J. Bhasker 著
孙海平 等译

Verilog® HDL综合实用教程

Verilog® HDL Synthesis
A Practical Primer

清华大学出版社

Verilog® HDL Synthesis
A Practical Primer

该书易于阅读，并提供了大量可综合的Verilog模型范例，为学生和从事逻辑设计的工程师快速掌握Verilog HDL综合方面的知识提供了捷径。

—— Vassilios Gerousis，
摩托罗拉公司高级工程师

通过本书，您可以：

- 迅速开始编写可综合的Verilog模型。
- 获悉哪些语言结构可用于综合，这些结构如何映射成硬件，以得到所期望的逻辑电路。
- 学习如何避免功能的不匹配。
- 立即开始使用许多常用的硬件元件模型，或针对应用稍作修改后为己所用。

该书是讲授基于Verilog的综合技术的理想教材，它不仅向读者演示了各种Verilog结构所得出的硬件，还展示了如何剪裁Verilog程序以获得所期望的硬件。

—— Jim Vellenga，Viewlogic Systems公司

该书揭示了仿真和综合时必然会出现差别的各种情形，精心挑选的案例使得初学者和有经验的设计者都能意识到这些在调试时难以发现但却极为普遍的陷阱。

—— Carlos Roman，贝尔实验室

这是一本极好的指南书，清晰、简洁地阐明了如何设计可综合的RTL模型。

—— Douglas J. Smith，"HDL Chip Design"的作者

该书采用示例引导的方式来组织，对Verilog初学者颇有价值。

—— Egbert Molenkamp，University of Twente（荷兰）

J. Bhasker 是IEEE PAR 1364.1 Verilog Synthesis Interoperability Working Group（Verilog SIWG）的主席，该组织致力于建立用于RTL综合的Verilog标准化子集。他是贝尔实验室所开发的ArchSyn综合系统的主要设计者之一。他曾为AT&T和Lucent的许多设计师讲授Verilog HDL语言和Verilog HDL综合课程。他还著有另外一本畅销书"A Verilog HDL Primer"。

ISBN 7-302-07714-2

9 787302 077145 >

定价：24.00元

J. Bhasker 著

Distinguished Member of Technical Staff
Bell Labs, Lucent Technologies

孙海平 等译

Verilog® HDL 综合实用教程

Verilog® HDL Synthesis
A Practical Primer

清华大学出版社
北京

J. Bhasker
Verilog® HDL Synthesis, A Practical Primer
EISBN: 0-9650391-5-3

Original English language edition published by Star Galaxy Publishing.

Chinese translation edition is published and distributed exclusively by Tsinghua University Press under the authorization by Star Galaxy Publishing, in the territories throughout the world.
本书中文翻译版由美国 Star Galaxy Publishing 授权清华大学出版社在全球范围内独家出版发行。

北京市版权局著作权合同登记号 图字 01-2003-4338

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

Verilog® HDL 综合实用教程/(美)巴斯克尔(Bhasker, J.)著；孙海平等译. —北京：清华大学出版社，2004.1
书名原文：Verilog® HDL Synthesis, A Practical Primer
ISBN 7-302-07714-2

Ⅰ.V… Ⅱ.①巴… ②孙… Ⅲ.硬件描述语言，Verilog HDL—程序设计 教材 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2003)第 108656 号

出 版 者：清华大学出版社 **地 址**：北京清华大学学研大厦
http://www.tup.com.cn **邮 编**：100084
社 总 机：010-62770175 **客户服务**：010 62776969
责任编辑：张 靓
封面设计：常雪影
印 刷 者：北京密云胶印厂
装 订 者：北京市密云县京文制本装订厂
发 行 者：新华书店总店北京发行所
开 本：185×230 **印张**：11.75 **字数**：241 千字
版 次：2004 年 1 月第 1 版 2004 年 1 月第 1 次印刷
书 号：ISBN 7-302-07714-2/TP・5649
印 数：1~4000
定 价：24.00 元

译 者 序

数字集成电路在过去 30 年里得到了长足发展，EDA（电子设计自动化）技术起到了至关重要的作用。其中，用于表达设计对象的硬件描述语言（HDL）采用形式化方法，不仅可以准确、直观地对数字电路进行建模和仿真，而且极大地提高了电子设计的效率和产出，为顺应半导体工艺技术和应用需求的急速发展提供了可能。目前我国 ASIC（专用集成电路）设计业的基本状况是芯片设计开发工作远远滞后于电子产品发展的需求，滞后于芯片生产线的吞吐能力。为了适应目前系统芯片（System-on-a-Chip，SoC）时代的设计需求，提升设计能力和效率，国内推广和开展基于 HDL 和各种 EDA 工具的设计方法已经成为迫切的需要。

Verilog 作为当今国际主流的 HDL 语言，20 世纪 90 年代初成为 IEEE 标准之后，在数字电路和芯片的前端设计中得到了更为广泛的应用。Verilog 最初是为了仿真和模拟数字电路和数字系统的运行而设计出的一种形式化语言，随着半导体工艺的迅猛发展，电路规模和芯片集成密度迅猛扩张，手工绘制电路图已不能满足设计需求了，此时人们就把作为模拟工具的硬件描述语言发展成电路设计的手段，在 EDA 工具的辅助下把采用硬件描述语言描述的设计对象自动综合成实际电路。

Verilog 的发展历程决定了其语义的丰富性。这种丰富性一方面决定了其模拟能力十分强大，另一方面决定了并不是每种描述出来的现象都可以直接生成实际电路，语言结构的仿真功能与所综合出的电路的功能可能不一致，各种语言结构搭配起来使用产生的似是而非的微妙差异都会导致综合出的电路功能完全不同。为避免歧义和误解，2002 年 11 月 IEEE 正式制定了 Verilog 寄存器传输级综合标准。

本书作者 Bhasker 博士是 Verilog 综合互操作性工作组的主席，长期从事有关 HDL 的研究。他不仅对各种 HDL 的语义有着深入、准确的理解，而且直接主持制定了 Verilog HDL 综合的 IEEE 标准。本书是这方面的权威著作。鉴于国内介绍 HDL 的各种书籍囿于语法层面的介绍，而没有着眼于综合的角度来讲解 Verilog 语言的语义本质，不能解决初学者和许多设计人员在设计、调试时的困惑，清华大学出版社适时引进并出版这本“Verilog® HDL Synthesis，A Practical Primer”以及 Bhasker 博士撰写的另一本著作“A SystemC™ Primer”的中文翻译版，对国内迫切需要此类图书的读者犹如久旱逢甘霖。

本书的鲜明特色在于帮助读者全而、正确地理解 Verilog 硬件描述语言的综合。本书以电路综合为目标，针对各种语言结构逐一讨论了其可综合性、仿真与综合时的语义差别以及相关的各种用法，给出了大量示例，对各种似是而非的用法作了对比，指出了其语义差别和所综合出的电路在功能上的差异。本书的另一特色在于详细介绍了设计模型的

优化技术和验证技术。

本书内容全面、深入浅出、适用面广，对于已经采用或打算采用 Verilog 语言作为电路设计手段的学生和工程人员而言是一本不可多得的好书。

全书由孙海平、刘方海和郑静翻译，孙海平统稿，何伟、徐学迅和郑静等作了修订和校对。在此，谨向为本书出版付出辛勤劳动的所有人员致以诚挚的感谢！

在翻译过程中我们力求译意准确，但限于水平，必然存在错误和不足之处，恳请读者通过电子邮件 hp.sun@263.net 批评指正。

译　者

2003 年 10 月

原 书 序

关于 Verilog HDL 综合的讨论早在 1988 年就已经展开。但时至今日，此领域的优秀教材尚未囊括其基本概念。这本关于 Verilog HDL 综合实用教程全面地介绍了这一新技术。它通过提供便于理解的、与综合技术相关的 Verilog 语义，揭开了 HDL 综合的神秘面纱。本书作者 Bhasker 是综合领域的专家，在此领域已研究了 14 年多。身为 IEEE 工作组主席，他利用自己的专长领导了 Verilog RTL 综合标准(PAR 1364.1)[①]的开发，此项工作是以 1998 年 4 月公布的 OVI[②] RTL 综合子集为基础开展的，而 Bhasker 正是 OVI 的 RTL 综合标准的缔造者之一。

Bhasker 的这本著作为学生和刚刚从事逻辑设计的人提供了系统掌握 Verilog HDL 综合的捷径。此书文字浅显易懂，列举了大量可综合的 Verilog HDL 模型示例。读者可以系统地了解 Verilog HDL 的语言结构、它们在综合时的含义、综合设计技术如何把这些结构转换成逻辑门电路，以及它们对设计验证的影响。此书给出了大量 Verilog HDL 模型示例及其等价的逻辑门电路。这些示例虽然简单，但展示了不同的逻辑建模方式，如组合逻辑、时序逻辑、基于寄存器和锁存器的设计、有限状态机、算术单元等。

本书不仅为初学者提供了 HDL 综合方面的知识，还讨论了一些高级论题，诸如如何从综合模型得出优化的逻辑等。其中，资源共享和分配是模型优化的论题之一。另一个独特的论题是对设计的验证。本书阐述了编写综合模型以确保得到可预测和可验证的结果的基本原则。尽管有一章立足于仿真，但是所讨论的那些概念同样可用于形式化验证。

本书是第一本对 Verilog HDL 综合进行全面论述的著作。作者 Bhasker 在 Lucent Technologies(朗讯科技)讲授 Verilog HDL 语言和 Verilog HDL 综合长达三年多。这本书是他 14 年来在 Verilog HDL 综合方面的研究成果的总结。尽管本书定位于初学者，但是专业人员也可从基本原则及综合建模的高级论题中获益。不容置疑，知识产权(IP)开发人员应遵循本书所推荐的建模方式。

Vassilios C. Gerousis
Motorola 高级工程师 Phoenix, Arizona
Open Verilog International 技术协作委员会主席

---

① 译者注：该工作组提交的“Verilog 寄存器传输级综合 IEEE 标准”已于 2002 年 11 月 10 日被批准成为 IEEE 标准，编号是 IEEE Std 1364.1™-2002。

② Open Verilog International

前　言

本书是 Verilog HDL 寄存器传输级综合方面的实用指南，提供了大量可综合的 Verilog HDL 示例，详细介绍了 Verilog HDL 综合所支持的各种语法结构，并且用示例说明了如何把这些可综合的结构搭配起来对各种硬件元件进行建模。本书还详细讲解了造成设计模型和综合出的网表功能不一致的常见原因，并给出了避免产生这些错误的建议。

对很多人而言，综合看起来像是一个黑箱，输入的是 Verilog HDL 描述的设计，而输出的是门级网表。这种黑箱方式看起来似乎存在着某些奥妙。正确理解综合过程中出现的各种变换，才能充分利用综合系统并充分发挥它的长处。Verilog HDL 终究是一种建模语言，本书的目的就是要通过介绍从硬件描述语言模型到网表这一综合过程中出现的各种变换来揭示黑箱中的奥秘。

Verilog 硬件描述语言通常指的是作为 IEEE 标准（IEEE Std 1364）的 Verilog HDL。它可以用来描述时序的和并发的行为，也可用来描述模型的结构。它支持在从体系结构级到开关级的多个抽象层次上描述设计。该语言支持对设计进行层次化建模，此外还提供了大量内建的基本元件，包括逻辑门和用户自定义的基本元件。各种语言结构都具有精确的仿真语义，因此可以用 Verilog HDL 仿真器来验证采用该语言编写出的模型。

通常，对于不同的人而言，“综合”有不同的含义。本书中，综合指的是对 Verilog HDL 描述的设计进行综合，该设计描述了组合逻辑和（或）时序逻辑。对于时序逻辑，清晰地描述了其受时钟控制的行为。这排除了讨论逻辑综合（用逻辑门基本元件描述的设计）和高层次综合（不用时钟信息来指定设计对象的行为）。综合过程把 Verilog HDL 模型转换成门级网表。通常假定目标网表是被模拟的逻辑与工艺无关的表示形式。目标工艺包含诸如逻辑门之类与工艺无关的通用功能块，以及诸如算术逻辑单元和比较器之类的寄存器传输级功能块。对于综合流程的后续阶段，如工艺转换（即从通用门到库中特定部分的映射）和模块绑定（即采用逻辑门基本元件来构建寄存器传输级功能块），本书均未涉及。

之所以很难编写一本关于综合的书，是因为其具有发展迅速的特性。因此，本书所提供的是大体上成立的基本信息，尽可能避开了模棱两可的论题（包括与特定实现相关的问题）。由于 Verilog HDL 语言的丰富性，描述一种行为可能存在着不止一种方式，本书提出了一两种可综合的建模方式。另外，并非该语言中的所有结构都是可综合的，因为 Verilog HDL 最初是被设计成一种仿真语言。因此，本书将介绍主流综合系统所支持的各种结构。

同样，本书还避免提及特定综合工具厂商提供的不同特性。然而，某些特定情况下也有必要介绍某种实现示例。此时，所介绍的特性都在朗讯科技的贝尔实验室开发的 ArchSyn（14.0 版本）综合工具中得到了实现。

注意：不是所有的综合系统都支持本书中描述的 Verilog HDL 结构。任何关于综合系统的专有特性的更多细节，读者都需要及时参考相应厂商的文档资料。

笔者是 Verilog 综合互操作性工作组的主席，该工作组目前正在开发 RTL 级综合的 IEEE 标准。

本书假定读者已经具备了 Verilog HDL 语言的基础知识。Star Galaxy 出版社的“A Verilog® HDL Primer”是一本关于 Verilog HDL 语言入门的好书。

本书面向电子工程师，尤其是那些对于理解综合的技巧感兴趣的电路与系统设计人员。本书不打算解释任何综合算法。作者相信一旦理解了综合结果会是什么，就能够编写出有效的设计模型，从而对综合出的设计对象的品质能有所控制。这是因为综合出的电路结构易受编写模型的方式的影响。

本书可以用作高校教材。在电子工程专业的教学大纲中，本书可以在计算机辅助设计方面的 VLSI（超大规模集成电路）课程中使用。学生可使用本书编写多种模型，并在任何可用的综合系统中对它们加以综合，以研究综合过程中出现的各种变换。在计算机科学专业的课程（如计算机辅助设计的算法课程）中，学生可以编写简单的综合程序来识别 Verilog HDL 语法的一个子集并生成综合的网表。本书中的示例可用作测试用例以供理解所生成的网表。

专业工程师将本书作为参考书也可以从中获益。工程师们可以在大量模型示例及其综合出的网表中直接寻找自己感兴趣的部分加以研究。

本书的组织结构

第 1 章介绍综合过程的基础知识，诸如什么是连线、触发器和状态以及如何确定对象的大小之类的内容。

第 2 章介绍 Verilog HDL 结构向逻辑门的映射。通过组合逻辑的示例说明如何把 Verilog HDL 结构变换成基础逻辑门以及它们的互连结构。还通过异步置位和清零、同步置位和清零、多时钟、多相位时钟等建模示例介绍了各种模拟时序逻辑设计的方式。

有时也有必要使用预先设计的功能块，因此第 2 章进一步介绍了如何对结构进行建模，包括在行为模型中采用部分结构建模。

第 3 章介绍如何把 Verilog HDL 的各种结构搭配起来对硬件元件进行建模。虽然第 2 章介绍了 Verilog HDL 向逻辑门的映射，但本章介绍的是另一方面，即如何用 Verilog HDL 来建立硬件元件的可综合模型。本章提供了许多通用硬件元件的详尽示例，如多路选择器、计数器、译码器以及算术逻辑单元等。

第 4 章介绍可应用于 Verilog HDL 模型以综合出优质网表的各种有效技术。

本章介绍的各种优化手段如果不能由综合系统自动实现，则需要由设计者手工加以实现。

有了 Verilog HDL 综合模型，通常还有必要用输入的设计模型来验证综合出的网表。第 5 章提供了编写检验综合结果的测试平台的策略。因为 Verilog HDL 不是为了综合而专门设计的语言，设计出的模型与综合出的网表可能会出现功能上的不一致。本章解释了产生那些分歧的原因。

为了说明典型的综合系统所支持的 Verilog HDL 可综合子集，附录 A 介绍了 ArchSyn 综合系统所支持的结构。但是，不同综合系统的可综合子集之间是存在差别的。

附录 B 给出了本书中综合出的网表用到的各种逻辑门的说明。

注意：本书展示的综合出的网表不是优化过的网表，因此在某些情况下这些逻辑可能不是最理想的。这是可以接受的，因为本书的目的是体现 Verilog HDL 到逻辑门之间的变换，而不是用来说明各种逻辑优化技术。本书中的有些网表已经被有目的地优化过了，因此可以把那些网表作为经典记录下来。

约定

本书的很多地方使用了词语“设计者”，它泛指任何阅读本书的读者。此外，术语“综合工具”和“综合系统”在本书中互换使用，它们指的都是读入 Verilog RTL 模型并生成门级网表的程序。

本书出现的所有 Verilog HDL 描述，保留字都用黑体印刷。有时候 Verilog HDL 源程序中出现的省略号（...）用来表示与当时讨论的内容不相关的代码。

本书采用的所有示例都已使用 ArchSyn 综合系统加以综合。附录 B 介绍了综合出的网表中使用的各种逻辑门。

致谢

衷心感谢以下为本书审稿和提供建设性意见的个人，他们提出了很多发人深省的评论，对于改进本书有着直接的帮助。真诚地感谢他们在百忙之中抽出时间和精力审阅本书。

a) Cliff Cummings，Sunburst Design 公司

b) Joe Pick，Synopsys 公司

c) Doug Smith，VeriBest 公司

d) Egbert Molenkamp，荷兰 Twente 大学

e) Carlos Roman、Jenjen Tiao、Jong Lee 和 Sriram Tyagarajan，朗讯科技，贝尔实验室

f) Jim Vellenga、Ambar Sarkar，Viewlogic Systems 公司

非常感谢他们！

还要向 Hao Nham 表示感谢，他为我在贝尔实验室创建了良好的工作氛围，并鼓励我在日常工作之外完成本书的编写。

当然，如同笔者撰写其他书那样，如果没有生活、家庭、我的妻子 Geetha 以及两个孩子 Arvind 和 Vinay 不断带给我欢乐、喜悦和继续写作的动力，本书就不可能得以完成。

J. Bhasker

1998 年 8 月

目　录

第1章 基础知识

Verilog HDL是一种硬件描述语言，它不仅可以在门级和寄存器传输级(Register-Transfer Level，RTL)描述硬件，也可以在算法级对硬件加以描述。因此，将采用Verilog HDL语言描述的设计转变成逻辑门构成的电路绝非简单的处理过程。

本章将介绍Verilog HDL模型映射成逻辑门所涉及的基础知识。

1.1 什么是综合?

综合就是从采用Verilog HDL语言描述的寄存器传输级电路模型构造出门级网表的过程。[①] 图1-1对此综合过程作了说明。综合可能是个中间步骤，它生成的网表是由用导线相互连接的寄存器传输级功能块(如触发器、算术逻辑单元和多路选择器)组成的。此时，就有必要使用被称为RTL模块构造器的程序了。该构造器用来针对用户指定的目标工艺从预定义元件库中构造或获取每一个必需的RTL功能块。

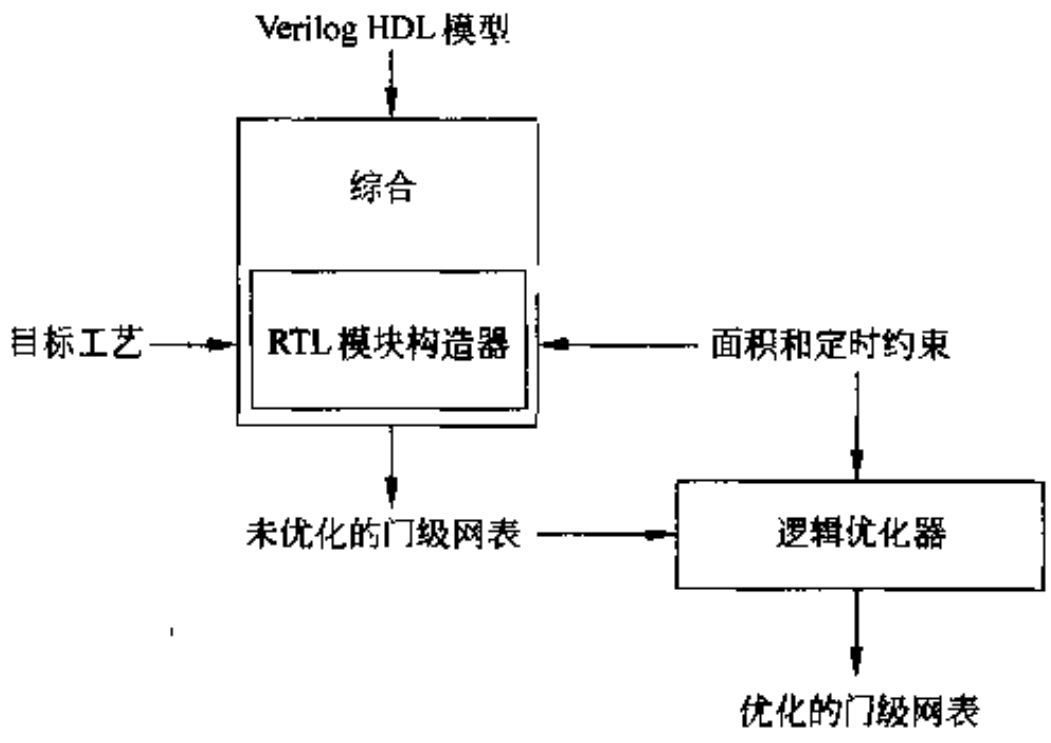

图1-1 综合过程

① 这是本书中使用的定义。

产生门级网表之后，逻辑优化器读入网表并以用户指定的面积和定时约束为目标优化网表。这些面积和定时约束也可以用来指导模块构造器恰当地选取或生成寄存器传输级功能块。

本书中，假定目标网表是门级的。附录 B 对综合出的网表中所使用的各种逻辑门作了介绍。但本书并不涉及模块构造和逻辑优化阶段的内容。

图 1-2 列出了 Verilog HDL 的各种基本元素和硬件中所使用的各种基本元件。必须使用某种映射机制或者构造机制将 Verilog HDL 元素转变成相应的硬件元件。由此，产生以下问题：

a）数据类型如何转变成硬件？

b）常量如何映射成逻辑值？

c）语句如何转变成硬件？

后续章节将更详细地讨论这些映射过程。

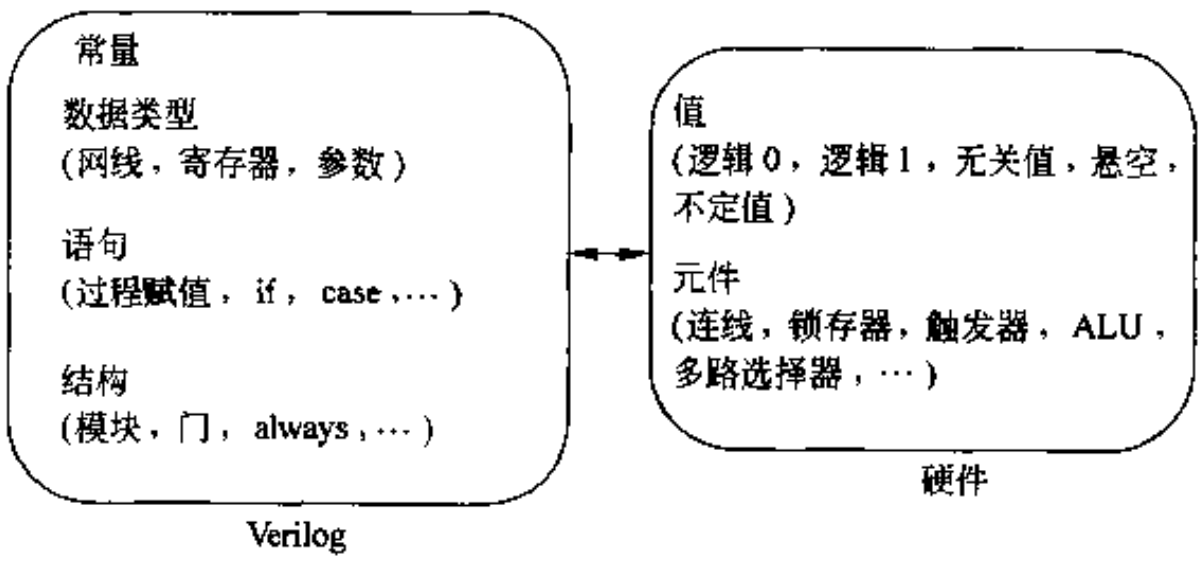

图 1-2 综合涉及的两个领域

1.2 设计流程中的综合

Verilog HDL 这种硬件描述语言允许用户在不同的抽象层次上对电路进行建模，这些层次从门级、寄存器传输级、行为级直至算法级。因此，同一个电路就可以有多种不同的描述方式，但不是每一种描述都是可综合的。事实上，Verilog HDL 原本被设计成一种仿真语言，而不是一种用于综合的语言。结果导致 Verilog HDL 中有很多结构没有相应的硬件可以对应，例如系统调用 **$display**。同样也不存在用于寄存器传输级综合的 Verilog HDL 标准子集。

正是由于存在这些问题，不同的综合系统所支持的 Verilog HDL 综合子集是不同的。由于 Verilog HDL 中不存在单个的对象来表示锁存器或触发器，所以每一种综合系

统都会提供不同的机制以实现锁存器或触发器的建模。因此各种综合系统都定义了自己的 Verilog HDL 可综合子集以及自己的建模方式。

图 1-3 中使用 Verilog HDL 以不同的方式描述了同一个电路。某综合系统支持对方式 A 和方式 B 的综合，但可能不支持对方式 C 的综合。这意味着综合模型在不同的综合系统之间通常是不可移植的。而方式 D 可能根本就不可综合。

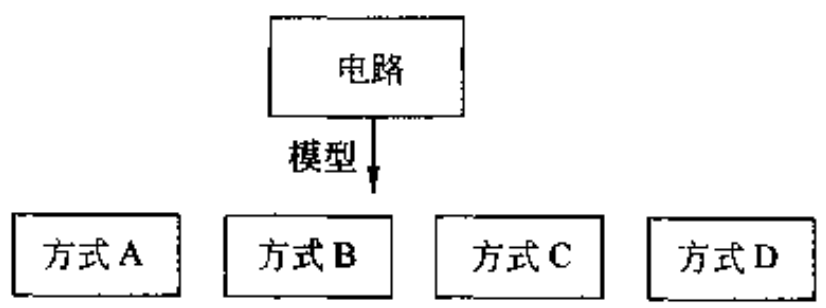

图 1-3 同样的行为，不同的建模方式

这一局限给设计者造成了严重障碍，因为设计者不仅需要理解 Verilog HDL，而且还必须理解特定综合系统的建模方式，才能编写出可综合的模型。图 1-4 是典型的设计流程，但 Verilog HDL 综合也并非总是遵循该流程。

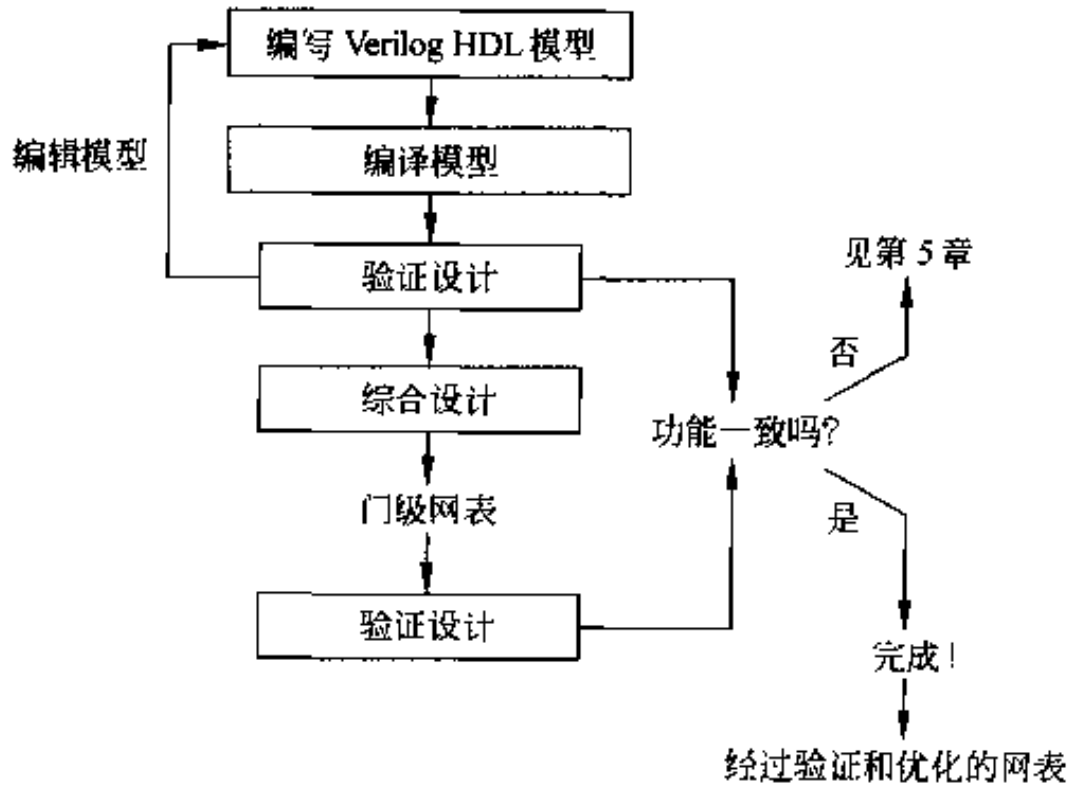

图 1-4 典型的设计流程

此设计流程的问题在于：如果编写 Verilog HDL 模型的时候并不了解综合系统的建模方式(这里假定是为了综合才编写那些模型，否则编写出不可综合的模型也无妨)，而只有进入综合阶段设计者才能了解特定综合系统建模方式的限制。这时可能需要改写模型，而且大量的时间可能已经浪费在“编写 Verilog HDL 模型”→“编译模型”→“验证设计”→“编辑模型”的循环之中。因此 Verilog HDL 综合应采用如图 1-5 所示的更实用的

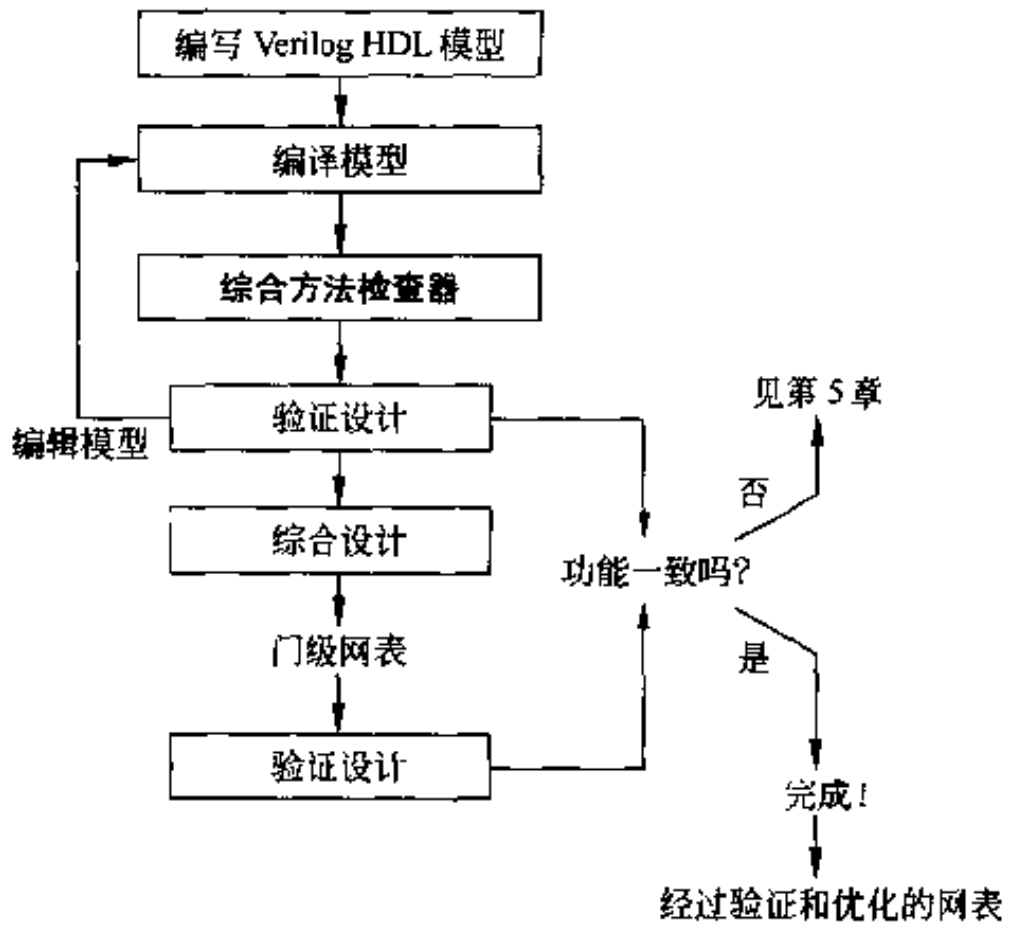

图 1-5 新的设计流程

设计流程。需要使用综合方法检查器来确保编写出的模型是可综合的。注意此项工作必须在第一个验证循环中完成。首先通过这种方式验证模型是可综合的，然后再对其进行综合。

1.3 逻辑值体系

硬件建模中常用的值有：

a) 逻辑 0

b) 逻辑 1

c) 高阻抗

d) 无关值

e) 不定值

Verilog HDL 对无关值之外的其他值都作了明确的定义。当值 x 被赋给某变量，综合系统就把此值视为无关值。Verilog HDL 的值与硬件建模中的值之间的对应关系如下：

a) *0* <--> 逻辑 0

b) 1 <--> 逻辑 1

c) z <--> 高阻抗

d) z <--> 无关值 (casex 与 casez 语句)

e) x <--> 无关值

f) x <--> 不定值

1.4 位 宽

1.4.1 数据类型

Verilog HDL 中,变量属于以下两种数据类型:

a) 网线数据类型

b) 寄存器数据类型

网线数据类型

网线声明语句中可明确地指定位宽。

```
wire [4:0] Dak;        // 5 位的 wire 网线
wor Ax;                // 1 位的 wor 网线
```

如果网线声明语句中未明确指定位宽,则默认位宽是 1 位。

能够综合的网线数据类型有:

```
wire  wor  wand  tri  supply0  supply1
```

wire 网线是最常用的网线类型。当多个驱动源驱动同一根 wire 网线时,那些驱动源的输出端就被短接在一起。请看下例:

```
module wireExample (BpW, Error, Wait, Valid, Clear);
  input Error, Wait, Valid, Clear;
  output BpW;
  wire BpW,
  assign BpW = Error & Wait;
  assign BpW = Valid | Clear;
endmodule
// 综合出的网表如图 1-6 所示
```

要采用“或逻辑”和“与逻辑”来实现多个驱动源驱动同一根网线的驱动效果,则分别

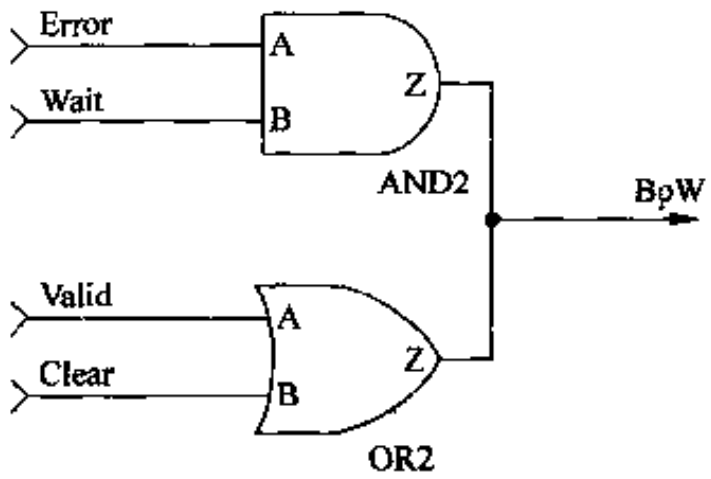

图 1-6　多个驱动源驱动同一根 wire 网线

需要使用 wor 和 wand 网线类型。综合时，驱动同一根 wor 网线的多个驱动源都连接到“或门”上，而驱动同一根 wand 网线的多个驱动源都连接到“与门”上。下例体现了这种综合效果：

```
module UsesGates (BpW, BpR, Error, Wait, Clear);
  input Error, Wait, Clear;
  output BpW, BpR;
  wor BpW;
  wand BpR;

  assign BpW = Error & Wait;
  assign BpW = Valid | Clear;

  assign BpR = Error ^ Valid;
  assign BpR = ! Clear;
endmodule
// 综合出的网表如图 1-7 所示
```

tri 网线与 wire 网线采用相同的方式进行综合。

由 supply0 网线综合出的连线固定连接在 0 电平(逻辑 0)上，而由 supply1 网线综合出的连线固定连接在 1 电平(逻辑 1)上。

寄存器数据类型

能够综合的寄存器[①]类型有：

```
reg     integer
```

① 寄存器类型的变量未必意味着它就是硬件上的一组触发器。请参见后续章节。

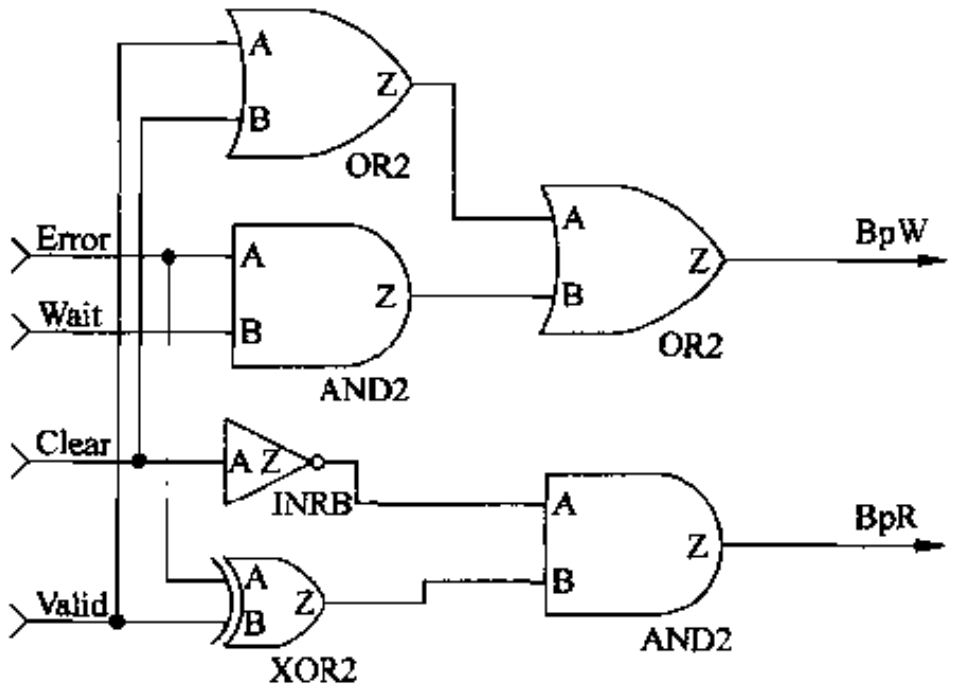

图 1-7 由多个驱动源驱动的 wor 网线和 wand 网线

寄存器声明语句可以明确地指定位宽，即该变量在硬件上相应的位数。例如：

```
reg [1:25] Cpt;          // 25 位的变量
reg Bxr;                 // 1 位的变量
```

如果寄存器声明语句未明确地指定位宽，则默认位宽是 1 位。

对于整数类型，其最大位宽是 32 位，并假定用二进制补码形式来表示。可以选择让综合系统对模型作数据流分析以确定各整型变量实际所需的最大位宽。例如：

```
wire [1:5] Brq, Rbu;
integer Arb;
...
Arb = Brq + Rbu;
```

能确定这里的 *Arb* 位宽是 6 位，因此用一个 6 位的加法器就足够了。*Arb* 最左端的位是进位位。

time 和 real 这两种寄存器类型不能用来综合。

1.4.2 常量

Verilog HDL 中有 3 种类型的常量：整型、实型和字符串型。其中，实型和字符串型的常量不能用来综合。

可用以下两种形式来表示整型常量：

a) 简单的十进制

b) 基数格式

当整数以简单的十进制形式出现，它被解释成有符号数。综合时用 32 位二进制补码形式来表示整型常量。用基数格式表示的整型常量被作为无符号数来处理。如果明确指定了整型常量的位宽，则综合时使用所指定的位宽，否则位宽就是 32 位。请看以下这些示例：

30	32 位有符号数
-2	二进制补码形式的 32 位有符号数
2'b10	位宽为 2
6'd-4	6 位无符号数(用 6 位来表示二进制补码形式下的 -4)
'd-10	32 位无符号数(用 32 位来表示二进制补码形式下的 -10)

1.4.3 参数

参数是命名常量。由于不允许指定参数的位宽，因此其位宽与所对应的常量的位宽相同。

```
parameter RED = -1, GREEN = 2;
parameter READY = 2'b01, BUSY = 2'b11, EXIT = 2'b10;
```

RED 和 *GREEN* 是两个 32 位有符号常量。*READY*、*BUSY* 和 *EXIT* 是 3 个位宽为 2 的参数。

1.5 值保持器的硬件建模

硬件中有 3 种基本的值保持器：

a) 连线

b) 触发器(边沿触发的存储元件)

c) 锁存器(电平敏感的存储元件)

Verilog HDL 中的变量既可以是网线数据类型的，也可以是寄存器数据类型的。综合时，会把网线类型的变量映射成硬件中的连线，而寄存器变量则要根据其被赋值的上下文环境来确定是映射成连线还是映射成存储元件(触发器或锁存器)。下面详细讨论寄存器类型的变量。

在 Verilog HDL 的整个仿真运行过程中，寄存器变量一直保持自己的值，因而把它推导成存储器。但是，这对于综合来说太笼统了。下例中的变量仅仅用作临时变量，因而没有必要将其映射成存储元件。

```
wire Acr, Bar, Fra;      // wire 是一种网线类型
```

```
reg Trq, Sqp;              // reg是一种寄存器类型
...
always @ (Bar or Acr or Fra)
begin①
  Trq = Bar & Acr;
  Sqp = Trq | Fra;
end
```

此例中，第一条语句对变量 *Trq* 进行赋值，然后第二条语句的右端表达式引用了该变量。Verilog HDL 的语义表明在整个仿真运行中 *Trq* 会一直保持它的值。但是，没必要将 *Trq* 的值保存在硬件存储元件中，因为它在赋值之后就立即被引用了。所生成的逻辑电路如图 1-8 所示。

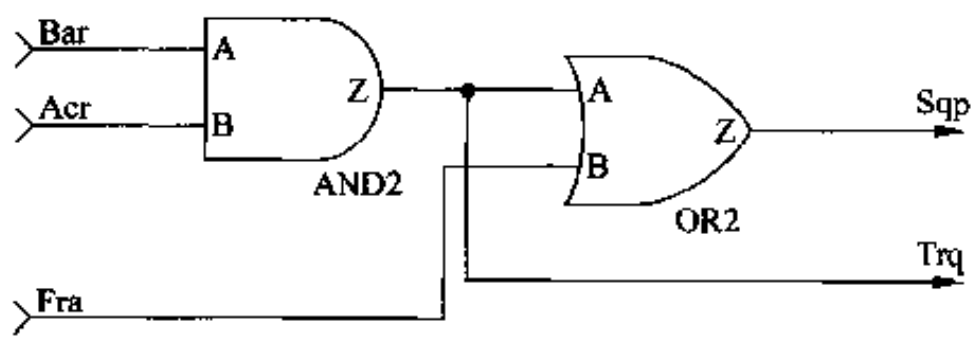

图 1-8 变量 *Trq* 是连线

请看下一个示例。此例中，变量 *Trq* 在赋值前就被引用了。

```
wire Acr, Bar, Fra;
reg Trq, Sqp;
...
always @ (Bar or Acr or Fra)
begin
  Sqp = Trq | Fra;
  Trq = Bar & Acr;
end
```

该 always 语句的 Verilog HDL 语义很明确：只要出现与事件表中列出的 *Bar*、*Acr* 和 *Fra* 相关的事件，就会执行 always 语句。因为 *Trq* 在被赋值之前就被引用了，在重复执行 always 语句的过程中 *Trq* 不得不保持其值，所以它被推导成存储器。不过，这还不足以明确如何把 *Trq* 构造成锁存器，因为 *Trq* 未在任何条件的控制下被赋值。此时，综合系统可能无法为其建立锁存器，而有可能生成如图 1-9 所示的电路。变量 *Trq* 再次被

① **begin...end** 是顺序块；出现在其中的所有语句都是顺序执行的。

综合成网线。但是，为使Verilog HDL 模型与综合出的网表功能一致，*Trq* 也必须放入 always 语句的事件表中。第 5 章将对此作更详细的讨论。

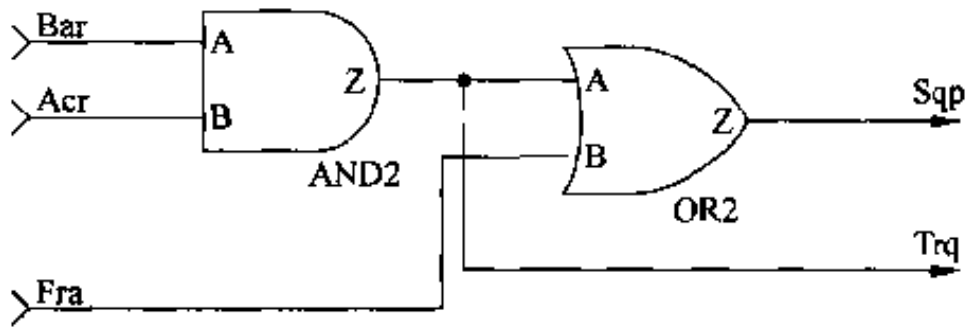

图 1-9　变量 *Trq* 未综合成锁存器

请看以下把变量推导成锁存器的示例：

```
wire Sat, Ant;
reg Fox, Sout;
...
always @ (Sat or Ant)
begin
  if (! Sat)
    Fox = Ant;
  Sout = ! Fox;
end
```

变量 *Fox* 在条件语句的 else 分支中未被赋值。当 *Sat* 为真时，须保持变量 *Fox* 的值，因此它被推导成锁存器。综合出的电路如图 1-10 所示。

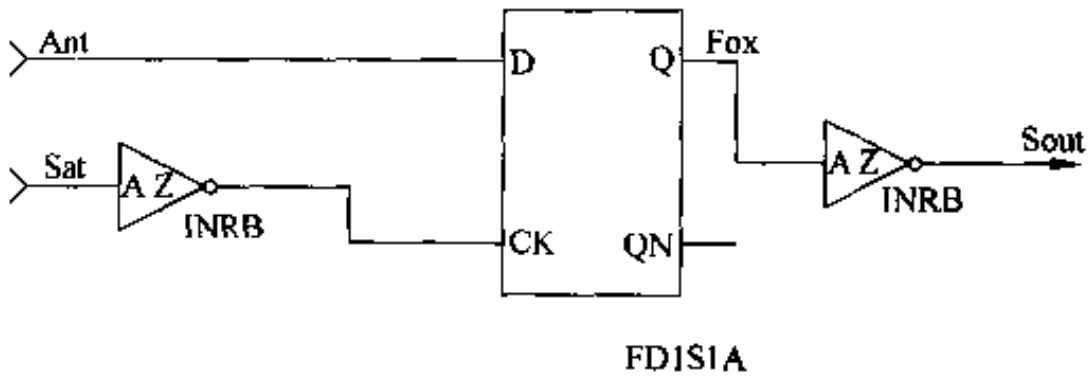

图 1-10　变量 *Fox* 是锁存器

那么如何才会推导出触发器呢？这取决于所采用的建模方式和变量被赋值的上下文环境。第 2 章将根据此示例以及其他一些示例讨论如何推导出触发器和锁存器。硬件中的存储器可以建模成一组触发器或锁存器。

第 2 章　从 Verilog 结构到逻辑门

第 1 章介绍了从 Verilog HDL 的类型和常量到硬件的映射。本章将介绍从 Verilog HDL 语句向硬件逻辑门的映射，还将解释算符、表达式和赋值语句是如何映射成硬件的。为便于阅读和理解，本章将采用食谱的编写方式，每一节介绍一种具体的综合结构或特性。值得注意的是，大部分综合出的网表未经优化，因而并没有实现硬件逻辑的最小化。

2.1　持续赋值语句

在硬件中，持续赋值语句表示用赋值语句右端表达式所推导出的逻辑来驱动该赋值语句左端的网线，而持续赋值的对象总是受组合逻辑驱动的网线。

请看下例：

```
module Continuous (StatIn, StatOut);
  input StatIn;
  output StatOut;

  assign StatOut = ~StatIn;      // 持续赋值
endmodule
// 综合出的网表如图 2-1 所示
```

图 2-1　由持续赋值语句得到的组合逻辑电路

此持续赋值语句描述了一个反相器，其输入端连接到 *StatIn*，输出端是 *StatOut*。通常，即使持续赋值语句中指定了延迟，综合系统也会忽略那些延迟。例如，以下持续赋值语句中的延迟 #2 在综合时会被忽略：

```
assign #2 EffectiveAB = DriverA | DriverB;
```

2.2 过程赋值语句

在硬件中，过程赋值语句表示用赋值语句右端表达式所推导出的逻辑来驱动该赋值语句左端的变量。注意：过程赋值语句只能出现在 always 语句中。①

有两种过程赋值语句：

a) 阻塞式

b) 非阻塞式

2.2.1 阻塞式过程赋值

请看以下阻塞式过程赋值示例：

```
module Blocking (Preset, Count);
  input [0:2] Preset;
  output [3:0] Count;
  reg [3:0] Count;

  always @ (Preset)
    Count = Preset + 1;
    // 阻塞式过程赋值
endmodule
// 综合出的网表如图 2-2 所示
```

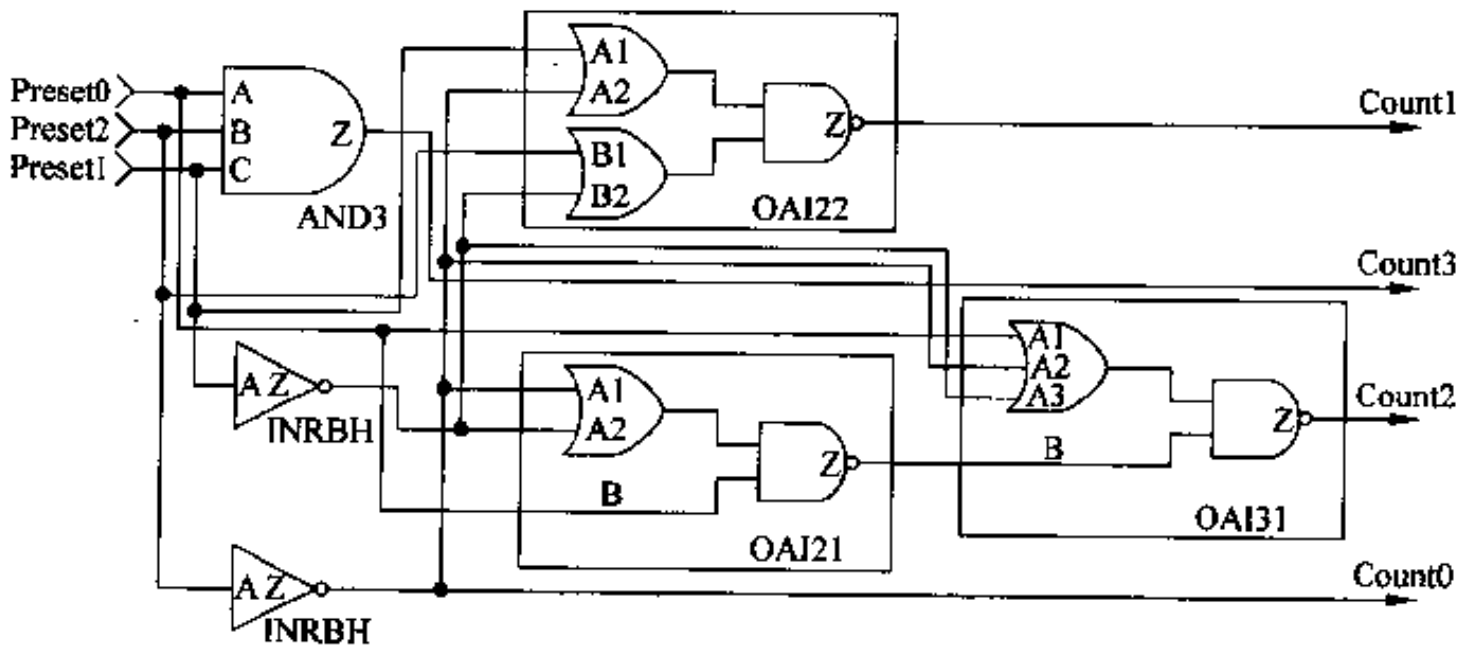

图 2-2 由阻塞式过程赋值得到的组合逻辑电路

① 过程赋值语句也可以出现在初始化语句中，但是初始化语句不支持综合。

此阻塞式过程赋值语句描述了一个加法器，以 *Preset* 和整数 1 为输入，加法结果赋给变量 *Count*。

2.2.2　非阻塞式过程赋值

请看以下非阻塞式过程赋值示例：

```
module NonBlocking (RegA, Mask, RegB);
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;

  always @ (RegA or Mask)
    RegB <= RegA & Mask;
    // 非阻塞式过程赋值
endmodule
// 综合出的网表如图 2-3 所示
```

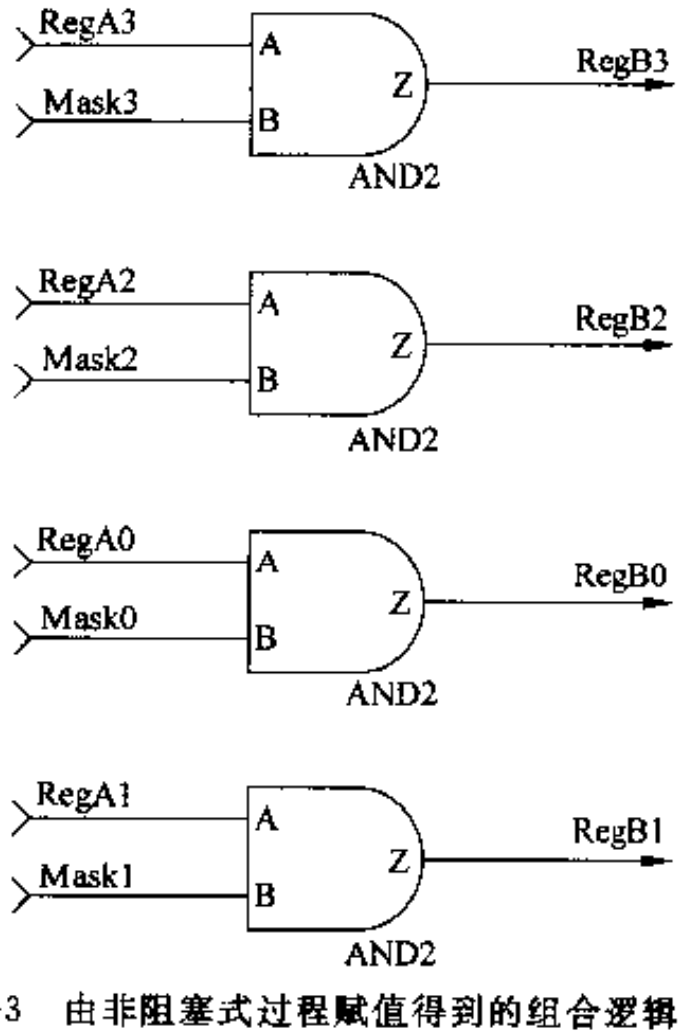

图 2-3　由非阻塞式过程赋值得到的组合逻辑电路

赋值语句的阻塞和非阻塞性质不会造成从赋值语句本身生成的组合逻辑电路有任何不同，但是会影响以后对赋值结果的使用。

由此建议，对组合逻辑进行建模使用阻塞式赋值，而对时序逻辑进行建模使用非阻塞式赋值。后续章节将对此作更多介绍。

在 2.18 节中将会对阻塞式和非阻塞式过程赋值之间的微妙差异作进一步的探讨。

2.2.3　赋值对象

过程赋值的对象会被综合成连线、触发器或锁存器，这取决于赋值语句在 Verilog HDL 模型中的上下文环境。例如，上例中的非阻塞式过程赋值若处于时钟的控制之下，即如下例所示，则该赋值对象就会被综合成触发器。

```
module Target (Clk, RegA, RegB, Mask);
  input Clk;
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;
  always @ (posedge Clk)
    RegB <= RegA & Mask;
endmodule
// 综合出的网表如图 2-4 所示
```

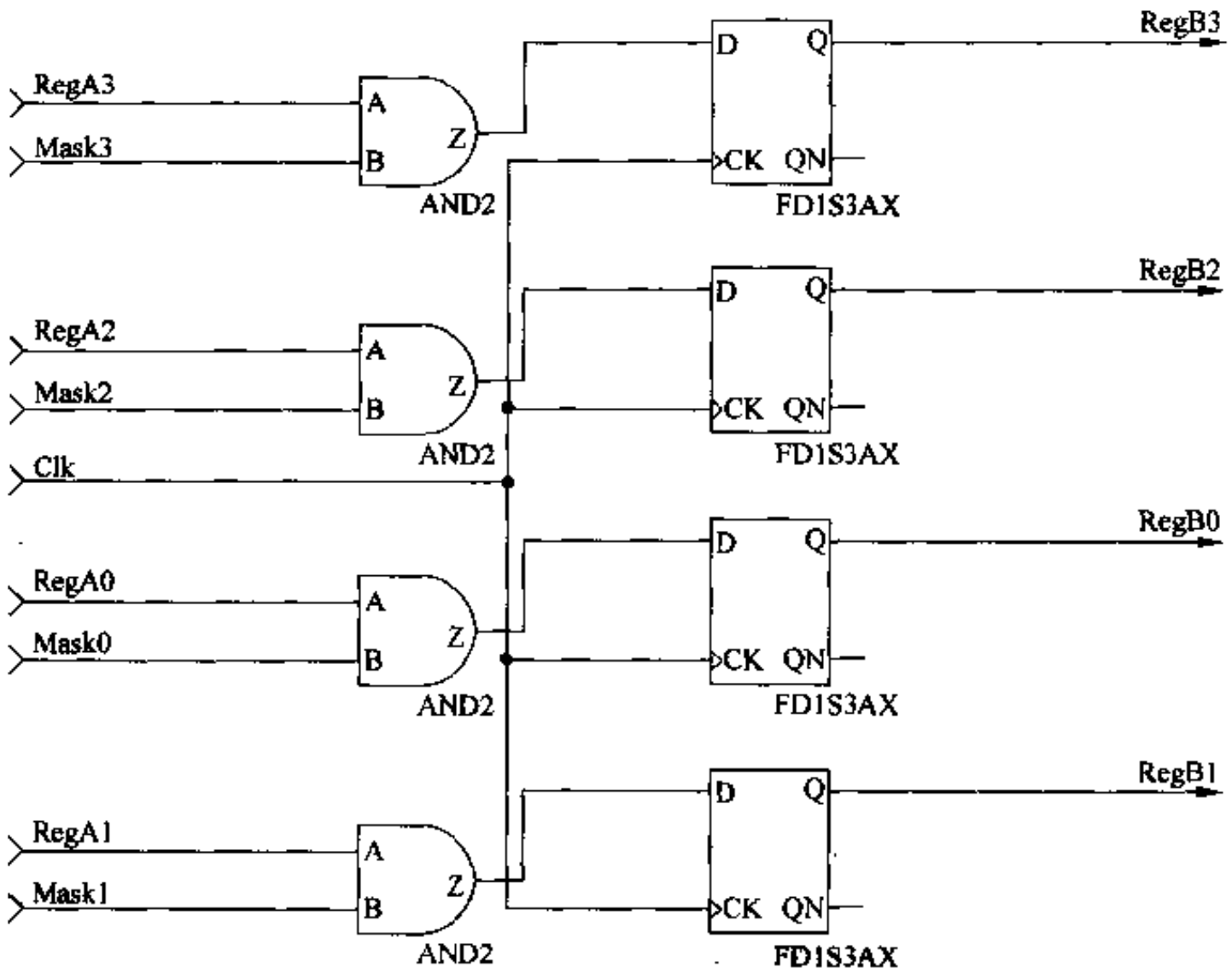

图 2-4 赋值对象是触发器

2.2.4 赋值限制

阻塞式和非阻塞式过程赋值中所指定的任何形式的延迟，包括延迟控制和语句内延迟，都会被综合系统所忽略。这可能会导致设计模型与综合出的网表功能不一致。

```
#5 RegB <= RegA & Mask;
  // 延迟控制 #5 被忽略
RegB = #2 RegA & Mask;
  // 语句内延迟 #2 被忽略
```

在单个用于综合的模型中，对于阻塞式和非阻塞式赋值的使用，还存在另一种限制：**对同一个赋值对象不能既使用阻塞式赋值，又使用非阻塞式赋值。**这意味着，对赋值对象一旦使用了阻塞式(或非阻塞式)赋值，其后对该对象赋值就只能继续使用那种赋值方式了。请看下例：

```
Count = Preset + 1;
```

```
...
Count <= Mask;        // 这是非法的,因为前面对 Count 的赋值是阻塞式赋值
```

2.3 逻 辑 算 符

逻辑算符能直接映射成硬件中的基本逻辑门。以下全加器模型采用了持续赋值：

```
module FullAdder (A, B, CarryIn, Sum, CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;

  assign Sum = (A ^ B) ^ CarryIn;
  assign CarryOut = (A & B) | (B & CarryIn) |
                    (A & CarryIn);
endmodule
// 综合出的网表如图 2-5 所示
```

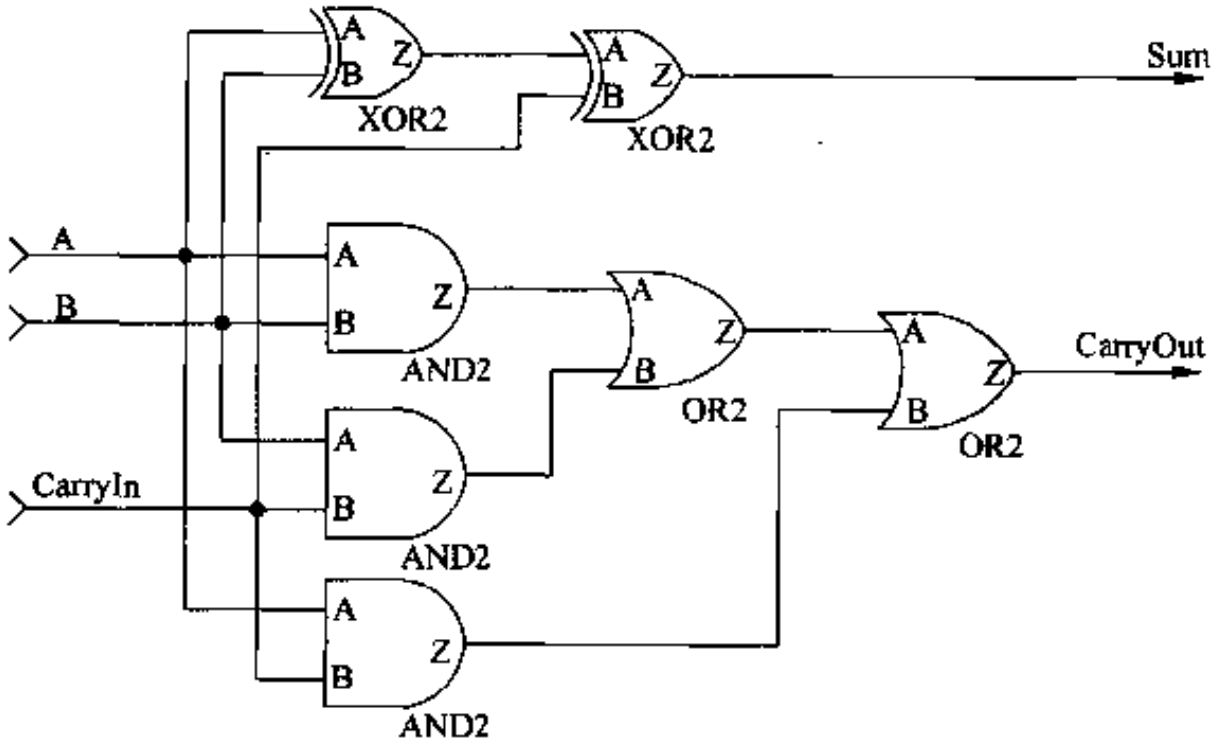

图 2-5　逻辑算符映射成基本逻辑门

2.4 算 术 算 符

在 Verilog HDL 中，寄存器类型被解释成无符号数，整数类型被解释成二进制补码形式的有符号数，解释时都把最右端的位作为最低有效位。因此，要综合成无符号算术算

符就需要使用寄存器类型，而要得到有符号算术算符就需要使用整型。

网线类型被解释为无符号数。

2.4.1 无符号算术

下例对无符号数使用了算术算符：

```
module UnsignedAdder (Arb, Bet, Lot);
  input [2:0] Arb, Bet;
  output [2:0] Lot;

  assign Lot = Arb + Bet;
endmodule
// 综合出的网表如图 2-6 所示
```

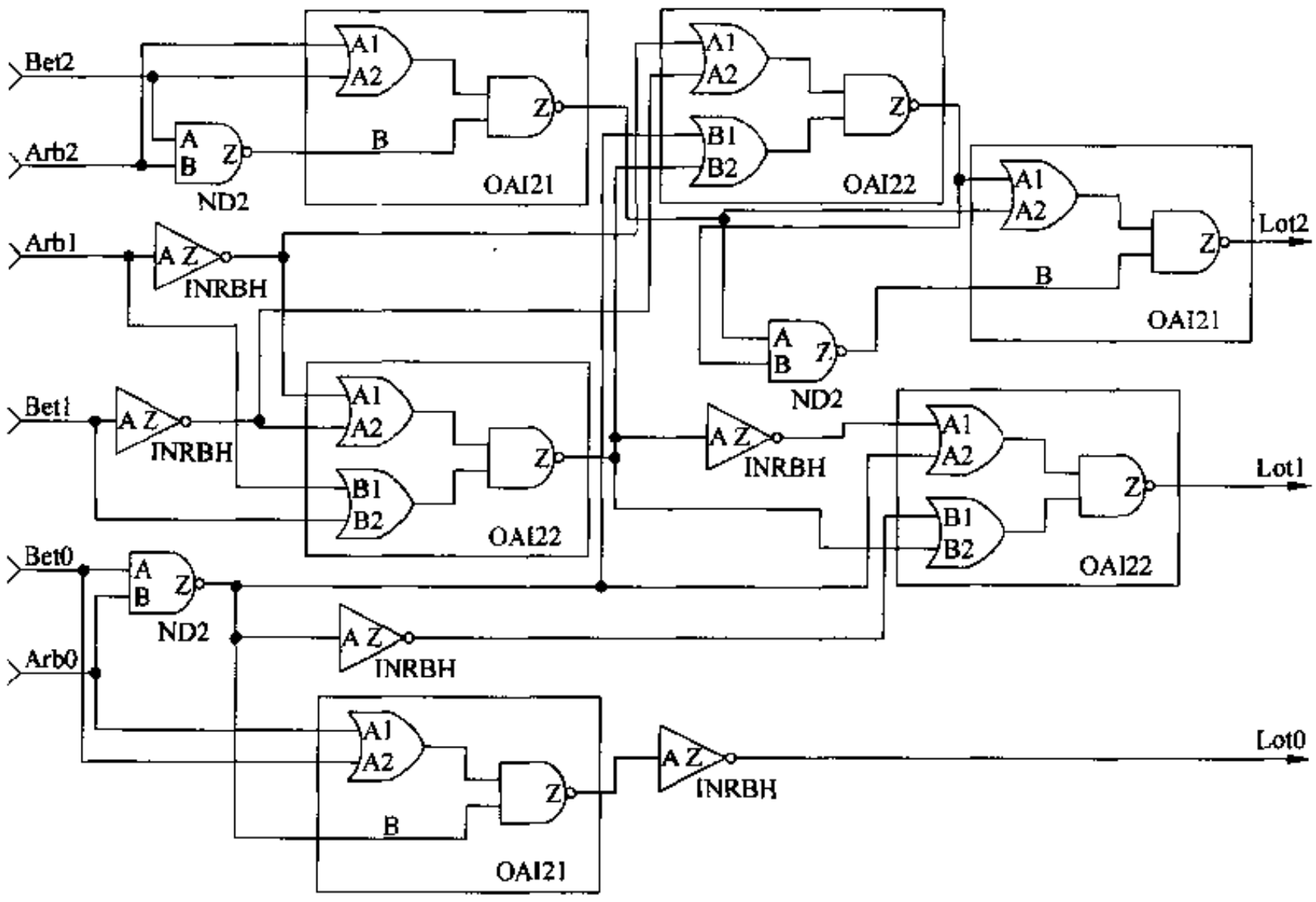

图 2-6 3 位加法器

此示例是对 3 位加法器进行建模。运算量都是无符号的，因为它们都是网线型的。它们最左端的位是最高有效位。

2.4.2　有符号算术

下例中运算量是有符号数，这是通过使用整型来实现的。

```
module SignedAdder (Arb, Bet, Lot);
  input [1:0] Arb, Bet;
  output [2:0] Lot;
  reg [2:0] Lot;
  always @ (Arb or Bet)
  begin: LABEL_A
    // 内有局部声明的顺序块应使用标号
    integer ArbInt, BetInt;
    ArbInt =  - Arb;      // 保存负数仅用于表明对有符号运算量作“+”运算
    BetInt = Bet;
    Lot = ArbInt + BetInt;
  end
endmodule
// 综合出的网表如图 2-7 所示
```

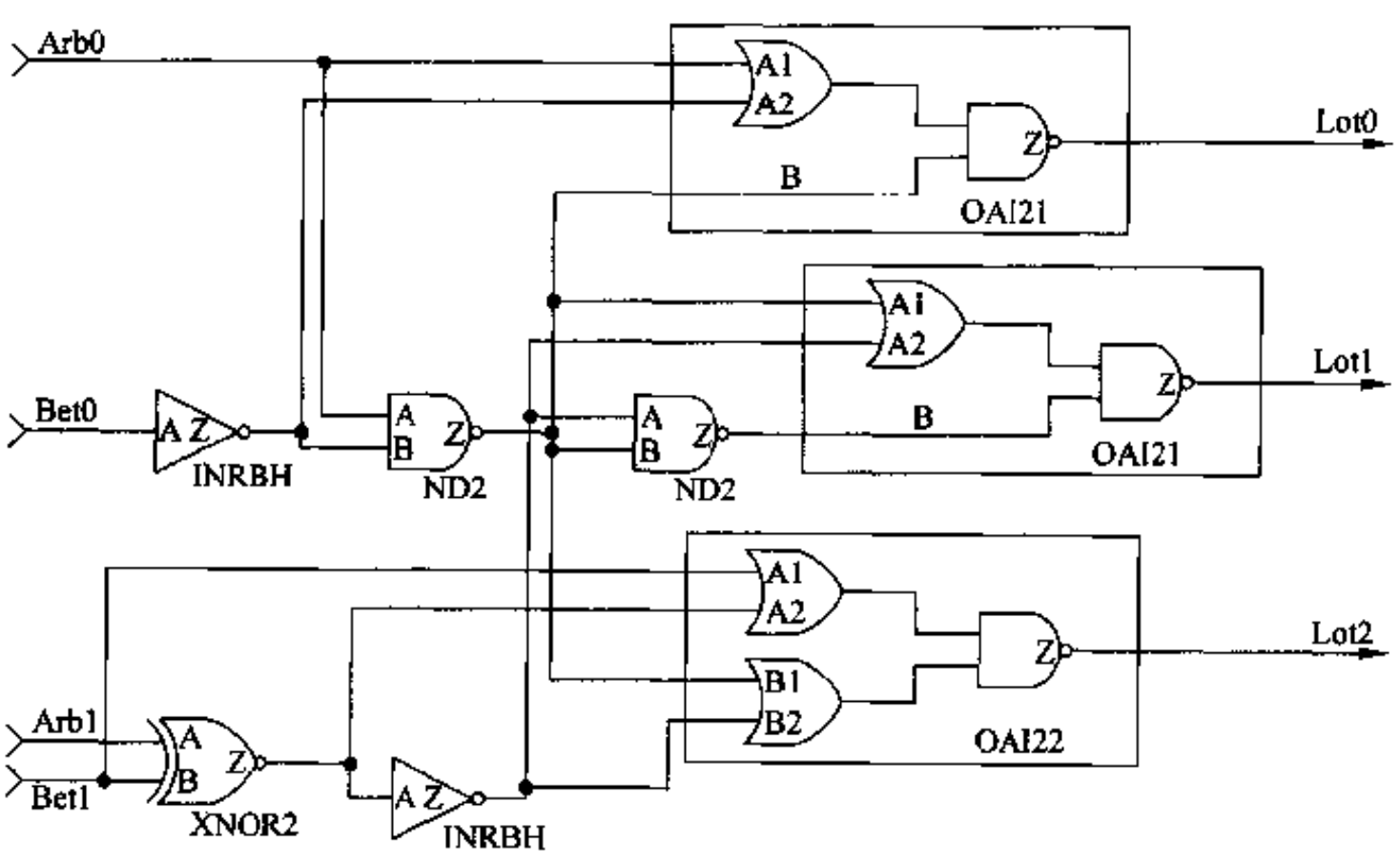

图 2-7　有符号加法器

注意：有符号运算量的加法器逻辑与无符号运算量的加法器逻辑相同，因为有符号数是用二进制补码形式来表示的。

2.4.3 进位的建模

对进位进行建模，通常的方法是让结果的位宽比两个运算量中位宽较大的那个多一位。另一种可行的方法是，加法结果拼接明确指定的进位位之后再用作赋值对象。请看这两种方法的示例：

```
wire [3:0] CdoBus, Sum;
wire [4:0] OneUp;
wire Bore;
...
assign OneUp = CdoBus + 1;
assign (Bore, Sum) = CdoBus - 2;
```

在第一个持续赋值语句中，运算结果为 5 位，其中 *OneUp*[4]是进位位。如果 *OneUp* 被声明成

```
wire [3:0] OneUp;
```

则会丢失进位位。第二个持续赋值语句中，*Bore* 就是减法运算的借位位。

2.5 关 系 算 符

能够综合的关系算符有：>、<、<=、>=。关系算符的建模与算术算符相似。对于关系算符，综合会产生不同的结果，这取决于被比较的究竟是无符号数还是有符号数。如果比较的是寄存器类型或者网线类型的变量，则综合出无符号关系算符。如果比较的是整型变量，则综合出有符号关系算符。以下关系算符示例中使用的运算量都是无符号数：

```
module GreaterThan (A, B, Z);
  input [3:0] A, B;
  output Z;

  assign Z = A[1:0] > B[3:2];
  // 变量 A 和 B 是网线类型的
endmodule
// 综合出的网表如图 2-8 所示
```

请看以下综合有符号关系算符的示例。此例中，关系算符的运算量是整型变量。

```
module LessThanEquals (ArgA, ArgB, ResultZ);
```

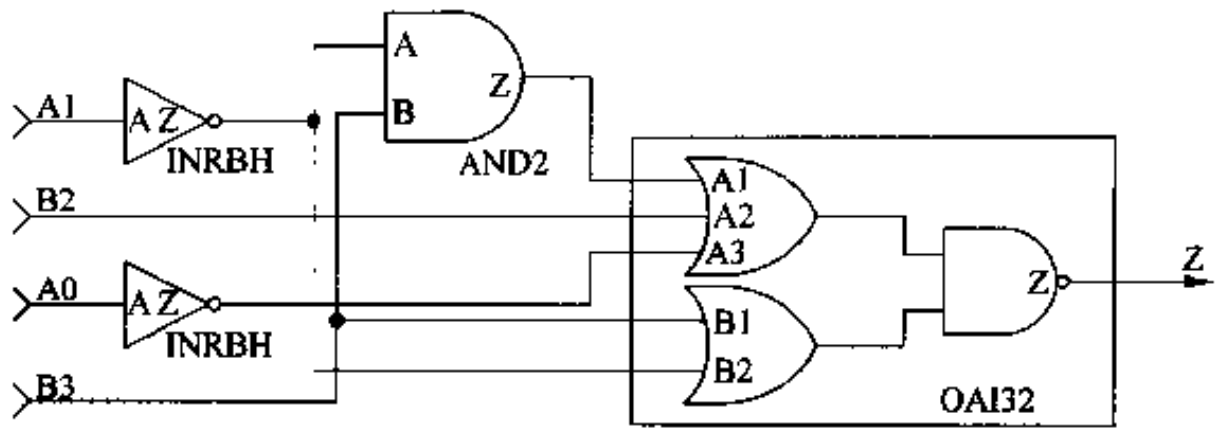

图 2-8　无符号关系算符“ > ”

```
input [2:0] ArgA, ArgB;
output ResultZ;
reg ResultZ;
integer ArgAInt, ArgBInt;

always @ (ArgA or ArgB)
begin
  ArgAInt = -ArgA;
  ArgBInt = -ArgB;
  // 保存负值仅用于表明对有符号数进行比较
  ResultZ = ArgAInt <= ArgBInt;
end
endmodule
// 综合出的网表如图 2-9 所示
```

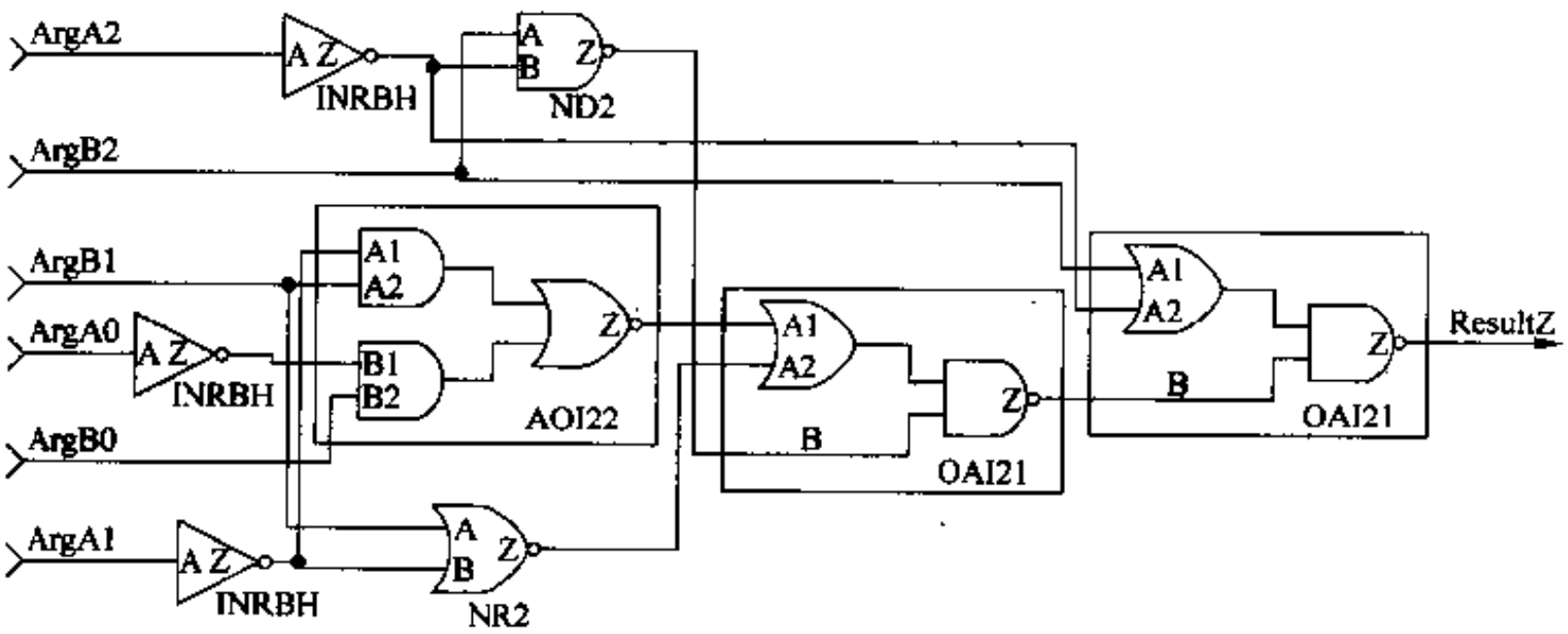

图 2-9　有符号关系算符“ <= ”

2.6 相等性算符

能够综合的相等性算符有：== 和 != 。而全等(case equality)算符 === 和非全等(case inequality)算符 !== 不能用于综合。

就所作的比较究竟是有符号比较还是无符号比较而言，相等性算符的建模方式类似于算术算符。下例中使用了有符号数。注意，此例中，相等性算符的运算量都是整型的，因为整型值表示的是有符号数。

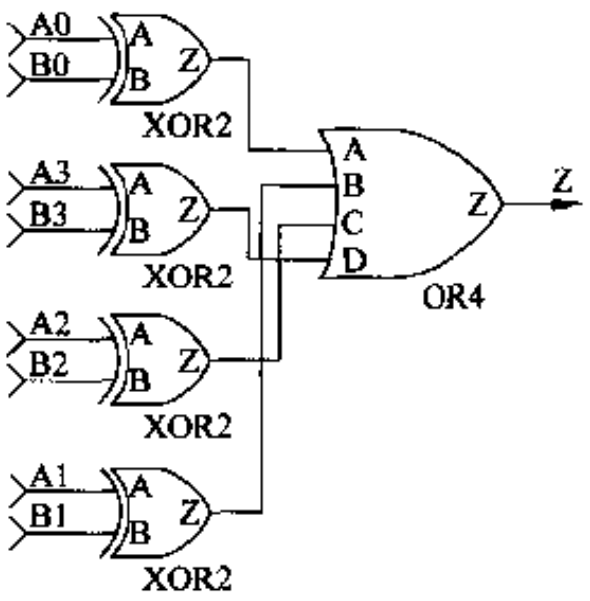

图 2-10 有符号关系算符“!=”

```
module NotEquals (A, B, Z);
  input [0:3] A, B;
  output Z;
  reg Z;

  always @ (A or B)
  begin: DF_LABEL
    integer IntA, IntB;

    IntA = A;
    IntB = B;
    Z = IntA != IntB;
  end
endmodule
// 综合出的网表如图 2-10 所示
```

2.7 移 位 算 符

Verilog HDL 综合系统支持左移(“<<”)和右移(“>>”)算符。移位腾出的各位都补 0。算符右边的运算量表示移位的位数，它既可以是常量，也可以是变量。在这两种情况下，产生的都是组合逻辑。如果位移量是常量，则只需重新连线就行了。如果位移量是变量，则会综合出通用移位器。请看以下两例。

```
module ConstantShift (DataMux, Address);
  input [0:3] DataMux;
  output [0:5] Address;
```

```
  assign Address = (~DataMux) << 2;
endmodule
// 综合出的网表如图 2-11 所示
```

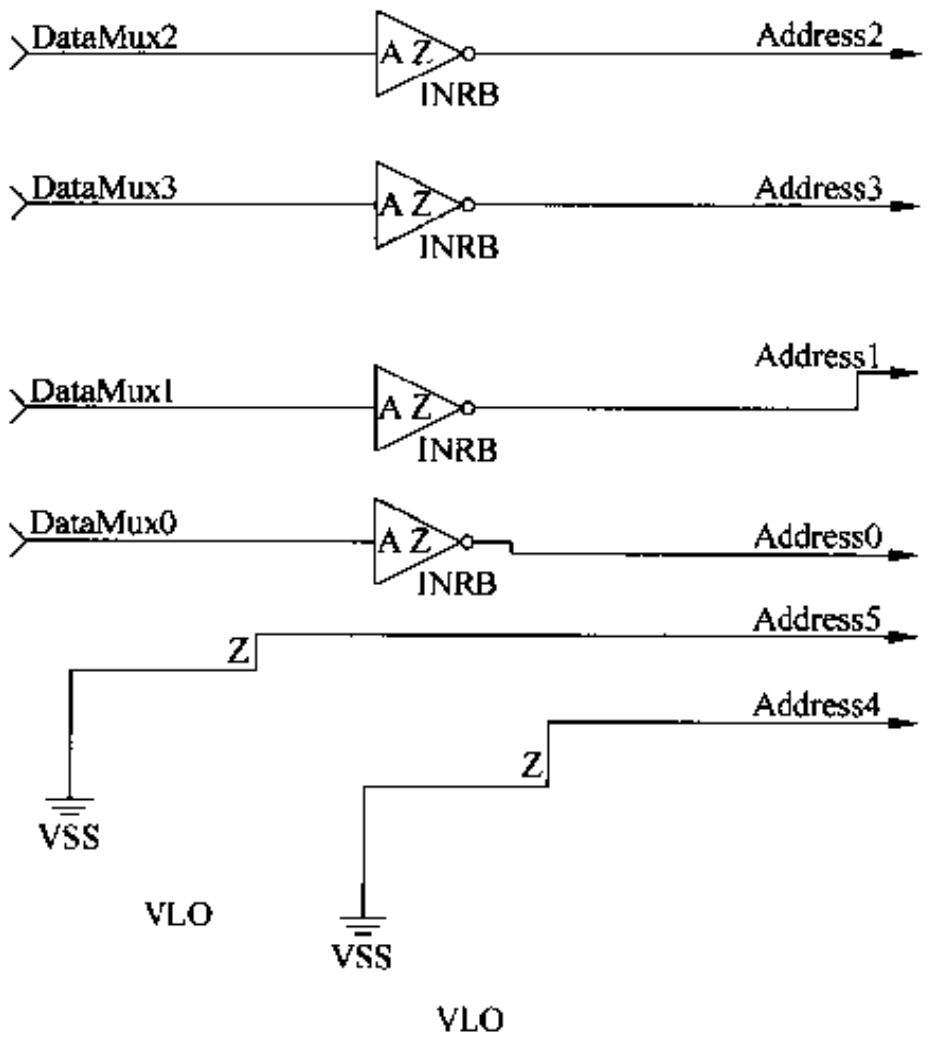

图 2-11　位移量是常量

```
module VariableShift (MemDataReg, Amount, InstrReg);
  input [0:2] MemDataReg;
  input [0:1] Amount;
  output [0:2] InstrReg;

  assign InstrReg = MemDataReg >> Amount;
endmodule
// 综合出的网表如图 2-12 所示
```

根据 Verilog HDL 规则，*ConstantShift* 模块执行左移运算时，从 *DataMux* 中移出的位并没有被舍弃，而是被移到 *Address* 的更高位上去了。如果 *Address* 和 *DataMux* 的位宽相同，则高位将被移走并被舍弃。

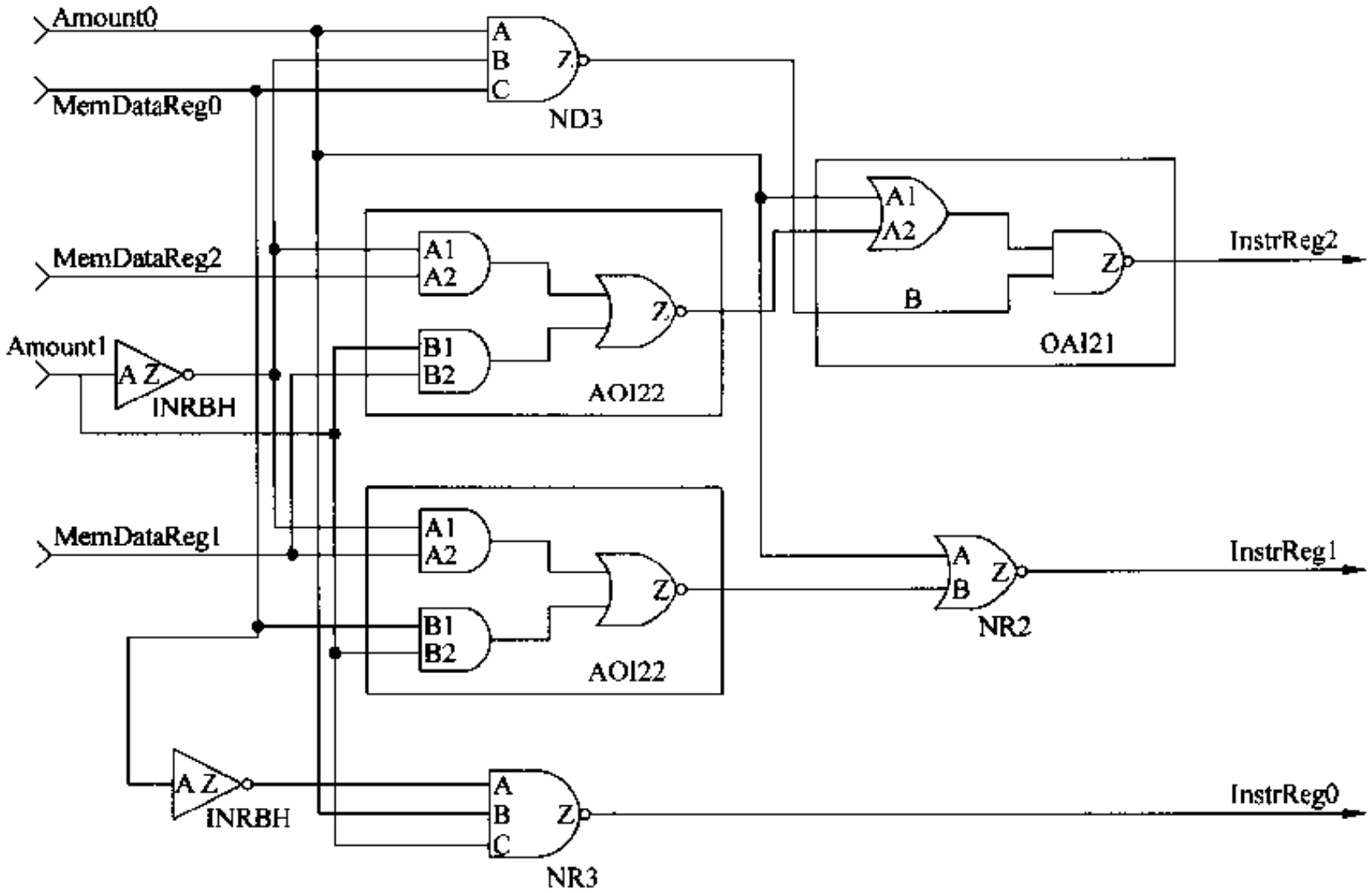

图 2-12　位移量是变量

2.8 向 量 运 算

下例表明表达式中可以使用向量运算量。A 的 4 个位与 B 的 4 个位逐位相与的结果再同 C 的 4 个位逐位相或。运算结果被赋给(从最右位开始赋值)目标网线 $RFile$。

```
module VectorOperations (A, B, C, RFile);
  input [3:0] A, B, C;
  output [3:0] RFile;

  assign RFile = (A & B) | C;
endmodule
// 综合出的网表如图 2-13 所示
```

请再看一个示例,其中逻辑算符的运算量是向量。此时,针对向量的各个位生成了一系列逻辑门。

```
module VectorOperands (Bi, Stdy, Tap);
  input [0:3] Bi, Stdy;
```

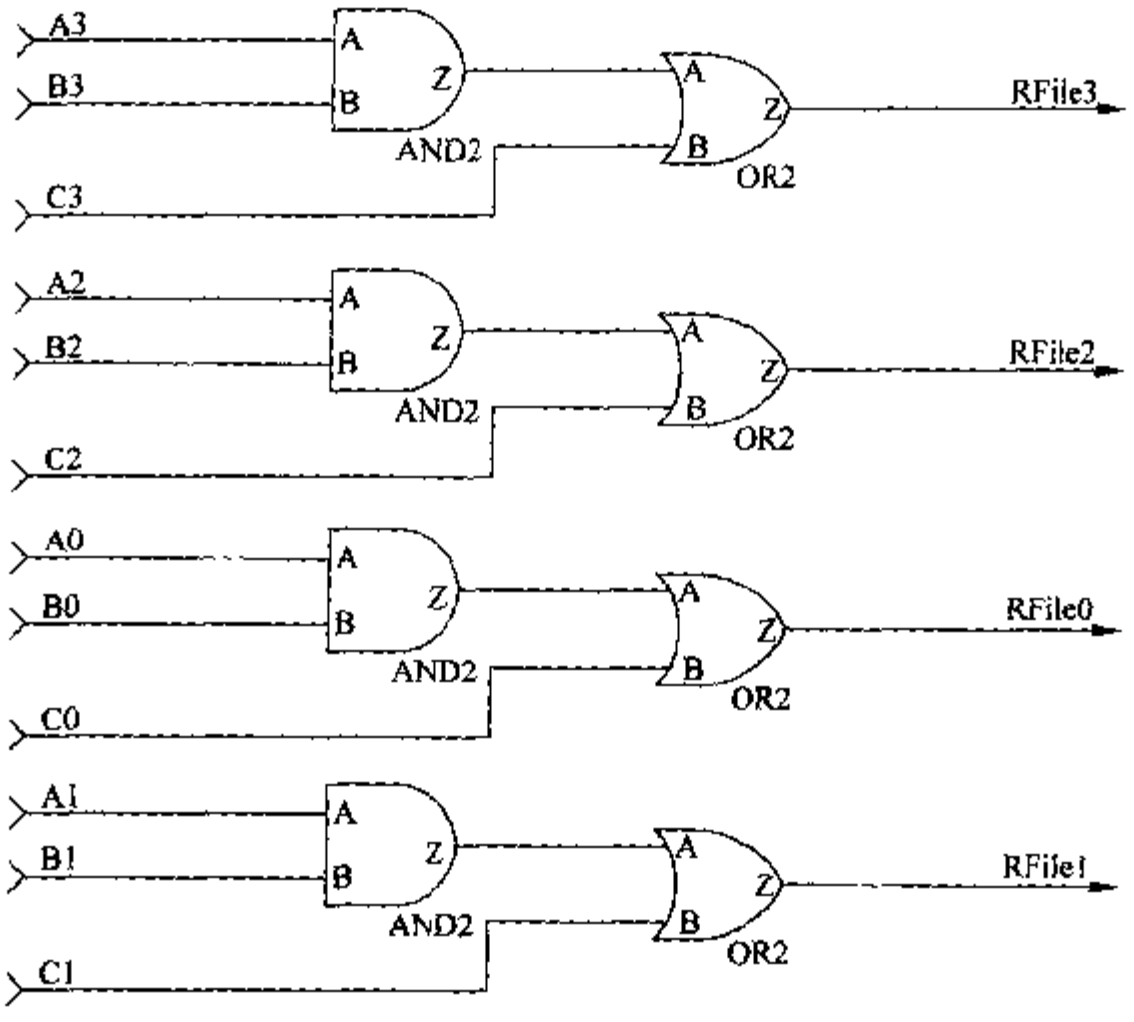

图 2-13　向量运算

```
    output [0:3] Tap;

    assign Tap = Bi ^ Stdy;
endmodule
// 综合出的网表如图 2-14 所示
```

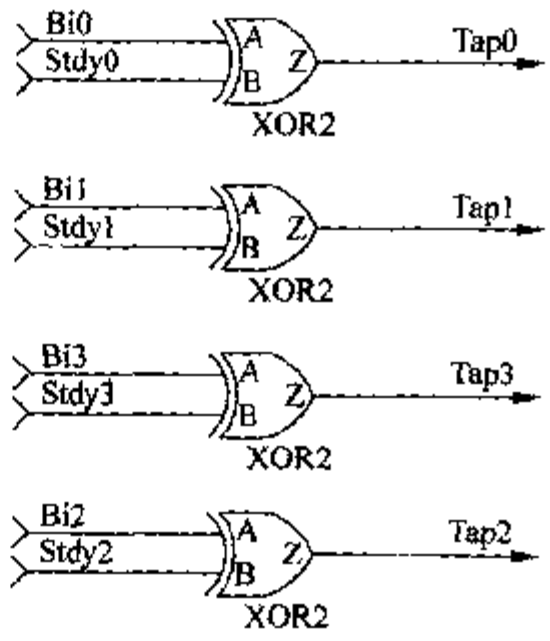

图 2-14　一组逻辑门

因为右端每一个运算量都是 4 位的，所以综合出 4 个异或门。

以上这些持续赋值示例中，持续赋值语句与综合出的逻辑电路之间存在着一一对应关系。这是因为持续赋值语句隐式地描述了这种电路结构。

2.9　部 分 选 取

模型中的运算可以使用部分选取。请看下例：

```
module PartSelect (A, C, ZCat);
```

```
    input [3:0] A, C;
    output [3:0] ZCat;

    assign ZCat [2:0] = {A [2], C [3:2]};
endmodule
// 综合出的网表如图 2-15 所示
```

其中 *ZCat*[2：0]和 C[3：2]就是部分选取。Verilog HDL 不支持非常量的部分选取。

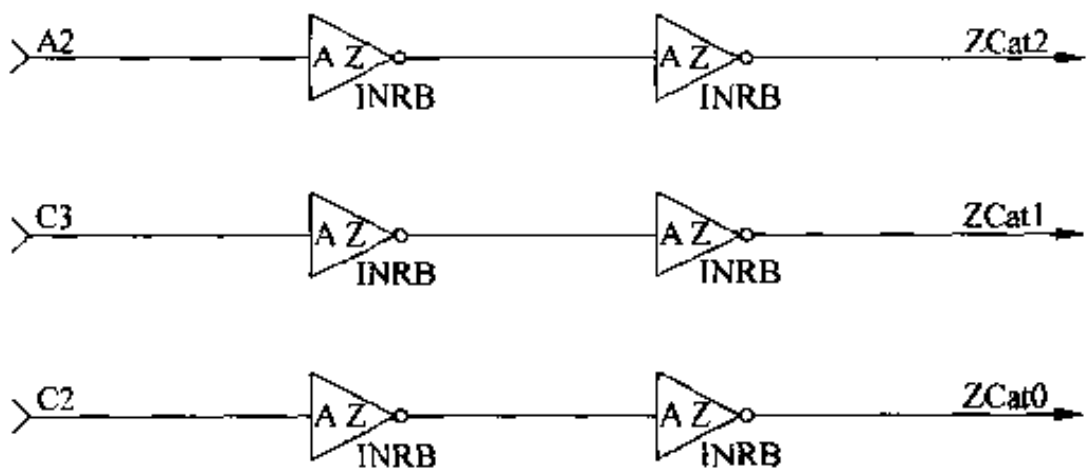

图 2-15　部分选取示例

2.10　位　选　取

位选取可以采用常量下标或者非常量下标。

2.10.1　常量下标

下例中，位选取下标使用了常量值。

```
module ConstantIndex (A, C, Reg_File, ZCat);
    input [3:0] A, C;
    input [3:0] Reg_File;
    output [3:0] ZCat;

    assign ZCat[3:1] = {A[2], C [3:2]};
    assign ZCat[0] = Reg_File[3];
endmodule
// 综合出的网表如图 2-16 所示
```

其中 *A*[2]、*ZCat*[0]和 *Reg_file*[3]就是位选取。拼接算符“{}”用于生成更长的数组。

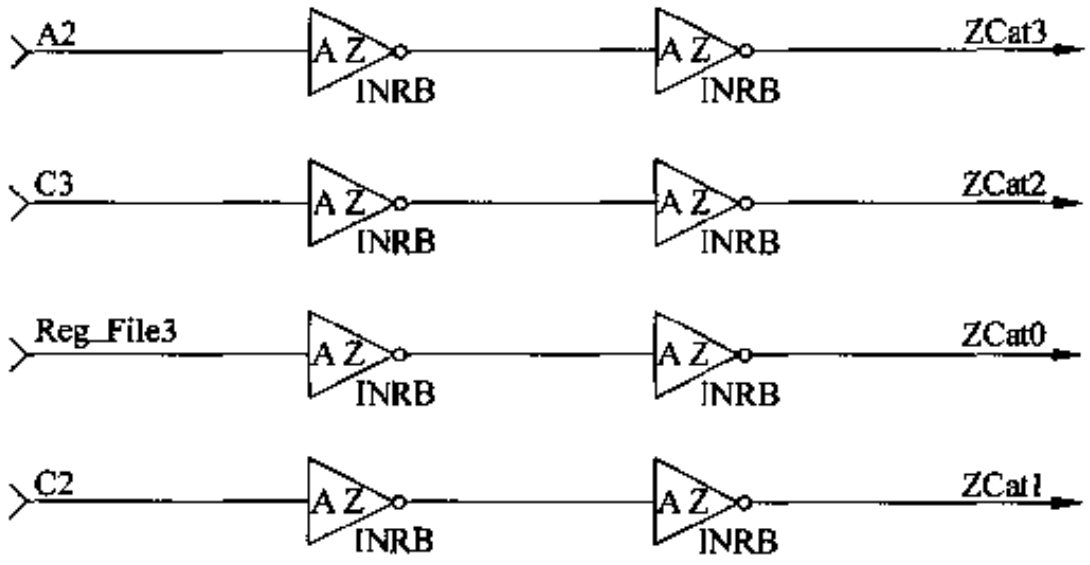

图 2-16　采用常量下标的位选取

2.10.2　表达式中的非常量下标

以下模型表明位选取可以采用非常量下标。

```
module NonComputeRight (Data, Index, Dout);
  input [0:3] Data;
  input [1:2] Index;
  output Dout;

  assign Dout = Data [Index];
endmodule
// 综合出的网表如图 2-17 所示
```

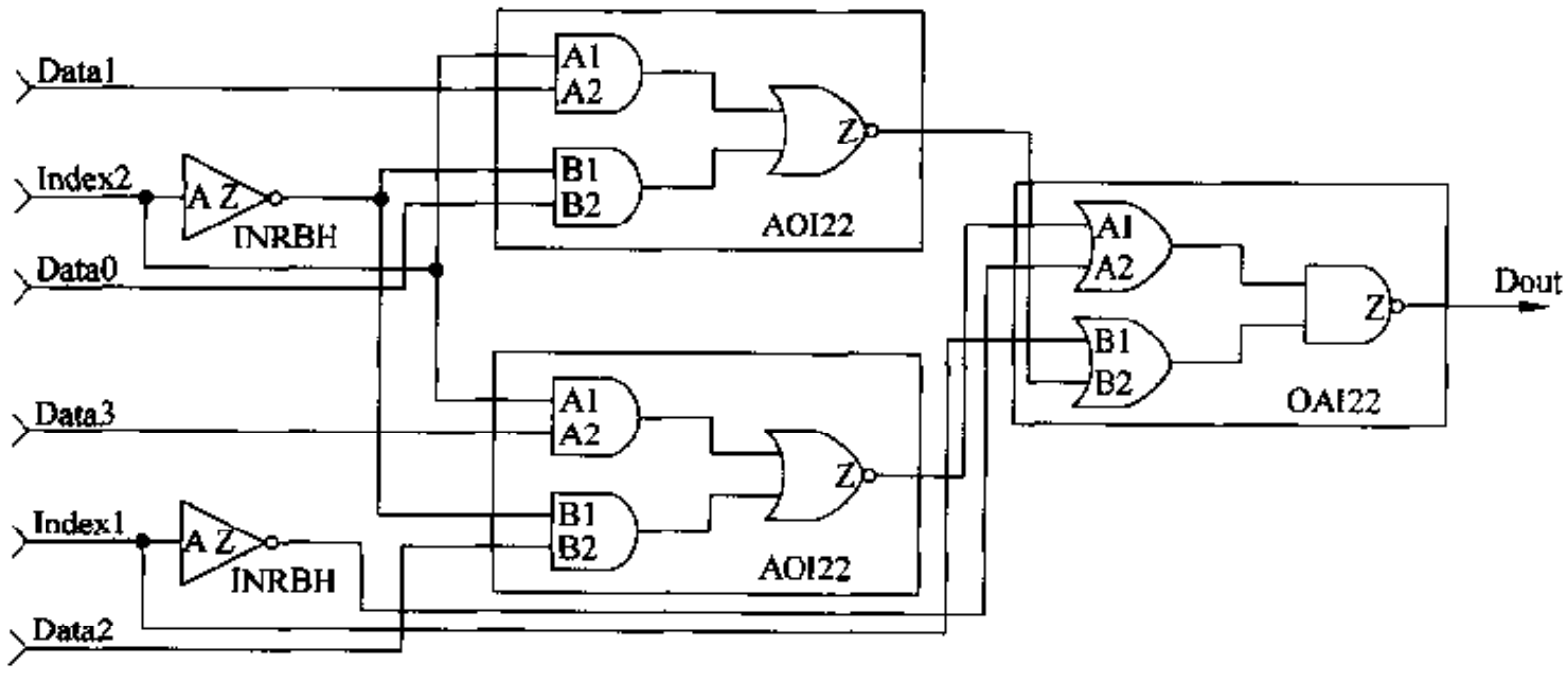

图 2-17　采用非常量下标的位选取产生多路选择器

在本例中，如综合出的网表所示，生成了多路选择器。

2.10.3 赋值对象中的非常量下标

请再看以下非常量位选取示例。此时位选取应用于赋值语句左端，因此这种行为被综合成译码器。

```
module NonComputeLeft (Mem, Store, Addr);
  output [7:0]Mem;
  input Store;
  iutput [1:3]Addr;
```

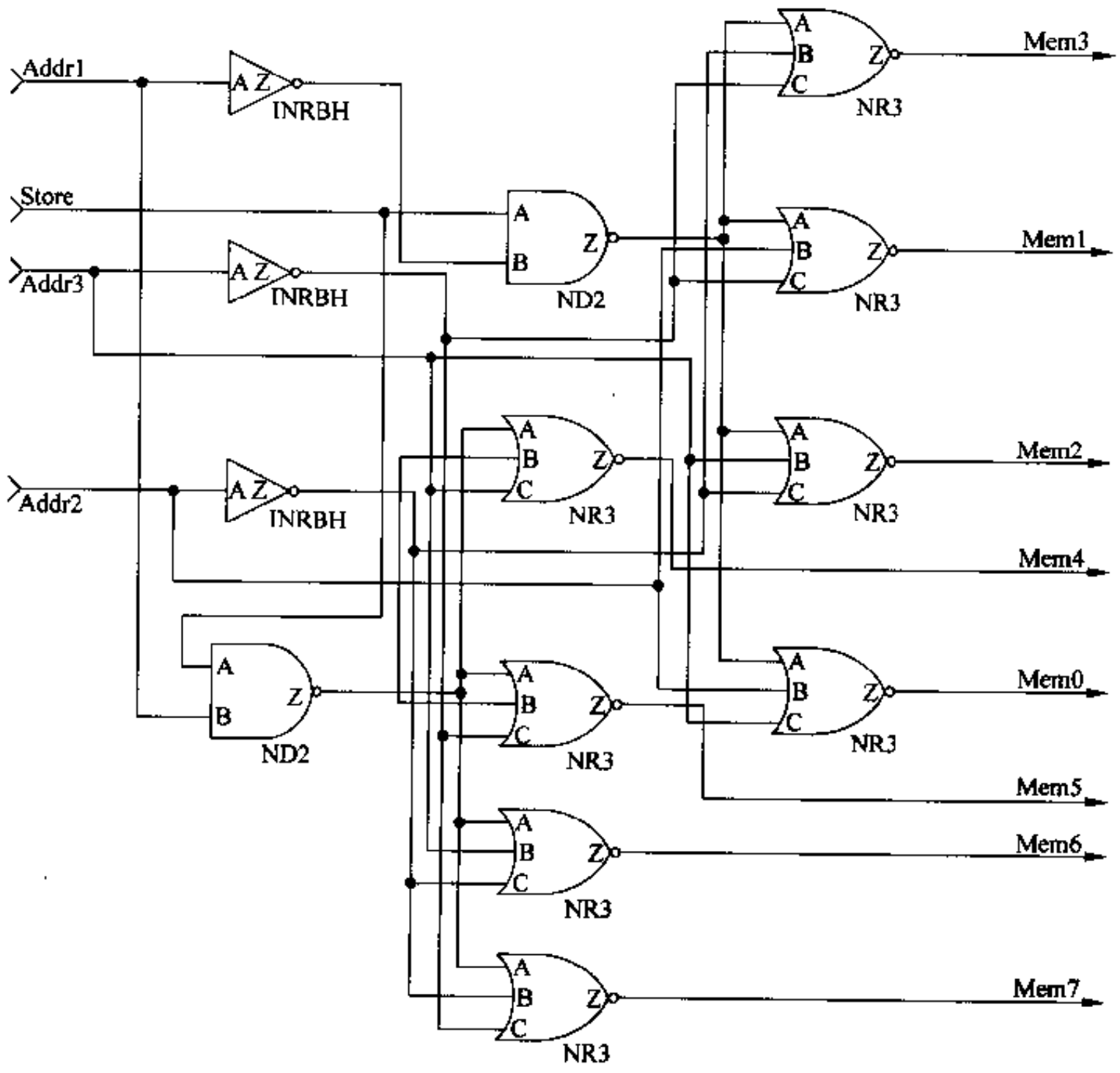

图 2-18 从非常量下标的位选取生成的译码器

```
  assign Mem [Addr] = Store;
endmodule
// 综合出的网表如图 2-18 所示
```

2.11　条件表达式

条件表达式根据条件值在两个表达式中加以选择。

<条件>？<表达式 1>：<表达式 2>

如果条件为真，则选择第一个表达式，否则选择第二个表达式。请看下例：

```
module ConditionalExpression (StartXM, ShiftVal,
                              Reset, StopXM);
  input StartXM,ShiftVal, Reset;
  output StopXM;
  assign StopXM = (! Reset) ? StartXM ^ ShiftVal :
                              StartXM | ShiftVal;
endmodule
// 综合出的网表如图 2-19 所示
```

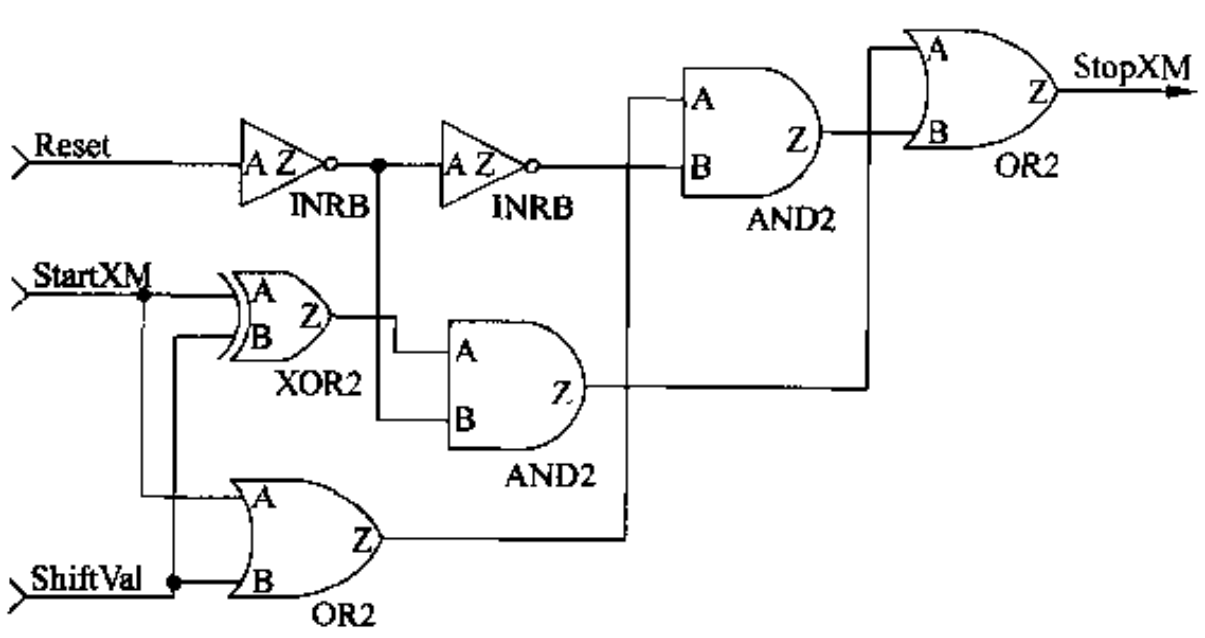

图 2-19　条件表达式生成的逻辑

2.12　always　语　句

always 语句用于电路过程行为的建模。下例中 always 语句内含有过程赋值语句。

```
module EvenParity (A, B, C, D, Z);
```

```
    input A, B, C, D;
    output Z;
    reg Z, Temp1, Temp2;

    always @ (A or B or C or D)
    begin
      Temp1 = A ^ B;
      Temp2 = C ^ D;
      Z = Temp1 ^ Temp2;
      // 注意,这里并不真正需要临时量。这里使用它们是
      // 为了解释顺序块中各语句的顺序行为。
    end
endmodule
// 综合出的网表如图 2-20 所示
```

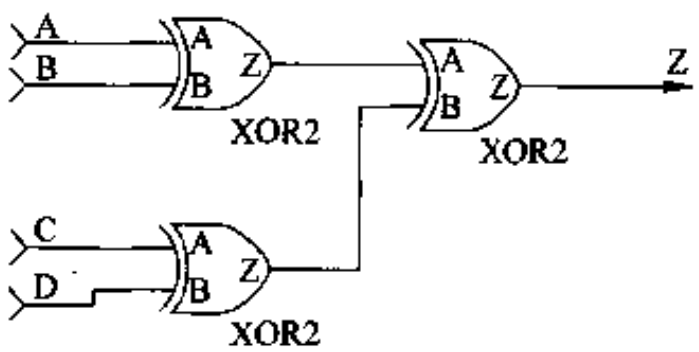

图 2-20　过程赋值语句

always 语句中读取的变量都必须出现在事件表(跟在符号"@"之后用括号括起来的表)中；否则综合出的网表可能与设计模型功能不一致。以下的简单示例说明了这一点。

```
module AndBehavior (Z, A, B);
  input A, B;
  output Z;
  reg Z;

  always @ (B)
    Z = A & B;
endmodule
// 综合出的网表如图 2-21 所示
```

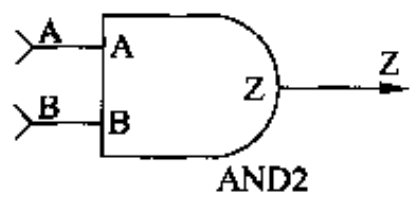

图 2-21　不完整的事件表

此 always 语句的语义表明：无论何时只要 B 端发生事件，就执行赋值语句使 Z 获得相应的值。而 A 端发生任何事件，都不会对 Z 的值产生影响。但是，由上面这个模型综合出的网表(如图 2-21 所示)是一个与门，这样任何时候 A 或者 B 发生变化，Z 的值都会即刻

更新，因此就出现了功能不一致。综合系统通常会对此发出一个警告，以指出事件表中遗漏了变量。

为防止出现这种问题，较好的做法是：always 语句事件表中应包括所有所读取的变量。不过这种做法仅对组合逻辑建模有效，而对时序逻辑建模，则需要另一种类型的事件表，后文将对此加以介绍。

如下例所示，always 内声明的变量用于存放临时值，因而无须将该变量推导成单独的硬件连线。

```
module VariablesAreTemporaries (A, B, C, D, Z);
  input A, B, C, D;
  output Z;
  reg Z;

  always @ (A or B or C or D)
  begin:VAR_LABEL
    integer T1, T2;

    T1 = A & B;
    T2 = C & D;
    T1 = T1 | T2;
    Z = ~T1;
    end
endmodule
// 综合出的网表如图 2-22 所示
```

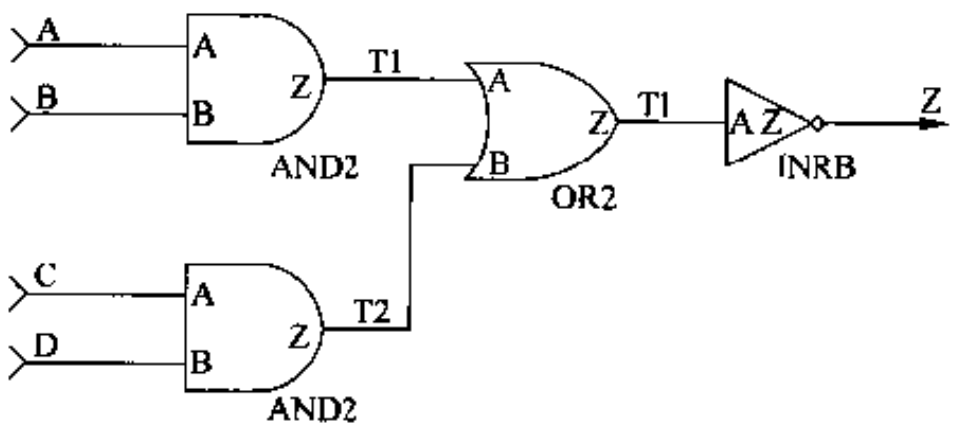

图 2-22　一个变量能表示多根连线

在此综合出的网表中，门 *AND2* 的输出就是变量 *T1*，因此门 *OR2* 的输出也是变量 *T1*。此例中，对整型变量的每一次赋值都被推导成单独的一根连线。

2.13 if 语 句

if 语句表示受条件控制的逻辑。请看下例：

```
module SelectOneOf (A, B, Z);
input [1:0] A, B;
output [1:0] Z;
reg [1:0]Z;

always @ (A or B)
  if (A > B)
    Z = A;
  else
    Z = B;
endmodule
// 综合出的网表如图 2-23 所示
```

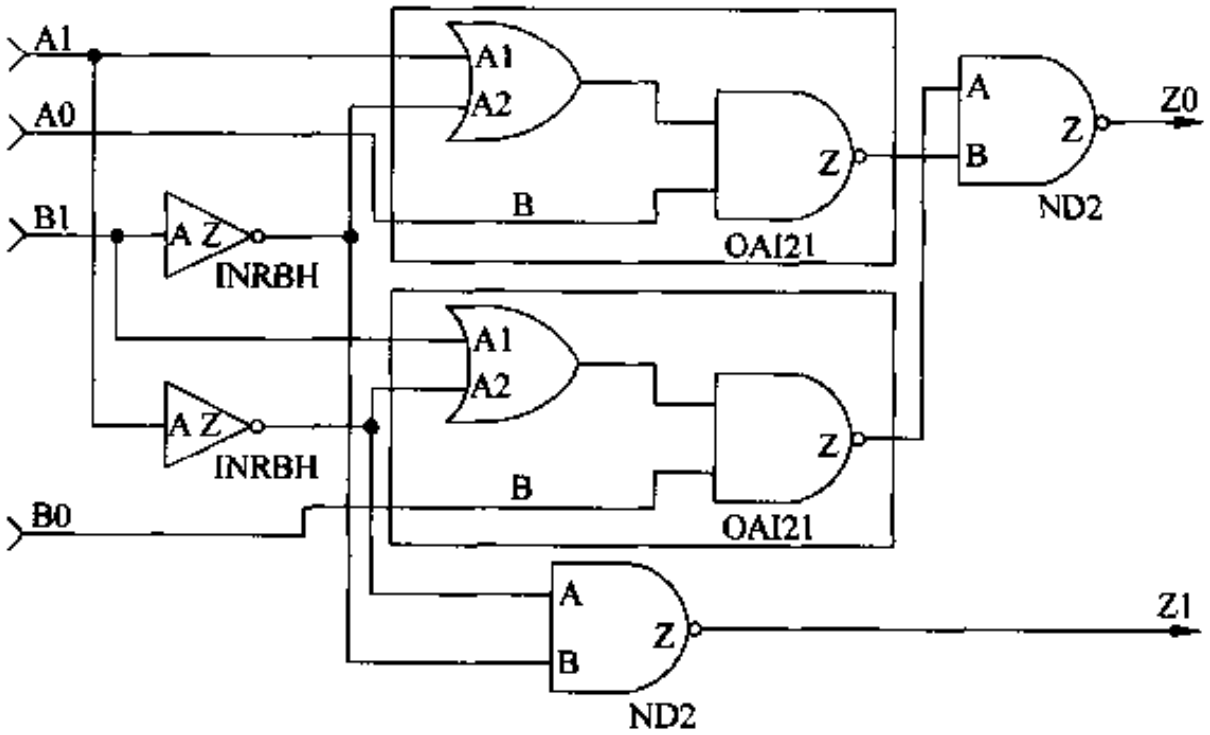

图 2-23 从 if 语句推导出的逻辑

请再看一个 if 语句示例：

```
module SimpleALU (Ctrl, A, B, Z);
  input Ctrl;
  input [0:1] A, B;
  output [0:1] Z;
```

```
  reg [0:1] Z;

  always @ (Ctrl or A or B)
    if (Ctrl)
      Z = A & B;
    else
      Z = A | B;
endmodule
// 综合出的网表如图 2-24 所示
```

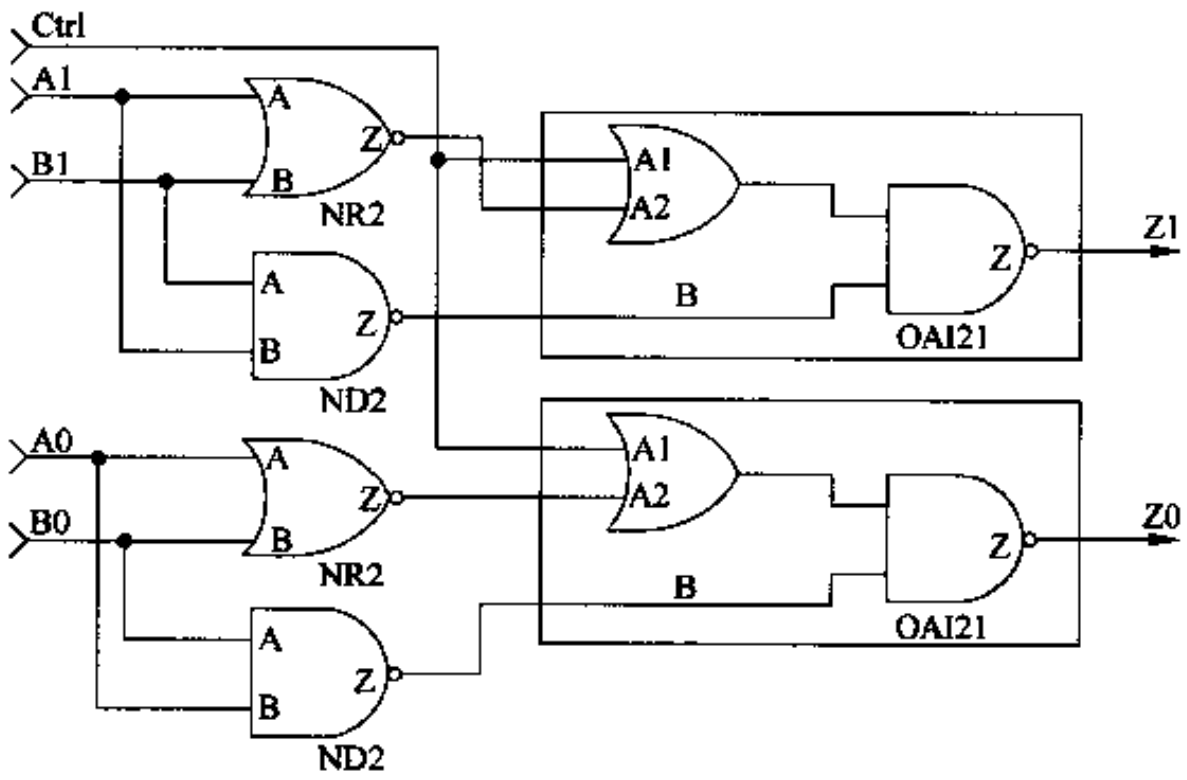

图 2-24　运算的条件选择

2.13.1　从 if 语句推导出锁存器

请考虑以下模块中的 always 语句。

```
module Increment (Phy, Ones, Z);
  input Phy;
  input [0:1] Ones;
  output [0:2] Z;
  reg [0:2] Z;
  always @ (Phy or Ones)
    if (Phy)
      Z = Ones + 1;
```

```
endmodule
// 综合出的网表如图 2-25 所示
```

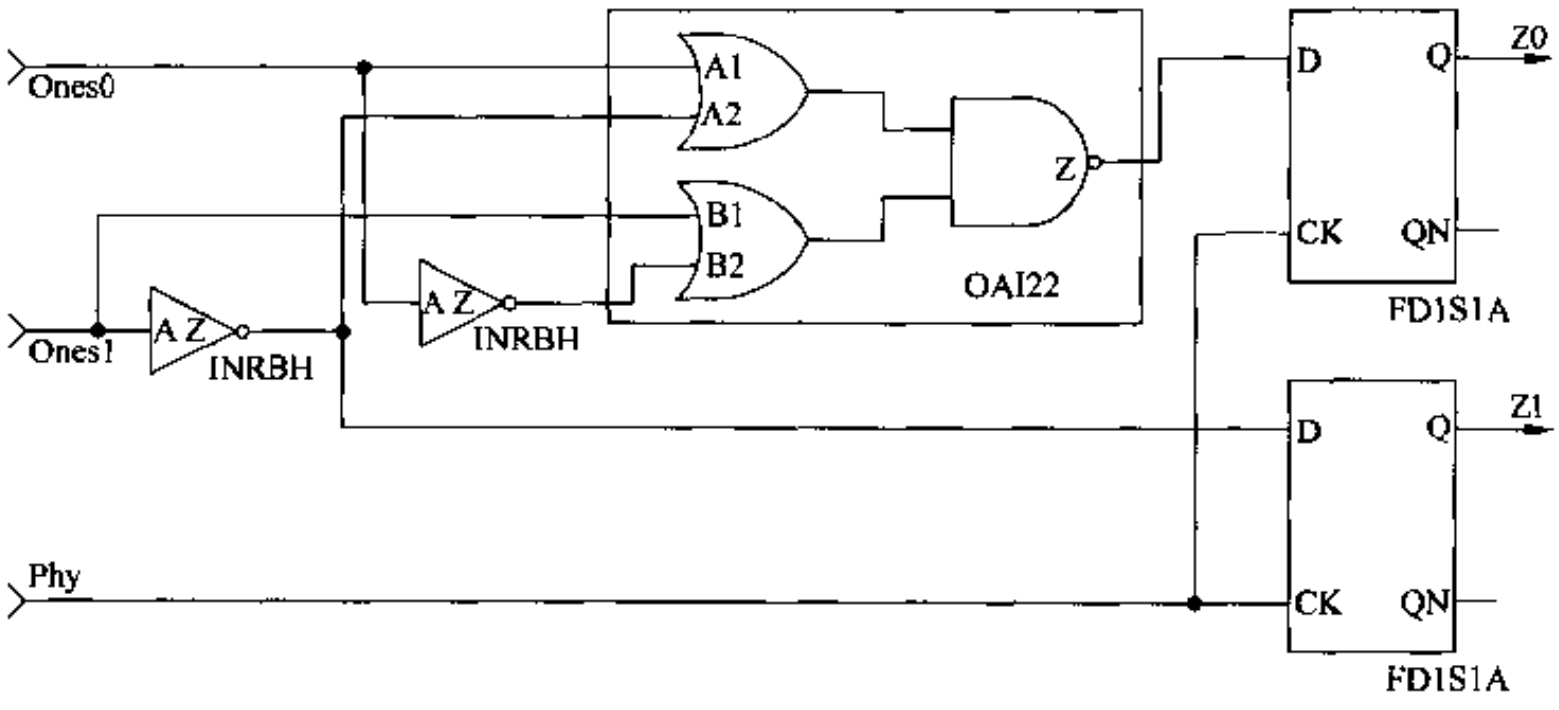

图 2-25 变量综合成锁存器

此 always 语句的语义表明：每次 *Phy* 或 *Ones*(出现在事件表中的变量)上有事件发生，就会执行 if 语句。如果 *Phy* 为 1，则变量 *Z* 得到 *Ones* 加 1 的值；如果 *Phy* 为 0，则 *Z* 保持其原值。这是通过使用锁存器来实现的。

推导出锁存器的一般规则是，如果变量未在 always 语句所有可能的执行过程中被赋值(比如变量未能在 if 语句的所有分支中都被赋值)，就推导出锁存器。

以下是变量未在 if 语句的所有分支中都被赋值的另一个例子。

```
module Compute (Marks, Grade);
  input [1:4] Marks;
  output [0:1] Grade;
  reg [0:1] Grade;

  parameter FAIL = 1, PASS = 2, EXCELLENT = 3;

  always @ (Marks)
    if (Marks < 5)
      Grade = FAIL;
    else if ((Marks >= 5) & (Marks < 10))
      Grade = PASS;
endmodule
// 综合出的网表如图 2-26 所示
```

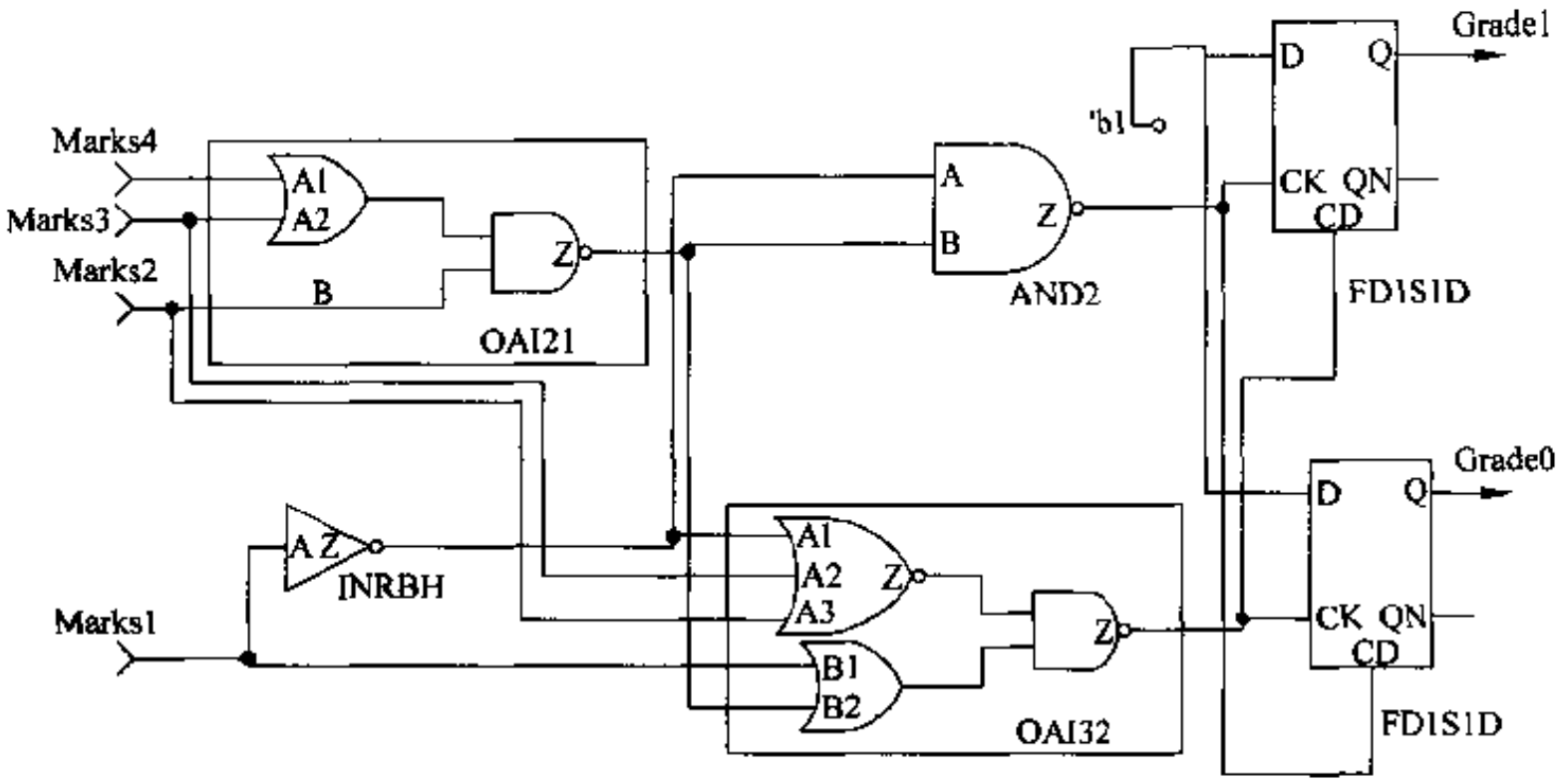

图 2-26　变量推导为锁存器

此例中，如果 *Marks* 值是 12，那么 *Grade* 值应该是多少呢？或许它是一个无关值，但从语义角度考虑，变量 *Grade* 应保持其原值，这是因为 *Marks* 值为 12 时变量 *Grade* 未被明确地赋值。因此 *Grade* 被推导成锁存器以便与寄存器变量的仿真语义保持一致。

如上例所示，推导出锁存器时应避免把算术运算作为条件表达式，因为在综合出的网表中各条件分支之间存在很高的竞态条件，这就会造成综合出的网表中的锁存值和 Verilog HDL 模型中的锁存值有出入。

如果变量未在 if 语句的所有条件分支中都被赋值，而且原本也不打算推导出锁存器，那么必须在 if 语句的所有分支中都对变量明确地赋值。对上例稍作修改，实现在所有分支中都对变量赋值，得到以下程序：

```
module ComputeNoLatch (Marks, Grade);
  input [1:4] Marks;
  output [0:1] Grade;
  reg [0:1] Grade;
  parameter FAIL = 1, PASS = 2, EXCELLENT = 3;

  always @ (Marks)
    if (Marks < 5)
      Grade = FAIL;
    else if ((Marks >= 5) && (Marks < 10))
      Grade = PASS;
    else
```

```
        Grade = EXCELLENT;
endmodule
// 综合出的网表如图 2-27 所示
```

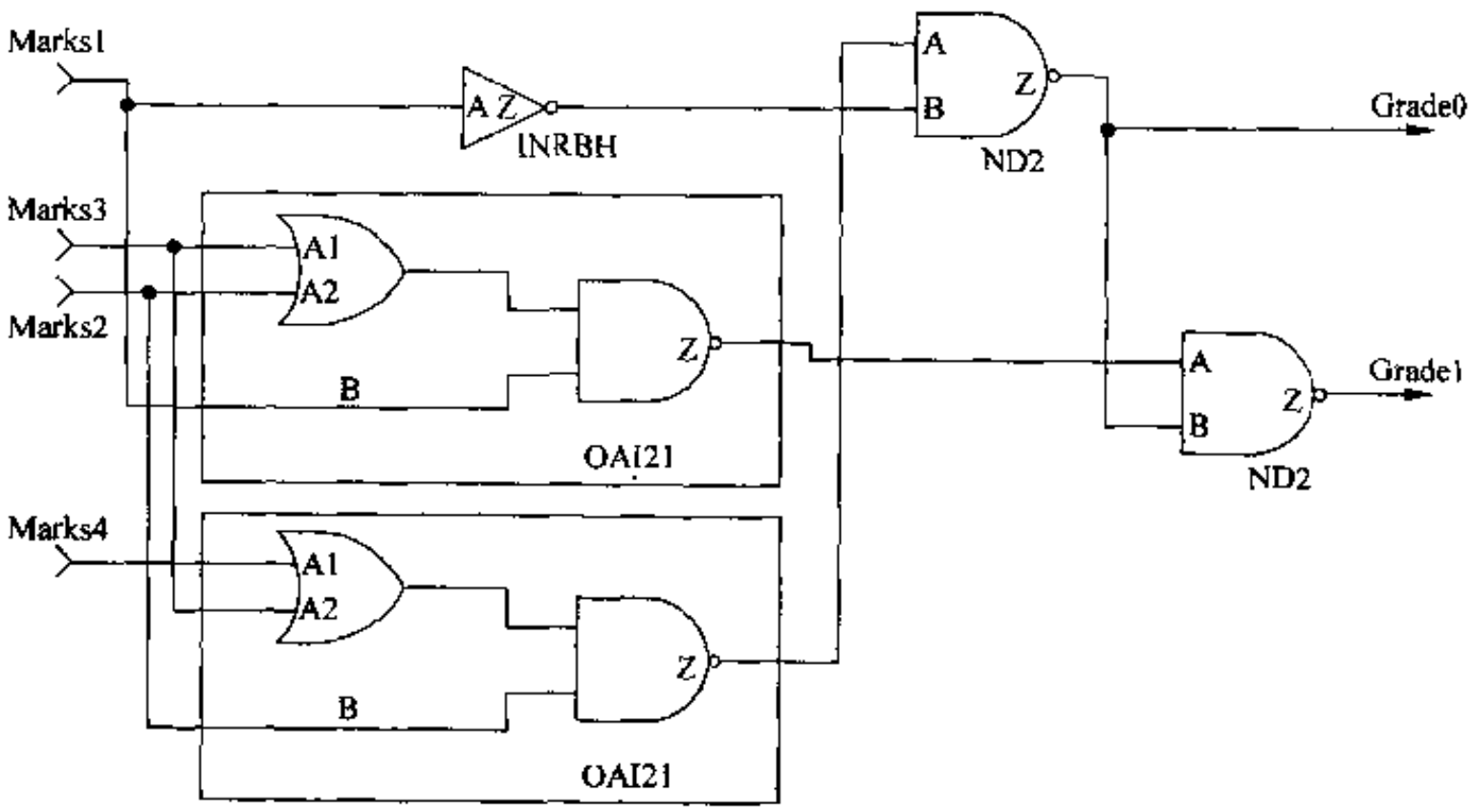

图 2-27 变量 Grade 不是锁存器

此例中，变量 *Grade* 不再是锁存器，这是因为 if 语句所有分支中都对该变量进行了赋值。

2.14 case 语 句

case 语句格式如下：

```
case（< case 表达式 >）
   < case 分支项 A1 >，< case 分支项 A2 >，… ：< 语句 A >
   < case 分支项 B1 >，< case 分支项 B2 >，… ：< 语句 B >
   …
   default                                    ：< 语句 D >
endcase
```

选择执行分支项与分支表达式值相匹配的第一个分支。分支项既可以是常量，也可以是变量。

请看以下 case 语句示例：

```
module ALU (Op, A, B, Z);
  input [1:2] Op;
```

```
    input [0:1] A, B;
    output [0:1] Z;
    reg [0:1] Z;

    parameter ADD = 'b00, SUB = 'b01, MUL = 'b10,
              DIV = 'b11;

    always @ (Op or A or B)
      case (Op)
        ADD : Z = A + B;
        SUB : Z = A - B;
        MUL : Z = A * B;
        DIV : Z = A/B; // 某些综合工具可能不支持综合 A/B 运算
      endcase
endmodule
// 综合出的网表如图 2-28 所示
```

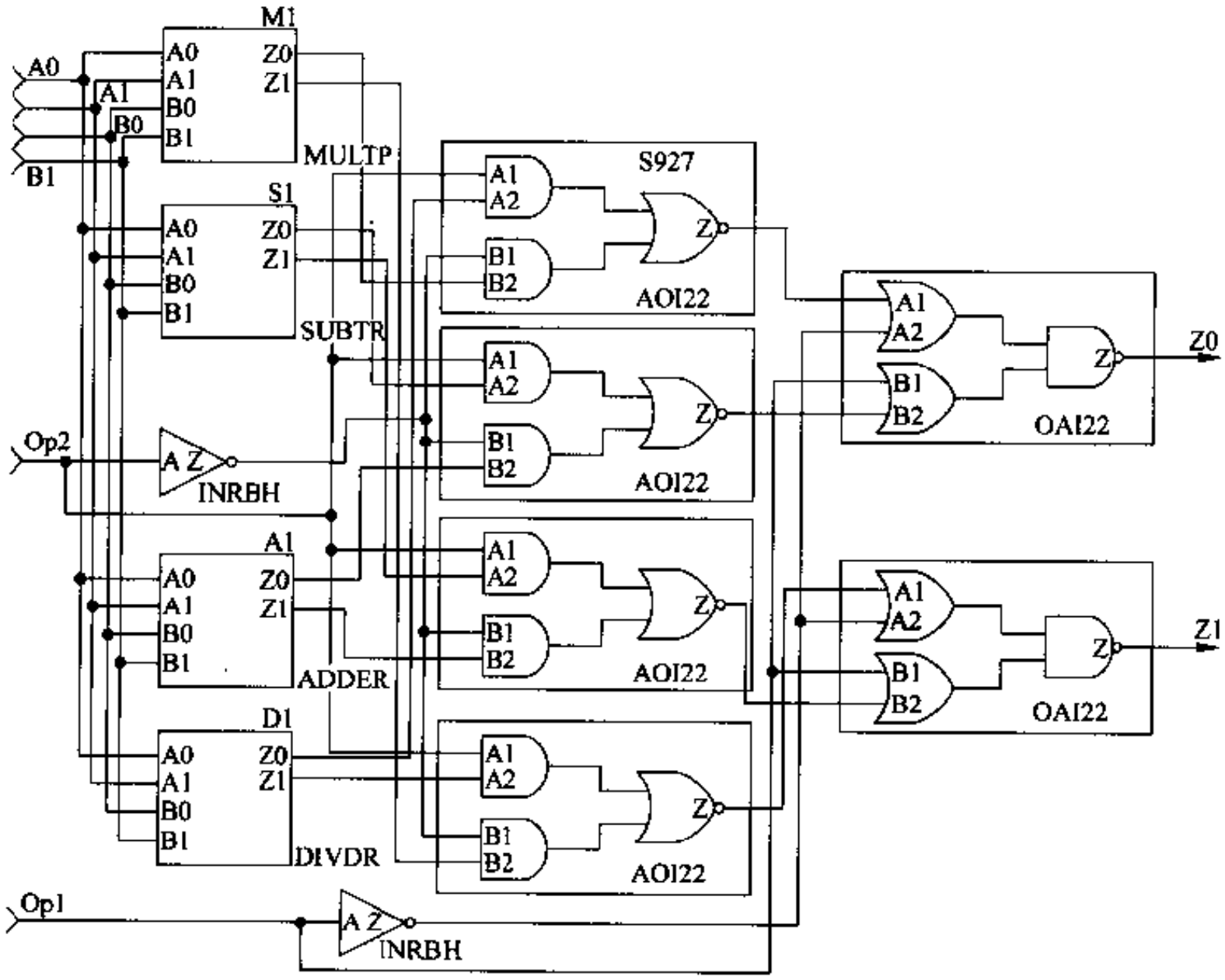

图 2-28　2 位的 ALU

case 语句的行为如同一个嵌套的 if 语句，也就是说，此 case 语句用第一个分支项(*ADD*)来核对分支表达式(*Op*)的值，如果不一致，继而用第二个分支项(*SUB*)来核对，依此类推……与上面这个 case 语句等价的 if 语句如下所示：

```
if (Op == ADD)
  Z = A + B;
else if (Op == SUB)
  Z = A - B;
else if (Op == MUL)
  Z = A * B;
else if (Op == DIV)
  Z = A/B;
```

以下是另一个 case 语句示例：

```
module CaseExample (DayOfWeek, SleepTime);
  input [1:3] DayOfWeek;
  output [1:4] SleepTime;
  reg [1:4] SleepTime;
  parameter MON = 0, TUE = 1, WED = 2, THU = 3, FRI = 4,
            SAT = 5, SUN = 6;
  always @ (DayOfWeek)
    case (DayOfWeek)
      MON,
      TUE,
      WED,
      THU:    SleepTime = 6;
      FRI:    SleepTime = 8;
      SAT:    SleepTime = 9;
      SUN:    SleepTime = 7;
      default:    SleepTime = 10;    // 恭喜!
      // default 分支涵盖了 DayOfWeek 取值 7 的情况
  endcase
endmodule
// 综合出的网表如图 2-29 所示
```

请再看一个 case 语句示例：

```
module SelectAndAssign (CurrentState, RFlag);
  input [0:1] CurrentState;
  output [0:1] RFlag;
  reg [0:1] RFlag;
```

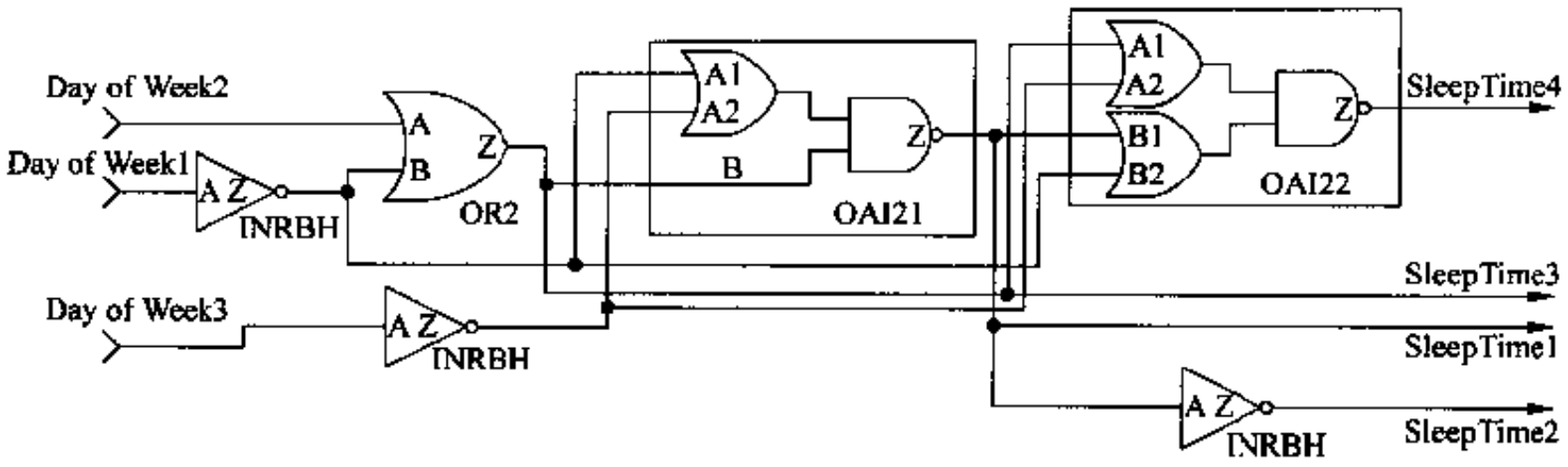

图 2-29　case 语句示例

```
parameter RESET = 2'b01, APPLY = 2'b11, WAITS = 2'b10,
          DONTCARE = 2'b00;

always @ (CurrentState)
  case (CurrentState)
    RESET: RFlag = WAITS;
    APPLY: RFlag = RESET;
    WAITS: RFlag = APPLY;
    default: RFlag = DONTCARE;
  endcase
endmodule
// 综合出的网表如图 2-30 所示
```

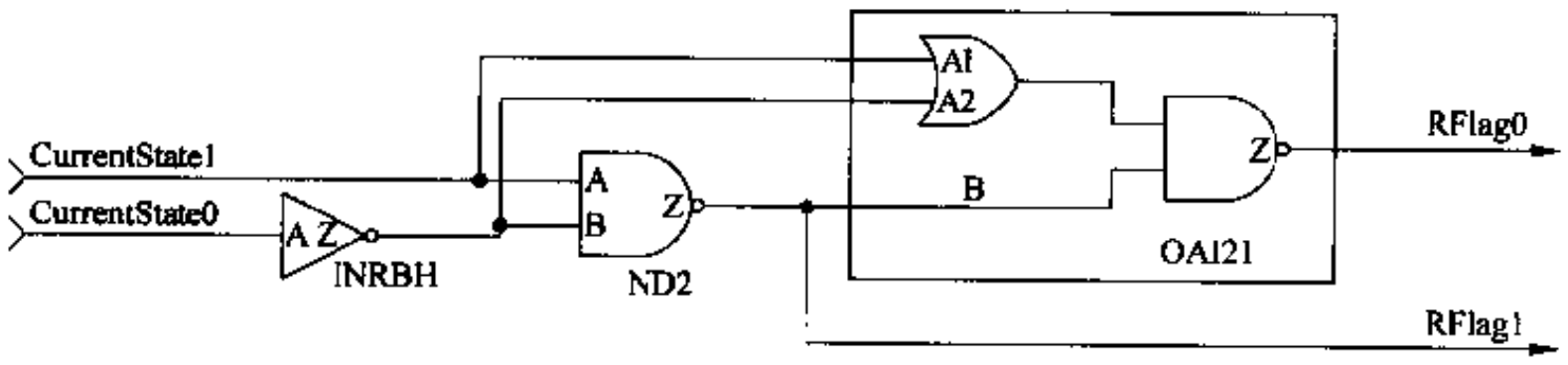

图 2-30　case 语句生成的逻辑

2.14.1　casez 语句

casez 语句中，case 分支项表达式中出现的 z 值会被视为无关值。字符“?”也可以用来代替字符 z 表示相同的含义。但不允许在 case 表达式中使用 z 值和 x 值。此外，也不允许在 case 分支项表达式中使用 x 值。请看以下 casez 语句示例：

```
module CasezExample (ProgramCounter, DoCommand);
  input [0:3] ProgramCounter;
  output [0:1] DoCommand;
  reg [0:1] DoCommand;

  always @ (ProgramCounter)
    casez (ProgramCounter)
      4'b???1 : DoCommand = 0;
      4'b??10 : DoCommand = 1;
      4'b?100 : DoCommand = 2;
      4'b1000 : DoCommand = 3;
      default : DoCommand = 0;
    endcase
endmodule
// 综合出的网表如图 2-31 所示
```

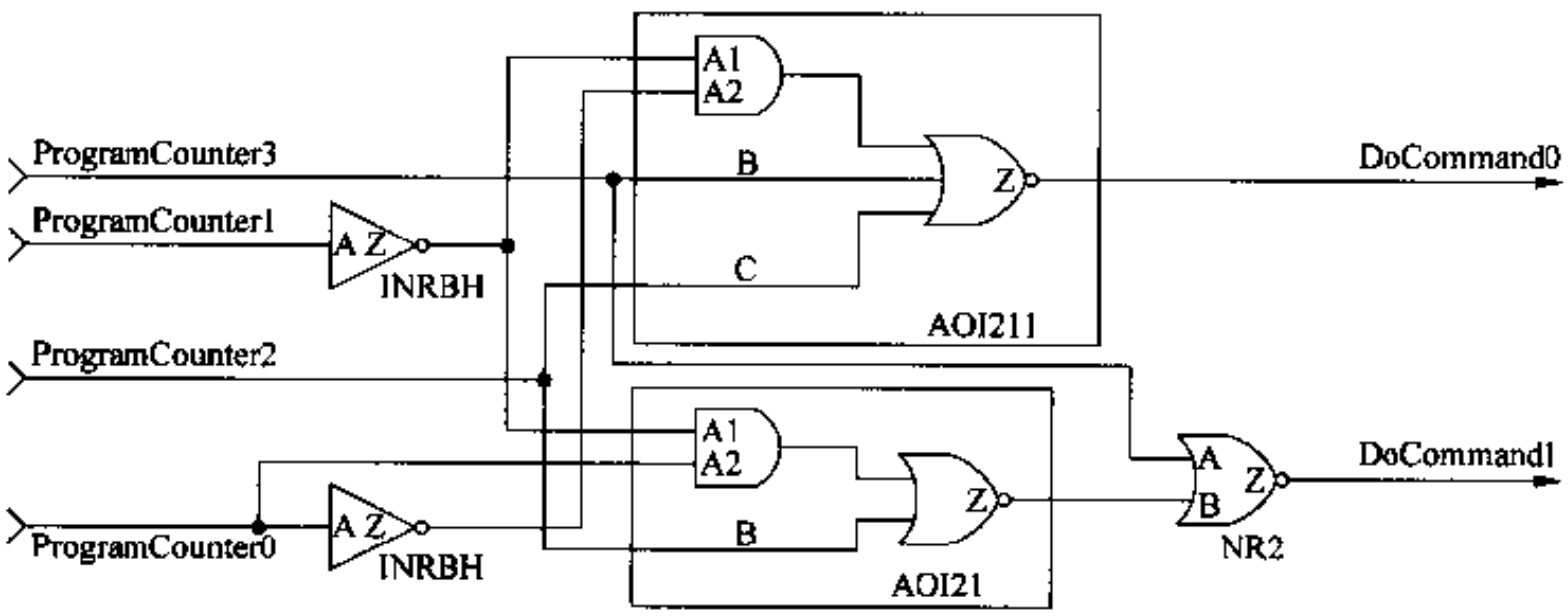

图 2-31　casez 语句示例

casez 语句等价于以下这条 if 语句(注意 case 分支项中的字符“?”表示无关值)。

```
if (ProgramCounter [3])
  DoCommand = 0;
else if (ProgramCounter [2:3] == 2'b10)
  DoCommand = 1;
else if (ProgramCounter [1:3] == 3'b100)
  DoCommand = 2;
else if (ProgramCounter [0:3] == 4'b1000)
  DoCommand = 3;
else
  DoCommand = 0;
```

2.14.2　casex 语句

casex 语句中，case 分支项表达式中的 x 值和 z 值(也可以用“?”来替代 z)会被视为无关值。如果打算综合，这些值就不能成为 case 表达式的一部分。下例中用 casex 语句实现优先级编码器的建模。

```
module PriorityEncoder (Select, BitPosition);
  input [5:1] Select;
  output [2:0] BitPosition;
  reg [2:0] BitPosition;

  always @ (Select)
    casex (Select)
      5'bxxxx1 : BitPosition = 1;
      5'bxxx1x : BitPosition = 2;
      5'bxx1xx : BitPosition = 3;
      5'bx1xxx : BitPosition = 4;
      5'b1xxxx : BitPosition = 5;
      default : BitPosition = 0;
    endcase
endmodule
// 综合出的网表如图 2-32 所示
```

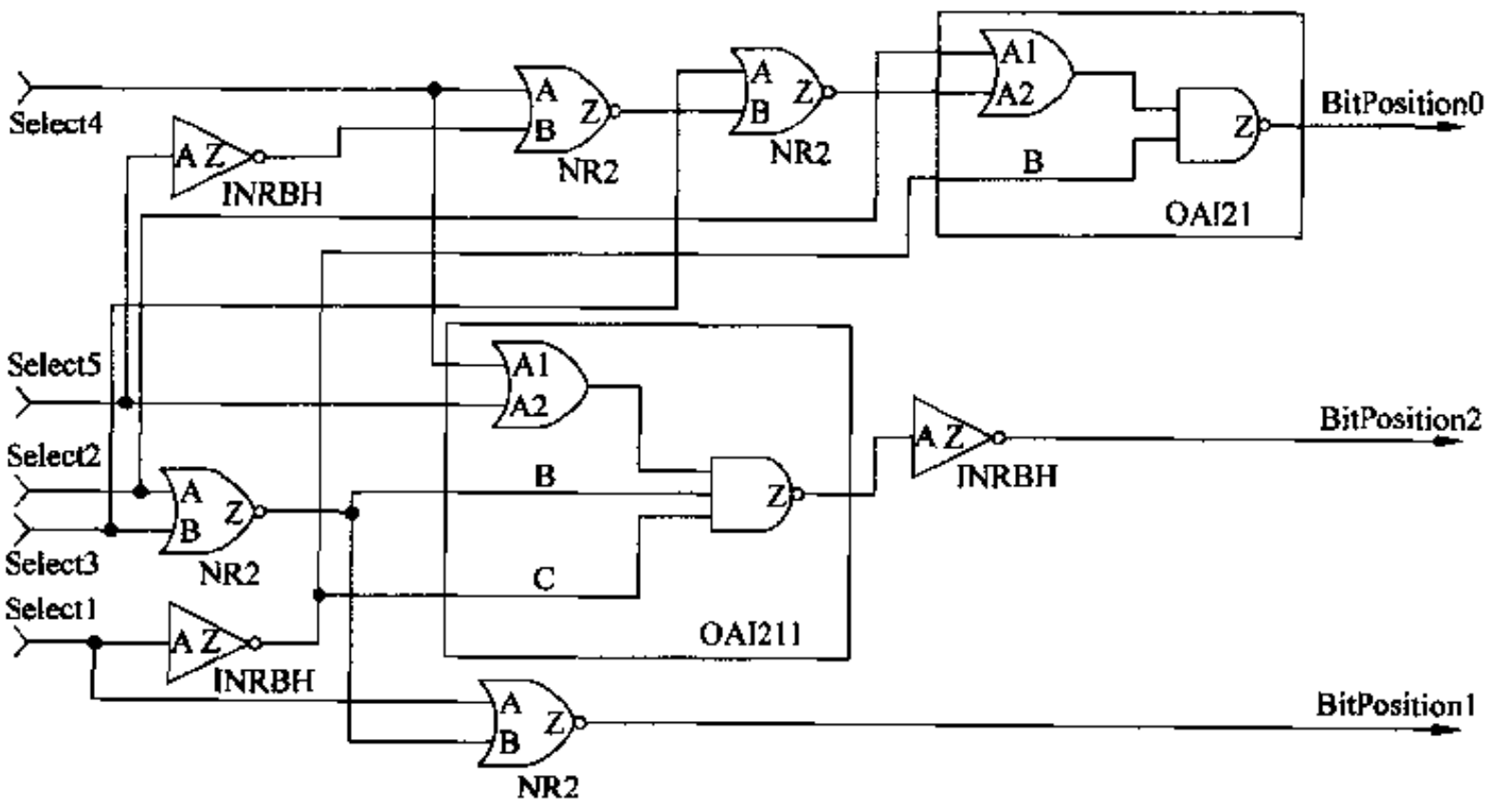

图 2-32　使用 casex 语句的优先级编码器

此 casex 语句的语义可用等价的 if 语句表示如下：

```
if (Select [1])
  BitPosition = 1;
else if (Select [2])
  BitPosition = 2;
else if (Select [3])
  BitPosition = 3;
else if (Select [4])
  BitPosition = 4;
else if (Select [5])
  BitPosition = 5;
else
  BitPosition = 0;
```

2.14.3 从 case 语句推导出锁存器

就像 if 语句那样，case 语句中被赋值的变量也可被推导成锁存器。如果变量未在 always 语句所有可能的执行过程中都被赋值(比如变量仅在 case 语句的某些分支中被赋值)，那么就会把该变量推导成锁存器。请看下例：

```
module StateUpdate (CurrentState, Zip);
  input [0:1] CurrentState;
  output [0:1] Zip;
  reg [0:1] Zip;

  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

  always @ (CurrentState)
    case (CurrentState)
      S0,
      S3: Zip = 0;
      S1: Zip = 3;
    endcase
endmodule
// 综合出的网表如图 2-33 所示
```

此例未针对输入信号 *CurrentState* 各种可能的取值对变量 *Zip* 加以赋值，因此，为了和寄存器变量的语义保持一致，须将 *Zip* 推导成锁存器。综合出的网表中可看到该锁存器。就锁存器推导而言，case 语句的行为等同于 if 语句。如果要避免推导出锁存器，则

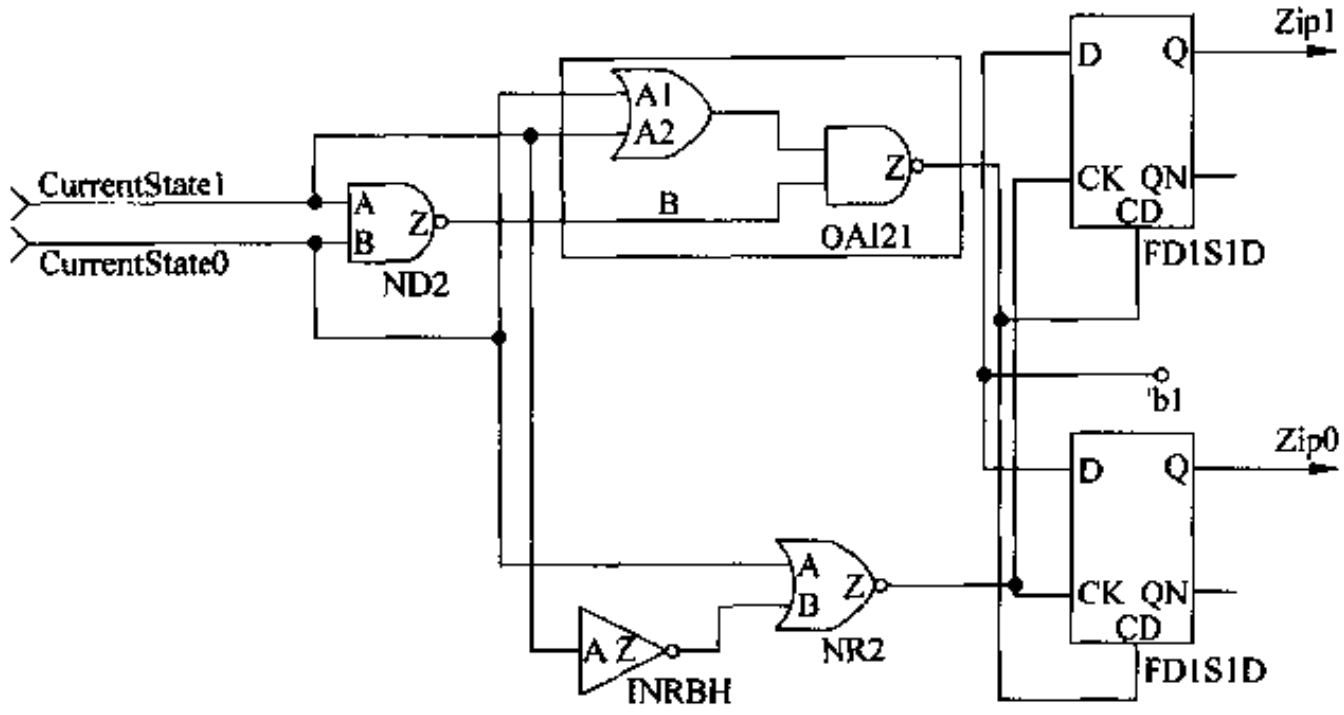

图 2-33　case 语句内的变量被推导成锁存器

需要像以下这段代码那样，在 case 语句之前加上对 *Zip* 赋初值的语句。对 *Zip* 的明确赋值使得它在 *CurrentState* 的各种取值情况下都能够被定义，从而在 always 语句所有可能的执行过程中也得以定义。

```
...
always @ (CurrentState)
begin
  Zip = 0;     // 新增加的语句

  case (CurrentState)
  ...
  endcase
end
```

锁存器推导规则对 casez 语句和 casex 语句也完全适用。

2.14.4　case 分支的全列举

从前面各节中可知，如果未在 case 表达式所有可能的取值情况下都对变量赋值，则该变量就会被推导成锁存器。有时，设计者知道 case 表达式不会取 case 分支项中未列出的任何其他值。请看下例：

```
module NextStateLogic (NextToggle, Toggle);
  input [1:0] Toggle;
  output [1:0] NextToggle;
```

```
  reg [1:0] NextToggle;

  always @ (Toggle)
    case (Toggle)
      2'b01 : NextToggle = 2'b10;
      2'b10 : NextToggle = 2'b01;
    endcase
endmodule
// 综合出的网表如图 2-34 所示
```

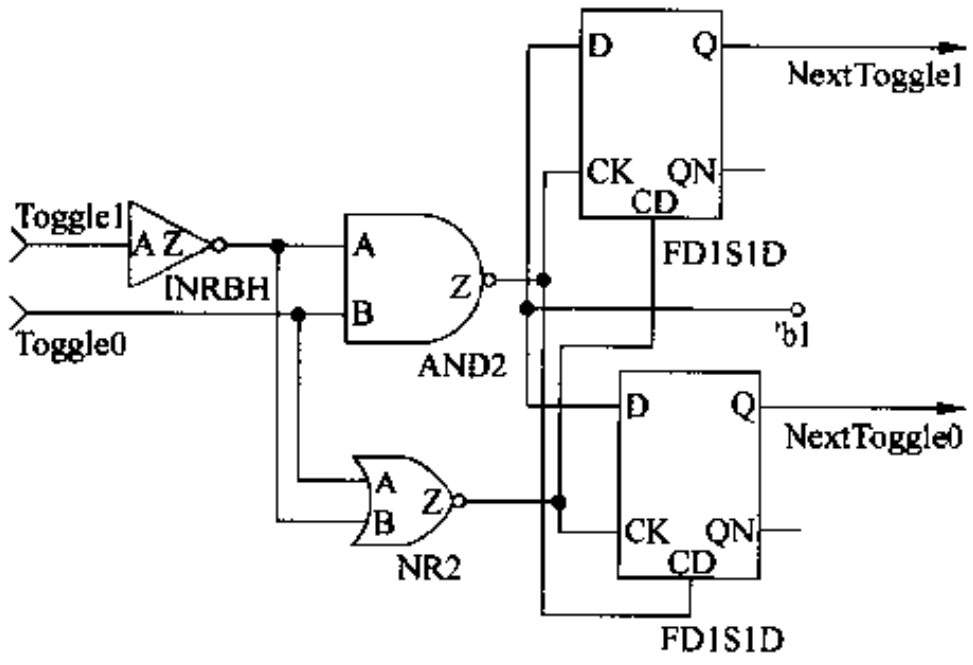

图 2-34 NextToggle 被推导成锁存器

设计者知道 *Toggle* 不可能取 2'b01 和 2'b10 之外的其他值。有必要让综合工具也知道这一点。如果不向综合工具提供该信息，就会把 *NextToogle* 推导成锁存器(图 2-34 中的两个 FD1S1D 单元)，因为在 case 表达式值为 2'b00 和 2'b11 时未对它赋值。这可以用综合指令 full_case 让综合工具知道。综合指令是设计模型中用来向综合工具提供附加信息的特殊代码。在与 case 语句相关的模型中综合指令 full_case 被指定为 Verilog HDL 注释形式；正因为综合指令都以注释形式出现，所以它对于设计模型的语义不产生丝毫影响。

综合工具在 case 语句中遇到这样的综合指令时就会认为：case 表达式所有可能的取值(仅对该设计而言)都已罗列出来了，不存在其他可能的取值。这样变量就在 case 语句的所有分支中都被赋值了，因而不会推导成锁存器。以下 *NextStateLogic* 模块中的 case 语句采用了这样的综合指令。

```
module NextStateLogicFullCase (NextToggle, Toggle);
  input [1:0] Toggle;
  output [1:0] NextToggle;
  reg [1:0] NextToggle;
```

```
  always @ (Toggle)
    case (Toggle)       // synthesis full_case
      2'b01 : NextToggle = 2'b10;
      2'b10 : NextToggle = 2'b01;
    endcase
endmodule
// 综合出的网表如图 2-35 所示
```

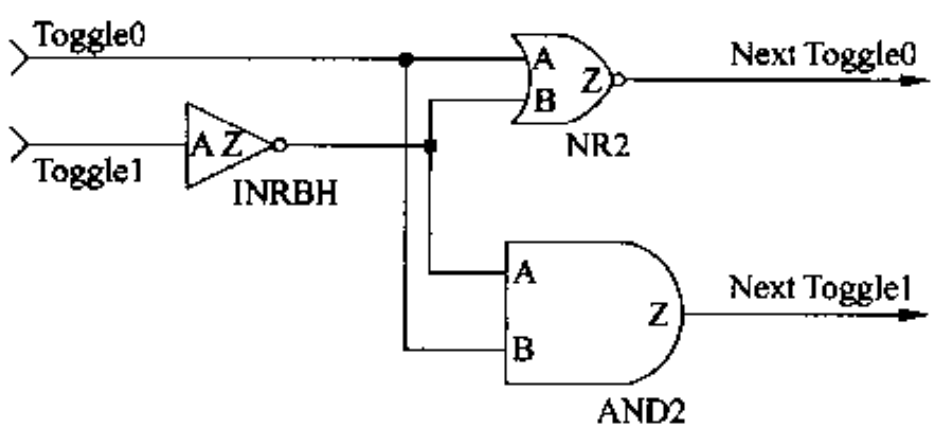

图 2-35　使用 full_case 指令：不产生锁存器

如综合出的网表所示，使用了 full_case 综合指令，就不会把 *NextToggle* 推导成锁存器。

另一种避免在上例中产生锁存器的方法是，在 case 语句中给出默认分支，或者在 case 语句之前对 case 语句中的所有变量(此例中的变量是 *NextToggle*)进行默认赋值。下例中使用默认分支来避免推导出锁存器。

```
always @ (Toggle)
  case (Toggle)
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
    default : NextToggle = 2'b01; // 假赋值
  endcase
```

下例的 always 语句中对 *NextToggle* 进行了默认赋值，因此没有把该变量推导成锁存器。

```
always @ (Toggle)
begin
  NextToggle = 2'b01;   // 默认赋值

  case (Toggle)
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
  endcase
end
```

警告：使用 full_case 指令可能会导致设计模型和综合出的网表功能不一致，请参见第 5 章中的相关示例。

2.14.5 并行 case 分支

case 语句的 Verilog HDL 语义表明了选取 case 分支的优先级顺序。case 表达式首先与第一个分支项进行比较，如果不相同，则再与下一个分支项进行比较，若还不相同再与下一个相比较，依次类推。case 语句暗示了检查分支项的优先级顺序。此外，Verilog HDL 中可能有两个或更多分支项的值相同或重叠，casex 和 casez 语句中也会出现这种情形。然而，因为有了执行的优先级顺序，只有出现在最前面的那个分支项才会被选中。

为了在综合成硬件时严格应用 case 语句的语义，便综合出嵌套的 if 式结构（优先级逻辑为：if…，else if…，else…）。请看以下 case 语句示例：

```
module PriorityLogic (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  always @ (Toggle)
    casex (Toggle)
      3'bxx1 : NextToggle = 3'b010;
      3'bx1x : NextToggle = 3'b110;
      3'b1xx : NextToggle = 3'b001;
      default : NextToggle = 3'b000;
    endcase
endmodule
// 综合出的网表如图 2-36 所示
```

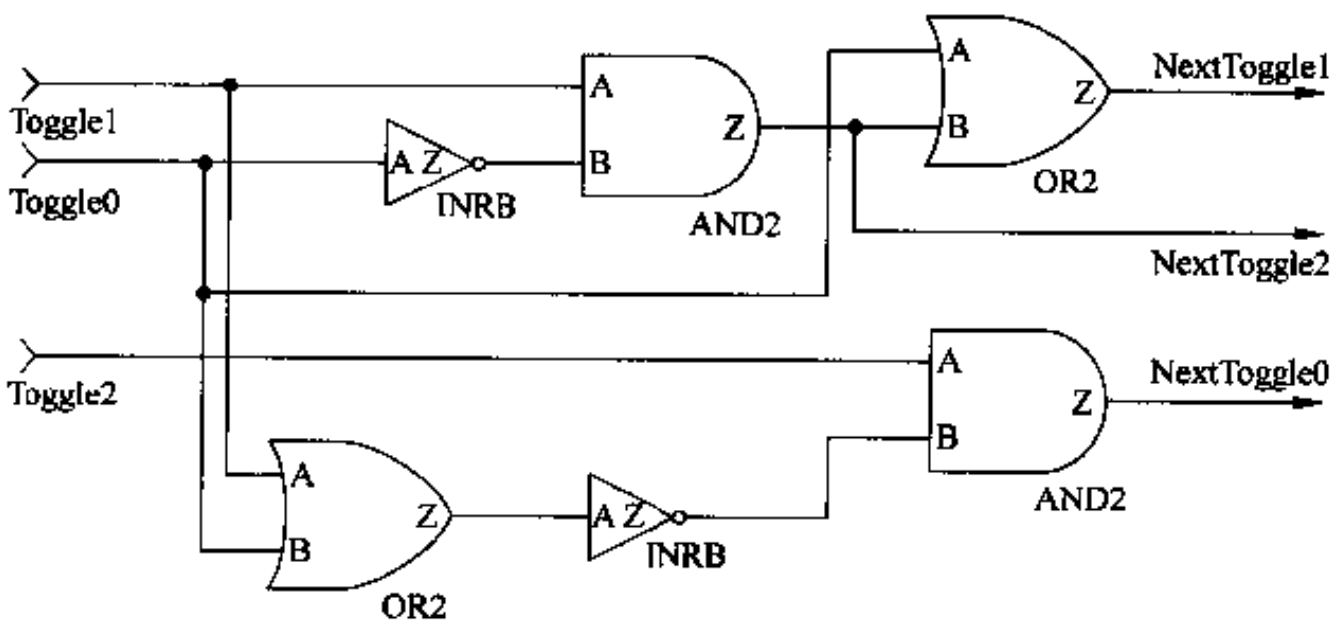

图 2-36 优先级逻辑选择各分支

此 case 语句的行为可以等价地表达成以下 if 语句：

```
if (Toggle[0] == 'b1)
  NextToggle = 3'b010;
else if (Toggle[1] == 'b1)
  NextToggle = 3'b110;
else if (Toggle[2] == 'b1)
  NextToggle = 3'b001;
else
  NextToggle = 3'b000;
```

如果设计者知道所有的分支项相互排斥，结果又会如何？此时，会把 case 语句的控制逻辑综合成译码器(各种可能的分支项并行地与 case 表达式相比较)，而不是综合成优先级逻辑(这可能会导致嵌套得很深——取决于 case 语句的分支数)。

要想让综合工具知道各分支项互相排斥，可以通过使用综合指令 parallel_case 来实现。case 语句中附加了该指令，综合工具就知道 case 语句各分支项互相排斥。因为综合指令以注释形式出现在 Verilog HDL 模型中，所以这不会影响模型的语义。这意味着不会把 case 语句的控制逻辑综合成优先级逻辑，而是综合成译码逻辑。以下 case 语句使用了 parallel_case 指令：

```
module ParallelCase (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  always @ (Toggle)
    casex (Toggle)        // synthesis parallel_case
      3'bxx1 : NextToggle = 3'b010;
      3'bx1x : NextToggle = 3'b110;
      3'b1xx : NextToggle = 3'b001;
      default : NextToggle = 3'b000;
    endcase
endmodule
// 综合出的网表如图 2-37 所示
```

与此 case 语句等价的综合解释如下(仅保证一个 if 条件为真)：

```
if (Toggle[0] == 'b1)
  NextToggle = 3'b010;

if (Toggle[1] == 'b1)
```

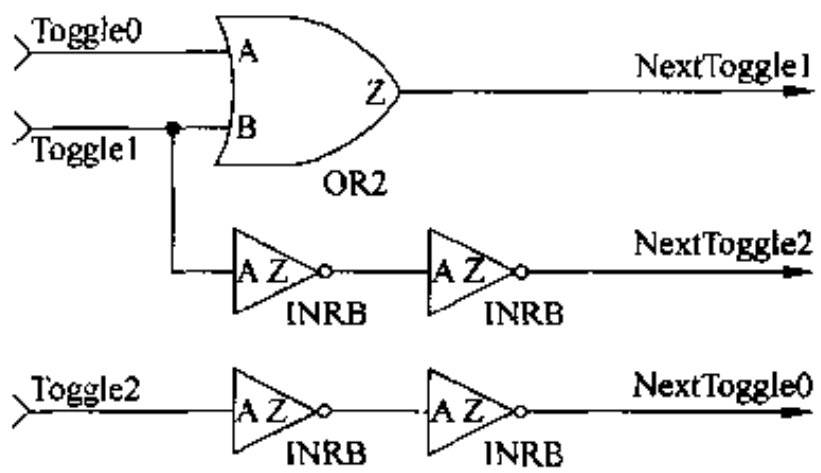

图 2-37 使用 parallel_case 指令：不产生优先级逻辑

```
  NextToggle = 3'b110;
if (Toggle[2] == 'b1)
  NextToggle = 3'b001;

if ((Toggle[0] ! = 'b1) &&
    (Toggle[1] ! = 'b1) &&
    (Toggle[2] ! = 'b1))
  NextToggle = 3'b000;
```

使用该综合指令，分支逻辑就能综合成如图 2-37 所示的译码逻辑；不使用该综合指令，就会综合出如图 2-36 所示的优先级逻辑。

警告：综合指令 parallel_case 可能会引起设计模型和综合出的网表功能不一致，第 5 章将会对此作更详尽的阐述。

2.14.6 非常量分支项

Verilog HDL 中，case 分支项可以使用非常量表达式。以下优先级编码器示例正是如此：

```
module PriorityEncoder (Pbus, Address);
  input [0:3] Pbus;
  output [0:1] Address;
  reg [0:1] Address;

  always @ (Pbus)
    case (1'b1)        // synthesis full_case
      pbus[0] : Address = 2'b00;
      pbus[1] : Address = 2'b01;
      pbus[2] : Address = 2'b10;
      pbus[3] : Address = 2'b11;
```

```
    endcase
endmodule
// 综合出的网表如图 2-38 所示
```

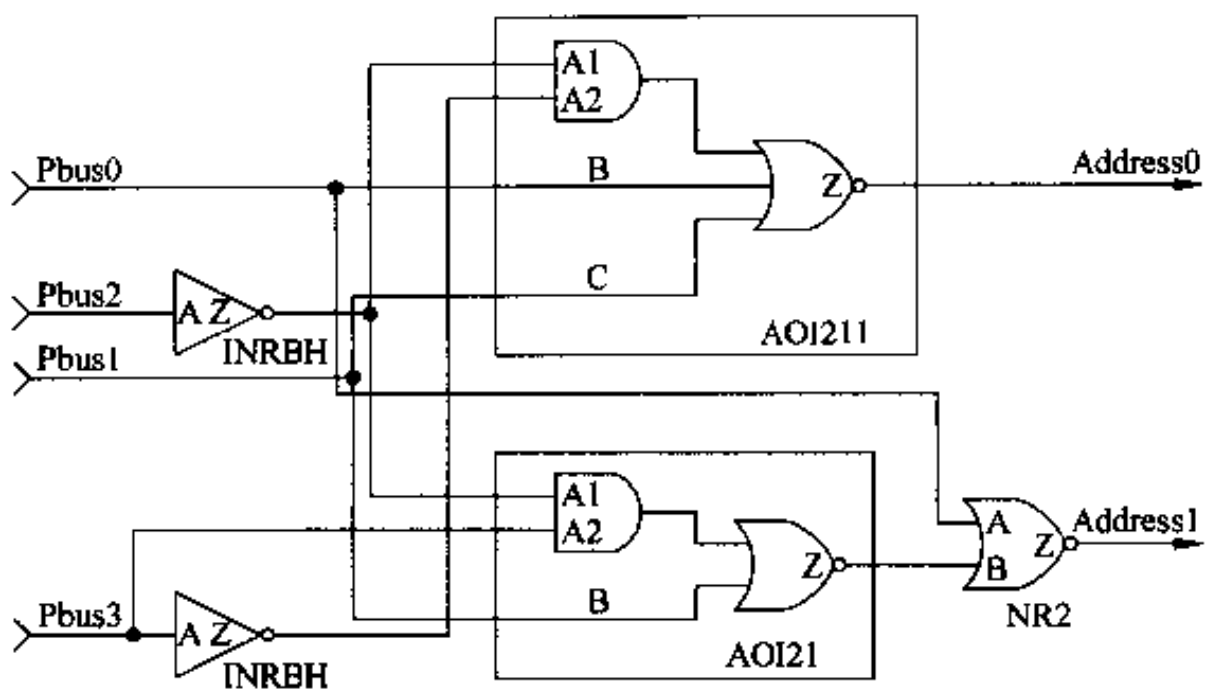

图 2-38　使用 case 语句的优先级编码器

此时有必要使用 full_case 综合指令，否则就会把 *Address* 推导成锁存器。或者在 case 语句之前对 *Address* 进行默认赋值，这样也能避免出现锁存器，从而不必使用综合指令。以下这条 always 语句正是如此：

```
always @ (Pbus)
begin
  Address = 2'b00;
  case (1'b1)
    Pbus[0] : Address = 2'b00;
    Pbus[1] : Address = 2'b01;
    Pbus[2] : Address = 2'b10;
    Pbus[3] : Address = 2'b11;
  endcase
end
```

2.15　再谈锁存器推导

使用分支未列举完全的 if 语句或者 case 语句就会推导出锁存器。也就是说，变量未在 if 语句或者 case 语句的所有分支中都被赋值，该变量就会被推导成锁存器。请看下例：

```
module LatchExample (ClockA, CurrentState, NextState);
  input ClockA;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (ClockA or CurrentState)
    if(ClockA)
      NextState = CurrentState;
endmodule
// 综合出的网表如图 2-39 所示
```

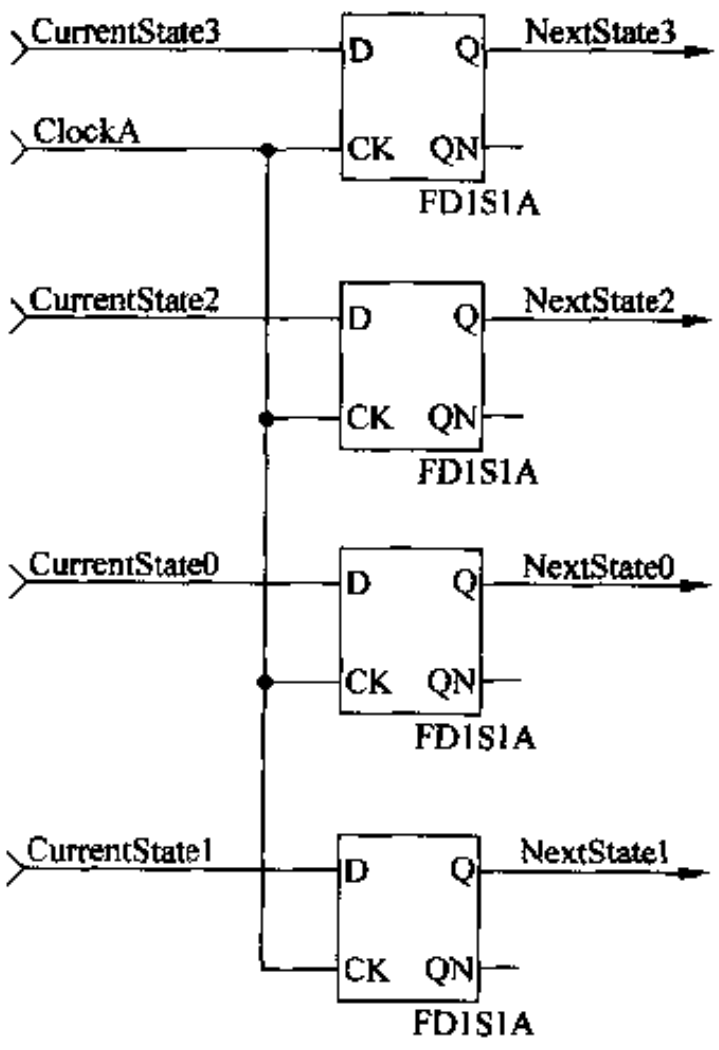

图 2-39 未完全列出所有分支的条件会推导出锁存器

变量 *NextState* 仅在 *ClockA* 为 1 的时候才被赋值。*ClockA* 为 0 时，*NextState* 保持原值，因此应推导成锁存器。

局部声明的变量

如果在 always 语句内局部声明的变量在条件语句(if 语句或者 case 语句)中未被完全赋值，则也会推导成锁存器。以下模块正是如此：

```
module LocalIntLatch (Clock, CurrentState, NextState);
  input Clock;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (Clock or CurrentState)
  begin: L1
    integer Temp;
    if (Clock)
      Temp = CurrentState;

    NextState = Temp;
  end
endmodule
// 综合出的网表如图 2-40 所示
```

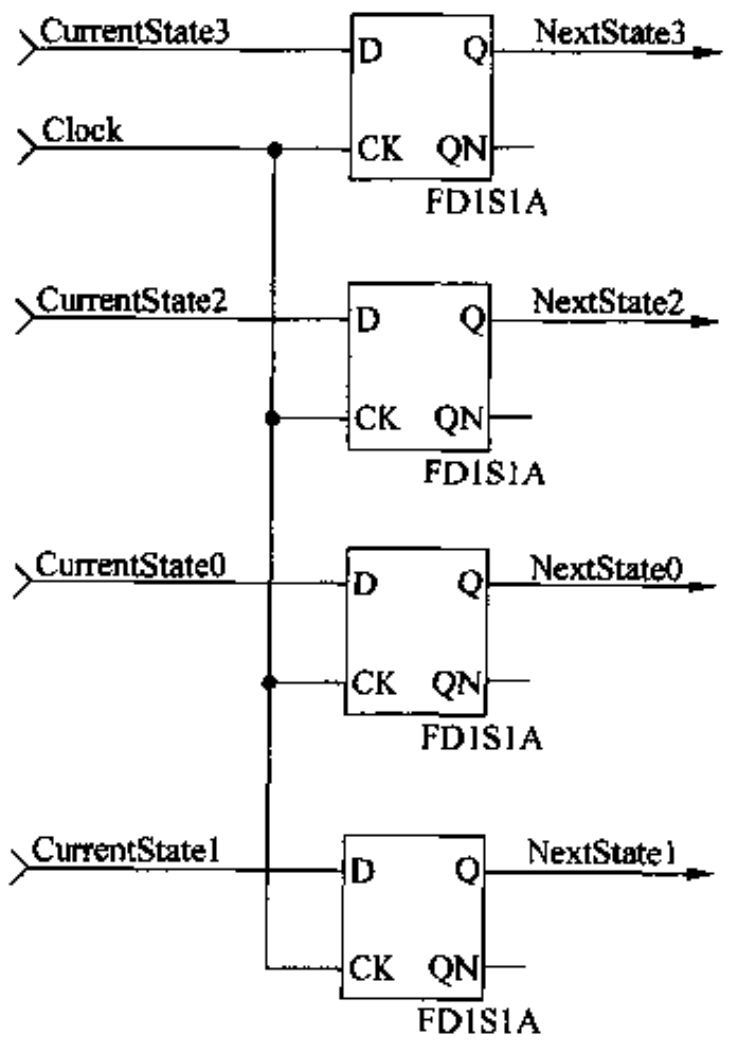

图 2-40　局部整数也能成为锁存器

引用前对变量赋值

如以下模块所示，如果变量在条件分支中被赋值和引用，则没有必要生成锁存器。这

是因为没有必要因为 *Clock* 的电平变化而保存变量 *Temp* 的值。

```
module LocalIrtNoLatch (Clock, CurrentState, NextState);
  input Clock;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (Clock or CurrentState)
  begin: L1
    integer Temp;

    if (Clock)
    begin
      Temp = CurrentState;
      NextState = Temp;
    end
  end
endmodule
// 综合出的网表与图 2-40 相同
```

赋值前引用变量

分支未列举完全的条件语句中,在赋值前就被引用的变量会被推导成锁存器。以下模块正是如此:

```
module RegUsedBeforeDef (ClockZ, CurrentState, NextState);
  input ClockZ;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  reg [3:0] Temp;

  always @ (ClockZ or CurrentState or Temp)
    if (ClockZ)
    begin
      NextState = Temp;
      Temp = CurrentState;
    end
endmodule
// 综合出的网表如图 2-41 所示
```

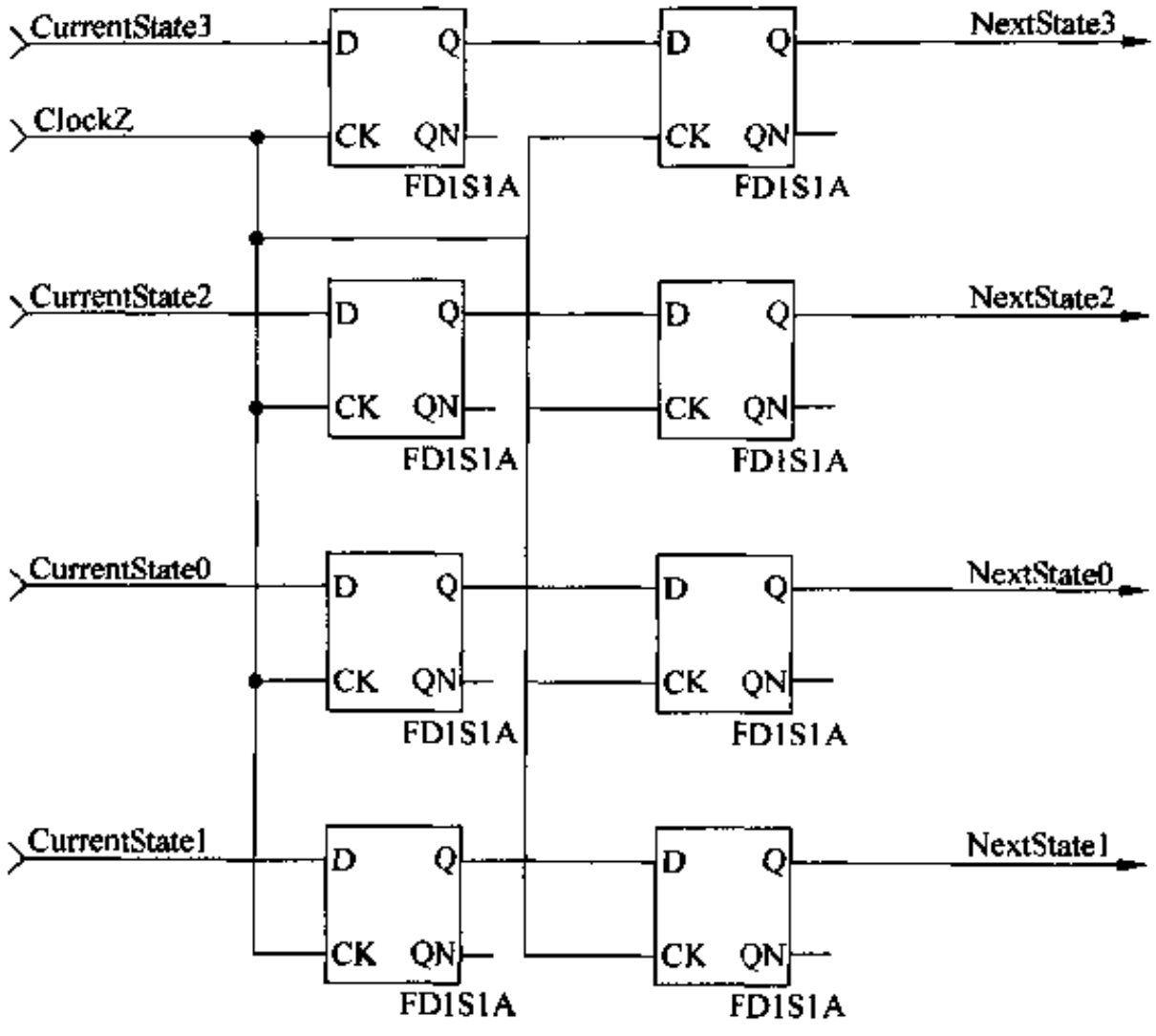

图 2-41　条件语句中变量在赋值前就被引用

如果 *Temp* 是整数，结果又会如何？

```
module LocalIntUsedBeforeDef (ClockY, CurrentState, NextState);
  input ClockY;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (ClockY or CurrentState)
  begin: L1
    integer LocalInt;

    if (ClockY)
    begin
      NextState = LocalInt;
      LocalInt = CurrentState;
    end
  end
endmodule
```

此时，综合系统会报告一个错误信息，以表明局部整数 *LocalInt* 在定义前就被引用了。还有可能综合系统仅仅产生一个警告信息而不为局部整数生成任何锁存器。

2.15.1 带异步预置位和清零的锁存器

如果推导成锁存器的变量在条件语句的某些分支中被赋给常量，被赋 1 的那些位是通过锁存器的预置位端得到赋值，而被赋 0 的那些位是通过清零端得到赋值。下例正是如此：

```
module AsyncLatch (ClockX, Reset, Set, CurrentState, NextState);
  input ClockX, Reset, Set;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (Reset or Set or ClockX or CurrentState)
```

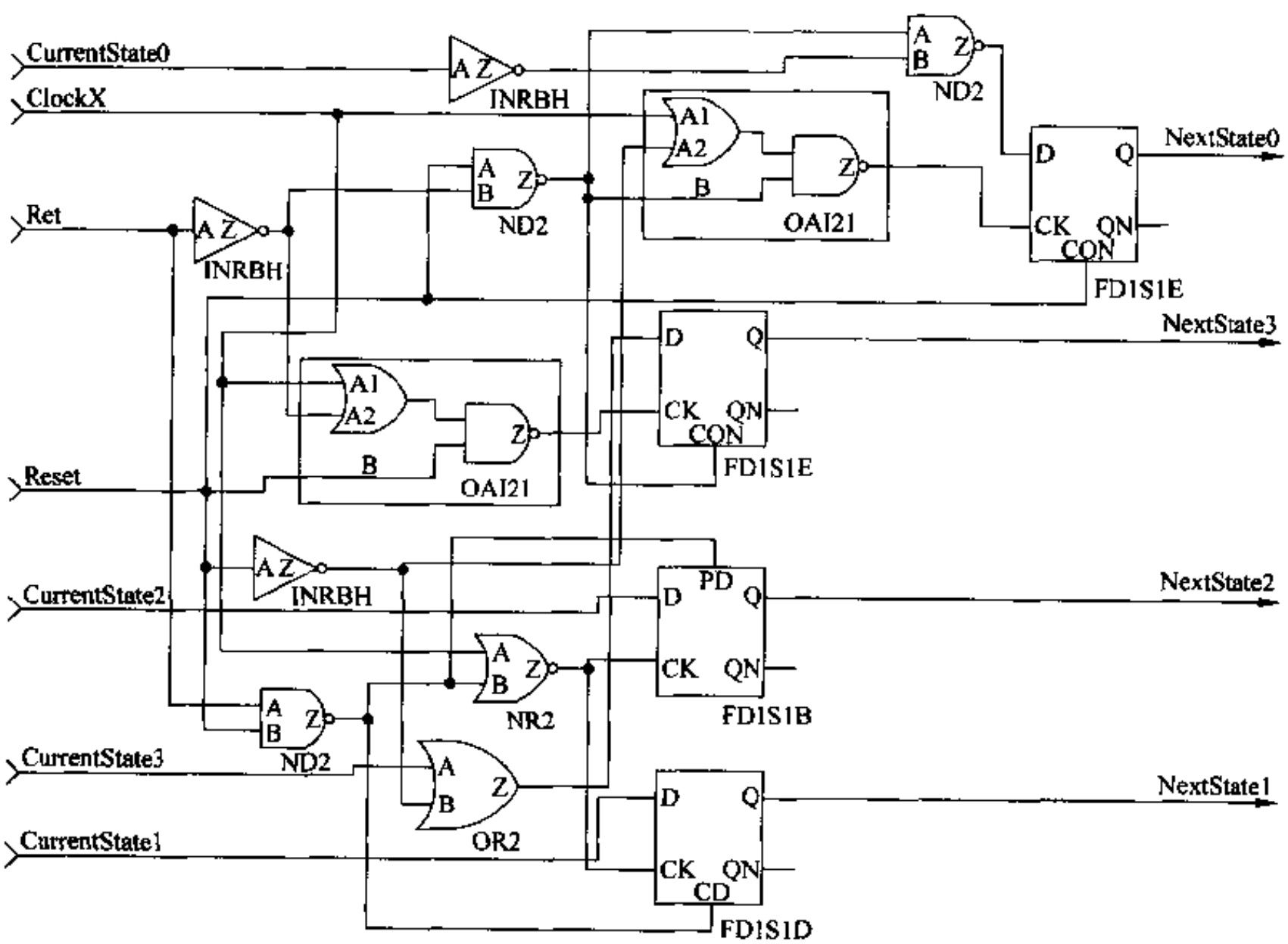

图 2-42 带异步预置位和清零的锁存器

```
    if(! Reset)
      NextState = 12;
    else if(! Set)
      NextState = 5;
    else if(! ClockX)
      NextState = CurrentState;
endmodule
// 综合出的网表如图 2-42 所示
```

NextState 会综合成 4 个锁存器，各锁存器拥有所需的预置位端或清零端。

对于上例，综合工具还可以选择不生成带异步预置位和清零的锁存器，而是将预置位清零逻辑引到简单锁存器的 D 输入端，这样就能得到如图 2-43 所示的综合出的网表。

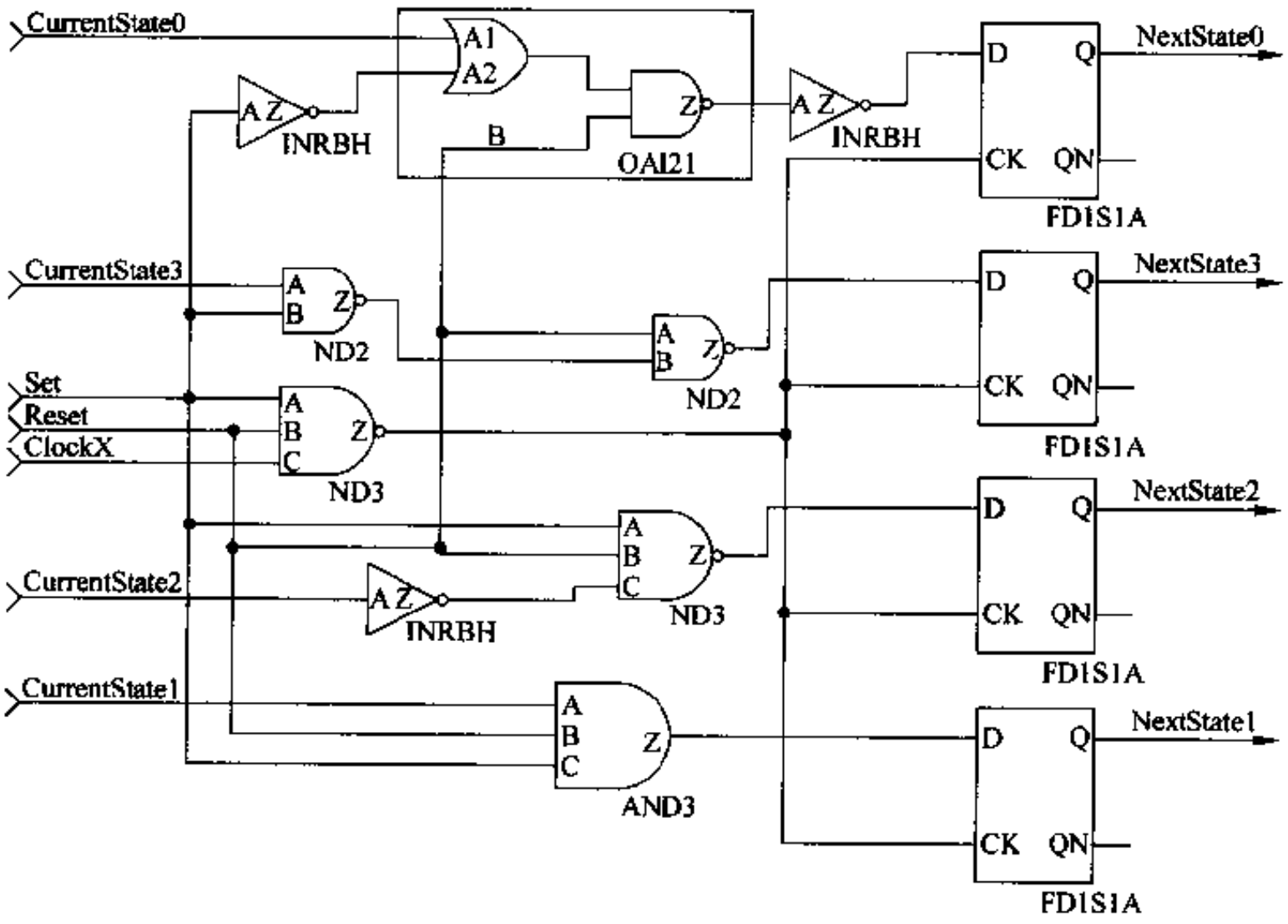

图 2-43　无异步预置位和清零的锁存器

2.16　循 环 语 句

Verilog HDL 中有 4 种不同类型的循环语句：

a）while 循环

b) for 循环

c) forever 循环

d) repeat 循环

for 循环语句是典型的能用于综合的循环语句。for 循环的综合是通过展开循环来实现的。也就是说，复制 for 循环中的所有语句，对循环变量每一个可能的取值复制一次。这就要求对 for 语句的循环边界加以限制，要保证循环边界都是常量。请看以下 for 循环语句示例。

```
module DeMultiplexer (Address, Line);
  input [1:0] Address;
  output [3:0] Line;
  reg [3:0] Line;

  integer J;

  always @ (Address)
    for (J = 3; J >= 0; J = J - 1)
      if (Address == J)
        Line[J] = 1;
      else
        Line[J] = 0;
endmodule
// 综合出的网表如图 2-44 所示
```

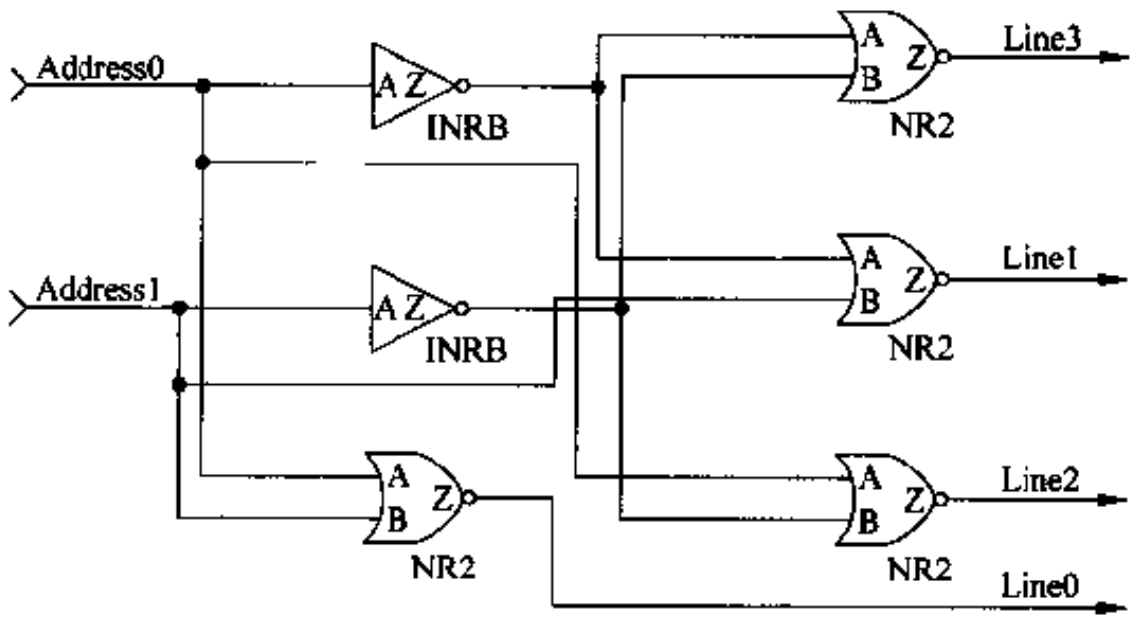

图 2-44　for 循环的示例

展开此 for 循环就得到以下 4 条 if 语句：

```
if (Address == 3) Line[3] = 1; else Line[3] = 0;
```

```
if (Address == 2) Line[2] = 1; else Line[2] = 0;
if (Address == 1) Line[1] = 1; else Line[1] = 0;
if (Address == 0) Line[0] = 1; else Line[0] = 0;
```

2.17　触发器的建模

在以下特定形式的 always 语句中对变量赋值会推导出触发器。

```
always @ (<时钟事件>)
  <语句>
```

其中*<时钟事件>*是以下两种形式之一：

```
posedge <时钟名>
negedge <时钟名>
```

always 语句的语义表明只在 *clock* 的上升沿或下降沿出现时才执行<语句>中的所有语句。这种特定的 always 语句被称为时钟控制的 *always* 语句。

对时序逻辑建模时，如果变量在时钟控制的 always 语句中被赋值并在该 always 语句之外引用了该变量的值，则建议对该变量采用非阻塞式过程赋值。这样可以防止设计模型与综合出的网表之间存在任何功能上的不一致。在时钟控制的 always 语句中非阻塞式赋值语句的赋值对象精确地模拟了时序元件的行为。

请看以下简单示例：

```
module PickOne (A, B, Clock, Control, Zee);
  input A, B, Clock, Control;
  output Zee;
  reg Zee;

  always @ (negedge Clock)
    if (Control)
      Zee <= A;
    else
      Zee <= B;
endmodule
// 综合出的网表如图 2-45 所示
```

可以看到，仅在时钟下降沿对输出 *Zee* 赋值，因此变量 *Zee* 被推导成下降沿触发的触发器。图中的触发器有一个数据选择，也就是说用输入 *Control* 从 *A* 或 *B* 中选择一个量作

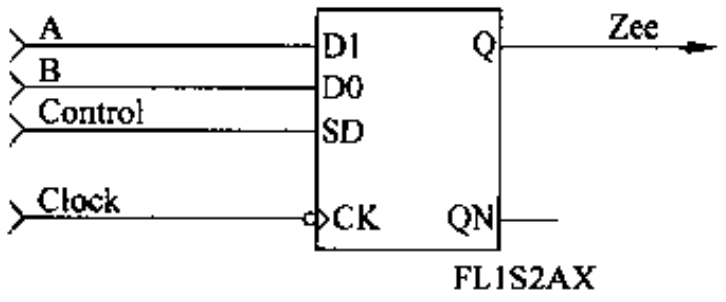

图 2-45　使用特定形式的 always 语句综合出的时序逻辑

为触发器的数据输入。

请再看一个示例：

```
module Incrementor (ClockA, Counter);
  parameter COUNTER_SIZE = 2;
  input ClockA;
  output [COUNTER_SIZE - 1:0] Counter;
  reg [COUNTER_SIZE - 1:0] Counter;

  always @ (posedge ClockA)
    Counter <= Counter + 1;
endmodule
// 综合出的网表如图 2-46 所示
```

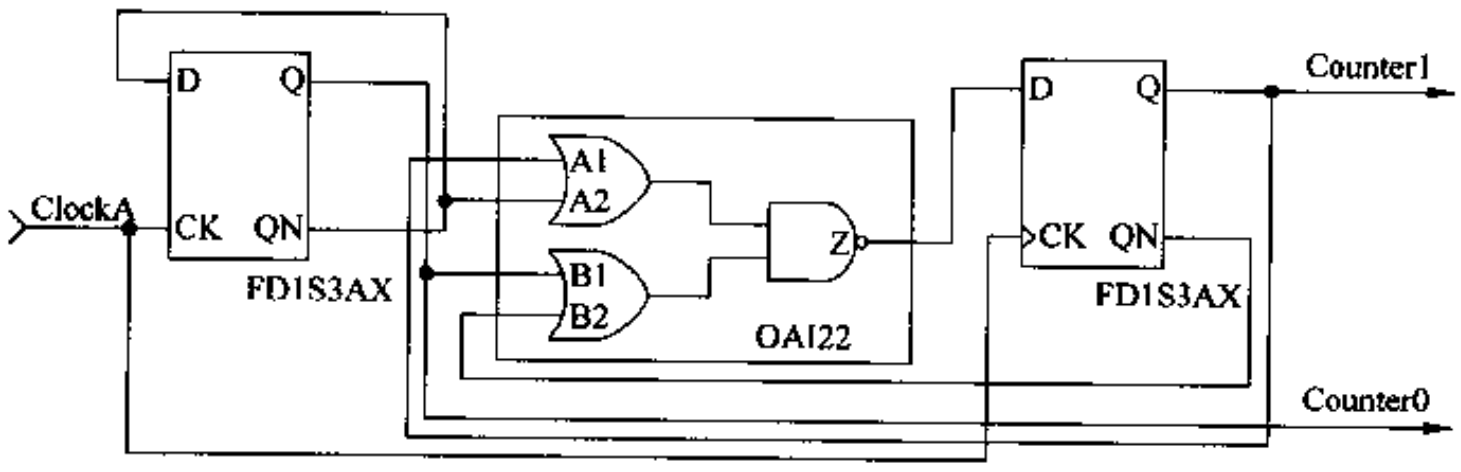

图 2-46　触发器的建模

此 always 语句的逻辑表明 *ClockA* 每出现一次上升沿，变量 *Counter* 就自动加 1。因为 *Counter* 在时钟沿的控制下被赋值，因此把它综合成上升沿触发的触发器。

以下可逆计数器模型用来说明触发器的建模。

```
module UpDownCounter (Control, ClockB, Counter);
  input Control, ClockB;
  output [1:0] Counter;
  reg [1:0] Counter;
```

```
    always @ (negedge ClockB)
      if (Control)
        Counter <= Counter + 1;
      else
        Counter <= Counter - 1;
endmodule
// 综合出的网表如图 2-47 所示
```

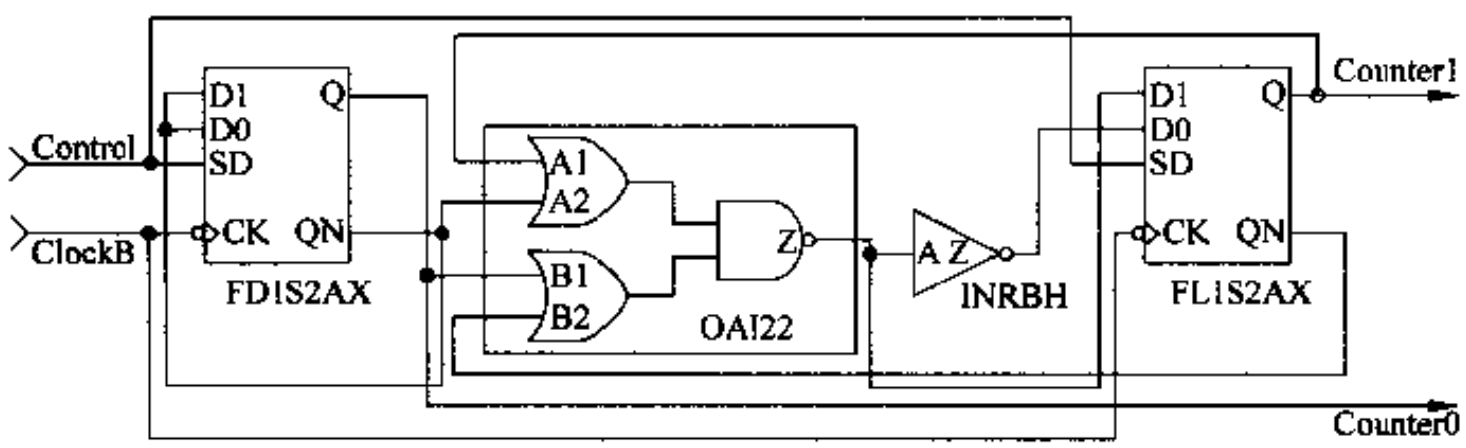

图 2-47　推导出下降沿触发的触发器

变量 *Counter* 在时钟 *ClockB* 下降沿的控制下被赋值，因此它被综合成两个下降沿触发的触发器。

触发器推导规则很简单：如果变量在时钟沿的控制下被赋值，则生成触发器。但有一个例外，若对变量的赋值和引用都仅出现在同一条 always 语句中，则该变量会被视作中间变量，而不会生成触发器。

请再看另一个示例：

```
module FlipFlop (Clk, CurrentState. NextState);
  input Clk;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (posedge Clk)
    NextState <= CurrentState;
endmodule
// 综合出的网表如图 2-48 所示
```

此例中，变量 *NextState* 仅在 *Clk* 上升沿被赋值。于是，寄存器变量 *NextState* 被推导成 4 个上升沿触发的触发器(用于保存 0 至 15 的值)。

如果想推导出下降沿触发的触发器，则须将时钟沿事件“**posedge** *clk*”改成“**negedge**

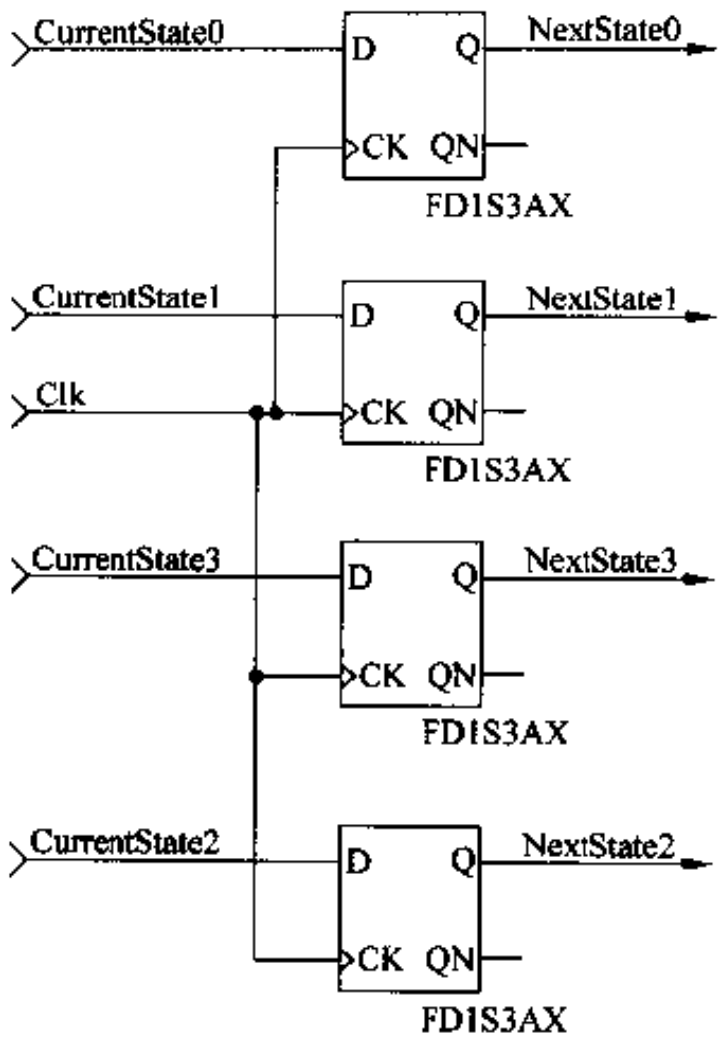

图 2-48 时钟控制下的变量被推导成触发器

Clk"。

在时钟沿的控制下对整型变量赋值也会被推导成触发器。下例中在时钟控制下对整型变量进行赋值。变量 *IntState* 被推导成 4 个触发器，变量的其他高位部分被优化掉了(因为它们未被引用)。

```
module FlipFlopInt (Clk, CurrentState,
NextState);
  input Clk;
  input [3:0] CurrentState;
  output [3:0] NextState;

  integer IntState;

  always @ (posedge Clk)
    IntState <= CurrentState;

  assign NextState = IntState;
endmodule
// 综合出的网表和图 2-48 相同
```

变量的局部引用

以上各示例都是在时钟的控制下对变量赋值，又在实现赋值的 always 语句之外引用了该变量的值，这样就需要将该变量的值保存在触发器中。

如果全局定义(即在 always 语句之外定义)的变量仅在某一条 always 语句中被引用了，结果又会如何？请看下例：

```
module GlobalReg (Clk, CurrentState, NextState);
  input Clk;
  input [3:0] CurrentState;
  output [3:0] NextState;
reg [3:0] NextState;

reg [3:0] Temp;

always @ (negedge Clk)
  begin
    Temp = CurrentState;
```

```
    NextState <= Temp;
  end
endmodule
```

尽管对 *Temp* 的赋值处于时钟控制之下，但是并没有把它推导成触发器，这是因为它在同一个时钟周期中先被赋值、再被引用。综合出的网表如图 2-48 所示。此时，*Temp* 仅用作临时量(即中间变量)。因此，应当采用阻塞式赋值来反映这样的事实——第二条语句中引用的 *Temp* 值其实是第一条语句中对 *Temp* 所赋的值。因为要把 *NextState* 推导成触发器，所以要对 *NextState* 采用非阻塞式赋值。

如果颠倒上例的语句次序，结果又会如何？此时，因为 *Temp* 在赋值之前就被引用了，其值需要在多个时钟周期内保持，因此把它推导成触发器。*Temp* 模拟了 always 语句的内部状态。下例正是如此，其中 *Temp* 在赋值前就被引用了。

```
module RegUseDef (Clk, CurrentState, NextState);
  input Clk;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;
  reg [3:0] Temp;

  always @ (negedge Clk)
  begin
    NextState <= Temp;
    Temp = CurrentState;
  end
endmodule
// 综合出的网表如图 2-49 所示
```

此时，变量 *Temp* 被推导成下降沿触发的触发器，*NextState* 也是如此。

如果变量是在 always 语句内通过局部声明得到的，结果又会如何？

在 always 语句内局部声明的变量(包括寄存器类型或整型)不会被推导成触发器。这可能会导致 Verilog HDL 模型和综合出的网表功能不一致。下例中，局部声明的变量 *Temp* 没有被推导成触发器。

```
module LocalVarAssignUse (Clk, CurrentState, NextState);
  input Clk;
  input [3:0] CurrentState;
  output [3:0] NextState;
```

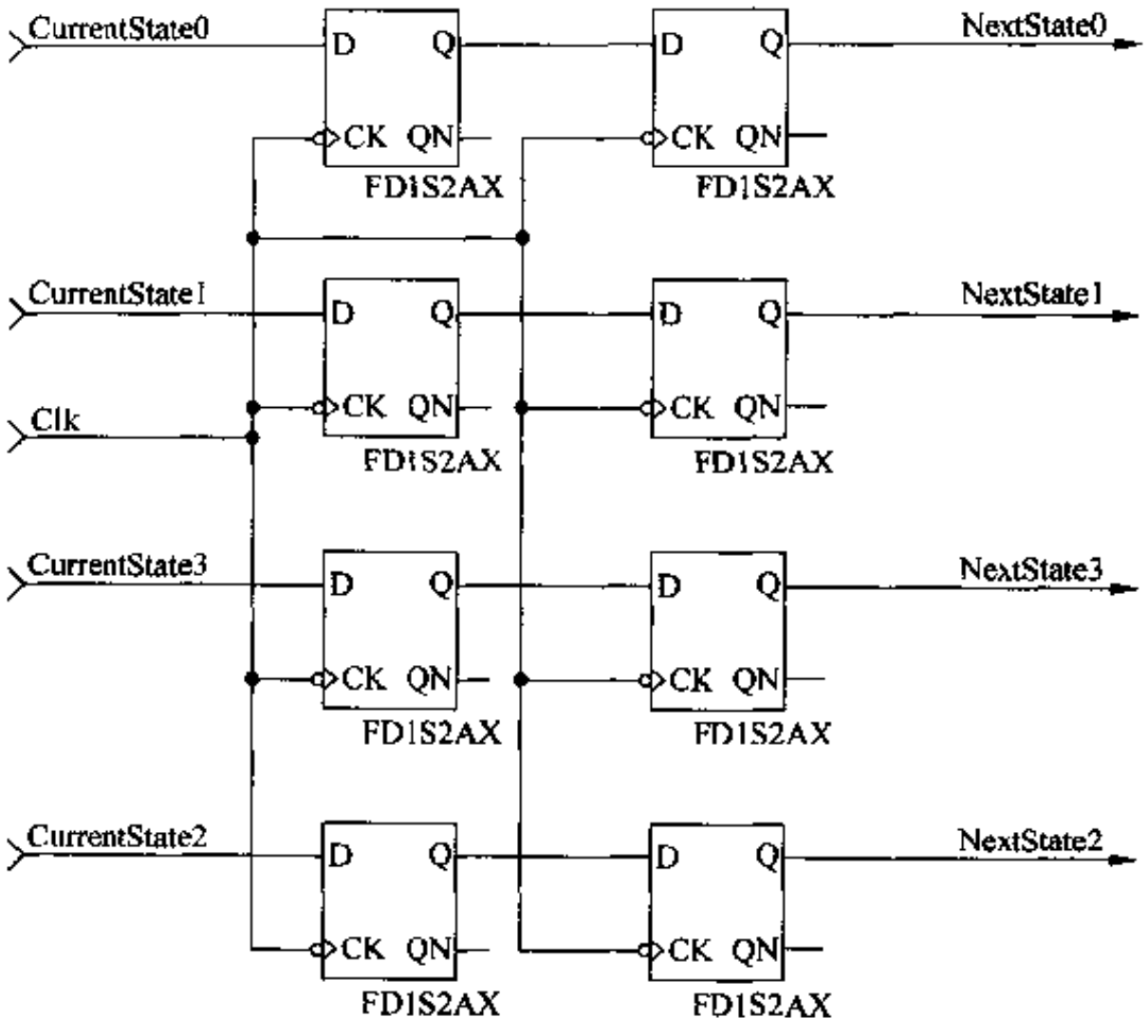

图 2-49　在定义前被引用的变量被推导成触发器

```
  reg [3:0] NextState;

  always @ (posedge Clk)
  begin: Label1A
    reg [3:0] Temp;

    Temp = CurrentState;
    NextState <= Temp;
  end
endmodule
```

变量 *Temp* 未被推导成触发器，这是因为它是在 always 语句中局部声明的变量，而且赋给该变量的值随后在同一个时钟沿上即被引用了。而 *NextState* 被推导成触发器，是因为它在给它赋值的 always 语句之外被引用了。综合出的网表与图 2-48 相同。

但是，如果颠倒上述语句的次序，则设计模型与综合出的网表就可能不一致了，这是因为没有把局部声明的变量推导成触发器。请看以下模型：

```
module LocalVarUseAssign (Clk, CurrentState, NextState);
  input Clk;
```

```
input [3:0] CurrentState;
output [3:0] NextState;
reg [3:0] NextState;

always @ (posedge Clk)
begin: LabelA
  reg [3:0] Temp;

  NextState <= Temp;
  Temp = CurrentState;
end
endmodule
```

综合出的网表与图 2-48 相同。注意：每个时钟沿上 *NextState* 都得到 *Temp* 在前一个时钟周期被赋的值，但是综合出的网表并非如此。由此建议，避免以这样的方式使用局部声明的变量。但愿综合工具能在 *Temp* 未被综合成触发器时发出警告信息。

2.17.1　多个时钟

模块中可能有多个受时钟控制的 always 语句。下例中的模型使用了多个时钟。

```
module MultipleClocks (Vt15Clock, AddClock, AdN, ResetN, SubClr, SubN, Ds1Clock, Ds1Add,
      Ds1Sub);
  input Vt15Clock, AddClock, AdN, ResetN, SubClr, SubN, Ds1Clock;
  output Ds1Add, Ds1Sub;
  reg Ds1Add, Ds1Sub;
  reg AddState, SubState;

  always @ (posedge Vt15Clock)
  begin
    Addstate <= AddClock ^ ~ (AdN | ResetN);
    SubState <= SubClr ^ (SubN & ResetN);
  end

  always @ (posedge Ds1Clock)
  begin
    Ds1Add <= AddState;
    Ds1Sub <= SubState;
  end
endmodule
```

// 综合出的网表如图 2-50 所示

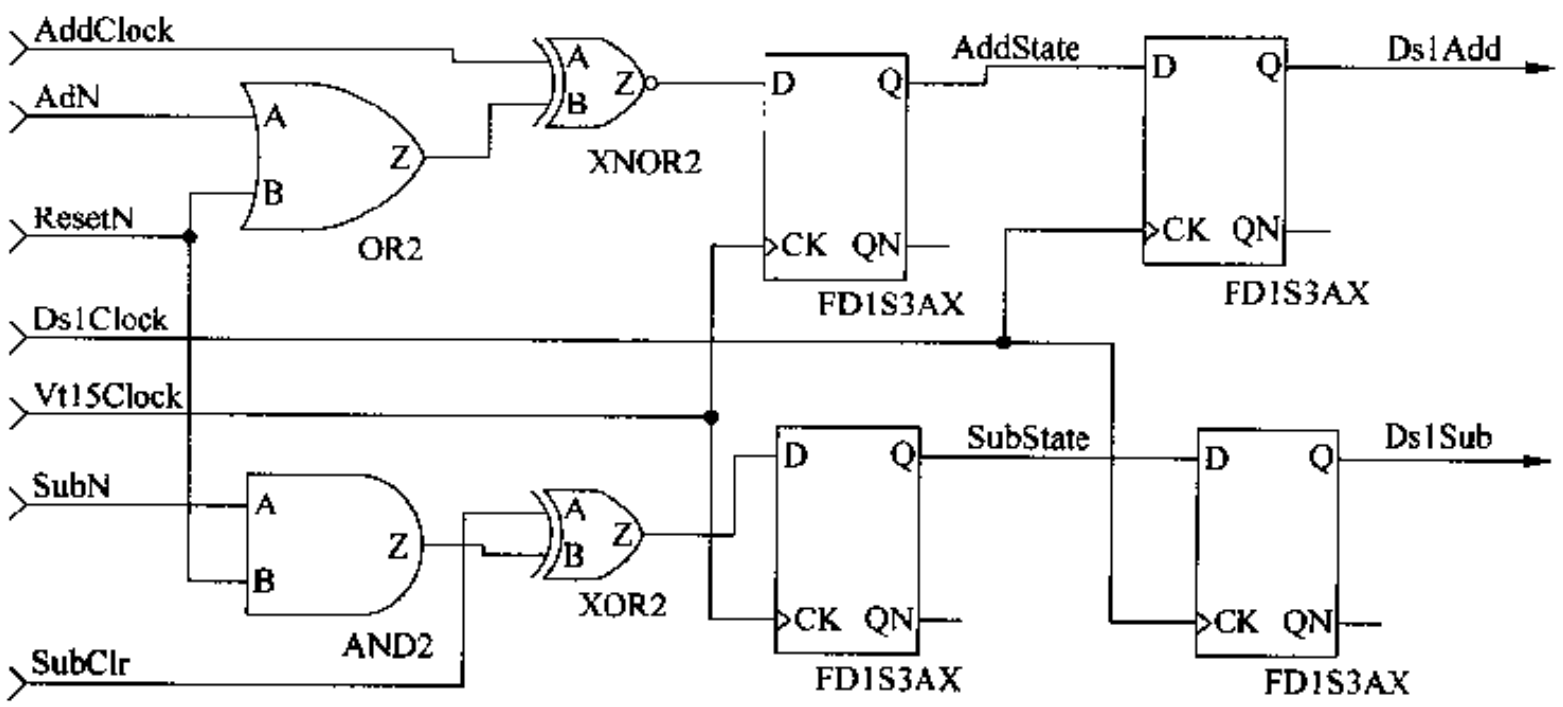

图 2-50 always 语句中有多个时钟

此模块有两条 always 语句。第一条 always 语句中的各语句受 *Vt15Clock* 的上升沿控制，而第二条 always 语句中的各语句受 *Ds1Clock* 的上升沿控制。

综合系统通常强行限制，对变量的赋值不能受多个时钟控制。例如，在第二条 always 语句中对 *AddState* 的赋值是非法的。

2.17.2 多相位时钟

有可能一个模块中出现多个受时钟控制的 always 语句，这些 always 语句使用同一时钟的不同边沿。下例就使用了同一时钟的两个不同相位。

```
module MultiphaseClocks (Clk, A, B, C, E);
  input Clk, A, B, C;
  output E;
  reg E, D;

  always @ (posedge Clk)
    E <= D | C;

  always @ (negedge Clk)
    D <= A & B;
endmodule
```

// 综合出的网表如图 2-51 所示

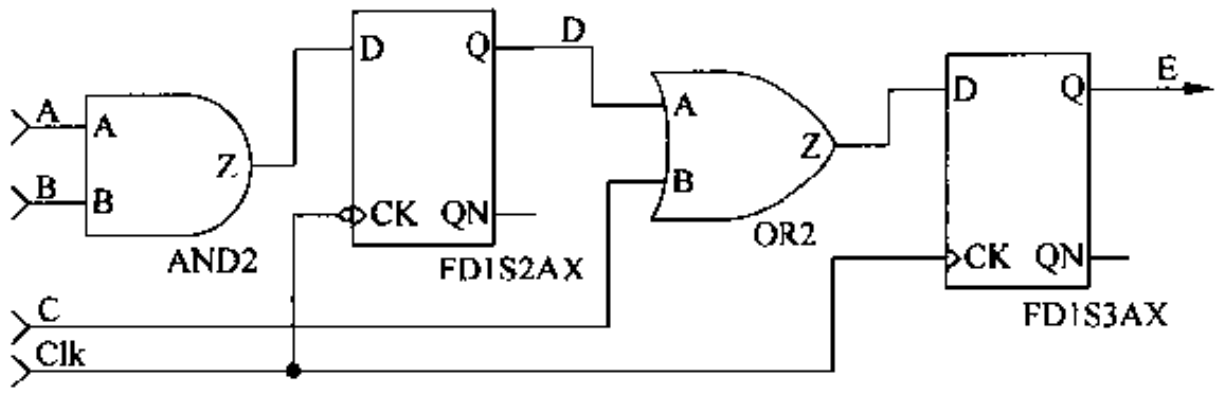

图 2-51　一个模块中同一时钟的不同沿

此模块中，第一条 always 语句中的各语句受 *Clk* 的上升沿控制，而第二条 always 语句中的各语句受 *Clk* 的下降沿控制。

在这种情况下，综合系统通常强行限制，对变量的赋值不能受两种不同的时钟条件（或者不同的时钟沿）控制。例如，在第一条 always 语句中对 *D* 的赋值是非法的。

2.17.3　使用异步预置位与清零

到目前为止，已论及简单的 D 型触发器的综合。如何才能推导出带异步预置位和清零的触发器呢？要生成这样的触发器，需使用一种特定形式的 if 语句。以下示例模板很好地说明了其用法。

```
always @ (posedge A or negedge B or negedge C... or posedge Clock)
  if(A)                          // posedge A
    < statement >                // 异步逻辑
  else if ( ! B)                 // negedge B
    < statement >                // 异步逻辑
  else if ( ! C)                 // negedge C
    < statement >                // 异步逻辑
  ...                            // 任意多个 else if
  else                           // 隐含的 posedge Clock
    < statement >                // 同步逻辑
```

always 语句事件表（符号“@”后用括号括起来的表）可以有任意多个边沿事件，既可以有上升沿（**posedge**），也可以有下降沿（**negedge**）。其中必须有一个时钟事件，其余的事件用于指定异步逻辑执行的条件。always 语句内仅有一条 if 语句，该 if 语句有若干个 else if 分支，每一个 if 都对应于事件表中的一个边沿事件，最后一个 else 隐含地对应于时钟沿事件。各 if 语句的条件必须与事件表中指定的边沿类型一致。例如，如果事件表中有“**posedge** A”，则相应的 if 语句应是 if (A)；如果事件表中有“**negedge** B”，则相应的 if 语句

应是 if (! B)。每个 if 分支(最后一个除外)中的语句都用于表达异步逻辑,而最后一个 else 分支用于表达同步逻辑。

如果变量在异步逻辑部分和同步逻辑部分都被赋值了,则该变量应综合成带异步预置位和清零的触发器。根据所赋的值不同,综合出的触发器可能是带异步预置位的触发器(被赋了非零值),也可能是带异步清零的触发器(被赋了零值),还可能是两者都有的触发器。

请看以下带异步预置位和清零的可逆计数器示例:

```
module AsyncPreClrCounter (Clock, Preset, UpDown, Clear, PresetData, Counter);
  parameter NUM_BITS = 2;
  input Clock, Preset, UpDown, Clear;
  input [NUM_BITS - 1:0] PresetData;
  output [NUM_BITS - 1:0] Counter;
  reg [NUM_BITS - 1:0] Counter;

  always @ (posedge Preset or posedge Clear or posedge Clock)
    if (Preset)
      Counter <= PresetData;
    else if (Clear)
      Counter <= 0;
    else                        // 隐含的 posedge Clock
    begin                       // 同步部分
      if (UpDown)
        Counter <= Counter + 1;
      else
        Counter <= Counter - 1;
    end
endmodule
// 综合出的网表如图 2-52 所示
```

像 *PresetData* 那样的异步数据输入端可能会引起问题。假定 *Preset* 为 1,随后 *PresetData* 发生变化。Verilog HDL 模型中不会反映出 *PresetData* 的这种变化,而在综合出的网表中这种变化却会波及 *Counter*。请避免或者谨慎使用异步数据输入。

请再看一个推导出带异步预置位和清零的触发器示例。

```
module AsyncFlipFlop (ClkA, Reset, Set, CurrentState, NextState);
  input ClkA, Reset, Set;
  input [3:0] CurrentState;
  output[3:0] NextState;
```

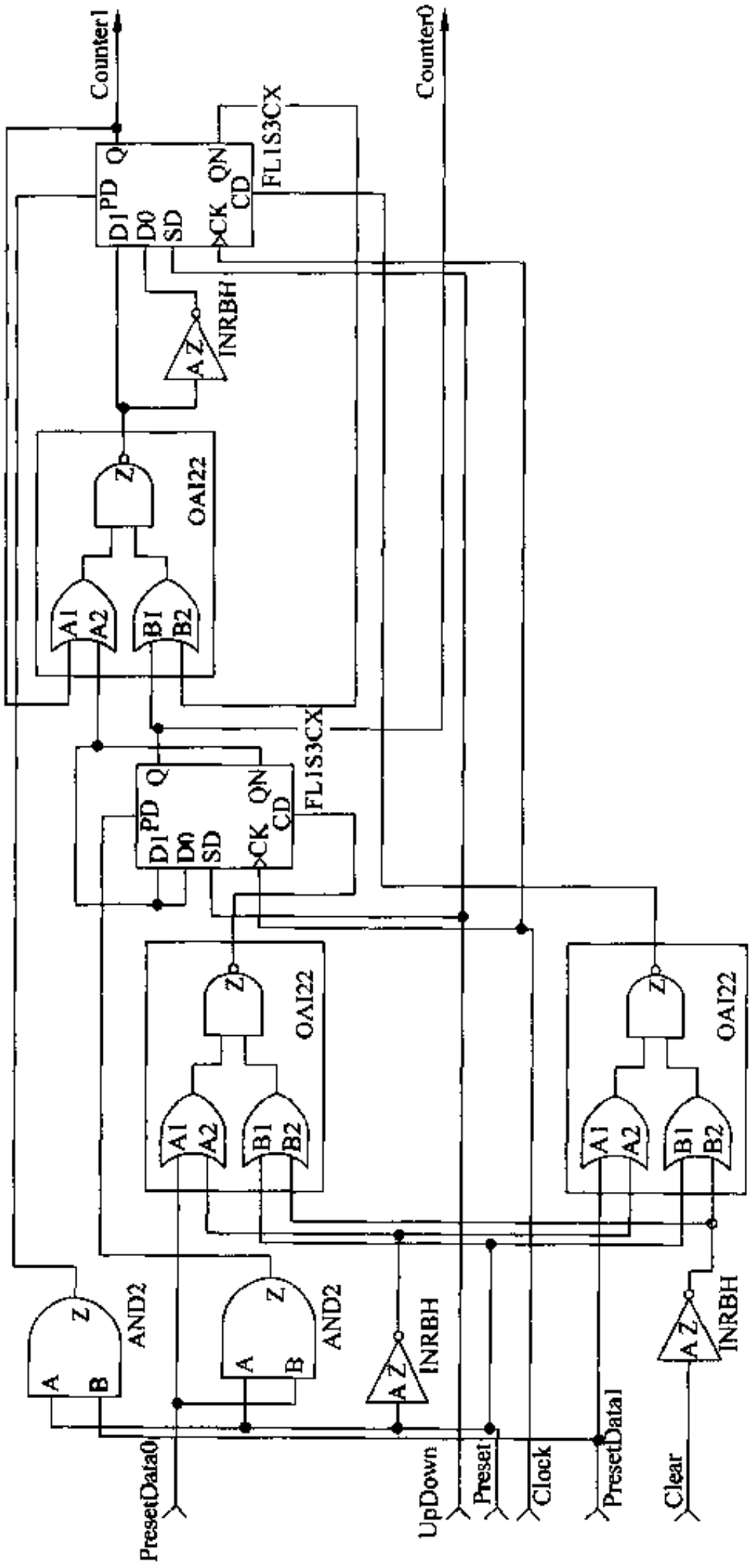

图 2-52　带异步预置位和清零的触发器

```
    reg [3:0] NextState;
    always @ (negedge Reset or negedge Set or negedge ClkA)
      if (! Reset)
        NextState <= 12;                          // 语句 A
      else if (! Set)
        NextState <= 5;                           //语句 B
      else
        NextState <= CurrentState;                //语句 C
endmodule
// 综合出的网表如图示 2-53 所示
```

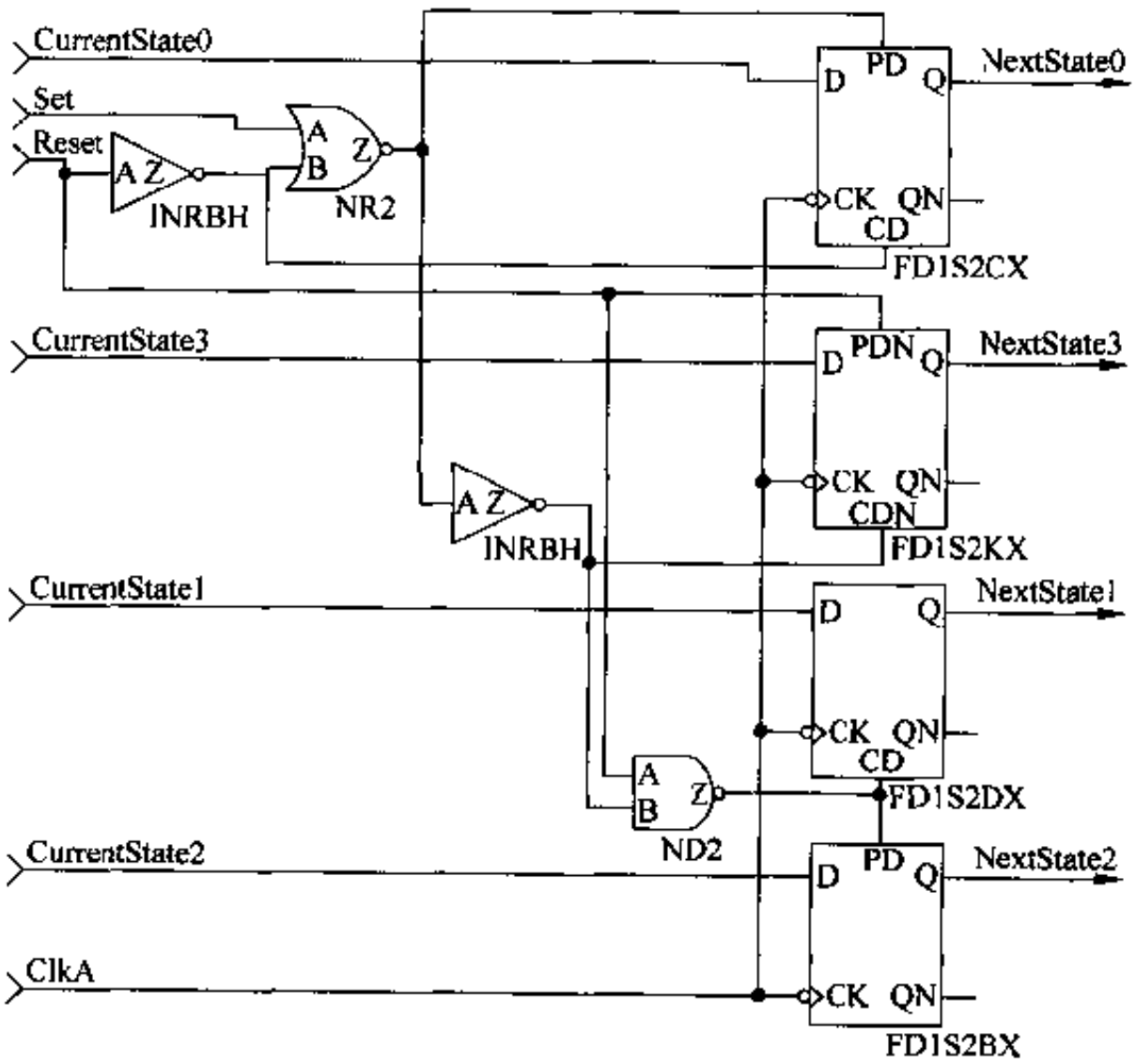

图 2-53 带异步预置位和清零的触发器

因为对 *NextState* 的赋值不仅受时钟沿的控制(语句 C),而且存在着异步赋值(语句 A 和语句 B),因此综合出了带异步预置位和清零的下降沿触发的触发器,如图 2-53 所示。注意:这里要用到 4 个触发器。第一个触发器对应于 *NextState* 的最高位,带有预置位和清零这两个异步端,这是因为它需要在 *Reset* 有效时被置位,在 *Set* 有效时被清零。类似地,第四个寄存器也带有预置位和清零这两个异步端,这是因为它需要在 *Set* 有效时被置位,在 *Reset* 有效时被清零。第二个触发器仅有一个预置位端,因为这两种情况下都使用

'b1 进行异步赋值；而第三个触发器仅有一个清零端，因为这两种情况下都使用'b0 进行异步赋值。

2.17.4　使用同步预置位和清零

如何模拟带同步预置位和清零的触发器呢？此时，只需要在时钟控制的 always 语句中描述同步预置位和清零逻辑就行了。请看下例：

```
module SyncPresetCounter (Clock, Preset, UpDown, PresetData, Counter);
  parameter NBITS = 2;
  input Clock, Preset, UpDown;
  input [0:NBITS - 1] PresetData;
  output [0:NBITS - 1] Counter;
  reg [0:NBITS - 1] Counter;

  always @ (negedge Clock)
    if (Preset)
      Counter <= PresetData;
    else
      if (UpDown)
        Counter <= Counter + 1;
      else
        Counter <= Counter - 1;
endmodule
// 综合出的网表如图 2-54 所示
```

有两种方式综合此模型。一种方式是把输入 *PresetData* 引到触发器的同步预置位输入端，另一种方法是把 *PresetData* 直接引到触发器的数据输入端。这里综合出的网表是采

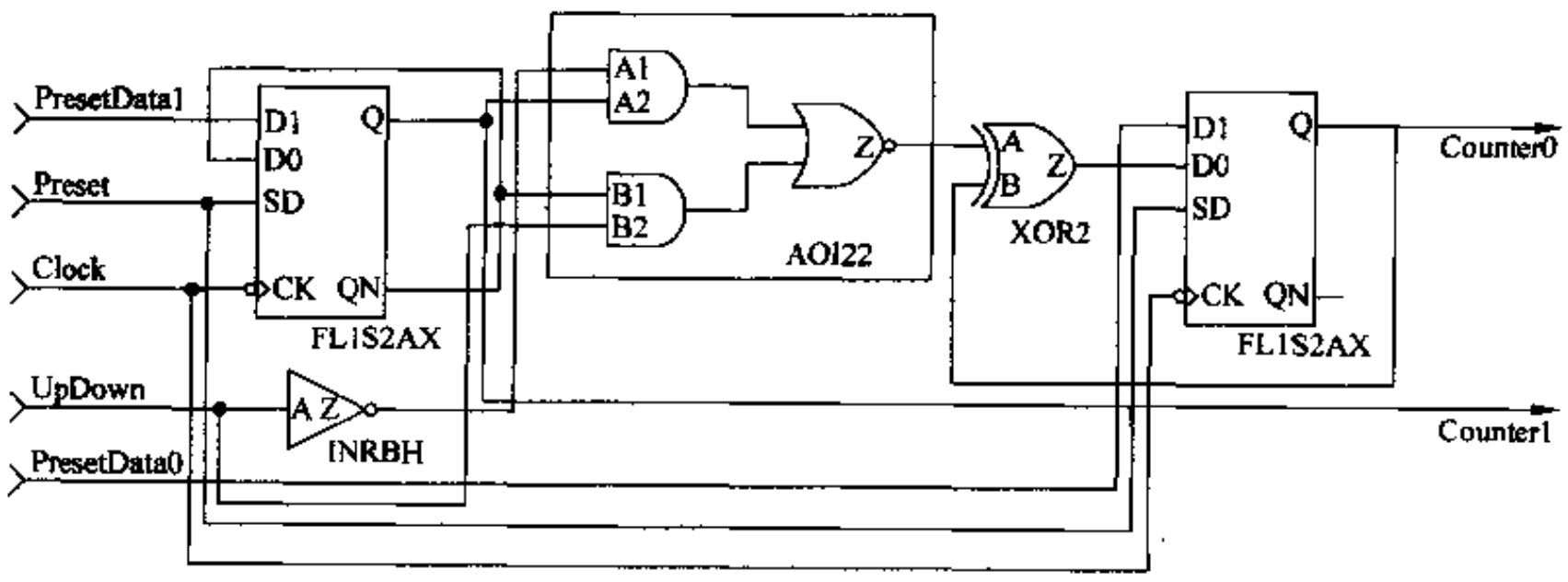

图 2-54　同步预置位清零逻辑被综合成组合逻辑

用第二种方式得到的，综合系统可以选择实现综合的方式。

请再看一个示例：

```
module SyncFlipFlop (ClkB, Reset, Set, CurrentState, NextState);
  input ClkB, Reset, Set;
  input [3:0] CurrentState;
  output [3:0] NextState;
  reg [3:0] NextState;

  always @ (negedge ClkB)
    if (! Reset)
      NextState <= 12;
    else if (! Set)
      NextState <= 5;
    else
      NextState <= CurrentState;
endmodule
// 综合出的网表如图 2-55 所示
```

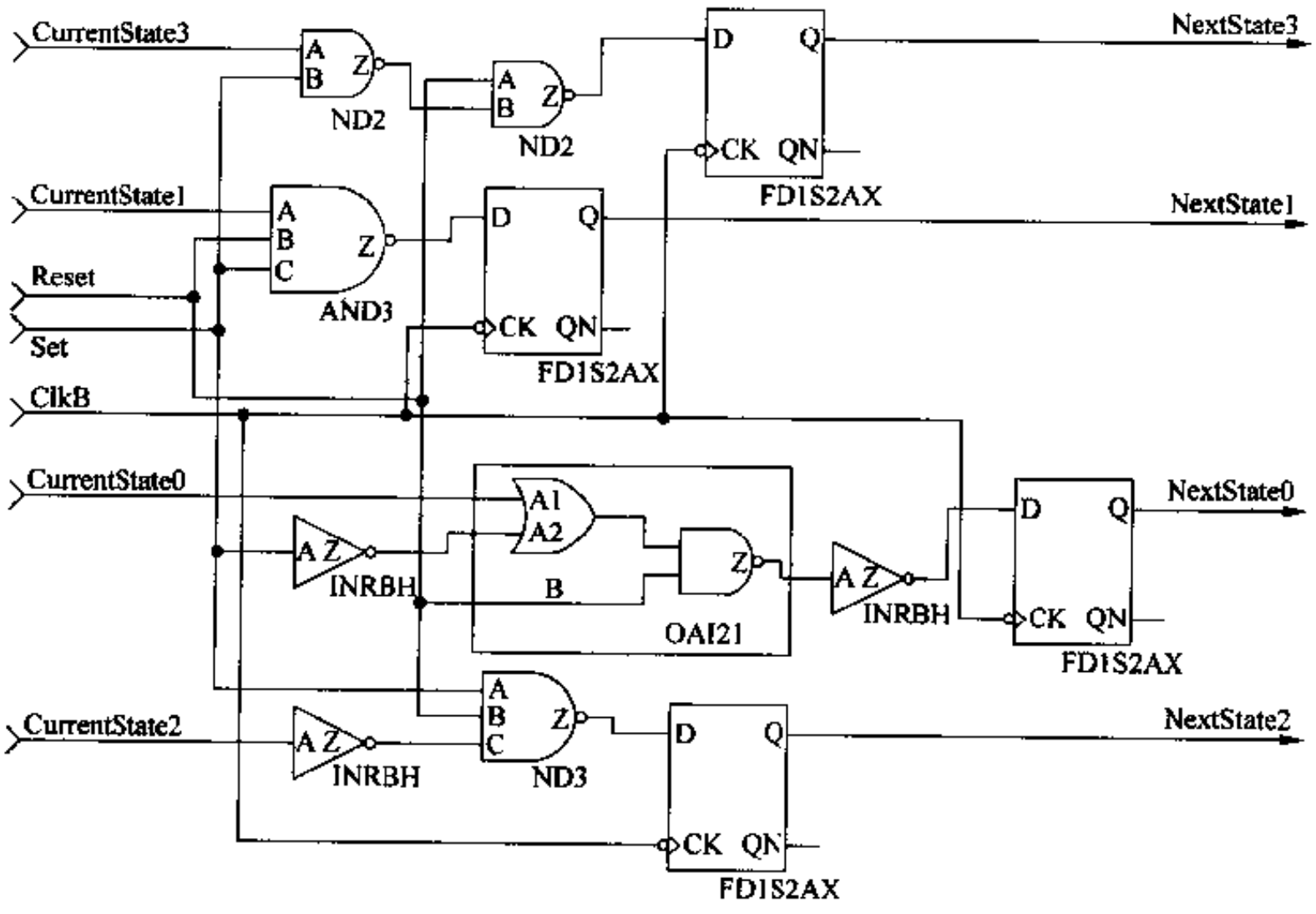

图 2-55　并非带同步预置位和清零的触发器

此例中，数值 12、5 以及变量 *CurrentState* 都是 *NextState* 的输入，应该让它们经过适当的选择线，再把多路选择结果送至由 *NextState* 综合出的触发器的 D 输入端。图 2-55 所示的网表正是如此。那么，怎样才能推导出带同步预置位和清零的触发器呢？综合系统可以通过提供某种选择机制来引导系统生成带同步预置位清零的触发器。

2.18　再谈阻塞式和非阻塞式赋值

在前面的章节中，我们建议对时序逻辑建模只使用非阻塞式过程赋值（除非在用作中间变量时应使用阻塞式赋值），而对组合逻辑仅使用阻塞式过程赋值。本书中的各示例都遵循这些建议，而本节将混合使用阻塞式和非阻塞式赋值，以阐明这两种过程赋值语句在用于综合时的语义差别。

综合时处理阻塞式赋值和非阻塞式赋值的方式不同，这是由于它们之间存在语义差别。对于阻塞式赋值，执行顺序块中下一条语句之前就已完成对左端对象的赋值（如果在 always 语句中仅有一条语句，则阻塞式赋值和非阻塞式赋值之间没有差别）。对于非阻塞式赋值，对左端对象的赋值被安排在该仿真周期结束时（即赋值并不立即生效）、执行下一条语句之前。请看下例：

```
module FlagBits (ClockB, Strobe, Xflag, Mask, RightShift, SelectFirst, CheckStop);
  input ClockB, Strobe, Xflag, Mask;
  output RightShift, SelectFirst, CheckStop;
  reg RightShift, SelectFirst, CheckStop;

  always @ (negedge ClockB)
  begin
    RightShift = RightShift & Strobe;     // 阻塞式
    SelectFirst <= RightShift | Xflag;    // 非阻塞式
    CheckStop <= SelectFirst ^ Mask;      // 非阻塞式
    end
  endmodule
// 综合出的网表如图 2-56 所示
```

此 always 语句含有一个带 3 条赋值语句的顺序块，第一条是阻塞式过程赋值，后两条是非阻塞式赋值。因为所有赋值行为都受时钟沿控制，所以 *RightShift*、*SelectFirst* 和 *CheckStop* 都被综合成下降沿触发的触发器。但是，信息连接到这些触发器的数据输入端的方式是不同的。由于对 *RightShift* 的赋值是阻塞式赋值，而第二条赋值语句引用的是其新值，这意味着 *RightShift* 触发器的数据输入端应通过逻辑门连接到 *SelectFirst* 触

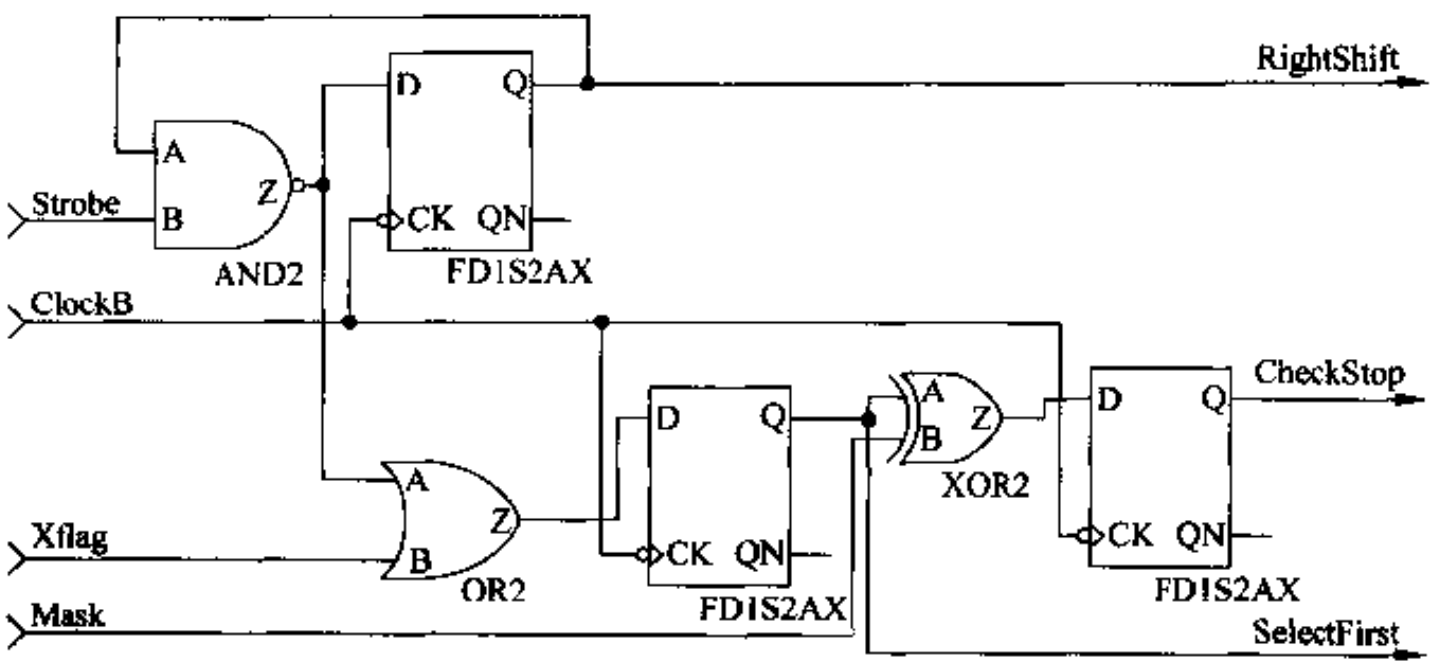

图 2-56 非阻塞式和阻塞式过程赋值

发器的数据输入端。由于 *SelectFirst* 是非阻塞式赋值，第三条赋值语句引用的是其原值，而不是在第二条赋值语句中对其所赋的新值。结果是，*SelectFirst* 触发器的输出（指其原值）被送至 *CheckStop* 触发器的数据输入端。对这些差别的解释可以由图 2-56 所示的综合出的网表得到证实。

请再看一个示例，它突显了阻塞式和非阻塞式赋值用于综合时的差异。

```
module NonBlockingExample (ClockZ, Merge, ER, Xmit, FDDI, Claim);
  input ClockZ, Merge, ER, Xmit, FDDI;
  output Claim;
  reg Claim;
  reg FCR;

  always @ (posedge ClockZ)
  begin
    FCR <= ER | Xmit;              // 第一条赋值语句
    if (Merge)
      Claim <= FCR & FDDI;         // 第二条赋值语句
    else
      Claim <= FDDI;
  end
endmodule
// 综合出的网表如图 2-57 所示
```

此示例中有 3 条赋值语句。顺序块中的语句是顺序执行的。但是，非阻塞式赋值的赋值对象总是在未来（即当前仿真时刻结束时）被赋值。因此，为了与非阻塞式赋值的语义保

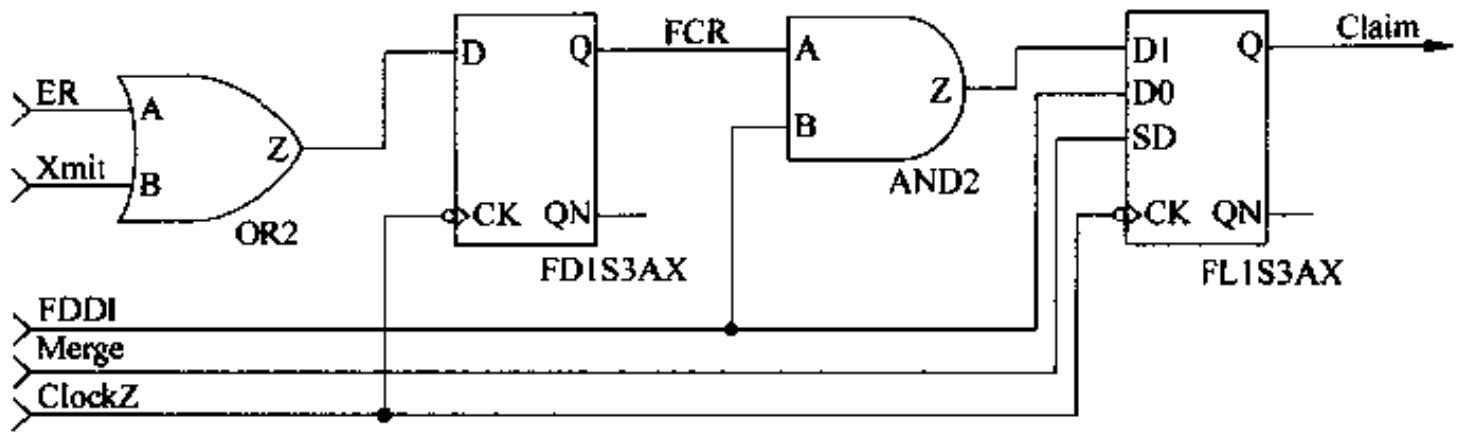

图 2-57　非阻塞式赋值

持一致，第二条赋值语句所引用的 *FCR* 是其原值，而不是第一条赋值语句中对其所赋的新值。这样，在综合出的网表中，触发器 *FCR* 的输出被送到产生 *Claim* 逻辑的电路中。

此例若采用阻塞式过程赋值(如下所示)，又会如何？

```
module BlockingExample (ClockZ, Merge, ER, Xmit, FDDI, Claim);
  input ClockZ, Merge, ER, Xmit, FDDI;
  output Claim;
  reg Claim;

  reg FCR;

  always @ (posedge ClockZ)
  begin
    FCR = ER | Xmit;              // 第一条赋值语句
    if (Merge)
      Claim = FCR & FDDI;         // 第二条赋值语句
    else
      Claim = FDDI;
  end
endmodule
// 综合出的网表如图 2-58 所示
```

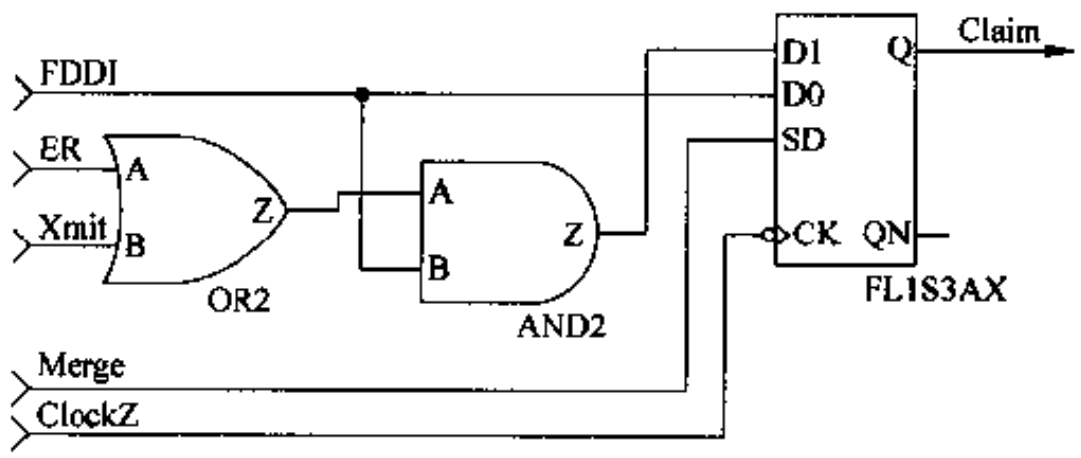

图 2-58　阻塞式赋值

对于该例，必须在执行第二条赋值语句之前就完成对 *FCR* 的赋值。因此，为了模拟这样的语义，第一条赋值语句的右端表达式必须用来构造 *Claim* 的数据输入端逻辑，综合出的网表正是如此。此外，没有把 *FCR* 推导成触发器，这是因为它先被赋值然后才被引用，在 always 语句的不同循环中不必保存它的值。

建议对时序逻辑采用非阻塞式赋值，而对组合逻辑采用阻塞式赋值，第 5 章将解释此建议背后的基本原理；这是为了防止 Verilog HDL 模型与综合出的网表之间出现功能上的不一致。注意：这仅仅是个建议，很多情况下，只要能够理解它们之间的语义差别，随便使用哪一种赋值方式都行。

2.19 函　　数

函数调用表达的是组合逻辑，这是因为 Verilog HDL 中函数调用是表达式的一部分。函数调用是通过把被调用的函数直接插入调用代码中这种展开方式来实现综合的。在函数内局部声明的任何变量都被当作纯粹的临时量来处理，这样的变量会被综合成连线。

请看以下函数调用示例：

```
module FunctionCall (XBC, DataIn);
  output XBC;
  input [0:5] DataIn;

  function [0:2] CountOnes;
    input [0:5] A;
    integer K;
    begin
      CountOnes = 0;

      for (K = 0; K <= 5; K = K + 1)
        if (A[K])
          CountOnes = CountOnes + 1;
    end
  endfunction
  // 如果 DataIn 中有两个以上 1，则向 XBC 返回 1
  assign XBC = CountOnes (DataIn) > 2;
endmodule
// 综合出的网表如图 2-59 所示
```

对函数调用直接插入展开，进而对 for 循环语句直接插入展开，就能得到以下代码：

```
CountOnes = 0;
```

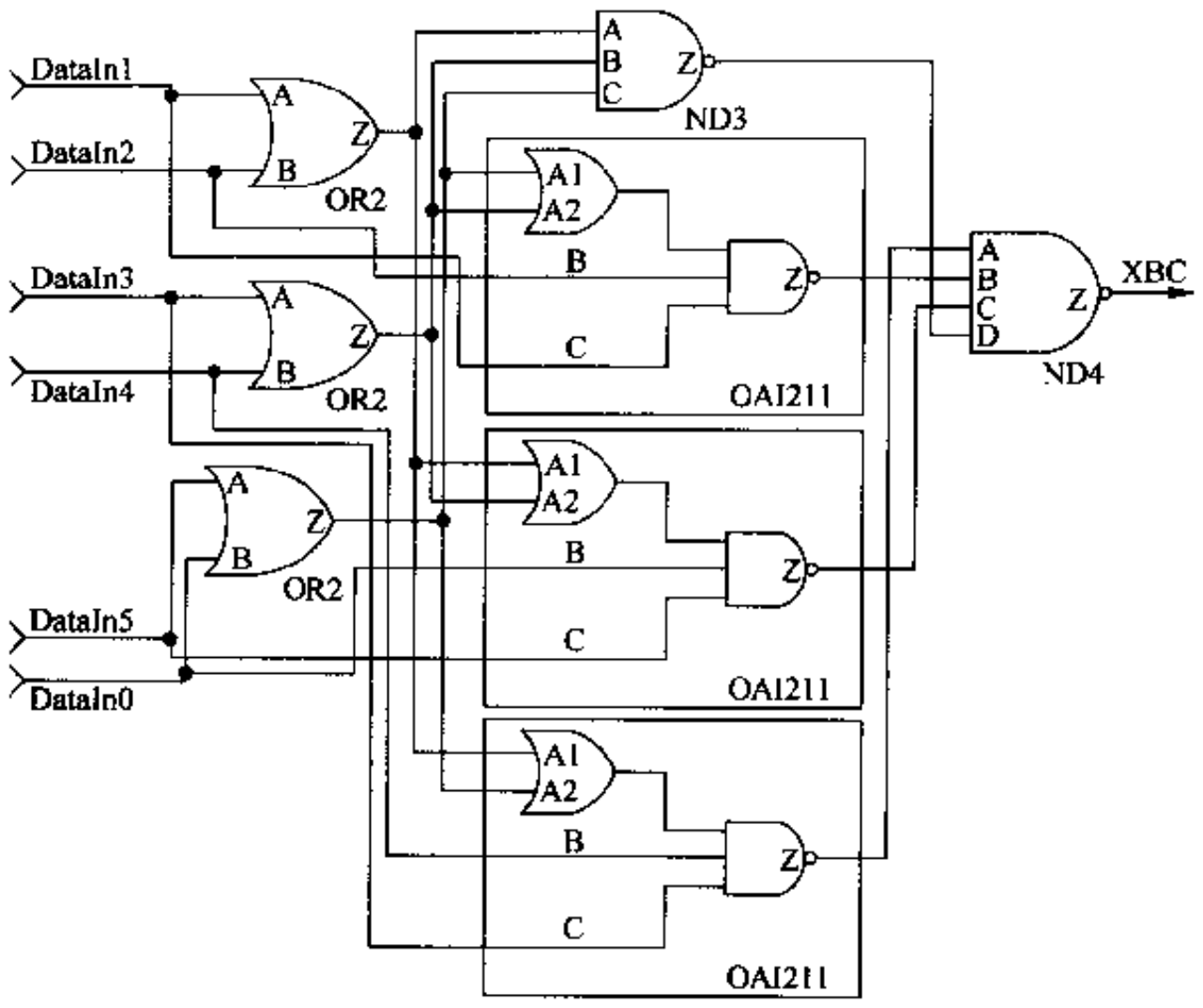

图 2-59　函数调用的示例

```
if (DataIn[0]) CountOnes = CountOnes + 1;
if (DataIn[1]) CountOnes = CountOnes + 1;
if (DataIn[2]) CountOnes = CountOnes + 1;
if (DataIn[3]) CountOnes = CountOnes + 1;
if (DataIn[4]) CountOnes = CountOnes + 1;
if (DataIn[5]) CountOnes = CountOnes + 1;
XBC = CountOnes > 2;
```

2.20　任　　务

任务调用既可以表达组合逻辑，也可以表达时序逻辑，这取决于任务调用出现的上下文环境。这样，任务调用的输出参数可能会成为存储器件，取决于对参数赋值的上下文环境。例如，若时钟控制的 always 语句(即带时钟事件的 always 语句)中有任务调用，则该任务调用的输出参数应被综合成触发器，这是由触发器推导规则决定的。综合系统通过把被调用的任务直接插入其余代码中实现任务调用，事实上并未将任务调用作为一个独立的结构层次来处理。

请看以下表示纯组合逻辑的任务调用示例：

```
module CombTask (ShA, ShB, ShCarryIn, ShSum, ShCarryOut);
  input [0:2] ShA, ShB;
  input ShCarryIn;
  output [0:2] ShSum;
  output ShCarryOut;
  reg [0:3] TempCarry;

  task AddOneBitWithCarry;
    input A, B, CarryIn;
    output Sum, CarryOut;
    begin
      Sum = A ^ B ^ CarryIn;
      CarryOut = A & B & CarryIn;
    end
  endtask

  always @ (ShA or ShB or ShCarryIn)
  begin: EXAMPLE
    integer J;
    TempCarry[0] = ShCarryIn;

    for (J = 0; J < 3; J = J + 1)
      AddOneBitWithCarry (ShA[J], ShB [J], TempCarry[J], ShSum[J], TempCarry[J + 1]);
  end

  assign ShCarryOut = TempCarry[3];
endmodule
// 综合出的网表如图 2-60 所示
```

综合工具对此任务调用和 for 循环直接插入展开，就得到以下代码：

```
TempCarry[0] = ShCarryIn;
ShSum[0] = ShA[0] ^ ShB[0] ^ TempCarry[0];
TempCarry[1] = ShA [0] & ShB[0] & TempCarry[0];
ShSum[1] = ShA[1] ^ ShB[1] ^ TempCarry[1];
TempCarry[2] = ShA [1] & ShB[1] & TempCarry[1];
ShSum[2] = ShA[2] ^ ShB[2] ^ TempCarry[2];
TempCarry[3] = ShA [2] & ShB[2] & TempCarry[2];
ShCarryOut = TempCarry[3];
```

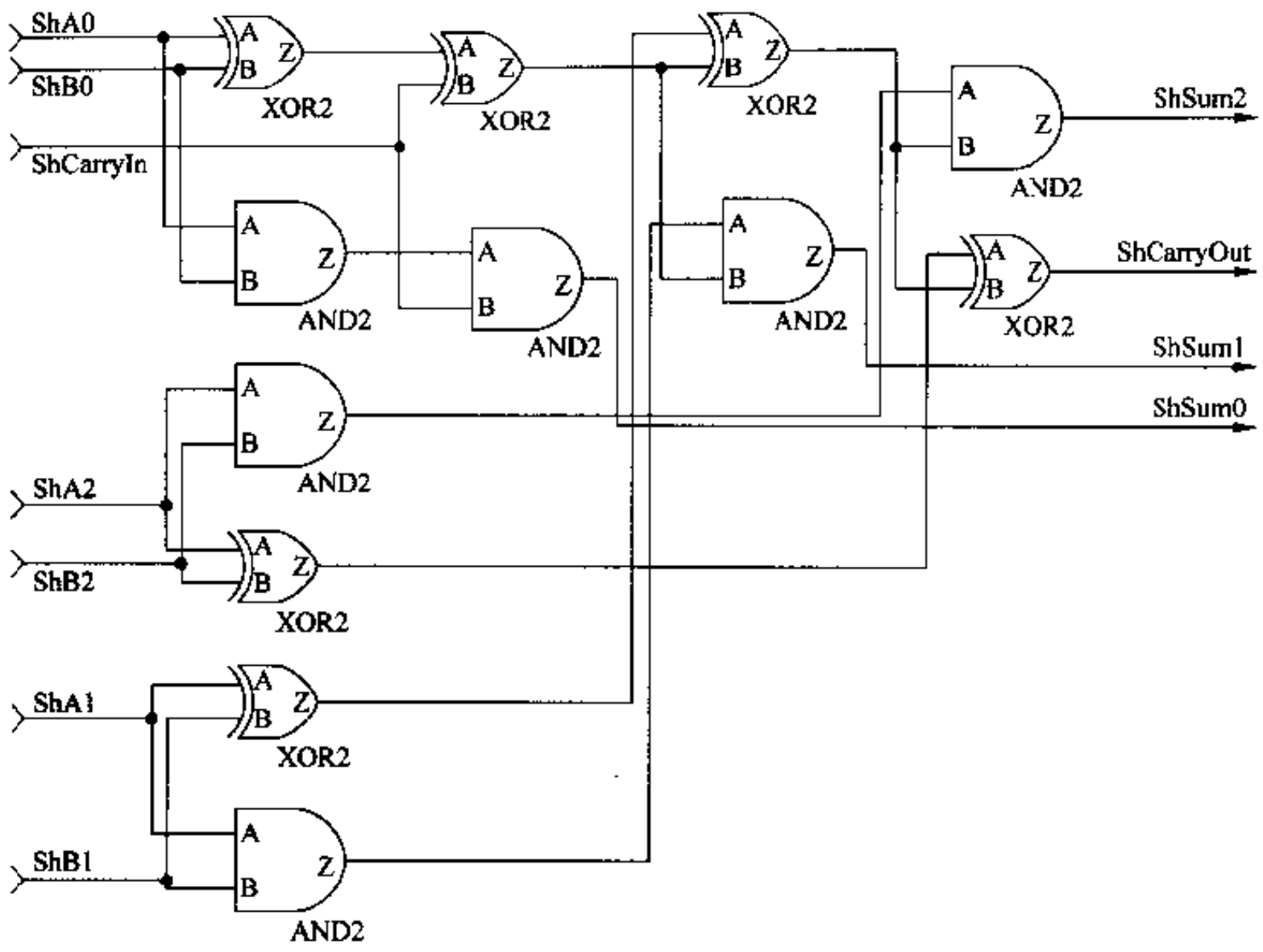

图 2-60　任务调用示例：组合逻辑

接下来请看在时钟沿控制下的任务调用示例：

```
module SynTask (ByteIn, ClockFa, ByteOut);
  input [3:0] ByteIn;
  input ClockFa;
  output [3:0] ByteOut;

  task ReverseByte;
  input [3:0] A;
  output [3:0] Z;
  integer J;
  begin
    for (J = 3; J >= 0; J = J - 1)
      Z[J] = A[3-J];
    end
  endtask
  always @ (negedge ClockFa)
```

```
    ReverseByte (ByteIn, ByteOut);
endmodule
// 综合出的网表如图 2-61 所示
```

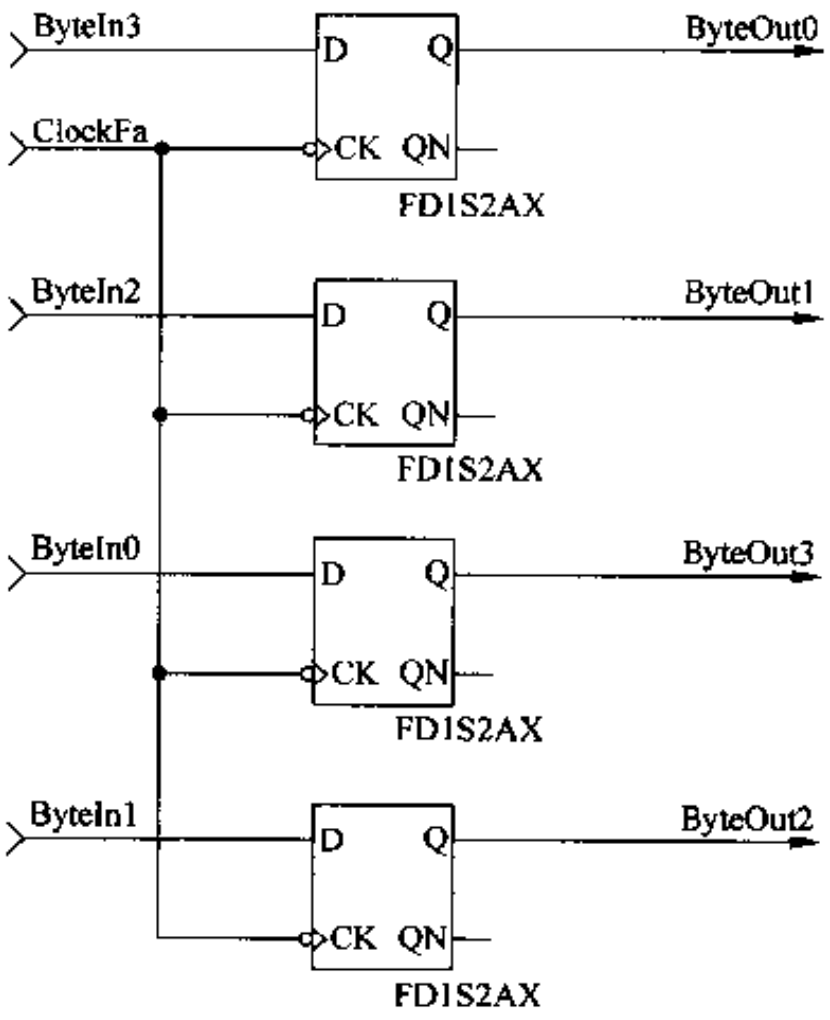

图 2-61　任务调用示例：同步逻辑

此例中，变量 *ByteOut* 在时钟 *ClockFa* 的控制下被赋值，于是它被综合成触发器。将此任务调用直接插入并展开后的代码如下：

```
ByteOut[3] = ByteIn[0];
ByteOut[2] = ByteIn[1];
ByteOut[1] = ByteIn[2];
ByteOut[0] = ByteIn[3];
```

2.21　使用 x 值和 z 值

Verilog HDL 中有两种非逻辑值：x(不定值)和 z(高阻抗)。本节将指明这两种值在综合时的应用范围。因为它们很可能会引起设计模型与综合出的网表在功能上的不一致，因此使用这些值必须非常小心。

2.21.1　x 值

可以在赋值语句中将 x 值赋给任何变量。此时，综合 x 值就会被当成无关值，综合系统会智能地选用逻辑 0 或者逻辑 1 以实现逻辑优化。

```
Reset = 'bx;      // 将无关值赋给 Reset
  // 综合系统会自动选用逻辑 0 或者逻辑 1
```

如果 case 语句(不包括 casex 和 casez 语句)的某些分支项中使用了 x 值，则综合时就不会执行那些 case 项所对应的分支，见下：

```
case (In)
  2'bx : Out = In;      // 用于综合时不会执行此分支
  default : Out = ~In;
endcase
```

这样就会出现功能不一致，此时综合工具会发出警告信息。因此需要避免在 case 语句(不包括 casex 和 casez 语句)的分支项中使用 x 值。

2.21.2　z 值

z 值用于生成三态门。可以在赋值语句中向变量赋 z 值，但是对于综合这样的赋值必须处于某种条件的控制之下，要么出现在 if 语句中，要么出现在 case 语句中。请看下例：

```
module ThreeState (Ready, DataInA, DataInB, Select1);
  input Ready, DataInA, DataInB;
  output Select1;
  reg Select1;

  always @ (Ready or DataInA or DataInB)
    if (Ready)
      Select1 = 1 'bz;
    else
      Select1 = DataInA & DataInB;
endmodule
// 综合出的网表如图 2-62 所示
```

在条件表达式中赋 z 值也可以得到三态门，请看下例：

```
module CondExprThreeState (Dnt, GateCtrl, Vcs);
  input Dnt, GateCtrl;
```

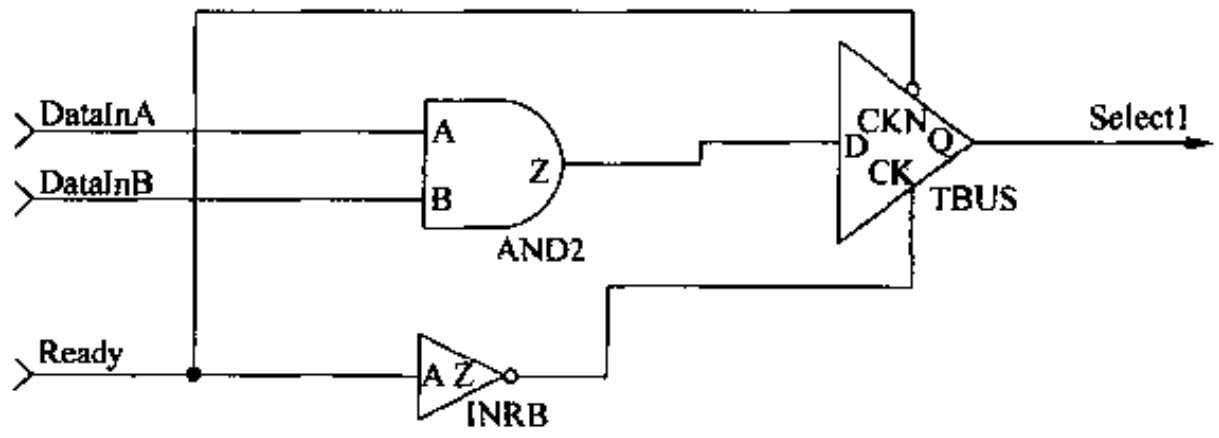

图 2-62　条件语句中赋 z 值会产生三态门

```
  output Vcs;

  assign Vcs = GateCtrl ? Dnt : 1'bz;
endmodule
// 综合出的网表如图 2-63 所示
```

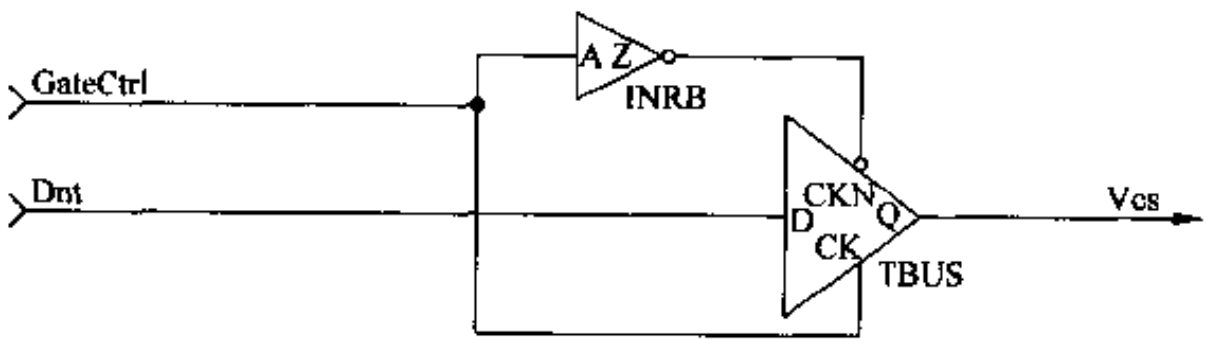

图 2-63　使用条件表达式的三态门

此外，若 case 语句(不包括 casex 和 casez 语句)的某些分支项中使用了 z 值，则综合时就不会执行那些 case 项所对应的分支，见下：

```
case (Select)
  2'b1z : DBus = | AFlow;      // 综合时不会执行此分支
  2'b11 : DBus = ^ AFlow;
  default : DBus = & AFlow;
endcase
```

此时就会出现功能不一致，好的综合工具会对此发出警告信息。要遵循该原则，只需要在 case 语句(不包括 casex 和 casez 语句)的分支项中避免使用 z 值。

如果在 always 语句中对变量赋 z 值，同时该变量被推导成触发器，则需要在触发器中保存三态门的使能逻辑。下例和上例基本相同，只是其 always 语句处于时钟事件的控制之下。

```
module ThreeStateExtraFF (Clock, Ready, DataInA, DataInB, Select1);
  input Clock, Ready, DataInA, DataInB;
```

```
    output Select1;
    reg Select1

    always @ (posedge Clock)
      if (Ready)
        Select1 <= 'bz;
      else
        Select1 <= DataInA & DataInB;
endmodule
// 综合出的网表如图 2-64 所示
```

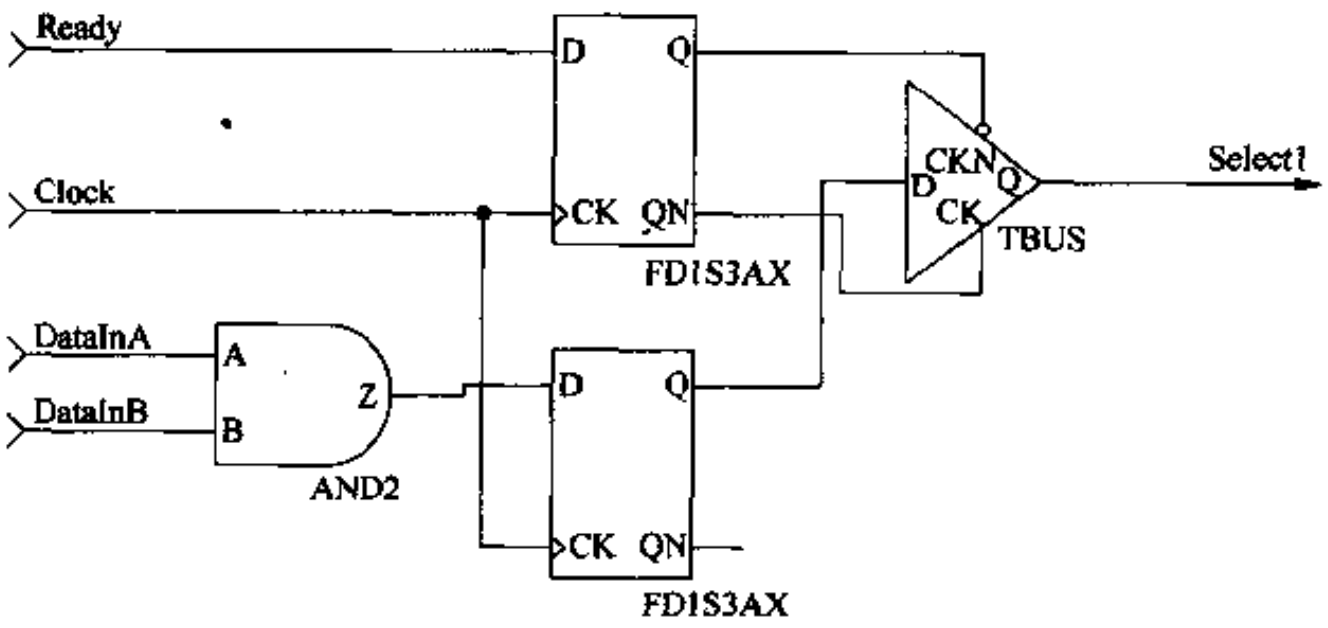

图 2-64　使用额外的触发器保持三态使能信号

请注意这里综合出两个触发器，一个是由 *Select1* 综合出来的，另一个则是由条件 *Ready* 综合出来的。如果不打算为 *Ready* 产生额外的触发器，则需要改写模型，把三态逻辑和触发器推导逻辑拆分成两个独立的 always 语句，如下一个模块 *ThreeStateNoExtraFF* 所示。模块 *ThreeStateExtraFF* 的行为不同于模块 *ThreeStateNoExtraFF* 的行为，前者的输出直接取决于 *Clock*，后者的输出不是直接取决于 *Clock*，而是直接取决于 *Ready*。

```
module ThreeStateNoExtraFF (Clock, Ready, DataInA, DataInB, Select1);
  input Clock, Ready, DataInA, DataInB;
  output Select1;
  reg Select1, TempSelect1;

  // Sequential logic;
  always @ (posedge Clock)
    TempSelect1 = DataInA & DataInB;

  // Combinational logic;
```

```
    always @ (TempSelect1 or Ready)
      if (Ready)
        Select1 = 'bz;
      else
        Select1 = TempSelect1;
endmodule
// 综合出的网表如图 2-65 所示
```

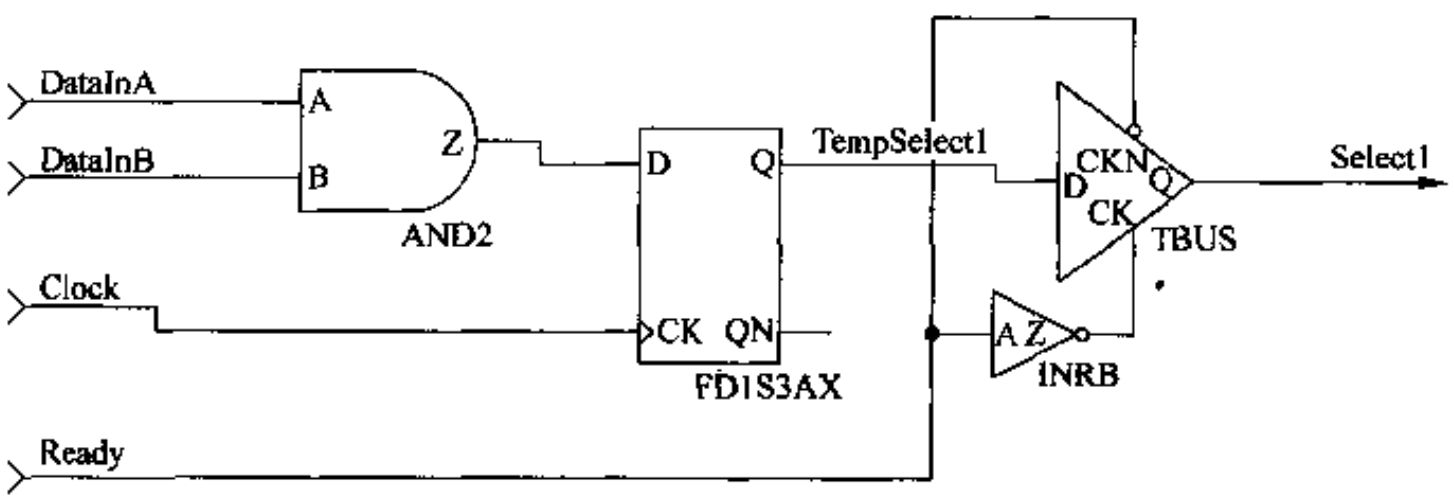

图 2-65 无额外的触发器

注意，此时引入临时变量 *TempSelect1* 用于两条 always 语句间的通信，第一条 always 语句对应的是时序逻辑部分，第二条 always 语句对应的是组合逻辑部分。临时变量 *TempSelect1* 只综合出一个触发器。

2.22 门级建模

模型中可以使用逻辑门实例化语句得到逻辑门基本元件的实例。能够综合的逻辑门基本元件有：and、nand、or、nor、not、xor、xnor、buf、bufif0、bufif1、notif0 和 notif1。门级基本元件的综合应依据其行为来生成相应的逻辑，最终把它映射到目标工艺上。对于后 4 种基本元件(三态门基本元件)，会综合出适当目标工艺中的三态门和额外的组合逻辑以实现其三态门行为。下例中，若控制信号为 1，则用两个输入量相“与”的结果来驱动总线；否则用它们相“或”的结果来驱动总线。

```
module GateLevel (A, B, Ctrl, Zbus);
  input A, B, Ctrl;
  output Zbus;
  // 没有必要对网线 AndOut 和 OrOut 作类型声明
  // 实例名 A1、O1 等也是可选的，不过建议使用这些名字以便于仿真调试
```

```
  and A1 (AndOut, A, B);     // 第一个端口是输出端,其余两个是输入端
  or O1 (OrOut, A, B);
  bufif0 B1 (Zbus, AndOut, Ctrl);     // 第一个端口是输出端
                                      // 第二个端口是输入端
                                      // 第三个端口是控制端
  bufif0 B2 (Zbus, OrOut, ! Ctrl);
endmodule
// 综合出的网表如图 2-66 所示
```

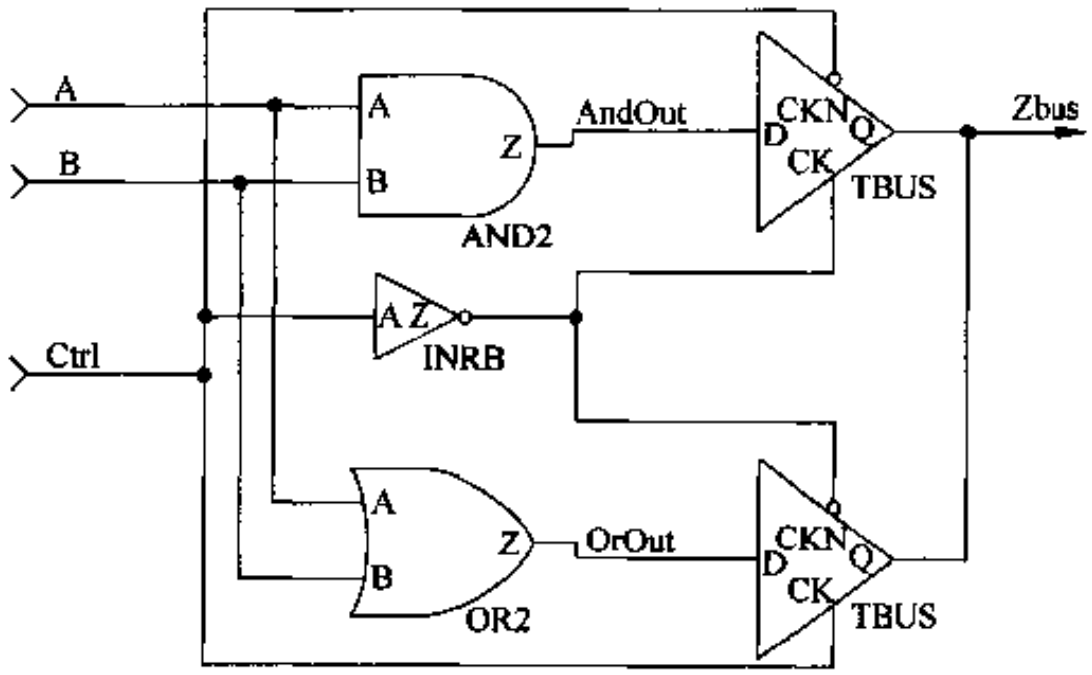

图 2-66　逻辑门的实例化

逻辑门实例化时出现的任何延迟都会被综合系统所忽略,这可能会导致 Verilog HDL 模型与综合出的网表功能不一致。

2.23　模块实例化语句

模块实例化语句可以写在模块声明中。综合系统把模块实例视作黑盒,不作进一步的处理,也就是说综合出的网表中的模块实例看起来就像是基本元件。以下全加器模块示例中含有一条模块实例化语句。注意,在图 2-67 所示的综合出的网表中,模块 *MyXor* 看起来像是在顶层模块 *FullAdderMix* 中得以描述的。

```
module FullAdderMix (A, B, CarryIn ,Sum, CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;
  wire Sft;          // 无须声明 Sft

  MyXor X1 (.In0(A), .In1(B), .Out(Sft));
```

```
  assign CarryOut = A & B & CarryIn;
  assign Sum = Sft ^ CarryIn;
endmodule
// 综合出的网表如图 2-67 所示
```

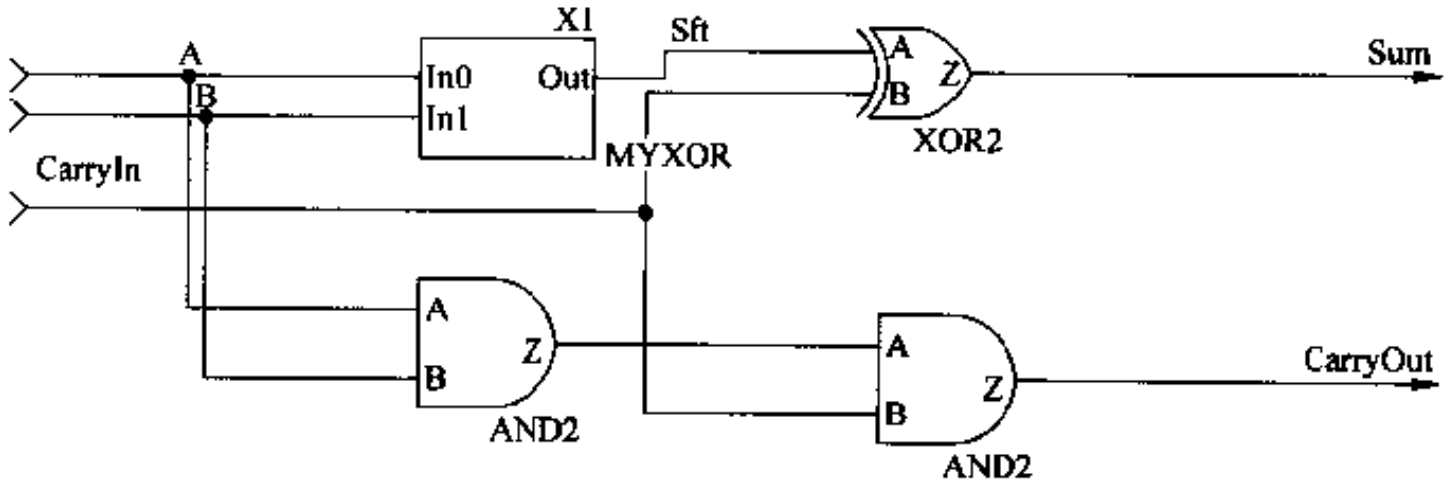

图 2-67 混合使用模块实例和行为表示

2.23.1 使用预定义功能块

在设计者不满意由综合工具所产生的电路的质量时，经常使用模块实例化语句来实例化预定义功能块。设计者也可能有一个预定义功能块库，其中有诸如存储器之类的预定义功能块。此时，设计者会更愿意用模块实例化语句来实例化预定义功能块，而不是去编写那个功能块的行为描述。这样，模块实例化语句在控制综合出的逻辑方面提供了某种灵活性，允许把一个或多个预定义功能块搭配起来使用。

实例化用户设计的乘法器

作为第一个示例，假定设计者不满意综合工具所生成的乘法逻辑，该逻辑可能是由以下代码生成的：

```
module MultiplyAndReduce (OpdA, OpdB, ReducedResult);
  input [1:0] OpdA, OpdB;
  output ReducedResult;
  wire [3:0] Test;

  assign Test = OpdA * OpdB;        // 乘法算符
  assign ReducedResult = & Test;
endmodule
```

此例中，设计者可按以下代码来实例化预定义乘法器。

```
module PreDefMultiplyAndReduce (OpdA, OpdB, ReducedResult);
  input [1:0] OpdA, OpdB;
  output ReducedResult;

  wire [3:0] Test;

  MyMult M1 (.Input1 (OpdA), .Input2 (OpdB), .Result (Test));

  assign ReducedResult = & Test;
endmodule
// 综合出的网表如图 2-68 所示
```

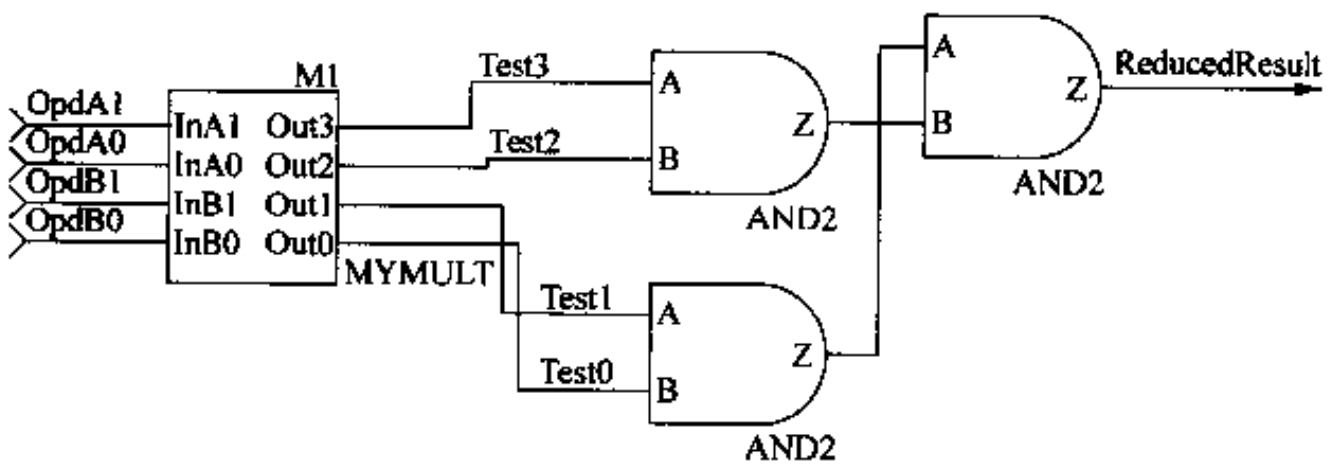

图 2-68　实例化预定义的乘法器

实例化用户专用触发器

请再看一个触发器的例子，设计者也许打算控制所生成的触发器的类型。通常，在时钟控制的 always 语句中被赋值的变量会被推导成触发器。但是，这样综合出的触发器对设计而言可能并非最理想的。设计者可能打算采用定制(custom-made)的触发器，而不是采用综合工具所生成的触发器。这样就需要重新建模，将预定义触发器实例化成模块实例。请看下例：

```
module PreDefFlipFlop (Dclock, Request, Dayp, DelS, Fop);
  input Dclock, Request, DayP, DelS;
  output Fop;
  reg Fop;
  wire NewRequest;                    // 可选的

  MyFlipFlop LabelF1 (.Data (Request), .Clock (Dclock), .Q (NewRequest));
// 上面这条模块实例化语句取代了下面这条 always 语句
```

```
    // always @ (posedge Dclock)
    //   NewRequest = Request;

    always @ (NewRequest or Dayp or DelS)
      if (NewRequest)
        Fop = DayP;
      else
        Fop = DelS;
endmodule
// 综合出的网表如图 2-69 所示
```

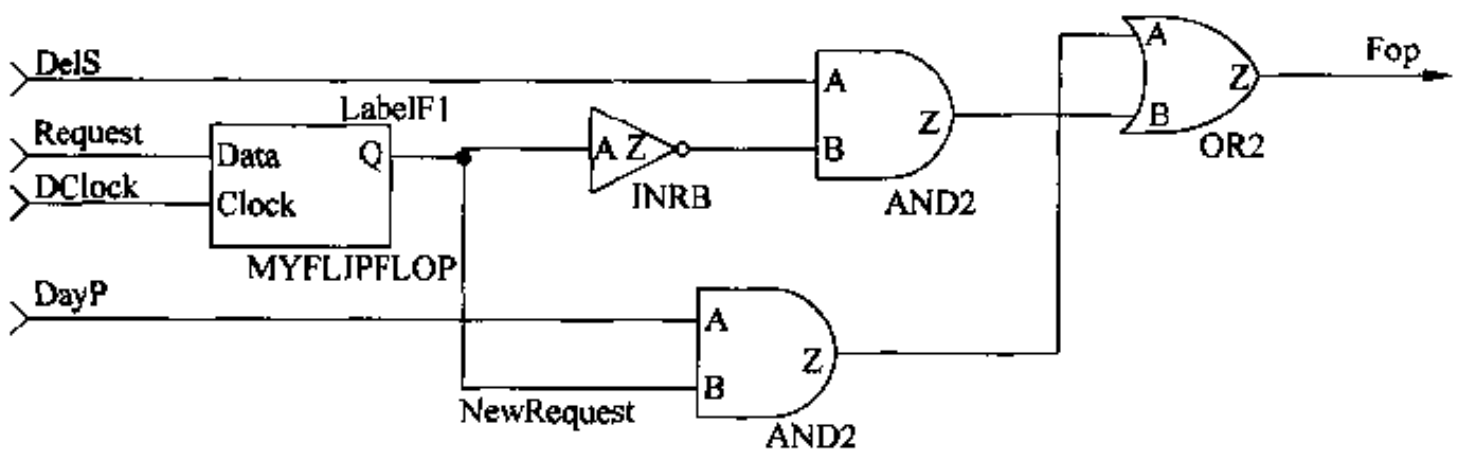

图 2-69　实例化预定义的触发器

请再看一个示例，这个 3 位的可逆计数器说明了如何使用预先设计的 D 型触发器并保持其计数器行为不变。需要添加的关键语句就是模块实例化语句。对于这样的模型，综合系统在综合出来的设计中保留预先建立的电路元件以实现预期的结果；综合出的网表正是如此。

```
module UpDownCntr (ClkA, UpDown, PresetClear, Q0, Q1,Q2);
  input ClkA, UpDown, PresetClear;
  output Q0, Q1, Q2;
  wire Bit01, Bit11, Bit12, Bit13, Qn0, Qn1, Qn2;

  assign Bit01 = UpDown ^ Q0;
  assign Bit11 = Bit01 ^ Qn1;
  assign Bit12 = UpDown ^ Q1;
  assign Bit13 = Bit01 | Bit12;
  assign Bit21 = Bit13 ^ Qn2;
```

```
    SpecialFF
      Lq0 (.D(Qn0), .Clk(ClkA), .PreClr(PresetClear), .Q(Q0), .Qbar(Qn0)),
      Lq1 (.D(Bit11), .Clk(ClkA), .PreClr(PresetClear), .Q(Q1), .Qbar(Qn1)),
      Lq2 (.D(Bit21), .Clk(ClkA), .PreClr(PresetClear), .Q(Q2), .Qbar(Qn2));
endmodule
// 综合出的网表如图 2-70 所示
```

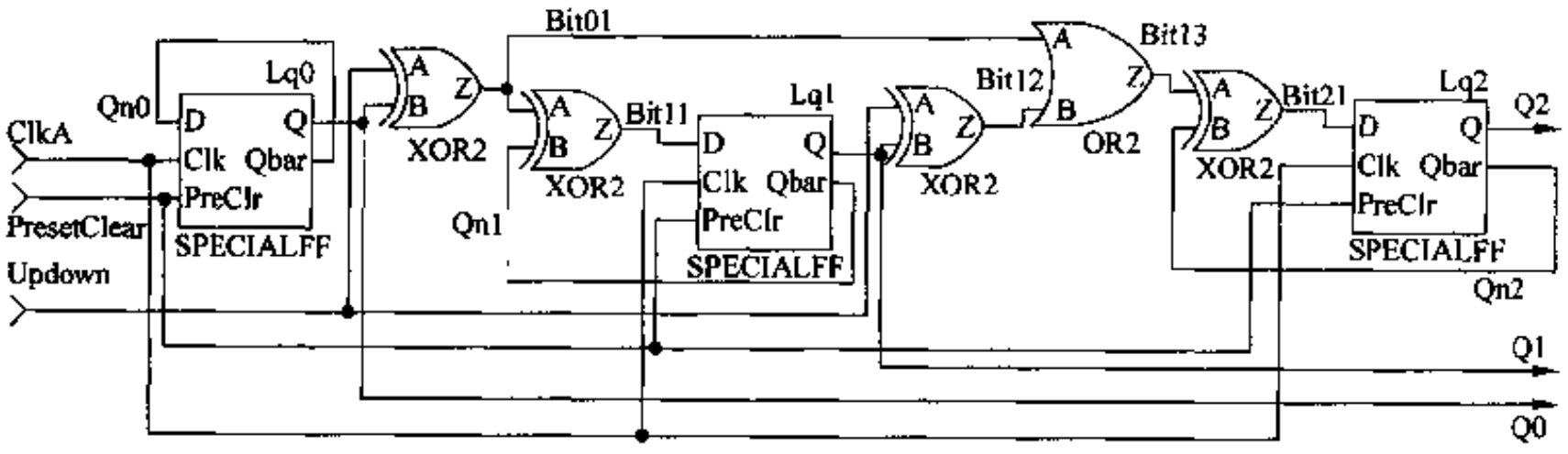

图 2-70　使用特殊的触发器

2.24　参数化的设计

Verilog HDL 中的参数为建立参数化设计模型提供了强有力的机制。请看以下这个 *N* 位寄存器的简单示例：

```
module NbitRegister (Data, Clock, Q);
  parameter N = 3;
  input [N-1:0] Data;
  input Clock;
  output [N-1:0] Q;
  reg [N-1:0] Q;

  always @ (negedge Clock)
    Q <= Data;
endmodule
// 综合出的网表如图 2-71 所示
```

模块 *NbitRegister* 综合后得到一个 3 位的寄存器。此模块是参数化的模块，这是因为通过参数指定了寄存器的位宽，这个参数很容易修改，它也可以通过在另一个模块中实例化

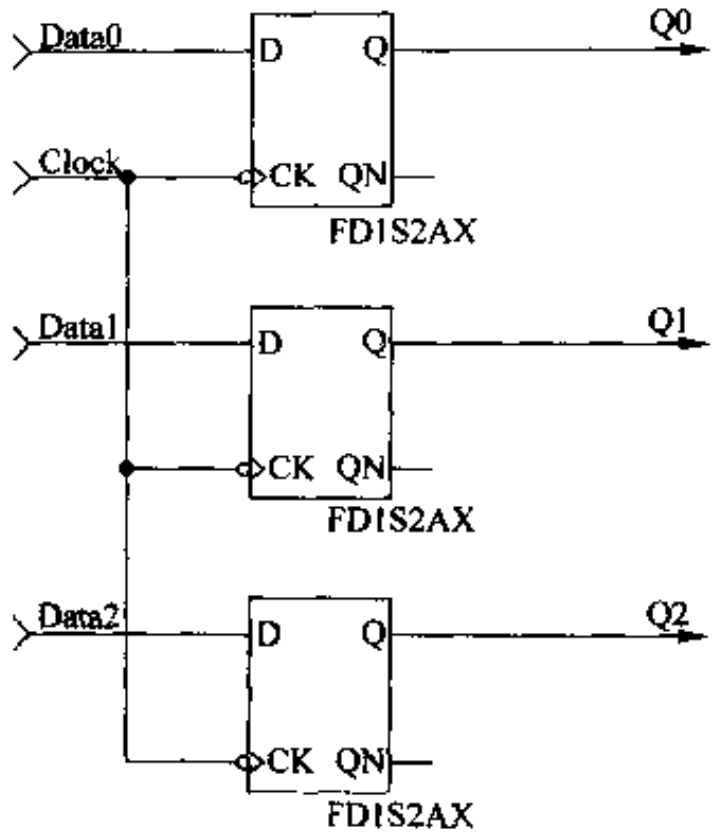

图 2-71 参数化的寄存器

该模块来改写。例如，如果打算得到 4 位的寄存器，一种方法是把模块 *NbitRegister* 中的 N 改成 4 并重新综合；另一种可行的方法是在另一个模块中实例化该模块，并为 N 指定新值。第二种方法较好，因为这样做不需要修改参数化的模块 *NbitRegister*。以下模块实例化了两个 2 位的寄存器，使用符号 # 来指定参数 N 的新值。

```
module ResolveBuses (BusA, BusB, BusControl, Clock, FinalBus);
  parameter NBITS = 2;
  input [NBITS:1] BusA, BusB;
  input BusControl, Clock;
  output [NBITS:1] FinalBus;

  wire [NBITS:1] SavedA, SavedB;

  RegisterFile # (NBITS) RfOne (.Data (BusA),.Clock (Clock), .Q(SavedA));
  RegisterFile # (NBITS) RfTwo (.Data (BusB),.Clock (Clock), .Q(SavedB));

  assign FinalBus = BusControl ? SavedA : SavedB;
endmodule
// 综合出的网表如图 2-72 所示
```

第 3 章将给出更多建模示例，以说明如何把各种 Verilog HDL 结构搭配起来使用。

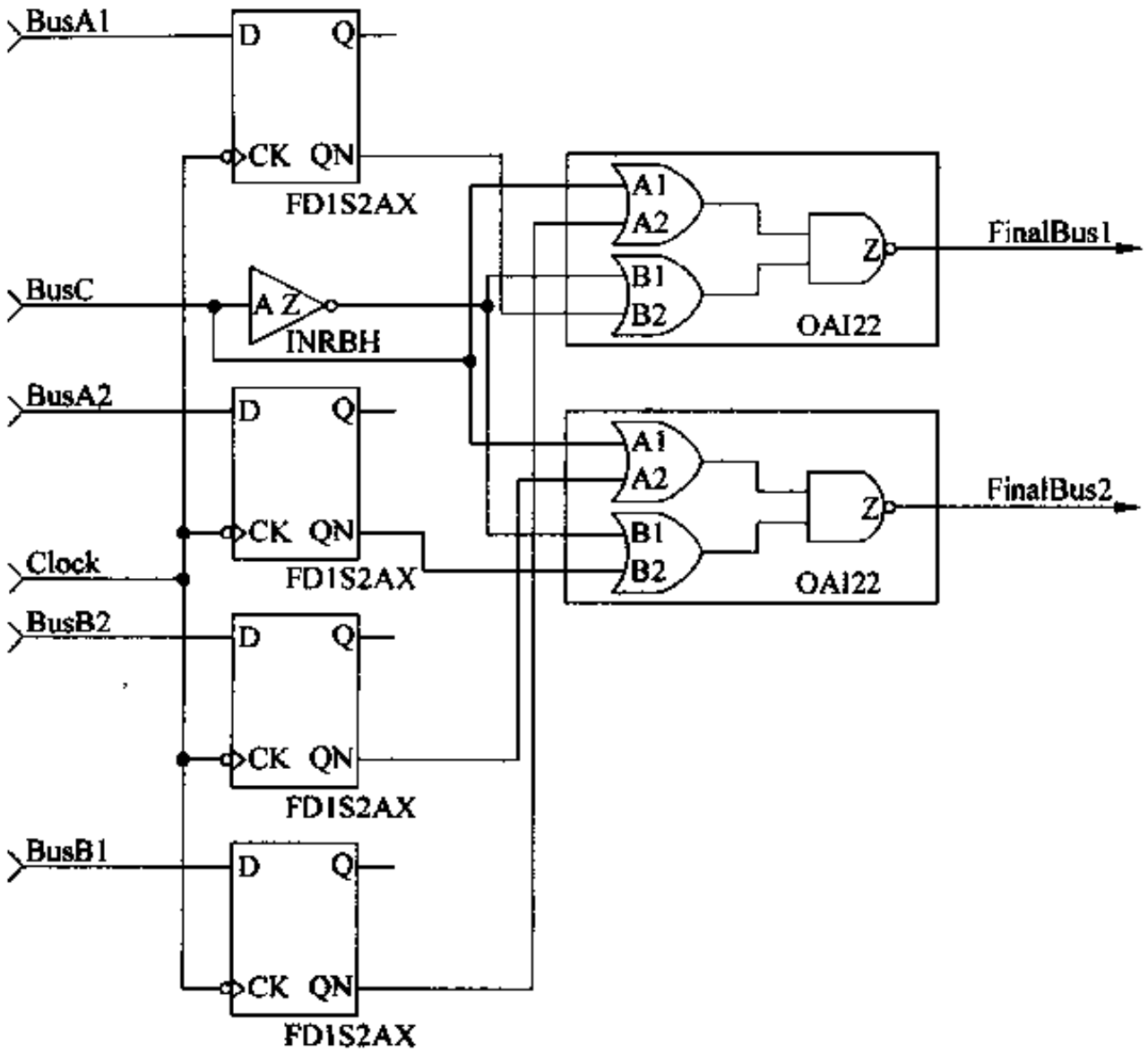

图 2-72　实例化一个参数化的寄存器

第3章　建模示例

第2章着眼于将 Verilog HDL 语句综合成门电路。本章将着眼于另一个视角，即以综合为目标对硬件元件进行建模，以及如何使用 Verilog HDL 才能实现这一任务。像前两章那样，本章会同时给出 Verilog HDL 模型和所综合出的电路图。

同时本章将提供许多更复杂的 Verilog HDL 综合示例。这些模型阐明了将各种 Verilog HDL 结构搭配起来建立可综合模型的方法。

从 Verilog HDL 描述中可综合出时序逻辑和组合逻辑。描述组合逻辑的两种主要方式是：

a）使用持续赋值语句：这是最自然的方式，因为它明确地体现了硬件中的并行性，也隐含地体现了并行结构。

b）在 always 语句的顺序块中使用过程赋值语句：各语句描述了组合逻辑块内中间值的代数运算，这是因为语义规定了顺序块中的所有语句都是顺序执行的。

采用第2章介绍的方式来描述 always 语句，就可以从其内部各语句中推导出时序逻辑元件（即触发器和锁存器）。最好不要用二维触发器阵列来综合存储器，因为此方法的效率不高。建立存储器的最好方式是采用模块实例化语句实例化预定义的存储器功能块。

3.1　组合逻辑的建模

描述组合逻辑的一种好方法是使用持续赋值语句。always 语句也可以用于描述组合逻辑，但是从描述中不能直观地看出所综合出的逻辑。如果使用持续赋值语句描述组合逻辑，则综合出的逻辑和描述完全一致。请考虑以下内建的自测试单元模型：

```
module BistCell (B0,B1,D0,D1,Z);
  input B0,B1,D0,D1;
  output Z;
  wire S1,S2,S3,S4;

  assign S1 = ~ (B0 & D1);
```

```
  assign S2 = ~ (D0 & B1);
  assign S3 = ~ (S2 | S1);
  assign S4 = S2 & S1;
  assign z = ~ (S4 | S3);
endmodule
// 综合出的网表如图 3-1 所示
```

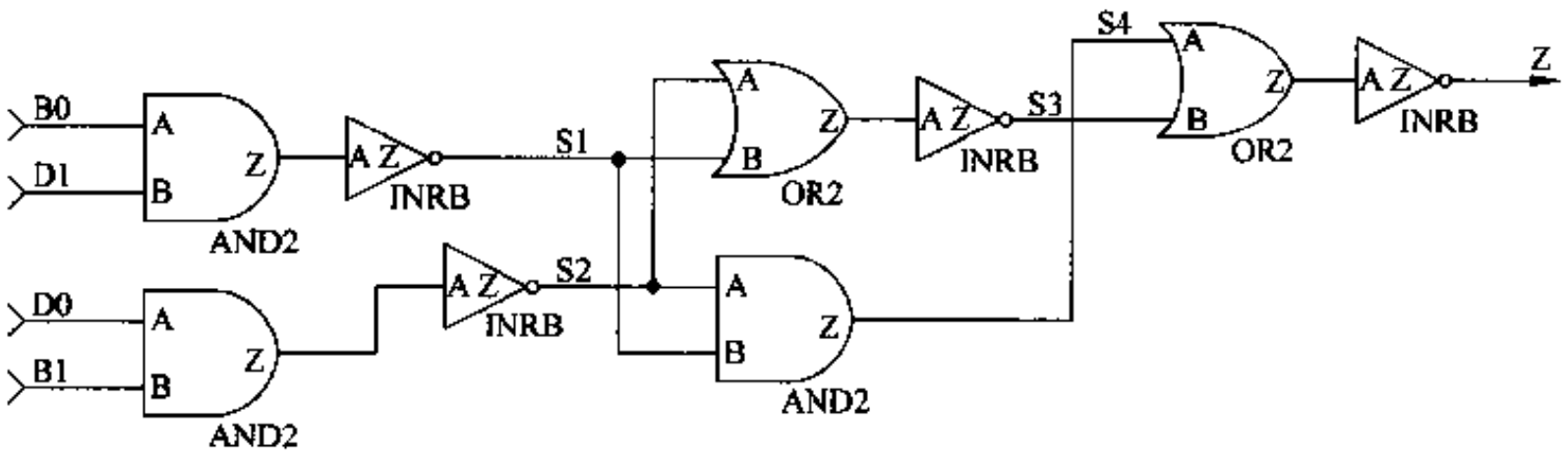

图 3-1 持续赋值产生的组合罗辑

注意：此综合出的电路结构非常类似于采用持续赋值语句描述出的电路结构。

以下仍是此单元的模型，只不过这次采用了 always 语句来描述。

```
module BistCellReg (B0,B1,D0,D1,Z);
  input B0,B1,D0,D1;
  output Z;
  reg Z;

  reg S1,S2,S3;

  always @ (B0 or D0 or B1 or D1)
  begin
    S1 = ~ (B0 & D1);
    S2 = ~ (D0 & B1);
    S3 = ~ (S2 | S1);
    S1 = S2 & S1;
     Z = ~ (S1 | S3);
  end
endmodule
```

在模块 *BistCell* 中，声明的每一个连线变量在综合出的网表中都有惟一的连线与之对应；而模块 *BitCellReg* 中的寄存器变量 S1 则非如此。注意：变量 S1 不止一处被用作临时

变量，但并不表示是同一根连线。虽然综合出的电路仍与图 3-1 所示的电路相同，但是体现不出 always 语句中的变量与综合出的网表中的连线之间一一对应的关系。

请看以下带有使能端的 2 选 1 多路选择器的组合逻辑模型示例：

```
module Mux2To1 (A, B, Select, Enable, ZeeQ);
  input [1:0] A,B;
  input select,Enable;
  output [1:0] ZeeQ;

  assign ZeeQ = (Enable) ? (Select ? A:B) : 'bz;
endmodule
// 综合出的网表如图 3-2 所示
```

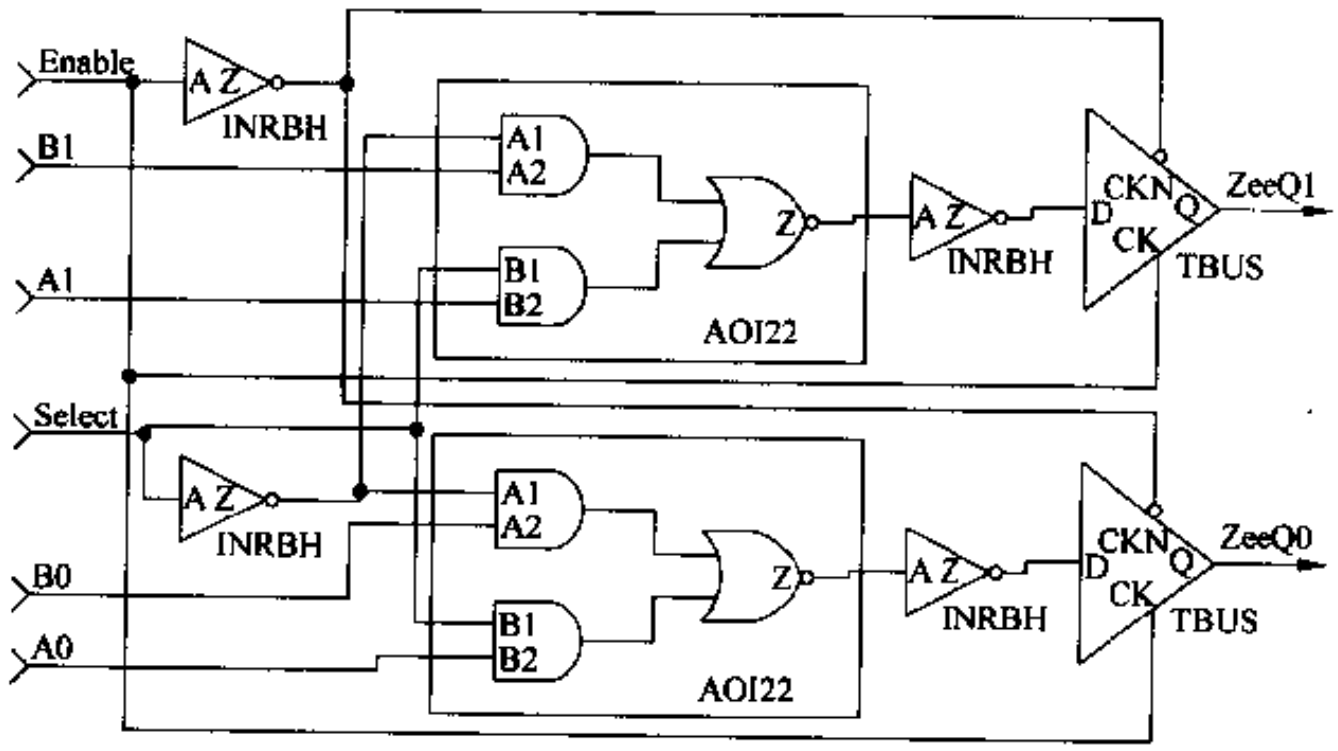

图 3-2　2 选 1 的多路选择器

3.2　时序逻辑的建模

可建立以下时序逻辑元件的模型：

a) 触发器：参见 2.17 节。

b) 带异步预置位和(或)清零的触发器：参见 2.17 节。

c) 带同步预置位和(或)清零的触发器：参见 2.17 节。

d) 锁存器：参见 2.15 节。

e) 带异步预置位和(或)清零的锁存器：参见 2.15 节。

3.3 存储器的建模

最好把存储器建模成一个元件。通常,综合工具不能有效地设计出存储器。一般采用更传统的技术来建立存储器。一旦建立了存储器模块,就可以在综合模型中使用模块实例化语句来实例化该模块。

```
module ROM (Clock,OutEnable,Address,Q,Qbar);
  input Clock,OutEnable;
  input [M-1:0] Address;
  output [N-1:0] Q,Qbar;

    // 在此描述存储器(也许是不可综合的)
endmodule

module MyModule(...);
  wire clk,Enable;
  wire [M-1:0] Abus;
  wire [N-1:0] Dbus;

  ROM R1 (.Clock(clk), .OutEnable(Enable),.Address(Abus), .Q(Dbus), .Qbar());
  ...
endmodule
```

一个寄存器堆(register file)可以建模成二维寄存器变量(Verilog HDL 中的二维寄存器变量可以被推导成存储器),从而可以被综合。请看以下寄存器堆示例:

```
module RegFileWithMemory (Clk, ReadWrite, Index, DataIn, DataOut);
  parameter N=2, M=2;
  input Clk,ReadWrite;
  input [1:N]Index;        // 范围不必与此相同
  input [0:M-1] DataIn;
  output [0:M-1] DataOut;
  reg [0:M-1] DataOut;
  reg [0:M-1] RegFile [0:N-1];

  always @ (negedge Clk)
    if (ReadWrite)
      DataOut <= RegFile[Index];
    else
      RegFile[Index] <= DataIn;
endmodule
// 2×2 的寄存器堆综合出的网表如图 3-3 所示
```

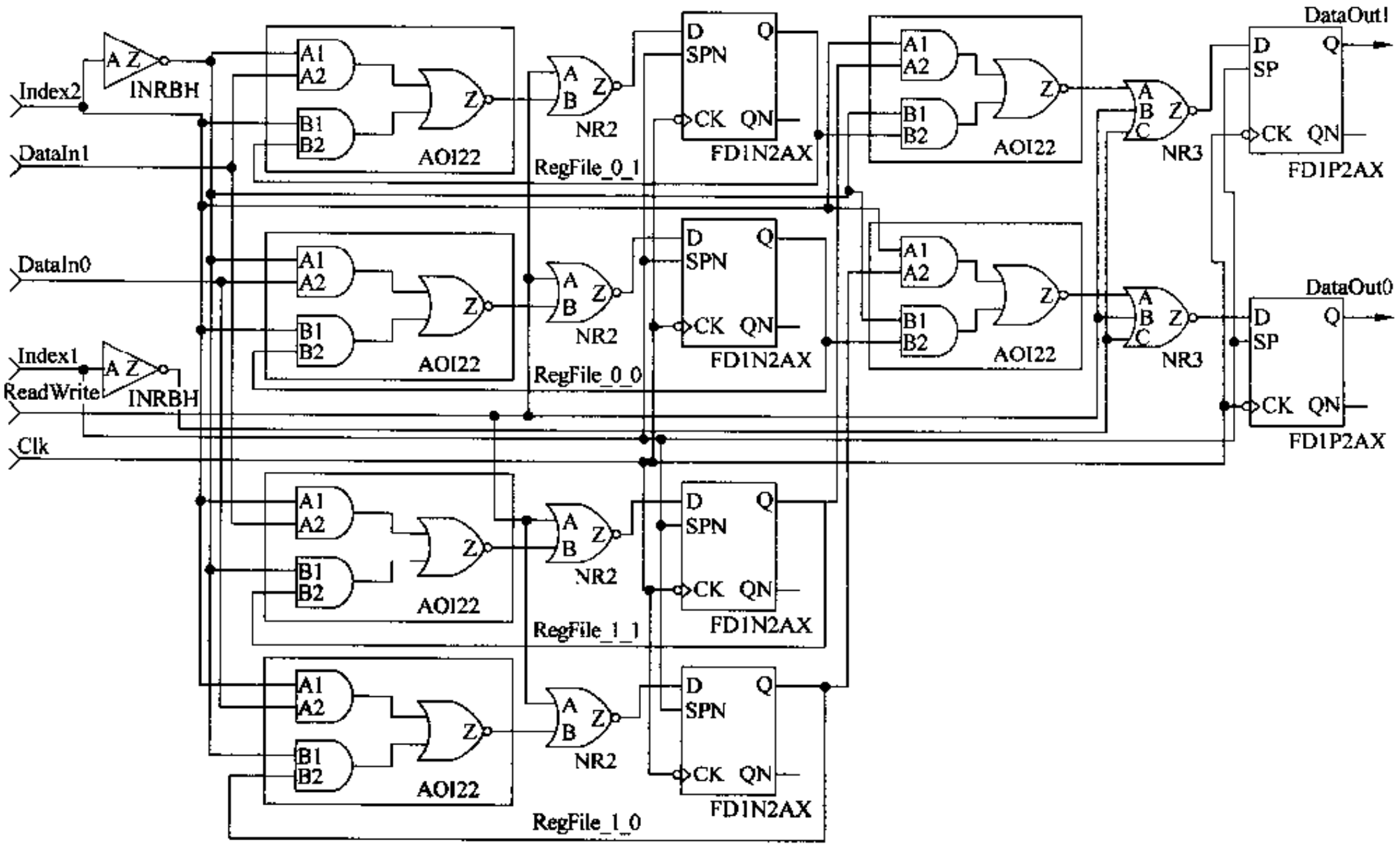
Index2
INRBH
DataIn1
DataIn0
Index1
ReadWrite
INRBH
Clk
A1
A2
B1
B2
AOI22
NR2
RegFile_0_1
RegFile_0_0
RegFile_1_1
RegFile_1_0
D
SPN
Q
CK
QN
FD1N2AX
A
B
C
NR3
DataOut1
DataOut0
SP
FD1P2AX

图3-3　2×2的寄存器样

注意：此例总共综合出 6 个触发器，其中 4 个用于寄存器堆 *RegFile*，2 个用于 *DataOut*。

3.4 编写布尔等式

布尔等式表示的是组合逻辑。最好用持续赋值语句来描述布尔等式。以下格雷码至二进制编码转换器示例中使用了布尔等式：

A B C	二进制码
0 0 0	0 0 0
0 0 1	0 0 1
0 1 1	0 1 0
0 1 0	0 1 1
1 1 0	1 0 0
1 1 1	1 0 1
1 0 1	1 1 0
1 0 0	1 1 1

```
module GrayToBinary (A, B, C, Bc0, Bc1, Bc2);
  input A, B, C;
  output Bc0, Bc1, Bc2;
  wire NotA, NotB, NotC;

  assign NotC = ~ C;
  assign NotB = ~ B;
  assign NotA = ~ A;

  assign Bc0 = (A & B & NotC) | (A & B & C) |
               (A & NotB & C) | (A & NotB & NotC);
  assign Bc1 = (NotA & B & C) | (NotA & B & NotC) |
               (A & NotB & C) | (A & NotB & NotC);
  assign Bc2 = (NotA & NotB & C) | (NotA & B & NotC) |
               (A & B & C) | (A & NotB & NotC);
endmodule
// 综合出的网表如图 3-4 所示
```

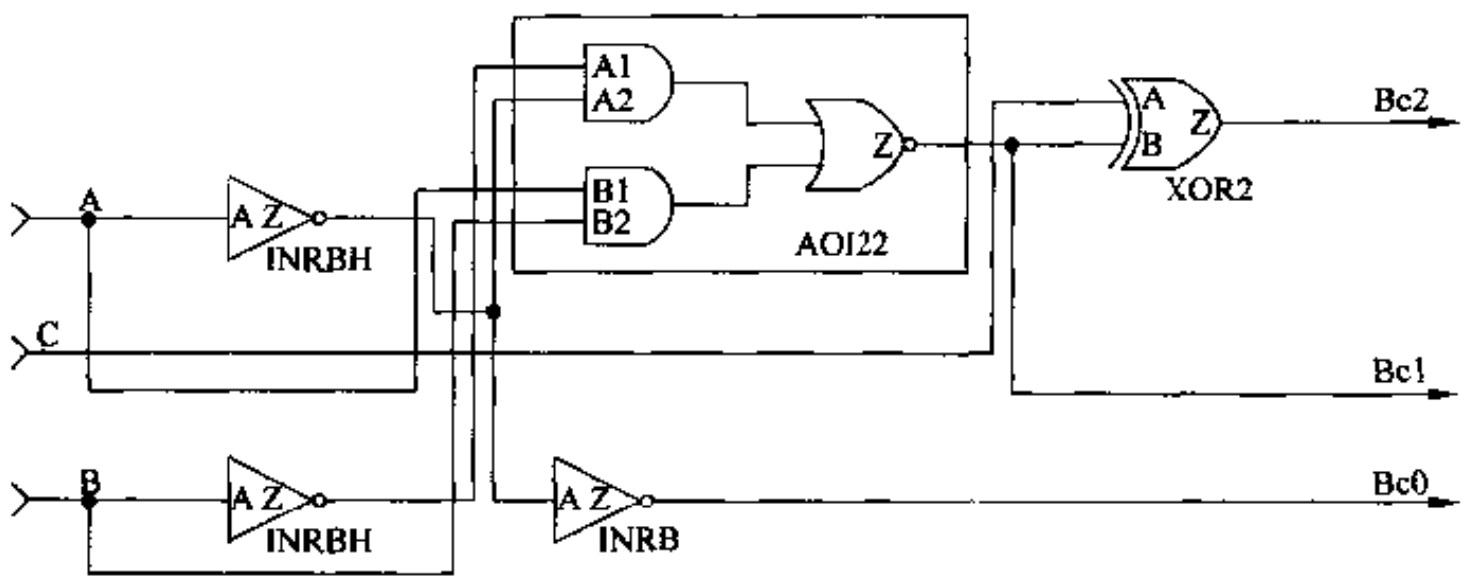

图 3-4 格雷码至二进制编码转换的逻辑电路

3.5 有限状态机的建模

3.5.1 Moore 有限状态机

在 Moore 有限状态机(finite state machine，FSM)中，电路的输出取决于机器的状态而与其输入无关。图 3-5 对此作了解释。既然输出仅取决于其状态，那么在 always 语句中使用 case 语句来描述 Moore 状态机不失为一种好办法。case 语句用于实现不同状态间的转换，并且每种状态的输出逻辑都在相应的分支中得以描述。always 语句事件表中可以有时钟事件，以表明它是受时钟控制的 always 语句，这就模拟了有限状态机在每个时钟沿上同步状态转移的条件。状态机的状态被建模成寄存器变量(reg 数据类型的变量)。

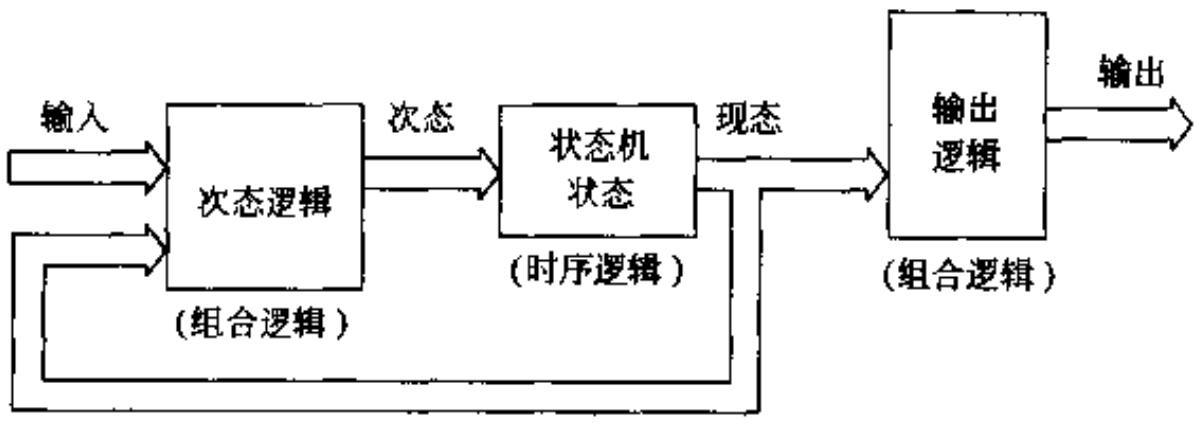

图 3-5 Moore 有限状态机

请看以下 Moore 有限状态机示例。寄存器变量 *MooreState* 用于实现状态机 4 种状态的建模。事件表指示状态转移与时钟的每个上升沿相同步。

```
module MooreFSM (A, ClkM, Z);
  input A, ClkM;
  output Z;
  reg Z;

  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
  reg [0:1] MooreState;

  always @ (posedge ClkM)
    case (MooreState)
      S0:
        begin
          Z <= 1;
          MooreState <= (! A) ? S0:S2;
        end
      S1:
        begin
          Z <= 0;
          MooreState <= (! A) ? S0:S2;
        end
      S2:
        begin
          Z <= 0;
          MooreState <= (! A) ? S2:S3;
        end
      S3:
        begin
          Z <= 1;
          MooreState <= (! A) ? S1:S3;
        end
    endcase
endmodule
// 综合出的网表如图 3-6 所示
```

综合此模型，会推导出 3 个触发器，两个用于保存状态机的状态值(*MooreState*)，另一个用于保存输出 Z 的值。对状态采用了顺序值编码。

上例中，输出也保存在触发器中。假如需要一个非锁存的输出，该如何处理？此时，可把对 Z 的赋值分离出来并放入另一条 always 语句中，如以下模型所示：

```
module MooreFSM2 (A, ClkM, Z);
```

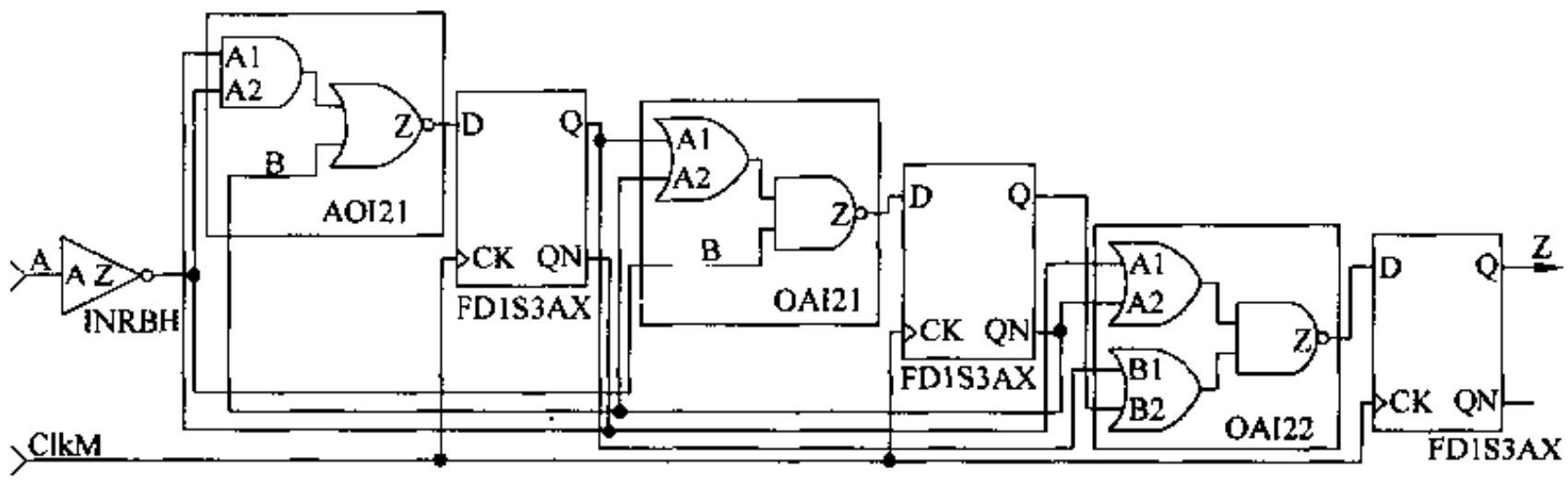

图 3-6 Moore 有限状态机模型综合出的网表

```
input A, ClkM;
output Z;
reg Z;

parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
reg [0:1] MooreState;
  always @ (posedge ClkM)
    case (MooreStae)
      S0 : MooreState <= (! A) ? S0 : S2;
      S1 : MooreState <= (! A) ? S0 : S2;
      S2 : MooreState <= (! A) ? S2 : S3;
      S3 : MooreState <= (! A) ? S1 : S3;
    endcase

  // 很明显,输出只取决于现态
  always @ (MooreState)
    case (MooreState)
      S0 : Z = 1;
      S1 : Z = 0;
      S2 : Z = 0;
      S3 : Z = 1;
    endcase
endmodule
// 综合出的网表如图 3-7 所示
```

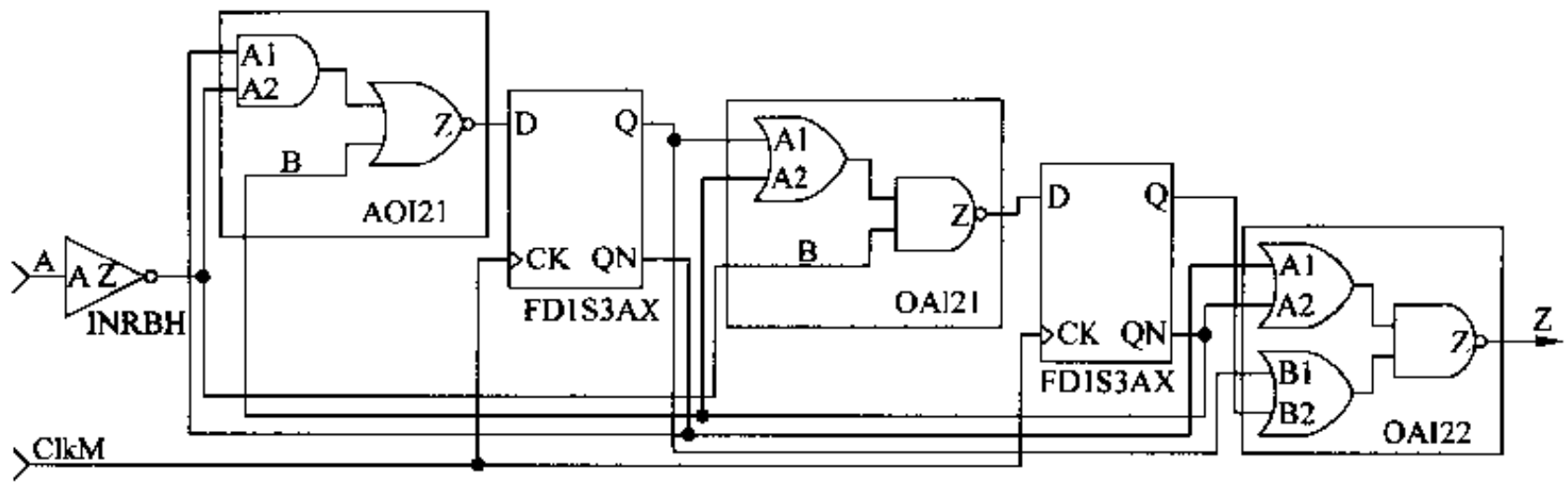

图 3-7 未锁存的输出

3.5.2 Mealy 有限状态机

Mealy 有限状态机中，输出既取决于机器状态，也取决于其输入。图 3-8 对此作了解释。注意：在此例中输出的变化可以异步于时钟。

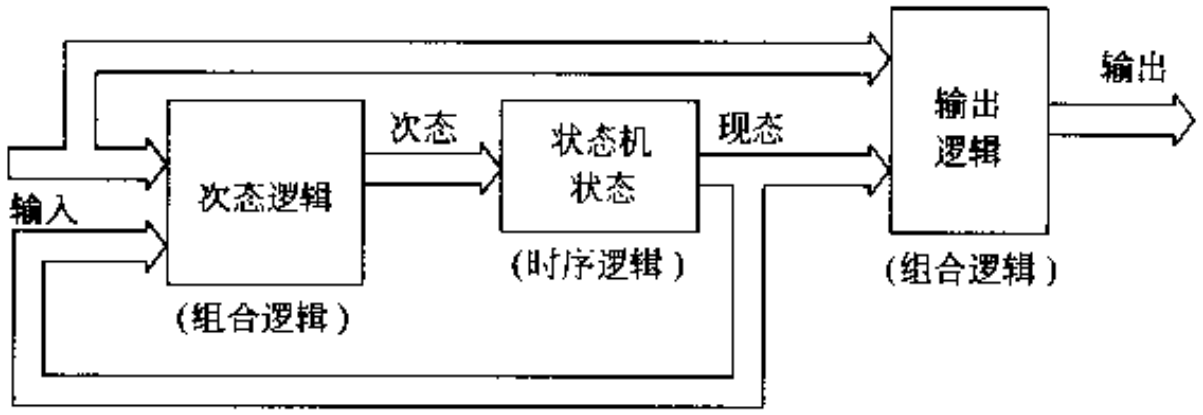

图 3-8 Mealy 有限状态机

描述 Mealy 有限状态机的最佳方式之一是使用两条 always 语句，一条用于描述时序逻辑，另一条用于描述组合逻辑(既包括次态逻辑，也包括输出逻辑)。这样做很有必要，因为输入上的任何变化都会直接影响被描述成组合逻辑的输出。状态机的状态被建模成寄存器变量。

请看以下 Mealy 有限状态机示例。变量 *MealyState* 用来保存状态机的状态，而 *NextState* 用于描述组合逻辑的 always 语句向描述时序逻辑的 always 语句传递信息。输入 *Reset* 异步地将状态复位成 *ST0*。

```
module MealyFSM (A, ClkB, Reset, Z);
  input A, ClkB,Reset;
  output Z;
  reg Z;
```

```
parameter ST0 = 4'b00, ST1 = 4'b01, ST2 = 4'b10;
reg [0:1] MealyState, NextState;

// 时序逻辑:
always @ (posedge Reset or posedge ClkB)
  if (Reset)
    MealyState <= ST0;
  else
    MealyState <= NextState;

// 组合逻辑:
always @ (MealyState or A)
  case (MealyState)
    ST0 :
      begin
        Z = (A) ? 1 : 0;
        NextState = (A) ? ST2 : ST0;
      end
    ST1 :
      begin
        Z = (A) ? 1 : 0;
        NextState = (A) ? ST0 : ST1;
      end
    ST2 :
      begin
        Z = 0;
        NextState = (A) ? ST1 : ST2;
      end
    default : // 需要定义默认行为
              // 否则会把变量 Z 和 NextState 推导成锁存器
      begin
        Z = 0; NextState = ST0;
      end
  endcase
endmodule
// 综合出的网表如图 3-9 所示
```

采用此给定的状态编码就会推导出两个触发器以保存变量 *MealyState* 的值。如下所示,在 case 语句中指定“full_case”就能避免给出默认分支。

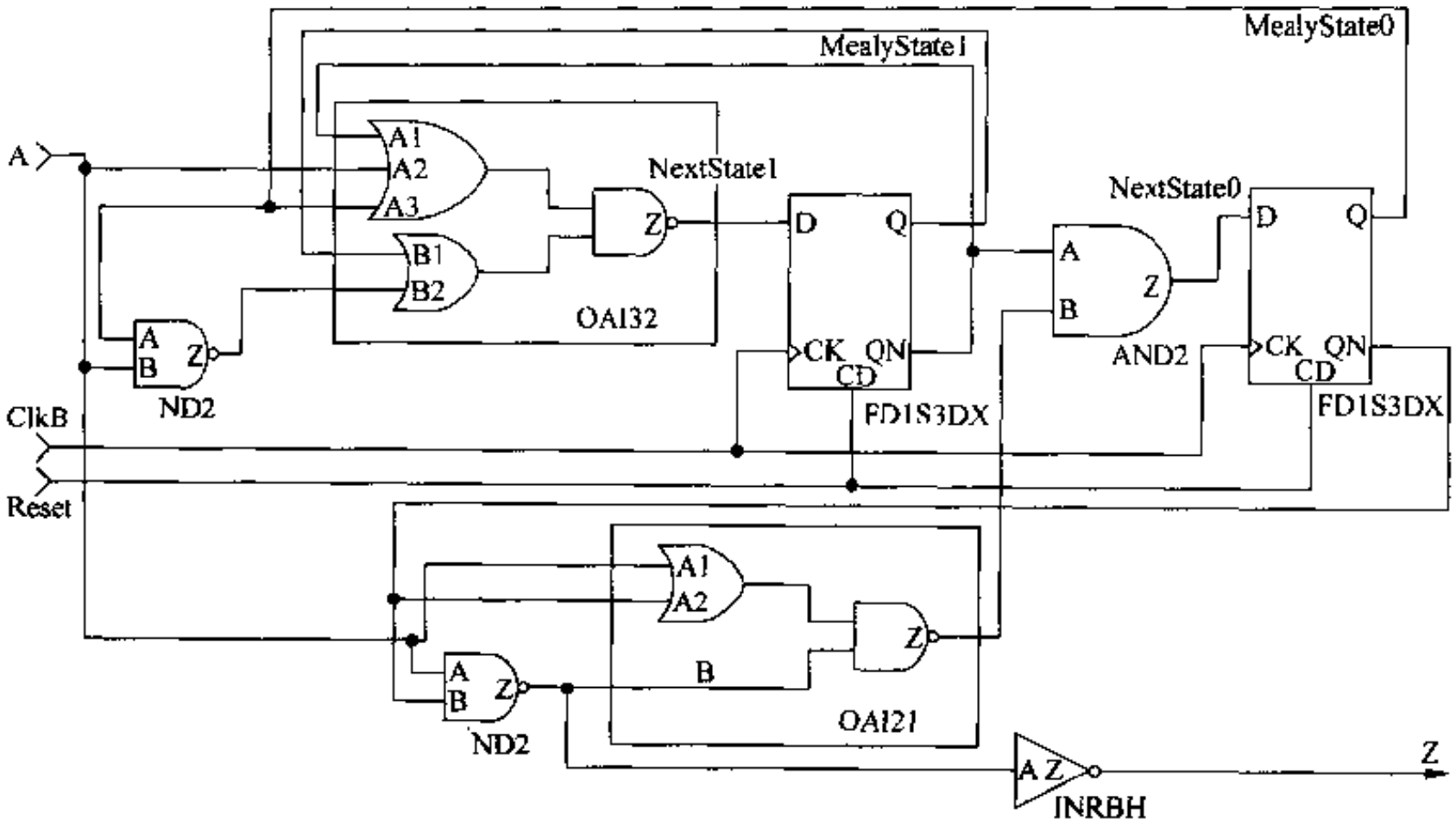

图 3-9 Mealy 有限状态机综合出的网表

```
always @ (MealyState or A)
  case (MealyState)     // synthesis full_case
    ST0 :
      begin
        Z = (A) ? 1 : 0;
        NextState = (A) ? ST2 : ST0;
      end
    ST1 :
      begin
        Z = (A) ? 1 : 0;
        NextState = (A) ? ST0 : ST1;
      end
    ST2 :
      begin
        Z = 0;
        NextState = (A) ? ST1 : ST2;
      end
endcase
```

此例中，没有把 *Z* 和 *NextState* 推导成锁存器，这是因为 full_case 综合指令表明了不可

能出现其他 case 分支项值。但是,避免推导出锁存器的首选方式是使用默认分支。

请再看一个 Mcaly 有限状态机示例,这次使用了独热码(one-hot)状态编码。

```
module MealyFSM2 (A, ClkC, Reset, Z);
  input A, ClkC, Reset;
  output Z;
  reg Z;

  parameter ST0 = 2'd0, ST1 = 2'd1, ST2 = 2'd2;
  reg [0:2] NextState, MealyState;

  // 时序逻辑:
  always @ (posedge Reset or posedge ClkC)
    if (Reset)
    begin
      MealyState <= 0;
      MealyState[ST0] <= 1'b1;
    end
    else
      MealyState <= NextState;

// 组合逻辑:
always @ (MealyState or A)
begin
  NextState = 3'b0;          // 默认赋值
  Z = 1'b0;

  case (1'b1)
    MealyState[ST0]:
      if (A)
      begin
        Z = 1'b1;
        NextState[ST2] = 1'b1;
      end
      else
        NextState[ST0] = 1'b1;
    MealyState[ST1]:
      if (A)
      begin
```

```
            Z = 1'b1;
            NextState[ST0] = 1'b1;
          end
          else
            NextState[ST1] = 1'b1;
        MealyState[ST2]:
          if (A)
            NextState[ST1] = 1'b1;
          else
            NextState[ST2] = 1'b1;
      endcase
    end
  endmodule
  // 综合出的网表如图 3-10 所示
```

3.5.3 状态编码

有很多种对有限状态机的状态进行建模的方法，在此将介绍一些最常用的方法。下面以前面的 *MooreFSM* 模块为例来介绍一些状态编码方法。

使用整数

最简单的方法就是用整数对状态赋值。

```
integer MooreState;
...
case (MooreState)
  0 : ...
    MooreState = 2;
    ...
  1 :
    ...
endcase
```

此方法的问题在于列出整数的所有可能取值是不切实际的，为了避免产生锁存器，必须给出默认的 case 分支，或者使用 full_case 综合指令。此方法的另一问题在于 HDL 代码的可读性不强。

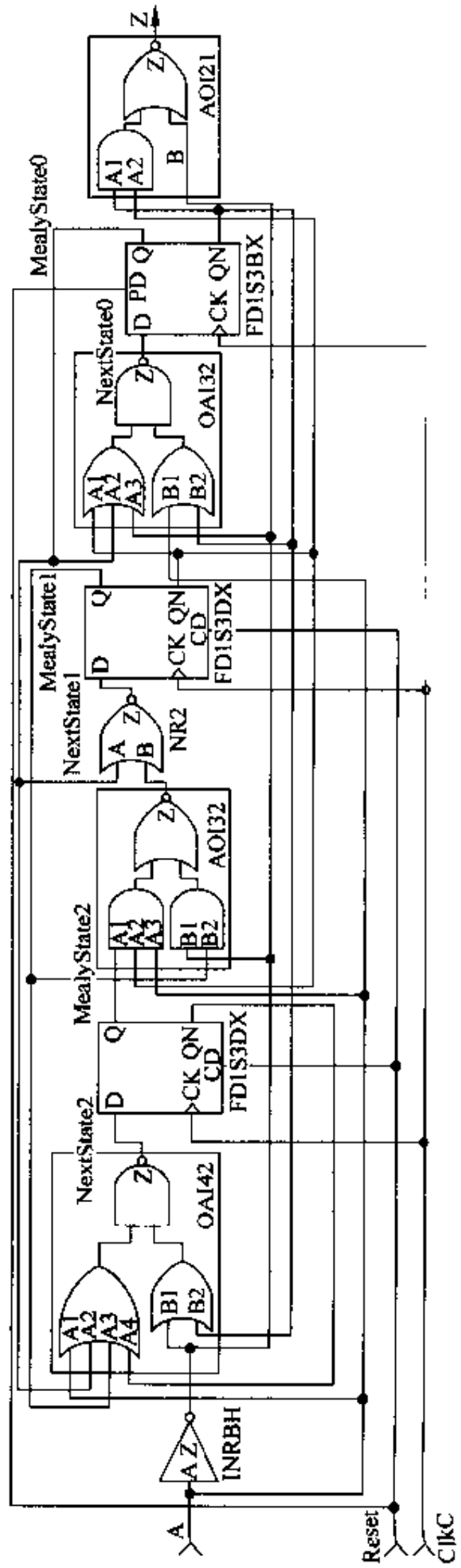

图 3-10 独热码编码的状态机示例

使用参数声明

另一种可行的方法是声明参数，并在 case 语句中使用这些参数。

```
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

reg [0:1] MooreState;
…
case (MooreState)
  S0 : …
    MooreState = S2;
    …
  S1 :
    …
endcase
```

此方法的优点是将对各状态的编码集中在一处进行明确地描述，修改起来比较容易。如果直接使用参数声明或者整型值，则综合系统使用整型值编码所需的最少位数。上例中，状态编码仅需要两位，因为最大的整型值是 3。

如果给定的是其他编码方式该怎么办？可以把各状态描述成位向量。

```
parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b100;

reg [0:2] MooreState;
…
case (MooreState)
  S0 :…
    MooreState = S2;
    …
  S1 :
    …
endcase
```

此时，状态编码需要的位数直接由参数的位数所指定，此例中是 3 位。当然，若参数的位数发生变化，则状态机的状态 *MooreState* 的位宽必须足以容纳新的参数位宽。

3.6 通用移位寄存器的建模

请看 3 位通用移位寄存器的综合模型。该通用移位寄存器具有以下功能：

a) 保存值

b) 左移

c) 右移

d) 值的载入

此通用寄存器可用作串入/串出移位寄存器、并入/串出移位寄存器、串入/并出移位寄存器以及并入/并出移位寄存器。其状态表如下：

功能	输入		次态		
	(S0	S1)	(Q[2]	Q[1]	Q[0])
保持	0	0	Q[2]	Q[1]	Q[0]
左移	0	1	Q[1]	Q[0]	RightIn
右移	1	0	LeftIn	Q[2]	Q[1]
载入	1	1	ParIn[2]	ParIn[1]	ParIn[0]

其综合模型如下：

```
module UnivShiftRegister (Clock, Clear, LeftIn,RightIn, S0, S1, ParIn, Q);
  input Clock, Clear, LeftIn, RightIn, S0, S1;
  input [2:0] ParIn;
  Output [2:0] Q;
  reg [2:0] Q;

  always @ (negedge Clear or posedge Clock)
    if (! Clear)
      Q <= 3'b000;
    else
      case ({S0, S1})
        2'b00 : ;
        2'b01 :
          Q <= {Q[1:0], RightIn};
        2'b10 :
          Q <= {LeftIn, Q[2:1]};
        2'b11 :
          Q <= ParIn;
      endcase
endmodule
// 综合出的网表如图 3-11 所示
```

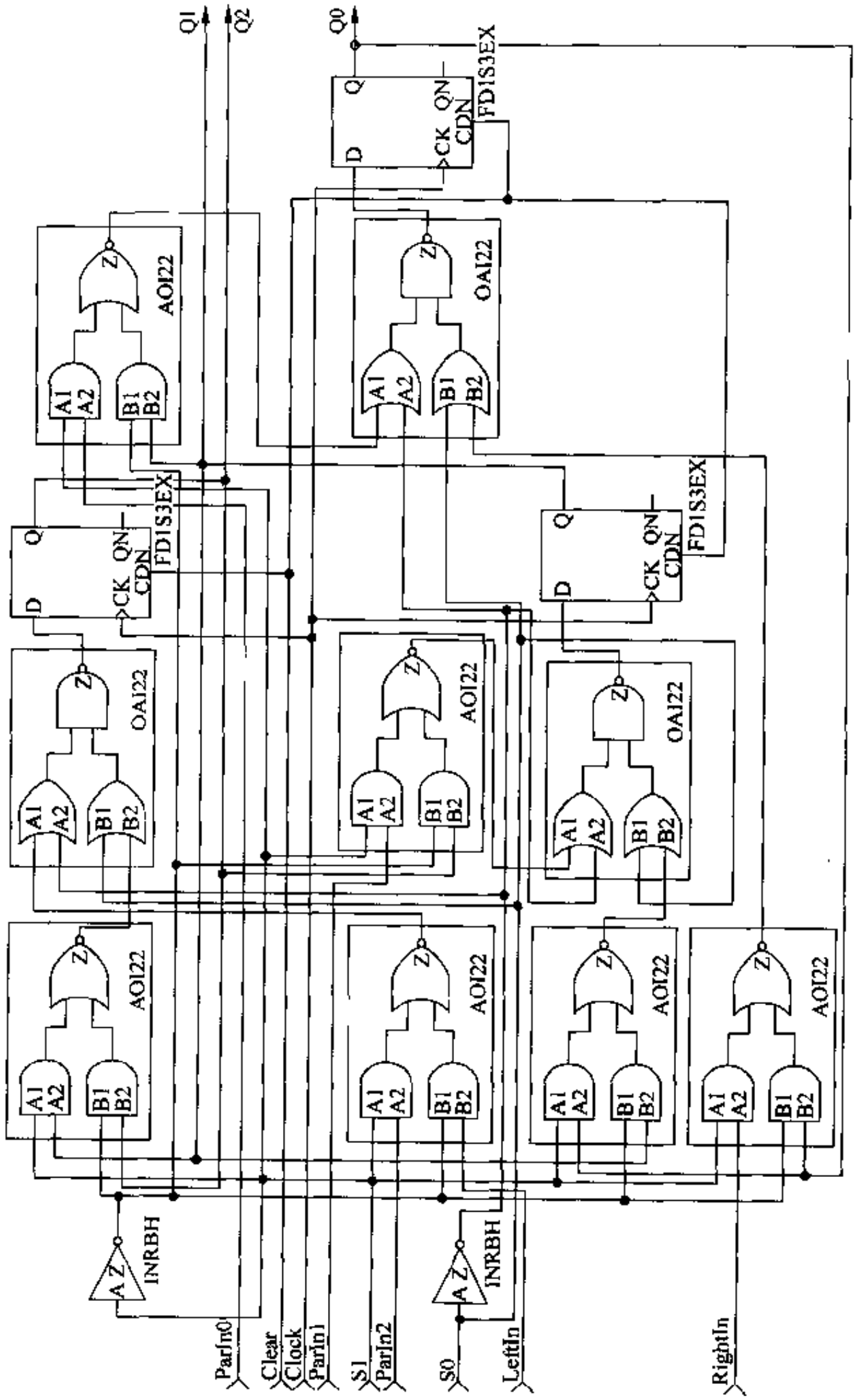

图3-11　3位通用移位寄存器

3.7 ALU的建模

3.7.1 参数化的ALU

请看以下参数化的 N 位算术逻辑单元(ALU)示例,它能实现异或、小于和递增运算。

```
module ArithLogicUnit (A, B, Select, CompareOut, DataOut);
  parameter N = 2;
  input [N-1:0] A, B;
  input [2:0] Select;
  output CompareOut;
  output [N-1:0] DataOut;
  reg CompareOut;
  reg [N-1:0] DataOut;

  parameter OP_XOR = 3'b001, OP_INCRA = 3'b010,
            OP_LT = 3'b100;

always @ (A or B or Select)
  case (Select)
    OP_INCRA :
      begin
        DataOut = A + 1;
        CompareOut = 'bx;
      end
    OP_XOR :
      begin
        DataOut = A ^ B;
        CompareOut = 'bx;
      end
    OP_LT :
      begin
        CompareOut = A < B;
        DataOut = 'bx;
      end
    default :
```

```
            begin
              CompareOut = 'bx;
              DataOut = 'bx;
            end
        endcase
    endmodule
    // 综合出的网表如图 3-12 所示
```

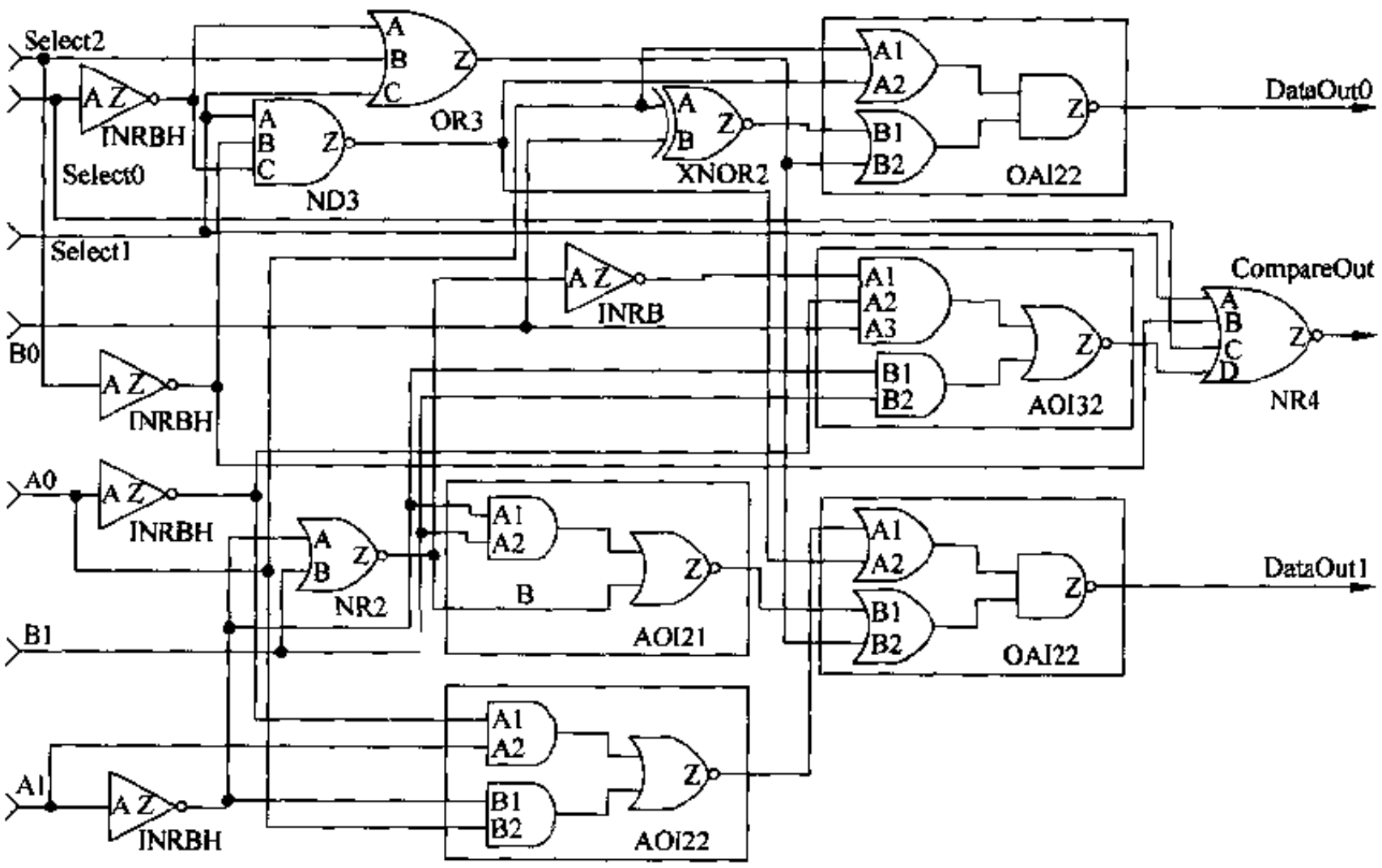

图 3-12　2 位的 ALU

在实例化此 ALU 时，指定不同的参数值就能综合出不同位宽的 ALU。以下的 4 位 ALU 示例正是如此。

```
module FourBitALU (A, B, Sel, Cmp, Data);
  parameter ALU_SIZE = 4;
  input [ALU_SIZE -1:0] A, B;
  input[2:0] Sel;
  output Cmp;
  output [ALU_SIZE - 1:0] Data;

  ArithLogicUnit # (ALU_SIZE) InstA (A, B, Sel, Cmp, Data);
endmodule
```

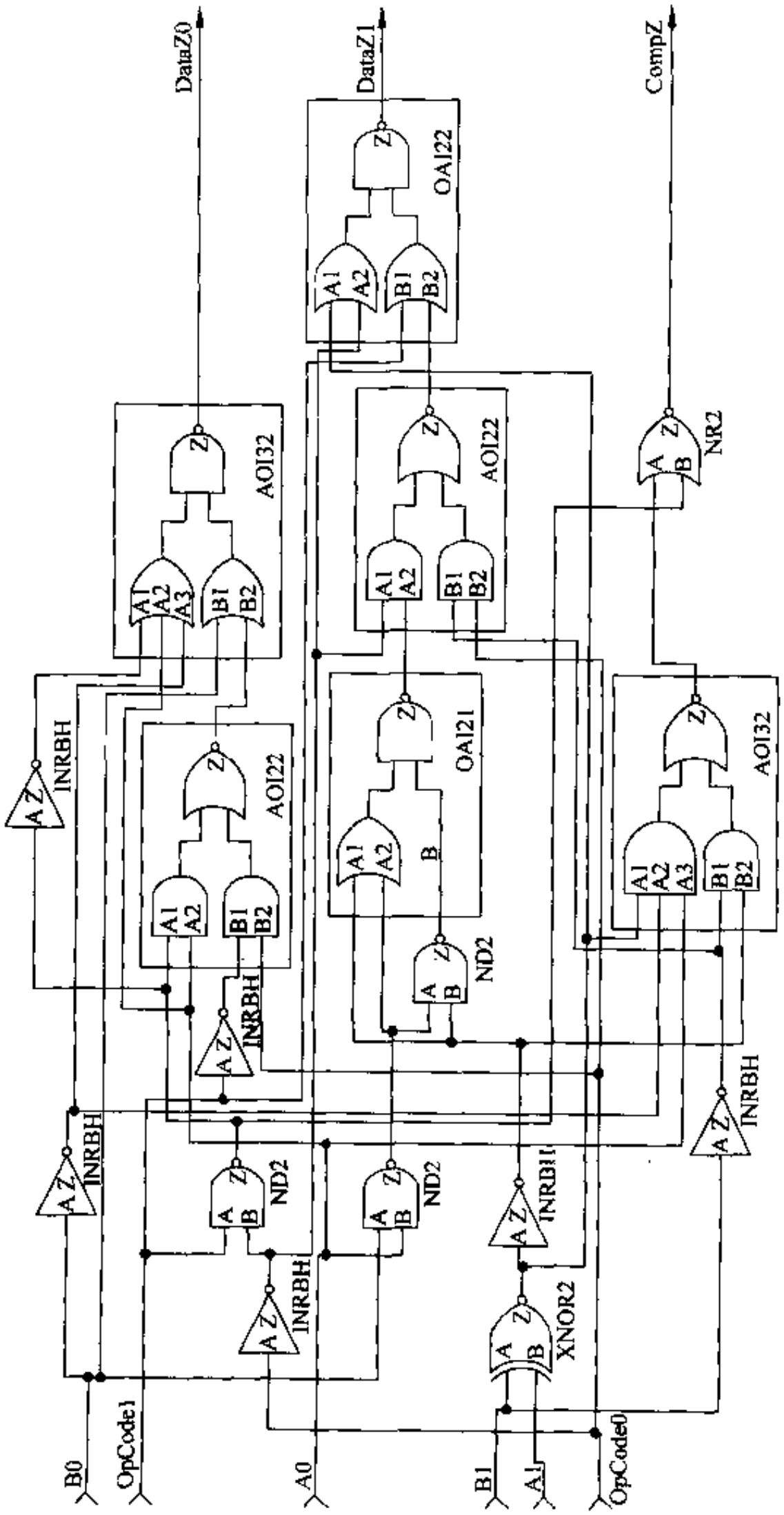

图 3-13 2位的自定义 ALU

3.7.2 简单 ALU

请看另一个简单 ALU 模型。该逻辑单元实现加、与非、大于和异或 4 种功能。带有条件表达式的持续赋值语句被用来对此 ALU 进行建模。

```
module CustomALU (A, B, OpCode, DataZ, CompZ);
  parameter NBITS = 2;
  input [NBITS-1:0] A, B;
  input [1:0] OpCode;
  output [NBITS-1:0] DataZ;
  output CompZ;
  parameter ADD_OP = 0; NAND_OP = 1, GT_OP = 2,
            XOR_OP = 3;
  assign DataZ = (OpCode == ADD_OP) ? A + B :
                 (OpCode == NAND_OP) ? ~ (A & B) :
                 (OpCode == XOR_OP) ? A ^ B :
                 'bx;
  assign CompZ = (OpCode == GT_OP) ? A > B : 'bx;
endmodule
// 综合出的网表如图 3-13 所示
```

3.8 计数器的建模

3.8.1 二进制计数器

请看以下参数化的带同步预置位和预清零控制的 N 位二进制可逆计数器模型。计数与时钟的上升沿同步。

```
module BinaryCounter (Ck, UpDown, PresetClear, LoadData, DataIn, Q, QN);
  parameter NBITS = 2;
  input Ck, UpDown, PresetClear, LoadData;
  input [NBITS-1:0] DataIn;
  output [NBITS-1:0] Q;
```

```
output [NBITS - 1:0] QN;

reg [NBITS - 1:0] Counter;

always @ (posedge Ck)
  if (PresetClear)
    Counter <= 0;
  else if (~ LoadData)
    Counter <= DataIn;
  else if (UpDown)
    Counter <= Counter + 1;
  else
      Counter <= Counter - 1;

  assign Q = Counter;
  assign QN = ~Counter;
endmodule
```

// 2 位的二进制计数器综合出的网表如图 3-14 所示

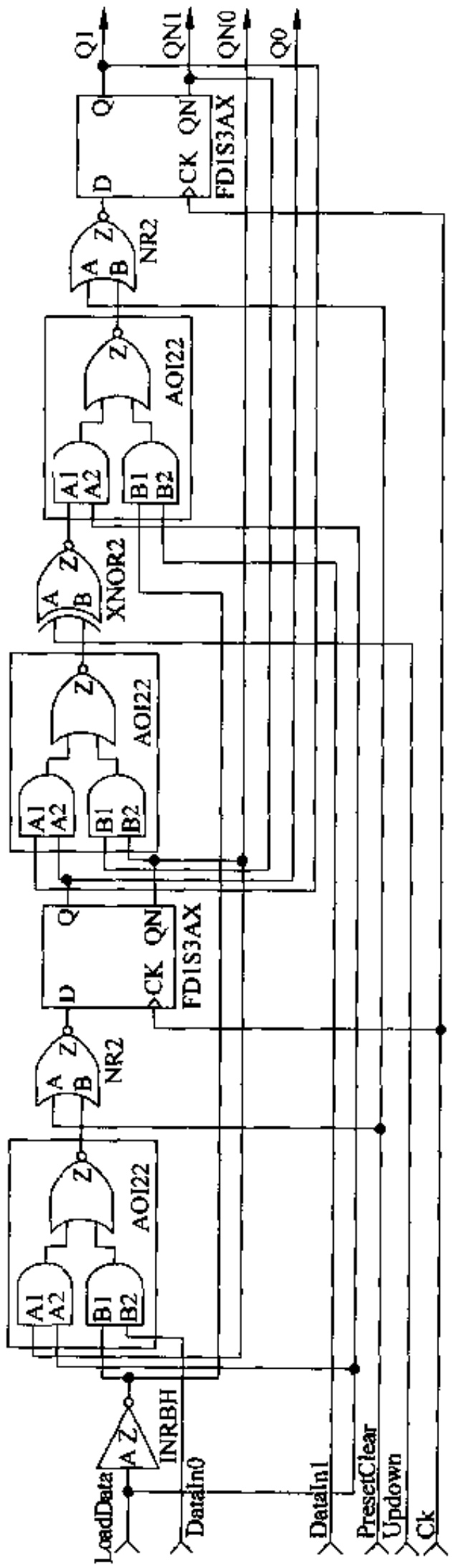

图 3-14 2 位的可清零、可载入的二进制可逆计数器

3.8.2 模 N 计数器

以下是二进制模 N 递增计数器模型。此计数器仅有一个同步预置位控制端，所有状态转移都发生在时钟上升沿。

```
// 计数器位数 : NBITS
// 模数 : UPTO
module ModuloN _ Cntr (Clock, Clear, Q,
QBAR);
  parameter NBITS = 2, UPTO = 3;
  input Clock, Clear;
  output [NBITS - 1:0] Q, QBAR;
  reg [NBITS - 1:0] Counter;

  always @ (posedge Clock)
    if (clear)
      Counter <= 0;
```

```
    else
      Counter <= (Counter + 1) % UPTO;

  assign Q = Counter;
  assign QBAR = ~ Counter;
endmodule
// 模 3 计数器综合出的网表如图 3-15 所示
```

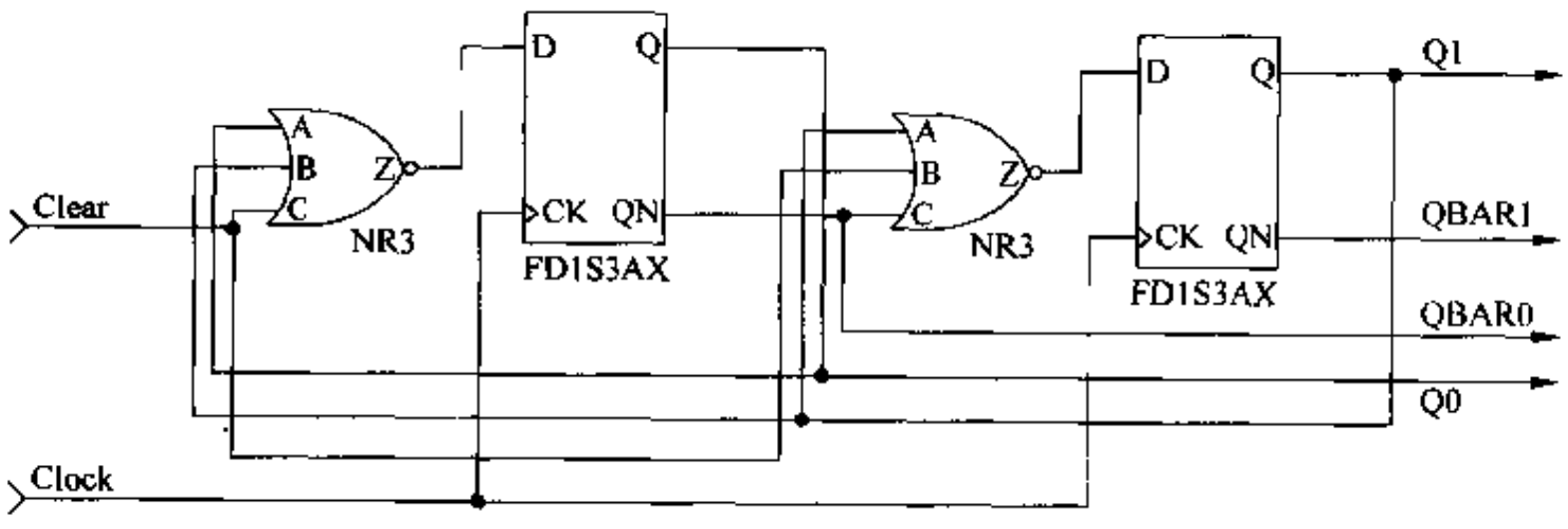

图 3-15　模 3 二进制计数器

3.8.3　约翰逊计数器

约翰逊计数器是移位型计数器。3 位约翰逊计数器的计数过程如下：

```
0 0 0
0 0 1
0 1 1
1 1 1
1 1 0
1 0 0
0 0 0
```

约翰逊计数器建模的关键在于：

a) 如果计数器最高有效位(最左边的那位)为 1,则从右端移入 0。

b) 如果最高有效位为 0,则从右端移入 1。

请看以下参数化的带异步预清零的 N 位约翰逊计数器模型。

```
module JohnsonCounter (ClockJ,PreClear, Q);
  parameter NBITS = 3;
```

```
    input ClockJ, PreClear;
    output [1:NBITS] Q;
    reg [1:NBITS] Q;

    always @ (negedge PreClear or negedge ClockJ)
      if (! PreClear)
        Q <= 0;
      else
      begin
        if (! Q[1])
          Q <= {Q[1:NBITS-1], 1'b1};
        else
          Q <= {Q[1:NBITS-1], 1'b0};
      end
endmodule
// 综合出的3位约翰逊计数器网表如图3-16所示
```

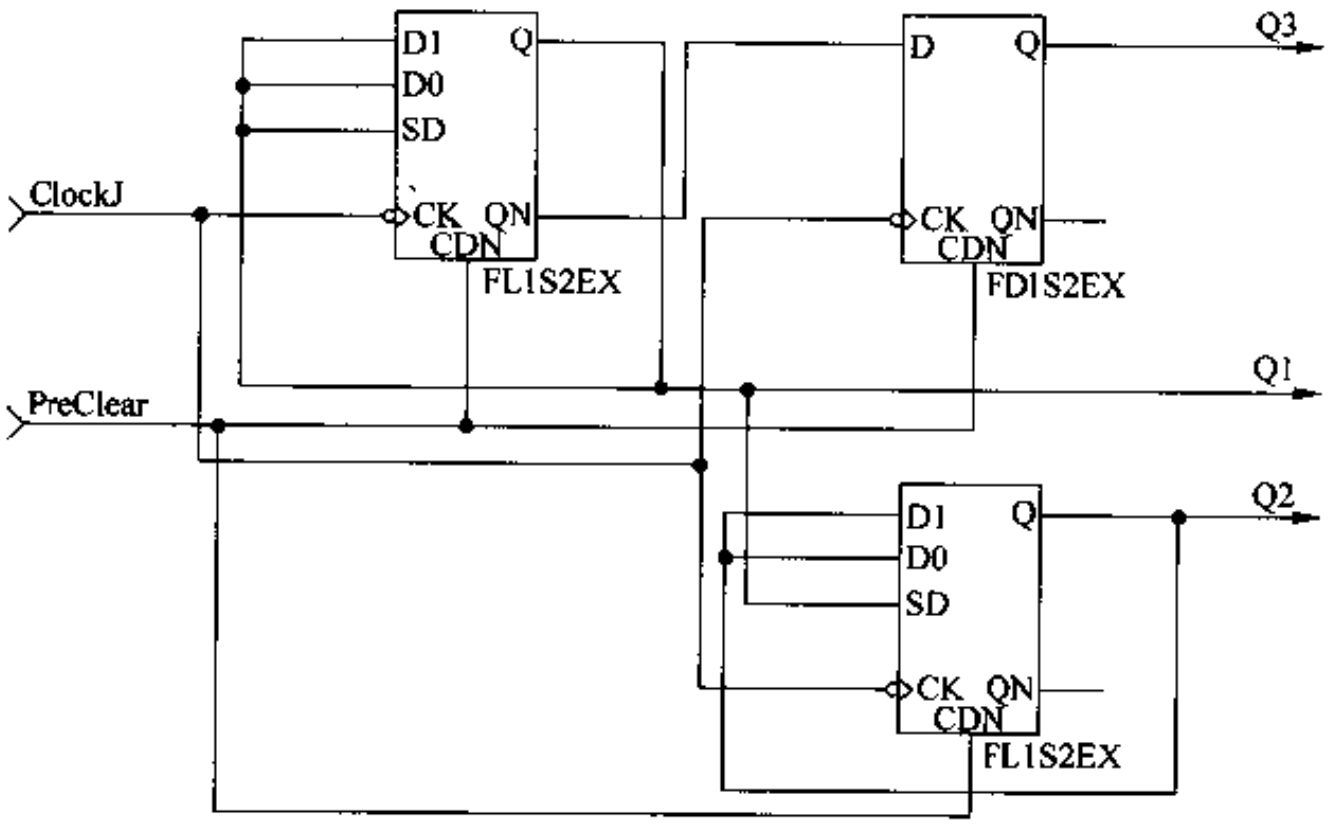

图3-16 3位的约翰逊计数器

3.8.4 格雷码计数器

格雷码计数器是实现以下转换逻辑的二进制计数器：

a) 格雷码第一位(即最左边的位)与其二进制码第一位相同。

b) 格雷码第二位是其二进制码第一位和第二位相异或的结果，依次类推；也就是说，二进制码的每一对相邻位相异或就得到了下一位格雷码。

例如，二进制计数值 4'b1100 对应的格雷码计数值是 4'b1010。请看以下参数化的带同步预清零的 N 位格雷码递增计数器的 Verilog HDL 模型。

```
module GrayCounter (ClockG, Clear, Q, QN);
  parameter NBITS = 3;
  input ClockG, Clear;
  output [1:NBITS] Q, QN;

  reg [1:NBITS] Counter, GrayCount;
  integer K;

  always @ (posedge ClockG)
    if (Clear)
      Counter <= 0;
    else
      Counter <= Counter + 1;

  always @ (Counter)
  begin
    GrayCount[1] = Counter[1];

    for (K = 2; K <= NBITS; K = K+1)
        GrayCount [K] = Counter [K - 1] ^
Counter[K];
  end

  assign Q = GrayCount;
  assign QN = ~GrayCount;
endmodule
```

// 综合出的 3 位格雷码计数器网表如图 3-17 所示

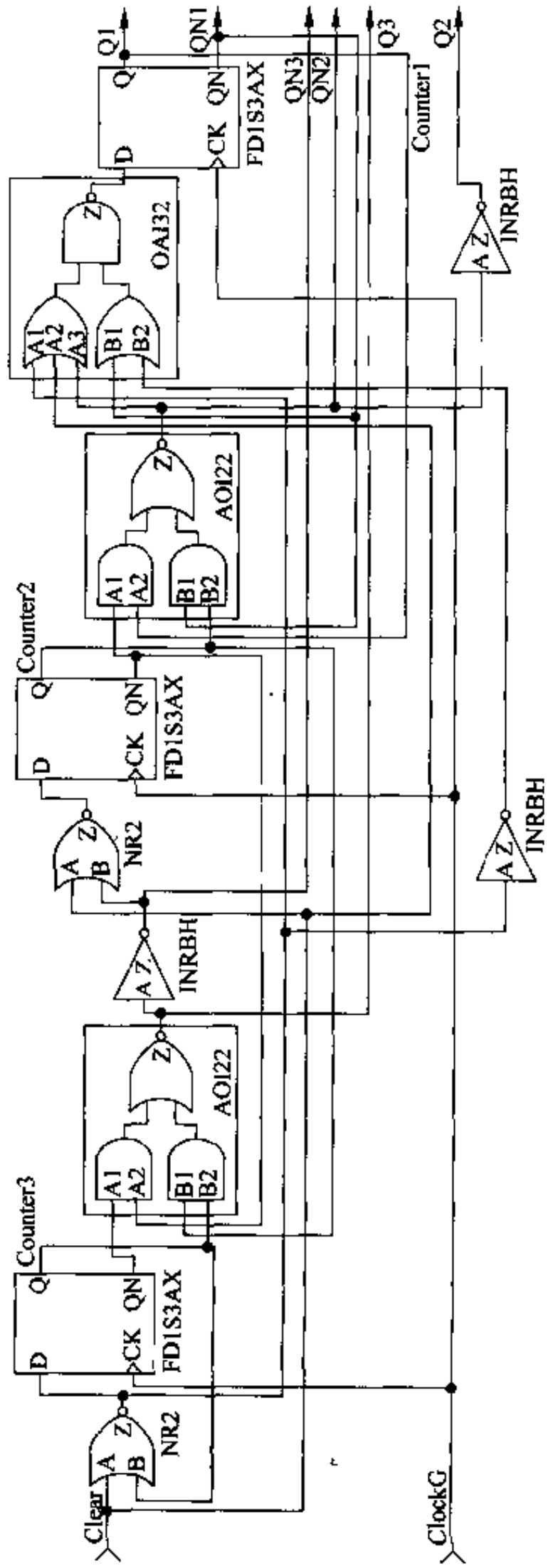

图 3-17　3 位格雷码计数器

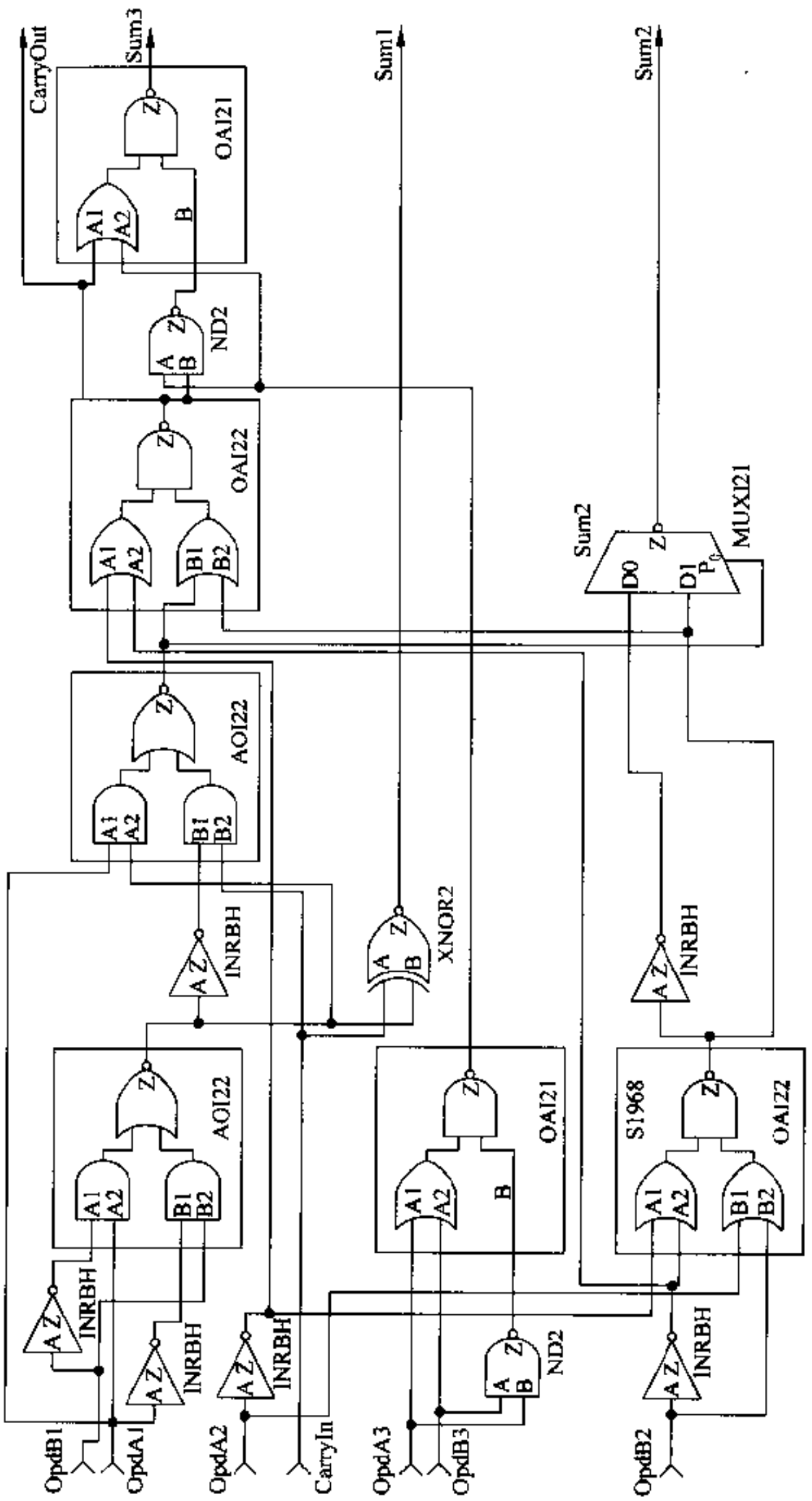

图 3-18 3位的带进位输入和进位输出的加法器

3.9 参数化加法器的建模

请看以下参数化的带进位输入和进位输出的 N 位加法器模型。其最左边的位是最高有效位。带进位输入和进位输出的 3 位加法器所综合出的网表如图 3-18 所示。

```
module AddWithCarryInCarryOut (OpdA, OpdB, CarryIn, CarryOut, Sum);
  parameter NUMBITS = 3;
  input [NUMBITS : 1] OpdA, OPdB;
  input CarryIn;
  output CarryOut;
  output [NUMBITS : 1] Sum;

  assign {CarryOut, Sum} = OpdA + OpdB + CarryIn;
endmodule
// 3 位参数化的加法器综合出的网表如图 3-18 所示
```

3.10 参数化的比较器的建模

请看以下参数化的 N 位二进制比较器模型。输入向量被视为无符号量，以实现数值的比较。

```
module Comparator (A, B, EQ, GT, LT, NE, GE, LE);
  parameter NUMBITS = 2;
  input [NUMBITS : 1] A, B;
  output EQ, GT, LT, NE, GE, LE;

  reg [5:0] ResultBus;
  // ResultBus 的各位从高向低依次是 EQ、GT、LT、NE、GE 和 LE

  always @ (A or B)
    if (A == B)
      ResultBus = 6'b100011;
    else if (A < B)
      ResultBus = 6'b001101;
    else //(A > B)
      ResultBus = 6'b010110;

  assign {EQ, GT, LT, NE, GE, LE} = ResultBus;
endmodule
// 2 位的比较器综合出的网表如图 3-19 所示
```

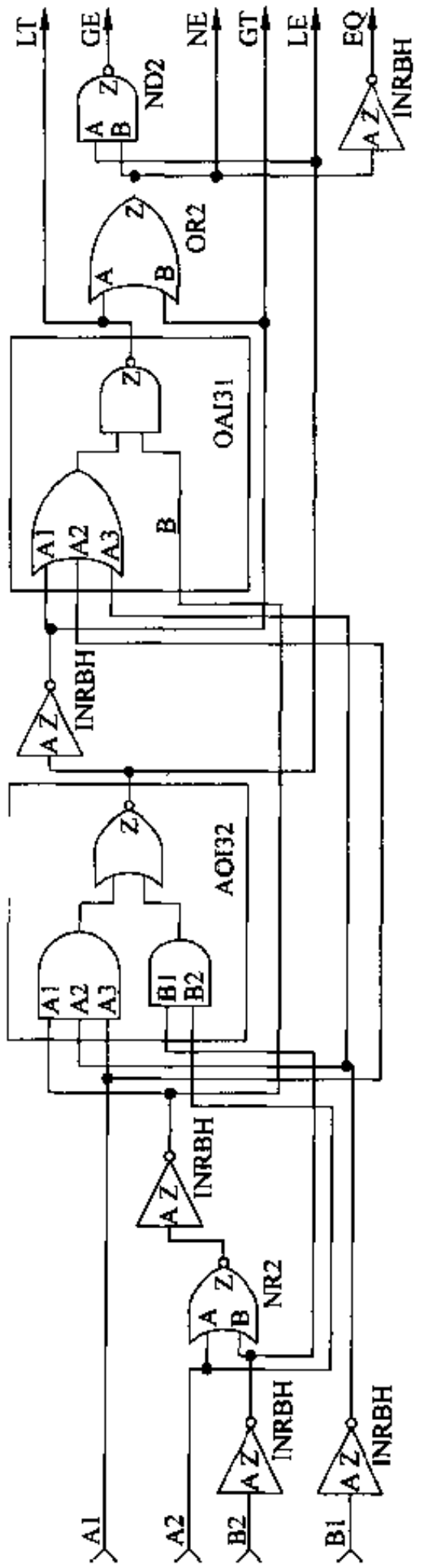

图 3-19 2位的比较器

3.11 译码器的建模

3.11.1 简单译码器

请看以下简单的 2-4 译码电路示例。它仅采用持续赋值语句来实现组合电路的建模。通常,综合系统会忽略赋值语句中所指定的任何延迟。

```
module SimpleDecoder (A, B, Enable, DecodeOut);
  input A, B, Enable;
  output [0:3] DecodeOut;
  wire Abar, Bbar;

  assign Abar = ~ A;
  assign Bbar = ~ B;
  assign DecodeOut[0] = ~ (Enable & Abar & Bbar);
  assign DecodeOut[1] = ~ (Enable & Abar & B);
  assign DecodeOut[2] = ~ (Enable & A & Bbar);
  assign DecodeOut[3] = ~ (Enable & A & B);
endmodule
// 综合出的网表如图 3-20 所示
```

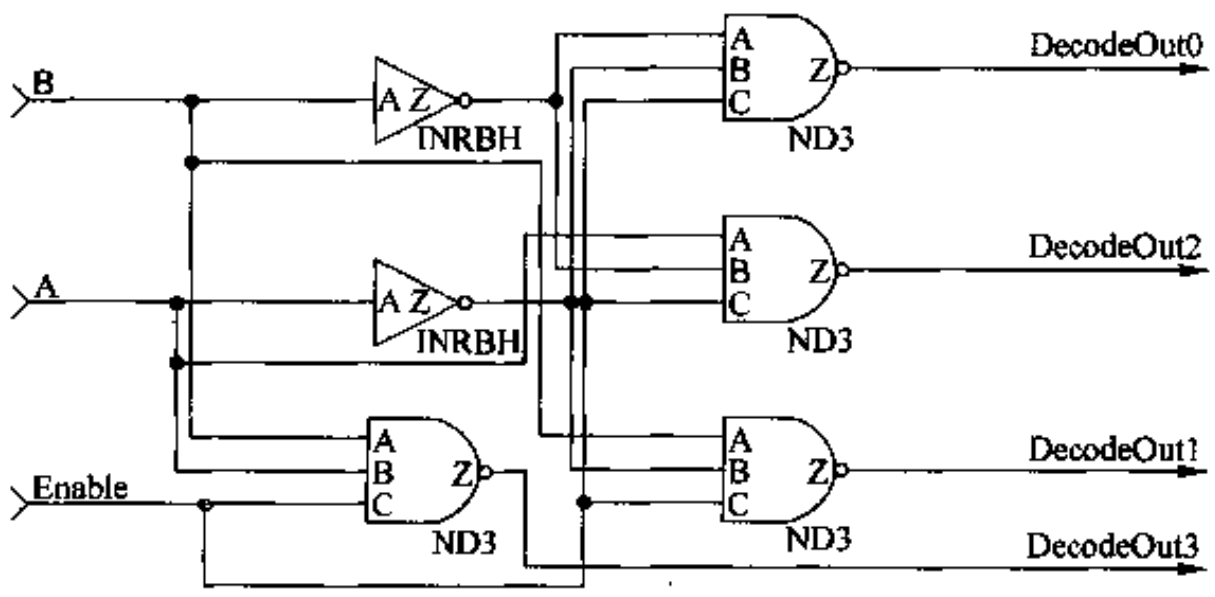

图 3-20 简单的 2-4 译码器

3.11.2 二进制译码器

请看以下参数化的 N 位二进制译码器模型：

```
module BinaryDecoder (SelectAddress, DecodeOut);
  parameter SBITS = 2;
  parameter OUT_BITS = 4;   // SBITS 应是 2 的幂
  input [SBITS - 1:0] SelectAddress;
  output [OUT_BITS - 1:0] DecodeOut;
  reg [OUT_BITS - 1:0] DecodeOut;

  integer k;

  always @ (SelectAddress)
    for(k = OUT_BITS - 1; k >= 0; k = k - 1)
      DecodeOut [k] = (k == SelectAddress) ? 'b1 : 'b0;
endmodule
// 2 位的二进制译码器综合出的网表如图 3-21 所示
```

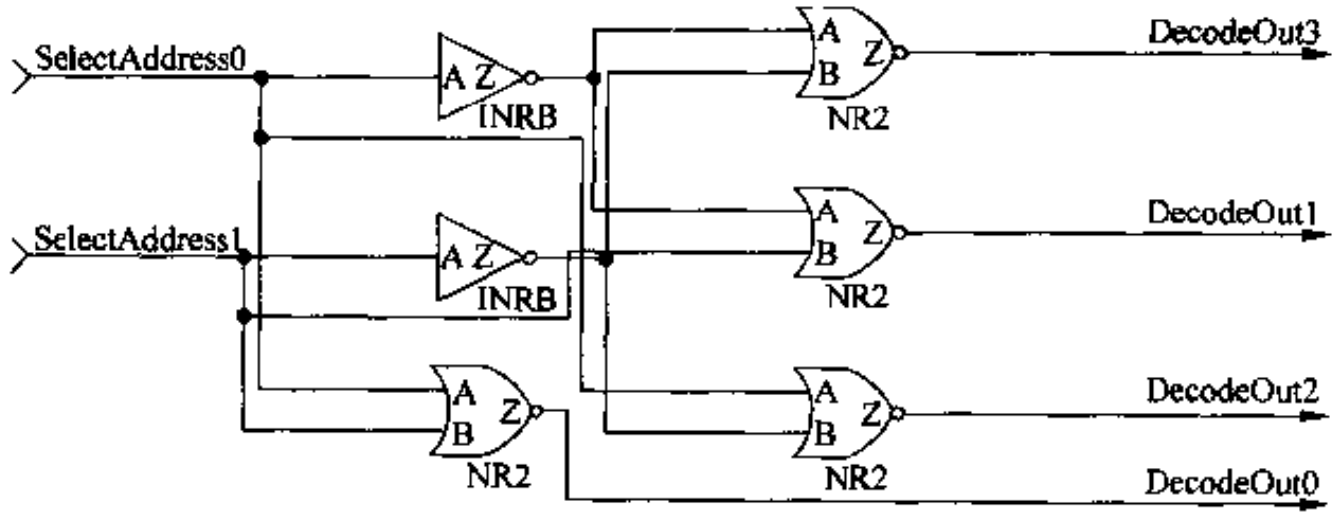

图 3-21 2 位的二进制译码器

3.11.3 约翰逊译码器

请看以下参数化的带有使能控制的 N 位约翰逊译码器模型。

```
module JohnsonDecoder (S, En, Y);
  parameter N = 3;
  input [0:N - 1] S;
  input En;
  output [0:2 * N - 1] Y;
  reg [0:2 * N - 1] Y;

  reg [0:2 * N - 1] Address;
```

```
    integer J;
    always @ (S or En)
      if (En == 'b1)
      begin
        Address = 0;

        for (J = 0; J < N; J = J + 1)
          if (S[J])
            Address = Address + 1;

        if (S[0])
          Address = 2 * N - Address;

        Y = 'b0;
        Y[Address] = 'b1;
      end
      else if (En == 'b0)
        Y = 'b0;
      else
        Y = 'bx;
endmodule
// 3 位的约翰逊译码器综合出的网表如图 3-22 所示
```

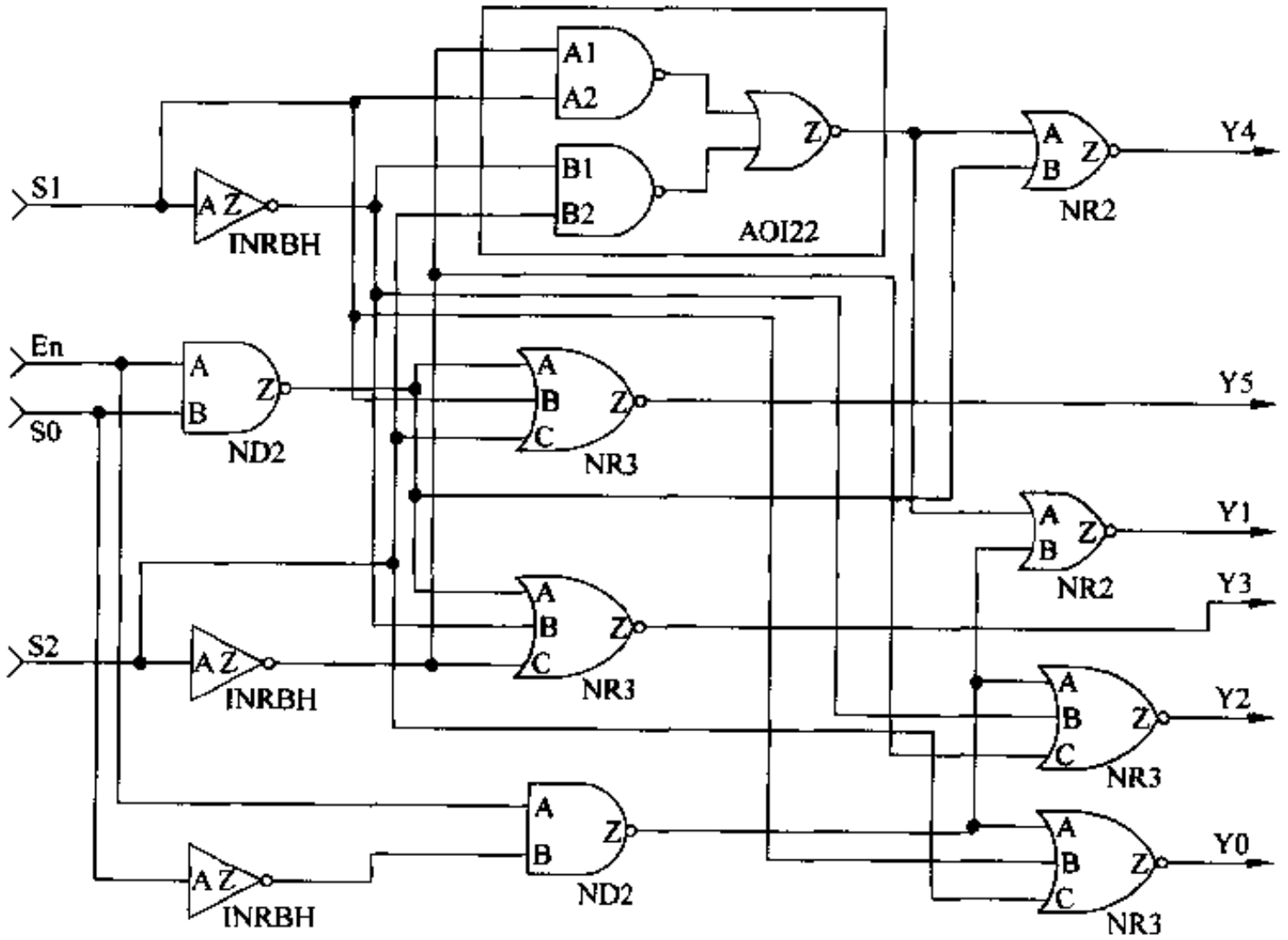

图 3-22　3 位的约翰逊译码器

3.12 多路选择器的建模

3.12.1 简单多路选择器

请看以下4选1的多路选择器电路模型。其中,持续赋值语句中的位选取用于实现组合逻辑的建模。

```
module SimpleMultiplexer (DataIn, SelectAddr, MuxOut);
  input [0:3] DataIn;
  input [0:1] SelectAddr;
  output MuxOut;

  assign MuxOut = DataIn [SelectAddr];
endmodule
// 综合出的网表如图 3-23 所示
```

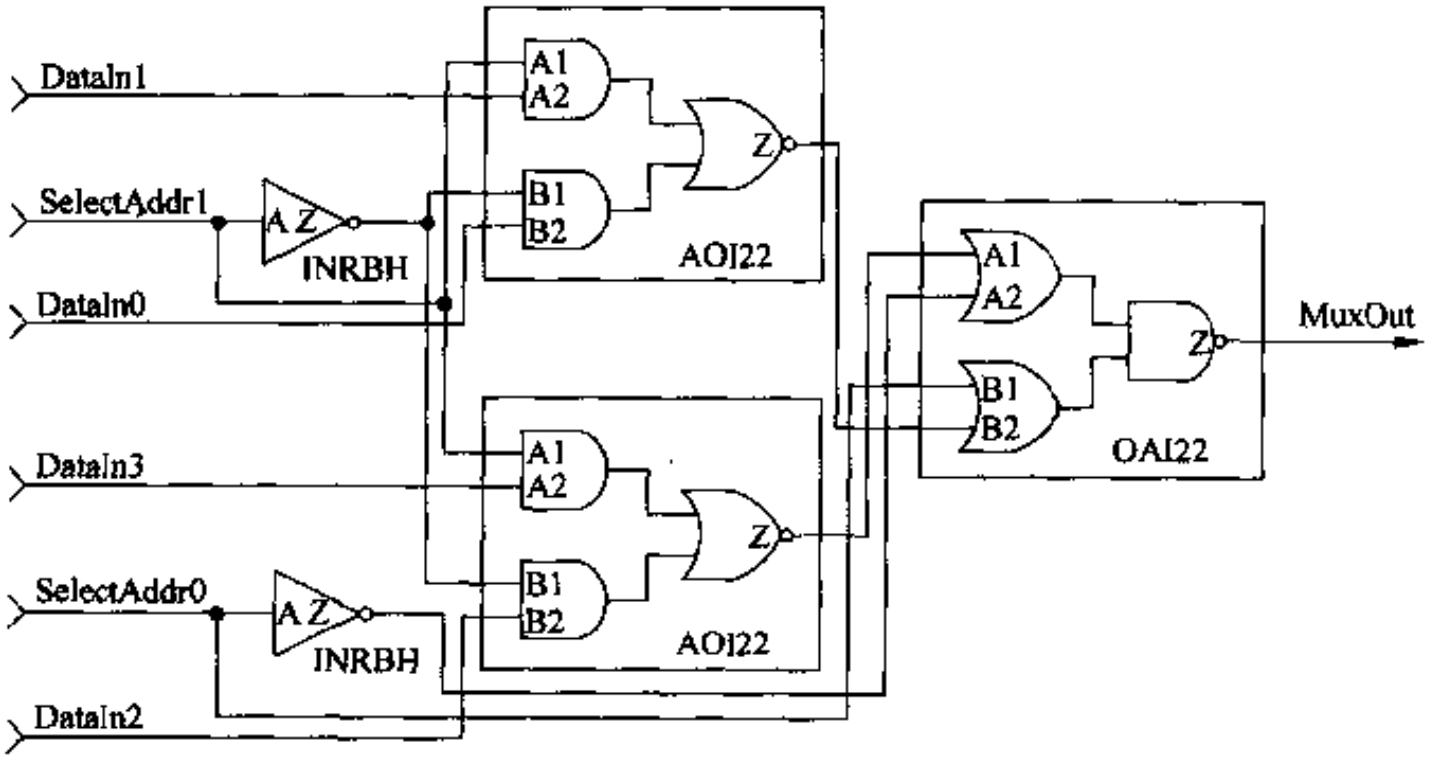

图 3-23 4选1的多路选择器

3.12.2 参数化的多路选择器

请看以下参数化的多路选择器模型示例。该多路选择器中每个字(word)的位数以及字的个数都被建模成参数。输入数据线用单个数组 *DataBus* 表示,多路选择器有若干

个未编码的选择线和一个使能信号，还提供了反相数据输出信号。所有的输出是否呈三态输出都取决于使能信号 *Enable*。

```
module BinaryMultiplexer (DataBus, Select, Enable, Y, Ybar);
  parameter NBITS = 2, WORDS = 2;
  input [NBITS * WORDS - 1:0] DataBus;
    // DataBus 是线性的二维数组
  input [WORDS - 1:0] Select;        // 译码选择线
  input Enable;
  output [NBITS - 1:0] Y, Ybar;
  reg [NBITS - 1:0] Y, Ybar;
  integer K;
  function [WORDS - 1:0] GetWordIndex;
    // 返回第一个 1 所在位的下标值
    input [WORDS - 1:0] DecodedSelect;
    integer Inx;
    begin
      GetWordIndex = 0;
      for (Inx = WORDS - 1; Inx > = 0; Inx = Inx - 1)
        if (DecodedSelect [Inx] == 'b1)
          GetWordIndex = Inx;
    end
  endfunction
  always @ (DataBus or Select or Enable)
    if (Enable == 'b1)
    begin
      for (K = 0; K < NBITS; K = K + 1)
        Y [k] = DataBus [GetWordIndex (Select) * NBITS + K];

      Ybar = ~Y;
    end
    else if (Enable == 'b0)
    begin
      Y = 'bz;
      Ybar = 'bz;
    end
    else
    begin
      Y = 'bx;
      Ybar = 'bx;
```

```
  end
endmodule
// 2×2 多路选择器综合出的网表如图 3-24 所示
```

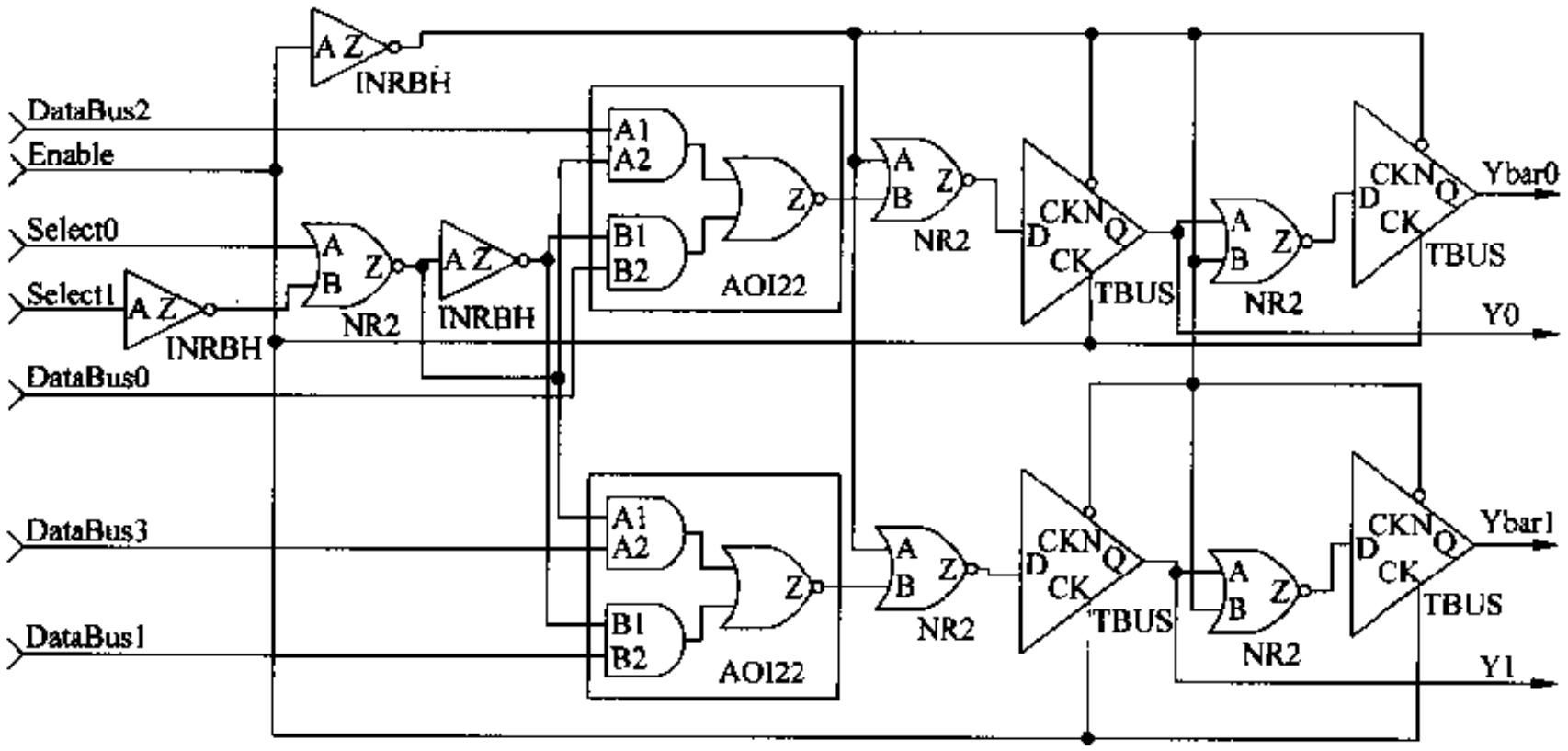

图 3-24 2×2 的二进制多路选择器

3.13 参数化的奇偶校验生成器的建模

请看以下参数化的 N 位奇偶校验生成器电路模型。此模型同时提供了奇校验输出和偶校验输出。

```
module ParityGenerator (DataIn, OddPar, EvenPar);
  parameter NBITS = 4;
  input [NBITS-1:0] DataIn;
  output OddPar, EvenPar;
```

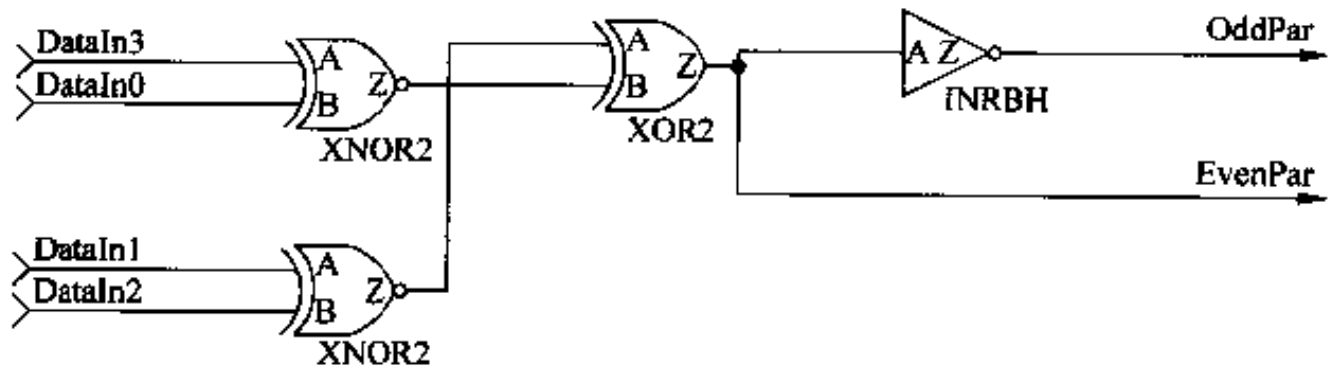

图 3-25 4 位奇偶校验生成器

```
  assign EvenPar = ^ DataIn;
  assign OddPar = ~ Evenpar;
endmodule
// 4 位奇偶校验生成器综合出的网表如图 3-25 所示
```

3.14 三态门的建模

三态门的建模是通过在某种条件的控制下对变量赋 z 值来实现的。请看下例：

```
module ThreeStateGates (ReadState, CpuBus, MainBus);
  input ReadState;
  input [0:3] CpuBus;
  output [0:3] MainBus;
  reg [0:3] MainBus;

  always @ (ReadState or CpuBus)
    if (ReadState)
      MainBus = 4'bz;
    else
```

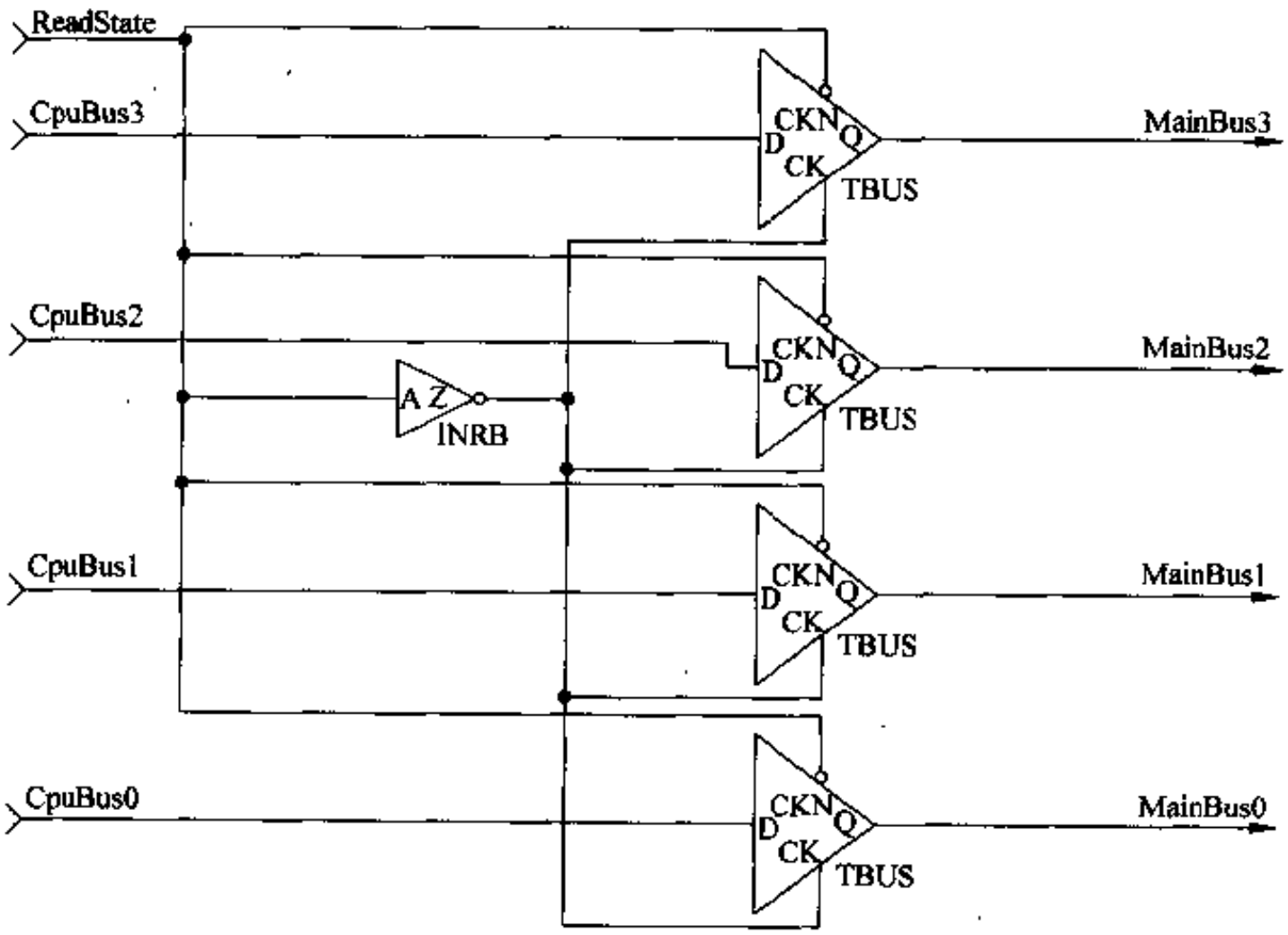

图 3-26 一组三态门

```
        MainBus = CpuBus;
    endmodule
    // 综合出的网表如图 3-26 所示
```

只要 *ReadState* 为真，*MainBus* 就是三态的。若 *ReadState* 为假，则把 *CpuBus* 的值赋给 *MainBus*。

3.15 数据流检测模型

以下模型能检测出输入 *Data* 所形成的数据流上是否出现连续的 3 个“1”。在时钟的每个下降沿上检测输入信号，若发现 3 个连续的“1”，则将输出信号置为真，否则置为假。

```
module Count3Ones (Data, Clock, Reset, SeqFound);
  input Data, Clock, Reset;
  output SeqFound;
  reg SeqFound;
  parameter PATTERN_SEARCHED_FOR = 3'b111;
  reg [2:0] Previous;

  always @ (negedge Clock)
    if (Reset)
    begin
      Previous <= 3'b000;
      SeqFound <= 1'b0;
    end
    else
    begin
      Previous <= {Previous [1:0], Data};
      SeqFound <= (Previous == PATTERN_SEARCHED_FOR);
    end
endmodule
// 综合出的网表如图 3-27 所示
```

综合此模型会推导出 4 个触发器，3 个用于 *Previous*，另一个用于 *SeqFound*。但是，稍加优化，就会发现用于 *Previous* 的触发器中有一个是不必要的，因而可以去除该触发器。此模型中，锁存了输出 *SeqFound*，这是因为对它的赋值受时钟沿的控制。如果不打算锁存输出，则对 *SeqFound* 的赋值必须放在 always 语句之外进行。这样就得到了以下

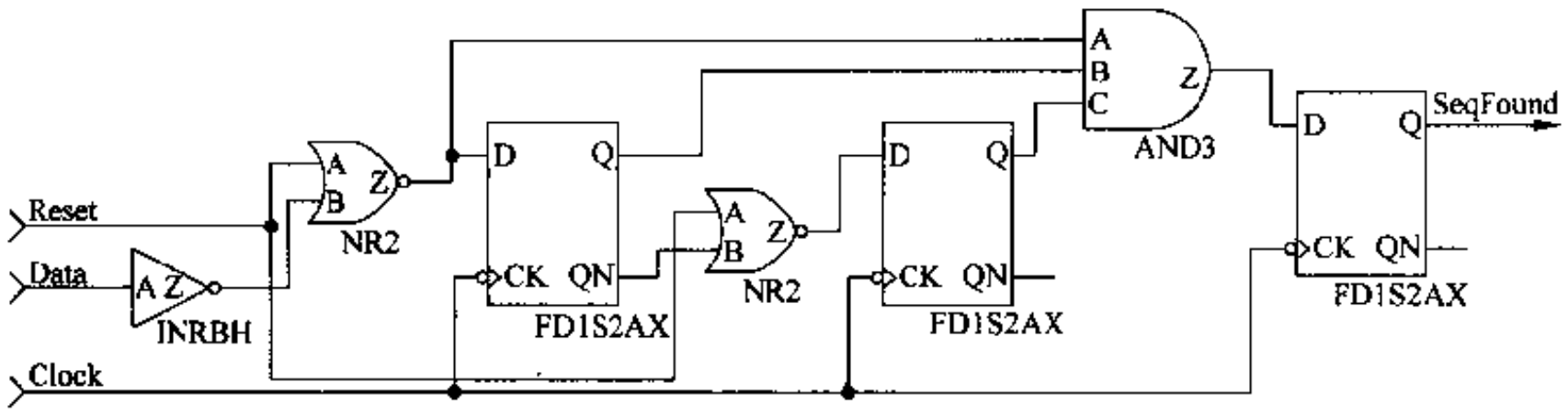

图 3-27　统计出现连续 3 个“1”的电路

模块：

```
module NoLatchedOutput (Data, Clock, Reset, SeqFound);
  input Data, Clock, Reset;
  output SeqFound;

  parameter PATTERN_SEARCHED_FOR = 3'b111;
  reg [2:0] Previous;

  always @ (negedge Clock)
    if (Reset)
      Previous <= 3'b000;
    else
      Previous <= {Previous [1:0], Data};

  assign SeqFound = (Previous == PATTERN_SEARCHED_FOR);
endmodule
// 综合出的网表如图 3-28 所示
```

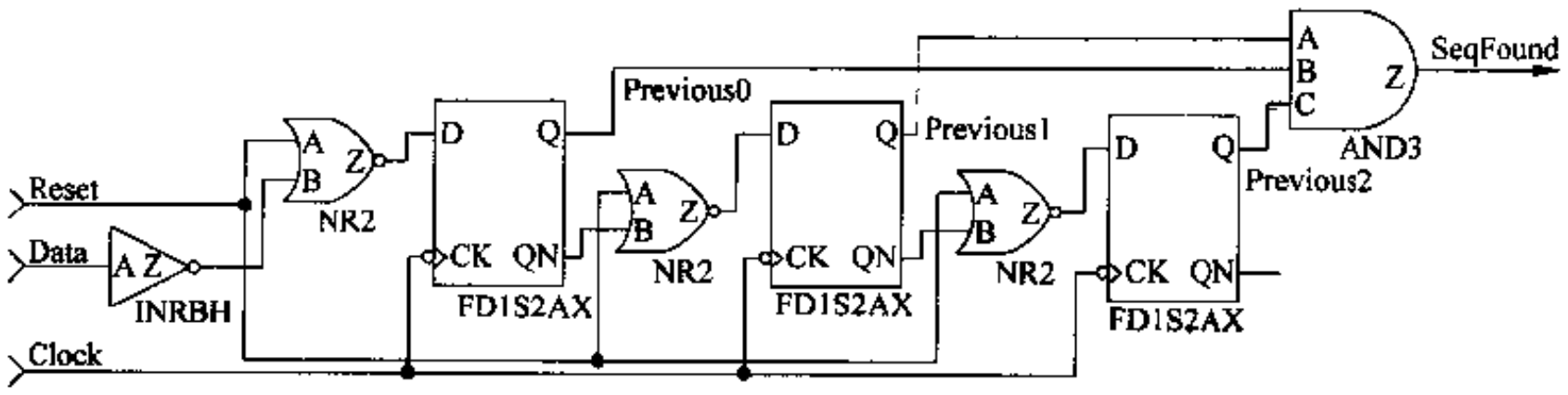

图 3-28　未锁存的输出

此模块中，未锁存输出 *SeqFound*，变量 *Previous* 被综合成 3 个触发器。注意，在这种情况下必须锁存 *Previous* 的每一位。

3.16 阶乘模型

以下模型能生成给定输入数据 *Data* 的阶乘，其输出结果 *Result* 和 *Exponent* 分别是尾数和指数，指数以 2 为基。输入信号 *Reset* 用来使模型复位。

```
module Factorial (Reset, Start, Clk, Data, Done,
                  Result, Exponent);
  input Reset, Start, Clk;
  input [4:0] Data;
  output Done;                  // 确认信号
  reg Done;
  output [7:0] Result, Exponent;
  reg [7:0] Result, Exponent;
  reg [4:0] InLatch;

always @ (posedge Clk)
begin: BLOCK_A
  integer NextResult, J;

  if ((Start && Done) || Reset)
  begin
    Result <= 'b1;
    Exponent <= 'b0;
    InLatch <= Data;
    Done <= 'b0;
  end
  else
  begin
    if ((InLatch > 1) && (! Done))
    begin
      NextResult = Result * InLatch;
      InLatch <= InLatch - 1;
    end
    else
```

```
        NextResult = Result;

      if (InLatch <= 1)
        Done <= 'b1;

      for (J = 1; J <= 5; J = J + 1)
      begin
        if (NextResult > 256)
        begin
          NextResult = NextResult >> 1;
          Exponent <= Exponent + 1;
        end
      end

      Result <= NextResult;
    end
  end
endmodule
```

综合后，变量 *InLatch*、*Result*、*Exponent* 和 *Done* 均被推导成触发器。

3.17 UART 模 型

请看以下可综合的 *UART* 电路模型。此电路将 RS-232 串行输入数据转换为并行数据输出，将并行输入数据转换为 RS-232 串行数据输出。数据字节的位宽为 8。如图 3-29 所示，此 *UART* 模型有 4 个主要的模块：*RX* 为接收器模块，*TX* 为发送器模块，*DIV* 为时钟分频器模块，*MP* 为微处理器模块。

第一个模块 *DIV* 是一个分频器，有两种运行模式：常规模式和测试模式。处于测试模式时，UART 运行速度是常规模式的 16 倍，数据发送率也是接收率的 16 倍。通过对端口 *MR* 置 0 使复位线置为低电平，从而使各模块都得以初始化。*TX* 模块接受来自微处理器接口模块 *MP* 的 8 位并行数据，并通过端口 *DOUT* 将数据串行输出到 RS-232 端口。反过来，*RX* 模块接收串行数据输入，并以 8 位并行格式发送至 *MP* 模块。再说一次，发送器以接收器的 16 倍速率工作。微处理器接口模块 *MP* 异步地控制 *RX*/*TX* 模块与微处理器数据总线之间的并行数据流。

UART 的顶层模型用模块实例化的方法将各模块结合在一起。微处理器实体 *MP*

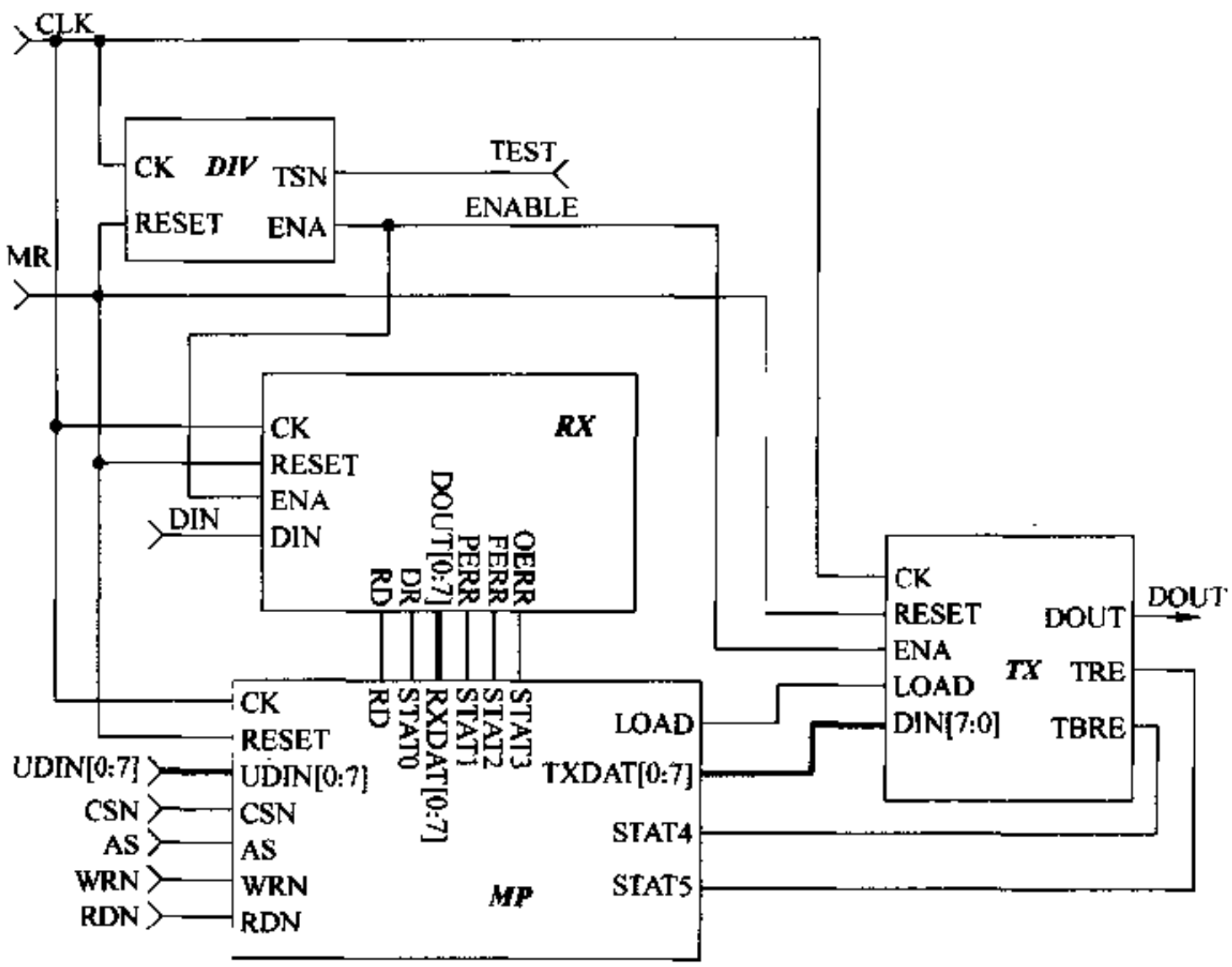

图 3-29 UART 电路

以结构化方式描述，也就是使用模块实例加以描述；其余 3 个模块以行为方式加以描述。本章只介绍行为级模块。

以下是发送器模块 TX 的行为模型，它是可综合的模型。变量 *TBR*、*TR*、*TRE*、*TBRE*、*DOUT*、*CBIT* 和 *PA* 都被推导成上升沿触发的触发器，这是因为对这些变量的赋值受时钟 *CK* 的控制。

```
module TX (CK, RESET, ENABLE, TLOAD, DIN, DOUT, TRE, TBRE);
  input CK, RESET, ENABLE, TLOAD;
  input [7:0] DIN;
  output DOUT, TRE, TBRE;

  reg DOUT, TRE, TBRE;

  reg [7:0] TBR, TR;
  reg [3:0] CBIT;
  reg PA;
```

```
always @ (posedge CK)
begin
  if (! RESET)
  begin
    TRE <= 'b1;
    TBRE <= 'b1;
    DOUT <= 'b1;
    CBIT <= 4'b0;
    PA <= 'b0;
  end
  else if (TLOAD)
  begin
    TBR <= DIN;
    TBRE <= 'b0;
  end
  else if (ENABLE)
  begin
    if (! TBRE && TRE)
begin
  TR <= TBR;
  TRE <= 'b0;
  TBRE <= 'b1;
end

if (! TRE)
  case (CBIT)
    4'b0000:
      begin
        DOUT <= 'b0;
        CBIT <= CBIT + 1;
      end
    4'b0001, 4'b0010, 4'b0011, 4'b0100,
    4'b0101, 4'b0110, 4'b0111, 4'b1000:
      begin
        DOUT <= TR[0];
        PA <= PA ^ TR[0];
        TR <= {1'b1, TR[7:1]};
```

```
        CBIT <= CBIT + 1;
      end
    4'b1001:
      begin
        DOUT <= PA;
        PA <= 'b0;
        TR <= {1'b1, TR[7:1]};
        CBIT <= CBIT + 1;
      end
    4'b1010:
      begin
        DOUT <= TR[0];
        TR <= {1'b1, TR[7:1]};
        CBIT <= CBIT + 1;
      end
    4'b1011:
      begin
        DOUT <= TR[0];
        TRE <= 1'b1;
        TR <= {1'b1, TR[7:1]};
        CBIT <= 4'b0000;
      end
    endcase
  end          // if (ENABLE)
end            // @ (posedge CK)
endmodule
```

以下是接收器模块 *RX* 的行为模型，它也是可综合的。变量 *START*、*CBIT*、*CSAM*、*DI*、*PI*、*SR*、*DR*、*DOUT*、*PERR*、*FERR* 和 *OERR* 均被推导成触发器。

```
module RX (CK, RESET, ENA, DIN, RD, DR, DOUT, PERR, FERR, OERR);
  input CK, RESET, ENA, DIN, RD;
  output DR;
  reg DR;
  output [7:0] DOUT;
  reg [7:0] DOUT;
  output PERR, FERR, OERR;
  reg PERR, FERR, OERR, START;
```

```
reg [3:0] CBIT, CSAM;
reg DI, PI;
reg [7:0] SR;

always @ (posedge CK)
begin
  if (! RESET)
  begin
    CBIT <= 0;
    CSAM <= 0;
    START <= 0;
    PI <= 0;
    DR <= 0;
    PERR <= 0;
    FERR <= 0;
    OERR <= 0;
  end          // if (RESET)
  else
  begin
if (RD)
  DR <= 0;

if (ENA)
  if (! START)
  begin
    if (! DIN)
    begin
      CSAM <= CSAM + 1;
      START <= 1;
    end
  end
  else if (CSAM == 8)
  begin
    DI <= DIN;
    CSAM <= CSAM + 1;
  end
  else if (CSAM == 15)
    case (CBIT)
```

```
0:
  begin
    if (DI == 1)
      START <= 0;
    else
      CBIT <= CBIT + 1;
      CSAM <= CSAM + 1;
    end
  1, 2, 3, 4, 5, 6, 7. 8:
    begin
      CBIT <= CBIT + 1;
      CSAM <= CSAM + 1;
      PI <= PI ^ DI;
      SR <= {DI, SR[7:1]};
    end
  9:
    begin
      CBIT <= CBIT + 1;
      CSAM <= CSAM + 1;
      PI <= PI ^ DI;
    end
10:
  begin
    PERR <= PI;
    PI <= 0;

    if (! DI)
      FERR <= 1;
    else
      FERR <= 0;

    if (DR)
      OERR <= 1;
    else
      OERR <= 0;

    DR <= 1;
    DOUT <= SR;
```

```
                CBIT <= 0;
                START <= 0;
              end
        endcase
        else  // ((0 <= CSAM < 8)|| (8 < CSAM < 15))
          CSAM <= CSAM + 1;
      end              // if (! RESET)
    end                  // @ (posedge CK)
  endmodule
```

以下是分频器模块 *DIV* 的可综合模型。此电路每 16 个时钟周期产生一个脉冲。如果输入 *TESTN* 为 0,则 *ENA* 置为 1。变量 *COUNT* 被推导成触发器。

```
module DIV (CK, RESET, TESTN, ENA);
  input CK, RESET, TESTN;
  output ENA;

  reg [3:0] COUNT;

  always @ (posedge CK)
    if (! RESET)
      COUNT <= 0;
    else if (! TESTN)
      COUNT <= 4'hF;
    else
      COUNT <= COUNT + 1;  // 加 1 计数

  // 组合逻辑部分
  assign ENA = (COUNT == 15);
endmodule
```

3.18 纸牌 21 点模型

请看以下纸牌 21 点程序的可综合模型。用一副纸牌来玩 21 点程序。牌 2 到 10 的得分和牌面相等,牌 A 的得分可为 1 或 11。游戏的目标是接收若干张随机产生的纸牌,使得总分(所有牌的分数总和)尽可能接近 21 而又不超过 21。

输入 *InsertCard* 表明程序已准备好接收一张新牌。在 *Clock* 上升沿处，若 *InsertCard* 为真，则接收新牌。输入 *CardValue* 表示新牌的牌值。如果接收了若干张牌之后总分在 17 到 21 之间，则置输出 *Won* 为真以表明游戏获胜。如果总分超过 21，那么程序检查上一次是否收到牌 A 并判断已将其接收为 1 还是 11。如果上一张接收到的是牌 A 并已将得分定为 11，则把得分由 11 改为 1，并准备好接收下一张新牌；否则，置输出 *Lost* 为真以表明落败。如果 *Won* 和 *Lost* 两者之一被置位，则将不再接收新牌。将 *NewGame* 置为真，就可让游戏重新开始。

```
module Blackjack (CardValue, Clock. InsertCard,NewGame, TotalPoints, Won, Lost);
  input [0:3] CardValue;
  input Clock, InsertCard, NewGame;
  output [0:5] TotalPoints;
  output Won, Lost;
  reg Won, Lost;
  reg AceAvailable, AceValueIs11;
  reg [0:5] TotPts;
  parameter TRUE = 1'b1, FALSE = 1'b0;

  always @ (posedge NewGame or posedge Clock)
  if (NewGame)
  begin
    Won <= FALSE;
    Lost <= FALSE;
    AceAvailable = FALSE;
    AceValueIs11 = FALSE;
    TotPts = 0;
  end
  else       // 上升沿
    if (InsertCard && ! Won && ! Lost)
    begin
      if (CardValue == 4'd11)
      begin
        AceAvailable = TRUE;
        AceValueIs11 = TRUE;
      end

      TotPts = TotPts + CardValue;

      if ((TotPts >= 17) && (TotPts <= 21))
```

```
        Won <= TRUE
      else if ((TotPts >= 22) && (TotPts <= 31))
      begin
        if (AceAvailable && AceValueIs11)
        begin
          AceValueIs11 = FALSE;
          TotPts = TotPts - 10;
        end
        else
          Lost <= TRUE;
        end
      end

    assign TotalPoints = TotPts;
endmodule
```

第 4 章　模型的优化

本章将介绍可用来改善 Verilog HDL 模型的电路性能的各种优化手段。在 C 语言编译器中，优化器通过代码重组、代码移位等手段产生优化的机器码以减少 C 语言代码的执行时间。逻辑优化器也可以采用这些优化手段。此外，综合所生成的逻辑易受模型描述方式的影响。把语句从一个位置移到另一个位置，或者拆分表达式都会对所生成的逻辑产生重大影响，这可能会造成综合出的逻辑门数有所增减，也可能改变其定时特性。

如图 4-1 所示，由 Verilog HDL 代码综合出的网表所提供的优化起点不同，决定了逻辑优化器所能到达的优化终点(最佳面积和最佳速度)也是不同的。采用不同结构改写同一个 Verilog HDL 模型会得到不同的优化起点。不幸的是，目前尚无一种算法能确定什么样的编码方式或优化手段能得到所期望的面积和延迟之间的均衡。

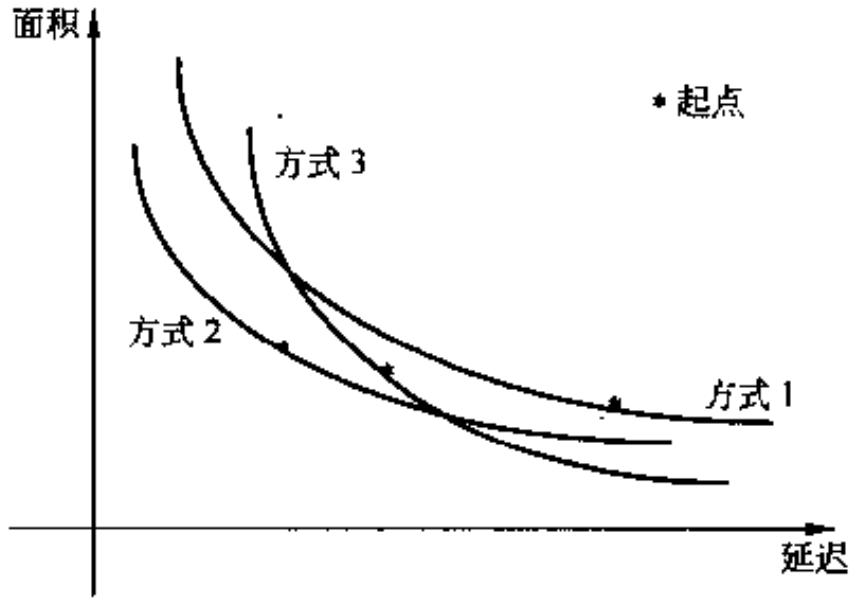

图 4-1　不同的编程方式得到不同的面积/延迟折中

本章将探讨各种优化手段，设计者通过改写 Verilog HDL 综合模型的相应代码就能采用这些优化手段。这些手段可用来减少算术算符和关系算符的数量，从而得到更高品质的设计，并相应地缩短综合过程的运行时间。

4.1　资 源 分 配

资源分配指的是互斥条件下共享算术逻辑单元(ALU)的过程。请考虑以下 if 语句：

```
if (MAX > 100)
```

```
    JMA = SMA + BMA;
  else
    JMA = SMA - CMA;
```

如果不采用资源分配，算符“+”和“-”就会被综合成两个单独的 ALU。但是，如果采用了资源分配，仅需一个 ALU 就可以实现“+”和“-”这两种运算。这是因为这两种算符总是互斥地使用。此外还生成了一个多路选择器，用来从 *BMA* 和 *CMA* 中选择合适的量接到 ALU 的第二个输入端上。图 4-2 是未采用资源分配时综合 if 语句所得到的硬件电路，图 4-3 是该例采用了资源分配所综合出的硬件电路。

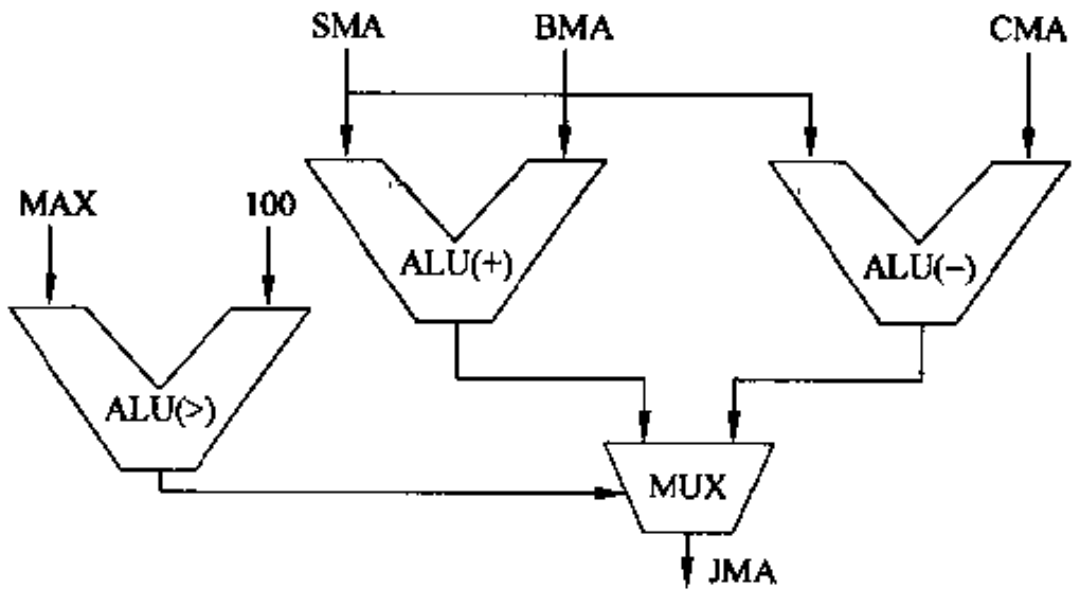

图 4-2　未采用资源分配

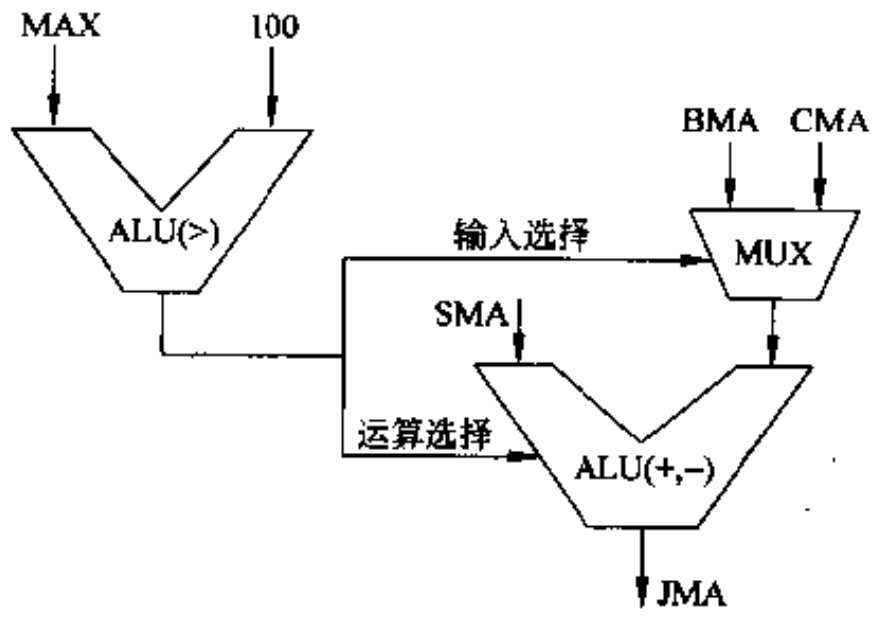

图 4-3　采用了资源分配

注意：共享 ALU，则须在 ALU 的某个输入端引入多路选择器，而这会增加路径的延迟。不过，由于共享了 ALU，生成的逻辑门数量减少了。如果综合工具不具备权衡的能力，设计者就不得不自己进行权衡。在“定时至上”的设计中，最好不要采用资源共享。

还有另外一些综合工具能够自动施加共享资源的变种方法。通常，能够共享的算符有：

a) 关系算符

b) 加法

c) 减法

d) 乘法

e) 除法

通常不值得为实现加法和乘法运算而生成一个 ALU。乘法算符和除法算符往往只在同类算符间实现共享。共享算符有以下几种可能的情况：

a) 算符相同，运算量相同：显然必须共享。例如：$A+B$ 与 $A+B$

b) 算符相同，有一个运算量不同：需权衡，因为需引入一个多路选择器。例如：$A+B$ 与 $A+C$

c) 算符相同，运算量都不同：须权衡，因为须引入两个多路选择器。例如：$A+B$ 与 $C+D$

d) 算符不同，运算量相同：共享有益。例如：$A+B$ 与 $A-B$

e) 算符不同，有一个运算量不同：须权衡，因为须引入一个多路选择器。例如：$A+B$ 与 $A-C$

f) 算符不同，运算量都不同：须权衡，因为须引入两个多路选择器。例如：$A+B$ 与 $C-D$

第 a)种情况可能是共享的最佳情况，其次依次是(d))、(b)，e))和(c)，f))。

改写模型也能实现资源分配。如下例所示：

```
if (! ShReg)
  DataOut = AddrLoad + ChipSelectN;
else if (ReadWrite)
  DataOut = ReadN + WriteN;
else
  DataOut = AddrLoad + ReadN;

// 手工进行资源分配后的代码：
if (! ShReg)
begin
  Temp1 = AddrLoad;
  Temp2 = ChipSelectN;
end
else if (ReadWrite)
begin
  Temp1 = ReadN;
  Temp2 = WriteN;
end
```

```
else
begin
  Temp1 = AddrLoad;
  Temp2 = ReadN;
end

DataOut = Temp1 + Temp2;
```

修改后的模型只产生一个加法器，且 if 语句隐含了连接在该加法器输入端的多路选择器。而最初的示例会综合出 3 个加法器。

4.2 公共子表达式

尽可能找出公共子表达式并重用已计算过的值，这在实践中往往很有用。请看以下简单示例：

```
Run = R1 + R2;
...
Car = R3 - (R1 + R2);
// 假定每次执行第二条赋值语句时第一条赋值语句已执行过了
// 注意：当这两条语句中的任何一条位于 if 语句和 case 语句内时，该假定就有可能不成立
```

综合工具如果不能找出公共子表达式，就会产生两个加法器，它们都会计算出相同的结果 $R1+R2$。逻辑优化工具也许能或不能找出这样的公共逻辑，这样就会导致设计结果更大。所以，找出公共子表达式并重用已计算过的值是很有用的。对于上例，可以把第二条赋值语句替换成“Car = R3 - Run;”。如果使用了诸如乘法器等更大的功能块，找出公共子表达式就变得尤为重要了。

4.3 代 码 移 位

可能会遇到 for 循环语句内某个表达式的值在每次循环中都不变化的情形。而综合工具通常按照指定的循环次数来展开 for 循环语句。这样，那些值不随着循环变量而改变的表达式在展开时就会造成代码冗余。同样，逻辑优化器也许能或不能聪明地优化这样的逻辑。如果在更高层次上进行优化，即在模型中进行优化，会有助于优化器优化代码中更为关键的部分。请看以下 for 循环示例：

```
Car = …
…
for (Count = 1; Count <= 5; Count = Count + 1)
begin
  …
  Tip = Car - 6;
  // 假定循环中未对 Car 赋新值
  …
end
```

此赋值语句右端表达式的值不随循环变量而变，即计算出的变量 *Tip* 的值独立于循环变量 *Count*。但是，综合工具会产生 5 个减法器，每循环一次就产生一个，这样就产生了额外的逻辑。而对于本例，只需要一个减法器就行了。

解决这种情况的最好办法就是把循环语句中那些不随循环过程而改变其值的表达式移至循环之外，而且这样能提高仿真效率。下例正是如此：

```
Car = …
…
Temp = Car - 6;          // 引入一个临时变量
for (Count = 1; Count <= 5; Count = Count + 1)
begin
  …
  Tip = Temp;
  // 假定循环中未对 Car 赋新值
  …
end
```

为产生更高效的代码，设计者应该进行这样的代码移位；这也让逻辑优化器能够从更好的起点开始优化。

4.4　公因子提取

公因子提取是从 if 语句或 case 语句的互斥分支中提取的公共子表达式。请看下例：

```
if (Test)
  Ax <= A & (B + C);
else
  By = (B + C) | T;
```

此 if 语句的互斥分支中都计算了表达式 $B+C$。因此，将该表达式作为因子提取出来放

在 if 语句之前是很有用的，这样综合工具就不会产生两个加法器了。如下所示：

```
Temp = B + C;          // 引入一个临时变量

if (Test)
  Ax = A & Temp;
else
  By = Temp | T;
```

提取公因子后就会综合出更少的逻辑(上例中只综合出一个加法器)，这样，逻辑优化器就能集中精力优化更为关键的部分。

4.5 交换律和结合律

某些特定情形下，也许有必要在实施上述各项优化手段之前进行交换运算。下例显示了在找出公共子表达式之前先进行交换运算有助于识别公共子表达式：

```
Run = R1 + R2;
...
Car = R3 - (R2 + R1);
```

对表达式“$R2+R1$”应用交换律，有助于识别在第一条赋值语句中也用到的公共子表达式“$R1+R2$”。

类似地，在实施上述任何优化手段之前也可以先应用结合律。请看下例：

```
Lam = A + B + C;
...
Bam = C + A - B;
```

注意，对第一条语句中的表达式应用交换律和结合律，就能识别“$C+A$”这个公共子表达式。找出该子表达式之后，就可以写成：

```
Temp = C + A;          // 引入一个临时变量
Lam = Temp + B;
Bam = Temp - B;
```

如果未应用交换律和结合律，综合工具就会生成 3 个加法器和一个减法器；在找出子表达式之后，就只生成两个加法器和一个减法器了，这进一步节省了逻辑。

4.6　其他优化手段

通常，综合工具还能够实现另外两种优化，这就是：

a）死代码消除

b）常量合并

这些优化通常是由综合系统完成的，设计者不必为此操心。尽管如此，下面仍将对它们进行解释。

死代码消除是指删除那些永远不会被执行的代码。例如：

```
if (2 > 4)
  Oly = Sdy & Rdy;
```

显然，没有必要将此综合成“与”门，因为该赋值语句永远不会被执行，它是死代码。

常量合并是指在编译时计算出常量表达式的值，而不是先实现这些逻辑然后用逻辑优化器将其消除。请看以下简单示例：

```
parameter FAC = 4;
...
Yak = 2 * FAC;
```

常量合并会在编译时计算出右端表达式的值并将其赋给 *Yak*，而不必为此表达式生成任何硬件，这样就节省了逻辑优化的时间。

4.7　触发器和锁存器的优化

4.7.1　消除触发器

理解综合工具的触发器推导规则很重要。不同综合工具的推导规则不尽相同。如果未遵循这些推导规则，综合出的网表中就可能含有远远多于实际需要的触发器。下例正是如此：

```
reg PresentState;
reg [0:3] Zout;
wire ClockA;
...
```

```
always @ (posedge ClockA)
  case (PresentState)
    0 :
      begin
        PresentState <= 1;
        Zout <= 4'b0100;
      end
    1 :
      begin
        PresentState <= 0;
        Zout <= 4'b0001;
      end
endcase
```

看上去此例的本意是要把 *PresentState* 的值保存在上升沿触发的触发器中。综合后，不仅为 *PresentState* 生成了一个触发器，还为 *Zout* 生成了 4 个触发器。这是因为对 *Zout* 的赋值受时钟的控制。为 *Zout* 生成这些触发器或许不是其本意。若不是，则须在一个单独的 always 语句中用 case 语句为 *Zout* 赋值，这样赋值就不受时钟的控制。如下所示，修改后的示例只生成了一个触发器。

```
always @ (posedge ClockA)      // 会推导出触发器
  case (PresentState)
    0 : PresentState <= 1;
    1 : PresentState <= 0;
  endcase

always @ (PresentState)      // 组合逻辑
  case (PresentState)
    0 : Zout = 4'b0100;
    1 : Zout = 4'b0001;
  endcase
```

4.7.2 消除锁存器

如果变量未在 case 语句或 if 语句的所有分支中都被赋值，那么可能导致产生锁存器。这是因为在 Verilog HDL 中，寄存器变量在 always 语句中被赋值就会被推导成存储器，也就是说若变量未在条件语句的所有分支中都被赋值，则需要用存储器来保存其值。

请看下例：

```
reg Luck;

always @ (Probe or Count)
  if (Probe)
    Luck = count;
```

当 *Probe* 为 0 时，*Luck* 为何值？*Luck* 必是其原值。这样就需要保存 *Luck* 的值，因此需要为该变量生成一个锁存器。

消除锁存器的最好方法是首先确定综合工具推导出的锁存器数目。然后设计者需要核查是否推导出的每个锁存器都确有必要成为锁存器。可能会出现这样的情况：设计者从未打算用锁存器，或者忘了指定所有条件值。最好是先核查综合出的锁存器，再确定为何综合出这些锁存器，如果需要消除任何不必要的锁存器，则修正代码。

有两种方法可用来消除上例中的锁存器。第一种方法是在 else 分支中也对该变量赋值：

```
always @ (Probe or Count)
  if (Probe)
    Luck = Count;
  else          // 添加了 else 子句
    Luck = 0;
```

第二种方法是在 if 语句之前先对变量进行初始化赋值。

```
always @ (Probe or Count)
begin
  Luck = 0;  // 明确地对变量进行初始化赋值
  if (Probe)
    Luck = Count;
end
```

4.8 设 计 规 模

小型设计综合起来更快

实验研究表明逻辑电路规模在 2000 至 5000 门时逻辑优化器的优化效果最佳。这意味着 Verilog HDL 模型中的 always 语句不能过长。对设计的描述应组织成多个 always

语句或者多个模块。

综合出的电路门数与 Verilog HDL 代码行数无关。2500 门的电路可能是由一段 10 行的 Verilog HDL 代码(可能包含一个 for 循环和/或若干向量)综合出来的,也可能是由一段 10 000 行的 Verilog HDL 代码(可能含有大的 case 语句,而其中的赋值都很简单)综合出来的。

综合过程的运行时间主要用于逻辑优化,它与设计规模呈指数关系。因此将各个子功能块的规模保持在可处理的设计范围内很关键。

层次

根据 always 语句来保持 Verilog HDL 模型中的层次关系是很有用的。这使得综合工具能产生各子电路的层次关系,以便逻辑优化器能有效地对此进行处理。

综合工具经常会自动保存大的数据通路算符的层次关系。例如:

```
reg [15:0] Zim, Rim, Sim;
...
Zim = Rim + Sim;
...
```

此时综合工具会把此 16 位加法器作为一个单独的层次来处理。

宏结构

综合不是用来建立 ROM 或 RAM 之类的存储器的正确机制。通常在工艺库中就可以获得预定义的 RAM。若需要 RAM 之类的模块,最好像对待元件那样,在模型中对其进行实例化,然后再综合此实例模型。综合工具只是为 RAM 建立一个黑盒,稍后设计者再把 RAM 模块连接至该黑盒中。

如果设计者遇到以下形式的语句,并期望综合工具实现一个高效的乘法器,则有必要采取类似措施。

```
Cyr = Rby * Ytr;        // 16 位自变量
```

设计者可能已设计出更好的乘法器。此时,设计者最好像实例化元件那样实例化乘法器,而不是采用乘法算符,因为乘法算符通过综合未必能得到高效的乘法器。

4.9 使 用 括 号

编写 Verilog HDL 代码时,必须对所产生的逻辑结构做到心中有数。很重要的一点就是使用括号。请看以下示例:

```
Result = Rhi + Rlo - PhyData + MacReset;
```

综合工具在综合右端表达式时遵循 Verilog HDL 的表达式演算规则，即从左至右进行演算，于是构造出如图 4-4 所示的电路。这样产生的逻辑结构的关键路径较长。好一点的做法是使用括号，如：

```
Result = (Rhi + Rlo) - (PhyData - MacReset);
```

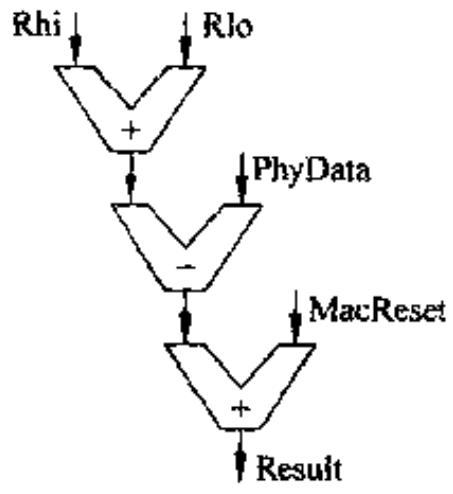

图 4-4　未使用括号综合出的电路

这样综合出的电路关键路径较短，如图 4-5 所示。使用括号也有助于找出公共子表达式。

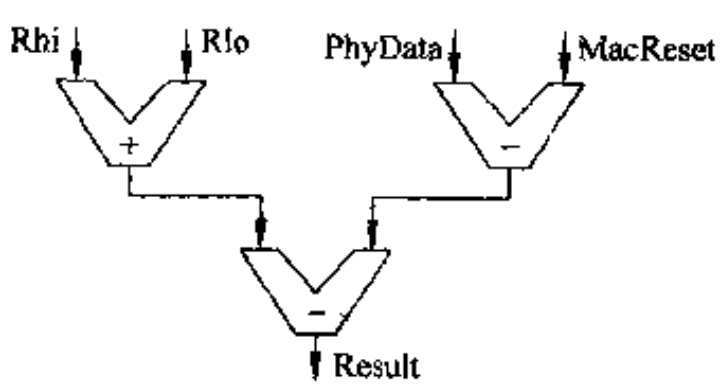

图 4-5　使用括号后综合出的电路

建议：在表达式中大量使用括号，以控制所综合出的逻辑电路的结构。

第5章　验　　证

Verilog HDL模型综合成网表后，通过验证来确保综合出的网表的功能与设计初衷一致是很重要的。此步骤之所以重要，是因为综合系统可能对 Verilog HDL 代码作了某些与设计初衷不一致的假设或解释。

在本章中，假定通过使用仿真对设计模型与综合出的网表进行功能验证来执行此步骤。将举例说明设计模型与综合出的网表之间可能存在功能不一致的，并阐明其成因，进而提出一些避免功能不一致的建议。

本章还假定综合过程产生 Verilog HDL 网表，如图 5-1 所示。Verilog HDL 网表是用网线相互连接在一起的模块实例的集合。

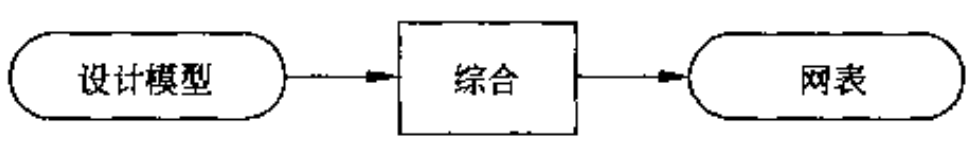

图 5-1　通过综合得到的网表

5.1 测 试 平 台

进行功能验证的一种方法是，将在设计模型的仿真过程中使用的那组激励拿来对网表进行仿真，将仿真结果保存在结果文件中，然后比较两者的仿真结果是否完全相同。此过程如图 5-2 所示。

另一种方法是编写测试平台。测试平台是用 Verilog HDL 编写的模型，能施加各种激励、比较输出响应、报告任何功能不一致之处。此过程如图 5-3 所示。请看以下全加器的测试平台。从向量文件“Inputs. vec”读入激励信号，其内容如下：

```
100
000
101
011
111
```

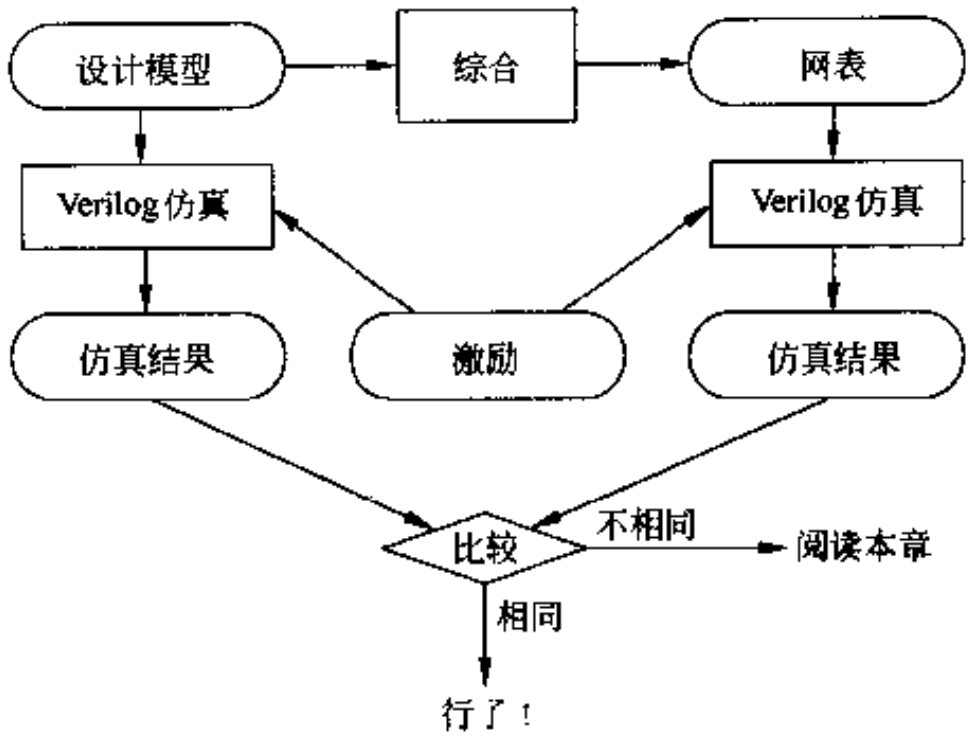

图 5-2　通过仿真进行验证

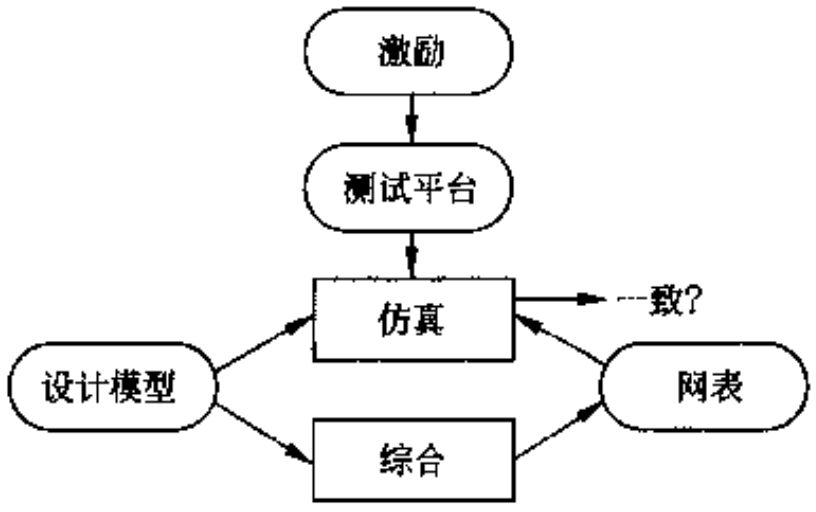

图 5-3　使用通用测试平台

```
module TestBenchFA;
  parameter WORDS = 5;
  reg [1:3] MemV [1:WORDS];
  reg A, B, Cin;
  wire SumBeh, CoutBeh, SumStr, CoutStr;
  integer J;

  // 实例化用于测试的设计模型
  FA_RTL F1 (A, B, Cin, SumBeh, CoutBeh);

  // 实例化综合出的网表模块
  FA_Netlist F2 (A, B, Cin, SumStr, CoutStr);

  initial
```

```
    begin
      // 读取文件中的输入向量:
      $readmemb ("Inputs. vec", MemV);

      // 将每个向量施加到设计模型和综合出的网表模块上
      for (J = 1; J <= WORDS; J = J + 1)
      begin
        {A, B, Cin} = MemV [J];
        #5;  // 等待 5 个时间单位,届时电路状态会稳定下来

        // 如果输出值不一致:
        if ((SumBeh ! == SumStr)||(CoutBeh ! == CoutStr))
          $display (" **** Mismatch on vector %b *****",
                    MemV [J]);
        else
          $display ("No mismatch on vector %b", MemV [J]);
      end
    end
endmodule
```

此测试平台将所出现的功能不一致错误全部打印出来。

后续各节中,将会看到由于综合时采用了不同的解释(相对于 Verilog HDL 语义而言)而导致功能不一致的一些例子。

5.2 赋值语句中的延迟

设计模型中指定的延迟可能会导致设计模型与综合出的网表功能不一致。请看以下在持续赋值语句中给出了延迟的加法器示例及其综合出的网表。

```
module Adder (A, B, C);
  input [0:3] A, B;
  output [0:3] C;

  assign #5 C = A + B;
endmodule

// 综合出的网表如下:
module AdderNetList (A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3);
  input A0, A1, A2, A3, B0, B1, B2, B3;
```

```
output C0, C1, C2, C3;

OAI21    C0_1       (C2_1, S248, S310, C0);
ND2      S310_1     (S248, C2_1, S310);
XOR2     S248_1     (A0, B0, S248);
AOI22    C2_2       (S241, S244, A1, B1, C2_1);
OAI22    C1_1       (S295, S244, S299, S241, C1);
INRB     S299_1     (S244, S299);
OAI22    S244_1     (B1, S242, A1, S243, S244);
INRB     S243_1     (B1, S243);
INRB     S242_1     (A1, S242);
INRB     S295_1     (S241, S295);
OAI22    S241_1     (S291, S237, S238, S239, S241);
INRB     S291_1     (S240, S291);
OAI21    C2_1       (S237, S240, S334, C2);
ND2      S334_1     (S237, S240, S334);
OAI22    S240_1     (B2, S238, A2, S239, S240);
INRB     S239_1     (B2, S239);
INRB     S238_1     (A2, S238);
ND2      S237_1     (A3, B3 S237);
OAI22    C3_1       (B3, S235, A3, S236, C3);
INRB     S236_1     (B3, S236);
  INRB     S235_1     (A3, S235);
endmodule
```

如果施加激励文件中的测试向量，比如每施加一次 1 ns，则因为网表中的所有模块实例都是无延迟的行为模型，延迟上的差异会导致设计模型与网表的仿真结果之间存在相位偏差。此时正确的做法是：

a）去除设计模型中的所有延迟；

b）以大于 5 ns 的周期施加测试激励，这种方法会更好些。

如果库模块的模型中存在延迟，那么在确定激励周期时必须将这些延迟考虑在内。

建议：为消除因设计模型中的延迟导致的功能不一致，必须计算出模型中的最大延迟。激励施加时间必须大于这个最大延迟。

5.3 悬空的端口

综合出的网表中可能会出现某些模块实例有悬空的输入端口的情况。下例正是如此。

```
module AOI22 (A, B, D, Z);
  input A, B, D;
```

```
  output Z;
  reg T1, T2, C;

  always @ (A or B or D)
  begin
    T1 = A & B;
    T2 = C & D;        // 不再对 C 赋值
    Z = ! (T1|T2);
  end
endmodule

// 综合出的网表如下:
module AOI22_ NetList (A, B, D, Z);
  input A, B, D;
  output Z;

  AND2    S0_1      (A, B, T1_1);
  AND2    S1_1      ( , D, T2_1);      // 第一个端口悬空
  OR2     S2_1      (T1_1, T2_1, T2_0);
  INRB    S3_1      (T2_0, Z);
endmodule
// 注意:此时尚未使用逻辑优化器,如果使用可能还能消除一些冗余门
```

注意,在此综合出的网表中,*AND2* 的模块实例 *S1_1* 的第一个输入端口处于开路状态。在模块 *AOI22_NetList* 的仿真过程中,开路输入取值 z[①],而模块 *AOI22*[②] 中未赋值的 *C* 取默认值 x。事实上,变量 *C* 在设计模型仿真和综合出的网表的仿真中分别取了不同的值,正是由于该变量在这两个领域中有不同的默认值,因而可能造成功能不一致。

建议:好的综合系统会对赋值前就被引用的做法(如模块 *AOI22* 中的变量 *C*)发出警告信息,请注意这些警告。

5.4 遗失的锁存器

第 2 章介绍了推导锁存器的各种规则。还介绍了那些规则的一种例外情形,即用作临时量的变量不会被推导成锁存器。然而,还有另外一些情形也不会将变量推导成锁存

① 在 Verilog HDL 中,未赋值的寄存器类型变量的默认值是 x,而网线类型变量的默认值是 z。

② 在附录 B 中描述了综合出的网表中所使用的各个逻辑门的行为。

器，尽管从代码序列中看起来理应推导出锁存器。

先考虑第一种情形：

```
wire Control, Jrequest;
reg DebugX;
...
always @ (Control or Jrequest)
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = DebugX;
```

此 always 语句中，变量 *DebugX* 在 if 语句的所有分支中都被赋值了。但是，数据流分析表明无须保存 *DebugX* 的值(因为当 *Control* 为假时 *DebugX* 在赋值前就被引用了)。此时，综合系统会产生警告信息，提示变量 *DebugX* 在赋值前被引用过，还提示这可能会导致设计模型与综合出的网表功能不一致。

再强调一次，推导出锁存器的规则是：

a) 变量在条件语句(if 或 case 语句)中被赋值。

b) 变量未在条件语句的所有分支中都被赋值。

c) 在 always 语句的多次调用之间需要保存变量值。

必须同时满足以上 3 个条件，才会将变量推导成锁存器。上例中 always 语句内的变量 *DebugX* 违反了规则 b)，因此不会将其推导成锁存器。

请再看一个示例：

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;

  Bdy = DebugX;
end
```

此 always 语句中，看起来 *DebugX* 和 *Bdy* 都应该推导成锁存器。但 *DebugX* 因违反了规则 b)而不会被推导成锁存器，而 *Bdy* 因违反了规则 a)也不会被推导成锁存器。虽然 HDL 语义表明需要保存 *Bdy* 的值，但综合系统在这种情况不会产生锁存器，相反它会发出 *Bdy* 在赋值前已被引用过的警告，并提示可能会发生功能不一致。

以下 always 语句中，*DebugX* 因违反规则 b)而不会产生锁存器，不过为 *Bdy* 产生了

一个锁存器。

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;
  if (Jrequest)
    Bdy = DebugX;
end
```

以下 always 语句又会是何种情形?

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;

  if (Jrequest)
    Bdy = DebugX;
  else
    Bdy = 'b1;
end
```

其实,变量 *DebugX* 和 *Bdy* 都不会推导出锁存器,尽管语义表明需要保存变量 *Bdy* 的值,但是综合系统不会产生锁存器,只是发出关于该变量在赋值前就已被引用过以及可能会存在功能不一致的警告。

5.5 再 谈 延 迟

综合系统通常忽略模型中的延迟,实际上忽略延迟很容易导致综合出的网表的仿真结果与设计模型的仿真结果不同。下例正是如此:

```
LX = #3 'b1;

if(CondA)
  LX = #5 'b0;
  ...
```

模型仿真表明，*LX* 在延迟 3 ns 后值为 1，若条件 *CondA* 为真则再延迟 5 ns 后其值又变为 0。但是，由于综合系统忽略了延迟，则当 *CondA* 为真时，在网线上起到的效果就像是将 0 值赋给了 *LX*，于是综合出与此相应的硬件来体现这种效果。注意，如果对综合出的网表进行仿真，当 *CondA* 为真时 *LX* 的值不会变成 1。

建议：避免在用于综合的设计模型中插入延迟。若实在有必要，则应在一处集中指定变量所有延迟的累计值，而不要将该变量的延迟分散在若干条语句中。

5.6 事 件 表

综合系统在综合时常常会忽略某个 always 语句的事件表。如果建模时对此未加以注意，则会导致功能不一致。请看以下简单示例：

```
always @ (Read)
  Grt = Read & Clock;
// 综合出的网表如图 5-4 所示
```

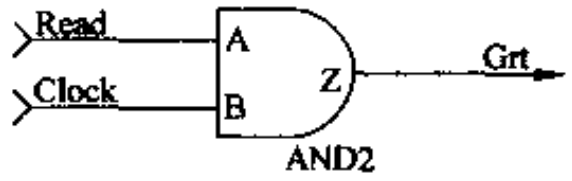

图 5-4 网表对 *Read* 和 *Clock* 都敏感

如图 5-4 所示，综合出的网表对 *Read* 和 *Clock* 两者的变化都加以判断，然而此 always 语句仅在 *Read* 变化时才得以执行。

再看一个由于 always 语句的事件表不完整而引起功能不一致的示例：

```
reg Rst;
reg [3:0] Pbus, Treg;
...
always @ (Rst)
  if (Rst)
    Treg = 0;
  else
    Treg = Pbus;
```

此 always 语句的事件表中没有变量 *Pbus*。但是综合出的网表在 if 条件为假时，*Pbus* 上的任何变化都会波及 *Treg*。这与设计模型的语义不一致，从而出现功能不一致。

建议：对于无时钟事件的 always 语句(即对组合逻辑建模)，其事件表中应包括该

always 语句引用的所有变量。

5.7 综 合 指 令

到目前为止已见过两种综合指令：full_case 和 parallel_case。这两种指令可能会引起设计模型与综合出的网表之间出现功能不一致。问题在于只有综合工具才认识这些指令，而仿真工具并不认识。对于这两种情况中的任何一种，设计者在指定指令时若不加注意，都会出现功能不一致。

请看以下 full_case 综合指令示例：

```
reg [1:0] CurrentState, NextState;
...
case (CurrentState)                    // synthesis full_case
    2'b01 : NextState = 2'b10;
    2'b10 : NextState = 2'b01;
endcase
```

此 full_case 指令告诉综合工具已列出 *CurrentState* 所有可能的取值，而其他未列出的取值都是无关值，因此综合工具不应为 *NextState* 产生锁存器。但是仿真结果并非如此。可能由于某种原因，*CurrentState* 的值是 2'b00。在这种情况下，case 语句仿真时看起来是将 *NextState* 的值保存了，但是综合出的网表中并没有保存其值。

请看以下 parallel_case 综合指令示例：

```
case (1'b1)                    // synthesis parallel_case
  Gate1 : Mask1 = 1;
  Gate2 : Mask2 = 1;
  Gate3 : Mask3 = 1;
endcase
```

此 case 语句仿真语义(parallel_case 指令以注释形式出现而被忽略)表明：若 *Gate*1 为 1，则把 1 赋给 *Mask*1；若 *Gate*2 为 1，则把 1 赋给 *Mask*2；若 *Gate*3 为 1，则把 1 赋给 *Mask*3。但是，用了 parallel_case 指令，综合出的就是并行译码器而不是优先级 if 结构了。这就出现了功能不一致。假定 *Gate*3 和 *Gate*1 同时为 1，结果又会如何？在 case 语句中，会执行第一个分支，而在综合出的网表中，分支 1 和分支 3 都有效。用 if 语句表示此 case 语句的语义，如下所示：

```
if (Gate1)
```

```
    Mask1 = 1;
  else if (Gate2)
    Mask2 = 1;
  else if (Gate3)
    Mask3 = 1;
```

综合出的网表的语义如下：

```
if (Gate1)
  Mask1 = 1;

if (Gate2)
  Mask2 = 1;

if (Gate3)
  Mask3 = 1;
```

建议：慎用综合指令 full_case 和 parallel_case，只在真正必要时才使用。

5.8 变量的异步预置位

综合带异步预置位和清零的触发器时，建议在异步条件下仅赋常量值。若异步读取变量，就可能出现功能不一致。请看下例：

```
module VarPreset (ClkZ, PreLoad, LoadData, PrintBus,QuickBus);
  input ClkZ, PreLoad;
  input [1:0] LoadData, PrintBus;
  output [1:0] QuickBus;
  reg [1:0] QuickBus;

  always @ (negedge PreLoad or posedge ClkZ)
    if (! PreLoad)
      QuickBus <= LoadData;  // 异步数据赋值
    else
      QuickBus <= PrintBus;
endmodule
// 综合出的网表如图 5-5 所示
```

此例将变量 *QuickBus* 综合成两个带异步预置位和清零的触发器。变量 *LoadData* 通过其他逻辑连接到这两个触发器的预置位和清零输入端。当 *PreLoad* 有效(即为 0)且

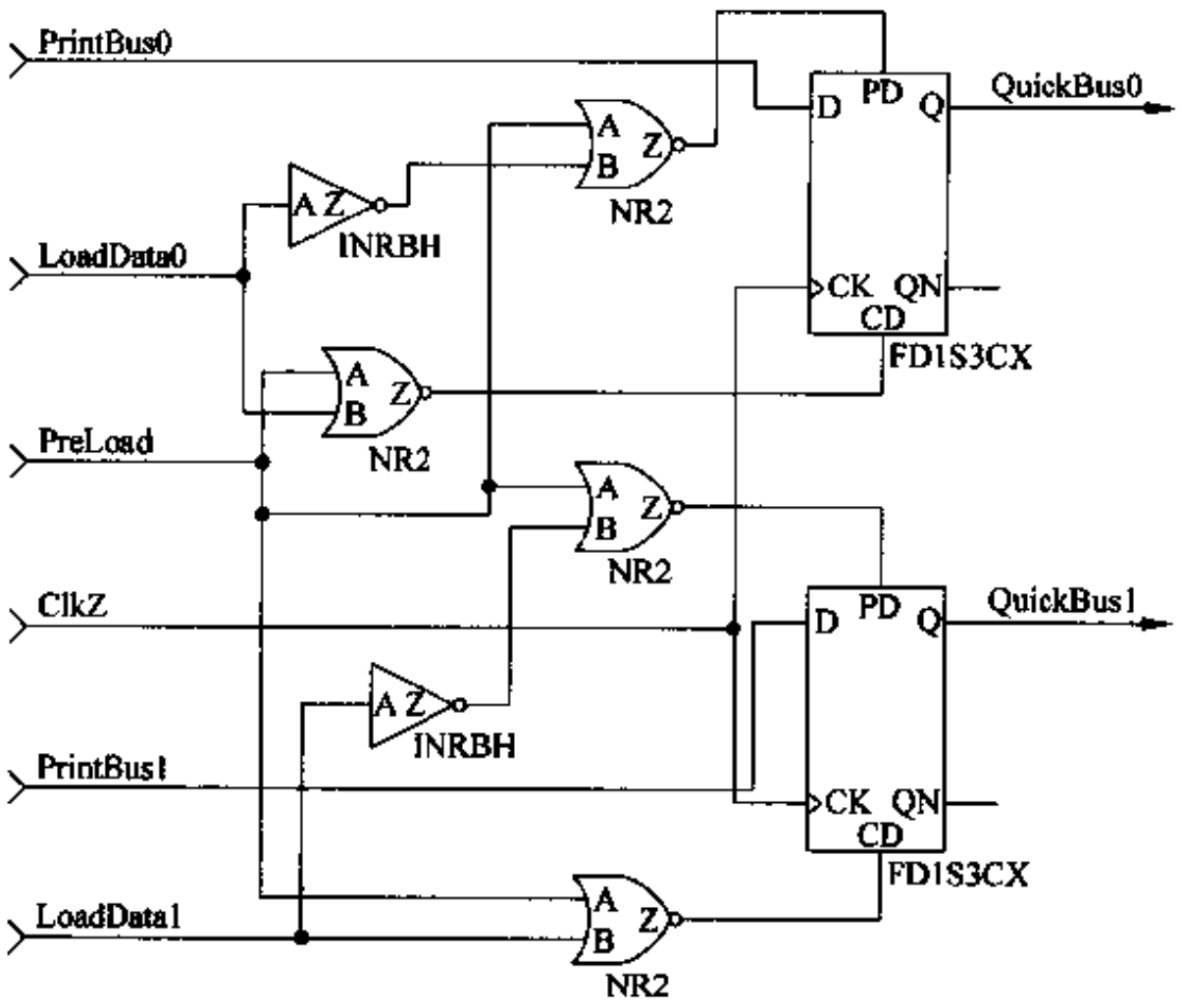

图 5-5 变量的异步预置位

LoadData 的值变化时，异步数据的变化立即影响这两个触发器的输出值。但是在设计模型中，*LoadData* 的任何变化都不会影响 *QuickBus* 的输出值。这样就出现了功能不一致。

建议：避免异步读取变量并将其值赋给触发器，否则应确保异步条件有效时异步数据输入不会发生任何变化。

5.9 阻塞式和非阻塞式赋值

在第 2 章中曾建议：

a) 阻塞式赋值用于组合逻辑建模。

b) 非阻塞式赋值用于时序逻辑建模；当对变量的赋值和引用都在同一条 always 语句中时，也可以采用阻塞式赋值。

本节将解释为何遵循上述建议很重要，否则就会面临功能不一致的风险。

5.9.1 组合逻辑

阻塞式赋值反映了组合电路中的数据流。考虑以下 always 语句：

```
reg TM, TN, TO, TZ;

always @ (A or B or C or D or E)
begin
  TM = A & B;
  TN = C & D;
  TO = TM|TN|E;
  TZ = ! TO;
end
```

此例中的所有赋值都是阻塞式赋值。顺序块中各语句的意思是首先计算 *TM* 的值，接着执行第二条语句对 *TN* 赋值，然后执行第三条语句对 *TO* 赋值，依次类推。这就实现了组合逻辑数据流的建模。

现在把这些阻塞式赋值改为非阻塞式赋值：

```
reg TM, TN, TO, TZ;

always @ (A or B or C or D or E)
begin
  TM <= A & B;
  TN <= C & D;
  TO <= TM|TN|E;
  TZ <= ! TO;
end
```

执行第一条赋值语句时，*TM* 的值并没有立即更新，而是被安排至当前仿真周期结束时再赋新值。由于所有语句的执行是在零时间内顺序发生的，因此当执行第三条语句时，计算 *TO* 的值使用的是 *TM* 的原值(此时 *TM* 还未得到其新值)。结果是，输出 TZ 未反映出此逻辑的“与或非”行为。问题在于 *TM*、*TN* 和 *TO* 的值都是在当前仿真周期结束时才被更新，它们更新后的值未用来重新计算逻辑值。

解决此问题的方法是将变量 *TM*、*TN* 和 *TO* 都放入 always 语句的事件表中，如下所示：

```
reg TM, TN, TO, TZ;
```

```
always @ (A or B or C or D or E or TM or TN or TO)
begin
  TM <= A & B;
  TN <= C & D;
  TO <= TM|TN|E;
  TZ <= ! TO;
end
```

这样，每当 *TM*、*TN* 或 *TO* 发生变化时，always 语句就会被重新计算，并最终得到正确的 *TZ* 值。由此可发现以下两个问题：

a) 非阻塞式赋值不反映逻辑流

b) 需要将所有赋值对象都列入事件表中

因此推荐对组合逻辑建模采用阻塞式赋值，这样就能很容易地避免以上两个问题。

5.9.2 时序逻辑

首先考虑对时序逻辑建模时，若禁止使用阻塞式赋值，结果会如何？考虑以下这两条 always 语句：

```
always @ (posedge ClkA)      // Label AwA
  … = DataOut;               // 读 DataOut 的值

always @ (posedge ClkA)      // Label AwB
  DataOut = …;               // 采用阻塞式赋值
```

标号为 *AwB* 的 always 语句以阻塞方式向 *DataOut* 赋值，而标号为 *AwA* 的 always 语句读取其值。如果按照程序中的书写顺序模拟这些 always 语句(事实上仿真器根据事件表中的事件变化来排定 always 语句的执行顺序)，在 *ClkA* 上升沿处，always 语句 *AwA* 读取了 *DataOut* 的当前值，然后 always 语句 *AwB* 再向 *DataOut* 赋新值。如果颠倒这两条 always 语句的顺序(或仿真器选择重新排定这两条 always 语句的执行顺序)，那么先执行 always 语句 *AwB*，导致零时间内将新值赋给 *DataOut*，随后 always 语句 *AwA* 读取的便是更新后的 *DataOut* 值。这看起来是由于 always 语句的执行顺序不同而得到不同的结果，但问题实际上是由于当这两条 always 语句都可以执行时，向 *DataOut* 的赋值是在零时间内发生并完成的。因此根据先执行哪一条 always 语句，*AwA* 中读取的 *DataOut* 值可能是其原值，也可能是其新值。

为了消除这种仿真行为的依赖性，最好在稍后对其进行强制赋值，应保证那时所有的变量读取都已完成。使用非阻塞式赋值可以做到这一点。在这种情况下，读取 *DataOut*

发生在当前时刻，而在当前仿真周期结束时(即所有的变量读取都已完成)才将新值赋给 *DataOut*。这使得模型的行为不再受 always 语句执行顺序的影响。请看以下采用非阻塞式赋值的 always 语句：

```
always @ (posedge ClkA)      // Label AwA
  … = DataOut;               // 读 DataOut 的值

always @ (posedge ClkA)      // Label AwB
  DataOut <= …;              // 采用非阻塞式赋值
```

由此可知：如果在某条 always 语句内对变量赋值而在该 always 语句外读取变量，那么赋值语句应是非阻塞式过程赋值语句。

如果对变量的赋值和读取都出现在同一个时钟控制的 always 语句内，又该如何处理呢？此时可以采用阻塞式赋值：

```
reg [9:0] Total;

always @ (negedge ClkB)
begin
  Total = LoadValue + 2;     // 阻塞式赋值
  if (Total == 21)           // 引用上一条语句中所赋的值
    NumBus <= ControlBus;
  else
    NumBus <= DataBus;
end
```

其中，变量 *Total* 在同一条 always 语句中先被赋值然后又被引用。此时，最好是在判断 if 条件之前就完成对 *Total* 的赋值。*Total* 是被先赋值后引用的临时变量，因此对其采用阻塞式赋值很合适。

如果采用了非阻塞式赋值，如下所示：

```
reg [9:0] Total;

always @ (negedge ClkB)
begin
  Total <= LoadValue + 2;    // 非阻塞式赋值

  if (Total == 21)           // 读取的是 Total 的原值
    NumBus <= ControlBus;
```

```
    else
      NumBus <= DataBus;
  end
```

在判断 if 条件时，*Total* 的值仍是其原值，而不是在上一条赋值语句中欲对其赋的新值。

因此，如果变量在实现赋值的 always 语句之外被引用，则建议对这样的变量采用非阻塞式赋值。此外，对于仅在一条 always 语句内被赋值和引用的变量，则建议采用阻塞式赋值。

附录 A　可综合的语言结构

本附录给出了 ArchSyn 综合系统 14.0 版本所能识别的可综合的 Verilog HDL 结构清单，以便读者获知哪些 Verilog HDL 结构是可综合的。并非所有综合工具的可综合子集都与该子集相同。

仅与仿真相关而与综合无关的那些结构被标记成“忽略的结构”；不可综合的结构被标记成“不支持”。语言结构可分类如下：

a）支持：能够综合成硬件的结构。

b）不支持：输入文件中一旦出现这样的结构就会中止综合过程。

c）忽略：综合时会发出警告信息，但声明语句例外。

以下列表中，第一列是 Verilog HDL 特性，第二列表明该特性是否支持综合，第三列给出了相关的注释和例外情况。

词 法 常 规

算符	支持	不支持全等算符和非全等算符
空白字符和注释	支持	
数	支持	
字符串	不支持	
标识符、关键字及系统名	支持	忽略系统名
文本替换	支持	

数 据 类 型

数值集合	支持	
寄存器和网线	支持	
向量	支持	
信号强度	忽略	
隐式声明	支持	
网线初始化	不支持	初始时连线处于悬空状态
网线类型	支持	
存储器	支持	
整数	支持	
时间	不支持	
实数	不支持	
参数	支持	

表 达 式

算符	支持	不支持全等算符和非全等算符
运算量		
网线和寄存器位寻址	支持	
存储器寻址	支持	
字符串	不支持	
最小/典型/最大延迟表达式	忽略	
表达式位宽	支持	

赋 值

持续赋值	支持	忽略其中的延迟和驱动强度值
过程赋值	支持	

逻辑门级和开关级建模

逻辑门和开关声明语法	支持	不支持其中的强度和延迟
逻辑门 AND、NAND、NOR、OR、XOR 和 XNOR	支持	
逻辑门 BUF	支持	
逻辑门 NOT	支持	
逻辑门 BUFIF1，BUFIF0，NOTIF1 和 NOTIF0	支持	
MOS 开关	不支持	
双向传输开关	不支持	
CMOS 门	不支持	
电源 PULLUP 和 PULLDOWN	不支持	
隐式网线声明	支持	
逻辑强度建模	不支持	
组合信号的强度和值	不支持	
助记格式	不支持	
非阻性器件造起的强度衰减	不支持	
阻性器件造起的强度衰减	不支持	
网线类型的信号强度	忽略	
逻辑门和网线延迟	忽略	
撤销逻辑门和网线名①	不支持	

用户定义的基本元件

	不支持	

① 译者注：此用法已作废。

行 为 建 模

过程赋值	支持	不支持时间声明 忽略用作延迟的定时控制 不支持用作事件的定时控制
条件语句	支持	
case 语句	支持	
循环语句		
forever 循环	不支持	
repeat 循环	支持	repeat 表达式必须是常量
while 循环	不支持	
for 循环	支持	对 for 循环变量的赋值必须是常量赋值
过程定时控制		忽略延迟定时控制,不支持事件定时控制
语句块语句	支持	不支持时间声明和事件声明
结构化过程		
初始化语句	忽略	
always 语句	支持	
任务	支持	不支持时间声明和事件声明
函数	支持	不支持时间声明和事件声明

任务和函数

任务和任务启用	支持	不支持时间声明和事件声明
函数和函数调用	支持	不支持时间声明和事件声明

命名块和任务的停用

	不支持	

过程持续赋值

	不支持	

层 次 结 构

模块	支持	
顶层模块	支持	
模块实例化	支持	
重载模块参数值	支持	不支持 DEFPARAM
宏模块	支持	
端口	支持	
层次名	不支持	
自动命名	支持	不支持系统产生的名字
作用域规则	支持	

specify 语句块

	不支持	

附录B 通 用 库

本附录介绍在本书中综合出的网表所使用的各种元件，以注释的方式对每个元件的功能作了说明。

```
module AND2 (A, B, Z);
  input A, B;
  output Z;
  // Z = A&B;
endmodule

module AOI21 (A1, A2, B, Z);
  input A1, A2, B;
  output Z;
  // Z = !((A1&A2) | B);
endmodule

module AOI211 (A1, A2, B1, B2, Z);
  input A1, A2, B1, B2;
  output Z;
  // Z = !((A1&A2) | B1 | B2);
endmodule

module AOI22 (A1, A2, B1, B2, Z);
  input A1, A2, B1, B2;
  output Z;
  // Z = !((A1&A2) | (B1&B2));
endmodule

module BN20T20D (A, ST, STN, PADI, Z, PADO);
  input A, ST, STN, PADI;
  output Z, PADO;
  // 双向缓冲器
  // Z = PADI;
  // PADI = 0 when (! A&&! STN) else
```

```
  //          1 when (A&&ST) else
  //          'bz;
endmodule

module BUF (A, Z);
  input A;
  output Z;
  // Z = A;
endmodule

module FD1P3AX (D, SP, CK, Q, QN);
  input D, SP, CK;
  output Q, QN;
  // 上升沿触发、高电平采样的静态D型触发器
endmodule

module FD1S1A (D, CK, Q, QN);
  input D, CK;
  output Q, QN;
  // 高电平敏感的静态D型触发器(锁存器)
endmodule

module FD1S1B (D, CK, PD, Q, QN);
  input D, CK, PD;
  output Q, QN;
  // 高电平敏感、高电平异步预置位的静态D型触发器(锁存器)
endmodule

module FD1S1D (D, CK, CD, Q, QN);
  input D, CK, CD;
  output Q, QN;
  // 高电平敏感、高电平异步清零的静态D型触发器(锁存器)
endmodule

module FD1S1E (D, CK, CDN, Q, QN);
  input D, CK, CDN;
  output Q, QN;
  // 高电平敏感、低电平异步清零的静态D型触发器(锁存器)
endmodule
```

```
module FD1S1F (D, CK, PD, CDN, Q, QN);
  input D, CK, PD, CDN;
  output Q, QN;
  // 高电平敏感、低电平异步清零、高电平异步预置位的静态 D 型触发器(锁存器)
endmodule

module FD1S2AX (D, CK, Q, QN);
  input D, CK;
  output Q, QN;
  // 下降沿触发的静态 D 型触发器
endmodule

module FD1S2BX (D, CK, PD, Q, QN);
  input D, CK, PD;
  output Q, QN;
  // 下降沿触发、高电平异步预置位的静态 D 型触发器
endmodule

module FD1S2CX (D, CK, PD, Q, QN);
  input D, CK, PD;
  output Q, QN;
  // 下降沿触发、高电平异步预置位、高电平异步清零的静态 D 型触发器
endmodule

module FD1S2DX (D, CK, CD, Q, QN);
  input D, CK, CD;
  output Q, QN;
  // 下降沿触发、高电平异步清零的静态 D 型触发器
endmodule

module FD1S2EX (D, CK, CDN, Q, QN);
  input D, CK, CDN;
  output Q, QN;
  // 下降沿触发、低电平异步清零的静态 D 型触发器
endmodule

module FD1S2FX (D, CK, PD, CDN, Q, QN);
  input D, CK, PD, CDN;
  output Q, QN;
```

```
  // 下降沿触发、低电平异步清零、高电平异步预置位的静态 D 型触发器
endmodule

module FD1S2GX (D, CK, PD, CDN, Q, QN);
  input D, CK, PD, CDN;
  output Q, QN;
  // 下降沿触发、低电平异步预置位的静态 D 型触发器
endmodule

module FD1S2IX (D, CK, CD, Q, QN);
  input D, CK, CD;
  output Q, QN;
  // 下降沿触发、高电平同步清零的静态 D 型触发器
endmodule

module FD1S2JX (D, CK, PD, Q, QN);
  input D, CK, PD;
  output Q, QN;
  // 下降沿触发、高电平同步预置位的静态 D 型触发器
endmodule

module FD1S2NX (D, CK, PDN, CD, Q, QN);
  input D, CK, PDN, CD;
  output Q, QN;
  // 下降沿触发、高电平异步清零、低电平异步预置位的静态 D 型触发器
endmodule

module FD1S2OX (D, CK, PD, CD, Q, QN);
  input D, CK, PD, CD;
  output Q, QN;
  // 下降沿触发、高电平同步清零、高电平同步预置位的静态 D 型触发器
endmodule

module FD1S3AX (D, CK, Q, QN);
  input D, CK;
  output Q, QN;
  // 上升沿触发的静态 D 型触发器
endmodule
```

```
module FD1S3BX (D, CK, Q, QN);
  input D, CK;
  output Q, QN;
  // 上升沿触发、高电平异步预置位的静态 D 型触发器
endmodule

module FD1S3CX (D, CK, Q, QN);
  input D, CK;
  output Q, QN;
  // 上升沿触发、高电平异步清零、高电平异步预置位的静态 D 型触发器
endmodule

module FD1S3EX (D, CK, CDN, Q, QN);
  input D, CK, CDN;
  output Q, QN;
  // 上升沿触发、低电平同步清零的静态 D 型触发器
endmodule

module FL1S2AX (D0, D1, CK, SD, Q, QN);
  input D0, D1, CK, SD;
  output Q, QN;
  // 下降沿触发、数据选择前置、支持扫描测试的触发器
endmodule

module FL1S2EX (D0, D1, CK, SD, CDN, Q, QN);
  input D0, D1, CK, SD, CDN;
  output Q, QN;
  // 下降沿触发、数据选择前置、低电平异步清零、支持扫描测试的触发器
endmodule

module FL1S3AX (D0, D1, CK, SD, Q, QN);
  input D0, D1, CK, SD;
  output Q, QN;
  // 上升沿触发、数据选择前置、支持扫描测试的触发器
endmodule

module FL1S3CX (D0, D1, CK, SD, Q, QN);
  input D0, D1, CK, SD;
  output Q, QN;
```

```
  // 上升沿触发、数据选择前置、高电平异步清零、高电平异步预置位、支持扫描测试的触发器
endmodule

module FL1S3EX (D0, D1, CK, SD, CDN, Q, QN);
  input D0, D1, CK, SD, CDN;
  output Q, QN;
  // 上升沿触发、数据选择前置、低电平异步清零、支持扫描测试的触发器
endmodule

module FS0S1D (S, R, CD, Q, QN);
  input S, R, CD;
  output Q, QN;
  // 高电平 S 输入有效、高电平 R 输入有效、高电平异步清零的 R-S 触发器(锁存器)
endmodule

module INRB (A, Z);
  input A;
  output Z;
  // Z = ! A;
endmodule

module INRBH (A, Z);
  input A;
  output Z;
  // Z = ! A; (与 INRB 相同)
endmodule

module ND2 (A, B, Z);
  input A, B;
  output Z;
  // Z = ! (A&B);
endmodule

module ND3 (A, B, C, Z);
  input A, B, C;
  output Z;
  // Z = ! (A&B&C);
endmodule
```

```
module ND4 (A, B, C, D, Z);
  input A, B, C, D;
  output Z;
  // Z = ! (A&B&C&D);
endmodule

module NR2 (A, B, Z);
  input A, B;
  output Z;
  // Z = ! (A | B);
endmodule

module NR3 (A, B, C, Z);
  input A, B, C;
  output Z;
  // Z = ! (A | B | C);
endmodule

module NR4 (A, B, C, D, Z);
  input A, B, C, D;
  output Z;
  // Z = ! (A | B | C | D);
endmodule

module OAI21 (A1, A2, B, Z);
  input A1, A2, B;
  output Z;
  // Z = ! ((A1 | A2) &B);
endmodule

module OAI22 (A1, A2, B1, B2, Z);
  input A1, A2, B1, B2;
  output Z;
  // Z = ! ((A1 | A2) &(B1 | B2));
endmodule

module OAI4321 (A1, A2, A3, A4, B1, B2, B3, C1, C2, D, Z);
  input A1, A2, A3, A4, B1, B2, B3, C1, C2, D;
  output Z;
```

```
  // Z = ! ((A1 | A2 | A3 | A4) & (B1 | B2 | B3):
  //      & (C1 & C2) & D);
endmodule

module OR2 (A, B, Z);
  input A, B;
  output Z;
  // Z = A | B;
endmodule

module OR4 (A, B, C, D, Z);
  input A, B, C, D;
  output Z;
  // Z = A | B | C | D;
endmodule

module TBUS (D, CK, CKN, Q);
  input D, CK, CKN;
  output Q;
  // Q = 'bz when (! CK && CKN);
  //     'b0 when (CK && ! D);
  //     'b1 when (! CKN && D);
endmodule

module XNOR2 (A, B, Z);
  input A, B;
  output Z;
  // Z = | (A ^ B);
endmodule

module XOR2 (A, B, Z);
  input A, B;
  output Z;
  // Z = A ^ B;
endmodule

module XOR2Z (A, B, Z, Z1);
  input A, B;
  output Z, Z1;
  // Z = A ^ B; Z1 = ! (A | B);
endmodule
```

参 考 文 献

1 Bhasker J. *A Verilog HDL Primer*. Allentown(PA): Star Galaxy Press, 1997

2 *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Std 1364-1995. IEEE, 1995

3 Lee James. *Verilog Quickstart*. MA: Kluwer Academic, 1997

4 Palnitkar S. *Verilog HDL: A Guide to Digital Design and Synthesis*. NJ: Prentice Hall, 1996

5 Sagdeo Vivek. *The Complete Verilog Book*. MA: Kluwer Academic, 1998

6 Smith Douglas. *HDL Chip Design*. AL: Doone Publications, 1996

7 Sternheim E, Singh R, Trivedi Y. *Digital Design with Verilog HDL*. CA: Automata Publishing Company, 1990

8 Thomas D, Moorby P. *The Verilog Hardware Description Language*. MA: Kluwer Academic, 1991

[General Information]
书名=Verlog HDL 综合实用教程
作者=
页数=172
SS号=0
出版日期=

封面
书名
版权
前言
目录
正文