x_4)=\sum m(0,2,4,6,7,9)+D(10,11)$$
$$g(x_1,\cdots,x_4)=\sum m(2,4,9,10,15)+D(0,13,14)$$

假设输入能提供原变量和反变量两种形式，设计一个能输出上述函数的成本最低的电路，再分别实现函数f和g，比较这两种实现方法的成本。

4.13 设有一个两输出的函数，其输出为：

$$f(x_1,\cdots,x_5)=\sum m(1,4,5,11,27,28)+D(10,12,14,15,20,31)$$
$$g(x_1,\cdots,x_5)=\sum m(0,1,2,4,5,8,14,15,16,18,20,24,26,28,31)+D(10,11,12,27)$$

同上题的要求。

* **4.14** 仅用与非门实现图4-23所示的逻辑电路。

* **4.15** 仅用或非门实现图4-23所示的逻辑电路。

4.16 仅用与非门实现图4-25所示的逻辑电路。

4.17 仅用或非门实现图4-25所示的逻辑电路。

* **4.18** 设计一个只用与、或、非门组成的成本最低的电路，实现以下函数：

$$f=x_3x_5+\bar{x}_1x_2x_4+x_1\bar{x}_2\bar{x}_4+x_1x_3\bar{x}_4+\bar{x}_1x_3x_4+\bar{x}_1x_2x_5+x_1\bar{x}_2x_5$$

4.19 用成本最低的电路实现以下函数：

$$f(x_1,\cdots,x_4)=\sum m(4,7,8,11)+D(12,15)$$

4.20 假设逻辑电路的最大扇入为2，以最简形式实现以下电路：

$$f(x_1,\cdots,x_4)=\sum m(0,3,4,7,9,10,13,14)$$

* **4.21** 假设输入变量只有原变量的形式，找出一个成本最低的电路实现以下函数。（提示：用函数分解方法。）

$$f(x_1,\cdots,x_4)=\sum m(0,4,8,13,14,15)$$

4.22 使用函数分解方法找出以下函数的最佳实现：

$$f(x_1,\cdots,x_5)=\sum m(1,2,7,9,10,18,19,25,31)+D(0,15,20,26)$$

这种实现与成本最低的积之和实现相比如何？给出两种实现的成本。

* **4.23** 用4.9节中的图表法找出以下函数成本最低的积之和实现：

$$f(x_1,\cdots,x_4)=\sum m(0,2,4,5,7,8,9,15)$$

4.24 设函数为：

$$f(x_1,\cdots,x_4)=\sum m(0,4,6,8,9,15)+D(3,7,11,13)$$

同上题的要求。

4.25 设函数为：

$$f(x_1,\cdots,x_4)=\sum m(0,3,4,5,7,9,11)+D(8,12,13,14)$$

同上题的要求。

4.26 说明以下与分配律相似的规则的合理性：

$$(A\cdot B)\#C=(A\#C)\cdot(B\#C)$$

$$(A+B)\#C=(A\#C)+(B\#C)$$

4.27 使用立方表示法和4.10节中介绍的方法，找出以下函数成本最低的积之和实现：

$$f(x_1,\cdots,x_4)=\sum m(0,2,4,5,7,8,9,15)$$

4.28 设函数为：

$$f(x_1,\cdots,x_5)=\bar{x}_1\bar{x}_3\bar{x}_5+x_1x_2\bar{x}_3+x_2x_3\bar{x}_4x_5+x_1\bar{x}_2\bar{x}_3x_4+x_1x_2x_3x_4\bar{x}_5+\bar{x}_1x_2x_4\bar{x}_5+\bar{x}_1\bar{x}_3x_4x_5$$

同上题的要求。

4.29 使用立方表示法和4.10节介绍的方法，找出函数 $f(x_1,\cdots,x_4)$的最低成本的积之和实现。函数f表示为：

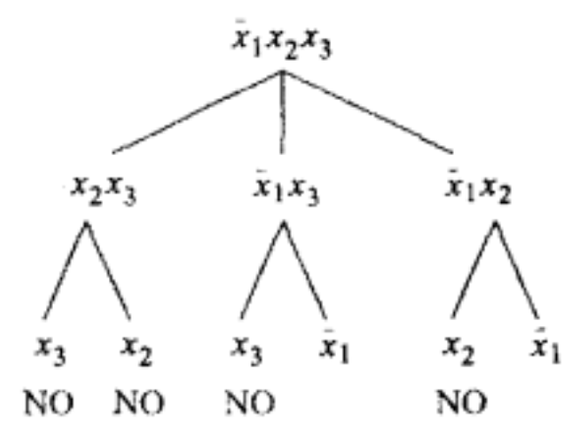
真值项集合$ON=\{00x0,100x,x010,1111\}$

不定项集合$DC=\{00x1,011x\}$

4.30 在4.10.1节中我们展示了如何应用星运算找出给定函数 f 的质蕴涵项。另一种求质蕴涵项的方法是对函数的初始覆盖中的蕴涵项进行扩展。去掉蕴涵项中的一个因子就可以得到一个更大的蕴涵项（去掉因子后被覆盖的顶点数增加）。只有在这个大的蕴涵项不包含使 $f=0$的顶点时它才有效。扩展所得到的最大的有效蕴涵项就是质蕴涵项。图4-9是函数 f 的卡诺图表示（例4.16也用到该函数），对函数 f 的蕴涵项$\bar{x}_1x_2x_3$进行扩展，扩展过程如图P4-1所示。从图4-9可以得出：

$$\bar{f}=x_1\bar{x}_2\bar{x}_3+x_1\bar{x}_2x_3+x_1x_2\bar{x}_3$$

图P4-1中的单词NO表示扩展所得项是无效的，因为它包含了一个或多个 $\bar{f}$ 的顶点。由图P4-1可以明显看出，扩展所得的最大有效蕴涵项是x_2x_3和 $\bar{x}_1$；它们是函数 f 的质蕴涵项。



图P4-1 蕴涵项$\bar{x}_1x_2x_3$的扩展

对例4.14中初始覆盖的4个蕴涵项进行扩展，从而求出函数 f 的所有蕴涵项。并对扩展算法和星运算求质蕴涵项的复杂性加以比较。

4.31 例4.17中函数的初始覆盖为C^0，按照习题4.30的要求对其蕴涵项进行扩展。

*** 4.32** 考虑逻辑表达式：

$$f=x_1\bar{x}_2\bar{x}_5+\bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5+x_1x_2x_4x_5+\bar{x}_1\bar{x}_2x_3\bar{x}_4+x_1\bar{x}_2x_3x_5+\bar{x}_2\bar{x}_3x_4\bar{x}_5+x_1x_2x_3x_4\bar{x}_5$$

$$g=\bar{x}_2x_3\bar{x}_4+\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5+x_1x_3x_4\bar{x}_5+x_1\bar{x}_2x_4\bar{x}_5+x_1x_3x_4x_5+\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_5+x_1x_2\bar{x}_3x_4x_5$$

证明f是否等于g。

4.33 图P4-2所示电路的输出为 f 和 g，假定输入提供原变量和反变量两种形式，问电路的成本是多少？请另外设计一个电路，以尽可能低的成本实现同样的功能，同时给出成本的值。

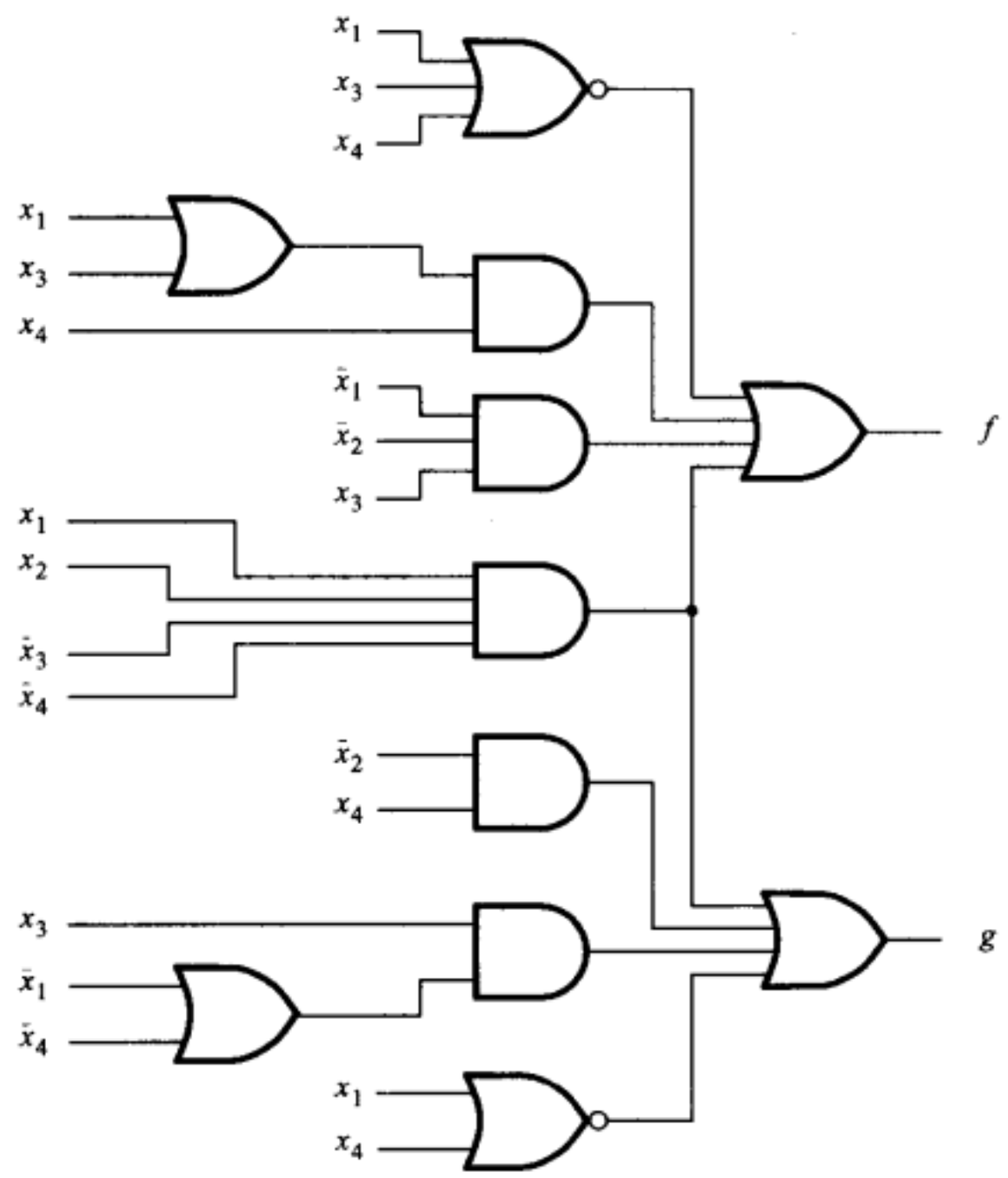
4.34 设电路图如P4-3所示，只允许用与非门实现，同上题的要求。

4.35 用Verilog的门级原语实现图4-25b中的电路。

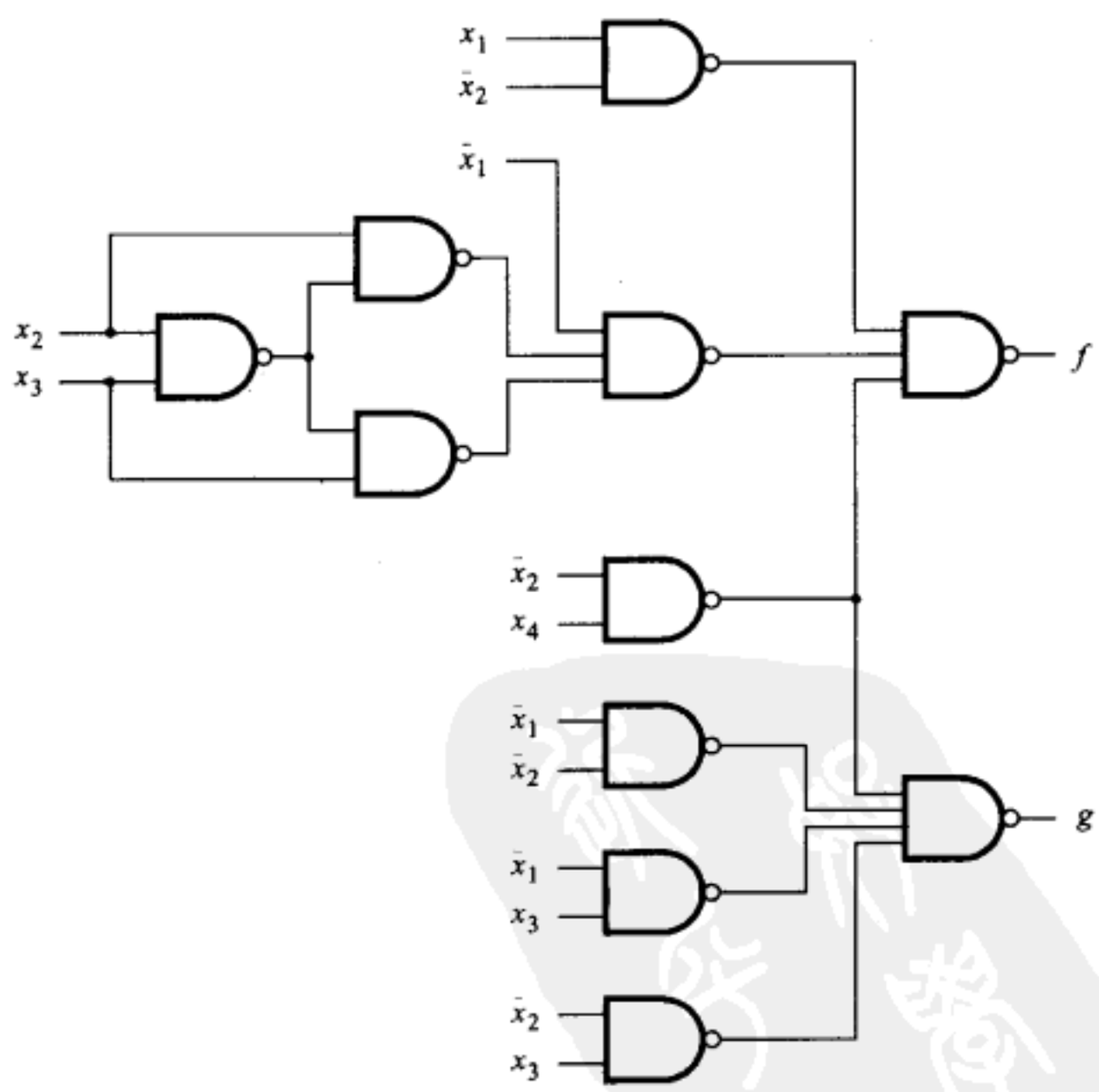
4.36 用Verilog的连续赋值实现图4-25b中的电路。

4.37 用Verilog的门级原语实现图4-27c中的电路。

4.38 用Verilog的连续赋值实现图4-27c中的电路。



图P4-2 习题4.33的电路



图P4-3 习题4.34的电路

4.39 用Verilog的门级原语实现图4-28b中的电路。

4.40 用Verilog的连续赋值实现图4-28b中的电路。

4.41 用Verilog的门级原语实现函数 $f(x_1,\cdots,x_4)=\sum m(0,1,2,4,5,7,8,9,11,12,14,15)$，使电路尽量简单。

4.42 用Verilog的连续赋值语句实现函数：

$$f(x_1,\cdots,x_4)=\sum m(0,1,2,4,5,7,8,9,11,12,14,15)$$

4.43 设函数为：$f(x_1,\cdots,x_4)=\sum m(1,4,7,14,15)+D(0,5,9)$，同习题4.41的要求。

4.44 用Verilog的连续赋值语句实现函数：$f(x_1,\cdots,x_4)=\sum m(1,4,7,14,15)+D(0,5,9)$。

4.45 用Verilog的门级原语实现函数：$f(x_1,\cdots,x_4)=\prod M(6,8,9,12,13)$，并使电路尽量简单。

4.46 用Verilog的连续赋值语句实现函数：$f(x_1,\cdots,x_4)=\prod M(6,8,9,12,13)$。

4.47 设函数为：$f(x_1,\cdots,x_4)=\prod M(3,11,14)+D(0,2,10,12)$，同习题4.45的要求。

4.48 用Verilog的连续赋值语句实现函数：$f(x_1,\cdots,x_4)=\prod M(3,11,14)+D(0,2,10,12)$。

参考文献

1. M. Karnaugh, "A Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of AIEE, Communications and Electronics* 72, part 1, November 1953, pp. 593–599.
2. R. L. Ashenhurst, "The Decomposition of Switching Functions," Proc. of the Symposium on the Theory of Switching, 1957, *Vol. 29 of Annals of Computation Laboratory* (Harvard University: Cambridge, MA, 1959), pp. 74–116.
3. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).
4. T. Sasao, *Logic Synthesis and Optimization* (Kluwer: Boston, MA, 1993).
5. S. Devadas, A. Gosh, and K. Keutzer, *Logic Synthesis* (McGraw-Hill: New York, 1994).
6. W. V. Quine, "The Problem of Simplifying Truth Functions," *Amer. Math. Monthly* 59 (1952), pp. 521–531.
7. E. J. McCluskey Jr., "Minimization of Boolean Functions," *Bell System Tech. Journal*, November 1956, pp. 1417–1444.
8. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
10. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
11. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
12. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
13. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
14. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
15. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
16. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
17. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
18. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

19. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer: Boston, MA, 1984).
20. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Synthesis Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, November 1987, pp. 1062–81.
21. E. M. Sentovic, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.
22. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill: New York, 1994).
23. N. Sherwani, *Algorithms for VLSI Physical Design Automation* (Kluwer: Boston, MA, 1995).
24. B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings: Redwood City, CA, 1988).

第5章　数的表示和算术电路

本章我们将学习进行算术运算的逻辑电路。我们将阐述如何对数进行加、减、乘的运算，以及如何用Verilog代码来描述相应的算术电路。这些电路为Verilog提供了极好的平台，以展示其在描述复杂逻辑电路组合中所具备的强大功能和灵活性。设计这些算术电路所涉及的概念也可以很容易地应用在其他各类电路中。

在着手处理算术电路设计问题之前，有必要讨论一下在数字系统中数是如何表示的。在前面几章中，我们处理的是用来表示开关状态或者某些一般条件的逻辑变量。现在我们用这些逻辑变量来表示数。指定一个数需要几个变量，每个变量对应着该数的一个数位。

5.1　数位的表示法

在处理数和进行算术运算时，使用标准符号是很方便的。所以我们用加号“+”表示加运算，用减号“-”表示减运算。在前面几章中，我们用“+”来表示逻辑或，用“-”来表示从集合中减去一个元素。本章中尽管我们会用相同的符号来表示不同的操作，但每个符号的具体意义还是可以根据上下文的含义很清楚地判断出来。在可能出现误解的地方，都给出了明确的说明。

5.1.1　无符号整数

最简单的数是整数。我们的讨论将从正整数开始，然后扩展到负整数。只有正值的数称为*无符号数*，可以有负值的数称为*有符号数*。带小数点的数，即实数的表示将在本章后面讨论。

在我们熟悉的十进制系统中，数由数字（位）组成，每个数位有十个可能的值（从0到9中的一个），每个数字（位）都表示10的幂的倍数。例如，数 8547可以表示为：$8\times10^3+5\times10^2+4\times10^1+7\times10^0$。我们通常不把数写成10的幂的形式，因为数字（位）所在的位置已经暗示了这一点。一般情况下，十进制整数可以用n个十进制数字（位）来表示：

$$D=d_{n-1}d_{n-2}\cdots d_1d_0$$

上式代表了值：

$$V(D)=d_{n-1}\times10^{n-1}+d_{n-2}\times10^{n-2}+\cdots+d_1\times10^1+d_0\times10^0$$

上式称为“*数字（位）表示法*”。

因为每一位都有十个可能的值，并且每一位的权重都是10的幂，所以我们说的十进制数是*基10*或者*基数为10*的数。十进制数大家都很熟悉，既方便又易于理解，但是要在数字电路中用数字量来代表十个数就不太好办了。在数字系统中我们使用的是只由0和1两个数字构成的二进制数，或者叫做*基2*的数。每个二进制数位叫做一个*位*（bit）。在二进制系统中，仍可用数位表示法来表示二进制数，所以二进制数

$$B=b_{n-1}b_{n-2}\cdots b_1b_0$$

就表示了一个具有以下值的整数：

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i \tag{5.1}$$

例如，二进制数1101的值是：

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

由于在不同的进制下特定数位基数的含义不同，因此我们在可能出现混淆的情况下，用下标来表示不同的进制。如用$(1101)_2$来表示二进制数1101。V的值可算得$V = 8 + 4 + 1 = 13$，因此：$(1101)_2 = (13)_{10}$。

请注意：整数的范围取决于表示该整数的二进制数所用的位数。譬如，四位二进制数所能代表的最大的数是$(1111)_2 = (15)_{10}$。一个比较大的数的例子是$(10110111)_2 = (183)_{10}$。总体上，$n$位二进制整数所表示的范围为$0 \sim 2^n - 1$。

在二进制数中，最右端的一位通常称为“最低有效位”（LSB），无符号数的最左端的一位具有最高的2的幂，通常称为“最高有效位”（MSB）。在数字系统中为方便起见通常把相邻的几个位看作一组，由四位组成的组叫做半字节（nibble），由八位组成的组叫做字节（byte）。

5.1.2 十进制数与二进制数之间的转换

使用式（5.1）就可以将二进制数转换为十进制数，然后用十进制算术运算来进行计算。然而，从十进制数转换成二进制数并不如此简单。可以用如下所示方法，将十进制数逐次除2来完成转换。假如有一个十进制数$D = d_{n-1}d_{n-2}\cdots d_1d_0$，其值为$V$，想要将其转换为二进制数$B = b_{n-1}b_{n-2}\cdots b_1b_0$。因此有：

$$V = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0$$

若将V除以2，得到：

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + b_{n-2} \times 2^{n-3} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

该整数的商是$b_{n-1} \times 2^{n-2} + b_{n-2} \times 2^{n-3} + \cdots + b_2 \times 2^1 + b_1$，余数是$b_0$。如果余数是0则$b_0 = 0$；如果余数是1则$b_0 = 1$。可以看到商是另一个二进制数，但只有$n-1$位而不是$n$位。再将此数除以2得到余数$b_1$，新的商是：

$$b_{n-1} \times 2^{n-3} + \cdots + b_2$$

如上步骤一直除下去，每一步决定一位，一直进行到商为0为止，就可以得到所有的位。图5-1以$(857)_{10} = (1101011001)_2$为例展示了转换的全过程。注意最先产生的是最低有效位（LSB），而最后产生的是最高有效位（MSB）。

$(857)_{10}$转化为二进制

			余数	
857 ÷ 2	=	428	1	LSB
428 ÷ 2	=	214	0	
214 ÷ 2	=	107	0	
107 ÷ 2	=	53	1	
53 ÷ 2	=	26	1	
26 ÷ 2	=	13	0	
13 ÷ 2	=	6	1	
6 ÷ 2	=	3	0	
3 ÷ 2	=	1	1	
1 ÷ 2	=	0	1	MSB

结果是$(1101011001)_2$

图5-1 十进制数转化为二进制数

5.1.3 八进制数和十六进制数的表示

数位表示法适用于所有进制的数。对于r进制（基数为r）的数：

$$K = k_{n-1}k_{n-2}\cdots k_1k_0$$

其值为：

$$V(K)=\sum_{i=0}^{n-1}k_i\times r^i$$

我们仅对有实用意义的数制感兴趣。我们用十进制数是因为人们广泛使用这种数；我们也将用二进制数，因为计算机必须使用二进制数。此外，还有两种基数数制也很有用，即八进制和十六进制。基为8的数叫做八进制（octal）数，基为16的数叫做十六进制（hexadecimal）数。八进制中每位数的范围是0到7，十六进制（通常简写为hex）中每位数可有16种取值，其中值相当于十进制数的10，11，12，13，14和15的数位分别用*A*，*B*，*C*，*D*，*E*和*F*来表示。表5-1给出了这些数制中的前18个整数。

表5-1 不同进制的数

十进制	二进制	八进制	十六进制	十进制	二进制	八进制	十六进制
00	0000	00	00	10	01010	12	0A
01	0001	01	01	11	01011	13	0B
02	00010	02	02	12	01100	14	0C
03	00011	03	03	13	01101	15	0D
04	00100	04	04	14	01110	16	0E
05	00101	05	05	15	01111	17	0F
06	00110	06	06	16	10000	20	10
07	00111	07	07	17	10001	21	11
08	01000	10	08	18	10010	22	12
09	01001	11	09				

计算机中所使用的数制主要是二进制。而使用八进制和十六进制的原因是它们可以简化二进制数的表示。一个八进制数位代表三个位（比特），因此要将一个二进制数转化为八进制数，可以把每三个位（比特）划为一组，每组使用相应的八进制数字（位）代替。例如，101011010111可作如下转换：

$$\underbrace{101}_{5}\ \underbrace{011}_{3}\ \underbrace{010}_{2}\ \underbrace{111}_{7}$$

这意味着$(101011010111)_2=(5327)_8$。如果一个二进制数的位数不是三的倍数，那么可以给它的最高位补零。例如，$(10111011)_2=(273)_8$，因为

$$\underbrace{010}_{2}\ \underbrace{111}_{7}\ \underbrace{011}_{3}$$

从八进制转换到二进制同样简单；将每一个八进制数字（位）用相应的三个二进制位（比特）代替即可。

同样，一个十六进制数字（位）也可以用四个位来代替。例如，一个16位的数可以表示为一个四位的十六进制数：$(1010111100100101)_2=(AF25)_{16}$。因为

$$\underbrace{1010}_{A}\ \underbrace{1111}_{F}\ \underbrace{0010}_{2}\ \underbrace{0101}_{5}$$

若二进制数的位数不是四的整数倍，则可以在最高有效位的左侧添零，如$(1101101000)_2=(368)_{16}$，因为

$$\underbrace{0011}_{3}\ \underbrace{0110}_{6}\ \underbrace{1000}_{8}$$

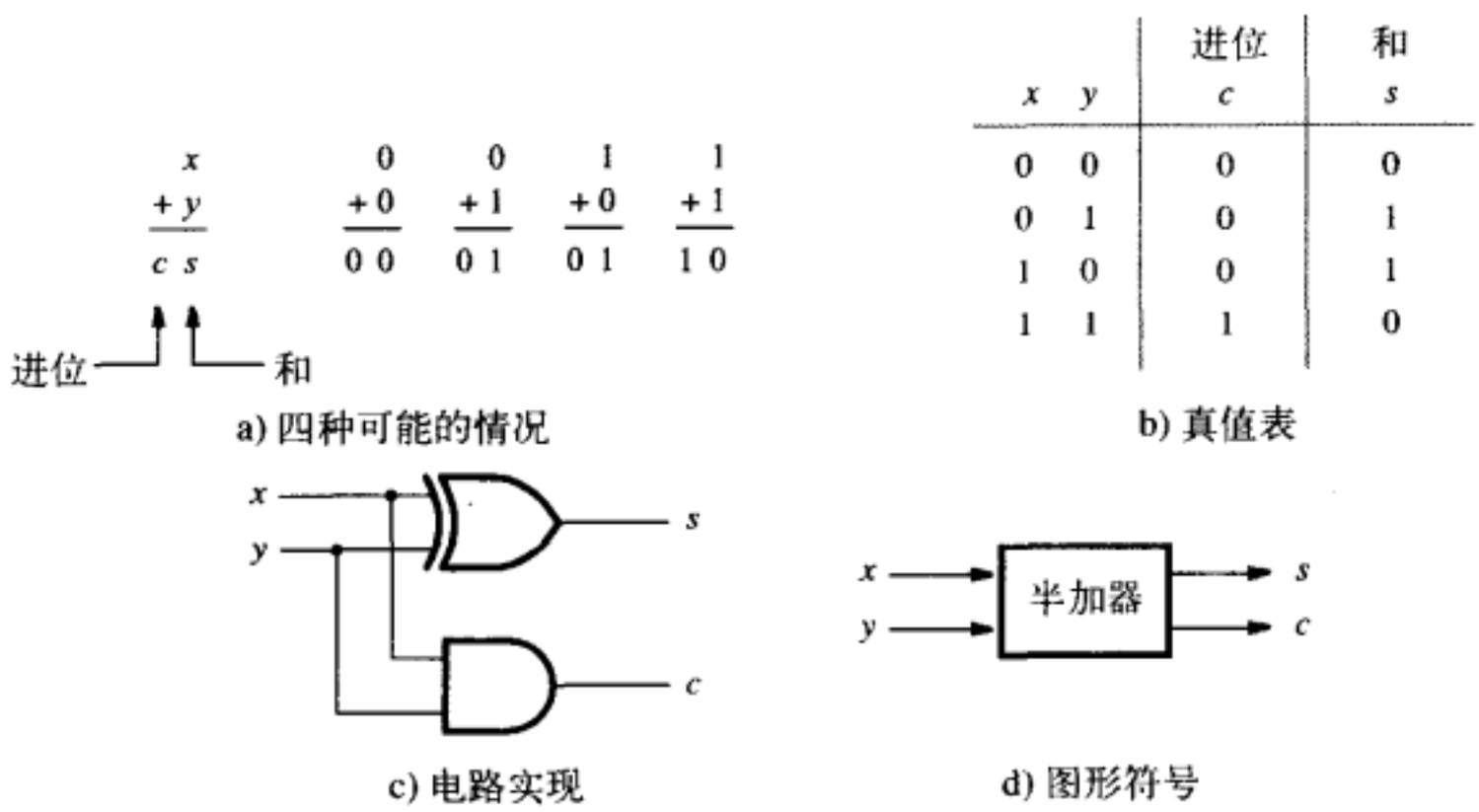
对于从十六进制到二进制的转换，可以简单地将每个十六进制数字（位）用四个表示相同数值的位来代替。

现代计算机用的二进制数通常有32或64位。可以写成 n 元组（有时叫做位向量）。这样表示不便于人们处理，而写成八进制或十六进制的形式就方便多了。由于一个数字系统在进行算术运算时经常涉及二进制数，因此我们将重点关注使用这些数的电路。为了方便起见，有时我们将用十六进制数来表示数。

上面已经介绍了最简单的数，即无符号整数，我们有必要能够处理其他类型的数。在本章后面将开始讨论有符号数、定点数和浮点数。但首先我们来考察一些能进行数值运算的简单电路，使读者体会到数字电路可以完成算术运算，以推动更进一步的讨论。

5.2 无符号数的加法

二进制加法与十进制加法的操作是一样的，不同的是它只有0和1两个数值。如图5-2a所示两个1位（比特）的数相加共有四种情况，最终得到的和需要用两位（比特）来表示，最右侧的位（比特）叫做“和”（sum, s），最左侧的位（比特）当两个加数为1时等于1，叫做“进位”（carry, c）。所有加运算可能出现的情况以真值表的形式在图5-2b中列出。表示和的位s是以x和 y为输入的异或（XOR）函数，这我们已经在3.9.1节介绍过；而表示进位的位c是以x和y为输入的与（AND）函数，实现这两个函数的电路如图5-2c所示。只对两个位（比特）进行加运算的电路叫做“半加器”。



x	y	进位 c	和 s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

b) 真值表

图5-2 半加器

当多位（比特）数相加时，我们可以看到更有意思的情况。在这种情况下，需要对每相应位的位（比特）各自做加法，但在每个相应位i上，加法运算还包括了从 $i-1$ 位到i位的进位。

图5-3所示为加运算的一个例子。操作数 $X=(01111)_2=(15)_{10}$ 和 $Y=(01010)_2=(10)_{10}$。注意到X 和Y 都有五个位（比特）。用五个位就可以表示0到31之间的数；所以所得的和 $S=X+Y=(25)_{10}$ 可以表示成五个位的二进制整数。注意每个位（比特）的下标，也就是 $X=x_4x_3x_2x_1x_0$ 和 $Y=y_4y_3y_2y_1y_0$。这幅图展示了在加运算的过程中进位的产生。例如，x_0和 y_0相加后进位为0，而x_1和y_1相加后进位为1，等等。

$X = x_4x_3x_2x_1x_0$	01111	$(15)_{10}$
$+Y = y_4y_3y_2y_1y_0$	01010	$(10)_{10}$
	1110 ← 产生的进位	
$S = s_4s_3s_2s_1s_0$	11001	$(25)_{10}$

图5-3 一个加法运算的例子

在第2和第4章中，我们曾先写出真值表指定电路的行为，再来设计逻辑电路。但这种方法在设计如图5-3所示的能进行两个五位数加法的运算电路时就不实用了，因为X 和 Y 每个都

有5个变量，相应的真值表总共需要10个输入变量，因此总共就会有$2^{10}=1024$ 种组合！更好的解决方案是分别考虑每一对 x_i 和 y_i 的加法运算。

处在0位的位（比特）由于没有进位，因此其加法电路与图5-2所示的相同。对于处在其他 i 位的位（比特），它们的加法电路的输入包括x_i，y_i 以及来自低位的进位c_i。图5-4a列出了输入变量为x_i，y_i，c_i，输出为和与进位项的函数的真值表。和位（比特）s_i 是 x_i，y_i，c_i 的模二加求和。而进位位（比特）c_{i+1} 在x_i，y_i，c_i的和为2或3时等于1。这些函数的卡诺图如图5-4b所示。进位函数的最简积之和实现表达式为：

$$c_{i+1}=x_iy_i+x_ic_i+y_ic_i$$

s_i 函数的最简积之和实现表达式为：

$$s_i=\bar{x}_iy_i\bar{c}_i+x_i\bar{y}_i\bar{c}_i+\bar{x}_i\bar{y}_ic_i+x_iy_ic_i$$

下面我们将看到，这个函数可以用异或门（XOR）写成更简洁的形式。

使用异或门

两个变量进行异或运算的定义是：$x_1\oplus x_2=\bar{x}_1x_2+x_1\bar{x}_2$。上面的和位（比特）表达式可以写成如下的只用异或操作符表示的形式：

$$\begin{aligned}s_i&=(\bar{x}_iy_i+x_i\bar{y}_i)\bar{c}_i+(\bar{x}_i\bar{y}_i+x_iy_i)c_i\\&=(x_i\oplus y_i)\bar{c}_i+(\overline{x_i\oplus y_i})c_i\\&=(x_i\oplus y_i)\oplus c_i\end{aligned}$$

异或运算满足结合律，所以可以写成：

$$s_i=x_i\oplus y_i\oplus c_i$$

因此一个三输入异或门就可以用来实现s_i。

异或门的输出是其输入变量的模二加求和。若输入为奇数个1，则输出为1；其他情况下，输出为0。因此异或运算有时也被称为奇函数（odd function）。从图5-4b所示的s_i 的卡诺图，可以清楚地看到异或函数不能合并成更大乘积项。实现图5-4a真值表的逻辑电路如图5-4c所示。该电路就是全加器。

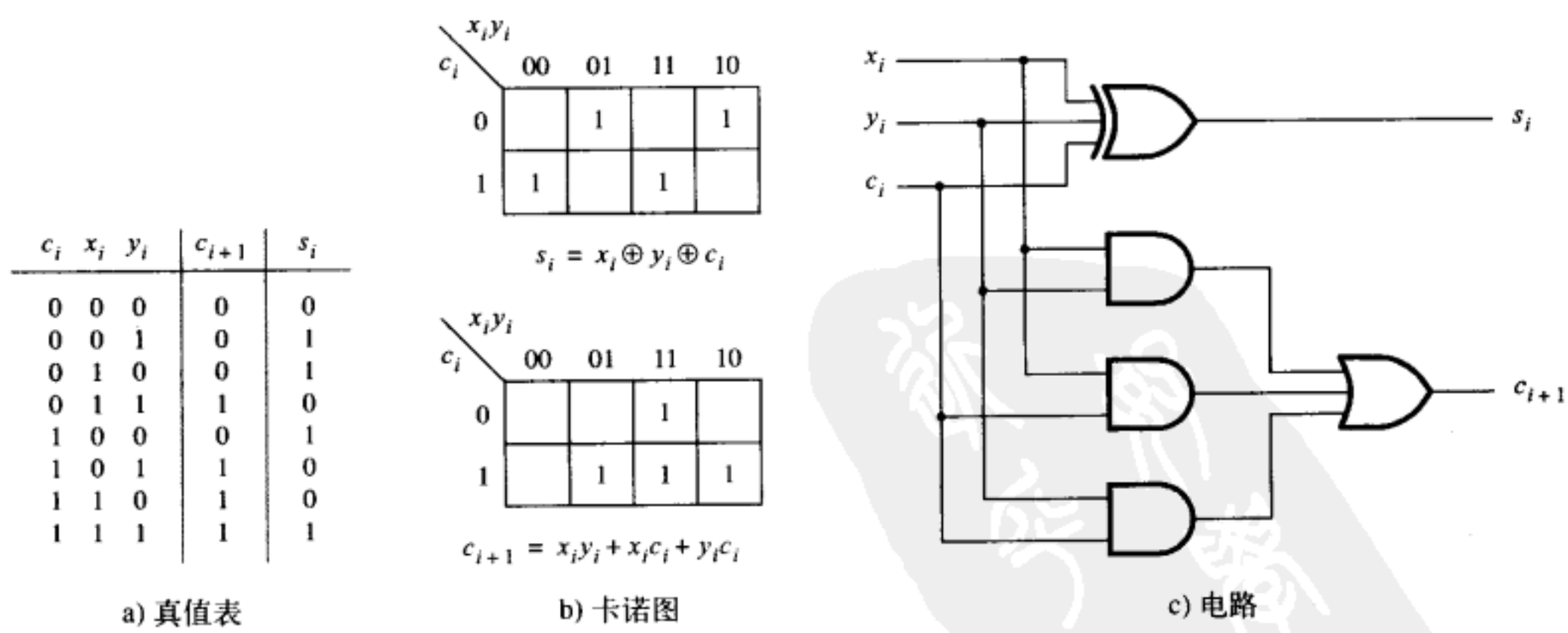


图5-4 全加器

异或门还有一个有趣的性质：可以把两个输入端异或门的其中一个输入作为控制信号来决定输出的究竟是另一个输入的原信号还是取其反的信号。可以从异或门的定义

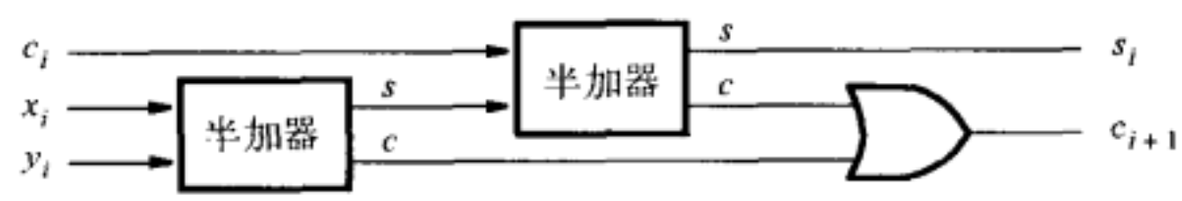
$x_i \oplus y_i = \bar{x}y + x\bar{y}$ 中清楚地看到这一点。比如将x看做控制信号，若 $x = 0$，则输出就等于 y 的值；而若 $x = 1$，则输出等于y 的非。上面我们通过代数推导得到逻辑表达式 $s_i = (x_i \oplus y_i) \oplus c_i$。通过下面的观察我们也可以很快地得到这个表达式。在图5-4a真值表的上半部分$c_i = 0$，和函数s_i就是x_i与 y_i的异或；下半部分$c_i = 1$，s_i 就是上半部分的非。观察到这一点，我们就可以直接写出上述表达式（$s_i = (x_i \oplus y_i) \oplus c_i$）。在5.3.3节中，我们将举一个重要的例子，说明如何使用异或门将一个输入信号作为控制信号，来传输另一个输入的原信号还是反信号。

在上面的讨论中遇到了异或运算的求反操作，记为 $\overline{x_i \oplus y_i}$。这种操作十分常见，通常称为同或（XNOR）运算，用特定的符号⊙来标记这种运算，由此可记作：$x \odot y = \overline{x \oplus y}$。

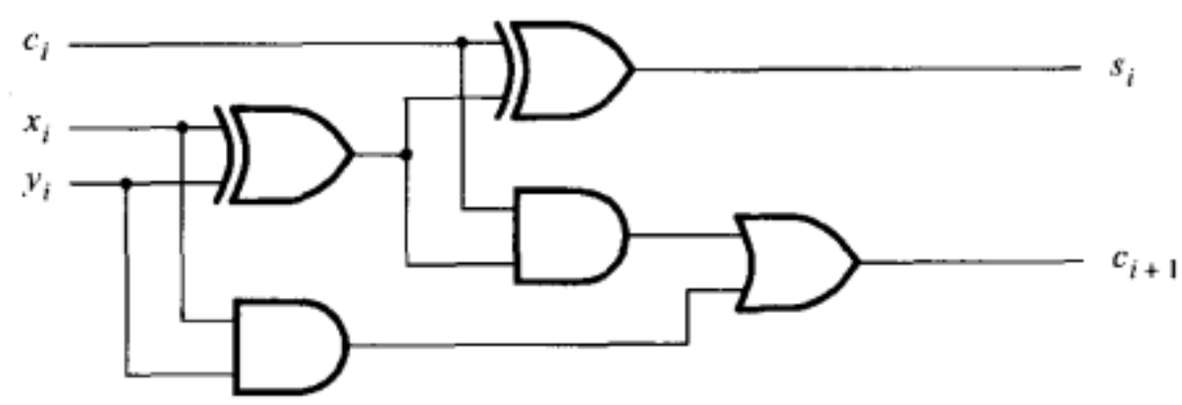
同或运算有时称为一致运算（coincidence operation）。因为当两个输入的值一致时，也就是同为0或者同为1时，输出为1。

5.2.1 全加器的分解

看到全加器电路这个名字，人们就会想到也许可以用半加器来组成全加器。用4.6.2节介绍过的多级电路可以实现全加器。如图5-5所示的电路，该电路用两个半加器组成一个全加器。读者可以自己验证这个电路功能的正确性。



a) 块图



b) 细节图

图5-5 全加器电路的分步实现

5.2.2 行波进位加法器

用手做加法时，我们必须从最低有效位开始把相应位的两个数字相加，一直加到最高有效位。若在第 i 位出现了进位，则这个进位将作为第 $i+1$ 位的加数。在逻辑电路中进行加法运算时也可以采用相同的办法。每一位用全加器电路表示，如图5-6所示，将它们连接起来。注意为了习惯起见最低有效位全加器画在最右侧，产生的进位从全加器的左侧输出，从右侧进入较高一位全加器的进位输入。

若X和Y作为加法器的输入，在求得稳定的加法运算的和值S之前，需要一段时间。每一个全加器在输出其有效的s_i 和c_{i+1}值之前都需要一定的延迟时间，设它是Δt，所以第一级全加器的进位c_1在执行输入x_0和 y_0求和的Δt时间后才能到达第二级全加器；第二级全加器的进位 c_2 经过$2\Delta t$时间的延迟后才能到达第三级全加器，依此类推。信号c_{n-1}要经过$(n-1)\Delta t$时间的延迟后才能得到稳定的和值。这就意味着最终的和值必须等待$n\Delta t$时间后才能得到。由于进位信号像波浪一样在全加器队列中传播，因此图5-6的电路称为行波进位加法器（ripple-carry adder）。

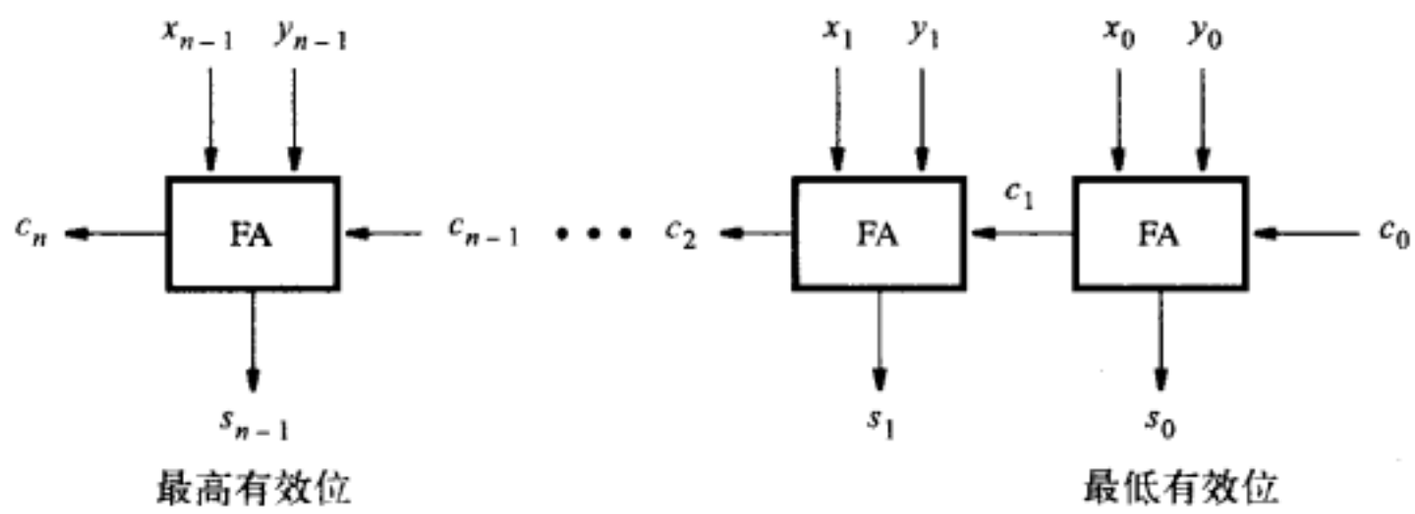


图5-6 一个n位的行波进位加法器

用行波进位加法器求得和值与进位输出必须的延迟时间取决于加数和被加数的大小。当加数和被加数为32位或64位数时，延迟就会长得不可接受。因为全加器电路的结构已经没有大幅度缩短信号延迟的空间，所以必须寻找 n 位加法器的新结构。在5.4节中，我们将讨论构建高速加法器的技术。

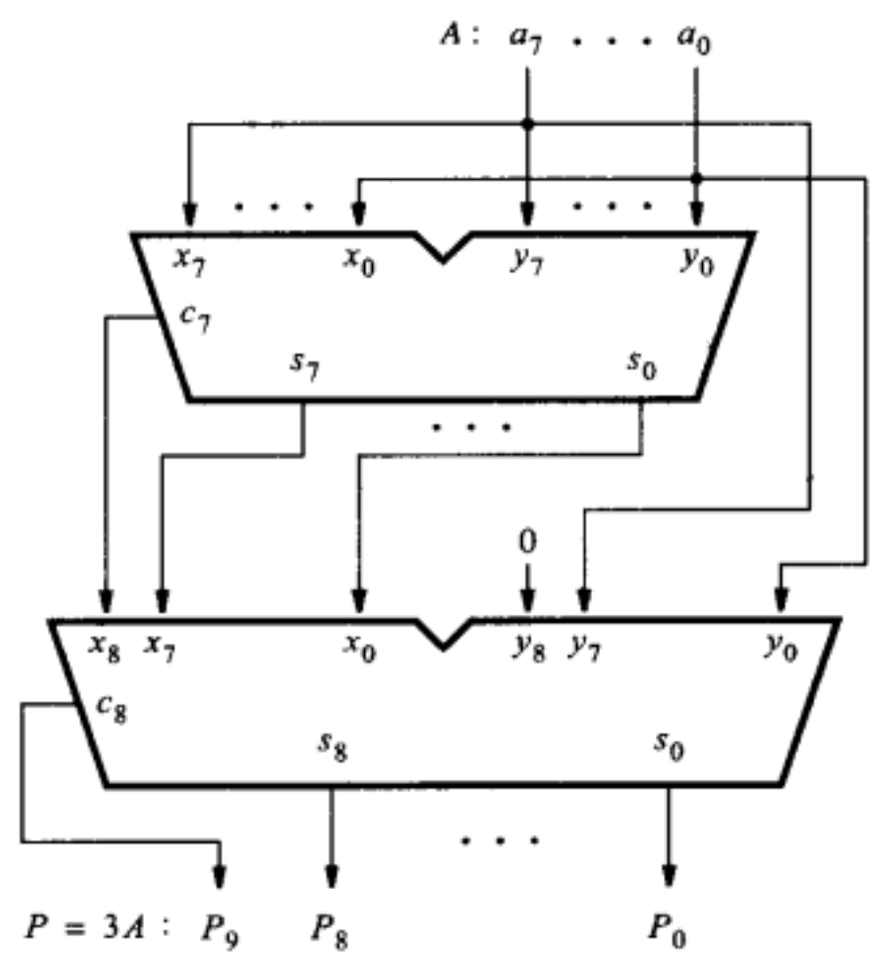
至此我们只讨论了无符号整数。无符号整数的求和运算在加法器的零级不需要进位。而在图5-6中，加法器的第零级有进位信号c_0，这样的行波进位加法器可以用来做减法，我们将在5.3节中对其进行详细的讨论。

5.2.3 设计举例

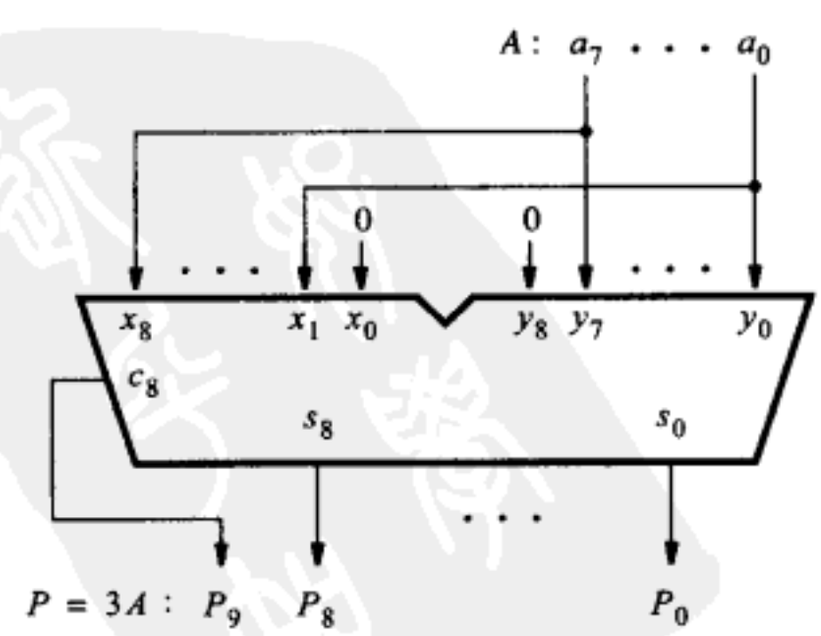
假设我们需要设计一个能进行8位无符号数与3相乘的乘法器电路。令这个数为$A = a_7a_6 \cdots a_1a_0$，令最终的乘积 $P = p_9p_8 \cdots p_1p_0$，即$P = 3A$。注意乘积要用10位数表示。

设计该电路的简单方法是用两个行波进位加法器把A加三次，如图5-7a所示。图中代表加法器的符号是通用的表示加法器的图形符号。根据图5-6，用字母x_i，y_i，s_i 和c_i 表示输入和输出。第一个加法器计算 $A+A = 2A$。它产生的计算结果是 8 位的和值以及从最高位的进位。第二个加法器计算 $2A+A = 3A$。由于第一个加法器产生的 $2A$ 共有9位，因此第二个加法器必须能处理由第一个加法器产生的9位数据。而第二个加法器的 y_i 输入是由只有8位的A信号驱动的，所以应把第九个输入端 y_8 接0。

这种方法直接了当，但效率太低。我们已经注意到，要得到$2A$可以将A的各位均向左移动1位，得到 $a_7a_6 \cdots a_1a_0 0$。根据式（5.1），这种形式就等于$2A$。因为$3A = 2A + A$，所以计算$3A$其实只用一个行波进位加法器就已足够，见图5-7b所示。该加法器与图5-7a的第二个加法器本质上具有相同的电路结构。请注意，输入x_0连接的是常数0，而图5-7a 第二个加法器x_0的值也总是0，尽管它连接着第一个加法器的最低有效位s_0。因为在第一个加法



a) 不成熟的设计



b) 有效率的设计

图5-7 8位无符号数乘3的电路

器中$x_0 = y_0 = a_0$，和的最低位 s_0 将总是为0，不论a_0是0还是1。

5.3 有符号数

在十进制系统中，将“+”或“−”标在最高有效位的左侧来表示数的正负。而在二进制系统中，数的符号是由其最左侧的位来决定的。正数的最左侧位为0，负数为1。因此，有符号数的最左侧位代表了它的符号，而余下的n-1位代表了它的大小，如图5-8所示。注意到最高有效位位置的不同是很重要的。在无符号数中，所有的位都代表了数的大小；因此在定义数的大小时，所有n位都是有意义的。因此最高位（MSB）就是无符号整数的最左侧的位b_{n-1}。在有符号数中，有 $n-1$个有意义的位，最高位是 b_{n-2}。

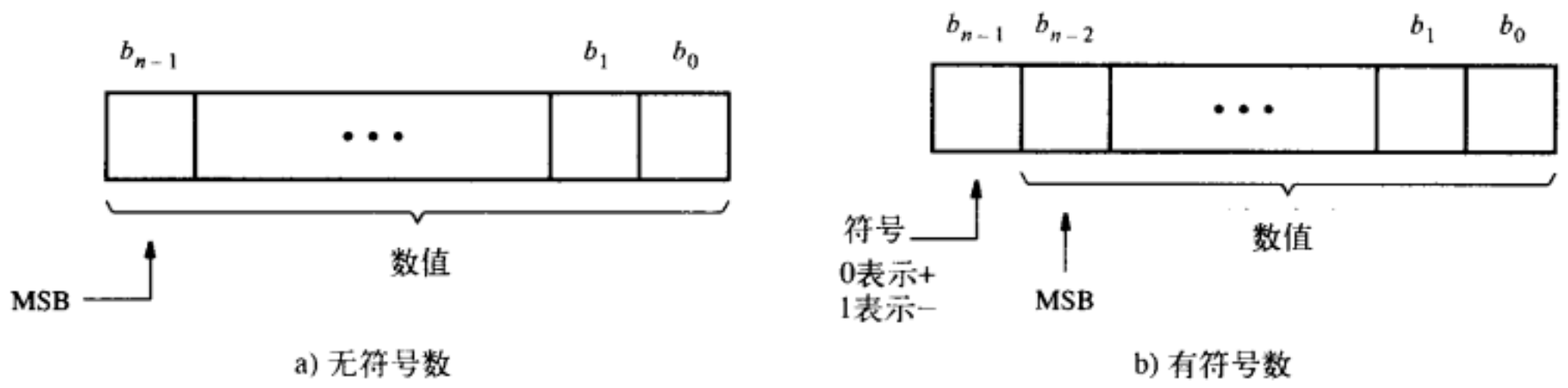


图5-8 表示整数的格式

5.3.1 负数

上一节解释了正整数的表示方法。负数可以用三种不同的方式来表达：符号−数值，1的补码和2的补码。

符号−数值的表示法

在我们熟悉的十进制表示法中，正数与负数的数值是用相同的方法来表示的，而用专门的正负号来区分数的正负。这种方式叫做数的“符号−数值”表示法。该方法同样适用于二进制，只是分别用符号位为“0”或者为“1”来表示数的正、负。例如，四位二进制数±5，可以分别用 +5 = 0101 和 −5 = 1101来表示。由于与用符号−数值表示法的十进制数十分相似，因此这种表示法很容易理解。然而，我们后面将会看到，这种表示方法并不十分适用于计算机的应用。更为合适的表示法是基于补码的表示法，下面我们接着讲解这种补码数制。

1的补码表示法

在补码数制中，负数是按照对应正数的减法操作定义的。我们将考虑两种关于二进制数的补码方案：1的补码和2的补码。在1的补码方案中，一个n位负数K是通过2^n-1减去与K绝对值相等的正数P得到的，即$K = (2^n - 1) - P$。例如，若$n = 4$，则 $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$。如果我们想把+5转换成负数，有$-5 = 1111 - 0101 = 1010$。类似地，$+3 = 0011$，$-3 = 1111 - 0011 = 1100$。很明显，“1的补码”可以简单地将数的每一位（包括符号位）取反即可得到。尽管1的补码比较容易得到，但是在下一节中，我们将看到它在算术操作中还存在着一些缺陷。

2的补码表示法

在2的补码表示法中，负数K是通过 2^n 减去与K绝对值相等的正数P得到的，即$K = 2^n - P$。我们还是用四位数的例子来加以说明：$-5 = 10000 - 0101 = 1011$，$-3 = 10000 - 0011 = 1101$。要得到2的补码，要求进行一次有进位的减法的运算。然而我们发现，如果K_1表示P的1的补码，K_2表示P的2的补码，即，

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

有$K_2 = K_1 + 1$。因此更简单的求某个数的2的补码的方法是先求出该数的1的补码，然后再加1，而求某数的1的补码是很容易的。以上就是算术运算逻辑电路中求解2的补码的方法。

为了让读者能够很快地求出某个数的2的补码，下面有一个简单的规则。

求解2的补码的规则 给定一个有符号数 $B = b_{n-1}b_{n-2}\cdots b_1b_0$，求$B$的2的补码 $K = k_{n-1}k_{n-2}\cdots k_1k_0$的方法是：从右到左检查$B$的每一位，保留$B$为0的位以及遇到第一个是1的位，然后将其余各位取反即可。

例如，若$B=0110$，那么保留$k_0 = b_0 = 0$和$k_1 = b_1 = 1$，再将其余位求反即 $k_2 = b_2 = 0$ 和 $k_3 = b_3 = 1$。求得$K=1010$。再如，$B=10110100$，作为练习，读者可以自己算得$K=01001100$，以验证此规则的正确性。

表5-2列出了在三种不同的有符号数表示法下，四位数的所有16种组合的解释。请注意：在“符号加数值”和“1的补码”两种表示法中都有两个0存在，而在“2的补码”表示法中只有一个0。也请注意，“2的补码”表示法能够表示的数的范围是−8到+7，而其他两种表示法所表示的范围都是 −7 到 +7。

表5-2 四位有符号数的表示

$b_3b_2b_1b_0$	符号加数值	1的补码	2的补码	$b_3b_2b_1b_0$	符号加数值	1的补码	2的补码
0111	+7	+7	+7	1000	−0	−7	−8
0110	+6	+6	+6	1001	−1	−6	−7
0101	+5	+5	+5	1010	−2	−5	−6
0100	+4	+4	+4	1011	−3	−4	−5
0011	+3	+3	+3	1100	−4	−3	−4
0010	+2	+2	+2	1101	−5	−2	−3
0001	+1	+1	+1	1110	−6	−1	−2
0000	+0	+0	+0	1111	−7	−0	−1

若n位数 $B = b_{n-1}b_{n-2}\cdots b_1b_0$ 是用2的补码表示法表示的，则它的值是

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \tag{5.2}$$

所以最大的负数100…00的值是-2^{n-1}，最大的正数011…11的值是$2^{n-1}-1$。

5.3.2 加法和减法

为了评价数的不同表示法的适用性，有必要研究一下它们在算术运算中的应用，尤其是在加法和减法中的应用。我们可以通过一些比较小的数来展示不同表示法各自的优缺点。下面我们将使用由一个符号位和三个数值位组成的四位数来验证。这些数必须足够小，以致它们和的数值才能用3位数来表示，也就是说它们的和的数值不能大于7。

计算正数加法使用三种表示法都一样，实际上与我们在5.2节讨论过的无符号数的加法是相同的。但是对于负数来说却有很显著的不同，如果两个加数的符号不同的话困难就产生了。

符号—数值表示法的加法运算

若两个相加的数符号相同，则两个用符号加数值法表示的数的相加是很简单的：大小相加，符号不变。但是如果两个加数异号，情况就复杂了一些。这就需要从大数里减去小数，意味着逻辑电路必须要有比较和减法的操作。我们很快将会看到不使用这种电路也可以完成

减法运算。因此在计算机中并不使用符号加数值表示法。

用1的补码做加法运算

1的补码的明显优点是只要将其相应正数的每一位取反，便可以产生负数。图5-9展示了两个数相加的情况。两个数相加共有四种不同的符号组合。从图的上半部分来看计算 5+2 = 7 和−5 + 2 = −3十分简单；将两个加数简单相加即可。但计算其他两种组合就不那么简单了。计算 5 + (−2) = 3 产生位向量10010。由于我们处理的是四位数，从符号位会有进位，而且最后得到2而不是正确的结果3。有趣的是，如果我们将符号位的进位拿掉并将其加到结果的最低有效位上，得到的就是正确结果3。纠正的过程在图中已标出。当计算(−5) + (−2) = −7时遇到的是同样的问题，最先得到的是错误结果0111，表示的是7而不是−7，我们使用同样的纠正方法将符号位的进位拿掉加到其最低有效位（LSB）上，如图5-9所示。

由这些例子得到的结论是使用1的补码有时很方便，但有时不方便。若遇到需要纠正的情况，则必须要再做一次加法。因此，两个1的补码数相加所花费的时间也许是两个无符号数相加所花费时间的两倍。

用2的补码做加法运算

使用在1的补码加法的例子中所用的相同的加数组合，图5-10展示了如何做2的补码的加法。5 + 2 = 7 和 −5 + 2 = −3很简单。5 + (−2) = 3 也得到了正确的结果 0011。从符号位产生的进位我们尽可以忽略。第四种情况(−5) + (−2) = (−7) 同样得到了正确的结果1001，即−7。同样我们忽略它符号位的进位。

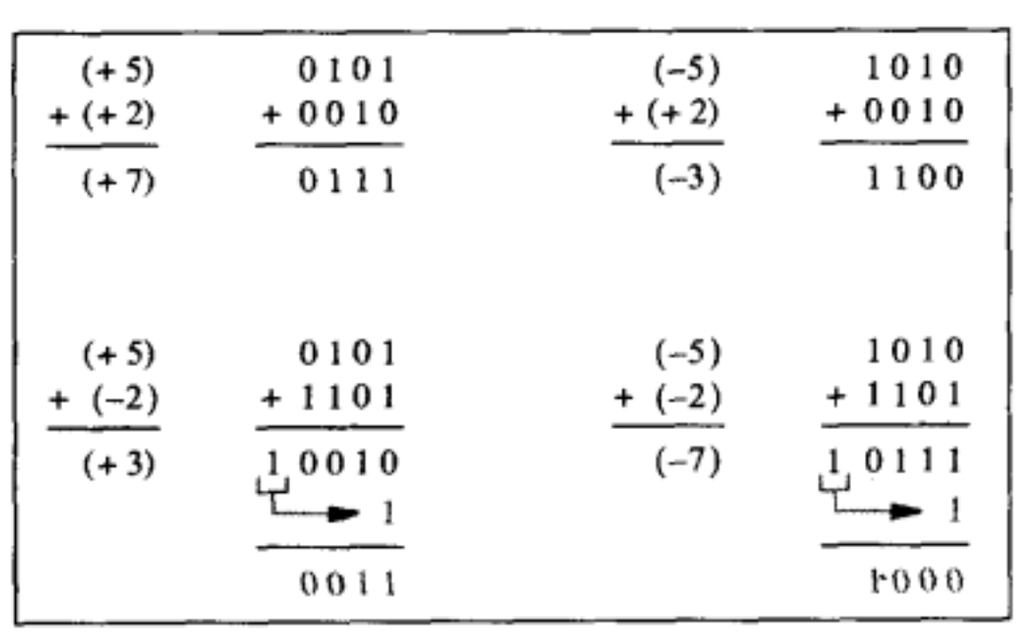


图5-9 1的补码做加法举例

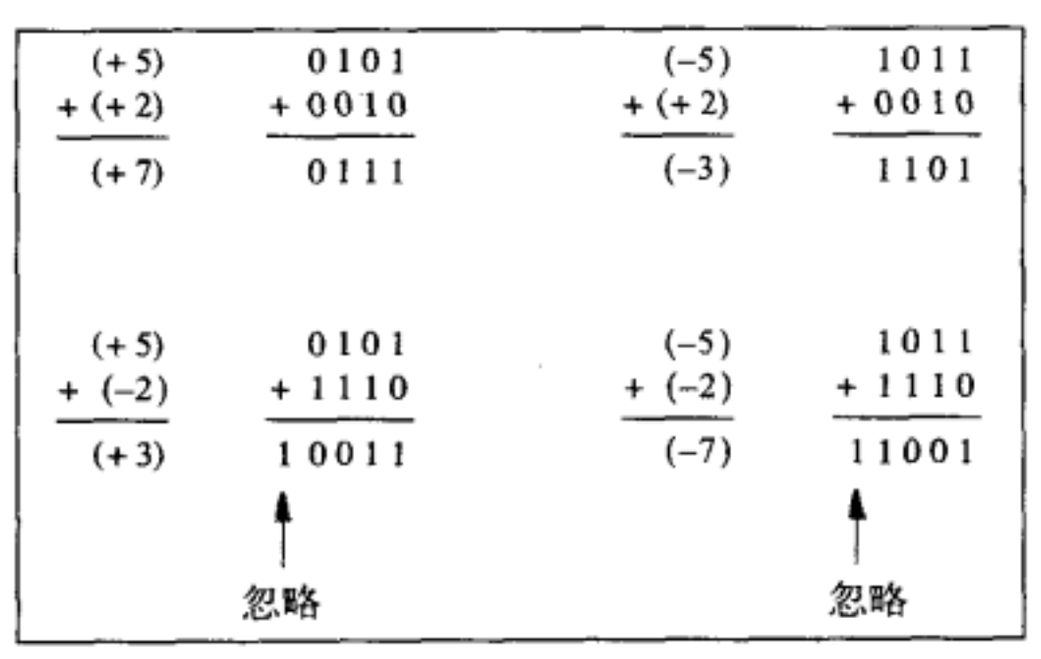


图5-10 2的补码做加法举例

由这些例子可以看出，2的补码的加法运算十分简单。当数相加的时候，结果总是正确的。如果有从符号位的进位，直接将它忽略就可以了。所以，不论操作数的符号是什么，加法运算的过程都是一样的。它可以由一个加法器电路来实现，就像我们在图5-6中展示的那样。所以 2 的补码数制是非常适合于加法电路应用的。下面我们将考虑它在减法操作中的应用。

用2的补码做减法运算

进行减法运算最简单的方法是将减数取反再加到被减数上。如图5-11所示，我们可以先求出减数的2的补码再进行加法运算。要计算5 − (+2) = 3，先找到减数2的补码1110，再加上0101，结果是0011 = +3（忽略符号位的进位）。计算(−5) − (+2) = (−7) 也是如此。在剩余的两种情况中没有进位出现，结果是准确的。

为了形象地演示图5-10和图5-11中进行加减法运算的过程。我们可以把所有的四位数，如图5-12所示，放到一个模十六的圆上。如果这些位代表的是无符号数，它们就是0到15；而如果是2的补码，则它们的范围就是−8到 +7。如果要做加法就按顺时针方向增加被加数的幅值。

例如，要求 −5 + 2就从1011（−5）开始顺时针移动两步，得到结果1101(−3)。要做减法就逆时针移动。例如，要求 −5 − (+2) 就从1011开始逆时针移动两步得到1001(−7)。

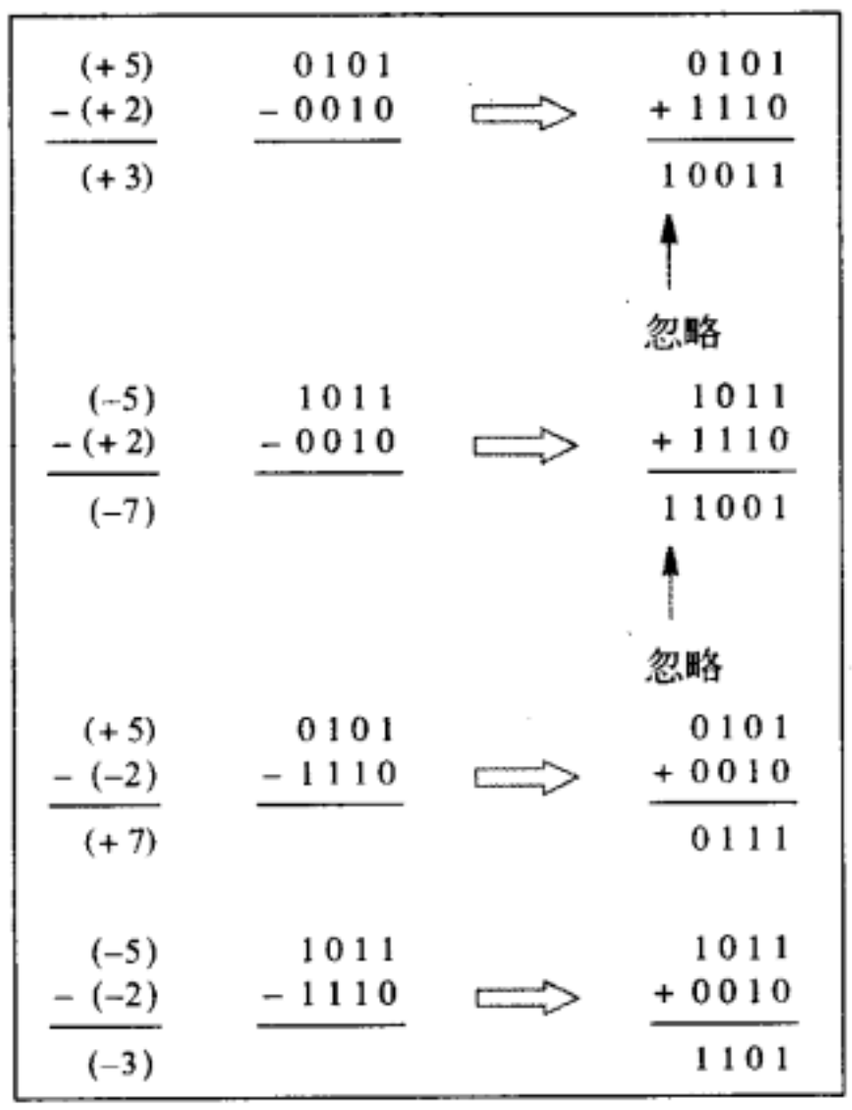


图5-11 用2的补码做减法举例

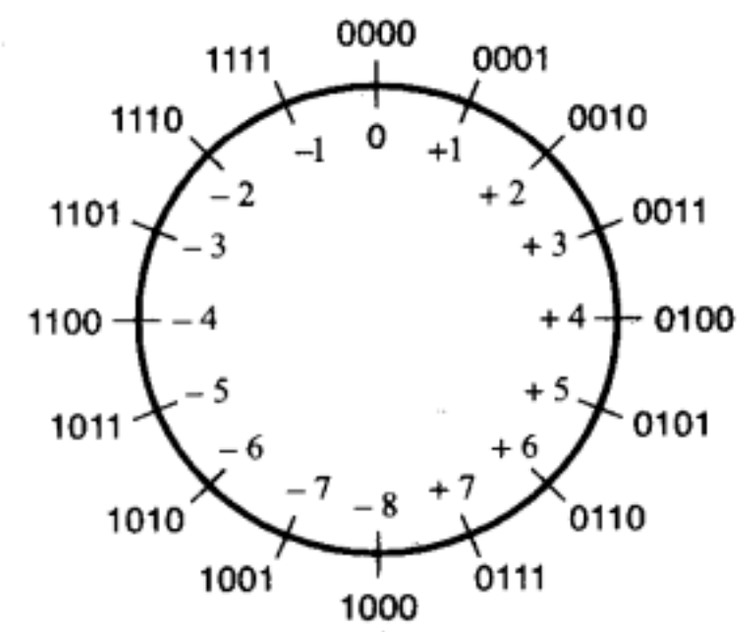


图5-12 四位2的补码数的图形解释

这一节的最主要的结论是减法也同样可以通过减数的2的补码用加法来实现，而不必考虑两个操作数的符号。因此，可以用同一个加法器电路来实现加法和减法。

5.3.3 加法器和减法器单元

加法和减法运算唯一的一点不同就是做减法时有必要用到一个操作数的2的补码。设X和Y是两个操作数，做减法时Y是减数。从5.3.1节我们知道2的补码可以从Y的1的补码加1得到。要在最小数位加1只需将进位比特c_0设为1即可。将Y各位取反得到Y的1的补码，用非门就可以实现，但我们需要一个更加灵活的电路，借助于这个电路，我们可以用Y的原值做加法，而用它的2的补码做减法。

在5.2节中我们解释了双输入的异或门可以用来实现将一个输入做控制信号，选择输出的信号是另一个输入的原值或者反相值。这种想法可以用在如下的加法/减法器单元的设计中。假设有一个控制信号来选择究竟进行加法还是减法运算，我们把它叫做 $\overline{\text{Add}}$/Sub，并规定它等于0时做加法，等于1时做减法。因此我们在Add上加一道横线表示加法。在信号名字上加一道横线是一种习惯记法，表示这个名字所代表的信号只在控制电平为0时有效。现在让Y的每一位依次接到每个异或门的一个输入端，而所有异或门的另一个输入端都接到 $\overline{\text{Add}}$/Sub，则当 $\overline{\text{Add}}$/Sub = 0时，异或门输出的就是Y的原值，当 $\overline{\text{Add}}$/Sub = 1时，异或门输出的就是Y的反相值（1的补码）。这样设计的加/减法器的电路如图5-13所示。该电路的主要部分是一个n位加法器，可以用图5-6的行波进位加法器来实现。注意控制端 $\overline{\text{Add}}$/Sub同时也连到加法器的最低位的进位输入端c_0。若这样连接，当$c_0 = 1$时将1加到Y的1的补码上，便可得到其2的补码，就可进行减法运算了。在进行加法运算时，我们有$c_i = 0$。

若想设计尽可能灵活的电路，使其在尽可能多的任务中发挥作用，则设计加法器/减法器两用部件所体现的概念是很重要的。采用上述方法实现该类型电路显著地减少了所用门的数目，大大降低了布线的复杂度。

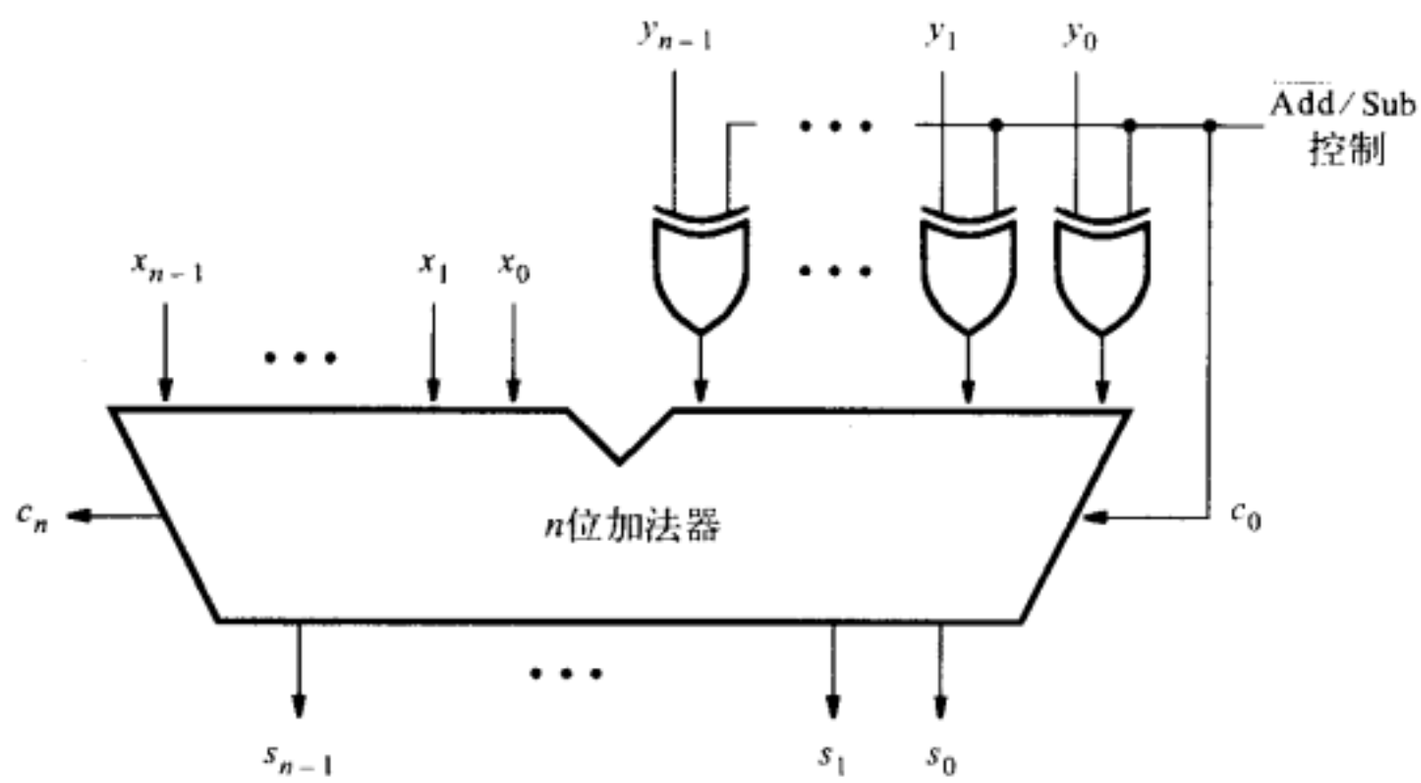


图5-13 加法器/减法器单元

5.3.4 基数补码方案

先求出减数的补码，通过加法来完成减法的思想不仅局限在二进制中。考虑十进制中的对应结构也可以帮助我们深入理解2的补码方案的机制。考虑两位十进制数的减法。像74－33＝41之类的计算很简单，因为减数的每一位都小于被减数的相应位，在计算过程中没有借位。但是要计算74－36＝38就有些复杂了，因为最小有效位的减法需要向前借位。

假设我们对计算式重新构造如下：

$$\begin{aligned}74-36&=74+100-100-36\\&=74+(100-36)-100\end{aligned}$$

现在需要进行两次减法运算。100减36同样需要借位，但是注意到100＝99＋1，因此写成如下形式就可以避免借位：

$$\begin{aligned}74-36&=74+(99+1-36)-100\\&=74+(99-36)+1-100\end{aligned}$$

括号里的运算不需要借位；只需用9减去对应的每位即可。可以看到，上面的表达式与图5-13中的电路所表示的2的补码之间有明显的相关性。用1减每一位数来求减数 Y 的1的补码与计算（99－36）十分相似。在十进制数中我们用9减去每一位数来求“9的补码”。在图5-13中我们使进位 $c_0=1$ 来得到 Y 的2的补码。在十进制系统中我们计算 $(99-36)+1=64$，所以64就是36的以10为基的补码。对于一个n位十进制数N，以10为基的补码定义为$K_{10}=10^n-N$，而以9为基的补码K_9定义为$K_9=(10^n-1)-N$。

所以所求的减法（74－36）可以通过加上减数的以10为基的补码执行，如下所示：

$$\begin{aligned}74-36&=74+64-100\\&=138-100\\&=38\end{aligned}$$

减法运算138－100的结果是显而易见的，只要删除138的最高位数1即可。正如曾在图5-11的减法示例中讨论过的那样，这样做类似于省略来自于图5-13所示的电路进位输出。

例5.1 设A和B是n位十进制数。用我们上面讨论过的以10为基的补码的方法，$A-B$可以这么来做：

$$A-B=A+(10^n-B)-10^n$$

若$A\geqslant B$，那么$A+(10^n-B)$ 产生进位1。这个进位相当于10^n，可以被忽略。

但是如果$A<B$，则$A+(10^n-B)$ 产生的进位是0，假设它的值是M，就有：

$$A - B = M - 10^n$$

可以重新写成

$$10^n - (B - A) = M$$

等式左侧是$(B - A)$的以10为基的补码。正数的以10为基的补码代表的是与其绝对值相同的负数。所以M就是当$A<B$时，$A-B$的正确计算结果。下面的例子也很好地说明了这种概念。■

例5.2 在处理二进制有符号数时我们用最左位的0表示正数，1表示负数。如果我们要构建十进制系统的电路，可以使用相同的方法。让最左侧的0表示正数，9表示负数。请注意：在十进制系统中9是0的以9为基的补码，就像在二进制系统中1是0的1的补码一样。

因此，三位数如$A = 045$ 和 $B = 027$是正数，大小分别为45和27。$A-B$可以进行如下操作：

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999-027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

得到了正确的结果+18。下面我们考虑被减数比减数小的情况，计算过程如下：

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

从这个表达式中我们看到似乎仍要计算982−1000，但从例5.1中我们可以看到它也可以写成：

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

因此，982表示的是以10为基的负数的补码，该数的绝对值等于$(A-B)$。从前面的计算我们知道$(A-B)=018$，即+18。因此有符号数982表示的是−18的以10为基的补码，这正是我们所需要的结果。■

例5.3 令$C=955$，$D=973$；因此C和D的值分别是−45和−27。C减去D可依照以下步骤进行：

$$\begin{aligned} C - D &= 955 - 973 \\ &= 955 + 1000 - 1000 - 973 \\ &= 955 + (999 - 973) + 1 - 1000 \\ &= 955 + 026 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

982正是−18的以10为基的补码，因此−18是正确的结果。现在考虑$D-A$的情况，其中$D = 973$，$A = 045$:

$$\begin{aligned} D - A &= 973 - 045 \\ &= 973 + 1000 - 1000 - 045 \\ &= 973 + (999 - 045) + 1 - 1000 \\ &= 973 + 954 + 1 - 1000 \\ &= 1928 - 1000 \\ &= 928 \end{aligned}$$

得到的结果928是－72的以10为基的补码。

这些例子说明了有符号数的减法可以不用带借位的减法操作来实现，唯一需要减法操作的地方就是构成减数的以9为基的补码的时候，而此时只需用9减去各位数字就行了。因此一个能计算9的补码并且带有一个一般加法器的电路就足够应付十进制有符号数的加减法运算。关键的一点是进行n位数的运算时，硬件只须处理n位数。所有最左端产生的进位都将被忽略。 ■

上面几个例子介绍的把减去一个数转化为加上该数的基数补码的概念具有一般性。如果基数是r，那么一个n位数N的以r为基的补码 K_r 可按下式求出：$K_r = r^n - N$。

而其$r-1$的补码K_{r-1}定义为$K_{r-1} = (r^n - 1) - N$。因此K_{r-1}被称为“基数减1的补码”，只要每个数位都用（$r-1$）减去N中每位的数字即可算得。构成基数减1的补码的电路比有借位减法操作的电路更简单一些。而用二进制数制时，其电路尤为简单，因为求1的补码只需要把每位反相就行了。

例5.4 在图5-11中我们曾说明如何由2的补码来进行减法运算。下面用我们讨论过的方法考虑如何计算(+5) － (+2)＝(+3)。每个数都用四位二进制数的形式表示，2^4的值就是10000，则：

$$\begin{aligned}0101 - 0010 &= 0101 + (10000 - 0010) - 10000\\ &= 0101 + (1111-0010) + 1 - 10000\\ &= 0101 + 1101 + 1 - 10000\\ &= 10011 - 10000\\ &= 0011\end{aligned}$$

由于5＞2，在第四位会有进位，这个进位代表了2^4，其二进制形式表示为10000。 ■

例5.5 现在计算 (+2) － (+5) ＝ (－3)，有：

$$\begin{aligned}0010 - 0101 &= 0010 + (10000 - 0101) - 10000\\ &= 0010 + (1111 - 0101) + 1 - 10000\\ &= 0010 + 1010 + 1 - 10000\\ &= 1101 - 10000\end{aligned}$$

由于2＜5，没有从第四个位的进位，计算结果1101就是－3的2的补码。请注意：

$$\begin{aligned}1101 &= 10000 + 0010 - 0101\\ &= 10000 - (0101 - 0010)\\ &= 10000 - 0011\end{aligned}$$

由此表明1101是 +3(0011)的2的补码。 ■

例5.6 最后考虑减数是负数的情况。如下计算(+5) － (－2) ＝ (+7)：

$$\begin{aligned}0101 - 1110 &= 0101 + (10000 - 1110) - 10000\\ &= 0101 + (1111-1110) + 1 - 10000\\ &= 0101 + 0001 + 1 - 10000\\ &= 0111 - 10000\end{aligned}$$

尽管5 ＞(－2)，但用2的补码表示时，－2的二进制补码1110比0101要大，因此没有从第四位产生的进位。答案0111就是 +7 的2的补码。请注意：

$$\begin{aligned}0111 &= 10000 + 0101 - 1110\\ &= 10000 - (1110 - 0101)\\ &= 10000 - 1001\end{aligned}$$

由此表明1001是－7的2的补码。 ■

5.3.5 算术溢出

我们假定加法和减法的运算结果的位数都在数的有效表示范围之内。若用n位表示一个有符号数，则运算结果必须在-2^{n-1}到$2^{n-1}-1$范围之内。若结果超出此范围，则说算术运算发生溢出。为了保证算术运算电路能正确地工作，电路中必须有能够检测是否发生溢出的机制。

图5-14列出了绝对值为7和2的两个数（以2的补码形式）相加的四种情况。由于我们用的是四位数，因此有3个位表示数的幅值，b_{2-0}。当两个数的符号不同时，不会发生溢出。但若两个数的符号相同，则运算结果的绝对值是9，无法用3位来表示，因此产生了溢出。设最高有效数位产生的进位为c_3，由符号位产生的进位为c_4，则由图5-14可以看出，判断溢出是否发生的关键是c_3和c_4的取值是否相同。

(+7)	0111	(−7)	1001
+ (+2)	+ 0010	+ (+2)	+ 0010
(+9)	1001	(−5)	1011
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0111	(−7)	1001
+ (−2)	+ 1110	+ (−2)	+ 1110
(+5)	10101	(−9)	10111
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

图5-14 决定是否发生算术溢出的例子

图5-14表明若两个进位值不同，则发生算术溢出，若两个进位值相同，则不会发生溢出，而能够产生正确的运算结果。事实上，以上的结论对于2的补码的加法和减法都适用。作为对上述结论的快速检查，我们重新考察图5-10所示的例子，这些例子中的操作数都很小，所以无论什么情况下都没有发生算术溢出。在图5-10上面的两个例子中，符号位和最大有效位的进位都是0；而在下面的两个例子中两个位的进位都是1。因此对于图5-10和图5-14的例子，判断算术溢出发生的条件是：

$$\begin{aligned}\text{溢出} &= c_3\overline{c}_4 + \overline{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

对于n位数，则有：

$$\text{溢出} = c_{n-1} \oplus c_n$$

由此可知，若我们对图5-13中的电路稍作修改，添加一个异或门，便可以指示是否发生算术溢出。

5.3.6 电路的性能问题

购买数字系统（例如计算机）时，购买者最关心的是该系统所能提供的性能和需要付出的价钱。总体而言，系统的性能越好，其价格也越高。然而，稍微多付出一点价钱，而使系统的性能有较大幅度的提高通常也是办得到的。因此，通常用价格/性能比（price/performance ratio）作为衡量系统是否值得购买的指标。

在计算过程中，加法和减法是最常遇到的基本运算，其运行速度对计算机系统的整体性能有重大的影响。为了更深入地理解这一点，让我们仔细考察一下图5-13所示的加法器和减法器的速度。从操作数X和Y到达输入端时刻算起，到加法器每一位和（Sum）的输出以及最终进位C_n的输出都变为（稳定）有效为止，是需要一段延迟时间的。我们所感兴趣的是此延迟时间的最大值。大部分延迟时间是由n位加法器的电路结构所引起的。假设加法器采用图5-6所示的行波进位结构，且每个一位全加器都采用图5-4c的电路结构，则该电路进位信号的延迟Δt 等于2级门的延迟。从5.2.2节，我们知道加法的最终结果需要等待$n\Delta t$延迟后才能有效，这段时间等于$2n$个门的延迟。除了行波进位路径的延迟，信号Y到达加法器的输入端还需要经过异或门（使Y保持原始值或者反相）的延迟。若异或门的延迟等于一个门的延迟，则图5-13

所示电路的总延迟等于$2n+1$个门延迟。若n比较大，例如$n = 32$ 或者$n=64$，则该延迟将使电路的性能变得极差，因而不得不放弃这种电路，从而寻找能更快地完成加法的电路。

任何电路的速度都受到该电路最长延迟路径的制约，对图5-13所示的电路，其最长的延迟路径从 y_i 的输入开始，经过异或门，并经过每级加法器的进位电路。该最长的延迟通常称为*关键路径延迟*，而造成该延迟的路径称为*关键路径*。

5.4 快速加法器

大规模数字系统的性能取决于构成该系统的各功能部件的运行速度。显然，采用高速的电路可以提高系统的性能。这可以通过应用高级新技术，使基本门电路的延迟减少，从而提高系统的性能；但我们也可以改变功能部件的整体结构，使系统性能有更显著的改进。本节将讨论n位加法器的另一种结构，这种结构能极大地降低加法运算所需要的时间。

超前进位加法器

为了减少由于并行加法器中的进位所引起的延迟，我们可以尝试快速判断每一位加法器的上一位的进位是0还是1。如果在相对较短的时间里能够做出正确的判断，加法器的整体性能就能得到提高。

由图5-4b可知第i级的进位函数为：

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

提出公因子，得到：

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

将上式改写为：

$$c_{i+1} = g_i + p_i c_i \tag{5.3}$$

其中，$g_i = x_i y_i$, $p_i = x_i + y_i$。

当x_i和y_i都等于1的时候，则不论上一级的进位c_i是多少，函数g_i都等于1。由于在这种情况下第i级一定会生成一个进位输出信号，因此 g 称为*生成函数*。就函数p_i而言，它的两个输入x_i和y_i至少有一个为1时函数p_i才是1，在这种情况下，若$c_i = 1$，则产生进位输出$c_{i+1}=1$。p_i为1的效果相当于使进位1经过第i级，传播到更高一位；因此p_i称为*传播函数*（propagate function）。

将式（5.3）扩展到第$i-1$级得到：

$$\begin{aligned} c_{i+1} &= g_i + p_i (g_{i-1} + p_{i-1} c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

对上式继续展开直至第0级为止，我们可推导得到以下表达式：

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \tag{5.4}$$

上式可以用两级与-或门电路实现，该电路能够很快地计算出进位输出c_{i+1}的值。基于此表达式的加法器称为*超前进位加法器*。

若想要深入理解式（5.4）的物理意义，把超前进位加法器的构造和行波进位加法器作一详细的比较是很有指导意义的。我们从对最小数位进行加法运算的两级加法器（即第0和第1级）的详细结构看起。图5-15所示的是行波进位加法器的最初两级加法器的结构，它的进位函数如式（5.3）所示，每一级本质上都与图5-4c所示的电路相同，但在具体结构上有一些修改，由于在积之和表达式（5.3）中求c_{i+1}时，提取了公因子，所以实现电路时添加了一个或门（以产生信号p_i），去掉了一个与门。

由于行波进位加法器中的进位信号必须经过很长路径的传播才能到达最高位，所以限制

了它的运算速度。在图5-15中，关键路径是从输入x_0和y_0到输出c_2，共经过了五个门，在图中用灰色线表示。n位加法器的后面几级所经历的路径与第一级相同，因此其关键路径上总共有$2n+1$个门的延迟。

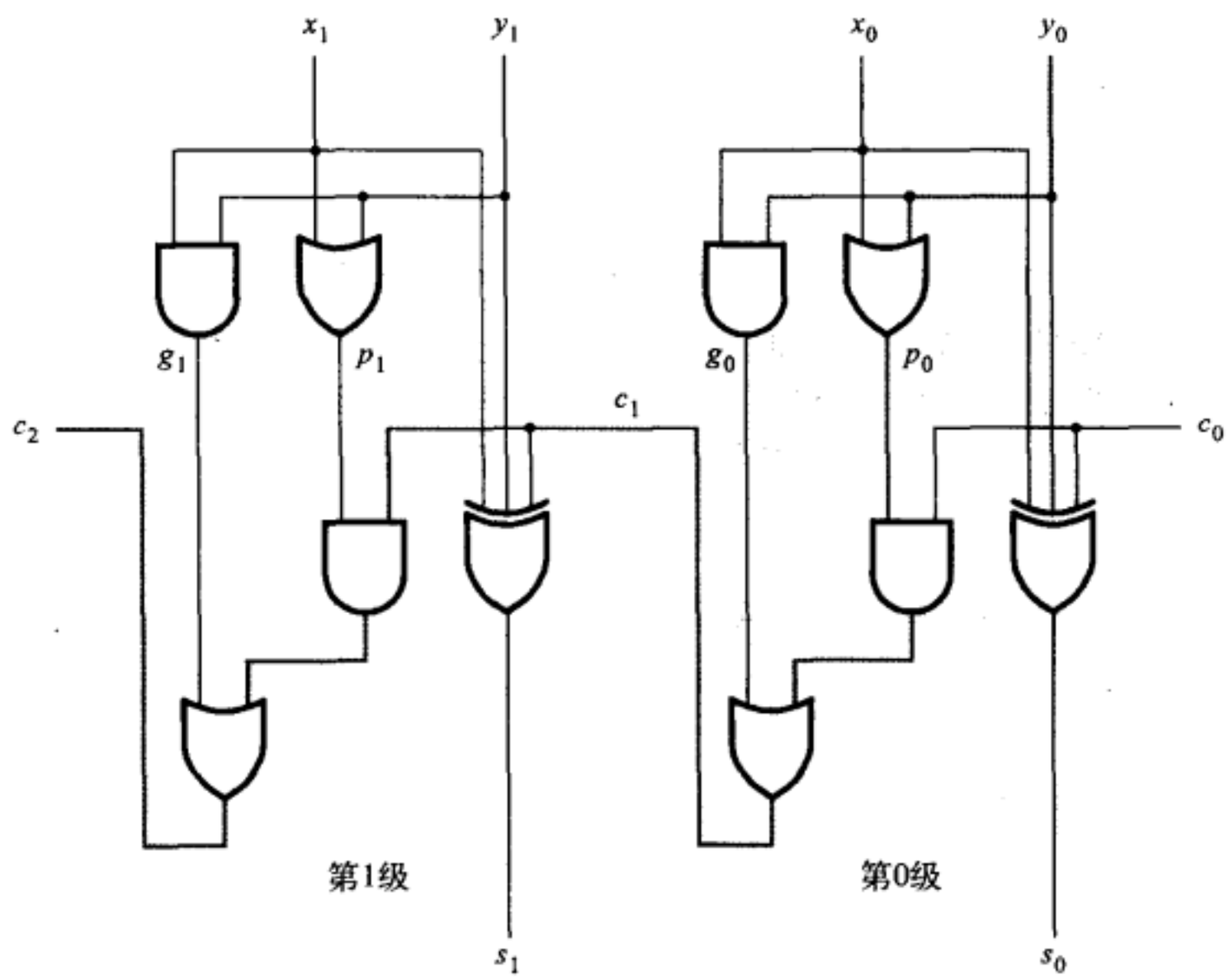


图5-15 基于表达式（5.3）的行波进位加法器

图5-16给出了超前进位加法器的最低两级，根据式（5.4）来计算其进位输出函数，即

$$c_1 = g_0 + p_0c_0$$

$$c_2 = g_1 + p_1g_0 + p_1p_0c_0$$

产生进位信号c_2的关键路径用灰色线标记。在该电路中，c_2与c_1都经过三个门的延迟，同时输出。若把这个电路扩展到n位，则最后一级的进位信号c_n也同样经过三个门的延迟后输出，因为表达式（5.4）正好对应于较大的两级与-或门电路。

所有的g_i和p_i信号在经过一个门的延迟后就已经确定，要得到所有进位信号还要两个门的延迟。最后，再有一个门的延迟（异或门）来生成所有的和位。这样，n位超前进位加法器总共需要四个门的延迟。快速估算出进位信号的值是确保超前加法器高性能的关键。

当n增大时n位超前进位加法器的复杂程度迅速增加。为了降低复杂度，我们可以使用分层结构来设计规模大的加法器。假设我们要设计一个32位加法器，可以先把它分解为四个8位模块：b_{7-0}位为模块0，b_{15-8}位为模块1，b_{23-16}位为模块2，b_{31-24}位为模块3；然后对于其中的每一块都可以将其当作8位超前进位加法器来设计。四个模块的进位输出信号分别是c_8，c_{16}，c_{24}和c_{32}。现在我们可以采用两种办法来产生进位：第一种办法是将这四个加法器模块按照行波进位加法器的四级连接起来。这样每个模块中进行超前进位的计算，而模块之间则按行波逐位进位的规则传播进位信号，其电路如图5-17所示。

第二种方法是模块间的进位不使用行波逐位进位的结构，而是添加一级超前进位的结构来计算模块间的进位，从而得到更加快速的进位电路。这种“层次化超前进位加法器”电路如图5-18所示。最上面一排的每个模块都是生成信号g_i和传播信号p_i的8位超前进位加法器，我们前面都已经讨论过了。然而不是从每个模块的最高有效位产生进位信号，而是每个模块

都为整个模块产生生成和传播信号。令G_j和P_j分别表示模块 j的生成信号和传播信号。第二级超前进位电路见图5-18的底部，G_j 和P_j 是此进位电路的输入，它的输出是所有模块间的进位信号。通过考察c_8的表达式，我们可以推导出模块0的生成信号G_0和传播信号P_0的表达式。

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4p_3 + p_7p_6p_5p_4p_3p_2$$
$$+ p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

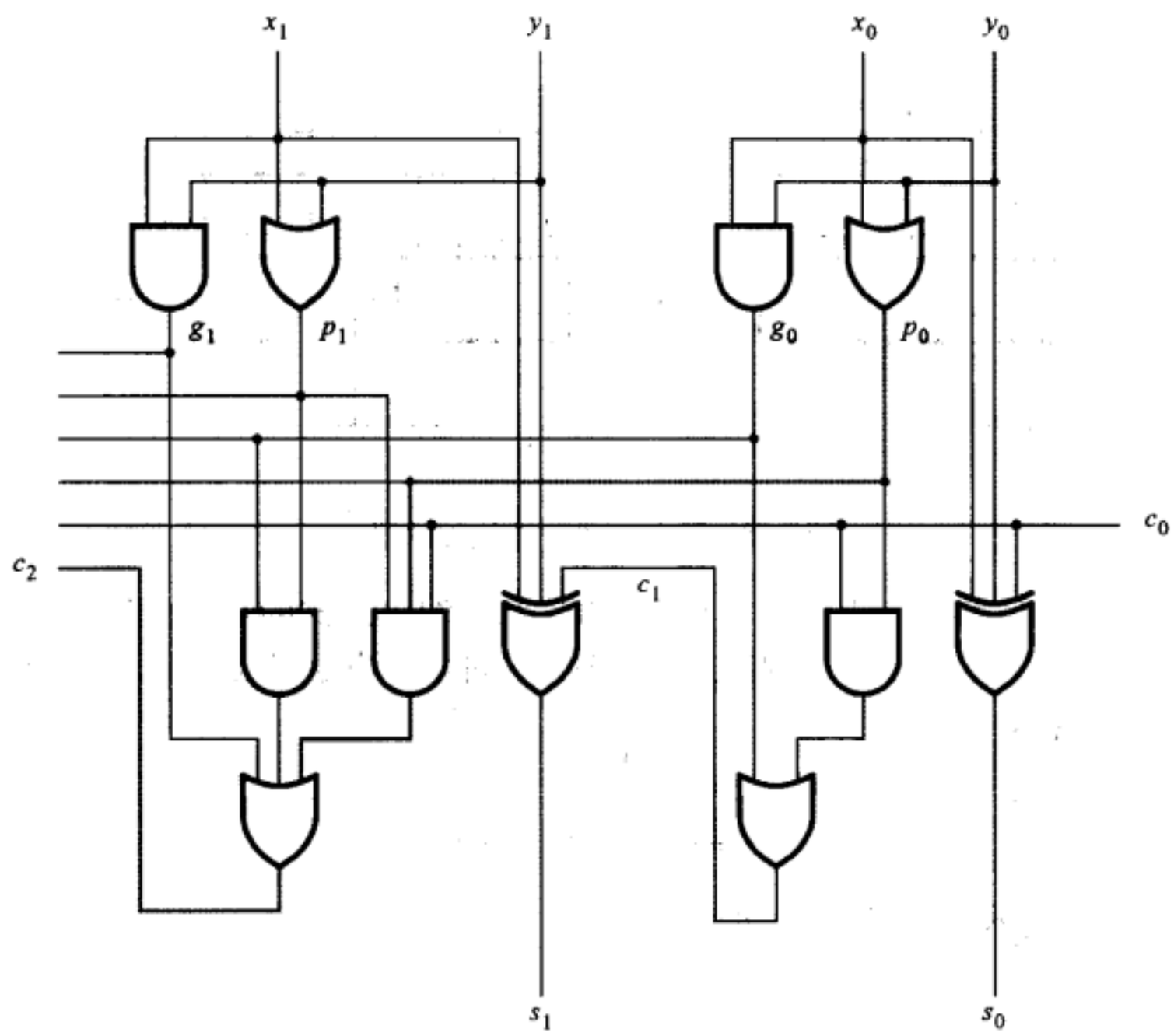


图5-16 超前进位加法器的最低两级

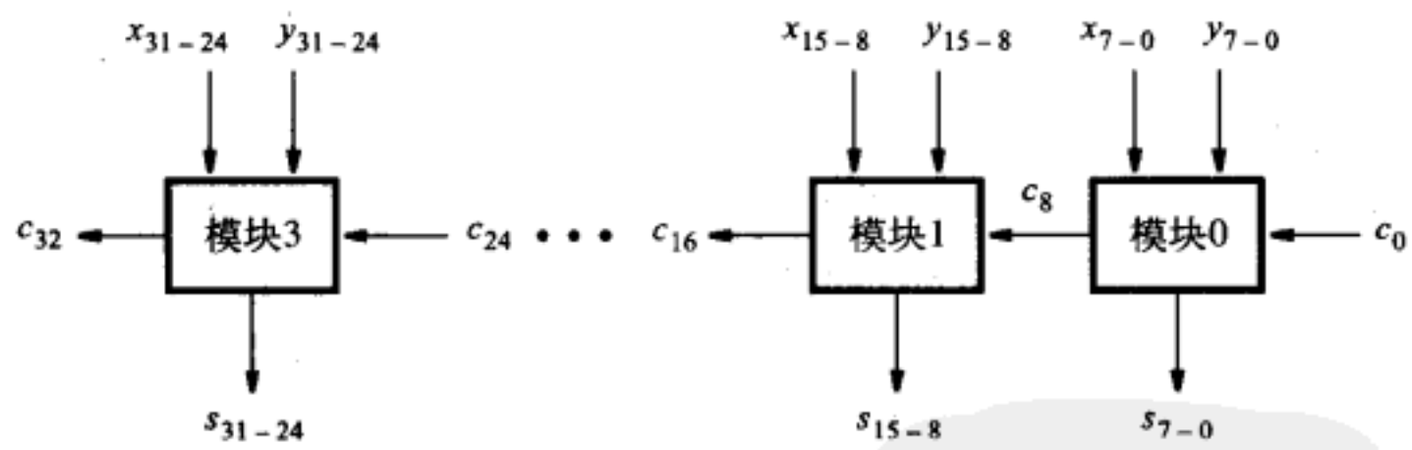


图5-17 块之间仍按照行波进位方式的层次化超前进位加法器

上式的最后一项表明，若所有8个传播函数都是1，则进位输入c_0就可以传过整个模块，因此：

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

c_8表达式的其余项则代表该模块产生进位的其他情况，因此：

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

由上面的推导，在层次结构的加法器中，我们可以将c_8的表达式写成：

$$c_8 = G_0 + P_0c_0$$

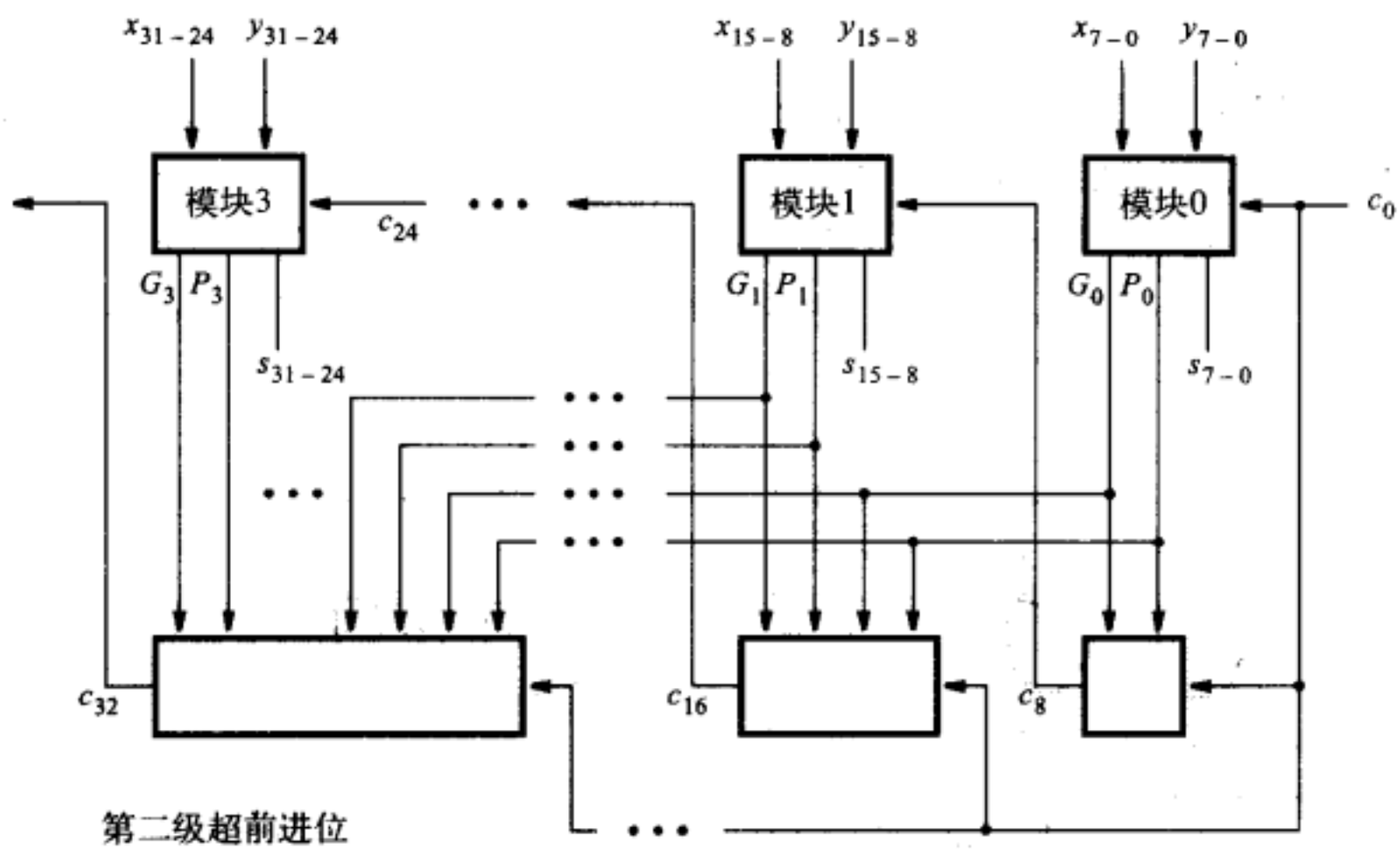


图5-18 层次化超前进位加法器

对于模块1来说，G_1和P_1的表达式与模块0的G_0和P_0的表达式的形式相同，除了下标i用$i+8$替代之外。信号G_2，P_2，G_3和P_3的表达式也可以用同样的方法求出。现在可以写出模块1的进位输出信号c_{16}的表达式为：

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1(G_0 + P_0 c_0) \end{aligned}$$

同样，可以写出c_{24}和c_{32}的表达式如下：

$$c_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_{32} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

使用这种结构，在产生G_j和P_j之后，还需两个门（一个与门和一个或门）的延迟，才能产生进位信号c_8，c_{16}和c_{24}。因为产生G_j和P_j需要三个门的延迟，所以c_8，c_{16}和c_{24}需要五级门延迟后才可以建立。因此，计算两个32位数相加所需的时间除了上面介绍的五级门的延迟外，还需要加上另外两级门的延迟来产生模块1，2，3的内部进位，产生每一个和位还需要一个异或门的延迟。因此，求得两个32位数相加和值总共需要8级门的延迟。

在5.3.5节中，我们曾推导出使用行波进位加法器来计算两个数的加法需要$2n+1$级门延迟。对于32位数的加法器而言，则需要65级门延迟。显然，超前进位结构的加法器在速度性能上有了很大幅度的提升，所付出的代价是电路结构变得更加复杂。

技术方面的考虑

前面的延迟分析是建立在不限制门电路输入个数的假设上的。从第3章和第4章我们知道，对当前门电路的实现技术而言，实际上输入端口的个数被限制在一个很小范围内。因此必须考虑扇入限制的因素。为了说明这一问题，让我们考虑前8个进位表达式：

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$\vdots$$
$$\begin{aligned} c_8 = {} & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 \\ & + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0 \end{aligned}$$

假设门的最大扇入是四个输入。那么，这些表达式就不可能用两级与或门电路来实现。最大的问题是c_8，因为它的其中一个与门需要9个输入；而且或门也同样需要9个输入。为了满足扇入限制的要求，我们可以将c_8表达式重新改写成：

$$c_8 = (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0$$

用电路实现上面的表达式，需要使用11个与门和3个或门。c_8的传播延迟由4项延迟组成：1）产生所有g_i和p_i的一个门延迟；2）产生括号里所有积之和项的两级门延迟；3）方括号里的乘积项的一个门延迟；4）最终各项或运算的延迟。因此要得到c_8的有效值需要经过五级门的延迟，而不是我们前面分析过的不考虑扇入限制时的三级门延迟。

因为扇入的限制降低了超前进位加法器的速度，某些低扇入器件的内部添置了实现快速加法器的专用电路。FPGA就是这类器件的一个例子，FPGA内部的逻辑块是基于查找表的。

在结束超前进位加法器的讨论之前，我们应该考虑一下与图5-16所示电路不同的另一种加法器，如图5-19所示。该电路可以实现与图5-16所示电路完全相同的功能。其中，第0级加法器就是用图5-5所示的电路实现的，该电路用2个两输入异或门实现求和（Sum）位的运算，而不是用1个三输入异或门。图5-19所示的第一个异或门的输出也可以用作传播信号p_0。因此图5-16中相应的或门就可以省略。构建第1级电路的方法与构建第0级的相同。

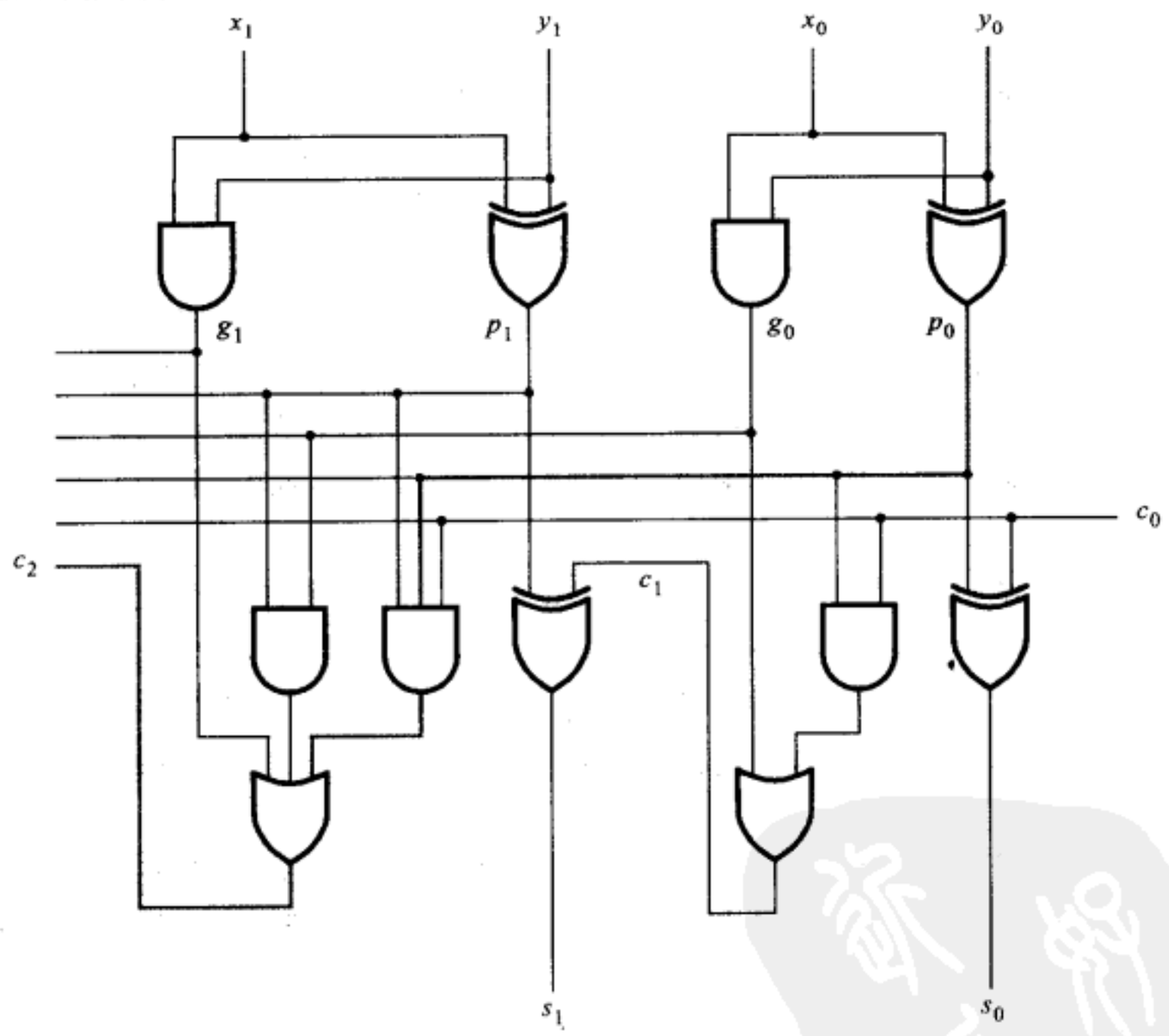


图5-19 超前进位加法器的另一种设计方案

图5-19所示的电路需要门的个数与图5-16所示电路相同。但这两种电路是否其中一个在某些方面更好一些呢？想要找到问题的答案，可以从电路的实现技术角度来考虑。若使用的是CPLD或FPGA（如图3-33和图3-39所示电路），则选择哪种电路都无所谓。若我们用积之和形式，把求和表达式写成：

$$s_i = x_i\bar{y}_i\bar{c}_i + \bar{x}_iy_i\bar{c}_i + \bar{x}_i\bar{y}_ic_i + x_iy_ic_i$$

因为CPLD中的宏单元可以实现四个乘积项，所以只用一个宏单元就可以实现1个三输入的异或门函数。

FPGA中的每个逻辑单元都可以实现任何三输入逻辑函数；因此很容易实现一个三输入的异或门。然而，假如现在我们要在某个客户定制芯片上实现超前进位加法器，且想使用在3.9.1节中讨论过的方法来制造，则三输入的异或门实际是用两输入的异或门来实现的。这就像我们在图5-19中求和位时做的那样，第一个异或门实现 $x_i \oplus y_i$（同时也是传播函数 p_i）。很明显，图5-19中所示的结构更有吸引力。以上讨论的要点说明逻辑电路的优化与将要采用的技术有关，CAD工具一般都会考虑到这一点。

超前进位加法器是众所周知的概念。市场上可以购买到专门实现超前进位电路部分的标准芯片，该电路被称为“*超前进位发生器*”。CAD工具常包含一些预先设计好的加法器子电路模块供人们使用，以便于设计更大规模的电路。

5.5 使用CAD工具设计算术电路

这一节将展示如何使用CAD工具进行算术电路的设计。我们将介绍两种不同的设计方法：使用原理图编辑输入工具的方法和编写Verilog代码的方法。

5.5.1 使用原理图编辑工具设计算术电路

一种常见的设计算术电路的途径是：使用原理图编辑工具绘制一张包括所需逻辑门的原理图。例如，若想要设计一个n位的加法器，可以先画出一张表示1位全加器的原理图，然后再将这n个1位全加器的实例连接起来，构成一个更高层次上的n位行波进位加法器电路的原理图。以这种方式绘制的层次化加法器原理图看起来将与图5-6中所示的电路图很类似。我们也可以用这种方法来构建如图5-13所示的加法/减法器电路。

这种方法最主要的问题是使用起来非常麻烦，尤其当处理的数据位数很大的时候，更是如此。若想要考虑构造超前进位加法器的原理图，则该问题显得越发明显。从5.4.1节我们已经看到，超前进位加法器的进位电路随着级数的增加会变得非常复杂，因此有必要对每一级加法器画一张单独的原理图。因此，用原理图编辑工具设计算术电路的更好的办法是使用预先定义的子电路模块。

在2.9.1节我们曾讲过，原理图输入工具提供了一个包含各种基本逻辑门的图形符号库。这些门可以用来创建相对简单的电路的原理图。除了基本的逻辑门之外，图形符号库还提供了一些常用的电路部件，诸如加法器之类的电路。在设计大型电路时，每个电路部件可以作为子模块调用。在CAD系统工具中一般称这类电路模块为*宏单元或宏函数*（macrofunction or megafunction）。

宏单元或宏函数可分为与技术有关和与技术无关两种。*与技术有关的宏函数*是为特定类型的芯片而设计的。例如，在5.4.1节中我们描述了一个扇入被限制在四个输入的超前进位加法器的表达式，实现此表达式的宏函数在技术上是明确的。而*与技术无关的宏函数*可以在任何类型的芯片上实现。若一个加法器宏函数可以代表不同类型芯片上的任何加法器电路，则该加法器宏函数就是技术无关的。

Quartus II的CAD系统有一个参数化模块库（LPM），这是宏函数库的一个很好的例子。LPM库中的每个模块都是与技术无关的，也是参数化的。这意味着它可以以多种方式应用。例如，LPM库中有一个称为lpm_add_sub的n位加法器模块。

包含lpm_add_sub模块的原理图如图5-20所示。这个模块有几个相关参数都可以通过CAD工具来进行设置。其中我们讨论所用到的两个最重要的参数是LPM_WIDTH和

LPM_REPRESENTATION。如上所述，参数LPM_WIDTH规定了加法器的位数。而参数LPM_REPRESENTATION则规定所使用的是符号数还是无符号数。从5.3.5节我们知道，这个参数的设定只影响到电路模块中决定是否有算术溢出的部分。对于图5-20所示的原理图而言，LPM_WIDTH＝16，使用的是有符号数，输入信号add_sub能够决定模块究竟执行的是加法运算还是减法运算。因此这个模块就代表了如图5-13所示的加法/减法器电路。

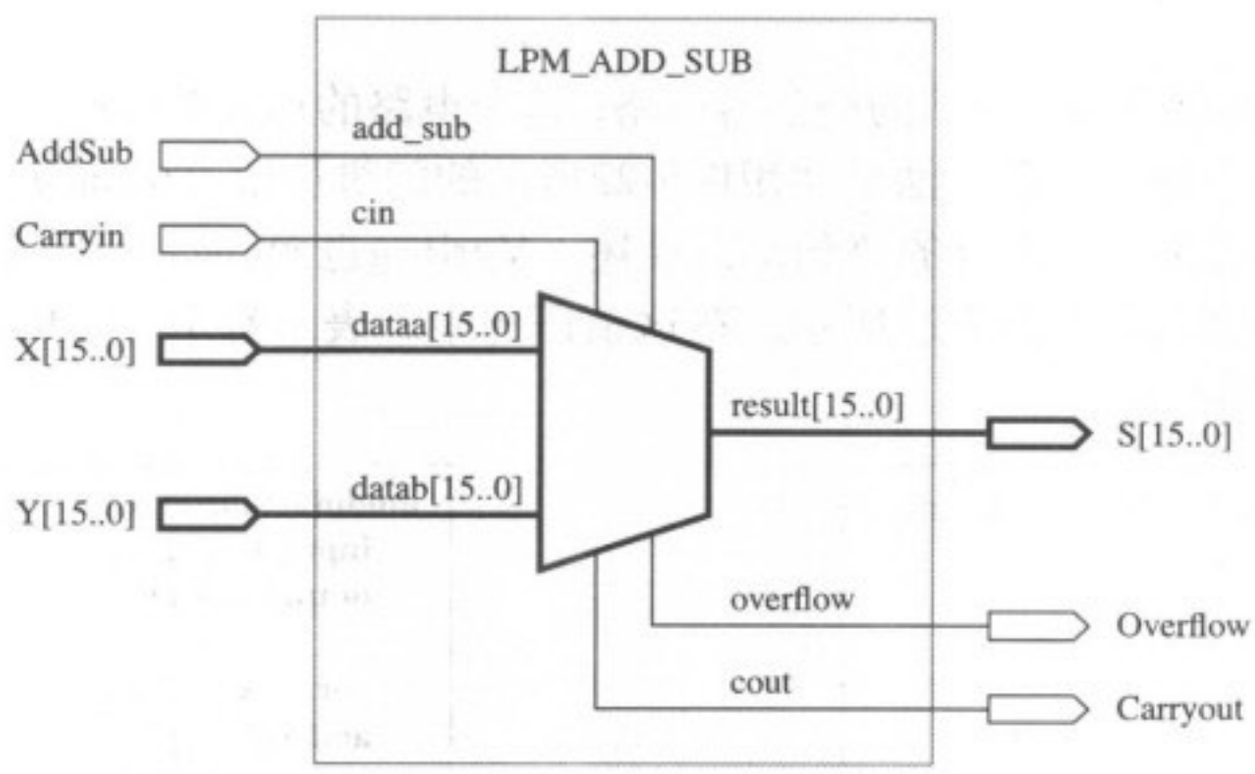


图5-20 使用一个LPM加法/减法器模块的原理图

由lpm_add_sub模块相加的两个数连接到该模块的两个端口：dataa[15..0]和datab[15..0]。名字中的方括号表示它们都是多位数。在原理图中，dataa[ ]和datab[ ]连接到16位的输入端口*X* [15..0]和*Y* [15..0]。*X* [15..0]的意思是信号*X*是一个16位数，各个位分别是*X* [15], *X* [14], …，*X* [0]。由lpm_add_sub模块产生的求和（sum）输出端口称为result[15..0]，连接到输出*S* [15..0]。从图5-20中我们还可以看到，LPM支持一个进位输入以及进位输出和溢出输出。

为了评估LPM的效率，我们配置了lpm_add_sub模块，用一个16位的加法器计算和、进位输出和溢出输出，这意味着不需要add_sub和cin信号。我们使用CAD工具在FPGA芯片上实现这个电路，然后对其性能进行仿真。图5-21给出了仿真结果。16位信号*X*、*Y*和*S*的值以十六进制的形式输出。在仿真刚开始的阶段*X*和*Y*的值都被设为0000，50 ns（纳秒）以后*Y*变为0001，这使得*S*变为0001。输入的下一次变化发生在150 ns，*X* 变为3FFF。要得到正确结果4000，加法器必须等待进位信号从第一级加法器传输到最后一级，这可以从*S*在得到稳定值之前的一系列快速跳变中看出。观察仿真器的参考框，图中粗垂直线所在的位置是160.93ns，也就是正确结果产生的时刻。由于输入的变化发生在150 ns，因此得到最终结果花费的时间是160.93 − 150 = 10.93 ns。在250 ns时刻*X*变成7FFF，得到的和信号*S*应该是8000，由于*S*超过了16位有符号数所能表示的最大范围，因此，*Overflow*信号被置1，表明发生了算术运算的溢出。

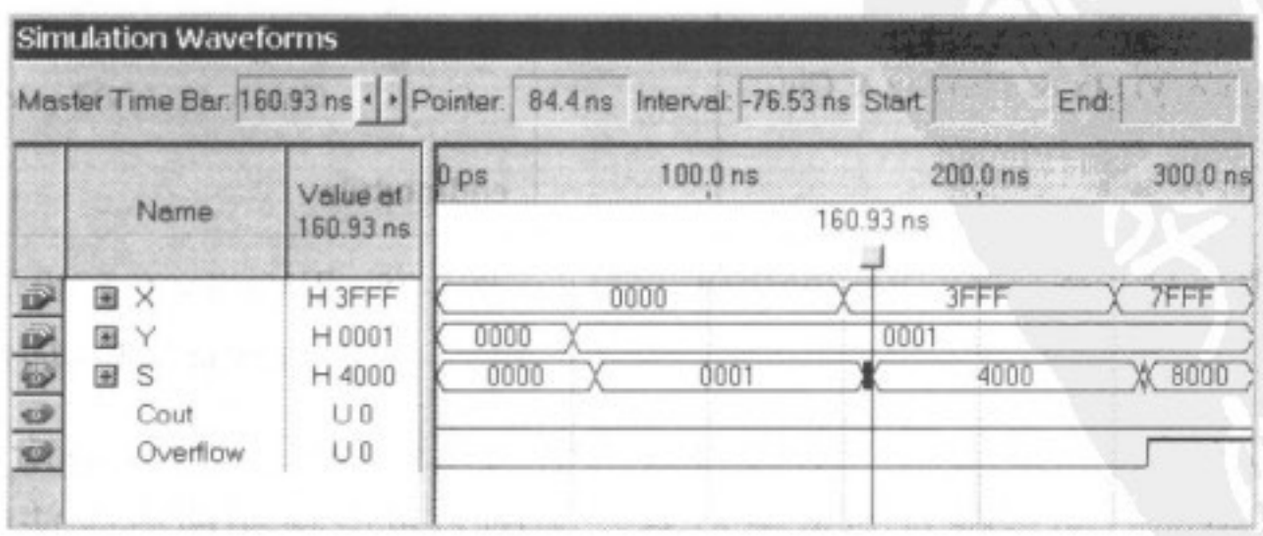


图5-21 LPM加法器的仿真结果

5.5.2 使用Verilog设计算术电路

在5.5.1节中我们曾经提到，构造n位加法器最直接的方法是绘制一张包含n个1位全加器的层次化原理图。若用Verilog语言来描述n位加法器，也可以用类似的方法，我们先设计一个1位的全加器模块，再定义一个高层次模块来实例引用n个1位全加器模块。下面我们将展示如何编写行波进位加法器的层次化模块，作为使用Verilog语言设计算术电路的第一次尝试。

假设我们想要实现图5-4c所示的全加器电路。这个电路的输入为Cin, x和y，输出为s和C_{out}。用Verilog来描述该电路的一种方法是使用图5-22所示的门级原语（primitive）。该电路中的三个与门中的每一个都用一条独立的语句加以描述。Verilog也允许将引用同一原语的多条语句合并为一条单独的语句，见图5-23所示。在这条语句中，表示每个与门的语句段需要用逗号隔开，最后才用分号结束。

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y);
   and (z2, x, Cin);
   and (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

图5-22 使用门级原语来设计全加器的Verilog代码

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y),
       (z2, x, Cin),
       (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

图5-23 用Verilog代码设计图5-22中所示电路的另一种写法

构造全加器的另一种方法是使用图5-24所示的逻辑表达式。其中异或操作用“^”操作符表示。同样，两条分开的连续赋值语句，也可以如图5-25所示那样，合并成一条单独的语句。

以上两种方法都可以综合出相同的全加器电路。我们现在可以为逐位进位加法器单独编写一个Verilog模块，在代码中实例引用子电路模块fulladd。图5-26所示为表示四位行波进位加法器的一种代码。该模块的名字为adder4，其中包含一个四位行波进位加法器的代码。一个加数的四个位由四个信号x_3, x_2, x_1, x_0表示，另一个加数由y_3, y_2, y_1, y_0表示，运算产生的和（sum）信号由s_3, s_2, s_1, s_0表示。该电路还有一个进位输入信号*carryin*，输入到最低有效位，以及一个进位输出信号*carryout*，从最高有效位输出。

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

图5-24 使用连续赋值构造全加器的Verilog代码

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

图5-25 用Verilog代码设计图5-24中所示电路的另一种写法

图5-26所示的四位加法器是用四条实例引用语句来描述的。每一条语句的开头都是所引用模块的名字fulladd，后面紧跟着的是实例名。实例名可以使用所有合法的Verilog名，但必须是唯一的。最低有效位加法器的名字是stage0，最高有效位的名字是stage3。在adder4模块

中将连接到fulladd模块每个输入输出端口的信号在后面逐一列出。这些信号列出的顺序与在fulladd中是相同的，即排列次序为：C_{in}, x, y, s, C_{out}。

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
   input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
   output s3, s2, s1, s0, carryout;

   fulladd stage0 (carryin, x0, y0, s0, c1);
   fulladd stage1 (c1, x1, y1, s1, c2);
   fulladd stage2 (c2, x2, y2, s2, c3);
   fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

图5-26 四位加法器的Verilog代码

每个fulladd模块实例后面列出的信号名隐含表示了这些全加器子模块是如何互相连接在一起的。例如，实例stage0的进位输出连接到实例stage1的进位输入。综合出的电路与图5-6所示的电路结构相同。fulladd模块可以如在图5-27所示的那样，编写在表示adder4模块的同一个Verilog源代码文件中，也可以单独为fulladd模块编写一个文件。若编写在单独的文件中，则必须在adder4模块的Verilog源代码文件中用编译指示语句清楚地标明 fulladd 文件所在的位置，以便编译器进行处理。

5.5.3 使用向量信号

图5-26所示加法器的每个四位输入和四位输出都用1位的信号来表示。使用多位信号，即向量来表示数字是一种更方便的表示办法。就像逻辑电路中多条信号线上的信号所表示的数，在Verilog代码中也可以用一个多位向量来表示数。输入向量的举例如下：

input [3:0] W;

这条语句将W定义为一个四位的向量。它的每一位都可以通过在方括号里写入其对应的位数来表示。因此最高有效位可表示为$W[3]$，而最低有效位可表示为$W[0]$，中间两位向量可表示为$W[2:1]$，符号W表示整个向量。用向量定义的四位加法器代码见图5-27所示。除了将输入信号X和Y以及输出信号S都定义成向量外，也可以将全加器之间的进位信号定义为三位的向量$C[3:1]$。请注意，输入stage0的进位信号仍为*carryin*，而从stage3输出的进位信号仍为*carryout*。

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

图5-27 用向量表示的四位加法器

Verilog将那些用于模块间的连接，即既不是输入也不是输出的信号，定义为**wire**类型的信号。在adder4模块中我们需要三个内部的进位信号，在下面语句中定义为三位的向量：

wire [3:1] **C**;

在图5-27中，信号$C[1]$用来连接第0级全加器的进位输出和第一级全加器的进位输入。同样，$C[2]$和$C[3]$用来连接加法器的其他级的进位输出/输入。

方括号内的信息给出了向量的位宽，譬如$W[3:0]$，方括号内冒号左边的数字表示最高有效位，右边表示最低有效位。因此，$W[3]$是最高有效位，$W[0]$是最低有效位。方括号内表示最高位和最低位的数的大小也可以反过来，譬如由 $Z[0:3]$所定义的四位向量中，$Z[0]$为最高有效位，$Z[3]$为最低有效位。在用向量表示数时，通常自然地会使用MSB和LSB术语；而在不表示数值时，位选择数（index）只是用来识别向量中的某个特定的位而已。

5.5.4 使用自动生成语句

图5-28所示的行波进位加法器的设计方法十分局限，因为该加法器的电路只能进行预先确定的四位数的运算。若要用Verilog语言设计一个32位的加法器，则需要32个全加器子电路的实例引用，每个实例引用都得用一个单独的语句定义。从设计者的观点而言，希望能够定义一个可以对任意位宽数据进行运算的加法器，而其数据位宽可以通过参数来设定。

Verilog语言允许使用一般的参量，这个参量可以被明确地赋以所需的数值。例如表示数的n位向量可以写成$W[n-1:0]$。若我们在Verilog语句中定义：

parameter n = 4;

则W的范围就是 [3:0]。

图5-6所示的行波进位加法器可以用如下逻辑语句描述：

$$s_k = x_k \oplus y_k \oplus c_k$$

$$c_{k+1} = x_k y_k + x_k c_k + y_k c_k$$

其中$k = 0, 1, \cdots, n-1$。我们也可以不用图5-27所示的实例引用的办法来定义加法器，而改用Verilog描述的逻辑表达式语句来定义想要的加法器。

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n- 1:0] X, Y;
   output reg [n -1:0] S;
   output reg carryout;
   reg [n:0] C;
   integer k;

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

图5-28 行波进位加法器的一般性描述

图5-28展示了定义n位加法器的Verilog代码。输入X，Y以及输出和（sum）值S都被声明为n位向量。为了使加法器电路进位信号的使用更加容易，我们定义了一个$n+1$位的向量C。$C[0]$位是最低有效位的进位输入信号，$C[n]$位是最高有效位的进位输出信号。因此，就该n位加法器而言，$C[0] = carryin$，$C[n] = carryout$。

图5-28中介绍了如何使用Verilog的**for**语句来描述行波进位加法器的重复性结构。与2.10.2节所介绍的**if-else**语句一样，**for**语句是必须放在**always**块中的过程性语句，如图5-28所示。在2.10.2节我们已经解释过，只有当**always**块的敏感电平列表中的变量发生变化时，该**always**块中被赋值的信号才会被重新赋值，而在此之前它们的值将保持不变。这类信号必须声明为**reg**型变量，譬如图5-28中的*carryout*, S和C信号；而X，Y 和*carryin*是敏感变量。

在我们的例子中，**for**循环由**begin**和**end**之间的两条语句定义。这两条语句定义了对应于循环变量为k的那级加法器的求和函数以及进位函数。k的范围是0到$n-1$，它的值每经过一次循环就递增1。可以看到Verilog中的**for**语句与C语言中的**for**循环有相似之处，但是Verilog中没有C语言中的“++”和“--”操作符，所以循环中的递增和递减只能写成$k = k+1$ 和 $k = k-1$，而不是 k++ 和 k--。

请注意：在Verilog代码中k 被声明为整型变量用于控制**for**循环的迭代次数，但它并不代表电路中的物理连接。**for**循环的作用只是在每一次循环迭代时重复执行循环内部的语句。例如，若在本例中k被设置为 2，则**for**循环相当于如下四条语句：

S[0] = X[0] ∧ Y[0] ∧ C[0];
C[1] = (X[0] & Y[0]) | (X[0] & C[0]) | (Y[0] & C[0]);
S[1] = X[1] ∧ Y[1] ∧ C[1];
C[2] = (X[1] & Y[1]) | (X[1] & C[1]) | (Y[1] & C[1]);

由于在**parameter**语句中声明n的值为32，所以此图中的代码实现的是32位加法器。

用生成（**generate**）能力

在图5-27所示的程序中，我们实例引用了四个全加器来指定一个四位的逐位进位加法器。借助于n次实例引用全加器子电路的循环语句，该方法可以用来指定一个n位的加法器。Verilog语言中的**generate**结构提供了我们想要的能力。该结构允许实例引用包括在**for**循环语句和**if-else**语句内的语句。若一个**for**循环语句包括在**generate**结构块中，则循环变量必须声明为**genvar**类型的变量。**genvar**类型的变量类似于整型（**integer**）变量，但是只能具有正值，而且只能用于**generate**结构块内部。

图5-29展示了如何通过实例引用n个全加器（fulladd）来编写addern模块。每个由编译器在**for**循环语句中生成的实例都有其唯一的实例名，该实例名是由编译器根据循环标记自动生成的。生成的实例名分别为 *addbit*.[0].*stage*,…, *addbit*.[n − 1].*stage*。这个代码与图5-28中的代码产生的结果完全相同。

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   wire [n:0] C;

   genvar i;
   assign C[0] = carryin;
   assign carryout = C[n];

   generate
      for (i = 0; i <= n-1; i = i+1)
      begin:addbit
         fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
      end
   endgenerate

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

图5-29 由生成（**generate**）语句指定的逐位进位加法器

5.5.5 Verilog中的线网和变量

Verilog可以为逻辑电路建模，用Verilog描述的逻辑元件之间的互连以及描述行为的过程性语句都可以为逻辑电路建模（这两种形式混合使用也可以）。逻辑元件之间的连接被定义为线网（nets），由过程语句产生的信号称为变量（variable）。

线网

线网代表电路中的节点。线网可以有许多不同的类型，若以综合为目的，则**wire**型线网是我们需要关心的唯一类型。在5.5.3节中我们已经使用过这种类型的线网。电路中的逻辑单

元的输出通过**wire**型变量可以连接到另一个逻辑单元的输入。我们可以用标量表示单个连接，用向量表示多个连接。例如在图5-26中，进位信号c_3、c_2和c_1都是标量，它们表示全加器模块间的连接。专用的连接是由全加器模块按实例引用的方式定义的。在图5-27中，同样的进位信号也可以被定义为一个三位向量*C*。注意观察到图5-26所示的进位信号并没有显示定义为**wire**型变量，这是因为在Verilog语法中，所有信号在没有指明类型时都被默认为线网型。当然，若将以下语句添加到图5-27所示的代码中也是正确的：

wire c3, c2, c1;

在图5-27所示的例子中，有必要先声明向量*C*的存在；否则Verilog编译器不会认为*C*[3]，*C*[2]和*C*[1]是构成*C*的信号。由于这些信号是线网，我们就把向量*C*声明为**wire**类型。

还有另一种称为**tri**类型的线网。这种线网是三态类型的线网，即它们除了逻辑0和1两种状态以外，还有高阻态*z*。**tri**型的线网与**wire**型线网在Verilog中是用同样的方式处理的，而使用**tri**这个名称只是为了增加包含三态门代码的可读性。我们将在7.14节中讨论**tri**型线网的应用。

变量

Verilog提供了变量用以从电路行为的角度来描述电路。在Verilog语句中，可以给变量赋值，这个值在执行过程中会一直保持不变，直到接下来的赋值语句给这个变量赋以新值而将其原值覆盖。Verilog语言中总共有**reg**和**integer**两种类型的变量。我们曾在2.10.2节中提到，由过程语句赋值的所有变量必须首先定义为**reg**或者**integer**类型。图5-28所示的标量*carryout*以及向量*S*和*C*就是**reg**型变量的例子；而同一个图中的循环变量 *k* 则被声明为**integer**型变量，用来表示循环的次数。这些变量对于描述电路的行为是很有用的；但它们与最终生成电路的信号通常没有直接的对应关系。

在附录A的A.6节中，我们将会对线网和变量作进一步的讨论。

5.5.6 算术赋值语句

我们在设计工作中经常会用到算术运算，因此若能直接用硬件描述语言描述算术运算的电路将会带来极大的方便。在Verilog语言中，可以使用算术赋值语句和向量来执行这种算术运算。若我们定义如下向量：

input [n−1:0] X, Y;
output [n−1:0] S;

则算术赋值语句

S = X + Y;

就代表了*n*位加法器。

除了常用于加法运算的“+”操作符之外，Verilog语言还提供了许多其他算术操作符。Verilog的操作符将在第6章的6.6.5节和附录A的A.7节中作详细讨论。包含上述语句的完整Verilog代码如图5-30所示。请注意：**always**块中只有一条语句，因此可以不必写**begin**和**end**。该代码定义了可以生成*n*位加法器的电路，但该加法器电路并不包含进位输出信号和算术溢出信号。包含这两个信号的*n*位加法器的代码如图5-31所示。

```
module addern (carryin, X, Y, S);
  parameter n = 32;
  input carryin;
  input [n-1:0] X, Y;
  output reg [n-1:0] S;

  always @(X, Y, carryin)
    S = X + Y + carryin;

endmodule
```

图5-30 使用算术赋值来确定一个*n*位（比特）加法器

从MSB位输出的进位信号可以用下面的表达式得到：

$$carryout = x_{n-1}y_{n-1} + x_{n-1}\bar{s}_{n-1} + y_{n-1}\bar{s}_{n-1}$$

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;

   always @(X, Y, carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n-1] & Y[n-1]) | (X[n-1] & ~S[n-1]) | (Y[n-1] & ~S[n-1]);
      overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
   end

endmodule
```

图5-31 带进位输出和溢出信号的n位加法器

（请注意：这只是一个普通的逻辑表达式，里面+号表示或操作。）我们已经在5.3.4节中定义了算术溢出的表达式为$c_n \oplus c_{n-1}$，在我们所举的例子中，c_n对应着进位输出，但是没有直接访问c_{n-1}的途径，c_{n-1}是来自于（$n-2$）位的进位输入。可以证明表达式$x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$就是等于$c_{n-1}$，因此：

$$overflow = carryout \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$$

建议读者把验证这些表达式的正确性当做练习。

另一种包含进位输出和算术溢出信号的方式如图5-32所示。代码中使用了n+1位向量Sum。最高位$Sum[n]$成为从$n-1$位输出的进位信号。将X，Y以及$carryin$三个数求和并赋给信号Sum的语句采用了一种特殊的语法。即在语句中使用了大括号：{1'b0, X}和{1'b0, Y}表示在n位向量X和Y的左侧添加了一个为0的空位，构成一个新的n+1位向量。在Verilog里面操作符{,}叫做*位拼接运算符*。

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;
   reg [n:0] Sum;

   always @(X, Y, carryin)
   begin
      Sum = {1'b0, X} + {1'b0, Y} + carryin;
      S = Sum[n-1:0];
      carryout = Sum[n];
      overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
   end

endmodule
```

图5-32 带进位输出和溢出信号的n位加法器的另一种描述

若A是一个m位的向量而B是一个k位的向量，则{A, B}就拼接成一个(m + k)位的向量，在这个新向量中高m位为A，而低k位为B。记号1'b0表示逻辑值为0的1位二进制数。图5-32的代码中使用位拼接运算符的目的是将从第n − 1位的进位赋给$Sum[n]$。实际上我们设置了$x_n = y_n = 0$，由此：

$$Sum[n] = 0 + 0 + c_{n-1}$$

这个例子很有用，因为它简明地介绍了拼接运算符的概念。但我们也可以用更简化的写法：

Sum = X + Y + carryin;

由于Sum是一个n+1位向量，因此在进行X和Y的加法操作时，会将X和Y当作n+1 位向量，即在X和Y左侧添加一个0位，再进行加法运算。

在图5-32的代码中，另一个需要注意的细节是声明语句：

S = Sum[n−1:0];

这样就把Sum的低n位赋给了输出S。另一条语句将这个加法运算的进位输出$Sum[n]$赋给信号$carryout$。

图5-31和图5-32所示的代码以加法器设计内容为背景，展示了Verilog语言的一些特点。笼统地说，给定的设计任务可以有多种不同的实现途径，贯穿本书我们都能看到这一点。现在让我们再用另一种方法设计一个*n*位加法器。在图5-32中使用了*n*+1位向量*Sum*作为产生的*n*位和向量*S*以及从第*n*−1级进位信号的中间变量。这需要用两条Verilog语句从*Sum*中取出我们所需的位。我们已经展示了如何使用拼接运算符给向量 *X* 和 *Y* 的最高位左侧添加0，将其扩展为*n*+1位向量。我们同时也指出这样做其实是没有必要的，因为若运算过程产生的结果值有更多位数的话，则系统会自动给向量最高位添加0。更加简洁的写法是在加操作等式的左侧应用拼接操作符号，将*carryout*和*S*向量拼接起来，见下式所示：

{carryout, S} = X + Y + carryin;

因此就没有必要再使用*Sum*信号作为中间变量，Verilog代码就可以简化成如图5-33所示的代码。既然图5-32和图5-33中的代码所描述的加法器的行为相同，因此Verilog编译器很可能从这两种不同的代码生成相同的电路，但图5-33中的代码更加简洁一些。

请注意：我们也可以用同样的方法来定义全加器，见图5-34所示。从图5-22到图5-25中的全加器全部都是通过基本逻辑运算来定义其结构的，而图5-34中的代码却不同，它所描述的是加法器的行为，Verilog编译器使用目标芯片的工艺技术来实现加法器的合适结构细节。

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;

   always @(X, Y, carryin)
   begin
      {carryout, S} = X + Y + carryin;
      overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
   end

endmodule
```

图5-33 简化的完整的*n*位加法器代码

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output reg s, Cout;

   always @(x, y, Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

图5-34 全加器的行为描述

Verilog编译器翻译代码时，它会生成加法器电路来执行“+”操作符的运算。当使用Quartus II CAD系统时，编译器所用的加法器实际上是图5-20所示的lpm_add_sub 模块。编译器将为这个模块自动设置参数，将parameter语句中声明的*n*设置成具体的数值，以便让该模块代表*n*位的加法器。为了完整起见，也应该提一下：我们可以在Verilog代码中直接实例引用lpm_add_sub模块，其方法与直接实例引用fulladd模块类似，见图5-27中的代码。附录A的A.12节中给出了这样一个例子供大家参考。

5.5.7 Verilog代码中数的表示

Verilog代码中的数可以作为常量给出。这些数可以用二进制（b）、八进制（o）、十六进制（h）或者十进制（d）来表示。数的位数可以是指定的，也可以是不指定的。

有指定位数的数的格式如下：

<位数>，<基数标识> <有效数字>

上面格式中的<位数>是指表示该数所需要的二进制数的位数，<位数>用十进制表示，<基数标识>分别为字母b，o，h或者d，而数字是用所用的基数表示的。例如十进制数2217可以用12位二进制数表示，可以用以下4种形式来表示2217：

12'b100010101001
12'o4251
12'h8A9
12'd2217

表示不指定位数的数不需要具体指定位数。例如，十进制数278可以用以下四种形式表示：

'b100010110
'o426
'h116
278

对表示十进制数而言，可以省略十进制的基数标识d。当在表达式中使用了未指定位数的数时，Verilog编译器会给该数指定一个确定的位数，该位数通常将与表达式中的其他操作数的位数相同。

将减号置于一个数的左侧就可以表示负数。因此，若−5被指定为−4'b101，它将被解释成为：用四位表示的5的二进制补码，即1011。

指定的位数有可能超出表示给定数所需要的位数。在这种情况下，将在给定数的左侧填充0来补足短缺的位数。然而，若表示某个数所需的位数超出了给定的位数，则多余的位数将被忽略。

在表示位数很多的数时，可以使用下划线字符来提高数字的可读性。例如12'b100010101001可以写成12'b1000_1010_1001，这样表示比较容易看清楚。当数字被用于电路设计时，数字就变成由电线传导和由逻辑门操作的信号。到目前为止，我们默认数字电路的信号只有0和1两个逻辑值。在第3章中我们看到信号线也可以为高阻态z。在第4章里，我们有时会用x来表示逻辑值既可能为0，也可能为1的不确定信号。Verilog语言中包含了信号所有四个可能的值：0，1，z，x。符号x用来表示一个未知的逻辑值，它可以是0，1或者z。假定给定Verilog设计中的常量为8'hz3，即等于8'bzzzz0011。若一个数写成8'bx或者8'hx，则其表示一个未知的8位数。z和x不能用在基数为10的表示中。如果有需要填充的情况，通常是0被填充到左侧。但若最左侧的数字是x或者z，则将相应的x或者z填充到左侧。注意信号x和z以及基数标识符b、o、h 和 d 都可以用大写或小写字母表示。

用向量表示的不同位宽的数可以用在算术运算中。假设*A*是8位的向量而*B*是4位的向量，则以下语句：

```
C = A + B;
```

将产生8位的运算结果，即和向量*C*。若*B*是正数，则该结果是正确的。

然而，若*B*是一个用2的补码表示的负数，则运算结果将产生错误。原因是，为了完成加法运算，*B*的左侧将被补0使其扩充为8位向量。要扩充一个数的数位并且保持这个数的值不变，如果这个数是正数话就在它的最高有效位补零；而如果它是负数的话就要在它的最高有效位补1。如此复制符号位的方法叫做*符号位扩展*。因此，为了得到正确的运算结果有必要对*B*进行符号位扩展，这可以用以下拼接语句来实现：

```
C = A + {4{B[3]}, B};
```

上式中4{B[3]}表示位B[3]被复制了四次；相当于写成{B[3]，B[3]，B[3]，B[3]}。这种记法称为*复制操作符*，我们将在第6章的6.6.5节对其进行讨论。若需要从第7位生成进位输出信号，则可以采用图5-32中使用的语句：

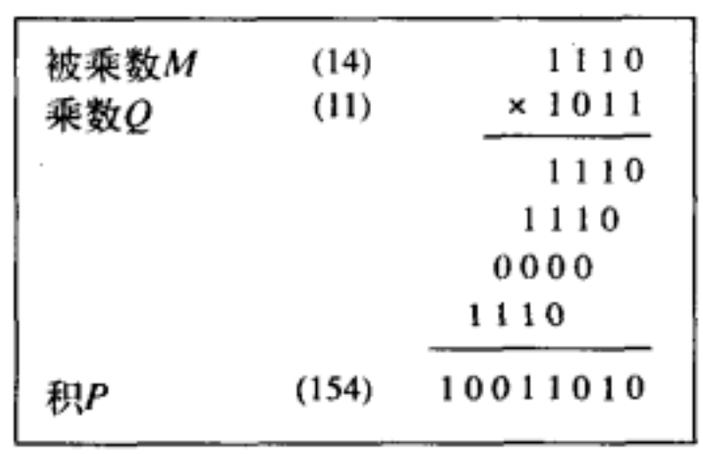
```
C = {1'b0, A} + {1'b0, 4{B[3]}, B};
```

5.6 乘法

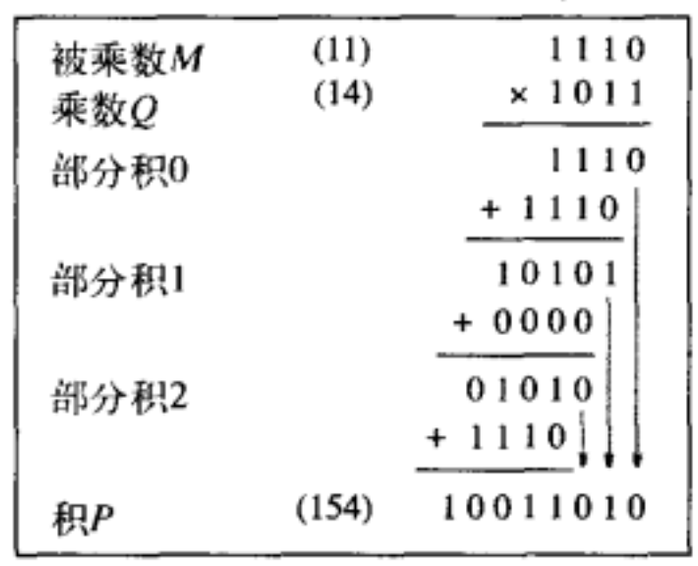
在讨论乘法的一般问题之前，我们应该注意到二进制数B乘以2的情况，这相当于在B最低有效位的右边添加一个0，或者说把B的所有位左移1位。因此若$B = b_{n-1}b_{n-2}\cdots b_1b_0$则$2 \times B = b_{n-1}b_{n-2}\cdots b_1b_0 0$（我们在5.2.3节中早就采用了该方法）。以此类推，B乘以2^k相当于把B左移k位，这对有符号数和无符号数都是正确的。

我们还应当考虑把一个二进制数B右移k位的情形。根据数的位置表示法，这相当于B被2^k除。对于无符号二进制数，这相当于在其最高有效位添加k个0。例如，若B是无符号数，则$B \div 2 = 0b_{n-1}b_{n-2}\cdots b_2b_1$。请注意，右移时$b_0$被丢失了。对于有符号数，右移时应当保持符号位不变。具体做法是：将每一位的值向右移1位而符号位保持不变。假定B是一个有符号数，则$B \div 2 = b_{n-1}b_{n-1}b_{n-2}\cdots b_2b_1$。例如，若$B$= 011000 = $(24)_{10}$，则$B \div 2$ = 001100 = $(12)_{10}$，$B \div 4$ = 000110=$(6)_{10}$。同样，若B = 101000= $-(24)_{10}$，则$B \div 2$ = 110100 = $-(12)_{10}$，$B \div 4$ = 111010 = $-(6)_{10}$。读者也应该观察到，绝对值小的正数，其第一个1的左侧有较多的0；绝对值小的负数，其第一个0的左侧有较多的1。

在完成了上述乘法的特殊情况分析之后，现在我们集中讨论乘法的一般情况。两个十进制数相乘的方法可以用于两个二进制数的相乘，现在先讨论两个无符号数的乘法运算。两个4位二进制数相乘的手工操作过程见图5-35a所示，自右至左地检查乘数的每一位，若是1则把被乘数作适当移位，然后相加以便形成部分积，若是0则不作处理。求出所有作适当移位的被乘数之和就得到我们想要的乘积。请注意，乘积占用的位宽为8位。



a) 手工实现的乘法



b) 硬件实现乘法运算

图5-35 无符号数的乘法

同样的方法可用于设计乘法器电路。为了简化讨论过程，我们仍旧令操作数的长度为4，并且采用以下符号：

$$\text{被乘数}\ M = m_3m_2m_1m_0$$
$$\text{乘数}\ Q = q_3q_2q_1q_0$$
$$\text{乘积}\ P = p_7p_6p_5p_4p_3p_2p_1p_0$$

实现乘法运算的简单方案是顺序操作，使用一个8位加法器实现部分积，部分积的初始值为0。首先检查q_0的值，若$q_0 = 1$，则使M和初始的部分积相加；若$q_0 = 0$，则使0和初始的部分积相加。接着检查q_1的值，如果$q_1 = 1$，则使$2 \times M$和部分积相加，而$2 \times M$的值也很容易获得，只要简单地把M左移1位即可。同样，如果$q_2 = 1$，则使$4 \times M$和部分积相加；若$q_3 = 1$，则使$8 \times M$和部分积相加。我们将在第10章说明如何实现此电路。

上述按照顺序操作的方法使乘法器电路的速度较慢，其原因在于只使用了1个8位加法器来实现部分积以及最终乘积的运算。若使用多个加法器来计算部分积，则可以得到速度快得多的乘法器。

5.6.1 无符号数的阵列乘法器

图5-35b显示了可以用多个加法器来进行乘法运算。其中每一步使用一个四位的加法器计算新的部分积。我们注意到随着计算的进展，低位对后续的加法计算不产生影响。因此它可以被直接送到最终乘积中去，当然，这些最低位也是部分乘积的一部分。

可以用和图5-35b类似的方式设计阵列结构的快速乘法器电路。我们的例子是4×4的结构。乘数和被乘数的符号和以前相同，它们分别为：$M = m_3m_2m_1m_0$，$Q = q_3q_2q_1q_0$。部分积0记作$PP0 = pp0_3\ pp0_2\ pp0_1\ pp0_0$，并且$PP0$等于$M$的每一位和$q_0$相与：

$$PP0 = m_3\,q_0\ \ m_2\,q_0\ \ m_1\,q_0\ \ m_0\,q_0$$

部分积$PP1$等于M的每一位和q_1相与后按以下方式和$PP0$相加：

同样，部分积$PP2$等于M的每一位和q_2相与后按上述方式和$PP1$相加。其余依此类推。

$PP0$:		0	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$
	+	m_3q_1	m_2q_1	m_1q_1	m_0q_1	0
$PP1$:		$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$

实现上述操作的电路被安排成阵列结构，见图5-36a所示。该阵列中有两种基本电路模块，图5-36b所示为位于第1行（顶部）的电路模块，图5-36c所示为第2和第3行的电路模块，用于实现部分积。从图5-36a中可以看出被乘数的移位方式，m_k沿着对角线从一个模块传到另一个模块。每个电路模块中的全加器起到行波进位加法器的作用，计算出自己的部分积。用其他类型的加法器[1]有可能设计出更快的乘法器。

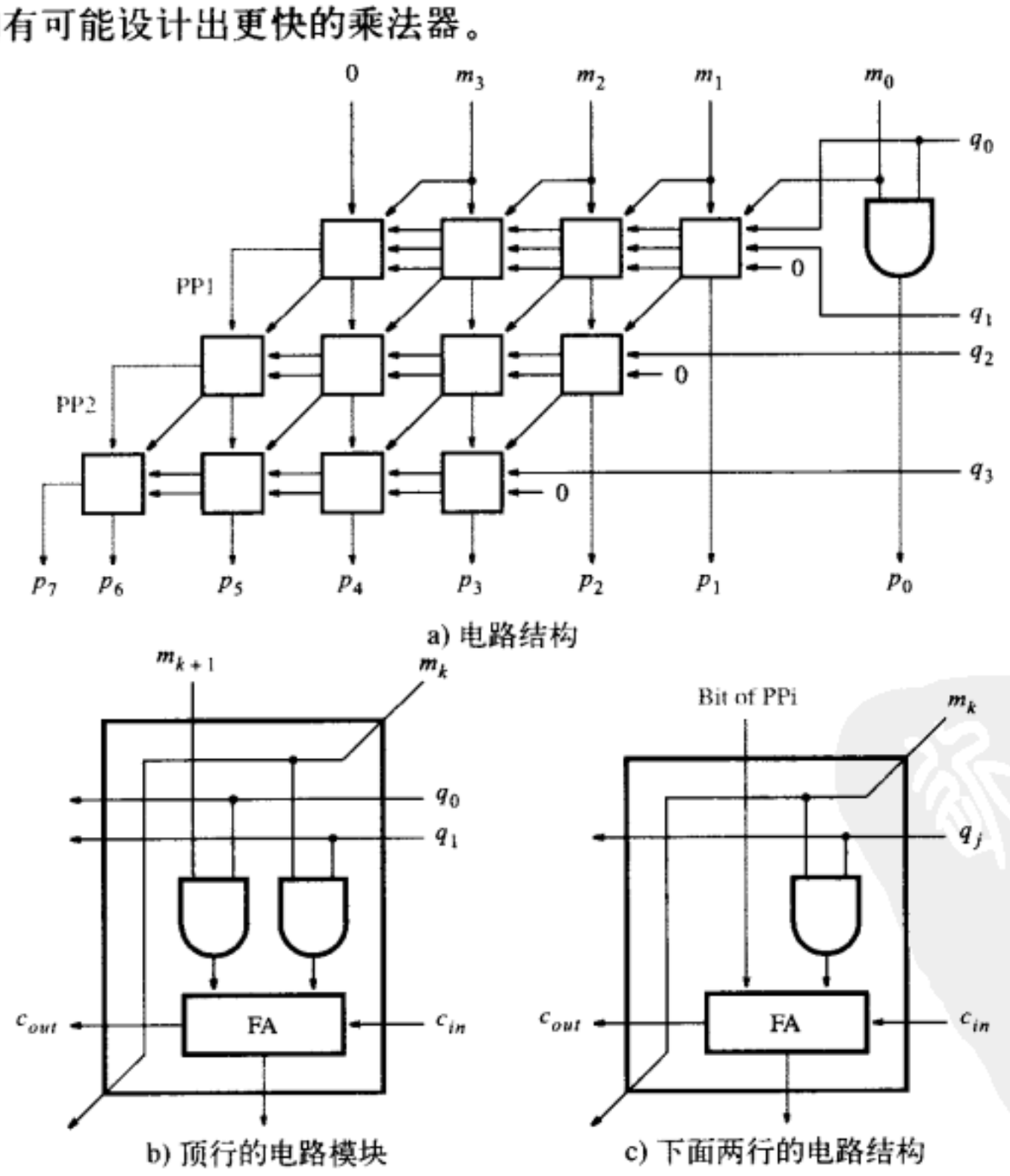


图5-36 4×4乘法器电路

5.6.2 有符号数的乘法

无符号数的乘法说明了乘法器电路设计中的主要问题，有符号数的乘法在某种程度上更复杂一些。

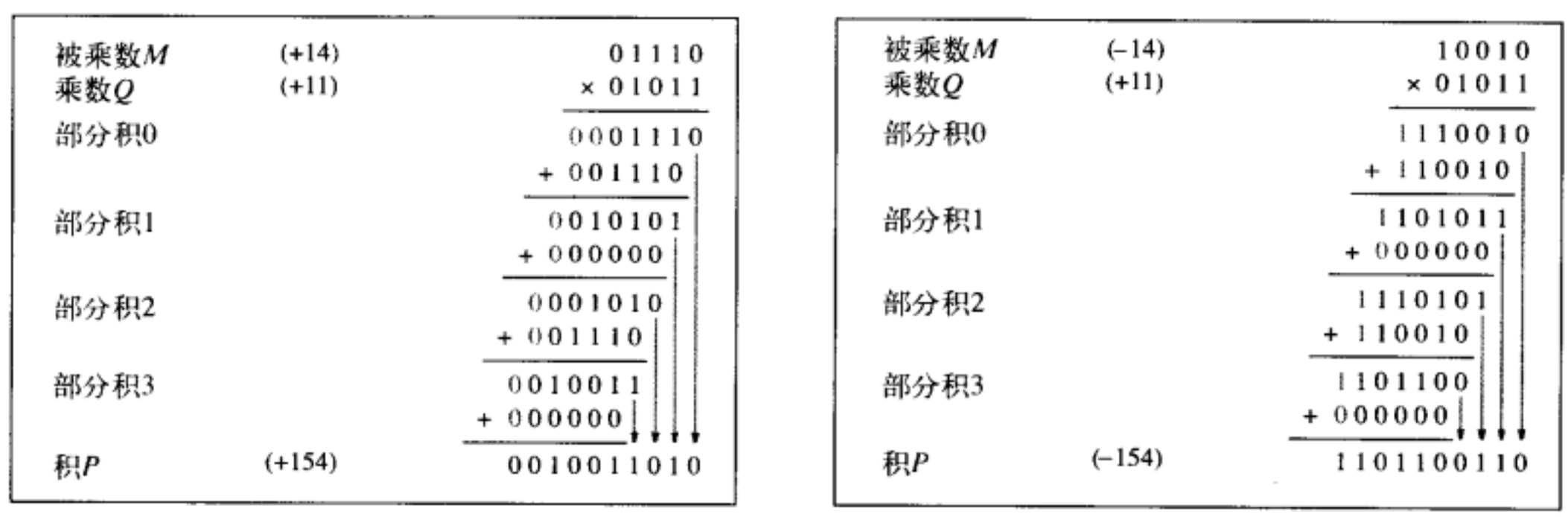
若乘数是正数，就可以用与无符号数乘法器实质上相同的方案实现有符号数的乘法器电路。对乘数取值为1的位，则应当把被乘数适当移位后加到部分积中去，被乘数既可以是正数也可以是负数。

因为被乘数M 适当移位后要加到部分积中去，因此必须正确地表示M 被移位后的数值。例如：假定乘数最右边两位都是1，则第一次加法产生的部分积应当是$PP1 = M + 2M$。若 $M = m_{n-1}m_{n-2}\cdots m_1m_0$，于是$PP1 = m_{n-1}m_{n-2}\cdots m_1m_0 + m_{n-1}m_{n-2}\cdots m_1m_0 0$。实现加法的电路必须是把两个等长的数相加，现在$M$的位宽是$n$，而$2M$的位宽是$(n+1)$，所以必须把$M$的位宽扩展到$(n+1)$。扩展的方法是：把$M$原来的符号位复制一个，添加在$M$的最左边，被扩展后的$M$变为：$M = m_{n-1}\ m_{n-1}\ m_{n-2}\cdots m_1\ m_0$。

上述扩展位宽的方法称为*符号扩展*，它不改变原来的数值。因为对于补码系统来说，在正数的最左侧添加0或在负数的最左侧添加1，不会改变原来的数值。

当被乘数移位后加到部分积中去时，有可能产生溢出位。为了避免发生溢出，部分积的位宽必须增加1位。图5-37a的例子说明了两个正数相乘的过程，扩展的符号位用灰色标记。图5-37b中的被乘数是负数。请注意，两种情况下乘积的位宽都是$2n$。

若乘数是负数，可以是把乘数和被乘数都转换成2的补码，因为这样做不会改变乘积的值。因此可以用正数乘法器的方案稍作修改便可以完成有符号数的乘法。



a) 被乘数为正数的乘法　　b) 被乘数为负数的乘法

图5-37 有符号数的乘法运算

这里给出了相对简单的有符号数乘法器的一种方案。还存在其他一些更复杂但效率更高的技术，有兴趣的读者可参考文献[1]。

我们已经讨论了进行加法、减法和乘法运算的电路，计算机中还需要进行除法运算的电路，但除法电路比较复杂，在此不作介绍。在第10章中将给出实现除法电路的例子，读者可以在计算机组成原理的书中或参考文献[1]和[2]中找到实现除法运算的各种技术。

5.7 数的其他表示方法

在前面几节中，我们介绍了用数位法表示的二进制整数。在数字系统中还用到了其他类型的数的表示方法。本节将简要地介绍其他三种类型的表示方法：定点数、浮点数和二进制编码的十进制数（BCD）。

5.7.1 定点数

定点数由整数和小数两部分组成，可以用数位表示法把它写为：

$$B = b_{n-1}\, b_{n-2} \cdots b_1\, b_0 \,.\, b_{-1}\, b_{-2} \cdots b_{-k}$$

该数的值为：

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

由于小数点的位置被固定在某处，因此称做定点数。若没有明确地指定小数点的位置，则假定在最低位的右边，此时定点数变为整数。

处理定点数的逻辑电路与处理整数的逻辑电路本质上是相同的。我们将不单独对其进行讨论。

5.7.2 浮点数

定点数的表示范围受限于该数的有效位数。例如，我们用1个符号和8位数字来表示十进制数，则其数值范围为0到±999 999 99。如果用8位数字表示小数，则可表示的数范围为0.000 000 01到±0.999 999 99。科学计算中经常要涉及数值很大或者数值很小的数，如果用定点数，所需要的位数会很多，因而用浮点数比较好。浮点数的表示形式为：

$$Mantissa \times R^{Exponent}$$

*Mantissa*代表整数，*R*代表基数，*Exponent*代表指数（或称阶码）。浮点数通常用格式化的形式表示，即小数点放在第一个非0数字的右边，例如5.234×10^{43}或6.31×10^{-28}。

IEEE为二进制浮点数制定了标准[3]，标准中规定了单精度（32位）浮点数和双精度浮点数（64位）的格式，见图5-38所示。

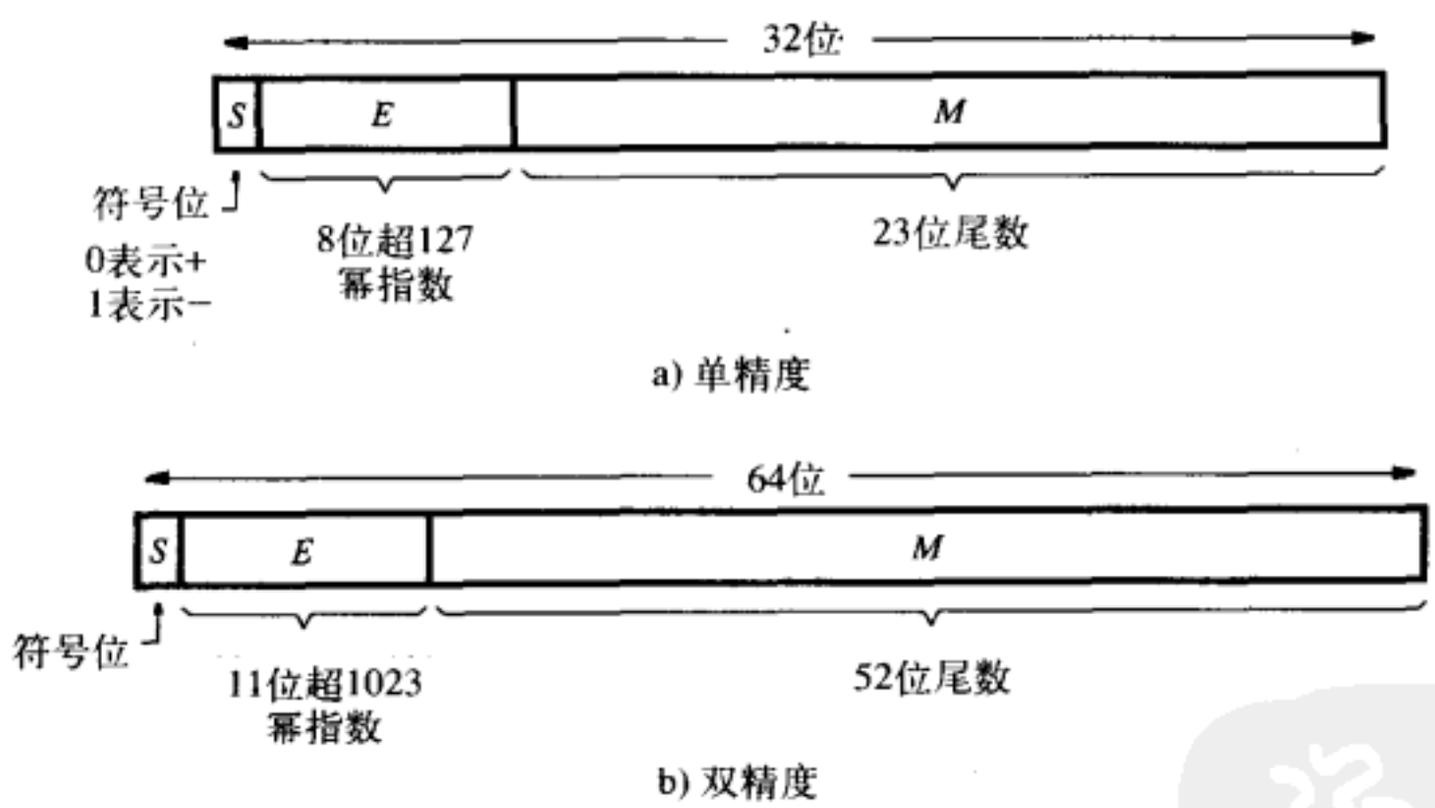


图5-38 浮点数的IEEE标准

单精度浮点数的格式

图5-38a描述了单精度浮点数的格式。最左边的1位是符号位（*S*），0表示正数，1表示负数。尾数域（*M*）23位，指数域（*E*）8位，和指数相关的基数（*R*）是2。因为浮点数既要能表示很大的数又要表示很小的数，所以指数可以是正数也可以是负数。位宽8位的有符号数的数值范围是−128~+127，IEEE标准不用普通的有符号数表示指数，而是把实际的指数值加上127后记为*E*，因此：

$$Exponent = E - 127$$

以此种方式表示的E是一个正数，E的数值范围为0～255。我们用E=0代表精确为0，并用E=255代表无穷大，因此E取1～254用于表示一整数，即一般数的指数范围在 −126～127之间。上述表示法有利于浮点数的加法和减法运算，因为加法/减法运算的第一步是比较指数的大小，以便确定尾数是否需要移位以及加/减有效位。

尾数域位宽23位，IEEE标准要求尾数取规格化形式，即尾数的最高位必须等于1，因此尾数部分的最高位就可以不必明确地在尾数域中表示出来。于是，如果M是尾数域中的位向量，则尾数的实际值应当是1.M。也就是说，尾数域23位，但能表示24位的尾数值。对于图5-38a所示的浮点数格式，它所表示的数值$Value$为：

$$Value = \pm 1.M \times 2^{E-127}$$

在这种表示方式下，尾数的精度大约是7位十进制数，指数区的范围在2^{-126}到2^{127}之间，大约相当于$10^{\pm 38}$。

双精度浮点数的格式

图5-38b所描绘的是双精度浮点数的格式，其位宽是64位，指数域和尾数域都增加了位宽，所表示的数在范围和精度上都有很大提高。指数域位宽11位，采用了超1023格式，即：

$$Exponent = E - 1023$$

E的数值范围为0到2047。我们用$E = 0$代表精确为0，并用$E = 2047$代表无穷大，因此E取1到2046用于表示一般数，即表示数的指数范围为 −1022到1023之间。

尾数域位宽52位，因此IEEE标准要求尾数取格式化形式，尾数的实际值应当是1.M。对于图5-38b所示的浮点数格式，它所表示的数值是：

$$Value = \pm 1.M \times 2^{E-1023}$$

用上述表示方式，尾数的精度大约是16位十进制数，指数区的范围大约在$10^{\pm 308}$之间。

浮点数的算术运算远比有符号数的算术运算复杂得多。因为浮点数的算术运算属于相当专业的领域，我们将不再深入讨论有关的逻辑电路设计问题，有兴趣的同学可以参阅文献[1]和[2]。

5.7.3 二−十进制编码

数字系统也可以采用二−十进制编码（简称BCD）来表示十进制数，即把每种十进制数码用二进制数形式表示。因为十进制数有10个数码（0,1,2, 3,4,5,6,7,8,9），所以每种十进制数码需要4位无符号的二进制数表示。十进制数码可用无符号二进制数表示，其对应关系如表5-3所示。由表看来，由于4位二进制代码可以有16种状态，而BCD编码只需要其中的10种状态，所以剩下的6种状态不会在逻辑电路中出现，在设计过程中通常将其当作无关项对待。某些早期计算机和手持式计数器可能采用BCD码，其主要优点在于这种编码风格容易被简单的面向十进制数的显示器显示。它的缺点在于，对BCD码进行算术运算的电路比较复杂，此外，这种编码方式浪费了16种状态中的6种。

表5-3 BCD编码

十进制数码	BCD码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

虽然BCD编码方式的重要性已经不如从前，但有时还是会遇到。为了说明BCD编码所需的电路的复杂性，我们将稍微详细地讨论BCD码的加法运算。

BCD码的加法运算

两个BCD码相加的和有可能大于9，此时必须对其进行校正。设

$$X = x_3x_2x_1x_0 \quad Y = y_3y_2y_1y_0 \quad S = s_3s_2s_1s_0$$

X和Y分别代表两个BCD码形式的数，S代表这两个数相加之和，$S = X+Y$。若$X+Y \leqslant 9$，则此时加法操作和4位无符号二进制数加法操作相同；若 $X+Y > 9$，则此加法结果需要两位BCD码来表示。此外，从4位加法器来的和Sum也可能错误。

BCD码加法运算需要在以下两种情况下施加校正操作：（1）和大于9小于15，用4位表示时无进位产生。（2）和大于15，用4位表示时有进位产生。在图5-39所示的第一种情况中，7+5 = 12 = Z，正确的结果应当是和$S = 2$，进位输出为1。出现这种情况的原因是：4位加法器是以16为模，而十进制加法运算应当以10为模。因此当BCD码加法之和超过9时，其校正应当是对4位加法器的结果再加上6，计算过程如下所示：

$$Z = X + Y$$

若$Z \leqslant 9$，则 $S = Z$ 且进位 = 0

若$Z > 9$，则 $S = Z + 6$ 且进位 = 1

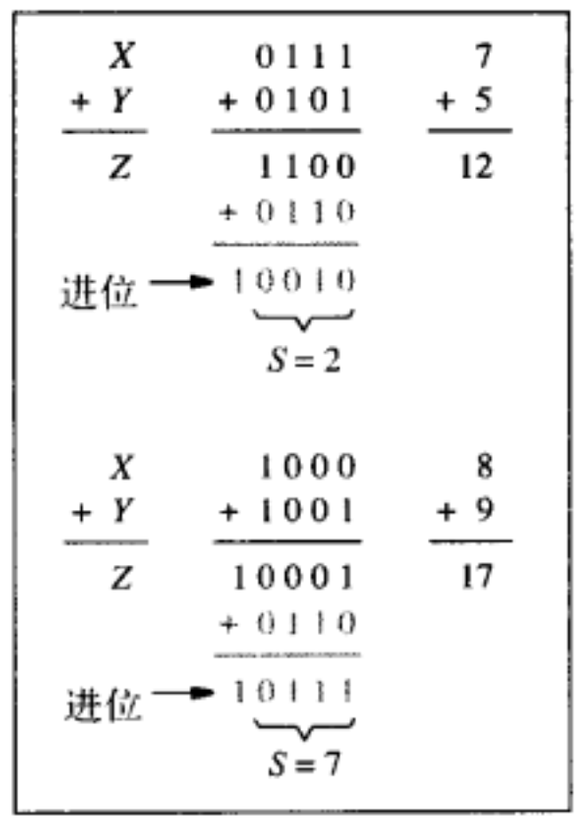


图5-39 BCD数字的加法运算

图5-39所示的第二种情况是$X+Y > 15$，此时4位加法器低4位（Z）的值是1，它不是正确的和（S）。但4位加法器产生输出（表示16），考虑此因素而采取的必要的校正是：对和的中间值Z再加上6，即$S=Z+6$。

图5-40所示为基于以上原则建立的1位BCD码加法器方块图。图中有一个检测块，当$Z > 9$时，该检测块的输出信号Adjust = 1，它作为多路器MUX选择端的输入信号，控制是否需要提供校正操作。第二个4位加法器产生正确的和（S），若Adjust = 0，则$S=Z+0$。若Adjust = 1，则$S=Z + 6$ 且*carry-out* = 1。

实现此框图的Verilog代码示于图5-41。输入信号X和Y以及输出信号S都被定义为位宽4位的数，中间和Z被定义为位宽5位的数。**if-else**语句用来提供上述的校正，因此不必使用具体的校正信号。

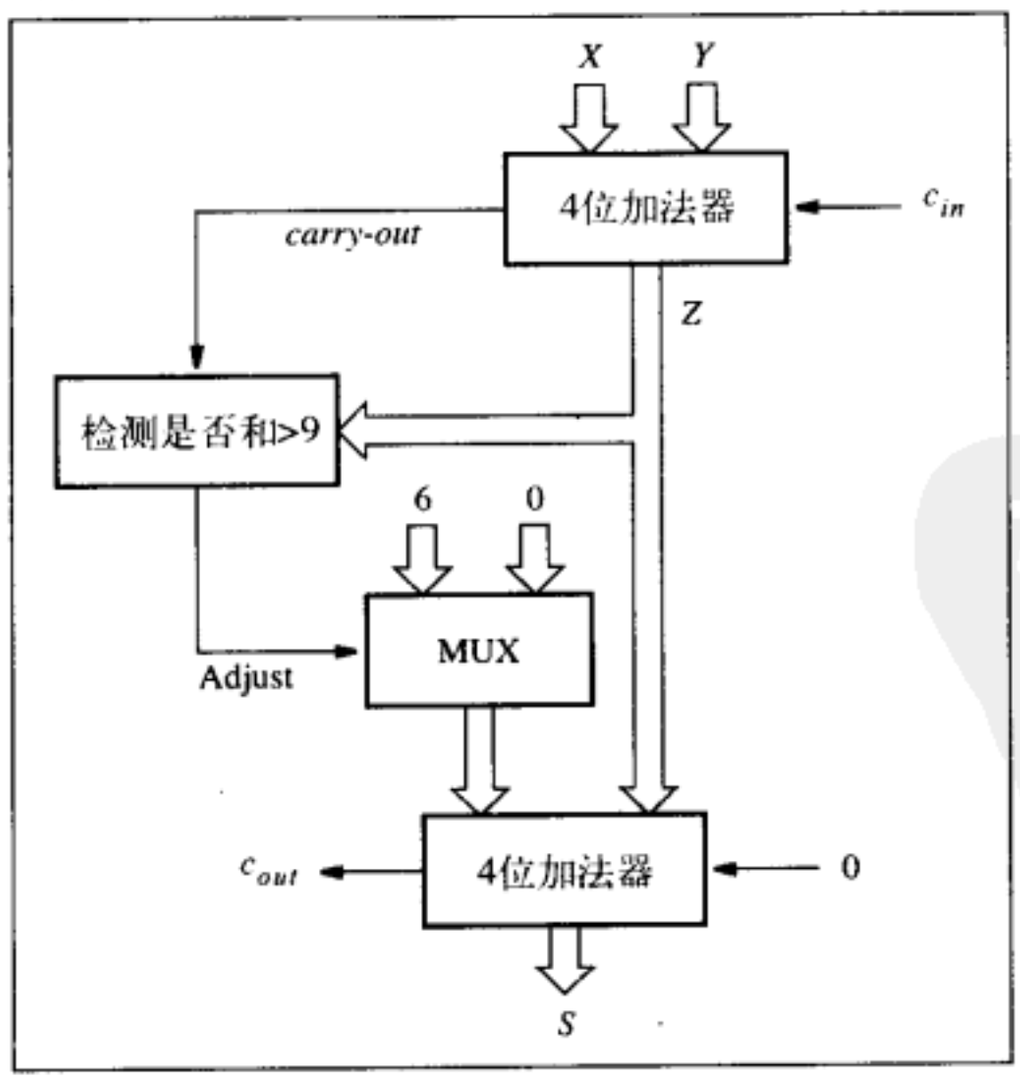


图5-40 1位BCD码加法器框图

```
module bcdadd (Cin, X, Y, S, Cout);
   input Cin;
   input [3:0] X, Y;
   output reg [3:0] S;
   output reg Cout;
   reg [4:0] Z;

   always @(X, Y, Cin)
   begin
      Z = X + Y + Cin;
      if (Z < 10)
         {Cout, S} = Z;
      else
         {Cout, S} = Z + 6;
   end

endmodule
```

图5-41 1位BCD码加法器的Verilog代码

以上是使用Verilog代码描述图5-40所示框图的行为，如果我们希望用手工设计方法推导出相应的电路，则可以采用下面的方法。首先我们需要定义校正函数：当求和的中间值Z大于9时，则校正信号Adjust应当为真，否则为0。用逻辑表达式表示如下：

$$\text{Adjust} = carry\text{-}out + z_3(z_2 + z_1)$$

校正操作是对z加6，我们可以不必使用一个完整的4位加法器实现校正操作，而采用较为简单的电路。请注意：加6操作对和的最低位s_0没有影响，因而 $s_0 = z_0$。只用一个2位加法器就可以求出s_2和s_1，另外，若此2位加法器的进位为0，则$s_3 = z_3$，如果进位为1，则$s_3 = \overline{z}_3$。图5-42是实现此方案的完整电路，该1位BCD加法器可作为一个基本模块。用它实现更大规模的BCD加法器，其方法和使用1位全加器组成规模更大的行波进位加法器相同。

BCD数的减法计算也可以用基数补码的原理处理。正如在二进制数系统中用2的补码处理二进制数一样，十进制数也可以用10的补码进行处理。这个问题留给读者作为作业（见习题5.19）。

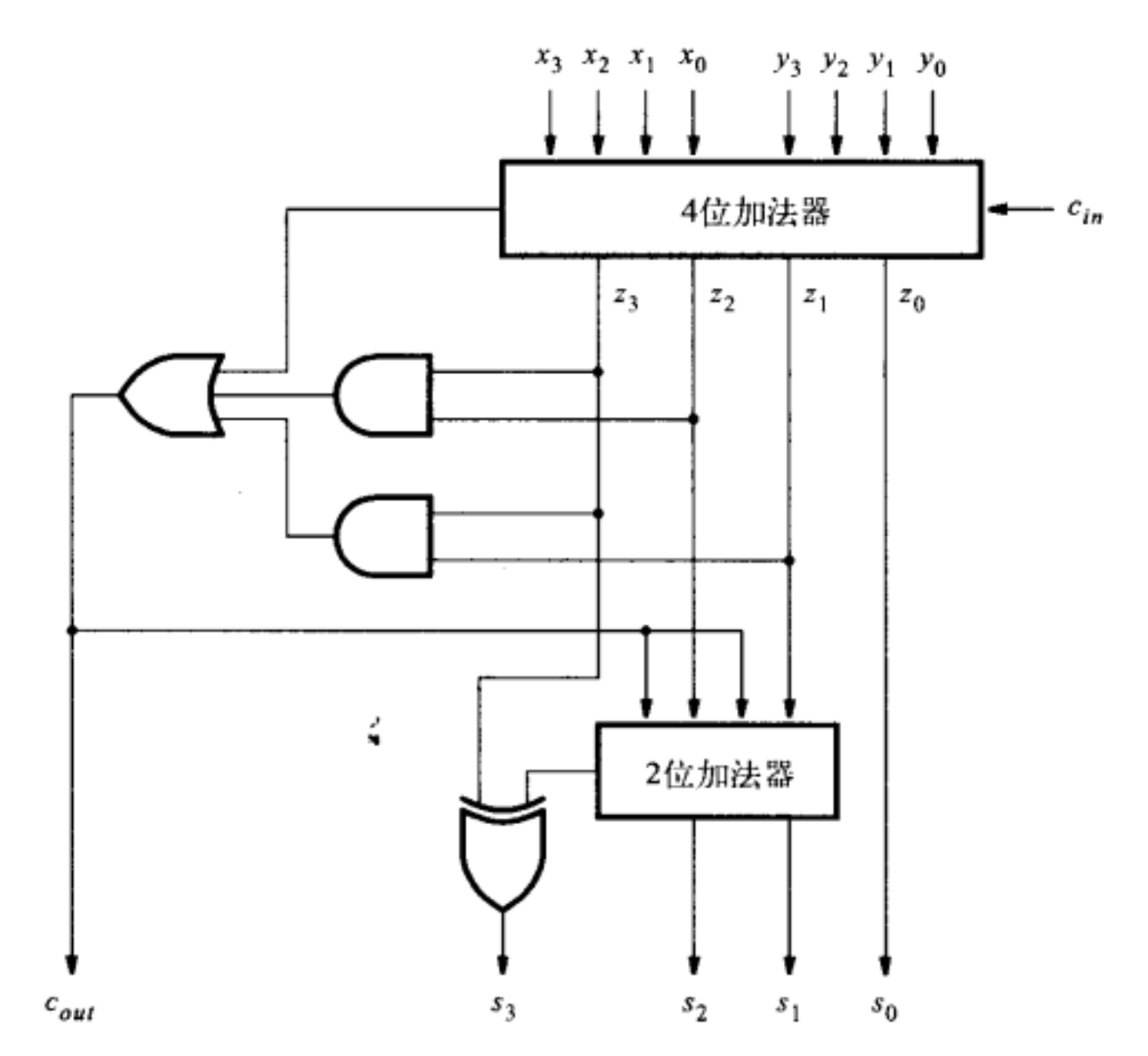


图5-42 1位BCD码加法器电路

5.8 ASCII字符码

ASCII代码是数字系统中用于表示字母、数字以及控制字符的最常用的标准代码，表5-4列出了这个标准代码的编码表。

ASCII代码用位宽为7位的字段表示128种字符。其中10个字符用于代表十进制数字0~9，其ASCII代码的高3位相同（$b_6b_5b_4 = 011$），而低4位（b_{3-0}）的编码不同，而且采用了二进制的编码方式。大写和小写字母的编码原则是易于文本信息的检索，字母A~Z的编码按照升序排列，因此在检索字母或字词时，对其相应的ASCII代码作简单的算术比较运算即可。

表示字母或者数字的字符称为字母-数字（alphanumeric）字符。ASCII字符除了包含字母-数字字符外，还包含标点字符（例如！和？）、常用符号（例如 $和%）以及一套控制字符。控制字符是计算机系统处理数据和传送设备之间的数据所需要使用的字符，例如回车符（CR）。回车符CR用于表示输出设备（例如打印机的滑动架或显示器的光标）应当回到最左边

的一列。

表5-4 7位ASCII码

位的位置	位的位置654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgement
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACQ	Acknowledgement	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

码格式位的位置 = 6 5 4 3 2 1 0

ASCII代码用于处理文本信息。ASCII代码也可以表示数字，但这种编码的操作数用于算术运算时却很不方便。因为这个原因，所以在作算术运算时最好把ASCII字符编码的数字，转化为前面讨论过的二进制形式。

ASCII标准使用位宽为7位的字段对字符编码。计算机系统中最常用的是字节，字节的位宽是8位。把ASCII代码填入一个字节通常采用以下两种方法：第一种方法是将字节的最高位（b_7）设置为0；第二种方法是将最高位设置成其他7位的奇偶校验，即用最高位来表明后面7位中1的个数是奇数还是偶数。

奇偶校验

在数字系统中，奇偶校验常用于查错。当数字信息从甲地传送到乙地（有可能经过长线）时，数据中某个位的值可能在传输的过程中受到破坏。例如，发送方发出的数据中某位取值应该为1，而接收方收到的数据却是0。假设某个数据由n位1/0信号组成，最简单的差错校验方式是增加一个位p，用p表明n位数据的奇偶性，从而使被传输的信息变为$n+1$位。奇偶校验的方法有两种：奇校验和偶校验。若使用偶校验，则应根据发送的n位中1的个数设置p的值，

使被传输的（n+1）位信息中取值为1的个数为偶数。若使用奇校验，则应根据发送的n位中1的个数设置P的值，使被传送的（n+1）位信息中取值为1的个数为奇数。发送方根据被传输的n位数据项产生p的值，接收方则根据接收到的（n+1）位信息（的奇偶性是否与预期的相同）来判断收到的数据是否正确。

奇偶位的产生以及奇偶校验电路可以用异或门实现。例如，设被传输的数据由$x_3x_2x_1x_0$四位组成，则偶校验位p可以表示为：

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

接收方的校验电路的逻辑表达式为：

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

若$c = 0$，表示接收到的数据项满足偶特性要求；若$c = 1$，则表示发生了传输错误。请注意，$c = 0$并不保证没有错误，如果在传输过程中有2位或者偶数位的值出现取反的错误，仍然满足偶特性的要求，但是这种错误是检测不出来的。相反，若在传送过程中有奇数个位的取值出现错误，则一定可以检测出来。

奇偶校验的吸引力在于它的简单性。关于更可靠检测机制的复杂方案请参阅有关的文献[4]。在9.3节中，我们将再次讨论奇偶校验电路。

5.9 问题求解举例

本节介绍一些读者可能会遇到的典型问题，且展示这一类问题如何求解。

例5.7 将十进制数 14959 转换成十六进制数表示。

解：通过连续地除以16，每一次得到余数就是十六进制数字，这样就可以将一个整型数转变为十六进制数。为了理解为什么这样做是正确的，考虑一个四位数 $H = h_3h_2h_1h_0$，其值可用下面的表达式表示：

$$V = h_3 \times 16^3 + h_2 \times 16^2 + h_1 \times 16 + h_0$$

若将V除以16，则得到下面的表达式：

$$\frac{V}{16} = h_3 \times 16^2 + h_2 \times 16 + h_1 + \frac{h_0}{16}$$

由上式，可以知道得到的余数就是h_0。图5-43展示了进行 $(14959)_{10} = (3A6F)_{16}$ 转换所需要的步骤。

转换 $(14959)_{10}$

			余数	Hex数字	
14959 ÷ 16	=	934	15	F	LSB
934 ÷ 16	=	58	6	6	
58 ÷ 16	=	3	10	A	
3 ÷ 16	=	0	3	3	MSB

转换的结果是$(3A6F)_{16}$

图5-43 从十进制到十六进制的转换

例5.8 将十进制小数0.8254转换成二进制数表示。

解：正如5.7.1节所表明的那样，二进制小数被表示为位图$B = 0.b_{-1}b_{-2}\cdots b_{-m}$，其值为

$$V = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

将上述表达式乘以2，得到如下表达式：

$$b_{-1} + b_{-2} \times 2^{-1} + \cdots + b_{-m} \times 2^{-(m-1)}$$

在上述表达式中，最左边的一项是小数点右边的第1位。余下的几项构成了另外一个二进制小数，我们可以用同样的方法处理。因此把一个十进制小数转换成二进制小数，只需把十进制数乘以2，若乘积小于1，则将被计算的位设置为0，若乘积大于1，则将被计算的位设置为1。重复这个计算过程直到得到足够多的位数，以满足精度的要求。请注意，不可能用一个完全精确的二进制小数来表示一个十进制小数。图5-44展示了产生$(0.8254)_{10}=(0.11010011\cdots)_2$转换所必需的计算过程。

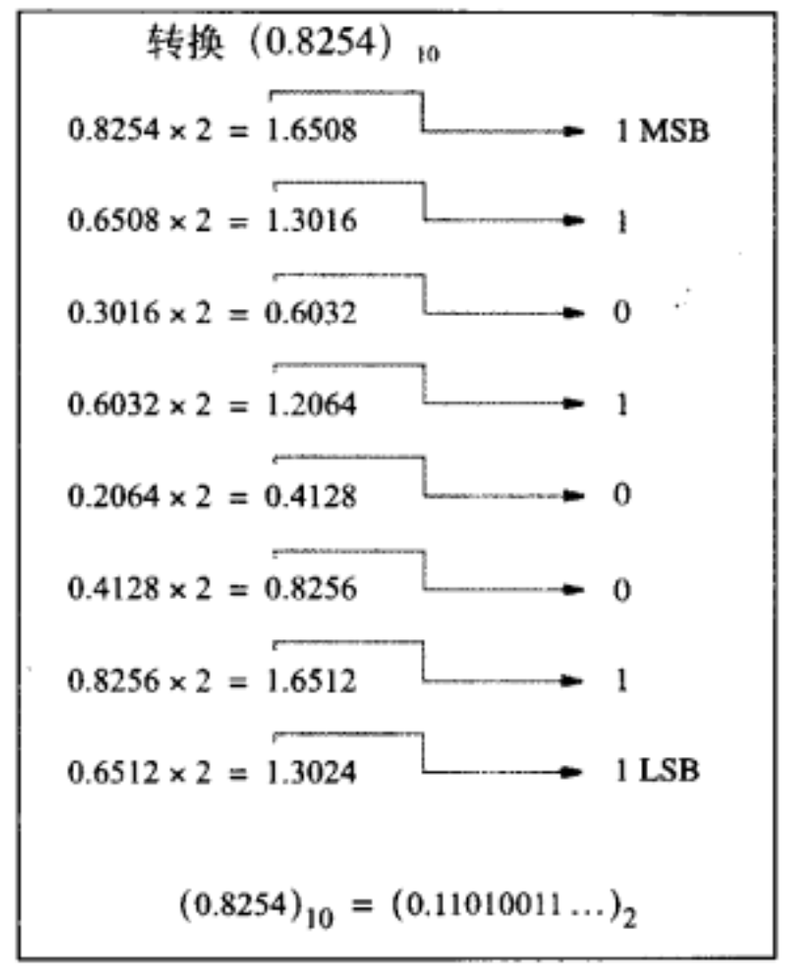


图5-44 把十进制小数转换成二进制小数

例5.9 将十进制定点数214.45转换成二进制定点数。

解：如图5-1所解释的那样，对该定点数的整数部分连续地除以2，而对该数的小数部分则连续地乘以2。完整的计算过程如图5-45所示，求得的结果为：$(214.45)_{10}=(11010110.0111001\cdots)_2$。

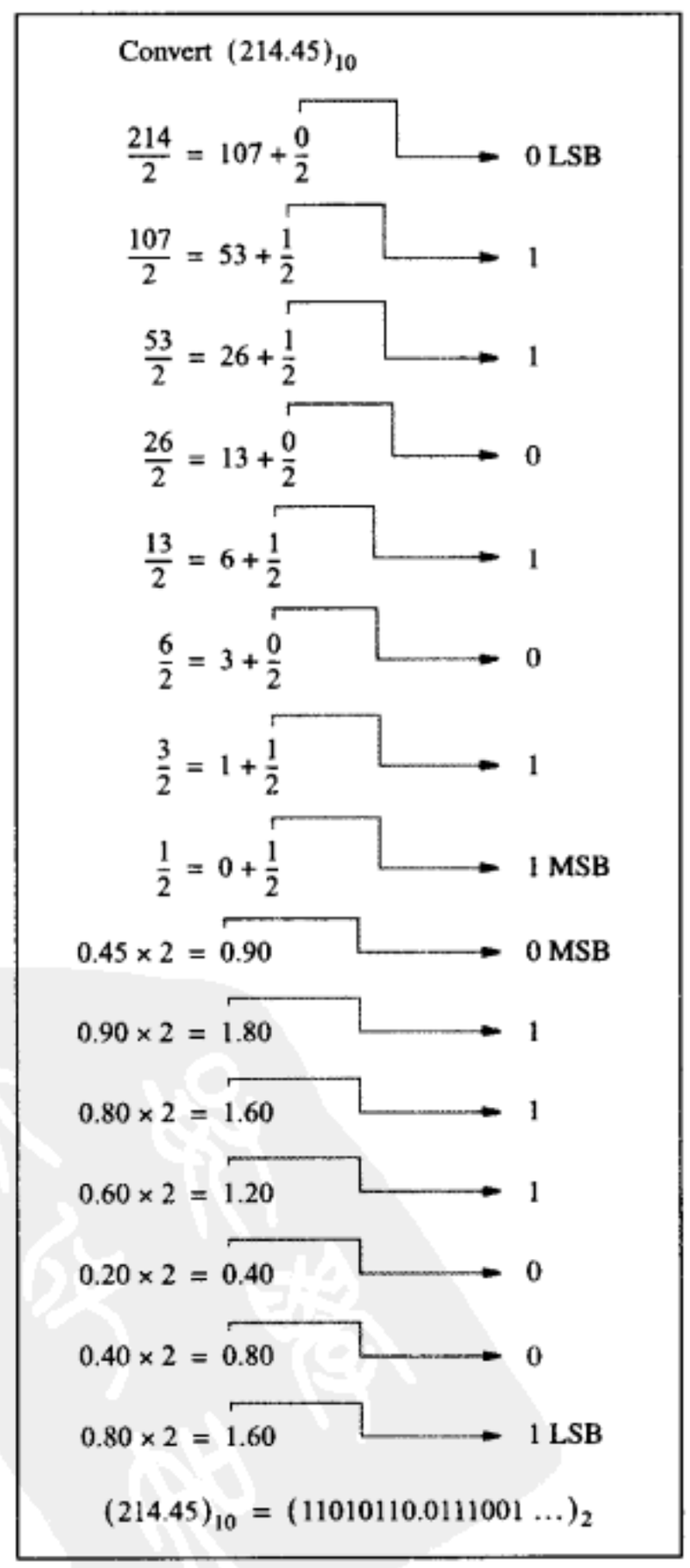


图5-45 把十进制定点数转换成二进制定点数

例5.10 在计算机处理数据的过程中，经常需要对数字进行比较。可以用图5-46所示的减法器电路对两个四位的带符号的数字$X=x_3x_2x_1x_0$和$Y=y_3y_2y_1y_0$进行比较。该减法器执行$X-Y$的操作。产生的三个输出表示如下：

- 若减的结果为0，则$Z=1$；否则$Z=0$。
- 若减的结果为负，则$N=1$；否则$N=0$。
- 若发生算术溢出，则$V=1$；否则$V=0$。

展示如何利用Z, N和V来确定$X=Y$，$X<Y$，$X\leqslant Y$，$X>Y$，$X\geqslant Y$五种情况。

解：首先考虑$X<Y$的情况，此时有下列可能性：

- 若X和Y的符号相同，且没有发生溢出，即$V=0$，则无论X和Y为正还是为负，其差将为负（即$N=1$）。
- 若X为负，Y为正，且没有发生溢出（即$V=0$），则差将为负（即$N=1$）；若发生溢出（即$V=1$），则差将为正（即$N=0$）。

因此，若$X<Y$，则$N\oplus V=1$。

$X=Y$的情况可以由$Z=1$来检测。$X\leqslant Y$的情况可以由$Z+(N\oplus V)=1$来检测。

最后两种情形只是简单地反相：若$\overline{Z+(N\oplus V)}=1$，则$X>Y$；若$\overline{N\oplus V}=1$，则$X\geqslant Y$。

例5.11 编写如图5-46所示电路的Verilog代码。

解：正如图5-47的代码所表明的那样，可以用图5-27所示的代码来确定该电路。请注意以下语句：

```
assign Z = !S;
```

它表示四位或非（*NOR*）逻辑功能，即

$$Z = \overline{s_3 + s_2 + s_1 + s_0}$$

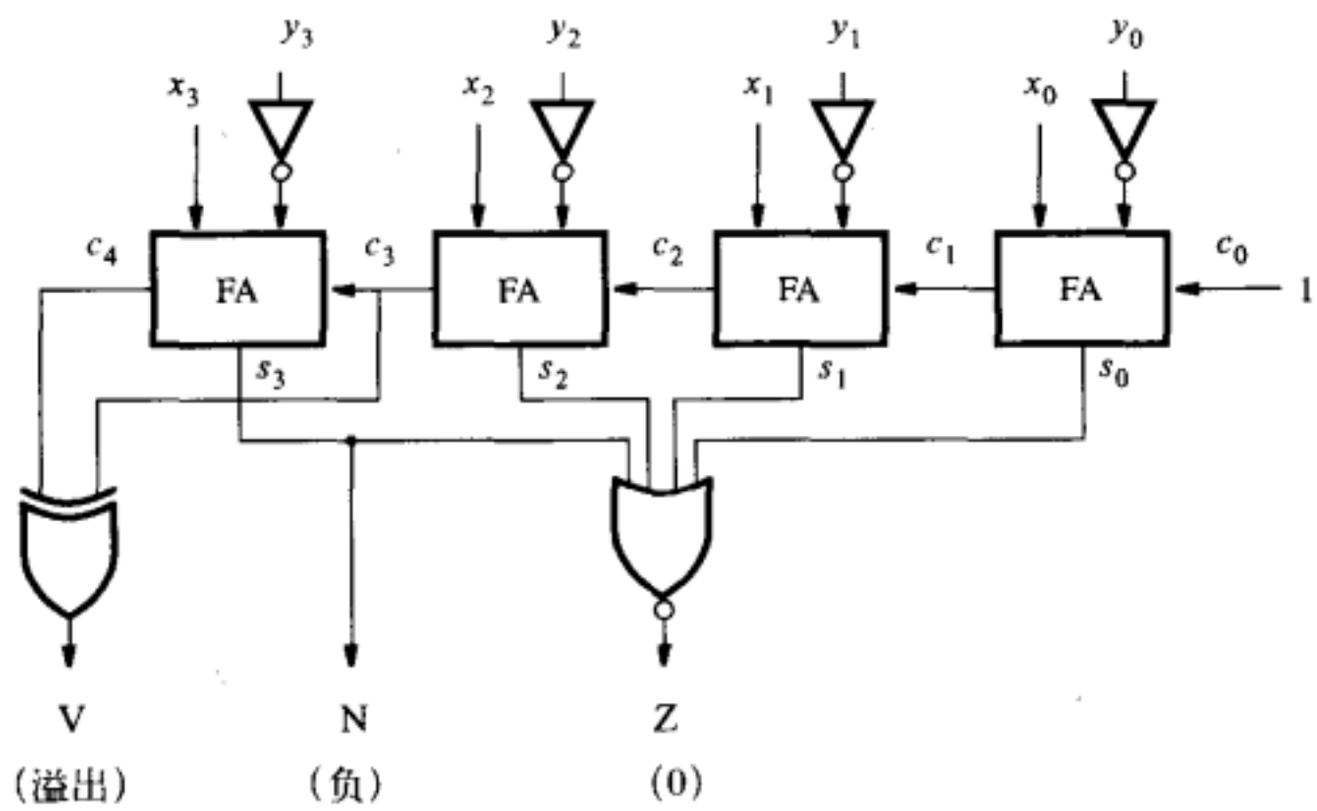


图5-46 一个比较器电路

当电路的规模较大时，如在比较器的操作数是32位的场合，仍旧每次单个地引用全加器显然很不方便。此时可以采用另外一种办法，即使用如图5-28所示的用循环语句自动生成多个全加器实例代码的方法，如图5-48所示。

```
module comparator (X, Y, V, N, Z);
   input [3:0] X, Y;
   output V, N, Z;
   wire [3:0] S;
   wire [4:1] C;

   fulladd stage0 (1'b1, X[0], ~Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], ~Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], ~Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], ~Y[3], S[3], C[4]);
   assign V = C[4] ^ C[3];
   assign N = S[3];
   assign Z = !S;

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

图5-47 比较器电路的结构化Verilog代码

```
module comparator (X, Y, V, N, Z);
   parameter n = 32;
   input [n-1:0] X, Y;
   output reg V, N, Z;
   reg [n-1:0] S;
   reg [n:0] C;
   integer k;

   always @(X, Y)
   begin
      C[0] = 1'b1;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ ~Y[k] ^ C[k];
         C[k+1] = (X[k] & ~Y[k]) | (X[k] & C[k]) | (~Y[k] & C[k]);
      end
      V = C[n] ^ C[n-1];
      N = S[n-1];
      Z = !S;
   end

endmodule
```

图5-48 比较器电路的Verilog代码

例5.12 图5-36所示的是一个四位乘法器电路。每一行由四个以逐位进位配置互相连接的全加器（FA）块组成。在每一行中，由进位信号逐位进位所造成的延迟对产生乘积输出所需要的时间有显著的影响。为了加快该电路的运行速度，可以使用如图5-49所示的电路结构。在这个电路中，给定行的进位信号被“存储”起来，且在下一行确切位的位置将其包括进来。

然后，在第一行，全加器可以用来对三个适当移位的由乘数选取的被乘数位进行相加运算。举例说明如下：在第2位处，这三个输入为m_2q_0，m_1q_1和m_0q_2。在最后一行，仍有必要使用逐位进位加法器。由这种方式连接的全加器阵列组成的电路称为*进位-存储*（*carry-save*）加法器阵列。

与图5-36所示的电路比较，图5-49所示电路的总延迟是多少？

解：在图5-36a所示的电路中，最长的路径是从第一行最右边的两个全加器到第二行最右边的两个全加器，再到最下面一行的所有四个全加器。因此该乘法器的延迟等于全加器块延迟时间的8倍，并加上形成到第一行第一个全加器的输入所必需的与门延迟。这几个延迟结合起来成为确定乘法器电路运行临界速度的延迟。

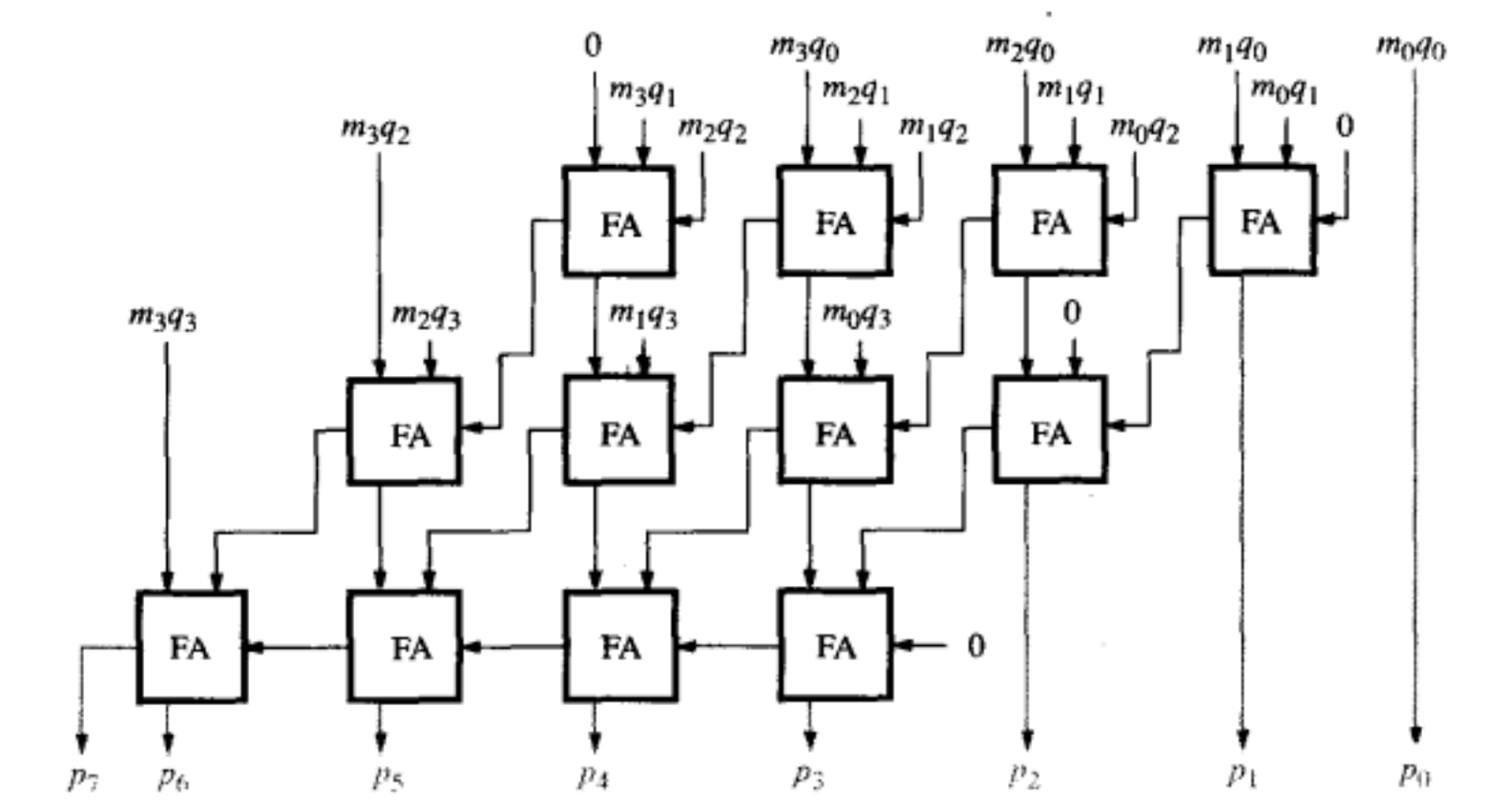


图5-49 进位-存储阵列乘法器

在图5-49所示的电路中，最长的路径是从第一行和第二行最右边的全加器到最下面一行的所有四个全加器。因此临界延迟等于全加器块延迟时间的6倍，并加上形成到第一行第一个全加器的输入所必需的与门延迟。

练习题

标有*号的习题答案参见书后。

* **5.1** 确定下列无符号数的十进制数值：

(a) $(0111011110)_2$

(b) $(1011100111)_2$

(c) $(3751)_8$

(d) $(A25F)_{16}$

(e) $(F0F0)_{16}$

* **5.2** 确定下列1的补码（反码）的十进制数值：

(a) 0111011110

(b) 1011100111

(c) 1111111110

* **5.3** 确定下列2的补码的十进制数值：

(a) 0111011110

(b) 1011100111

(c) 1111111110

* **5.4** 将十进制数73，1906，−95以及−1630分别转化为用以下形式表示的有符号的12位数：

(a) 符号和幅值

(b) 1的补码

(c) 2的补码

5.5 对以下所示8位2的补码进行算术运算，指出是否发生了溢出。把这些2的补码转化为十进制数的符号和幅值形式，检查算术运算的结果是否正确。

```
  00110110      01110101      11011111
+ 01000101    + 11011110    + 10111000
----------    ----------    ----------
  00110110      01110101      11010011
- 00101011    - 11010110    - 11101100
```

5.6 证明异或运算满足结合律，即$x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$。

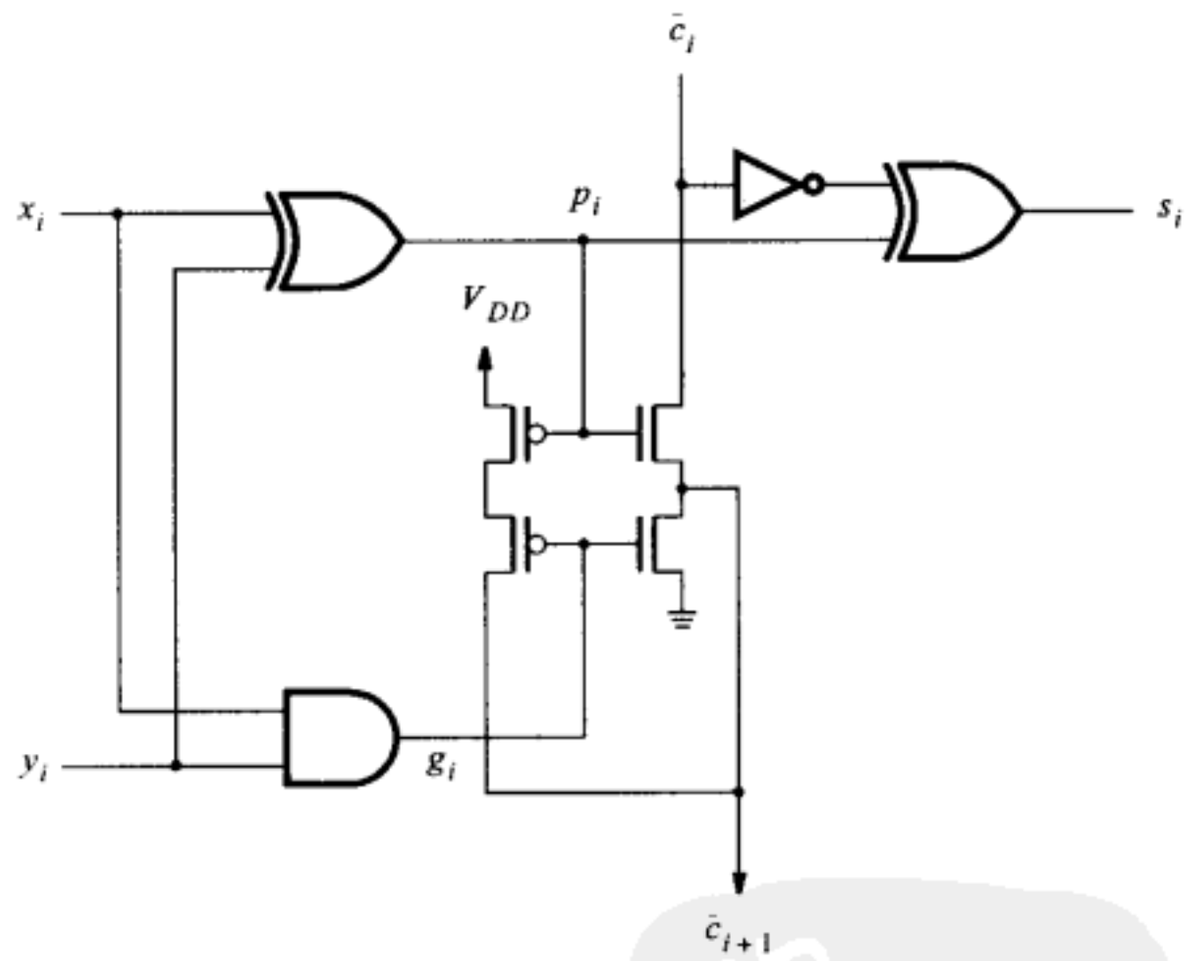
5.7 请说明图5-5所示电路实现了图5-4a所示的全加器功能。

5.8 5.3节中给出了求二进制数2的补码的规则：即自右至左地扫描该二进制数，将所遇到的0和第一个1保持不变，剩下的各位一律求反。请证明此规则的正确性。

5.9 判断n位有符号数相加是否溢出的表达式为：Overflow $= c_n \oplus c_{n-1}$，请证明其正确性。

5.10 在5.5.4节中，我们声明了加法器电路中从$k-1$位输出的进位信号c_k的逻辑表达式为：$c_k = x_k \oplus y_k \oplus s_k$，其中 x_k 和y_k 是输入，s_k是和。请验证此表达式的正确性。

* **5.11** 考虑图P5-1所示电路，该电路能作为行波进位加法器的一个基本单元吗？讨论其优缺点。



图P5-1 习题5.11的图

* **5.12** 请确定n位超前进位加法器所用的门数，假定没有扇入约束。可使用任意输入端口的与门、或门以及异或门实现。

* **5.13** 请确定实现一个8位超前进位加法器所需的门数，假定门的最大扇入为4。

5.14 图5-18所示为层次化的超前进位加法器的结构，请以两个2位加法器作为基本块，实现一个此种结构的4位加法器。

5.15 请指出图5-33所示乘法器中的关键延迟路径，并统计此关键路径上有多少个门的延迟？

5.16 (a) 写出描述图5-36b所示电路块的Verilog模块。用CAD工具综合该代码，并且验证它的正确性。

(b) 写出描述图5-36c所示电路块的Verilog模块。用CAD工具综合你的代码，并且验证它的正

确性。

(c) 写出描述图5-36a所示4 × 4乘法器的Verilog模块，把本题(a)和(b)所完成的设计作为子电路，写出层次化的描述。用CAD工具将此代码综合为电路，并验证其功能的正确性。

* **5.17** 考虑图P5-2所示的Verilog代码。假设输入信号和输出信号之间的关系已经给定，此代码描述的电路功能是什么？从功能描述的角度出发，评价此描述风格是否好。

```
module problem5_17 (IN, OUT);
   input [3:0] IN;
   output reg [3:0] OUT;

   always @(IN)
      if (IN == 4'b0101)  OUT = 4'b0001;
      else if (IN == 4'b0110)  OUT = 4'b0010;
      else if (IN == 4'b0111)  OUT = 4'b0011;
      else if (IN == 4'b1001)  OUT = 4'b0010;
      else if (IN == 4'b1010)  OUT = 4'b0100;
      else if (IN == 4'b1011)  OUT = 4'b0110;
      else if (IN == 4'b1101)  OUT = 4'b0011;
      else if (IN == 4'b1110)  OUT = 4'b0110;
      else if (IN == 4'b1111)  OUT = 4'b1001;
      else  OUT = 4'b0000;

endmodule
```

图P5-2 习题5.17的代码

5.18 设计一个产生BCD码字的9的补码的电路。（提示: 数字d的补码是$9 - d$。）

5.19 推导一个实现BCD码相减的方案，画出此减法器电路的方块图。（提示：若BCD操作数以10为基数的补码形式表示，则有利于减法器的实现。此时符号位为0表示正数；符号位为9表示负数。）

5.20 写出习题5.19推导出的电路的完整Verilog代码。

* **5.21** 假定我们想知道3位无符号数中有多少个取值为1的位，请设计一个能完成此任务的电路，要求电路最简单。

5.22 对6位无符号数，完成题5.21所要求的任务。

5.23 对8位无符号数，完成题5.21所要求的任务。

5.24 对3位十进制数，画出一个类似于图5-12的图形解释。若最左边1位为0则表示正数；若为9，则代表负数。用几个例子（包括加法和减法）验证答案的正确性。

5.25 在三进制数系统中，存在三个数字：0，1，2。图P5-3定义了一个三进制数的半加器，请设计一个实现此半加器的电路。要求用二进制编码表示三进制数，例如每个三进制数用2位表示。令$A = a_1a_0$，$B = b_1b_0$，$Sum = s_1s_0$，进位信号$Carry$是二进制信号。编码方案为：$00 = (0)_3$, $01 = (1)_3$, $10 = (2)_3$。要求电路的成本最低。

A B	*Carry*	*Sum*
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

图P5-3 三进制半加器

5.26 使用题5.25中所描述的方法，设计一个三进制全加器电路。

5.27 考虑下列减法运算：26 − 27=99和18 − 34=84。用5.3.4节中提到的概念，解释为什么上述答案（99和84）可以被认为是符号数减法的正确结果。

参考文献

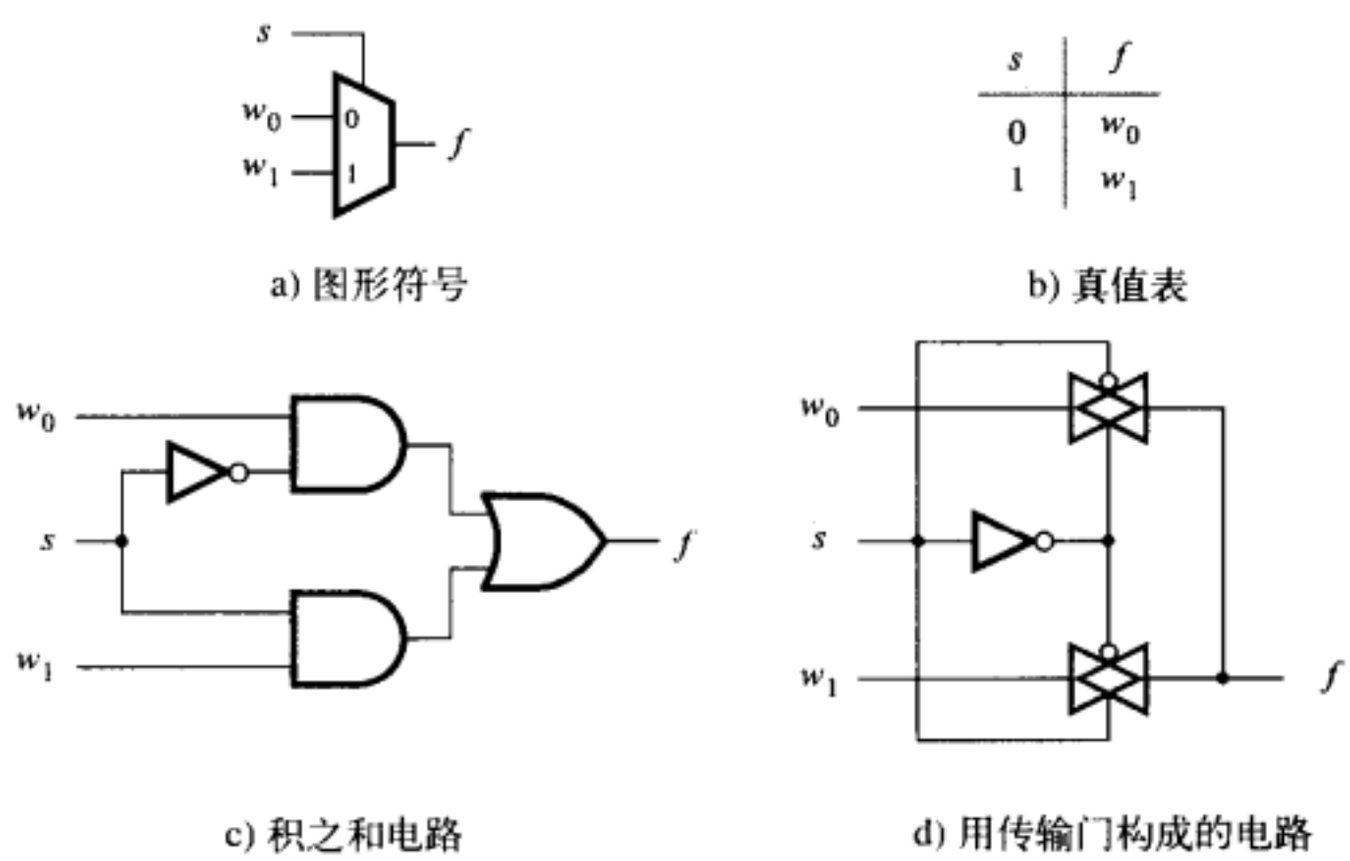
1. V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization*, 5th ed. (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed. (Morgan Kaufmann: San Francisco, CA, 2004).
3. Institute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic," *Computer* 14, no. 3 (March 1981), pp. 51–62.
4. W. W. Peterson and E. J. Weldon Jr., *Error-Correcting Codes*, 2nd ed. (MIT Press: Boston, MA, 1972).

第6章　组合电路构件块

在前面几章中，我们已经介绍了一些设计逻辑电路的基本技术。在实际应用中，有几种类型的逻辑电路块经常被用作模块来实现大规模设计。本章将介绍这样一些电路模块，并将给出应用举例。此外本章还包括了很重要的一节，其中介绍了Verilog语言的一些重要特性。

6.1　多路选择器

在第2章和第3章中，我们已经简单地介绍了多路选择器。它是由几路数据输入、一位或多位的选择控制和一路数据输出所组成的。多路选择器将从多路输入中选取其中一路将其传送到输出。由选择控制信号值决定输出的是第几路输入信号。图6-1中展示的是一个2选1多路选择器。图6-1a给出了常用的2选1多路选择器的图形符号，选择控制s的值决定了输出f的值等于w_0还是w_1。多路选择器的功能也可用真值表来表示，如图6-1b所示。图6-1c给出了2选1多路选择器用与或（积之和）形式表示的逻辑电路，图6-1d展示了如何用传输门构成2选1多路选择器。



a) 图形符号

s	f
0	w_0
1	w_1

b) 真值表

c) 积之和电路

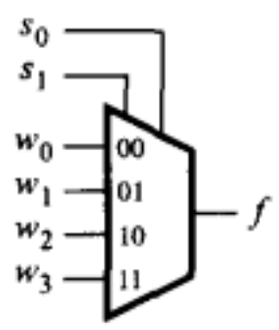
d) 用传输门构成的电路

图6-1　2选1多路选择器

图6-2a描绘了一个有四路数据输入（$w_0 \cdots w_3$）及两个选择控制信号（s_1和s_0），规模较大的多路选择器。如图6-2b中真值表所描述的那样，s_1 s_0组成了一个两位的二进制数，由s_1 s_0的值选定多路器的输出值等于哪路输入值。图6-2c是4选1多路选择器的逻辑电路，用与门和或门表示（积之和）。它实现的逻辑功能如下式所示：

$$f = \bar{s}_1\bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1\bar{s}_0 w_2 + s_1 s_0 w_3$$

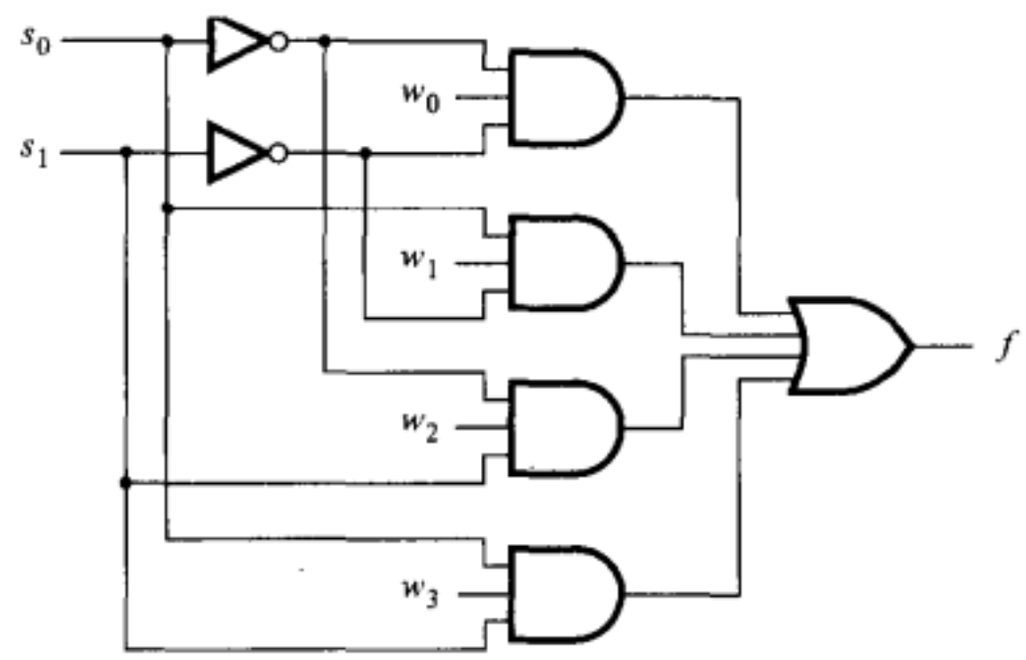
由同样的途径可以构造出更大的多路选择器。一般情况下，多路选择器中数据输入的路数n是2的整数幂。拥有n路输入的多路选择器（从w_0到w_{n-1}）需要$\lceil \log_2 n \rceil$个选择控制端。更大规模的多路选择器可以由较简单的多路选择器构成。例如，4选1多路选择器可以用三个2选1多路选择器实现，见图6-3所示。如果4选1多路选择器是由传输门实现的，那么图6-3中所使用的结构是普遍适用的。图6-4显示了如何用五个4选1多路选择器组成一个16选1多路选择器。



a) 图形符号

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

b) 真值表



c) 电路

图6-2 4选1多路选择器

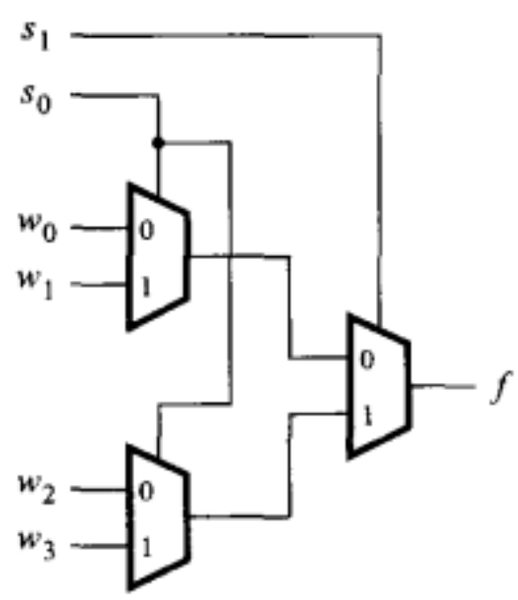


图6-3 由2选1多路选择器构建的4选1多路选择器

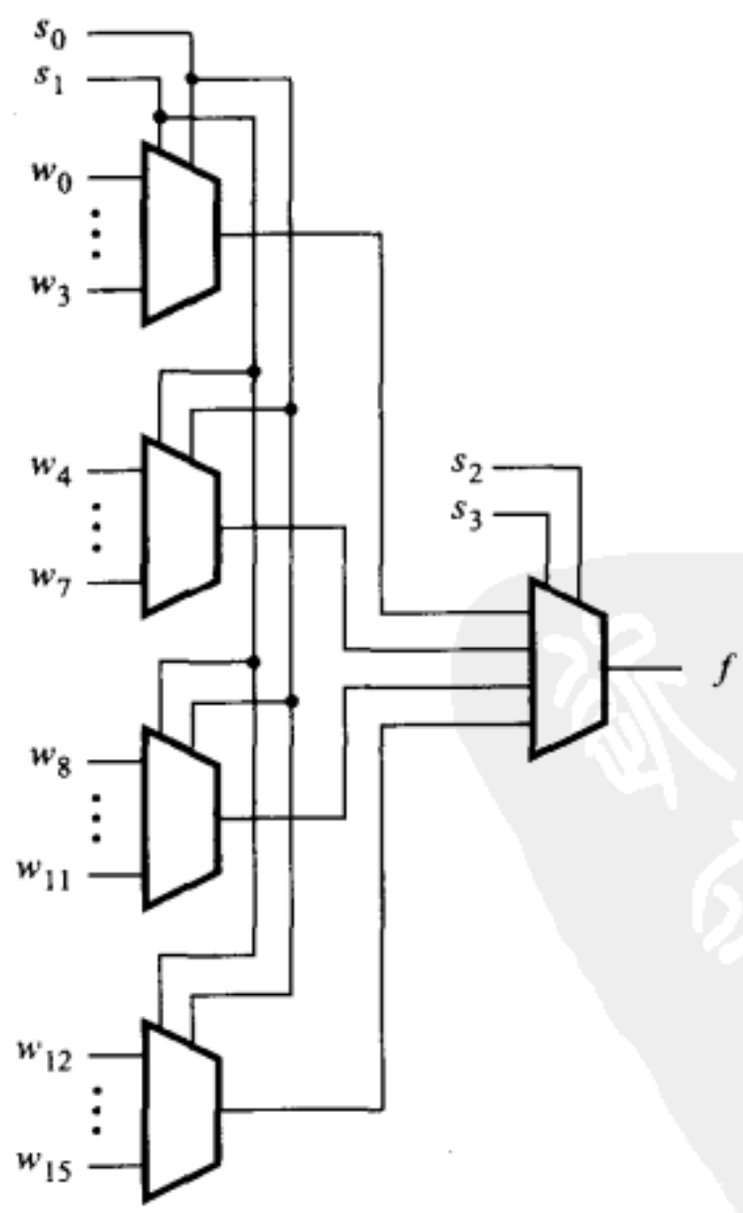
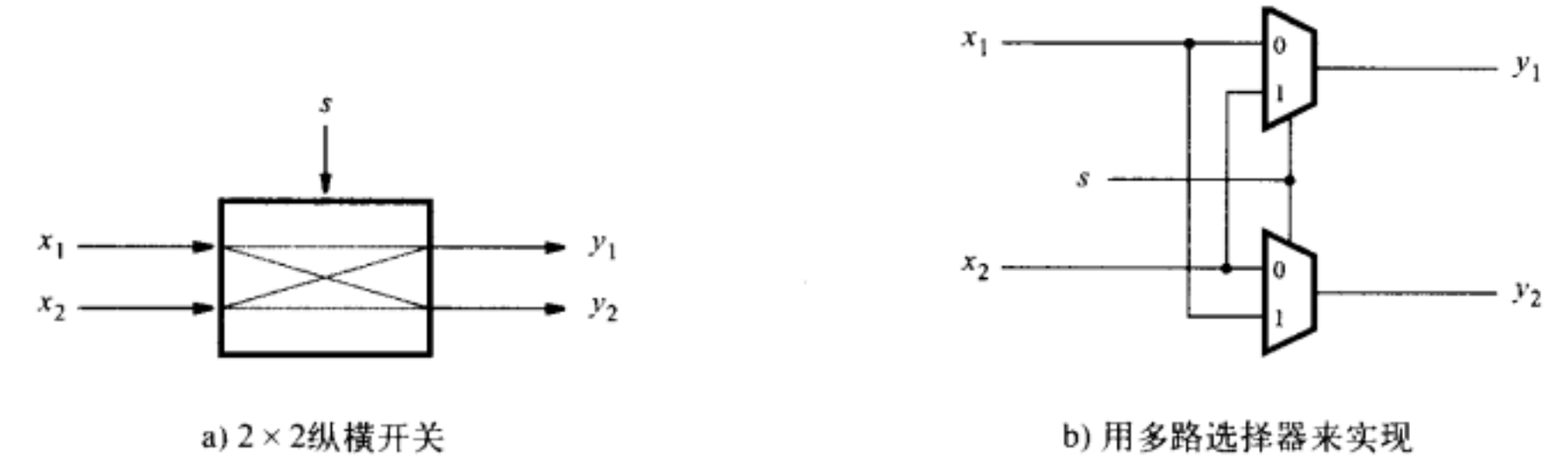


图6-4 16选1多路选择器

例6.1 图6-5所示的电路有两个输入信号（x_1和x_2）、两个输出信号（y_1和y_2）和一个选择控制输入信号（s）。该电路的功能是在s的控制下让x_1连接y_1，x_2连接y_2，或是x_1连接y_2，x_2连接y_1（见图中连线所示）。这种有着n个输入和k个输出的电路，它的功能就是把任意输入连接到任意输出，通常称之为$n \times k$纵横开关（crossbar switch）。我们可以用不同个数的输入和输出构造不同规模的纵横开关。有两个输入和两个输出的纵横开关称为2×2纵横开关。

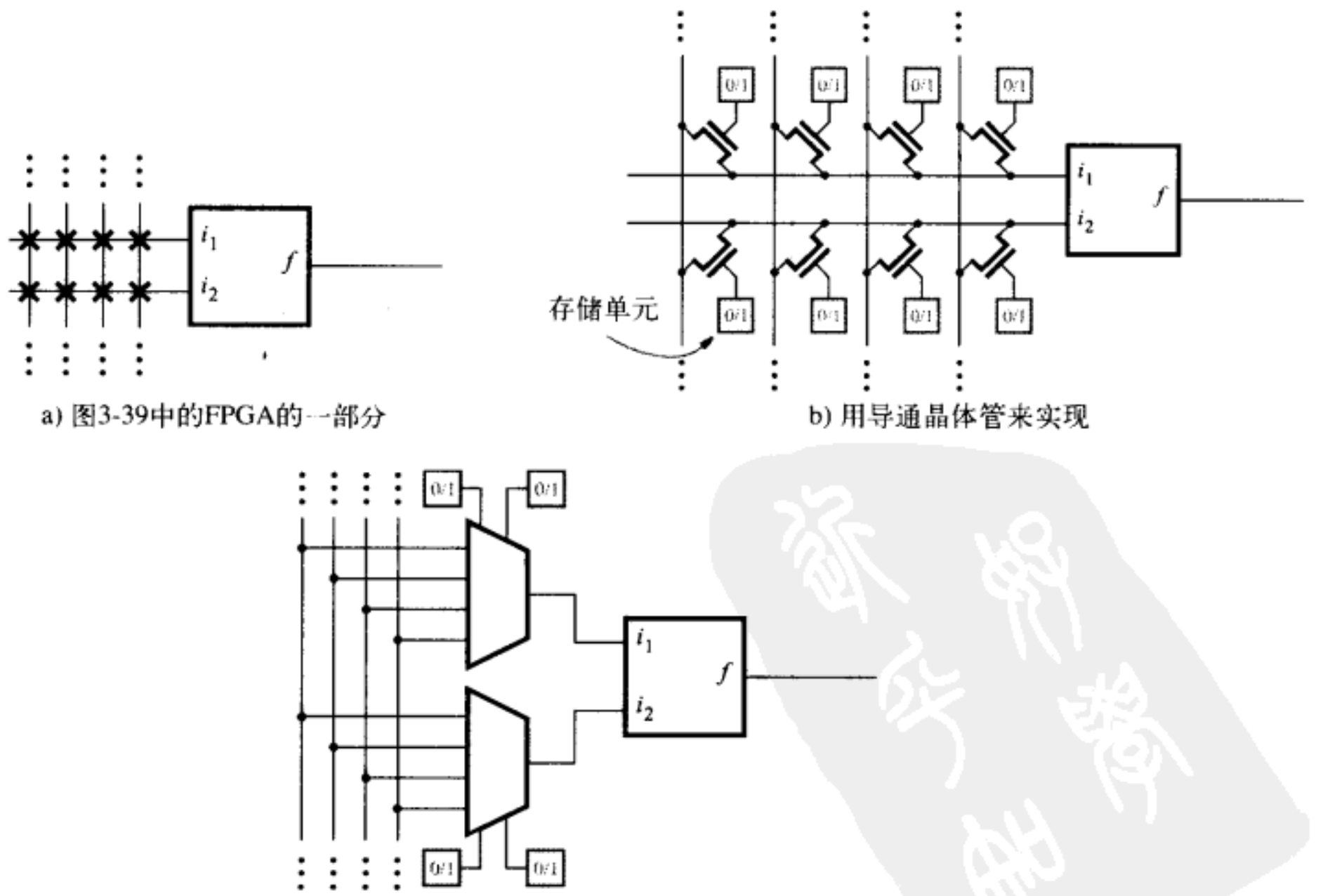
图6-5b展示了如何用2选1多路选择器实现2×2纵横开关。多路选择器的选择控制端由输入信号s控制，当$s = 0$时，x_1连接y_1，x_2连接y_2；而当$s = 1$时，x_1连接y_2，x_2连接y_1。在实际应用中，当用一组线连接另一组线时，如果其连接方式经常发生变化，那么就要用到纵横开关。



a) 2×2纵横开关　　b) 用多路选择器来实现

图6-5 多路选择器的实际应用

例6.2 在3.6.5节中，我们曾介绍过现场可编程门阵列（FPGA）芯片。图3-39描述了用来实现某一特定电路的简单的FPGA，该FPGA中的逻辑块有两个输入信号，布线通道有四条路径。能把逻辑块的输入或输出互连起来的可编程开关，在图中用X形表示。图6-6a截自于图3-39。为了看得更清楚，该图仅绘出一个简单的逻辑块，以及与输入端有关的开关和连线。



a) 图3-39中的FPGA的一部分　　b) 用导通晶体管来实现

c) 用多路选择器来实现

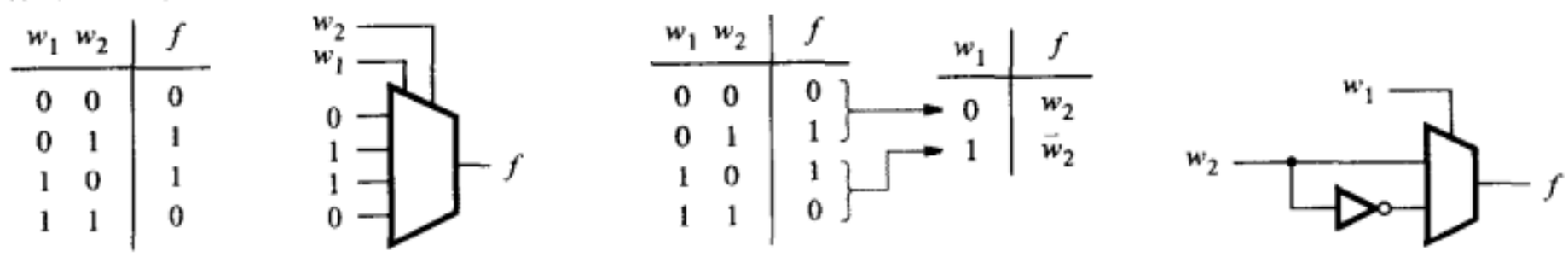
图6-6 实现FPGA中的可编程开关

实现可编程开关的方法之一，见图6-6b所示。图6-6a中的每个“X”在图b中用一个由存储单元控制的NMOS晶体管来实现，这种类型的可编程开关在图3-68中也出现过。在3.6.5节中，我们曾简单地介绍过存储单元；在10.1节中我们将更详细地介绍存储单元。在存储单元中，每个单元存储了一个逻辑值（1或0），并把该值赋予单元的输出。每个存储单元由若干个晶体管组成，因此在所示图中，8个存储单元占用了不少的芯片面积。

利用多路选择器能有效地减少存储单元的数目，如图6-6c所示。每一个逻辑块的输入由一个4选1的多路选择器提供，其中多路选择器的选择信号由存储单元来控制。这种方案只需要用4个存储单元就够了，而不必用8个。因此市售FPGA广泛采用基于多路选择器的方案。■

6.1.1 用多路选择器的逻辑函数的综合

如上所述，多路选择器（简称多路器）在很多实际应用中都是很有用的，同时它也广泛应用于逻辑综合。分析如图6-7a所示的例子。从真值表可以得出该逻辑函数为 $f = w_1 \oplus w_2$，这个函数可以用一个4选1的多路器来实现，把真值表中每一行的 f 值作为常数输入多路器，而多路器的选择信号由 w_1 和 w_2 驱动。因此，对应于每个w_1 w_2值，输出 f 都与真值表中对应行的函数值相等。

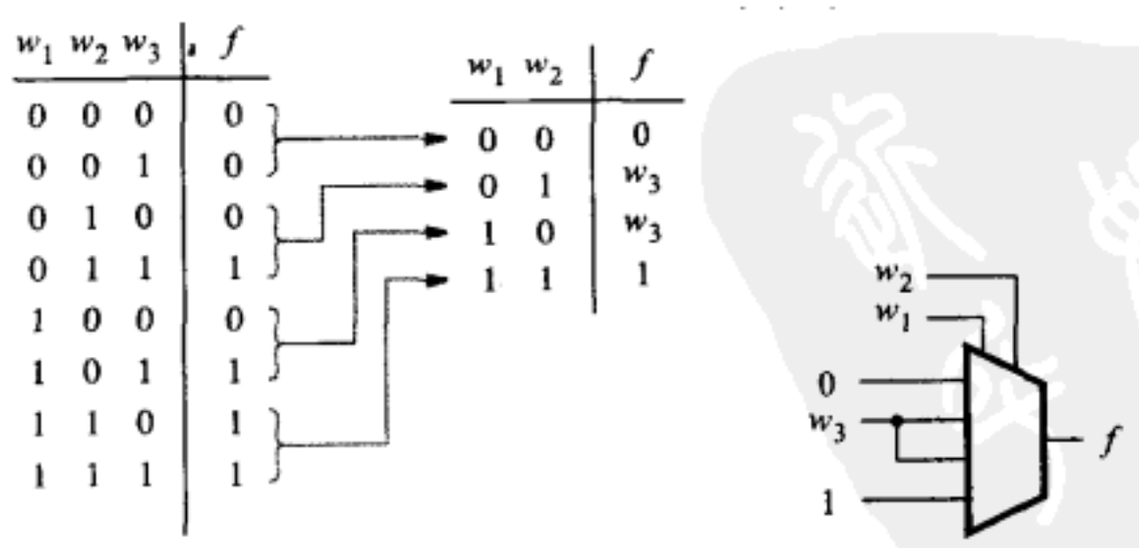


a) 用4选1多路器来实现函数　　b) 化简后的真值表　　c) 电路

图6-7 利用多路器来实现逻辑综合

以上的实现方式直接易懂，但并非很有效率。更好的实现方法是先把真值表化简，如图6-7b所示，化简后的真值表只用一个2选1的多路器就可以实现该逻辑函数。在该例中，其中一个输入信号w_1被用来作为2选1多路器的选择信号，此时真值表中，对应于每个w_1值，f 将被重新赋值。当$w_1 = 0$时，f 值与输入值w_2 相等，而当$w_1 = 1$时，f 值与 $\overline{w}_2$的值相等。化简后真值表的实现电路如图6-7c所示。这种方法可以用来综合实现任何逻辑函数的电路。

例6.3 图6-8a给出了三输入表决器的真值表。所谓表决器就是在输入的信号中，如果1的数目多，结果就是1，如果0的数目多，结果就是0。对图6-8a中的真值表进行化简，便可以知道只用一个4选1多路器就可以实现该逻辑函数。在这三个输入信号中，任何两个信号都可以用来作为多路器的选择信号。如果把w_1和w_2作为选择信号，就可以得到如图6-8b所示的电路图。



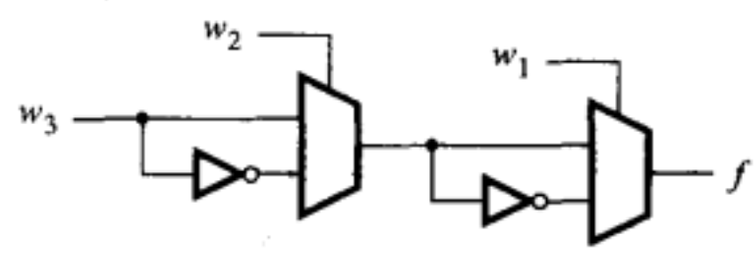
a) 化简真值表　　b) 电路

图6-8 利用4选1多路器来实现三输入表决器 ■

例6.4 图6-9a说明了如何用2选1多路器来实现函数$f = w_1 \oplus w_2 \oplus w_3$。当$w_1 = 0$时，$f$等于$w_2$与$w_3$的异或，而当$w_1 = 1$时，$f$是$w_2$与$w_3$的同或（异或非）。利用图6-7的结果可以得知，图6-9b中左边多路器的输出结果是$w_2 \oplus w_3$，而右边的多路器则用w_1的值来选择$w_2 \oplus w_3$或者$w_2 \oplus w_3$的非。我们把所求函数式写成$f = (w_2 \oplus w_3) \oplus w_1$的形式，则能直接得出此电路。

w_1	w_2	w_3	f	
0	0	0	0	$w_2 \oplus w_3$
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	$\overline{w_2 \oplus w_3}$
1	0	1	0	
1	1	0	0	
1	1	1	1	

a) 真值表



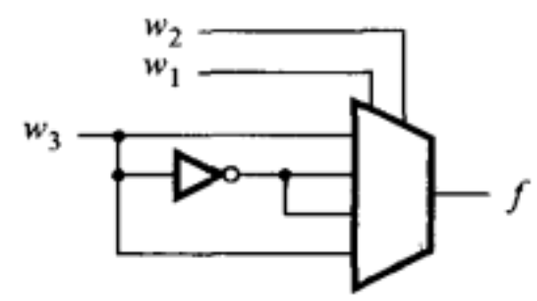
b) 电路

图6-9 利用2选1多路器实现3位输入的异或逻辑

图6-10 给出了用一个4选1的多路器实现3位输入信号的异或逻辑的实现方式。选择w_1和w_2为该4选1多路器的选择信号，可以得到图6-10b所示电路。

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

a) 真值表



b) 电路

图6-10 用4选1多路器来实现3位输入的异或逻辑 ■

6.1.2 用香农展开的多路选择器综合

图6-8至图6-10阐明了如何改写真值表以便用多路器来实现逻辑函数的方法。在所示的例子中，多路器的每一个输入都是常数0或1，或是一些变量以及它们的非。除了用一些简单的输入外，还可以把其他复杂的电路连至多路器作为输入，即允许把逻辑函数综合成多路器和其他逻辑门的组合。假设我们想要用一个2选1多路器，利用上述方法来实现如图6-8所示的3位输入表决器逻辑，则图6-11给出了一个实现该功能的直观方法。真值表化简为如图6-11a右图所示，如果$w_1 = 0$，则$f = w_2 w_3$，如果$w_1 = 1$，则$f = w_2 + w_3$。用w_1作为2选1多路器的选择信号，则可得出图6-11b所示的电路图。

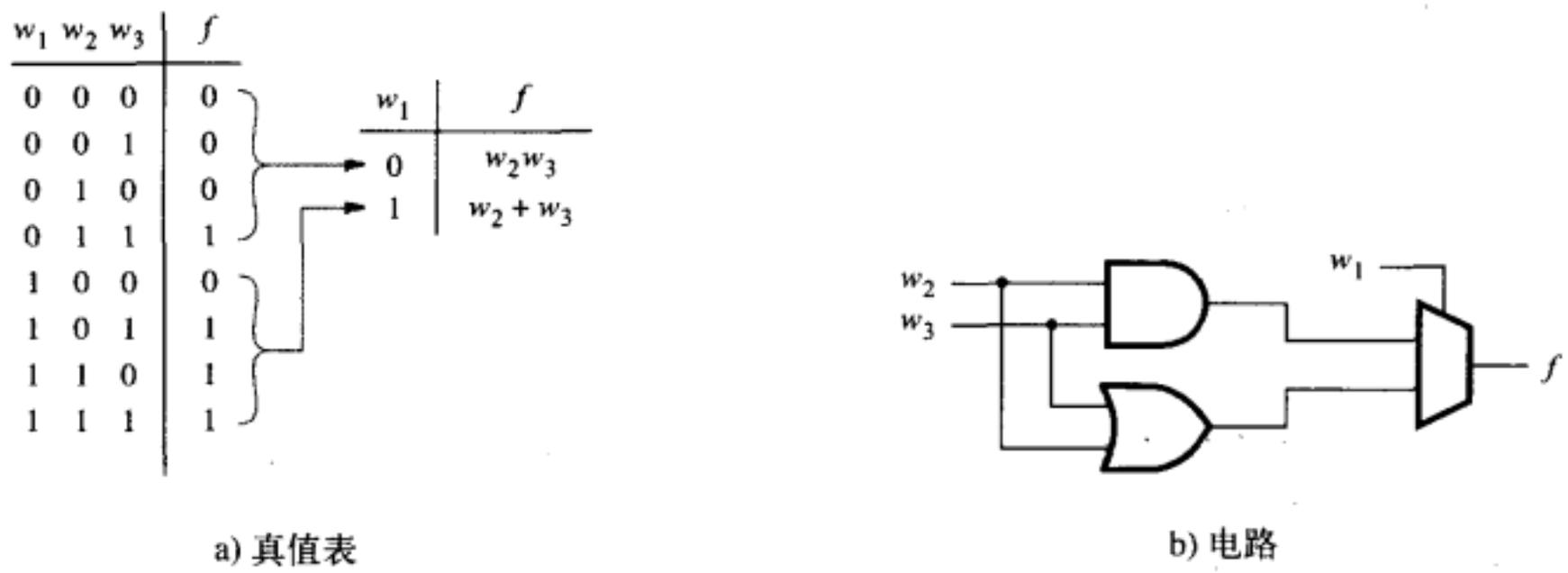
电路实现的过程可用如下的代数运算解释。图6-11a中的逻辑可以用如下的积之和（即与或形式的）表达式表示：

$$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$$

它可以表示为

$$f = \overline{w}_1(w_2w_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2w_3)$$
$$= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

与图6-11b中的电路图相对应。



a) 真值表　　b) 电路

图6-11　用一个2选1多路器来实现3位输入表决器

用多路器来实现逻辑函数，要求给出该逻辑函数的表达式，且该表达式必须分解成由选择输入变量（1/0）确定的两部分。香农（Claude Shannon）提出的展开定理能给我们提供帮助[1]。

香农展开定理

任何布尔函数$f(w_1, \cdots, w_n)$都可以表示成如下形式：

$$f(w_1, w_2, \cdots, w_n) = \overline{w}_1 \cdot f(0, w_2, \cdots, w_n) + w_1 \cdot f(1, w_2, \cdots, w_n)$$

无论n为何值，这一展开都是成立的。这一定理的证明我们将留给读者作为练习（见练习题6.9）。

为了说明这一用法，我们以三位表决器为例，它的布尔函数为

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

把该函数以w_1展开，得

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

这就是我们在上面所得到的式子。

将同样的方法用于三位输入的异或函数，我们得到：

$$f = w_1 \oplus w_2 \oplus w_3$$
$$= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3})$$

由此得出的电路图如图6-9b所示。

在香农表达式中$f(0, w_2, \cdots, w_n)$项称为对应于$\overline{w}_1$的f的余因子式，简写为$f_{\overline{w}_1}$。同样地，$f(1, w_2, \cdots, w_n)$项称为对应于w_1的f的余因子式，简写为f_{w_1}。所以香农展开定理可以简写成

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

一般来说，若函数表达式对应于变量w_i展开，则f_{w_i}可表示为：

$$f(w_1, \cdots, w_{i-1}, 1, w_{i+1}, \cdots, w_n)，且 f(w_1, \cdots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

以不同w_i为基础展开的逻辑表达式会有不同，见例6.5所示。

例6.5　对函数$f = \overline{w}_1w_3 + w_2\overline{w}_3$，以$w_1$来做香农分解，可得

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$
$$= \overline{w}_1(w_3 + w_2) + w_1(w_2\overline{w}_3)$$

若以w_2来做香农分解，则得

$$f = \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2}$$
$$= \overline{w}_2(\overline{w}_1 w_3) + w_2(\overline{w}_1 + \overline{w}_3)$$

而若以w_3来做香农分解，则得

$$f = \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3}$$
$$= \overline{w}_3(w_2) + w_3(\overline{w}_1)$$

用w_1和w_2来分解，所需的成本是一样的，但若以w_3做香农分解，则所需成本要低一些。在实际应用中，计算机辅助设计工具（CAD）做此类分解时总会尝试不同的分解方法，然后从中选择最佳即成本最低的方案。

香农展开也能以多个变量进行分解，以$w_1 w_2$为例，所得结果如下：

$$f(w_1,\cdots,w_n) = \overline{w}_1\overline{w}_2 \cdot f(0,0,w_3,\cdots,w_n) + \overline{w}_1 w_2 \cdot f(0,1,w_3,\cdots,w_n)$$
$$+ w_1\overline{w}_2 \cdot f(1,0,w_3,\cdots,w_n) + w_1 w_2 \cdot f(1,1,w_3,\cdots,w_n)$$

上述展开式可用4选1多路器实现。若香农展开定理以所有n个变量展开，则该表达式就是在2.6.1节中所定义的规范积之和（与或）形式。 ■

例6.6 假设我们想要用2选1的多路器和其他必须的门电路来实现以下函数：

$$f = \overline{w}_1\overline{w}_3 + w_1 w_2 + w_1 w_3$$

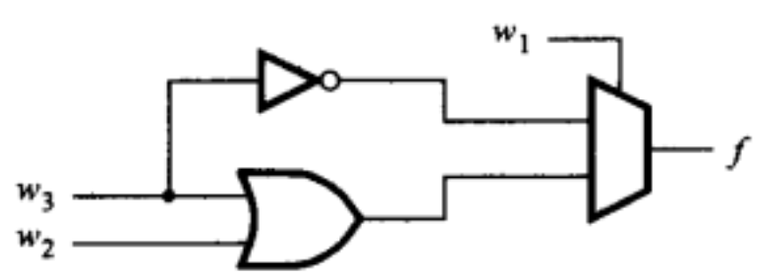
若以w_1为基础来进行香农展开，则函数可以表示为：

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$
$$= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3)$$

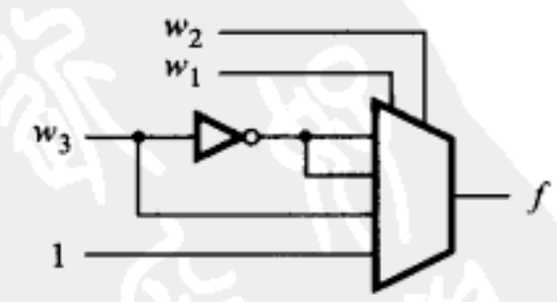
与以上逻辑表达式对应的电路图如图6-12a所示。若现在我们想用一个4选1多路器而不用2选1多路器来实现电路，以w_2做进一步的分解，则得到如下表达式：

$$f = \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2}$$
$$= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1)$$

实现电路如图6-12b所示。



a) 用2选1多路器



b) 用4选1多路器

图6-12 例6.6中的综合电路 ■

例6.7 分析三位输入的表决器函数：

$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

希望只用一个2选1多路器便能实现该逻辑函数。以w_1对上述逻辑表达式进行香农展开，得到如下表达式：

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

令$g = w_2w_3$且$h = w_2 + w_3$，以w_2展开g和h，得到

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

对应上述表达式的电路如图6-13所示。它与图6-8所示电路（由真值表推导出的4选1多路器）是等价的。

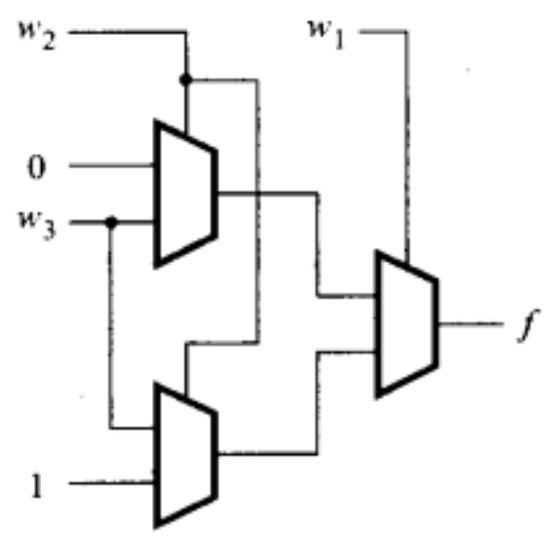


图6-13 例6.7中的综合电路 ■

例6.8 在3.6.5节中，我们曾提到大多数FPGA都使用查找表实现组合逻辑。假设某FPGA使用3输入查找表（3-LUT）来实现组合逻辑。因为查找表能存储真值表的输出，所以三输入查找表可以实现任何3变量的逻辑函数。根据香农展开定理，任何一个4输入变量的逻辑函数都可以用最多3个三输入查找表来实现。分析下面的函数：

$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

以w_1为基础对该函数做香农展开，得到：

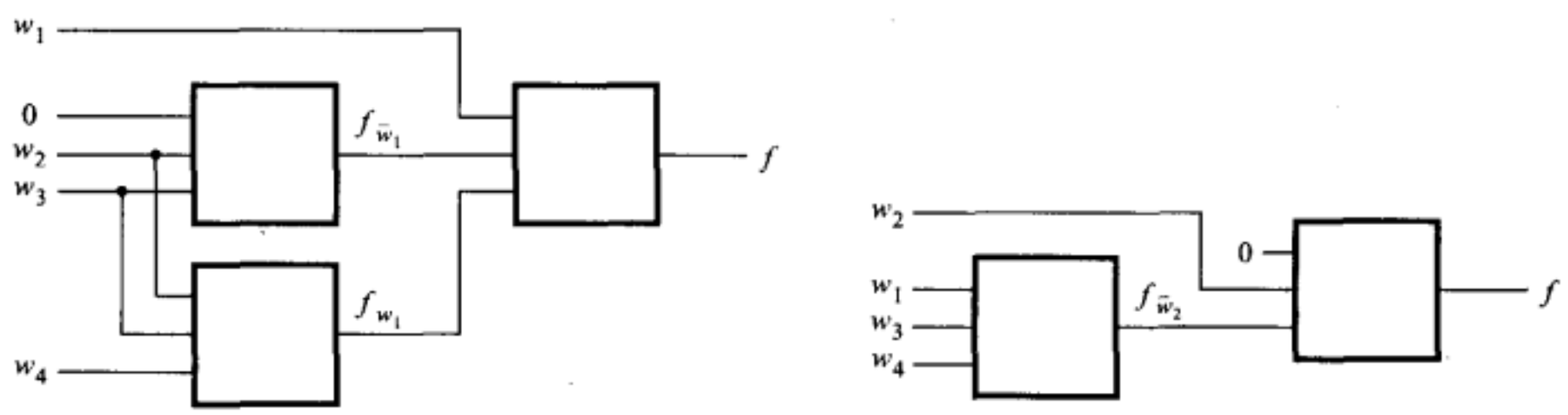
$$\begin{aligned} f &= \overline{w}_1f_{\overline{w}_1} + w_1f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

用3个三输入查找表实现该表达式的电路，如图6-14a所示。如果改用以w_2对该函数式进行香农展开，则可得到：

$$\begin{aligned} f &= \overline{w}_2f_{\overline{w}_2} + w_2f_{w_2} \\ &= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3w_4) \end{aligned}$$

注意到上式中$\bar{f}_{\overline{w}_2} = f_{w_2}$，因此只需要2个三输入查找表便可实现该表达式，见图6-14b所示。在6-14图中，右边的三输入查找表实现两个变量的函数$\overline{w}_2f_{\overline{w}_2} + w_2\bar{f}_{\overline{w}_2}$。

任何逻辑函数都可以用多路器来实现，因此通用芯片往往用多路器作为基本逻辑资源。Actel公司[2]和QuickLogic公司[3]的FPGA芯片其内部的逻辑块也都是由多路器组成的。德州仪器公司生产的门阵列芯片中也含有基于多路器的逻辑块[4]。



a) 用3个三输入查找表　　b) 用2个三输入查找表

图6-14　例6.8的综合电路 ■

6.2 译码器

译码器电路的作用是对已编码的信息进行译码。二进制译码器（见图6-15所示）是一个有n路输入和2^n路输出的逻辑电路。每组输入值对应于一个输出信号，即对应每次输入，只有一条输出信号值为真。译码器有一个使能信号（En），用来控制输出信号使能。若$En = 0$，则译码器没有任何输出，即所有输出均为0。若$En = 1$，则输入$w_{n-1}\cdots w_1\ w_0$的值决定了输出信号的值。在n位二进制码中，若码的变化只是对应不同的一位被置为1，其余各位都为0，则该二进制编码被称为*独热码编码*(one-hot encoded)，意思是那个被置为1的码位看起来是“热”的，而二进制译码器的输出信号就是独热码编码。

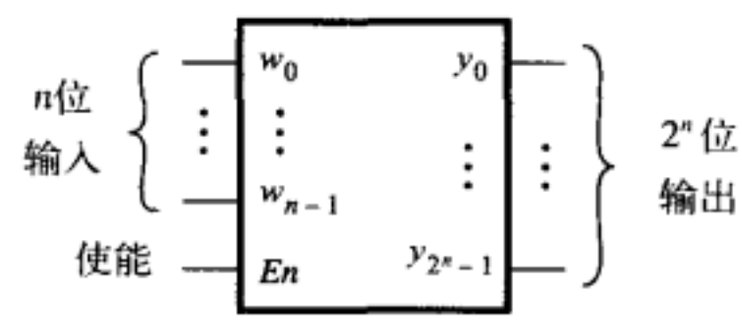
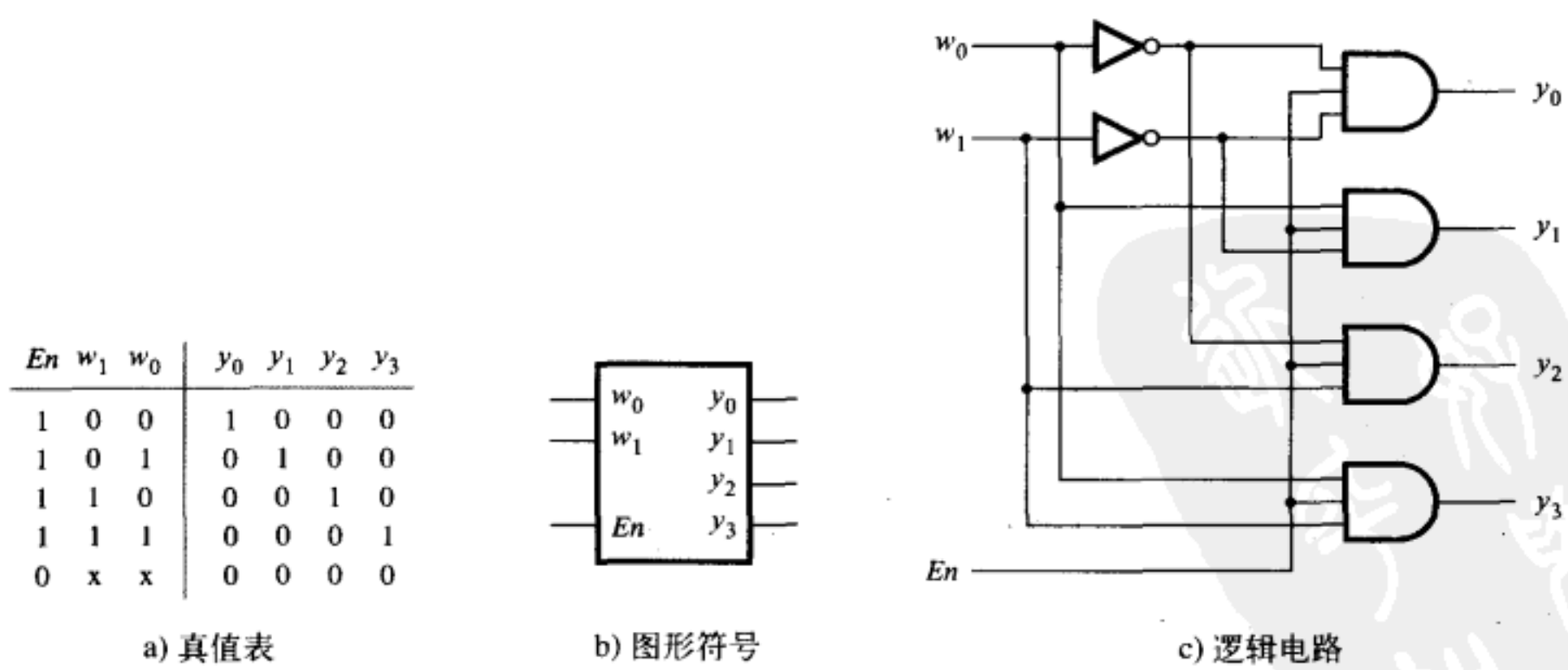


图6-15　n位输入，2^n位输出的译码器

2到4译码器（见图6-16所示），它的输入信号为w_1和w_0。对每个二进制输入信号$w_1\ w_0$，译码器的输出是在y_0，y_1，y_2，y_3四位中选择一位，使其变为有效。译码器的输出能被设计成高电平有效或低电平有效，在图6-16中采用的是高电平有效。当输入信号$w_1\ w_0$分别为00，01，10，11时，对应输出y_0，y_1，y_2，y_3分别为1。该译码器的图形符号见图6-16b所示，而它的逻辑电路则如图6-16c所示。



En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

a) 真值表　　b) 图形符号　　c) 逻辑电路

图6-16　2到4译码器

更大的译码器可以由如图6-16c所示的与或结构组成，也可以由小译码器构成。图6-17说

明如何用两个2到4译码器来组成一个3到8译码器。输入信号w_2驱动这两个译码器的使能输入。若$w_2 = 0$，则图中上面那个译码器使能，若$w_2 = 1$，则下面的那个译码器使能。由此思路可以构成任何大小的译码器。图6-18说明了怎样用五个2到4译码器来组成一个4到16译码器。该译码器的结构呈现树状，因而被称为译码树。

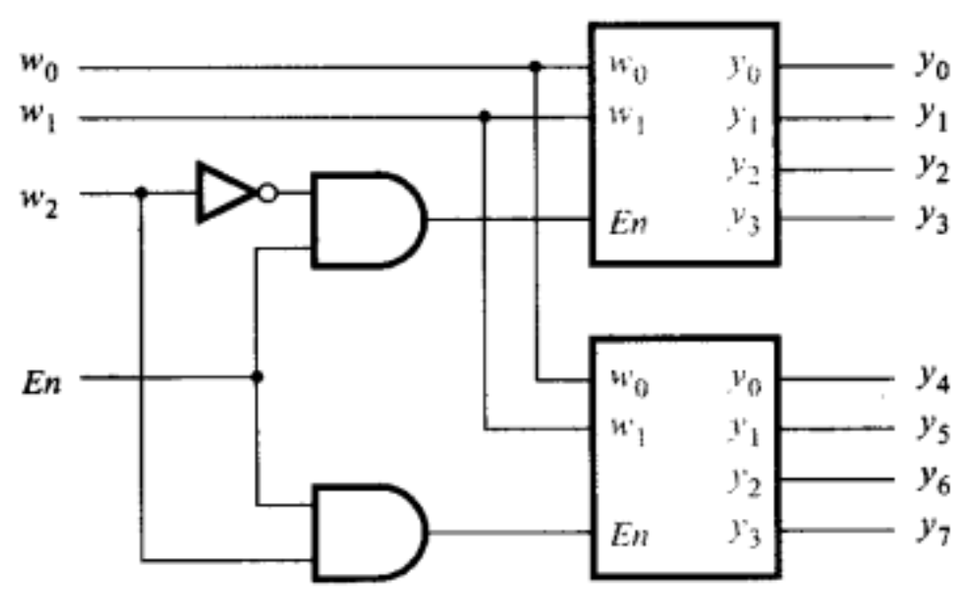


图6-17 用两个2到4译码器构成一个3到8译码器

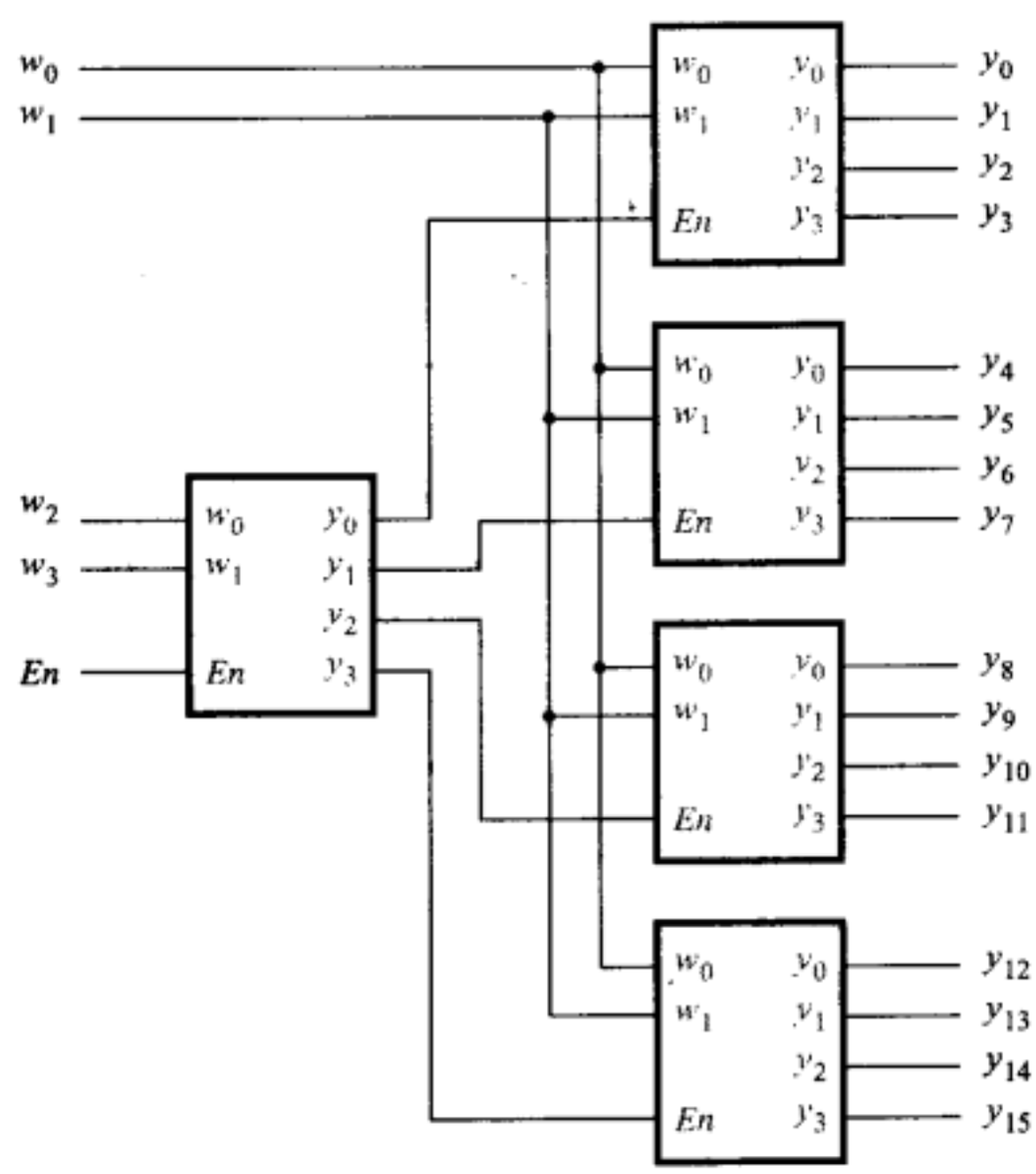


图6-18 用译码树来构建4到16译码器

例6.9 译码器广泛地应用于多种实际场合。我们曾在图6-2c中展示了如何用积之和（与门和或门）的形式来构成4选1多路器。该方法需要用与门来区分输入选择信号$s_1 s_0$的四个不同的组合值。而译码器的作用是对输入的信号值进行区分，所以可以被用来构成多路器，见图6-19所示。在这种情况下，译码器不再需要使能信号来控制，因而使能信号被置为1。译码器的四个输出可用作多路器的四个选择控制信号，从而从4个输入中选取其中一个将其输出。■

例6.10 图3-59中曾给出了如何用两个三态缓冲器来构成一个2选1多路器。同样，利用三态缓冲器，再加上译码器，我们也可以构建任何规模的多路器。图6-20是一个范例。根据选择线上传输来的信号，译码器使4个三态缓冲器中的一个导通、其他3个断开，从而将相应

数据输出至f。通过以上介绍，我们知道多路器可以由不同方式来实现。选择积之和方式、传输门方式还是三态缓冲器方式取决于芯片上可利用的资源。例如，大多数FPGA用查找表生成组合逻辑块，而内部没有三态缓冲器。因此在这种芯片上多路器必须用查找表以积之和的形式来实现（见例6.33）。■

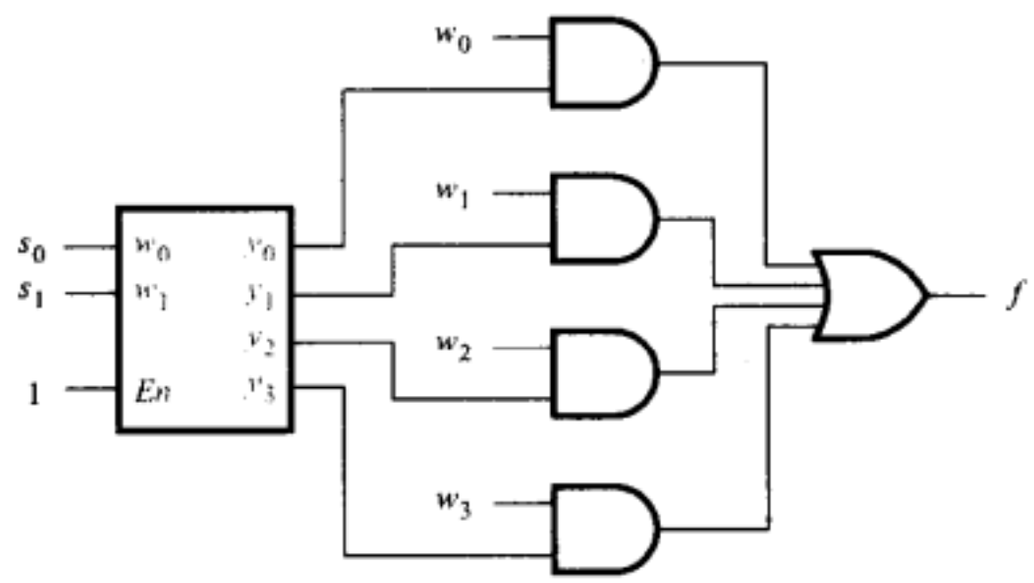


图6-19 用译码器来构建一个4选1的多路器

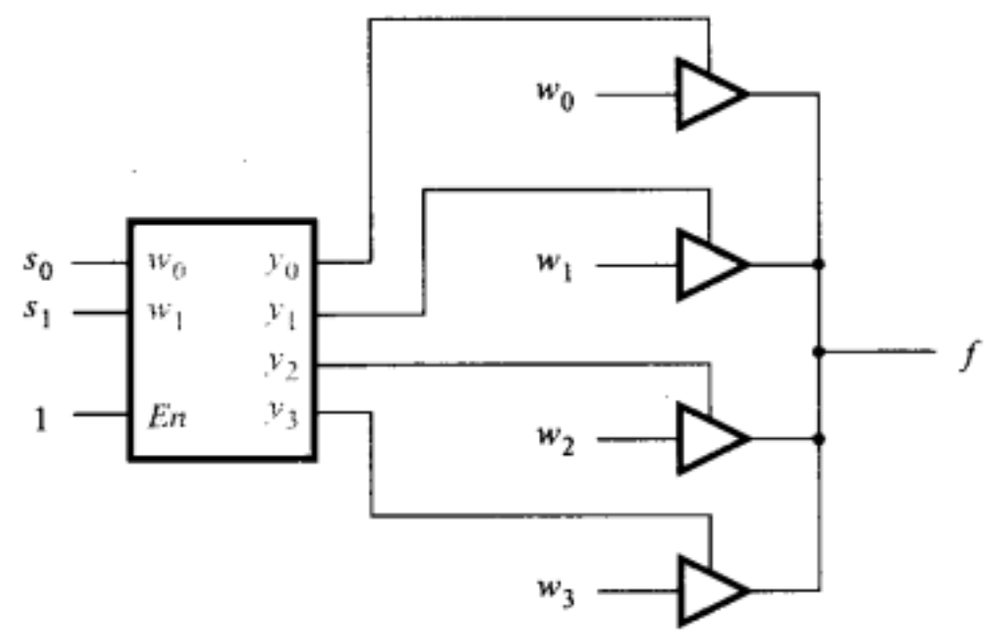


图6-20 用一个译码器和三态缓冲器来构建4选1多路器

多路分解器

我们曾在6.1节中提到，多路选择器有n路数据输入，一路数据输出，以及$\lceil \log_2 n \rceil$个选择控制信号。多路选择器的用途是在选择信号的控制下把输入的n路数据中的一路转接到输出端。而*多路分解器*的作用正好与此相反，即把单路输入数据转接到多路的某一路输出，所以称为多路分解器。多路分解器能用译码器电路来实现。例如，2到4译码器能起到1到4多路分解器的作用。如图6-16所示，我们把输入使能信号En看成是多路分解器的数据输入信号，而输出多路信号为y_0至y_3，而$w_1 w_0$的值则决定y_0至y_3中哪一路输出En的值。为了弄清楚该电路是怎样工作的，分析图6-16a中的真值表。当$En = 0$时，所有的输出都被置为0（包括由$w_1 w_0$的值所选中的输出信号），而当$En = 1$时，$w_1 w_0$的值所对应的那路输出被置为1（即跟随En）。

总的来说，一个n到2^n译码器电路能用作1到2^n多路分解器。然而，在实际应用中，译码器电路通常用作译码器，很少用作多路分解器。在很多应用场合，译码器的使能输入信号En实际上没有什么用处，因此可以省略。此时当输入数据值为$w_{n-1} \cdots w_0$时，译码器总能产生相应的输出信号，使y_0，…，y_{2^n-1}中的一个有效。在例6.11中使用的译码器没有En使能输入。

例6.11 译码器的最重要应用之一是用在存储器块中，存储器块是用于存储信息的。数字系统例如计算机，需要存储大量的电子信息，就需要存储器块。存储器块中有一种称为*只读存储器*（ROM），只读存储器是由存储单元组成的，每个存储单元存储着一个固定的逻辑值，0或者1。只读存储器的实例如图6-21所示。存储单元被排列成2^m行，每行有n个单元。因此每一行都能存储n位信息。只读存储器中每一行的位置是由它的*地址*决定的，在图6-21中，最上面存储器的行地址为0，而在最下面的行地址为$2^m - 1$。存储在这些行内的信息能通过$Sel_0 \cdots Sel_{2^m-1}$线选信号来访问，用一个m到2^m译码器来产生线选信号，如图6-21所示。译码器的输入信号用来选择特定的行地址，因此被称为*地址线*。存储在行内的信息由只读存储器的数据输出端$d_{n-1} \cdots d_0$输出，因此这些数据输出线被称为*数据线*。图6-21展示每条数据线都有一个对应的三态缓冲器，这个三态缓冲器是由只读存储器的输入信号驱动使能的，这一输入信号称为*读取信号*。为了从只读存储器中访问或读取数据，必须把目的地址赋予地址线，而且读取信号要被置为1。

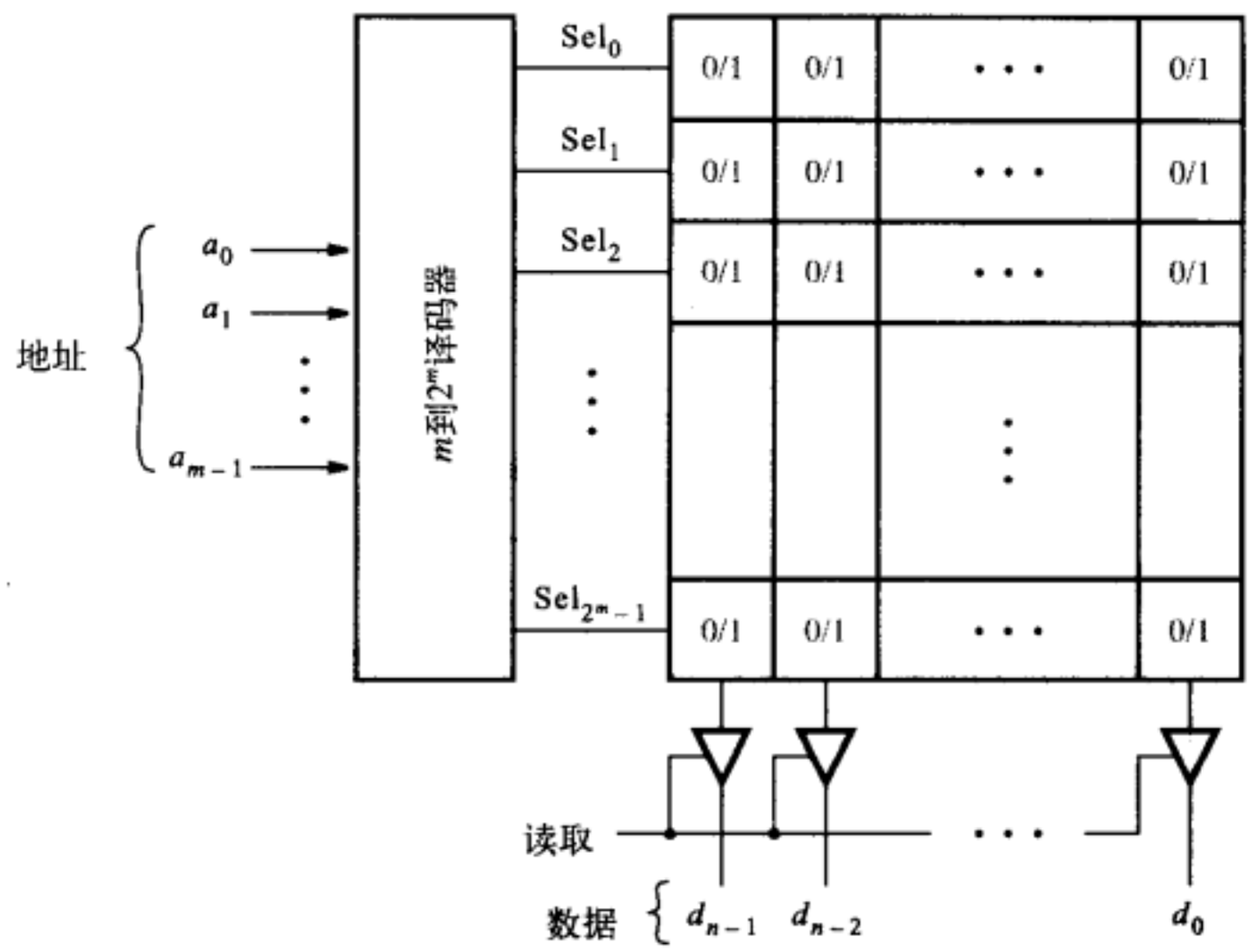


图6-21 $2^m \times n$ 位只读存储器块电路图

目前市面上存储器的类型有很多种。对于一般的只读存储器，存储的信息只能从存储单元中读出，但无法改写（见习题6.31）。还有一种只读存储器，存储的信息不仅可以从中读出，还可以修改写入信息，读出为常规操作，改写需要特殊的操作过程。这种存储电路块称为可编程只读存储器（PROM）。可编程只读存储器的存储单元通常由电可擦除可编程只读存储器（EEPROM）晶体管构成，在3.10节中我们曾介绍过用于PLD 中的EEPROM 晶体管。至于其他类型的存储块我们将在10.1节中讨论。 ■

6.3 编码器

编码器的作用与译码器相反，它是把给定的信息转换成更紧凑的形式。

6.3.1 二进制编码器

二进制编码器把来自于2^n条输入线的信息编码转换成n位二进制码，见图6-22所示。准确地说，每次输入的2^n位信号中只能有一位为1，其余均为0（即独热码），而编码器的输出信号为一个二进制数，表明对应的哪个输入位为1。图6-23a给出了4位输入2位输出编码器（简称4到2编码器）的真值表。观察真值表可知，当输入w_1或w_3为1时，输出y_0为1，当输入w_2或w_3为1时，输出y_1为1。因此这些输出信号能由图6-23b所示电路生成。注意，我们假设输入信号是独热码编码，那些有两位或两位以上被设置为1的输入组合并没有列在真值表中，这些输入形式不予考虑。

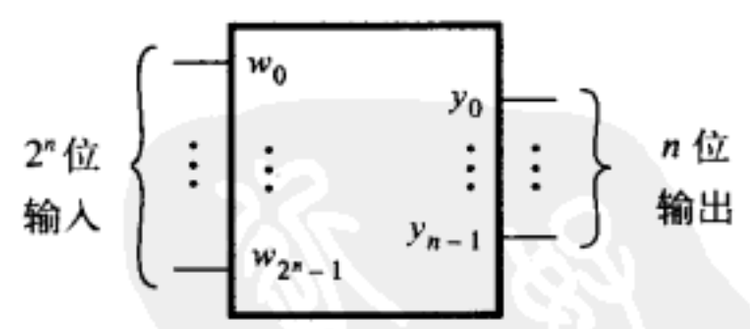
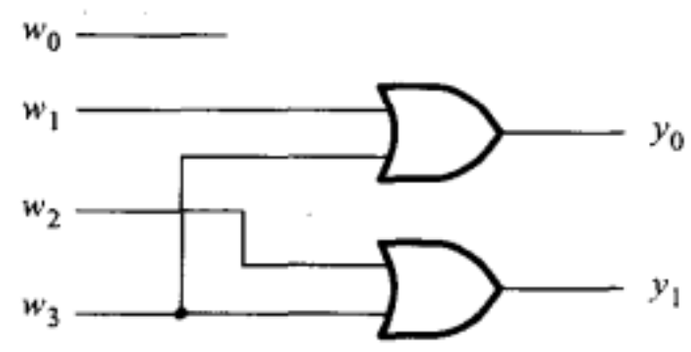


图6-22 一个2^n位输入、n位输出的二进制编码器

编码器可以用来减少给定独热码信息的比特数。当在数字系统里传输独热码信息时，常用编码器将信息编码以减少传输线。独热码信息如果需存储起来以便于以后使用，编码器也很有用，因为这样做可以减少需存储代码的位数。

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

a) 真值表



b) 电路

图6-23 4到2编码器

6.3.2 优先级编码器

另外有一类编码器也很有用，它根据输入信号的优先级进行编码，称为*优先级编码器*。在优先级编码器中，每路输入信号都有一个与之对应的优先级别，编码器的输出能指出哪一路是具有最高优先级输入为1（有效）的路（位）。当具有最高优先级的输入路（位）有效时，其他优先级较低路（位）上的输入变化全被忽略。图6-24所示的真值表列出了4到2优先级编码器的逻辑关系。该真值表说明，w_0 的优先级最低，w_3 的优先级最高。输出$y_1 y_0$表示一个二进制数，说明从w_0 到w_3 中哪一路（位）是被置为1的输入信号中优先级最高者。当然，输入信号的任何路（位）都没有被置为1也是有可能的，输出z就是用来区分这种情况的。当至少有一路（位）输入被置为1时，z被置为1，当所有的输入都等于0时，z被置为0。在这种情况下y_1 和y_0 就没有意义，因此真值表中的第一行中y_1 和y_0 的取值可以不予考虑。

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

图6-24 4到2优先级编码器的真值表

分析真值表中的最后一行可以很容易明白优先级编码器的行为逻辑。当w_3 为1时，输出$y_1 y_0 = 11$，因为w_3 的优先级最高，w_2 ,w_1 和w_0 的输入值对输出无影响。为了反映它们的值对输出无关紧要，w_2 ,w_1 和w_0 的值在真值表中用x来表示。真值表中的倒数第二行表明当$w_3 = 0$时，若 $w_2 = 1$，则输出$y_1 y_0 = 10$。同样，当w_3 和w_2 都为0时，若$w_1 = 1$，则输出$y_1 y_0 = 01$，而只有在仅$w_0 = 1$时，输出$y_1 y_0$才被置成00。

用第4章曾介绍过的方法，我们可以把图6-24所示的真值表综合成优先级编码器电路。仔细观察并分析真值表，可以找到一种更简便的综合方法：即设置一组中间变量：$i_0, \cdots, i_3$，仅当下标为k 的输入信号w_k 是所有取值为1的输入信号中优先级最高者时，具有同样下标的中间变量i_k 才取值为1。由此推导得到中间变量$i_0, \cdots, i_3$的逻辑表达式为

$$i_0 = \overline{w}_3\overline{w}_2\overline{w}_1 w_0$$
$$i_1 = \overline{w}_3\overline{w}_2 w_1$$
$$i_2 = \overline{w}_3 w_2$$
$$i_3 = w_3$$

优先级编码器的其余电路与图6-23所示的二进制编码器有着同样的结构，利用这些中间变量，可以得到如下输出信号的逻辑表达式：

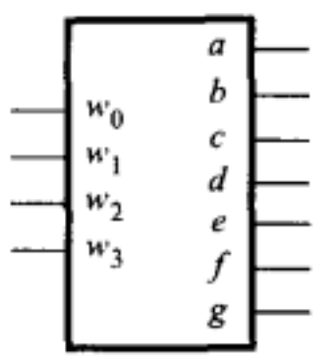
$$y_0 = i_1 + i_3$$
$$y_1 = i_2 + i_3$$

最后输出z的逻辑表达式为：

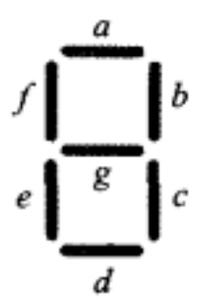
$$z = i_0 + i_1 + i_2 + i_3$$

6.4 码型转换器

编码器和译码器电路的用途是把一种形式的编码（输入）转换为另一种形式的编码（输出）。举例来说，3到8译码器是把输入的二进制数转换为独热码送到输出端。而8到3编码器的作用正好相反。除此之外，还存在着许多种类型的码型转换器。常见的例子是BCD码转换为七段数码管显示用的译码器，它是把二进制编码的十进制数（BCD）转换成驱动数码显示管的信息。如图6-25a所示，该电路把输入的BCD码转换成用于驱动数码管各段显示的7个信号。数码管的每一段是一个小发光二极管（LED），当有电信号驱动时，它便发光。在图中，这七段分别用字母a至g来标记。BCD码转换成7段码的译码器真值表见图6-25c所示。对输入$w_3,\cdots,w_0$的每一个值，7个输出信号驱动数码管中相应的LED，显示十进制数字。注意，完整的16行真值表的最后6行在图中并没有示出。这些情况是不需要考虑的，因为它们是非法的BCD码，所以在处理BCD码的有关电路中不会出现。在第4章中我们曾讨论过，实现真值表的电路能由综合工具自动生成，此处就不予介绍。最后，必须注意虽然传统上我们用*译码器*来称呼这类电路，但事实上，还是把它叫做*码型转换器*更贴切，而*译码器*这个术语更适用于产生独热码输出的电路。



a) 码型转换器



b) 7段数码显示

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

c) 真值表

图6-25 BCD码至7段显示码的码型转换器

6.5 算术比较电路

第5章曾介绍二进制数的加法、减法和乘法的算术电路。下面介绍另一种很有用的比较两个二进制数相对大小的算术电路，这种电路称为*比较器*。本节将考虑如何设计一个比较器，该比较器的输入为两个n位无符号二进制数A和B。比较器有三个输出分别为：$AeqB$，$AgtB$和$AltB$。当A等于B时，$AeqB$输出为1。当A大于B时，$AgtB$输出为1。当A小于B时，$AltB$输出为1。

把三个输出表示成A和B的函数，由此得出真值表，我们想要设计的比较器则可以由该真值表推出。然而，就算n的值不大，真值表也是庞大的。一种更好的方法是把A和B按位成对地比较分析。我们用一个小例子来说明这种方法，假设$n = 4$，$A = a_3a_2a_1a_0$，$B = b_3b_2b_1b_0$。定义一组中间变量i_3，i_2，i_1和i_0。当与i_k有相同下标的A和B的对应位相等时，i_k为1，也就是说，$i_k = \overline{a_k \oplus b_k}$。因此比较器的$AeqB$输出端可用下面的逻辑表达式表示：

$$AeqB = i_3 i_2 i_1 i_0$$

$AgtB$的表达式能由把A和B的对应位按从最高位到最低位顺序比较分析来获得。第一位a_k和b_k不相等的值决定了A比B大还是小。当$a_k = 0$且$b_k = 1$时，$A < B$。当$a_k = 1$且$b_k = 0$时，则$A > B$。由

此推出输出$AgtB$的逻辑表达式如下：

$$AgtB = a_3\overline{b}_3 + i_3a_2\overline{b}_2 + i_3i_2a_1\overline{b}_1 + i_3i_2i_1a_0\overline{b}_0$$

i_k 信号确保了只有按从左到右的顺序分析A和B的对应位，第一个不相同位的值才决定$AgtB$的值。$AltB$的值能由其他两个输出值得到，表达式为：

$$AltB = \overline{AeqB + AgtB}$$

实现4位比较器的逻辑电路如图6-26所示。用这种方法可以设计出n（任意多）位的比较器。

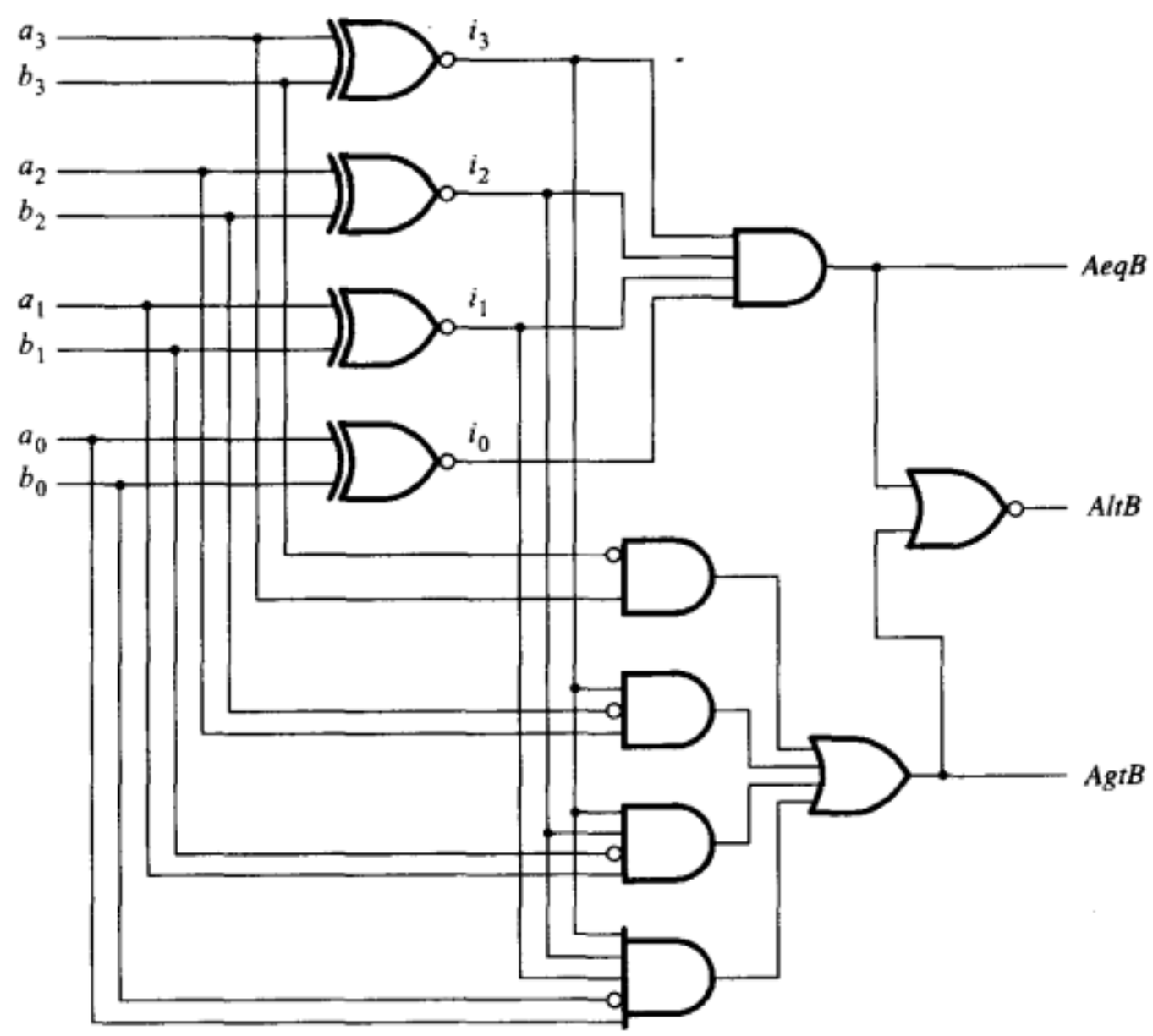


图6-26 4位比较器电路

像大多数逻辑电路一样，比较器电路也可以通过多种方式来设计。第5章的例5.10给出了设计比较器电路的另一种方式。

6.6 用Verilog表示组合电路

我们已经介绍了一些有用的构建块电路，现在考虑如何用Verilog语言来描述这些电路块。我们将用电路的行为（而不是用门或者逻辑表达式）来描述这些块。同时，我们将对以前使用过的行为级Verilog结构给出更严格的描述，并且将介绍一些新的Verilog结构块。

6.6.1 条件操作符

在逻辑电路中，经常需要在不同的条件状态下对几个可能的信号或值进行选择。典型的例子就是多路选择器，它是根据选择信号的值把对应的输入信号值赋给输出。为了实现这样简单的选择，Verilog语言提供了一种条件操作符（? :），它是根据条件表达式的真伪在两个值中选择一个。在语法中使用时，条件操作涉及三个操作数：

条件表达式 ? 真表达式 : 伪表达式

如果条件表达式为1，即为真，则选择真表达式的值，否则，选择伪表达式的值。例如，有这样一条声明语句：

A = (B <C)? (D+5) : (D+2) ;

它意味着，如果B小于C，则A的值为D+5，否则，A的值为D+2。在这个表达式中我们加入了圆括号来增加表达式的可读性；事实上，它们是不需要的。条件操作符不仅可以用在连续赋值语句中，也可以用在**always**块的过程语句中。

例6.12 2选1多路器可以在一个赋值语句中用一个条件操作符来定义，如图6-27所示。该模块叫做mux2to1,它有三个输入信号，分别为w_0, w_1和s，有一个输出信号f，输入信号s用来作为选择依据。当输入的选择信号s的值为1时，输出f 等于w_1，否则，输出f 等于w_0。图6-28示出了同样的多路器在**always**块中使用条件操作符怎样来定义。

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output f;

   assign f = s ? w1 : w0;

endmodule
```

图6-27 用条件操作符来实现2选1多路器

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
      f = s ? w1 : w0;

endmodule
```

图6-28 用条件操作符来实现2选1多路器的另一种描述

我们可以用同样的方法，即嵌套条件操作符来定义4选1多路器，如图6-29所示。该模块名为mux4to1，它有两个选择输入信号s_1和s_0（见图6-2），现在用一个两位的向量S表示。第一个条件表达式检测s_1的值，当$s_1 = 1$时，则检测s_0的值，当$s_0 = 1$ 时，f 被赋予w_3的值，当$s_0 = 0$时，f 被赋予w_2的值。这与图6-2b中的真值表的第三行和第四行是对应的。同样，如果$s_1 = 0$，则右边的条件操作符在$s_0 = 1$ 时选择$f = w_1$，而当$s_0 = 0$ 时，$f = w_0$。由此便可实现真值表中的前两行。

```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output f;

   assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

图6-29 用条件操作符来实现4选1 多路器

6.6.2 if-else 语句

在前面的章节中，我们曾使用过**if-else**语句，它的语法结构为：

if（条件表达式）语句1；
else 语句2；

这里条件表达式可以使用表A-1中的操作符。若表达式值为真，则执行语句1，或执行在关键字**begin**和**end**之间的语句块；如果表达式值为假，则执行语句2或语句块。

例6.13 图6-30展示了如何用**if-else**语句来描述2选1多路器。该语句表明当$s = 0$时，w_0的

值赋给f，若$s=1$，则w_1的值赋给f。

if-else语句可以用来实现更大规模的多路器。图6-31展示了4选1多路器，**if-else**语句把f置为输入信号$w_0, \cdots, w_3$中的哪一个取决于s的值。编译该段代码，可得到如图6-2c所示的电路。

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
      if (s == 0)
         f = w0;
      else
         f = w1;

endmodule
```

图6-30 用**if-else**语句给2选1多路器编码

```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output reg f;

   always @(*)
      if (S == 2'b00)
         f = w0;
      else if (S == 2'b01)
         f = w1;
      else if (S == 2'b10)
         f = w2;
      else
         f = w3;

endmodule
```

图6-31 用**if-else**语句给4选1多路器编码

图6-32给出另一种定义该电路的方法。在这种情况下，用一个4位的向量w来代替四个1比特信号（w_0, w_1, w_2, w_3），而且s的四个不同值用十进制来描述，而不是用二进制数。

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output reg f;

   always @(W, S)
      if (S == 0)
         f = W[0];
      else if (S == 1)
         f = W[1];
      else if (S == 2)
         f = W[2];
      else
         f = W[3];

endmodule
```

图6-32 4选1多路器的另一种描述

```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output f;
   wire [0:3] M;

   mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
   mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
   mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
   mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
   mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule
```

图6-33 16选1多路器的层次化编码 ■

例6.14 图6-4给出了如何用五个4选1多路器来构成一个16选1的多路器。图6-33给出了如何引用五个mux4to1实例模块从电路构造的角度来编写该电路的Verilog代码。输入mux16to1模块的数据是一个16位的向量W，输入选择信号是一个4位的向量$S16$。在这段Verilog代码中，需要为图6-4中左边的四个4选1多路器的输出信号命名，用一个4位的向量M为四个输出命名。$Mux1$为第一个引用的多路器实例，对应于图6-4左边最上面的那个多路器。它的四个输入端口，对应于图中的$w_0, \cdots, w_3$，由信号$W[0], \cdots, W[3]$分别驱动。$S16[1:0]$用于把连到mux4to1模块两位S端口上的$S16[1]$和$S16[0]$拼接在一起。同样，$Mux2$，$Mux3$和$Mux4$分别为引用的多路器实例，代表左边剩下的三个多路器。图6-4中右边的多路器在代码中为引用的多路器实例$Mux5$。信号$M[0]$，…，$M[3]$连接到它的数据输入端，在Verilog语句中用$S16[3:2]$描述的$S16[3]$和$S16[2]$连接到它的选择输入端。实例$Mux5$的输出端产生mux16to1的输出f，编

译这段代码得到如下的多路器表达式：

$$f = \bar{s}_3\bar{s}_2\bar{s}_1\bar{s}_0 w_0 + \bar{s}_3\bar{s}_2\bar{s}_1 s_0 w_1 + \bar{s}_3\bar{s}_2 s_1\bar{s}_0 w_2 + \cdots + s_3 s_2 s_1\bar{s}_0 w_{14} + s_3 s_2 s_1 s_0 w_{15}$$

既然图6-33的代码中实例引用了mux4to1模块，因此需要把图6-32中的代码包含在与mux16to1模块相同文件中，或者把mux4to1模块放在同一路径下的另一个文件中，或者放在一个特殊路径的目录下，以便Verilog编译器能找到它。由观察可知，如果用图6-31中的代码来实现mux4to1模块，则我们不得不把所有的端口分别列出，如用*W*[0]，*W*[1]，*W*[2]，*W*[3]，而不能用一个向量*W*[0：3]就可以表示。 ■

6.6.3 case语句

if-else语句为我们提供了一些手段可以根据表达式值的真伪，在两个分支之间做选择。当可选择的分支有很多时，基于**if-else**语句的代码会使程序晦涩难懂。遇到这种情况，我们通常改用Verilog语言中的**case**语句，其定义如下：

```
case（控制表达式）
    分支表达式1：语句；
    分支表达式2：语句；
      ⋮
    分支表达式j: 语句；
[default：语句；]
endcase
```

控制表达式和每个分支表达式的比较是一位一位地进行的。若遇到一个或多个分支表达式与之匹配时，只有首先与控制表达式匹配的分支表达式后面的语句才能得到执行。若所有的分支表达式都不与控制表达式的值相匹配，则执行默认项**default**后面的语句。否则，Verilog编译器会综合出存储元件（锁存器）来处理没有列出的可能性，我们将在第7章里讨论这种情况。

例6.15 **case**语句可以用来定义4选1多路器，如图6-34所示。选择向量*S*的4个可能的值用十进制数来表示，当然，也可以用二进制数来表示。

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output reg f;

   always @(W, S)
      case (S)
         0: f = W[0];
         1: f = W[1];
         2: f = W[2];
         3: f = W[3];
      endcase

endmodule
```

图6-34 用**case**语句来定义4选1多路器 ■

例6.16 图6-35说明了如何用**case**语句来描述2到4二进制译码器的真值表。这个模块称为dec2to4。输入数据是2位的向量*W*，使能输入信号是*En*。四位向量*Y*代表四个输出信号。在图

6-16a所示的译码器的真值表中，输入端按顺序列为$En\ w_1\ w_0$。为了在控制语句中描述这三个信号，Verilog代码使用了拼接操作符使En和W信号合并为一个三位向量。**case**语句中的四个分支项与图6-16a所示的真值表对应，图中$En = 1$，此时译码器的输出与真值表头四行的输出值是一致的。最后一个分句使用了**default**关键字而且设置译码器的输出为0000，因为它代表了所有其他情况，即$En = 0$时的情况。 ■

例6.17 2到4译码器可以使用**if-else**语句并配合**case**语句的方式来描述，如图6-36所示。当$En = 1$时，才对**case**语句的条件项进行比较；否则，输出信号Y的所有4位都将被设置为0。 ■

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

图6-35 2到4二进制译码器的Verilog代码

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
   begin
      if (En == 0)
         Y = 4'b0000;
      else
         case (W)
            0: Y = 4'b1000;
            1: Y = 4'b0100;
            2: Y = 4'b0010;
            3: Y = 4'b0001;
         endcase
   end

endmodule
```

图6-36 2到4二进制译码器的另一种形式的代码

例6.18 针对图6-18中的4到16译码器的树形结构，可编写出如图6-37所示的代码。输入是一个4位的向量W和一个使能信号En。输出信号是一个16位向量Y。这个电路使用了5个图6-35或图6-36中描述的2到4译码器的实例。在图6-18中最左边的译码器的输出被指定为4位的向量M，如图6-37所示。

```
module dec4to16 (W, Y, En);
   input [3:0] W;
   input En;
   output [0:15] Y;
   wire [0:3] M;

   dec2to4 Dec1 (W[3:2], M[0:3], En);
   dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
   dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
   dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
   dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

图6-37 4到16译码器的Verilog代码 ■

例6.19 图6-38展示了另一个有关**case**语句的例子。seg7模块描述了图6-25所示的7段BCD译码器。BCD译码器的输入是4位的向量bcd，7个输出信号是一个7位向量leds。**case**语句

的分支项按顺序排列，与图6-25c所示的真值表一致。注意，在**case**语句的右侧有一句注释，说明了7个输出信号分别用字母（从*a*到*g*）标记。这些标记向读者说明在Verilog代码中*leds*向量的每位与图6-25所示的7段发光二极管之间的联系。最后一个**case**条件分支把所有的7位设置为x。我们知道，在Verilog语法中x被用来表示无关项。在图6-25中我们曾经讨论过无关项，此时无关项是指译码器的输入向量（*bcd*）不能表示为合法的BCD数字。

```
module seg7 (bcd, leds);
   input [3:0] bcd;
   output reg [1:7] leds;

   always @(bcd)
      case (bcd)   //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase

endmodule
```

图6-38 BCD到7段译码器的Verilog代码 ■

例6.20 算术逻辑单元（ALU）是一个能够对*n*位操作数进行各种布尔和算术操作的逻辑电路。在3.5节中，我们讨论了7400系列的标准芯片。我们曾经提到该系列芯片中的有些芯片包含一些基本的逻辑门，而另外一些则包含一些经常使用的逻辑电路。以74381芯片这个算术逻辑单元（ALU）为例，表6-1说明了74381芯片的功能。该芯片有两个4位的数据输入（*A*和*B*），一个3位的选择输入*s*，和一个4位的输出*F*。正如此表所示，*F* 被定义为输入*A*和*B*的各种布尔或算术操作。在表6-1中，+ 指算术加，－指算术减。为了避免混淆，此表用XOR、OR和AND表示布尔逻辑操作。每一个布尔逻辑操作都以位为单位完成。例如，$F = A$ AND B 会产生一个4位的运算结果：即$f_0 = a_0 b_0$，$f_1 = a_1 b_1$，$f_2 = a_2 b_2$和$f_3 = a_3 b_3$。

表6-1 74381 ALU的功能

操作	输入$s_2 s_1 s_0$	输出 F
Clear	000	0000
$B-A$	001	$B-A$
$A-B$	010	$A-B$
ADD	011	$A+B$
XOR	100	A XOR B
OR	101	A OR B
AND	110	A AND B
Preset	111	1111

图6-39展示了74381ALU的功能是如何用Verilog代码来描述的。**case**语句直接与表6-1相对应。为检查这段代码的功能，我们将该代码段综合成可以在PLD中运行的电路，图6-40是生成电路的时序仿真波形图。对于每个不同的*s*，该电路产生相应的布尔或算术操作。

```
// 74381 ALU
module alu (s, A, B, F);
   input [2:0] s;
   input [3:0] A, B;
   output reg [3:0] F;

   always @(s, A, B)
      case (s)
         0: F = 4'b0000;
         1: F = B - A;
         2: F = A - B;
         3: F = A + B;
         4: F = A ^ B;
         5: F = A | B;
         6: F = A & B;
         7: F = 4'b1111;
      endcase

endmodule
```

图6-39 描述74381 ALU芯片功能性的代码

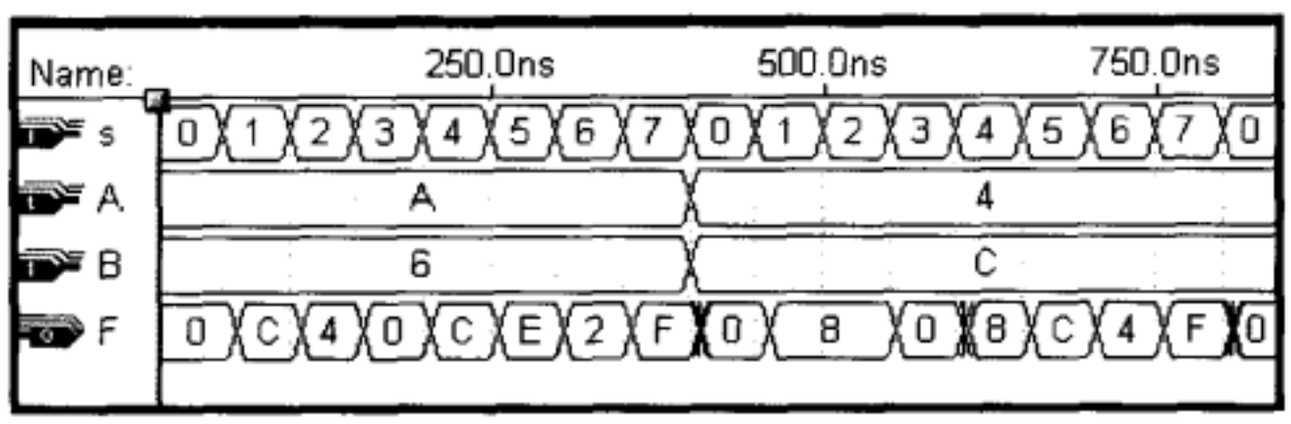


图6-40 图6-39ALU 模块的时序仿真波形图 ■

casex和casez语句

case语句的条件分支项中可能用到0，1，z和x四种不同的逻辑值。确定表达式与某分支项是否匹配需要进行逐位比较。Verilog为**case**语句提供了两种稍有些不同的扩展，它们以不同的方式来处理 x 和 z。**casez**语句认为在条件分支项和控制表达式中出现的所有的 z 都为无关项。而**casex**语句认为所有的 z 和 x 都为无关项。

例6.21 图6-41列出了由图6-24真值表定义的优先级编码器的Verilog代码。预先设想的优先级编码方案，由于使用了**casex**语句，从而变为现实。第一个条件分支说明，若输入w_3是1，则输出$y_1 y_0 = 3$。这个赋值语句与输入w_2, w_1或w_0的值无关；因此它们的值不会影响输出的结果。只有当$w_3 = 0$时，才进行**casex**语句另外一个分支项的匹配。第二个分支项，若$w_2 = 1$，那么$y_1 y_0 = 2$。若$w_2 = 0$，且$w_1 = 1$，则下一个分支项会使$y_1 y_0 = 1$。若 $w_3 = w_2 = w_1 = 0$ 而且$w_0 = 1$，则第四个分支项会使 $y_1 y_0 = 0$。

只要当至少一个输入数据为1时，这个优先级编码器的输出z必然被设定为1。在**always**块开始时，该输出即被置1 。若四个分支项中没有一个与W的值匹配，那么将会执行**default**语句。它由一个两语句的块组成，重新设置z为0，而且说明输出Y可以被设置为任意形式，因为它将被忽略。

```
module priority (W, Y, z);
   input [3:0] W;
   output reg [1:0] Y;
   output reg z;

   always @(W)
   begin
      z = 1;
      casex (W)
         4'b1xxx: Y = 3;
         4'b01xx: Y = 2;
         4'b001x: Y = 1;
         4'b0001: Y = 0;
         default: begin
                     z = 0;
                     Y = 2'bx;
                  end
      endcase
   end

endmodule
```

图6-41 优先级编码器的Verilog代码 ■

6.6.4 for循环语句

若想要设计的电路表现出某种规律性，则该电路就有可能用**for**循环语句来定义。我们曾在5.5.4节中介绍**for**循环语句，该语句对生成常见的行波进位加法器的规范很有用。**for**循环语句的结构如下：

for (循环变量初值; 循环变量终值; 循环步长) 语句;

循环控制变量必须是整型变量，循环开始时，循环变量被设置为初始值。循环控制变量被用在**for**后面的语句中或用在由**begin**和**end**关键字描述的语句块中。在每次迭代后，控制变量增加一个步长。在控制变量达到循环变量终值时迭代停止。

与高级编程语言中的**for**循环不同，Verilog **for** 循环不能具体确定在连续循环迭代中即时发生的变化。然而，在每一次迭代中，它可指定不同的子电路。在图5-29中，用**for**循环语句定义了由全加器子电路级联所组成的一个 n 位行波进位加法器（ripple-carry adder）。**for**循环可用于定义许多其他结构，如下面两个例子所示。

例6.22 图6-42展示了如何用**for**循环语句来描述2到4译码器电路。该循环的效果是针对$k = 0,\cdots,3$重复**if-else**语句四次。若 $W = 0$ 且$En = 1$，则第一个循环设置 $Y[0] = 1$。类似地，另外三个循环会根据W和En的值来设置$Y[1]$，$Y[2]$和$Y[3]$。这种方法可以被用来描述一个很大规模的n到2^n译码器，只需要增加向量W和Y的维数，且令$n-1$为循环变量k的终值即可。 ■

例6.23 图6-24所示的优先级编码器可以由图6-43中列出的Verilog代码来描述。在**always**块中，输出端y_1和y_0首先被置为无关状态，而且z被清为0。然后，若四个输入端$w_3,\cdots,w_0$中的一个或更多等于1，则**for**循环将把y_1y_0的值设置成与值为1最高优先级输入的循环变量相匹配。注意，循环的每一次进行，对应更高一级的优先权。Verilog语法指定在**always**块中被多次赋值的信号将保持最后一次所赋的值。因此，对应最高优先权为1的迭代输入，将覆盖前面任何迭代语句对Y的赋值。 ■

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;
   integer k;

   always @(W, En)
      for (k = 0; k <= 3; k = k+1)
         if ((W == k) && (En == 1))
            Y[k] = 1;
         else
            Y[k] = 0;

endmodule
```

图6-42 用for循环语句描述的2到4二进制译码器

```
module priority (W, Y, z);
   input [3:0] W;
   output reg [1:0] Y;
   output reg z;
   integer k;

   always @(W)
   begin
      Y = 2'bx;
      z = 0;
      for (k = 0; k < 4; k = k+1)
         if (W[k])
         begin
            Y = k;
            z = 1;
         end
   end

endmodule
```

图6-43 用for循环语句描述的优先级编码器

6.6.5 Verilog操作符

在本节中我们将讨论对逻辑电路综合有用的Verilog操作符。表6-2列出了一组操作符，这些操作符反映了所进行的操作。更完整的操作符列表见表A-1。为了说明各种操作所产生的结果，我们将用3位向量$A[2:0]$, $B[2:0]$ 和 $C[2:0]$以及标量f 和w 来说明。

表6-2 Verilog 操作符

操作符类型	操作符号	完成的操作	操作数个数
按位	~	1的补码	1
	&	按位AND	2
	|	按位OR	2
	^	按位XOR	2
	~^或^~	按位XNOR	2
逻辑	!	NOT	1
	&&	AND	2
	||	OR	2
缩减	&	缩减AND	1
	~&	缩减NAND	1
	|	缩减OR	1
	~|	缩减NOR	1
	^	缩减XOR	1
	~^或^~	缩减XNOR	1
算术	+	加	2
	−	减	2
	−	2的补码	1
	*	乘	2
	/	除	2
关系	>	大于	2
	<	小于	2
	>=	大于等于	2
	<=	小于等于	2

（续）

操作符类型	操作符号	完成的操作	操作数个数
等于	==	逻辑相等	2
	!=	逻辑不等	2
移位	>>	右移	2
	<<	左移	2
拼接	{,}	拼接	任意个数
重复	{{}}	重复	任意个数
条件	?:	条件	3

按位操作符

按位操作符按位对操作数进行位运算。~ 操作符对操作数按位取反，举例如下：

C = ~ A ;

产生的结果为 $c_2 = \overline{a}_2, c_1 = \overline{a}_1$ 和 $c_0 = \overline{a}_0$，这里a_i 和c_i 是向量A和C的位。

此外，按位操作符也可对两个多位操作数进行操作。举例如下：

C = A & B;

生成$c_2 = a_2 \cdot b_2, c_1 = a_1 \cdot b_1$和 $c_0 = a_0 \cdot b_0$。类似地，操作符 | 和 ^ 执行按位或和按位异或操作。操作符^~也可以写成~^，按位同或操作举例如下：

C =A ~^ B;

产生的结果为：$c_2 = \overline{a_2 \oplus b_2}$，$c_1 = \overline{a_1 \oplus b_1}$ 和 $c_0 = \overline{a_0 \oplus b_0}$。若两个操作数的位宽不同，则从左侧填0，把短缺位补齐。

两个向量操作数进行按位操作的运算结果可以赋值给标量函数。在此情况下，只有操作数的最低有效位涉及此操作。因此语句

f = A ^ B;

的运算结果产生标量$f = a_0 \oplus b_0$。

按位操作也可能涉及包含不确定逻辑值（x）的操作数。那么，这些操作将按照图6-44中的真值表执行。例如，若 P = 4'b101x 和 Q = 4'b1001，则 P & Q = 4'b100x，而 P | Q = 4'b1011。

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

图6-44 按位运算真值表

逻辑运算符

! 操作符和 ~ 操作符对标量操作的效果一样。因此，f = !w = ~w。若这两个求反的操作

符的操作数是向量，则产生的效果是不同的，即若$f = !A$，则 $f = \overline{a_2 + a_1 + a_0}$ 。

&&操作符实现AND操作，表达式如下：

f = A && B;

产生的结果为：$f = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$。同样，||操作符实现逻辑或操作，表达式如下：

f = A || B;

产生的结果为：$f = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$。

缩减运算符

缩减运算符对单向量操作数的位进行操作，产生1位的输出。使用缩减操作符&的逻辑表达式如下：$f = \&A$。产生的结果为$f = a_2 \cdot a_1 \cdot a_0$。

同样，$f = \wedge A$产生的结果是 $f = a_2 \oplus a_1 \oplus a_0$，其他运算不再赘述。考虑5.8节中曾经讨论过的奇偶校验函数，用这个函数作为使用缩减运算符的例子。产生n位向量X的奇偶校验位p的异或电路可以如下定义：$p = \wedge X$。

算术运算符

我们早在第5章中就已经遇到过算术运算符，它们执行标准的算术运算。 例如加操作符（+）运算表达式$C = A + B$，将A与B的和（3位）赋值给C。而减操作符（－）运算表达式，例如$C = A - B$，将A与B之差赋值给C。而求负操作例如$C = -A$，把A的2的补码赋值给C。

大部分CAD综合工具支持加法、减法和乘法运算，但是通常不支持除法运算。当Verilog编译器遇到算术运算符时，通常必须使用合适的库模块，才能综合成电路。

关系运算符

关系运算符经常在**if-else**语句和**for**语句中用作条件判断。这些运算符与C语言中对应的运算符功能相同。使用关系运算符的表达式如果为真，将返回值1，如果为假，则返回值0。若操作数存在任何x（未知）或者z（高阻）位，则表达式将取值x。

例6.24 图6-45举例说明在**if-else**语句中如何使用关系运算符。定义的电路是曾在6.5节中描述过的4位比较器。

```
module compare (A, B, AeqB, AgtB, AltB);
   input [3:0] A, B;
   output reg AeqB, AgtB, AltB;

   always @(A, B)
   begin
      AeqB = 0;
      AgtB = 0;
      AltB = 0;
      if (A == B)
         AeqB = 1;
      else if (A > B)
         AgtB = 1;
      else
         AltB = 1;
   end

endmodule
```

图6-45 4位比较器的 Verilog 代码

等于运算符

若A等于B，则表达式($A == B$)为真，否则为假。!=（不等）操作符的功能正好相反。若任一操作数中含有 x 或 z 值的位，则结果为不定值。

移位运算符

根据常量指定的位数，可对向量操作数进行相应位数的左移或者右移操作。移位后出现的空位要用0来填补。例如下列表达式：

B = A<< 1;

执行后产生的结果为：$b_2 = a_1, b_1 = a_0, b_0 = 0$。类似地，$B = A>> 2$; 执行后产生的结果为：$b_2 = b_1 = 0, b_0 = a_2$。

位拼接操作符

位拼接操作符将两个或者几个向量拼接成一个更大的向量。例如，

D = {A, B};

定义一个6位向量$D = a_2 a_1 a_0 b_2 b_1 b_0$。类似地，下面的拼接操作：

E = {3'b111, A, 2' b00};

生成一个8位向量$E = 111a_2 a_1 a_0 00$。

复制操作符

复制操作符允许对同一个向量进行重复多次的拼接，而重复的次数将在重复常量中指明。例如，{3{A}}与{A, A, A}相同。{4{2'b10}}产生一个8位向量10101010。

复制操作符可以与拼接操作符配合起来应用。例如，{2{A}, 3{B}}等价于{A, A, B, B, B}。我们在5.5.6节曾经介绍过拼接和复制操作符，而且举例说明了它们在加法器电路中的应用。

条件操作符

在6.6.1节中我们已经对条件操作符做过全面的介绍。

操作符的优先级

Verilog操作符的优先级在表6-3中说明。优先级的顺序从顶到底；位于顶部行的操作符优先级别最高，在底部行的优先级别最低。同一行的操作符优先级相同。

表6-3 Verilog操作符的优先级

操作符类型	操作符号	优先级
取反	! ~ −	最高优先级
算术	* /	
	+ −	
移位	<< >>	
关系	< <= > >=	
等于	== !=	
缩减	& ~&	
	^ ~^	
	| ~|	
逻辑	&&	
	||	
条件	?:	最低优先级

设计者可以使用括弧改变Verilog代码中操作符的优先级顺序，以避免任何可能发生的误解。使用括弧可以使代码的含义明确易懂。

6.6.6 生成结构

在5.5.4节中引入了**generate**循环能力，这个循环能力可以用于创建多个子电路实例。可以由关键字**generate**（生成）和**endgenerate**（终止生成）定义的语句块来定义子电路块。用生成指针变量可多次实例引用该子电路块。此变量可以用**genvar**关键字来定义，而且它只能取正整数值。注意，普通的整型变量不能用作生成结构的指针变量。

例6.25 图6-46说明如何用生成结构来描述一个n位的行波进位加法器。此子电路是一个全加器，该全加器的结构用图5-23所示的原语（primitive）门定义。**for**循环使全加器原语模块实例引用了n次。

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   wire [n:0] C;

   genvar k;
   assign C[0] = carryin;
   assign carryout = C[n];
   generate
      for (k = 0; k < n; k = k+1)
      begin: fulladd_stage
         wire z1, z2, z3;  //wires within full-adder
         xor (S[k], X[k], Y[k], C[k]);
         and (z1, X[k], Y[k]);
         and (z2, X[k], C[k]);
         and (z3, Y[k], C[k]);
         or (C[k+1], z1, z2, z3);
      end
   endgenerate

endmodule
```

图6-46 用生成循环语句定义n位行波进位加法器

在这个例子中，**for**语句被用在生成块中，用以控制生成对象的选择。此生成块也可以含有**if-else**语句和**case**语句，用以决定生成哪个对象。■

生成结构是一个有效的机制，大大增强了Verilog的设计能力。因为生成结构最近才增补进入Verilog标准，所以有些综合工具不支持生成结构的综合。

6.6.7 任务和函数

在高级编程语言中，为了避免在程序中多次重复编写特定的例行程序，可以使用子程序和函数。Verilog语言也提供类似的能力，即Verilog的任务和函数。它们可以把大型设计模块化，从而使Verilog代码更加容易理解。

Verilog任务

任务用关键字**task**声明，其中包含若干条语句，最后以**endtask**结尾。这样的语句模块就称为任务。任务必须包含在调用该任务的模块中。它可以有输入和输出端口。任务的端口并

不是调用该任务模块的端口，而是用来将它和调用它的模块连接的端口。任务端口仅仅用来在模块和任务之间传递数据。

例6.26 在图6-33中，我们列出了16选1多路选择器的Verilog代码，该代码实例引用了五个4选1多路选择器电路，这个4选1多路器电路由一个独立的名为mux4to1的模块提供。16选1多路选择器也可以使用任务来描述，见图6-47所示。注意观察关键的不同点。任务mux4to1包含在模块mux16to1中。在**always**模块中通过**case**语句来调用任务。任务的输出必须是一个变量，因此*g*是**reg**型。

```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   always @(W, S16)
      case (S16[3:2])
         0: mux4to1 (W[0:3], S16[1:0], f);
         1: mux4to1 (W[4:7], S16[1:0], f);
         2: mux4to1 (W[8:11], S16[1:0], f);
         3: mux4to1 (W[12:15], S16[1:0], f);
      endcase

   // Task that specifies a 4-to-1 multiplexer
   task mux4to1;
      input [0:3] X;
      input [1:0] S4;
      output reg g;

      case (S4)
         0: g = X[0];
         1: g = X[1];
         2: g = X[2];
         3: g = X[3];
      endcase
   endtask

endmodule
```

图6-47 Verilog代码中任务的使用

Verilog函数

函数用关键字**function**声明，包含若干条语句，最后以关键字**endfunction**结尾。函数必须至少有一个输入，而且只返回一个值到函数被调用的位置。

例6.27 图6-48说明了如何使用函数来完成图6-47中的代码。Verilog编译器实质上在每一个调用函数的地方插入函数。因此，条件分支语句

```
0: f = mux4to1 (W[0:3], S16[1:0]);
```

变成

```
0: case (S16[1:0])
      0: f = W[0];
      1: f = W[1];
      2: f = W[2];
      3: f = W[3];
   endcase
```

函数使mux16to1模块变得紧凑和易读。

```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   // Function that specifies a 4-to-1 multiplexer
   function mux4to1;
      input [0:3] X;
      input [1:0] S4;

      case (S4)
         0: mux4to1 = X[0];
         1: mux4to1 = X[1];
         2: mux4to1 = X[2];
         3: mux4to1 = X[3];
      endcase
   endfunction

   always @(W, S16)
      case (S16[3:2])
         0: f = mux4to1 (W[0:3], S16[1:0]);
         1: f = mux4to1 (W[4:7], S16[1:0]);
         2: f = mux4to1 (W[8:11], S16[1:0]);
         3: f = mux4to1 (W[12:15], S16[1:0]);
      endcase

endmodule
```

图6-48 用函数完成图6-47中的代码

Verilog函数可以调用另外一个函数，但不能调用另外一个Verilog任务。而任务可以调用另一个任务也可以调用函数。在图6-47中，在调用某任务的**always**块后，才定义该任务。而在图6-48中，在调用某函数的**always**块之前就已定义了该函数。对任务和函数来说，这两种情况Verilog标准都是允许的。但是，有些CAD工具要求必须先定义好函数，才能调用函数。

6.7 小结

本章介绍了一些电路结构块。使用这些块构建大型电路的例子将在第7章和第10章中介绍。为了有效地描述结构块电路，我们已经介绍了一些Verilog结构。在许多情况下，可以使用不同的结构以用不同的方式描述一个给定的电路。能够用**if-else**语句描述的电路也可以用**case**语句或者也许可以用**for**循环语句来描述。总而言之，没有严格的规则规定何时哪一种风格会比另一种风格更好。随着经验的积累，使用者会培养出感觉，可以判断出在特定的设计环境中哪种风格会工作得更好些。个人的偏好也会影响Verilog代码的风格。

Verilog不是编程语言，而且Verilog代码的编写也与计算机程序的编写不同。本章所介绍的语句可以用于生成大型、复杂的电路。设计这样的电路的一种好方法是使用已定义好的模块来构建它。我们举例说明了如何用这个方法进行多路选择器、译码器、编码器等的设计。在第7章和第8章中还将再举一些用本章介绍过的语句描述的例子。在第10章中，提供了几个用Verilog代码描述大型数字系统的例子。若要对Verilog有更多了解，读者可以查阅更专业的参考书籍[5-11]。

在下一章中，我们将介绍能够在记忆元件中存储逻辑信号值的逻辑电路。

6.8 问题求解举例

本节列举了读者可能遇到的某些典型问题，并且展示了这些问题是如何求解的。

例6.28 设有逻辑函数 $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$，请用3-8二进制译码器和或门实现该函数。

解：该译码器为函数的每个最小项分别产生各自的输出，然后将这些输出连接到一个或门的输入端。实现该逻辑函数的电路如图6-49所示。

例6.29 推导实现8-3二进制编码器的电路。

解：该译码器的真值表如图6-50所示。在该真值表中，我们只画出了单个变量为1的那些行，其他行被当作无关项处理。从该真值表可以看到，想要的电路可以用下列表达式定义：

$$y_2 = w_4 + w_5 + w_6 + w_7$$

$$y_1 = w_2 + w_3 + w_6 + w_7$$

$$y_0 = w_1 + w_3 + w_5 + w_7$$

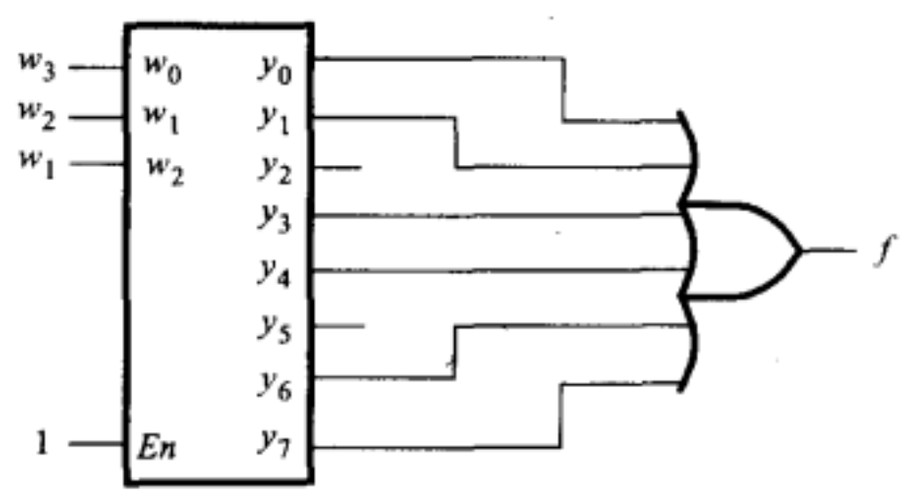


图6-49 例6.28的电路

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

图6-50 8-3二进制编码器的真值表

例6.30 设有逻辑函数

$$f(w_1, w_2, w_3, w_4) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

请用4选1多路器和尽可能少的门实现该逻辑函数，假设输入信号w_1, w_2, w_3, w_4和w_5全都没有可用的反相信号。

解：因为输入变量w_1和w_4比其他三个变量更多地出现在该逻辑函数的乘积项中，所以对这两个变量进行香农展开（Shannon's expansion）。上述表达式变为：

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1 w_4 f_{w_1 w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1 w_4(w_3 w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1 w_2(1) \end{aligned}$$

可以用或非门来实现 $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$。我们还需要一个与门和一个或门。完整的电路如图6-51所示。

例6.31 在第4章中，我们曾指出卡诺图的行和列的标号都采用格雷码。采用格雷码的任意两个相邻码只有一位不同。图6-52画出了三位二进制码及其对应的格雷码。请根据图6-52，设计一个可将二进制码转换成格雷码的电路。

解：根据图6-52，通过观察很容易就可以得到如下表达式：

$$
\begin{aligned}
g_2 &= b_2 \\
g_1 &= b_1\overline{b}_2 + \overline{b}_1 b_2 \\
&= b_1 \oplus b_2 \\
g_0 &= b_0\overline{b}_1 + \overline{b}_0 b_1 \\
&= b_0 \oplus b_1
\end{aligned}
$$

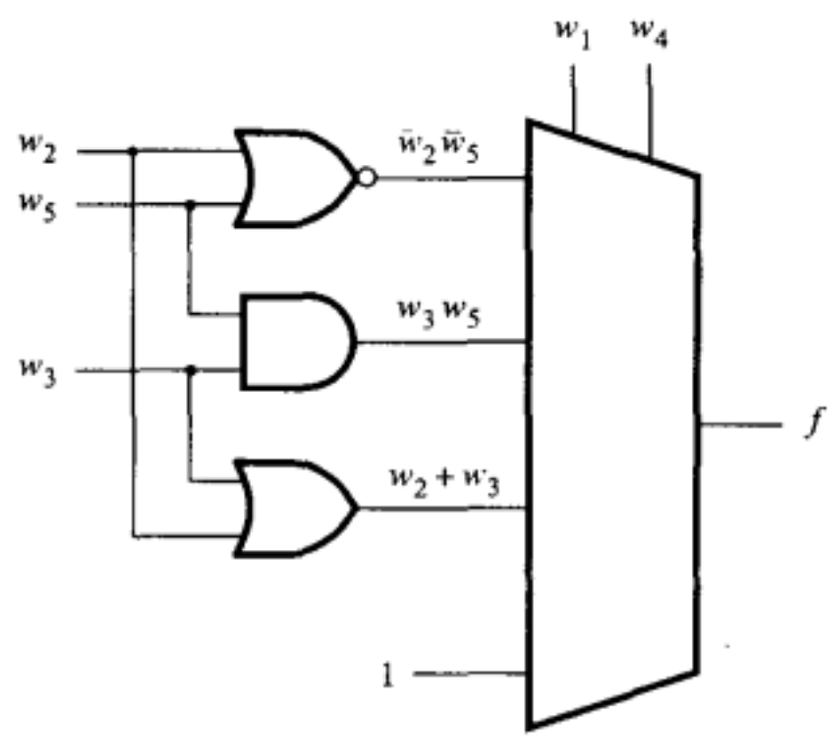


图6-51 例6.30的电路

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

图6-52 二进制码到格雷码的转换

例6.32 在6.1.2节中，我们曾展示过任何逻辑函数都可以用香农展开定理分解。对一个四变量函数$f(w_1, w_2, w_3, w_4)$中的w_1进行香农展开，即

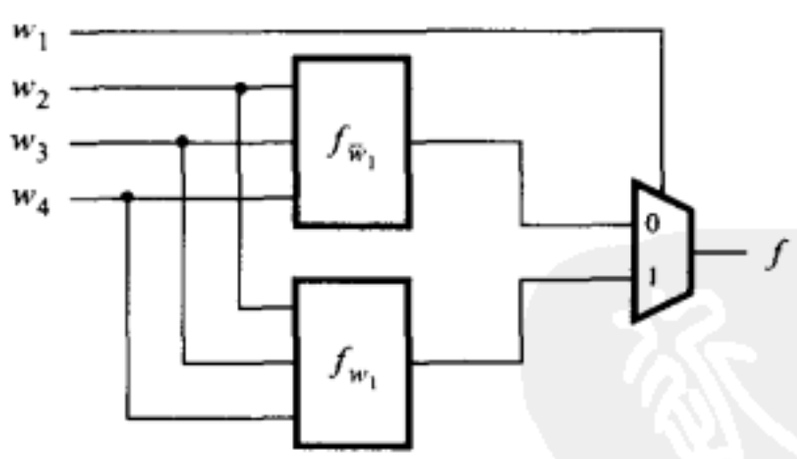
$$f(w_1, \cdots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

实现上述表达式的电路如图6-53a所示。

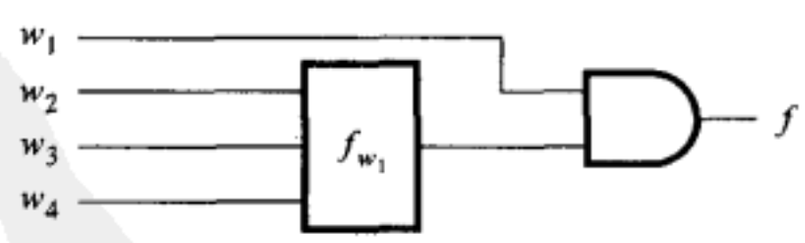
(a) 若分解产生 $f_{\overline{w}_1} = 0$，则图中的多路器可以被单独一个逻辑门所替换。画出该电路。

(b) 若分解产生 $f_{w_1} = 1$，重复求解 (a) 中的问题。

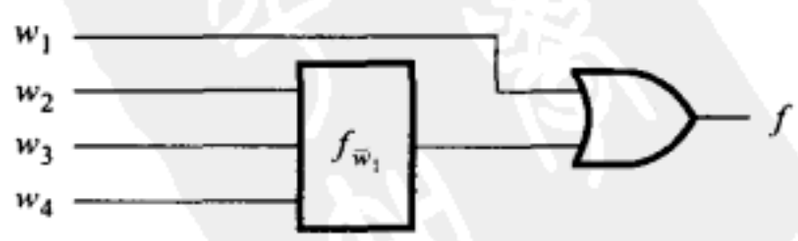
解：想要的电路如图6-53b和c所示。



a) 函数f的香农展开



b) 问题(a)的解答



c) 问题(b)的解答

图6-53 例6.32的电路

例6.33 有几种商用FPGA，其逻辑块采用四输入查找表。为了构建一个4选1多路器，最少需要几个四输入查找表？设4选1多路器的选择输入为s_1和s_0，数据输入为w_3, w_2, w_1和w_0。

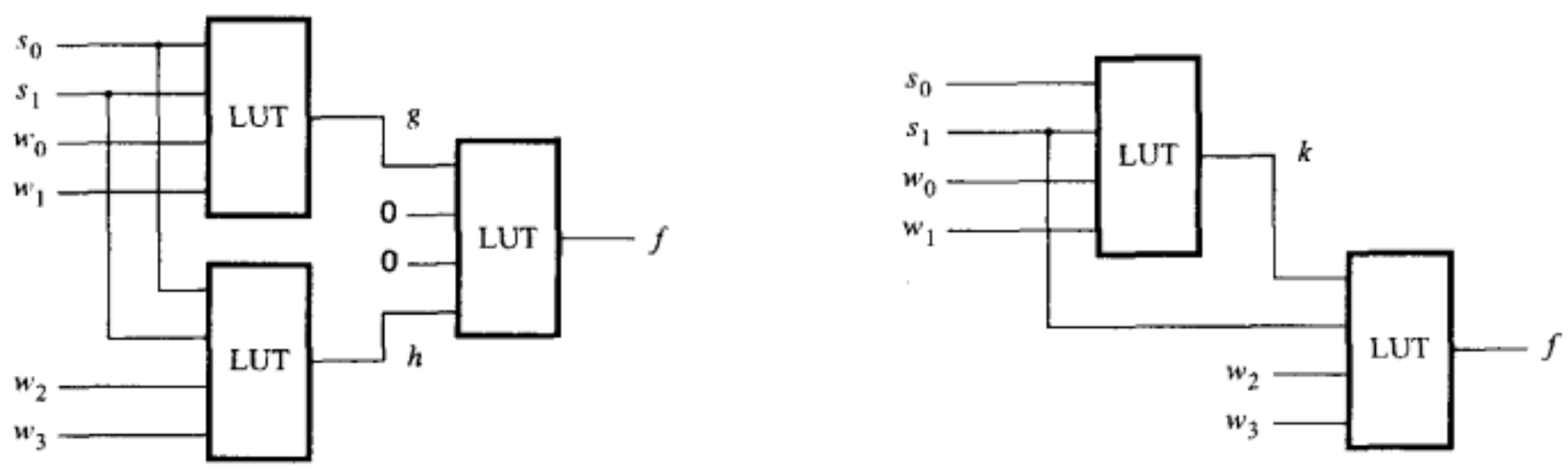
解：最简单的办法是直接利用定义4选1多路器的逻辑表达式：

$$f = \bar{s}_1\bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1\bar{s}_0 w_2 + s_1 s_0 w_3$$

令 $g = \bar{s}_1\bar{s}_0 w_0 + \bar{s}_1 s_0 w_1$ 和 $h = s_1\bar{s}_0 w_2 + s_1 s_0 w_3$，则 $f = g + h$。根据这个分解可推导出如图6-54a所示的电路，该电路需要三个四输入查找表。

在设计逻辑电路时，有时会涌现一个很聪明的想法导致更简单的电路实现。图6-54b展示了如何根据以下观察，有可能用两个四输入查找表来实现该4选1多路器。图6-2b的真值表表明：当$s_1 = 0$时，输出是w_0还是w_1由s_0的值决定。这可以由第一个查找表产生。第二个查找表在$s_1 = 1$时，必须在w_2或w_3之间做出选择。但是这个选择只有在知道s_0值的前提下才能做出。因为四输入查找表不可能有第五个输入，所以必须将更多的信息从第一个查找表传送到第二个查找表。观察到当$s_1 = 1$时，输出f将在w_2或w_3之间做出选择，此时没有必要知道w_0和w_1的值。因此，这时可以通过配置第一个查找表将s_0的值（而不是w_0或者w_1的值）传到第二个查找表。这可以通过将第一个查找表配置为下列表达式来实现：

$$k = \bar{s}_1\bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1 s_0$$



a) 用三个查找表　　b) 用两个查找表

图6-54 例6.33的电路

然后，将第二个查找表配置为以下表达式来实现：

$$f = \bar{s}_1 k + s_1\bar{k} w_3 + s_1 k w_4$$

例6.34 在数字系统中经常有必要使用能够将向量向左或者向右移动一位或多位的电路。请设计一个电路，当控制信号*Shift*等于1时，能将一个四位的向量$W = w_3 w_2 w_1 w_0$向右移动一位。令电路的输出是一个四位的向量$Y = y_3 y_2 y_1 y_0$，以及一个信号k，使得电路具有这样的功能：若$Shift = 1$，则$y_3 = 0, y_2 = w_3, y_1 = w_2, y_0 = w_1$。若$Shift = 0$，则$Y = W, k = 0$。

解：可以用如图6-55所示的五个2选1多路器实现这个想要的电路。控制信号*Shift*被用作每个多路器的选择输入信号。

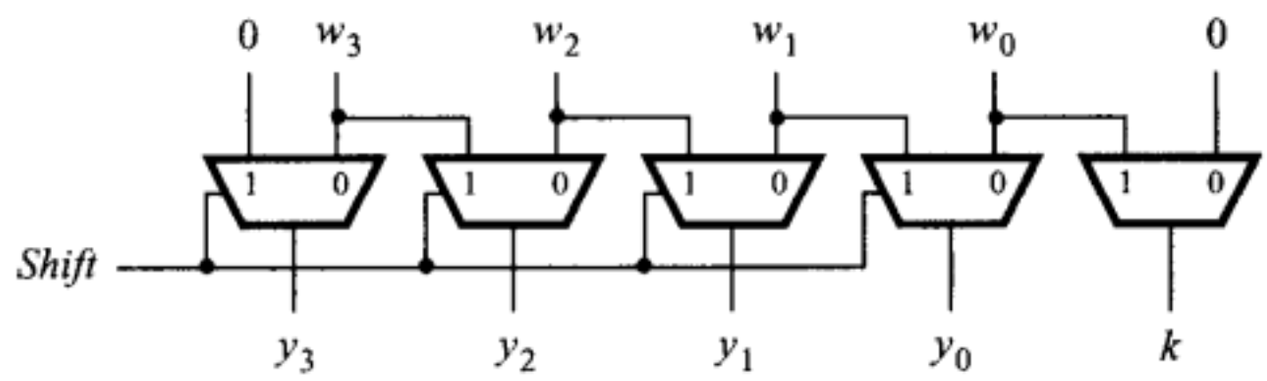


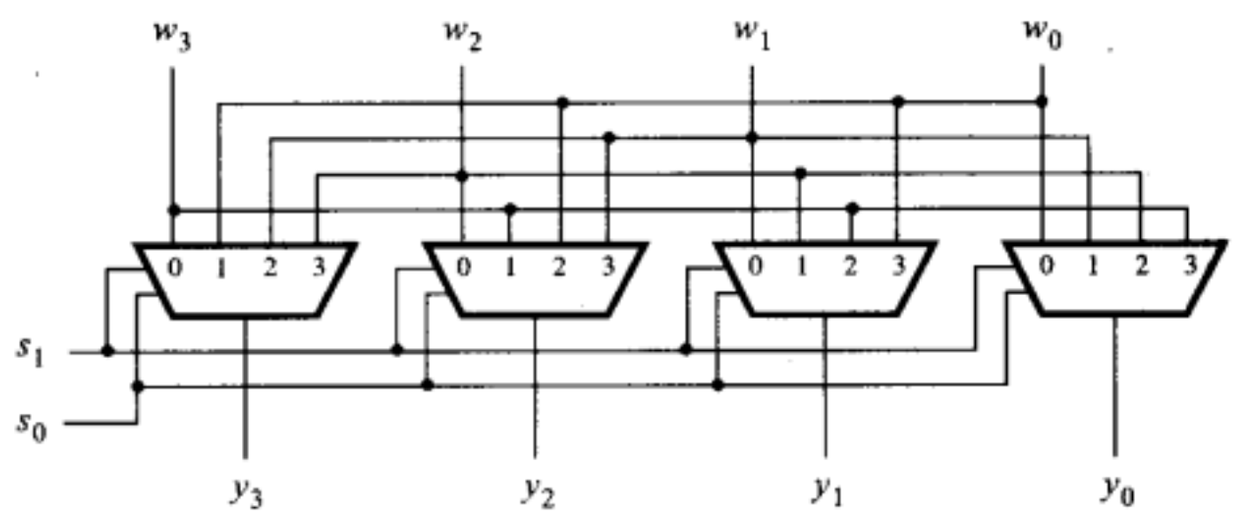
图6-55 移位器电路

例6.35 例6.34中的移位器电路可以将输入向量的每个位向右移动一位。移位后将最左边的空位填为0。更灵活的电路每次可以移动多位。如果向右移出的位能够放置到空出来的最左边的位，则该电路可以有效地将输入向量旋转着移动指定的位数。这样的电路叫做*环型移位器*（barrel shifter），也称桶状移位器。请设计一个由两个控制信号s_1和s_0控制的可以分别旋转移动0，1，2或者3位的环型移位器。

解：图6-56a给出了电路需要的动作。环型移位器可以用四个4选1多路器实现，如图6-56b所示。控制信号s_1和s_0被用作输入多路器的选择信号。

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

a) 真值表



b) 电路

图6-56 环型移位器电路

例6.36 写出表示图6-19所示电路的Verilog代码。使用图6-35中的dec2to4模块作为所编写的代码中的子模块。

解：该代码如图6-57所示。请注意dec2to4 模块可以包括在同一个文件中，正如图6-57中所示的那样，但也可以放在属于该项目目录的另外一个独立文件中。

例6.37 写出表示图6-55所示的移位器电路的Verilog代码。

解：方法之一是指定如图6-58Verilog代码所表示的电路结构。**if-else**语句结构用来定义想要移动的位。典型的Verilog编译器将用图6-55所示的2选1多路器来实现这个代码。另外一种方法是利用6.6.5节定义的移位操作符，如图6-59所示。

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output f;
   wire [0:3] Y;

   dec2to4 decoder (S, Y, 1);
   assign f = |(W & Y);

endmodule

module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

图6-57 例6.36的Verilog代码

```
module shifter (W, Shift, Y, k);
   input [3:0] W;
   input Shift;
   output reg [3:0] Y;
   output reg k;

   always @(W, Shift)
   begin
      if (Shift)
      begin
         Y[3] = 0;
         Y[2:0] = W[3:1];
         k = W[0];
      end
      else
      begin
         Y = W;
         k = 0;
      end
   end

endmodule
```

图6-58 图6-55所示电路的Verilog代码

```
module shifter (W, Shift, Y, k);
   input [3:0] W;
   input Shift;
   output reg [3:0] Y;
   output reg k;

   always @(W, Shift)
   begin
      if (Shift)
      begin
         Y = W >> 1;
         k = W[0];
      end
      else
      begin
         Y = W;
         k = 0;
      end
   end

endmodule
```

图6-59 描述图6-55所示电路的另外一种Verilog代码

例6.38 写出描述如图6-56所示的环型移位器的Verilog代码。

解：图6-60所示的代码是一种可能的解决方案。环型移位的功能是这样产生的：将两个输入向量W拼接起来，然后将由拼接得到的8位向量向右移位，移动的位数由输入变量S控制，于是该8位向量中的高四位便形成想要的输出Y。

```
module barrel (W, S, Y);
   input [3:0] W;
   input [1:0] S;
   output [3:0] Y;
   wire [3:0] T;

   assign {T, Y} = {W, W} >> S;

endmodule
```

图6-60 描述环型移位器电路的Verilog代码

练习题

标有*号的习题答案参见书后。

6.1 说明如何用一个3到8二进制译码器和一个或门来实现函数$f(w_1, w_2, w_3) = \sum m\ (0,2,3,4,5,7)$。

6.2 说明如何用一个3到8二进制译码器和一个或门来实现函数$f(w_1, w_2, w_3) = \sum m\ (1,2,3,5,6)$。

***6.3** 设逻辑函数 $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$,使用真值表推导出用2选1多路器实现该逻辑函数的电路。

6.4 设逻辑函数 $f = \overline{w}_2\overline{w}_3 + w_1 w_2$ ，使用真值表推导出用2选1多路器实现该逻辑函数的电路。

***6.5** 设逻辑函数$f(w_1, w_2, w_3) = \sum m\ (0,2,3,6)$，用香农展开定理推导出用2选1多路器和其他必要门实现该逻辑函数的电路。

6.6 设逻辑函数为$f(w_1, w_2, w_3) = \sum m\ (0,4,6,7)$，同6.5题的要求。

6.7 设逻辑函数为 $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1 w_3$ ，说明如何多次重复应用香农展开定理来推导出 f 的最小项。

6.8 设逻辑函数为 $f = w_2 + \overline{w}_1\overline{w}_3$ ，同6.7题的要求。

6.9 证明在6.1.2节中介绍的香农展开定理。

***6.10** 6.1.2节展示了用积之和形式表示的香农展开定理。用对偶性原理，求出其等价的和之积表达式。

6.11 设逻辑函数为 $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1 w_2 w_3$ ，利用最少个数的2输入查找表设计出实现该逻辑函数的电路，并写出每个查找表所实现的真值表。

***6.12** 就上题中的函数，最简积之和表达式的门电路成本是14，其中包含4个门和输入到这4个门的10个信号。应用香农展开定理推导出一个多级电路，使之成本更低，并且给出电路的成本。

6.13 设逻辑函数为：$f(w_1, w_2, w_3, w_4) = \sum m\ (0,1,3,6,8,9,14,15)$，用最少个数的3输入查找表设计该电路。

***6.14** 举出两个逻辑函数的例子，使其都有五个输入信号：$w_1, \cdots, w_5$，并可以用两个4输入查找表来实现该逻辑函数。

6.15 对例6.30中的函数f，关于变量w_1和w_2而不是w_1和w_4进行香农展开，与图6-51中的电路相比较，得到的电路是什么样的?

6.16 Actel公司生产了一款名为Act1的FPGA系列器件，该器件含有基于多路器的逻辑块，如图P6-1所示。请说明如何仅用一个Act1逻辑块就可以实现以下逻辑函数：

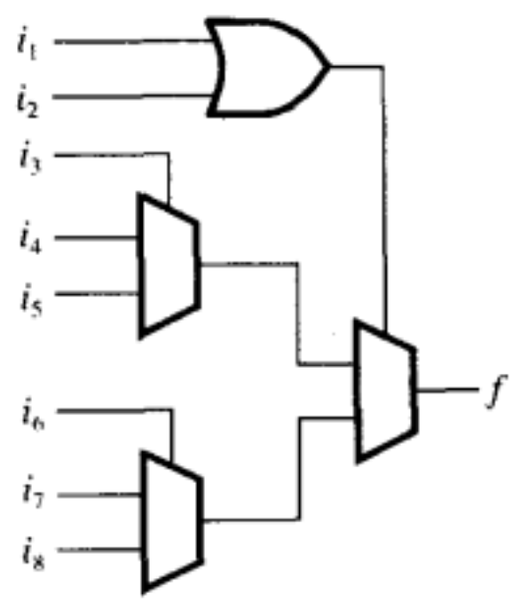
$$f = w_2\overline{w}_3 + w_1 w_3 + \overline{w}_2 w_3$$

6.17 说明如何使用Act1逻辑块来实现以下逻辑函数：

$$f = w_1\overline{w}_3 + \overline{w}_1 w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$$

请注意，此芯片中没有非门，因此信号的求反必须用该逻辑块中的多路选择器产生。

***6.18** 考虑图P6-2中的Verilog代码，此代码代表的电路是什么类型？请评价该代码的风格对所描述的电路而言是否是一种好的选择。



图P6-1 Actel Act 1逻辑块

```
module problem6_18 (W, En, y0, y1, y2, y3);
   input [1:0] W;
   input En;
   output reg y0, y1, y2, y3;

   always @(W, En)
   begin
      y0 = 0;
      y1 = 0;
      y2 = 0;
      y3 = 0;
      if (En)
         if (W == 0)  y0 = 1;
         else if (W == 1)  y1 = 1;
         else if (W == 2)  y2 = 1;
         else  y3 = 1;
   end

endmodule
```

图P6-2 习题6.18的代码

6.19 使用**case**语句编写描述习题6.2的逻辑函数的Verilog代码。

6.20 用Verilog语言编写描述4到2二进制编码器的代码。

6.21 用Verilog语言编写描述8到3二进制编码器的代码。

6.22 图P6-3列出了图6-42所示的2到4译码器代码的改进版。此代码除了有一个错误外其余都是正确的。错误在哪里?

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output reg [0:3] Y;
   integer k;

   always @(W, En)
      for (k = 0; k <= 3; k = k+1)
         if (W == k)
            Y[k] = En;

endmodule
```

图P6-3 习题6.22的代码

6.23 推导8到3优先级编码器的电路。

6.24 用**casex**语句，编写8到3优先级编码器的Verilog代码。

6.25 用**for**循环语句，编写8到3优先级编码器的Verilog代码。

6.26 用**if-else**语句编写一个名为if2to4的Verilog模块，该模块描述一个2到4译码器；再编写另一个名为h3to8的Verilog模块，该模块描述一个3到8译码器。

6.27 编写描述（二进制）6到64译码器的Verilog模块，该模块的名字为h6to64。请用图6-18所示的树状结构，在图中，6到64译码器实例引用了9个习题6.26所创建的h3to8译码器模块。

6.28 编写描述图6-19所示电路的Verilog代码。请在你的代码中使用图6-35所示的dec2to4模块作为子电路。

* **6.29** 求出图6-25所示7段数码显示器的输出信号a，b和c的最简的积之和表达式。

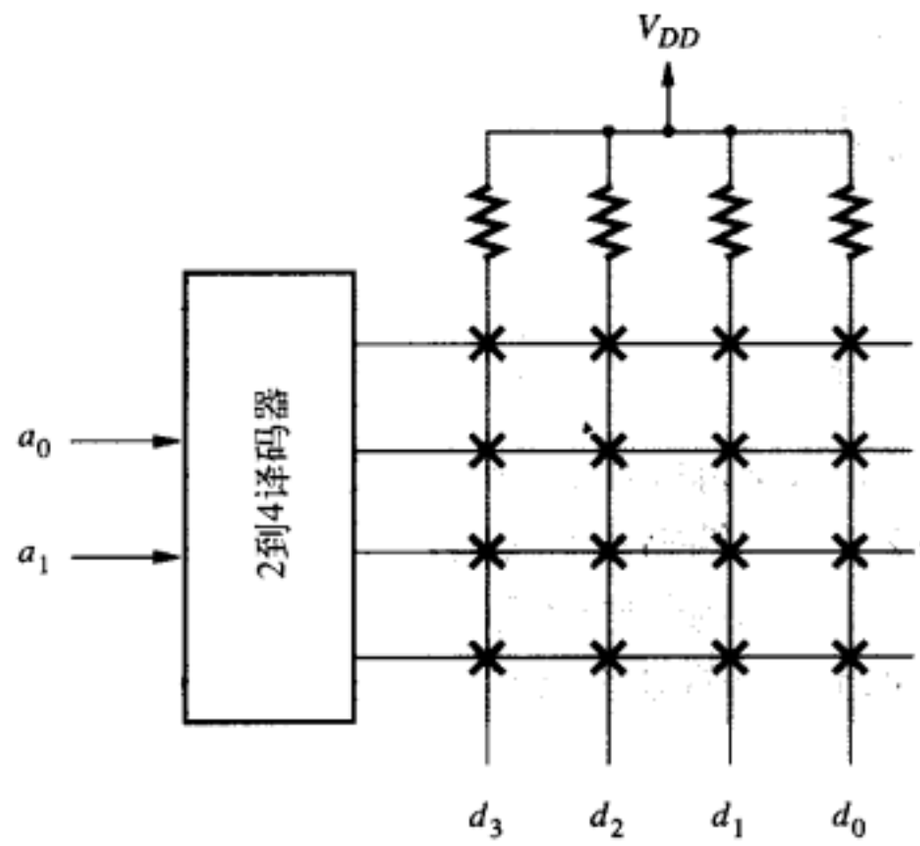
6.30 求出图6-25所示7段数码显示器的输出信号d，e，f和g的最简的积之和表达式。

6.31 图6-21展示了ROM结构方块图。图P6-4所示的电路实现了一个有4行和4列的小型ROM。图中每个×代表一个开关，当该地址被读到时，该开关决定了ROM产生的值为1还是0。

(a) 说明如何用一个NMOS晶体管来实现×表示的开关。

(b) 用你在（a）中所设计的开关，画出完整的4×4 ROM电路图。该ROM应该被编程为存储以下数据：第0行（顶行）存储数据0101，第1行存储1010，第2行存储1100，第3行（底行）存储0011。

(c) 用图3-64所示的EEPROM单元，说明如何用可编程开关（而不是永远为1或者0）来实现每一个(×)的。简要说明如何使用该存储单元。



图P6-4 一个4 × 4 ROM 电路

6.32 利用习题6.31的(a) 中设计的存储单元，画出一个可以完成如下逻辑功能的小规模ROM的完整电路。

$$d_3 = a_0 \oplus a_1$$
$$d_2 = \overline{a_0 \oplus a_1}$$
$$d_1 = a_0 a_1$$
$$d_0 = a_0 + a_1$$

参考文献

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.
2. Actel Corporation, "MX FPGA Data Sheet," *http://www.actel.com.*
3. QuickLogic Corporation, "pASIC 3 FPGA Data Sheet," *http://www.quicklogic.com.*
4. R. Landers, S. Mahant-Shetti, and C. Lemonds, "A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays," *IEEE Journal of Solid-State Circuits* 30, no. 4 (April 1995).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
6. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
7. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed.,

(Prentice-Hall: Upper Saddle River, NJ, 2003).

8. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

第7章 触发器、寄存器、计数器和简单处理器

在前面几章中，我们曾讨论过组合电路，组合电路的每个输出信号值仅取决于施加于输入端的信号值。还存在另一类逻辑电路，其电路的输出值不只是取决于当前的输入值，也取决于电路过去的行为。这类电路包括可以存储逻辑信号值的存储元件，存储元件的内容代表了电路的状态。当电路的输入发生改变时，新输入的值可以使电路的状态不发生任何变化，也可以使电路进入另一状态。由于输入信号发生变化，从而使电路的状态随时间的进展发生一系列变化，这样的电路就称为时序电路。

本章将介绍可以用作存储元件的电路。但是，首先我们将借助于一个简单例子来说明为什么需要这种电路。假设我们需要对报警器进行控制，如图7-1所示，报警器具有对控制输入On/$\overline{\text{Off}}$做出响应的机制，当On/$\overline{\text{Off}}$ = 1时，报警器启动，当On/$\overline{\text{Off}}$ = 0时报警器关闭。想要报警器执行的操作是：当发生意外事件时，传感器立即产生一个正电压信号（即Set），随即接通报警开关。一旦报警器触发后，报警器必须保持有效，即使传感器的输出已经返回0，报警器也要继续发出警报。借助于Reset输入信号，可以手动关闭报警器。电路需要一个记忆电路来记住：复位信号到达之前报警器必须一直保持有效。

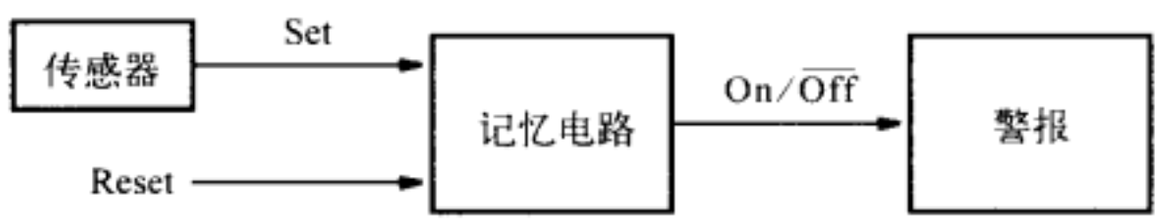


图7-1 一个警报系统的控制结构

图7-2所示为简单记忆电路的工作原理，记忆电路由两个非门组成。若我们假设$A = 0$，则$B = 1$，于是电路将会永远保持这些值。我们称这个电路处于由这些值定义的状态中。若我们假设$A = 1$，则$B = 0$，于是电路就会永远处于第二种状态。因此这个电路有两种可能的状态。但是该电路没有实际用途，因为缺少改变电路状态的具体手段。

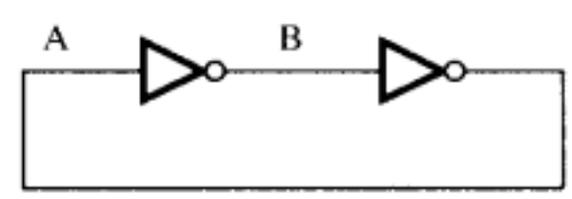


图7-2 一个简单的记忆电路

图7-3所示是一个比较有用的电路。在图7-2所示电路的基础上，图7-3添加了两个传输门（3.9节讨论过的那类传输门），用以改变电路的状态。第一个传输门（TG1）用来连接数据输入端和电路中的点A；第二个传输门（TG2）在反馈回路中充当开关，用以保持电路的状态。

传输门由加载信号*Load*控制。若$Load = 1$，则*TG*1导通，点*A*将与输入信号*Data*的值相同。因为当前存储在*Output*的值有可能与*Data*的值不同，当$Load = 1$时*TG*2截止，从而反馈回路断开，不管电路原来存储的是什么值，现在将使得$Output = A = Data$。当*Load*变为0时，*TG*1截止，*TG*2导通，反馈回路导通，只要$Load = 0$记忆电路就会一直保持其已有的状态。这个记忆电路虽然不能直接应用于图7-1所示的系统，但在许多其他领域已有广泛的应用，我们后面将会介绍。

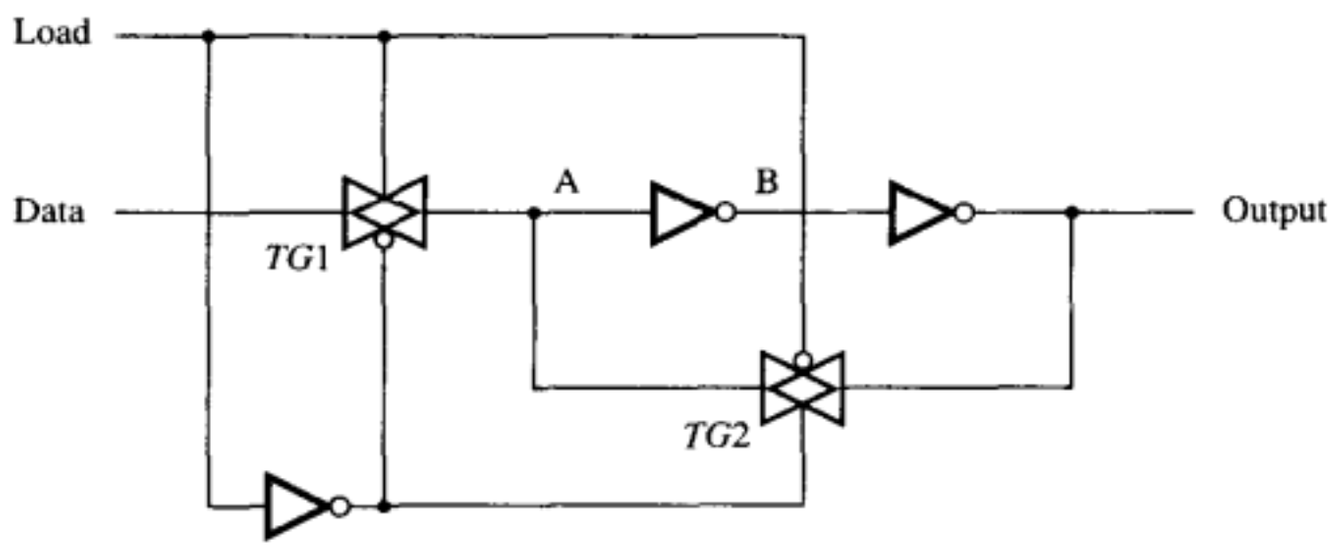


图7-3 一个可控的记忆电路

7.1 基本锁存器

不用传输门，只用普通的逻辑门，我们也可以来构建类似的记忆电路。图7-4所示是由或非门构成的记忆电路。它的输入信号*Set*（置位）和*Reset*（复位），可以用来改变存储元件的电路状态*Q*。图7-5a是该记忆电路的另一种更常用的画法，电路中的两个或非门用交叉耦合的方式相连，组成基本锁存器。图7-5b中的特征表描述了它的特性。当复位端*R*、置位端*S*同时为0时，锁存器保持原状态。这个状态可能是$Q_a = 0$，$Q_b = 1$；也可能是$Q_a = 1$，$Q_b = 0$，在特征表中输出端Q_a、Q_b的值分别用0/1和1/0表示。注意，本例中Q_a和Q_b的值是互为反向的。当$R = 0$，$S = 1$时，锁存器置位，$Q_a = 1$，$Q_b = 0$。当$R = 1$，$S = 0$时，锁存器复位，$Q_a = 0$，$Q_b = 1$。第四种情况是$R = S = 1$，这时Q_a、Q_b均为0。图7-5b中的表类似于真值表。但是，由于它不表示组合电路，组合电路的输出值仅由当前输入值确定，所以通常称它为特征表而不是真值表。

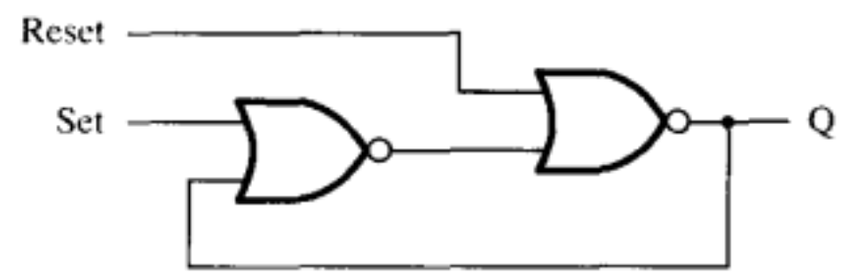
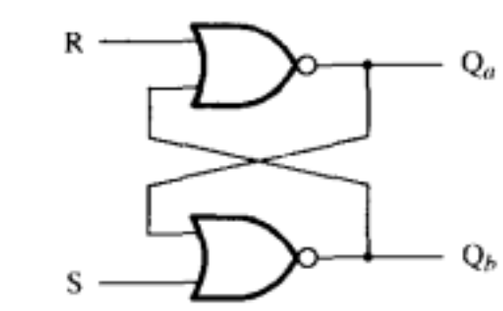


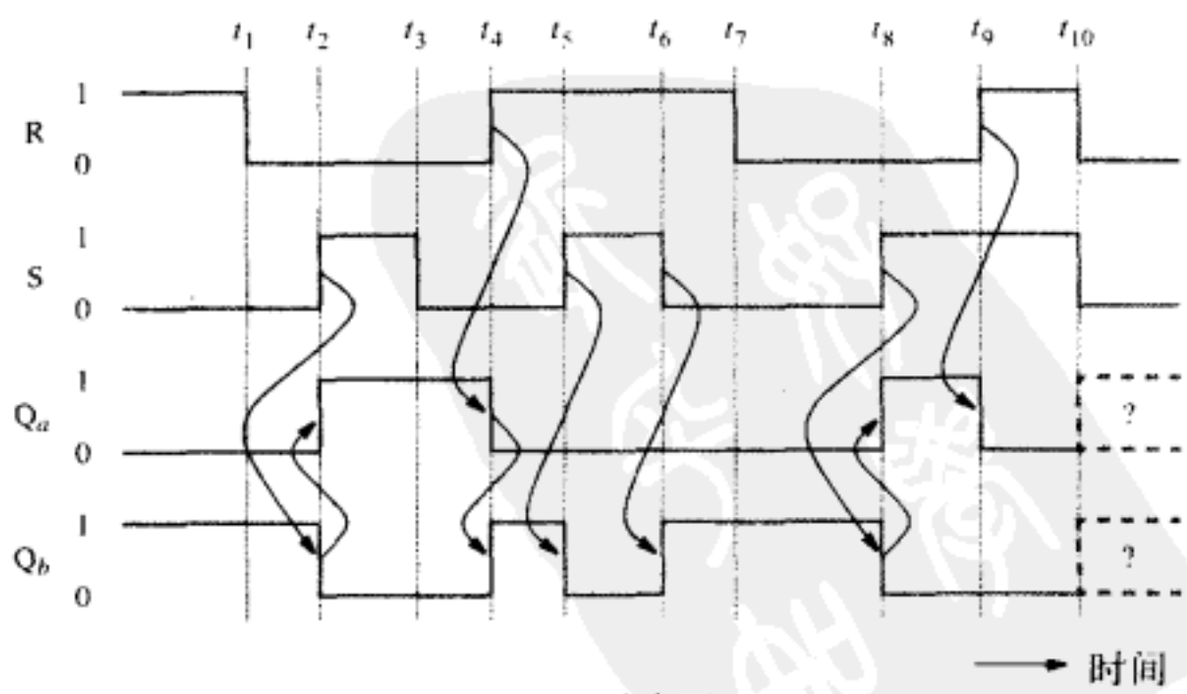
图7-4 用或非门组成的记忆电路



a) 电路

S	R	Q_a	Q_b	
0	0	0/1	1/0	（不改变）
0	1	0	1	
1	0	1	0	
1	1	0	0	

b) 特征表



c) 时序图

图7-5 一个由或非门构建的基本锁存器

图7-5c所示的是锁存器的时序图（其前提是假设通过或非门的传播延迟可以忽略）。当然，在实际电路中，由于门的延迟，波形的变化应该有延时。我们假设初始值$Q_a = 0$，$Q_b = 1$。锁存器的状态在时间t_2之前保持不变，在t_2时刻S变成1，从而Q_b变成0，于是使Q_a为1。这种因果关系在图中用箭头指示。当S在t_3时刻变为0时，状态并不改变，因为S和R此时都为0。在t_4时刻我们让$R = 1$，使Q_a变为0，从而Q_b变成1。在t_5时刻，S和R都是1，强制使Q_a和Q_b都等于0。t_6时刻，只要S变回0，Q_b就又会等于1。t_8时刻，我们有$S = 1$，$R = 0$，使$Q_b = 0$，$Q_a = 1$。很有趣的情况在t_{10}时刻出现。从t_9到t_{10}，由于$R = S = 1$，我们有$Q_a = Q_b = 0$。现在若R和S在t_{10}时刻变为0，Q_a和Q_b就都会变成1；但是若Q_a和Q_b都等于1，就会立刻强制$Q_a = Q_b = 0$。在$Q_a = Q_b = 0$和$Q_a = Q_b = 1$之间会存在振荡。若通过两个或非门的延时严格相同，振荡会无限地持续。在实际电路中，通过这些门的延时总是有些不同，锁存器的状态最终会停留在两个稳定状态中的某个状态，但是我们无法知道将会稳定在哪一个状态。这种不确定情况在波形图中用虚线标出。

上面关于振荡的讨论说明：虽然基本锁存器是非常简单的电路，但为了完全理解该电路的行为，必须仔细地分析。就一般情况而言，任何电路只要包含一条或者多条反馈回路，该电路的状态便与逻辑门的传递延时有关，设计这类电路必须十分小心。我们将在第9章详细讨论有关时序的问题。

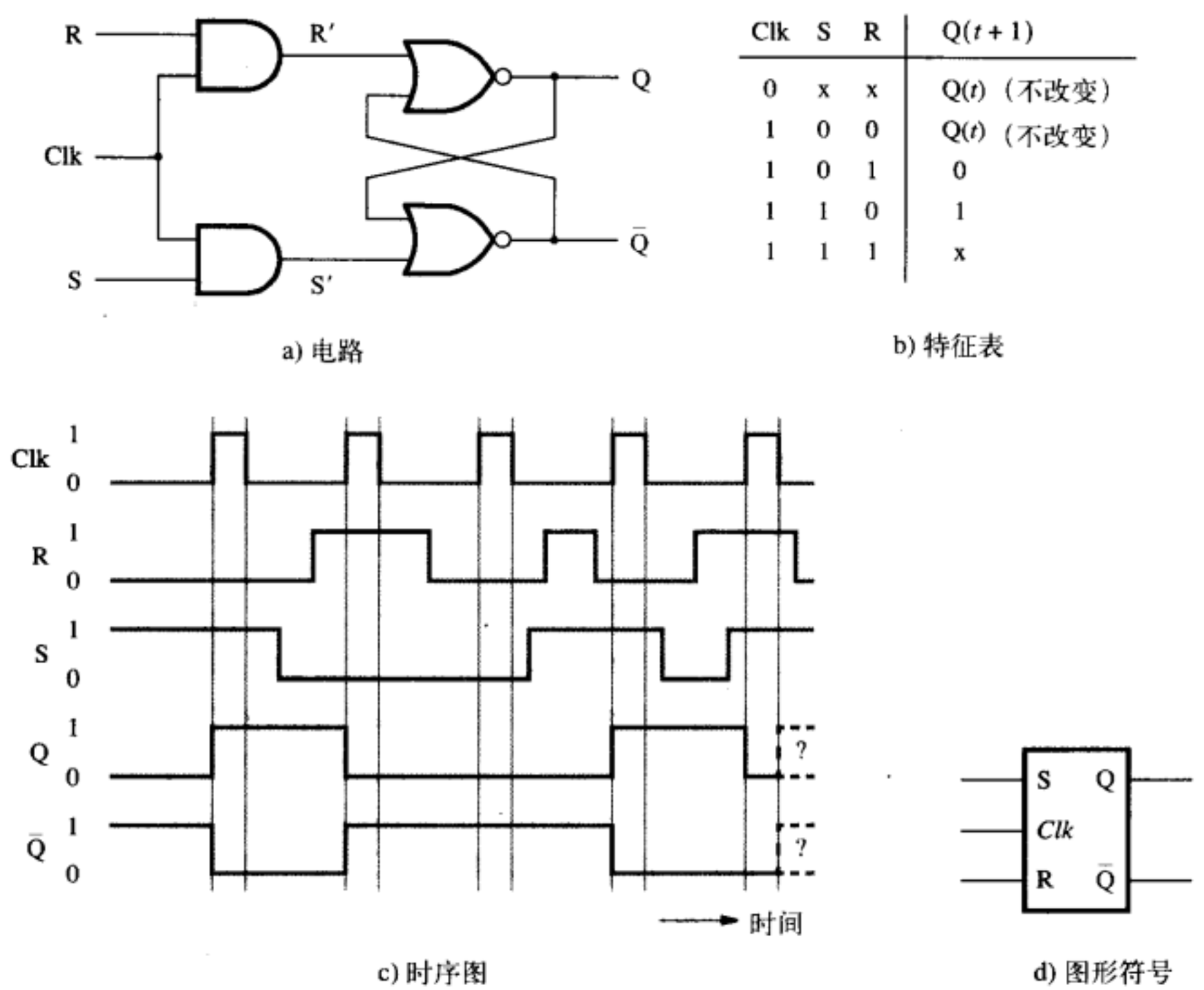
图7-5a所示的锁存器能实现图7-1中记忆电路所需的功能，将置位信号*Set*连接到输入端S，复位信号*Reset*连接到输入端R即可。输出端Q_a提供需要的On/ $\overline{\text{Off}}$ 信号。为了初始化报警器的操作，复位锁存器，因此报警器处于关断状态。当传感器产生逻辑值1，锁存器置位，Q_a等于1，于是报警器机制启动。若传感器输出又变回0，此时锁存器保持$Q_a = 1$的状态，因此警报保持启动状态。关闭警报的唯一途径是复位锁存器，让*Reset*输入端为1便可完成复位操作。

7.2 门控SR锁存器

在7.1节中，我们看到基本SR锁存器可以用作记忆元件。它能记住当S和R输入端都为0时的状态。SR锁存器的状态随着S和R输入端信号的改变而改变，其状态的改变总是发生在输入信号改变时。若我们不能控制其输入信号的改变时间，那么就无法知道锁存器什么时候会改变其输出状态。

在图7-1所示的报警系统中，若能够用一个控制输入信号*Enable*来启动或者停止报警器的工作，也许是我们想要的。于是，当处于使能模式时，报警系统就能实现上面描述的功能。而在禁止模式，即使将置位输入信号*Set*从0变为1，也不能启动报警器。图7-5a中的锁存器不能提供期望的操作。而对锁存器电路做一些修改，就能使它仅当*Enable* = 1时，才会对输入信号S和R的变化做出响应。而当*Enable* = 0时，则保持其输出状态不变。

图7-6a所示为修改后的电路，添加了两个与门提供期望的控制。当控制信号*Clk*等于0时，无论信号S和R的值是多少，输入锁存器的R'和S'的值都会是0。因此只要*Clk* = 0，锁存器就会保持它的当前状态。当时钟*Clk*变成1时，S'和R'信号将会分别同S和R信号相同。因此在这个模式中，锁存器的行为将像7.1节中描写的那样。请注意，我们用*Clk*而不用*Enable*这个名字称呼允许锁存器置位或者复位的控制信号。其原因是数字系统中经常用到这类电路，其记忆元件的状态变化只发生在定义好的时间间隔中，如同受到时钟的控制那样。定义这些时间间隔的控制信号通常称为时钟信号。*Clk*这个名字反映了该控制信号的本质。



Clk	S	R	Q(t+1)
0	x	x	Q(t)（不改变）
1	0	0	Q(t)（不改变）
1	0	1	0
1	1	0	1
1	1	1	x

a) 电路　　b) 特征表

c) 时序图　　d) 图形符号

图7-6 门控SR锁存器

这类使用控制信号的锁存器电路，称为*门控锁存器*。因为我们的电路有置位、复位的功能，所以称它为*门控SR锁存器*。图7-6b描述了它的行为，定义了在t+1时刻的Q输出的状态，即输入为S，R，Clk输出为$Q(t+1)$的函数。当$Clk = 0$时，无论R，S输入端的值是什么，锁存器将保持t时刻的状态$Q(t)$。这里用$S = x$，$R = x$表示，x表示信号值为0或者1（在第4章中也曾用过这种表示方式）。当$Clk = 1$时，电路的功能与图7-5所示的基本锁存器相同。若$S = 1$则置位，若$R = 1$，则复位。特征表的最后一行，$S = R = 1$时，状态$Q(t+1)$没有定义，因为我们不知道它究竟是0还是1。这与7.1节中图7-5所示的t_{10}时刻时序图描述的情况一致。此时，输入信号S和R均从1变成0，将引起我们讨论过的振荡情况。若$S = R = 1$，只要Clk从1变成0，这种情况立刻就会发生。为确保门控SR锁存器的操作有意义，当时钟从1变回0的时候，必须避免出现S和R输入信号都等于1的情况，这一点是至关重要的。

图7-6c所示是门控SR锁存器的时序图。图中显示的Clk信号是一个周期总等于1、很有规律的周期信号，用以表明Clk信号为实际系统中常用的时钟信号。图中展示了几种信号值组合产生的效果。可以看到，我们用Q表示其中一个输出，另一个输出用$\overline{Q}$表示，而不用图7-5所示的Q_a和Q_b。既然$S = R = 1$这种未定义模式在实际电路中必须避免，所以当锁存器处于正常操作状态时，其输出值必定互为反相。因此，我们通常将输出$Q = 1$时称为锁存器置位，$Q = 0$时称为复位。图7-6d所示为门控SR锁存器的图形符号。

用与非门实现的门控SR锁存器

目前为止，我们一直用交叉耦合的或非门来实现基本锁存器。我们也可用与非门来构造锁存器。用与非门实现的门控SR锁存器见图7-7所示。该电路的行为可以用图7-6b中的表来描

述。请注意：该锁存器电路中时钟是由与非门选通的，而不是由与门选通的。还请注意，与图7-6a所示电路比较，图7-7中的S和R输入信号是上下颠倒的。用与非门实现的电路比用与门的电路需要的晶体管少。我们更倾向于使用的是图7-7所示的电路，而不是图7-6a所示的电路。

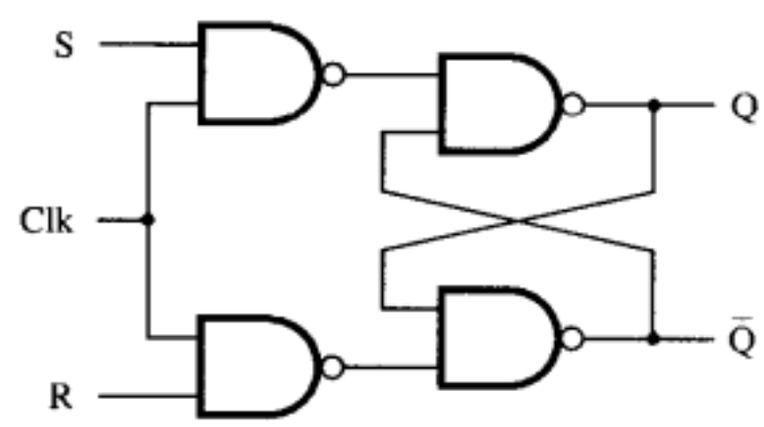


图7-7 用与非门实现的门控SR锁存器

7.3 门控D锁存器

在7.2节中，我们介绍了门控SR锁存器，并展示了如何将其作为记忆元件应用于图7-1所示的报警系统。该锁存器还有许多其他方面的应用。本节将描述在实践中更有用的另一种门控锁存器。它只有一个数据输入D，并且在时钟信号的控制下存储输入端的值。这种锁存器称为门控D锁存器。

为了说明引入门控D锁存器的必要，请回忆第5章（图5-13）中讨论过的加法器/减法器。当时我们描述电路如何进行加法运算，并没有讨论由加法器求得的结果(和位)将会如何处理。加法器/减法器单元通常用作计算机的运算部件。加法或者减法运算的结果通常用作下一步运算的操作数。因此，必须将加法器产生的结果储存起来，以便在需要时可以取用。我们也许会想用几个基本锁存器来记住这几个位，每一位用一个锁存器。在本章中，我们不说用锁存器记住一位的值，而说用锁存器存储该位的值或者简单地说“存储该位”，这样意思更清楚明了。我们应该把锁存器当作存储元件来考虑。

但是用基本锁存器是否能完成预期的操作呢？我们当然可以在加法运算开始前，将所有的锁存器复位。然后，把和(Sum)的每一位与锁存器的输入端S相连接。若和位是1，则锁存器被置位为1状态；否则锁存器保持0状态。若在加法运算开始时，和的每一位均为0，经过在加法器的传播延迟之后，某些位变成了1，求得期望的和，这种情况下加法器当然能正常工作。但是存在于加法器中的传播延迟给上述安排造成了极大的麻烦。假设我们用行波进位加法器，当输入数据X和Y与加法器连接时，因为进位信号在0和1之间变化，所以和信号可能会在电路中发生很多次变化。图5-21所示的波形图就显示了这种情况。我们碰到的麻烦是：试想和的某一位连接到锁存器的输入端S，则该位在0与1之间波动，最后稳定在0状态，可是锁存器一旦被置为1，就将保持下去（除非有*reset*信号作用），因此很有可能出现把错误的求和值保存起来的情况。

若不用基本锁存器，而改用门控SR锁存器，这样便可以解决由加法器和位变化所造成的存储问题。然后，我们可以在加法器产生正确的求和结果期间，安排时钟信号变为0。在加法器电路中，允许最大传播延迟之后，时钟电平应该变成1，以把求得的和存储在门控锁存器中。一旦存储完毕，时钟就可以变成0，以确保存储的值可以保留至下次时钟变成1时。为完成期望的操作，在将和位的值加载到这些锁存器中之前必须复位所有的锁存器。用这样的方法来处理该问题不免有些笨拙，我们更倾向于使用门控D锁存器。

图7-8a展示了门控D锁存器的电路。它是基于门控SR锁存器的但又有所不同，它的输入端S和R不是分开的，而只有一个数据输入端D。为了方便起见，图中标出了相当于S和R的输入端。若$D = 1$，则$S = 1$且$R = 0$，使锁存器处于状态$Q = 1$。若$D = 0$，则$S = 0$且$R = 1$，于是$Q = 0$。当然，状态变化只能发生在$Clk = 1$时。

应该注意到门控D锁存器不可能出现$S = R = 1$这个麻烦状态。这是非常重要的优点。门控D锁存器只在$Clk = 1$时输出Q才跟随着输入D值而变化，一旦Clk变成0，锁存器的状态便被冻

结，输出Q保持不变，直到时钟信号变成1时为止。因此，在时钟从1变为0时，门控D锁存器把输入信号D存储起来。图7-8还示出了门控D锁存器的特征表、图形符号和时序图。

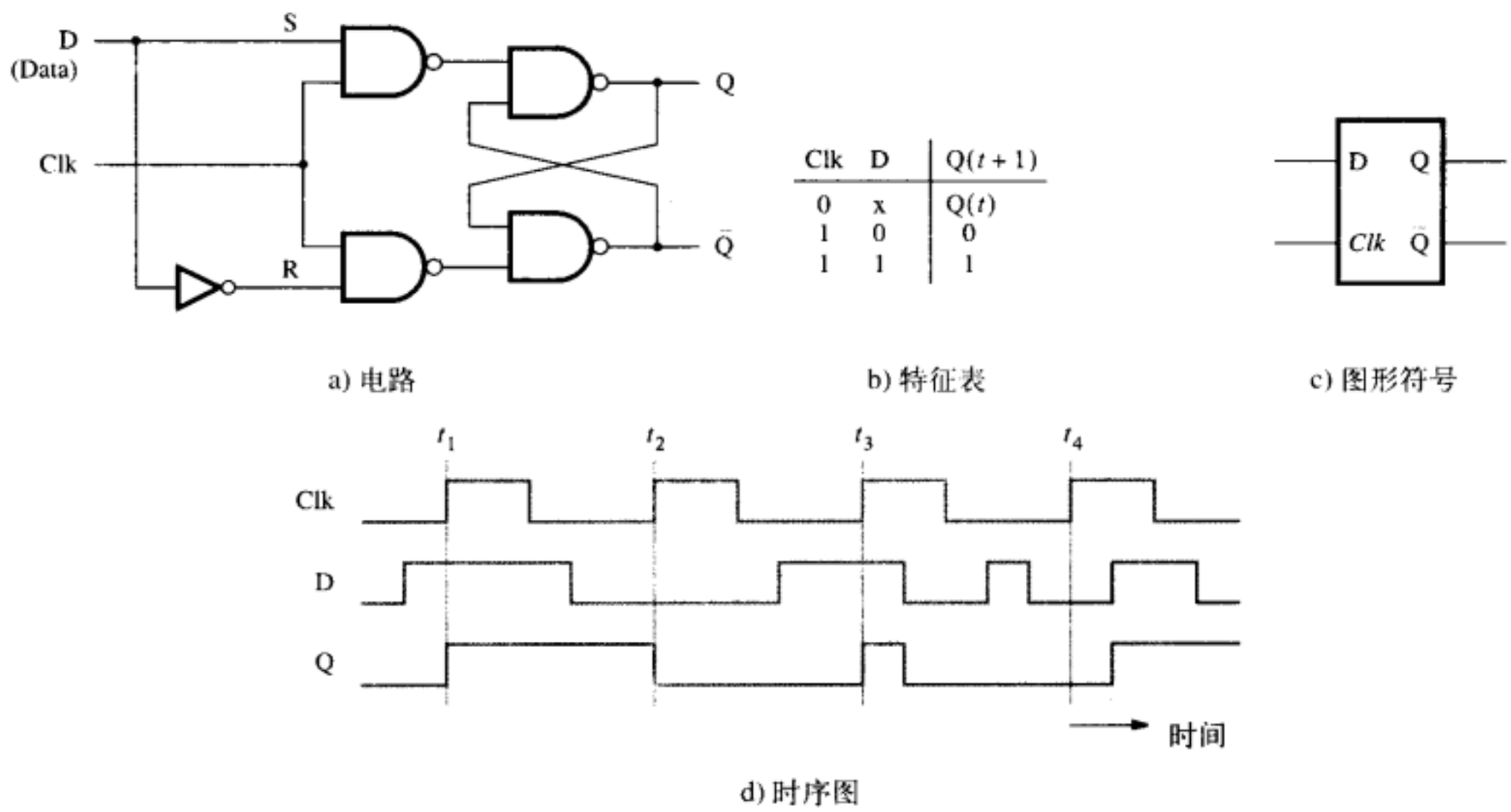


图7-8 门控D锁存器

时序图说明在$Clk = 1$时，若D信号变化，输出Q会如何变化。在t_3时刻，第3个脉冲到达，因为$D = 1$，所以输出Q也随之变成1。Q的这个值在Clk为0时存储。但是，当时钟信号仍处于高电平的中途，将D变成0，使得Q也跟着变成0。当$Clk = 0$时，Q的这个值便会被保存起来。现在，在t_4时刻到来之前，时钟信号保持为0，锁存器的状态此刻也就不会发生进一步的变化。需要注意的关键点是：只要$Clk = 1$，输出Q就跟随输入的D而变化，此刻它们的波形是一致的。但当时钟电平为0时，输出Q就不能发生任何变化。在第3章中，我们曾介绍过逻辑值由高电平和低电平体现。由于门控D锁存器的输出受到时钟输入电平高低的控制，因此这种锁存器称为*电平敏感型锁存器*。从图7-6到图7-8所示的电路，全都属于电平敏感型电路。我们将在7.4节介绍另一种存储元件，其输出仅在时钟信号电平发生变化的那一时刻才可能发生改变，这种元件称为*沿触发存储元件*。

现在让我们重新考察图7-3所示电路。仔细分析电路后发现，它的行为特点与图7-8a所示的电路完全一致。图7-3中的输入$Data$和$Load$分别与图7-8a中的输入D和Clk对应。图7-3中的输出$Output$与图7-8a中的输出Q对应。而图7-3中的点B与图7-8a中的$\overline{Q}$对应。因此，图7-3所示的电路也是门控D锁存器。与图7-8a所示的电路相比，该电路的优点是所用的晶体管较少。

传播延迟的影响

在前面的讨论中，我们忽略了传播延迟的影响。而在实际电路中必须考虑这些延迟。以图7-8a所示的门控D锁存器为例，当时钟信号从1变成0的时刻，它将当前输入D的值存储在锁存器中。若此刻D信号是稳定的，则存储的数据是可靠的。但若此时信号D仍在变化中，则不能预料存储的究竟是什么。因此，产生信号D的逻辑电路设计者必须保证在时钟信号发生变化的关键时刻，信号D是稳定的。

图7-9所示为关键时间区间。在*Clk*信号下降沿到来之前信号*D*必须稳定的最短时间叫做锁存器的建立时间t_{su}。在时钟信号的下降沿之后信号*D*必须保持稳定的最短时间叫做锁存器的保持时间t_h。t_{su}和t_h的值依赖于芯片的制造技术。集成电路芯片厂家在他们芯片的数据手册上提供这个数据。现代CMOS制造工艺的典型值是：$t_{su} = 0.3$ns，$t_h = 0.2$ns。我们将在7.13节举一些例子说明建立时间和保持时间是如何影响电路操作速度的。而将建立时间和保持时间不能满足要求时的存储元件行为，放在10.3.3节中讨论。

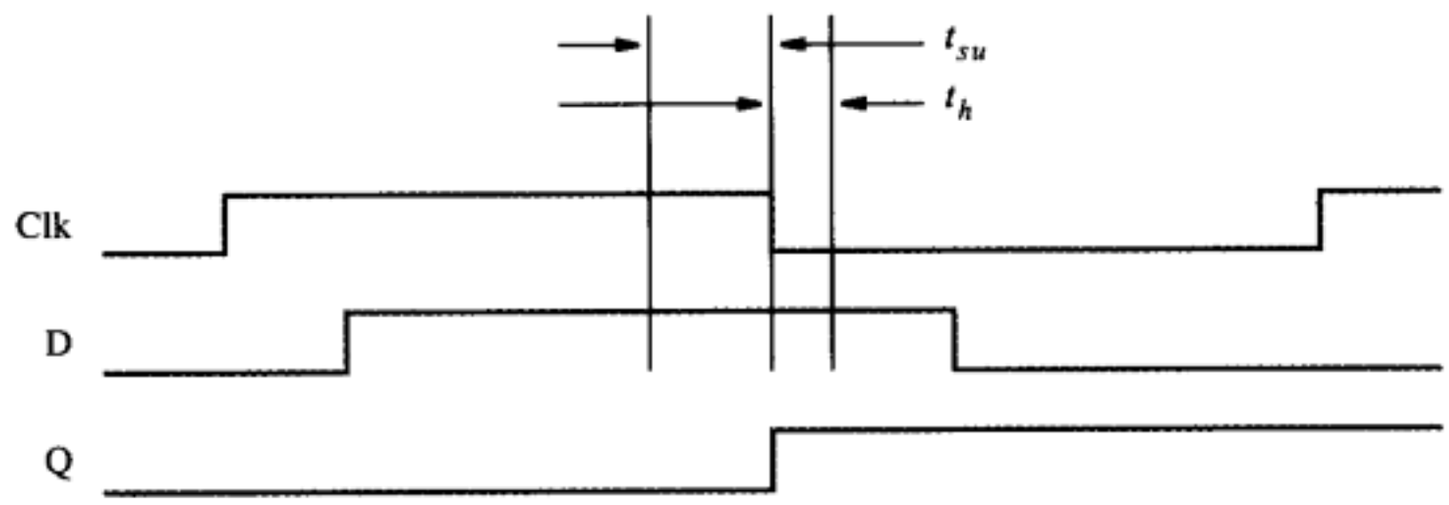


图7-9 建立时间和保持时间

7.4 主从D触发器和沿触发的D触发器

在电平敏感型锁存器中，在时钟信号有效（在我们的例子中，有效是指时钟电平为1）期间，锁存器的状态跟随输入的变化而变化。在7.8节和7.9节中，我们将会看到需要一种在一个时钟周期内其状态最多改变一次的存储元件。本节将讨论具有这种行为特性的两种电路。

7.4.1 主从D触发器

考虑图7-10a所示的电路，它由两个门控D锁存器组成。前者称为主锁存器，当*Clock* = 1时，改变状态。后者称为从锁存器，当*Clock* = 0时改变状态。此电路的运行情况描述如下：当时钟为高电平的时候，主锁存器的状态跟随输入信号*D*的值，而从锁存器的状态保持不变。换言之，*Clock* = 1期间，Q_m的值随*D*的变化而变化，而Q_s的值则保持不变。当时钟信号变为0后，主锁存器的状态不再随着输入信号*D*的变化而变化。与此同时，从锁存器的输出则跟随信号Q_m的变化改变状态。因为Q_m在*Clock* = 0时不发生变化，因此从锁存器在一个周期内最多发生一次状态的变化。从外部观察者的角度来看，电路连接到从锁存器的输出端，主从电路在时钟的负跳变沿（时钟由1变成0时刻）改变其状态。在一个周期内，连接到主锁存器的输入信号*D*可能发生很多次变化（译者注：当*Clock* = 1时，Q_m也跟随着变化多次，而当*Clock*变为0后，Q_m不再变化，此时对第2个锁存器而言，时钟电平为1），而在信号Q_s处观察到的只能是时钟负跳变沿时刻的Q_m。换言之，输出信号Q_s是在时钟负跳变沿时刻采集到的输入信号*D*的瞬时值，它是一个稳定的值。

图7-10所示的电路称为主从D触发器。触发器这个术语用来表示在时钟的跳变沿时刻改变状态的存储元件。图7-10b所示为触发器的时序图，图7-10c所示为图形符号。在触发器的图形符号中，我们用 > 以表示该触发器是由时钟沿触发的，在时钟输入端画一个圈表示该触发器是由负跳变沿触发的。

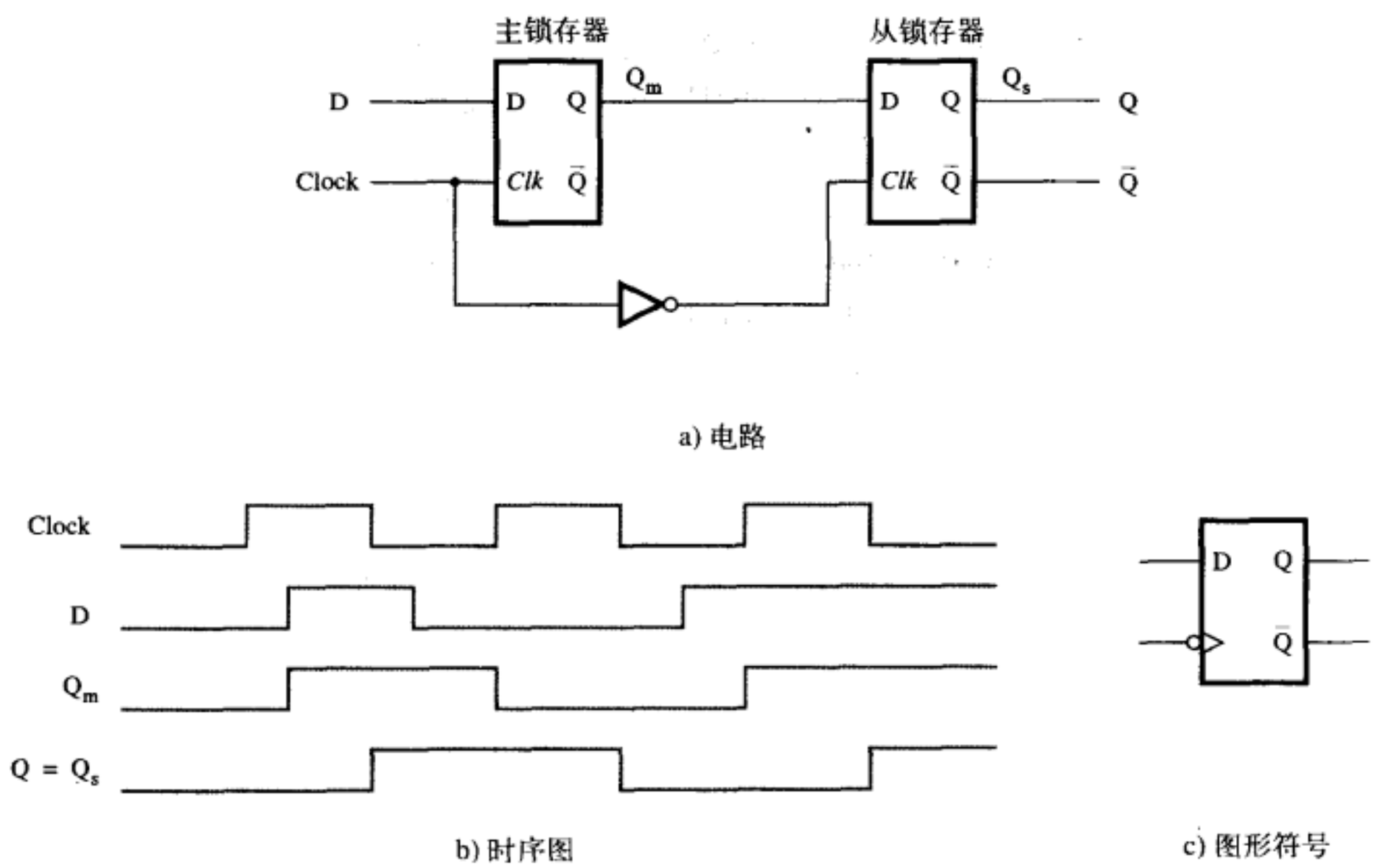


图7-10 主从D触发器

7.4.2 沿触发的D触发器

图7-10b中的主从D触发器的输出信号是由时钟信号的负跳变沿触发的。把时钟直接与从锁存器连接，时钟信号反相后再与主锁存器连接，该电路就可以改成由时钟的正跳变沿触发。而图7-11a所示为另一个不同的电路，该电路也可以完成与主从触发器相同的任务。

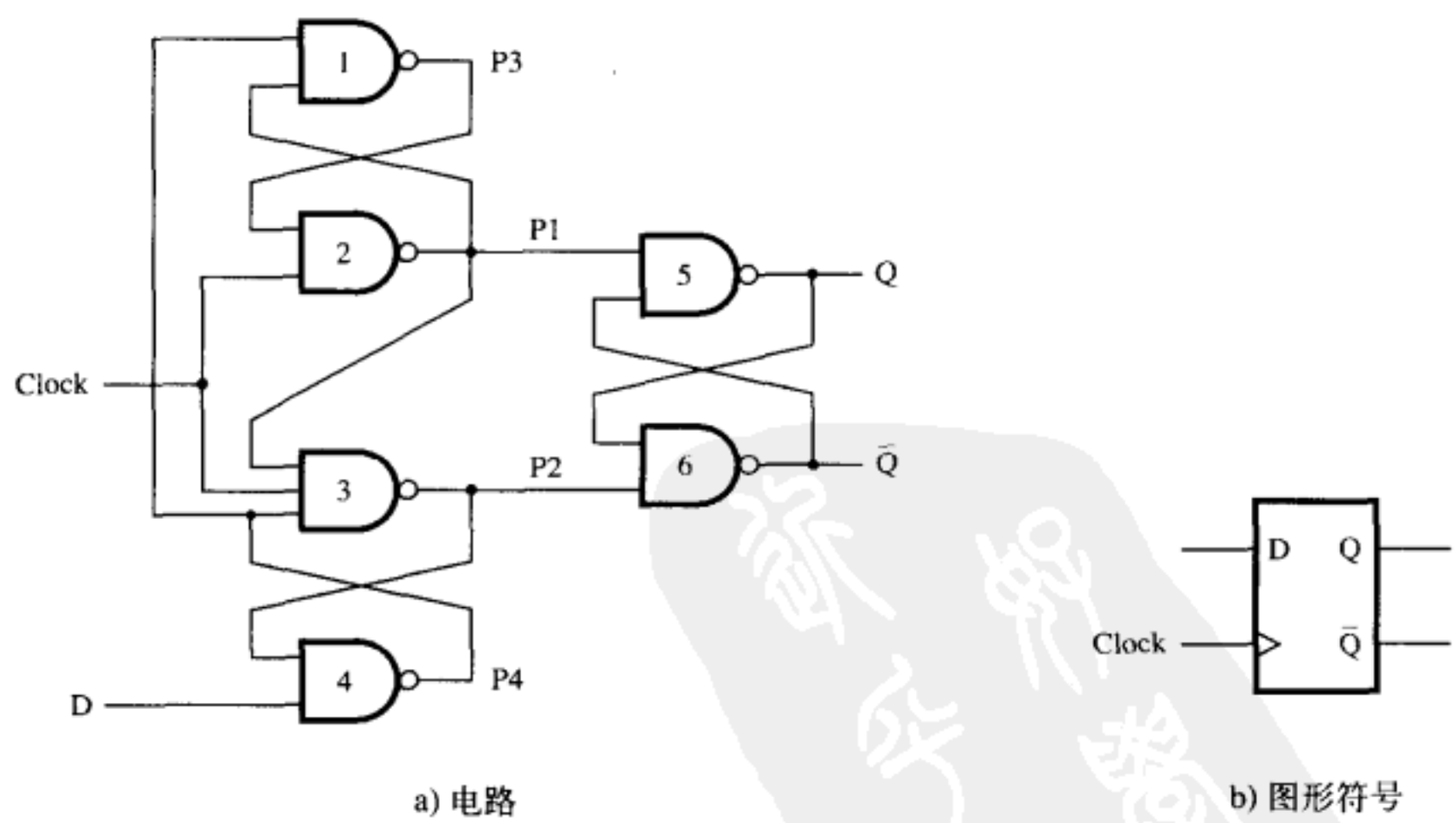


图7-11 一个上升沿触发的D触发器

该电路只需要6个与非门，因此用的晶体管较少。电路的操作如下：当$Clock = 0$时，门2和门3的输出为高电平，因此$P1 = P2 = 1$，使由门5和门6组成的输出锁存器保持当前状态。同时，信号$P3$等于D的值，$P4$等于$\overline{D}$。当$Clock$变成1，就会发生如下变化：$P3$和$P4$的值通过

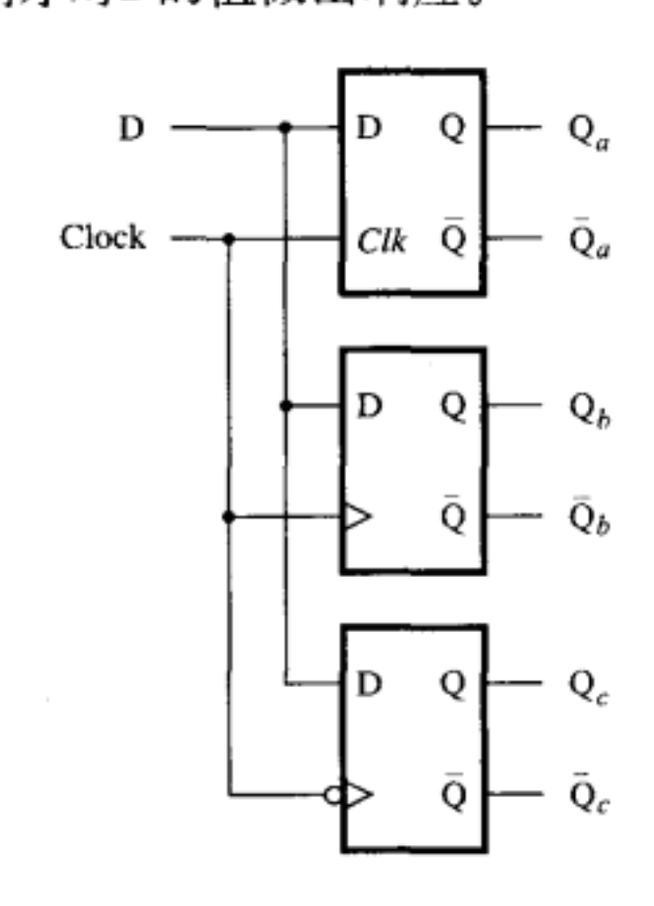
门2和门3传输使$P1 = \overline{D}$，$P2 = D$。为了使电路可靠地运行，在时钟由0变成1时，信号$P3$和$P4$必须保持稳定。因此触发器的建立时间等于D输入通过门4和门1到$P3$点的延迟时间。触发器的保持时间等于门3的延迟时间，因为一旦$P2$的值稳定后，D端再发生变化就没关系了。

为了保证电路可靠地运行，有必要说明，在*Clock*变到1之后，只要Clock等于1，D的任何后续变化都不会对锁存器的输出产生影响。我们必须考虑两种情况。第一种情况，假设在时钟正跳变沿时刻$D = 0$，则$P2 = 0$，因而在$Clock = 1$期间，门4的输出保持为1（不管输入D后续如何变化）。第二种情况是，若时钟正跳变沿时$D = 1$，则$P1 = 0$，使门1和门3的输出都等于1，而不管输入D是什么。因此，在$Clock = 1$期间，触发器将不理会输入D的变化。

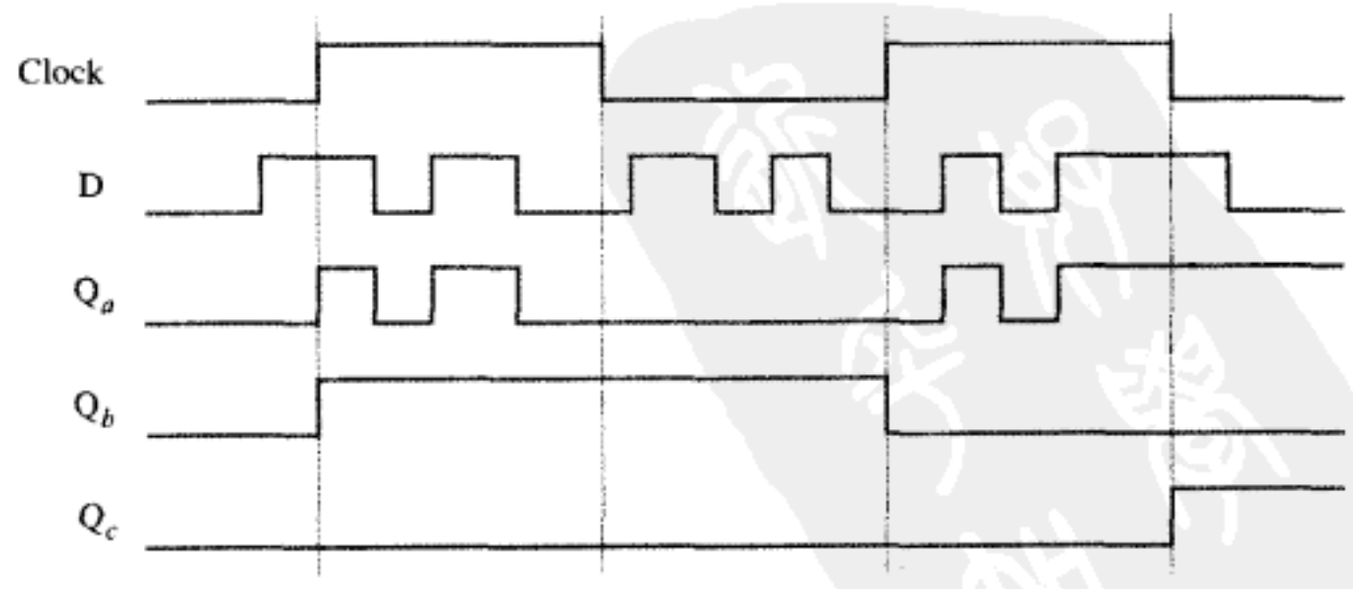
图7-11b给出了这种触发器的图形符号。图中的时钟输入表明该触发器是由时钟的正跳变沿触发的。用或非门构成的类似电路，可以用作负跳变沿触发的触发器。

电平敏感存储元件与跳变沿触发的存储元件之间的比较

图7-12画出了在相同数据和时钟输入的前提下，三种不同存储元件的输出波形。第一个元件是门控D锁存器，是电平敏感型的。第二个是正跳变沿触发的D触发器，第三个是负跳变沿触发的D触发器。为突出这些存储元件的不同之处，输入信号D在时钟的半个周期里变化多次。请注意观察，在时钟为高电平期间，门控D锁存器的输出就会跟随着输入信号D的变化而变化。而正跳变沿触发器的输出只在时钟从0变到1时刻才对D的值做出响应；负跳变沿触发器的输出只在时钟从1变到0时刻才对D的值做出响应。



a) 电路



b) 时序图

图7-12 电平敏感型和跳变沿触发型D存储元件的比较

7.4.3 有清零端和预置信号的D触发器

触发器经常被用于有许多可能状态的电路中，电路的响应不仅取决于当前的输入，也取决于那时的电路状态。我们将在第8章中讨论触发器电路的一般形式。计数器电路就是触发器应用的一个简单例子。计数器用来记录（也许在一段时间内的）某事件发生的次数。我们将在7.9节详细讨论计数器。一个计数器由几个触发器组成，它的输出被解释为一个数字。计数器电路可以递增计数，也可以递减计数。使计数器处于已知的初始状态（初始计数）也是一个很重要的功能。显然，必须有可能使计数器清零，这意味着所有触发器的输出必须为0，即$Q = 0$；而且必须有可能预先将每个触发器设置为1（即$Q = 1$），以便为计数器设置特定的初始值。上述这些特性可以合入图7-10和图7-11所示的电路，具体做法见下面所述。

图7-13a所示为用与非门实现的图7-10a电路。主级（锁存器）只是图7-8a所示的门控D锁存器。从级（锁存器）则用较简单的门控SR锁存器（见图7-7），并没有用与主级完全相同的锁存器。这样做为电路减少了一个非门。

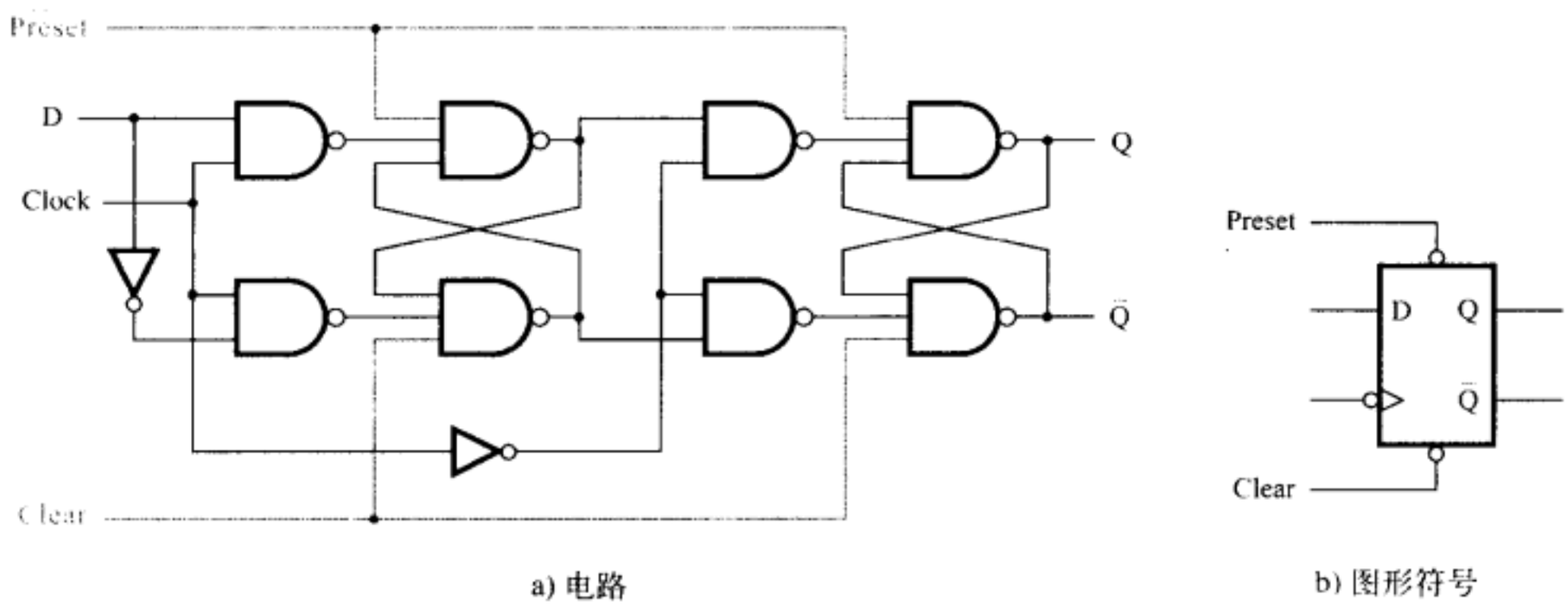


图7-13 带清零和预置数的主从D触发器

提供清零和预置数（preset）功能的简单方法是在交叉耦合锁存器的每个与非门的输入端添加一个输入信号，如图中灰色线所示。若输入$Clear = 0$，则强迫触发器进入状态$Q = 0$；若$Clear = 1$，则该输入对于与非门来说没有任何影响。同理，若$Preset = 0$，则强制触发器进入状态$Q = 1$，而若$Preset = 1$，则没有任何作用。为了表明输入信号*Clear*和*Preset*是低电平有效的，我们在图中的名字上加一横线。应该注意，用这种触发器的电路不能使*Clear*和*Preset*同时为0。图7-13b所示为这种触发器的图形符号。

可以在图7-11a所示的跳变沿触发器中做相似的修改，见图7-14a所示。同样，输入信号*Clear*和*Preset*都是低电平有效。当它们等于1的时候对触发器不产生任何影响。

在图7-13a和图7-14a的电路中，输入信号*Clear*和*Preset*变为低电平时都能立即产生效果。例如，若$Clear = 0$，则触发器会立刻进入状态$Q = 0$，不管此时刻时钟信号的值是什么。若信号*Clear*不用考虑时钟信号，就能将触发器清零，则我们说该触发器有*异步清零端*。在实际工作中，我们更愿意采用在时钟有效沿情况下的复位。*同步清零*可以用图7-14c所示的电路实现。若输入信号$\overline{Clear}$等于1，则触发器正常操作。但是，若$\overline{Clear}$变成0，则在下一个时钟的正跳变沿触发器会被清零。我们将在7.10节中更详细地讨论触发器的清零问题。

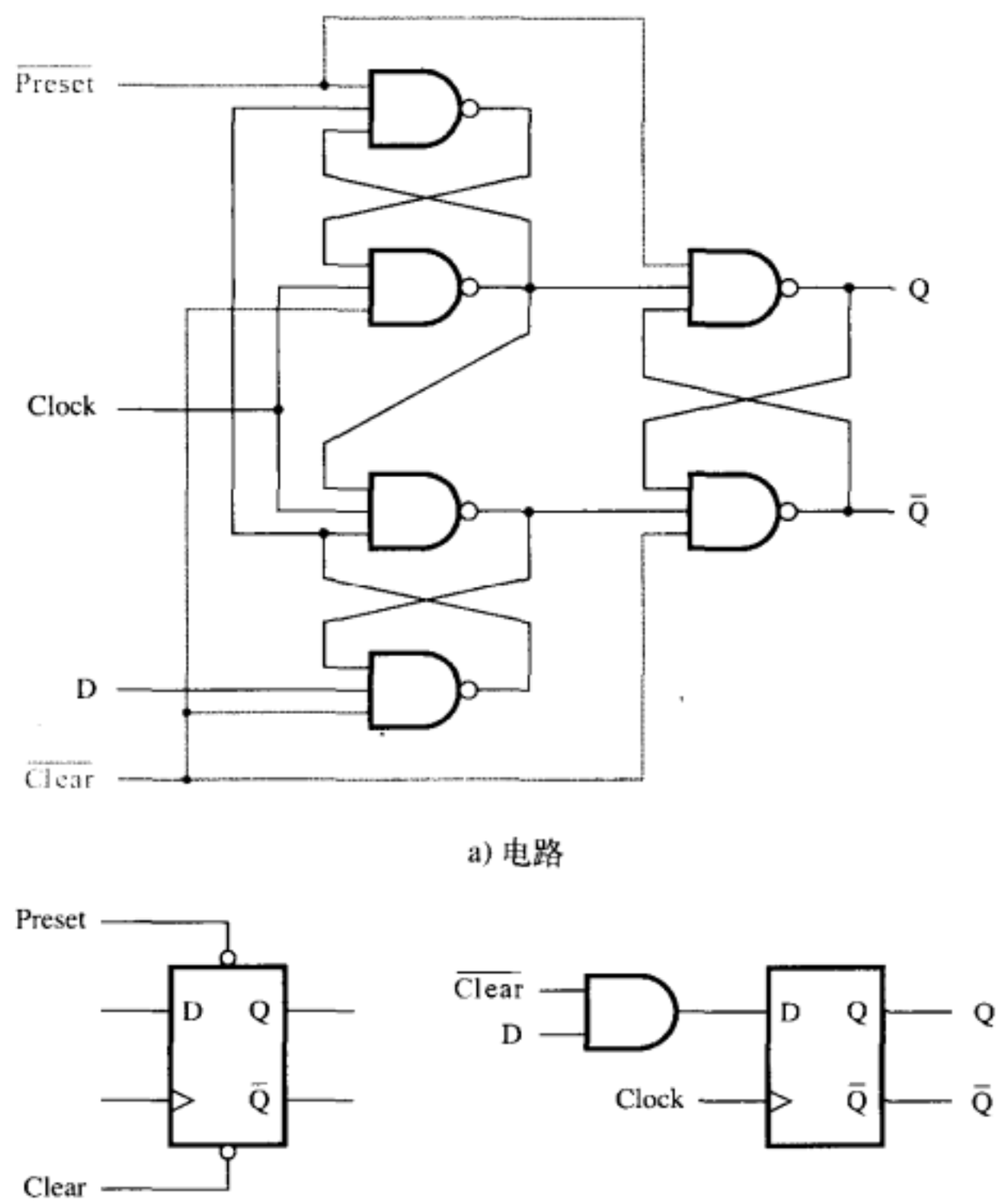
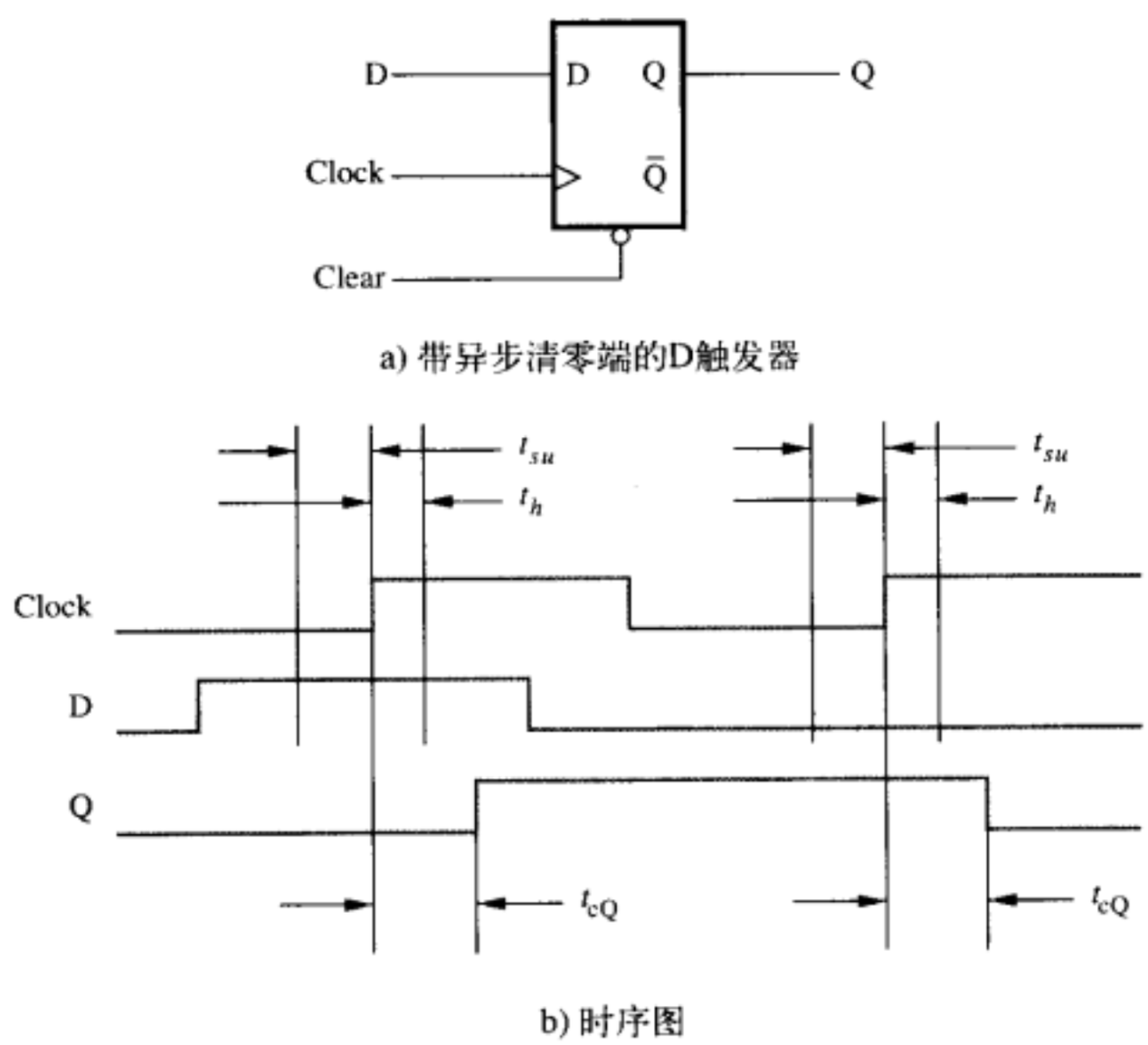


图7-14 带清零和预置信号的正跳变沿触发的D触发器

7.4.4 触发器的时序参数

在7.3.1节中，我们曾经讨论过与锁存器有关的问题。在实际工作中，这些问题与触发器电路问题同样重要。图7-15a展示了带异步清零端、由正沿触发的触发器，图7-15b则说明该触发器的某些重要的时序参数。在时钟的正跳变沿时刻，数据被加载进入触发器的输入端D。该逻辑值必须在时钟跳变沿发生前，在建立时间t_{su}期间稳定。而数据必须在时钟沿到达后的保持时间t_h期间保持稳定。如果建立时间或者保持时间不能严格地符合电路的要求，则该触发器就有可能进入不稳定状态，我们通常称这种状态为亚稳态（metastability）。我们将在10.3节中讨论这个概念。

正如图7-15所表明的那样，从时钟的正跳变沿起，到Q的值根据D的值发生相应变化所需要的时间为时钟到Q的传播延迟t_{cQ}。一般情况下，Q从1变化到0和从0变化到1所需要的延迟时间并不完全一致，但为了简单起见，我们假设这两个时间是相等的。对于商业芯片中的触发器而言，通常需要用两个值定义t_{cQ}，分别表示在实际运行中的最大延迟和最小延迟。由于芯片制造过程中各种因素的影响，这些延迟可在一定的范围内分布，芯片说明书中标明的延迟指标通常指的就是这样一个范围。在7.15节中我们提供了几个例子说明触发器的时序参数对电路运行的影响。



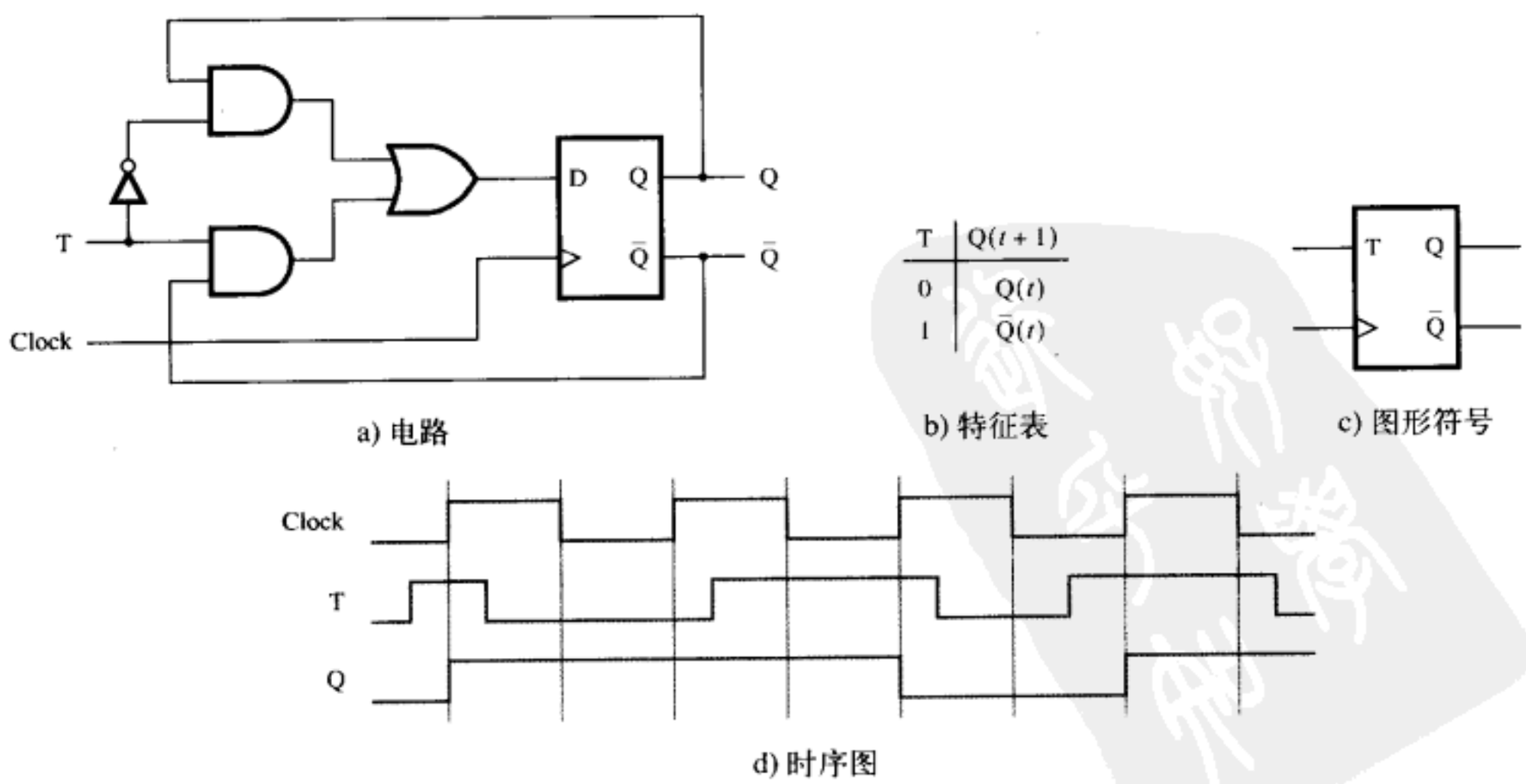
a) 带异步清零端的D触发器

b) 时序图

图7-15 触发器的时序参数

7.5 T触发器

D触发器是一个通用的存储元件，有许多种用途。在D触发器输入端添加一些简单逻辑电路，它就可以变成另一种类型的存储元件。图7-16a所示的电路做了很有意思的修改。该电路用一个正跳变沿触发的D触发器实现。在信号T的控制下，使D触发器的数据输入等于Q或者等于$\overline{Q}$。在时钟的每一个上升沿，触发器都有可能改变其状态$Q(t)$。若$T=0$，则$D=Q$，状态保持不变，也就是说$Q(t+1)=Q(t)$。但是若$T=1$，则$D=\overline{Q}$，那么新的状态$Q(t+1)=\overline{Q}(t)$。因此，当正跳变沿到来时，该电路的操作是若$T=0$，则该电路保持它的当前状态，而若$T=1$，则该电路状态翻转。



a) 电路

b) 特征表

c) 图形符号

d) 时序图

图7-16 T 触发器

图7-16b用特征表的形式说明了电路的操作。任何可以实现该特征表的电路都叫做T触发器。T触发器这个名字来自于它的行为，$T = 1$时触发器的状态“翻转”（英文为：toggle）。我们将在7.9节看到，翻转的特点使得T触发器成为构建计数器电路的一个有用元件。

可配置的触发器

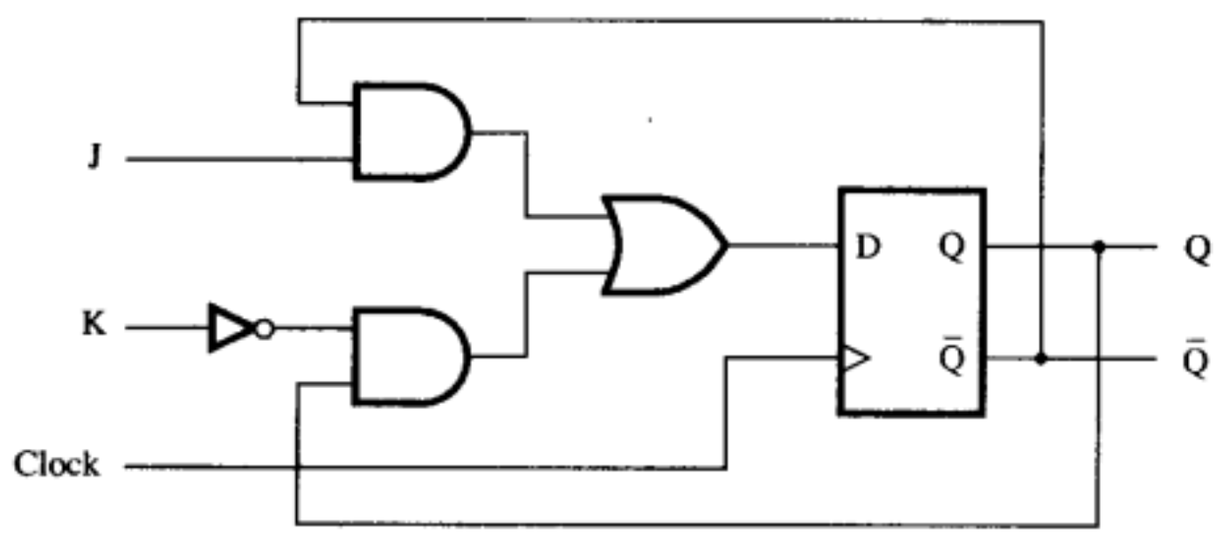
在实现电路时，有时用某种类型的触发器可能会比用其他类型的触发器更节省资源也更方便。通用芯片内的触发器往往都是可配置的，例如PLD这类芯片就是如此， 这意味着，芯片内的触发器电路可以配置为D触发器、T触发器或者其他类型的触发器。例如，在某些PLD中，触发器可以配置成D触发器或者T触发器（见习题7.6和7.8）。

7.6 JK触发器

根据图7-16a所示的电路，我们可以推导出另一种有趣的电路。该电路不同于T触发器只有一个T输入端，它有两个输入端J和K，见图7-17a所示。该电路的输入端D定义为:

$$D = J\overline{Q} + \overline{K}Q$$

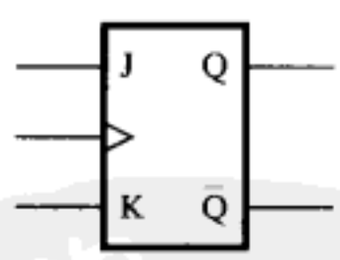
图7-17b给出了对应的特征表。该电路被称为JK触发器。它将SR触发器和T触发器的行为以一种有用的方式结合起来。对所有的输入，除了$J = K = 1$以外，若令$J = S$，$K = R$，其行为同SR触发器一样。对于$J = K = 1$的情况，SR触发器必须避免，此时JK触发器将其状态翻转，其功能与T触发器相同。



a) 电路

J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q̄(t)

b) 特征表



c) 图形符号

图7-17 JK 触发器

JK触发器是一种很灵活的电路。它可以像D触发器和SR触发器一样直接用于存储的目的。但是它也可以用作T触发器，只要将J和K输入端连接在一起就可以了。

7.7 术语小结

我们所用的技术术语都是很普通的，但是读者应该理解技术文献中有关锁存器和触发器术语的不同解释。本书中的术语总结如下：

基本锁存器是两个或非门或者与非门的反馈连接组成的电路，该电路可以存储一位的信息。用S输入端可将该电路置为1，用R输入端可将该电路复位为0。

门控锁存器是包括输入门和控制输入信号的基本锁存器。当控制信号为0时，该锁存器保持它已存在的状态，当控制信号为1时其状态可以改变。在我们的讨论中，将控制输入规定为时钟。我们考虑了两种类型的门控锁存器：

- **门控SR锁存器**用输入S和R分别使锁存器置为1或复位为0。
- **门控D锁存器**用输入D强迫锁存器进入与输入D相同的逻辑值。

触发器是基于门控锁存器原理的存储元件，它的输出状态的改变只能发生在控制时钟信号的跳变沿。我们介绍过两种类型的触发器：

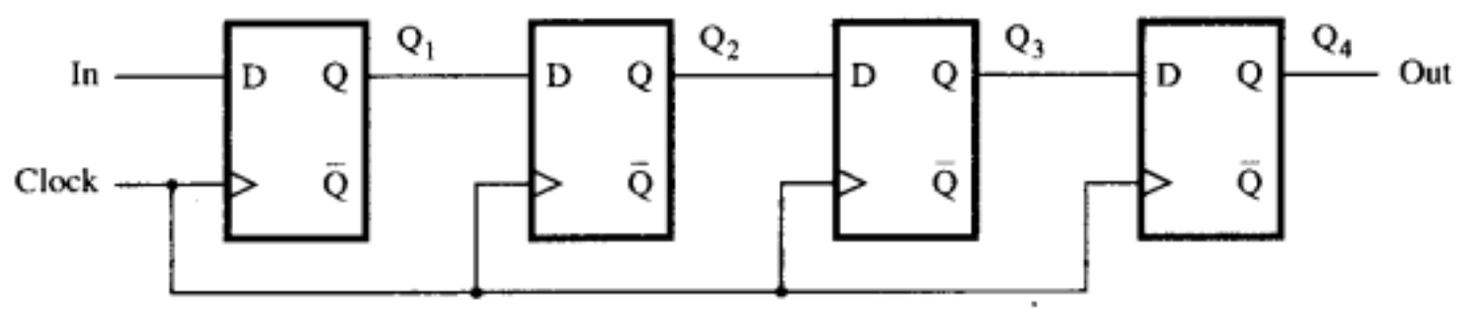
- **沿触发的触发器**只有在时钟信号的有效跳变沿时刻，才能接受当前的输入值。
- **主从触发器**由两个门控锁存器构成。主锁存器在时钟周期的前半个周期有效，从锁存器在另外半个周期有效。触发器的输出值在时钟跳变沿时刻改变，该时钟跳变沿激活了进入从级锁存器的传送。主从触发器可以是沿触发的，也可以是电平敏感的。若主锁存器是门控D锁存器，则其行为与沿触发的触发器一致。若主锁存器是门控SR锁存器，则该触发器就是电平敏感的（见练习题7.19）。

7.8 寄存器

一个触发器可以存储一位信息。由n个触发器组成的电路可以用来存储n位信息，例如n位的数，我们把这n个触发器称为一个*寄存器*。寄存器中每个触发器共用同一个时钟，每个触发器都按前面章节中所描述的那样工作。寄存器这个术语仅仅是指由n个触发器组成的结构。

7.8.1 移位寄存器

在5.6节中，我们解释了对给定的数乘2的操作，即左移一位，末位补零。类似地，右移一位可以实现除2操作。有移位功能的寄存器称为*移位寄存器*。



a) 电路

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

b) 序列举例

图7-18 一个简单的移位寄存器

图7-18a给出了一个四位的移位寄存器，可以使数据右移一位。数据位用连续的方式载入移位寄存器的In输入端。每个触发器的内容在时钟正跳变沿时刻传递给下一个。图7-18b说明了传递过程，即当8个连续时钟周期里In的值分别为10111000时各触发器的状态是如何变化的，假设所有触发器的初始状态为0。

实现移位寄存器必须用沿触发的触发器或者主从触发器。电平敏感型的门控锁存器并不适用，因为在时钟信号等于1的期间内，In值的变化可能会传递到多个锁存器。

7.8.2 并行存取的移位寄存器

在计算机系统中，经常有必要传输n位数据。这可以用n条分开的电线一次实现所有位的传输，我们称这种形式的传输为*并行传输*。也可以只用一条线，一次传一位，经过n个连续的时钟周期实现所有位的传输。这种形式的传输称为*串行传输*。为了串行传输一个n位的数据，可以将全部n位的数据在一个时钟周期里并行加载到一个n位的移位寄存器中。在接下来n个时钟周期，将寄存器的内容逐次移位，实现串行传输。我们经常也需要做相反的操作，即数据位是串行接收的，经n个时钟周期后，移位寄存器的内容便可以作为一个n位的数据并行存取。

图7-19展示了一个允许并行存取的四位移位寄存器。与一般移位寄存器的连接不同，每个触发器的输入D连接到两个不同的信号源。一个信号源是前级触发器，用于移位寄存器的操作。另一个信号源是与被加载的触发器逐位对应的外部输入信号，作为并行操作的一部分。控制信号 $\overline{\text{Shift}}$/ Load用来选择操作的模式。若 $\overline{\text{Shift}}$/ Load = 0，则电路为移位寄存器工作模式；若 $\overline{\text{Shift}}$/ Load = 1，则并行输入数据便加载到寄存器中。这两种操作都发生在时钟的正跳变沿时刻。

在图7-19中，我们之所以选择用Q_3，…，Q_0来标记触发器是因为移位寄存器经常被用来存储二进制信息。通过观察所有触发器的输出，便可以并行读取寄存器的内容。触发器也可以通过串行的方式存取数据，当寄存器的内容在连续时钟周期下逐位移位时观测Q_0的值即可。这种可以串行加载和并行读取数据的电路叫做*串－并转换器*。而功能相反的电路是*并－串转换器*。图7-19所示的电路可以实现这两种功能。

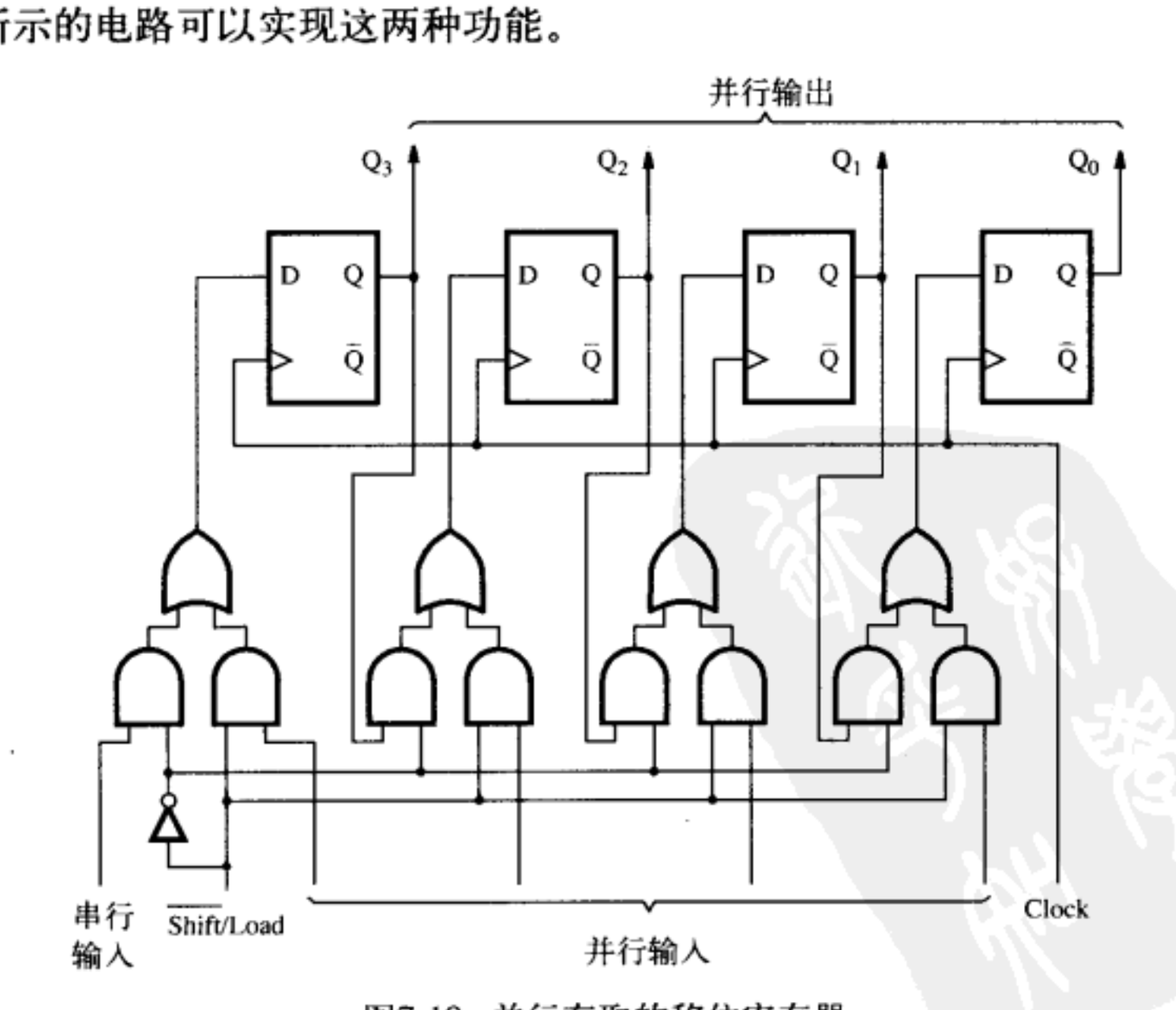


图7-19 并行存取的移位寄存器

7.9 计数器

在第5章中，我们阐述了进行算术运算的电路。我们展示了如何设计加法/减法器电路，可以用一个简单的逐位进位的级联结构，它造价低，但是运行速度慢；也可以用一个较复杂的超前进位结构，它造价高，但运行速度较快。本节考察一种特殊的加法和减法操作，这种操作专门用于计数。我们特别想设计的是可以加1或减1的电路。在数字系统中，计数器电路的应用非常广泛。计数器可以用来记录特定事件的发生次数、产生控制系统中不同任务的时间间隔、记录特定事件之间的时间间隔等。

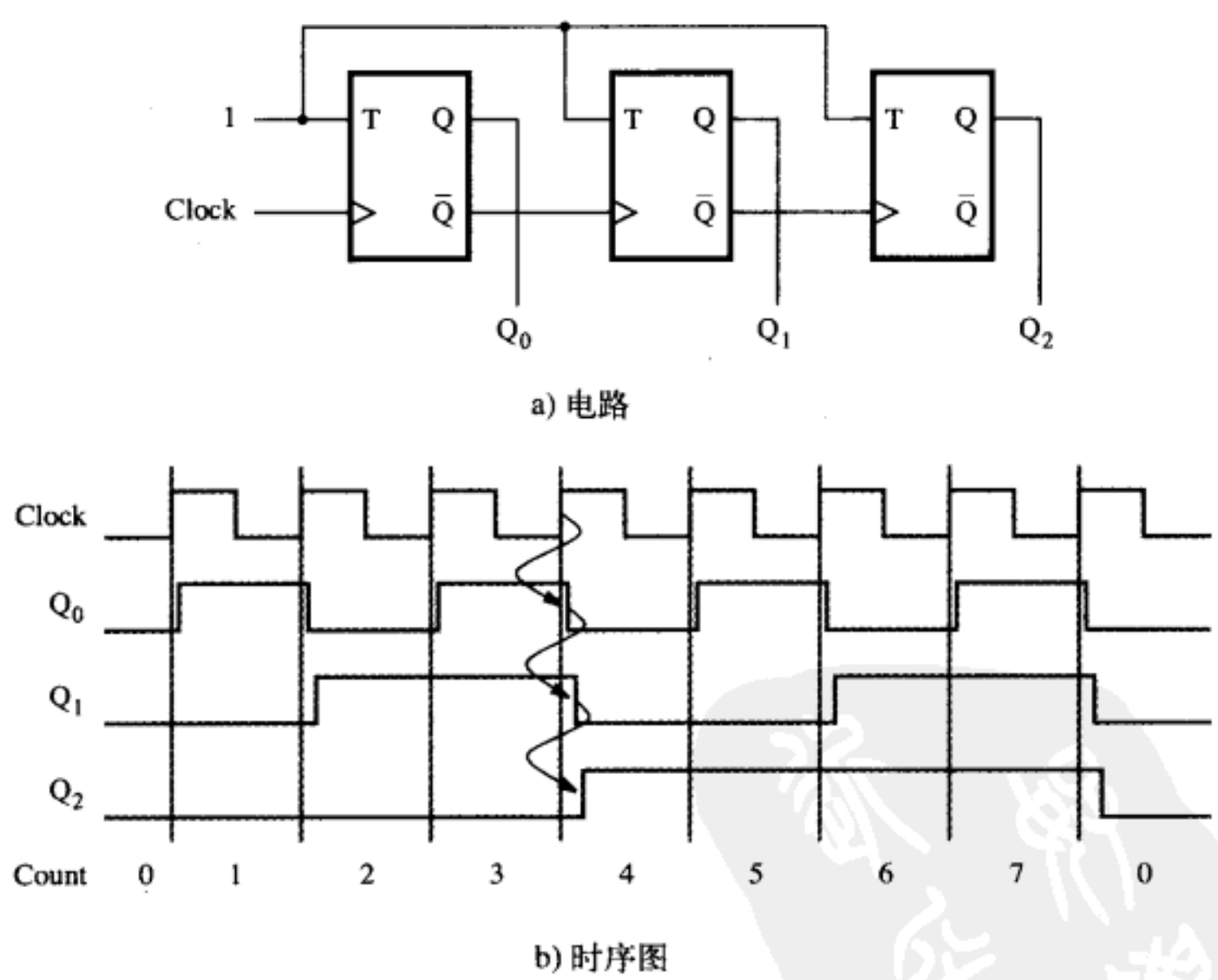
计数器可以用第5章中讨论过的加法/减法器电路来实现，也可以用7.8节讨论过的寄存器来实现。但是，因为我们只需要计数器有1的变化，故没有必要用如此复杂的电路。我们可以用更简单价格更便宜得多的电路来实现。我们将说明如何用T触发器和D触发器来设计计数器电路。

7.9.1 异步计数器

最简单的计数器电路可以用T触发器构成，因为翻转特性很自然地适用于计数操作的实现。

由T触发器构成的递增计数器

图7-20a给出了一个三位的计数器，计数范围从0到7。三个触发器的时钟输入用级联的方式连接。每个触发器的T输入端与1相连，这意味着触发器的状态在每个时钟的正跳变沿翻转。我们假设本电路的目的是计算发生在*Clock*输入端的脉冲个数。因此，第一个触发器的时钟输入端与*Clock*线相连。另两个触发器的时钟输入由前一级的$\overline{Q}$输出驱动。因此，随着前级触发器的状态从$Q=1$改变到$Q=0$，$\overline{Q}$同时产生正跳变沿，后面触发器的状态便发生翻转。



a) 电路

b) 时序图

图7-20 三位递增计数器

图7-20b展示了计数器的时序图。Q_0的值每个时钟周期改变一次。变化发生在时钟正跳变沿之后，稍有延迟。延迟是由于触发器的传播延迟引起的。因为第二个触发器由$\overline{Q_0}$作时钟，所以Q_1的值在Q_0的下降沿之后变化，稍有延迟。同理，Q_2的值在Q_1信号的下降沿之后变化，稍有延迟。若我们把$Q_2Q_1Q_0$的值作为一个数来看，则时序图说明计数顺序是0，1，2，3，4，

5，6，7，0，1，等等。电路是一个模8计数器，因为它递增计数，我们叫它递增计数器。

图7-20a中的计数器有三个部分，每部分包含一个触发器。只有第一部分直接对*Clock*信号作出响应；我们称这部分与时钟同步。另两部分在延迟之后对时钟作出响应。例如，当*Count* = 3，下一个时钟脉冲会使*Count*变成 4。如图7-20b时序图中的箭头所示，这个变化需要三个触发器的所有状态翻转。可以观察到Q_0在*Clock*的正跳变沿时刻经过一小段延迟之后变为0。而此时Q_1和Q_2触发器还没有发生变化；因此有一小段时间内$Q_2\,Q_1\,Q_0$ = 010。在第二个传播延迟之后Q_1变为0，这时$Q_2\,Q_1\,Q_0$ = 000。最后，Q_2在第三段传播延迟之后变为1，这时电路达到稳定状态，$Q_2\,Q_1\,Q_0$ = 100。这个行为与图5-6所示的行波进位加法器的逐位进位很相似。图7-20a所示的电路是一个异步计数器，或者叫行波计数器。

由T触发器构成的递减计数器

把图7-20a中的电路作简单的修改就得到图7-21a所示的电路。这两个电路的不同之处仅在于，图7-21a中的第二个和第三个触发器的时钟输入是由其前级触发器的输出端 Q 驱动，而不是输出端 $\overline{Q}$。图7-21b给出了时序图，说明了这个电路的计数顺序是0，7，6，5，4，3，2，1，0，7，等等。因为它的计数方式是递减的，所以称它为递减计数器。将图7-20a和图7-21a所示的电路结合在一起，便可以构成一个既可以递增又可以递减的计数器。在练习题7.16中，我们把设计这样的计数器作为练习。

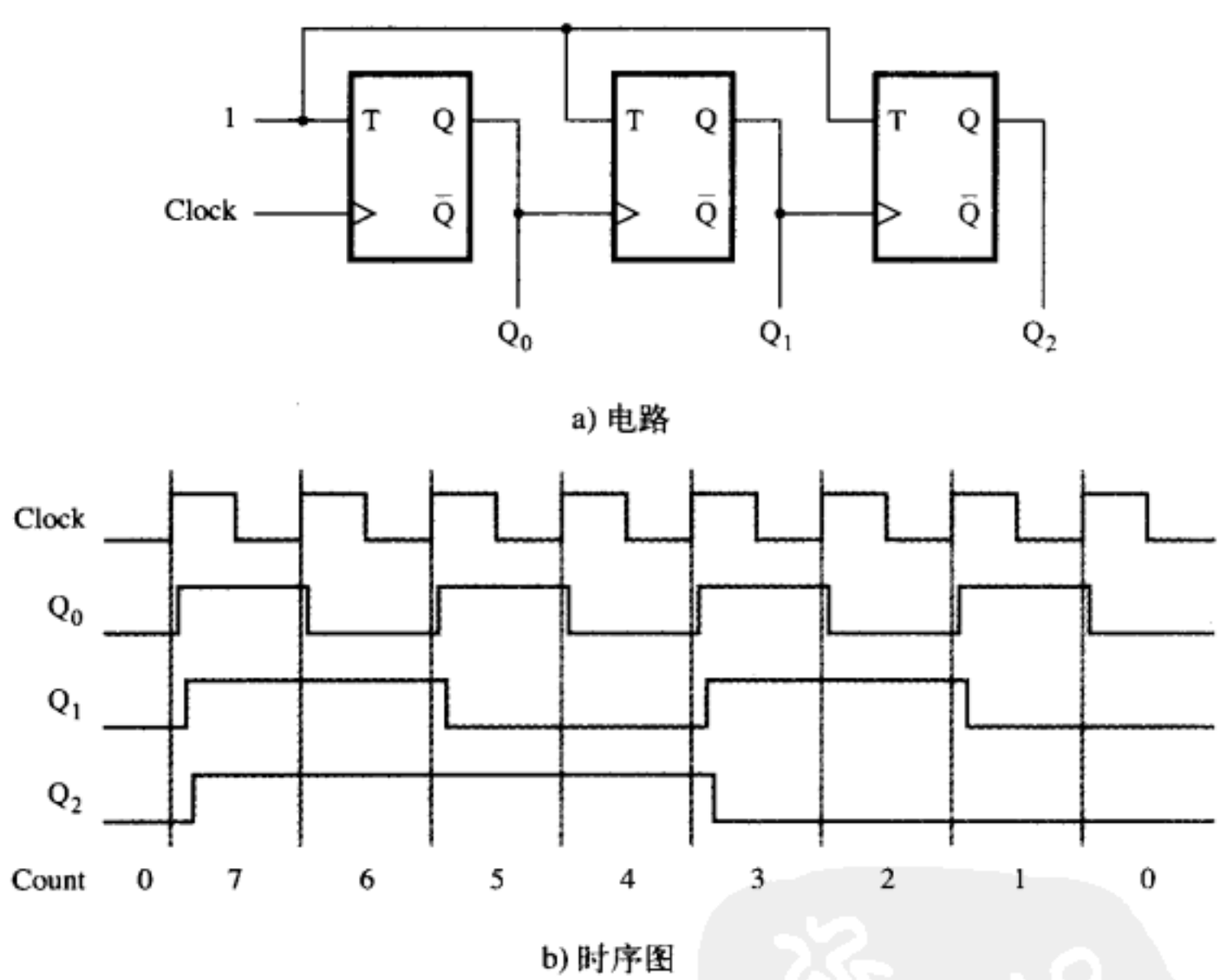


图7-21 三位递减计数器

7.9.2 同步计数器

图7-20a和图7-21a所示的异步计数器是简单的，但是运行速度不快。若用这种方式构成位数很多的计数器，由时钟级联引起的延迟可能会太长，以致不能满足速度性能的需求。用下面讨论的方法，将所有的触发器用同一时钟触发，我们可以构建一个速度更快的计数器。

由T触发器构成的同步计数器

表7-1展示了在8个连续时钟周期下，三位递增计数器的内容（假设计数器的初始值为0）。

观察表7-1中每一行位的取值模式，很明显Q_0位每个时钟周期都变化。Q_1位仅在$Q_0 = 1$的时候变化。Q_2位仅在Q_1和Q_0同时为1时发生变化。一般情况下，对于一个n位递增计数器，给定的触发器仅在它前面的所有触发器都处在$Q = 1$的状态时才会改变状态。因此，若用T触发器来实现计数器，则T输入端应定义为：

表7-1 同步递增计数器的导出

时钟周期	Q_2 Q_1 Q_0
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1
8	0 0 0

Q_1 改变
Q_2 改变

$$
\begin{aligned}
T_0 &= 1 \\
T_1 &= Q_0 \\
T_2 &= Q_0 Q_1 \\
T_3 &= Q_0 Q_1 Q_2 \\
&\vdots \\
T_n &= Q_0 Q_1 \cdots Q_{n-1}
\end{aligned}
$$

图7-22a所示是一个基于以上表达式的四位计数器。在这个例子中，我们不采用逐级增加与门的办法，因为这样做会带来扇入（fan-in）问题，所采用的是分解的方法，如图所示。这样的电路结构不会使计数器的响应变慢，因为所有的触发器都在时钟正跳变沿加片刻延时后改变其状态。请注意，Q_0的变化需要通过几个与门的延迟才能传到计数器的高位触发器，这需要一定的时间。这个时间一定不能超过时钟周期。实际上，它必须小于时钟周期减去触发器的建立时间。

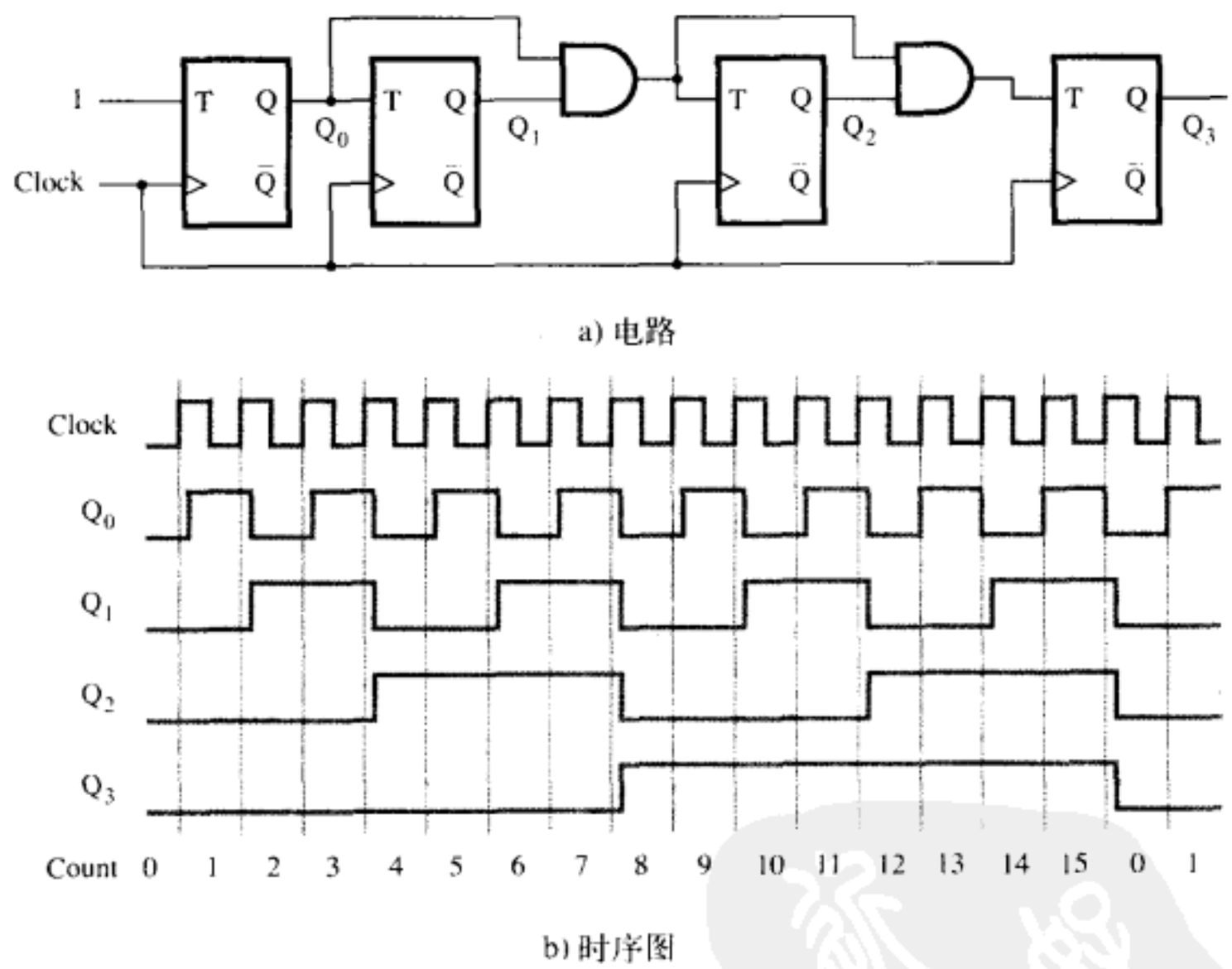


图7-22 四位同步递增计数器

图7-22b所示的时序图展示了模16递增计数器电路的行为。因为所有变化发生在*Clock*信号有效沿之后的相同延迟后，所以该电路称为同步计数器。

使能和清零功能

每个时钟脉冲的到来都会使图7-20至图7-22所示的计数器改变其计数值。我们经常需要能禁止计数器的运行使得计数值保持当前状态。这可以通过添加一个使能控制信号来实现，见图7-23所示。该电路是在图7-22的计数器的基础上增加一个与门，使能信号直接输入第一个

触发器的T输入端。把使能端也连向各级与门链路，这意味着若$Enable = 0$，则所有触发器的T输入都等于0。若$Enable = 1$，则计数器就会像前面解释的那样操作。

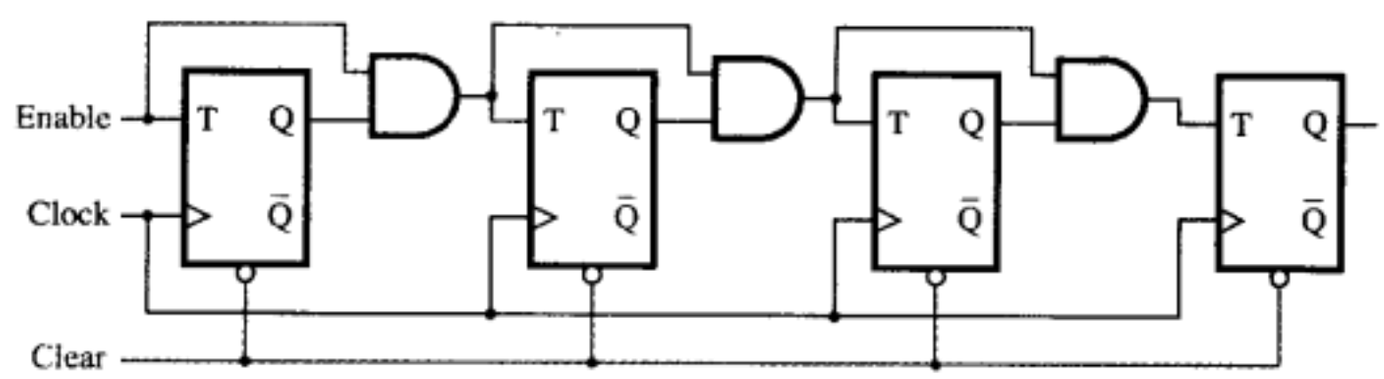


图7-23 添加了使能和清零能力的递增计数器

在很多实际应用中计数器的起始值必须为0。若触发器可以被清零的话，这就能很容易地实现（如7.4.3节中所解释的）。将所有触发器的清零输入端连在一起，用一个清零控制输入来驱动即可。

由D触发器实现的同步计数器

虽然T触发器的翻转特性使它成为实现计数器的自然选择，但也可以用其他类型的触发器来构造计数器。JK触发器即可用来构建计数器，所用的方法与用T触发器构造计数器相同。只要把J端和K端连接在一起，JK触发器就变成了T触发器。我们现在考虑用D触发器来达到这个目的。

如何用D触发器来实现计数器不是那么明显。我们将在第8章介绍推导该电路的正式方法。这里将介绍一种满足需求的电路结构，而将推导方法留到第8章再介绍。图7-24给出了一个四

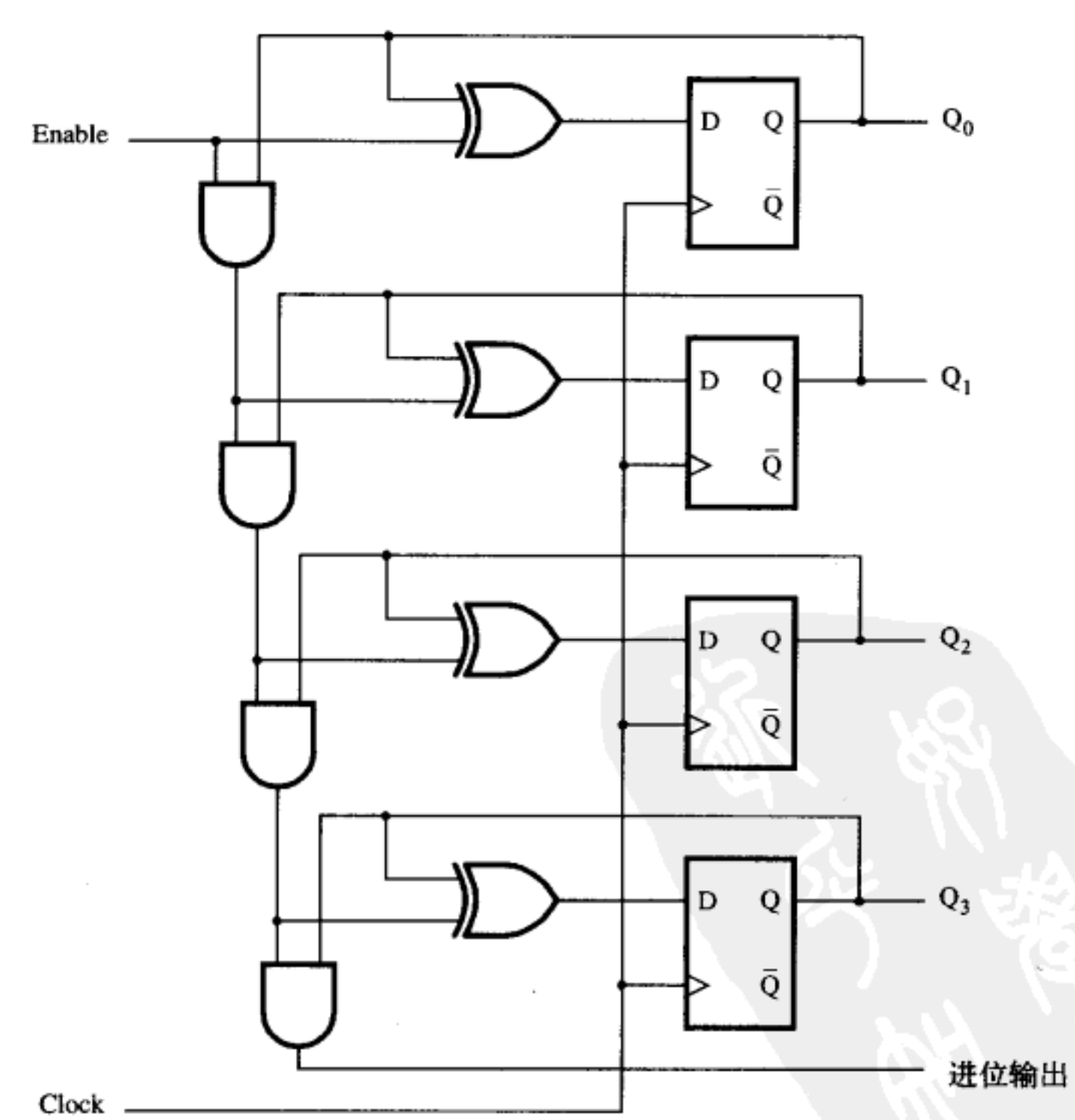


图7-24 用D触发器构成的四位计数器

位递增计数器，计数顺序是0，1，2，…，14，15，0，1，等等。计数用触发器的输出端$Q_3Q_2Q_1Q_0$表示。若我们假设使能端$Enable = 1$，则触发器的D输入端可以用以下表达式来定义：

$$D_0 = \overline{Q_0} = 1 \oplus Q_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

更大的计数器的第i级用以下表达式定义：

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2}\cdots Q_1Q_0$$

我们将在第8章讲解如何推导这些表达式。

我们已添加了使能控制信号，所以若$Enable = 1$，计数器便可对时钟脉冲进行计数。实际上，为了实现图7-24的电路，将上面的表达式修改后表示如下：

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$
$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$$

计数器的运行是以我们对表7-1所作的观察为基础的，这就是第i级触发器的状态只在它前面的所有触发器都处于状态$Q = 1$时才发生改变。这使组成第i级的与门输出为1，从而使与D_i连接的异或门的输出等于Q_i。否则，异或门的输出将使$D_i = Q_i$，从而使该触发器保持原状态。这类似于超前进位加法器的进位传递（见5.4节）；因此与门链可以认为是进位链。尽管该电路只是一个四位计数器，我们仍可以添加一个与门来产生“进位输出”。有了这个进位输出，将两个这样的四位计数器拼接成一个八位计数器就变得非常容易。

最后，读者应该注意到图7-24所示的计数器本质上与图7-23所示的电路相同。我们曾经在图7-16a中展示，添加以下门控信号可以将D触发器改造为T触发器：

$$D = Q\overline{T} + \overline{Q}T = Q \oplus T$$

因此，在图7-24的每一级，D触发器和相连的异或门实现了T触发器的功能。

7.9.3 可并行置数的计数器

计数器通常必须从零开始计数，这个状态可以用触发器的清零功能实现（如图7-23）。但是，有时希望从不同的起始值开始计数。为实现这种操作模式，计数器电路必须具有可加载初始值的输入端。用清零和预置两个输入端来实现这一目标是可行的，下面我们将讨论一个更好的方法。

图7-24所示的电路可以修改成具有并行加载功能的电路（见图7-25所示）。在每个触发器的输入端D前插入一个2选1多路器。多路器的一个输入用来产生正常计数操作，另一个输入提供加载数据。当$Load = 0$时，电路计数。当$Load = 1$时，新的初始值$D_3D_2D_1D_0$被加载到计数器中。

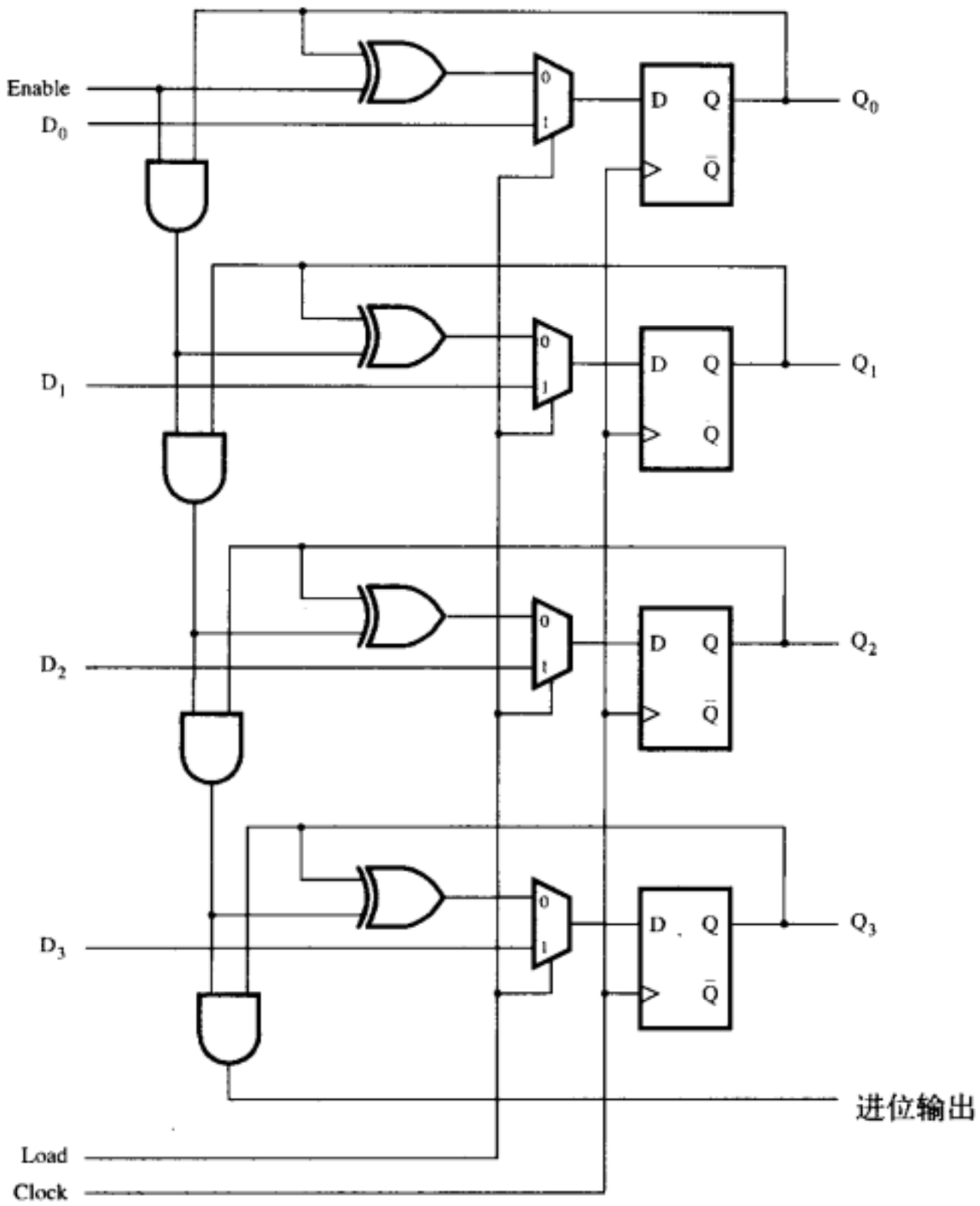


图7-25 具有并行加载能力的计数器

7.10 同步复位

我们已注意到在开始计数之前，是否可以将计数器的内容清零或者复位是很重要的。这一功能可以通过添加单个触发器的清零功能实现。但是，对在正常的计数过程中将计数复位至0，我们也许也有兴趣。n位的递增计数器的功能自然就是模2^n计数器的功能。假设我们希望有一个模不是2的幂次的计数器。例如，我们可能想要设计一个模6计数器，该计数器的计数序列为：0，1，2，3，4，5，0，1，等等。

最直接的方法是当计数到5时使计数器复位。可以用一个与门检测5的到来。实际上，只要探知$Q_2 = Q_0 = 1$就足够了，在我们需要的计数顺序中只有数5才有这种情况。图7-26a给出了基于这种方法的电路。该电路使用了图7-25所示那种类型的三位同步计数器。计数器的并行加载特征用来在计数至5时复位计数器。复位行为发生在计数到5后的时钟上升沿，包括把$D_2 D_1 D_0 = 000$加载到触发器中。期望的计数顺序实现了（见图7-26b所示的时序图），在一个时钟周期里建立起计数的每个值。因为计数器在时钟有效沿时复位，我们称这种类型的计数器为*同步复位计数器*。

现在考虑用单个触发器的清零特性（来实现）的可行性，而不是用并行载入的方法。图7-27a中的电路说明了一种可能性。它用的是图7-22a中的计数器结构。因为清零输入端低电平有效，可以用与非门来检测5的发生，同时复位所有的触发器。从概念上看起来这个电路应该可以很好地工作，但仔细考察后发现该电路有一个隐患。图7-27b画出了该电路的时序图，说明当计数值等于5时，便会出现问题。只要计数等于这个值，与非门就会触发复位行为。触发

器在与非门察觉到5之后，经过很短时间的延迟，然后立刻被清零。计数值等于5的维持时间依赖于电路中门的延迟，而不是时钟。因此，信号值$Q_2Q_1Q_0 = 101$只保持了很短一段时间（远小于一个时钟周期）。在特定的情况下，这样的计数器可能适用，也可能完全不能满足要求。比如，若计数器应用于一个所有操作都是用同一时钟进行同步的数字系统中，则这个表示$Count = 5$的狭窄脉冲可能无法被系统中的其他部分看到。为了解决这个问题，我们可以试着用一个模7计数器替代（假设系统忽略了表示6的窄脉冲）。这不是设计电路的好方法，因为在实际电路中出现的意外脉冲经常会造成不可预知的后果。图7-27a中用的方法称为*异步复位*。

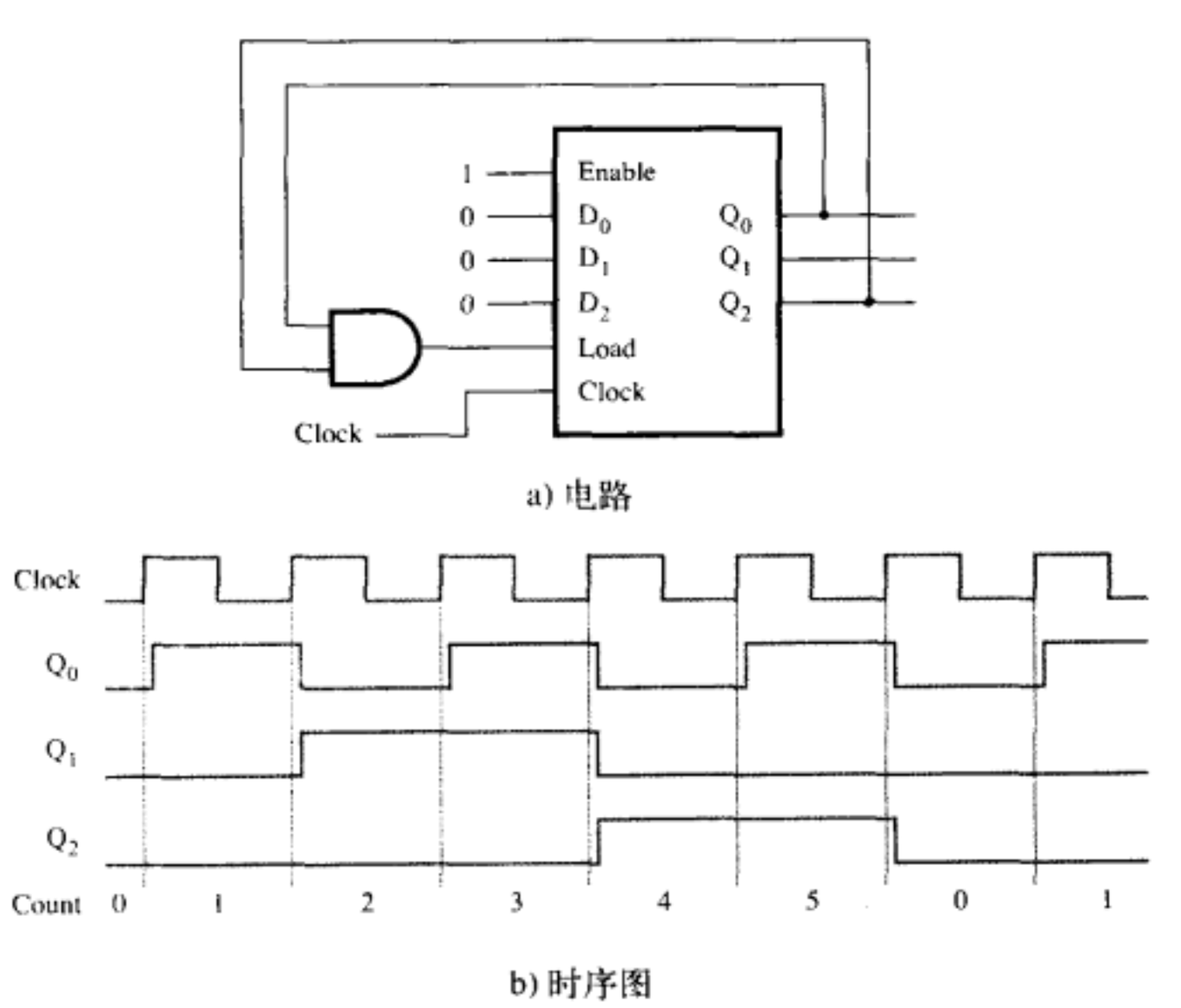


图7-26 具有同步复位功能的模6计数器

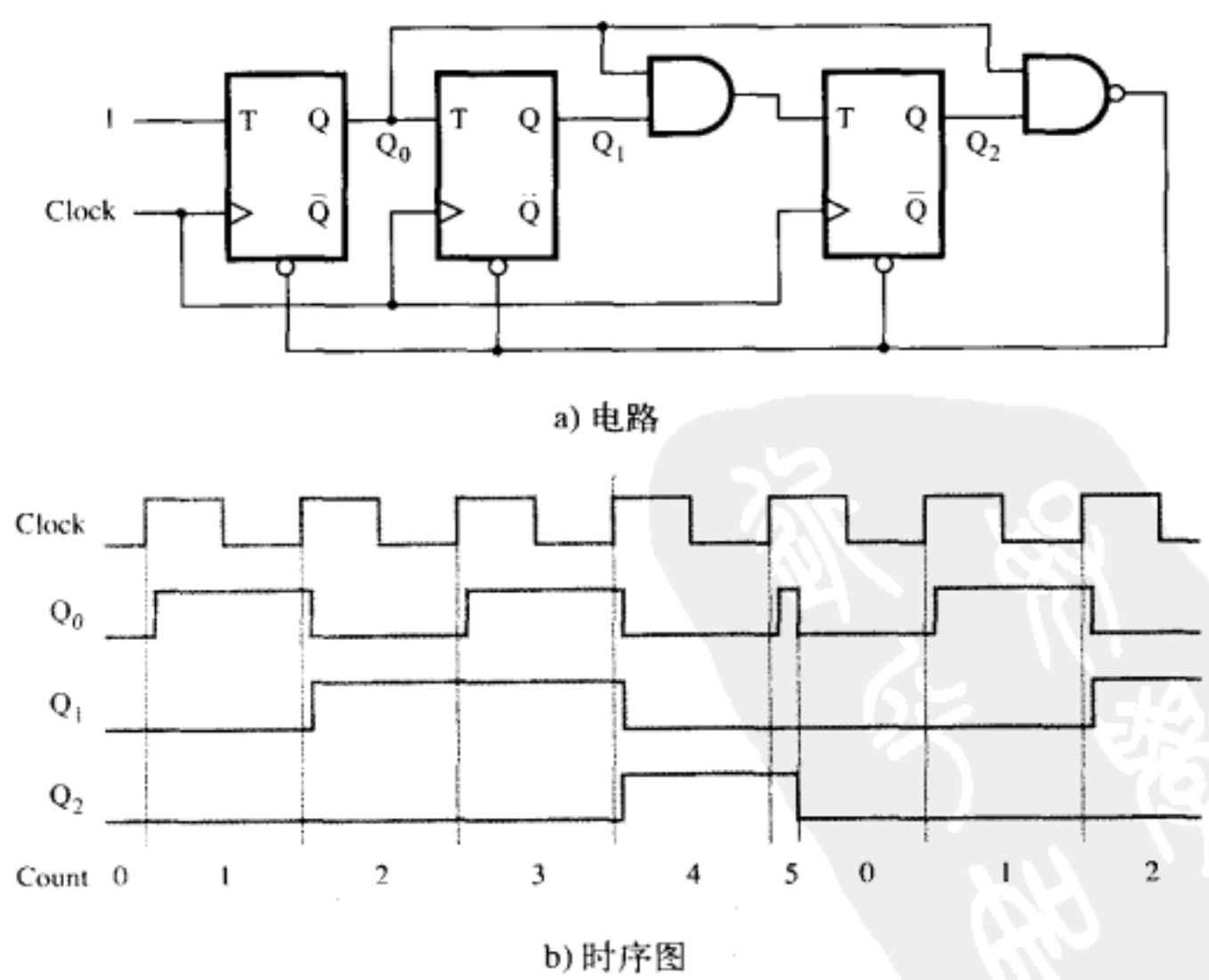


图7-27 有异步复位功能的模6计数器

图7-26b和图7-27b所示的时序图告诉我们，选用同步复位方案比选用异步复位方案好。在正常的计数顺序不得不被某些非零值的加载所中断的情况下，我们也可以通过观察得到同样的结论。新的计数值可以通过并行加载的方法送到计数器中。通过对单个触发器清零和设置状态来得到期望的数值，也存在在异步复位方案中讨论过的同样问题。

7.11 其他类型的计数器

本节讨论其他三种在实际应用中经常用到的计数器。第一种使用十进制计数顺序，另外两种产生不是二进制数的码序列。

7.11.1 BCD计数器

二-十进制（BCD）计数器可以用7.10节中解释过的方法设计。图7-28展示了一个两位的BCD计数器。它由两个模10计数器组成，每个BCD数字用一个模10计数器，该计数器可以用图7-25中所示的并行加载四位计数器实现。注意，在模10计数器中，当计数值达到9以后必须将四个触发器复位。因此，当$Q_3 = Q_0 = 1$时，每一级的*Load*输入为1，从而使0在下一时钟的正跳变沿时加载到触发器中。每当表示个位的第1级计数器BCD_0为9时，必须启动表示十位的第2级计数器，这样在下一时钟的脉冲到来时，计数值才可以增加。这可以通过使BCD_1的*Enable*信号只在$BCD_0 = 9$的时钟周期里为高电平，而其余时间都保持低电平来实现。

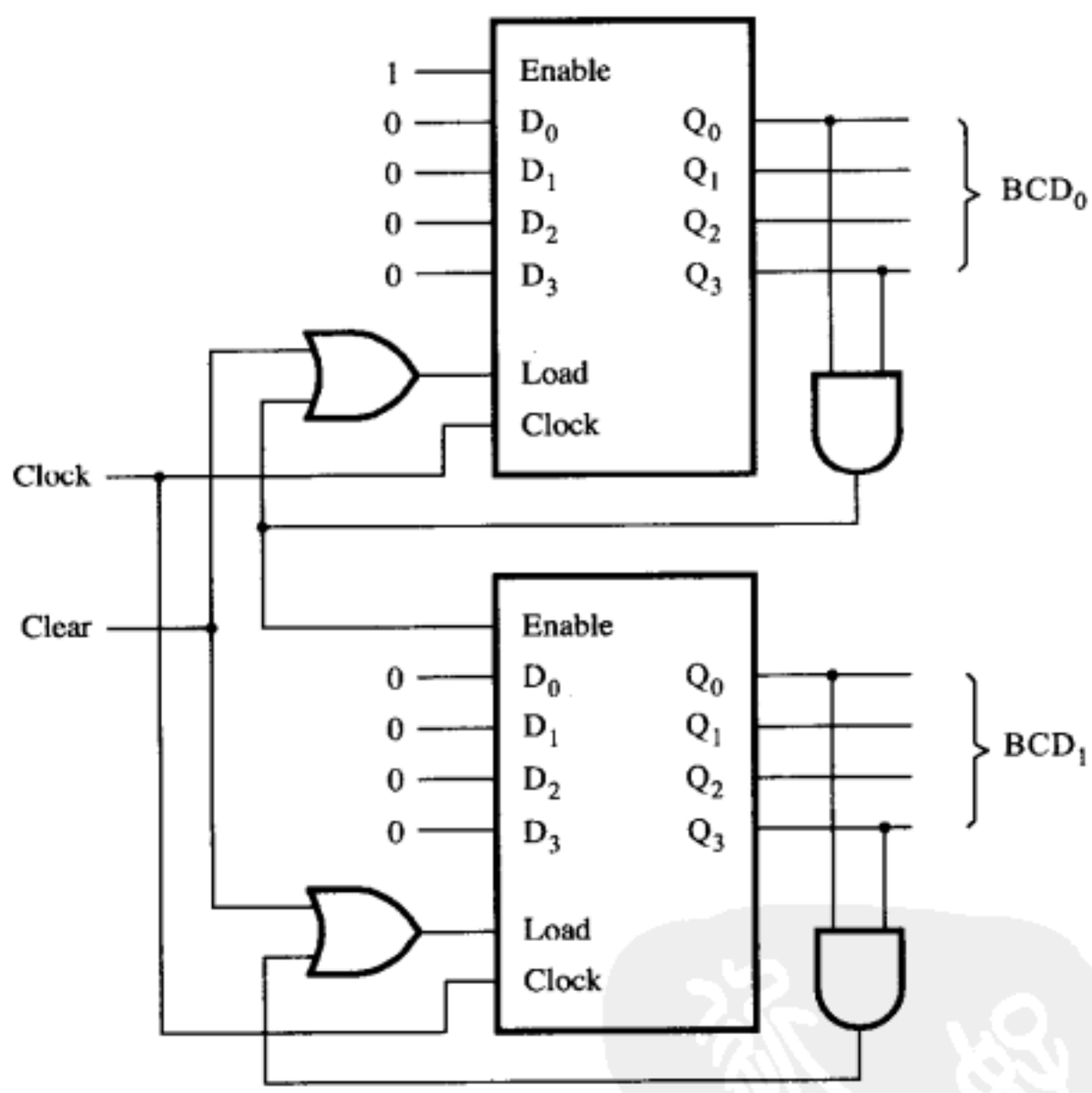


图7-28 一个两位的BCD计数器

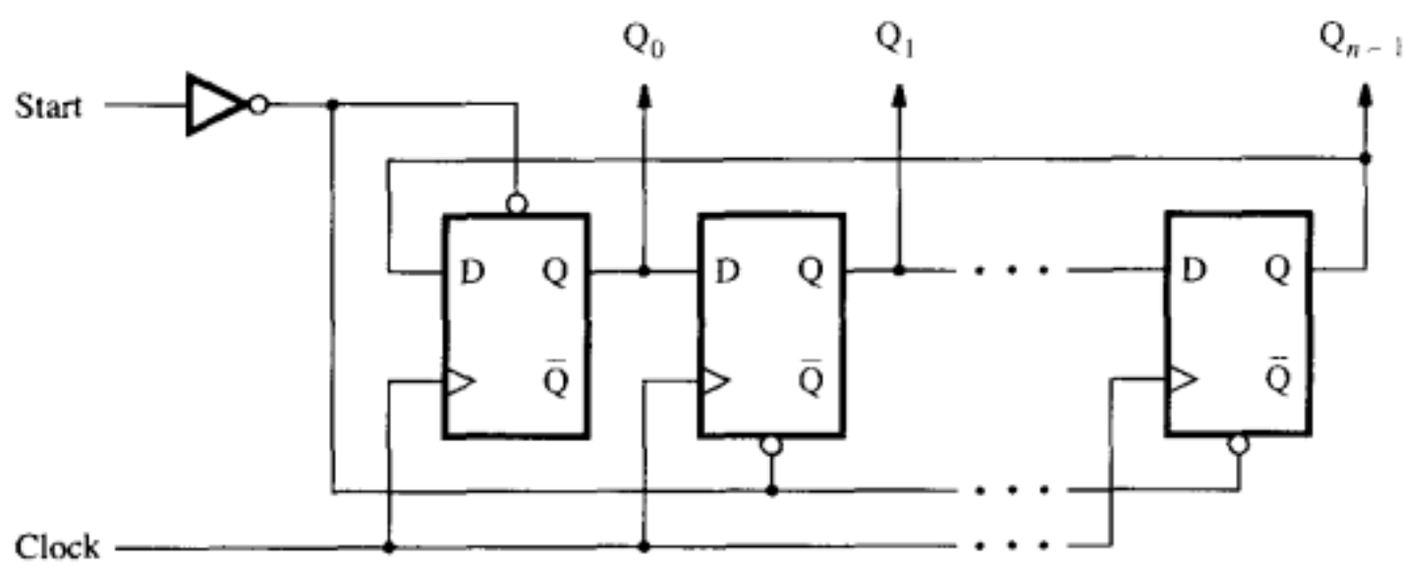
在实际电路中，必须能够通过控制信号的改变把计数器的内容清零。可以在电路中添加两个或门来实现计数器清零。控制输入端*Clear*可以用来把0加载到计数器中。在这种情况下，高电平时*Clear*端有效。图7-81给出了两位BCD计数器的Verilog代码。

在任何数字系统中，通常都会有一个或几个时钟信号来驱动所有的同步电路。在前面介绍的计数器（同前面图中展示的所有计数器一样）中，我们假设的目的是计数时钟脉冲的数目。当然，这些计数器可以计数任何信号的脉冲数目（若用此脉冲来代替时钟信号）。

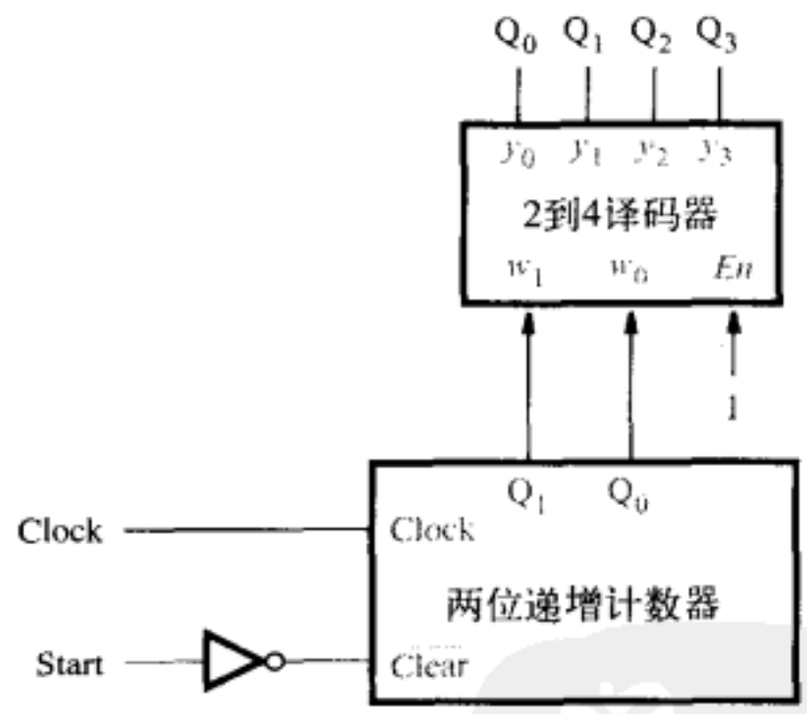
7.11.2 环形计数器

在前面介绍的计数器中，计数值由计数器中的触发器状态来表示。所有情况下，计数值都是一个二进制数。使用这种计数器时，若希望当计数达到某特定值时产生一个动作，则必须检测这个计数值是否达到。这可以用与门实现，见图7-26至图7-28的电路所示。

我们可以设计一种类似于计数器的电路，该电路中的每个触发器只有在某特定计数值时$Q_i = 1$，而在其他所有的计数值时$Q_i = 0$。因此$Q_i = 1$直接表明了某对应计数值的发生。实际上，因为这个计数值不是二进制数，所以把触发器的输出看作编码更合适些。这样的电路可以由一个简单的移位寄存器实现，如图7-29a所示。移位寄存器的最后一级的输出（Q）反馈至第一级作为输入，从而构成了一种环形结构。若一个1进入这个环，则1会在连续时钟周期中在这个环中移位。例如，在一个4位的环形结构中，则可能产生的码$Q_0 Q_1 Q_2 Q_3$将是1000，0100，0010，0001。正如6.2节中介绍过的那样，这种编码只有一个码元为1，其余码元都是0，我们称这种编码为独热码。



a) n位的环形计数器



b) 四位环形计数器

图7-29 环形计数器

图7-29a中的电路称为环形计数器。必须在第一个触发器注入1才能启动环形计数器的操作。这通过*Start*控制信号实现，使最左面的触发器置1，其他触发器则清为0。我们假设*Start*信号的所有变化发生在有效时钟沿之后，以免违反触发器的时序参数。

图7-29a中的电路可以用来构建一个任意位（n位）的环形计数器。对于$n = 4$的特殊情况，图7-29b展示了如何用一个两位递增计数器和译码器实现一个环形计数器。当*Start*置位为1时，计数器复位至00。在*Start*变回0之后，计数器正常计数。2到4译码器（曾在6.2节介绍）把计数器的输出转变为独热码。对于计数值00，01，10，11，00等，译码器对应产生$Q_0 Q_1 Q_2 Q_3 =$

1000，0100，0010，0001，1000等。这个电路结构可用于较大的环形计数器，只要位数是2的幂。我们将在7.14节中举例说明如何用图7-29b中的环形计数器作为子电路构成较大型的电路。

7.11.3 Johnson计数器

若我们把环形计数器的最后一个触发器的 $\overline{Q}$（而不是Q）作为反馈接到第一个触发器的输入，就可以得到一个有趣的改变，如图7-30所示。这个电路就是著名的Johnson计数器。n位的这种类型的电路可以产生长度为$2n$的计数序列。例如，一个四位的计数器可以产生计数序列0000，1000，1100，1110，1111，0111，0011，0001，0000等。请注意在这个序列中，相邻的两个码之间只有一位不同。

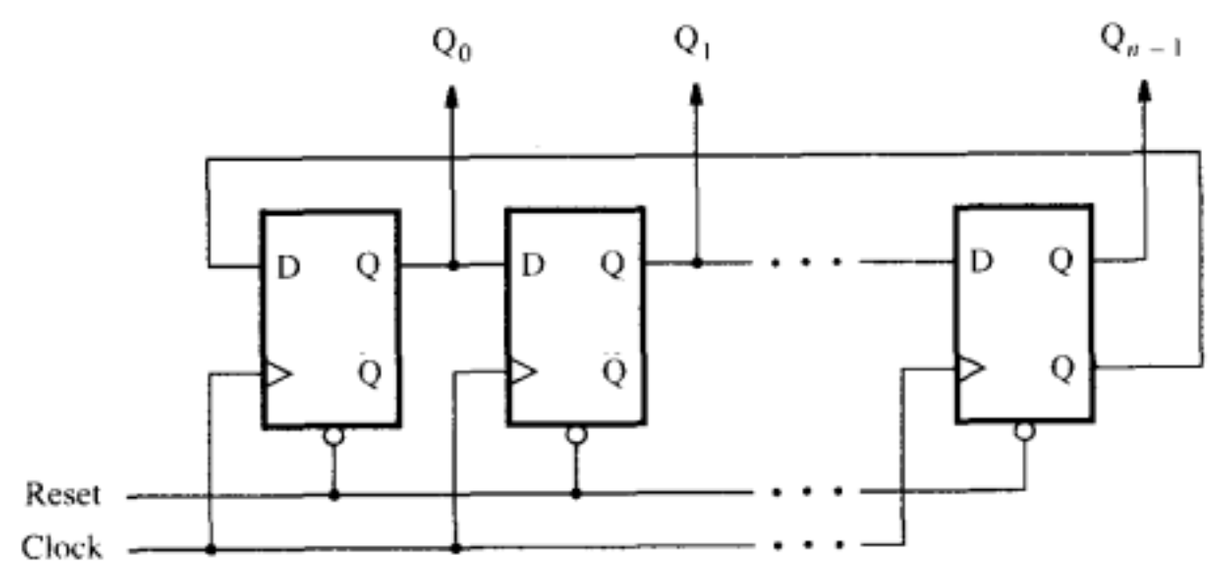


图7-30 Johnson计数器

为了对Johnson计数器进行初始化，必须如图所示对所有的触发器进行复位。注意，如果初始化进行得不正确，不管是Johnson计数器还是环形计数器，都不能生成想要的计数序列。

7.11.4 计数器设计要点

本章介绍的时序电路，即寄存器和计数器，都有规则的结构，可以用直观的方法进行设计。在第8章中，我们将介绍设计时序电路更正规的途径，并说明如何用这种途径推导本章中所介绍的电路。

7.12 用CAD工具在设计中加入存储元件

本节说明了如何用原理图输入方法或者编写Verilog代码的方法设计包含存储元件的电路。

7.12.1 在电路原理图中添加存储元件

创建电路的一种方法是绘制电路原理图，该电路原理图以逻辑门为基本元件构造出锁存器和触发器。因为这些存储元件可以有许多种用途，大多数CAD系统的库中包含这些模块供设计者选用。图7-31展示了由图形编辑器绘制的电路原理图，该电路图中包括三种类型的触发器，这些触发器都属于CAD系统库中的部件。图中最上面的部件是门控D锁存器，中间的部件是正跳变沿触发的D触发器，下面的部件是正跳变沿触发的T触发器。D触发器和T触发器包含异步清零端和置1端（低电平有效）。若这些输入端在

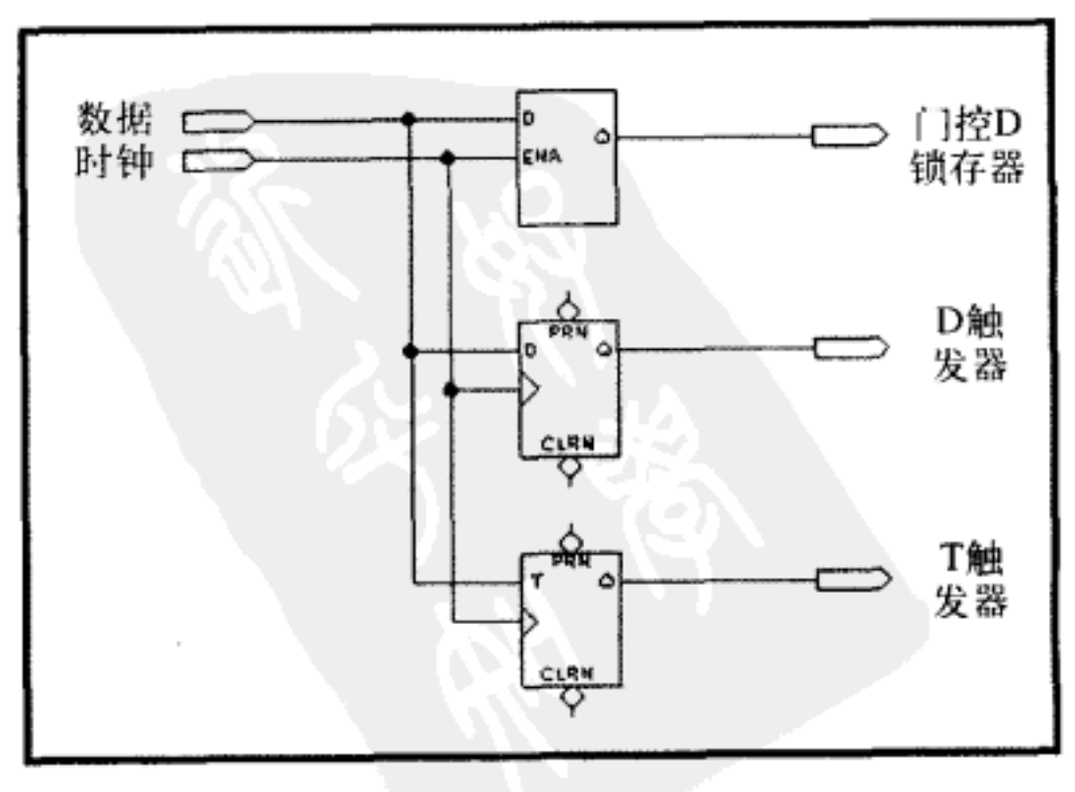


图7-31 同一图中的三种类型的存储元件

图中没有连信号，则CAD工具自动给它们分配为1的默认值，使其无效。

当门控D锁存器被综合成芯片中的电路时，CAD工具产生的不是如7.2节中所示的那种由交叉耦合的或非门或者与非门组成的结构。在某些芯片中，例如CPLD芯片，综合器产生的可能为图7-32所示的与-或电路。该电路的功能等价于7.2节中的交叉耦合电路。芯片电路中用到的是积之和形式的电路，因为这种形式的电路更适合于CPLD的宏单元来实现。有一点应该说明：从功能的角度来看，该电路可以通过去掉有两个输入端（*Data*和*Latch*）的与门而得到简化。这个门去掉后，最上面的与门在时钟为1时把值存入锁存器，最下面的与门在时钟为0时保持存储的值。但是，若没有这个门，电路可能会存在一个称为静态冒险的时序问题。关于冒险的详细解释将在9.6节讲解。

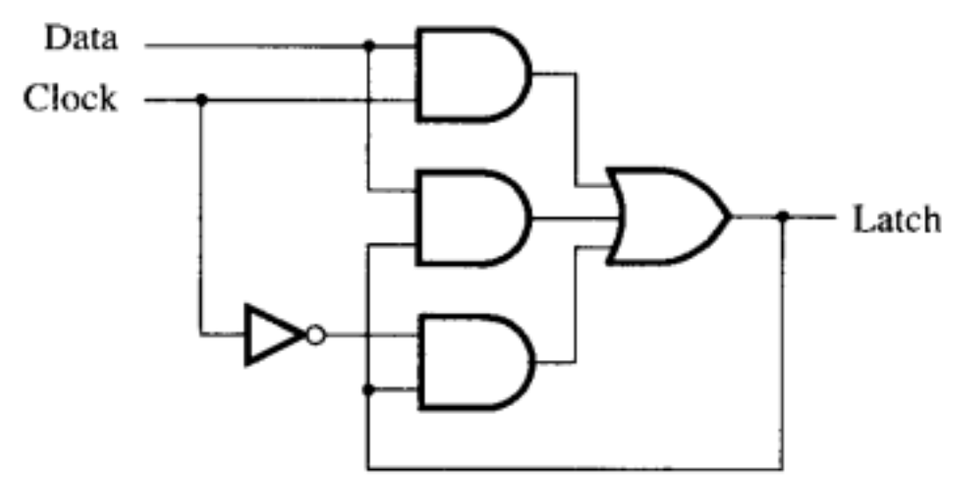


图7-32 由CAD工具产生的门控D锁存器

图7-31中的电路可以在CPLD中实现，如图7-33所示。D触发器和T触发器都可以用芯片中的触发器来实现，芯片中的触发器可以根据需要随意配置为D触发器或者T触发器。实现图7-31所示电路所需要的门和连接线在图中用浅色表示。

图7-34画出了CPLD芯片的具体实现电路（如图7-33所示）的时序仿真波形。*Latch*信号

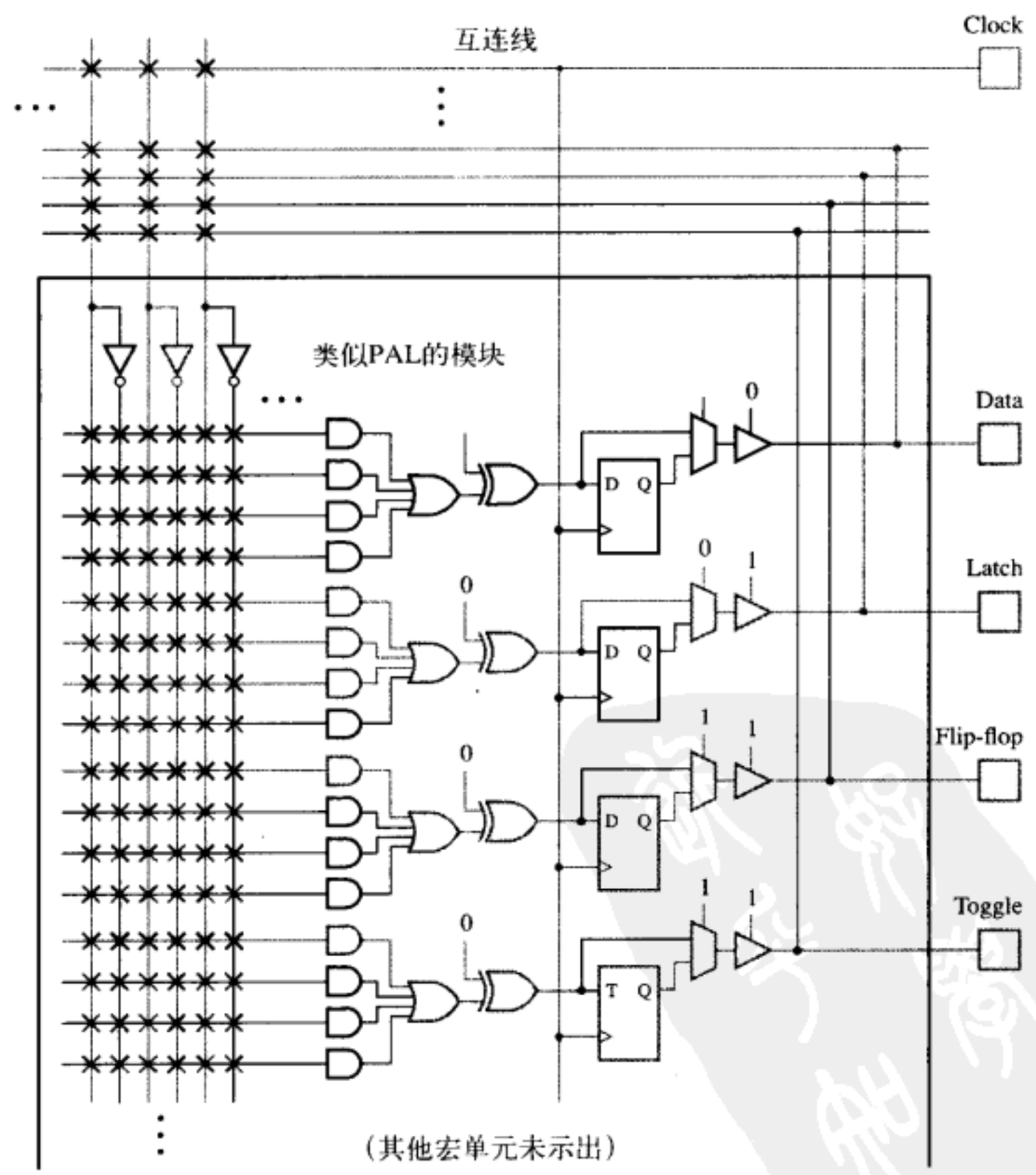


图7-33 图7-31在CPLD中的实现图

（该信号为门控D锁存器的输出）如图7-32中显示那样实现，在*Clock*信号为1的时候，跟随*Data*输入值变化。由于芯片中的传播延迟，锁存器（*Latch*）信号相对于*Data*信号有一定的时延。由于触发器（*Flipflop*）信号是D触发器的输出，它只在时钟正跳变沿之后改变。同理，T触发器的输出（图中用*Toggle*表示）在时钟上升沿且*Data* = 1时翻转。时序图说明了从时钟正跳变沿发生在芯片的输入端，直到触发器的输出改变出现在芯片的输出引脚之间的延迟。这个时间叫做时钟到输出之间的延时t_{co}。

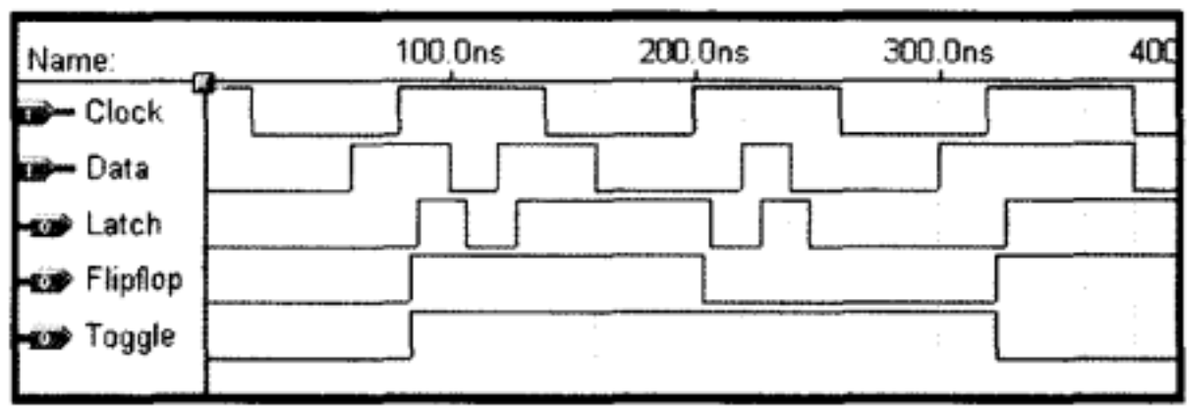


图7-34 图7-31所示存储元件的时序仿真图

7.12.2 用Verilog代码实现存储元件

在6.6节，我们曾描述过许多种Verilog语句结构。我们现在展示如何用这些语句描述存储元件。

指定存储元件的一种简单方法是用**if-else**语句来描述由数据电平和输入时钟变化而产生的期望响应。考虑以下**always**块：

```
always@(Control or B)
if(Control)
A = B;
```

这里的*A*是一个**reg**型变量。这段代码说明若变量*Control* = 1，则*A*的值应该等于*B*的值。但是声明语句中没有说明若*Control* = 0时会产生什么行为。在缺少赋值的情况下，Verilog编译器假定由**if**顺序语句引起的*A*值，在**if**语句下次做出判断之前必须保持已有的值。具有这种蕴涵记忆功能的电路可以用锁存器实例的引用来实现。

```
module D_latch (D, Clk, Q);
  input D, Clk;
  output reg Q;

  always @(D, Clk)
    if (Clk)
      Q = D;

endmodule
```

图7-35 门控D锁存器的代码

例7.1（门控D锁存器的代码） 图7-35所示的代码定义了一个叫做*D_latch*（D锁存器）的模块，该模块的输入端为*D*和*Clk*，输出端为*Q*。**if**子句定义了当*Clk* = 1时，输出*Q* 必须等于*D*的值。因为没有写出**else**子句，所以综合后产生的电路将是一个锁存器，当*Clk* = 0时使*Q*的值保持原来的值。因此，该代码描述了一个门控*D*锁存器。敏感列表包括*Clk*和*D*，因为这两个信号都可以引起输出*Q*值的变化。 ■

always构造可以用来定义对敏感列表内的信号变化做出响应的电路。虽然在目前我们所举的例子中，**always**块都是对信号电平敏感的，但它也可以用来说明由信号特定沿引起的响应。可以用Verilog关键字**posedge**和**negedge**说明想要的沿，这些关键字可以用来实现沿触发的电路。

```
module flipflop (D, Clock, Q);
  input D, Clock;
  output reg Q;

  always @(posedge Clock)
    Q = D;

endmodule
```

图7-36 D触发器的代码

例7.2（D触发器的代码） 图7-36定义了一个名为触发器（flipflop）的电路，这是一个正跳变沿触发的D触发器。敏感列

表只包括时钟信号，因为时钟信号是唯一可以引起*Q*输出端发生变化的信号。关键字**posedge**说明变化只可能发生在*Clock*的正跳变沿。这时输出*Q*的值与输入*D*的值相等。因为**posedge**出现在敏感列表中，所以*Q*将是触发器的输出。 ■

7.12.3 阻塞赋值和非阻塞赋值

到目前为止，在所有我们举的Verilog例子中，都用等号进行赋值，例如：

```
f = x1 & x2;
```

或者

```
C = A + B;
```

或者

```
Q = D;
```

这种等号叫做阻塞赋值。Verilog编译器按照这些语句在**always**块中编写的先后次序顺序地执行。若一个变量是由阻塞赋值语句赋值的，则这个新赋的值会被该块中后面所有的语句使用。

例7.3 考虑图7-37中的代码。因为**always**块对时钟的正跳变沿敏感，Q1和Q2都将作为D触发器的输出。然而，由于代码中所用的是阻塞赋值，所以这两个触发器不会像读者希望的那样以级联的方式连接。第一条语句Q1 = D使Q1的值等于D的值。新的Q1值就被用在紧接的语句Q2 = Q1中，结果得到Q2 = Q1 = D。综合得到的电路为两个平行的触发器，见图7-38所示。综合工具可能将删除其中一个多余的触发器作为优化措施。

```
module example7_3 (D, Clock, Q1, Q2);
   input D, Clock;
   output reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 = D;
      Q2 = Q1;
   end

endmodule
```

图7-37 两个级联触发器的错误代码

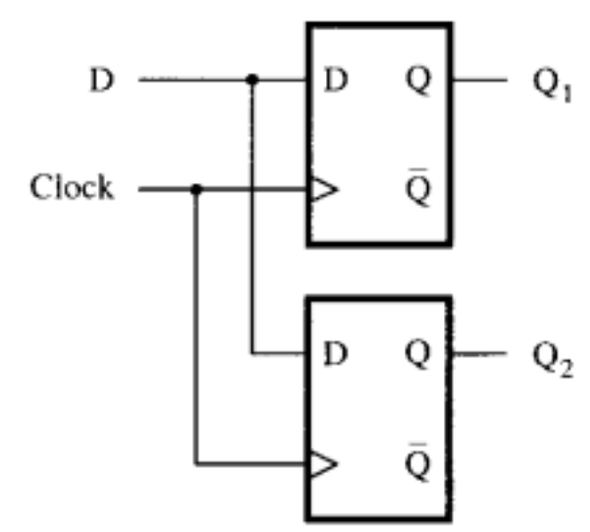


图7-38 例7.3的电路 ■

Verilog也提供非阻塞赋值，用 < = 符号表示。**always**块中所有非阻塞赋值语句在求值时所用的值全部都是进入**always**时，各个变量已具有的值。因此，某给定变量在块的所有语句中的值是相同的。非阻塞的意思是每条赋值语句的结果直到**always**块的结尾才能看到。

例7.4 图7-39给出了与图7-37中一样的代码，但用的是非阻塞赋值语句。在以下两条语句中：

```
Q1< = D;
Q2< = Q1;
```

在**always**块中的语句赋值开始时，两条语句中的变量Q1和Q2都有各自的值，然后在**always**块结束时，这两个变量同时变成各自的新值，Q1变成D，而Q2变成刚进入**always**块时的Q1值。因此这段代码生成了触发器级联连接的电路，具体实现为图7-40所示的移位寄存器。

```
module example7_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end

endmodule
```

图7-39 级联触发器的代码

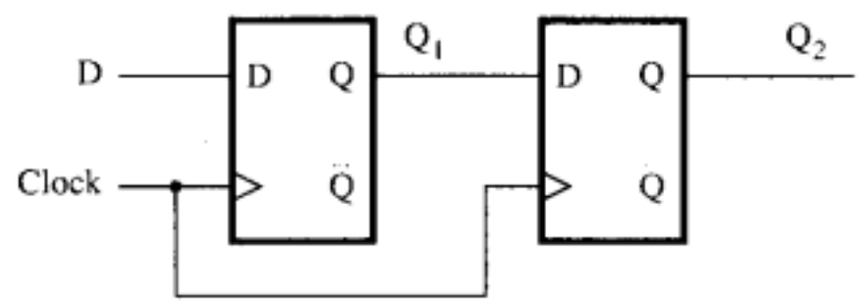


图7-40 由图7-39定义的电路

阻塞赋值和非阻塞赋值之间的区别用下面两个例子进一步加以说明。

例7.5 在图7-41中用阻塞赋值定义一段代码，该段代码除了触发器外还涉及一些门。图7-42所示为具体的实现电路。*f* 和*g*都是D触发器的输出，这是因为**always**块中的敏感列表是用时钟正跳变沿指定的。由于用的是阻塞赋值，所以由声明语句f = x1 & x2产生的新的*f* 值马上可以用在接下来的语句g = f | x3中。因此，产生x1 & x2的与门连向或门输入端，或门的输出连接到触发器g的数据端，如图7-42所示。

```
module example7_5 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output reg f, g;

   always @(posedge Clock)
   begin
      f = x1 & x2;
      g = f | x3;
   end

endmodule
```

图7-41 例7.5的代码

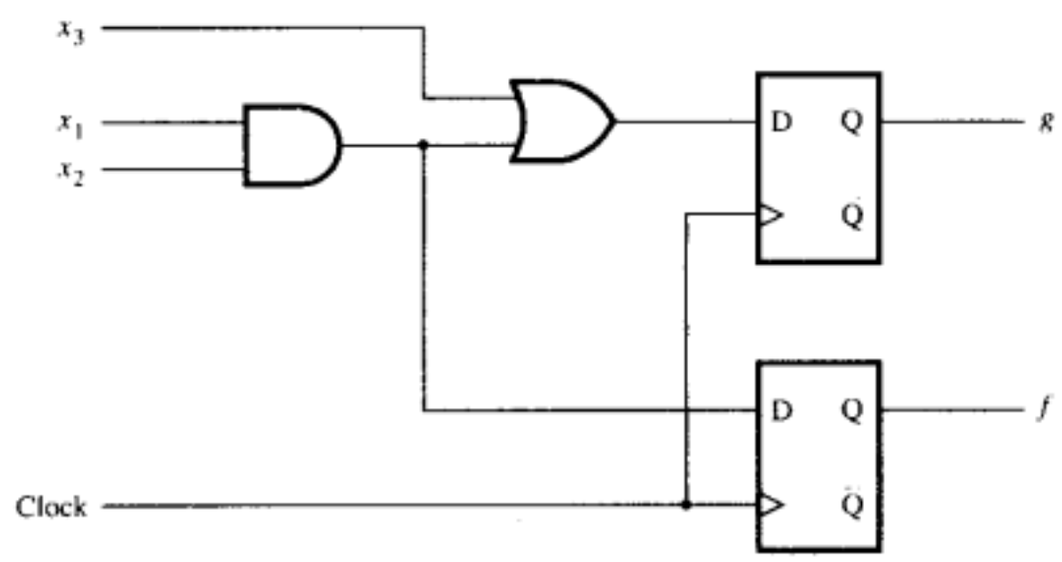


图7-42 例7.5的电路

例7.6 若用非阻塞赋值，如图7-43所示，*f* 和*g*的值被同时更新。因此，前一个*f* 值用来更新*g*的值，这意味着产生*f* 的触发器的输出端被连接到或门的输入，该或门的输出再连接到触发器*g* 的数据输入端，具体电路如图7-44所示。

```
module example7_6 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output reg f, g;

   always @(posedge Clock)
   begin
      f <= x1 & x2;
      g <= f | x3;
   end

endmodule
```

图7-43 例7.6的代码

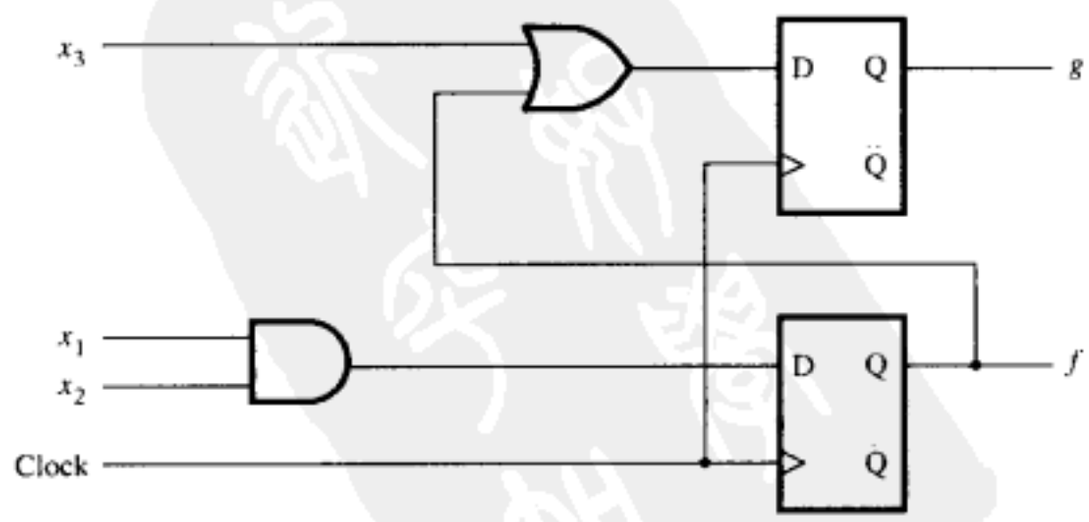


图7-44 例7.6的电路

若把给f 和g 赋值的两条语句次序颠倒一下，则会综合成什么样的电路呢？思考这个问题很有趣。对于图7-41中的代码，影响会比较大。若g 先求值，则第二个声明语句不依赖于第一个，因为f 不依赖于g。综合后生成的电路会同图7-44中的电路一样。语句顺序颠倒对图7-43所示的非阻塞赋值的代码则没有任何影响。

用阻塞赋值描述时序电路很容易生成错误的电路，如图7-38所示。阻塞赋值语句对语句顺序的依赖可能综合成错误的电路，因而是有风险的，如前面的例子所示。所以，最好用非阻塞赋值来描述时序电路。

7.12.4 组合逻辑电路的非阻塞赋值

到目前为止，我们很自然地会想到这样的问题：非阻塞赋值是否可以用于描述组合逻辑电路。答案是在大多数情况下可以用，但是当**always**块中后面的赋值语句依赖于前面赋值语句的结果时，非阻塞赋值会产生无意义的电路。例如，假设我们有一个三位的变量$A = a_2 a_1 a_0$，我们希望产生一个组合逻辑函数f，当A中相邻两位为1时，f 就等于1。用阻塞赋值描述这个函数的一种方法如下：

```
always @ (A)
begin
   f = A[1] & A[0];
   f = f | (A[2]&A[1]);
end
```

这些声明语句实现了想要的逻辑函数，就是$f = a_1 a_0 + a_2 a_1$。现在考虑用非阻塞赋值将代码改为：

```
f <= A[1]&A[0];
f <= f |(A[2]&A[1]);
```

对应于这段代码，Verilog语义方面有两点是很关键的：

1) 非阻塞赋值语句的结果仅在**always**块中所有语句求值结束后才可以看到。

2) 当**always**块中同一变量多次赋值后，只保留最后一次赋值的结果。

在这个例子中，在我们进入**always**块时f 有一个未说明的初始值。第一条语句赋值$f = a_1 a_0$，但是这个结果对于第二条语句是不可见的，它只能看到原始的还未赋值的f 值。所以第二条语句越过（删除）了第一条语句，产生逻辑函数$f = f + a_2 a_1$。这个表达式同组合逻辑电路不一致，因为它表示的虽然是一个与-或逻辑电路，但或门的输出却反馈回到它自身的输入（译者注：因此该电路就不再是组合逻辑了）。为避免产生不想要的时序电路，描写组合逻辑电路时最好使用阻塞赋值。

7.12.5 具有清零功能的触发器

用一个特殊的敏感列表和**if-else**顺序语句的特殊形式，可以产生具有清零（或置1）信号的触发器。

例7.7（异步清零） 图7-45所示是一个具有低电平有效的异步复位（清零）输入端的D触发器模块。当复位输入*Resetn*等于0时，触发器的输出Q复位为0。注意敏感列表里说明*Resetn*的负跳变沿与时钟的正跳变沿都可以触发事件。我们不能忽略关键字**negedge**，因为敏感列表里不能既包括沿触发信号又包括电平敏感型信号。 ■

例7.8（同步清零） 图7-46所示的代码展示如何描述一个具有同步复位输入的D触发器。在这个例子中，复位信号只在时钟正跳变沿时起作用。代码生成的电路如图7-14c所示，其中包含一个连向触发器D输入端的与门。 ■

```
module flipflop (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output reg Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

图7-45 含异步复位端的D触发器

```
module flipflop (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output reg Q;

   always @(posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

图7-46 含同步复位端的D触发器

7.13 用CAD工具在设计中加入寄存器和计数器

本节将介绍如何用CAD工具在设计的电路中添加寄存器和计数器。我们将通过举例说明如何用原理图输入和编写Verilog代码这两种方法实现此目的。

7.13.1 在电路原理图中添加寄存器和计数器

在5.5.1节中，我们解释了CAD系统通常包含有一些现成的子电路库。我们曾以加法器/减法器（lpm_add_sub）模块为例，介绍了参数化模块库（LPM）。LPM包括了可以构成触发器、寄存器、计数器以及其他许多有用电路的模块。图7-47展示了表示lpm_ff 模块的符号。该模块是一个寄存器，由一个或者多个正跳变沿触发的触发器（可以是D触发器或T触发器）构成。该模块包含有可以设置触发器数目和类型的参数。在本例中，我们选四个D触发器。附录C中的自学教程解释了如何配置该模块。

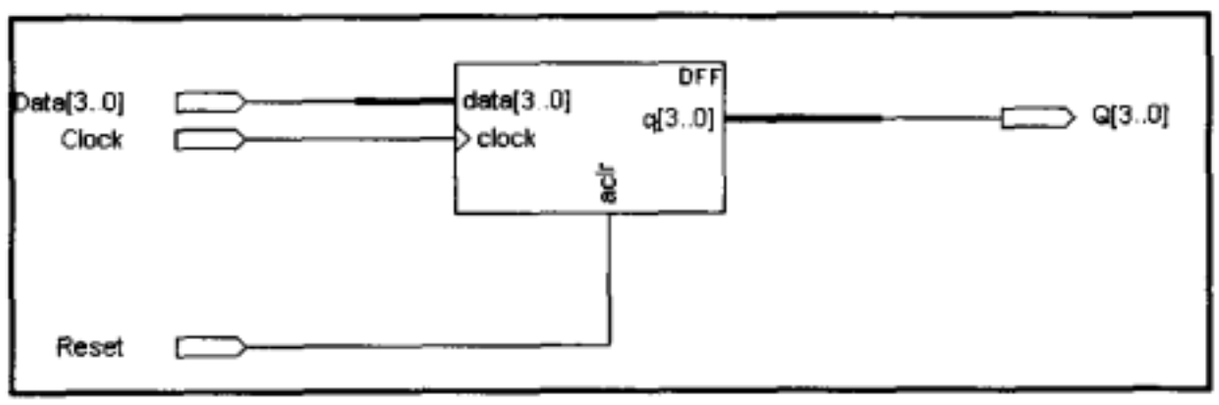


图7-47 lpm_ff参数化触发器模块

四个触发器的输入端*D*，在图形符号上用*data*[3..0]标记，与四位的输入信号*Data*[3..0]相连接。在电路原理图中，模块的异步复位（即清零）输入端（高电平有效），用*aclr*标记。触发器的输出*q*，与标有*Q*[3..0]的输出符号相连接。

在7.3节中，我们曾讲过D触发器的一种有用的应用就是保持算术计算的结果，例如，保存加法器电路的输出。图7-48给出了一个例子，当中用了两个LPM模块，lpm_add_sub和lpm_ff。我们曾经在5.5.1节中描述过lpm_add_sub模块。其参数（图7-48中没有显示）设置将模块配置为一个四位的加法器电路。加法器的四位数据输入*dataa*由*Data*[3..0]信号驱动。所求得的和位*result*，连向lpm_ff模块的*data*输入端，lpm_ff被设置成具有异步清零端的四位D寄存器。寄存器产生电路的输出*Q*[3..0]（图的最左端）。这个信号反馈到加法器的另一个输入端

datab。为方便在下面的讨论中引用，加法器的和位也作为电路的输出*Sum*[3..0]。若起初寄存器清零为0000，且每个时钟周期都有一个新的数施加到输入端，则电路可以将*Data*[3..0]输入端的二进制数和寄存器累积的和相加。实现这种功能的电路叫做累加器电路。

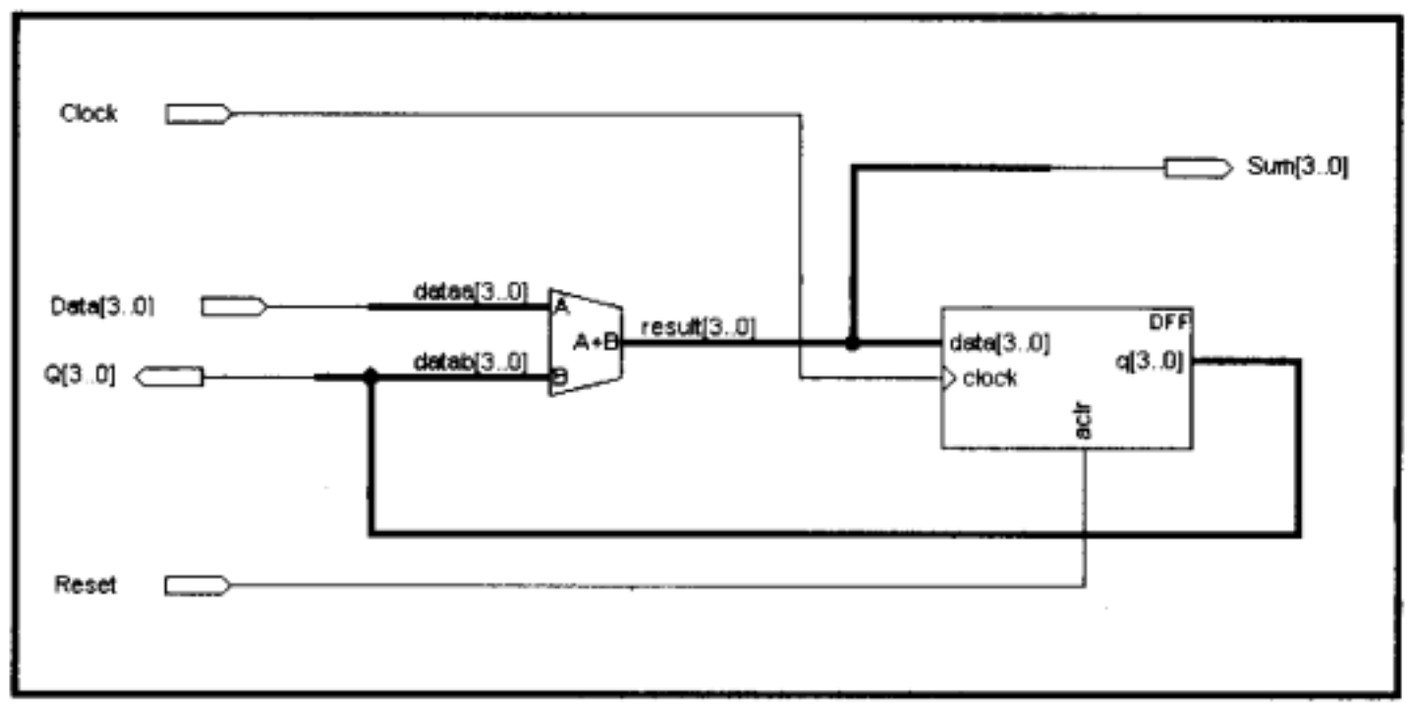


图7-48 有寄存反馈的加法器

我们从原理图综合出电路并用超前进位结构实现了该四位加法器。图7-49所示为该电路的时序仿真波形。电路复位后，*Data*输入端被置为0001。加法器产生计算得到的和值：0000+0001 = 0001，然后在60ns时刻及时地由时钟沿触发存入寄存器。经过t_{co}延迟后，*Q*[3..0]变成0001，这使加法器产生新的计算结果：0001 + 0001 = 0010。产生新的计算结果所需要的时间取决于加法器电路的速度，在本例的情况下，需要12.5ns才能产生运算结果。新的累加结果直到下次时钟正跳变沿之后，在100ns时刻才出现在Q端。然后加法器产生0011作为下一个累加结果。当*Sum*从0010变向0011时，时序图中出现了一些振荡，它们是由于加法器电路的传播延时引起的。这些振荡不会在Q输出端看到，因为*Sum*在下一时钟上升沿到来时是稳定的。在时间坐标上往前移到180ns时刻，*Sum* = 0100，该值被存入寄存器。加法器产生新的累加值0101。然后在200ns时*Data*变成0010，从而累加值变成：0100+0010 = 0110。在下一时钟的正跳变沿，*Q*被置为0110；电路中暂时出现的累加值*Sum* = 0101在*Q*输出端不会看到。在每一个连续时钟的正跳变沿，电路继续把0010同*Q*输出端相加。

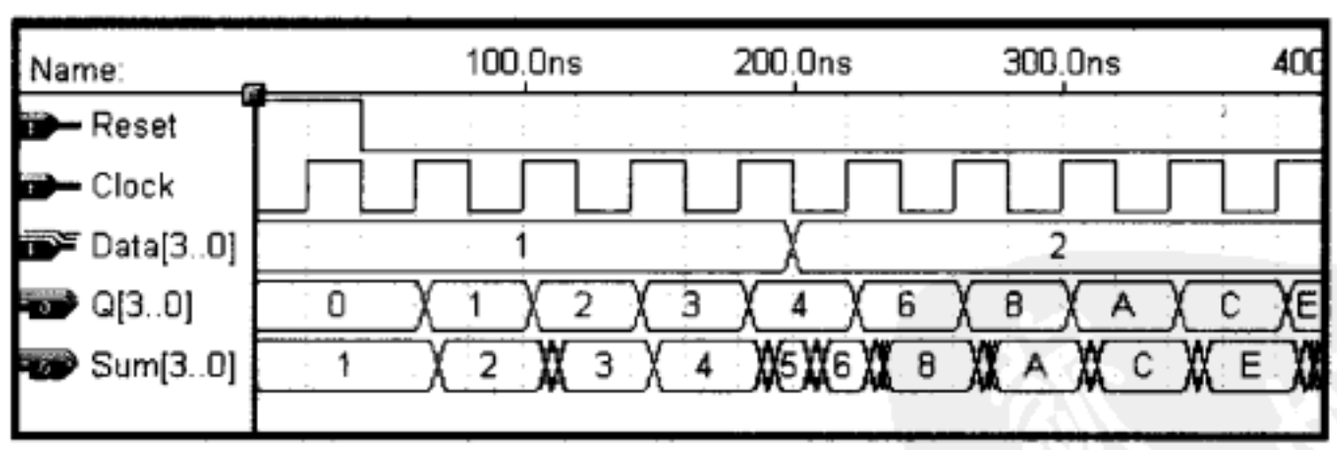


图7-49 图7-48电路的时序仿真

对该电路的行为进行仿真后，我们应该考虑是否可以得出该电路在某种程度上能正常工作的结论。理想情况下，应该对电路输入信号的所有可能组合进行全面的测试，没有发现任何问题后，再宣布该电路能按照要求正常地工作，这样做才是谨慎的。然而，实际工作中，这种测试通常不可行，因为存在的输入组合数目很大。至于图7-48中的电路，我们可以验证加法器产生的计算结果是正确的，也可以检查寄存器中的每个触发器是否能正常地存储0或者1。我们将在第11章中讨论与电路测试有关的问题。

为使图7-48中的电路正常工作，必须满足以下的时序约束。若寄存器是由时钟正跳变沿触发的，则寄存器输出端信号值的改变必须通过连接加法器输入端*datab*的反馈路径传递。然后，加法器产生一个新的和，这个和必须传向寄存器的输入端*data*。对于用来实现电路的芯片，需要的总延迟为14ns。这段延迟时间可以分解成以下几部分：从寄存器被时钟触发到其输出端的变化到达加法器的输入端*datab*耗时2ns。加法器产生新的计算结果需要8ns，计算结果传到寄存器的*data*输入端需要4ns。在图7-49中，时钟周期为40ns。因此，在新的计算结果到达寄存器的输入端*data*之后，在下一时钟上升沿到来之前还有40 − 14 = 26ns。*data*输入端在时钟沿到达之前的建立时间（t_{su} = 3ns）期间，必须保持稳定。因此，我们有26 − 3 = 23ns的保持时间可以用。要保证电路能正常运行，时钟周期最多还可以减少23ns。但是，若时钟周期比40 − 23 = 17ns还少，则电路就不能正常运行。CAD系统提供的工具可以自动确定电路正常工作必须的最小时钟周期。附录D中的教程说明了如何用与本书配套的工具实现最小时钟周期分析。

7.13.2 在Verilog代码中使用库模块

在Verilog代码中可以实例引用模块库（例如LPM库）中已定义的现成子电路。图7-50所示的代码中实例引用了lpm_shiftreg模块，这是一个*n*位的移位寄存器。模块的参数用defparam语句设置。移位寄存器的触发器个数为4，由设置参数lpm_width = 4实现。该模块还可以配置为左移或者右移，由设置参数lam_direction = “RIGHT”实现从左至右的移位。该代码模块使用高电平有效的异步清零*aclr*，高电平有效的并行加载*load*（允许移位寄存器数据输入端通过*data*加载并行数据*R*）。当进行移位时，*shiftin*输入端的值移向最低位的触发器，溢出的那一位出现在*q*并行输出端的最高位。代码用名字关联的方法把移位模块的输入和输出信号同模块的端口相连，例如，输入信号*R*与模块的*data*端口连接，在实例引用语句中用 .data（R）说明。同理，.aclr（Reset）说明输入信号*Reset*与模块的*aclr*端口连接，等等。当转换成电路时，lpm_shiftreg产生的电路结构如图7-19所示。

```
module shift (Clock, Reset, w, Load, R, Q);
   input Clock, Reset, w, Load;
   input [3:0] R;
   output [3:0] Q ;

   lpm_shiftreg shift_right (.data(R), .aclr(Reset), .clock(Clock),
      .load(Load), .shiftin(w), .q(Q)) ;
      defparam shift_right.lpm_width = 4;
      defparam shift_right.lpm_direction = "RIGHT";

endmodule
```

图7-50 lpm_shiftreg模块的实例化

逻辑电路中常用到的多种类型的计数器，都可以在参数化模块库中已定义的子电路模块中找到。例如，lpm_counter模块是可并行加载的宽度可变的计数器。

7.13.3 在Verilog代码中使用寄存器和计数器结构

除了实例引用已定义的子电路实现寄存器、移位寄存器、计数器等类似的电路之外，这些电路也可以用Verilog代码描写。图7-45所示是D触发器的代码。描写*n*位寄存器的一种方法是写一个包括*n*个D触发器子电路实例的层次化代码。更简单的途径是用与图7-45中同样的代码实现，只需把*D*输入和*Q*输出定义为多位信号即可。

例7.9（*n*位寄存器） 因为逻辑电路中通常需要各种位宽的寄存器，定义寄存器模块让它的触发器数目可以很容易地改变，这是非常有用的。图7-51中给出了*n*位寄存器的代码。参数*n*说明了寄存器中触发器的数目。通过修改参数，该代码可以代表任何位宽的寄存器。 ■

例7.10（一个四位移位寄存器） 假设我们想编写Verilog代码来表示图7-19所示的四位并行存取的移位寄存器。一种途径是编写层次化代码，该代码引用四个实例子电路。每个实例

引用的子电路由一个2选1多路器和D触发器组成，多路器的输出与触发器的输入D相连接。图7-52定义了一个名为muxdff 的模块，该模块表示的就是该子电路。两个数据输入端分别命名为D_0和D_1，由选择信号Sel控制存入触发器的为哪个信号。**if-else**语句说明在时钟正跳变沿时，若$Sel = 0$，则Q被赋值为D_0，否则，Q被赋值为D_1。■

```
module regn (D, Clock, Resetn, Q);
   parameter n = 16;
   input [n-1:0] D;
   input Clock, Resetn;
   output reg [n-1:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

图7-51 有异步清零端的n位寄存器的代码

```
module muxdff (D0, D1, Sel, Clock, Q);
   input D0, D1, Sel, Clock;
   output reg Q;

   always @(posedge Clock)
      if (!Sel)
         Q <= D0;
      else
         Q <= D1;

endmodule
```

图7-52 D输入端有一2选1多路器的D触发器代码

图7-53定义了四位移位寄存器。$Stage3$实例引用了muxdff模块，为最高位触发器（输出端Q_3），$Stage0$也同样实例引用了muxdff模块，为最低位的触发器（输出端Q_0）。当$L = 1$时，寄存器并行加载R输入端的数据；$L = 0$时，寄存器从左往右移位。串行数据从w输入端移入最高位Q_3。

例7.11（四位移位寄存器的另一种代码） 图7-54所列出的是四位移位寄存器另一种风格的代码。移位寄存器用例7.4给出的方法定义，而不是用子电路的方法。所有行为发生在时钟上升沿。若$L = 1$，寄存器并行载入R输入端的四位数据。若$L = 0$，寄存器的内容向右移位，并且w输入端的值加载到最高位Q_3。■

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output wire [3:0] Q;

   muxdff Stage3 (w, R[3], L, Clock, Q[3]);
   muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
   muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
   muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

图7-53 四位移位寄存器的层次化代码

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output reg [3:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] <= Q[1];
         Q[1] <= Q[2];
         Q[2] <= Q[3];
         Q[3] <= w;
      end

endmodule
```

图7-54 四位移位寄存器的另一种代码

例7.12（一个n位移位寄存器） 图7-55列出了可以用来表示任意位宽的移位寄存器的代码。其中参数n（默认设为16）用来设置触发器的个数。这段代码与图7-54所示的代码基本相同，不同之处有两点：第一，R和Q的位宽用n来定义；第二，描述移位操作的**else**子句用**for**循环语句来实现，可适用于由任意多个触发器组成的移位操作。■

例7.13（递增计数器） 图7-56表示具有复位输入*Resetn*和使能输入*E*的四位递增计数器。计数器中触发器的输出用名为Q的向量表示。**if**语句指定若*Resetn* = 0，则计数器异步复位。**else if** 子句说明若*E* = 1，在时钟正跳变沿时刻，计数增加。 ■

```
module shiftn (R, L, w, Clock, Q);
   parameter n = 16;
   input [n-1:0] R;
   input L, w, Clock;
   output reg [n-1:0] Q;
   integer k;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         for (k = 0; k < n-1; k = k+1)
            Q[k] <= Q[k+1];
         Q[n-1] <= w;
      end

endmodule
```

图7-55 一个*n*位移位寄存器的代码

```
module upcount (Resetn, Clock, E, Q);
   input Resetn, Clock, E;
   output reg [3:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q <= 0;
      else if (E)
         Q <= Q + 1;

endmodule
```

图7-56 四位递增计数器的代码

例7.14（有并行载入端的递增计数器） 图7-57中的代码定义了一个递增计数器，该计数器除了复位输入端，还有一个并行加载输入端。并行数据由输入向量*R*提供。第一个**if**语句与图7-56所示的代码一样实现异步复位。**else if**子句说明，若*L* = 1，则计数器中的触发器在时钟正跳变沿时从输入*R*并行加载数据；若*L* = 0，则在使能输入*E*的控制下计数器递增。 ■

例7.15（有并行载入端的递减计数器） 图7-58所列出的是名为downcount的递减计数器的代码。使用递减计数器时通常先给其加载某个初始值，然后再递减。代码中初始值用向量*R*表示。在时钟正跳变沿，若*L* = 1，则计数器加载来自于输入*R*的值，若*L* = 0，则计数器递减。计数器也有一个使能输入*E*。若*E* = 0，即使有效时钟沿到来，触发器的内容也不发生变化。 ■

```
module upcount (R, Resetn, Clock, E, L, Q);
   input [3:0] R;
   input Resetn, Clock, E, L;
   output reg [3:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q <= 0;
      else if (L)
         Q <= R;
      else if (E)
         Q <= Q + 1;

endmodule
```

图7-57 有并行载入端的四位递增计数器

```
module downcount (R, Clock, E, L, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Clock, L, E;
   output reg [n-1:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else if (E)
         Q <= Q - 1;

endmodule
```

图7-58 有并行载入端的递减计数器

例7.16（递增/递减计数器） 图7-59给出了递增/递减计数器的Verilog代码。该代码结合了图7-57和图7-58中所定义的计数器的功能。该计数器有一个控制信号*up_down*，该信号可以用来控制计数的方向（递增或递减）。它还有一个整型变量*direction*，递增计数时为1，递减计数时为－1。

```
module updowncount (R, Clock, L, E, up_down, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Clock, L, E, up_down;
   output reg [n-1:0] Q;
   integer direction;

   always @(posedge Clock)
   begin
      if (up_down)
         direction = 1;
      else
         direction = -1;
      if (L)
         Q <= R;
      else if (E)
         Q <= Q + direction;
   end

endmodule
```

图7-59 递增/递减计数器的代码

7.14 设计举例

本节列举了几个数字系统的范例，这些例子使用了本章和第6章描述的一些构件块。

7.14.1 总线结构

数字系统通常包含一组用来存储数据的寄存器。图7-60所示的数字系统有k个n位的寄存器（从$R1$到Rk）。每个寄存器都连接到由n条连线组成的一条公共线，数据传进或者输出寄存器都必须经由这条公共线。这条公共线组通常称为总线（bus）。在实际系统中除了寄存器外，还有许多其他类型的电路块与总线相连。图7-60展示了如何使用输入控制信号*Extern*将来自于其他电路块的n位数据放置到总线上。通过总线，存储在任意一个寄存器上的数据也能传送到与该总线相连接的其他寄存器或者电路块。

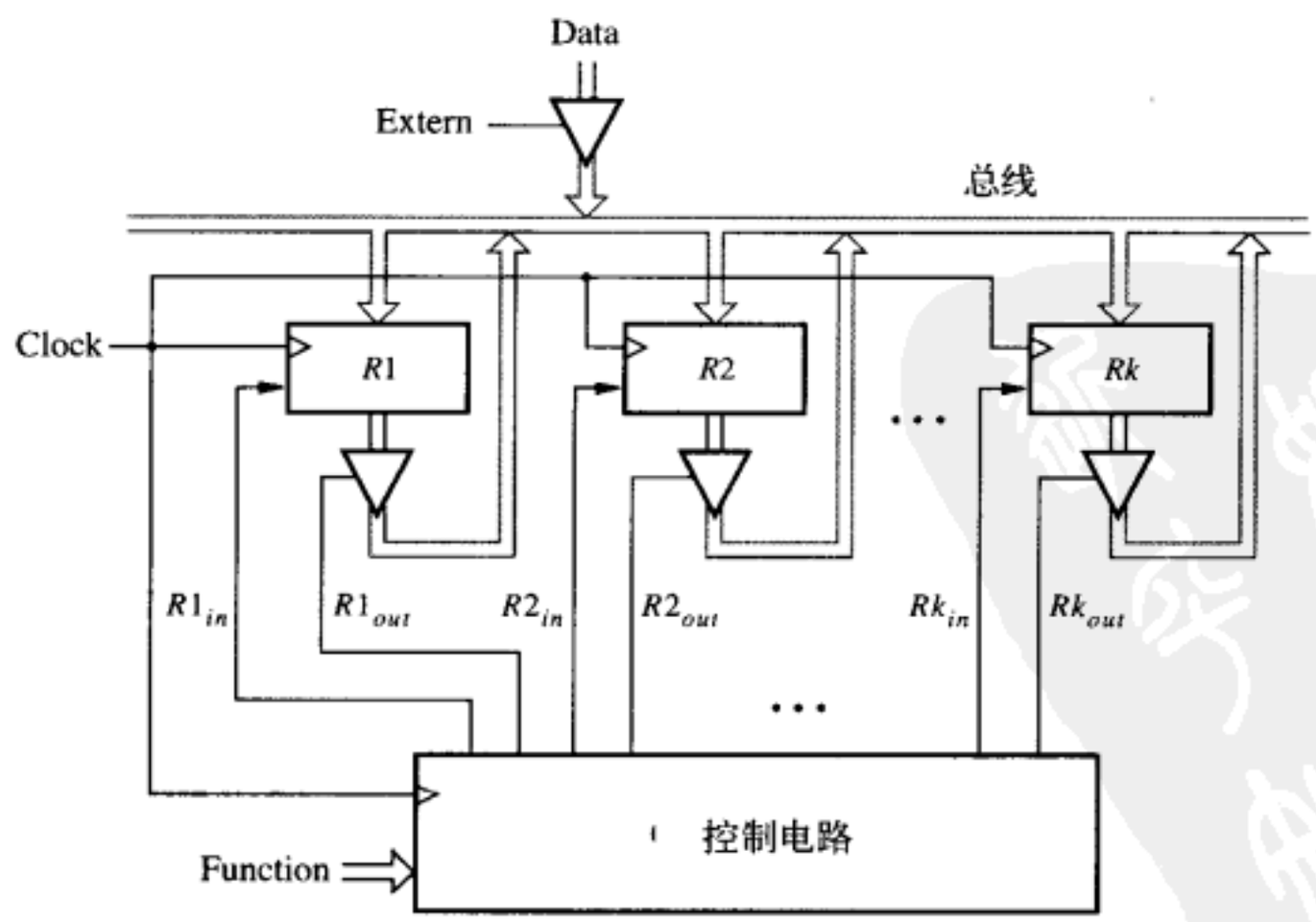


图7-60 含有k个寄存器的数字系统

最重要的是必须保证在每一给定时刻只允许一个电路块把自己的数据放置到总线上。在图7-60中，每个寄存器都通过 n 位的三态缓冲器连接到总线上。必须用一个控制电路确保在任何给定的时刻，三态缓冲器的使能输入信号（$R1_{out}$, …, Rk_{out}）中只有一个是有效的，从而保证只有一个寄存器的输出可以放在总线上。控制电路还产生$R1_{in}$, …, Rk_{in} 信号，用来控制将数据加载到每个寄存器。总的来说，控制电路可以完成许多功能，例如把存储在一个寄存器中的数据传送到另一个寄存器，以及其他一些操作。图7-60展示了一个名为*Function*的输入信号，该信号指示控制电路完成特定的任务。该控制电路与输入的时钟信号是同步的，控制k个寄存器的时钟信号与输入的时钟信号是同一个信号。

图7-60中寄存器如何与总线连接图解不够详细。为了使示意图更清晰易懂，请看图7-61，图中只画出2个两位的寄存器，位数更多的寄存器读者可以根据理解自己想象出来。对寄存器$R1$，用了两个三态缓冲器将每个触发器的输出连接到总线上，而三态缓冲器的使能控制信号为$R1_{out}$。

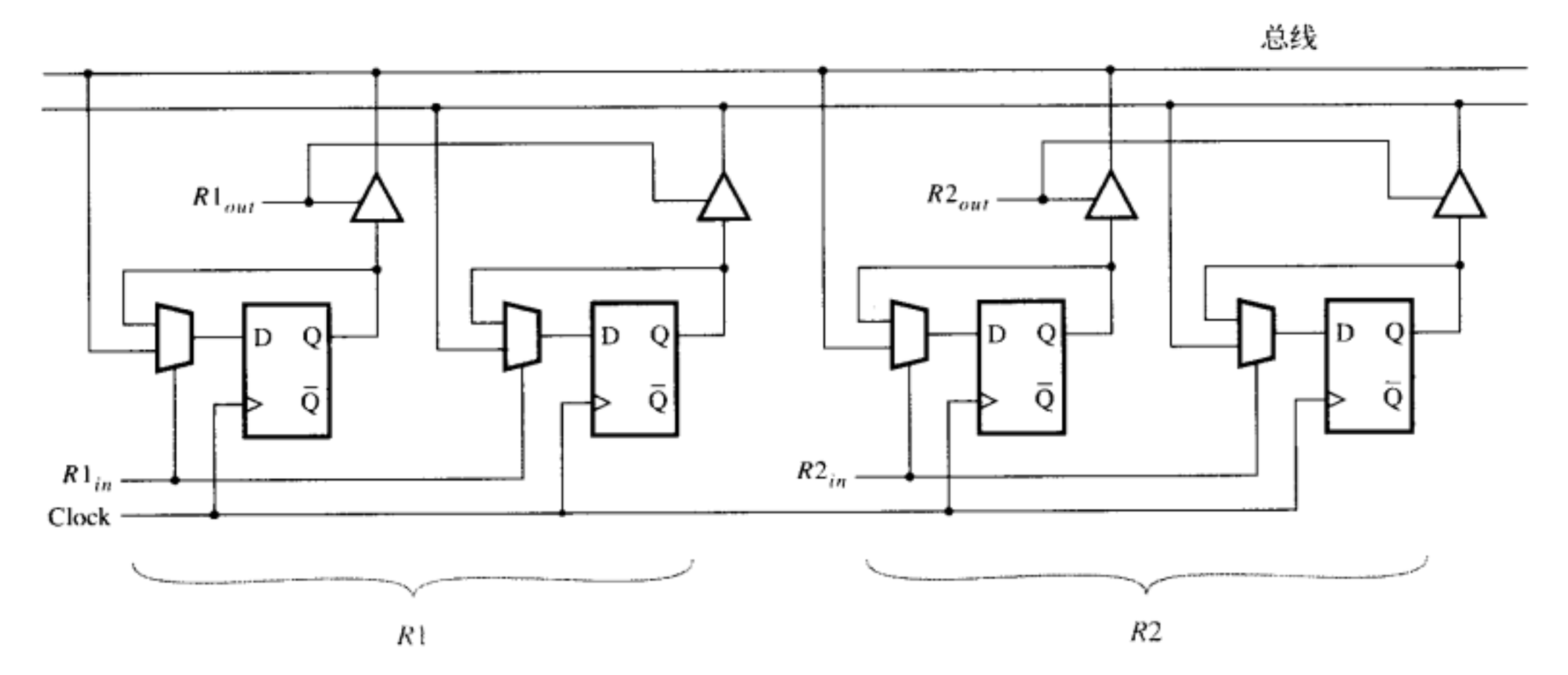


图7-61 寄存器与总线连接的细节图

每个触发器的输入D都与到2选1多路器的输出连接，多路器的选择控制信号为$R1_{in}$。若$R1_{in} = 0$，则触发器从其输出Q加载数据，所以存储的数据值不变。而若$R1_{in} = 1$，则触发器从总线加载数据。若不想使用多路器的输出，还可以把触发器的输入直接与总线连接。此时有必要控制输入所有触发器的时钟信号，以确保只在需要加载新数据时，才给该寄存器发出时钟信号。这并不是一种好的解决方案，因为不同的触发器所需要的触发时钟信号将在不同时刻到达，这将导致所谓的*时钟偏移*（clock skew）问题。关于触发器时钟偏移问题，我们将在10.3节中详细讨论。

图7-60所示的系统可以有许多种用法，具体的用法不但取决于控制电路的设计，也与总线连接的寄存器和电路块的个数多少有关。举一个简单的例子，假设一个数字系统有三个寄存器分别为$R1$、$R2$和$R3$。每个寄存器都连接到总线上，见图7-61所示。我们将设计一个控制电路，该电路只有一个功能，即用$R3$作为临时寄存器，把寄存器$R1$内的数据与$R2$内的数据进行交换。

交换过程需要分三个步骤进行，每步需要一个时钟周期。第一步，将$R2$中的数据传送到寄存器 $R3$中；接着将 $R1$中的数据传送到寄存器$R2$中；最后将 $R3$中的数据（即原来$R2$中的数据）再传送到寄存器 $R1$中。请注意，我们提到寄存器 Ri 中的数据被传送到另外一个寄存器Rj，这个行话通常用来表明寄存器Rj中的新数据将是Ri 中数据的拷贝。传送后，Ri 中的数据

没有丢失也没有发生改变。因此，更精确的说法应该是将寄存器Ri中的数据复制一份再送到寄存器Rj中。

移位寄存器用作控制

设计交换操作控制电路的方法有许多种。一种方法是利用自左向右的移位寄存器，见图7-62所示。假设复位输入信号用来对触发器清零，此时控制信号$R1_{in}$、$R1_{out}$等都还没有作用，因为此时移位寄存器的输出为0。串行输入w 通常情况下为0。我们假设输入w的值的改变与时钟信号的变化同步，略迟于时钟的有效跳变沿。该假设是合理的，因为信号w通常是由受同一个时钟控制的某电路所产生的输出。当想要进行交换操作时，w被设置为1，维持一个时钟后w变为0。在下一个有效时钟沿后，最左边那个触发器的输出变为1，于是$R2_{out} = R3_{in} = 1$。寄存器$R2$中的数据被放置到总线上，在下一个有效时钟沿时刻，该数据被加载到寄存器$R3$中。该时钟沿同时也会将寄存器中的内容移位变为010，由此产生$R1_{out} = R2_{in} = 1$。请注意，因为此刻 $w = 0$，第一个触发器被清零，使得$R2_{out} = R3_{in} = 0$。此刻寄存器$R1$中的数据被放置到总线上，在下一个有效时钟沿时刻该数据被加载到寄存器$R2$中。经过这个时钟跳变沿后，移位寄存器的内容变为001，由此产生$R3_{out} = R1_{in} = 1$。寄存器$R3$中的数据被放置到总线上，在下一个有效时钟沿时刻该数据被加载到寄存器$R1$中。

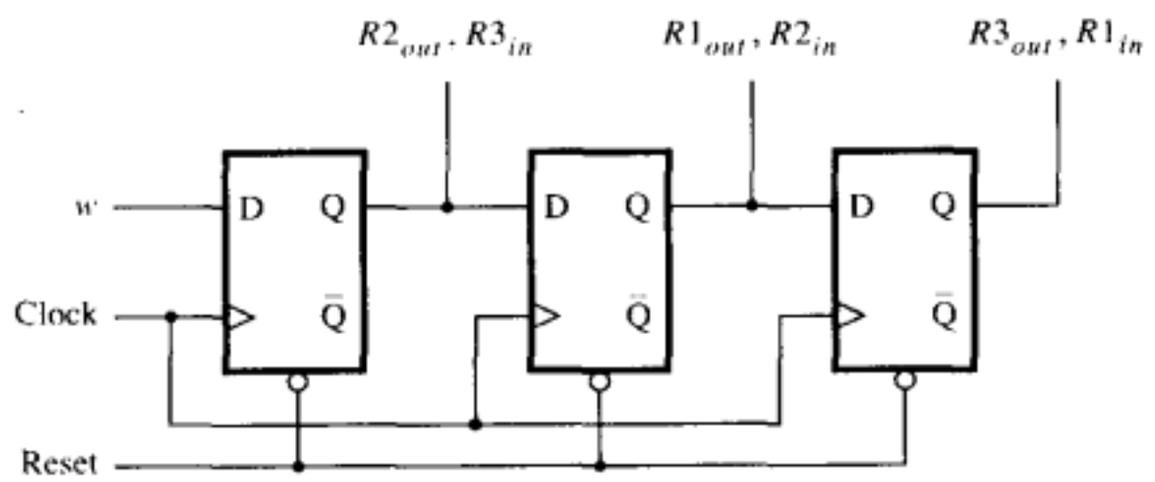


图7-62 移位寄存器实现的控制电路

用图7-62所示的控制电路，对w的响应不够快，当w变为1时，并不能立即进行交换操作，需要等到下一个有效时钟沿时才能开始。若对控制电路做一些修改，我们就能使得在w变为1的同时（即一个时钟沿）就开始进行交换操作。图7-63所示的电路说明了这种方法。复位信号用来将移位寄存器的内容复位到100，把最左边的触发器预先设置为1，将其他两个触发器清零。只要$w = 0$，就没有输出控制信号。当w变为1时，信号$R2_{out}$ 和 $R3_{in}$ 便立即变为1，于是$R2$中的数据就放到了总线上。下一个有效时钟跳变沿到达时，立即将总线上的数据加载到$R3$，并同时将移位寄存器右移一位，使其内容变为010。因为信号 $R1_{out}$ 此刻为1，则$R1$中的数据就出现在总线上。下一个时钟跳变沿到达时，立即将总线上的数据放入$R2$，并同时将移位寄存器的内容右移一位，使其内容变为001。此时$R3$中的数据便出现在总线上。下一个时钟跳变沿到达时，便将总线上的数据放入$R1$，并同时将移位寄存器右移一位，其内容变成100。我们假设$w = 1$ 只维持了一个时钟周期；因此控制电路的输出信号此刻仍然全部为0。由于我们以很特殊的方式引入图7-63所示的电路，读者可能不清楚这个电路是如何设计出来的。在8.3节，我们将用更正规的途径讲解该电路是如何设计的。

图7-63所示的电路假设最左边的触发器具有置1输入。若该触发器只有清0输入，则可以使用图7-64所示的等价电路。图7-63 与图7-64所示的电路只有很小的不同，图7-63的电路最左边触发器的输入来自于或门，输出为Q，而图7-64的电路最左边触发器的输入来自于或非门，输出为 $\overline{Q}$。

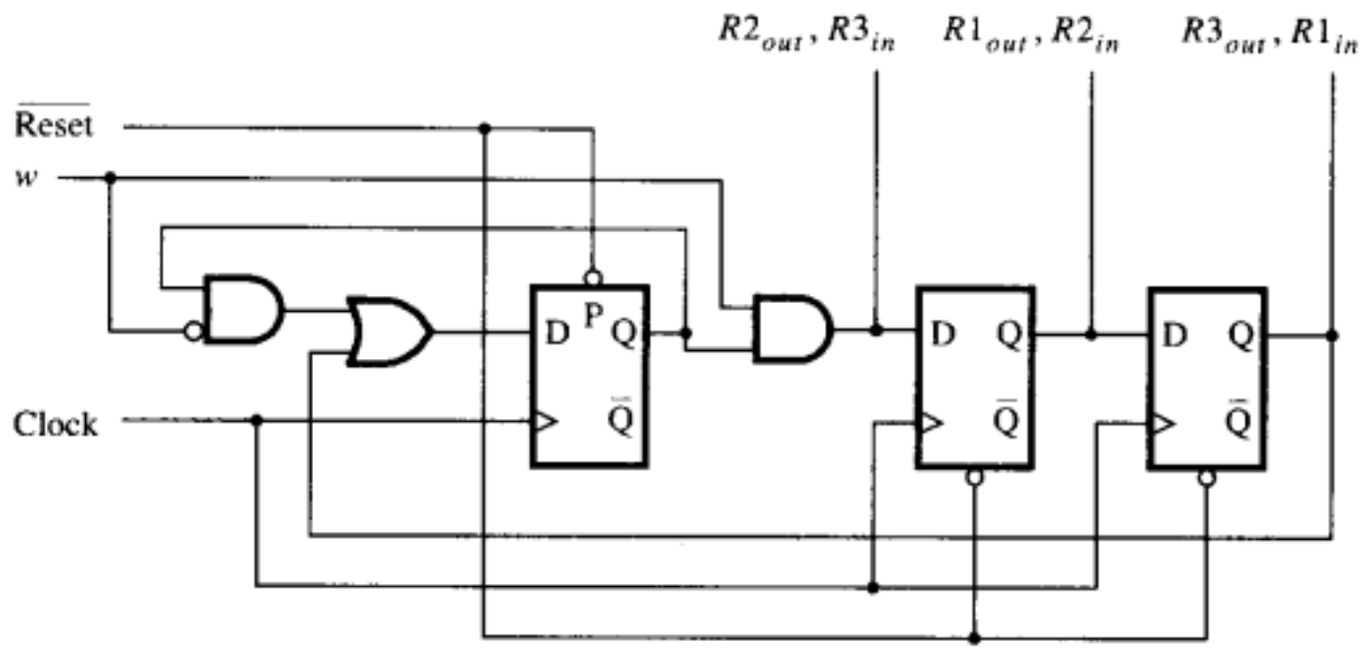


图7-63 修改后的控制电路

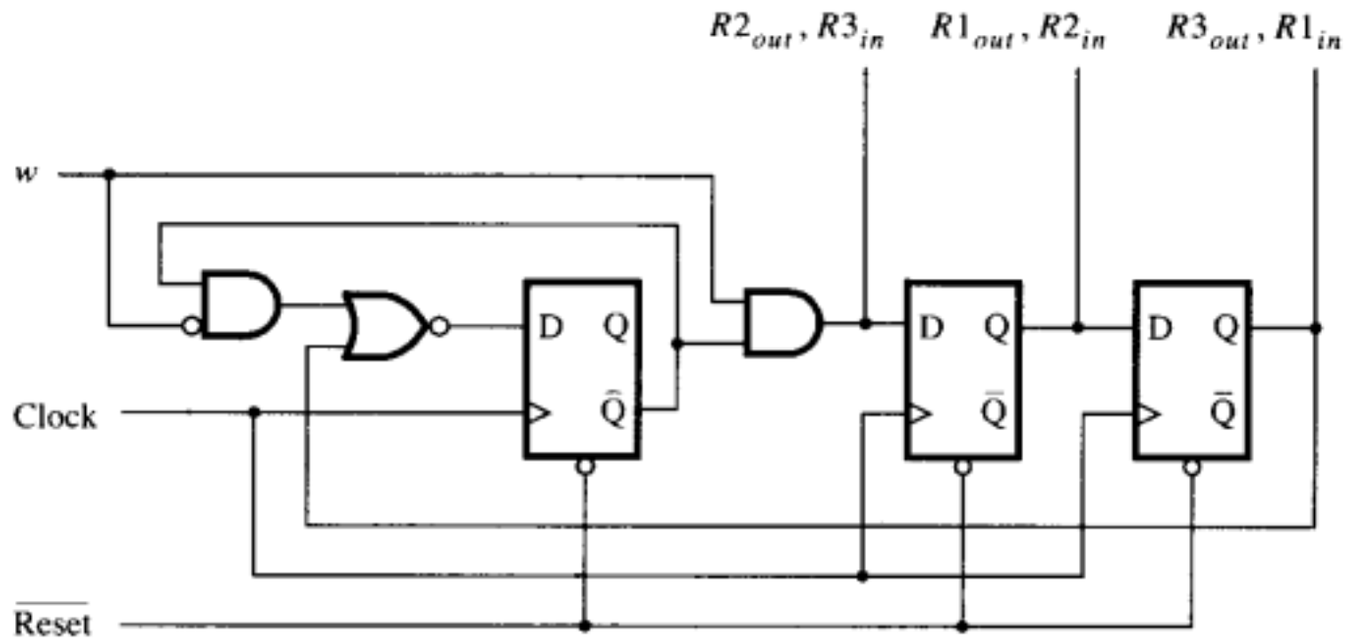


图7-64 图7-63所示电路的改进版

用多路器实现总线

在图7-60所示的电路中，我们用三态缓冲器来控制总线的访问。另一种实现总线访问的途径是使用多路器，见图7-65所示。每个寄存器的输出都连接到多路器。该多路器的输出连接到寄存器的输入，由此来实现总线。多路器的选择输入确定哪一个寄存器中的数据出现在总线上。虽然图7-65上只画出了一个多路器的符号，但实际上对于寄存器中的每一位都需要一个多路器。例如，假设有4个八位的寄存器（从$R1$到$R4$），加上由外部提供的八位数据$Data$。为了将它们连接起来，我们需要用八个5选1多路器。

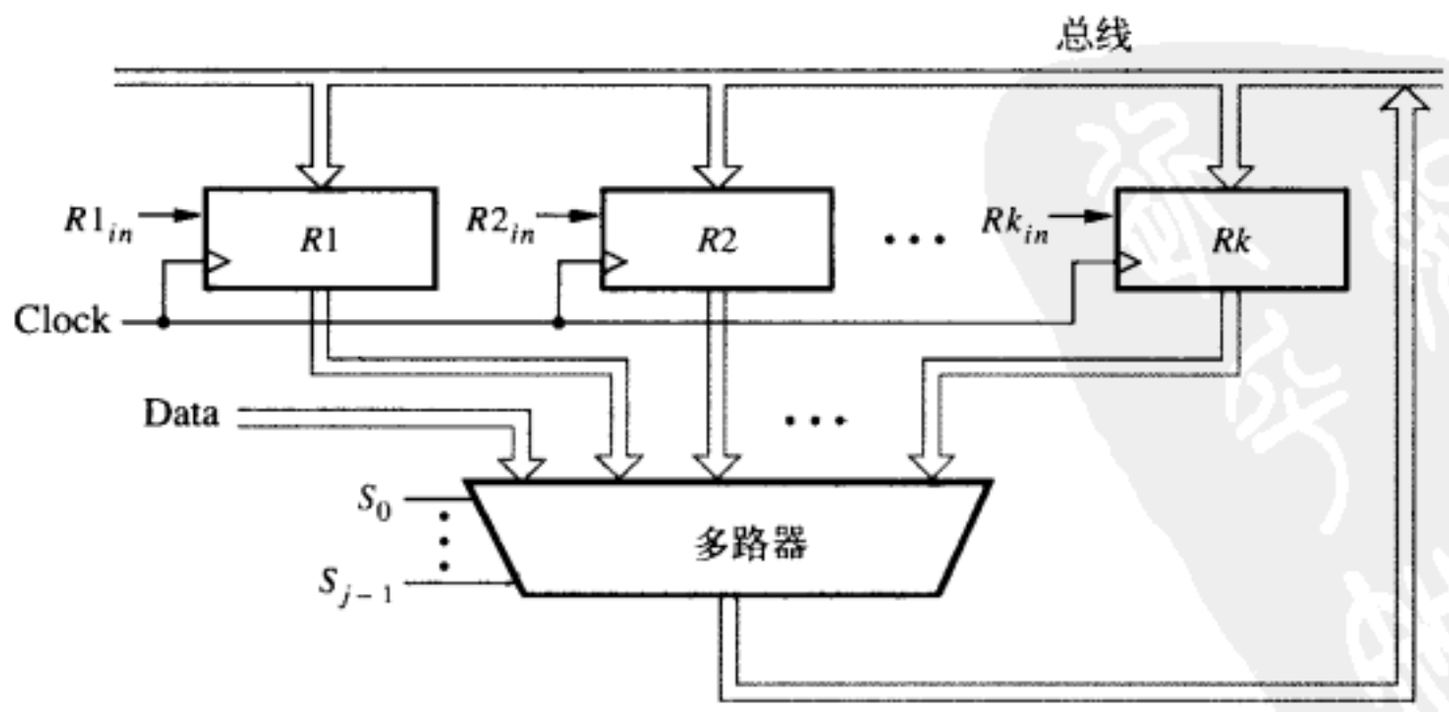


图7-65 用多路器来实现总线

在图7-62中，我们曾用一个移位寄存器来实现控制电路。类似的方法也可以用于多路器实现的总线，用来控制将数据加载到寄存器中的信号（如$R1_{in}$）仍能直接连接到移位寄存器的输出。但是不再使用如$R1_{out}$之类的控制信号将寄存器内保存的数据放置到总线上，我们必须产生多路器的选择控制信号才能将数据放置到总线上。实现该目标的一个方法是把移位寄存器的输出连接到编码器，并将编码器产生的输出作为多路器的选择控制信号。我们曾在6.3节中讨论过编码器电路。

用三态缓冲器或用多路器实现总线的方案都是合理的。但是有些类型的芯片，例如PLD，片内没有足够数量的三态缓冲器，甚至于实现中等规模总线所需要的缓冲器数量都不够。在这一类芯片中，基于多路器的解决方案是唯一切实可行的办法。在实际工作中，电路是用CAD工具设计的。若设计者用三态缓冲器来描述电路，而目标器件中这一类缓冲器的数量不够，则CAD工具会自动选用多路器来生成等价的电路。

Verilog 代码

前面介绍了实现两个寄存器之间交换存储数据的电路，本节列出了该电路的Verilog代码。我们首先给出图7-60所示电路的代码，该电路用三态缓冲器实现总线功能；接着给出图7-65所示电路的代码，该电路用多路器实现总线功能。代码是用层次化风格编写的，使用了表示寄存器、三态缓冲器和移位寄存器的子电路模块。图7-66给出了图7-61所示的n位寄存器的代码。寄存器的位数可以用参数n设置，该参数的默认值为8。该寄存器被定义为：若输入信号 $Rin = 1$，则组成寄存器的n个触发器被来自于输入信号R的n个数位加载。否则这n个触发器保持当前存储的数据不变。

```
module regn (R, Rin, Clock, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Rin, Clock;
   output reg [n-1:0] Q;

   always @(posedge Clock)
      if (Rin)
         Q <= R;

endmodule
```

图7-66 图7-61所示的n位寄存器的代码

图7-67所列出的代码表示了由n个三态缓冲器组成的子电路，每个三态缓冲器的使能控制信号都是输入信号E。输入到缓冲器中的信号是n位的信号Y，缓冲器的输出信号是n位的信号F。条件赋值语句说明：若$E = 1$，则缓冲器的输出为$F = Y$，否则缓冲器的输出为高阻z。条件赋值语句用一个未规定位数大小的数来定义高阻的情况。Verilog编译器将把这个未规定位数大小的表示高阻的值 z 设置成与向量Y的位数相等，即n。但我们不能把表示高阻的z定义为n'bz，因为Verilog语法规定不能使用参数来规定位数的大小。

```
module trin (Y, E, F);
   parameter n = 8;
   input [n-1:0] Y;
   input E;
   output wire [n-1:0] F;

   assign F = E ? Y : 'bz;

endmodule
```

图7-67 n位三态模块的代码

图7-68中的代码定义了实现图7-62所示控制电路的移位寄存器。触发器的个数由生成参数(generic parameter) m 设置，该参数的默认值为4。该移位寄存器具有低电平有效的异步复位输入信号。移位操作是由**for**循环语句定义的，采用与例7.12类似的语句风格。

图7-69所示的代码表示一个类似于图7-60所示的数字系统，该系统有3个八位的寄存器，分别为$R1$、$R2$和$R3$。图7-60所示的电路包含三态缓冲器，该缓冲器用来存放由总线提供的n位外部数据。在图7-69中，模块*tri_ext*实例引用了这几个三态缓冲器模块（trin）。八个缓冲器每个都是由输入信号*Extern*使能的，而且输入缓冲器的数据与八位数据信号*Data*连接。当$Extern = 1$时，则数据*Data*的值就被放置到总线上，在代码中用信号*BusWires*来表示总线。*BusWires*向量既表示电路的输出又表示内部总线的连线。我们声明该向量是**tri**（三态）类型的，而不是**wire**类型的。对Verilog编译器而言，对关键字**tri**和关键字**wire**的处理方法是完全一致的。命名**tri**类型，就明显地提醒读者代码综合后的电路连接将具有三态的能力。

```
module shiftr (Resetn, w, Clock, Q);
   parameter m = 4;
   input Resetn, w, Clock;
   output reg [1:m] Q;
   integer k;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
      begin
         for (k = m; k > 1 ; k = k - 1)
            Q[k] <= Q[k - 1];
         Q[1] <= w;
      end

endmodule
```

图7-68 图7-62所示的移位寄存器的代码

```
module swap (Data, Resetn, w, Clock, Extern, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock, Extern;
   input [1:3] RinExt;
   output tri [7:0] BusWires;
   wire [1:3] Rin, Rout, Q;
   wire [7:0] R1, R2, R3;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;

   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   assign Rout[1] = Q[2];
   assign Rout[2] = Q[1];
   assign Rout[3] = Q[3];

   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   trin tri_ext (Data, Extern, BusWires);
   trin tri_1 (R1, Rout[1], BusWires);
   trin tri_2 (R2, Rout[2], BusWires);
   trin tri_3 (R3, Rout[3], BusWires);

endmodule
```

图7-69 与图7-60类似的数字系统

我们假设存在一个名为*RinExt*的三位的控制信号，该信号允许从总线上将外部提供的数据加载到寄存器*R*1或*R*2或*R*3中。为了使电路原理图简单明了，在图7-60中，我们没有画出输入信号*RinExt*，但该信号将由产生*Extern*和*Data*的同一个外部电路块生成。当*RinExt*[1] = 1时，总线上的数据被加载进入寄存器*R*1；当*RinExt*[2] = 1时，总线上的数据被加载进入寄存器*R*2；当*RinExt*[3] = 1时，总线上的数据被加载进入寄存器*R*3。

在图7-69所示的代码中，我们用具体模块名*control*实例引用了三位移位寄存器模块*shiftr*。

移位寄存器的输出是三位信号Q。定义模块*shiftr*中触发器个数的参数*m*，其默认值为4。因为我们只需要实例引用一个三位的移位寄存器，所以必须改写参数*m*的值。该参数值可以用以下语句设置：

defparam control.m = 3;

defparam语句定义了表明的参数值。实例模块参数的命名规定为：实例名.参数名。在上面的例子中，实例名为*control*，参数名为*m*。

图7-69中接下去的三条语句将Q连接控制信号，这些控制信号确定数据何时加载到每个寄存器上，控制信号用三位的信号*R*in表示。代码中的信号*Rin*[1]、*Rin*[2]和 *Rin*[3]分别对应图7-60中的控制信号$R1_{in}$，$R2_{in}$和$R3_{in}$。如图7-62所示，数据什么时候加载到寄存器*R*3，由移位寄存器中最左边的输出Q[1]控制。类似地，寄存器*R*2的加载由*Q*[2]控制，寄存器*R*1的加载由*Q*[3]控制。*Rin*中的每一位与*RinExt*相应的位相或，这样外部提供的数据就能像上面讨论过的那样存储到寄存器中。代码也将移位寄存器的输出连接到三态缓冲器的使能控制信号*Rout*上。图7-62展示*Q*[1]被用来将*R*2中的数据放置到总线上；因此*Rout*[2]被赋值为*Q*[1]。类似地，*Rout*[1]被赋值为*Q*[2]，*Rout*[3]被赋值为*Q*[3]。代码中的其余语句实例引用了系统中的寄存器和三态缓冲器。

使用多路器的Verilog代码

图7-70展示了如何把图7-69所示的代码修改成使用多路器而不使用三态缓冲器。利用图

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;
   reg [1:0] S;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;
   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q, Data, R1, R2, R3, S)
   begin
      // Encoder
      if (Q == 3'b000)  S = 2'b00;
      else if (Q == 3'b100)  S = 2'b10;
      else if (Q == 3'b010)  S = 2'b01;
      else  S = 2'b11;

      // Multiplexers
      if (S == 2'b00)  BusWires = Data;
      else if (S == 2'b01)  BusWires = R1;
      else if (S == 2'b10)  BusWires = R2;
      else  BusWires = R3;
   end
endmodule
```

图7-70 用多路器实现总线

7-65所示的电路结构，总线用八个4选1多路器实现。每个4选1多路器上的三个数据输入端分别连接到寄存器 *R*1、*R*2 和 *R*3上的一位。每个4选1多路器上的第四个数据输入连接到输入信号*Data*的一位，允许外部提供的数据写入寄存器。当移位寄存器的内容是000时，多路器选择将外部输入数据*Data*放置到总线上。该数据由输入信号*RinExt*选择放入寄存器：若*RinExt*[1] = 1，则放入寄存器*R*1；若*RinExt*[2] = 1，则放入*R*2；若*RinExt*[3] = 1，则放入*R*3。

图7-69中控制三态缓冲器与总线连接的使能信号为*Rout*，在用多路器实现时，这个信号可以省略，但我们必须为多路器提供通路选择控制信号。在图7-70中，移位寄存器的输出为*Q*。这些由*Q*信号生成的控制信号*Rin*对寄存器所起的作用与图7-69中是相同的。我们在关于图7-65的讨论中曾经提到，在移位寄存器的输出和多路器的通路选择控制信号之间需要一个合适的编码器。图7-70中的第一个**if-else**语句段描述了该编码器。该段代码生成多路器的通路选择信号，在代码中命名为*S*。若移位寄存器的内容为000，则*S* = 00，若移位寄存器的内容为100，则*S* = 10，依次类推，见所列出的代码。多路器是用第二个**if-else**语句段描述的，这段语句表示：若*S* = 00，则将数据*Data*的值放置到总线（*BusWires*）上，若*S* = 01，则将寄存器*R*1中的数据放置到总线上，依次类推。用这个方案，当交换操作无效时，多路器把来自于*Data*的输入放置到总线上。

如上描述，图7-70使用了两段**if-else**语句，一段描述了编码器，另一段描述了总线多路器。更简单的写法只需要一段**if-else**语句即可，见图7-71所示代码。在这段代码中，每条语句直接指定了对应不同的移位寄存器的输出，哪个信号应该出现在总线（*BusWires*）上。由图7-70所示的代码和由图7-71所示的代码生成的电路是等价的。

图7-72所示的是由图7-71代码综合后产生电路的时序仿真波形。在前一半仿真时间里，电

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;

   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];

   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q, Data, R1, R2, R3)
   begin
      if (Q == 3'b000)  BusWires = Data;
      else if (Q == 3'b100)  BusWires = R2;
      else if (Q == 3'b010)  BusWires = R1;
      else  BusWires = R3;
   end

endmodule
```

图7-71 图7-70所示代码的简化版

路被复位，寄存器$R1$和$R2$的内容被初始化。十六进制数55被加载到$R1$，十六进制数AA被加载到$R2$。在275ns时钟跳变沿时刻，图7-72中在此时刻用垂直参照线做了标记，将$w = 1$加载到移位寄存器。然后$R2$的内容(AA)随即出现在总线上，并在325ns时钟沿时刻加载到$R3$。在该时钟沿之后，移位寄存器的内容变为010，存储在$R1$中的数据(55) 随即出现在总线上。在375ns时钟沿时刻把该数据加载到$R2$，并将移位寄存器的内容变为001。 $R3$的内容 (AA)此刻出现在总线上，在425ns时钟沿时刻被加载到$R1$，此刻移位寄存器的内容变为000，数据交换完成。

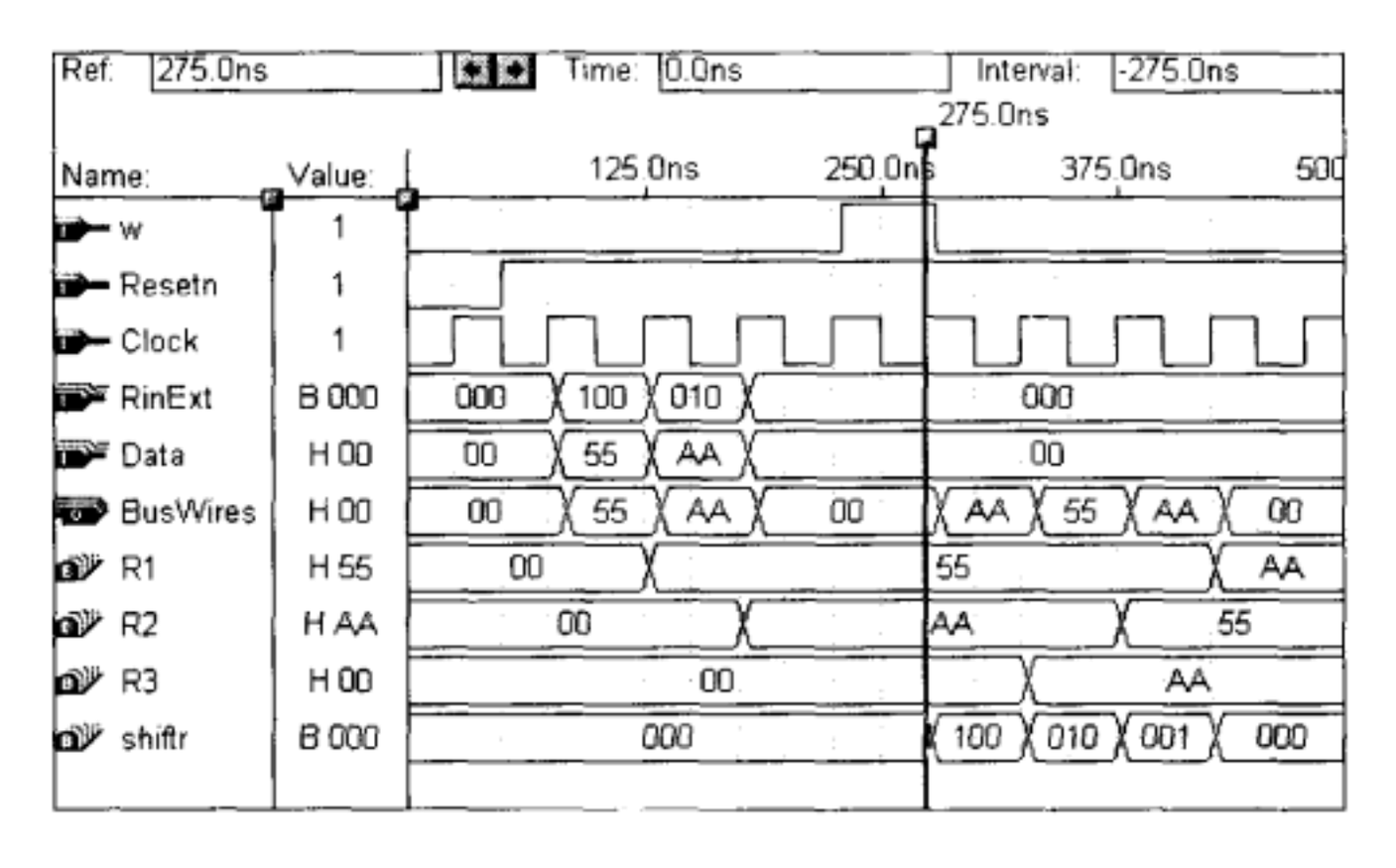


图7-72 图7-71中代码的时序仿真

7.14.2 简单的处理器

图7-73所示是类似于图7-60中的数字系统的第二个例子。该系统有四个n位的寄存器$R0,\cdots,R3$，这些寄存器通过三态缓冲器连接到总线。外部数据可以从n位数据输入$Data$加载

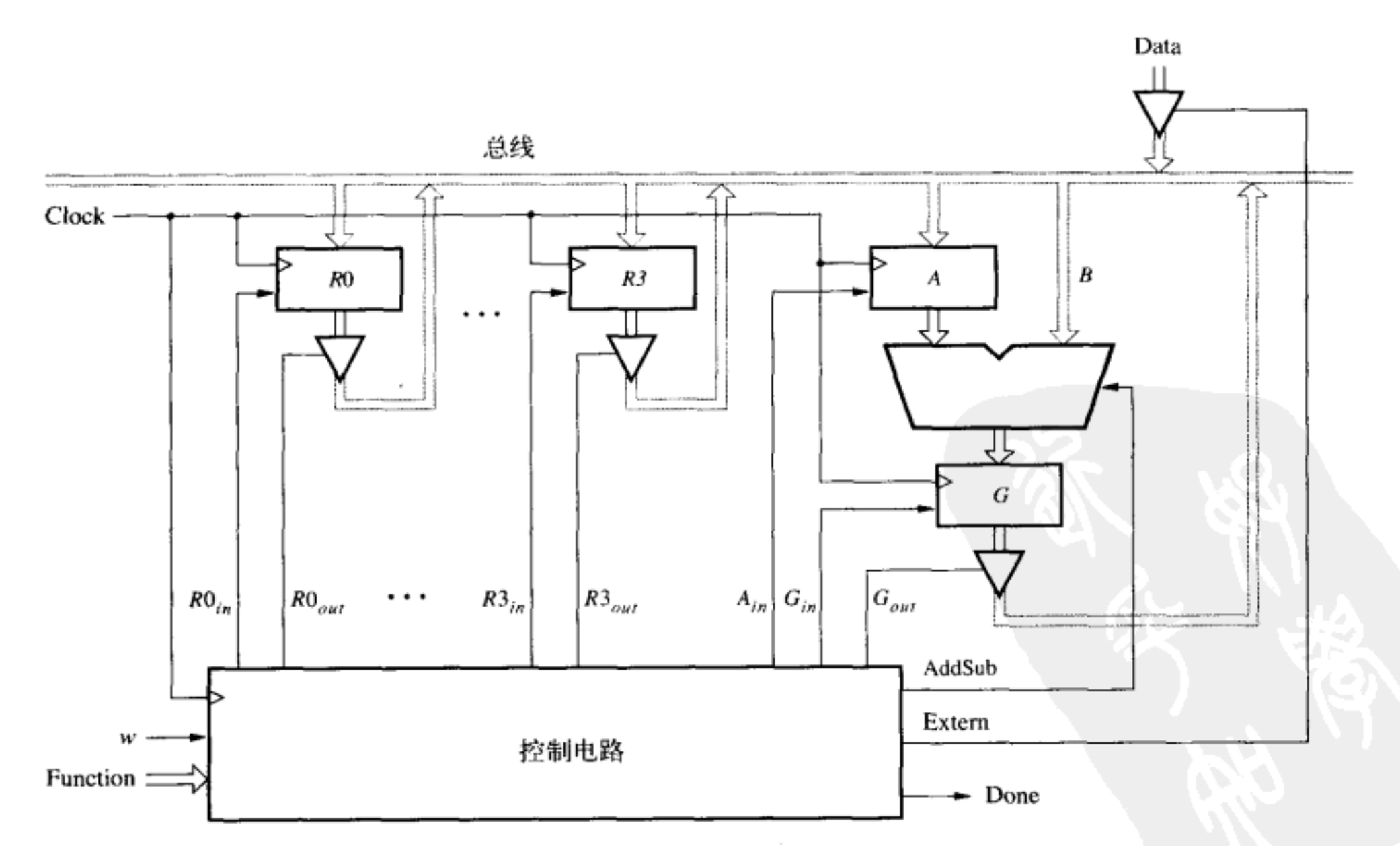


图7-73 能实现简单处理器功能的数字系统

到寄存器中，通过由*Extern*使能信号控制的三态缓冲器可以将存储在寄存器中的数据连接到总线。该系统还包含一个加法器/减法器模块。加/减法器模块的一个输入数据是由一个n位的寄存器A所提供，而另外一个输入数据B则直接连接到总线。若加/减控制信号 $AddSub = 0$，则该模块产生求和操作 $Sum = A+B$；若$AddSub = 1$，则该模块的运算结果为 $A-B$。为了进行减法运算，我们假设加法器/减法器内含有异或门实现对B求2的补码运算，在5.3节我们曾经讨论过这个问题。加法器/减法器产生的输出存储在寄存器G中，而寄存器A和G受信号 A_{in}, G_{in} 和 G_{out}的控制。

图7-73所示的系统可以完成许多种工作，其控制电路的设计决定该系统能完成哪些操作。让我们以设计能完成表7-2所示四种操作的控制电路为例，来说明这个问题。表的左边一列为操作名和操作数；右边一列为操作实现的功能。例如加载操作Load所实现的功能为：$Rx \leftarrow Data$，即把外部数据*Data*经由总线传送到寄存器Rx（Rx为从$R0$到$R3$中的任意一个寄存器）。移动操作Move将存储在寄存器Ry中的数据拷贝到寄存器Rx中。在表中的方括号，例如$[Rx]$，表示的是寄存器的内容。因为只需要总线上的一次操作，所以Load或Move操作都只需要一个时钟周期便能完成。而加法和减法需要三个步骤：第一步，将寄存器Rx的内容经由总线传送到寄存器A。第二步，寄存器Ry的内容放到总线上，加法/减法器模块完成所需要的运算，并将计算结果存储在寄存器G中。最后，第三步，寄存器G的内容被加载到Rx中。

表7-2 处理器进行的操作

操作	执行的功能
Load, *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx]+[Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx]-[Ry]$

能完成表7-2所列操作类型的数字系统通常称为处理器。处理器在某一给定时刻执行的操作是由控制电路的输入信号*Function*指定的。若将输入信号w置为1，操作立即启动，当操作完成时，控制电路的输出信号*Done*置为1。

在图7-60所示的电路中，我们用移位寄存器实现其控制电路。图7-73所示的系统，也可以用类似的方法实现控制电路。为了说明不同的实现途径，我们采用计数器为基础的方法设计控制电路。该电路产生必要的控制信号用于每个操作的每一步。因为最长的操作（加法和减法）需要三个阶段（时钟周期），所以必须使用两位的计数器。图7-74所示是一个两位的递增计数器连接到一个2到4译码器。在6.2节中我们曾讨论过译码器。将译码器的使能信号永远置为1，则译码器永远处在译码工作状态。译码器的每个输出代表了操作过程中的一个步骤。当计数器的值为00时，译码器的输出$T_0 = 1$，表示当前无任何操作。当计数器的值变为01时，译码器的输出$T_1 = 1$，表示进入操作的第一步骤；$T_2 = 1$和$T_3 = 1$分别对应于加法和减法操作的第二和第三个步骤。

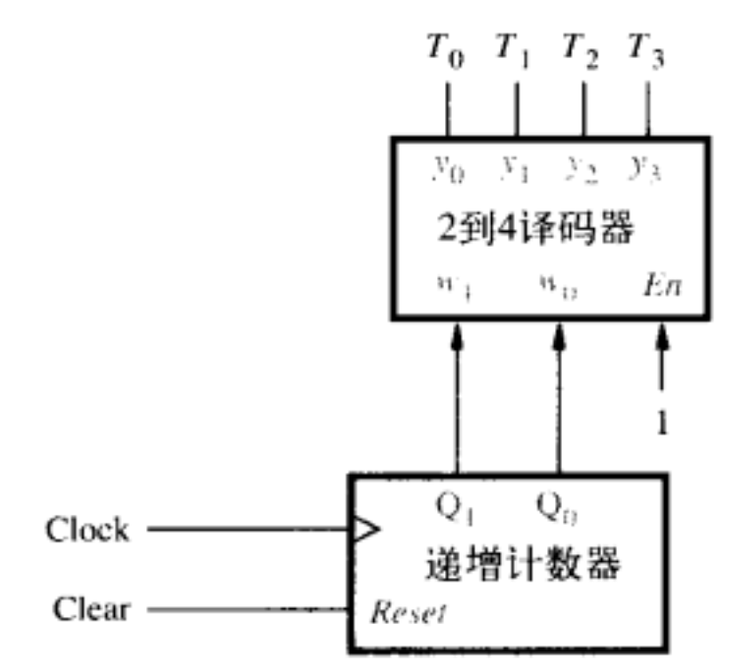


图7-74 处理器控制电路的一个部件

在从T_0到T_3的每个步骤中，取决于执行的操作，控制电路必须产生不同的控制信号值。图7-75展示了所执行的操作是由输入信号*Function*的值（6位）指定的。*Function*值的最左边两位 f_1 和 f_0 用于表示执行什么操作，令$F = f_1 f_0$，则 $F = 00$，01，10，11 分别对应于加载（Load）、移动（Move）、加（Add）、减（Sub）操作。*Function* 输入信号中的$Rx_1 Rx_0$ 是一个二进制数，表示的是操作数Rx，而$Ry_1 Ry_0$ 也是一个二进制数，表示的是操作数Ry。当信号$FR_{in} = 1$时，输入信号*Function*的值被存入6位的功能寄存器。

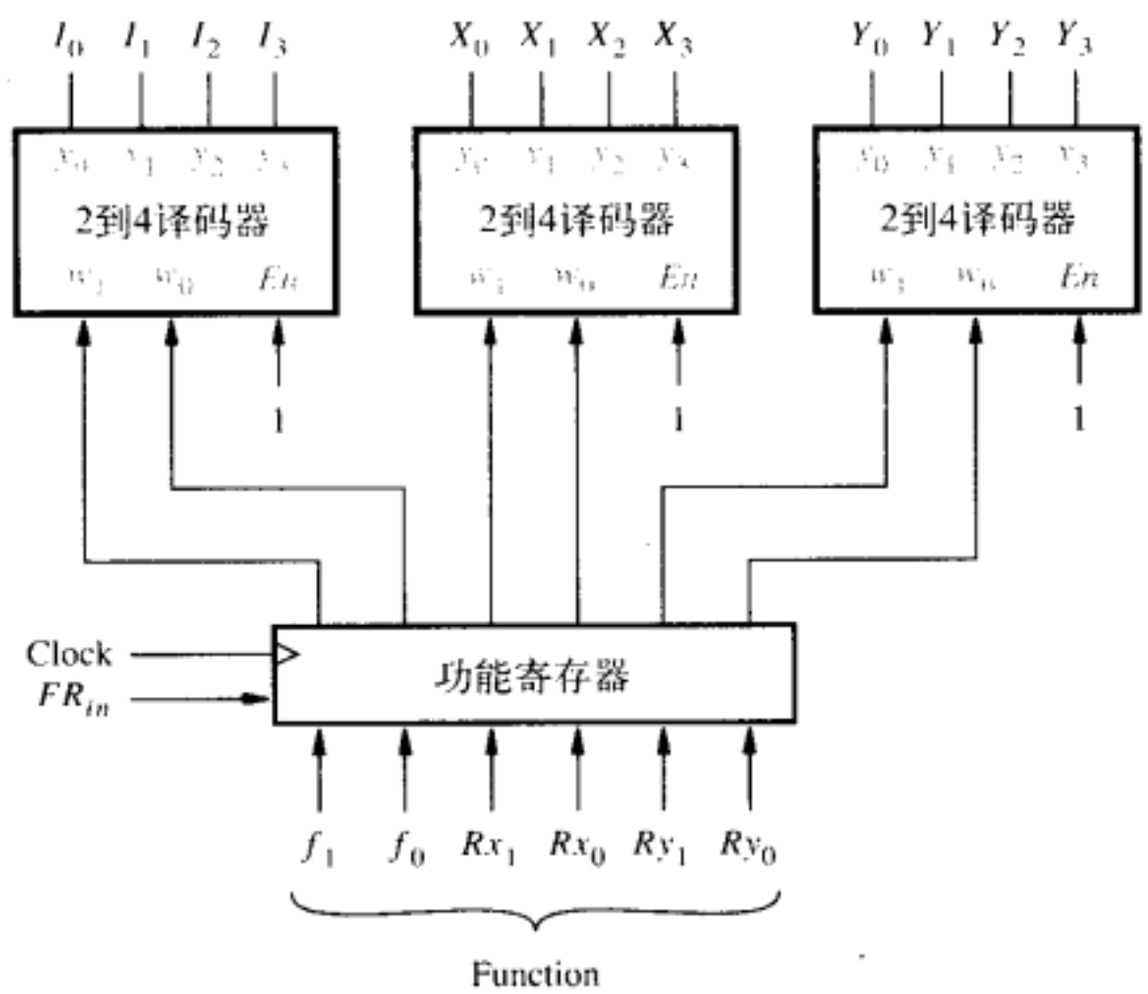


图7-75 功能寄存器和译码器

图7-75还展示了三个2到4译码器，用于对输入信号*Function*中的已编码信息 F, Rx和Ry 进行译码。我们很快就可以看到，添加这些译码器后给电路的设计带来了方便，因为译码器的输出使得许多控制信号的逻辑表达式显得比较简单。

图7-74和图7-75所示的电路组成了控制电路的一部分。利用输入信号 w 和译码器产生的控制信号 T_0，…，T_3，I_0，…，I_3，X_0，…，X_3和 Y_0，…，Y_3，我们便能展示如何推导控制电路的其余部分。控制电路必须产生输出信号*Extern*，*Done*，A_{in}，G_{in}，G_{out}，*AddSub*，$R0_{in}$，…，$R3_{in}$和$R0_{out}$，…，$R3_{out}$，才能对电路的操作进行有效的控制。控制电路还必须产生图7-74和图7-75所示电路所用的输入信号*Clear*和FR_{in} 。

对所有的操作，控制信号*Clear*和FR_{in}（的定义）都是相同的。信号*Clear*用来确保只要 $w = 0$，计数器的值就保持00，并且不执行任何操作；而且在每次操作结束时刻，信号*Clear*用来将计数器的值置为00。因此其逻辑表达式为：

$$Clear = \overline{w}T_0 + Done$$

信号FR_{in}用于在w变为1时，将输入的操作信号*Function*加载到功能寄存器。因此，

$$FR_{in} = wT_0$$

控制电路的其余输出取决于当前的操作和执行的具体步骤。每个信号必须产生的具体值见表7-3所示。表中的每一行对应于特定的操作，每一列表示一个时阶（time step）。信号 *Extern* 只在加载（Load）操作的第一个时阶时等于1，因此实现该信号的逻辑表达式为：

$$Extern = I_0 T_1$$

信号$Done = 1$ 的条件是：加载（Load）和移动（Move）操作的第一个时阶，以及加法（Add）和减法（Sub）操作的第三个时阶。因此，

$$Done = (I_0 + I_1)\, T_1 + (I_2 + I_3)\, T_3$$

只在执行加法（Add）和减法（Sub）操作时，信号A_{in}, G_{in} 和G_{out} 才有可能为1，A_{in} 在时阶T_1为1；G_{in} 在时阶T_2为1；G_{out} 在时阶T_3为1。信号*AddSub*在加法操作时必须置0，在减法操作时必须置1。于是就得到以下的逻辑表达式：

$$A_{in} = (I_2 + I_3)T_1$$
$$G_{in} = (I_2 + I_3)T_2$$
$$G_{out} = (I_2 + I_3)T_3$$
$$AddSub = I_3$$

信号X_0，…，X_3或者信号Y_0，…，Y_3 可用来确定$R0_{in}$，…，$R3_{in}$的值。在表7-3中，这些动作可以用写成$R_{in} = X$ （或者$R_{in} = Y$）来表示。其含义是：$R0_{in} = X_0$，$R1_{in} = X_1$，$R2_{in} = X_2$，$R3_{in} = X_3$。同样，可以用$R_{out} = X$ 或者 $R_{out} = Y$ 来确定$R0_{out}$，…，$R3_{out}$的值。

表7-3 插入每个操作/时阶的控制信号

	T_1	T_2	T_3
(Load): I_0	*Extern*, $R_{in} = X$, *Done*		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, *Done*		
(Add): I_2	$R_{out} = X$, A_{in}	, $R_{out} = Y$, G_{in}, *AddSub* = 0	G_{out} $R_{in} = X$, *Done*
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$ G_{in} *AddSub* = 1	G_{out}, $R_{in} = X$, *Done*

认真考察表7-3，可以写出$R0_{in}$和$R0_{out}$的表达式，然后再写出其他寄存器控制信号的表达式。从表中我们可以看出：在执行加载（Load ）和移动（Move）操作的第一步，以及在加法和减法操作的第三步，都将X_0 的值赋给$R0_{in}$ ，其表达式如下：

$$R0_{in} = (I_0 + I_1)\, T_1 X_0 + (I_2 + I_3)\, T_3 X_0$$

类似地，移动（Move）操作的第一步，执行$R0_{out} = Y_0$ 。在加法和减法操作的第一步，执行$R0_{out} = X_0$ ，在加法和减法操作的第二步，$R0_{out} = Y_0$ 。由此得到：

$$R0_{out} = I_1 T_1 Y_0 + (I_2 + I_3)(T_1 X_0 + T_2 Y_0)$$

$R1_{in}$ 和$R1_{out}$ 的表达式与$R0_{in}$ 和$R0_{out}$ 的表达式基本相同，不同之处仅在于分别用X_1和Y_1代替X_0和Y_0。$R2_{in}$，$R2_{out}$，$R3_{in}$，$R3_{out}$的表达式可以用同样的方法写出。

图7-74和图7-75所示的电路，加上由上述表达式表示的电路，共同构成图7-73中的控制电路。

处理器是应用极其广泛的有用电路。我们只介绍了处理器设计的最基本方面。然而这里所介绍的技术可以用来设计真正的处理器，例如现代微处理器。感兴趣的读者可以参阅计算机组成的书籍来了解处理器设计的细节[1-2]。

Verilog 代码

本节中我们用两种不同风格的Verilog代码来描述图7-73所示的系统。第一种风格用三态缓冲器表示总线，这段代码给出了上面展示的控制电路的输出的逻辑表达式。第二种风格的代码用多路器表示总线，并用对应于表7-3的**case**语句来描述控制电路的输出。

图7-56所示是一个递增计数器的Verilog代码。该计数器（名为upcount）代码的修改版见图7-76所示。该电路改用高电平有效的同步复位输入信号。用Verilog代码

```
module upcount (Clear, Clock, Q);
  input Clear, Clock;
  output reg [1:0] Q;

  always @(posedge Clock)
    if (Clear)
      Q <= 0;
    else
      Q <= Q + 1;

endmodule
```

图7-76 带同步复位的两位递增计数器

编写的可用作处理器部件的其他子电路包括：dec2to4、regn和 trin 模块，分别见图6-35、图7-66和7-67所示。

描述该处理器的完整代码见图7-77所示。 *counter*和*decT*实例引用了图7-74所示的子电路upcount和dec2to4。请注意，我们已经假定upcount电路有一个高电平有效的复位输入信号*Reset*，该信号用于把计数器初始化为00。 语句**assign** Func = {F, Rx, Ry} 用来把操作符和操作数拼接成6位的信号*Func*，该信号表示传送到图7-75所示功能寄存器的输入信号。functionreg 模块表示输入为*Func*以及输出为*FuncReg*的功能寄存器。*decI*、*decX* 和*decY*实例引用了图7-75所示的译码器（dec2to4）。这几条语句后面列出了前面导出的表示控制电路输出的逻辑表达式。为了生成表示$R0_{in}$，…，$R3_{in}$和 $R0_{out}$，…，$R3_{out}$的表达式，用了**for**循环语句。在代码的最后，定义了加法器/减法器模块，并实例引用了三态缓冲器和寄存器子电路模块。

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
   input [7:0] Data;
   input Reset, w, Clock;
   input [1:0] F, Rx, Ry;
   output wire [7:0] BusWires;
   output Done;
   reg [0:3] Rin, Rout;
   reg [7:0] Sum;
   wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
   wire [1:0] Count;
   wire [0:3] T, I, Xreg, Y;
   wire [7:0] R0, R1, R2, R3, A, G;
   wire [1:6] Func, FuncReg;
   integer k;

   upcount counter (Clear, Clock, Count);
   dec2to4 decT (Count, 1'b1, T);

   assign Clear = Reset | Done | (~w & T[0]);
   assign Func = {F, Rx, Ry};
   assign FRin = w & T[0];

   regn functionreg (Func, FRin, Clock, FuncReg);
      defparam functionreg.n = 6;
   dec2to4 decI (FuncReg[1:2], 1'b1, I);
   dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
   dec2to4 decY (FuncReg[5:6], 1'b1, Y);

   assign Extern = I[0] & T[1];
   assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
   assign Ain = (I[2] | I[3]) & T[1];
   assign Gin = (I[2] | I[3]) & T[2];
   assign Gout = (I[2] | I[3]) & T[3];
   assign AddSub = I[3];

   // RegCntl
   always @(I, T, Xreg, Y)
      for (k = 0; k < 4; k = k+1)
      begin
         Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
            ((I[2] | I[3]) & T[1] & Y[k]);
         Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
            ((T[1] & Xreg[k]) | (T[2] & Y[k])));
      end
```

图7-77 处理器代码

```
    trin tri_ext (Data, Extern, BusWires);
    regn  reg_0 (BusWires, Rin[0], Clock, R0);
    regn  reg_1 (BusWires, Rin[1], Clock, R1);
    regn  reg_2 (BusWires, Rin[2], Clock, R2);
    regn  reg_3 (BusWires, Rin[3], Clock, R3);

    trin  tri_0 (R0, Rout[0], BusWires);
    trin  tri_1 (R1, Rout[1], BusWires);
    trin  tri_2 (R2, Rout[2], BusWires);
    trin  tri_3 (R3, Rout[3], BusWires);
    regn  reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub, A, BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A - BusWires;

    regn  reg_G (Sum, Gin, Clock, G);
    trin  tri_G (G, Gout, BusWires);

endmodule
```

图7-77 （续）

用多路器和case语句

我们曾用图7-65说明也可以不用三态缓冲器而改用多路器来实现总线。图7-78所示的Verilog代码描述了用多路器实现的处理器。该代码说明了描述处理器控制电路的另一种途径。这段代码与图7-77中的代码不同，没有给出表示*Extern*、*Done*等信号的逻辑表达式。相反，用了**case**语句来表示表7-3所示的信息。每个控制信号首先被赋值0（默认初始值）。这样做是必须的，因为只有当某控制信号需要为1的时候，**case**语句才把1赋予该控制信号，如表7-3所示。正如我们在7.12.2节中解释的那样，当某个信号未被赋值时，该信号应保持当前的值。这意味着在综合后的电路中将存在反馈连接，会产生记忆的结果。为了避免出现这个问题，我们给每个涉及**case**语句的控制信号都赋一个默认的初始值0 。

```
module  proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input  [7:0] Data;
    input  Reset, w, Clock;
    input  [1:0] F, Rx, Ry;
    output  reg  [7:0] BusWires;
    output  reg  Done;
    reg  [7:0] Sum;
    reg  [0:3] Rin, Rout;
    reg  Extern, Ain, Gin, Gout, AddSub;
    wire  [1:0] Count, I;
    wire  [0:3] Xreg, Y;
    wire  [7:0] R0, R1, R2, R3, A, G;
    wire  [1:6] Func, FuncReg, Sel;

    wire  Clear = Reset | Done | (~w & ~Count[1] & ~Count[0]);
    upcount  counter (Clear, Clock, Count);
    assign  Func = {F, Rx, Ry};
    wire  FRin = w & ~Count[1] & ~Count[0];
    regn  functionreg (Func, FRin, Clock, FuncReg);
```

图7-78 处理器代码的另一种写法

```
    defparam functionreg.n = 6;
  assign  I = FuncReg[1:2];
  dec2to4  decX (FuncReg[3:4], 1'b1, Xreg);
  dec2to4  decY (FuncReg[5:6], 1'b1, Y);

  always @(Count, I, Xreg, Y)
  begin
    Extern = 1'b0; Done = 1'b0; Ain = 1'b0; Gin = 1'b0;
    Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
    case (Count)
      2'b00: ; //no signals asserted in time step T0
      2'b01:   //define signals in time step T1
        case (I)
          2'b00: begin //Load
                    Extern = 1'b1; Rin = Xreg; Done = 1'b1;
                 end
          2'b01: begin //Move
                    Rout = Y; Rin = Xreg; Done = 1'b1;
                 end
          default: begin //Add, Sub
                     Rout = Xreg; Ain = 1'b1;
                   end
        endcase
      2'b10: //define signals in time step T2
        case(I)
          2'b10: begin //Add
                    Rout = Y; Gin = 1'b1;
                 end
          2'b11: begin //Sub
                    Rout = Y; AddSub = 1'b1; Gin = 1'b1;
                 end
          default: ; //Add, Sub
        endcase
      2'b11:
        case (I)
          2'b10, 2'b11: begin
                          Gout = 1'b1; Rin = Xreg; Done = 1'b1;
                        end
          default: ; //Add, Sub
        endcase
    endcase
  end

  regn  reg_0 (BusWires, Rin[0], Clock, R0);
  regn  reg_1 (BusWires, Rin[1], Clock, R1);
  regn  reg_2 (BusWires, Rin[2], Clock, R2);
  regn  reg_3 (BusWires, Rin[3], Clock, R3);
  regn  reg_A (BusWires, Ain, Clock, A);
      // alu
      always @(AddSub, A, BusWires)
      begin
        if (!AddSub)
          Sum = A + BusWires;
        else
          Sum = A - BusWires;
      end

      regn reg_G (Sum, Gin, Clock, G);
      assign Sel = {Rout, Gout, Extern};
```

图7-78 （续）

```
always @(Sel, R0, R1, R2, R3, G, Data)
begin
   if (Sel == 6'b100000)
      BusWires = R0;
   else if (Sel == 6'b010000)
      BusWires = R1;
   else if (Sel == 6'b001000)
      BusWires = R2;
   else if (Sel == 6'b000100)
      BusWires = R3;
   else if (Sel == 6'b000010)
      BusWires = G;
   else BusWires = Data;
end

endmodule
```

图7-78 （续）

在图7-77所示的代码中，译码器*decT*和*decI*被分别用于对信号*Count*和储存的输入*F* 值进行译码。译码器*decT*的输出为T_0, …, T_3, 译码器*decI*的输出为I_0, …, I_3。在图7-78所示的代码中，没有使用这两个译码器，因为在这段代码中它们没有起任何作用。取而代之的是：信号*I*被定义为一个两位的信号，且使用了两位的信号*Count*，而没有使用信号*T*。这几个信号都用在**case**语句中。在该代码中，*I* 被设置为功能寄存器的最左边两位，即输入信号*F* 的值。

在代码中有两条嵌套的**case**语句。第一条**case**语句枚举了所有可能的*Count*值。在这条**case**语句中，每个分支表示了表7-3中的一列，每个分支中又有一个嵌套的**case**语句，该语句枚举了 *I* 的四个取值。正如代码中注释语句所表明的那样，嵌套的**case**语句正好对应于表7-3所给定的信息。

在图7-78所示代码的最后一段，总线是用一条表示多路器的**if-else**语句描述的。该多路器根据输入的选择控制信号（R_{out}、G_{out} 和 *Extern*）选择合适的数据放置到总线上。

图7-77和图7-78所示的代码综合后生成的电路在功能上是等价的。图7-78所示代码风格的优点在于它不需要对表7-3进行手工分析，便能推导出图7-77代码所需要的表示控制信号的逻辑表达式。采用图7-78所示风格的代码，Verilog编译器能自动分析这些**case**语句，从而很快地产生这些表达式。图7-78所示风格的代码不易发生由于不小心而产生错误。而且采用这种风格的代码，当需要为处理器添加其他一些操作功能时，代码的修改也会变得很直接和简单。

我们曾把图7-78所示的代码综合成电路用一个芯片来实现。图7-79所示为该芯片电路的时序仿真波形。在该时序仿真波形图中，$w = 1$下的每个时钟周期表明一个操作的开始。在仿真时间为250ns时刻，遇到第一个这样的操作，此时输入*F* 和*Rx* 的值都为00。因此执行的操作对应于“将数据*Data*加载到*R*0”。数据*Data*的值为2A，在下一个时钟正跳变沿，2A被加载到*R*0。下一次操作将55加载到寄存器*R*1，接下去将22加载到寄存器*R*2。在850ns时刻输入信号*F*的值为10，而$Rx = 01$，$Ry = 00$。该操作是“将*R*1与*R*0相加”。在下面一个时钟周期里，*R*1的内容（55）出现在总线上。在950ns时刻的时钟沿把该数据加载到寄存器*A*中，该时钟沿同时将*R*0的内容（2A）放置在总线上。加法器/减法器模块产生正确的和信号*sum*(7F)，在1050ns时刻，该和信号*sum*被加载到寄存器*G*。该时钟沿后，在1150ns时刻，寄存器*G*的新内容（7F）被放置到总线上，并被加载到寄存器*R*1。在时序图上还显示了两个操作。在1250ns的那个操作（“将*R*1移至*R*3”），将 *R*1的内容 (7F) 拷贝至 *R*3。最后，在1450ns的那个操作（“将*R*3减去*R*2”）将*R*3的内容 (7F)减去*R*2的内容 (22)，得到正确的运算结果：7F − 22 = 5D。

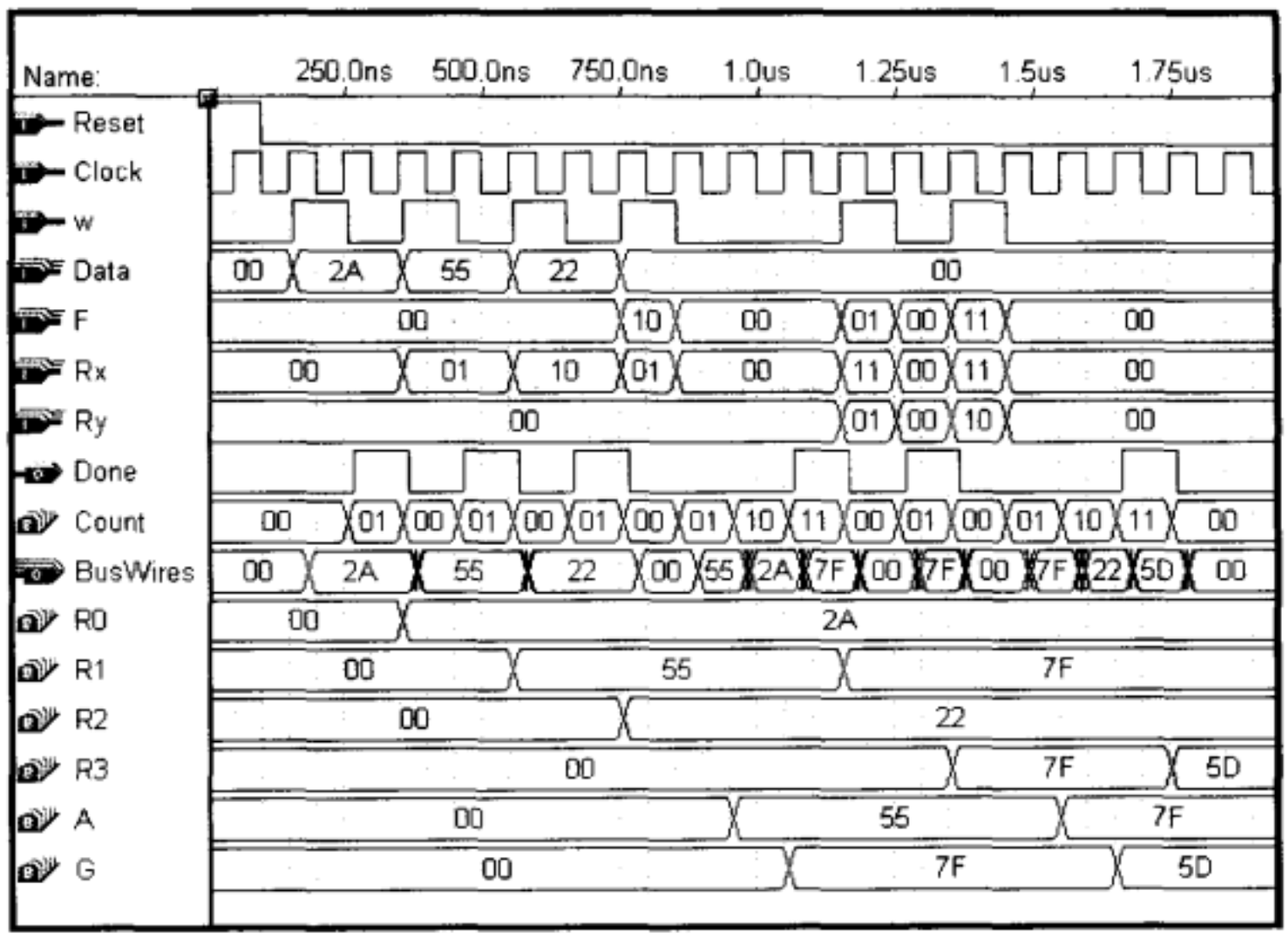


图7-79 图7-78所示的Verilog代码的时序仿真波形

7.14.3 反应计时器

在第3章中我们曾经介绍过逻辑门的延迟一般小于1ns，使用这种逻辑门的电子器件，其运行速度相当快。在本例中，我们用逻辑电路来测量一个运行速度特别缓慢的器件，即人的反应速度。

现在让我们来设计一个可以用来测量人对特定事件作出反应所需时间的电路。该电路点亮一个称为*发光二极管*（LED）的小灯泡，被测试的人看到发光二极管点亮后，尽可能快地按下开关。该电路可以测量从发光二极管点亮起至开关被测试者按下之间的时间。

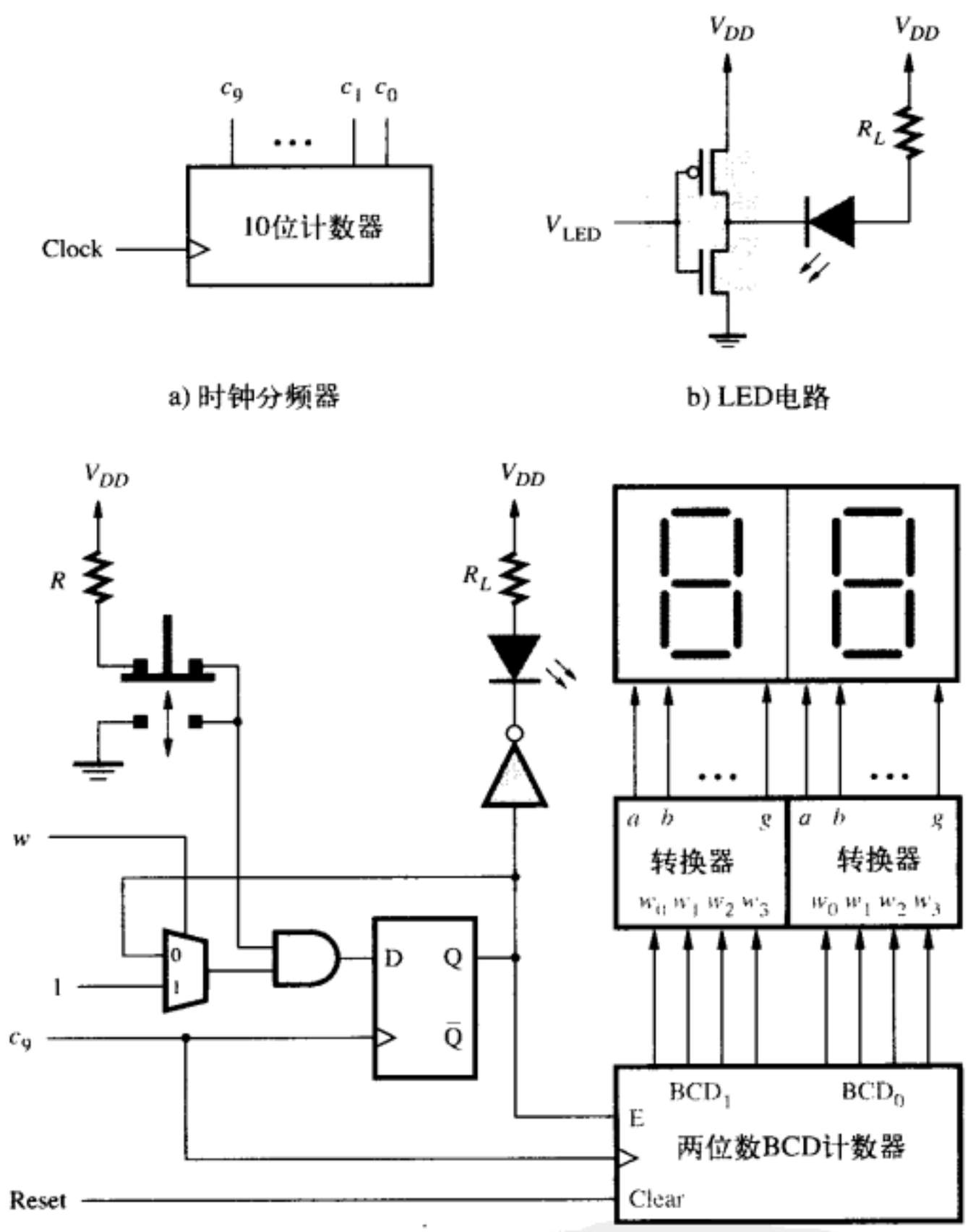
为了测量人对事件的反应时间，需要有一个频率合适的时钟信号。在本例中，我们使用了100Hz的时钟信号，即测量时间的分辨度为1/100秒。人对LED发光的反应时间用两个数字来显示，表示的时间范围为：从1秒的 00/100到99/100。

数字系统通常有几个频率很高的时钟信号用于控制各种子系统。假设某数字系统的输入时钟信号频率为102.4kHz。利用该时钟信号，可以用计数器构成*时钟分频器*得到所需要的100Hz信号。图7-22所示为四位计数器的时序图。该图展示该计数器最低位的输出Q_0是一个周期信号，其频率是输入时钟信号的一半。因此，我们把输出Q_0看成是时钟信号的2分频。同理，输出Q_1是时钟信号的4分频。就一般情况而言，n位计数器的输出Q_i 的频率为输入时钟频率的2^{i+1}分频。在时钟频率为102.4kHz的情况下，需要用10位计数器，见图7-80a所示。计数器的输出c_9的时钟频率就是所需要的100Hz，因为102400Hz/1024 = 100Hz。

反应计时器电路必须能够开启和关闭LED。在图7-80b中，用灰色图形符号表示的为LED，该图形符号边上的两个小箭头表示LED开启时发出的光。LED有两个电极，左边的是*负极*，右边的是*正极*。为了点亮LED，负极的电平必须低于正极，这样就有电流流经LED。若LED两端的电压相等，则LED关闭。

图7-80b展示了用反相器控制LED的方法。若输入电压$V_{\text{LED}} = 0$，则负极电压等于V_{DD}，因此LED关闭。但是若输入电压$V_{\text{LED}} = V_{DD}$，则负极电压等于0V，因此LED开启。流经LED的电流量受到电阻R_L的限制。该电流流经LED和反相器的NMOS晶体管。因为电流流入反相器，

所以我们称反相器吸收（sink）电流。在不会造成永久性破坏的前提下，逻辑门所能吸收的最大电流通常称为I_{OL}，该符号表示“当输出为低电平时的最大电流”。选择电阻R_L的值时要注意确保电流小于I_{OL}。假设反相器位于PLD器件内部，在PLD器件的技术说明书上，典型的I_{OL}值规定为12 mA上下。若V_{DD} = 5V，则$R_L \approx 450\Omega$，这是因为5V/450Ω=11mA（导通时LED两端实际上存在一个小的电压降，但为了简单起见，我们将其忽略不计）。由LED所发出的光亮度与电流是成正比例关系的。若11mA不够大，则我们应该如3.5节中所介绍的那样，采用缓冲器芯片中的反相器，因为缓冲器可以提供较大的LED驱动电流I_{OL}。



a) 时钟分频器　　b) LED电路

c) 按钮开关，LED及7段显示器

图7-80 反应计时器电路

完整的反应计时器电路如图7-80c所示，用了一个来自于图7-80b所示的反相器，图中把反相器涂成灰色。图7-80c中左上方有一个按钮开关的图形符号。该开关正常情况下将上面两个触点连接起来，见图所示。当按下去时，开关使下面两个触点连接，一旦放手，按钮开关弹回到正常（上面）位置。按照图上按钮开关的连接，正常情况下开关输出逻辑1，当按钮开关被按下后放手，相当于产生从1变为0又变为1的负脉冲。

当被按下时，按钮开关使得D触发器同步复位。该触发器的输出决定LED是点亮还是熄灭，它也为两位数BCD计数器提供了使能输入信号。在7.11节我们曾讨论过，BCD计数器的每位

数字为四位二进制数，其取值范围从0000到1001。因此，两位BCD码计数器的计数序列可以看作从00到99的十进制数。图7-28所示为BCD码计数器电路。图7-80a所示的为时钟分频器，其输出为c_9。在图7-80c所示的电路中，c_9连接到触发器和计数器的时钟输入端。反应计时器电路按照以下步骤操作：首先按下按钮开关，熄灭LED，并且禁止计数器计数。接着复位信号*Reset*有效将计数器清0。输入信号*w*在正常情况下为0，使触发器保持为0，并阻止计数器的值发生变化。将*w*设置为1维持一个c_9时钟周期，反应测试随即开始。当时钟信号c_9的下一个正跳变沿到达时，触发器的输出变为1，随即点亮LED。假定一个时钟周期后，*w*回到0，因为电路中存在着连接到触发器输入端D的2选1多路器，故触发器的输出仍保持为1。因此LED继续点亮，计数器上显示的数字每1/100秒递增1。计数器中的每个数字连接到一个曾介绍过的7段显示器（见图6-25）。当用户再次按下按钮开关后，触发器被清0，随即LED熄灭，计数器停止计数。两位数显示器显示从$w = 1$将LED点亮开始算起，到被测试者产生反应按下按钮开关为止所经历的时间，分辨度为1/100秒。

Verilog 代码

为了用Verilog代码描述图7-80c所示的电路，我们可以利用BCD码计数器和7段码转换器子电路。7段码转换器子电路的代码见图6-38所示，这里没有重写。代表图7-28所示电路的BCD码计数器的Verilog代码见图7-81所示。两个BCD数码的输出用2个四位的信号*BCD*1和*BCD*0表示。清零输入同时为该计数器的两个数字提供了同步复位。若$E = 1$，则计数器的值在时钟的正跳变沿时刻递增1；若$E = 0$，则计数器的值不变。每个数字的取值范围从0000到1001。

图7-82给出了反应计时器的代码。输入信号*Pushn*表示由按钮开关产生的值。输出信号 *LEDn*表示用于控制LED的反相器的输出。两个7段显示器受从*Digit*1 到*Digit* 0的七位信号的控制。

```
module BCDcount (Clock, Clear, E, BCD1, BCD0);
   input Clock, Clear, E;
   output reg [3:0] BCD1, BCD0;

   always @(posedge Clock)
   begin
      if (Clear)
      begin
         BCD1 <= 0;
         BCD0 <= 0;
      end
      else if (E)
         if (BCD0 == 4'b1001)
         begin
            BCD0 <= 0;
            if (BCD1 == 4'b1001)
               BCD1 <= 0;
            else
               BCD1 <= BCD1 + 1;
         end
         else
            BCD0 <= BCD0 + 1;
   end

endmodule
```

图7-81 图7-28所示的两位BCD计数器的Verilog代码

```
module reaction (c9, Reset, w, Pushn, LEDn, Digit1, Digit0);
   input c9, Reset, w, Pushn;
   output wire LEDn;
   output wire [1:7] Digit1, Digit0;
   reg LED;
   wire [3:0] BCD1, BCD0;

   always @(posedge c9)
   begin
      if (Pushn == 0)
         LED <= 0;
      else if (w)
         LED <= 1;
   end

   assign LEDn = ~LED;
   BCDcount counter (c9, Reset, LED, BCD1, BCD0);
   seg7 seg1 (BCD1, Digit1);
   seg7 seg0 (BCD0, Digit0);

endmodule
```

图7-82 反应计时器的Verilog代码

图7-61所示的电路表明如何将寄存器*R*设计成由控制信号R_{in}来控制。若 $R_{in} = 1$，则数据在有效时钟跳变沿时刻将数据加载到寄存器中；若$R_{in} = 0$，则寄存器中存储的数据不变。在图7-80

所示的电路中，触发器所用的控制方法与之相同。若$w = 1$，则值1被加载到触发器中；若$w = 0$，则触发器中存储的值不变。该电路用图7-82所示的**always**块描述，电路中还包括一个同步复位输入信号。我们之所以选择用同步复位清0，这是因为触发器的输出是连接到BCD计数器的使能输入端E上的。由7.3节的讨论可知，连接到触发器的所有信号必须满足建立和保持时间的约束条件才能可靠地加载到寄存器。按钮开关可以在任何时刻按下，与时钟信号c_9并没有同步关系。对图7-80所示的触发器使用同步复位清0，可以避免计数器中可能出现的时序问题。

触发器的输出为LED，把该信号反相便产生控制LED点亮或者熄灭的信号 *LEDn*。*LEDn* 应该由逻辑芯片外的独立缓冲器产生，该缓冲器被连接到芯片封装的输出引脚上。若反应计时器电路用PLD实现，则该缓冲器输出的逻辑0电流约为$I_{OL} = 12\text{mA}$ 。在图7-82所示代码的末尾部分，实例引用了BCD计数器和7段码转换器子电路模块。

图7-83所示是在芯片上实现的反应计时器电路的仿真波形。信号*Pushn*的初始状态被设置为0，用它来仿真将按钮开关按下使LED熄灭的情况，片刻后信号*Pushn*又回到1。使复位信号 *Reset* = 1，计数器清为0。当w变为1，该电路将 *LEDn* 设置为 0，随即LED点亮。经过一段时间后，按钮开关将被按下。在仿真中，我们随意假设经过18个c_9时钟周期后将*Pushn*设置为0。这样的假设表示被测试者对*LED*点亮的反应时间大约为0.18秒。从人的角度看，这段时间是非常短的，而对电子电路而言，则是非常长的一段时间。价格低廉的个人计算机在0.18秒的时间内就可以完成几千万次操作。

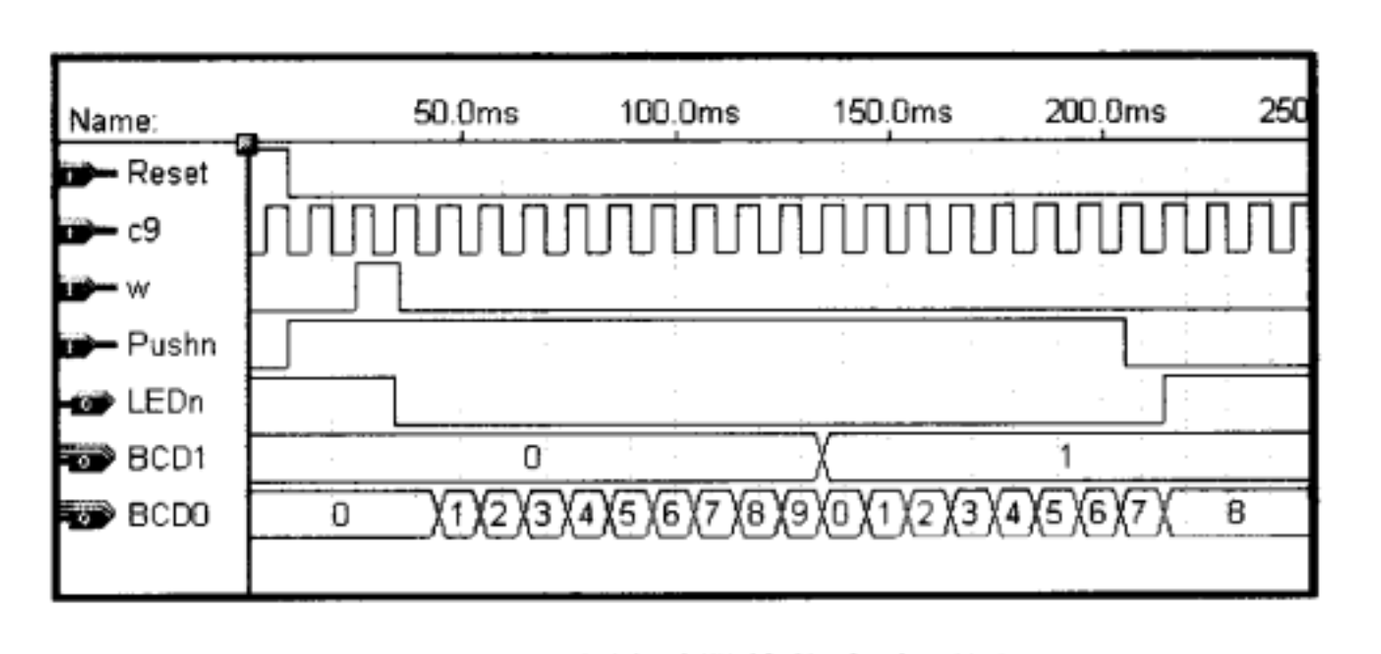


图7-83 反应计时器的仿真波形图

7.14.4 寄存器传输级代码

到现在为止，我们已经介绍了大部分需要进行综合的Verilog结构。所举的例子大部分都是利用**if-else**语句、**case**语句、**for**循环语句和其他过程性语句编写的行为代码。使用类似于计算机编程语言的风格来编写行为代码是可能的，但在用计算机编程风格编写的代码中，往往带有许多循环和条件分支的复杂控制流；我们甚至很难预测由高层次综合工具生成的具体电路究竟是什么样的。本书中，我们不使用高层次风格的代码。与此相反，我们编写的Verilog代码能很容易地与被描述的电路联系起来。绝大部分设计的模块都相当小，描述也非常简单。规模大的设计由许多较小的模块互相连接而成。这种设计途径通常称为寄存器传输级（RTL）风格的代码。在实际工作中，编写RTL风格的代码是最广泛的设计方法。RTL代码的特点在于直接用代码表示控制流；该风格的代码也包含很容易理解的用简单的方法互相连接在一起的子电路。

7.15 触发器电路的时序分析

在图7-15中，我们曾经展示了与D触发器相关的时序参数。图7-84给出了一个使用D触发

器的简单电路。我们想要计算该电路能正常工作的最大的时钟频率F_{max}，并且想确定该电路的保持时间是否不够长。在技术文献中，这种类型的电路分析通常叫做时序分析。假设该触发器的时序参数为：t_{su} = 0.6ns, t_h = 0.4ns，0.8ns $\leqslant t_{cQ} \leqslant$ 1.0 ns。给t_{cQ}参数规定一个范围的原因，我们曾在7.4.4节提及，因为延迟参数分布在一定范围内，这样处理是现成集成电路芯片常用的办法。

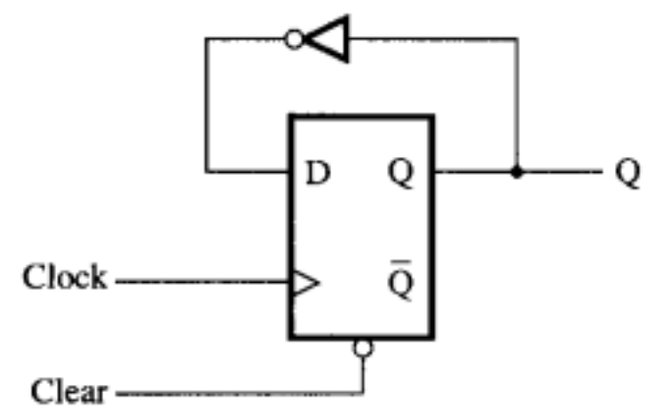


图7-84 一个简单的触发器电路

为了计算最小的时钟信号周期$T_{min} = 1/F_{max}$，我们必须考虑在触发器中从开始到结束的所有路径。在这个简单的电路中，只有一条这样的路径，这条路径开始于数据被时钟信号的正跳变沿加载进入触发器，经过t_{cQ}的延迟后传播到Q的输出端，再传播通过非门，同时必须满足D输入端的建立时间的要求。因此：

$$T_{min} = t_{cQ} + t_{NOT} + t_{su}$$

由于我们关注的只是计算出最长的延迟时间，所以应该用t_{cQ}的最大值。为了计算出t_{NOT}，我们将假设通过任何逻辑门的延迟都可以用$1 + 0.1k$进行计算，其中k是该门的输入信号的个数。对非门而言，k = 1.1ns，因此得到如下T_{min}和F_{max}值：

$$T_{min} = 1.0 + 1.1 + 0.6 = 2.7\text{ns}$$
$$F_{max} = 1/2.7\text{ns} = 370.37\text{MHz}$$

当然，有必要检查电路中的保持时间是否违反规定。在这种场合，我们必须核查从时钟信号的正跳变沿到D输入值改变的最短延迟。该延迟由$t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9$ns给定。因为1.9ns > t_h = 0.4ns，所以保持时间足够长，没有违反规定。

再举一个触发器电路时序分析的例子，请考虑图7-85所示的计数器电路。假设所用的触发器的时序参数与图7-84中用过的触发器相同，请计算该电路能正常运行的最高频率。再次假设通过逻辑门的传播延迟可以用$1 + 0.1k$来计算。

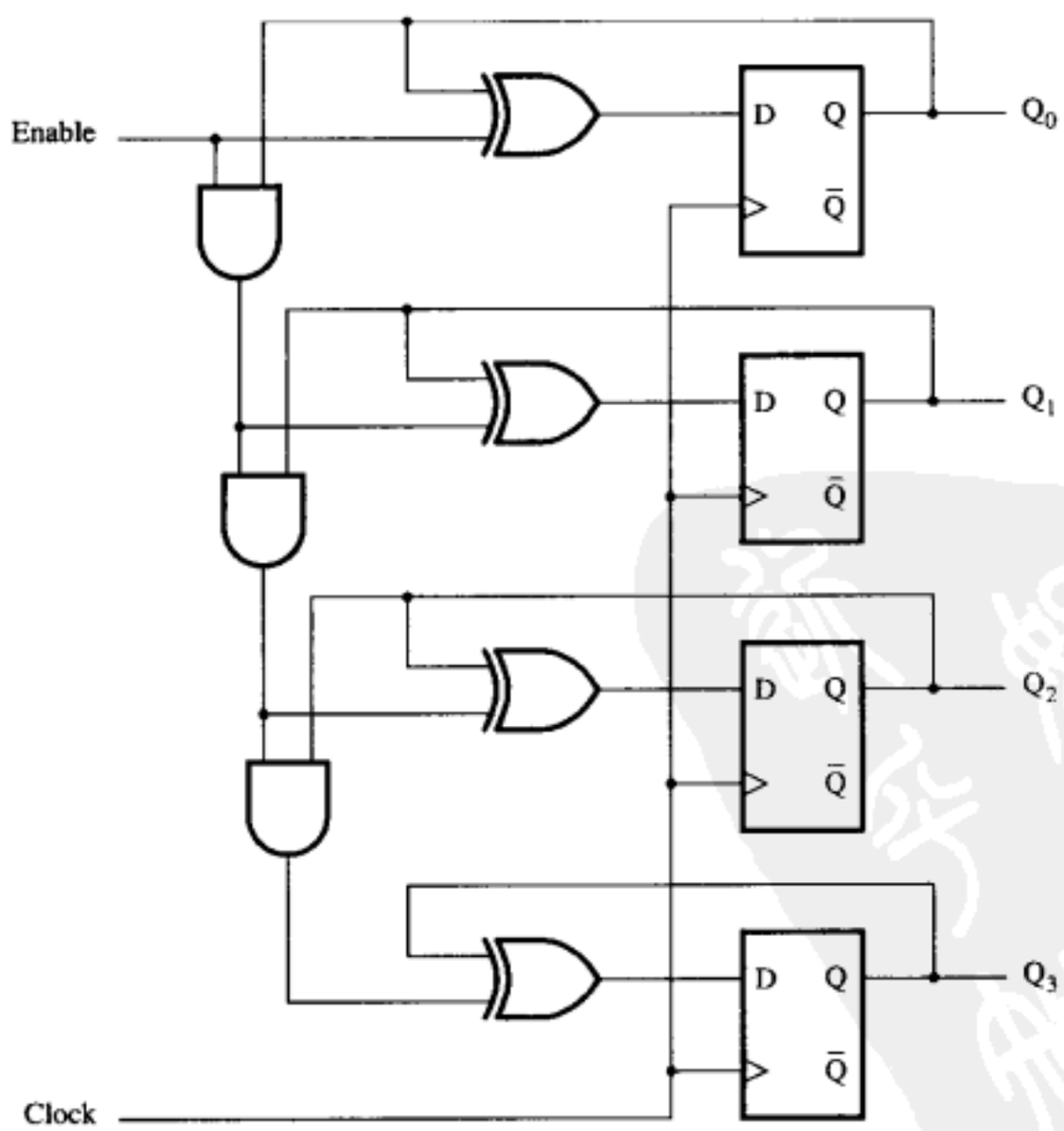


图7-85 一个四位的计数器电路

在这个电路中，存在着四个触发器从开始到结束的许多路径。最长的路径从触发器Q_0起到触发器Q_3结束。在某个电路中最长的路径称为*关键路径*。关键路径的延迟包括触发器Q_0的时钟信号到Q的延迟、通过三个与门的传播延迟和一个异或门的延迟。我们还必须考虑触发器Q_3的建立时间。因此，得到

$$T_{min} = t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su}$$

用t_{cQ}的最大值，得到

$$T_{min} = 1.0 + 3(1.2) + 1.2 + 0.6\text{ns} = 6.4\text{ns}$$
$$F_{max} = 1/6.4\text{ns} = 156.25\text{MHz}$$

该电路的最短路径是从每个触发器通过异或门反馈到该触发器本身的输入端。沿每个这样路径的最小延迟为 $t_{cQ} + t_{XOR} = 0.8 + 1.2 = 2.0\text{ns}$。因为 $2.0\text{ns} > t_h = 0.4\text{ns}$，因此保持时间足够长，没有违反规定。

在上面的分析中，假设时钟信号同时到达所有四个触发器。我们现在将重复这个分析，假设时钟信号同时到达触发器Q_0, Q_1和Q_2，但到达触发器Q_3有一些延迟。时钟到达不同的触发器之间的时间差称为*时钟偏差*（clock skew），记作t_{skew}，时钟偏差可以由许多原因引起。

在图7-85中，电路的关键路径是从触发器Q_0起到触发器Q_3。然而，Q_3的时钟偏差使得这个延迟减少，因为时钟偏差在数据被加载进该触发器前提供了附加的时间。如果考虑增加1.5ns的时钟偏差，则从触发器Q_0起到触发器Q_3的路径延迟由 $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} - t_{skew} = 6.4 - 1.5\text{ns} = 4.9\text{ns}$ 给定。该电路现在还存在一个不同的关键路径，该路径从触发器Q_0起到触发器Q_2结束。这条路径的延迟为

$$\begin{aligned} T_{min} &= t_{cQ} + 2(t_{AND}) + t_{XOR} + t_{su} \\ &= 1.0 + 2(1.2) + 1.2 + 0.6\text{ns} \\ &= 5.2\text{ns} \\ F_{max} &= 1/5.2\text{ns} = 192.31\text{MHz} \end{aligned}$$

在这种场合，时钟偏差导致电路的最高时钟频率提高。但是，如果时钟偏差是负的，即触发器Q_3的时钟到达时间比其他触发器更早一些，则将会造成该电路的最高时钟频率F_{max}降低。

因为数据加载到触发器Q_3被时钟偏差延迟了，所以对所有起始于Q_0, Q_1 和Q_2而以Q_3为结束点的路径，都会产生使触发器Q_3的保持时间需要增加到$t_h + t_{skew}$的影响。在该电路中，这种最短的路径是从触发器Q_2到Q_3的路径，其延迟时间为$t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2\text{ns}$。因为$3.2\text{ns} > t_h + t_{skew} = 1.9\text{ns}$，所以保持时间足够长，没有违反规定。

如果对时钟偏差值$t_{skew} \geqslant 3.2 - t_h = 2.8\text{ns}$，重复以上保持时间的分析，则会出现保持时间不够的情况。当$t_{skew} \geqslant 2.8\text{ns}$时，该电路将不可能在任何频率下可靠地运行。由于时钟偏差的存在会引起复杂的电路时序问题，所以好的电路设计方法必须保证时钟信号到达所有触发器的偏差尽可能小。我们将在10.3节讨论时钟的同步问题。

7.16 小结

本章介绍了数字系统中起基础存储元件作用的电路。这些元件用来构建更大一些的单元，例如寄存器、移位寄存器和计数器。有许多技术资料[3-11]介绍这方面的内容。我们已经说明了如何用Verilog代码来描述触发器的电路。关于Verilog的更多信息，可以见参考文献[12-19]。关于设计带触发器电路的更规范的方法，我们将在下一章中介绍。

7.17 问题求解举例

本节列举了读者可能遇到的某些典型问题，并且展示了这些问题是如何求解的。

例7.17 考虑图7-86a所示的电路。假设输入信号C由一个占空比为50%的方波驱动。画出显示点A和点B波形的时序图。假设通过每个门的传播延迟为Δ秒。

解：点A和点B波形的时序图如图7-86b所示。

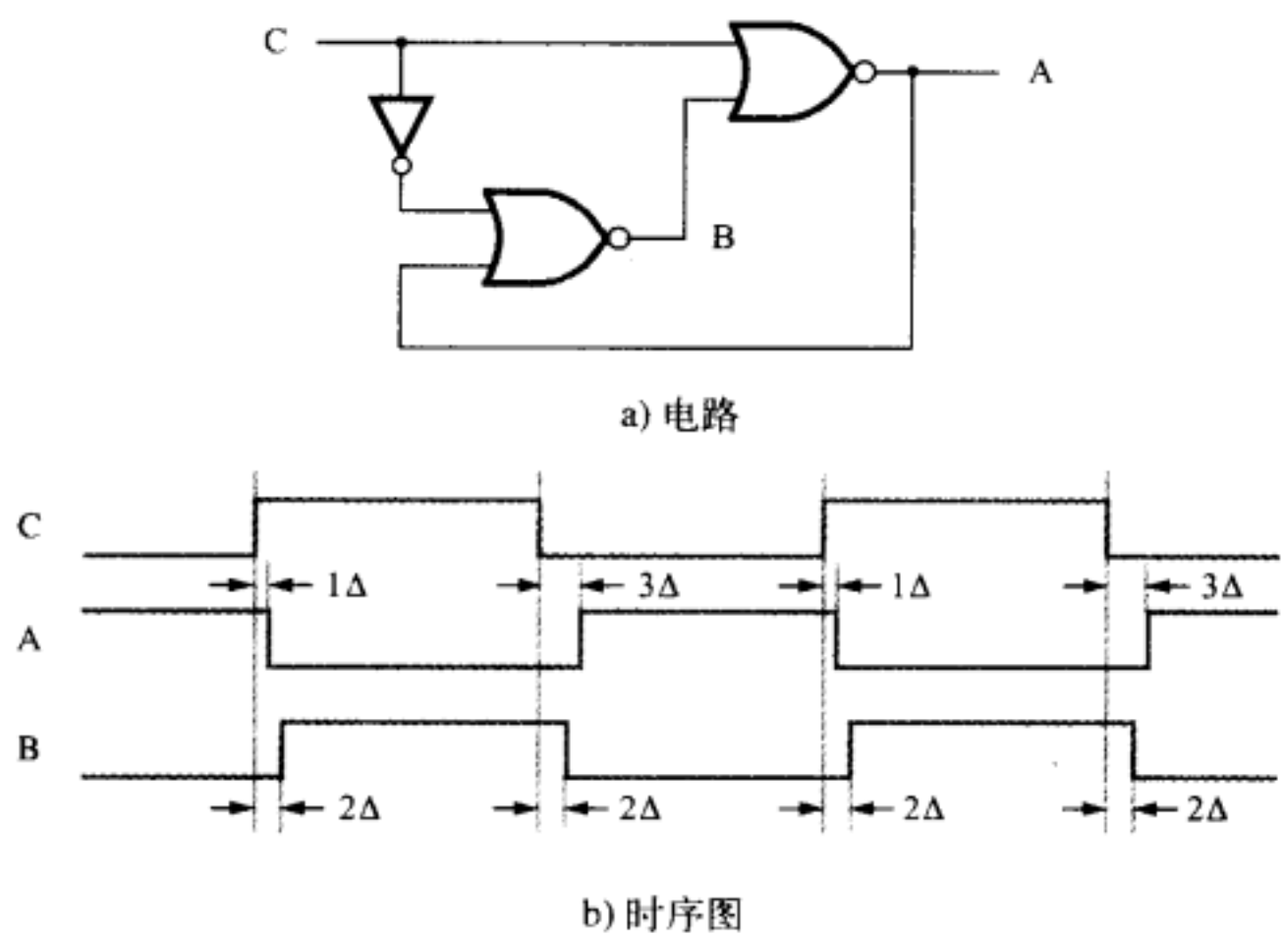


图7-86 例7.17的电路

例7.18 确定图7-87所示电路的功能行为，假设输入信号w是由一个方波信号驱动的。

解：当两个触发器都被清零时，它们的输出是$Q_0 = Q_1 = 0$。在清零输入变高后，输入信号w的每个脉冲都会引起触发器状态如图7-88所示的改变。请注意该图展示了由脉冲上升沿引起的变化已发生后这几个信号的状态。

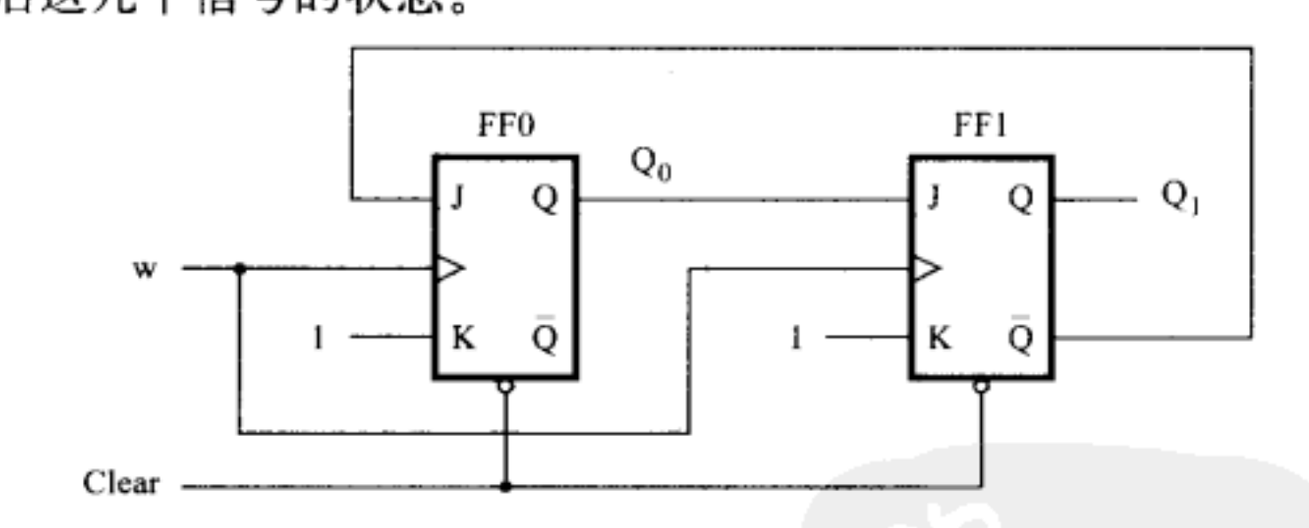


图7-87 例7.18的电路

时间间隔	FF0			FF1		
	J_0	K_0	Q_0	J_1	K_1	Q_1
Clear	1	1	0	0	1	0
t_1	1	1	1	1	1	0
t_2	0	1	0	0	1	1
t_3	1	1	0	0	1	0
t_4	1	1	1	1	1	0

图7-88 图7-87所示电路的行为总结

在连续的几个时钟间隔里，Q_1Q_0 的值按 00，01，10，00，01，10，…序列发生变化。因此，该电路生成了计数序列：0，1，2，0，1，2，…。于是该电路是一个模3计数器。

例7.19 图7-74展示了一个能产生四个时序控制信号T_0，T_1，T_2和T_3的电路。请设计一个能产生六个这样的信号（从T_0到T_5）的电路。

解：图7-74的方案可以利用图7-26所示的模6计数器以及产生6个时序信号的译码器进行扩展。另外一个更为简单的办法是用约翰逊（Johnson）计数器。用三个结构如图7-30所示的D型触发器，可以产生$Q_0Q_1Q_2$的6种取值，如图7-89所示。接着，只用6个二输入与门，如图7-89所示，我们便能得到想要的信号。请注意用约翰逊（Johnson）计数器不可能产生 $Q_0Q_1Q_2$ 等于010 和101 的取值，所以这两种取值被当作无关项处理。

时钟周期	Q_0	Q_1	Q_2	控制信号
0	0	0	0	$T_0=\overline{Q}_0\overline{Q}_2$
1	1	0	0	$T_1=Q_0\overline{Q}_1$
2	1	1	0	$T_2=Q_1\overline{Q}_2$
3	1	1	1	$T_3=Q_0Q_2$
4	0	1	1	$T_4=\overline{Q}_0Q_1$
5	0	0	1	$T_5=\overline{Q}_1Q_2$

图7-89 例7.19的时序信号

例7.20 设计一个能用来控制自动售货机的电路。该电路有五个输入：Q（2角5分），D（1角），N（5分），*Coin*和*Resetn*。当把硬币投入自动售货机时，硬币传感器根据相应的输入（Q, D或者 N）产生一个脉冲。为了标记该事件的发生，自动售货机还在*Coin*线上生成一个脉冲。该电路用*Resetn*低电平有效信号复位。当至少投入30分硬币时，该电路便启动输出Z。如果投入的硬币超过30分，售货机不会自动找钱。

用下面的元件设计电路：一个六位加法器，一个六位寄存器，任意个数的与门、或门和非门。

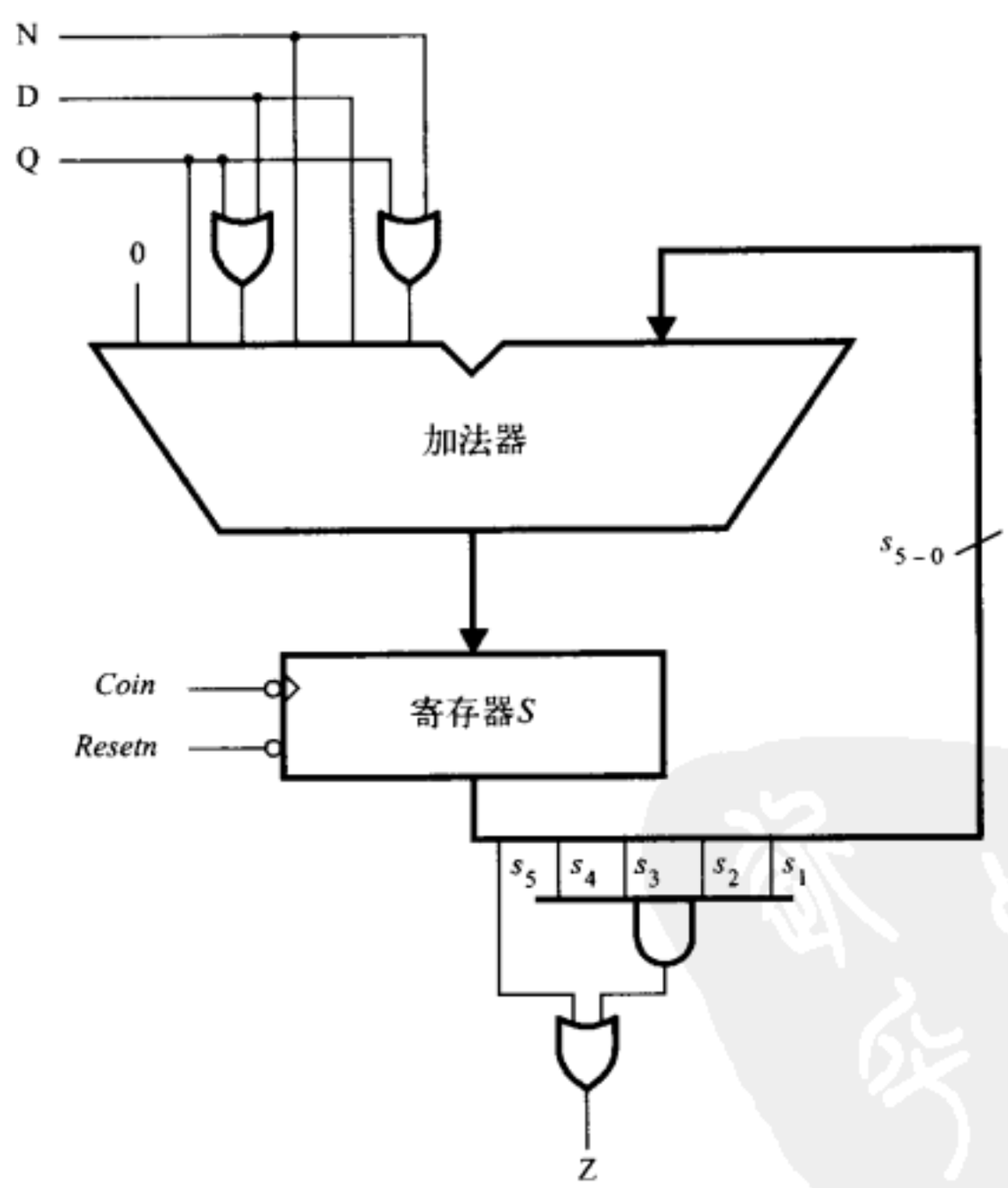


图7-90 例7.20的电路

解：图7-90所示为自动售货机电路的一种方案。每个硬币的值用对应的五位二进制数表示。投入硬币的值被加到当前的总值中，总值保存在寄存器S中，所需要的输出Z的表达式如下：

$$Z = s_5 + s_4 s_3 s_2 s_1$$

该寄存器是由*Coin*信号的负跳变沿触发的。这就允许经由加法器的传播延迟，可确保正确的求和值能存入该寄存器。

在第9章里，我们将展示如何才能用更结构化的途径来设计这种类型的控制电路。

例7.21 写出图7-90所示电路的Verilog代码。

解：图7-91列出了该电路的Verilog代码。

```
module vend (N, D, Q, Resetn, Coin, Z);
   input N, D, Q, Resetn, Coin;
   output Z;
   wire [4:0] X;
   reg [5:0] S;

   assign X[0] = N | Q;
   assign X[1] = D;
   assign X[2] = N;
   assign X[3] = D | Q;
   assign X[4] = Q;
   assign Z = S[5] | (S[4] & S[3] & S[2] & S[1]);

   always @(negedge Resetn, negedge Coin)
      if (Resetn == 1'b0)
         S <= 5'b00000;
      else
         S <= {1'b0, X} + S;

endmodule
```

图7-91 例7.21的代码

例7.22 在7.15节，我们曾介绍了图7-85所示的计数器电路的时序分析。重新设计该电路，想办法减少触发器之间的逻辑延迟，以使该电路能以更高的时钟频率运行。

解：正如7.15节所介绍的那样，该计数器电路的性能由于级连与门的延迟而受到限制。为了提高电路的性能，可以按照图7-92的说明重新安排与门的位置。在这个重新设计的电路中，最长的延迟路径（起始于触发器Q_0结束于Q_3）提供了最小的时钟周期。

$$\begin{aligned} T_{min} &= t_{cQ} + t_{AND} + t_{XOR} + t_{su} \\ &= 1.0 + 1.4 + 1.2 + 0.6\text{ns} = 4.2\text{ns} \end{aligned}$$

重新设计的计数器的最高时钟频率可以达到：F_{max} = 1/4.2ns = 238.1MHz，而原来的计数器的时钟频率只有156.25MHz。

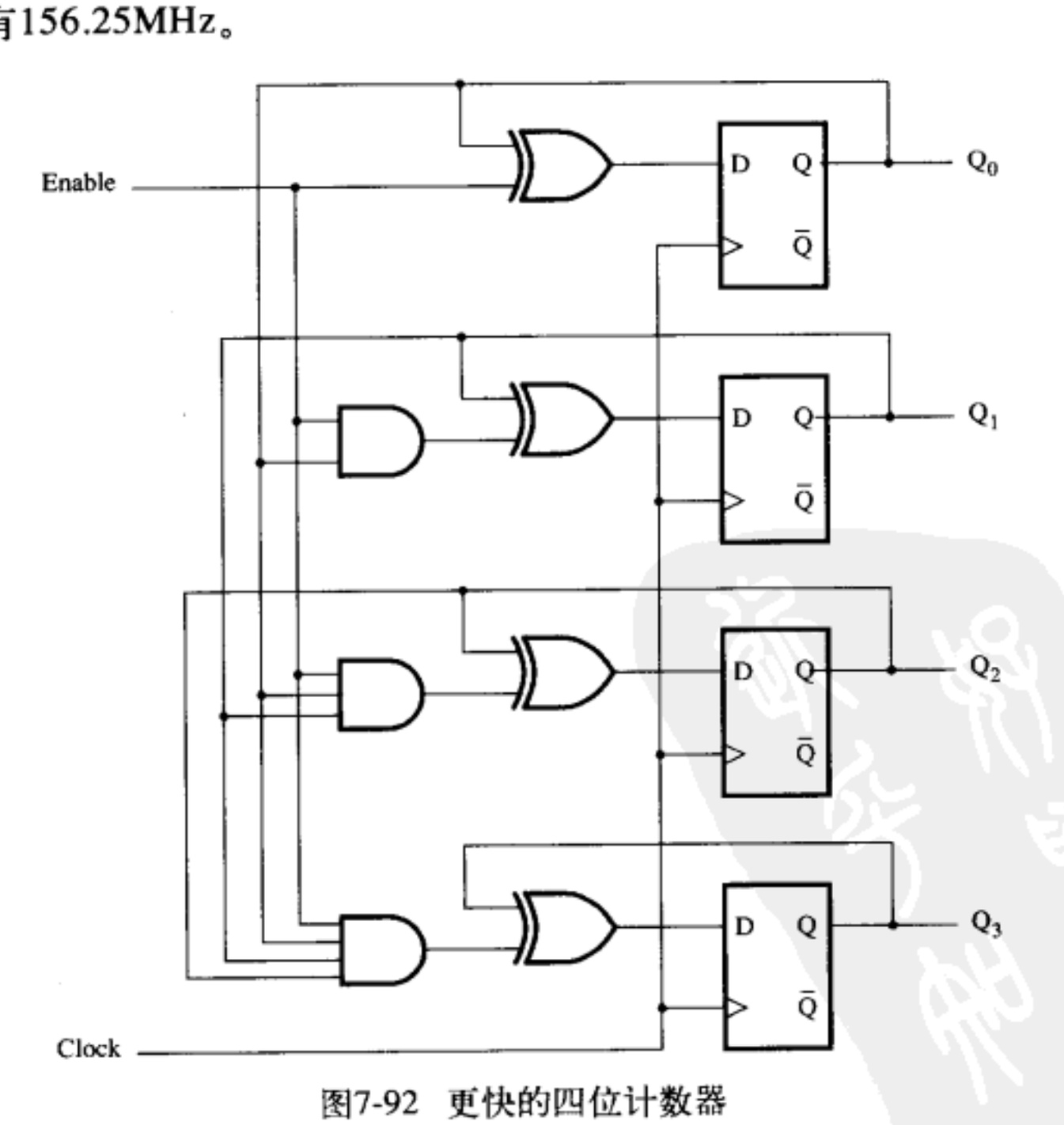
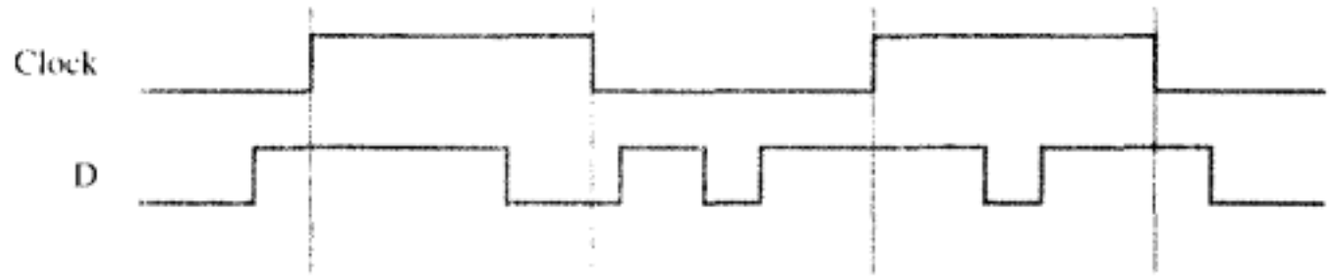


图7-92 更快的四位计数器

练习题

标有*号的习题答案参见书后。

7.1 假设将图P7-1所示的信号D和$Clock$输入到图7-12所示的电路，请画出信号Q_a，Q_b和Q_c的波形图。



图P7-1 练习题7.1的时序图

7.2 图7-3所示的电路能否修改成SR锁存器？请对您的回答作出解释。

7.3 图7-5所示为一个用或非门构成的锁存器。画出用与非门构成的功能相同的锁存器。写出其特征表，画出时序图。

***7.4** 画出只用与非门实现的门控SR锁存器的电路图。

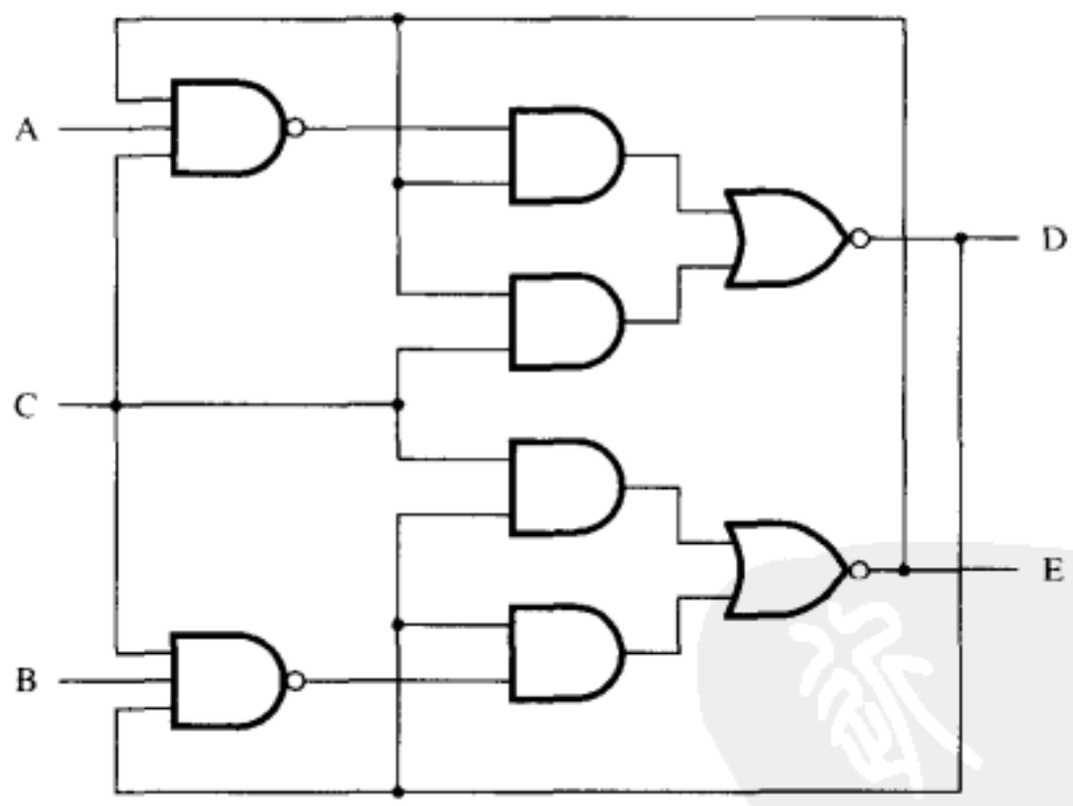
7.5 假设有一个100MHz的时钟，画出用D触发器产生50MHz和25MHz时钟信号的电路，并画出这三个时钟信号的时序图，假定有合理的延迟。

***7.6** SR触发器是一个有置1和清0输入端的类似于门控SR锁存器的触发器，画出用D触发器和其他逻辑门如何构成SR触发器的电路图。

7.7 当时钟信号变为0时，若输入信号S和R均为1，则图7-6所示的门控SR锁存器的行为不能预测。解决这个问题的一种方法是创建一个倾向于置1的门控SR锁存器，该锁存器在$S = R = 1$的条件下，使得锁存器的输出为1。请设计倾向于置1的门控SR锁存器，并画出其电路。

***7.8** 画出如何用T触发器和其他逻辑门构成JK触发器的电路图。

7.9 考虑图P7-2所示电路。假设两个与非门比图中其他逻辑门有更长得多的延迟时间（大约四倍）。那么这个电路与本章中曾经讨论过的电路有什么不同？



图P7-2 练习题7.9的电路图

7.10 写出带异步清零输入的T触发器的Verilog代码，请用行为描述，不要用结构描述。

7.11 写出表示JK触发器的Verilog代码，请用行为描述，不要用结构描述。

7.12 用CAD工具综合练习题7.11编写的Verilog代码。对综合后产生的电路进行仿真，展示能验证其功能的时序图。

7.13 万能（universal）移位寄存器可以从左向右移位，也可以从右向左移位，并且还具有并行加载

的能力。请画出这样的移位寄存器的电路图。

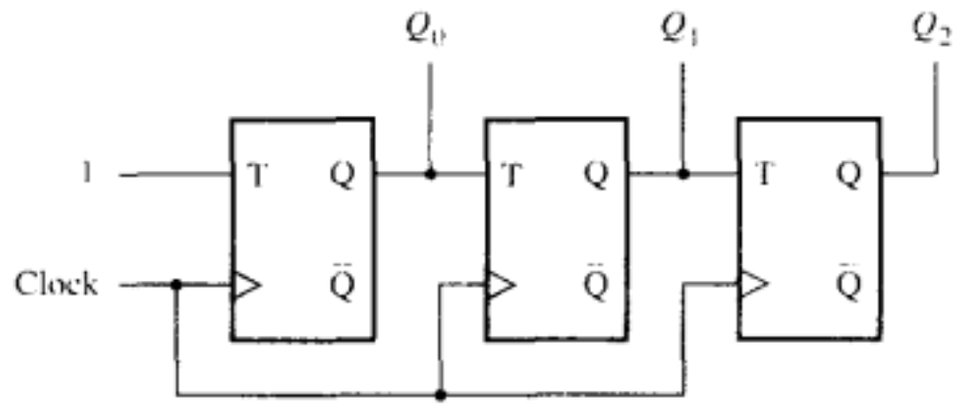
7.14 写出n位移位寄存器的Verilog代码。

7.15 设计一个四位带并行加载的同步计数器。使用T触发器，不要使用7.9.3节使用的D触发器。

***7.16** 用T触发器设计一个三位的递增/递减计数器。该计数器应该包含 $\overline{UP}$/Down 控制输入。若 $\overline{UP}$/Down = 0，则该电路为递增计数器，若 $\overline{UP}$/Down = 1，则该电路为递减计数器。

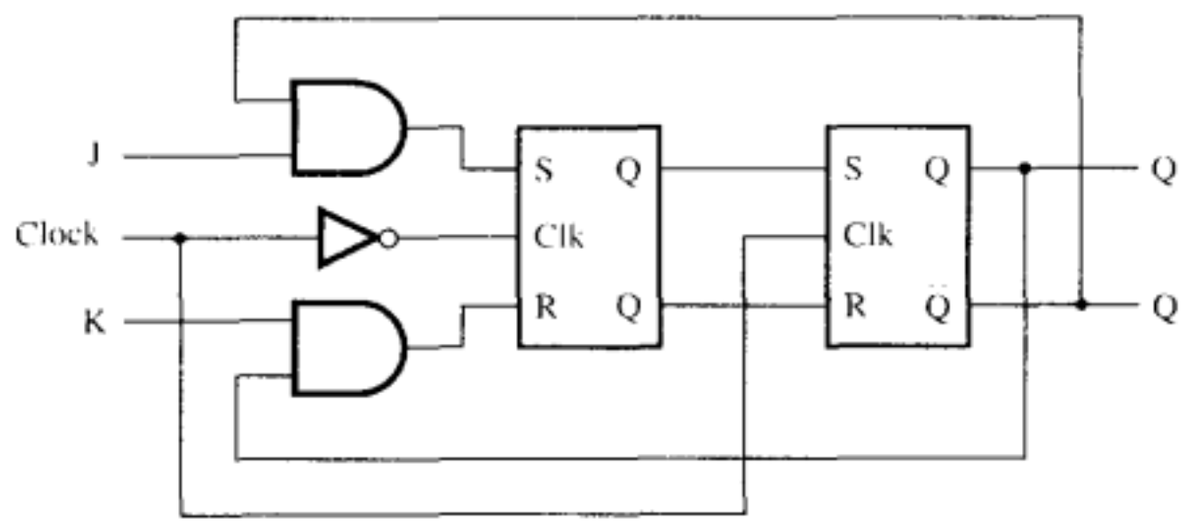
7.17 用D触发器设计重新做一遍练习题7.16。

***7.18** 图P7-3所示的电路看似一个计数器。该电路产生的序列是什么？



图P7-3 练习题7.18的电路图

7.19 考虑图P7-4所示的电路。该电路与图7-17所示电路相比较有什么不同？这两个电路能否用于同一个目的？如果不能，它们之间关键的不同点是什么？



图P7-4 练习题7.19的电路图

7.20 用或非门构建一个类似图7-11a所示的电路，使其功能等价于负跳变沿触发的D触发器。

7.21 编写描述带同步复位端的模12递增计数器的Verilog代码。

***7.22** 对图7-25所示的计数器中的触发器，假设其建立时间 t_{su} = 3ns，保持时间 t_h = 1ns，并且经过触发器的传播延迟为1ns。假设每个与门、异或门和2选1多路器的传播延迟都等于1ns，该电路能正确操作的最大时钟频率是多少？

7.23 编写代表8位Johnson计数器的Verilog代码。用CAD工具综合该代码，进行时序仿真，观察计数时序。

7.24 用图7-55所示代码的风格编写表示环型计数器的Verilog代码。您的代码中应该使用参数n，用它来设置计数器中触发器的个数。

***7.25** 编写Verilog代码，描述图7-48所示电路的功能。

7.26 图7-69列出了Verilog代码，该代码描述了可以把两个寄存器$R1$和$R2$中的内容通过暂存的寄存器$R3$进行交换的数字系统。用你的CAD工具为该系统创建一个等价的电路图。综合此原理图的电路，进行时序仿真。

7.27 用图7-63所示的控制电路，重新做一遍练习题7.27。

7.28 修改图7-71所示代码，采用图7-63所示的控制电路。对该代码进行综合并在芯片上实现，进行

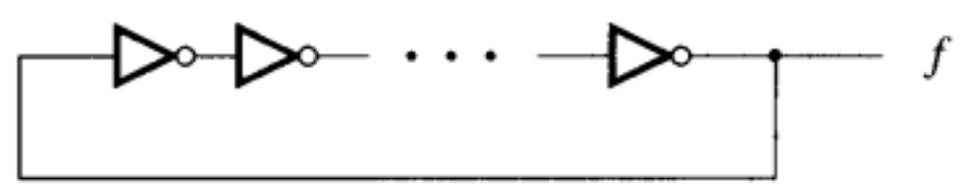
时序仿真（观察其波形）。

7.29 在7.14.2节中，我们设计了一个处理器，该处理器能进行表7-3所列出的操作。设计一个改进的电路，该电路可进行*Rx*和*Ry*互换的操作。该操作将寄存器*Rx*和*Ry*的内容互换。因为需要进行5种类型的操作，而不是4种，所以用三位的信号 $f_2 f_1 f_0$ 来表示输入信号*F*，见图7-75所示。在系统中添加一个新的寄存器*Tmp*，在交换操作期间用来暂时存储数据。如7.14.2节曾做过的那样，写出表示控制电路输出的逻辑表达式。

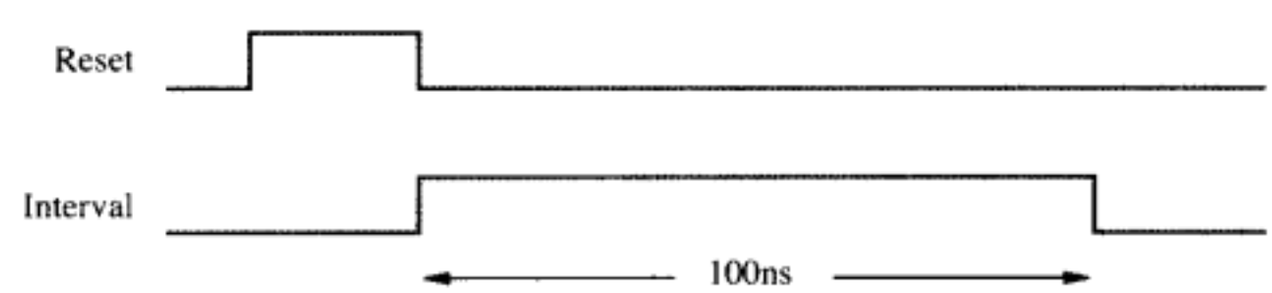
7.30 环型振荡器是一个由奇数（*n*）个反相器连接成的环状的电路，见图P7-5所示。每个反相器的输出都是具有确定频率的周期信号。

(a) 假设所有反相器完全一致，则它们的延迟 t_p 也都相等。令某个反相器的输出信号为*f*，求用 *n* 和 t_p 来表示信号 *f* 的周期的表达式。

(b) 请设计一个电路，可以用实验手段测量环型振荡器中某个反相器的延迟t_p。假设存在复位信号*Reset*和另一个信号*Interval*，这两个信号的时序如图P7-6所示。信号*Interval* 维持为1值的时间长度已知。假设该时间长度为100ns。请用*Reset*和*Interval*信号以及来自于(a)题的信号*f* 设计一个电路，可以用实验手段测量反相器的延迟t_p。在设计中，你可以使用逻辑门和子电路，例如加法器、触发器、计数器、寄存器等。

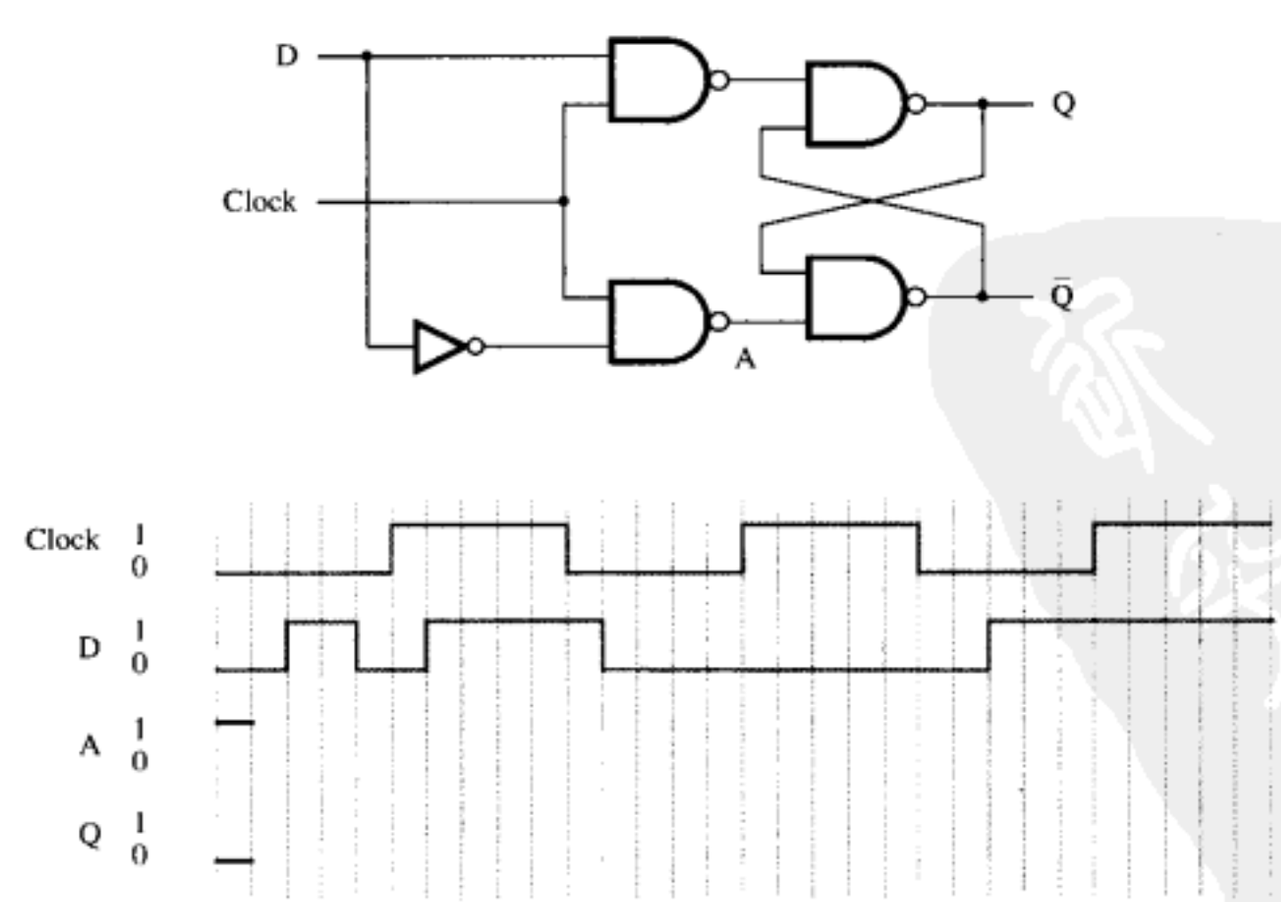


图P7-5 环型振荡器



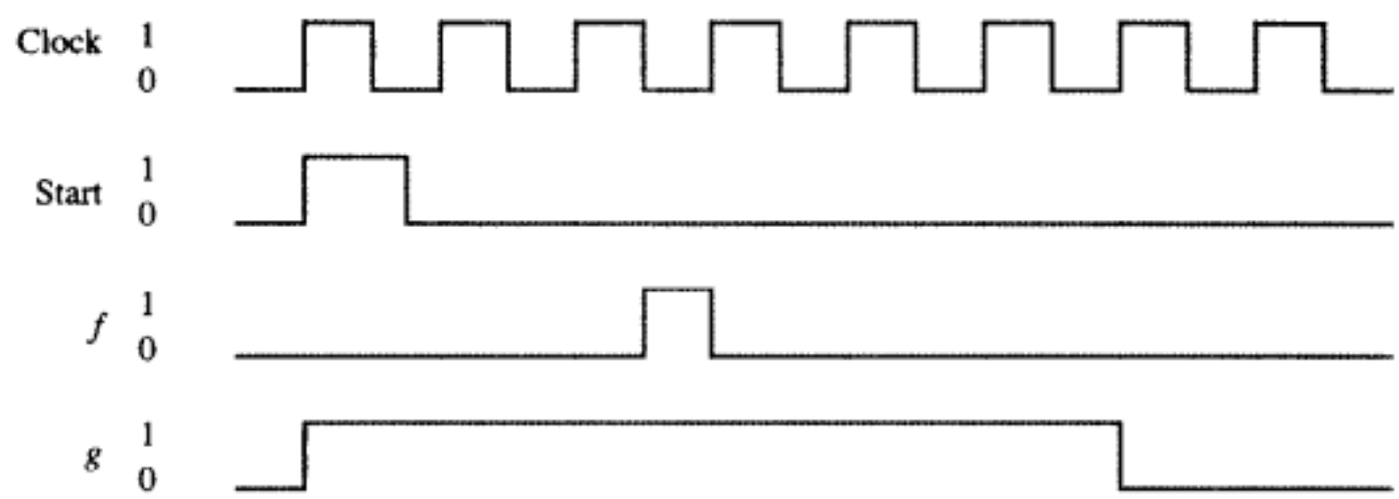
图P7-6 练习题7.30的信号时序图

7.31 门控D锁存器的电路见图P7-7所示。假设通过与非门或者反相器的延迟是1ns。请完成图P7-7中尚未画出的信号时序，图中的时间分辨度为1ns。



图P7-7 练习题7.31的电路和时序图

*** 7.32** 某逻辑电路有两个输入 *Clock* 和 *Start*，两个输出 *f* 和 *g*。该电路的行为用图P7-8所示时序图描述，当*Start*输入端收到一个脉冲，该电路随即在输出端 *f* 和 *g* 产生脉冲信号，具体波形见图P7-8所示时序图。请你只用带复位输入的由时钟正跳变沿触发的三位同步计数器和基本门电路，设计一个能产生以上时序的合适电路。在设计中，可假设所有逻辑门和计数器的延迟都是可忽略的。



图P7-8 练习题7.32的时序图

7.33 下列代码可以检查 *n* 位向量中相邻的1。

```
always @(A)
begin
   f = A[1] & A[0];
   for (k = 2; k < n; k = k+1)
      f = f | (A[k] & A[k-1]);
end
```

这段代码使用的是阻塞赋值，产生的逻辑函数为 $f = a_1 a_0 + \cdots + a_{n-1} a_{n-2}$。若我们把这段代码中的阻塞赋值改成非阻塞赋值，则该电路产生的逻辑函数是什么？

7.34 图P7-9所示的Verilog代码表示一个3位的线性反馈移位寄存器（LFSR）。这种类型的电路生成的伪随机数序列每2^n-1 个时钟周期后重复出现，*n* 表示反馈移位寄存器（LFSR）中触发器的个数。把代码综合成电路并在芯片中实现该LFSR。画出其电路原理图。在该LFSR中加载 001，接着启动该寄存器计数，产生的计数序列是什么？

7.35 用图P7-10所示的Verilog代码重新做一遍练习题7.34。

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
         Q <= {Q[2], Q[0] ^ Q[2], Q[1]};

endmodule
```

图P7-9 线性反馈移位寄存器的代码

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
         Q <= {Q[2], Q[0], Q[1] ^ Q[2]};

endmodule
```

图P7-10 线性反馈移位寄存器的代码

7.36 图P7-11所示的Verilog代码除了使用了阻塞赋值外，等价于图P7-9所示的代码。画出该代码表示的电路图，产生的计数序列是什么？

7.37 图P7-12所示的Verilog代码除了使用了阻塞赋值外，等价于图P7-10所示的代码。画出该代码表示的电路图，产生的计数序列是什么？

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] = Q[2];
         Q[1] = Q[0] ^ Q[2];
         Q[2] = Q[1];
      end

endmodule
```

图P7-11 练习题7.36的代码

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] = Q[2];
         Q[1] = Q[0];
         Q[2] = Q[1] ^ Q[2];
      end

endmodule
```

图P7-12 练习题7.37的代码

参考文献

1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed., (Morgan Kaufmann: San Francisco, Ca., 2004).
3. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
4. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
6. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, N.J., 2002).
7. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, N.J., 1997).
8. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
9. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, N.J., 1995).
10. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, Ma., 1993).
11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, N.J., 1986).
12. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 2001).
13. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
14. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
15. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).

16. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
17. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
18. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
19. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

第8章　同步时序电路

在前面几章中，我们考虑了输出完全由当前输入值决定的组合逻辑电路，还讨论了如何以触发器的形式来实现简易的存储元件。触发器的输出值取决于触发器的状态，而不是任何给定时刻的输入值；但输入值确实是造成触发器状态可能发生改变的原因。

在本章中，我们将介绍一种常见的电路类型，这种电路的输出取决于电路过去的行为和当前的输入值。这种电路称为时序电路。大多数情况下，我们用时钟信号来控制时序电路的操作，这种电路称为同步时序电路。我们也可以不用时钟信号来控制电路，这种电路称为异步时序电路。同步电路比较容易设计，并且应用在大量的实际电路中。这就是本章要介绍的内容，异步电路将在第9章中介绍。

同步时序电路是由组合逻辑和一个或多个触发器实现的。常见的同步时序电路的构造如图8-1所示。该电路有一组基本输入信号W，并产生一组输出信号Z。触发器的输出值就是电路的状态Q。在时钟信号的控制下，触发器输出状态的改变取决于馈入这些触发器输入端的组合逻辑。这样电路就从一种状态转变到了另一种状态。为了确保在一个时钟周期内只发生一次状态变化，触发器必须是沿触发类型的。这种类型的触发器可以由时钟的正跳变沿（0到1）触发，也可以由负跳变沿（1到0）触发。我们将使用术语有效时钟沿来指代引起状态改变的时钟沿。

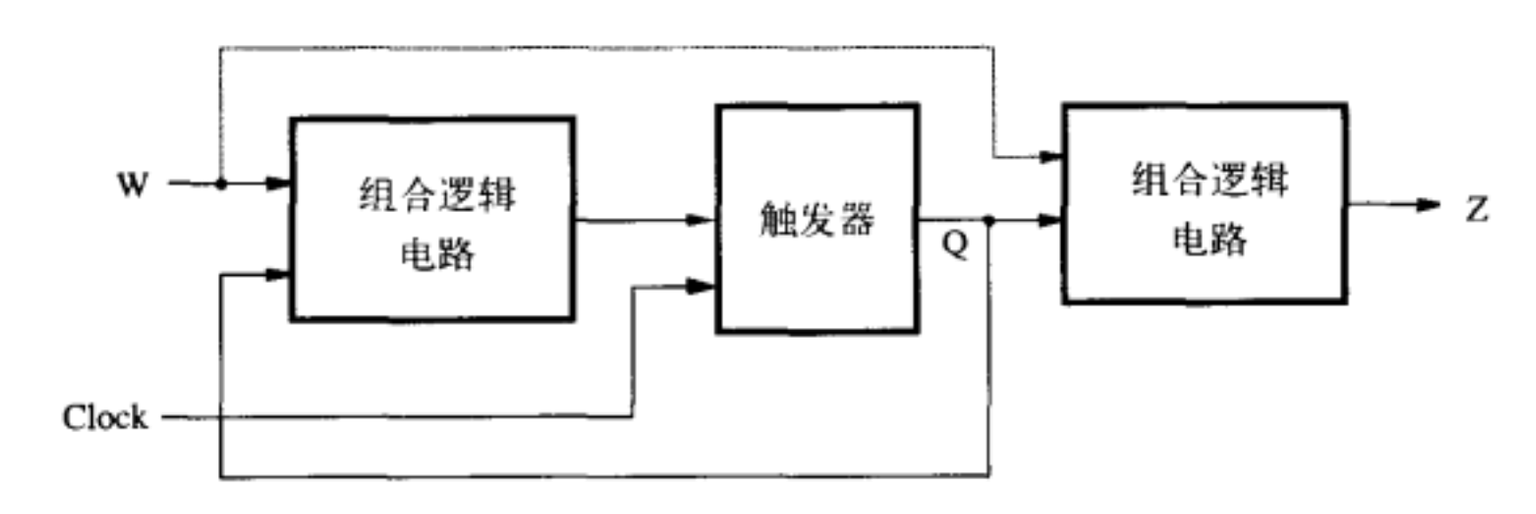


图8-1　时序电路的一般形式

提供触发器输入信号的组合逻辑有两个输入源：原始输入W和触发器当前的输出Q。因此触发器状态的改变取决于当前状态和原始输入值。

图8-1表明时序电路的输出是由另一个组合电路产生的，即输出是触发器的当前状态和原始输入共同作用的结果。尽管输出总是依赖于当前状态，但输出不必直接依赖于原始输入。因此，图中灰色的连接线可能有也可能没有。为了区分这两种可能情况，我们习惯上把输出只依赖于当前状态的时序电路称为摩尔（Moore）型电路，把同时依赖于当前状态和原始输入的时序电路称为米利（Mealy）型电路。之所以这样命名是为了纪念Edward Moore和George Mealy这两位先驱，他们在20世纪50年代研究了这类电路的行为特性。

时序电路也被称为有限状态机（FSM），这种更正式的称谓经常出现在技术文献之中。之所以这样称呼这类电路，是因为它们的功能行为可以用有限的状态个数来表示。我们以后将经常用有限状态机或状态机这两个术语来称呼本章中所涉及的时序电路。

8.1 基本设计步骤

我们将通过一个简单的例子来介绍时序电路的设计方法。假设我们想要设计一个满足下列要求的电路：

1) 电路有一个输入w和一个输出z。

2) 电路的所有变化均发生在时钟信号的正跳变沿。

3) 如果在两个连续的时钟周期内输入w等于1，那么输出z等于1，否则z值为0。

因此，该电路可以检测出在输入信号w中是否有两个或两个以上连续的1出现。这种能对输入信号中特定码流的出现进行检测的电路通常称为序列检测器（sequence detector）。

从上面的说明可以明显地看到，输出z不仅取决于当前的w值。为了说明这一点，我们在11个时钟周期内考虑w和z信号值的序列，如图8-2所示。任意取w值，输出z值是根据要求产生的。这个输入值和输出值序列说明，对于一个给定的输入值，输出值可能是0或1。例如在t_2和t_5时钟周期，$w = 0$，但在t_2时$z = 0$，而在t_5时$z = 1$。类似地，在t_1和t_8时$w = 1$，但在t_1时$z = 0$，而在t_8时$z = 1$。这就意味着z不是只由当前输入值w决定的，电路中必定还存在某些不同的状态，z的值还取决于这些状态。

时钟周期：	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

图8-2 输入和输出信号的序列

8.1.1 状态图

设计有限状态机的第一步是确定需要多少个状态，并且哪几种状态转变（从一种状态到另外一种状态）是可能发生的。完成这项工作没有固定的方法，但设计者一定要认真地思考状态机必须完成什么功能。较好的方法是选择某个特殊状态作为*初始状态*，即当开启电源或者施加复位信号时电路应该进入的状态。举个例子，假设初始状态为A，只要输入w为0，电路不需要作任何反应，这样每一个有效时钟沿时刻电路都保持在A状态。当$w = 1$时，状态机需要识别出这一改变，转变到一个不同的状态，我们称该状态为B。这种转变是在$w = 1$后的下一个有效时钟沿时刻发生的。在状态B下，因为没有在两个连续的时钟周期内出现$w = 1$的情况，与状态A类似，电路保持输出值$z = 0$。在状态B下，如果在下一个有效时钟沿$w = 0$，电路应该重新回到状态A；如果在下一个有效时钟沿$w = 1$，电路应该转变到第三种状态C，并且产生输出$z = 1$。只要保持输入$w = 1$，电路就一直持续状态C，并继续维持输出$z = 1$。当w变为0时，状态机应该重新回到状态A。既然前面的描述已经考虑了状态机在不同状态下所有可能遇到的输入值w，我们可以推断要实现期望的状态机必须具有三种不同状态。

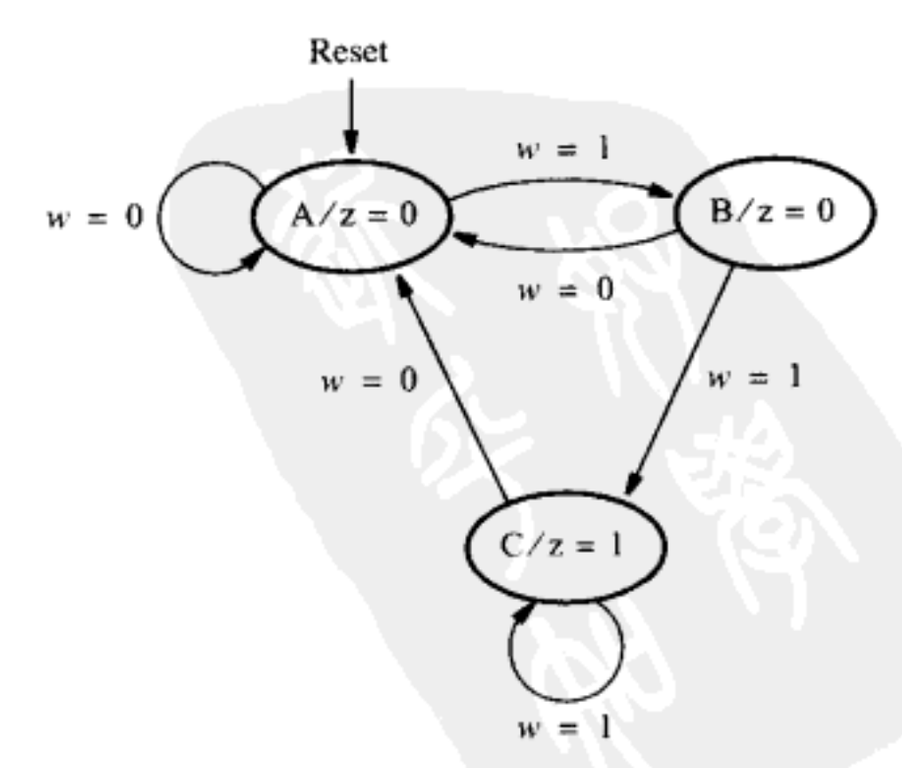


图8-3 简单时序电路的状态图

至此，我们已经用一种非正规的方式确定了状态之间可能发生的转变，下面将用一种更规范的步骤来设计相应的时序电路。时序电路的行为可以用很多不同的方法来描述，概念上最简单的方法是用状态图表示。状态图把电路状态表示为节点（用圆形表示），状态之间的转移用曲线表示。图8-3所示的状态图定义

了我们用文字详细说明的行为，A、B和C三种状态在图中用节点的形式表示。节点A代表初始状态，也是电路在施加输入$w = 0$值后将进入的状态。在这一状态下，输出z只能为0，在节点内表示为$A/z = 0$。只要$w = 0$，电路始终处于状态A，我们用从该节点出发并回到该节点的标有$w = 0$的圆弧表示。从状态A转变到状态B的转移记录了（在$w = 0$情况之后）首次出现$w = 1$的情况，用从状态A出发指向状态B的弧线表示，弧线上的标签$w = 1$表示了这一转变的条件。在状态B下，输出始终为0，在节点内表示为$B/z = 0$。

当电路处于状态B时，如果w在下一个有效时钟沿内仍旧等于1，B状态将会转移到状态C。在状态C下，输出$z = 1$，如果在接下来的时钟周期内保持$w = 1$，电路将仍旧留在C状态，并保持$z = 1$。然而，不论电路处在状态B还是状态C，一旦w变为0，在下一个有效时钟沿时刻，都将转移到状态A。

在图中我们明确标明复位（Reset）输入信号是用来强迫电路进入状态A的，而不管电路当时所处的状态是什么。我们本来可以将复位信号当作输入该电路的一个外来信号，且当复位信号（Reset）输入时，无论电路处于什么状态，都立即转移到起始状态A（如果这样做状态图上需要多画许多转移线——译者注）。但这样做会使得状态图变得不必要地复杂。有限状态机中的状态是用触发器实现的。正如第7章中讨论过的一样，触发器通常具有复位能力，我们可以利用复位（Reset）信号的这个能力将所有的触发器一起清零。在图8-3中我们采用了简化的方法来表示复位信号的作用，从而使得状态图尽可能地清晰易懂。

8.1.2 状态表

虽然状态图提供了一种易于理解和进一步实现的描述时序电路行为的方法，但是把状态图中所包含的信息用表格的形式表示出来也非常方便。图8-4展示了上述时序电路的状态表。这个表格表示了对于不同的输入信号w，电路从每一个当前状态到下一个状态的转变。需要注意的是，输出z是当前状态下指定的输出，即电路在当前时间内产生的输出。

当前状态	下一状态		输出 z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

图8-4 图8-3时序电路的状态表

现在介绍最终电路的设计步骤。为了更好地理解基本的设计概念，我们先浏览一下用手工完成每个设计阶段的传统方法。接着再讨论利用现代计算机辅助设计（CAD）工具的自动化设计技术。

8.1.3 状态分配

图8-4的状态表用A，B和C三个字母定义了三种状态。在逻辑电路的实现过程中，每一个状态都用状态变量的特定组合取值来表示。每个状态变量可以用一个触发器实现。因为必须实现三种不同的组合状态，使用两个状态变量就足够了。令这两个状态变量分别为y_1和y_2。

现在把图8-1所示的总体方块图修改成我们的例子，见图8-5所示，来说明实现符合要求的状态机的电路结构。两个触发器分别表示状态变量的两位。在这个图中，我们没有定义触发器的类型，这个问题将会在下一小节说明。由图8-3和图8-4可知，输出z只由电路的当前状态决定，这样图8-5中的方块图说明z是一个只取决于y_1和y_2的函数，我们的设计是摩尔型。我们需要设计一个组合电路，用y_1和y_2作为输入信号，对所有可能的输入信号的取值，产生正确的输出信号。

信号y_1和y_2还反馈给输入组合电路，由输入组合电路的输出决定有限状态机的下一个状态。这个组合电路仍然使用原先的输入信号w，输出为两个信号Y_1和Y_2，用来设置触发器的状态。每个有效时钟沿将会引起触发器的状态向Y_1和Y_2转变。因此，Y_1和Y_2被称为下一个状态

变量，y_1 和y_2 被称为当前状态变量。我们需要设计一个组合电路，其输入为w，y_1 和y_2，并且对于这些输入信号的所有组合值，输出Y_1 和Y_2 将引起状态机向下一个满足设计要求的状态转变。设计过程的下一步是创建定义该电路（行为）的真值表，以及产生信号z的电路。

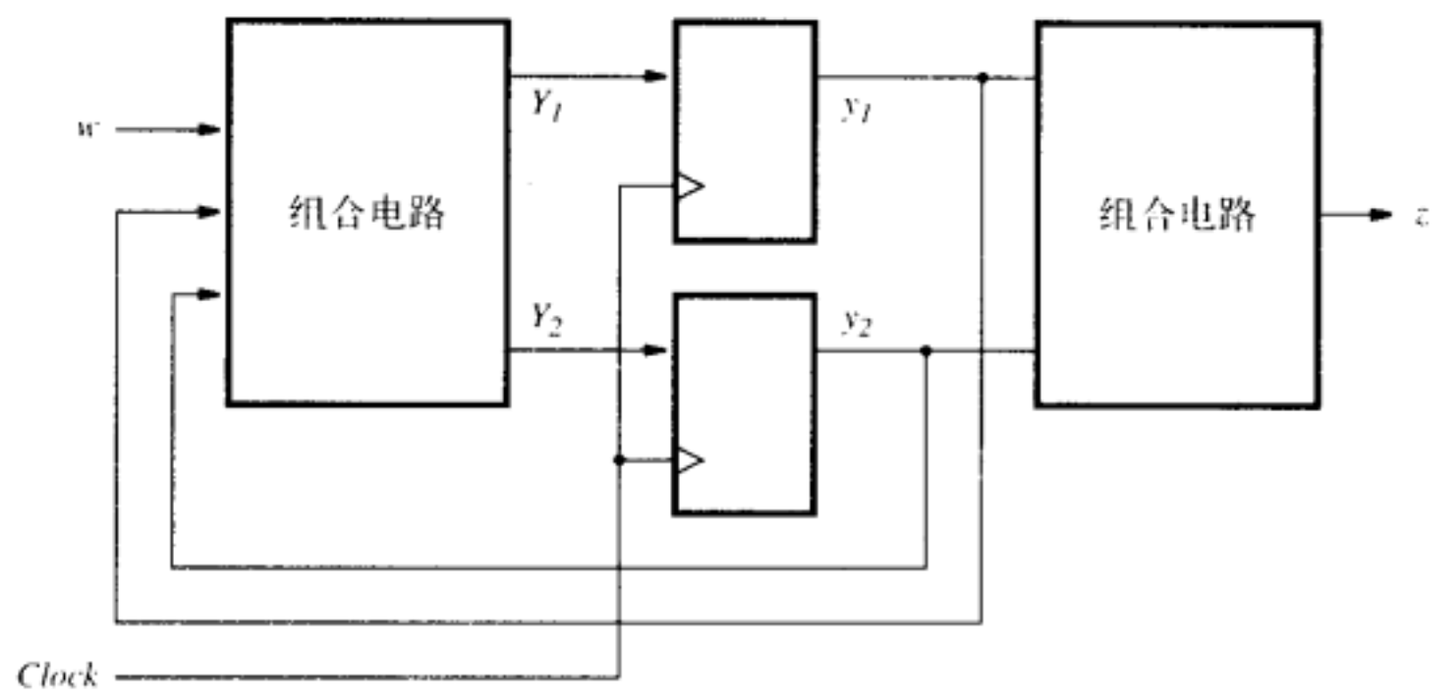


图8-5 由输入w、输出z和两个状态触发器组成的时序电路的总体方块图

为了生成理想的真值表，我们给每个状态指定一组由变量y_1 和y_2 组成的值。图8-6给出了一种可能的赋值，状态A，B，C分别用y_2y_1 = 00，01和10表示。第四种情况y_2y_1 = 11在这个例子中是不需要的。

图8-6所示的表格类型通常称为状态分配表。这个表可以直接用作输入为y_1 和y_2，输出为z的真值表。尽管对于下一个状态函数Y_1 和Y_2，该表没有正规真值表的外观，这是由于表中每个w值对应着分开的两列，但很显然，该真值表已经包含了由输入信号w，y_1 和y_2 的组合逻辑定义的下一个状态函数的所有信息。

	当前状态	下一状态		输出
		w = 0	w = 1	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

图8-6 图8-4对应的时序电路的状态分配表

8.1.4 触发器的选择以及下一个状态和输出表达式的推导

根据图8-6所示的状态分配表，我们可以推导出表示下一个状态和输出函数的逻辑表达式。但我们必须首先确定电路中所使用的触发器类型。最简单的选择是使用D触发器，因为在这个例子中，只是将Y_1 和Y_2 的值在有效时钟沿时刻存入触发器，并产生新的值y_1 和y_2 而已。换句话说，如果把触发器的输入信号称为D_1 和D_2，这些信号与Y_1 和Y_2 是一样的。请注意，在图8-5中的方块图正好对应于使用D触发器的情形。而对于其他类型的触发器，如JK触发器，下一个状态变量和触发器的输入值之间没有那么直接的关系，我们将在8.7节中讨论这一情况。

必要的逻辑表达式的推导过程见图8-7所示。我们采用卡诺图方法，以便于读者验证表达式的正确性。上节提到在图8-6中，我们只需要用二变量四种可能组合中的三种情况来表示状态。第四种情况y_2y_1 = 11在该电路中是永远不会发生的，因为电路只限定在A，B和C三种状态下转变，因此我们把第四种情况视为无关情况。这种无关情况在卡诺图中用d表示。考虑到无关情况可以简化表达式，得到：

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = w(y_1 + y_2)$$
$$z = y_2$$

如果不考虑到无关情况，得到的表达式会比较复杂。如图8-7中的中间阴影区域。

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	1	0	d	0

不考虑无关情况：$Y_1 = w\bar{y}_1\bar{y}_2$

考虑无关情况：$Y_1 = w\bar{y}_1\bar{y}_2$

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	0	1	d	1

不考虑无关情况：$Y_2 = wy_1\bar{y}_2 + w\bar{y}_1y_2$

考虑无关情况：$Y_2 = wy_1 + wy_2 = w(y_1 + y_2)$

$y_2 \backslash y_1$	0	1
0	0	0
1	1	d

$z = \bar{y}_1y_2$

$z = y_2$

图8-7 图8-6所示时序电路逻辑表达式的推导

既然$D_1 = Y_1$，$D_2 = Y_2$，与上面表达式对应的逻辑电路可以用如图8-8所示的电路来实现。注意，该电路中包含了一个时钟信号和一个低电平有效的复位信号。把触发器的清零输入与外部的*Resetn*信号连接起来，如图所示，这样就提供了一个简单手段以便强迫电路进入已知的状态。若我们给电路施加*Resetn* = 0，则两个触发器都将被清除为0，使有限状态机进入状态y_2y_1 = 00。

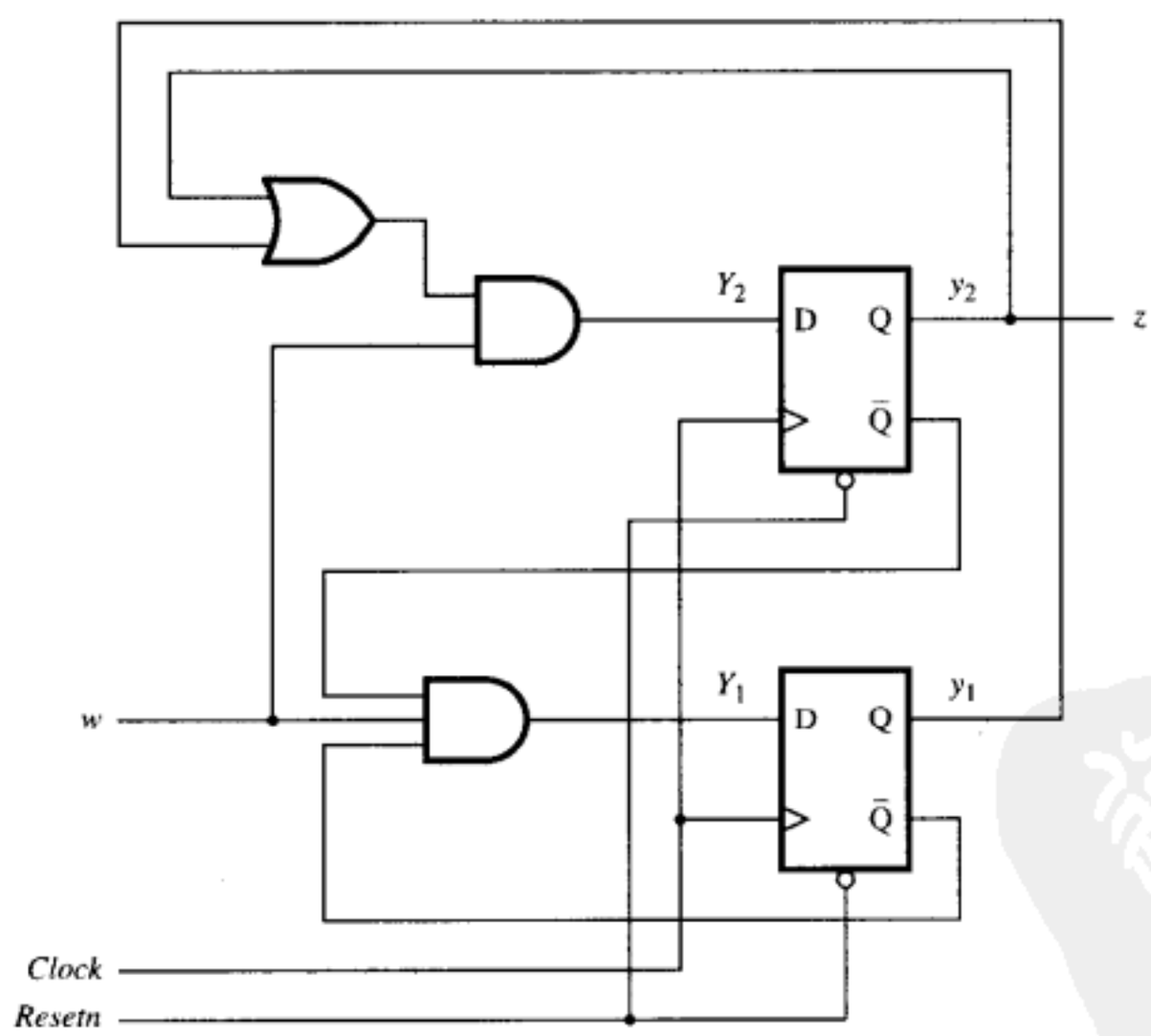


图8-8 与图8-7对应的时序电路的最终实现

8.1.5 时序图

为了完全理解图8-8所示电路的工作情况，让我们考虑其时序图，见图8-9所示。该图描绘了与图8-2所示序列值对应的信号波形。

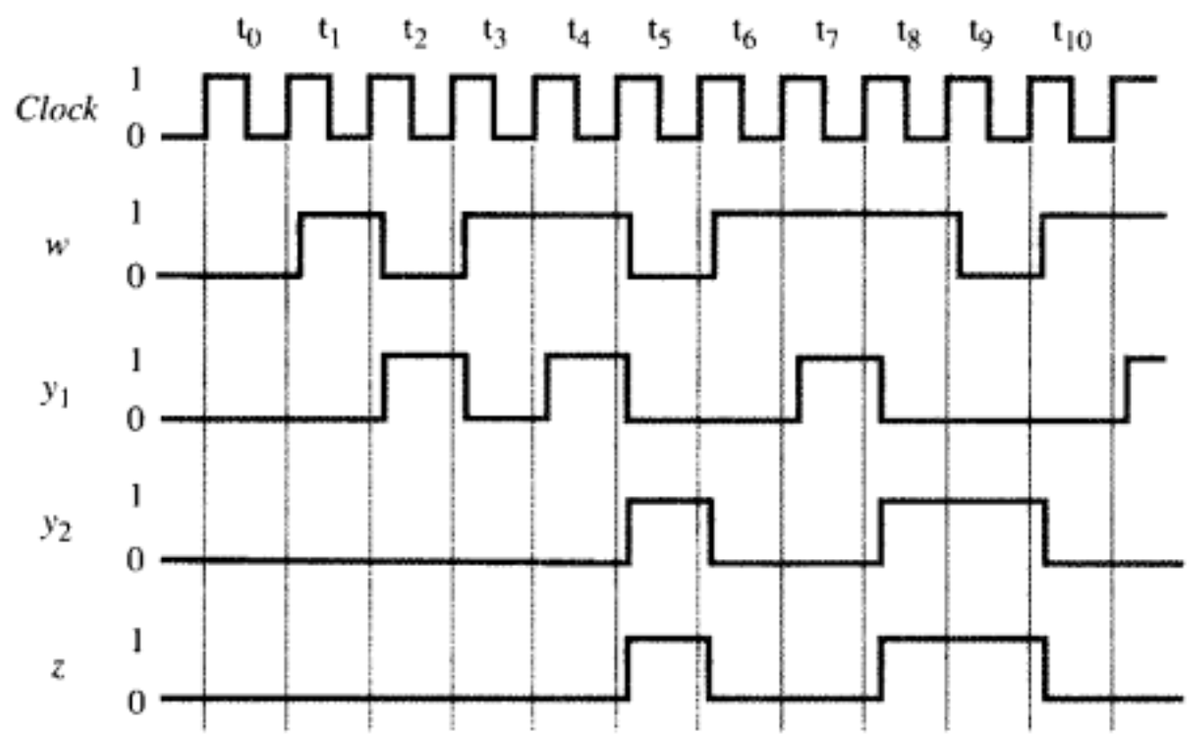


图8-9 图8-8电路的时序图

由于我们采用的是正跳变沿触发的触发器，所有信号的改变均发生在时钟正跳变沿时刻后的瞬间。由时钟沿引起的延迟时间取决于信号通过触发器时的传播延迟。请注意，图中输入信号w也是在有效时钟沿后的瞬间才发生变化的。这一假设是很合理的，因为在典型的数字系统内，输入信号（例如w）也许是另外一个使用相同时钟源的同步电路的输出。我们将在10.3节讨论输入信号与时钟信号的同步问题。

需要注意的关键点是：在有效时钟沿过后的很短时间w就会发生变化，因而使得w值在几乎完整的时钟周期内等于1（或0），在此周期内电路不会有任何改变，直到下一个时钟周期开始时，正跳变时钟沿引起触发器改变其状态。因此，w值必须在两个连续的时钟周期内为1，才能使电路进入状态C，进而产生输出$z = 1$。

8.1.6 设计步骤小结

我们可以把设计同步时序电路的步骤总结如下：

1) 取得想要设计的电路的设计需求说明。

2) 先选取一个起始状态，由此逐步推导出状态机。然后，确定电路的技术说明，考虑电路输入变量的所有取值情况，为了对输入取值变化作出响应，应添加必要的新状态。画出状态转移图以便跟踪访问过的状态。把这件工作完成后，便能得到画有状态机所有经历过的状态，并标出从一个状态到另外一个状态的转移条件的完整的状态图。

3) 根据状态图建立一个状态表。为了方便，也可以不先画出状态图而直接创建状态表。

4) 在前面所举的时序电路示例中，状态机只有三个状态；因此，建立这样一个没有更多状态的状态表是很容易的。然而，在实践中经常遇到需要处理大量状态的电路。在这种情况下，不可能一下子就能得到效果最佳的状态表。几乎可以肯定地说，创建的状态往往比真正需要的多。这个问题可以在状态最简单化的过程中不断地加以修正。我们将在8.6节中深入讨论状态最简化的过程。

5) 确定表示全部状态所需要的状态变量个数，并完成状态分配。给定时序电路的状态分配方案可以有多种，有些方案比其他的好。在前面的例题中我们使用了看起来自然的状态分配方式。在8.2节中我们将重温这个例题，并展示不同的状态分配有可能简化电路的设计。

6) 选择电路中所用的触发器类型。推导出两个逻辑表达式：(a) 表示下一状态的逻辑表达式；(b) 表示电路输出的逻辑表达式。前面的表达式控制所有触发器的数据输入。到目前为止，我们只用过D触发器。我们将在8.7节中考虑使用其他触发器。

7) 实现逻辑表达式所代表的电路。

例8.1 我们已经用一个简单的时序电路说明了设计步骤。从读者的角度来看，电路输入信号的最小变化周期必须大于时钟信号周期的两倍，否则该电路是不可能实现的。现在让我们看一个与实际应用有密切联系的例子。

7.14节介绍了总线的概念，并说明了将寄存器存储的数据从一个寄存器向另一个寄存器传输所必须的连接。图7-60所示的电路说明了如何用三态缓冲器将选中寄存器中的数据放到总线上，也说明了总线上的数据如何加载到寄存器中。图7-62展示了如何使用移位寄存器来实现将寄存器$R1$和$R2$所存储内容互换的控制机制。现在我们将用有限状态机来设计期望的控制机制。

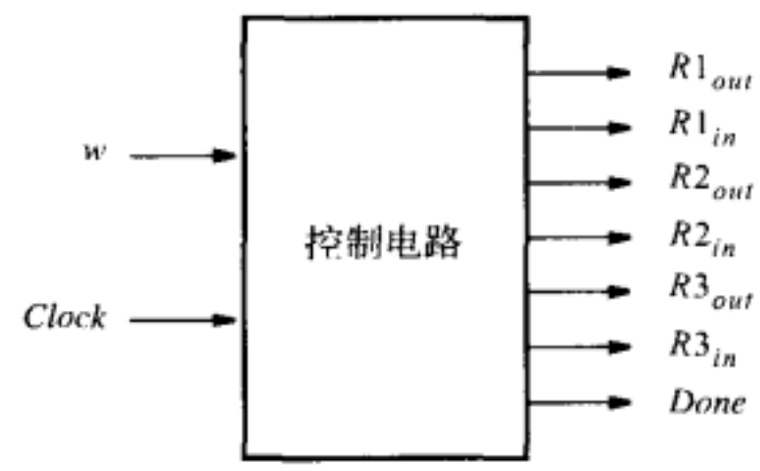


图8-10 例8.1中所需的信号

寄存器$R1$和$R2$中存储的内容可以借助于寄存器$R3$作为临时存储单元进行交换，具体操作步骤如下：先使控制信号$R2_{out} = 1$和$R3_{in} = 1$，把$R2$的内容先加载到$R3$中。然后使$R1_{out} = 1$和$R2_{in} = 1$，把$R1$的内容传入$R2$。最后使$R3_{out} = 1$和$R1_{in} = 1$，把$R3$的内容（即最初$R2$的内容）传入$R1$。由于这一步完成了所需的交换，我们将通过信号$Done = 1$来表明任务完成。假设持续一个时钟周期的输入信号w引发这次交换。图8-10描述了这个想要的控制电路的外部信号。图8-11所示为时序电路的状态图，该时序电路产生所需的输出控制信号序列。请注意，为了使状态图简明，我们仅仅标出输出信号为1时的情况。在所有其他情况下，输出信号都为0。

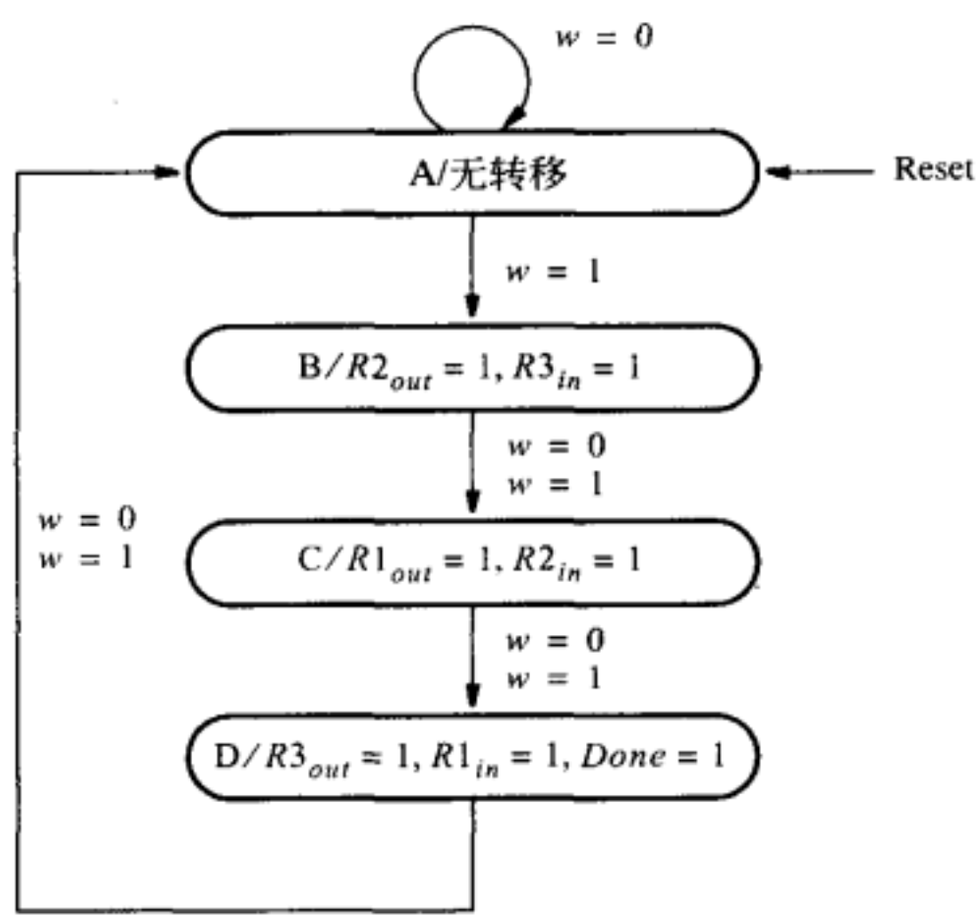


图8-11 例8.1的状态图

在起始状态A，没有标出任何转移，所有的输出信号都为0。电路保持该状态，直到要求进行交换的信号w变为1时，才离开该状态。在状态B，将$R2$的内容传送到$R3$的控制信号变成有效。下一个有效时钟沿便把$R2$的内容放入$R3$。不管w为0还是1，电路进入状态C。在状态C，把$R1$的内容加载到$R2$的控制信号有效。在下一个有效时钟沿完成$R1$到$R2$的加载，同时不管w为何值，状态都转移到D。最后一步，从$R3$到$R1$的加载发生在离开状态D返回状态A的时钟沿时刻。

图8-12用状态表的方式表述了同样的信息。因为有四种不同的状态，所以需要用两个状态变量y_2和y_1。最简单的状态分配方式是把状态取值$y_2y_1 = 00$、01、10、11分别分配给A、B、C、D四种状态，得到如图8-13所示的状态分配表。使用D触发器来赋值，可以得到下一状态的表达式，如图8-14所示。它们是：

$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

输出控制信号的逻辑表达式为：

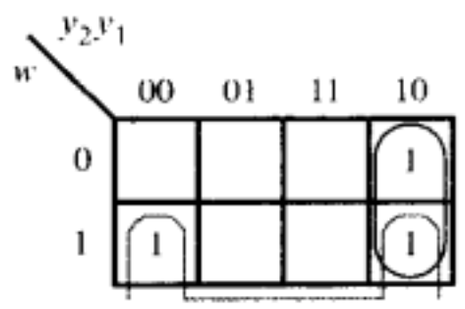
$$R1_{out} = R2_{in} = \bar{y}_1 y_2$$
$$R1_{in} = R3_{out} = Done = y_1 y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

当前状态	下一状态		输出						
	$w=0$	$w=1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

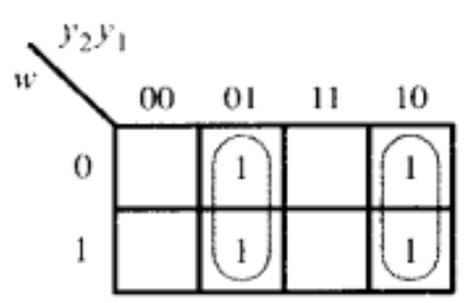
图8-12 例8.1的状态表

	当前状态	下一状态		输出						
		$w=0$	$w=1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	10	10	0	0	1	0	0	1	0
C	10	11	11	1	0	0	1	0	0	0
D	11	00	00	0	1	0	0	1	0	1

图8-13 图8-12时序电路的状态分配表



$Y_1 = w\bar{y}_1 + \bar{y}_1y_2$



$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2$

图8-14 图8-13时序电路的下一状态表达式的推导

由这些表达式可以得到如图8-15所示的电路。这个电路显得比图7-57中的移位寄存器更复杂些，但它仅使用了两个触发器，而不是三个触发器。

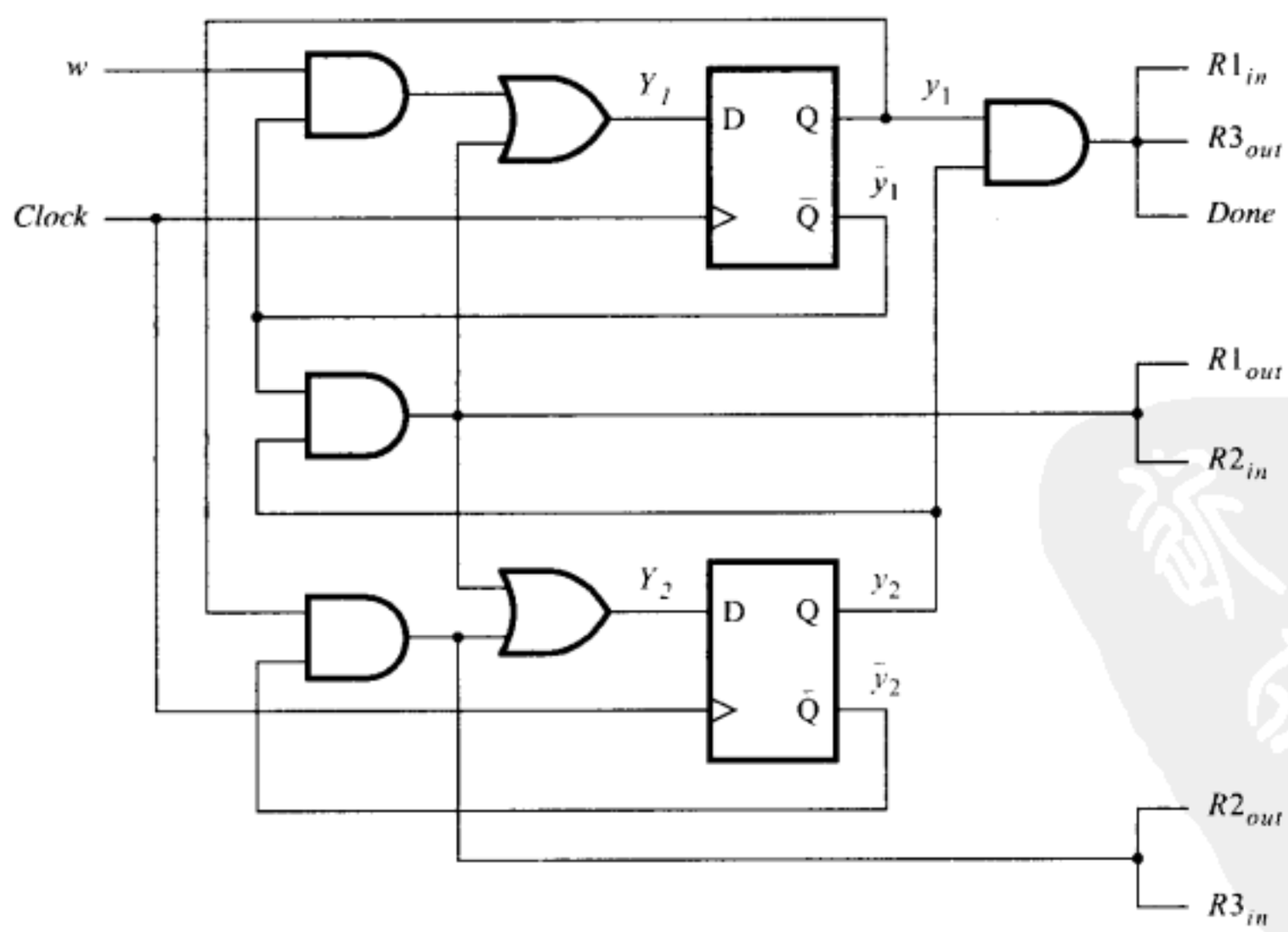


图8-15 图8-13时序电路的最终实现

8.2 状态分配问题

在引入时序电路设计的基本概念之后，我们应该仔细分析一下是否还有其他可能的解决办法。在8.1.6节中，我们曾提到有的状态分配效果较好，有的较差。我们用图8-4举例说明。我们早就知道图8-6中的状态分配可以得到看起来更为简单的电路，见图8-8所示。但是，是否可以找到更好的状态分配方式，使图8-4所示的有限状态机的电路进一步简化呢？

图8-16展示了另一种状态分配方案。在这种方案中，$y_2y_1 = 00$、01 和11分别分配给状态A、B 和C。$y_2y_1 = 10$ 无用，我们将它当作无关项对待。若再次选用D触发器来实现电路，则由该状态分配图可求出表示下一状态和输出信号的表达式：

$$Y_1 = D_1 = w$$
$$Y_2 = D_2 = wy_1$$
$$z = y_2$$

这些表达式定义了图8-17所示的电路。将这个电路与图8-8所示的电路做比较，可以看到新电路的成本更低，因为构建电路所需的门的个数更少些。

	当前状态	下一状态		输出
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	dd	dd	d

图8-16 改进后的状态分配（图8-4中的时序电路）

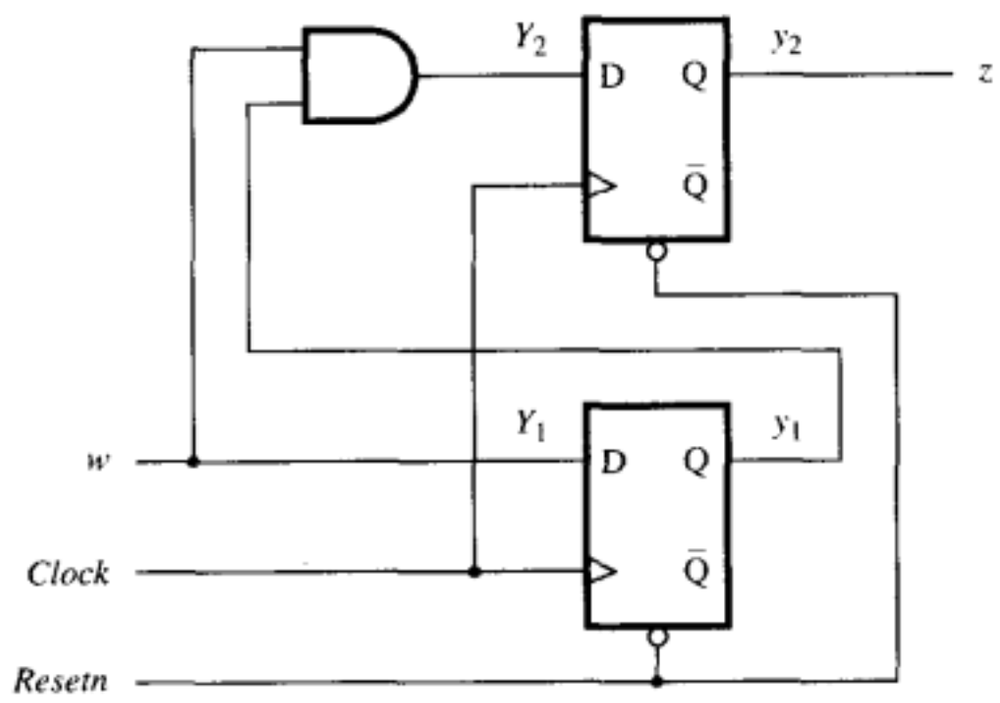


图8-17 改进状态分配后（见图8-16）最终实现的电路

实际电路的规模通常远比我们的例子大得多，不同的状态分配方案对最终实现电路的成本有着实质性影响。虽然人们非常想找到最佳的状态分配方案，但对规模大的电路，找到最佳状态分配方案往往是不可能的。由于方案过多，通过穷举所有可能的方案来寻找最佳的状态分配是不切实际的。现代计算机辅助设计（CAD）工具经常采用启发式方法进行状态分配的优化。这些方法通常是公司的专利，其技术细节很少公开发表。

例8.2 在图8-13中，我们对图8-12所示时序电路进行简单的状态分配。现在我们考虑把分配给状态C和状态D的取值交换一下（见图8-18所示）所产生的影响。根据图8-19，可以写出下一个状态的表达式如下：

$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$
$$Y_2 = y_1$$

输出表达式如下：

$$R1_{out} = R2_{in} = y_1y_2$$
$$R1_{in} = R3_{out} = Done = \bar{y}_1y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

根据这些表达式设计的电路将比图8-15所示的电路简单一些。

	当前状态	下一状态		输出						
		$w=0$	$w=1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	11	11	0	0	1	0	0	1	0
C	11	10	10	1	0	0	1	0	0	0
D	10	00	00	0	1	0	0	1	0	1

图8-18 图8-12所示时序电路改进后的状态分配

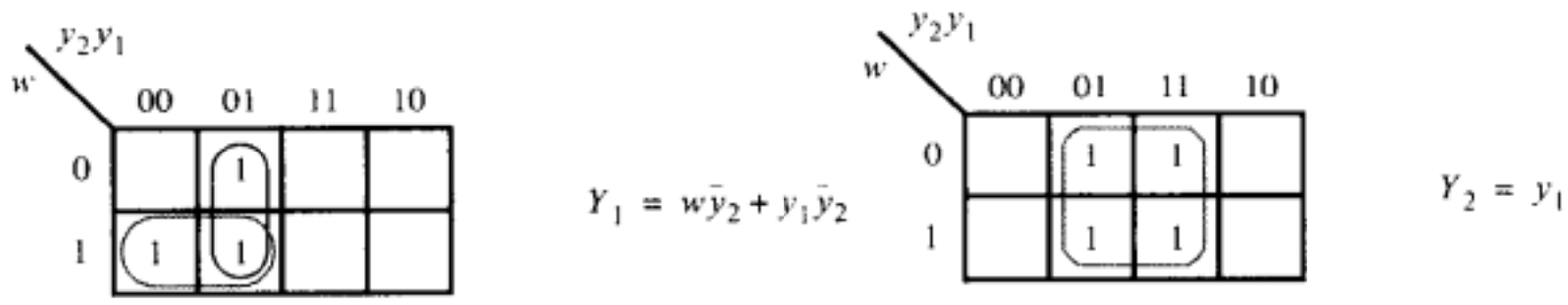


图8-19 图8-18所示时序电路的下一状态表达式的推导

独热编码

另一种有趣的方法是将时序电路的状态用状态变量中某个位为1来定义。用这种方法，状态变量的所有位中只能有一位为1，其余各位都必须为0。为1的那个变量被认为是“热的”。这种状态分配方法称为**独热编码**。

	当前状态	下一状态		输出
		$w=0$	$w=1$	
	$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
A	001	001	010	0
B	010	001	100	0
C	100	001	100	1

图8-20 图8-4时序电路的独热码状态分配

图8-20表示如何将独热码应用到图8-4时序电路的状态分配中。因为该电路有三种状态，所以必须使用三个状态变量。状态变量分配方式确定为：状态$A = y_3y_2y_1 = 001$，状态$B = y_3y_2y_1 = 010$，以及状态$C = y_3y_2y_1 = 100$。该状态变量的其他五种取值没有用。在推导下一个状态和输出表达式时，其余五种取值可被当做无关项处理。使用上述状态分配方案，可得到如下表达式：

$$Y_1 = \overline{w}$$
$$Y_2 = wy_1$$
$$Y_3 = w\overline{y}_1$$
$$z = y_3$$

上述表达式并不比图8-16所示的状态分配更简单些。尽管在本情况下，独热码分配方式并无优势可言，但在很多情况下，用独热码分配状态的方法是有优势的。

例8.3 如图8-21所示，图8-12的时序电路可以应用独热码状态分配来实现。共需要四个状态变量，状态A、B、C、D分别被编码为$y_4y_3y_2y_1 = 0001$、0010、0100、1000。状态变量的其余12种取值可作为无关项处理。表示下一状态的表达式如下：

$$Y_1 = \overline{w}y_1 + y_4$$
$$Y_2 = wy_1$$
$$Y_3 = y_2$$
$$Y_4 = y_3$$

我们只要仔细地观察图8-11的状态表就能得到这些表达式，注意到这一点是很有启发的。若FSM处于状态A，且$w = 0$，或者若FSM处于状态D，则触发器y_1应置为1；因此$Y_1 = wy_1 + y_4$。若FSM的当前状态为A且$w = 1$，则触发器y_2应置为1；因此$Y_2 = wy_1$。若FSM 的当前状态为B或C，则触发器y_3和y_4分别应置为1；因此有$Y_3 = y_2$和$Y_4 = y_3$。

	当前状态	下一状态		输出						
		$w = 0$	$w = 1$							
	$y_4y_3y_2y_1$	$Y_4Y_3Y_2Y_1$	$Y_4Y_3Y_2Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0001	0001	0010	0	0	0	0	0	0	0
B	0010	0100	0100	0	0	1	0	0	1	0
C	0100	1000	1000	1	0	0	1	0	0	0
D	1000	0001	0001	0	1	0	0	1	0	1

图8-21 图8-12时序电路的独热码状态分配

输出表达式就是触发器的输出，因此有

$$R1_{out} = R2_{in} = y_3$$
$$R1_{in} = R3_{out} = Done = y_4$$
$$R2_{out} = R3_{in} = y_2$$

这些表达式比例8.2中求出的简单些，但所需的触发器是四个，而不是两个。 ■

独热编码状态分配的一个重要特点是其输出表达式比由状态变量最少的分配方案所求得的输出表达式简单。较简单的输出表达式可以导致响应速度更快的电路。例如，若时序电路的输出刚好就是触发器的输出，如我们例子中的情况，则触发器的翻转马上就可以得到输出信号。若输出表达式比较复杂，则必须考虑实现这些表达式的电路中门的传播延迟。我们将在8.8.2节中考虑这个问题。

考虑到以上这点的这些例子说明了给定有限状态机的时序电路实现方案可以有许多种。每一种实现方案的电路成本和时序特性都不相同。在下一节中，我们将讨论另一种有限状态机的建模方法，该方法能导致更多可能性。

8.3 米利型状态模型

我们曾介绍过的都是一些时序电路的例子，在这些时序电路中每个状态都有特定的输出与之对应。正如在本章开始时所解释的那样，这种有限状态机被称为摩尔（Moore）型有限状态机。现在让我们来探讨米利（Mealy）型有限状态机，这种状态机的输出值是根据电路所处的状态与当前的输入而产生的，这给时序电路的设计带来了更大的灵活性。我们将把前面的例子稍做改动用以介绍米利型有限状态机。

在8.1节中，第一个时序电路的本质是在连续两次时钟周期中都检测到输入信号$w = 1$时，便产生输出$z = 1$。设计要求当第二个时钟继续检测到$w = 1$的下一个时钟周期内，输出z变为1。现在我们假设把在下一个时钟周期内输出z为1的要求改为在第二个时钟检测到$w = 1$的同时将输出z置为1，则输入输出的时序应该如图8-22所示。为了说明如何理解图8-22所示的行为，我们先选定状态A作为起始态。只要$w = 0$，状态机就会停留在状态A，产生的输出为$z = 0$。当$w = 1$出现时，状态机便转移到新状态B，以记录输入$w = 1$已经到来。当状态机正处于状态B时，若w继续保持为1，假如$w = 1$至少已经维持了两个连续的时钟周期，状态机仍应该保持在状态B，并输出$z = 1$。一旦w变为0，z应马上变为0，并且在下一个时钟的有效沿返回状态A。因此，由图8-22指定的行为可以用一个两状态FSM实现，状态图如图8-23所示。之所以只需

两个状态，是因为我们允许输出值z既取决于当前的输入信号w，又与状态机的当前状态有关。图8-23的状态图表明，如果状态机处于状态A，若$w = 0$则状态机一直处于状态A，并且输出z为0。这用标有$w = 0/z = 0$的弧表示。当w变为1时，输出z继续为0，直到在下一个时钟有效沿进入状态B时刻，输出z才变为1。这用标有$w = 1/z = 0$的弧表示。在状态B只要$w = 1$，则输出就是1，状态继续留在B，这用标有$w = 1/z = 1$的弧表示。然而，在状态B 若$w = 0$，则输出将会变为0，并且在下一个时钟有效沿返回状态A。理解的关键点是在当前的时钟周期期间，输出值z应标注在从当前状态节点发出的弧上。

时钟周期：	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

图8-22 输入和输出信号的时序

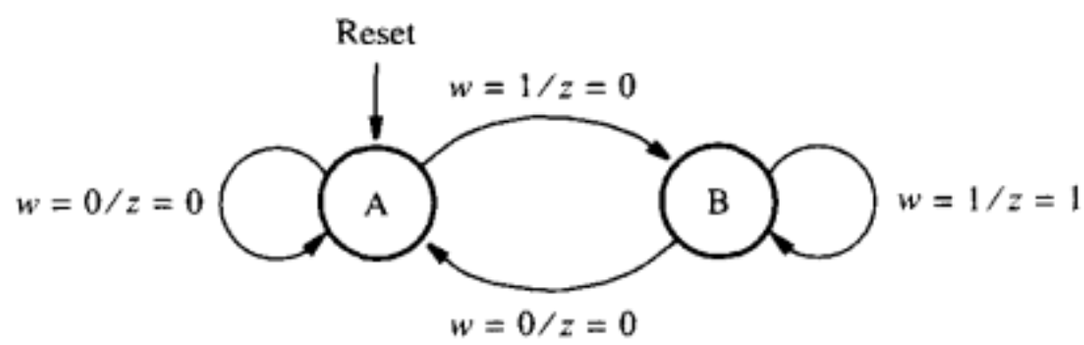


图8-23 一个实现图8-22任务的有限状态机的状态图

我们可以用8.1节所介绍的设计步骤来实现图8-23所示的有限状态机。状态表见图8-24所示。该状态表说明输出z不仅取决于状态机的当前状态，而且还取决于输入w的当前值。图8-25给出了状态分配表。因为仅有两个状态，一个状态变量y就可满足设计要求。假设y 用D触发器实现，所需的下一个状态表达式和输出表达式如下：

$$Y = D = w$$

$$z = wy$$

实现的电路图和时序图如图8-26所示。时序图与图8-22所示的输入输出序列相对应。

当前状态	下一状态		输出 z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

图8-24 图8-23中有限状态机的状态表

	当前状态	下一状态		输出	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

图8-25 图8-24中有限状态机的状态分配表

米利型FSM具有较大的灵活性，因此通常可以使电路得到简化。假定设计需求只是为了检测连续两个时钟期间输入w为1，我们所举的生成了图8-8、图8-17和图8-26电路的那些例子，清楚地说明这一情况。然而，我们应当注意到，图8-26的电路就输出行为而言与图8-8和图8-17的电路并不完全相同。不同点在于图8-26b所示的波形图中输出信号与图8-9的波形图比较有一个时钟周期的移位。若我们想要用米利FSM方案产生与图8-9完全一致的输出波形，则还需要在图8-26a的电路中添加一个触发器，见图8-27所示。该触发器只是将信号z作为输入，延迟一个时钟周期后，以信号Z输出。通过这一修改，我们有效地把米利型电路转换成了输出信号为Z的摩尔型电路。请注意，图8-27所示的电路与图8-17的电路本质上是完全一致的。

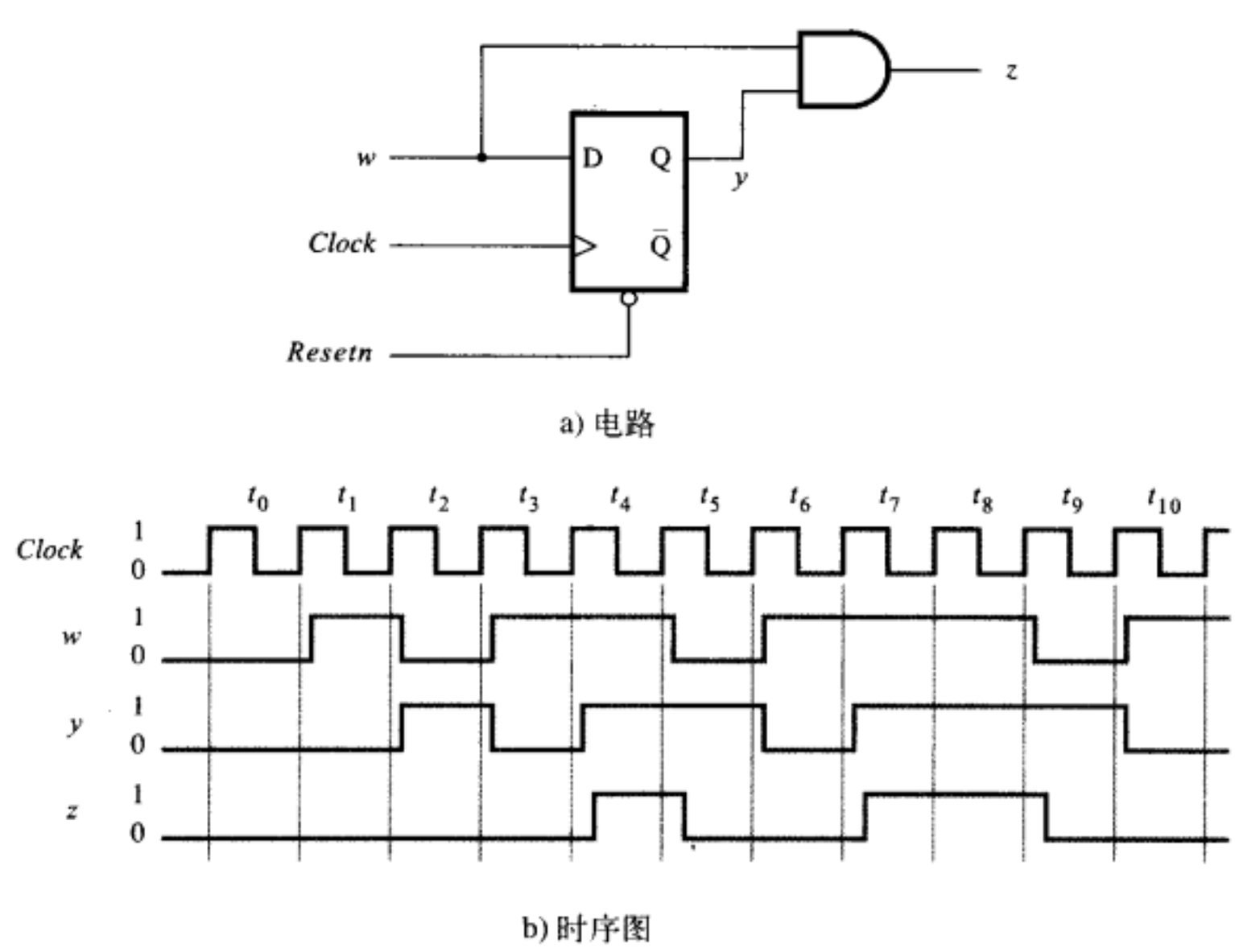


图8-26 图8-25状态分配表的FSM实现

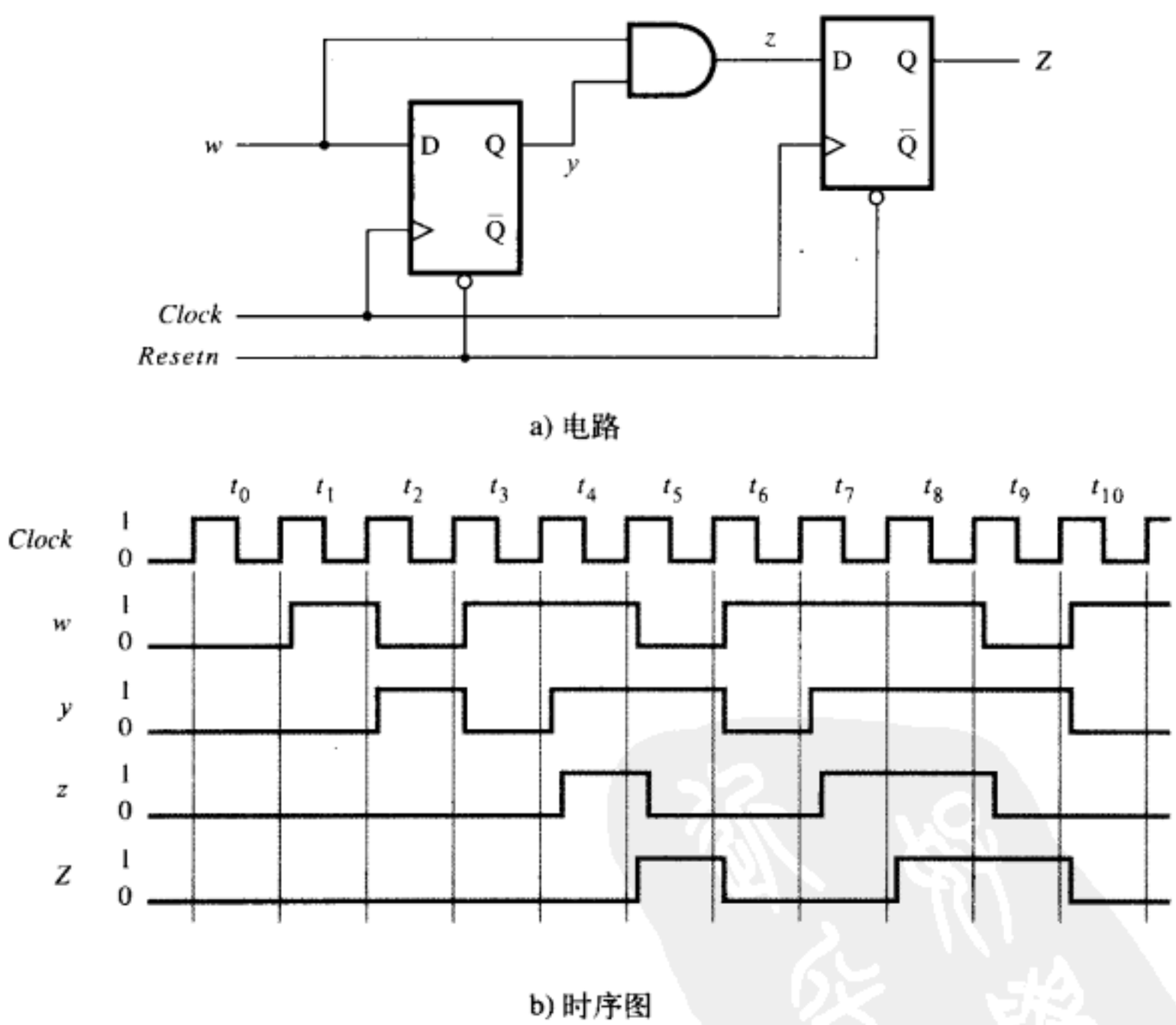


图8-27 实现图8-2指标要求的电路

例8.4 在例8.1中，我们考虑可以将两个寄存器的内容进行交换的控制电路，用摩尔型有限状态机实现了该控制电路。使用米利型FSM也可以完成相同的任务，见图8-28所示。状态A仍然作为复位状态。但是一旦w从0变为1，输出控制信号$R2_{out}$和$R3_{in}$马上变为有效。这两个信

号保持有效，直到下一个时钟周期的开始，当电路离开状态A进入状态B时，才变为无效。在状态B，对于$w = 0$和$w = 1$，输出$R1_{out}$和$R2_{in}$都是被打开的。最终，在状态C令$R3_{out}$和$R1_{in}$有效从而完成交换。

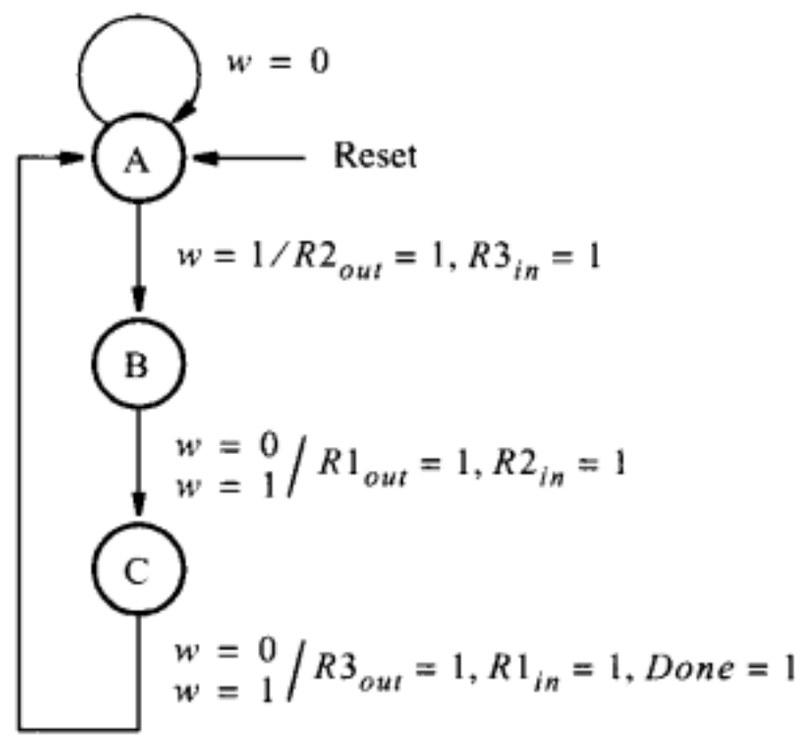


图8-28 例8.4的状态图

该米利型FSM实现控制电路需要三个状态。这并不意味着电路必然有所简化，因为仍然需要用两个触发器来实现状态变量。米利型FSM实现方法与摩尔型FSM实现方法的最重要的不同点是输出信号的时序。由图8-28所示电路实现的有限状态机，其输出控制信号的生成比由例8.1和例8.2推导得到的电路早一个时钟周期。

请注意，使用图8-28所示的有限状态机，$R1$和$R2$中内容交换的全过程需要三个时钟周期，初始和结束状态都是A。使用例8.1的摩尔型有限状态机，交换过程需要四个时钟周期，才能回到起始状态A。

假设我们使用独热编码方案来实现这个有限状态机，则需要三个触发器，A、B、C三个状态分别被分配为$y_3 y_2 y_1$ = 001、010、100。考察图8-28所示的状态转移图，通过观察可以求出下一个状态的表达式。若该有限状态机在状态A并且$w = 0$，或者该有限状态机处在状态C，触发器y_1的输入应当为1；因此$Y_1 = \overline{w} y_1 + y_3$。如果该有限状态机在状态$A$并且$w = 1$，触发器$y_2$应当被置为1；因此$Y_2 = wy_1$。若该FSM的当前状态为$B$，触发器$y_3$应当被置为1；因此$Y_3 = y_2$。输出表达式的推导留给读者作为练习，也可以通过观察得到。相应的电路见7.14节的图7-63所示，该电路是通过一种专门的解决方法推导出来的。■

前面的讨论涉及与时序电路设计有关的基本原理。虽然理解这些基本原理是必须的，但当遇到设计规模大的电路时，这几个例子所展示的用手工推导电路表达式的解决方法会变得非常困难和烦琐。下面将介绍如何用计算机辅助设计（CAD）工具来大大简化设计任务。

8.4 用CAD工具设计有限状态机

我们可以用复杂的CAD工具来设计有限状态机，本节将介绍这些CAD工具。用CAD工具设计有限状态机的初级方法如下：设计者通过前面描述的手工方法根据状态图推导出由触发器和逻辑门组成的电路。通过绘制电路原理图或者编写结构性的硬件描述语言（HDL）代码的方式，将该电路输入到CAD工具中。然后，用CAD系统对电路的行为进行仿真，并用CAD工具在某种芯片（如PLD）上，自动地实现电路。

根据状态图，用手工的方法综合出电路是一件非常繁重的工作。用CAD工具意味着可以使我们免除这种令人乏味的任务。现在已开发出能更好地利用CAD工具来进行有限状态机设计的方法。一种更好的方法是直接把状态图输入到CAD系统中，让计算机自动地完成整个综合过程。CAD工具通过两种主要途径来支持这种方法。一种方法是允许设计者用类似原理图输入工具的图形输入工具绘制状态图，设计者用圆圈表示状态，用弧线表示状态转移并指明状态机产生的输出。另一种更受欢迎的方法是使用硬件描述语言（HDL）编写描述状态转移图的代码，下面讲解这种方法 。

很多硬件描述语言都为设计者提供了描述状态转移图的样板。为了说明这是如何实现的，我们提供了一段Verilog代码，该代码表示的是8.1节第一个例子所示的用手工设计的简单状态机。然后我们用CAD工具将该代码综合成能在芯片上实现的电路。

8.4.1 摩尔型有限状态机的Verilog代码

Verilog语言并没有定义描述有限状态机的标准方式。因此，只要符合Verilog的语法，给定的FSM可以用多种形式的Verilog模块来描述。图8-29所示是一个Verilog代码的例子，该代码描述了图8-3所示的FSM。该代码直接反映了图8-5所示的FSM结构。模块simple有输入端口*Clock*、*Resetn*、*w*和输出*z*。两位向量*y*和*Y*分别代表了状态机的当前状态和下一个状态。用**parameter**语句分配了状态值，使其与图8-6所示的状态值一致。

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)   Y = B;
            else     Y = A;
         B: if (w)   Y = C;
            else     Y = A;
         C: if (w)   Y = C;
            else     Y = A;
         default:    Y = 2'bxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define output
   assign z = (y == C);

endmodule
```

图8-29 图8-3所示的有限状态机的Verilog代码

状态转移用两个独立的**always**块来指定。第一个**always**块描述了所需的组合电路。下一状态向量*Y*的值随着敏感列表信号*w*和*y*的变化而变化。该电路由对每个*y*和*w*值都给出对应*Y*值的**case**语句来定义，**case**语句的每个分支项对应状态机的当前状态。相关的**if-else**语句规定了将要转移的下一个状态。因为状态机只有三个状态，我们在**case**语句中添加了**default**子句，告诉Verilog编译器未分配的第四个状态可以被当做图8-6曾描述过的无关项处理。这一部分代码对应图8-5中左边的那块组合电路。

第二个**always**块将触发器引入了电路。它的敏感列表由复位和时钟信号组成。当输入*Resetn*变为0时，异步的复位信号起作用，有限状态机进入状态*A*。**else**语句保证了在每一个时钟正跳变沿之后，信号*y*应当取得信号*Y*的值，因而实现了状态的变化。

这是一个只有在状态*C*输出*z*才等于1的摩尔型有限状态机。FSM的输出用若$y = C$，则$z = 1$这样简单的条件赋值语句来定义。这条语句实现了图8-5中右边的那块组合电路。

8.4.2 Verilog代码的综合

为了给综合工具生成电路举一个例子，我们将图8-29所示的代码综合成CPLD芯片中实现的电路。综合产生的电路由两个触发器组成，触发器的输入为Y_1和Y_2，输出为y_1和y_2。由综合工具产生的下一个状态的表达式为：

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = wy_1 + wy_2$$

输出表达式为：

$$z = y_2$$

这些表达式与Y_1、Y_2和z三变量卡诺图（见图8-7所示）中的未用状态$y_2y_1 = 11$被作为无关项处理的情况相对应。

图8-30画出了用CPLD实现的有限状态机电路的一部分。为了让图简单些，只有实现y_1、y_2和z的两个宏单元所使用的逻辑资源才被显示出来。电路所用到的宏单元部分在图中用灰色线条标明。

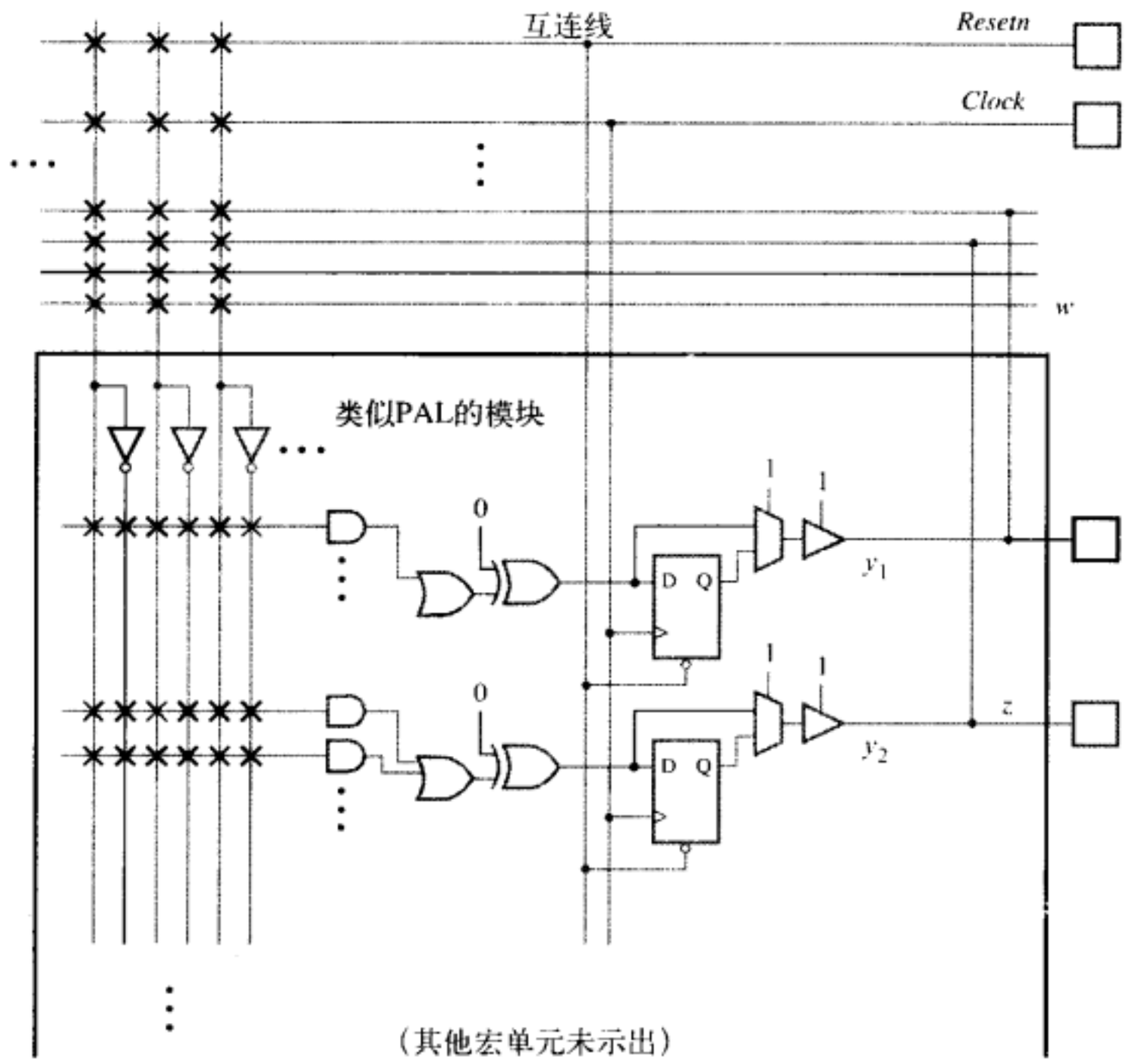


图8-30 图8-3所示有限状态机在CPLD中的实现

在图上可以看到，输入到电路的信号w被连接到CPLD中的一根互连线。产生信号w的源节点没有画出。假设CPLD中包含连接到状态机的一些其他电路，w有可能来自于输入引脚，也有可能来自于另一个宏单元的输出。*Clock*信号连接到芯片上专门用于时钟信号的引脚。该专门的引脚就是芯片的全局时钟线，该连线把时钟信号分配给芯片中所有触发器的时钟信号端。全局时钟线在把时钟信号连接分配到芯片的每个触发器时，可以将时钟的到达时间差或时钟偏差最小化。时钟偏差的概念将在10.3节中讨论。全局线也被用于复位信号。

图8-30中上面那个宏单元产生状态变量y_1。另一个宏单元产生y_2。上面那个宏单元产生信号y_1所需的乘积项，如图所示。该宏单元中其他乘积项的连线在图中没有画出，但每条线都被置0以不影响或门。或门的输出通过另一个输入是0的异或门连接到表示输出信号y_1的触发器的数据端。尽管异或门除了产生一个小的传播延时外，对电路的行为并没有产生什么影响，但该异或门是宏单元的一部分，在电路的实现时是不可以省去的。多路器的选择输入被置为1，使得信号y_1传输到三态缓冲器。跟异或门的情况类似，我们设计的电路并不需要缓冲器，但它是宏单元中的固有组成部分，所以必须使用缓冲器；因此将它的输出使能控制信号置1。信号y_1被连接到CPLD的互连线上，并被反馈回宏单元。我们观察到尽管y_1并不是电路的输出，却用一个信号通路将y_1连接到芯片的引脚。因此，这个引脚不能被用于任何其他目的。除了涉及两个乘积项之外，y_2的实现类似于y_1。y_2信号被连接到标记为z的引脚，实现了所要求的输出信号。

图8-31说明了如何将电路引脚分配到芯片的引脚（该芯片是一个44引脚PLCC封装的小型CPLD）。图中芯片封装的顶部被画成切开一部分，揭示了图8-30中两个宏单元概念性的视图，它们被用灰色标出。这个简单的电路只使用了该器件中很小的一部分资源。

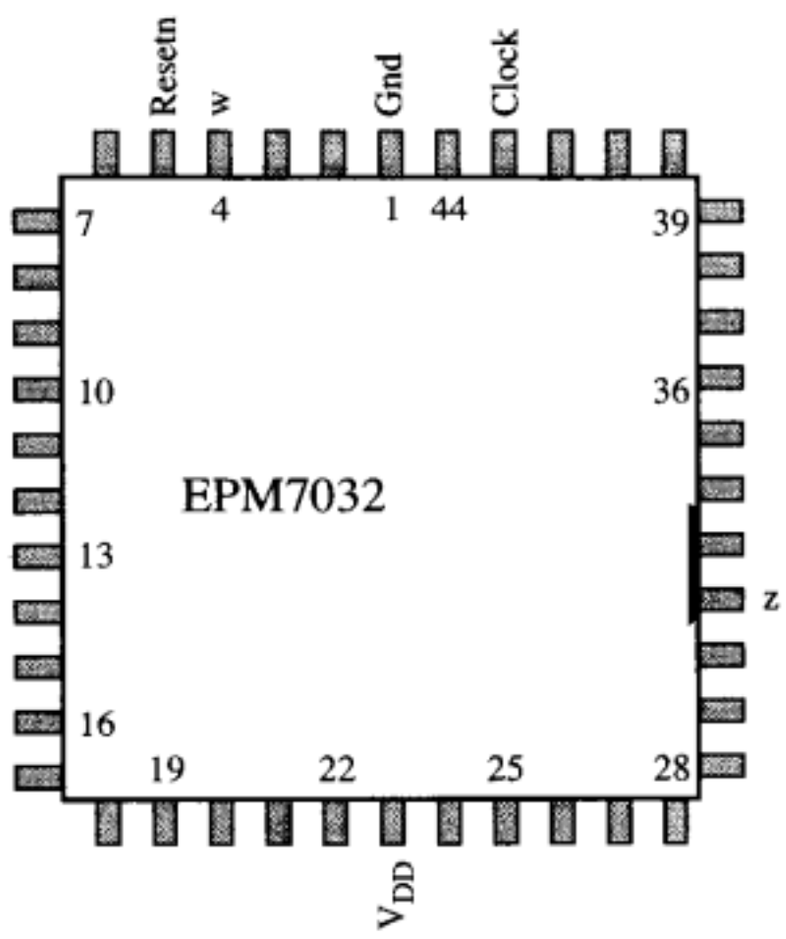


图8-31 在一片小规模CPLD上的图8-30的电路

8.4.3 电路的仿真和测试

用时序仿真可以测试CPLD芯片实现的电路行为，见图8-32所示。该图所示波形与图8-9的时序图相对应，假定所用的时钟周期为100ns。在仿真的开始阶段，信号*Resetn*被置0，然后被置1。在*w*被置为1的连续两个时钟周期后，电路将在以后的一个时钟周期内，使输出信号*z* = 1 。当*w*被连续三个时钟周期置1时，则 *z*变为两个周期内保持为1。为了增加可读性，我们用字母*A*、*B*、*C*来表示状态的变化。(与本书配套的仿真器实际显示的是这三个状态对应的二进制代码。)

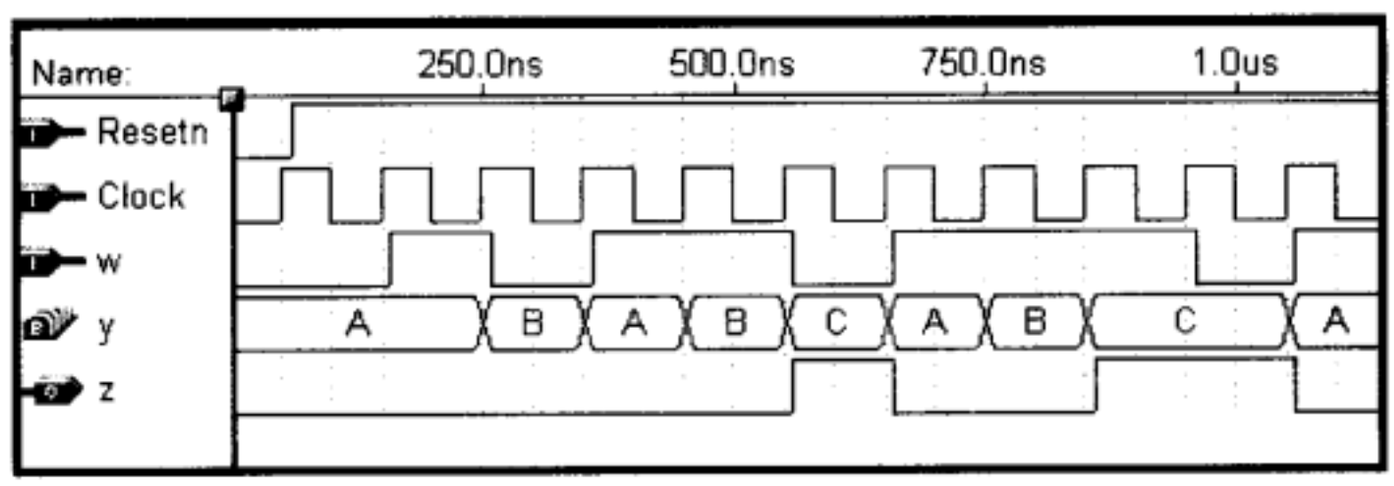


图8-32 图8-30中电路的仿真结果

检查仿真的输出后，我们还应考虑以下的问题：我们是否就此能得出实现电路的功能是正确的并满足所有需求的结论呢？对于我们的简单例题，这个问题不难回答，因为电路的输入仅有一个，其行为也简单易懂。可以很容易地看到电路工作是否正常。但是，就总体而言，通常很难确认在所有输入序列情况下，电路是否都能正常地运行，因为若要确认这一点，必须经过大量的输入序列的测试才有可能。对于大规模的有限状态机，设计者必须仔细考虑，在仿真中用哪些输入数据进行测试才能达到测试目的。

8.4.4 另一种风格的Verilog代码

在本节的前面部分，我们曾提到Verilog语言并未指定编写描述有限状态机的代码的标准风格。图8-29所示的代码只是其中一种风格。图8-33所示的代码为该有限状态机的另一种风

格的描述。在这种风格中，我们在定义所需的组合逻辑的**always**块中指定了输出z。其效果与图8-29所示的代码一致。

图8-34采用了不同的途径，只使用了一个**always**块。状态用两位向量y来表示。用沿敏感列表（包含复位和时钟信号）的**always**块给出所需的状态转换。当*Resetn*变为0时，实现异步复位。其他的状态转移用与图8-3对应的**case**语句来定义。**default**语句指明$y = y_2 y_1 = 11$可以被当做无关项。

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state and output combinational circuits
   always @(w, y)
   begin
      case (y)
         A: if (w)   Y = B;
            else     Y = A;
         B: if (w)   Y = C;
            else     Y = A;
         C: if (w)   Y = C;
            else     Y = A;
         default:    Y = 2'bxx;
      endcase
      z = (y == C);   //Define output
   end

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

endmodule
```

图8-33 图8-3所示状态机的另一个版本代码

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else
         case (y)
            A: if (w)   y <= B;
               else     y <= A;
            B: if (w)   y <= C;
               else     y <= A;
            C: if (w)   y <= C;
               else     y <= A;
            default:    y <= 2'bxx;
         endcase

   // Define output
   assign z = (y == C);

endmodule
```

图8-34 图8-3所示状态机的第三个版本的代码

定义z的赋值语句被放在**always**块之外。这引起如下正确的操作：**always**块在时钟正跳变升沿执行所有变化，编译器由此推断出表示状态变量y_1和y_2的触发器。当状态机进入状态C，即$y_2 y_1 = 10$时，赋值语句**assign**马上使z变为1。考虑无关项，这条语句等价于$z = y_2$。注意这条赋值语句不能如图8-33所示那样放在**always**块内部。这样做会使编译器推断用一个触发器来表示z，从而使z信号的变化相对于y_2延迟了一个时钟周期。

我们已列出了这个有限状态机示例的三种不同风格的Verilog代码。Verilog编译器对每个版本的代码进行处理后所产生的电路可能略有不同，读者也已能完全理解这一点，因为一个给定的逻辑函数可以有很多种实现方法。然而，由这三个版本的代码生成的电路所提供的功能是一致的。

例8.5 图8-35列出的代码展示了：如何借助于图8-29所示风格，用Verilog语言来编写描述图8-11中FSM的代码。该有限状态机有四个状态，分别使用状态变量所有的四种可能取值，因此在**case**语句中就没有必要编写**default**子句。

```
module control (Clock, Resetn, w, R1in, R1out, R2in, R2out, R3in, R3out, Done);
   input Clock, Resetn, w;
   output R1in, R1out, R2in, R2out, R3in, R3out, Done;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)   Y = B;
            else     Y = A;
         B:          Y = C;
         C:          Y = D;
         D:          Y = A;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define outputs
   assign R2out = (y == B);
   assign R3in = (y == B);
   assign R1out = (y == C);
   assign R2in = (y == C);
   assign R3out = (y == D);
   assign R1in = (y == D);
   assign Done = (y == D);

endmodule
```

图8-35 图8-11所示有限状态机的Verilog代码

8.4.5 用CAD工具的设计步骤小结

在8.1.6节中，我们曾对用人工推导时序电路的设计步骤进行过总结。现在我们已看到CAD工具能自动完成很多工作。但是，认识到CAD工具并不能完全代替人工设计步骤是很重要的。参照8.1.6节给出的步骤，前两步，也就是得到有限状态机和状态图，仍然需要人工完成。若用状态图信息作为输入，CAD工具便可以自动地完成用逻辑门和触发器生成电路的任务。另外，除8.1.6节给出的设计步骤外，我们还应当添加测试和仿真阶段。关于该问题的详细讨论将放在第11章。

8.4.6 在Verilog代码中进行状态分配

在8.2节中，我们曾看到状态分配的优劣必然会影响设计电路的复杂性。状态分配过程的目标显然是降低电路的成本。成本的优化只是为了减少门和触发器的个数。但也有可能基于其他考虑，例如，当实现的芯片是CPLD时，CAD软件将试图用优化的状态编码求出功能相同而乘积项总数最少的最终电路。

在从图8-29到图8-35所示的Verilog代码示例中，设计者都必须借助于参数声明（**parameter**）语句来实现状态的分配。然而，Verilog编译器通常具有自动寻找最佳的状态分配方案的能力。大多数编译器在进行编译时，当在代码中遇到指定风格（诸如本章所用的典型风格）的代码段时，便能识别出有限状态机。当编译器检测到有限状态机后，它便通过各种手段，例如寻找更好的状态分配、尝试使用独热编码、利用器件结构的特点等手段来优化

实现过程。用户可以选择允许编译器使用其自动优化状态机的功能，也可以选择禁止这项功能，以便编译器按照通常的方法来处理Verilog语句。

8.4.7 用Verilog语句来编写米利型有限状态机

编写米利型有限状态机（FSM）与编写摩尔型FSM的方法很类似。图8-36给出了图8-23所示FSM的Verilog代码。描述状态转移的方法与我们在图8-29所示的第一个Verilog的例子相同。变量*y*和*Y*分别代表当前状态和下一个状态，可以取的值为*A*和*B*。与图8-29的代码相比，主要的不同点在于输出代码的编写方式不同。图8-36中，在**case**语句内定义了输出*z*，也定义了状态的转移。当FSM处在状态*A*时，输出*z*应当为0，但是当它处在状态*B*时，输出*z*应该与*w*的值一致。因为**always**块的敏感列表中包括*w*，若FSM此时正处在状态*B*，*w*的变化应立刻反映到FSM的输出*z*，这正好符合米利型FSM的需求。

```
module mealy (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg y, Y;
   parameter A = 1'b0, B = 1'b1;

   // Define the next state and output combinational circuits
   always @(w, y)
      case (y)
         A:  if (w)
             begin
                z = 0;
                Y = B;
             end
             else
             begin
                z = 0;
                Y = A;
             end
         B:  if (w)
             begin
                z = 1;
                Y = B;
             end
             else
             begin
                z = 0;
                Y = A;
             end
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

endmodule
```

图8-36 图8-23中米利型状态机的Verilog代码

在CPLD芯片上实现的由图8-36代码指定的有限状态机是根据我们在8.3节中用人工方法求出的表达式得到的。图8-37所示为综合后具体电路的仿真时序。输入波形*w*和我们在图8-32中摩尔型FSM曾用过的一个信号一样。我们设计的米利型FSM行为正确，连续第二个时钟检测到*w*为1，*z*马上变为1。

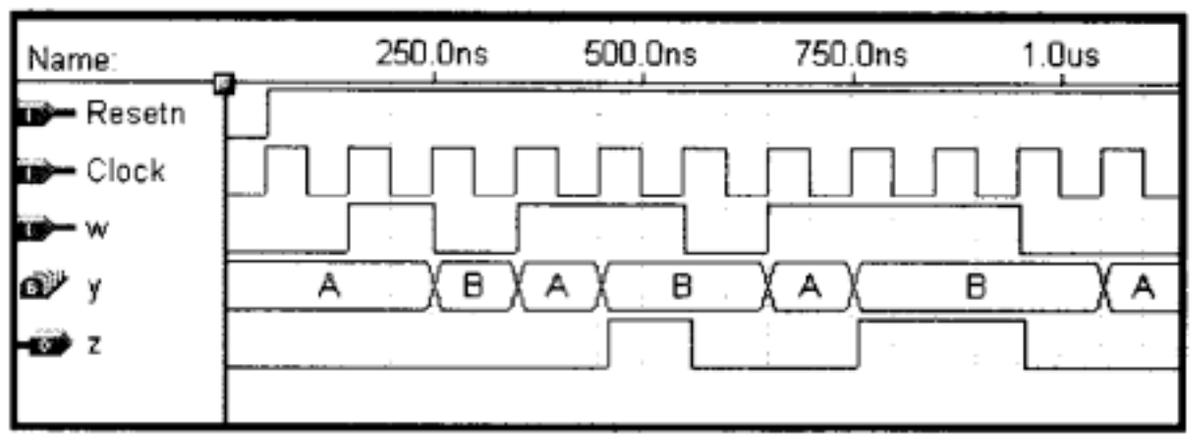


图8-37 米利型状态机的仿真结果

在本节给出的仿真结果中，输入信号w的所有变化都发生在时钟上升沿后的瞬间。这是以8.1.5节的假设为基础的：在实际电路中，w信号必须与控制有限状态机的时钟信号同步。图8-38说明了若w不满足上述条件将会引起的问题。在图8-38中，我们假设w在时钟的负跳变沿发生变化，而不是在时钟的正跳变沿发生变化，而状态机的状态改变是发生在时钟的正跳变沿。若输入w的第一个脉冲持续了100ns，这不应该使得输出z变为1。但电路的行为却不是这样。在信号w变为1后，时钟的第一个正跳变沿使状态机从状态A变到状态B。电路一进入状态B，输入信号w还会继续延时50ns，这导致z变为1。在w返回0后，z也返回0。因此输出z就产生了一个错误的50ns脉冲。

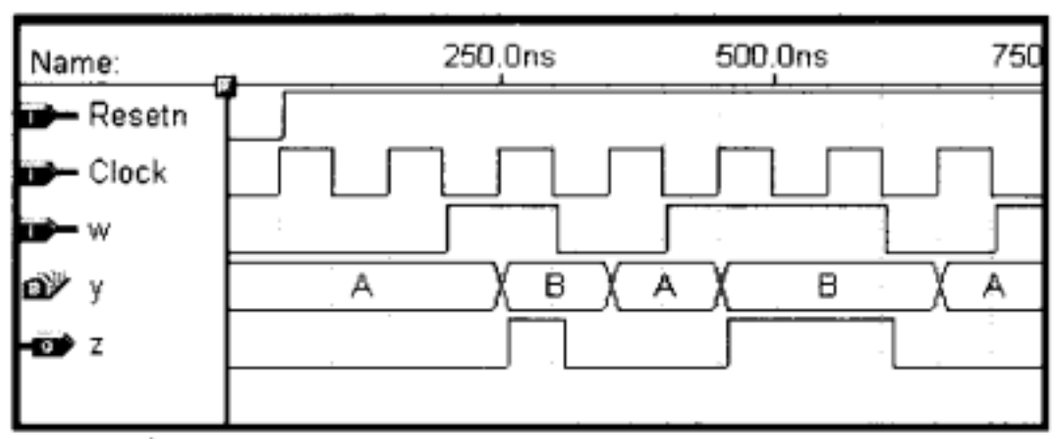


图8-38 米利型状态机的异步输入的潜在问题

我们应进一步深入研究这个问题将会引起的后果。若z被用来驱动由不同时钟控制的另一个电路，则这个不应该出现的脉冲可能会引起很大的问题。但是如果z被用作由同一时钟信号控制的某一电路的输入（很可能是另一个有限状态机），若在下一个时钟正跳变沿（考虑电路的建立时间）前，$z = 0$，则该50ns的脉冲将会被该电路忽略不计（即对后续电路不会发生任何作用）。

8.5 串行加法器举例

现在用另一个简单的例子来说明整个设计过程。在第5章中我们曾详细地讨论过二进制数的加法。我们解释了几种两个n位数并行相加的方案：从行波进位加法器到超前进位加法器。在这些方案中，加法单元的速度是一个重要的设计参数。高速加法器的结构比较复杂，因此必须耗费更多的资源。若速度并不是最重要的考虑因素，则选用每次进行一位数相加的串行加法器是降低电路成本的好方法。

8.5.1 用米利型有限状态机实现的串行加法器

设$A = a_{n-1}a_{n-2}\cdots a_0$和$B = b_{n-1}b_{n-2}\cdots b_0$为两个无符号数，相加产生$Sum = s_{n-1}s_{n-2}\cdots s_0$。我们的任务是设计电路实现一个周期内的对应位的串行相加。这个过程从a_0和b_0相加开始。在下一个时钟周期，a_1和b_1相加，同时加上可能来自低位的进位，依此类推。图8-39为一种实现方式的示意图。它包括三个移位寄存器，分别用来在计算过程中保存A、B、Sum值。假设

输入移位寄存器有如图7-19所示的并行加载能力，则加法任务从把A和B的值加载到寄存器开始。在每一个时钟周期，加法器状态机执行一个对应位的加法运算，在周期的末尾，相加结果被放入寄存器Sum。我们使用正跳变沿触发器，因此所有变化在时钟正跳变沿之后很快发生，延时因不同触发器而略有所不同。同时，三个移位寄存器的内容全部右移一位，使当前的和位移入Sum，并把下一对应输入位a_i和b_i放入加法器的有限状态机。

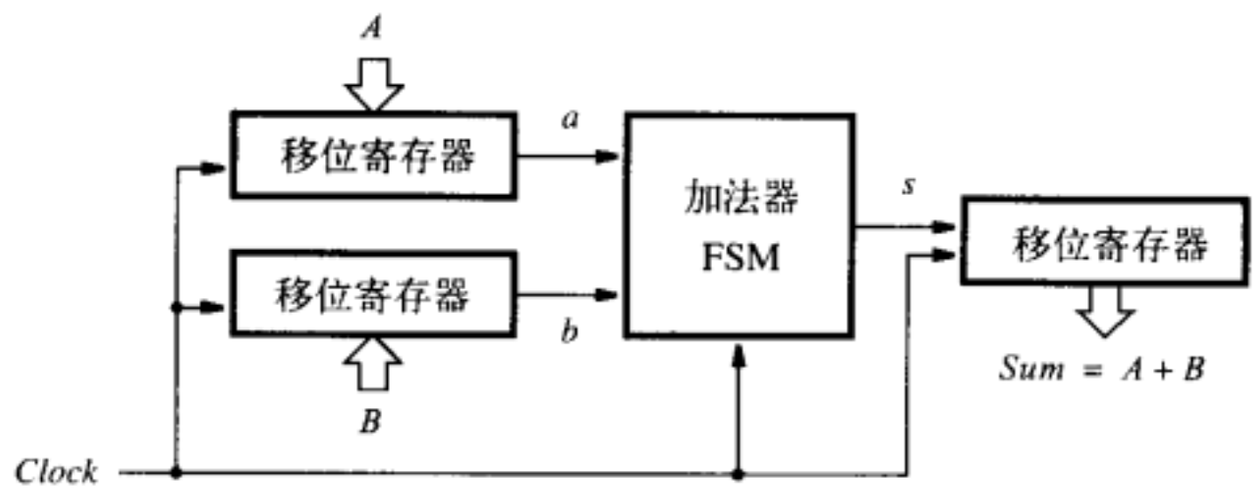


图8-39 串行加法器的框图

现在我们可以准备设计所要求的状态机电路。该电路不可能是组合电路，因为上一位的进位值不同，电路必须产生不同的行为。因此，状态机需要根据进位值为0和1，产生两种不同的状态，分别用状态G和H来表示。图8-40画出了适用于该电路的米利模型状态图。状态和当前输入a和b的取值决定了输出值s。每一个状态转移用符号ab/s标注，表示与给定输入取值a，b对应的输出值为s。在状态G，输入取值00将会产生一个输出$s = 0$，有限状态机的状态保持不变。对于输入值01和10，将产生输出$s = 1$，有限状态机会保持在状态G。但对于输入值11，产生输出 $s = 0$，状态转移到H。在状态H，输入取值01和10会产生输出$s = 0$，而输入值11会产生$s = 1$。在以上三种情况中状态机都保持在状态H。然而，当输入取值为00时，产生输出s为1，状态返回G。

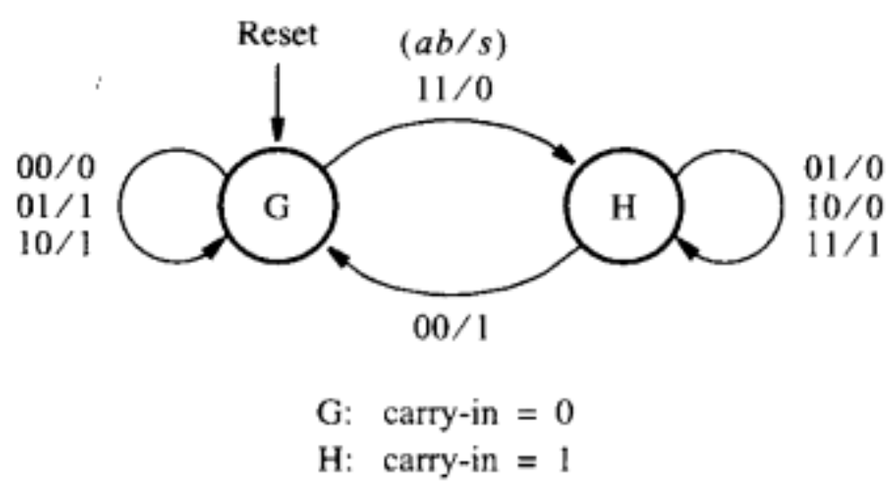


图8-40 串行加法器FSM的状态图

对应的状态表如图8-41所示。表示两个状态需要使用一个触发器。状态分配可按照图8-42所示的状态表进行。该状态分配所对应的下一状态和输出表达式为：

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

将这些表达式与5.2节中全加器的表达式相比较，显然，y是进位输入，Y是进位输出，s是全加器的和。因此，图8-39所示的全加器有限状态机方块是由图8-43所示的电路组成的。在开始进行加法操作时，可以用$Reset$信号对触发器进行复位。

当前状态	下一状态				输出 s			
	$ab=00$	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

图8-41 串行加法器的状态表

当前状态	下一状态				输出			
	$ab=00$	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

图8-42 串行加法器的状态分配表

串行加法器是一个简单的电路，可以用于任意多位的两个数相加。图8-39的结构仅受限于移位寄存器的大小（位数）。

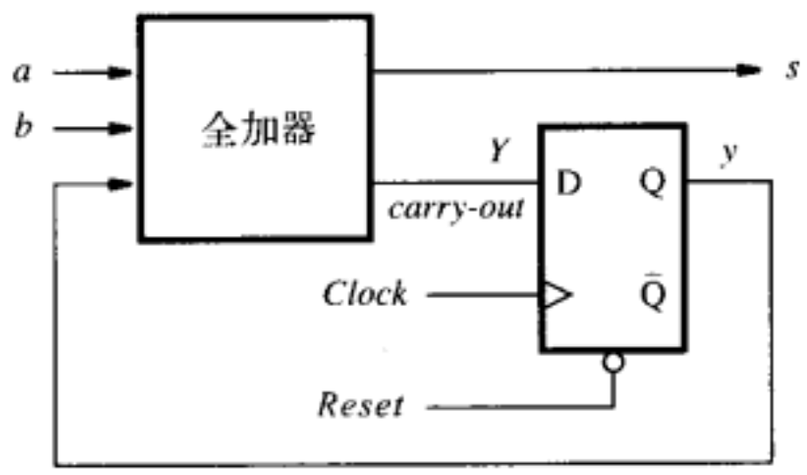


图8-43 图8-39中加法器有限状态机的实现电路

8.5.2 用摩尔型有限状态机实现的串行加法器

在前面的例子中，我们看到米利型有限状态机可以很好地满足实现串行加法器电路的需求。现在尝试用摩尔型有限状态机来完成相同的任务。我们从图8-40的状态图开始讨论。在摩尔型有限状态机中，输出只取决于状态机的状态。因为在G和H两个状态中，输入a和b的四种取值对应两种不同的输出，所以摩尔型有限状态机需要两个以上的状态。我们可以把G和H这两个状态中的每个状态再划分成两个不同的状态，然后推导出合适的状态图。我们不用状态G，而是用状态G_0和状态G_1来分别表示进位输入为0，但是加法器求得的和分别为0和1的两种情况。同样，我们可以用H_0和H_1来代替H。这样图8-40的信息能够直接映射到图8-44的摩尔型状态图中。

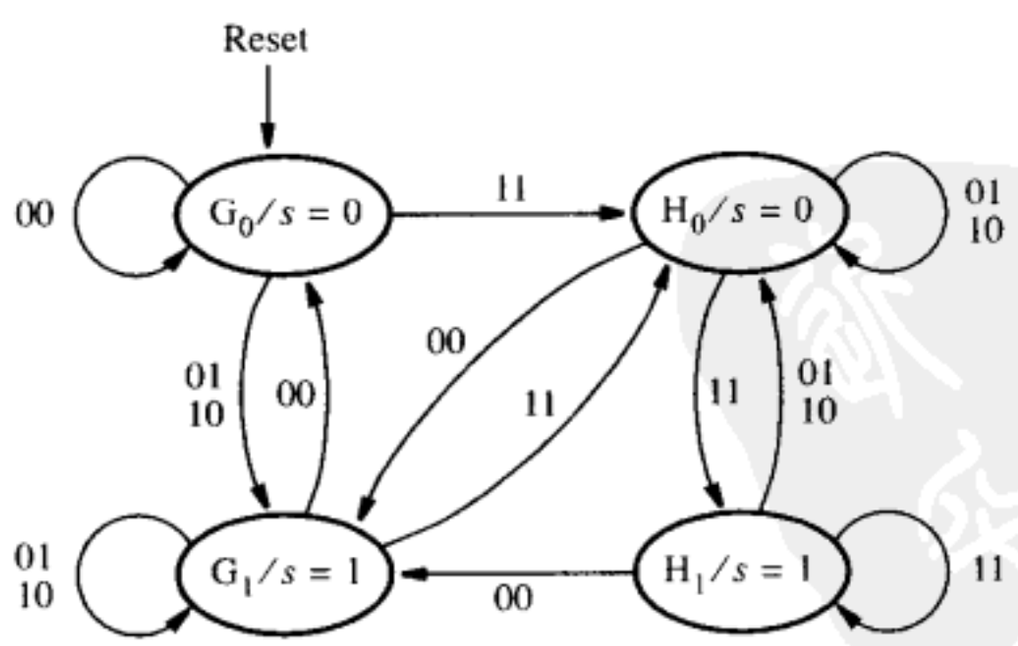


图8-44 摩尔型串行加法器的状态图

对应的状态表如图8-45所示，状态分配表如图8-46所示。表示下一个状态和输出的表达式分别是：

$$Y_1 = a \oplus b \oplus y_2$$
$$Y_2 = ab + ay_2 + by_2$$
$$s = y_1$$

Y_1 和 Y_2 的表达式分别是全加器电路的和表达式和进位表达式。状态机如图8-47所示。很有趣的是图8-43与图8-47很相像。二者唯一的不同之处在于：与米利型时序电路比较，在摩尔型电路中，其输出信号s多经过了一个触发器，因此延迟了一个周期。回想一下，我们曾在前面的例子中观察到这一不同，见图8-26和图8-27所示。

当前状态	下一状态				输出
	$ab=00$	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

图8-45 摩尔型串行加法器的状态表

当前状态 y_2y_1	下一状态				输出
	$ab=00$	01	10	11	s
	Y_2Y_1				
00	00	01	01	10	0
01	00	01	01	10	1
10	01	10	10	11	0
11	01	10	10	11	1

图8-46 图8-45中的状态机的状态赋值表

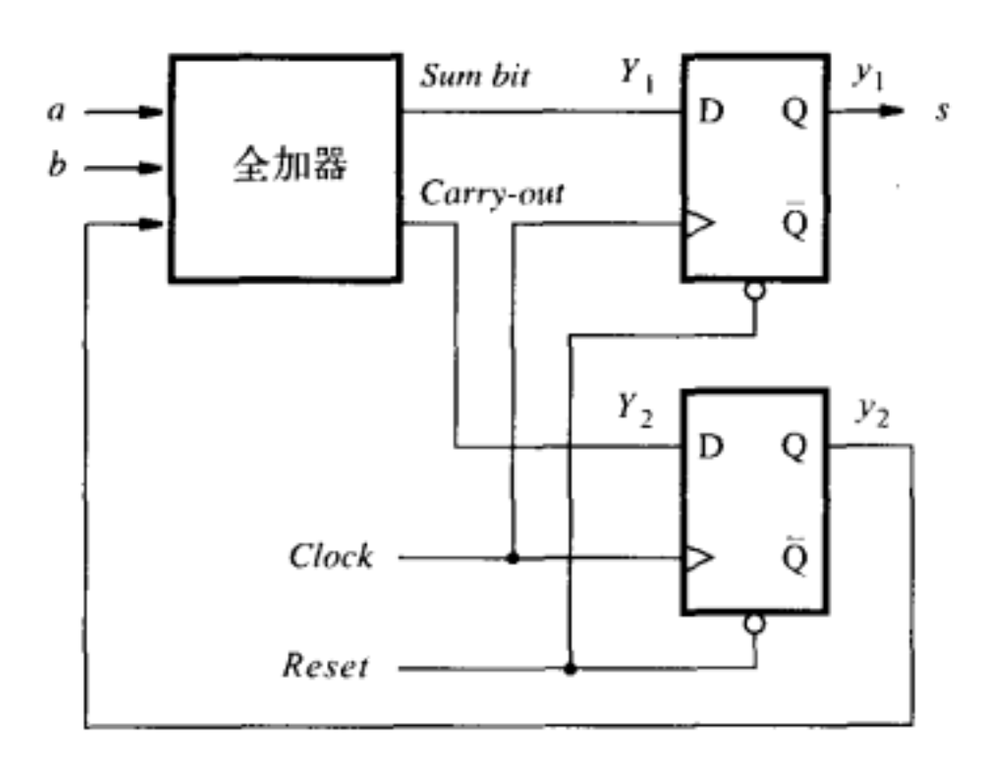


图8-47 摩尔型串行加法器的实现电路

米利型与摩尔型FSM的关键不同在于：米利机中，输入的变化马上影响输出，而摩尔机中，需要等待有效时钟周期沿到来由输入的变化使状态机进入新的状态后，输出才发生变化。我们建议读者画出图8-43和图8-47电路的时序图，那样才能够更好地理解这两种状态机最重要的不同点。

8.5.3 串行加法器的Verilog代码

串行加法器的Verilog代码描述由移位寄存器和加法器有限状态机两部分组成。我们首先设计移位寄存器，然后把它用作串行加法器的一个子电路。

移位寄存器子电路

图7-55给出了n位移位寄存器的Verilog代码。在串行加法器中，在时钟正跳变沿时刻，具有防止移位寄存器中内容发生变化的能力是非常有用的。图8-48给出了名为

```
module shiftrne (R, L, E, w, Clock, Q);
  parameter n = 8;
  input [n-1:0] R;
  input L, E, w, Clock;
  output reg [n-1:0] Q;
  integer k;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else if (E)
    begin
      for (k = n-1; k > 0; k = k-1)
        Q[k-1] <= Q[k];
      Q[n-1] <= w;
    end

endmodule
```

图8-48 带使能输入的右向移位寄存器的代码

*shiftrne*的移位寄存器的代码，该移位寄存器具有输入使能控制信号E。当$E = 1$时，移位寄存器的行为与图7-55中相同。若把E置为0，则可以防止移位寄存器中的数据发生变化。输入E通常被称为使能信号，它对多种类型的电路有用，我们将在第10章中讨论这些问题。

完整的代码

串行加法器的代码如图8-49所示。它用三个寄存器具体地代表输入A和B以及输出Sum。电路复位时，把并行数据加载到移位寄存器中。使用图8-36所示的代码风格，加法器状态机的状态转移图用两个**always**块表示。除了图8-39所示的串行加法器的部件外，Verilog代码中还包含一个递减计数器，当要求的所有n位求和结果全都出现在输出移位寄存器中时，它应使加法器停止操作。当电路复位时，把串行加法器中的位数值n加载到计数器中。当计数器减到0，它将停止操作并禁止输出移位寄存器的值再发生变化。

```
module serial_adder(A, B, Reset, Clock, Sum);
   input [7:0] A, B;
   input Reset, Clock;
   output wire [7:0] Sum;
   reg [3:0] Count;
   reg s, y, Y;
   wire [7:0] QA, QB;
   wire Run;
   parameter G = 1'b0, H = 1'b1;

   shiftrne shift_A (A, Reset, 1'b1, 1'b0, Clock, QA);
   shiftrne shift_B (B, Reset, 1'b1, 1'b0, Clock, QB);
   shiftrne shift_Sum (8'b0, Reset, Run, s, Clock, Sum);

   // Adder FSM
   // Output and next state combinational circuit
   always @(QA, QB, y)
      case (y)
         G: begin
               s = QA[0] ^ QB[0];
               if (QA[0] & QB[0])  Y = H;
               else  Y = G;
            end
         H: begin
               s = QA[0] ~^ QB[0];
               if (~QA[0] & ~QB[0])  Y = G;
               else  Y = H;
            end
         default:  Y = G;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Reset)  y <= G;
      else  y <= Y;

   // Control the shifting process
   always @(posedge Clock)
      if (Reset)  Count = 8;
      else if (Run)  Count = Count - 1;
   assign Run = |Count;

endmodule
```

图8-49 串行加法器的Verilog代码

图8-49列出了可实现一个8位的串行加法器的Verilog代码。内部连线QA和QB分别表示图8-39所示移位寄存器的并行输出，该移位寄存器的输入分别为A和B。变量s代表加法器有限状态机的输出。

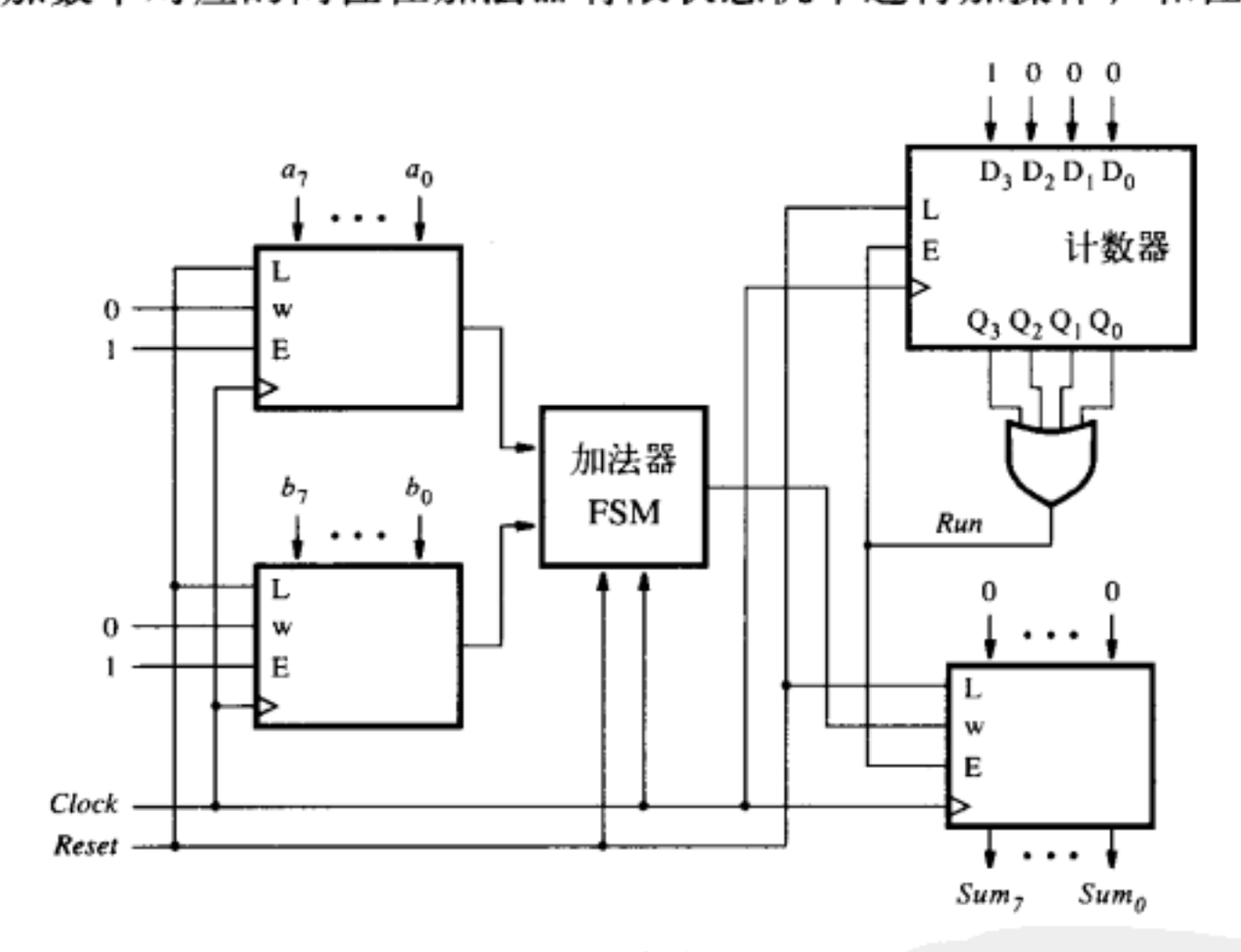
在图8-39中，输入为*A*和*B*的移位寄存器并没有串行输入或使能输入端口。然而，在图8-49所示代码中三次实例引用的*shiftrne*部件（移位寄存器），则具有串行输入和使能输入端口，所以必须有输入信号与它们连接。两个移位寄存器的使能输入端都连接到逻辑值1。移位寄存器的串行输入端的值无所谓，所以可以设置为0或1；我们选择连接到0。移位寄存器由信号*Reset*并行加载。我们选择使用高电平有效的复位信号。输出移位寄存器不需要并行数据输入，所以并行输入端的所有信号线全部都连接0。

第一个**always**块描述了状态的转移和图8-40所示加法器有限状态机的输出。观察图8-40所示的状态转移图，就能得到输出的定义：当FSM处在状态*G*时，加法器的和（sum） $s = a \oplus b$。当FSM处在状态*H*时，加法器的和 $s = \overline{a \oplus b}$ 。第二个**always**块实现触发器*y*，并在*Reset* = 1时，提供同步复位。

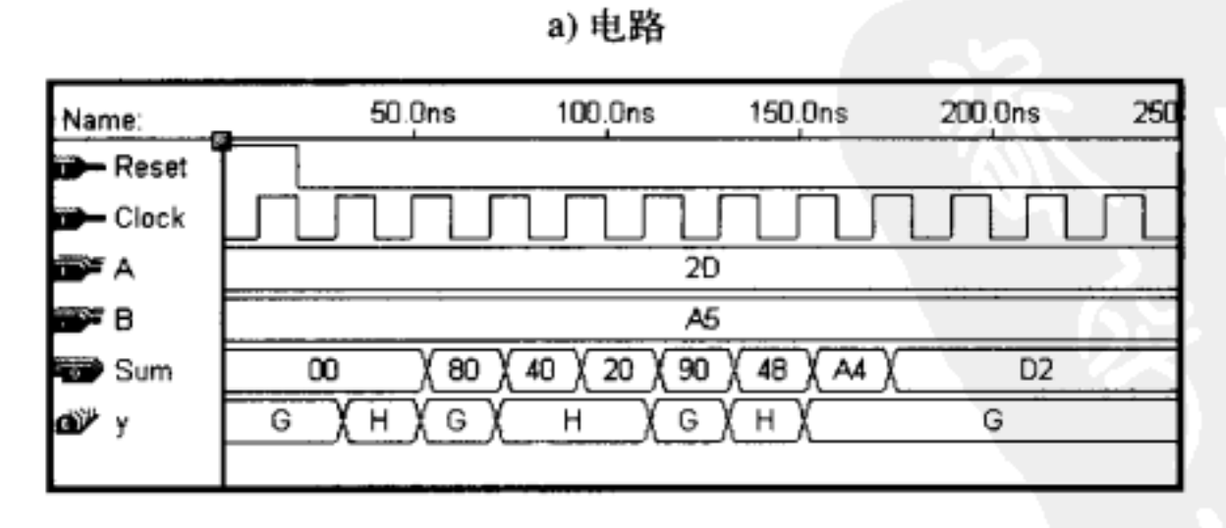
输出移位寄存器的使能输入被命名为*Run*。该使能信号来自于在第三个**always**块中指定的递减计数器的输出。当*Reset* = 1时，计数器被初始化为8。然后只要*Run* = 1，计数信号*Count*每个时钟周期递减1。当*Count* = 0时，通过缩减的或操作(reduction OR operator)，*Run*被置为0。

Verilog代码的综合和仿真

由图8-49所列出的代码综合生成的电路结构见图8-50a所示。计数器的所有输出相或，产生控制信号*Run*，输出到移位寄存器和计数器的使能端。串行加法器电路的时序仿真波形见图8-50b所示。首先将电路复位，把*A*和*B*的值存入输入移位寄存器，递减计数器置8。在每一个时钟周期，两个加数中对应的两位在加法器有限状态机中进行加操作，和位被移到输出移位



a) 电路



b) 仿真结果

图8-50 综合后的串行加法器

寄存器中。8个时钟周期之后，输出移位寄存器中已存入了正确的计算结果（sum），接着控制信号*Run*变为0，移位暂停。

8.6 状态最小化

前面所举的有限状态机的例子都很简单，我们可以很容易地看出所用的状态数就是实现所需功能的最少状态数。当设计者必须面对复杂的有限状态机设计任务时，刚开始设计时，有限状态机所用的状态数往往比实际需要的要多些。状态数最小化有很大的实际意义，因为状态的减少意味着表示状态的触发器可以减少，而且有限状态机所需要的组合电路的复杂性也可以降低。

若有限状态机的状态数能够减少，则原始设计中必定有一些状态与其他一些状态对状态机整体表现所做的贡献是等价的。我们可以用下面的定义对此作出更正规的描述。

定义8.1 *无论S_i和S_j是否为初始态，当且仅当每种可能的输入序列都产生相同的输出序列时，这两个状态S_i和S_j才被称为等价。*

定义能搜寻任意等价状态的最小化过程是可能的。手工实现这个搜寻过程非常麻烦，但可以使用CAD工具来自动完成。考虑到搜寻等价状态过程的冗繁，我们暂不在这里做深入的研究。然而，为了表现状态最小化的效果，我们将介绍另一种更有效但搜寻范围不大的方法。

在给定的状态机中寻找明显地不等价的状态，往往比搜寻等价状态更容易一些。用这种思路可以来定义一种简单的状态最小化过程。

8.6.1 划分最小化的步骤

假设状态机只有一个单独的输入w。若在状态S_i，输入信号$w = 0$，状态转移到S_u，则我们说S_u是S_i的0继承（0-successor）。同理，若在状态S_i，输入信号$w = 1$，状态转移到S_v，则我们说S_v是S_i的1继承（1-successor）。总而言之，我们把所有S_i的继承看作是它的k继承。当状态机只有一个输入的时候，k可以为0或1。但若状态机有多个输入时，则k表示所有可能输入取值的集合。

从定义8.1可知，若状态S_i和S_j是等价的，则它们对应的k继承（对所有的k）也是等价的。根据这一事实，我们可以阐明将状态机状态作为一个集合考虑所涉及的最小化过程，并把该集合划分成由明显不等价的子集合组成的几个部分。

定义8.2 *一个划分由一个或多个块组成，该划分中的每个块包含一个状态子集，这些状态可能有些是等价的，但在给定块中的状态必须与其他块中的状态不等价。*

让我们先假定所有的状态是等价的；从而形成初始划分P_1，在P_1中所有的状态都在同一个块中。第二步，我们将形成划分P_2，在P_2中状态的集合被划分成若干块，以使得每块中的状态产生相同的输出。很明显，产生不同输出的状态不可能是等价的。接着，通过测试每个块中各个状态的k继承是否仍在同一块中，我们继续形成新的划分。这样，在每次新的划分中形成了新的块。当一个新的划分和前面的划分相同时，这个过程便结束了。然后，在任一分块中的所有状态都等价。为了说明这个过程，请考虑以下的例子（例8.6）。

例8.6 图8-51所示是某特定有限状态机的状态表。为了求出最少状态个数，让我们应用划分的方法。第一次划分时，将所有的状态放在同一个块中，即

$$P_1 = (ABCDEFG)$$

第二次划分，将具有不同输出的状态（注意这个状态机是摩尔型的）分开，这意味着必须将状态A、B、D与状态C、E、F、G区分开。因此新的划分有两个块：

$$P_2 = (ABD)\,(CEFG)$$

现在我们必须考虑每个块中的0继承和1继承。对于块（ABD），0继承分别是（BDB）。既然所有的继承状态都在P_2的同一个块中，所以仍然应该假定状态A、B、D也许是等价的。（ABD）的1继承是（CFG）。既然这些继承也在P_2的同一个块中，所以我们仍旧应该保留（ABD）作为P_3的一个块。接下去考虑块（$CEFG$）。该块的0继承分别是（$FFEF$），它们也在P_2所在的同一个块中。其1继承是（$ECDG$）。因为这些状态并不都在P_2的同一个块中，这意味着（$CEFG$）中至少有一个状态和其他状态不等价。具体地说，状态F和状态CEG一定不同，因为它的1继承是D，和C、E、G在不同的块。因此，

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

图8-51 例8.6中的状态表

$$P_3 = (ABD)\ (CEG)\ (F)$$

重复这个过程产生以下这些划分。（ABD）的0继承为在同一个P_3块下的（BDB）。1继承为在不同块中的（CFG）。因为F与C和G不在同一块中，因此状态B不能等价于A和D。（CEG）的0继承和1继承分别为（FFF）和（ECG）。两个子集都在块P_3中。因此，

$$P_4 = (AD)\ (B)\ (CEG)\ (F)$$

如果用相同的方法来检查块（AD）和（CEG）的0继承和1继承。我们发现：

$$P_5 = (AD)\ (B)\ (CEG)\ (F)$$

因为$P_5 = P_4$，而且没有新的块产生，因此每个块中的状态都是等价的。若某些块中的状态不等价，则它们的k继承必然在不同的块中。因此状态A和D等价，C、E、G等价。既然每个块可以用一个状态表示，所以实现图8-51中状态表定义的有限状态机只需四个状态。若用符号A来表示图中的A和D，用符号C来表示状态CEG，则缩减后的状态表将如图8-52所示。■

上述状态最小化处理帮助我们找到一种解决方法，只需要用两个触发器便可以实现最简状态表中的四个状态，而原先的设计却需要用三个触发器。尽管并非一定能找到最简状态，但为了电路的简化，我们总是希望能用更少状态的FSM来实现电路。

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

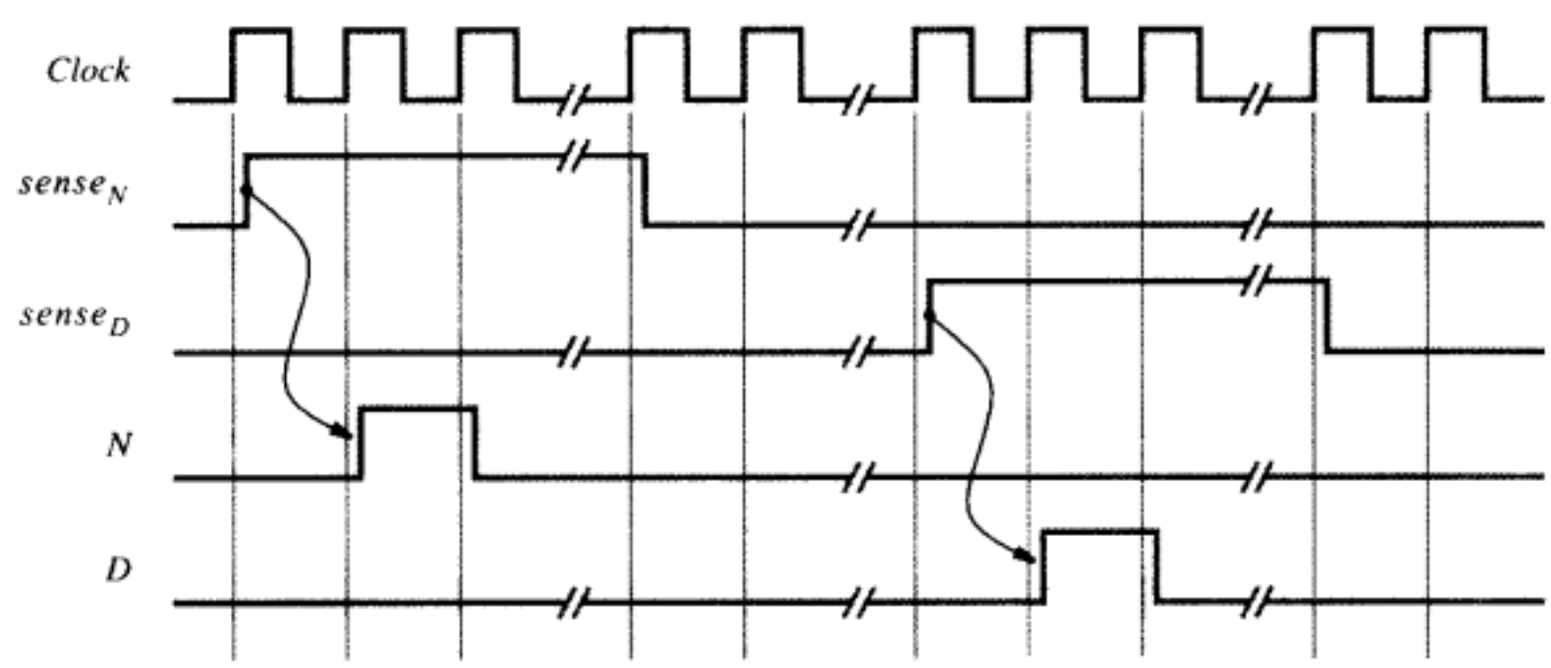
图8-52 例8.6中的最小化的状态表

状态最小化的概念是基于这样一个事实：虽然两个有限状态机并不相同，但对所有可能的输入，就产生的输出而言，其表现行为是完全一致的。这两个状态机尽管其实现电路可能有很大不同，但在功能上却是等价的。总的来说，决定任意两个有限状态机是否等价并不容易。我们的最小化过程保证化简后的有限状态机与原先的状态机在功能上是一致的。我们鼓励读者通过使用CAD工具，实现图8-51和图8-52状态表表示的两个状态机，并进行仿真，在这个过程中体会到实际功能等价的直观感觉。

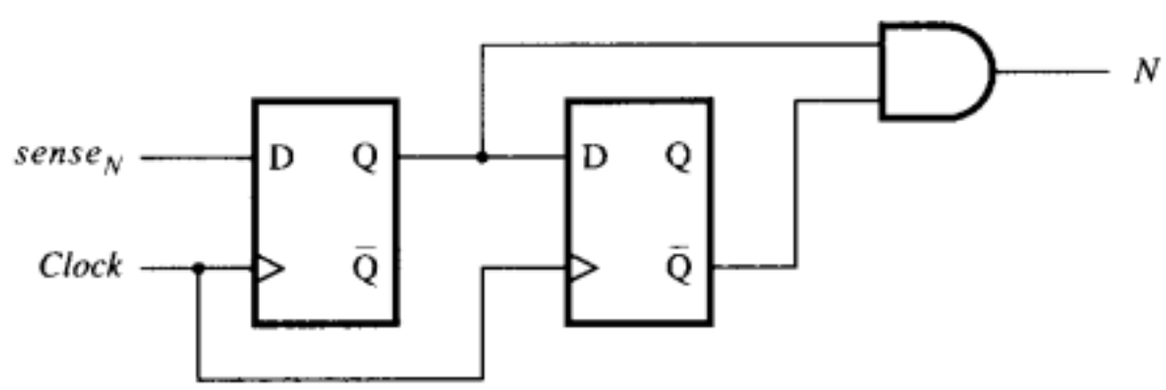
例8.7 下面再举一个最小化的例子，即考虑设计一个可用在自动售货机上的实际时序电路。假设投币式自动售货机根据以下条件出售糖果：

- 机器接受5分和1角硬币。
- 从机器中得到一块糖果需要15分。
- 如果投入20分，机器将不会找零，但会记录多收了5分，并等待消费者的第二次选购。

自动售货机的所有电信号都与时钟信号*Clock*的正跳变沿同步。时钟信号的准确频率对这个例子并不重要，但我们还是假设时钟周期为100ns。当自动售货机的硬币接收装置检测到投入5分或1角硬币时，分别产生两个信号，$sense_D$或$sense_N$。因为硬币接收装置是一种机械装置，因此其响应速度相对于电路来说慢得多， 投入硬币可以使$sense_D$或者$sense_N$置1，并维持许多个时钟周期。我们假设硬币接收装置还产生另外两个信号，分别为*D*和*N*。在$sense_D$变为1后，信号*D*被置为1只维持一个时钟周期，在$sense_N$变为1后，信号*N*被置为1只维持一个时钟周期。*Clock*、$sense_D$、$sense_N$、*D*、*N*之间的时序关系如图8-53a所示。波形上的截断标志表示$sense_D$和$sense_N$被置为1可以维持很多个时钟周期。同时，连续两次投入硬币之间可能会有任意长时间的间隔。因为硬币接收装置每次只能接收一个硬币，因此不可能同时把*D*和*N*置为1。图8-53b说明了如何由$sense_N$产生信号*N*。



a) 时序图



b) 产生*N*的电路

图8-53 自动售货机的信号

基于这些假设，我们可以用相当直接的方式画出如图8-54所示的初始状态转移图。有限状态机的输入为*D*和*N*，起始状态为*S*1。只要$D = N = 0$，状态就会保持在*S*1，用标有 $\overline{D}\cdot\overline{N}=1$ 的弧来表示。投入一角硬币随即导致把状态变为*S*2，投入五分硬币会把状态变为*S*3。在这两种情况下，投入的硬币的总金额都不到15分，不足以令自动售货机给出糖果。这是用输出信号*z*等于0来表示的，例如在*S*2/0和*S*3/0斜线符号右边的0就表示输出$z = 0$。由于$D = N = 0$，状态机继续留在状态*S*2或*S*3，直到再投入另一枚硬币，状态机才离开状态*S*2或*S*3。在状态*S*2，投入一枚5分硬币将使状态转移到*S*4，而投入一枚一角硬币将使状态转移到*S*5。在两种状态下，投入硬币的总金额足够后，都可以让输出机制送出糖果；因此状态节点标有*S*4/1和*S*5/1。在状态*S*4中，总金额为15分，也就是说在下一个有效时钟沿，状态机会转到复位状态*S*1。因为状态机仅在状态*S*4逗留100ns，远比投入一枚硬币的时间短，所以离开状态*S*4的条件 $\overline{D}\cdot\overline{N}$ 可以保证为真。

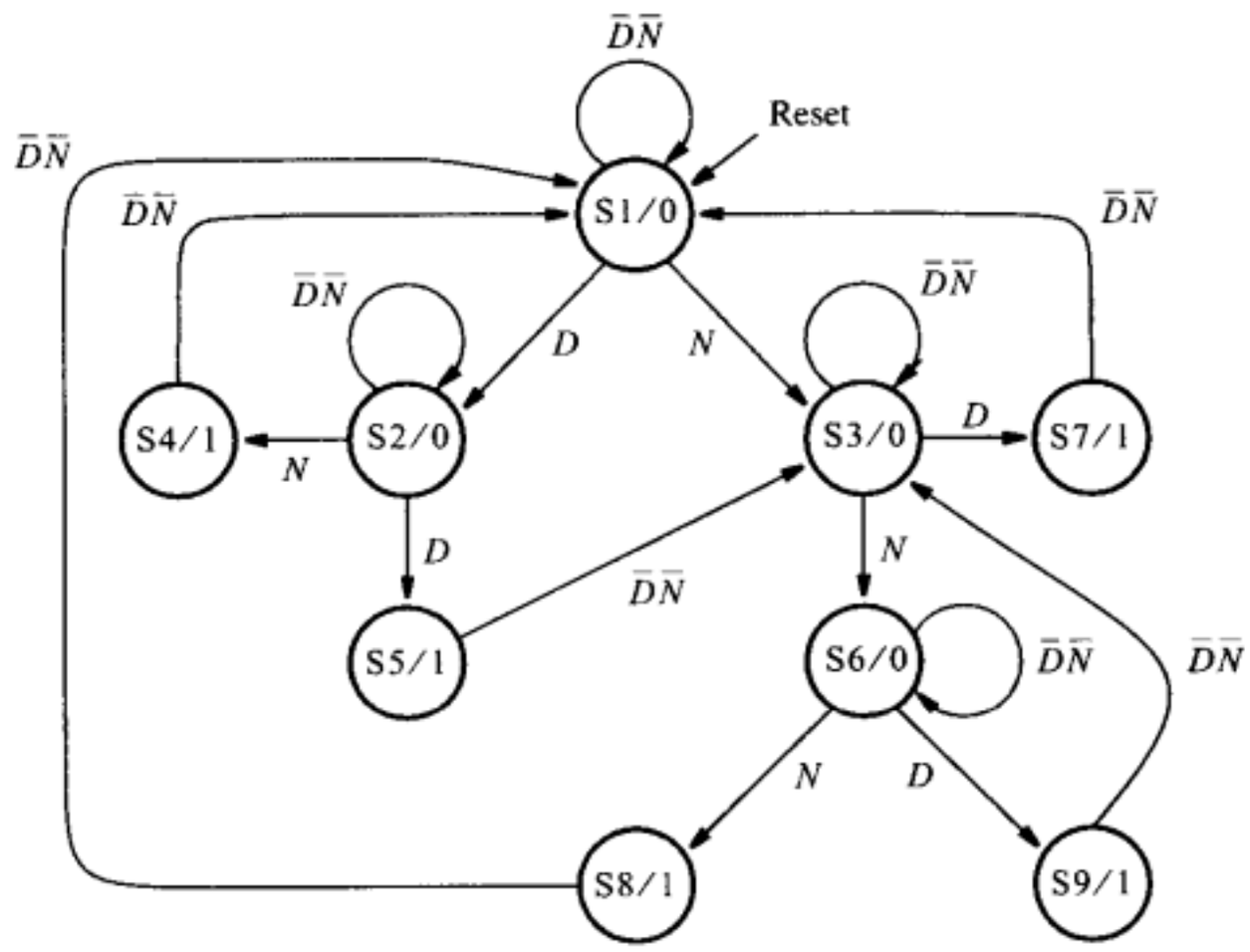


图8-54 例8.7的状态图

状态$S5$表示投入的硬币总金额已经达到20分。糖果被自动售货机送出，在下一个时钟沿，状态机转移到状态$S3$，该状态表示还有5分的余额。根据同样的推理，从状态从$S3$将可能迁移到$S6$，$S7$，$S8$直到$S9$。这样就可以完整地画出期望的有限状态机的状态转移图。包含相同信息的状态表见图8-55所示。

当前状态	下一状态 DN = 00	01	10	11	输出 z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

图8-55 例8.7的状态表

请注意在图8-55所示的状态表中，条件$D = N = 1$被表示为无关项。还请注意，在状态$S4$、$S5$、$S7$、$S8$ 和$S9$中也含有其他无关项。对于投入一枚新的硬币而言，状态机从一个状态转移到另外一个状态所需的时间实在太短了，所有这些无关项都是对应于由此造成的那些没有必要检查信号D和信号N的场合的。

使用最小化过程，我们得到如下的划分：

$$P_1 = (S1, S2, S3, S4, S5, S6, S7, S8, S9)$$
$$P_2 = (S1, S2, S3, S6)(S4, S5, S7, S8, S9)$$
$$P_3 = (S1)(S3)(S2, S6)(S4, S5, S7, S8, S9)$$
$$P_4 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$
$$P_5 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$

最后的划分有5个块。用$S2$表示与其等价的$S6$，用$S4$表示与其等价的$S7$和$S8$，用$S5$表示与其等价的$S9$，由此得到图8-56所示的最小化的状态表。实现该状态表的实际电路，可以用前面章节介绍的方法来设计。

当前状态	下一状态				输出
	$DN=00$	01	10	11	z
S1	S1	S3	S2	—	0
S2	S2	S4	S5	—	0
S3	S3	S2	S4	—	0
S4	S1	—	—	—	1
S5	S3	—	—	—	1

图8-56 例8.7的最小化的状态表

在本例中，我们使用直接方法画出最初的状态图，然后使用划分的过程来进行最小化。图8-57用状态图的方式表达了图8-56中状态表的信息。观察该状态转移图，读者也许就能明白，使用下面的推理方法直接得到最优的状态图是切实可行的。

假设每个状态对应于已投入硬币的不同总金额。状态$S1$、$S3$、$S2$、$S4$、$S5$分别对应于已投入硬币总金额为0、5、10、15、20分的情况。对状态做这样的解释，我们不难定义想设计的有限状态机的转移曲线。在实际工作中，设计者通常能设计出没有大量冗余状态的初始设计。

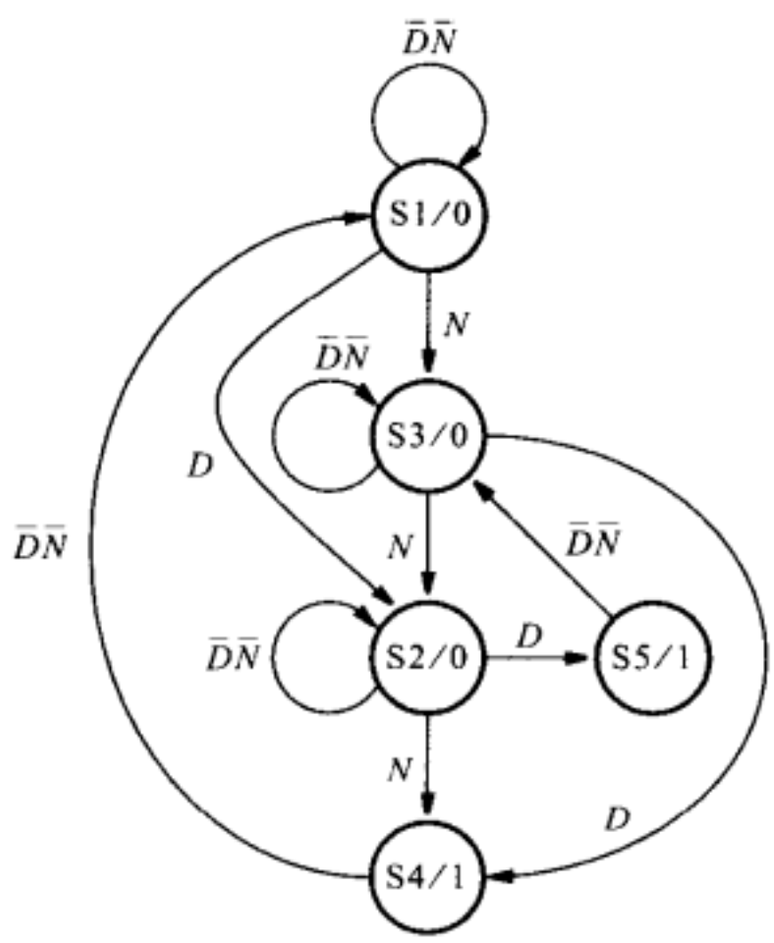


图8-57 例8.7的最小化的状态图

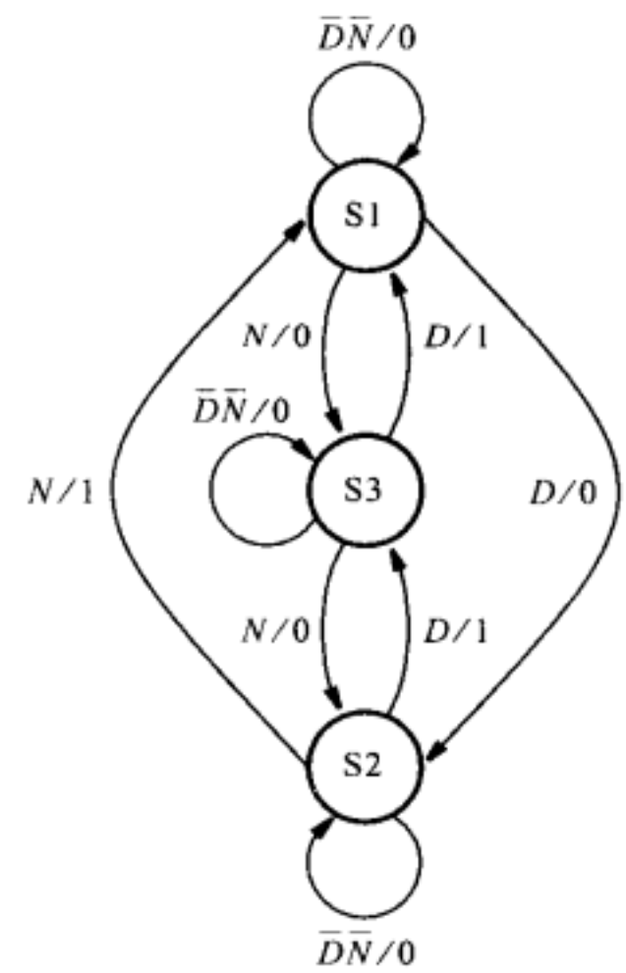


图8-58 例8.7的米利型有限状态机

我们已经找到一种需要五种状态的解决方法，该方法是实现要求的销售控制任务的摩尔型有限状态机的最少状态个数。由8.3节我们知道，虽然并不一定能得到整体上更简洁的设计，但米利型有限状态机所需的状态数比摩尔型状态机少。如果使用米利模型，我们可以取消图8-57中的状态$S4$和$S5$。结果如图8-58所示。这种方法仅需三种状态，但输出函数变得更复杂。我们建议读者通过完成图8-57和图8-58所示有限状态机的设计步骤来比较两种实现方式的复杂性。■

8.6.2 未完全指定的有限状态机

在状态表的所有输入都已经指定的情况下，状态最小化的划分方法可以很好地工作。图8-51中定义的有限状态机就是这种情况。这种类型的有限状态机称为*完全指定的有限状态机*。若状态表的一个或多个输入并未被完全指定，对应于无关项，则该有限状态机被称为*不完全指定的有限状态机*。图8-55所示的就是这种类型有限状态机的一个例子。如例8.7中所见到的，该状态机的这种划分方法也是可行的。但就总体而言，当涉及不完全指定的有限状态机时，这种划分方法的效果比较差，见例8.8所作的解释。

例8.8 考虑图8-59所示的有限状态机，因为我们假定在状态B或G时，不会出现输入$w = 1$的情况，所以存在四个未指定的项。相应的，这两种情况下，既没有状态转移，也没有指定输出值。该状态机与图8-55表示的状态机的重要不同在于：该状态机有一些输出未被指定，而图8-55表示的状态机的所有输出都是确定的。

当前状态	下一状态		输出 z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	—	0	—
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	—	0	—

图8-59 例8.8中的不完全指定的状态表

例8.6和例8.7所说明的针对摩尔型有限状态机的状态划分最小化过程也适用于对米利型有限状态机进行状态化简。若对所有输入取值，两个状态其对应的输出相同，则这两个状态被认为是等价的，因而被放在某个划分的同一个块内。为了进行划分，我们可以给未指定的输出假定一个特定值。在不知道应该赋为0还是1的情况下，我们先把这些输出都定为0。所以前两个划分为：

$$P_1 = (ABCDEFG)$$
$$P_2 = (ABDG)\ (CEF)$$

注意状态A、B、D和G在同一个块内，因为它们对于$w = 0$和$w = 1$的输出都等于0。状态C、E和F因为具有相同的输出行为也在同一个块中；若$w = 0$，则它们都产生输出$z = 0$，若$w = 1$，则它们都产生输出$z = 1$。继续状态的划分过程得到如下这些划分：

$$P_3 = (AB)\ (D)\ (G)\ (CE)\ (F)$$
$$P_4 = (A)\ (B)\ (D)\ (G)\ (CE)\ (F)$$
$$P_5 = P_4$$

结果得到由6个状态指定的有限状态机。

接着考虑另一种假设，即把图8-59中未指定的两个输出全都置为1。由此得到如下的状态划分：

$$P_1 = (ABCDEFG)$$
$$P_2 = (AD)\ (BCEFG)$$
$$P_3 = (AD)\ (B)\ (CEFG)$$
$$P_4 = (AD)\ (B)(CEG)\ (F)$$
$$P_5 = P_4$$

这种解决方法只需要四个状态。显然，把未指定的输出定为0还是1，对于状态简化是相当重要的。

我们将不再深入研究不完全指定的有限状态机的状态最小化问题。正如我们已经提到的，直接在定义8.1的基础上开发等价状态的最小化技术是可行的。在很多关于逻辑设计的书中，都可以找到有关这种方法的介绍[2, 8-10, 12-14]。

最后，有必要强调：减少给定有限状态机的状态数并不一定能得到简化的设计。有趣的是，在8.2节中曾讨论过的状态分配，对于实现电路的简化，要远比状态最小化产生的效果大。在现代的设计环境下，设计者可以依靠CAD工具高效率地完成有限状态机的设计。 ■

8.7 用时序电路方法设计计数器

在本节中，我们将讨论如何用时序电路的一般设计方法进行计数器电路的设计。我们由

第7章得知，计数器可以通过级联的触发器和某些门逻辑来实现，其中每级把输入脉冲计数除以2。为了使所举的例题比较简单，我们选择计数值很小的计数器，然后说明如何将简单的设计扩展为计数值很大的计数器。想要设计的计数器的技术指标如下：

- 计数序列为0, 1, 2, …, 6, 7, 0, 1,…。
- 输入信号为w。在每个时钟周期里，都考虑这个输入信号的值。若$w = 0$，则当前计数值保持不变；若$w = 1$，则计数值加1。

可以使用前面章节中介绍的方法，把计数器设计为同步时序电路。我们首先介绍经典的用手工设计计数器的方法，这种方法展示了涉及设计过程的基本概念。然后，我们将展示如何用CAD工具来完成设计任务，用CAD工具设计比用手工简单得多，同时指出如何在实际工作中完成设计任务。

8.7.1 模8计数器的状态图和状态表

图8-60给出了所需计数器的状态图，每个状态对应一个计数值。图中，状态A对应计数值0，状态B对应计数值1，依此类推。我们标出了实现计数序列所需的状态转换。请注意输出信号被指定为只取决于给定时间的计数器状态，因此这是摩尔型时序电路。该状态图可以表示为图8-61所示的状态表。

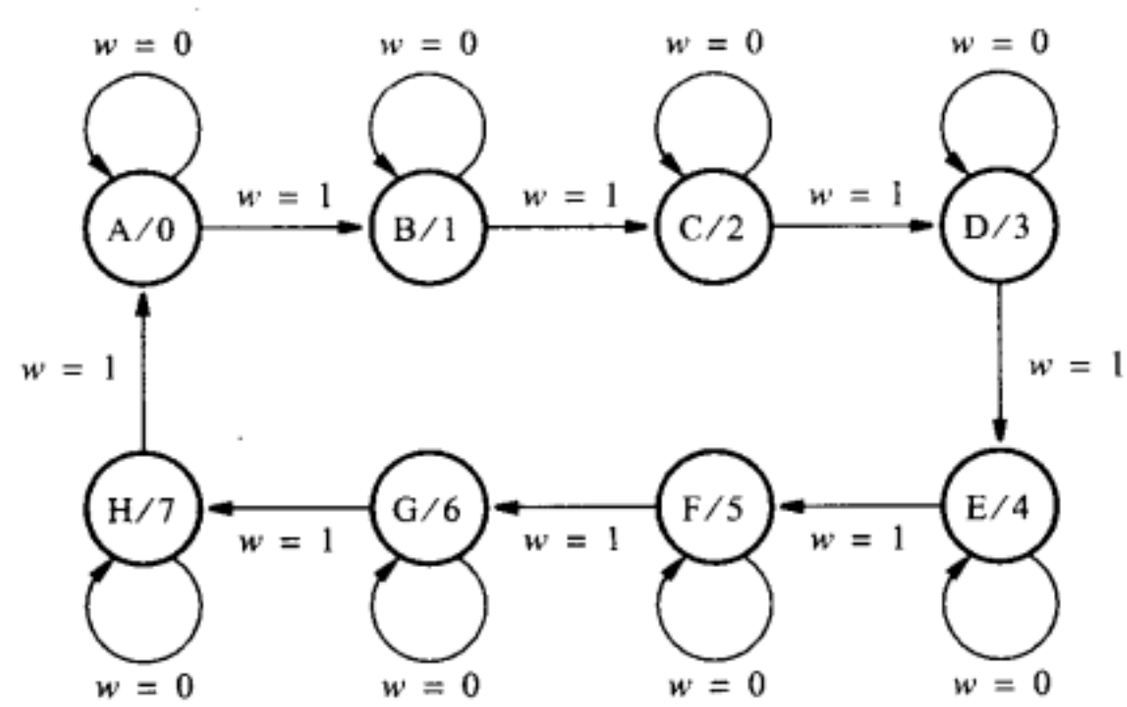


图8-60 计数器的状态图

当前状态	下一状态		输出
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

图8-61 计数器的状态表

8.7.2 状态分配

8个状态需要三个状态变量来表示。令表示当前状态的状态变量分别为y_2、y_1和y_0。令Y_2、Y_1和Y_0表示对应的下一个状态的函数。最方便（而且最简单）的状态分配方式是把计数器在

该状态下应该输出的二进制数分配给对应的状态作为状态编码。因此，所需的输出信号将与表示状态变量的信号一致。这就导致了图8-62所示的状态分配表。

设计的最后步骤是选择触发器的类型，并推导出控制触发器输入的表达式。最容易的方式是使用D触发器。我们首先尝试这种方法，然后展示使用JK触发器的方法。在这两种情况下，触发器都必须是跳变沿触发的，以保证在一个时钟周期期间，触发器只发生一次变化。

	当前状态 $y_2y_1y_0$	下一状态 $w=0$ $Y_2Y_1Y_0$	下一状态 $w=1$ $Y_2Y_1Y_0$	计数 $z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

图8-62 计数器的状态分配表

8.7.3 用D触发器实现的计数器

当使用D触发器来实现有限状态机的时候，每个下一状态的表达式Y_i都连接到实现状态变量y_i的触发器的D输入端。下一状态表达式可以从图8-62所示的信息中推导出。用图8-63所示的卡诺图，可以得到下面的逻辑表达式：

$$D_0 = Y_0 = \overline{w}y_0 + w\overline{y}_0$$
$$D_1 = Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$$
$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$$

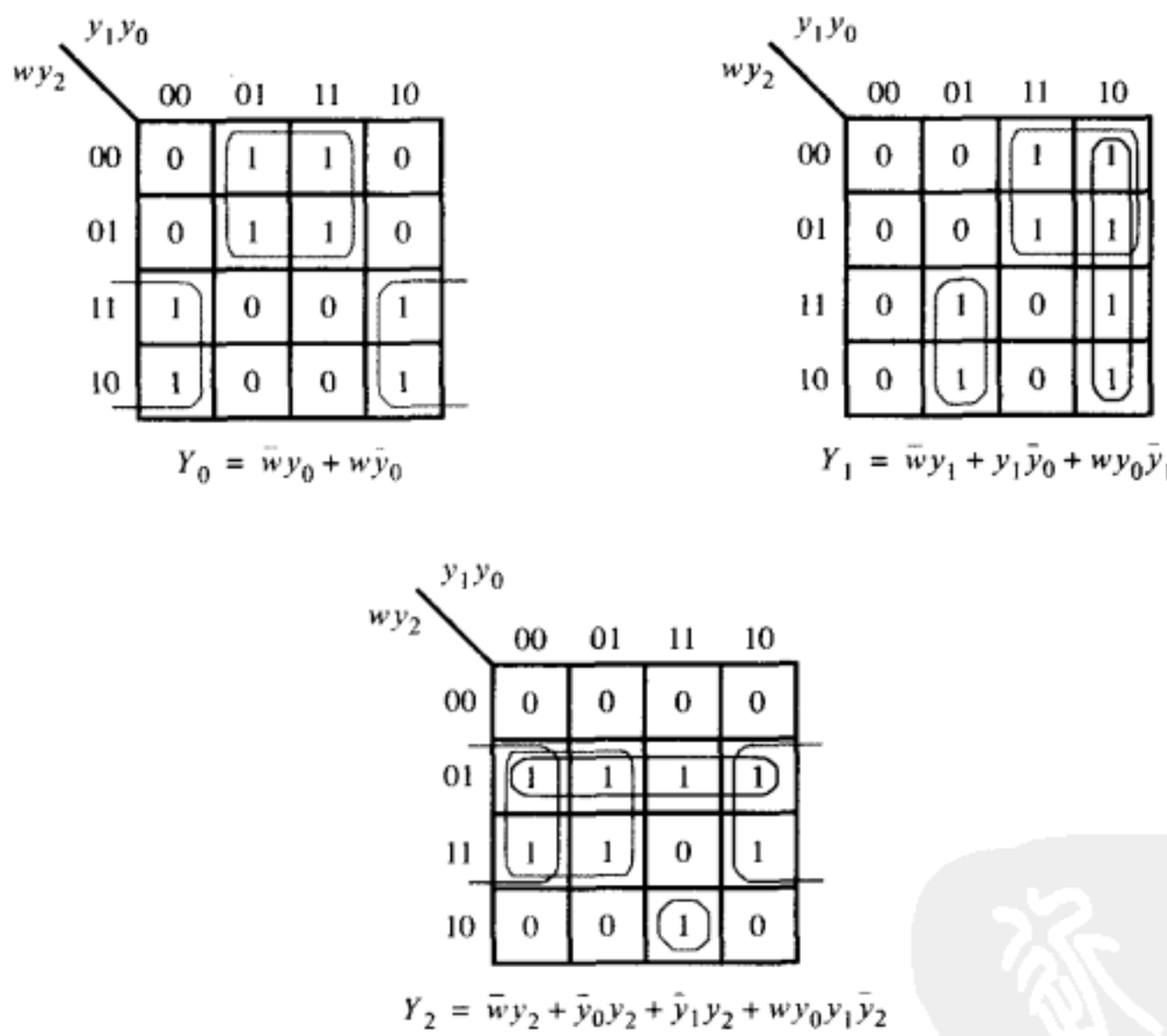


图8-63 由D触发器构成计数器的卡诺图

计数器最后实现的电路如图8-64所示。如何将该电路扩展为更大的计数器的方法很不明显，因为并没有从D_0、D_1、D_2的表达式中得到明确设计方法。然而，我们可以把这些表达式改写为如下表达式：

$$D_0 = \overline{w}y_0 + w\overline{y}_0$$
$$= w \oplus y_0$$

$$
\begin{aligned}
D_1 &= \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1 \\
&= (\overline{w} + \overline{y}_0)y_1 + wy_0\overline{y}_1 \\
&= \overline{wy_0}y_1 + wy_0\overline{y}_1 \\
&= wy_0 \oplus y_1 \\
D_2 &= \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2 \\
&= (\overline{w} + \overline{y}_0 + \overline{y}_1)y_2 + wy_0y_1\overline{y}_2 \\
&= \overline{wy_0\overline{y}_1}y_2 + wy_0y_1\overline{y}_2 \\
&= wy_0y_1 \oplus y_2
\end{aligned}
$$

由以上逻辑表达式，我们可以清楚地看出如何推导出如图7-24所示的电路。

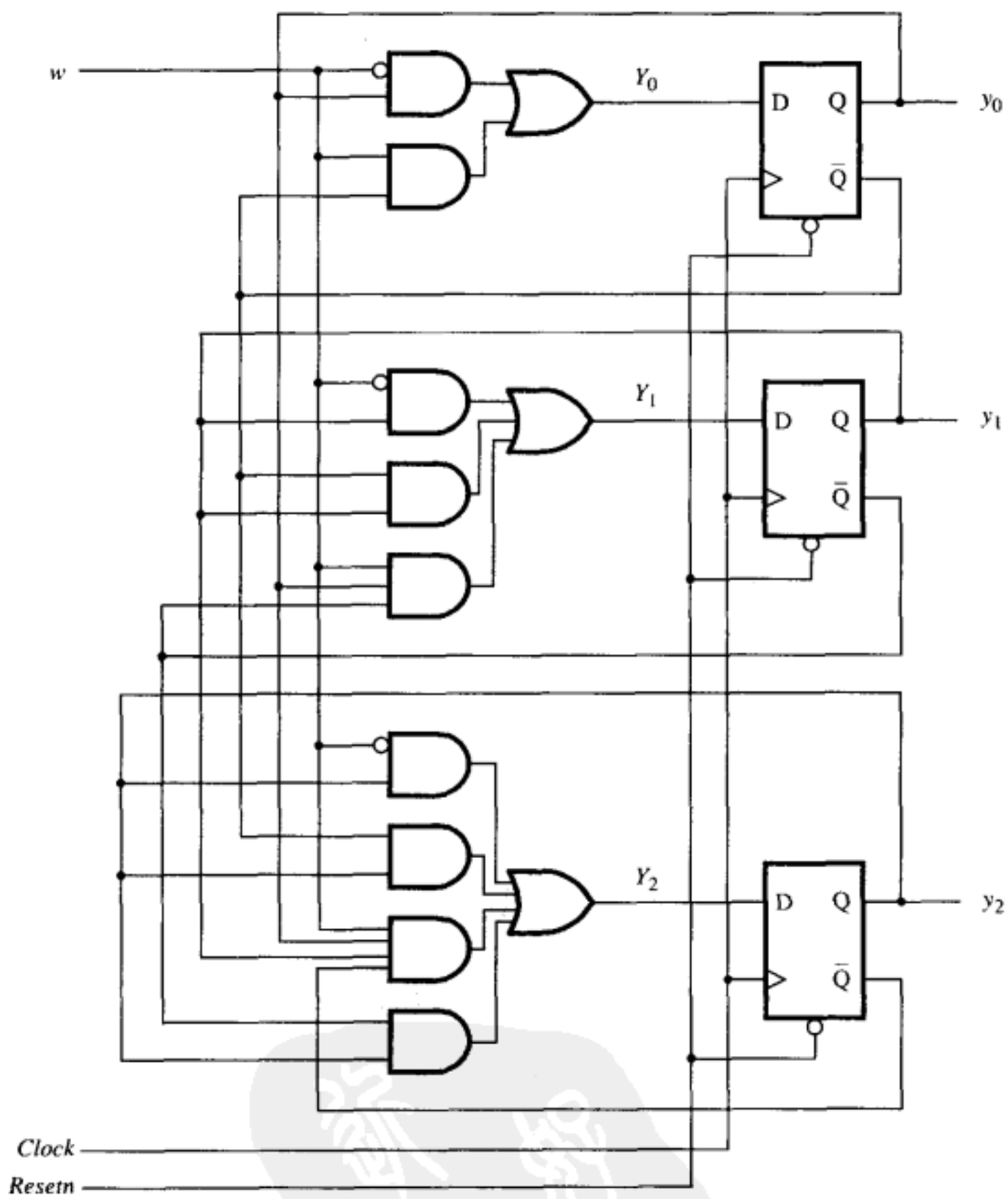


图8-64 用D触发器实现的计数器电路图

8.7.4 用JK触发器实现的计数器

另一种实现计数器的方法是使用JK触发器。使用JK触发器来实现由图8-62所示状态表指定的时序电路，需要为每个触发器的J和K输入端推导出逻辑表达式。下面列出的控制是需要的：

- 若触发器处于状态0，并保持状态0，则$J = 0$，$K = d$（d表示K可以等于0或1）。
- 若触发器处于状态0，并要转为状态1，则$J = 1$，$K = d$。

• 若触发器处于状态1，并保持状态1，则$J = d$，$K = 0$。

• 若触发器处于状态1，并要转为状态0，则$J = d$，$K = 1$。

遵循这几条指导原则，我们便可以创建一个真值表来指定设计中三个触发器其J和K端所需要的输入值。图8-65展示了根据图8-62所示的表修改后的状态分配表，其中包含了J、K的输入表达式。为了理解这个表是如何推得的，首先考虑第一行，在该行，当前的状态为$y_2 y_1 y_0 = 000$。若$w = 0$，则下一个状态仍然是$Y_2 Y_1 Y_0 = 000$。因此每个触发器的当前值都是0，并都保持0。这意味着三个触发器的输入控制均为$J = 0$，$K = \mathrm{d}$。继续观察第一行，如果$w = 1$，下一个状态会是$Y_2 Y_1 Y_0 = 001$。因此触发器y_2和y_1仍然保持为0，控制信号$J = 0$和$K = d$。然而通过控制$J = 1$和$K = d$，使触发器y_0必定完成从0到1的转变。通过考虑当前状态$y_2 y_1 y_0$和转移到新状态$Y_2 Y_1 Y_0$所必须的控制信号，用同样方法可以推导得到表中的其他行。

	当前状态 $y_2y_1y_0$	触发器输入								计数 $z_2z_1z_0$
		$w = 0$				$w = 1$				
		$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	
A	000	000	0*d*	0*d*	0*d*	001	0*d*	0*d*	1*d*	000
B	001	001	0*d*	0*d*	*d*0	010	0*d*	1*d*	*d*1	001
C	010	010	0*d*	*d*0	0*d*	011	0*d*	*d*0	1*d*	010
D	011	011	0*d*	*d*0	*d*0	100	1*d*	*d*1	*d*1	011
E	100	100	*d*0	0*d*	0*d*	101	*d*0	0*d*	1*d*	100
F	101	101	*d*0	0*d*	*d*0	110	*d*0	1*d*	*d*1	101
G	110	110	*d*0	*d*0	0*d*	111	*d*0	*d*0	1*d*	110
H	111	111	*d*0	*d*0	*d*0	000	*d*1	*d*1	*d*1	111

图8-65 用JK触发器实现的计数器的激励表

状态分配表本质上就是状态表，只是状态表中的每个状态都经由状态变量进行了编码。当用D触发器来实现有限状态机时，状态分配表中的下一个状态输入项直接对应于必须施加到D触发器输入端的信号。若使用其他类型的触发器，情况就不一定相同。触发器必须被激励从而使状态自当前状态转移到下一个状态，以这样的形式所给出的状态信息的表，通常称为*激励表*。图8-65中的激励表表明如何使用JK触发器。在很多书中，甚至由D触发器实现的状态机也使用激励表这个术语，在这种场合，激励表与状态分配表是同义词。

推出图8-65所示的激励表后，也就有了一张输入为y_2、y_1、y_0和w，输出为J_2、K_2、J_1、K_1、J_0、K_0的真值表。我们可以利用如图8-66所示的方法求出这些输出的表达式。所求得的表达式为：

$$J_0 = K_0 = w$$
$$J_1 = K_1 = wy_0$$
$$J_2 = K_2 = wy_0 y_1$$

根据以上表达式，我们可以画出计数器的电路，见图8-67所示。显然，这种设计方法可以很容易地扩展到更大的计数器。表达式$J_n = K_n = wy_0 y_1 \cdots y_{n-1}$定义了计数器中每级的电路。请注意，实现乘积项$wy_0 y_1 \cdots y_{n-1}$的与门个数随着级数的递增而增加。当我们顺着计数器的最低位逐级向高位推导时，计数器中表示低位的项可以用来简化表示高位的项。由此便可以求出更规则的电路结构。简化后的表达式如下：

$$J_2 = K_2 = (wy_0)\, y_1 = J_1 y_1$$
$$J_n = K_n = (wy_0 \cdots y_{n-2})\, y_{n-1} = J_{n-1}\, y_{n-1}$$

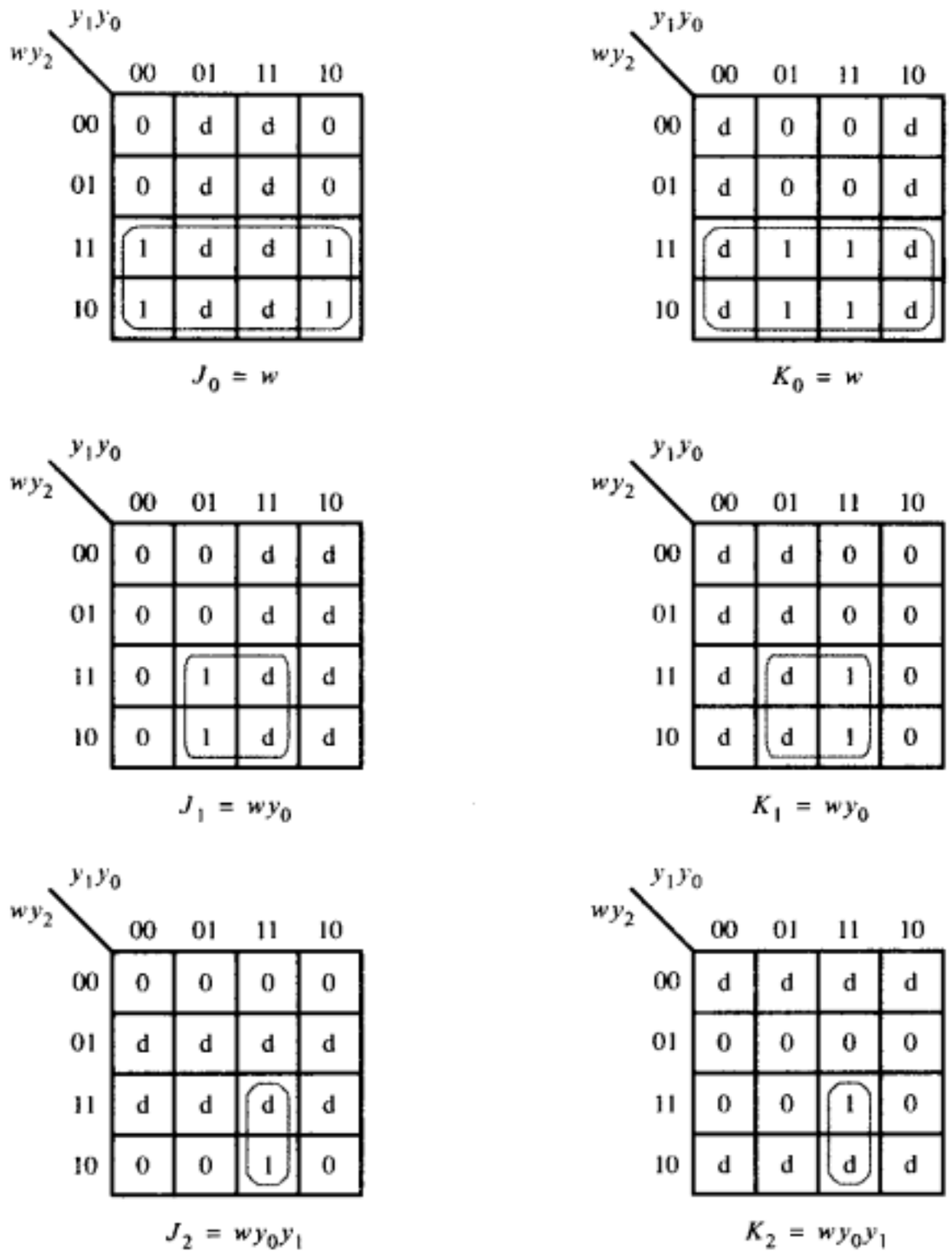


图8-66 计数器里的JK触发器的卡诺图

使用这种递推的方式，计数器电路的实现如图8-68所示。在这个电路中除了第一级外，其余所有的级看起来很相似。请注意，这个电路与图7-23中的电路具有相同的结构，因为把触发器的输入J和K连接在一起，它就变成了一个T触发器。

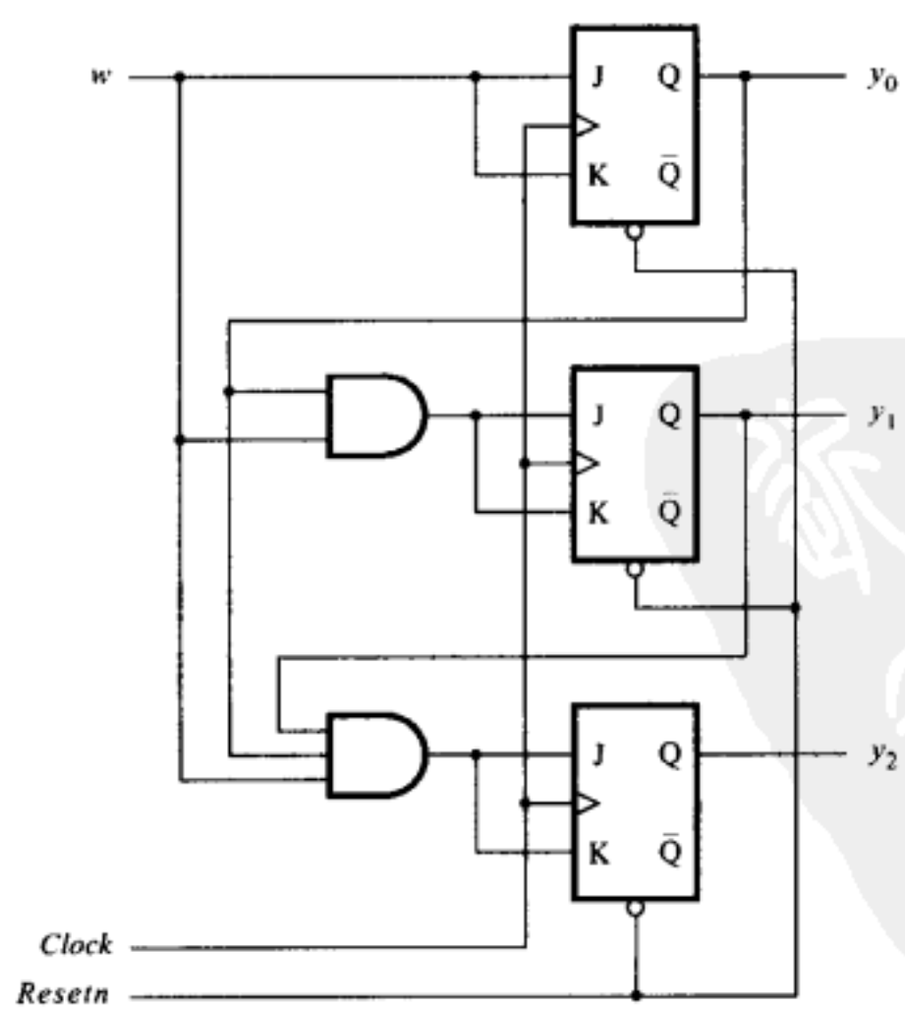


图8-67 用JK触发器实现的电路图

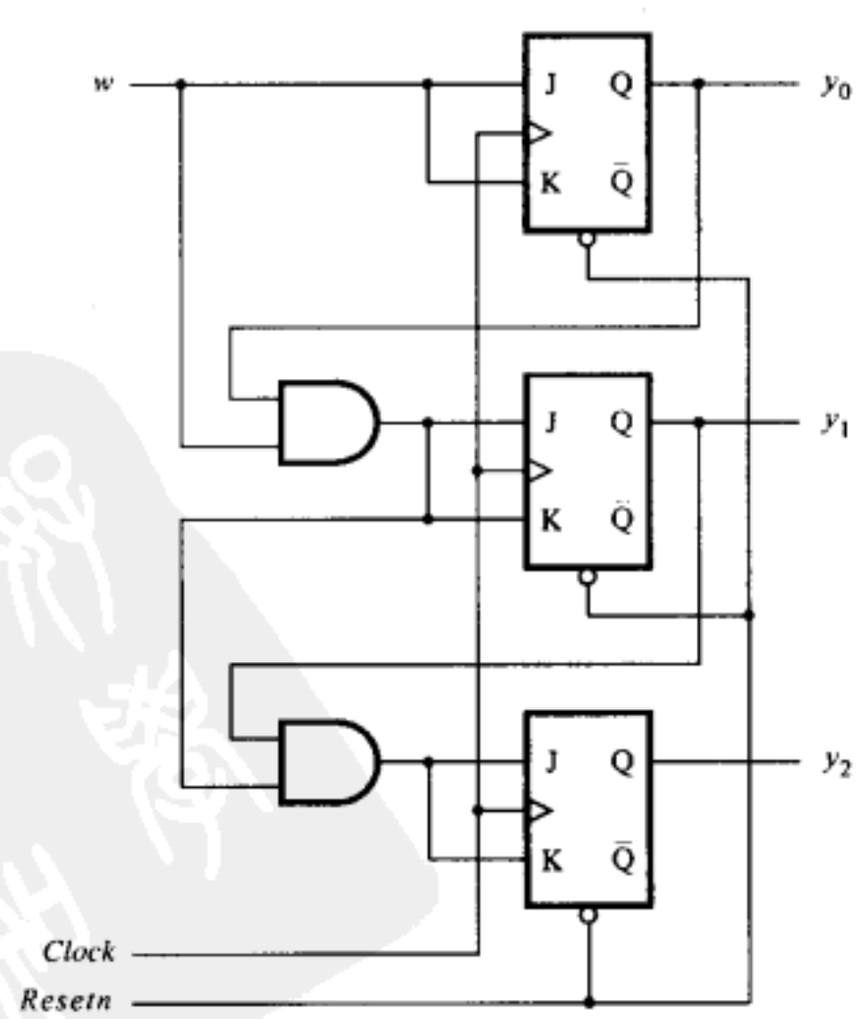


图8-68 计数器的分解实现的电路图

8.7.5 举例：一个不同的计数器

学习了普通的计数器设计方法之后，我们现在使用这些知识来设计一个略有不同的类似计数器的电路。假设我们想设计一个可对输入信号线w上的脉冲进行计数的三位计数器。但是，计数值并非0, 1, 2, 3, 4, 5, 6, 7, 0, 1,…，而是0, 4, 2, 6, 1, 5, 3, 7, 0, 4,…。这些计数值由触发器的值直接输出，而不需要用更多的门。换言之，Count = $Q_2Q_1Q_0$。

因为我们要对输入信号线w上的脉冲进行计数，所以可用w作为触发器的时钟输入。因而计数器电路总是处于使能状态，并且只要在w信号线上出现下一个脉冲，计数器应该改变其状态。需要的计数器可以使用有限状态机方式直接进行设计。图8-69和图8-70给出了需要的状态表和相应的状态分配表。使用D触发器，可以求出表示下一个状态的表达式如下：

$$
\begin{aligned}
D_2 &= Y_2 = \bar{y}_2 \\
D_1 &= Y_1 = y_1 \oplus y_2 \\
D_0 &= Y_0 = y_0\bar{y}_1 + y_0\bar{y}_2 + \bar{y}_0 y_1 y_2 \\
&= y_0(\bar{y}_1 + \bar{y}_2) + \bar{y}_0 y_1 y_2 \\
&= y_0 \oplus y_1 y_2
\end{aligned}
$$

根据以上逻辑表达式实现的电路如图8-71所示。

当前状态	下一状态	输出 $z_2z_1z_0$
A	B	000
B	C	100
C	D	010
D	E	110
E	F	001
F	G	101
G	H	011
H	A	111

图8-69 本例的状态表

当前状态 $y_2y_1y_0$	下一状态 $Y_2Y_1Y_0$	输出 $z_2z_1z_0$
000	100	000
100	010	100
010	110	010
110	001	110
001	101	001
101	011	101
011	111	011
111	000	111

图8-70 图8-69的状态分配表

读者应该将本电路与图7-24所示的一般递增计数器电路进行比较。取那个计数器电路的前三级，将输入*Enable*置为1，并令*Clock* = w。这两个电路除了在计数位的顺序上稍有不同外，本质上是一致的。在图7-24中，最上面的触发器对应于该计数器的最低有效位，但在图8-71中，最上面的触发器对应于该计数器的最高有效位。这并非只是一个巧合。在图8-70中，需要的计数器被定义为*Count* = $y_2y_1y_0$。然而，若把定义状态的最高位与最低位倒过来看，并仍旧用二进制数解释，便有*Count* = $y_0y_1y_2$，这样状态A, B, C, …, H的值分别为0, 1, 2, …, 7。这些值与普通的三位递增计数器的计数值是一致的。

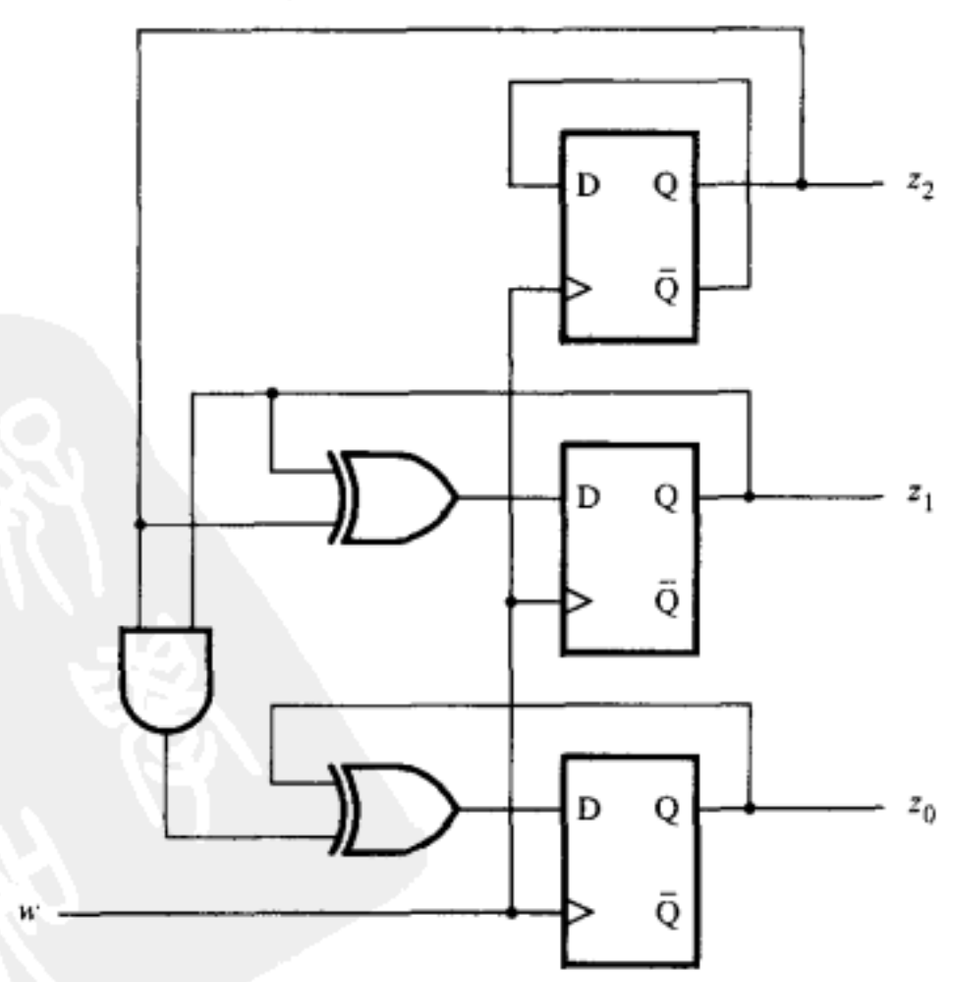


图8-71 实现图8-70状态分配表的电路

8.8 用作仲裁器电路的有限状态机

在本节中，我们将展示一个比前面的例子稍微复杂一些的有限状态机（FSM）的设计。该FSM的目的是产生控制信号，将给定系统中的共享资源合理地分配给多个设备使用，每次只允许一个设备能使用该共享资源。假设系统的所有信号仅能在时钟正跳变沿时刻改变。每个设备都提供一个输入信号给FSM，该输入信号叫做*请求信号*，FSM发给每个设备单独一个输出信号，称为*认可信号*。某设备通过设置请求信号来申请使用资源。只要共享资源未被占用，FSM应对所有发出的请求信号都予以考虑。基于优先级方案，FSM从中选择一个发出请求信号的设备并输出其认可信号。一旦该设备使用完资源后，它应撤回请求信号。

我们假设系统有三台设备，分别称为设备1、设备2和设备3。我们很容易理解有限状态机（FSM）是如何处理多个设备请求共享资源的。请求信号被命名为r_1、r_2和r_3，认可信号被叫做g_1、g_2和g_3。设备被分配给不同的优先级，例如设备1的优先级最高，设备2的优先级其次，设备3的优先级最低。因此，若有多个设备同时发出请求信号，则FSM将把认可信号发给提出请求的设备中优先级最高的设备。

所需的有限状态机（FSM），被设计成摩尔型FSM，其状态图如图8-72所示。最初，FSM复位后处于状态*Idle*。没有发出任何认可信号，共享资源未被任何设备占用。FSM还有另外三个状态，分别叫做*gnt*1、*gnt*2和*gnt*3。每个这样的状态将会把一个认可信号发给一个设备。

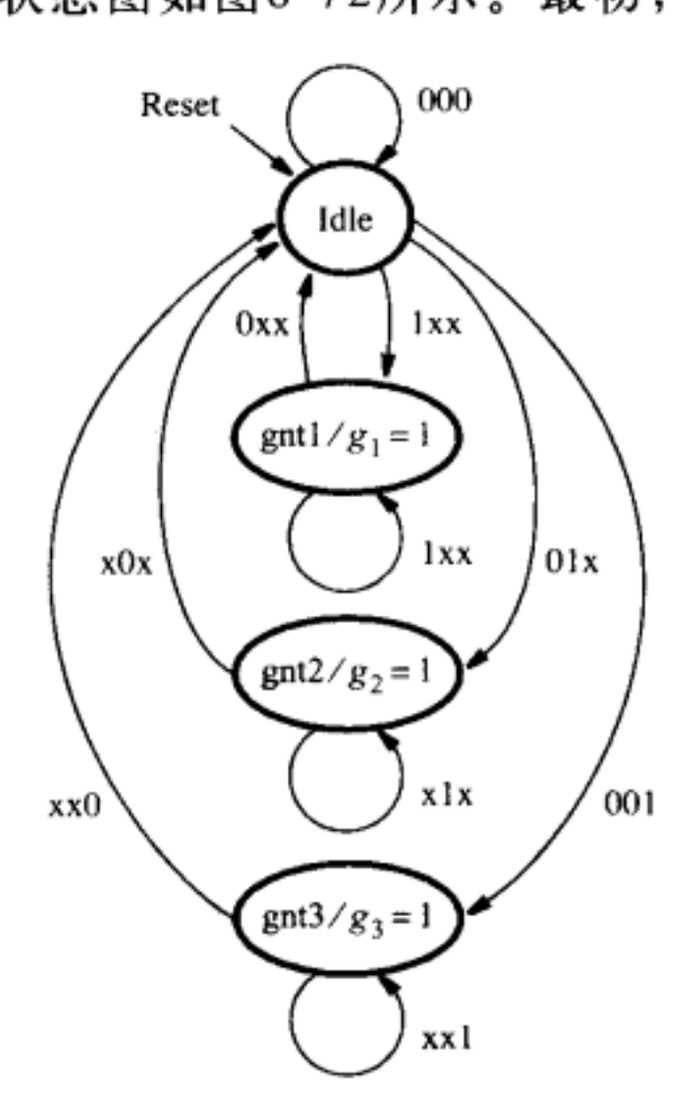


图8-72 仲裁器的状态图

当请求信号全都为0的时候，有限状态机（FSM）留在状态*Idle*。在状态图中，条件$r_1 r_2 r_3 = 000$用标有000的弧线表示。当一个或多个请求信号变为1时，FSM根据优先级转移到相应的认可状态。当r_1有效时，设备1将收到认可信号，因为r_1有最高的优先级。这种情况用转移到状态*gnt*1（并使认可信号$g_1 = 1$）的标有1xx的弧线来说明。1xx的含义是指当请求信号r_1为1时，考虑优先级的原因，r_2和r_3的信号成为无关项。和以前一样，我们使用符号x来表示对应变量的值可以为0或1。只要r_1为1，FSM 便留在状态*gnt*1。当$r_1 = 0$时，弧线0xx会在下个正跳变沿使状态返回*Idle*，并使$g_1 = 0$。若此时其他请求信号有效，则FSM在下一个时钟正跳变沿便转移到新的认可状态。

使状态转移到*gnt*2的弧线用01x做标记。这个标记具有优先级的含义，因为它表示$r_2 = 1$但$r_1 = 0$的情况。类似地，进入状态*gnt*3 的条件为001，这表示只有一个提出请求的信号r_3。

图8-73给出了状态图的另一种形式。与图8-72唯一的不同是弧线上的标记不同。图8-73使用了一个更简单的表示方法，看起来更直观。状态*Idle*到状态*gnt*1的弧线标注为r_1，而不是1xx。该标记表示如果$r_1 = 1$，则有限状态机（FSM）转移到状态*gnt*1，不必考虑其他输入。状态*Idle*转移

到$gnt2$的条件为$\bar{r}_1 r_2$，表示条件为$r_1 r_2 = 01$，r_3则为无关项。标记状态图上的弧线并没有标准的方法。一些设计者喜欢图8-72的方法，而另一些人则习惯类似于图8-73的方式。

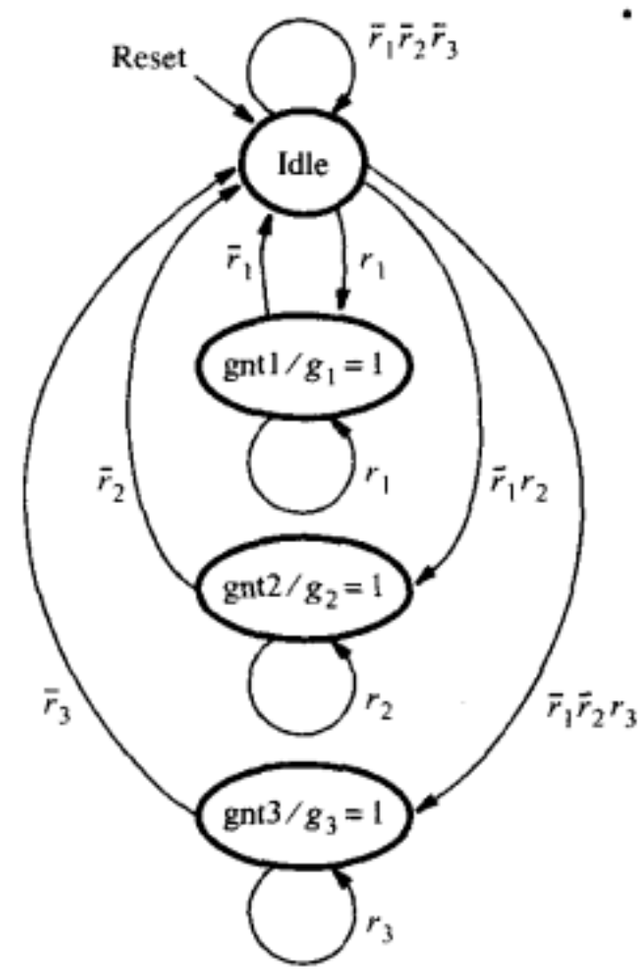


图8-73 仲裁器的另一种形式的状态图

图8-74给出了状态机的Verilog代码。三个请求信号和认可信号分别用三位的向量r和g表示。状态转移采用图8-29代码所用风格的**case**语句描述。如状态*Idle*所示，使用嵌套的**case**语句更容易描述所需的优先级关系。若$r_1 = 1$，则下一个状态为$gnt1$。若$r_1 = 0$，则执行另一个部分，它会继续考虑，若$r_2 = 1$，则下一个状态为$gnt2$。只有当所有优先级高的信号都没有提出申请时，低级别请求信号才有机会被认可。

进入每个认可状态的转移都是直接的。只要$r_1 = 1$，有限状态机便停留在状态$gnt1$；若$r_1 = 0$，则下一个状态便为*Idle*。其他认可状态都具有相同的结构。

认可信号g_1、g_2和g_3在代码结尾处定义。当状态机在状态$gnt1$时，g_1的值被置为1；否则g_1被置为0。类似地，其他的认可信号也仅在进入对应的认可状态时才置1。

```
module arbiter (r, Resetn, Clock, g);
   input [1:3] r;
   input Resetn, Clock;
   output wire [1:3] g;
   reg [2:1] y, Y;
   parameter Idle = 2'b00, gnt1 = 2'b01, gnt2 = 2'b10, gnt3 = 2'b11;

   // Next state combinational circuit
   always @(r, y)
      case (y)
         Idle:   casex (r)
                    3'b000:   Y = Idle;
                    3'b1xx:   Y = gnt1;
                    3'b01x:   Y = gnt2;
                    3'b001:   Y = gnt3;
                    default:  Y = Idle;
                 endcase
         gnt1:   if (r[1])  Y = gnt1;
                 else       Y = Idle;
         gnt2:   if (r[2])  Y = gnt2;
                 else       Y = Idle;
         gnt3:   if (r[3])  Y = gnt3;
                 else       Y = Idle;
         default:           Y = Idle;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Resetn == 0)  y <= Idle;
      else  y <= Y;

   // Define output
   assign g[1] = (y == gnt1);
   assign g[2] = (y == gnt2);
   assign g[3] = (y == gnt3);

endmodule
```

图8-74 仲裁器的Verilog代码

8.8.1 仲裁器电路的实现

现在我们开始讨论用CPLD和FPGA实现的仲裁器有什么差别。由于这两种器件的结构是不同的，因此仲裁器在两种不同器件上的实现其内部结构会有所不同，状态机越复杂，这种不同就越明显。为此，我们将一个8个设备的更大的仲裁器在两种不同的器件上实现，看看它们的结构究竟有什么不同。仲裁器的请求信号分别为r_1，r_2，…，r_8，认可信号分别为g_1，g_2，…，g_8。我们不能直接使用图8-74中的代码，必须修改一下才能用。如何将图8-74中的代码扩展到8个设备的仲裁器的代码是很容易理解的，因此这里不准备再讲解代码的编写。

在CPLD中实现

我们首先考虑在CPLD上实现仲裁器。为了表示有限状态机的9个状态，综合工具使用四个触发器，分别为y_4、y_3、y_2和y_1。复位状态*Idle*被分配为代码$y_4y_3y_2y_1 = 0000$。其他的状态分别被编码为：*gnt*1 = 0001，*gnt*2 = 0010，*gnt*3 = 0100，*gnt*4 = 1000，*gnt*5 = 0011，*gnt*6 = 0101，*gnt*7 = 0110和*gnt*8 = 1001。

综合工具为什么做出这种选择的原因并不明显。工具考虑很多种不同的状态分配方案，并将电路成本最低的那一种方案作为最终选择。对于CPLD上的实现，综合工具尝试选择在最终电路中乘积项最少的状态分配方案。

为了理解电路的复杂性，我们需要考察产生认可信号和状态触发器输入信号的逻辑表达式。每个认可信号的表达式都可以由产生认可信号的状态编码直接得到。例如，状态*gnt*8被编码为1001，因此可得 $g_8 = y_4\bar{y}_3\bar{y}_2y_1$ 。

反馈到状态触发器输入端的逻辑更为复杂。例如，由工具得到的触发器y_4的输入Y_4的表达式为：

$$Y_4 = \bar{r}_1\bar{r}_2\bar{r}_3\bar{r}_5\bar{r}_6\bar{r}_7r_8\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4 + \bar{r}_1\bar{r}_2\bar{r}_3r_4\bar{y}_1\bar{y}_2\bar{y}_3 + r_8\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4 + r_4\bar{y}_1\bar{y}_2\bar{y}_3y_4$$

图8-75给出了在CPLD上实现的时序仿真波形。为简单起见，只显示三个请求信号r_1、r_2和r_8，认可信号也只显示g_1、g_2和g_8。仿真一开始，状态机复位，三个请求信号r_1、r_2和r_8都置1。尽管在状态图中没有显示，所有其他的请求信号都被置为0。状态机首先进入状态*gnt*1并且认可信号$g_1 = 1$。在r_1变为0之后，状态机将返回状态*Idle*。在下一个时钟周期，转移到状态*gnt*2，使认可信号$g_2 = 1$。在r_2变为0之后，状态机将返回状态*Idle*，然后转移到状态*gnt*8，使认可信号$g_8 = 1$。这个仿真结果表明，用Verilog代码综合生成的电路很好地满足了优先级需求。

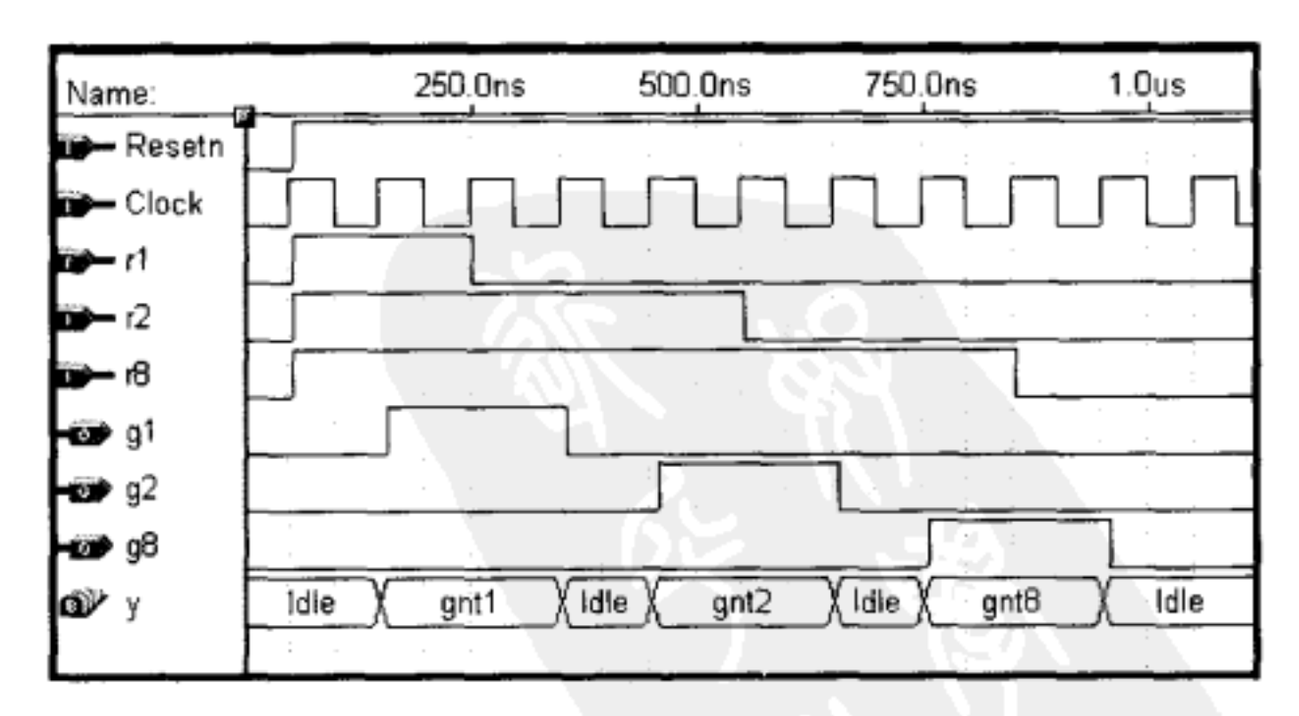


图8-75 仲裁器电路的仿真结果

图8-76展示的是仿真结果的部分细节。对显示的波形做了一些安排，使得当$g_8 = 1$时，只显示信号*Clock*、g_8和*y*的波形。仿真结果展示，在状态机进入状态*gnt*8后，需要延迟约7ns后

才产生g_8信号。这个延迟时间对应于函数 $g_8 = y_1\bar{y}_2\bar{y}_3y_4$ 这个四输入与门所需要的时间。我们将在8.8.2节中展示：有可能优化实现电路的时序，使得状态机刚一进入认可状态便立即输出认可信号。

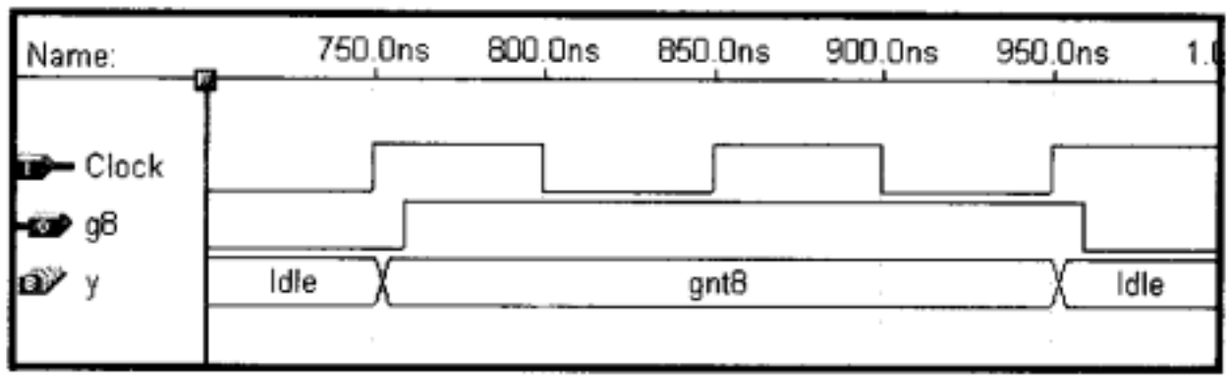


图8-76 仲裁器电路的输出延时

FPGA中的实现

下面考虑如何在FPGA芯片中实现上述仲裁器有限状态机（FSM）。在FPGA上运行的由综合工具生成的有限状态机电路用了9个触发器$y_9, y_8, \cdots, y_1$表示9个状态，而不是如在CPLD上那样，用四个触发器来表示9个状态。状态分配分别为：*Idle* = 000000000, *gnt*1 = 110000000, *gnt*2 = 101000000, *gnt*3 = 100100000, *gnt*4 = 100010000, *gnt*5 = 100001000, *gnt*6 = 100000100, *gnt*7 = 100000010, *gnt*8 = 100000001。该状态分配方案非常类似于独热编码，与独热编码唯一的不同点是最左边的触发器的输出y_9为1。这样便可实现简单的复位，当所有触发器都被复位时，所有状态变量都变为0，即进入*Idle*状态。

在4.6节中，我们曾经讨论过某些类型的芯片存在逻辑门扇入数有限的问题。我们曾提到在这类芯片中，有大量输入的逻辑函数必须分解为若干个较小的函数。对于一个有限状态机而言，这意味着若连接每个状态触发器的逻辑电路有很多输入的话，则有可能需要几级门结构。这会增加电路的传播延迟，并降低运算速度。对于上述在CPLD中实现的仲裁器有限状态机，我们展示了触发器y_4输入端的逻辑表达式。若该表达式用四输入查找表FPGA实现，则电路中总共需要8个查找表，其中有3个查找表是串联的。

通过比较，上述采用9个状态变量的状态分配方案可以得到更简单得多的电路。例如，对于触发器y_8的输入，综合工具会产生 $Y_8 = r_1y_8 + r_1\bar{y}_9$ 。因为该表达式只有四个输入，所以可以用一个四输入查找表来实现。同理，另外8个下一状态表达式也变得相对简单。为了观察状态分配对有限状态机运行速度的影响，我们对比在FPGA芯片上实现的两种版本的电路：一种如上所述，用9个状态触发器实现；另一个用4个触发器，其状态分配如前面CPLD中的状态分配。比较的结果是当使用9个状态变量时，仲裁器有限状态机可以达到的最高时钟频率为88.5MHz，然而当使用4个状态变量时，最高时钟频率仅为54.1MHz。请注意，电路的运行速度不但取决于特定的目标芯片，也取决于CAD工具中的综合选项。

我们必须考虑生成认可信号所需逻辑电路的复杂性。当使用9个触发器时，认可信号的产生非常容易，不需要任何其他信号，因为每个触发器的输出就是认可信号。例如，$g_8 = y_1$。

8.8.2 有限状态机输出延迟的最小化

图8-76展示了由CPLD芯片实现的仲裁器电路产生的、有传播延迟的认可信号。一旦电路进入认可状态，某个认可信号将延迟7ns后变为有效。该延迟是由生成认可信号的电路所引起的，认可信号的改变依赖于状态触发器的值。然而，正如我们在FPGA实现中所看到的那样，当使用独热编码时，每个认可信号都是由状态寄存器中某个触发器直接输出的。因此，不需要添加任何其他电路便可以产生认可输出信号。

图8-77所示的时序仿真波形是对由CPLD实现的采用独热编码的仲裁器电路进行仿真后得到的。从电路进入某认可状态起，到产生某认可输出信号之间有一个很小的延时。认可信号必须通过CPLD芯片中的缓冲器，才能将触发器的输出和CPLD封装的管脚之间连接起来。因为信号的传递需要时间，所以出现了一个很小的延迟，大约只有2ns。由于设计技术指标通常要求输出的延迟尽可能地小，所以这类时序优化工作在实践中往往是由时序电路的设计者完成的。

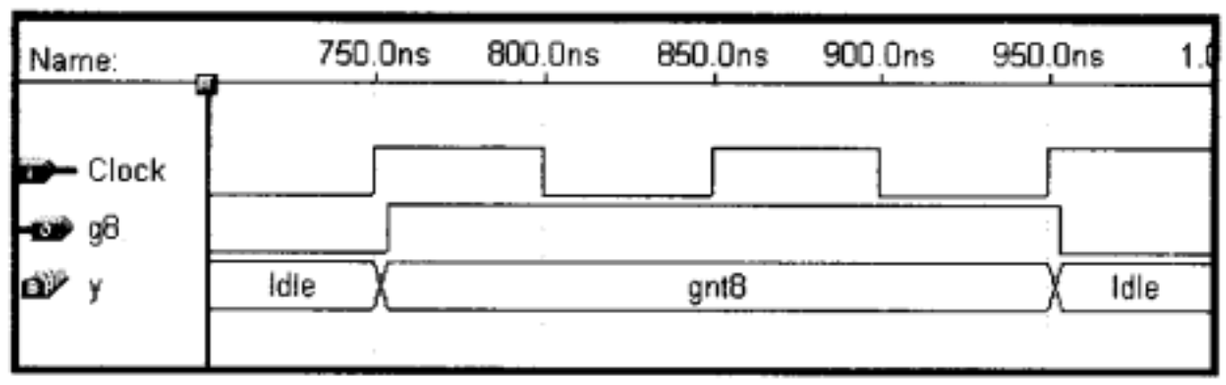


图8-77 使用独热编码的输出延时

8.8.3 小结

上面的仲裁器有限状态机是一种在许多类型的系统中都很有用的实际电路。通过总线与多个设备相连的计算机系统就是这样一个例子。该仲裁器必须做一些修改才能应用于这种计算机系统。由于优先级方案的缘故，高优先级设备有可能任意长时间地阻止低优先级设备收到认可信号。这种情况通常被称为低优先级设备的“饥饿”。把仲裁器FSM修改为能考虑到这个问题并不困难（参见练习题8.37）。

8.9 同步时序电路的分析

除了知道如何设计同步时序电路，设计者还必须掌握分析现成电路行为的能力。分析任务要比综合任务简单得多。本节将展示如何进行同步时序电路的分析。为了分析电路，我们只需要将综合过程的步骤颠倒过来。触发器的输出表示当前的状态，触发器的输入决定了电路将进入的下一状态。根据这些信息，我们可以建立电路的状态分配表。由这个状态分配表可以进一步得到状态表和相应的状态转移图，并给每个状态起一个名字。我们将在下面的例子中看到，电路中使用的触发器类型是必须考虑的因素。

例8.9（D触发器） 图8-78给出了包含两个D触发器的有限状态机。令y_1和y_2为当前状态变量，Y_1和Y_2为下一状态变量。表示下一状态和输出的表达式分别为：

$$Y_1 = w\bar{y}_1 + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

既然有两个触发器，有限状态机就（FSM）有四个状态。分析FSM的好起点是首先假设触发器的初始状态为$y_1 = y_2 = 0$。从Y_1和Y_2的表达式，我们可以得到如图8-79a所示的状态分配表。例如，在表的第一行$y_1 = y_2 = 0$。然后$w = 0$导致$Y_1 = Y_2 = 0$，而$w = 1$导致$Y_1 = 1$和$Y_2 = 0$。该状态的输出为$z = 0$。另一行可以用相同的方式得到。把状态定名为A、B、C和D便可以得到图8-79b所示的状态表。从这个表中可以清楚地看到，在复位后，若输入w连续三个周期保持为1，则有限状态机就会产生输出$z = 1$。因此，该FSM所起的作用是检测输入w线上串行序列的序列检测器。

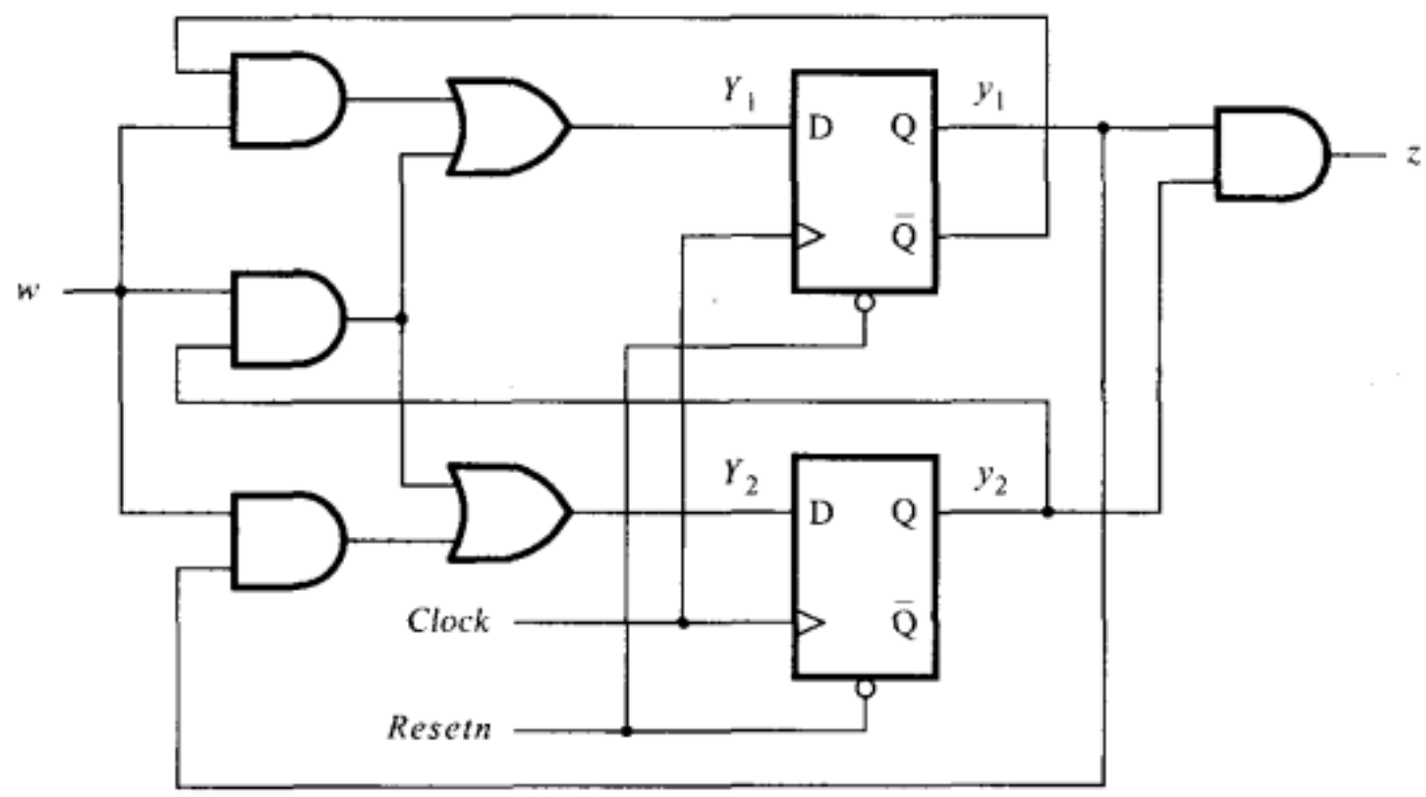


图8-78 例8.9的电路

当前状态 y_2y_1	下一状态 $w=0$ Y_2Y_1	下一状态 $w=1$ Y_2Y_1	输出 z
00	00	01	0
01	00	10	0
10	00	11	0
11	00	11	1

a) 状态分配表

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

b) 状态表

图8-79 图8-78所示电路的状态分配表和状态表

例8.10（JK触发器） 现在考虑图8-80所示的有两个JK触发器的电路。输入与触发器的关系表达式为：

$$J_1 = w$$
$$K_1 = \bar{w} + \bar{y}_2$$
$$J_2 = wy_1$$
$$K_2 = \bar{w}$$

输出为 $z = y_1 y_2$。

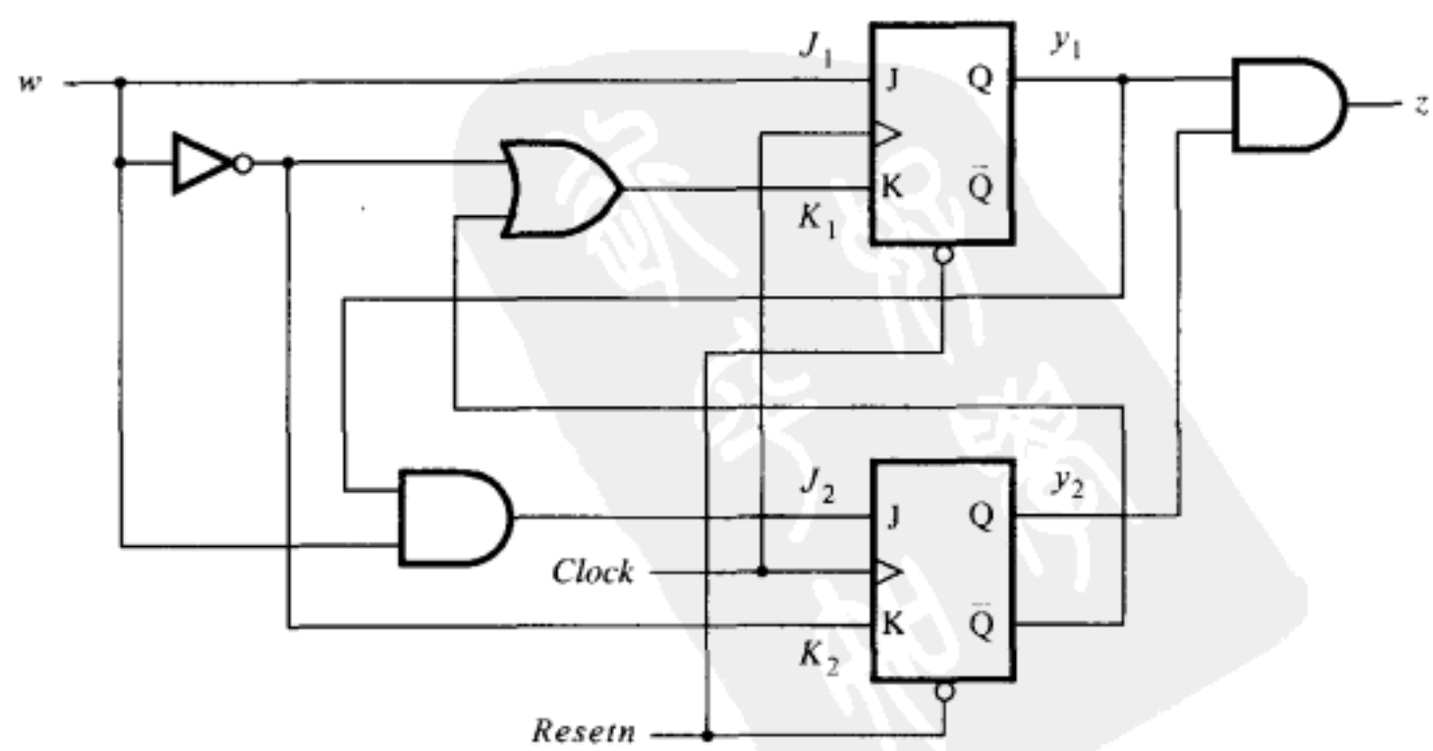


图8-80 例8.10中的电路

从这些表达式可以得到图8-81所示的激励表。通过分析表中的各个项，我们可以建立状态分配表。例如，若当前状态$y_2y_1 = 00$且$w = 0$，因为$J_2 = J_1 = 0$和$K_2 = K_1 = 1$，所以两个触发器都将保持0状态；由此$Y_2 = Y_1 = 0$。假定当前状态$y_2y_1 = 00$且$w = 1$，因为$J_2 = K_2 = 0$和$J_1 = K_1 = 1$，所以触发器y_2不变而触发器y_1将置为1；最终得到$Y_2 = 0$和$Y_1 = 1$。若当前状态$y_2y_1 = 01$且$w = 0$，因为$J_2 = J_1 = 0$和$K_2 = K_1 = 1$，所以触发器y_1复位，且使状态$y_2y_1 = 00$；最终得到$Y_2 = Y_1 = 0$。类似地，若当前状态$y_2y_1 = 01$且$w = 1$，则$J_2 = 1$和$K_2 = 0$将y_2置1；由此$Y_2 = 1$，同时$J_1 = K_1 = 1$使$y_1 = 0$，从而得到$Y_1 = 0$。这使当前状态变成$y_2y_1 = 10$。完成这个过程，我们发现最后得到的状态分配表与图8-79a中的相同。因此，我们得到结论：图8-78和图8-80所示的电路实现的是相同的有限状态机。

当前状态 y_2y_1	触发器输入				输出 z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
00	01	01	00	11	0
01	01	01	10	11	0
10	01	01	00	10	0
11	01	01	10	10	1

图8-81 图8-80中电路的激励表

例8.11（混合使用不同类型的触发器） 在一个电路中可以混合使用不同类型的触发器。图8-82所示的电路就是由一个D触发器和一个T触发器所组成的。电路的表达式为：

$$D_1 = w(\bar{y}_1 + y_2)$$
$$T_2 = \bar{w}y_2 + wy_1\bar{y}_2$$
$$z = y_1y_2$$

根据这些表达式可以得到图8-83所示的激励表。因为y_2是T触发器的输出，所以仅当$T_2 = 1$时，y_2才会改变状态。因此，若$y_2y_1 = 00$且$w = 0$，因为$T_2 = D_1 = 0$，则电路的状态不会发生改变。若$y_2y_1 = 01$且$w = 1$，则$T_2 = 1$，将使y_2变为1；$D_1 = 0$使$y_1 = 0$，因此有$Y_2 = 1$且$Y_1 = 0$。若$w = 0$且$y_2y_1 = 10$或11时，则$T_2 = 1$。在以上两种情况下$D_1 = 0$，因此T触发器的状态从1变为0，同时将D触发器清零，也就是说下一个状态为$Y_2Y_1 = 00$。完成了这个分析，我们再次得到图8-79a中的状态分配表。因此该电路也是图8-79b所示的状态表所代表的有限状态机的另一种实现方法。

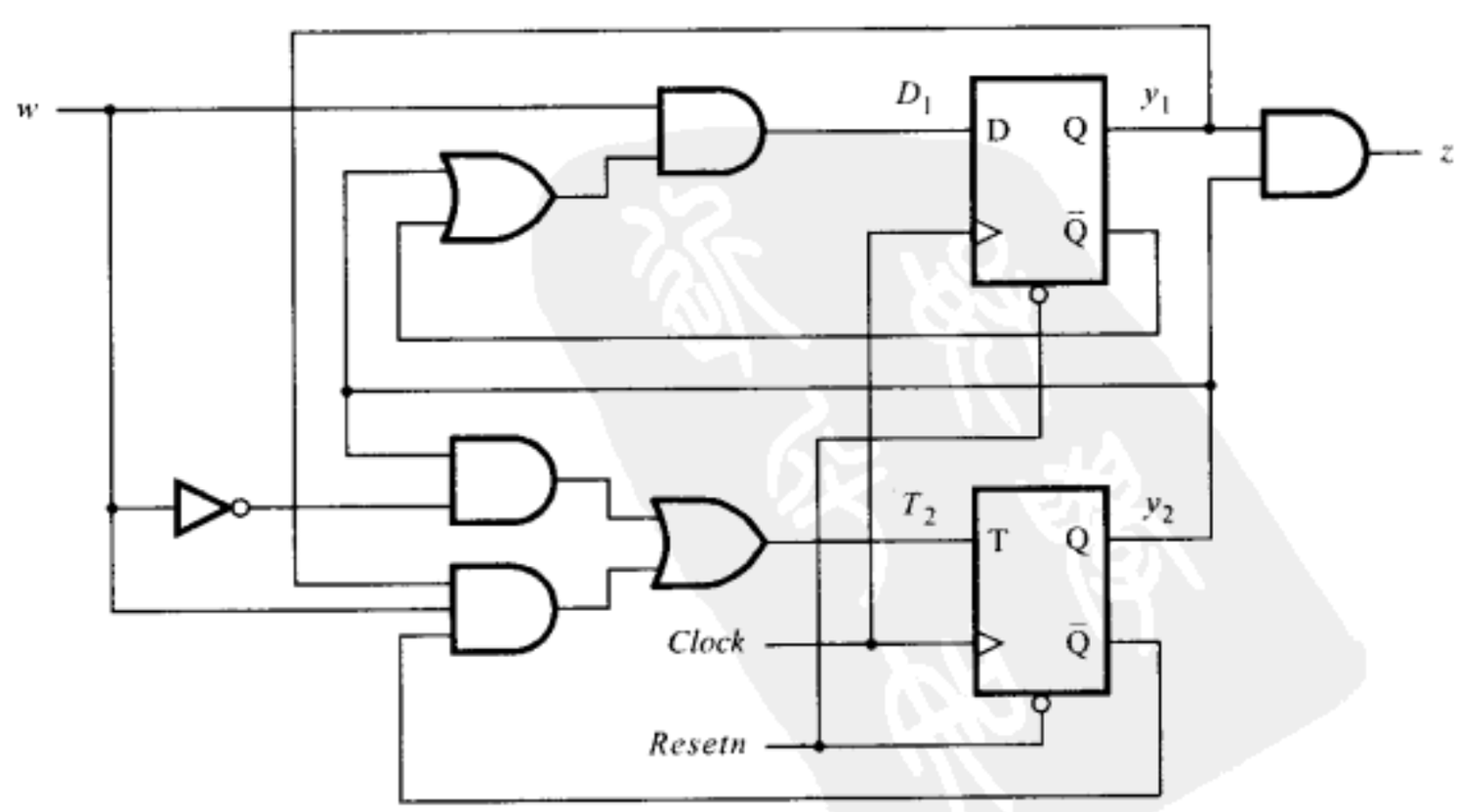


图8-82 例8.11的电路

当前状态 y_2y_1	触发器输入 $w=0$ T_2D_1	触发器输入 $w=1$ T_2D_1	输出 z
00	00	01	0
01	00	10	0
10	10	01	0
11	10	01	1

图8-83 图8-82的电路的激励表

8.10 算法状态机图

本章所用的状态图和状态表用于描述只有几个输入和输出的有限状态机是很方便的。较大型的状态机通常使用另一种不同的表示方法，即所谓的*算法状态机*（ASM）图。ASM图是一种用于表示状态转移并产生有限状态机输出的流程图。图8-84所示是表示ASM图的三种基本方框。

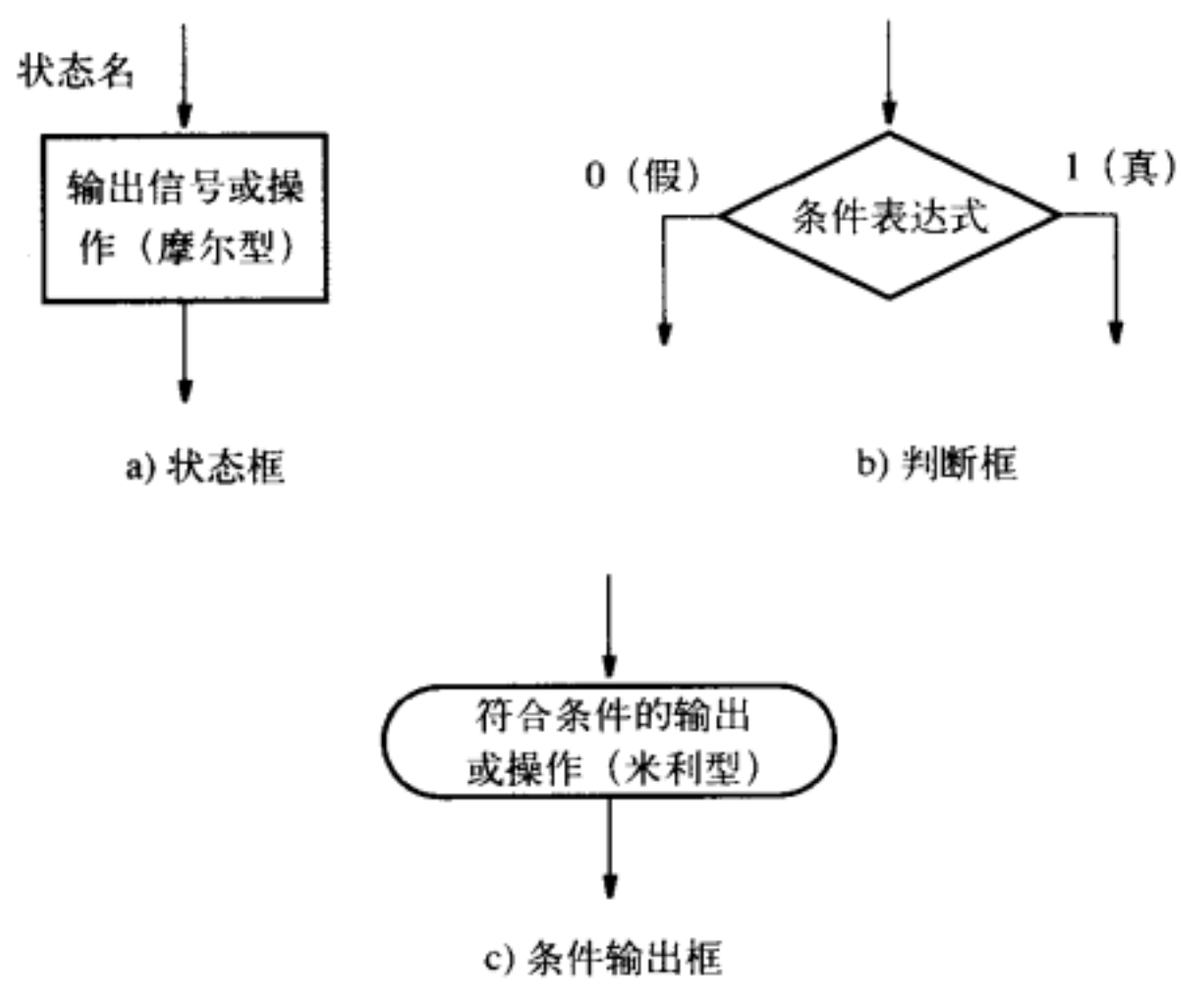


图8-84 算法状态机图所用的元素

- **状态框**是一个矩形的方框，用于表示有限状态机的一个状态。它等价于状态图中的一个节点，或者状态表中的一行。状态名标在方框外面的左上角。摩尔型状态机的输出列在方框内。列在方框内的输出只是那些仅取决于（定义该状态的）状态变量值的输出；我们简单地将它们称为*摩尔输出*。通常只需写出必须有效的信号名，例如只写z就表示输出z必须等于1，因而不必再写$z = 1$。方框内也标出必须采取的操作，例如，$Count \leftarrow Count + 1$ 说明计数器的内容必须增加1。当然，这只是一种简单的方法，用于说明使计数器增加的控制信号必须变成有效。我们将在第10章中用这种方法来描述较大系统的行为。
- **判断框**是一个菱形的方框，用于判断表达式是否成立。根据判断所得的结果，进入不同的路径。条件表达式由一个或者多个输入到有限状态机（FSM）的信号组成。例如，w表示判断是由输入w的值所决定的，而$w_1 \cdot w_2$ 表示：若$w_1 = w_2 = 1$，则进入为真的分支路径，否则进入为假的分支路径。

- **条件输出框**是一个椭圆型方框，用来表示输出的信号是米利型的。米利型输出取决于状态变量的值以及FSM的输入信号值。我们将把这一类型的输出简称为*米利输出*。而确定是否产生输出的条件在菱形的判断框内指定。

图8-85所示是算法状态机（ASM）图，该图表示的是图8-3所示的有限状态机。状态框之间的转移是根据输入变量w值的测试结果而确定的。在每种情况下，若$w = 0$，则从判断框转移到状态A。若$w = 1$，则从状态A转移到B，或者从B转移到C。在状态C中若$w = 1$，则FSM仍旧保持该状态。该图指定了一个摩尔输出z，只有在C状态时，$z = 1$，正如状态框所表明的那样。在状态A和B，z的值是0（无效），状态框内空白隐含表示$z = 0$。

图8-86提供了米利（状态机）输出的例子，该图表示了图8-23所示的FSM。若状态机的状态为B，且$w = 1$，则输出$z = 1$。条件输出框中的z表明了这一点。在所有其他状态，不必写$z = 0$，因为这是隐含的。当$w = 0$时，在状态B，以及当$w = 0$或1时在状态A中，不写z，就意味着$z = 0$。

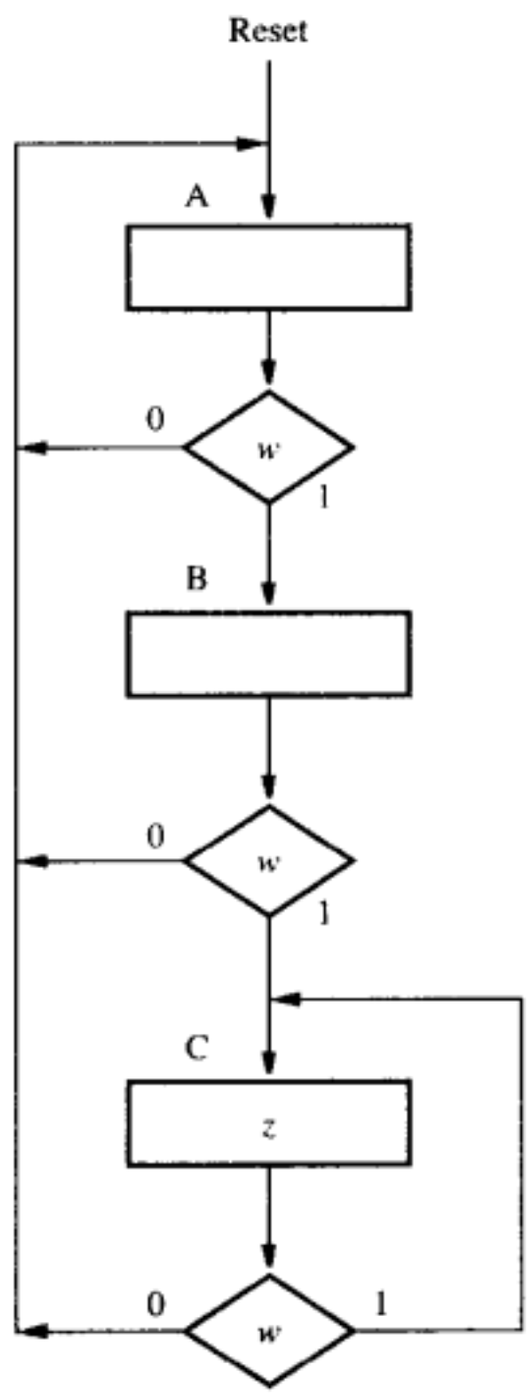


图8-85 图8-3所示FSM的算法状态机图

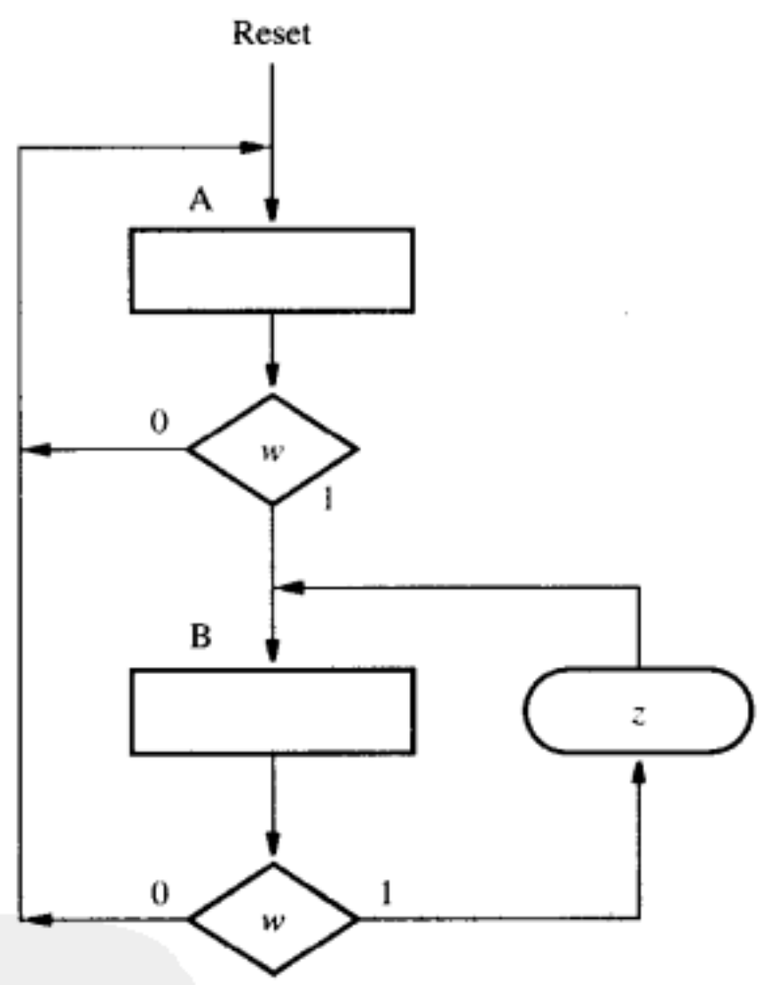


图8-86 图8-23所示FSM的算法状态机图

图8-87 画出了图8-73所示的仲裁状态机的算法状态机图。画在*Idle*状态框下的判断框指定若$r1 = 1$，则该状态机进入状态$gnt1$。在状态$gnt1$，状态机的输出信号为$g1$。该$gnt1$状态框右边的判断框指定若 $r1 = 1$，则状态机将保持状态$gnt1$，若$r = 0$，则状态机进入状态*Idle*。画在*Idle*状态框下的标有$r2$ 的菱形判断框指定：若$r2 = 1$，则该FSM进入状态$gnt2$。只有在首先检查$r1$的值后，然后跟着对应于 $r1 = 0$的箭头才能到达该菱形判断框。同样，标有$r3$的菱形判断框只有当$r1$和 $r2$两者都为0时，才能到达。因此，该ASM图描述了仲裁器所必需的优先权方案。

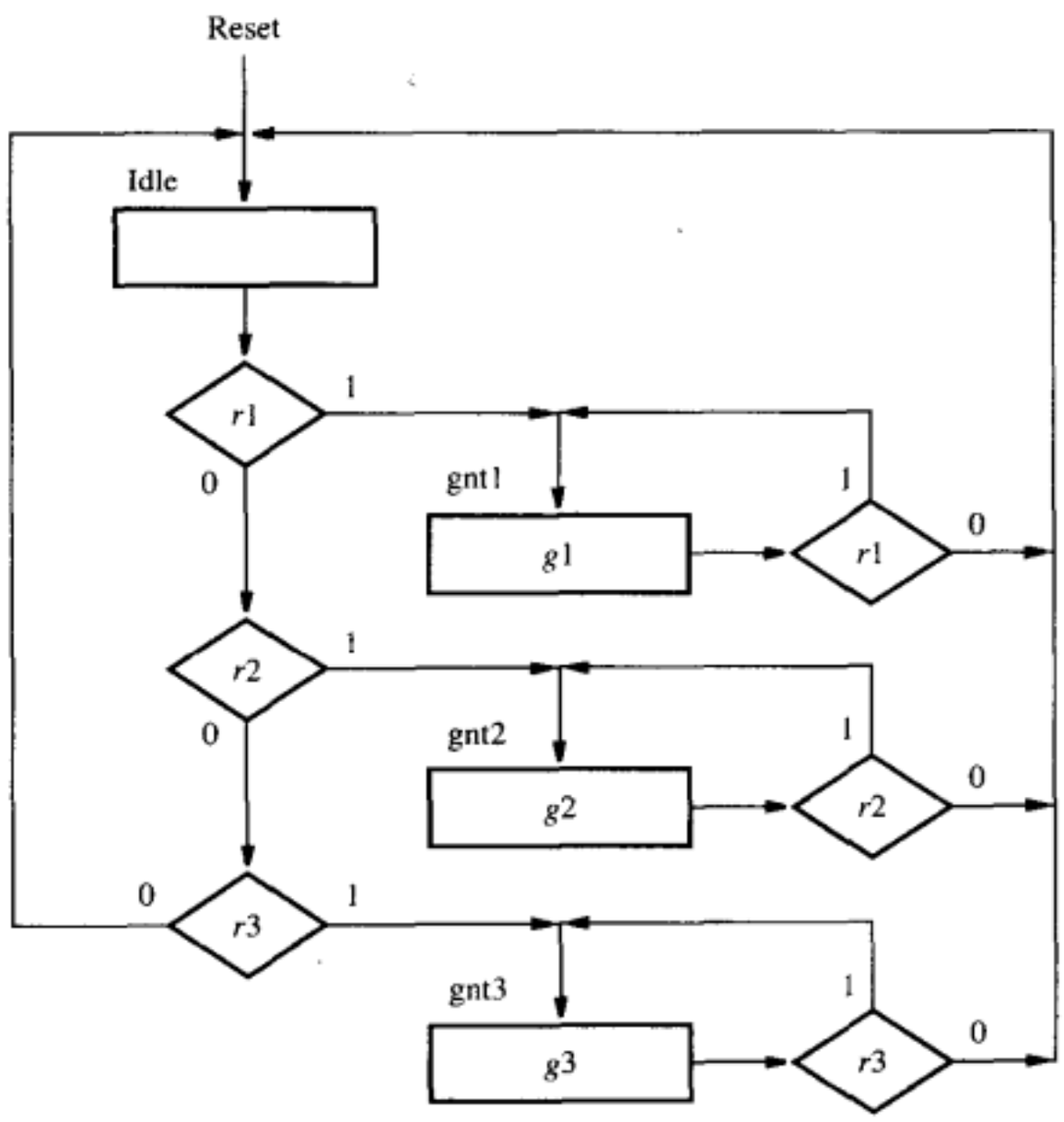


图8-87 图8-73所示仲裁器FSM的算法状态机图

ASM图类似于传统的流程图。与传统的流程图的不同点在于，ASM图包含了时序信息，因为ASM图隐含说明了：该有限状态机只能在每个有效时钟沿时刻才有可能从一个状态改变（转移）到另外一个状态。这里所介绍的ASM图都十分简单，其目的是经由一些判断和条件输出方框的具体例子来介绍ASM图这个术语。ASM图有时候还使用的另一个术语是ASM块，这个术语是指单个的状态框以及该状态框可能连接的任何判断和条件输出框。ASM图可以用来描述复杂的电路，该电路可以包括一个或者几个状态机，以及诸如移位寄存器、计数器、加法器和多路器等其他电路。我们将用ASM图作为设计第10章中较复杂电路的辅助方法。

8.11 时序电路的形式化模型

本章用不太正规的途径介绍了同步时序电路的设计，因为我们认为这样做最能容易地抓住设计此类电路的本质概念。我们也可以用更正规的方式来讨论同一个话题，许多其他书籍采用正规的方式，即把重点放在开关理论方面，而并不重视用CAD工具进行设计的方法和技术。形式化模型通常给出一个简明的指标，很难将该指标与以描述性表示方法为主导的方式相匹配。在本节中，我们将描述一个形式化模型，该模型表示一类通用的时序电路，包括那些同步类型的时序电路。

图8-88表示一个通用的时序电路。该电路的输入为$W = \{w_1, w_2, \cdots, w_n\}$，输出为$Z = \{z_1, z_2, \cdots, z_m\}$，当前状态变量为$y = \{y_1, y_2, \cdots, y_k\}$，下一个状态变量为$Y = \{Y_1, Y_2, \cdots, Y_k\}$。该电路最多可以有$2^k$个状态，即$S = \{S_1, S_2, \cdots, S_{2k}\}$。在状态变量的反馈路径中存在延迟元素，确保在经过一段延迟后，Y的值将被y的值取代。在同步时序电路的场合，延迟元素是触发器，在时钟信号的有效沿时刻，同步时序电路的状态发生改变。因此，延迟取决于时钟周期。时钟周期必须足够长，大于组合电路的传播延迟，此外也要满足触发器的建立和保持参数。

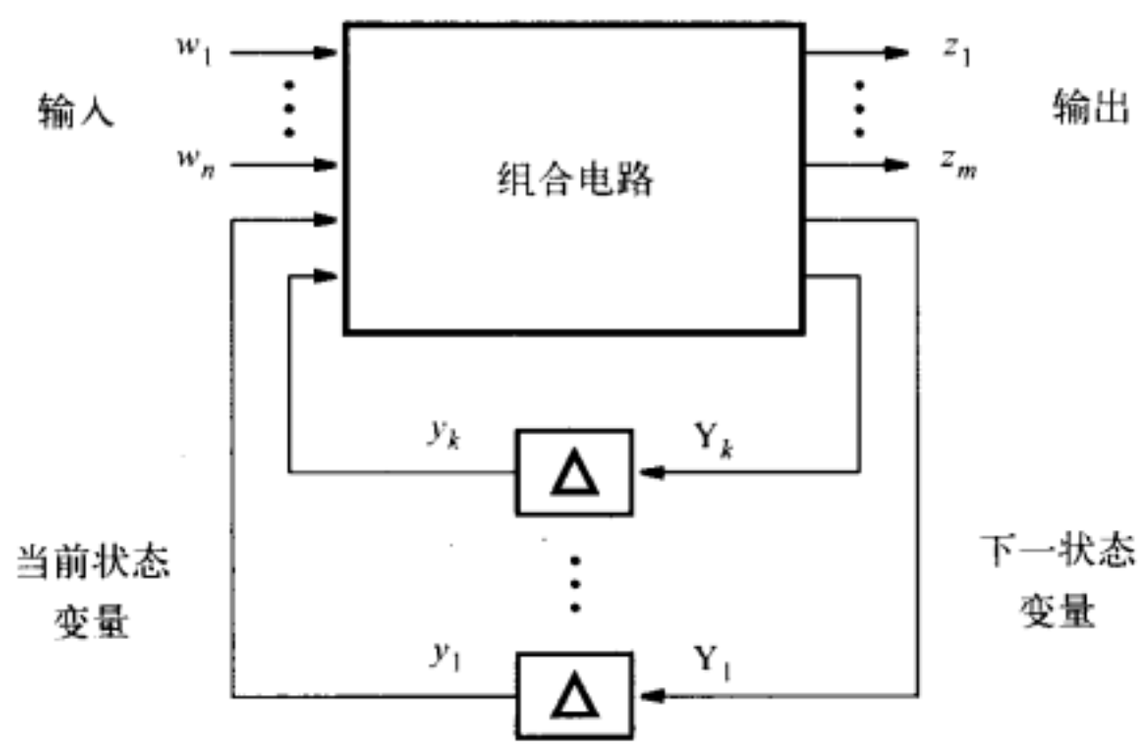


图8-88 时序电路的通用模型

用图8-88所示的模型，同步时序电路M可以从形式上定义为由5部分组成。

$$M = (W, Z, S, \varphi, \lambda)$$

上式中：

- W、Z和S分别为有限的非空的输入、输出和状态的集合。
- φ 是状态转移函数，例如 $S(t+1) = \varphi[W(t), S(t)]$。
- λ是输出函数，例如对摩尔模型而言，$\lambda(t) = \lambda[S(t)]$；对米利模型而言，$\lambda(t)=\lambda[W(t), S(t)]$。

该定义假定t 与 $t+1$之间的时间为1个时钟周期。

在下一章中我们将会看到，延迟并非一定需要受到时钟的控制。在异步时序电路中延迟只是由于各种门的传播延迟而引起的。

8.12 小结

由于时序电路存在着延迟以及闭路的循环，从而使该电路的行为具有特色，在不同的输入条件下，电路的状态可以在一个集合内变化。当前的输入值并非是决定电路行为的唯一因素，若输入取值确定，则当电路所处的状态不同时，电路的行为将是不同的。

经由时序电路的传播延迟必须考虑在内，本章所介绍的设计技术是建立在以下假设的基础之上的，即电路的所有变化都必须由时钟信号的有效沿触发。这样的电路之所以能正确地运行，必须确保当时钟信号的有效沿到来时刻，所有的内部信号都已经稳定了。因此时钟周期必须大于电路中最长的传播延迟。

在实际设计工作中，同步时序电路得到了广泛的应用，常用的CAD工具都支持同步时序电路的设计。所有关于逻辑电路设计的教科书都花费相当大的篇幅讲解同步时序电路的设计。参考文献[1-14]列出了几本比较著名的书籍。

在下一章中，我们将介绍另一类不同的时序电路，这种时序电路不使用触发器来表示电路的状态，也不用时钟脉冲来触发状态的变化。

8.13 问题求解举例

本节列举了读者可能遇到的某些典型问题，并且展示了这些问题是如何求解的。

例8.12 设计一个有限状态机。该状态机具有一个输入w和一个输出z，它是一个序列检测器，能检测出前两个输入的w值为00或者11，此时该状态机应产生输出$z = 1$，否则产生输出$z = 0$。

解：在8.1节我们曾介绍过能检测出连续出现1的序列检测器。用同样的途径，利用图8-89的状态图，我们也可以写出解决上述问题的有限状态机。状态C表示发现两个连续的0或两个以上连续的0，状态E表示发现两个连续的1或者两个以上连续的1。相应的状态表如图8-90所示。

可以用8.6节介绍的分割最小化过程减少状态的个数，得到下面这些分割表达式：

$$P_1 = (ABCDE)$$
$$P_2 = (ABD)\ (CE)$$
$$P_3 = (A)\ (B)\ (C)\ (D)\ (E)$$

· 因为所有五个状态都是需要的，所以我们不得不用三个触发器。

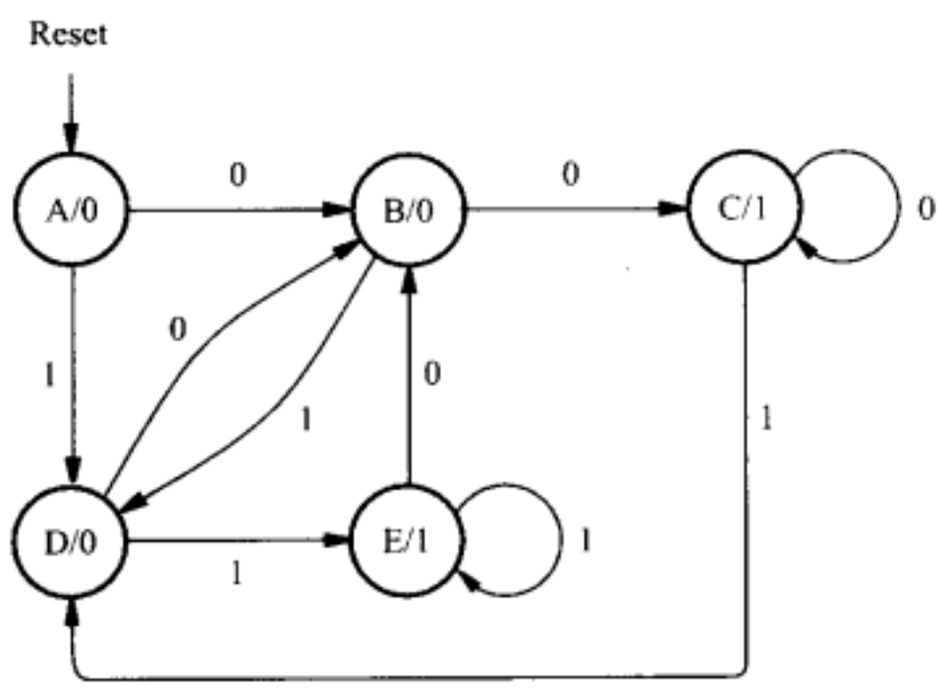


图8-89 例8.12中的状态图

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	B	D	0
B	C	D	0
C	C	D	1
D	B	E	0
E	B	E	1

图8-90 图8-89中有限状态机的状态表

直接做状态分配就可以得到如图8-91所示的状态分配表。状态码 $y_3y_2y_1$ = 101，110，111可以被当作无关条件。接着可以得到下一个状态的表达式：

$$Y_1 = w\bar{y}_1\bar{y}_3 + w\bar{y}_2\bar{y}_3 + \bar{w}y_1y_2 + \overline{wy_1}\,\bar{y}_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2 + w\bar{y}_2\bar{y}_3$$
$$Y_3 = wy_3 + wy_1y_2$$

输出表达式为

$$z = y_3 + \bar{y}_1y_2$$

这些表达式似乎不必要地复杂，看来我们还是应该找到一个更好的状态分配。观察到只有当状态机被复位输入信号（*Reset*）复位后才能到达状态A，所以在为状态B，C，D和E分配的四个码中都令 y_3 = 1 。于是得到如图8-92所示的状态分配表。

	当前状态 $y_3y_2y_1$	下一状态 $w=0$ $Y_3Y_2Y_1$	下一状态 $w=1$ $Y_3Y_2Y_1$	输出 z
A	000	001	011	0
B	001	010	011	0
C	010	010	011	1
D	011	001	100	0
E	100	001	100	1

图8-91 图8-90中有限状态机的状态分配表

	当前状态 $y_3y_2y_1$	下一状态 $w=0$ $Y_3Y_2Y_1$	下一状态 $w=1$ $Y_3Y_2Y_1$	输出 z
A	000	100	110	0
B	100	101	110	0
C	101	101	110	1
D	110	100	111	0
E	111	100	111	1

图8-92 图8-90中有限状态机改进的状态分配表

根据图8-92所示的状态分配表，可得到表示下一个状态和输出的表达式：

$$Y_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$Y_2 = w$$
$$Y_3 = 1$$
$$z = y_1$$

这显然是一个好得多的解决方案。

例8.13 用两个有限状态机实现例8.12所示的序列检测器。其中一个有限状态机能检测出连续出现1的情况，而另外一个有限状态机能检测出连续出现0的情况。

解：图8-16和图8-17给出了一个能很好地检测出连续出现1的情况的有限状态机。该状态机的下一个状态和输出表达式为

$$Y_1 = w$$
$$Y_2 = wy_1$$
$$z_{ones} = y_2$$

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z_{zeros}
D	E	D	0
E	F	D	0
F	F	D	1

a) 状态表

	当前状态 y_4y_3	下一状态 $w=0$ Y_4Y_3	下一状态 $w=1$ Y_4Y_3	输出 z_{zeros}
D	00	01	00	0
E	01	11	00	0
F	11	11	00	1
	10	*dd*	*dd*	*d*

b) 状态分配表

图8-93 检测两个零序列的有限状态机

图8-93定义了一个能检测出连续出现0的情况的有限状态机。该状态机的表达式为

$$Y_3 = \overline{w}$$
$$Y_4 = \overline{w}y_3$$
$$z_{zeros} = y_4$$

组合后电路的输出为

$$z = z_{ones} + z_{zeros}$$

例8.14 推导出一个米利（Mealy）型有限状态机，使它的功能与例8.12描述的序列检测器完全一致。

解：图8-94画出了想要的有限状态机的状态图。图8-95画出了其对应的状态表。实现该有限状态机需要两个触发器。图8-96给出了状态分配表，由此得到下一个状态和输出的表达式如下：

$$Y_1 = 1$$
$$Y_2 = w$$
$$z = \overline{w}y_1\overline{y}_2 + wy_2$$

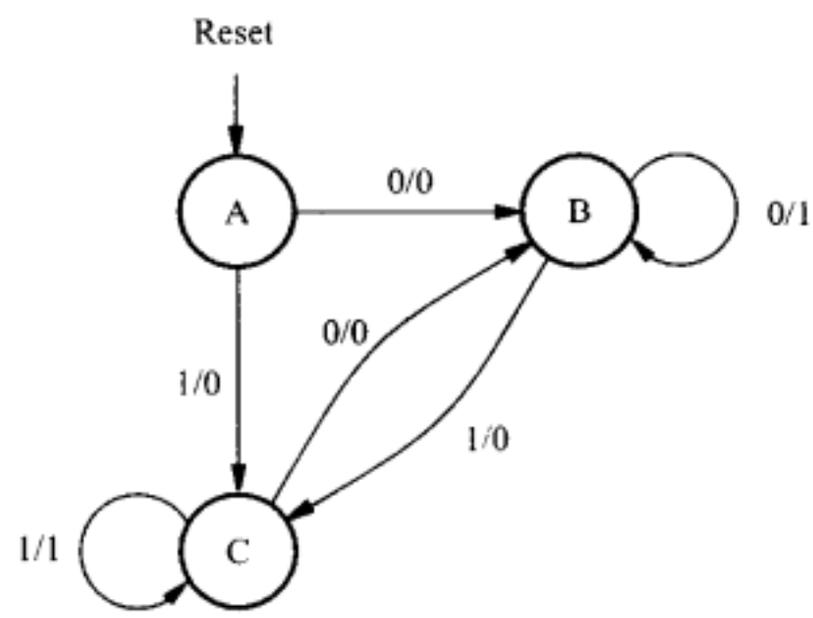


图8-94 例8.14的状态图

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z $w=0$	输出 z $w=1$
A	B	C	0	0
B	B	C	1	0
C	B	C	0	1

图8-95 图8-94中有限状态机的状态表

	当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 $w=0$	输出 $w=1$
	y_2y_1	Y_2Y_1	Y_2Y_1	z	z
A	00	01	11	0	0
B	01	01	11	1	0
C	11	01	11	0	1

图8-96 图8-95中有限状态机的状态分配表

	当前状态 $y_3y_2y_1$	触发器输入 $w=0$ $Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$w=1$ $Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	输出 z
A	000	100	1d	0d	0d	110	1d	1d	0d	0
B	100	101	d0	0d	1d	110	d0	1d	0d	0
C	101	101	d0	0d	d0	110	d0	1d	d1	1
D	110	100	d0	d1	0d	111	d0	d0	1d	0
E	111	100	d0	d1	d1	111	d0	d0	d0	1

图8-97 图8-92中有限状态机用JK触发器的激励表

例8.15 用JK触发器实现图8-92所示的有限状态机。

解：图8-97 画出了该状态机的激励表，由此可推导出该状态机的下一个状态和输出表达式：

$$J_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$K_1 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$J_2 = w$$
$$K_2 = \overline{w}$$
$$J_3 = 1$$
$$K_3 = 0$$
$$z = y_1$$

例8.16 写出实现图8-89所示有限状态机的Verilog代码。

解：采用图8-29所示风格的代码，便能实现所要求的有限状态机。该状态机的代码如图8-98所示。

```
module sequence (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [3:1] y, Y;
   parameter [3:1] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)   Y = D;
            else     Y = B;
         B: if (w)   Y = D;
            else     Y = C;
         C: if (w)   Y = D;
            else     Y = C;
         D: if (w)   Y = E;
            else     Y = B;
         E: if (w)   Y = E;
            else     Y = B;
         default:    Y = 3'bxxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define output
   assign z = (y == C) | (y == E);

endmodule
```

图8-98 图8-89中有限状态机的Verilog代码

例8.17 写出实现图8-94所示有限状态机的Verilog代码。

解：采用图8-36所示风格的代码，便能实现所要求的米利型有限状态机。该状态机的代码如图8-99所示。

```
module seqmealy (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b11;

   // Define the next state and output combinational circuits
   always @(w, y)
      case (y)
         A:  if (w)
             begin
                z = 0; Y = C;
             end
             else
             begin
                z = 0; Y = B;
             end
         B:  if (w)
             begin
```

图8-99 图8-94中有限状态机的Verilog代码

```
            z = 0; Y = C;
        end
        else
        begin
            z = 1; Y = B;
        end
    C:  if (w)
        begin
            z = 1; Y = C;
        end
        else
        begin
            z = 0; Y = B;
        end
    default:
        begin
            z = 0; Y = 2'bxx;
        end
    endcase

  // Define the sequential block
  always @(negedge Resetn, posedge Clock)
    if (Resetn == 0)  y <= A;
    else  y <= Y;

endmodule
```

图8-99 （续）

例8.18 在计算机系统中，经常要用到数据的串行发送，即每次只发送一位信号，以节省互连电缆的成本。这是指发送端的并行数据必须转换成串行一位一位地发送，而接收端收到后必须将这些位转换为并行的形式。假设我们想用这种方式发送ASCII字符。正如5.8节所讲解的那样，标准的ASCII码用七位二进制码来定义每个字符。通常，一个字符占用一个字节，此时，字节的第八位可以被设置为1或者0，用作其他位的奇偶校验，以保证可靠的传输。

并行到串行的传输可以借助于移位寄存器来完成。假设某个电路接收到表示ASCII字符的并行数据$B = b_7, b_6, \cdots, b_0$，还假设$b_7$被设置成0。假设该电路生成一个奇偶校验位$p$，在串行发送字节时最后发送的位是$p$而不是$b_7$。图8-100给出了一个可能的电路。用一个有限状态机来产生奇偶校验位，借助于多路选择器将这个校验位包含到输出位流中发送出去。用一个三位的计数器来确定当计数器计到7时发送奇偶校验位p 。请设计这样一个有限状态机。

解：随着移位寄存器的移位，待发送的位被逐位移出，有限状态机检查所有的位，并且记录已经发出的1的个数是奇数还是偶数。若已发出的7位中1的个数是奇数，则将p设置为1（对偶校验而言——译者注）。因此，该有限状态机必须具有两个状态。图8-101所示的是状态表、状态分配表和最后实现的电路。下一个状态的表达式是

$$Y = \overline{w}y + w\overline{y}$$

输出p正好等于y。

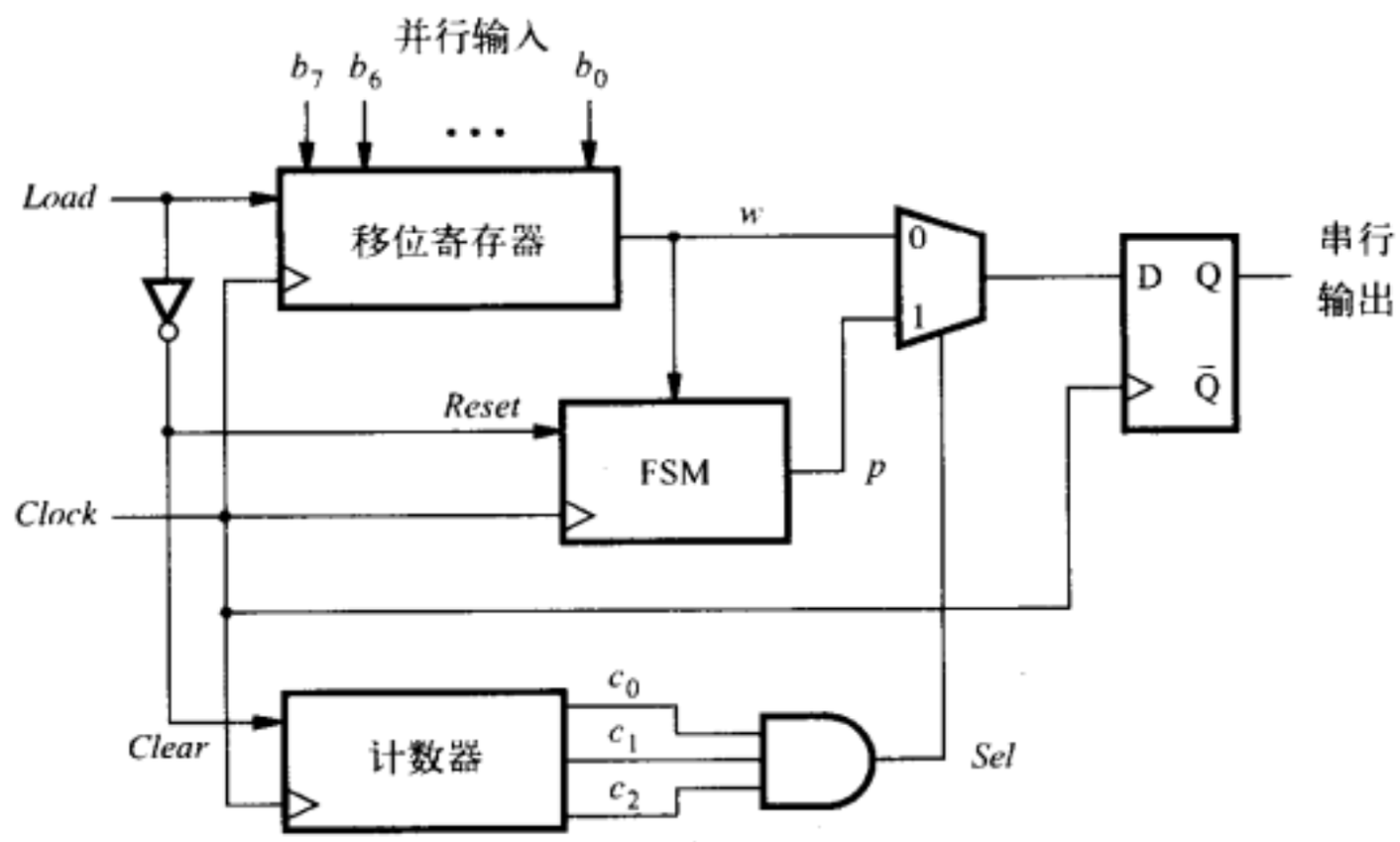


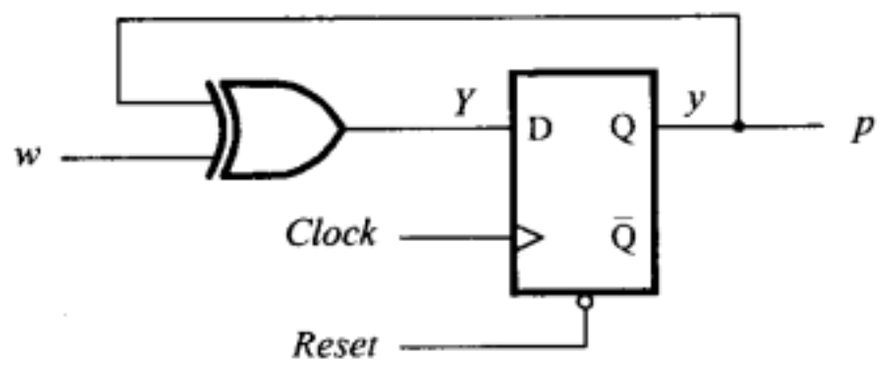
图8-100 并行到串行转换器

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 p
S_{even}	S_{even}	S_{odd}	0
S_{odd}	S_{odd}	S_{even}	1

a) 状态表

当前状态 y	下一状态 $w=0$ Y	下一状态 $w=1$ Y	输出 p
0	0	1	0
1	1	0	1

b) 状态分配表



c) 电路

图8-101 产生奇偶校验位的有限状态机

练习题

标有*号的习题答案参见书后。

***8.1** 用D触发器实现由图P8-1的状态分配表定义的有限状态机电路。

当前状态 y_2y_1	下一状态 $w=0$ Y_2Y_1	下一状态 $w=1$ Y_2Y_1	输出 z
00	10	11	0
01	01	00	0
10	11	00	0
11	10	01	1

图P8-1 练习题8.1和8.2的状态分配表

***8.2** 用JK触发器实现由图P8-1的状态分配表定义的有限状态机电路。

8.3 推导输入为w输出为z的有限状态机的状态图。当w为1001或者1111时，该状态机必须产生$z = 1$，

否则$z = 0$。允许输入码型的重叠，该电路期望行为的举例如下：

$$w: 0101111001100111 11$$
$$z: 00000010010001 0011$$

8.4 写出习题8.3描述的有限状态机的Verilog代码。

***8.5** 推导出一个单输入单输出摩尔型有限状态机的最小状态表。该FSM可在输入序列中检测到110或者101码型，产生输出1。该FSM应该能检测出重叠的码型。

***8.6** 用米利型有限状态机，重做一遍习题8.5。

8.7 推导能实现图8-51和图8-52所示状态表的电路。状态最小化对电路实现成本的影响是什么？

8.8 推导能实现图8-55和图8-56所示状态表的电路。比较两个电路的实现成本。

8.9 某时序电路有两个输入w_1和w_2，一个输出z。其功能是将输入的两个序列进行比较。若在四个连续的时钟周期里$w_1 = w_2$，则电路输出$z = 1$，否则$z = 0$。举例说明如下：

$$w_1: 0110111000110$$
$$w_2: 1110101000111$$
$$z: 0000100001110$$

推导出该电路。

8.10 写出习题8.9所描述的有限状态机的Verilog代码。

8.11 某FSM有一个输入w，一个输出z。在四个连续的时钟脉冲期间，输入了四个w值的序列。该FSM可以检测出是否施加了$w: 0010$或者$w: 1110$的序列，若施加了，则$z = 1$，否则$z = 0$。经过四个连续的时钟脉冲后，该FSM复位，为下一个序列做准备。推导出该FSM的状态表，并对所需要的状态数进行最小化。

***8.12** 写出担当三位奇偶校验位发生器的有限状态机的最小状态表，在三个连续时钟周期内，在输入w的每个三位序列中若观察到1的个数为奇数，则FSM产生奇偶校验位$p = 1$。

8.13 写出习题8.12所描述的有限状态机的Verilog代码。

8.14 画出图8-43和图8-47所示电路的时序图，假设这两个电路的输入信号a和b的变化相同，请计算传播延迟。

***8.15** 画出图P8-1所示状态分配表的状态表，用A，B，C，D表示状态表中的四行。假设新的状态分配表使用独热编码。A状态用$y_4 y_3 y_2 y_1 = 0001$表示，B，C和D状态分别用0010，0100和1000表示。把该状态表综合成由D触发器构成的电路。

8.16 修改习题8.15画出的电路，用$y_4 y_3 y_2 y_1 = 0000$来表示复位状态A，其余三个状态（B, C, D）的编码根据需要做必要的修改。（提示：你不必重新综合电路。）

***8.17** 在图8-59中，假设状态B和G中未指定的输出分别为0和1，写出该有限状态机的最小状态表。

8.18 在图8-59中，假设状态B和G中未指定的输出分别为1和0，写出该有限状态机的最小状态表。

8.19 画出实现由图8-57和图8-58定义的有限状态机的电路。你是否能由此得到关于用摩尔型或者米利型状态机实现电路的复杂性的任何结论？

8.20 用D触发器设计一个可以对信号线w上的脉冲个数进行计数的计数器。显示序列0, 2, 1, 3, 0, 2, …的计数值。

***8.21** 用JK触发器重新做一遍习题8.20。

***8.22** 用T触发器重新做一遍习题8.20。

8.23 用D触发器设计一个模6计数器，该计数器的计数序列为：0, 1, 2, 3, 4, 5, 0, 1, …。若计数器的使能输入$w = 1$，则该计数器计算时钟的脉冲个数。

8.24 用JK触发器重新做一遍习题8.23。

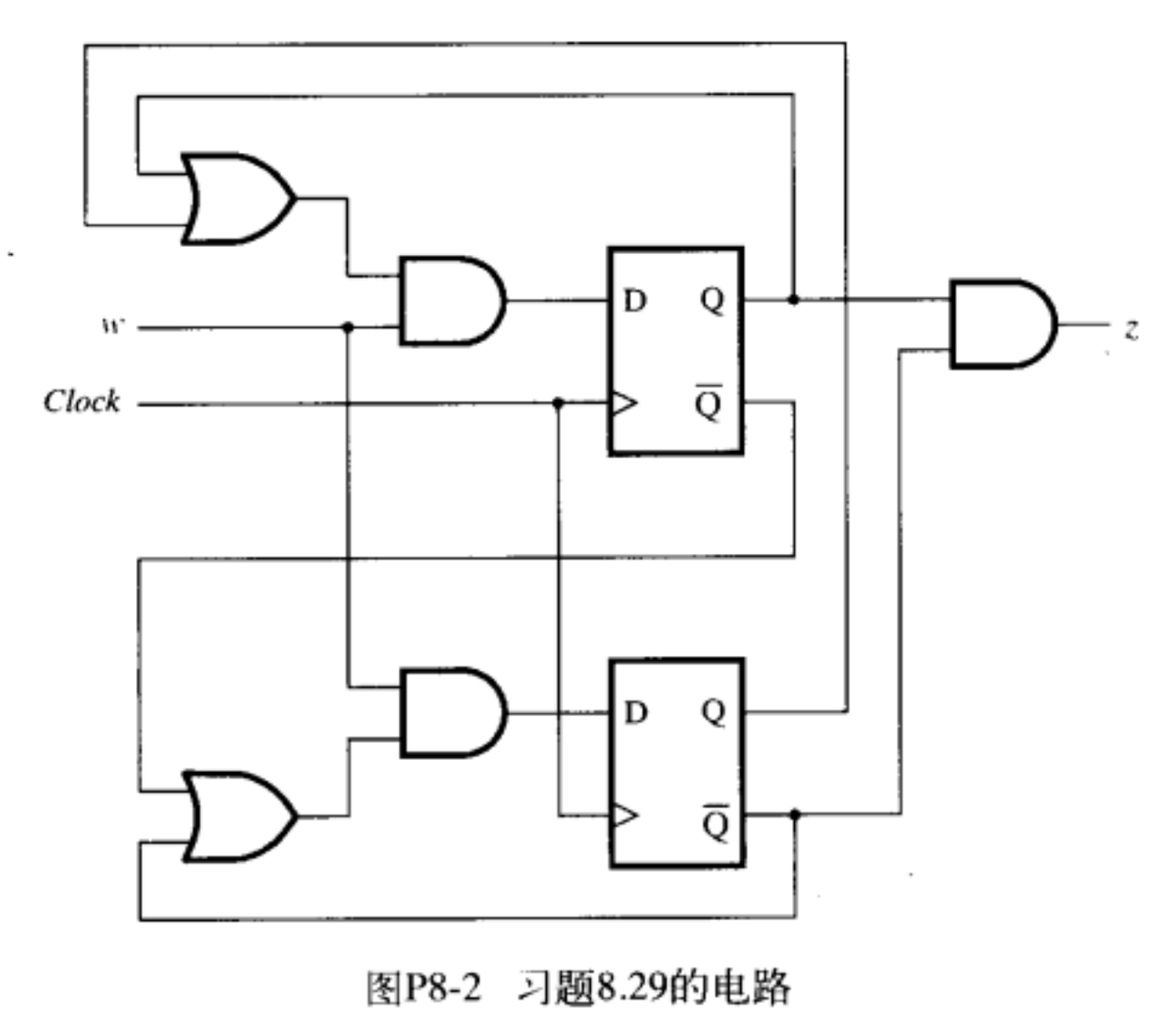
8.25 用T触发器重新做一遍习题8.23。

8.26 用D触发器设计一个类似三位计数器的电路，该计数器由输入信号w控制。若$w=1$，则该计数器的内容加2，若计数器计到8或者9，则重新从0开始计数。因此若当前的状态为8或者9，则下一个状态便分别变成0或者1。若$w=0$，则计数器的内容减1，其动作与一般的递减计数器一致。

8.27 用JK触发器重新做一遍习题8.26。

8.28 用T触发器重新做一遍习题8.26。

* **8.29** 画出图P8-2所示电路的状态表。该电路能检测信号线w上输入的什么序列？



图P8-2 习题8.29的电路

8.30 用图8-29所示的代码风格，编写图8-57所示的有限状态机的Verilog代码。

8.31 用图8-34所示的代码风格，重新做一遍习题8.30。

8.32 用图8-29所示的代码风格，编写图8-58所示的有限状态机的Verilog代码。

8.33 用图8-34所示的代码风格，重新做一遍习题8.32。

8.34 编写图P8-1所示的有限状态机的Verilog代码。

8.35 用ASM图的形式，表示图8-57所示的有限状态机。

8.36 用ASM图的形式，表示图8-58所示的有限状态机。

8.37 若器件1和器件2不断地提出请求，8.8节所定义的仲裁器有限状态机（见图8-72）有可能使得器件3永远得不到服务的机会，即使在Idle（空闲）状态，不是器件1，便是器件2总会提出请求。修改这个有限状态机，确保器件3能得到服务，使得器件3若提出请求，器件1和2只能得到一次服务，然后器件3的服务请求便能得到认可。

8.38 编写习题8.37所设计的有限状态机的Verilog代码。

8.39 考虑一个比例8.1介绍的更一般的任务。假设在处理器的总线上连接着四个n位的寄存器。若使控制信号R_{out}有效，寄存器R的内容就可以放置到总线上 。若控制信号R_{in}有效，在时钟信号的有效沿，总线上的数据就可以被加载到寄存器中。假设三个寄存器$R1$，$R2$和$R3$都被用做一般的寄存器，而第四个寄存器$TEMP$被用做特定场合下的暂存寄存器。

我们想要实现R_i 和R_j 所存储数据的交换操作。完成这个操作的步骤如下（每个动作需要一个时钟周期）：

$$\text{TEMP} \leftarrow [\text{Rj}]$$
$$\text{Rj} \leftarrow [\text{Ri}]$$
$$\text{Ri} \leftarrow [\text{TEMP}]$$

两个输入信号w_1和w_2被用来指明两个寄存器必须按照如下次序互换：

若$w_2w_1 = 01$，则$R1$和$R2$的内容互换；
若$w_2w_1 = 10$，则$R1$和$R3$的内容互换；
若$w_2w_1 = 11$，则$R2$和$R3$的内容互换。

规定交换的输入的取值存在三个时钟周期，设计一个电路，能产生所需要的控制信号：$R1_{out}$，$R1_{in}$，$R2_{out}$，$R2_{in}$，$R3_{out}$，$R3_{in}$，$TEMP_{out}$和$TEMP_{in}$。推导出该电路的下一个状态的表达式和输出表达式，并使电路的成本最低。

8.40 观察图8-100所示的电路，写出定义该电路的Verilog代码。

8.41 请根据8.5节讲述的串行加法器电路，设计出可计算操作数A和B之间差的串行减法器电路。（提示：利用5.3.1节介绍的求2的补码的原理，生成B的2的补码。）

8.42 在练习题8.41中我们设计了串行减法器，请写出定义该串行减法器的Verilog代码。

参考文献

1. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
2. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
3. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
4. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
5. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, (PWS Publishing Co.: 1997).
6. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
7. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
8. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
9. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., (Wiley: New York, 1993).
10. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. T. L. Booth, *Digital Networks and Computer Systems*, (Wiley: New York, 1971).
13. Z. Kohavi, *Switching and Finite Automata Theory*, (McGraw-Hill: New York, 1970).
14. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, (Prentice-Hall: Englewood Cliffs, NJ, 1966).

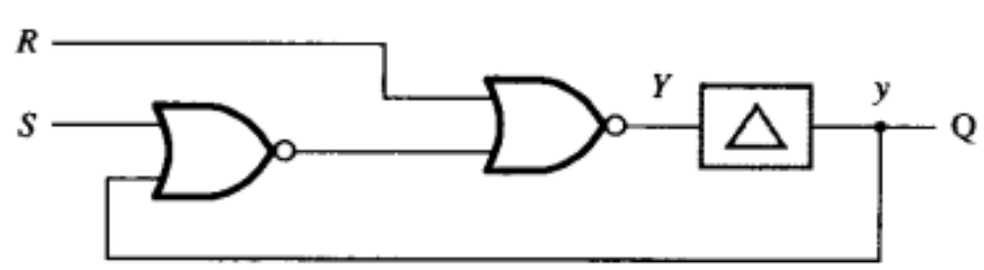
第9章　异步时序电路

上一章中我们介绍了同步时序电路的设计，在同步时序电路中，表示状态变量的触发器均工作在同一时钟信号下。该时钟信号由周期性的脉冲信号组成。状态的转移只能发生在时钟脉冲的正跳变沿或负跳变沿。因此，同步时序电路被称为工作于*脉冲模式*。本章介绍的是：既不工作于脉冲模式也不使用触发器来表示状态变量的时序电路。这种电路被称为*异步时序电路*。

在异步时序电路中，状态的转移并不是由时钟脉冲触发的，而是由任一给定时刻电路的每个输入的逻辑电平（1或0）来决定的。为了保证可靠的操作，电路的输入必须一次只能有一个信号发生变化。此外，两个输入信号的变化之间必须有足够的时间间隔，以保证电路能达到*稳定状态*（即所有的内部信号已不再发生变化）。满足上述约束条件的电路被称为工作于*基础模式*的异步时序电路。

9.1　异步行为

为了引入异步时序电路的概念，我们先回顾一下图7-4中介绍的基本锁存器电路。这是一个SR锁存器，图9-1a中又把它重新画了一遍。其中的反馈回路使得电路的状态得以保持，从而使电路具有时序电路的特性。这是一个异步电路，因为电路输出值Q的改变无需等待同步时钟脉冲的到来。当输入信号S（置位）或R（复位）发生改变的时候，输出Q的值作为输入信号的响应，将在信号通过或非门的一个很短的传播时间之后发生改变。在图9-1a中，经过两个或非门的组合传播延时用标有△符号的方块来表示。此时，或非门的符号就可以代表没有延时的理想的门。在此我们继续使用第8章中的符号，输出Q对应于电路的*当前状态*，用*当前状态变量*y来表示。y的值被反馈回电路的输入端，并由电路产生下一个*状态变量*的值，在此我们用Y来表示电路的*下一状态*。如图所示，在经过Δ延时后，y被赋值为Y的值。至此我们可以看到，画出的逻辑电路图与图8-88中时序电路的一般模型是一致的。



a) 带门延迟模型标记的电路

当前状态 y	下一状态 $SR=00$	01	10	11
	Y	Y	Y	Y
0	0	0	1	0
1	1	0	1	0

b) 状态分配表

图9-1　SR锁存器的分析

通过分析SR锁存器电路，我们得到如图9-1b所示的状态分配表。当当前状态为$y = 0$且输入为$S = R = 0$时，电路将产生输出$Y = 0$。由于$y = Y$，故电路的状态将不会发生改变。此时，我们称在这样的输入条件下，电路是*稳定的*。现在假定R的值改变为1而S仍旧保持为0，此时电路将产生输出$Y = 0$，且保持稳定。再假定S接下来改变为1而R保持1，由于输出Y的值没有发生改变，电路仍继续保持稳定。接着，令R改变为0，且保持$S = 1$不变，即输入为SR=10，将使得电路产生输出$Y = 1$。因为此时$y \neq Y$，电路不再是稳定的。经过△时间延迟后，电路将

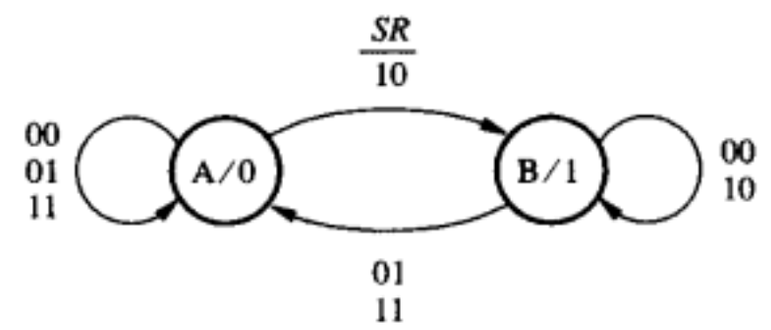
转变为新的当前状态：$y=1$。一旦达到这样的新状态后，只要SR继续保持为10，Y的值将保持为1。电路再次达到稳态。上面的推导过程完全适用于当前状态为$y=1$情况下的分析。

在异步时序电路中，稳态的概念至关重要。对于给定的输入值，若电路达到了某个具体的状态并保持在该状态，则称该状态为稳态。为了明确地指出何种条件下电路才是稳定的，通常的做法是把状态分配表中的稳定状态用圆圈作标记，见图9-1b所示。

通过这个状态分配表，我们可以得到如图9-2a所示的状态表。状态A和B分别表示$y=0$和$y=1$的状态。因为电路的输出Q仅取决于当前状态，所以该电路是摩尔型的有限状态机(Finite State Machine)。表示该有限状态机行为的状态图见图9-2b所示。

当前状态	下一状态				输出
	$SR=00$	01	10	11	Q
A	(A)	(A)	B	(A)	0
B	(B)	A	(B)	A	1

a) 状态表



b) 状态图

图9-2 SR锁存器的有限状态机模型

前面的分析指出，使用与第8章同步时序电路类似的方法，可以将异步时序电路的行为表述为有限状态机。现在考虑进行相反的操作，即给定的是如图9-2a的状态表，我们可以按如下步骤把状态表综合成异步电路：在执行完状态分配之后，我们得到了图9-1b中的状态分配表。该表列出的是一个真值表，输入变量为y，S和R，输出变量为Y。并导出最小的和之积表达式，如下式所示：

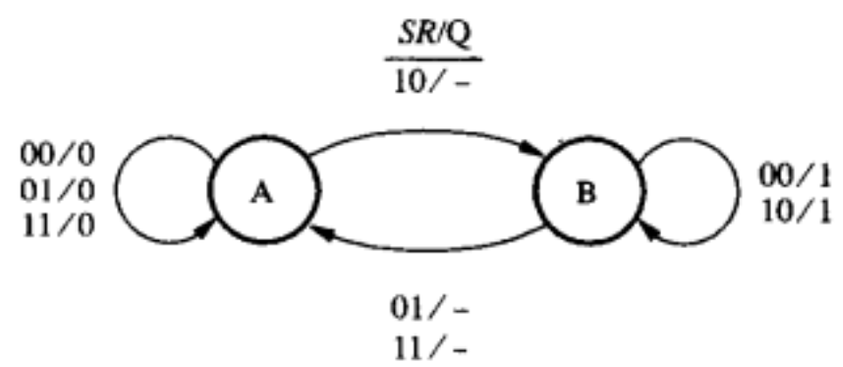
$$Y=\overline{R}\cdot(S+y)$$

如果我们使用第8章中的方法来设计同步时序电路，输出Y会被连接到触发器的D输入端，而且还需要一个时钟信号来控制状态变化发生的时刻。但因为我们综合的是异步电路，所以在反馈回路中并没有插入触发器。相反，我们只使用必要的逻辑门也可把输出信号反馈回来作为当前状态的输入y，实现上述逻辑函数的电路。若用或非门来实现则可得到如图9-1a中所示的电路。这个简单的例子告诉我们：同步时序电路和异步时序电路可以通过类似的技术来进行综合。然而，我们很快就会发现，对于更复杂的异步电路而言，设计任务将十分困难。

为了更深入地探讨异步电路的本质，研究如何用米利模型的形式来表示SR锁存器是一件很有意思的事情。如图9-3所示，当电路处于稳态时，电路的输出与摩尔模型是一样的，即状态A时为0，状态B时为1。现在来考虑电路状态发生转移时会有什么变化。假设当前状态为A，且SR的输入从00转变到10。由状态表可知，有限状态机的下一个状态为B。当电路变为B状态，输出Q将变为1。但在米利模型中，输入的变化应该会立刻影响到输出。因此，当电路仍处于A状态，SR变为10应该导致输出$Q=1$。我们应该在状态表的第一行对应位置中写下1，不过我们并不这么写，而是将它设为不定值（–）。因为只要电路一旦达到B状态，输出Q将会马上变为1，让Q更早一点儿变为1就没有什么意义了。不指定该项的值（译者注：即当$SR=10$时Q的初始值），意味着我们可以任意给它赋0或者1，这样做可以使实现状态表的电路变得更简单些。基于同样的道理，对应从B转到A的两个输出项（译者注：即当$SR=01$或$SR=11$时Q的初始值）也可被置为不定值（–）。

当前状态	下一状态				输出 Q			
	SR = 00	01	10	11	00	01	10	11
A	A	A	B	A	0	0	–	0
B	B	A	B	A	1	–	1	–

a) 状态表



b) 状态图

图9-3 SR锁存器的米利模型表示

利用状态分配A时为$y = 0$，B时为$y = 1$，状态分配表表示了Y和Q的真值表。此时Y的最简表达式与摩尔模型中的一致。为了求得Q的表达式，我们需要将表中未指定值的项设置成0或1。若把状态分配表第一行中未指定值的项规定为0，而第二行中未指定值的项规定为1，则得到简化的布尔方程 $Q = y$，因此生成的电路如图9-1a所示。

术语

在前面的讨论中，我们使用了与以前章节中论及同步时序电路时相同的术语。然而，当处理异步时序电路时，习惯使用的是另外两个不同的术语。我们更习惯于称“*流程表*”（flow table），而不是“状态表”。流程表这个名字更能体现状态流如何随着输入信号的变化而改变。我们更常提到的是“*转移表*”（transition table）或“*激励表*”（excitation table），而不是状态分配表。本章中，我们将会用到流程表和激励表这两个术语。流程表将定义电路状态的变化和必定产生的相应输出，而激励表描述用状态变量表示的电路状态的转移。激励表这个术语来自这样一个事实：状态表中的改变是通过激励下一个状态变量，使其开始转向新状态来实现的。

9.2 异步电路分析

要进一步掌握异步电路，分析几个例子是十分有帮助的。我们先回忆一下图8-88所示的一般模型，若假设反馈回路的延迟代表了电路的传播延迟，则每个门可以看做是理想的零延迟的门电路。

例9.1（门控D锁存器） 在第7章和第8章中，我们使用门控D锁存器作为由同步时钟控制的电路的关键部件。把这个锁存器当作异步电路来加以分析将对我们很有启发，在这个异步电路中时钟只是输入信号之一。首先假定D信号和时钟输入信号不会同时变化，这是合理的，这样就满足了异步电路的基本要求

图9-4a展示了一个门控D锁存器的电路图，该电路图的风格类似于图8-88中的模型。我们曾在图7-8中介绍此电路，并曾在7.3节中讨论过。该电路的下一个状态的表达式为：

$$\begin{aligned} Y &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y) \\ &= CD + \overline{C}y + Dy \end{aligned}$$

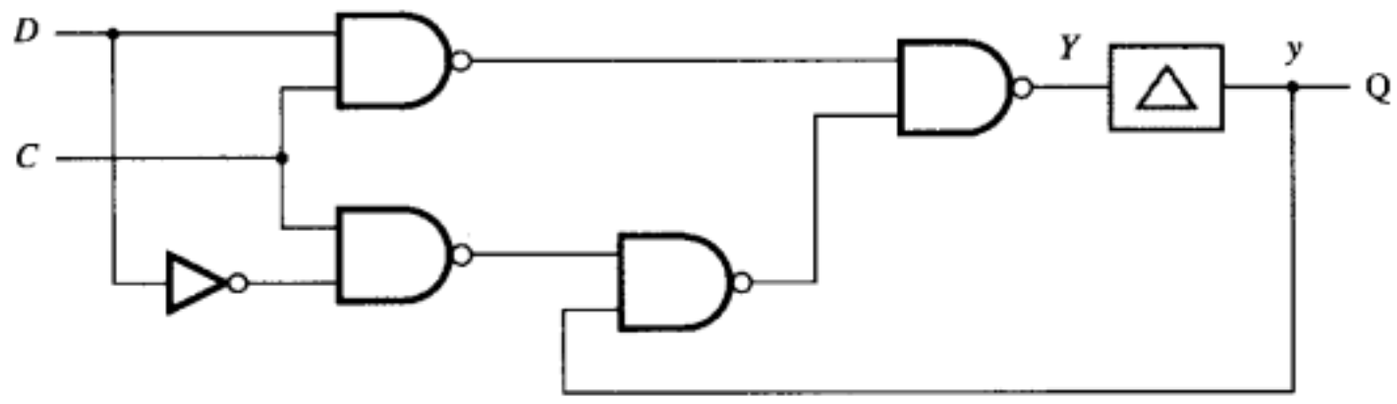
表达式中的Dy项是冗余的，可以将其删除而不会影响Y的逻辑函数。从而得到最简表达式如下：

$$Y = CD + \overline{C}y$$

电路实现中需要冗余项Dy是因为该冗余项可以避免电路中出现*冒险竞争*的情况，我们将在9.6节详细讨论冒险竞争的问题。

根据Y的逻辑表达式，将C、D和y的所有组合值代入，便可以计算的出如图9-4b所示的激励表。我们注意到，电路的状态只有在$C = 1$且D与当前状态y的值不同的情况下才会发生改变，

其他情况下电路保持稳定。用符号A和B来分别代表$y=0$和$y=1$状态，从而得到如图9-4c和d 所示的流程表和状态图。



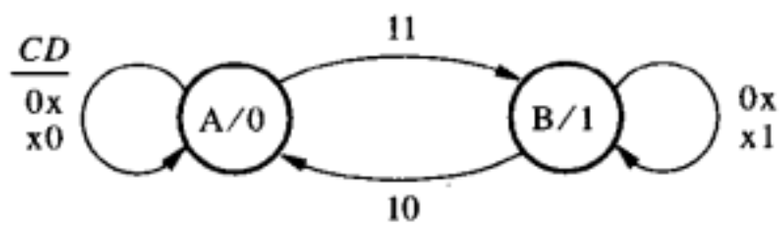
a) 电路

当前状态 y	下一状态 $CD=00$	01	10	11	输出 Q
	Y	Y	Y	Y	
0	0	0	0	1	0
1	1	1	0	1	1

b) 激励表

当前状态	下一状态 $CD=00$	01	10	11	输出 Q
A	A	A	A	B	0
B	B	B	A	B	1

c) 流程表



d) 状态图

图9-4 门控D锁存器电路

例9.2（主从D触发器） 在例9.1中我们将门控D锁存器作为异步电路进行了分析，事实上，所有的实用电路都是异步电路。然而，如果电路的行为由一个时钟信号严格地加以控制，我们就可以像在第8章中那样，使用一些更简单的假定。回想一下，在同步时序电路中，所有的信号值的改变都与时钟信号同步。现在我们将把另一个同步电路当做异步电路来进行分析。

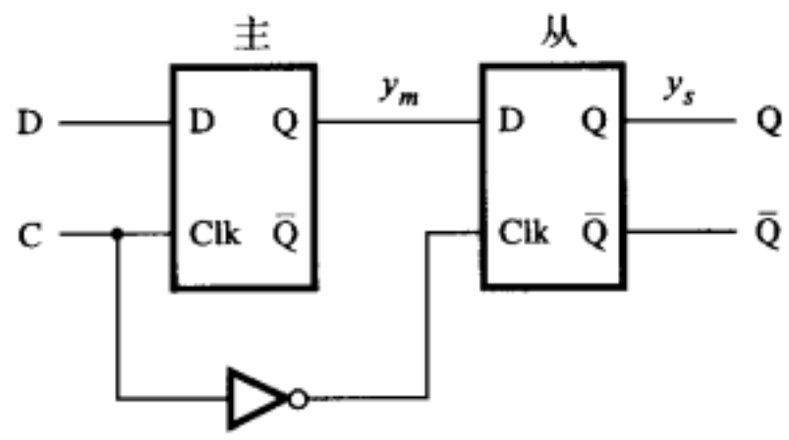


图9-5 主从式D触发器的电路

如图7-10所示，我们用两个门控D锁存器组成了一个主从D触发器。图9-5重新绘制了该电路。我们可以把该电路当做两个门控D锁存器的级联来进行分析。利用从例9.1得到的结论，该电路简化后的下一个状态的逻辑表达式如下：

$$Y_m = CD + \overline{C}y_m$$
$$Y_s = \overline{C}y_m + Cy_s$$

表达式中的下标m和s分别代表触发器的主级和从级。从上述表达式得到如图9-6a所示的激励表。我们将四个状态分别定义为$S1$至$S4$，得到如图9-6b所示的流程表。相应的状态图见图9-7所示。

当前状态 $y_m y_s$	下一状态 $Y_m Y_s$ $CD=00$	01	10	11	输出 Q
00	00	00	00	10	0
01	00	00	01	11	1
10	11	11	00	10	0
11	11	11	01	11	1

a) 激励表

当前状态	下一状态 $CD=00$	01	10	11	输出 Q
S1	S1	S1	S1	S3	0
S2	S1	S1	S2	S4	1
S3	S4	S4	S1	S3	0
S4	S4	S4	S2	S4	1

b) 流程表

当前状态	下一状态 $CD=00$	01	10	11	输出 Q
S1	S1	S1	S1	S3	0
S2	S1	—	S2	S4	1
S3	—	S4	S1	S3	0
S4	S4	S4	S2	S4	1

c) 带不定项的流程表

图9-6 例9.2的激励表和流程表

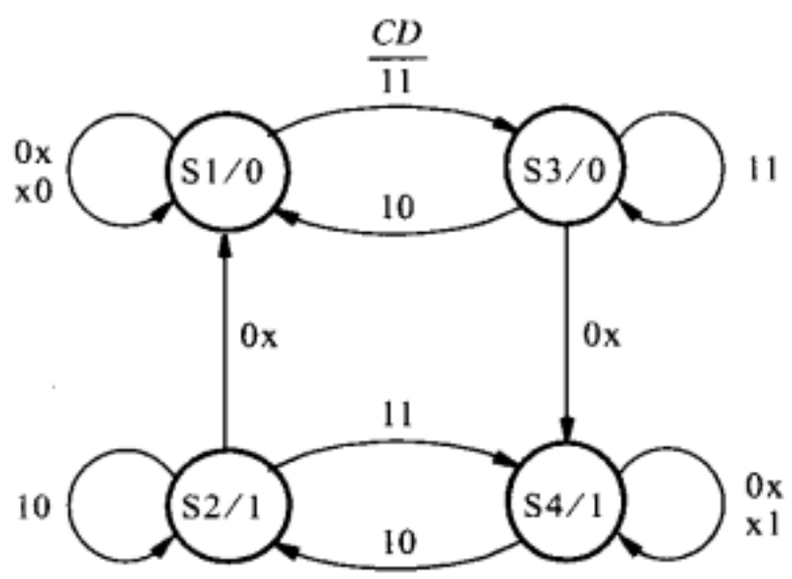


图9-7 主从D触发器的状态图

让我们更详细地分析这个有限状态机的行为。状态$S1$，此时$y_m y_s = 00$，除了$CD = 11$的情况外，所有其他的CD取值情况下，状态$S1$保持稳定。当$C = 1$时，D的值被保存在主触发器中，因此$CD = 11$将导致触发器的状态改变到$S3$，在该状态中，$y_m = 1$且$y_s = 0$。如果此时输入D重新变为0，而时钟信号仍保持为1，则触发器重新回到状态$S1$。$S1$状态和$S3$状态之间的转变表明：若$C = 1$，则主级的输出$Q_m = y_m$，将跟随输入信号D的变化而不影响从级。当时钟变为0时，电路将从$S3$状态转移到$S4$状态。在状态$S4$中，由于来自于主触发器的信号在时钟的负跳变沿时刻传送到了从触发器，从而主从两个触发器都将被置为1。此时触发器将保持在$S4$状态，直到时钟信号变为1且输入D变为0，此时电路状态才转移至$S2$状态。在$S2$状态中，主触发器被清为0，但从触发器仍保持1。此时触发器再次可能改变到$S4$状态，因为当$C = 1$的时候，主触发器将跟随输入信号D而改变。当时钟信号变低时，电路将从状态$S2$变为状态$S1$。

在图9-6与图9-7中，我们曾指出触发器只有一个输出Q，当电路被视为一个负沿触发的触发器时，人们是这样看的。从外部观察者的角度来看，触发器只有两种状态：0和1。但是从内部来说，触发器由主触发器和从触发器组成，因此产生了上面所描述的4种状态。

我们应该检查一下这样的基本假设：同一时刻只能有一个输入信号发生变化。如果电路

稳定于CD =10的状态$S2$，电路不可能受到CD = 01的影响，从状态$S2$转移到状态$S1$，因为这种C和D两个输入信号同时发生改变的现象是不应该发生的。因此，在流程表的第二行中，CD = 01对应的项应该被标记为未定，而不是标记为由$S2$状态转变为$S1$状态。从状态$S2$至状态$S1$的转变只发生在CD从10改变到00的时候。同样，如果电路稳定地处于CD = 11的$S3$状态，此时使CD = 00并不能使电路改变为$S4$状态。流程表中的对应项同样也应该标记为未定。最后得到的流程表如图9-6c所示。

若我们将分析过程颠倒过来，用如图9-6a所示的状态分配，来综合Y_m 和Y_s 的逻辑表达式，便可以得到：

$$Y_m = \overline{C}D + Cy_m + y_m D$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

表达式中的$y_m D$和$y_m y_s$是冗余项。如前所述，把它们综合进电路是为了避免出现冒险竞争，这一问题我们将在9.6节中讨论。 ■

例9.3 考虑图9-8所示的电路，该电路可以用以下的逻辑表达式表示：

$$Y_1 = y_1\overline{y}_2 + w_1\overline{y}_2 + \overline{w}_1\overline{w}_2 y_1$$
$$Y_2 = y_1 y_2 + w_1 y_2 + w_2 + \overline{w}_1\overline{w}_2 y_1$$
$$z = \overline{y}_1 y_2$$

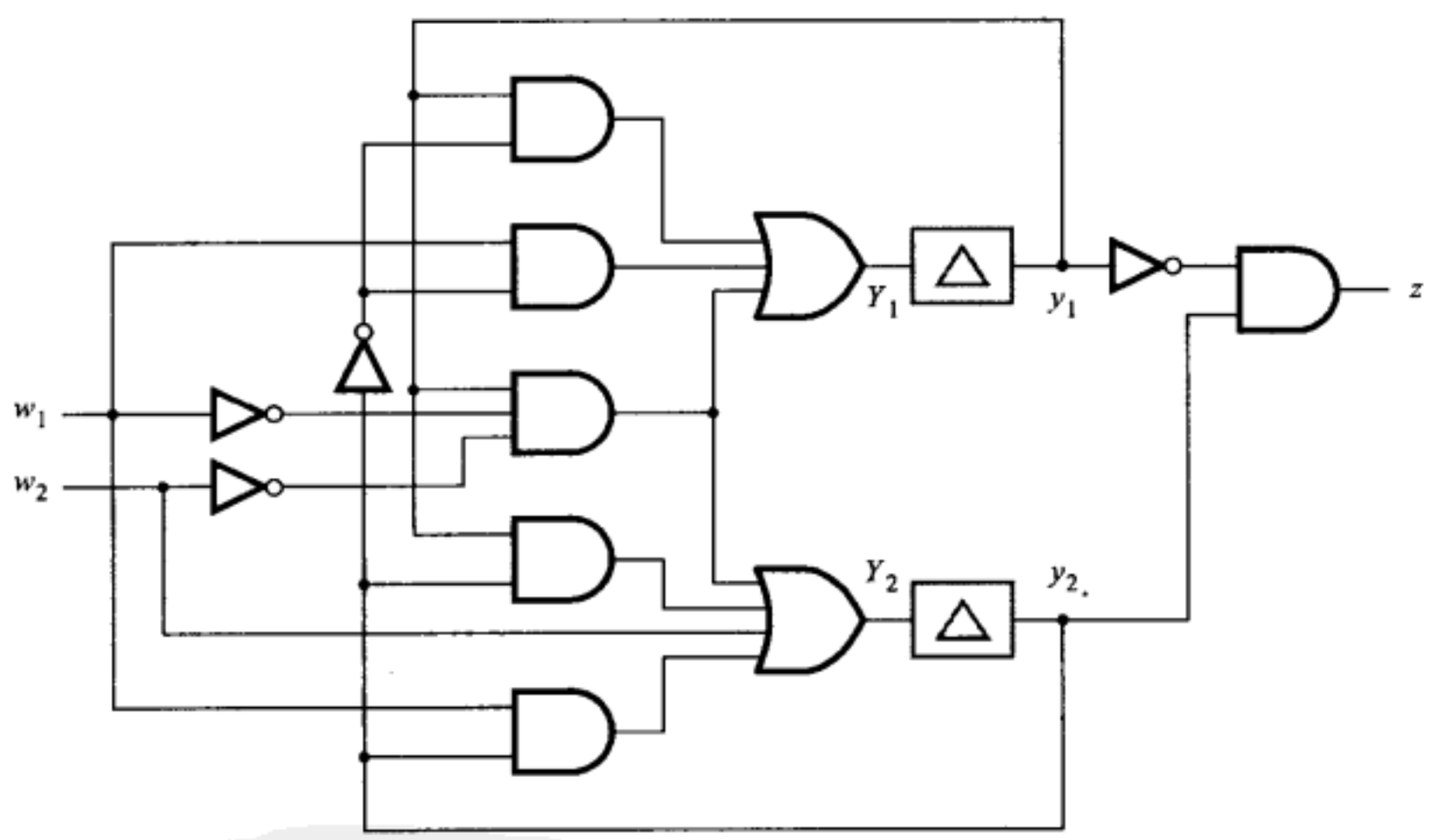


图9-8 例9.3的电路

相对应的激励表与流程表由图9-9给出。

流程表中的一些状态转移在实际电路中是不可能发生的，因为我们假定w_1 和w_2是不能同时改变的。在状态A中，输入取值$w_2 w_1$ = 00时，电路处于稳定状态A。电路的输入不可能在没有经过01或10的情况下直接变为11，经过01或10将分别使电路达到新状态B或C。因此，在输入$w_2 w_1$ = 11的情况下，从状态A的转移项可以留着不作指定。同样，若电路处于稳定状态B，此时$w_2 w_1$ = 01，也不可能将输入改变为$w_2 w_1$ = 10使电路直接进入状态D，则表中对应项的内容也应该不予以指定。若电路处于稳定状态C，此时$w_2 w_1$ = 11，也不可能将输入直接改为$w_2 w_1$ = 00使电路进入状态A。然而，每次改变一个输入是可以使电路从状态C转移到状态A的，因为无论在$w_2 w_1$ = 01还是$w_2 w_1$ = 10时，电路还都能保持在稳态C中。

当前状态 y_2y_1	下一状态 $w_2w_1=00$	01	10	11	输出 z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	00	01	10	11	0
01	11	01	11	11	0
10	00	10	10	10	1
11	11	10	10	10	0

a) 激励表

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z
A	A	B	C	D	0
B	D	B	D	D	0
C	A	C	C	C	1
D	D	C	C	C	0

b) 流程表

图9-9 图9-8中的电路的激励表和流程表

若输入$w_2w_1=00$时电路处于稳态D，就出现了一种新的情况。此时表中$w_2w_1=11$的项看起来似乎应该被设置为不确定，因为在稳态D时，输入值这样的改变是不可能发生的。但假设$w_2w_1=01$时，电路处于稳态B。此时令输入改变为$w_2w_1=11$，则电路将进入D状态。电路确实进入D状态，但这种输入条件下，电路进入状态D后并不稳定。当电路刚进入状态D，输入条件$w_2w_1=11$立刻要求电路进入状态C。然后，只要两个输入都保持1，电路将稳定在状态C。我们可以得出结论：输入条件为$w_2w_1=11$时，表中表示从状态D转移为状态C的项是有其意义的，不应被省略。电路从稳态B经过状态D达到稳态C的转移过程说明：从某一稳态直接到另一稳态的转移要求并非是强制的，中间允许有一个过渡状态。电路由一个稳定状态到达另一个稳定状态经由的中间状态称为*不稳定状态*。只要不稳定状态不会产生不想要的输出信号，经由不稳定的中间状态的转移过程是无害的。举例说明：状态转移过程发生于两个稳态之间，此时电路期望的输出信号应该为0，若稳态之间的状态转移经由了一个不稳定的中间状态，导致期望输出信号出现了1的非稳态，这是不可接受的。即使电路非常快地通过了不稳定状态，输出信号的尖脉冲仍有可能造成麻烦。在本例中这样的过程并没有什么问题。当电路处于稳态B时，输出$z=0$。当输入改变为$w_2w_1=11$，电路进入状态D的转移过程中，输出保持为0。只有当电路最终转移到状态C时，输出z才会改变为1。因此，输出z从0到1的改变在电路状态的整个转移过程中只发生了一次。

图9-10所示为一个修改后的流程表，其中展示了未指定的状态转移。该表说明了图9-8中电路的状态转移行为。如果不了解某给定电路如何进行工作，我们很难为该电路找到实际应用。所幸的是，在实际工作中电路的用途都是知道的，并且设计者所完成的分析，就是为了确保电路按期望的那样运行。在我们的例子中，很显然电路在状态C时，会产生输出$z=1$。进入状态C是电路在某些输入模式下得到的结果，而这些输入模式是用其他三个状态检测到的。由图9-10可以得到如图9-11所示的状态图。

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z
A	A	B	C	—	0
B	D	B	—	D	0
C	A	C	C	C	1
D	D	C	C	C	0

图9-10 例9.3改进后的流程表

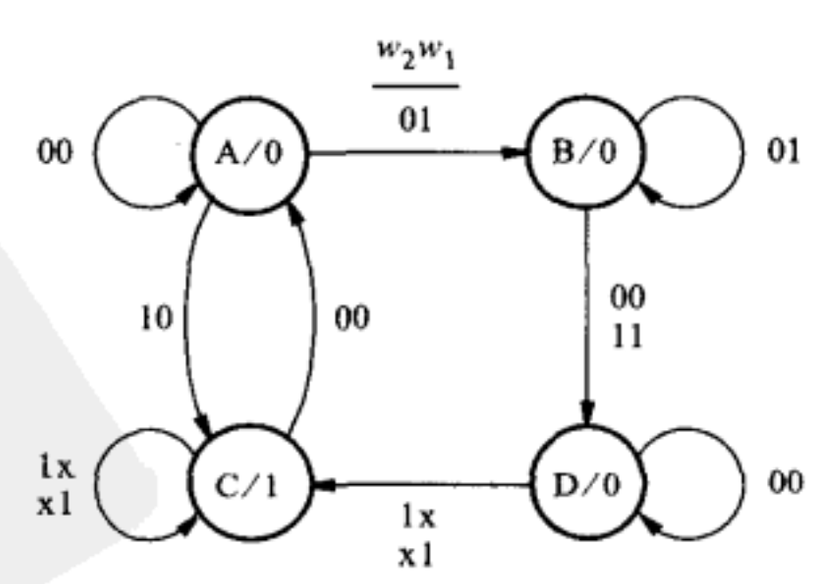


图9-11 例9.3的状态图

该状态图实际上实现了一个简单的自动售货机的控制功能。该售货机接受两种硬币：一角和五分，提供诸如糖果的商品。如果w_1代表五分硬币而w_2代表一角硬币，则只有当总价达到10分的时候才能使有限状态机进入C状态，售出糖果。硬币机械装置一次只能接收一枚硬币，这意味着$w_2 w_1 = 11$的情况永远不可能发生。因此，上面提到从状态B到状态C中间经过不稳定状态D的转移是不会发生的。可以看出，状态B和状态D都表示当前已积累了五分。状态B表示硬币接收器刚检测到收进一枚五分硬币，而状态D则表示已存有五分且硬币接收器当前是空的。在状态D时，还可以投入一枚五分或一角硬币，无论哪个操作都会导致状态转移到C。D状态并不区别两种硬币，因此若总共投入15分钱后，机器不会找零。当处于状态A的时候，一枚一角硬币将直接使机器转移到状态C。由于$w_2 w_1 = 11$的情况不会发生，因此允许我们将流程表填写成如图9-12所示的形式。若我们使用图9-9a中的状态分配来综合Y_1和Y_2的积之和逻辑表达式，则最后将得到如图9-8所示的电路。■

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	(C)	(C)	–	1
D	(D)	C	C	–	0

$w_2 \equiv$ 一角　　$w_1 \equiv$ 五分

图9-12 简单自动售货机的流程表

分析过程的步骤

我们借助于几个范例，用图表说明了分析的过程。其中必需的步骤如下：

1) 首先我们用通用模型的形式（见图8-88所示）对给定的电路进行解释。在该模型中，每一条反馈回路都被切断，并且在每个切断点上插入一个延迟元件。延迟元件的输入信号代表相应的下一个状态变量Y_i，而输出信号则表示当前的状态变量y_i。断点可以设置在环形反馈回路上的任意地方，但是每条回路（状态变量）只允许有一个断点。这样做，必须设置的断点数量将会最少，因而除了延迟元件的输出外，电路中不存在任何其他反馈。这个断点的最小数目有时被称为割集。请注意，对同一个给定回路，断点位置的不同，通过分析得到的流程表可能会不一样。但从所施加的输入和产生的输出而言，这两个不同的流程表反映了相同的功能行为。

2) 下一个状态和输出表达式均来自于对给定电路进行的分析。

3) 推导出与下一个状态输出表达式相对应的激励表。

4) 填写流程表，为每个特定的编码状态起一个（任意的）名字。

5) 若有需要，则根据流程表画出相应的状态图。

9.3 异步电路综合

异步时序电路综合的基本步骤与第8章中讨论过的同步电路综合的步骤相类似，但也有些不同。由于异步电路本质上的特点，使得异步电路的设计难度更大些。我们通过调研几个设计示例来说明这些不同。异步电路设计的基本步骤如下：

1) 根据实现所需功能行为的有限状态机描述，推导并画出状态图。

2) 推导出流程表，并尽可能地减少状态的个数。

3) 进行状态分配，并得出激励表。

4) 求得下一个状态和输出的逻辑表达式。

5) 构建能实现上述逻辑表达式的电路。

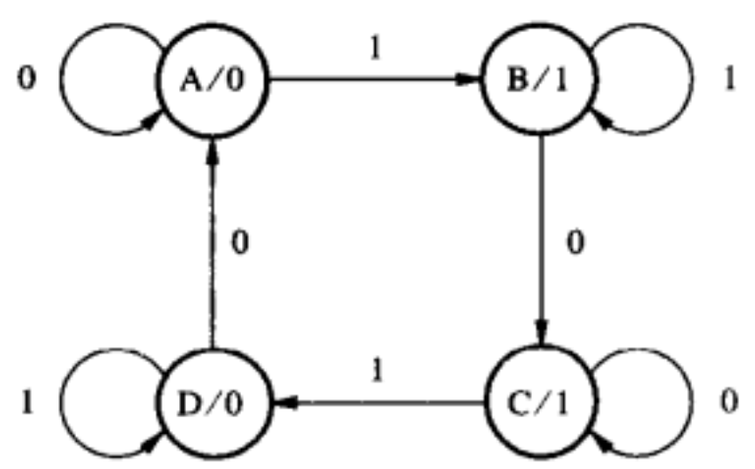
在推导出状态图或直接求得流程表的时候，最重要的是确保当电路处于稳定状态的时候能产生正确的输出信号。若电路必须经过一个不稳定状态，必须确保该状态不会产生不想要

的输出信号。

状态化简并不那么直接和容易。在9.4节中我们将讨论状态化简的步骤。状态分配的目的并非只是降低所设计的最终电路的成本。在异步电路中，状态分配不当会导致电路不可靠。我们将通过以下几个例子来说明这个问题。

例9.4（串行奇偶校验发生器） 假设我们想要设计这样一个电路，其输入为w，输出为z。将脉冲序列输入w，若前面输入的脉冲个数为偶数，则输出z等于0，若前面输入的脉冲个数为奇数，则输出z等于1。因此电路的行为相当于一个串行奇偶校验发生器。

设状态A表示已经收到的脉冲个数为偶数。由摩尔模型，当电路处于状态A时输出z将等于0。只要$w=0$，电路都应该保持在稳定状态A，这样的过程可以用一个从状态A起到状态A结束的弧线来表示。因此，当$w=0$时状态A是稳定的。当下一个脉冲序列到达的时候，输入$w=1$应该使FSM转到一个新的状态，即状态B，该状态使输出$z=1$。当FSM到达状态B时，只要$w=1$，它就将保持稳定于该状态，该过程可以用从状态B起到状态B结束的弧线来表示。下一个输入值的改变发生在w又变成0时。与之对应，有限状态机必须将状态转移到新状态，此时$z=1$，该状态对应这样一个事实：已接收到一个完整的脉冲，也就是说，输入w从1变回0。令该状态为状态C，则该状态在$w=0$的条件下保持稳定。而下一个脉冲的到来又会使得$w=1$，从而使有限状态机转移到新的状态D，该状态表明已接收到了偶数个脉冲，且最后的那个脉冲还未消失。状态D在$w=1$的情况保持稳定，同时使输出$z=0$。最后，当脉冲结束w回到0的时候，有限状态机回到状态A，该状态表明接收到偶数个脉冲，且当前输入$w=0$。由此可以画出如图9-13a所示的状态图。



a) 状态图

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	A	B	0
B	C	B	1
C	C	D	1
D	A	D	0

b) 流程表

图9-13 奇偶校验发生器的异步FSM

了解该图的关键点在于考虑只要能分清输入脉冲个数的奇偶之分，为何仍然必须用四个状态而不是两个。虽然状态B和状态C都说明观测到已输入的脉冲个数为奇数，但是这两个状态却不能合为一个。若我们只用状态B来达到此目的，则该状态必须添加一条边上标有0从状态B起到状态B结束的弧线。现在的问题是缺少了状态C，当输入$w=1$时，必须有一个从状态B直接到状态D的转移过程，作为新脉冲输入时产生的响应。而既要让状态B在$w=1$的条件下保持稳定，又要让状态B在同样的输入条件进入状态D，是根本不可能的。类似地，我们可以表明状态A和状态D也不能合为一个状态。

图9-13b展示了直接对应于状态图的流程表。在很多情况下，设计者可以直接求出流程表。我们使用得最多的是状态图，因为状态图提供了更为简明直观的说明FSM中状态转移关系的图形。

下一步工作是用状态变量的方式给各状态赋值。由于在我们的FSM中有4个状态，所以至少需要两个状态变量。将两个变量定义为y_1和y_2。作为第一次进行状态分配的尝试，令状态A，

B，C和D分别编码为：$y_2y_1 = 00$，01，10和11。由这样的赋值我们得到如图9-14a所示的激励表。不幸的是，该激励表有一个严重的缺陷。实现该激励表的电路在输入$w = 1$的条件下，稳定于状态$D = 11$。但考虑一下，若将输入改变为$w = 0$会发生什么情况。根据激励表，电路应当转移到状态$A = 00$，且在该状态中保持稳定。问题是在从$y_2y_1 = 11$转移到$y_2y_1 = 00$的过程中，两个状态变量都要改变其值才行。这不可能在同一时刻发生。在异步电路中，下一个状态变量的值是由具有不同传播延迟的逻辑门网络确定的。因此我们应该合理地假定某个状态变量将比另一个稍微早一点发生改变，从而使得电路对输入产生预先想不到的响应。假设y_1先改变，则电路的状态将从$y_2y_1 = 11$转移到$y_2y_1 = 10$，即进入状态C。一旦进入状态C，若此时$w = 0$，则电路将会保持当前的状态，这显然是一个错误的结果。另一方面，假设y_2先改变，则电路的状态将从$y_2y_1 = 11$转移到 $y_2y_1 = 01$，即进入状态B。由于此时$w = 0$，电路将试图转移到$y_2y_1 = 10$，即进入状态C。这就要求y_1和y_2都发生改变。假设在从$y_2y_1 = 01$转变的过程中y_1先发生变化，电路最后将发现自己处于$y_2y_1 = 00$的状态，即正确的目标：状态A。上述讨论指出：如果y_2先发生改变的话，所要求的电路从状态D到状态A的转移能正确地实现；若y_1先发生改变则不行。因此转移的成功与否取决于信号y_1和y_2之间竞争的结果。

当前状态 y_2y_1	下一状态 Y_2Y_1 $w = 0$	$w = 1$	输出 z
00	00	01	0
01	10	01	1
10	10	11	1
11	00	11	0

a) 不好的状态分配

当前状态 y_2y_1	下一状态 Y_2Y_1 $w = 0$	$w = 1$	输出 z
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

b) 好的状态分配

图9-14 对图9-13b的状态分配

某个输入信号的改变有可能引起多个状态变量同时发生改变，从而导致电路状态转移的不确定性。这种不确定性必须去除。术语*竞争条件*就是用于描述此类不可预知的行为的。我们将在9.5节中详细讨论该问题。

把当前的状态变量看做电路的输入，用这样的方式可以消除竞争条件。这就是说，在同一时间只能允许一个状态变量发生改变。就我们的例子而言，将状态分配定义为$A = 00$，$B = 01$，$C = 11$和$D = 10$就可以达到目的。由此产生的激励表如图9-14b所示。读者不难验证，所有的状态转移只涉及单个状态变量的改变。

从图9-14b可以求得下一个状态和输出的逻辑表达式如下：

$$Y_1 = w\bar{y}_2 + \bar{w}y_1 + y_1\bar{y}_2$$
$$Y_2 = wy_2 + \bar{w}y_1 + y_1y_2$$
$$z = y_1$$

Y_1和Y_2表达式中最后的乘积项是为了处理可能产生的冒险竞争，该内容将在9.6节中讨论。相应的电路如图9-15所示。

考虑一下如何通过同步电路来实现串行奇偶校验发生器将是一件很有意思的事情。所需的全部电路只是一个触发器，每个输入脉冲到来时，该触发器就改变自身的状态。图9-16所示的正跳变沿D触发器就能完成这个任务，假设触发器的初始状态为Q = 0。该触发器的逻辑复杂度跟图9-15所示的电路完全相同。确实，如果我们使用上述Y_1和Y_2的表达式，并用C替换

w，D替换 $\bar{y}_2$，y_m 替换y_1，y_s 替换y_2，将得到例9.2中所列的主从式D触发器的激励表达式。图9-15所示电路实际上就是一个负跳变沿触发的主从式触发器，且将该触发器的输出Q端（y_2）的非连接到其输入D端上。电路的输出z连接到触发器的主级输出上。

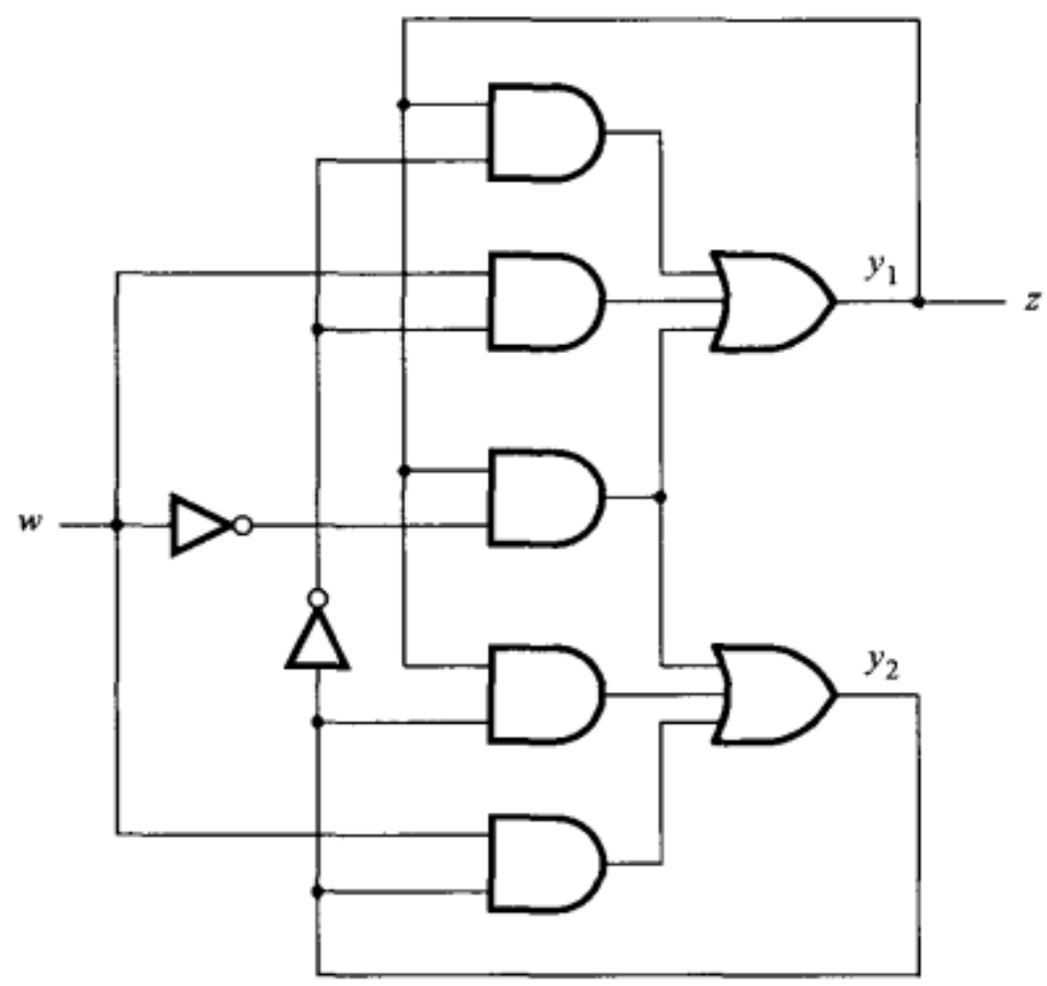


图9-15 实现图9-13b中的FSM的电路

例9.5（模4计数器） 在第7章和第8章中，我们曾阐述过如何使用触发器来构成计数器。现在我们将采用异步时序电路的解决方案来综合一个计数器。图9-17所示是模4增1计数器的状态图，该计数器统计输入信号线w上的脉冲个数。该电路必须能够对所有输入信号的变化作出响应，即无论在每个脉冲的正跳变沿还是负跳变沿，该电路必须都必须执行特殊的动作。所以，需要有8种状态来处理4个连续脉冲的跳变沿。

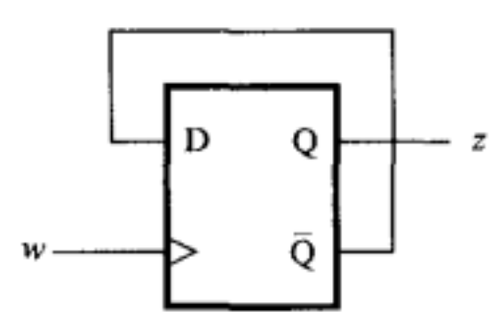


图9-16 例9.4的同步电路解决方案

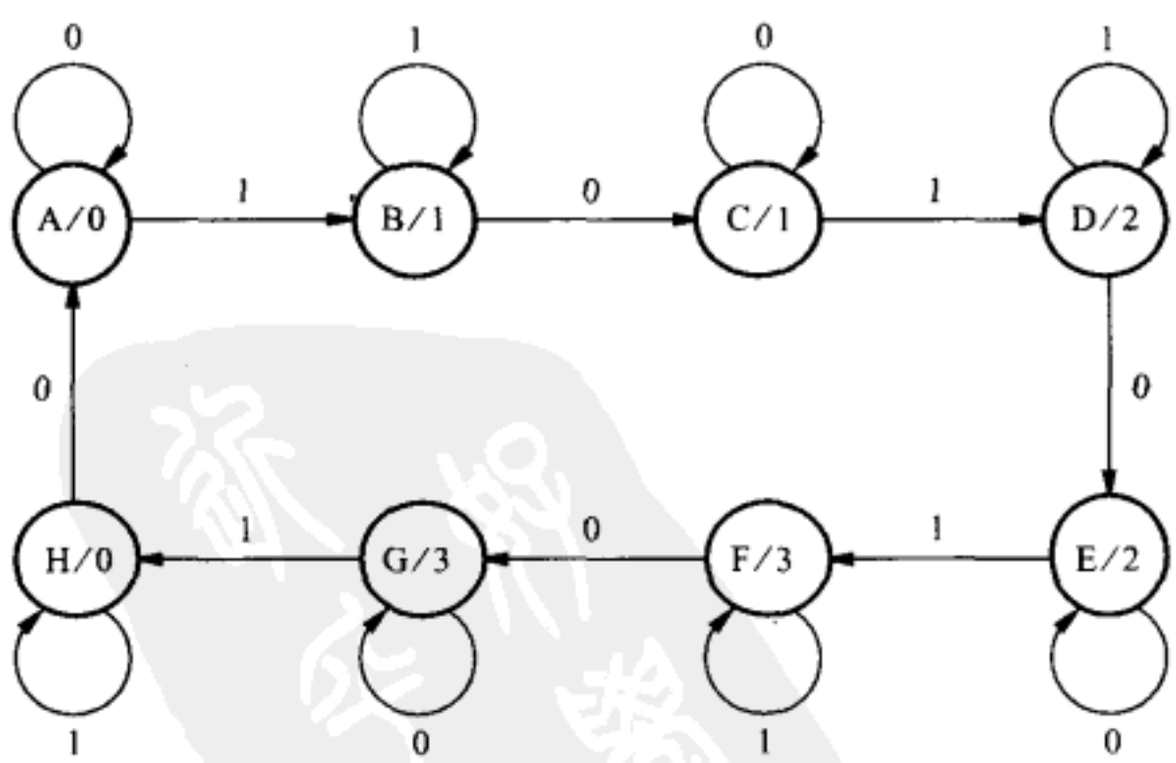


图9-17 模4计数器的状态图

开始时计数器处于状态A，当$w = 0$时，保持在状态A。当w改变为1时，计数器进入状态B，若维持$w = 1$不变，则稳定于状态B。当w回到0时，计数器进入状态C，若维持$w = 0$不变，则

稳定于该状态。当w再次变为1时，计数器则变为状态D，并继续这样的方式来改变状态。根据摩尔模型，各个状态对应着特定的计数值。每个特定的计数值对应着两个状态：一个是当脉冲到来w从0变到1时FSM所进入的状态；另外一个是当w在脉冲结束回到0时FSM所进入的状态。状态B和状态C对应的计数值为1，状态D和状态E对应的为2，而状态F和状态G对应的为3，状态A和状态H对应的计数值为0。

图9-18所示为计数器的流程表和激励表。状态分配的选择确保了所有状态之间的转移每次只能改变一位状态变量的值，从而消除可能发生的冒险竞争。计数器的输出用二进制数编码的变量z_1和z_2表示。从激励表可以得到下一个状态和输出的逻辑表达式如下：

$$
\begin{aligned}
Y_1 &= \overline{w}y_1 + wy_2y_3 + w\overline{y}_2\overline{y}_3 + y_1y_2y_3 + y_1\overline{y}_2\overline{y}_3 \\
&= \overline{w}y_1 + (w + y_1)(y_2y_3 + \overline{y}_2\overline{y}_3) \\
Y_2 &= wy_2 + \overline{w}y_1\overline{y}_3 + \overline{y}_1y_2 + y_2\overline{y}_3 \\
Y_3 &= wy_3 + y_1y_3 + \overline{y}_1y_2\overline{w} + y_2y_3 \\
z_1 &= y_1 \\
z_2 &= y_1y_3 + \overline{y}_1y_2
\end{aligned}
$$

这些表达式定义了实现所要求的模4脉冲计数器的电路。

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	A	B	0
B	C	B	1
C	C	D	1
D	E	D	2
E	E	F	2
F	G	F	3
G	G	H	3
H	A	H	0

a) 流程表

当前状态 $y_3y_2y_1$	下一状态 $w=0$ $Y_3Y_2Y_1$	下一状态 $w=1$ $Y_3Y_2Y_1$	输出 z_2z_1
000	000	001	00
001	011	001	01
011	011	010	01
010	110	010	10
110	110	111	10
111	101	111	11
101	101	100	11
100	000	100	00

b) 激励表

模8输出 $z_3z_2z_1$
000
001
010
011
100
101
110
111

c) 计数跳变沿的输出

图9-18 模4计数器的流程表和激励表

在前面的推导过程中我们设计了这样一个计数电路，该电路的状态在输入信号w的每个跳变沿均发生改变，总共需要有8个状态。由于要求该电路能对完整的脉冲计数，即包含上升沿和下降沿的脉冲，因此输出计数z_2z_1的值每经过两个状态才发生一次改变。该FSM的行为就像一个同步时序电路，只有当w从0变到1时其输出计数值才发生改变。

假设现在我们想要统计信号w值的改变次数，即统计输入信号w的跳变沿个数。图9-17和图9-18中指定的状态转移过程实际上也定义了实现该目标功能的模8计数器的FSM。我们只需要为每个状态指定各自独立的输出即可，计数器的输出结果见图9-18c所示。$z_3z_2z_1$的值表示了计数序列0，1，2，…，7，0。利用这样的输出的规定和图9-18b中的状态分配表，可以求得输出的表达式如下：

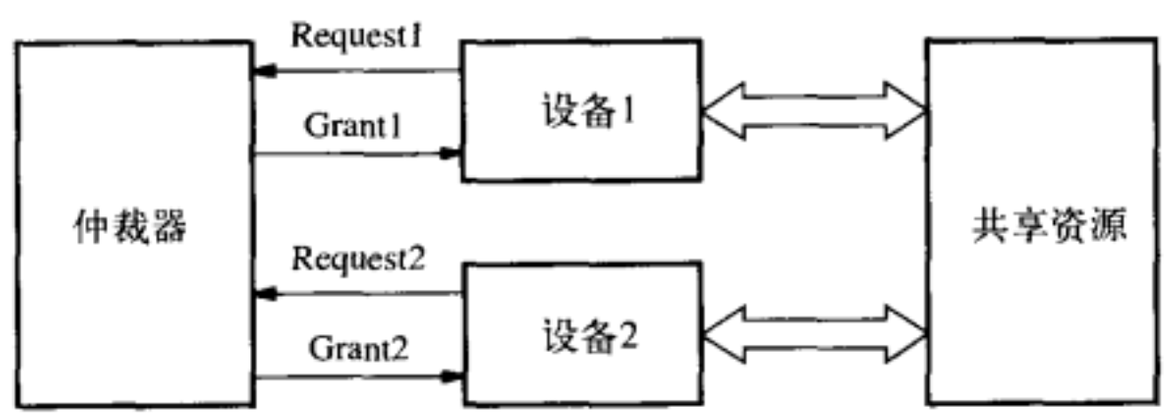
$$z_1 = y_1 \oplus y_2 \oplus y_3$$
$$z_2 = y_2 \oplus y_3$$
$$z_3 = y_3$$

■

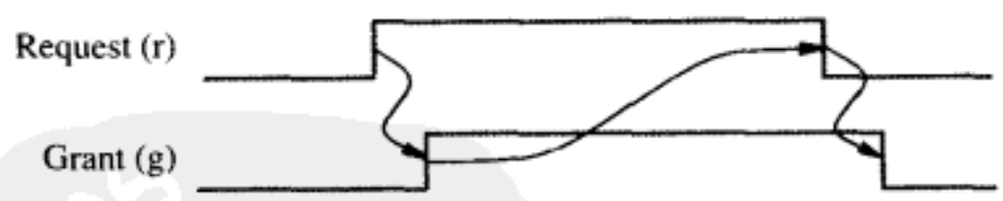
例9.6（简单的仲裁器） 在计算机系统中，有些资源由几个不同的设备共享经常是很有用的。通常，某个资源在同一时刻只能被某一个设备使用。当不同的设备都想要使用某资源的时候，就需要先提出申请，该申请将由仲裁电路来处理。当有两个或更多的申请时，仲裁器会根据某种优先权方案，如同我们在8.8节中讨论过的那样，从多个申请中选择某个申请进行处理。

现在我们将考虑一个简单的仲裁器例子，该仲裁器是用异步时序电路实现的。为了保证例子比较小，假设有两个设备在竞争使用某个共享资源，如图9-19a所示。每一个设备都通过两个信号：*Request*（请求）和*Grant*（同意）与仲裁器通信。当一个设备需要使用共享资源时，它首先将自己的请求信号置为1，然后一直等待到仲裁器以*Grant*（同意）信号响应这次请求为止。

图9-19b呈现了在异步环境中两个实体之间进行通信的常用方案，即称为*握手信号*的方案。一般需要使用两个信号才能提供握手机制。一个设备通过提出申请（置$r = 1$）来启动通信过程，当共享资源可用的时候，仲裁器便发出同意信号$g = 1$来作出响应。当设备接收到同意信号时，该设备开始使用所请求的共享资源。当设备对资源的使用完毕时，令$r = 0$来撤销自己提出的请求。当仲裁器看到$r = 0$的信号，它解除同意信号使$g = 0$。图中的箭头表示了该方案中的因果关系；一个信号的变化会引起另一个信号发生变化。有因果关系的信号之间发生变化的间隔时间取决于该电路的具体实现。该电路的关键点在于，采用这种解决方案不需要同步的时钟。



a) 仲裁结构



b) 握手信号

图9-19 仲裁器举例

图9-20展示了这个简单仲裁器的状态图。电路有两个输入信号（请求信号r_1和r_2）和两个输出信号（同意信号g_1和g_2）。该状态图绘出了所需*FSM*的摩尔模型，其中状态转移弧线标记为$r_2 r_1$，状态输出弧线标记为$g_2 g_1$。状态*A*为静止状态，此时没有任何请求。状态*B*表示设备1得到允许使用资源的情形，而状态*C*表示设备2得到了允许。因此状态*B*在$r_2 r_1 = 01$时保持稳定；而状态*C*在$r_2 r_1 = 10$时保持稳定。为了遵守异步电路设计的规则，我们假设输入r_1和r_2在同一时刻只能有一个有效。因此，处于状态A的电路不可能从$r_2 r_1 = 00$的状态直接改变到$r_2 r_1 =$

11的状态。状态$r_2 r_1 = 11$仅仅发生在当一个设备在得到允许后还没有使用完共享资源时，第2个请求就来到的情况。这种情况只能发生在状态B和状态C。当FSM稳定地处于状态B或者状态C时，如果r_2和r_1都变为1，它将继续保持原来的状态。

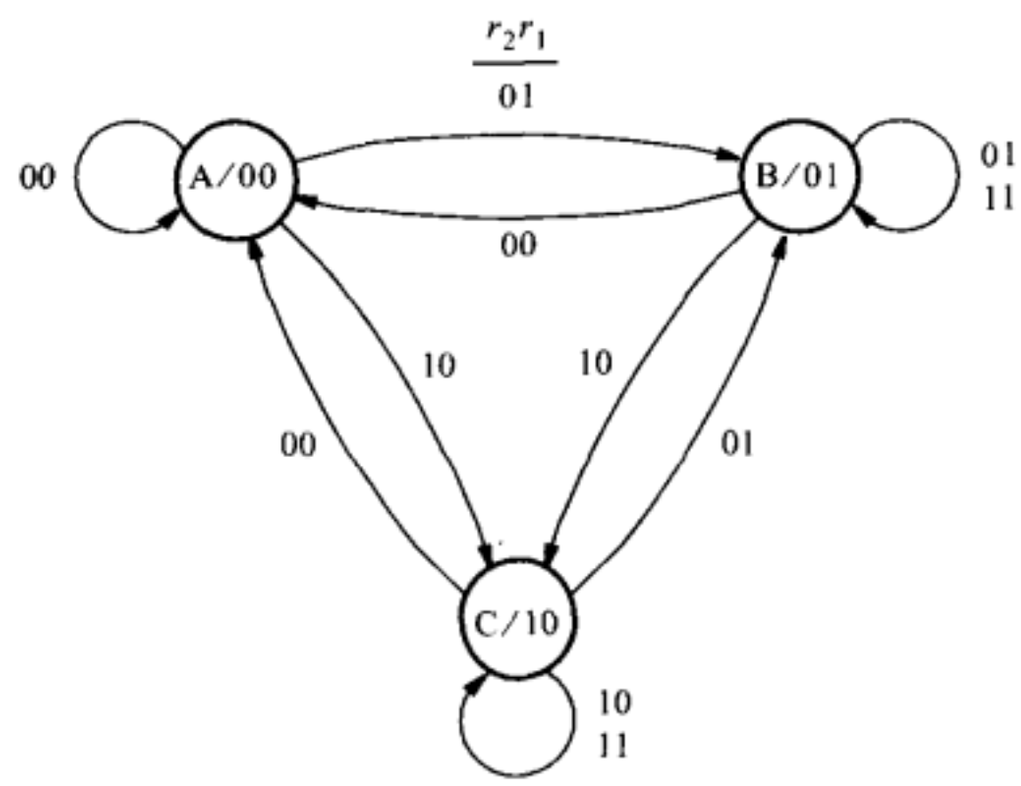


图9-20 仲裁器的状态图

该仲裁器的状态流程表如图9-21a所示，激励表如图9-21b所示。在图中显然不可能找到一种状态分配可以使状态A、状态B和状态C之间的转移只涉及一个状态变量的改变。图中所选择的分配中，所有从状态A起或到状态A的转移都能被恰当地处理，但是状态B和状态C之间的转移则包含了两个状态变量y_1和y_2的变化。假设电路在输入值为$r_2 r_1 = 11$的条件下稳定于状态B。现在令输入改变为$r_2 r_1 = 10$，这必然引起状态转移到状态C，这意味着状态变量必须从$y_2 y_1 = 01$改变为$y_2 y_1 = 10$。如果y_1的改变比y_2快，则电路将会瞬时进入状态$y_2 y_1 = 00$，然后转移到最后想要的状态，因为在输入$r_2 r_1 = 10$的条件下，状态A将由指定的状态转移到状态C。但是如果y_2改变得比y_1快，电路将达到状态$y_2 y_1 = 11$，该状态在流程表中没有定义。为了确保电路即使在这种情况下也能进入要求的状态C，我们可以将状态$y_2 y_1 = 11$标记为状态D添加到激励表中，且指定一个相应的状态转移，见图9-21b所示。当电路在$r_2 r_1 = 11$的条件下稳定于状态C时，若r_2从1变为0的时候，电路则不得不转移到状态B，也将出现类似的情况。

当前状态	下一状态 r_2r_1 = 00	01	10	11	输出 g_2g_1
A	A	B	C	–	00
B	A	B	C	B	01
C	A	B	C	C	10

a) 流程表

	当前状态 y_2y_1	下一状态 r_2r_1 = 00	01	10	11	输出 g_2g_1
		Y_2Y_1				
A	00	00	01	10	–	00
B	01	00	01	10	01	01
C	10	00	01	10	10	10
D	11	–	01	10	–	*dd*

b) 激励表

图9-21 仲裁器的实现

添加的状态D的输出值$g_2 g_1$被标记为无关（don't-care）状态。无论其中哪个输出从0变为1或者从1变为0，只要当电路处于稳定状态时总能产生正确的输出值，则变化发生的具体时间并不重要。指定无关状态可以简化输出函数。在化简过程中，确保未指定的输出将不会产生有可能导致错误行为的值是十分重要的。由图9-21b可发现，当电路经过不稳定状态D的瞬间，

让输出变为$g_2g_1 = 11$是可以的。该输出在这个例子中不会产生破坏作用，因为此时某设备刚用完共享资源，不会再次使用该资源，直到该资源的同意信号回到0，表明这次与仲裁器的握手已结束了为止。请注意，当电路从状态B转移到状态C的时候，若出现这个条件则g_1从1变到0稍微慢些，而g_2从0变到1稍微快点。类似地，如果这个过程是从状态C转到状态B，此时g_1从0变到1将稍微快一点，而g_2从1变到0将稍微慢一点。在这两种情况中，无论g_1或g_2上都不会出现毛刺。

由激励表可推导出如下所示的（表示下一个状态和输出的）逻辑表达式：

$$Y_1 = \bar{r}_2 r_1 + r_1 \bar{y}_2$$
$$Y_2 = r_2 \bar{r}_1 + r_2 y_2$$
$$g_1 = y_1$$
$$g_2 = y_2$$

将上面前两个表达式改写为如下的形式：

$$Y_1 = r_1(\bar{r}_2 + \bar{y}_2)$$
$$= r_1 \overline{r_2 y_2}$$
$$Y_2 = r_2(\bar{r}_1 + y_2)$$

根据上面的表达式，可得到如图9-22所示的电路。可以看到，该电路对输入信号变化的响应十分迅速。这跟8.8节中讨论过的仲裁器电路形成了非常鲜明的对比，那个仲裁器电路的最小响应时间由同步时钟的周期决定。

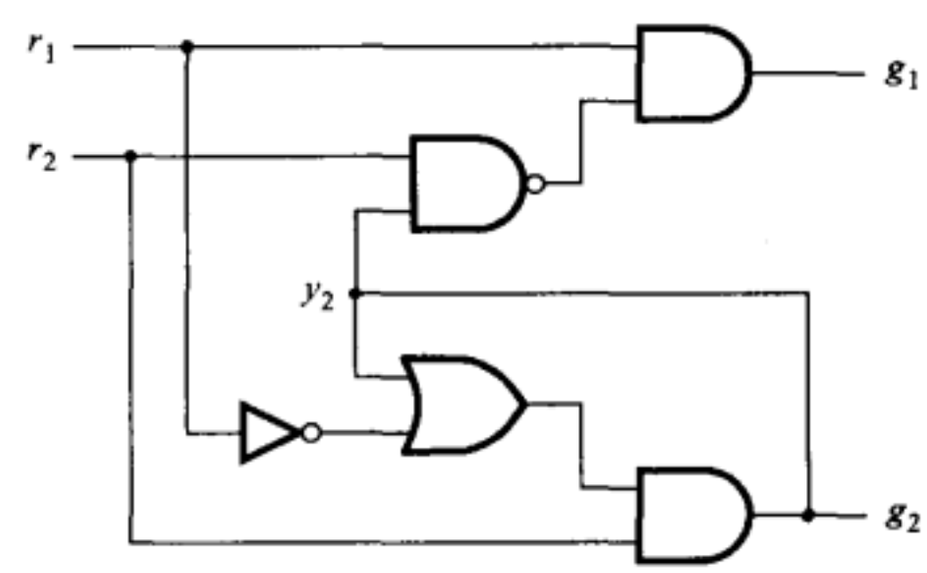


图9-22 仲裁器电路

由状态B和状态C之间的状态转移产生的冒险竞争难点还可以通过别的方式来解决：我们可以简单地禁止电路进入不确定状态。图9-23a展示了一个改进后的流程表，其中从状态B到状态C的转移必须经由状态A完成。若电路稳定在状态B，此时输入从$r_2r_1 = 11$改变到$r_2r_1 = 10$，则电路首先会转移到状态A。一旦电路进入状态A，而状态A在输入为$r_2r_1 = 10$的条件下是不稳定的，因此电路将继续转向状态C。此时绕道不稳定状态A是可以接受的，因为绕道过程中输出$g_2g_1 = 00$，这与仲裁器所期望的动作一致。从状态C转移到状态B也可以通过同样的途径处理。根据如图9-23b所示的改进后的激励表，可推出如下所示的（表示下一个状态和输出的）逻辑表达式：

$$Y_1 = r_1 \bar{y}_2$$
$$Y_2 = \bar{r}_1 r_2 \bar{y}_1 + r_2 y_2$$

当前状态	下一状态 $r_2r_1 = 00$	01	10	11	输出 g_2g_1
A	A	B	C	—	00
B	A	B	A	B	01
C	A	A	C	C	10

a) 改后的流程表

当前状态 y_2y_1	下一状态 $r_2r_1 = 00$	01	10	11	输出 g_2g_1
	Y_2Y_1				
00	00	01	10	—	00
01	00	01	00	01	01
10	00	00	10	10	10

b) 改进后的激励表

图9-23 避免出现图9-21a所示的临界竞争的另一种方法

这些表达式使得生成的电路不同于图9-22所示的电路。然而，两个电路都能实现仲裁器要求的功能。

接下来我们将尝试使用米利模型规则来设计同样的仲裁器。由图9-20可以看出状态B和状态C是根本不同的，因为在同样的输入$r_2 r_1 = 11$的条件下，这两个状态却产生不同的输出。相比之下，状态A很特别，无论何时只要$r_2 r_1 = 00$，它就将产生输出$g_2 g_1 = 00$。如果使用的是米利模型，这种情形可以在状态B和状态C中指定。图9-24给出了一个合适的状态图。图9-25介绍了流程表和激励表，并因此得到以下的表达式：

$$Y = r_2\bar{r}_1 + \bar{r}_1 y + r_2 y$$
$$g_1 = \bar{r}_1 y$$
$$g_2 = r_2 y$$

尽管只需要一个状态变量，与图9-22所示的摩尔模型电路比较，这个电路需要更多的门才能实现。■

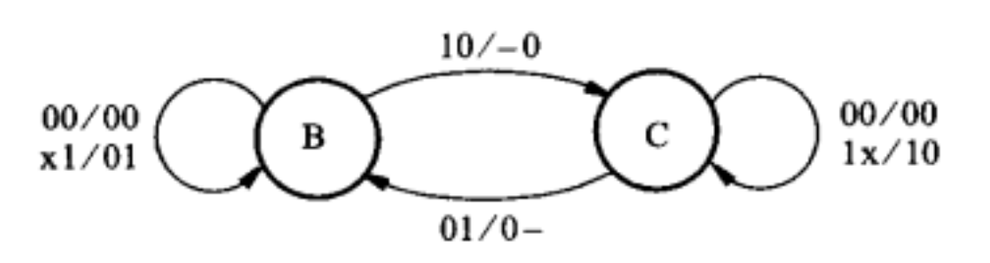


图9-24 仲裁器FSM的米利模型

当前状态	下一状态 r_2r_1 = 00	01	10	11	输出g_2g_1 00	01	10	11
B	B	B	C	B	00	01	-0	01
C	C	B	C	C	00	0-	10	10

a) 流程表

当前状态 y	下一状态 r_2r_1 = 00	01	10	11	输出 00	01	10	11
	Y				g_2g_1			
0	0	0	1	0	00	01	$d0$	01
1	1	0	1	1	00	$0d$	10	10

b) 激励表

图9-25 仲裁器FSM的米利模型实现

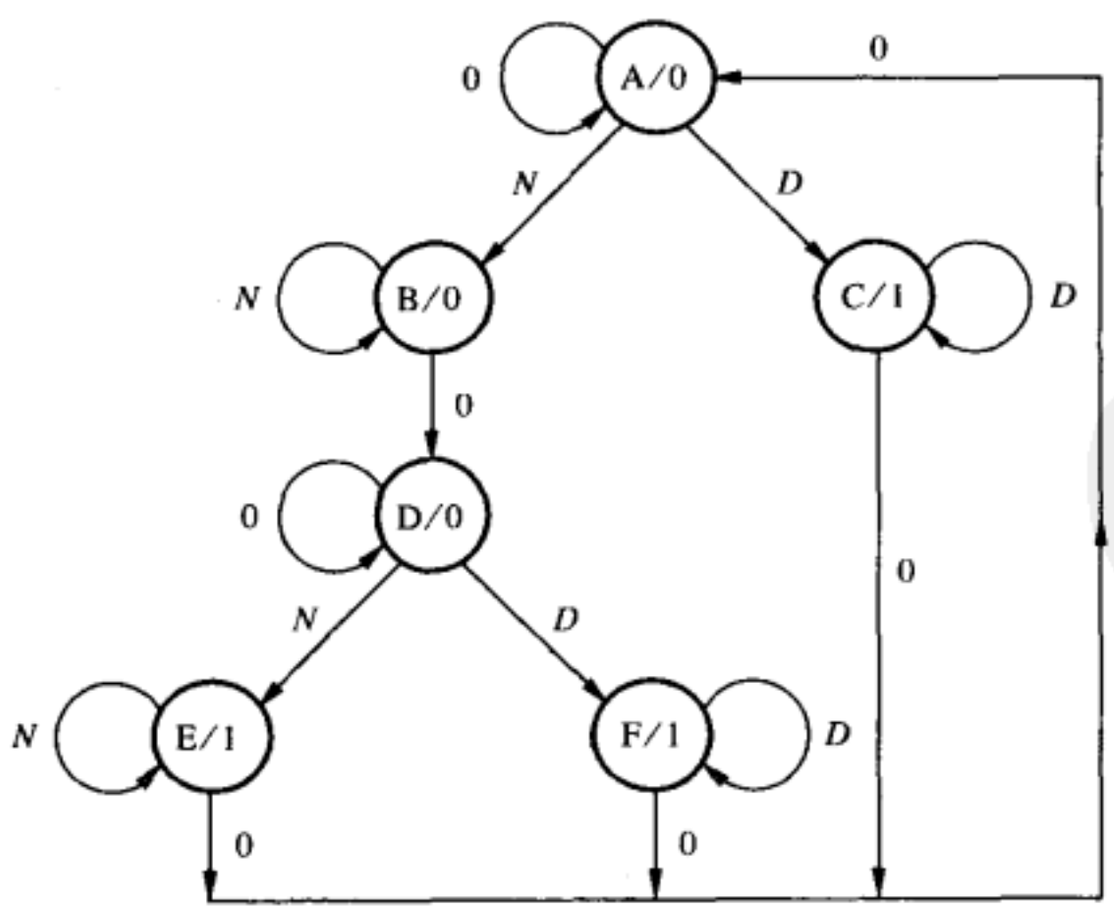
上面几个例子告诉我们一个重要的概念：必须非常小心地注意状态的分配，以避免状态变量发生变化时出现冒险竞争现象。在9.5节中我们将就这个问题进行更深入的讨论。

我们曾作过这样一个基本的假设：对仲裁器FSM的请求输入信号每次只能有一个发生改变，以使得电路在下一个改变发生之前能达到稳定状态。如果设备是完全独立的，设备的请求信号可以在任意时刻产生。假设设备每几秒钟才会产生一个请求信号，而仲裁器只需要几个纳秒就可以从一个稳定状态转到另一个稳定状态，因此两台设备几乎不可能产生时间上足够靠近的请求信号而使得仲裁器产生错误的输出。然而，尽管因为请求信号同时到达而造成错误的可能性极其微小，但却不是零。如果这个微小的错误概率不可容忍的话，我们就可以将两个请求信号馈入一个特别的元件：*互斥元件*。该元件有两个输入及两个输出。如果输入都为0，则输出也都为0。如果只有一个输入为1，则相应的输出也为1。而当两个输入都为1时，电路使其中一个输出为1而另一个保持为0。采用互斥元件将使仲裁器的设计稍微有些变化，因为输入$r_2 r_1 = 11$的情况再不会出现，因此图9-21中$r_2 r_1$为11的列上的项都成为无关项。关于互斥元件及多个输入信号同时改变的问题在参考文献[6]中有详细的介绍。最后，我们应该注意到在同步电路中，当输入电路的一个或多个信号并非来自于由一个共同时钟所控制的电路时，将会产生类似的问题。我们将在10.3.3节中来处理这个问题。

9.4 状态化简

在第8章中我们曾介绍过，减少实现给定功能的FSM所需的状态个数通常可以使得状态变量也减少，这意味着在相应的同步时序电路中需要的触发器也减少。在异步时序电路中，减少状态的数量同样有用，因为这样做可以使实现电路变得更简单。

当设计异步FSM的时候，因为设计者必须遵守在同一个时刻只能有一个变量发生改变的约束，所以最初的流程表里可能会有很多未指定（无关）项。例如，假设我们想要设计例9.3中的简单售货机的FSM。回想一下，该机器接受五分和一角的硬币，且当收到的钱数到达10分的时候给出糖果；若投入1角5分，则机器并不找零。对于这样的FSM，其初始的状态图我们可以用直接的方式来得到，也就是列举所有能使总金额至少为10分的投币序列。图9-26a 展现了一个以摩尔模型定义的可能的状态图。图中以复位状态A开始，FSM保持该状态直到有币投入。该情况由一个标号为0的弧线来表示，此时，$N = D = 0$。现在我们用一个标号为N的线来表示硬币感应器接收到了一枚五分硬币且产生信号$N = 1$。类似地，用标号D来说明接收到了一枚一角硬币。如果此时$N = 1$，则FSM将变到一个新的状态，即状态B，只要$N = 1$继续，FSM将继续保持在该状态。因为状态B表示总金额为5分的钱币已经投入，此时的输出为0。当处于状态A时接收到了一枚一角硬币，则FSM转到一个新的状态，即状态C。随后D保持等于1，FSM将保持在C状态，且应该给出一份糖果，即令FSM的输出为1。以上两个过程是从状态A开始的仅有的状态变化，因为在同一时刻不可能同时插入两枚硬币，这就意味着$DN = 11$的情况不用考虑。此外，由于硬币感应器在接收到第一枚硬币的一段时间后将继续探测第二枚插入的硬币，在状态B中必须有使$DN = 00$的变化。该操作与在一个时刻只能有一个输入变量发生改变的要求是一致的，所以从$DN = 01$直接到$DN = 10$的变化是不允许的。在状态B中，$DN = 10$的状态是不可能发生的，因此也不必考虑。输入$DN = 00$将FSM改变到一个新的状态，即状态D。该状态表明现在金额为5分且当前硬币感应器中已没有硬币。进入状态D后，机器可能接收到一角或者5分的硬币。若$DN = 01$，则机器改变到状态E，该状态表示当前金额为10分，且产生输出为1。若$DN = 10$，则FSM转移到状态F，产生输出为1。最后，当FSM进入状态C、E或F中的任何一个后，唯一可能的输入是$DN = 00$，只有该输入才能使得FSM返回状态A。



a) 初始状态图

当前状态	下一状态 DN = 00	01	10	11	输出 z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	F	–	0
E	A	(E)	–	–	1
F	A	–	(F)	–	1

b) 初始流程表

图9-26 简单自动售货机状态机的推导

图9-26b给出该FSM的流程表，该表明确地指出了所有的无关项。这些无关项给我们提供了某种程度的灵活性，可根据不同情况，减少FSM所需要的状态数量。请注意，该表中的每一行都只有一个稳定的状态，这样的表（即每一行只有一个稳定状态的表）通常称之为*基本流程表*（primitive flow table）。

人们开发了若干种技术可用来进行状态化简。在本节中，我们将介绍一种分两个阶段的状态化简方法。第一阶段，应用8.6.1节中所阐述的划分过程。假设流程表中潜在的等价行必定产生同样输出。作为一个附加的约束，对于可能等价的两行，任何无关项必须位于下一个状态的相同的列中。因此把这些等价状态合并为一个状态将不会丢失任何无关项及这些项提供的灵活性。第二阶段，利用无关项把这些等价行合并起来。若不同的两行不包含有冲突的下一个状态项，则这两行就可以合并。这表示这两个等价行的下一个状态项对于任何给定的输入都是相同的，或者其中一个为无关项，或者两行均为同一个稳定状态。若使用的是摩尔模型，则这两行（两个状态）必须产生同样的输出。若所用的为米利模型，则这两行对于均已进入稳定状态的任何输入取值，必须产生同样的输出。

例9.7 我们现在将说明如何将图9-26b所示的流程表简化为如图9-12所示的优化形式。状态化简过程的第一阶段工作是进行8.6.1节中所述的状态划分。状态A和D都在输入$DN = 00$的情况下进入稳定，产生的输出都是0；同时这两个状态的无关项的位置也相同。状态C和F在输入$DN = 10$的情况下进入稳定，都产生$z = 1$且有同样的无关项。状态B和E有相同的无关项，但是当它们在输入条件$DN = 01$的情况下进入稳定，状态B产生$z = 0$，而状态E却产生$z = 1$；它们的输出各不相同。由此，得到如下的初始的状态划分：

$$P_1 = (AD)\ (B)\ (CF)\ (E)$$

状态A和D的后续状态在$DN = 00$时为（A，D），在$DN = 01$时为（B，E），而在$DN = 10$时为（C，F）。由于状态对（B，E）在P_1中不属于相同的块，因此状态A和D是不等价的。C和F的后续状态在$DN = 00$时为（A，A），而在$DN = 10$时为（C，F），每一对都属于P_1中的一个块。由此，第二次状态划分的结果为：

$$P_2 = (A)\ (D)\ (B)\ (CF)\ (E)$$

C和F在P_2中的后继都属于P_2中相同的块，这意味着：

$$P_3 = P_2$$

由此得出结论C行和F行是等价的。将它们合并为一行并将图9-26所示流程表中所有的F替换为C，从而得到如图9-27所示的流程表。

当前状态	下一状态				输出
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	C	–	0
E	A	(E)	–	–	1

图9-27 图9-26b中有限状态机化简的第一步

接下来我们可以试着利用存在的无关项来合并流程表中的某些行。表中唯一能跟其他行合并的行是C，它可以和A或E中的一个合并，但并非两者都可以。C和A合并意味着新的状态在输入条件00下稳定时必须产生输出$z = 0$，而在输入条件10下稳定时必须产生输出$z = 1$。只有通过用米利模型才能达到这样的结果。另外一个选择是合并C和E，则新状态在输入条件$DN = 01$和10的情况下稳定，并产生输出1。这个结果可以通过摩尔模型达到。将C和E合并为一个状态C，同时将表中所有填E的项改变为填C，从而得到如图9-12所示的流程表。可以看出当C和E合并后，新行C必须包含原行C和E中所有的特性。当$DN = 00$时，这两行都将A指定为它们的下一个状态。原E

行在输入$DN = 01$的情况下稳定，因此合并后的新行（C行）在同样的输入下也必须稳定。同样地，原C行在$DN = 10$下为稳定状态，这也要反映在合并后的新行中。所以，新行中的下一个状态项分别为A，Ⓒ，Ⓒ，分别对应输入取值为00，01和10的情况。 ■

合并步骤

在例9.7中，决定哪些行应该合并变得很容易，因为唯一的可能性是将C行合并到A行或E行中。我们选择将C行和E行合并，因为这样做可以用摩尔模型来实现，从而得到实现输出z的更简单的表达式。

一般情况下，在规模大的流程表中，通常有更多的行合并机会。在这样的情况下，有必要建立一个更结构化的程序来完成行合并的选择。利用状态相容性（compatibility of states）的概念可以定义有用的合并步骤。

定义9.1 如果在任何输入条件下，两个状态S_i和S_j（表中的两行）都不会出现状态冲突，则称这两个状态是相容的。因此，对每组输入取值，下列条件之一必须为真：

1) 状态S_i和S_j都有相同的后续。

2) 状态S_i和S_j都是稳定的。

3) 状态S_i和S_j的后继之一为无关项或者两者均为无关项。

此外，若状态S_i和S_j有指定的输出，则这两个输出必须相同。

考虑如图9-28所示的基本流程表。让我们来检查不同状态的相容性，假设我们愿意保留该FSM的摩尔型规范的输出。状态A只与状态H相容，状态B则与状态F 和状态G 相容，而状态C不与其他任何状态相容，状态D则与状态E相容；相容的状态还有F 与G，G与H。换句话说，存在下列相容的状态对：（A，H），（B，F），（B，G），（D，E），（F，G）和（G，H）。各状态间的相容性关系可以方便地由如下合并图（merger diagram）的形式来表示：

- 合并图中的每个标名的点对应于流程表中同名的行。
- 画一条线将任意两个对应于相容状态的点连接起来。

从合并图中我们可以选择最佳可能的合并，从而导出简化的流程表。

图9-29所示是对应于图9-28基本流程表的合并图。该图指出行A可以与H合并，但必须是在H没有与G合并的前提下，因为在A和G之间并没有线连接。行B可以与行F 和G合并，而F也可以与G合并，由此B，F 和G是两两相容的。对任何行集合，若其中任意行对之间都是两两相容的话，则该集合中的所有行就都可以被合并成为一个状态。因此，状态B，F 和G可以被合并为一个状态，但必须在状态G没有与H合并的前提下。状态C不能跟任何其他的状态合并。状态D和E可以合并。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	Ⓐ	H	B	–	0
B	F	–	Ⓑ	C	0
C	–	H	–	Ⓒ	1
D	A	Ⓓ	–	E	1
E	–	D	G	Ⓔ	1
F	Ⓕ	D	–	–	0
G	F	–	Ⓖ	–	0
H	–	Ⓗ	–	E	0

图9-28 基本流程表

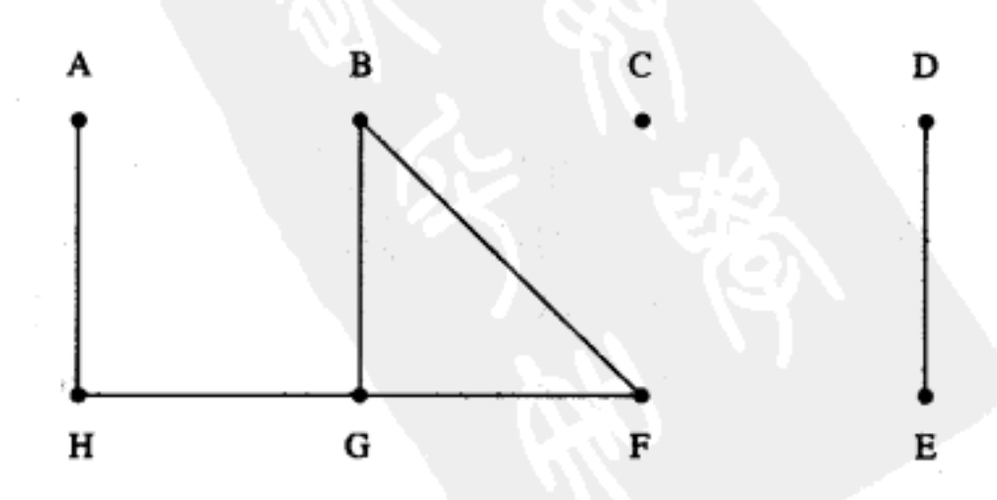


图9-29 图9-28中流程表的合并图（保持摩尔模型规范）

将状态合并使流程表包含的状态数尽可能地少是一个精明的策略。本例中，最好的方案是将相容状态对(A，H)，(B，F，G)和(D，E)合并，从而得到如图9-30所示的简化流程表。当通过合并两行或多行而创建了新行后，新行中的所有指定项必须覆盖（包含）合并前原行中的全部指定项。合并A和H创建了新的行A，新行A代替原来的两行时，要求新行A在w_2w_1 = 00和01的情况下都要保持稳定，因为原来的A在w_2w_1= 00的情况下稳定，而H则在w_2w_1 = 01的情况下保持稳定。同时，还必须指定行B在w_2w_1 = 10情况下的下一个状态，和行E在w_2w_1 = 11情况下的下一个状态。因为在状态E和D合并后，原E融入新状态D，所以在这之后，新行A必须指定的下一个状态项为：Ⓐ，Ⓐ，B和D，分别对应输入取值为00，01，10和11的情况。使用新行B替换了原来的行B，F和G时，必须使得新行B在w_2w_1 = 00和10的情况下保持稳定。在输入为w_2w_1 = 01的情况下，下一个状态项必须指定为D，以满足原来状态F的要求。而按照原来的状态B的规定，在w_2w_1 = 11的情况下新行B的下一个状态项必须为C。请注意，原来的状态G并没有指定在 w_2w_1 =01和11情况下的状态转移要求，因为它对应的下一个状态项都是无关项。行C则基本不做改变，除了其对应的在w_2w_1 = 01情况下的下一个状态项的名字由H改为A之外，其他项保持原样不变。类似地，原来的行D和E由新行D代替。请注意，合并化简后的流程表（见图9-30所示）仍为摩尔模型。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	Ⓐ	Ⓐ	B	D	0
B	Ⓑ	D	Ⓑ	C	0
C	–	A	–	Ⓒ	1
D	A	Ⓓ	B	Ⓓ	1

图9-30 图9-28流程表经化简后的摩尔型流程表

到目前为止，我们只考虑了合并那些仍允许我们保持如图9-28所示FSM的摩尔型规范的行。若我们想要改变到米利模型，则存在另一些合并的可能性。图9-31给出了图9-28所示FSM的完全合并图。其中细线连接的相容状态可被合并为有摩尔型输出的新状态，与图9-29所示的合并图一致。粗线连接的状态，则只有当使用米利型输出时才能被合并。

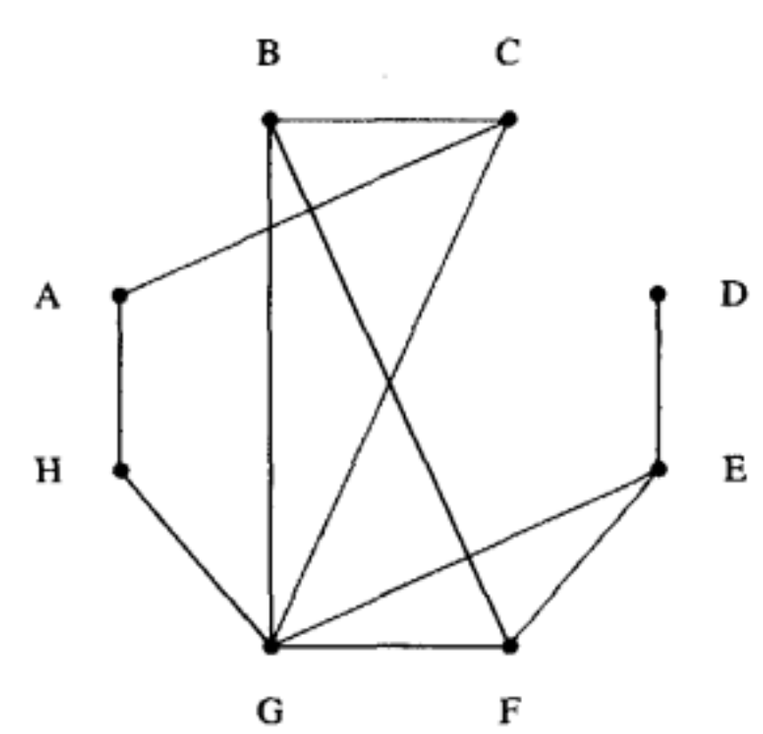


图9-31 图9-28的完全合并图

在本例中，转到米利模型未必能得到更简化的电路。尽管存在着几种合并的可能，但由所有这些合并所简化的流程表还至少需要4个状态。这样的简化结果一点也不比如图9-30所示的解决方案好。例如，有一种合并方案选择这样的划分：(A，H) (B，C，G) (D，E) (F)。另一种选择这样的划分：(A，C) (B，F) (D，E) (G，H)。下面不再逐一分析这些可能性，我们将在例9.9中讨论指定米利型输出时所涉及的问题。

状态化简步骤

从基本流程表起，对流程表逐步进行化简所需的步骤总结如下：

1) 用划分方法消除基本流程表中的等价状态。

2) 为得到的流程表建立合并图。

3) 选择一组可被合并的相容状态子集，在覆盖所有状态的前提下，试着尽可能减少所需子集的数量。每个状态必须只能被所选子集之一包含。

4) 合并所选子集的行，从而得到简化后的流程表。

5) 重复步骤2至4，看是否还能进一步化简。

为了进行状态合并，选择最佳的相容状态子集可能是一个非常复杂的任务，因为对于许多大型FSM而言，存在大量应考察的可能性。关于如何应付这个复杂的问题，测试-查错方案应该是解决这个问题的合理方法。

例9.8 考虑图9-32所示的初始流程表。为了应用划分步骤，我们先确定状态对(A，G)、(B，L)和(H，K) 是可能的等价行，因为每对中的两行其输出均相同，且它们的无关项位于同一列中。从这一方面看，剩下的行都各不相同。因此，得到第一次划分结果如下：

$$P_1 = (AG)\ (BL)\ (C)\ (D)\ (E)\ (F)\ (HK)\ (J)$$

现在(A，G)在$w_2w_1 = 00$的情况下的后续为(A，G)，在$w_2w_1 = 01$的情况下为(F，B)，在$w_2w_1 = 10$的情况下为(C，J)。由于F 和B 不属于同一个块，C和J也不属于同一个块，所以A和G是不等价的。(B，L)的后继分别为(A，A)、(B，L)和(H，K)，它们都属于同一块。(H，K)的后继为(L，B)、(E，E)和(H，K)，它们也都属于同一块。因此，得到第二次划分结果如下：

$$P_2 = (A)\ (G)\ (BL)\ (C)\ (D)\ (E)\ (F)\ (HK)\ (J)$$

重复进行后继测试表明(B，L)和(H，K)的后继仍属于同一块，因此：

$$P_3 = P_2$$

合并行B和L并以B命名得到的新行，合并行H和K并以H命名得到的新行，从而得到图9-33中的流程表。

当前状态	下一状态				输出
	$w_2w_1 = 00$	01	10	11	z
A	A	F	C	—	0
B	A	B	—	H	1
C	G	—	C	D	0
D	—	F	—	D	1
E	G	—	E	D	1
F	—	F	—	K	0
G	G	B	J	—	0
H	—	L	E	H	1
J	G	—	J	—	0
K	—	B	E	K	1
L	A	L	—	K	1

图9-32 例9.8的流程表

当前状态	下一状态				输出
	$w_2w_1 = 00$	01	10	11	z
A	A	F	C	—	0
B	A	B	—	H	1
C	G	—	C	D	0
D	—	F	—	D	1
E	G	—	E	D	1
F	—	F	—	H	0
G	G	B	J	—	0
H	—	B	E	H	1
J	G	—	J	—	0

图9-33 使用划分步骤得到的简化后的流程表

图9-34给出了该流程表的合并图。从该图可以看出应该把行B和H合并成一行，将合并后的新行标记为B。合并图同样表明应该将行D和E合并，得到的新行称为行D。余下的行可有多种合并的选择。可以把行A和F合并，但如果这样做，就不能将行F和J合并了。也可以把行C和J合并，或者把行G和J合并。我们将选择把行A和F合并成为新的A行，合并行G和J成为新的G行。所做的合并选择在图中用阴影表示。得到的流程表如图9-35所示。为了检查该表是否还可以进一步合并，我们可以再建立一个合并图，见图9-36所示。由该图可以很明显看出行C和G还可以合并，令合并后的新行称为C行。这样就得到再也不能进一步化简的流程表，见图9-37所示。

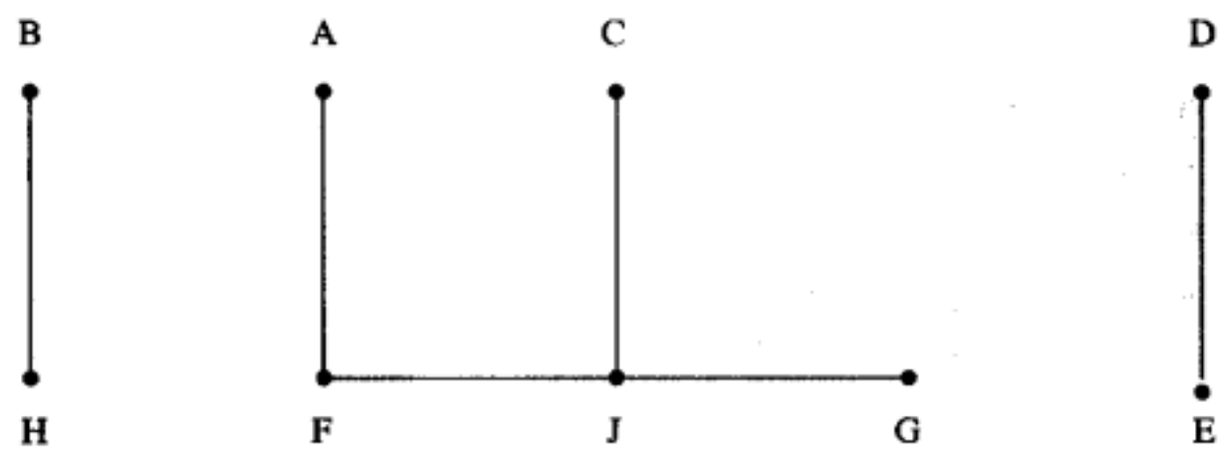


图9-34 图9-33的合并图

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	A	C	B	0
B	A	B	D	B	1
C	G	—	C	D	0
D	G	A	D	D	1
G	G	B	G	—	0

图9-35 由图9-34中的合并图得到的简化结果

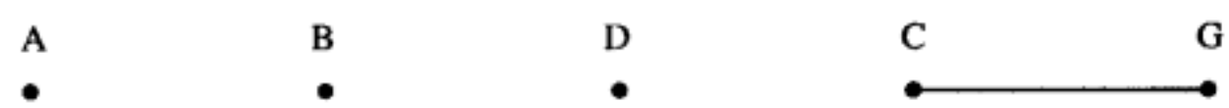


图9-36 图9-35的合并图

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	A	C	B	0
B	A	B	D	B	1
C	C	B	C	D	0
D	C	A	D	D	1

图9-37 例9.8的简化流程表

例9.9 考虑如图9-38所示的流程表。对该表应用划分得到：

$$P_1 = (AFK)\ (BJ\ )(CG)\ (D)\ (E)\ (H)$$
$$P_2 = (A)\ (FK)\ (BJ)\ (C)\ (G)\ (D)\ (E)\ (H)$$
$$P_3 = P_2$$

把B和J合并为新状态B，把F和K合并为新状态F，得到如图9-39所示的流程表。

图9-40a给出了该流程表的一个合并图，指出了（那些在保留FSM摩尔模型特征前提下的）合并可能性。在该例子中，B和F 可以合并，C和H也可以合并，从而得到一个包含六行的流程表。

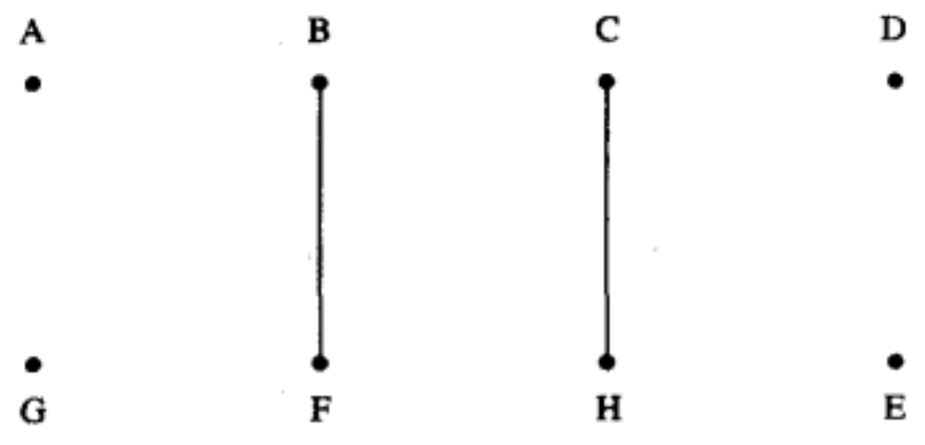
接下来考虑：如果我们想改为米利模型的话，会有哪些合并的机会。当从摩尔模型改为米利模型时，在米利模型中的稳定状态必须产生与它在摩尔模型中同样的输出。此外，很重要的一点是必须确保在米利模型中的状态转移不会造成输出信号出现意外的毛刺。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	G	E	—	0
B	K	—	B	D	0
C	F	C	—	H	1
D	—	C	E	D	0
E	A	—	E	D	1
F	F	C	J	—	0
G	K	G	—	D	1
H	—	—	E	H	1
J	F	—	J	D	0
K	K	C	B	—	0

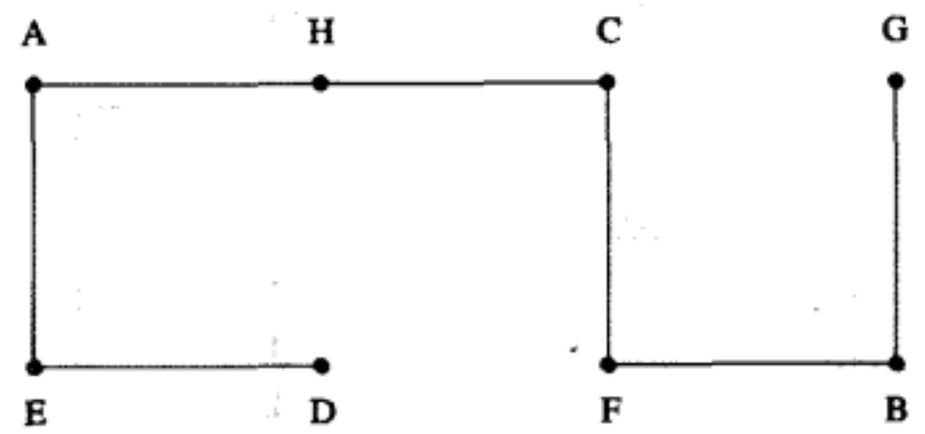
图9-38 例9.9的流程表

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	G	E	—	0
B	F	—	B	D	0
C	F	C	—	H	1
D	—	C	E	D	0
E	A	—	E	D	1
F	F	C	B	—	0
G	F	G	—	D	1
H	—	—	E	H	1

图9-39 经划分步骤后的化简结果



a) 保留摩尔模型



b) 完成的合并图

图9-40 图9-39的合并图

图9-41说明了如何将图9-39所示的FSM表示为米利模型的形式，其中表示下一个状态的那些项并没有做任何改变。在图9-41中，每个稳定状态的输出必须与摩尔模型流程表中相应行的输出一样。例如，当状态A稳定在输入w_2w_1 = 00条件下时，输出z = 0。同样，当状态B，D和F分别稳定在相应输入w_2w_1 = 10，11和00条件下时，产生同样的输出z =0。类似地，当状态C，E，G和H分别稳定在输入w_2w_1 =01，10，01和11的情况下时，产生输出z = 1。若从一个稳定状态到另一个稳定状态的转移要求输出从0变为1，或从1变到0时，变化发生的准确时间并不是很重要的，这一点我们在9.1节讨论图9-3时已经解释过。举个例子：假设FSM在输入w_2w_1 = 00的情况下稳定于状态A，并产生输出z =0。若此时输入改变为w_2w_1 = 01，则FSM必须转移到状态G，并产生输出z =1。由于电路进入状态G之前就变为1并不重要，因此行A中对应该转移的输出项可以被指定为无关项，所以我们就将该输出项设置为无关项。稳定状态A也有可能转移到

状态E，由于此时z从0变为1，所以状态E也允许指定另一个无关输出项。B行中出现了一种不同的情况。假设电路在输入$w_2w_1=10$的情况下稳定于状态B。若输入变为11，则必然导致电路转移到稳定状态D，且输出z必须在状态转移过程中保持为0。于是行B的输出项，在输入$w_2w_1=11$的条件下被指定为0。若不将该输出项指定为0而指定为无关项，则在电路的实现中有可能将该项当成1来看待。这样一来，若输入从10改变为11，则电路会从状态B转移到D，而此时输出z中将产生一个0→1→0的毛刺。我们可以用同样的推理来确定图9-41中其他的输出项。

由图9-41可以得出如图9-40b所示的合并图。图9-40b中用粗线连接的行只能通过指定输出项为米利型才能进行合并，而黑线连接的行是那些即使在其输出为摩尔型的情况下也可以合并的行，它们都对应图9-40a中的合并图。选择一组相容状态的子集：(A，H)，(B，G)，(C，F)和(D，E)，该FSM可以仅用4个状态来表现。将状态A和H合并为新的状态A，状态B和G合并为新的状态B，状态C和F合并为新的状态C，状态D和E合并为新的状态D，由此我们得到如图9-42所示的简化流程表。表中每一项都满足被合并的那些相应行指定的要求。

当前状态	下一状态				输出 z			
	w_2w_1 = 00	01	10	11	00	01	10	11
A	A	G	E	–	0	–	–	–
B	F	–	B	D	0	–	0	0
C	F	C	–	H	–	1	–	1
D	–	C	E	D	–	–	–	0
E	A	–	E	D	–	–	1	–
F	F	C	B	–	0	–	0	–
G	F	G	–	D	–	1	–	–
H	–	–	E	H	–	–	1	1

图9-41 用米利模型形式表示图9-39的FSM

当前状态	下一状态				输出 z			
	w_2w_1 = 00	01	10	11	00	01	10	11
A	A	B	D	A	0	–	1	1
B	C	B	B	D	0	1	0	0
C	C	C	B	A	0	1	0	1
D	A	C	D	D	–	–	1	0

图9-42 例9.9的简化后的流程表 ■

例9.10 再以如图9-43所示的流程表为例来考虑，由划分步骤可得到：

$$P_1=(AF)\ (BEG)\ (C)\ (D)\ (H)$$
$$P_2=(AF)\ (BE)\ (G)\ (C)\ (D)\ (H)$$
$$P_3=P_2$$

分别用状态A和B来代替状态F 和E，则得到如图9-44所示的流程表，相应的合并图见图9-45所示。很明显，状态A，B和C可以合并为新的状态A。同样，状态D，G和H也可以合并为新的状态D。从而得到如图9-46所示的只有两行的简化流程表。在本例中，我们用的又是米利模型，因为被合并的稳定状态D和H产生输出为$z=1$，而状态G产生的输出却为$z=0$。

当前状态	下一状态				输出 z
	w_2w_1 = 00	01	10	11	
A	A	B	C	–	0
B	F	B	–	H	0
C	F	–	C	H	0
D	D	G	C	–	1
E	A	E	–	H	0
F	F	E	C	–	0
G	D	G	–	H	0
H	–	G	C	H	1

图9-43 例9.10的流程表

当前状态	下一状态				输出 z
	w_2w_1 = 00	01	10	11	
A	A	B	C	–	0
B	A	B	–	H	0
C	A	–	C	H	0
D	D	G	C	–	1
G	D	G	–	H	0
H	–	G	C	H	1

图9-44 经划分步骤化简后的流程表

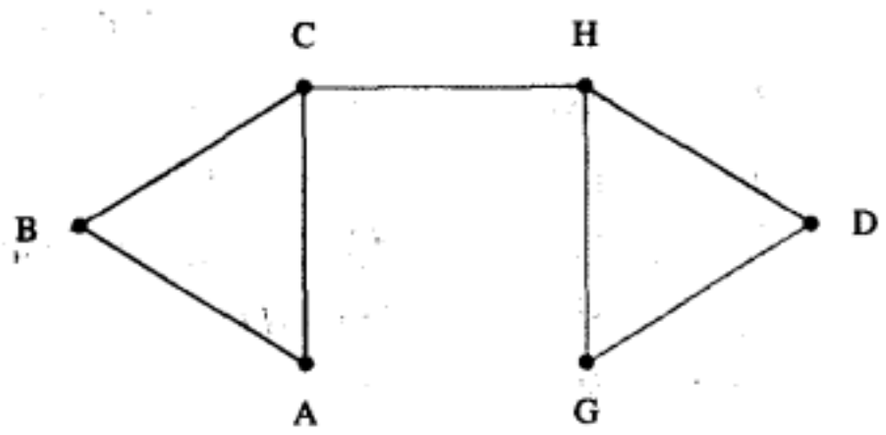


图9-45 图9-44的合并图

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z 00	01	10	11
A	(A)	(A)	(A)	D	0	0	0	—
D	(D)	(D)	A	(D)	1	0	—	1

图9-46 例9.10化简后的流程表 ■

9.5 状态分配

第9.3节的几个例子充分说明了异步FSM的状态分配是一件相当复杂的任务。改变状态变量值所需要的时间取决于电路的传播延迟。因此确保两个或两个以上变量的值在同一个时刻发生改变成为不可能做到的事情。为了实现电路可靠地运行，多个状态变量应该的改变必须加以控制，同一时刻只允许一个变量发生改变。这是通过电路设计来完成的，必须确保所设计的电路在发生状态转移时，只有一个（位）状态变量发生改变。

FSM中的各个状态都以位（比特）序列的形式来编码，分别表示状态变量的不同取值。两个位串序列之间，位相同而值不同的位数之总和称为汉明距离（Hamming distance）。举例说明：位串序列0110和0100的汉明距离为1，而0110和1101的汉明距离为3。利用该术语，理想的状态分配是使得从一个稳定状态到另一个稳定状态的每个转移的汉明距离都为1。若理想的状态分配不可能实现时，则必须找到利用未分配的状态或经过不稳定的中间过渡状态转移的方法。有时候，还有必要增加状态变量的数目以提供需要的灵活性。

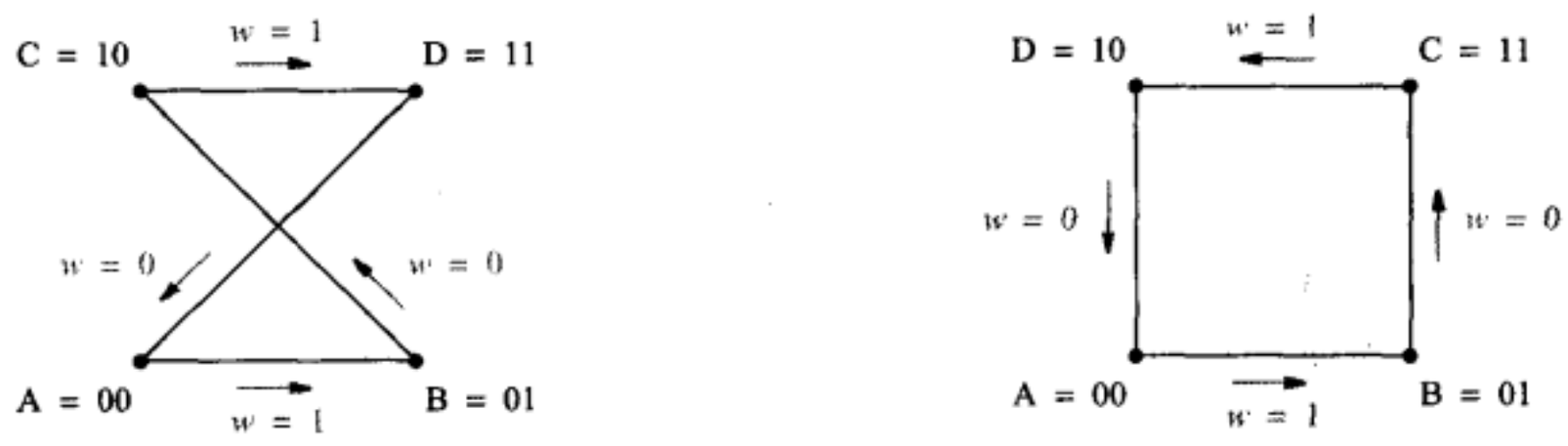
例9.11 考虑如图9-13所示的奇偶校验发生器的FSM。图9-14给出了两种可能的状态分配。如图9-13b流程表所示的状态转移也可以通过状态转移图的形式描述，见图9-47所示。流程表中的每一行可以表示为状态转移图中的一个相应的点。表示流程表中的4行需要用4个点，并且把这4个点置于正方形的四个角，每个角有一个相关的编码表示状态变量y_2y_1的一个取值。这些在图中的编码，其中y_2y1 = 00位于正方形的左下角，其余的依次类推位于正方形相应的角，对应于4.8.1节中介绍的二维立方体坐标，见图4-33所示。

图9-47a展示若采用图9-14a所示的状态分配，则会发生什么。换言之，假设状态分配为$A = 00$，$B = 01$，$C = 10$和$D = 11$的情况下。若$w = 1$，可能发生从状态A到状态B的转移，只引起y_1的改变。若$w = 1$时，还可能发生从状态C到状态D的转移，也只引起y_1的变化。然而，由$w = 0$引起的从状态B到状态C的转移，会引起y_2和y_1同时发生变化。同样，若$w = 0$，还可以引起从D到A的状态转移，也会引起两个状态变量同时发生变化。两个变量同时改变的状态转移，在图中对应于对角线。

图9-47b展示如图9-14b所示的状态分配方案的效果。该分配方案将C和D的取值作一下交换。在此情况下，所有4个状态转移都沿着二维立方体的直角边进行，并且这些转移都只引起单个状态变量的改变。这正是理想的状态分配。 ■

例9.12 图9-21a给出了仲裁器FSM的流程表。该FSM中使用的状态分配为$A = 00$，$B = 01$和$C = 10$，其状态转移图见图9-48a所示。在本例中，状态之间存在多种可能的转移。例如，状态A与B之间存在两种可能的转移：当$r_2r_1 = 00$的时候，从状态B到A的转移，以及当$r_2r_1 = 01$的时候，从状态A到B的转移。本例同样有一条对角线路径，对应于B和C之间的状态转移，而这种情况是应该避免的。可行的解决方案是引入第4个状态D，如图9-48b所示。此时B和C

之间的状态转移可以通过经由不稳定状态D来实现。当$r_2 r_1 = 10$时，电路将先从状态B转移到状态D，再从状态D转移到状态C，而不是直接从状态B转移到状态C。



a) 对应于图9-14a　　b) 对应于图9-14b

图9-47　图9-13中的状态转移

a) 图9-21a中的转移　　b) 使用额外的状态D

图9-48　图9-21中的仲裁器FSM的状态转移

若想利用如图9-48b所示的状态分配，则需要将流程表改为如图9-49所示。其中状态D对任何输入取值都是不稳定的，而且若输入为$r_2 r_1 = 00$或01，则无法进入状态D，因此流程表中的这些项都被设置成待定。同时注意到我们将状态D的输出指定为$g_2 g_1 = 10$，而不是让其输出为待定。若发生从稳定状态经由不稳定状态到达另一个稳定状态的转移过程时，该不稳定状态的输出必须与该状态转移所涉及的两个稳定状态之一的输出相同，以确保在经过这个不稳定状态的时候，不会产生错误的输出。

当前状态	下一状态				输出 g_2g_1
	$r_2r_1 = 00$	01	10	11	
A	(A)	B	C	–	00
B	A	(B)	D	(B)	01
C	A	D	(C)	(C)	10
D	–	B	C	–	10

图9-49　基于图9-48b中的状态转移改进后的流程表

将该流程表与图9-21b中的那个激励表相比较非常有意思。在那个激励表中也添加了附加状态D。在图9-21b中，因外部输入的变化而引起的状态转移有可能出现状态变量中两位同时改变的情况，由于电路竞争现象，这可能会在输出中引起毛刺干扰，所以需要添加状态D，且指定一些必要的状态转移来消除这种毛刺干扰。在图9-49中，状态D则被用于产生有次序的状态转移，从而可防止任何由竞争引起的毛刺。■

9.5.1 转移图

用来表示流程表中所指定的状态转移过程的图称之为*转移图*（transition diagram）。在某些书中这样的图也被称为*状态邻接图*（state-adjacency diagram）。这些图为我们寻找合适的状态分配方案提供了便利。

若状态转移图中没有出现任何对角线路径，则该转移图中各状态对应的编码将是一个合

理的状态分配。解释状态分配要求的一般方法是必须能够把状态转移图嵌入到一个k维的立方体之上，因为立方体相邻顶点之间的转移的汉明距离都是1。理想情况下，一个n变量FSM的状态转移图可以表示为一个n维的立方体，这正如图9-47b和图9-48b介绍的例子所示。若这样做不可能，则必须在电路中引入附加的状态变量，我们将在稍后的例子中介绍这种情况。

图9-47和图9-48中的状态转移图呈现了与给定FSM状态转移有关的所有信息。若FSM的规模较大，用这样的图来表示状态转移将显得杂乱无章，此时还可以用另一种更简单的表示形式，我们将在下面介绍。

状态转移图必须能表示对应于输入变量各种取值下的状态转移。转移的方向（例如从状态A到B或从状态B到A）并不重要，因为我们要确保的只是所有转移的汉明距离都必须为1。状态转移图必须表示所有可能的经过不稳定状态并进入稳定状态的每个转移所产生的效果。对于流程表中给定的某一行，在不同的输入取值下可能会有两个或两个以上的稳定状态项。在状态转移图中，使用标记把进入稳定状态的转移明显地标识出来是十分有用的。为了给每个稳定状态项设置不同的标记，我们将用数字1，2，3，…来表示这些项。因此，如果状态A在两个输入状态下都进入稳定状态，我们将把状态A在一种输入取值下标记为1，而在另一种输入取值下标记为2。

图9-50展示了把图9-21a中的流程表重新进行标记后的流程表。我们随意地选择了将状态Ⓐ标记为1，而将状态Ⓑ的两次出现分别标记为2和3，将状态Ⓒ的两次出现分别标记为4和5。所有在下一个状态列中的项都用此形式来做标记。改用标记识别的状态转移图见图9-51a所示。图9-48a也给出了同样的信息。实际上，图9-48a中的转移图包含了更多的信息，因为其中的箭头指出了每个转移的方向。还请注意，转移图中的每条边也标记上了转移所对应的输入取值r_2r_1，然而图9-51a中的各条边是用上面讲述过的数字稳态标签标识的。

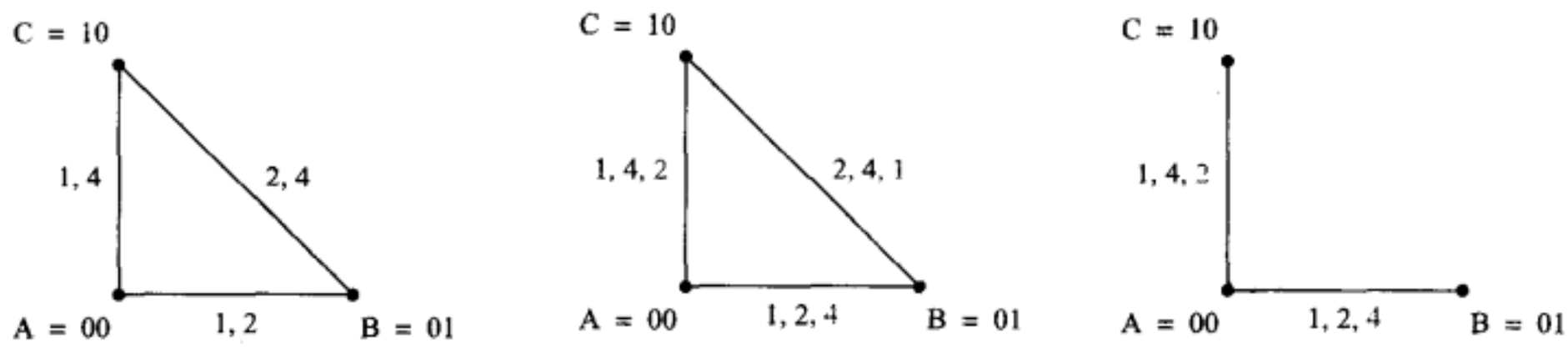
当前状态	下一状态				输出
	r_2r_1 = 00	01	10	11	g_2g_1
A	1	2	4	—	00
B	1	2	4	3	01
C	1	2	4	5	10

图9-50 图9-21a经重新标记后的流程表

图9-50指出可以从状态A或者状态C出发进入到稳定状态2（即状态B的两种稳态之一）。在图9-51a的转移图中连接相应顶点的路径上有一个对应的标记2，用来表示这种情况。从状态分配的观点来看，困难在于从状态C到状态B的路径是对角线的。在例9.12中这个问题通过引入一个新状态D而得以解决。通过更细致地检查图9-50中的流程表，我们可以看到，若令状态C到状态B的转移经由状态A实现的话，则可以实现仲裁器FSM所要求的功能行为。也就是说，若电路稳定于状态C，当输入取值为r_2r_1 = 01时，导致状态转移到A，而一旦到达状态A，电路立刻转移到状态B。如图9-51a所示，我们可以通过将2标记到连接状态C和A的路径边上来指出使用这条路径的可能性。

类似的情况也发生在B到C的状态转移中，该转移标记为4。若在输入取值为r_2r_1 = 10的情况下，使电路先从状态B转移到A，且立刻转移到C，也可以实现B到C的状态转移。在图9-51a中连接B和A的一条边上添加一个标记4，同样也可以用来表明这个转移。

只要两个状态在重新标记过的流程表中有着相同的未画圈的标记，那么这两个状态之间就存在着状态转移的可能路径。在图9-50中，若r_2r_1 = 00时，使用标记1，则还存在第三条这样的可选路径。但这条可选择的路径没有什么用，因为由图9-51a所示的状态分配，从状态B或C到A的状态转移只需要一个状态变量发生变化。因此，在这种输入取值条件下，进行状态B和C之间的状态转移将没有什么好处。



a) 图9-50中的转移　　b) 完整的转移图　　c) 选定的转移图

图9-51　图9-50的转移图

为了在状态转移图中呈现存在多个可供选择的路径，我们将在图中用不同字体的数字标记对应的转移。由此，完整的转移图将以细体数字表示所有到达稳定状态的直接转移，而用黑体数字表示经由不稳定状态的间接转移。图9-51b呈现的是图9-21a流程表的完整转移图。

图9-51b中的转移图不能被嵌入到二维的立方体中，因为其中的一些转移需要对角线路径。图9-51b中*B*和*C*之间路径上标记的1可以不予以考虑，因为它表示的只是一条不必使用的可选择路径。但是，状态*B*和*C*之间标记为2和4的转移是必需的。该转移图描述了一条通过状态*A*的可选择的路径，边上标有2和4，因此可选择这条路径，从而图中对角线连接边可以被去掉。由此我们得到了如图9-51c所示的可以嵌入到二维立方体中的转移图。我们得出结论：状态分配*A*=00，*B*=01和*C*=10是合理可用的，但是流程表必须加以修改，以指定状态之间的转移通过哪条可选路径。修改后的流程表与前面采用特别方法来设计的流程表（见图9-23a所示）完全一致。

本例最后需要提醒注意的是：可选路径对FSM所产生输出的影响。若$r_2 r_1 = 01$，则电路从稳定状态*C*经过不稳定状态*A*到达*B*的转移产生的输出为：$g_2 g_1 = 10 \rightarrow 00 \rightarrow 01$，而并非图9-21a中所指定的$g_2 g_1 = 10 \rightarrow 01$。对仲裁器FSM而言，这样做不会造成任何问题，在例9.6中我们曾经对类似问题做过解释。

转移图的推导步骤

转移图通过如下的途径从流程表推导得出：

1) 按前面所介绍的方法先推导出重新标记后的流程表。对于给定的输入取值，所有目的地为同一个稳定状态的转移都用相同的数字做标记。经过不稳定状态并最终到达稳定状态的转移也用与稳定状态项等价的数字做标记。

2) 用顶点来表示流程表的相应行。

3) 若两个顶点V_i和V_j，在重新标记流程表的任意列中有相同的数字，则用一条边将它们连接起来。

4) 对于V_i和V_j拥有相同数字的每一列，用该数字对连接V_i和V_j的边做标记。我们将使用普通数字来标记直接转移到画圈（稳定）状态的转移，而对于流程表中V_i和V_j的下一个状态项为未画圈的项时，则用黑体数字来做标记。

请注意，在第1步中曾提到在重新标记的流程表中，在给定的输入取值下，经过不稳定状态到达稳定状态的转移，将以最后到达的稳定状态的等价数字做标记。举例说明如下：若想要根据图9-23a中的流程表推导出状态转移图，则该流程表必须被重新标记，以得到如图9-50所示的新的流程表。当$r_2 r_1 = 01$的时候，从稳定状态*A*到稳定状态*B*的转移被标记为2。从稳定状态*C*到不稳定状态*A*的转移也被标以同样的标签，因为这个转移最终还是转移到了稳定状态*B*。

9.5.2 未指定的下一个状态项的利用

流程表中的未指定项可以为寻找合理的状态分配提供某种程度的灵活性。下面的例子给

出了一种可能的方法，该例子还说明了推导状态转移图全过程的每个步骤。

例9.13 考虑图9-52a中的流程表。该FSM有7个稳定状态项，将这些项从1到7按顺序做标记，从而得到图9-52b。在本例中，状态1和2对应于状态A，3和4对应于状态B，5和6对应于状态C，而7对应于状态D。在$w_2w_1=00$的列中有一个从C到A的状态转移，标记为1；一个从D到B的状态转移，标记为3；因为在这两个转移中1和3都是后续稳定状态。同样，在$w_2w_1=11$的列中，分别有从B到C和D到A的状态转移，分别标记为6和2。在$w_2w_1=01$的列中，有从A到B的状态转移，标记为4。状态C在此输入取值情况下保持稳定，标记为5。表中没有指定的转移可进入该稳定状态。仅当C状态稳定于输入$w_2w_1=11$情况（在表中标记为6的状态），而且当w_2w_1接着转变为01的情况下，才能达到该状态。请注意，当输入取值从11改变到01的时候FSM仍然保持在稳定状态C，反之亦然。$w_2w_1=10$的列说明了如何处理不稳定状态。一旦FSM进入状态C，它将继续转移到稳定状态D，D被标记为7。因此，7就被用作标记从状态A到C再到D的整个转移过程。

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z_2z_1
A	A	B	C	A	00
B	B	B	D	C	01
C	A	C	D	C	10
D	B	—	D	A	11

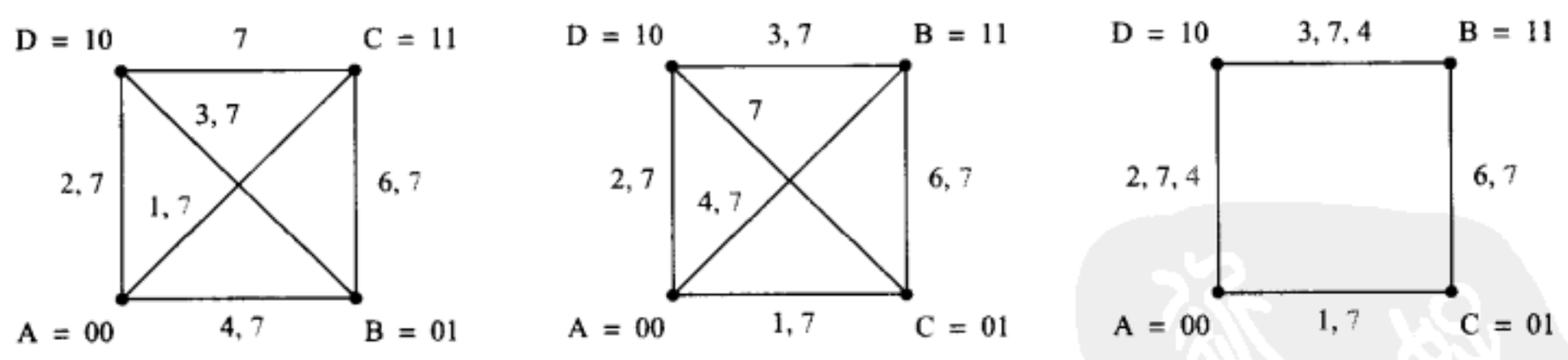
a) 流程表

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z_2z_1
A	1	4	7	2	00
B	3	4	7	6	01
C	1	5	7	6	10
D	3	—	7	2	11

b) 重新标记的流程表

图9-52 例9.13的流程表

以行A，B，C和D作为正方形的4个顶点，可以先试着画出状态转移图的初稿，见图9-53a所示。该转移图展示了所有状态对之间的转移，看起来不可能找到一种状态分配可以使得所有转移的汉明距离为1。如果使用的状态分配为$A=00$，$B=01$，$C=11$和$D=10$，则状态A和C或者B和D之间的对角线转移要求两个状态变量同时发生改变。标记为7的从状态B到D的对角线路径是不需要的，因为存在另一条标记为7的从B到D的路径，中间经由状态A或者C。可惜的是，标记为1和3的对角线路径无法去除，因为对这些转移而言，无法找到别的可选路径。



a) 最初的转移图　　b) 第二个转移图　　c) 最终的转移图

图9-53 图9-52的转移图

为寻找更合适的状态分配，我们接着做第二次尝试，将B和C的状态编码交换一下，从而得到图9-53b所示的转移图。前面关于标记为7的可选路径的讨论已经告诉我们，从状态C到D的对角线路径可以取消。同时，标号为7的状态A与B之间的对角线路径也可以去掉。但A与B之间标记为4的对角线转移并没有别的路径可选，因此无法消除。仔细观察图9-52b所示的流程表，我们可以看到在$w_2w_1=01$的列中有一个无关项。可以通过将该无关项标记为4，来消除

这个对角线转移。在这种情况下，转移图将在A和D以及B和D之间的连接边上显示标号4。由此，状态A和B之间的对角线转移就可以取消，从而得到如图9-53c所示的转移图。该图可以嵌入到一个二维立方体中，这意味着状态分配$A = 00$，$B = 11$，$C = 01$和$D = 10$是可用的。

为使图9-53c所示的转移图可实际应用，必须将该FSM的流程表修改为如图9-54a所示的样子。图9-52a中的无关项现在明确地指定了到状态B的转移。根据图9-53c的修改，在输入取值$w_2 w_1 = 01$的情况下，从状态A到B的转移必须经过状态D，因此，必须修改第一行（状态A）中的对应项，以保证该状态转移的正确执行。同样，当$w_2 w_1 = 10$时，FSM必须转移到状态D。若电路刚好处于状态C中，则到状态D的转移必将经由状态A或B来进行。在图9-54a中，我们选择了经由状态B的路径。

当前状态	下一状态 $w_2w_1 = 00$	01	10	11	输出 z_2z_1 00	01	10	11
A	A	D	D	A	00	00	11	00
B	B	B	D	C	01	01	11	01
C	A	C	B	C	-0	10	1-	10
D	B	B	D	A	-1	0-	11	00

a) 修改后的流程表

	当前状态 y_2y_1	下一状态 $w_2w_1 = 00$	01	10	11	输出 00	01	10	11
		Y_2Y_1				z_2z_1			
A	00	00	10	10	00	00	00	11	00
B	11	11	11	10	01	01	01	11	01
C	01	00	01	11	01	-0	10	1-	10
D	10	11	11	10	00	-1	0-	11	00

b) 激励表

图9-54 图9-52a中FSM的实现

图9-52a所示的原始流程表是以摩尔模型的形式来定义的。图9-54a所示的改进后的流程表则要求使用米利模型，因为前面描述过的经由不稳定状态的转移必须产生正确的输出。首先考虑一下在输入取值$w_2 w_1 = 01$时从状态A开始的转移。处于稳定状态A时，电路必须产生输出$z_2 z_1 = 00$。进入稳定状态B后，输出必须变为01。问题是该转移要求短暂地经过状态D，而在摩尔模型中电路将产生临时的输出$z_2 z_1 = 11$。因此在输出z_2上将产生瞬间脉冲干扰，z_2的值将发生0→1→0的改变。为了避免出现不想要的瞬间脉冲干扰，状态D在此输入取值条件下产生的输出z_2必须为0，这要求使用如图9-54a所示的米利模型。我们观察到，当z_2在输入取值$w_2 w_1 = 01$下并处于状态D中必须为0时，z_1却可以为0或1，因为z_1正在从状态A中为0改变为状态B中为1。因此，可以将z_1在状态D时的值当做未指定，从而被看做无关项。同样的情况也发生在当输入取值$w_2 w_1 = 10$下，电路从状态C经由B转移到D的时候。此时输出必须从10改变为11，这意味着贯穿这次转移的过程z_2必须保持为1，包括经由状态B的短暂时刻，但此时摩尔模型的输出为01（这将会引起输出出现瞬间脉冲干扰）。

改进后的流程表和优化的状态分配使我们得到如图9-54b所示的激励表。从该激励表可以推导出表示下一个状态和输出的表达式，见9.3节中的几个例子。 ■

9.5.3 使用附加状态变量进行的状态分配

借助于图9-52a中的一个未指定的转移项，可以找到合适的状态分配，这正如9.5.2节所介绍的那样。在一般情况下，这样的机会不一定存在。对一个n行的流程表，往往找不到一个只使用$\log_2 n$个状态变量的无竞争的状态分配方案。此时，只能通过添加状态变量的方法来解决这个问题。存在三种添加状态变量的方法，我们用以下的例子来说明这些方法。

例9.14（借助于附加的不稳定状态） 考虑一下由图9-55a所示的流程表指定的FSM。该流程表被重新标记后变成图9-55b。图9-56a则描绘了与其对应的转移图。从转移图中可以看到，所有的两个顶点（行）之间都存在着转移。对现有顶点无论进行何种重新排列都无法将这样的转移图映射到一个二维立方体上。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z_2z_1
A	A	A	C	B	00
B	A	B	D	B	01
C	C	B	C	D	10
D	C	A	D	D	11

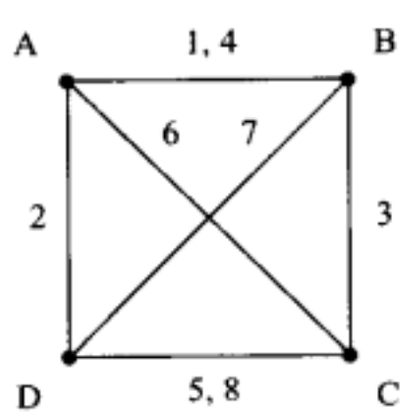
a) 流程表

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z_2z_1
A	1	2	6	4	00
B	1	3	7	4	01
C	5	3	6	8	10
D	5	2	7	8	11

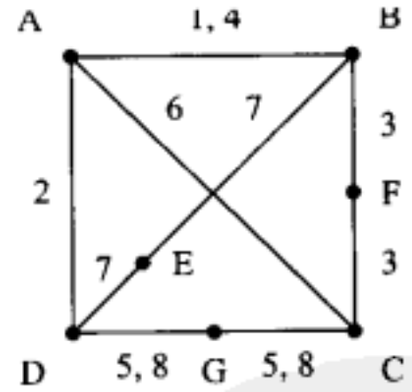
b) 重标记后的流程图

图9-55 例9.14中的FSM

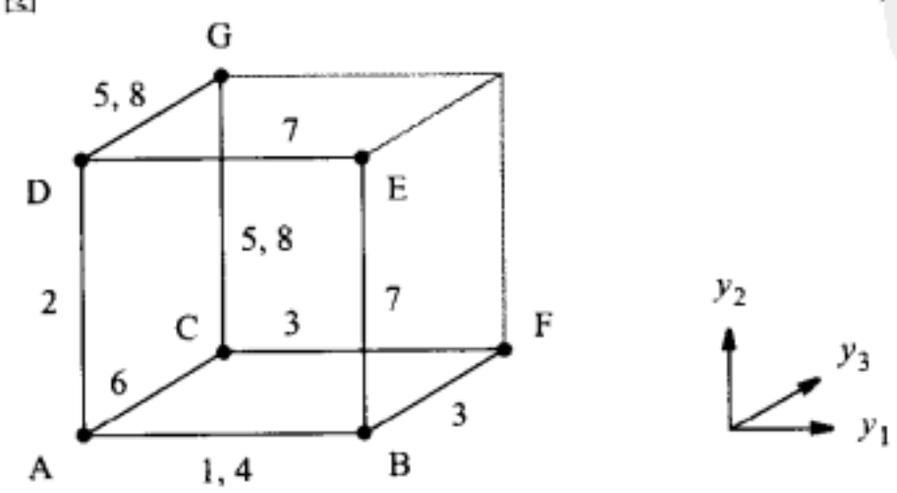
现在让我们添加一个新的状态变量，以便能找到一个方法将该转移图映射到一个三维立方体上。借助于3个状态变量，可以令对状态A的分配与对状态B，C和D的分配之间的汉明距离都为1。举例说明：令$A = 000$，$B = 001$，$C = 100$和$D = 010$。但是这么一来状态对（B，C），（B，D）和（C，D）的汉明距离就不可能为1了。解决这个问题的办法是在转移路径中添加一些顶点，如图9-56b所示。顶点E将B和D隔开，而顶点F和G则将路径（B，C）和（C，D）隔开。与原来的转移相关的标记现在则添加到了被隔开路径的每一段上。从而得到的结果转移图便可以嵌入到如图9-56c所示的一个三维立方体中，立方体中的细线条构成了我们想要的



a) 转移图



b) 改进的转移图



c) 嵌入的转移图

图9-56 图9-55的转移图

转移路径。于是在输入取值$w_2w_1 = 10$时，从B到D的状态转移是经由顶点E完成的（标记为7）。在输入取值$w_2w_1 = 01$时，从状态C到B的转移是经由点F完成的（标记为3）。在输入取值$w_2w_1 = 11$时，从状态C到D的转移是经由G完成的（标记为8）；在输入取值$w_2w_1 = 00$时，从状态D到C的转移也是经由G完成的（标记为5）。因此，改进后的流程表必然如图9-57所示。附加的三个状态是不稳定的，因为电路在任何输入取值下都不会保持稳定在这些状态中。电路只是从一个稳定状态转移到另一个稳定状态时短暂地经过这些状态。我们观察到E，F和G这些状态只需要对单个或两个输入取值作出反应完成状态转移即可，因此没有必要指定其他输入取值的情况，因为在功能正常的电路中这些无关的输入取值的情况是永远不会发生的。

图9-57a中的输出可以用米利模型来指定。关键的一点就是必须确保当电路经由不稳定的状态的时候能产生恰当的输出，以避免在输出信号中出现不想看到的毛刺干扰。

当前状态	下一状态 $w_2w_1 = 00$	01	10	11	输出 z_2z_1
A	A	A	C	B	00
B	A	B	E	B	01
C	C	F	C	G	10
D	G	A	D	D	11
E	—	—	D	—	-1
F	—	B	—	—	01
G	C	—	—	D	1-

a) 修改后的流程表

	当前状态 $y_3y_2y_1$	下一状态 $w_2w_1 = 00$	01	10	11	输出 z_2z_1
		$Y_3Y_2Y_1$				
A	000	000	000	100	001	00
B	001	000	001	011	001	01
C	100	100	101	100	110	10
D	010	110	000	010	010	11
E	011	—	—	010	—	-1
F	101	—	001	—	—	01
G	110	100	—	—	010	1-

b) 激励表

图9-57 例9.14改进后的表

若我们按照图9-56c的右边那样进行状态变量的分配，则从改进后的流程表就能推导出如图9-57b所示的激励表。根据该激励表，就能直接推导出下一个状态和输出的表达式。 ■

例9.15（等价状态对的使用） 给每个现有的状态引入一个等价的新状态，这是增加状态分配灵活性的另一种途径。由此，状态A可以用两个状态$A1$和$A2$来代替，而电路为两个新状态$A1$和$A2$产生的输出与原来为A产生的输出完全一样。同样，其他状态也可以用等价状态对来取代。图9-58显示了怎样借助于等价对替换的方法，用一个三维的立方体来为一个4行的流程表找到一个合适的状态分配方案。在该方案中，我们将4个等价状态对做出合理的安排，使得所有状态对之间存在的汉明距离最小（即都为1）。举例说明：状态对（$B1$，$B2$）相对与状态$A1$（或$A2$），$C2$和$D2$的汉明距离都为1。

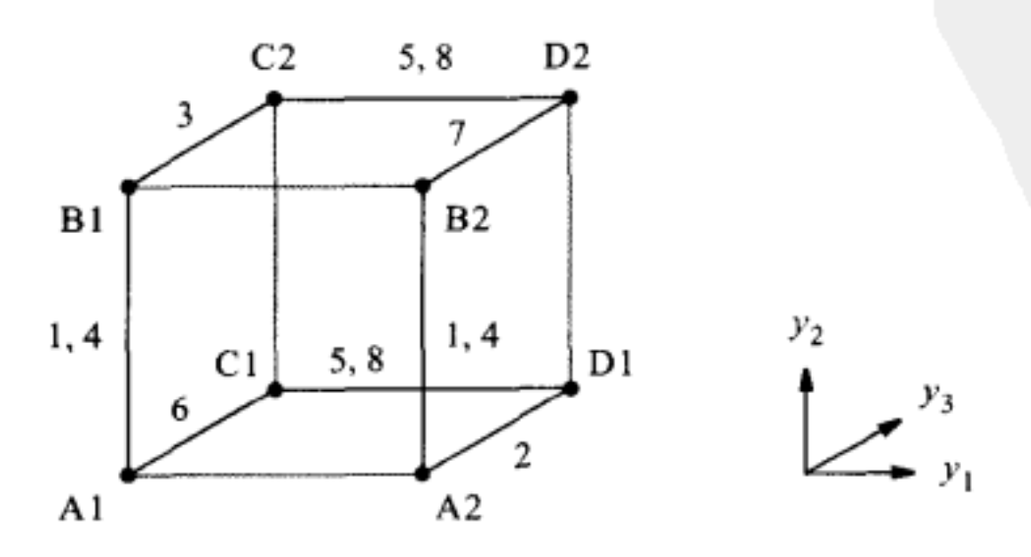


图9-58 每行用两个节点表示嵌入后的状态转移图

图9-56a所示的转移图可以嵌入如图9-58所示的三维立方体中。对于图9-56a所示的转移图中的每个顶点，在立方体中有相应的两个顶点可供选择，因此嵌入后的转移图就不再包含任何的对角线路径。使用这样的状态分配，就必须将图9-55a中的流程表改为如图9-59a所示的形式。原来流程表中的每个转移在修改后的表中转换为相应等价状态对之间的转移，所以表中的许多项要重新填写。若原来表中的某状态在某输入取值下是稳定的，则其对应等价状态对中的两个状态在相同的输入取值下也必然是稳定的。因此，当输入取值w_2w_1 = 00或01的时候，状态$A1$和$A2$保持稳定，在输入取值w_2w_1 = 01或11时，状态$B1$和$B2$保持稳定，其他以此类推。在任何给定的时刻，FSM可能处于表示同一原始状态的等价状态对（两个状态）中的某个状态。必须能从两个等价状态中的任意一个状态转移到其他状态去。例如，图9-55a规定当输入取值为w_2w_1 = 11的时候，FSM必须从状态A改变到状态B。在经过改进的流程表中，等价的转移为从状态$A1$到$B1$或者从状态$A2$到$B2$。如果FSM在状态A中稳定，且输入取值从00改为10，则电路将改变到状态C。在经过改进的流程表中的等价转移为从状态$A1$转移到$C1$；如果FSM此时刚好处于状态$A2$中，电路将先得转变为状态$A1$，然后再转移到$C1$。同理可以推导出图9-59a修改后的流程表中的其他各项。

输出是使用摩尔模型来指定的，因为不稳定状态只有那些涉及从等价状态对的一个成员到另一成员的转移，以及两个成员都产生了相同输出的转移才会产生。例如，在先前描述过的从状态A到C的转移中，如果开始状态为$A2$，则必须先转移到状态$A1$，再转移到状态$C1$。即使状态$A1$在w_2w_1 = 10的情况下不稳定也没关系，因为它的输出跟状态$A2$一样。因此，若原始的流程表是用摩尔模型来定义的，则改进后的流程表仍旧可以用摩尔模型来完成。

使用图9-58所示的状态变量分配，可以得到如图9-59b所示的激励表。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z_2z_1
A1	A1	A1	C1	B1	00
A2	A2	A2	A1	B2	00
B1	A1	B1	B2	B1	01
B2	A2	B2	D2	B2	01
C1	C1	C2	C1	D1	10
C2	C2	B1	C2	D2	11
D1	C1	A2	D1	D1	11
D2	C2	D1	D2	D2	11

a) 改进后的流程表

	当前状态 $y_3y_2y_1$	下一状态 w_2w_1 = 00	01	10	11	输出 z_2z_1
		$Y_3Y_2Y_1$				
A1	000	000	000	100	010	00
A2	001	001	001	000	011	00
B1	010	000	010	011	010	01
B2	011	001	011	111	011	01
C1	100	100	110	100	101	10
C2	110	110	010	110	111	10
D1	101	100	001	101	101	11
D2	111	110	101	111	111	11

b)激励表

图9-59 例9.15改进后的流程表和激励表

9.5.4 独热状态分配

前面所介绍的基于将流程表嵌入到立方体中的方法可以得到优化的状态分配，但是这个方法需要反复地尝试，对于大型FSM的状态分配而言无疑是一种很笨拙的方法。另一种方案是使用独热编码的方法，该方法比较直观，但实现成本较昂贵。若FSM的流程表中的每一行都被分配一个独热码，则通过与状态转移两端稳定状态之间的汉明距离都为1的不稳定中间状态，便可以实现两个稳定状态之间无竞争的状态转移。例如，假设状态A分配的编码为0001而状态B为0010，那么从状态A到B的无竞争的状态转移可以经过一个不稳定状态0011来实现。同样，

如果状态C分配的编码为0100，那么，从状态A到C的转移可以经由不稳定状态0101来完成。

利用这个解决方案，图9-55a中的流程表可以被改进为如图9-60所示。给A，B，C和D四个状态分别分配一个独热码。还有必要添加6个不稳定状态（即从E到J），用以处理必要的转移，如图所示。这些不稳定状态是为特定的转移专门定做的，而对于不相关的其他输入取值，这些添加的不稳定状态将被视为无关项。

状态分配	当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z_2z_1
0001	A	A	A	E	F	00
0010	B	F	B	G	B	01
0100	C	C	H	C	I	10
1000	D	I	J	D	D	11
0101	E	—	—	C	—	-0
0011	F	A	—	—	B	0-
1010	G	—	—	D	—	-1
0110	H	—	B	—	—	01
1100	I	C	—	—	D	1-
1001	J	—	A	—	—	00

图9-60 使用独热编码的状态分配

表的输出可以用摩尔模型来指定。在有些场合，某个输出信号的值是否发生改变是无所谓的。例如，状态E是用来帮助从A到C的状态转移的。由于在状态A中，输出为z_2z_1 = 00，而在状态C中输出为z_2z_1 = 10，则当电路瞬间经过不稳定状态E时，z_2的值是否会发生变化并不是什么重要问题。

虽然实现起来比较直观，但是独热编码的实现成本却比较高，因为它需要n个状态变量来实现一个n行的流程表。在逻辑电路的设计中，设计者常面临权衡得失的难题：设计虽简单但实现成本却较高，这需要认真仔细地分析。

9.6 冒险

在异步时序电路中，信号中不存在意外的瞬间脉冲干扰十分重要。设计者必须了解瞬间脉冲干扰（即毛刺）所有可能的来源，且确保电路在状态转移中不会产生这类毛刺干扰。由电路的结构和电路中信号的传播延迟引起的瞬间脉冲干扰称为冒险。图9-61说明了两种冒险。

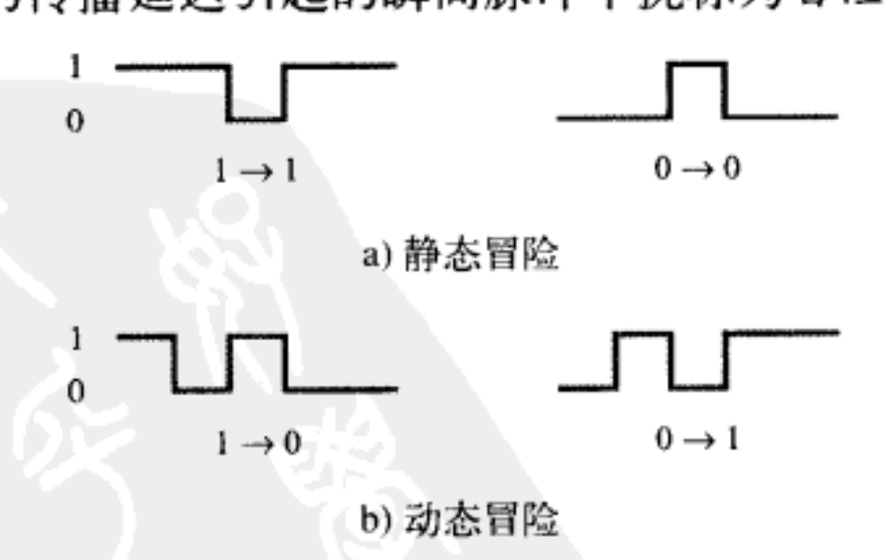


图9-61 冒险的定义

当输入变量的值发生改变时，某信号本该保持在某确定的逻辑值，但是该信号却在保持确定逻辑值期间，发生过瞬间的改变，这就是所谓的静态冒险。见图9-61a所示，有一种静态冒

险是：本该保持1的信号，却在瞬间内曾降为0，最后又恢复到1。另一种静态冒险则是：本该保持0信号，却在瞬间内曾升为1，最后又恢复到0，因此产生了一个瞬间脉冲干扰（即毛刺）。

另一种不同类型的冒险可能发生在当某信号本该从1改变到0或者从0改变到1的时候。若这样的变化在信号稳定到新的值之前包含了瞬间的振荡，见图9-61b所示，则称此时存在一个动态的冒险。

9.6.1 静态冒险

图9-62a描述了一个存在静态冒险的电路。假设当$f = 1$时，电路处于$x_1 = x_2 = x_3 = 1$的状态。现在令x_1从1改变到0。理想的情况下电路应该保持$f = 1$。但是若将门的传播延迟包括在考虑范围内，让我们再来分析一下究竟会发生些什么：x_1值的改变在q点电平发生变化之前就已经在p点处被观察到，因为从x_1到q的路径上有一个附加的门（非门）。因此，q点的信号在变为1之前，p点的信号已经变为了0。所以在一个短时间内，曾经出现p和q的值都为0的情形，使得f瞬间降为0很快又重新恢复到1。这就产生了如图9-61a左边所示的那种信号波形。

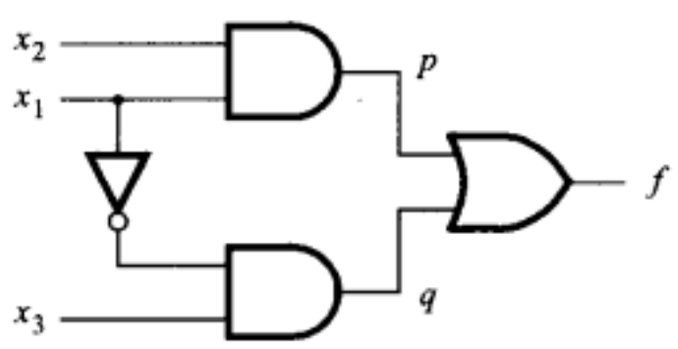
输出信号f上的毛刺可以用以下办法消除。电路实现的逻辑表达式为：

$$f = x_1x_2 + \bar{x}_1x_3$$

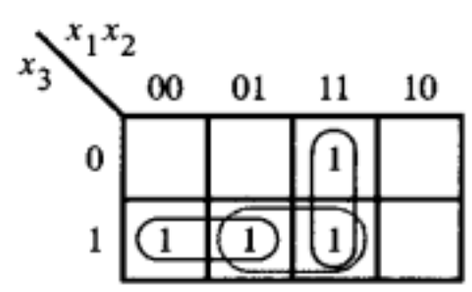
相应的卡诺图如图9-62b所示。用两个乘积项实现黑线圈着的质蕴涵项。上面解释过的冒险发生在从质蕴涵项x_1x_2到质蕴涵项$\bar{x}_1x_3$的转移过程中。冒险可以通过添加用灰线圈出（较大的圈）的第三个质蕴涵项来消除（这是2.5节性质17a中定义的合意项）。此时电路实现的逻辑表达式为：

$$f = x_1x_2 + \bar{x}_1x_3 + x_2x_3$$

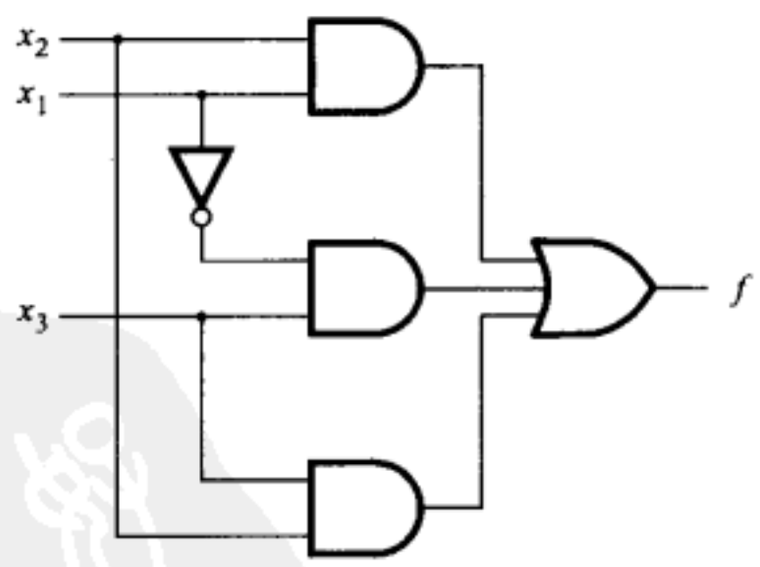
此时x_1从1到0的变化不会对输出f产生影响，因为当$x_2 = x_3 = 1$的时候，乘积项x_2x_3等于1，而不用关心x_1取什么值。从而得到如图9-62c所示的无冒险电路。



a) 存在冒险的电路



b) 卡诺图



c) 无冒险的电路

图9-62 一个静态冒险的例子

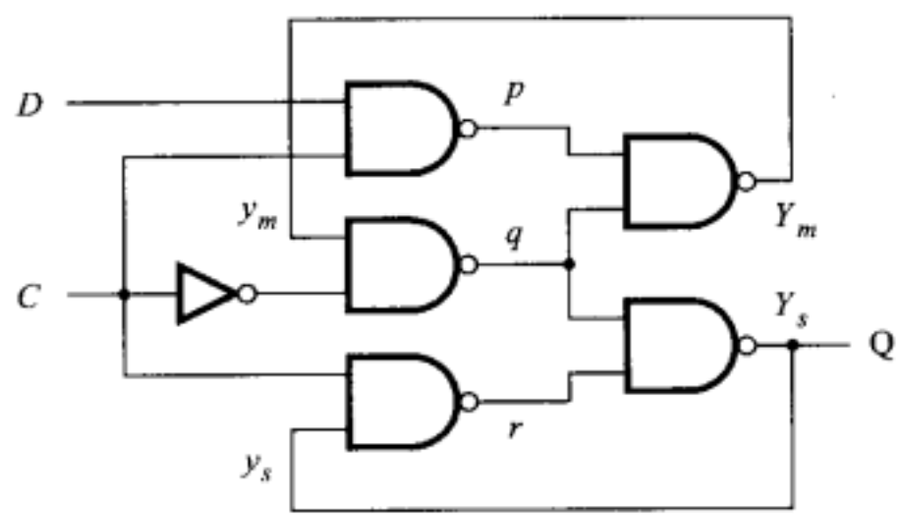
只要卡诺图中两个相邻的1没有被单个乘积项所覆盖，就存在着产生冒险现象的可能。因此消除冒险的一种方法是找到一个覆盖，使得该覆盖中的一些乘积项能够包含每一个相邻的1的对。于是，当输入变量的改变引起两个相邻1之间的转移时，因为这两个相邻的1项被包含在一个乘积项中，所以不会产生瞬间的脉冲干扰。

在异步时序电路中，冒险可以使得电路进入错误的稳定状态。例9.16说明了这样的情况。

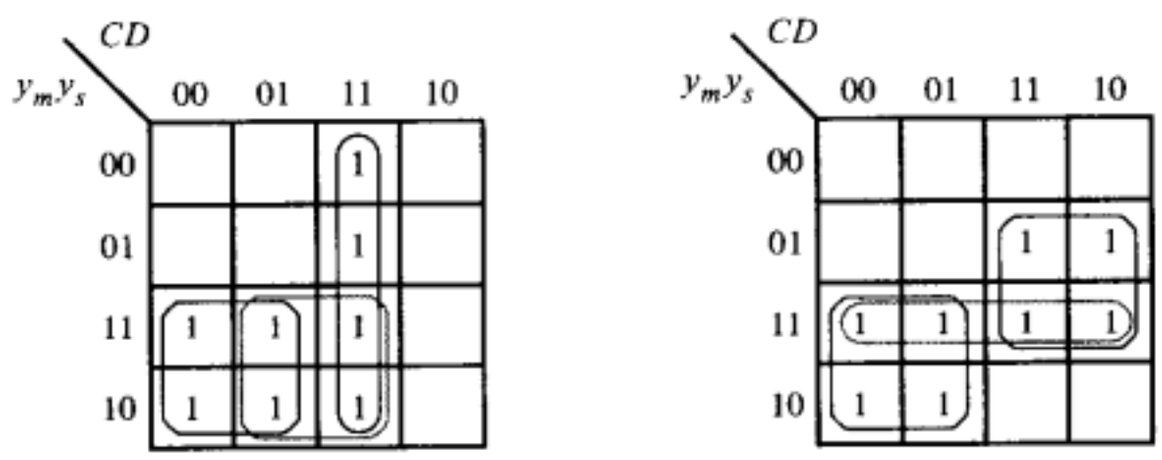
例9.16 在例9.2中，我们分析了实现主从D触发器的电路。根据图9-6a中的激励表，我们可以尝试综合能实现所要求的函数Y_m和Y_s且成本最低的电路。所得到的逻辑表达式如下：

$$
\begin{aligned}
Y_m &= CD + \overline{C}y_m \\
&= (C \uparrow D) \uparrow (\overline{C} \uparrow y_m) \\
Y_s &= \overline{C}y_m + Cy_s \\
&= (\overline{C} \uparrow y_m) \uparrow (C \uparrow y_s)
\end{aligned}
$$

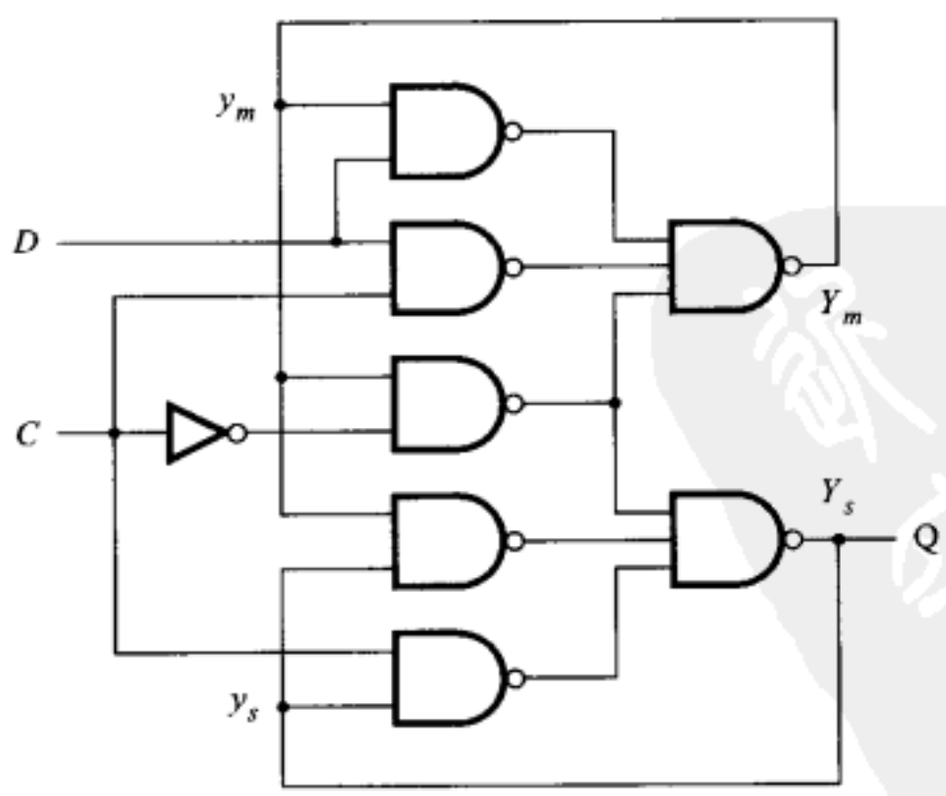
对应的电路如图9-63a所示。乍看之下，该电路似乎比第7章中讨论过的触发器更有吸引力，因为它的成本更低。问题是该电路存在静态冒险。



a) 最小成本电路



b) 图9-6a中Y_m和Y_s的卡诺图



c) 无冒险的电路

图9-63 用两级实现的主从D触发器

图9-63b所示是函数Y_m和Y_s的卡诺图。成本最低的实现方案基于卡诺图中用黑细线圈定的质蕴涵项。为了理解静态冒险如何影响电路的行为，我们先假设当前$Y_s = 1$且$C = D = 1$。此时，电路产生$Y_m = 1$。现在令C从1改变到0。为了保证触发器的正确动作，Y_s必须保持为1。如图9-63a所示，若C改变到0，则p和r的信号值应该都变为1。由于信号通过非门的延时，q可能还是1，从而导致电路产生$Y_m = Y_s = 0$。而Y_m的反馈将使q保持为1。结果使得电路进入并停留于一个错误的$Y_s = 0$的稳定状态中。

为了避免出现这样的冒险，必须将卡诺图中灰色线圈定的项也包含进来，从而得到如下的逻辑表达式：

$$Y_m = CD + \overline{C}y_m + Dy_m$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

得到的用与非门实现的电路见图9-63c所示。

通过将Y_m和Y_s的表达式改写为如下形式，便可以得到另一种用与非门实现的功能相同的电路：

$$\begin{aligned} Y_m &= CD + (\overline{C} + D)y_m \\ &= (C \uparrow D) \uparrow ((\overline{C} + D) \uparrow y_m) \\ &= (C \uparrow D) \uparrow ((C \uparrow D) \uparrow y_m) \\ Y_s &= \overline{C}y_m + (C + y_m)y_s \\ &= (\overline{C} \uparrow y_m) \uparrow ((\overline{C} \uparrow y_m) \uparrow y_s) \end{aligned}$$

这些表达式准确地与图7-13所示的电路相对应。 ■

例9.17 前面那些例子似乎告诉我们，通过将实现某给定函数的积之和电路中的所有质蕴涵项全部包含进来，就可以避免引入静态冒险。这的确是事实，但是并非总是有必要包含所有的质蕴涵项，而只需要将那些相邻的1的对的乘积项包含进覆盖即可，没有必要覆盖那些无关顶点。

考虑图9-64中的函数。实现该函数的无冒险电路应该包含那些圈定的项，从而得出如下的逻辑表达式：

$$f = \overline{x}_1 x_3 + x_2 x_3 + x_3 \overline{x}_4$$

为防止冒险并不一定需要质蕴涵项，例如$\overline{x}_1\overline{x}_2$就可以不需要，因为它只覆盖卡诺图最左边那一列的两个1，而这些1可以由$\overline{x}_1 x_3$覆盖。 ■

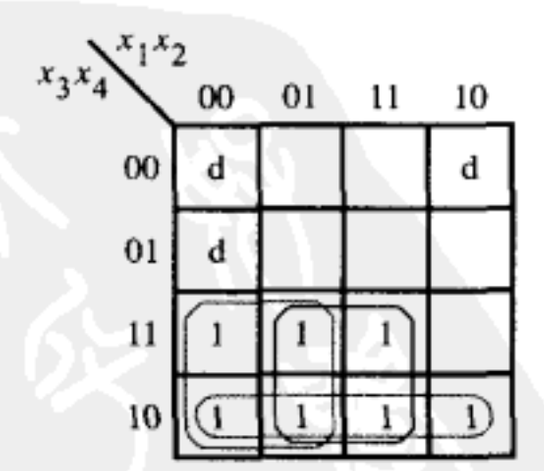


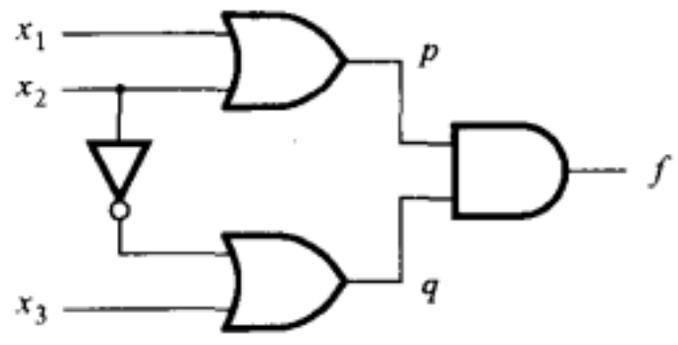
图9-64 例9.17函数的卡诺图

例9.18 静态冒险也可以在其他类型的电路中发生。图9-65a描绘了存在冒险现象的和之积电路。在该电路中，若$x_1 = x_3 = 0$，x_2从0变为1，则此时f应该保持为0不变。然而，若p点的信号改变得比q点的信号稍快一点，则在一个很短的时间段里p和q都会等于1，从而导致f上出现一个$0 \rightarrow 1 \rightarrow 0$的瞬间脉冲干扰。

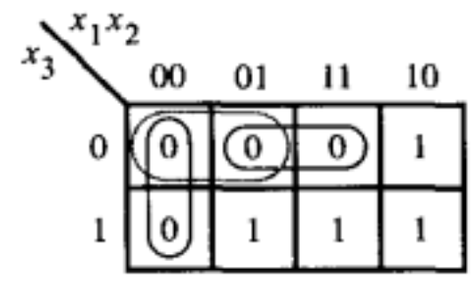
在和之积（用或与逻辑实现的）电路中，相邻的0之间的转移有可能导致冒险。因此若想设计无冒险的和之积电路，则必须包含覆盖所有相邻0对的和(sum)项。在本例中，卡诺图中灰色线圈定的项必须包含在内，从而得到如下的逻辑表达式：

$$f = (x_1 + x_2)(\overline{x}_2 + x_3)(x_1 + x_3)$$

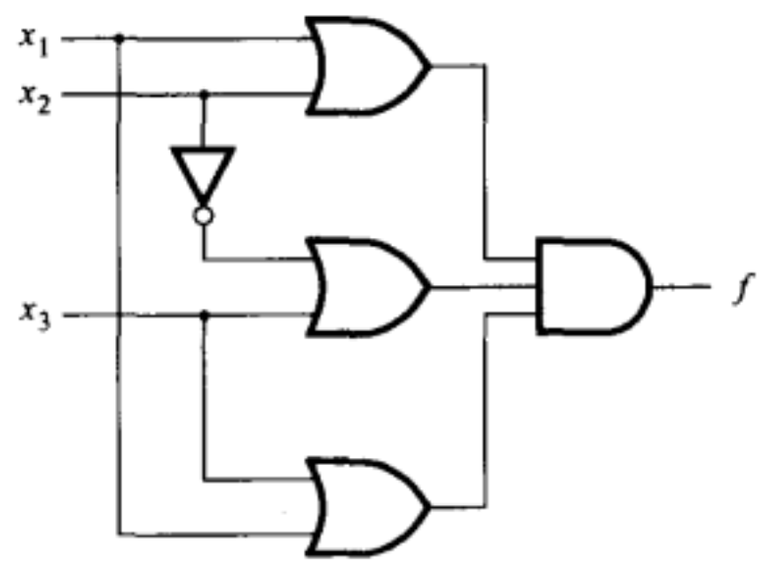
其实现电路如图9-65c所示。



a) 存在冒险的电路



b) 卡诺图

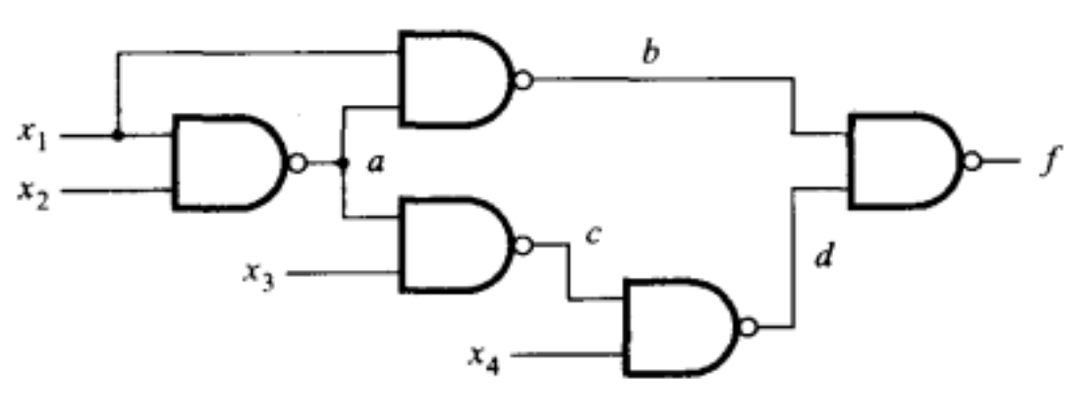


c) 无冒险电路

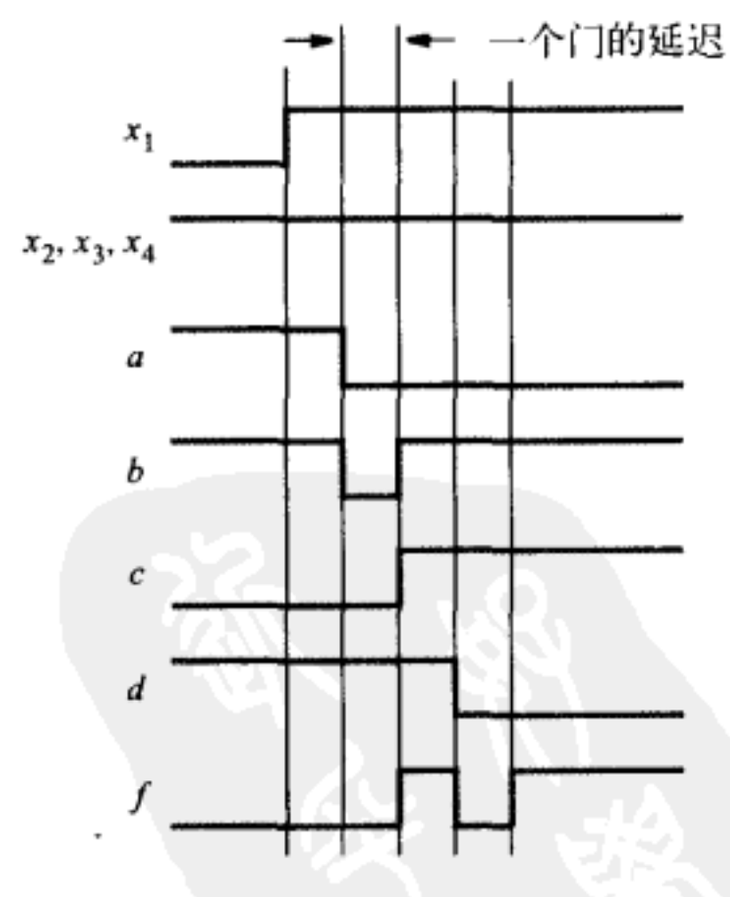
图9-65 和之积（用或与逻辑实现的）电路中的静态冒险 ■

9.6.2 动态冒险

动态冒险使得输出信号在0→1或1→0的转变过程中出现瞬间脉冲干扰。图9-66展示了这样一个例子。假设所有的与非门有相同的延迟，则可以画出如图9-66所示的时序图。两条垂直线之间的间隔时间对应于一个门的延迟。可以看到输出f中出现了一个不该出现的毛刺脉冲，应该想办法消除。



a) 电路



b) 时序图

图9-66 存在着动态冒险的电路

考虑一下实现该电路的函数是令人感兴趣的，其表达式如下：

$$f = x_1\bar{x}_2 + \bar{x}_3x_4 + x_1x_4$$

这就是成本最低的积之和表达式。若以这样的形式来实现电路，则得到的电路将不会出现任何静态或动态冒险。

动态冒险是由电路结构引起的，该电路结构对某给定信号的变化存在着多条传播路径。若输出信号改变了三次，在本例中即为0→1→0→1，则初始输入信号的变化至少可以沿着3条不同的路径传播至输出。存在动态冒险的电路中必然有某些部分也存在着静态冒险。从图9-66b所示的波形图上，我们可以发现线b上的信号就存在着静态冒险。

在第4章中，我们曾讨论过利用提取公因子或者分解技术来简化逻辑的方法，由此方法设计得到的多级电路，却往往会出现动态冒险。这种类型的冒险既难以检测，又很难处理。设计者通常可以通过利用两级电路，并确保其中不存在静态冒险的简单方法来避免出现动态冒险。

9.6.3 冒险的意义

异步时序电路中出现的瞬间脉冲干扰有可能使得电路进入错误的状态，且有可能稳定在该错误状态。因此，产生下一个状态变量的电路必须无冒险。只要遵守同一时刻只能有单个变量发生改变的原则，就足以消除冒险，而异步时序电路的基本前提就是，原始输入和状态变量的值必须同一时刻只能有一个改变。

在第4～6章关于组合逻辑电路的讨论中，我们忽略了冒险竞争现象，因为我们假设这些电路的输出完全取决于输入值。在同步时序电路中，输入信号在触发器的建立和保持时间内必须保持稳定。因此相对于时钟信号有效沿而言，在触发器的建立和保持时间之外即使出现毛刺干扰也没有什么关系。

9.7 一个完整的设计实例

在前面的几节中，我们检查了异步时序电路设计的各个方面。在本节中我们给出一个完整的设计实例，覆盖了与设计有关的所有必须步骤。

自动售货机控制器

借助于自动售货机的控制机制，我们能深入地理解数字电路的某一类应用。在第8章中我们曾在同步环境中用它举例。在9.2节中，我们曾用自动售货机举例说明如何分析设计对象。现在考虑一个与例8.7类似的自动售货机控制器，来看一下如何用异步时序电路来实现该控制器。控制器的设计需求如下：

1) 机器接受5分和1角硬币。

2) 总金额必须达到15分才能给出糖果。

3) 如果投入的硬币为20分，机器将不会找零。

硬币是一次一枚地投入机器的。当硬币感应机制检测到已投入一枚5分硬币或1角硬币，随即分别产生信号$N = 1$和$D = 1$。同一时刻不可能发生$N = D = 1$的情况。当投入的硬币总金额等于或超过15分的时候，机器将给出糖果，随即复位到初始状态。

图9-67显示了所需FSM的状态图。该状态图的推导采用了直接了当的方法，即将投入5分和1角硬币所有可能的序列都以树状结构枚举出来。为了使该状态图清楚有序，用标签D和N分别表示输入条件$DN = 10$和$DN = 01$。输入条件$DN = 00$被简单地标记为0。

在进入下列两组状态后，机器会给出糖果：

1) 当投入硬币总金额为15分后，机器进入状态F、H和K。

2) 当投入硬币总金额为20分后，机器进入状态I和L 。

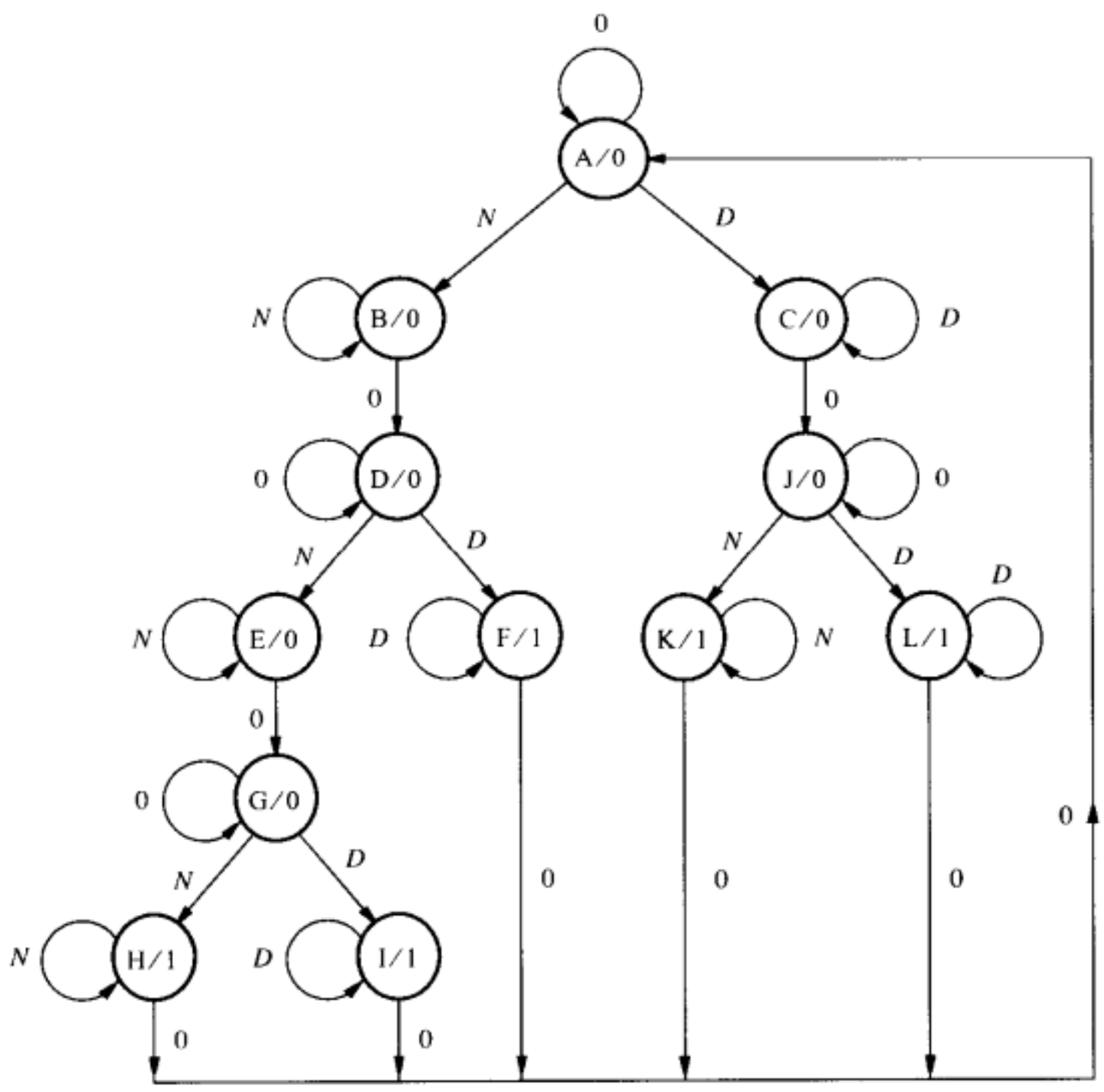


图9-67 自动售货机控制器的初始状态图

状态图对应的流程表见图9-68所示，可以用如下的划分步骤对该流程表进行简化：

$$P_1 = (ADGJ)\ (BE)\ (C)\ \ (FIL)\ (HK)$$
$$P_2 = (A)\ (D)\ (GJ)\ (B)\ (E)\ (C)\ (FIL)\ (HK)$$
$$P_3 = P_2$$

用G表示等价状态G和J，用F 表示等价状态F、I和L，用H表示等价状态H和K，便可以得到部分化简后的流程表，见图9-69所示。该流程表的状态合并图见图9-70所示，从该合并图可以看出状态C和E可以合并，F和H也可以合并。从而得到最后简化的流程表，见图9-71a所示。同样的信息以状态图的形式表示见图9-72所示。

当前状态	下一状态				输出
	DN = 00	01	10	11	z
A	A	B	C	–	0
B	D	B	–	–	0
C	J	–	C	–	0
D	D	E	F	–	0
E	G	E	–	–	0
F	A	–	F	–	1
G	G	H	I	–	0
H	A	H	–	–	1
I	A	–	I	–	1
J	J	K	L	–	0
K	A	K	–	–	1
L	A	–	L	–	1

图9-68 自动售货机控制器的初始流程表

当前状态	下一状态				输出
	DN = 00	01	10	11	z
A	A	B	C	–	0
B	D	B	–	–	0
C	G	–	C	–	0
D	D	E	F	–	0
E	G	E	–	–	0
F	A	–	F	–	1
G	G	H	F	–	0
H	A	H	–	–	1

图9-69 状态简化过程的第一步

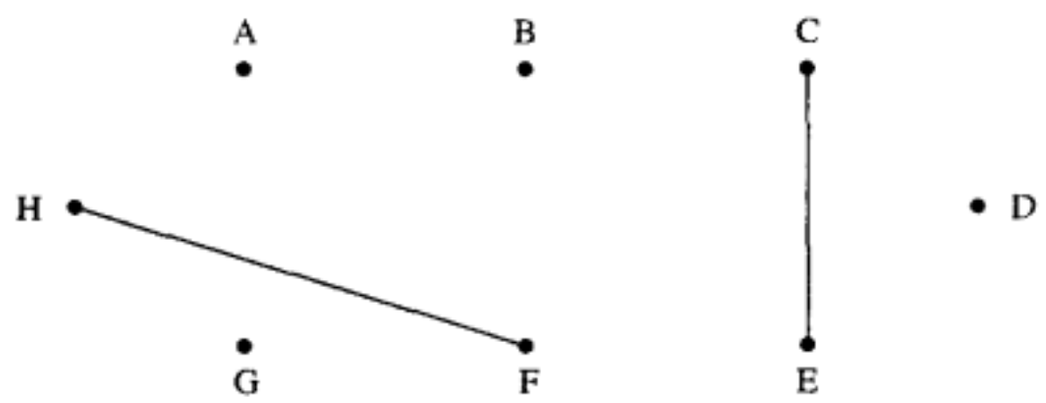


图9-70 与图9-69对应的合并图

当前状态	下一状态 DN = 00	01	10	11	输出 z
A	A	B	C	—	0
B	D	B	—	—	0
C	G	C	C	—	0
D	D	C	F	—	0
F	A	F	F	—	1
G	G	F	F	—	0

a) 简化的流程表

当前状态	下一状态 DN = 00	01	10	11	输出 z
A	1	2	4	—	0
B	5	2	—	—	0
C	8	3	4	—	0
D	5	3	7	—	0
F	1	6	7	—	1
G	8	6	7	—	0

b) 重标记的流程表

图9-71 化简后的流程表

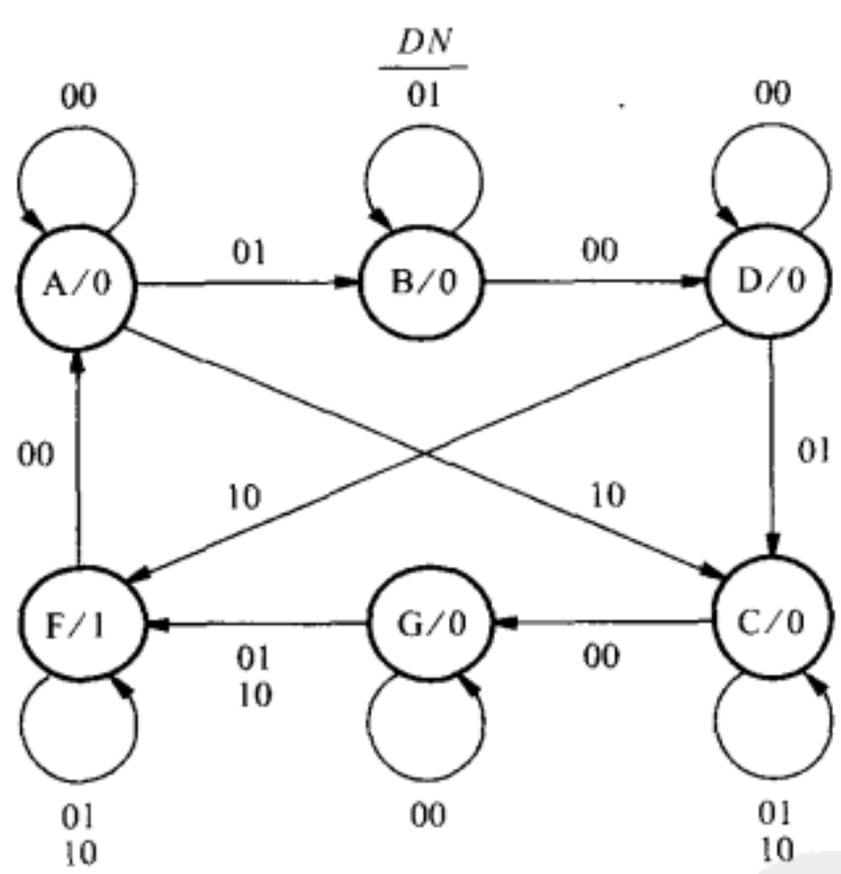
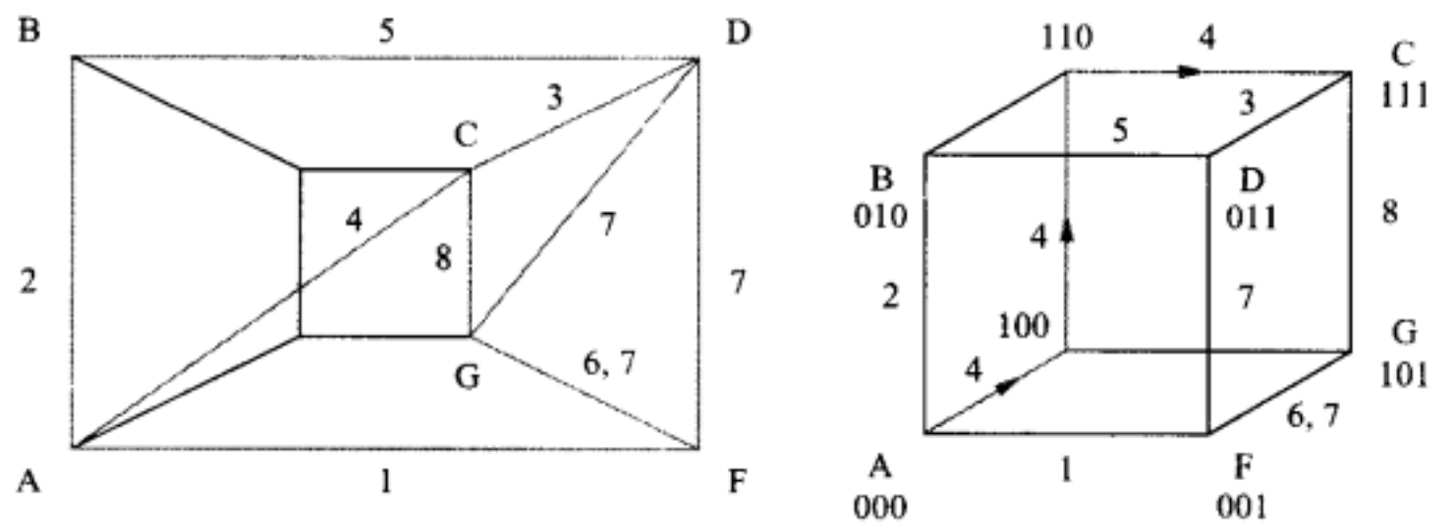


图9-72 自动售货机控制器的状态图

接下来我们必须找到一个合适的状态分配。重新标记后流程表如图9-71b所示，其中每个稳定状态都与一个唯一的数字相关联。由此可得到如图9-73a所示的转移图。因为我们想把该图插入到三维的立方体中，所以得到如图9-73a所示的8个顶点的状态转移图。在该图上出现两个对角线的转移。其中状态*D*和*G*之间的转移（标记为7）不用管它，因为它只是一条可选路径。而从状态*A*到*C*的转移（标记为4）是必需的，而且可以经由那些图9-73b中以箭头线标出的未使用的状态来实现*A*到*C*的转移。这样，状态转移图就可以被嵌入到如图9-73b所示的三维的立方体中。利用该图中的状态分配，就能得到如图9-74所示的激励表。



a) 转移图　　b) 嵌入立方体中

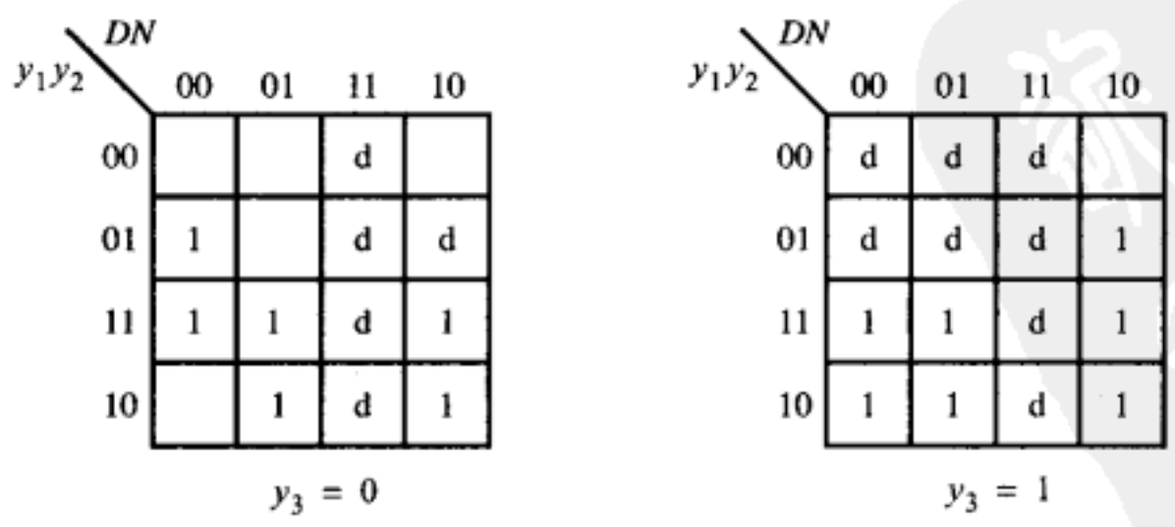
图9-73　确定状态分配

	当前状态 $y_3y_2y_1$	下一状态 DN = 00	01	10	11	输出 z
		$Y_3Y_2Y_1$				
A	000	(000)	010	100	—	0
B	010	011	(010)		—	0
C	111	101	(111)	(111)	—	0
D	011	(011)	111	001	—	0
F	001	000	(001)	(001)	—	1
G	101	(101)	001	001	—	0
	100	—	—	110	—	0
	110	—	—	111	—	0

图9-74　基于如图9-73b所示的状态分配的激励表

图9-75给出了各个次（下一个）状态函数的卡诺图。从这些图中，我们可以得到以下这些无冒险表达式：

$$Y_1 = \overline{N}y_2 + Ny_1 + Dy_1 + y_1y_3 + y_1y_2$$
$$Y_2 = N\overline{y}_1 + Ny_2 + \overline{y}_1y_3 + \overline{D}y_2y_3 + Dy_2y_3$$
$$Y_3 = D\overline{y}_1 + y_2y_3 + Ny_1y_2 + \overline{D}y_3\overline{N}$$



a) Y_1的图

图9-75　图9-74中各个函数的卡诺图

y_1y_2 \ DN	00	01	11	10
00		1	d	
01	1	1	d	d
11	1	1	d	
10			d	

$y_3 = 0$

y_1y_2 \ DN	00	01	11	10
00	d	d	d	1
01	d	d	d	1
11		1	d	1
10			d	

$y_3 = 1$

b) Y_2的图

y_1y_2 \ DN	00	01	11	10
00			d	1
01			d	d
11		1	d	
10			d	

$y_3 = 0$

y_1y_2 \ DN	00	01	11	10
00	d	d	d	1
01	d	d	d	1
11	1	1	d	1
10	1		d	

$y_3 = 1$

c) Y_3 的图

图9-75 （续）

在这些表达式中，除了$y_1 y_2$项外的所有乘积项都是实现成本最低的积之和（译者注：原文为和之积有错误，应该为积之和才对）（即用与或门实现的）逻辑电路所必需的。$y_1 y_2$ 项是为了防止Y_1表达式中出现冒险才添加的。输出表达式为：

$$z = y_1\bar{y}_2\bar{y}_3$$

9.8 小结

异步时序电路的设计比同步时序电路的要更加困难些。由冒险竞争带来的困难给我们提出了必须小心处理的问题。目前，针对异步电路设计的CAD工具还有许多问题，并不能提供有效的帮助。因此，大部分设计者在实际应用中往往还是采用同步时序电路来完成设计任务。

异步时序电路最重要的优点是运行速度快，因为异步时序电路不需要时钟，电路运行的速度完全取决于电路的传播延迟。在由几个子电路组成的异步系统中，某些子电路有可能比其他电路的运行速度更快些，因此系统全面的性能有可能得到改善。相比之下，在同步系统中时钟周期必须足够长，以适应最慢的那部分电路的延迟，而这样就使电路整体的速度性能受到很大的牵连。

若电路系统是由两个或两个以上工作于不同时钟下的同步电路所组成的，则在设计的这样的系统时异步电路技术十分有用。这类电路之间的信号交换实质上往往是异步的。

从读者的视角而言，将异步电路当做一个极好的媒介是非常有益的，它能帮助我们更深入和全面地理解数字电路。异步电路说明了电路结构中固有的信号传播延迟和冒险竞争现象带来的后果。通过电路中稳定状态和不稳定状态的存在，用图示的方法演示并说明了稳定性的概念。读者可以通过查阅参考文献[1-6]，进一步了解有关异步时序电路的知识。

9.9 问题求解举例

本节列举了读者可能遇到的某些典型问题，并且展示了这些问题是如何求解的。

例9.19 推导描述图9-76所示电路行为的流程表（flow table）。

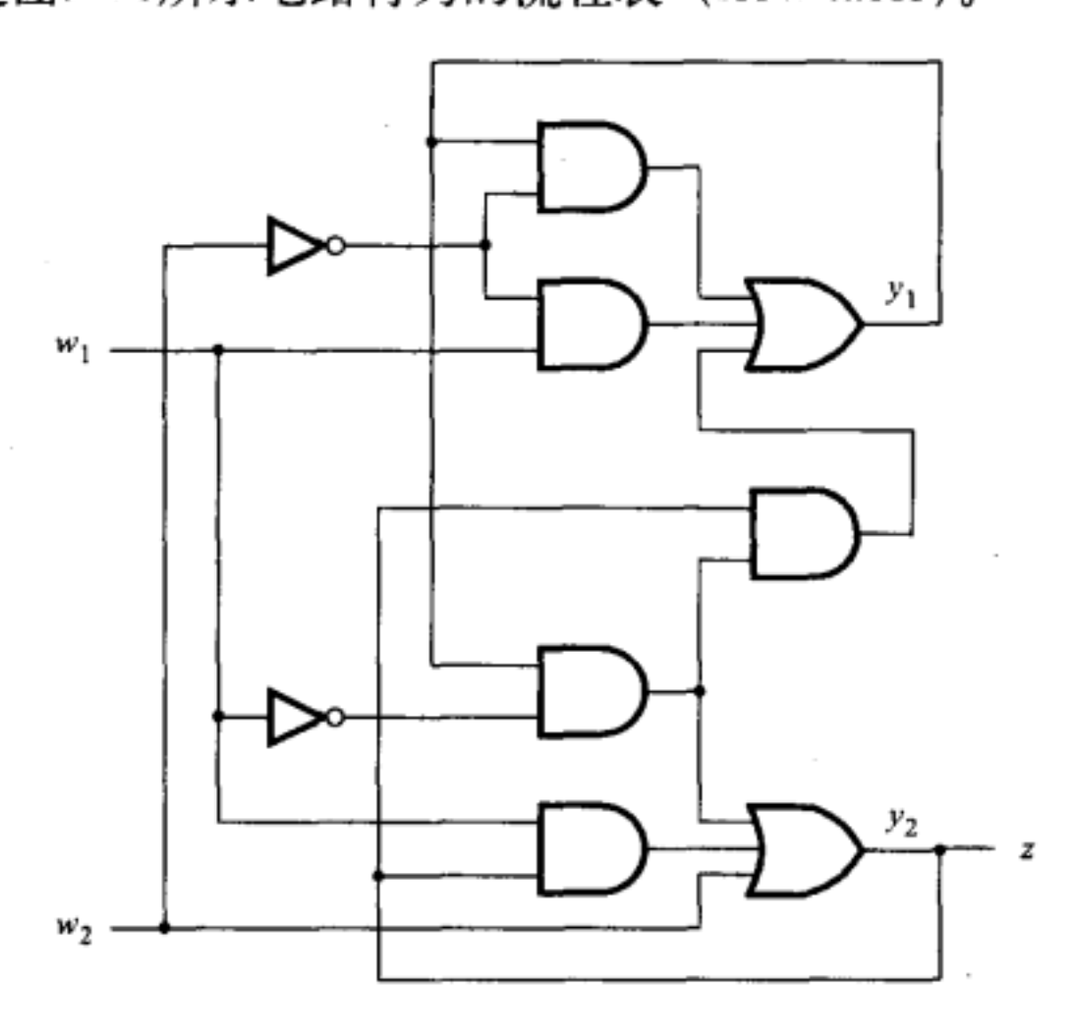


图9-76 例9.19 的电路

解：模仿图9-8电路中描述门传播延迟的方法，图9-76所示的电路可以用如下的下一个状态表达式和输出表达式来描述：

$$Y_1 = w_1\overline{w}_2 + \overline{w}_2 y_1 + \overline{w}_1 y_1 y_2$$
$$Y_2 = w_2 + \overline{w}_1 y_1 + w_1 y_1$$
$$z = y_2$$

从这几个表达式可以推导出如图9-77a所示的激励表。假设状态分配为：$A = 00, B = 01, C = 10$，$D = 11$， 由此推出如图9-77b所示的流程表。

当前状态 y_2y_1	下一状态 $w_2w_1 = 00$	01	10	11	输出 z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	00	01	10	10	0
01	11	01	10	10	0
10	00	11	10	10	1
11	11	11	11	10	1

a) 激励表

当前状态	下一状态 $w_2w_1 = 00$	01	10	11	输出 z
A	A	B	C	C	0
B	D	B	C	C	0
C	A	D	C	C	1
D	D	D	D	C	1

b)由电路实现的流程表

当前状态	下一状态 $w_2w_1 = 00$	01	10	11	输出 z
A	A	B	C	C	0
B	D	B	—	C	0
C	A	D	C	C	1
D	D	D	D	C	1

c) 最终的流程表

图9-77 图9-76所示电路的激励表和流程表

因为在一个给定的稳定状态，电路的输入信号每次只能有一个发生变化，因此流程表中的某个项（格）可以被指定为不确定项。当电路稳定于状态B，而且输入取值$w_2 w_1 = 01$时，就是这种情况。此时两个输入不能同时发生改变，这就意味着流程表中对应的项应该被指定为不确定。但是当电路稳定于状态A，且输入取值$w_2 w_1 = 00$时，会出现不同的情况。在这种情况下，我们不能把$w_2 w_1 = 11$列的转移标记为不确定。这是因为，若该电路处于稳定状态B，则当w_2从0变化到1时，该电路的状态必须能够转到状态C。状态B和C分别被实现为$y_2 y_1 = 01$和 $y_2 y_1 = 10$。因为两个状态变量都必须改变其值，所以从01到10的改变将可能经由中间状态11或者00发生，这取决于该电路中不同路径的延迟。若y_2先发生改变，则电路将通过不稳定状态D，然后进入稳定状态C。但是，若w_1先发生改变，则该电路在进入稳定状态C前，必须经由不稳定状态A。因此，必须指定第一行进入状态C的转移。这是安全竞争的一个例子，在这个例子中，电路能达到正确的目的状态，而与电路不同路径的传播延迟没有任何关系。最终的流程表如图9-77c所示。

例9.20 在图9-76所示的电路中，是否会出现冒险竞争？

解：图9-78给出了在例9.19中推得的下一个状态表达式的卡诺图。从该卡诺图可以看到，所有的质蕴涵项都被包括在Y_1的表达式中。但是Y_2的表达式只包括四个质蕴涵项中的三个。当$w_2 y_2 y_1 = 011$且w_1从0变化到1（或者从1变化到0）时，该电路存在一个静态的竞争。该竞争可以通过给Y_2表达式添加第四个质蕴涵项$y_1 y_2$而去除掉。

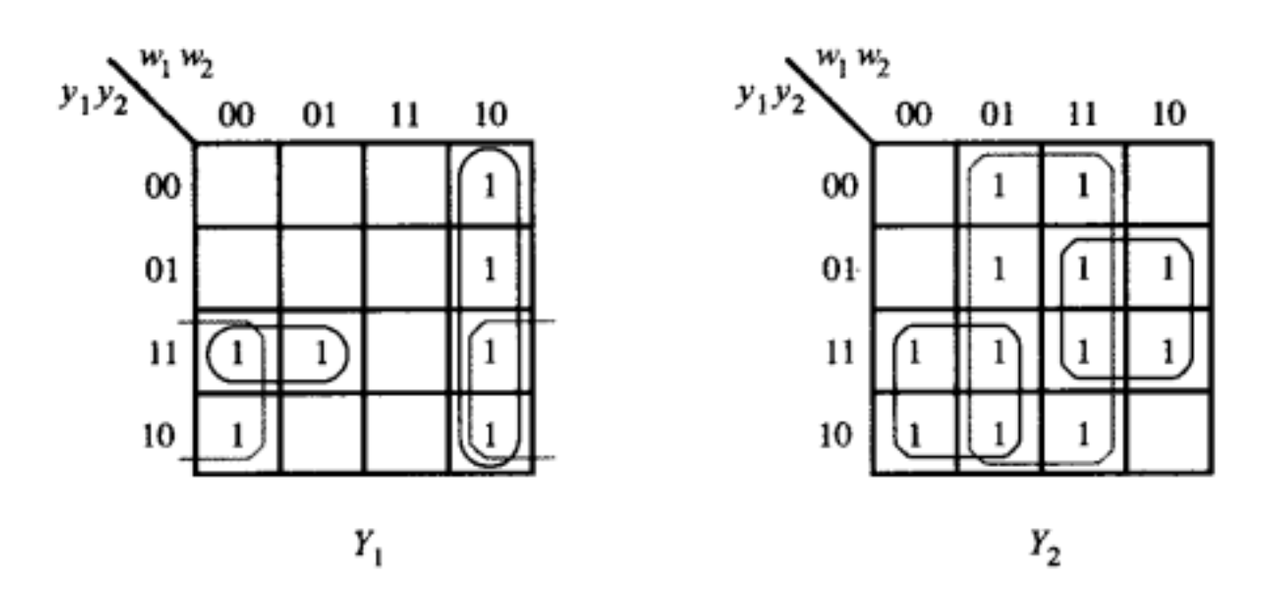


图9-78 图9-76所示电路的卡诺图

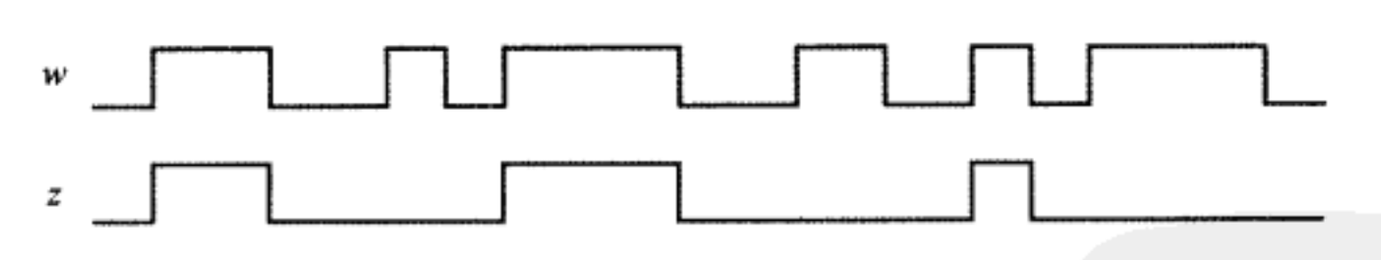
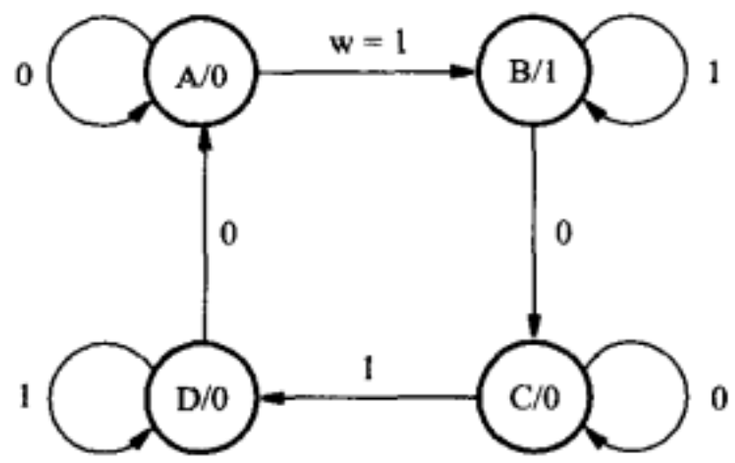


图9-79 例9.21的波形图

例9.21 某电路只有一个输入w 和一个输出 z。向输入端w 输入一系列脉冲。请设计一个电路使得输出端 z 的波形必须隔一个脉冲与输入端w的脉冲保持一致，具体波形如图9-79所示。（用门级电路实现，不能用触发器。——译者注）

解：图9-80展示了该电路的状态图和相应的流程表。将这个状态图与图9-13所示的例9.4中定义的有限状态机（该FSM定义了一个串行的奇偶校验位发生器）做一个比较。唯一的不同点在输出信号上。在本题中，只在状态B下，$z = 1$。因此，下一个状态的表达式与例9.4是相同的。而输出表达式为

$$z = y_1\bar{y}_2$$



a) 状态图

当前状态	下一状态 $w=0$	下一状态 $w=1$	输出 z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	0
D	A	(D)	0

b) 流程表

图9-80 例9.21的状态图和流程表

例9.22 考虑图9-81中的流程表。化简该流程表，并且找到一种状态分配，使得在保留摩尔模型的前提下，尽可能简单地实现该有限状态机。推导出该流程表的激励表。

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z
A	(A)	E	C	—	0
B	—	E	H	(B)	1
C	G	—	(C)	F	0
D	A	(D)	—	B	1
E	G	E	—	B	0
F	—	D	C	(F)	0
G	(G)	E	C	—	0
H	A	—	(H)	B	1

图9-81 例9.22的流程表

解：使用图9-81流程表上的分割过程，可得到下列分割表达式：

$$P_1 = (ACEFG)\ (BDH)$$
$$P_2 = (AG)\ (B)\ \ (C)\ (D)\ (E)\ (F)\ (H)$$
$$P_3 = P_2$$

将A和G结合在一起便可得到如图9-82所示的流程表。这个流程表的合并图如图9-83所示。将状态(A, E), (C, F) 和 (D,H) 合并可以得到简化的流程表，如图9-84所示。为了找到好的状态分配，我们重新对这个流程表进行标记，如图9-85所示，并构建如图9-86a所示的转移图。在这个转移图中，唯一的问题是从状态D到状态A的转移（该转移被标记为1）。如果我们在流程表中指定的话，可以使从D到A的转移经由状态C进行。这样就使得不再需要从D到A的直接转移，如图9-86b所示。最后得到的流程表和相应的激励表如图9-87所示。

例9.23 设有逻辑函数$f(x_1, x_2, x_3, x_4, x_5) = \sum m(2, 3, 14, 17, 19, 25, 26, 30) + D(10, 23, 27, 31)$。推导出表示逻辑函数$f$的无冒险竞争且成本最低的积之和表达式。

解：该函数的卡诺图如图9-88所示。根据卡诺图可以得到所需要的表达式如下：

$$f = x_1\bar{x}_3x_5 + x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_2\bar{x}_3x_4x_5$$

在上面的表达式中，前三个乘积项覆盖了该卡诺图中所有的1。为了避免当$x_2x_3x_4x_5$ = 0011时，x_1从0变到1（或者从1变到0）产生的冒险竞争，所以第四项是需要的。这样每对相邻的1都可以被该表达式中的某个质蕴涵项所覆盖。

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	E	C	—	0
B	—	E	H	B	1
C	A	—	C	F	0
D	A	D	—	B	1
E	A	E	—	B	0
F	—	D	C	F	0
H	A	—	H	B	1

图9-82 分割过程后的简化

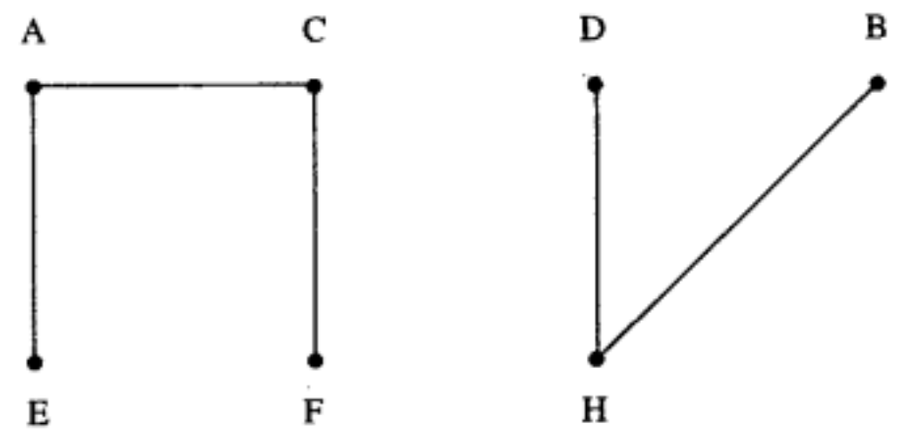


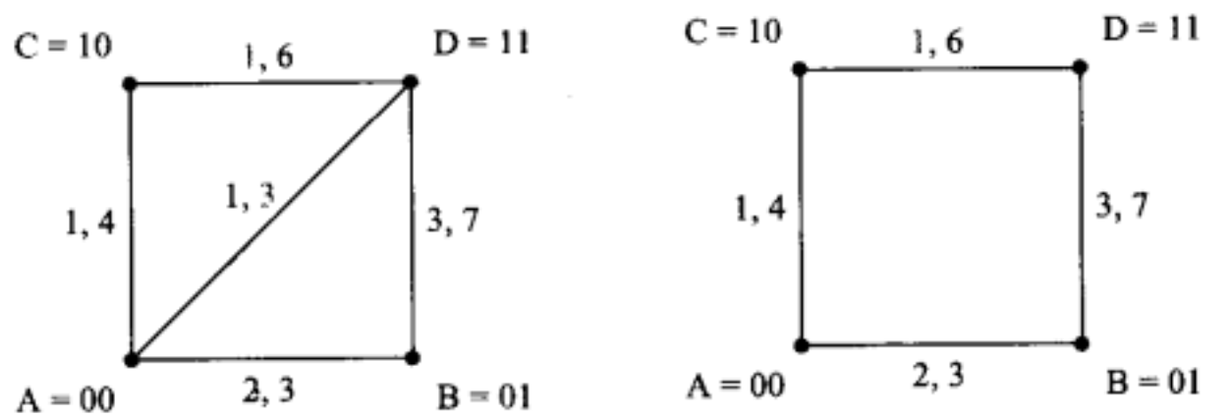
图9-83 图9-82中流程表的合并图

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	A	C	B	0
B	—	A	D	B	1
C	A	D	C	C	0
D	A	D	D	B	1

图9-84 图9-82中有限状态机的简化流程表

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	1	2	4	3	0
B	—	2	7	3	1
C	1	6	4	5	0
D	1	6	7	3	1

图9-85 图9-84中重新标记的流程表



a) 初始转移图　　b) 扩张的转移图

图9-86 图9-85中的转移图

当前状态	下一状态 w_2w_1 = 00	01	10	11	输出 z
A	A	A	C	B	0
B	—	A	D	B	1
C	A	D	C	C	0
D	C	D	D	B	1

a) 最终的流程表

当前状态 y_2y_1	下一状态 w_2w_1 = 00 Y_2Y_1	01 Y_2Y_1	10 Y_2Y_1	11 Y_2Y_1	输出 z
00	00	00	10	01	0
01	—	00	11	01	1
10	00	11	10	10	0
11	10	11	11	01	1

b) 激励表

图9-87 例9.22中的激励表和流程表

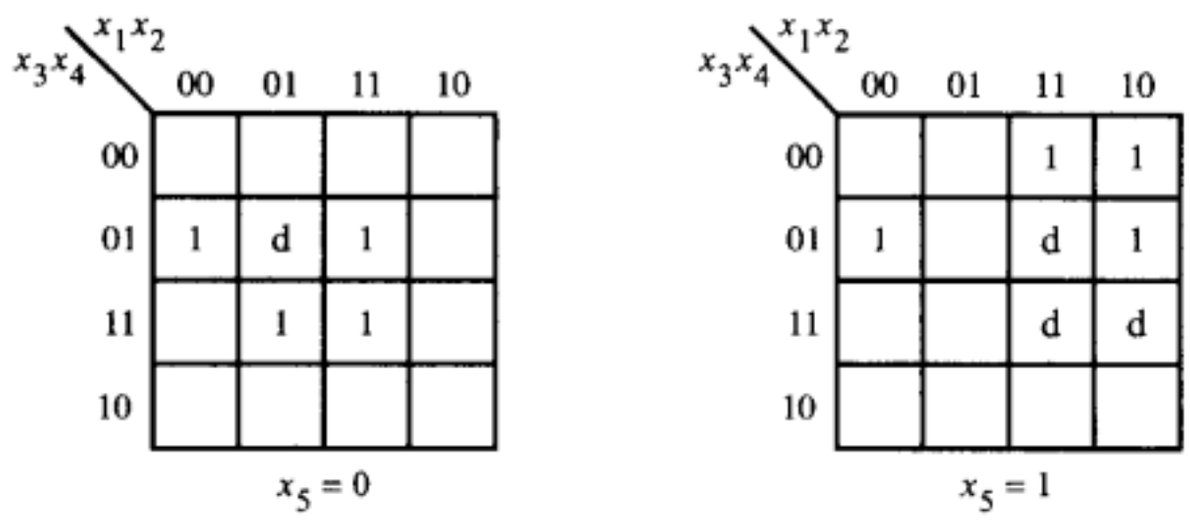
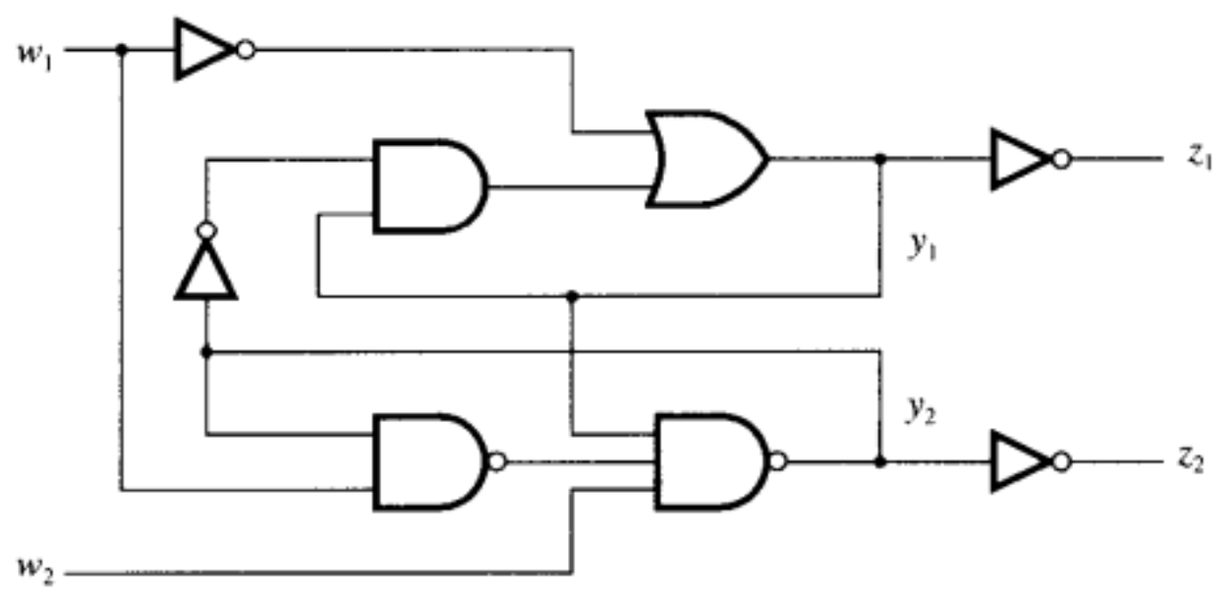


图9-88 例9.23中的卡诺图

练习题

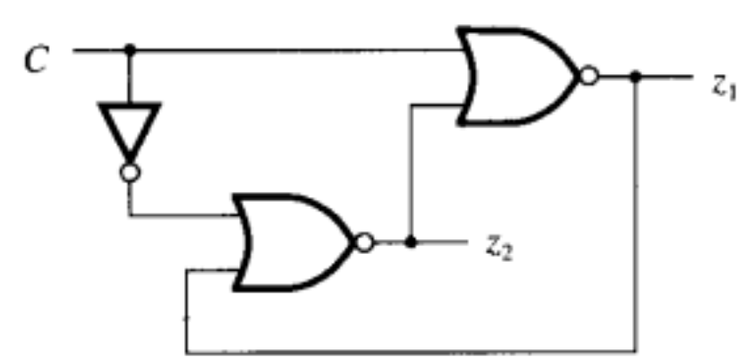
标有*号的习题答案参见书后。

***9.1** 推导出描述如图P9-1所示电路的行为的流程表。将你的解决方案和图9-21中的流程表进行比较，看看有没有相似之处？



图P9-1 练习题9.1的电路

9.2 思考图P9-2所示的电路。画出信号C，z_1和z_2的波形。假设C是一个方波型时钟信号，且每个门的传播延迟都是△。用流程表的形式描述该电路的行为，确保流程表能产生要求的信号波形。（提示：用米利模型。）



图P9-2 练习题9.2的电路

9.3 推导出与图P9-3中的流程表功能行为相同的最小流程表。

9.4 推导出与图P9-4中的流程表功能行为相同的最小摩尔型流程表。

9.5 为图9-42中的流程表找到一个合适的状态分配，使所用的状态尽可能少，并推导出表示下一状态和输出的逻辑表达式。

9.6 为图9-42中的流程表找到一个合适的状态分配，必须使用如例9.15所介绍的等价状态对。推导出表示下一个状态和输出的表达式。

9.7 为图9-42中的流程表找到一个状态分配，必须使用独热码。推导出表示下一个状态和输出的表达式。

当前状态	下一状态 $w_2w_1=$ 00	01	10	11	输出 z
A	A	B	C	–	0
B	D	B	–	–	0
C	P	–	C	–	0
D	D	E	F	–	0
E	G	E	–	–	0
F	M	–	F	–	0
G	G	H	I	–	0
H	J	H	–	–	0
I	A	–	I	–	1
J	J	K	L	–	0
K	A	K	–	–	1
L	A	–	L	–	1
M	M	N	O	–	0
N	A	N	–	–	1
O	A	–	O	–	1
P	P	R	S	–	0
R	T	R	–	–	0
S	A	–	S	–	1
T	T	U	V	–	0
U	A	U	–	–	1
V	A	–	V	–	1

图P9-3 练习题9.3的流程表

当前状态	下一状态 $w_2w_1=$ 00	01	10	11	输出 z
A	A	B	C	–	0
B	K	B	–	H	0
C	F	–	C	M	0
D	D	E	J	–	1
E	A	E	–	M	0
F	F	L	J	–	0
G	D	G	–	H	0
H	–	G	J	H	1
J	F	–	J	H	0
K	K	L	C	–	1
L	A	L	–	H	0
M	–	G	C	M	1

图P9-4 练习题9.4的流程表

***9.8** 利用图9-40a中的合并图来实现图9-39所示的FSM。

9.9 为图P9-5中的流程表所定义的FSM找到一个合适的状态分配，并根据该状态分配推导出FSM的下一个状态和输出的表达式。

当前状态	下一状态 $w_2w_1=00$	01	10	11	输出 z
A	A	B	C	–	0
B	D	B	–	G	0
C	F	–	C	G	0
D	D	E	C	–	1
E	A	E	–	G	0
F	F	E	C	–	0
G	–	B	C	G	1

图P9-5 习题9.9的流程表

***9.10** 为下面这个逻辑函数找到一个成本最低且无冒险的电路实现。

$$f(x_1,\cdots,x_4)=\sum m(0,4,11,13,15)+D(2,3,5,10)$$

9.11 对下面这个逻辑函数重复练习题9.10的要求。

$$f(x_1,\cdots,x_5)=\sum m(0,4,5,24,25,29)+D(8,13,16,21)$$

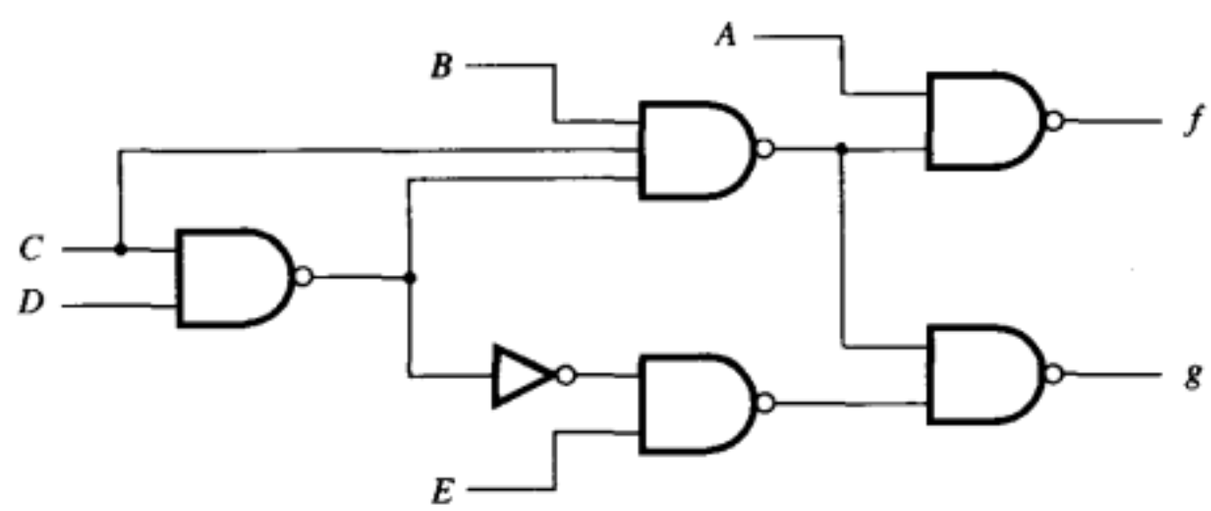
***9.12** 为下面的逻辑函数找到成本最低且无冒险的和之积（或与逻辑）的实现。

$$f(x_1,\cdots,x_4)=\prod M(0,2,3,7,10)+D(5,13,15)$$

9.13 对下面的逻辑函数重复练习题9.12的要求。

$$f(x_1,\cdots,x_5)=\prod M(2,6,7,25,28,29)+D(0,8,9,10,11,21,24,26,27,30)$$

***9.14** 考虑图P9-6中的电路，该电路存在冒险吗？

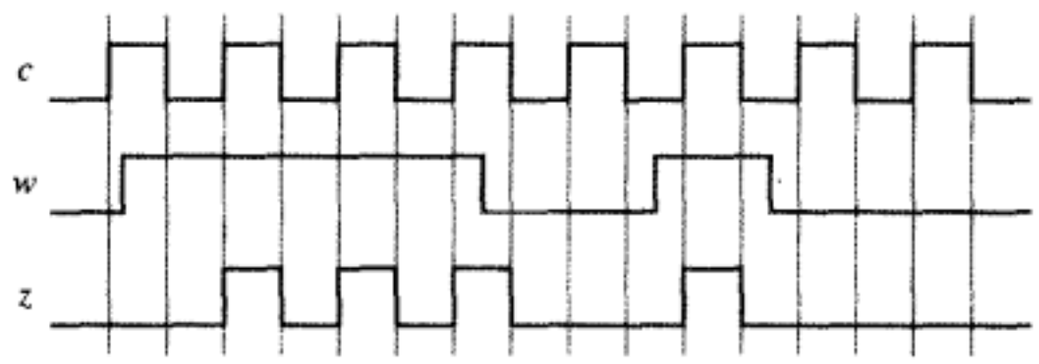


图P9-6 习题9.14的电路

9.15 请独自设计一个存在动态冒险的原创电路。

9.16 一个自动售货机的控制机制接受五分和一角的硬币。当投入硬币的金额达到20分时，机器随即递交货物；如果存入金额为25分时，机器不找零。设计实现所要求的控制的FSM，使用尽可能少的状态。寻找合适的状态分配，且据此推导出表示下一个状态和输出的逻辑表达式。

***9.17** 设计一个满足以下规定的异步电路。电路有两个输入：一个为时钟输入c，另外一个为控制输入w。当$w=1$的时候，输出z复制时钟脉冲；当$w=0$的时候，$z=0$。输出z上出现的脉冲必须为完整的脉冲。因此，如果当w从0改变为1的时候，$c=1$，那么电路将不会在z上产生部分的脉冲，而是等到下一个时钟脉冲到来的时候才产生$z=1$的输出。如果当w从1改变为0的时候，$c=1$，那么必须保证这个时钟脉冲仍旧能被完整地输出，即只要$c=1$，则$z=1$。图P9-7展示了要设计的电路的操作波形。



图P9-7 习题9.17的波形

9.18 设计要求同练习题9.17，但是对电路行为的要求有所改变：当$w = 1$的时候，输出z应该只有一个脉冲；若c上产生了多个脉冲，则只有一个脉冲会被复制到z上。

9.19 例9.6描述了两台设备竞争共享资源的简单仲裁器。设计一个类似的仲裁器，可用于3台设备竞争共享资源。若同时发生多个请求（即一台设备已被准许使用共享资源，在放弃请求之前，其他两个设备也都提出了自己的请求），假设设备之间的优先权顺序如下：设备1> 设备2> 设备3。

9.20 在例9.6的讨论中，我们提到了还可以用互斥元件来防止对FSM的两个请求输入同时为1。为这种情况设计一个仲裁器。

9.21 在例9.21中，我们设计了一个电路，该电路每隔一个脉冲可以在输出端 z 将输入到w端的脉冲复制一个，请设计一个类似的电路，可以每隔两个脉冲在输出端 z 将输入到w端的脉冲复制。

9.22 在例9.22中，我们将状态D和状态H合并以实现图9-82中的有限状态机。根据图9-83所示的合并图，另外一种办法是将状态B和状态H合并。请推导出用这种方法合并图的实现，画出该FSM的激励表。

参考文献

1. K. J. Breeding, *Digital Design Fundamentals*, (Prentice-Hall: Englewood Cliffs, NJ, 1989).
2. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., (Wiley: New York, 1993).
3. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
4. N. L. Pappas, *Digital Design*, (West: St. Paul, MN, 1994).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
6. C. J. Myers, *Asynchronous Circuit Design*, (Wiley: New York, 2001).

第10章　数字系统设计

在前面几章中，我们曾向读者展示了许多种简单电路（例如多路器、译码器、触发器、寄存器和计数器）是如何设计的。上述简单电路可以用作电路的构件。在本章中，我们将列举一些较复杂的电路示例，这些复杂的电路可以由简单的电路构件来搭建，而数字系统就是由这些较复杂的电路所组成的。为了从实际出发，我们所举的数字系统设计示例的规模都不是很大，但所展示的设计技术适用于任何规模的数字系统。在列举了几个示例之后，我们将讨论一些具体的实际问题。例如，如何保证单个和多个芯片中触发器时钟信号的可靠性，如何处理输入信号与时钟的不同步等类似问题。

数字系统由两部分组成：*数据路径*（datapath）*电路*和*控制电路*。数据路径电路用于数据的存储和操作，以及数据从系统的一部分到另一部分的传输。数据路径电路包含一些构造块，如寄存器、移位寄存器、计数器、多路器、译码器、加法器等。控制电路用于控制数据路径电路的操作。在第8章中，我们所说的控制电路是指有限状态机。

10.1　构件块电路

我们将举几个数字系统的示例，展示如何设计它们的数据路径电路和控制电路。这些例子用到一些前面几章介绍过的构件块。本章中用到的一些构件块将在下面描述。

10.1.1　有使能输入的触发器和寄存器

在许多使用D触发器的应用中，当有效时钟沿到来时刻，若能阻止已经存储在触发器中的数据发生变化是很有用的。我们曾经在图7-61中展示了如何在触发器前添加一个多路器来提供这样的能力。图10-1重新绘制了该电路。当$E = 0$时，触发器的输出不能改变，因为多路器把触发器的输出信号Q与D连接起来了。但是当$E = 1$时，多路器把外部输入信号R与D连接起来。还有一种办法，可以不使用图10-1所示的多路器，也能实现触发器的使能特性。该办法使用一个两输入的与门，该与门的一个输入连接到时钟信号，另外一个输入连接使能控制信号E，与门的输出连接到触发器的时钟。这个方法看来比使用多路器的方法简单些，但是我们将在10.3节看到，这样做在实际运行中会出现问题。在本章中，我们建议用多路器的方法而不主张用与门控制时钟的方法，来实现触发器的使能控制。

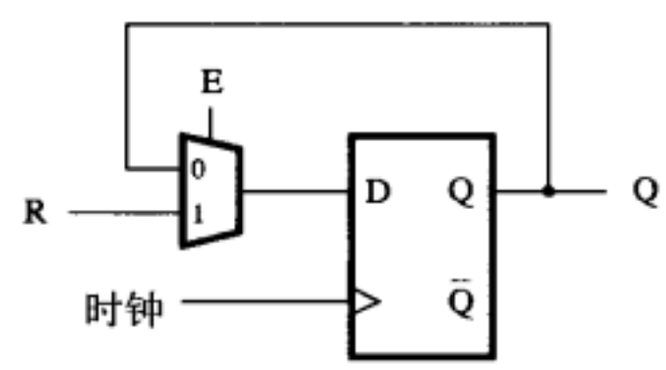


图10-1　带使能输入的触发器

```
module rege (R, Clock, Resetn, E, Q);
  input R, Clock, Resetn, E;
  output reg Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
      Q <= 0;
    else if (E)
      Q <= R;

endmodule
```

图10-2　带使能端的D触发器的Verilog代码

带异步复位输入和使能输入的D触发器的Verilog代码见图10-2所示。我们可以通过用E控制的n个二选一多路器，把使能功能扩展到n位寄存器。每个触发器

的多路器i，可以选择外部信号位R_i或者触发器的输出Q_i。带异步复位输入和使能输入的n位寄存器的Verilog代码见图10-3所示。

```
module regne (R, Clock, Resetn, E, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output reg [n-1:0] Q;

   always @(posedge Clock, negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else if (E)
         Q <= R;

endmodule
```

图10-3 带使能端的n位寄存器的Verilog代码

10.1.2 有使能输入的移位寄存器

通过使用使能输入信号E，就能够禁止移位寄存器的移位操作，这是很有用的。我们曾经在图7-19中展示：可以用多路器把一般的移位寄存器改造成具有并行置数能力的移位寄存器。图10-4展示了如何用附加的多路器，使移位寄存器添加使能的功能。如果并行加载控制信号$L = 1$，该寄存器就被并行加载。但是如果$L = 0$，那么附加的多路器只有在当$E = 1$时，才选择将新的数据一位一位地加载到触发器中。

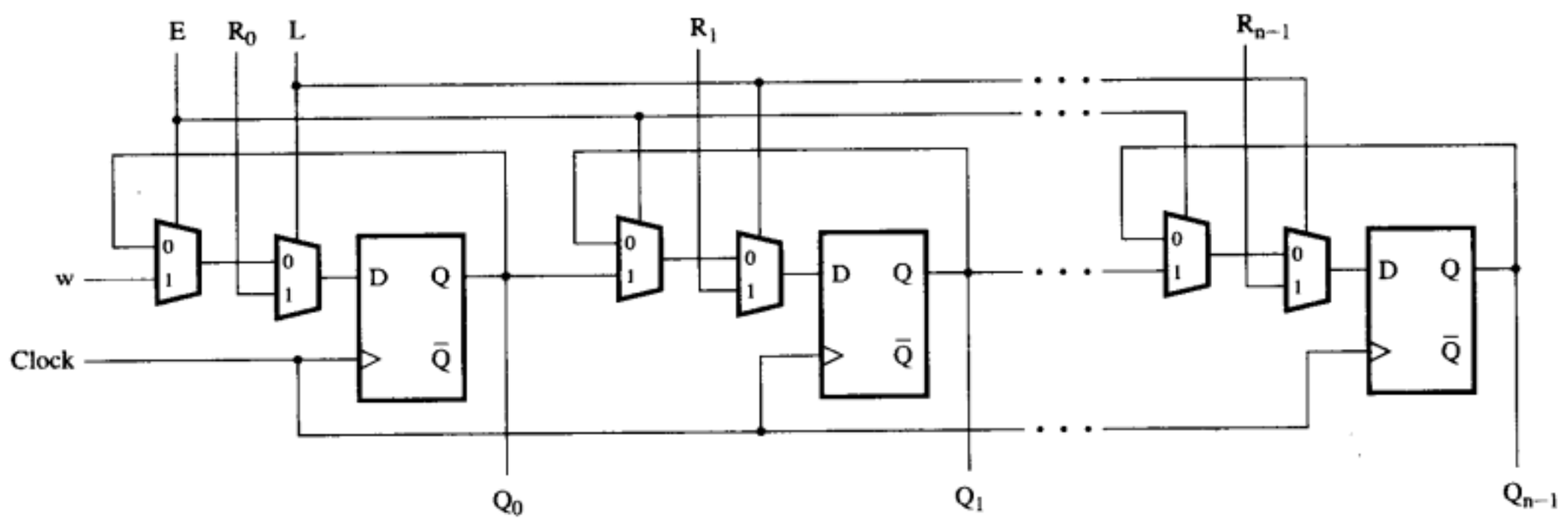


图10-4 具有并行加载和使能控制输入的移位寄存器

表示图10-4电路的Verilog代码见图10-5所示。当$L = 1$时，寄存器被从R输入的数据并行地加载。当$L = 0$并且$E = 1$时，在移位寄存器中的数据自右向左移位。（译者注：图上的左右和作者理解的左右不同，作者是从一般数据位MSB…LSB来理解左右的，准确的意思应该是从低位向高位逐位移位。）

```
module shiftlne (R, L, E, w, Clock, Q);
   parameter n = 4;
   input [n-1:0] R;
   input L, E, w, Clock;
   output reg [n-1:0] Q;
   integer k;

   always @(posedge Clock)
   begin
      if (L)
         Q <= R;
      else if (E)
      begin
         Q[0] <= w;
         for (k = 1; k < n; k = k+1)
            Q[k] <= Q[k-1];
      end
   end

endmodule
```

图10-5 自右向左移位的带使能控制的移位寄存器

10.1.3 静态随机存取存储器

我们已经介绍了几种类型的能够存储数据的电路。假设我们需要存储m个数据项，每个数据项由n位组成（其中m是一个很大的数字）。可行的方法之一是为每个数据项安排一个n位的寄存器。我们还需要设计控制电路来访问每个寄存器，以便把数据加载（写）入每个寄存器，或者把数据从中读出来。

当m是一个很大的数字的时候，用单个寄存器来存储数据是一个很笨拙的方案。较好的方案是利用*静态随机存取存储器*（SRAM）块。SRAM块是一个二维的SRAM单元的阵列，每个单元能存储一位信息。如果我们需要存储m个每个n位的项，就需要用一个$m \times n$ SRAM 阵列。$m \times n$被称为该SRAM 阵列的*纵横比*（aspect ratio）。（译者注：也称为存

储器深度和位宽比。）

SRAM单元类似于图7-3所示的存储单元。因为SRAM块可包含大量的SRAM单元，每个单元在集成电路芯片上占有的空间必须尽可能地小。因为这个原因，存储单元必须尽可能地少用晶体管。图10-6描绘了实际应用中普通的存储单元，它的操作如下。为了把数据存入单元，输入信号*Sel* 被设置成1，准备存储的数据值被放置在输入*Data*上。SRAM单元可以包含一个独立的反相数据的输入，在图中用灰色的晶体管表示。为了简单起见，我们假设在SRAM单元中不包括这个灰色的晶体管。在等待足够长的时间让数据通过两个非门的传播延迟反馈后，*Sel* 变成0。存储的数据就会永久保留在反馈回路中。可能会出现这样的问题：当*Sel* = 1时，*Data* 的值可能不同于反馈回路中的小的非门的输出值。因此由*Sel* 控制的晶体管会企图驱动存储单元到一个逻辑值，而该逻辑值可能正好与小非门的输出值相反。为了解决这个问题，在反馈回路上的非门必须用一个较小的（弱驱动）晶体管，以便它的输出可以被新的数据改写。

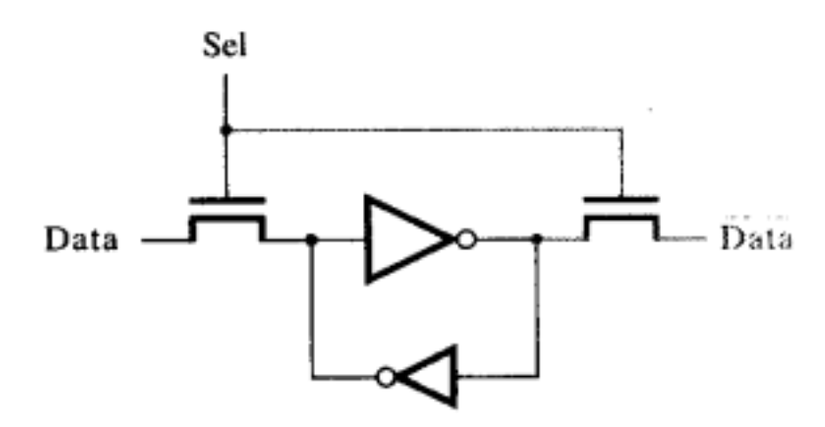


图10-6 一个SRAM单元

为了从存储数据的单元读出数据，我们只要让*Sel* = 1。在这种场合，数据节点没有被外部电路的任何值驱动，所以SRAM单元能够把存储的数据放到这个节点上。该数据信号通过一个缓冲器（图中没有画出），将SRAM块中存储的一个数据输出。

SRAM块包含SRAM单元阵列。图10-7显示了一个两行的阵列，每行包含两个单元。在阵列的每列中，单元的数据节点是连接在一起的。每一行*i*具有一个独立的选择输入Sel_i，那是用来读写该行存储单元的内容的。较大的阵列每行上有更多的单元连接到Sel_i，而且有更多的行。SRAM块中必须包含控制阵列中每行存储单元读写的电路。图10-8描绘了一个图10-7类型的$2^m \times n$ 阵列，该阵列具有一个译码器，用以驱动阵列中的每行的*Sel* 输入信号。译码器的输入叫做地址输入。这个术语来自于阵列中行的位置可以被认为是行的地址。该译码器有*m*条地址输入线，可以产生2^m个选择输出。如果写输入控制信号为1，那么在输入线$d_{n-1},\cdots,d_0$ 上的数据位将被存储到由地址选定的行存储单元中。如果读输入控制信号为1，那么由输入地址信号选定的行中存储的数据将会出现在输出线$q_{n-1},\cdots,q_0$ 上。在许多实际的应用中，数据的输入和输出是连接在一起的。因此写和读输入信号一定永远不能同时为1。

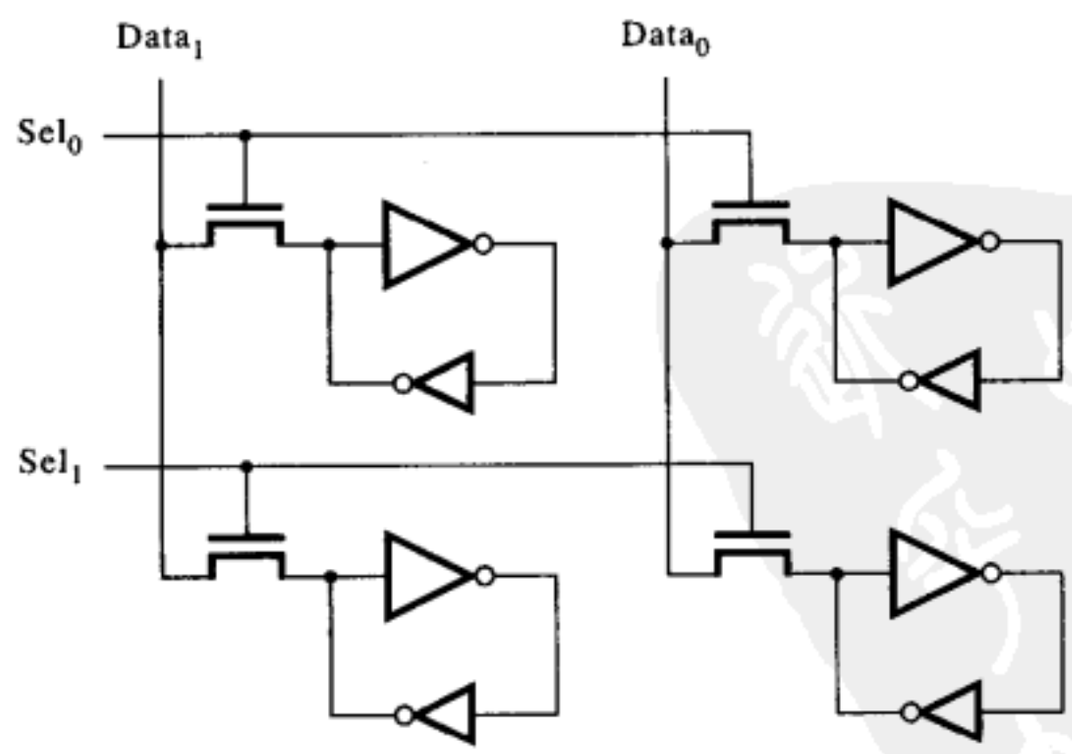


图10-7 一个由2 × 2 SRAM单元构成的阵列

存储器块的设计一直是研究开发的热门课题。我们只是描述了一种类型的存储器块的基

本操作。读者可以参考论述计算机体系结构的书籍，得到更多的信息[1,2]。

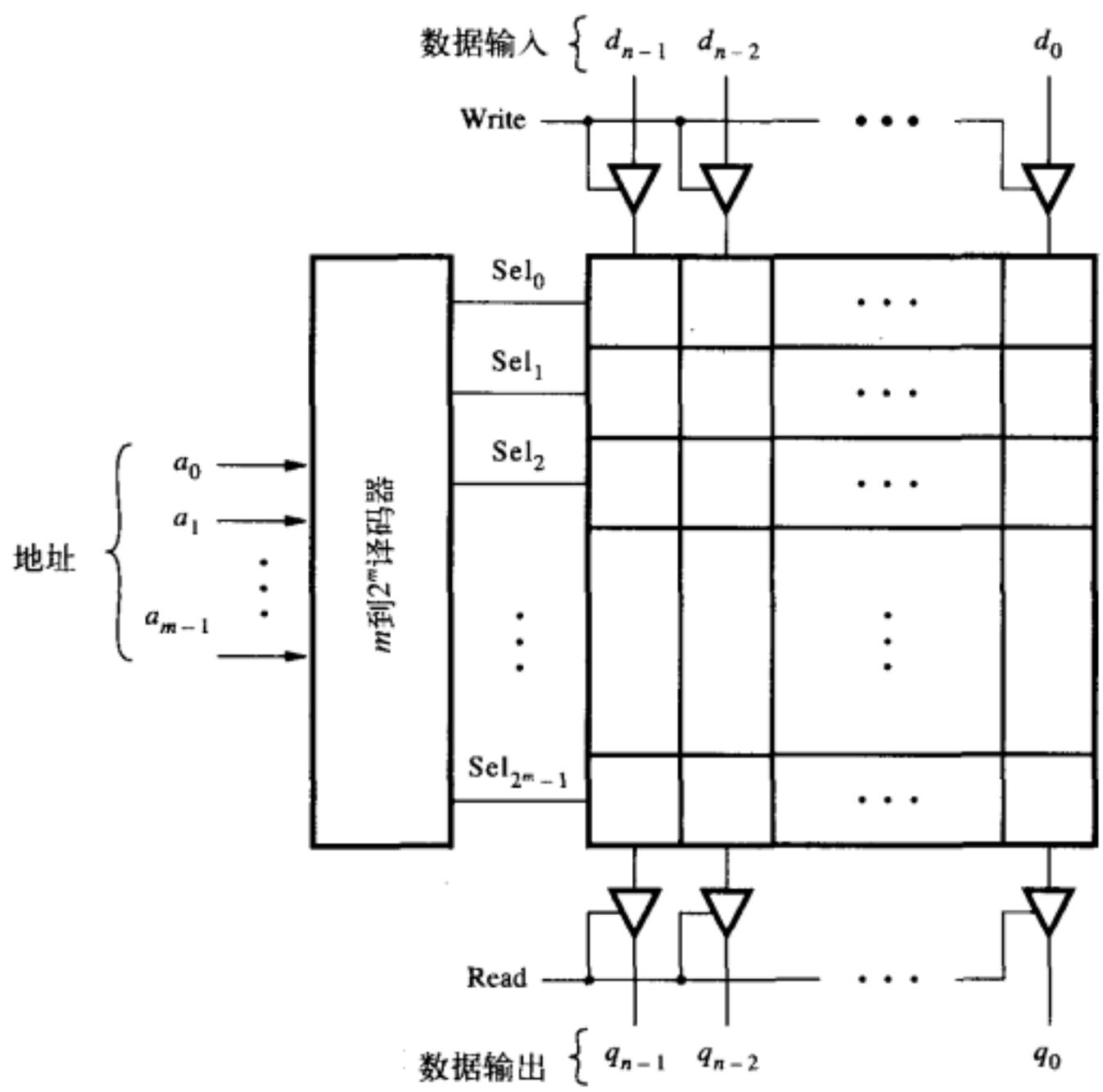


图10-8 一个由$2^m \times n$个单元构成SRAM块

10.1.4 可编程逻辑器件中的SRAM块

有些可编程逻辑器件内部包含一些SRAM块，SRAM是实现电路芯片中的一部分。有一种普通的芯片，其内部包含许多个SRAM块，每个SRAM块有4096个SRAM单元。该SRAM块可以根据正准备实现的设计需要被配置成不同的存储器深度和位宽比。用一个SRAM块可以实现从512×8 到4096×1的不同的存储器深度和位宽比，多个SRAM块可以组织起来构成更大的存储器阵列。为了在电路中包含SRAM块，设计者可以使用预先建立的模块，那些模块作为CAD工具的一部分，由一个库提供。

10.2 设计举例

我们在8.10节曾经介绍过算法状态机（ASM）图，并展示如何用算法状态机来描述有限状态机。算法状态机图也能被用来描述包括数据路径和控制电路在内的数字系统。我们将通过几个例子，用图示详细说明如何用算法状态机图来辅助数字系统的设计。

10.2.1 位计数电路

假设我们想对寄存器A中值为1的位的个数进行计数。图10-9展示了能够用来完成这个任务的一步一步的过程（也叫算法）的伪代码。这个伪代码假设了A存储在一个能自左向右移位的寄存器中。由该算法产生的结果存储在名为B的变量中。当A不再含有任何1的位时，即$A = 0$时，该算法便结束了。在循环的每次迭代过程中，如果A的最低位（LSB）是1，那么B就增加1；不然B保持不变。在

```
B = 0;
while A ≠ 0 do
    if a0 = 1 then
        B = B + 1;
    end if;
    Right-shift A;
end while;
```

图10-9 位计数器的伪代码

每次迭代结束的时候，A向右移位一位。

图10-10给出了表示图10-9算法的算法状态机图。代表起始状态$S1$的状态方块指定B被初始化为0。我们假设存在输入信号s，它用来表示被处理的数据是否被加载进入A，以便算法状态机的启动。标有s的菱形判断方块规定，只要$s = 0$，算法状态机一直保留状态$S1$。里面写着Load A的条件输出块表明：如果在状态$S1$时$s = 0$，那么A被从外部输入数据加载。

当s变为1时，算法状态机进入$S2$状态。在表示$S2$状态方块下的菱形判断方块检查A是否为0。如果$A = 0$，那么位计数操作完成了；因此算法状态机应该进入$S3$状态。如果$A \neq 0$，那么该有限状态机仍旧留在$S2$状态。算法状态机图最底下的菱形判断块检查a_0的值。如果$a_0 = 1$，那么B就增加1，在图中用标有$B \leftarrow B + 1$的两头弧形的块表示。如果$a_0 = 0$，那么B保持不变。在状态$S3$中，B保存了计数的结果，这就是A中值为1的位的个数。一个$Done$输出信号被置成1，表明算法的结束；算法状态机仍旧留在状态$S3$，直到s变为0。

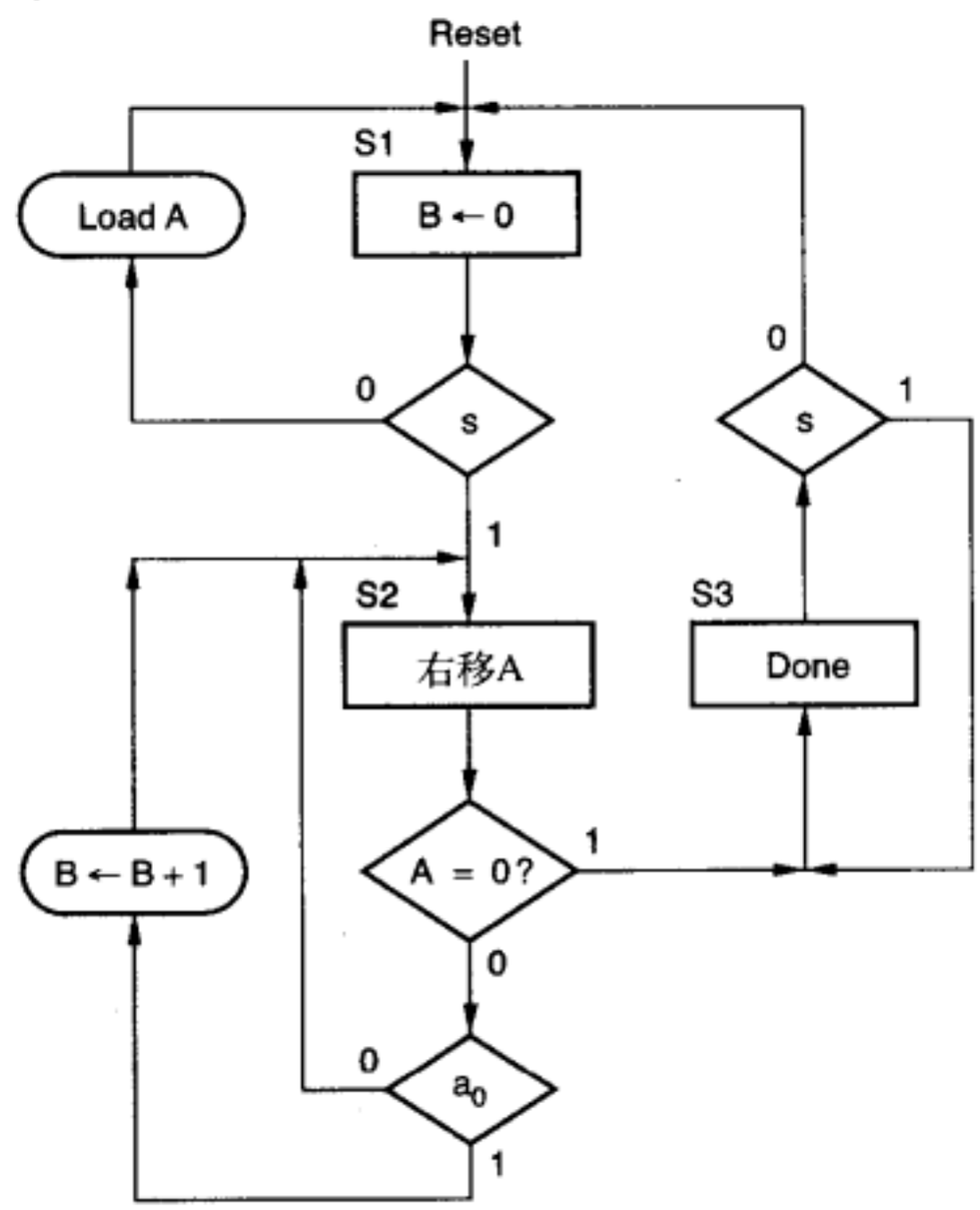


图10-10 图10-9中伪代码的算法状态机图

10.2.2 算法状态机图蕴涵的时序信息

在8.10节我们曾经说过，除了蕴涵时序信息外，算法状态机（ASM）图类似于传统的流程图。我们可以用位计数这个例子说明这个概念。考虑表示$S2$状态的ASM块，在图10-10中用阴影区表示。在传统的流程图中，当进入$S2$状态时，首先A的值会被右移。接着我们会检查A的值，如果A的最低位为1，我们会立刻给B加1。但是因为算法状态机图表示的是一个时序电路，A和B表示的是触发器的输出，因此它们的变化一定发生在有效时钟沿出现后。控制状态改变的时钟和控制着A和B改变的时钟是同一个时钟信号。因此在$S2$状态，判断A是否为0的菱形判断块和检查a_0值的菱形判断块，必须先检查A的位后，才能移位。如果$A = 0$，那么有限状态机在下一个有效时钟沿时刻将会变到$S3$状态（这个时钟沿也使A移位，但并未产生任何影响，因为A在此时早已经是0了）。另一方面，如果$A \neq 0$，那么有限状态机仍旧留在$S2$状态，不会进入$S3$状态。与此同时，A仍旧在移位，如果a_0的值为1，B继续增加1。这些时序问题将在图10-14中用图示详细地加以说明，图10-14展现的是表示实现该算法状态机图的电路的仿真结果。我们在下面的讨论中将展示该电路是如何设计的。

数据路径电路

通过检查位计数电路的算法状态机图，我们可以推断出实现数据路径必需的电路元件的类型。我们需要一个能对A实现自左向右移位的寄存器。这个移位寄存器还必须具有并行加载的能力，因为在$S1$状态的两头，弧形的条件输出块把数据加载到该寄存器。B需要一个计数器才行，而且B也需要具有并行加载能力，在$S1$状态时可以将其初始化为0。在数字系统的实际设计工作中，还需要设置$Reset$复位信号，这有两个目的：1）当电源接通时，对电路进行初始化；2）从错误中恢复电路功能。该状态机之所以能从$S3$状态回到$S1$状态，是由$s = 0$造成的；

因此我们不应该假设该*Reset*复位信号是用来对*B*计数器清零的。

图10-11所示是数据路径电路。移位寄存器的串行输入$w = 0$，因为该电路不需要串行输入。移位寄存器的加载控制和输入使能信号分别为*LA*和*EA*。移位寄存器的并行输入信号为*Data*，并行输出信号为*A*。用一个*n*位输入的或非门来测试*A*是否为0。该或非门的输出为*z*，若$A = 0$，则$z = 1$。请注意，表示*n*位输入的或非门在图中却只画了一条输入线，但在线上用短斜杠和*n*做了标记，表示该或非门有*n*个输入。该计数器的位数为$\log_2(n)$，并行输入连接着0，并行输出信号名为*B*。该计数器也有并行加载控制信号*LB*和输入使能控制信号*EB*。

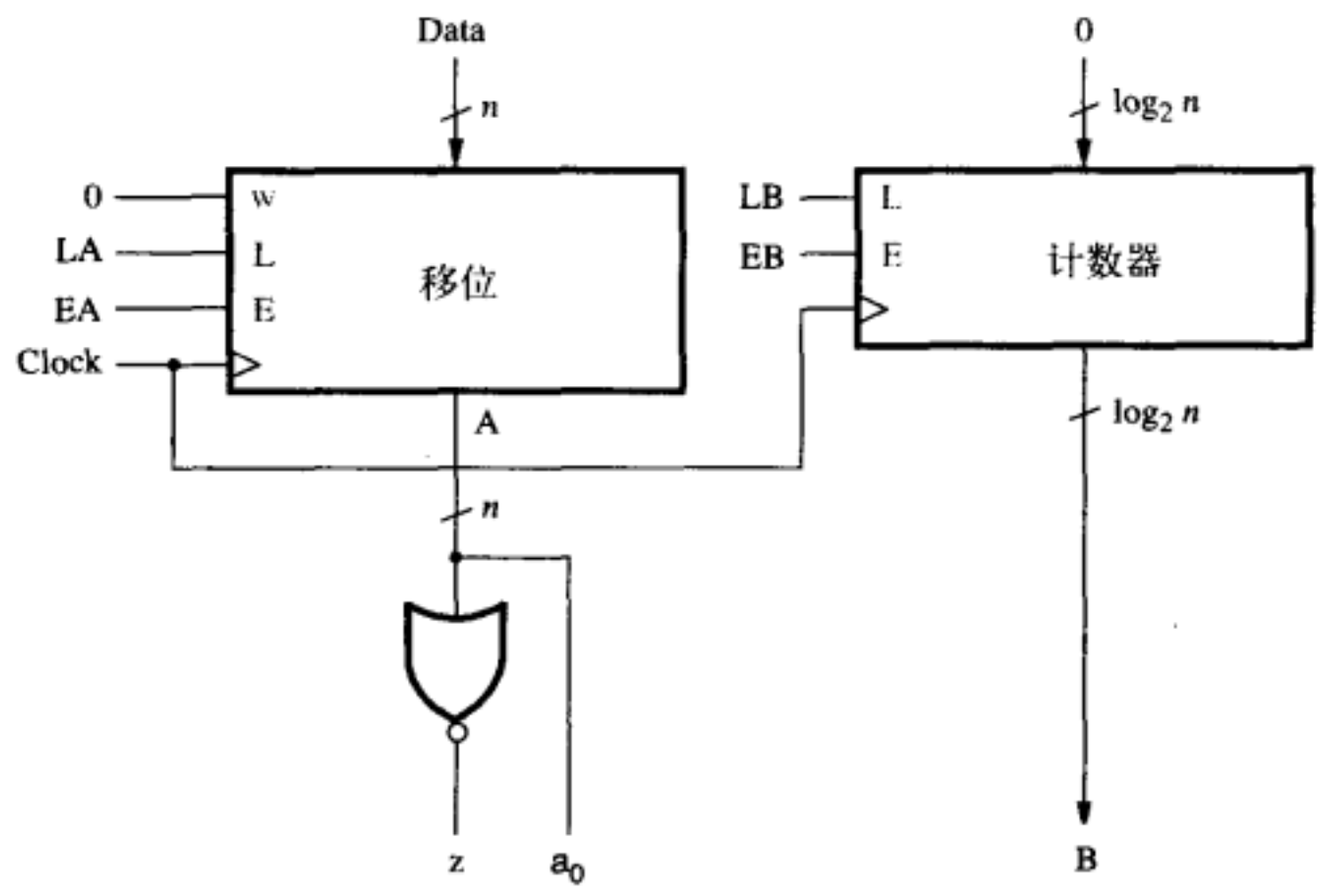


图10-11 图10-10中算法状态机图的数据路径

控制电路

为了方便起见，我们可以绘制第二张算法状态机图，图上只画有控制电路所必需的有限状态，见图10-12所示。该有限状态机的输入信号为*s*，a_0和*z*，并且产生的输出为*EA*，*LB*，*EB*和*Done*。在*S*1状态时，*LB*有效，0被并行地加载到计数器。注意这些控制信号，如*LB*，我们不写成$LB = 1$而只是写*LB*，表示此信号得到确认（有效）。假设当合法数据出现在移位寄存器的并行输入端时，由外部电路驱动$LA = 1$，于是该移位寄存器的内容，在*s*转变为1之前，就被初始化为并行的*Data*输入。在*S*2状态时，*EA*信号有效，使移位寄存器进行移位操作，而只有当$a_0 = 1$时，计数器*B*的使能信号才有效。

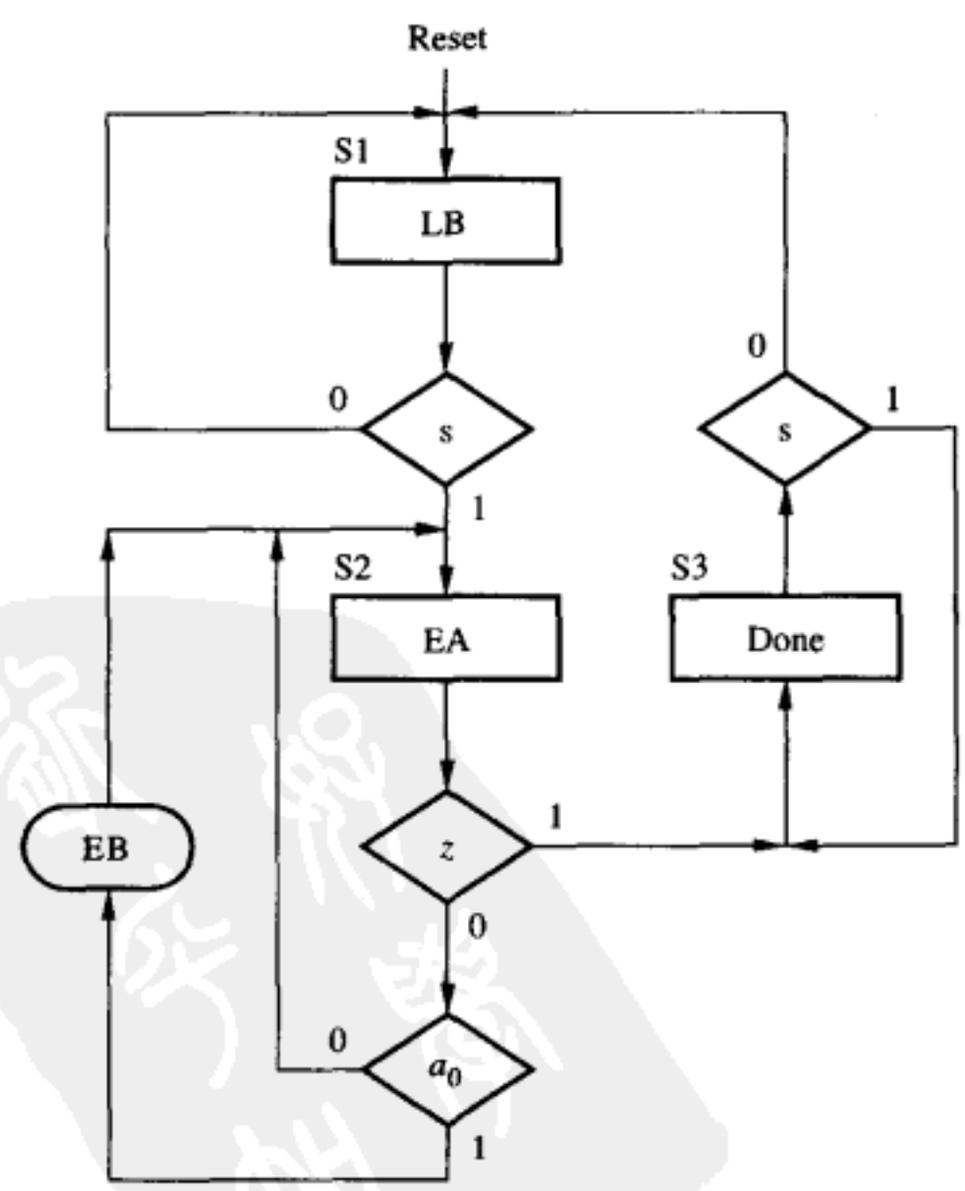


图10-12 位计数器数据路径电路的算法状态机图

Verilog 代码

位计数电路可以用Verilog代码来描述，见图10-13所示。我们把*A*定义为八位的向量，*B*为四位的向量。图10-12所示的算法状态机图可以直接转换为描述所要求的控制电路。信号*y*被用来表示有限状态机的当前状态，而*Y*表示有限状态机的下一个状态。该有限状态机用三个

always块描述：标有*State_table*的块指定状态的转移，标有*State_flipflop*的块表示触发器的状态，标有*FSM_output*的块指定每个状态中生成的输出信号。在*FSM_output*块的开始为每个输出信号指定了默认值，以后的每个输出值则由case语句指定。

第四个always块定义了实现*B*的正向计数器。移位寄存器*A*的实例引用放在代码的末尾。用缩减或非操作符定义了*z*信号。我们把图10-13所示的代码用芯片实现，并进行时序仿真。图10-14所示是为$A = 00111011$所做的仿真结果。电路复位后输入信号*LA*被设置成1，然后想处理的数据$(3B)_{16}$被放在*Data*输入上。当*s*改变为1时，下一个有效时钟沿，便使有限状态机把状态转变成*S*2。在这个*S*2状态，只要$a_0 = 1$，每个有效时钟沿都将使*B*增加1，然后移位*A*。当$A = 0$时，在下一个时钟沿将引起*FSM*改变至状态*S*3，其中*Done*设置为1，*B*有正确的结果$B = 5$。为了更彻底地检查电路看一看设计的电路是否正确，我们应该用不同的输入数据多试验几次。

```
module bitcount (Clock, Resetn, LA, s, Data, B, Done);
   input Clock, Resetn, LA, s;
   input [7:0] Data;
   output reg [3:0] B;
   output reg Done;
   wire [7:0] A;
   wire z;
   reg [1:0] Y, y;
   reg EA, EB, LB;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

   always @(s, y, z)
   begin: State_table
      case (y)
         S1: if (!s) Y = S1;
             else Y = S2;
         S2: if (z == 0) Y = S2;
             else Y = S3;
         S3: if (s) Y = S3;
             else Y = S1;
         default: Y = 2'bxx;
      endcase
   end

   always @(posedge Clock, negedge Resetn)
   begin: State_flipflops
      if (Resetn == 0)
         y <= S1;
      else
         y <= Y;
   end

... continued in Part b.
```

a)

```
   always @(y, A[0])
   begin: FSM_outputs
      // defaults
      EA = 0; LB = 0; EB = 0; Done = 0;
      case (y)
         S1: LB = 1;
         S2: begin
                EA = 1;
                if (A[0]) EB = 1;
                else EB = 0;
             end
         S3: Done = 1;
      endcase
   end

// datapath circuit

   // counter B
   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         B <= 0;
      else if (LB)
         B <= 0;
      else if (EB)
         B <= B + 1;

   shiftrne ShiftA (Data, LA, EA, 1'b0, Clock, A);
   assign z = ~| A;

endmodule
```

b)

图10-13 位计数器电路的Verilog代码

10.2.3 移位相加实现的乘法器

在图5-36中，我们曾经展示过两个无符号二进制数的乘法。该电路使用两个同样的二维阵列子电路，每个子电路包含一个全加器和一个与门。对于逻辑门较多的电路，这个解决方案也许并不合适。另外一种方法是利用一个移位寄存器结合一个加法器，类似用手工计算的

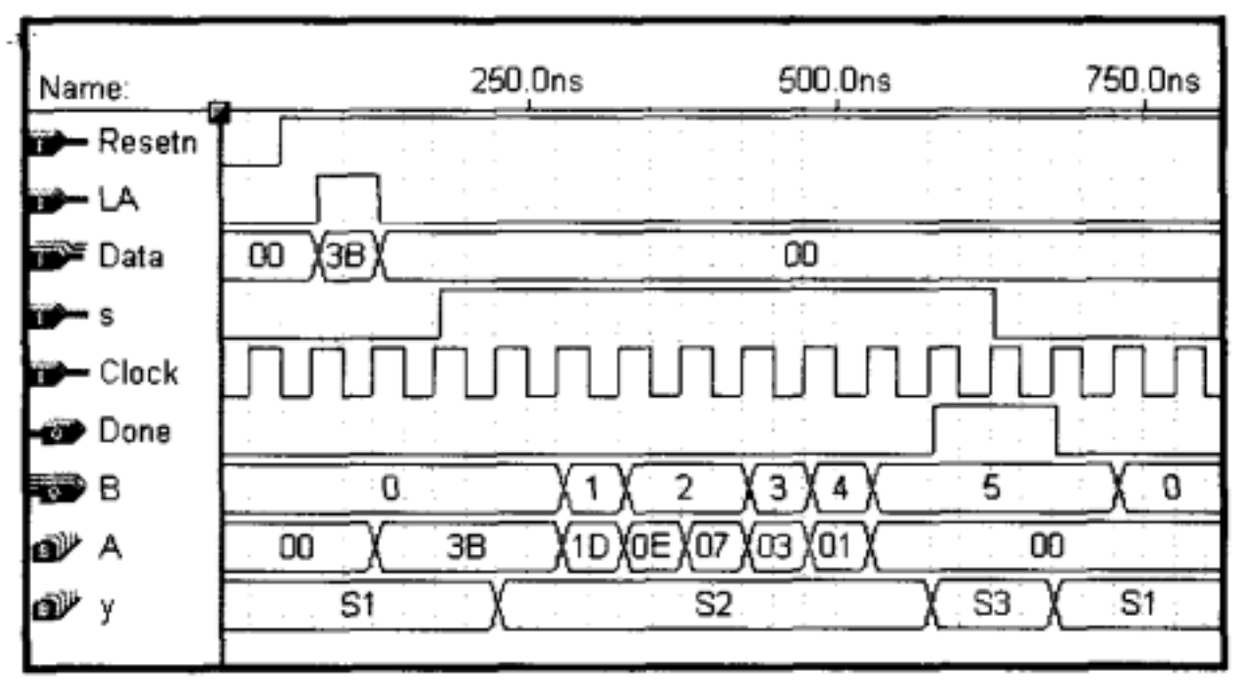
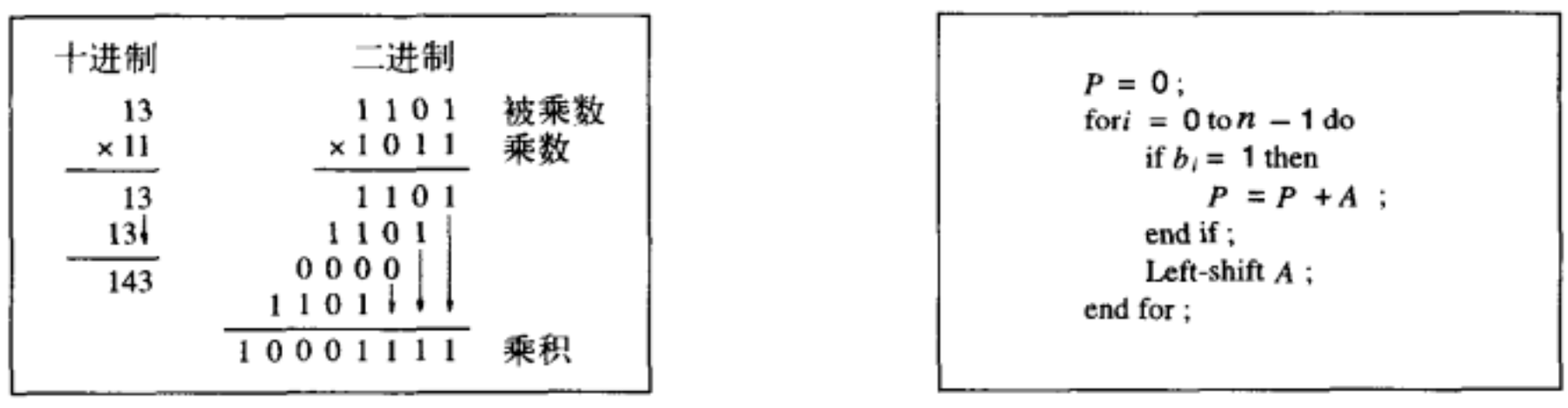


图10-14 位计数器电路的仿真结果

传统方法来实现乘法。图10-15a用图示详细地解释了用手工计算两个二进制数乘法的步骤。乘积是由一系列加法运算形成的。对于乘数中每个为1的i位，把被乘数左移i位，再相加就得到乘积。这个算法可以用图10-15b所示的伪代码描述，在代码中A是被乘数，B是乘数，P是乘积。



a) 手工方法　　b) 伪代码

图10-15 一种乘法算法

图10-16给出了表示图10-15b所示算法的算法状态机图。我们假设s输入信号用来控制乘法运算过程的开始。只要s为0，状态机继续保持在$S1$状态，可以自外部输入信号线加载A和B数据。在$S2$状态，我们测试B的最低有效位（LSB）的值，如果最低有效位的值为1，则把A与P相加；否则P保持不变。当B全部为0时，状态机进入$S3$状态，因为这时，P已经是最后的乘积了。当状态机处于$S2$状态时，每个时钟周期我们把A的值向左移一位，正如图10-15b中的伪代码规定的那样；我们把B的值向右移一位，以便在每个时钟周期的有效沿时刻，b_0可以被用来确定A是否应该与P相加。

数据路径电路

至此，我们可以定义数据路径电路了。为了实现A，我们需要一个自右向左的$2n$位的移位寄存器。一个$2n$位的寄存器放置P，而且还必须有一个使能输入，因为在$S2$状态 $P \leftarrow P + A$ 的赋值是在两头弧形的条件输出块内的。为了产生$P + A$ 需要有一个$2n$位的加法器。请注意，在$S1$状态，P 被0加载，只有在$S2$状态时，P 才被加法器的输出加载。我们不能假设复位信号被用来给P清0，因为从$S3$状态返回$S1$状态是基于输入信号s的，而不是基于复位输入信号的。因此对于每一个加到P的输入，二选一多路器是必需的，确定究竟选择0，还是从加法器输出的和（Sum）位。为了产生B，需要一个n位的自左向右移位的寄存器，还需要一个具有n 个输入端的或非门来测试B是否为0。

图10-17展示了数据路径电路，对移位寄存器的控制信号做了标记。移位寄存器的输入数据（保持A ）为$DataA$。因为该移位寄存器有$2n$位，高n位数据输入线连接到0。单个多路器符

号的输出连接到保存P的寄存器。这单个符号实际上代表了$2n$个二选一多路器，它们的控制端都连接到$Psel$信号。

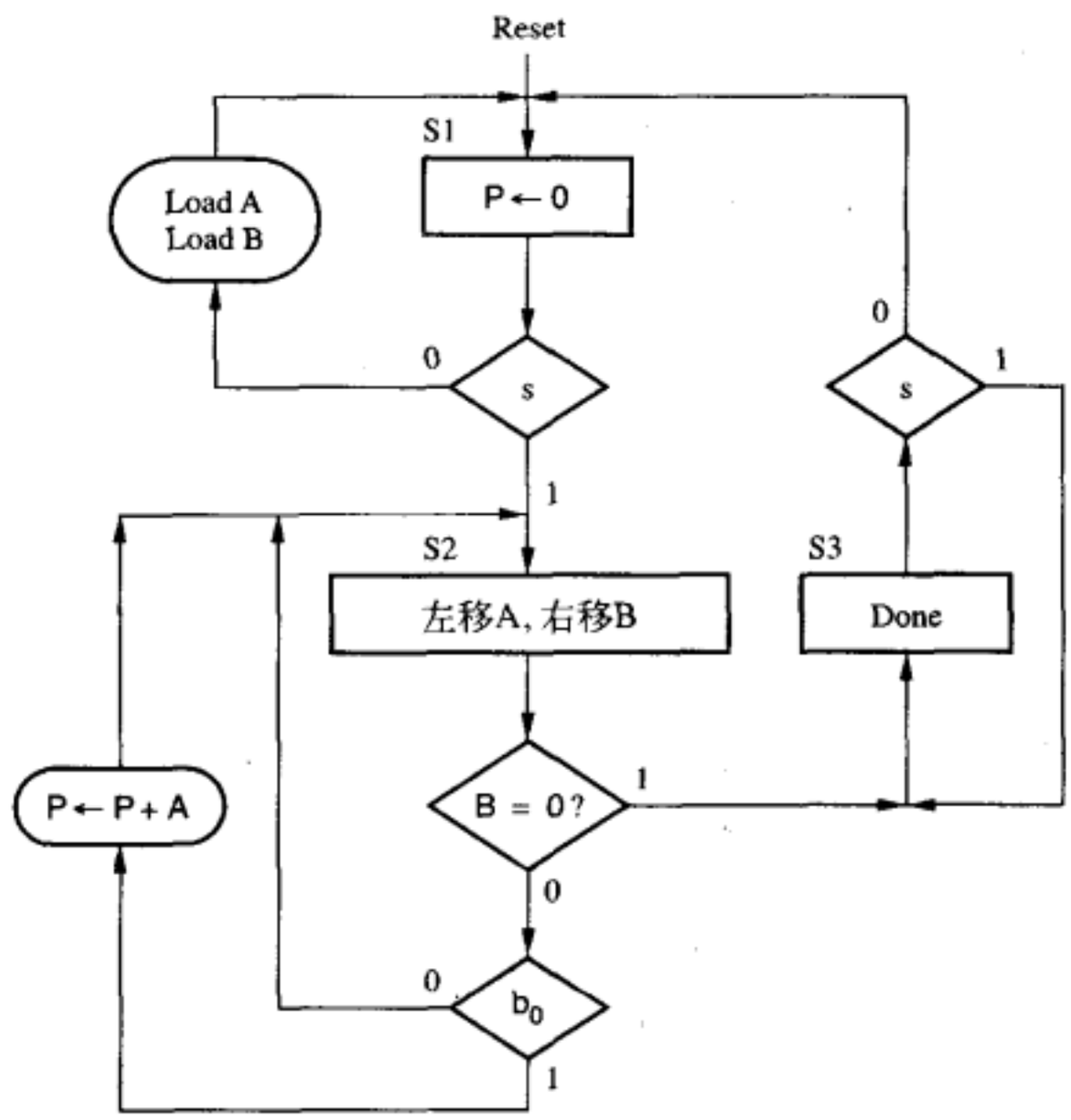


图10-16 乘法器的算法状态机图

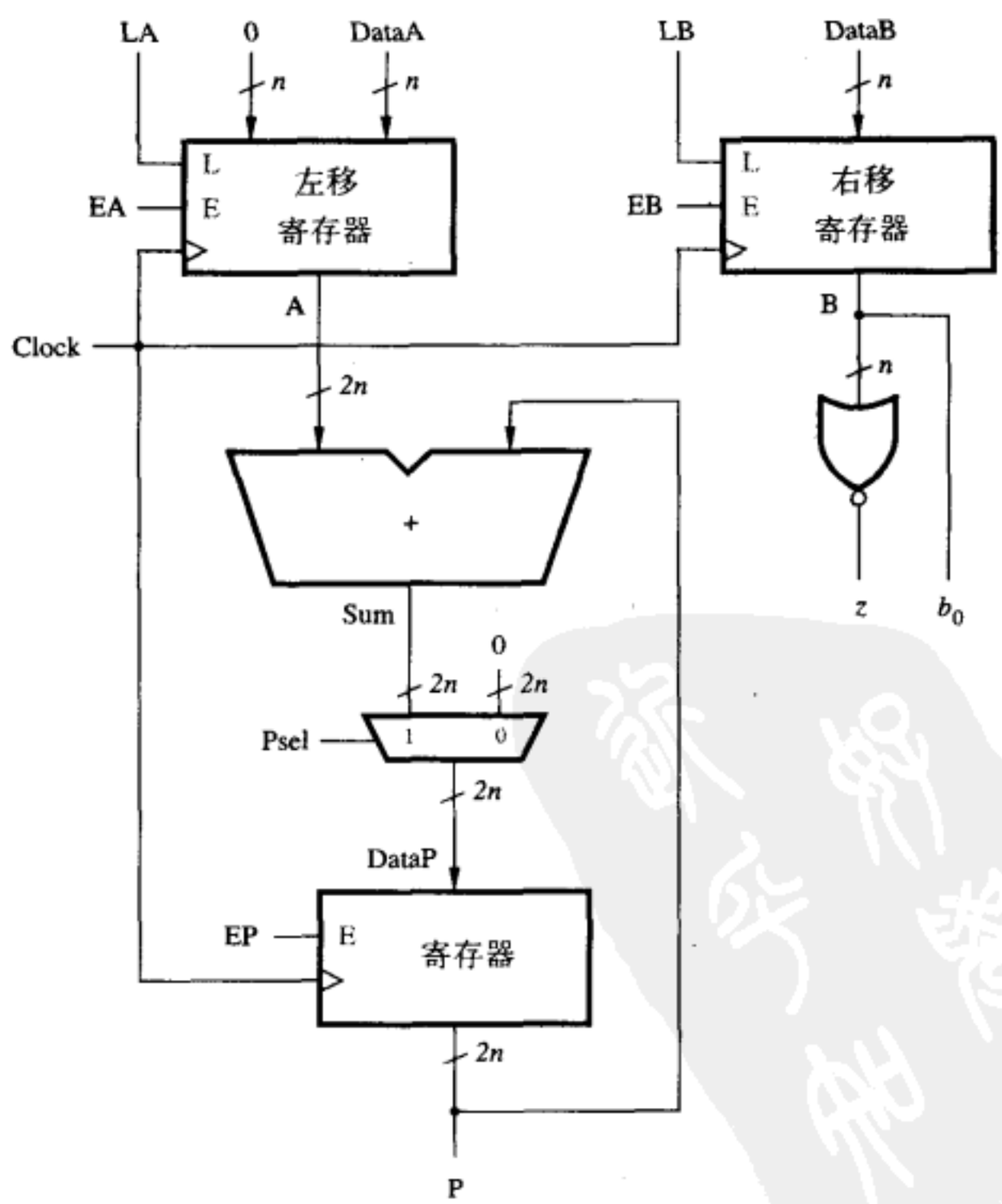


图10-17 乘法器的数据路径电路

控制电路

图10-18所示是表示乘法器所需控制信号的算法状态机图。在状态$S1$，设置$Psel$为0，EP为有效，所以寄存器P被清零。当$s = 0$时，可通过外部电路控制的LA和LB信号将输入的并行数据加载到移位寄存器中。当$s = 1$时，状态机进入$S2$状态，此时，$Psel$置为1，并允许（使能）A和B的移位。若$b_0 = 1$，则P的使能信号有效。当$z = 1$时，状态机进入状态$S3$，只要$s = 1$，状态便一直保留在$S3$，并且把$Done$设置成1。

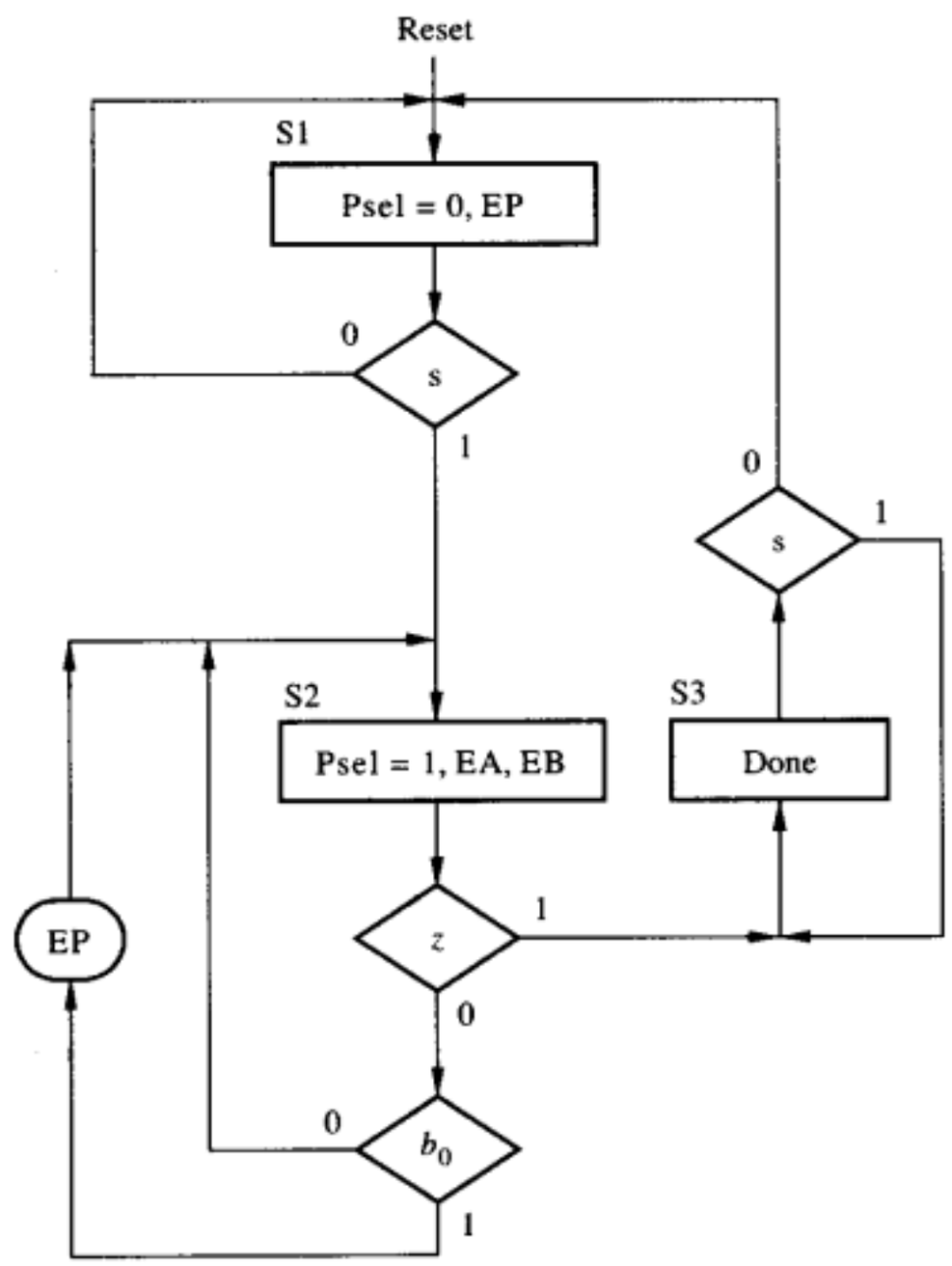


图10-18 该乘法器控制电路的算法状态机图

Verilog 代码

该乘法器的Verilog代码见图10-19所示。A和B的位数用参数n设置。对于$2n$位宽的寄存器，位的个数被设置成$n + n$。通过修改参数的值，该代码可以用于任意规模的乘法器。标有$State_table$和$State_flipflop$的两个always块，分别定义了状态的转移和状态触发器。控制电路的输出由标有FSM_output的always块指定。并行输入数据的移位寄存器A是$2n$位宽的，而$DataA$只有n位宽。因此，拼接操作{{n{1'b0}}, DataA}被用来在$DataA$的高位预先添加n个0，以加载进移位寄存器。寄存器P需要的多路器是用循环语句定义的，该循环语句定义了$2n$个二选一多路器。图10-20给出了由这个代码生成的电路的仿真结果。在电路复位后，LA和LB被设置成1，被乘数放在$DataA$和$DataB$的输入线上。s被设置成1之后，有限状态机（y）转变成$S2$状态，继续保留该状态，直到$B = 0$。在$S2$状态中的每个时钟周期，A向左移动一位，B向右移动一位。在$S2$状态的三个时钟周期中，对应于B中三个为1的位，A的内容被加到P中。当$B = 0$时，该有限状态机进入$S3$状态，P中包含了正确的乘积值：$(64)_{16} \times (19)_{16} = (9C4)_{16}$。这个数对应的十进制数为$100 \times 25 = 2500$。

```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
   parameter n = 8;
   input Clock, Resetn, LA, LB, s;
   input [n-1:0] DataA, DataB;
   output [n+n-1:0] P;
   output reg Done;
   wire z;
   reg [n+n-1:0] DataP;
   wire [n+n-1:0] A, Sum;
   reg [1:0] y, Y;
   wire [n-1:0] B;
   reg EA, EB, EP, Psel;
   integer k;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;
```

图10-19 乘法器电路的Verilog代码

```
always @(s, y, z)
begin: State_table
   case (y)
      S1:  if (s == 0) Y = S1;
           else Y = S2;
      S2:  if (z == 0) Y = S2;
           else Y = S3;
      S3:  if (s == 1) Y = S3;
           else Y = S1;
      default: Y = 2'bxx;
   endcase
end

always @(posedge Clock, negedge Resetn)
begin: State_flipflops
   if (Resetn == 0)
      y <= S1;
   else
      y <= Y;
   end
always @(s, y, B[0])
begin: FSM_outputs
   // defaults
   EA = 0; EB = 0; EP = 0; Done = 0; Psel = 0;
   case (y)
      S1:  EP = 1;
      S2:  begin
              EA = 1; EB = 1; Psel = 1;
              if (B[0]) EP = 1;
              else EP = 0;
           end
      S3:  Done = 1;
   endcase
end
   end

//datapath circuit

   shiftrne  ShiftB (DataB, LB, EB, 1'b0, Clock, B);
      defparam  ShiftB.n = 8;
   shiftlne  ShiftA ({{n{1'b0}}, DataA}, LA, EA, 1'b0, Clock, A);
      defparam  ShiftA.n = 16;

   assign   z = (B == 0);
   assign   Sum = A + P;

   // define the 2n 2-to-1 multiplexers
   always @(Psel, Sum)
      for (k = 0; k < n+n; k = k+1)
         DataP[k] = Psel ? Sum[k] : 1'b0;

   regne  RegP (DataP, Clock, Resetn, EP, P);
      defparam RegP.n = 16;

endmodule
```

图10-19 （续）

该电路产生最后乘积所需时钟周期的数目是由*B*中最左边那个为1的数字决定的。有可能减少所需要的时钟个数，但是*A*和*B*移位寄存器必须复杂些才行。如果*B*中最右面的两位都为0，那么*A*和*B*移位寄存器可以在一个时钟周期内移动两位。同样，如果*B*中最低的三位都为0，那么可在一个时钟周期内移动三位，以此类推。一次可以移动几位的移位寄存器可以用桶状移

位寄存器建造。我们给读者留一个家庭作业：把该乘法器改用桶状移位寄存器实现。

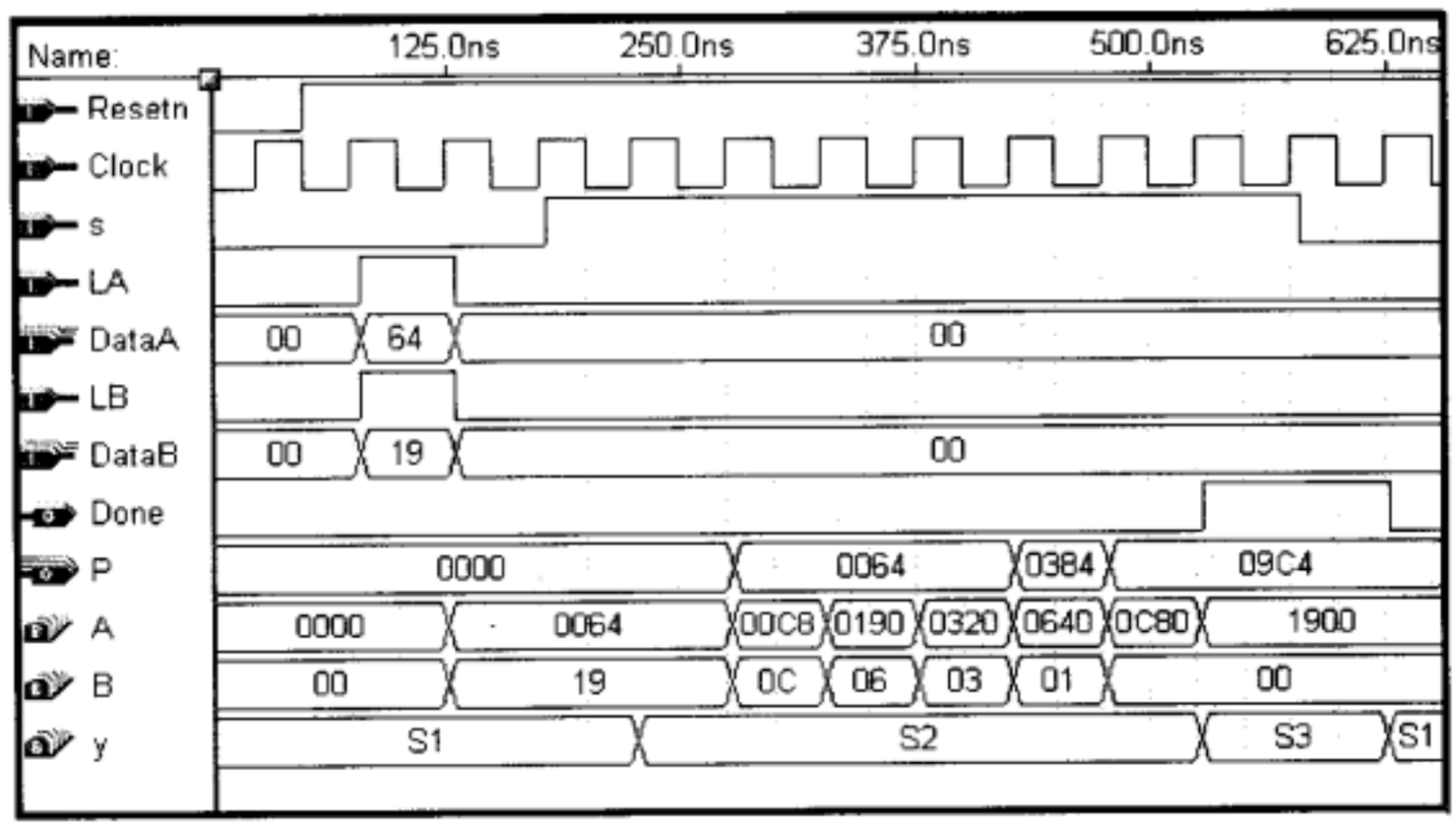
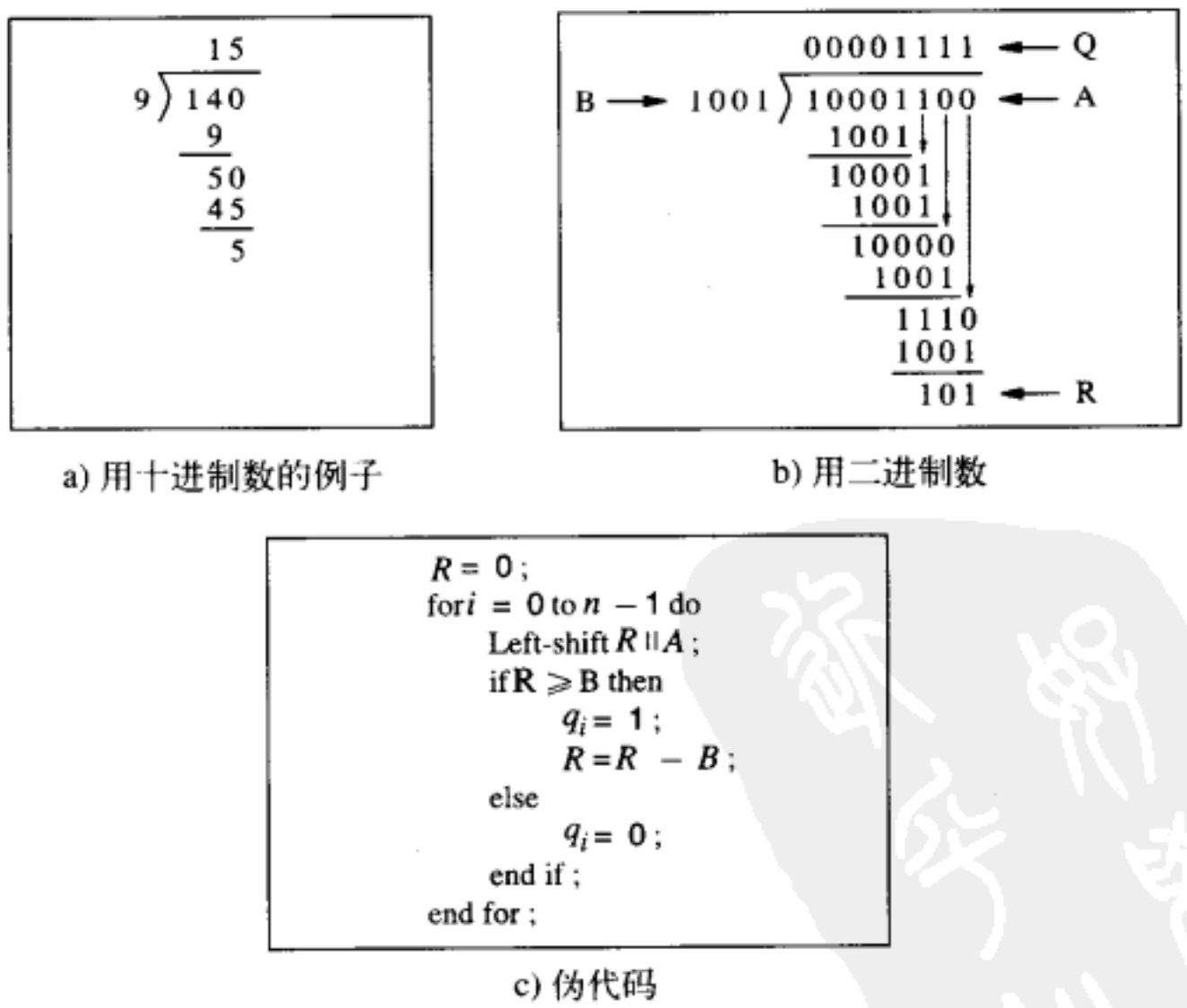


图10-20 乘法器电路的仿真结果

10.2.4 除法器

前面的例子实现了传统的用手工完成乘法运算的电路。在本例中，我们将设计一个能实现传统的用手工完成除法运算的电路。图10-21a 给出了一个用手工进行长除运算（long-hand division）的例子。除法运算的第一步是试图将除数9除以被除数的第一个数字1，发现不够除。接着，我们将除数9除以14，得到商的第一位数字是1。接着进行减法：14−9 = 5，得到被除数的最后一个数字0，形成50，然后得到商的第二个数字是5。余数是50−45 = 5，因此得到商是15。用二进制数字，如图10-21所示，所涉及的运算步骤完全一致，只是更加简单，商的每个数字只能为0或1。



a) 用十进制数的例子

b) 用二进制数

c) 伪代码

图10-21 除法运算

给定两个无符号n位的数A和B，我们想要设计一个电路，该电路能产生n位的输出Q和R，其中Q是A/B的商，R是余数。图10-21b所示步骤可以通过A中的数字向左移位，一次一位地进

入移位寄存器R实现。每次移位操作后，比较R与B。如果$R \geqslant B$，则在商的某一适当位放上一个1，然后从R中减去B；否则在商的某一适当位放上一个0。用伪代码描述这一个算法如图10-21c所示。$R \parallel A$这个符号用来表示一个$2n$位的移位寄存器，该移位寄存器的最左边n位用的是R寄存器，最右边n位用的是A寄存器。

图10-15b乘法器的伪代码在每次循环迭代时检查数字位b_i。在图10-16的算法状态机图中，我们将B向右移位，使得b_0总包含所需的数字。同样在长除运算的伪代码中，每次循环迭代导致数字q_i为1或为0。完成这件事的直接方法是在每次循环迭代时把1或者0移位进入Q的最低位。表示除法电路的算法状态机图见图10-22所示。信号C代表一个计数器，该计数器在起始状态$S1$时被初始化为$n-1$。在$S2$状态，R和A两个移位寄存器一起向左移一位，然后进入$S3$状态，如果$R \geqslant B$，则从R中减去B。当$C=0$时，状态机便进入$S4$状态。

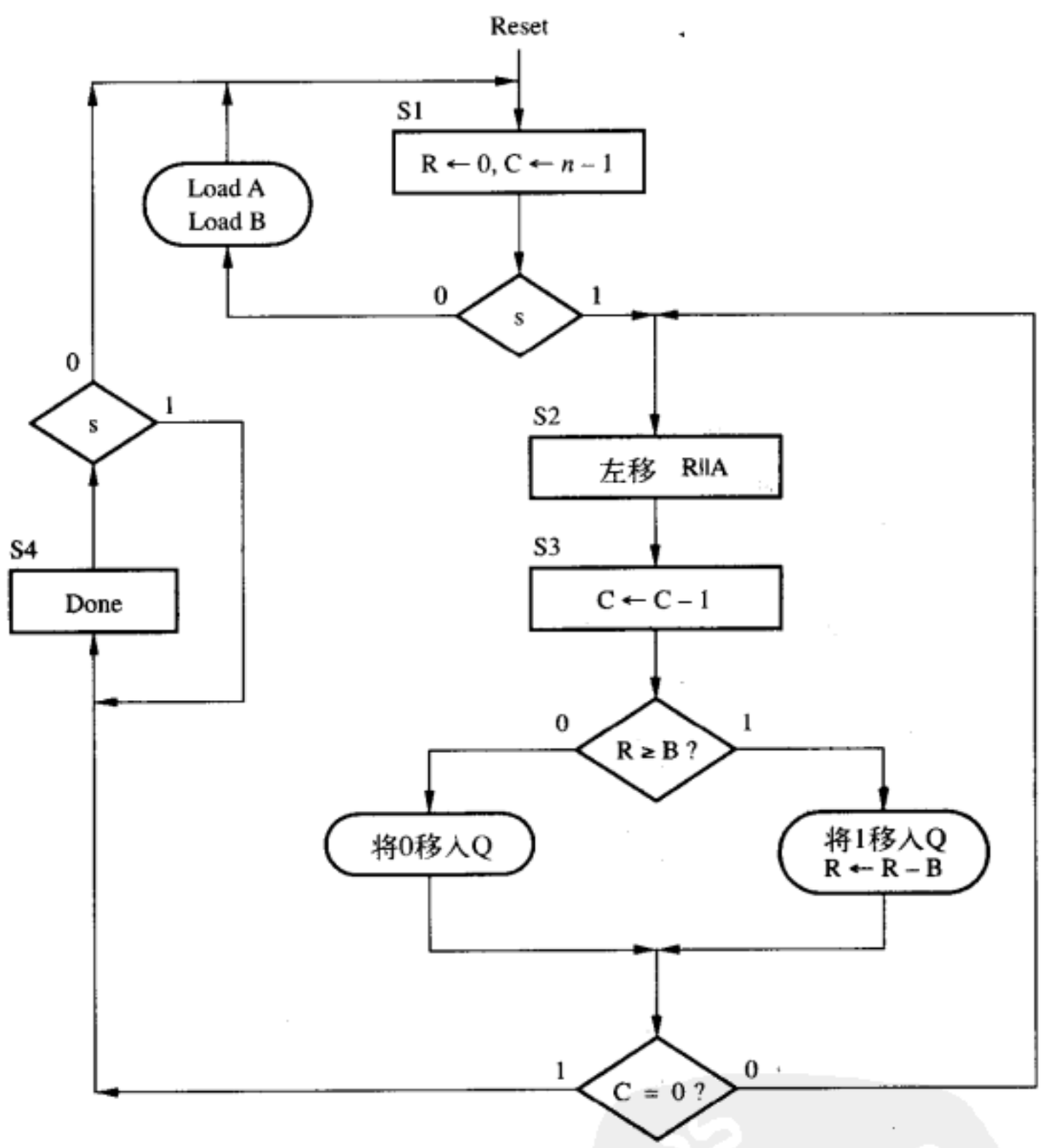


图10-22 除法器电路的算法状态机图

数据路径电路

我们需要n位移位寄存器，可以为A，R和Q做自右向左的移位。还需要一个n位寄存器可以寄存B的值，另外还需要一个减法器，可以计算$R-B$的值。我们可以采用一个加法器模块，在该加法器中，若将进位输入设置成1，则B就取反。若条件$R \geqslant B$为真，则该加法器模块的进位输出C_{out}的值为1。因此进位输出C_{out}可以被连接到保存Q的移位寄存器的串行输入，以便在$S3$状态时，可以将进位输出C_{out}逐位移位进入Q。因为在$S1$状态，R被加载为0，在$S3$状态，从加法器的输出需要一个多路器将并行数据输入到R中。图10-23描绘了数据路径电路。请注意还需要一个递减计数器来实现C，以及当$C=0$时，输出为1的或非门，这些都没有在电路图中画出。

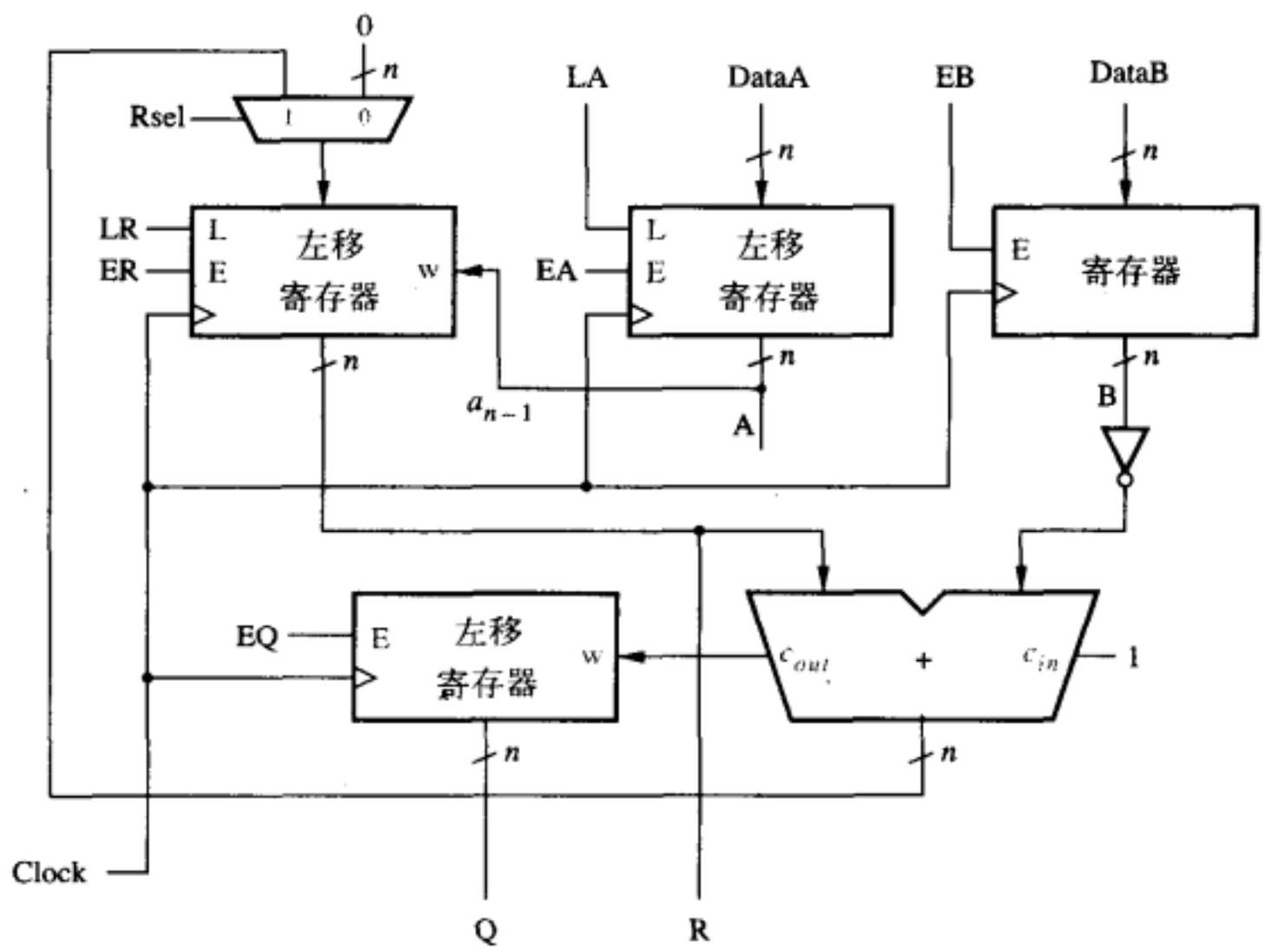


图10-23 除法器的数据路径电路

控制电路

图10-24所示是除法器的算法状态机图，该算法状态机图只显示了除法器进行运算所必需的控制信号。在$S3$状态，C_{out}的值决定了加法器输出的和值是否被加载到R中。Q的移位使能信号在$S3$状态下有效。我们不必指定加载到Q的是0还是1，因为在数据路径电路中C_{out}已被连接到Q的串行输入。我们把编写Verilog代码的任务留给读者作为练习，该Verilog代码表示了图10-24的算法状态机图和图10-23的数据路径电路。

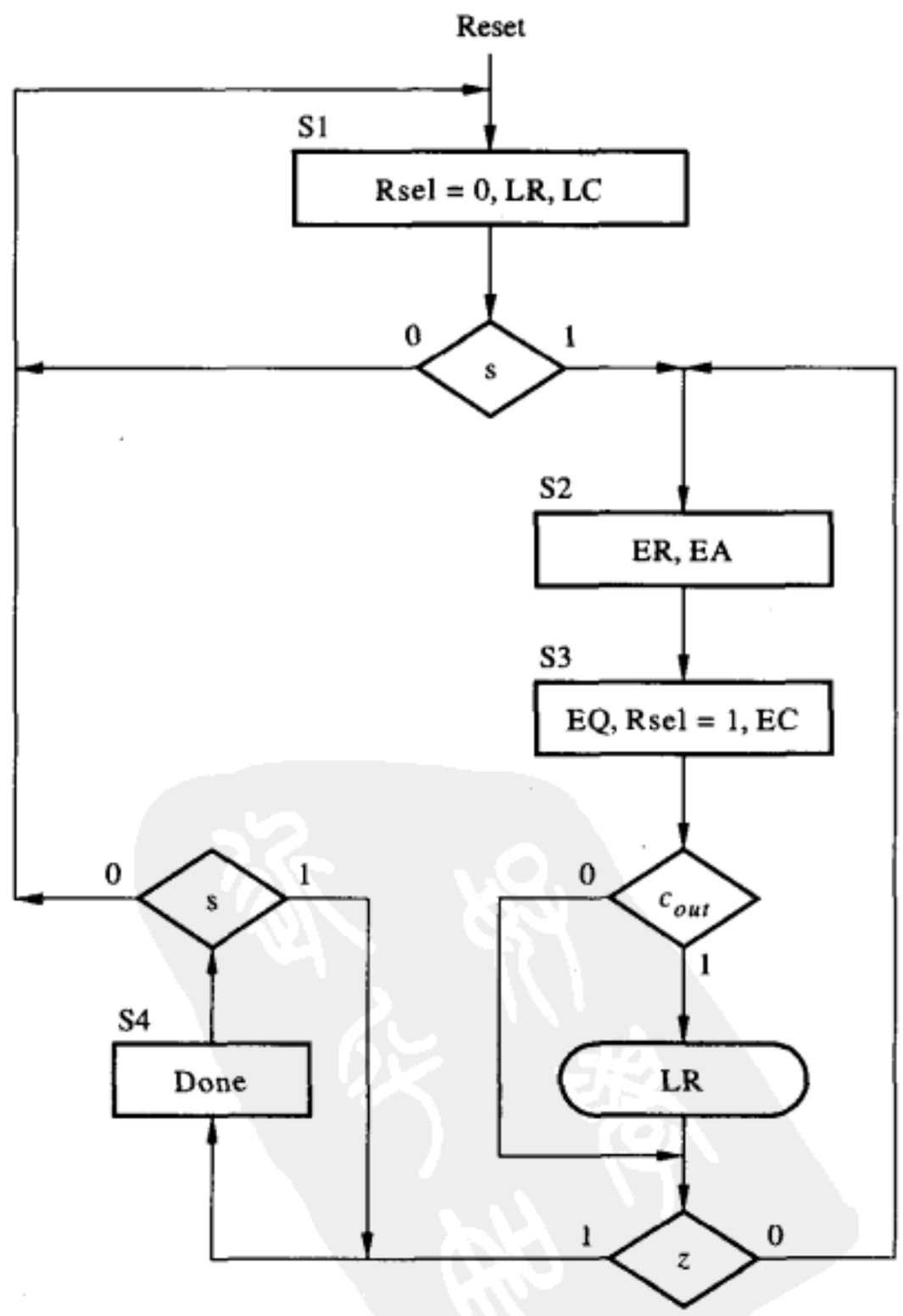


图10-24 除法器控制电路的算法状态机图

除法器电路的改进

用图10-24的算法状态机图，会使电路在$S2$和$S3$状态下，循环$2n$个时钟周期。若能将它们合并成一个状态，则所需的时钟周期数便会降至n。在状态$S3$，若$C_{out}=1$，则将加法器的输出（即减的结果）从加法器加载到R，并且（假设$z=0$）转到状态$S2$。在状态$S2$我们让R（和A）左移。为了将$S2$和$S3$合并成一个新的状态$S2$，我们需要将加法器的输出置于R最左边的那些位，同时将A的最高有效位移入R的最低有效位。这一步可以通过为R的最低有效位单独设置一个触发器来实现。设这个触发器的输出为rr_0。在状态$S1$，

当$s = 0$时，该触发器被初始化为0。否则，该触发器被置为A的最高有效位。在状态$S2$，若$C_{out} = 0$，则R左移，并将rr_0移入R。但若$C_{out} = 1$，则就从加法器的输出端并行将数据加载到R中。

图10-25展示了如何用n个时钟周期完成图10-21b给出的除法运算。图中的表格给出了除法过程中每一步R，rr_0，A和Q的值。在图10-23所示的数据路径电路中，我们为Q单独设置了一个移位寄存器。该寄存器其实并不需要，因为商的数字可以移入寄存器A的低位。在图10-25中被移入寄存器A的Q的数字用浅色标出。表的第一行表示将初始值读进寄存器A（和B）中，并将R和rr_0清0。在第二行（标作时钟周期0），黑色斜箭头表示A的最高位（1）被移入rr_0。此时$R \parallel rr_0$中的数为000000001，比B（1001）小。在时钟周期1，rr_0被移入R，A的最高位移入rr_0。同时，一个0被移入Q（A）的最低位。$R \parallel rr_0$中的数为000000010，仍比B（1001）小。因此，在时钟周期2进行了与时钟周期1同样的操作。时钟周期3、4也一样。此时$R \parallel rr_0$中的数为000010001，由于它比B大，在时钟周期5减法000010001−1001 = 00001000的结果载入R。A的最高位（1）仍被移入rr_0，并且把一个1移入Q。在时钟周期6、7、8中，$R \parallel rr_0$中的数都比B大，因此在这三个周期中，减法$R \parallel rr_0 - B$的结果都被加载到R中，1移入Q。在时钟周期8之后，便得到正确的运算结果：$Q = 00001111$和$R = 00000101$。rr_0不是最后结果的一部分。

B → 1001) 10001100 ← A

	时钟周期	R	rr_0	A/Q
	Load A, B	0 0 0 0 0 0 0 0	0	1 0 0 0 1 1 0 0
0	Shift left	0 0 0 0 0 0 0 0	1	0 0 0 1 1 0 0 0
1	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 0 1	0	0 0 1 1 0 0 0 0
2	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 1 0	0	0 1 1 0 0 0 0 0
3	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 1 0 0	0	1 1 0 0 0 0 0 0
4	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 1 0 0 0	1	1 0 0 0 0 0 0 0
5	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	1	0 0 0 0 0 0 0 1
6	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	0	0 0 0 0 0 0 1 1
7	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 1 1	0	0 0 0 0 0 1 1 1
8	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 0 1	0	0 0 0 0 1 1 1 1

图10-25 用n(=8)个周期完成除法的例子

如图10-26所示的算法状态机图给出了改进后的除法器电路所需的控制信号的值。信号ER0用来控制寄存器rr_0。当ER0=0时，0被加载到触发器；当ER0为1时，A的最高位被加载到触发器。在状态$S1$，若$s = 0$，则LR值为有效，以便将R初始化为0。寄存器A和B可以接受外部输入的数据。当s变为1时，状态机转入状态$S2$，同时$R \parallel R0 \parallel A$左移。在状态$S2$，若$c_{out} = 1$，则加法器的结果将并行加载到$R$中。同时$R0 \parallel A$左移（这种情况下$rr_0$不移入$R$）。若$c_{out} = 0$，则$R \parallel R0 \parallel A$左移。算法状态机图展示了如何控制寄存器的并行加载以及使能信号，以完成期望的操作。

改进后的除法器数据路径电路如图10-27所示。正如在图10-25中讨论过的那样，商Q的数字被移入寄存器A。注意，加法器模块有两个n位的输入数据，其中之一是由寄存器R的低$n-1$位与rr_0拼接在右边而组成的。

Verilog代码

图10-28列出了描述改进后的除法器电路的Verilog代码。参数n表示操作数的位数。正如前面几个例子一样，*State_table*、*State_ flipflops*和*FSM_output*三个always块描述了控制电路。在代码的最后部分采用实例引用的方法，描述了数据路径电路中的移位寄存器和计数器。图10-25中的信号rr_0在代码中用信号R0表示。该信号由*muxdff*部件输出，描述该子电路的代码见图7-52所示。注意产生和（Sum）信号的加法器有一个输入被定义为R和R0拼接而成。R输入所需的多路器用*DataR*信号表示。这个多路器在代码的最后一句定义。

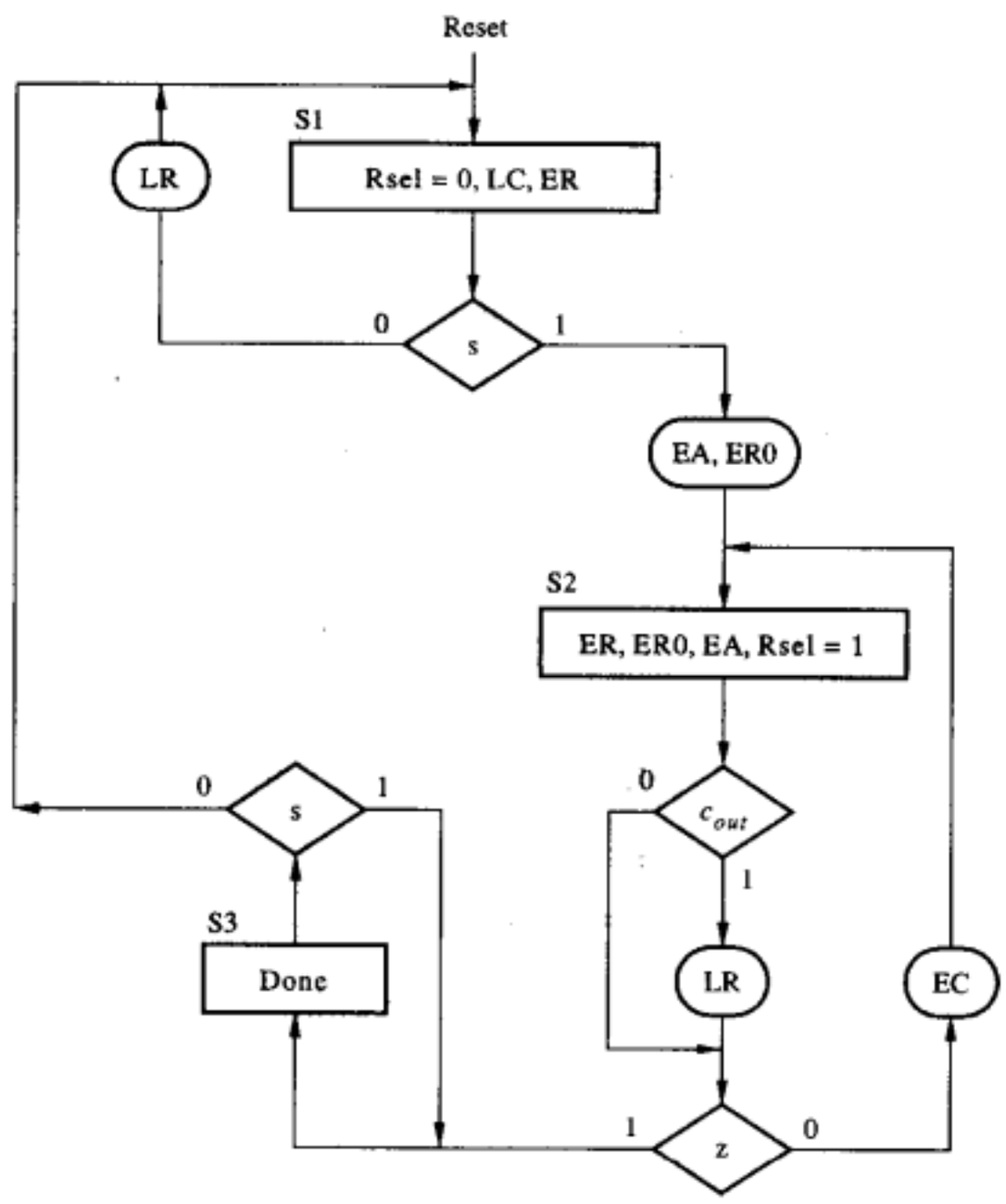


图10-26 改进后的除法器控制电路的算法状态机图

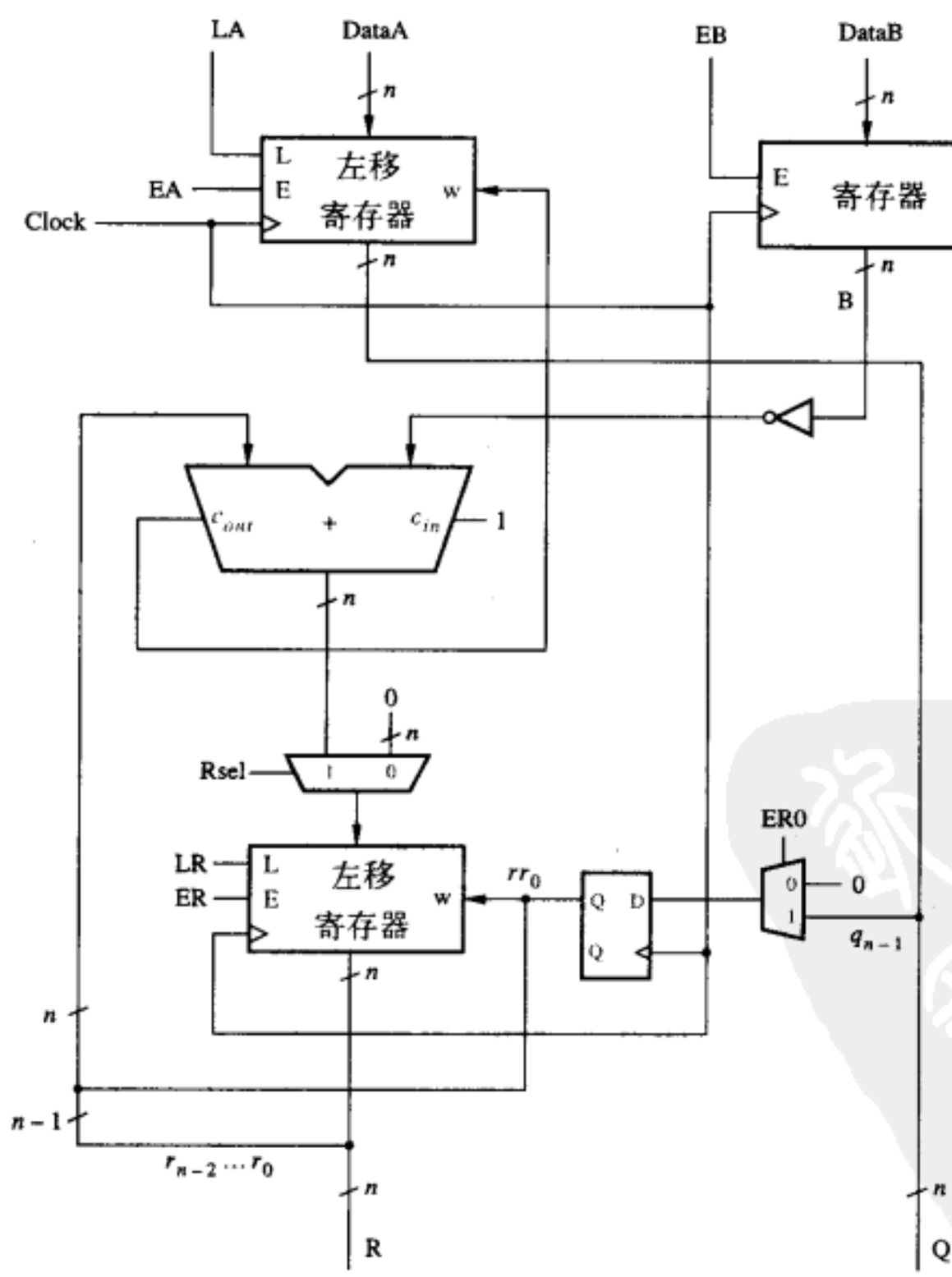


图10-27 改进后的除法器的数据路径电路

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done);
   parameter n = 8, logn = 3;
   input Clock, Resetn, s, LA, EB;
   input [n-1:0] DataA, DataB;
   output [n-1:0] R, Q;
   output reg Done;
   wire Cout, z, R0;
   wire [n-1:0] DataR;
   wire [n:0] Sum;
   reg [1:0] y, Y;
   wire [n-1:0] A, B;
   wire [logn-1:0] Count;
   reg EA, Rsel, LR, ER, ER0, LC, EC;
   integer k;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

   always @(s, y, z)
   begin: State_table
      case (y)
         S1:  if (s == 0) Y = S1;
              else Y = S2;
         S2:  if (z == 0) Y = S2;
              else Y = S3;
         S3:  if (s == 1) Y = S3;
              else Y = S1;
         default: Y = 2'bxx;
      endcase
   end

   always @(posedge Clock, negedge Resetn)
   begin: State_flipflops
      if (Resetn == 0)
         y <= S1;
      else
         y <= Y;
   end
   always @(y, s, Cout, z)
   begin: FSM_outputs
      // defaults
      LR = 0; ER = 0; ER0 = 0; LC = 0; EC = 0; EA = 0;
      Rsel = 0; Done = 0;
      case (y)
         S1:  begin
                 LC = 1; ER = 1;
                 if (s == 0)
                 begin
                    LR = 1; ER0 = 0;
                 end
                 else
                 begin
                    LR = 0; EA = 1; ER0 = 1;
                 end
              end
         S2:  begin
                 Rsel = 1; ER = 1; ER0 = 1; EA = 1;
                 if (Cout) LR = 1;
                 else LR = 0;
                 if (z == 0) EC = 1;
```

图10-28 除法器电路的Verilog代码

```
                else EC = 0;
            end
        S3:  Done = 1;
    endcase
  end

//datapath circuit

  regne  RegB (DataB, Clock, Resetn, EB, B);
    defparam RegB.n = n;
  shiftlne  ShiftR (DataR, LR, ER, R0, Clock, R);
    defparam ShiftR.n = n;
  muxdff  FF_R0 (1'b0, A[n-1], ER0, Clock, R0);
  shiftlne  ShiftA (DataA, LA, EA, Cout, Clock, A);
    defparam  ShiftA.n = n;
  assign  Q = A;
  downcount  Counter (Clock, EC, LC, Count);
    defparam Counter.n = logn;

  assign  z = (Count == 0);
  assign  Sum = {1'b0, R[n-2:0], R0} + {1'b0, ~B} + 1;
  assign  Cout = Sum[n];

  // define the n 2-to-1 multiplexers
  assign  DataR = Rsel ? Sum : 0;

endmodule
```

图10-28 （续）

由代码生成的电路的仿真结果如图10-29所示。数据$A = A6$，$B = 8$被载入，然后s被置1。电路进入状态$S2$，同时将R，$R0$，A左移。在仿真结果中，存储A的移位寄存器被标记为Q，因为当除法操作结束时，该移位寄存器中存有商的值。状态$S2$的前三个有效时钟沿，由$R||R0$表示的数比在B（8）中的数小，因此在每个时钟沿$R||R0||A$左移，并且0被移入Q。在状态$S2$的第四个时钟周期，R的内容为$00000101=(5)_{10}$，$R0$为0，因此$R||R0 = 000001010 = (10)_{10}$。在下一个有效时钟沿，加法器的输出$10-8 = 2$被加载到$R$，1被移入$Q$。在状态$S2$，经过$n$个时钟周期后，电路转到状态$S3$，得到正确结果$Q = 14 = (20)_{10}$和$R = 6$。

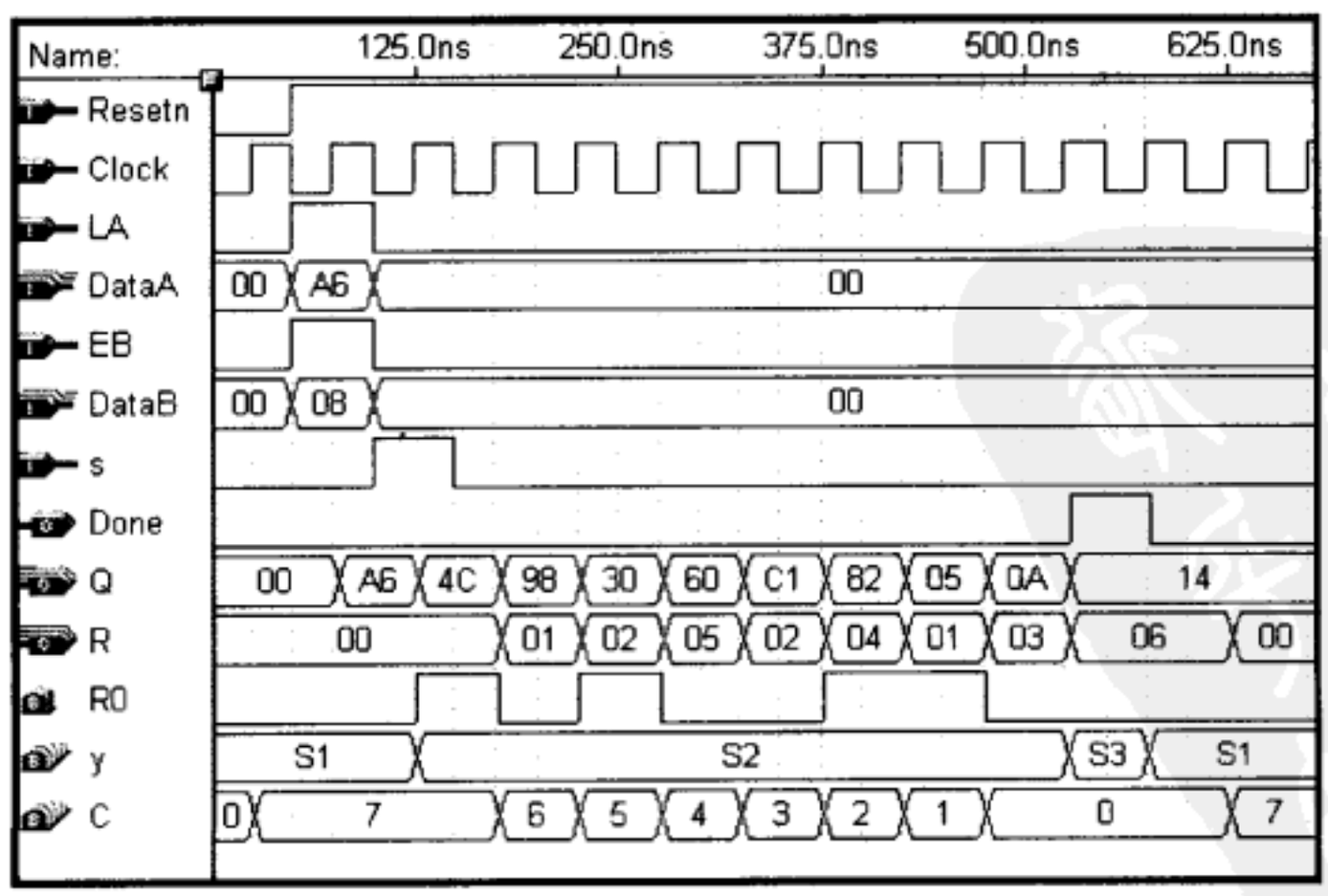


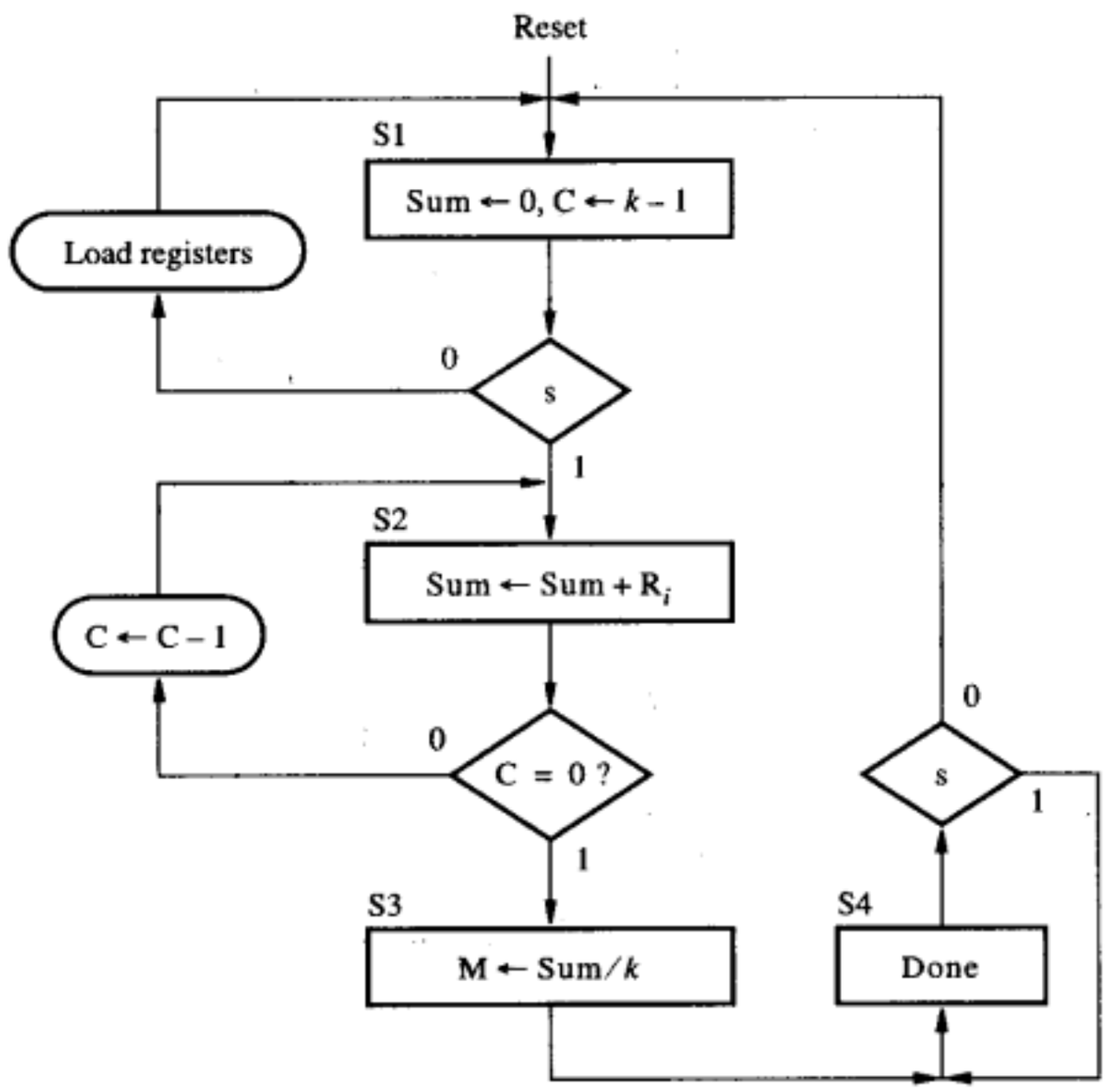
图10-29 除法器电路的仿真结果

10.2.5 算术平均值

假设有k个n位的数存储在一组寄存器R_0、…，R_{k-1}中。我们希望设计一个电路，计算出这些寄存器中各数据的平均值M。图10-30a所示为一种合适算法的伪代码。每进行一次循环都将一个寄存器R_i中的数据加到变量Sum中。在求和计算结束后，求出$M = M/k$便可得到算术平均值。我们假定使用了整数除法器，因此同时会产生一个余数R（尽管在代码中没有显示）。

```
Sum = 0;
for i = k − 1 down to 0 do
    Sum = Sum + Ri
end for;
M = Sum ÷ k;
```

a) 伪代码



b) 算法状态机图

图10-30 求k个数的平均值的算法

图10-30b给出了算法状态机图。当启动输入信号s为0时，寄存器从外部读取数据。当s变为1时，状态机进入状态$S2$，在$C \neq 0$期间，保持状态$S2$不变，并计算总和（C为计数器，代表图10-30a中的i）。当C减到0时，状态机转入状态$S3$并计算$M = Sum/k$。从上一个例子中，我们知道除法运算需要多个时钟周期，但在这个算法状态机图中我们并没有把它表示出来。除法运算结束后，状态机进入状态$S4$，并将$Done$置1。

数据路径电路

该任务中的数据路径电路要比前面例子中的复杂些（如图10-31所示）。我们需要一个带使能输入的寄存器来保存Sum。为了简单起见，我们假设Sum能够用n位数表示，且不会溢出。在Sum寄存器的输入端应该有一个多路器，若处于状态$S1$，则选择0，而若处于状态$S2$，则选择加法器的输出结果。Sum寄存器为加法器提供一个输入数据，而另一个输入数据则必须从k个寄存器中选择某个提供。要从多个寄存器中选取某个提供相应数据，一种方法是将它们连到一个k选1多路器的数据输入端，然后将其输出连接到加法器的输入。多路器的选择信号由计数器C提供。除法运算可以用10.2.4节设计的除法器电路完成。

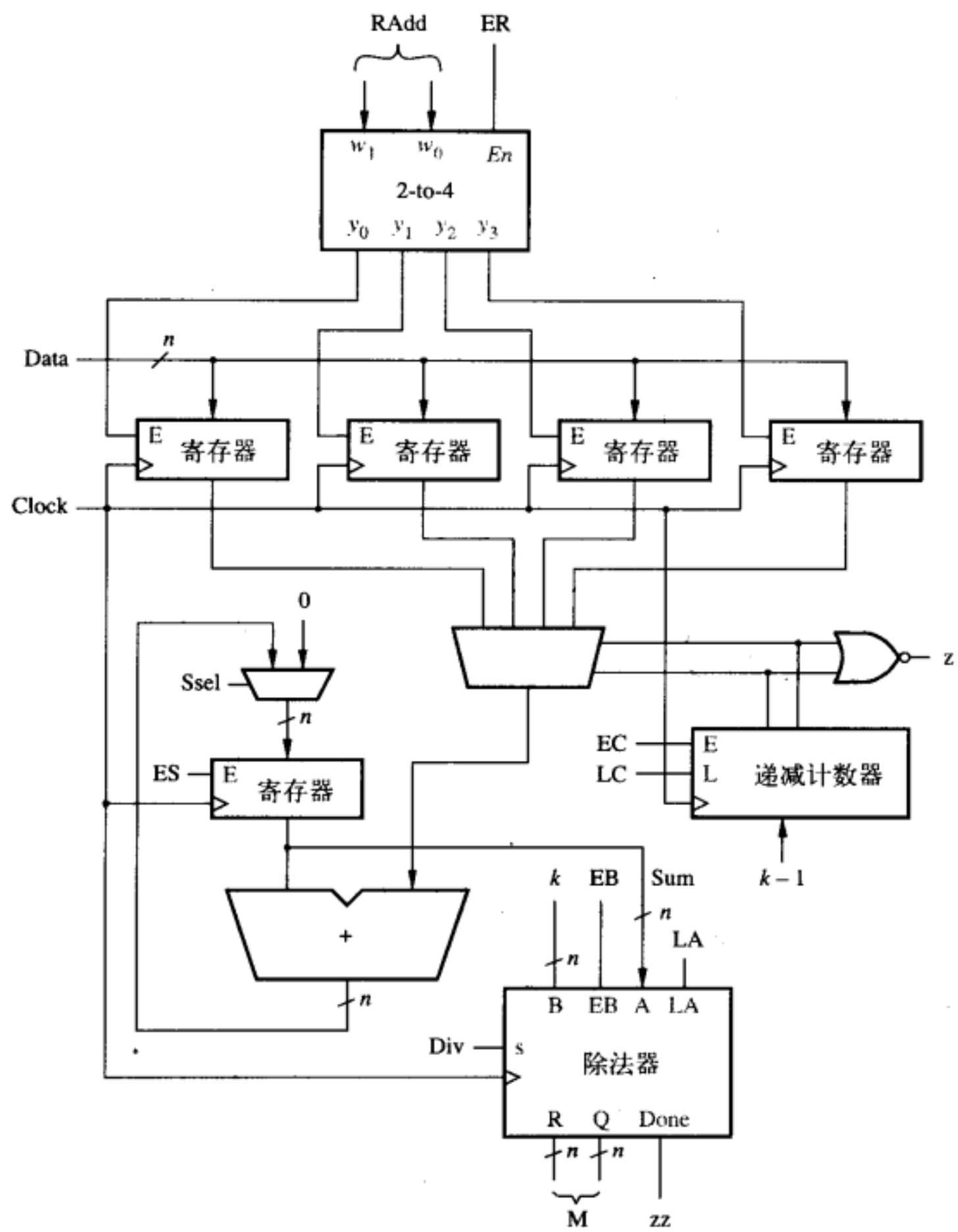


图10-31 求平均值操作的数据路径电路

图10-31所示的电路是基于$k = 4$画出的，对k取更大值的情况，可以采用类似的电路结构。注意图中寄存器R_0到R_3的使能输入端连接到一个2到4译码器的输出，该译码器的输入为2位的*RAdd*，表示寄存器地址（register address），其使能端由信号*ER*驱动。所有寄存器都通过同一条*Data*总线加载数据。由于$k = 4$，我们可以将除法操作简化为将*Sum*向右移两位，这可以由一个移位寄存器在一个周期内完成。若想要设计一个更通用的电路以适应任何k值，则必须使用10.2.4节中设计的除法器电路。

控制电路

图10-32给出了控制如图10-31所示电路的有限状态机的算法状态机图。在状态*S*1，数据可以并行地加载到寄存器$R_0, \cdots, R_{k-1}$中。正如上面讨论过的，只加载数据没有必要施加控制信号，因为寄存器可在输入*ER*和*RAdd*的控制下加载。当$s = 1$时，有限状态机转到状态*S*2，这时*Sum*寄存器的使能端*ES*有效，从而允许计数器*C*开始递减。当*C*减到0时，$z = 1$，状态机进入状态*S*3，此时*LA*和*EB*有效，将*Sum*和k分别加载到除法器电路的*A*、*B*输入端。然后FSM进入状态*S*4，信号*Div*有效，启动除法运算。计算完成后，除法器将*zz*置1，并转到状态*S*5。于是，平均数*M*由除法器的输出端*Q*和*R*送出。在状态*S*5，信号*Div*必须保持有效，以防止除法器重新初始化其

寄存器。应该注意的是：在图10-30b的算法状态机图中，计算$M = Sum/k$只用了一个状态，而在图10-32中则用了$S3$和$S4$两个状态。将$S3$和$S4$合并成一个状态是可行的，这个问题作为作业，让读者自己完成（见练习题10.6）。

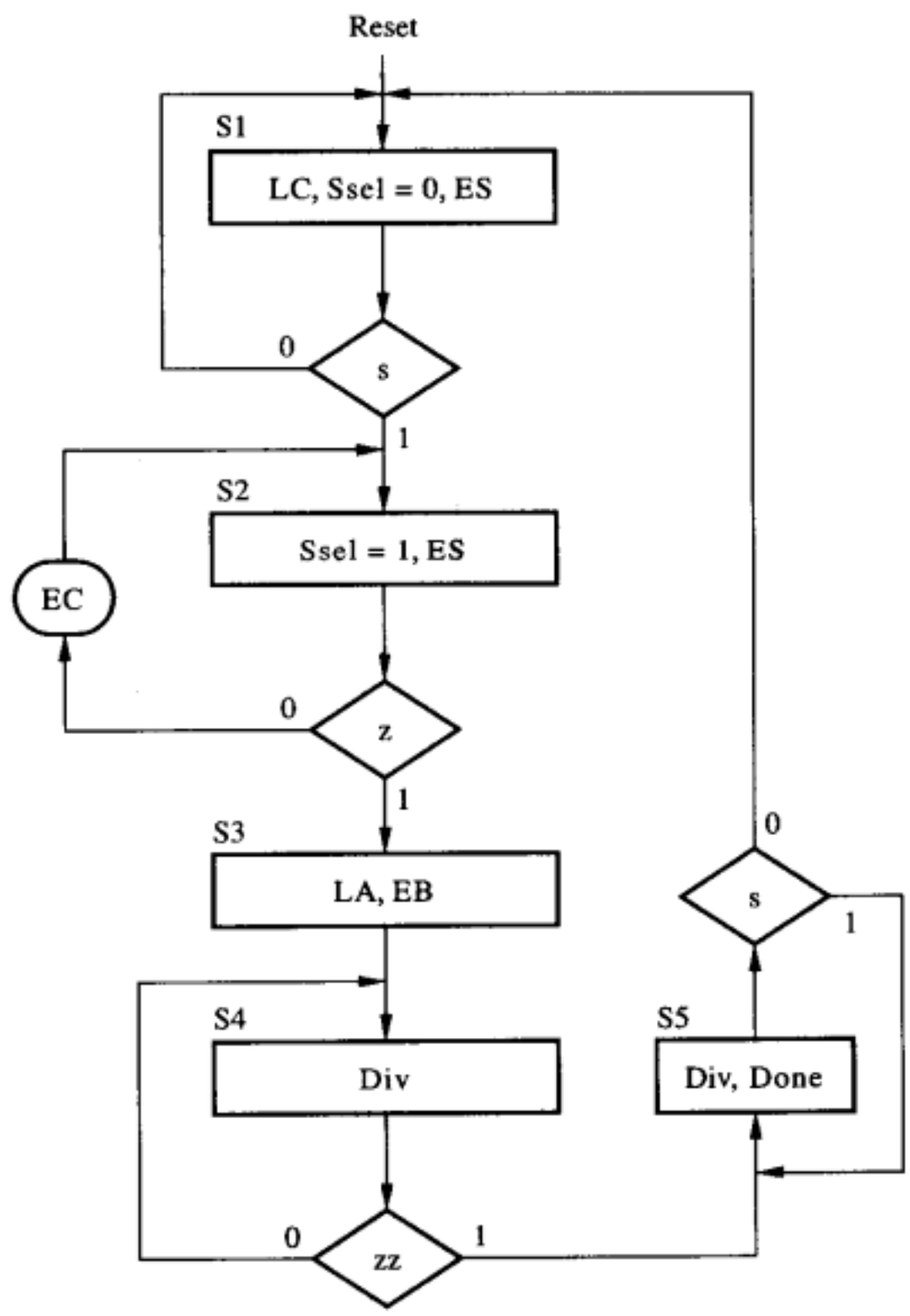


图10-32 求平均值控制电路的算法状态机图

另一种数据路径电路

在图10-31中，寄存器$R_0, \cdots, R_{k-1}$通过多路选择器连接到加法器上。另一种实现连接的方法是在这k个寄存器的每个输出端都加上三态缓冲门，并将所有给定位置上的三态缓冲门连接到加法器的相应输入。递减计数器C的输出连接到2到4译码器，则译码器的输出可用来在适当的时候打开某个寄存器的三态缓冲门（当状态机处在状态$S2$时）。我们将在图10-42中展示用这种方式使用三态缓冲门的例子。

当k的值较大时，最好使用 k 行 n 列的SRAM块，而不要用 k 个寄存器。CAD工具通常提供预先定义的表示SRAM的模块。若所设计的电路准备用全定制芯片实现，则CAD工具必须保证芯片中的SRAM块是符合要求的。有些可编程逻辑器件中包含的SRAM块，其行、列数可以根据需求配置成不同的数。本书配套光盘提供的CAD系统能提供*lpm_ram_dq*模块，该模块属于参数化模块标准库(LPM)中的一个模块。

图10-33给出了求算术平均值电路的原理图（其中参数$k = 16$，$n = 8$）。该原理图是使用与本书配套的CAD工具绘制的。原理图中的四个图形符号表示四个用Verilog描述的子电路（依次命名为*downcnt*，*regne*，*divider*，*meancntl*）模块。除法器（divider）子电路模块的代码在图10-28中给出。*meancntl*子电路表示图10-32中的有限状态机，而该FSM的Verilog代码没有给出。该原理图中还包括一个连接到和（*Sum*）寄存器上的多路器、一个加法器，以及一个检测计数器C是否到达0的或非门。计数器的输出为SRAM块（*MReg*）提供地址输入。

SRAM块总共有16行8列。在图10-31中用一个译码器产生控制信号，将数据加载到k个寄存器中的每一个。要将数据从这些寄存器中读出，还用到了计数器C。为了使图10-33所示的原理图简单易懂，我们用了计数器来读取SRAM块中的数据，但省略了往SRAM块中写数据的问题。修改*meancntl*的代码，让计数器C为SRAM块提供地址以加载初始数据是可行的，但在这里我们不深究这个问题。

为了进行仿真，我们可以利用CAD系统的功能，将原始数据存储在SRAM块中。我们将0存入R_0，1存入R_1……15存入R_{15}。图10-34展示了FPGA芯片中实现电路的时序仿真结果。图中只显示了$C = 5$这点之后的部分仿真的波形。此时，*meancntl*的状态机处于状态$S2$，*Sum*正在累加过程中。当C减至0时，*Sum*计算出正确值，即$1+2+3+\cdots+15=120=(78)_{16}$。状态机进入状态$S3$，停留一个时钟周期，然后进入并保持在状态$S4$，直到除法完成。当FSM进入状态$S5$时，

就得出了正确的计算结果：$Q = 7$，$R = 8$。

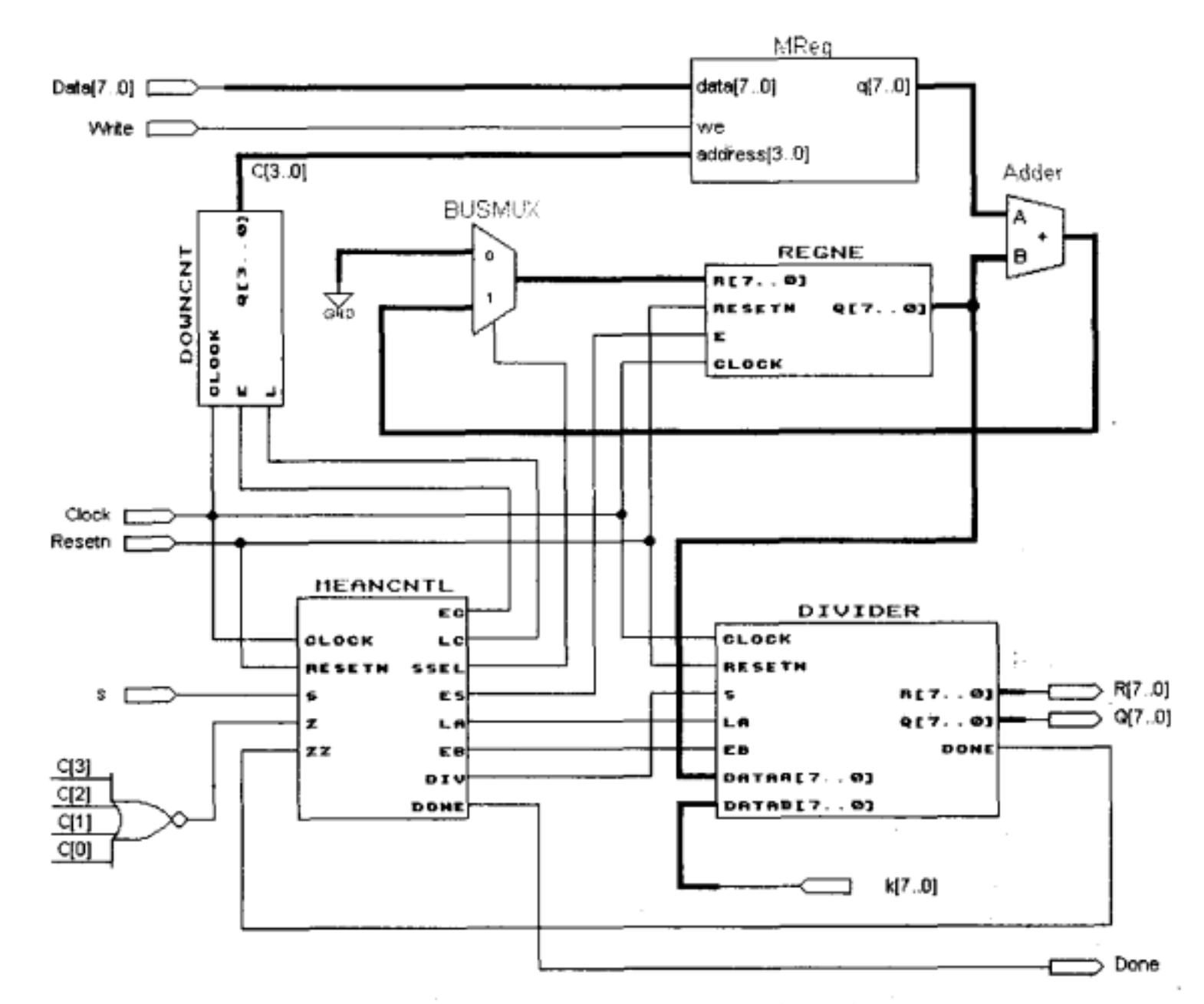


图10-33 包含SRAM块的求平均值电路的原理图

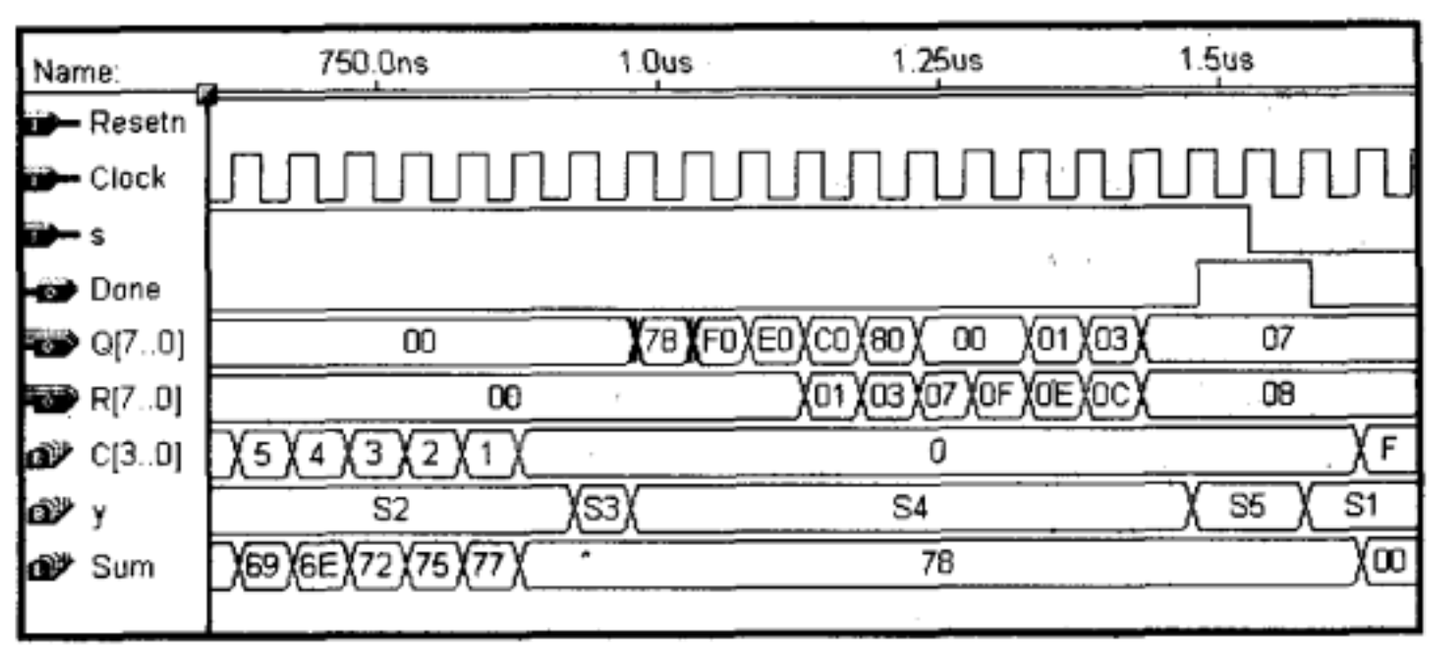


图10-34 使用SRAM的求平均值电路的仿真结果

10.2.6 排序操作

假设一个由k个n位无符号数组成的序列存储在一组寄存器（从R_0到R_{k-1}）中，希望设计一个电路，将该序列（即存储在寄存器组中的数据）按升序重新排列。图10-35给出了一种简单排序算法的伪代码。该算法的基本思想是：在子序列R_i, …, R_{k-1}中找出最小的数并将其存入R_i，其中$i = 0, 1, 2, \cdots, k-2$。每进行一次外层循环都将R_i的值赋给A，而每次内层循环都将A与另一个寄存器R_j进行比较。若R_j中的数比A小，则将R_i与R_j中的数互换，并且A被赋予新的R_i值。

```
for i = 0 to k − 2 do
    A = R_i ;
    for j = i + 1 to k − 1 do
        B = R_j ;
        if B < A then
            R_i = B ;
            R_j = A ;
            A = R_i ;
        end if ;
    end for ;
end for ;
```

图10-35 排序操作的伪代码

图10-36给出了排序算法的算法状态机图。在初始状态$S1$，当

$s = 0$时，寄存器从外部数据输入端加载数据，并且计数器C_i（即代表外层循环变量的i）被清零。当状态机进入状态$S2$，R_i内的数据被加载到A，同样C_j（代表内层循环计数变量j）也被初始化为i。状态$S3$被用来初始化j的值，使其等于$i+1$。状态$S4$将R_j的值加载到B中。然后在状态$S5$，将A和B进行比较，若$B<A$，状态机进入状态$S6$。在状态$S6$和$S7$中，R_i和R_j的值互换。在状态$S8$中，将R_i中的值加载到A。尽管这一步只在$B<A$的情况下才有必要，但无论A与B哪个较大都这样操作，控制流会更简单些。如果C_j不等于$k-1$，状态机从$S8$转到$S4$并继续保持内层循环。如果C_j等于$k-1$且C_i不等于$k-2$，则状态转移到$S2$，并保持外层循环。

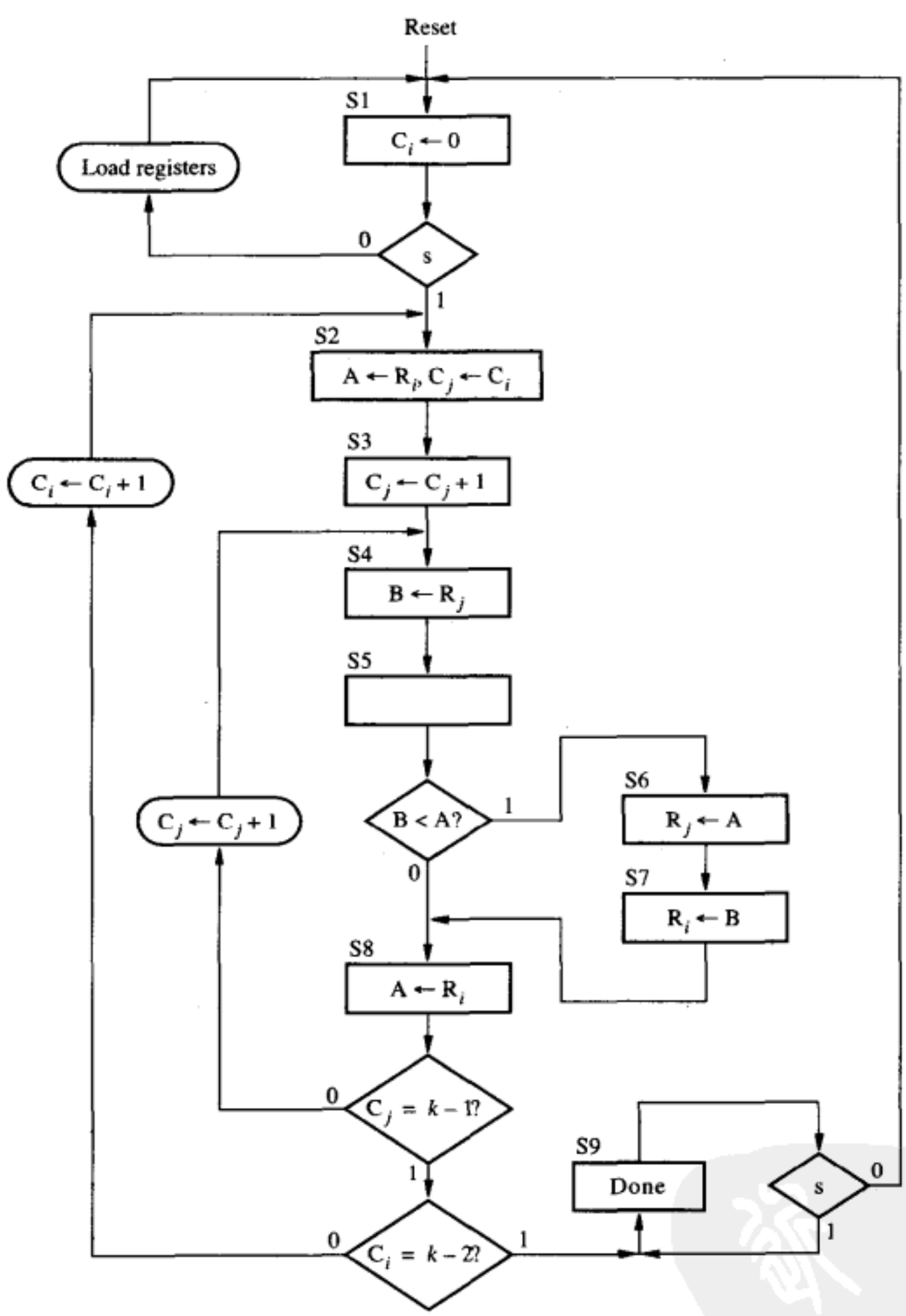


图10-36 排序操作的算法状态机图

数据路径电路

要实现满足图10-36所示的算法状态机图的要求的数据路径电路，有很多种方法。图10-37和图10-38描述了一种可行的方法。图10-37展示了如何用4选1多路器将寄存器$R_0, \cdots, R_{k-1}$与寄存器A、B相连。为了简便，我们假设$k = 4$。寄存器A、B连到一个比较器子电路并通过多路器返回到寄存器$R_0, \cdots, R_{k-1}$的输入端。这些寄存器可以从$DataIn$信号线加载初始数据（未排序数

据)。通过使*WrInit*控制信号有效，并将寄存器地址加到*RAdd*输入端，数据将被加载到各寄存器。由*Rd*控制信号驱动的三态门用来将寄存器的数据输出到*DataOut*输出端。

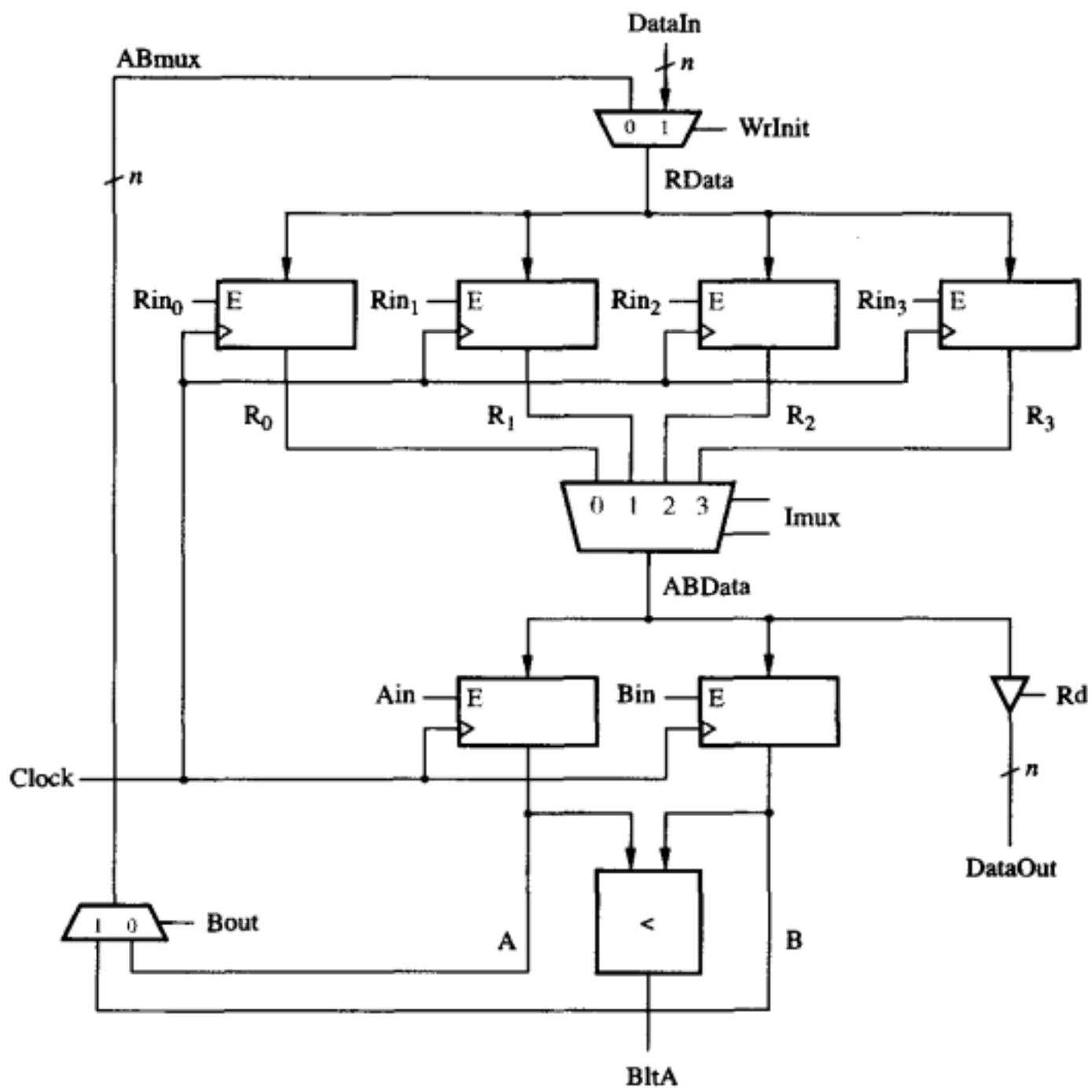


图10-37 排序操作的部分数据路径电路

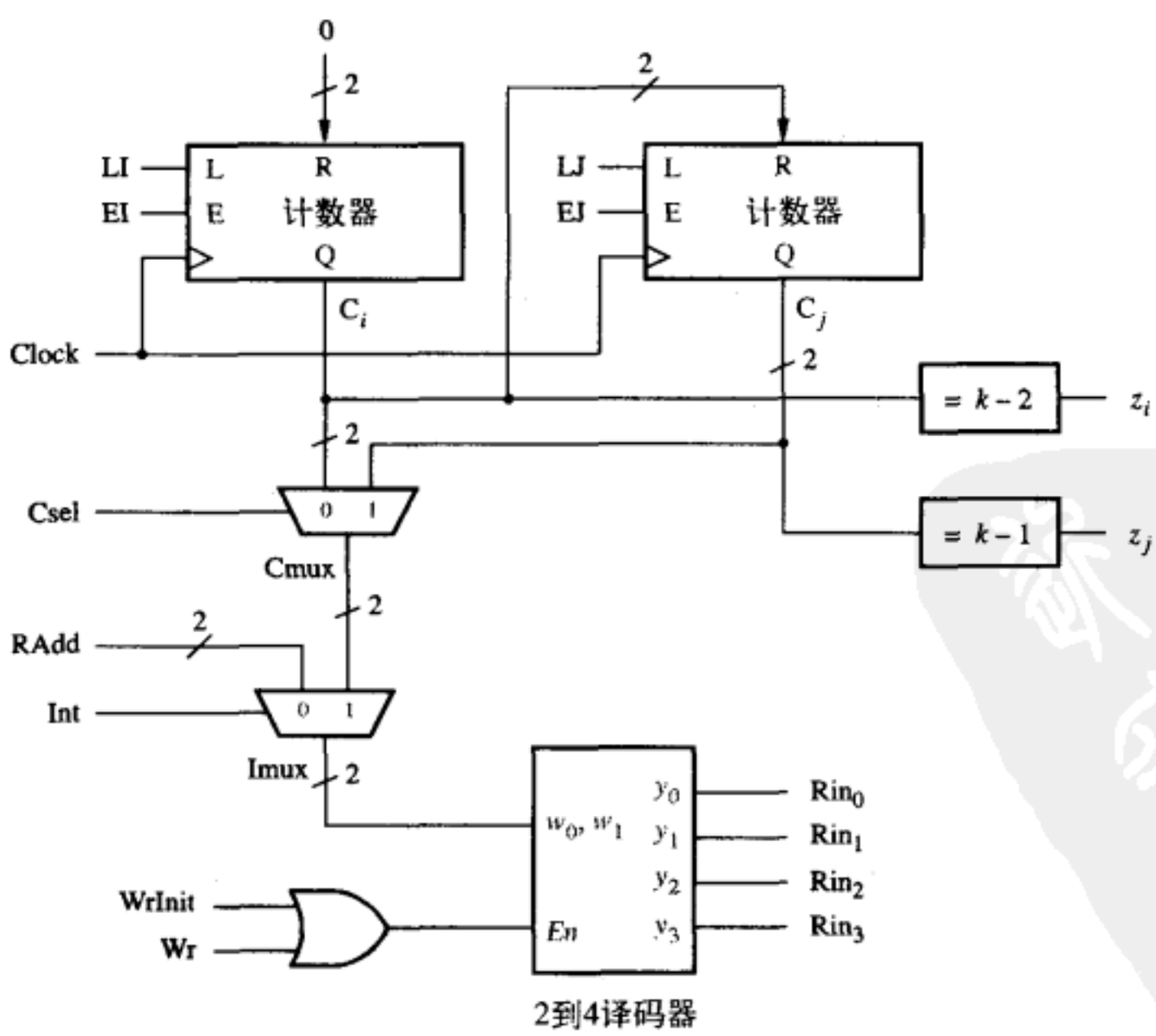


图10-38 排序操作的部分数据路径电路

信号Rin_0，…，Rin_{k-1}由图10-38所示的2到4译码器控制。如果Int =1，则译码器由计数器C_i或C_j之一驱动。如果Int = 0，则译码器由外部输入的$RAdd$驱动。当$C_i = k-2$和$C_j = k-1$时，分别将z_i和z_j置1。数据路径电路中所用的控制信号的算法状态机图见图10-39所示。

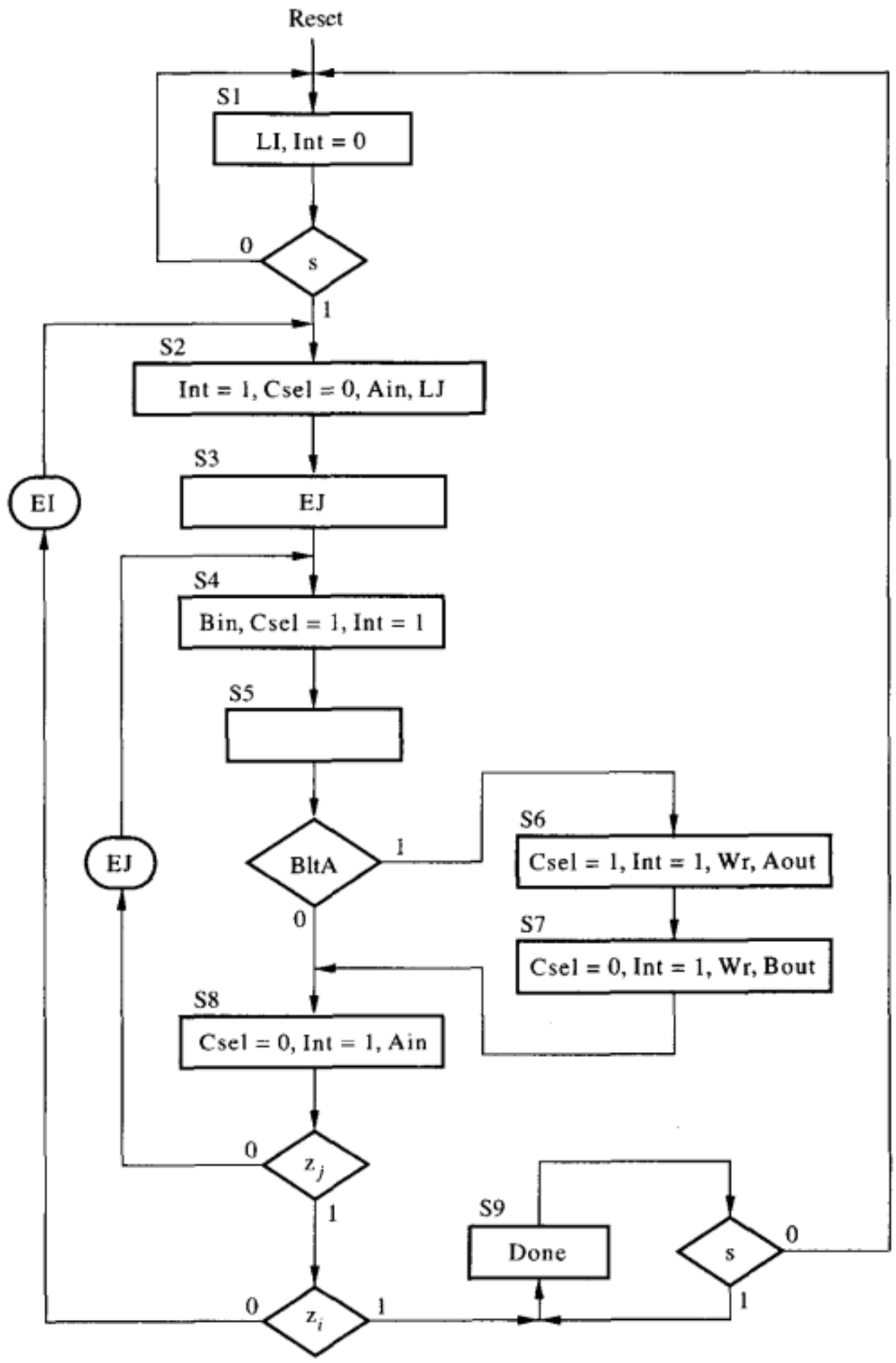


图10-39 控制电路的算法状态机图

Verilog代码

图10-40给出了排序操作的Verilog代码。描述控制排序操作的FSM（有限状态机）所用的方法与前面的例子相同，即用*State_table*（状态表）、*State_flipflops*（状态转移）和*FSM_output*（状态机输出）三个always块进行描述。在always块之后，实例引用了寄存器R_0到R_3，还有A和B。计数器C_i和C_j也分别用实例名*OuterLoop*（外层循环）和*InnerLoop*（内层循环）引用。输出为*CMux*和*IMux*的多路器用条件操作符表达式描述。图10-37中的4选1多路器用case语句定义，为每个不同的*IMux*指定相应的*ABData*信号值。图10-38所示的输出为Rin_0…Rin_3的2到4译码器用case语句实现对拼接信号{Rin_3，Rin_2，Rin_1，Rin_0}的赋值。最后，代码指定z_i和z_j的值，并为输出信号*DataOut*定义了三态门。

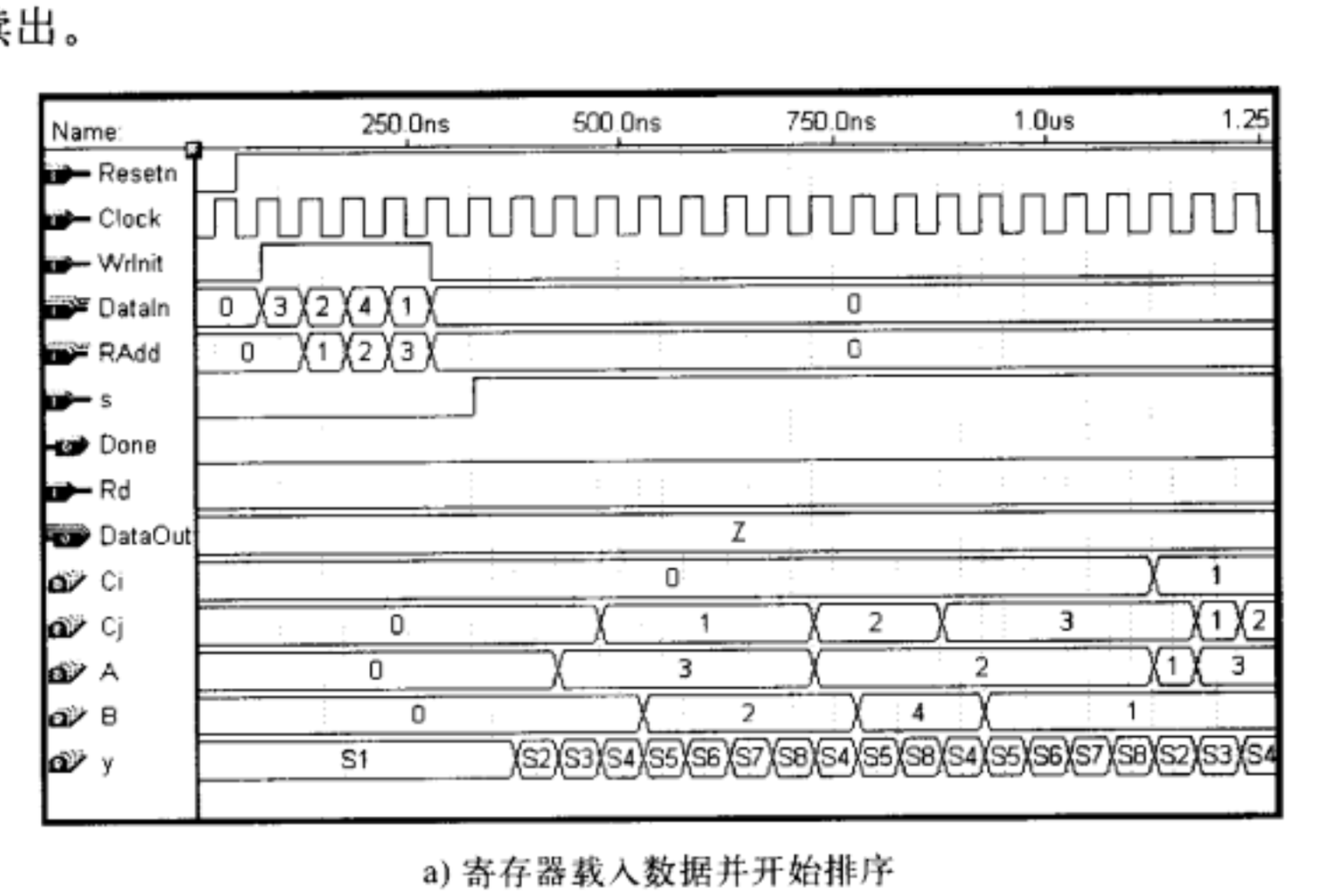
```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
   parameter n = 4;
   input Clock, Resetn, s, WrInit, Rd;
   input [n-1:0] DataIn;
   input [1:0] RAdd;
   output [n-1:0] DataOut;
   output reg Done;
   wire [1:0] Ci, Cj, CMux, IMux;
   wire [n-1:0] R0, R1, R2, R3, A, B, RData, ABMux;
   wire BltA, zi, zj;
   reg Int, Csel, Wr, Ain, Bin, Bout;
   reg LI, LJ, EI, EJ, Rin0, Rin1, Rin2, Rin3;
   reg [3:0] y, Y;
   reg [n-1:0] ABData;

// control circuit
   parameter S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
   parameter S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

   always @(s, BltA, zj, zi, y)
   begin: State_table
      case (y)
         S1:  if (s == 0) Y = S1;
              else Y = S2;
         S2:  Y = S3;
         S3:  Y = S4;
         S4:  Y = S5;
         S5:  if (BltA) Y = S6;
              else Y = S8;
         S6:  Y = S7;
         S7:  Y = S8;
         S8:  if (!zj) Y = S4;
              else if (!zi) Y = S2;
              else Y = S9;
         S9:  if (s) Y = S9;
              else Y = S1;
         default: Y = 4'bx;
      endcase
   end

   always @(posedge Clock, negedge Resetn)
   begin: State_flipflops
      (Resetn == 0)
       y <= S1;
      else
       y <= Y;
   end

   always @(y, zj, zi)
   begin: FSM_outputs
      // defaults
      Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0;
      Wr = 0; Ain = 0; Bin = 0; Bout = 0;
      case (y)
         S1:  begin LI = 1; Int = 0; end
         S2:  begin Ain = 1; LJ = 1; end
         S3:  EJ = 1;
         S4:  begin Bin = 1; Csel = 1; end
         S5:; // no outputs asserted in this state
         S6:  begin Csel = 1; Wr = 1; end
         S7:  begin Wr = 1; Bout = 1; end
         S8:  begin
```

图10-40 排序电路的Verilog代码

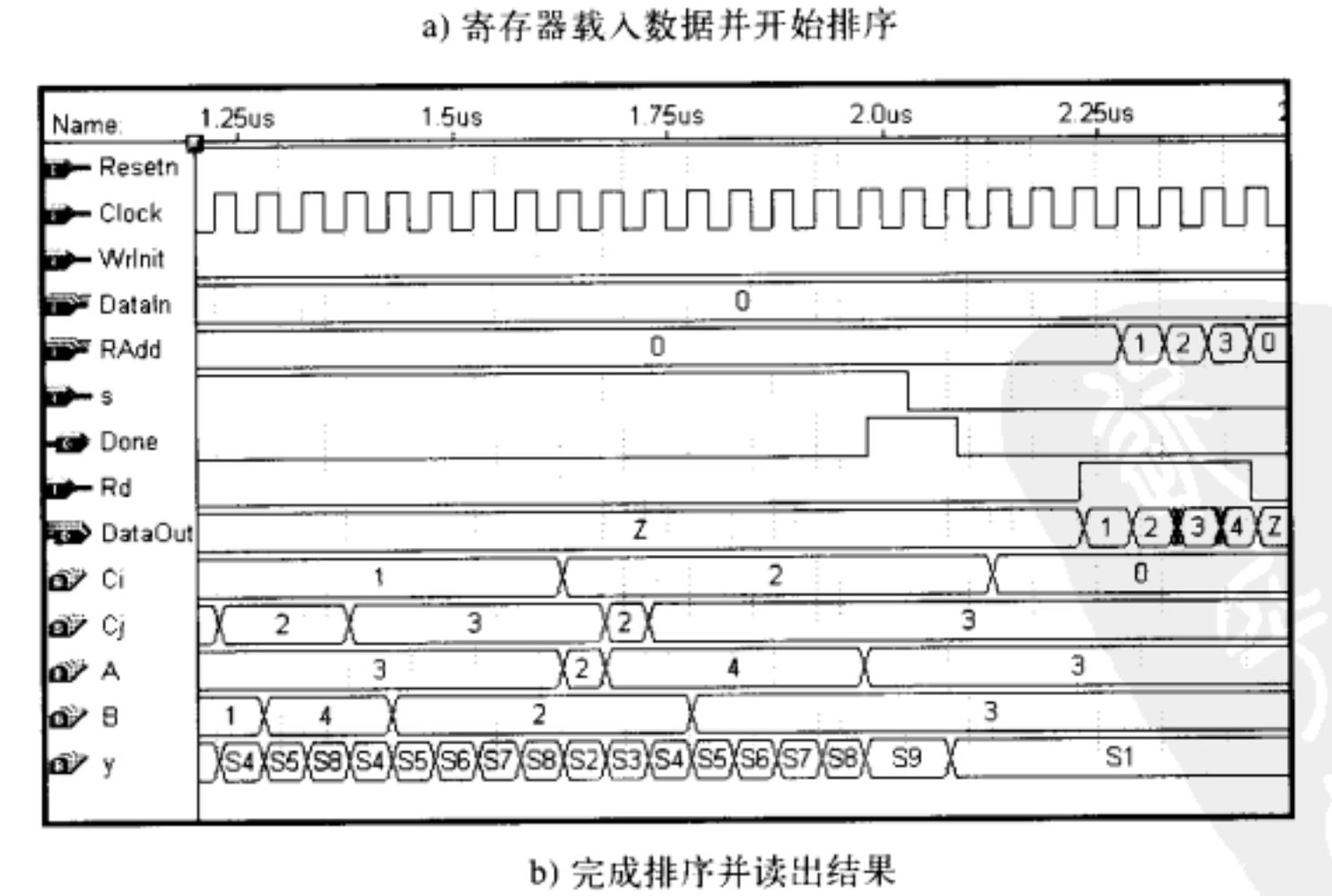
```
                    Ain = 1;
                    if (!zj) EJ = 1;
                    else
                    begin
                        EJ = 0;
                        if (!zi) EI = 1;
                        else EI = 0;
                    end
                end
            S9:  Done = 1;
        endcase
    end

    //datapath circuit

    regne  Reg0 (RData, Clock, Resetn, Rin0, R0);
        defparam Reg0.n = n;
    regne  Reg1 (RData, Clock, Resetn, Rin1, R1);
        defparam Reg1.n = n;
    regne  Reg2 (RData, Clock, Resetn, Rin2, R2);
        defparam Reg2.n = n;
    regne  Reg3 (RData, Clock, Resetn, Rin3, R3);
        defparam Reg3.n = n;

    regne  RegA (ABData, Clock, Resetn, Ain, A);
        defparam RegA.n = n;
    regne  RegB (ABData, Clock, Resetn, Bin, B);
        defparam RegB.n = n;

    assign  BltA = (B < A) ? 1 : 0;
    assign  ABMux = (Bout == 0) ? A : B;
    assign  RData = (WrInit == 0) ? ABMux : DataIn;

    upcount  OuterLoop (2'b00, Resetn, Clock, EI, LI, Ci);
    upcount  InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

    assign  CMux = (Csel == 0) ? Ci : Cj;
    assign  IMux = (Int == 1) ? CMux : RAdd;

    always @(WrInit, Wr, IMux, R0, R1, R2, R3)
    begin
        case (IMux)
            0:  ABData = R0;
            1:  ABData = R1;
            2:  ABData = R2;
            3:  ABData = R3;
        endcase

        if (WrInit || Wr)
            case (IMux)
                0:  {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
                1:  {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
                2:  {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
                3:  {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
            endcase
        else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;
    end

    assign  zi = (Ci == 2);
    assign  zj = (Cj == 3);
    assign  DataOut = (Rd == 0) ? 'bz : ABData;

endmodule
```

图10-40 （续）

我们用FPGA芯片实现了图10-40中的代码。图10-41给出了某次仿真的结果。图10-41a显示了仿真结果的前半部分，从0到1.25μs，图10-41b给出了后半部分，从1.25μs到 2.5μs。电路复位后，*WrInit*被置为1，维持4个时钟周期，未排序的数据利用*DataIn*和*RAdd*两条输入线写入到四个寄存器中。当*s*变为1后，状态机变成状态*S*2。从状态*S*2到*S*4，$R_0(3)$ 的数据被加载到寄存器*A*，$R_1(2)$ 的数据被加载到寄存器*B*。在状态*S*5，比较*A*、*B*的值，因为$B < A$，则状态机用*S*6和*S*7两个状态交换寄存器R_0和R_1内的数据。在状态*S*8，R_0的数据再次加载到*A*中（此时*A*为2）。由于z_j无效，状态机使计数器C_j增加1，并使状态回到*S*4。此时$R_2(4)$的内容被加载到寄存器*B*，状态机进入状态*S*5。由于$B = 4$，$A = 2$，*B*并不小于*A*，状态机直接进入状态*S*8，接着回到状态*S*4。此时$R_3(1)$的内容被加载到*B*，与$A = 2$在状态*S*5进行比较。由于$B < A$，R_0和R_3交换，状态机转移到状态*S*8。此刻，寄存器中的内容为$R_0 = 1$, $R_1 = 3$, $R_2 = 4$, $R_3 = 2$。由于$z_j = 1$而$z_i = 0$，状态机转移到状态*S*2，执行外层循环的下一次迭代。随着仿真时间的继续，在图10-41b中，状态机进入状态*S*8，且$C_i = 2$, $C_j = 3$。然后状态机转到状态*S*9，并将*Done*置为1。将*Rd*置为1，并用*RAdd*选择各寄存器，正确排序后的数据随即从相应的寄存器中读出。



a) 寄存器载入数据并开始排序



b) 完成排序并读出结果

图10-41 排序操作的仿真结果

另一种数据路径电路

在图10-37中，我们用多路器来连接数据路径电路中的各个寄存器。另一种途径是用三态门来连接各个寄存器（如图10-42所示）。我们曾在7.14节中提到过，连接寄存器的一组公用的线（n条）称为总线。图10-42所示的电路有两条总线，一条总线将寄存器R_0, …, R_3的输出连接到寄存器A、B的输入。另一条则将A、B的输出连接回寄存器R_0, …, R_3的输入。当多路器提供寄存器之间的连接（如图10-37所示）时，总线这个术语也可以用来指代寄存器之间的连接。

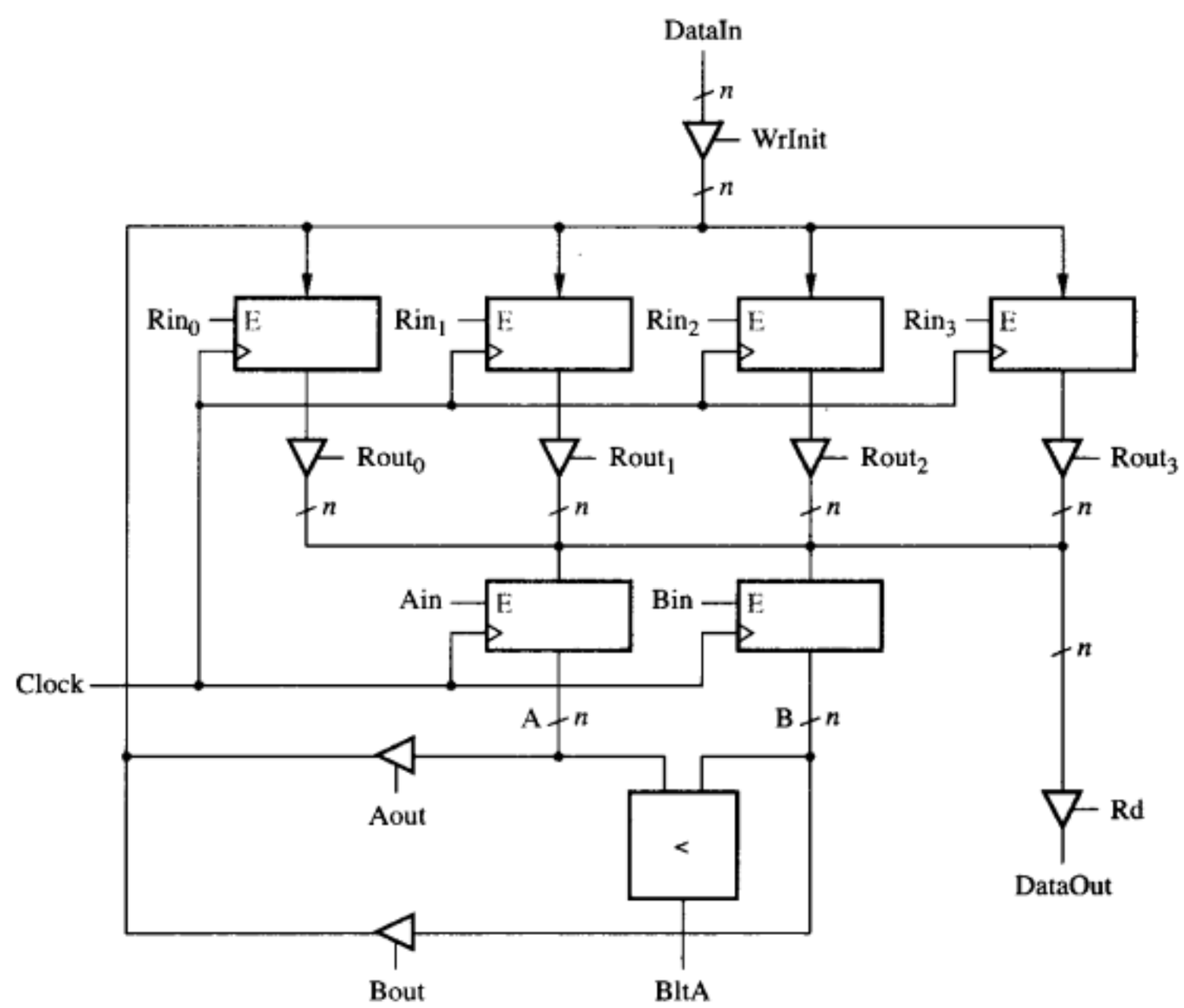


图10-42 使用三态门的数据路径电路

图10-42中的电路只是在图10-38中的电路的基础上做了一点点修改。在10.38中，$IMux$信号被连接到一个产生Rin_0, …, Rin_3的2到4译码器上。若用图10-42中的电路，则还需要一个连接到$IMux$的译码器来产生控制信号$Rout_0$, …, $Rout_3$。图10-39中的算法状态机图描述的控制电路对图10-42所示的数据路径电路也适用。

我们曾在10.2.5节提到，对于大的k值，最好用SRAM块存储数据，而不要用单个的寄存器。排序电路可以做一些修改以适应使用k行n列SRAM块。在这种情况下，数据路径电路类似于图10-37所示的电路，但不需要用4选1多路器，因为SRAM的数据输出可直接连到寄存器A和B。我们仍需要用图10-38中的电路，但是不需要其中的2到4译码器，因为$IMux$信号直接与SRAM的地址输入相连。SRAM的写输入信号$write$由输入为$WrInit$和Wr的或门驱动。可以用Verilog代码描述这个排序电路，其中代表SRAM的器件需要从预定义模块库中实例引用。而图10-40所示的控制电路代码不需要做任何修改（见练习题10.11）。

10.3 时钟同步

在上面几节中，我们举了几个电路中含有许多个触发器的例子。在第9章我们曾提到，为了保证时序电路的正确运转，最根本的是需要细致地考虑与存储元件有关的时序等方面的问题。本节讨论同步时序电路的某些时序方面的问题。

10.3.1 时钟偏差

图10-1展示了如何用使能输入来控制触发器，从而决定是否在有效时钟沿到来时刻改变其存储值。另一种实现时钟使能特性的方法如图10-43所示。电路用一个与门强制使触发器的时钟输入端在$E = 0$时的值为0。这种电路比图10-1中的那个简单，但是在实际应用中可能会出现问题。考虑某个时序电路有许多个触发器，其中一些有使能输入，而另一些没有。如果用图10-43的电路，没有使能输入的触发器将比有使能输入的触发器稍早一点观测到时钟信号的变化。这种时钟信号在不同时刻到达不同触发器的情形就是*时钟偏差*。图10-43只给出了产生时钟偏差的一种可能的来源。类似的问题会出现在芯片中，由于连接到各个触发器的时钟信号线长度略有不同，时钟沿到达各个触发器的时刻就有偏差。

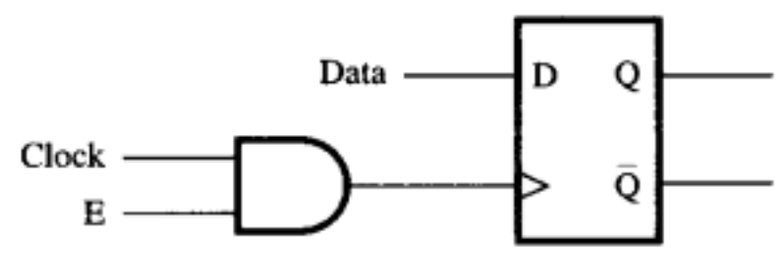


图10-43 时钟使能电路

为了理解时钟偏差可能导致的问题，考虑图10-11中所示的例子：位计数器的数据路径电路。移位寄存器的最低位a_0用于确定位计数器是否加1。假设存在时钟偏差，使得时钟信号到达移位寄存器的触发器比到达位计数器的触发器要稍早些。时钟偏差可能会在 a_0 的值使计数器加1之前，使移位寄存器移位。因此，即使a_0的值为1，图10-11中的信号EB也可能无法引起位计数器在正确的时钟沿的递增。

为了使同步时序电路能正确地运行，根本的办法是尽可能减小时钟偏差。包含很多触发器的芯片，例如可编程逻辑器件（PLD），都使用经仔细设计的时钟分配网络，把时钟信号分配到各个触发器的时钟端口。图10-44所示是时钟分配网络的例子。标着*ff*的每个节点表示各触发器的时钟输入；为清晰起见，触发器没有画出来。图中左边的缓冲器提供时钟信号。时钟信号被用以下方式连接分发到各个触发器：该方式确保各个触发器与时钟源之间的连线长度相同。由于连线部分的外观跟字母H相似，因此时钟分配网络也被称为H树。在PLD中，术语*全局时钟*（global clock）与时钟网络有关。PLD芯片通常会提供一个或者多个与所有触发器相连的全局时钟。当设计可在这样的芯片中实现的电路时，好的设计习惯是在电路中将所有的触发器只与单独一个全局时钟相连。必须避免采用使能信号控制的逻辑门，如图10-43所示，将时钟信号连接到触发器。

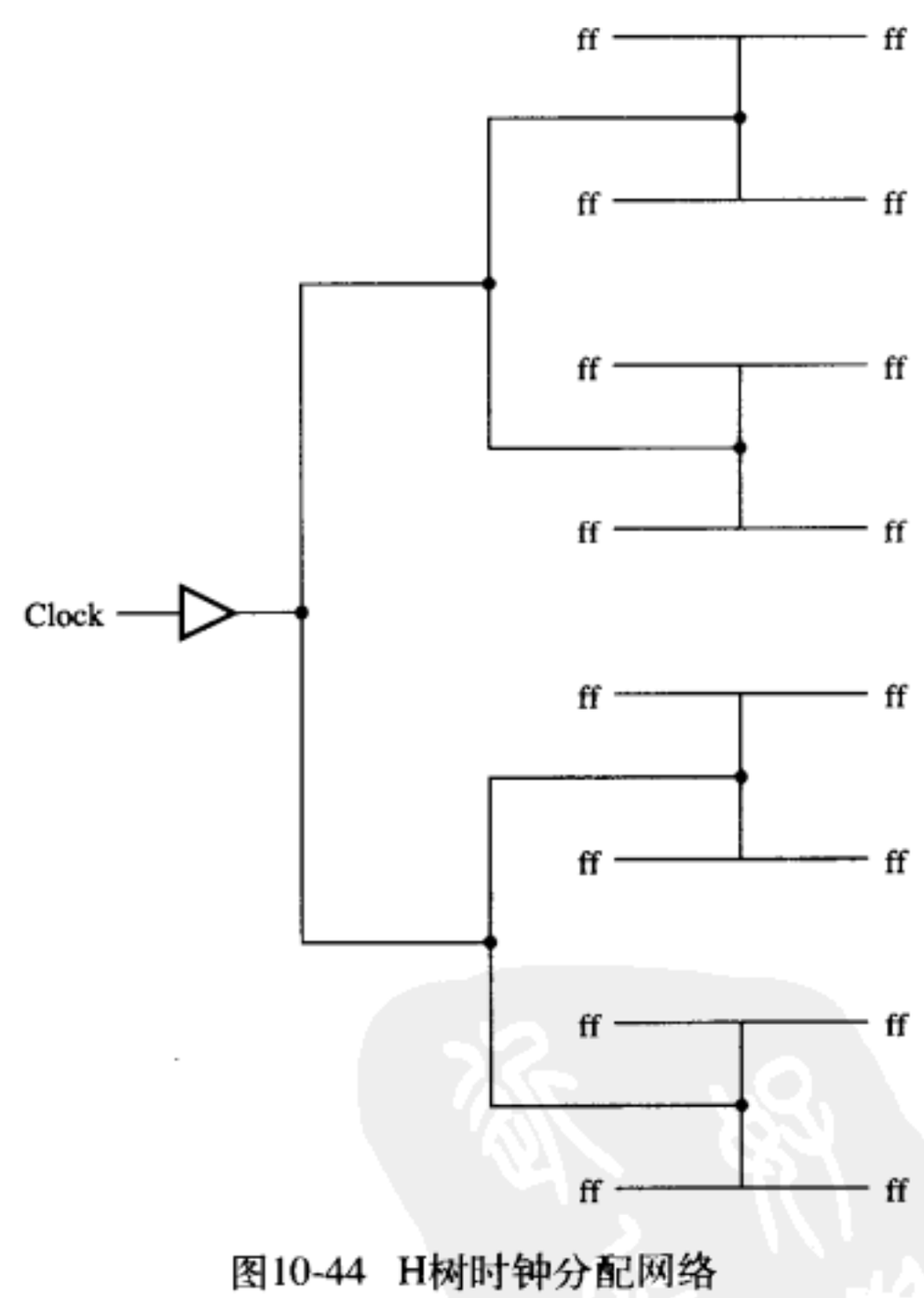


图10-44 H树时钟分配网络

确保时序电路首次接通电源时复位到某个指定状态是很有用的。好的设计习惯是将所有触发器的异步复位（清零）输入端都连接到提供最小时钟偏差的复位信号走线网络。PLD通常为了这个目的提供了全局复位走线网络。

10.3.2 触发器的时序参数

我们曾在7.3.1节中讨论了存储元件的时序参数。由时钟沿能可靠地存入触发器的数据必

须在有效时钟沿到来之前已经稳定了一段时间（t_{su}），同时在时钟沿到来后还必须稳定地保持一段时间（t_h）。输出Q的值在*clock-to-Q*延迟一段时间（t_{cQ}）之后才发生改变。芯片上Q值的变化传递到输出管脚还需要一段时间，即输出延迟时间（t_{od}）。这些时序参数说明了在不考虑触发器在集成电路芯片中与其他电路如何连接的前提下，单独一个触发器的行为。

图10-45中描述了作为集成电路一部分的触发器，展示触发器的时钟端B、数据端D和输出端Q与芯片管脚之间的连接关系。芯片上每个管脚都有一个输入缓冲器。其他电路也可以连接到触发器上；阴影方框代表一个连接到触发器数据端D的组合电路。芯片管脚和触发器之间的传播延迟在图中分别标记为t_{Data}、t_{Clock}和t_{od}。

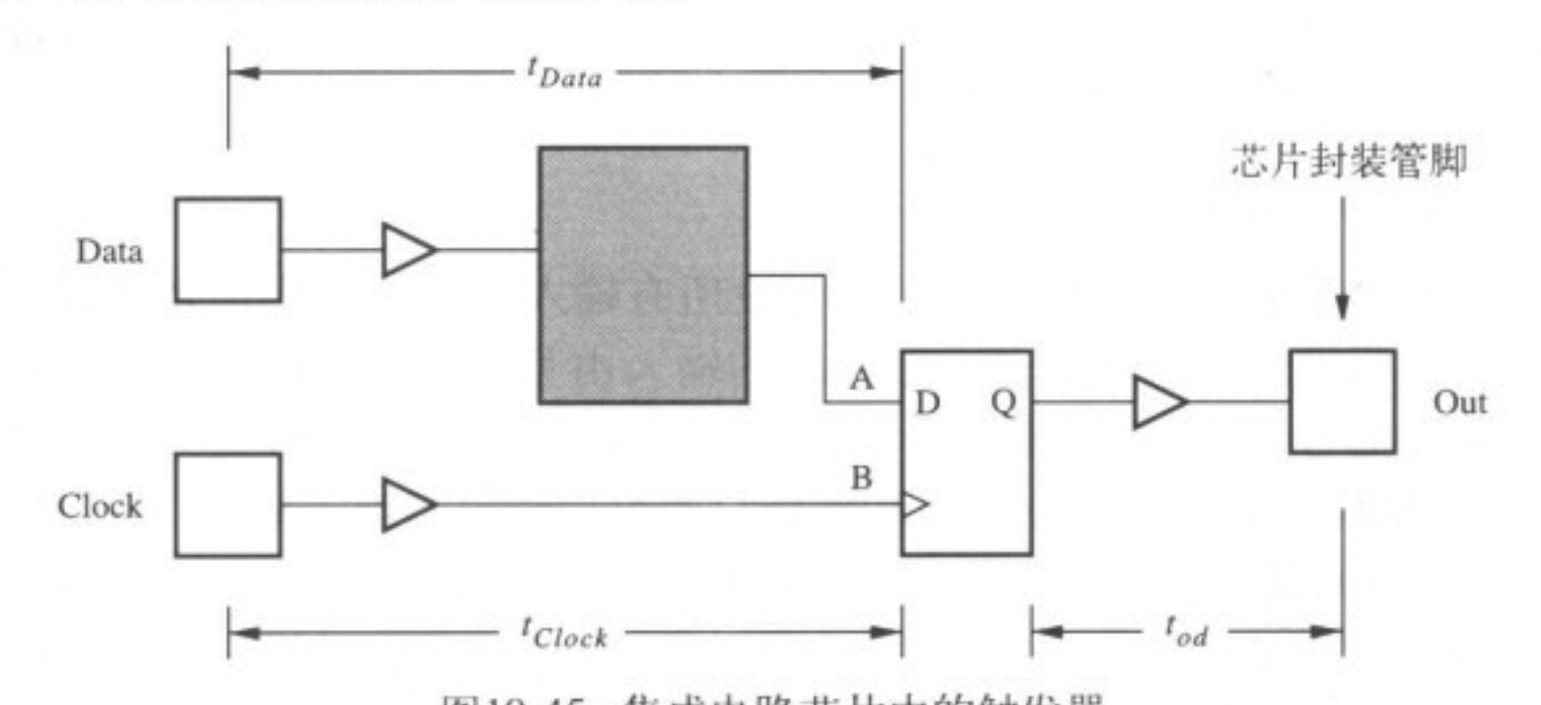


图10-45 集成电路芯片中的触发器

在数字系统中，从一个芯片输出的信号通常用作另一个芯片的输入信号。通常，所有芯片中的触发器都由偏差很小的同一时钟来驱动。信号必须从一个芯片的触发器的Q输出端向另一芯片的触发器的数据输入端传递。为确保符合所有的时序要求，必须考虑本芯片的输出延迟和另一芯片的输入延迟。

延迟时间t_{co}定义为：从有效时钟沿到达芯片封装的时钟管脚算起，直到触发器的输出在该芯片的某个输出管脚发生变化为止所需要的时间。该延迟由三个主要部分组成：t_{Clock}，t_{rd}和t_{od}。时钟信号必须首先从芯片的输入管脚向触发器的时钟输入端传递，这个延迟在图10-45中以t_{Clock}标记；在*clock-to-Q*延迟t_{cQ}之后；触发器产生的新输出传向芯片输出管脚所需的时间为t_{od}。例如，某市售CPLD芯片中的时间参数为t_{Clock} =1.5ns，t_{cQ} = 1ns，t_{od} = 2ns。这些参数给出了从有效时钟沿到芯片输出管脚电平发生变化之间的延迟时间t_{co} = 4.5ns。

若芯片之间相隔的距离很大，则必须考虑它们之间的传播延迟。但就大多数情况而言，芯片之间的距离较小，因此芯片之间的信号传递时间是可以忽略的。一旦信号到达芯片的输入管脚，两个有关的值t_{Data}和t_{Clock}（见图10-45）都必须考虑。例如，在图10-46中，我们假设t_{Data} = 4.5ns，t_{Clock} =1.5ns。芯片中的触发器的建立时间指定为t_{su} = 3ns。图中*Data*信号在时钟上升沿到来之前3ns稳定（译者注：原文为从低到高变化）应该可以满足建立时间的需要。数据信号需4.5ns才能到达触发器，而时钟信号只需1.5ns。标记为A的信号和标记为B的时钟信号同时到达触发器，所以建立时间的要求得不到满足，触发器就不可能有稳定的输出。为避免这种情况，必须增加从芯片外面看到的建立时间。（译者注：即延迟*Data*信号的到达。）

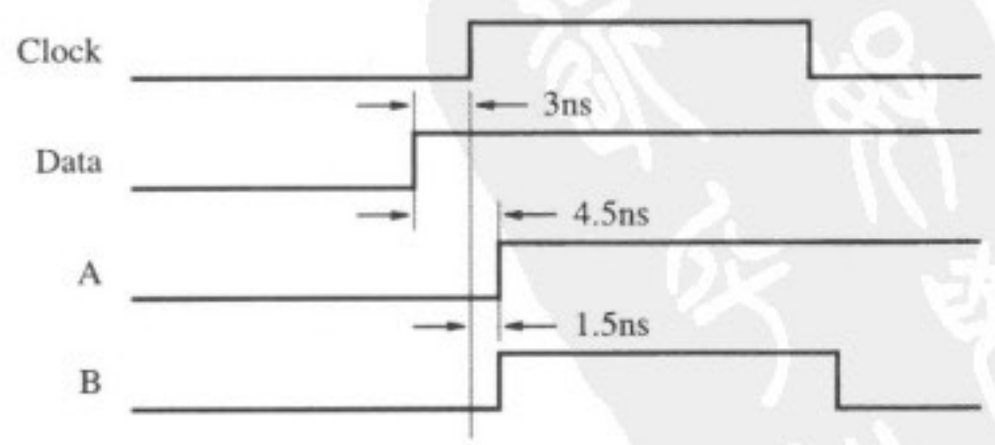


图10-46 芯片中触发器的时序

触发器的保持时间也会受芯片级延迟的影响。通常会引起保持时间的减少而不是增加。例如，用图10-46中的时间参数，假设保持时间t_h = 2ns。假设芯片上*Data*管脚信号恰好与*Clock*管脚的有效时钟沿同时到达，则*Clock*时钟信号的改变到达节点B将比*Data*信号到达节点A提早4.5−1.5=3ns。因此，尽管*Data*数据的变化在管脚处同*Clock*时钟沿一致，而在触发器处所需要的2ns的保持时间还能得到满足。（译者注：本节的讨论是在*Data*信号的变化频率和时钟频率相同的前提下。）

在大规模电路中，确保触发器的时序参数总能满足要求是一个挑战。无论触发器本身的时序参数以及由时钟信号和数据信号引发的相对延迟，都必须考虑到。CAD系统提供可以自动检查所有触发器建立时间和保持时间的工具。这项任务可以由时序仿真器完成，也可以用专门的时序分析工具完成。

10.3.3 触发器的异步输入

在我们所举的同步时序电路的例子里，假设所有输入信号的变化都紧跟在一个有效时钟沿之后发生。这个假设的基本原理是一个电路的输入由另一电路的输出产生，并且这两个电路采用同一个时钟信号。在实际电路中，某些输入可能来自于与时钟无关的信号。若这些信号连接到触发器的*D*输入端，则建立时间和保持时间很可能得不到满足。

若触发器的建立时间或者保持时间不能得到满足，则触发器的输出可能出现与逻辑0 和逻辑1都不对应的电平。这时我们说触发器处于*亚稳态*。触发器最终将稳定在两个稳定状态中的一个，或者1或者0，但是从亚稳态到稳定状态所需要的时间是无法预知的。常用的处理非同步输入的方法的例子，见图10-47所示。异步输入数据连接到两位移位寄存器的输入。第一个触发器的输出（在图中标记为*A*）有时可能会变成亚稳态，但是如果时钟周期足够长，这样*A*就能在下一个时钟脉冲到来之前恢复到稳定的逻辑值。因此第二个触发器的输出不会变成亚稳态，可以安全地连向电路的其他部分。同步电路在信号可以被电路的其余部分使用前引入了一个时钟周期的延迟。

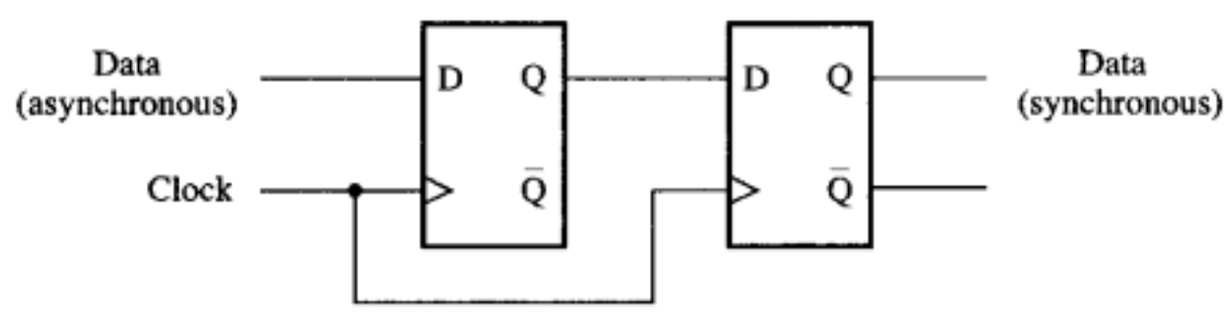


图10-47 异步输入

市售商用芯片，例如可编程逻辑器件（PLD），详细标明了为解决亚稳态问题，采用图10-47所示电路时所允许的最小时钟周期。在实际电路中，不可能保证管脚*A*在时钟沿到来时总是稳定的。器件技术说明书详细地给出节点*A*稳定的概率，该概率用时钟周期的函数描述。我们不再深入探讨这个问题，有兴趣的读者可以查阅参考文献[10，11]中更详细的讨论。

10.3.4 开关的抖动

逻辑电路的输入有时由机械开关产生。机械开关存在的问题是当开关切换时会因跳离接触点而出现抖动。图10-48a展示了一个向逻辑电路提供逻辑输入的单刀单掷开关。如果开关打开，数据信号就是1。如果开关闭合，数据就变成了0，但开关在切换时常会抖动一小段时间，从而引起数据在1和0之间振荡。抖动通常持续10ms左右。

解决使用单刀单掷开关时的抖动问题没有简单的途径。若必须使用这种开关，可行的解决方法就是用电路来解决，如用一个计数器，来测定一个适当的长的延迟，等待抖动的停止（见练习题10.23）。

处理开关抖动问题的更好的途径见图10-48b所示。它用一个单刀双掷开关和一个基础SR锁存器来产生逻辑电路的输入。当开关掷向下端时，锁存器的R输入端是0，$Data$端是0。当开关掷向上端时，锁存器的S输入端变成0，而将Data端置为1。如果开关离开上端时抖动，锁存器的输入变成了$R = S = 1$，值$Data = 1$就被锁存。当开关掷向下端时，$Data$变成0，并且如果开关抖动的话，这个值也会被锁存。注意，当开关抖动时，它会完全在S和R端之间抖动；它仅仅是轻微地抖动，稍微偏离触点，再回到该触点。

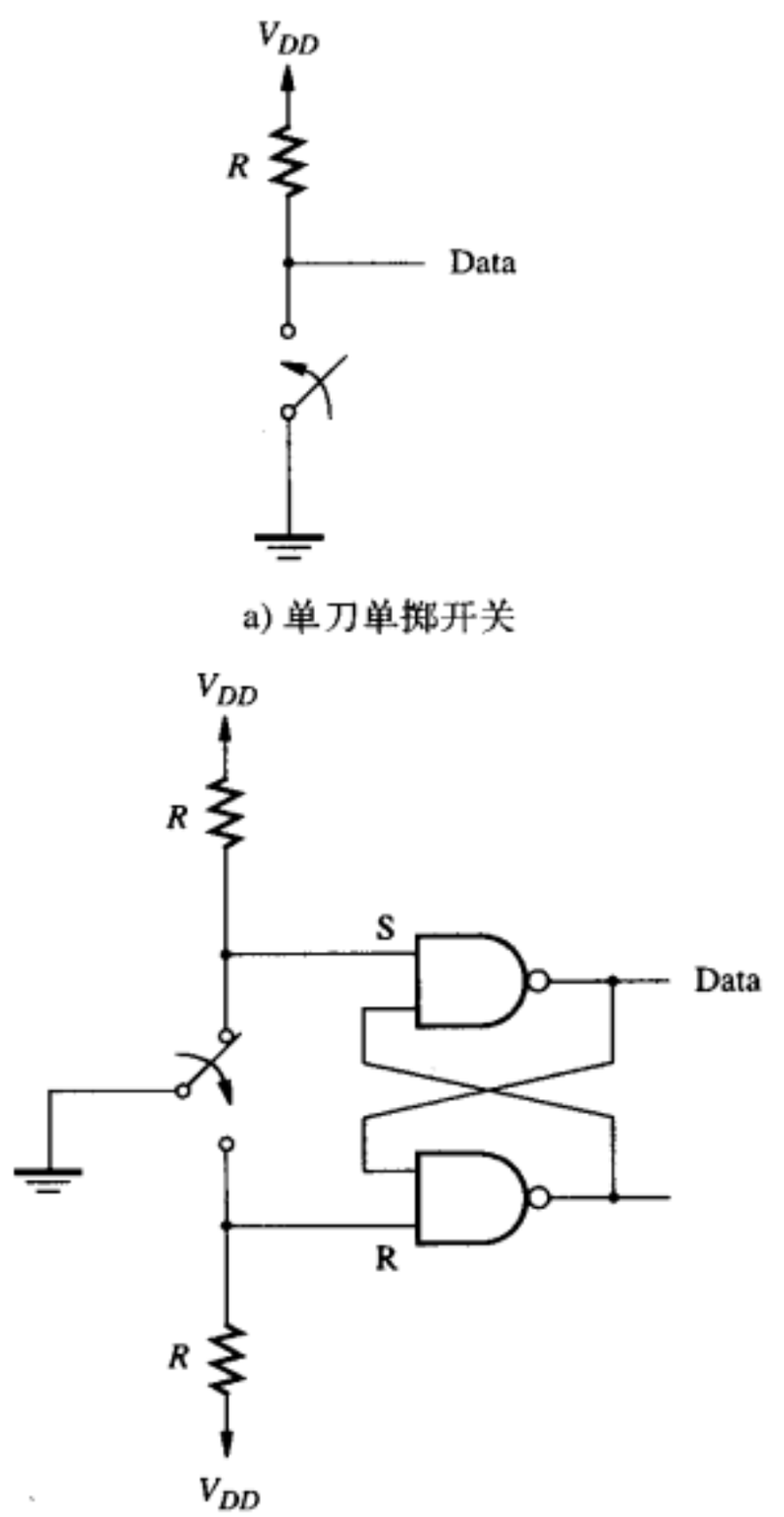


图10-48 开关防抖电路

10.4 小结

本章举了几个数字系统的例子，这些数字系统包括了一个或者几个有限状态机，还有一些电路构件，诸如加法器、寄存器、移位寄存器和计数器。我们展示了如何用算法状态机图来帮助设计数字系统，并且介绍了如何用Verilog代码来描述电路。我们还讨论了一些实际问题，例如时钟偏差、异步输入的同步化以及开关的抖动。有一些不错的书也全面地讲解了本章所介绍的内容，见参考文献 [3-10]。

练习题

10.1 图10-4所示的电路表示的是一个并行加载控制输入和使能输入互相独立的移位寄存器。请画出一个不同的移位寄存器电路，要求该移位寄存器只在使能输入有效时才能并行加载数据。

10.2 图10-10所示的算法状态机图描述了一个二进制位计数器，包含状态$S1$，$S2$和$S3$的摩尔型输出，并包含状态$S2$的一个米利型输出。

(a) 试说明如何修改算法状态机图，使状态S2只包含摩尔型输出。

(b) 给出与(a)中的控制电路对应的算法状态机图。

(c) 编写修改后的控制电路的Verilog代码。

10.3 图10-17所示是移位-加实现的乘法器的数据路径电路。该电路用了一个移位寄存器存放操作数B，以便b_0可以用来决定是否应该将A加到P。另一种方法是用一个普通的寄存器存放操作数B，用一个计数器和一个多路器在乘法运算的每一步选择位b_i。

(a) 画出不使用移位寄存器而使用普通寄存器存放操作数B的算法状态机图。

(b) 画出对应(a)的数据路径电路。

(c) 画出对应(b)的控制电路的算法状态机图。

(d) 编写相应乘法器电路的Verilog代码。

10.4 根据图10-23所示的数据路径和图10-24所示的算法状态机图表示的控制电路，编写除法器电路的Verilog代码。

10.5 10.2.4节展示了如何实现传统的用手工完成的长除法。另一种实现整数除法的方法是进行重复的减法操作，其伪代码如图P10-1所示。

```
Q = 0;
R = A;
while ((R−B) > 0) do
    R = R−B;
    Q = Q + 1;
end whie;
```

图P10-1 整数除法的伪代码

(a) 给出表示图P10-1中伪代码的算法状态机图。

(b) 画出与(a)的算法状态机图对应的数据路径电路。

(c) 画出对应(b)的控制电路的算法状态机图。

(d) 编写该除法器电路的Verilog代码。

(e) 讨论该除法电路与10.2.4节中的除法电路的优缺点。

10.6 在图10-32所示的算法状态机图里，状态$S3$和$S4$用来计算平均值$M = Sum/k$。画出将$S3$和$S4$两个状态合并为一个状态$S3$后的算法状态机图。

10.7 写出由练习题10.6的算法状态机图表示的有限状态机的Verilog代码。

10.8 在图10-36所示的算法状态机图中，我们在状态$S2$进行赋值：$C_j \leftarrow C_i$，在状态$S3$递增$C_j = C_j + 1$。如果在状态$S2$进行$C_j = C_i + 1$的赋值，有没有可能将状态$S3$去掉？说明这种变化可能对数据路径电路与控制电路产生哪些影响。

10.9 图10-35所示是排序操作的伪代码，在这段代码中，被排序的寄存器用变量i和j做索引。在图10-36所示的算法状态机图中，变量i，j分别由计数器C_i和C_j实现。另一种方法是用两个移位寄存器来实现。

(a) 重新设计排序操作电路，用移位寄存器实现，而不用计数器索引寄存器$R_0, \cdots, R_3$。

(b) 编写(a)部分所设计电路的Verilog代码。

(c) 讨论该电路与用计数器的电路的优缺点。

10.10 图10-42所示的排序操作的数据路径电路用三态缓冲器进行寄存器数据的存取。用原理图捕获工具画出如图10-42所示的电路原理图。三态缓冲器实例引用原语（基本元件）库中的*tri*模块。实例引用*lpm_compare*模块作为比较器子电路。用Verilog代码创建其他必要的子电路，创建相应的图形符号来表示它们，并在原理图中将这些图形符号与数据路径模块连接起来。选择某个芯片实现该电路，并给出仿真结果。参看附录B，C，D中如何使用CAD工具的指令。

10.11 图10-40给出了排序电路的Verilog代码。请说明：若要使用$k \times n$ 的SRAM块，应该如何修改代码。用*lpm_ram_dq*模块作为SRAM块。选择同步SRAM选项，以使SRAM内容的所有变化都与时钟同步。（提示：用时钟求反信号对SRAM的操作进行同步，因为这种方法允许直接使用如图10-40所示有限状态机的Verilog代码，不必进行修改。）

10.12 设计一个电路，该电路能计算操作数N（存于n位寄存器中）以2为底的对数（$\log_2 N$）。给出设计过程的每个步骤和所作的假设。编写描述该电路的Verilog代码。

10.13 图10-33所示为求平均数电路的原理图。编写代表该电路的Verilog代码，使用寄存器数组，不用SRAM块。对于除法器子电路，用移位操作实现被4除，而不用10.2.4节所设计的除法器电路。

10.14 10.2.5节中设计的电路用了一个加法器完成寄存器中所存数的求和计算。用来计算$M = Sum/k$的除法器子电路中也包含一个加法器。重新设计该电路，只使用一个加法器完成求和运算与除法运算。只画出连到加法器所需的附加电路，并解释其工作原理。

10.15 编写练习题10.14所设计电路的Verilog代码，包含数据路径电路和控制电路。

10.16 图10-35所示的排序操作的伪代码中，用到寄存器A和B来保存寄存器中需要排序的数据。若只用寄存器A保存排序过程中的临时数据，试给出伪代码。画出表示所需数据路径电路的相应算法状态机图和必须的控制电路。按图10-37的风格，用多路器连接各寄存器。单独画出控制电路的算法状态机图。

10.17 编写练习题10.16所设计的排序电路的Verilog代码。

10.18 在7.14.1节中，我们展示了一个包含三个寄存器（从$R1$到$R3$）的数字系统，还设计了一个控制电路用来交换寄存器$R1$和$R2$中的内容。画出表示该数字系统和交换操作的算法状态机图。

10.19 (a) 对于由练习题10.18推导出的算法状态机图，画出另一个算法状态机图，该ASM图规定了所需的控制其数据路径电路的信号。假定用多路器来实现连接寄存器的总线，如图7-65所示。

(b) 写出练习题10.18中数字系统的完整的Verilog代码，包含(a)中描述的控制电路。

(c) 将(b)中的Verilog代码综合成电路，进行时序仿真，验证其功能的正确性。

10.20 在7.14.2节中，我们设计了一个处理器电路。画出描述该处理器功能的算法状态机图。

10.21 (a) 对于练习题10.20中得到的算法状态机图，画出另一个算法状态机图，该*ASM*图指定了控制该处理器的数据路径电路所需的控制信号。假设用多路器来实现总线与寄存器（从*R*0到*R*3）之间的连接。

(b) 编写练习题10.20中的数字系统的完整的Verilog代码，其中包括(a)中描述的控制电路。

(c) 将(b)中编写的Verilog代码综合成电路，进行时序仿真，验证其功能的正确性。

10.22 考虑在公路交叉口控制交通灯的电路。该电路产生输出信号*G*1，*Y*1，*R*1和*G*2，*Y*2，*R*2。它们分别表示两条路上红灯、绿灯和黄灯的状态。若某输出为1，则相应的灯亮。灯的控制服从下面的规则：*G*1点亮后必须保持一段时间t_1，然后熄灭。*G*1熄灭后*Y*1应该立刻点亮，它必须持续一段时间t_2再熄灭。当*G*1或*Y*1点亮时，*R*2必须点亮，且*G*2和*Y*2熄灭。*Y*1熄灭后，*G*2立即点亮并持续时间t_1。*G*2熄灭后*Y*2点亮并持续时间t_2，当*G*2或*Y*2点亮时，*R*1必须点亮，而*G*1和*Y*1必须熄灭。

(a) 画出描述该交通灯控制器的算法状态机图。假设已有两个递减计数器，一个用来测量t_1延时，另一个用来测量t_2延时。每个计数器都有并行载入和使能输入，用来加载合适的计数值分别表示t_1和t_2，并允许计数器递减至0。

(b) 画出其控制电路的算法状态机图。

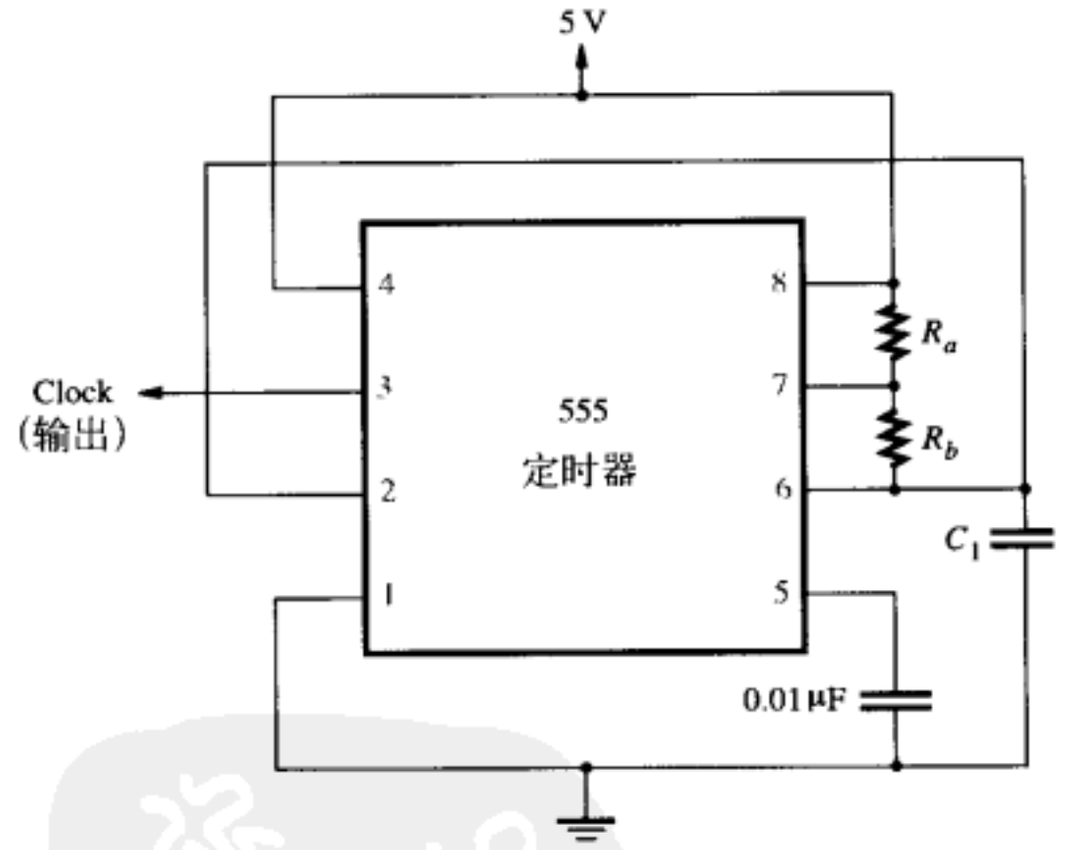
(c) 编写交通灯控制器的完整的Verilog代码，包括(a)中的控制电路和表示t_1和t_2的计数器。选用合适的时钟信号来同步电路，并将表示t_1和t_2的计数值设成合适的值。给出该电路的仿真结果。

10.23 假设你需要一个如图10-48a所示的单刀单掷开关。说明如何用计数器实现*Data*信号的防抖。（提示：设计一个有限状态机，*Data*作为输入，消除抖动后输出*z*。假定有102.4kHz的时钟信号可以使用。）

10.24 时钟信号通常由专用芯片产生，例如555可编程定时器（如图P10-2所示）。通过选择电阻R_a和R_b的阻值和电容C_1的容量，555定时器可以产生所需的时钟信号。时钟周期和占空比都是可调的。占空比指一个时钟周期里高电平所占的百分比。下面的等式定义了芯片所产生的时钟信号：

$$时钟周期 = 0.7\,(R_a + 2R_b)C_1$$

$$占空比 = (R_a + R_b) / (R_a + 2R_b)$$



图P10-2 555可编程定时器芯片

(a) 若要产生占空比为50%，频率约为500kHz的时钟信号，试确定R_a，R_b，C_1的值。

(b) 若要求占空比为75%，重新计算各值。

参考文献

1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed. (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed. (Morgan Kaufmann: San Francisco, CA, 2004).
3. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
10. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
11. C. J. Myers, *Asynchronous Circuit Design*, (Wiley: New York, 2001).

第11章　逻辑电路测试

在前面的几章中，我们讨论了逻辑电路的设计。在设计过程全部结束之后，我们希望所设计的电路能够如预期的那样工作。可怎样才能证明我们所设计的电路确实能完成预期的功能呢？证明这一点的关键就是要证明电路表现出了预期的电路行为，而且还要满足所有强加于该设计的时序约束条件。本书中已有几处讨论了时序问题。本章主要讨论用于验证给定电路功能的某些测试技术。

对逻辑设计进行测试有几个原因。电路刚开发完成的时候，需要验证它是否满足功能和时序要求。正确的设计在制造时会生产许多个管芯，每个管芯还需要测试，以保证在生产过程中未引入任何瑕疵。这是非常重要的。如果怀疑管芯中还存在某些问题，安装在现场的设备内的电路也要列入测试范围。

所有测试技术的基础都是将已定义好的输入信号集（称为tests或测试向量）加到电路上，并将其输出和预期功能正确的电路的表现相比较。难点就体现在怎样用相对较少的测试数量却能充分地证明电路的正确性。将所有可能的情况都写成测试的穷举法对于大型设计来说是行不通的，因为实在有太多的可能了。

11.1　故障模型

如果电路有故障，比如晶体管坏了或是某根连线接错了，电路的功能就会出错。很多东西可能出错，从而导致各种各样的问题。晶体管开关会出现故障，以致它永远保持开或者关的状态。电路里的连线可能短路，连接到电源（V_{DD}）或地上了，或者连线只是断开了。不该相连的两根连线也可能会连上。由于实现逻辑门的电路出现了故障，该逻辑门可能生成错误的输出信号。处理各种各样的错误问题是个劳神费力的麻烦事。幸运的是，我们可以将测试的过程限定在一些简单的故障中，从而得到大体上满意的结果。

11.1.1　固滞模型

这里讨论的大多数电路都使用逻辑门作为构建电路的基本部件。表示这种电路故障的理想模型是假设所有的故障都表现为一些连线（输入或输出门）被永远固定在逻辑0或1。连线w上传导着的信号并非是我们期望的，若该信号值一直对应着逻辑值0，这时则称w固滞于0（w is stuck-at-0），表示为$w/0$。若线w上传导着的信号不是我们想要的，其值一直为逻辑1，则称w固滞于1，表示为$w/1$。

举一个明显是固滞故障的例子，门的输入被错误地连接到电源电压上，即连接在V_{DD}或地上。还有一些其他类型的故障，它们引起的问题看上去就好像连线被固定在了一个特定的逻辑值上，固滞故障模型用于这些场合也能发挥作用。逻辑门电路出现故障所产生的实际影响取决于逻辑门电路制造时采用的特定工艺技术。我们将把注意力集中在固滞故障模型上，假设固滞故障是唯一会出现的故障，然后去检查测试过程。

11.1.2　单个故障和多个故障

电路之所以出现故障是因为电路出现一个或多个故障。同时处理多个故障很困难，因为每个故障都有几种不同的表现形式。实用的方法是只考虑单个故障。实际经验表明，一组可

以找出所有单个故障的测试集合，也能找出大部分多源故障。

若我们设计一组合适的信号用作测试激励，把它同时输入到存在故障的电路和无故障的好电路中，比较这两个电路产生的输出，则可以推知电路何处出现故障。假设每次测试都可以发现一个或几个故障。为某给定电路专门设计的能测试出全部故障的激励信号组就叫做测试集（test set）。

11.1.3 CMOS电路

CMOS（互补金属氧化物半导体）逻辑电路表现出一些特殊的故障行为。晶体管可能永久地保持开或关（亦称“短路”）的状态，从而失去作用。很多这类故障表现为固滞故障，但一些又会产生完全不同的行为。举个例子，晶体管出现短路故障时，可能会引起从V_{DD}到地的持续电流，而使输出电压处于中间状态，不是逻辑0也不是逻辑1。晶体管出现开路故障可能会使输出电容保持在充电状态，因为本来可让其放电的开关开路了，结果使组合逻辑电路表现出时序逻辑电路的行为。

CMOS电路的特定测试技术超出了本书的范围。在参考文献[1-3]中可以找到有关此课题的入门材料。在参考文献[4-6]中介绍了关于CMOS 电路测试的比较深入的研究材料。我们在本书中将假设使用固滞模型可以为所有电路产生具有足够故障覆盖率的测试集。

11.2 测试集的复杂度

组合电路和时序电路的测试有很大的不同。不管组合逻辑电路的设计规模如何大，我们都有办法进行充分的测试。而时序电路的测试则具有更大得多的挑战性，因为被测电路的行为不仅受到由外部输入的测试信号的影响，还与测试信号输入当时的电路状态有关。如果电路的设计者在设计时没有考虑到电路的可测性，就很难对时序电路进行测试。其实，设计比较容易测试的电路是可能的，我们将在11.6节中讨论这个问题。现在我们先从组合电路的测试入手。

组合电路的测试是很容易理解的，只需要在被测电路的输入端施加所有可能的激励信号，然后检查输出值是不是和真值表中定义的一致。这个方法对于小电路来说很管用，因为测试集不大。但是对于输入变量很多的大电路来说，这种方法就变得完全不可行。幸运的是，我们并不需要对具有n个输入端的电路施加所有2^n种可能的测试集。能够检测出所有单个故障的完整测试集合所需要的测试量往往远小于2^n种可能的测试集。

图11-1a 是一个简单的三输入电路，我们想要确定该电路的最小测试集。穷举测试集包括：所有三个输入信号的八个组合，牵涉到有五条连线的电路，这五条连线分别标记为a, b, c, d和f（见图）。每条连线都可以被固滞在0或1。

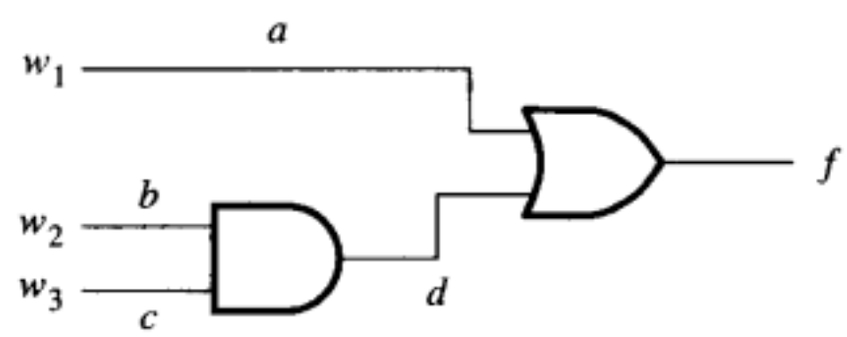
图11-1b列出了所有八种可能的输入组合作为电路的测试向量。测试向量$w_1w_2w_3 = 000$可以检测出连线a, d 和 f 固滞为1的故障。在正确的电路中，输入这个测试向量，输出结果$f = 0$。但是，如果任意一个$a/1$, $d/1$或者$f/1$故障出现，在输入向量为000时，电路就会输出$f = 1$。测试向量为001时，在正确的电路中，输出应该为$f = 0$，而若出现$a/1$, $b/1$, $d/1$ 或者 $f/1$故障，则输出变为$f = 1$。这个测试向量可以检测出四种故障的出现。我们说它*覆盖*了这些故障。最后的测试向量111只能检测出一个故障，$f/0$。

仔细检查图11-1b就能导出覆盖该电路所有故障的最小测试集。某些故障只能被一个测试向量覆盖，这就意味着必须把这些测试向量包含在测试集中。$b/1$故障只能被001覆盖。$c/1$故障只能被010覆盖。$b/0$, $c/0$和 $d/0$故障只能被011覆盖。所以，这三个测试向量是不可少的。对于剩下的故障而言，测试向量的选择余地就大了。选了测试向量001，010和011，则覆盖了除了$a/0$之外所有的故障。这个故障可以被三个不同的测试向量覆盖。再在余下的测试向量中

任意挑出一个，如100，便可以得到该电路的一个完整的测试集如下：

测试集 = {001，010，011，100}

由上得出结论，使用四个测试向量即可检测出该电路所有可能的固滞故障，没有必要使用八个测试向量，而如果我们只是简单地用完整的真值表来测试电路的话，就需要用到八个测试向量。



a) 电路

测试 $w_1w_2w_3$	故障检测									
	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$f/0$	$f/1$
000		√						√		√
001		√		√				√		√
010		√				√		√		√
011			√		√		√		√	
100	√								√	
101	√								√	
110	√								√	
111									√	

b) 各种输入取值检测出的故障

图11-1 简单电路的故障检测

对于给定的有n个输入端的电路来说，完整的测试集的大小通常远小于2^n。但是，在实际情况中这个测试集的规模仍然可能太大，无法接受。而且，生成最小测试集对于中等规模的电路已经相当困难。当然，图11-1这样简单的解决方案并不实用。下一节，我们要研究一种更有趣的解决途径。

11.3 路径的敏感化

考虑电路中所有连线的每个故障，从而推导出测试集，在11.2节我们曾这样做过。从实用的角度来看，这个方法并没有什么吸引力。因为需要考虑的连接线和故障实在太多了。另一种更好的方法是处理构成一条路径的几条连线，把这条路径作为一个整体来考虑，使用一个测试向量便可在这条路径上检测出几个故障。激活一条路径，使得沿着该激活路径传递的信号变化能直接对输出信号产生影响。

图11-2 给出了一条从输入w_1到输出f的路径。这条路径包含连线a, b, c和f，并经过3个门。在确保电路中其他路径不会影响输出f的值后，这条路径就被激活了。为此，必须把输入w_2置为1，以保证b处的信号只由a处的信号值决定；必须把输入w_3设为0，以保证w_3不会影响与或门的输出；必须把w_4设为1，使其不影响与门。这样，如果w_1=0，输出就为$f = 1$，反之$w_1 = 1$，则$f = 0$。于是这条路径就被激活了。在技术文献中，通常不用路径激活这种说法，而用更专门的术语来描述这一现象，我们说这条路径被敏感化（sensitized）了。

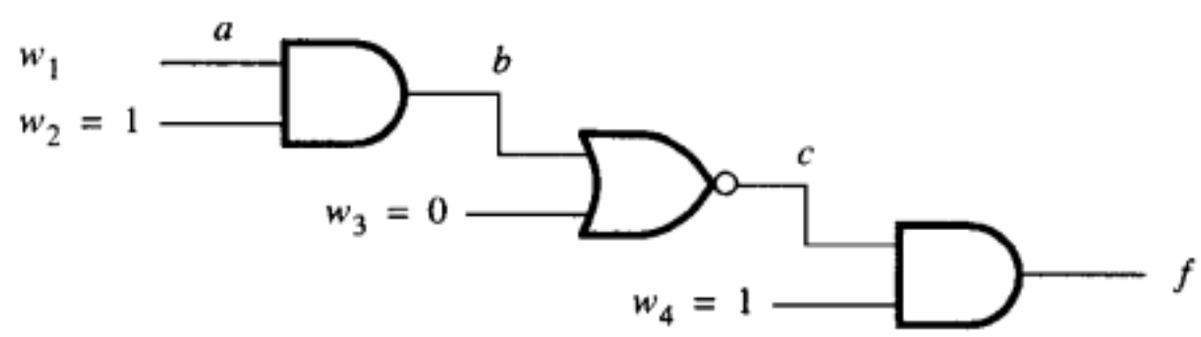


图11-2 一条敏感化的路径

为了使从与门或者与非门的某个输入端起始的路径敏感化，所有其他的输入端都要置1。为了使从或门或者或非门的某个输入端起始的路径敏感化，所有其他的输入端都要置0。

现在，让我们考虑沿着敏感路径由故障产生的影响。在图11-2中如出现$a/0$故障，则将导致$f = 1$（即使$w_1 = 1$也没有用）。如果发生$b/0$或$c/1$故障，会出现同样的问题。因而测试量$w_1w_2w_3w_4 = 1101$可以检测出$a/0$, $b/0$ 和$c/1$故障。类似地，如果$w_1 = 0$，输出应该是$f = 1$。但如果任意一个$a/1$，$b/1$或$c/0$出现了，则输出$f = 0$。所以，这三个故障可以用测试向量0101测出。沿着敏感路径的任意一个固滞故障只用这两个测试向量就可以检测出来。

给定电路中路径的数量比单根连线的数量要少得多。这就意味着基于敏感路径产生测试集的方法也许会有更大的吸引力。下面的例子详细说明了这种可能性。

例11.1（敏感路径的测试） 考虑图11-3中的电路。这个电路中有5条路径。当$w_2 = 1, w_4 = 0$时，路径w_1-c-f就被敏感化了。w_3是0是1都没有关系。因为$w_2 = 1$使得连线b上的信号等于0，从而不管w_3怎样，d都被强制等于0。由上，设置$w_2w_3w_4 = 1x0$（x 意味着w_3的值无关紧要），这条路径被敏感化了。现在测试向量$w_1w_2w_3w_4 = 01x0$ 和 11x0就可以检测出这条路径上的所有故障了。第二条路径，w_2-c-f 可以由1000和1100测出。路径w_2 -b- d -f可以由0010和0110测出。路径w_3- d -f可以由x000和x010测出。第五条路径，$w4$-f 由0x00和0x01测出。总共不到十个测试向量就足以测试全部路径。可以看到，0110可以起到01x0的作用，1100起到11x0的作用，1000起到x000的作用，0010起到x010的作用。因此，整个测试集就是：

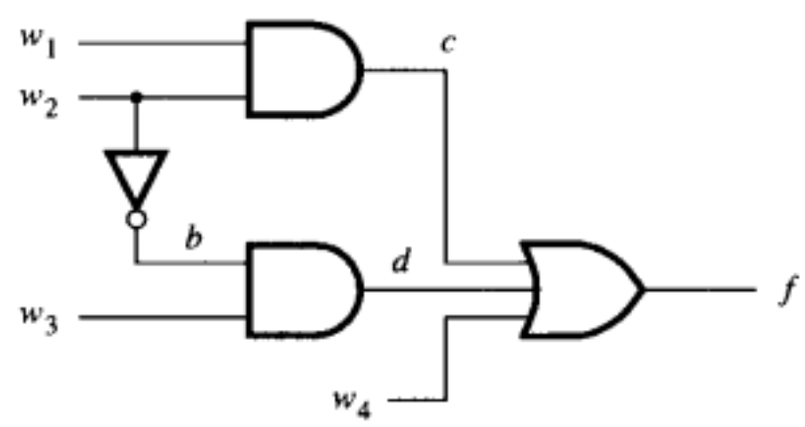


图11-3 例11.1的电路

$$测试集 = \{0110,1100,1000,0010,0x00,0x01\}$$

虽然这个解决途径比较简单，但在大规模电路中它仍然不实用。但是路径敏感化这个概念非常有用，在接下来的讨论中我们将能看到这一点。 ■

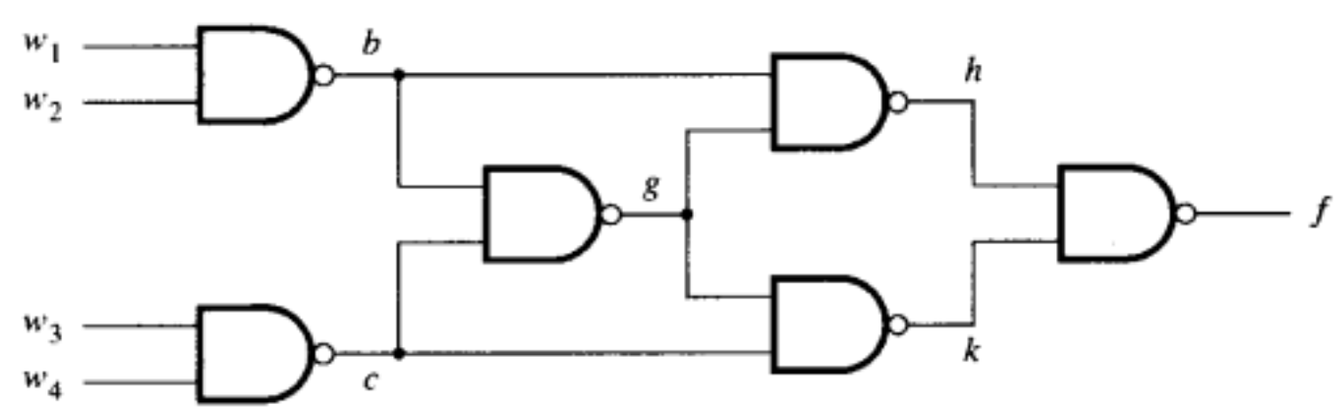
特定故障的检测

假设我们怀疑图11-3中的电路有一个故障，即连接线b被固滞在了1。由故障产生的影响经过路径传递最终有可能到达输出f，将故障传递路径敏感化后，在f处我们可以观察到由某个故障产生的影响，用这样的方法可以推导出测试故障覆盖的测试向量。设置$w_3 = 1, w_4 = 0$和$c = 0$，从b到d到f的路径才能传递信号。要使$c = 0$，只需使$w_1 = 0$即可。如果b被固滞在1，还需要再添加一个测试输入。这个输入使得连线b上的值变为0，于是好电路与故障电路的输出值就不同了，故将w_2设置1。因此，检测$b/1$故障的测试向量为$w_1w_2w_3w_4 = 0110$。

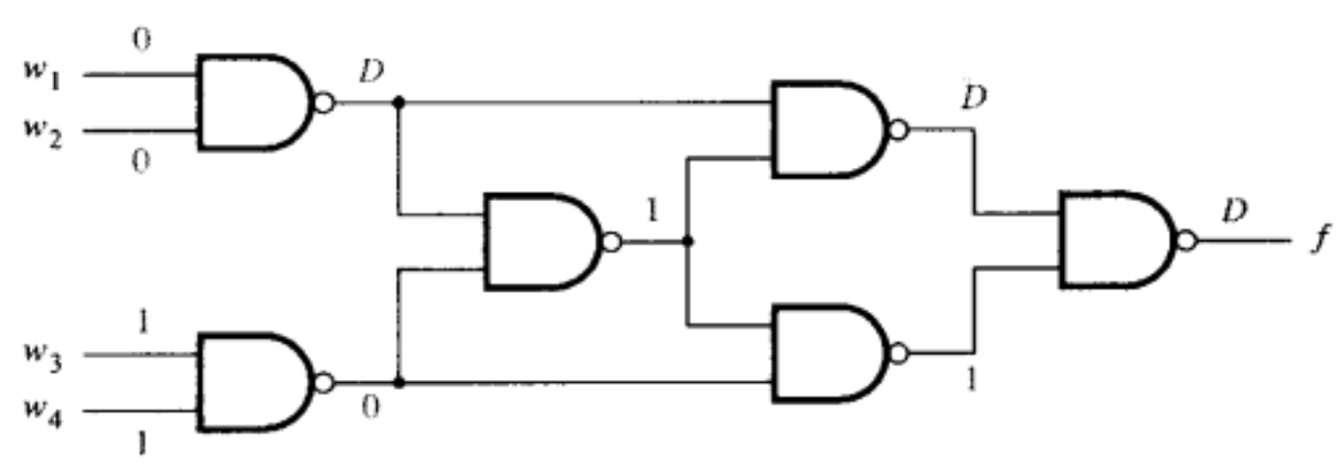
将故障的影响传递到输出（即将信号路径敏感化）的方法，通常可以用来检测给定连线上的故障。这项工作需要我们给沿着路径的各个门的其他输入端赋值才能做到。这些值必须通过给原始输入端赋特定的值而取得，这一点不总是可以做到的。例11.2详细说明了这个过程。

例11.2（故障的传递） 故障的影响沿着敏感化路径上的各个门传递，如果它经过了一个非门，信号的极性就会改变。设符号D表示固滞为0的一般故障。在经过与门和或门时，固滞为0的故障的影响不会改变。如果D是与门（或门）的一个输入，其他输入被设置为1(0)，则此门的输出就与D的行为完全一致。但是，如果D作为非门、与非门或者异或门的一个输入，输出就会表现出固滞为1故障，表示为$\overline{D}$。

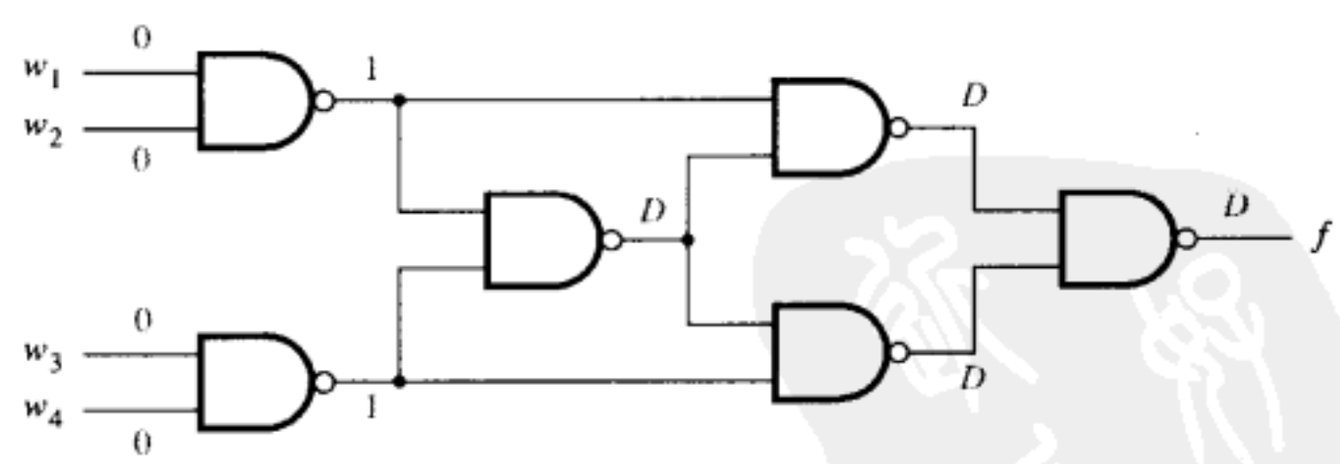
图11-4展示了使用D和$\overline{D}$符号来解释故障的影响是如何传递的。首先假设线b上出现固滞为0故障，记为$b/0$。我们想让该故障的影响沿着路径b-h-f传递。这由图11-4b所示即可得。设置$g = 1$，故障的影响传到连线h。h表现为固滞在1，表示为$\overline{D}$。接下来，设置$k = 1$，故障的作用传播到f。因为最后的与非门使信号值翻转，输出就等于D，也就是$f/0$。这样正确的电路输出应该为1，而有故障的电路则输出0。接下来，我们要确认$g = 1$和$k = 1$，这可以由给最初始输入变量赋合适的值而得到。这被叫做一致性检查。通过设置$c = 0$，g和k都被强制为1，而设置$w_3 = w_4 = 1$则令$c = 0$。最后，为了使得故障D能传播到线b上，就需在电路上加信号使b值为1，这只要使w_1或w_2之一等于0就可做到。因此，测试向量$w_1w_2w_3w_4 = 0011$便可以检测出故障$b/0$。



a) 电路



b) 检测$b/0$故障



c) 检测$g/1$故障

图11-4 故障的检测

接下来假设连线g固滞在了1，表示为$\overline{D}$。我们将b和k置1，以使此故障沿着路径g-h-f传播。为使$b = 1$，设置$w_1 = w_2 = 0$。为使$k = 1$，令$c = 0$。但是我们还得通过加个信号将$\overline{D}$故障传播出去。这个信号在正确的电路中应该使$g = 0$。这只能在$b = c = 1$时才能得到。问题就是在同时我们还需要令$c = 0$，以使$k = 1$。所以，一致性测试失败了，$g/1$故障在这条路径上不能传递。

另一种可能是同时沿着两条路径传递故障的影响，如图11-4c所示。在这种情况下，故障沿着路径g-h-f和g-k-f传递。这需要设置$b = 1$，$c = 1$，这正好就是上面我们解释过的情况。测试向量0000就可以检测出$g/1$，我们的目标也就达到了。观察得到，若与非门的两个输入都接D（或$\overline{D}$），则输出值就是$\overline{D}$（或是D）。

路径敏感化可以用来传递故障的影响。人们曾多次试图利用这种想法以推导出高效故障测试集的生成方法。图11-4说明了D-算法的主要精神。这个算法是最早的实用故障检测方法之一[7]。以这个基本方法为起点，也发展出一些其他技术[8]。 ■

11.4 树状结构的电路

树状结构电路，即每个门只有一个扇出，是很容易测试的。积之和（sum-of-product）或者和之积（product-of-sum）是这种电路最普遍的形式。因为从每一个原始的输入端到电路的输出端都有一条特定的路径，所以只根据原始输入端，就可以生成故障测试向量。我们可以用图11-5中的与或项逻辑图（积之和图）来解释这个概念。

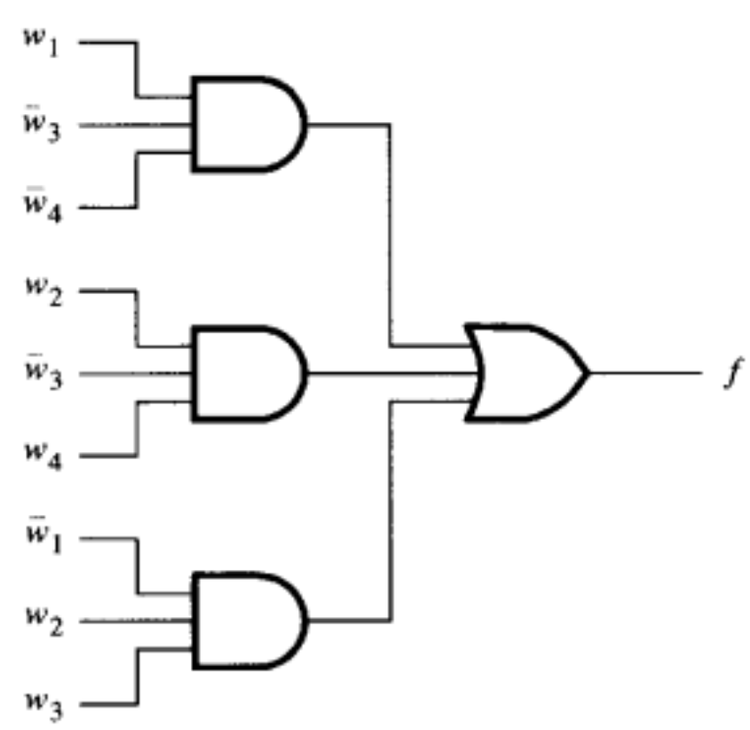


图11-5 树状结构的电路

如果与门的任意一个输入端固滞在0，用下面的方法可以检测出这个故障。将这个门的所有输入端都置1，并使其他的与门都输出0。在正确的电路中这样做使$f = 1$，而在故障的电路中$f = 0$。因为有三个与门，所以这样的测试向量需要三个。

测试固滞为1的故障要稍微复杂一些。测试与门的一个输入端是否出现了固滞为1故障的方法是：用逻辑0来驱动这个输入端，而其他输入端都连接逻辑1。如果这个门没问题，则输出为0；有问题，则输出为1。同时，其他与门必须产生输出0，只要使这些与门的输入端中至少有一个为0即可。

图11-6表示了怎样生成需要的测试向量。前面的三个测试向量用来测试固滞为0故障。第四个测试向量可以检测最上面那个门的第一个输入端或者其他两个门的第三个输入端是否出现了固滞为1故障。观察到，在每种情况下，被测试的输出端由逻辑0驱动，而其他输入端都等于1。这就引入了测试向量$w_1w_2w_3w_4 = 0100$。很明显，我们要使用一个测试向量来测试尽量多的输入。第五个测试向量可以检测出最上面的门的第二个输入端和最底下那个门的第一个输入端；它并不测试中间那个门的任何输入端。要用的测试向量为1110。还需要三个测试向量来检测这些与门还没检测到的输入端是否出现了固滞为1的故障。这样，完整的测试集为：

	编号	积项			测试
		$w_1\overline{w}_3\overline{w}_4$	$w_2\overline{w}_3w_4$	$\overline{w}_1w_2w_3$	$w_1w_2w_3w_4$
固滞0测试	1	1 1 1	0 1 0	0 0 0	1 0 0 0
	2	0 1 0	1 1 1	1 1 0	0 1 0 1
	3	0 0 0	1 0 1	1 1 1	0 1 1 1
固滞1测试	4	0 1 1	1 1 0	1 1 0	0 1 0 0
	5	1 0 1	1 0 0	0 1 1	1 1 1 0
	6	1 1 0	0 1 1	0 0 0	1 0 0 1
	7	1 0 0	1 0 1	0 1 1	1 1 1 1
	8	0 0 0	0 0 1	1 0 1	0 0 1 1

图11-6 电路（见图11-5）测试集的推导

测试集 = {1000，0101，0111，0100，1110，1001，1111，0011}

11.5 随机测试

到目前为止，我们只考虑了怎样为给定的电路生成一个确定性的测试集。我们主要依靠

了路径敏感化这个概念。一般说来，当电路的规模变得更大时，生成这样的测试集会很困难。换一种方法来做会更好，我们将随机生成这些测试向量。本节着重探讨这种方法。

w_1w_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

图11-7 所有二变量函数

图11-7给出两个变量能组成的所有函数。对一个n变量的函数，存在着2^{2n}个可能的函数；对于$n = 2$，则存在2^{2^2}=16种二变量函数。以异或函数为例，其逻辑原理图如图11-8所示。我们现在考虑该电路中连线b、c、d、h和k上可能出现的固滞为0和固滞为1的故障。如图11-9所示，每个故障都将使该电路变成故障电路，即电路所实现的功能不再是异或逻辑。为了测试这个电路，我们可以施加一个或多个输入向量，用以把正确的电路与图11-9所列出的那些故障电路区分开。任意选择$w_1w_2 = 01$作为第一个测试向量。这个测试向量可以把可能正确的电路和确实有故障的电路区分开。输出$f = 1$的电路有可能是正确的，而输出$f = 0$的电路一定存在故障，所以可以确定f_0、f_2、f_3和f_{10}为故障电路。再选择测试向量$w_1w_2 = 11$。这个测试向量可以把可能正确的电路和确实有故障的电路区分开，f_5、f_7和f_{15}一定存在故障。其中，f_3在使用$w_1w_2 = 01$时已经被测试过了。第三个测试量我们选择$w_1w_2 = 10$。它可以把可能正确的电路和确实有故障的电路区分开，f_4和f_{12}一定存在故障。这三个测试向量都是随机抽取的，但已能检测出图11-9中列出的所有电路故障。而且请注意：前两个测试向量就能从九种可能的电路故障中分辨出七种。（译者注：原文中这一段文字意思表达不清楚，存在不少问题，一般的读者理解有困难。译者根据自己的理解在原文的基础上做了适当的修改。请注意，因为f_1、f_6、f_8、f_9、f_{11}、f_{13}、f_{14}的故障不会出现，但原文中没有提到，译者提醒读者先理解这一点。）

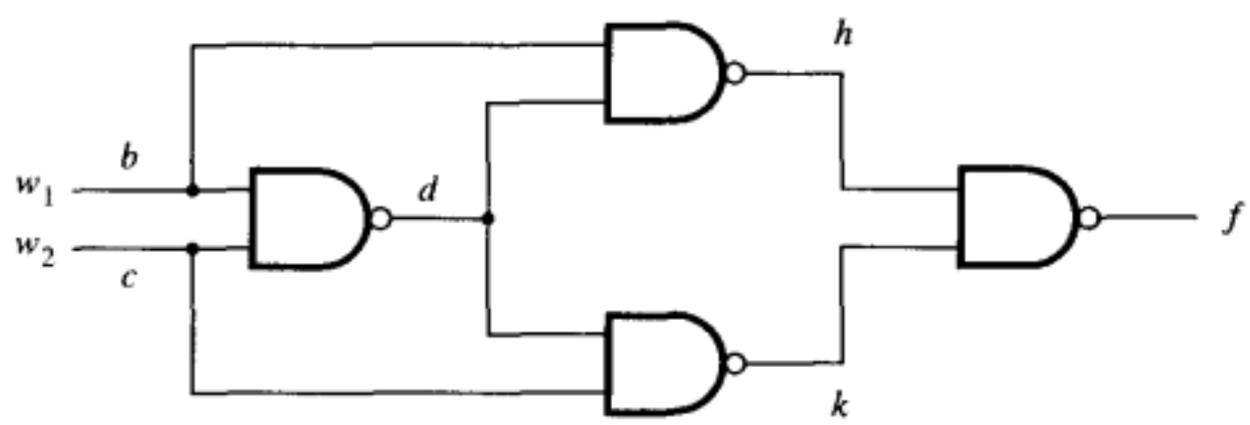


图11-8 异或电路

这个例子说明随机抽取的测试向量合在一起也可能生成合适的测试集。随机抽取测试向量的效率又如何呢？从图11-7我们看到，用任意四个可能的测试向量就可以将输出正确的函数从八个输出错误的函数中分辨出来，因为它们的输出是不一样的。这八个错误函数可被一个测试向量检测出来的数量占全部可能函数数量的一半（对于两个变量的情况就是2^{2^2-1}）。这个测试是分辨不了正确的函数和七个输出值与正确函数输出值相同的故障函数的。施加第二个测试向量，可分辨出剩下的七个故障函数中的四个，因为它们的输出和正确的不一样。依此类推，加一个新的测试量就能将未被检测出来的测试量数目减半。所以，先施加的几个测试向量检测出大部分可能故障的机率是很高的。更加专业地说，每个故障电路被第一个测试向量检测出来的可能性是：

$$P_1 = \frac{1}{2^{2^2}-1} \cdot 2^{2^2-1} = \frac{8}{15} = 0.53$$

这就是输出值与正确电路不同的故障电路的个数和故障电路总个数的比值。

这个推理过程可以扩展到n个变量的函数中。此时，第一个测试向量总共检测出$2^{2^n}-1$个可能故障函数中的2^{2^n-1}个。这样，如果施加m个测试向量，故障电路可能被检测出来的概率为：

$$P_m = \frac{1}{2^{2^n}-1} \cdot \sum_{i=1}^{m} 2^{2^n-i}$$

以图的形式表示这个表达式，见图11-10。结论是：随机测试是很有效率的，即使在非常大的电路中，施加了几十个测试向量后，故障就有可能被检测出来。

故障	电路实现
$b/0$	$f_5 = w_2$
$b/1$	$f_{10} = \overline{w}_2$
$c/0$	$f_3 = w_1$
$c/1$	$f_{12} = \overline{w}_1$
$d/0$	$f_0 = 0$
$d/1$	$f_7 = w_1 + w_2$
$h/0$	$f_{15} = 1$
$h/1$	$f_4 = \overline{w}_1 w_2$
$k/0$	$f_{15} = 1$
$k/1$	$f_2 = w_1 \overline{w}_2$

图11-9 不同故障产生的影响

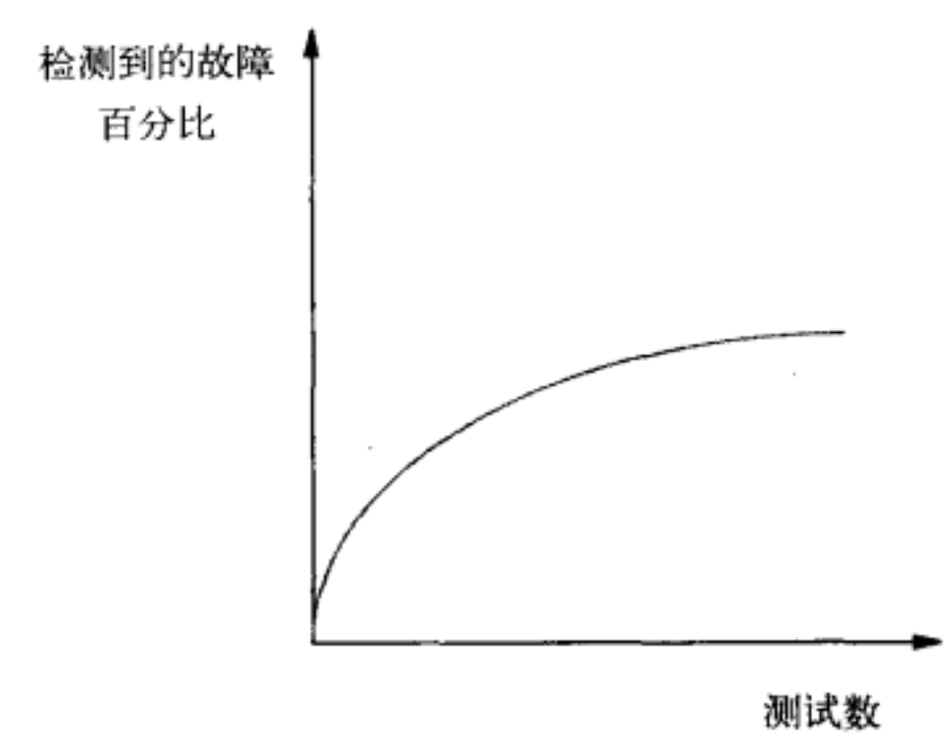


图11-10 随机测试的效率

对于扇入不很高的电路来说，随机测试尤其有效。如果扇入高，可能有必要恢复使用其他测试方法。例如，假设所有的与门都有很多个输入端。在它的输入端检测固滞为1的故障就有困难。随机测试就不一定能检测出存在的故障。但是用11.4节介绍的方法倒是有可能检测出这些错误。

随机测试的简单性是一个非常吸引人的特色。因为这个原因，再加上测试集的高效率，这种测试技术在实际工作中经常用到。

11.6 时序电路的测试

从前几节我们可以看到，无论确定的测试集或者随机的测试集，对于组合电路的测试来说都是很有效的。可时序电路的测试却困难得多。存储元件的出现允许时序电路处在不同的状态，而且电路对外部测试输入信号的响应取决于电路所处的状态。

比较组合电路的行为和其真值表定义的行为，我们就可以进行组合电路的测试。我们也采用相同的方法测试时序电路，比较时序电路的行为和状态表指定的行为。时序电路的测试必须检查电路各状态之间的转换是否正确，输出是否正确。这个方法看似简单，实际上却极其复杂。最大的问题是，大多数情况下，从电路的外部引脚很难观察到状态变量；而如果状态变量在电路的外部引脚上很难观察到，我们就很难确定电路处在什么状态。尽管如此，我们仍然需要充分地验证状态的转移已正确地完成。对于非常小的电路这种方法可能奏效，可对于实际规模的电路来说，这种方法却不可行。更好的方法是在设计时序电路时就考虑到使测试比较容易进行。

可测试性设计

同步时序电路由产生输出和下一个状态函数的组合电路，以及在时钟周期中能保持状态

信息的各触发器组成。时序电路的总体模型见图8-88所示。组合网络部分的输入包括原始输入（从w_1到w_n）和当前状态的变量（从y_1到y_k）。输出包括原始输出端（从z_1到z_m），和下一个状态变量（从Y_1到Y_k）。如果我们可以在组合网络部分的所有输入端加上测试向量，而且在其所有输出端又能看到结果，这个组合网络就可以用前几节所讲的方法来测试。问题是，如何才能将测试向量加到当前状态的输入端上，如何在下一个状态的输出端上去观察呢？

我们可以这样做，在每一个当前状态的变量的路径上添一个二选一多路器，这样组合网络的输入可以在状态变量的值和部分测试向量的值之间做选择。状态变量由相应触发器的输出得到。这个方法有一个很大的缺陷，即每一个多路器的第二个输入（测试向量）要由外部引脚直接得到，如果测试向量很多的话，就需要很多的引脚。换一种方法，我们提供这样一种连接，允许将测试向量每次一位地移入电路，就可以减少引脚的数量，但是这样做是以测试时间的延长为代价的。人们提出好几种基于这一思想的方案。下面介绍其中的一种。

路径扫描技术

有一种称为路径扫描的流行技术，该技术在触发器的输入端添加了多路选择器，若选择正常操作方式，则触发器成为时序电路中的状态寄存器，若选择移位方式，则触发器被用做测试向量移位寄存器电路的一部分。图11-11展示了用3个触发器构成的常规路径扫描电路。每个触发器的D输入端通过一个二选一多路器与相应的下一个状态的变量相连，或与一个串行路径相连。这个串行路径将所有的触发器连接成一个移位寄存器。控制信号 $\overline{Normal}/Scan$ 选择多路器的输入是哪个。在正常（normal）操作中，触发器的输入由下一个状态的变量Y_1、Y_2和Y_3驱动。

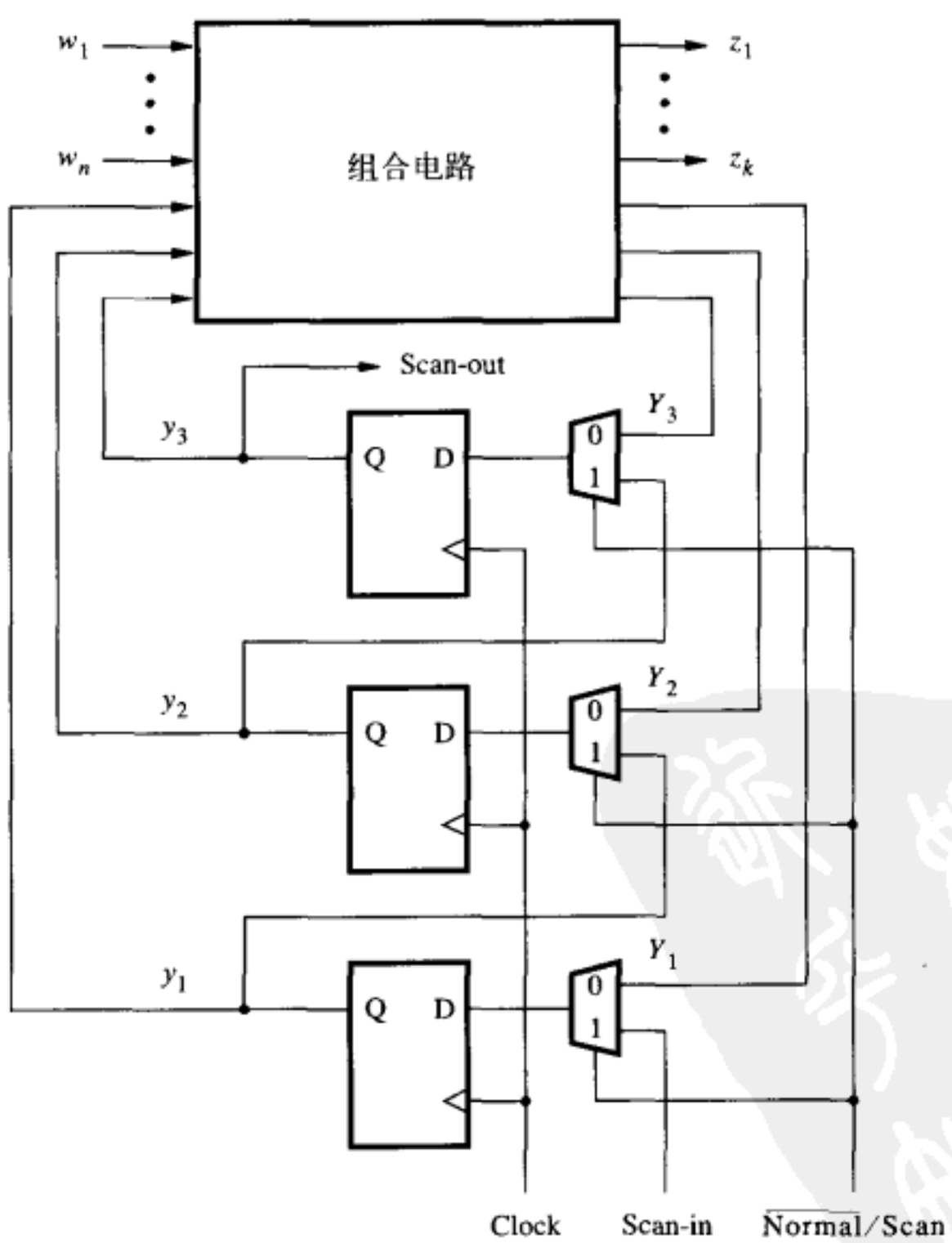


图11-11 路径扫描的安排

为了便于测试，把寄存器连接成为移位寄存器的形式（即把Q_i连接到D_{i+1}上），通过逐位移位的方式把当前状态的变量Y_1、Y_2和Y_3移到（扫描）输入测试端口上成为测试向量的一部分。第一个触发器的输入可以由外部引脚（Scan-in）引入。输出从最后一个触发器给出，引脚为Scan-out。

路径扫描技术包括以下几步：

1) 将由0、1组成的一串数字，比如01011001，按时钟周期依次扫描（移位）进去，观察扫描（移位）输出的字串是不是一样，以测试触发器是否工作正常。

2) 在$w_1w_2\cdots w_ny_1y_2y_3$上施加测试向量，观察$z_1z_2\cdots z_mY_1Y_2Y_3$上生成的值，以测试组合电路部分。具体操作步骤如下：

- 设置$\overline{Normal}/Scan=1$，用三个时钟周期将测试向量中的一部分：$y_1y_2y_3$扫描进触发器。
- 设置$\overline{Normal}/Scan=0$，时序电路处在正常工作模式，在一个时钟周期里，测试向量的$w_1w_2\cdots w_n$部分就得到了赋值。观察输出$z_1z_2\cdots z_m$。这时得到的值$Y_1Y_2Y_3$下载到触发器中。
- 设置$\overline{Normal}/Scan=1$，触发器的内容在3个时钟周期内被逐位移位（扫描）到输出端，使得测试结果的$Y_1Y_2Y_3$部分可以在外面观察到。同时，可以把下一个测试向量逐位移位（扫描）进入触发器以节省总的电路测试时间。

下面的例子展示了为进行路径扫描测试而设计的特殊电路。

例11.3 图8-78所示的电路可以辨识出特定的输入序列，如8.9节所述，该电路经修改后添加了扫描路径，如图11-12所示，使得测试变得更加容易了。其组合电路部分没有改变，包括4个与门和2个或门，在这两个图中都是一样的。

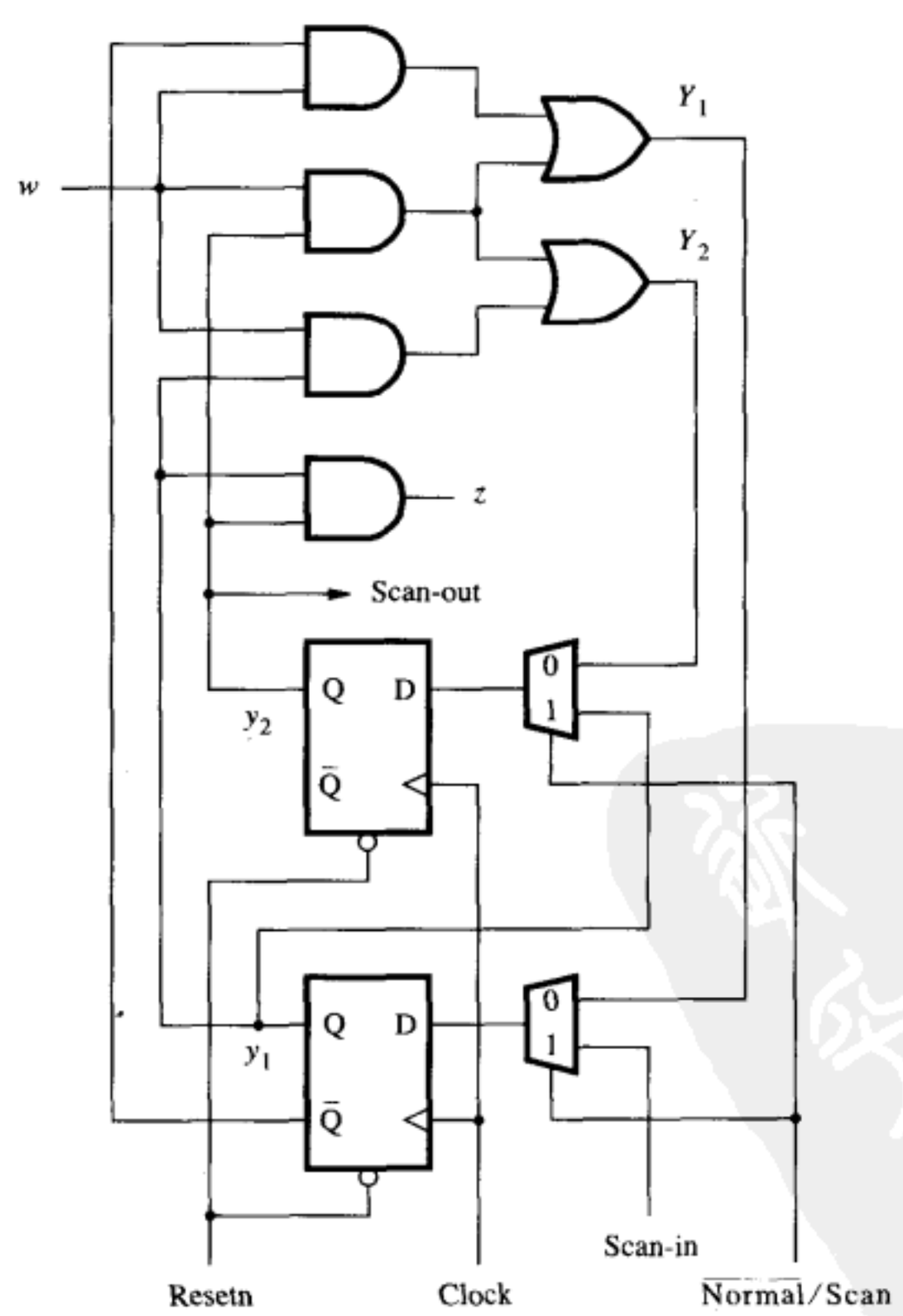


图11-12 例11.3的电路

触发器可以通过如上面所述的那样把一串0和1的序列扫描到其中，看是否能按原样输出用以测试触发器。在w、y_1和y_2上施加测试向量来测试组合电路。我们来看随机测试的方法。为wy_1y_2随机选择了四个测试向量$wy_1y_2 = 001, 110, 100$和111。为了施加第一个测试向量，先用两个时钟周期将$y_1y_2 = 01$扫描进触发器。然后用一个周期，输入$w = 0$使电路工作在正常状态下。这样就将测试向量$wy_1y_2 = 001$加入到与或门组成的电路上。这个测试的结果是$z = 0$，$Y_1 = 0$和$Y_2 = 0$。z的值可直接观察到。Y_1和Y_2的值可下载到相应的触发器里，并在接下来的两个时钟周期里扫描到输出端。当这些值全都被扫描输出后，下一个测试向量中$y_1y_2 = 10$就可以被扫描进触发器了。这样就需要5个时钟周期完成一个向量的测试，但是最后两个周期可以与第2个向量的测试重叠。第3个和第4个向量的测试都如此进行。所有这4个向量的测试需要的总时间是14个时钟周期。 ■

上述方法是基于前几节讲述过的测试（时序电路的）组合逻辑部分的方法来测试时序电路的。扫描路径方法也可以用来测试时序电路，通过依次观察状态表中所有状态的转移来实现。只要将表示状态的状态变量值扫描进入各触发器，就可以很容易地将电路置于某给定状态。通过观察其外部输出，并且使扫描输出呈现目标状态的值就可以检测状态转移的结果。有关这个方法的具体细节留给读者思考（参见习题11.16）。

扫描路径法的局限性在于：若在正常操作中使用了触发器的异步置位和复位功能，则该方法就不能正常发挥作用。我们已经建议过最好使用同步的置位和复位的方法。若设计者想使用异步置位和复位方式，则可以使用例如*电平敏感扫描设计*（level-sensitive scan design）等设计技术来设计可测试电路[1,9]。读者可以参考介绍这一技术的文献。

11.7 内建自测试

到目前为止，我们假设逻辑电路的测试是这样进行的：外部施加测试输入并将其输出结果与电路的期望行为作比较。这要求被测电路必须连接外部测试设备。一个有趣的问题是：我们是否可让电路本身具有可测试能力，因而就不再需要外部测试设备？这种内建的测试能力使得电路自己本身就能完成测试。本节将讲述提供*内建自测试*（built-in self-test，BIST）的一种方案。

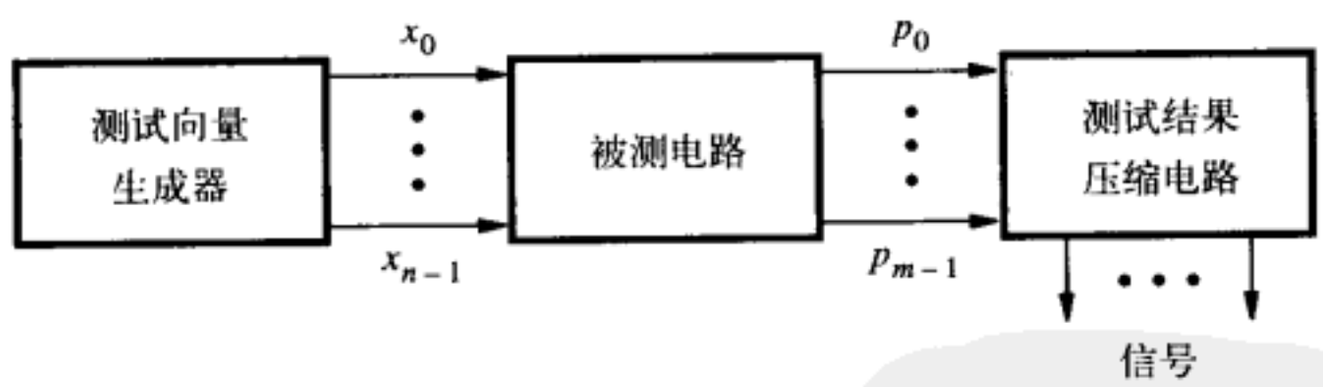
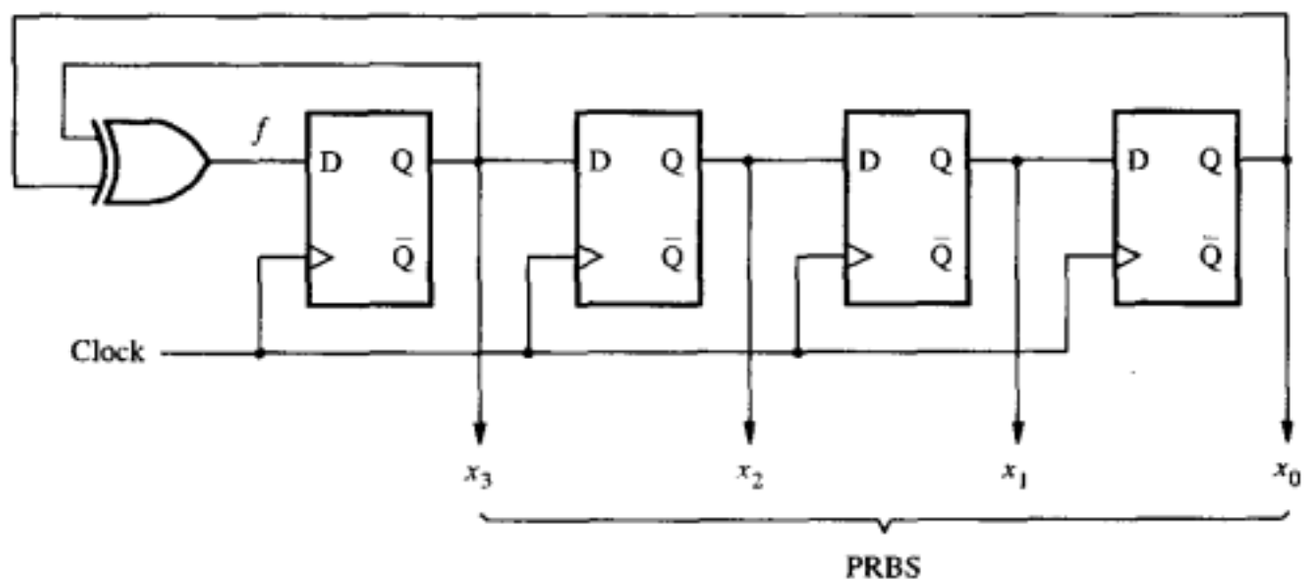


图11-13 测试的安排

图11-13是一种内建自测试结构的示意图，其中测试向量生成器生成测试电路时所需要的测试向量。在11.5节中，我们曾提到随机测试向量的测试效果不错，其故障覆盖率取决于进行测试的次数。对于施加在电路上的每一个测试向量，必须确定电路应产生的响应。正常电路的响应可以通过CAD工具的仿真得到。而测试所期望的响应必须预先保存在芯片中，以便电路测试时能进行比较。

在芯片上生成测试向量的实际方法是使用伪随机测试。伪随机和随机测试就随机性而言具有相同的特点，但伪随机（测试生成器）产生的测试向量是固定的并可以随意重复。用带反馈的移位寄存器很容易实现伪随机测试生成器。图11-14示意了一个小型的可使用的伪随机测试向量生成器。一个4位的移位寄存器，第1级和第4级的输出信号通过一个异或门反馈到第

1级的输入。随着连续的时钟周期信号，可以生成15个不同的测试向量。若刚开始就将移位寄存器设置为：$x_3x_2x_1x_0 = 1000$，则生成的测试向量就如图11-14b所示。请注意，0000不能用做移位寄存器的初始状态，因为这样做，电路就会永远死锁在该状态。



a) 电路

x_3	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	…
x_2	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	…
x_1	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	…
x_0	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	…
f	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	1	…

b) 生成的序列

图11-14 伪随机二进制序列发生器（PRBSG）

图11-14的电路隶属于称为线性反馈移位寄存器（linear feedback shift registers, LFSR）的一类电路。从n位移位寄存器的不同级通过异或门反馈连接到电路第1级，可能生成2^n-1种测试码序列，这些测试码具有随机数的特性。这样的电路可广泛用作产生纠错码。其电路的工作原理在很多参考书中都有介绍[1~3，10]。Peterson 和Weldon在书[11]中给出了n种不同反馈连接所能生成的最大长度的伪随机序列的测试码表。

伪随机序列生成器（pseudorandom binary sequence generator），简称PRBSG，给出了一个简单的生成测试向量的方法。该方法要求测试时电路应有的响应由CAD仿真工具生成。接下来的问题是如何检查电路确实产生这样的响应。在芯片上保存大量的测试的响应并不可行，芯片上最重要的部分当然还是主电路部分了。一种实用的解决办法就是将测试的结果压缩成一个代码。用线性反馈移位寄存器（LFSR）电路可以实现这一点。LFSR电路不只是把反馈信号作为输入，压缩电路还包含有被测电路产生的输出信号。图11-15表示的是一个单输入压缩电路（single-input compressor circuit，SIC）。其反馈连接和图11-14的伪随机序列生成器一样。输入端p就是电路测试时的输出。当施加了一些测试向量后，p端的结果值驱动单输入压缩电路（SIC），并且按照线性反馈移位寄存器（LFSR）的功能，生成一个4位的码。单输入压缩电路（SIC）生成的码叫做被测试电路相对给定序列测试向量的签字（signature）。该签字表示一个简单的模式，可以解释为所有加上去的测试向量的结果。将其与一个事先已知的正确码比较，就可以知道被测试电路的工作是否正常了。保存一个只以比较为目的的n位码，只需要付出很小的代价。由于LFSR的压缩电路的随机特性，所以可产生较全面的由电路故障引起的代码覆盖[12]。

若被测电路有多个输出，则可以用多输入的线性反馈移位寄存器（LFSR）。图11-16说明4个输入端口（从p_0到p_3）是如何施加到图11-14的基本电路中的。4位签字提供了一个很好的机制，能很好地区分4位代码不同序列的差别。这些差别可在多输入压缩电路（multiple-input

compressor circuit，MIC）的输入端显现出来。

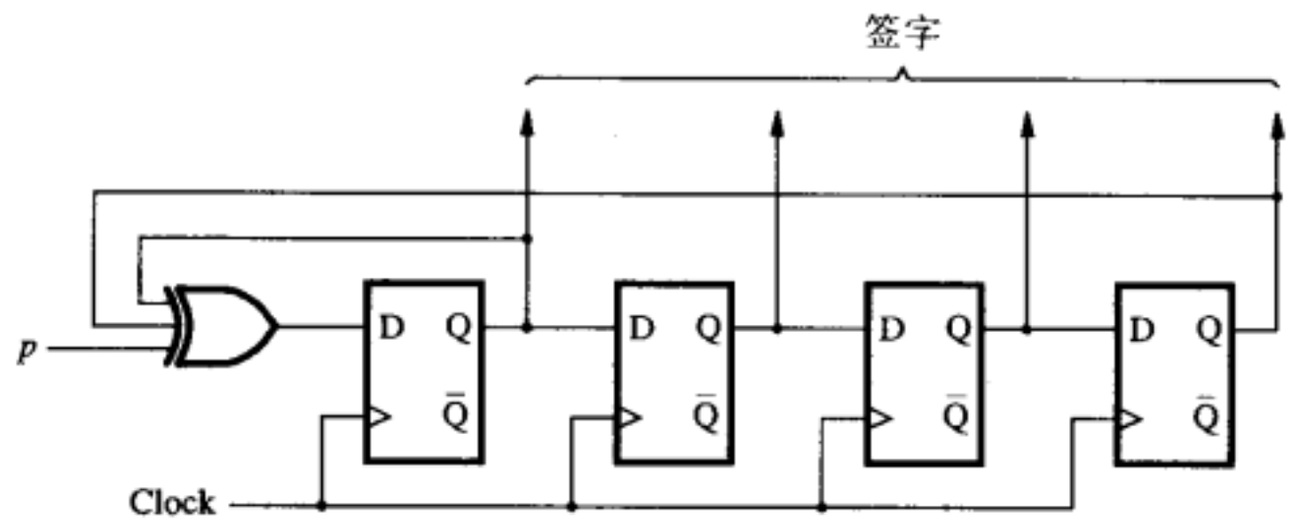


图11-15 单输入压缩电路（SIC）

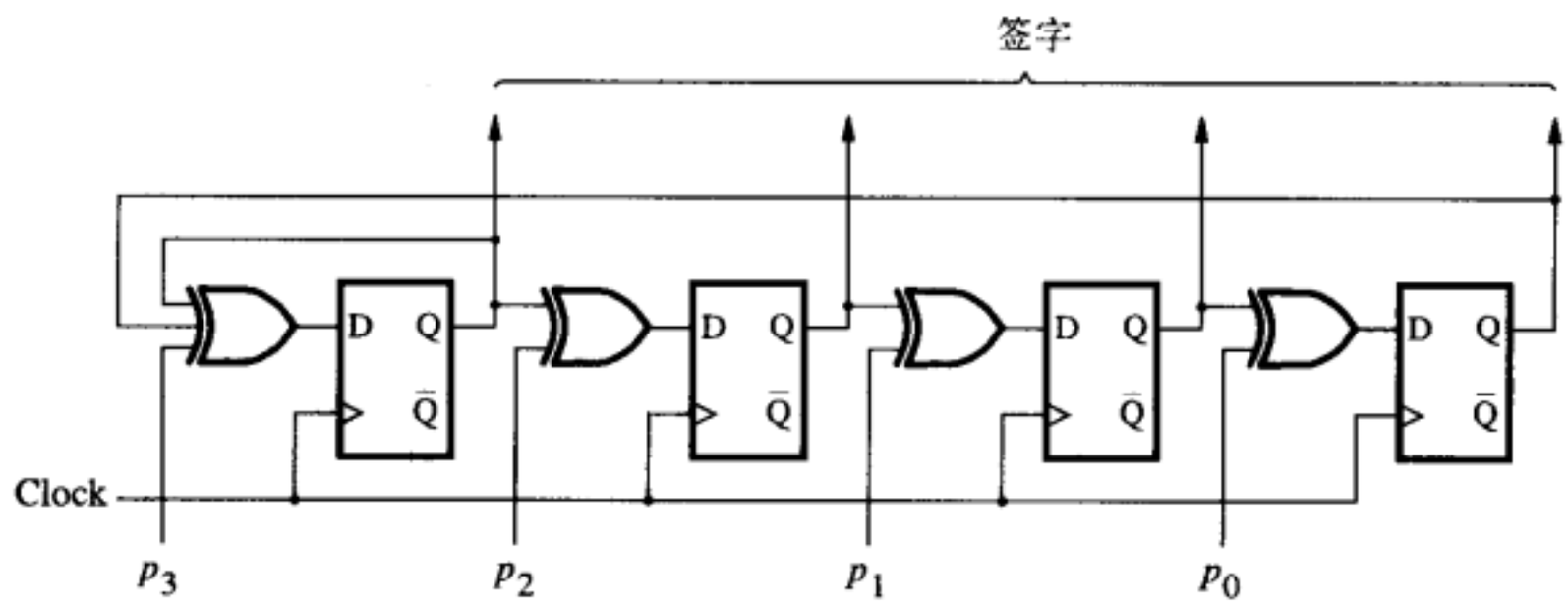


图11-16 多输入压缩电路（MIC）

时序电路的一种完整的内建自测试方案可以如图11-17那样实现。扫描路径方案可以提供一种可测电路。测试向量不是像原来那样施加在原始输入端$W = w_1w_2\cdots w_n$上，而是由内部生成作为$X = x_1x_2\cdots x_n$端的输入。用多路器可实现从W到X的切换作为组合电路的输入。伪随机二进制序列生成器（PRBSG-X）生成X的测试码。经由下一信号所施加的那部分测试码y，由第二个伪随机二进制序列生成器（PRBSG-y）产生。这些测试码被逐位扫描进入各触发器中，请参阅11.6节所述。

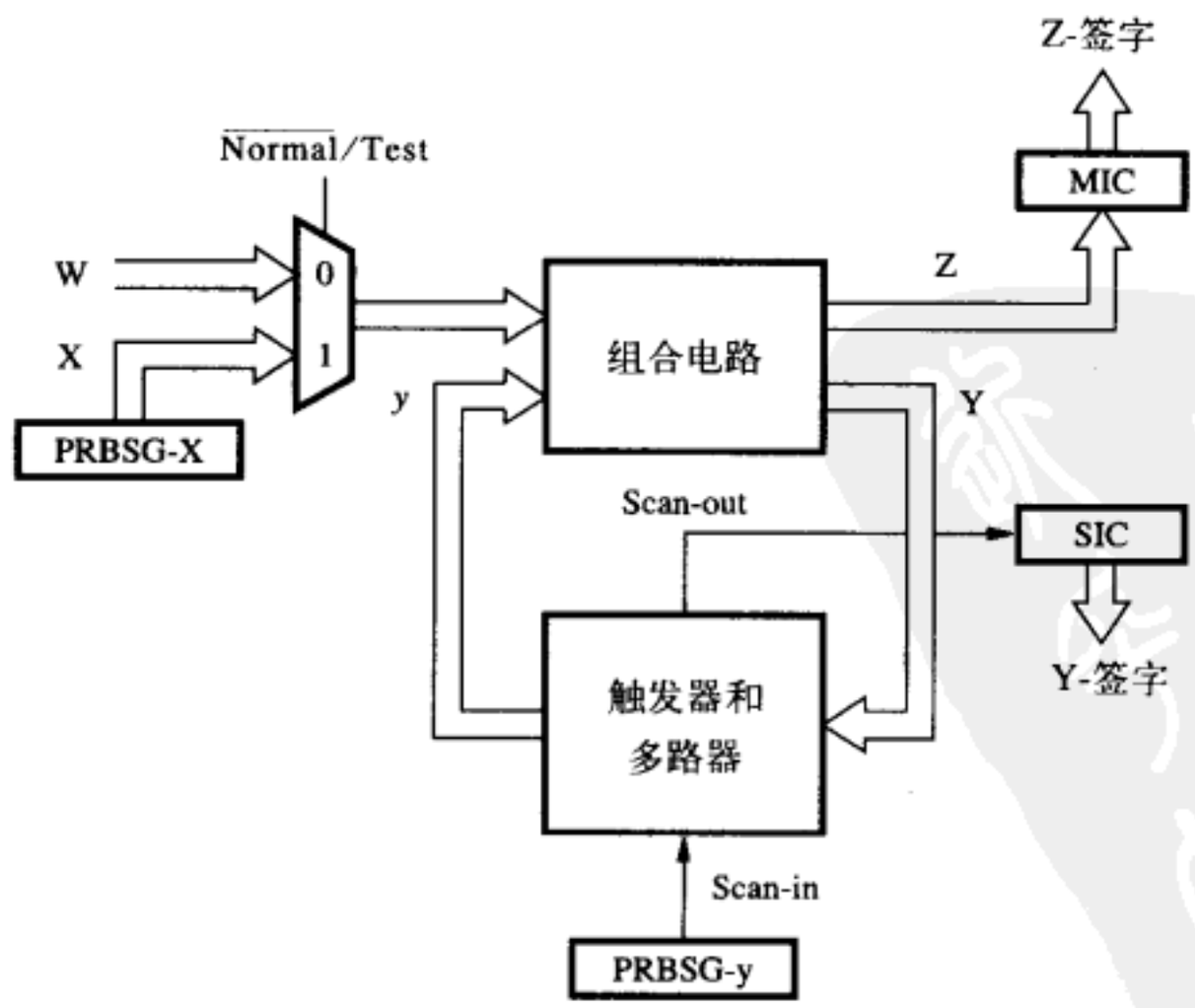


图11-17 时序电路中的内建自测试电路

测试输出使用两个压缩电路进行压缩。原始输出码$Z = z_1z_2\cdots z_m$用MIC电路压缩，而那些用于下一个状态的连线上的向量$Y = y_1y_2\cdots y_k$，用SIC电路压缩。这两个电路相应生成Z-签字和Y-签字。测试过程和例11.3一样，只是测试结果与正确电路应有的响应只做一次比较，而且测试过程结束时两个签字要和所保存的模式相比较。图11-17并没有将存储模式的电路和比较电路画出来。我们可以不将存储应有结果的签字模式作为设计电路的一部分，而是可以将MIC和SIC移位寄存器的内容移出到两个输出的引脚上，在外部和期望的签字进行必要的比较。请注意，与前面提到的将扫描出的每个测试向量的结果与预期的结果进行比较的方法相比，使用这种签字测试的方法可以显著地减少测试时间。

内建自测试方法的效率取决于线性反馈移位寄存器（LFSR）生成器和压缩器（也用作比较器）电路的长度。移位寄存器越长效果越好[13]。测试时检查不到电路故障的一个原因是伪随机生成的测试并不能完全覆盖所有可能的故障。另一个原因是故障电路的输出所生成的签字正巧和正确电路生成的签字一样。这也是有可能发生的，因为压缩过程可能导致一些信息的丢失，比如，两个完全不同的输出码压缩后可能产生同一个签字。这就是所谓的*签字混淆*（aliasing）问题。

11.7.1 内建逻辑块观察器

内建自测试（BIST）的本质是芯片内部就具有生成测试向量和对结果进行比较的能力。不需要用两个电路分别完成这两部分功能，而只用一个电路就可以实现这两个目的。图11-18描述了这样的电路结构。我们把这个电路叫做*内建逻辑块观察器*（BILBO）[14]。这个4位电路和图11-14具有一样的反馈连接。

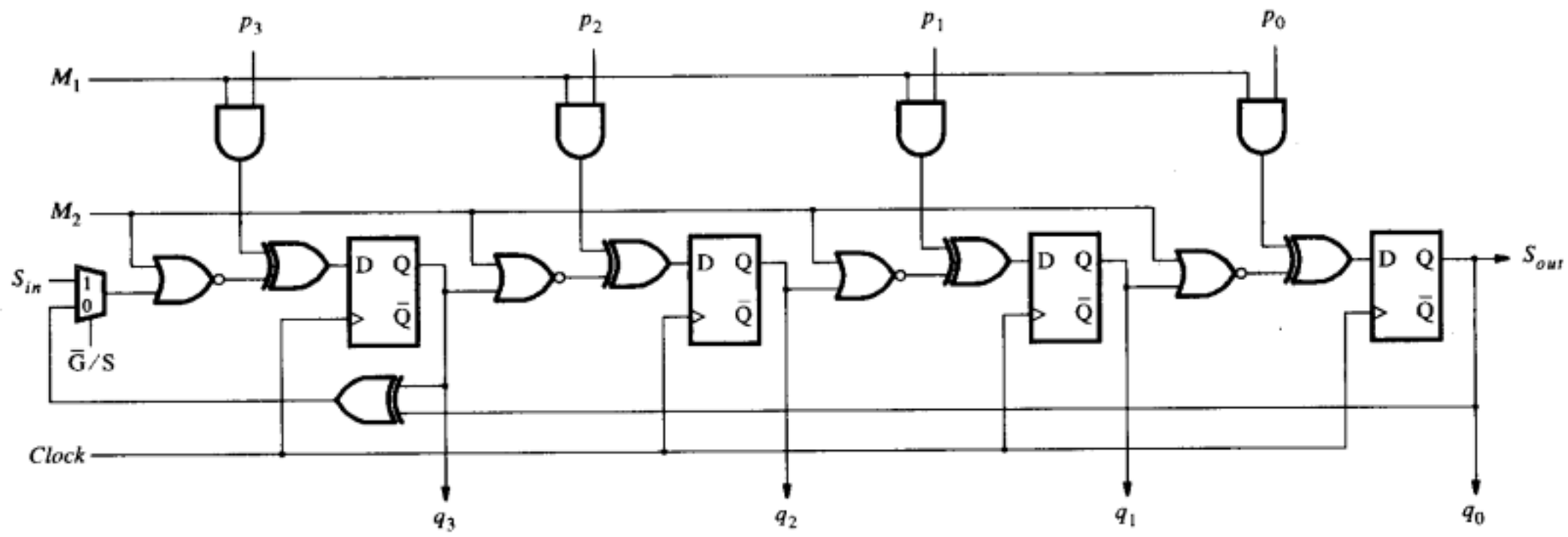


图11-18 四位内建逻辑块观察器（BILBO）

内建逻辑块观察器（BILBO）有四种操作模式，可以由模式位M_1和M_2来控制。这四种模式描述如下：

- $M_1M_2 = 11$为正常系统模式，所有触发器分别受到输入信号p_0到p_3的控制。在此状态下，每一个触发器都可能用于实现有限状态机的状态变量，把p_0到p_3当作y_0到y_3。
- $M_1M_2 = 00$为移位寄存器模式，当输入控制信号$\overline{G}/S = 1$时，允许测试向量逐位扫描移入，并把测试结果逐位扫描输出。当$\overline{G}/S = 0$时，电路成为二进制伪随机序列发生器。
- $M_1M_2 = 10$为签字模式，施加在输入端p_0到p_3上的模式序列被压缩成q_0到q_3用作为模式的签字。
- $M_1M_2 = 01$，复位模式，所有触发器置0。

使用内建逻辑块观察器（BILBO）电路的有效方法如图11-19所示。组合电路部分可以分

为两个（或者更多的）部分来测试。内建逻辑块观察器电路既可以用来提供部分电路的输入，也可以接受其他部分的输出。测试过程包含两个阶段：1）将内建逻辑块观察器(BILBO)1用作随机二进制序列生成器，提供组合网络1 (CN1)部分的测试向量。在这个阶段内，BILBO2作为压缩器（即比较器）并生成测试的签字。通过将BILBO2设置为移位寄存器模式，可以将签字逐位移出。2）BILBO1和BILBO2的角色对换，测试CN2，重复刚才的过程。

测试过程的详尽步骤如下：

1. 把测试向量逐位扫描输入到BILBO1，复位 BILBO2的所有触发器。

2. 在给定的时钟周期中将BILBO1用作伪随机序列信号发生器，用BILBO2产生签字分析结果。

3. 把BILBO2中的内容逐位扫描输出，在外面比较签字分析；然后逐位向BILBO2扫描输入CN2的原始测试向量；把BILBO1 中的所有触发器复位。

4. 在给定的时钟周期中将BILBO2用作伪随机序列信号发生器，用BILBO1产生签字分析结果。

5. 把BILBO1中的内容逐位扫描输出，在外面与期望的输出比较，做签字分析。

用上述方法可以将BILBO电路用作测试目的。在其他场合，BILBO电路用作正常的系统工作模式。

11.7.2 签字分析

前面我们已介绍了如何使用签字分析实现有效的内建自测试机制。将一个长序列的测试结果压缩成一个签字的主要思想源于惠普公司在上世纪70年代生产的一种设备，该设备叫签字分析仪[15]，以后在此基础上逐渐发展成熟。签字分析法的名字正好是指这样一种测试方法，即用签字来代表测试的结果。

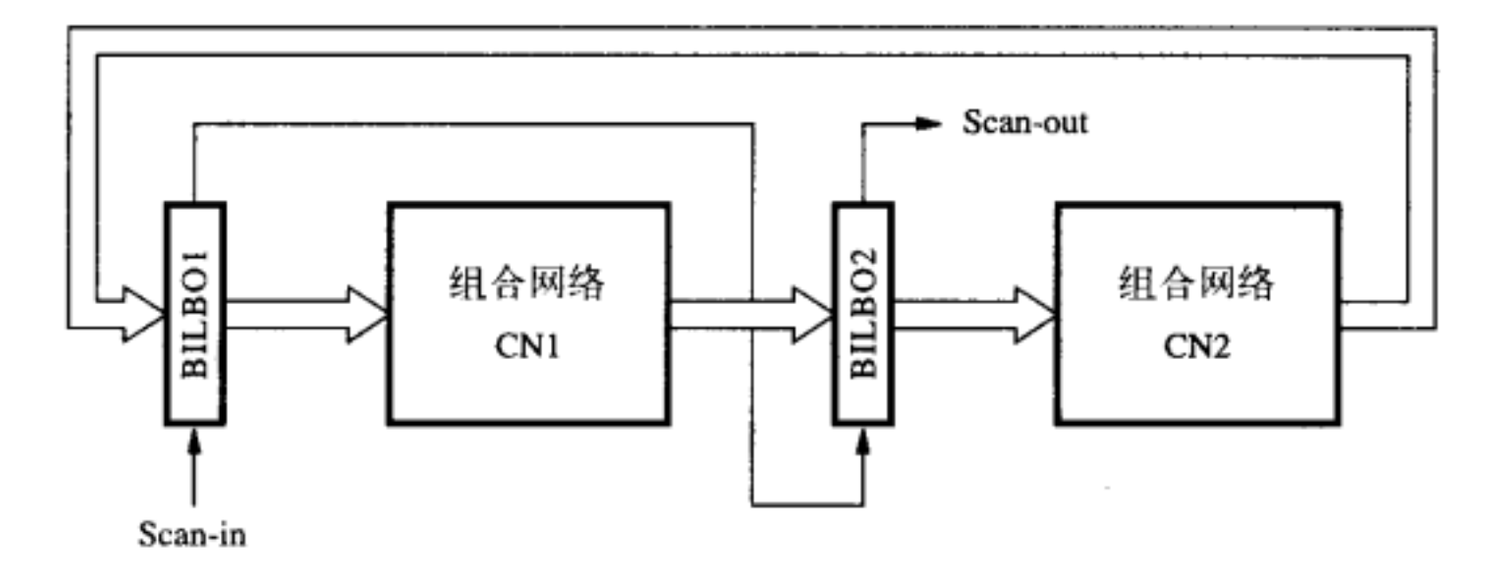


图11-19 测试用内建逻辑块观察器（BILBO）

签字分析法对于数字系统尤其合适。因为数字系统天生就具有生成人们想要的测试模式的能力。有一个计算机系统的例子可以说明，其各部分都工作在由软件控制下生成的测试模式的激励下。

11.7.3 边界扫描

前几节讨论的测试技术既适用于单个芯片上的电路，也适用于印刷电路板上实现的电路。只要能在电路上施加测试向量，又可以观察其输出，该电路就是可测试的。这涉及电路原始输入和输出的直接访问。

芯片被焊到电路板上后，通常就不可能用探针接触引脚来进行测试了。这样就影响了测试过程，除非提供一些间接连接的引脚。扫描路径的概念可以扩展到电路板级电路。假设芯片的每个原始输入输出引脚通过D触发器相连，而且处在测试状态时，所有的触发器都连接成

一个移位寄存器。然后用两个引脚作为串行序列的输入输出端，测试向量就可以用移位寄存器逐位扫描进入或输出触发器。测试时，将一个芯片的串行输出引脚连接到另一个芯片的序列输入引脚上，使得所有芯片的所有引脚连成一个板级移位寄存器。这个方法在实际应用中已经得到普及，目前已形成IEEE 1149.1标准[16]。

11.8 印制电路板

本书中介绍的各种设计和测试技术都可以用于任何逻辑电路的设计和测试，无论该电路是用单个芯片实现的，还是由安置在印制电路板上的几块芯片来实现。本节将讨论几个实际问题。这些实际问题是在把一个或多个电路模块连接成大型数字系统在印制电路板上实现时产生的。

典型的印制电路板（PCB）包含多层连线。PCB在生产时，每层的连线版图已经生成。各层之间由绝缘材料分隔，并像三明治那样压在一起形成电路板。各层连线之间的连接通过专为此目的而设置的通孔完成。芯片和其他部件既可以焊在PCB的上面，也可以焊在PCB下面。

在前面的几章中，我们已经非常详细地讨论了如何用CAD工具来设计用单个芯片（比如PLD）实现的数字电路。如果电路需要用多芯片来实现，我们需要另外一种设计PCB的CAD工具，这个工具与单芯片CAD工具是不同的。PCB的CAD工具可以把多个芯片连接起来实现完整的数字系统。很多公司都提供这种PCB的CAD设计工具，比如Cadence 和明导（Mentor Graphics）公司。PCB的CAD设计工具能自动地确定在PCB上各个芯片的安放位置，但是设计者也可以手动确定某个芯片在PCB上的位置，这就叫做*布局*过程。若芯片和其他元件（如接线端和电容器）的布局已经确定，PCB的CAD设计工具可以自动生成电路板每层的走线版图(layout)。这个过程叫做*布线*（routing）。同样，设计者也可以干预布线，手动为某些接点布线。但是，因为连接线有成千上万条，所以自动生成布线的解决方案是至关重要的。

除了前面几章讨论过的设计问题外，由PCB实现的大型电路还受制于某些其他因素。PCB线迹上的信号可能会受噪声问题的影响，噪声是由串扰（crosstalk）、电源产生的电压毛刺和长线末端的反射所引起的。

串扰

两条紧靠的并列的连线之间存在电容耦合，一根线上的脉冲会引起临近线上相似（通常会小很多）的脉冲。这就叫做*串扰*。我们不希望串扰存在，因为它引起了噪声问题。

画时序图时，我们通常画的是理想的波形，有陡峭的时钟沿，很好地定义了逻辑0和1的电压。在实际电路中，相应的信号可能和理想的情况相差很多。正如3.8.4节所解释的那样，电路中的噪声会影响电平，从而造成电路出现麻烦。例如，在某一时间点，噪声使一个本应为逻辑1的信号衰减到其下一个门认为其为逻辑0，这样电路中的误操作就可能发生了。因为噪声的影响是随机的，我们经常很难检测到它们。

为了减小串扰，应该小心地避免紧邻平行长线的出现。因为PCB大小有限，加之连接线又很多，这往往很难做到。多层布线可以减轻一些串扰问题。

电源噪声

当CMOS电路的状态发生改变时，电路中会出现瞬间电流，在电源（V_{DD}和地）连线上表现为一个电流脉冲。因为PCB上的线迹铜箔有一个小的电导，这样的电流脉冲会引起连线上的一个电压毛刺（短脉冲）。这样的电压毛刺积累产生的效应可能引起电路的误操作。

在芯片附近的V_{DD}和地线间连接一个小电容，可以显著减小由CMOS开关瞬间电流感应而引起的电压毛刺。因为这些毛刺具有很高的频率，芯片附近连接的小电容就能直接把它们短

路了，因此这些电压毛刺便可从电压源线路中旁路掉。这样的电容叫做旁路电容，不会对电源的直流电压产生不利的影响。

大规模芯片，例如PLD，经常需要几个V_{DD}和地线连接。遇到这种情况，我们建议为每对连接V_{DD}和地的引脚配上旁路电容。举个例子，生产厂商建议，连接PLD电源的每对引脚都要并联0.2μF的电容，该去耦电容离PLD芯片越近越好。

反射现象和终端匹配

时钟频率低时，PCB上的走线行为和电路中简单连线的行为一样。但是，在高时钟频率时，我们就必须考虑传输线效应了。信号沿长线传播时会因连接线的内阻而减弱，因串扰而出现噪声，到达连线的终端时还有可能产生反射。反射若未能在下一个有效时钟沿到来前消失，就会引起问题。传输线效应不属于本书的讨论范围。我们只谈谈可以通过在传输线上布置“终端匹配”部件而得以预防信号的反射。终端匹配元件可以简单到只是个电阻，该电阻和传输线的表观电阻相匹配，该电阻也称为传输线的特征阻抗。还有一些其他形式的终端匹配元件。为了深入地学习这些方法，读者可以参阅参考资料[17-18]。

11.8.1 印制电路板的测试

生产出来的PCB必须经过彻底充分的测试。生产过程中的瑕疵可能引起一些连接线开路，或者因两个相近连线上的焊点而短路。电路板上可能存在由于设计过程中的疏忽而引入的隐患。最后，板上的某些芯片和部件也可能存在缺陷。

上电

电路板测试的第一步是接通电源。最坏的情况是某个芯片因为短路而烧毁（极端糟糕的情况下，芯片的封装可能被烧裂）。假设问题不是这样，我们有必要检查是不是有某个芯片变得很烫手。过热是问题严重的征兆，必须加以纠正。

我们必须检查每块芯片的电源和地的连接是否都正确，而且电平是否和指定的大小相符。

复位

电路板测试的第二步就是将PCB上所有的电路复位到预定的起始点。这是指将所有的触发器复位，通常，在一条所有触发器共用的复位线上施加电平就可以实现。核实起始状态是否正确是很重要的。

低层功能测试

因为实际电路可以极其复杂，所以明智的做法是先测试基本的功能。关键的测试是首先证实控制信号能正确地运行。

使用分而治之的方法，先测试简单的功能，再测试复杂一些的。

全面的功能测试

较小的子电路模块的运行验证完毕之后，有必要测试PCB上整个系统的功能。错误的数量取决于设计过程中仿真的彻底程度。总体上说来，对大型数字系统做非常全面的仿真比较难，所以电路板上很有可能发现某些错误。常见的错误由以下原因引起：

- 生产过程中的错误，例如走线错误、部件损坏或者电源电压接错。
- 技术指标说明的错误。
- 设计者误解了介绍芯片功能和使用方法说明书上的资料。
- 生产商提供的技术说明书资料存在错误。

如前面提到过的，PCB包含多层连线，每层内都有几千条连线。找到PCB的故障和进行维修是一件既困难又费时的任务，尤其当错误发生在电路板的内线层（相对于顶层和底层）时更是如此。

时序

接下来我们要验证电路的时序。好的策略是刚开始调试时先使用较慢的时钟。如果电路工作正常，再逐渐增加时钟频率，直到达到需要的工作频率。

时序问题是由于电路中通过不同的传播路径的延迟而引起的。这些延迟来自于逻辑门和它们之间的连线。关键在于，电路中所有触发器的所有数据输入端在时钟信号的有效沿到来前必须确保进入稳定，必须满足触发器数据建立时间的要求。

可靠性

我们希望一个数字系统能可靠地工作很长时间。它的可靠性可能受几个因素影响，比如时序、噪声和串扰等问题。

信号的时序必须提供一些安全容限，以允许传播延迟中一些小的变化。如果时序安排太紧凑，电路也许能正确运行一段时间，但是最终还是会因为小的时序错误而出现故障。芯片的时序会随温度而发生变化，所以如果不坚持考虑温度约束，电路就会出现故障。用风扇通常可以解决电路的冷却通风问题。

11.8.2 测试仪器

测试PCB上的电路需要一些特别的仪器。

示波器

使用示波器可以检查单个信号的细节。这个仪器显示出信号的电压波形，可以展示与传播延迟和噪声相关的那些潜在的问题。在示波器上展示的波形显示了信号的实际电压值。它不能描绘出简化的具有完美边沿的方波的理想波形。如果使用者只想要看信号的逻辑值是0还是1，可以使用另一种叫做逻辑分析仪的仪器。

逻辑分析仪

示波器只能同步观察到很少几个信号，而逻辑分析仪却允许同时观察数十个或数百个通道的信号。它通过将探针连接到电路的各点上，以这一组点作为其输入。它将信号数字化，并在屏幕上以波形形式显示被测信号。逻辑分析仪有一个强大的功能，它可以将一小段时间内信道上的一系列变化记录到内部存储器中。这样操作人员就可以根据需要显示此信息中的任意一段。典型的逻辑分析仪可以记录下几个毫秒内发生的事件，这在普通的数字信号中已经包括很多个周期了。

在调试阶段，观察电路在测试条件下所产生的正确波形对调试工作帮助不大。关键是要观察故障发生时的波形。逻辑分析仪可用来记录触发事件前后一段时间内有关信号的变化。使用者必须指定触发事件。比如说，假设我们怀疑电路故障是由两个控制信号（A和B）同时发生引起的，而设计说明书上明确规定这两个控制信号为互斥。我们可以在输入为A和B的双输入与门输出端设置一个触发点，当输出为1时，随即触发。找到合适的触发事件很难，要靠使用者的直觉和经验。

想要有效地使用逻辑分析仪，我们要将探针连到电路上一些有用的（对测试有用的）测试点上。所以在设计PCB时提供这些“测试”点就很重要。

11.9 小结

制成的芯片产品必须经过测试以确保期望的功能。本章中讨论的所有技术都和这种测试有关。测试向量的生成和施加测试后电路要求的反应都是建立在电路设计是正确的基础上的。我们要测试的是物理实现的正确性。

在设计过程中还会出现另一个方面的测试问题，即设计者必须确保设计的电路符合设计

技术指标和规范。从测试点的观点来看，没有证明可靠的电路可用来生成我们想要的测试向量。CAD工具可以帮助我们为设计好的电路产生测试向量，但它们并不能确认这个电路是不是真的完成了设计者要达到的功能。设计错误往往使制成的电路芯片在功能上与需求规定的指标有某些不同。

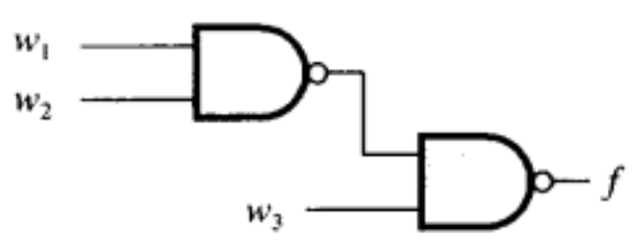
小规模电路可以进行充分的测试以验证它们的功能。组合电路可以通过测试检查它是否按照真值表工作。时序电路可以通过测试检查它是否按照状态表进行转换。如果电路设计时考虑到了测试的话，测试工作就会容易得多，如11.6.1节所示。大规模电路不能进行穷举测试，因为必须施加的测试向量实在太多。在这种场合就需要设计者的智慧了，他们必须要确定一组可管理的测试向量集合，以证明电路的正确性。

练习题

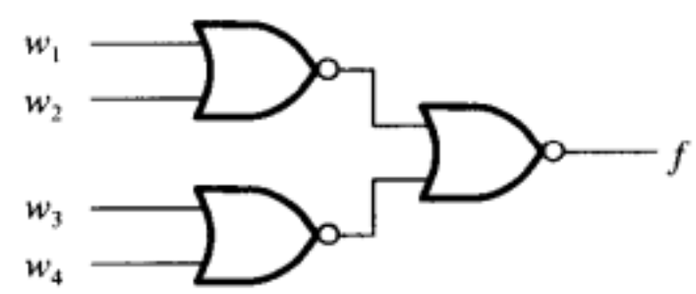
标有*号的习题答案参见书后。

*** 11.1** 设被测电路如图P11-1所示，导出类似于图11-1b所示的表，通过8个可能的测试，表明各种固滞为0和固滞为1的故障覆盖。找到该电路的最小测试集合。

11.2 设被测试电路如图P11-2所示，重复上题同样的工作。



图P11-1 练习题11.1的电路



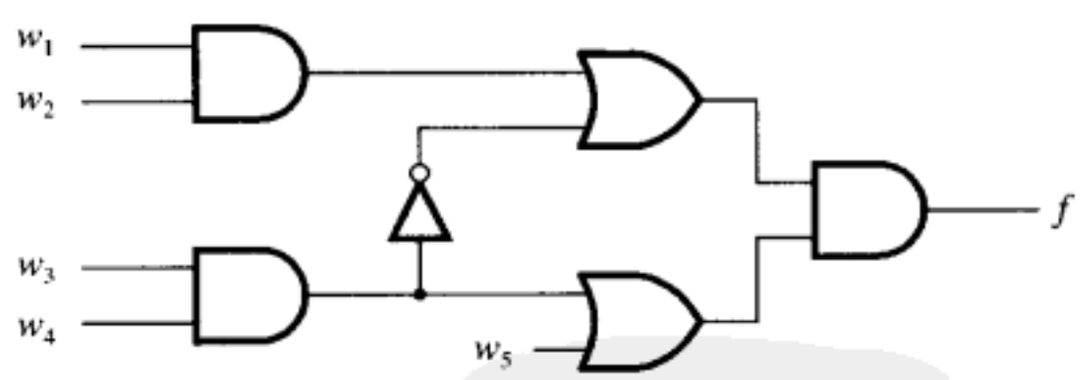
图P11-2 练习题11.2的电路

*** 11.3** 设计一测试来区别实现以下两个表达式的电路。

$$f = x_1x_2x_3 + x_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_4 + \bar{x}_1x_3\bar{x}_4$$

$$g = (\bar{x}_1 + x_2)(x_3 + x_4)$$

11.4 设被测电路如图P11-3所示，敏感化该电路的每条路径，以获得包含最少测试个数的完整测试集合。



图P11-3 练习题11.4的电路

*** 11.5** 设被测试电路如图11-4a所示，写出能检测出以下故障的测试：

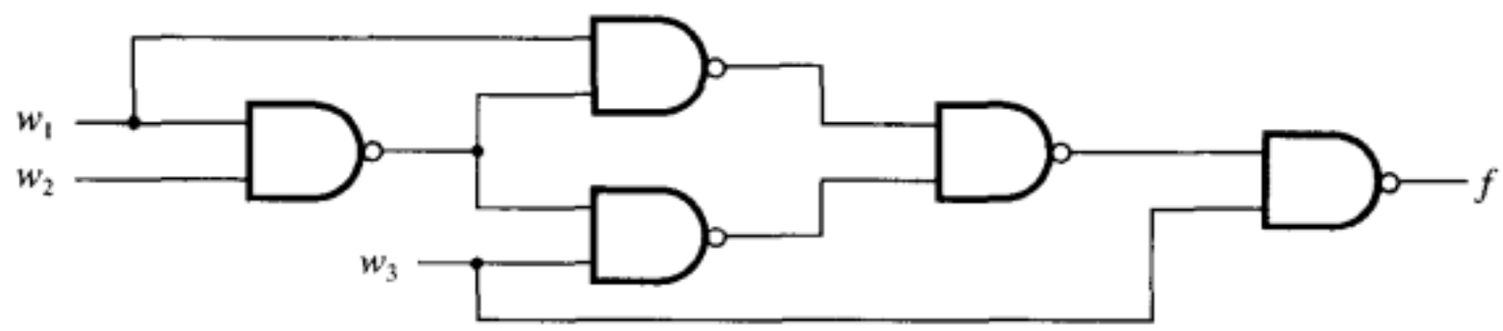
$$w_1/0,\ w_4/1,\ g/0,\ c/1$$

11.6 假设随机地选取5组测试输入信号：$w_1w_2w_3w_4 = 0100, 1010, 0011, 1111$和 0110 来测试图11-3所示的电路，用这些测试得到的单个故障覆盖率是百分之几？

11.7 把电路换成如图11-4a所示的电路,重复上题的工作。

11.8 把电路换成如图11-5所示的电路,重复习题11.6的工作。

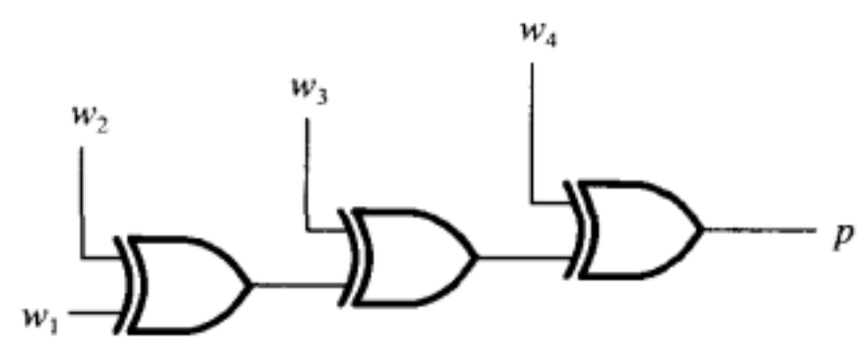
*** 11.9** 设被测试电路如图P11-4 所示，是否本电路中所有单个固滞为0或者固滞为1的故障都是可测试的？如果不能，请解释为什么？



图P11-4 练习题11.9的电路

11.10 证明在所有门的扇出都为1的电路中，任何能检测出所有输入线上单个故障的测试集合必定能检测出整个电路的全部单个故障。

*** 11.11** 图P11-5所示的电路可确定四位数据单元的奇偶校验位。导出能在该电路中检测出所有单个固滞为0和固滞为1的故障的最小测试集合。如果异或门是用图4.26c所示电路实现的，你的测试集是否还能检测出所有故障？你的结果是否能扩展到一般的涉及n 位的数据单元？



图P11-5 练习题11.11的电路

*** 11.12** 导出能检测出图6-16c所示译码器电路所有单个故障的测试集合。

11.13 列出图11-4a所示电路中能用测试集合（$w_1w_2w_3w_4$ = 1100, 0010和0110）中的每组检测出的所有单个故障。

11.14 敏感化图11-12所示电路中组合电路部分的每条路径，以便得到尽可能小的完整的测试集合。展示如何施加你的测试集合到该电路进行测试，并说明完成必要的测试需要多少个时钟周期？

11.15 画出表示测试图11-12所示电路必须的控制流程的算法状态机图（ASM图）。

11.16 图11-12所示电路为图8-79所示的有限状态机提供了容易的可测试的电路结构。在例11.3中，我们展示了如何通过随机选择的测试向量，对组合电路部分进行测试，来完成该电路的测试工作。企图确定该电路是否真的实现了图8-79b所示的由功能指定的状态表也许是另一个不同的测试解决方案。这个方案可以借助于使电路经历状态表指定的所有转移过程来完成。例如，在施加$Resetn = 0$ 信号后，电路进入状态A。我们必须通过扫描输出期望的$y_2y_1 = 00$，来验证该电路是否真的已经被强制进入状态A。以后的每次状态转移都必须检查。为了验证若$w = 0$，则A的状态保持($A \rightarrow A$)，我们有必要使输入$w = 0$，并通过使 $\overline{Normal} / Scan = 0$，允许进行一个周期的正常操作。必须观察输出$z$的值。在观察到扫描输出$y_2y_1$是否等于00后，才能观察到输出$z$的值。同时，下一个测试向量必须扫描进入。如果本测试牵涉到验证若$w = 0$，状态是否从$B \rightarrow A$，则需要把$y_2y_1 = 01$的测试向量扫描进去。这个过程继续进行下去，直到所有的状态转移过程都得到验证。

必须在表中填入在该电路完整测试期间每个时钟周期的 $\overline{Normal} / Scan$，*Scan-in*, *Scan-out*, w, 和z的信号值，还有已测试过的状态转移，以确保测试的完备。

11.17 编写描述如图11-12所示电路的Verilog代码。

11.18 画出ASM（算法状态机）图，该图描述了测试某数字系统必须的控制信号，该数字系统的测试电路使用了如图11-18和图11-19所示的BILBO（内建逻辑块观察器）结构。

参考文献

1. A. Miczo, *Digital Logic Testing and Simulation* (Wiley: New York, 1986).
2. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. F. H. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).

4. Y. M. El Ziq, "Automatic Test Generation for Stuck-Open Faults in CMOS VLSI," Proc. 18th Design Automation Conf., 1981, pp. 347–54.
5. D. Baschiera and B. Courtois, "Testing CMOS: A Challenge," *VLSI Design*, October 1984, pp. 58–62.
6. P. S. Moritz and L. M. Thorsen, "CMOS Circuit Testability," *IEEE Journal of Solid State Circuits* SC-21 (April 1986), pp. 306–9.
7. J. P. Roth et al., "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Computers* EC-16, no. 5, (October 1967), pp. 567–80.
8. J. Abraham and V. K. Agarwal, "Test Generation for Digital Systems," in D. K. Pradhan, *Fault-Tolerant Computing*, vol. 1, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. T. W. Williams and K. P. Parker, "Design for Testability—a Survey," *IEEE Transactions on Computers* C-31 (January 1982), pp. 2–15.
10. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
11. W. W. Peterson and E. J. Weldon Jr., *Error-Correcting Codes*, 2nd ed. (MIT Press: Boston, MA, 1972).
12. J. E. Smith, "Measures of Effectiveness of Fault Signature Analysis," *IEEE Transactions on Computers* C-29, no. 7 (June 1980), pp. 510–4.
13. R. David, "Testing by Feedback Shift Register," *IEEE Transactions on Computers* C-29, no. 7 (July 1980), pp. 668–73.
14. B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Techniques," Proceedings 1977 Test Conference, IEEE Pub. 79CH1609-9C, October 1979, pp. 37–41.
15. A. Y. Chan, "Easy-to-Use Signature Analyzer Accurately Troubleshoots Complex Logic Circuits," *Hewlett-Packard Journal*, May 1997, pp. 9–14.
16. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.
17. *High-Speed Board Designs*, Application Note 75, Altera Corporation, January 1998.
18. L. Y. Levesque, "High-Speed Interconnection Techniques," Technical Report, Texas Instruments Inc., 1994.

第12章　计算机辅助设计工具

我们曾在2.9节介绍过CAD工具，并且还在其他章节里简明地讨论了这些CAD工具。本章中工具（tool）这个词是指允许用户完成特定任务的软件程序。在本章中，我们将通过一个小的设计范例，展示在CAD设计流程的不同阶段是如何对设计进行处理和优化，从而更加深入地描述典型CAD系统中的某几个工具。

12.1　综合

图12-1（拷贝自图2-29）给出了CAD系统的概要。通常先把想要设计的电路用硬件描述语言（如 Verilog HDL）加以描述。接着用CAD流程中的综合工具对 Verilog 代码进行处理。综合是从用户的技术指标产生逻辑电路的过程。图12-2展示了综合处理过程中三个典型的阶段。

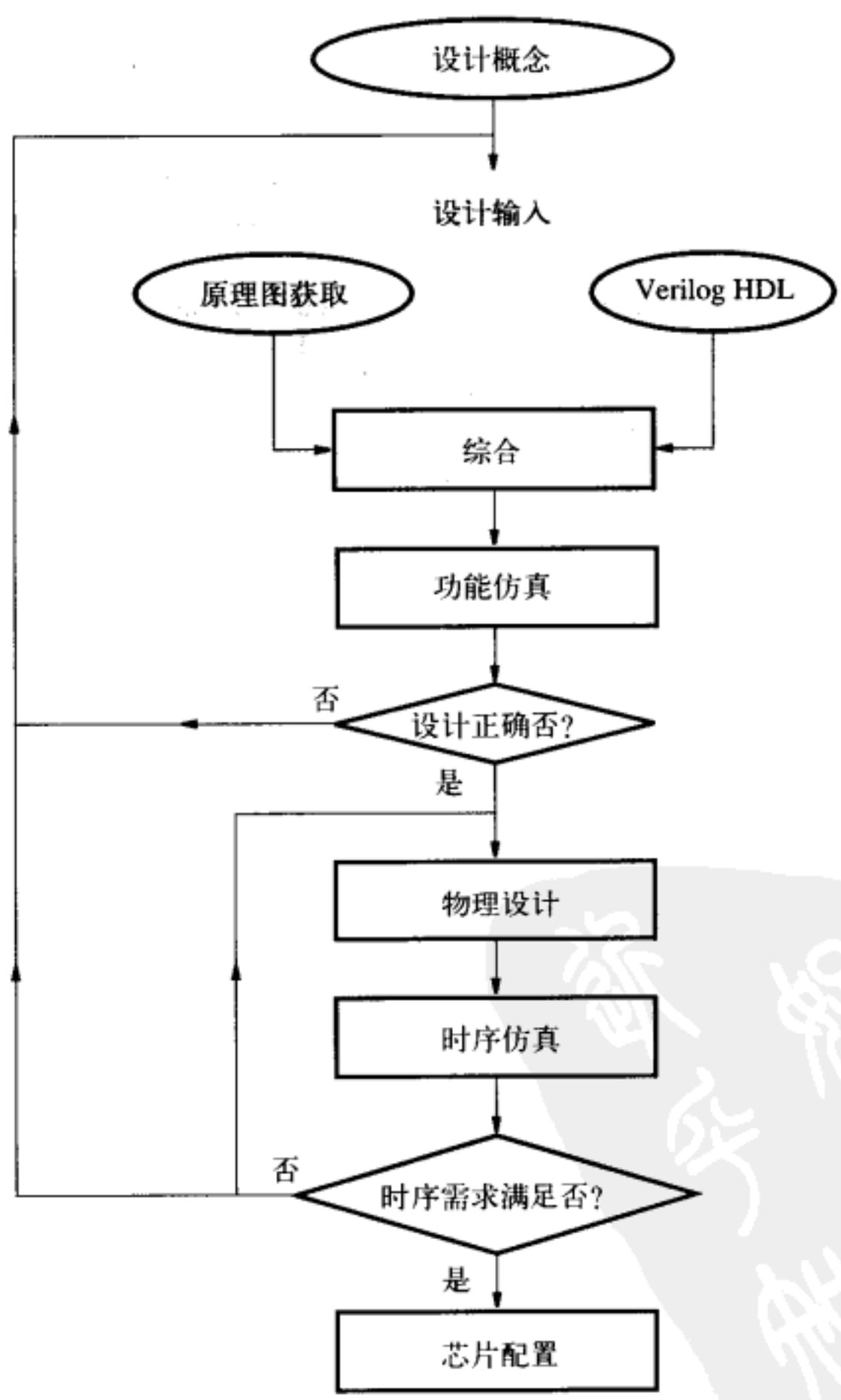


图12-1　典型的CAD系统

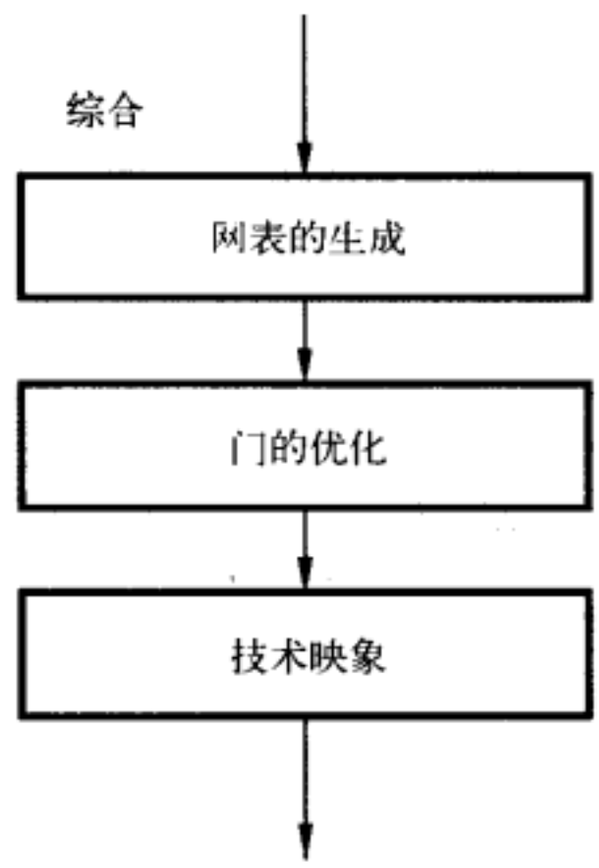


图12-2 综合工具中的步骤

12.1.1 网表的生成

网表（netlist）生成阶段对代码的语法进行检查，报告任何发现的错误，例如，信号名没有定义、括号忘了、关键字错误等。一旦所有的错误均被改正后，便根据Verilog 代码的语义生成电路网表。电路网表就是用逻辑表达式描述电路，其中还包括了组成该电路的一些元件，如加法器、触发器和有限状态机。

12.1.2 门的优化

第二个阶段是进行门的优化，这种优化所进行的就是第4章所描述的逻辑处理，优化阶段对网表进行一系列处理，以得到就优化目标而言更好的等价电路。正如我们曾经在2.9.2节讲述过的那样，如何判断一种电路比另外一种电路更好一些，可依据不同的准则：电路的价格、运行的速度，或者两者的结合。

以前面讨论过的由综合阶段所产生的结果为例，考虑图12-3所示addersubtractor 模块（加/减法器）的 Verilog 代码，该模块定义了能够对n位数字进行加法或者减法运算，并在一个寄存器中将结果进行累加操作的电路。根据这段代码，综合工具生成了与图12-4对应的网表。输入的两个数，即$A = a_0, \cdots, a_{n-1}$ 和 $B = b_0, \cdots, b_{n-1}$，在进行加法或者减法运算之前，这两个数被放置在$Areg$和$Breg$两个寄存器中。若A 和B 是从外部异步地输入到寄存器中，则这两个寄存器与电路的运算操作可以同步进行。控制输入信号Sel 决定了运算操作的模式。若$Sel = 0$，则A被选作加法器的输入；若$Sel = 1$，则结果寄存器 $Zreg$ 被选中。执行的操作是加法运算还是减法运算由控制输入信号 $AddSub$ 确定。图12-4 中寄存 $A, B, Sel, AddSub$ 和 $Overflow$ 信号的触发器由图12-3a最下面的几句代码得到。多路选择器由mux2to1模块生成，加法器由图12-3b 的adderk 模块生成。被连接到寄存器B的异或门，以及产生$Overflow$（溢出）输出信号的异或（XOR）函数，由addersubtractor 模块的最后几句代码生成。

```
// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
   parameter n = 16;
   input [n-1:0] A, B;
   input Clock, Reset, Sel, AddSub;
```

图12-3 累加器电路的Verilog代码

```
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, H, M, Z;
    wire carryout, over_flow;

// Define combinational logic circuit
    assign H = Breg ^ {n{AddSubR}};
    mux2to1 multiplexer (Areg, Z, SelR, G);
        defparam multiplexer.k = n;
    adderk nbit_adder (AddSubR, G, H, M, carryout);
        defparam nbit_adder.k = n;
    assign over_flow = carryout ^ G[n-1] ^ H[n-1] ^ M[n-1];
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
        begin
            Areg <= 0; Breg <= 0; Zreg <= 0;
            SelR <= 0; AddSubR <= 0; Overflow <= 0;
        end
        else
        begin
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
        end
endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Selm, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Selm;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Selm)
        if (Selm == 0) F = V;
        else F = W;
endmodule

// k-bit adder
module adderk (carryin, X, Y, S, carryout);
    parameter k = 8;
    input [k-1:0] X, Y;
    input carryin;
    output [k-1:0] S;
    output carryout;
    reg [k-1:0] S;
    reg carryout;

    always @(X or Y or carryin)
        {carryout, S} = X + Y + carryin;
endmodule
```

图12-3 （续）

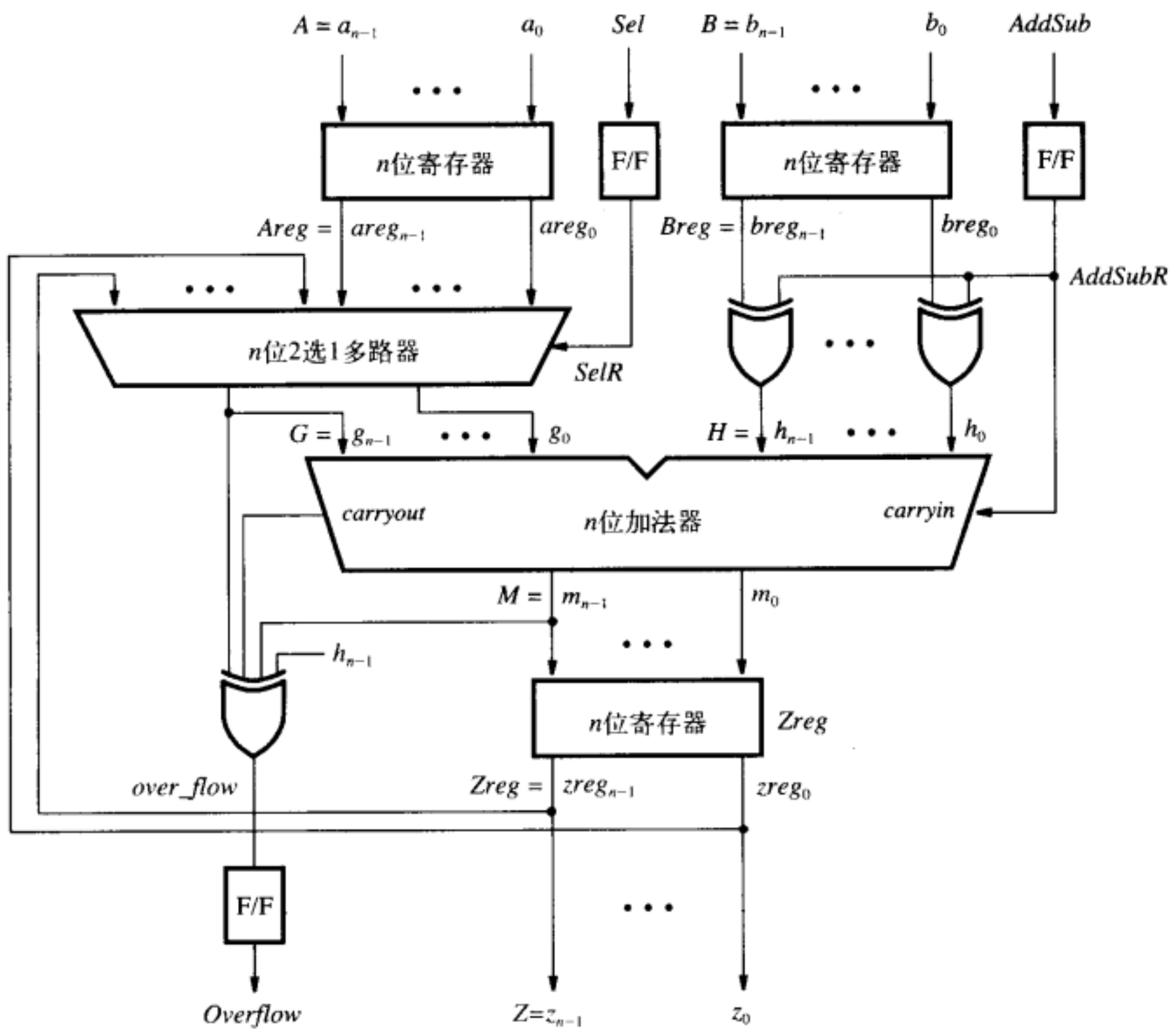


图12-4 与图12-3代码对应的电路

12.1.3 技术映象

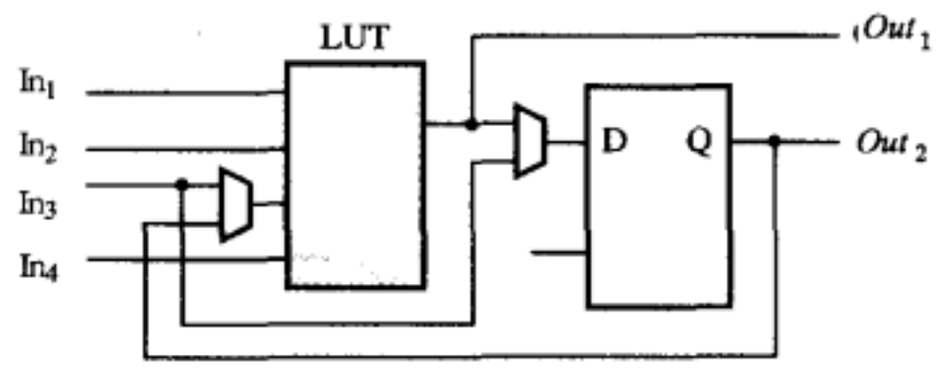
综合过程的最后阶段是技术映象（Mapping）。这个阶段决定如何用目标芯片中的可用资源来实现网表中的每个元件。为了看到技术映象的结果，假定我们已经选择用FPGA来实现这个例子中的电路。在3.6.5节中，我们曾经展示过FPGA中包含二维逻辑块阵列。图3-38给出了包含一个三输入查找表（LUT）和一个触发器的简单逻辑块的图。该块有一个输出，设计者可以选择究竟从查找表输出还是从触发器输出。

图12-5a画出了一个更灵活的逻辑块。该逻辑块包含一个四输入查找表和一个触发器，具有两个输出；该逻辑块提供了一个多路选择器，可以从查找表输出或者将输入*In*3直接加载到触发器。另外一个多路选择器允许将触发器储存的值反馈到查找表的一个输入。可以用多种方法或者模式来使用该逻辑块。最简单的办法是在查找表中直接实现最多为四输入的逻辑函数，并将函数的值储存在触发器中，查找表和触发器都可以为该逻辑块提供输出。图12-5的b到e说明了使用该逻辑块的四种模式。b和c说明只使用查找表或者只使用触发器提供输出，不能同时使用两者。d说明逻辑块的输出只由查找表提供，而查找表的一个输入被连接到触发器的输出。

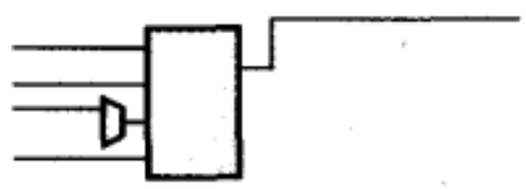
在第5章中，我们曾经说过FPGA中通常包含专用的高速加法器电路。图12-5e展示了这种电路的一种实现方法。查找表被分成两半使用，其中一半产生查找表三个输入的和（sum）函数，另外一半产生这三个输入的进位函数（请返回到3.6.5节，回顾用两个三输入查找表构建

一个四输入查找表的内容）。和函数能提供该逻辑块的输出，或者把输出储存在触发器中，而进位函数输出的是全加器的进位信号。

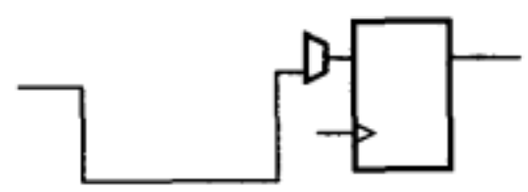
该进位输出直接连接到相邻的逻辑块，作为相邻块的进位输入。相邻块又生成下一级的进位输出，再作为下一个逻辑块的进位输入，一直继续下去。用这种方法，相邻逻辑块之间的直接连接可以用来组成快速进位链。



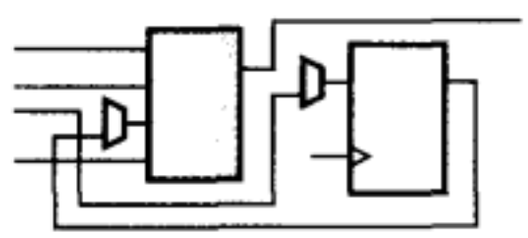
a) FPGA中的一个逻辑单元



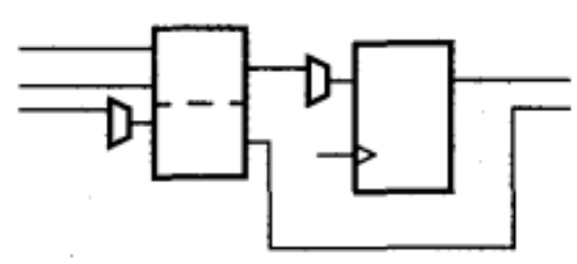
b) 组合模式



c) 同步模式



d) 同步反馈模式



e) 算术（同步）模式

图12-5 FPGA逻辑块的不同模式

图12-6展示了经过技术映象的部分电路，其对应的网表来自于图12-4。每个逻辑块用单独标出，并在图的左下角做了一个标记，标明所用的是图12-5的哪一种模式。该图展示了来自于图12-4的 h_0位，它是由模式（d）的逻辑块产生的。该逻辑块用触发器储存原始输入b_0的值，然后在它的查找表中实现异或（XOR）函数，该逻辑运算需要进行减法操作以求得数B的补数。异或（XOR）门的另一个输入是由模式（c）的逻辑块提供的，*AddSub*输入值储存在该逻辑块的触发器中。该触发器还驱动15个其他实现 h_1, …, h_{15}信号的逻辑块，但图12-6中并没有画出这些块。

这个*AddSub* 触发器被连接到加法器的第一个逻辑块的进位端。该逻辑块用模式(e)来产生求和（sum）以及进位输出。求得的和（sum）被储存在一个输出为 z_0 的触发器中，而进位信号被连接到加法器的下一级。图12-6展示了如下形式的进位函数：

$$c_1 = (\overline{c_0 \oplus h_0}) \cdot h_0 + (c_0 \oplus h_0) \cdot g_0$$

上面这个表达式在功能上等价于第5章所用的那个表达式（其形式为$c_1 = c_0h_0 + c_0g_0 + h_0g_0$），但上面的表达式与FPGA中实际进位链的连接非常接近。图12-6加法器中的最后一个逻辑块并不需要触发器，因为求和（sum）输出信号必须被直接连接到产生溢出（*Overflow*）信号的逻辑块。该求和（sum）输出信号不能既由组合逻辑的输出又由寄存器的输出同时提供，所以需要一个独立的模式（c）的逻辑块来产生 z_{15} 输出信号。

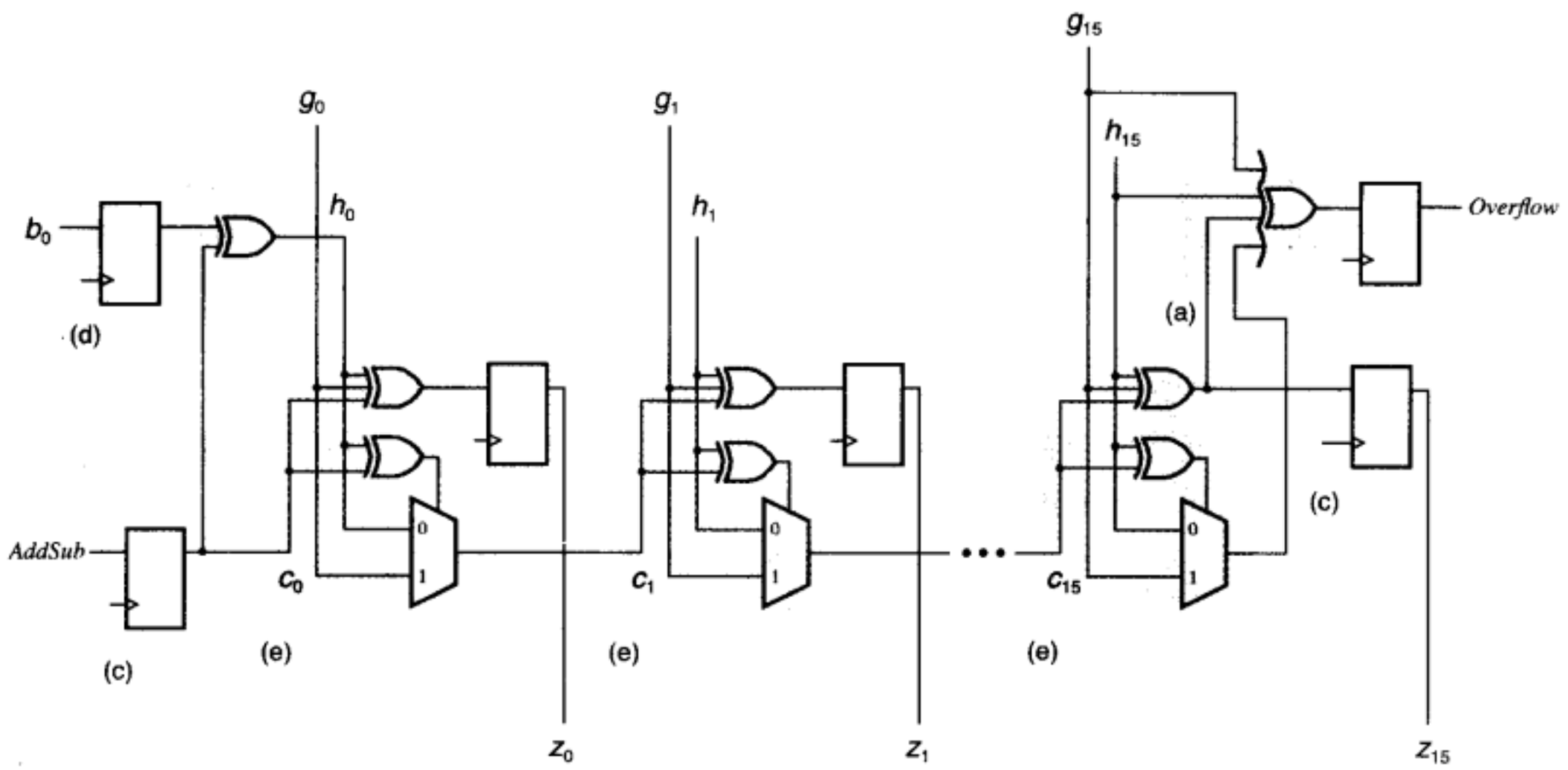


图12-6 图12-4所示电路经过技术映象后的一部分

图12-6所展示的只是由技术映象工具所创建的实现该加法器电路的几种逻辑块。一般情况下，某逻辑功能电路可以用多种不同的技术途径来实现，实现方法虽然不同，但在逻辑上是等价的。读者可以阅读参考资料 [1-3]更深入地了解不同的映象技术。

12.2 物理设计

图12-1中综合阶段下面的步骤是功能仿真和物理设计。正如我们在2.9节中讲述过的，功能仿真涉及对综合后生成的网表施加测试向量，观察并检查生成的网表是否能对输入信号产生正确的输出响应。功能仿真假设生成的电路没有传播延迟，因为检查的目的是想知道基本的逻辑功能是否正确，而不是时序是否正确。用于功能仿真的网表可以是技术映象之前的，也可以是技术映象之后的。附录B中提供了利用本书附带的软件进行功能仿真的例子，我们在这里不再进一步讨论这个问题。

一旦由综合生成的网表被证实其功能是正确无误的，我们便可以着手进行物理设计阶段的工作。该阶段的工作是确定由综合生成的网表如何在目标芯片上实现。图12-7说明了物理设计所涉及的三个步骤：布局、布线和静态时序分析。

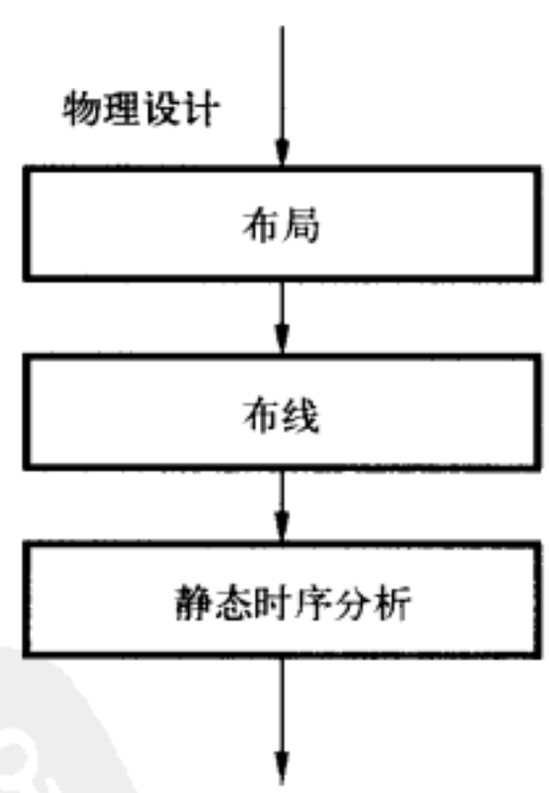


图12-7 物理设计的步骤

12.2.1 布局

布局阶段所做的工作是确定技术映象网表中每个逻辑块在目标器件中的具体位置。图12-8所示的是布局结果举例。该图展示的是FPGA芯片中的一小部分，图中画出了一个逻辑块阵列。白的方块表示尚未占用的逻辑块，而灰色的方块表示实现图12-4所示电路已占用的逻辑块。该电路总共用去53个逻辑块（包括那些在图12-6中显示的块）。图12-8还显示了该电路一些原始输入端的位置，其输入引脚分布在芯片的四周。

为了找到好的布局解决方案，必须对每个逻辑块放在不同的地方进行考虑。对于包含成千上万个逻辑块的大规模电路，这个问题很难解决。为了体会该问题所涉及的复杂性，可以

考虑一下对于某个给定的电路究竟有多少种不同的布局解决方案。假设某电路由N个逻辑块组成，而某FPGA内部正好有N个逻辑块，若我们想用这个FPGA来实现该电路，究竟有多少种布局方案呢？布局工具对第1个逻辑块有N种可选择的位置，而对第2个逻辑块有$N-1$种可选择的位置；对第3个逻辑块有$N-2$种可选择的位置，以此类推。将这些可能的选择相乘，总共有$(N)(N-1)\cdots(1) = N!$ 种可能的布局解决方案。即使N的值并不很大，$N!$也是一个巨大的数字，这就意味着必须用探索（heuristic）法来找到好的解决方案，由于可能性太多，我们只能考虑解决方案总体中很小的一部分。典型的商业布线工具先用一个初始布局配置，然后以迭代方式将逻辑块移动到不同的位置，每次迭代都计算实现电路的运行速度或者成本，并对该方案进行评价，试图从比较中找到较好的布局方案。一直以来，业界就对布局问题进行着广泛的研究，参考资料[4-7]详细地描述了这个问题。

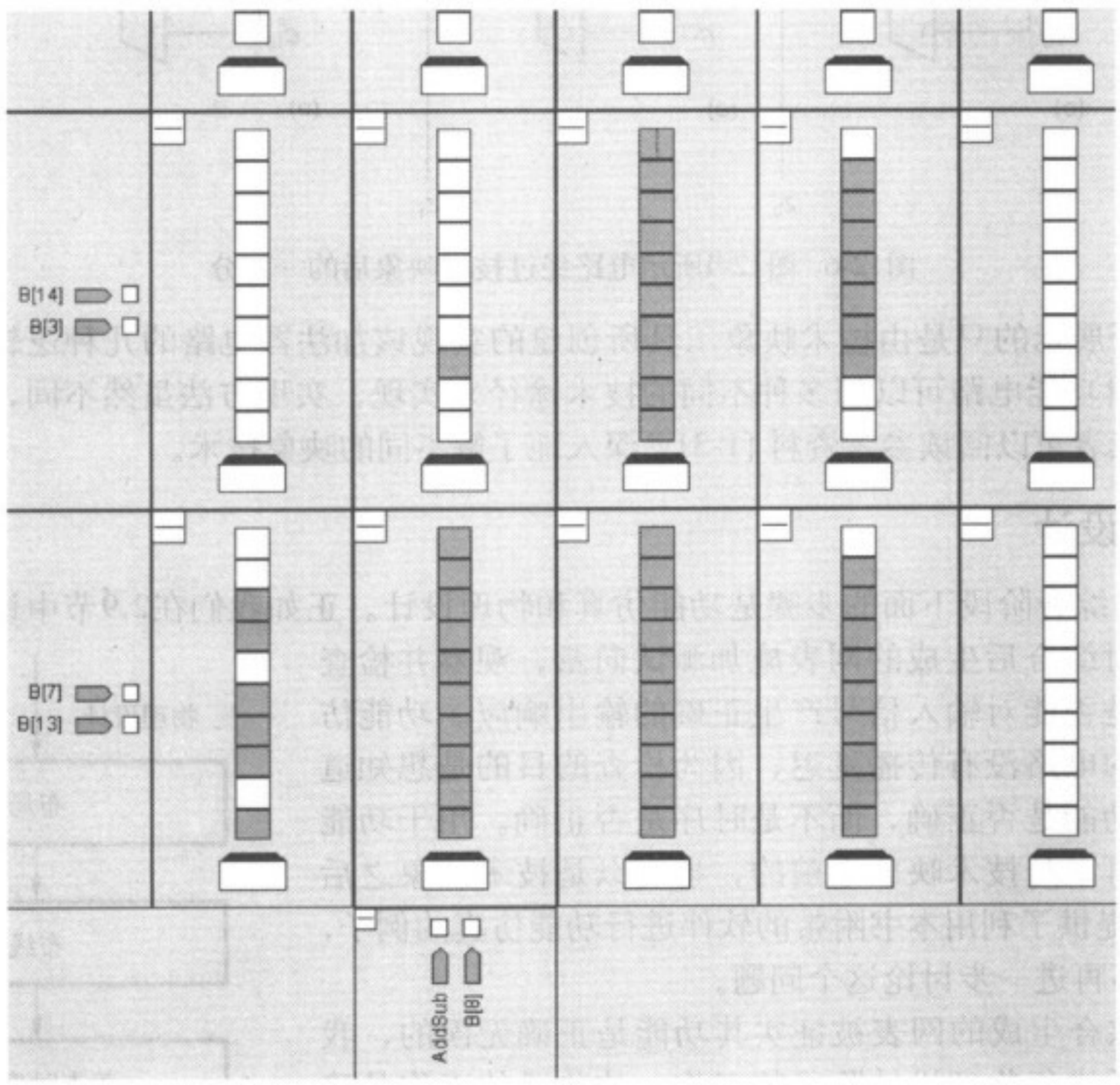


图12-8 图12-6所示电路的布局

12.2.2 布线

电路中的每个逻辑块在芯片中的位置一旦被确定后，布线阶段马上就开始用芯片中现成的连线将这些逻辑块连接起来。图12-9所示的是图12-8所示布局的一种布线解决方案。除了展示逻辑块外，图12-9也显示了该芯片中的某些连线。在图上用灰色的线条表示用来实现电路的线网。该图不但画出了单条长度不等的线网，而且也画出了用灰色小方块表示的扎成捆的线网。布线的CAD工具总是试图最有效地利用不同种类的线网，如有效地连接进位链。图12-9展示了来自于图12-6的进位路径的一个例子。黑色的线网表示进位链将该加法器的不同级连接起来（结束点为溢出（*Overflow*）寄存器）的线网。在参考资料 [3]、[5-6] 和[8]中，我们可以找到对于布线工具更深入的讨论。

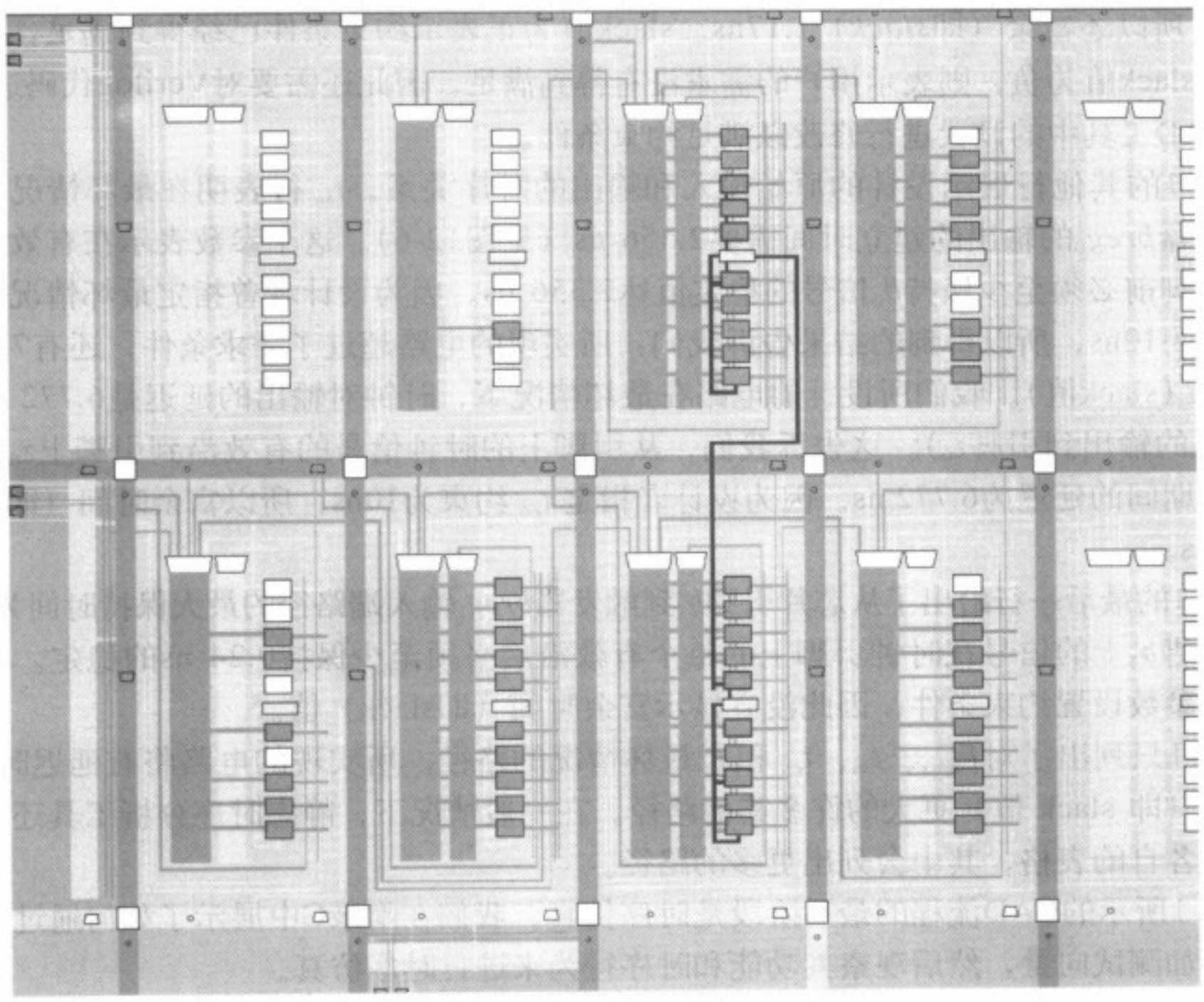


图12-9 图12-8所示布局的布线

12.2.3 静态时序分析

布线完成后，因为CAD系统计算芯片中所有块和线网产生的时序延迟，所以具体电路所产生的时间延迟已经知道了。静态时序分析工具查核该延迟信息，生成一系列的表格，将电路的性能做了量化。表12-1给出了时序分析结果的一个例子，该表列出了四个参数：F_{max}，t_{su}，t_{co}和t_h。F_{max}值指定了该电路时钟的最高频率。该值是由电路中任意两个触发器之间最长的传播延迟（通常叫做关键路径）所确定的。正如10.3节所展示的那样，该路径延迟必须从逻辑块和线网的延迟算起，还必须加上触发器时钟到Q的延迟（t_{cQ}）和建立（t_{su}）参数。在我们的例子中，该关键路径的延迟是$1 / (261.1 \times 10^6) = 3.83$ ns。F_{max}行最后的两列显示该路径起始于*AddSub*触发器，结束于溢出（*Overflow*）触发器，如图12-6所示。

表12-1 静态时序分析结果的总结

参数Parameter	实际值	需求值	富余时间	从	到
F_{max}	261.1 MHz	200 MHz	1.17 ns	*AddSub*	*Overflow*
t_{su}	2.356 ns	10.0 ns	7.644 ns	b_0	$breg_0$
t_{co}	6.772 ns	10.0 ns	3.228 ns	$zreg_0$	z_0
t_h	0.240 ns	N/A	N/A	b_1	$breg_1$

大多数CAD系统允许用户指定所设计电路的时序约束。在表12-1中，我们假设用户已经指定该电路时钟（*clock*）的最高频率必须达到200MHz。由CAD工具获取的需求和实现结果之间的差距称为slack。在该表中，传播延迟必须小于$1/(200 \times 10^6) = 5$ ns，而实际延迟只需要

3.83 ns，所以还富余（即*slack*）1.17ns。slack值为正表示约束条件已经得到满足，还有一些富余。若slack值为负，则表示用户的需求没有得到满足，因此还需要对Verilog代码进行修改，或者对CAD工具中的设置进行修改以满足约束条件。

表12-1的其他行展示设计的原始输入和输出的时序关系。t_{su} 行表明在最坏情况下从引脚 b_0 到触发器$breg_0$的输出的建立时间需要2.356 ns（见图12-6）。这个参数表示在有效时钟沿到达时钟引脚前必须至少保持b_0信号的稳定值达2.356 ns。因为设计者曾指定最坏情况下的建立时间约束为10ns，所以得到的结果告诉我们：所实现的电路超过了需求条件，还有7.644 ns的富余时间（slack值）。我们所设计的电路在最坏情况下，时钟对输出的延迟是6.772 ns（从触发器$zreg_0$的输出到引脚z_0）。这告诉我们，从引脚上的时钟信号的有效沿到引脚上z_0信号发生相应变化期间的延迟为6.772 ns。因为设计者指定 t_{co} 约束为10ns，所以富余时间（即slack 值）为3.228 ns。

表12-1的最后一行给出了从芯片引脚b_1到触发器$breg_1$输入端路径的最大保持时间为0.24 ns。因此，引脚b_1上的信号在时钟引脚上的每个有效沿后必须至少保持0.24 ns的稳定。我们假定没有为该参数设置约束条件，因此没有显示富余时间（即slack）值。

表12-1只列出了对F_{max}、t_{su}、t_{co} 和 t_h 最坏情况的路径。所实现的电路将有延迟时间更小、富余时间（即 slack 值）更大的许多其他路径。在一般情况下，静态时序分析工具还会为每个参数提供各自的表格，其中会列出更多的路径。

图12-1所示的CAD流程的最后阶段是时序仿真。我们在附录C中展示了如何通过对所实现的电路施加测试向量，然后观察其功能和时序行为来进行时序仿真。

12.3 小结

在本章中，我们简明地讲解了一个典型的设计流程，由于存在功能强大的CAD工具，所以可以实现这样的流程。我们只考虑了商业CAD系统中一个最重要的可用工具。为了学习更多的知识和技能，读者可以参阅参考资料[1-8]，或者访问CAD工具提供商的网站。表12-2列出了一些主要的CAD工具供应商，列出了网站地址和一些流行的CAD产品的名称。

表12-2 主要的CAD工具产品

供 应 商	网 站 地 址	产 品 名 称
Altera	altera.com	Quartus II
Mentor Graphics	mentorgraphics.com	ModelSim, Precision
Synplicity	synplicity.com	Synplify
Synopsys	synopsys.com	Design Compiler, VCS
Xilinx	xilinx.com	ISE

参考文献

1. R. Murgai, R. Brayton, A. Sangiovanni-Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays*, (Kluwer Academic Publishers, 1995).
2. J. Cong and Y. Ding, *FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs*, (in IEEE Transactions on Computer-aided Design 13 (1), January 1994).
3. S. Brown, R. Francis, J. Rose, Z. Vranesic, *Field-Programmable Gate Arrays*, (Kluwer Academic Publishers, 1995).

4. M. Breuer, *A Class of Min-cut Placement Algorithms*, (in Design Automation Conference, pages 284–290, IEEE/ACM, 1977).
5. Carl Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, (Kluwer Academic Publishers, 1988).
6. V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, (Kluwer Academic Publishers, 1999).
7. M. Sarrafzadeh, M. Wang, and X. Yang, *Modern Placement Techniques*, (Kluwer Academic Publishers, 2003).
8. L. McMurchie and C. Ebeling, *PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs*, (in International Symposium on Field Programmable Gate Arrays, Monterey, Ca., Feb. 1995).

部分习题答案

第2章

2.7 (a) 是 (b) 是 (c) 不是

2.12 $f = x_1x_3 + x_2x_3 + \bar{x}_2\bar{x}_3$

2.15 $f = (x_1 + x_2)(\bar{x}_2 + x_3)$

2.20 $f = x_2x_3 + x_1\bar{x}_3$

2.23 $f = (x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$

2.28 $f = x_1x_2 + x_1x_3 + x_2x_3$

2.32 $f = (x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)\times$
$(\bar{x}_1 + x_2 + x_3)$

2.33 $f = \bar{x}_1x_3 + \bar{x}_1x_2 + x_2x_3 + x_1\bar{x}_2\bar{x}_3$

2.40 该电路见图1。

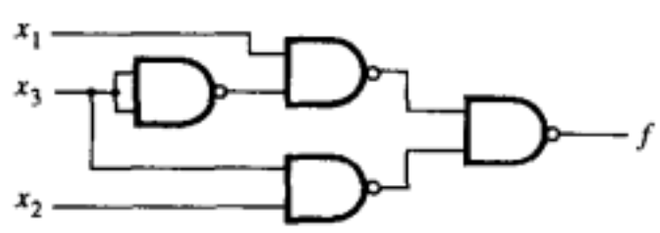


图 1

2.42 该电路见图2。

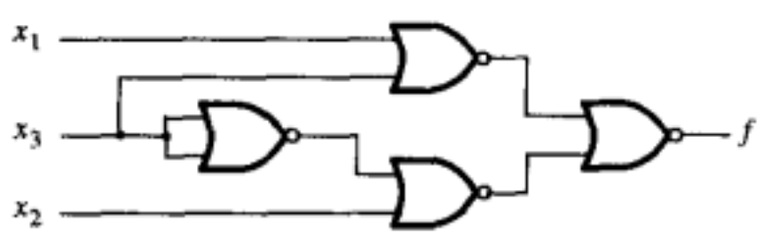


图 2

第3章

3.4 电路见图3。

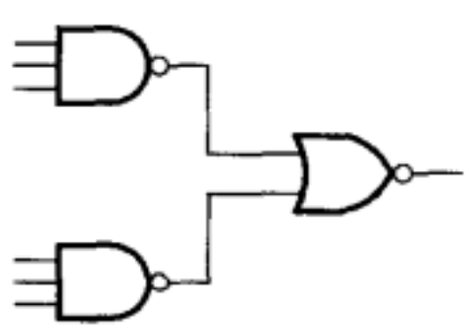


图 3

所需要的三极管数目是16。

3.8 完整的电路见图4。

3.12 所要求的电路见图5。

3.14 (a) I_D=800μA (b) I_D=78μA

3.17 R_{DS}=947Ω

3.25 (a) NM_H = 0.5V NM_L = 0.7V

(b) V_{OL}=0.8V NM_L=0.2V

3.28 (a) P_{NOT_gate}=163μW (b) P_{total}=8.2W

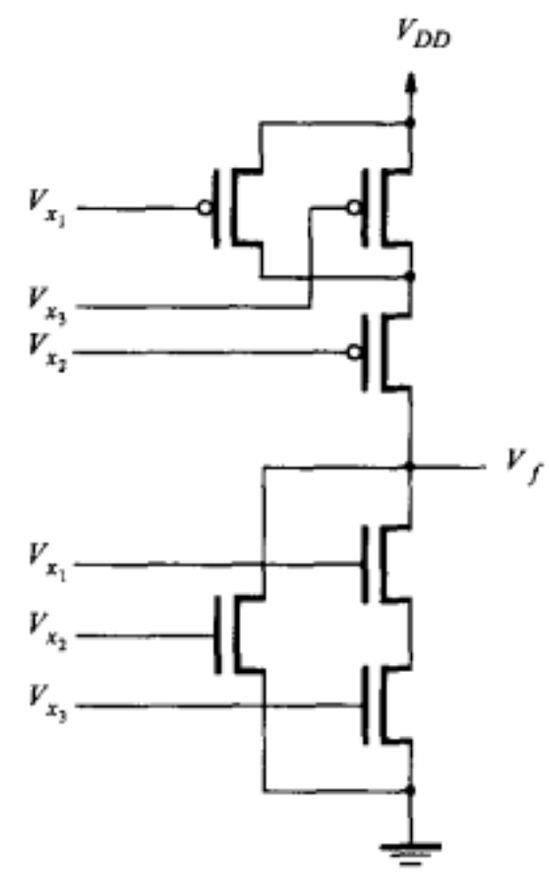


图 4

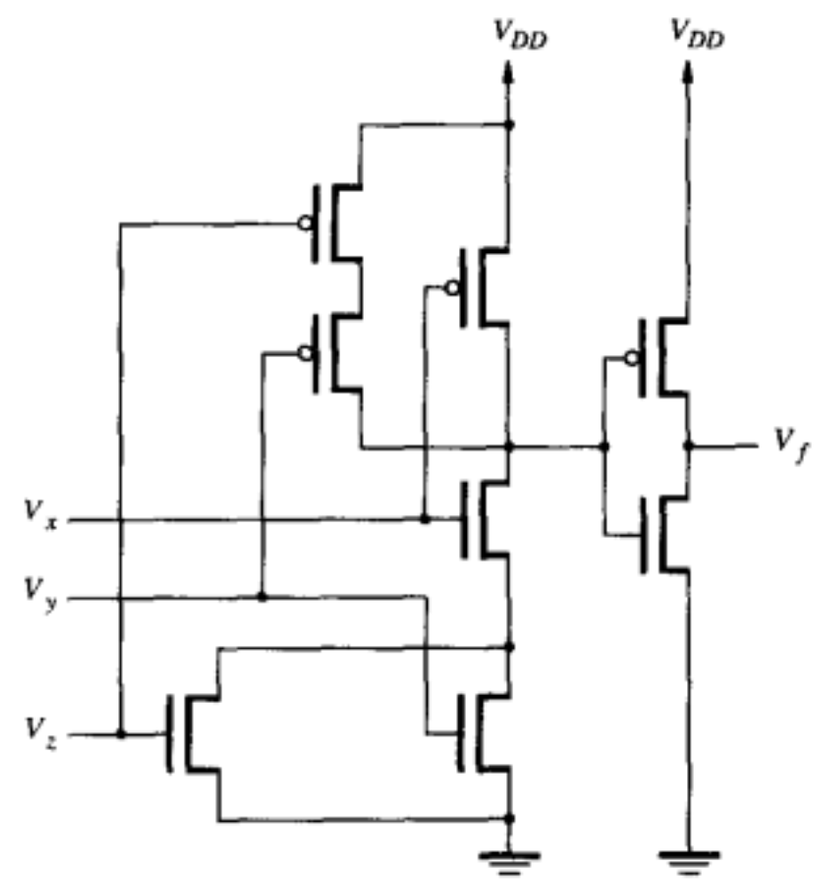


图 5

3.32 在CMOS的或非门中，两个NMOS三极管是并联的。若这两个NMOS三极管中只有一个“连通”，则出现输出为低电平时驱动电流最差的情况。因此，每个三极管电极的宽/长比必须与反相器中的NMOS相同，即$W_n / L_n = 2$。

在PMOS的或非门中，两个PMOS三极管是串联的。若这两个PMOS三极管电极的宽/长比为W_p / L_p，则这两个三极管可以被认为是宽/长比为$W_p /2L_p$的三极管。因此，每个PMOS三极管电极的宽度必须两倍于反相器电极的宽度，即$W_n/L_n = 8$。

3.45 $f = x_2 + x_1\bar{x}_3$。相应的电路见图6。

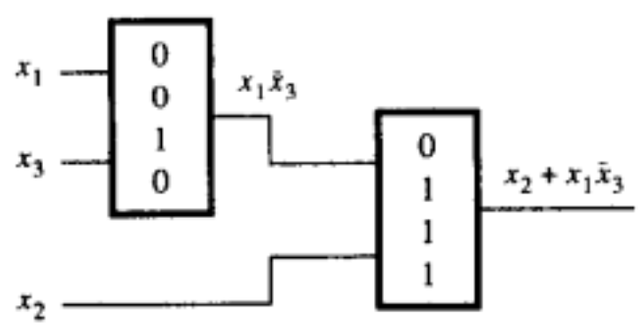


图 6

3.55 图P3-11中的电路是一个两输入端的异或门。该电路有两个缺点：当两个输入都为0时，该PMOS三极管必将 f 驱动至0，导致 $f = V_T$。另外，当 $x_1 = 1$，$x_2 = 0$时，NMOS三极管必将 f 驱动至1，导致 $f = V_{DD} - V_T$。

第4章

4.1 积之和形式：

$$f = \bar{x}_1 x_2 + \bar{x}_2 x_3$$

和之积形式：

$$f = (\bar{x}_1 + \bar{x}_2)(x_2 + x_3)$$

4.2 积之和形式：

$$f = x_1\bar{x}_2 + x_1 x_3 + \bar{x}_2 x_3$$

和之积形式：

$$f = (x_1 + x_3)(x_1 + \bar{x}_2)(\bar{x}_2 + x_3)$$

4.5 积之和形式：

$$f = \bar{x}_3\bar{x}_5 + \bar{x}_3 x_4 + x_2 x_4\bar{x}_5 + \bar{x}_1 x_3\bar{x}_4 x_5 + x_1 x_2\bar{x}_4 x_5$$

和之积形式：

$$f = (\bar{x}_3 + x_4 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5)(x_2 + \bar{x}_3 + \bar{x}_4)\times$$
$$(x_1 + x_3 + x_4 + \bar{x}_5)(\bar{x}_1 + x_2 + x_4 + \bar{x}_5)$$

4.9 $f = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4$

4.11 这个语句是错误的。以计数器为例，考虑 $f(x_1, x_2, x_3) = \sum m\,(0, 5, 7)$。

由此得到成本最低的积之和形式 $f = x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ 是唯一的。

而成本最低的和之积形式却有两个：

$$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(x_1 + \bar{x}_2)$$
$$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(\bar{x}_2 + x_3)$$

4.12 在组合电路中：

$$f = \bar{x}_2 x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3 x_4 + \bar{x}_1 x_2 x_3$$
$$g = \bar{x}_2 x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3 x_4 + x_1 x_2 x_4$$

f 和 g 两个表达式的前三个乘积项可以共享，因此总成本为31。

4.14 $f = (x_3 \uparrow g) \uparrow ((g \uparrow g) \uparrow x_4)$，其中 $g = (x_1 \uparrow (x_2 \uparrow x_2)) \uparrow ((x_1 \uparrow x_1) \uparrow x_2)$

4.15 $\bar{f} = (((x_3 \downarrow x_3) \downarrow g) \downarrow ((g \downarrow g) \downarrow (x_4 \downarrow x_4))$，其中 $g = ((x_1 \downarrow x_1) \downarrow x_2) \downarrow (x_1 \downarrow (x_2 \downarrow x_2))$。于是，$f = \bar{f} \downarrow \bar{f}$。

4.18 $f = \bar{x}_1(x_2 + x_3)(x_4 + x_5) + x_1(\bar{x}_2 + x_3)(\bar{x}_4 + x_5)$

4.21 $f = g \cdot h + \bar{g} \cdot \bar{h}$，其中 $g = x_1x_2$，$h = x_3 + x_4$

4.23 $f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1 x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_2x_3x_4$

4.32 将这两个函数分别用卡诺图形式表示，很容易就可以看到 $f = g$。

第5章

5.1 (a) 478 (b) 743 (c) 2025 (d) 41567 (e) 61680

5.2 (a) 478 (b) −280 (c) −1

5.3 (a) 478 (b) −281 (c) −2

5.4 这几个数字用不同的数制表示见图7。

十进制数	符号和幅度	1的补数	2的补数
73	000001001001	000001001001	000001001001
1906	011101110010	011101110010	011101110010
−95	100001011111	111110100000	111110100001
−1630	111001011110	100110100001	100110100010

图 7

5.11 是的，能工作。在 $i > 0$的那些级中，产生 c_i 的非门不需要用。

其缺点是 $\bar{c}_i = 1$ 经由最上端的NMOS三极管的传播“差”。优点是产生c_{i+1} 所需要的三极管少。

5.12 由表达式5.4，每个c_i需要 i 个与门和1个或门。因此，为了确定所有的 c_i 信号，我们需要 $\sum_{i=1}^{n}(i+1) = (n^2 + 3n)/2$ 个门。此外，还需要 $3n$个门来产生所有的g, p 和 s 函数。因此，总共需要 $(n^2 + 9n)/2$ 个门。

5.13 84个门。

5.17 图P5-2中的代码表示乘法器。该乘法器将输入的低两位与高两位相乘，得到四位的乘积输出。

5.21 可以用一个全加器电路，将该数字的其中两位连接到全加器输入x和y，第三位连接到进位输入。观察进位输出和求和位，就可以知道这三条输入线上有几位为1。见图8。

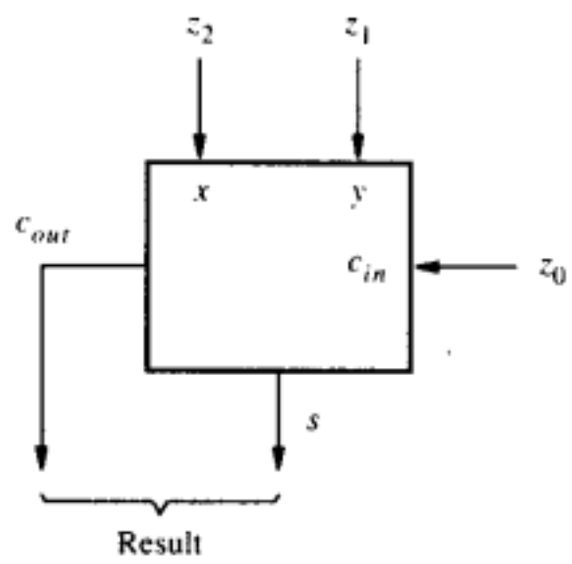


图 8

第6章

6.3 见图9。

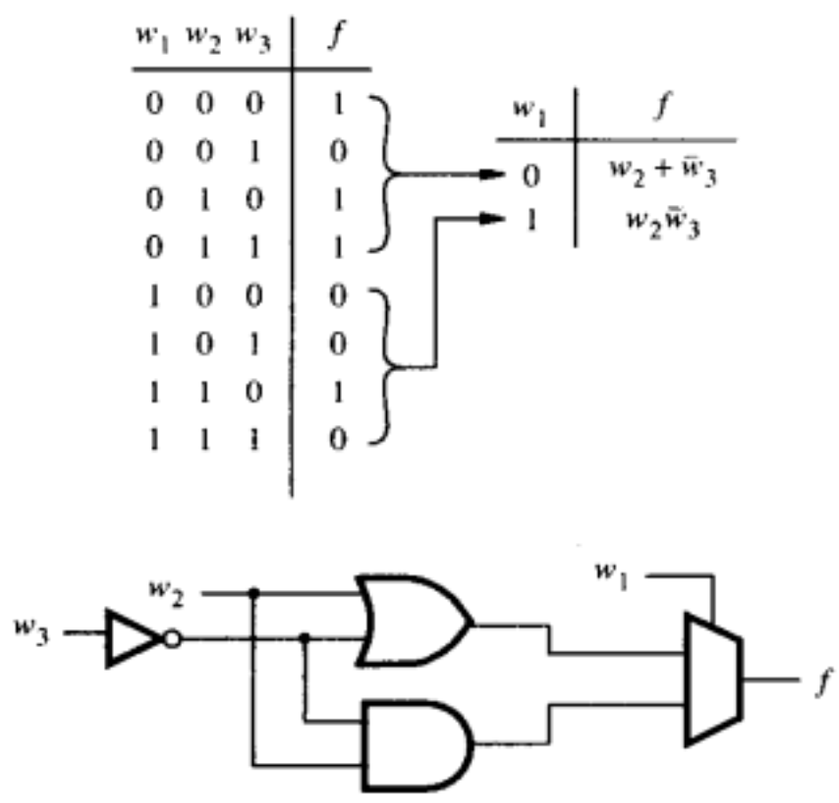


图 9

6.5 得到的电路见图10。

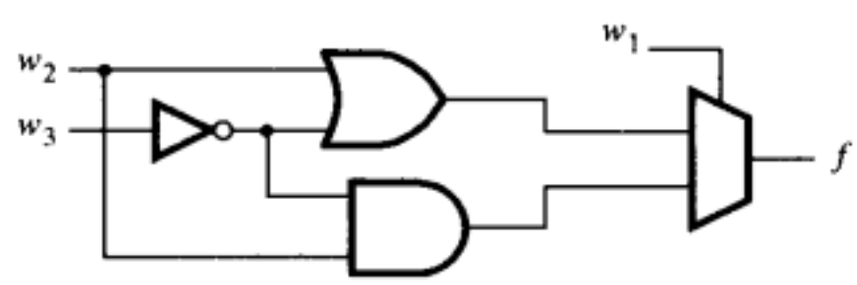


图 10

6.10 $f(w_1, w_2, \ldots, w_n) = [w_1 + f(0, w_2, \ldots, w_n)] \cdot [\overline{w}_1 + f(1, w_2, \ldots, w_n)]$

6.12 以w_2项作香农展开得到

$$f = \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3)$$
$$= w_2 \oplus (\overline{w}_1 + \overline{w}_3)$$
$$= w_2 \oplus \overline{w_1 w_3}$$

该电路的成本是：2个门 + 4 个输入= 6。

6.14 任何5变量函数都可以用两个4输入查找表实现。举例说明如下：如果将两个4-LUT级连起来，将前面的那个4-LUT的输出作为后面那个4-LUT的输入，就可以实现具有以下形式的任何函数：

$$f = f_1(w_1, w_2, w_3, w_4) + w_5$$
$$f = f_1(w_1, w_2, w_3, w_4) \cdot w_5$$

6.18 图P6-2所示的代码是一个具有使能控制输入的2-4译码器。但译码器的代码风格不好，不容易理解。此外，Verilog编译器经常会把 **if** 语句编译成为多路选择器。在这种情况下，编译后得到的译码器可能由具有使能En信号的一些多路选择器组成。

6.29

$$a = w_3 + w_2 w_0 + w_1 + \overline{w}_2 \overline{w}_0$$
$$b = \overline{w}_1 \overline{w}_0 + w_1 w_0 + \overline{w}_2$$
$$c = w_2 + \overline{w}_1 + w_0$$

第7章

7.4 该电路见图11。

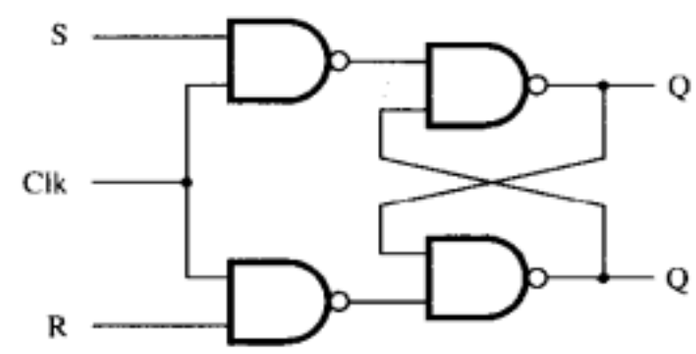


图 11

7.6 电路见图12。

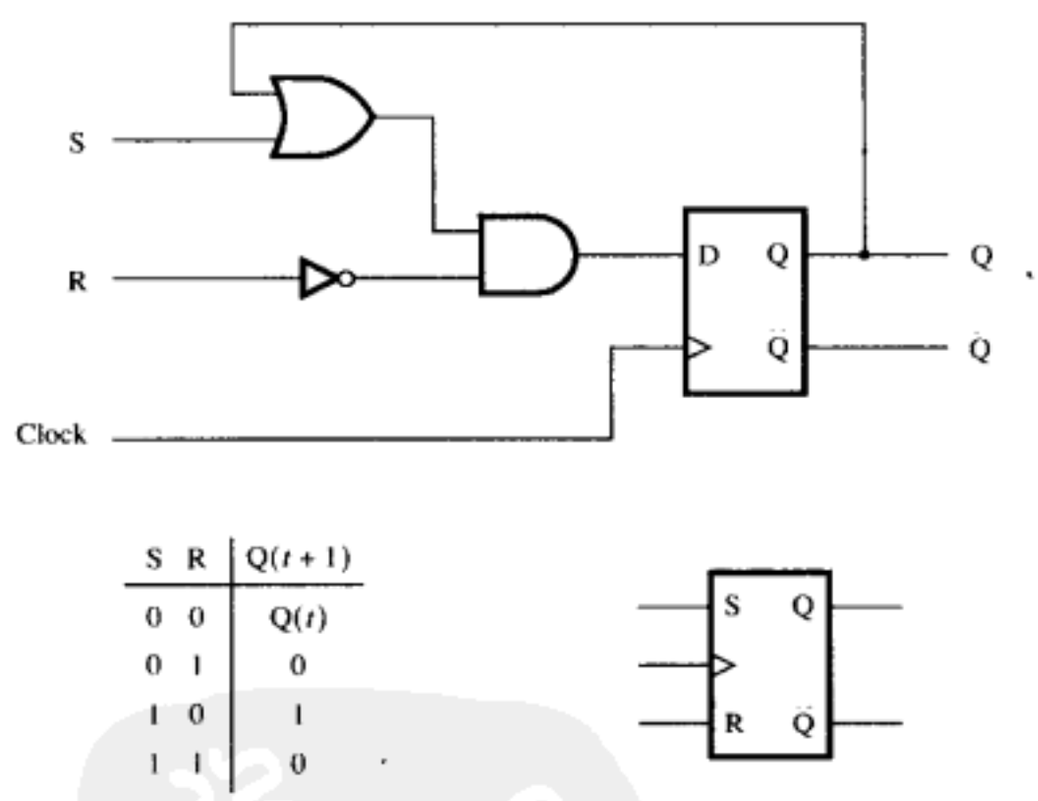


图 12

7.9 该电路的功能类似于负沿触发的JK触发器，其中 $J = A, K = B, Clock = C, Q = D, \overline{Q} = E$。

7.16 图13所示的电路就是想要的计数器。

7.18 该计数序列是 000, 001, 010, 111。

7.22 该电路最长的延迟是从FF_0 的输出到FF_3的输入。这个延迟总共有5ns。因此，该电路能可靠运行的最小周期是$T_{min} = 5 + 3 + 1 = 9$ ns，最高频率是$F_{max} = 1/T_{min} = 111$ MHz。

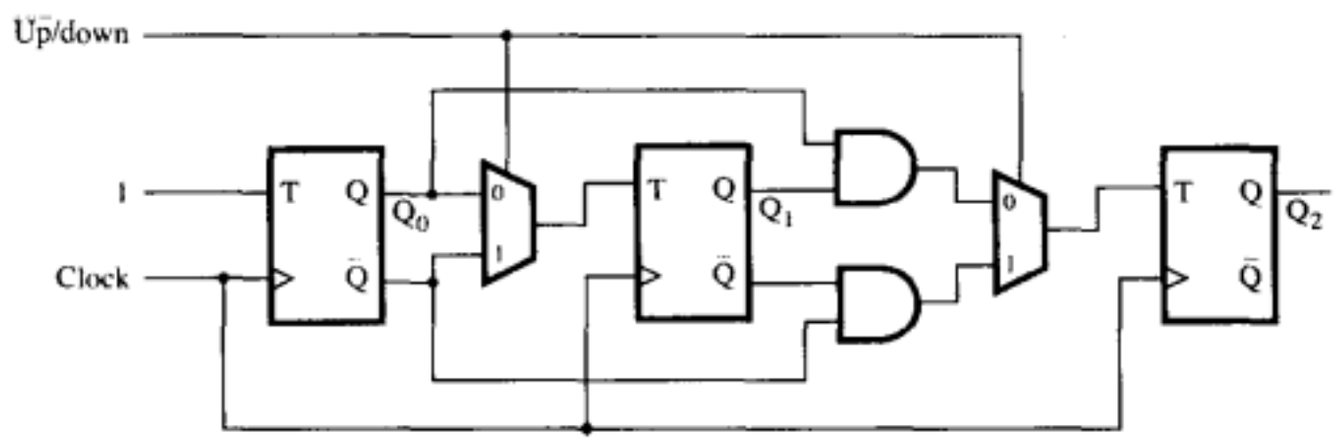


图 13

7.25 其Verilog 代码可以是

```
module accumulate (Reset, Clock, Data, Q);
   input Reset, Clock;
   input [3:0] Data;
   output reg [3:0] Q;

   always @(posedge Reset, posedge Clock)
      if (Reset)
         Q <= 0;
      else
         Q <= Q + Data;

endmodule
```

7.33 合适的电路见图14。

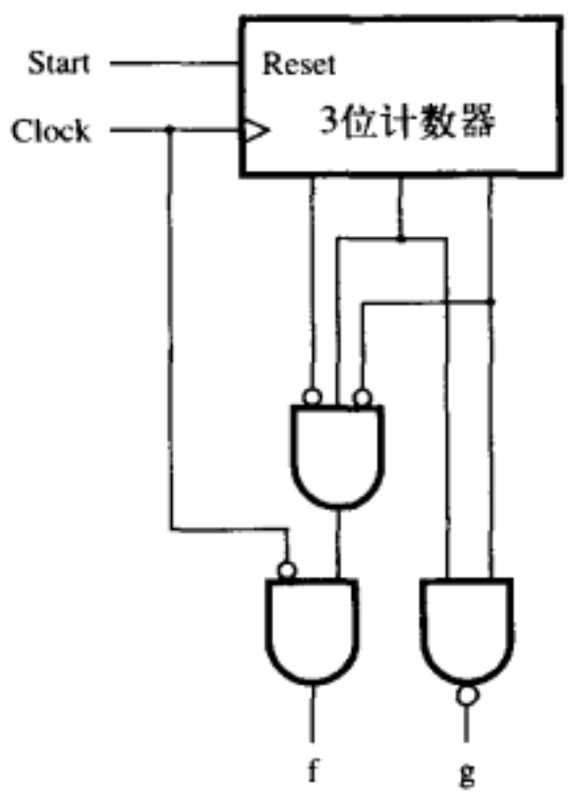


图 14

第8章

8.1 该触发器的输入表达式是

$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_1\overline{y}_2$$
$$D_1 = Y_1 = w \oplus y_1 \oplus y_2$$

其输出表达式是$z = y_1 y_2$。

8.2 该触发器的输入表达式是

$$J_2 = \overline{y}_1$$
$$K_2 = w$$
$$J_1 = \overline{w}y_2 + w\overline{y}_2$$
$$K_1 = J_1$$

其输出表达式是$z = y_1 y_2$。

8.5 最小状态表见图15。

当前状态	下一状态		输出
	$w = 0$	$w = 1$	z
A	A	B	0
B	E	C	0
C	D	C	0
D	A	F	1
E	A	F	0
F	E	C	1

图 15

8.6 最小状态表见图16。

当前状态	下一状态		输出z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	D	C	0	0
C	D	C	1	0
D	A	B	0	1

图 16

8.12 最小状态表见图17。

当前状态	下一状态		输出
	$w = 0$	$w = 1$	p
A	B	C	0
B	D	E	0
C	E	D	0
D	A	F	0
E	F	A	0
F	B	C	1

图 17

8.15 下一个状态的表达式是

$$D_4 = Y_4 = \overline{w}y_3 + wy_1$$
$$D_3 = Y_3 = \overline{w}(y_1 + y_4)$$
$$D_2 = Y_2 = \overline{w}y_2 + wy_4$$
$$D_1 = Y_1 = w(y_2 + y_1)$$

其输出表达式是$z = y_4$。

8.17 最小状态表见图18。

当前状态	下一状态		输出z	
	$w=0$	$w=1$	$w=0$	$w=1$
A	A	C	0	0
C	F	C	0	1
F	C	A	0	1

图 18

8.21 想要的电路见图19。

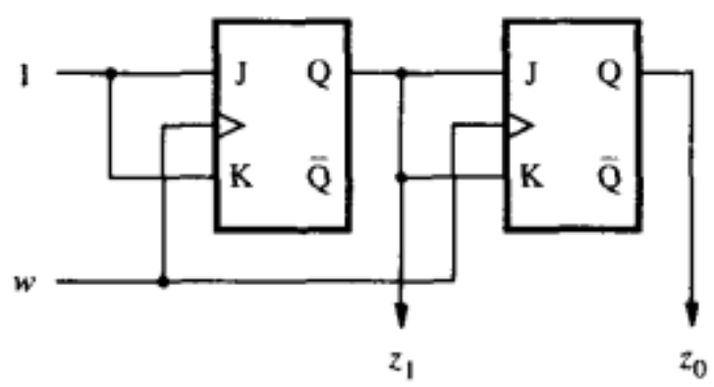


图 19

8.22 想要的电路见图20。

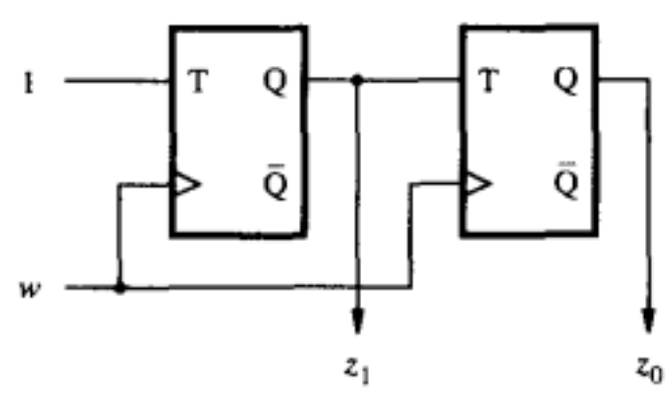


图 20

8.29 状态表见图21。

当前状态	下一状态		输出
	$w=0$	$w=1$	z
A	A	C	0
B	A	D	1
C	A	D	0
D	A	B	0

图 21

每当w端的输入序列中0后面跟着偶数个1的时候，该电路便可产生$z = 1$。

第9章

9.1 流程表见图22。

若进行状态 A↔D 和 B↔C 的内部交换，则该行为与图9-21a所示的流程表所描述的一致。

9.8 利用图9-40a所示的合并图，图9-39所示的有限状态机见图23。

当前状态	下一状态				z_2z_1
	$w_2w_1=00$	01	10	11	
A	D	C	D	C	11
B	D	D	(B)	(B)	10
C	D	(C)	D	(C)	01
D	(D)	C	B	C	00

图 22

当前状态	下一状态				输出
	$w_2w_1=00$	01	10	11	z
A	(A)	G	E	–	0
B	(B)	C	(B)	D	0
C	B	(C)	E	(C)	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
G	B	(G)	–	D	1

图 23

9.10 成本最低的无冒险竞争的逻辑实现表达式是

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1x_3x_4$$

9.12 成本最低的无冒险竞争的和之积表达式是

$$f = (x_1 + x_2 + x_4)(x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_3 + \bar{x}_4)\times$$
$$(x_2 + \bar{x}_3 + x_4)$$

9.14 若 $A = B = D = E = 1$ 且 C 从 0 变到1，则f从 0→1→0，且g从0→1→0→1。因此，f上有一个静态冒险，而g上有一个动态冒险。

9.17 激励表见图24。

下一个状态表达式是 $Y = w\bar{c} + cy + wy$ 。注意：这里添加 wy项是为了防止静态冒险竞争。

其输出表达式是 $z = cy$。

第11章

11.1 最小测试集合必须包括测试$w_1w_2w_3 = 011, 101$和 111，以及 000, 010或100之一。

11.3 这两个函数只有在顶点 $x_1x_2x_3x_4 = 0111$ 处不同。因此，这两个电路可以通过施加该输入取值加以区别。

11.5 测试是$w_1w_2w_3w_4 = 1111, 1110, 0111, 1111$。

11.9 如果输入线网w_1被粘连在1上，是不能检测出

当前状态 y	下一状态				输出			
	$wc = 00$	01	10	11	00	01	10	11
	Y				z			
0	0	0	1	0	0	0	0	0
1	0	1	1	1	0	1	0	1

图 24

来的。原因是该电路是高度冗余的，它实现的逻辑函数是 $f = w_3(\overline{w}_1 + \overline{w}_2)$，这只需要一个简单的电路就可以实现。

11.11 测试集合 = {0000, 0111, 1111, 1000}。该集合将与实现的异或门一起工作，如图4-26c所示。

对n位而言，可以用同样的样本。因此，测试集合= {00…00, 011…1, 11…1, 100…0}。

11.12 在如图6-16c所示的译码器电路中，只要使能信号En有效，这四个与门就使能。当En = 1时，要求的测试集合必须包括 w_1和 w_2 所有的四个取值。若En粘连在1上，也有必要用测试 $w_1w_2En = 000$ 将其检测出来。因此，完整的测试集合应该是：000, 001, 011, 101和111。

成功的逻辑电路设计人员必须深入理解基本概念，并熟练地掌握计算机辅助设计工具。本书很好地兼顾了基本概念的教学和计算机辅助工具的实际应用，着重介绍电路的综合及如何在实际的芯片上实现电路。书中通过许多例子来引入基本概念，这些例子涉及简单的电路设计，当中不但用手工的方法，也用现代的基于计算机辅助设计的方法来完成设计。本书使用的计算机辅助设计工具是当前流行的Quartus II软件，它能自动地把Verilog设计映射到复杂可编程逻辑器件(CPLD)和现场可编程门阵列(FPGA)中，而这两种器件是工业界最常用的可编程逻辑器件。

作者简介

Stephen Brown 拥有加拿大新布鲁斯威克大学电机工程学士学位，多伦多大学电机工程硕士和博士学位。目前他是多伦多大学电机与计算机工程系教授，同时也是Altera公司多伦多技术中心的开发工程和大学计划部主任。其研究方向包括：现场可编程超大规模集成电路技术和计算机体系结构。他曾四次荣获电机工程、计算机工程和计算机科学课程的最佳教学奖。

Zvonko Vranesic 拥有加拿大多伦多大学电机工程学士、硕士和博士学位。目前他是多伦多大学电机与计算机工程系及计算机科学系教授。他还参与Altera公司多伦多技术中心的研究和开发工作。他代表加拿大参加过多次国际象棋比赛，拥有国际象棋大师的头衔。其研究方向包括：计算机体系结构和现场可编程超大规模集成电路技术。

随书光盘中包括Quartus II软件和本书中的Verilog代码，以便学生能第一手操作、运行和测试书中的设计范例。另外，光盘中还包括附录部分的PDF文件。

上架指导：电子/数字逻辑

投稿热线：(010) 88379511
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计：杨宇梅

ISBN 978-7-111-22182-1
定价：65.00 元(附光盘)