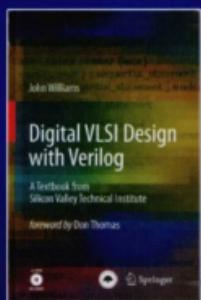


Verilog数字 VLSI设计教程

Digital VLSI Design with Verilog
A Textbook from Silicon Valley Technical Institute



[美] John Williams 著
李 林 陈亦欧 郭志勇 译
李广军 审校



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

含光盘1张

Verilog数字VLSI设计教程

Digital VLSI Design with Verilog

A Textbook from Silicon Valley Technical Institute

美国硅谷技术学院培训教材

本书分成多个课程段，讲授数字IC设计中常用技能与技术、工程设计中通常遇到的具体设计调试方法。其中包括数字IC设计流程中会遇到的诸多典型实例（计数器类型与结构、数据存储与Verilog阵列、状态机、FIFO等）以及典型问题（上升—下降延迟、串并转换、时序检查等），尤其是IC设计中PLL设计应用、时序仿真中的延迟反标注、DFT、设计验证等IC工程设计中的实用技术。通过给出设计实例，讲解此类问题的解决方案。

本书重在提高工程实践能力，读者对象为有一定硬件设计经验和数字电路基础的工程师以及掌握Verilog基本语法和数字设计基础知识的本科生。该书给出多个各自独立的单元，分别针对某个具体设计实例或设计中需要解决的问题展开详细讨论。自学的读者可以根据工作或学习的实际需要重点学习某些单元。作为培训教程，培训师可根据客户需求从众多练习中精选一部分开专题讲座。

光盘内容

光盘中包含了本书所有练习的完整答案。此外，还有一个后缀名为.tar的压缩文件，里面的内容就是这些答案，这只是为了方便读者将答案复制到Linux或UNIX环境里去。

在使用这张光盘之前，请先读光盘里的ReadMe.txt文件。

光盘里misc目录下的文件有练习要用到的文件列表和其他一些非核心设计的文件。目录里还有VCS和QuestSim仿真工具的PDF简要操作文档。

Springer

ISBN 978-7-121-10991-1



9 787121 109911 >

定价：45.00 元
(含光盘1张)



策划编辑：马 岚

责任编辑：李秦华

责任美编：李 雯



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

国外电子与通信教材系列
美国硅谷技术学院培训教材

Verilog 数字 VLSI 设计教程

Digital VLSI Design with Verilog
A Textbook from Silicon Valley Technical Institute

[美] John Williams 著

李林 陈亦欧 郭志勇 译
李广军 审校

电子工业出版社
Publishing House of Electronics Industry
北京 · BEIJING



内 容 简 介

本书分成多个课程段，讲授数字IC设计中常用技能与技术、工程设计中通常遇到的具体设计调试方法。其中包括数字IC设计流程中会遇到的诸多典型实例（计数器类型与结构、数据存储与Verilog阵列、状态机、FIFO等）以及典型问题（上升—下降延迟、串并转换、时序检查等），尤其是IC设计中PLL设计应用、时序仿真中的延迟反标注、DFT、设计验证等IC工程设计中的实用技术。通过给出设计实例，讲解此类问题的解决方案。

本书重在提高工程实践能力，读者对象为有一定硬件设计经验和数字电路基础的工程师以及掌握Verilog基本语法和数字设计基础知识的本科生。该书给出多个各自独立的单元，分别针对某个具体设计实例或设计中需要解决的问题展开详细讨论。自学的读者可以根据工作或学习的实际需要重点学习某些单元。作为培训教程，培训师可根据客户需求从众多练习中精选一部分开设专题讲座。

Translation from the English language edition:

Digital VLSI Design with Verilog by John Williams

Copyright © 2008 Springer, The Netherlands

as a part of Springer Science + Business Media

All rights Reserved

本书简体中文专有翻译出版权由Springer Science + Business Media授予电子工业出版社。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2009-0868

图书在版编目（CIP）数据

Verilog 数字 VLSI 设计教程 / (美) 威廉斯 (Williams, J.) 著；李林，陈亦歌，郭志勇译。—北京：电子工业出版社，2010.7
(国外电子与通信教材系列)

书名原文：Digital VLSI Design with Verilog: A Textbook from Silicon Valley Technical Institute

ISBN 978-7-121-10991-1

I. ①V… II. ①威… ②李… ③陈… ④郭… III. ①超大规模集成电路－电路设计－高等学校－教材
②硬件描述语言，Verilog－程序设计－高等学校－教材 IV. ①TN470.2 ②TP312

中国版本图书馆 CIP 数据核字（2010）第 100099 号

策划编辑：马 岚

责任编辑：李秦华

印 刷：北京市李史山胶印厂

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787 × 1092 1/16 印张：20.75 字数：531 千字

印 次：2010 年 7 月第 1 次印刷

定 价：45.00 元（含光盘 1 张）

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zits@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

序

2001年7月间，电子工业出版社的领导同志邀请各高校十几位通信领域方面的老师，商量引进国外教材问题。与会同志对出版社提出的计划十分赞同，大家认为，这对我国通信事业、特别是对高等院校通信学科的教学工作会很有好处。

教材建设是高校教学建设的主要内容之一。编写、出版一本好的教材，意味着开设了一门好的课程，甚至可能预示着一个崭新学科的诞生。20世纪40年代MIT林肯实验室出版的一套28本雷达丛书，对近代电子学科、特别是对雷达技术的推动作用，就是一个很好的例子。

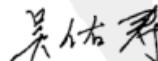
我国领导部门对教材建设一直非常重视。20世纪80年代，在原教委教材编审委员会的领导下，汇集了高等院校几百位富有教学经验的专家，编写、出版了一大批教材；很多院校还根据学校的特点和需要，陆续编写了大量的讲义和参考书。这些教材对高校的教学工作发挥了极好的作用。近年来，随着教学改革不断深入和科学技术的飞速进步，有的教材内容已比较陈旧、落后，难以适应教学的要求，特别是在电子学和通信技术发展神速、可以讲是日新月异的今天，如何适应这种情况，更是一个必须认真考虑的问题。解决这个问题，除了依靠高校的老师和专家撰写新的符合要求的教科书外，引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，是会有好处的。

一年多来，电子工业出版社为此做了很多工作。他们成立了一个“国外电子与通信教材系列”项目组，选派了富有经验的业务骨干负责有关工作，收集了230余种通信教材和参考书的详细资料，调来了100余种原版教材样书，依靠由20余位专家组成的出版委员会，从中精选了40多种，内容丰富，覆盖了电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等方面，既可作为通信专业本科生和研究生的教学用书，也可作为有关专业人员的参考材料。此外，这批教材，有的翻译为中文，还有部分教材直接影印出版，以供教师用英语直接授课。希望这些教材的引进和出版对高校通信教学和教材改革能起一定作用。

在这里，我还要感谢参加工作的各位教授、专家、老师与参加翻译、编辑和出版的同志们。各位专家认真负责、严谨细致、不辞辛劳、不怕琐碎和精益求精的态度，充分体现了中国教育工作者和出版工作者的良好美德。

随着我国经济建设的发展和科学技术的不断进步，对高校教学工作会不断提出新的要求和希望。我想，无论如何，要做好引进国外教材的工作，一定要联系我国的实际。教材和学术专著不同，既要注意科学性、学术性，也要重视可读性，要深入浅出，便于读者自学；引进的教材要适应高校教学改革的需要，针对目前一些教材内容较为陈旧的问题，有针对性地引进一些先进的和正在发展中的交叉学科的参考书；要与国内出版的教材相配套，安排好出版英文原版教材和翻译教材的比例。我们努力使这套教材能尽量满足上述要求，希望它们能放在学生们的课桌上，发挥一定的作用。

最后，预祝“国外电子与通信教材系列”项目取得成功，为我国电子与通信教学和通信产业的发展培土施肥。也恳切希望读者能对这些书籍的不足之处、特别是翻译中存在的问题，提出意见和建议，以便再版时更正。



中国工程院院士、清华大学教授

“国外电子与通信教材系列”出版委员会主任

出版说明

进入21世纪以来，我国信息产业在生产和科研方面都大大加快了发展速度，并已成为国民经济发展的支柱产业之一。但是，与世界上其他信息产业发达的国家相比，我国在技术开发、教育培训等方面都还存在着较大的差距。特别是在加入WTO后的今天，我国信息产业面临着国外竞争对手的严峻挑战。

作为我国信息产业的专业科技出版社，我们始终关注着全球电子信息技术的发展方向，始终把引进国外优秀电子与通信信息技术教材和专业书籍放在我们工作的重要位置上。在2000年至2001年间，我社先后从世界著名出版公司引进出版了40余种教材，形成了一套“国外计算机科学教材系列”，在全国高校以及科研部门中受到了欢迎和好评，得到了计算机领域的广大教师与科研工作者的充分肯定。

引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，将有助于我国信息产业培养具有国际竞争能力的技术人才，也将有助于我国国内在电子与通信教学工作中掌握和跟踪国际发展水平。根据国内信息产业的现状、教育部《关于“十五”期间普通高等教育教材建设与改革的意见》的指示精神以及高等院校老师们反映的各种意见，我们决定引进“国外电子与通信教材系列”，并随后开展了大量准备工作。此次引进的国外电子与通信教材均来自国际著名出版商，其中影印教材约占一半。教材内容涉及的学科方向包括电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等，其中既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。我们还将与国外出版商一起，陆续推出一些教材的教学支持资料，为授课教师提供帮助。

此外，“国外电子与通信教材系列”的引进和出版工作得到了教育部高等教育司的大力支持和帮助，其中的部分引进教材已通过“教育部高等学校电子信息科学与工程类专业教学指导委员会”的审核，并得到教育部高等教育司的批准，纳入了“教育部高等教育司推荐——国外优秀信息科学与技术系列教学用书”。

为做好该系列教材的翻译工作，我们聘请了清华大学、北京大学、北京邮电大学、南京邮电大学、东南大学、西安交通大学、天津大学、西安电子科技大学、电子科技大学、中山大学、哈尔滨工业大学、西南交通大学等著名高校的教授和骨干教师参与教材的翻译和审校工作。许多教授在国内电子与通信专业领域享有较高的声望，具有丰富的教学经验，他们的渊博学识从根本上保证了教材的翻译质量和专业学术方面的严格与准确。我们在此对他们的辛勤工作与贡献表示衷心的感谢。此外，对于编辑的选择，我们达到了专业对口；对于从英文原书中发现的错误，我们通过与作者联络、从网上下载勘误表等方式，逐一进行了修订；同时，我们对审校、排版、印制质量进行了严格把关。

今后，我们将进一步加强同各高校教师的密切关系，努力引进更多的国外优秀教材和教学参考书，为我国电子与通信教材达到世界先进水平而努力。由于我们对国内外电子与通信教育的发展仍存在一些认识上的不足，在选题、翻译、出版等方面的工作中还有许多需要改进的地方，恳请广大师生和读者提出批评及建议。

电子工业出版社

教材出版委员会

主任	吴佑寿	中国工程院院士、清华大学教授
副主任	林金桐	北京邮电大学校长、教授、博士生导师
	杨千里	总参通信部副部长，中国电子学会会士、副理事长
		中国通信学会常务理事、博士生导师
委员	林孝康	清华大学教授、博士生导师、电子工程系副主任、通信与微波研究所所长 教育部电子信息科学与工程类专业教学指导分委员会委员
	徐安士	北京大学教授、博士生导师、电子学系主任
	樊昌信	西安电子科技大学教授、博士生导师 中国通信学会理事、IEEE 会士
	程时昕	东南大学教授、博士生导师
	郁道银	天津大学副校长、教授、博士生导师 教育部电子信息科学与工程类专业教学指导分委员会委员
	阮秋琦	北京交通大学教授、博士生导师 计算机与信息技术学院院长、信息科学研究所所长 国务院学位委员会学科评议组成员
	张晓林	北京航空航天大学教授、博士生导师、电子信息工程学院院长 教育部电子信息科学与电气信息类基础课程教学指导分委员会副主任委员 中国电子学会常务理事
	郑宝玉	南京邮电大学副校长、教授、博士生导师 教育部电子信息与电气学科教学指导委员会委员
	朱世华	西安交通大学副校长、教授、博士生导师 教育部电子信息科学与工程类专业教学指导分委员会副主任委员
	彭启琮	电子科技大学教授、博士生导师、通信与信息工程学院院长 教育部电子信息科学与电气信息类基础课程教学指导分委员会委员
	毛军发	上海交通大学教授、博士生导师、电子信息与电气工程学院副院长 教育部电子信息与电气学科教学指导委员会委员
	赵尔沅	北京邮电大学教授、《中国邮电高校学报（英文版）》编委会主任
	钟允若	原邮电科学研究院副院长、总工程师
	刘彩	中国通信学会副理事长兼秘书长、教授级高工 信息产业部通信科技委副主任
	杜振民	电子工业出版社原副社长
	王志功	东南大学教授、博士生导师、射频与光电集成电路研究所所长 教育部高等学校电子电气基础课程教学指导分委员会主任委员
	张中兆	哈尔滨工业大学教授、博士生导师、电子与信息技术研究院院长
	范平志	西南交通大学教授、博士生导师、信息科学与技术学院院长

译者序

本书与其他讲述使用 Verilog HDL 设计数字集成电路的教材不同，书中除了详细讲授了 Verilog 的语法和用法，还把数字 IC 前端设计的全流程贯穿于书中的各个章节。由于这本书源自美国硅谷技术学院（Silicon Valley Technical Institute，拥有 Synopsys 及 Mentor 的认证）最近的数字 IC 培训课程，从而使中国读者有机会接受最先进、最权威的数字 IC 工程设计培训。

本书第 1 章的首个练习，是基于 90 nm 的 TSMC 库用 DC 去综合一个简单设计并产生 SDF 文件。由此可见，本书的确非常贴近实际 IC 工程项目设计。有些工程师或读者虽然已学习了 Verilog 语言，但在参与实际数字 IC 项目设计后却仍会感到茫然，这是由于进行 RTL 设计仅仅是数字 IC 设计众多环节中的一环，这本书可为这些读者解惑。本书作者为 John Williams，有多年的 IC 设计及教育培训经验，他深知一个数字 IC 设计工程师或读者最迫切需要了解和掌握什么，那就是：具有在最短的时间里承担、完成实际 IC 工程设计任务的能力。

本书共有 24 章，按作者的计划，一个星期将会完成两章内容的学习，学习完本书的时间是三个月。在这三个月内，读者将循序渐进地学习理解 Verilog 的基本语法知识，以及如何用 Verilog 来设计常用电路（状态机、FIFO 等）、事件队列、Verilog 强大的时序定义及检查功能、DFT、BIST、SDF 和电路综合等内容。同时，把目前热门的 Serdes 和 PLL 设计技术作为工程实例贯穿于全书，且每一步实际操作都有详细的说明，光盘里附有完整的源代码、testbench 及 DC 的综合脚本等。

译者曾接触过一些年轻的 IC 设计工程师，他们能够利用 Verilog 写出可以实现复杂功能的逻辑电路，但当他们在涉及后仿真、基于库的延迟信息分析等工程设计环节中，表现出分析描述时序相关语句的能力却很弱。显然，这是因为他们学习 Verilog 设计 IC 时未涉及到这方面的内容和训练。本书在这方面做了全面的讨论，读者可以深入了解这些知识。

本书对 Verilog 及数字 IC 前端设计做了深入的讲授，本书对希望从事数字 IC 前端设计和对于从事 FPGA 开发的读者来说都是一本很好的教材。读者可以根据自己的实际情况重点安排其学习计划。由于本书源自工程培训课程，本书也很适合用做培训教材。培训教师可根据实际课堂需求安排学生的学习计划。

本书由李林、陈亦欧、郭志勇翻译，李广军审校。电子工业出版社为本书的后期出版做了大量的工作。在此，对所有为本书出版提供帮助的人士表示诚挚的谢意！

由于译审者水平有限，加之时间仓促，译文中难免有不妥之处，敬请读者不吝指正。

序 言

自从 20 世纪 80 年代中期 Phil Moorby 首创了 Verilog 以来, Verilog 这门 HDL 语言及其用法不断发展。由于当时数字电路设计的平均规模大约是 10 000 门, 因此当时 Verilog 的主要用途就是提供一种仿真的方法来确保设计的正确、有效。随着设计规模的急剧扩大, 先完成 RTL 级设计, 再完成自动逻辑综合, 这已经成为现在大多数设计的标准流程。随着这一进程的逐步成熟, Verilog 这门语言本身也在不断演进和更新。

这些年出版了很多关于 Verilog 的书。为了发展和推广这门语言, 我自己也和 Phil Moorby 一起合作提供了大量的范例。随着 Verilog 语言的 5 个新版本及其用法的不断演进, 这些范例也已随之更新。

然而, 本书的视角和以前的很多书籍都不同, 很独特。本书的作者 John Williams 多年从事 ASIC 设计和教学。他给大家带来了对 Verilog 这门语言非常深入的讲解, 并且详细地指导读者怎样配合逻辑综合工具来使用这门语言。没有其他的 Verilog 书籍这么深入浅出地涉及了这个话题。

如果想学习 Verilog, 并且想尽快地利用这门语言进行可综合的设计, 那么这就是你需要的书了。本书按章节, 循序渐进地为读者灌输了新的概念, 以及如何把这些新概念和知识运用在设计中。此外, 每一章都附有精心设计的练习和详细的练习指导。读者可以根据书中的章节来安排自己的学习进度。这也是一条我们自己走过的路。也就是说, 先学习基础概念, 然后在练习里运用它们, 逐渐深入到高级的课题中去。由于作者对于 Verilog 的教学经验非常丰富, 因此本书的章节组织得很恰当合理, 对读者学习 Verilog 会有极大帮助。

Don Thomas
宾夕法尼亚州; 匹兹堡
2008

前　　言

本书基于作者在硅谷技术学院 (Silicon Valley Technical Institute) (位于美国加州圣何塞市) 多年的教学经验, 由作者在教学课件及对应的课程练习的基础上提炼出来的。

据作者所知, 这门课程是到目前为止唯一做到了以下三点的:

- (a) 完整、深入地讲解了 Verilog 语言
- (b) 实现了全双工串行 / 解串器的仿真模型
- (c) 实现了可综合的数字锁相环

在这门课程的发展和准备本书的过程中, 硅谷技术学院的 CEO, Ali Iranmanesh 博士给予了极大的鼓励和帮助。在这里, 作者表示由衷的感谢。



目 录

第 0 章 概述	1
0.1 课程描述	1
0.2 如何使用本书	1
0.3 参考文献	3
0.4 推荐的互动 Verilog 教程	5
第 1 章 Verilog 入门	6
1.1 练习 1	6
1.2 Verilog 矢量	16
1.3 练习 2：操作数	18
1.4 小结	19
阅读 Palnitkar (2003) (可选)	21
第 2 章 Verilog 基础知识 1	22
2.1 更多的语言结构	22
2.2 练习 3：参数和转换	28
2.3 过程控制	30
2.4 练习 4：非阻塞控制	35
阅读 Palnitkar (2003) (可选)	39
第 3 章 Verilog 基础知识 2	40
3.1 线型，仿真和扫描	40
3.2 练习 5：简单的扫描	48
阅读 Palnitkar (2003) (可选)	53
第 4 章 锁相环和串行 / 解串器入门	54
4.1 锁相环和串行 / 解串器工程	54
4.2 练习 6：PLL 时钟	62
第 5 章 存储与数组	71
5.1 数据存储与 Verilog 数组	71
5.2 练习 7：存储器	80
阅读 Palnitkar (2003) (可选)	83
第 6 章 计数器	84
6.1 计数器的类型与结构	84

6.2 练习 8: 计数器	89
阅读 Palnitkar (2003) (可选)	92
第 7 章 强度和竞争	93
7.1 竞争和操作符的优先级	93
7.2 数字基础: 三态缓冲和解码器	99
7.3 练习 9: 强度和竞争	100
7.4 接着讨论 PLL 和串行 / 解串器	105
7.5 练习 10: PLL 行为级锁定	114
阅读 Palnitkar (2003) (可选)	116
第 8 章 状态机和 FIFO	117
8.1 状态机和 FIFO 设计	117
8.2 练习 11: FIFO	130
阅读 Palnitkar (2003) (可选)	133
第 9 章 事件	134
9.1 上升 - 下降延迟和事件计划	134
9.2 练习 12: 计划	141
阅读 Palnitkar (2003) (可选)	145
第 10 章 内建器件	146
10.1 内建的门及线型	146
10.2 练习 13: 网表	151
阅读 Palnitkar (2003) (可选)	153
第 11 章 顺序控制和并发	154
11.1 顺序控制和并发	154
11.2 练习 14: 并行	163
阅读 Palnitkar (2003) (可选)	165
第 12 章 层次和 generate	166
12.1 层次命名和 generate 块	166
12.2 练习 15: generate	175
阅读 Palnitkar (2003) (可选)	179
第 13 章 函数、任务和串并转换	180
13.1 串并转换	180
13.2 练习前预习: 解串器	182
13.3 练习 16: 串并转换	185
第 14 章 UDP 和开关级模型	189
14.1 用户定义原语、时序参数和开关级模型	189

14.2 练习 17: 元件	196
阅读 Palnitkar (2003) (可选)	200
第 15 章 参数和层次	201
15.1 参数的类型与模块连接	201
15.2 练习 18: 连线	203
15.3 层次命名和设计划分	207
15.4 练习 19: 层次	211
第 16 章 配置和时序	214
16.1 Verilog 的配置	214
16.2 时序弧和 specify 延迟	215
16.3 练习 20: 时序	221
阅读 Palnitkar (2003) (可选)	224
第 17 章 时序检查和断言	225
17.1 时序检查和脉冲控制	225
17.2 练习 21: 时序检查	233
阅读 Palnitkar (2003) (可选)	236
第 18 章 解串器和升级 PLL	237
18.1 串行序列解串器	237
18.2 重新设计 PLL	238
18.3 练习 22: 串行序列解串器	245
第 19 章 升级解串器	256
19.1 并行解串器	256
19.2 练习 23: 解串器	258
第 20 章 完成串行 / 解串器	273
20.1 串行器和串行 / 解串器	273
20.2 练习 24: 串行 / 解串器	274
第 21 章 可测性设计和全双工串行 / 解串器	283
21.1 可测性设计	283
21.2 练习 25: 扫描和 BIST	289
21.3 全双工串行 / 解串器的 DFT	295
21.4 练习 26: 测试 SerDes	296
第 22 章 SDF	304
22.1 SDF 反标	304
22.2 练习 27: SDF	305

第 23 章	Verilog 语言总结	309
23.1	Verilog 语言总结	309
23.2	课后练习（继续完成练习 23 及以后的练习）	313
阅读	Palnitkar (2003) (可选)	313
第 24 章	深亚微米的问题及其验证	314
24.1	深亚微米的问题及其验证	314
24.2	课后练习（继续完成练习 23 及以后的练习）	319
阅读	Palnitkar (2003) (可选)	319

第0章 概述

0.1 课程描述

本书可以用做为期 12 周课程的教材和练习指导手册，每周学习两章。本书已经按学习进度划分了章节。

本课程适用于已经获得电子工程方向本科学位，或有相同数字设计经验的人员。除了有数字设计的背景要求外，还要求课程的参加者熟悉一门现代的软件编程语言，如 C 语言。

如果要完整地学习书中的内容并完成所有的练习，大概需要在这 12 周里每周都拿出 12 个小时的时间。当然，读者可以根据自己的情况来调整进度。因此，学习这本书的时间会比参加固定进度的课程的时间要灵活得多。

本书的部分内容

讨论：模块和层次关系；阻塞和非阻塞赋值；组合逻辑；时序逻辑；行为模型；RTL 模型；门级模型；硬件时序和延迟；参数；基本的系统任务；时序检查；Generate 语句；仿真事件计划；条件竞争；综合操作；可综合的结构；网表仿真；综合控制原语；Verilog 对综合优化的影响；SDF 文件；测试结构；补错基础等。

练习的工程：移位和扫描寄存器；计数器；存储器与 FIFO；数字锁相环（PLL）；串并转换器；串行/解串器；原语门；开关级设计；网表反标等。

0.2 如何使用本书

作者推荐读者按章节顺序阅读这本书。一个知识点可能会在书中的多个部分讲到，通常是这样的：先是简要的，不完整的提到了一个新的想法或语法的特性；然后在数页之后，可能再继续深入地讨论这个话题或知识点。

每章的最后都有补充阅读的内容，阅读的书籍是两本广受好评的 Verilog 书籍。一本书的作者是 Thomas 和 Moorby，另一本书的作者是 Palnitkar。如果读者在学习并完成了练习之后，觉得有的内容仍然很难理解。那么，读者也许应该去读这个简介后的那些参考资料。

0.2.1 光盘中的内容

光盘里包含了本书所有练习的完整答案。此外，还有一个后缀名为 tar 的压缩文件，里面的内容就是这些答案，这只是为了方便读者将答案复制到 Linux 或 UNIX 环境里去。

在使用这张光盘之前，请记得先读光盘里的 ReadMe.txt 文件。

光盘里 misc 目录下的文件有练习 1 要用到的文件列表和其他一些非核心设计的文件。目录里还有 VCS 和 QuestSim 仿真工具的 PDF 简要操作文档。

misc 目录里还有作者自己写的非私有的 Verilog 库文件。在进行网表仿真的时候，这些库可以提供接近实际情况的仿真模型。如果你在设计里使用的是 TSMC（台积电）的库，那么这些仿真模型是不准确的。但是，用这些库来学习是没有问题的。请记住，这些库仅能在练习的设计中使用。如果你的设计会在 TSMC 流片，请使用符合 Synopsys 标准的、有正确反标时序信息的 TSMC 库。

0.2.2 完成练习

本书包含了详尽的练习步骤。在做一个练习之前，可以把光盘对应的目录里的文件和目录都复制到你的练习环境中去。

读者在进行练习的时候，必须要有使用仿真工具的权限。如果读者没有使用 EDA 工具的权限也没有关系，光盘中附带了试用版的 Silos 仿真工具。除了串行/解串器这个大工程 Silos 可能应付不了之外，其余练习的仿真都可以用这个工具来完成。你还可以使用学生版的 Aldec 仿真工具，它的功能还要强大一些。Verilog 的仿真工具通常也会随着 FPGA 硬件开发套件免费赠送。但是，如果要用 ASIC 库做网表仿真，则需要更高级的仿真工具，例如 VCS 和 QuestaSim。

如果有条件，最好使用 Synopsys 的综合工具 DC 和仿真工具 VCS 来完成本书的练习。设计 VCS 这个软件的目的就是为了辅助大规模集成电路的设计，而 VCS 也是进行大规模集成电路设计最好的仿真工具之一。

请记住，无论是哪一种 Verilog 仿真工具都不会完整地支持 Verilog 的所有特性。不同的仿真工具支持的功能是有差别的。在设计练习的答案时，作者尽量保证了这些代码在多种仿真工具里都没有问题。

作者完成光盘里的练习用的是如下配置：软件是 Synopsys Design Compiler (Z-2007.03 SP2) 和 VCS(MX 2007)，操作系统是 Red Hat Enterprise Linux 3，硬件的配置是 384 MB 的内存和 1 GHz 的 x86 CPU。综合的时候用到的库是 TSMC 90 nm 前端的库（典型的 PVT 参数）。部分专业的读者可能对这些信息感兴趣。

0.2.3 私有信息和许可限制

对于公开发表 VCS、DC 或 QuestaSim 的工作性能细节的行为，可能需要 Synopsys 或 Mentor 的书面允许。因此，如果读者使用的是这些 EDA 工具来完成本书的练习，作者建议读者在没有确保自己得出的信息没有包含任何有可能泄露自己所使用工具的详细特性或其他私有信息之前，不要复制或发表这些相关的信息。这是许可的问题，与版权、技术专利这些都无关。请先了解你所使用的 EDA 工具的许可限制。

对于 TSMC 的库来说也是这样的。虽然前端库仅被设计用来完成综合、布局和时序验证，但是它们也许包含了 TSMC 的秘密。如果没有 TSMC 和 Synopsys 的特殊允许，不要将 TSMC 库和相关文档里的任何内容复制到别处。

光盘里的 Verilog 仿真库虽然看起来也像是私有的，但是，这个仿真库并不是由 Synopsys 或 TSMC 产生的。应该认为它们是有版权的，但它们却不是私有的。对于购买了这本书的读者，出于个人学习的目的，可以复制或修改这些模型。

虽然在综合时，综合工具使用了 Synopsys 所有的综合库，但综合出来的网表不是私有的。而关于综合网表的某些综合性能细节的内容，在没有得到 Synopsys 的特许前，不应该公开发表。

0.3 参考文献

(链接之后的日期是该链接被作者使用时的新日期)

- Accellera Organization. *System Verilog Language Reference Manual v. 3.1a*. Draft standard available free for download from Accellera web site at <http://www.accellera.org/home>.
- Anonymous. "Design for Test (DFT)", Chapter 3 of *The NASA ASIC Guide: Assuring ASICs for Space*. http://klabs.org/DEI/References/design_guidelines/content/guides/nasa_asic_guide/Sect.3.3.html (2003-12-30 更新).
- Anonymous. *SerDes Transceivers*. Freescale Semiconductor, Inc. <http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01HGPJ2350NbKQ> (2004-11-16).
- Barrett, C. (Ed.) *Fractional/Integer-N PLL Basics*. Texas Instruments Technical Brief SWRA029, August, 1999. <http://focus.ti.com/lit/an/swra029/swra029.pdf> (2004-12-09).
- Bertrand, R. "The Basics of PLL Frequency Synthesis", in the *Online Radio and Electronics Course*. <http://www.radioelectronicschool.com/reading/pl1.pdf> (2004-12-09).
- Bhasker, J. A *Verilog HDL Primer* (3rd ed.). Allentown, Pennsylvania: Star Galaxy Publishing, 2005.
- Cipra, B. A. "The Ubiquitous Reed-Solomon Codes". *SIAM News*, 26(1), 1993. http://www.eccpage.com/reed_solomon_codes.html (2007-09-18).
- Cummings, C. E. "Simulation and Synthesis Techniques for Asynchronous FIFO Design" (rev. 1.1). Originally presented at San Jose, California: The *Synopsys Users Group Conference*, 2002. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1_rev1_1.pdf (2004-11-22).
- Cummings, C. E. and Alfke, P. "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons" (rev. 1.1). Originally presented at San Jose, CA: The *Synopsys Users Group Conference*, 2002. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2_rev1_1.pdf (2004-11-22).
- IEEE Std 1364-2005. *Verilog Hardware Description Language*. Piscataway, New Jersey: The IEEE Computer Society, 2005. Revised in 2001 and reaffirmed, with some System Verilog added compatibility, in 2005. If you plan to do any serious work in verilog, you should have a copy of the standard. It is not only normative, but it includes numerous examples and explanatory notes concerning every detail of the language. In this text, we refer to "verilog 2001" syntax where it is the same as in the 2005 standard.
- Keating, M., et al. *Low Power Methodology Manual for System-on-Chip Design*. Springer Science and Business Solutions, 2007. Available from Synopsys as a free PDF for personal use only: <http://www.synopsys.com/lpmm> (2007-11-06).

- Knowlton, S. "Understanding the Fundamentals of PCI Express". Synopsys White Paper, 2007. Available at the Technical Bulletin, Technical Papers page at <http://www.synopsys.com/products/designware/dwtb/dwtb.php>. Free registration and login (2007-10-17).
- Koeter, J. *What's an LFSR?*, at <http://focus.ti.com/lit/an/scta036a/scta036a.pdf> (2007-01-29).
- Mead, C. and Conway, L. *Introduction to VLSI Systems*. Menlo Park, CA: Addison-Wesley, 1980. Excellent but old introduction to switch-level reality and digital transistor design and fabrication.
- Palnitkar, S. *Verilog HDL* (2nd ed.). Palo Alto, CA: Sun Microsystems Press, 2003. A good basic textbook useful for supplementary perspective. Also includes a demo version of the *Silos* simulator on CD-ROM. Our daily *Additional Study* recommendations include many optional readings and exercises from this book.
- Seat, S. "Gearing Up Serdes for High-Speed Operation", posted at http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=16504769 (2004-11-16).
- Suckow, E. H. "Basics of High-Performance SerDes Design: Part I" at http://www.analogzone.com/iot_0414.pdf; and, Part II at http://www.analogzone.com/iot_0428.pdf (2004-11-16).
- Sutherland, S. "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need it". Based on an *HDLCon 2000* presentation. http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf (2005-02-03).
- Thomas, D. E. and Moorby, P. R. *The Verilog Hardware Description Language* (5th ed.). New York: Springer, 2002. A very good textbook which was used in the past as the textbook for this course. Includes a demo version of the *Silos* simulator on CD-ROM. Our *Additional Study* recommendations include many optional readings and exercises from this book.
- Wallace, H. "Error Detection and Correction Using the BCH Code" (2001). <http://www.aqdi.com/bch.pdf> (2007-10-04).
- Wang, D. T. "Error Correcting Memory - Part I". <http://www.realworldtech.com/page.cfm?ArticleID=RWT121603153445&p=1> (2004-12-15: there doesn't seem to be any Part II).
- Weste, N. and Eshraghian, K. *Principles of CMOS VLSI Design: A Systems Perspective*. Menlo Park, CA: Addison-Wesley, 1985. Old, but overlaps and picks up where Mead and Conway leave off, especially on the CMOS technology *per se*.
- Zarrineh, K., Upadhyaya, S. J., and Chickermane, V. "System-on-Chip Testability Using LSSD Scan Structures", *IEEE Design & Test of Computers*, May-June 2001 issue, pp. 83-97.
- Ziegler, J. F. and Puchner, H. *SER - History, Trends, and Challenges*. San Jose, Cypress Semiconductor Corporation, 2004. (stock number 1-0704SERMAN). Contact: serquestions@cypress.com.
- Zimmer, P. "Working with PLLs in PrimeTime - avoiding the 'phase locked oops'". Drafted for a Synopsys User Group presentation in 2005. Downloadable at <http://www.zimmerdesignservices.com> (2007-04-12).

0.4 推荐的互动 Verilog 教程

Evita Verilog 教程。在 MS Windows 的环境里使用。读者可以免费下载并使用。这个教程的赞助商是 Aldec，下载的网址是 <http://www.aldec.com/Downloads>。这个教程里的语言规范是 Verilog-1995。

这个互动的教程包含动画，选择对错的小测验，Verilog 语言的参考和搜索工具。此外，在这个教程里有些微小的错误。这个教程运行起来很简单，它从另外一个视角介绍了 Verilog 这门语言，对于刚开始学习本书前几章内容的读者来说也许会有帮助。



第1章 Verilog入门

1.1 练习 1

练习步骤

这是一个很基础的练习，读者只需按步骤提示操作即可。在陈述练习步骤前，先进行如下说明：Verilog源文件的扩展名是.v。用于练习的设计文件已在文本文件Intro_Top.vcs中列出，读者可以直接用VCS仿真工具调用这个文件列表进行仿真。

光盘misc目录下提供了一个include文件Extras.inc。还需要一个TestBench.v来调用它，或者把这个文件直接复制到读者所使用的仿真工具可以访问的目录下。在自己的TestBench.v里添加`include "../VCS/Extras.inc"`即可。注意，`include`不以分号结尾。如果注释掉`include`，仿真不会受到影响，但是仿真期间不会产生VCD文件^①。

第1步。用文本编辑器打开Lab01目录下顶层设计文件Intro_Top.v，其对应的电路结构图如图1.1所示。

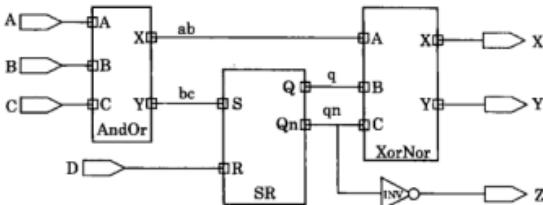


图1.1 Intro_Top.v对应电路的示意图

在Verilog的源文件Intro_Top.v里，文件的说明和代码的注释都是以//作为开头。为了便于说明问题，下面附上了整个文件的内容。

我们看到，几乎所有的代码都被包含在“module”（模块）中。Intro_Top.v顶层模块包括有端口声明，线型声明，一个赋值语句（连续赋值的表达式）和三个元件例化的描述。基于元件例化的描述通常用于表示模块的结构设计。

```
// =====
// Intro.Top: Top level of a simple design using verilog
// continuous assignment statements.
//
// This module contains the top structure of the design, which
```

① testbench产生了仿真设计时用到的激励，并检查了与激励对应的输出结果。国内有书籍把testbench翻译成测试平台。但由于testbench已是业界通用的术语，本书直接使用testbench这个名词——译者注。

```

// is made up of three lower-level modules and one inverter gate.
// The structure is represented by module instances.
//
// All ports are wire types, because this is the default; also,
// there is no storage of state in combinational statements.
//
// ANSI module header.
// -----
// 2004-11-25 jmw: v. 1.0 implemented.
// =====
module Intro.Top( output X, Y, Z, input A, B, C, D);
//
wire ab, bc, q, qn; // Wires for internal connectivity.
//
// Implied wires may be assumed in this combinational
// design, when connecting declared ports to instance ports.
// The #1 is a delay time, in 'timescale units:
//
assign #1 Z = ~qn; // Inverter by continuous assignment statement.
//
AndOr InputCombo01 (.X(ab), .Y(bc), .A(A), .B(B), .C(C));
SR SRLatch01 (.Q(q), .Qn(qn), .S(bc), .R(D));
XorNor OutputCombo01 (.X(X), .Y(Y), .A(ab), .B(q), .C(qn));
//
endmodule // Intro.Top.

```

第2步。TestBench.v 是对以上设计进行逻辑仿真的 Verilog 模块，代码内容如下所述。
图 1.2 给出了等效的原理图。

```

// =====
// TestBench: Simulation driver module (stimulus block)
// for the top-level block instance of Intro.Top.
//
// This module includes an initial block which assigns various
// values to top-level inputs for simulation. initial blocks
// are ignored in logic synthesis.
//
// No module port declaration.
// -----
// 2004-11-25 jmw: v. 1.0 implemented.
// =====
//
'timescale 1 ns/100ps // No semicolon after 'anything.
//
module TestBench; // Stimulus blocks have no port.
//
wire Xwatch, Ywatch, Zwatch; // To connect to design instance.
reg Astim, Bstim, Cstim, Dstim; // To accept initialization.
//
initial
begin
//
// Each '#' precedes a delay time increment, here in 1 ns units:
//
#1 Astim = 1'b0; // For Astim, 1 bit, representing a binary 0.
#1 Bstim = 1'b0;
#1 Cstim = 1'b0;

```

```
(other stimuli omitted here)
#50 Dstim = 1'b1;
#50 Astim = 1'b0;
#50 Cstim = 1'b0;
#50 Dstim = 1'b0;
#50 $finish;           // Terminates simulation 50 ns after the last stimulus.
end // No semicolon after end.
//
// The instance of the design is named Topper01, and its
// ports are associated by name with stimulus input and simulation
// output wires:
//
Intro.Top Topper01 (.X(Xwatch), .Y(Ywatch), .Z(Zwatch)
                     ,.A(Astim), .B(Bstim), .C(Cstim), .D(Dstim)
                   );
//
endmodule // TestBench.
```

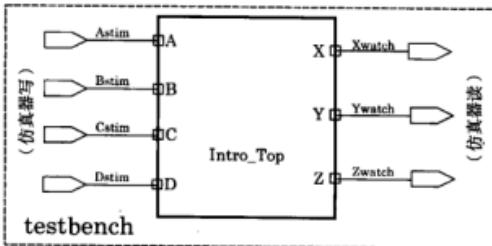


图 1.2 练习 1 中的 testbench 对应电路的示意图

Verilog 仿真工具从被测设计左边的输入端口施加激励，同时从右边的输出端口读取结果。由于 TestBench.v 里的测试模块并不实现设计功能，因此它不会被综合。

有两个 Verilog 的符号很接近，在使用时需要注意：

- 时标说明符 `timescale 采用符号是反引号 “`”，这个符号同样在宏的定义和编译指示中也被使用，如 `define、`include、`ifdef、`else、`endif 等。
- 数据宽度说明符，如 1'b1 等定义中，采用符号是单引号 “'”。

第 3 步。简要地浏览一下设计中的其他三个模块，分别为 AndOr.v、SR.v 和 XorNor.v。

```
// =====
// AndOr: Combinational logic using & and | .
// This module represents simple combinational logic including
// an AND and an OR expression.
//
// ANSI module header.
//
// -----
// 2004-11-25 jmw: v. 1.0 implemented.
// =====
module AndOr (output X, Y, input A, B, C);
  //
  assign #10 X = A & B;
  assign #10 Y = B | C;
  //
endmodule // AndOr.
```

代码里用到的赋值语句是连续赋值语句，以关键字assign开头。如图1.3所示，代码实现了图中的电路。我们看到，连续赋值实现了线的互连。符号“=”把其左边和右边的信号相连，这种连接关系为永久连接，在仿真期间也不会被改变。

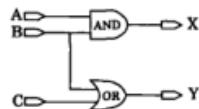


图 1.3 AndOr.v 对应
电路的示意图

assign 的意义为“连到该线”。

文字“# 10”代表仿真工具的仿真延时。

```
// =====
// SR: An S-R Latch using ~ and &.
// This module represents the functionality of a simple latch,
// which is a sequential logic device, using combinational
// ~AND expressions connected to feed back on each other.
//
// ANSI module header.
//
// -----
// 2005-04-09 jmw: v. 1.1 modified comment on wire declarations.
// 2004-11-25 jmw: v. 1.0 implemented.
// =====
module SR (output Q, Qn, input S, R);
    wire q, qn; // For internal wiring.
    //
    assign #1 Q = q;
    assign #1 Qn = qn;
    //
    assign #10 q = ~(S & qn);
    assign #10 qn = ~(R & q );
    //
endmodule // SR.
```

SR.v 代码中 4 个连续赋值语句所表达的映射含义如图 1.4 所示。

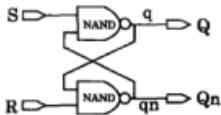


图 1.4 SR.v 对应电路的示意图

```
// =====
// XorNor: Combinational logic using ^ and ~|.
// This module represents simple combinational logic including
// an XOR and a NOR expression.
//
// ANSI module header.
//
// -----
// 2005-04-09 jmw: v. 1.1 modified comment on wire declarations.
```

```

// 2004-11-25 jmw: v. 1.0 implemented.
// =====
module XorNor (output X, Y, input A, B, C);
    wire x; // To illustrate use of internal wiring.
    //
    assign #1 X = x; // Verilog is case-sensitive; 'X' and 'x' are different.
    //
    assign #10 x = A ^ B;
    assign #10 Y = ~(x | C);
    //
endmodule // XorNor.

```

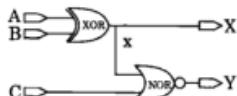


图 1.5 XorNor.v 对应
电路的示意图

另外 3 个连续赋值语句所描述的互连映射原理如图 1.5 所示。

第 4 步。用仿真工具载入 TestBench.v 并进行仿真。具体操作方法可以参照光盘提供的 PDF 格式的文档 (VCS Simulator Summary 或 QuestaSim Simulator Summary)。

先不去管仿真结果,只需让仿真工具正常运行且没有报告错误就可以了。读者会在自己使用的仿真工具上观察到对应的波形显示结果,如图 1.6 至图 1.8 所示。

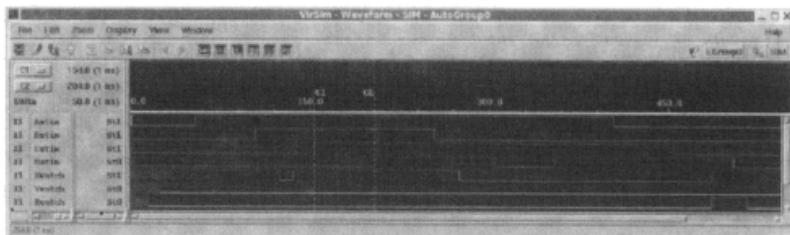


图 1.6 用 Synopsys 的仿真工具打开 Intro_Top 的仿真波形

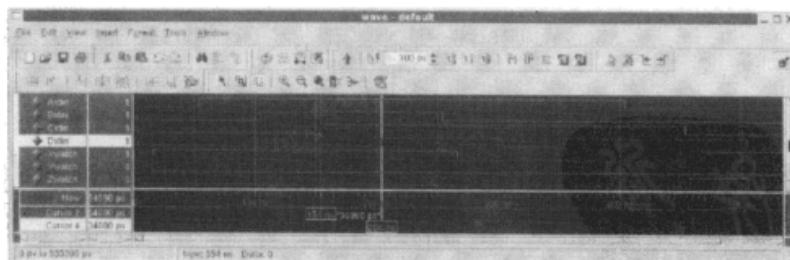


图 1.7 用 Mentor 的 QuestSim 仿真工具打开 Intro_Top 的仿真波形

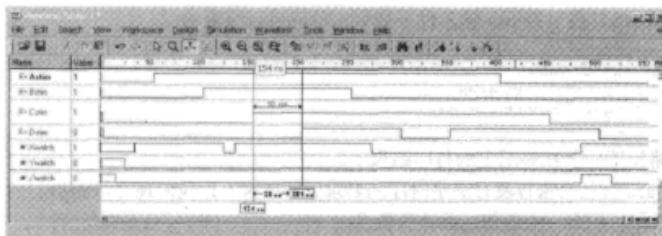


图 1.8 用 Aldec 的 Active-HDL 仿真工具打开 Intro_Top 的仿真波形

第5步。Design Compiler (DC) 综合工具可用来完成对数字电路的综合和优化，其文本对话接口 dc_shell-t 是其标准的接口形式。通常，需要添加很多 dc_shell 选项和嵌入到代码当中编译向导，经过多次调试后，从而最后可以产生满足时序和面积要求的网表。采用修改文本格式脚本的方式在综合优化过程中有很多好处，相反，图形化的综合操作在调试综合生成网表时会有效率低下、不易记录、不易复制和易于出错等缺点。

使用 DC 图形化界面的命令是 design_vision。本书里很少用到这种方式，我们只会用它来显示电路的原理图。本书使用的是 Tcl (Tool Control Language) 脚本接口形式，命令是 dc_shell-t，在使用 design_vision-t 时，-t 也是默认的选项。

由于 Intro_Top 模块基本上采用的是门级描述，因此没什么好综合的。但是连续赋值语句描述并不是门级描述，需要综合工具把它们综合为门级电路。

用下面的指令将设计模块载入到综合工具，且综合生成门级网表。

本书用到的 90 nm 工艺库源于 TSMC 的工厂。

综合指令

A. 打开综合工具

```
dc_shell-t -f Intro.Top.sct
```

屏幕先会出现一些文本信息，然后出现 dc_shell-t 提示符。在 dc_shell 提示符出现前将会运行一个包含有多条命令的一个.scr 文件 (Synopsys 脚本)。为了解释 Tcl 的语法，这里使用.sct 代替.scr。

在后面会详述这些信息，现在先回过头来看看 dc_shell-t 是从哪打开的。向前阅读确认有没有出错的提示，如果只是警告信息则可以忽略。

Intro.Top.sct 中的命令完成了如下工作：

- 指定综合工具使用的工艺和图形符号库；
- 把内存中的设计逻辑库与某一个目录相关联；
- 分析从磁盘中加载到库的设计；同时建立对应操作条件 (NCCOM = Nominal-Case, COMmercial temperature range) 和延时参数 (负载模型)。

完成上述操作后，操作命令设置了一些专门的约束条件 (如芯片的最大面积、最大时延、预计负载和驱动能力)，进入交互模式后停下来。当前设计为 Intro_Top。

B. 在 dc_shell 提示符下，输入以下命令，等待运行结束

```
dc_shell-xg-t> compile
dc_shell-xg-t> report_area
dc_shell-xg-t> report_timing
```

C. 粗略浏览一下在终端上的打印结果

负的 slack 信息意味着没有满足时序的要求，不过这不是我们现在需要担心的问题。对特定的库而言，芯片面积为近似的晶体管数量。

D. 将结果保存到磁盘

```
dc_shell-xg-t> write -hierarchy -format ddc
```

使用 write 命令将结果保存到 Synopsys 的.ddc 文件里。尽管这些二进制文件保存大设计时效率很高，但由于我们希望得到的是 Verilog 文件，所以，将结果写到 Verilog 的网表文件后退出：

```
dc_shell-xg-t> write -hierarchy -format verilog -output Intro.Netlist.v
dc_shell-xg-t> exit
```

E. 观察综合后的网表原理图

网表是以 Verilog 格式保存的。可以用文本编辑器打开查看。由于目前这个设计模块既小又简单，我们也可以将其转换成原理图后通过 Synopsys 的 GUI 界面来观察。在屏幕提示符下，键入如下指令

```
design_vision
```

GUI 出现后，使用 File/Read 命令读取 DC 刚生成的 Intro_Netlist.v 文件。然后，单击顶部菜单栏中间位置的“and gate”图标，可观察到电路的原理图。

DC 默认的是保留设计的层次结构，所以读者会注意到原理图和早期设计的原理图没有什么变化。

双击模块可以观察到其子结构，单击向上的箭头返回。

观察完原理图后，使用命令 design_vision File/Exit 退出。

F. 下一步，重新运行 DC shell 打平（flatten）网表的层次结构
打平通常会改善设计的时序和面积。

```
dc_shell-t -f Intro.Top.sct
```

当命令执行完毕之后，粗略地检查一下终端上的输出信息，确保没有错误，执行下一步。

```
dc_shell-xg-t> ungroup -all -flatten
dc_shell-xg-t> compile -map_effort high
dc_shell-xg-t> report_timing
```

现在时序已经得到改善，且已经实现了.sct中包含的所有约束条件。保存已经被综合的去掉层次的网表，但不要退出：

```
dc_shell-xg-t> write -hierarchy -format verilog -output Intro.TopFlat.v
```

在综合出的网表仍然被保存在内存里的时候，再做一件事，输出 SDF (Standard Delay Format) 文件。该文件的内容是综合工具利用工艺库计算出的延时信息。在后面的章节里，我们还会深入地学习这种文件格式。

该文件的后缀名为.sdf。

```
dc_shell-xg-t> write_sdf Intro.TopFlat.sdf
dc_shell-xg-t> exit
```

退出DC后，查看打平后和优化后的网表文件电路图。再次激活综合 GUI 界面，在 design_vision 中使用 File/Read 命令，打开新生成的 Intro.TopFlat.v 文件。

观察综合后的电路图，看起来应该和图 1.9 所示结果类似。网表里的每一个基本门电路都会根据物理库被映射成掩模块、布局并最终生产成为芯片。

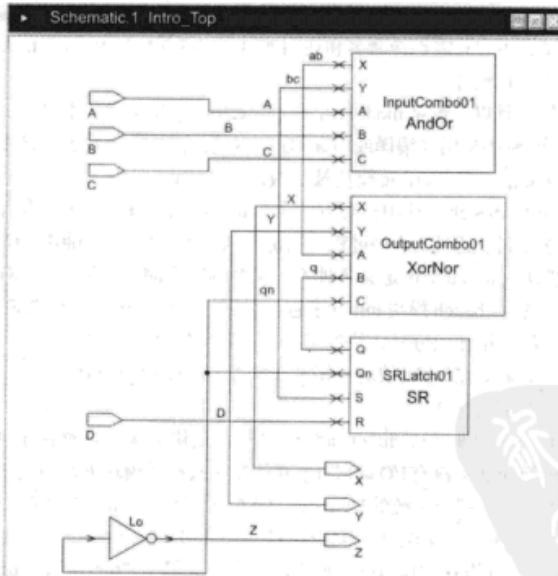


图 1.9 design_vision 画出的 Intro_Top 网表的结构层次图

观察完电路图后，单击 File/Exit 结束练习。

对术语“flatten”的解释

虽然“flatten”通常意味着去掉层次结构，但是在DC里，“flatten”有其特殊的含义。在DC里，flatten仅仅意味着打平一个逻辑表达式而不是打平设计。DC中去掉层次的命令是“ungroup”。

DC的用户可以用命令“set_flatten”来打平一个两层布尔型积之和组合逻辑的层次。采用“set_structure”命令完成逻辑层次的分解，它是“set_flatten”命令的反向操作。

1.1.1 练习后的思考

关键字。在练习1中使用到了Verilog语言的几个关键字：module, endmodule, assign, wire, reg。

Verilog的关键字使用的都是小写字母，而Verilog本身是对大小写敏感的。读者只需在命名的时候把至少一个字母改成大写就可以和关键字加以区分。例如，如下声明module Module…endmodule，尽管会给使用带来不便，但也是符合语法的。

注释。Verilog语言中，注释标识和C++注释标识相同：采用“//”进行行注释；“/*”和“*/”进行块注释。块注释允许对多行注释，注释标识也可以放在某行中间部分。

模块。在Verilog中，用户唯一可以声明的对象类型是模块(module)，其他对象类型已被预先定义和声明了，用户只需重新命名和使用即可。这既简化了Verilog语言，也使得编译器可以安全地对代码进行编译。

一个完整的模块是以关键字module开头，以endmodule结尾。用户在中间编写相应代码来定义模块的功能。模块是用来编译或仿真的最小对象类型，它等效于电路原理图中的设计模块。对综合工具来讲，一个Verilog模块被看做是一个“设计”。

initial块。上面testbench模块中包含有一个initial块。initial块中包含有可被仿真工具读取和执行的一个或多个过程声明，这些过程声明都是从0时刻仿真时间起执行的，而且只执行一次。在testbench模块的initial中可定义各种不同延时的时序测试变量，用来对设计的模块进行仿真测试。练习1的testbench模块initial中包含很多语句块。这些语句块被包在关键字begin和end之间。语句的结尾加上分号，块的结尾不需要分号。

为了终止仿真进程，在initial块的末尾调用任务\$finish结束仿真。否则，仿真将会一直运行（和仿真工具的配置有关）。

模块头。模块以模块头(module header)开始。模块头从模块命名开始到第一个分号结束。除了testbench模块通常没有I/O端口外，其他模块头都对模块的端口进行了声明。Verilog的模块头有两种格式，分别为传统的Verilog-1995格式和新一些的ANSI-C Verilog-2001格式。由于新的格式更加简洁且不易出错，因此本书只使用新的格式。

在模块头中，ANSI格式对端口的声明由端口方向关键字“output”、“input”或者“inout”开始。如果信号位宽超过1比特，则这些关键词后面还应该说明端口的位宽。端口排列的顺序没有具体的要求，但最好将输出放在前面（后面会解释原因）。方向关键字的后面是I/O端口

的名字，如果还有其他的同向端口，则端口名之间以逗号“,”隔开；再定义其他方向的关键字和相应的端口名字。

下面是一个对具有1个32比特输出、2个16比特输入和2个1比特输入的模块头的例子：

```
module ALU (output[31:0] Z, input[15:0] A, B, input Clock, Ena);
```

赋值声明。练习1包含两类不同的Verilog赋值声明：对wire变量的连续赋值和对reg变量的阻塞赋值（testbench的initial块中有该类赋值声明）。Verilog中，wire型变量只能用做连线，而不能进行过程赋值。而过程语句中的reg型变量值的变化方式不止一种。为了获取过程块中的reg的结果，可以用连续赋值将reg变量的值赋给wire变量或送至I/O端口，其过程如下例所述。

```
module And2 (output Z, input A, B);
reg Zreg;
// A connection statement:
assign #1 Z = Zreg; // Put the value of Zreg on the output port.
// Procedural statements:
initial
begin
    Zreg = 1'b0;
#2 Zreg = A && B;
#5 $finish;
end
endmodule
```

这段代码里的与（and）门实际是没有用的（initial块在时间 $2 + 5 = 7$ 的时候就结束了仿真），但是这段代码说明了寄存器和线型是怎样用在一起的。仿真开始前，Zreg被置为0；由于对Z的赋值存在延迟，在仿真时间1时，Z由不定态变为0；在仿真时间2时，Zreg的值随着输入端口的值的改变而发生改变；在仿真时间3时，Z的值也被更新。

testbench中过程阻塞式赋值用的是符号“=”。还有另外一种被称为非阻塞赋值的过程赋值，它的符号是“<=”。在后面的章节里，我们会学习到有关非阻塞赋值的内容。

简单的布尔操作符。Verilog中的运算符同C/C++中的运算符几乎一样。尤其是对于逻辑运算符，“!”，“&&”和“||”，它们的返回值是0或者1，分别代表假或者真。还有一种按位运算符“&”，“|”，“^”（异或）和“~”，它们是按位进行逻辑运算的。

四种逻辑电平。除了逻辑“1”和“0”之外，Verilog还定义了两个特殊的逻辑状态，分别是：“x”和“z”。其中，x表示信号取值要么为1要么为0，但是仿真工具无法判定信号值是1还是0；z代表“截止”（三态截止）。通常，除了多个驱动器连接在同一条线上的情形，它的含义与x相同。

明确指出每一个Verilog表达式的位宽是一种很好的习惯。即使位宽是1，我们也应该指明。所以逻辑值“0”通常被写为“1'b0”。规则是这样的：位宽在引号的前面，紧接着是数字的进制（‘b’，‘h’或者‘d’），最后是表达式的值。

因此，4'b000z说明这是一个4比特宽的数，它的LSB处于三态状态。12'b101和12'h5具有相同的值5，并且数据宽度都是12 bit。

延时。延时描述以“#”开始，数值只能用十进制来表示。延时值的单位由 timescale 宏来定义，这个宏通常出现在每个待编译源文件的开头。例如，当编译指示为`timescale 1 ns/100 ps，则 #1 相当于 1 ns 的延时，且仿真工具对所有延时计算的精度为 100 ps。

器件例化。模块的实例 (instance) 是完成一个大规模设计的基础。任何一个模块在被例化之前都必须先被声明。模块实例的概念有时和器件 (component) 实例的概念是等效的，尤其这个实例是一个小规模的器件库中的一个器件时。而例化 (instantiation) 就是声明实例的过程，以分号结束。

例化的基本语法如下：模块名 例化名 端口映射；例如在练习 1 中，待测试的设计在 testbench 中是这样被例化的：

```
Intro.Top Topper01 (.X(Xwatch), .Y(Ywatch), .Z(Zwatch),
                     .A(Astim), .B(Bstim), .C(Cstim), .D(Dstim)
                   );
```

被例化的模块是 Intro_Top；例化后的实例名为 Topper01，实例名后小括号里的内容是端口映射的信息。

端口映射。前面例化的例子说明了如何将 Intro_Top 的端口映射到 testbench 中的线型上：每一个端口名字前有一个点“.”，与这个端口相连的 wire 被一对圆括号括起来，紧接在端口名字的后面。

1.2 Verilog 矢量

Verilog 的矢量用于按比特排序的数据，总线或者是逻辑位。

在 Verilog 中，矢量和数组的结构十分相似，我们将在后面讨论数组。

声明后的矢量能够把它的值赋给另外一个矢量而不必经过显式的指明其比特区间。也可以只部分选择矢量中的某一位或者某几位赋值给其他变量或被其他变量赋值。而选择 (select) 操作就意味着明确地指出要被选取的比特。下述的表达式都是符合语法的：

```
reg[7:0] HighByte, LowByte1, LowByte2;
reg Bit;
...
HighByte = LowByte1; // Assigns one vector to another.
...
Bit      = LowByte2[7]; // This is called a "bit select".
...
LowByte2[3:0] = LowByte2[7:4]; // Two examples of "part select".
```

上述所有的赋值都是过程赋值，因为一个寄存器类型 (reg) 的变量只能被过程赋值。然而为了简化起见，我们省略了模块声明和包含过程声明的块。

值得注意的是 Verilog 与 C 语言声明之间的差别：Verilog 中的序列区间是在类型名 (reg 或者 wire) 之后，而不是在被声明的变量名之后！

Register的意思是“regular line-up”，数据被寄存是寄存器传输级（Register-Transfer Level, RTL）里的一个基本的特征，而RTL设计是数字仿真和综合中最常用的抽象级层次设计。

不要把reg直接理解成寄存器！把它理解成晶体管要好一些，但仍然是不准确的。reg声明代表对信息的存储，即使它被声明为一个比特时也是如此。在Verilog中，reg变量表示一个逻辑状态；wire绝对不能被当做是reg，即使在总线的应用中它有时会被上拉。wire只起到不同位置逻辑连接的作用，它不能够存储信息。wire和reg都被叫做变量的原因是，在仿真中它们的值能够被事件所改变。

只有当驱动wire的逻辑值发生改变时，wire的值才会改变，而reg变量的值能够在initial或者always块中通过赋值来改变。对一个reg变量的赋值被称做过程赋值，通过在一个initial或always过程块中单个的过程语句来实现对reg的赋值。

连续赋值通过连续赋值语句（关键字：assign）来实现，代表和连线永久性的连接。例如：

```
module ModuleName(...);
  wire x;
  ...
  assign x = a & b | c; // LHS must be wire; RHS wire or reg.
endmodule
```

在练习1中，Intro_Top设计中所有的赋值都是对wire的连续赋值，由于没有reg型变量（testbench除外），第一次练习中没有用到过程赋值。

过程赋值在过程块(always或initial关键字)中进行，代表了按某种顺序发生的逻辑变化。例如：

```
module ModuleName(..., input a, b, c);
  reg x;
  wire y;
  //
  assign y = a & b;
  assign y = y | c; // y wired to two drivers probably is an error!
  ...
  always @{a,b,c} // LHS must be reg; RHS may be wire or reg.
    begin
      x = a & b;
      x = x | c; // x gets c | (a & b). No problem.
    end
endmodule
```

在任意赋值语句中，常量可以使用二进制、八进制、十进制和十六进制的表达式。

1'b1, 1'b0, 1'bx, 1'bz, 8'b00zz_xx11, 64'h33b4_1223_1112_af01.

8'b1010_0010 is the same as 8'ha2 or 8'hA2 or 8'o242 or 8'd162.

在逻辑仿真中，显示表达式的限制往往变得不安全。这是由于十六进制、十进制或者八进制的表达式所采用的值容易使人混淆。如果有一个十六进制的z或者x，那么所有的4位都被认为是z或者x。十进制和八进制也是同样如此。例如，将8'b000z_xx11赋给一个十六进制变量则显示为8'hzx。由于x的强度比z更高，所以4'b0z0x会被显示成4'hx。

诸如#10这样的时间表达式被逻辑综合工具所忽略，且综合工具以一种特殊的方式将未知的值看成是已知的值。对综合工具而言，无论是x还是z，都和确定的值一样，没有实际的意

义。对仿真来讲，x 表示仿真工具不能确认到底是 1 还是 0；z 代表三态门的功能，而不是一个逻辑状态。不同的综合工具各自都有对 x 和 z 处理的方法，目的是让综合前后的设计的仿真结果一致（时序信息除外）。

矢量中的各位被定义后就不能随意改变顺序。例如，把 8'ha2 赋予一个 8 bit 的变量 reg[7:0]，则比特[7]就是 MSB，其值为 1；如果 8'ha2 被赋给 reg[0:7]，则比特[0]就是 MSB，其值为 1。MSB 总是位于 Verilog 矢量区间的最左边。

```
// Some example vector operations:  
'timescale 1ns/100ps  
module Vector;  
    reg [0:15] MyBus; // A vector of 16 bits of storage.  
                      // The verilog MSB always is on the left.  
    //  
    wire[15:0] mybus; // Another vector, a 16-bit bus.  
    ...  
    MyBus[0:2] = mybus[10:8]; // This is called a "part select".  
    MyBus = mybus;  
    MyBus[0:15] = mybus[15:0];  
    ...  
endmodule
```

'timescale 的延时表达式允许使用小数，例如：#0.1。下面是例子：

```
'timescale 1ns/10 ps  
...  
#0.05 X = Y & Z; // The delay is 50.ps.
```

1.3 练习 2：操作数

在目录 Lab02 里完成本次练习。

练习步骤

第 1 步。矢量练习

- 新建一个名为 Vector.v 的文件，然后把上一节课程中那个模块声明的例子复制到这个文件当中，在你认为合适的地方添上缺少的部分（“...”）。用连续赋值来驱动 mybus，在一个 initial 过程块中使用阻塞式赋值来完成课程例子中的赋值。自行定义时间延迟，用 VCS 仿真来检查你做的对不对。
- 现在尝试颠倒总线的比特顺序。添加如下语句：MyBus[0:2] = mybus[8:10] 或者 MyBus[15:0] = mybus[15:0]。然后再仿真，会出现什么情况？
- 尝试在 initial 块中使用 mybus = MyBus，看会出现什么问题？
- 声明一个位宽为 48 比特，名为 My48bits 的 reg 型变量，在 initial 过程块中这样对它进行赋值：#5 My48bits = 'bz; #5 My48bits = 'bx; #5 My48bits = 'b0; #5 My48bits = 'b1；注意在上述表达式中并没有指明变量的位宽。添加 #5 \$finish，然后仿真，会出现什么情况？如果使用 1'bz, 1'bx，又会出现什么情况？

第2步。布尔操作练习

- A. 声明三个 reg[4:0]的矢量，分别命名为 X, A 和 B。仿真如下的赋值情况：初始化 $A = 5'b01010$; $B = 5'b11100$; 初始化之后，顺序的执行： $X = A \& B$; $X = A \mid B$; $X = A \wedge B$; $X = \sim A \mid \sim B$ 。
- B. 接着执行， $X[0] = \&A$; $X[1] = \mid A$; $X[2] = \&B$; $X[3] = \wedge A \& \sim \wedge B$; $X[4] = A[4] \wedge B[4]$;
- C. 使用圆括号把操作数分组： $X = ((\sim A \& \wedge B) \mid (A \& B)) \wedge A$;

这些操作数也可以用于宽度不相等的矢量，我们将在后面讨论。

1.3.1 练习后的思考

值得注意的问题：

对逻辑 0, x, z 的矢量扩展（没有对 1 的扩展）；

二进制运算符与缩位（Reduction）运算符的对比；

如果时间精度是 `timescale 1 ns/1 ns，使用延时语句 #0.1 会出现什么结果？

1.4 小结

1.4.1 VCD 文件的保存

逻辑仿真工具可以生成VCD(Value-Change Dump)文件。VCD文件的格式是由IEEE Std. 1364的18节规定的。简言之，该文件采用一个ASCII文件记录仿真信息，是一种非常紧凑的格式。由于采用了ASCII文本格式，因此工程师可以用自己编写的脚本或者其他工具所打开并处理它。例如，可以利用它来统计仿真的覆盖率，或检测仿真时时序的违例。

VCD文件由一个头文件和一个表组成，头文件包含设计模块名称之类的信息，表中是变量名字的缩写。端口上的信息可以有选择地被保存，这和处理模块中变量的方法是一样的。变量名由一个ASCII符号表示，例如“!”、“#”或者“(”。VCD文件的主体是仿真时间的列表和紧接着在那个时刻后面的变量值构成。仿真时间的格式为#time (绝对时间，而不是延时这种相对时间)。也正是因为这个原因，仿真工具输出的这一类型的文件通常被叫做“时间-数值”(time-value)文件。例如：

```
...
$scope module TestBench $end
...
$var reg 1 # Cstim $end
$var reg 1 $ Dsttim $end
$var wire 1 % Xwatch $end
...
#30
0#
#40
0$
...
```

被保存的变量值可以采用 4 值逻辑（即 4 种状态）或者是逻辑值加上强度来表达。由于 VCD 文件格式是一种 IEEE 的标准，因此这种文件格式能够被很多工具所支持。VCD 文件的一个重要用处就是可以用它来评估设计的动态功耗 (dynamic power dissipation)。这个问题超出了本书讨论的范围，因此不再详述。

1.4.2 综合的重要性

有人也许会认为对于前端 (front-end) 设计工程师来说，掌握好仿真的技能就够了。其实这个观点很不正确。虽然仿真工具可以验证设计的源文件，也可以验证综合后的网表，但是在使用源文件进行仿真时——仿真用到的时序信息是设计者基于估计和期望值来手动输入的。只有综合出了网表以后，门电路的传输延时，建立时间、保持时间，其他时钟约束和反标 (back-annotated) 的路径延迟才是准确的。

更重要的是，有了网表才能制造出芯片，因此它具有市场价值。而仿真波形不能被产品化或者被销售，所以只有对设计工程师来说，仿真波形才是有意义的。一个好的网表是工程师为公司所做的贡献，而网表几乎都是利用逻辑综合工具综合出来的。

1.4.3 SDF 文件的保存

SDF 文件可以被用来反标网表的延时信息。逻辑综合工具能够生成这种类型的文件，但通常布局布线工具产生的 SDF 文件里的延时信息更准确。SDF (Standard Delay File, 标准延时文件) 和 lisp、EDIF 类似，它用到的符号几乎都是圆括号。

例如，一个缓冲器的 SDF 信息如下：

```
...
(CELL
  (CELLTYPE "BUFFD0")
  (INSTANCE U3)
  (DELAY
    (ABSOLUTE
      (IOPATH I Z (0.064:0.064:0.064) (0.074:0.074:0.074))
    )
  )
)
...
(INTERCONNECT U3/Z U4/A (0.002:0.002:0.002) (0.003:0.003:0.003)
...
```

SDF 文件中包含一个网表所需的全部时序信息，其中延时信息可以通过静态时序分析从库中提取，也可以通过其他的手段提取出来。在后面的课程里，还会继续学习有关这个文件的知识。我们将会学习 SDF 里的时序信息是怎样反标至网表里的。

1.4.4 补充学习

阅读 Thomas and Moorby (2002) 第 1 章关于 Verilog 基础知识的内容。

阅读 Thomas and Moorby (2002) 第 3 章可综合的 Verilog 的 3.4 节 (3.4.4 节关于 casez 和 casex 可以不看)。

做 Thomas and Moorby (2002) 1.6 节的练习 1.2。

在 Lab01 目录下找到.VCD 文件并在文本编辑器中打开，它是在做 Lab01 仿真时生成的，这是因为在 Lab01 中的 testbench 里包含了这样一条指令 (directive)：“include “..../VCS/Extras.inc””。观察这个 VCD 文件，注意它的特点。本书中我们不会涉及到 VCD 文件的使用，但了解它却是大有好处的。Bhasker (2005) 的 10.10 节中有关于它的详述。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003)^①。通读第 1 章到第 4 章的练习，可以尝试做一些练习。

学习 4.1 节中的 S-R 锁存器模型，在 Palnitkar 附带的光盘中有一个可用的例子，它位于 Chap04/Chap_EG/SR_Latch.v。后续练习中将会更多地用到 S-R 锁存器。

学习 2.6 节中的行波进位 (ripple-carry) 计数器的例子。附带的光盘里有源代码，它位于 Chap02/Chap_EG/ripple.v。在后续的课程里，我们还会学习到更多关于计数器的知识。

阅读 9.5.6 节，对 VCD 文件有一个大致的了解；阅读 10.4 节，获得对 SDF 文件大致的了解；我们将在本书的最后一章完成一个关于 SDF 文件格式的练习。

^① 本书已由电子工业出版社引进出版。其中文书名为《Verilog HDL 数字设计与综合》(第二版) (ISBN 978-7-121-08947-3) ——编者注。

第2章 Verilog 基础知识 1

2.1 更多的语言结构

我们现在来更深入地学习 Verilog 这门语言，这样大家才能够做出一些有价值的设计。同时，我们也会加强对一些基本结构的理解，这些结构将会在后续的课程中频繁用到。我们还会学习移位寄存器的基本概念并在实验中设计一个移位寄存器。

传统的模块头格式，在本书所有的练习里，我们都不会使用传统格式。不仅如此，作者建议读者在所有的新设计里，都不采用传统格式。但是，一些工具（包括一些老的教科书）仍然采用传统格式来书写模块头，所以还是有必要学习 Verilog-1995 格式的模块头。

传统格式遵循 K&R C 规范，而新的格式遵循的是 ANSI C 规范。在这两个格式里，每个端口的名字都被自动地关联到一个同名的连线上，所有端口的默认属性都是 wire 类型。

二者的主要区别在于，在 Verilog-2001 里，模块头是模块的第一条语句，头的声明是头的一部分。Verilog-2001 里的第一个分号指明了头的结尾。

在 Verilog-1995 里，只有端口的名字在第一个声明中（第一对圆括号中）；位宽和端口方向的声明可以位于模块主体的任何一个地方。

从二者的区别我们可以发现，按照传统的格式，每个端口的名字都至少要被输入两遍：在第一对圆括号里面输入第一次，在声明为宽度和端口方向时输入第二次。在一个规模较小的模块中，这样做还不算太麻烦；但如果一个模块有成百上千的输入输出端口，出错的可能就大大增加了。

在传统的格式里，由于每个输出端口总需要输入两次，若要对它进行过程赋值的话，这个输出端口还需要被输入第三次——在第三次中把它声明成 reg 类型。把输出端口声明成 reg 类型，省掉了用连续赋值再把 reg 的值送至输出端口的工作。在我们的工作中，一般都使用这种方法。

前面描述的看似高效的方法会带来三个问题：第一个问题是现在对同一个名字有了三个不同的声明，这就容易使得设计者感到疑惑，因为设计者的出发点是对于不同对象的每一个声明都应该拥有不同的名字。第二个问题，这样做使得代码维护容易出错，因为一旦设计中的某些 I/O 端口有所改动，那么这三条不同的语句就必须被同时修改。第三个问题，这个规则要求我们区别对待不同的输出端口——在圆括号里，仅从它们的命名是看不出区别的。但在后面被声明为 reg 变量的端口要采用过程赋值语句在 always 块中实现赋值，没有被声明为 reg 变量的端口只能被诸如连续赋值语句之类的语句来实现赋值。

同时，按照传统的格式，由于没有规定模块头应该处在模块里的一个什么位置。原则上，模块头从最开始，直到在模块内的某处完成了最后一个声明的 I/O，确定了它的位宽和方向之后，才是模块头的结束。

上述所有的问题在Verilog-2001中得到了解决。所有I/O端口的声明只能被放在模块头中，并且只能被定义一次。

传统的格式当中还存在两个设计管理方面的问题：

首先，在Verilog-1995版本中，修改模块内容可以影响到模块头的定义，尤其是I/O的位宽，因此在项目开始的阶段不能够把明确模块头的定义完全确定下来。在Verilog-2001版本中就不存在这样的问题了，可以很方便地确定模块头的定义并且可以要求设计人员在没有得到管理许可的情况下不得擅自修改模块头。

其次，在Verilog-1995版本中，在声明了参数（被命名的配置常数，详见下文）之后，它的用法是模糊不清的。比如：哪些参数可以在模块例化时被重写？哪些参数必须被严格限制到模块的内部使用？而对于Verilog-2001这个版本来说，尽管没有在标题中被声明的参数仍然可以在例化的时候被重写。但是可以明确要求设计小组在写代码时只能重写标题中的参数。

下面用两个模块声明的例子来说明不同格式中模块头的差异，为了简明起见，模块的功能被忽略了。Verilog-1995版本的模块头格式为：

```
module MyCounter
  (CountOut, CountReady, StartCount, Load, Clock, Reset);
  output [15:0] CountOut;
  output      CountReady;
  input  [15:0] StartCount;
  input       Load, Clock, Reset;
  reg [15:0]   CountOut; // Could be anywhere in the module.
...
endmodule
```

Verilog-1995版本中模块头没有结束的标志，相当于它永远也没有结尾。

Verilog-2001中模块头的格式如下：

```
module MyCounter
  (output [15:0] CountOut, output CountReady
   , input [15:0] StartCount, input Load, Clock, Reset);
// Header has ended; module itself now begins:
  reg [15:0] CountOutR;
  assign CountOut = CountOutR;
...
endmodule
```

二者的另一个差异是，在Verilog-2001中对CountOut的连续赋值可以允许有多个不同的上升和下降延时值。而过程赋值只能有一个延时值。也就是说，在Verilog-1995旧版本中直接将CountOut声明为reg类型就意味着它只能有一个上升和下降延时值。几种不同的Verilog延时赋值的方法将在后面讨论。

在Verilog-2001中，仍然可以将一个输出端口声明成一个reg类型：

```
module My2001Module (output reg [15:0] CountOut, ...);
```

然而，在人工手动完成的设计里应该避免这种用法。尽管声明reg类型的输出端口只会带来很小一点的不便，但在我们的课程中不会使用这种用法。

注意：input和inout端口不能被声明为reg变量，因为一个input端口若被声明为reg类型，只能在过程块里通过过程赋值来改变它的值。但模块头里显然不会有过程块。因此，一个reg

类型的 input 端口永远都不会被初始化。在例化这个模块时，它会和连在这个 input 端口上的信号发生冲突，并在那个 input 端口上产生一个持续的（也是不合法的）x。

Verilog 注释。与 C++ 类似，注释有两种不同的格式：

- 单行注释 (Line Comment)：“//”可以在一行的任意位置开始
- 块注释 (Block Comment)：注释以 “/*” 开始，直到 “*/” 结束

注释的例子：

```
assign X = a && b; // Put the AND of a and b on X.
//
assign Y = a /*left operand*/ && b /*right operand*/;
//
assign Z = (a>5)? a&b : a||b;
/* The preceding assignment puts the AND of a and b
on Z if a is greater than 5; otherwise, it puts
the OR of a and b on Z. */
```

未定义的宏可能会影响到注释。例如 `ifdef undefined_name 中的 undefined_name 宏变量如果没有被预先定义过，这会使 `endif 之前所有的内容都被 Verilog 编译器（仿真器或者综合器）所忽略。因此，一个未定义的宏能够有效地用来注释部分代码。设计工程师可以通过 `define 这个宏变量来恢复被注释掉的代码，重新把它们加入到将待编译的代码中。下面会详细叙述这个问题。

注释也能够被用来增加适用于特定工具的非 Verilog 语言的约束条件：行注释中可以包含针对专用工具的关键词，因此综合的指示或者其他与 Verilog 代码结构有关的特性可以被嵌入到行注释中。工具会检查所有的注释，去寻找这些关键词。下面是这类注释的一个典型例子：“//rtl_synthesis …”，在 rtl_synthesis 关键词的后面会紧接着一个命令或者是一些约束条件。

Always 块。在仿真时候，一个 always 块中的敏感变量表 (sensitivity list)（事件控制表达式）中的值如果发生了改变，这个块中的语句会被重新读取或者重新执行。逻辑综合工具通常会对每个 always 块进行逻辑综合。如果 always 块中敏感变量里漏掉了某个变量，这意味着当这个变量发生变化时，always 块也不会被重新执行。这可能会导致该模块综合后出现锁存器或其他电路。我们将在本书的后面详细讨论这个问题。

Thomas and Moorby(2002)里有一些没有敏感变量表的 always 块的例子。这种用法并没有错，但在本章中，为了保证我们的设计风格，应避免使用这种方式。在设计实践中，尽量不要省略事件控制条件（敏感变量表），因为把事件控制条件放在每个 always 块的最前头可以使人们很容易理解这个块的功能。在本章中，我们不推荐使用不包含敏感变量的 always 块。应该在 always 的后面紧跟着事件控制的条件（“always @(variable_list)”）。

```
...
always                                // Avoid this style.
#10 ClockIn <= !ClockIn;
...
always@(ClockIn)                      // Recommended style.
#10 ClockIn <= !ClockIn;
```

initial 块。**initial** 块没有敏感变量表。因为 **initial** 块的代码只在仿真 0 时刻之前才会被读取一次，在那个时候还不能控制事件的发生。所有的 **initial** 块都会被综合工具所忽略。由于 **initial** 块只能按照与仿真 0 时刻的时间差来规划事件的先后顺序，而不能并发地被激活或者被中止。因此在仿真的代码里，最好只使用一个 **initial** 块。如果使用多个 **initial** 块，则各种事件之间的先后顺序就很难判断了。

这个规则也是有例外的。如果需要第二个 **initial** 块来产生一个仿真的时钟（使用 **forever**），或者是产生一个与设计功能不相关的行为，例如定义一个 SDF 文件读取操作或者是创建一个 \$monitor 任务等，则还是可以有其他的 **initial** 块。

连续赋值。为了内容的完整性，我们在这里再次讲到了连续赋值。Verilog 中最常用的三种并发执行的块结构分别是 **always** 块、**initial** 块和连续赋值语句，除这三种块之外还包括结构（分层次）设计实例。用 Verilog 设计实现这三种块和实例就是设计工程师的主要工作。连续赋值对等式右边任何值的变化敏感。举例如下。

一个 **wire** 赋值（连续赋值）对右边的表达式敏感：

```
assign #2 X = A[0] && B[0] || !Clock;
```

过程赋值仅对它的事件控制列表列出的信号敏感。

例如，**Clock** 的变化不会引起 **Xreg** 值发生变化：

```
always# (A[0], B[0]) Xreg = A[0] && B[0] || !Clock;
```

仅当 **Clock** 变为 1`b1 的时候，**Yreg** 的值才会更新：

```
always# (posedge Clock) Yreg = a && b;
```

矢量和矢量值。所有的矢量都被认为表达的是具体的数值。大多数的矢量类型（**reg** 或者 **net**）都被默认当做无符号数。**integer** 和 **real** 是个例外，它们被默认当做有符号数。通常，**real** 类型是不可综合的。

假设在没有溢出的情况下，不管是无符号数还是有符号数，它们都是二进制的一串数值而已；而当这个值被当做某种类型做比较时：有符号数的 MSB 被用来表示这个数字的符号 [1 代表为负数，规则和 2 的补码 (2's complement) 一样]，而无符号数的 MSB 则是位权最高的那一位。无论采用什么样的二进制格式，一个无符号数永远也不能为负值。

下面是一个和符号位有关的例子：

```
reg[31:0] A;
integer I;
...
A = -1; // Default is to assume decimal integers.
I = -1; // Now, both A and I hold 32'hffff.ffff.
//
if ( I > 32'h0 )
    $display("I is positive.");
else $display("I is not positive."); //Prints "I is not positive."
if ( A > 32'h0 )
    $display("A is positive.");
else $display("A is not positive."); //Prints "A is positive."
```

对于 Verilog 逻辑运算符 “!”， “**&&**”，和 “**||**” 而言，它们的操作数矢量如果所有位都为 0 则被视为 false；只要有一个比特不为 0，就视为 true。

Verilog 位运算符 (bit wise) “**&**”，“**!**” 和 “**^**” 是按位进行逻辑运算的。一元位运算符 ~ 如果在写矢量前，作用是对矢量中的每一位取反。一元位运算符还可以起到缩减运算符 (reduction operator) 的作用。例如，**&A** 的意思是对矢量 A 中的所有位逐位相与。

没有被命名的常数被称为 literal。一个 literal 表达式包含：一个位宽说明，一个界定符 (', delimiting tick symbol)，一个进制说明，最后加一个表示数值的数字，表示为位宽'进制(数值)。例如，**5'h1d** 定义了一个 5 bit 宽的矢量 literal，其十六进制的值为 1d (相当于十进制的 29)，同样 **16'h1d** 同样等于十进制的 29，只不过比刚才的位宽要更宽一些。

如果位宽和进制被省略，那么这个 literal 的值被默认为是十进制的，它会被当做是一个 32 比特的有符号整数。

矢量类型转换没有专门的转换运算符，只需很简单的步骤就可以实现。Verilog 实际上没有用户自定义的类型，因此这个规则十分简单：

1. 矢量每一位的位权已被预先定义好，MSB 总是位于最左边。
2. 表达式中所有的值都按最右边的一位 (LSB) 对齐。
3. 赋值语句右边的表达式的位宽在运算符开始工作之前都会被自动调整为目的操作数的位度。
4. 位宽的调整有两种类型，如果原值位数过长则扔掉超出的位数，如果原值位数不够则高位填 0。如果操作数是有符号数，则按原来的 MSB (符号位) 填充。

矢量操作的类型转换如下所示：

```
reg X;
reg[3:0] A, B, Z;
...
X = A && B; // X gets 1'b0 only if either A or B is 4'b0.
Z = A && B; // Z gets 4'b0 only if either A or B is 4'b0.
           // Otherwise, Z gets 4'b0001
Z = A & B; // Each bit of Z get the and of the
           // corresponding bits of A and B.
X = !Z;    // X gets 1'b1 only if all bits of Z are 0.
X = ~Z;    // Z narrows to its LSB and is inverted; X gets !Z[0].
```

截断 (truncation) 和扩展 (widening) 操作的例子如下 (目前暂时只涉及无符号数)。我们看到，一个变量在被声明的时候也可以被初始化，但是这样做仅仅为了仿真：

```
reg      A = 1'b1, B = 1'b0;
reg[3:0] C = 4'b1001, D = 4'b1100;
reg[15:0] E = 16'hffff;
...
C = A | B; // C gets A=4'b0001 | B=4'b0000 --> 4'b0001.
C = E & D; // C gets E=4'b1110 & D=4'b1100 --> 4'b1100.
E = A + C; // E gets A=16'h1 + C=16'h9 --> 16'ha.
```

Verilog 还包括参数声明。参数相当于被命名的常数，声明参数的时候必须给这个参数赋一个值。如果在模块头中声明了参数，那么这个被声明的值就是这个参数的默认值。在这个模块被例化的时候，这个参数值可以被重置。

在模块内声明参数：

```
module ModuleName ( ... I/O's ... ); // <-- semicolon ends header.
...
parameter Awid = 32, Bwid = 16;
...
reg[Awid-1:0] Abus, Zbus;
wire[Bwid-1:0] Bwire;
...
endmodule
```

在模块头声明参数：

```
module ModuleName
#(parameter Awid = 32, Bwid = 16) // <-- no comma!
( ... I/O's ... ); // <-- semicolon ends header.
...
reg[Awid-1:0] Abus, Zbus;
wire[Bwid-1:0] Bwire;
...
endmodule
```

参数默认是无符号数，参数的位宽默认为刚好容纳下这个参数值的位宽。

用宏进行条件注释。采用 Verilog 块注释符 /* 和 */ 注释掉不需要的代码。

为了避免不可综合的代码导致综合工具报错，这种注释十分有用。但通常一个更好的方法是综合工具在读取一个 Verilog 文件之前，先定义一个全局宏 DC。这个宏是空的，但是它必须被声明。这样，我们就可以采用 `ifndef DC（如果 DC 没有被定义）和 `endif 来注释掉那些在综合的时候会出问题的代码行，使得它们对综合工具不可见但并不影响仿真。

例如，由于综合工具不能综合 Verilog 系统任务。所以我们采用 DC 宏来使得 Verilog 系统任务声称对综合工具不可见而对仿真编译器可见：

```
always@(posedge Clk)
begin
Xreg = NewValue;
`ifndef DC
$display("time=%05d: Xreg=[%h]", $time, Xreg);
`endif
Yreg = ...
end
```

参考书中提到的 Silos 仿真器不识别 `ifndef，但是识别 `ifdef 和 `else；因此上面这个例子的更通用的写法如下：

```
always@(posedge Clk)
begin
Xreg = NewValue;
`ifdef DC
`else
$display("time=%05d: Xreg=[%h]", $time, Xreg);
`endif
...
...
```

尽可能地少用`define或其他全局宏定义是一个好的编码风格。在进行大规模的设计时,很有可能由于疏忽使得不同的宏定义了相同的名字。这样,在编译的时候,有可能会发生旧值被新值替换的错误,从而导致设计出错。

为了避免出现这种由于编译顺序引起的错误,一个被定义的宏在发挥完它的作用后应该立即去除对该宏的定义,并且最好是在同一个文件中取消。`timescale是一个例外。下面是一个宏的例子:

```
'define BLOCKING
#define Awid 32
module ModuleName ( ... I/O's ... );
  ...
  reg['Awid-1:0] Abus, Zbus; // The ' is required for value!
  ...
  always@( ... )
    begin
      `ifdef BLOCKING
        Qreg = Areg | Breg;
      `else
        Qreg <= Areg | Breg;
      `endif
    end
  endmodule
`undef BLOCKING // Use 'undef unless you want these to
`undef Awid    // be seen in subsequently compiled files.
```

2.2 练习 3: 参数和转换

在目录 Lab03 里完成以下练习。

练习步骤

第 1 步。综合一个参数化位宽的设计。目录 Lab03 中的设计包括一个顶层的 Verilog 模块 ParamCounterTop.v 和两个子模块 Counter.v、Converter.v。图 2.1 给出了电路图,只有在端口映射时名称发生了改变的内部端口才被专门标注出来。如端口 OutEnable 和引脚 Enable 相连。

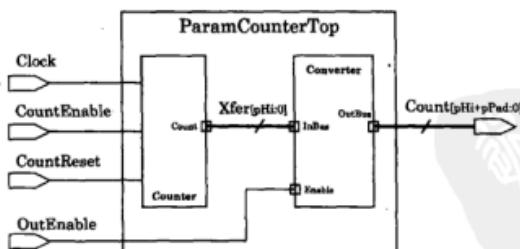


图 2.1 Lab03 的电路图 (第 1 步)

这是一个可以被复位的向上计数器,计数的结果输出到位宽可配置总线的低位上。这个设计的目的是为了说明如何给一个设计配置参数。在这个练习中只有总线的位宽是参数。

在做这个练习的时候不用考虑仿真需要的东西。可以在综合之前使用 DC 宏注释掉整个 testbench 模块。

一切就绪之后，把 PADWIDTH 的值改为 8，然后进行综合。先按面积进行优化，然后按速度进行优化。在针对面积进行优化时，只需在综合的时候注释掉所有的与速度相关的约束条件即可（但是要保留设计规则）；在针对速度进行优化时则需要在综合的时候保留那些与速度相关的约束条件，但是综合的目标是面积尽可能小。实际设计中面积和速度往往是相互关联的。

在完成将 PADWIDTH 的值设成 8 的练习之后，把 pPad 值改大。再次进行综合，看会有什么结果。

第2步。创建一个用来仿真算术运算的设计，然后尝试做一些对无符号数和有符号数的操作，观察仿真结果。

新建一个文件 Integers.v，在其中创建一个 Verilog 模块，然后检查下述语句的执行结果（如果想在一个 initial 块中同时执行它们，也可以把这些变量的名字改为不同）：

- A. integer A = 16, B = -8, X; X = A + B; X = A - B; X = B - A;
- B. 把所有的变量声明为 reg[31:0]（使用 32'd 进行赋值）后重复上述语句；
- C. (选做) 进行如下改变后重复执行上述语句：只把 X 声明为 integer；只把 A 声明为 integer；只把 B 声明为 integer；

第3步。为了观察对不同类型的变量截断和扩展的效果，新建一个文件 Truncate.v，在其中创建一个 Verilog 模块，完成下面的步骤，定义以下几种数据对象，integer Int；reg[7:0] Byte；reg[31:0] Word；reg[63:0] Long；reg[127:0] Dlong。把每一个变量都初始化为 'b0。然后：

- A. 给上述所有变量赋值 6'd1，然后赋值 -6'd1，观察会出现什么情况，分别用十进制和十六进制来显示；
- B. 先赋值 36'd1，再赋值 -36'd1；
- C. 分别对 Int 赋值 1 和 -1，然后分别使用这两个不同值的 Int 对其他变量像 A、B 那样赋值；
- D. 对 Word 赋值 32'h7eee_777f，然后用它对其他变量进行如上的赋值。这个值的 MSB 为 0，因此它代表一个正数；
- E. 把 32'h777_eee7 赋值给 Word，然后用它对其他变量进行如上的赋值。这个值的 MSB 为 1，因此它是一个负数。

2.2.1 练习后的思考

注意：虽然本书中没有用到，但负的矢量边界是可以有的。矢量的宽度总是等于两边边界的差值加 1。

例如下面的声明：

```
reg[3:-7] CoeffA;
```

它会被工具识别成一个 11 bit 的十进制小数， $LSB = 2^{-7} = 1/2^7 = 1/128$ 。在进行 DSP 建模时，会用到这样的数据类型。

2.3 过程控制

2.3.1 Verilog 中的过程控制

在这一节里，我们将会学习三种过程控制结构：if, case 和 for。这三种结构只能用在进程块（initial 块或 always 块）中。

if 语句用于比较条件的优先级或者数值的范围。当 if 被用来描述是时钟或异步控制事件之类具有互斥特征的对象时，由于只需要输入几个条件即可。因此，使用 if 要比其他控制结构简单得多。

```
if (expr) statement1;
else if (expr) statement2;
else statement3;
```

case 用于选择一系列可列举出来的值，通常这些值没有优先级的区分。例如，实现一个查找表或者一个小规模的存储地址电路。Verilog 中的 case 会自动退出选择逻辑，不像 C 语言中 switch 后的 case 那样还会继续往下顺序执行。永远都不要省略 default 语句是一个好习惯。我们在后面还会继续探讨这个问题。

```
case (expr)
    alt1: statement1;
    alt2: statement2;
    ...
    default: default_stmt;
endcase;
```

供 case 语句选择的分支选项一般是常量表达式，但也可以是变量；而 case 表达式通常是变量。

for 语句是 Verilog 中常用的循环结构；它工作的方式同 C 语言中的 for 语句相同。但是，C 语言中的一元递增和递减表达式却是不允许的，例如 i++，++i，i-- 和 --i，在 Verilog 中只能使用 i = i + 1 和 i = i - 1 来实现。

```
for (loop var init; loop reentry expression; loop var update) statement (s);
```

2.3.2 组合逻辑与时序逻辑

- 条件表达式运算符，control_expr?True_expr : False_expr。它的用法就好像是在使用一个函数，表达式的结果是返回值。条件表达式运算符是一个表达式而不是一条语句。它会被综合工具综合成组合逻辑。如果 control_expr 的值为真（非 0），那么这个表达式的值就等于 True_expr，否则等于 False_expr。例如：

```
wire[31:0] X;
integer A, B;
...
// Put the greater of A or B into X; A if they are equal:
assign #2 X = (A>=B)? A : B;
```

由于if不能用在连续赋值语句中(只能用在过程块中),如果要把选通的结果连在线型上,条件表达式运算符就显得非常有用了。

- 简单的 always@ 块语法: 在事件控制(敏感变量表)中使用“,”而不是Verilog-1995版本中的“or”。如果敏感变量对电平的变化敏感,那么它就是组合逻辑。如果过程块仅对时钟的上升沿或者下降沿敏感,那么它就是时序逻辑。

always@ (posedge Clk, posedge Reset) 与旧版本的 always@ (posedge Clk or posedge Reset) 是等效的。

把事件控制嵌入到 always 块中也是可以的:

```
always@(posedge clk)
begin
    xbus[1] <= 1'b1;
    @(posedge Enable) // Execution stops here until Enable goes to '1'
        begin
            Dbus      <= 8'haa;
            xbus[7:4] <= 'b0;
        end
    xbus[2] <= 1'b0;
    ...
end
```

- 如果组合逻辑比较复杂,可以在 always 块中使用阻塞赋值,再把最后的结果放在块外的连续赋值表达式的右边。

```
always@(Ain, Bin, Cin, Din, templ, temp2)
begin
    templ = Ain^Bin;
    temp2 = Cin^Din;
    ComboOut = (templ & temp2) | Ain^Din;
end
assign #5 OutBit = ComboOut | OtherComboOut; // Collect the delays here.
```

- 非阻塞赋值。同样用在过程块中。非阻塞赋值和阻塞赋值有两点区别:(a)在仿真中,非阻塞赋值不会阻塞仿真工具读取下一条语句;(b)在任何仿真时刻,它同阻塞语句一样被计算,但必须要等到所有的阻塞赋值和线型的值更新完了之后才会去执行它。
- 在 always@ 块使用非阻塞赋值语句表示时序逻辑。虽然非阻塞赋值的计算和阻塞赋值的计算是同时进行的,但非阻塞的赋值却要等到所有的阻塞赋值执行完之后,且发生了某个时钟事件时才会进行。这样也就保证了组合逻辑的输入值不会在时钟事件到来之前因为非阻塞的更新而改变。

避免: 用阻塞赋值来描述时序逻辑,用非阻塞赋值来描述组合逻辑。

避免: 在一个过程块中混合使用阻塞和非阻塞赋值。

避免: 采用延时#0的方法来避免竞争条件。

- 实现时钟和寄存器。有时钟,隐含地说明了是时序逻辑。因为受时钟驱动的值在时钟的跳变沿之间需要保持不变。目前我们不考虑对电平敏感的 latch 的情况。时序逻辑电路就是基于事件控制中的 posedge 和 negedge 来实现控制的。只对一个边沿敏感的变量就意味着这个变量的值在它敏感边沿相反的边沿被存储。

一个时钟产生器通常只出现在 testbench 中，通过对电平敏感的 always 块中的一条简单的具有延时的非阻塞赋值语句来实现。

```
reg Clock;
...
always@(Clock)
#10 Clock <= !Clock; // ~Clock also OK.
```

上述结构使用了 always 块语句的并行处理特点，在这个例子里，赋值语句必须是非阻塞的，因为这样 always 块的事件控制才会对变量的变化敏感。时钟 reg 变量必须在另外的某个地方被初始化。

另一种定义时钟的方法是在 always 块或 initial 块（更为常用）中使用过程 forever 语句。由于是过程语句，因此 forever 必须被包含在模块的并行块中，例如：

```
reg Clock;
...
initial // Only for clock generation.
begin
Clock <= 1'b0;
forever
#10 Clock <= !Clock;
end
```

上面的代码也可以用阻塞赋值来实现。然而，对于阻塞赋值，任何和时钟有关的值都需要特别注意，以保证能满足数据建立的条件要求。

需要注意的是，不要用 initial 块来初始化要被综合的设计或者要转换成真正硬件的设计：这是软件仿真编程和硬件实现编程的一个非常重要的区别。

2.3.3 Verilog 字符串和消息

- **Verilog 字符串类型。**这里的字符串指的是文字字符串，我们只能在系统任务中用它来打印提示信息。字符串的值可以被存储到 reg 类型的变量或存储对象（byte 数组）中。在使用 unicode 文本或者其他非 ASCII 文本字符编码时要特别小心。和 C 语言里 printf 函数的功能很类似，系统的消息任务也是把信息打印到仿真工具控制台的屏幕上。要获得更多关于仿真中打印消息的内容，可以参考 Thomas and Moorby (2002) 中 B.4 节和 F.1 节。

三个最常用的消息任务：

\$display(format, args_for_display)在仿真工具计算出等式右边（RHS）的值之前（即非阻塞赋值之前）打印。

\$strobe(format, args_for_display)在仿真工具计算出等式右边（RHS）的值之后（即非阻塞赋值之后）打印。

\$monitor(format, args_for_display)在仿真工具计算出等式右边（RHS）的值和完成了非阻塞赋值之后，如果某个参数发生变化，则会激活执行这个语句。通常在 initial 块中使用这个语句，附加一定的条件或延时。

断言。断言是基于设计者设计运行结果的预期而嵌入到设计中的检查机制。它被用来检查一个条件是否成立，并且在仿真中可以产生 warning 或者 error。和设计一个仿真程序一样，断言既是验证也是设计的一部分。

假设断言认为某些条件应该为真。那么当这些条件为真时，断言机制不会有任何行为；而当这些条件不为真时，断言就会打印警告信息，甚至中断当前的仿真。因此，断言可以提醒设计者注意那些因为设计改变或不常见的、较复杂的仿真条件下可能被忽略的条件。

尽管断言的主要功能是在被断言的条件不成立的时候提醒设计者，它也可以用来产生仿真错误，从而暂停或停止正在运行的仿真。不过我们暂时只讨论断言消息。

一个用到了 \$time 系统任务的断言的例子如下：

```
reg X, Y;
...
if (X!=Y)
$strobe("\n***Time=%04d. X=%1b == Y=%1b failed.\n", $time, X, Y);
```

消息不仅仅有文字信息，一般都还会打印出变量的取值。因此，为了提高这些变量值的可读性，还有一些专用的输出格式。

最常用的格式符如下（* 代表最常用的）：

- * %h 十六进制整数
- * %d 十进制整数
- %o 八进制整数
- * %b 二进制整数
- %v 强度级别 (strength level)
- %c 单个的 ASCII 字符
- * %s 字符串
- * %t timescale 单元中的仿真时间
- %u 0, 1 二值数据
- %z 0, 1, x, z 四值数据
- * %m 模块例化名 (见下)

除了 %t 以外，其他的整数格式之前都可以加上一个整数 “n” 来强制数据最小的显示宽度。另外，“0n” 表示在数据前的空白处添 0，例如：

```
integer x; ...; x = 5;
$display("%s = [%04b] = [%4b].", "The value", x, x);
```

这段代码会在仿真控制台上打印出：“The value = [0101] = [101]”。如果使用 “%b” 而没有前面的整数，那么 101 的前面就会出现 29 个空白，因为整数默认的位宽是 32 比特。

有一个特殊的字符串格式选项：%m，它看起来像是一个格式说明符，但实际上不是。相反，它在系统任务执行后输出被例化的完整层次名称 (hierarchical name)。关于层次名称 (hierarchical name) 在第 6 章中有详细解释。

2.3.4 移位寄存器

移位寄存器是一组寄存器，这组寄存器里存储的比特可以进行上移或下移。通常上移也被叫做左移，下移也被叫做右移。上和下的术语源于指寄存器所储存的无符号数的值范围的变化。同样，计数器也有向上计数和向下计数之分。汇编语言里也有同样的概念，代表 CPU 或存储器中寄存器中的内容向左或者向右移位。

要理解移位的过程，就要理解数的二进制表达形式。例如，有一个存储了如下内容的 8 比特寄存器：8'b0001_1001。经过一次上移之后，它的值为：8'b0011_0010。最左边的一位消失了，在最右边加了一个 0（因为 0 被默认的移到寄存器中）。一次下移之后原来的值变为 8'b0000_1100。如果这个寄存器 MSB 的输入和 LSB 的输出在一次下移的时候硬件上是相连的，那么下移后的结果就应为 8'b1000_1100；LSB 的 1 被移到了 MSB 当中。

当一个移位寄存器与其他的设计连在一起的时候，新的输入比特在移位寄存器的哪一端被移入，取决于寄存器的连接方式。对被移出的比特的处理也要视设计而定。

2.3.5 重收敛设计要点

用实现可编程存储的移位寄存器来讨论重收敛扇出（reconvergent fanout）的问题是很合适的。下面，我们用一个时钟的例子来说明这个问题。

为了让移位寄存器的功能不仅仅是一个 1 比特的 FIFO 或一个延迟控制器，就必须使它具有能被禁止的功能；这一功能可以使移位寄存器在一个或若干个时钟周期中保留住当前值。一种简单的实现方法就是用门控时钟，如图 2.2 所示。

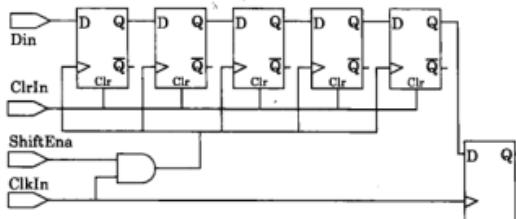


图 2.2 当时钟被关掉时，数据被保存。但这个时钟可能导致重收敛错误

我们看到，在上面这个电路里，控制数据移位的时钟也同时驱动一个独立的触发器（位于图的右侧）。这样这个时钟就拥有了重收敛扇出。问题在这里：时钟上的与门会导致经过它的时钟（控制移位寄存器移位的时钟）比不经过它的时钟（直接到达外部触发器的时钟）到达外部触发器的时间更晚一些。因此，即使本设计所采用的每个元件都满足建立条件，还是有可能出现这样的情况：外部触发器的时钟翻转太快来不及捕获移位寄存器从最后一个 D 触发器移出的值。

对使用门控时钟（gated clock）的设计来说，这是一个通病。它可以通过采用以下手段来解决：使用有使能控制引脚的移位寄存器；或者对所有独立于移位寄存器的元件和从移位寄存器接收数据的元件施加相同延时的时钟。

设计电路的时候要始终考虑到有可能会出现重收敛的情况。综合工具可以识别重收敛问题，可以将图 2.2 中的包含门控时钟的设计综合成一个不会出现上述建立错误的网表。当然，

如果设计者希望外部的逻辑电路接收的是延迟了的数据，那么就必须给综合工具加约束来限制它的综合行为。本书中，我们不会再继续深入讨论这部分的内容。在后面的练习里，会通过手动给移位寄存器中的每一个触发器添加移位使能逻辑来解决重收敛问题。

2.4 练习4：非阻塞控制

进入到Lab04目录进行本次实验。如果需要的话，可以在这个目录下创建子目录。

练习步骤

第1步。在一个Verilog模块中使用下述的例子创建三种不同的D触发器。添加一个断言，当这个D触发器被同时置位和清零时发出警告。通过仿真来验证你的设计，然后分别针对面积和速度进行综合。

简单的D触发器：

```
always@(posedge clk1) Q <= D;
```

异步清零的D触发器：

```
always@(negedge clk1, posedge clr)
begin
  if (clr == 1'b1)
    Q <= 1'b0;
  else Q <= D;
end
```

异步置数和清零的D触发器：

```
always@(posedge clk2, negedge pre.n, negedge clr.n)
begin
  if (clr.n == 1'b0) Q <= 1'b0; // clear has priority over preset.
  else if (pre.n == 1'b0) Q <= 1'b1;
  else Q <= D;
end
```

第2步。编写一个同步清零的D触发器的Verilog模型，并通过仿真来验证你的设计。

第3步。与第1步相对应，使用下述的例子来创建三种不同的D锁存器。你也许会希望复制第1步中的程序并修改触发器的代码来实现锁存器。在下例中，选择一种“简单的D锁存器”来完成实验。同第1步类似，添加一个断言，当这个D锁存器被同时置位和清零时发出警告。仿真验证你的设计，然后分别针对面积和速度进行综合。通过观察网表的电路图，确认综合工具生成的三个锁存器的确是我们所需要的。

简单的D锁存器：

```
always@(D) if (ena1==1'b1) Q = D;
```

在外部某处声明一个ena1变量；如果ena1和D一样，也是模块的输入。设想如果敏感变量表为always@(ena1)会怎样？

下述代码会生成哪种类型的D锁存器？

```
always@(D, ena) if (ena==1'b1) Q = D;
```

具有异步清零和低电平使能的 D 锁存器：

```
always@(D, clr, ena.n)
begin
  if (clr == 1'b1)
    Q = 1'b0;
  else if (ena.n==1'b0) Q = D;
end
```

具有异步置位低电平清零的 D 锁存器：

```
always@(D, pre.n, clr.n, ena2)
begin
  if (clr.n == 1'b0) Q = 1'b0; // clear has priority over preset.
  else if (pre.n == 1'b0) Q = 1'b1;
  else if (ena2 == 1'b1) Q = D;
end
```

第4步。串行载入的移位寄存器。在每一个时钟到来的时候，这个移位寄存器移位；没有时钟的时候，它保持当前值，电路图如图 2.3 所示。

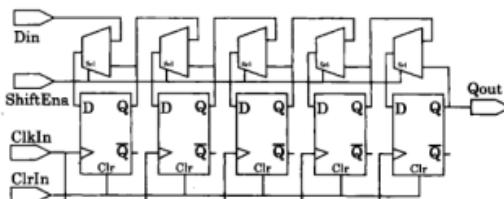


图 2.3 具有串行置位的移位寄存器

编写一个具有异步清零的 D 触发器的 Verilog 模型。用这个 D 触发器组成一个 5 比特的串行载入移位寄存器。触发器的输出是 Q 和 Qn。这个串行载入移位寄存器有一个 D 输入。把数据载入这个移位储存器需要 5 个时钟；它能被异步清零。为了使这个移位寄存器在时钟翻转的时候仍然能够保持数据，采用多路选择器(mux)给每一个触发器的 D 输入端提供二选一的输入：(a) 在移位使能的情况下，多路选择器应该把前一个触发器的 Q 连接到后一个触发器的 D 上；(b) 移位被禁止的时候，多路选择器应该把每一个触发器的 D 和这个触发器自身的 Q 连接起来。

图 2.4 中的电路图代表了一个二输入的多路选择器。

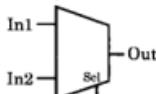


图 2.4 多路选择器的电路图符号

多路选择器模型的代码可以这样写：

```
always@(sel, in1, in2)
  if (sel==1'b0)
    outbit = in1;
  else outbit = in2;
```

多路选择器是组合逻辑而不是时序逻辑，所以应采用阻塞赋值。注意如果敏感变量表中的 in1 或 in2 中的一个变量被省略的话，那么就会出现锁存的状态，这时它就不是一个多路选择器而是一个某种类型的锁存器了。

另一种书写多路选择器的方式（在本次练习中不要这样用）：

```
assign outbitWire = (sel==1'b0)? in1 : in2;
```

D 触发器或多路选择器模型不应与移位寄存器位于同一个文件内。

简单仿真你的设计以验证其功能。

选做：分别针对面积和速度进行两次综合。

第5步（选做）。并行载入的移位寄存器。修改第4步的设计，使它能够在一个时钟里给这个移位寄存器载入5比特的数据。

最简单的实现方法是给第4步中的多路选择器增加第三个输入选择：这个选择使得5比特的数据输入总线中的每一个比特都和对应触发器的 D 端相连，如图 2.5 所示。

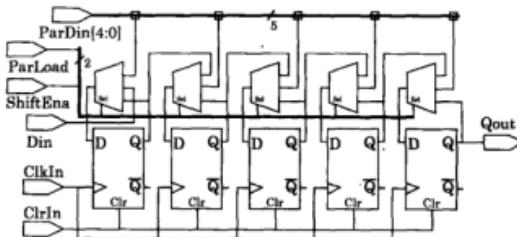


图 2.5 并行载入的移位寄存器

在这个并行载入的设计里，当移位被禁止时，每一个触发器的 Q 端都被连接到它自己的 D 端上；移位被使能时，每一个触发器的 Q 端都被连接到下一个触发器的 D 端；当选择进行并行载入数据的时候，每一个触发器的 Q 端悬空（最后一个 D 触发器除外），每一个触发器的 D 端都连接到数据并行输入总线的某一位上。

无论多路选择器具体是怎样实现的，一定要保证它能在这三种我们期望的连接方式中做出正确的选择。

第6步。在一个 always 块中重新编写第4步中的串行移位寄存器，将其改成一个过程模型：

```
always@(posedge ShiftClock)
begin
    QReg[0] <= Din;
    QReg[1] <= QReg[0];
    QReg[2] <= QReg[1];
    QReg[3] <= QReg[2];
    QReg[4] <= QReg[3];
end
```

这种类型的设计完全不需要 DFF 模型。为了增加第 4 步的多路选择器，你可以在等式的右边使用条件表达式来取代 Qreg：

```
always@(posedge ShiftClock)
begin
    QReg[0] <= (ShiftEna==1'b1)? Din : QReg[0];
    QReg[1] <= (ShiftEna==1'b1)? QReg[0] : QReg[1];
    QReg[2] <= (ShiftEna==1'b1)? QReg[1] : QReg[2];
    QReg[3] <= (ShiftEna==1'b1)? QReg[2] : QReg[3];
    QReg[4] <= (ShiftEna==1'b1)? QReg[3] : QReg[4];
end
```

这个移位寄存器只需使用一个 always 块来实现而不涉及层次设计。顺便提一下，如果在每条赋值语句的前面加上一个相同的延时，例如 “#1”，那么每次移位都会延迟 #1 的时长，而不改变逻辑功能。但是某些仿真器不能仿真这种带有延时的语句，这也是我们在过程块中不使用延时的原因之一。

通过仿真来验证功能（参见图 2.6），然后尝试分别按面积和速度优化的原则来进行综合。如果你使用的综合工具不能很好地综合带有延时的非阻塞赋值语句，也没有关系，请无论如何都试着综合一下，看看会出现什么样的结果。



图 2.6 移位寄存器模型的仿真结果。输出有 0.5 ns 的延迟，没有在代码中写出来

然后去掉所有的延时语句，再次进行综合。

如果在上述代码中用非阻塞赋值代替阻塞赋值，会出现什么样的情况？在上述代码中，完成一次移位共需多长的时间？

可以去掉这个过程块中移位寄存器的延时，或把赋值语句改成阻塞赋值之后再综合它。但是如果用的是阻塞赋值，那么这些阻塞赋值的语句就必须按照上述所示的相反顺序来排列（对于具有相同延时的非阻塞赋值语句来说，顺序是无关紧要的，因为等式右边的值在把它被赋给等式左边之前是不会被更新的）。

想要了解非阻塞赋值的更多特性，可以仿真目录 Lab04 中的 Scheduler.v 文件。

2.4.1 练习后的思考

对综合工具来说，锁存器会带来问题。这往往是无意中造成的。为什么这么说呢？

你觉得综合器使用结构建模和使用过程建模，哪种可以综合出更好的电路？为什么？如何改进触发器模型？是否应该加入 x 和 z 状态呢？

在串行载入移位寄存器中，如何添加一个断言来保证至少装入 5 个有效数据之前移位不会被禁止？

2.4.2 补充学习

和前面布置的一样，阅读 Thomas and Moorby (2002) 的第 1 章和 3.4 节。做 1.6 节的练习 1.2。

阅读 Thomas and Moorby (2002) 的第 2 章。我们将要学习 Verilog 中的有限状态机，因此可以稍微预习一下 FSM（有限状态机）。

关于锁存器，学习 Thomas and Moorby (2002) 的 2.3.2 节中的例 2.7，然后做 2.9 节的练习 2.2。

选做：Thomas and Moorby (2002) 的 5.2 节中关于参数的部分，忽略 defparam。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003) 的 9.2.2 节关于参数的内容；3.2.9 节、3.3.1 节和 9.5.3 节关于字符串和消息的内容；14.4 节至 14.6 节关于逻辑优化的内容。

如果对过程控制还存在疑惑，通读第 7 章的例子看能否给你帮助。你能发现在 Palnitkar CD 中 7.11 节中练习所对应的答案很有用；但是通常来说，在过程中使用 forever 或者 while，如同使用 always 而不使用事件控制一样，是不推荐的做法。

阅读 15.15.2 节来获得对基于断言的验证的大体认识，这种方法在现代复杂的大型设计中是十分重要的。

第3章 Verilog 基础知识2

3.1 线型，仿真和扫描

3.1.1 变量和常数

正如前面所提到的，变量有两种不同的类型，`reg`型和线型（`net`）。然而实际情况要复杂得多。

`reg`变量是无符号的，与它相对应的类型`integer`和`real`也是无符号的。这些类型都可以被过程赋值，并且会保留这个值直到下一条语句赋值为止。`integer`和`real`的位宽都是32比特，`reg`的位宽最小是1比特，最大的位宽由工具支持的最大位宽决定。为了增加灵活性，Verilog-2001版本规定`reg`变量也可以是有符号类型，也就相当于一个具有任意位宽的`integer`变量了。由于我们使用的数字设计工具不能综合`real`变量，所以在接下来的课程里基本上看不到`real`。

在Verilog里，`net`并不是某种具体的类型，它表明的是一种通用的连接特性。一个`net`变量的类型可以是`wire`，`tri`，`wand`，`wor`，也可以是其他一些我们以后要学到的类型。`wire`和`tri`的功能是完全相同的，为了便于记忆，它们有不同的名字。如果有多个信号同时驱动`wire`可能是发生了设计错误，而对`tri`来说则有可能是三态驱动。`wand`主要被用来实现多驱动，它对驱动进行线与（`wire and`）操作，`wor`对驱动进行线或（`wire or`）操作。因此可看出，`wand`和`wor`是在提供连接之后再附加了一种逻辑操作。

两个或者更多的并行赋值语句会对`wand`和`wor`的多驱动产生影响。可能会有如下两种方式：(a) 两个或更多的实例的输出引脚被连在一起；(b) 对同一个线型进行多个连续赋值。必须说明的是，在现代的CMOS设计中，`wand`和`wor`都很少被用到。

常数可以是数值、参数值或者字符串。

由于`integer`的位宽固定是32比特，因此将常数赋给`integer`时不需要声明位宽。但对每一个常数都列出它的位宽的是个很好的习惯，这样的话就能很清楚地知道操作的位宽，也有助于更好的定义结果。4个最简单的1比特状态的例子如下：

`1'b1`, `1'b0`, `1'bx`, `1'bz`; 或十六进制或者二进制表达式
`7'h15`, `16'b0001_0101_1100_1111`等。

参数的值默认是无符号的，当参数被初始化为一个十进制的数值时，参数的位宽默认是32比特；参数的位宽也可以自行声明。例如：

```
parameter x = 5; y = 11; // 默认为无符号数，32比特宽
parameter[4:0] z = 5'b11000; // 5比特宽
```

Verilog中没有字符串类型。但是在仿真打印消息或者断言时，可以用引号来包含要打印的字符串。字符串也可以存储在一个足够宽的`reg`变量中。但是除了把错误消息保存到RAM或者ROM里的情形，在其他的数字设计中很少用到这种方法。下面是一个打印字符串的例子：

```
$display("Error at time=%04d.", $time);
```

Verilog 中没有全局的变量或者常数，模块里要用的变量都必须在当前模块中声明。

3.1.2 标识符

Verilog 中的标识符 (identifier) 是指变量和常数被声明的名字，模块的名字、块或其他对象的名字。标识符遵循以下规则：

- 由 ASCII 码中的数字，字母和下划线 `_` 组成
- 对大小写敏感
- 名字的长度视工具的限制而定
- 只能以字母或 `V` 开头

以 `V` 开头的标识符被称为转义标识符 (escaped identifier)，它可以包含除空格以外的任意 ASCII 符号。标识符以空格结束，但空格只是作为界定符而不是名字的一部分。转义标识符通常不会由工程师手动写在 Verilog 设计源代码里，它通常被工具用来避免由翻译或移植引起的名字冲突。

3.1.3 并行块与过程块

并行块。并行块 (concurrent block) 的代码里的前后位置对仿真结果没有影响。例如，Verilog 源文件有可能是按照某种特定的顺序来编译，编译的先后顺序和仿真的先后顺序是无关的。最重要的并行块就是 module。连续赋值语句 initial 块和 always 块在一个模块中是并行执行的。其他类型的并行块包括 primitive 和 specify 块等，将会在后续的课程中讨论。

过程块。过程块 (procedural block) 位于并行块中，在仿真的时候被顺序读取。在执行的时候，也是按照代码被读取的顺序来执行的。过程块包括：

- 阻塞赋值语句
- 非阻塞赋值语句
- 过程控制语句 (if, for, case)
- 函数或任务调用 (后面讨论)
- 事件控制 (“@”)
- fork-join “并行” 语句 (后面讨论)
- 嵌套在 begin…end 中的过程块

3.1.4 Verilog 其他特性

宏 (macro) 类似于 C 语言当中的预处理指令，只不过是以 `#` 而不是 `##` 开头。宏并不是严格意义上的语言元素，它不与其他任何特定的结构相关，可以出现在代码中任意一行上。例如：

```
'define, 'timescale, 'ifdef, 'include
```

系统任务 (system task) 和系统函数 (system function) 在本课程中只做简单的介绍。它们属于仿真结构，只能出现在过程块中。例如，\$strobe, \$display, \$sdf_annotation, \$stop, \$finish 等。

时序检查将在后面详细介绍。它们是并发执行的，预先定义好的断言，只能在 specify 块中使用，例如 \$width, \$setup, \$hold。

PLI (Programming-Language Interface, 编程语言接口) 是一个基于 C 语言的常规库，它允许用户能够自行定义新的系统函数和系统任务，目的是增加 Verilog 仿真器的功能。在后续的课程中我们会接着讨论 PLI，但是在整个课程中我们不会用到 PLI。

3.1.5 Backus-Naur 范式

Backus-Naur 范式（简称 BNF）是一种定义语言语法的方法，被广泛应用于 Verilog 和其他规范化的文本当中。它定义了文本语言的语法规则，能够有效地减小语言的含糊性。在本课程里不会用到它，但是知道它却是有好处的。在 Thomas and Moorby (2002) 的附录 G 中有详细介绍。

BNF 本质上是分层的，十分简单：语法中主要元素的合法子结构都被列在 “::=” 之后，子元素都被列在 “:=” 之后。为了简单起见，我们来看一个变量的例子：

```
variable ::= reg | net
          net := wire | tri | wor
```

这样我们就可以推断出，任意一个变量肯定是 reg, wire, tri 或者 wor 中的一种。通过这个 BNF 例子，可以很明显地判断出把 parameter 当做变量来使用是错误的。

3.1.6 Verilog 语义

Verilog 是一种硬件描述语言，它代表的是硬件。在 VLSI 背景下，Verilog 代表的就是逻辑门和连接线。逻辑门对应于 reg，模块实例，integer 等，而连接线对应于各种类型的 net。

和其他的编程语言一样，Verilog 也是通过表达式 (expression) 和语句 (statement) 来实现功能的。一个表达式就是一个能被计算的式子 (代表一个值)，一个表达式的例子是逻辑求和或逻辑与，例如 “5+7”，“A&&B”。一个表达式只代表了某种计算，而不会改变任何东西。

与表达式不同，语句实现对变量的赋值，因此能改变变量的值。被改变的值要么被局部存储起来，要么被传送到其他某个地方。例如，在一个过程块中，“Zab = A && B;”就把 A 和 B 的逻辑与的值赋给了一个名为 Zab 的变量。下面两个等价的并行语句，连续赋值语句 “assign Z = A && B;” 和元件例化 “and and01(Z, A, B);” 都改变了一个名为 Z 的 net 型变量的值。

必须对语句加以控制，常用的方法是用变量值的变化来控制。例如，“assign Z = A&B;”，每当 A 或者 B 中的任意一个或两个都发生变化的时候，这条语句就会被再次执行。一个 always 块中包含若干的表达式和语句，设计者必须给这样的一个块加上控制事件（敏感变量表），这样只有当敏感变量表中的某个变量发生变化的时候，所有的块语句才会被重新读取。例如一个被 “always@(A)” 控制的块，即使它的块语句中包含这样一条语句 “Z reg = A&B;”，也只有当 A 的值发生变化的时候它才会被重新读取。但对于一个被 “always@(*)” 控制的块而言，只要块中表达式包含的任意一个变量发生变化时，这个块都会被重新读取。

锁存器和多路选择器。只对电平敏感（不考虑 posedge 和 negedge 边沿敏感），且 always 块中的敏感变量表中没有包含在块中出现过的所有变量（称为不完整的敏感变量表），那么这个块的功能就是某种类型的锁存器；拥有完整的敏感变量表则表明这是一个由组合门或多路选

择器等组合而成的组合逻辑电路。但如果控制条件中包含的if或case忽略掉了某些值的话，那么即使敏感变量表是完整的，也有可能出现锁存的状态。

例如，一个包含有“wire a, b, sel;”和“reg z;”声明的模块。考虑下述4种不同的always块：

always@(a, b, sel)	always@(*)	always@(sel)	always@(a, b)
if (sel==1'b1) z = a; else z = b;			

左边的两段代码代表了多路选择器，因为表达式中所有的变量都被包含在了敏感变量表中，并且if中的每一个选择项都被赋予了一个输出(z)。另外两个always块代表了某种非标准的锁存器，当敏感变量表中的变量不发生改变时实现锁存。最右边的这个always块对语句中等式右边的每一个变量都敏感，因此它代表了一个抽象意义上的锁存器：它锁存的是选择信号，而不是输出值。

下面是一个没有过程控制结构的非标准锁存器：

```
always@(a,v) // typo!  
Z = a | b;
```

b值的变化对z的锁存并无影响，这种类型的锁存器通常是由敏感变量表中的输入错误造成的。

去掉一个多路复用器中的sel ==1'b0，可以生成一个透明锁存器。代码如下，等效电路图如图3.1所示。

```
// verilog simple transparent latch:  
always@(D,Sel)  
if (Sel==1'b1)  
Q = D;
```

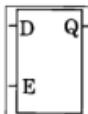


图3.1 简单的透明锁存器，Sel被改成了E(enable)

上述always块的写法是可综合的锁存器的推荐写法，如果综合库中有这种锁存器的话，综合工具会在库中直接找到它。

另外一种可综合的锁存器是通过连续赋值来实现：

```
assign Z = (Sel==1'b1)? D : Z;
```

尽管前面的always块通常会被综合成锁存器元件库中已有的类型，但一个连续赋值的锁存器会被综合成带有反馈的显式组合逻辑。

连续赋值的锁存器有可能被综合成图3.2所示的电路。

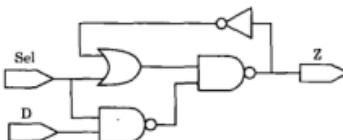


图 3.2 用连续赋值语句产生的简单、透明的锁存器

连续赋值语句会被综合工具当做组合逻辑来处理，所以综合工具也许不会去综合库中寻找一个时序元件来实现这个电路。基于这个原因，使用组合逻辑来实现一个锁存器不是一个好办法（尽管我们在后续的课程中会用组合逻辑来实现 S-R 锁存器，但那是为了教学的需要）。

当采用组合逻辑实现锁存器的时候，具体的门结构取决于综合工具，在优化的时候有可能出现不可预测的结果。因此，相比直接采用一个简单的时序结构而言（如前面的 always 块，参见图 3.1），采用组合逻辑实现的锁存器的时序（包括可能出现的毛刺）容易出现问题。为了避免这些问题，应采用时钟控制的结构（即触发器）而不是由使能控制的结构（即透明锁存器）来保存数据。

3.1.7 时序逻辑模型

现在我们来研究 Verilog 的一些建模方式，这些方式能够产生精确的综合结果。我们将完全摒弃锁存器而只使用触发器，原因将在后面说明。

时钟。Verilog 中的时钟块必须是一个带有边沿表达式（posedge 或 negedge）的 always 块结构。一个由时钟控制的块不能包含一个以上的时钟，也不能对电平敏感。但是，表达式里允许出现多个边沿。例如：

```
always@(posedge clk) ...
always@(negedge clk) ...
always@(posedge clk, negedge clear) ...
```

下面的写法不能被综合，因此不推荐使用。

```
always@(posedge clk, clear, preset) ...
```

异步控制。异步控制（Asynchronous Control）通常包括单独的预置数值，清零，或二者都有。如果同时包含这两者，那么无可避免会出现优先级的问题。例如：

```
always@(posedge Clk, negedge Preset_n, negedge Clear_n)
  if (Preset_n==1'b0)
    Q <= 1'b1;
  else if (Clear_n==1'b0)
    Q <= 1'b0;
  else
    Q <= D;
```

在这个模型中，如果 Preset_n 和 Clear_n 同时有效，Preset_n 的优先级比 Clear_n 高。不管这两个信号同时出现是不是发生了某种错误，一个正确的网表应该包括这种体现出优先级的逻辑。不过，有时设计者的本意并不是要设计一个有优先级的电路，只是希望设计一个有两个控制引脚的单个门电路。

如果可能的话，应尽量避免一个以上的异步控制：如果综合库中不包含支持预置数值和清零的器件，那么对应的代码有可能无法被综合成电路；即使综合库中有类似的器件，也可能出现综合后的网表不符合设计本意的问题。如果有可能的话，可以给综合工具添加一个限制条件或者是某个指令，从而强制综合工具在综合库中针对某种设计选择相应的具有对应优先级的器件。如果没有添加上述的控制，综合工具则有可能自行加入逻辑以实现源代码中的优先级，或者完全忽略代码中的异步控制的优先级。

竞争条件。如果代码造成在仿真的时候出现不确定的值(0或1)，那么就形成了竞争条件(Race condition)。这通常意味着实际对应的硬件可能工作不正常。

例如，在一个 always 块中，有延时的非阻塞赋值语句会被并发执行，形成了竞争条件：

```
always@(posedge Clk)
begin
#2 X <= 1'b1;
#2 X = 1'b0;
#3 Y = X; // Ambiguous value used.
end
```

不同的 always 块之间也会出现这种情况：

```
always@(posedge Clk) #1 X = a;
always@(posedge Clk) #1 X = b;
```

为了避免大多数的竞争条件，决不要在一个 always 块中混用阻塞赋值和非阻塞赋值；也决不要在多个 always 块中对同一个变量进行赋值。虽然上述用法可以在仿真中使用，但不能被综合工具综合。

另外，把大的功能块或模块的输出寄存起来是一个很好的做法。所有输出端上的触发器能保证在时钟沿到来的时候把当前值（而不是会导致冲突的值）输出到其他器件的输入端上。

最后，绝不要在 testbench 的 initial 块中之外的地方使用 #0 延时。原因将在后续的课程中说明。就目前而言，记住只要不要像下面这样写就好：

```
always@(posedge Clk)
begin
#0 Q1 <= a; // Likely error! Never use #0!
#0 Q2 <= b; // Likely error!
...
end
```

可综合的语言子集。并不是所有的 Verilog 语句都能被综合，为了使你的设计既能有效地仿真也能被正确地综合成电路，下面有一些经验之谈：

- 不要在敏感变量表中混合使用边沿敏感表达式和电平敏感表达式。
- 如果要在 always 块中使用延时表达式，只能在阻塞赋值时使用。综合工具不能综合任何有延时的非阻塞赋值，要么报错，要么发出严重警告。后面将解释这是为什么。
- 给仿真结果增加延时的时候，尝试使用下面这种方法：不要在 always 块中增加延时；可以先估计出对应的总延时，然后把这个总的延时加到 always 块外的一条连续赋值语句当中。例如：

```

module MyModule (output X, Y, rest of sensitivity list);
local declarations
...
assign #5 X = Xreg; // estimated total delay = 5.
assign #7 Y = Yreg; // estimate = 7.
...
always@(posedge Clock) // A very strange flip-flop!
begin
  x1 = (a && b) ^ c;
  Xreg = x1 | x2;
  Yreg = &(y1 + y2);
end
endmodule

```

3.1.8 可测性设计 (Design For Test): 扫描练习简介

设计工具可以自动将扫描 (scan) 插入到设计当中。通常，在设计通过了仿真后，在设计阶段的后期，扫描被插入到设计中。理解插入扫描的机制会帮助你更好的理解和解决插入造成的错误和低效率的问题。

之所以被称为扫描，是因为扫描寄存器允许硬件测试点中的逻辑状态被扫描（即线性的移位）。

扫描的目的是为了观察一个设计或者是整个芯片的内部变化。扫描寄存器即硬件测试点，设计者或者测试工程师利用它去了解硬件内部的状态。扫描寄存器不仅仅用于仿真，它们会一直被包含在设计中并且成为物理芯片的一部分。

主要有两种类型的扫描：内部 (internal) 扫描和边界 (boundary) 扫描。

在进行内部扫描时，所有的输入输出（全部或部分），设计中的组合逻辑，都变成了扫描的元素。这是通过将设计中所有的触发器或锁存器用扫描触发器或扫描锁存器代替来实现的。很明显，任何组合逻辑块的 I/O 都必须是一个芯片的引脚或者一个触发器或锁存器之类的时序元件。

用一组特别的测试端口来控制扫描操作；这组端口被标准化组织定义为 JTAG (Joint Test Activity Group)。

新的扫描元件的功能与被它们所取代的元件是一样的，不同的是它们被很多的 mux 串组成了一条扫描链，也就是一个遍布部分或整个设计的巨大移位寄存器。当这些扫描 mux 处于运行模式或正常模式的时候，扫描寄存器的工作方式和插入扫描之前没有任何区别；当它们处于扫描模式的时候，它们将被扫描输入和链上的数据都移出，以待检查和纠错。因此通过把输入扫描进扫描链，然后正常运行，再扫描所有的输出，就能测试出设计中的每个组成部分是否工作正常，找出设计错误或硬件的随机问题（如果有的话）。

图 3.3 至图 3.5 显示的是扫描链是怎么被插入到时序器件（例中为触发器）里去的。

在本章后，我们会做插入扫描的练习。由于最初的设计中并不包含任何可以被替代的时序器件，我们还需要做些小修改。

边界扫描多用于板级 (board-level) 的硬件测试；它能够监视芯片的边界。在边界扫描中，芯片的引脚和芯片内部的门器件相连，同时它们连接到外部的扫描测试逻辑电路上。这样可以使即使芯片被安装到电路板上，仍然可以通过引脚移进移出测试数据，从而实现测试和观察测试的结果。边界扫描省去了在每个硬件测试点加上针床或其他的外部硬件探针的步骤。现代

球栅(ball-grid)芯片的封装拥有超过1000个的引脚，彼此之间的距离为毫米级的，这就使得针床的方法不太现实，并且也容易损害这些微小、脆弱的触点。

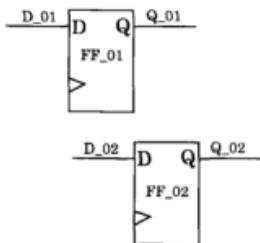


图3.3 未插入扫描的设计

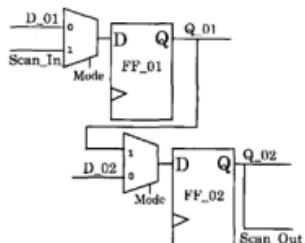


图3.4 插入多路选择器

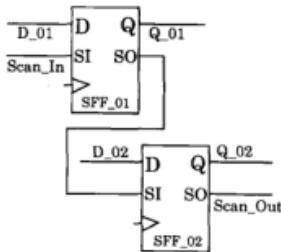


图3.5 自带扫描的库器件

边界扫描通常和某些内置的自测协议结合在一起。除了JTAG口之外，边界扫描通常还需要一个测试行为端口（Test Activity Port, TAP）控制器。它是一个内置的状态机，由于结构复杂，我们暂不研究。

在下面的练习里，我们将要运用学过的移位寄存器的知识，来给练习1中的Intro_Top设计添加扫描触发器。普通的内部扫描只会给一个寄存器增加一个多路选择器，而我们将会给每个触发器增加两个多路选择器。这是因为在练习1的原设计里没有用到触发器，因此我们除了扫描之外，还需要多用一个多路选择器来旁路扫描触发器的触发器功能。虽然看起来我们是在研究扫描插入，但实际上是一些Verilog的练习。

JTAG的测试端口由5个特定的1比特的端口组成：一个扫描输入端口，一个扫描输出端口，一个扫描时钟端口，一个扫描清零端口和一个扫描模式端口。

我们把扫描插入到练习1设计中的目的是为了了解整个流程；时钟问题和逻辑的安排将会在课程的后面再次讲到。

在练习开始之前，概括一下我们将要做的事情：

首先，我们会在Intro_Top的组合逻辑中加入触发器和JTAG端口。由于Intro_Top是纯组合逻辑，这就意味着我们将给每一个I/O都添加一个触发器。在添加完触发器之后，用时钟控制它们来重现原设计的功能。由于唯一的时钟是JTAG的扫描时钟，因此用它做这个设计里的时钟。这时，JTAG的其他端口还没有被用到。Intro_Top设计就变成了与扫描时钟同步的设计，除了有同步延时之外，功能相比原设计并无变化。

然后，给每一个触发器加入两个多路选择器。两个多路选择器的状态应该相同，都处于运行模式下（普通模式）。如下面例子，Intro_Top 的两个输出都换成了有多路选择器的触发器，如图 3.6 所示。

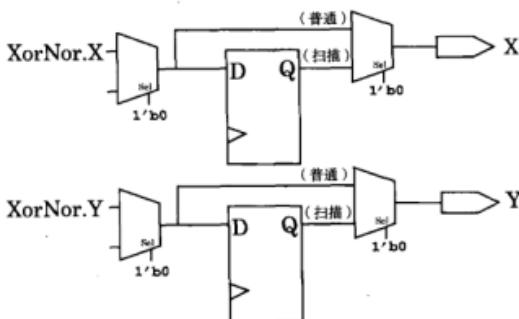


图 3.6 Intro_Top 的两个输出都换成了有多路选择器的触发器。0 意味着处于普通模式

现在，即使我们让这些多路选择器处于扫描模式，它们也不能实现扫描。因为扫描的输入和多路选择器的输入还没有连接起来。为了能够立即通过仿真来验证这个设计，我们在设计外面给多路选择器加入触发器。这意味着 Intro_Top 又变成了纯组合逻辑，没有了同步延时，但是多了一些由复用器带来的传播延时。

注意图 3.6 中的设计，它处于普通模式，触发器的状态被忽略，也不会对扫描时钟做出响应。

最后，我们把悬空的多路选择器的输入顺序连接起来。在下面的练习里可以看到，这样做能将触发器串到一起，让它们能够在电路处于扫描模式的时候发挥移位寄存器的作用。所有的多路选择器的输出都已经被连接了起来。因此，不需要修改输出。

现在我们开始练习。

3.2 练习 5：简单的扫描

练习步骤

在目录 Lab05 里完成这个练习。

这个目录里的文件 Intro_Top 和练习 1 里用到的文件几乎是一样的，只做了很小的改动。每一个模块的文件中都包含了修改记录。把修改记录放在文件最开头的注释里是个很好的习惯。它把设计相关信息传递给了其他的设计者，而且当设计者在隔了较长的时间之后再来看自己写的代码时，记录可以提示设计者，他曾经做了些什么。修改记录的细节取决于项目的代码风格。在本练习的最后，还有可用来综合脚本文件。

第 1 步。 在 Intro_Top 设计中加入一个 JTAG 测试端口。加入 5 个 1 比特的端口，先不连接它们。这 5 个端口分别是：ScanMode, ScanIn, ScanOut, ScanCir 和 ScanClk。除了 ScanOut 是输出，其余的端口都是输入。这个模块头应该和下面的模块类似：

```
module Intro_Top (output X, Y, Z, input A, B, C, D
    , output ScanOut
    , input ScanMode, ScanIn, ScanClr, ScanClk
);
```

第2步。给顶层设计（是Intro_Top.v，不是TestBench.v）中的每一条I/O路径都插入D触发器（把它叫做FF），但JTAG的I/O除外。

也许你想使用练习4中的FF设计。它应该包含一个上升沿时钟和一个高电平有效的异步清零端。

为了实现这个目的，在Intro_Top中修改端口和连线即可，不用修改练习1里的各个子模块。

把每个输入都连接到一个新FF的D端上；然后把这些FF的Q端连在这些输入原来驱动的对象上。对每一个FF的Q端都要声明一个新的wire连接。

如果不给wire和例化的器件起好记易懂的名字，就容易造成设计上的错误。例如，假设你的FF模块被命名为DFFC（D flip-flop with clear，带清零端的D触发器）。对于顶层的A输入端口，要想把它连接到练习1中实例名为InputCombo的模块的A引脚上，则应该这样做：

```
wire toInputCombo_A;
...
DFFC Ain_FF(.Q(toInputCombo_A), .D(A), ...);
```

对输入端采取这样的处理之后，对输出端也采用同样的方法来处理。在Intro_Top中，把新加入FF的Q端口连接到原来的输出端口上，然后把原来驱动这些输出的线连接到新加入FF的D端。同样，也需要声明新的wire来实现这种连接。

在连接完所有FF的数据端口之后，把ScanClk输入连接到每一个FF的时钟引脚上；把ScanClr连接到每一个FF的异步清零引脚上。现在所有的FF都应该类似于这样：

```
DFFC myFFinstanceName (..., .Clk(ScanClk), .Clr(ScanClr));
```

在TestBench.v中给ScanClk和ScanClr添加驱动，并保证它们被连接到了顶层的设计当中。

在整个的课程当中，我们都会一直坚持这样一个写testbench的好习惯：专门为testbench里一个被驱动的设计输入声明一个reg变量；把这个变量命名为reg*stim，其中*是输入端的名字。例如，Intro_Top中的A输入端口应被testbench中的一个名为Astim的reg变量来驱动。同样，也应该在testbench中为每一个输出都声明一个wire，并命名为*watch。例如，一个设计的输出端X应该与testbench中一个名为Xwatch的wire相连。这样就能仿真输入并观察结果了。这种给testbench命名的方式给我们带来了方便，不用去查看testbench的代码就能知道设计中的变量和testbench的变量的连接关系。

用时钟控制FF来将testbench中的激励加入到设计的输入中，并移出当前（而不是更新后的）*watch输出。然后，仿真过了一段时间，等组合逻辑发挥作用之后，再次用时钟控制FF来移出新产生的*watch值而并不改变输入的值，这些*watch值是由当前的输入和设计的组合逻辑决定的。

编译并仿真修改后的设计，从testbench向下，一级一级地检查FF和时钟连接的正确性。

第3步。检查时序。观察练习5里的子模块会发现有很多10个单位的延时，所以一定要保证在testbench中定义的时钟周期足够长，比如可以将周期定义成50个单位。这样的话，模块才有足够长的处理时间：

```
...
always@(ClockStim) #25 ClockStim <= !ClockStim;
//  
initial  
begin  
#0 ClockStim = 1'b0;  
...  
end
```

testbench中的ClockStim应该被连接到修改后的设计的ScanClk端口上。同时还需要一个新的ClearStim来驱动设计中的ScanClr。由于本练习不需要触发器复位，只需将ClearStim置为0即可。通过简单的仿真来检查设计的时序，如图3.7所示。

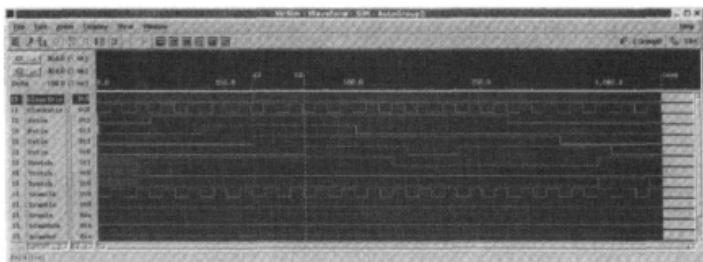


图3.7 Intro_Top 改成同步设计之后的波形图

练习1中的组合逻辑加上我们后来添加的时序逻辑，就组成了一个新的同步设计。之所以把它叫做是同步(synchronous)设计，是因为它的所有操作都是和ScanClk时钟同步的。输入在每个时钟的上升沿进入，结果在下一个时钟的上升沿输出。

选做：综合这个设计，然后观察综合后的电路。

第4步。加入多路选择器来去除所有的同步行为。先只加入多路选择器而不用着急把它们连接起来。在Intro_Top中使用always块或者连续赋值语句来实现多路选择器，不要使用(结构)器件。

每一个FF都需要一个mux来驱动它的D输入，从而使得它的输入是正常输入或来自于扫描链上。第二个多路选择器被用于连接到正常输出或去掉到正常输出的连接。如果没有第二个多路选择器，那么FF的Q输出就会和驱动下一个FF输入D端的变量发生竞争。

最左边的(前一个扫描)输入如图3.8所示，它应该被一个常数值驱动，例如1'bx；先不要连接下一个扫描的连线。可以声明一个wire，给它取个好记的名字。例如称它为THIS_IS_X，并把它的值赋为1'bx。然后再使用它给那些在目前阶段输入状态应设为未知的多路选择器输入端赋值。

这样做的结果是，图3.8中的FF在正常模式下会完全被它的两个多路选择器旁路掉。因此，在正常模式下我们就得到了练习1设计中的纯组合逻辑。

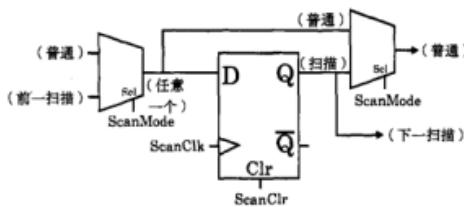


图 3.8 Intro_Top 中的寄存器被加上了两个多路选择器。通常，扫描操作只需要一个多路选择器

写多路选择器的代码的时候，用下面的方法来决定多路选择器的选择功能：当 ScanMode 为 1 时，FF 被串入扫描链；当 ScanMode 为 0 时，FF 处于正常的工作模式。给每一个实现多路选择器的语句加上一个小延时，如 #1。

在 testbench 里令 ScanMode = 0，然后进行仿真。得到的结果除了多路选择器带来了延时以外，应该与没有添加 FF 和多路选择器的练习 1 的仿真结果一致。

练习 1 里设计实现的功能再次被实现，只是多了一些暂时没有发挥作用的 FF 和多路选择器，到目前为止，它们只是带来了一点延时。

注意，并没有对子模块做任何修改。原来的设计是纯组合逻辑的，我们只是在设计的顶层加入了一些扫描元素。结果看起来像是边界扫描，实际上它是针对一个内部不含有任何时序门逻辑的内部扫描。

选做：综合这个设计，然后观察综合后的电路。

第 5 步。创建一条扫描链。在这个阶段，FF 已经被多路选择器旁路掉了，没有发挥作用。同时所有多路选择器的扫描输入端也是悬空的，没有被任何信号驱动。因此可以用多路选择器的扫描输入端把 FF 连接成一个移位寄存器，而不会影响到整个设计的功能。

如图 3.8 所示，选择一个 FF，把它的 Q 端连接到 ScanOut 端口，然后把另外一个 FF 的 Q 端连接到驱动第一个 FF 的输入端的多路选择器的输入引脚上。然后重复这个过程，直到所有的 FF 被连在一起。这样，除了第一个多路选择器，在上一步中多路选择器不确定的输入端现在被连接到了 FF 的 Q 端。第一个多路选择器的输入仍是未知的。ScanOut 应被连接到扫描链的最后一个 FF 的 Q 端上。

最后，把驱动最后一个 FF 的 D 端的多路选择器的输入引脚连接到 ScanIn 端口上，使这个 FF 成为扫描链上的第一个 FF。结果如图 3.9 所示，只是为了简单起见，省略了 ScanMode 和 ScanClk。

你觉得被虚线连起来的扫描链看起来眼熟吗？因为它就是一个移位寄存器！

第 6 步。使用有意义的例化名。给这些触发器改名，让它们的名字能够代表它们在扫描链中的顺序：例如，某个触发器是扫描链中第一个触发器，它被输入端口 A 驱动，那么就可以把它命名为“A_FF_s01”；而扫描链上的第 4 个触发器，输出 X 驱动的触发器则可以被命名为“FF_X_s04”。

选做：对设计进行综合，然后尝试在网表中找出扫描链。

第 7 步。添加一个仿真安全网（simulation safety net）。移进或移出所有扫描链中的内容，需要 7 个扫描时钟周期；通常来说，视 FF 的顺序而定，扫描进新的激励或者是移出新的结果消耗的时钟周期要少一些。假设扫描模式中一个连续执行超过 8 个时钟周期的操作是错误的；在 testbench 中编写一个断言，当扫描模式连续执行超过 8 个时钟周期的时候，就发出警告。

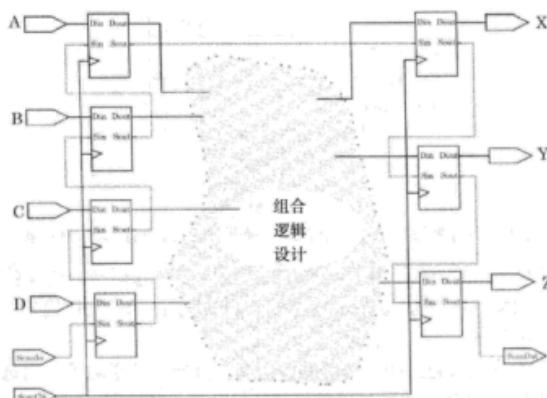


图 3.9 练习 5 扫描插入的逻辑示意图。在普通模式下，Din 和 Dout 是正常的连线，而 Sin 和 Sout 为开路。在扫描模式下，Sin 是 FF 的 D 输入，Sout 是 FF 的 Q 输出。Din 和 Dout 为开路

图 3.10 显示了一个 testbench 中包含了安全网 (safety net) 断言的仿真结果。这个断言位于 testbench 中，因此不会被综合工具识别，也不会对综合后的网表产生影响。

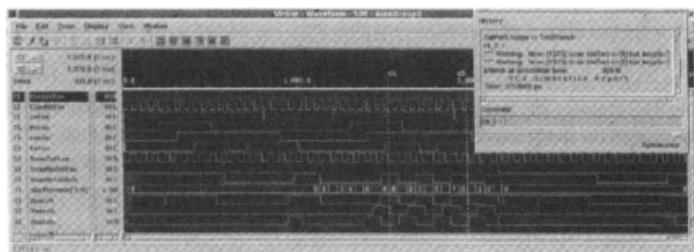


图 3.10 用完成了第 7 步后的 Intro_Top 进行仿真

第 8 步。 仿真设计以加深对它的理解。针对面积进行综合（从顶层综合而不是 testbench），观察优化后的网表，找到多路选择器并统计它们的个数。

选做： 完成了针对面积的综合之后，不退出综合进程。把设计打平，然后选中增量映射 (incremental-mapping) 选项再次综合这个设计。最后，仍然不退出综合进程，再次综合这个打平后的网表以获得最优的面积。这样的操作之后，面积将获得很大的改善。

第 9 步 (选做)。 综合工具会自动地通过用扫描器件来替代时序器件以达到插入扫描的目的，例如用带扫描的触发器来代替普通的触发器。要利用综合工具的这一特性，设计中就必须包含时序器件。现在我们不管手动插入扫描后的设计，回归到原来那个纯组合逻辑的 Intro_Top 设计上。

保持 TestBench.v 不变，从目录 Lab01 里复制一个 Intro_Top.v。在 Intro_Top 的输出驱动和输出端口 (X, Y 和 Z) 之间加入 D 触发器来实现数据的保存。这个 D 触发器可以使用你自己设计的 DFFC 模块。但要稍加修改，去掉 Qn 端。这三个触发器就可以当做综合工具自行添加

扫描时所需的时序器件。增加一个名为 Clk 的时钟输入端和一个名为 Clr 的清零端来控制这些触发器，如图 3.11 所示。

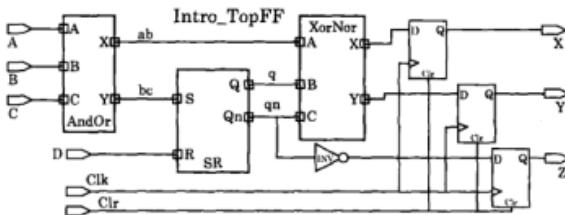


图 3.11 加入了触发器后的 Intro_TopFF，支持扫描自动插入

然后，增加一组使用以下标准名字的 JTAG 端口，暂不用连接：tms (test-mode select)，tdi (test-data in) 和 tdo (test-data out)。其他两个 JTAG 端口 tck 和 trst 与 Clk 和 Clr 共享端口。

把 Intro_Top 重命名为 Intro_TopFF。修改 TestBench.v 使得可以仿真。

可以使用在目录Lab05中为你准备的脚本利用综合工具自动插入扫描。在综合了这个更新后的设计之后，在文本编辑器中（也可以使用 design_vision）查看综合后的网表。

3.2.1 练习后的思考

如果你的设计工作在扫描模式里，每次位移都有可能导致组合逻辑的输入值发生改变，引发了一系列的逻辑变化，从而导致出现 glitch，这个问题值得关注吗？

怎样定义一个操作协议来尽可能地减小在扫描链移位的时组合逻辑发生的变化？

怎样修改扫描链使得组合逻辑在扫描模式下可以保持某种常态？

3.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 4.1 节关于并行处理和 4.2 节关于事件控制的内容。

阅读 4.5 节和 4.6 节中关于不命名块和（选做）的 4.9 节 fork-join 的内容。在 4.9 节中，注意 fork 和 join 可以被用来代替 begin 和 end。然而，在每一个 fork 和 join 块中加入 begin 和 end 可以增强可读性，也易于修改；如果要用 `ifdef DC 来使得包含有 fork-join 的代码是可综合的，那么 begin 和 end 是必需的。

做 4.10 节里的练习 4.1。

（选做）通读 Thomas and Moorby (2002) 附录 G 关于 BNF 的部分，直到你理解了应该怎么样用它。

阅读 Palnitkar (2003) (可选)

阅读 4.6.1 节关于编写可综合的 Verilog 代码的内容。

阅读 14.9 节里的练习 1，综合一个 RTL 加法器。

阅读 14.9 节里的练习 2 和练习 3。

学习 Palnitkar 附带光盘（第 14 章，目录 ex.1）中综合的例子。这个例子中包含预综合门级网表和一个 Verilog 的基本元件模型库。

第4章 锁相环和串行/解串器入门

4.1 锁相环和串行/解串器工程

在本章中，我们将会完成一个锁相环和一个并串转换器的设计。

贯穿于整本书，我们将渐进完成一个符合 PCI Express 规范的串行/解串器（Serializer-Deserializer, SerDes）的设计。

4.1.1 锁相环

一个典型的锁相环（Phase-Locked Loop, PLL）由以下三部分组成：输出时钟产生器，相位比较器，可变频率振荡器（Variable-Frequency Oscillator, VFO）。常见压控振荡器（Voltage-Controlled Oscillator, VCO）就是VFO的一种。PLL 会比较输入时钟相位和VFO产生的输出时钟相位之间的差别，并且通过这个差别来调整VFO产生的时钟频率。从理论上讲，一个理想的PLL可以锁定任何输入时钟频率；但实际上，特定的PLL只会准确地锁定一个较窄频率范围内的输入时钟。

虽然PLL总是锁定在输入的时钟频率，但如果我们在产生输出时钟的逻辑里加入专用的计数逻辑，用计数分频后的时钟和输入时钟进行相位比较，再直接把VFO未分频的时钟输出。这样，这个锁相环可以产生数倍于输入频率的时钟，从而实现了倍频。

4.1.2 1倍数字PLL

PLL一个重要的作用就是消除时钟的延迟。对于1倍的PLL，输出时钟和输入时钟的频率相同。通常它被用在时钟树的末端，用来消除时钟插入后带来的延迟，使得时钟树末端的时钟相位和时钟树输入节点的相位几乎保持一致（参见图4.1）。

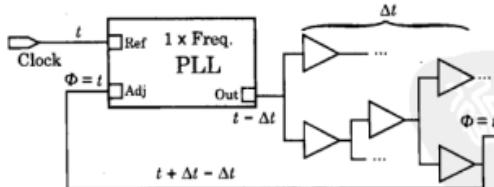


图 4.1 利用 PLL 来消除时钟树的延迟

接着，我们来思考应该怎样设计这样的一个PLL。

对于1倍的PLL来说，不需要实现倍频。根据前面的介绍，它的结构图应该如图4.2所示，包含一个时钟相位比较器和一个VFO。

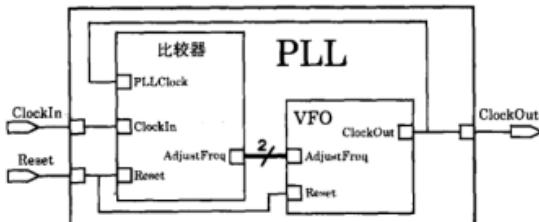


图 4.2 1 倍的数字 PLL 电路图。Comparator 和 VFO 之间的控制总线为 2 比特宽

如果我们用一个实型 (real) 变量来保存时钟翻转的延迟值。通过在仿真过程中调整这个延迟值，从而实现一个可变频率的 VFO。这样的代码是不可综合的，但是我们现在不用去关心综合的问题。

用伪代码来表示这个 VFO，代码如下：

```
real ProgrammedDelay;
...
begin
ProgrammedDelay = some delay value;
...
#ProgrammedDelay PLLClock = ~PLLClock; // The VFO oscillator.
...
ProgrammedDelay = some new delay value;
...
#ProgrammedDelay PLLClock = ~PLLClock; // Frequency varied.
...
end
```

我们定义比较器给 VFO 的加快频率的指令是 2'b11，减慢频率的指令是 2'b00。

接下来，假设已经定义了用来产生 VFO 基准频率的翻转延迟值，并且延迟值的调整量也定好了，那么，这个 VFO 的 Verilog 代码应该如下：

```
module VFO (output ClockOut, input[1:0] AdjustFreq, input Reset);
reg PLLClock;
real VFO_Delay;
assign ClockOut = PLLClock;
/
always@(PLLClock, Reset)
if (Reset==1'b1)
begin
VFO_Delay = 'VFOBaseDelay;
PLLClock = 1'b0;
end
else begin
case (AdjustFreq)
2'b11: VFO_Delay = VFO_Delay - 'VFO_Delta;
2'b00: VFO_Delay = VFO_Delay + 'VFO_Delta;
// Otherwise, leave VFO_Delay alone.
endcase
#VFO_Delay PLLClock <= ~PLLClock; // The oscillator.
end
endmodule // VFO.
```

为了使延迟值的调整能够立即体现出来，这段代码里用到的几乎都是阻塞赋值。但是我们看到，VFO 的振荡使用了非阻塞赋值。我们应该尽量避免在一个 always 块里同时使用阻塞和非阻塞赋值，对于这样的写法一定要特别谨慎。

比较器的设计要复杂一些。我们希望它既能准确地调整 VFO 的频率，又使用尽量少的资源。如果使用 90 nm 工艺来实现它，它的工作频率可以达到很高。实现这个比较器的一个很简单的办法是：用一个时钟来对另外一个时钟的高电平宽度进行计数。如果每次高电平时计数都为 1，那么说明两个时钟的频率是吻合的。

```
module JerkyComparator
    (output[1:0] AdjustFreq, input ClockIn, PLLClock, Reset);
reg[1:0] Adjr;
assign AdjustFreq = Adjr;
reg[1:0] HiCount;
// 
always@ (ClockIn, Reset)
if (Reset==1'b1)
begin
    Adjr = 2'b01; // 2'b01 or 2'b10 are no-change codes.
    HiCount = 'b0;
end
else if (PLLClock==1'b1)
    HiCount = HiCount + 2'b01;
else begin
    case (HiCount)
        2'b00: Adjr = 2'b11; // Better speed it up.
        2'b01: Adjr = 2'b01; // Seems matched.
        default: Adjr = 2'b00; // Must be too fast.
    endcase
    HiCount = 'b0; // Initialize for next ClockIn edge.
end
endmodule // JerkyComparator.
```

这段代码里仍然使用的是阻塞赋值，没有必要用到非阻塞赋值。比较器工作时两个时钟的相位关系的波形如图 4.3 所示。

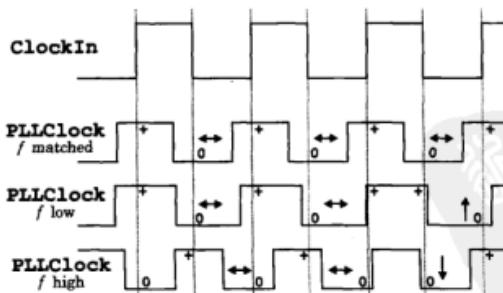


图 4.3 Comparator 的波形。向上的箭头说明 $\text{HiCount} > 1$ ；向下的箭头说明 $\text{HiCount} = 0$ ；水平的双向箭头说明 $\text{HiCount} = 1$

用这个模型来做仿真是没有问题的，在下面的练习里，我们会用一个很类似的模型来仿真。但是，这个模型的锁定时间很长，并且可能会出现假锁的情况。下面，我们来详细讨论如何改进这个模型。

- 首先，应该考虑到HiCount计数可能会溢出，计数值重新翻转到2'b00。2'b00的作用是加快产生时钟频率，这使得频率的调整方向和真实情况正好相反。如果我们把HiCount reg的位宽由2比特扩展至3比特，可以解决这个问题。
- 接着，来解决调整精度的问题。这可以通过把高电平的计数值在多个时钟里求平均的办法来解决；当两个时钟同步时，这个计数值应该一直都是1。

我们将利用下面这个办法来实现对高电平计数求平均。声明一个寄存器类型用来保存这个平均值。在另外一个 always 块里，如果时钟到来时，这个高电平计数超过或小于门限值，则把这个平均值加1或减1。

在完成了上述的升级后，我们实现了一个可以锁定输入时钟的1倍数字PLL。它的功能和传统的模拟PLL很类似。这个设计的代码由两部分组成：第一部分是比较器的代码；第二部分是把比较的结果求平均，用来产生VFO调整的控制信息。

```
module SmoothComparator
  (output[1:0] AdjustFreq, input ClockIn, PLLClock, Reset);
reg[1:0] Adjr;
assign AdjustFreq = Adjr;
// 
reg[2:0] HiCount; // Counts PLL highs per ClockIn.
reg[1:0] EdgeCode; // Locally encodes edge decision.
reg[3:0] AvgEdge; // Decision variable.
reg[2:0] Done; // Decision trigger variable.
// 
always@(ClockIn, Reset)
begin : CheckEdges
  if (Reset==1'b1)
    begin
      EdgeCode = 2'b01; // The value of EdgeCode will be used to
      HiCount = 'b0; // increment or decrement AvgEdge.
    end
  else if (PLLClock==1'b1) // Should be 1 of these per ClockIn cycle.
    HiCount = HiCount + 3'b1;
  else begin // Check to see how many PLL 1's we caught:
    case (HiCount)
      3'b000: EdgeCode = 2'b00; // PLL too slow..
      3'b001: EdgeCode = 2'b01; // Seems matched.
      default: EdgeCode = 2'b11; // PLL too fast.
    endcase
    HiCount = 'b0; // Initialize for next ClockIn edge.
  end
end // CheckEdges.
always@(ClockIn, Reset)
begin : MakeDecision
  if (Reset==1'b1)
    begin
      Adjr = 2'b1; // No change code.
      Done = 'b0;
      AvgEdge = 4'h8; // 7..9 mean no adjustment of VFO freq.
    end
  else begin // Update the AvgEdge & check for decision:
    case (EdgeCode)
      2'b11: AvgEdge = AvgEdge + 1; // Add to PLL edge count.
    end
  end
end
endmodule
```

```

2'b00: AvgEdge = AvgEdge - 1; // Sub from PLL edge count.
// default: do nothing.
endcase
Done = Done + 1;
if (Done=='b0) // Wrap-around.
begin
    if ( AvgEdge<7 )
        Adjr = 2'b11; // Better speed it up.
    else if ( AvgEdge>9 )
        Adjr = 2'b00; // Must be too fast.
    else Adjr = 2'b01; // No change.
    AvgEdge = 4'h8; // Initialize for next average.
end
end // MakeDecision.
endmodule // SmoothComparator.

```

请读者注意 MakeDecision always 块里的 case 语句。它没有 default 分支，并且它没有覆盖到所有的条件项。在综合的时候，这样的代码会导致综合工具产生锁存器，使得综合的结果可能和我们的预期不相符。不过我们只用这个文件来做仿真，而不会去综合它，因此在我们这个工程里可以暂时不管这个问题。

在目录 Lab06/Lab06_Ans 里，有这个 1 倍 PLL 设计的源文件，文件和书中列出的代码可能有些细微的差别。这个模型被设计成最高可以支持 400 MHz 的输入时钟，这几乎达到了用来综合的 90 nm 通用工艺库的上限。当时钟是 400 MHz 时，时钟周期是 2.50 ns，VFO 的半周期延迟是 1.25 ns。testbench 会使得这个延迟缓慢的增大。

图 4.4 和图 4.5 是部分的仿真结果。

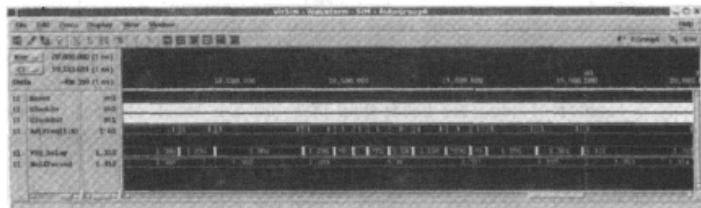


图 4.4 1 倍的 PLL。用 VCS 仿真 20 μ s 的结果里的最后 3 μ s

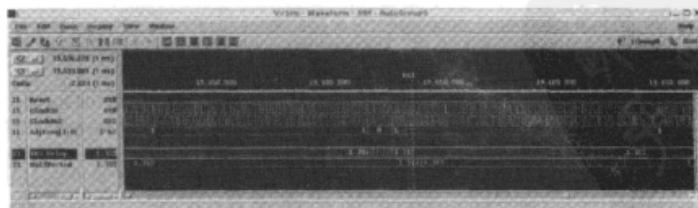


图 4.5 上图中锁定过程的细节，发生在光标 C1 附近

如果读者希望了解更多关于 PLL 消除延迟、PLL 设计静态时序分析等的细节内容，请参考 Zimmer 的文章。本书简介里的参考资料里列出了这篇文章。

4.1.3 SerDes 和 PCI Express 简介

PCI Express (PCIe) 总线是用来取代 PCI (Personal Computer Interconnect) 总线的新的串行总线协议。PCI 总线是 1995-2005 年间在 PC 上非常流行的一种 32 比特的并行总线协议。串行总线回避了并行总线里每个比特信号歪斜 (skew) 不一致的问题，而这个问题正是制约并行总线无法高速工作的原因。PCI Express 和 PCIx 是不同的两种协议，PCIx 仍然是一种并行的总线协议，它仅仅是把 PCI 的时钟进行了倍频处理。

对于工作在 66 MHz 时钟条件下的 PCI 总线，每秒它能传送 $66 \times 10^6 \times 32$ 比特，大约是 2×10^9 比特 (2 Gb/s) 的数据。而 PCI Express 的一个串行通道 (术语叫做 lane) 能以 2.5 Gb/s 的速率同时收发数据。这个传输速度的大幅提升是因为半导体的高速数字信号处理技术取得了突飞猛进的进展。过去的 10 年里，半导体行业飞速发展，原来只能用离散模拟器件实现的很多功能现在都被集成到了一颗芯片中。

完整的 PCI Express 规范最早提出于 2002 年中期，最多支持 32 个传输速率为 2.5 Gb/s 的通道。因此，单向的最大传输速度为 80 Gb/s 。第二版的 PCI Express 规范里将每个通道的单向传输速度提升为 5 Gb/s 。和 PCI 一样，PCI Express 也是用在主板上进行短距数据传输的。例如，它可以用来实现 CPU 到 RAM，显卡或其他 I/O 端口控制器的数据传输。PCI Express 和 PCI 相比，它的优点不仅仅是提供更高的传输速度，而且在进行电路板布线时，PCI Express 的成本也更低。当然，随之而来的是数据管理，串行和解串的设计变得非常复杂。

我们举个例子来说明 PCI Express 的好处。假设一个典型的显示终端的分辨率是 1280×1024 像素，在全彩的 CMYK 模式下，每一个像素点用 32 比特来表示。这样，一个图像帧的数据量为 $1280 \times 1024 \times 4$ 字节，这大约是 5 MB。如果显示终端的刷新率为 70 Hz，则传输速率要达到 350 MB/s。如果有一个并行总线的位宽是 128 比特，假设它的工作频率是 33 MHz，则它的传输速率约为 500 MB/s，可以满足传输的需求。但是，这样的总线会占用很大的面积。我们看到，第二版 PCI Express 规范里的传输速率比这个速度高，由于是串行协议，面积也会小很多。因此，我们看到，无论是从性能的角度还是从经济的角度，PCI Express 都要好很多。

对于时钟频率达到了 GHz 级别的电路来说，会遇到很多模拟信号的问题。例如，每一组串行线实际上是一对差分数据线，两对差分数据线组成了全双工的通道。对于数字设计来说，我们认为这些信号线能够无误地传输信号，线的概念仅仅是信号的通道。而物理上实现这些信号线的时候，可能是不同的形式：简单的传输线，双绞线，甚至是同轴电缆。在做数字设计时，我们不考虑模拟信号的传输问题，而且设计的前提是认为它们没有问题。我们将要实现的串行/解串器包含一个完整的串并接口。和 PCI Express 的规范略有不同，这是一个全数字的接口。这样的好处是使得我们的设计和具体的实现无关。书中的参考文献里列出了一些关于高速信号传输模拟方面的资料，读者可以自行阅读。

对于利用串行/解串器来进行数据传输的两个系统，要发送的并行数据会被先保存下来。必须保证并行总线的长度足够短，从而我们可以不重点考虑并行总线里每个比特的信号歪斜。并行数据往往被保存在一个缓冲区里，通常是一个 FIFO (First In, First Out 的堆栈 memory)。接着，它们被串行化，然后一个一个比特地被发送给目标系统。在接收端，输入的串行数据被解串 (转换成并行数据)，然后被保存到缓冲区，按照并行的格式发送到接收系统的并行总线。

上。为了实现数据的同步，接收端必须从接收的串行数据里提取出数据的时钟信息。这里的时钟其实和接收数据的帧边界是同一个概念。如果串行/解串器的工作状态是稳定的，它内部的 PLL 可以产生和输入的串行数据同步的时钟信号。如果 PCIe 的 lane 的两端都处在同一个时钟域，则两端的电路可以共享一个 PLL；如果 lane 两端的时钟不同步，则需要两个 PLL。

4.1.4 本书里设计的串行/解串器

我们将要开始设计支持全双工模式的串行/解串器，这和 PCI Express 里 lane 的功能类似。这意味着双向的传输可以同时进行，而不会去共享端口和资源。在我们的设计里，只需要两根线就可以实现这个需求，每一根线实现一个方向的数据传输。

我们的 SerDes 的输入是 32 比特的并行数据，这个 SerDes 按字节把并行数据转换成 16 比特一帧的串行数据流。64 个比特组成一个完整的数据包。我们计划每帧只传送一个字节的实际数据，因此这个传输效率低于 PCI Express。但是这样的好处是串行数据里的时钟信息可以很容易的提取出来。由于是串行数据，一次只能处理一个比特。而实际的并行数据位宽为 32 比特，因此需要 PLL 把并行数据的时钟倍频 32 倍。

为了让倍频后的时钟不至于过高，在这个设计里，我们规定并行数据的处理时钟频率为 1 MHz。这样，串行数据的传输速率是 32 Mb/s，大约是 PCI Express 单向 lane 的 1/100。为了将并行数据按字节隔离开，我们在每一个有用字节后都加上了一个字节的帧边界，共同组成了一个完整的帧。因此，32 比特的数据在转换成串行数据之后，是 4 帧共 64 个比特。下面是一个 64 比特串行数据流的例子：

```
64'bxxxxxxxxx00011000xxxxxxxx00010000xxxxxxxx0001000xxxxxxxx00000000.
```

这行数据里的 x 表示实际数据，中间的非 x 数据是帧的间隔（pad）。

倍频后的串行时钟频率是 32 MHz。由于 32 比特有用数据加上包格式之后变成了 64 比特，因此，每两个时钟才能发送（和接收）一个 32 比特的有效并行数据。PCI Express 规范里的数据包可能会大至 128 比特，在我们这个设计里不采用这样的数据格式。

在本书的后面章节里，当我们完成了这个 SerDes 工程之后，会看到利用专门为本书准备的门级库，可以迅速地完成设计的综合和优化。

图 4.6 是并串转换部分的结构示意图。

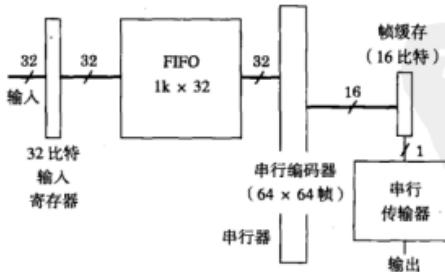


图 4.6 并串转换器的数据流

接收端串并转换部分的结构示意图如图 4.7 所示。

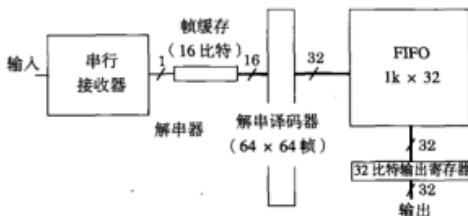


图 4.7 串并转换器的数据流

在 PCI Express 或其他类似的串行/解串器里，接收端从输入串行数据流里提取时钟信息，并把这个时钟信息转换成方波时钟（往往用模拟器件来完成）再送给 PLL。PLL 利用这个时钟来产生的输出时钟，即使输出时钟比输入时钟的频率快很多倍，PLL 也必须保证输出时钟和输入时钟完全同相。

图 4.8 给出了一个内存芯片上的真正的时钟波形。



图 4.8 一个频率为 400 MHz 的好时钟（照片出自 LSI 逻辑展览，Denali MemCon 2004）

这是一个工作在 400 MHz 频率下好时钟的波形。图 4.8 中一个竖格的电压是 300 mV。PCI Express 数据线上的信号波峰到波谷的电压约为 1 V。通过这个图可以看到，当时钟工作在 400 MHz 下时，模拟信号产生的问题开始变得严重起来。

4.1.5 32 倍数字 PLL

在下面的练习里，我们将会实现一个不可综合，但是能够把时钟倍频 32 倍的 PLL Verilog 模型。

接下来，我们将把前面讲到的 1 倍 PLL 的功能由锁相改为锁频，并把修改后的 PLL 运用在我们的 SerDes 工程里。数字同步会同时完成锁相的工作。为了节省写代码的时间，我们将不在计数平均判断锁定的逻辑上花时间了。为了避免比较器对随机相位误差产生响应，我们将产生一个外部采样指示脉冲，仅当这个采样指示脉冲有效时，VFO 才会去调整它自己的频率。否则 VFO 将根据之前的信息自由运行。

我们会把这个设计改写成可综合的设计，虽然综合后产生的网表的功能可能不正确，但是仍然可以综合出这个网表来。在后面的课程里，会重新设计这个 PLL，使它成为一个真正的可综合设计，并且用它综合出来的网表是正确的。现在，我们用这个不可综合的 PLL 模型就可以了。SerDes 工程需要这个 PLL 去锁定利用串行数据产生的时钟。

这个 PLL 的结构示意图如图 4.9 所示，其中 $n = 5$ 。

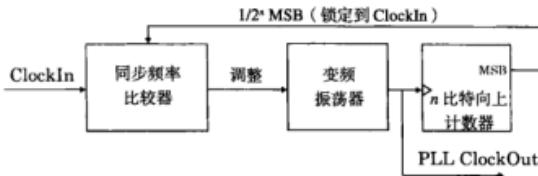


图 4.9 PLL 的方框图，说明了如何将时钟倍频至 $\text{ClockIn} \times 2^n$

一个简单的，可复位的 5 比特向上计数器（从 0 开始递增）的代码是这样的：

```

reg[4:0] Count;
always@(posedge Clock, posedge Reset)
begin
    if (Reset==1'b1)
        Count <= 5'h0;
    else Count <= Count + 5'h1;
end

```

练习中还会有电路图和更多的细节。

4.2 练习 6：PLL 时钟

在 Lab06 的目录下完成下面的练习。

练习步骤

在这个练习里，我们将完成一个 PLL 和对应的并串转换器的设计。由于综合工具不能够实现代码里的延时，所以在实际工程里，为了可以综合整个工程，通常我们会把这种不可综合的设计（这里是 PLL）用一个预先综合过的硬核或模拟的 IP（Intellectual Property）来替代。

除了在 testbench 里和产生时钟时用到了延迟，我们将不在设计这个 PLL 时介绍关于 #delay 的知识。

把这个 PLL 分成 3 个部分，如图 4.10 所示。这个 PLL 被划分成了 VFO，比较器和计数器。

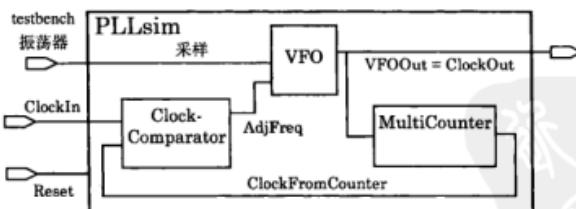


图 4.10 用 Verilog 来实现的这个 PLL，标注了连线的名称，复位信号没有画在上面

第 1 步。先新建一个名为 PLLsim 的顶层模块。在这个模块里例化三个子模块：VFO，Clock-Comparator 和 MultiCounter。和以前一样，每一个模块对应一个独立的文件，名称也是一样的。这个顶层输入有三个输入：ClockIn，Reset 和 Sample，一个输出：ClockOut。

按照图 4.10 中的电路来完成对这个模块的声明。

- 对于子模块 Clock-Comparator，声明输入端口：ClockIn, Reset 和 CounterClock；声明 2 比特的输出端口：AdjustFreq。要实现更精确的频率调整也可以，但是 2 比特对于我们的设计来说已经够用了。
- 对于子模块 VFO，声明 2 比特的输入端口：AdjustFreq, 1 比特的输入端口：SampleCmd 和 Reset，声明输出端口：ClockOut。
- 对于子模块 MultiCounter，声明输入端口：Clock 和 Reset，声明输出端口：CarryOut。

完成了上面的声明之后，按照图 4.10 完成 PLLsim 模块里的端口连线。

你也可以在 PLLsim.v 里添加一个 testbench 模块，并在这个 testbench 里例化 PLLsim。可以利用这个 testbench 进行一些简单的编译以检查连线是否有问题。

第2步。完成 PLL VFO 的模型。这里，我们将完成它的一个模块：它根据名为 AdjustFreq 的 2 比特输入总线的值来对自己产生的时钟进行微小的调整。这个 2 比特总线为 1 时，说明频率不需要调整；如果总线的值为 0，则说明频率需要减慢；如果值是 2 或 3，则说明频率需要加快。

1 MHz 的并行处理时钟说明 VFO 的基准频率是 32 MHz。因此，VFO 的时钟周期为 31.25 ns，半时钟周期为 15.625 ns。由于我们只使用了整型变量，没有办法表达 1 ns 以下的延迟。因此，我们产生的频率是较粗糙而不精准的。

对于这个不可综合的 PLL 模型来说，我们将利用外部指示信号：Sample 对时钟进行采样。每一次对 VFO 频率的调整值将是半周期的 1/16。即每一次 Sample 命令到来时，时钟周期的调整量略小于 1ns。换句话说，如果 AdjustFreq > 2'b01，VFO 的时钟周期将减少约 1ns；如果 AdjustFreq < 2'b01，VFO 的时钟周期将增加约 1 ns。

Verilog 允许把参数声明成实型（real）或有符号的（signed），这对我们实现这个 PLL 模型是有帮助的。但是很多仿真工具并不支持用参数来保存实型数。为了代码的灵活性，我们把需要用到的参数都放到一个单独的 include 文件中去，这个文件的名字叫 PLLsim.inc。

```
'timescale 1ns/100ps
`define HalfPeriod32BitBus 500.0 // ns half-period at 1 MHz.
`define VFOBaseDelay 'HalfPeriod32BitBus/32.0 // At 32 MHz.
`define VFO_DelayDelta 1 // ns.
`define VFO_MaxDelta 2 // ns.
```

最后，需要防止由于 PLL 产生的频率过高从而跑飞或产生的频率过低从而停止输出时钟，在你的 VFO 模型里加上频率的限制，使它不会远离基准频率。

由于定义的宏不能够在仿真过程中改变（实际上，在编译的时候，它们被仿真工具用实际值替换掉了），因此，我们会使用它给模型中的变量赋初值。这样就可以根据自己的需要将这些宏改成整型或实型。在后面的章节里，当需要进行仿真的时候，我们会把这些宏改成实型；如果需要进行综合，则把这些宏改成整型。

接下来，在需要使用这些宏定义的文件的头部加上下面这句话：

```
'include "PLLSim.inc"
```

只需要在 PLLsim.v 和 VFO.v 这两个文件的头部加上上面这个命令。其中，PLLSim.v 会在仿真时把时间精度设置传递给其余的子模块。有两点需要注意：(a) 不管是并行时钟还是串行时钟，

都只和`HalfPeriod32BitBus这一个值有关; (b) 在指定这个值的时候用的是小数。在 Verilog 里, 这是强制让延迟按实型进行计算的一种方法。如果没有这些小数点, 除法的结果会被四舍五入成整数。当然, 如果进行的是一个可综合的设计, 所有的变量都必须是整数。

在这之后, 在 VFO 里把下面这段代码里除了 VFO_ClockOut 的信号都声明成整型 (integer), 并且加上这段初始化的逻辑。

```
always@ (Reset, SampleCmd, VFO.ClockOut)
if (Reset==1'b1)
begin
    BaseDelay      = `VFOBaseDelay;
    VFO.Delta      = `VFO.DelayDelta;
    VFO.MaxDelta   = `VFO.MaxDelta;
    VFO.Delay     = `VFOBaseDelay;
    VFO.ClockOut = 1'b0;
end
else (below)
```

下面这段代码是上面的 else 部分。这部分代码可以调整 VFO 的半周期延迟。

```
else // as above.
if (SampleCmd == 1'b1)
begin
if ( AdjustFreq>2'b01
    && (BaseDelay - VFO.MaxDelta < VFO.Delay) )
    // If floor is lower than current:
    VFO.Delay = VFO.Delay - VFO.Delta;
else if ( AdjustFreq<2'b01
    && (fill this in)
    // else, leave VFO.Delay alone.
```

由于 VFO_MaxDelta 的限制, 可以去掉有两级 if 的 case 语句。

利用这里得到的 VFO_Delay, 就可以产生 PLL 的时钟了。

```
always@ (Reset, SampleCmd, VFO.ClockOut)
if (Reset==1'b1)
    ... (as above) ...
else begin
    ... (freq. adjustments) ...
    `ifndef DC
    // No delayed nonblocking assignments:
    #VFO.Delay VFO.ClockOut = ~VFO.ClockOut;
    `else
    #VFO.Delay VFO.ClockOut <= ~VFO.ClockOut;
    `endif
end // main else.
```

由于综合工具不支持有延迟的非阻塞语句, 因此我们在产生振荡时钟时多增加了一个 `ifndef 的条件编译选项。当我们要综合这段代码时, 可以先定义 DC 这个宏。

在这个 always 块的敏感变量里加入 SampleCmd 的原因是希望每当收到了这个命令, 就会调整 PLL, 使输出时钟和输入时钟同步。

在完成了 VFO 的设计之后, 用 PLLsim 来对它进行仿真。产生一个临时时钟, 并把它送给比较器, 让比较器产生 0~3 的调整量。观察 VFO 是否正确使用了这个调整量产生了时钟。

第3步。完成PLL的比较器。如图4.3所示，模块Clock-Comparator比较了两个时钟之间的频率差别，并且根据这个差别来调整输出时钟频率，使得两个时钟频率相同。

由上面的VFO设计，仅当VFO收到了采样命令时，VFO才会去调整它产生的时钟。因此，不用把采样命令送给Clock-Comparator，但是必须保证VFO能够周期性的收到采样命令。

为了保证Clock-Comparator里的寄存器在开始工作的时候处在确定的状态，Clock-Comparaotr必须有复位输入。

实现一个纯数字的频率比较器有很多种办法。我们这样做：在每一个输入时钟的高电平周期里数输出时钟翻转的次数。

总共有三种情况：

- 如果数到的翻转次数大于1，则说明输出时钟频率应该降低。
- 如果数到的翻转次数等于1，则说明输出时钟频率不需要调整。
- 如果数到的翻转次数等于0，则说明输出时钟频率应该加快。

为了实现这个功能，声明一个名为VarClockCount的2比特寄存器类型变量，用它对PLL的时钟(CounterClock)边沿进行计数。2比特的位宽支持计数到3，如果两个时钟频率相差不大，这个范围是足够的。为了避免VarClockCount溢出的情况，也可以把它声明成32比特的Verilog整形变量，但是没有这个必要。

完成后的ClockComparator的代码如下：

```
...  
always@( ClockIn, Reset ) // This is the synchronizing clock.  
Begin  
if (Reset==1'b1)  
begin  
    AdjustFreq = 2'b01;  
    VarClockCount = 2'b01;  
end  
else // CounterClock is the clock to synchronize. Notice that it  
// is not on the sensitivity list; the inferred latch may be  
// expected to cause synthesis problems.  
if (CounterClock==1'b1)  
    VarClockCount = VarClockCount + 2'b01;  
else begin  
    case (VarClockCount)  
        2'b00: AdjustFreq = 2'b11; // Better speed it up.  
        2'b01: AdjustFreq = 2'b01; // Seems matched.  
        default: AdjustFreq = 2'b00; // Better slow it down.  
    endcase  
    VarClockCount = 2'h00; // Initialize for next ClockIn edge.  
end
```

需要注意的是，在这个always块里，我们是在每个外部输入时钟的边沿去完成比较，而不是用PLL产生的时钟作为触发比较的条件。这是因为PLL产生的时钟正是我们要调整并使它和输入时钟同步的信号。也可以把VarClockCounter的计数值作为控制信号直接输出给VFO，但是在这个模块里的处理增加了设计的自由度，可以根据需要来修改调整量而不用去改VFO的代码。

当PLL的输出时钟CounterClock为高时，我们在每一个ClockIn发生翻转时把计数加1；在CounterClock为低时，根据其为高电平时的计数来产生适当的调整量。如果这个计数等于0，

则说明 CounterClock 快了；如果计数等于 1，说明时钟是同步的；如果计数大于 1（输出时钟为高时至少采样到了两个输入时钟的变沿），则说明输出时钟慢了。

完成了这些步骤之后，用 testbench 直接例化 PLLsim，把 VFO 和 Clock-Comparator 放在一起仿真。记得在仿真之前先产生 Clock-Comparator 需要的复位信号。

第 4 步。完成 PLL 倍频计数器。这是一个每 32 个时钟就溢出一次的简单计数器。Thomas and Moorby (2002) 详细介绍了计数器建模的细节（例如，2.2 节至 2.6 节，6.5.3 节和 6.7 节）；我们在后面的章节里会深入地讨论计数器的结构。

一个典型的行为级向上计数器的 Verilog 代码如下：

```
reg[HiBit:0] CountReg
...
always@(posedge ClockIn, posedge Reset)
  if (Reset=='1'b1)
    CountReg <= 'b0;
  else CountReg <= CountReg + 1'b1;
```

注意：不能直接用参数来指定信号的位宽。也就是说，“BitHi'b1”的写法是不行的。如果我们不指定位宽，例如，写成 CountReg <= CountReg + 1，在语法上是没有问题的。但是工具会默认最后那个 1 的位宽是 32 比特。因此，应该指明位宽。这样工具在进行编译的时候处理起来更容易。

把计数器的 MSB 定义成进位比特。利用这个信号，可以每隔 32 个时钟产生一个进行输出。由于这个计数器是 5 比特的，因此，这个进位比特每隔 16 个时钟就会翻转一次。这个计数器 always 块的敏感变量是时钟的上升沿和复位的上升沿。

第 5 步。测试完整的 PLL。在顶层加入了这个计数器之后，这个 PLL 的设计就是完整的了。在 testbench 里，在每个 ClockIn 的上升沿之后都产生 Sample 正脉冲。从顶层 PLLsim 一级验证这个 PLL 设计，部分结果如图 4.11 和图 4.12 所示。你可以改变图 4.10 里的输入时钟 ClockIn，观察 CarryClock 和 ClockOut 是怎么逐渐锁定 ClockIn 的。

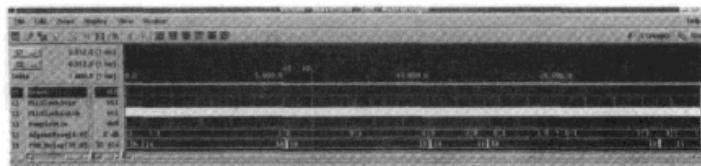


图 4.11 PLLsim 的仿真结果

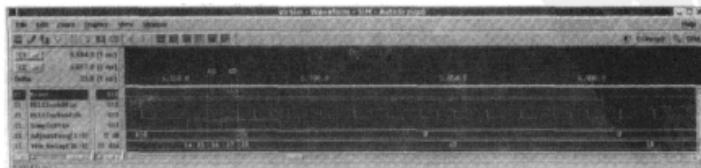


图 4.12 时间单位放大后的波形图

到这里，本章里关于 PLL 的内容都已经结束了。这个练习里其他的内容都是关于其他话题的了。

对于 PLL 里用到的 VFO 模块来说，在进行后端设计时，它是一个有固定面积和引脚分布的硬核，例如一个 IP。本书不介绍布局布线，因此我们不会涉及有关硬核的话题。如果我们需要综合包含这个设计的工程时，可以给这个 VFO 模块加上 `set_dont_touch` 的指令，让综合工具不去综合它。否则，我们得到的 PLL 网表也不能正常工作。

对于任何一个层次的实例来说，都可以用 `set_dont_touch` 来避免它被打平或被优化（参见附带光盘里的综合命令总结）。`set_dont_touch` 是个很有用的命令，可以把它放到综合脚本里，也可以把它放在 Verilog 代码里。

如果把它放在综合脚本里，可以在修改综合策略的时候不去修改 Verilog 的源代码。如果 `set_dont_touch` 的实例是固定不变的，那么，把 `set_dont_touch` 放在设计里也是一个很好的办法。

32x 数字 PLL 的 Verilog 设计到这里就完成了。我们将会用这个设计来进行 SerDes 的仿真，并且还会重新设计它以满足综合的需求。

第6步。完成并串转换器的模型。把这个转换器的名字叫做 `ParToSerial`，接下来会详细介绍这个转换器的设计。这个并串转换器的并行数据位宽是固定的，输入时钟是串行时钟 `SerClock`。假设并行数据线上的数据都和时钟同步，我们还需要一个输入有效的指示信号来指示并行数据有效且是稳定的，端口如图 4.13 所示。

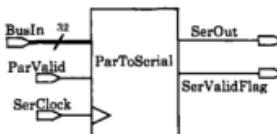


图 4.13 并串转换器

当 `ParValid` 为高时，并行数据会被转成串行数据发送出去。但是当数据已经发送完，`ParValid` 仍然为高时，不再发送任何数据。更不会重复地把已经发送的数据再次发送出去。

串行的协议很简单：每一个时钟发送一个比特，先发送 MSB，同时把信号 `SerValidFlag` 拉高；当发送了 LSB 之后，再把 `SerValidFlag` 拉低。

我们已经知道了这一条编码规则：不要在多个 `always` 块里对一个数据对象进行赋值。这不仅仅是完成一个优秀设计的基本要求，同时也是综合的要求。由于每一个串行时钟都要输出数据，因此这个 `always` 块里的敏感列表里应该包含串行时钟。

我们可以在设计里加上一个 `Done` 标志来表示串行化的过程是否已经完成。这实际上是设计了一个非标准的状态机：当 `ParValid` 刚拉高的时候，实现了一个从完成到未完成的状态转换；当串行数据都已经发送完成的时候，又返回到完成的状态。直到 `ParValid` 有效之后，再次进入未完成的状态，再次开始发送串行数据。换句话说，信号 `Done` 在 `ParValid` 无效时被清零。我们会在本书的后面章节讨论状态机的设计。

基于 32 比特的总线位宽，下面是实现上面所讲功能的代码：

```

module ParToSerial (output SerOut, SerValidFlag
                     , input SerClock, ParValid, input[31:0] BusIn);
    integer ix;
    reg SerValid, Done, SerBit;
    assign #1 SerValidFlag = SerValid;
    assign #2 SerOut = SerBit;
    always@(posedge SerClock)
        begin // Reset everything unless ParValid:
        if (ParValid==1'b1)
            if (SerValid==1'b1)
                begin
                    SerBit <= BusIn[ix]; // Current serial bit.
                    if (ix==0)
                        begin
                            SerValid <= 1'b0;
                            Done      <= 1'b1;
                        end
                    else ix <= ix - 1;
                end // SerValid was asserted.
            else begin // No start yet:
                if (Done==1'b0)
                    begin
                        SerValid <= 1'b1; // Flag start on next SerClock.
                        ix      <= 31; // Ready to start on next SerClock.
                    end
                SerBit <= 1'b0; // Serial bit default.
            end
        end // ParValid not 1; reset everything:
        begin
            SerValid <= 1'b0;
            Done      <= 1'b0;
            SerBit   <= 1'b0; // Serial bit default.
        end // if ParValid else
    end // always
endmodule // ParToSerial.

```

可以用参数来替换上面代码用到的总线位宽，请读者同时也完成它的testbench。在本书附带光盘目录 Lab06 下提供了专门用来让读者进行修改的文件：ParToSerial_unfinished.v。

完成这个并串转换模块的设计，用ANSI风格的Verilog参数来定义并行数据总线的位宽。这样，当我们要修改这个位宽时就很方便了。

把整型的ix声明成reg类型。在设计它的位宽的时候，应该让它尽量小且又能满足数据传输的要求。图4.14和图4.15是典型的仿真结果。

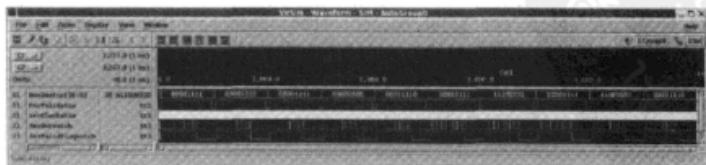


图 4.14 并串转换器的仿真结果

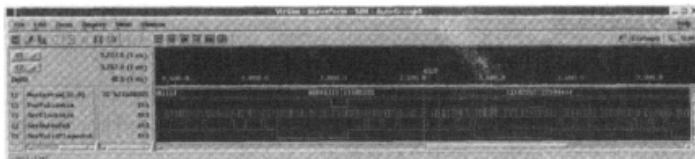


图 4.15 时间单位放大后的波形图

第7步。串行帧编码器。作为本练习里最后的内容，将完成一个编码器，这个编码器把输入的并行数据转换成更宽的包含帧边界信息的并行数据。这是数据再被串行化传送出去之前的最终格式。

参考本章前面的串行器结构图。在每一个输入时钟的上升沿都采样并行输入数据。由于每个有用数据字节之后，都插入了一个字节的帧边界，因此 32 比特的实际数据最后扩展成为了 64 比特的数据，如图 4.16 所示。

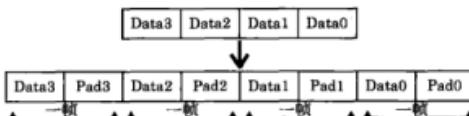


图 4.16 串行/解串器的包格式。每一个原始数据后都加上了帧边界

请读者回忆前面讲到的帧格式，每一个帧的边界里都包含了当前帧是 32 比特实际数据里第几个字节的信息。

由于 32 比特的数据包含 4 个字节，因此，这个计数值只需要用 2 个比特来表示。由于帧的边界有 8 个比特，我们在这个计数值左右各填 3 个 0 组成一个字节。下面是帧边界的具体格式：边界 0 = 8'b000_00_000；边界 1 = 8'b000_01_000；边界 2 = 8'b000_10_000；边界 3 = 8'b000_11_000。

这个串行数据流的格式如下（x 代表原始的输入数据）：

64'bxxxxxxxxx00011000xxxxxxxxx00010000xxxxxxxxx00001000xxxxxxxxx00000000.

设计一个名为 SerFrameEnc 的模块，它的作用是在每个输入时钟到来的时候把输入的并行数据编码成上面的格式。在写代码的时候，利用参数来指定输入输出的位宽。仿真这个设计并检查结果，图 4.17 是仿真的波形。仿真之后综合这个模块，先后按面积和速度的优化原则来综合这个设计。

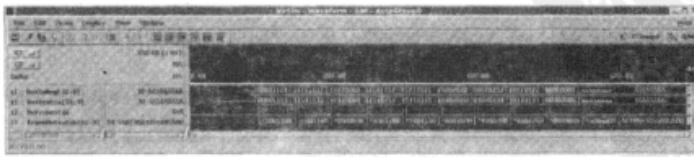


图 4.17 SerFrameEnc 的仿真结果，生成网表按速度优化

4.2.1 练习后的思考

有很多个模块的设计里，`timescale 应该放在什么地方？

Verilog 怎样区分整型和实型常数？

怎么修改这个比较器使得它在每个时钟进行一次比较，而不是每个时钟的跳变沿都进行比较？如果这样改了之后，PLLsim 里定义的常数应该怎么改？

这个帧编码的设计看起来效率很低，总共用了 64 比特才能实现对 32 个有用数据的编码。怎样设计才能使编码的效率更高？对于工作速度达到 2 Gb/s 的 PCI Express 的串行/解串器来说，这个效率是否真的非常重要？我们将来会接着讨论这个问题。

4.2.2 补充学习

在 2008 年里，串行/解串器是一个很热门的话题。读 Freescale 网站的一篇综述串行/解串器的文章：<http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01HGpJ2350NbkQ> (2004-11-16)。

下面这篇文章介绍了关于 PLL 的基本常识以及它的发展历史，作者是 Ron Bertrand's，“The Basic of PLL Frequency Synthesis”，源自无线电和电子在线课程：<http://www.radioelectronicschool.com/reading/pll.pdf> (2004-12-09)。

(可选) 对于本章里涉及的模拟信号的问题，可以读 S. Seat 的文章，“Gearing Up Serdes for High-Speed Operation”，http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=16504769 (2004-11-16)。

(可选) 下面是两篇非常深入地讨论到了串行/解串器模拟设计的相关问题的文章，尤其是它讲到了 PLL 的内容，作者是 E. H. Suckow，“Basics of High-Performance SerDes Design”，Part I，http://www.analogzone.com/iot_0414.pdf；Part II，http://www.analogzone.com/iot_0428.pdf (2004-11-16)。

(可选) Kevin Edwards 在 EuroDesign Con 2004 会议上发表了一篇非常精彩的关于 PCI Express 文章，介绍了 PCI Express 协议和相关系统架构的内容，“PCI Express - IP for a Next-Generation I/O Interconnect”。这篇文章非常符合 IP 的规范。在 Mentor Graphics 的网站免费注册后，可以读到，http://www.techonline.com/community/tech_group/embedded/tech_paper/37455 (2005-06-11)。

(可选) S. Knowlton 的文章列举了在进行 PCI Express 设计时会遇到的系统架构和模拟信号传输速度的问题，但是这篇文章只对购买了 Synopsys 的正版用户才可见。Knowlton 介绍了 PCI Express 里的每一个组件，介绍的方式和本章的方式很接近。他还介绍了关于包缓冲，ECC 功能和 SerDes 的相关设计；而在 PCIe 标准里，SerDes 的设计属于模拟信号领域的范畴。

第5章 存储与数组

5.1 数据存储与 Verilog 数组

本章中，将学习数据的存储与数据的完整性，我们还会用 Verilog 数组进行存储器的建模。

5.1.1 存储器：软硬件描述

存储器（memory）可以保存数据或其他信息以供将来使用。从硬件范畴上来讲，存储器可以被分为两大类：随机存取（random-access）和序列存取（sequential-access）。

随机存取存储器（Random Access Memory, RAM）是计算机最常用的存储器芯片。对于 RAM 来说，用相同的操作方法可以访问它任何一个存储地址。每个存储地址上的数据位宽相等，读取每个存储地址里的数据所花的时间也是相同的。这类存储器中常见的有：EPROM、DRAM、SRAM、闪存 RAM 和 ROM 等。

序列存取存储器（Sequential Access Memory）包括磁带、软盘、硬盘、光盘，例如 CD 和 DVD。序列存储的过程随着存储地址的变化而变化。例如，当磁带播放到末尾的时候，必须循环播放才能找到下一个数据；又或者，当存取地址发生变化时，磁盘读/写的头部必须重新定位。

这里将不讨论序列存取的存储器模型，因为 Verilog 里没有这样的模型。但是，Verilog 里有针对 RAM 的数组模型。

一个小问题：关于 RAM 芯片的存储容量，文献里和数据手册里的描述有些不同，容易混淆。硬件数据手册在描述存储容量时通常给出地址的总个数（不包括错误检测或奇偶校验的部分）以及一个地址的存储位宽：地址数（storage in bits） \times 位宽（word width）。因此，一个标识为“256k \times 16”的 RAM 芯片可以存储 $256 \text{ kb} = 256 \times 1024 = (2^8 \times 2^{10}) = 2^{18}$ bit。如果要计算地址空间，只需要用容量除以位宽即可。例如，对于上例的存储容量，如果存储位宽为 1，则需要 18 根地址线；如果存储位宽的单位是字节，则地址总线的宽度为 $(2^{18}/2^3) = 15$ 比特；如果保存的是芯片硬件定义的 16 比特双字节，则地址总线的宽度为 14 比特。

计算机的硬件制造商往往会用多个 1 比特或 4 比特字宽的 memory 并行拼成和计算机字宽相匹配的 memory。例如，8 个“256kb \times 1”的 DRAM 芯片可以组成一个 256 kb 的存储器。如果存储器需要支持奇偶校验，则共需要 9 个这样的 DRAM。显然，32 个这样的芯片就构成了 1 MB（兆字节）存储器。

在 Palnitkar (2003) 的附录 F 中介绍了一种 DRAM 存储器模型。作者用软件方法描述存储器的容量：字的数量 \times 字宽。一个这样的 DRAM 芯片可以存储 $(256 \times 16) \text{ kb} = (256 \times 1024) \times (16) = 2^{(8+10)+4} = 2^{22} = 4 \text{ Mb}$ （兆比特），其中， $1 \text{ Mb} = 1024 \times 1024 = 2^{20}$ bit。为了准确的对存储器进行建模，必须要准确理解描述的含义。

5.1.2 Verilog 数组

之前讨论的都是 Verilog 中的向量。向量的声明是在 Verilog 预定义的类型后面加上一个范围。例如，`reg [15:0]` 表示 16 位的寄存器型变量。

Verilog 中数组也是用范围表达式定义的。但是，数组的范围跟在数组名的后面。照惯例，数组的范围和向量的范围是分开的。例如，“`reg [15:0] WideMemory [1023:0];`” 定义了一个名为 `WideMemory` 的存储器，它包含 1024 个地址，每个地址上可保存一个 16 比特（字类型）的 `reg` 类型变量。

注意，数组范围中如果冒号后面那个数是 0，冒号前那个数是地址容量减 1。`WideMemory` 地址总线的位宽是 10 比特。

定义 Verilog 数组的语法如下：

```
reg [vector log indices] Memory_Name[array location indices];
```

虽然不常用，但是在 `memory` 中也可以使用有符号的 `reg` 类型，例如 `integer`。

下面是一个关于存储器地址的例子。

```
reg[7:0] Memory[HiAddr:0]; // HiAddr is a parameter >= 22.
reg[7:0] ByteRegister;
reg[15:0] WordRegister; // This vector is 16 bits wide.
...
ByteRegister      <= Memory[12]; // Entire memory word = 1 byte.
WordRegister     <= Memory[20]; // Low-order byte from the memory word.
WordRegister[15:8] <= Memory[22]; // High-order byte from the memory word.
...
```

硬件 RAM 有地址的限制，例如，CPU 对 `memory` 的访问只能以字为单位，并且访问的时候不能仅读取一个比特或者字的一部分比特，而是必须读取整个字。Verilog 的存储器模型也曾经有同样的限制，不能以比特或片段为单位去访问一个存储器的数据（数组对象），除非每个字是逐比特保存的。但是从 Verilog-2001 开始，就没有这样的限制了。

假设一个存储器字宽为 64 比特，但系统字宽为 32 比特，下面的这段代码在 Verilog-2001 和 Verilog-2005 里是合法的。

```
reg[63:0] Memory[HiAddr:0]; // HiAddr is a parameter > 56.
reg[7:0] ByteRegister;
reg[31:0] WordRegister; // This vector is 32 bits wide.
...
ByteRegister      <= Memory[57]; // Entire memory word (truncated).
ByteRegister     <= Memory[50][15:8]; // 2nd byte from a memory word.
Memory[56][63:32] <= WordRegister; // To the high half of memory word 56.
...
```

在声明存储器时，向量和数组的范围是被分别指明的。但使用存储器时，向量和数组的大小范围跟在存储器名后面。存储器地址紧跟在定义的存储器名后面，后面接着是向量范围。选取比特的时候是从右边开始数的，和普通的比特选取的操作是一样的。

Verilog (Verilog-2001) 支持多维数组，例如：

```
reg[7:0] MemByByte[3:0][1023:0];
```

这条代码定义的存储器包含 1024 个 32 比特的对象，每个对象是 4 个字节。或者说，这个存储器包含 4096 个 8 比特的对象。因此，“ByteReg <= MemByByte[3][121];”这条语句读出的是保留在地址为 121 上的最高位字节的数据。这里的 “[3][121]” 是地址而不是截取。在 Verilog 中，不能使用变量来选取一部分比特，只能用变量来选择某一个单独的比特。由于上例中序号的含义是地址，因此，在地址的索引表达式里可以使用变量。

多维数组允许使用任意维数，可是一般很少超过三维。可以采取部分选取的办法把维数降低一维。当然，这种部分选取只有当索引表达式是常量时才合法（必须全是确定的数、参数或者常量表达式）。例如：

```
reg[7:0] Buf8;
reg[7:0] MemByByte[3:0][1023:0]; // 2-D (call byte 3 the high-order byte).
reg[31:0] MemByWord[1023:0]; // 1-D.
integer i, j;
...
i = 3;
Buf8 <= MemByByte[i][j]; // High-order byte (3,j) stored in Buf8.
Buf8 <= MemByWord[j]; // Low-order byte stored.
Buf8 <= MemByWord[j][31:24]; // Part-select; high-order byte stored.
Buf8 <= MemByWord[j][(i*8)-1:(i-1)*8]; // ILLEGAL! i is a variable!.
...
```

Thomas and Moorby (2002) 在附录 E.2 中讨论了多维数组的问题。

另外，在一个表达式里同时访问存储空间的多个位置是非法的。例如，声明了这样一个 memory： reg [7:0] Memory [255:0]，像这样的代码：“HugeRegister <= Memory [57:56];”这是非法的；又如，定义了 reg [31:0] MemByByte [1023:0]，代码为 “MyIllegalByte <= MemByByte [121:122][31:28];”。这样的代码跨越了地址界限，想从两个不同的地址各取出 4 个比特来，这样的写法也是非法的。对 memory 的读写操作一次只能访问一个地址，和普通向量的单比特选取类似，地址可以使用变量。

上一段话的最后一句说明了不能直接对整个数组对象进行赋值；可以采用循环的方式对它进行赋值，每次访问一个地址。

对 Verilog 的存储器访问有下面的一些限制：

- 每次只能访问一个地址。
- Verilog-2001 中单比特选取和部分选取都是合法的，但不同工具的实现可能不一致。
- 不能用变量进行部分选取或单比特选取。
- VCS 和 Silos (演示版) 都不能显示 memory 的波形，而 QuestaSim 和 Aldec 可以。

因此，如果要对 memory 进行数据存取的操作，应该先访问 memory 的一个地址，再将这个地址保存的数据赋给一个向量，这个向量值的波形可以被显示出来，也可以根据要求对其进行单比特选取或部分选取。这种方式对仿真工具和综合工具都是有效的。例如：

```

parameter HiBit = 31;
reg[HiBit:0] temp;           // The vector.
reg[HiBit:0] Storage[1023:0]; // The memory.
reg[3:0] BitNo;      // Assigned elsewhere.
...
temp = Storage[Addr];
HiPart = temp[HiBit:(HiBit+1)/2]; // A parameter is a constant.
LoPart = temp[((HiBit+1)/2)-1:0];
HiBit = temp[BitNo];        // Bit-select by variable is allowed.
...

```

5.1.3 一个简单的 RAM 模型

RAM 模型只需要以下几个条件：Verilog 存储器、地址与读写控制信号。例如：

```

module RAM (output[7:0] Obus
            , input[7:0] Ibus
            , input[3:0] Adr, input Clk, Read
            );
reg[7:0] Storage[15:0];
reg[7:0] ObusReg;
//
assign #1 Obus = ObusReg;
//
always@(posedge Clk)
if (Read==1'b0)
    Storage[Adr] <= Ibus;
else ObusReg     <= Storage[Adr];
endmodule

```

5.1.4. 位拼接操作符

现在，来介绍一种 Verilog 的语法：位拼接(concatenation)。可以使用位拼接操作符“{…}”把 1 个或多个比特的向量拼接在一起，组成一个新的位宽更宽的向量。这样做可以不用频繁地去声明临时变量。如果要保存一个变量值，只需要把它赋值到一个足够宽的向量上去就可以了。

例如，下面的代码是在 8 比特的数据的 MSB 前面增加一个奇偶校验位，构成一个 9 比特的数据。

```

reg[7:0] DataByte;          // The 8 bit datum, without parity.
reg[8:0] StoredDataByte; // High bit will be 9th (parity) bit.
...
StoredDataByte <= {"DataByte", DataByte}; // A 9-bit expression.

```

同样地，一个字 (word) 可以用两个字节拼接而成，如：Word <= {HiByte, LoByte}；

5.1.5 存储器数据的完整性

这是一个目前很热门的话题，它包含的内容很多。本节中，只给出奇偶校验的代码实例，但会介绍检错和纠错 (ECC) 的原理。

由于硬件自身的缺陷或外界的干扰，例如射频 (RF) 或核辐射，硬件可能会发生随机故障。仅用 Verilog 是无法对这些故障进行讨论，但在本书开头的补充资料里可以查到相关的背景知识。

错误检查 (Error Checking) 通常可以是对各个地址进行奇偶校验，可以是计算校验和，也可以是对某些数据比特位的特殊值进行检查的方法来实现。当某个比特位的数据出了错，可以检测到这个错误并警告使用者或者进行纠错。基本的原理是额外保存检测数据，当出现问题时，只要数据和检测位没有同时出错，这个错误就能够被检测出来。

奇偶校验 (Parity Checking) 在 RAM 里很常用。每个地址的比特 1 (或者 0) 的数目都被记了下来，然后每个地址分配一个额外的比特，根据 1 (或者 0) 的比特数目将它赋值为 1 或 0。

显然，1 (或 0) 的个数不是偶数 (偶校验) 就是奇数 (奇校验)。偶校验里，奇偶校验位使得 “1” (或 “0”) 的总数为偶数。而奇校验里为奇数。通常，到底是偶检验还是奇校验，或者到底是统计 1 的总数还是 0 的总数并没有实质的区别。因此，从现在起，只讨论对 1 的偶校验。这意味着有偶数个 1，也就是说它们的和为 0。因此，保存一个字节需要 9 个比特 (8 个数据比特 + 1 个校验比特)。如果其中任何一个比特发生了错误，错误就能被检测出来。但是如果两个比特都发生了错误，这种错误就检测不到了。

如果忽略进位，可以用异或方法计算校验和。因此可以用 Verilog 里的异或操作符 (^) 来计算。例如：

```
reg[HiBit:0] DataVector;
reg[HiBit+1:0] DataWithParity;
...
// Compute and store parity value:
DataWithParity = {^DataVector, DataVector};
// Check parity on read:
DataVector = (^DataWithParity==1'b0)
    ? DataWithParity // Parity bit is discarded.
    : 'b0; // Assign zero on parity error.
```

奇偶校验的速度很快，而且也不影响运行速度。另外，它的实现很简单，只需要一个简单的异或操作符 (^) 就可以自动求出所有比特的奇偶和值。因此，奇偶校验被广泛地应用在硬件设计中。

因此，我们看到，要对 memory 进行奇偶校验建模是件很容易的事。

对于大型的数据对象，如串行数据帧或磁盘上的文件等，通常会使用校验和 (checksum) 的方式。校验和通常是计算对象里所有 1 (或所有字节) 的和。和奇偶校验不同，校验和的结果是和，而不是单比特的异或和值。如果数据发生变化使得和值发生改变，这样的错误就能被检查出来。例如对比特求和，一个单词里的 ASCII 码 “a” 变成了 “b”，错误就被检查到了；如果是对字节进行求和，文件里少了一个 ASCII 码字，错误也能被检查到。

与奇偶校验不同，校验和的值通常和被检验数据分开存放。它一般用于简单的差错检测或差错检测纠正 (ECC) 编码。

计算校验和值时，有以下几种方式：

- 计算各字节、各帧或者各个数据包的和值。
- 计算各比特的和值。
- 计算 “1” 或者 “0”的和 (如果没有进位，则和奇偶校验一样)。
- 计算某种编码的和值。例如 CRC (循环冗余校验)。在硬件上实现 CRC 的一种方式是采用线性反馈移位寄存器 (LFSR)。

为了降低漏检的概率，比如说两个比特或字节同时出错，可以对存储的数据进行部分编码。例如，在传输串行数据时可使用 LFSR 为每个对象计算类似于和值的参数。LFSR 中采用异或门（xor）将当前值与延时之后的某个反馈值进行异或，如图 5.1 所示。

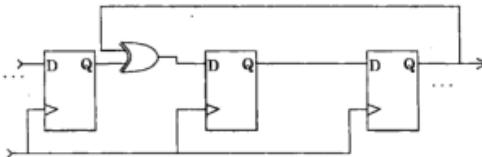


图 5.1 通用三阶 LFSR，xor 用当前值与反馈值进行运算

每当存储对象被访问时，它在寄存器里进行移位，移位结果和保存的值进行比较。移位会产生延时，但除此之外不会再有其他的延时。

Thomas and Moorby (2002) 详细讨论了 CRC 的原理。他们给出了如图 5.2 所示的例子。

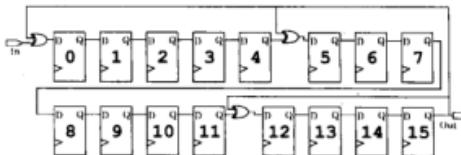


图 5.2 LFSR 的硬件特征多项式。Q[15:0]的值代表 Thomas and Moorby (2002) 的 11.2.5 节里定义的 $x^{16} + x^{12} + x^5 + 1$ 。有效存储数据对这个多项式做模除法，返回值为 0。为了简略，省略了所有的时钟

5.1.6 差错检测与纠正

差错检测与纠正（Error Checking and Correcting, ECC）不仅能检错，还能在一定程度上纠错。所有的 ECC 算法都能找到并纠正数据对象（如存储器空间）中的单个错误。有些 ECC 算法能够检测两个或更多的错误，并能纠正一个以上的错误。纠错是由硬件完成的，几乎不用什么时间。所有商用计算机的 RAM 芯片都有内建 ECC 功能。硬盘和光盘也常有 ECC。

ECC 的思想是建立在校验和之上的。数据和它的校验值存放在不同位置，读取数据时将它与校验值进行比较，如果有一个比特位改变了，这个错误能够被定位。因此，计算机或其他器件能够将其改正，输出正确的值。

这里，仅仅简略概括地介绍如何用奇偶校验实现 ECC，在补充阅读里有更详细的内容。

源于奇偶校验的 ECC。令 pT 为 8 比特数据的奇偶校验位，假设只有一个比特位会出错，并且假设使用偶数个“1”的校验规则。如果奇偶校验出错了，这个数据对象（8+1 比特）就是错的，由于没有办法纠正这个错误，只有不使用这个数据。

但是，假设计算了两个奇偶校验位， $p1$ 是低四位（0~3）的校验位， pT 是 $p1$ 与 8 比特数据的校验位。如果 pT 出错，但 $p1$ 没错，器件会认为错误是出在高四位（4~7）或 pT 里，如果 $p1$ 和 pT 都出错，错误可能是出在前四位（0~3）或 $p1$ ，但不会是 pT 。如果 $p1$ 出错，但 pT 没错，可以肯定至少发生了两个错误，而在这个例子中不会考虑这种情况。

如果有三个奇偶校验位，包括前面提到的 p_1 和 p_T ，以及由所有偶数位（0, 2, 4, 6）数据计算得到的 p_E ，就可以把错误范围缩小到 4 比特位宽的半字节中的 2 个比特内。如果再加上由各个半字节的低 2 位计算得到的第四个奇偶校验位 p_L ，就能准确知道错误的位置。纠正的方法是将该错误位的值扔掉，补上正确值，使得看上去像没有出错一样。这样做的代价是要用 12 个比特来表示 8 比特的数据。在这个简单的例子里，虽然多用 50% 的空间，但没有降低速度。

对一个字节长度数据，前面介绍的过程可以总结为下面的几个步骤：

假设每个字里只有一个硬件错误，并且定义：reg [7:0] Word;

1. 定义： $p_T = \sim \text{Word}[7:0]$ ；如果 Word 里任意 1 个比特改变了， p_T 的值都会翻转，系统能检测到 p_T 的翻转。表示成：8'bxxxxxxxxx。
2. 定义变量： $p_N = \sim \text{Word}[3:0]$ ；如果低半字节中 1 个比特变化， p_N 和 p_T 的值会翻转。这样系统就能分辨出是 Word 的前半字节还是后半字节出问题了。表示成：8'bxxxxxxxxx。
3. 定义偶校验位： $p_E = \sim [\text{Word}[6], \text{Word}[4], \text{Word}[2], \text{Word}[0]]$ ；系统能够确定出错的是奇数位还是偶数位，以及是高半字节还是低半字节。表示成：8'bxxxxxxxxx。
4. 定义低半字节校验位： $p_L = \sim [\text{Word}[5:4], \text{Word}[1:0]]$ ；使用 p_T, p_N, p_E 和 p_L 这 4 个校验位可以定位错误比特并将它纠正。表示成：8'bxxxxxxxxx -> 8'bxxxxxxxxx。

采用这种 ECC 算法，每 8 比特数据需要 4 个额外的校验位。

实际的 ECC。通常，当数据对象大于一个字节后会采用 ECC，而它们所需的校验位只占被测数据的很小比例。统计学认为，用一组过约束的模式系数替换校验位的确定元素法比在校验树上进行二进制搜索来修复单个错误更有效。这通常采用 LFSR 硬件实现并应用代数域理论来对校验位的数据规律进行编码。大块数据中，不论错误出在什么地方，多比特错误能够进行某种程度上的纠正。512 字节数据块的校验位少于 64 字节。这个宽度和简单的以字节为单位检错但不纠错的方式是相等的。

为了描述实际 ECC 过程的机制，我们抛开复杂的群理论并采用最简化的方法：在奇偶校验的基础上，采用 8 比特数据，附加一个由 8 比特校验向量构成的校验和：

8 比特的数据作为字的高 8 位放在左边；数据的右边，也就是字的低 8 位上是 8 比特的校验和。最左边的校验比特是所有数据位上的奇偶校验和（1 的偶校验），它旁边的校验比特是数据的低 7 位的奇偶校验和，下一个校验比特是数据的低 6 位的奇偶校验和，以此类推到最后一个校验比特，它和数据的最低位相等。

这种方法采用带有自反馈存储元件的 LFSR 可以简单地在硬件上实现，如图 5.3 所示。



图 5.3 实现 ECC 复杂度最小的 LFSR

实现这个LFSR时，先把它初始化成16'b0，数据从最低位移入，移位16次。得到的结果就是左边8比特为数据，右边8比特为所求的异或值。得到的结果串行传递，每次移位消耗一个时间周期，初始值也可以通过存储空间的并行总线一次载入。

下面来看ECC如何工作。假设数据字节是1010_1011，加入了校验和的字是1010_1011_1001_1001。

现在，假设在串行传输时有一个比特发生错误，例如，数据的最低位发生翻转，变成了1010_1010_1001_1001。

接收端计算出的校验和为0110_0110，明显和收到的校验和不符。我们不认为这是因为校验和的多个错误而造成这种现象。这个例子没有闭合形式的解决方法，接收端会提出8种所有可能的错误假设，并根据这些假设对数据进行修改，每种假设对应数据的某一位翻转。

收到数据为1010_1010_1001_1001，如果只有1比特出错，则有下面8种可能：

假 设	纠正后数据	对应的校验和
h0	0010_1010	1110_0110
h1	1110_1010	1010_0110
h2	1010_1010	1000_0110
h3	1011_1010	1001_0110
h4	1010_0010	1001_1110
h5	1010_1110	1001_1010
h6	1010_1000	1001_1000
h7	1010_1011	1001_1001

假设h7的校验和与收到的校验和是相等的，因此，ECC会把数据的最低位翻转以纠错。

现在，假设有两个错误，分别为最高位和最低位。同样地，我们不认为是校验和出了这么多错。

收到数据为0010_1010_1001_1001，如果只有1比特出错，则有下面8种可能：

假 设	纠正后数据	对应的校验和
h0'	0010_1010	0110_0110
h1'	0110_1010	0010_0110
h2'	0000_1010	0000_0110
h3'	0011_1010	0001_0110
h4'	0010_0010	0001_1110
h5'	0010_1110	0001_1010
h6'	0010_1000	0001_1000
h7'	0010_1011	0001_1001

表中所有校验和都与收到的不同，而h7'对应的校验和与收到的很接近（只有1比特差异）。因此，可以接受h7'，把数据的最低位翻转。然后继续尝试，列出8种假设进行第二轮纠错。这样会进行2比特的ECC，纠正所有错误。实际系统中，距离的闭合形式是通过代数的Galois域来量化和最小化的。但上面这个简单的例子说明了允许多比特ECC校验和的基本原理。

如果想了解ECC校验和编码算法与计算过程的更多细节，请参阅参考文献中Cipra和Wallace的文章。

5.1.7 串行/解串器的帧边界的奇偶校验

简单的奇偶校验值可以极大地改善串行/解串器串行数据帧的效率。但是，我们不会在本课程的设计里运用奇偶校验。我们感兴趣的是 Verilog 设计，虽然 64 比特数据包的效率不高，但是它很简单，能够让我们在仿真中发现 Verilog 的设计错误。虽然我们的这个设计不会投入生产，但仍然不希望在设计里存在错误。

考虑下面这个用来使本地 PLL 时钟与串行数据中的时钟同步的方法。我们不在每个字节后面添加 8 比特编码信息，对每个数据只增加一个比特，使得每个字节变成 9 比特宽度。这样，32 比特的串行数据将会变成下面这样：

```
36'bXXXXXXXXXPXXXXXXXPXXXXXXXPXXXXXXXP
```

每个字节的奇偶校验位紧跟在字节后面，这种情况下假设数据的最高位先发送到串行线路上。每个字节的最高位由大写字母 X 表示，校验位由 P 表示。与上一个练习中的第 8 步相比较，用下划线强调每个字节的边界，可写成下面的形式：

```
36'bXXXXXXXXPXXXXXXXPXXXXXXXPXXXXXXXP
```

假设要利用这样的帧使得 PLL 时钟与之同步。如果担心字节边界会有 1 比特抖动，就计算奇偶校验；如果移动 1 比特，奇偶校验值可能变化，那就可能要调整 PLL 重新进行同步。如果使用对“1”的偶校验，帧里面只要有一个“1”出错就会导致校验错误，但如果“0”出错就不会。因此，帧检测有 50% 的正确率。

但这还不够好。我们需要可靠的同步。因此，假设使用对“1”的偶校验，确保在帧出错时至少有一个方向的奇偶校验出错。简单地在帧末尾增加一个新的比特：触发比特，即帧中某个比特位的相反值。

因为对“1”的偶校验通常指校验值和字的异或值为 0，接收端奇偶校验的硬件会检验表达式的校验值，这个值（校验值和字异或的结果）通常为 0。因此，如果字的最高位被固定在某个值上，校验位不会变化，校验就会出错。

所以，我们把新的触发比特加在帧里，放在（MSB-9）位置上，并要求在发送端计算奇偶校验值时忽略它。现在，数据包里用 10 个比特来表示 8 比特数据，奇偶校验位 P 在数据最低位后面，而用带下划线的小写 x 表示的触发比特跟在奇偶校验位后面并取数据最高位的相反值。在发端计算 P 值时不会计算触发比特。

```
40'bgXXXXXXXXPxXXXXXXXXPxXXXXXXXXP
```

每个字节里的最高位用大写字母 X 表示。发送端的奇偶校验位是比特 P；P 是每个 10 比特数据帧的结尾。x 是触发比特，例如 x 跟在第一个 P 的右边，置为第一个 X 的倒数。

现在有了一些进展：正确的帧是下面这种形式，下划线表示接收端检测到的帧边界。每一帧的第一个比特是 X 的取反 x，接收端会忽略这一比特：

```
40'bgXXXXXXXXPxXXXXXXXXPxXXXXXXXXP
```

如果接收端收到的帧有 1 比特滞后错误，即

```
40'bgXXXXXXXXPxXXXXXXXXPxXXXXXXXXP
```

后面跟着：

```
40'bgXXXXXXXXPxXXXXXXXXPxXXXXXXXXP
```

这个滞后错误使得接收端忽略了发送端发出的真正最高位，并得到它的取反，这会导致奇偶校验位出错，并检测到这个错误。不论已经混乱的数据（最高位值为 x 的数据）中传输数据的最低位是否恰好等于奇偶校验位的值，某些时候首位错误能够可靠的被检测到。由此得到的粗略的帧错误检测率差不多能达到 75%。

因此，可以设计出比精确同步滞后、有很细微偏差的 PLL，但是它可以检测出 1 比特的帧数据错误。上面这种方法让设计者用每字节不超过 10 比特的数据帧保证了接收 PLL 的可靠性。这里每字节 10 比特数据的比例是实际 PCI 高速总线设计中常用的假设，因为我们忽略了很多复杂的串行化过程，如曼彻斯特编码等。

5.2 练习 7：存储器

练习步骤

在目录 Lab07 下完成这个练习。

第 1 步。试着仿真下面的这段代码。用常量对 RHS 变量逐字进行初始化，然后看哪一个能工作。

```
reg[63:0] WordReg;
reg[07:0] ByteReg;
reg[15:0] DByteReg;
reg[63:0] BigMem[255:0];
reg[3:0]  LilMem[255:0];
...
BigMem[31]      <= WordReg;
WordReg         <= BigMem;
LilMem[127:126] <= ByteReg;
LilMem          <= ByteReg[3:0];
DByteReg        <= ByteReg;
ByteReg         <= DByteReg + BigMem[31];
WordReg[12:0]    <= BigMem[12:0][0];
```

第 2 步。设计一个带奇偶校验的，“ $1\text{kb} \times 32$ ”的静态 RAM 的 Verilog 模型。模块名为“Mem1kx32”。完成之后用这个模型来仿真。

这个 RAM 的地址位宽为 5 比特；用 Verilog 的参数来定义位宽和地址总数。这样一来，如果需要改变 RAM 的容量，只需修改这些参数就行了。奇偶校验比特对芯片外部的逻辑来说是不可见的，因此对它来说没有地址的概念。

可以在本章前面讲到的简单 RAM 模型的基础上来创建自己的 RAM 模型。在一个 always 块里实现读和写的功能。当然，这个模块会比那个简单的 RAM 要复杂一些。

这个 RAM 有读和写两组 32 比特的数据端口。再给这个 RAM 增加一个时钟，一个异步的芯片使能输入。这个使能的作用是当它为低时，读数据端口上的输出值为 z，同时时钟对 RAM 无效。

提供两个方向的控制输入，分别控制读和写。只有在时钟上升沿检测到的读或写控制信号的变化，读写才会起作用。如果读写都无效，读数据端口由上一次读出的值驱动；如果读写都有效，则进行读操作，但数据无效。

使用带延时的连续赋值语句为输出信号产生适当的延时。提供输出数据有效信号指示外部设备现在正在进行读操作，并且读出的数据是稳定而有效的。不用担心会发生读持续有效、地址和时钟同时变化这种情况。假设其他的系统使用的是这个RAM，它会根据你的规范来设计它的操作的。

另外，提供校验错误输出，如果读数据时检测到奇偶校验错误，该信号为高电平，并且该值持续为高直到输入地址发生了变化，参见图 5.4 里面的模块结构图。

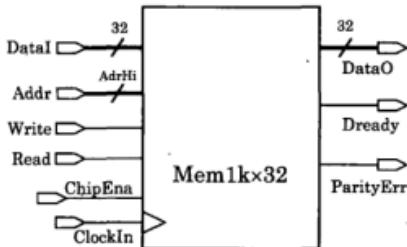


图 5.4 Mem1kx32 RAM 的结构图

因为这是静态 RAM，所以忽略动态 RAM 的特性，如 ras、cas 和 refresh。设计时使用触发器，不要使用 latch。把模型放在文件里，跟在模块后面。

在设计里加入断言语句，当发生奇偶校验错误时，在屏幕上打印错误信息。当然，你的仿真模型不可能遇到硬件错误，但如果设计错误导致奇偶校验位出错时，这条消息也会通知你。你也可以人为制造错误，例如在模型里放一条临时的阻塞语句使得异或操作生成奇偶校验位时出错。

第3步。检查你的RAM。向某个地址里写数据，然后通过仿真来验证是否能够正确读取（参见图 5.5）。

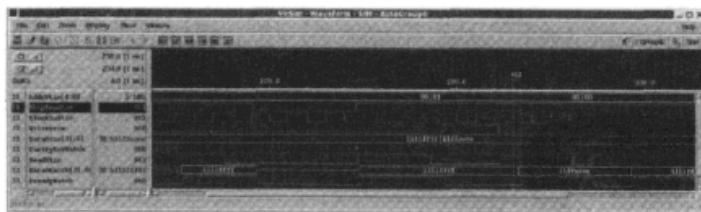


图 5.5 读写端口分离的单端口 Mem1kx32 的粗略仿真

第4步。完成上面各步骤后，仔细地仿真来确保设计的正确性。在测试平台的 initial 块里增加一个 for 循环向 memory 的各个地址写有一定规律的数据（如模 3 加法计数器的计数值），并将保存的数据显示出来。使用 \$display()任务来显示数据。对边界情况要额外注意，例如地址 0 和地址 31。在每个地址上，奇偶校验位是第 32 位。下面给出了这类循环的例子。注意在这个测试平台模块中如何指定实例 Mem1kx32_inst 中的数据对象（“MemStorage”）。

```

for (...)

begin
  ...
#1 DbusIn = (some data depending on loop);
#1 Write   = 1'b1;
#10 Write  = 1'b0;
SomeReg = Mem1kx32.inst.MemStorage[j];
$display('...', $time, addr, SomeReg[31:0], SomeReg[32]);
end

```

第5步。修改你的RAM设计使它只有一个双向的数据端口。采用下面的方法：把上面用的工作模型复制到新的文件里，命名为“Mem1kx32Bidir.v”，然后在文件里声明一个新的空模块，并为它取名。新模块的端口名和原模块Mem1kx32的完全一致，只是数据端口为双向的，如图5.6所示。

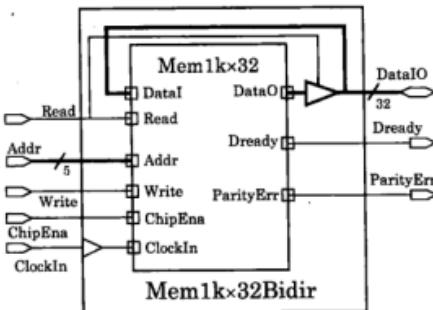


图 5.6 为 Mem1kx32 提供双向数据总线的外壳的结构图

在新的 Mem1kx32Bidir 里面例化已有的 RAM，把除了数据端口之外的所有端口一一对应连接。

在外壳模块里增加一条连续赋值语句，在读无效的时候关掉 DataIO 的驱动^①。同时，用连线将 DataI 和 DataIO 连起来。仿真时，通过对两个连续地址相继进行读写操作来验证新 RAM 是否正确（参见图 5.7）。

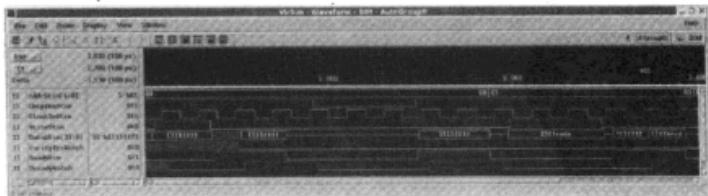


图 5.7 带双向读写端口的单端口 Mem1kx32Bidir 的粗略仿真

^① 原书有误，图中无 DataO，结合上下文，判断此处应为 DataIO ——译者注。

如果这是某个大工程里的一个小设计，你需要把双向 RAM 和单向 RAM 分别写在两个文件里。但是，如果把原 RAM 模型、RAM 的新的双向外壳模型及测试平台这三个模块放在同一个文件里，我们做这个练习的时候会变得简单一些。

第6步。以面积为优化目标，综合双向数据总线存储器设计，用文本编辑器查看生成的网表。然后重新以速度为优化目标进行综合，并查看网表。

5.2.1 练习后的思考

位拼接操作符什么时候有用？

怎样对模块实例进行层次化调用？

使用双向数据总线有什么优点？

如何改进标志位 Dready？当读信号有效，地址发生了变化会怎么样？如果由于输入信号而导致输出不可预期，这算是 RAM 的设计问题吗？

如何将 RAM 的触发器改成锁存器？

是否真的需要 ChipEna？为什么在读无效的时候还需要有输出？

5.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 5.1 节，学习 Verilog 中端口连接规则。

阅读 Thomas and Moorby (2002) 中第 166 页的 6.2.4 节，学习如何让这种奇偶校验的方法能够用在 Hamming 码的 ECC 格式上（每字节 12 比特）。

阅读 Thomas and Moorby (2002) 的附录 E.1 和 E.2，学习向量、数组和多维数组。

（选学）阅读 Neal R. Wagner 著的 “The Laws of Cryptography: The Hamming Code for Error Correction”，网址是：<http://www.cs.utsa.edu/~wagner/laws/hamming.html>(2004-12-15)。它讲述了一个简洁明了的 ECC 处理方法，扩展并改进了现有的内容。遗憾的是，这些内容被作者标记为过时了，它也许会成为一本书的一部分，可以从这个网站下载这本书。不过它讲的内容已经和本课程没什么关系了。

阅读 Palnitkar (2003) (可选)

4.2.3 节讨论了 Verilog 中端口连接规则。

仔细阅读附录 F 中 DRAM 存储器的 Verilog 行为级模型。它用到很多我们还没有讲到的 Verilog 语法。因此，你可以把它放在一边，过几周之后再来读。本书附带光盘上的 Silos 仿真器也许不支持其中的一些语法。

第6章 计数器

6.1 计数器的类型与结构

6.1.1 计数器简介

计数器是一种数据对象，它的计数值按固定步长增大或减小。因此，计数器的值可能为连续的0, 1, 2…或者2, 4, 6…单比特计数器在“0”和“1”之间翻转。图6.1所描述的是一个n比特无符号递增的二进制寄存器类型计数器，它的值从0开始，下面各行数据以1为步长连续递增。

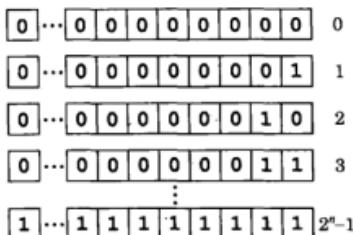


图 6.1 二进制加法计数器。MSB 在左边，连续计数时有 1 到 n 个比特发生翻转。

上图所示的计数器的值在空间上是非常紧密的，用 n 个比特表示 2^n 个不同的值，但每一个进位值都会导致几个（常常是多个）比特位同时发生翻转。不同的时钟下，进位和比特翻转会导致不同的延迟。所有的翻转都有可能会造成片内的串扰噪声或毛刺，使得计数器或相邻器件产生错误。

1, 2, 4, 8, 16, …这一组值本身并不是计数值。然而, 如果计数器的值由寄存器中“1”所在的比特位置来表示, 每次所在的位置编号以相同的步长增大, 如 $2^0, 2^1, 2^2, 2^3, \dots, 2^n$, 它也可以被看做是计数值。实际上是对寄存器的值用 \log , 取对数后进行计数。独热计数器 (one-hot counter) 就是一个这样的例子, 因为它的寄存器里有且只有一个比特位上的值为“1”。

独热计数器在硬件实现时采用移位寄存器，用一个“1”对其初始化。按照计数器里的连续的比特格式，独热计数器如图 6.2 所示。

主要注意的是，如果用二进制(2^n)的方式来表示一个数值，我们无法表示0。同时，这种方法的存储容量是有限制的，因为对于 n 个不同的值，需要 n 个比特来存储。从面积上来说，这种计数器是不划算的。但是由于每个时钟周期只有两个比特发生变化，所以它速度很快，噪声也很小（但不是最小的）。在一定程度上，这种做法是用面积的代价来换取了速度。

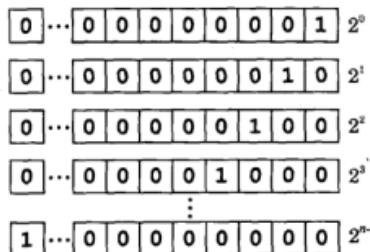


图 6.2 独热加法计数器。每次计数有两个比特发生翻转

6.1.2 术语：行为，过程，RTL 和结构

Verilog 至少有三种编码风格可以完成同样功能的计数器设计：行为级编码（behavioral），过程化编码（procedural）和结构化编码（structural）。有时候用 RTL（Register Transfer Level）这个词来描述设计风格，RTL 和行为级编码，过程化编码有一些重叠的内容。

行为级的 Verilog 语言允许使用基本的操作符进行运算，不需要考虑硬件的实现。简单的计数器可以写成：Count <= Count + 1, Count <= Count - 1, Count <= Count + Incr 等。综合工具会将行为级的代码转化为网表。我们看到，仅从上面这种方式直接来看，我们并不知道 Count 中哪个比特位会变化以及它会怎样变化。

在仿真过程中，过程化赋值语句所赋的值可能被另一条语句的赋值覆盖。过程化赋值语句只能放在过程块里。最常见的一种赋值是阻塞赋值。例如：

```
initial // Cycle the reset by time 10:
begin
#0 Reset = 1'b0;
#1 Reset = 1'b1;
#9 Reset = 1'b0;
end
```

行为级编码可以是过程赋值语句或连续赋值语句。过程赋值语句可能是基于位操作的，这不算是行为级编码。

我们可以用过程赋值语句对计数器的寄存器类型变量进行赋值。但是对计数器操作的细致程度和上面行为级代码的例子不一样，它不是直接操作整个计数器的所有变量，而是直接操作计数器的每一个比特。用异或表达式和比特赋值语句构成的计数器可能是过程化编码，但不是行为级编码。例如，在 always 块里，Count[2] <= Count[1]^Carry 就是对一个比特进行过程赋值的例子。注意，连续赋值语句 assign Count[2] = Count[1]^Carry 不是过程化赋值语句。计数器是时序器件，连续赋值语句能描述加法，但不能描述计数过程。

过程化编码和行为级编码的界限并不明显。通常，在不同的语境下，相同的代码既可以被理解成 RTL（通常是一类特殊的过程化编码）也可以被理解成行为级。回忆在第 2 章中 PLL 使用的简单计数器：我们使用了比特宽度的表达式 5'h1 来表示加法的步长，所以它是个 RTL 描述，而非行为级描述。

作者认为如果代码没有明确的对端口或寄存器进行比特位赋值的话, 使用行为级这个术语是合理的。这样的话, 不用对声明进行结构化的说明。

这与 Thomas and Moorby (2002) 的 1.2 节以及 Palnitkar (2003) 的第 7 章的使用方法是一致的。这本书里行为级描述很少, 因为它不是 RTL 风格的。

从结构上来说, 如果想在预定的工艺下手工将与库兼容的门电路实例化在网表里, Verilog 完全支持这么做。在这种环境下, 采用连续赋值语句的结构化连接代表了电路中的组合逻辑和简单连线。Palnitkar (2003) 里把数据流建模分成了包含连续赋值语句的特殊的一类。然而, 这种划分方法是不规范的。可以确定的是连续赋值语句是非过程化的 (在最新的 Verilog 规范中也不能把它算在过程块里)。

用另外一种方法来阐明这个术语。假设要描述一个 2 比特的二进制加法计数器。在一系列声明之后, 使用下面的 Verilog 语句:

行为级	<code>Count <= Count + 1;</code>	<i>// Count is reg.</i>
RTL	<code>Count[1:0] <= Count[1:0] + 2'b01;</code>	<i>// Count is reg.</i>
RTL	<code>always@(posedge Clock) Count[0] <= ~Count[0];</code>	
	<code>always@(Count[0])</code>	
	<code>if (Count[0]==1'b0) Count[1] <= ~Count[1];</code>	
结构化	<code>// Count is net:</code>	
	<code>DFF Bit0(.Q(Count[0]), .Qn(Wire0), .D(Wire0), .Clk(Clock));</code>	
	<code>DFF Bit1(.Q(Count[1]), .Qn(Wire1), .D(Wire1), .Clk(Wire0));</code>	

用综合工具进行综合时, 要避免使用门级的结构化编码。有时候, 我们会用结构化的控制语句去调整综合后网表。但对于复杂的结构化编码, 结果不一定是最好的。对于一些手工设计的非最优化复杂的结构, 综合工具的逻辑优化功能可能无法对它进行优化。这样的结果是, 采用特定库里的门电路会让设计变得依赖于特定的工艺。如果想用其他的工艺实现这个设计, 手工对门电路进行实例化会比综合器自动从工艺库里选择门电路复杂得多。

6.1.3 加法器表达式和计数器表达式

指出加法器和计数器的区别很重要。加法器可以用一个表达式来完成, 例如 $X \leq A + B$ 。即使是在函数里, 加法也仅仅是等号的右边那个加法表达式。而计数器把当前的计数值保存在寄存器里。所以, 加法可以放在组合或时序的过程语句里, 而计数只能放在时序的过程语句中。尽管加法计数器在实现时进行的是加 1 的操作, 但是无法用组合逻辑来保存计数值, 必须将它寄存起来。于是, 连续不同取值的 count 和 next-count 可以按照出现的顺序来区分了。count 的下一个值 (next-count) 依赖于 count 的当前值。

计数需要用加法表达式进行赋值, 如图 6.3 所示。

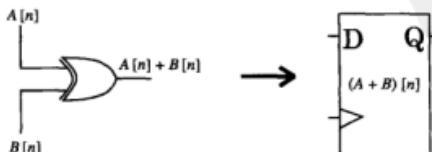


图 6.3 加法的结果被保存起来

6.1.4 计数器结构

接下来,讨论几种不同的计数器结构。下面介绍的这些计数器里,假设计数值都是从D触发器的Q端口输出的二进制值。这里只使用D触发器,因为无论是手工设计还是综合后产生的网表,D触发器都是最常用的时序存储单元。

所有基于D触发器的二进制计数器的构成元件都是翻转(toggle)触发器,有时也叫做T触发器,它是一种特殊的D触发器,它的输出端口~Q连在输入端口D上。

行为级描述的D触发器中,由D到Q的toggle触发器应该是非阻塞的,以确保更新的值是基于上一个时钟周期的赋值。

```
//  
// Basic toggle flip-flop:  
//  
wire Qn.D;  
...  
DFF Toggle01( .Q(Q), .Qn(Qn.D), .D(Qn.D), .Clk(ClkIn) );  
...
```

这段代码描述的器件如图6.4所示。

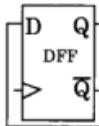


图6.4 基本的T触发器

只需在D触发器的外面加一个简单的外壳,就可以实现T触发器。

```
module TFF (output Q, input Clock, Reset);  
// DFFC is the DFF above, with clear (reset) added.  
wire Qn.D;  
DFFC Toggle1( .Q(Q), .Qn(Qn.D), .D(Qn.D), .Clk(Clock), .Rst(Reset) );  
endmodule
```

6.1.4.1 纹波计数器

纹波(ripple)计数器的面积最小,易于结构化实现。这种计数器里的触发器不是用时钟而是用前一级输出数据的边沿作为时钟来驱动的。前一级的输出连在后一级的时钟端,每当时钟的输入端口的数据产生了上升沿,输出就会翻转。这个计数器必须从一个确定的状态开始工作,这要求我们对它进行复位,否则,这个计数器自己的翻转就没有意义了。除了触发器,这个计数器不需要其他逻辑。例如,3比特的纹波计数器就如图6.5所示的这么简单。

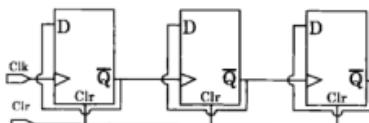


图6.5 由D触发器构成的3比特纹波计数器结构图,输出没有画出,MSB在最右端。时钟只送给了LSB

因为纹波计数器中高阶的触发器没有时钟，只有等到低阶触发器的输出变化后才能变化，所以当计数器的位宽增大时，这是个线性渐慢器件。在每个时钟上升沿之后，伴随着纹波的形成，触发器输出端会出现很多毛刺。然而，各阶的进位不会立刻由时钟传输下去，因此和同步计数器相比，功耗和噪声都减小了。

6.1.4.2 超前进位（同步）计数器

这类计数器增加了超前进位逻辑，在每个时钟有效沿对所有寄存器比特进行更新。这样一来，除非所有的低阶比特都为“1”，否则所有比特通常会受时钟驱动并同时翻转。这样，尽管所有比特都是由时钟驱动的，但各比特只有等到前一比特有进位时，才能变为“1”。

如果不考虑加法器的位宽，同步计数器几乎可以按照时钟速度运行。然而，因为所有比特翻转都和时钟同步，同步计数器偶尔会产生极大的瞬时功率，而这种尖峰功率会生成噪声，从而产生上文提到的逻辑错误。

注意，图 6.6 中的同步计数器的每个寄存单元（触发器）的输入端口都是由 1 比特加法器（异或门）驱动的。做设计的时候也应该遵循这个原则，组合逻辑后应该跟上寄存器，以保证电路的时序。

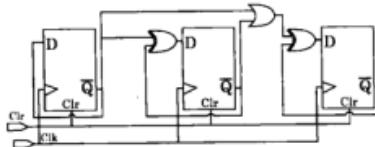


图 6.6 3 比特同步计数器的结构图。时钟连在每个触发器的时钟端上。前面的 1 (反相的) 进行或操作，然后用异或门进行隔离，直到进位溢出到该比特

6.1.4.3 独热计数器和环形计数器

与纹波计数器或同步计数器不同，独热 (one hot) 计数器和环形 (ring) 计数器不会把寄存器里所有可能的比特状态用光。

独热计数器已经在前面介绍过了。

环形计数器是将独热计数器的最高位移入最低位而得到的计数器。环形计数器中，单个“1”或者由复位逻辑或并行载入的常数在寄存器里无休止循环。环形计数器可用来生成任意占比的时钟脉冲。当用它来产生时钟时，环形计数器在某种意义上来说是个逆 PLL，它产生的时钟频率低于，而不是高于或等于输入时钟的频率。

6.1.4.4 格雷码计数器

在所有常见的各种计数器硬件结构中，格雷码 (Gray Code) 计数器在开关功耗和噪声上效率最高。它是由设计者 F. Gray 的名字来命名的。

格雷码计数器和其他简单的二进制加法计数器一样，会用到所有的比特状态。但相邻的两个计数值只会有一个寄存器不一样，如图 6.7 所示。这个效率是用额外的编码逻辑换来的，编码逻辑的功能是让计数值按特定的顺序发生变化。并且，将格雷码的值变为同等的 2ⁿ 二进制计数值也需要解码逻辑。

在本节不会使用格雷码。对于很在意开关功耗和噪声最小化的设计，使用格雷码会是一个好的选择。

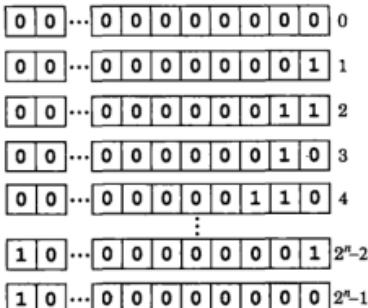


图 6.7 无符号格雷码加法计数器。MSB 在左边。到达底部状态 ($2^n - 1$) 时，下一个计数值返回 0 状态

格雷码的另一个应用是对不同时钟域的计数值进行同步。例如，生成连续的、一致的地址或者状态机的状态编码时会用到格雷码。正如参考文献里列出的 Cummings 的文章里所解释的，因为每次只有一个比特发生变化，对于格雷码来说，采样到计数器中间无效状态（除了有效计数值之外的其他值）的概率就比其他常用的计数器的概率小。

6.2 练习 8：计数器

在 Lab08 目录下面完成这个练习。

练习步骤

注意：在这个练习中，对于一个受时钟上升沿驱动的计数器，把计数值被时钟下降沿采样的次数叫做 miscount。

第 1 步。设计一个纹波计数器。在 Lab04 我们完成了一个上升沿触发器 (FF) 的设计。如果需要，把 FF 模型改成两个输出端口，分别是 Q 和 Qn。Qn 在逻辑上是 $\text{!}Q$ (或者 $\sim Q$)，可以由模型中 $\text{Qn} \leq \sim \text{D}$ 得到。如果模型的输出没有延迟，在给 Q 和 Qn 赋值时添加 #3 (3 ns) 的延时。

接下来，使用 FF 元件并按照图 6.5 里的结构设计一个 4 比特纹波计数器的 Verilog 结构化模型。把这个计数器的模块命名为 Ripple4DFF。

对模型进行仿真，得到最快速度下计数器从 0 变为 4'b1100 (十六进制数: 4'hC) 所需要的仿真时间。逐渐缩短时钟周期进行多次仿真，直到计数器出错。把此时的仿真时间记下来，以便日后参考。在另一个文件里使用测试平台。

注意：你发现的错误计数值仅仅依赖于延时，而仿真中无论时钟速度是多少，触发器都会翻转。这是因为仿真器实现不了延时语句里的惯性延时。因此，即使我们遇到了实际硬件电路由于时钟过快从而导致工作不正常的问题，也必须等到学习了路径延时 specify 块之后，才能在仿真时暴露这个问题。

综合 Ripple4DFF，分别对它的面积和速度进行优化。将进行了速度优化的网表保存下来，在本练习的后面部分会用到。

第2步。生成一个同步计数器。按照图 6.6 中的设计，采用和上面一样的 FF 模块来生成一个结构化的 4 比特同步计数器。将这个同步计数器命名为 Synch4DFF。用 Verilog 连续赋值语句来描述或门和异或门的连接。

用同一个 testbench 来仿真这两个计数器，把仿真结果进行比较。和前面一样，记录下从 0 变到 4'b1100 的最短时间。

分别用面积和速度两种约束来进行综合，把按速度优化条件产生的网表保存下来。

接下来，用同一个 testbench 对进行了速度优化的纹波计数器的网表和同步计数器的网表进行仿真，再看它们能跑多快。为了进行仿真，需要编译 Verilog 的库文件 Name_v2001.v，因为在综合后的网表里要用到 Verilog 的门级元件。如果使用 VCS 进行仿真，在.vcs 文件里的库文件名前加上 -v。

第3步。写一个行为级的计数器模型。在新的模块文件 Counter4.v 里面，声明一个 4 比特寄存器，用下面的方法进行计数器的行为级建模。

```
reg[3:0] CountReg;
...
always@(posedge ClockIn, posedge Clear)
begin
  if (Clear==1'b1)
    CountReg <= 0;
  else CountReg <= CountReg + 1;
end
```

在用连续赋值语句把 CountReg 的值赋给输出端口 CountOut 时，添加一个延时。该延时应该和第一步中 FF 的延时 "#3" 相匹配。接着对前面两个结构化计数器进行仿真，并比较时间。

然后，分别按面积和速度为约束条件进行综合。

最后，做个小实验。复制一份 Counter4.v 文件，把代码改成：CountReg <= CountReg - 1。仿真并注意当数值变为 0 后会如何翻转。寄存器类型是无符号的，如果按照向下的方向，0 的下一个值是这组寄存器能表示的最大正值。

第4步。用 Verilog 中特殊的线型来实现逻辑。把模型 Synch4DFF 另存为名为 Synch4DFFWor 的文件。在新文件里，把同步计数器中的 Verilog 或门表达式用 wor 替换。不要给新的或逻辑添加延时（我们会在后面的课程里深入学习上升延时和下降延时的区别），再次对其进行仿真，并以面积和速度为约束条件进行综合，把结果与 Synch4DFF 做比较。

记住在 CMOS 工艺库里线或是不可用的，因为上拉和下拉的强度是相同的。如果你的代码里用的是 wor，综合工具会用特殊工艺的线或门缓冲器件或其他元件，或者直接用库里的或门来驱动它。

第5步。使用 PLL 作为计数器时钟。在 Lab08 下面新建一个目录，把 Lab06 里的整个 PLLsim 模型都复制进去。把 PLLsim 模块及其对应的文件名改成 PLLTop。在一个名为 ClockedByPLL 的模块里同时例化纹波计数器、同步计数器和行为级计数器这三种计数器，将 PLL 的时钟输出连接到每个计数器的时钟输入。

用1 MHz的时钟作为PLL的输入(ClockIn)，如图6.8中框图所示。把以前模块文件中的仿真精度和预定义的代码仅放在新设计(ClockedByPLL)的顶层里，在testbench里例化这个顶层模块。简单地进行仿真，确保所有的计数器都在计数(参见图6.9和图6.10)。

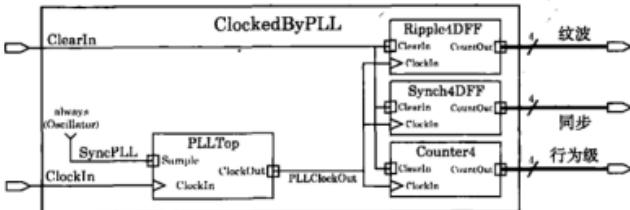


图 6.8 练习 6 里 PLL 作为 3 个计数器的时钟。复位没有画出来

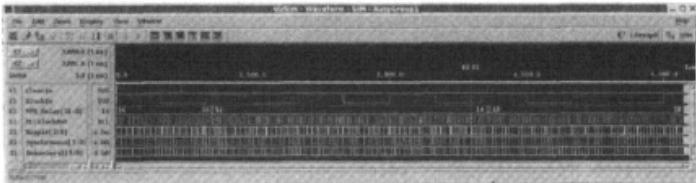


图 6.9 ClockedByPLL 中 3 个计数器的仿真波形

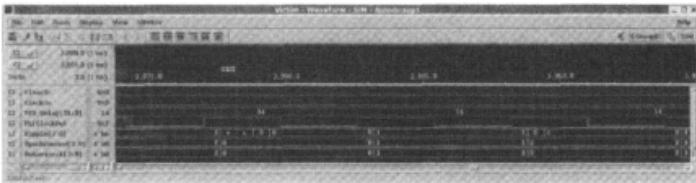


图 6.10 ClockedByPLL 中计数器的局部放大，可以看到毛刺

6.2.1 练习后的思考

比较同步计数器和纹波计数器仿真时序有什么区别？

如果D触发器的延时是约3 ns，为什么纹波计数器没有准确地以10 ns为周期进行计数？

仿真过程中，纹波计数器的输出在稳定之前出现的中间状态会令人感到困惑。这也说明在实际的系统中，毛刺或窄脉冲可能受总线的驱动从而导致电流的突然增大，或者是产生了噪声。应该怎样修改我们的设计使得数据总线不受这些噪声的干扰？

行为级计数器(Counter4)由加法计数器改成了减法计数器。如果用类似的方法对纹波计数器或同步计数器进行升级，能有什么简单的方法吗？

6.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 1.4.1 节，学习如何将计数器模型应用在设计里。

阅读 Thomas and Moorby (2002) 的附录 A.14 到附录 A.17，学习如何用设计状态机的方法来对计数器进行建模。

阅读 Palnitkar (2003) (可选)

为了充分说明 Verilog 的特点，Palnitkar 在很多不同的地方对计数器进行了建模。如果你对计数器 perse 感兴趣，可能会想运行 6.7 节中的例子。Palnitkar CD 中给出了解决方法。

2.2 节和 6.5.3 节里对纹波计数器进行了描述。

12.7 节中的练习 5 和练习 6 描述了用 J-K 触发器构成的同步计数器。注意：Thomas and Moorby (2002) 的 9.4 节对开关级 J-K 触发器进行了建模。



第7章 强度和竞争

7.1 竞争和操作符的优先级

这一章将进一步学习以下这些 Verilog 的知识点：驱动强度（drive strength），条件竞争（race condition），操作符优先级（operator precedence）和命名块（named block）。接着，会讨论怎样用一个锁相环来锁住一个串行的数据流。

7.1.1 Verilog 线性和强度

只有当一个线型有多个驱动时，才会有强度（strength）的概念。

在第3章中，已经讨论了一些特殊的线型：tri, wand, wor。在这一章中，只介绍强度的不同类型，而关于开关级建模的内容会放到后面的章节里再讲。

除了高阻态（Verilog 语言规定的4种基本逻辑状态之一，符号为 z），在进行大规模集成电路的 RTL 设计或行为级建模时，基本上不会用到强度。

根据 Verilog 的语言规范（IEEE 1364 的 4.4 节，7.9 节至 7.13 节），有两种强度，分别是：驱动强度（drive strength）和充电强度（charge strength）。

7.1.1.1 驱动强度

下面按递减的顺序来排列驱动强度的等级，它们分别是：supply, strong, pull 和 weak。“z”是一个逻辑状态，而不是强度，然而它的实际效果比 weak 还要弱。

在进行 RTL 或门级模型的设计时，只会用到强驱动（1, 0, x）或比 weak 还弱的驱动（z）。驱动强度只在如下的两种情况才会被用到：(a) 用连续赋值语句对一个线型进行赋值；(b) Verilog 内建的原语逻辑门的门级输出。对于声明成 reg 型的变量，驱动强度是无效。把 reg 型的值赋给一个线型变量之后，才会有强度的概念。

7.1.1.2 充电强度

下面按递减的顺序来排列充电强度的等级，它们分别是：large, medium 和 small。应该把它们理解成芯片内的电容。开关级（switch level）模型用来描述通过晶体管的电流通路，而这个电流的大小和器件本身的电容是密切相关的。唯一一个允许被声明成有充电强度的线型是 trireg。这种类型被叫做 trireg，是因为它们被看做是充电存储单元，而 reg 类型被看做是逻辑电平的存储单元。

如前所述，除了 z，强度和进行门级或行为级设计没有关系。只有在进行开关级电路的仿真时，才会用到强度。开关级的模型包括通过（pass）型晶体管，单 NMOS 或单 PMOS，或用来组合成一个逻辑门的基本门器件。

从某种意义上来说，Verilog 里的开关级模型是一片空白。对于开关级电路来说，为了准确地描述电路的行为，通常需要进行模拟仿真。Verilog 的开关级模型从功能上来说是正确的，但是时序上是不准确的。然而，Verilog 的开关级模型需要专注的问题就是时序，而其他的因素，例如：噪声，互绕耦合强度（crosstalk coupling strength），漏电流（leakage current）这些

都不在 Verilog 所包含的范围之内。但如果只要描述时序，不用开关级模型也已经足够了。因此，开关级模型在现在的商业库里用的较少，并不常见。

此外，库模型所使用的语言通常不是 Verilog，而是 ALF (IEEE Advanced Library Format) 或提供商所使用的特定的一种语言，例如 Liberty。这是因为一套完整的库模型需要包含和模拟电路相关的仿真数据，例如：不同负荷下的转换速率 (slew rate)；也有和仿真无关的数据，例如：布线的几何效应，布线的规则，线型负载和其他超出 Verilog 描述范围的内容。

如果发生了竞争，Verilog 会用强度的方法来解决竞争的问题。仅当不同的逻辑电平 1 或 0 发生了竞争，且强度也相同时才会有不确定的结果。对于其他的竞争来说，竞争的结果是强度更强的那个逻辑电平。普通的逻辑门，模块实例或 Verilog 表达式，它们输出的强度总是强的，和输入没有关系。

下面是 Verilog 里强度顺序的列表（可以参考 Thomas and Moorby (2002) 的 10.10.2.2 节）：

强度等级	关键词 (s)	逻辑等级 (s)	强度编号
supply	supply1		7
strong	strong1	1, x	6
pull	pull1		5
large	large		4
weak	weak1		3
medium	medium		2
small	small		1
highz	highz1	z	0
highz	highz0	z	0
small	small		1
medium	medium		2
weak	weak0		3
large	large		4
pull	pull0		5
strong	strong0	0, x	6
supply	supply0		7

其中，如果在 RTL 或行为级的代码里给线型变量赋值，上表列出了对应的默认逻辑电平强度等级编号（请注意，逻辑电平本身不是强度）。下面会讲到改变默认强度的办法。

上表中的强度等级是用来解决竞争的手段：高等级的结果会覆盖低等级的结果；如果两个相同强度等级的信号发生了竞争，则结果是不确定。

可以在声明实例或线型的时候，在声明之前用圆括号包含强度等级编号来指定强度；也可以在对线型进行连续赋值操作时，在目标对象的前面指定强度。驱动强度必须以一对的形式出现 (high, low)；如果是充电强度，则强度只有一个 (size)。下面是其例子：

```
wire OutWire;
...
and (strong1, weak0) and01( OutWire, in1, in2, in3);
nor (pull1, pull0) nor31( OutWire, in1, in2, in3);
trireg (large) LargeCapOnNet;
wire (supply1, supply0) TiedTo1 = 1'b1;
// ---
wire UpClock;
...
assign (strong1, weak0) UpClock = ClockIn;
```

在上面的例子里，多驱动的 OutWire 的真值表如下：

in1&in2&in3	and01 输出	nor31 输出	输出连线
1	strong1	(pull1 logically impossible)	(strong1)
1	strong1	pull0	strong1
0	weak0	pull1	pull1
0	weak0	pull0	pull0

在这里，只关心和强度相关的概念。将在以后再讨论开关级建模的细节。

两个相关的过程赋值语句分别是 force 和 release。仿真器认为它们的优先级比最高的强度还要高，可以任意覆盖任何一个信号值。除非是写 testbench 或在进行调试，我们不应该使用这两个语句。它们的用法是这样的：force object = level；object 的类型是 reg 型，它强制指定了对象的逻辑电平，直到 release object；结束。

还有一对功能类似的过程语句是 assign 和 deassign，我们不推荐在设计里使用它们，因此也不会继续学习它们的细节。

7.1.2 竞争条件

在 Verilog 里，模块里的每一个块都会在仿真周期里并行地计算输入值（或 RHS，right hand side）；这包含例化和 initial 块。仿真事件的顺序受以下两个因素的影响：(a) 根据延迟的积累，事件发生在特定的某个仿真时间；(b) 它们在过程块（always 或 initial）中的位置。

对于仿真时每一个最小的时间单位，都会有一个非常重要的语句执行顺序，只是这个顺序从仿真波形上看不出来而已。我们将会在后面的章节学习与 Verilog 的仿真器事件队列相关的知识。

现在的综合工具对过程块中并发操作的支持并不是很好；同时，综合工具也不支持设计里的延迟表达式。为了建立可综合的事件顺序，需要把先发生的事件作为输入（RHS），把后发生的事件作为输出（LHS）；或者用 reg 型变量的状态变化来控制事件发生的顺序。

时间的顺序包含单一顺序，也包含多种顺序。如果一个事件在整个仿真周期只发生一次（例如硬件的复位），则认为它是单一的，它往往和其他的事件都没有关系；如果某些事件每个时钟都会发生，或者是每隔一个周期就会发生一次，则这种事情不能单独安排它的顺序，必须要和这个时刻里其他的事件放在一起安排它们的顺序。

当某些状态依赖于两个或多个相关的事件，并且它们之间的顺序不可预测时，可能会出现竞争条件。显然，条件同时满足时，就会形成竞争。到底谁应该先发生呢？这肯定是设计者不想看到的结果。通常来说，不能预测结果的硬件是无法使用的。如果综合工具不能把形成这种结果的代码找出来，并给出错误提示，那么就有可能综合出不能使用的硬件电路。

最简单的竞争条件是在两个或多个并行块里同时对一个信号对象进行赋值。如果这些块不是并行的，就不会有竞争，硬件电路也不会出问题。下面是一个会引起问题的例子：

```
always@{posedge clockIn}
begin
#1 a = b;
(其他语句)
end
...
always@(a, b, c) #2 ena = (a & b) | c;
...
always@(a, b)    #2 ena = a ^ b;
```

上例里实现延迟的过程是这样的：如果在事件控制表达式里的变量跳变成了 1 或 0（把由其他值跳变成 1 的情况叫做产生了上升沿），当满足条件时，always 块里代码被仿真器读到。当延迟的时间达到了代码里指定的时间时，开始计算 RHS 并把结果进行赋值。

这段代码里形成竞争的部分是对 ena 的赋值。为了简化问题，对 a, b 或 c 的赋值不在这段代码里列出。代码里竞争的关系是很明显的。不难想象，在一个大型的模块里，由于复杂的并发关系，会导致发生时序的不确定和竞争。

有的读者也许会这么来理解这个竞争的时序关系：在时钟的上升沿，时间等于 0。接着，在时间等于 1 的时候，a 得到了 b 的值，这导致 a 跳变成 1 或 0。这触发了另外两个 always 块。在时间等于 3 的时候，由第二个 always 块里的代码，ena 会被赋值。当执行到第三个 always 块时，ena 的值会怎么样呢？……看来是出问题了。

更糟糕的是，如果 c 在时间等于 1 时发生了变化会怎样呢？它会触发第二个 always 块，这样，ena 的值可能同时受表达式 $(a \& b) \mid c$ 或 $a \wedge b$ 的影响。不只是如此，任何时钟的变化都有可能对 ena 的值产生影响。

此外，如果在时钟上升沿之前，a 和 b 的值就相等。这样，时钟沿不会改变 a 的值，因此，在时钟的上升沿之后，仿真器不会去读另外两个 always 块。但是，由于 a 和 b 的赋值逻辑并不在这段代码里，这两个 always 块可能在任何时候由于 a 和 b 发生了变化而被触发。情况越来越复杂了。不能这样做设计。

为了避免类似的竞争，切记不要在一个模块的不同 always 块里对同一个信号进行赋值。避免竞争有一个简单的解决方法，那就是在同一个时钟的上升沿去触发所有的 always 块，并且保证接下来的延迟不超过半个时钟周期。另外一个办法则是利用同一个时钟产生两个或更多不同相位的时钟，再利用这些时钟去避免竞争。

再回到这个例子，如果不利用时钟要去解决这个例子里的竞争问题，要做的第一步很明显：把所有对 ena 的赋值都放在一个 always 块里去。

```
always@(posedge clockIn)
begin
#1 a = b;
(其他语句)
end
...
always@(a, b, c)
begin
#2 ena = (c==1)? 1 : a & b; // An if could be used here.
#2 ena = a ^ b;
end
```

改成这样之后，对 ena 赋值的顺序就很清楚了。而且这样的写法也不会出现由于有很多并行的赋值操作，某些操作被忽略，没有被设计者注意到的情况。也许有读者有疑问：为什么要把赋值一个接一个地排下来？这样，前面的赋值会很快被后面的赋值覆盖掉，相当于产生了毛刺。但是，这样的写法至少解决了竞争的问题。我们可以再来这样改写这段代码（改写后的代码并不一定是原设计的本意，因为那个设计的本意本来就是模糊的）。

```

always@(posedge clockIn)
begin
#1 a = b;
(其他语句)
end

always@(a, b, c) #2 ena = (c==1)? 1 : a ^ b;

```

在讨论产生竞争的条件时，我们也不应该忽略initial块可能造成的影响。Verilog允许在一个模块里出现任意数量的initial块，除了initial块只被执行一次，其余可能产生竞争的条件和always很类似。由于综合工具会忽略掉initial块，因此，它们不会造成综合上的问题。但是会造成综合和仿真不一致。

只能在testbench里用initial块；在有特殊用途的设计里也可以用到initial块，例如在反标SDF时。但是，我们仍然要指出以下几点：

在实现并行功能时，always块和initial块是等效的。在0时刻，always块和initial块都有可能第一个被执行，从而完成代码里的赋值或其他功能。有的时候，可能在always块里和initial块里会存在操作同一个信号的逻辑，在进行仿真的时候，我们不应该忽视这种情况。下面举出了一个例子，如果always块在initial块之前被执行，在时间等于0的时候，Reset为1'b0的情况就被漏掉了。即使Reset在0时刻之后又变成了0，但也许Abus就不能被置成0了。

```

initial
begin
Reset      = 1'b0;
#0 Reset = 1'b1;
...
end
always@(ClockIn, Reset)
if (Reset==1'b0)
    Abus <= 'b0;
else Abus <= Inbus;

```

如果把阻塞赋值和非阻塞赋值都放在一个过程块里，可能还会引起其他的问题。切记，千万不要这样做。首先检查信号是否受时钟驱动，如果是时钟驱动信号，则只用非阻塞赋值。对于组合逻辑，首先要保证信号的建立时间，而任何一个时序上的变化都会在整个逻辑上传递。如果逻辑是有顺序的，但不受时钟驱动(latch)，则阻塞赋值和非阻塞赋值都可以用，但是一定要注意综合的结果是否符合设计的要求。如果有必要，可以重新划分你的设计（一个模块内的设计），把时序逻辑和组合逻辑彻底分开。

在结束对竞争条件的讨论之前，需要指出的是，和其他大多数数字仿真器一样，Verilog的仿真器也使用一种脉冲传播机制，这种机制叫做惯性延迟(inertial delay)。一个输入脉冲必须具有一定的惯性才能够通过一个门电路并且产生输出。默认地，如果一个模块被指定了传输(路径)延迟，同时一个外部输入的变化电平的持续时间小于这个传输延迟，这个输入将被忽略掉。这是一种过滤毛刺的形式，只有当毛刺足够宽时，才会穿过这个门器件并且影响到输出。在物理上，这是一个很简单的能量响应模型：很小的脉冲被认为没有足够的能量去翻转一个门器件；而路径延迟的值就是用来衡量这个能量大小的。和其他的HDL语言不同，Verilog允许设计者去调整这个惯性脉冲的宽度。我们将在后面的课程里学到这个内容。

除非用 specify 块（后面会讲到）来指定延迟，专门为大规模集成电路设计的仿真工具往往不能正确地仿真惯性延迟。因此，除非你已经真正测试过，否则不要在连续赋值或过程块里利用人工指定的延迟去消除毛刺。

7.1.3 关系表达式对于不确定的处理

我们还会更深入地学习这个内容。

关系表达式把 x 理解成非真也非假，也就是说，它既不是 1 也不是 0。因此，x 或 z 会导致结果不能为真。

例如：

```
reg[3:0] A, B;
...
A = 4'b01x1;
B = 4'b0001;
if (.A > B)
    X = 1'b1;
else X = 1'b0;
```

上面代码的结果将会使 x 变成 1'b0。也许会令某些读者吃惊，即使对于下面的代码，结果仍然是一样的。

```
if ( A == A )
    X = 1'b1;
else X = 1'b0;
```

为了能够明确的、单独的处理 Verilog 里的 4 种逻辑状态，可以使用 case 语句。对于 A 等于 4'b01x1 的情况，利用下面的代码可以让结果 X 输出 1'b1。

```
case (A)
4'b0000: X = 1'b0;
4'b0011: X = 1'b0;
4'b01x1: X = 1'b1;
default: X = 1'bx;
endcase
```

Case 语句的条件分支进行的是完全对比。它不允许出现通配符（wildcard）。还有两种特殊的 case 语句：casex 和 casez，它们是允许使用通配符的，我们暂时不学习和它们相关的内容。

7.1.4 Verilog 操作符和优先级

我们已经多次用到了 Verilog 里的比特操作符。下面是一个完整的操作符列表，Thomas and Moorby (2002) 的附录 C 和 IEEE 1364 的 5.1 节里也有详细的介绍。

符 号	类 型	符 号	类 型	符 号	类 型
-	位操作	*	算术运算	>	关系运算
&	位操作	/	算术运算	<	关系运算
-&	缩位符	+	算术运算	>=	关系运算
	位操作	-	算术运算	<=	关系运算
-	缩位符	%	算术运算	==	等价运算
^	位操作	**	算术运算	!=	等价运算

(续表)

符 号	类 型	符 号	类 型	符 号	类 型
$\sim\sim\sim$	缩位符	!	逻辑运算	$==$	等价运算
$>>$	移位操作	$\&\&$	逻辑运算	$!=$	等价运算
$<<$	移位操作	$\ $	逻辑运算	$? :$	条件运算
$>>>$	算术移位	{ }	拼接		
$>>>$	算术移位	{ n[] }	复制		

除了符号 \sim 之外的所有比特操作符都是缩位符。异或非(xnor)的表达方式有两种，作者喜欢使用 $\sim\sim$ 。

如果操作数是1比特，则逻辑操作符和与其对应的比特操作符的结果是一样的。1或者 $1'b1$ ，表达的含义都是真(true)。当我们在对总线、寄存器组进行操作时，或综合一组门器件时，最好明确操作数的位宽，从而避免错误。

算术右移操作符(>>>)在进行移位的时候会一直保留符号位，而其他的右移操作符首位是补0。

综合工具通常对幂指数运算符(**)有特殊的要求：要么两个操作数都必须是常数，要么指数必须是2的整次幂。在实际的工程里，都会用 $1<<n$ 这个表达式来替换 2^{**n} 。

全等和不全等操作符(==或!=)不是用在case里的，它们通常用在if或其他的条件表达式里。它们的作用和case(不包含casex和casez)里的条件对比是一样的，可以区分x和z。和其他的相等运算操作符不一样，它们的返回值不能为x。全等操作符不支持通配符，因此不会把x和z看做是通配符。case可以提供类似的功能，而casex和casez不行。

赋值操作符可以用来对总线的部分比特进行操作，例如：“Xbus[15:0] <= {Abus[4:1], 6{1'bz, 1'b0}};”，赋值后的Xbus的低12比特是6组交替的z和0。

除了条件操作符，对于其余的操作符来说，当优先级相同时，是按照从左到右的顺序执行的。而一元运算符的优先级是最高的。下面是运算符优先级的列表，在IEEE 1364里也有详细的介绍。

优 先 级	运 算 符
最低 = 0	?:(条件操作){ } (拼接)
1	$\ $
2	$\&\&$
3	$ $
4	$\sim\sim\sim$
5	$\&$
6	$== != === != ==$
7	$< <= > >=$
8	$<< >> <<< >>>$
9	$+ -$ (二进制)
10	$* / \%$
11	$**$
最高 = 12	$+ - ! & ^ -\& - -\sim \sim -$ (一元运算)

7.2 数字基础：三态缓冲和解码器

在开始做练习之前，让我们来看看一些用Verilog实现的基本器件。

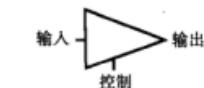


图 7.1 标出了端口的 bufif1

首先，介绍图 7.1 所示的三态缓冲器件（three-state buffer）。bufif1 是 Verilog 的原语门器件，当其为打开状态时，它的作用是一个简单的放大器，或者说是缓冲门；当它关闭时，它的输出是 z。因此，它的端口包括一个输出位，一个输入位和一个控制位。

`bufif1 optional_InstName (out, in, control);`

当控制位是逻辑 1 时，这个门处于打开状态。显然，这个缓冲是一个由 if 1 来控制其打开的器件，而这就是这个缓冲器为什么叫这个名字的原因。我们会利用这个器件来练习刚学到的不同驱动强度之间的竞争。

接着，来学习一个解码器（decoder）。对于一个 2-4 解码器来说，最简单的实现办法是使用 case 语句。我们可以用 case 来组成一个查找表电路。以二进制格式的 Sel 作为条件，这个查找表列举了所有的情况，结果被送给了 xReg。在下一个实验里，我们会用到这个例子。

```
reg[3:0] Sel, xReg;
...
always@ (Sel)
begin
  case (Sel[1:0])
    2'b00: xReg = 4'b0001;
    2'b01: xReg = 4'b0010;
    2'b10: xReg = 4'b0100;
    2'b11: xReg = 4'b1000;
  default: xReg = 'bx; // e. g., to handle an 'x' in Sel.
  endcase
end
```

这是目前第一次在练习里完整地使用到了 case 语句。记住很重要的一点，不要在 case 里留下未覆盖的条件分支。请务必加上 default 指明未覆盖的条件分支应该怎样处理。在这个例子里，我们在 default 分支里把 xReg 赋值成 x，告诉综合工具我们并不关心 default 这个条件。这样的处理会使综合工具在进行综合时有更好的面积结果。忽略某些条件分支有可能会产生并非设计本意的 latch。在后面的章节，还会继续学习在 Verilog 中非常有用 case 语句。

7.3 练习 9：强度和竞争

在目录 Lab09 下完成不是可选的那些练习。如果你有 Thomas and Moorby 或 Palnitkar 的书，这两本书都随书附带了演示版的 Silos 仿真器，读者可以用这个工具完成本次练习里可选的那些步骤，这是一个关于连线强度的练习。需要注意的是，演示版的 Silos 并不是被设计来完成大型设计的仿真的。

这些可选步骤可能在专门针对大规模集成电路仿真而优化的工具上不能被正确执行。因此这些仿真工具不会被用来仿真强度的问题，而仿真速度和对可综合 Verilog 语法的正确解析才是它们的目标。强度的语法是不可综合的；而竞争产生的 z，在实际的大型设计里意味着不确定。

练习步骤

在 Lab09 目录下完成这个练习。

新建一个名为 Netter 的模块，这个模块的逻辑如图 7.2 所示。

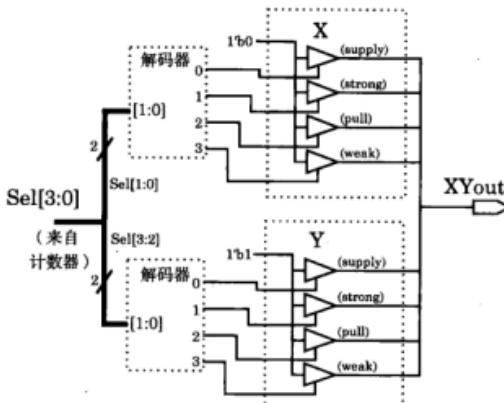


图 7.2 Netter 的功能示意图。虚线框说明那是一块逻辑，而不是子模块

由图 7.2 可知，Netter 的输出全都是三态缓冲门。在 Verilog 里，不允许把简单的逻辑连线 (wor, wand) 作为输出。这也许是因为没有办法给这类连线关联延迟的原因。而我们知道，门器件的输出是有延迟的。

我们希望设计这样一个实验，能够通过这个实验观察 4 种驱动强度之间的竞争。为了便于观察结果，我们将有选择性的关闭和打开一些缓冲门，使得一次只有两个门是打开的。也就是说，在某一个时刻，两个门的控制端接逻辑 1，而其余门的控制端接逻辑 0。这样，可以清楚地看到强度竞争的结果。

4 种驱动强度一共可以组成 16 个两两的组合。为了便于说明问题，把图 7.2 里的上面 4 个 bufif1 称为 X 缓冲，把下面 4 个 bufif1 称为 Y 缓冲。X 缓冲的输入都为 0，Y 缓冲的输入都为 1。X 或 Y 里的每一个缓冲都被指定了不同的驱动强度。我们再设计一个 4 比特的二进制计数器，把它的低 2 比特和 X 的解码器相连，把它的高 2 比特和 Y 的解码器相连。这样，只需对计数值进行解码，就能保证在 X 和 Y 里一次只有一个缓冲门是打开的。解码后，二进制计数器被解析成了独热 (one-hot) 的格式。也就是说，2 比特的数值被解析成 4 比特的数值，在这 4 比特的新数值里，一次只会有一个比特为 1。

随着计数值的变化，4 比特的计数器会覆盖到 4 个比特能出现的所有组合。如果再把 X 和 Y 缓冲的输出连在一起，就能够得到 X 缓冲中的一个缓冲与对应的 Y 缓冲中的一个缓冲构成的所有可能组合。

第 1 步。 设计两个解码器。从代码头开始设计模块 Netter。按照图 7.2 中的电路，先声明一个名为 Netter 的模块，它有一个 4 比特的输入，将解码器 X 的输出 xReg 用内部总线 xChoice

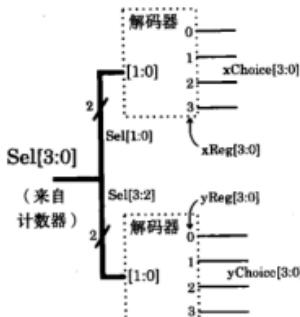


图 7.3 Netter 解码器输出的连线

连出来，如图 7.2 所示。用一个包含 case 语句的 always 块来实现这个解码器的功能，该 always 块如前文（7.2 节最后一个代码段）所示。然后在该 always 块里用另一个 case 语句实现解码器 Y。其中，解码器 X 的输入接 4 比特输入（Sel）的低 2 比特，解码器 Y 的输入接 4 比特输入（Sel）的高 2 比特解码器 Y 的输入，yReg 连在新的内部总线 yChoice 上。不要急着把连线 xChoice 接在任何东西上。

此时，未完成的模块 Netter 的结构如图 7.3 所示。

由于 always 块里 case 条件分支的输入是常数，因此 always 块的敏感变量列表里除了 Sel 不应该有其他的信号。

用VCS或QuestaSim来检查你目前的模型，在你的testbench里，增加输入为x的测试向量。这些工具都是被设计来完成大型集成电路设计的功能，对驱动强度的支持并不好，但是它们会帮助你找到语法错误。

第2步(可选)。在第1步的基础上，加入下面的代码。在例化bufif1的时候指定它们的强度，下面的这段例化应该被例化两次。第3步里有具体的连线提示。

```
bufifl (supply1, supply0) inst_name( . . );
bufifl (strong1, strong0) inst_name( . . );
bufifl (pull1, pull0) inst_name( . . );
bufifl (weak1, weak0) inst_name( . . );
```

第3步(可选)。在声明了8个bufifl之后,这样来连接解码器的输出。

```

wire[3:0] xChoice; // Just to rename xReg.
...
assign Xin = 1'b0;
assign Yin = 1'b1;
...
assign xChoice = xReg;
//
bufif1 (supply1, supply0) SupplyBufX(SupplyOutX, Xin, xChoice[0]);
bufif1 (strong1, strong0) StrongBufX(StrongOutX, Xin, xChoice[1]);
... (2 more X and 4 more Y)

```

第4步(可选)。把8个缓冲的输出都连在一起，只需用连续赋值把所有的输出都赋值给同一根线即可。这样的赋值形式当然是并行的，因此，没有必要考虑它们在Verilog源代码里排列的位置。

```
...  
assign XYwire = SupplyOutX;  
assign XYwire = StrongOutX;  
... (6 more) ...
```

接下来的工作就是在例如Silos这样的仿真器里观察XYwire这个信号对于不同驱动强度之间的差别了。竞争的结果也可以通过Netter的输出端口送给外部。但如果这个结果被赋值给一

个寄存器 (reg) 类型，强度的信息不会被保留下，而且只有强度为 strong 的逻辑电平值能维持。

不要在Netter里加入时钟，因为你所需要的是一個组合逻辑(选通和解码)。虽然用在case里的信号被声明成了reg，但实际上它们并不会保存之前的状态。如果能保证你覆盖到了case所有的情况，那么这些信号仍然只会产生组合逻辑。

第5步(可选)。在你的 testbench 里产生一个计数器，用计数值来驱动 Sel，从而实现所有竞争条件。

在观察了仿真结果之后，不加面积约束去综合这个 Netter 模块。检查综合后的网表，看看综合工具是怎样实现这个解码器的，再观察强度相关的逻辑是否被保留了下来。加上面积约束再综合一次，比较两个网表的区别。

关于线型强度和竞争的练习到这里就结束了。之后的练习不再有只能用 Silos 才能仿真的地方了。

第6步。竞争条件练习。新建一个名为 Racer 的模块，在这个模块里加入如下两个 always 块。

```
always@(DoPulse)
begin
#1 RaceReg = 1'b0;
#1 RaceReg = 1'b1;
#1 RaceReg = 1'b0;
end
//
always@(DoPulse) #1 RaceReg = ~RaceReg;
```

在 testbench 里例化 Racer，给它增加一个翻转的输入信号：DoPulse，这个信号会翻转数次。仿真并观察结果。第一个 always 块在每一个 DoPulse 边沿的 2 ns 后都产生了一个 1 ns 宽的正脉冲信号。第二个 always 块的作用是让 RaceReg 产生翻转。如果翻转发生在 RaceReg 第一次被赋值为 0 之后，这个脉冲可能提前并且变宽。如果在 RaceReg 第一次被赋值为 0 之前发生了翻转，旧值会被 0 替代。改变这两个 always 块的顺序，再仿真，观察结果是否发生了变化。仿真结果有可能会发生变化，这是因为并行块的执行顺序没有统一规定，由工具的开发商自行决定。通常，并行块的执行顺序是按它们在代码中的位置来决定的。

图 7.4 和图 7.5 给出了 VCS 或 QuestaSim 工具仿真出来的波形。Silos 和 Aldec 仿真的波形又是怎样的呢？

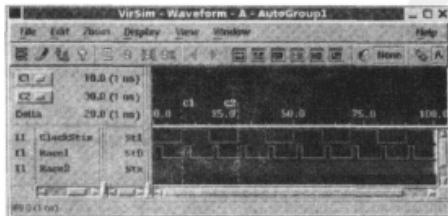


图 7.4 实现翻转的 always 在前

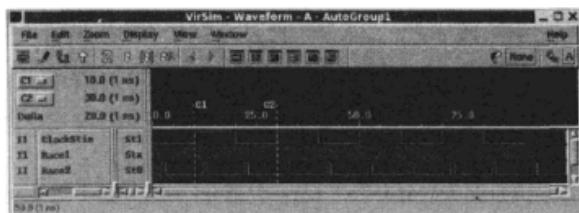


图 7.5 实现翻转的 always 在后

第7步。操作符和优先级。读者通常容易混淆与 (and) 和或 (or) 的逻辑运算和位运算之间的差别。对于 1 比特的变量，它们的运算结果是一样的。而对于多比特的变量，它们有着非常重要的区别。

对于一个 3 比特的位屏蔽 (bit-mask) 变量来说，下面是一些例子（左边屏蔽右边）：

```
3'b110 & 4'b1111,
3'b010 & 5'b00111,
3'b101 & 3'b111.
```

操作符 “&” 是一个位操作符。因此，上面表达式的结果分别是 110, 10 和 101。如果两边的位宽不一致，则高位自动补 0。

而操作符 “&&” 是一个逻辑操作符，如果它两边的操作数都不是 0，则这个表达式的结果是 1（等效于 1'b1）。其余的情况是 0。因此，表达式：3'b010 && 6'b111000 的结果是 1。如果把这个操作符换成 “&”，则表达式：3'b010 & 6'b111000 的结果是 0。

思考把上面的表达式的值赋给一个位宽为 8 比特的总线，结果是什么？如果表达式里某些比特为 x 或 z，结果又是什么？

逻辑操作符会返回 x 吗？

等于和全等操作符在处理 x 时，结果一样吗？

新建一个名为 Operators 的小模块，仿真前面的 3 比特位屏蔽表达式，另外一个操作数是 8 比特。分别把操作符写成 “&” 和 “&&” 并仿真。

可选步骤：如果还有时间，用仿真工具计算下面两个表达式的结果。其中，a 和 b 都为 1，c 和 d 都为 0。

```
NoParens = a&!d^b|a&~d^a||~a^b^c^a&&d;
Parens   = ((a&(!d)^b) | (a&(~d)^a)) || ((~a)^b^c^a) && d;
```

做这个练习的目的是让读者具备读糟糕代码的能力。通过这个例子，我们学到，当自己写代码时，应该多用圆括号和空格来划分复杂的表达式。

7.3.1 强度练习后的思考

VCS 和 QuestaSim 是怎么处理竞争的？

如果两个并行的语句对同一个线型赋值同样的逻辑电平，这时候竞争还重要吗？

7.4 接着讨论 PLL 和串行 / 解串器

7.4.1 命名块

在下一章里，我们还会学习到很多 Verilog 的结构，这些结构都是可以被命名的。在 Verilog 块结构的第一个 begin 之后，分别是冒号和名字。这个名字就是给这个结构的命名。显然，只有过程语句才能被放在块中，因此，也只有这样的块才能被命名。如果这个块没有 begin，则不能被命名。如果有必要，可以给任何过程语句包上 begin 和 end，从而给它们加上名字。命名的后面不能有分号。如果这个名字没有以数字开头，则任何的数字和字母的组合都是可以的。

例如：

```
always@ (negedge clk)
  begin : MyNegedgeAlways
  ...
  end
```

或

```
begin : Loop_1_to_9
for (j=1; j<=9; j = j + 1)
  begin
  ...
  end // for j.
end // Named loop block.
```

虽然对一个块进行命名，这个名字本身是不实现功能的。但是，正如模块名的作用一样，可能会在综合后的网表里或仿真列表里找到这个命名块。因此，命名块可以用来帮助调试或优化网表。

虽然命名本身并没有功能，但是却可以利用块的命名来实现新的功能。例如，把有循环语句 (for, while, forever 等) 的块进行了命名，可以用 disable 语句来终止这个命名块的进程。这一点，和 C 语言里的 break 语句的功能很像。

7.4.2 串行 / 解串器里的 PLL

首先请回忆第4章中的串行/解串器的示意图。由于要实现的是一个全双工的系统，因此，串行发送 (Tx) 和接收 (Rx) 装置各需要两套。串行 / 解串器的每一端都处在不同的时钟域里。这些时钟用来在两个系统之间的并行数据总线上传送数据。

为了让我们的设计更具有通用性，有意认为两个系统的时钟是同频但是异步的。对于每一个发送端，都需要一个串行时钟把数据在串行数据线上传输。对于解串的接收端，也需要一个同样频率的时钟来接收数据。我们把系统的时钟域设计成 1 MHz，而串行总线是以 32 MHz 的频率在工作。这些时钟由 4 个独立的锁相环 (PLL) 产生，分别送给发送端和接收端。根据现在的技术，用模拟 PLL 可以很容易地产生频率比为 32 : 1 的时钟。

为了避免不必要的复杂性，在讨论串行数据传输时，我们将不会讨论差错检测、纠错、加密、压缩、边沿检测以及包恢复等话题。要完整实现这些协议对本课程来说花的时间太多了。

串行/解串器的发送端的功能很明了：并行数据的处理时钟是1 MHz，发送端及对应的PLL把这些并行数据串行，然后以32 Mb/s的速率把它们发送出去。发送端的PLL会锁住系统时钟，且它的时钟相位仅受系统时钟的影响。由于我们的包格式，每32比特的数据对应的包有64个比特，因此，需要两个系统时钟才能传送一个32比特的字（word）。接收端也必须在两个系统时钟里完成数据的解串。

而接收端的设计要复杂得多。由于两个系统的时钟是独立的（参见图7.6），因此，接收端的PLL必须要从发送端送出的串行数据里提取出发送时钟。这意味着接收端必须找到输入数据包的边界，并且通过数据输入的速率提取出1 MHz的时钟送给接收端的PLL。利用这个提取出来的1 MHz的时钟，PLL首先与之同步，并且产生32 MHz的接收端串行时钟用来解串。在完成了解串之后，数据将受接收系统的时钟驱动。

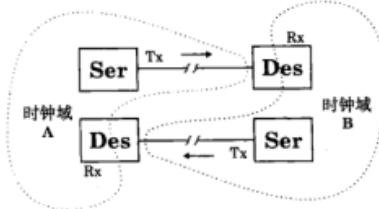


图7.6 跨越两个时钟域的全双工串行 / 解串器

为了处理发送时钟和系统时钟的异步问题，每一个串行器都需要一个FIFO来保存输入的并行数据。我们将根据串行数据的速率和并行数据的速率来设计这个FIFO的大小，以保证不会漏掉数据。

在接收端也需要有FIFO来解决同样的问题。在发送和接收两个系统时钟同步了之后，发送FIFO减小了丢失数据的可能性。在许多系统里，往往只有接收端才需要FIFO；在这个系统里，我们给发送端也加上了一个FIFO，将来可以把功能升级成支持任意的发送端时钟。

在后面的章节里，还会讨论更多关于时钟同步的话题。

7.4.3 串行 / 解串器的包格式

为了构造一组包含同步信息的串行数据流，我们将使用前面已经讨论过的数据包格式：

```
64'bxxxxxxxxx00011000xxxxxxxx00010000xxxxxxxx00001000xxxxxxxx00000000,
```

上面的x表示8比特有效数据里的某一个比特。先发送和接收的是MSB。为了给接收端的PLL提供一个同步的时钟，我们将在这个串行数据流中寻找从2'b11到2'b00的递减计数值。

为了讨论起来简单，假设系统循环发送4个字节。这4个字节分别是ASCII码的a, b, y和z。它们对应的编码是8'h61, 8'h62, 8'h79, 8'h7a。

则对应的串行数据流是这样的：

```
//      'a'      pad 3  'b'      pad 2  'y'      pad 1  'z'      pad 0  
64'b01100001.00011000.01100010.00010000.0111001.00001000.0111010.00000000
```

编码后的数据流一共有 $8 \times 8 = 64$ 个比特。

为了满足测试这个Verilog设计的需要，我们需要按32 Mb/s的速率发送这个数据流。为了简单起见，重复发送即可。

接下来，我们将会给读者展示一个基于软件的编程思想的，行为级（或者说基于过程的）的Verilog PLL模型，这个PLL能锁住串行时钟的频率。这个模型是不可综合的，在后面的章节里，我们会讨论如何把这个不可综合的模型修改成可综合的。下面一节的内容是讨论用软件的思想来进行Verilog的编程的话题。

7.4.4 行为级PLL同步

如果用Verilog里的for语句来对数据流进行采样计数，再用while来控制循环执行的条件，那么这个行为级的同步模型，或者更准确一点，这个过程化的同步模型实现起来就很简单了。在这个模型里，我们会使用到结构for(j = j; j == j; j = j)。在实际的工程里，综合工具要求for的迭代次数是常数。为了实现循环不停地发送数据，我们把while的条件写成1: while(1)。

下面是这个过程化模型的部分代码，读者会注意到这个模型并不是彻底地用行为级代码写出来的。它还是包含和比特相关的RTL过程操作。这个模型的主要代码如下：

```
integer i, j;
reg CurrentSerialBit; // A design data object.
reg[63:0] Stream; // For now, this is a fixed data object.

...
// Concatenation used for documentation purposes, only:
Stream = {32'b01100001.00011000.01100010.00010000,
           32'b01110001.00001000.01110101.00000000};

begin : While.i
while(1) // Exit control of this loop will be inside of it.
begin
    for (i = 63; i >= 0; i = i - 1) // Repeats every 64 bits, if no disable.
        begin
            #31 CurrentSerialBit = Stream[i]; // The delay value will be explained.
            (do stuff with CurrentSerialBit ... then disable While.i)
        end // for i.
    end
end // End named block While.i.
... // Pick up execution here after disable While.i.
```

由于while(1)总是打开的，因此，实际上控制while(1)的条件是在这个循环之外的While.i。当执行到了disable While(1)之后，包含在这个命名为While.i的块begin和end之间的代码都不再被执行，而是从这个块结束的地方接着执行。前面曾经提到过，这和C语言里的break语句的作用很类似。

Disable语句的作用是：在Verilog里，任何过程化的语句都可以放在begin和end之间，在begin之后可以给这段代码起个名字。disable通过调用这个名字来取消执行这一段代码。和C语言里的goto一样，应该尽量避免使用disable。因为disable使得代码的执行顺序变得复杂，当我们需要调试代码和对代码进行优化升级的时候，我们的工作会变得很困难。

在完成了上面的代码后，接下来研究怎样使得PLL和串行数据流同步。那段代码里的for的循环周期是 $31\text{ ns} \times 64\text{ bit} = 1.984\text{ }\mu\text{s}$ ，这个周期是1MHz时钟周期的一半。因此，要用两个系统时钟才能处理一个并行数据。

由之前的练习，我们知道哪个PLL会产生32MHz的时钟并且不断地监视1MHz的ClockIn时钟。在本章的这个设计里，只要PLL接收到了采样指示信号的上升沿，它就会根据ClockIn的边沿来对它产生的时钟频率进行微小的调整。因此，我们必须保证PLL的输入时钟ClockIn是准确的1MHz，同时，我们还应该产生周期性的采样命令信号。

在收到了每一个数据包之后，都会发出一个采样的指示信号。通过检测帧间隔的2比特递减计数的LSB的翻转，产生一个内部的时钟（EClock）并把这个时钟提供给PLL。因此，每两个EClock会采样一次。把EClock连在PLL的ClockIn输入端上。图7.7是一个时序示意图。

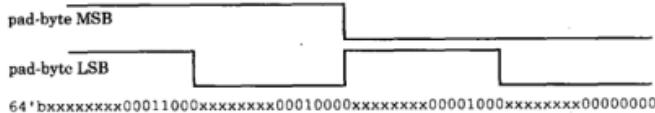


图 7.7 EClock 的频率两倍于包的频率

我们要利用数据流里的递减计数来提取出EClock。首先，检测3个连续的0，接下来是nn的格式，它是一个2比特的计数值，再接下来又是3个连续的0。这些帧间隔之间的是8比特的有效数据。每隔一个数据字节，帧间隔里的计数值就减1。再次说明，先发送的串行数据流里的MSB。

我们这样来建立一个是否同步的标准：只有我们至少连续4次正确地读到了递减计数值且最后以2'b00结束之后，才认为实现了数据同步。在实现了同步之后，把计数器的LSB的逻辑电平赋值给EClock。这相当于把EClock初始化成1'b0。

如果根据计数值的判断，发生了失锁。我们将停止发送采样命令；并且把EClock的电平保留在失锁前的那个电平上。直到再次锁定，才去再次调整EClock。此时，虽然发生了失锁，但是PLL仍然按照失锁前最新的调整结果去产生时钟。而用不用这个时钟，是其他逻辑的问题了。

由判决同步和失锁的标准，问题的关键在于如何鉴别数据流里的8'b000nn000，nn代表2比特的计数值。

为了解决这个问题，我们把这个模型循环的各种条件以及相关的细节画在图7.8所示的流程图上。

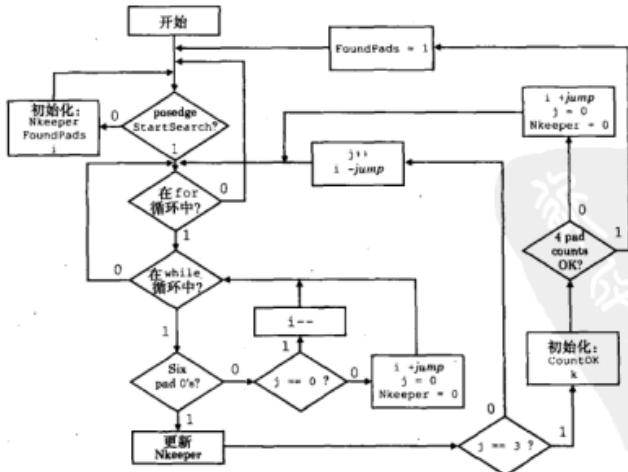


图 7.8 FindPatternBeh 的流程图

根据这个流程图，数据是串行输入的。因此，我们可以用一个for语句来检查输入的每一个比特，把它们的编号变量叫做i，i被保存在一个6比特的寄存器里。由于Verilog里的reg类型默认是无符号的，因此，i的计数值是从0数到63的单向计数。

首先来考虑这种情况，假设i的当前值使得MSB + 1个字节 = SerVect[63-8] = SerVect[55]，即确认计数字节间隔的首个0。采用Verilog实现的代码段如下。

```
// Assume SerVect is a saved, serial 64-bit vector:  
reg[5:0] i; // i traverses the serial stream (64-bit vector).  
reg n1Bit, n2Bit; // Two 1-bit regs to hold the expected nn pad count.  
...  
FoundPads = 1'b0;  
begin : While.i // Name the block to exit it.  
while(1)  
begin  
if ( SerVect[i]==1'b0 && SerVect[i-1]==1'b0 && SerVect[i-2]==1'b0  
&& SerVect[i-5]==1'b0 && SerVect[i-6]==1'b0 && SerVect[i-7]==1'b0  
)  
begin  
#1 FoundPads = 1'bl; // 1 means true here, for later use.  
#1 n1Bit = SerVect[i-4]; // Save the padded nn = {n2, n1} value.  
#1 n2Bit = SerVect[i-3];  
disable While.i; // Exit the while block, if found.  
end // if.  
...  
i = i - 1;  
end // While.i statement.  
end // While.i named block.
```

研究上面的代码。注意到并没有出现前面例子里的j。确定你理解或i的当前值指向MSB间隔字节的首个0时，上面这段代码与帧类型的关系。接着，移动i：假设在该类型，i的当前值现在位于比特23。如果这样的话，由if语句中大的`&&`表达式，搜索部分选出的SerVect[23:16]中的所有0，看能否捕获nn计数值。

While(1)循环控制里的`i = i - 1`。说明如果在当前(i)位置的类型匹配不成功，则`&&`类型匹配测试将会在SerVect变量的下一个重要性较低的(`i - 1`)个位置上执行。这是因为假定数据流的USB先到达，这样的话，随着时间推移，假定集中考虑的位置上重要的比特越来越少。当然，在上面SerVect的比特23这个特例中，匹配成功，While_i块被禁止执行，不会再在`i - 1` = 比特22时执行新的if语句，如上面代码所示。

这段代码里用到的赋值都是阻塞赋值。这段代码如何被解释成实际的电路我们并不关心，现在我们关心的重点只是搭建出一个行为级的模型而已。代码里的延迟仅仅是为了在仿真波形里区分出信号的边沿而已，赋值发生的真正时刻是在对应的语句被读走之后。这和软件的处理方式是一样的。在后面的章节里，我们还会继续修改这个模型，一些赋值会被升级成非阻塞赋值。

我们也可以用下面这种嵌套6个if的方式来完成找帧的逻辑。

```
if (SerVect[i]==1'b0)  
if (SerVect[i-1]==1'b0)  
...  
...
```

但是这种编码风格并不好。我们知道，`&&` 表达式是按从左到右的顺序来执行的，如果最左边的等式不成立，`&&` 的结果直接就是失败。用`&&` 实现的速度不会比嵌套的`if`慢。然而，综合工具在把这两种写法转换成电路时是有区别的。我们应该修改掉这种嵌套的`if`格式。先不去做这件事，把这件事记在脑子里。

无论如何，只有当搜索的6个填充0比特被找到，上面的代码才有可能设置`FoundPads = 1`。然后，这样的类型匹配可能是一种巧合，我们可能在数据比特的中间错误地匹配上`i`。因此，让我们进一步尽力地搜索。

接下来，声明一个名为`Nkeeper`的8比特宽的向量。然后在上面已经实现的功能的基础上检查是否出现4个连续的2比特的计数值。根据这个设计需求，上面的代码被更新成这样：

```

reg[7:0] Nkeeper; // Stores 4 2-bit nn values.
reg[5:0] i;        // Indexes into a saved 64-bit SerVect vector.
reg[2:0] j;        // Counts which of 4 assumed nn's we are on.

...
FoundPads = 1'b0;
i = starting value;
for (j=0; j<=3 ; j=j) // The for j increment is nonfunctional.
begin : While.i
  while(1)
    begin
      if ( six pad 0's; same as above )
        begin
          #1 Nkeeper[2*j+1] = SerVect[i-3]; // MSB of nn.
          #1 Nkeeper[2*j]   = SerVect[i-4]; // LSB.
          #1 j = j + 1;
          #1 i = i - 16; // Jump ahead to the assumed next pad byte.
          #1 disable While.i;
        end // if.
      i = i - 1;
    end // while.
  end // While.i block.

```

小问题：这段代码里的延迟是便于读者能够清晰地理解仿真事件的执行顺序而有意添加的。当真正在使用这个模型时，这些延迟应该被去掉。而且，随着设计的升级，这个模型里的一些赋值逻辑会被升级成非阻塞赋值。对于阻塞赋值的情况，需要注意这些延迟加在一起不能超过一个时钟。否则，这个模型有可能不能正常工作。

这段代码基本上和被改写之前是一样的，唯一的区别就是它会在`j`小于4时循环4次。为了简化这段代码，对比一致时产生一个标志的逻辑没有加在里面。`j`的位宽不能小于3比特。因为如果`j`的位宽为2时，计数只能从0数到3。这个`for`循环就永远也退不出来了。

上面的代码还有一个问题：如果输入的数据比较特殊，这部分逻辑可能会不断地判断出满足帧的条件。同时，`Nkeeper`也会被有可能不是帧计数的随机值填满。例如，输入的数据是这样的数据流：它几乎全都是0，偶尔还有一些1。

为了检查连续的4个2比特的帧计数值。我们接着升级上面的代码。当没有找到能匹配的数据流时，`i`减1；如果找到了能匹配的数据流，则`i`应该被减去16，接着去检查下8个比特是否是下一帧的间隔。如果完成了上面的那一步，但是接下来没有找到能匹配的数据，则`j`应该清0，而`i`应该加15，从第一个比特之后的下一个比特重新开始检查帧格式。

升级后的代码如下：

```

reg[7:0] Nkeeper; // Stores 4 nn values.
reg[5:0] i; // Indexes into a saved 64-bit SerVect vector.
reg[2:0] j // Counts which of 4 assumed nn's we are on.

...
FoundPads = 1'b0;
i = starting value;
for ( j=0; j<=3; j=j ) // No increment.
begin : While.i
  begin(1)
    begin
      if ( six pad 0's; same as above )
        begin
          #1 Nkeeper[2*j+1] = SerVect[i-3]; // MSB.
          #1 Nkeeper[2*j] = SerVect[i-4]; // LSB.
          if (j==3) // We're done:
            begin
              #1 FoundPads = 1'b1;
              disable While.i;
            end
          // If j wasn't 3, jump ahead for another look:
          #1 j = j + 1;
          #1 i = i - 16;
          end
        else // No six-0 match this time.
          if (j==0)
            #1 i = i - 1; // First nn not found yet.
          else begin // We found at least one nn, but now failed:
            i = i + 15; // Drop back after jumping by mistake.
            j = 0; // Reset nn counter; we're not in a pad byte.
            end
        end // while(1).
      end // While.i.
    
```

当上面的代码被执行了之后，帧格式里4个连续的2比特递减计数值应该都已被保存在了Nkeeper里。由这些分析可知，对应的比特序号应该是 $16*j - 1$ ，而不是像代码里那样仅仅多数15。我们会在后面再来修改这段代码。除了这个问题，我们已经完成了找到包格式信息的设计了吗？

不，我们还没有检查这些计数值。为了确认已经找到了帧间隔的计数值，也就是说为了确认的确找到了串行数据帧的边界，应该检查这个递减的计数序列。如果这些保存下来的数值不符合递减的计数原则，则说明我们并没有找到帧的边界，还应该继续去同步。而当我们找到了帧边界之后，应该退出执行上面的代码。

在令 $\text{FoundPads} = 1$ 的地方加入递减计数的检查最容易。在 $j = 0$ 的时候做这个检查没有意义，因为 nn 的值只有一个。但当 $j > 0$ 的时候，就可以检查是否当前的 $2nn$ 比前一个 $2nn$ 小1。

首先，我们把初始化的条件放到另外一个对StartSearch下降沿有效的always块里。这样的话，即使两个always块对同一个信号进行了操作，由于两个操作发生在同一个信号的不同边沿，也可以避免产生竞争。除非由于延迟的问题，操作被从一个边沿延迟到了另外一边沿。但是这种情况在这个设计里不会发生，因为StartSearch不是时钟信号，只是一个搜索的有效使能。

```

always@(negedge StartSearch)
begin
#1 Nkeeper = 'b0; // Init count keeper every search start.
#1 FoundPads = 1'b0; // Move this init here.
#1 i = StartI;
end

```

整段实现搜索代码应该如下所示。

```

always@(posedge StartSearch) // Clocked, maybe sequential logic?
begin : AlwaysSearch
for (j=0; j<=3; j=j) // No increment. '<=' is relational operator!
begin : While.i
while(1)
begin
if ( six pad 0's; same as above )
begin
#1 Nkeeper[2*j+1] = SerVect[i-3]; // MSB of pad count.
#1 Nkeeper[2*j] = SerVect[i-4]; // LSB of pad count.
// Check whether done:
if (j==3) // We have 4 apparent nn values; do they count down?
begin
#1 CountOK = 2'b00;
for (k=1; k<=3; k=k+1)
begin
// Use concatenation to get 2-bit nn values:
#1 nPrev = { Nkeeper[2*k-1], Nkeeper[2*k-2] };
#1 nNow = { Nkeeper[2*k+1], Nkeeper[2*k] };
if ((nNow+1)==nPrev) #1 CountOK = CountOK + 1;
end // for k.
if (CountOK==3) // Total of 4 were OK; so,
begin // issue a pulse and stop everything:
#1 FoundPads = 1'b1;
#1 disable AlwaysSearch;
end
else begin // If not a down-count, start all over:
#1 i = i + 16*j - 1; // See * below:
#1 j = 0;
#1 Nkeeper = 'b0;
end // if CountOK.
end // if j==3
else begin // j not 3:
#1 j = j + 1;
#1 i = i - 16; // Jump ahead, for another padded nn.
#1 disable While.i;
end // else not j==3.
end // if 6 zero matches.
else if (j==0) // First nn not found yet:
#1 i = i - 1;
else begin // * This was not first apparent nn found.
#1 i = i + 16*j - 1; // Drop back after jump by mistake.
#1 j = 0; // Reset nn counter.
#1 Nkeeper = 'b0; // Reinit count keeper.
end
end // while(1) block.
end // While.i labelled block.
end // always AlwaysSearch.

```

这段代码能够正确识别出我们构造的那段串行数据包。如果再做一些小修改，对于任意的串行输入，它都可以找到符合我们设计的帧边界。

在光盘的Lab10目录里，有这段代码的源文件和对应的testbench，这个文件的名字叫FindPatternBeh.v。

7.4.5 行为级代码的综合

我们看到，这段代码的风格和C语言很类似。综合工具在综合这样的代码时，效率不高。实际上，代码FindPatternBeh.v是不可综合的。

不可综合的一个原因是代码里到处都是带有延迟的阻塞语句。我们可以做一个小实验，去掉延迟：代码仿真依然是正确的，综合可能也没有问题了。用下面这段代码来进行分析。

```
if (CountOK==3)
    ...
else begin // If not a down-count, start all over:
#1 i = i + IJump*j - 1;
#1 j = 0;
#1 Nkeeper = 'b0;
end
...
...
```

如果这个always的功能依赖于延迟#1才是正确的，那么对应的模块综合出来功能肯定是对的。

此外，如果用非阻塞赋值替换掉这里的阻塞赋值，则有可能会产生竞争条件。如果在用非阻塞赋值替换阻塞赋值的同时，同时修改对应的延迟值。例如，把j=0的那行代码写成#2 j<=0;，结果是仿真有可能是正确的，但是综合出来的电路是有问题的。因此，基于目前的代码，我们把所有的延迟都去掉就可以了。

除了综合时延迟带来的问题，如果要重写一个可综合的行为级模型，且这个模型的功能依赖于特定的延迟，这是一件很困难的事情。读者可以思考一下怎样修改这个行为级的模型，在后面将讲到可综合的电路实现时，思路会跟这个版本有一些具体的区别。

7.4.6 可综合的，基于帧边界的PLL同步

让我们再次回到同步这个问题：系统收到了串行的数据流，我们希望利用这个数据流产生一个同步的时钟。由于在这个工程里，一个包有64比特，因此，我们只考虑64比特数据流的情况。

用类似C语言的风格来实现这个功能：假设我们有一个动态的窗口在寄存器里保存当前64比特的串行数据流。接着，在这个窗口里找对应的帧边界。如果找到了帧的边界，就让时钟和这个有特殊含义的边界同步。如果在这个窗口里没有找到边界，则把窗口里的寄存器依次移位一个比特，再来找边界。由于这样的处理方式使得每一个比特的组合都能被检查到，只要窗口里有边界信息，我们就能把它找出来。

因此，我们需要保证的就是数据窗口的完整性。这样的处理可以保证我们检查到64比特采样数据里所有的32比特格式信息。对于采样的数据，我们不需要进行任何处理去改变它的原值，只有这样才能保证数据的格式信息不丢失。

首先，我们要确定帧的边界格式；接着，在这 64 比特串行数据里试图找出这个边界格式。对于串行数据流的处理就是这么多了。

对数据流进行处理的目的是让 64 比特正好对齐这个窗口。假如当前窗口对应的 64 比特数据没有包含完整的数据及协议格式，则进行一次移位，移进新值并移走旧值，再进行检查。直到这 64 比特是一个完整的数据包为止。

基于这种思路，我们接着完成这个功能的代码：

```
reg[63:0] SerVect = // The current 64-bit window to scan.
/*      60          50          40          30          20          10          0
 * 32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210 */
64'b01100001.00011000.0110010.00010000.01111001.00001000.01111010.00000000;
...
localparam[PadHi:0] p0 = 8'b0000_00_000; // The pad patterns.
localparam[PadHi:0] p1 = 8'b0000_01_000; // localparams can't be overridden.
localparam[PadHi:0] p2 = 8'b0001_10_000;
localparam[PadHi:0] p3 = 8'b0001_11_000;
...
if ( SerVect[55]==p3[7] && SerVect[54]==p3[6] && SerVect[53]==p3[5]
    && SerVect[52]==p3[4] && SerVect[51]==p3[3]
    && SerVect[50]==p3[2] && SerVect[49]==p3[1] && SerVect[48]==p3[0]
    && SerVect[39]==p2[7] && ... (total of 32 compares)
) Found = 1'b1;
else Found = 1'b0;
... (etc.)
```

在后面的章节，我们还会继续深入讨论这个问题。在本章里，这个设计就做到这里了。

7.5 练习 10：PLL 行为级锁定

在目录 Lab10 下完成这个练习。

练习步骤

在这个练习里，我们将完成利用 for 语句来采样串行数据流的练习。Thomas and Moorby (2002) 里介绍了 for, while 和 forever 之间的区别。实际上，for 可以完成其他两个语言结构的所有功能，因此，在这个练习里，我们将会大量的用到 for 语句。while 和 forever 的用法要简单一些，for 用起来稍为复杂一些，但是实现的功能也更多一些。

第 1 步。 在目录 Lab10 下面新建子目录 PLLsync。把 Lab08/Lab08_Ans/PLL 目录下的所有内容都复制到 PLLsync 目录下来。复制过来的设计里产生的 PLL 时钟驱动了三个不同结构的计数器。设计的顶层模块名叫 ClockedByPLL。

在完成练习 8 之后，也许你有一份自己的设计了，但是还是请你用练习答案目录下的源文件来完成这个练习。

重组设计文件。假设在这个设计中，同一个位置有太多的文件，新建一个名为 PLL 的子目录，将 5 个 PLL 模块文件移到这个目录中。这些文件分别是 ClockComparator, MultiCounter, 构成 PLL 的 VFO 子模块和 PLLTop.inc。文件 PLLTop.v 是新设计的顶层文件，里面包含了例化的子模块也要加入该目录。PLLTOP.inc 是一个包含常数定义的 include 文件。

在上一级目录（目录 PLLsync 下）新建仿真工具需要的文件列表，PLLTOP.vcs。试着用仿真工具调用这个文件列表编译一下，检查文件的位置有没有问题。

第2步。重命名并修改设计顶层。把原来那个调用了三个计数器的顶层模块名由ClockByPLL改成PLLSync。只保留行为级的那个计数器。这样，PLLSync的输出只有一个，而且就是那个行为级计数器的输出。完成之后，删掉已经没用的文件：DFF.v。

PLLSync的模块示意图应该和图7.9类似。类似的电路在练习8里被叫做ClockedByPLL。

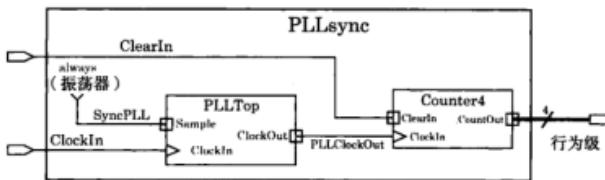


图 7.9 PLLSync 的方框图

把输入时钟ClockIn设成1MHz，仿真PLLSync的功能。

第3步。修改帧格式逻辑，产生时钟和采样指示。把FindPatternBeh.v（参考图7.10的左半部分）改名为EClockSample.v（E代表提取，extract）。把FindPatternBeh.v里的模块FindPattern改名为EClockSample。

修改模块EClockSample的端口，使它输出我们需要的提取时钟和采样指示。如果把寄存器Q的输出又再次返回给输入D，那么这个寄存器叫做翻转（toggle）寄存器，或者把它叫做T寄存器，如图7.10右半边画的那样。

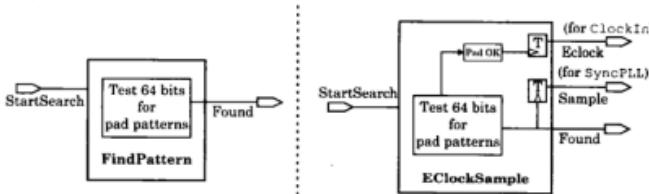


图 7.10 FindPattern 可以升级成 EClockSample，为 PLLSync 提供了输入。图中的 T 意为 toggle，每收到一个串行的输入，StartSearch 就会被触发

先不要尝试去完成本章后面所讲的没有循环的、可综合的找帧边界的办法，也不要担心testbench里的实际采样速率。只要用这个不可综合的FindPattern循环代码从串行数据里把EClock和采样指示提取出来即可。

EClockSample会持续的输出帧计数信息的LSB，把这个信号的名字改成EClock（在图7.11里，它和EClockWatch相连）。当Found为高时，EClock有效（处于同步状态）；当Found为低时，EClock不可信。

先不要把PLLSync的ClockIn和EClockSample输出的EClock相连。我们先记住，当EClock处于同步状态时，会用EClock去调整PLL的频率；如果EClock已经失锁了，则任PLL按照失锁前的状态自由的产生时钟。

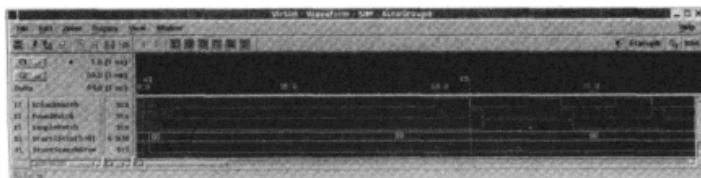


图 7.11 EClockSample 的仿真结果

7.5.1 练习后的思考

应该怎么判断锁定和失锁？

7.5.2 补充学习

阅读 Thomas and Moorby (2002) 的 4.6 节关于 disable 的内容。

阅读 Thomas and Moorby (2002) 的 6.5.1 节关于 bufif1 真值表的内容。

阅读 Thomas and Moorby (2002) 的 10.1 节和 10.2 节关于强度和竞争的内容。

阅读 Thomas and Moorby (2002) 附录 C.1 和附录 C.2 关于 Verilog 操作符优先级的内容。操作符的功能非常重要，读者务必要真正理解；而操作符的优先级不一定全部都要记住，可以通过在表达式里加上圆括号来避免容易混淆的优先级。

阅读 Palnitkar (2003) (可选)

3.2.1 节和附录 A 关于 Verilog 内建线性和强度的内容。

6.1.2 节关于线型声明的内容。

阅读 6.3 节和 6.4 节关于 Verilog 操作符的内容。

第8章 状态机和FIFO

8.1 状态机和 FIFO 设计

在本章中，我们将学习 FIFO 及其对应的控制状态机的设计。首先，学习一些能够简化 Verilog 代码的方法。

8.1.1 Verilog 任务和函数

设计的结构是靠模块实例中的层次来定义的。模块里可以包含各种功能单元，并按照需要对它们进行多次调用。然而有时候，过程中会使用重复的功能单元，而这些功能单元对设计的层次来说是不可见的。正如，使用操作符 “+” “&” 或 “==” 对数字或逻辑状态进行操作远比使用加法器、与门和比较器方便。同样，在 always 语句里使用定义好的寄存器比用各种门电路搭建寄存器要快得多。在 Verilog 中，为了进行简便操作，设计者常常会使用任务（task）和函数（function）。

在模块中，如果要使用这两类结构就要先对它们进行声明。虽然也可以通过层次关系来调用它们，但这是一种不好的设计方法，不鼓励使用。如果不同的模块要对它们进行重用，则需要专门用一个文件来存放各种定义。当某个模块需要调用它们的时候，使用`include 引入文件。因为任务和函数都不会描述设计结构，所以它们的声明里不能用本地线型（wire）把数据连接到其他地方；但是，它们可以使用本地的寄存器变量（reg）来保存临时数据。

任务和函数的区别在于复杂度和时序。这里讨论的任务和函数是用户自己定义的，而不是 Verilog 语言自带的系统任务和函数。

复杂度：函数比任务简单。任务里可以调用函数或其他任务；但函数只能调用函数而不能调用任务。因此，任务可能会比函数复杂。另外，执行函数时只能改变一个值，也就是它的返回值；事实上，函数仅仅是定义一个表达式，每次调用该函数的时候，都会重新计算表达式的值。任务执行的时候可以改变外部的寄存器对象的数值。因此，当一个任务被执行完之后，可能很多值已经发生了变化。

尽管非阻塞的延时赋值语句可能不是同步电路，但任务里可能会包含它们。而函数里只会有不包含延时的阻塞赋值语句。

时序：函数里不会包含延时。任务里可以包含自定义的延时或事件控制，但函数里不能。因为任务可以在特定的仿真时间里执行，它们可以用于描述硬件的并发性。

函数只是为了代替复杂的表达式，函数在仿真的0时刻开始执行并返回值，类似于一个简单的表达式的执行过程。

8.1.1.1 任务和函数的定义

定义任务时使用的端口列表和模块头很类似。定义时用 task 开头，以 endtask 结尾。下面是定义并调用任务的一个简单实例。

```

task SwizzleIt (output[3:0] SwizOut, input[3:0] SwizIn, input Ena);
begin
  if (Ena==1'b1)
    #7 SwizOut = { SwizIn[2], SwizIn[3], SwizIn[0], SwizIn[1] };
  else #5 SwizOut = SwizIn;
end
endtask
...
always@(posedge GetData)
  SwizzleIt( Bus2, Bus1, SwizzleCmd );

```

函数是一个临时寄存器类型表达式，函数执行完后返回表达式的值。因此，定义函数时要指定它的宽度，并且，它只包含输入信号。尽管函数只有输入信号，没有输出信号或双向信号，但定义输入信号的时候，必须使用关键字 `input`。尽管实际操作中很少在连续赋值语句中调用函数，但事实上这种方式是可行的。函数可以调用与时间独立的系统函数或任务，例如 `$display`。

调用函数或任务的时候，只能通过位置关系将实参传递给形参，而不能用名字映射的方式进行传递。

函数的定义实体是放在 `function` 和 `endfunction` 之间的。下面是一个定义和调用函数的例子。

```

reg[7:0] CheckSum;
function[7:0] doCheckSum ( input[63:0] DataArray );
  reg[15:0] temp1, temp2; // Just to illustrate local declarations.
  begin
    temp1 = DataArray[15:0] ^ DataArray[31:16];
    temp2 = DataArray[63:48] ^ DataArray[47:32];
    doCheckSum = temp1[7:0] + temp2[7:0] ^ temp1[15:8] + temp2[15:8];
  end
endfunction
...
#2 CheckSum = doCheckSum(Dbus);

```

8.1.1.2 任务的数据共享

因为任务执行的时候会影响它定义的数据对象，因此可能会出现在同一个仿真时间周期里对同一个任务有两个不同调用的问题。两个不同的调用会对同一组任务数据进行操作（任务只定义了一次；两个不同的调用同时作用在一个任务定义上面），因此，并发的操作会导致无法预期的事件行为，可能会对仿真的硬件造成致命错误。

为了避免对定义的内部数据进行共享并允许迭代，任务和函数会被定义为 `automatic`，表示它们的数据是复制后按照迭代执行过程的顺序放入仿真CPU的堆栈里。定义为 `automatic` 可以避免在不同任务或函数调用时本地数据进行共享。在 C 语言中迭代调用函数对本地变量进行操作时也是相同的原理。

可以参考随书所带的光盘中 Lab1 目录下的例子：Find3Mod.v，来学习关键词 `automatic` 的使用方法。`automatic` 的功能在不同的 EDA 工具中不是完全相同的。因此，为了代码的重用性，应该尽量要少用。

8.1.1.3 任务和函数是有名块

最后，任务和函数是有名块。可以使用任务的名字在任务内或调用它的任意地方停止它。任务和函数也可以包含有名块。函数只能在内部停止，因为它会在 0 时刻执行。然而，可以用表达式停止函数内部的有名块。

8.1.2 可综合的 PLL 同步函数

在第7章中，简要提出了用可综合的方式改写FindPattern模块的办法。这个办法是用一个包含32条表达式的if语句来检查64比特数据流的格式。我们也可以这样来做，调用4次函数，每次检查8个比特。

例如，下面的函数checkPad()使用for循环来执行检查。程序运行时，函数在仿真的0时刻被过程化执行，因此它可以写成软件风格。不必担心仿真和综合会不一致。当然，这不包含有延时的情况，但我们假设可综合的FindPattern不会引起仿真对象的更新。

```

function // 64-bit vector      8-bit pad pattern    offset in the stream
checkPad ( input[VecHi:0] Stream, input[PadHi:0] pad, input[AdrHi:0] iX );
reg OK;           // Flags pattern match.
integer i;        // i is the data stream offset.
      , j;        // j is the pad-byte pattern offset.
begin
  i = iX;          // Init to stream MSB to be searched.
  OK = 0;          // Init to failed state.
begin : For // Capitalized, this is not a verilog keyword.
  for (j=PadHi; j>=0; j = j-1)
    begin
      if (Stream[i]==pad[j])
        OK = 1;
      else begin
        OK = 0;
        disable For; // Break the for loop.
      end
      i = i - 1;
    end // for loop.
  end // For.
  checkPad = OK;
end
endfunction

```

注意“disable For;”这条语句，它停止了名称为For的语句块里的所有操作，然后执行最后一行语句“checkPad = OK;”这条语句结束函数的执行，因为Verilog中只要函数被赋值，它就立即返回该值。

两个问题：

- 将索引变量定义成integer，因为在有符号比较中会使用for循环结束变量：为了结束for循环，表达式的值必须小于0。综合在优化过程中会去掉32比特的integer中没有用到的比特，只剩下了一个足够大小的寄存器。
- for循环是放在一个被幽默的命名为For的程序块内。这是合法的。因为Verilog是大小写敏感的，Verilog中所有的关键字都是小写。对程序块命名是为了在不匹配的时候使用 disable语句来触发提前终止仿真运行。

假设我们在声明本地参数时采用下画线连接字节的方式，如：

```
...
localparam[PadHi:0] pad.00 = 8'b000.00.000;
localparam[PadHi:0] pad.01 = 8'b000.01.000;
localparam[PadHi:0] pad.10 = 8'b000.10.000;
localparam[PadHi:0] pad.11 = 8'b000.11.000;
...
```

函数 checkPad()可以用下面的方式调用：

```
...
if (    checkPad(Stream, pad.11, OffSet-(1*PadWid))
    && checkPad(Stream, pad.10, OffSet-(3*PadWid))
    && checkPad(Stream, pad.01, OffSet-(5*PadWid))
    && checkPad(Stream, pad.00, OffSet-(7*PadWid))
)
    FoundPads = 1;
else
    FoundPads = 0;
```

这比在 if 语句里进行 32 个等于比较的可读性（可重用性）要强。

参考 Lab11 目录下的 FindPattern.v，里面有上述所有的函数调用的实现。

8.1.3 并发的 fork-join

我们学习了由不同 always 块或 initial 块提供的孤立、并发、独立的操作。另外，还学习了由顺序块提供的逐行执行的过程操作。我们看到了过程评估和并发评估的延时块赋值语句的不同点。Verilog 提供了一种语句之间带受控并发性的结构。这种结构被称为平行块 (parallel block)，或者 fork-join 结构。它是仿照 UNIX 操作系统中 fork 系统的调用进行建模的。

fork-join 的结构目前是不可综合的，但可以用于仿真。由于通常 IP 是不会被综合的，所以可以在仿真中用 fork-join 结构来描述将用于其他综合网表的 IP 核的硬件模型。

只要能使用过程块的地方就能使用 fork-join 块。在 fork-join 块里能包含任意数目的语句。当仿真并发的执行这些语句时，它们之间的初始化顺序通常是未知的，事实上是随机的。fork-join 块的特殊之处是它有个等待点，所有并发语句会在等待点上相互等待、重新同步。

实际上，fork 之后的语句都是独立执行的线程，当它们中最一个执行完毕后，它们在由 join 指示的单时间点归拢在一起。归拢操作应用在 join 后面的顺序语句上。

在下面的代码段中，仿真过程里 DataBus 上面会有很多毛刺。如果不能通过修改代码段中的延时来解决这个问题，毛刺就会一直存在。

```
always@(posedge Clk)
begin
#1 DataBus[0] <= 1'b0;
#2 DataBus[1] <= 1'b1;
#4 DataBus[3] = 1'b0;
#3 DataBus[2] <= 1'b1;
OutBus = DataBus; // Some OutBus bits change before others.
end
```

Verilog 中，带延时的非阻塞赋值语句是并发执行的，但是在有的仿真器里，仿真的结果可能会不正确。无论如何，在 fork-join 块里读取结果能避免毛刺。

```

always@(posedge Clk)
begin
  fork
    #1 DataBus[0] <= 1'b0; // These could be blocking assignments; they
    #2 DataBus[1] <= 1'b1; // still would be scheduled concurrently.
    #4 DataBus[3] = 1'b0;
    #3 DataBus[2] <= 1'b1;
  join
  OutBus = DataBus;      // All OutBus bits change together.
end

```

当你在仿真时需要多条赋值语句同时执行，并且必须使用过程延时时，请考虑使用 fork-join 块来实现。其他时候避免使用这个结构。

8.1.4 Verilog 的状态机

状态机有两大类：Mealy 型和 Moore 型。Moore 型状态机的输出只与当前状态有关，而 Mealy 型状态机的输出不仅取决于当前状态，还受到输入的直接控制，并且可能与状态无关。例如，状态机可能在几个状态下循环，但一个独立的器件可能不是随时都准备好了来接收状态机的输出。如果这样的话，另一个器件可能把状态机的部分或所有输出都置为高阻或使用复用器转换输出。这样的状态机是 Mealy 型状态机。这里我们不再继续深入讨论这个特性，因为状态机的本质是状态如何改变，而不是输出逻辑由什么控制。

状态机最少包含一个状态寄存器，状态寄存器的值随着时间的改变以固定的方式改变。一个最简单的状态机的例子是翻转触发器，例如上一章我们使用的波纹计数器。如果当前状态值为“1”，下一个时钟周期时，状态值会变为“0”。如果当前状态值为“0”，那么下一个时钟周期时，状态值会变为“1”。任何一个数字计数器都是简单的状态机。但通常来说，工程上用的状态机要复杂得多。我们使用泡泡图（bubble diagram）或流程图来描述状态机的复杂功能。在现代微处理器的数据手册里有大量的这类图表。

当用 Verilog 来描述一个简单状态机的设计时，应将状态寄存器的控制和状态机状态里的组合逻辑分开，这是一个良好的编码风格。这种分离并不是 Verilog 要求的，但它简化了状态机的理解和维护，通常也减少了所需的程序量，如图 8.1 所示。



图 8.1 Verilog 状态机的抽象图。分离功能使得在逻辑部分的代码中可以使用阻塞赋值语句

状态更新逻辑包含状态寄存器，不能被外设读取。组合逻辑使用输入和当前状态值来对输入赋值，并改变状态机的下一状态。

8.1.5 FIFO 功能

FIFO是先入先出(First-In, First-Out)的首字母缩写。先写入FIFO的数据会先读出来，有点类似流水线操作，先流入的会先流出来。

FIFO是一类寄存器或存储器堆栈。与由任意指针或地址值独立计算的存储器相反，堆栈结构是典型的由当前值计算地址的存储器。与FIFO相反的结构是LIFO(Last-In, First-Out, 后人先出)也很容易理解。LIFO是标准的微处理器堆栈：最后进入堆栈的值最先被弹出来。

LIFO可以用单个寄存器、堆栈指针控制，因为推入LIFO的数据的位置与弹出的数据的位置相同。然而，FIFO更加复杂，因为和流水线类似，它要控制首尾两端。LIFO可以只有一个堆栈指针，而FIFO必须有两个不同的指针，分别指向读数据和写数据的地址。

如图8.2所示，FIFO的存储空间包含由水平方向的矩形表示的n个寄存器和预定义方向的数据流。数据可能以一种速率写入FIFO，以另一种速率读出。FIFO的一种应用是作为芯片里两个不同时钟域之间的缓存。突发的读写操作被FIFO缓解了，这样，数据在两个时钟域上都能以令人满意的速率传递，而不需要在每个时钟节拍上等待另一个时钟。FIFO的其他应用是在RS-422或以太网串行链路上，这两种链路通常都需要在不同的时钟域之间进行通信。

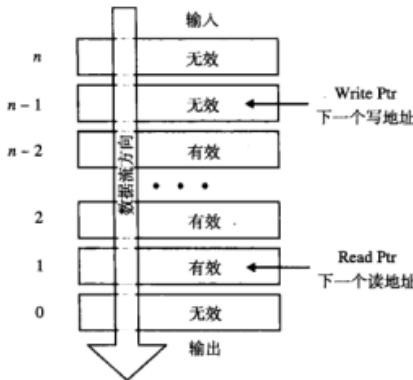


图8.2 先入先出功能赋予FIFO寄存器文件特殊的指令

在表示FIFO的读写指针时有两种主要的习惯用法：(a)这些指针被看做是指向FIFO中下一个有效的地址(寄存器)；或者(b)它们被看做是指向最常用的地址。本章中，我们采用前一种用法，即指针指向下一个地址。

读指针如图8.2所示，必须指向FIFO中的有效数据，即下一个被读数据；写指针必须指向没有使用的寄存器，即下一个数据写入的位置。这样的话，写指针指向当前无效的数据，通常是被读出的数据，因此所指寄存器里没有另外的值。

如果数据流由上到下的层次如图8.2所示。在指针被调用后，它们都会向上移动。当写指针到达寄存器 n ，即 FIFO 的顶部时会发生什么事？很简单，如果寄存器的底部没有有效数据，它会绕过寄存器的顶部回到寄存器的底部。我们在前面无符号 reg 类型计数器绕过最大值回到 0 时已经看到了同样的过程，而事实上，无符号计数器可以用来控制 FIFO 的读写指针。

FIFO 的构成被分成了如图 8.3 所示的几部分。有一组数据存储寄存器、一个读地址和一个写地址指针以及控制各指针值的无符号计数器。状态机可以中断或继续计数过程，并监视当前计数值。

对 FIFO 寄存器的地址转换可以用于将计数值转化为格雷码地址。这样，寄存器地址可以使用编码后的值，而不是简单、顺序的二进制计数值。图中右侧垂直方向的小箭头指明计数只能沿一个方向进行，这里是向上方向。如果计数能沿两个方向，即既能向上计数又能向下计数，FIFO 里的数据流就没有确定的方向，这个 FIFO 就不成立了。把计数值到地址的转换封装到 Verilog 的任务里（task），转换的原理可以任意改变，而不需要更改 FIFO 的实现。

8.1.6 FIFO 操作的细节

在我们尝试用 Verilog 描述 FIFO 之前，先仔细观察它操作的方式。首先，假设 FIFO 是空的，就是说所有的数据都已经被读出，还没有新的数据写入。描述它的方式取决于读指针什么时候会读取最后一个有效数据。

例如，假设写指针指向如图 8.3 所示的从顶部数下来第二个寄存器，那么 FIFO 看起来就像图 8.4 中 Empty₁ 的样子。

水平方向上的箭头放在指针所指的寄存器上，垂直方向上的小箭头指示数据流规定的流向。用“W”标记的写指针能自由地向上移动，用“R”标记的读指针在 j 位置读取数据后，不能移动到 $j+1$ 位置上，因为那个位置上的数据是无效的。

在上面描述的那种情况下，假设写入了一个写数据。FIFO 不再为空。这个数据将会被写入寄存器 $j+1$ ，W 移动并指向下一个无效寄存器 $j+2$ 。现在我们可以读取 $j+1$ 位置的数据了，因此 R 应该指向 $j+1$ 。此时的 FIFO 如图 8.5 所示，图下注明了 Empty₁+W。

我们来看另外一种情况，和 FIFO 空是等效的。假设当写指针指向寄存器底部时 ($n=0$, 参见图 8.2)，最后一个有效数据已经被读出。接着，读指针必须无效，并再一次不指向任何地址。我们通过将 R 放在 W 的右边来指示这种情况，因为每当 W 进行一次写操作，R 将会准备好去读这个被写的寄存器。因此，第二种 FIFO 空的情况可以描述成图 8.6 中的 Empty₂。

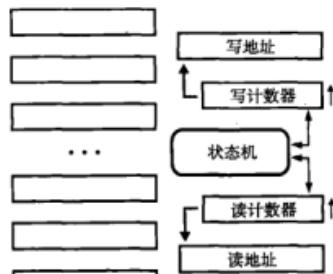


图 8.3 FIFO 的构成部分

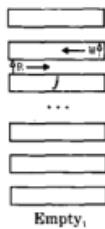


图 8.4 空 FIFO 的寄存器寻址

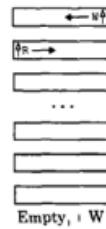


图 8.5 几乎空 FIFO 的寄存器寻址

这种情况下，当下一个写操作发生时，FIFO 不再为空，W 将从 0 位置移开，指向寄存器 1 位置，R 将会有效，指向寄存器 0 位置。这个过程如图 8.7 所示，用 $\text{Empty}_j + W$ 表示。

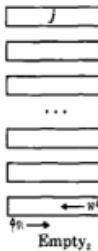


图 8.6 空 FIFO 的寄存器寻址

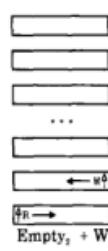


图 8.7 几乎空 FIFO 的寄存器寻址

很好，我们可以从这里猜到无论什么时候 R 都比 W 位置低，因为 R 永远不能向上移动到与 W 指向相同的寄存器。因此，R 永远不会跨过或超过 W。

然而，此时看看 FIFO 的另外一种情况，即当它为满的情况。这表示最近没有读出足够的数据，所有可能使用的寄存器都写满了，因此不允许继续写数据，此时 W 指针必须无效。

假设这种情况就发生在 W 向寄存器 j 写入了最后一个数据时，如图 8.8 所示，由 Full_j 表示。

噢，我们猜错了。很容易发现，现在 R 恰巧在 W 下一个可能位置的前面。W 不能跨过 R，因为向有效的寄存器 $j+1$ 写数据是不允许的。

如果现在有一个读操作，FIFO 不再是满的，寄存器 $j+1$ 会空出来，此时的情况如图 8.9 所示，由 Full_j-R 表示。

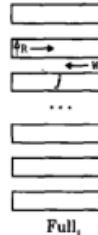


图 8.8 满 FIFO 的寄存器寻址

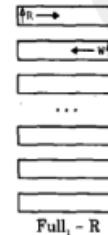


图 8.9 几乎满 FIFO 的寄存器寻址

读操作执行之后，R指向 $j+2$ ，W获得许可指向 $j+1$ ，此时可以进行写操作了。

最后，让我们看看另一种满的情况，即R在FIFO的底部——寄存器0位置上。如果是这样，W先指向无效数据，一旦读操作使得FIFO空出一个寄存器，W就会指向寄存器0位置。FIFO满的情况如图8.10所示，由Full₂表示。

Full₂中W实际上没有指向任何地方，因为W是无效的。但是，我们知道它的第一个可能有效的位置，它已经准备好绕过寄存器的顶端到达寄存器的0位置，这个过程只需要一次读操作就可以完成。

读操作过后，FIFO不再为满，变为如图8.11所示的情形，用Full_{2-R}表示。



图 8.10 满 FIFO 的寄存器寻址

图 8.11 几乎满 FIFO 的寄存器寻址

我们看到现在R指向寄存器1，W有效并指向寄存器0，即图中FIFO的底部。

与FIFO有数据传递的其他器件很关注FIFO的空或满情况。FIFO满了后，向它提供数据的器件必须停止传递数据；FIFO空了后，接收FIFO输出数据的器件必须停止接收数据。因此，通常FIFO必须提供输出标志位指示它是否为空或者满。

8.1.7 用Verilog设计FIFO

FIFO由存储器（寄存器阵列）和控制模块构成。控制模块被设计成状态机。正如前面介绍的，我们要把状态机的下一状态转移逻辑和其他组合逻辑分开。图8.12给出了一种我们可以使用的状态机电路示意图。



图 8.12 FIFO 的模块结构图

接下来，将讨论状态机的读写命令。这些术语指的是当读或写地址送给状态机后，状态机向RAM发出的命令。

记住会有外设使用FIFO，并向FIFO发送读写请求。状态机会遵循这些请求向RAM发送命令。当FIFO为满时，写请求会被忽略；当FIFO为空时，读请求会被忽略。

在对上面的 FIFO 进行详细描述时，当读写无效时会遇到读写指针的位置问题。这两个指针一个在空状态下无效，一个在满状态下无效。从表面上看，我们无法用 Verilog 来表示没有指向任何地址的指针。从另一方面来说，研究前面的细节揭示出当我们恰好处于 FIFO 的空或满的迁移状态时，指针 R 和 W，以及它们的下一位置，都在数字上被定义好了的。

因此，我们回避设计接近满和接近空，并将满和空处理成包含无效指针的特殊情况。

为了简洁，让 R 和 W 代表它们所指向的 FIFO 寄存器中标号为 n 的位置。接下来，当 $R == W-1$ 时，FIFO 再读一次就空了；当 $W == R-1$ 时，FIFO 再写一次就满了。这些都是很容易描述的算术关系。当然，在实际操作中， $R == W$ 状态对两个有效指针来说是禁止出现的，其中一个指针必须是无效的。而 R 和 W 的其他值都允许用简单算术来描述正确的操作，不需要特殊注意。每次使用后，指针值都简单地加 1。

这样，我们能把 FIFO 描述成只包含如下 5 个可能读或写转移状态的状态机。

normal	正常。读操作可能会使它转移到 almost_empty 状态；写操作可能会使它转移到 almost_full 状态。不允许进行其他状态的转移。对于一个包含 4 个寄存器的 FIFO，没有 normal 状态，a_empty 和 a_full 状态之间直接进行状态转移
a_empty	接近空。读操作使它转移到 empty 状态，写操作使它转移到 normal 状态
a_full	接近满。写操作使它转移到 full 状态，读操作使它转移到 normal 状态
empty	空。不允许读操作，写操作使它转移到 a_empty 状态
full	满。不允许写操作，读操作使它转移到 a_full 状态

这类状态机的状态转移图（泡泡图）如图 8.13 所示。

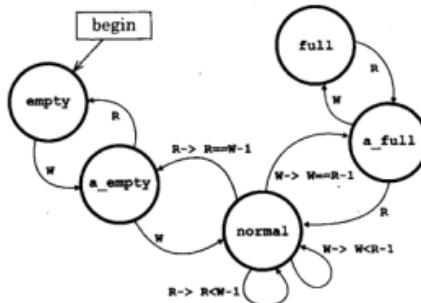


图 8.13 FIFO 状态机转移图。“begin”是加电或复位转移。假设 FIFO 有 5 个或更多的存储寄存器，R 和 W 是下一个动作的地址。

我们从定义状态及状态寄存器开始。对于有5个状态的状态机来说，主要是从3比特的二进制寄存器和5比特的独热寄存器中进行选择。在这里我们采用二进制编码。

状态机必须保存状态，因此需要一个时钟作用下的时序逻辑状态转移模块。把我们之前的约束用于编码，要把非阻塞赋值和阻塞赋值分开，因此在一个小时钟触发模块里使用非阻塞赋值语句实现状态转移。因为组合逻辑决定当前状态的下一状态，所以我们只需要在时序模块里用组合逻辑中的下一状态变量对下一状态进行赋值。状态转移逻辑可以用阻塞赋值语句。我们只在状态机模块里编写状态赋值的代码；FIFO寄存器将放在另外一个模块里。

我们还需要两个时序元件：读计数器和写计数器。它们必须由时钟驱动使得状态的变化仅仅发生在FIFO指针处于已知、一致的状态下。因此，仅允许在与地址计数器及状态寄存器更新相反的时钟沿读取组合逻辑的输出。如果状态寄存器在时钟上升沿进行更新，这表示地址计数器也在时钟上升沿进行更新，而只能在时钟为低电平时读取组合逻辑块的值，执行所有更新。计数器更新有多种方法，其中，在组合逻辑块中调用任务可能是最简单的一种方法：

```
task incrRead; // Called while clock is low.
begin
  @(posedge Clk)
    ReadCount = ReadCount + 1;
  end
endtask
```

调用这个任务的时候，它停在事件控制的位置，等待下一个时钟上升沿的到来。当下一个时钟上升沿到来时，读计数器的地址加1然后退出。类似的任务可以用来声明写计数器逻辑。

如果我们决定采用独热或格雷码地址计数器而不是二进制计数器，用任务来递增计数器的值并对其进行比较，比浏览、编辑各个隔离的组合逻辑代码实现的计数器修改更简单，并且在早期开发中出错少。

把注意力转到状态编码并忽略状态转移逻辑，目前我们得到下面的代码：

```
//
// Don't allow any other module to affect the state encoding,
// so, use localparams:
//
localparam empty = 3'b000, // all 0 = empty.
           aEmpty = 3'b010, // LSB 0 = close to empty.
           normal = 3'b011, // aEmpty < normal < aFull.
           aFull = 3'b101, // MSB 1 = close to full.
           full = 3'b111; // all 1 = full.
//
// The sequential block controlling state transitions:
//
always@(posedge Clk, posedge Reset)
  if (Reset==1'b1)
    CurState <= empty;
  else CurState <= NextState;
// End sequential state transition block.
```

我们不能在时钟触发的模块里对NextState进行复位，因为综合工具不支持同时在时序逻辑和其他非时序的组合逻辑里对NextState进行赋值。因此，必须在组合逻辑里为NextState设计复位。通常来讲，同步器的这个行为是件好事情，它帮我们避免竞争。

因为必须处理读、写请求，所以控制状态转移的逻辑块应该同时对状态变化和读写请求敏感。

组合块的第一部分如下所示，其中命名为“xxxReg”的变量是对输出连线xxx进行时序赋值的reg：

```
always@{ReadReq, WriteReq, CurState, Clk} // NOTE: Request, not Register.
if (Clk==1'b0) // Only read after a negedge of clock.
begin
  case (CurState)
    empty: // Combines reset unique conditions
```

```

// with a simple empty state during operation:
begin
if (Reset==1'b1)
begin
// Reset conditions:
FullFIFOReg = 1'b0; // Clear full flag.
WriteCount = 'b0;
WriteCmdReg = 1'b0;
ReadCmdReg = 1'b0;
NextState = empty;
end
//
// Generic empty conditions:
EmptyFIFOReg = 1'b1; // Set empty flag.
ReadCmdReg = 1'b0; // Disable RAM read.
// One transition rule:
if (WriteReq==1'b1 && ReadReq==1'b0)
begin
ReadCount = WriteCount; // Could also init to Adr 0.
incrWrite; // Call task, which blocks on posedge Clk.
WriteCmdReg = 1'b1; // Issue a RAM write.
EmptyFIFOReg = 1'b0; // Clear empty flag.
NextState = a.empty;
end
else ReadCount = 'bz; // Nowhere.
end // empty state.
a.empty: begin
...
end // a.empty state.
normal: begin
...
end // normal state.
a_full: begin
...
end // a.full state.
full: begin
...
end // full state.
default: NextState = empty; // Always handle the unexpected!
endcase
end // always.

```

这里可能会被忽视的一个问题是在这个代码块里读取了 WriteCount 的值，但 WriteCount 并没有出现在敏感信号列表里，在逻辑综合时可能会出现不需要的 latch。为了避免这个问题，在下一次设计中，将把 always @ (ReadReq, WriteReq, CurState, Clk) 改成 always @ (*)，以避免未来出现任何不愉快的意外。

再来考虑一个状态并为它设立转移规则。normal 状态看起来是个不错的例子。我们按如下方式进行。

只有进行读或写操作后，状态机才会发生状态转移。时钟有效沿到来时，如果 FIFO 既没有读，又没有写，可以考虑让状态机处于“idle”状态。对于状态机出现的一些问题，使用明确的 idle 状态是有用的，但对我们这里的设计，idle 状态是不必要的，因为我们没有把时钟的事件当做是与状态机相关的事件。我们只关心读或写操作。两个时钟沿为操作的有序执行和逻辑的转换提供了时间。

在normal状态下，对任何的读或写操作，只须检查下一状态是否应该是a_empty或a_full。除非是对状态机进行复位，我们的设计不会直接从normal状态转移到empty或full状态的。同时，我们也知道在normal状态下要求以下两个关系式必须成立：

写计数器的值 > 读计数器的值 + 1

读计数器的值 > 写计数器的值 + 1

因此，状态转移时跳出normal状态的最简单的方法是对计数器的新值额外加1，并将它与另一个计数器的当前值比较，比较它们是否相等。如果它们的值相等，状态机则跳出normal状态。

考虑到上面提到的所有问题，normal状态的组合逻辑 case 选择块应该如下所示：

```
normal: begin
    // On a write:
    if ( {WriteReq, ReadReq}==2'b10 ) // Concatenation.
        begin
            ReadCmdReg = 1'b0; // Disable RAM read.
            WriteCmdReg = 1'b1; // Issue a RAM write command.
            incrWrite; // Call task, which blocks on posedge Clk.
        //
        // Transition rule: Check for a.full:
        if ( ReadCount == WriteCount+1 )
            NextState = a.full;
        else NextState = normal;
        end
    // On a read:
    if ( {WriteReq, ReadReq}==2'b01 )
        begin
            WriteCmdReg = 1'b0; // Disable RAM write.
            ReadCmdReg = 1'b1; // Issue a RAM read.
            incrRead; // Call task, which blocks on posedge Clk.
        //
        // Transition rule: Check for a.empty:
        if ( ReadCount+1 == WriteCount )
            NextState = a.empty;
        else NextState = normal;
        end
    end // normal state.
```

为了降低比较2比特的复杂度，我们把它们拼成一个2比特的表达式。读操作的if语句应该嵌套在写操作的if语句的else部分，作为不满足条件的另一个选择。目前，这里省略了else部分，因为变量的值是受限的，并且将会增加一些代码帮助设计者阅读和理解。

需要注意的是，当在执行写操作时，表示写请求状态的比特可能被外部电路翻转。也就是说，normal状态里两个独立的if语句（读以及接下来的写）可能会出现竞争与冒险。如果这样的话，最好能用if-else语句或者嵌套的case语句。

在最后才进行状态转移的赋值，这是为了确保在状态转移之前当前状态下的所有操作都已进行完毕。

最后，注意到在上面的代码中，我们将一个计数器的值加1后与另一个计数器的值进行比较。“+1”操作可能是个问题，因为表达式的宽度变得不确定：表达式是整数（“1”）还是很小的寄存器变量？在Verilog中对变量赋值时，赋值结果采用目标变量的类型和位宽。然而上

文代码中的表达式没有赋给任何变量。整数是有符号类型，我们最不想看到的是表达式是一个负的计数值。为了避免由不同仿真器实现而可能带来的问题，在这个模型的下一个示例中，我们会先把结果赋给与某个计数器具有相同位宽的寄存器变量，再比较是否相等。

上面代码中省略了延时，因为阻塞赋值语句保证了赋值的先后顺序，而它与在每条语句前使用“#1”不会有明显的差别。此外，综合会在逻辑的不同分支引入多种不可预知的延时，并且无论如何，布局布线所反标的延时会覆盖模型中预设的延时值。

在代码中添加延时的一个原因是可以较为准确地调整其他设备连在状态机控制器总线的数据延迟。不过，在顶层加上延时信息会更好一些，如 FIFOStateM_header.v 所示。

```
module FIFOStateM
  #(parameter AdrHi=4) // 2** (AdrHi+1) registers. Default=32.
  ( output[AdrHi:0] ReadAddr
  , ...
  , Clk, Reset
  );
  ...
  reg[AdrHi:0] ReadAddrReg, WriteAddrReg;
  ...
  reg      EmptyFIFOReg, FullFIFOReg
            , ReadCmdReg, WriteCmdReg;
  ...
  assign #1 ReadAddr = ReadAddrReg;
  assign #1 WriteAddr = WriteAddrReg;
  assign #1 EmptyFIFO = EmptyFIFOReg;
  assign #1 FullFIFO = FullFIFOReg;
  assign #1 ReadCmd = ReadCmdReg;
  assign #1 WriteCmd = WriteCmdReg;
  ...
  
```

在单独的赋值语句中增加延时可以展宽仿真工具中显示的波形，从而让设计者更容易地查看仿真结果。然而，必须要让仿真工具能够理解，认识这些延时，仿真工具会把它们当成是设计的一部分。设计者还是需要经常检查一下这些延迟，有没有随着设计的需求而更新。通常我们应该把延迟加在顶层文件，通过连续赋值把延迟信息添加到端口上。

8.2 练习 11: FIFO

在 Lab11 目录下完成这个练习里的所有内容。

练习步骤

第 1 步。 使用练习 10 里的一个任务。将文件 FindPatternBeh.v 从 Lab10 目录下面复制到 Lab11 目录下，改名为 FindPatternTask.v。像往常一样，将模块名改成和文件名一样。在这个文件里，有一段重复的代码：

```
#1 i = i + IJump+j - 1;
#1 j = 0;
#1 Nkeeper = 'b0;
```

显然，两个完全相同的事件应该做相同的事情；如果必须改动一个，另一个也应该进行相同的改动。这是使用任务或函数的理想情况。因为包含了延时，所以它必须是任务。

将上面的代码封装成一个任务，然后在出现这段代码的两个地方分别调用这个任务。简单地运行仿真来验证它的正确性。

第2步。断言任务。在本练习中的各步中（包括前一步）的某处使用下面的 ErrHandle 断言代码，在出错或有可疑条件下对使用者提出警告。在各种可能操作下面进行测试（参见图 8.14）。Lab11 目录下有一个包含这个任务的文件示例。

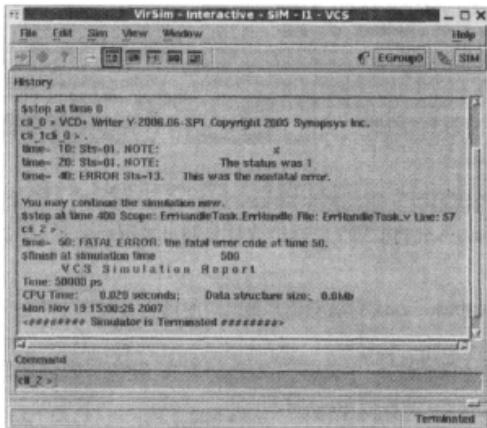


图 8.14 ErrHandleTask 的测试仿真

Sts 类型位宽为 4 比特，允许 16 种不同的状态情况。因为 reg 是无符号的，这也意味着把值传递给任务时，4'b1000 或大于 4'b1000 的值会用于表示负的状态值。Action 有 4 种可能值：

- 0: Sts 为 0 或 Msg 为无效值。这种情况是正常状态，任务不返回值。
- 1: Sts 为 -1 (4'hf)。这种情况是致命错误状态，仿真结束 (\$finish)。
- 2: Sts 的值小于 -1 (4'hf>Sts>=4'h8)。这种情况是错误状态，仿真暂停 (\$stop)，但可以继续运行。
- 3: Sts 为正值，(4'h1--4'h7)。这种情况是警告或信息状态，Msg 将打印在仿真控制台上，但不采取其他动作。

```

task ErrHandle(input[3:0] Sts, input[255:0] Msg);
reg[1:0] Action;
begin
  if (Sts==4'h0 || Msg=='b0)
    Action = 2'b00; // == 0.
  else if (Sts==4'hf) Action = 2'b01; // Sts == -1, 2's complement.
  else if (Sts>=4'h8) Action = 2'b10; // Sts < -1, 2's complement.
  else
    Action = 2'b11; // Sts > 0.
  //
  case (Action)
    2'b00: Sts = 0; // Do nothing.
    2'b01: begin
      $display("time=%4d: FATAL ERROR. %s"

```

```

        , $time,
        Msg);
$finish;
end
2'b10: begin
    $display("time=%d: ERROR Sts=%02d. %s"
            , $time,           Sts,      Msg);
    $display("\nYou may continue the simulation now.");
    $stop;
end
default: $display("time=%d: Sts=%02d. NOTE: %s"
                  , $time,           Sts,      Msg);
endcase
end
endtask

```

当然，这对仿真波形没有影响。然而，断言信息会出现在仿真器控制台窗口中，如图 8.14 所示。

第 3 步。 FIFO 的状态机设计。文件 FIFOStateM_header.v 里已经提供了模块头与部分设计。同时还包含了 testbench 的框架。将这个文件复制过来重新命名为 FIFOStateM.v，按下面步骤完成设计：

- 修改组合逻辑的 always 块的敏感信号列表，使得它对仿真中程序块读取的任意变量的变化都敏感。
- 完成其他状态的赋值与状态转移。对状态机进行仿真，检查地址生成是否正确。在 testbench 里使用 for 循环把 FIFO 写满，再把它读空，检验地址、状态寄存器和标志位操作的正确性。
- 对设计进行综合，先按面积进行优化，再按速度进行优化。

第 4 步。 将寄存器文件和 FIFO 状态机连在一起，构成完整的、实用的 FIFO。在后面的练习里，我们还会改进存储器和控制器的功能。现在，进行如下操作。

从目录 Lab07 里将静态 RAM 设计 Mem1kx32.v 复制到目录 Lab11 下。在 Lab11 下建立新文件 FIFO_Top.v 及新模块 FIFO_Top。在文件 FIFO_Top.v 里，实例化 FIFOStateM 和 Mem1kx32，将它们连接起来。在顶层为 FIFO 添加输入输出数据总线。把读地址和写地址合成一组复用的地址线，在顶层文件里用连续赋值语句和条件操作符来完成这项操作。

进行了充分的仿真实证你的 FIFO 能正常工作后（参见图 8.15 和图 8.16），综合这个 FIFO，再分别按面积和速度的条件来优化这个 FIFO。

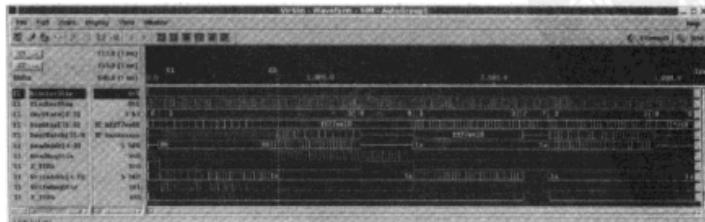


图 8.15 FIFO 仿真的一部分——不稳定，但明显避免了竞争

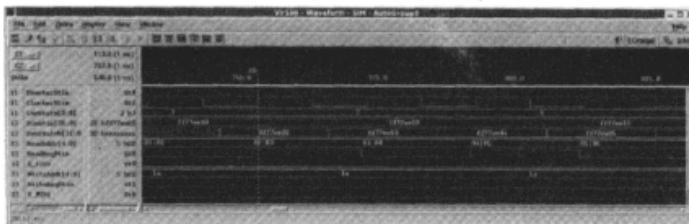


图 8.16 FIFO 仿真波形的局部放大，显示第一次从寄存器文件里读出数据的过程

第5步。如果时间允许，再检查一下状态机的组合逻辑块是否会出现前面提到的竞争状态。竞争状态可以通过下面的方式避免：使用一条语句在每个状态下的某个时间点进行检查，断定所要求的操作（或状态转移）应该是什么。

8.2.1 练习后的思考

跨时钟域的 FIFO 设计和使用格雷码有什么关系？

8.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 3.5 节中函数与任务部分。

阅读 Thomas and Moorby (2002) 的 4.9 节的 fork-join 部分。

阅读并理解 Thomas and Moorby (2002) 的 6.3 节练习 6.9 中简单存储器模型。

(选读) Thomas and Moorby (2002) 包含状态机建模的多种不同视角。如果想了解更多方面的内容，尝试阅读 1.3.1 节、2.6~2.7 节、第 7 章及附录 A.14~A.17 节。也可以看下面的“阅读 Palnitkar (2003)”。

阅读 Palnitkar (2003) (可选)

阅读第 8 章中任务和函数的内容。

附录 F.1 中的代码描述了一个可综合的 FIFO。使用计数器来跟踪 FIFO 状态是重要的，后面有些附录讨论了格雷码计数器与其他的 FIFO 细节。注意 Palnitkar 的附录 F 中的 FIFO 只有 4 个存储寄存器。

7.9.3 节和 14.7 节中有状态机模型。仔细阅读这些内容，学习如何实现控制。

第9章 事件

9.1 上升-下降延迟和事件计划

本章我们将会学习关于延迟和时序的知识。

9.1.1 延迟表达式的类型

本章中，我们会学习门级延迟和过程延迟，而路径延迟、器件内部延迟和开关级延迟的内容我们会稍后再讲。

常规延迟和计划延迟。两种表示延迟的方式分别是：常规（regular）和计划（scheduled）。在 Thomas and Moorby (2002) 的 4.7 节里，计划延迟被叫做内部赋值（intra-assignment）。虽然 Thomas and Moorby (2002) 的作者认为计划延迟是有用的，并且在 8.3.1 节里还列举了使用计划延迟的例子。但实际上，计划延迟用得很少。常规延迟出现在目标语句的左边，而计划延迟出现在赋值号（= 或 <=）的右边，下面是具体的例子。

```
#(delay) variable1 = value;           // 1. regular blocking.  
#(delay) variable2 <= value;         // 2. regular nonblocking.  
variable3 = #(delay) value;           // 3. scheduled blocking.  
variable4 <= #(delay) value;          // 4. scheduled nonblocking.  
#(delay) variable5 = #(delay) value;  // 5. both, blocking.  
#(delay) variable6 <= #(delay) value; // 6. both, nonblocking.
```

常规延迟将在延迟的时间达到指定的时间后，再去计算等式右边（RHS，Right Hand Side）的值，从而执行这个被延迟的语句。

综合工具在检查到非阻塞赋值的延迟后，都会给出错误信息或严重的警告提示。而阻塞赋值的延迟会被综合工具忽略掉。前面曾经提到过，综合工具不能实现延迟的具体值。尤其是对那些并发的，并且同时有时间关系的事件来说，综合工具执行有延迟的非阻塞赋值的顺序可能不符合设计的要求。

如果采用的是计划延迟，则会立即计算 RHS 的取值，知道延迟的时间达到指定的时间后，再赋值并完成这个赋值过程。Verilog 里的计划延迟和 VHDL 里的 transport 延迟是等效的。

在上面的例子里，有两种延迟都有的表达式。在稍后的练习里，我们会进一步接触到。

术语的区别

Thomas and Moorby (2002) 的 8.4.3 节介绍了常规事件（regular event）这个术语；这说明它和我们这里讲到的常规延迟（regular delay）是有区别的。这个在 Thomas and Moorby (2002) 里被叫做常规事件的术语，在 IEEE 1364 里其实是主动事件（active event）的概念。下面是详细的解释。

作者强烈建议大家不要使用计划延迟。这种操作就好像要用RC电路模拟数字电路的仿真一样，是不真实的。而且，这个结构在过程块的内部又创建了一个新的并发的线程，降低了仿真工具事件控制功能的效率。为了达到并发的目的，可以使用多个独立的 always 块或连续赋值语句来实现。如果写的代码不需要是可综合的，可以使用 fork-join。但是大家要记住，fork-join 只会在所有的并发语句都执行结束之后才会接着执行。

也许有的读者会觉得使用计划延迟可能看波形会更方便一些。然而，据作者所知，还没有仿真工具可以在波形上区分出计划延迟和常规延迟。这么做不但没有好处，还有可能把模型的延迟（建立和保持时间）和并不属于设计本身的计划延迟混在一起，从而有可能导致仿真的错误。

例如，假设一个受时钟驱动的 D 触发器被输入了一个数据，它的建立时间在波形里看起来很不明显，如图 9.1(a)所示。如果在这种情况下，认为增加了计划延迟，则看起来像是保证了建立时间的要求。但是，如果这样做的前提是这个设计已经没有其他的时序问题了。否则，这个人为增加的延迟可能会掩盖设计本身的时序问题。

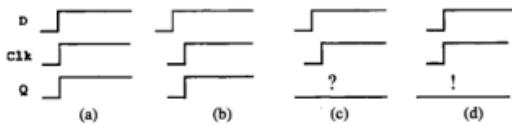


图 9.1 计划延迟会带来混淆。(a) 最初的波形；(b) 给 Clk 和 Q 添加了计划延迟；
(c) 外部事件的变化导致建立时间不满足，但计划延迟掩盖了这个问题；
(d) 发生了和(c)一样的外部事件，由于没有加计划延迟，使得问题被暴露出来

Thomas and Moorby (2002) 的 8.4.1 节指出，可以把有计划延迟的赋值看成是赋值的中间加入了对临时的、不可见的寄存器的赋值，从而达到了增加延迟的目的。比如说，对于一个延迟时间 d ，“ $x \leq #d y;$ ”等效于：“ $temp = y; #d x = temp;$ ”。在这个例子里，在插入延时 $#d$ 的过程中， $temp$ 对其他的赋值是不可见的。这种处理实际上浪费了仿真器的内存，有可能产生设计上的问题。由于这种延迟不可见，不能利用输入信号来取消它。因此，如果要用符合实际的惯性延迟来消除这个计划延迟就不可行了。

多值线延迟表达式。对 CMOS 工艺来说，门的电路结构是比较对称的。但是，对于上升和下降，即是变成 0 的延迟和变成 1 的延迟来说，还是有差别的。例如，容性的反应对于电源和地的响应时间是不同的。此外，NMOS 管变成 1 的时间很长，而 PMOS 管变成 0 的时间很长。为了体现这些差异，Verilog 的原语门和线连接时序表达式支持指定两个或三个延时值。例如：

```
assign #(3, 5) OutWire_001 = newvalue;
bufif1 #(3,5,7) GateInst_001(OutWire_001, InWire_001, Control_001);
```

如果是两个参数，它们的顺序是 (rise, fall)；如果是三个参数，它们的顺序是 (rise, fall, toz)。它们的含义分别是：从 0 到 1 的上升延迟，从 1 到 0 的下降延迟。如果器件支持的话，toz 的含义是变化到高阻需要的时间。后面我们将学习它们之间的具体差别。在过程赋值语句中不允许出现这样的延迟表达式，在过程语句里只允许有一个延迟参数。

总结起来，连接延迟表达式可以有 1~3 个延迟参数。它们的含义如下表所示：

#(every_delay)	可以表达所有延迟
#(rise_delay, fall_delay)	rise_delay 是变化到 1 的延迟 fall_delay 是变化到 0 的延迟
#(rise_delay, fall_delay, z_delay)	当参数为两个时，它们分别是 rise_delay 和 fall_delay z_delay 是变化到 z 的延迟

如果有到“x”的延迟或有两个到“z”的延迟，则最短的那一个延迟有效，这是由延迟的悲观处理原则决定的。

多值延迟表达式用在以下场合：

- 连续赋值语句
- 原语元件例化

特殊的计划。在我们开始学习 Verilog 的事件队列之前，还有一些问题需要仔细思考。首先，要知道，当连续赋值或过程块中的延迟是手工输入的时候，很多专门用来仿真大规模 VLSI 网表的仿真工具可能不能正确地仿真出惯性延迟。

如果仿真工具在过程块里遇到了 #0 的非阻塞赋值，仿真工具会在发现这个赋值延迟后马上安排当前的任务调度。当没有延迟的时间执行完之后，会执行到有延迟的任务。这有可能会改变事件的执行顺序。例如下面的代码：

```
x <= 1'b1;
y <= 1'b0;    // z gets the new 1'b1 value of x.
#0 z <= x;
```

当然，如果设计者写的是可综合的代码，就不会遇到这样的问题。

如果在赋值语句里没有增加延迟，则这些语句是按照排列的顺序来执行的，执行它们的时间间隔是 0。

```
x <= 1'b1;
y <= 1'b0;    // z gets the old value of x;
z <= x;        // x gets the 1'b1 originally scheduled.
```

如果对同一个信号有多个冲突的赋值，最后一个读到的值有效。

```
#2 x <= 1'b1;
#2 x <= 1'b0;    // x will be scheduled for 1'bz
#2 x <= 1'bz;    // at 2 time units from current.time.
```

一些工具可能不能正确地处理这样的情况。当然，如果你写的是可综合的代码，就不会有这样的问题了。但是有时，比如说是在写 testbench 的时候，还是可能用到这样的用法的。

9.1.2 Verilog 仿真事件队列

仿真的规范是内建在 Verilog 语言中的，这部分规范包括执行事件的队列，仿真时间的计算等。从现在开始，除非专门指出，时间这个词会被用来专指仿真时间。

IEEE 1364 指出，Verilog 语言的仿真基于分层的事件队列（stratified event queue）。符合 Verilog 规范的仿真器会先从没有延迟的事件开始，先执行它们，然后把时间设成 0，然后按时间的顺序依次执行各个事件。Verilog 的语言规范并没有规定当多个事件被安排在同一个时刻

来调度时，应该先执行哪个。只要是同一层（stratum）的事件，什么顺序都是可以的。因此，对于不同的仿真工具，执行的过程有可能是不同的，但是结果必须是一样的。如果设计工程师非常关心这个问题，那他就需要写出能够避免这个问题的代码。所以，要理解事件队列的分层是非常重要的。

图9.2是Verilog的层次事件队列示意图。2005年之前，IEEE 1364规定PLI的相关层是可选的，而现在是必须支持的。

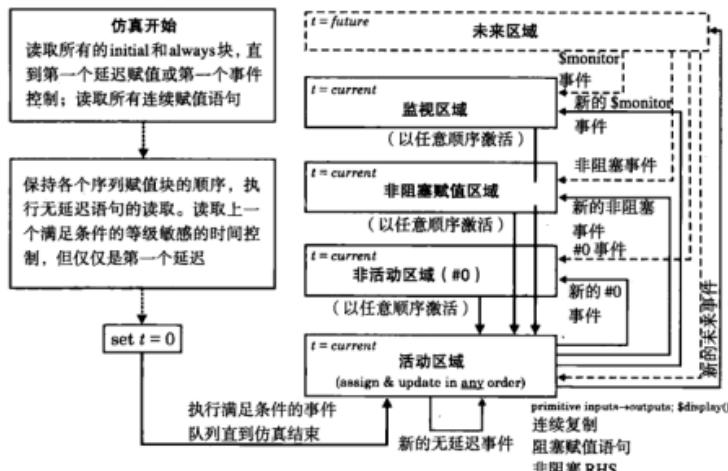


图9.2 Verilog的事件队列。右边是事件的队列，和左边的仿真初始化过程是分开的

记住这个概念：事件（event）也就是一个量的变化。这个变化由输出驱动引起（结构性的）或由赋值语句引起。如果仅仅是计算输入或等号右边的表达式不算是事件。

图9.2里的实线部分表示的是当前时刻对数据的存取，或当前的控制流中具体的一步。例如，图中的实线部分（To active in any order）的含义是在执行完了所有的活动事件之后（所有的新值都赋给了对应的变量之后），开始处理非活动的事件。实线的意思是所有的非活动事件都会被移动至活动区域。不管Verilog代码里这些语句是按什么样的顺序来排列的，或者不管按什么样的顺序移到非活动区域里的，当它们从非活动区域移至活动区域时，这些语句的信息都丢掉了。

有趣的是，尽管#0意味着零延迟，但是被增加了#0的事件是被放置在非活动区域里的。直到所有的活动事件被处理完之后，它们才是活动的。活动事件包括：没有延迟的阻塞赋值；作用于Verilog的原语输入上没有延迟的变化；或者是被延迟到当前时刻来执行的事件。

图9.2里其他的实线说明了活动区域可以触发新的非活动事件（新的#0事件），新的非阻塞赋值等。当所有的活动事件都处理完了之后，并且当前时刻不会再有新的事件了（从非活动区域的转换，非阻塞的赋值或监视区域），时间会进入下一个事件的时刻。而这个时间的前进过程是已经计算好的。图9.2中虚线的含义是新的数据会进入分层的队列。

在非阻塞赋值层之后，监视层的系统任务\$strobe被调用。这也是\$strobe能输出非阻塞赋值结果的原因。

9.1.3 简单的分层队列例子

在目录 Lab12 里，有一个叫 InactiveStratum.v 的范例程序。你可以仿真这段代码来理解我们讲的内容。下面是这段代码，和光盘里的代码比起来，这里的代码省略掉了一些注释和空格。

```
'timescale 1ns/100ps
module InactiveStratum;
reg Clk, A, Z, Zin;
always@(posedge Clk)
begin
    A = 1'b1;
    #0 A = 1'b0;
end
`ifdef Casel
// Case 1: Z inactive:
always@(A) #0 Z = Zin; always@(A) Zin = A;
`else
// Case 2: Zin inactive:
always@(A) Z = Zin; always@(A) #0 Zin = A;
`endif
//
initial
begin
#50 Clk = 1'bz;
#50 Clk = 1'b0;
#50 Clk = 1'b1;
#50 $finish;
end
endmodule
```

无论执行哪一个条件编译的分支，都有两个敏感变量为 A 的 always 块。这个 testbench 里只产生了一个时钟的上升沿，并触发了 A 的两个连续的变化。第一个变化是从 0 到 1；第二个，在 0 延迟后，A 从 1 变成 0。

分层事件队列是这样来安排这个仿真顺序的：

1. 时间为 150 时间单位之前，除了时钟，什么都不会发生。在时间为 150 时间单位时，活动区域里的时钟变成 1，产生了一个上升沿。
2. 当时间等于 150 时间单位时，仿真器开始一个新的活动事件，把代码里最靠上的那条赋值语句放入时间等于 150 时间单位时的活动区域。
3. 在计算并执行了赋值这个活动事件后，仿真器把 1 赋给 A。
4. 由于是阻塞赋值，紧接着的那条赋值语句被放在了时间等于 150 时间单位时的非活动区域。
5. 由于 A 变成了 1，将会产生敏感的条件。

下面的分析基于是代码进入 Case 1 分支还是 Case 2 分支。

Case 1

5. 两个 always 块都还没有处于敏感状态。没有增加延迟的 A 给 Zin 的赋值进入了时间等于 150 时间单位时的活动区域。而有 #0 延迟的那部分赋值变成了时间等于 150 时间单位时的非活动区域里的新事件。

6. 只有一个活动事件，因此 A 为 1, Zin 为 1。
7. 没有其他的活动事件，时间等于 150 时间单位时的非活动区域里的两个事件按随机的顺序被移至活动区域。这两个事件为：A = 1'b0 和 Z = Zin。
8. 由于 Zin 已经是 1'b1 了，因此，Z 也为 1'b1；A 变成 1'b0。只是先后顺序是不确定的。
9. A 的变化再次触发了 Case 1 里的 always 块。这使得对 Zin 的赋值被放入活动区域并且被执行；而对 Z 的赋值被放入非活动区域。
10. Zin 的值变成了 1'b0，活动区域现在为空。非活动区域的事件又被放入活动区域，因此，完成了对 Z 的赋值，Z 变成了 1'b0。
11. 所有 always 块里的语句都已经被仿真器读到，因此，没有活动事件了。所有的 always 块再次变得敏感。仿真器会把所有的非阻塞语句不经计算直接放入时间等于 150 时间单位时的活动区域里（这段代码里没有）。在这之后，监视区域里的所有其余事件被放入时间等于 150 时间单位时的活动区域里（这段代码里没有）。
12. 在时间等于 200 时间单位时，仿真工具把 \$finish 放入活动区域，并且执行它，终止当前这个进程。

按 Case 1 分支执行，最后的结果应该为：A = 0, Z = 0, Zin = 0。要注意的是，由于对 Z 的赋值有延迟，当 A 变化时，Zin 的新值保证可以传递给 Z。

Case 2

5. 两个 always 块都还没有处于敏感状态。没有增加延迟的 Zin 给 Z 的赋值进入了时间等于 150 时间单位时的活动区域。而有另一个 always 块里 #0 延迟的那部分赋值变成了时间等于 150 时间单位时的非活动区域里的新事件。
6. 只有一个活动事件。由于 Zin 还没有被赋值，因此它的值是 x；同时，这个值被传递给了 Z。
7. 没有其他的活动事件，时间等于 150 时间单位时的非活动区域里的两个事件按随机的顺序被移至活动区域。这两个事件为：A = 1'b0 和 Zin = A。
8. 我们可以确切地知道 A 会变成 1'b0，但是 Zin 会变成什么，我们还需要仔细地讨论。由于 A 的值不是 1 就是 0，因此，我们可以知道 Zin 不是 1'b1 就是 1'b0，而不会是 x。
9. A 的变化再次触发了 Case 1 里的 always 块。这使得对 Z 的赋值被放入活动区域；而对 Zin 的赋值被放入非活动区域。
10. Z 的值不是 1'b0 就是 1'b1，活动区域现在为空了。非活动区域的事件又被放入活动区域并且被执行，因此，Zin 变成了 1'b0。
11. 所有 always 块里的语句都已经被仿真器读到，因此，没有活动事件了。所有的 always 块再次变得敏感。仿真器会把所有的非阻塞语句不经计算直接放入时间等于 150 时间单位时的活动区域里（这段代码里没有）。在这之后，监视区域里的所有其余事件被放入时间等于 150 时间单位时的活动区域里（这段代码里没有）。
12. 在时间等于 200 时间单位时，仿真工具把 \$finish 放入时间等于 200 时间单位时的活动区域，并且执行它，终止当前这个进程。

按 Case 2 分支执行，最后的结果应该为：A = 0, Z = 0 或 1, Zin = 0。要注意的是，由于对 Zin 的赋值有 #0 的延迟，从而在对 Z 进行赋值时产生了竞争条件。

这样的代码非常复杂而且难以理解，在实际的工程里，我们不应该写出这样的代码。不然的话，除了容易出错（假设这个模块里有更多的 always 块）之外，维护这样的代码也是一件非常困难的事情。并且，在光盘的 Lab12 里的文件的注释里，专门指出了这个代码在两个很常用的仿真工具里会得出不同的仿真结果。

在实际的代码里，不应写出这么混淆的关系。这个例子违背了三个编码规则：

- 在受时钟驱动的块里使用了非阻塞语句。
- 决不要在设计里使用 #0 延迟。
- 不要使用特殊行为的锁存器，例如，这个例子里的两个 always 块。

9.1.4 事件控制

除了在过程块里按代码的先后顺序执行，还有两种事件的控制办法：@ 语句和 wait 语句。而其他的语句只有被执行和不被执行之分。

@ 语句。 @ 语句只允许使用在过程块中。它的作用是仿真器在读下一条表达式的时候需要等一段时间。@ 对边沿也是敏感的。我们已经大量地使用到了它的这个特点，因此就不用再举例了。

@ 的事件表达式可以是对象数据的名字，也可以是某一个确定的边沿（上升沿或下降沿）。当表达式的值从其他的逻辑值变成了逻辑 1 或 0 之后且满足表达式的要求之后，表达式为真，@ 及其之后的语句被读取并被执行。当条件是信号的边沿时，当信号拉高变成 1 (posedge) 或拉低变成 0 (negedge) 时，和刚才一样，@ 及其之后语句被执行。另外，@ 语句本身是不会改变仿真时的任何数据的。

如果把 @ 放在 always 后 (always @)，则可以并行的读取过程块的内容。

对于声明的事件，虽然在实际的工作里很少用到。但为了完整性，本书还是讲一讲这个知识点。声明事件以 Verilog 关键字 event 开头。对于事件对象来说，可以给它指定一个名字。这样，在同一个模块里，可以用这个名字来调用这些对象 (@ statements)。它的语法是 “event MyEventName;”，接下来（可以不紧挨着）是这个事件的具体描述。例如，用事件来判断一个任务：

```
@(MyEventName) do_something;
```

上面的 do_something 可以是任意的描述语句。触发一个事件的格式是单箭头加上事件名，例如：

```
if (expr) -> MyEventName;
```

声明事件的功能就像 C 语言里的 goto 语句，它增加了维护设计的复杂度，设计者应该尽量不使用这样的语法结构。如果需要调用事件，事件的内容应该是一个函数或任务。

wait 语句。 wait 语句是一种对电平敏感的结构。对它起作用的是逻辑表达式，而不是设计对象。它的语法是：

```
wait (expr) statement;
```

当表达式的值为真时，则执行 wait 后的语句。例如，“wait (x>5) x = 0;”。虽然在 VHDL 里常常用到 wait，但是在 Verilog 里，用 wait 的情况却很少。原因应该是事件控制 @ 的功能更强大。

9.1.5 事件队列小结

从上面的内容可看到，最好不要在过程块里或设计的任何一个部分里使用`#n`延迟。如果实在需要，只把`#n`加在模块的输出上。

如果我们采纳了上面的这条建议，则在写可综合的代码时，完全可以不用考虑事件队列的复杂性。只须考虑下面3个问题即可：

1. 在一个特定的仿真时刻里，`begin-end`块里的非阻塞赋值使用的是旧值。直到所有的语句都执行完了之后，新值才会更新进去。
2. 和C语言一样，`begin-end`块里的阻塞赋值总是使用更新后的值。在同一个仿真时刻里，阻塞语句在非阻塞语句之前完成。
3. 因此，一般来说，在受时钟边沿驱动的块里使用非阻塞赋值，在其他的块里使用阻塞赋值。这样的写法可以正确地表述建立时间和顺序行为。综合工具也能很好的理解。

为了理解Verilog这门语言，就必须要理解分层的事件队列。因此，在后面的例子里我们还会经常性地使用到过程延迟。但是请读者记住，在实际的工程里，一般来说，是不应该使用过程延迟的。

9.2 练习 12：计划

在目录Lab12下完成这个练习。

练习步骤

第1步。两个计划与延迟的例子。

```
module SchedDelayA;
reg a, b;
initial
begin
#1 a <= b;
#1 a = 1'b1;
#2 a = 1'b0;
#2 a = 1'b1;
#1 a = 1'b0;
#0 b = 1'b1;
#0 b = 1'b0;
#0 b <= 1'b1;
#5 $finish;
end
//
always@(b) a = b;
//
always@(a) b $$= a;
//
endmodule // SchedDelayA.
```

```
module SchedDelayB;
reg a, b;
initial
begin
#0 b = 1'b1;
#0 b = 1'b0;
#0 b <= 1'b1;
#1 a <= b;
#1 a = 1'b1;
#2 a = 1'b0;
#2 a = 1'b1;
#1 a = 1'b0;
#5 $finish;
end
//
always@(a) b <= a;
//
always@(b) a = b;
//
endmodule // SchedDelayB.
```

在仔细阅读了上面两段代码之后，请读者思考一些问题。你可以通过仿真来得到答案，但是请先思考并先得出自己的答案后再去做仿真。

在 SchedDelayA 里, a 什么时候第一次拉高? b 呢?

在 SchedDelayA 里, a 什么时候最后一次发生变化? b 呢?

对于 SchedDelayB, 请回答同样的问题。

第 2 步。具有上升下降延迟的计划与延迟的例子。

```
module SchedDelayC;
wire a;
reg b, c, d;
initial
begin
#0 b = 1'b0;
c = 1'b1;
d = 1'b1;
#5 d = 1'b0;
#10 c = 1'b1;
#20 $finish;
end
//
assign #(1,3,5) a = c;
assign #(2,3,4) a = d;
//
endmodule // SchedDelayC.
```

A. a 什么时候第一次得到 1 或 0 的逻辑值?

B. 对 b, c 和 d 第一次赋值的顺序是可预见的吗? 如果不能, 为什么?

C. a 的终值是什么?

第 3 步。混合的计划赋值。仿真 Lab12 目录下的 Scheduler.v 文件, 仿真并观察结果。理解调度阻塞赋值, 零延迟赋值, 非阻塞赋值和监视事件先后顺序的区别。图 9.3 是仿真结果。Verilog 代码里的注释解释了为什么是这样的仿真结果。



图 9.3 Scheduler.v 的仿真结果

可选: 接下来, 我们会用 Silos 进行仿真。对于对 Verilog 的编码风格有严格要求的仿真器来说, 可能对阻塞和非阻塞混合赋值的情况支持得并不好。如果工具有问题的话, 就仔细阅读下面的这段代码。这个例子说明了常规延迟的用法。在目录 Lab12 下, 打开文件 BothDelay.v 并仔细阅读。这个设计是基于 9.1 节里常规延迟和计划延迟部分的那个例子而改写的。

试着去推测每一个变化什么时候会发生。如果使用 Silos 来仿真这个模块，你会发现在执行到 initial 块的 end 的时候，结果可能会让你吃惊？能解释原因吗？

最后，假设有这样的一个模块：

```
initial
begin
    Y <= 1'b0;
#0 Y <= 1'b1;
#0 Z <= 1'b1;
    Z <= 1'b0;
$display("display: #04d: Y=%1b Z=%1b", $time, Y, Z);
$strobe("strobe: #04d: Y=%1b Z=%1b", $time, Y, Z);
$finish;
end
```

当这个模块的语句执行完之后，Y 和 Z 的终值是什么？哪个系统函数输出的值是正确的？在初始化阶段，仿真器是把延迟赋值放在非活动区域里还是放在非阻塞延迟区域？这样写代码的目的是什么？到现在为止，我们应该能够理解为什么综合工具对于过程延迟语句，特别是非阻塞赋值里的延迟会拒绝或给出警告。

第4步。事件控制的敏感变量。下面是两个不同的模块。新建名为 EventCtl.v 的文件，在这个文件里的 EventCtl 模块里例化这两个模块。仿真并检查它们的功能。图 9.4 是仿真的结果，仿真后再综合它们。

```
module EventCtlPart(output xPart, yPart, input a, b, c);
reg xReg, yReg;
assign xPart = xReg;
assign yPart = yReg;
always@(a,b)
begin: PartList
    xReg <= a & b & c;
    yReg <= (b | c) ^ a;
end
endmodule // EventCtlPart.
// 
module EventCtlLatch(output xLatch, yLatch, input a, b, c);
reg xReg, yReg;
assign xLatch = xReg;
assign yLatch = yReg;
always@(a)
begin: aLatch
    if (a==1'b1)
        xReg <= b & c;
end
always@(b)
begin: bLatch
    if (b==1'b1)
        yReg <= (b | c)^a;
end
endmodule // EventCtlLatch.
```

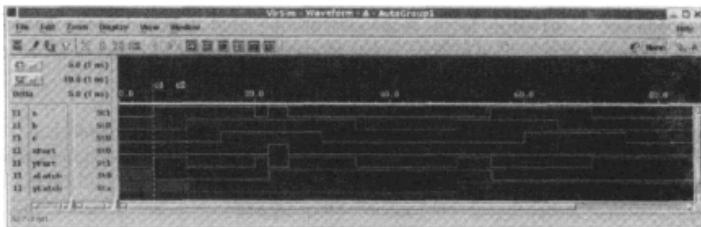


图 9.4 EventCtl 的仿真结果。以 *Part 和 *Latch 结果的信号分别来自两个模块

敏感变量列表里如果漏掉了敏感变量，综合的时候会产生锁存。模块的名字用于在综合里产生的网表里定位具体的信号。

综合工具的综合策略是尽量不产生锁存。如果你的设计的确需要锁存，为了保证的确能产生这个锁存，除了不完整的敏感变量列表外，最好写一个1比特输出的锁存模块。

综合并检查结果，看看是否产生了你需要的锁存。

第5步。上升-下降延迟。上升-下降延迟通常用在网表结构里，例如说门器件或IP(Intellectual Property)。稍做修改，也可以在普通的RTL代码里使用它们。

在一个名为 RiseFall 的模块里，使用 tR, tF 和 tZ 三个常规延迟，描述所有的寄存器行为。下面是具体的要求：

1. 上升延迟是4个时间单位。
 2. 下降延迟是3个时间单位。
 3. 高阻延迟是5个时间单位。

所有的上升-下降延迟都描述的是线或端口的特征，而不是语句本身执行的特征。所以，把它们放到过程执行代码里是没有用的，比如说，把它们放到 always 块里是无效的。

因此，为了给下面的代码里增加上升-下降延迟，需要用连续赋值语句把寄存器的值赋给线型。在给这些线型变量起名时，用同样的名字，只是后面不再有 Reg。

```

reg[3:0] OutBusReg;
reg[7:0] DataBusReg;
reg Out2valReg, Out3valReg;
...
always@{negedge Clk)
begin
OutBusReg  <= 4'bzz01;
DataBusReg <= 8'b1111.0zzz;
Out2valReg <= 1'b1;
Out3valReg <= 1'bz;
end
always@{posedge Clk)
begin
OutBusReg  <= 4'b0101;
DataBusReg <= 8'b1zzz.0000;
Out2valReg <= 1'b0;
Out3valReg <= 1'b0;
end

```

指定延迟并仿真包含上面代码的模块，图 9.5 是一种结果。

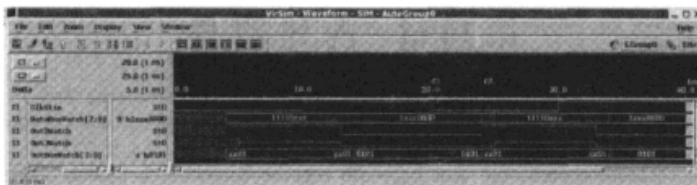


图 9.5 RiseFall 的仿真结果，说明了这些延迟的区别

9.2.1 练习后的思考

Verilog 事件队列应该分更多的层还是更少的层？

思考并解释 BothDelay.v 的结果。

有名块 (named block) 综合之后的结果是什么？

@(*)的作用是什么？

为什么不要在组合逻辑块里使用非阻塞赋值？

如果 1 变成了 z，这算是上升沿还是下降沿？如果 0 变成了 z 呢？

9.2.2 补充练习

阅读 Thomas and Moorby (2002) 的 6.5 节关于上升，下降和三态连接延迟的规范。

阅读 Thomas and Moorby (2002) 第 8 章关于过程和行为事件的计划。

(可选) Thomas and Moorby (2002) 的 4.7 节解释了计划延迟 (intra-assignment) 的用法。理解这个应该尽量避免这样的用法的原因。作者在书中指出，即使它看起来会很有用，但实际上它不会比用常规非阻塞赋值延迟的效果更好。

(可选) 如果你感兴趣，可以在 IEEE 1364 的 11 节里学习更多有关 Verilog 事件队列的知识。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003) 第 7 章。特别关注 7.2.2 节和 7.3 节关于 # 延迟对等号右边的赋值的作用 (intra-assignment delay)。我们应该避免这样的写法，但是我们要理解这个写法的含义。

阅读 5.2.1 节关于上升 - 下降延迟的内容。上升 - 下降延迟主要用在网表里描述时序。

第10章 内建器件

10.1 内建的门及线型

我们将在本章深入学习Verilog网表的知识。Verilog中的网表是由基本的门器件和分层的其他元件（有时是个较复杂的模块）共同组成的结构。

10.1.1 Verilog 内建的门

Thomas and Moorby (2002) 的 6.2.1 节和附录 D 讨论了 IEEE 1364 的 7 节中 Verilog 所有的内建 (built-in) 的原语门 (primitive gate)。我们先把它们都列出来 (开关级的原语暂时不列出来)。可以看到，这些门都不是时序逻辑，虽然有时可以认为三态门保存了最近一个非 z 的逻辑状态，但是我们不把这种情况算做是时序逻辑。

多 输入	多 输出	单输入输出	单 输出
and	buf	bufif1	pullup
nand	not	bufif0	pulldown
or		notif1	
nor		notif0	
xor			
xnor			

在前面的练习里，我们已经用到了 bufif1，而 notifx 仅仅是反相的 bufifx。

在这些门器件的端口列表里，输出端口总是排列在最左边，并且在例化实例的时候可以不需要实例名。Verilog 里的原语门是唯一可以在例化实例时不指定实例名的器件。在前面的练习里，我们看到很多使用这些门的例子，同时它们还被指定了强度。

应该避免使用 pullup 和 pulldown，因为它们是不可综合的。

10.1.2 隐含的线型声明

有的时候，在使用线型 (wire) 时，可以不用明确声明。当一条连接路径连在两个端口之间时，只需要在端口里写上名字就隐含地把这个名字声明成了线型。下面这个完整模块的声明：

```
module NoNets (output Xout, input Ain, Bin);
    and And01(Xout, Ain, Bin);
endmodule
```

下面这个单向传输模块的声明也是完整且正确的。只是在综合的时候，这样的写法会被优化掉：

```
module NoThing (output Xout, input Ain);
    assign Xout = Ain;
endmodule
```

10.1.3 线型和它们的默认值

IEEE 1364 的 19 节里解释了改变隐含线型的默认类型是合法的，由编译向导`default_nettpe 来完成这个工作。默认的类型可以是任何一种类型。在没有指定默认类型的情况下，默认的类型“wire”，这也是我们最常用的类型。在以前的练习里，曾经用过 wor 类型，其实各种类型的用法都是一样的。下面是默认值可以选择的各种类型。

类 型	连线功能
wire	仅作为连接
tri	连接，和 wire 等价，名字不同
tri0	下接到逻辑 0，强度等价（下拉）电阻
tril	上接到逻辑 1，强度等价（上拉）电阻
wand	驱动逻辑等级的逻辑与
triand	逻辑与，与 wand 等价，名字不同
wor	驱动逻辑等级的逻辑或
trior	逻辑或，与 wor 等价，名字不同
trireg	唯一的。当所有驱动都处于高阻（z）逻辑状态下，将（电容）电荷存储在给定的强度等级

默认的类型还可以指定成none，一会儿就会讲到它的用法。Verilog 里的两种线型：supply0 和 supply1，其作用都是上拉。它们不能作为隐含线型的默认类型。我们看到，所有的线型都是门。而 RTL 级和行为级的模型都和 Verilog 的强度表达式无关，和我们马上就要学到的开关级结构也无关。

使用`default_nettpe 带来的好处不大，风险却不小。有可能会出现这种情况：一个模块用到了隐含线型的默认类型，例如说是这个默认类型是 wor。也许某种原因，wor 被其他的部分的代码用到，或 wor 被移到了其他的位置，结果出现在了`default_nettpe wor 之前。这时编译和仿真就会出错，而且往往这些错误看起来是莫名其妙的，不容易查找。

最重要的是，默认的类型可以是 none。在编译器读到了代码`default_nettpe 之后，不再支持隐含的线型声明，所有的线型都必须专门声明。

最安全的办法就是不要用`default_nettpe。如果要用到这个编译向导，那就把默认的类型设成 none。

10.1.4 wire 和 reg

Thomas and Moorby (2002) 的 5.1 节解释了 Verilog 的端口连接规则。设计工程师有时会忽视这些规则，尤其是在写 testbench 而不是写用来实现硬件的代码时。而仿真工具是不会忽视这些规则的。

下面是连接的规则。

- 第一，驱动信号的类型必须是 reg 或任何类型的线型。如图 10.1 所示，输入信号在对应的模块里已经被声明成“reg InReg;”和“wire InWire;”。
- 第二，在一个模块之外，受这个模块输出驱动的变量必须是线型，不能是寄存器。道理很简单：寄存器可以保持数值，也可以用做驱动（第一条规则）。如果把寄存器直接和输出端口相连，驱动的功能和端口输出的功能之间就产生了竞争。图 10.2 说明了这种情况。

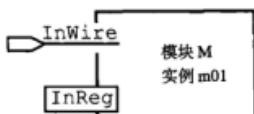


图 10.1 “M m01(…, In1(InWire), .In2(InReg));” 模块

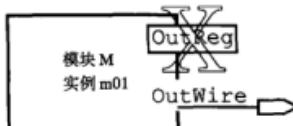


图 10.2 “M m01(…, Out1(OutWire), .Out2(OutReg), …);” 模块

- 第三，模块内被模块输入直接驱动的信号必须是线型。也就是说，模块内的信号和输入端口之间的连线必须是线型。这个规则是必须遵守的，即使有的代码看起来像是寄存器受输入信号直接驱动，其实它们之间还有隐含的线型。
- 第四，如果是双向端口 (inout)，则这个端口的两侧都应该和线型连接在一起。这是因为要同时满足输入和输出的条件，只有线型可以同时满足两边的要求。Thomas and Moorby (2002)指出，inout 端口只能连在三态门上。但实际上，利用强度的概念可以解决内外竞争的问题，所以也可以用一根 wire 来实现连接。

10.1.5 端口和参数的语法

我们已经做了很多关于端口和参数的练习。我们说过，端口和参数的声明格式都很类似。都是 ANSI 格式。在例化时，端口的连接格式也是很类似的。和延迟的值一样，参数声明紧跟着符号 “#”。按 ANSI 格式声明时，很容易区分它们，因为关键字 parameter 总是出现在参数声明之前。例如，下面的例子里就没有用到延迟（虽然有一个参数叫 Delay，但它的含义并不是 Verilog 里的延迟）。

```
module DeviceM
  #(parameter BusWidth=8, Delay=1) // No resemblance to a delay time.
  (output[BusWidth-1:0] OutBus,
   input[BusWidth:1] InA, InB, input Clk, Rst);
  ...

```

下面的这段代码例化了上面的那个模块，参数的传递是按照名字来进行的。显然，这里也没有延迟。

```
DeviceM #( .BusWidth(16), .Delay(5) ) // Default overrides, not delays.
DevM_01 ( .OutBus(Dbus), .InA(ArgA), .InB(ArgB)
          , .Clk(ClockIn), .Rst(Rest)
        );
```

当按位置而不是名字来传递参数的时候，数字的含义就显得比较混淆。假设还是例化上面的那个模块，只是这次按照位置的顺序来传递参数（非 ANSI 格式）。下面的代码看起来就像是在指定延迟。

```
DeviceM #(16, 5) // Default overrides may resemble delays.
DevM_01 ( .OutBus(Dbus), .InA(ArgA), .InB(ArgB)
          , .Clk(ClockIn), .Rst(Rest)
        );
```

我们用一个 Verilog 语言进行例化的例子来做比较。

```
xnor #(16, 5) Adder.U12 ( Z, A, B, C );
```

和延迟一样，最后指定的参数才是最终有效的。作者强烈推荐读者使用按名字传递参数的格式。

10.1.6 用 SR 锁存器组成一个 D 触发器

接下来，我们用三个由 nand 门组成的 SR 锁存器来组成一个 D 触发器 (D flip-flop)。

当与非门 (nand) 的输出为 1 时，至少有一个输入为 0；当输出为 0 时，两个输入都为 1。让我们先来研究输出为 0 的情况。

把 SR 锁存器的 S 和 R 两个输入端口连在一起。

如图 10.3 所示，输出是 q 和 qn。假设 q = 0，则 qn 必然是 1 且 In 也必然是 1。同理，当 In = 1 时，qn 也可以等于 0。因此，当 In = 1 时，输出保持，把这个叫做锁存状态。当 In = 0 时，显然 q 和 qn 都等于 1。

这个电路实际上并没有锁存住任何输入，但是它说明了锁存的行为。

接下来，给这个电路增加一个输入，我们希望这个输入能够被锁存下来，如图 10.4 所示。



图 10.3 单输入的 SR 锁存器

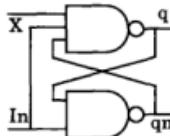


图 10.4 给单输入的 SR 锁存器再增加一个输入

为了锁存数据，必须让 In 和 X 都等于 1。当 X = 1 时，如果 In 是 0，q 和 qn 都为 0；如果 In 变成了 1，输入被锁存。

在锁存的状态下，如果 X 是 0，则 q 为 1，qn 为 0。如果 X 再变回 1，在锁存状态下，q 和 qn 仍然会保持 1 和 0 的值。

在这第二个设计里，如果把 In 连上时钟或异步复位，把 qn 看成要存储的数据，则 X 起的是输入信号的作用。假设在 X = 0 的情况下，让 In 从 1 变成 0 再变成 1：qn 一直都是 0。如果我们把 X 取反，并把 qn 的这个 0 值再放到另外一个 SR 锁存器中保存，电路的结构就和 D 触发器更接近了。

我们再来看一个输入没有连在一起的标准 SR 锁存器，如图 10.5 所示。

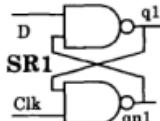


图 10.5 当 D 为 0 时，这个 SR 锁存器的行为和 D 触发器一样

图 10.5 所示触发器的真值表

time	D	Clk	q1	qn1			
0	0	0	1	1			
1	0	1	1	0	qn1	跟随	D
2	1	1	1	0	qn1	锁存	D
3	1	0	0	1	q1	跟随	Clk
4	1	1	0	0	q1	锁存	Clk
5	0	1	1	0	qn1	跟随	D

由上面的真值表，当输入数据为0时，这个锁存器的行为看起来和一个正向边沿触发器的行为一样。

为了使得在输入数据为1时，锁存器也能有同样的行为，我们按图 10.6 那样，把输入数据取反。

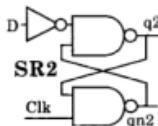


图 10.6 当 D 为 1 时，这个 SR 锁存器的行为和 D 触发器一样

图 10.6 所示触发器的真值表

time	D	!D	Clk	q2	qn2		
0	1	0	0	1	1		
1	1	0	1	1	0	qn2	跟随
2	0	1	1	1	0	qn2	锁存
3	0	1	0	0	1	q2	跟随
4	0	1	1	0	0	q2	锁存
5	1	0	1	1	0	qn2	跟随

由上面的两个真值表综合可知，如果我们要用 SR 锁存器构造出一个 D 触发器：当 D 为 0 时，需要把 qn1 连在 Q 端；当 D 为 1 时，需要把 qn2 连在 Q 端。这两个输出的选通不能通过选通器 mux 来实现，因为 mux 不能保存过去的值。

当时钟为低时，可以用第三级 SR 锁存器来锁住当前的值。要实现这个功能很简单，只须把第三级 SR 锁存器的输入端和有时钟输入的与非门的输出连接起来即可。

如图 10.7 所示，这就是我们设计的 D 触发器。

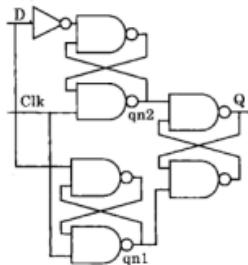


图 10.7 完成后的 D 触发器

图 10.7 所示触发器的真值表

time	D	!D	Clk	qn1	qn2	Q
0	1	0	0	1	1	?L
1	1	0	1	1L	0	1
2	0	1	1	0	0L	1
3	0	1	0	1	1	1L
4	0	1	1	0	1L	0
5	0	1	1	0	1L	0
6	0	1	0	1	1	0L
7	1	1	0	1	1	0L

L为锁存

10.2 练习 13：网表

在目录 Lab13 下完成这个练习。

练习步骤

第1步。门级的D触发器。按照图 10.8 所示设计自己的 D 触发器模块，把它命名为 DFFGates；用 Verilog 里的原语 nand 门来搭这个电路。这个 D 触发器要求被时钟的上升沿触发，被高电平清零。和以前一样，把 DFFGates 模块保存在同名的文件里。

建议：先用 nand 原语例化出多个实例，给它们编好名字，然后在你的网表里利用这些实例名来调用它们（参见图 10.8）。你还可以思考如果不用基本 nand 门，而是用 SR 锁存器来组成这个 D 触发器应该怎么做。本书附带的光盘里自带了用 SR 锁存器组成 D 触发器的电路图的 PDF 文件。

仿真你的 D 触发器设计，确认它是正确的。

第2步。门级同步计数器。找到 Lab08 中的 Synch4DFF，这是一个用行为级的 D 触发器组成的同步计数器的设计。如果还没有完成 Lab08 里的步骤，用光盘中答案里的代码。

把 DFFC.v 和 Synch4DFF.v 复制到目录 Lab13 里头来。打开 Synch4DFF.v 再另存为 Synch4DFFGates.v，并相应地修改模块名。

在文件 Synch4DFFGates.v 里，把实例 DFFC 替换成 DFFGates.v。仿真这个设计，结果如图 10.9 所示。

第3步。综合。综合 Synch4DFF 和 Synch4DFFGates 这两个设计。综合的约束条件除了以下几点，其余要求都是相同的。

先按面积优先来综合，再按速度优先来综合。当按面积优先来综合时，除了面积约束，不要再加任何其他的约束了，把面积参数设成 0。

当按速度优先来综合时，除了指定时钟周期和最大输出延迟的约束，不指定其他的约束。

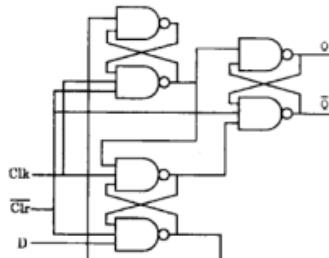


图 10.8 利用与非门实现的 D 触发器

比较按面积和按速度综合出来的面积，体会之间的差别。

可选步骤：用按速度优化综合出来的网表仿真，结果如图 10.10 和图 10.11 所示。

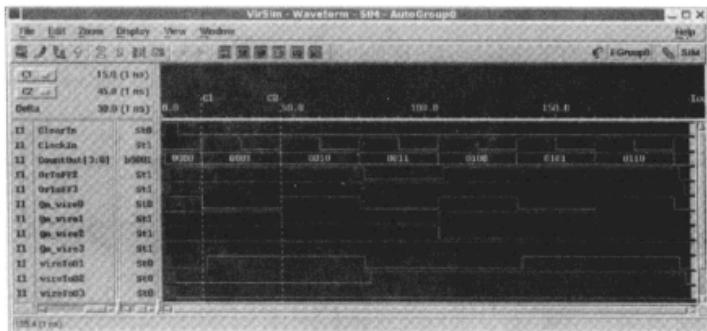


图 10.9 同步计数器的仿真结果

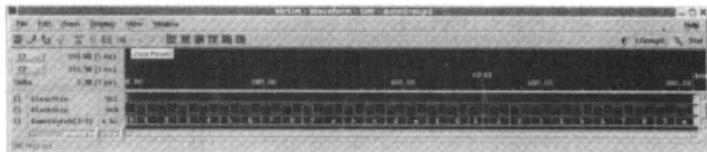


图 10.10 按速度优化的 Synch4DFFGates 的网表的仿真结果

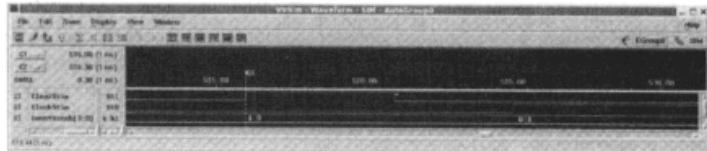


图 10.11 在 Synch4DFFGates 设计里，计数器复位时的波形局部放大图

10.2.1 练习后的思考

思考内建门和强度之间的关系。

对一个结构化的设计来说，逻辑优化和库映射之间的关系是什么？

当综合一个行为级和门级的设计时，比如练习里的第 3 步，综合工具在进行映射时，哪一种描述风格会更符合综合工具的需求。为什么？

10.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 6.2.1 节至 6.2.3 节里关于线型和原语的内容。

阅读 Thomas and Moorby (2002) 第 10 章关于强度和开关级建模的内容。

阅读 Thomas and Moorby (2002) 的 5.1 节关于端口连接规则的内容。

阅读 Palnitkar (2003) (可选)

4.2.3 节有关于 Verilog 端口连接规则的内容。

阅读 10.1 节关于在网表中添加门延迟的内容。如果你感兴趣，仔细阅读整个第 10 章。在本书后面的章节里，我们还将学习有关元件内部延迟的知识。

完成 5.4 节里的练习。

仿真 5.1.4 节里的 Verilog 范例代码。Palnitkar 的光盘里就有这些代码，因此读者不用自己再去写代码了。如果你在代码里加入了 `timescale，仿真的结果会更真实一些。

6.5 节里有 5.1.4 节讲到的器件的过程化模型，比较它们之间的区别。Palnitkar 还在 6.5.3 节的图 6.4 中给出了一个 D 触发器的门级电路图。



第11章 顺序控制和并发

11.1 顺序控制和并发

我们会在本章复习并继续深入学习 Verilog 顺序控制语句的知识。我们也会学习到并发的任务控制和有名块 (named block)。

11.1.1 Verilog 顺序控制语句

for 是 Verilog 里最常用的循环控制语句，综合工具对它的支持比其他的一些语句要好一些。为了完整性，我们也会介绍到其他的语句。随着本书课程的深入，在你的设计中，会经常用到 for 语句。

在前面的练习中，我们多次用到了 if 语句。我们会接着学习 case 语句。除此之外，还有 forever, repeat 和 while 语句。

forever

它的用法是在 forever 后紧接着表达式或语句块。

一旦 forever 开始工作之后就不会停下来。不过我们可以利用其他的 Verilog 语句来强制停下它。forever 几乎总是用在 initial 块中。通常，当仿真工具执行到 \$stop 和 \$finish 时，会终止 forever 循环。

```
initial
begin : Clock.gen
Clock1 = 1'b0;
forever #10 Clock1 = ~Clock1;
$finish;
end
...
initial
begin : Test.vectors
Abus = 32'h0101_1010;
Dbus = 'b0;
#5 Reset = 1'b0;
...
#220 $finish; // Terminate the simulation after about 10 clocks.
end
```

并发的时钟产生器，如 “always @(Clock1) #10 Clock1 <= ~Clock1;” 和上面的 forever 语句是不等效的。主要的区别是：用 forever 产生时钟的办法使用的是阻塞赋值，除非在其余相关的组合逻辑中专门增加延时，否则可能无法保证电路的建立时间。

另外一个区别是对 forever 来说，每 10 个时间单位，Clock1 就会翻转一次，和其余的条件无关。而对 always 的这个逻辑来说，只有当前仿真事件都完成了之后，Clock1 的值才会发生改变。因此，always 不能用阻塞的方式来产生时钟。

而且，always 是一种并发的结构，不是顺序执行。在顺序执行块之中不能包含并发的 always 逻辑。

forever 的这个用法是不可综合的，因为没有对应于这个 initial 块的硬件电路。当然，上面的那个 always 写法也是不可综合的。

上面那个 Clock_gen 里的 \$finish 实际上永远也不会发生，因为 forever 执行完了之后才会轮到它。

下面是两种终止 forever 的办法。

```
// Technique 1:
...
begin
  forever
    begin
      #1 Count = Count + 1; // Include a delay or @!
      if (Count>=1000) $finish;
    end
  end
```

```
// Technique 2:
begin: Mod16.Counter
  forever
    begin
      Count = Count + 1;           // Count is an integer or wide reg.
      #5 Dbus[4:0] = Count%16;   // This delay avoids hanging the simulator.
    end
  end // Mod16.Counter.
// Somewhere else, or in the loop, put disable Mod16.Counter.
```

如果我们要产生一个可以关断并重新运行的时钟，可以这样写：

```
always@(posedge Run)
begin : RunClock1
  Clock1 = 1'b0;
  forever #10 Clock1 <= !Clock1;
end
always@(negedge Run) disable RunClock1;
```

在产生时钟的时候，记得把建立和保持时间的需求也考虑进去。

repeat。它的用法是：repeat (number_of_times)，然后紧接着其他的语句或块。repeat 是一种严格的循环计数结构。当仿真工具发现 repeat 语句时，把 number_of_times 存下来，然后运行指定的次数。和 for 迭代不同，number_of_times 的值不能在循环过程中改变。可以用前面提前终止 forever 的方法提前终止 repeat 循环。

下面是使用 repeat 的一个例子：

```
...
i = 0;
repeat(32)
begin
  #2 Abus[i] = LocalAbus[i+32] + AdrOffset;
  i = i + 1;
end
```

我们看到，repeat 的用法和 for 的比较类似。而在 repeat 中用 disable 来终止循环的办法可能会导致不易检查出来的错误。如果一个块就有数百行，找到类似的控制命令会很困难，而作者就曾设计过超过 1000 行的状态机。总体来说，作者推荐使用 for 而不是 repeat。除非不需要控制循环的状态，才可以使用 repeat。

while。它的用法是：while（表达式），然后紧接着其他的语句或块。在工具第一次读到 while 的时候，工具会判断表达式的值，然后每一次执行完整段语句时，再重新判断表达式的值，直到表达式的值为假为止。只要 while 的表达式不为 0，while 就会执行后面的语句。如果表达式为 0，则退出 while 循环接着执行下面的语句。

while 的表达式使得 while 比 repeat 可以应用在更多的场合。而在 while 和 for 中，都可以加入时间延迟，因此，在很多情况下，while 可以和 for 互换。下面是其例子：

```
// Example 1:
...
SleepState = 1'b1;
while(SleepState==1'b1)
begin
    slowRefresh; // A task.
    #SleepDelay Status = CheckUserInput; // A function call.
    if (Status!=Asleep) SleepState = 1'b0; // Exit the while.
end
// Example 2:
Count = 10;
while(Count > 0)
begin
    ...
    (do stuff)
    #3 BusA[Count] = BusB[Count];
    Count = Count - 1;
end
```

下面用 for 写出来的语句和上面用 while 写出来的语句等效。

```
// Example 1:
...
for (SleepState = 1'b1; SleepState==1'b1;
     SleepState = (Status!=Asleep)? 1'b0: 1'b1
   )
begin
    slowRefresh;
    #SleepDelay Status = CheckUserInput;
end
// Example 2:
...
for(Count=10; Count>0; Count=Count-1)
begin
    ...
    (do stuff)
    #3 BusA[Count] = BusB[Count];
end
```

while 的表达式不处理循环条件的更新，这个更新的过程发生在其他的语句中。这使得 while 在处理控制信号的时候，可以增加延迟信息。这对 while 来说是一个独特的优点。

对 for 来说，for 表达式中是不允许加入延迟信息的。因此，为了加入延迟信息，有时需要想办法把控制 for 循环的条件拿到表达式外面去。

for循环中，通常所有的控制都是在前面，即语句块的头部；这让大家更易于理解控制和调试错误。基于这个原因，推荐大家在从道理上可能时，从循环控制中优先选择for语句。

11.1.2 case

语句if主要优点是支持关系表达式，所有的条件都可以放到一个表达式里头去。与此对应的语句是case，一次只能指定一个确定的值。虽然多个条件之间可以用逗号隔开，但是仍然是需要明确指定的。不过，case条件列出来之后像一个表，可读性会比一连串的if…else链好一些。除此之外，if和case的主要区别在于它们处理“x”和“z”是不一样的。

在if判断两个包含“x”和“z”的逻辑状态是否相等时，把“x”认为是1或0，并不是一个确定的值。因此，对于一个包含了“x”和“z”的变量X，即使对于“if(X==X)”这样的表达式也是认为不成立的，而会执行else分支。

而case却不是这么处理的，case对指定的比特类型进行匹配，无论其中是否含有“x”或“z”。如果case表达式中的向量含有“x”或“z”的某种逻辑状态，并且其中一个可选项包含相同类型，case语句认为找到匹配，并根据该匹配选项选择性地执行语句。

如下例所示：

```

X = 4'b101x;
Y = 4'b101z;
//
if (X==4'b101x) ...; // Evaluates as 'x'; won't match the value above.
if (X==Y) ...;       // Won't match; and if (X!=Y) won't, either.
if (X==X) ...;       // This won't match, either!
//
case (X)
  4'b1010: ...
  4'b1011: ...
  4'b101z: ...
  4'bxxxx,
  4'bzzzz: ...; // Comma separation is legal.
  default: ...; // This will execute because nothing else did.
endcase
case (X)
  4'b1010: ...
  4'b101x: ...; // This matches and will execute.
  4'b101z: ...
  4'bxxxx,
  4'bzzzz: ...
  default: ...
endcase

```

另外一点需要专门提醒的是关于条件操作符“?:” 的用法。有时可以用它替代if语句。对于1比特的逻辑运算来说，它们就是等效的。对于多比特的条件操作来说，除非位宽匹配，否则当“x”或“z”出现在条件中时，这使得表达式返回值的每一位都变为“x”。

为了让if能够进行完全比较，Verilog提供了全等符(case equality operator), “==”(不全等是“!=”)。在前面的内容里，我们简要提到过它们。这个操作符的比较结果不会出现“x”。它们把“x”和“z”看成“1”和“0”一样来看待。而且和用到的语句无关，无论在if还是在case里，效果都一样。

```

X <= 4'b101x;
Y <= 4'b101z;
//
if (X==4'b101x) ... else ...; // expr = 1; the if executes.
if (X!=Y) ... else ...; // expr = 1; the if executes.
if (X!=Y) ... else ...; // expr = x; the else executes.
//
// The conditional operator is not an if but accepts case equality:
Z = (X==Y)? 1'b0; // Assigns 1'bx.
Z = (X==Y)? 1'b1: 1'b0; // Assigns 1'b0.
Z = (X==Y)? 1'b1: 1'b0; // Assigns 1'b1.

```

除了 case, Verilog 里还有 casex 和 casez 两种语句。

casex, casex 把 “x” 和 “z” 都看成通配符 (wildcard)。也就是说，任何状态都和 “x” 和 “z” 匹配，即使是 “x” 和 “z” 自己。和 case一样，casex 仍然以 endcase 结束。由于 “z” 在这里的作用是通配符，因此用 “z” 来表示任意一个值会导致意义上的混乱。为了含义清楚，这种情况应该用 “?”，而不是任意用 “x” 或 “z”。

```

X <= 4'b101x; Y <= 4'b101z; // X and Y are reg[3:0].
// Example 1:
casex (X)
  4'b100z: ...; // No match.
  4'b10xx: ...; // Executes, because it matches and comes first.
  4'b11xz: ...; // Can't execute on this value of X.
  4'bxxxx: ...; // Would execute if 4'b10xx didn't.
  default: ...; // Would execute if nothing else did.
endcase
// Example 2: Some sort of bit-mask or decoder:
casex (Y)
  4'b????: ...; // Executes, because of casex LSB wildcard 'z' in Y above!
  4'b??1?: ...; // Would execute if the first one didn't.
  4'b?1???: ...; // Can't execute on this value of Y.
  4'b!???: ...; // Would execute if nothing above did.
  default: ...; // Would execute if no '1' or wildcard in Y.
endcase

```

从上面的例子，能够发现 casex 的一个好处。我们能够用较少的条件来表达大量的组合。在上面的例子里，如果不用 casex，可能会用一个很长的 if...else 链，或者用 case 把 4 比特对象的 16 种 “0” 和 “1”的组合中的 12 种相互独立的无效情况都写出来。

然而，用 casex 会很危险而且容易出错。上面的例子里，设计者的意图是只检查 Y 中 4 个比特位的 1。但如果在仿真时，对 Y 的赋值发生了错误，或 Y 中出现了高阻，casex 很有可能在没有出现 1 的时候发生状态的变化，从而违背了设计意图。如果用 if 语句会好一些，可以这样写：

```

Y <= 4'b101z; // Y is reg[3:0].
//
if      (Y[0]==1'b1) ...; // 'z' means no match (but == 1'bx would match).
else if (Y[1]==1'b1) ...; // This one executes.
else if (Y[2]==1'b1) ...; // No, not '1' and below the first match.
else if (Y[3]==1'b1) ...; // No, below the one that first matches.
else     ...;           // Executes if no '1' in Y.

```

如上面的代码，虽然每一行我们都需要输入不同的条件，但这种写法使得代码有良好的可读性。但是，如果 Y 被声明成 reg [0:3] 而不是 reg [3:0]，还是有可能会出错。

如果考虑到综合的因素，应该更加慎重地考虑是否要使用 casex。综合工具会按照自己工作最优的条件去优化 casex 里不关心的状态。因为设计者并不关心信号是否处于那些状态，综合工具便可以“为所欲为”。因此，综合工具往往会忽略通配符的匹配规则。从而综合出的网表和 Verilog 源代码在功能上会有所差异。对于上一个 casex 的例子来说，default 条件里的内容也可能会被合并到条件 4'bxxxx 里面去了。而设计者可能会忽略这一点。

因为 casex 的通配符可能会导致这样那样的问题，因此，不推荐在任何地方使用 casex。如果只需要选择一小部分数据，或遇到必须要使用通配符的情况，可以用 casez 来替换 case 或 if。对于 casez 来说，最起码不把“x”也看成通配符，综合工具处理这些不关心的状态时也会更容易。

casez 和 casex 不同，casez 只把“z”看做是通配符。除了“z”，“?”也可以看成是通配符。

```
X <= 4'b1x00; // X is reg[3:0].
// Example 1: No match for any of the alternatives:
casez (X)
  4'b100z: ...;
  4'b10xx: ...;
  4'b10xz: ...;
  4'bxxxx: ...;
  4'b0zzz: ...;
default: ...; // Executes.
endcase
// Example 2: '?' is same as 'z'
casez (X)
  4'b????: ...; // No match.
  4'b??1?: ...; // No match.
  4'b?1???: ...; // No match.
  4'b1???: ...; // Executes; X[3] matches 1==1, and others are wild.
default: ...; // Can execute on X == 4'b0000, or on anything
             // with no '1' or 'z' anywhere, etc.
```

我们还是应该尽量避免使用 casez。如果想使用 casez，首先应该先考虑能不能用 case 或 if 来代替。如果的确有需要，还是可以考虑使用 casez。而 casex 是一定不能用的，因为 casex 很有可能会导致综合出来的网表和原始的设计不一致。

11.1.3 并行过程

下面我们会经常地使用到线程这个名词。虽然软件的并行操作（parallelism）也有基于进程的线程概念，但和这里的线程是有区别的。

我们已经学过了 fork-join 并行块，并且在练习中也用到了。我们还应该把并行处理的方式扩展到所有的仿真线程上去。最简单的方式是利用并行任务。

在每一个任务里实现多个并行线程有许多优点：

- (a) 线程实现简单，因为它只能包含任务中的语句。
- (b) 过程块中的并行任务使得在做设计时调试起来很方便，能够很容易地知道一个事件什么时候开始，什么时候结束；但是这和多个 always 块并行执行是不同的。

(c) 块的名字（任务名）通常会在综合后被保存在网表里；不然的话，设计者应该考虑给 always 块命名或用其他办法替换这个任务。

(d) 任务中的语句都是任务内部里使用的，因此修改任务内部代码的可以不影响到其他的逻辑。如果这些代码不在任务中，那么有可能在设计需要小改动时，很多处代码都需要修改。

让我们来看一个用 always 块实现的并行结构。这段代码的作用是：检查 4 比特数据总线是否发生了翻转，并通过 2 比特的总线输出哪个比特发生了翻转。需要用并行结构的原因是需要同时检查 4 个输入数据线上的变化。如果在过程块中做这个检查的话，则意味着不同比特之间存在优先级，检查的顺序被人为分了先后。

```
...
task CheckToggle(input[1:0] BitNo);
begin
  #1 ToggledReg = BitNo;
end
endtask
//
always@(posedge CheckInBus) CheckToggle(0);
always@(posedge CheckInBus) CheckToggle(1);
always@(posedge CheckInBus) CheckToggle(2);
always@(posedge CheckInBus) CheckToggle(3);
...
...
```

可以把上面多个 always 块的写法换成一个包含 fork-join 的 always 块。

```
...
always@(posedge CheckInBus)
begin
  fork
    CheckToggle(0);
    CheckToggle(1);
    CheckToggle(2);
    CheckToggle(3);
  join
end
...
...
```

严格来说，这两个表达方式是不等效的。即使 4 个比特都发生了翻转，fork-join 块也不会退出。这意味着这 4 个并行的任务会一直工作，直到仿真结束。假设输入的某 1 比特一次翻转都没有发生，则其他比特的翻转会触发对应的任务，并且不停地加到仿真事情队列中去，直到 4 个比特都发生了翻转，才会退出 4 个比特的并行检查。这样会造成软件的内存泄漏，有可能导致仿真失败或仿真的速度变得非常慢。

这里要得出的结论是：当使用 fork-join 的时候，一定要保证 join 会被执行。

当阻塞赋值和非阻塞赋值混用的情况，fork-join 的作用体现得很明显。图 11.1 就是这样的一个例子。当然，在实际的设计中，混用阻塞赋值和非阻塞赋值是非常不好的习惯。

在 Verilog 里还有另外一种实现并行处理的方法，那就是把需要并行执行的功能分别放在不同的模块或实例中。和电路板上不同的芯片一样，不同的模块总是并行执行的。

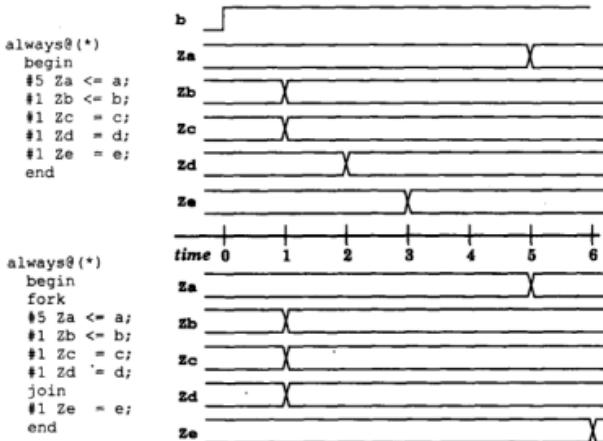


图 11.1 fork-join 改变了事件的先后顺序

11.1.4 Verilog 命名有效的范围

到目前为止所有的代码里，我们都是先声明一个变量，再使用它。这是基于我们曾经使用过 C 语言或者其他编程语言的经验。和 C 语言不同，Verilog 没有全局变量概念，变量必须在模块内被声明。因此，一个 Verilog 程序的基本结构是这样的：

```

module module_name ( I/O's );
  reg name declarations; ...
  wire name declarations; ...
  (programming stuff, using previously declared (or implied) names in statements)
endmodule

```

对于其他的编程语言，变量的声明必须摆在代码的最前面。先声明（避免命名冲突），再使用。而在 Verilog 里却没有这个限制，只要在用到这个变量之前声明它就可以了。此外，任务和函数也可以在模块里的任何位置声明。

Verilog 的变量名可以在模块内被声明，也可以在任务和函数中声明，甚至在 always 块中也可以声明变量。对于编译器来说，这些名字只在本地块中有效，不会和块外同样的名字发生冲突。

Verilog 规定，只要在有名的区域里，就可以声明变量。因此，在模块的任何一个地方声明一个新的 reg 型变量或 wire 型变量都是可以的，甚至在 always 块和 initial 块之间声明新变量也是允许的。从声明的地方，到这个文件的结尾，声明都有效。

在使用这个特性时，应该谨慎。当然，这个特性也是有好处的，我们可以紧挨着 always 块声明这个块要用到的变量，这可能在文件的开头，离 always 很远的地方来声明这些变量要好。这个特点决定了 Verilog 比 C 语言更加地面向对象。因为不同对象的逻辑可以做得更集中。如下例所示：

```

module ...
... (500 lines of verilog) ...
reg[7:0] ClockCount;
always@(negedge ClockIn)
begin : Ticker
  if (StartCount==1'b1)
    ClockCount = 'b0;
  else ClockCount = ClockCount + 8'h1;
  if (ClockCount >= 8'h3a) (do something);
end

```

综合工具可能会不支持上面的写法，因为在 always 块里对同一个变量有多个同一优先级的 if 语句。

要注意的一种情况是下面的这种写法，这个计数器的声明被放在了 always 块的内部，仿真工具可能不会保存当前计数器的值。因为在每一个 ClockIn 的下降沿，这个寄存器都要被重新声明一次，而它的初始值是 4'bzzzz。

```

always@(negedge ClockIn)
begin : Ticker
  reg[7:0] ClockCount; // ERROR! Redeclared every time always is read!
  if (StartCount==1'b1)
    ClockCount = 'b0;
  else ClockCount = ClockCount + 8'h1;
  ...
end

```

而对于在任务内部声明的变量来说，它们是静态的。不管这个任务被调用了多少次，变量的数值总是会被保留下来的。如果多个并行的实例同时调用了同一个任务，这个变量的值还是不会被改变，因此这些实例实际上是共享了这个变量。只有一种任务是例外的，那就是 automatic 类型的任务，声明这类任务的写法是：task automatic Mytask。每一个实例调用的 automatic 任务都有自己的一份数据。由于 automatic 类型有这种特性，因此可以用它来进行递归运算。

上面的 CheckToggle 这个例子，实际上它不会像我们希望的那样去工作。因为输入 BitNo 会被隐含地声明成 reg，并且只声明一次。而所有的 CheckToggle 都共享同一个输入值。

所以，所有的 CheckToggle 都会检查 InBus 的同一个比特。这和把这个任务写成没有 I/O 的任务，且把 reg[1:0] BitNo 声明在任务之外是一个效果。任务内部的变量和传递给任务的变量（任务的 I/O）都是静态变量，对这个任务的所有调度都共享一个数值。有时，这种数据的共享性质正是设计者需要的，这给不同实例之间的通信提供了手段。

如果有的任务会在一个仿真周期里被调用一次以上，而且设计者希望每一次调用都是独立的，不会共享数据。那么，可以用关键字 automatic 来声明一个任务：

```

...
task automatic CheckToggle(input[1:0] BitNo);
  begin
    @(InBus[BitNo]) #1 ToggledReg = BitNo;
  end
endtask
...

```

除了任务，函数也可以被声明成 automatic 类型来实现递归的操作。下面是一个递归函数的例子：

```
...
function automatic[31:0] Factorial(input[3:0] N);
begin
    if ( N>1 )
        Factorial = N * Factorial(N-1);
    else Factorial = 1;
end
endfunction
...
```

注意观察上例中定义函数位宽的方式，输入被隐含地当做 reg 来处理。当外部调用 Factorial 这个函数时，函数只有在当 N 等于 1 的时候才会返回，返回的结果是 $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$ 。递归的函数和任务是不可综合的，因此，在进行可综合的设计时，应该避免使用递归。

11.2 练习 14：并行

在目录 Lab14 下完成这个练习。

练习步骤

第 1 步。forever 块。用 forever 块写一个产生时钟的 testbench，并完成功能仿真。

第 2 步。repeat 块。在完成第 1 步的基础上，增加一个受时钟驱动且包含 repeat 块的 always 块，这个 repeat 的作用是以 0, 1 交替的形式初始化一个 32 比特的总线（…010101…），一次只初始化一个比特，不是直接对总线进行赋值。完成后仿真。

第 3 步。while 块。在第 2 步的基础上，新增加一个包含 while 语句的 always 块。这个 while 的作用是检查这个总线某一个特定的输出值，即 1 和 0 的组合（具体是什么值由自己决定），一次只检查一个比特。当发现了这个值之后，利用 while 输出一个标志信号。不要在 while 里使用 disable 语句。完成后仿真。

第 4 步。case 语句。用 case 实现一个编码器，它的输入为 8 比特（一次只有一个比特为 1），输出为 3 比特。输出的作用是找到第几个比特的输入是 1（从 0 开始）。在完成了这一步之后，再把这个输出翻译成 ASCII 的格式（‘0’ = 8'h30, ‘1’ = 8'h31, …）。完成后仿真。

第 5 步。并行练习。要理解这个概念会更加困难一些。做这个练习的时候不用考虑到设计的可综合性。

完成一个名为 CPU_Board 的顶层模块，它包含 CPU 和 WatchDog 两个 always 块。按照下面的叙述来完成两个 always 块。这里的目的仅仅是做练习，通常，CPU 和 WatchDog 是被分放在两个模块中的（参见图 11.2）。

模块 CPU_Board 有一套 32 比特的输出总线 Abus，一套双向总线 Dbus，输入信号为 Halt 和 Clk。此外，它还有三个 1 比特的输出：RecoveryMode, INT00, INT00_Ack。模块中的两个 always 块使用同一个时钟的不同边沿。

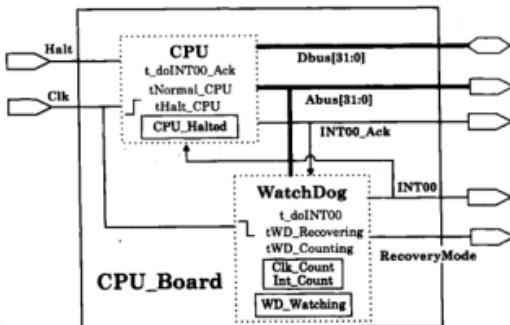


图 11.2 CPU 和 WatchDog 的设计。虚线表示逻辑块而不是设计层次

在这个练习里，我们并不关心 CPU 块的功能。因此，用 \$random 来产生随机的 CPU 总线数据。如下所示：

```
...
always@(posedge Clk)
begin : CPU
  ...
#1 Dbus = $random($time); // $time repeats only between simulations;
#1 Abus = $random($time); // so should the (32-bit) random patterns.
...
end
```

WatchDog 的作用非常简单。它并不完成系统本身的功能，它被用来监视一个复杂设备的状态，如果状态发生了异常，可以利用 WatchDog 把这个设备从异常状态中恢复过来。

通常，WatchDog 对当某种状态或行为进行计数，当计数值达到了门限但状态或行为还未改变，则中断当前操作或自动复位整个系统。通过这种方式，可以避免在高可靠的软硬件里发生死锁（deadlock）。并行系统的一个缺点就是不同的器件之间可能会一直相互等对方的释放某种资源从而发生死锁。

A. WatchDog 的 always 块。这部分逻辑完成以下 4 个功能：

- 每当 CPU 的地址发生变化之后，从 0 开始对时钟进行计数。如果地址发生了变化，计数器清 0。
- 当计数值超过了 10，则输出 INT00 脉冲，试图中断 CPU 当前的操作。INT00 脉冲会被周期性地持续发出，直到收到了 CPU 的应答信号为止。
- 在输出 INT00 的同时，还应拉高 RecoveryMode 信号，通知其他的外围设备系统发生了异常。
- 当收到了 CPU 的应答信号 INT00_Ack 之后，WatchDog 拉低 RecoveryMode 输出，停止产生 INT00 脉冲，重新开始计数。

B. CPU 的 always 块。这部分逻辑完成以下 4 个功能：

- 如上面提到的一样，产生 Dbus 和 Abus 上的数据。
- 当收到外部的 Halt 信号时（由 testbench 产生），终止执行当前的命令。

(c) 在没有收到 Halt 信号时，相应的 INT00 中断。

(d) 根据当前的状态决定是发出 INT00_Ack 响应信号还是暂时不响应，让 INT00 中断接着发送。

用两个 always 块和至少一个 Verilog 的任务在模块 CPU_Board 里实现以上要求。

建议先写完一个 always 块的代码之后再去写第二个。CPU 这部分的逻辑要简单一些。因此，可以先完成这部分，图 11.3 是仿真的结果。



图 11.3 CPU_Board 设计的仿真波形

在这个实验里，我们产生中断的计数值是 10。而在实际的嵌入式 CPU 里，中断计数通常 是 5 个时钟或 10 个时钟。

11.2.1 练习后的思考

用 casex 有什么不好？

怎么样用状态机来实现这个 CPU_Board？需要几个状态机？

11.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 3.1 节 ~3.4.4 节关于过程控制结构的内容。

注意以下几点：repeat 语句是可综合的 Verilog 语法。绝对不要用 casex；casez 也要少用，除非实在有必要才用 casez。

复习 Thomas and Moorby (2002) 的 4.9 节关于 fork-join 的内容。

(选读) Mike Turpin 关于综合 casex 和 casez 的危险性的一篇文章：“The Dangers of Living with an X(bugs hidden in your Verilog)”，网址是 http://www.arm.com/pdfs/Verilog_X_Bugs.pdf。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003) 一书的 7.5~7.7 节里讲到的内容。

第 12 章 层次和 generate

12.1 层次命名和 generate 块

本章我们将会介绍几种常用的描述硬件的语言结构。

12.1.1 层次命名的有效范围

前面的章节曾经提到过，在 Verilog 里，可以在设计中的任何一个位置操作另外一个位置的信号。这套方法和我们所熟悉的基于路径的文件系统很相似。当指定一个文件时，可以从根目录（对应于顶层设计）一级指定它的位置，也可以用当前的目录（对应于当前的模块）来描述文件所处的位置。在 UNIX 或 Linux 系统里，名字的分隔符是 “/”；而在 Verilog 里，分隔符是 “.”。

例如图 12.1 中的电路，A 是顶层模块，B、C 和 D 是 A 中的实例。在 D 里还例化了 E。A 是模块名，其余的都是实例名。实例的类型各有不同，这是因为它们被不同的模块，如 Dtype 等例化。

这种组织结构被叫做层次，即每一个模块只被另外一个模块所包含。这个概念就好像是楼层里的房间，示意图如图 12.1 所示。

一种用来表示这种机制的办法是“家族树”（family tree），从一个根，向外向下扩展。如果是设计树，每一个树上的单元只有一个上级单元，如图 12.2 所示。

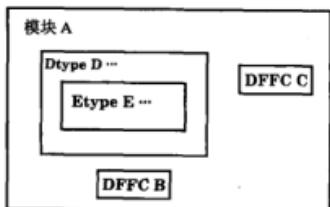


图 12.1 模块 A 及其对应的层次结构

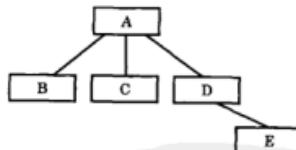


图 12.2 模块 A 及其对应的层次树

Verilog 中的命名是按层次来区分的，命名的有效性并不只对模块有效，对声明的任务，fork-join 块，变量等都同样适用。任何以 begin 开头的过程都可以被命名。

在 Verilog 中，可以在任何一级通过指明全路径实现对一个名字的引用。虽然可以这么用，但是这并不意味着我们推荐使用这种方式。

例如，参考图 12.1 和图 12.2，模块 DFFC 是一个寄存器。因此它例化的 C 也是一个寄存器。因此，在 A 中，模块 DFFC 实例化 C，如果用层次命名来表示它的输出 Q，则应该是 C.Q。例如，可以把 C.Q 送给 B.D 的输入端口：

```
assign B.D = C.Q;
```

需要注意的是，由于上面的写法是用相对路径来指定对应的端口，因此，只能在 A 中这么做。A 知道其下两个寄存器分别例化的实例名，因此，在引用端口的时候不用指定全路径。

当使用层次引用时，作为参考的那一层（当前的顶层）必须用它的模块名而不是实例名来作为参考。

还是讨论这个 D 输入端（如果模块 D_Type 里有的话），按照这个原则，上面的赋值可以写成：

```
assign A.B.D = A.C.Q;
```

在模块 D_Type 例化的实例 D 中这样赋值：assign B.D = C.Q 是非法的，因为实例 B 和实例 C 并没有例化在 D 中。

让我们再举一个例子。假设模块 E_Type 有一个输出端口连在名为 OutBus 的线上，模块 DFFC 中有名为 OutBus 的线。如果想把 OutWire 和这个端口相连，在模块 DFFC 里，可以这么写：

```
assign A.D.E.OutBus = OutWire;
```

这句话也可以这么写：

```
assign D.E.OutBus = C.OutWire;
```

由上可知，层次命名是一个复杂且危险的特性，我们应该严格限制使用它，因为它可能会导致设计错误。通常，它们只被用来引用当前模块之内的信号。层次命名的参考规则就像软件里臭名昭著的 goto 语句，用处不大而且容易产生 bug。

不建议在设计实际电路时使用层次引用；一个好的设计应该通过模块的端口来实现互联，而不是通过跨越命名空间的层次引用。然而，在设计 textbench 或断言时，这种层次引用是被允许且有用的，因为在 textbench 和断言中使用层次引用并不会破坏设计的层次性。

如果我们想参考模块中的子结构，但并不推荐使用层次命名。下面，我们会学到利用 generate 结构可以很方便地建立由上至下的层次。

12.1.2 Verilog 中的数组实例

Verilog 里有一个很有趣的特性是可以把实例也组合成数组。这和前面学到的内存数组的概念比较接近。数组内部元素的位宽由枚举的范围而不是本身的位宽来确定的。例如：

```
and #(3,5) InputGater[2:10](InBus, Dbus1, Dbus2, Dbus3);
```

上面的写法里，InBus 等 4 个总线上同样序号的比特被 and 引用。这和下面的写法是等效的：

```
and #(3,5) InputGater[2] (InBus[2], Dbus1[2], Dbus2[2], Dbus3[2]);
...
and #(3,5) InputGater[10](InBus[10], Dbus1[10], Dbus2[10], Dbus3[10]);
```

我们看到，方括号中是数组和端口的序号。

数组实例中的实例可以是用户定义的原语或模块。数组的范围可以使用参数。也可以使用连接（concatenation）表达式。例如：

```
MyBuffer Xbuff[1:3](.OutPin(OutBus), .InPin({InBus[5], InBus[3], InBus[7]}));
```

下面的写法和上面的写法是等效的：

```
MyBuffer Xbuff[1] (.OutPin(OutBus[1]), .InPin(InBus[5]));
MyBuffer Xbuff[2] (.OutPin(OutBus[2]), .InPin(InBus[3]));
MyBuffer Xbuff[3] (.OutPin(OutBus[3]), .InPin(InBus[7]));
```

注意：Silos 仿真工具（演示版）不支持编译用户自定义的数组实例。

下面，我们将学习可以灵活描述总线对象的 generate 语句。当然，如果是简单的总线结构，用数组来描述也是没有问题的。

12.1.3 generate 语句

generate 是调用过程控制语句的并行结构。generate 里可以使用条件语句 if 或者 case，但通常都是使用 for 来产生对象数组。generate 语句里可以包含门级器件，模块实例，寄存器，线，连续赋值表达式，always 或 initial 块等。但是在 generate 语句里不允许再包含 generate 语句。

总体来说，有两种调用 generate 的形式，条件和循环。我们会先介绍条件 generate，在条件编译里介绍它的知识。

12.1.4 条件宏和条件 generate

在 C 或 C++ 里，有很多编译的指令，也称为宏。它们的作用是进行条件编译。例如，下面列出了很多编译指令，每一行都以“#”开头，说明这是一条编译指令。

```
#define MacroName
...
#ifndef MacroName2
... (compile something)
#else
... (compile something different)
#endif
```

Verilog 里也有类似的指令，只是在 Verilog 里这些命令以重音符 (accent grave) `` 开头。这个符号也被叫做反引号 (backquote)，而 “#” 在 Verilog 被用来表示延迟或参数。在 Verilog 源代码的开头，我们都见过了 `timescale。在练习 1 里我们还用到了 `include，我们还讨论过关于 `default_nettype 的内容。

Verilog 里的编译向导包括 `define, `ifdef 等 (阅读 IEEE 1364 的 19 节或本书的第 23 章获得更多的内容)。它们的用法并不复杂，因此我们不会专门用篇幅来介绍它们，在用到它们的时候再来详细介绍。但是，理解编译向导和条件 generate 之间的区别是很重要的。

和其他的 Verilog 语句一样，generate 也可以使用参数。因此，利用 generate 语句，可以完成参数化的 IP 设计。

下面的这段代码说明了如何利用 generate 例化出两个不同的选通器 Mux01。

```
parameter Latch = 1;
...
generate
if (Latch==1)
    Mux32BitL Mux01(OutBus, ArgA, ArgB, Sel, Ena);
else Mux32Bit Mux01(OutBus, ArgA, ArgB, Sel, Ena/*unused internally*/);
endgenerate
```

利用条件编译，也可以完成同样的功能。

```
'define Latch 1 // Macro name, but no macro; used as flag.
...
`ifndef Latch // NOT `ifdef 'Latch; that substitutes a '1'
    Mux32BitL Mux01(OutBus, ArgA, ArgB, Sel, Ena);
`else
    Mux32Bit Mux01(OutBus, ArgA, ArgB, Sel, Ena/*unused internally*/);
`endif
```

不过，我们推荐读者使用条件 generate。这是因为 `define 的有效时刻是从编译工具第一次遇到它们的地方起开始有效，这样会带来两个问题：(a) 由于编译顺序的关系，当执行到 `ifndef 的时候，例如这个 Latch 参数还没有被定义（或者是被 `undef 了）；(b) 这个 Latch 可能被其他的代码里被 `define 过一次。这些问题有可能会带来不可预期的问题，而且可能并不会输出警告信息。

在前面讲到 `default_nettype 时，我们也遇到了类似的问题。应该尽量避免使用除了 `timescale 之外的编译向导。如果的确有使用编译向导的必要，只在一个文件里用，且在文件的最后用 `undef 每一个编译向导。

12.1.5 循环 generate

循环的 generate 可以用来描述不同规格的硬件，但是它的主要作用是用来描述参数化的、重复的硬件结构。而这种结构对写代码来说往往是费时的，容易出错的（即使是粘贴、复制也是这样）。循环的 generate 仍然可以包含条件结构，但是这里我们不再深入讨论这个复杂的话题。

genvar 类型

genvar 是一种特殊的，非负的，仅有循环 generate 才会用到的实型类型，它对综合和仿真是不可见的。当编译器把 generate 语句转换成实际的多组硬件时，genvar 是硬件的序号。这个转换过程可以看做是一个展开的过程。就像是展开一个毯子一样，循环的语句被一一展开，形成用来仿真的网表。

在 generate 语句里，可以使用多个 genvar。只需在使用它之前先声明即可。每一层嵌套的循环都必须使用各自单独的 genvar 类型。用来描述循环的语法结构几乎都是 for 语句。

12.1.6 generate 块和实例名

必须给包含循环的块命名；块的名字和 genvar 的值共同组成了对应的实例名。

从效果上来说，generate 块是一种基本的结构，相当于实例之中的子模块。这使得在 generate 块内部不用给对象的名字编号，不必使得它们的名称是唯一的。

不管是不是循环的 generate 块，都不能在其内部包含其他的 generate 块；Verilog 的语法禁止这么做。同时，还不允许包含 I/O 的声明，specify 块或参数的声明。但是，嵌套的循环是允许的。

假设我们要产生一个 8 比特的总线，总线的每一个比特驱动一个 D 触发器的 D 端，D 触发器的 Q 端送给一个三态缓冲的输入。

假设库里 D 触发器的类型名称是：DFFa；而实例名以 FF 开头。

输入数据的总线是 DBus，经过缓冲的输出是 DBusBuf。此外，还有 Clk 和 Rst。三态缓冲控制端的连线叫做 QEna。功能示意图参见图 12.3。

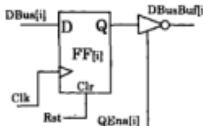


图 12.3 寄存器组的一部分

用 generate 来产生对应结构的代码如下：

```
// module I/O's include 8-bit DBusBuf output, and DBus & QEna inputs.
...
generate
  genvar i;
  for(i=0; i<=7; i=i+1)
    begin : BuffedBus // Here is the block name.
      wire QWire, QWireNot;
      DFFa FF (.Q(QWire), .D(DBus[i]), .Clr(Rst), .Clk(Clk));
      not Inv(QWireNot, QWire);
      bufif1 Buf(DBusBuf[i], QWireNot, QEna[i]); // notifi would be OK here.
    end,
  endgenerate
```

在这段代码里，我们使用了 Verilog 的原语 not 和 bufif1。其实，notifi 也能实现同样的功能。

在 generate 循环展开了之后，是 8 个有名块的阵列，分别是 BuffedBus[0], …, BuffedBus[7]。每一个展开的块都包含它自己的 Qwire, QWireNot, 三态门，一个 DFFa，一个 bufif1 和一个 not。逻辑就是这样通过序号 i 把电路展开的。

为了说明这个过程，当序号为 0 时，展开的逻辑实际上包含下面的部分：

```
BuffedBus[0].QWire
BuffedBus[0].QWireNot
BuffedBus[0].FF(.Q(BuffedBus[0].QWire), .D(DBus[0]), .Clr(Rst), .Clk(Clk))
BuffedBus[0].Inv(BuffedBus[0].QWireNot, BuffedBus[0].QWire)
BuffedBus[0].Buf(DBusBuf[0], BuffedBus[0].QWireNot, QEna[0])
```

DBus[0], QEna[0] 和 DBusBuf[0] 中的 0 来自于外部的序号而不是来自于 i。

展开的结果并没有产生名字类似于 FF[i] 的寄存器，但是得到了实现同样功能的 BuffedBus[i].FF。

展开的电路示意图参见图 12.4。

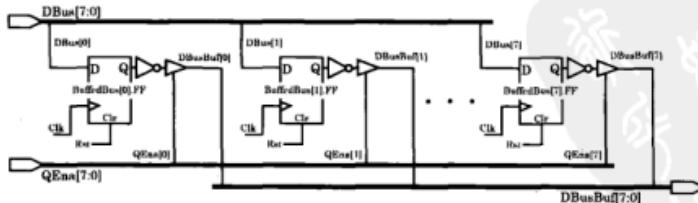


图 12.4 展开后的寄存器组，每一个寄存器都有独立的输出使能

再看看上面 generate 那段代码的作用：它决定了展开的实例名称，并且，它把 generate 后的实例和外部编号的对象相连。没有必要给 generate 的内部信号编号，因为展开后它们都处于

不同的实例中。例如，第三个 not 在网表里，引用它的方式应该是 BuffedBus[2].Inv。如果块名变成了 DB，则这个 not 变成了 DB[2].Inv。

如果外部对象就是以向量或数组形式存在的，则在 generate 循环中的引用必须使用 genvar 类型，把 generate 内部的对应信号排序，从而可以和外部的信号有一一对应的映射关系。如果外部信号不以向量和数组形式存在，不会在展开的块中被复制成多份，而是和所有的内部信号都连接上。我们看到，Clk 和 Rst 就是这么处理的。

可以在 generate 循环里声明线型、reg 型和整型。generate 循环里还允许有任务和函数，但是它们的声明不能放在 generate 循环里，而只能放在对应的模块中。

在理解了循环展开的过程之后，如果需要用 generate 把外部信号和内部信号相连，则 genvar 的编号还会被用做连线的序号。例如：

```
...
wire[7:0] QWire;
//
generate
  genvar i;
  for(i=0; i<=7; i=i+1)
    begin : IxedBus // Here is the block name.
      DFFa FF (.Q(QWire[i]), .D(DBus[i]), .Clr(Rst), .Clk(Clk) );
      notifl Nuf(DBusBuf[i], QWire[i], QEma[i]);
    end
  endgenerate
```

在这个例子里，当 i 为 0 时，这个缓冲器的名字为 IxedBus[0].Nuf。它通过外部连线 Qwire[0] 被 IxedBus[0].FF.Q 驱动，并驱动外部信号 DBusBuf[0]。它的控制开关是 QEma[0]。

下面的代码和上面的代码等效，只是驱动 Nuf 的连线是在内部声明的。

```
for(i=0; i<=7; i=i+1)
  begin : IxedBus // Here is the block name.
    wire Qwire;
    DFFa FF (.Q(QWire), .D(DBus[i]), .Clr(Rst), .Clk(Clk) );
    notifl Nuf(DBusBuf[i], QWire, QEma[i]);
  end
```

另外一种实现这个功能的写法是把整个寄存器和对应的缓冲逻辑都放到一个新模块中去，再给它取个名字，例如 FF。然后再用 generate 来循环产生每一个部分。

下面是一个用 generate 来实现组合逻辑解码器的代码。

```
parameter NumAddr = 1024;
...
generate
  genvar i;
  for (i=0; i<NumAddr; i=i+1)
    begin : Decode
      assign #1 AdrEna[i] = (i==Address)? 1'b1: 1'b0;
    end
  endgenerate
```

上面的代码产生了和变量 Address 相同位宽的 10 比特输入总线，输出总线 AdrEna 是 1024 比特宽的。当 Address 的值和 0~1023 某一个具体的值（假设是 n）相等时，这第 n 个 AdrEna

在一个单位延迟后会被拉高，且其比特被拉低或维持低电平。这个功能可以用来使能对 RAM 里某一个 word 的读操作。

如果不使用 generate，仅仅用简单的过程语句也可以实现这个解码器。代码如下所示：

```
parameter NumAddr = 1024;
integer i;
...
reg[NumAddr-1:0] AdrEna;
always@(Address)
begin : Decoder
  for (i=0; i<NumAddr; i=i+1)
    #1 AdrEna[i] = (i==Address)? 1'b1: 1'b0;
end
```

12.1.7 用 generate 来实现解码树

让我们来看一个更大的解码器设计。如果设计者使用的库元件是 4-16 解码器，这个解码树看起来应该如图 12.5 所示。

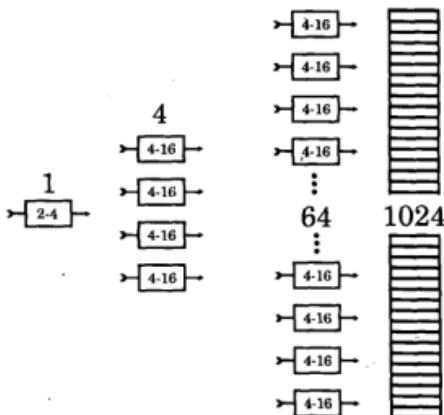


图 12.5 由 4-16 解码器组成的 10-1024 解码器

最左边的第一级解码器选择了 4 个第二级解码器。第二级解码器每一个都和 16 个第三级解码器相连，总共是 64 个。每一个第三级解码器都解析了 16 个地址，总共组成了 1024 个地址（对应于 10 比特的总线位宽）。

需要给每一个解码器都增加使能端，如果解码器的使能无效，则解码器所有比特都输出 0。图 12.6 是这个电路的示意图。

电路和功能很容易对应起来。对于一个完整的解码树来说，第一级的两个输入对应于 10 比特输入的最低两个比特。

再往下每一级都可以表示 4 比特的信息，因为这 4 个比特正好可以组成二进制的格式。因此，第二级对应于接下来的 4 个比特：2~5；而第三级对应于 6~9 比特。

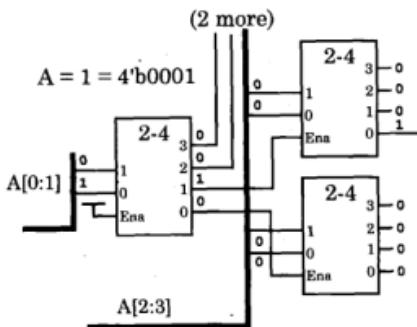
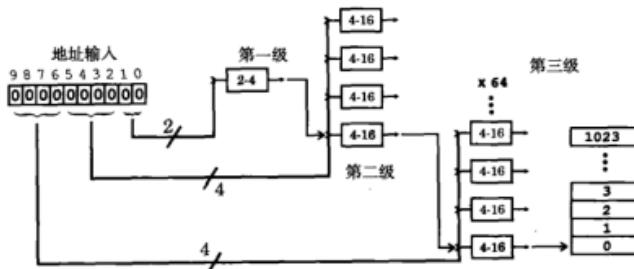
图 12.6 当 $A = 1$ 时的工作情况，为了简单，使用的是 2-4 译码器

图 12.7 说明了当选中地址 0 时的解码器路径。

图 12.7 地址 $10'h0$ 被解码的过程。箭头较大的分支是正在工作的分支

通常，对于这种寻址的方式，设计者并不关心要解码的值（地址）是否真正和总线上地址组成的数是完全一样的。由于读和写地址都用的是同一种表达方式，因此同样的地址肯定对应的是同一个单元。而真正重要的是，这种寻址方式对应的地址必须是唯一的。例如，图 12.7 中的电路表示的是 $10'h0$ 。如果地址是 $10'h1$ ，它会选中第二级解码器 1（从 0 开始计数），电路地址对应的数实际上是 256。而对应用的层面来说，并不关心地址 1 是否对应于电路中的 256。

由于第一级解码器非常简单，因此没有必要用 generate 语句来描述它。假设在综合库中已经有一个名为 Dec4_16 的 4-16 解码器元件。我们用它来实现自己的解码器。其中，输入信号都以 in 为前缀，输出信号以 Decoded 为前缀。

```
// Level 1 decode is trivial:
wire[3:0] DecodedL1;
wire[1:0] inL1;
wire[15:4] DecodedUnused; // Stub to suppress warnings.
//
// Get Address LSB's:
assign inL1 = Address[1:0];
// The level 1 decode, using verilog concatenation:
//      output[15:0] , input[3:0]
Dec4_16 U1({DecodedUnused, DecodedL1}, {2'b0, inL1}); // Done!
...
```

接着，利用 generate 来产生第二级解码器。

在实现了第二级解码器之后，可以在它的基础上完成第三级解码器的代码。第二级解码器只有 4 个，还可以用人工来一个一个地编写。但对于第三级解码器来说，它有 64 个，如果靠手工来写这么多代码就是件很困难的事情了。因此，一定会用到 generate 语句。下面是第二级解码器的代码：

```
// The Level 2 decode, which requires 4 decoders, fully utilized:
wire[16*4-1:0] DecodedL2;
wire[3:0] inL2;
assign inL2 = Address[5:2];
//
generate
  genvar i;
  // Generate the 4 4-16's:
  for(i=0; i<=3; i=i+1)
    begin : DL2
      wire[15:0] tempL2;
      Dec4_16 U2(tempL2, inL2); // Each gets bits 2 - 5.
    //
    // Compose the decoded address from L1 and L2, and assign the bit:
    assign DecodedL2[(16*(i+1)-1):16*i] =
      (DecodedL1[i]==1'b1) ? DL2[i].tempL2: 'b0;
  end
endgenerate
```

第三级解码由 64 个解码器组成。如果不使用 generate 语句，实现起来会非常困难。经过这一级解码器，1024 个比特都可以被选中。

```
// The Level 3 decode requires 64 x 4-16's:
reg[(16*4)*16-1:0] AddrEna;
wire[3:0] inL3;
assign inL3 = Address[9:6];
//
generate
  genvar j;
  // Generate the 64 4-16's:
  for(j=0; j<=4*16-1; j=j+1)
    begin : DL3
      wire[15:0] tempL3;
      Dec4_16 U3(tempL3, inL3); // Each gets bits 6 - 9.
    //
    // Compose the decoded address from L2 and L3, and assign the bit:
    assign AddrEna[(16*(j+1)-1):16*j] =
      (DecodedL2[j]==1'b1) ? DL3[j].tempL3: 'b0;
  end
endgenerate
```

在完成 generate 结构时，最重要的规则是要记住 generate 的结果都是结构性的。也就是说，在整个仿真期间，generate 里的结构是不可变的。

12.1.8 generate 循环的有效范围

思考下面三段代码的区别：

```

generate
for (i=0; i<Max; i = i+1)
begin : Stuff
reg temp;
and A(ABus[i], InBus[i], InBus[i+1]);
always@(posedge Clk) #1 temp <= &InBus;
assign #1 OutBit1 = temp;
end
endgenerate

```

```

reg temp;
always@(ABus) #1 temp <= &ABus;
assign #1 OutBit1 = temp;
generate
for (i=0; i<Max; i = i+1)
begin : InAnd
and A(ABus[i], InBus[i], InBus[i+1]);
end
endgenerate

```

```

generate
reg temp;
for (i=0; i<Max; i = i+1)
begin : InAnd
and A(ABus[i], InBus[i], InBus[i+1]);
end
always@(ABus) #1 temp = &ABus;
assign #1 OutBit1 = temp;
endgenerate

```

12.2 练习 15: generate

在目录Lab15下完成这个练习。记得分别保存第1步和第2步的结果，我们将在第3步比较前两步的结果。

练习步骤

我们会用generate结构来重写Mem1kx32这个memory。在将来的FIFO设计中，我们将会用到这个memory。

第1步。数组实例。前面我们曾经实现了8位的总线缓冲电路。用数组实例替换掉上面的第二段代码（名为IxedBus，用到了notif器件的那段代码），不用generate语句。仿真你的设计并综合出网表。把网表保留下来（在下面还要用到）。

第2步。generate实例。把前面名为BuffedBus的generate块放入一个模块中，修改相应的参数定义。在generate循环中使用参数，使它和genvar一起确保总线位宽的连贯性。仿真这个设计。用与第1步一样的约束来综合这个设计，并把网表和第1步产生的网表相比。看看网表有没有区别？

第3步。面积和速度的对比。任意选第1步或第2步的设计，用按面积综合（没有速度约束）和按速度综合（没有面积约束），比较两个综合的结果。

第4步。用 generate 产生 RAM。在练习7（参见第5章）里，我们用行为级的 Verilog 代码设计了名为 Mem1kx32 的 RAM 模块。

让我们再来实现这个RAM，这一次，模块名为 Mem1kx32gen。这个RAM的位宽是33比特（第33比特用来实现奇偶校验）。

可以按照下面的步骤来完成这个练习：

- 删除 memory 中原来的数组声明。只把模块 Mem1kx32 的 I/O 复制到 Mem1kx32gen 中。模块 Mem1kx32 的结构不适合加入 generate 结构，因此不要再在这个旧文件上花时间了。但是，原来的 testbench 还是可以用的。因此，把练习7用的 testbench 复制过来。
- 完成一个输入是 D 和 Clk，输出是 Q 的 D 触发器 RTL 行为级模块，模块名是 Bit。这个模块不需要清零输入也不需要在输出 Q 取反。
- 用 generate 语句和 genvar 类型的变量 j 来完成这个 33 比特的寄存器组。

和图12.8一样，每一个 Bit 实例都连在同样的时钟上。对于一个 32 位的字来说，每一个比特的 D 和 Q 都和外部对应的信号相连。展开的电路示意图参见图 12.9。

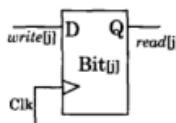


图 12.8 矢量中的第 j 个单元

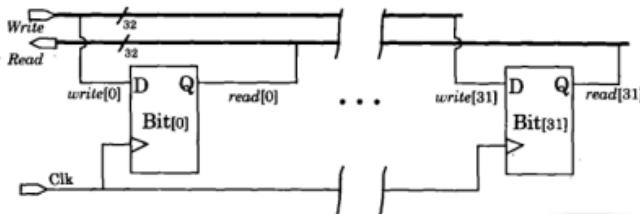


图 12.9 展开后的电路结构图

在完成了32比特的储存单元后，再用 generate 单独产生第33比特，来进行奇偶校验。

- 再用一个 generate 语句和 genvar i 调用第4步 C 里的 generate 结构，产生 4 个 RAM 的存储空间（地址）。为了只输出选中的字的值，最简单的做法是给每一个比特都加上输出缓冲。这样，当它的地址无效时，它的输出（寄存器的 Q 端）也是无效的。通过对地址线解码来实现这个无效信号的产生。

这将是 memory 模型的原型。你的 testbench 可以利用这 4 个地址测试到所有的边界情况。图 12.10 是对应的电路示意图。

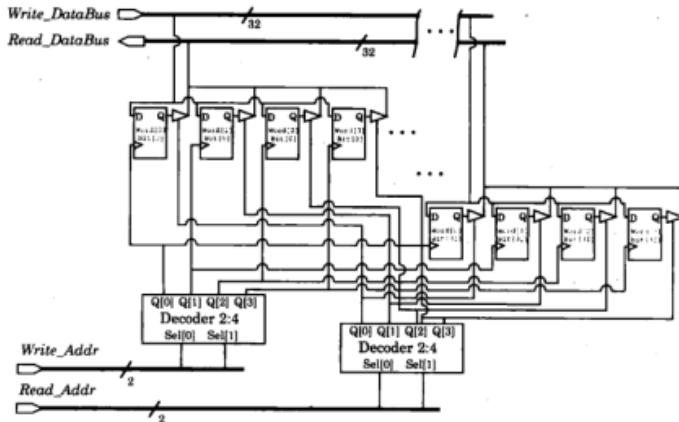


图 12.10 4 地址 memory 的示意图

E. 定义一个参数 DHiBit, 通过它来间接地控制 genvar j。

如果数据总线的位宽是利用参数 DHiBit 来定义的: reg[DHiBit:0], 则这个位宽是:

```
localparam DWidth = DHiBit+1;
```

接下来

```
genvar j; ... for (j=0; j<DWidth; ...
```

用同样的方法, 用 AHiBit 定义地址总线的位宽。则这个位宽是:

```
localparam AWidth = AHiBit+1;
```

地址的总数为:

```
localparam NumAddrs = 1<<AWidth; // Same as 2**AWidth.
```

于是这样来控制 i,

```
genvar i; ... for (i=0; i<NumAddrs; ...
```

例如, 当 AHiBit 为 1 时, 地址总线的位宽是 2, 因此地址的总数为 $1 << 2 = 4$ 。还可以用类似的方法来计算 AWidth-1, 用它来表示 memory 的最大范围。

F. 根据下面的要求给这个 RAM 加上完整的功能。在添加了完整的功能之后, 仿真一些地址的读写操作。把 AHiBit 改成 4 从而可以表示完整的 1 kB × 32 的 RAM (仿真波形参见图 12.11)。综合和优化这个 RAM 的过程需要花一些时间 (打开答案中的综合脚本, 看看为什么需要这么长的时间), 因此, 在把 RAM 包进了 FIFO 模块中之后再来做这个综合。



图 12.11 generate 的 32 字 Mem1kx32gen RAM 仿真波形

RAM Mem1kx32gen 需求

把带奇偶校验的 1 kB × 32 的 Verilog 静态 RAM 模型叫做 Mem1kx32gen。

用 generate 语句产生一个基于 D 触发器的 memory 核心。它有两个功能：一是实现存储的阵列；二是可以通过地址来选通。可以使用行为级或 RTL 级的 Verilog 语句来完成这个核心的设计。

用参数来描述地址总线的位宽和整个地址空间。奇偶校验不需要地址，对于芯片外的器件来说，它是不可见的。

RAM 有读和写两个 32 比特的数据总线。RAM 还有一个异步的 chip enable 信号，当它无效时，输出数据为“z”；但是它不会影响到 RAM 内已经存储的数据。当 chip enable 无效时，RAM 不工作。

RAM 有读和写两个控制信号输入。读写操作只在时钟的上升沿有效。如果读地址的条件没有变，读数据应该一直保持在读的数据总线上。如果发生了读写同时有效的情况，则读操作优先而忽略写操作。

为数据和地址总线的变化增加一些延迟，增加输出有效端口，通知外部电路发生了读操作且读数据稳定并有效。不用担心在读有效时，地址和时钟同时发生变化的情况。使用这个 RAM 的系统会保证这个时序。

增加奇偶校验错误输出端口，若在读操作时发现了奇偶校验错误，则将奇偶校验错误输出一直拉高，直到这个地址的数值没有错误时再拉低。

第 5 步。 实现一个包含这个 RAM 的 FIFO。在目录 Lab15 下新建一个子目录，把练习 11 里的 FIFO 模型复制过来。这个模型应该包括 3 个文件：FIFO_Top.v, FIFOStateM.v 和 Mem1kx32.v。

把 FIFO 里的 Mem1kx32 替换成用 generate 产生的 Mem1kx32gen。对于 demo 板的 Silos 仿真工具来说，这个过程可能大了一些。

仿真这个 FIFO 的功能。先综合这个 memory 而不是 FIFO。先不要综合这个 FIFO 的原因如下所述。

首先，就算我们只指定按面积优化的原则来综合这个工程，而不增加其他的约束（如扇出，负载，驱动限制等）和综合速度的约束，综合这个工程的时间也比较长。因为这个工程的规模相对大一些（约 3 万个等效晶体管的规模）。通常，类似的寄存器阵列不会用综合工具调用基本门来实现，而使用的是宏单元或库。

综合的报告会通知我们没有任何逻辑和奇偶校验比特相连。这是因为在这个模型里，我们没有用到奇偶校验这个功能。

其次，综合这个设计：按面积最小优化，时钟的周期为 500 ns，输出的最大延迟为 50 ns。除此之外，不增加任何的综合约束。如果增加任何的一种约束都会使得优化的时间显著加长。原因不在设计本身，而是 generate 造成的。不要给任何受时钟驱动信号增加输入输出延迟，因为这同样会明显地增加优化的时间。

再次，这一步是可选的，根据你的时间来决定是否完成这一步。

按照下面的步骤完成综合的过程：按面积最小优化，定义时钟周期为 500 ns，不加别的约束。在综合的时候，在 DC 中把 memory 模块设为 set_dont_touch，然后用同样的综合脚本，采用增量编译的方式来综合这个设计。

在进行编译前，还可以设置以下约束。

```
set_drive    10.0 [all.inputs]
set_load    30.0 [all.outputs]
set_max_fanout 30 [all.designs]
#
set_max_delay 50 [all.outputs]
set_output_delay 1 [all.outputs] -clock Clocker
set_input_delay 1 [all.inputs] -clock Clocker
```

拿综合出来的 FIFO 网表来仿真是有问题的。问题不是出在 generate 语句上，而是这个 FIFO 控制器的状态机是个不可综合的设计。在后面的章节我们将会继续深入讨论关于这个 FIFO 的内容。

12.2.1 练习后的思考

把数组形式的实例和用 generate 产生的实例做比较。

如果在同一个模块里，一个整型变量，一个是 genvar 类型的变量，两个变量的名字可以一样吗？

generate 块有自己的命名有效范围吗？

12.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 3.6 节关于命名有效范围和层次关系的内容。

阅读 Thomas and Moorby (2002) 的 5.3 节和 5.4 节关于数组实例和 generate 块的内容。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003)。

阅读附录 C 中关于编译指令的内容。仅从名字就能猜到很多命令的功能。

阅读 7.8 节关于 generate 的内容。

第13章 函数、任务和串并转换

13.1 串并转换

为了深入学习 Verilog，我们把串行/解串器的工程放在了一边。现在，接着来实现串行/解串器的解串器的部分。

13.1.1 简单的串并转换器

我们已经实现了串行帧解码器（参见第4章中的练习6，第8步）。我们还学习了利用帧数据包协议实现一个同步PLL，并且用这个PLL的输入串行数据流中解码出我们想要的数据的过程（参见第7章）。

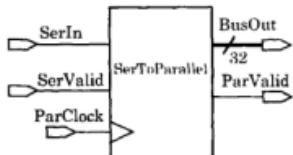


图 13.1 串行转换器示意图

在本章中，我们将不仅仅是把模块整合在一起，或把它改成可综合的设计，而是深入地学习串并转换的内容。

图 13.1 里的电路是一个简单的串行转换器。并行总线上的输出数据经过寄存器的锁存，和时钟同步。此外，这个电路至少还要有一个串行输入。设计一个纯组合逻辑的解串器是可行的，但是如果输出不和时钟同步，别的模块很难使用这个不和时钟同步的输出。

这个设计还需要输入一个串行时钟。如果串行时钟的频率和并行时钟的频率相同，则可以直接使用这个并行时钟。如果它们的频率不相同，则需要在进行串行处理时自行产生这个时钟。例如在我们的串行/解串器工程里，串行时钟是利用串行数据来产生的。

设计里还需要一些内部寄存器用来保存中间数据。另外，平行输出的数据必须都用寄存器寄存一拍，否则输出有可能不能满足时序的要求。

还可以输出一个标志信号，说明输出的平行数据什么时候有效。实现这个功能可以通过对时钟计数来实现。输入端还需要一个有效标志，把它叫做 SerValid，它说明了输入的串行数据什么时候有效。这个 SerValid 信号可以随着串行数据而翻转。如果我们能使串行数据和时钟完全同步，那么也可以不用这个有效标志。由于在这个串并转换器的并行侧有移位寄存器，因此我们可以在并行侧加入可选的复位输入；而串行侧的接收逻辑并不会控制串行数据的发送器，因此，在串行侧不需要加入复位输入。

串并转换器功能小结

- 并行输出数据经过寄存器锁存
- 并行侧有输入时钟
- 串行输入数据
- 串行侧有输入时钟（可选）

- 内部的解串寄存器
- 并行输出有效标志（可选）
- 串行输入有效标准（可选）
- 并行侧的复位输入（可选）
- 串行侧不需要复位输入

在这个串行/解串器的工程里，串行时钟包含在串行数据流之中，不管有没有 SerValid 这个标志信号，都必须从数据流中分离出时钟信号。这说明在时钟的相位和频率被恢复之前，会损失一些数据。但是没有单独的时钟也是有好处的，没有时钟相位的问题，也不存在信号之间的互相干扰、噪声等问题。只要数据有效，就能恢复出时钟。这样的处理使得数据信号和设计里的任何其他信号都无关，为了确保当今频率达到吉赫兹的高速数据传输电路的低误码率，这一点是非常重要的。

13.1.2 用函数和任务来描述解串器

如果不考虑帧边界的问题，那么在 Verilog 里，并行化再简单不过了：我们依次把串行数据移至寄存器里，直到个数等于并行输出的位宽后把移位寄存器里的数放到并行总线上，然后接着移位。同时，我们假设并行时钟是足够快的，不会因为这个时钟而损失串行数据。

下面是用 Verilog 来实现一个简单的解串器。

第一，让我们来完成移位。用一个函数来完成这个功能。这个函数每隔一段延迟时间就进行一次移位操作。对于仿真来说，完成这个移位操作所需的时间是 0 仿真时间。由于 Verilog 有自建的移位操作符，因此，这个函数只须使用这个操作符即可完成。

```
parameter ParHi = 31;
...
function[ParHi:0] Shift1(input[ParHi:0] OldSR, input NewBit);
  reg[ParHi:0] temp;
begin
  temp = OldSR;
  temp = temp<<1; // MSB goes lost.
  temp[0] = NewBit;
  Shift1 = temp;
end
endfunction
```

当我们调用 Shift1 这个函数时，只须把当前移位寄存器的值和串行输入的值传递给这个函数即可。这个函数可以写得更简单一些，上面那样写只是为了使读者更清楚。

第二，让我们来完成转换。把移位寄存器的值放到并行的数据总线上。隔一些延迟后，输出一个 ParValid 有效标志。可以用任务来实现这个功能。

```
parameter ParHi = 31;
...
reg ParValidFlagr;
reg[ParHi:0] ParSR, ParBusReg;
//           rise, fall
assign #( 1,      0 ) ParValidFlag = ParValidFlagr;
...
```

```

task Unload32; // Copies the parallel SR to the output bus.
begin          // Also clears the SR for the next word.
    ParValidFlagr = 1'b0; // Lower the parallel-valid flag.
    ParBusReg     = ParSR; // Transfer the data.
#5 ParSR       = 'b0; // Clear the SR.
    ParValidFlagr = 1'b1; // Raise the flag.
end
endtask

```

这里的延迟只是为了说明问题。请记住，如果要完成一个可用的设计，不要在过程执行代码里加入延迟。如果需要人为添加模块输出的块延迟，尽量保证这个延迟值接近真实值。

第三，让我们来完成这个移位过程。每一个时钟移位寄存器都进行一次移位，从而保证串行数据的有效。

实际上，没有必要在每次完成第二步的转换之后都去清空这组移位寄存器。通过用一个计数器对移位的状态进行计数，就能够很清楚地知道当前的移位状态。只是如果我们在每次转换完成之后把移位寄存器都清0，在仿真时候，看波形更容易理解一些。

函数 Shift1 里的寄存器 temp 也是为了仿真时看波形更容易，这个函数写的更紧凑一些为：

```

function[ParHi:0] Shift1(input[ParHi:0] OldSR, input NewBit);
begin
    OldSR = OldSR<<1;
    Shift1 = {OldSR[ParHi:1],NewBit};
end
endfunction

```

用一个 always 块调用这些函数和任务实现整个的功能。

```

always@(posedge ParClk, posedge ParRst)
begin : Shifter
    if (ParRst==1'b1)
        begin
            N      <= 0; // N counts the bits shifted.
            ParSR   <= 'b0; // The shift register.
            ParBusReg <= 'bz; // The parallel out bus.
            ParValidReg <= 1'b0; // ParValid.
        end
    else if (SerValid==1'b1) // Ignore the serial line if 0.
        begin
            ParSR <= Shift1(ParSR, SerIn); // function called.
            N     <= N + 1;
            if (N>ParHi) // If 32 bits shifted.
                begin
                    Unload32; // task called.
                    N <= 0;
                end
            end // SerValid.
        end // Shifter.

```

13.2 练习前预习：解串器

下面的练习包含一些非常重要的内容。在完成了下面的内容之后，一个初步的串行/解串器就基本上完成了。

首先，必须要理解解串译码器（模块 DesDecoder）的功能：它能在输入的串行数据流中找到帧的边界。同时，它还恢复了和发送端同步的1 MHz的时钟。

在我们的设计中，每一帧有16比特，并且以特定的8比特数据作为帧的边界。DesDecoder会从数据流中找到有效的数据位。这和前面练习里用到的4-16解码器不一样，不是一个简单的逻辑译码过程。通过解析输入的数据包里数据，时钟发送端的时钟也被恢复了出来。

解串器可能会横跨两个不同的时钟域。串行数据的输入速率只和发送端的时钟频率有关；而解串器使用接收端的时钟把FIFO中的串行数据读出来。因此，这个FIFO的输入是发送时钟域的，而FIFO的输出是接收时钟域的。

再来看Deserializer的PLL，它产生了一个频率为1 MHz的时钟。DesDecoder希望这个时钟和发送端的时钟同步。产生这个时钟的过程类似于一个压控振荡器的工作过程。输出时钟的频率只与解串的操作和输入的数据有关。PLL的比较器有两个时钟输入：一是自己产生的1 MHz时钟，而这个时钟被32倍频之后用来接收串行数据；二是DesDecoder从串行数据里恢复出来的时钟。如果这个时钟和输入的串行数据流是完全同步的，那么发送端送过来的每一个数据包都能够被正确解析。

如果DesDecoder解码失败，则意味着时钟没有同步。PLL的比较器不停地比较自己产生的1 MHz的时钟和从数据中恢复出来时钟。只有在DesDecoder成功地解出第一个数据包之后，PLL才会调整时钟的频率。PLL会细微地调整时钟频率使得它和数据包的边界对齐。

这个设计用整型变量来产生了两个时钟。这两个时钟分别是：发送端自己使用的时钟和Deserializer的PLL产生的那个时钟。由于1 MHz时钟的1/32不是一个整数，只能被四舍五入处理，因此，这个PLL不像模拟PLL那样能够真正的锁定一个频率。但是，它能够产生一个频率足够接近的时钟。如果用这个时钟来解码，可以在串行数据失锁之前解码出足够的数据。

在开始进行练习之前，读者可以先回顾一下在第4章中制定的关于完成串行/解串器设计的计划。在本练习里，我们不完成恢复发送端时钟这一步，而是把它留到将来再完成。我们只需完成对数据包的解码就可以了。

图13.2是第4章中里串并转换器（解串器，deserializer）的示意图。在这个练习里，我们会完成一个简化后的解串器。

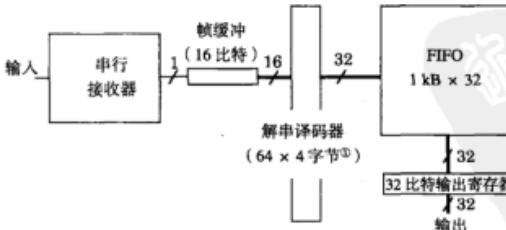


图13.2 第4章中的解串器的数据流示意图

① 原书此处为frames，疑有误——编者注。

回忆一下第4章中练习6第4步的内容，当时我们定义一帧有16比特，其中8个比特是有效的串行数据。

数据流的格式是这样的：帧由数据和帧的间隔组成，帧的间隔先是3个比特的0，接着是2比特的计数值，标志着当前的比特是4比特中的第几个比特，接着再跟着3比特的0。32比特的实际数据用“x”来表示，每一个x都表示一个1或者0。发送的时候先发送MSB，即是说先发送左边的，再发送右边的。

```
64'bxxxxxxxx00011000xxxxxxxx00010000xxxxxxxx00001000xxxxxxxx00000000,
```

13.2.1 Deserializer 的小改动——早期的 ECO

图13.2只是一个数据流的示意图，它说明了数据在进入解码器之前应该被转换成16比特的并行数据。然后，如果此时时钟还没有实现同步，帧缓冲（Frame Buffer）无法从它保存的数据里找到边界。仅从这个图里看不出来帧缓冲（Frame Buffer）里保存的16比特数据是否跨过了数据的边界。

帧缓冲是一个串并转换器，它需要和真正的数据边界对齐。这需要图13.2里的解码器利用PLL把恢复的时钟反馈给Serial Receiver。

从数据流的角度来思考，把帧缓冲的功能放到解码器的逻辑里会更好一些，而不把它视为一个单独的模块。

解码器会自己来处理串行的数据流。新的方块图如图13.3所示。

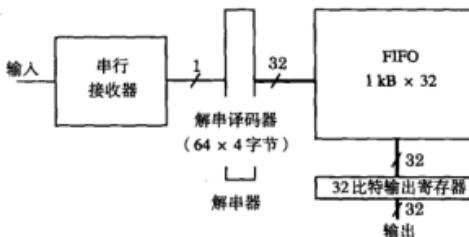


图13.3 升级后的解串器的数据流示意图

在本练习中，我们并不关心PLL时钟同步的功能。假设PLL的功能在串行接收器的这个方块内实现，帧缓冲这个方块就可以被删掉了；同时，这样划分模块的话也不需要译码器输出反馈了。

这个更新后的方块图仅仅是个数据的流向示意图，并不包含任何的时钟信息。

由于需要从64比特里解码出32比特的有效数据，因此需要两个1MHz的ParClk时钟的时间来解码32个有效的比特。在练习10的第4步里讨论了这些内容。在这里我们只假设从端口SerIn输入的串行数据速率是32Mb/s，SerClk是32MHz的外部时钟。

13.2.2 一个设计划分的问题

抛开数据的流向，我们再来思考另外一个问题：帧同步的那部分逻辑应该放到解串译码器里还是应该和PLL时钟逻辑一起放在串行接收器里？帧同步的这部分Verilog代码其实已经

基本完成了(参见第7章中的练习10里的范例代码)。帧同步和串行数据时钟提取从本质上来说实际上是等效的。

如果我们把串行时钟提取放在串行接收器里,则PLL的所有逻辑也都在串行接收器里了。在这个条件下,还需要额外增加移位寄存器或寄存器阵列来处理数据边界的问题。串行接收器在发送串行数据的同时把帧的边界信号一块儿发给解串译码器(DesDecoder)。这样,DesDecoder里的并行化逻辑会变得非常简单。

如果串行时钟提取逻辑放到DesDecoder里,则PLL可以放到串行接收器里,也可以放到DesDecoder里。如果把PLL放到串行接收器里,则DesDecoder需要给串行接收器反馈同步信息;如果把PLL放到DesDecoder里,则在DesDecoder里需要实现数字模拟混合的电路,在进行后端处理时需要特殊处理。

综合上面的各种考虑,我们还是把PLL放在串行接收器里,而在DesDecoder里完成帧同步(串行时钟提取)的工作。这样的划分使得串行和模拟部分功能无关,从而减低这部分数字逻辑设计的复杂度。DesDecoder会从串行的数据流中恢复出1MHz的时钟并把它再返回给PLL,图13.4说明了这样的过程。

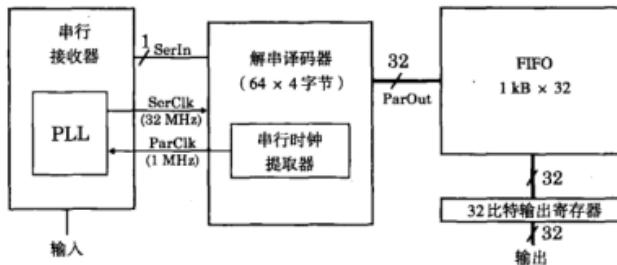


图 13.4 Deserializer 最终的细节方块图

1MHz的时钟是个很低的频率。除了考虑一下DesDecoder返回给PLL的时钟的传输延迟以及时钟相位偏移之外,我们不需要对它进行什么特殊的处理。

在完成了ECO之后,可以做我们的练习了。

13.3 练习 16: 串并转换

在目录Lab16下完成以下练习。

练习步骤

先做一些小练习热身,然后我们会讨论并实现一个初步的解串器。

第1步。解串器。利用上面讲到的知识,实现一个解串器。这个解串器要求实现上面讲到的功能,并且包含SerToParallel的所有I/O端口。这个解串器只需要数够32个串行比特,再用同一个时钟(相反的那个时钟沿)把它们再放到并行总线上就可以了。假设这些数据是没有经过帧同步的,且每当SerValid有效时都应采样这32个比特的数据。

用至少三组串行数据仿真这个设计。用\$random来产生串行数据(参见图13.5和图13.6)。不要花时间来综合这个设计。

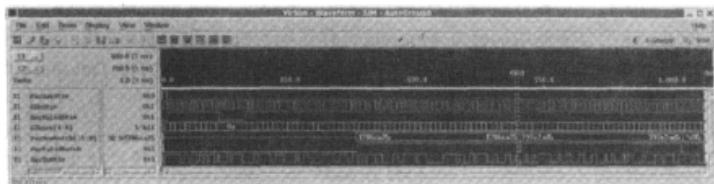


图 13.5 解串器的仿真波形

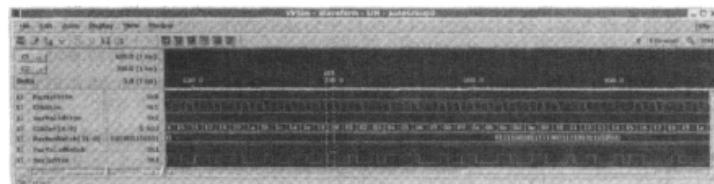


图 13.6 串行计数翻转时放大的波形

第2步。解串数据流的同步。升级第1步里解串器的设计。当解串器检测到连续12个0时，停止解串并抛弃这12个0。但这12个连续0之前的数据应当被保存下来。当检测到连续12个1之后，解串器再次开始工作。这12个连续1之后的数据和之前保存的数据拼在一起，解串器利用这些数据继续解串。

可以用两个标志信号来说明检测到了全0和全1的两种数据流，它们分别是：Found_stop 和 Found_start。

先验证这个检测的逻辑是否工作正常。然后在设计里增加一组并行寄存器，一直把输入数据保存在这组寄存器中直到Found_stop有效为止。当Found_start有效时，继续往这组寄存器中保存数据。Found_stop使得Found_start无效；反之亦然。每当保存了32个比特，就把这32个比特放到输出总线上，同时输出有效信号parallel_valid。

实现这个需求的方式可以有很多种。不管解串器有没有给总线发送数据，解串器内部的移位寄存器总是在不停移位的。

图13.7是这个设计仿真的结果，但是这个设计是不可综合的。

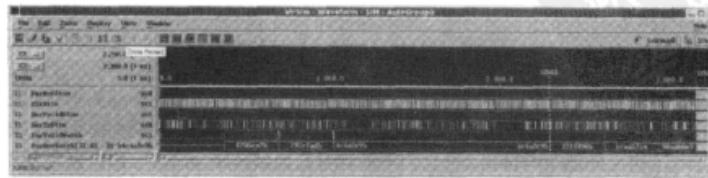


图 13.7 同步但不可综合的解串器的仿真波形

第 3 步。解串器帧解码。为了完成数据帧同步，我们建立一条比练习 10 里 PLL 锁定的判决稍为宽松一些的准则。

我们这样判定完成了同步：当数据 8'b000_00_000 被送进了 SerIn 之后，就认为已经成功同步。解串的工作和同步的工作是并行的。也可以把判决的条件加强一些：直到 4 组边界值移进了移位寄存器之后才认为是同步成功。

在完成了同步之后，每数够 16 个从 SerIn 输入的比特，ParClk 就会翻转一次。因此，接收到的 64 个比特对应于两个 ParClk 的周期，也就是 4 个边沿。而这和当前是否同步没有关系。

而失步的判决是这样的：如果两比特连续计数器的数值 ($\dots, 2'b11 \rightarrow 2'b10 \rightarrow 2'b01 \rightarrow 2'b00, \dots$) 不再连续，或者是移位的值填不满 64 个比特了，我们都认为是发生了失步。但是，不管是上面的哪种情况，都不会影响到 ParClk，因为它是由 PLL 时钟产生的，除非 PLL 收到了调整频率的指令，否则这个频率是不会变的。

在失步的状态下，不应该进行解串的操作。这时，不完整的数据包应该被丢弃。必须等到再次同步上之后，再进行解串的操作。

在重新同步的过程中，ParClk 信号可能会产生毛刺。对于 DesDecoder 来说，我们可以把它设计成不受这个毛刺的影响。而这正是我们在这个解串器中加入 FIFO 的一个原因。

总结 DesDecoder 的功能：DesDecoder 的输入是串行输入的数字信号，DesDecoder 受外部产生的时钟 SerClk 驱动，而解串器是不使用这个 SerClk 时钟的。完成对应的 test-bench，这个 testbench 里应该有 SerClk，符合帧格式的 64 比特的 SerIn 数据流，图 13.8 是其端口示意图。

DesDecoder 从串行数据流中提取出 1 MHz 的时钟，并用它作为并行 32 比特数据的输出时钟。先不考虑 PLL 的功能，只产生一个和串行输入信号同相的 32 MHz 的串行时钟即可。端口 SerIn 接串行数据；SerClk 接串行输入时钟；ParClk 是从串行数据中提取的输出时钟；ParBus 是输出的 32 比特总线，受时钟 ParClk 驱动。

可以利用练习 10 或者其他练习中的代码，完成这个 DesDecoder 的功能。可以部分的粘贴复制以前代码中的片段，这样完成这部分的代码会更容易一些。尤其是移位寄存器的那部分逻辑，和本练习里第 2 步的内容非常接近。

可以把整个设计分成若干小的任务，通过调用这些任务来实现整个功能。

仿真你的设计。图 13.9 和图 13.10 是仿真的结果。综合这部分逻辑的工作我们将会放到以后再做。

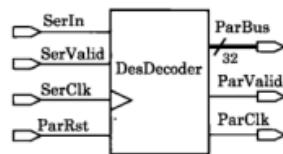


图 13.8 第 1 版解串译码器的方块图

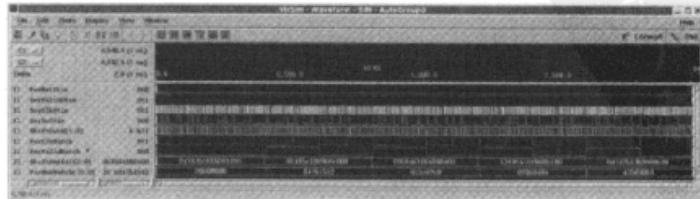


图 13.9 DesDecoder 的仿真结果

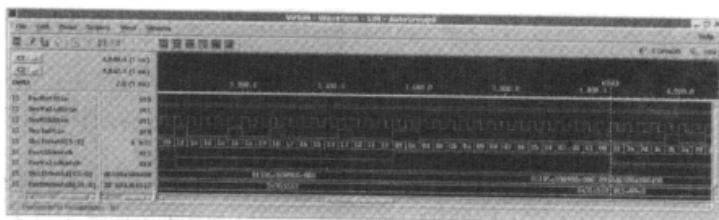


图 13.10 DesDecoder 局部放大的仿真结果

13.3.1 实验后的思考

思考如果数据包的位宽变大，应该怎样修改设计（如增大到 128 比特）？

13.3.2 补充学习

再次阅读本练习里的每一个步骤，同时复习本练习参考到的以前的练习。

第14章 UDP 和开关级模型

14.1 用户定义原语、时序参数和开关级模型

我们先把串行/解码器的设计放到一边，继续深入地学习 Verilog。在这一章里，我们将从门级的角度深入地理解设计中用到的基本元件。

14.1.1 用户定义原语 (UDP)

用户定义原语 (User-Defined Primitives, UDP) 给用户创建小规模集成电路 (Small-Scale Integrated, SSI) 的器件模型提供了条件。仿真这样的模型对仿真服务器的要求很低，同时速度也很快。UDP 的目标元件可以是任何一种寄存器或复杂的组合逻辑门。

从 Verilog 的层次上来说，UDP 和模块 (module) 是同一级别的，不存在谁的层次更高的问题。它们实现不了的功能，模块也实现不了。通常，多个 UDP 模型以模块的形式被放在同一个库文件里。通常，这些原语在被例化的时候，在它们的上一层会被加一个壳 (wrapper)，设计者往往在这个 wrapper 中添加时序信息，实现多个输出端口等基础的功能。

UDP 可以有多组输入，但是只允许有一组输出。Verilog 规范 1364 规定了对于组合逻辑 UDP 来说，输入可以多于 9 个；对时序逻辑来说，输入可以多于 10 个。输入输出端口都必须是 1 比特宽。具体的功能体现在查找表中。

UDP 是不可综合的。

UDP 的 Verilog 语法结构是：关键字 primitive，UDP 名称，I/O 声明，寄存器声明（如果是时序逻辑），初始化块和真值表 (table)。如果是 ANSI 格式的头，初始化可以在头里就完成。但是本书不会这样做。对于 table 来说，组合逻辑和时序逻辑的 UDP 有所区别。

为了简单并且仿真速度快，UDP 不允许包含延迟，高阻 (z) 和双向 (inout) 端口。如果有高阻 (z) 被送给 UDP 的端口，在 UDP 内部会被当做不定态 (x) 来处理。如果想用库里的元件来搭出 UDP 的模型可能会很困难，因为库中的元件往往和特定的工艺相关的，不具有一般性。

对于组合逻辑 UDP 来说，真值表的典型格式如下：

(inputs in declared order) : (output);

例如，三输入与门的真值表如下：

```
table
...
1 0 0 : 0;
1 1 1 : 1;
1 1 x : x;
...
endtable
```

当然, Verilog 已经有了多输入的与门原语。一个更实用的 UDP 与或逻辑可以写成如下:

```
primitive u2AndOr(output uZ, input uA1, uA2, uOr);
//
// Models uZ = (uA1 & uA2) | uOr.
//
table
// Output on right; inputs in declared order:
// and'ed inputs      or'ed input
//   uA1 uA2           uOr       uZ
//   0   0             0        : 0;
//   1   0             0        : 0;
//   0   1             0        : 0;
//   1   1             ?        : 1;
//   ?   ?             1        : 1;
endtable
endprimitive
```

术语解释: 逻辑门通常已经被包含在 ASIC 库中。不管是用 UDP 来实现它们的, 它们总是用它们的功能和输入的个数来命名的。A 代表与, O 代表或, I 代表反相 (inversion), 等等。统一的命名规则使得维护这个库变得更加方便。例如, 一个单元实现的功能如下: ($A \& B$)||C, 则它会被命名为 AO21; 如果它的功能是: $!((A \& B) \& (C \& D \& E))$, 则它有可能会被命名为 AOI23。

对于时序逻辑的 UDP 来说, 它的结果被冒号分成两列。左边的那一列, 也就是两侧都被冒号围起来的那一列, 它的取值代表当前的状态; 右边的那一列, 它的取值代表在当前状态和输入的条件下, 下一个状态的取值。

例如, 下面的 UDP 用来表述一个寄存器。

```
primitive uFF(output reg uQ, input uD, uClk, uRst);
//
initial uQ = 1'b0; // Not same as a module initial block.
//
table
// Output on right; inputs in declared order:
// current next
// uD uClk uRst  uQ
//   0 (01) 0 : ? : 0 ; // Clock in 0
//   1 (01) 0 : ? : 1 ; // Clock in 1
//   0 (0?) 0 : 0 : 0 ; // Default to keep same 0
//   1 (0?) 0 : 1 : 1 ; // Default to keep same 1
// Unclocked:
//   ? (1?) 0 : ? : - ; // Ignore negedge.
//   (??) ? 0 : ? : - ; // Retain state.
// Reset asserted:
//   ? ? (01): ? : 0 ; // Posedge reset
//   ? (??) 1 : ? : 0 ; // Ignore clock edge
//   (??) ? 1 : ? : 0 ; // Ignore clock state
//   ? ? 1 : ? : 0 ; // Ignore clock state
endtable
endprimitive
```

虽然不是必需的, 但 UDP 真值表里的每一行的内容都应该用空格或 table 隔开。过去, UDP 设计被保存在打孔的卡片 (Hollerith Card) 上, 工作站和工程师通过读卡片来解析空格是一个

很费时间的工作。现在的设计条件不同了，为了提高代码的可阅读性，请不要吝惜空格。UDP的每一行都以分号“;”结束。

时序逻辑UDP中的边沿敏感是用边沿的两个状态来描述的。这两个状态包含在圆括号中，左边的是初始状态，例如，(01)表示上升沿，(10)表示下降沿。真值表的每一行只允许有一个边沿。如果某个输入会影响到电平和边沿，先计算边沿的变化，再计算电平的变化。这意味着发生冲突时最终起作用的是电平的运算结果。和case表达式里的一样，“?”表示不关心这个输入取值。“-”表示输出取值不改变。

UDP中的真值表一定要覆盖到所有的情况，否则仿真工具从真值表查找出的结果就有可能不正确。由于需要考虑到边沿的每种情况，这条规则使得时序逻辑UDP的真值表变得很复杂。

如果输出需要考虑到“x”，那么上面的 UDP 设计还需要再增加几行。

和 Verilog 内建的基本门一样（与门，或门等），在例化 UDP 时可以不写出实例名。例化的时候可以包含延迟表达式。

下面我们总结有关 UDP 的知识点。

UDP 是基于查找表的原语。关键字是 primitive。

和模块属于同一设计层次。

优点：

- 例化时可以不指定例化名。
- 例化时可以指定延迟。
- 编译和仿真的速度快。

限制：

- 只允许一个 1 比特的输出。
- 输入位宽必须为 1 比特（可以多至 9 个输入）。
- 不支持时序和参数的声明。
- 内部不再有 specify 或其他的块。
- 不支持“z”和双向端口。

在 VLSI 设计中不使用 UDP。

不可综合。

有时会被用来开发 Verilog 的仿真库。

14.1.2 延迟的悲观处理原则

下面来讨论 Verilog 的延迟。首先我们需要理解功能和时序这两个互相联系又不相同的概念。

功能边沿。当在任何其他电平值（包括“x”）之后是“1”，则这是一个功能上升沿。always 的事件表达式和其他的边沿表达式都会把这种情况视为功能上的上升沿事件。类似地，如果结果变成了“0”，则在功能上被视为下降沿，被当做是下降沿事件。

时序边沿。当时序表达式 (rise, fall) 用来描述 “0” 到 “1”的变化时，上升沿的延迟是 rise。如果发生了 “1” 到 “0”的变化，则下降沿的延迟是 fall。但是，如果变化涉及到了 “x” 或 “z”，则没有这种对应关系。

当发生了从其他电平到 “x”的变化时，无论是表达式 (rise, fall) 还是表达式 (rise, fall, to_z) 的上升沿或下降沿都是没有意义的。这时，仿真器会自动把延迟设为最小的延迟时间。这种快速进入不确定状态的处理延长了处于不确定态的时间，因此被称为关于 1 或 0 的硬件值的“悲观处理”。

如果发生了从 “x” 到其他电平值的变化，仿真工具会自动使用最大的延迟时间，这也是悲观处理原则的一种体现。

Thomas and Moorby (2002) 的 6.5.2 节中的表 6.6 也详细解释了延迟的悲观处理原则。

14.1.3 门级时序三参数

在第 7 章中，我们学到了指定输出强度的办法。通过圆括号中加入一到两个强度关键字来指定门级输出的强度。例如下面这个 NMOS 的或门：

```
or (strong0, weak1) or.01(out_or, in1, in2, in3, in4);
```

当指定延迟时，也是类似的写法，除非延迟值跟在 “#” 符号的后面：

```
or #2 or.01(out_or, in1, in2, in3, in4);
```

如果只有一个延迟参数，圆括号可以不写。如果既有强度又有延迟，则强度放在延迟的左边：

```
or (strong0, weak1) #(2,1) or.01(out_or, in1, in2, in3, in4);
```

我们看到，圆括号里可以有两个或三个延迟参数。具体的规则如下所示：

一个参数	输出变化都被赋予同样的延迟
# inst_name(...);	
两个参数	上升沿使用参数 tr；下降沿使用参数 tf
#(tr, tf) inst_name(...);	
三个参数	前两个参数的用法和两个参数一样。第三个参数指变化到 “z”的延迟时间
#(tr, tf, tz) inst_name(...);	

如上表所示，从 “0” 到 “1”的变化使用 rise 参数；从 “1” 到 “0”的变化使用 fall 参数。当变化涉及到 “x” 或 “z” 时，仿真工具使用悲观原则来决定取值，也就是说，对于有两个参数且门支持 “z” 输出的情况，进入 “z”的延迟是两个参数里较小的一个；而对于有两个或三个参数的情况来说，从 “z” 或 “x”的恢复到 “0” 或 “1”的延迟是参数里最大的那个，而变成 “x”的延迟仍然是最小的那个。

深入讨论延迟。过去设计电路时，往往都是通过在电路板上放置小规模集成电路和分离的被动器件来完成的。逻辑仿真工具在对整个电路设计进行仿真时，需要考虑到电路板上不同的温度、电压、工艺等条件，因此，对板上每颗芯片的最小延迟和最大延迟是独立计算的，算出的时间差异由任意同时估计的最小、最大延时允许的位置 (= 悲观处理) 赋值 “x” 进行仿真，如图 14.1 所示。

现在的 VLSI 或深亚微米 (deep-submicron) 的设计里，输出都是通过时钟锁存和时钟同步的。仿真时延迟悲观处理的原则并不是很有用，而且整个芯片往往都工作在一个条件下，

比如说，快、典型、慢，因此，每次只须对一种条件进行仿真即可。而同时计算最小-最大延迟通常在进行静态时序分析时才会用到，后面的章节会讲到相关的内容。

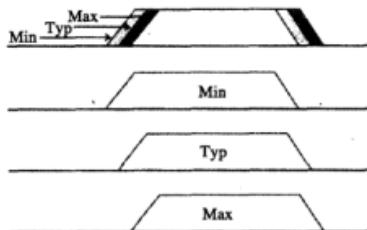


图 14.1 最上面的那个波形说明了板级仿真边沿是不确定的；而对一颗芯片进行仿真，例如下面那三个图，它们的延迟是确定的

在目前的VLSI设计中，仿真工具不会被用来选择电路工作时的条件，这些条件包括：最佳条件、典型条件和最差条件，用什么条件由设计工程师来决定。库里用来综合的门器件包含了温度、电压、工艺等信息，仿真工具需要从多种条件里选择需要的一种作为门器件的延迟计算条件。由于一次仿真只能选择最小延迟、典型延迟、最大延迟之中的一种，因此，如果要覆盖到所有的延迟情况，则需要多次仿真。VCS或其他的Verilog仿真工具都可以通过命令行来选择某一种特定的延迟。典型延迟是默认的延迟。

对一些超大规模的设计来说（90 nm或更先进的工艺），为了计算时序的不确定性，可能有在一个仿真里用多种延迟条件的需求。一个例子是当芯片进行了功能或工作模式的切换之后，温度可能会发生变化，从而导致延迟发生了变化。类似的情况越多，芯片温度变化可能越频繁。在这样的情况下，类似以前那种对整个电路板上的器件进行仿真的做法可能再次变得有必要了。

给门级输出指定三种延迟（triplet）只须把上表中 tr, tf 和 tz 三个参数都分别替换成 t_min : t_typ : t_max 即可，参数之间用冒号隔开。

上面或门的例子被改成这样：

```
or #(1:2:3) or_01(out.or, in1, in2, in3, in4);
```

类似地，如果用 bufif1 例化出一个低关断时间的实例，应该这样来指定延迟：

```
bufif1 #(1:3:4, 1:2:4, 6:7:8) triBuf_2057(OutBit, InBit, CtlBit);
```

没有规则强制规定三种延迟的顺序，但一般都指定 min:typ:max 的顺序为 $\text{min} \geq \text{typ} > \text{max}$ 。

Verilog 中涉及延迟计算的功能不只是这些，我们还将在后面的课程里讲授关于库元件内部延迟的内容。

14.1.4 开关级元件

在第7章中，我们介绍了Verilog里强度的概念和线的不同类型。在第10章中，我们继续学习了线的类型和Verilog内建的门类型。现在我们来学习开关级（switch-level）的建模。开关级里的门器件都是由最基本的元件（晶体管）组成，并且都是用开和关来描述的。

为了进行开关级的建模，我们需要开关级的原语，包含：MOS, CMOS, bidirectional 和 source 开关。

下面是开关级原语的列表：

MOS switches:

nmos, rrmos (like bufif1)
pmos, rpmos (like bufif0)
cmos, rcmos

Bidirectional pass switches:

tran, rtran
tranif1, rtranif1
tranif0, rtranif0

Switch-level net:

trireg

Power sources:

pullup, pulldown

MOS 开关。 MOS 是金属氧化物半导体 (Metal Oxide Silicon) 的简称，它是半导体行业最重要的工艺之一。MOS 管的源级提供能量，从而可以通过控制源级切断或打开输出的电流。

对 P 型 MOS 管 (PMOS) 来说，加在沟道上的电压为负时，空穴增多，沟道导通。同理，对 N 型 MOS 管 (NMOS) 来说，加在沟道上的电压为正时，电子增多，沟道导通。由于空穴的数量比电子少。因此，在 CMOS 工艺里，P 管的那部分总是要比 N 管的那部分大一些，从而使得两部分的导电能力接近。通常，P 管接电源，N 管接地。由于器件的特性，CMOS 工艺主要用来实现与非、非等，而不是与、buffer 等。

MOS 的原语包 NMOS, PMOS, RNMOS 和 RPMOS。这里的 R 是 “Resistive” 的缩写，带 R 的器件的电阻要更高一些。

Verilog 里所有 MOS 器件的功能和之前我们学到的 bufif1 (NMOS) 或 bufif0 (PMOS) 是一样的。只是带 R 的器件会降低输入的强度而已。两种强度不会被带 R 的器件降低，分别是 small 和 highz。请看第 7 章中的 Verilog 强度表。下面是强度转换的规则。Thomas and Moorby (2002) 的 10.2.4 节和 IEEE Std 1364 的 7.12 节都给出了这个规则。

MOS 强度规则			
强 度		强度关键字	
In	Out	In	Out
supply	pull	supply0/1	pull0/1
strong	pull	strong0/1	pull0/1
pull	weak	pull0/1	weak0/1
large cap	medium cap	large0/1	medium0/1
weak	medium cap	weak0/1	medium0/1
medium cap	small cap	medium0/1	small0/1
small cap	small cap	small0/1	small0/1
highz	highz	highz0/1	highz0/1

CMOS 开关。 CMOS 是互补金属氧化半导体 (Complementary Metal Oxide Silicon) 的简称。CMOS 的功能由一对 NMOS 和 PMOS 组合而成，而在实际的芯片里，CMOS 的物理结构也是这样的。和 CMOS 由两个门器件组成一样，CMOS 开关也有两个输入。显然，RCMOS 开关是由一对 RNMOS 开关和 RPMOS 开关组成的。

Verilog 把 1 认为是高电平，0 认为是低电平。因此，N 管在输入为 1 时导通，P 管在输入为 0 时导通。

图 14.2 是由两个 MOS 管组成一个 CMOS 开关的示意图。

在 Verilog 里, NMOS 和 PMOS 原语将输出拉高至“1”和将输出拉低至“0”的强度是一样的。而实际上, P 管和 N 管产生的这两个逻辑电平的强度是不一样的。对 N 管来说, 它产生的 0 的强度应该比 1 的强度高; 对 P 管来说正好相反。

Thomas and Moorby (2002) 的 10.2.1 节详细的给出了静态 RAM 单元的模型, 在 10.1 节里给出了移位寄存器的模型。但是却没有指出建立这种模型的目的, 也没有指出应该什么时候用高精度的 SPICE 对一个器件来建模。

CMOS 的性能比单独用 PMOS 和 NMOS 要好很多。因此, 它们被大量地运用在现在的设计中。

现在让我们来思考如果利用 P 管和 N 管实现一个取反的缓冲器。利用开关级的模型, 我们可以很容易地实现这一点。把 PMOS 放在 supply1 里, 再把 NMOS 放在 supply0 里, 这样就完成了 Verilog 中的取反 (not)。

可能有读者会想, 如果把图 14.3 里的 P 管和 N 管的上下位置颠倒过来, 这个器件不就是一个不取反的缓冲了吗? 逻辑上说是没有问题的, 最后的输出也的确没有被取反。但是, 在实际的芯片里, P 管在 supply0 里的工作速度会非常的慢, 效率也很低。把 N 管放到 supply1 里效果也一样。因此, 在实际中, 非反相的缓冲从来不用 CMOS 工艺来实现。如果要用 CMOS 工艺实现非反相的缓冲, 则会用两个反相器来实现, 大的反相器放在输入端, 小的反相器放在输出端。

CMOS 开关不会减小输入的强度。与 bufifx 不同, CMOS 开关有两个控制输入。它的端口声明如下:

```
(r)cmos optional_inst_name ( out, in, N.ctl, P.ctl)
```

上面的两个控制端只要有一个是处于打开状态, 这个 CMOS 管就是打开的。因此, 有两个控制端的目的并不是为了添加新的功能, 而是出于连线的需要。

双向传输开关。这一类原语指的是传输门, 因此包含 tran(参见图 14.4), tranif1 和 tranif0, 与之对应的高阻传输门分别是 rtran, rtranif1, rtranif0。它们都不支持延迟赋值。和 tireg 不一样, 它们不存储当前的状态。它们的作用仅仅是把一个信号从一端送到另一端, 用来描述晶体管模型里的连线。



图 14.4 传输门电路符号

除了有两个 inout 端口, 例化 tran 类型传输门的格式和 buf 一样。tranifx 类型传输门有两个 inout 端口和一个控制端口。因此, 除了两个 inout 端口, 它的结构和 bufifx 比较类似。在芯片进行布局布线时可能会用到传输门, 补充学习里还有关于传输门其他的一些内容。

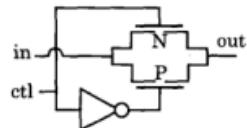


图 14.2 NMOS 和 PMOS 并联组成 CMOS bufif1

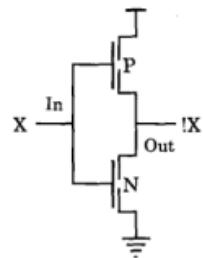


图 14.3 NMOS 和 PMOS 串联组成 CMOS 的非门

源开关。有两种 source 开关：pullup 和 pulldown。这两种开关都只有一个输出。pullup 以 source1 的强度把线型拉高；pulldown 以 source0 的强度把线型拉低，鉴于 Verilog 中对晶体管在开关级的建模通常假定为工作在增强模式（通常关闭）下，这类开关被假定为消耗模式（通常开启）下的大型晶体管，或者直接连在 IC 电源或地上。

14.1.5 开关级的线型:trireg

IEEE 1364 的 7.13.2 节和 7.14 节都描述了 trireg 的功能。用一句话来说，trireg 是用来储存当前逻辑状态的有时效性的寄存器。它的部分特性和电容类似。

trireg 独特的地方在于它能够使用 small, medium 和 large 强度。对 trireg 来说，这些强度意味着进入高阻状态（“z”）的时间，也就是电容的容量。如果 trireg 受高阻 “z” 驱动，它不会进入高阻态，而会一直保持之前非高阻的值。只有在 trireg 被指定了延迟的情况下，非高阻的值会在延迟时间过了之后从 “1” 或 “0” 变成 “x”。

trireg 的强度用于确定在多个 trireg 冲突时 “x”的延时。这是充电强度：small, medium 和 large 唯一的用途。若强度相等，采用仿真的悲观原则来确定 trireg 衰变到 “x”的延时，就和确定其他时序冲突的方式类似。

当声明 trireg 类型时，必须要指定它的延迟。指定延迟的格式和我们前面学到的指定三个延迟参数的格式不一样。如果指定的延迟有三种取值，前两个参数的含义是上升时间和下降时间，第三个参数的含义是到 “x” 而不是到 “z”的时间。trireg 类型不能进入状态 “z”，但它的初始状态可以为 “z”。如果 trireg 受 “z” 驱动，当延迟时候到了之后，输出会变成 “x”。

如果没有指定 trireg 类型的延迟，trireg 将会一直维持在上次的状态（“1”，“0” 或 “x”）。下面是声明一个 trireg 类型的例子：

```
trireg (medium) #(3, 3, 10) medCap1;
```

可以用连续赋值的形式和 trireg 的线型相连，也可以通过开关级实例的端口映射来相连。

下面是使用 trireg 线型的一个例子：

```
pullup(vdd);
trireg (small) #(3,3,10) TriS;
trireg (medium) #(6,7,30) TriM;
trireg (large) #(15,16,50) TriL;
// Pass transistor network:
tran (TriM, TriS); // left always wins.
tran (TriM, TriL); // right always wins.
// NMOS network:
rnmos #1 (TriM, TriS, vdd); // input has no effect.
rnmos #1 (TriS, TriL, vdd); // input controls output.
rnmos #1 (TriM, TriL, vdd); // Contention on output.
```

14.2 练习 17：元件

在目录 Lab17 下完成这个练习。

练习步骤

很多 VLSI 的仿真工具不完全支持对开关级 Verilog 模型的仿真。如果你用的是仿真工具 Silos，除了高阻值的原语（以 r 开头），其他的特性 Silos 基本都支持。

第1步。组合逻辑 UDP。在一个名为 AndOr2Not4 的模块里设计这样一个 UDP 功能: $X = (\neg a \mid \neg b) \& (\neg c \mid \neg d)$, 从 a 到 d 都是输入, X 是输出。

小规模分立元件的数据手册或大规模集成电路的库可能会有包含类似功能的器件。

建议: 在注释里把查找表每一列的名字都列出来, 按这个顺序完成整个查找表。这样会减少出错的可能。

在 testbench 里例化你写的这个元件, 验证它的功能。

第2步。时序逻辑 UDP。设计一个名为 AndLatch 的 UDP, 它是个锁存器, 有两个数据输入, 在输出锁存数据之前两个输入数据先经过了与。

在模块中例化这个锁存器, 并仿真验证它的功能。

第3步。反相器的开关级模型。新建一个名为 Nottingham 的模块, 它有一个 1 比特的输入和两个 1 比特的输出。把 PMOS 和 NMOS 原语组合成图 14.3 的取反的模型。实际上, 这就是 CMOS 中的反相器。把这个反相器的输出连至 Nottingham 模块的任意一个输出端口上。

再直接用 Verilog 的 not 功能例化一个反相器, 把它的输出连至 Nottingham 模块的另外一个输出端口上。这样, 就可以在仿真时比较两种反相器对同一种输入的结果。

你可以这样来检查这两个结果: 把两个输出都送到一个 xor 门里。如果 xor 输出了“1”, 则说明两个反相器的输出不一致。

第4步。CMOS 控制输入。给上一步设计的模块增加第三个输出端口, 并把它连在 CMOS 元件的输入上。在 testbench 中声明新的连线, 用来连接新的端口。

按下表的各种条件来进行仿真。你能预测在各种条件下的仿真结果吗(下表中的“unconn”即是高阻 “z”)?

CMOS 真值表			
out	in	n-ctl	p-ctl
1	1	1	1
1	1	1	0
1	0	0	0
1	0	0	1
0	1	1	1
0	1	1	0
0	0	0	0
0	0	0	1
1	x	x	x
1	x	x	0
1	x	0	1
1	1	unconn	unconn
1	unconn	0	0

第5步。用传输门组成选通器。传输门最简单的应该是组成选通器 (mux)。控制输入使得一次只有一个传输门被打开, 从而输入的电平传到了输出。这个输出作为整个组成选通器的输出。而此时另外一个传输门的输出值可能是不定态。

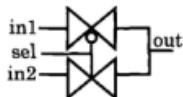


图 14.5 用 tranif 组成的双输入 mux

对于一个 2 输入的选通器来说，只需要把 tranif1 和 tranif0 的输出连在一起，再把它们的控制输入连在一起就可以了。

图 14.5 所示电路的对应 Verilog 代码如下：

```
tranif0 UpperTran(Out, In1, Sel);
tranif1 LowerTran(Out, In2, Sel);
```

只用传输门和连线，设计一个名为 TranMux4 的 4 输入选通器。仿真验证你的设计。

在验证了这个设计之后，用 rtranifx 替换 tranifx 再仿真（Silos 可能不能仿真出正确的结果）。

第 6 步。与非门和或非门。新建一个名为 nand_nor 的模块，它有三个输入，两个输出。两个输出分别叫 nand 和 nor。按照图 14.6 所示的电路图设计开关级的 3 输入与非门和 3 输入或非门的模型。仿真验证你的设计（参见图 14.7）。

仿真通过之后，在输出端加上 CMOS 的反相器，使得它们的功能变成与和或。

第 7 步。 trireg 脉冲过滤。新建一个名为 RCnet 的模块，用 trireg 开关来实现图 14.8 所示的 RC 网络。

用 RNMOS 表示上图中的电阻，左边是输入，右边是输出。强度为 large 的 trireg 的延迟为 #(15, 15, 50)，强度为 medium 的 trireg 的延迟为 #(7, 7, 30)，强度为 small 的 trireg 的延迟为 #(3, 3, 10)。仿真并观察波形（参见图 14.9）。

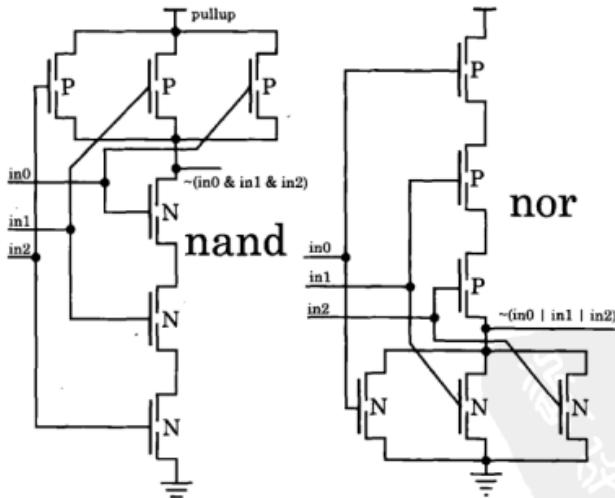


图 14.6 3 输入的 CMOS 与非门和或非门



图 14.7 仿真的波形

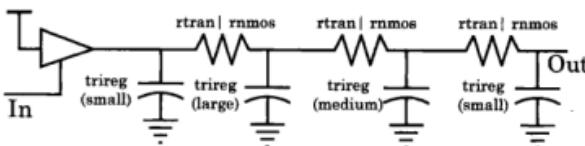


图 14.8 用来进行数字仿真的 RC 网络

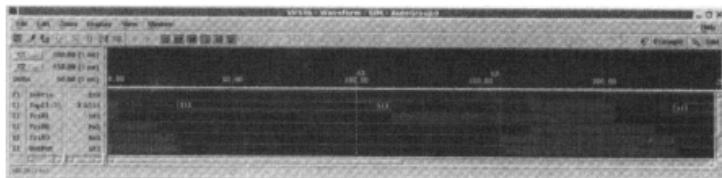


图 14.9 加入了 RNMOS 之后的仿真波形

用 rtran 替换 RNMOS，再仿真（参见图 14.10）。

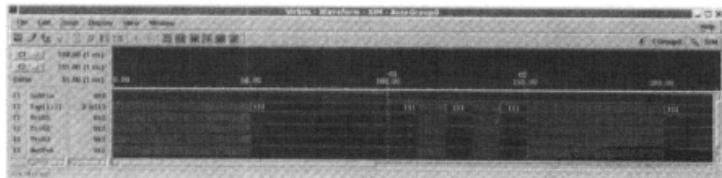


图 14.10 加入了 rtran 之后的仿真波形

14.2.1 练习后的思考

当给 trireg 的输出指定延迟时，它的三个延迟参数是什么意思？

可以用 rtran 来仿真有延迟的情况吗？

14.2.2 补充学习

阅读 Thomas and Moorby (2002) 的 6.5.2 节关于延迟冲突和悲观处理原则的内容。

阅读 Thomas and Moorby (2002) 的 6.5.3 节和 6.5.4 节关于时间单位和时序三参数的内容。

阅读 Thomas and Moorby (2002) 第10章关于开关级建模的内容。不用去管“minisimulation”那部分代码。

阅读 Palnitkar (2003) (可选)

阅读 Palnitkar (2003)。

5.2 节关于门级延迟的基础知识。

第 11 章关于开关级建模的内容。

第 12 章关于 UDP 的内容。尤其要注意 12.5 节里 UDP 和模块的特性对比。做 12.7 节的第一题(a): 2 输入选通器。把你的答案和 Palnitkar 书中光盘的答案相比。

11.2.3 节关于开关级锁存器或寄存器的内容。在书中的光盘里有一个名叫 cff.v 的寄存器模型，拿它仿真并观察结果。



第 15 章 参数和层次

15.1 参数的类型与模块连接

15.1.1 参数的特点

- 参数的默认属性是无符号的整型常数

```
parameter Name = value;
```

- 也可以是符号数（但是很多工具不支持）

```
parameter signed Name = -value;
```

- 也可以声明成实型（不可综合，而且很多工具也不支持）

```
parameter real Name = float_value;
```

- 参数可以指定位宽

```
Parameter [6:0] Name = 7_bits_of_value;
```

- 参数声明可以放在模块的任何一个地方，但是 localparam 类型只能在模块主体中声明

```
localparam Name = value;
```

- 必须在声明参数的地方给参数赋值。参数默认的位宽总是足够宽的。

15.1.2 参数声明的 ANSI 格式

```
module ModuleName #(parameter Name1 = value1, ...0) (ports decs);
```

我们推荐读者使用 ANSI 格式的参数声明，这种格式按名称传递参数的具体值。如下例所示：

```
// Declaration:  
module ALU #(parameter DataHiBit=31, OutDelay=5, RegDelay=6)  
    (output[DataHiBit:0] OutBus, ...);  
  
// Instantiation:  
ALU #(DataHiBit(63), .RegDelay(7)) // OutDelay gets the default.  
ALU1 (.OutBus(ResultWire), ...);  
...
```

15.1.3 传统的声明格式

```
module ModuleName (PortName1, PortName2, OutPortName1, ...);  
    parameter Name1 = value1; ...  
    direction[Name1:0] PortName1; // direction = output, input, inout.  
    direction[range1] PortName2; ...  
    reg[range1] OutPortName1; ... // output assigned procedurally.  
    ...
```

和 ANSI 格式一样，传统的模块头声明以分号结束。但是，模块里参数声明的位置是任意的，这样的做法使得模块间的接口容易产生混淆并发生错误。

15.1.4 例化格式

ANSI 格式和传统的例化格式是一样的。

15.1.4.1 按名称重定义实例的参数

```
ModuleName #( . ParamName1( value1 ), . ParamName2( value2 ) ... )
  moduleInstName ( . PortName1( NetName1 ), ... );
```

15.1.4.2 按位置重定义实例的参数

```
ModuleName #( value1, value2, ... )
  moduleInstName ( . PortName1( NetName1 ), ... );
```

我们不推荐按位置来重定义实例的参数。

参数用起来就像是无符号的 reg 常数。当参数的值赋给一个变量时，或参数被用在一个表达式当中，它的位宽和类型会自动与期望匹配。不过，也可以专门指定参数的位宽。比如说，parameter [7:0] CountInit = 8'hff; 这个例子就指定了参数的位宽，可以提醒设计者这个参数要赋值的变量的位宽。

参数和变量一样，也可以被声明成有符号数 signed。如果一个算术表达式所有的操作数都是符号数，那么这个算术表达式就是一个有符号的运算。虽然参数的数值可以在例化时被重定义，但是参数的位宽和符号是不允许重定义的。

如下例所示：

```
// Note: 377 = 12'h179; -377 = 12'he87.
parameter signed[31:0] mul.coeff = -120*Pi; // Gets -376.9911 = -377.
//
// If the next were overridden by -120*Pi, it would get 32'hffff.fe87:
parameter signed[31:0] div.coeff = 32'b0000.0179;
```

在例化模块时才重定义参数的数值是唯一被推荐用来重定义参数数值的方式。

注：和 Thomas and Moorby (2002) 或 Palnitkar (2003) 同时发行的 Silos demo 仿真器可能不支持有符号的参数。

15.1.5 ANSI 标准端口和参数选项

按名称和位置来传递参数都符合 ANSI 的标准，但是不能同时使用这两种办法。当按位置来传递参数时（再次说明不推荐使用这种方式），所有的参数必须按从左到右的顺序全部排列齐。如下例所示：

```
module ALU #(parameter DataHiBit=31, OutDelay=5, RegDelay=6)
  (output[DataHiBit:0] OutBus, ...);

  ALU   #(31,5,8) // Must supply first two to change third one.
  ALU1 (.OutBus(ResultWire), ...);

  ALU   #(DataHiBit(31),5,8) // Illegal.
  ALU2 (.OutBus(ResultWire[63:32]), ...);
```

15.1.6 传统格式的模块头格式和选项

在这里，我们回头看一下 Verilog-1995 的模块头的格式。这个格式是 Kernighan 和 Ritchie 在早期的 ANSI C 的函数头格式（“K&R”C）的基础之上发展出来的。虽然这种格式现在已经不再使用了，但是很多自动化工具，例如网表生成器或文件转换器等还在使用这种格式。基于这样的原因，理解这种格式还是很重要的，而且我们也可能会遇到需要手动修改网表来解决问题的情况。

模块的头定义里包含了端口的名称。紧接着，参数被声明，然后是端口的输入输出方向和位宽。在这之后，输入输出端口的类型被指定。通常，这是指模块的输出是由寄存器驱动还是直接连在组合逻辑的输出上。我们看到，和 ANSI 格式一样，参数值可以用来改变输入输出端口的位宽。

如下例所示：

```
module ALU (OutBus, InBus, Clock); // Parens & contents optional.  
parameter DataHiBit=31, OutDelay=5, RegDelay=6;  
output [DataHiBit:0] OutBus;  
input ...  
reg [DataHiBit:0] OutBus;  
...
```

ANSI 格式要更新一些，它可以避免重复的名称并且能减少出错的可能，因此我们应该使用 ANSI 格式。

无论是使用 ANSI 格式还是使用传统格式，在例化的时候重定义参数的效果是一样的。

15.1.7 defparam

在 Verilog 里还有一种语法结构可以修改任何一个模块里的参数值，而不用去管这个模块是不是和当前设计相关。它的语法如下：

```
defparam hierarchical_path.to.parameter = new.value;
```

这是一种很危险的结构，在将来的 Verilog 语言标准中，它很有可能被去掉。它使得任意一个模块中的参数值可以轻易的被修改。

defparam 是 localparam 存在的主要理由是：除了不能被 defparam 修改参数值，localparam 和 parameter 的作用是一样的。但是，localparam 不允许在 ANSI 格式的模块头中使用，因为它不能够被重新定义。

希望读者在设计里绝对不要使用 defparam。就像 goto 语句，或分层的相互引用，它们带来的麻烦会大于它们带来的好处。

15.2 练习 18：连线

在 Lab18 的目录下完成这个练习。

练习步骤

第 1 步。传统格式端口映射。把模块命名为 nonANSItop，按传统的端口映射格式重写下面的模块。

```

module ANSItop #(parameter A=1, B=3, parameter signed[4:1] List=4'b1010)
  (output [3:0] BusOut, output ClockOut
   , input [3:0] BusIn, input ClockIn
   , input [1:0] Select
  );
  reg ClockOutReg;
  assign #(2,3) ClockOut = ClockOutReg;
  ...
endmodule

```

给这个模块增加任意的功能。

这一步之后，会有两个模块，分别在文件 ANSItop.v 和 nonANSItop.v 中，下一步会在仿真器中编译。

第2步。参数重定义。建立一个新文件 ParamOver.v，在新的模块 ParamOver 中，分别实例化第1步中建立的 ANSItop 和 nonANSItop 各两次。对每一个 ANSItop 和 nonANSItop 来说，把 B 重定义成 20，把 List 重定义为 -2。

在仿真器中查看设计的层次结构（参见图 15.1）。QuestaSim 可以被用来查看参数的具体数值，而 VCS 不支持显示参数值。为了便于查看，在 initial 块里把所有的参数按位宽分别赋给一个 64 bit 宽的寄存器的对应比特，并且在仿真器中观察 0 时刻时寄存器的值。由于所有的参数按比特被拼到了一起，按波形查看它的结果会很方便。这个寄存器在 initial 块里没有做任何事情，因此在综合的时候，它会被综合工具忽略。

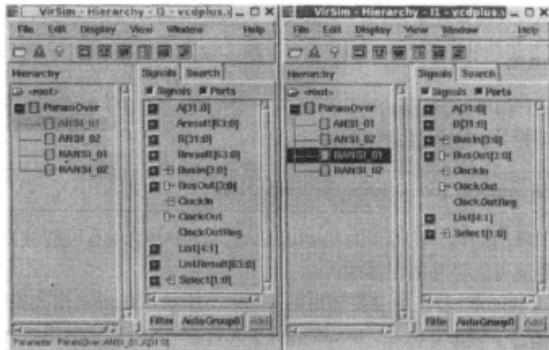


图 15.1 ParamOver 的层次结构

问题：第1步中的 List 是一个有符号的参数，它的默认值是什么？

第3步。参数位宽重定义。再次例化 ANSItop，按位置方式把 A 重定义为 8'hab。通过 initial 块里定义的那个 64 比特寄存器，在仿真器中观察结果。

第4步。参数类型重定义再次例化 ANSItop，这次把 A 和 B 定义成 signed，但是初始值保持不变。

按名称方式把 A 重定义为 8'hab，把 B 重定义为 -120*π（参见图 15.2）。

通过 initial 块里定义的那个 64 比特寄存器，在仿真器中观察结果。把显示格式设成补码 (2's complement)，观察有符号数的取值。

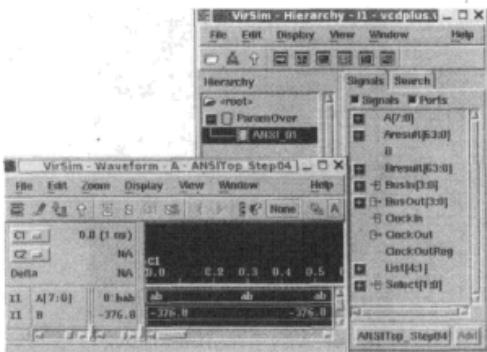


图 15.2 以补码显示出来的参数值

第 5 步。 `define 和顺序的问题。HierDefine 这个练习的文件在 Lab18 下 HierDefine 子目录里。进入 HierDefine 目录找到如下文件：HierDefine.v、Level2.v、Level3.v 和 Level4.v。双向的数据总线在每一层都会被配置以适合不同层次的需求。这是个完整的数据流的例子，可以应用到树里叶变换上。

其层次结构如图 15.3 所示。

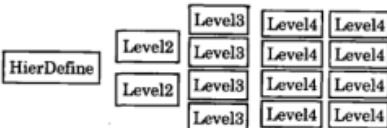


图 15.3 HierDefine 里实例的层次图

A. 建立一个以.vcs 结尾的文件，确保文件顺序能够让 VCS 正确地编译文件。文件中的“...”是省略的意思，读者需自行将其注释掉（参见图 15.4）。

如果.vcs 文件里的顺序和当前的不同会怎么样呢？如果 Level2 包含的宏定义 `define Wid 16 和 `define ResWid 16 有重复时又会怎么样呢？

B. 如果设计的位宽需要增加或减少，怎么利用 `define 来修改 HierDefine 中的参数，从而让我们能够方便、迅速的更改设计呢？

第 6 步。 参数传递的深度。假设有一个分为 4 层的设计叫做 HierParam，除了已经声明了参数而且没有用 `define 之外，其余和之前的 HierDefine 一模一样。

DataB 的位宽必须随层次关系减半，如图 15.5 所示。

为了简化问题，所有的模块声明都被放在了在 Lab18 目录下的 HierParam.v。

修改 HierParam.v，重新定义参数的默认值，使得只需修改一个参数，每一个子模块里 DataB 的位宽都能满足需求。不改变这个文件里参数默认的赋值。编译并在仿真器里观察文件层次后，检查你的结果（参见图 15.6）。

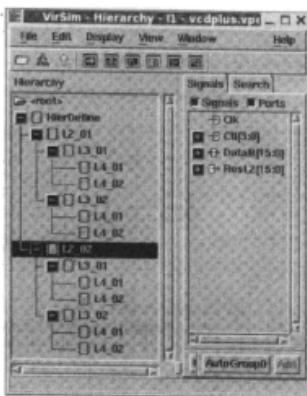


图 15.4 VCS 中 HierDefine 的层次

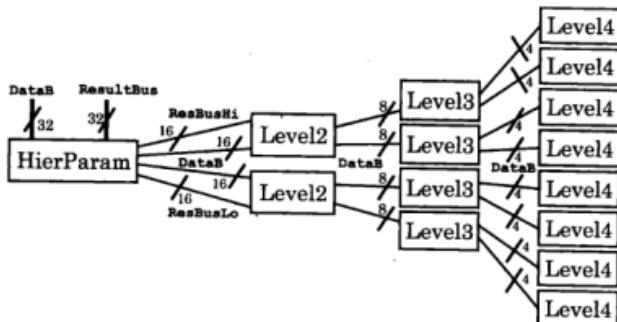


图 15.5 HierParam 实例层次的方框图

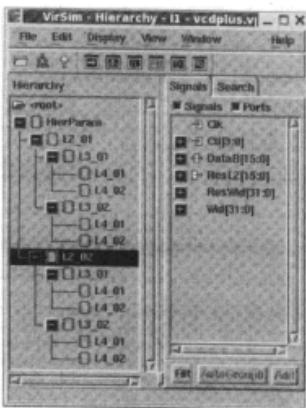


图 15.6 VCS 中 HierParam 的层次

15.2.1 练习后的思考

如果设计者需要同时给一个模块传递新的延迟值和新的参数值应该怎么办？

15.3 层次命名和设计划分

15.3.1 层次命名的引用

我们已经学习了关于层次的一些知识，主要是模块实例的层次和例化新实例时层次的命名。

编译器（仿真工具或综合工具）是靠层次来区分 Verilog 设计的。当设计一个子模块或者在验证它的时候，编译器总是会按层次关系把它和顶层设计联系起来。当顶层模块被例化到了更大的设计里去的时候，编译器会自动更新这个层次关系，并且把例化的设计和新的顶层设计联系起来。

因此，在层次引用时（A.B.C …），引用关系应当总是限定在当前模块内部（请回想第 12 章中讲述的 generate 语句），或限定在包含该子模块的模块中。由这样层次关系，设计者可以控制设计的主干和分支。这种引用关系是安全的。因此一个模块依赖于它的子模块来实现它的功能，修改了分支（即模块中的实例）意味着模块被重新设计过。在这个重新设计的过程中，层次命名可以修改。

如果引用关系没有被限定在当前的模块内部，则很有可能会出问题。假设有一个模块，它的功能是最基本的“与”，可能很多模块都会例化这个实现基本功能的模块。如果引用这个与门的时候用到了其他模块的层次关系（因为这个与门也被例化在其他的模块中），则很有可能因为引用时层次不对等原因导致例化出问题。一个典型的例子是器件库里的 Verilog 器件模型，任何设计都可能会调用这些模型，如果不按向下的层次关系来引用它，则很有可能引用出来的模块不能工作。

15.3.2 声明的有效区间

这里我们做一个有效区间的复习。在第 11 章和第 12 章中，我们曾经介绍过一些概念。

有效区间（scope），意味着一个范围，在 Verilog 里，可以理解成可见性。可见性有一个隐含的意思，那就是一个名字有可能被再次声明或再次定义。可见范围更大的名字会覆盖住可见范围小的名字。打个比方说：有两个不同的模块，每个模块内部都有一根叫 Clock 的线，但显然它们是不同的两根线。因为这两根线在模块的内部，它们的有效范围不会达到模块名外部，所以它们对这个两个模块来说，互相都是不可见的。

在一个过程块里（包括在 task 和 function 里），一个名称都必须在第一次使用之前先声明。但是有名块的声明可以摆放在模块中的任意位置。

以下是一些有效区间的区别：

- `define：没有实质的区间限制，只要从编译器按编译顺序读到这种宏定义后就开始有效（和文件的结构层次无关）。
- 模块名：当前模块里都有效。

- 模块例化名：我们可以例化出任意数量、不同层次的实例。但是最顶层的实例必须在一个模块中例化。
- 并行处理块名：包括一个模块内部的 initial 或 always 块。并行模块总是在当前的模块里有效，这意味着对并行块来说，没有层次的概念。然而，利用 generate 产生出来的块展开后，有可能会有层次。
- 序序处理块名：和所有 begin-end 包含的顺序处理块一样，task、function 这些对象都有可能和其他的块构成层次。
- 基本命名：包含变量（reg 类型、wire 类型等）和常量（parameter 类型、localparam 类型、specparam 类型等），它们都是有有效区间的，但是它们无法定义自己的有效区间。

其余的语法结构，例如表达式或延迟等，没有自己的名字，因此，它们没有有效区间的概念。

在这里，我们简要地学习一下 Verilog 的 config 块。config 有自己的名字，它可以是设计对象或某种特征的集合，但是它不完成设计功能。config 块可以存在于模块的任何一个位置。与其说它是一个有意义有层次的块，倒不如说它更像是 makefile，或者是系统的文件、目录。config 中的命名通常和外部库中的对象有关联。

15.3.3 设计的划分

通常，一个大的工程会被划分成多个较小的设计来降低设计的复杂度，而且多个较小的设计可以由多个工程师同时进行开发以加快开发进度。另外，一些 EDA 工具也要求对设计进行划分。

做设计划分（partition）时，最重要的一件事是设计接口。每一部分设计都会实现各自的功能，工程的其他部分都可能会调用这部分设计，因此，必须在制定设计规范时就考虑好此设计所有的应用。

接口应该有逻辑性并且直观，这样其他的工程师或设计者自己都能容易的理解接口的含义。在制定接口时，应该把诸如接口类型、位宽、时序和协议之列的问题都事先考虑清楚。当我们决定要修改接口的时候，一定要慎重，因为它可能会影响到系统里和它相关的所有模块。

在 Verilog 语言基础上发展出来的 System Verilog 语言把 interface 定义成一种语言结构。这种结构预先就定义了模块的头和相关的输入输出功能，设计者可以不经修改的把一种定义好的接口直接运用到多个模块上，避免了接口不会被频繁的修改。

对 Verilog（或 System Verilog）语言来说，设计的划分总是体现在模块上的。模块（module）是 Verilog 里能被单独编译的最小单位。如果重新划分了设计，则可能会产生新的模块，而其他部分的设计可以被保留下。因此，设计者只需集中精力设计和验证这个新的模块就可以了，从而提高了设计的效率。

下面是做设计划分时的一些基本原则。

时钟域。在划分设计的时候，首先应该考虑到划分时钟域。使用不同时钟的逻辑或和省电模式相关的逻辑都应该考虑被分别设计在不同的模块中。当有跨时钟的应用时，应该设计相关的时钟同步逻辑（下面会讲到）。

电压岛 (Voltage island)。工作电压不同的逻辑应该被划分在不同的部分中。一般我们使用的基本门器件都来自于不同提供商提供的不同的元件库。虽然，现在一些 CMOS 库能够工作在一个较宽的电压范围里，比如说 3~5 V，但最佳工作状态需要的电压总是在一个较窄的范围里的。按电压划分了设计之后，电压转换器件的选取也变得更容易了。

电压转换器件放在它们自己所在的区域里，或者被转换的电压区域里。

输出缓存。实际复杂结构的各个部分都应该被时钟寄存、输出锁存，使得将它的内部时序与其他部分分离开，成为独立的结构。

时序控制。对于一个关键的跨模块的控制信号（例如读 memory 时的等待指示），往往会在端口利用寄存器等调整它的时序，或用时钟的另一个沿来进行触发。

IP 重用。现在的超大规模集成电路设计者往往会购买商业 IP (Intellectual Property) 来实现自己 IC 中的一部分功能。最典型的例子是微处理器的核和 memory 单元。应该把这些 IP 独立划分。

测试考虑。做设计划分的时候还应该考虑到内部扫描和边界扫描。总之，设计的时候应该保证做测试时能够监测到关键信号。用带扫描端的寄存器替换基本的寄存器会给测试带来很大的好处，同时也不会影响性能。也就是说，设计者要保证在替换了扫描器件后，不用重新设计仿真矢量。

综合考虑。通常，综合工具的逻辑优化功能对于特定规模的逻辑设计可以达到最好的优化效果，这个规模大概是 300~50 000 个晶体管这个级别。考虑到不同的库，这个级别大致相当于 30~5000 元件，大概是 75~15 000 个门。如果一个功能块过大，综合优化的时候可能会过长，延长了调试的周期；如果一个功能块太小，优化的余地可能会较小，有可能综合优化起不到什么效果。做设计的时候，就应该先考虑到这些因素，以满足综合工具最佳的工作条件。

在做工程的第一次综合的时候，可以让综合工具自动完成整个流程。然后，设计者再仔细分析这一次综合的结果，比如面积、时序等，然后再人工加约束来约束综合过程。在有了完整的综合约束后，设计者还是有可能会要手动更改综合的网表结果。在综合出来的网表的基础上再加上人工修改，这样的结果可能会比任何一个综合工具的结果都要好。但是，在综合的时候，设计者往往会上加上很多“set_dont_touch”命令，这些指令有可能会阻碍综合工具综合出一个好的结果。

综合工具通常会把一些跨逻辑边界的元件的位置做一些调整，从而保证综合工具可以更好地优化电路。和不改变设计划分的综合相比，被打平 (flattened) 综合出来的网表优化的效果可能会更好。除非是只有一个模块的设计，否则这种优化方法只应该在设计的最后阶段才能使用。因为这种综合方式会打散原有模块间的逻辑关系，而且这个优化过程是不可逆的，会使得手工修改网表变得非常困难。

即使是被打平的网表，后端的布局工具还是可能重构出和原始 Verilog 层次设计一样的物理布局。这是因为在综合工具或优化工具在唯一化和打平之后，Verilog 模块的名字被保留了下来。

15.3.4 跨时钟域的同步

时钟域是由这个设计里所使用的时钟决定的。虽然用 PLL 或分频产生的时钟的频率和原始时钟不一样，但是在这一节里，我们不把这种情况归到不同的时钟域里去。因为产生出来的

时钟的相位和原始时钟的相位是锁定的，要处理的问题也仅仅是时钟的偏移（skew）和抖动（jitter）而已。

当跨时钟域传送数据的时候，我们会遇到单时钟设计里不会遇到的问题。当设计中有两个完全不同的时钟时，这两个时钟的相位关系是完全随机的。用一个时钟去采样另一个时钟驱动的数据时，可能会采样到这个数据的中间状态，一个既不是1也不是0的状态。事实上，在目前的高速电路里，尤其是时钟达到了吉赫兹等级的设计中，这种采样到中间状态的情况是难免的。

打个比方来说明这个问题：从空中往地面垂直地释放小薄片，任其自由落下，那么，这个小薄片从释放到落地的整个过程中，一直都维持在垂直状态的概率是非常小的。但是，如果我们重复这个过程的次数足够多，比如说100万次，也许就能找到在1 s 内都维持垂直下落的小薄片。如果做这个实验的次数足够多，也许能找到下落时垂直状态维持时间达到10 s 的小薄片。

同样地，我们总能找到由于采样时机不正确，导致输出不确定的情况。图 15.7 就描述了这么一种情况。

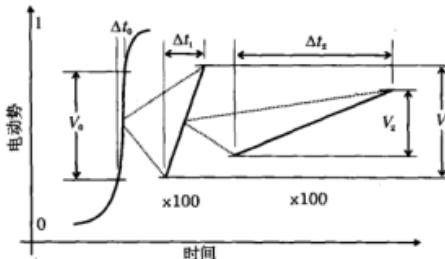


图 15.7 各类电压输出从顶部的1到底部的0占有一定跨度。图中没有标注刻度。

如果 $\Delta t_0 = 100\Delta t_1$, $\Delta t_1 = 100\Delta t_2$, 则 $V_1 \approx V_0/100$; $V_2 \approx V_1/100$, 近似表明

门输出处于 Δt 中间时刻的概率 p 分别为: $p_1 = 100 p_0$ 和 $p_2 = 100 p_1$

受时钟驱动的电路在进行采样时，输入电压可能恰好处于高低门限之间。这时，电路就不能正确地把1和0传递给下一级了。如果电路继续工作下去，可能这个错误就会发展到无法纠正的地步了。

我们可以将输入用接收时钟打两拍的办法来解决这个问题。如图 15.8 所示，输出的信号在输出时钟域里被打了一拍，在接收时钟域里，这个信号被接收时钟打了两拍。然后，即使这么做，接收时钟仍然有可能会采样到输出信号的中间状态。但是，由于被两个同步寄存器保存了两次，信号的不确定状态却不会传递下去。

接收的两级寄存器中总有一级能采样到达到输入门限的信号。两级寄存器的输出都处于不确定状态的可能性是非常小的，我们完全可以忽略这种情况。如果有必要，可以采取在设计中增加校验逻辑来确保数据完全正确。

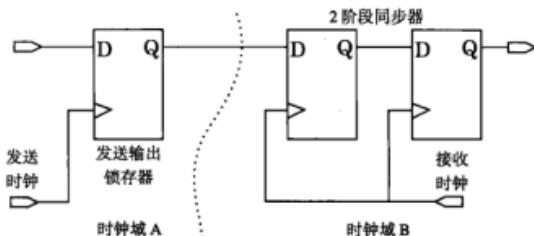


图 15.8 跨时钟域数据同步的办法。时钟域A和时钟域B的工作时钟各不相同，在时钟域B里把数据打两拍

15.4 练习 19：层次

在 Lab19 目录下完成以下练习。

练习步骤

第1步。新建一个名为 MiscModules.v 的文件，输入如下的空模块（只有模块头）：

```
module Wide(output[95:0] OutWide, input[71:0] InWide);
endmodule
module Narrow(output[1:0] OutNarrow, input[1:0] InNarrow);
endmodule
module Bit(output Out, input In);
endmodule
```

第2步。将一个 1 比特宽的信号连在一个 2 比特宽的端口上。

用第1步产生的文件，在 Narrow 中例化 Bit，并将它的端口连接至 Narrow 的端口上。

```
module Narrow(output[1:0] OutNarrow, input[1:0] InNarrow);
  Bit Bit1( .Out(OutNarrow), .In(InNarrow) );
endmodule
```

不要特意将这个 1 比特信号连接至 2 比特端口某个特定的比特上。思考最终的结果。事实上，Narrow 的某一个比特会和输入的 1 比特信号相连。会是哪一个呢？

做个小实验来验证你的想法。比如说：

```
assign Out = 1'b1;
```

通过仿真来观察到底哪个比特被赋值了。

第3步。容易混淆的层次关系。不要改修第2步里完成的文件，再把 Narrow 例化在 Bit 里。故意不去连接 Narrow 的端口。这样的做法符合 Verilog 规范吗？请自行编译 MiscModules 并仿真检查结果。

第4步。将一个 2 比特宽的信号连在一个 1 比特宽的端口上。在 Narrow 里注释掉第2步例化的 Bit 实例，在 Bit 中保留 Narrow 实例。类似于第2步，把 Narrow 的端口连在对应的端口上（不要有意地连接某一个比特）。在 Narrow 随便做一个赋值，再仿真试试最后是哪根线被连上了。

第5步。重复第4步，用Narrow和Wide替换Bit和Narrow。你应该能够预计这样做的结果，可以仿真来看看结果。

第6步。时钟同步仿真写一个名为ClockDomains的测试模块。这个模块有两个时钟，慢时钟的半周期为2.011 ns，快时钟的半周期为1.301 ns。为了能让仿真环境能够达到这样的时间精度，请将`timescale设为1 ns/1 ps。

如图15.9的电路所示，设计一个3比特的计数器模块，Counter3，把计数器的3个比特都作为输出，给这个模块加上复位端。在这个练习里，我们让这个计数器对时钟的边沿进行计数，而不是时钟的周期，也就是说，用always@(clk)而不是always@(posedge clk)。在这个测试模块里将这个计数器例化两次，分别用两个不同的时钟来驱动它们。

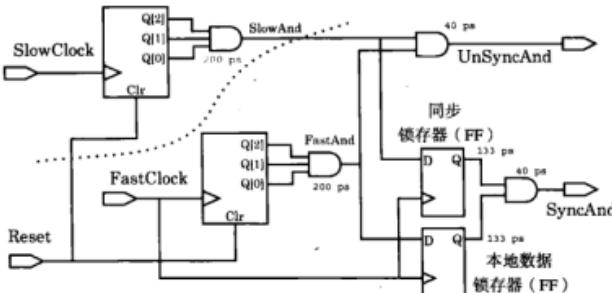


图 15.9 ClockDomains 的电路示意图。两个时钟域的输出直接相与产生 UnSyncAnd。这两个输出同时也被快时钟打了一拍再相与，产生 SyncAnd。图中标注的延迟只是一个大概的值

把两个计数器各自的3个比特都做一个与运算，分别把结果直接送给两个线类型，并且在赋值时加一个比如200 ps的延迟。这两根线将在快时钟域里做比较，当它们都为1时，快时钟域将会进行一些处理（在本例中，暂时先把这部分空下来）。

A. 为了观察这两根与的结果，把这两个结果再与一下，并且通过连续赋值语句把结果赋送给一个名为UnSyncAnd的线类型。赋值的时候，再加上一个很小的延迟，是其他延迟的1/5就可以了。请用十进制小数（1/5.0）来使系统仿真的延迟精度可以计算到小数。用你的模型来做仿真。你会看到UnSyncAnd有时会为1，有时却是一个很窄的毛刺。因为两个时钟频率的不同，这个信号的正脉冲的宽度会一直在变化。

B. 在快时钟域里增加一个同步寄存器用来同步慢时钟的与结果。当快速时钟变高时，去采样慢速时钟的与结果就可以了。再给这个过程加一点延迟。

我们并不需要对快时钟的与结果做同步，但需要用一个寄存器来保持这个结果的值。因此，再增加一个寄存器用来锁存住这个结果。

```

localparam AndDelay = 0.200; // 200 ps 3-input and gate delay.
localparam LatchLagDelay = AndDelay/1.5;
...
reg HoldSlowAnd; // The synchronizing latch storage.
always@(posedge FastClock)
  if (FastClock==1'b1)
    #LatchLagDelay HoldSlowAnd = SlowAnd; // Sample the slow-domain and.
...

```

最后，再把这两个锁存后的与结果再与一下。

```
assign #(AndDelay/5.0) SyncAnd = HoldFastAnd & HoldSlowAnd;
```

仿真的时候注意比较 SyncAnd 和 UnSyncAnd (参见图 15.10 和图 15.11)。会发现 SyncAnd 的波形看起来很干净，看不到毛刺。而 UnSyncAnd 的波形看起来就很糟糕了，又有毛刺，又不规则。

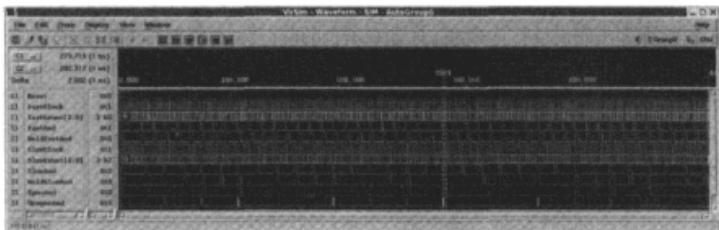


图 15.10 ClockDomains 的仿真结果

因为我们做仿真的时钟是理想的，通过调整延迟就可以减少或去除出现毛刺的情况。同样，对于 UnSyncAnd 这个信号来说，也可以通过调整延迟的办法来改变采样时机从而去掉一些毛刺。当两个时钟完全不相关时，为了解决它们的同步问题，并不是要把时钟本身做调整，而是在接口的地方完成异步时钟和数据的同步。

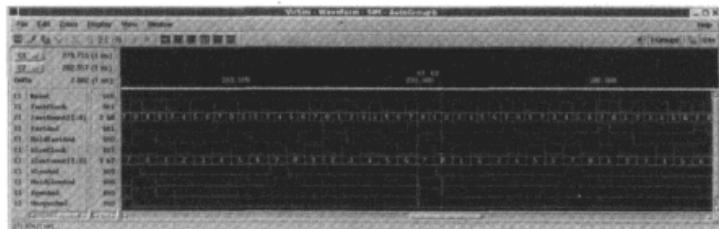


图 15.11 局部放大的 ClockDomains 的仿真结果

15.4.1 练习后的思考

之前我们说过，综合工具不支持延迟语句。因此，建议不要在顺序执行语句中加入延迟语句。一种替代方法是，我们可以将合成的延迟放在对模块 output 和 inout 赋值的连续赋值语句中。这对我们做设计划分有什么影响吗？

15.4.2 补充学习

复习参数、层次命名、Thomas and Moorby (2002) 3.6 节和 5.1~5.2 节中的连接规则。

Palnitkar 大量使用了传统的模块头声明；Thomas and Moorby (2002) 则在前言 17~19 页里概要地描述了这些区别。

第16章 配置和时序

16.1 Verilog 的配置

大多数工程都是先设计行为模型（比如C语言的模型），在利用行为模型确定了设计的划分和接口之后，再进入RTL的编码阶段。这样的设计流程导致一个模块的具体实现现在不同版本里可能存在巨大的差异。为了让工程能正常工作，必须有一个完整的包含所有设计文件的列表。设计者往往通过makefile和shell的脚本来管理同一个设计不同的版本。这里，我们再介绍一种Verilog语言内置的管理版本的办法。

16.1.1 库

库（library）是由EDA供应商的工艺、技术决定的基本门电路的集合：Synopsys有自己的库，其他的EDA供应商也有自己的库。Verilog提供了一种把库映射到工程中的办法：library mapping file。在这个课程的开始，我们已经给出了一个用Synopsys的DC综合时包含库的例子。在那个最简单的例子里，库的名字与其对应的目录排在一起，目录中对应的内容都对应于这个库。

16.1.2 Verilog 的配置

在Verilog-2001(IEEE Std 1364, Section 13)里，配置(configuration)的作用被定义成使得模块的升级更容易，使得库能够更通用。

配置的内容由关键字config开始，到出现关键字endconfig的地方为止。配置描述了设计利用的所有有用的库和其他一些有用的内容。它保存在设计文件中，和module是同级别的。

config的格式如下所述：

```
config config_name;  
design design_top_name;  
default list_of_libraries;  
cell cell_name use library_name;  
instance inst_name [use] instance_liblist [.cell];  
endconfig
```

上面黑体字的内容都是Verilog的关键字，其余的文字描述了配置的具体内容。其中，只有design和default是必需的。这里的default虽然和case中的default是同一个单词，但是它们的含义却是完全不同的。下面分别详细介绍关键字的含义。

- config: config后的内容是配置的名字。可以随着设计用途的改变，改变config的名字。假设有一个CPU的设计，最早的配置名称是CPU_BusXfer；在换了新的库之后，配置改名为CPU RTL；后来又发展成了CPU_FloorPlanned。工程中，设计会根据不同的config名来读取相关的信息。

另一个例子是，不同的config可用于仿真而非综合。

- **design**: design 指定了设计中用到的库和设计的顶层模块。指定库时应该按照 Verilog 的层次命名规则。如果库的名字是 IntroLib，则顶层的设计文件应该这样表示：

```
design IntroLib.Intro.Top;
```

- **default**: 除了用关键字 instance 专门指明之外的库，default 按照文件中调用的顺序列举了设计中用到所有的库的具体名称。比如说，如果我们用的是 Synopsys 的综合库 class.db，则这条命令应该这么写：

```
default class;
```

- **cell**: cell 指明了库中的具体单元（模块）。命令 use 则指明了是哪一个库。

- **instance**: 在 default 库列表中找不到模块或门在这里专门被列出。命令 use 可专门指定某一个特定的单元（这条命令不是必需的）。举个例子，在练习 1 里，我们曾经用过异或 (^) 表达式。假设在库 special.db 还有一个异或门的综合模型，而且我们想在设计中使用这个模型。则应该这样指定这个异或门：

```
instance IntroLib.Intro.Top.XorNor.xor_01 special;
```

如果用综合出来的库，这样指定就可以了：

```
instance Intro.Top.XorNor.xor_01 special;
```

库和配置的维护和开发工具密切相关，并且是一个很深入的话题，在这里就不继续深入讨论这个问题了。

Verilog 的配置并没有给 Verilog 本身增加什么新功能。而现在的配置信息在文件系统里就已经有了。如果用得不好，Verilog 的配置这个功能点本身有可能因为把配置信息搞混淆从而导致设计出错。据作者所知，还没有开发工具完整地支持 config，因此这部分内容没有课后练习。

16.2 时序弧和 specify 延迟

接下来，我们来学习模块里的时序弧。

16.2.1 弧和路径

我们规定，模块的延迟是沿着时序弧（timing arc）分布的。当设计的模块最终成为物理芯片时，不同点之间的时间关系就很重要了。

时序弧可以理解成为模块里两点之间的是时序延迟。这些延迟无法用表达式、赋值等方式表示出来，因为它们和位置是密切相关的。时序弧代表的距离可能很短，也可能较长，穿过了由很多门组成的组合逻辑，也可能经过时序逻辑。时序弧可能存在与时钟和端口之间，或端口和内部逻辑之间。不管时序弧存在于什么地方，它都代表着这段路径的延迟。

如图 16.1 所示，时序弧可以存在于输入与输出（从 A 或 B 到 C 或 D）之间。如果我们规定从 A 到 C 存在一条时序弧，那么从 A 到 D 也存在一条时序弧。同样，A 或 B 到 E 或 G，F 或 H 到 C 或 D 之间，F 到 G 之间都可以存在时序弧。在这一个图中，我们看不到 E 到 F 或 G 到 H 的时序弧，但这样的时序弧却可以存在于实例 1 (instance 1) 和实例 2 (instance 2) 中。如果我们要计算 B 到 D 的时序弧时，E 到 F 或 G 到 H 的时序弧的值也是需要计算的。

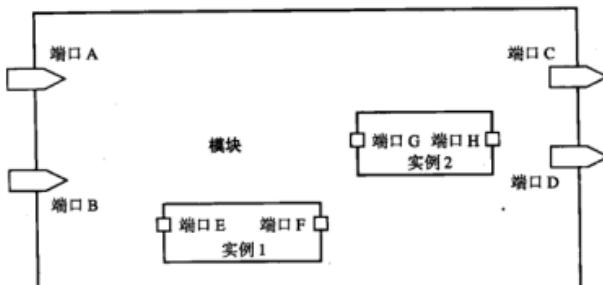


图 16.1 时序弧只存在于端口和 Pin 脚之间。图中的 A, B, E, G 是输入，其余是输出

图 16.1 里并没有画出连线，我们可以通过任意添加连线来学习时序弧。

下面，我们将会详细讨论延迟的概念。对于仿真工具和静态时序分析工具来说，延迟是非常重要的。

16.2.2 分布式延迟和块延迟

在我们之前做过的练习里，这两种延迟都已经遇到过了。分布式延迟 (distributed delay) 是指可以将延迟时序弧分解为多个小延迟的延迟。也可以这么说，一个较长的时序弧经过了多个端点，每一段是一个小的子弧。而对块延迟 (lump delay) 来说，设计者已经计算出了整块逻辑的延迟，中间的子弧不能人为设置任何延迟，即使是用 #0 作为延迟值也不行。

下面是分布式延迟和块延迟的比较。

```
module DistributedDelay (output Z, input A, B);
  wire Node;
  assign #1 Z = Node;           // Output port delay.
  and #(2,3) (Node, A, B);    // Pin delay.
endmodule
//
module LumpedDelay (output Z, input A, B);
  wire Node;
  assign #(3,4) Z = Node;      // Total delay lumped on output port.
  and (Node, A, B);
endmodule
```

在有子模块的模块中，分布式延迟和块延迟概念的区别才有实际意义。对于单模块来说，这两者之间的区别不明显。我们用一个较复杂的例子来说明这个问题。

```
'timescale 1ns/100ps
module FourFlopsRTL #(parameter DClock = 2, DBuf = 1)
  (output[3:0] Q, input[3:0] D, input Ena, Clk);
  reg[3:0] QReg;
  wire[3:0] Qwire;        // Not used yet.
  //
  always@(posedge Clk)
    #DClock QReg <= D;
  //
  assign #DBuf Q = (Ena==1'b1)? QReg: 'bz;
  //
endmodule
```

从 Clk 到 QReg, QReg 到 Q 之间都有延迟, 但是只有从 D、Ena、Clk 到 Q 的延迟才是时序弧。

由参数可知, D 到 Q 的延迟是 $2 + 1 = 3$ ns, Ena 到 Q 的延迟是 1 ns, Clk 到 Q 的延迟是 $2 + 1 = 3$ ns。要把这些延迟划分成分布式延迟或块延迟都是没意义的。

再来看看下面这个更复杂一些的例子。FourFlopsStruct 和 FourFlopsRTL 的功能和时序是完全一样的, 但是它包含了子模块。

```
module FourFlopsStruct #(parameter DClk = 2, DBuf = 1)
    (output[3:0] Q, input[3:0] D, input Ena, Clk);
    wire[3:0] QWire;
    //
    DFF #(DClk(DClk)) DReg[0:3](.Q(QWire), .D(D), .Clk(Clk));
    assign #DBuf Q = (Ena==1'b1)? QWire: 'bz;
endmodule // FourFlopsStruct.

// -----
module DFF #(parameter DClk = 2) (output Q, input D, Clk);
    reg QReg;
    always@(posedge Clk) QReg <= D;
    assign #DClk Q = QReg;
endmodule // DFF.
```

FourFlopsStruct 虽然和 FourFlopsRTL 的时序是一样的, 但是在 FourFlopsStruct 中的延迟是分布式延迟, 而实例 DReg 的输出延迟是块延迟。

我们可以在声明线型的时候, 指定其延迟值。例如, wire #3 DataOut, 这样的方式来指定延迟和用连续赋值语句通过一个临时变量来指定延迟的办法是等效的。

虽然对于深亚微米的设计来说, 线延迟在整个延迟中所占的比例已经不容忽视。但在前端设计的时候是不考虑线延迟的。在实际工作中, 目前只考虑门之间的时序弧。至于线延迟, 在用后端工具进行了布局布线之后, 才会得到准确的线延迟。这时, 才将线延迟反标回网表再来做时序仿真。

也可以把 FourFlopsStruct 的延迟重新设计成块延迟。

```
module FourFlopsStructL #(parameter DClk = 2, DBuf = 1)
    (output[3:0] Q, input[3:0] D, input Ena, Clk);
    wire[3:0] QWire;
    localparam DTot = DBuf + DClk;
    //
    DFF DReg[3:0](.Q(QWire), .D(D), .Clk(Clk));
    assign #DTot Q = (Ena==1'b1)? QWire: 'bz;
endmodule // FourFlopsStructL.

// -----
module DFF(output Q, input D, Clk);
    reg QReg;
    always@(posedge Clk)
        QReg <= D;
    assign Q = QReg;
endmodule // DFF.
```

在块延迟里可以分别表示上升沿和下降沿的传输时间, 对 DTot 来说, 格式如下: (DTotR, DTotF)。为了表示三种延迟, 最小延迟、典型延迟和最大延迟, 还可以写成这样的格式: (DTotRmin:DTotRtyp:DTotRmax, DTotFmin:DTotFtyp:DTotFmax)。

16.2.3 specify 块

specify 块以关键字 specify 开始, 以关键字 endspecify 结束。在模块里, specify 块的和 always 块、initial 块一样, 都是被并行处理的。specify 块并不包含功能性的设计, 但是它包含了 Verilog 库中门器件或其他模型准确的时序参数。

specify 块中的时序参数用起来比在声明、赋值或例化时添加时序信息方便得多。specify 使得设计者不需要知道模块的功能就可以建立模块的时序模型, 而模块的功能可以留到以后再来实现。例如, 设计者可以用 specify 块来建立一个加密 IP 的静态时序模型, 而不会泄露这个 IP 的设计细节。

我们还会学习时序检查的其他相关知识, 但是到目前为止, specify 是我们接触到的语法结构里唯一能够在模块中进行时序检查的语句。时序检查函数和系统函数前都有 “\$” 这个符号, 为了避免混淆, 不允许在 specify 块中使用系统函数。

specify 块中可以有 specparam 的定义、时序检查、脉冲过滤的条件和模块的路径延迟。现在, 我们就来学习关于 specparam 的定义和模块路径延迟的知识。

模块的端口声明对与这个模块里的 specify 块来说是可见的, 而模块里声明的寄存器或模块例化实例的端口不能在 specify 中引用。只有模块中的普通连线、parameter 值和 localparam 值可以在 specify 块中被引用。

综上所述, specify 块和 always 块、initial 块和 generate 块是平级的。这个块以 specify 开始, 以 endspecify 结束。

specify 块可以包含的内容:

- parameter 或 localparam 的引用
- 模块端口和连线的引用
- specparam 定义
- 模块的延迟
- 时序检查

specify 块不能包含的内容:

- 其他的块
- 除了 specparam 之外的声明
- 被赋值的寄存器或连线
- 其他模块中的实例
- 所有的任务和函数, 包括系统任务和函数

16.2.4 specparam

specparam 是仅在 specify 块中使用的 Verilog 参数。specparam 也可以在 specify 块外部被声明, 但是很多工具不支持这样的写法。

这种特殊参数的作用是方便工具从模块中提取准确的时序信息。specparam 只有一种用法: 被赋给一个或多个数值。如下面的格式所示:

```
specparam Name = (x, y, z);
```

上式中的 x 、 y 、 z 表示时间延迟。

当我们试图建立一个模块的完整时序信息时，把这些信息都放在 specparam 中是个很好的做法。这些信息可以被 specify 块中的其他的表达式所引用。我们可以给这些信息起一些方便记忆的名字。

如下例，这样来说明路径延迟

```
module ALU (output[31:0] Result, input[31:0] ArgA, ArgB, input Clk);
...
specify
  specparam tRise = 5, tFall = 4;
  ...
  (Clk *> Result) = (tRise, tFall); // A simple full-path delay.
endspecify
...
endmodule
```

specparam 可以按照最小 - 典型 - 最大的方式来赋值，如下例所示：

```
specify
  specparam tRise = 2:3:4, tFall = 1:3:5;
  ...
  (other stuff; maybe complicated)
  ...
  (Clk *> Q,Qn) = (tRise, tFall);
endspecify
```

16.2.5 并行路径延迟和全路径延迟

路径延迟的表达式有两种，全路径 ($*>$) 和并行路径 ($=>$)。全路径延迟是所有端口的每一个比特可能组成的时序弧的集合。全路径延迟可以是扇出时序弧（如上例中的 $Clk *> Result$ ），也可以是扇入时序弧。

并行路径延迟只存在于等位宽的端点之间，扇入和扇出的弧延迟不是并行路径延迟。通常，并行路径延迟被用于都是 1 比特位宽的端口之间，如果端口位宽超过 1，则对应比特的延迟被一一映射。

如果扇入是 wor 或 wand 这样直接连接在端口上的逻辑，则用哪一种规则去描述这个路径延迟都是可以的。如果要给这样的扇入指定路径延迟，必须把这个逻辑替换成单输出的等效门。

下面是路径延迟的例子：

```
module FullPath (output[2:0] QBus, output Z, input A, B, C, Clock);
  ... (functionality omitted) ...
specify
  specparam tAll=10, tR=20, tF=21;
  (A,B,C *> QBus) = tAll;
  (Clock *> QBus) = (tR, tF);
endspecify
endmodule
// -----
module ParallelPath (output Z, input A, B, C, Clock);
  ... (functionality omitted) ...
```

```

specify
  specparam tAll=10, tR=20, tF=21;
  (Clock => Z) = tAll;
  (A => Z)      = (tR, tF);
  (B => Z)      = tAll;
endspecify
endmodule

```

specify 的延迟可以多达 6 个。Thomas and Moorby (2002) 的 6.6 节里做了详细的介绍，specify 块支持指定 6 组不同的延迟，它们的顺序如下，(0_1, 1_0, 0_z, z_1, 1_z, z_0)。但是在实际的工程中，几乎没有这么用。这些通常都是工具给定的，综合工具或布局布线工具给出的结果会更准确，工程师也不会那么辛苦了。

16.2.6 条件延迟和边沿延迟

路径条件表达式的延迟可以是任意的。常见的 Verilog 操作符都可以在表达式中使用，1, x, z 都被表达式看成非假。如果表达式的结果不是 1 比特的，则结果的真假（true 和 false）由结果的 LSB 来决定。表达式中只允许有最简单的语句，不允许有 else, case 和等有其他条件分支的结构。关键字 posedge 和 negedge 的含义和以前没有区别。如下例所示：

```

// output[3:0] Z, input[3:0] A, input Clk, Clear are declared ports.
...
specify
  specparam ClkR=2, ClkF=3, ClearRF=1, AThruR=4, AThruf=5;
  ...
if (Clk && !Clear) (A => Z) = (AThruR, AThruf);
if (A[0] && A[3]) (A[1], A[2] >> Z) = AThruR; // Lists are OK
if (A[1] && A[2]) (A[0], A[3] >> Z) = AThruf;
if (!Clear)          (negedge Clk >> Z) = ClearRF;
if (!Clear)          (posedge Clk)>> Z) = (ClkR, ClkF);
                           (posedge Clear >> Z) = ClearRF;
endspecify

```

路径延迟可以指定极性，即路径延迟和端口脉冲边沿的方向有关。这样，当输出发生变化时，由对应的路径就可以计算出对应延迟。如下例所示：

```

// output Q, Qn, input D, Clk: Q1 and Q2 are logically equivalent.
...
specify
  specparam tR_Q = 5, tF_Q = 6.5;
  ...
  ( posedge Clk => (Q1 +:D) ) = (tR_Q, tF_Q );
  ( posedge Clk => (Q2 -:D) ) = (tR_Q, tF_Q );
endspecify

```

在上例中，时钟 Clk 驱动数据从 D 传向 Q。如果 Q1 是 0, D 是 1，则 tR_Q 是延迟值；如果 Q1 是 1, D 是 0，则 tF_Q 是延迟值。

在指定路径延迟时，符号“-”说明 D 的极性被取反。此时，D 的延迟取值符号为“+”时是相反的。所以，如果 Q2 是 0, D 是 1，则 tF_Q 是延迟值；如果 Q2 是 1, D 是 0，则 tR_Q 是延迟值。

并行路径延迟和全路径延迟都可以指定延迟。“+”不对极性取反，“-”则要取反。“-=>”说明要把它左边的参考规则取反，它的右边是并行路径延迟。对全路径延迟来说，规则也是一样的。如下例所示：

```
...
specify
  [clk -> Q1, Q2] = t.ClkTog; // Delay when Q1 | Q2 goes opposite clk.
  [clk +> Q1, Q2] = t.ClkReg; // Delay when Q1 | Q2 goes same way as clk.
endspecify
```

这些特性对开关级（switch-level）电路的建模很有帮助。

在指定延迟时，可能需要明确的指出信号在各种状态之间的转换（包含x和z）。这说明仅用两个参数（rise, fall）或三个参数（rise, fall, turnoff）来表达延迟是不够的，而是应该用6个参数来表达包含z的所有状态，或用12个参数来表示包含x和z的所有状态。关于这一点，请查阅 Thomas and Moorby (2002) 附录 G.8 获取更多的内容。

16.2.7 specify 和其他延迟的冲突

由 Thomas and Moorby (2002) 的 6.6 节和 IEEE 1364 标准的 14.4 节，如果 specify 块内指定了一条路径的延迟，specify 外的代码也指定了同样一条路径的延迟。如果两个延迟不同，默认为最大的延迟为准。

但是需要提醒读者的是，不同的开发工具可能会根据自己的需要选择这两种延迟中的一种，也许会和 Verilog 规定的默认规则不一样。

16.2.8 specify 延迟的冲突

由 IEEE 1364 标准的 14.3.3 节规定，当某个端点被指定了不同的延迟时，以延迟值小的值为准。

16.3 练习 20：时序

用目录 Lab20 下的 Verilog 程序来完成这部分练习。

练习步骤

在目录 Lab20/SpecIt 下的 Verilog 程序里使用了负时序。在进行这个练习之前，先把 SpecIt 目录复制至上一层，即 Lab20 下。设计 SpecIt 的顶层电路示意图如图 16.2 所示。它的两个子模块的电路示意图如图 16.3 和图 16.4 所示。前面的章节曾经讲过，模块的名称和它例化的实例的名称不属于同一个命名区间，因次，它们的名字可以完全一样。但是，我们并不推荐这样做。

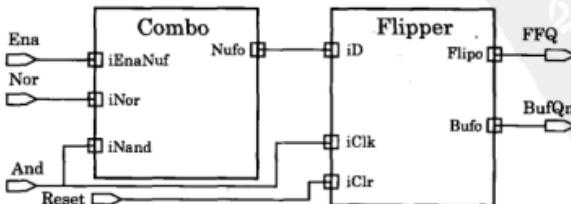


图 16.2 SpecIt 的顶层电路图。图中列出了端口名称和实例名

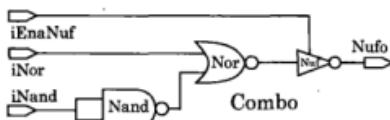


图 16.3 Combo 的电路图。输入信号的名称以 i 开头，输出信号的名称以 o 结尾

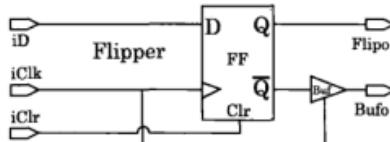


图 16.4 Flipper 的电路图。输入信号的名称以 i 开头，输出信号的名称以 o 结尾

第1步。在 testbench 中例化出 SpecIt，并设计一个 4 比特的递增计数器，把这 4 个比特按从 MSB 到 LSB 的顺序和 SpecIt 的端口连上； $\{\text{MSB}, \dots, \text{LSB}\} = \{\text{Reset}, \text{Ena}, \text{Nor}, \text{And}\}$ 。这个计数器的时钟频率为 10 MHz，并用仿真工具来检查你的设计有没有语法错误，如图 16.5 所示。

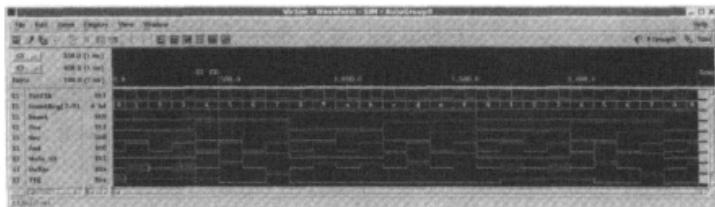


图 16.5 SpecIt 的仿真波形

第2步。块延迟和分布式延迟。在 Combo 里，分别给 Nand、Nor 和 Nufo 的输出设置 3 ns、5 ns 和 7 ns 的延迟。这样，从 iNand 到 Nufo 会有 12 ns 或 15 ns 的分布式延迟。从仿真波形里观察这个延时，如图 16.6 所示。

如果两个输入同时改变并且被指定不同的延迟，会有什么样的结果？

给 SpecIt 中的实例 Combo 设置 10 ns 的块延迟。由于 Combo 只有一个输出，所以延迟对这个输出的影响是没有任何疑问的。那么，这么做有什么潜在的冲突呢？

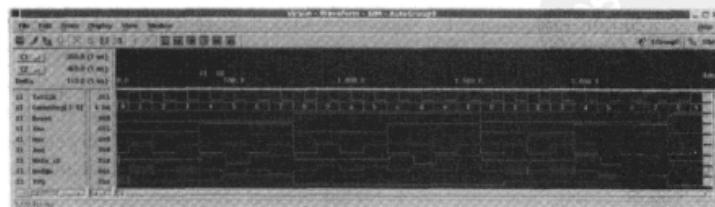


图 16.6 有分布式延迟的 SpecIt 的仿真波形

第3步。造成路径延迟和分布式延迟的冲突。把Combo.v另存为一个新的文件，文件名叫ComboSpec.v，保留模块名也不改变，也保留第2步里加入的分布式延迟。把SpecIt工程的文件列表中的Combo.v替换成ComboSpec.v。

A. 在ComboSpec.v里，用specify块指定模块Combo的所有输入和所有输出之间都有10 ns的全路径延迟。用specparam来指定这个参数，并仿真观察结果。

B. 将specify块的延迟设置成：上升延迟为20 ns，下降延迟为21 ns，关断（turnoff）延迟为22 ns。会发生什么呢？

根据Verilog的规范IEEE 1364，当延迟发生冲突时，我们应该按照悲观的原则来处理：当发生1到0的跳变时，应该使用最长的延迟；当有信号跳变到不定态x时，应该使用最短的延迟；要从不定态x跳变到其他状态，则应该使用最长的延迟。所有由specify指定的新延迟都大于Combo实例中默认的分布式延迟。

C. 不理会上一步用specify指定的具体延迟值，在Combo中用localparam定义一个参数叫InstTime，它的值是25 ns，把Combo里的门电路分布式延迟都替换成25 ns。再来仿真，观察结果如何？

D. 保留第3步C里定义的分布式延迟，定义另外一个参数：localparam tZ = 30，把它作为specparam里的关断时间。再仿真观察结果。这个例子说明了利用参数来改变模块时序模型的好处。

有的仿真工具可能不支持用parameter或localparam来定义specparam。如果遇到了这种情况，用localparam来替换specparam就可以了。

E. 在第3步D里，试着在不改变其他地方的同时把全路径延迟赋值改为并行延迟赋值，这样会发生什么呢？你使用的仿真工具应该报错或者至少给出一条警告。

第4步。造成并行延迟和全延迟的冲突。把Flipper.v另存为FlipperSpec.v，保留原来的模块名不变。把最早的那份Combo.v和FlipperSpec.v组合在一起，在这一步里，将用这个版本来做仿真。

A. 利用specify中的specparam，给模块Flipper（不是DFFC模块）里的output都添加并行路径延迟。并使Bufo的所有输入都有1 ns的延迟，端口Flipo有5 ns的上升延迟，有8 ns的下降延迟。

如下例所示：

```
specparam tClkQR=5, tClkQF=8, dBuf=1, tClr=2;  
...  
(posedge iClk => Bufo) = (tClkQR+dBuf, tClkQF+dBuf);  
...
```

B. 假设一个时钟的下降沿在Bufo被关断，在specify块中添加一个1 ns的关断延迟，但是只使用上升和下降延迟。

C. 再给Flipo端口也增加一个2 ns的并行路径延迟。仿真并观察结果。

D. 在完成了A、B、C三步之后，用一个列表给Flipper里iClk到两个输出端口的路径上都增加30 ns的全路径延迟。仿真并观察结果。你会很容易的在波形中发现这30 ns带来的波形延迟。

E. 把上一步的全路径延迟从 30 ns 改成 0 ns。仿真并观察结果。问题：如果一个端口被指定了不同的延迟，仿真工具使用的是哪个值呢？

第 5 步。用第 4 步 E 里的 Flipper，恢复第 2 步里 Combo 的那些延迟值，再仿真。你应该能看到和图 16.7 一样的波形。

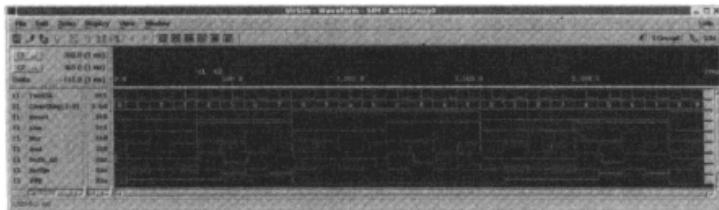


图 16.7 完成第 5 步后，SpecIt 的仿真波形

综合 SpecIt，注意不要用错了文件，综合之后阅读时序报告。

产生 SDF 文件（回忆第 1 章的内容），用一个文本编辑工具打开这个文件，简单地看一下。你会发现文件中有很多三个一组的延迟值。在后面的课程里，我们还会继续学习关于这种文件的知识。如果你用反标了 SDF 的网表仿真，你看到的波形如图 16.8 所示。

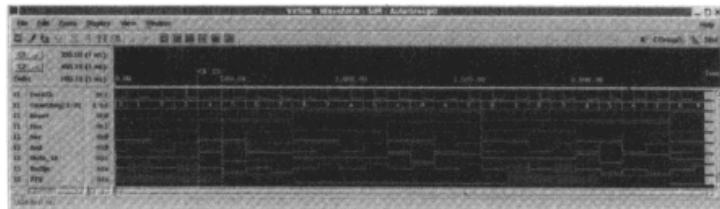


图 16.8 完成第 5 步，再反标注了 SDF 后，SpecIt 的仿真波形

16.3.1 练习后的思考

如果模块最后输出的状态相同，但内部的延迟不同，仿真工具会怎样来解决这个问题？输出状态不同的时候又是怎样的呢？

16.3.2 补充学习

阅读 Thomas and Moorby (2002) 的 6.6 节关于 specify 块延迟的内容。

阅读 Palnitkar (2003) (可选)

阅读 5.2 节和 6.2 节关于门级赋值时序的内容。

阅读第 10 章关于路径延迟和其他延迟的内容。

试着解答 10.6 节的 1、2、3 三题。

第17章 时序检查和断言

17.1 时序检查和脉冲控制

17.1.1 时序检查和断言

时序检查函数看起来和系统函数一样，都以“\$”开头。但是它们和系统函数却是有本质区别的（参见 IEEE Std 1364 的第 15 章）。系统函数是顺序执行的代码，而时序检查函数只能在 specify 块中出现，并且是并行执行的。有的系统函数，例如 \$display，可以在断言的时候使用，而它输出的结果可能看起来很像是时序检查的结果。

在第 8 章中，我们学到了用简单的 Verilog 系统函数可以组成断言（assertion）检查。在某些语言里，断言是一种内建的语法结构，用来检查某些变量或信号在某些条件下的工作状态，并且打印输出对应的警报或错误提示。

综合工具或静态时序分析工具分析信号的输入和对应的输出，如果满足了特定的条件，也可以通过断言机制输出信息或中断当前的进程。

在综合网表和对其进行静态时序分析时，我们需要使用到 Liberty 库，它的功能和 Verilog 里的时序检查一样。但是，Verilog 的仿真工具不支持 Liberty 库中的检查功能，只有在下面两种情况下才会用到 Liberty 库：(a) 在进行综合优化的时候，为了避免和约束发生冲突；(b) 静态时序分析期间。

断言（assertion）、调试（debugging）和覆盖率（code coverage）的区别如下：断言说明从设计者的观点来说，某些功能点可能会出现错误；调试发生在发现了错误之后；而覆盖率可以用来衡量调试的工作量。可以这么说，时序检查是一种内建的、专用的断言机制。当某种设计约束被违反时，时序检查机制都会向控制台程序输出消息。仿真工具往往支持把时序检查信息保存至 log 文件。

仿真时，时序检查并不改变仿真波形或仿真信号的结果。然而，时序检查里的确有一种 notifier 机制，当仿真发生了时序违例时，可以改变仿真的流程。

前面提到过，时序检查只能出现在 specify 块里，不允许出现在 always 块里，也不允许出现在顺序执行块 task 或 function 里。时序检查在 specify 块中，通过连线检查整个设计的时序，如果它们检测到发生了时序违例，则输出信息。若仅通过观察波形的方式来进行调试，则很有可能会忽略时序检查功能检查出来的这些信息。

下面归纳时序检查的特征：

- 它们的语法都是这样的：`$name_of_timing_check`
- 从功能上来说，它们都是预定义的断言
- 它们和顺序执行的断言的区别如下：
 - 时序检查只允许出现在 specify 块中

- 都有内建的触发逻辑
- 并行执行
- 时序检查功能集成在仿真工具中，进行时序检查可能会增加仿真时间

17.1.2 时序检查基础

在进行仿真时，时序检查通常是检查硬件的工作是否符合设计规范最有效的办法。对于仿真来说，所有的时序检查都是基于以下两点：参考事件（reference event）和数据事件（data event）。这些事件可以用仿真中的一个或多个变量来调度。要注意的是，数据事件中的数据（data）和我们以前提到的用时钟和控制信号驱动的数据是两个不同的概念。时序检查会在这两种事件上增加特定的条件，当条件（也就是逻辑表达式）为真时，时序检查将不进行任何处理。通常，参考事件被理解成固定的时间，而数据事件则指在一个时间范围内所发生的违反时序要求的事件。

这里介绍两个相关的术语：时间戳（timestamp）和时间检查（timecheck）。在时序检查中，当第一个参考事件或数据事件发生时，它的时间（即时间戳）被记录下来。当再次发生了这类事件之后，时间检查会检查这两个时间的条件。

这些检查中的时间限制通常被定义成仿真时间里的一个范围。把时间限制设成0可以很方便地关闭信号歪斜之外的时序检查功能。

通常，即使发生了多个时间检查事件，时序检查仅在时间戳事件（第一个事件）发生时才被触发。有一些时序检查支持打印连续的违例信息，比如说检查总线上不同的比特。

通常，我们在 specparam 中定义时序检查的时间限制，且这个时间限制是常数。如果检查的是设计中的矢量（多比特位宽的变量），违例仍然只被触发一次。

如果需要给时序检查传递参数或其他一些输入，只能按照位置对应的办法来传递这些数据。

17.1.3 Verilog 中的 12 种时序检查

下面是按应用来分的一个完整的列表。不管它们检查的对象是什么，效果都是一样的。

时钟 - 时钟检查	时钟 - 数据检查	时钟控制检查	数据检查
\$skew	\$setup	\$recovery	\$width
\$timeskew	\$hold	\$removal	\$period
\$fullskew	\$setuphold*	\$recrm*	\$nochange

* 尽量不要用。

下面将详细介绍每一种时序检查，还会列出这些时序检查需要的输入。针对 \$width，还列出其可选的时间参数。如果读者需要深入了解如 edge specifier 或 remain-active flag 等关于时序检查的所有信息，请参考 IEEE 1364 规范。

在 QuestaSim 工具中，时序检查是被当做断言来处理的。因此为了使用时序检查，请打开它的断言功能选项。

17.1.3.1 时钟 - 时钟检查

\$skew。这个功能检查两个变量事件之间的延迟。延迟必须为非负值。通常，参数事件（时间戳）是时钟，接着是数据事件，并进行时间检查。如果没有数据事件，也就没有时序违例。

如下例所示：

```
ref. event      data event      limit expression
$skew(posedge Clock, posedge GatedClock,      MaxDly);
```

\$timeskew。若超过了指定的时间，不管有没有数据事件，都会发生时序违例。

\$fullskew。这个检查支持两个非负的延迟。第一个延迟指的是从参考事件到数据事件之间的延迟，第二个延迟指的是数据事件到参考事件之间的延迟。

如下例所示：

```
ref. event R      data event D      R-D limit  D-R limit
$fullskew(posedge Clock, posedge GatedClock,  MaxRD,      MaxDR);
```

17.1.3.2 时钟 - 数据检查

\$setup。在发生了数据事件之后，又发生了参考事件（通常是时钟），且这个时间间隔过短，则这个检查会被触发。指定的时间间隔必须为非负值。数据事件发生的时刻是触发时间窗口的起始时间。在这个检查里，时间截是数据事件发生的这个时刻。

如下例所示：

```
data event  ref. event  time limit
$setup(    D,        posedge Clk,  MinD_Clk );
```

\$hold。在发生了参考事件（通常是时钟）之后，又发生了数据事件，且这个时间间隔过短，则这个检查会被触发。指定的时间间隔必须为非负值。参考事件发生的时刻是触发时间窗口的起始时间。在这个检查里，时间截是参考事件发生的这个时刻。

如下例所示：

```
ref. event      data event      time limit
$hold( posedge Clk,      D,      MinClk_D );
```

\$setuphold。这种检查支持两个时间间隔，且均可为负值。若两个时间间隔均为非负值，则这个检查把\$setup和\$hold的功能合在了一起。但是，这种用法容易导致出错，因此，若没有特殊的需求，应尽量少用。下面的章节会有关于这点的解释。

17.1.3.3 异步时钟控制检查

从概念上来讲，这些检查检查了信号之间的内部延迟。而这些信号通常是时钟和异步信号。时间截对应于仿真中发生的一个事件，它既可以是参考事件也可以是数据事件。请参考图17.1的波形。

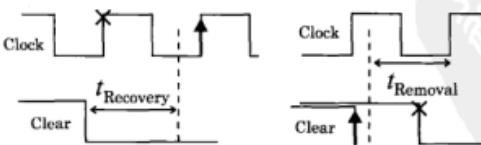


图17.1 恢复时间和移除时间的示意图。Clear高有效，无效的边沿被加上了×，有效的边沿被加上了箭头

\$recovery。恢复指的是从异步控制信号（例如置位和清零）无效到下一个有效时钟沿之间的恢复时间。它的含义是：当控制信号从有效变成无效之后，需要多长的时间才能恢复时钟的功能。这个时间间隔必须为非负值。

如下例所示：

```
ref. event    data event      limit
$recovery(negedge Clk, posedge Clk, MinClk.Clk);
```

\$removal。 移除指的是从有效时钟沿到下一个异步控制信号无效之间的时间。它的含义是：在有效的时钟沿之后，控制信号还需要维持多长时间才能达到控制的结果。这个时间间隔必须为非负值。

如下例所示：

```
ref. event    data event      limit
$removal(negedge Clk, posedge Clk, MinClk.Clr);
```

注：演示版的 Silos 可能不支持 \$removal。

\$recem。 这个检查支持两个时间限制，恢复时间在前，移除时间在后。如果两个时间限制都为正，则这个检查和 \$recovery 加 \$removal 的效果是一样的。\$recem 的时间限制允许是负值。使用两个时间限制容易导致出错，因此，若没有特殊的需求，应尽量少用。后面会专门讨论负值的时间限制。

17.1.3.4 数据检查

\$width。 它的作用是检查某个 1 比特信号的脉冲宽度。第一个跳变沿作为时间截事件，如果从时间截到下一个反向跳变沿之间的时间间隔过短，则 \$width 将会被触发。这个时间间隔必须为非负。\$width 支持两个时间参数，另外一个时间参数是毛刺门限 (glitch threshold)。如果仿真时检测到了小于毛刺门限时间的小毛刺，则不将其视为时序违例。

如下例所示：

```
ref. edge    width    glitch thresh.
$width(posedge Reset, MinWid,     MinWid/10);
```

\$period。 如果在一个很短的时间里一个信号同向的边沿连续产生了两次，则这个检查会触发一条时序违例。时间的间隔必须为非负值。

如下例所示：

```
ref. edge    period
$period(posedge Clk, MinCycle);
```

\$nochange。 它的作用是用来检查这种情况：某个信号的边沿触发了一个参考事件（这个时间也就是时间截），从这个参考事件算起，往后的某一段时间不应该有数据事件（时间检查）发生，否则会触发一条时序违例。这个检查需要两个时间参数：第一个参数是时序违例区间的起始时间，第二个参数是时序违例区间的结束时间。时间参数允许为负。

如下例所示：

```
// Require DBus constant during entire positive phase:
    ref. edge    data event    lead shift   lag shift
$nochange(posedge Clk,           DBus,          0,           0);

// Violation starts 1 before negedge; ends 2 after posedge:
specparam MinSetup = 1, MinHold = 2;
$nochange(negedge Clk, EBus, MinSetup, MinHold);
```

```
// Shift the check 1 unit later:  
specparam MinSetup = -1, MinHold = 1;  
$nochange(negedge Clk, FBus, MinSetup, MinHold);
```

注：据作者所知，目前没有任何一种仿真工具支持 \$nochange 这个功能。

17.1.4 负时间限制

建议读者不使用以下两种支持负参数的时序检查：setuphold 和 recrem，因为它们增加了检查的复杂度。对设计的检查不应比设计本身还复杂，否则，检查本身的错误可能使设计者花更多的时间来调试虚假的时序违例，甚至导致设计者忽略设计本身的失误。

如果我们使用的是一个IP的Verilog模型，而这个模型没有包含足够的内部时序信息，则设计者需要在IP的接口加入时序检查，而其内部信号的时序信息是检查不到的。在这种情况下，就有必要引入负的时间限制了。

让我们通过一个例子来看为什么有必要这么做。请思考图17.2中的建立和保持时间。

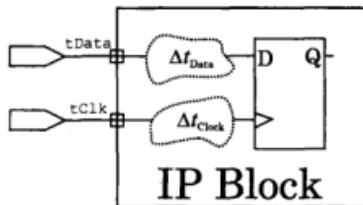


图17.2 IP内部的寄存器是不可见的。只能通过IP边界的事件发生时间 tData 和参考事件的发生时间 tClock 来判断

有三种可能性：

(a) 时钟和数据到寄存器的延迟是一样的；(b) 时钟到寄存器的延迟更大一些，时钟插入buffer可以导致这个现象；(c) 数据到寄存器的延迟更大一些，数据经过的组合逻辑可以导致这个现象。

这三种可能性造成的信号延迟如图17.3所示。

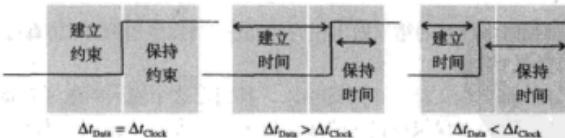


图17.3 歪斜的时间限制原理。阴影区域表示时序元件确定宽度的要求。如果额外的延时相等，正常的建立和保持时间检查和约束时间相同。若数据延时大于参考延时，则必须增大建立约束时间并减小保持约束时间；若数据延时小于参考延时，则必须增大保持约束时间并减小建立约束时间

当IP造成的延迟超过了建立和保持时间的限制时，建立或保持时间的极限被超过，从而产生了负的时间限制（参见图17.4）。

作者反对使用 \$setuphold 和 \$crecm。如果内部延迟的差异是已知的，我们可以不再使用 \$setuphold 和 \$crecm，而是通过把延迟加在临时的线上来调整数据和时钟的关系。例如，若 $\Delta t_{Data} > \Delta t_{Clock}$ ，则应该在时钟上加延迟，从而设计者可以在设计中直接使用规范中规定的时序信息来进行时序检查。

```
wire ClockToHold;
assign #tCancel ClockToHold = Clock; // tCancel from IP vendor or experiment.
specify
$hold(posedge ClockToHold, DataIn, tHold); // tHold from data book.
...
```

设计的其余部分不会用临时的线型（例如 ClockToHold）调整延迟。在综合的时候，这些线型也会被综合工具优化掉。在线型上加的延迟并不是为了消除整个IP的延迟，仅仅为了调整延迟量从而避免产生负的时间限制。

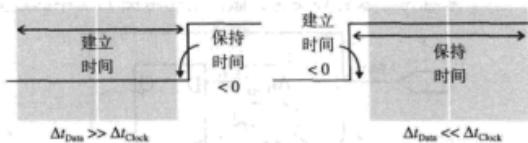


图 17.4 当延迟相差过大时，产生了负的时间限制

17.1.5 时序检查条件事件

在时序检查中，可以使用逻辑表达式。它的操作符是 `&&&`，和其他表达式中的 `&&` 是一个含义。若希望进行检查，则表达式必须为真。在 `&&&` 开头的表达式中，可以使用 `==`、`!=`、`==!`、`!=!`、`~` 这些操作符。

在时序检查里，条件表达式 RHS 必须为 1 比特操作的表达式。虽然使用矢量是允许的，但是只有 LSB 才参与逻辑运算。

下面的例子里，我们不希望在 Ena 为低且 D 的数据发生变化时产生违例。

```
$setup( D&&(Ena==1'b1), posedge Clk, MinD.Clk );
```

17.1.6 时序检查通知

仿真器可以根据时序检查的结果做出相应的动作。当发生违例时，仿真可以被终止，也可以通过断言输出详细的信息。

所有的时序检查都至少支持一个可选输入。我们把这个输入叫做通知寄存器（notifier reg）。这个 1 比特的寄存器对模块里包含的 specify 块是可见的。这意味着任何一个寄存器的时序违例都可以通过这个通知寄存器来传递。通常，设计者在检查到通知寄存器的通知后，会用 \$stop 终止仿真或通过断言输出失败信息。

当时序检查被时序违例触发后，notifier 的值会在 0 和 1 之间进行翻转。当发生违例时，若 notifier 的值是 x（不定态），则它将翻转成 0；若 notifier 的值是 z（高阻），则它将仍然保持 z 值。最后这一条特性提供了在不关闭时序检查的条件下关闭通知功能的办法。

必须在使用到这个寄存器之前声明它。

如下例所示：

```

reg Notify;
...
always@(Notify) $stop;
...
specify
...
$setup( D, posedge Clk, MinD.Clk, Notify );
endspecify

```

17.1.7 脉冲过滤

除了可选的通知 (notifier) 功能，时序检查不会影响到仿真。而脉冲过滤是仿真器的一个重要功能。

通常，对默认的惯性 (inertial) 延迟来说，比门电路延迟还要短的脉冲在门电路的输入端被移除。在 Verilog 里，这种脉冲被过滤掉的情况对应于错误时间 (error limit) 和拒绝时间 (rejection limit) 正好相等的情况。

在 Verilog 里，错误时间总是大于或等于拒绝时间。若拒绝时间小于错误时间，对于一个输入来说，延迟对其的影响可以分为三种：

- (a) 当脉宽大于错误时间时，脉冲被延迟，但是波形保持不变。
- (b) 当脉宽小于错误时间但大于拒绝时间时，脉冲被延迟，脉宽缩短至拒绝时间。
- (c) 当脉宽小于拒绝时间时，脉冲被过滤掉。

这些时间限制可以通过仿真器选项或 SDF 来设置。这里，我们只讲述 specify 块中的 PATHPULSE，我们可以把它看做一种特殊的 specparam。

PATHPULSE。PATHPULSE 是 Verilog 里唯一不用小写来表示的保留关键字。它的语法形式较为独特，它的两个参数都指定了时序的路径，“\$”是它的分隔符。它改变了对应路径的时序。例如，将 PATHPULSE 应用于端口 Ain 和 Bout 之间的路径应该这样写：

```
specparam PATHPULSE$Ain$Bout = (r_limit, e_limit);
```

其中，*r_limit* 是拒绝时间，*e_limit* 是错误时间。图 17.5 是一个例子，和 specparam 类似，PATHPULSE 也可以只有一个参数。错误时间是可选的，可以只指定拒绝时间，此时的错误时间将默认地被认为是和对应的门延迟相等。用在路径中的名称必须事先声明，且不允许使用比特选择或部分选择规则。

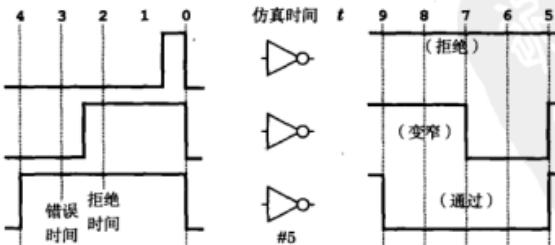


图 17.5 PATHPULSE 过滤脉冲的效果。假设 PATHPULSE\$ = (2, 3)，在没有 PATHPULSE 说明时，因为门延时为 5，所以所有脉冲都会被简单的惯性延迟所忽略

省略路径也是允许的，如这样的写法：“PATHPULSE\$ = (r_limit, e_limit);”或“PATHPULSE\$ = r_limit;”。这样的写法把这个模块里所有的路径都设置了同样的值。如果这样的写法和指定了路径的 PATHPULSE 同时存在，指定了路径的 PATHPULSE 会在那条路径上覆盖掉所有路径的统一设置。

如下例所示：

```
module NewInertia #(parameter r_limit = 3, e_limit = 4:5:6)
    (output Z, input A, B, C);
    ...
    specify
        ... (delays; timing checks) ...
        specparam PATHPULSE$ = r_limit; // Module default.
        specparam PATHPULSE$B$Z = ( r_limit, e_limit );
    endspecify
endmodule
```

specify 块可以指定全路径延迟。如果全路径延迟和 PATHPULSE\$ 同时存在，则 PATHPULSE\$ 只对第一条从输入到输出的路径有效，而对其他的路径延迟是不起作用的。因此，PATHOULSE 通常用来指定单比特位宽（组成）的并行路径，而不是全路径。

注：不同于VCS, Silos 和 QuestaSim 可能不支持 PATHPULSE。当脉宽小于拒绝时间且打开了 +pathpulse 选项之后，这两个工具仅仅会输出“x”；而在 VCS 里，除了输出“x”，还会输出警告信息。

17.1.8 延迟悲观处理原则的改进

回忆第 14 章中讲到的处理延迟的悲观原则。有时，悲观原则是多余的。

除了 PATHPULSE，还有 4 种保留的 specparam 类型：pulsestyle_onevent, pulsestyle_ondetect, showcancelled 和 noshowcancelled。

和 PATHPULSE 不一样，它们只对指定的输出变量的延迟使用增强功能的“x”功能。在用这 4 种类型给任何一条路径指定延迟之前，都必须在 specify 块中先声明。

pulsestyle_onevent。这是处理竞争时的默认属性：竞争产生的“x”被正常处理。即当仿真器第一次检查到发生竞争的地方，才会产生“x”。

pulsestyle_ondetect。这条规则比上面一条更加悲观一些：只要仿真器检查到了竞争，就会产生“x”。在这种条件下，“x”的输出会比默认的输出稍早一些。当仿真器在门电路的输入端检测到有可能使输出为“x”的输入时，立即输出“x”。这样做的目的是为了让设计者在仿真时尽早发现不符合预期的状态。这种只要发现就立即输出“x”的做法使得设计者得以了解这个不符合预期的状态最早在什么时候就可能会发生。在一个大型设计中，如果发生了这种情况，应该马上触发 \$stop，而不是继续进行其他的事件流程，从而节省调试时间。

showcancelled。有时，由于不同的上升和下降延迟，导致处于“x”状态的时间为 0 甚至是负值。如果出现了这种情况，仿真器默认会忽略它并且不给出任何提示。若把一个输出端口送至 specparam，则说明仿真器在原来输出第一个跳变沿的地方，又加上了一个时间宽度为 0 或负的“x”脉冲。这个脉冲的宽度和输入脉冲的宽度是一样的。如果，这个输入端口被送至 pulsestyle_ondetect，输出“x”的时间点会早于仿真里实际应输出“x”的时间，使得输出脉冲“x”的宽度被展宽。

处于调试的考虑，我们也会把 showcancelled 列表里的变量再送给 noshowcancelled specparam 用来观察其默认的延迟行为。

注：Silos、QuestaSim 和 VCS 似乎都不支持上述提到的这些特性。不过，这些工具有一些选项可以实现类似的功能。

17.1.9 和时间相关的其他类型

除了 specparam，在 specify 块中使用的 Verilog 标准类型还有 time 和 realtime 类型。

time 是无符号的 reg 类型，它的默认位宽不低于 64 比特。它用于 testbench 和其他需要进行长时间仿真的时序检查中。可以在这个 reg 类型的值赋值给多比特位宽的线之后，再送给 specify 块用。

除了名字不一样，realtime 和 time 其实是一样的。与此类似的还有 tri 和 wire，除了名字，其余都是一样的。

time 和 realtime 可以在模块中任何允许使用 reg 类型的地方使用。我们应谨慎使用 time 和 realtime，把它们引入到设计中并没有增加新的功能，但是使得语法却变得更复杂了一些。一种比较好的办法是把变量命名为和时间相关而不是去声明一个时间类型。这样，每当我们用到这个变量的时候，便知道它和时间相关，而且也不会增加复杂度。由于这两个类型并没有增添新的功能，因此，有可能有的仿真工具不支持它们。

17.2 练习 21：时序检查

在目录 Lab21 下完成这个练习。子目录 PLLsync 已经提前被建好了，里面包含了练习 10（参见第 7 章）中的整个 PLLsync 设计。

练习步骤

首先确认你使用的仿真工具的 PATHPULSE 和时序检查的选项都已经打开。

回忆一下，PLLsync 这个设计由以下几个部分组成：PLLsync 的顶层由 PLLsync 和 Counter4 组成。PLLsync.v 中有一个叫 PLLsyncTst 的 testbench。在名为 PLL 的子目录里有 5 个文件：一个 include 文件和另外 4 个设计文件。这后 4 个文件分别包括了模块 PLLTop、ClockComparator、MultiCounter 和 VFO。

我们将通过 VFO 来说明即使是固定时钟，也可以来调整它的延迟信息。同时，还会利用这个模块来说明时序检查的相关功能。

第 1 步。开始仿真。进入 PLLsync 目录。用提供的 testbench，对 PLLsync 做一个较长时间的仿真，比如说，25 000 ns。由于`VFO_MaxDelta 在 PLL 的 include 文件中定义的是 2，因此，这个 VFO 的振荡周期不超过 4 ns。如果你在 VFO 中观察 VFO_Delay 这个变量，会发现它会以 1 ns 的间隔在 14~18 ns 之间切换（对于 32 MHz 的 PLL 时钟来说，这个平均值应为 15.625 ns）。这说明这个计数器有 28~36 ns 的延迟。

第 2 步。建立时间检查。假设希望在给 VFO 的 Sample 上升沿去采样计数器的数值。需要保证这两个信号之间有 5 ns 的建立时间。在 PLLsync 这个模块中，Sample 信号来自于 SyncPLL，同时，计数器的值被命名为 Behavioral 并且被输出。接下来，要用 \$setup 来检查 PLLsync 是否

会违反 5 ns 的建立时间规则。用 specparam 指定这个限制，然后仿真，并观察仿真器控制台的输出结果。然后再把这个限制改成 0 ns，使这个检查无效。

第3步。保持时间检查。给保持时间限制设成 8 ns。仿真，观察到违例后将限制改成 0 ns。

第4步。恢复和移除。将 testbench 改成 PLLsync 连续收到两个 ClearIn 脉冲；两个脉冲之间的间隔为 50 ns，脉宽为 500 ns。脉冲之间的间隔应该以一个时钟的边沿为中心。

将 ClockIn 从 ClearIn 下降沿算起的恢复时间设为 100 ns，并在这些边沿增加 100 ns 的溢出检查。这样的值会触发恢复时序违例和移除时序违例。仿真并观察结果。然后把检查时间缩短，保证不产生时序违例，然后再仿真。

第5步。宽度和周期检查。在 PLLsync 中，用宽度检查来检查 ClearIn 低电平的时间是否会小于 100 ns。仿真并观察结果。然后，将检查宽度设成 0，使得这个检查无效。用基于上升沿的周期检查来确保 PLL 时钟的输出周期不小于 30 ns。仿真后再关掉这个功能。

关于 PLLSync 的练习就做到这里。下面，先把练习 8（参见第 6 章）中的 DFFC.v 模型（带置位的 D 触发器）复制至目录 Lab21。

请按步骤依次完成下面的内容。

第6步。脉冲过滤。如果输入变化的过快会怎么样？

要回答这个问题，我们先来修改 DFFC 模型。

首先去掉 DFFC 文件里的 timescale 设置，并把原来通过连续赋值加在 Q 和 Qn 上的 #1 赋值改成 #0.001 (1 ps)，让它很小但又不为 0。

在模块定义之后添加一个 specify 块，加入如下的延迟，并给它们取好记的名字：

设 Q 到时钟上升沿的延迟为 (rise: 1000 ps, fall: 800 ps)；Qn 的延迟为 (rise: 1100 ps, fall: 850 ns)。

clear 到 Q 或 Qn 的延迟为 (rise: 700 ps, fall: 900 ps)。

在 testbench (DFFC_Tst 中，参见目录 Lab21)，把 DFFC 例化成一个不停翻转的寄存器。这个 testbench 逐渐把时钟周期从 10 ns 减到 10 ps。同时，这个 testbench 还提供了周期为 50 ns，占空比为 50% 的 clear 信号。为了支持 1 ps 的仿真周期，请设置 `timescale 1 ns/1 ps。

为了让寄存器能翻转，将输出 Qn 用线连至输入 D。

A. 肉眼检查（由于使用了浮点数，仿真时间会长一些）。仿真并观察波形。当 D 触发器的功能变得不稳定时，此时的时钟频率是多少？提示：Q 和 Qn 都必须要工作。这和模型中的延迟有什么关系？通过观察图 17.6 和图 17.7 得到结果。



图 17.6 DFFC 的仿真波形

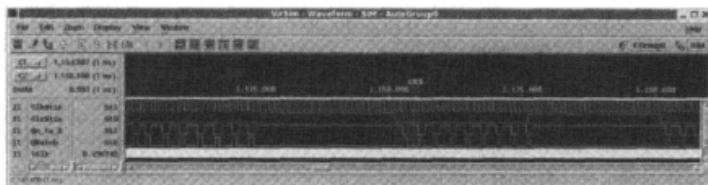


图 17.7 DFFC 的仿真波形里时序违例的例子

B. 可靠性。假设我们需要给任何到 Q 或 Qn 的路径都增加 1100 ps 的错误时间和 500 ps 的拒绝时间的限制。通过在 DFFC 的 specify 块中增加一个或多个 PATHPULSE specparam 来完成这个要求。仿真并观察仿真结果，然后注释掉 PATHPULSE。

第7步。宽度。我们希望 Q 和 Qn 的高电平都至少能够维持 1 ns 的时间。

- A. 增加 \$width 时间检查并仿真。
- B. 将最宽度限制改成 0.850 ns 并仿真，我们会看到产生了一些违例。
- C. 在 DFFC 中声明一个新的寄存器变量，名称为 Notify，并把它作为第 4 个参数送至 \$width 中；为了过滤掉毛刺，把第三个参数改成 0。在 specify 块中增加如下的 always 块：

```
...
$width(poposedge Qn, twMinQQn, 0, Notify);
endspecify
// 
always@(Notify) $stop;
```

现在重新仿真。在每个 \$width 违约后，可以随意继续仿真过程。把 specparam 的最小宽度值改成 0.500 ns 来去掉宽度违例，关于宽度的检查就告一段落了。

第8步。建立和保持时间在高时钟频率下会有很多方式揭露问题。其中一种是增加建立和保持时间约束。

- A. 为 Clk 和 D 之间时序关系加上 1 ns 的建立时间和 500 ps 的保持时间检查，然后仿真观察结果。
- B. 把 \$setup 和 \$hold 的连接至寄存器 Notify 并且再仿真。减小 specparam 中 setup 和 hold 的值，使得不再有建立和保持时间的违例。最短的违例时间是多少？第一次发生最短违例的仿真时间又是多少？
- C. 在 testbench 中，把 timescale 改为 10 ns/1 ns，并把建立和保持时间改成会导致冲突的最小值。仿真并观察会发生什么？有没有注意到在这个很粗的时间分辨率下仿真结束的非常快？

把时间分辨率改成 10 ns/100 ps 和 10 ns/10 ps，再仿真。仿真的结果又是什么？

再把时间分辨率改成 1 ns/1 ps，关掉 \$width, \$setup 和 \$hold 检查，然后再仿真。

第9步。歪斜检查。在时钟和 clear 之间加入 \$skew 检查。如果时钟在 clear 之后的高电平时间超过 49.99 ns，则将会产生违例。仿真并观察结果。然后关掉这个检查：可以注释掉这个检查，也可以把这个时间限制改成超过 50 ns。

第 10 步。恢复和移除。这可能是最容易弄错的时间检查了。这两个检查的作用是检查异步信号之间的关系，例如置位端或时钟。

A. 恢复检查。用 \$recovery 检查从 Clk 的下降沿直到 Clk 的上升沿之间的时间没有超过 10 ps 的情况。仿真并观察第一次违例发生在什么时候？把时间限制改成 0 关掉这个检查。

B. 移除检查。用 \$removal 检查 Clk 的上升沿之前的 10 ps 里 Clr 的下降沿是否有未移除的情况。仿真并观察结果（参见图 17.8）。通过改变时间限制来关掉这个检查。

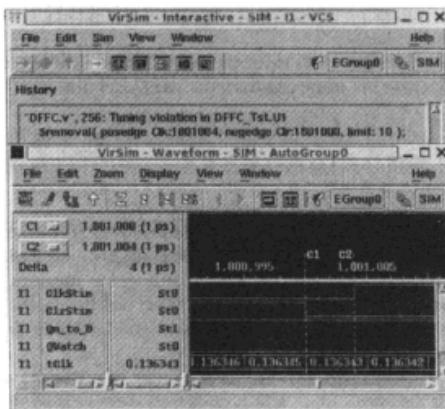


图 17.8 \$removal 产生的时序违例

17.2.1 补充学习

阅读 Thomas and Moorby (2002) 的 8.1 节和 8.4.4 节，深入了解关于惯性延迟的知识。

阅读 Palnitkar (2003) (可选)

阅读 10.3 节关于时序检查的内容。

做 10.6 节后的习题 6~8。

第18章 解串器和升级PLL

18.1 串行序列解串器

我们先来回顾工程 serdes，看看还剩下哪些工作。

在第4章中，我们介绍了 serdes 工程。在那一章的 PLL Clock 练习 6 里，完成了一个基本的 SerDes PLL，在第 8 步，还完成了并串转换的基本框架。在第 7 章的练习 10 (PLL 的锁定) 里，添加了串行帧同步的功能。在练习 11 里，还设计了一个 FIFO，虽然由于这个 FIFO 的敏感列表和锁存器的原因，网表的功能是不正确的，但用这个 FIFO 来仿真是完全没有问题的。在此基础上，在第 13 章的串并练习 16 里，我们设计了串并解码模块：DesDecoder。

在这个练习中，我们将会重新设计 PLL 使它能够被正确综合，而这个 FIFO 设计暂时不去改它。完成了这个练习之后，将会完成整个串并转换器的设计。

图 18.1 从数据流的角度说明了我们目前做的工作在整个设计里所处的位置。

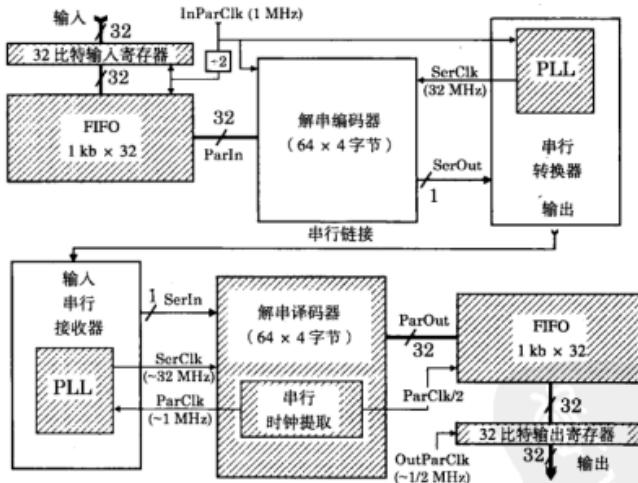


图 18.1 serdes 工程的数据流和时钟分布。上一半是 Serializer，下一半是 Deserializer，整个设计在练习 16 中被更新。设计中的模块在前面的练习里被分别完成。PLL 和 FIFO 目前还不能综合，但功能仿真是正确的。

为了完成整个串并换转，首先我们需要修改 PLL 使得它能够被正确综合。在更新了 PLL 的设计之后，这次练习里主要的内容就是重新整理整个设计了。

18.2 重新设计 PLL

PLL 目前的问题在于 VFO：这个振荡器用到非阻塞赋值时，赋值的延迟是可变的。为了产生一个网表（功能是错误的），我们通过宏开关使得编译器只能看见阻塞赋值而看不见非阻塞赋值。

用数字电路的描述语言去设计一个模拟器件（比如说我们用到的这个 PLL）并不是一个好办法。我们无法用语言来描述一个随着时间逐渐变化的电容，并且用它来控制 PLL 中的 VCO（Variable-Capacitance Oscillator）。

18.2.1 改进的 VFO 采样

在练习 6、练习 8 和练习 10 里，为了减少 VFO 输出频率调整的次数，我们将一个外部信号 Sample 送给 PLL。我们并没有用边沿平滑（edge-averaging）或其他的一些平滑手段来处理 ClockComparator 的输出。如图 18.2 所示，虽然采样脉冲的办法不是必需的，但是它的确使得 VFO 没有在每个时钟有细微偏差的时候都去调整。

为了使得 PLL 能够做到自适应，我们不再把外部的采样脉冲送给 VFO。我们会把用比较器驱动的采样时钟做到 PLL 设计中去，这个电路每一个有效的时钟就会工作一次，工作频率大约是 1 MHz。

为了保证采样的有效性，必须使得比较器的计数值在采样脉冲有效之前已经稳定。我们可以通过在外部输入时钟上加上如图 18.3 所示的库延迟单元，从而使得采样脉冲到得更晚。接下来，计数器使用的时钟是由 VFO 产生的时钟。

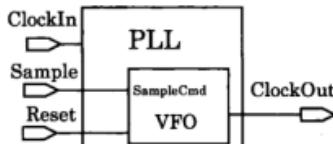


图 18.2 练习 6 里的 VFO 受 sample 信号的控制

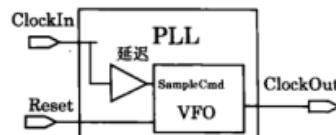


图 18.3 新的 VFO 设计

这样写代码来加入这个库延迟单元：

```
module PLLTop (output ClockOut, input ClockIn, Reset);
// ...
Library_DelayCell DelayU1 (.Z(SampleWire), .I(ClockIn));
// 
// (dont_touch DelayU1 synthesis directives)
// 
VFO VF0U1 ( .ClockOut(MHz32), .AdjustFreq(AdjFreq)
, .Sample(SampleWire), .Reset(Reset) );
```

18.2.2 可综合的可变频率振荡器

练习 6 中的 PLL 综合出来的网表是没有办法使用的。我们通过定义 Verilog 宏使得综合工具只能看见阻塞赋值而看不见非阻塞赋值。

练习6中的VFO综合出来的网表里的确有振荡器的功能，但是它的频率无法改变，它只能以综合约束允许的最高频率来工作。用典型的约束条件综合出来的电路是工作在6GHz的或非门。这个用阻塞赋值的PLL综合后产生了如图18.4所示的锁存器。

我们会用目标库中速度最快的延迟单元来完成这个可以工作在很高时钟下的振荡器设计。除了用来描述单元自身的延迟和门级的传输延迟，在Verilog文件里不再人为增加延迟。

为了能够控制快速振荡器产生的频率并把它作为VFO的时钟输出，我们用快速时钟驱动一个计数器，其溢出的周期就是新时钟的周期。通过调整溢出的门限值，就可以控制VFO的频率。

频率由线延迟来控制，而线延迟可以用generate表示出来。当设计反相器时，通过选择延迟在80~90ps之间的那些库单元来控制整个反相器的延迟。振荡器的输出被改名为FastClock并且在VFO的内部被使用。如图18.5所示，用或非门替换了原反相器，因为这么做支持再加入复位端。

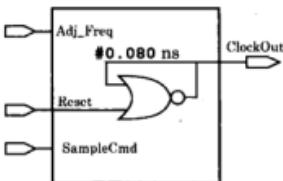


图18.4 练习6里的VFO的综合结果

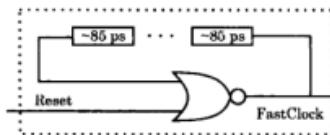


图18.5 新VFO的电路示意图

下面是振荡器的Verilog代码：

```

reg FastClock;
wire WireToDelay, WireFromDelay;
assign WireToDelay = ~FastClock; // oscillation here.
// -----
// The always block allows initialization:
always@(WireFromDelay, Reset)
  if (Reset==1'b1)
    FastClock <= 1'b0;
  else FastClock <= WireFromDelay;
// -----
// The delays control the (fixed) fast oscillator speed:
LibraryDelayCell Delay0(.Out(Wire1), .In(WireToDelay));
LibraryDelayCell Delay1(.Out(Wire2), .In(Wire1));
...
LibraryDelayCell DelayN(.Out(WireFromDelay), .In(WireN));
// (synthesizer dont.touch on all Delay instances)

```

我们必须添加DC的set_dont_touch命令来阻止综合工具把我们认为增加的线延迟给优化掉。对于这种延迟线在设计内部的情况，用set_dont_touch会比在.sct文件中加命令更方便。这样写出来的代码就是完全可以综合的。

把这个振荡器的输出命名为FastClock，并作为时钟驱动计数器，把计数器的门限值叫做VaryFreq，用来控制VFO的时钟频率，如图18.6所示。

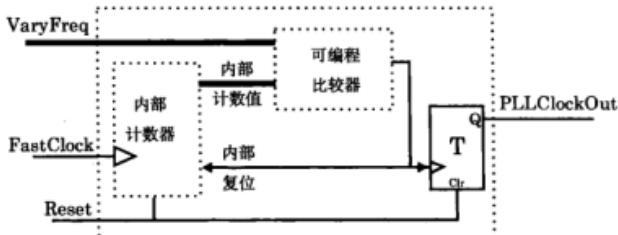


图 18.6 新的 VFO 加入了可编程的计数器，它的输入时钟是 FastClock

VFO 内部的比较器（不要把它与 PLL 里的 ClockComparator 混淆）将计数值与计数器的溢出门限 VaryFreq 做比较，当计数值达到门限时，计数器被复位。计数的门限可以调整。当计数器被复位时，VFO 的输出时钟同时翻转，从而产生了时钟。

下面是这部分的 Verilog 代码：

```
// Assume a VaryFreq vector declared which
//      sets the VFO frequency:
reg[HiBit:0] Count;
reg PLLClockOut;
always@(posedge FastClock, posedge Reset)
  if (Reset==1'b1)
    begin
      PLLClockOut <= 1'b0;
      Count      <= 'b0;
    end
  else begin
    if (Count>=VaryFreq) // Programmable limit.
      begin
        PLLClockOut <= ~PLLClockOut;
        Count      <= 'b0;
      end
    else Count <= Count + 1;
  end
```

对于 32xPLL 来说，我们需要足够快的库元件来让计数器工作得足够快从而使 PLLClockOut 的频率能够准确地在 32 MHz 左右变化。如果时钟的半周期是 16 ns 且精度为 1 ns，则我们需要一个能在 20 ns 里计数到 20 的计数器，也就是说，一个工作在 1 GHz 频率下的 5 比特计数器。对于 130 ns 或更先进的工艺库来说，达到这个速度是没有问题的。

下面我们来看看可综合的 1xPLL 设计。

18.2.3 可综合的频率比较器

32xPLL 这个模块受 1 MHz 的时钟驱动并且采样 VFO 的 PLL 输出。下面是其代码：

```
// OLD, unsynthesizable VFO:
always@(ClockIn, Reset)           // The input system clock.
  if (Reset==1'b1)
    ...
    else if (CounterClock==1'b1)   // The PLL MultiCounter output clock.
      VarClockCount = VarClockCount + 2'b01;
```

```

else begin
    case (VarClockCount) // The comparator object.
        2'b00: AdjustFreq = 2'b11;
        2'b01: AdjustFreq = 2'b01;
        default: AdjustFreq = 2'b00;
    endcase
    VarClockCount = 2'b00;
end

```

虽然这样的一个PLL仿真没有问题，但是这种写法会导致综合工具产生锁存。

我们可以这么做来避免产生锁存：将上面的计数器分成两个2比特的计数器，一个被外部的PLL 1 MHz 的输入时钟驱动，另一个被PLL 内部的1 MHz 溢出计数器产生的时钟驱动。每个时钟到来时都把计数值进行比较，从而得到两个时钟的相对关系。只要两个计数值不相等，就会去调整VFO 的频率。

如图18.7所示，给外部输入ClockIn 增加延迟，这样，在PLL 内部时钟沿到来的时候，可以触发比较。这样就避免了时钟的竞争，同时保证了对于这两个1 MHz 的时钟来说，每次比较时都有足够的建立时间。不过要注意的是，这里增加延迟和PLL 内部采样使用延迟是全部不同的两个功能。

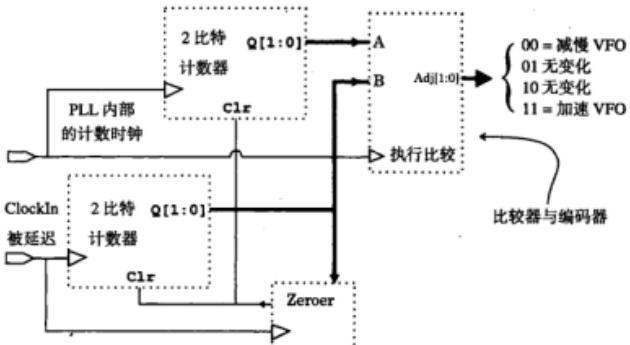


图18.7 可综合的，对边沿敏感的ClockComparator。虚线说明那是一个模块。图中没有画出复位信号

这两个2比特的计数器对32xPLL MultiCounter计数器产生的时钟输出（时钟频率约1 MHz）连续计数。在每一个PLL时钟的上升沿，对应边沿的计数值都被用来做比较，从而产生了调整VFO且及时更新的调整量。模块Zeroer不断地检测ClockIn的计数值，当计数值为3时，它同时把两个计数器清零。

VFO调整逻辑的实现可以使用嵌套的case语句。这个结构位于always块中，受PLL的溢出时钟MultiCounter驱动。

```

case (ClockIn.EdgeCount) // The external ClockIn edges.
    2'b00: case (VFO.EdgeCount) // The MultiCounter overflow edges.
            2'b00: AdjustFreq = 2'b01; // No change.

```

```

    default: AdjustFreq = 2'b00; // Slow the counter.
  endcase
2'b01: case (VFO.EdgeCount)
    2'b00: AdjustFreq = 2'b11; // Speed up the counter.
    2'b01: AdjustFreq = 2'b01; // No change.
  default: AdjustFreq = 2'b00; // Slow the counter.
endcase
2'b10: case (VFO.EdgeCount)
    2'b10: AdjustFreq = 2'b10; // No change.
    2'b11: AdjustFreq = 2'b00; // Slow the counter.
  default: AdjustFreq = 2'b11; // Speed up the counter.
endcase
default: case (VFO.EdgeCount) // Includes 2'b11 for initialization:
    2'b11: AdjustFreq = 2'b10; // No change.
  default: AdjustFreq = 2'b11; // Speed up the counter.
endcase
endcase

```

18.2.4 把设计升级为 400 MHz 1xPLL

上面所有的工作都是为我们的 serdes 工程设计一个输出 1 MHz 时钟，32：1 的 PLL。那么我们对在练习 6 之前的第 4 章中谈到的那个 400 MHz，不可综合的 1xPLL 还能做些什么吗？

对于这样的设计来说，如此高速的时钟是我们首先要考虑的问题。上一节中的内部计数器等逻辑都可以保留下来，但是需要减少 VFO FastClock 的延迟。因此，在 VFO 的振荡器中，应该使用 90 nm 工艺库里的高速器件。

把设计中的 `NumElems 由 5 改为 3，新的 FastClock 逻辑如下所示：

```

reg FastClock;
wire['NumElems:0] WireD;
//
generate
  genvar i;
  for (i=0; i<'NumElems; i = i+1)
    begin : DelayLine
      DEL005 Delay85ps (.Z(WireD[i+1]), .I(WireD[i]));
    end
  endgenerate
//
always@(Reset, WireD)
begin : FastClockGen
  if (Reset==1'b1)
    FastClock = 1'b0;
  else // The free-running clock gets the output of the delay line:
    FastClock = WireD['NumElems];
end
// The instantiated inverter:
INVDO Inv75ps (.ZN(WireD[0]), .I(FastClock));

```

上面使用到的 DEL005 是 TSMC 库里的元件，它的延迟为 85 ps。为简单起见，没有列出 set_dont_touch 编译指令。

可综合的 VFO 的代码如下所示：

```

always@(posedge ClockIn, posedge Reset) // ClockIn delayed for setup.
begin : FreqAdj
  if (Reset==1'b1)
    DivideFactor <= 'InitialCount;
  else begin
    case (AdjustFreq)
      2'b00: // Adjust f down (delay up):
        if (DivideFactor < DivideHiLim)
          DivideFactor <= DivideFactor + `VFO_Delta;
      2'b11: // Adjust f up (delay down):
        if (DivideFactor > DivideLoLim)
          DivideFactor <= DivideFactor - `VFO_Delta;
    endcase // Default: leave DivideFactor alone.
  end
end // FreqAdj.

```

其中，在产生 FastClock 的计数器里，`InitialCount 被设置成最大值的一半。

在更快一些的 1 倍设计里，比较器的逻辑稍为复杂一些：

```

always@(posedge PLLClock, posedge Reset) // The delayed PLL ClockIn.
begin : EdgeComparator
  if (Reset==1'b1) AdjustFreq = 2'b01;
  else
    case (ClockInN) // Count from the PLL external input clock.
      2'b00: begin
        case (PLLClockN) // Count from the VFO output clock.
          2'b00: AdjustFreq = 2'b01; // No change.
          2'b01: AdjustFreq = 2'b00; // Slow the counter.
          2'b10: AdjustFreq = 2'b00; // Slow the counter.
          2'b11: AdjustFreq = 2'b00; // Slow the counter.
        default: AdjustFreq = 2'b01; // No change.
      endcase
      end
      2'b01: begin
        case (PLLClockN)
          2'b00: AdjustFreq = 2'b11; // Speed up the counter.
          2'b01: AdjustFreq = 2'b01; // No change.
          2'b10: AdjustFreq = 2'b00; // Slow the counter.
          2'b11: AdjustFreq = 2'b00; // Slow the counter.
        default: AdjustFreq = 2'b01; // No change.
      endcase
      end
      2'b10: begin
        case (PLLClockN)
          2'b00: AdjustFreq = 2'b11; // Speed up the counter.
          2'b01: AdjustFreq = 2'b11; // Speed up the counter.
          2'b10: AdjustFreq = 2'b10; // No change.
          2'b11: AdjustFreq = 2'b00; // Slow the counter.
        default: AdjustFreq = 2'b10; // No change.
      endcase
      end
      2'b11: begin
        case (PLLClockN)
          2'b00: AdjustFreq = 2'b11; // Speed up the counter.
          2'b01: AdjustFreq = 2'b11; // Speed up the counter.
          2'b10: AdjustFreq = 2'b11; // Speed up the counter.
          2'b11: AdjustFreq = 2'b10; // No change.
        end
      end
    endcase
  end
end // EdgeComparator.

```

```

    default: AdjustFreq = 2'b10; // No change.
  endcase
end
default: AdjustFreq = 2'b10; // No change; allows initialization.
endcase
end

```

在Lab22_Ans目录里,有一个子目录PLL_1x_Demo,里面就是上面讲到的可综合的400 MHz、1xPLL的源代码。这个模块不能解析比振荡器的器件延迟更小的延迟时间(大约80 ps)。这说明它永远锁定不上一个漂移的时钟,只能逼近。

这个400 MHz设计的部分波形结果如图18.8至图18.11所示。

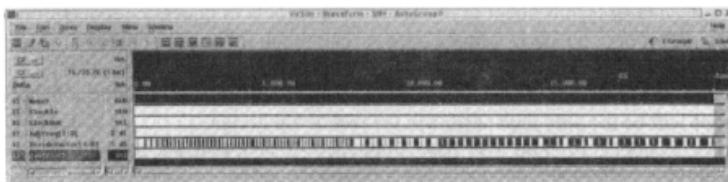


图 18.8 最初的 1x PLL 20 μ s 的仿真波形

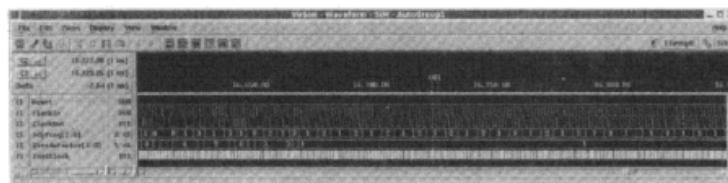


图 18.9 最初的 1x PLL 20 μ s 锁定时的波形

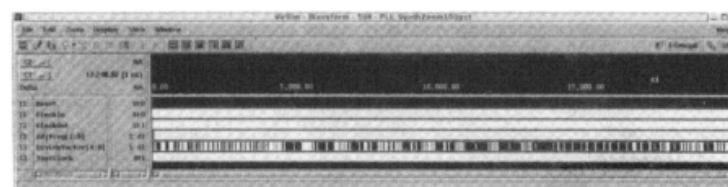


图 18.10 用可综合的 1x PLL 的网表进行 20 μ s 的仿真

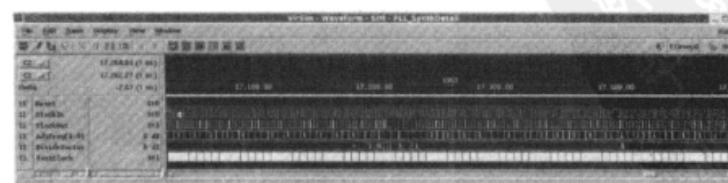


图 18.11 可综合的 1x PLL 20 μ s 的网表锁定时的波形

18.2.5 增强模块的可移植性

模块的端口名称非常重要，我们不应轻易地改变端口命名及定义，以保证工程的其他模块能正确调用这个模块。模块的端口名往往是设计者自行命名的，若在一个大型的设计里，需要例化的模块端口名和顶层的端口名有较大的差别，但直接修改需例化的模块又不方便的情况下，在需例化的模块上加一个符合命名需求的“壳”（wrapper）是一个简单有效的解决办法。不需要修改文件名，在功能模块之上再加一个新的模块壳即可。把对应的端口用连线连接之后，对设计中的其他模块来说，只有这个新的“壳”才是可见的了。

例如，假设你的一个已经写好了并且验证好了的模块的端口声明如下：

```
module MyModule(output[31:0] OutBus, ..., input ClockIn);
```

如果工程需要的输出端口叫“DataBus”，时钟叫“Clock”，只需要在MyModule.v这个文件里，把MyModule的名字稍为修改一下，并用这个新名字新建一个模块，在新模块里例化MyModule并把对应的线连上就可以了，下面是一个例子。

```
// -----
// This wrapper renames the MyModule ports as required:
module MyModule (output[31:0] DataBus
    ...
    , input Clock
);
    MyModule_WrapperU1 (.OutBus(DataBus), ... (I-to-I wiring) ...
    , .ClockIn(Clock));
endmodule
//
// -----
// Begin original MyModule design (notice the underscore in the name):
//
module MyModule_ (output[31:0] OutBus, ..., input ClockIn);
    ... (valuable, tested functionality) ...
endmodule
```

在一个大型的设计中，不同的模块应该被保存在各自独立的Verilog文件中。而wrapper和其对应的模块共存在一个Verilog文件之中是极少数的特例。

18.3 练习 22：串行序列解串器

请在目录Lab22下完成以下练习。

练习步骤

第1步。整理Lab21版本的PLL设计。目录Lab22里有一个答案子目录和一个到包含本节中使用到的所有Verilog模型的库文件。

进入Lab22目录，从旧的Lab21/Lab21_Ans目录下复制PLLSync目录下的所有内容(cp -pr)至新的Lab22目录。本次练习相对复杂一些，因此建议读者在实验的每一步尽量使用本书提供的文件而不要使用自己编写的代码。

练习21的文件结构如图18.12所示。

把新的Lab22/PLLsync目录下的所有内容都迁移到上一层(目录Lab22中)。在Lab22里,删掉Counter4.v这个不再使用的文件,再删掉PLLsync这个空目录。

把PLLsync.v重命名为PLLTst.v;把PLLsync.vcs改为PLLTst.vcs;并把PLLTop.inc移至目录Lab22。

重新整理后的Lab22中的文件结构应该如图18.13所示。

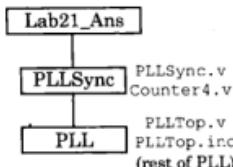


图 18.12 Lab21 PLL 里原来的文件结构

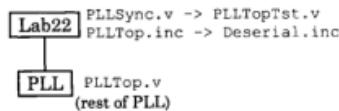


图 18.13 Lab22 PLL 重新整理后的文件结构

到这里,Lab22/PLL目录下的文件变成了解串器的内容。把PLLTOP.inc重命名为Deserial.inc,并把PLL/VFO.v里从PLLTOP.inc做的引用改为Deserial.inc。我们会在Lab22目录下运行VCS和DC,因此不要给VFO.v指定路径。

在目录Lab22中,按照下面的要求编辑PLLTst.v:

- 删掉所有的“Step *”定义,并把`includeg改成`include“Deserial.inc”。
- 删掉旧PLLsync模块里除了产生采样脉冲的always块,我们会把它移到testbench里去。
- 把testbench的名称改为PLLTst。
- 把testbench里待测的模块名改成PLLTOP,它的一个输出端口为.ClockOut;连在这个端口上的1比特宽的线名为.PLLClockWatch。
- PLLTOP还应有一个名为.Reset的复位端口,一个名为.Sample的输入端口,刚才提到的那个always块产生的采样脉冲来驱动它。

编辑文件PLLTst.vcs,使它只包含文件PLLTst.v及其在PLL子目录下对应的目录名。

现在,在Lab22/PLL目录下,我们有了PLL设计的所有文件。在Lab22目录下还有一个单独的testbench文件。对应的include文件是Lab22/Deserial.inc。

粗略地运行仿真,只要Lab22目录下的设计可以运行就行了。

第2步。验证当前的PLL设计不能综合成一个正确的网表。

进入PLL子目录。从练习15的目录里把.sct文件复制至Lab22/PLL目录中。修改这个.sct文件,以便用它来综合这个PLL设计。到目前为止,这个PLL设计应该包含PLLTOP.v,MultiCounter.v,ClockComparator.v和VFO.v。把这个.sct文件重命名为PLLTOP.sct,然后更改这个脚本使得综合后的网表被自动保存为PLLTOPNetlist.v。综合这个PLL。综合可以顺利的结束,但是综合出来的网表是有问题的。

回到目录Lab22,用这个综合出来的网表进行仿真,看看结果是不是有问题。我们在这个目录里新建一个名为PLLTOPNetlist.vcs的仿真文件,它的内容如下所示:

```
PLLTst.v
PLL/PLLTst.v
-v verilog_library_2001.v
```

Verilog_library_2001.v是Verilog模型的库文件(在本书附带的光盘的misc目录下有这个文件);作者已经修改了这个文件,使得延时值更符合这里的设计需要。选项-v表明这是一个库文件,这意味着仿真工具只编译这个设计中用到的库元件,而不去编译整个文件。但这个VFO的网表仍然是不正确的,如图18.14所示。

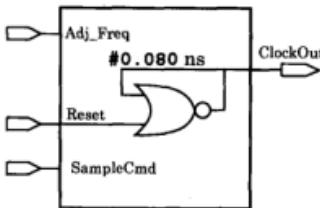


图18.14 VFO综合出来的网表仍然是不正确的。由于敏感变量表不全,导致Adj_Freq和SampleCmd两个端口没有被连接

第3步。重新设计可综合的PLL。

由本章前面的讨论,你应该可以重写出ClockComparator和VFO模块。

PLL的Sample输入端口应该被去掉,我们可以在PLLTst里让ClockIn通过一个延迟单元,并把它直接送给VFO作为Sample脉冲。请从Verilog库中选择对应的延迟单元的仿真模型。在综合脚本里加上set_dont_touch的编译指令来保证这些延迟单元不会在综合时被优化掉。

从ClockComparator送给VFO的AdjustFreq应该被改成:2'b00时减速,2'b11时加速为其他值的时候维持原值。这会使得VFO的频率调整有些滞后。

需要注意的是代码里的敏感变量列表里有多个信号的边沿条件存在,对应的电路会受到这种条件的影响。在综合的时候,综合工具会明确地报告出这一类的违例。

如果你认为这个练习的时间会超过1~2小时,可以把PLL下的子目录Lab22_Ans的所有文件都复制到Lab22/PLL目录下直接使用。

我们人为放置的那个延迟单元的延迟会小于100 ps,因此,为了得到准确的仿真时序,请把.inc文件里的`timescale改成1 ns/1 ps。虽然更高的时钟分辨率使得仿真变得慢了一些,但是这样才能准确计算延迟单元对设计带来的影响。

第4步。验证目前的PLL可以综合出正确的网表。

综合PLLTst并做网表仿真,VCS的仿真环境使用Lab22下的即可。如果这个网表正确的(参见图18.15),你会看到PLL网表产生的时钟和直接用源代码产生的时钟是一样的。

第5步。整理这个解串器的设计。下面将完成图18.16所示的串并转换器。

进入目录Lab22。以PLLTst*开头的文件名都没有用了,删掉它们。在Lab22里,新建一个Deserializer的文件,其对应的模块名也叫这个名字。再另外新建一个空的testbench,叫做DeserializerTst。



图 18.15 可综合的 PLL 的网表仿真结果

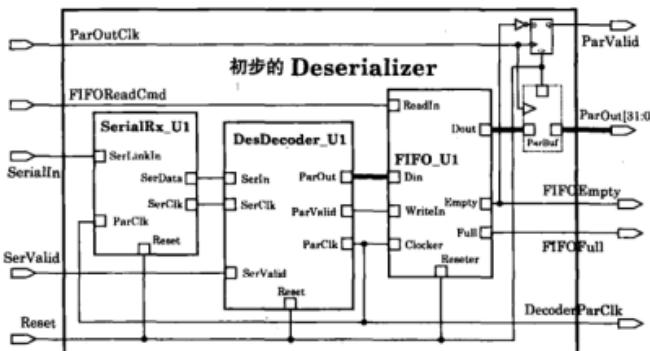


图 18.16 Deserializer 初步的完整电路图

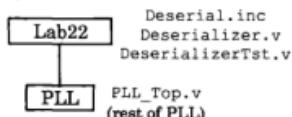


图 18.17 Deserializer 的文件结构

在 Deserializer 模块里，例化出如下三个实例：FIFO (FIFO_Top)，DesDecoder (练习 16 中完成的) 和一个名为 SerialRx 的新的串行接收模块，把它们的端口空着就是了。文件组织结构应该如图 18.17 所示。

由图 18.1 的数据流和图 18.16 的电路图可知，Deserializer 有名为 SerialIn、ParOutClk 的输入，有一个名为 ParOut 的 32 比特数据总线输出给 FIFO_Top。在你的代码里产生这些端口。

DesDecoder 的输出端口 ParClk 和 FIFO_Top 的 Clocker 输入端口连在一起，为了避免将发送端的 1 MHz 时钟和接收端的接近 1 MHz 的并行时钟混淆，把这根线命名为 DecoderParClk (“decoder's parallel-bus clock”)。

对于 FIFO 来说，有 FIFOFull 和 FIFOEmpty 两个输出，它们会被送至顶层并送出去。虽然在这个练习里我们不需要用到这两个端口，但是如果其他的系统要用到我们这个 serdes 设计，就会用到它们。

一个外部 SerValid 作为 Deserializer 的输入被送至 DesDecoder 模块，Deserializer 还应输出 ParValid 信号。最后，Deserializer 还应有复位“Reset”信号。

在 Deserializer 里声明参数 AWid (地址宽度)，默认值为 5，对应于 32 比特。

在把 Deserializer 的数据输出位宽 DWid 也声明成参数，默认值为 32。在整个设计里，只要有这个数据总线，都应该用这个参数去说明它的位宽。虽然在这个设计里不会更改这个参数的具体数值，比如说将输出总线位宽改为 16 比特或 64 比特，但我们仍将 DWid 表示成 2 的整次幂形式。这样的话，如果将来 Serializer 的输入位宽需要更改，从而串行通道两端的位宽可以做到相互独立，我可以很方便地修改 DWid 的位宽。这对设计的扩展来说是个非常好的特性。

testbench 文件 DeserializerTst.v 应该包含文件 Deserial.inc 中的全局时序定义，因此在 testbench 里加入 include 语句。

第6步。FIFO。在目录 Lab22 下新建一个子目录 FIFO，把 Lab11/Lab11_Ans 中的 FIFO 设计完整的复制过来，应该包含以下文件：FIFO_Top.v, FIFOStateM.v 和 Mem1kx32.v。删掉这些文件里所有的 timescale 定义。

给 FIFO_Top 加上参数 AWid 和 DWid。如前所述，只要用到这个位宽的总线都应使用这个参数来指定位宽。把 FIFO 和其子模块（FIFOStateM 和 Mem1kx32）的位宽、深度都用参数替换掉。

给 FIFO_Top 增加 1 比特宽的 Full 和 Empty 端口。把它们和第 5 步 Deserializer 增加的新端口连起来。删掉 E_FIFO 和 F_FIFO 这两根线。再删掉 ReadWire, WriteWire 和 ResetFIFO 这三根线，用与端口名字对应的线来替代它们。如果 Mem1kx32 使用的时钟经过了反相（“..., .ClockIn(~Clocker), ...”），去掉取反的符号，让这个 memory 受时钟的上升沿驱动。

FIFO_Top.v 还包含一个 testbench。和刚才一样，把 AWid 和 Dwid 也设置成 testbench 的参数。保证 FIFO_Top 的 .Full 和 .Empty 这两个输出和 testbench 的对应端口正确连接。Testbench 应该包含了 Deserial.inc。把 testbench 的部分单独保存在一个名为 FIFO_TopTst.v 的文件中，将来的调试会用到这个文件。

如果还有其他注释掉的 Mem1kx32.v 和 FIFO_TopTst.v 的 testbench，都把它们删掉。万一这样造成了错误，只要从以前的练习目录里把缺少的 testbench 复制过来就可以了。

给 Mem1kx32 增加 Reset 端口。Reset 的作用是把这个 memory 里所有的存储单元都清零，这样，将来做仿真的时候就不会有奇偶校验的问题了。

最后，简单地用 FIFO_TopTst 做仿真，只要引用的文件和 FIFO 的大致功能是对的就行了。

第7步。Deserializer 的文件结构。在目录 Lab22 里新建子目录 DesDecoder 和 SerialRx，分别用来放解串和串行接收的代码。

把目录 Lab16_Ans 下的 DesDecoder.v 复制至目录 DesDecoder。删掉 DesDecoder.v 里的 timescale 定义。把参数 DWid 传给这个模块。把输出总线的名字由 ParBus 改成 ParOut，把端口 ParRst 改成 Reset。如果这时 testbench 还没有被删掉，删掉它。

新建名为 SerialRx 的目录并新建一个名为 SerialRx.v 的文件。这个文件里应有如下的内容：

```
module SerialRx(output SerClk, SerData
                  , input SerLinkIn, ParClk, Reset
                  );
    assign SerData = SerLinkIn;
    //
    PLLTop PLL_RxUI (.ClockOut(SerClk), .ClockIn(ParClk), .Reset(Reset));
    //
endmodule // SerialRx.
```

这段代码说明了我们将会在模块 SerialRx 中例化 PLL 设计。

如图 18.18 所示，目录 Lab22 下应该有 DesDecoder, FIFO, PLL 和 SerialRx 4 个子目录。在后面的练习中，Lab22 将会被改名为 Deserializer。

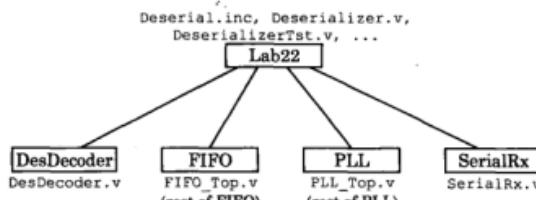


图 18.18 Lab22 的文件结构

在模块 Deserializer 里，例化模块 FIFO_Top，并且按照图 18.1 的数据流把它和实例 DesDecoder 和 SerialRx 都连起来。

对于 DesDecoder 和 SerialRx 这两个模块，它们端口的位宽和外部要传递的参数密切相关。我们可以这么做：在 Deserializer.v 里，先忽略模块的声明，把模块按照例化和端口映射的格式先列出来。看起来应该是下面的样子：

```

module Deserializer
...
// -----
// Structure:
//
FIFO.Top #( .AWid(AWid), .DWid(DWid) )
FIFO.U1
(
    .Dout(), .Din(),
    .ReadIn(), .WriteIn(),
    .Full(), .Empty(),
    .Clocker(), .Resetter(Reset)
);
//
DesDecoder #( .DWid(DWid) )
DesDecoder.U1
(
    .ParOut(), .ParValid(),
    .ParClk(), .SerClk(),
    .SerIn(), .SerValid(), .Reset(Reset)
);
//
SerialRx
SerialRx.U1
(
    .SerClk(), .SerData(),
    .SerLinkIn(), .ParClk(),
    .Reset(Reset)
);
//
endmodule // Deserializer

```

接着，按照例化时的格式来完成模块的声明。例如，把 FIFO.Top .Din() 改写成 output [DWid-1:0] .Din(DecodeToFIFO)，把 DesDecoder .ParOut() 改写成 output [DWid-1:0] .ParOut(DecodeToFIFO)。

这种由上至下定义和声明模块端口的办法在大规模电路设计中是一种重要的技巧。

换句话说，这是一种例化的地方就声明模块的办法。比如说，要在 Deserializer.v 里例化 FIFO，把下面这段代码复制到 Deserializer.v 中。

```
module FIFO.Top #(parameter AWid = 5 // FIFO depth = 2AWid.
                  , DWid = 32 // Default width.
                  )
  ( output [DWid-1:0] Dout(wire [DWid-1:0] FIFO.Out)
  , input [DWid-1:0] Din(wire [DWid-1:0] DecodeToFIFO)
  , output Full(wire FIFOFull), Empty(wire FIFOEmpty)
  ...
);
```

上面的代码在端口的声明里包含了位宽，这种写法是不符合语法规规的。显然，这只是一个中间步骤。

把这些信息复制进入每个模块各自独立的文件中。然后，把不符合语法的地方都去掉。

最后，修改 Deserializer 模块里例化部分的代码，把它改成这样：

```
FIFO.Top #( .AWid(AWid), .DWid(DWid) )
FIFO.Top.U1                                // Instance name.
( .Dout(FIFO.Out)                         // pin-out (= port map).
, .Din(DecodeToFIFO)
, .Full(FIFOFull), .Empty(FIFOEmpty)
...
);
```

再回到子模块里，加上复位和其他有用的信号，例如 FIFO 的读，写或空等信号。

例化的所有子模块都必须使用 Deserializer 定义的位宽参数，深度参数只传给 FIFO。

完成 DeserializerTst.vcs 文件，用仿真器载入 Deserializer.v 并编译，只需检查连线没有问题即可。如果发现了问题，请自行修改。

第8步。Deserializer 的并行输出。

在顶层 Deserializer 里定义一组名为 ParBuf 的寄存器，把它作为顶层的数据输出（参见图 18.16）。

PLL 从串行数据流中提取出了 1 MHz 的并行数据时钟。按照数据格式，64 比特里的 32 比特才是原始数据，因此，并行数据的发送速度是 500 kb/s。

0.5 MHz 的 ParOutClk 把数据打入缓冲区。

```
always@ (posedge ParOutClk, posedge Reset)
begin : OutputBuffer
  if (Reset == 1'b1)
    ParBuf <= 'b0; // To be wired to the ParOut port.
  else ParBuf <= FIFO.Out;
end
```

可以在 testbench 里产生 ParOutClk。上面的代码说明，如果 FIFO 被复位，FIFO 的输出数据是 0。还有另外一个信号 ParValid，可以把它放到和 ParBuf 一个 always 块中进行处理。

第 9 步。串并解码器。进入 DesDecoder 子目录，把 Lab16/Lab_Ans 里的 DesDecoder 复制过来。

把 Lab16_Ans 里的 DesDecoder.v 另存为 DesDecoderTst.v，删掉设计源代码，只在文件中保留 testbench。把时间精度 (timescale) 改成 1 ns/1 ps，删掉 DC 的编译向导语句，增加对 DWid 的引用。

在新的 DesDecoderTst.v 里，如果实例 DesDecoder 是先移串行数据的 LSB，把顺序改过来，先输出 MSB。删掉 testbench 里所有除了 initial 块中的和串行时钟产生逻辑的延迟。

按下面的要求修改 DesDecoder.v：

- 增加 1 比特输出端口：WriteFIFO。
- 删掉所有的 #delay 表达式，并删掉相关的注释。
- 把模块里的 4 个 always 块合并成 2 个：ClkGen 的敏感信号是 SerClock 或 Reset；第二个 always 块如下：

```
always@{negedge SerClock, posedge Reset}
begin
    Shift1;
    Decode4;
    Unload32;
end
```

为了避免和串行数据移位的竞争，这里使用的是时钟 SerClock 的下降沿。

- 用一个简单的 if 来检查 ParClk==1'b0 的情况。删掉 Unload32 这个为了练习 16 而准备的 task。
- 在 ClkGen 这个 task 里，除了复位，把其他所有的 if 都放到 if (SerValid==YES) 这个条件下。

完成以上的步骤后，修改 DesDecoderTst.vcs，并单独用 DesDecoder 仿真。检查新的 DesDecoder testbench 的连线是否正确。除了综合会有点问题，这个模块已经几乎可以用了。

第 10 步。DeserializerTst 的 testbench。我们在练习 16 的 DesDecoder 的 testbench 的基础上来改写这个 testbench。主要的问题在于：练习 16 里的 DesDecoder 的 testbench 需要串行时钟，而 Deserializer 不需要串行时钟。在这个练习里，Deserializer 模块需要从串行数据流中提取中并行时钟，并且用这个并行时钟去产生和输入串行数据流同步的串行时钟。

作为一个初步的版本，我们一开始不会对Deserializer 进行过于复杂的测试。这个 testbench 会用非常接近 PLL 基本速率的时钟去产生串行数据流。这样的串行数据让我们能够很容易地去验证这个设计的各个功能。在后面的章节里，我们将会使串行数据流远离这个基本速率。

完成了上面的修改后，这个 testbench 能够和在 Lab16 里的一样产生符合我们要求的串行数据了。设 testbench 里的串行时钟的半周期是 15.6 ns。这和 PLL 的时钟很接近了，PLL 输出为 $500/32 \text{ ns} = 15.625 \text{ ns}$ ，经 Verilog 整数截取后是 15 ns。在目前的这个设计中，如果时钟偏离 15 ns 越远，说明时钟同步越差，Deserializer 就会丢掉更多的串行数据。

用 testbench 产生一个独立的 1/2 MHz 的时钟，基于这个时钟从 Deserializer 的 FIFO 输出寄存器读数据。在 testbench 中例化 Deserializer 并仿真。仿真结果应该如图 18.19 和图 18.20 所示，虽然同步过程造成了一些数据丢失，但是仍然能从 FIFO 中获得一些正确的数据。

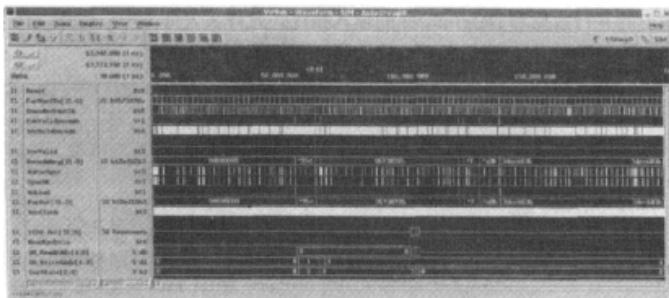


图 18.19 Deserializer 的仿真结果

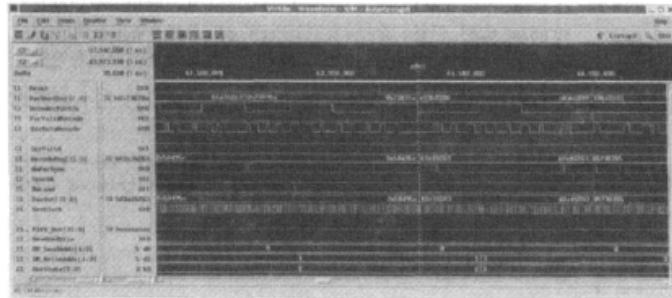


图 18.20 局部放大的 Deserializer 的仿真结果

第11步。仿真检查。如果你的Deserializer已经多多少少可以工作了(可以产生并行数据时钟并且偶尔能够正确地输出并行数据),那么,接着做下面的步骤:

A. 删掉 DesDecoder 里的多余的 WriteFIFO 输出端口。什么时候写 FIFO 是由串并输入功能决定的，从当 FIFO 满了之后，什么时候停止发送串行数据是由外部的系统来决定的。把 DesDecoder 的 ParValid 输出端口在 FIFO_Tap 的 WriteIn 输入端口连起来。

B. 仿真并观察结果。因为这个FIFO并没有收到读请求，因此它会很快的被填满。

FIFO 填满之后，把它一次清空。为了实现这个功能，在 testbench 里需要增加读命令，可以通过断言 FIFOReadCmd 来完成这个功能，如图 18.21 所示。

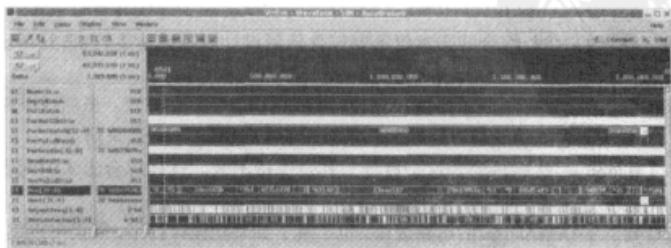


图 18.21 改进后的 Deserializer 的仿真结果

C. 把输出的并行数据和 Deserializer 的输入相比，允许丢失部分串行数据。图 18.22 是最后的结果。

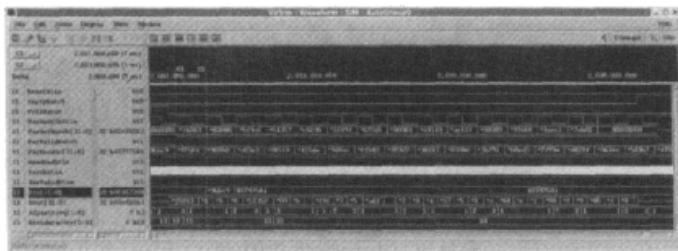


图 18.22 在 $t = 63\ 942\ 550\ ns$ 时，Deserializer 的 Din 输出的第一个数据是 $32'h62ef_6263$ 。但是，当它被复制到 ParWordIn 总线并被保存到 FIFO 时，数据变成了 $32'h61ef_6263$ 。显然，这个设计还需要接着升级

目前只保证能从 FIFO 中读出写入的数据（分两个地址写入）即可，先不要考虑特殊情况。回忆第 8 步，读并行输出的代码应该如下所示：

```
always@(posedge ParOutClk) // 1/2 MHz clock domain of the receiving system.
  if (Reset==1'b1 || F.Empty==1'b1)
    begin
      ParBuf <= 'b0; // Zero the output buffer.
      ParValidReg <= 1'b0; // Flag its contents as invalid.
    end
  else begin // Copy data from the FIFO:
    ParBuf <= (FIFO.ReadCmd==1'b1)? FIFO.Out : 'b0;
    // Flag the buffer value validity:
    ParValidReg <= FIFO.ReadCmd && (~FIFOEmpty);
  end
```

还有两个问题，我们将来再讨论：

(a) FIFO 的 0x00 和 0x1f 地址被读写到了吗？(b) 可以同时读写 FIFO 吗？这样的话，FIFO 就不容易被填满或被读空了。

第 12 步。综合。综合整个 Deserializer 设计。我们综合这个设计目的只是为了说明这个设计可以正确地被综合。也许设计中有的代码需要修改或从 `ifdef DC 块中移出来。

PLL 是可综合的了，但 FIFO 还不是。FIFO 的网表里会缺少连线并且有很多悬空的输出。

为了减少综合的时间，综合的脚本里不要包含任何设计规则或速度约束，只指定按面积最小的原则来综合。

按照保留层次关系综合出来的网表大概包含 35 000 个等效晶体管。虽然这个网表看起来是完整的，但实际上它的功能是有问题的，所以没有必要用这个网表来仿真。

先打平 (flatten)，然后再综合这个设计，其余保持不变。这样综合出来的网表会去掉未连接的模块间的互连逻辑，从而使得网表更小一些。

我们现在知道了 Deserializer 的电路规模，也了解了如果把设计打平，规模可以缩小的程度。

18.3.1 练习后的思考

最小面积和最小延迟的综合约束应该分别怎么设置?

综合约束和设计规则的区别是什么?

思考本地延迟、面积最优和约束。

什么是增量编译?

18.3.2 补充学习

(可选)复习 Thomas and Moorby (2002)的 5.1 节和 5.2 节关于参数和例化的内容(不用看 defparam)。



第 19 章 升级解串器

19.1 并行解串器

我们将在本章里升级解串器的设计，并且将整合一个功能略有缺陷但模块完整的串行接收器。

如图 19.1 所示，由这个电路示意图可知，最主要的改动是 FIFO 被升级成支持并行读写的操作（双口，dual-port）。

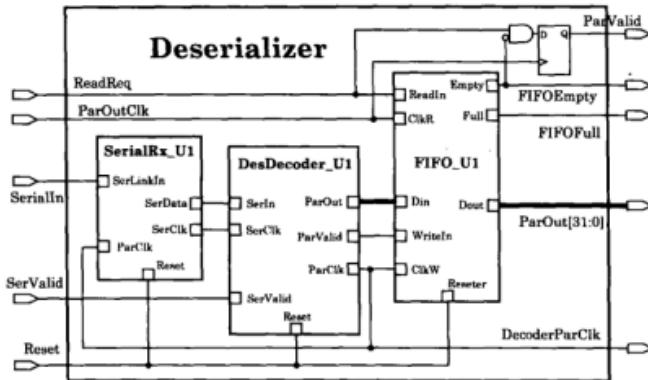


图 19.1 并行解串器的电路示意图。其中，FIFO 在两个时钟域中

为了完成这个升级，首先要把 RAM 改成双口，然后还需要升级 FIFO 的状态机（FIFO 控制器），使得它可以根据 memory 的使用情况处理当前时钟读和写的操作。读和写发生在时钟的某个边沿，地址和状态的改变发生在对应的反相沿。我们还会把 Deserializer 修改成可综合的设计，虽然有个别功能可能会有点问题。

19.1.1 双口 memory

双口（dual port）memory 可以处理同时（同一个时钟）读请求和写请求。双口 memory 的读写地址必须不相同，这是 FIFO 控制器而不是 memory 的要求。读和写操作可以来自不同的时钟域。在这里，把我们要用的这个新 memory 叫做“DPMem1kx32”。

下面是要做的工作：

- 把 FIFO_Top 的 Mem_Enable 信号置为“1”，使得这个寄存器阵列（register file）总是有效的。这是因为我们要一直使用这个 memory，并不是因为要把它升级成双口 memory 才改它。

- 给 memory 增加独立的读写地址端口。
- 给 memory 增加独立的读写时钟端口。
- 升级读写逻辑，使得两个操作无关。

19.1.2 异步 FIFO 的状态机

这比升级双口 memory 要复杂，归纳如下：

- 增加独立的读写时钟（和 FIFO 共享）。
- 支持同时的读写操作。
- FIFO 控制器只会根据读或写操作跳转一次状态。FIFO 的读写操作可能是异步的，因此，这两个时钟都有可能导致状态的跃迁。
- 增加防止一次读或写多个地址的保护逻辑。
- 状态机状态的跃迁由来自不同时钟的读写操作决定。

剩下的内容将在练习里学习。

19.1.3 可综合的 FIFO

为了升级双口 FIFO，综合会遇到的主要问题如下：

- 设计中的延迟信息对综合是没有用的。
 - 删掉这些延迟。
- 任务 incrRead 和 incrWrite 包含边沿敏感的控制逻辑，但是它们在变敏感信号的组合逻辑块中被调用。
 - 分两步来解决这个问题：
 - (a) 首先，要保证在一个时钟内，这些函数只能被调用一次。其余的问题先留着。
 - (b) 删除任务的事件控制逻辑，拆分任务，把它们放到一个外部的 always 块里去。这样就可以综合了。
- 状态机时钟产生器包含一个有冗余敏感变量的 always 块。综合的时候可能会形成锁存。
 - 状态机的时钟将直接来自于相互独立的读写时钟。
- 除了状态寄存器，计数器的计数也是时序逻辑。因此，不能简单的把状态机分成状态寄存器（时序逻辑）和组合逻辑两部分。这里的组合逻辑有可能导致综合出锁存。
 - 我们会把状态机里的组合逻辑升级成时序逻辑，并且用与状态跃迁使用的时钟沿相反的时钟沿。

19.1.4 可综合的解串器

- 删掉 Shift1 always 块中的临时寄存器。
- 重写任务 ClkGen，使其成为一个上升沿有效的 always 块，这样能有效避免综合出锁存。
 - 把 SerClock 门从 ClkGen 移到一个组合逻辑块里去。
- 把旧 DesDecoder 任务里的阻塞赋值都改成非阻塞赋值。
- 通过在 Unload32 always 块中增加小计数器（ParValidTimer）延长 ParValid 的有效时间，保证至少 8 个时钟。

19.2 练习 23：解串器

在目录 Lab23 下完成这个练习。

练习步骤

第 1 步。复制 Lab22 的 Deserializer。在目录 Lab23 下新建一个子目录 Deserializer。把目录 Lab22/Lab22_Ans/Lab22_Ans_Step12 里的内容都复制到 Deserializer 子目录。

把复制过来的网表和 log 文件都删掉。

目前的 Deserializer 设计包含一个可综合的 PLL，一个不能同时读写的单口 RAM 和一个不能被综合的 FIFO。如果这个目录里没有 Verilog 的仿真库（库名_v2001.v），把它链接过来。如果你用的是 Windows 操作系统，直接把它复制过来。

用 Lab22 的 testbench 给 Deserializer 简单地做一下仿真。PLL 的同步功能可能会工作不正常，但是整个设计大致看起来还是可以工作的。

第 2 步。Deserializer 的边界情况（corner case）和 FIFO 的调试。在的 Deserializer/FIFO 目录里，有 FIFO，FIFO 的 testbench，还有仿真用的.vcs 文件。

把 FIFO 读写的间隔改得足够长并仿真。先不管数据的具体数值是什么样的，仿真地址 0x00 和 0x1f 被同时读写时 FIFO 有什么样的响应。如果 0x00 或 0x1f 这两个地址在仿真里始终没有被同时读写，修改模块 FIFOStateM 来达到这个目的。最后修改 testbench，同时读写这个 FIFO。仿真并观察结果。

在本练习里，FIFO 会被重新设计。首先删掉 FIFO 所有模块里所有的 #delay 表达式。然后，为了区别新旧文件，把所有文件名里的下划线去掉，并修改相应的模块名。Deserializer/FIFO 目录里的文件应该如下：

```
default.cfg  
FIFOTop.v  
FIFOTopTst.v  
FIFOTopTst.vcs  
FIFOStateM.v  
Mem1kx32.v
```

文件中所有的地址宽度在练习 22 里已经用 AWid 替换了。文件 default.cfg 只有 VCS 在用它，不重要。

第 3 步。同时读写 memory。在对 Deserializer 的仿真里，我们已经看到当 FIFO 已满时，写请求被忽略掉；当有读请求时，写请求也被忽略了。如果你没有在仿真里观察到这个现象，重新再仿真看看。

我们的 FIFO 里用的 Mem1kx32 只有一条地址总线，不支持同时读写。单口 RAM 调试起来更容易一些，但是它会限制 FIFO 的性能。

这个 FIFO 控制器的逻辑是读优先，也就是说，同时读写时，读有效而写无效。我们要改掉这个逻辑。首先删掉 FIFOTop.v 里的线 Mem_Enable。把实例 Mem1kx32 的 ChipEna 端口改成 1'b1，使 memory 一直都有效。由于 FIFO 里只有这一个 RAM，因此我们可以不用这个片使能信号。

虽然使能一直都有效了，但是这个 memory 仍然不支持同时读写，因为：(1) 地址线还是只有一套；(2) Mem1kx32 里的 if-else 逻辑使得读和写是互斥的。另外，在 FIFO Top 里的逻辑也使得读的优先级高于写。

显然，我们需要重新设计这个 memory。

请记住，当 ParValidDecode 被 DesDecoder 置为有效时，Deserializer 就会产生一个 FIFO 的写请求。

第 4 步。 双口 FIFO 的 memory。单口 Mem1kx32 的 I/O 如图 19.2 所示。

把 Mem1kx32.v 改名为 DPMem1kx32.v (DP 意为双口 “Dual-Port”), 并修改对应的模块名。同时也请修改 FIFO Top.v 和.vcs 中的相关部分。

编辑模块 DPMem1kx32, 给它增加第二套地址端口：把原有的地址输入端口改成 AddrR, 再加上一套 AddrW。宽度都由 AWid 指定。

同上，也这样修改时钟端口，把原有的时钟叫做 ClkR，同时增加 ClkW。结果如图 19.3 所示：

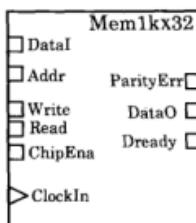


图 19.2 单口 Mem1kx32 的 I/O

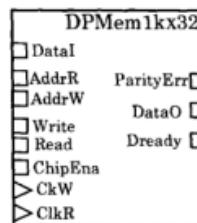


图 19.3 把 Mem1kx32 升级成双口 memory

在模块内部，把 ChipClock 改名成 ClockR，并且声明一条新的连线：ClockW。

虽然在 FIFO 中，我们不再使用 ChipEna，但是我们希望这个片选在 memory 内部还是有效的。可以这么做：用输入 ChipEna 来控制两个时钟，做两个 mux，无效时输出 1'b0。

```
assign ClockR = (ChipEna==1'b1)? ClkR : 1'b0;
assign ClockW = (ChipEna==1'b1)? ClkW : 1'b0;
```

图 19.4 是门控时钟 ClkR 的示意图，两个时钟的电路显然是一样的。同时，也请保留 ChipEna 控制的 Dready 和 DataO。

读和写到目前仍然是独立的。把读和写逻辑分成到两个 always 块中去，后面有代码。由于我们已经可以通过条件操作符来打开和关闭时钟，读和写这两个 always 块里的 ChipEna 就可以去掉了。

从硬件的角度来说，先把 memory 的内容清 0 是没有必要的。但是对于奇偶校验来说，未定的初始值可能会产生 “x” 或 “z”。因此，出于奇偶校验的需求，我们需要用 Reset 去初始化 memory。复位信号 Reset 会被送到读和写的两个 always 块，但是只有写逻辑才能初始化 memory。请务必记住不要在多个 always 块中操作一个变量，因为这样很有可能会产生竞争条件。

这样的话，分离的读和写 always 语句块应该写成下面的形式：

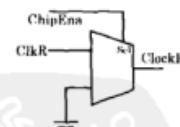


图 19.4 ClkR 的门控逻辑

```

...
always@(posedge ClockR, posedge Reset)
begin : Reader
if (Reset==1'b1)
    (init Parityr, Dready, DataOr)
else if (Read==1'b1)
    (do parity check and read)
end // Reader.
//
always@(posedge ClockW, posedge Reset)
begin : Writer
if (Reset==1'b1) // Zero the memory:
    for (i=0; i<MemHi; i=i+1) Storage[i] <= 'b0;
else if (Write==1'b1)
    (do write)
end // Writer.
...

```

按上面的代码把这两个独立的 always 块命名为 Reader 和 Writer。Reader 块要进行奇偶校验，因此，奇偶校验错误输出应该被复位成 0。而 Writer 块需要初始化 memory，即是给所有的地址都写 0。当写无效时，不应向 memory 写 “z”（如果读写是共享一套双向的总线，这样是可以的）。

在网表里，综合工具总是把寄存器的名字加上 “*_reg”的后缀。所以，按我们目前的命名习惯，寄存器的名称总是以 “Reg” 结尾，则网表里的名字就是 *Reg_reg。这显然是多余的。因此，在 DPMem1x32 和其他的 FIFO 模块里，把所有的寄存器都由 *Reg 结尾改成由 *r 结尾。

删掉 FIFO Top 里 SM_MemAddr 的声明和赋值。把新增的两套输入地址总线和状态机对应的输出连在一起。再把 DPMem1kx32 的两个时钟都连上正确的时钟。简单验证一下这个例化了 DPMem1kx32 的 FIFO。

接下来，FIFO Top 和 DPMem1kx32 无须再修改，我们需要更新状态机的设计，因为它所包含的代码只支持单口 memory 的操作。

第 5 步。更新 FIFO 状态机 I 的设计。这个过程比较复杂，我们分两个阶段进行。

首先，进入 Deserializer 目录，把所有对 FIFO_Top 的引用都改成引用 FIFO Top，把所有对 Mem1kx32 的引用都改成引用 DPMem1kx32。然后仿真。我们看到 ParValid 信号变高，说明有串行数据需要写。但是，却没有产生写操作。这是因为 FIFO 的状态机用 if-else 把读的优先级置于写之上，从而避免了同时读写的竞争。

我们现在已经有了两条读写的总线，下一步来把状态机中的读写功能分离开，使得支持并发的读写操作。

进入 FIFO 子目录。在 FIFOStateM 中，把端口 Clk 改为 ClkR，再增加另外一个时钟输入端口 ClkW。然后，在同一个模块中将两个地址增加任务都修改成锁存信号，以保护地址在每个时钟周期里最多只能变化一次。例如，读地址任务该被改为如下形式：

```

reg LatchR, LatchW; // Reset these to 1'b0 in SM comb. block.
...
task incrRead; // Unsynthesizable!
begin
if (LatchR==1'b0)
begin

```

```

LatchR = 1'b1;
@(posedge ClkR)
  ReadCount = ReadCount + 1;
LatchR = 1'b0;
end
end
endtask

```

将 FIFOStateM 中的两个任务：incrRead 和 incrWrite 都照上面的方式进行修改。

还有一个很关键的问题：除非只有一个时钟，否则状态机的状态是不确定的。但是，模块 FIFOStateM 有两个时钟输入。因此，需要利用这些输入来产生一个时钟。用一个简单的，对变化敏感的 always 块来完成这个工作：

```

always@{ClkR, ClkW, Reset}
  if (Reset==1'b1)
    StateClock = 1'b0;
  else StateClock = !StateClock;

```

上面的写法是不可综合的（ClkR 和 ClkW 都没有用到），暂时不去改它。做仿真的时候，可以把这个 StateClock 当做时钟。声明一个名为 StateClock 的寄存器，并把它和上面的代码一起加在 FIFOStateM 的前面。把状态机的时钟从 Clk 改成 StateClock，让状态机的组合逻辑块在 StateClock==1'b0 时有效。同样地，把 incrRead 和 incrWrite 任务里的上升沿有效改成 StateClock 有效。

其次，在状态机的 empty 和 full 状态里，去掉读写操作相互作用的代码。特别是在 empty 状态里，没有必要再用 if 去检查读请求；在 full 状态里，if 只用来检查读请求。

a_empty 和 a_full 的代码暂时不去修改。

在 normal 状态里，可以通过计数器来检查并发的读写操作引起的状态跃迁。因此，在这个状态里，把读和写的条件都放到各自的 if-else 里去，使得两个操作互不相关。请确认所有相关的操作都被删掉，比如说下面的代码：“WriteCmdr = 1'b0;”就应该从读逻辑中被删除。

同时，从这两个 if 里提取出所有的状态跃迁逻辑并放到另外的 if 条件中。因为在一个时钟里可能会有读写两个请求，我们需要等到最后一个请求结束后才能决定跃迁到哪个状态。

修改后的状态跃迁代码应该如下：

```

(read & write command logic)
// Set the default:
NextState = normal;
//
// Check for a.full (R == W+1):
tmpCount = WriteCount+1;
if (ReadCount==tmpCount)
  NextState = a.full;
//
// Check for a.empty (W == R+1):
tmpCount = ReadCount+1;
if (WriteCount==tmpCount)
  NextState = a.empty;

```

到这里，normal 状态已经可以支持独立的读和写操作了。虽然设计的修改还没有结束，可能读者已经希望通过简单的仿真来检查错误了。现在，FIFO 地址等于 0x00 和 0x1f 的边界情况应该正常了，同样，地址等于 0x01 和 0x1e 的情况也应该是正常的。

第 6 步。新的读写时钟。在 FIFO Top.v 里，把输入 Clocker 改名为 ClkR，并增加输入 ClkW。对解串器里的 FIFO 来说，ClkR 用于接收时钟域，ClkW 用于 DesDecoder 发送并行数据。在 FIFO Top 里，把这两个时钟直接连到 DPMem1kx32 和 FIFOStateM 的对应端口上。

在 Deserializer.v 文件中，按照刚才的命名习惯，把 ParValidReg 改成 ParValidr。现在是把所有已有设计的命名由 *Reg 改为 *r 的好机会。

同样，把 FIFOReadCmd 输入端口改成 ReadReq；这样可以把 FIFO 外来的读请求和 FIFO 内读寄存器阵列的命令区别开来。请同时修改 DeserializerTst 对应的端口名。

在 Deserializer 里，修改 FIFO Top 的端口映射，把 Clocker 改成 ClkW，并新增一个和 ParOutClk 连在一起的 ClkR 端口。

简单地验证 Deserializer。比起刚才的仿真结果，仿真的错误应该更少一些了。虽然这个工程还没有最后完成，但 FIFO 的输入输出端口和图 19.1 所示的是一样的。

第 7 步。升级 FIFO 状态机 II。现在来修改 a_empty 和 a_full 状态，使得读写操作在这两个状态里不相关。目前的问题有：(a) 一旦发生了一种操作，另外一个操作不被响应；(b) 两种操作有优先级；(c) 状态机不检查计数器，不能区分读或写命令，更不能区分两种命令同时到来的情况。

编辑 FIFOStateM.v，在 a_empty 和 a_full 两个状态里，在写逻辑块中删除对 ReadCmdr 的赋值，在读逻辑块中删除对 WriteCmdr 的赋值。理由和修改 normal 状态一样，但是保留状态跃迁的那些代码。

在 a_empty 块里，从 memory 的命令逻辑里把那些负责状态跃迁的代码放到 a_empty 块的最下面，这样就可以用计数器而不是 memory 的命令来控制状态的跃迁。

用寄存器 tempCount 来控制计数器，tempCount 的位宽是 AWid。

```
// In this state, we know W == R+1; ...
// Memory command logic:
if (WriteReq==1'b1) ...
if (ReadReq==1'b1) ...
//
// Transition logic:
// Set default:
NextState = a.empty;
//
// Check for change:
tempCount = ReadCount+2; // Destination determines wrap-around.
if (WriteCount==tempCount)
    NextState = normal;
else if (WriteCount==ReadCount)
    NextState = empty;
```

对 a_full 也做同样的处理。

我们再回来回顾一下组合逻辑块里的命令逻辑。对 normal 状态来说，它的命令逻辑应该和下面一样：

```

// On a write:
if (WriteReq==1'b1)
begin
  WriteCmdr = 1'b1;
  incrWrite; // Call task, which blocks on posedge StateClock.
end
// On a read:
if (ReadReq==1'b1)
begin
  ReadCmdr = 1'b1;
  incrRead; // Call task, which blocks on posedge StateClock.
end
(transition logic...)

```

由上面的代码看到，写请求调用 incrWrite，使得下一个时钟上升沿的操作被屏蔽，从而看不到读请求。

如果是仿真，可以把读和写请求都放在 fork-join 块里，使得并发的读写请求可以被处理，接下来会同时处理并发的读和写，它们都会在上升沿阻塞。当它们阻塞后，将解除命令处理的阻塞，允许转化逻辑为下一个时钟上升沿更新 next_state 与地址计数器。代码如下所示：

```

fork // unsynthesizable! But do it for now.
// On a write:
if (WriteReq==1'b1)
begin
  WriteCmdr = 1'b1;
  incrWrite;
end
// On a read:
if (ReadReq==1'b1)
begin
  ReadCmdr = 1'b1;
  incrRead;
end
join
(transition logic...)

```

如上所述，升级 a_empty, normal 和 a_full 的代码。

在地址处理块中利用 StateClock 的上升沿处理计数，代码如下所示：

```

task incrRead; // Still incorrectly synthesized (but not done yet).
begin
  if (LatchR==1'b0)
  begin
    LatchR = 1'b1;
    @(posedge StateClock) // Read must not block longer than write.
    ReadCount = ReadCount + 1;
    LatchR = 1'b0;
  end
end
endtask

```

综合工具不支持 fork-join 块（网表对应的硬件电路总是并行运行的）。因此，用 DC 的宏使得 fork-join 对 DC 不可见。

最后，用 localparams 来声明状态，并把状态机的名字从 statename 改成 statenameS，最后的 S 意味着这是一个状态机的名字，而不是一个普通的信号。例如：“localparam[2:0] emptyS = 3'b000;”。可以用编辑工具的替换功能快速地完成状态机名的替换。

仿真并观察结果。在 testbench 里控制先不读，让 FIFO 满，然后再一直读，直到把 FIFO 读空。比起这个练习刚开始的时候，这个设计的功能完整的多了。

第 8 步。升级成可综合的 FIFO 状态机。用目前的设计综合出来的网表是不能用的。问题不在 PLL 上，而是在 FIFO 上：

- 产生 StateClock 的逻辑在一个变敏感变量的 always 块中，但一些变量并没有出现在这个块里，综合后有可能有的信号被悬空。
- 虽然更新地址的任务是时钟边沿有效的，但是并没有在时钟边沿去调用它们。这样的处理有可能在综合后形成锁存。

我们用以下方法来解决这些问题。

第 8 步 A。testbench。首先要确保我们的 testbench 对 RTL 的 FIFO 和 FIFO 的网表都是有效的。

保证读和写请求都是脉冲信号，先连着发写操作把 FIFO 写满，再连着发读操作把 FIFO 读空，然后还需要用 testbench 测试读写并发的情况。

按上面的要求升级 testbench。如果留给这个练习的时间不多了，可以直接把目录 Lab23_Ans/Lab23_AnsStep08A 中 FIFOTopTst.v 里的 testbench 直接复制过来用。暂时先不要关注功能的细节。

第 8 步 B。可综合的 StateClock 产生器。

在 FIFOStateM 里，把 always 块用敏感信号只有两个时钟的 always 块替换掉。如下所示，这样的写法是可以综合的。

```
reg StateClockRaw;
wire StateClock; // See code example below.
//
always@(ClkR, ClkW)
  StateClockRaw = !(ClkR && ClkW);
```

虽然用异或（`xor`，符号是 ^）要简单一些，但是它不能处理两个时钟 ClkR 和 ClkW 的相位非常接近的情况，所以我们会用到 `nand` 表达式。表达式 `and` 也是可以的，只是对于 CMOS 工艺来说，`nand` 比 `and` 要更简单，速度更快一些。

输出的时钟叫 `StateClockRaw`，而不是 `StateClock`，原因如下所述。

代码中不再人为加入延迟，因此，也不会再有惯性延迟过滤。这意味着当读和写的时钟相位发生偏移时，`StateClockRaw` 上会有毛刺。这个问题主要影响的是仿真，对于网表来说没有这个问题。因为我们之前从代码中删除了所有的延迟，以后会一直有这个问题。

为了解决这个毛刺问题，可以把 `StateClockRaw` 连在一个能过滤掉小毛刺的延迟库元件上。再把这个延迟元件的输出送给 `StateClock`。这样，对于毛刺的处理在仿真的时候和在网表里都是一样的了。我们可以用练习 22 里用到的那个元件来当做这里的这个延迟元件。

我们可以通过在脚本里增加 Tcl 命令来避免这个延迟元件在综合的时候被优化掉。这部分脚本如下所示：

```
// Glitch filter:  
DEL005 SM_DeGlitcher1 (.Z(StateClock), .I(StateClockRaw));  
  
//  
//synopsis dc.tcl.script.begin  
// set.dont.touch SM_DeGlitcher1  
// set.dont.touch StateClock  
//synopsis dc.tcl.script.end
```

注释掉原来的 StateClock 产生逻辑，用上面的办法来产生时钟。把练习 22 的 Library-Name_v2001.v 链接或复制过来，让这个延迟元件的 Verilog 仿真模型文件包含在.vcs 文件列表中。在引用.vcs 之前的地方加上 -v 参数，让仿真工具只编译工程用到的模型，而不是把整个库文件都编译一次。

不要忙着综合，先粗略地对 FIFO 做一下仿真。设计 testbench，使得在仿真结束之前，FIFO 至少发生一次满和空的情况。不用太在意仿真的结果，因为我们还要改 FIFO 的设计。

第 8 步 C。 可综合的产生 FIFO 地址的任务。为了达到这个目的，我们需要重新设计 FIFO 状态机组合逻辑块里的每一个状态。

任务里的事件控制逻辑是不能被正确的综合的。因此，把这些逻辑从任务里提出来，单独放到 always 块中，用 StateClock 的上升沿来驱动 always 块里的逻辑来更新地址计数器。这种边沿敏感的 always 块可以根据条件更新计数器，并且这种写法是可综合的。

当计数器被分离出来之后，把任务改写成对所有变化敏感，这样才能正确的综合。

这样改后的新任务会保存地址状态，并且给 FIFO 寄存器阵列发出读或写命令。always 块同步的更新寄存器阵列的地址。

新 always 块在复位时可以初始化 fullS 和 emptyS 状态的地址。这样，always 块包含了对所有地址的操作。

下面我们来实现一个当输入为 1'b1，地址就会增加的任务；当输入为 1'b0 时，任务将会中止相关的操作。这个 always 块和这个任务应该这么写。

首先，定义如下寄存器，在描述任务和状态逻辑时会用到：

```
reg[AWid-1:0] ReadAr, WriteAr      // Address counter regs.  
, OldReadAr, OldWriteAr // Saved posedge values.  
, tempAr;                // For address-wrap compares.
```

在对 FIFOStateM 的输出进行连续赋值时，ReadAr 和 WriteAr 应该在表达式的右边。以 Old 开头的变量用来保持上一次的值。

接下来，我们对各个新的 always 语句块及相应任务进行声明。通常这些任务只能特殊地处理 FIFO 的空和满。

对于读寄存器阵列来说，always 块应该这样写：

```
always@(posedge StateClock, posedge Reset)  
begin : IncrReadBlock  
if (Reset==1'b1)
```

```

    ReadAr <= 'b0;
else begin
    if (CurState==emptyS)
        ReadAr <= 'b0;
    else if (ReadCmdr==1'b1)
        ReadAr <= ReadAr + 1;
end
end

```

如果是任务，应该这样写：

```

task incrRead(input ActionR);
begin
if (ActionR==1'b1)
begin
if (CurState==emptyS)
begin
    ReadCmdr = 1'b0;
    OldReadAr = 'b0;
end
else begin
    if (OldReadAr==ReadAr) // Schedule an incr.
        ReadCmdr = 1'b1;
    else begin // No incr; already changed:
        ReadCmdr = 1'b0;
        OldReadAr = ReadAr;
    end
end
end
else begin // ActionR is a reset:
    ReadCmdr = 1'b0;
    OldReadAr = 'b0;
end
end
endtask

```

如果FIFO是空的，照理说，往哪里写数据都是可以的。按照这个思路，emptyS里的ReadAr的初始值我们可以不去关心。但是，我们希望计数器从一个确定值开始增加。也就是说，当状态跃迁到emptyS时，两个计数器必须从同一个值开始相加，比如说0。综合的时候，寄存器必须在一个always块中进行赋值，因此写指针（写计数器）必须在它自己的那个always块中被初始成0。

当FIFO满了之后，所有的存储空间都被存满了数据。被读取的第一个数据的地址是预先确定的，因此，写操作的always语句块，作为唯一一个被允许修改WriteAr的值的语句块，必须初始化WriteAr的值，将它从不确定状态转化为在fullS状态中读取的第一个正确值。这个值等于ReadAr的值，当然，该值会在StateClock的下一个上升沿到来时增加。

写操作的always写法如下所示：

```

always@(posedge StateClock, posedge Reset)
begin : IncrWriteBlock
if (Reset==1'b1)
    WriteAr <= 'b0;
else begin

```

```

    case (CurState)
    emptyS: WriteAr <= 'b0;      // Set equal to read addr.
    fullS: WriteAr <= ReadAr; // Set equal to first valid addr.
    endcase
    if (CurState!=fullS && WriteCmdr==1'b1)
        WriteAr <= WriteAr + 1;
    end
end

```

写操作的任务写法如下所示：

```

task incrWrite(input ActionW);
begin
if (ActionW==1'b1)
begin
if (CurState==fullS)
begin
    WriteCmdr = 1'b0;
    OldWriteAr = ReadAr;
end
else begin
    if (OldWriteAr==WriteAr) // Schedule an incr.
        WriteCmdr = 1'b1;
    else begin // No incr; already changed:
        WriteCmdr = 1'b0;
        OldWriteAr = WriteAr;
    end
end
end
else begin // ActionW is a reset.
    WriteCmdr = 1'b0;
    OldWriteAr = 'b0;
end
endtask

```

1'b1 还是 1'b0 应该被传递到 FIFOStateM 状态控制组合逻辑块里所有的任务中去。当 incrRead 和 incrWrite 在状态 emptyS 的复位逻辑中被调用时，它们的输入是 1'b0。其余的时候如果调用到它们，它们的输入是 1'b1。删掉设计中的 LatchR 和 LatchW。

用 ReadAr, WriteAr 和 tempAr 替换组合逻辑块里对 ReadCount, WriteCount 和 tmpCount 的引用，这些变量将通过连续赋值送到 FIFOStateM 的输出端口。除了状态跃迁的表达式中，这些变量不应该出现在其他的组合逻辑块中。

组合逻辑块还需要升级。首先去掉 fork-join 块，用 'DC 的编译命令使它们无效。

虽然去掉了 fork-join 块，但我们还得仿真并发的读写请求。只利用任务就能实现这个功能。可以用 case 表达式列举出读写请求的每一种情况。

我们可以直接调用升级后的任务，无论是单独的请求或并发的请求都没问题。下面是 a_emptyS, a_fullS 或 normalS 状态里实现这个功能的例子。

```

...
case ({ReadReq,WriteReq})
2'b01: begin
    incrWrite(1'b1); // On a write.

```

```

ReadCmdr = 1'b0;
end
2'b10: begin
incrRead(1'b1); // On a read.
WriteCmdr = 1'b0;
end
2'b11: // On a read and write:
begin
incrRead(1'b1);
incrWrite(1'b1);
end
default: begin // No request pending:
ReadCmdr = 1'b0;
WriteCmdr = 1'b0;
end
endcase
...

```

在 emptyS 状态或 fullS 状态中，只可能有一种操作，因此，对于它们来说，只会有一种任务。但是，仍然要保证没有请求时，任务的操作无效。

剩下来的工作是把 FIFO 改成可综合的。通常来说，一个状态机由两部分组成：一个由时钟驱动的逻辑块来保存状态信息；一个组合逻辑块来决定状态的跃迁。

地址计数器和状态寄存器是用时序逻辑进行处理，除此之外，其余的逻辑都是组合逻辑。如果综合这样的设计，还是有可能会形成锁存，并且可能导致功能不正常。下一步，我们会把组合逻辑改成时序逻辑来避免形成锁存。

为了避免竞争条件，让这个组合逻辑块对 StateClock 的下降沿敏感。同时，把异步复位也加入这个块中。这段新的代码应该是下面这个样子：

```

...
always@(negedge StateClock, posedge Reset)
begin
if (Reset==1'b1)
begin
// Reset conditions:
NextState = emptyS;
incrRead(1'b0); // 0 -> reset counter.
incrWrite(1'b0); // 0 -> reset counter.
end
else
case (CurState)
emptyS:// (The other previously combinational logic goes here)
...
endcase
...

```

用 testbench 仿真这个 FIFO，保证新设计的功能没有问题。

额外练习

如果时间充裕的话，综合生成一个网表，并用这个网表仿真。为了能够正确的综合出结果，需要解决建立和保持时间的一些违例，参考本书附带的光盘 Lab23/Lab23_Ans/Lab23_AnsStep08C/Deserializer/FIFOTop.sct 文件。综合大概要用半个小时。

为了缩短综合时间，在综合工具进入“Area Recovery Phase”后5分钟，按一次“control+E”，暂停综合进程。利用稍后提示的文件信息终止综合优化。综合可以正常的运行，只是优化做的不彻底而已。

FIFO网表的仿真结果应该和FIFO RTL代码的结果非常接近了。还有一些时序的问题，暂时不去管它。

第8步D(可选)。验证第8步C的综合结果是不正确的。如果时间允许的话，完成这一步，否则阅读这一步的内容即可。

进入目录Lab23/Deserializer，用Lab23中的脚本Deserializer.sct综合。用综合出的网表仿真。

仿真的结果是失败的。控制FIFO写的ParValid工作不正常。发送时钟域的并行数据时钟的脉宽大约是160 ps，大约每2 ns重复一次。问题出在产生并行数据时钟的DesDecoder上。虽然DesDecoder原有的边沿触发任务的问题被解决了，仿真也没问题，但网表的功能还是不对。

第9步。综合DesDecoder。仔细阅读DesDecoder目录下的DesDecoder.v文件。任务ClkGen是问题所在：它在部分变化敏感的always块中被调用，有可能会产生非标准的latch。综合工具要产生完全符合这种设计意图的latch会很困难。

并行时钟产生器需要被重写。首先把DesDecoder中的所有以Reg结尾的寄存器名都改成以r结尾。再把注释掉的任务都删掉。

把产生时钟的ClkGen任务改写成名为ClkGen的always块，敏感变量是时钟的上升沿，和其余模块的时钟边沿反相。如下例所示：

```
always@(posedge SerClock, posedge Reset)
begin : ClkGen
  if (Reset==1'b1) // Respond to external reset.
    begin
      ParClkr <= 1'b0;
      Count32 <= 'b0;
    end
  else begin // If not a reset:
    if (SerValid==YES)
      begin
        // Resynchronize this one:
        if (doParSync==YES)
          begin
            ParClkr <= 1'b0; // Put low immediately.
            Count32 <= 'b0;
          end
        else begin
          Count32 <= Count32 + 1;
          if (Count32==5'h0) ParClkr <= ~ParClkr;
        end
      end // if SerValid.
    end // not a reset.
  end // ClkGen.
```

边沿触发可以避免产生锁存器。但是还需要接着修改，删掉 ClkGen 块里的 SerClk 门，用条件选择来产生 SerClock。

```
assign SerClock = (SerValid==YES)? SerClk : 1'b0;
```

如果还在用 Shift1 临时寄存器，删掉这段代码，直接对 FrameSR 进行移位操作。代码如下所示：

```
always@{negedge SerClock, posedge Reset}
begin : Shift1
// Respond to external reset:
if (Reset==YES)
    FrameSR <= 'b0;
else begin
    FrameSR    <= FrameSR<<1;
    FrameSR[0] <= SerIn;
end
end
```

下面，我们让设计变得更容易配置。仿真的时候，我们可以看到 ParValid 输出的脉冲很窄，像毛刺一样的。我们来把这个脉冲的高电平改来可以维持数个 SerClock 时钟。

引入一个受快速时钟 SerClock 驱动的小计数据，用它来拉低 ParValid 的高电平。把下面的代码加到 DesDecoder 模块的最前面：

```
localparam ParValidMinCnt = 8; // Minimum number of SerClocks
                                // to hold ParValid asserted.
localparam ParValidTwid = // Width of ParValidTimer reg.
    ( 2 > ParValidMinCnt )? 1
    : ( 1<<2 ) > ParValidMinCnt )? 2
    : ( 1<<3 ) > ParValidMinCnt )? 3
    : ( 1<<4 ) > ParValidMinCnt )? 4
    : ( 1<<5 ) > ParValidMinCnt )? 5
    : ( 1<<6 ) > ParValidMinCnt )? 6
    : 7; // Thus, width is declared automatically.
//
reg[ParValidTwid-1:0] ParValidTimer;
// ...
```

上面的写法可以自动计算出 ParValid 高电平时间变量的位宽。

接着，Unload32 块应该被写成这样：

```
always@{negedge SerClock, posedge Reset}
begin : Unload32
if (Reset==YES)
begin
    ParValidr <= NO; // Lower the flag.
    ParOutr   <= 'b0; // Zero the output.
    ParValidTimer <= 'b0;
end
else begin
    if (UnLoad==YES)
begin
    ParOutr      <= Decoder; // Move the data.
    ParValidr    <= YES; // Set the flag.
end
end
end
```

```

    ParValidTimer <= 'b0;
end
else begin
    if (ParValidTimer<ParValidMinCnt)
        ParValidTimer <= ParValidTimer + 1;
    if (ParValidTimer==ParValidMinCnt && ParClk==1'b0)
        ParValidr <= NO; // Terminates assertion.
    end
end // UnloadParData.
end

```

在验证了新的DesDecoder之后，综合，并用网表仿真。先不要综合整个Deserializer设计。

第 10 步（可选）。综合Deserializer，并用网表仿真。在第9步的基础之上，通过完成对Deserializer的综合和仿真来结束本练习。我们会看到综合出来的网表只有部分功能可以正常工作。

当没有特殊的约束条件时，FIFO 为 32 字宽，Deserializer 的综合过程大约会持续 15 个小时，这个设计约是 75 000 个等效晶体管的规模。在答案目录下有一个.sct 的文件，如果你要综合Deserializer，请参考这个文件。在回家之前才开始运行综合会是个好主意。

之前已经完成了一些初步的综合，用大约 1 个小时的时间生成了一个网表文件（用第 8 步 C 的办法快速的综合出一个网表）。我们可以在第 10 步的答案目录 BadNetlist 中找到这个网表文件。

在Deserializer 仿真正确和进行综合之前，我们也许需要调试我们的设计。仿真会遇到的主要是串行时钟的同步问题，在到来的完整的 64 比特数据包被正确接收和译码之前，各时钟频率误差必须在 1/32（接近 3%）之内。因此，testbench 发送的串行时钟应该和Deserializer PLL 的时钟频率非常接近。作者使用的是半周期为 15.5 ns 的时钟。

需要思考如下问题：

- DesDecoder 可综合的并行时钟产生器被改成了对边沿敏感的设计，而不是对任何变化都敏感。因此，仿真的速度可能会减半。
- DesDecoder 频繁的调用 doParSync，如果输入一直都是零，则有可能会导致提取的并行时钟无效。
- 当进行 RTL 仿真和网表仿真时，PLL 里的 `VFO_MaxDelta 应该是不一样的。
- VFO.v 中的 VFO 的限制应该简单地定义：`DivideFactor ± `VFO_MaxDelta（若它本来还没有如此定义）。
- 在 VFO 中为源代码和网表仿真选择一个恰当的延迟等级（`NumElems），取值为 5 时会很合适。
- 在顶层的 testbench 里，仔细调整仿真时钟和串行时钟的频率。

本练习并不是要完成整个Deserializer 的设计，但是子设计里的每个子模块的源代码和网表仿真都要保证没有问题（PLL，DesDecoder 和 FIFO 三个子设计）。

Deserializer 已经能够成功将 FIFO 中的一些数据读出，并正确的送到输出数据总线上。图 19.5、图 19.6 和图 19.7 是相关的波形。

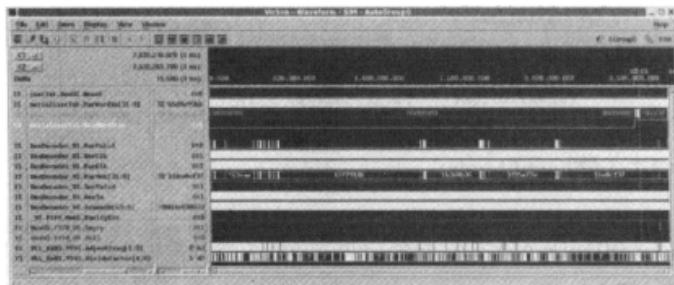


图 19.5 练习 23 解串器的网表的仿真结果(没有反标 SDF)。FIFO 的状态从空进入了非空。当接收到 ReadCmdStim 信号之后，数据被读出并送至并行数据总线

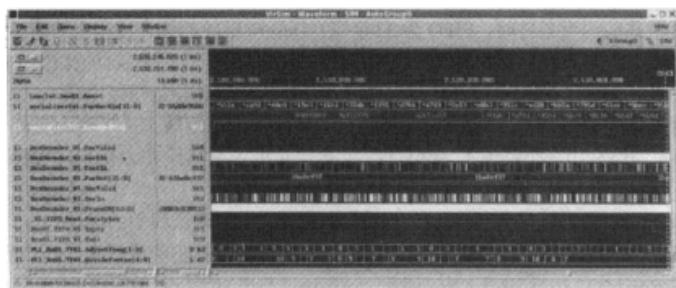


图 19.6 数据从 Deserializer FIFO 中读出的过程

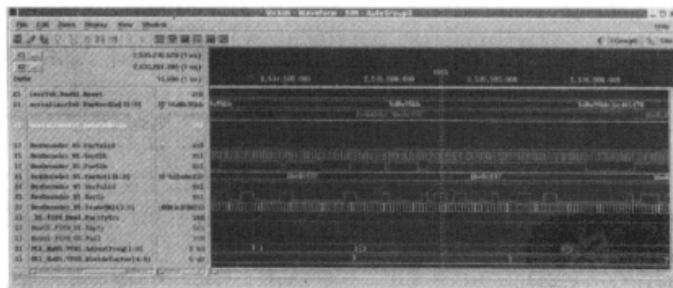


图 19.7 输入串行数据时的波形

19.2.1 练习后的思考

思考调试 Verilog 代码的方法。

19.2.2 补充学习

复习 Thomas and Moorby (2002) 的 4.9 节中关于 fork-join 的内容。

第20章 完成串行/解串器

20.1 串行器和串行/解串器

在第18章的一开始，我们就给出了SerDes的方块图。图20.1是SerDes升级后的方块图。其中，时钟2分频的电路被移到了Serialization Encoder里。

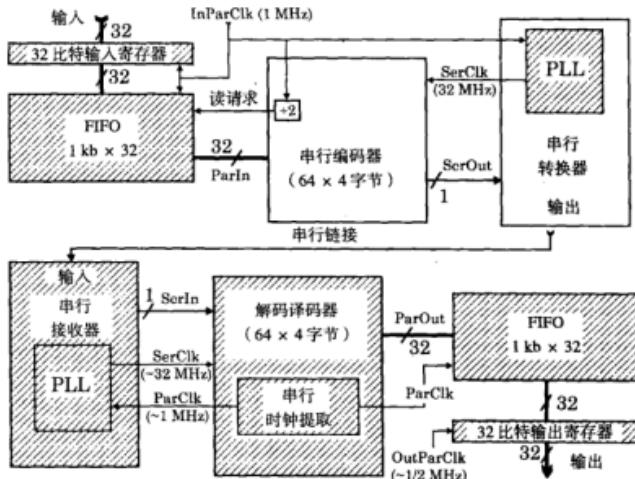


图20.1 SerDes的方块图。阴影部分的模块已经实现

虽然串行器的结构看起来和解串器较类似，但实际上实现起来要简单得多。首先，不需要从数据流中提取1 MHz的时钟，串行器的PLL只需把1 MHz的时钟32倍频作为串行数据的输出时钟即可。其次，串行器的FIFO只工作在一个时钟域。

Serializer用到的PLL与FIFO模块和Deserializer用到的是相同的。但是，为了实现数据的发送，还需要新建SerEncoder和SerialTx模块。

20.1.1 SerEncoder模块

SerEncoder模块的功能是串行编码（Serialization Encoder），从串行器的FIFO中读数据，并发送串行的数据包。

数据包被串行地发送给了SerialTx模块。

20.1.2 SerialTx模块

模块SerialTx最终把串行数据送到串行数据线上。

由于发送端的 PLL 包含在模块 SerialTx 的内部，因此模块 SerEncoder 的时钟也来自于模块 SerialTx。

20.1.3 串行 / 解串器

由于已经完成了 Serializer 的设计，完成整个串行 / 解串功能剩下的工作就比较简单了。只需在模块 SerDes 中例化 Serializer 和 Deserializer，把它们的对应端口用线连起来，再编写相应的 testbench 就完成了剩下的工作。这个贯穿本书的工程在这节里会告一段落。但是，在后面的章节里学习到 DFT 的时候，我们还会用到这个工程。

20.2 练习 24：串行 / 解串器

在目录 Lab24 下完成以下练习。

练习步骤

第 1 步。 使用之前的设计。在目录 Lab24 中新建目录子 SerDes，在 SerDes 下再建一个叫 Deserializer 的子目录。

把目录 Lab23/Lab23_Ans/Lab23_AnsStep10/Deserializer 里的所有内容都复制到目录 Lab24/SerDes/Deserializer 里。如果还想调试练习 23 里的设计，可以用自己的文件。但是下面的步骤都基于目录 Lab23 里提供的文件。

现在还没有必要把练习 23 里的 testbench 和综合环境等都复制过来。如果读者觉得有必要，可以晚些时候再复制。

第 2 步。 整理 SerDes 要用的文件。虽然整个 SerDes 用到的 PLL 和 FIFO 的设计文件都是一样的（例化的时候可能被传递不同的参数），但是整理一下要用到的文件还是有必要的。

目录结构图如图 20.2 所示。

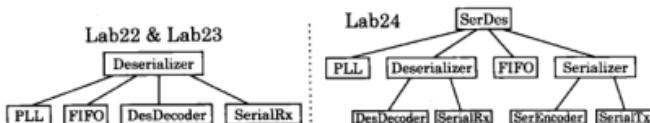


图 20.2 练习 24 的文件整理

在目录 SerDes 下，新建目录 Serializer，再把 PLL 和 FIFO 的子目录向上移一级。目录 Serializer，Deserializer，FIFO 和 PLL 都在目录 SerDes 下。

把文件 Deserial.inc 移至目录 SerDes 下，把它重命名为 SerDes.inc。

在目录 SerDes 中建一个名为 SerDes.vcs 的仿真文件列表，用仿真工具载入 SerDes.vcs，检查新的目录层次还有没有问题。

第 3 步。 建立 Serializer 模块。在目录 Serializer 下，新建 SerialTx 子目录并新建一个名为 SerialTx.v 的新文件。文件里除了例化了 PLL，没有别的逻辑。如果你觉得有必要，你也可以参考一下 SerialRx.v。

类似地，在目录 Serializer 下新建目录 SerEncoder，新建名为 SerEncoder.v 的文件，对应的模块是 SerEncoder。可以把 DesDecoder 的模块头复制到 SerEncoder.v 里，然后在它的基础上修改，图 20.3 是对应电路的整体结构示意图。

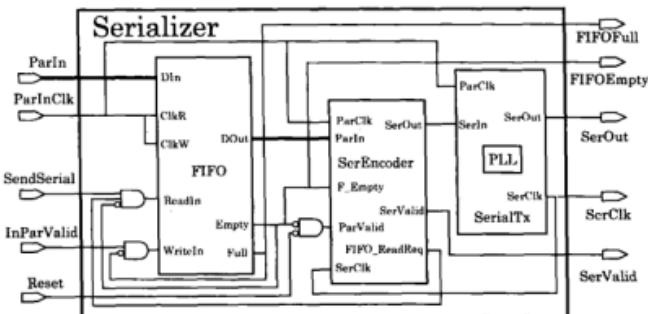


图 20.3 Serializer 的电路整体结构示意图，只画出了部分的 Reset 信号

SerDes 发送和接收端的串行数据的格式必然是一样的。因此，把和端口相关的 localparam 参数复制到一个位于目录 SerDes 下新的 include 文件里：SerDesFormats.inc。为了方便将来的调试，把解释性的注释也复制过来。由于数据包的组包和解析是由 DesDecoder 和 SerEncoder 里的具体逻辑完成的，所以不用考虑这个 include 文件移植。除了注释，这个文件里的内容应该如下所示：

```
// SerDesFormats.inc:
localparam[0:0] YES = 1'b1; // For general readability.
localparam[0:0] NO = 1'b0;
localparam[7:0] PAD3 = 8'b000_11_000;
localparam[7:0] PAD2 = 8'b000_10_000;
localparam[7:0] PAD1 = 8'b000_01_000;
localparam[7:0] PAD0 = 8'b000_00_000;
```

由于编码和解码的端口都使用了同样的数值，把它们参数化会给将来的维护带来很大的好处。当然，如果是具体的格式发生了变化，则需要编码和解码两边都升级设计。

在目录 Serializer 下，新建名为 Serializer.v 的文件，在文件里声明模块 Serializer，并在这个模块里例化 FIFO、SerEncoder 和 SerialTx。

可以把模块 Deserializer 里的声明复制过来，这样，不用增加新的端口，只需要修改对应端口的名字和数据方向即可。FIFO 大小、地址和数据位宽等参数可以保留下，名字和具体的数值都不用改。当然，如果有需要，可以在例化的时候传递新的参数。

Serializer 的端口应该如下：

输出：SerOut, SerValid, FIFOEmpty, FIFOFull, SerClk。

输入：ParIn (32 比特), InParValid, ParInClk, SendSerial, Reset。

在 Serializer 中，需要读控制信号将发送 FIFO 中的并行数据读出并送给串行编码器。在 Deserializer 中，需要读控制信号将接收 FIFO 中的数据读出并送给接收处理逻辑。通过这两个读控制信号，FIFO 完成了在这个连续通信系统里做缓冲的功能。当然在 FIFO 空和满的时候需要专门来处理，但是这需要通过协议来保证不发生类似的极端情况，不是我们需要考虑的问题。

由于 Serializer 需要写控制信号来把并行总线上的数据写入 FIFO，因此除了把 1 MHz 的并行数据输入时钟 (ParInClk) 送给 FIFO，Serializer 还负责产生 FIFO 的读写请求信号。

Serializer 还负责给 SerEncoder 的 ParValid 输入端提供 FIFO 有效的指示信号 (F_Valid)，指示由 FIFO 出来的并行数据什么时候有效。用一个连续赋值就可以实现这个功能：

```
assign F.Valid = !F.Empty && !Reset;
```

其余的控制信号这样来产生：

```
assign F.ReadReq = !F.Empty && SerEncReadReq && SendSerial;
assign F.WriteReq = !F.Full && InParValid;
```

这里以 F 开头的信号说明是 FIFO 的端口，SerEncReadReq 连在 SerEncoder 的 FIFO_ReadReq 输出端口上，而 InParValid, SendSerial 是 Serializer 的输入端口。

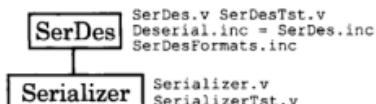


图 20.4 Serializer 的文件结构

在目录 Serializer 里新建一个什么也不做的名为 SerializerTst.v 的 testbench。仅仅用它仿真来检查一下连线和文件位置。

另外，在文件夹 SerDes 中生成空的占位文件，将其命名为 SerDes.v 和 SerDesTst.v。

Serializer 文件的结构图如图 20.4 所示。

第 4 步。完成串行数据发送模块。类似于 Deserializer 里的子模块 SerialRx，模块 SerialTx 很简单。它只需要：(a) 用简单的连续赋值语句把串行数据从输入送至输出；(b) 正确的例化 PLL。

第 5 步。完成串行数据编码模块。请参考上面的图 20.3 和图 20.5。

我们会把这个模块设计成可综合的设计。

模块 SerEncoder 需要使用串行和并行的时钟将数据组帧，并且串行化地把数据送给模块 SerialTx。这两个时钟都是由 SerialTx 的 PLL 子模块提供的锁相时钟，与模块 SerEncoder 本身的功能并没有关系，因此没有必要提取时钟的相位信息。而除了 FIFO 已经为空的情况，串行化的过程不能够被中止。

在练习 22 里，解串用到的 2 分频时钟是由 Deserializer 的 testbench 产生的，这是因为解串器的工作频率是 1 MHz，且给自己提供了正确的并行数据。这个时钟是和并行数据同步的，接收逻辑使用的是经时钟打过一拍的 ParOut 数据。但在串行端却不能这样，因为串行端不只使用了 1 MHz 的时钟，它还要监测并行数据的速率。这意味着 1/2 MHz 的时钟也会被用在 Serializer 的设计中。SerEncoder 负责输出这个时钟。

用外部 (testbench) 产生的 1 MHz (ParClk) 送给 Serializer。因此，在模块 SerEncoder 里必须要有将输入时钟 ParClk 2 分频的逻辑，并用这个分频时钟来产生 FIFO 的读请求。

除此之外，如图 20.5 所示，在模块 SerEncoder 里，还需要另外三个功能块。

- Loader 块。每个 2 分频时钟 (HalfParClk) 的上升沿都产生一个读请求。这个 always 块完成两个功能：

- (1) 从 FIFO 中读出 32 比特的数据。当 FIFO 为空时，数据为 0。
- (2) 通过判断 FIFO 是否为空来控制 SerValid 信号。

- Shifter 块。这个 always 块在每个 SerClk 的上升沿都产生读请求，并且产生了串行数据。Shifter 通过一个模 64 的计数器来判断 SerEncoder 输入缓存的数据是起始位还是数据位，并把数据送到串行线上输出。

- FIFO Reader 块。这个逻辑每两个 ParClk 时钟产生一个有效的 32 比特数据。它的时钟 HalfParClk 是 ParClk 2 分频后的结果。

这个逻辑很简单，如下所示：

```
assign FIFO_ReadReq
    = HalfParClk && !F.Empty && ParValid && !Reset;
```

完成的 SerEncoder 的示意图如图 20.5 所示。

第 6 步。用 SerializerTst 对 Serializer 仿真并调试。当 FIFO 已满时，FIFO 应该停止响应写请求。同样，当 FIFO 为空时，在最后一个有效数据发出去之后，串行数据无效。

Serializer 典型的仿真结果如图 20.6 至图 20.8 所示。

第 7 步。仿真 SerDes。在模块 SerDes 里例化 Serializer 和 Deserializer，并把对应的串行数据线相连，可能必须对一些连线进行重命名，例如把从这两个模块的并行输出端口的连线命名为不同名字。

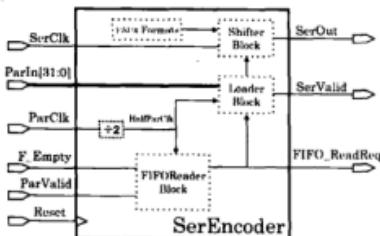


图 20.5 SerEncoder 示意图

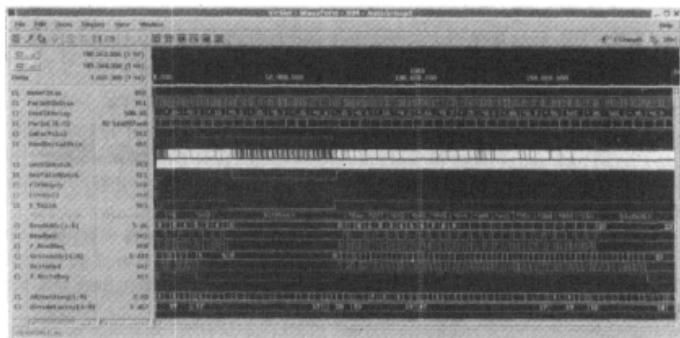


图 20.6 Serializer 的仿真波形。FIFO 分别进入了满和空的状态

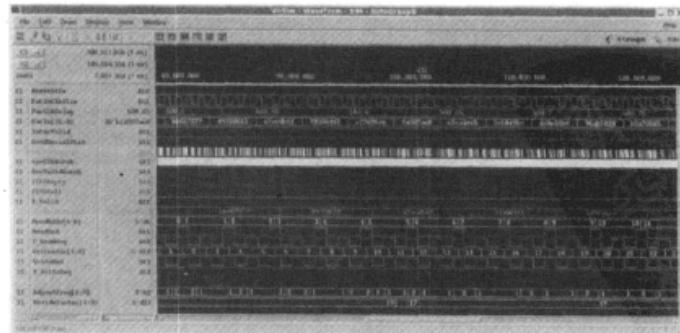


图 20.7 Serializer 的仿真波形的局部放大，显示了并行数据是怎么被处理的

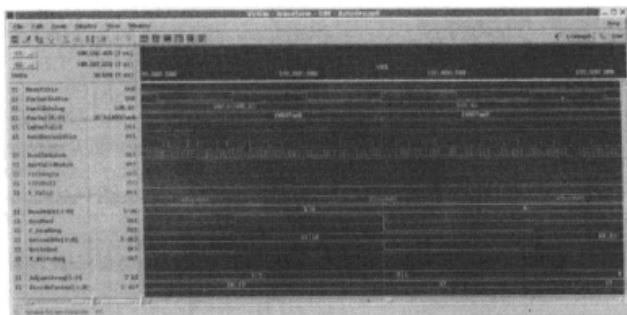


图 20.8 把 Serializer 的仿真波形放的更大，能清楚地看到串行时钟

把 SerializerTst 重命名为 SerDesTst，并用它来仿真。如果你设计的 FIFO 状态机有这个需求，你可以在仿真的初始阶段产生两个复位脉冲。

图 20.9 和图 20.10 是 SerDes 典型的仿真结果。

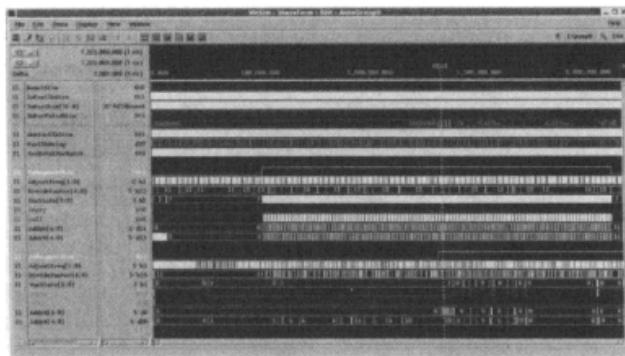


图 20.9 用完整的 SerDes 代码来进行仿真。从波形上看到，对 FIFO 的操作是正确的

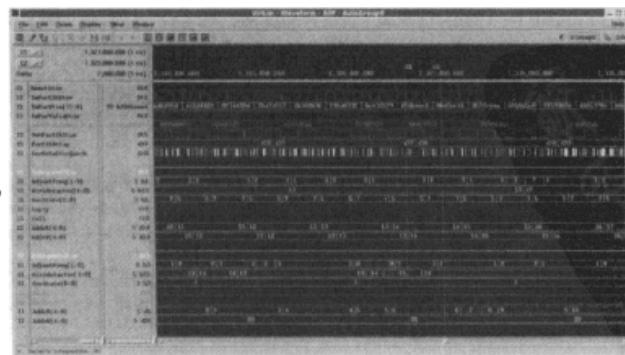


图 20.10 图 20.9 所示波形的局部放大，显示了处于接收时钟域的并行数据

可选练习：在 SerDes 已经可以工作了之后，修改 Deserializer 使得它有两组并行输出数据。一组用接收时钟来锁存；再用另一个时钟域里的，解串 PLL 内部的时钟（ParClk）来锁存总线数据。

仿真并观察结果。观察 32 比特输出数据的歪斜（skew）。这个结果说明了数字电路里时钟同步时可能会出现的问题。当然，数字的仿真工具是不可能把第 15 章中讨论的中间态问题模拟出来的。

第 8 步。SerDes 子模块的综合。要完整，优化地综合整个 SerDes 工程对这个课程来说需要的时间太长了。为了写出合理详尽的时序约束，需要反复综合 SerDes 好几次，逐渐的来优化综合脚本。

对于本课程来说，我们只需证明 SerDes 下的每个设计都是可综合的就可以了。在各自的目录下综合下面的设计：

- PLL
- FIFO
- Deserializer.DesDecoder
- Serializer.SerEncoder

Deserializer.SerialRx 和 Serializer.SerialTx 只是两个 wrapper，除了例化了 PLL 没有别的东西，因此没有必要现在综合它们，只综合 PLLTop 就可以了。

在答案子目录下有综合的脚本。虽然 DesDecoder 是一个相对较小的设计，但是由于有很多时序约束，综合的时间可能要数小时。仔细阅读综合脚本注释里提供的跳过某一步的方法。

同样地，对 FIFO 这个设计来说，它虽然只有一个模块，但是它的设计规模并不小，综合的时间可能也会很长。阅读 FIFO 的综合脚本里的注释，看看怎么尽快地得到可用的网表。

综合完上述设计之后，用仿真源代码的 testbench 来对网表进行仿真。这两者的仿真结果应该是几乎一样的。

可选练习：综合完每一个子设计之后，产生 SDF 文件。用 TSMC 的 Verilog 仿真核心库（tcbn90ghp.v，不是之前用的 *_v2001.v），并选择最优的负载模型延迟对网表进行仿真。这两个库的延迟并没有太大的区别，但是 TSMC 的时序检查会打印信息通知设计者有建立或保持时间需要调整或特别注意。

综合出来的网表等留在各自的文件中，更高一层的仿真或综合都不会用到它们了。

图 20.11 至图 20.19 是用 Verilog-2001 库来仿真的典型结果。反标了 SDF 之后，仿真的波形略有不同。

PLL 的典型结果如图 20.11 和图 20.12 所示。



图 20.11 PLL 网表仿真的结果

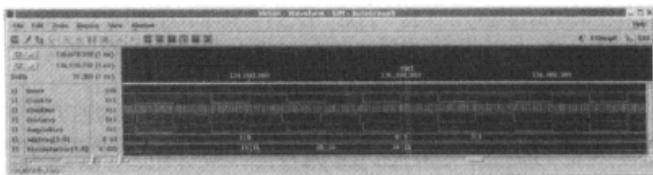


图 20.12 局部放大 PLL 网表仿真的结果，可以看到串行时钟

FIFO 的典型结果如图 20.13 和图 20.14 所示。

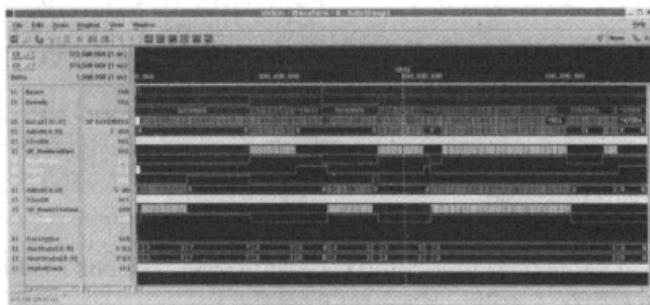


图 20.13 FIFO 网表仿真的结果

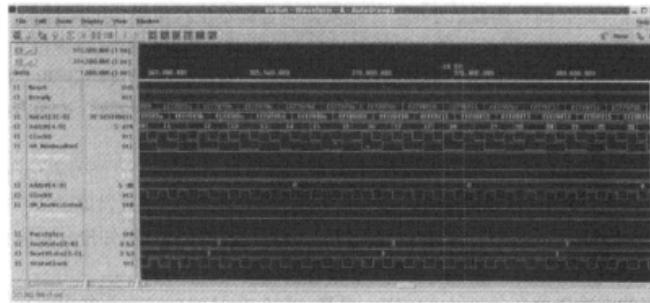


图 20.14 局部放大 PLL 网表仿真的结果，可以看到一个 32 比特并行数据的传输过程

DesDecoder 的典型结果如图 20.15、图 20.16 和图 20.17 所示。



图 20.15 DesDecoder 网表仿真的结果

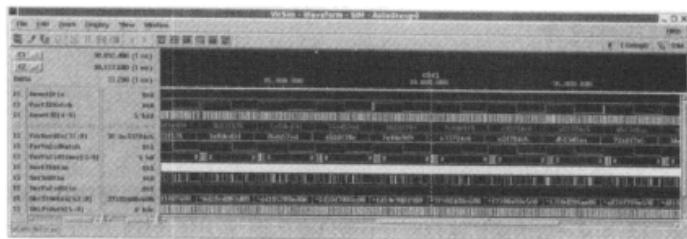


图 20.16 局部放大 DesDecoder 网表仿真的结果

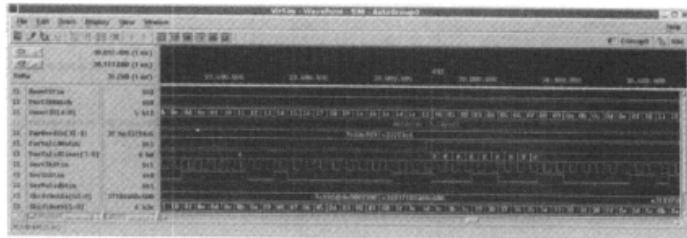


图 20.17 局部放大 DesDecoder 网表仿真的结果

SerEncoder 的典型结果如图 20.18 和图 20.19 所示。

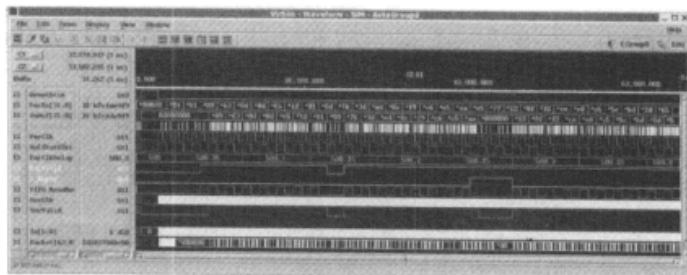


图 20.18 SerEncoder 网表仿真的结果

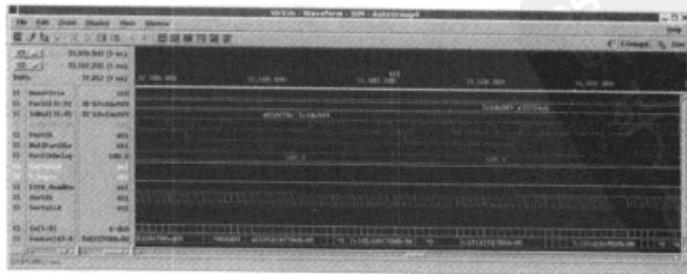


图 20.19 局部放大 SerEncoder 网表仿真的结果，可以看到串行时钟

20.2.1 练习后的思考

串行和解串的模块有哪些可能不兼容的地方吗？

串行 / 解串器的断言和时序检查是怎样的？

20.2.2 补充学习

把你的 shift 寄存器和 Thomas and Moorby (2002) 的 10.1 节中的开关级 MOS shift 寄存器相比。思考既然综合工具可以为我们综合出一个移位寄存器，那为什么我们还需要写开关级的模型呢？



第21章 可测性设计和全双工串行/解串器

21.1 可测性设计

21.1.1 可测性设计概述

可测性设计(Design For Test, DFT)不仅仅是一种技术,而且是一种方法学。实现了DFT则意味着硬件是可以测试的,换句话说,这意味着即使设计已经变成了芯片,它的某些功能仍然是可测的。可测性设计需要从两个方面来考虑,分别是设计错误(design errors)和硬件缺陷(hardware failures)。

设计错误只能通过错误的功能体现出来。大多数的设计错误会在仿真和综合的阶段被解决掉。这意味着在开发源代码阶段需要测试,在完成布局布线得到了准确的延时信息并将这个信息反标回网表后还需要测试。在这个阶段,逻辑和时序错误可以在为流片(tape out)制作掩模(mask)之前被修正。

硬件缺陷显然只能在设计真正转换成了物理硬件之后才能被检查出来。最常见的错误和芯片生产的质量有关,比如说一些门出现固定1或其他的错误。要发现类似的错误必须在设计时就提供了测试方法,这样才能在有缺陷的硬件上查到这个问题。在量产阶段,芯片的产生总会出现随机的制造错误。必须在提交给用户之前把有缺陷的芯片找出来。为了达到这个目的,也需要有检查硬件中错误的方法。

在绑定引脚和封装时,还可能会引入短路或开路的缺陷。这是很严重的问题,但是要发现这种问题却并不难,因为它们都位于芯片的边界上,可以较容易地观察它们的状态。

还有一类问题被称为软(soft)错误。它们主要是由温度,外力的挤压,未知的电磁干扰,离子辐射等原因造成的。把这类错误称为软错误是,因为出现这种问题总是暂时的,这类问题有可能会自行消失,有可能在复位后或其他的条件下可以继续正常工作。如果硬件的某个门器件间歇性发生问题,这不能算是软错误,而应该算做硬件缺陷。本章将不再讨论软错误的内容。

21.1.2 断言和约束

阻止产生硬件缺陷的第一道防线是优秀的软件。这意味着要完成一个优秀的设计,除了需要制定一份优秀的设计规范外,仿真工具、综合工具、静态时序分析工具等都要能够提供详尽的警告提示。如果合理的运用断言和仿真向量,优秀的软件可以帮助设计工程师发现潜在的问题。在展开硬件的设计工作之前,还需要确保硬件的功能规范是详细和正确的。此外,Verilog和综合库中的时序检查功能也非常重要,好的库可以提供完善的时序检查功能来检查设计是否满足时序的约束。

下面,让我们再从硬件的角度深入讨论这个问题。

21.1.3 可观测性

可观测性 (Observability) 是衡量检测硬件逻辑功能跳转和数据翻转能力的尺度。100% 的可观测性说明硬件内部的每一个状态都可以被观测到。先不考虑测试数据的组合顺序, 只考虑内部的每一个存储单元 Q (寄存器, 锁存器, memory 单元) 以及芯片的每一个输入引脚 I。把输入的数量写为 N_I , 把内部存储单元的数量写为 N_Q 。对数字逻辑来说, 每一个单元都是 1 比特, 因此, 状态总数为:

$$N = 2^{N_Q + N_I}$$

对于一个 8 kB 的 cache 来说, 它的总状态数为 $2^8 \times 8 \times 1024$, 约为 10^{20},000 。再把这个结果和系统中不含 cache 的那部分状态总数乘起来, 可以想象, 这是一个什么样的结果。这也是我们一般只用输入和存储器单元的数量 (以 2 为底的对数) 而不是实际的数值来表述系统的状态数量的原因。

如果是一个小规模的板级设计, 可以把希望检查的测试点都从电路板上引出。用自动测试机自动地和这些测试点接触, 输入测试向量, 并观察现象。如果发现了问题, 可以人工去修正或直接扔掉。

但是, 如果是大规模的集成电路, 这个办法是没有用的。首先, 除了 I/O 引脚, 其余的部分都被封装了起来。其次, 由于现在球栅 (ball grid) 封装的引脚多达上千个, 虽然用探针直接接触 I/O 引脚不会对芯片造成物理损害, 但是实际操作起来也是很困难的。

因此, 为了确保芯片的可观测性, 这种观测的逻辑必须在芯片内部实现。而不是等没有测试的芯片到手了之后, 再想办法怎么样去测试它。

21.1.4 覆盖率

覆盖率 (Coverage) 是用来衡量在进行仿真或实际硬件测试时测试向量质量的标准。覆盖率和可观测性有关。在给定可观测点的情况下, 覆盖率说明了测试向量覆盖到的可观测状态的数量。

用软件进行仿真时, 显然所有的点是可观测的。但考虑到覆盖率测试的需要, 在设计测试向量时应考虑实际芯片只能通过 I/O 来观测内部的状态。这样, 软件的仿真测试向量可以用在硬件的测试机上。从而可以通过对比硬件和软件仿真的结果来检查硬件的功能。

覆盖率工具 coverage metrics 的作用是检查有缺陷的 pin 脚, 端口, 门, 也就是我们常说的固定为 1 或 0 的缺陷。如果测试向量能够使得每个观测点都能翻转, 则这个测试向量的覆盖率就是 100%。缺陷仿真器是一种用来检查观测点翻转的专用仿真工具。

在测试硬件时, 一条测试向量可能超过或没有达到一个给定的覆盖率标准。为了减少花在测试硬件上的时间, 在达到覆盖率标准的同时, 我们希望测试向量能够尽量的短, 或测试向量的条数能尽量的少。在生产线上, 时间就是金钱。

覆盖率体现的是一条(组)测试向量和所有观测点之间的关系, 并不是指和观测到的观测点之间的关系。因此, 硬件的覆盖率通常都是小于 100% 的。

虽然硬件中的 ECC 功能 (如果有的话) 可以检查到测试没有找出来的问题。但是这仅仅对存储的数据有效, 对控制信号是没有作用的。而测试的目的是找到并去除硬件里所有的问题。

覆盖率小结：

覆盖率是用来衡量观测完整程度的标准。

- 覆盖率是利用一组测试向量来测试的一种方法。
- 100% 的覆盖率说明每个观测状态都被测试到。
- 如果覆盖率很低，则需要重新设计向量。
- 通常没有达到 95% 的覆盖率被认为是低覆盖率。

覆盖率可以认为是软件（仿真）过程。

- 关系到每一行被执行到的代码或表达式。
- 是个非马尔可夫过程。
- 与功能仿真无关。

覆盖率可以认为是硬件过程。

- 可以认为是随机错误检查。
- 通常检查硬件的固定为 1 或 0 的错误。
- 体现为硬件的功能检查。

21.1.5 边界情况与穷举测试

硬件设计的验证与测试是一个很大的话题，在这里，我们只接触一些基本原理。

想想前面所举的 8 kB cache 的例子我们可以知道，对于一个有实用价值芯片的覆盖率要达到 100% 是不现实的。假设有一条长度为 1024 比特的向量，测试向量的时钟是 1 GHz，测试这个 8 kB cache 所有状态的时间将会持续 $10^{19.980}$ 年。这是通过 $10^7 \text{ 分钟} / \text{年} = 10^{9+7} \text{ 向量} / \text{年} = 10^{9+7+3} = 10^{19}$ 比特 / 年计算出来的。而宇宙也大约只存在了 10^{10} 年。

幸好我们不用把所有的状态都覆盖到了之后才算是确保了硬件的功能没有问题。在相邻的存储单元处于某种状态时，就可能发现某种错误。比如，把相互交替的 'b010101… 和 'b101010… 向量灌入单元阵列，通常这样就能发现某一位或相邻两位上的错误。把这样的测试向量和 walking 1（除了一个在不停移位的 1，其余全是 0），walking 0（除了一个在不停移位的 0，其余全是 1）的方法结合起来，可以使我们对硬件寄存器缺陷情况的了解达到一定的程度。

接着再来看这个 8 kB cache 的例子，由于 memory 总是按地址排列的，因此只有对相邻的比特才能使用这样的测试方法。上面提到的相互交替的向量，假设每个向量输出两次，则每比特需要 4 个向量。对于 walking 1 和 walking 0 来说，如果用 20 条这样的向量来测试一个比特，这样的话，8 kB 共共需要 $8 \times 1024 \times 20$ 个 8 比特的向量，大约是 10^6 个状态。假设硬件测试向量的深度是 1024 比特，因此，总共只需要 1300 条向量。用 1 MHz 的时钟来测试这个 cache 的话也不要 1~2 ms。实际上，1 ms 的时间足够用来测试晶圆 (wafer) 上的 8 kB cache 了。因此，我们应该精心设计测试向量。

边界情况 (corner case) 通常指一部分特殊的取值。上面提到的测试方法从某种程度上来说也是一种边界情况。因为它只测试到了相邻位变化时的情况。而实际上，整个 memory 只有一部分存储单元和测试的情况类似。

更一般的情况，边界情况指的是用特殊的且独立的取值来进行测试。比如说，为了验证一个 Verilog 的从 $i = 0$ 到 $i = 127$ 的 for 迭代过程，可以选择 $i = -1, 0, 1, 126, 127$ 和 128 作为边界情况来专门验证。

对于实际的芯片，应该特别关注与电压岛的边界或时钟域相关的信号。在设计硬件的测试向量时，全 1 和全 0 的情况也应该加在边界情况里。

通常在测试芯片时，随着芯片外部的使能，关闭，读，写信号发生变化而变化的状态信号应该要求更高的覆盖率。记得在进行边界情况测试时检查每一个有可能出错的地方。

21.1.6 边界扫描

边界扫描（Boundary Scan）增强了对板上芯片输出引脚的可观测性。在电路板上专门把芯片的 I/O 引出来，而不用自动测试机上的针床（bed of nails），或者是人工使用探针；然后在这些引出的 I/O 上输入激励并观察结果。这使得我们可以通过 I/O 来观察芯片内部的结果。

每一个实现了边界扫描的芯片都内置了测试接口 TAP（Test Access Port）。TAP 控制器的实现和第 3 章中讲到的内部扫描练习一样，都很简单。它内部有一个状态机，能够自动地完成测试模式的切换，从而节省使用外部测试机的时间。

和第 3 章中讲到的 JTAG 内部扫描端口一样，TAP 有 5 个端口，分别是：TDI（Test Data In），TDO（Test Data Out），TCK（Test Clock），TMS（Test Mode Select）和 TRST（Test controller Reset）。除了端口 TMS，TAP 可以和芯片的其他功能共享剩下的 4 个端口。通过 TMS 来选择是否工作在测试模式下。

图 21.1 是边界扫描的电路示意图。在测试模式下，扫描锁存器（或寄存器）是连在一起的。输入被 TCK 打入，并从输出移出。每一个扫描锁存器都和输入、输出选通器连在一起。这样，在普通工作模式下的任意时刻，只要 TMS 和 TCK 有效了，被保存在输出寄存器中的逻辑状态就被逐一移出并供检查。

边界扫描可以与内部扫描、BIST 两者或者两者之一共存在一个芯片里。

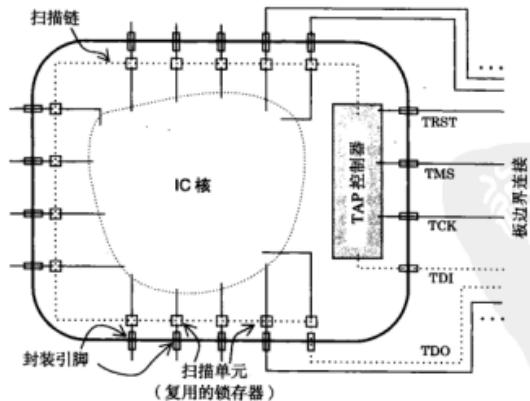


图 21.1 假设这颗内置边界扫描逻辑的芯片已经被焊在了电路板上。没有画出电路板上的其他 IC，也没有画出控制每一个扫描单元的 TCK 和 TMS 连线。这颗芯片工作在测试模式下，虚线是扫描链，把所有的扫描单元连在一起。边界扫描使得对于电路板来说，引脚的状态是可测的，而不是使得 IC 内部的状态对电路板来说是可测的。

21.1.7 内部扫描

在第 2 章中，我们已经介绍了通过选通器来实现的寄存器内部扫描（internal scan）。简单来说，内部扫描用扫描单元把芯片中的寄存器替换掉，使得所有的寄存器（甚至多个芯片）都被连在一起，在测试模式下可以组成一个寄存器链。本章最后推荐了一些文献，它们讲到了多种实现内部扫描的方法。

有时，全扫描（full scan）和内部扫描（internal scan）两个术语被混用。内部扫描组成的扫描链可能没有包含一些存储单元（例如 memory），并没有实现全扫描。因此，我们还是建议不要混用这两个术语。

全内部扫描（full internal scan）把芯片里的每一个寄存器都穿在一条链上。由于输入是已知的，因此，这意味着全内部扫描使得我们去观测全部 $2^{N \times M}$ 种内部状态成为可能。对于应用于救生设备、宇宙飞船、潜艇这类必须要正确工作的芯片来说，这是很重要的。而类似的系统往往会被设计得足够简单，以保证达到 100% 的覆盖率是可以实现的。

内部扫描不包含引脚（pad）。图 21.2 和图 21.3 是实现内部扫描的具体例子。

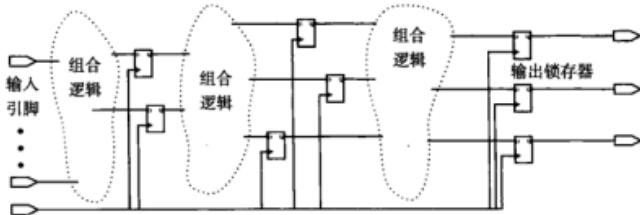


图 21.2 一个不包含内部扫描的设计。输出被锁存，组合逻辑用虚线表示，每一个时序单元都受时钟驱动

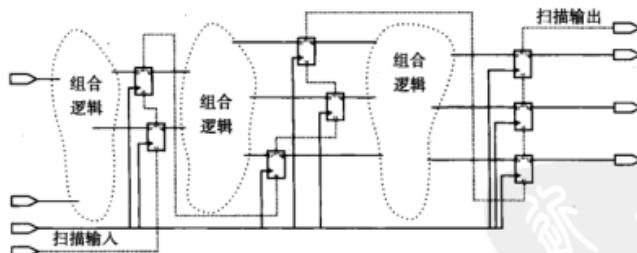


图 21.3 给上面的设计插入扫描链。用扫描单元把寄存器都替换掉。扫描链的路径从扫描输入开始，到扫描输出结束。没有画出扫描时钟和扫描模式控制

出于同步的考虑，几乎所有的芯片输出都是寄存器输出。因此，输出应该都是在扫描链上的。有些逻辑可能不能放到扫描链中。对于全扫描来说，只有不影响最终功能的逻辑才不把它放进扫描链里。进行这样的处理往往是由于设计上的失误或生产线出现了问题。一个典型的例子是 386-SX 微处理器：每当检测到 386 核（die）的片内 cache 出现了问题，就禁止这颗芯片的 cache 功能，然后正常封装这颗芯片，最后按一个较低的价格以 386-SX 的型号卖这种没有 cache 功能的芯片。当然，这种 386 处理器会有数千个无用的门器件。

这些不可观测的逻辑仍然会影响芯片的参数，因为它们仍旧会分走时钟的电流，和正常工作的逻辑门一样会存在漏电流。只要它们在这颗芯片上，就会增加芯片的功耗。

需要再次提醒的是，内部扫描和边界扫描不是互斥的关系，它们可以共存于一颗芯片里。

21.1.8 BIST

内建自测试 (Built In Self Test, BIST) 并不是仅在芯片设计里才用到的概念。PC 在开机启动时都会运行存在 ROM 里的自检程序。当低级格式化硬盘时，磁盘控制器 ROM 里的程序，会自动的去读写硬盘每一个比特进行检查。

BIST 的一个优点是它不需要外部的测试仪器，而且只有一个控制输入和一个结果输出。对于大多数的情况，BIST 的端口可以和其他的功能共享芯片的引脚。而且，有 BIST 功能的芯片可以在测试机上同时进行测试。因此，虽然 BIST 功能一定程度上增大了芯片的面积和复杂度，但是却节省了量产芯片的测试时间。

由上面的分析，我们看到 BIST 对芯片 I/O 的影响可以忽略不计，但是为实现 BIST 而增加的芯片面积却是不可忽略的。首先，BIST 的控制器和控制算法必须在芯片的内部实现；其次，为了实现 BIST，芯片还需要额外的布线。此外，如果芯片的功能很复杂，例如，芯片内部有很多的寄存器或支持很多种工作模式，对于这样的情况，必须精心设计 BIST 电路，这增大了电路设计的工作量。

插入 BIST 的过程如图 21.4 和图 21.5 所示。

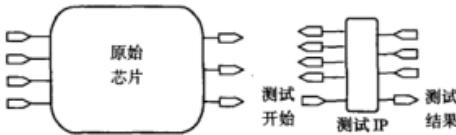


图 21.4 芯片和 BIST 控制器 IP 的示意图

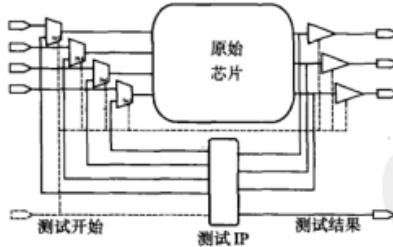


图 21.5 插入 BIST 的典型做法

BIST 通常对面积较大和具有重复性结构的 IC 来说（例如 memory 芯片），测试效率最高。如果使用的 memory 还有空闲空间，可以利用 BIST 找出有缺陷的 memory 单元并用没有问题的单元替换掉有问题的单元，从而提高良品率 (yield)。

BIST 可以和扫描逻辑共存。可以为 BIST 功能增加一种测试模式。这个测试模式的流程如下：灌入预置的数据，结果被保存下来，比对结果，复位或继续执行。为了提高测试效率，BIST 中往往内置了用来产生伪随机测试向量的状态机。

BIST 在不能直接测试的系统里很常见，比如说航天设备或潜艇等。在火星计划里的设备就实现了非常完备的 BIST 和避免灾难的多余度系统。由于宇宙射线的原因，整个设备暴露在高辐射的环境里，因此，对每一种可能出现的软错误都要做到专门处理，保证系统能继续正常工作。

总之，DFT 是一种设计方法学。这种方法学包含了解决边界情况和覆盖率问题的方法，而且这种方法学无论是对软件还是硬件，无论是设计阶段，仿真阶段还是最后的硬件实现都有用。我们平时做的仿真，错误仿真，为内部扫描或边界扫描插入专用硬件电路，内建自测电路和利用引脚来观察芯片内部状态都可以算到 DFT 的方法中。

21.2 练习 25：扫描和 BIST

在目录 Lab25 下完成以下练习。

练习步骤

第 1 步。 内部扫描练习。在目录 Lab25 下新建名为 IntScan 的子目录。如果你还没有完成练习 5 里的第 9 步（参见第 3 章），只把纯组合逻辑那部分复制过来，然后再做第 9 步。把设计更名为 Intro_TopFF，用综合工具插入扫描单元，检查结果，接着做本次实验的下一步。

如果你已经完成了这一步，就直接做本次实验的下一步。

第 2 步。 综合工具插入边界扫描。边界扫描的用途是测试电路板上的芯片。和 BIST 一样，对于小设计它的效率并不高。在本练习里，我们将用综合工具在练习 1 里原始设计的基础上插入边界扫描。选择练习 1 的目的是，因为我们的 SerDes 工程比较复杂，要给 SerDes 插入边界扫描会比较麻烦。

A. 在目录 Lab25 下新建名为 BScan 的子目录。把练习 1 里的原始 Intro_Top 设计复制过来，用 TestBench.v 做一下简单仿真检查线的连接。然后，把目录 Lab25 下 Verilog 的 pad 模型封装文件和综合脚本都复制过来。如果还没有建立链接，可以直接从本书附带光盘的 misc 目录下复制过来。

B. 增加测试端口。自动边界扫描需要 TAP。因此，在 Intro_Top 里增加一个新的输出端口： ScanOut，并增加 4 个新的输入端口： ScanIn, ScanMode, ScanClk 和 ScanRst。这是 JTAG 内部扫描 TAP 的常用名字。

C. 例化 pad。为了能让综合工具插入边界扫描，我们的设计中必须有 pad。需要三种类型的 pad：输入，输出和三态输出。三态输出是 TAP 的 TDO (ScanOut) 要用到的。

在 Verilog 的设计里，pad 被加在设计的顶层模块。虽然也可以在顶层模块之上再加一层把顶层模块包起来，然后再把 pad 加到这一层。但是如果直接把 pad 加到顶层模块里而不改变端口的名字，这样原来的 testbench 就可以继续用了。

符合要求的 TSMC pad 库不能用来被综合。因为它库里的 pad 是多用途的，可以通过控制从而实现输入、输出或双向的功能。把这样的 pad 加在顶层里会使得仿真和综合都变得非常麻烦。

为了完成这个练习，作者从库中选出了三个 pad，并提供了对应的封装文件为仿真来使用。封装文件是链接到目录 Lab25 的 tcpadlibename_3PAD.v 文件。

封装文件的名字，只能在本练习中用来例化 pad。它们的名字分别如下：输出 pad：PDC0204CDG_18_Out；输入 pad：PDC0204CDG_18_In；三态输出 pad：PDC0204CDG_18_Tri。数字“0204”的含义是驱动强度（02 = 2 mA），“18”说明 I/O pad 的电压是 1.8 V（对于 1.0 V 的核心电压来说）。端口的声明方式就是典型的 Verilog 标准用法。

```
module PDC0204CDG.18.Out (output PAD, input I);
module PDC0204CDG.18.Tri (output PAD, input I, OEN);
module PDC0204CDG.18.In (output C, input PAD);
```

库中的 OEN 低电平有效。对于 wrapper 里的三态门，这个使能被改成了高电平有效。

在 Intro_Top 里，X、Y 和 Z 是输出端口名，A、B、C 和 D 是输入的端口名。在 Verilog 里，每一个端口都隐含与一根与端口名同名的线相连。在我们声明了新的以 to 和 from 开头的连线后，Intro_Top 看起来应该是这样的：

```
PDC0204CDG.18.Out Xpad1( .PAD(X), .I(toX) ); // X is port; toX is wire.
PDC0204CDG.18.Out Ypad1( .PAD(Y), .I(toY) );
PDC0204CDG.18.Out Zpad1( .PAD(Z), .I(toZ) );
PDC0204CDG.18.In padA1( .C(fromA), .PAD(A) );
PDC0204CDG.18.In padB1( .C(fromB), .PAD(B) );
PDC0204CDG.18.In padC1( .C(fromC), .PAD(C) );
PDC0204CDG.18.In padD1( .C(fromD), .PAD(D) );
```

TAP 的端口要和对应的 pad 相连。先把 pad 里连 core I/O 的那部分端口悬空，让综合工具来完成这些连线的工作。

```
PDC0204CDG.18.Tri TDOpad1( .PAD(ScanOut)/*, .I(), .OEN() */ );
PDC0204CDG.18.In padTMS1( /*.C(), */ .PAD(ScanMode) );
PDC0204CDG.18.In padTDI1( /*.C(), */ .PAD(ScanIn) );
PDC0204CDG.18.In padTCK1( /*.C(), */ .PAD(ScanClk) );
PDC0204CDG.18.In padTRST1( /*.C(), */ .PAD(ScanRst) );
```

在进行综合和网表优化时，把 pad 实例当做普通实例处理即可。为了防止在综合的时候 pad 的连线被修改，在 Intro_Top.v 里加上编译指令，把所有的 pad 实例都没成 dont_touch。可以用通配符进行设置，例如：“*pad*”。

综合脚本.sct 会告诉综合工具，哪些端口（参见第 2 步 B）是我们想用来作为 TAP 的端口的。

D. 综合边界扫描逻辑。在例化了 pad 并完成了连线之后，用目录 Lab25 里的综合 BScan.sct 脚本，给设计插入边界扫描单元和 TAP 控制器。编译后输出综合结果，用 design_vision 查看综合结果，并阅读网表 Intro_TopNetlistBSD.v。可以试着从 pad tdo 沿着路径，反向地在电路中找到对应的 TAP 控制器。

这个练习仅仅是给大家展示怎样处理 TAP 端口，并没有完成控制器的设计。因此，这个网表是不能用来仿真的。

剩下的练习是完成 memory BIST 的设计。

第 3 步。为 DPMem1kx32 RAM 设计 BIST。

A. 建立工作目录。新建名为 BIST 的子目录，把练习 24 目录 FIFO 下的 DPMem1kx32.v 复制到这个目录里来。把这个 memory 文件改名为 Mem.v，模块名也做相应的修改。

B. 产生一个基准的综合结果给以后使用做准备。然后设置仿真的 testbench。这个准备工作让你再次熟悉（已仿真的）存储器功能，开发进度并降低设计错误的可能性。

首先，修改参数，把 A 里重命名的那个 RAM 改为 32×32 bit 的存储器，设置成按面积优化、综合它。和通常一样，只在脚本里添加简单的设计规则，例如最大扇出等。把综合的面积报告先复制到另外一个文本文件里，将来可以用来做对比。把这个综合约束也备份一份。不要用这个网表来仿真，我们的目的只是比较加入了 BIST 后面积的变化。

为了仿真原始的设计文件，在新文件 MemBIST_Tst.v 中例化 A 里重命名的 RAM，并新建一个简单的 testbench，能够说明这个 memory 的读写功能正常就可以了。读和写共用一个时钟，memory 的大小为 32×32 。层次结构应该如图 21.6 所示。

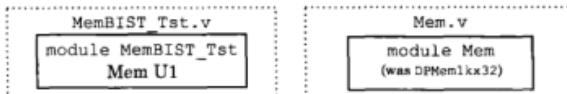


图 21.6 练习 25, 第 3 步 B。这两个 Verilog 的文件，一个是 testbench，一个是 memory 模型。Mem 在 MemBIST_Tst 中被例化

C. 准备实现 BIST。BIST 的作用是用来检测随机的硬件缺陷。在本练习里，为了方便，假设缺陷的出现是相互独立的。对于真正量产的芯片来说，我们也是基于这个假设找到了大多数芯片出现的缺陷的。

必须确认 BIST 的参数和 memory 的深度、宽度参数一致。这样，就可以在 BIST 模块里使用同样的参数定义。因为这个 memory 是支持奇偶校验的，为了检查是否有任何一个比特出现错误，甚至是软错误，我们应该一直检查奇偶校验。

我们将会这么来实现 BIST：

第一组测试序列（test pattern）：验证地址的有效性。给每一个地址写不同的值，再把这些值读出来和写入的值进行比较。这能检查出永久性短路和开路的地址线，也能检查到地址译码逻辑的错误。

我们的写入值应该从 0 开始计数，直到地址最大值。在每个字中的多个偏移位置复制这个计数值，它也很方便查看。这种类型序列从地址 0 到地址 31（ $5'h0$ 到 $5'h1f$ ）的计数值可以为：

`32'h0e0.e0e0, 32'h1e1.e1e1, ..., 32'hffff.ffff`

第二组测试序列：给 memory 的每个比特都写“1”，读出并检查结果。再写“0”检查一遍。这可以检查出上一步没有检查出来的某个比特固定为“1”或“0”的问题。

虽然并没有彻底充分验证这个 memory，但是对于本练习来说，就只做到这里了。如果希望进一步检查，还可以用 walking 1 等测试方法。测试大容量 RAM 的测试向量需要精心设计，从而使得芯片可以自己将预置的测试向量灌入 memory。而且所有的测试可以很快做完，对外部检测设备的需求最小。

第 4 步。设计 BIST 的接口。BIST 的逻辑应该单独放在一个模块里。在后端布局布线的时候，memory 总是以一个独立的块来摆放的。因此，BIST 只能布局在 memory 之外。

BIST 是通过地址总线和数据总线来读写 memory 的。读写可以共享时钟。为了实现 BIST，需要一个输入送到被测 memory 通知什么时候启动测试，还至少需要一个输出信号输出测试结果。在本练习里，这些端口将被独立使用，而不和其他的功能共享端口。在自检测试时，模块 Mem 不响应外部的读写请求，也不输出内容。

实现以上目标最好的办法是把Mem和BIST都包含在同一个顶层中。这样，在自测时，BIST和对应的 Mem 就不会受外部信号的影响。

第 5 步。 创建 BIST 的 wrapper。在文件 MemBIST_Top.v 中新建名为 MemBIST_Top 的模块，并复制文件 Mem。把 Mem 的端口复制到这个文件里来，再加上一个新的输入端口：DoSelfTest 和两个新的输出端口：Testing 和 TestOK。

按照上面的叙述修改了 MemBIST_Top 的文件头之后，为每个 I/O 明确定义连线，用连续赋值语句将 Mem 的所有 I/O 直接连在 MemBIST_Top 中对应的 I/O 上。明确的连线在本练习中是非常重要的，它能在练习后面部分简化 BIST 和 Mem 之间的互连。这些连线不占用面积或时延开销，因为不论如何，综合器会将含蓄的连线转化为明确的连线。

层次结构看起来应该如图 21.7 所示。

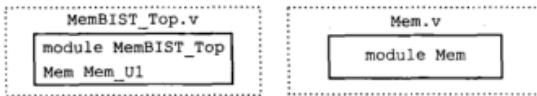


图 21.7 Lab25 第 5 步 MemBIST_Top wrapper 的端口和 Mem 的端口几乎一样。Mem 在 MemBIST_Top 被例化

我们在第 3 步里完成了 MemBIST_Tst.v。在 MemBIST_Top 里例化了 Mem 之后并连接了相应的端口之后，把 MemBIST_Tst.v 更名为 MemBIST_TopTst.v，同时修改 testbench 的模块名。把 MemBIST_TopTst.v 里例化的 Mem 改成 MemBIST_Top。简单做一下仿真，检查连线有没有问题。

第 6 步。 定义 BIST 的接口。把文件 MemBIST_Top.v 另存为 BIST.v，生成新的内建自测试模块 BIST。

但在新文件里进行改动之前，在这个文件里把例化 Mem 的代码再粘贴复制一份，并把实例名从 Mem 改成 BIST。在这个设计里，因为 Mem 已经有了完整的接口，所以先将 BIST 实例连上去最容易，接着，在复制的 BIST.v 文件里完成对要求的 BIST 的 I/O 端口的声明。

完成的层次结构如图 21.8 所示，接下来将修改 BIST 实例，这样就知道如何修改 BIST.v 了。

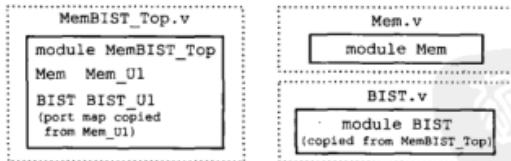


图 21.8 练习 25，第 6 步。利用 MemBIST_Top 来产生 BIST 的端口

接下来，让我们来看应该如何修改 MemBIST_Top.v 中的 BIST 实例。

A. 将参数 AdrHi 和 DWid 传递给 BIST。因此，可以保留从 Mem 复制过来的那部分代码。删掉端口 Dready 和 ChipEna。

B. Mem 的数据输入输出端口显然仍然是需要的，因此在 BIST 中保留这些端口。接着，我们会复用 Mem 的这些总线，让它们直接连在 MemBIST_Top 的端口或 BIST 的端口上。我们将

Mem实例的输出端口 DataO 和 BIST 实例的输入端口 DataI 相连，反之亦然。这样，就在 BIST 实例中保留了 DataI 和 ataO 端口。

C. 给 BIST 增加地址输出端口，写请求（ReadCmd）和读请求（WriteCmd）端口。因为在进行 BIST 测试时，不会有同时有效的读写请求。因此，BIST 的地址线同时接在 RAM 的读写地址输入端口上。

D. 给 BIST 增加时钟和硬件复位输入。把 BIST 的时钟和 RAM 的读时钟连在一起，用这个时钟来驱动 BIST 电路。

E. 增加 BIST 专用的 I/O，输入：DoSelfTest，输出：Testing 和 TestOK。这个端口直接连在 MemBIST_Top 的顶层端口上。把 RAM 的 ParityErr 输出送给 BIST，用来监测奇偶校验的结果。

修改到这里，MemBIST_Top 里的 BIST 实例应该看起来是这个样子的：

```
wire ClkRw, Resetw, ...; // Assigned from MemBIST.Top module inputs.
...
BIST #( .AdrHi(AdrHi), .DWid(DWid) )
  BIST.U1
    (.DataO(), .Addr(), .ReadCmd(), .WriteCmd() // outputs.
    , .Testing(), .TestOK() // outputs.
    , .DoSelfTest(), .ParityErr(), .DataI() // inputs.
    , .Clk(ClkRw), .Reset(Resetw) // inputs.
  );
```

第7步。在 MemBIST_Top 里实现 BIST 控制。在更新了如上的端口名后，完成相应的连线。对于选通逻辑，只需要简单的用连续赋值表达式就可以了。如果在第5步里，你就是用连续赋值来实现实例 Mem 和 MemBIST_Top 文件头之间的连线的。那么，只需多声明一些线型并增加一些条件表达式就可以了。

当输入DoSelfTest拉高时，这个边沿将会启动测试过程。MemBIST_Top 将拉高信号 Testing 并进入测试模式。在整个测试过程中，Testing 将一直为 1。当自测完成且没有发现问题时，TestOK 将拉高，否则，它将一直为 0。

在完成了以上步骤之后，回到 BIST.v。在模块 BIST 里删掉除了文件头和参数的所有内容，同时更新模块的端口名。用 MemBIST_TopTst 简单地对 MemBIST_Top 进行仿真，检查一下连线。

第8步（可选）。实现 BIST 的功能。在实现其他的功能之前，应该首先定义测试模式。下面是一种办法：

```
...
reg AllDone // Flag completion of testing for internal BIST use.
  , Testngr; // Sets test mode for the BIST.
//
assign Testing = Testngr; // Testing is a BIST output port.
//
always@(posedge Clk, posedge Reset)
begin : TestSequencer
  if (Reset==1'b1)
    begin
      Testngr <= 1'b0;
      AllDone <= 1'b0; // Normal level (noncommittal).
      ...
    end
  end
```

```

else // Must be a clock:
begin : TestClocked
if (DoSelfTest==1'b0)
    begin // Init, but leave TestOK alone:
        Testngr <= 1'b0;
        AllDone <= 1'b0;
        ...
    end
else Testngr <= 1'b1; // Entering test mode for this clock.
...

```

所有的寄存器都以“r”结尾。Testngr 的上升沿启动自测试过程。

受时钟驱动的 always 块用来产生 BIST 测试时的状态机。在第 3 步里，我们测试了 memory 所有地址的读写。把每一种测试都包含在各自的 always 块里产生比较合理。这些 always 块会被测试序列控制器分别按次序调用。测试序列控制器每一个时钟都去检查当前的测试状态；每当一种测试运行完毕之后，当前的这个 always 块会发出一个标志信号，通知测试序列控制器当前测试已经完成。

完成 BIST 设计是个很好的可选练习，但是完成这个设计至少需要用一天的时间。因此，作者已经在目录 Lab25/Lab25_Ans 下为读者提供了完整的 BIST 代码，它在文件 BIST_Done.v 中。

把 BIST_Done.v 复制到你的 BIST 目录里。复制 BIST.v 并以另外一个名字保存起来，看懂 BIST_Done.v 之后，将它复制或链接到 BIST.v 中取代原先空的接口模型。简单仿真检查连接是否正确。

如果希望在 BIST 上做更多的练习，可以把答案中的部分逻辑用 Verilog 的任务（task）来替换。测试序列控制器有 6 个比较重复的阶段。可以用把一个任务调用 6 次来完成。

仿真，检查你的改动是否正常。结果如图 21.9 所示。



图 21.9 MemBIST 的仿真结果

第 9 步。综合完整的 MemBIST_Top，按第 3 步 B 用到的同样的综合约束来优化面积。在约束条件不是很苛刻的条件下，综合的过程会持续 10 分钟左右。不要用这个快速产生的网表来仿真。因为还没有做时钟约束，因此仿真的结果可能是不正确的。但综合出的面积是基本准确的。把这个面积和没有 BIST 的面积做比较。

可选练习。在进行了面积比较之后，在综合脚本里加上时钟和最大输出延迟的约束，并修正保持时间的违例，重新综合（答案目录下有具体的数值）。这次综合大约耗时半个小时或更长，但是仿真的结果是正确的，如图 21.10 所示。

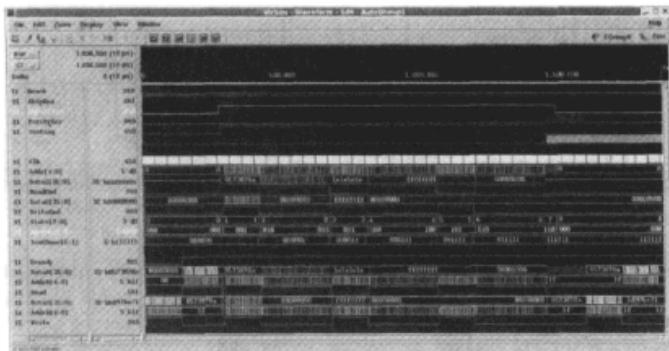


图 21.10 MemBIST 网表的仿真结果

21.2.1 练习后的思考

BIST 的面积有多大，和原始的 DPMem1kx32 比较。

如果 BIST 的逻辑使用任务（task）来写的，会对综合造成什么影响？

21.3 全双工串行/解串器的 DFT

今天我们会完成给 SerDes 添加 DFT。首先会给练习 24 里的 SerDes 添加全双工（full-duplex）的功能，这样它可以用做 PCI Express 的桥。之后，再添加测试逻辑。

21.3.1 全双工的串行/解串器

全双工的设备有一对串行数据线，一条输入，一条输出。两条线是独立的，可以同时接收和发送数据。

为了使讨论具有普遍性，假设图 21.11 左右两侧的 FIFO 深度不同。假设系统 A 的 FIFO 深度是 8，系统 B 的 FIFO 深度是 16。由于全双工桥往往工作在不同芯片之间，因此，假设的条件是很有可能出现的。例如，通信芯片可能来自于不同的供货商，它们采用了不同的技术，不同的 IP 设计出来的。

理解全双工桥的层次结构是很重要的。每一个 SerDes 设备都可以作为全双工里一端（包含发送和接受）。因此，两个 SerDes 可以组成完整的全双工桥。具体的组合方式可以有两种，图 21.12 画出了这两种方法的区别。

我们选择上面的那种实现方法。这使得在设计中例化我们的 SerDes 很方便。在实际中，串行和解串系统中（或一个芯片里）相隔的距离可能会比较远，因此将并行数据转换成串行数据再来传输是很有必要的。

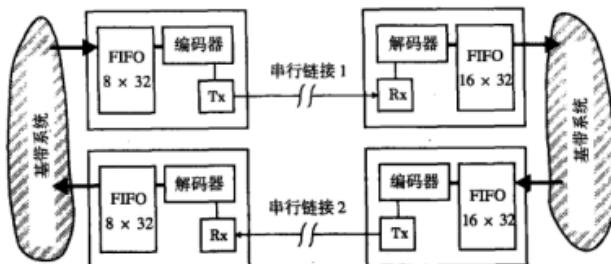


图 21.11 练习 24 里的两个 SerDes 组成了一个全双工的通道。在设计的两侧用两个不同的 testbench 来表示两个不同的通信系统。FIFO 的深度因为不同系统的数据特征而不同

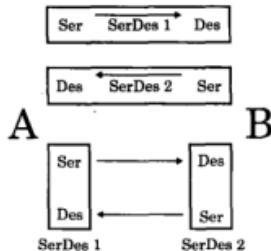


图 21.12 用一对 SerDes 组成全双工系统的两种办法

21.3.2 加入测试逻辑

在开始本练习之前，可以先思考下面的问题：

- 怎样使 SerDes 设计具有可测性？
- 如果没有测试逻辑，可观测性会怎样？
- 应该在什么地方添加断言？
- 应该加入多少内部扫描？
- 边界扫描有用吗？

21.4 练习 26：测试 SerDes

在 Lab 26 里新建子目录完成本练习的内容。下面有详细的具体步骤。

练习步骤

第 1 步。整理全双工串行/解串器的端口。在目录 Lab26 下新建子目录 FullDup，在 FullDup 下再新建一个名为 SerDes 的子目录。把目录 Lab24/Lab24_Ans/Lab24_Step08 下的内容都复制到这个 SerDes 目录里。在进行这个练习的时候，先用答案目录中的源代码。如果你愿意，可以在完成练习之后再用自己实现的 SerDes 把这个实验再做一遍。

下面，需要把练习 24 里的单向 SerDes 升级成全双工的 SerDes。

把目录 SerDes 下除了 SerDes.v 之外的所有文件都复制到上一层目录 FullDup 中。包括以下文件：SerDes.inc，SerDesFormats.inc，SerDes.vcs 和 SerDesTst.v。SerDes.v 和原目录中的 4 个子目录保持不变。

把 FullDup 目录中的新文件的文件名里的 SerDes 都改成 FullDup。不要去改文件里的内容。

第 2 步。 复制一个 testbench 作为模板，在它的基础上为 FullDup.v 升级。

把 FullDupTst.v 另存为 FullDup.v。编辑 FullDup.v，把除了 SerDes 实例及其端口映射之外的部分都删掉。

在 FullDup.v 里，把剩下的这部分代码再粘贴复制一次。把排在上面的实例名叫做 SerDes_U1，把下面的实例名叫做 SerDes_U2。

打开子目录 SerDes 下的 SerDes.v，把 SerDes 的模块头声明复制到 FullDup.v 的顶部。把模块名改成 FullDup。

现在，FullDup.v 包含了 SerDes.v 的模块端口声明和两个不同名字的 SerDes 的实例 FullDup.v 里的顶层模块名为 FullDup。

下一步，将对 FullDup.v 中的 SerDes 做一些小改动。然后才能完成整个全双工的 FullDup。

第 3 步。 重新定义 FIFO 的大小。将图 21.11 中的两个系统 A 和 B 的 FIFO 的位宽和深度都改变是很有趣的。保持 FIFO 的宽度为 32 比特（包含奇偶校验的话是 33 比特），只修改 FIFO 的深度。由于我们将 FIFO 状态机里的计数器设计成可以自行翻转，因此，FIFO 的深度必须是 2 的整次幂。

如图 21.11 所示，A 和 B 中 FIFO 的深度不为 32，A 中 FIFO 的深度是 8，而 B 中 FIFO 的深度是 16。因此，对 SerDes 的 A 和 B 两部分来说，需要两组不同的参数。

删掉 FullDup.v 里的 AWid 旧参数，用 4 个新参数替换它。每一个新参数对应于一个 FIFO。结果如下所示：

```
module FullDup #(parameter DWid = 32
               , RxLogDepthA = 3, TxLogDepthA = 3 // 3 -> 8 words.
               , RxLogDepthB = 4, TxLogDepthB = 4 // 4 -> 16 words.
               )
... (port declarations) ...
```

新的参数应该传递到 SerDes 的实例中。把 SerDes_U1 作为图 21.11 上部分的那个串行通路 (Serial Link 1)。

先不管端口映射，这样来传递参数：

```
...
SerDes #( .DWid(DWid)
          , .RxLogDepth(RxLogDepthB)
          , .TxLogDepth(TxLogDepthA)
        )
SerDes_U1 ( .ParDataOutRcvr ... // A sender; B receiver.
           ...
           SerDes #( .DWid(DWid)
                      , .RxLogDepth(RxLogDepthA)
                      , .TxLogDepth(TxLogDepthB)
                    )
SerDes_U2 ( .ParDataOutRcvr ... // B sender; A receiver.
           ...
```

SerDes 还不能使用这些新参数，因此我们需要修改它使它可以接受这些新参数。

FullDup 里的 SerDes 需要两个 FIFO 深度的参数。一个是接收数据（解串）FIFO，一个是发送数据（串行）FIFO。对 SerDes 这一层来说，A 和 B 是没有区别的。在文件 SerDes.v 中，把 AWid 改成 RxLogDepth 或 TxLogDepth。Tx 的参数对应于 Serializer.AWid，Rx 的参数对应于 Deserializer.AWid。在模块头里指定深度信号的宽度为 5，这将覆盖例化时 3 和 4 的参数值。

没有必要修改低一级的参数。顶层定义的参数将会被传递到下一层。

由于我们之前在参数的传递上做了很多功夫，现在只需方便地修改顶层的参数即可，内部的参数值和参数名都不需要修改。

第 4 步。完成全双工串行 / 解串器的连线。接着，我们需要完成 FullDup.v 的端口，并把它们和 SerDes 相连。

除了串行的数据线，SerDes 之间不应该有其他通信，这使得我们的设计变得简单。接下来，应该给连接到实例 SerDes 的连线和 FullDup 的端口都取上合理的名字，并且尽可能地将 FullDup 的 I/O 数量减到最少。

首先，来把整个系统的数据流关系理顺：SerDes_U1 把数据从 A 传送给 B（参见图 21.11）；SerDes_U2 把数据从 B 传送给 A。因此，在 A 侧，SerDes_U1 的串行数据输出应该和串行数据线 SerLineXfer 相连。A 侧串行器的输入是 ParDataIn, InParClk, InParValid, Reset 和 TxRequest。B 侧 SerDes_U1（解串器）的输出为 ParOutRxClk（B 时钟域）和 ParOutTxClk（A 时钟域）。B 侧解串器的输入为：OutParClk, Reset 和 RxRequest。

SerDes_U2 的连接关系和上面相反。

为了避免混淆，接着来修改连线的名字。首先，把 A 或 B 加在连线信号名的前面，说明它们来自系统的哪一端。当决定了端口的设计之后，再把这些 A 和 B 移至信号名的最后，例如：RxLogDepthA。

这样对实例的引脚进行了处理。为了避免连续的基本错误或冲突，我们从对它们重命名开始。开始时，我们不确定是“A”还是“B”；当决定之后，把选定的字母移到边线名的尾部，如“RxLogDepthA”等。

因此，我们从对各个 SerDes_U1 和 SerDes_U2 的 A 连线增加前缀“A”，以及为各个连在 SerDes 实例上的其他连线增加前缀“B”开始。例如，SerDes_U1 中，我们有 AInPariStim, AInParClkStim 等。这样一来，唯一没有“A”或“B”的连线为上面给出的 A 和 B 之间共享的 ResetStim。

之后，回到模块 FullDup，把当前的端口声明再粘贴复制一次。把其中一份的端口名前面都加上“A”，另一份都加上“B”。

下面，让我们来简化剩下的工作。

简化 A，删掉 FullDup 里的一个 Reset 端口，只剩下一个。

简化 B，A 和 B 之间的数据交互，是系统内部的数据传递过程。因此，串行数据线不需要对 FullDup 外的逻辑可见。因此，声明两条串行线：SerLine1 和 SerLine2，并用它们分别替换掉 SerDes_U1 和 SerDes_U2 里 SerLineXfer 端口上的连线。剩下的输出端口都没用了，删掉它们。

简化 C，假设 A 和 B 工作在不同的时钟域。在 A 里，采样 SerDes_U1 输入数据的时钟和输出数据给 SerDes_U2 的时钟是一样的。新建一个名为 ClockA 的时钟输入，连在 SerDes_U1.

InParClk 和 SerDes_U2.OutParClk 上。在新建一个名为 ClockB 的时钟输入，请读者自行连线，并删除多余的端口。

简化 D，在练习 24 结尾的时候，在可选练习里给 SerDes 增加了两组并行的数据输出：一组数据受接收时钟驱动，另外一组受串行数据中的并行时钟驱动。这两个时钟分别叫做 ParDataOutTxClk 和 ParDataOutRxClk。这样做的目的是为了便于观察它们之间的差异。

如果读者已经完成了这一步，每一个 SerDes 都会有并行的两组输出端口。FullDup 不需要这两组端口，仅需要输出受接收时钟驱动即可。删掉 ParDataOutTxClk 端口，如果它被注释掉了，也删掉它。这样，在每一个 SerDes 实例里只有一组并行输出数据端口。

简化 E，现在来看输入 RxRequest 和 TxRequest。这两个输入使得调试 SerDes 和 FIFO 变得很方便。因此，把这 4 个请求端口保留下来。声明 4 个线型变量，用它们分别给这 4 个端口赋值。下面将给出这段代码，把 TxRequest 固定置成 1；用输入信号 ParValid 实现对两端的控制：当任意一端中的 ParValid 有效时，这一端中的 TxRequest 就会有效。这使得我们可以移除 FullDup 端口并简化 SerDes 的操作。

简化 F，封装。除了 ClockA、ClockB 和 Reset，修改 FullDup 的端口名，输入端口以 In 开头，输出端口以 Out 结尾。同样，除了 Reset，把所有的 A 中的端口名改成以 A 结尾，把所有的 B 中的端口名改成以 B 结尾，这使得原来的 A 和 B 前缀变成了后缀。目的是让信号名更有规律，避免错误。

在完成了上面几步之后，FullDup.v 看起来应该是这样的：

```

'include "FullDup.inc" // timescale & period delays.
//
module FullDup
  #(parameter DWid = 32           // 32 bits wide.
    , RxLogDepthA = 3, TxLogDepthA = 3   // 3 -> 8 words deep.
    , RxLogDepthB = 4, TxLogDepthB = 4   // 4 -> 16 words deep.
  )
  ( output[DWid-1:0] OutParDataA, OutParDataB
  , input[DWid-1:0] InParDataA, InParDataB
  , input InParValidA, InParValidB
  , ClockA, ClockB, Reset
  );
//
wire SerLine1, SerLine2
  , RxRequestA, RxRequestB, TxRequestA, TxRequestB;
//
assign RxRequestA = 1'b1;
assign RxRequestB = 1'b1;
assign TxRequestA = InParValidA;
assign TxRequestB = InParValidB;
//
SerDes #((
  .DWid(DWid)
  , .RxLogDepth(RxLogDepthB)
  , .TxLogDepth(TxLogDepthA)
))
SerDesU1 // Ports reordered:
  (.ParOutRxClk(OutParDataB), .SerLineXfer(SerLine1)
  , .RxRequest(RxRequestB), .ParDataIn(InParDataA)
  , .InParValid(InParValidA), .TxRequest(TxRequestA)
  , .InParClk(ClockA), .OutParClk(ClockB), .Reset(Reset)
);
//
(similarly for SerDes.U2; but, with SerLine2, and with 'A' & 'B' suffices reversed)

```

简单地做一下仿真，检查连线、文件名等有没有问题。有一些仿真工具可能会默认地按相对路径的方式读取 `include 文件中的内容。大家需要注意一下。

可以修改 SerDesTst.v 这个 testbench（在第 1 步里已经改成了 FullDupTst.v）。如果还要加快速度，把目录 Lab26_Ans/FullDup_Step4 下的 FullDupTst.v 复制过来用就行了。

用仿真工具检查所有的 SerDes 实例，FIFO 实例。确保没有关于位宽等的警告信息，这样，练习才能正常的继续下去。图 21.13 和图 21.14 是一些 FullDup 的仿真结果。

第 5 步。加入断言。在 Lab26 目录下新建子目录 FullDupChk。在这个新目录里完成这一步练习。

在串行/解串设计中，除了 FIFO memory 模型的奇偶校验断言，忽略了所有的断言和时序检查。现在让我们来解决这个问题。

在 FullDup 这一级里，是看不到 FIFO 的端口的，但仍然需要知道 FIFO 什么时候为满，什么时候为空。而满和空的标志位于 SerDes 中，以及例化出的两个实例 Serializer 和 Deserializer 的端口中。

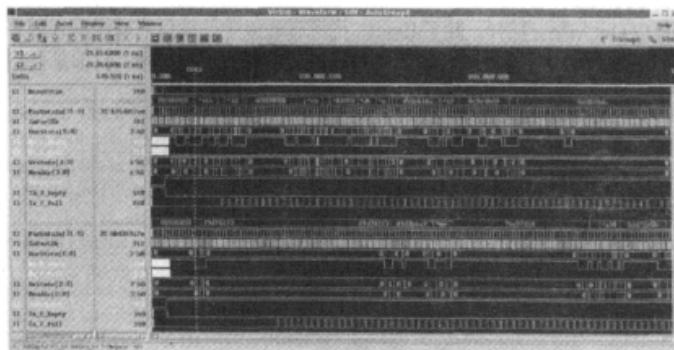


图 21.13 全双工 SerDes 的仿真波形。Serdes U1 的波形在上半部，U2 在下半部。每个方向上的数据是随机且不同的。A 和 B 的 FIFO 的深度不一样

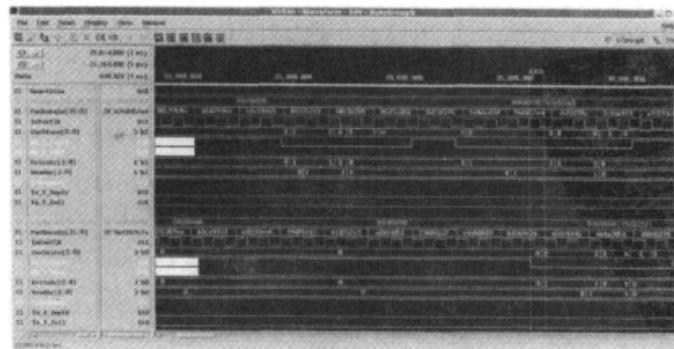


图 21.14 两个并行时钟之间的时钟歪斜

给信号 SerDes 增加 4 个断言，两个位于 Serializer，两个位于 Deserializer，用来检查 FIFO 标志无效的情况。每一个断言都应打印输出警告信息。断言可以是你自己设计的，也可以是第 8 章中练习 11 里产生的断言。打开 %m 选项输出断言是在哪个实例里被触发。

打印的警告信息并不是说设计出现了问题，但这说明了 A 和 B 之间有可能漏掉了一些数据，而这并不是由 FullDup 的硬件检测出来的。我们可以利用这些断言警告来判断 FullDup 的 FIFO 深度是否需要改变。

完成修改之后仿真。你应该能观察到断言检查到了数个 FIFO 为空的情况。

如果你还没有完成，修改 testbench 使能 Serializer 的输出 (Tx)。在你的 testbench 应该保证 ClockA 和 ClockB 是完全独立的。为了节省时间，可以从目录 Lab26_Ans/FullDup_Step4 中把 testbench 复制过来。这个 testbench 比较复杂，它产生了随机的输入数据且两个时钟都有各自独立的漂移。

第 6 步。检查 DesDecoder 控制信号的脉宽。在目录 FullDupChk 中完成这一步。用目录 Lab26_Ans 里的或相近的 testbench 做仿真，输出 DesDecoder 里的信号。模块里的 ParValid, ParClk, doParSync 和 SyncOK 的脉宽都可以被检查到。

现在来研究这些脉宽。我们希望检查到最窄的正脉冲。在 DesDecoder 里加入 \$width 检查，当正脉冲的宽度小于 specparam 定义的时间时，会输出违例警告。给每一个时序检查都声明不同的 specparam。

在仿真时改变 specparam 的值。例如，在设置了 specparam 和 4 个 \$width 时序检查之后，把一个 specparam 设为 0，把其余的 3 个设成 100 (100 ns)；这样只有一个会有效。然后，运行一个相对长时间的仿真，比如说 $500 \times 10^3 \sim 1 \times 10^6$ ns。

如果还没有报告违例，把 specparam 里的非 0 参数设置成 50~100 ns，或更大一些。接着仿真，观察报告的违例信息。如果违例太多，控制仿真工具把仿真停下来。根据违例的提示修改参数，找到不发生违例的最大参数值。

同样，对其余三组 specparam 参数分别仿真，直到找到了最大的不违例参数为止。

现在，任何使得脉宽变得过短的异常状态都会被这些检查检查出来。在下表中记录下你的时序检查宽度。

变量名	最大不违例的宽度 (ns)
ParValid	
ParClk	
doParSync	
SyncOK	

要找到异常脉宽的脉冲是要花些功夫的。水手常说一句话：“小错误也能翻船” (Loose glitch sinks chips)。对芯片设计也一样，小错误也能毁掉整个芯片。

第 7 步。在这一步里，回到目录 FullDup。把目录 Lab24/Lab24_Ans 里的综合脚本作为模板，新建一个名为 FullDup.sct 的综合脚本。

用这个脚本综合这个设计。不添加面积的约束，但是给所有的输出都添加严格的最大延迟限制。不要在脚本里定义时钟。在编译之前，用默认的条件把设计打平，这样才能让产生的网表和下一步我们插入了扫描逻辑的网表能够做比较。记得把面积报告输出到一个 log 文本文件里头去，一会儿会用到。

这一步的目的仅仅是为了说明设计是可综合的。由于没有指定时钟，因此，仿真是有问题的。如果增加了时钟约束，再修正了保持时间的违例，则综合出的网表将是一个可综合且仿真也没有问题的网表。本书附带的光盘里给出了多种可用的综合约束。图 21.15 和图 21.16 给出了完全约束，完整的串行/解串器工程的仿真波形。

第 8 步。插入扫描逻辑。新建一个名为 FullDupScan 的子目录，把 FullDup 目录下的所有内容都复制过去。简单做一下仿真，检查文件的位置和连接。在目录 FullDupScan 里完成剩下的练习。

用综合工具插入扫描的步骤如下：

A. 把复制过来的综合脚本改名为 FullDupScan.sct，修改输出的网表和 log 文件的名字，使得一看就知道是插入了扫描的结果。

在脚本文件里，删掉用练习 25 里插入扫描的命令替换掉编译命令。这些命令可以在文件 Lab25_Ans/IntScan/DC_Scanned/Intro_TopFFScan.sct 中找到。你还需要修改 Clk 和 Clr 的名称，还有扫描时钟的周期。修改 FullDup.v 里的模块头，增加 tms 和 tdi 两个输入，增加 tdo 输出。

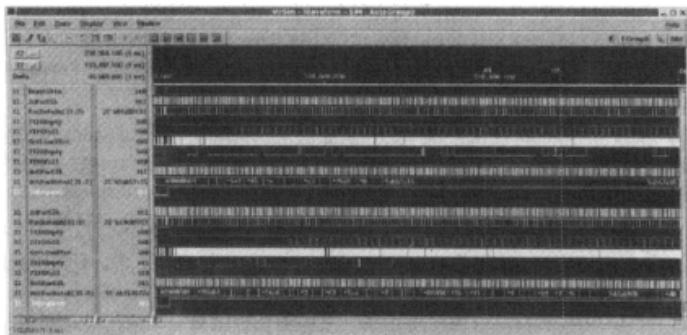


图 21.15 全双工 SerDes 网表仿真的结果



图 21.16 全双工 SerDes 网表仿真的局部细节。A 端的输入并行数据
0x0fd2_8f1f (193 ms) 被正确的传送到 B 端 (235 ms)

B. 综合并插入扫描器件。插入扫描的综合耗时比单独综合的时间长一些。记得输出面积报告，以便和之前的面积报告做比较。

之前的综合约束应该不难满足插入扫描之后的要求。这是因为，扫描只是把普通的存储单元替换了扫描单元。而对每一个寄存器来说，增加的逻辑是只是增加了一个选通逻辑。不考虑库之间的差别，典型的扫描寄存器的面积比不带扫描的寄存器大约30%。这一部分练习的作用是为了说明插入扫描对面积的影响。先不要用这个网表来仿真，由于综合约束里并没有包含时钟，所以 TAP 控制器还不能正常工作，从而电路无法工作在扫描模式下。

C. 用本文工具打开这个插入了扫描器件的网表，看看扫描器件是怎么实现替换的。

21.4.1 练习后的思考

在 FullDup 中，还可以加入什么其他的时序检查？

插入扫描器件后对面积的影响是什么？

给硬件测试工程师提供扫描测试点有多困难呢？换句话说，如果需要设计一条覆盖整个 FullDup 扫描链的扫描向量，这个向量应该有多长呢？我们应该怎样产生这条扫描向量，从而使得它可以覆盖到每个应该被覆盖到的扫描点呢？

21.4.2 补充学习

阅读 The NASA ASIC Guide: Assuring ASICs for SPACE，第 3 章：Design for Test(DFT)。网址是：http://klabs.org/DEI/References/design_guidelines/content/guides/nasa_asic_guide/Sect.3.3.html (最后更新于 2003-12-30)。

如果你对 LFSR 伪随机数产生感兴趣，阅读 Koeter 的文件：What's an LFSR?。网址是：<http://focus.ti.com/lit/an/scta036a/scta036a.pdf> (最后更新于 2007-01-29)。

(可选) 把练习 25 第 8 步中的 BIST.v 清空，只保留端口。自己实现整个 BIST 功能。

第 22 章 SDF

22.1 SDF 反标

现在，我们深入学习有关 SDF 的知识。

22.1.1 反标

反标（back annotation）是指用新的属性来标注网表。通常，标注的属性是延迟的时间。当工具在运行时，新的结果被反馈回网表，从而完成了反标的全过程。网表本身并没有被修改，反标的内容被保存在另外一个文件中。

虽然综合工具可以在优化网表的时候估计路径的延迟。但这些延迟是综合工具内部的负载模型提供的。在给定了路径长度之后，综合工具选择库中的各种元件来达到约束的时序要求。这样估计出来的延迟和在完成了芯片布局布线之后再得到的实际延迟比起来是不准确的。对于估计的准确性来说，10% 的误差已经是一个很好的结果了，更多的情况误差还要大一些。

在完成了网表的布局规划（floorplan）和布局布线（place and route）后，对于芯片中每个元件来说，得到了准确的延时信息。仿真的时候，不再使用库中自带的延时信息，而是使用反标的延时信息。这说明保存延时信息的文件需要包含网表里每一个器件的时序信息。

22.1.2 Verilog 中的 SDF

用来反标网表延迟的标准文件格式是 SDF（Standard Delay Format）。在 IEEE 1497 里有详细的描述；在 IEEE 1364 的 16 节有它和网表的作用描述。除了 SDF，还有其他格式的反标文件，例如 SPEF（Standard Parasitic Exchange Format），一般在进行芯片的物理布局时会用到 SPEF。SDF 包含的是延迟时间，而 SPEF 包含的是电容值。图 22.1 是运用到反标的一个典型的设计流程。本书将不涉及 SPEF 的内容。

SDF或其他反标文件里的时序信息将会替换掉 Verilog 库中 specify 块里的路径延迟，还包括条件延迟。传输路径上的线延迟也可以用 SDF 来反标，但是在 Verilog 里，这部分延迟往往被折算成逻辑块到输出端口的延迟。路径延迟和 SDF 里的关键字“IOPATH”相关联，线延迟和关键字“INTERCONNECT”相关联。关键字“CELL”用来描述任何一个器件或模块。除了三个一组的时序参数，SDF 唯一特殊的用法是它的圆括号；SDF 文件看起来有点像 EDIF 格式的文件。

在 SDF 文件里，SDF 的关键字都是用大写字母来表示的，但是系统并不区分大小写；而对网表里器件的引用是区分大小写的。

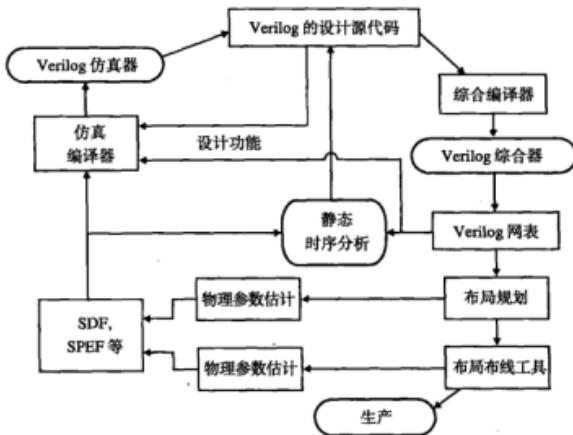


图 22.1 反标在典型的设计流程中所处的位置。估计的延迟或物理参数被保存在设计文件之外的单独文件中，设计工具可以自动将其与网表对应

22.1.3 Verilog 仿真中的反标

这个过程很直接也很简单。综合工具先产生一个网表，然后，在这个网表的基础上，再产生对应于这个网表的 SDF。通常，SDF 是在布局布线之后由布线工具来产生的。另外一个产生 SDF 的简单办法是在综合工具产生了 Verilog 格式的网表之后，直接产生对应的 SDF。

在有了 Verilog 网表以及对应的 SDF 文件之后，用 Verilog 的系统函数 \$sdf_annotation ("sdf_file_name"); 将 SDF 信息反标入网表。这样，当仿真工具开始仿真时，被反标的模块（通常是整个设计）不再使用 Verilog 元件库中的时序信息，而使用 SDF 文件中的时序信息。

如果需要在一个设计里同时反标多个 SDF 文件，问题就变得复杂一些。用关键字 ABSOLUTE（本练习里用到的 SDF 文件里都有它）使得最后一次读到的延迟信息有效。用关键字 INCREMENT 使得处理延迟的方法是累加的。

SDF 小结：

SDF (Standard Delay Format) 定义于 IEEE 1497 之中：

- 它是由三个一组的时序参数组成的。
- 可以由综合工具、静态时序工具或布局布线工具产生。
- SDF 文件包含层次，可以用在任何一层的 Verilog 模块上。
- 使用 SDF 的时候把下面的系统函数包含在 initial 块里：\$sdf_annotation("file_name.sdf");。

22.2 练习 27：SDF

在目录 Lab27 下完成以下练习。

练习步骤

第 1 步。仿真整个设计。在目录 Lab27 下新建三个子目录：orig, ba1 和 ba2。把练习 1 中的 Intro_Top 设计复制到 orig 目录中去。使用包括 TestBench.v 在内的原始文件。

在 TestBench.v 中注释掉 `include Extras.inc 后仿真。让波形窗口显示设计顶层的输入和输出。

如果你用的是 VCS，保存 VCS 配置文件。在做下面的练习的时候，只要打开这个文件，窗口的大小和显示的信号都和保存时是一样的。

注意：不能关掉仿真工具。最容易的办法就是每一步都在一个新的终端窗口中来完成。

第 2 步。用网表仿真。完成了第 1 步之后，把刚才复制过来的 Intro_Top.sct 用目录 Lab27 下的文件替换掉。这个新文件更新了设计规则以保证设计的面积最小。你可以用文本编辑器同时打开两个文件，比较两个文件之间的区别。

用新的 Intro_Top.sct 文件综合这个设计。新的脚本将自动产生网表和 SDF 文件。这个 SDF 的延时信息是基于综合库的线性模型计算出来的，但是它的时序已经比只用 Verilog 设计源文件准确得多了。

新建一个包含 TestBench.v 的文件列表，并且包含 LibraryName_v2001.v 这个 Verilog 库文件。仿真并和第 1 步得到的波形比较。

如图 22.2 所示，这个网表仿真的结果和 Verilog 源代码的仿真结果有些差异。这是因为和 Verilog 源代码里的延迟比起来，库里 (LibraryName_v2001.v) 实际的延迟要小得多。我们明显可以看到，输出 Xwatch 上的毛刺在网表仿真里已经不见了。

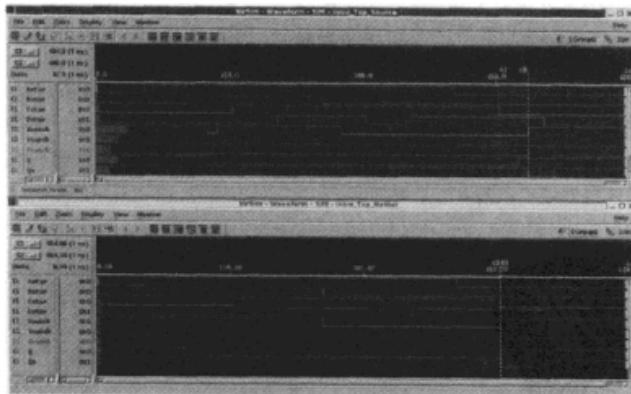


图 22.2 源文件（上面的）和综合出来的网表的时序差别

在完成了对仿真时序的检查之后，保留网表仿真的波形窗口。关掉原始设计的波形窗口或者直接结束它的仿真进程。

第 3 步。反标 SDF 并仿真。把 TestBench.v，网表文件，文件列表和第 2 步产生的 SDF 都复制到目录 ba1 里。

用文本编辑工具打开网表，找到模块 Intro_Top。在模块里增加 initial 块指定 SDF 文件。例如：

```
initial $sdf_annotate(''Intro.TopNetlist.sdf'');
```

仿真并把仿真结果和第 2 步用原始网表仿真的结果比较。两个结果比较起来应该很接近，但是会有很小的时序差异（参见图 22.3）。因为原始网表仿真使用的时序信息是 Verilog 元件库 LibraryName_v2001.v 里提供的估计延迟，而综合工具产生的 SDF 里的时序信息来自于 LibraryName.lib（Synopsys 综合工具使用的是 LibraryName.db）。

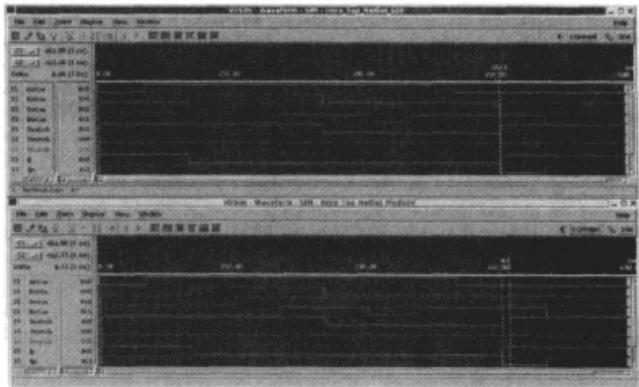


图 22.3 仅用网表（上面的）和反标了 SDF 仿真的时序差别

观察完了波形之后，把第 2 步的仿真波形关掉，保留这一步的仿真波形。

第 4 步。修改 SDF 并仿真观察区别。把第 3 步 ba1 目录下的所有内容复制到 ba2 目录下。

为了理解布局布线工具是怎么输出反标信息的，可以通过修改 Intro_Top.Z 这个输出端口的时序来体会。请记住，这一步仅仅是为了达到教学的目的，而在实际工作中，除非有特殊的需求，我们绝不应该手动修改 SDF。

用文本编辑工具打开网表，找到模块 Intro_Top，并找到输出端口 Z。这个端口可能连在一个反相器上。找到驱动 Z 的最后一个单元，它的单元类型为 ctype，实例名为 inst_name。

打开 SDF，找到如下的这部分：

```
(CELL
  (CELLTYPE '' ctype '')
  (INSTANCE inst_name)
```

在实例名下面，会有 DELAY 和 IOPATH 的语句。当对反标了 SDF 的网表进行仿真时，IOPATH 决定了路径的延迟。第一组三个一组的时序参数是上升延迟，第二组三个一组的时序参数是下降延迟。

你可以打开 LibraryName_v2001.v，看看这里面的延迟信息和综合工具产生的延迟信息有什么区别。而这个区别正是使我们的仿真结果看起来不同的原因。另外，SDF 里的延迟总是针

对某一个具体的实例的，而库里的延迟对应于元件。综合工具用来计算 SDF 延迟的库是 LibraryName.lib，它在 Synopsys 库的安装目录里。

把输出 Z 的 IOPATH 上升延迟改成原来的 20 倍，下降延迟改成原来的 10 倍。用 “//” 注释掉原来的延时信息。保存文件并仿真。新延迟的效果应该非常明显，如图 22.4 所示，和未改动的 SDF 反标文件相比，最终 Z 的上升沿在仿真波形里应该到得更晚且变得更窄。

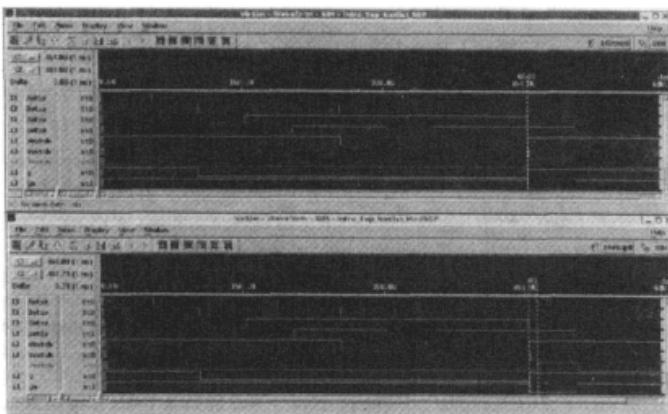


图 22.4 反标后的时序：原始 SDF（上面的）和手动修改过的 SDF

可选练习。在第 4 步里，人为修改 SDF 里的典型延迟和最大延迟，再仿真并观察结果。

22.2.1 练习后的思考

如果 \$sdf_annotation 任务在错误的模块中被调用会怎么样？

怎么反标网表中的一部分逻辑？

22.2.2 补充学习

复习本次练习中的第 2 步。详细分析为什么顶层输出端口 X 在用设计源文件进行仿真时变成了 0，而用网表进行仿真时却没有？

第 23 章 Verilog 语言总结

23.1 Verilog 语言总结

23.1.1 Verilog-1995 和 2001 (2005) 的区别

我们已经多次体会到了 Verilog-1995 和 Verilog-2001 的一个区别：ANSI-C 模块头的声明格式。Verilog-2001 新的特性还包括：支持 generate，多维数组（Verilog-1995 只支持一维数组），有符号数（在 Verilog-1995 里，只有整型和实型可以是有符号的），用来递归调用函数或任务的 automatic 关键字。此外，SDF 的用法也略有不同，VCD 文件的特性也有部分增强。还有一些改动这里就不再讨论了，具体请参考 Sutherland 的文章（这篇文章也是今天的补充学习的内容）。

23.1.2 可综合的 Verilog 语法规子集回顾

下面是简要的回顾。

综合工具不支持的语法如下所述：

- 延迟表达式
- Initial 块
- 时序检查
- Verilog 的系统任务或函数

总体来说，只要是和仿真时间有关的都不能被综合。综合工具在约束和优化时序时使用的器件延迟信息来自于它的库模型，并不是来自于 specify 块或其他的 Verilog 结构。

如果赋值表达式中含有延迟信息，这些延迟会被忽略掉。

综合工具可以综合 generate 块，循环或条件表达式。

无论在不在一个 always 块，都不要同时用阻塞和非阻塞表达式对同一个变量赋值。良好的编码习惯可以让综合工具综合出更优化的电路。

对于类似于 latch 的功能，即是说对电平有效的锁存结构，综合工具综合这种结构是有困难的。为了避免 latch 带来的问题：(a) always 块的敏感变量列表不要漏掉敏感变量；(b) 列举 case 的时候要列举全。

23.1.3 本书中没有涉及到的 Verilog 结构

我们已经基本上讲述了所有关于 Verilog 语言的内容，包括对 VLSI 设计用处不大的内容也讲到了。但是，还是省略了一些不重要或基本上没有工具支持的语法结构。

下面是完整的列表：

- 所有的 Verilog 关键字都列在 IEEE Std. 1364 的附录 B 中。Thomas and Moorby (2002) 的 8.5 节也有一个完整的列表，表中包含了我们没有讲的内容。
- 文件输入输出处理。对应的系统任务是 \$readmemb 和 \$readmemh 等。
需要注意的是，Verilog 文件系统在 Windows 里的分隔符是 “/”，而不是 “\”。
Thomas and Moorby (2002) 里的 F.4 节和 F.8 节给出了完整的文件系统函数。在 Bhasker (2005) 的 3.7.4 节中也有关于 I/O 文件系统的讨论。
- 系统任务和函数。由于本书以设计和综合为主，伴随着这条主线，我们讲到了相应的系统任务。有的系统任务没有讲到。在 Bhasker (2005) 的 10.3 节里有关于系统任务的详细内容。
- 编译指令（`directive）。这部分是非常容易理解的。对很多指令来说，只从名字就能知道它的功能。本章后面会有详细的列表。
- 属性。布尔表达式被符号 “(* ” 和 “*)” 包起来。这种写法和 Pascal 语言的注释的语法是一样的。例如，(* dont_touch U1.nand002*)。Verilog 的属性和工具密切相关，被用来控制综合或仿真行为。例如（//synopsys…）这样的用法就不被所有的工具所支持。
- Defparam, force-release, assign-deassign。虽然 force-release 在构造特殊功能的 testbench 和调试时比较有用。但是，正如我们已经讲到的那样，应该避免用这样的写法来建模。
- 把模块的输出端口声明为寄存器（reg）。这是 Verilog-2001 从 Verilog-1995 继承下来的一个特性。在实际应用中，不推荐这种做法，这使得端口上的延迟不可见。而且，对于 wire 和 reg 类型，还需要使用各自不同的赋值语句（assign 和 always）。和声明内部寄存器再连续赋值给线型相比，这种做法并没有节省多少时间。
- Verilog PLI。本章的后面会讲到它。

另外，除了 Verilog，还有一部分内容我们也没有深入讨论。

- 没有深入讨论综合约束。Synopsys DC 的命令参考手册讲到了数百条关于综合约束的命令，上千条选项，如果打印的话会超过 2000 页。
- Synopsys DC Shell（dc_shell）或 design_vision 模式。Shell 的语法和 Tcl 相近，但是 Synopsys 已经不推荐使用它了。
- Synopsys 设计约束语言（Synopsys Design Constraint Language，SDC）。这种语言是 Synopsys 基于 Tcl 发展起来的免费语言。SDC 使得脚本更具可移植性，可以为多种工具所用，例如：综合工具，静态时序工具，功率估计，覆盖率分析和形式验证。如果工具支持的话，它还可以和 VHDL 或其他的语言，例如 SystemVerilog 或 SystemC 同时使用。

23.1.4 所有的 Verilog 系统任务和函数列表

在 IEEE 1364 Std 的 17 节中有详细的描述。所有的系统任务和函数可以分成 10 类，下表是一个完整的分类。通常从名字就能推测出来它的功能。当然，Std 和其他文档里有详细的说明。很多系统任务只用在 testbench 里。

据作者所知,还没有工具支持所有的系统任务或函数。然而,所有的工具都支持本书中用到的那些任务和函数。本书中用到的部分在下表中以粗体表示。

如果为了实现某种特殊的功能需要用一种新的,以前没用过的任务或函数,最好先试试手头的工具是不是支持这个任务或函数。

类 型	任务和函数名
Display	\$display \$displayb \$displayy \$displayo \$monitor \$monitorb \$monitoro \$monitoroff \$strobe \$strobed \$strobede \$strobeo \$writel \$writelb \$writelb \$writelb \$writelb \$writelb \$writelb
Time	\$time \$time \$realtime
Sim. Control	\$finish \$stop
File I/O	\$fclose \$fdisplay \$fdisplayb \$fdisplayy \$fdisplayo \$fgetc \$fflush \$fgets \$fmonitor \$fmonitorb \$fmonitorh \$fmonitoro \$readmemb \$swrite \$swriteo \$format \$fscanf \$fread \$fseek \$fopen \$fstroke \$fstrokeb \$fstrokeb \$fstrokeo \$ungetc \$ferror \$rewind \$fwrite \$fwriteb \$fwritech \$fwriteo \$readmemh \$fwriteb \$fwritech \$odf_annotation \$ssacf \$ftell
Conversion	\$bitstorel \$itor \$signed \$realtobits \$rtoi \$unsigned
TimeScale	\$printtimescale \$timeformat
Command Line	\$test \$plusargs \$value \$plusargs
Math	\$clog2 \$ln \$log10 \$exp \$sqrt \$pow \$floor \$ceil \$sin \$cos \$tan \$asin \$acos \$atan \$atan2 \$hypot \$sinh \$cosh \$tanh \$asinh \$acosh \$atanh
Probabilistic	\$dist_chi_square \$dist_exponential \$dist_poisson \$dist_uniform \$dist_erlang \$dist_normal \$dist_t \$random
Queue Control	\$q_initialize \$q_remove \$q_exam \$q_add \$q_full
VCD	\$dumpfile \$dumpvars \$dumpoff \$dumpon \$dumpports \$dumpportoff \$dumpportson \$dumpall \$dumpportsall \$dumpportlimit \$dumpportslimit \$dumpflush \$dumpportsflush \$comment \$end \$date \$enddefinitions \$scope \$timescale \$uscope \$var \$version \$vdclose
PLA Modelling	\$async\$and\$array \$async\$and\$array \$async\$or\$array \$async\$nor\$array \$sync\$and\$array \$sync\$and\$array \$sync\$or\$array \$sync\$nor\$array \$sync\$and\$plane \$async\$and\$plane \$async\$or\$plane \$async\$nor\$plane \$sync\$and\$plane \$sync\$and\$plane \$sync\$nor\$plane \$sync\$nor\$plane

23.1.5 所有的 Verilog 编译指令

下面列出了所有的 Verilog 编译指令 (Compiler Directive)。据作者所知,还没有工具支持所有的这些编译向导。如果用到了工具不支持的编译指令,工具一般会忽略这个指令或输出警告信息。前面讲过,如果用 `defined 定义过一个宏,那么最好在文件的结尾再加上 `undef。

```
'begin.keywords 'celldefine 'default.nettype 'define 'else
'elsif 'endcelldefine 'endif 'end.keywords 'ifdef 'ifndef
'include 'line 'nunconnected.drive 'pragma 'resetall 'timescale
'unconnected.drive 'undef.
```

23.1.6 Verilog PLI

PLI 是可编程语言接口 (Programming Language Interface) 的缩写。严格来讲,PLI 并不是 Verilog 语言的一部分。因此,在前面的章节我们没有讲到它。但是,在 IEEE 1364 这个 Verilog 规范中仍然讲到了 PLI。

PLI 其实就是 C 语言,它是基于 C 语言的,常用的,可调用库的总成。PLI 使得 Verilog 的仿真工具可以运行用户自行定义的系统任务,甚至于运行的不全是 Verilog 仿真,而是利用仿

真工具来运行和 Verilog 相关的应用的仿真。例如：错误仿真，时序的运算或网表报告产生器等。和系统内建的任务一样（例如 \$display），用户自定义的任务也是用“\$”开头的。

对于电路设计工程师来说，PLI 的用处可能不大。但是对于 EDA 工具的开发团队来说，PLI 是很有价值的。一个典型的应用是：如果某一个厂商提供的仿真工具不支持某种特性，另一家提供商则需要通过 PLI 来让这个仿真工具支持这种特性。

如果在 VCS 仿真工具编译新模块的时候仔细观看它打印的消息，我们会发现 VCS 是一个专用的编译，链接工具，它把要仿真的代码变成了可执行的 C 程序。VCS 的 GUI 是一个已经编译好的程序，它的作用是处理这些可执行 C 程序进程间的通信，可执行 C 程序的 I/O 信息被 GUI 格式化并打印出来。为了在仿真时节省内存，VCS 可以工作在没有 GUI 的文本模式，具体的操作请查阅 VCS 的帮助文件。对于其他的工具，例如 QuestaSim，Silos，Aldec 和其他的 Verilog 仿真工具，都可以在不打开 GUI 的情况下进行仿真。

过去，Verilog 的仿真基于解释（interpreted，就像 BASIC 和 shell 脚本一样）而不是编译（compile）。除了运行仿真的时候比较慢以外，基于解释的仿真和现在基于编译的仿真差不多的。

PLI 的工作基于仿真工具（实际上是基于其内部的数据结构和运行的代码），如图 23.1 所示。仿真工具可以获取的原始数据有两种。一种是 Verilog 的设计源文件，另一种是综合工具产生出来的 Verilog 格式的网表。它们都可以通过脚本和文本编辑器被修改。而仿真工具的内部数据结构则是不公开的二进制格式。对于一个可仿真的设计来说，这个内部数据必然会包含能表示设计的对象，器件的模型，连线和其他一些 Verilog 的对象。总线和端口的每一比特都会被单独表示。如果是网表仿真，所有的循环肯定已经被展开。在这些对象之上，仿真工具的数据数据库包含了状态，延迟，仿真时间等信息。

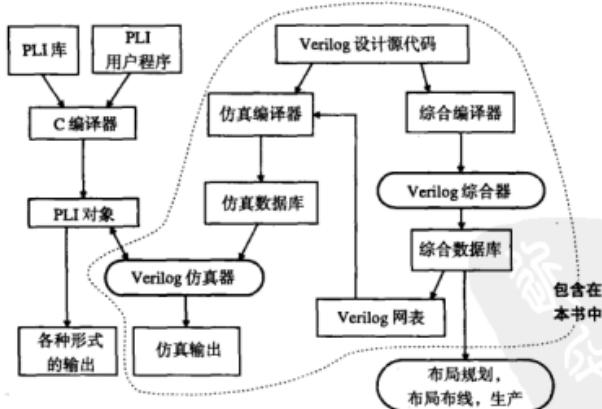


图 23.1 PLI 和仿真、综合的关系图

对于 Verilog-2005 来说，PLI 仅由 VPI（Verilog Procedural Interface）组成。在其之前的 PLI 包含两部分：TF（task function）和 ACC（Access）。现在，这两部分都被融入了 VPI 中。Palnitkar (2003) 的第 13 章介绍了这三种分类的更多细节。本书将不再继续讨论这部分内容。

23.2 课后练习（继续完成练习 23 及以后的练习）

23.2.1 补充学习

阅读 Stuart Sutherland 在 HDLCon 2000 大会上宣讲的文章，“The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need it.” 这篇文章主要讲述了 Verilog-2001 增强的那部分特性，内容非常精彩。网址是 http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf（最后更新于 2005-02-03）。

阅读 Thomas and Moorby (2002)前言 pp.xvii-xx 关于 Verilog-2001 增强特性的总结。

阅读 Palnitkar (2003) (可选)

阅读 9.5.1 节和 9.5.5 节，内容是文件 I/O。并查看随书附带光盘中的 Memory.v 的例子。

阅读 14.6 节关于可综合逻辑编码的内容。

阅读第 14 章所有关于可综合逻辑的内容。

阅读第 13 章关于 PLI 的内容。



第24章 深亚微米的问题及其验证

24.1 深亚微米的问题及其验证

24.1.1 深亚微米设计遇到的问题

对于使用Verilog的设计工程师来说，我们只从电气特性的角度来讨论深亚微米（deep submicron）设计，不讨论和制造工艺相关的话题，例如：离子注入（ion implantation），光学近邻效应修正（Optical Proximity Correction，OPC），曝光掩模的紫外线波长，浸液光刻（immersion lithography）等。

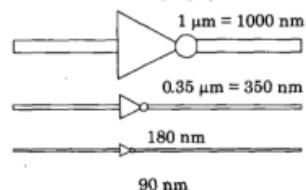


图 24.1 不同工艺器件的大小和线宽的示意图

20世纪90年代后半期提出了深亚微米这个概念。它指的是制造工艺不超过 250 nm ($1/4\text{ }\mu\text{m}$) 的电路。在这种电路中，路径延迟开始超过门器件的延迟，从而成为设计时需要主要考虑的延迟因素。图24.1是它们之间相互大小的一个示意图。

深亚微米设计有一些普遍性的问题：如果我们把线宽叫做 L ，那么 L 会对整个延迟有着重要的影响。下面是一些影响延迟的重要原因：

- 电子在晶体管里流动的速度和导体的面积 L^2 是成比例的。但是，路径上的延迟和 RC 时间常数却不能满足这个比例关系。随着线宽的减小，电路 R 线性的增大，电容 C 也随之减小。因此，虽然制造工艺让线宽变得更小了，但 RC 常数还是基本上不变的。在计算路径延迟时，路径的长度变得更重要了。
- 当门器件缩小之后，器件之间的连线长度没有像 L 一样显著的减小（参见图24.1）。
- 由于芯片上集成了更多的门，假设这个数值为 N 。门器件之间的互连线大约是 $2N$ 到 N^2 之间。虽然 L 减小了，但是随着芯片复杂性的提高，门器件的平均最终扇出增加了。因此，计算负载能力的估计延迟是最重要的一项工作。

除了传输延迟导致的时序问题， 90 nm 及其以下工艺还有漏电和噪声的新问题。特别是当尺寸缩小之后，电容的变化以及随之而增加的交叉互耦（cross-coupled）噪声的影响变得更加明显了。

对于一个大型的低电压设计来说，小的间距需要更加准确的库来描述门器件的特性。

门器件的延迟不能仅仅把库和综合模型里的延迟数值直接加起来，还要考虑到器件输出连接的器件，输入电压的转换速率（slew rate），电流的驱动能力和器件在芯片上的工作条件（工艺，电压和温度，简称PVT）。

随着器件的缩小，芯片制造时发生随机问题的可能性增加。因此，需要芯片制造商用复杂的铜线工艺替换铝线工艺。

随着器件的缩小和时钟频率的提高，器件的漏电流成为了一个最大的问题。为了减小功耗，必须要控制器件的翻转率，并把芯片的供电电压从5V降低成1V左右。

由于功率和电压的平方成正比，为了控制功耗，可以让芯片的不同区域使用不同的电压。因此，电压转换器就显得很有必要了。当不同电压域的时钟和数据有交互时，都必须经过电压转换。对许多电池供电的数字设备来说，为了实现低速模式或睡眠模式，设计者利用专门的隔离单元把芯片里不同位置的逻辑隔离开来。通常睡眠模式需要额外的存储单元保存当前的状态。

除了芯片制作过程中的随机因素，在制作掩模时，光刻时的光的衍射（diffraction）也成为一个需要考虑的重要因素。为了提高制造精度，需要引入光学近邻效应修正（Optical Proximity Correction）技术。图24.2展示了衍射修正的需求。

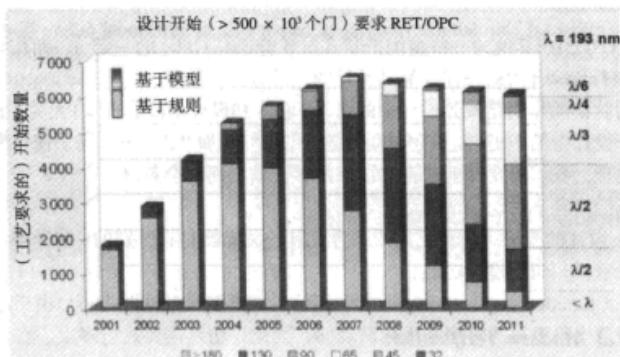


图24.2 数字设计里光学近邻效应修正的需求（数据来自W.Staud, Invarium, Inc.）

衍射问题和量子效应实际上是同一个问题。对光来说，光量子的波长是掩蔽光子的波长。对制造结构来说，量子具有的波长是逻辑栅电子的波长。Low- k 电介质的漏电流可以根据电子的量子隧道效应直接计算出来，而量子隧道效应决定了片上CMOS器件的场效应晶体管元件的性能。

由于设计变得越来越大，内部全扫描链的处理也变得更加的复杂。这种条件下的内部扫描变得有层次并且可能由多条链组成，最后通过选通器连到TAP上。TAP来控制当前测试哪一条链。

在现代时钟速率下，RF交互也变得重要起来，特别是对于串行传输领域，例如以太网，无线天线接口和PCI高速总线。

所有的这些问题都可以通过合理的物理设计来解决。虽然物理设计并不在本书讲的范围内，但是如果工程师在编写Verilog源代码时有良好的设计风格，功能模块的划分也很合理，则可以降低物理设计时的难度。

制造工艺还会继续发展，直到无法解决量子的不确定性的时侯才会在这条路上停下来。目前还不清楚数字电路将来最先进的工艺能发展到什么程度，但应该是小于 10 nm，大概是原子直径的 50~100 倍。

24.1.2 更大的问题

我们要记住，随着芯片尺寸不断的缩小，深亚微米的设计问题会越来越严重。然而，根据业界的统计，即使对 130 nm 的工艺来说，导致重新流片（respin）最主要的问题仍然是功能错误（逻辑设计的失误）和与时钟相关的设计错误。由这些数据可知，对于 90 nm 工艺来说，不能正常工作的芯片里只有约 25% 的是由于以上提到的深亚微米带来的问题导致的。

由此看到，使用 Verilog 的设计工程师应确保自己设计功能的正确性，确保信号的时序，为自己的系统设计正确的时钟。这样能最大可能减小流片失败的风险。

24.1.3 现代验证

Palnitkar (2003)和其他的一些组织把断言看成验证（工具）的一部分。而作者认为应该把断言和时序检查与综合约束一样，都是设计的一部分。

作者认为验证的作用是找到设计中的错误，它应和设计手段（工具）无关。断言通常由设计工程师出于设计上的考虑（应该怎样仿真这个设计）而加在设计中。为了使得发现的问题能够被设计者发现，我们需要保证断言的结果能够被输出而不会忽略。

下面让我们接着来讨论验证的话题。

对 Verilog 设计的验证和仿真工具或综合工具的编译器是有区别的。编译器仅仅检查源代码是否有语法错误，而不检查功能。

功能验证（function verification）通常利用仿真工具和设计同时进行。由于综合工具是基于器件库而不是代码里写的时序来优化网表的，因此，仿真需要进行多次。在反标了延迟之后，还需要对网表进行仿真。

现在的数字芯片很容易就会超过 500 万门的规模，从而可能无法在传统的工作站上仿真或调试。因此，工程师们往往利用分布式的大型计算系统，或利用 FPGA 硬件来仿真。

时序验证（Timing Verification）是在芯片流片（tape out）之前最重要的一步。芯片中的路径有可能上千万条，任何一个路径时序上的问题都有可能导致再花上百万美元重新设计或生产芯片。另外，如果由于设计上的失误使得芯片推迟上市，造成的损失可能会更大。

时序验证的工作主要不是由仿真工具来完成的，因为仿真工具无法对上千万条路径都进行仿真。而静态时序分析工具可以从网表和 SDF 里分析出芯片的时序。对于一个现代的大型设计来说，STA 往往一天就能运行完。

24.1.4 形式验证

在进行功能仿真时，重要的一步是形式验证（formal verification）。形式验证工具工作时是静态的，使得短时间完成一个大型设计的验证成为可能。

形式验证可能分成如下几类：

- 等效性检查 (Equivalence Checking)。这是最流行的一种形式验证方式。工具会检查一个新网表和一个“黄金标准”(golden standard)网表的一致性。通常这个工具输出的结果可能为以下几种：等效，功能上不同，不确定是否等效。

如果直接拿网表和 RTL 的源代码来对比，结果有可能不是很理想。因为形式验证工具可能会对代码风格有一些特殊的要求。但是，任何可综合的代码都可以用来和其对应的网表进行等效性检查的。

还有一些系统级的等效性检查工具，不过目前都是实验性质的，并没有被广泛使用。

- 模型检查 (Model Checking)。实现模型检查的工具和检查 ALF 里的库元件特性或 SPICE 模型的工具比较类似。模型是用 PSL 语言 (Property Specification Language, IEEE 1850) 写的，需要检查用 Verilog 写的模型是否和原模型一致。可以把这类工具理解成提供内容检查工具。通常，要实现检查所有的实验细节是不现实的。这类工具往往还可以处理简单的时序逻辑。

- 形式证明 (Formal Proving)。这类工具检查的是设计中逻辑状态的不连贯性。检查组合逻辑时，工具会运行的很轻松。如果检查的是时序逻辑，整个过程就变得复杂起来了。

有一些工具在提供了形式验证功能的同时，又提供了一部分仿真工具的功能。这样，如果遇到形式验证处理很困难的代码，可以用仿真的办法再对这部分代码进行验证。从而保证了验证能覆盖到所有的代码。

24.1.5 芯片设计的非逻辑因素

本书里，我们只学习了数字电路的设计。数字设计要求正确的功能和对应于正确时序的正确操作。然而，除了这两点之外，芯片里，门器件之间的相互影响和它们在芯片中的位置也会影响到芯片的正常工作。芯片里的门器件必须有良好的供电 (VDD 和 VSS)，同时不能影响其他的门器件。

- 电源分区。前面已经提到了，芯片可以按不同的区域工作在不同的电压下。这么做的主要原因是降低了功耗（尤其对于电池供电设备来说）或者是控制芯片的温度。此外，如果使用的是别人提供的 IP，IP 对供电等也可能有自己特殊的需求。

许多设备支持普通模式和睡眠模式。在睡眠模式里，时钟被停掉。对于没有工作的部分，它的电源可能会被断掉从而降低漏电功耗。对许多设计来说，芯片会自动保存进入睡眠模式前的状态，且当它从睡眠模式恢复成普通模式时，还能回到以前的状态。

芯片的核心通常被 I/O pad 包起来。对于单电源供电的芯片来说，芯片内有两个独立的电源和地。如果芯片是多电压芯片，则每一个电源域都需要各自独立的电源和地。这样，除了供电，还能起到隔离噪声的作用。

在任意一个电压区域中，在覆盖区域内部的输电网上提供电压值可能出现的高电平和低电平。每行上的基本单元通过过孔将电源引脚连在高电平和低电平供电上。输电网提供多种电流的源和汇，使得短时间的电涌被平均了，减少了在各个独立门上的变化。

- 噪声估计。在流片之前，芯片应该进行针对噪声的仿真或静态分析。门器件不应将耦合 (coupling) 的状态传递给另外一个门器件。耦合是容性的。当需要计算芯片内部的变化导致门器件的电压变化时，感应的耦合电容也应该被计算在内。电压变化还可以导致尖

端放电，造成额外的功耗并产生噪声。耦合噪声可能会改变逻辑电平。它引起的一个更常见的错误是改变了时序：耦合电压会影响到它附近正在改变的电压值，有可能会加速或减慢电压变化的速度。这些变化可能会导致发生建立和保持时间错误。

为了防止噪声，应该在设计中添加防护措施（在开关逻辑轨道上增加不用的电源和地轨道）或者增加邻近轨道之间的距离，从而减小耦合现象带来的影响。长距离的并行轨道应重新布线。也可以改变（与芯片垂直的方向上）轨道的厚度以降低串扰，因为厚的轨道串扰的可能性高于薄的轨道。然而，在很多制造过程中，金属和半导体厚度是预先定好的，不能在制造时改动。

在任何情况下，如果要制造可靠的芯片，则邻近逻辑的噪声，特别是轨道的噪声是必须被控制和消除的。

24.1.6 System Verilog

System Verilog 是被 IEEE 采纳的一种 Accellera 标准。最新的 Accellera 版本是 3.1a；这个版本可以从 Accellera 的网站上免费下载。但只有 IEEE 承认的版本才能被认为是正式版本。

这种语言规则是 Verilog-2001 的一个扩展集，它包含了很多 C++ 的特性，并且有一整套断言的规则。

提出 System Verilog 的一个重要目的是使得 C++ 和 Verilog-2001 的接口变得更容易。另外一个重要的目的是让工程师能够更直接的去表达一个复杂的、系统级的设计和断言描述。

下面列举了 System Verilog 的一些特性：

- 用户自定义类型（C++ 中的 `typedefs`）和类似 VHDL 的类型控制。
- 指针，引用，动态和队列类型。
- 仿真事件进度扩展。
- 类似于 Pascal 的嵌套 module 声明，类似于 VHDL 的 packages 声明，类似于 C++ 的 class 声明（只能继承一次）。
- 新的变量类型 logic，可以在顺序执行和并行执行的块中被赋值；新的 2 状态类型：bit。
- 用 iff 检查时序

```
always@(a iff Ena==1'b1) ...;
```

- 不再使用 `timescale，而使用 timeunit 和 timeprecision。
- 类似于 C 语言的 break，continue 和 return（不需要指定块名）。
- Final 块；always 块的新变种：always_latch，always_ff 和 always_comb。
- 新的 interface 类型，使得重用 I/O 变得更容易。
- 自己就包含了一个断言的语言子集。
- Covergroup 可以实现功能覆盖率测试。
- 新的可编程接口，DPI（Direct Programming Interface）。

作者觉得以下几点新特性是最重要的：

- Packages
- Break，continue 和 return

- Interface 类型
- 断言

24.2 课后练习（继续完成练习 23 及以后的练习）

24.2.1 补充学习

（可选）阅读 Thomas and Moorby (2002) 的 11.1 节。这一节里给出了翻转测试，这个测试用一个加法器结构来表示了电源的损耗。

你可以试着完成这部分的 Verilog 代码。

阅读 Palnitkar (2003) (可选)

阅读 15.1 节关于仿真、竞争和用硬件提高验证速度的内容。同时阅读 15.4 节中的小结。

阅读 15.3 节关于形式验证的内容。

