



陈俊杰

(美)

Meyyappan Ramanathan

等译

编著

Srikant Vijayaraghavan

SystemVerilog Assertions Assertions

SystemVerilog Assertions
A Practical Guide for



A Practical Guide for SystemVerilog Assertions

“当前 SoC 的复杂度的提高，以及尽快进入市场和首次流片成功的多重压力使基于断言的验证成为一种需求。本书指出了如何有效地使用断言的方法。”

—— Satish S. lyengar

ASIC 工程部门总监

Crimson Microsystems, Inc.

“无论对 SystemVerilog Assertions (SVA)的新手还是高级用户，本书都颇有裨益。首先简单易懂地介绍了基于断言的验证(ABV)的概念；然后从更开阔的视角讨论 SVA 提供的种种思想。许多真实的实例贯穿全书，这是非常有用的。”

—— Irwan Sie

IC 设计总监

ESS Technology, Inc.

“SystemVerilog Assertions 是一种崭新的语言。它能够在设计早期发现并隔离设计缺陷。本书使用多个实例展示了如何用 SVA 验证复杂的协议和内存设计。无论对设计工程师还是验证工程师，本书都是一本非常好的参考手册。”

—— Derrick Lin

工程部资深总监

Airgo Networks, Inc.

ISBN 7-302-13441-3



9 787302 134411 >

定价：39.80 元

 Springer

读者信箱: fwkbook@tup.tsinghua.edu.cn

信息网站: springeronline.com



SystemVerilog Assertions

应用指南

(美) Srikanth Vijayaraghavan 编著
Meyyappan Ramanathan
陈俊杰 等译

清华大学出版社

北 京

EISBN: 0-387-26049-8

A Practical Guide for SystemVerilog Assertions

Srikanth Vijayaraghavan, Meyyappan Ramanathan

Copyright©2005 by Springer Press Ltd.

Authorized translation from the English language edition published by Springer Ltd.

All Rights Reserved. For sale in the People's Republic of China only.

Chinese simplified language edition published by Tsinghua University Press.

本书中文简体字版由施普林格出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2006-4712

版权所有，翻印必究。举报电话：010-62782989 13501256678 13801310933

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

本书防伪标签采用特殊防伪技术，用户可通过在图案表面涂抹清水，图案消失，水干后图案复现，或将表面膜揭下，放在白纸上用彩笔涂抹，图案在白纸上再现的方法识别真伪。

图书在版编目(CIP)数据

SystemVerilog Assertions 应用指南/(美)维加亚拉哈文(Vijayaraghavan,S.), (美)拉门那斯(Ramanathan, M.)著；陈俊杰 等译。

—北京：清华大学出版社，2006.10

书名原文：A Practical Guide for SystemVerilog Assertions

ISBN 7-302-13441-3

I . S … II .①维…②拉…③陈… III. 电子电路—电路设计：计算机辅助 设计 IV.TN702

中国版本图书馆 CIP 数据核字(2006)第 081310 号

出版者：清华大学出版社 地址：北京清华大学学研大厦

http://www.tup.com.cn 邮编：100084

社总机：010-62770175 客户服务：010-62776969

组稿编辑：王军

文稿编辑：徐燕萍

封面设计：孔祥丰

版式设计：孔祥丰

印刷者：北京牛山世兴印刷厂

装订者：北京市密云县京文制本装订厂

发行者：新华书店总店北京发行所

开本：153×230 印张：20.5 字数：294 千字

版次：2006 年 10 月第 1 版 2006 年 10 月第 1 次印刷

书号：ISBN 7-302-13441-3/TP · 8442

印数：1 ~ 3000

定价：39.80 元(含光盘)

教师信息反馈表

为了更好地为您提供相关服务，使您的教学工作更加轻松高效，也使您的教学成果能够得以推广，请按下表填写您的基本信息、意见和要求。然后剪下寄到：北京清华大学出版社第五事业部(邮编 100084)；您也可以把意见反馈到 fwkbook@tup.tsinghua.edu.cn。邮购咨询电话：010-62770177/75 转 3505。您的支持将对我们的工作提供有益的帮助，谢谢合作！

一、基本信息：

姓名：_____ 所在院校：_____

职称/职务：_____ 系别专业：_____

联系电话：_____ E-mail：_____

通信地址：_____ 邮 编：_____

二、您所授课程的学术领域（或课程名）：

电子信息与电气工程 计算机软件及理论 计算机网络 操作系统

计算机科学理论 数据库 信息系统 信息安全

计算机系统结构 计算机语言与程序设计

计算机基础理论 图形学和 CAD

多媒体技术 计算机通信技术 人工智能

其他：_____

三、请选择您所需新书信息的学科，我们将根据您选择的内容定期给您发送新书信息。

电子信息 计算机语言与程序设计 计算机网络

计算机原理与硬件技术 计算机操作系统

图形图像、多媒体技术 计算机辅助设计 办公自动化 软件工程

软件理论与系统 数据库 信息安全

人工智能 信息系统与工程 计算机在其他领域中的应用

其他：_____

四、您希望我们就本书提供哪些服务？

五、近期您有何教材或专著出版计划？

六、您对清华大学出版社教材营销工作有何评价及建议？

我们对您的反馈表示真诚的谢意！

出 版 说 明

电子信息产业是一项新兴的高科技产业，有“朝阳产业”之称，有着巨大的潜力和广阔的发展前景。近年来，我国电子信息产业的飞速发展，大大推动了对电子信息类人才的需求，迫切需要我国的高等院校能够培养出大批符合企业要求的电子信息类人才。

教育与教材的关系始终是密不可分的，教材的合适与否会直接影响到培养人才的质量好坏。虽然目前我国高校中现行的电子信息类教材曾经对我国电子信息类人才的培养做出了非常重要的贡献，但是确实普遍存着一些问题，如“课程系统老化”、“内容落伍”、“惯性大，更新速度慢”、“针对性差”、“缺乏原创精品”等等，教学内容和课程体系的改革已经成为目前教学改革过程中的当务之急。

基于这种背景，我们决定在国内引进并推出一套“国外电子信息经典教材”，通过系统地研究和借鉴国外一流大学的相关教材，为我国高校的课程改革和国际化教学进程提供参考和推动作用。

为了组织该套教材的出版，我们在国内聘请了一批资深的专家和教授，共同成立了教材编审委员会。由编委会结合目前国内高校电子信息类专业的课程体系和教学内容，从 McGraw-Hill Education、Thomson Learning、John Wiley & Sons 和 Springer 等一批国际著名的教育出版集团，精选出一套“国外电子信息经典教材”。列选的每本教材都经过了国内相应领域的资深专家推荐和审读，对于一些基础类的专业课程，我们列选了多种不同体系、不同风格和不同层次的教材，以供不同要求和不同学时的同类课程使用。为了确保该套教材的质量，我们聘请了高校相应专业的资深教师和相应领域的专家担纲译者，加强了该套教材各个出版环节的编审力量和质量控制。另外，为了丰富国内的教学资源，

我们在引进教材的同时也积极引进了教材配套的教学资源。

该套教材的读者对象为电子信息与电气工程类专业的本科生，同时兼顾相关工程学科各专业的本科生或研究生。该套教材既可作为相应课程的教材或教学参考书，也适于相应技术领域的工程师和技术人员参考或自学。

尽管我们作了种种努力，但该套教材书目选择的恰当性，内容的合理性，都还有待于通过教学实践来检验。首先感谢选用该套教材的广大教生对我们的支持，同时期待广大读者积极为该套教材提出意见或建议。

清华大学出版社

序

在 20 世纪 80 年代中期, Gateway 设计自动化公司(Gateway Design Automation Inc.)发明了 Verilog, 那时集成电路的设计过程与现在有很大的不同。Verilog 的作用和功能从一开始就不断演变, 直至发展成为今天的 SystemVerilog。

ASIC(专用集成电路)的功能验证逐渐成为一项困难的事情, 至于到底有多难一直处在推测和争论中。2001 年, EE Times 引用了 Cisco 系统工程副总裁 Andreas Bechtolsheim 对验证的困难程度所作的一个较高的估计:

“设计验证所费的时间仍然占整个芯片开发时间的 80%。”

然而在 2004 年, EE Times 对参加设计自动化会议(DAC)的 662 名专业人士做了一个民意测验, 得出功能验证占整个集成设计过程的 22%这样一个结论。

22% 和 80% 的巨大差距显示了人们对验证和集成电路设计与开发的其他阶段的认识有多么模糊。许多验证工作是设计工程师自己完成的, 但仍然是验证过程的一部分, 而这部分的验证工作同样可以受益于辅助专业验证工程师的验证工具。

不管实际的比例占多少(假设能够精确测量出来), 功能验证依然是集成电路设计的一个重要组成部分。验证也是硅片一次成功的关键一步。即使当掩膜的成本超过了 100 万美元, 但相对于由于返工而耽搁几周导致的错失良机来说, 这些钱是微不足道的。任何工具, 只要能减少验证费用和提高尽早流片的可能性, 都应该积极采纳。

断言作为软件开发的一部分已经很多年了, 而基于断言的验证(Asserion-Based Verification, ABV)才刚刚流行起来。奇怪的是, 硬件描述过程与软件设计过程在某种程度上越来越相似。然而, 在硬件设计中我们想要断言和检验的属性, 与软件世界里的属性有着根本上的不同。

硬件和软件编程模式的差异是时序。硬件语言如: Verilog,

有表示时序的机制，而过程式编程语言(C、C++、Java 等)却没有。所以用不着奇怪在断言的软件方法中没有定义处理时序的方式。

SystemVerilog 是 Gateway 公司的 Verilog 的最新后继者，包括了 SystemVerilog 断言(SystemVerilog Assertions, SVA)——允许工程师把 ABV 应用到他们设计中的一系列工具。SVA 有很丰富的语法，可以在序列、属性和(完全的)断言中支持时间概念。

有了 SVA，设计和验证工程师可以把对硬件设计的期望行为进行编码，可以创建对总线协议的详细检验。这些(相对)简洁的描述能用于模拟、形式验证和作为设计的附加文档。

SVA 将会在集成电路的设计和验证的方法上有很重大的影响，这一点是很明了的。学习了 SVA 的语法及如何把它应用于您的设计中，您将受益匪浅。本书将会帮助您学习和应用 SVA。书中举了一些例子，包括 PCI(外设部件互连)总线协议，主要用来说 明如何编写 SVA 及其模拟结果。

本书中 SVA 语言的详细例子对理解基于时序的断言概念和语法很有帮助。这些例子使这本书更加名副其实。它是所有 SystemVerilog 设计和验证工程师的必备用书。

最后，我的女儿 Stevie，声称没有人会读一本书的序。如果读完了本书的序，请您发个简短的电子邮件告诉她。她的 e-mail: steviechayut@gmail.com。

谢谢！

Ira Chayut
验证架构师，Nvidia 公司

中文版序

超大规模集成电路随着系统芯片(SoC)时代的到来其功能日益复杂，规模也日益庞大，无论是芯片设计工程师还是芯片验证工程师都面临着前所未有的挑战。设计工程师和验证工程师需要借助更为有效的高级设计语言和验证技术来实现 SoC 的正确设计。

SystemVerilog 作为一种硬件描述和验证语言 HDVL(Hardware Description And Verification Language)，是随着技术发展和市场需求应运而生的。SystemVerilog 为 Verilog-2001 标准提供了一系列的扩展，这些扩展有效地提升了硬件设计、模拟验证和测试平台的整体效率，显著地降低了芯片设计的风险，有助于缩短产品的上市时间。SystemVerilog 于 2005 年 11 月成为电子设计的一种新标准语言(SystemVerilog, IEEE Std 1800-2005)。

在当今流行的验证技术中，基于断言的验证(Assertion-Based Verification, ABV)是一种很有价值的主流验证技术。作为 SystemVerilog 的重要组成部分，SystemVerilog Assertion 提供了丰富的断言指令，使得设计工程师和验证工程师可以快速地对复杂设计验证行为进行定义。

近年来，我国大陆地区的半导体行业发展迅速，从芯片的设计到制造、测试、封装，国内已基本形成了完整的 IC 产业链。但是，我们同时也应看到，与其他发达国家和地区相比，无论是自主创新、技术开发还是人才培养方面都存在着较大的差距。我们应该群策群力，积极支持通过多种方式引进国外先进的技术出版物和书籍。《SystemVerilog Assertions 应用指南》的翻译出版就是引进集成电路设计优秀技术书籍的一个范例。本书可作为电子通讯专业高年级本科生和研究生的教学用书，也可作为设计工程师和验证工程师的应用参考书。

国家 863 集成电路设计专项总体组组长
国际集成电路人才基地专家委员会主任

严晓浪 教授

2006 年 8 月

译者的话

由于研发项目的缘故，有机会比较早地接触 Open Vera Assertion(OVA)以及后来的 SystemVerilog Assertions(SVA)这项新技术。然而几个月以后，研发项目做完，对这项技术还只是“不求甚解”。

2005 年在公司总部 MontainView 参加年度 Offsite 会议，资深市场总监 Steve Smith 把这本由 Springer 出版社刚刚出版的新书介绍给我时，顿时有一种如获珍宝、相见恨晚的感觉！

本书的英文作者 Srikanth Vijayaraghavan 和 Meyyappan Ramanathan 是两位资深应用顾问。他们对于现代数字系统设计流程、先进的验证环境以及不同的 IC 设计特点有着深刻独到的见解。本书的特点是不仅系统地介绍了 SVA 这种硬件验证语言(HVL)的基本语法，而且针对不同类型的 IC 设计深入浅出地介绍了 SVA 的应用。本书分 8 章，其中第 0, 1, 2 章介绍了 ABV(基于断言的验证)方法学、SVA 的语法及用一个实例介绍了 SVA 的应用。第 3, 4, 5, 6, 7 章分别讨论了 SVA 在各种典型设计中的应用。这些典型设计模型包括了：有限状态机(FSM)，数据通道，存储控制器，基于 PCI 局部总线系统和测试平台(testbench)。无论是对刚刚接触断言的新手还是资深设计验证工程师，本书都是案前必备的一本参考书！

几个月艰苦的翻译工作是一种磨练，也是一种提升。国内可以参阅的关于 SVA 的资料非常有限。许多技术词汇的翻译在业内还没有达成共识，有些词汇几乎是第一次翻译。这对于翻译团队来说是一个巨大的挑战。我们常常对如何恰当地翻译一个词汇进行反复的争论和推敲。在翻译审校过程中，发现了几处英文原书的错误，经原作者同意，在中文版中得以纠正。

本书的原序，前言，第 0 章，第 2 章和第 3 章由陈俊杰和马青娇翻译。第 1 章由黄劲楠翻译。第 4 章和第 5 章由叶擎双翻译。第 6 章和第 7 章由杨天翔翻译。陈俊杰负责全书的整理和审校。

本书虽然经过仔细校对，但由于译者水平有限，难免存在错误和疏漏之处，希望读者批评指正。欢迎大家通过 SystemVerilogAssertion@googlegroups.com 和译者联系。

衷心地感谢下列业内资深人士以及国内外教授学者审阅本书并提出了许多宝贵的意见：王欣，Steve Ye，刘敬军，陈海慧，郑昊，唐叔平，谢原等。

衷心地感谢严晓浪教授百忙之中为本书写了中文序。

衷心地感谢公司同仁特别是 Steve Smith, Sadeghian, Chris K, Alessandro Fasan, 潘建岳, 冯成, 鲍志伟, 赵懿等的鼓励和支持。

衷心地感谢 Synopsys 公司和清华大学出版社的支持，本书的中文版才得以如期和读者见面。

最后，衷心感谢所有家人的支持。感谢女儿陈怡冰，你的诞生是一个最美丽的奇迹！

陈俊杰
2006 年 8 月

前　　言

2002 年中期，我们的经理给我们发了一个电子邮件，问道：“谁愿意去支持 OVA？”我们从脑子中迸出的第一个想法就是“究竟什么是 OVA？”和其他几个工程师交谈后，我们知道它是 OPEN VERA 语言的一个子集。OVA 是指“OPEN VERA 断言(Open VERA Assertions, 简写 OVA)”，它是一种描述性的语言，能描述时序上的条件。就如同过去一样，为了满足对技术的渴求，我们同意做 OVA 的支持。在两个月内，我们学习了这种语言，并开始培训客户，在六个月内培训了 200 个左右的客户。客户洪水般涌进教室，给我们留下深刻的印象。我们确信这是验证领域下一件最好的事情。当客户们匆忙接受完培训，他们并没有开发任何 OVA 的代码。这是因为验证技巧和这种语言都是新的。一些工具刚开始支持这些语言结构。没有多少 IP(Intellectual Property)可以使用。很自然，客户并没有我们想像的那么满意。

同时，Synopsys 公司把 OPEN VERA 语言捐献给 Accellera 委员会，使其成为 SystemVerilog 语言的一部分，其他几个公司为 SystemVerilog 语言的形成作了一些贡献。在 DAC 2004，Accellera 委员会把 SystemVerilog 3.1 定为一个标准。断言语言被纳入 SystemVerilog 语言并成为了标准的一部分。这就是通常所说的“SystemVerilog Assertion(SVA)”。我们继续培训客户基于断言的验证，不过现在仅仅教 SVA。我们能清楚地看到客户更习惯于使用预开发的断言库，而不乐意编写定制的断言代码。是什么阻碍了他们？是工具吗？不，工具是现成的。是语言吗？或许，但它如今已是一个标准，所以不应该是它。

经过一番深入的讨论，我们认识到，缺乏例子来演示 SVA 的结构可能是阻碍客户使用这项新技术的原因。比较典型的是缺乏专家指导导致了如此低的采纳率。这时我想到出版一本关于 SVA 的“烹饪书”可能有用——即一本充满例子的书，这本书可以作为指导书，用来教授这种语言。这个项目就是这样启动的。我们

努力把过去两年中在教授这门科目时所学的东西写出来。但是在这个领域还有很多东西需要去学，这本书只是把我们所学到的跟大家分享。

如何阅读这本书

这本书的写作方式可以使工程师快速掌握 SystemVerilog 断言。

第 0、1 和 2 章，可以使您充分了解基础语法和一些通用的模拟技巧。阅读完这三章，读者应该能在他们的设计 / 验证环境中写断言。

第 3、4、5 和 6 章是不同类型的设计的“烹饪书”。读者如果在他们自己的环境里遇到类似的设计可以参考这些章节，以这些章节作为起点开始写断言。这些章节也可以作为指导。

如果您是基于断言验证的新手，则需要阅读完第 0 章~第 2 章，才能开始其他章节。如果您熟悉 SVA 语言，就可以根据需要参考这些章节。

第 0 章——这是关于基于断言的验证(ABV)方法论的白皮书。这一章介绍了 ABV 的方法学和功能覆盖的重要性。

第 1 章——用简单的例子讨论了 SVA 的语法和详细分析了在动态模拟中执行 SVA 结构的过程。包括了模拟波形和事件表以供读者参考。要了解每个 SVA 结构的细节，用户可以参考 SystemVerilog 3.1 a 语言参考手册(LRM)的第 17 章。

第 2 章——用一个实例系统说明 SVA 模拟的方法。主题囊括了协议解析、模拟控制和功能覆盖。

第 3 章——用两个不同的有限状态机(FSM)模型作为例子，举例说明如何用 SVA 验证 FSM。

第 4 章——举例说明用 SVA 验证一个数据通道。用 JPEG 设计的一部分来演示如何用 SVA 验证控制信号和数据。

第 5 章——举例用 SVA 验证一个存储控制器。这个控制器支持不同类型的存储如：SDRAM、SRAM、FLASH 等。

第 6 章——举例用 SVA 验证一个基于 PCI 局部总线的系统。使用了一个 PCI 系统配置的例子，用 SVA 验证不同的 PCI 协议。

第 7 章——用一个测试平台(testbench)的例子验证断言，也讨论了在验证断言的精度背后的理论。

随书附一张光盘。本书中的所有例子都可以用 VCS 2005.06 发行版运行，也包括运行这些例子的脚本范例。VCS 是 Synopsys 公司的注册商标。

致谢

下面的人由于他们对完成本书所做出的巨大贡献，在这里作者对他们表示真诚的谢意：

Anupama Srinivasa, DSP 解决方案架构师, AccelChip 公司;

Jim Kjellsen, Staff 应用顾问, Synopsys 公司;

Juliet Runhaar, 资深应用顾问, Synopsys 公司;

我们同样感谢下面的人，他们参与审阅本书并提供了很多建设性的建议：

Ira Chayut, Bohran Roohipour, Irwan Sie, Ravindra Viswanath, Parag Bhatt, Derrick Lin, Anders Berglund, Steve Smith, Martin Michael, Jayne Scheckla, Rakesh Cheerla, Satish Iyengar

有用的链接

www.systemVerilogforall.com——我们维护的网页，提供关于 SystemVerilog 语言的技巧、例子和讨论。

www.accellera.org——Accellera 委员会的官方网站。可以从这里下载 SystemVerilog LRM。这里还有一些有用的论文和有关最新标准的介绍。

目 录

第 0 章 基于断言的验证	1
第 1 章 SVA 介绍	5
1.1 什么是断言	5
1.2 为什么使用 SystemVerilog 断言(SVA)	6
1.3 SystemVerilog 的调度	8
1.4 SVA 术语	9
1.4.1 并发断言	9
1.4.2 即时断言	10
1.5 建立 SVA 块	11
1.6 一个简单的序列	12
1.7 边沿定义的序列	13
1.8 逻辑关系的序列	15
1.9 序列表达式	15
1.10 时序关系的序列	16
1.11 SVA 中的时钟定义	18
1.12 禁止属性	19
1.13 一个简单的执行块	21
1.14 蕴含操作符	21
1.14.1 交叠蕴含	22
1.14.2 非交叠蕴含	23
1.14.3 后续算子带固定延迟的蕴含	24
1.14.4 使用序列作为先行算子的蕴含	25
1.15 SVA 检验器的时序窗口	27
1.15.1 重叠的时序窗口	29
1.15.2 无限的时序窗口	30

1.16	“ended” 结构	32
1.17	使用参数的 SVA 检验器	35
1.18	使用选择运算符的 SVA 检验器	36
1.19	使用 true 表达式的 SVA 检验器	38
1.20	“\$past” 构造	39
1.21	重复运算符	42
1.21.1	连续重复运算符[*]	43
1.21.2	用于序列的连续重复运算符[*]	44
1.21.3	用于带延迟窗口的序列的连续重复运算符[*]	46
1.21.4	连续运算符[*]和可能性运算符	47
1.21.5	跟随重复运算符[->]	48
1.21.6	非连续重复运算符[=]	50
1.22	“and” 构造	51
1.23	“intersect” 构造	54
1.24	“or” 构造	56
1.25	“first_match” 构造	58
1.26	“throughout” 构造	60
1.27	“within” 构造	61
1.28	内建的系统函数	63
1.29	“disable iff” 构造	65
1.30	使用 “intersect” 控制序列的长度	66
1.31	在属性中使用形参	68
1.32	嵌套的蕴含	70
1.33	在蕴含中使用 if/else	71
1.34	SVA 中的多时钟定义	73
1.35	“matched” 构造	75
1.36	“expect” 构造	76
1.37	使用局部变量的 SVA	77
1.38	在序列匹配时调用子程序	79
1.39	将 SVA 与设计连接	81
1.40	SVA 与功能覆盖	83

第 2 章 SVA 模拟方法论	85
2.1 一个被验证的实例系统	85
2.1.1 主控设备	86
2.1.2 中间设备	88
2.1.3 目标设备	90
2.2 块级验证	91
2.2.1 SVA 在设计块中的应用	92
2.2.2 仲裁器的验证	92
2.2.3 模拟中针对仲裁器的 SVA 检验	94
2.2.4 主控设备的验证	96
2.2.5 模拟中针对主控设备的 SVA 检验	98
2.2.6 胶合(Glue)的验证	100
2.2.7 模拟中针对胶合逻辑(glue logic)的 SVA 检验	102
2.2.8 目标设备的验证	104
2.2.9 模拟中针对目标设备的 SVA 检验	106
2.3 系统级验证	108
2.4 功能覆盖	114
2.4.1 实例系统的覆盖率计划	115
2.4.2 功能覆盖小结	124
2.5 用于创建事务日志的 SVA	124
2.6 用于 FPGA 原型测试的 SVA	127
2.7 SVA 模拟方法的小结	131
第 3 章 SVA 在有限状态机中的应用	133
3.1 设计例子—— FSM1	134
3.1.1 FSM1 的功能描述	134
3.1.2 FSM1 的 SVA 检验器	139
3.2 设计实例—— FSM2	143
3.2.1 FSM2 的功能描述	144
3.2.2 FSM2 的 SVA 检验器	148
3.2.3 有时序窗口协议的 FSM2	155
3.3 在 FSM 中应用 SVA 的小结	159

第 4 章 SVA 用于数据集约型(DATA iNTENSIVE)的设计	161
4.1 简单乘法器的检验	161
4.2 设计实例——算术单元	163
4.2.1 WHT 算术	163
4.2.2 WHT 硬件的实现	164
4.2.3 WHT 模块的 SVA 检验器	165
4.3 设计实例——JPEG 的数据通路设计	168
4.3.1 三模块的深入探讨	169
4.3.2 用于 JPEG 设计的 SVA 检验器	172
4.3.3 JPEG 模型的数据检验	176
4.4 数据集约型设计的小结	182
第 5 章 SVA 储存器	183
5.1 存储控制系统实例	183
5.1.1 CPU-AHB 接口操作	183
5.1.2 存储控制器的操作	186
5.2 SDRAM 的验证	189
5.3 SRAM/FLASH 的验证	208
5.4 DDR-SDRAM 的验证	215
5.5 存储器 SVA 的小结	217
第 6 章 SVA 协议接口	219
6.1 PCI 简介	220
6.1.1 一个 PCI 读出事务的实例	222
6.1.2 PCI 写入事务实例	223
6.2 PCI 系统实例	224
6.3 情形 1——主控 DUT 设备	225
6.4 情形 2——目标 DUT 设备	243
6.5 情形 3——系统级断言	259
6.6 用于标准协议的 SVA 小结	263
第 7 章 对检验器的检验	265
7.1 断言验证	266

7.2 双信号 SVA Assertion Test	268
7.2.1 双信号的逻辑关系	268
7.2.2 电平敏感逻辑关系激励的产生	269
7.2.3 边沿敏感逻辑关系激励的产生	272
7.2.4 双信号的时序关系	275
7.2.5 时序关系激励的产生	276
7.2.6 双信号的重复关系	286
7.2.7 双信号 ATB 环境	290
7.3 一个 PCI 检验器的 ATB 实例	301
7.4 检验器检验小结	305



基于断言的验证

——引入断言的原因

数字电路的规模和复杂度的不断增长，使得功能验证成为一项巨大的挑战。在过去 10 年中，验证领域产生了几项新的技术，其中一些技术已占有一席之地，成为验证过程中的必需步骤。

图 0-1 显示的是一个验证环境，为目前绝大多数的验证团队所采用。这里有两项重要的新技术，几乎所有的验证工程师都用到它们。

- (1) 约束随机测试平台(Constrained random Testbench)
- (2) 代码覆盖(Code coverage)工具

验证的目标是彻底地验证被测设计(DUT)，确保其中没有功能缺陷。在这个过程中，应该有一个方法来衡量验证的完整性。而代码覆盖工具提供了验证完整性的基本衡量标准。在代码覆盖中收集的数据对设计的功能没有概念，但它提供按行执行代码的信息。在模拟中确保 DUT 的每行至少执行一次，就会得到某个可信级，利用代码覆盖工具能实现这一点。最后，完成验证过程必须有时间性。一个众所周知的事实是任何验证环境的瓶颈是效率。

传统上，设计都是使用能验证设计中特定功能的激励来测试的。设计的复杂性迫使验证工程师使用随机测试平台生成更多真实的验证情景(scenario)。高级的验证语言，例如 OPEN VERA，在创建复杂测试平台时得到了广泛应用。

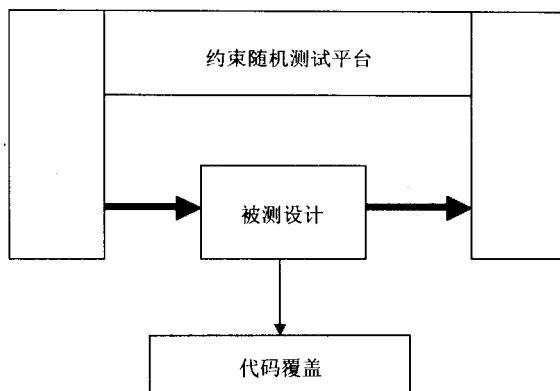


图 0-1 在基于断言的验证之前

通常，测试平台承担三个不同的任务：

- (1) 产生激励
- (2) 自检机制
- (3) 衡量功能覆盖

测试平台的首要目标是产生高质量的激励。高级语言如 OPEN VERA 有内建的机制就可以轻而易举地生成复杂的激励模式。这些语言支持面向对象编程，这有助于改善激励的产生及测试平台模型的重用。

一个测试平台还应提供优秀的自检机制。在后期处理模式 (post-processing mode)下对设计进行调试不能解决所有问题。像波形调试这样的机制倾向于发现人为的错误，而且在当今设计如此复杂的情况下不是很可行。应该有种方法使每个测试能自动和动态地检验期望的结果。这样调试过程会变得简单，同时回归测试 (regression test)更为有效。自检过程通常着眼于以下两个特定的方面：

- (1) 协议检验
- (2) 数据检验

协议检验的目标是控制信号。控制信号的正确性是验证任何设计的核心。数据检验的目的是检验正在处理的数据的完整性。例如，在一个网络的设计中传输的数据包是否损坏？数据检验通

常要求某些级别的格式和消息，在测试平台环境内会对它们进行有效处理。

功能覆盖用于衡量验证完整性，它的衡量标准要包含两项指定的信息：

- (1) 协议覆盖(Protocol Coverage)
- (2) 测试计划覆盖(Test Plan Coverage)

协议覆盖在所有合法和不合法的情况下衡量一个设计。换句话说，它是用来衡量一个设计的功能说明书(functional specification)中确定的所有功能是否都测试过。另一方面，测试计划覆盖用来衡量测试平台的穷尽性。例如，测试平台有没有生成所有可能大小的包？CPU 是否可以对任何可能的存储地址空间进行读写？协议覆盖直接从设计信号来衡量，而测试计划覆盖用测试平台环境中内嵌的方法很容易来衡量。

SystemVerilog 断言改变了验证环境，使得其中不同实体的优势都达到了最大值。图 0-2 是包含了基于断言的验证(ABV)的验证环境。

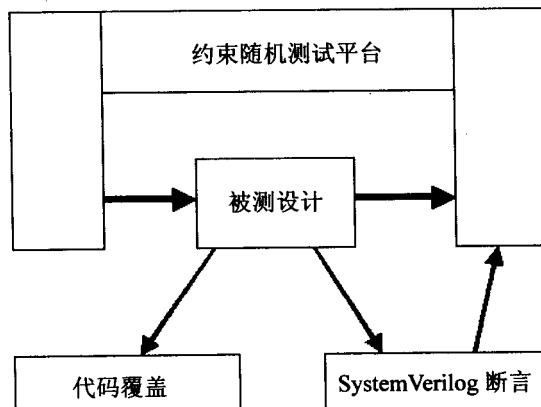


图 0-2 SystemVerilog 断言之后

SVA 着重处理在测试平台中被分散在不同部分中讨论的两大类：

- (1) 协议检验
- (2) 协议覆盖

这两类更加贴近设计信号，用 SVA 比在测试平台中能更有效

地处理它们。通过把断言直接与设计相连，模拟环境的性能和生产率都有了极大的提高。表 0-1 总结了基于 SVA 的验证环境的重新排列。

虽然 SVA 直接和设计信号交互，但是它仍然可以和测试平台很有效地共享信息。通过在模拟中动态共享信息，可以建立高效的交互测试平台环境。模拟中结合代码覆盖和功能覆盖的信息可以更有效衡量验证过程是否完全。

表 0-1 新的验证环境

	测 试 平 台	SVA
SVA 前	激励产生	
	协议检验	
	数据检验	N/A
	协议覆盖	
	测试计划覆盖	
SVA 后	激励产生	协议检验
	数据检验	协议覆盖
	测试计划覆盖	

本书将要介绍 SVA 语言，用精心设计的一些例子来说明它的使用模式和它的益处。书中将会讲述如何通过写一些高质量的断言尽早发现设计的缺陷，讨论如何用真实的设计实例和编写断言的过程来验证一个设计，以及如何衡量一个真实设计的功能覆盖和如何动态地使用功能覆盖的信息生成更加复杂的测试平台。此外，在相关的地方还将会讨论编码风格和模拟方法论的实践。



SVA 介绍

——学习 SVA 语法

1.1 什么是断言

断言是设计的属性的描述。

- 如果一个在模拟中被检查的属性(property)不像我们期望的那样表现，那么这个断言失败。
- 如果一个被禁止在设计中出现的属性在模拟过程中发生，那么这个断言失败。

一系列的属性可以从设计的功能描述中推知，并且被转换成断言。这些断言能在功能的模拟中不断地被监视。使用形式验证技术，相同的断言能被重用来验证设计。断言，又被称为监视器或者检验器，已经被用作一种调试技术的方式，在设计验证流程中使用了很长时间。传统上，它们由过程语言，比如 Verilog，来实现。它们也能用 PLI 和 C/C++的程序来实现。下面的代码显示了相互断言条件检查的 Verilog 实现，其中信号 a 和信号 b 不能同时为高电平。如果这种情况发生，则显示这是一个错误信息。

```
'ifdef ma
if(a & b)
$display("Error: Mutually asserted check")
```

```
failed.\n");
`endif
```

这种监视器仅作为模拟的一部分而存在，因此只有当需要时才被纳入设计环境中。这可以通过允许 Verilog 代码条件编译的指令“`ifdef”来实现。

1.2 为什么使用 SystemVerilog 断言(SVA)

虽然 Verilog 可以很容易地用来实现一些检查，它仍有一些不足之处：

- (1) Verilog 是一种过程语言，因此并不能很好地控制时序。
- (2) Verilog 是一种冗长的语言，随着断言的数量增加，维护代码将变得很困难。
- (3) 语言的过程性这一本质使得测试同一时间段内发生的并行事件相当困难。在一些情况下，一个 Verilog 的检验器甚至可能无法捕捉到所有被触发的事件。
- (4) Verilog 语言没有提供内嵌的机制来提供功能覆盖的数据。
用户必须自己实现这部分代码。

SVA 是一种描述性语言，可以完美地描述时序相关的状况。语言的描述性本质提供了对时间卓越的控制。语言本身非常精确且易于维护。SVA 也提供了若干个内嵌函数来测试特定的设计情况，并且提供了一些构造来自动收集功能覆盖数据。

例子 1.1 显示了分别用 Verilog 和 SVA 实现的检验器。这个检验器验证当信号 a 在当前时钟周期为高电平时，下面 1~3 个时钟周期内，信号 b 应该变为高电平。

例子 1.1 分别用 Verilog 和 SVA 实现的断言实例

```
// Sample Verilog checker

always @(posedge a)
begin
```

```

repeat (1) @ (posedge clk);
  fork: a_to_b

    begin
      @ (posedge b)
      $display
        ("SUCCESS: b arrived in time\n", $time);
      disable a_to_b;
    end

    begin
      repeat (3) @ (posedge clk);
      $display
        ("ERROR: b did not arrive in time\n", $time);
      disable a_to_b;
    end

  join
end

// SVA Checker

a_to_b_chk:
assert property
@ (posedge clk) $rose(a) |-> ##[1: 3] $rose(b));

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

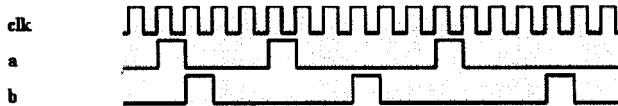


图 1-1 断言实例的波形

例子 1.1 很清楚地显示了 SVA 的优势。本章将讨论 SVA 语法。检验器代表着一种非常简单的协议。使用 SVA 实现只需要一行代码，而相同的协议描述用 Verilog 则需要好几行。此外，失败和成功的条件必须在 Verilog 里额外地被定义，而 SVA 中断言失败会

自动显示错误信息。例子模拟的结果如下：

```
SUCCESS: b arrived in time 127
vtosva.a_to_b_chk:
started at 125s succeeded at 175s

SUCCESS: b arrived in time 427
vtosva.a_to_b_chk:
started at 325s succeeded at 475s

ERROR: b did not arrive in time 775
vtosva.a_to_b_chk:
started at 625s failed at 775s
Offending '$rose(b)'
```

1.3 SystemVerilog 的调度

SystemVerilog 语言被定义成一种基于事件的执行模式。在每个时隙(time slot)，许多事件按照安排的顺序发生。这个事件的列表依照标准定义的算法执行。依照这个算法，模拟器可以防止任何在设计和测试平台互动中的不一致。断言的评估和执行包括以下三个阶段：

预备(Preponed) 在这个阶段，采样断言变量，而且信号(net)或变量(variable)的状态不能改变。这样确保在时隙开始的时候采样到最稳定的值。

观察(Observed) 在这个阶段，对所有的属性表达式求值。

响应(Reactive) 在这个阶段，调度评估属性成功或失败的代码。

图 1-2 显示了一个简化了的 SystemVerilog 事件进程安排流程表。要彻底地理解 SystemVerilog 的进程安排算法，请参考 <<SystemVerilog 3.1a LRM>>[1]。

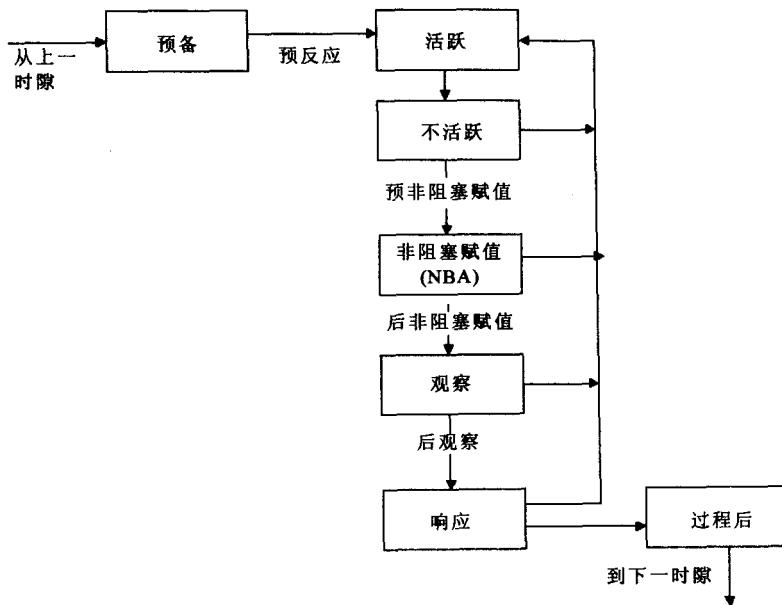


图 1-2 SV 事件调度流程简图

1.4 SVA 术语

SystemVerilog 语言中定义了两种断言：并发断言和即时断言。

1.4.1 并发断言

- 基于时钟周期。
 - 在时钟边缘根据调用的变量的采样值计算测试表达式。
 - 变量的采样在预备阶段完成，而表达式的计算在调度器的观察阶段完成。
 - 可以被放到过程块(procedural block)、模块(module)、接口(interface)，或者一个程序(program)的定义中。
 - 可以在静态(形式的)验证和动态验证(模拟)工具中使用。
- 一个并发断言的例子如下：

```
a_cc: assert property (@(posedge clk) ! (a && b));
```

图 1-3 显示了并发断言 a_cc 的结果。所有的成功显示成向上的箭头，所有的失败显示成向下的箭头。这个例子的核心内容是属性在每一个时钟的上升沿都被检验，不论信号 a 和信号 b 是否有值的变化。

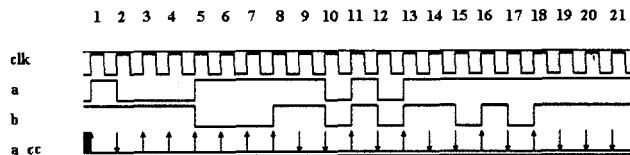


图 1-3 并发断言例子的波形

1.4.2 即时断言

- 基于模拟事件的语义。
- 测试表达式的求值就像在过程块中的其他 Verilog 的表达式一样。它们本质不是时序相关的，而且立即被求值。
- 必须放在过程块的定义中。
- 只能用于动态模拟。

一个即时断言的例子如下：

```
always_comb
begin
    a_ia: assert (a && b);
end
```

即时断言 a_ia 被写成一个过程块的一部分，它遵循和信号 a、b 相同的事件调度。当信号 a 或者信号 b 发生变化时，always 块被执行。区别即时断言和并发断言的关键词是“**property**”。图 1-4 显示了即时断言 a_ia 的结果：

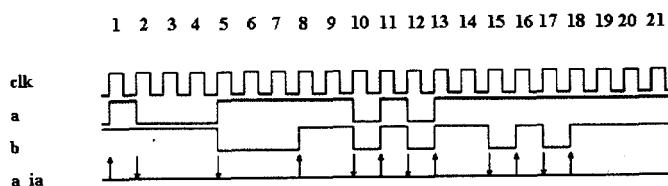


图 1-4 即时断言例子的波形

1.5 建立 SVA 块

在任何设计模型中，功能总是由多个逻辑事件的组合来表示的。这些事件可以是简单的同一个时钟边缘被求值的布尔表达式，或者是经过几个时钟周期的求值的事件。SVA 用关键词“sequence”来表示这些事件。序列(sequence)的基本语法是：

```
sequence name_of_sequence;  
    <test expression>;  
endsequence
```

许多序列可以逻辑或者有序地组合起来生成更复杂的序列。SVA 提供了一个关键词“property”来表示这些复杂的有序行为。属性(property)的基本语法是：

```
property name_of_property;  
    <test expression>; or  
    <complex sequence expressions>;  
endproperty
```

属性是在模拟过程中被验证的单元。它必须在模拟过程中被断言来发挥作用。SVA 提供了关键词“assert”来检查属性。断言(assert)的基本语法是：

```
assertion_name: assert property (property_name);
```

建立 SVA 检验器的步骤如图 1-5 所示：

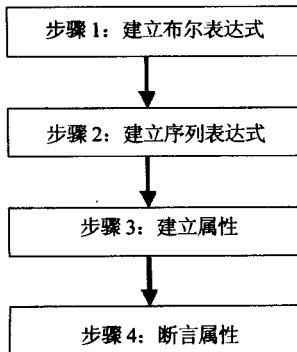


图 1-5 SVA 建立块图示

1.6 一个简单的序列

序列 s1 检查信号“a”在每个时钟上升沿都为高电平。如果信号“a”在任何一个时钟上升沿不为高电平，断言将失败。注意，这相当于“ $a == 1'b1$ ”。

```
sequence s1;
  @(posedge clk) a;
endsequence
```

图 1-6 显示了信号“a”和序列在模拟中对这个信号响应的波形。信号“a”在第七个时钟上升沿变为 0。这一变化在第八个时钟周期被采样到。因为并行断言使用进程安排中预备(“prepend”)阶段采样到的值，在第七个时钟周期，序列 s1 采样到的信号“a”的最稳定的值是 1。因此序列成功。在第八个时钟周期，信号“a”被采样的值为 0，因此序列失败。一个向上的箭头表示一次成功，一个向下的箭头表示一次失败。表 1-1 总结了信号“a”每个时钟周期的采样值，直到第十五个时钟周期。

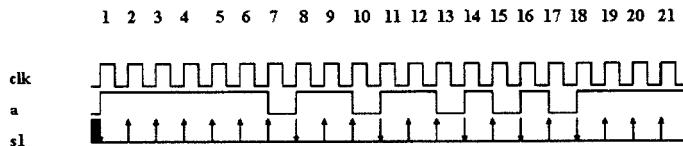


图 1-6 简单序列 s1 的波形

表 1-1 序列 s1 的真值表

时 钟 数	信号“a”的采样值
1	0
2	1
3	1
4	1
5	1
6	1
7	1

(续表)

时 钟 数	信号“a”的采样值
8	0
9	1
10	1
11	0
12	1
13	1
14	0
15	1

1.7 边沿定义的序列

序列 s1 使用的是信号的逻辑值。SVA 也内嵌了边缘表达式，以便用户监视信号值从一个时钟周期到另一时钟周期的跳变。这使得用户能检查边沿敏感的信号。三种这样有用的内嵌函数如下：

\$rose(boolean expression or signal_name)

- 当信号/表达式的最低位变成 1 时返回真。

\$fell(boolean expression or signal_name)

- 当信号/表达式的最低位变成 0 时返回真。

\$stable(boolean expression or signal_name)

- 当表达式不发生变化时返回真。

序列 s2 检查信号“a”在每一个时钟上升沿都跳变成 1。如果跳变没有发生，断言失败。

```
sequence s2;
  @ (posedge clk) $rose (a);
endsequence
```

图 1-7 显示序列 s2 响应信号“a”跳变的情况。标记 1 显示了序列 s2 的第一个成功。在时钟周期 1，信号“a”的值从 0 变到 1。

在这个时钟周期，信号“a”在序列中的采样值是 0。在时钟周期 1 之前，信号“a”没有被赋值，因此值被认定为“x”。值从 x 到 0 的转化不是上升沿，因此序列失败。在时钟周期 2，信号“a”在序列中的采样值是 1。值从 0 到 1 的转化是上升沿，因此序列 2 在时钟周期 2 成功。在时钟周期 9 的标记 2 显示了另一个成功。表 1-2 总结了信号“a”前 9 个时钟周期的转化以及序列如何采样和更新值。

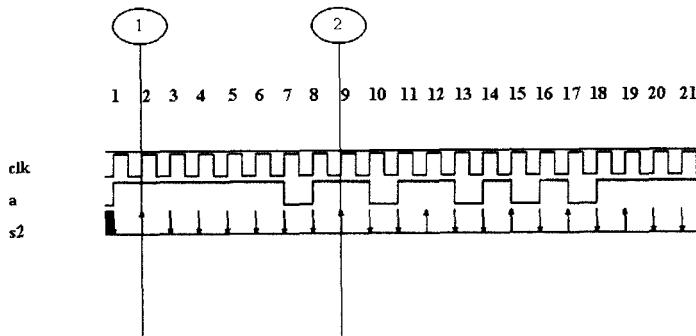


图 1-7 边沿定义的简单序列的波形

表 1-2 序列 s2 的真值表

时钟数	信号“a”从上一个 时钟周期得来的采	信号“a”在这个 时钟周期的采样	序列 s2 的状态
	样值	值	
1	X	0	失败
2	0	1	成功
3	1	1	失败
4	1	1	失败
5	1	1	失败
6	1	1	失败
7	1	1	失败
8	1	0	失败
9	0	1	成功

1.8 逻辑关系的序列

序列 s3 检查每一个时钟上升沿，信号“a”或信号“b”是高电平。如果两个信号都是低电平，断言失败。

```
sequence s3;
  @(posedge clk) a || b;
endsequence
```

图 1-8 显示序列 S3 如何根据信号“a”和“b”做出反应。标记 1 显示了时钟周期 12，信号“a”和“b”的采样值都是 0，因此序列失败。同理，在标记 2 所在的时钟周期 17，序列也失败。在所有其他时钟周期，信号“a”和信号“b”至少有一个其值为 1，因此在这些时钟周期，序列都成功。

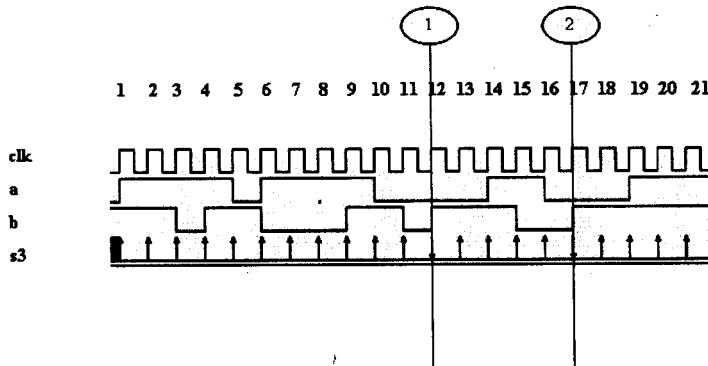


图 1-8 序列 s3 的波形

1.9 序列表达式

通过在序列定义中定义形参，相同的序列能被重用到设计中具有相似行为的信号上。例如，我们可以定义下面这个序列

```
sequence s3_lib (a, b);
  a || b;
endsequence
```

通用的序列 s3_lib 能重用在任何两个信号上。例如，我们有两个信号“req1”和“req2”，它们中至少一个信号应该在时钟周期的上升沿为 1，我们可以使用下列的序列。

```
sequence s3_lib_inst1;
    s3_lib (req1, req2);
endsequence
```

一些在设计中常见的通常的属性可以被开发成一个库以便于重用。比如，one-hot 状态机检查，等效性检查等都适合放在这样的检验器库中。

1.10 时序关系的序列

简单的布尔表达式在每个时钟边缘都会被检查。换句话说，它们只是简单的组合逻辑检查。很多时候，我们关心的是检查需要几个时钟周期才能完成的事件。也就是所谓的“时序检查”。在 SVA 中，时钟周期延迟用“##”来表示。例如，##3 表示 3 个时钟周期。

序列 s4 检查信号“a”在一个给定的时钟上升沿为高电平，如果信号“a”不是高电平，序列失败。如果信号“a”在任何一个给定的时钟上升沿为高电平，信号“b”应该在两个时钟周期后为高电平。如果信号“b”在两个时钟周期后不为 1，断言失败。注意，序列以信号“a”在时钟上升沿为高电平开始。

```
sequence s4;
    @(posedge clk) a ##2 b;
endsequence
```

图 1-9 显示了序列 s4 在模拟过程中的响应。表 1-3 总结了信号“a”和信号“b”在每个时钟周期的采样值。

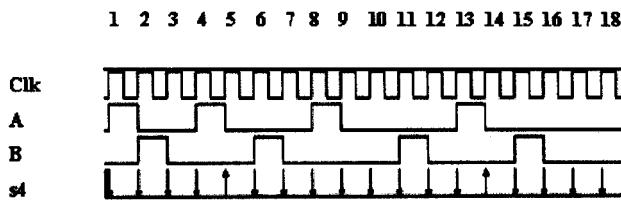


图 1-9 序列 s4 的波形

与前面小节中的例子不同的是，序列 s4 的开始时间和结束时间不同。如果信号“a”在任何时钟周期不为高电平，序列在同一个时钟周期开始并失败。如果信号“a”是高电平，序列开始。在两个时钟周期后，如果信号“b”是高电平，序列成功(第 5 和第 14 时钟周期)。另一方面，如果在两个时钟周期后，信号“b”不是高电平，序列失败。应注意的是，在图中，成功的序列总是标注在序列开始的位置。

表 1-3 序列 s4 的真值表

时钟 数	“a”的 采样值	“b”的 采样值	S4 有效	S4 状态
			开始	
1	0	0	否	失败
2	1	0	是	失败 (开始于 2, 结束于 4)
3	0	1	否	失败
4	0	0	否	失败
5	1	0	是	成功(开始于 5, 结束于 7)
6	0	0	否	失败
7	0	1	否	失败
8	0	0	否	失败
9	1	0	是	失败 (开始于 9, 结束于 11)
10	0	0	否	失败
11	0	0	否	失败
12	0	1	否	失败
13	0	0	否	失败
14	1	0	是	成功(开始于 14, 结束于 16)
15	0	0	否	失败
16	0	1	否	失败
17	0	0	否	失败

1.11 SVA 中的时钟定义

一个序列或者属性在模拟过程中本身并不能起什么作用。它们必须像下面的例子那样被断言才能发挥作用。

```
sequence s5;
  @(posedge clk) a ##2 b;
endsequence

property p5;
  s5;
endproperty

a5 : assert property(p5);
```

注意，序列 s5 中指定了时钟。这是一种把检查和时钟关联起来的方法，但是还有其他的方法。在序列、属性，甚至一个断言的语句中都可以定义时钟。下面的代码显示了在属性 p5a 的定义中指定时钟。

```
sequence s5a;
  a ##2 b;
endsequence

property p5a;
  @(posedge clk) s5a;
endproperty

a5a : assert property(p5a);
```

通常情况下，在属性(property)的定义中指定时钟，并保持序列(sequence)独立于时钟是一种好的编码风格。这可以提高基本序列定义的可重用性。

断言一个序列并不一定需要定义一个独立的属性。因为断言语句调用属性，在断言的语句中可以直接调用被检查的表达式，如下面的断言 a5b 所示。

```

sequence s5b;
a ##2 b;
endsequence

a5b : assert property(@(posedge clk) s5b);

```

当我们在断言的陈述中要调用已经定义了时钟的序列，就不能再次在断言语句中定义时钟。下面的断言 a5c 就显示了这种错误的编程风格。

```

a5c : assert property(@(posedge clk) p5a); // Not
allowed

```

1.12 禁止属性

在之前的所有例子中，属性检查的都是正确的条件。属性也可以被禁止发生。换句话说，我们期望属性永远为假。当属性为真时，断言失败。

序列 s6 检查当信号“a”在给定的时钟上升沿为高电平，那么两个时钟周期以后，信号“b”不允许是高电平。关键词“not”用来表示属性应该永远不为真。

```

sequence s6;
@(posedge clk) a ##2 b;
endsequence

property p6;
not s6;
endproperty

a6 : assert property(p6);

```

图 1-10 显示了断言 a6 如何在模拟过程中响应。我们注意到检验器在标记 1 和 2 显示的两个位置(时钟 5 和 14)失败。在这两个时钟周期，发生了被禁止的序列，断言因此失败。

另一方面，在信号“a”有效的两个位置(时钟 2 和时钟 9)检验器成功。因为从这两个时钟周期开始检查，两个时钟周期以后

信号“b”不为高，因此检验器成功。在其他时钟周期中，信号“a”都不为高，因此检验器都自动成功。表 1-4 总结了信号“a”和信号“b”在每个时钟周期的采样值。

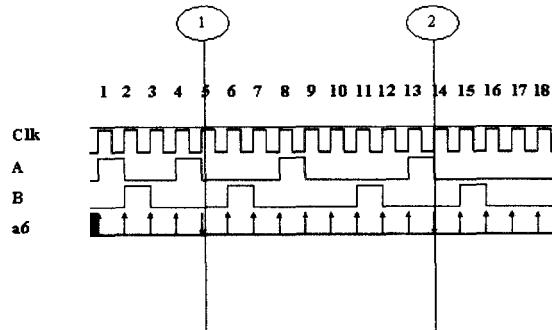


图 1-10 带禁止属性的 SVA 检验器的波形

表 1-4 属性 p6 的真值表

时钟数	“a”的	“b”的	s6 有效	a6 状态
	采样值	采样值	开始	
1	0	0	是	成功 (同一时钟周期)
2	1	0	是	成功 (开始于 2, 结束于 4)
3	0	1	是	成功 (同一时钟周期)
4	0	0	是	成功 (同一时钟周期)
5	1	0	是	失败 (开始于 5, 结束于 7)
6	0	0	是	成功 (同一时钟周期)
7	0	1	是	成功 (同一时钟周期)
8	0	0	是	成功 (同一时钟周期)
9	1	0	是	成功 (开始于 9, 结束于 11)
10	0	0	是	成功 (同一时钟周期)
11	0	0	是	成功 (同一时钟周期)
12	0	1	是	成功 (同一时钟周期)
13	0	0	是	成功 (同一时钟周期)
14	1	0	是	失败 (开始于 14, 结束于 16)
15	0	0	是	成功 (同一时钟周期)
16	0	1	是	成功 (同一时钟周期)
17	0	0	是	成功 (同一时钟周期)

1.13 一个简单的执行块

SystemVerilog 语言被定义成每当一个断言检查失败，模拟器在默认情况下都会打印出一条错误信息。模拟器不需要对成功的断言打印任何东西。读者同样也可以使用断言陈述中的“执行块”(action block)来打印自定义的成功或失败信息。执行块的基本语法如下所示。

```
assertion_name :  
    assert property(property_name)  
        <success message>;  
    else  
        <fail message>;
```

下面显示的检验器 a7 在执行块中使用了简单的显示语句来打印成功和失败信息。

```
property p7;  
    @ (posedge clk) a ##2 b;  
endproperty  
  
a7 : assert property(p7)  
    $display("Property p7 succeeded\n");  
    else  
        $display("Property p7 failed\n");
```

执行块不仅仅局限于显示成功和失败。它可以有其他的应用，例如：控制模拟环境和收集功能覆盖数据。这些主题将在第 2 章详细讨论。

1.14 蕴含操作符

属性 p7 有下列特别之处：

- (1) 属性在每一个时钟上升沿寻找序列的有效开始。在这种情况下，它在每个时钟上升沿检查信号“a”是否为高。

(2) 如果信号“a”在给定的任何时钟上升沿不为高，检验器将产生一个错误信息。这并不是一个有效的错误信息，因为我们并不关心只检查信号“a”的电平。这个错误只表明我们在这个时钟周期没有得到检验器的有效起始点。虽然这些错误是良性的，它们会在一段时间内产生大量的错误信息，因为检查在每个时钟周期都被执行。为了避免这些错误，某种约束技术需要被定义来在检查的起始点不有效时忽略这次检查。

SVA 提供了一项技术来实现这个目的。这项技术叫作“蕴含”(Implication)。

蕴含等效于一个 if-then 结构。蕴含的左边叫作“先行算子”(antecedent)，右边叫作“后续算子”(consequent)。先行算子是约束条件。当先行算子成功时，后续算子才会被计算。如果先行算子不成功，那么整个属性就默认地被认为成功。这叫作“空成功”(vacuous success)。蕴含结构只能被用在属性定义中，不能在序列中使用。

蕴含可以分为两类：交叠蕴含(Overlapped implication)和非交叠蕴含(Non-overlapped implication)。

1.14.1 交叠蕴含

交叠蕴含用符号“|->”表示。如果先行算子匹配，在同一个时钟周期计算后续算子表达式。下面用一个简单的例子解释。属性 p8 检查信号“a”在给定的时钟上升沿是否为高电平，如果 a 为高，信号“b”在相同的时钟边沿也必须为高。

```
property p8;
  @(posedge clk) a |-> b;
endproperty

a8 : assert property(p8);
```

图 1-11 显示了断言 a8 在模拟中的响应。表 1-5 总结了信号“a”和信号“b”的采样值和断言的状态。表中一共显示了三种结果。当信号“a”检测为有效的高电平，而且信号“b”在同一个时钟

沿也检测为高，这是一个真正的成功。若信号“a”不为高，断言默认地自动成功，则称为空成功。相应的，失败指的是信号“a”检测为高且在同一个时钟沿信号“b”未能检测为有效的高电平。

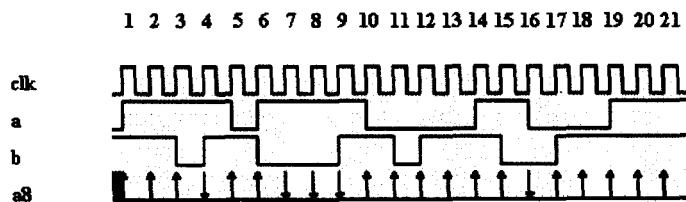


图 1-11 属性 p8 的波形

表 1-5 属性 p8 的真值表

时 钟 数	“a”的采样值	“b”的采样值	a8 的状态
1	0	1	空成功
2	1	1	真正的成功
3	1	1	真正的成功
4	1	0	失败
5	1	1	真正的成功
6	0	1	空成功
7	1	0	失败
8	1	0	失败
9	1	0	失败

1.14.2 非交叠蕴含

非交叠蕴含用符号“ $|=>$ ”表示。如果先行算子匹配，那么在下一个时钟周期计算后续算子表达式。后续算子表达式的计算总是有一个时钟周期的延迟。下面以属性 p9 举个简单的例子。该属性检查信号“a”在给定的时钟上升沿是否为高，如果为高，信号“b”必须在下一个时钟边沿为高。

```

property p9;
  @ (posedge clk) a |=> b;
endproperty

a9 : assert property(p9);
  
```

图 1-12 显示了断言 a9 在模拟中的响应。表 1-6 总结了信号“a”和信号“b”的采样值以及断言的状态。应注意的是，断言在当前时钟周期开始，在下一个时钟周期成功的情况才是真正的成功。相应的，如果属性有一个有效的开始(信号“a”为高)，且信号“b”在下一个时钟周期不为高，属性失败。

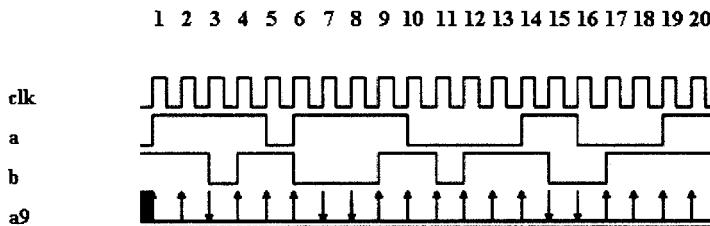


图 1-12 属性 p9 的波形

表 1-6 属性 p9 的真值表

时钟数	“a”的		a9 的状态
	采样值	采样值	
1	0	1	空成功
2	1	1	真正的成功 (开始于 2, 结束于 3)
3	1	1	败 (开始于 3, 结束于 4)
4	1	0	真正的成功(开始于 4, 结束于 5)
5	1	1	真正的成功 (开始于 5, 结束于 6)
6	0	1	空成功
7	1	0	失败 (开始于 7, 结束于 8)
8	1	0	失败(开始于 8, 结束于 9)
9	1	0	真正的成功(开始于 9, 结束于 10)

1.14.3 后续算子带固定延迟的蕴含

属性 p10 检查如果信号“a”在给定时钟上升沿为高，在两个时钟周期后信号“b”应该为高。类似的检查在前面已经用不使用蕴含的方式介绍过了。使用蕴含使得所有误报的错误都被消除。只有属性有效开始(信号“a”为高)时，才进行后续算子的检查(信

号“a”）。图 1-13 显示了属性 p10 的一个模拟的例子。表 1-7 总结了属性 p10 中信号的采样值。

```
property p10;
  @ (posedge clk) a |> ##2 b;
endproperty

a10 : assert property(p10);
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

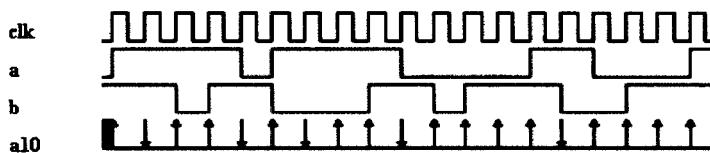


图 1-13 属性 p10 的波形

表 1-7 属性 p10 的真值表

时钟数	“a”的采样值		“b”的采样值		a10 的采样值
	采样值	采样值	采样值	采样值	
1	0	1	1	0	空成功
2	1	1	1	1	失败 (开始于 2, 结束于 4)
3	1	1	1	1	成功 (开始于 3, 结束于 5)
4	1	0	1	1	成功 (开始于 4, 结束于 6)
5	1	1	1	1	失败 (开始于 5, 结束于 7)
6	0	1	1	1	空成功
7	1	0	1	1	失败 (开始于 7, 结束于 9)
8	1	0	1	1	成功 (开始于 8, 结束于 10)
9	1	0	1	1	成功 (开始于 9, 结束于 11)

1.14.4 使用序列作为先行算子的蕴含

属性 p10 在先行算子的位置使用的是信号。先行算子同样可以使用序列的定义。在这种情况下，仅当先行算子中的序列成功时，才计算后续算子中的序列或者布尔表达式。在任何给定的时

钟周期，序列 s11a 检查如果信号“a”和信号“b”都为高，一个时钟周期之后信号“c”应该为高。序列 s11b 检查当前时钟上升沿的两个时钟周期后，信号“d”应为低。最终的属性检查如果序列 s11a 成功，那么序列 s11b 被检查。如果没有监测到有效的序列 s11a，那么序列 s11b 将不被检查，属性检查得到一次空成功。

```
sequence s11a;
  @(posedge clk) (a && b) ##1 c;
endsequence

sequence s11b;
  @(posedge clk) ##2 !d;
endsequence

property p11;
  s11a |-> s11b;
endproperty
```

图 1-14 显示了断言 a11 在模拟中的表现。标记 1s 和 1e 表明了一个成功的属性检查的起始和结束。标记 2s 和 2e 标出了一个失败的起始和结束。在时钟周期 11，信号“a”和信号“b”都为高。这表明 2 个时钟周期以后，即时钟周期 14，信号“d”应该为低。但是在例子中的波形上信号“d”为高电平，因此属性失败。

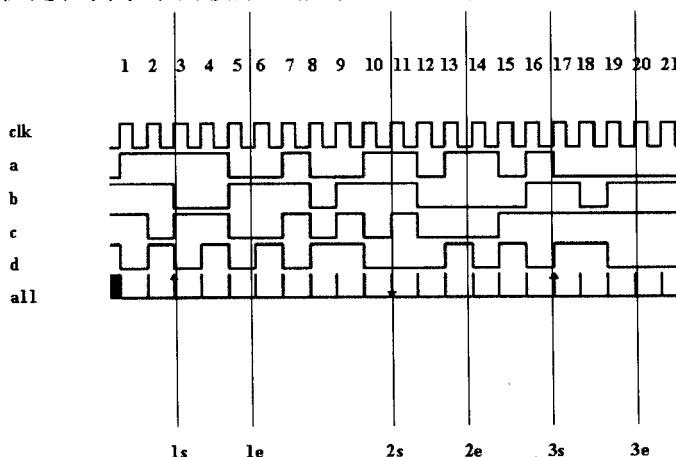


图 1-14 属性 p11 的波形

图中所有的空成功都用简单的竖线表示。标记 3s 和 3e 显示了一个成功的属性检查的起始和结束。表达式 “`a && b`” 在时钟周期 17 为真，在一个时钟周期后，信号 “`c`” 像预期的一样为高。因此在时钟周期 18，序列 `s11a` 成功。正如被期望的那样，接着信号 “`d`” 在两个时钟周期后为低。因此，属性在时钟周期 20 成功。

1.15 SVA 检验器的时序窗口

到目前为止，带延迟的例子使用的都是固定的正延迟。在下面几个例子中，我们将讨论几种不同的描述延迟的方法。

属性 `p12` 检查布尔表达式 “`a && b`” 在任何给定的时钟上升沿为真。如果表达式为真，那么在接下去的 1~3 周期内，信号 “`c`” 应该至少在一个时钟周期为高。SVA 允许使用时序窗口来匹配后续算子。时序窗口表达式左手边的值必须小于右手边的值。左手边的值可以是 0。如果它是 0，表示后续算子必须在先行算子成功的那个时钟边沿开始计算。

```
property p12;
  @ (posedge clk)  (a && b) |-> ##[1:3] c;
endproperty

a12 : assert property(p12);
```

图 1-15 显示了属性 `p12` 在模拟中的响应。每声明一个时序窗口，就会在每个时钟沿上触发多个线程来检查所有可能的成功。`p12` 实际上以下面三个线程展开。

```
(a && b) |-> ##1 c 或
(a && b) |-> ##2 c 或
(a && b) |-> ##3 c
```

属性有三个机会成功。所有三个线程具有相同的起始点，但是一旦第一个成功的线程将使整个属性成功。应当注意，在任何时钟上升沿只能有一个有效的开始，但是可以有多个有效的结束。这是因为每个有效的起始可以有三个机会成功。

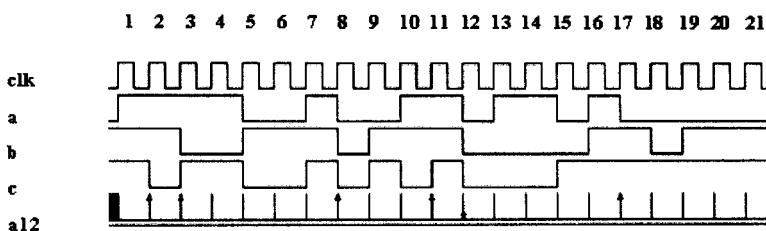


图 1-15 p12 的波形

表 1-8 总结了属性计算过程中所有相关信号的采样值。在任何给定的时钟上升沿，如果信号“a”和信号“b”不全为高，那么属性得到一个空成功。另一方面，如果信号“a”和信号“b”都为高，那么属性就有一个有效的开始。如果信号“c”在之后的 1~3 个时钟周期内都没被检测到高电平，属性失败。

表 1-8 属性 p12 的真值表

时钟数	“a”的	“b”的	“c”的	p12 的	a12 的状态
	采样值	采样值	采样值	有效开始	
1	0	1	1	否	空成功
2	1	1	1	是	真正的成功(开始于 2, 结束于 4)
3	1	1	0	是	真正的成功(开始于 3, 结束于 4)
4	1	0	1	否	空成功
5	1	0	1	否	空成功
6	0	1	0	否	空成功
7	0	1	0	否	空成功
8	1	1	1	是	真正的成功(开始于 8, 结束于 10)
9	0	0	0	否	空成功
10	0	1	1	否	空成功
11	1	1	0	是	真正的成功(开始于 11, 结束于 12)

(续表)

时钟数	“a”的采样值		“b”的采样值		“c”的采样值		p12 的有效开始	a12 的状态
12	1	1	1	1	1	1	是	失败 (开始于 12, 结束于 15)
13	0	0	0	0	0	0	否	空成功
14	1	0	0	0	0	0	否	空成功
15	1	0	0	0	0	0	否	空成功
16	0	0	0	1	1	1	否	空成功
17	1	1	1	1	1	1	是	真正的成功(开始于 17, 结束于 18)

注意，属性在时钟周期 2 和 3 都有有效的开始。这两个有效的开始同时在时钟周期 4 成功。时钟周期 2 开始的检查在两个时钟周期后检测到信号“c”为高。而时钟周期 2 开始的检查在一个时钟周期后检测到信号“c”为高。这两个都是有效情况，因此它们都成功了。在时钟周期 12 同样有一个有效的开始。属性在时钟周期 13, 14 和 15 都检测信号“c”是否为高。由于信号“c”在这所有三个可能的时钟周期始终为低，检测失败。

1.15.1 重叠的时序窗口

属性 p13 与属性 p12 相似。两者最大的区别是 p13 的后续算子在先行算子成功的同一个时钟沿开始计算。

```
Property p13;
  @ (posedge clk)  (a && b) |-> ##[0: 2] c;
endproperty
```

```
a13 : assert property(p13);
```

图 1-16 显示了 p13 在模拟中的响应。与属性 p12 最大的区别在于一个成功的开始发生在时钟周期 12。这个成功是因为检查发生了重叠。信号“c”的值在先行算子成功的同一个时钟沿被检测为高。

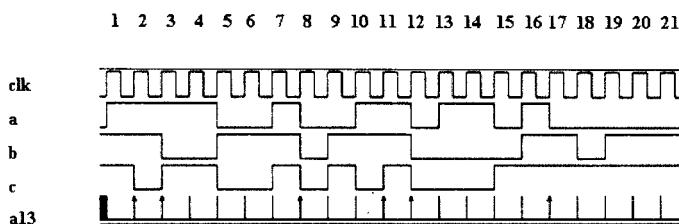


图 1-16 属性 p13 的波形

1.15.2 无限的时序窗口

在时序窗口的窗口上限可以用符号“\$”定义，这表明时序没有上限。这叫作“可能性”(eventuality)运算符。检验器不停地检查表达式是否成功直到模拟结束。因为会对模拟的性能产生巨大的负面影响，所以这不是编写 SVA 的一个高效的方式。最好总是使用有限的时序窗口上限。

属性 p14 在任何给定的时钟上升沿检查信号“a”是否为高。如果为高，那么信号“b”从下一个时钟周期往后最终将为高，而信号“c”在信号“b”为高的时钟周期开始往后最终将为高。

```
property p14;
  @(posedge clk) a |-> ##[1: $] b ##[0: $] c;
endproperty

a14 : assert property(p14);
```

图 1-17 显示了属性 p14 在模拟中的响应。表 1-9 总结了断言 a14 和相关信号的采样值。值得注意的是，真正的成功可能在任意个时钟周期后结束。如果一个有效的开始发生，而信号“b”或信号“c”在模拟结束前始终不为高，这些检查被报告为“未完成检验”(incomplete check)。因为信号“b”和信号“c”可以重叠地满足检验，整个检查有可能在一个时钟周期内结束。时钟周期 17 显示了这样一种情况，当信号“a”在时钟周期 17 被检测为高，且信号“b”和信号“c”在时钟周期 18 都被检测为高。

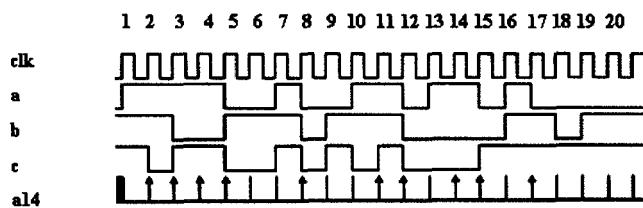


图 1-17 属性 p14 的波形

表 1-9 p14 的真值表

时钟数	"a"的 采样值	"b"的 采样值	"c"的 采样值	p14 有 效开始	a14 的状态
1	0	1	1	否	空成功
2	1	1	1	是	真正的成功 (开始于 2, 结束于 4)
3	1	1	0	是	真正的成功 (开始于 3, 结束于 8)
4	1	0	1	是	真正的成功 (开始于 4, 结束于 8)
5	1	0	1	是	真正的成功 (开始于 5, 结束于 8)
6	0	1	0	否	空成功
7	0	1	0	否	空成功
8	1	1	1	是	真正的成功 (开始于 8, 结束于 10)
9	0	0	0	否	空成功
10	0	1	1	否	空成功
11	1	1	0	是	真正的成功 (开始于 11, 结束于 12)
12	1	1	1	是	真正的成功 (开始于 12, 结束于 17)
13	0	0	0	否	空成功
14	1	0	0	是	真正的成功 (开始于 14, 结束于 17)
15	1	0	0	是	真正的成功 (开始于 15, 结束于 17)
16	0	0	1	否	空成功
17	1	1	1	是	真正的成功 (开始于 17, 结束于 18)

1.16 “ended” 结构

到目前为止，定义的序列都只是用了简单的连接(concatenation)的机制。换句话说，就是将多个序列以序列的起始点作为同步点，来组合成时间上连续的检查。SVA 还提供了另一种使用序列的结束点作为同步点的连接机制。这种机制通过给序列名字追加上关键词“ended”来表示。例如，`s.ended` 表示序列的结束点。关键词“ended”保存了一个布尔值，值的真假取决于序列是否在特定的时钟边沿匹配检验。这个 `s.ended` 的布尔值只有在相同时钟周期有效。

序列 `s15a` 和 `s15b` 是两个需要多个时钟周期来完成的简单序列，属性 `p15a` 检查序列 `s15a` 和序列 `s15b` 满足两者间隔一个时钟周期的延迟分别匹配检验。属性 `p15b` 检查相同的协议，但是使用了关键词“ended”。在这种情况下，两个序列在结束点同步。由于使用了结束点，两个序列间加上了两个时钟周期的延迟，来保证断言检验的协议与 `p15a` 相同。

```
sequence s15a;
  @ (posedge clk) a ##1 b;
endsequence

sequence s15b;
  @ (posedge clk) c ##1 d;
endsequence

property p15a;
  s15a |=> s15b;
endproperty

property p15b;
  s15a.ended |-> ##2 s15b.ended;
endproperty

a15a: assert property(p15a);
a15b: assert property(p15b);
```

图 1-18 显示了属性 p15a 和 p15b 在模拟中的响应。表 1-10 总结了断言 a15a 和 a15b 的状态。断言 a15a 的第一个真正的成功发生在时钟周期 2。当信号“a”被检测为高，检验在时钟周期 2 被激活。当信号“d”在时钟周期 5 检测为高时，检验完成。断言 a15b 的第一次真正成功出现在时钟周期 3。在时钟周期 3，当序列 s15a 成功，即信号“b”被检测为高时检验被激活。接着在时钟周期 5 当序列 s15b 成功，或者说信号“d”被检测为高时，检验完成。

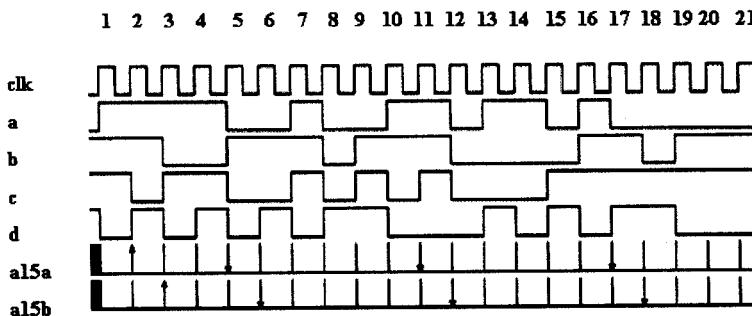


图 1-18 使用“ended”的 SVA 检验器的波形

断言 a15a 的第一次失败发生在时钟周期 5。当信号“a”在给定的时钟上升沿为高，且一个时钟周期以后(时钟周期 6)信号“b”紧接着为高时，检测到一个有效的起始点。这样就会检查后续算子，因为信号“c”在下一个时钟周期不为高，检验失败于时钟周期 7。

相应地，断言 a15b 的第一个失败出现在时钟周期 6。当序列 s15a 在时钟周期 6 成功结束时，检测到一个有效的起始点。接着检验后续算子在时钟周期 8 是否得到一个有效的结束点。因为信号“c”在时钟周期 7 不像期望的那样为高，序列的结束点的值为假，导致检验在时钟周期 8 为假。

上述例子中，我们用了两种不同的方法来实现统一个检验。第一种方法基于序列的起始点来同步序列。第二种方法基于序列的结束点来同步序列。

表 1-10 使用“ended”的 SVA 检验器的真值表

时钟数	“a”的采样值		“b”的采样值		“c”的采样值		“d”的采样值		a15a 的状态	A15b 状态
1	0		1		1		1		空成功	空成功
2	1		1		1		0		真正的成功 (开始于 2, 结束于 5)	空成功
3	1		1		0		1		空成功 (开始于 3, 结束于 5)	真正的成功
4	1		0		1		0		空成功	空成功
5	1		0		1		1		失败 (开始于 5, 结束于 7)	空成功
6	0		1		0		0		空成功	失败 (开始于 6, 结束于 8)
7	0		1		0		1		空成功	空成功
8	1		1		1		0		空成功	空成功
9	0		0		0		1		空成功	空成功
10	0		1		1		1		空成功	空成功
11	1		1		0		0		失败 (开始于 11, 结束于 13)	空成功
12	1		1		1		0		空成功	失败 (开始于 12, 结束于 14)
13	0		0		0		0		空成功	空成功
14	1		0		0		1		空成功	空成功
15	1		0		0		0		空成功	空成功
16	0		0		1		1		空成功	空成功
17	1		1		1		0		失败 (开始于 17, 结束于 20)	空成功

1.17 使用参数的 SVA 检验器

SVA 允许像 Verilog 那样在检验器中使用参数(parameter)。这为创建可重用的属性提供了很大的灵活性。比如，两个信号间的延迟信息可以在检验器中用参数表示，那么这种检验器就可以在设计只有时序关系不同的情况中重用。例子 1.2 显示了一个带延迟默认值参数的检验器。如果这个检验器在设计中被调用，它使用一个时钟周期作为延迟默认值。如果在实例化时重设检验器中延迟参数值，那么同一个检验器就可以被重用。在例子 1.2 中，模块“top”有两个“generic_chk”的实例。实例 i1 将延迟参数改写为 2 个时钟周期，而实例 i2 使用默认的 1 个时钟周期。

例 1.2 使用参数的 SVA 检验器的例子

```
module generic_chk (input logic a, b, clk);

parameter delay = 1;

property p16;
    @ (posedge clk) a |>> ##delay b;
endproperty

a16: assert property(p16);

endmodule

// call checker from the top level module

module top(. . .);
    logic clk, a, b, c, d;
    .
    .
    generic_chk #(delay(2)) i1 (a, b, clk);
    generic_chk i2 (c, d, clk);

```

```
endmodule
```

图 1-19 显示了两个检验器实例 i1 和 i2 在模拟过程中对信号变化的响应。

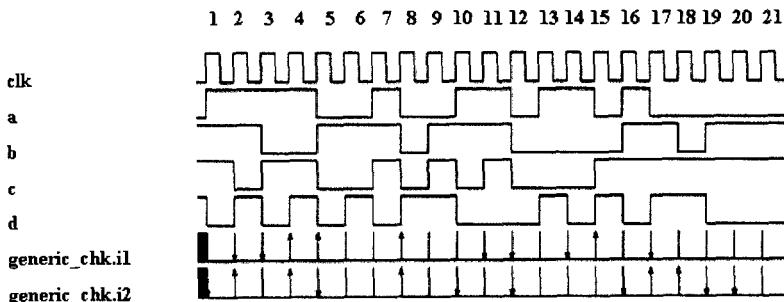


图 1-19 带参数的 SVA 检验器的波形

1.18 使用选择运算符的 SVA 检验器

SVA 允许在序列和属性中使用逻辑运算符。属性 p17 检查如果信号“c”为高，那么信号“d”的值与信号“a”的相等。如果信号“c”不为高，那么信号“d”的值与信号“b”的相等。这是一个组合的检验，在每个时钟上升沿被执行。

```
property p17;
  @ (posedge clk) c ? d == a : d == b;
endproperty

a17: assert property (p17);
```

图 1-20 显示了属性 p17 在模拟中的响应。表 1-11 总结了断言 a17 的状态和涉及的信号的采样值。在时钟周期 1，信号“c”被检测为高，因此检验期望信号“d”和信号“a”有相等的值。但是信号“d”被检测为高，而信号“a”为低，所以检验失败。

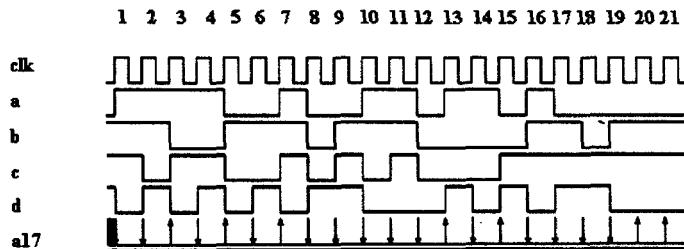


图 1-20 使用选择运算符的 SVA 检验器的波形

表 1-11 使用选择运算器的 SVA 检验器的真值表

时钟数	“a”的	“b”的	“c”的	“d”的	a17
	采样值	采样值	采样值	采样值	的状态
1	0	1	1	1	失败
2	1	1	1	0	失败
3	1	1	0	1	成功
4	1	0	1	0	失败
5	1	0	1	1	成功
6	0	1	0	0	失败
7	0	1	0	1	成功
8	1	1	1	0	失败
9	0	0	0	1	失败
10	0	1	1	1	失败
11	1	1	0	0	失败
12	1	1	1	0	失败
13	0	0	0	0	成功
14	1	0	0	1	失败
15	1	0	0	0	成功
16	0	0	1	1	失败
17	1	1	1	0	失败

1.19 使用 true 表达式的 SVA 检验器

使用 true 表达式，可以在时间上延长 SVA 检验器。这代表一种忽略的状态，它使得序列延长了一个时钟周期。这可以用来实现同时监视多个属性且需要同时成功的复杂协议。

序列 s18a 检查一个简单的条件。序列 s18a_ext 检查相同的条件，但是序列的成功被往后移了一个时钟周期。当这个序列被用在一个属性的现行算子时，它会造成一些差异。两个序列的结束点不同，因此开始检查后续算子的时钟周期也不一样。

属性 p18 检查先行算子中的 s18a.end，如果成功，两个时钟周期后，检查 s18b.end 是否成功。属性 p18_ext 检查 s18a_ext 在先行算子中是否成功。这个成功与 s18a.ended 的成功相同，但是早了一个时钟周期。因此属性 p18_ext 的后续算子需要在一个时钟周期而不是像 p18 中定义的两个时钟周期后成功。属性 p18 和 p18_ext 检查相同的情况，但是他们在先行算子中的成功点却不同。

```
'define true 1

sequence s18a;
  @ (posedge clk) a ##1 b;
endsequence

sequence s18a_ext;
  @ (posedge clk) a ##1 b ##1 `true;
endsequence

sequence s18b;
  @ (posedge clk) c ##1 d;
endsequence

property p18
  @(posedge clk) s18a.ended |-> ##2 s18b.ended;
endproperty

property p18_ext
  @(posedge clk) s18a_ext.ended |-> s18b.ended;
```

```

endproperty

a18: assert property(p18);
a18_ext: assert property(p18_ext);

```

图 1-21 显示了属性 p18 和 p18_ext 在模拟中的响应。可以清楚地看到与断言 a18 比较，断言 a18_ext 的起始点被推迟了一个时钟周期。

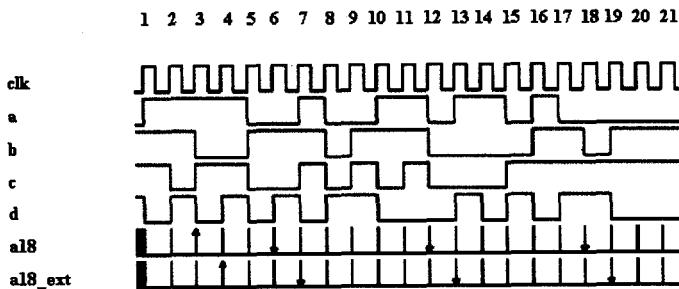


图 1-21 使用 true 表达式的 SVA 检验器的波形

1.20 “\$past” 构造

SVA 提供了一个内嵌的系统任务 “\$past” ，它可以得到信号在几个时钟周期之前的值。在默认情况下，它提供信号在前一个时钟周期的值。结构的基本语法如下。

```
$past (signal_name, number of clock cycles)
```

这个任务能够有效地验证设计到达当前时钟周期的状态所采用的通路是正确的。属性 p19 检验的是在给定的时钟上升沿，如果表达式($c \&& d$)为真，那么两个周期前，表达式($a \&& b$)为真。

```

property p19;
  @(posedge clk) (c && d) |->
    ($past((a&&b), 2) == 1'b1);
endproperty

a19: assert property(p19);

```

图 1-22 显示了属性 p19 在模拟中的响应。表 1-12 总结了断言 a19 的状态和相关信号的采样值。断言在时钟周期 1 失败。在时钟周期 1，信号“c”和信号“d”都为高，断言有一个有效的开始。于是检验器的后续算子需要比较两个周期前的表达式($a \& \& b$)的值。但是由于不可能得到两个信号在时钟周期 1 之前两个周期的历史，信号的值被当作“x”，因此检验器在时钟周期 1 失败。

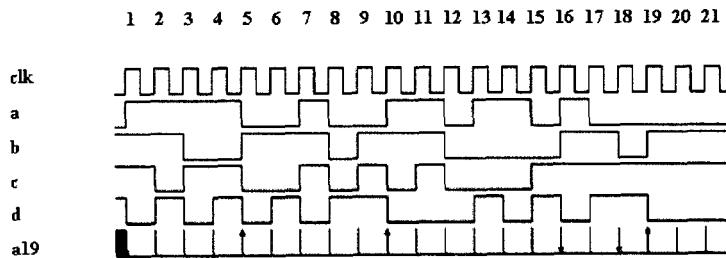


图 1-22 使用“\$past”的 SVA 检验器的波形

检验在时钟周期 5 有一个真正的成功。在时钟周期 5，由于信号“a”和信号“b”都为高，断言有一个成功的开始。后续算子检查在时钟周期 3 时表达式($a \& \& b$)是否为真。正如期望的那样，在时钟周期 3，信号“a”和信号“b”都被检测为高，因此检验成功。

检验在时钟周期 16 失败。在时钟周期 16，由于信号“a”和信号“b”都为高，断言也有一个成功的开始。后续算子检查在时钟周期 3 时表达式($a \& \& b$)是否为真。信号“a”如期望的那样为高但是信号“b”为低。这使得表达式($a \& \& b$)为假，检验失败。

表 1-12 使用“\$past”的 SVA 检验器的真值表

时钟数	“a”的	“b”的	“c”的	“d”的	a19 的状态
	采样值	采样值	采样值	采样值	
1	0	1	1	1	失败
2	1	1	1	0	空成功
3	1	1	0	1	空成功
4	1	0	1	0	空成功
5	1	0	1	1	真正的成功
6	0	1	0	0	空成功

(续表)

时钟数	“a”的	“b”的	“c”的	“d”的	a19
	采样值	采样值	采样值	采样值	的状态
7	0	1	0	1	空成功
8	1	1	1	0	空成功
9	0	0	0	1	空成功
10	0	1	1	1	真正的成功
11	1	1	0	0	空成功
12	1	1	1	0	空成功
13	0	0	0	0	空成功
14	1	0	0	1	空成功
15	1	0	0	0	空成功
16	0	0	1	1	失败
17	1	1	1	0	空成功

带时钟门控的\$past 构造

\$past 构造可以由一个门控信号(gating singal)控制。比如，在一个给定的时钟沿，只有当门控信号的值为真时才检查后续算子的状况。使用门控信号的\$past 构造的基本语法如下：

```
$past (signal_name, number of clock cycles, gating signal)
```

属性 p20 与属性 p19 相似。但是只有当门控信号 “e” 在任意给定的时钟上升沿有效时检验才被激活。

```
property p20;
  @ (posedge clk) (c && d) |->
    ($past((a&&b), 2, e) == 1'b1);
endproperty

a20: assert property(p20);
```

1.21 重复运算符

如果信号“start”在任何给定的时钟上升沿跳变为高，接着从下一个时钟周期起，信号“a”保持三个连续时钟周期为高，然后下一个时钟周期，信号“stop”为高。

像上述描述的序列可以使用下面的 SVA 代码来检验。

```
@(posedge clk) $rose(start) |->
    ##1 a ##1 a ##1 a ##1 stop
```

如果信号“a”需要在很多个周期中保持高电平，编写这样一个检验器可能会非常冗长。而且这个例子要求信号“a”连续地保持高电平。当我们只希望检查信号“a”是否在被检测时保持为高，而不一定是三个连续的时钟周期的时候，协议就会变得复杂起来。换句话说，信号“a”需要连续地或者间歇地重复自己三次。

SVA 语言提供三种不同的重复运算符：连续重复(consecutive repetition)，跟随重复(go to repetition)，非连续重复(non-consecutive repetition)。

连续重复——允许用户表明信号或者序列将在指定数量的时钟周期内都连续地匹配。信号的每次匹配之间都有一个时钟周期的隐藏延迟。连续重复运算符的基本语法如下所示。

```
signal or sequence [*n]
```

“n”是表达式应该匹配的次数。

比如， $a[*3]$ 可以被展开成下面的式子。

```
a ##1 a ##1 a
```

而序列 $(a##1b)[*3]$ 可以展开为

```
(a ##1 b) ##1 (a ##1 b) ##1 (a ##1 b)
```

跟随重复——允许用户表明一个表达式将匹配达到指定的次数，而且不一定在连续的时钟周期上发生。这些匹配可以是间歇的。跟随重复的主要要求是被检验重复的表达式的最后一个匹配

应该发生在整个序列匹配结束之前。跟随重复运算符的基本语法如下所示。

```
signal [->n]
```

参考下面的序列：

```
start ##1 a[->3] ##1 stop
```

这个序列需要信号“a”的匹配(即信号“a”的第三次，也就是最后一次重复的匹配)正好发生在“stop”成功之前。换句话说，信号“stop”在序列的最后一个时钟周期匹配，而且在前一个时钟周期，信号“a”有一次匹配。

非连续重复——与跟随重复相似，除了它并不要求信号的最后一次重复匹配发生在整个序列匹配前的那个时钟周期。非连续重复运算符的基本语法如下所示。

```
signal [=n]
```

在跟随重复和非连续重复中只允许使用表达式，不能使用序列。

1.21.1 连续重复运算符[*]

属性 p21 检查在检验有效地开始两个时钟周期后，信号“a”在连续的三个时钟周期为高，再过两个时钟周期，信号“stop”为高。下一个时钟周期，信号“stop”为低。

```
property p21;
    @ (posedge clk) $rose(start) |->
        ##2 (a[*3]) ##2 stop ##1 !stop;
endproperty

a21: assert property(p21);
```

图 1-23 显示了属性 p21 在模拟中的响应。波形中显示了 2 个失败和 1 个真正的成功。其他成功都是空成功。

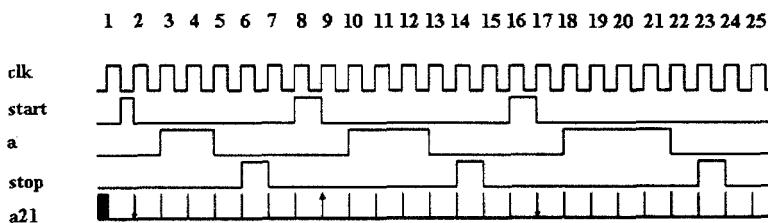


图 1-23 使用连续重复的 SVA 检验器的波形

断言失败于时钟周期 2——时钟周期 2 有一个有效的开始信号。检验器接着检验信号“a”是否从时钟周期 4 的上升沿开始有连续三个时钟周期为高。信号“a”在时钟周期 4 和 5 为高，但是在时钟周期 6 为低。因此检验失败。检查从时钟周期 2 开始，在时钟周期 6 失败。

断言成功于时钟周期 9——在时钟周期 9 检测到一个有效的开始。于是检验器检查信号“a”是否在时钟周期 11 开始的 3 个连续时钟周期都为高。信号“a”在时钟周期 11、12、13 都像预期的那样被检测为高。两个时钟周期后(时钟周期 15)，信号“stop”如预期地为高。一个时钟周期以后，“stop”被检测为低。至此检验成功。注意，检查从时钟周期 9 开始，结束于时钟周期 16。

断言失败于时钟周期 17——在时钟周期 17 检测到一个有效的开始。于是检验器检查信号“a”是否在时钟周期 19 开始的 3 个连续时钟周期都为高。信号“a”在时钟周期 19、20、21 都像预期的那样为高。接着检验器检查信号“stop”在时钟周期 23 是否为高，但是没检测到。因此检验失败。可以看到，信号“a”保持了 4 个时钟周期的高电平。但是检验器只需要检查 3 个重复，因此直接继续检查信号“stop”。整个检查从时钟周期 19 开始，失败于时钟周期 23。

1.21.2 用于序列的连续重复运算符[*]

属性 p22 检查有效开始的两个时钟周期以后，序列(a ##2 b)重复三次，接着再过两个时钟周期，信号“stop”为高。

```

property p22;
  @(posedge clk) $rose(start) |->
    ##2 ((a ##2 b)[*3]) ##2 stop;
endproperty

a22: assert property(p22);

```

图 1-24 显示了属性 p22 在模拟中的响应。图中共显示了 2 个失败和 1 个真正的成功。

失败 1——第一个失败由标记 1s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器期望序列(a ##2 b)重复 3 次。但是序列只重复了两次。因此检验器失败，失败点由标记 1e 标出。

成功 1——唯一一个真正的成功由标记 2s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器开始检查序列(a ##2 b)是否重复 3 次。序列如预期地重复了 3 次。在序列重复被检验后，再过两个时钟周期，信号“stop”也如期望地为高。因此检验器成功，成功点由标记 2e 标出。

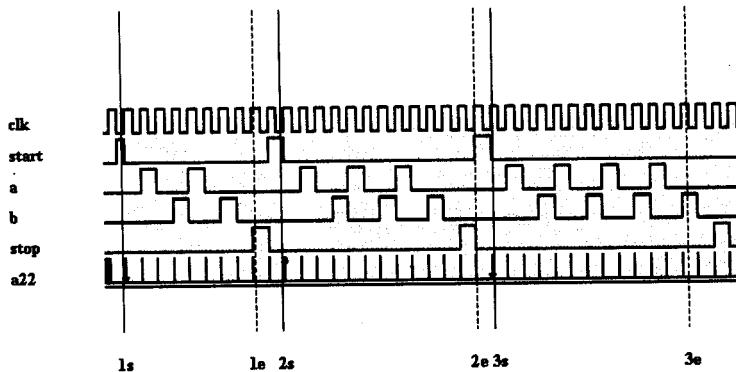


图 1-24 检查连续重复的序列的 SVA 检验器的波形

失败 2——第二个失败由标记 3s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器开始检查序列(a ##2 b)是否重复 3 次。序列如预期地重复了 3 次。当序列重复被检验到后，信号“stop”被期望在两个时钟周期后为高，但是失败了。因此检验器失败，失败点由标记 3e 标出。

1.21.3 用于带延迟窗口的序列的连续重复运算符[*]

属性 p23 检查在有效开始的两个时钟周期后，序列(a ##[1:4] b)重复 3 次，接着再过两个时钟周期，信号“stop”为高。实际上，这个序列有一个时序窗口，使得情况变得有些复杂。

```
property p23;
  @(posedge clk) $rose(start) |->
    #(2 ((a ##[1:4] b) [*3])) ##2 stop;
endproperty

a23: assert property(p23);
```

主序列(a ##[1: 4] b)[*3]可以被扩展成：

```
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b))
```

图 1-25 显示了属性 p23 在模拟中的响应。图中有 2 个失败和 1 个真正的成功。

失败 1——第一个失败由标记 1s 标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(a ##[1:4] b)重复 3 次。但是序列只重复了 2 次。因此检验器失败，失败点由标记 1e 标出。可以看到成功的两个重复分别是(a ##1 b)和(a ##2 b)。

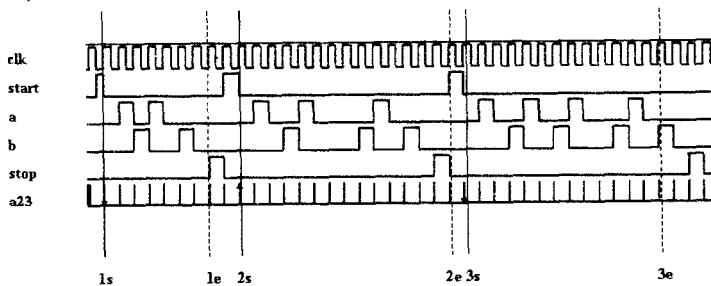


图 1-25 检查连续重复的带延迟窗口的序列的 SVA 检验器的波形

成功 1——唯一的真正成功由标记 2s 标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(a

`##[1: 4] b`)重复 3 次。序列如预期地重复了 3 次。在成功重复之后，信号“stop”如期望地在两个时钟周期后为高。因此检验器成功了，成功点由标记`2e`标出。可以看到成功的三个重复分别是(`a ##2 b`)，(`a ##4 b`)和(`a ##2 b`)。

失败 2——第二个失败由标记`3s`标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(`a ##[1: 4] b`)重复 3 次。序列如预期地重复了 3 次。在成功地重复之后，信号“stop”被期望地在两个时钟周期后为高，但是失败了。因此检验器失败，失败点由标记`3e`标出。可以看到成功的三个重复分别是(`a ##2 b`)，(`a ##2 b`)和(`a ##3 b`)。

1.21.4 连续运算符[*]和可能性运算符

属性`p23`指定了一个重复序列的时序窗口。同样的，重复的次数也可以是一个窗口。比如，`a[*1: 5]`表示信号“a”从某个时钟周期开始重复 1~5 次。这个定义可以展开成下面的表达式。

```
a or
(a ##1 a) or
(a ##1 a ##1 a) or
(a ##1 a ##1 a ##1 a) or
(a ##1 a ##1 a ##1 a ##1 a)
```

重复窗口的边界规则与延迟窗口的相同。左边的值必须小于右边的值。右边的值可以是符号“\$”，这表示没有重复次数的限制。

属性`p24`显示了一个带没有重复次数限制的有限的检查。它检验有效开始两个时钟周期后，信号“a”将保持为高，直到信号“stop”跳变为高。

```
property p24;
  @(posedge clk) $rose(start) |->
    ##2 (a[*1: $]) ##1 stop;
endproperty

a24: assert property(p24);
```

图 1-26 显示了属性 p24 在模拟中的响应。图中有 1 个失败和 1 个真正的成功。

失败 1——一个有效的开始发生在时钟周期 3，如标记 1s 所示。检查期望在两个时钟周期后，信号“a”将保持为高直到信号“stop”有效。信号“a”一直为高直到时钟周期 7。在时钟周期 8，“a”被检测为低，但是信号“stop”仍然不为高。因此检验在时钟周期 8 失败，如标记 1e 所示。

成功 1——一个有效的开始发生在时钟周期 11，如标记 1s 所示。检查期望在两个时钟周期后，信号“a”将保持为高直到信号“stop”有效。信号“a”一直为高直到时钟周期 15。在时钟周期 16，“a”被检测为低，但是信号“stop”如期望地为高。因此检验在时钟周期 16 成功，如标记 2e 所示。

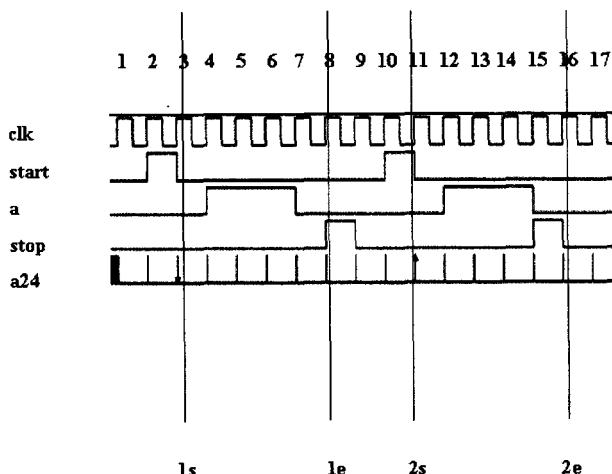


图 1-26 使用连续重复和可能性运算符的 SVA 检验器的波形

1.21.5 跟随重复运算符[>]

属性 p25 检查如果在任何时钟上升沿有有效的开始，两个时钟周期后，信号“a”连续或者间断地出现 3 次为高，接着信号“stop”在下一个时钟周期为高。

```
property p25;
  @ (posedge clk) $rose (start) |->
```

```

##2 (a [->3]) ##1 stop;
endproperty

a25: assert property(p25);

```

图 1-27 显示了属性 p25 在模拟中的响应。图中显示共有 1 个失败、1 个成功和一个未完成的检查。

失败 1——标记 1s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号如预期地重复 3 次，在信号“a”的第 3 次匹配后，信号“stop”没能如期望的那样在下一个时钟周期为高。因此检查在标记 1e 所示位置失败了。

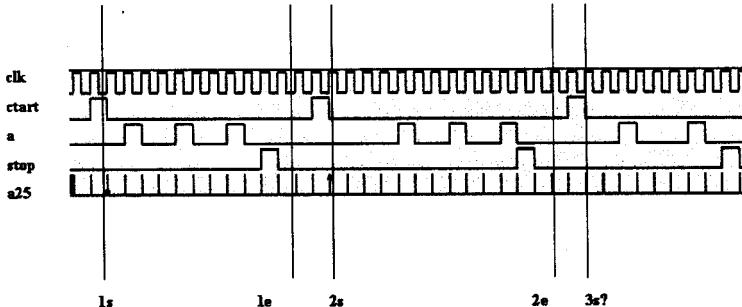


图 1-27 采用跟随重复操作符的 SVA 检验器的波形

成功 1——标记 2s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期，信号“a”将重复 3 次。信号如预期地重复 3 次。在信号“a”的第 3 次匹配后，在下一个时钟周期信号“stop”如期望的那样为高。因此检查在标记 2e 所示位置成功了。

未完成 1——标记 3s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号重复了两次，模拟就结束了。应注意到，在模拟周期结束前，信号“stop”出现了一次有效。由于重复语句还没有完成，这个有效的“stop”并没有发生任何作用。检验 3 个重复的语句阻塞了信号“stop”的检验。因此在模拟结束时这个检查未能完成。

1.21.6 非连续重复运算符[=]

属性 p26 检查如果在任何时钟上升沿有有效的开始信号，两个时钟周期后，在一个有效的“stop”信号前，信号“a”连续或者间断地出现 3 次为高，然后一个时钟周期后“stop”应该为低。p26 和 p25 做的是相同的检查，唯一的不同是 p26 使用的是非连续 (non-consecutive) 重复运算符而不是跟随(go to)重复运算符。这表示在属性 p26 中，在信号“stop”有效匹配的前一个时钟周期，信号“a”不一定需要有有效的匹配。

```
Property p26;
  @ (posedge clk) $rose(start) |->
    ##2 (a[=3]) ##1 stop ##1 !stop;
endproperty

a26: assert property(p26);
```

图 1-28 显示了属性 p26 在模拟中的响应。图中显示有 2 个真正的成功和 1 个未完成的检查。

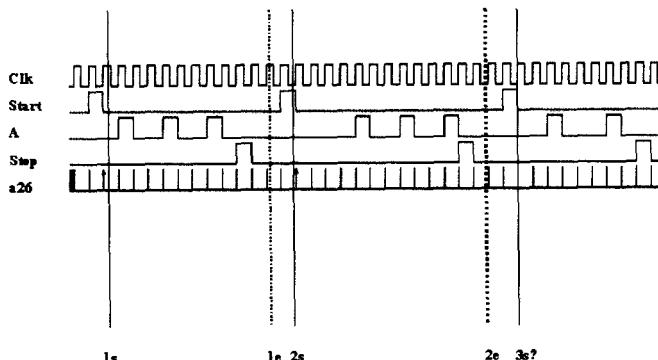


图 1-28 使用非连续重复运算符的 SVA 检验器的波形

成功 1——标记 1s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号“a”如预期地重复 3 次，在“a”的第三个匹配后，期望一个有效的信号“stop”，但是不必在下一个时钟周期发生。实际上，在信号“a”的第三次匹配的两个时钟周期后有一个有效的信号“stop”，因此

检验如标记 1e 所示的成功了。这就是跟随重复和非连续重复的不同之处。在相同情况下，属性 p25 由于使用的是跟随重复而失败了。

成功 2——标记 2s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号“a”如预期地重复 3 次，在“a”的第三次匹配后，期望一个有效的信号“stop”，但不必在下一个时钟周期发生。实际上，在信号“a”的第三次匹配的 1 个时钟周期后有一个有效的信号“stop”，因此检验如标记 2e 所示的成功了。

未完成 1——标记 3s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。实际上，信号“a”重复了两次，在第 3 次重复出现前，模拟结束了。同样应注意，信号“stop”在模拟周期结束前曾经出现为高。因为重复语句还没有完成，所以这个“stop”并没有起到任何作用。信号“a”重复三次的语句阻塞了信号“stop”的检验。因此在模拟结束时检验未完成。这个行为与跟随重复(“go to” repetition) 相同。

1.22 “and” 构造

二进制运算符“and”可以用来逻辑地组合两个序列。当两个序列都成功时整个属性才成功。两个序列必须具有相同的起始点，但是可以有不同的结束点。检验的起始点是第一个序列的成功时的起始点，而检验的结束点是使得属性最终成功的另一个序列成功时的点。

序列 s27a 和 s27b 是两个独立的序列。属性 p27 将两者用运算符“and”组合起来。当两个序列都成功时，属性成功。

```
sequence s27a;
  @ (posedge clk) a##[1:2] b;
endsequence

sequence s27b;
```

```

endsequence

property p27;
  @(posedge clk) s27a and s27b;
endproperty

a27: assert property(p27);

```

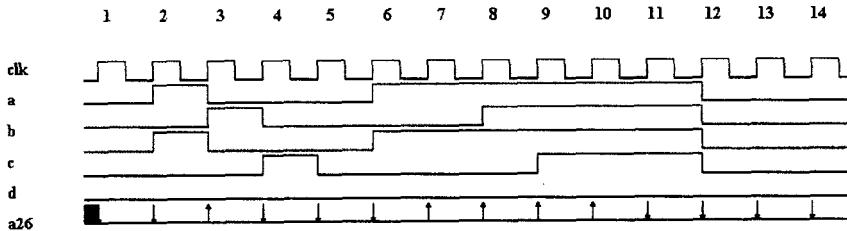


图 1-29 使用“and”构造的 SVA 检验器的波形

图 1-29 显示了属性 p27 在模拟中的响应。表 1-13 总结了断言 a27 的状态和所有相关信号的采样值。一共有三种结果。一种是没有有效开始所导致的失败。当在给定的时钟边沿信号“a”或者信号“c”不为高(时钟周期 1, 2, 4, 5, 6, 13, 14)。

表 1-13 使用“and”的 SVA 检验器的真值表

时钟数	“a”的	“b”的	“c”的	“d”的	有效	a27
	采样值	采样值	采样值	采样值	开始	的状态
1	0	0	0	0	否	失败
2	0	0	0	0	否	失败
3	1	0	1	0	是	成功 (开始于 3, 结束于 5)
4	0	1	0	0	否	失败
5	0	0	0	1	否	失败
6	0	0	0	0	否	失败
7	1	0	1	0	是	成功 (开始于 7, 结束于 10)

(续表)

时钟数	"a"的采样值	"b"的采样值	"c"的采样值	"d"的采样值	有效开始	a27的状态
8	1	0	1	0	是	成功 (开始于 8, 结束于 10)
9	1	1	1	0	是	成功 (开始于 9, 结束于 11)
10	1	1	1	1	是	成功 (开始于 10, 结束于 12)
11	1	1	1	1	是	失败 (开始于 11, 结束于 14)
12	1	1	1	1	是	失败 (开始于 12, 结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

还有 5 个不同的成功，它们各自的长度也不同。一个有效的检验开始于时钟周期 7 和时钟周期 8，但是它们同时结束于时钟周期 10。从时钟周期 7 开始的检验，信号“b”在时钟周期 9 为真，且信号“d”在时钟周期 10 为真。而从时钟周期 8 开始的检验，信号“b”在时钟周期 9 为真，且信号“d”在时钟周期 10 为真。

此外还有两个失败，分别在时钟周期 11 和 12。它们有相同的长度，但是失败的原因却不同。对于从时钟周期 11 开始的检验，信号“b”在时钟周期 12 为真。但是信号“d”在时钟周期 13 和 14 都不为真，因此检验失败于时钟周期 14。对于从时钟周期 12 开始的检验，信号“b”在时钟周期 3 不为真。而且在时钟周期 14 信号“b”和信号“d”都不为真，因此检验在时钟周期 14 失败。

1.23 “intersect” 构造

“intersect” 运算符和 “and” 运算符很相似，它有一个额外要求。两个序列必须在相同时刻开始且结束于同一时刻。换句话说，两个序列的长度必须相等。

属性 p28 检验与属性 p27 相同的情况。唯一的区别是它使用的是 “intersect” 构造而不是 “and” 构造。

```

sequence s28a;
  @ (posedge clk) a##[1: 2] b;
endsequence

sequence s28b;
  @ (posedge clk) c##[2: 3] d;
endsequence

property p28;
  @ (posedge clk) s28a intersect s28b;
endproperty
a28: assert property(p28);

```

图 1-30 显示了属性 p28 在模拟中的响应。表 1-14 总结了断言 p28 的状态和所有相关信号的采样值。图 1-30 也显示了在同一条件下使用 “and” 构造的断言 a27 的结果，从而可以帮助理解 “and” 和 “intersect” 的区别。

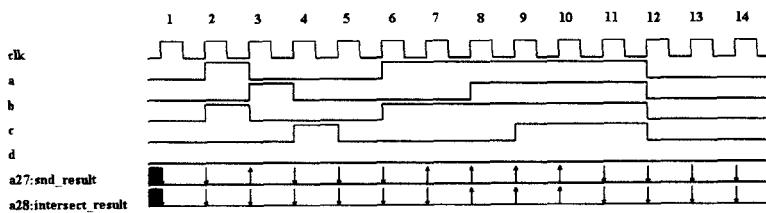


图 1-30 使用 “intersect”的 SVA 检验器的波形

由于缺少有效开始而导致的失败与前一节相同。第二类的失败是由于属性中单个序列未能成功匹配。这类的失败发生在时钟

周期 11 和 12。第三类的失败是在属性中所有单个序列都匹配时发生的。这些失败是由于两个序列达到匹配的时间长度不一致而发生的。在时钟周期 3 显示的失败中，序列 s28a 花了一个时钟周期实现匹配（“a”在时钟周期 3 为真，且“b”在时钟周期 4 为真），而序列 s28b 花了两个时钟周期实现匹配（“c”在时钟周期 3 为真，且“d”在时钟周期 5 为真）。在时钟周期 7 显示的失败中，s28a 花了两个时钟周期完成匹配（“a”在时钟周期 7 为真，且“b”在时钟周期 9 为真），而 s28b 花了三个时钟周期完成匹配（“c”在时钟周期 7 为真，且“d”在时钟周期 10 为真）。

三次成功分别发生于时钟周期 8, 9, 10。在所有这三个例子中，两个序列都经过相同的时间长度匹配。

在时钟周期 8 显示的成功中，序列 s28a 匹配了两次，分别在时钟周期 9 和 10。序列 s28b 同样也匹配了两次，分别在时钟周期 10 和 11。两个序列匹配的共同时间长度是两个时钟周期。因此 intersect 在 s28a 和 s28b 都在时钟周期 10 时匹配成功，它们各自的长度是两个时钟周期。

表 1-14 使用“intersect”的 SVA 检验器的真值表

时钟数	“a”的采样值				“b”的采样值				“c”的采样值				“d”的采样值				有效开始	a28 的状态
	采样值	采样值	采样值	采样值														
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	否	失败
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	否	失败
3	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	是	失败(序列匹配经历的长度不同)
4	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	否	失败
5	0	0	0	1	0	0	1	0	0	1	0	1	0	0	1	0	否	失败
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	否	失败
7	1	0	1	0	1	0	1	0	0	1	0	1	0	0	1	0	是	失败(序列匹配经历的长度不同)
8	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0	1	是	成功(开始于 8, 结束于 10)
9	1	1	1	1	0	1	1	1	0	0	1	1	0	0	1	1	是	成功(开始于 9, 结束于 11)

(续表)

时钟数	“a”的采样值				有效开始	a28 的状态
	采样值	采样值	采样值	采样值		
10	1	1	1	1	是	成功 (开始于 10, 结束于 12)
11	1	1	1	1	是	失败 (开始于 11, 结束于 13)
12	1	1	1	1	是	失败 (开始于 12, 结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

在时钟周期 9 所示的成功中，序列 s28a 在时钟周期 10 和 11 各有一次匹配。序列 s28b 也有两次匹配，分别在时钟周期 11 和 12。两个序列匹配的通用长度为两个时钟周期。因此 intersect 构造因为 s28a 和 s28b 都在时钟周期 11 匹配而成功。每个序列的长度都是两个时钟周期。

在时钟周期 10 所示的成功中，序列 s28a 在时钟周期 11 和 12 各有一次匹配。序列 s28b 在时钟周期 12 也有一次成功。两个序列匹配的通用长度为两个时钟周期。因此 intersect 构造因为 s28a 和 s28b 都在时钟周期 12 有匹配而成功。每个序列的长度都是两个时钟周期。

1.24 “or” 构造

二进制运算符 “or” 可以用来逻辑地组合两个序列。只要其中一个序列成功，整个属性就成功。

序列 s29a 和 s29b 是两个独立的序列。属性 p29 将两者用 “or” 运算符组合起来。当其中任一序列成功时，属性就成功。

```

sequence s29a;
  @ (posedge clk) a##[1:2] b;
endsequence

sequence s29b;
  @ (posedge clk) c##[2:3] d;
endsequence

property p29;
  @ (posedge clk) s28a or s28b;
endproperty

a29: assert property(p29);

```

图 1-31 显示了属性 p29 在模拟中的响应。表 1-15 总结了断言 a29 的状态和所有相关信号的采样值。图 1-31 也显示了使用“and”构造的断言 a27 的结果，比较两个结果就很容易理解“and”构造和“or”构造的区别。由于没有有效开始而发生的失败和前面小节介绍的相同，第二类失败是由于其中的序列都不匹配引起的。发生在时钟周期 12 的失败就是这种失败，两个序列在它们各自的时序窗口都没能匹配，因此检验失败。

使用“and”构造和“or”构造的成功几乎是相同的，主要的区别在于匹配的时间。当序列 s29a 成功时“or”运算符就匹配了，而不需要等待序列 p29b 结束。

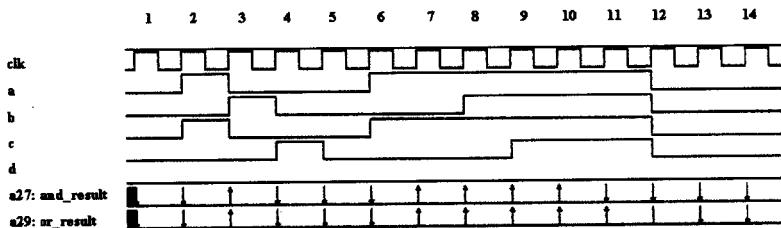


图 1-31 使用“or”的 SVA 检验器的波形

表 1-15 使用“or”构造的 SVA 检验器的波形

时钟 数	“a”的 采样值	“b”的 采样值	“c”的 采样值	“d”的 采样值	有效	a29 的状态
1	0	0	0	0	否	失败
2	0	0	0	0	否	失败
3	1	0	1	0	是	成功(开始于 3, 结束于 4)
4	0	1	0	0	否	失败
5	0	0	0	1	否	失败
6	0	0	0	0	否	失败
7	1	0	1	0	是	成功(开始于 7, 结束于 9)
8	1	0	1	0	是	成功(开始于 8, 结束于 9)
9	1	1	1	0	是	成功(开始于 9, 结束于 10)
10	1	1	1	1	是	成功(开始于 10, 结束于 11)
11	1	1	1	1	是	成功(开始于 11, 结束于 12)
12	1	1	1	1	是	失败(开始于 12, 结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

其中一个使用“and”构造的失败发生在时钟周期 11，但是当使用“or”构造时，属性却成功了一次。原因是属性的第一部分序列 s29a 在时钟周期 12 匹配，使得属性立即成功。而在使用“and”构造时，单是这个成功并不是充分条件，第二部分的序列也必须匹配，但是在给定的时序窗口内，并没能出现。因此，相同的条件下，属性 p27 在时钟周期 14 失败了。

1.25 “first_match” 构造

任何时候使用了逻辑运算符(如“and”和“or”)的序列中指定了时间窗，就有可能出现同一个检验具有多个匹配的情况。

“first_match”构造可以确保只用第一次序列匹配，而丢弃其他的匹配。当多个序列被组合在一起，其中只需时间窗内的第一次匹配来检验属性剩余的部分时，“first_match”构造非常有用。

在下面的例子中，属性用运算符“or”将两个序列组合在一起。这个属性的几个可能的匹配如下所示。

```
a ##1 b;
a ##2 b;
c ##2 d;
a ##3 b;
c ##3 d;
```

当检验属性 p30 时，第一次匹配保留下，其他匹配都被丢弃了。

```
sequence s30a;
  @ (posedge clk) a ##[1:3] b;
endsequence

sequence s30b;
  @ (posedge clk) c ##[2:3] d;
endsequence

property p30;
  @ (posedge clk) first_match(s30a or s30b);
endproperty

a30: assert property(p30);
```

图 1-32 显示了属性 p30 在模拟中的响应。图中显示了两次成功，分别在时钟周期 3 和 9。在时钟周期 3 的成功基于序列(c ##2 d)的匹配。在时钟周期 9 的成功基于序列(a ##1 b)的匹配。在这两种情况下，第一次序列匹配就使得整个属性成功。

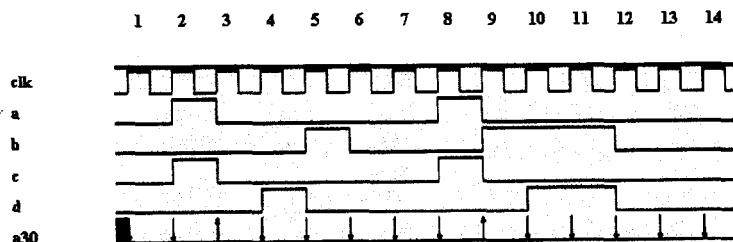


图 1-32 使用“first_match”构造的 SVA 检验器的波形

1.26 “throughout” 构造

蕴含(implication)是目前讨论到的允许定义前提条件的一项技术。例如，要对一个指定的序列进行检验，必须某个前提条件为真。也有这样的情况，要求在检验序列的整个过程中，某个条件必须一直为真。蕴含只在时钟边沿检验前提条件一次，然后就开始检验后续算子部分，因此它不检测先行算子是否一直保持为真。为了保证某些条件在整个序列的验证过程中一直为真，可以使用“throughout”运算符。运算符“throughout”的基本语法如下所示：

```
(expression) throughout (sequence definition)
```

属性 p31 检查下列内容：

- 在信号“start”的下降沿开始检查。
- 检查表达式($\neg a \& \neg b$) $\#\#1 (c[->3]) \#\#1 (a \& b)$)。
- 序列检查在信号“a”和“b”的下降沿与信号“a”和“b”的上升沿之间，信号“c”应该连续或间断地出现 3 次为高电平。
- 在整个检验过程中，信号“start”保持为低。

```
property p31;
  @(posedge clk) $fell(start) |->
    (!start) throughout
      (#1 (!a && !b) ##1 (c[->3]) ##1 (a & b));
endproperty

a31: assert property (p31);
```

图 1-33 显示了属性 p31 在模拟中的响应。检验在时钟周期 3 成功，在时钟周期 16 失败。

成功 1——信号“start”在时钟周期 3 被检测到一个下降沿，因此属性的先行算子成功。一个周期后，信号“a”和信号“b”如期在时钟周期 4 为低。之后，信号“c”如期望地分别在时钟周期 6, 9, 11 重复三次。接着在时钟周期 12，信号“a”和信号“b”如期为高。因此属性从时钟周期 3 开始，在时钟周期 12 成功。注

意到，信号“start”从时钟周期 3 一直到时钟周期 12 保持低电平。这是本次检验成功的关键。

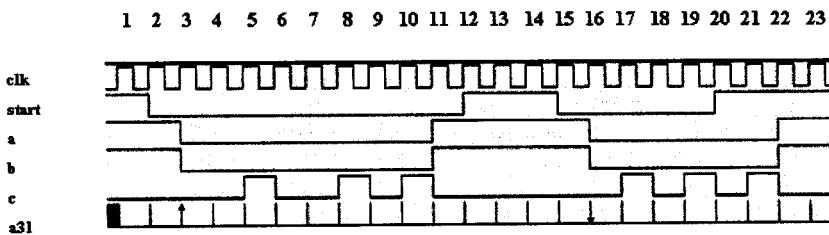


图 1-33 使用“throughout”构造的 SVA 检验器的波形

失败 1——信号“start”在时钟周期 16 检测到一个下降沿，因此属性的先行算子成功。一个周期后，信号“a”和信号“b”如期在时钟周期 17 为低。之后，期望信号“c”重复为高三次。我们分别在时钟周期 18 和 20 发现两次重复。但是在时钟周期 21，信号“c”的第三次重复还没出现，信号“start”就被检测出高电平，因此检验在时钟周期 21 失败。违背了“throughout”的条件导致了整个检验的失败。

1.27 “within” 构造

“within”构造允许在一个序列中定义另一个序列。

```
seq1 within seq2
```

这表示 seq1 在 seq2 的开始到结束的范围内发生，且序列 seq2 的开始匹配点必须在 seq1 的开始匹配点之前发生，序列 seq1 的结束匹配点必须在 seq2 的结束匹配点之前结束。属性 p32 检查序列 s32a 在信号“start”的上升沿和下降沿之间发生。信号“start”的上升和下降由序列 s32b 定义。

```
sequence s32a;
@ (posedge clk)
  ((!a&&!b) ##1 (c[>3]) ##1 (a&b));
endsequence
```

```

sequence s32b;
  @ (posedge clk)
    $fell (start) ##[5:10] $rose (start);
endsequence

sequence s32;
  @ (posedge clk) s32a within s32b;
endsequence

property p32;
  @ (posedge clk) $fell (start) |-> s32;
endproperty

a32: assert property (p32);

```

图 1-34 使用了与 throughout 运算符用的例子相同的设计条件来显示属性 p32 在模拟中的响应。检验有两个有效的开始：一个在时钟周期 3，另一个在时钟周期 16。在这两个点，检测到信号“start”的下降沿。

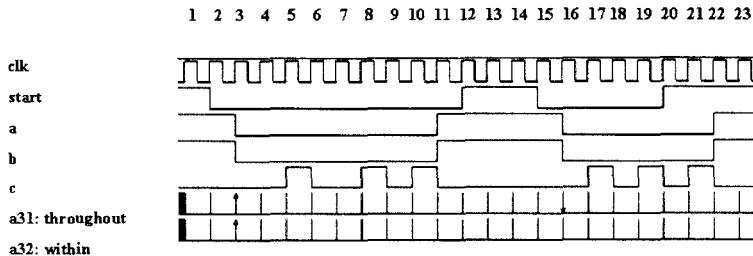


图 1-34 使用“within”构造的 SVA 检验器的波形

成功 1——从时钟周期 3 开始的检验成功了。信号“start”的下降沿在时钟周期 3，上升沿在时钟周期 13。在这两个时钟周期间，信号“c”分别在时钟周期 6, 9, 11 被检测到三次高电平。因此检验成功。

未完成 1——从时钟周期 16 开始的检验未能完成。信号“start”的下降沿在时钟周期 16，上升沿在时钟周期 21。在这两个时钟周期间，信号“c”分别在时钟周期 18 和 20 被检测到两次高电平。信号“c”的第三次重复出现在时钟周期 22，但是在时钟周期 21

检测到信号“start”为高。这是一个失败，但是由于信号“c”使用的是跟随重复(“go to” repetition)运算符，它按照阻塞序列的规则来执行。这使得检查失败并且在模拟中发出一个未完成的信息。

1.28 内建的系统函数

SVA 提供了几个内建的函数来检查一些最常用的设计条件。

\$onehot(expression)——检验表达式满足“one-hot”，换句话说，就是在任意给定的时钟沿，表达式只有一位为高。

\$onehot0(expression)——检验表达式满足“zero one-hot”，换句话说，就是在任意给定的时钟沿，表达式只有一位为高或者没有任何位为高。

\$isunknown(expression)——检验表达式的任何位是否是 X 或者 Z。

\$countones(expression)——计算向量中为高的位的数量。

断言语句 a33a 检验向量“state”是“one-hot”。断言语句 a33b 检验向量“state”是“zero one-hot”，断言语句 a33c 检验向量“bus”是否有任何位为 X 或 Z。断言语句 a33d 检验向量“bus”中等于 1 的位的个数大于 1。

```
a33a: assert
      property (@(posedge clk) $onehot(state));
a33b: assert
      property (@(posedge clk) $onehot0(state));
a33c: assert
      property (@(posedge clk) $isunknown(bus));
a33d: assert
      property (@(posedge clk) $countones(bus)>1);
```

图 1-35 显示了上述断言在模拟中的响应。表 1-16 总结了每个断言的状态和向量“state”和“bus”的采样值。注意，断言 a33a

在时钟周期 2 失败，因为所有位都为零。“one-hot”要求在任何时钟上升沿都只有一位为高。另一方面，断言 a33b 成功因为它检查“zero one-hot”，而对于这种构造，所有位都为零是合法的。a33a 和 a33b 都在时钟周期 5, 6, 7, 8 失败，因为有超过一位为高。断言 a33c 在任何时候向量“bus”的值不为 X 或 Z 时失败。它在时钟周期 5, 6, 7 成功，因为向量的值为 Z。断言 a33d 在时钟周期 2, 3, 5, 6, 7 失败，因为值为高的位的个数没超过 1。断言 a33d 在时钟周期 4, 8 成功，因为向量“bus”在这两个时刻都有两位为高。

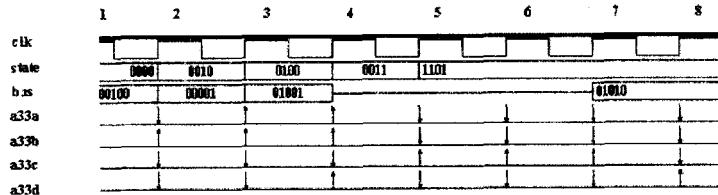


图 1-35 使用内建的系统函数的 SVA 检验器波形

表 1-16 使用内建函数的 SVA 检验器的真值表

时钟数	“state”的		“bus”的		a33a - \$onehot	a33b - \$onehot0	a33c - \$isunknown	a33d - \$countones
	采样值	采样值	状态	状态				
2	0000	00100	失败	成功	失败	失败	失败	失败
3	0010	00001	成功	成功	失败	失败	失败	失败
4	0100	01001	成功	成功	失败	成功	失败	成功
5	0011	Z	失败	失败	成功	失败	失败	失败
6	1101	Z	失败	失败	成功	失败	失败	失败
7	1101	Z	失败	失败	成功	失败	失败	失败
8	1101	01010	失败	失败	失败	失败	失败	成功

1.29 “disable iff” 构造

在某些设计情况中，如果一些条件为真，则我们不想执行检验。换句话说，这就像是一个异步的复位，使得检验在当前时刻不工作。SVA 提供了关键词“**disable iff**”来实现这种检验器的异步复位。“**disable iff**”的基本语法如下。

```
 disable iff (expression) < property definition>
```

属性 p34 检查在有效开始后，信号“a”重复两次，且 1 个周期之后，信号“b”重复两次，再过一个时钟周期，信号“start”为低。在整个序列过程中，如果“reset”被检测为高，检验器会停止并默认地发出一个空成功的信号。

```
property p34;
  @ (posedge clk)
  disable iff (reset)
    $rose(start) |=> a[=2] ##1 b[=2] ##1 !start ;
endproperty

a34: assert property(p34);
```

图 1-36 显示了属性 p34 在模拟中的响应。标记 1s 标出了一个有效的开始，在有效开始后，信号“a”重复为高两次，接着信号“b”重复为高两次，然后信号“start”如期望的为低。

在整个序列的过程中，信号“reset”如期望的始终不被激活，因此检验在标记 1e 处成功。第二个有效开始由标记 2s 标出。在有效开始后，信号“a”重复为高两次，接着复位信号“reset”在信号“b”重复两次之前被激活。这使得检查失效，属性得到一个空成功。

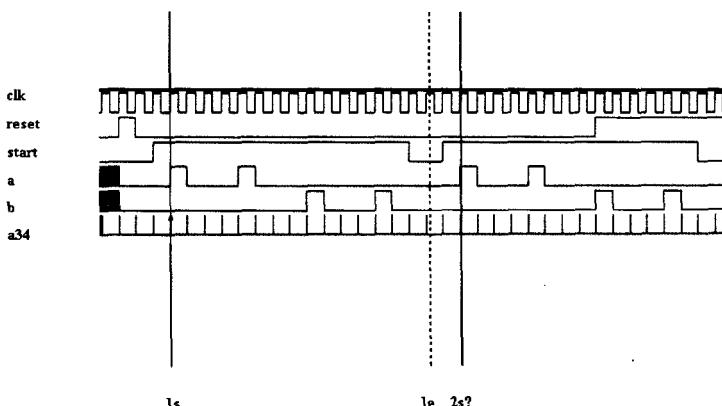


图 1-36 使用“`disable iff`”构造的 SVA 检验器的波形

1.30 使用“`intersect`”控制序列的长度

在 1.23 节讨论的“`intersect`”运算符可以有效地控制序列的长度，尤其是在时序窗口未定义上界的情况下。每当使用一个可能性(eventuality)运算符时，检验器成功所需的时钟周期数没有限制。运算符 `intersect` 提供了一个定义可能性运算符可以使用的最小和最大时钟周期数的机制。

属性 `p35` 定义了一个序列来检验在给定时钟边沿，如果信号“`a`”为高，那么从下一个时钟周期开始信号“`b`”最终将为高，接着在下一个时钟周期开始信号“`c`”最终也会为高。这个序列每当信号“`a`”为高时就开始，并且可能一直到整个模拟结束时才成功。这可以使用带 `1[*2:5]` 的 `intersect` 运算符来加以约束。这个 `intersect` 的定义检查从序列的有效开始点(信号“`a`”为高)，到序列成功的结束点(信号“`c`”为高)，一共经过 2~5 个时钟周期。

```

property p35;
  (@(posedge clk) 1[*2:5] intersect
    (a ##[1:$] b ##[1:$] c));
endproperty

a35: assert property(p35);
  
```

图 1-37 显示了属性 p35 在模拟中的响应。表 1-17 总结了断言 a35 的状态和相关信号的采样值。在一个给定的时钟边沿，如果信号“a”未被检测为高，那么这是一个失败。这种情况发生在时钟周期 1, 3, 4, 5, 11 和 13，这些时刻没有有效开始。

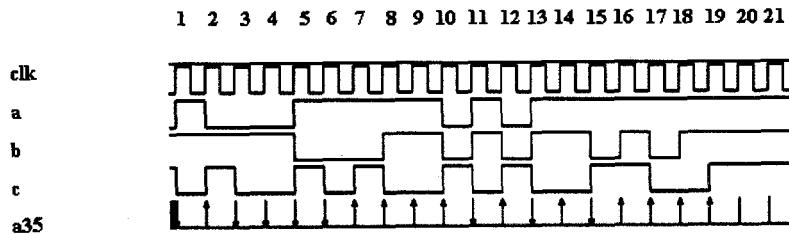


图 1-37 使用 intersect 来控制序列长度的 SVA 检验器的波形

检验在时钟周期 2, 7, 8, 9, 10, 12 和 14 成功。可以看到序列从开始到结束至多花了 5 个时钟周期。检验在时钟周期 6 有一个真正的失败。在时钟周期 6 检测到信号“a”为高，而且在时钟周期 9 信号“b”为高。但是在整个检查达到允许的最大长度，即时钟周期 10，信号“c”依然未能为高，因此检验在时钟周期 10 失败。可以看到信号“c”在时钟周期 11 为高，但是这已经太晚了。

表 1-17 使用 intersect 构造来控制序列长度的 SVA 检验器的真值表

时钟 数	"a"的 采样值	"b"的 采样值	"c"的 采样值	有效	a35 的状态
				开始	
1	0	1	1	否	失败
2	1	1	0	是	成功 (开始于 2, 结束于 6)
3	0	1	1	否	失败
4	0	1	0	否	失败
5	0	1	0	否	失败
6	1	0	1	是	失败 (开始于 6, 结束于 10)
7	1	0	0	是	成功 (开始于 7, 结束于 11)
8	1	0	1	是	成功 (开始于 8, 结束于 11)
9	1	1	0	是	成功 (开始于 9, 结束于 11)
10	1	1	0	是	成功 (开始于 10, 结束于 13)

(续表)

时钟数	"a"的采样值	"b"的采样值	"c"的采样值	有效开始	a35 的状态
11	0	0	1	否	失败
12	1	1	0	是	成功 (开始于 12, 结束于 16)
13	0	0	1	否	失败
14	1	1	0	是	成功 (开始于 14, 结束于 16)
15	1	1	0	是	失败
16	1	0	1	是	成功
17	1	1	1	是	成功

1.31 在属性中使用形参

可以用定义形参(formal arguments)的方式来重用一些常用的属性。属性“arb”使用了 4 个形参，并且根据这些形参进行检验。其中还定义了特定的时钟。SVA 允许使用属性的形参来定义时钟。这样，属性可以应用在使用不同时钟的相似设计模块中。同样的，时序延迟也可以参数化，这使得属性的定义更具有普遍性。

属性首先检查有效开始。在给定的时钟上升沿，如果在信号“a”的下降沿后的 2~5 个时钟周期内出现信号“b”的下降沿，那么这就是一个有效开始。如果先行算子匹配，那么属性接着检查信号“c”和信号“d”的下降沿在下一个时钟周期出现，并且确保它们在 4 个连续的周期都保持为低。接着一个周期后，必须检测到信号“c”和信号“d”都为高，且再过一个时钟周期应该检测到信号“b”为高。

假定这是处理三个具有相似信号的不同主控设备的仲裁器的协议，可以很容易地重用前面定义的属性来检验所有 3 个主控设备的接口。断言 a36_1, a36_2 和 a36_3 定义了每个主控接口用的检验器，分别使用了各个接口对应的信号作为属性 arb 的参数。

```
property arb (a, b, c, d);
  @ (posedge clk) ($fell(a) ##[2:5] $fell(b)) | ->
```

```

##1 ($fell(c) && $fell(d)) ##0
(!c&&!d) [*4] ##1 (c&d) ##1 b;
endproperty

a36_1: assert property(arb(a1, b1, c1, d1));
a36_2: assert property(arb(a2, b2, c2, d2));
a36_3: assert property(arb(a3, b3, c3, d3));

```

图 1-38 显示了每个接口对应的断言在模拟过程中的响应。断言 a36_1 有一个有效开始，并且检验成功。断言 a36_3 有一个有效开始但是检验失败了。

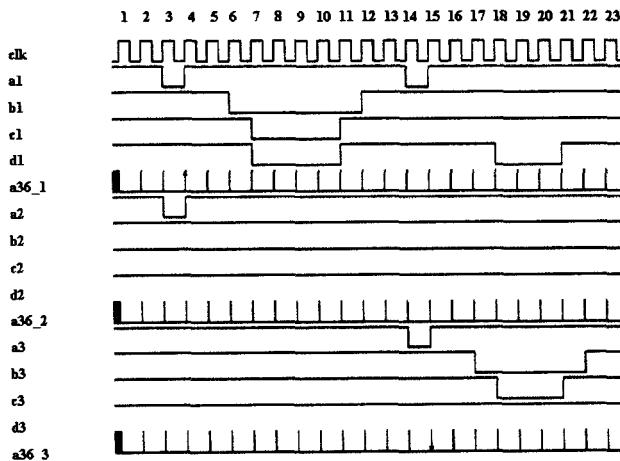


图 1-38 在属性中使用形参的 SVA 检验器的波形

成功 1(a36_1)——在时钟周期 4，当信号“a1”的下降沿出现时，检验开始。期望在 2~5 个时钟周期内信号“b1”出现一个下降沿，下降沿在时钟周期 7 如期出现。在下一个时钟周期，信号“c1”和信号“d1”如期望的为低，并且必须保持为低 4 个时钟周期。它们在时钟周期 8~11 都为低。接着在时钟周期 12，信号“c1”和“d1”如期望的为高。然后在时钟周期 13，信号“b1”如期望地为高。因此检验开始于时钟周期 4，成功于时钟周期 13。

失败 1(a36_3)——在时钟周期 15，当信号“a3”的下降沿出现时，检验开始。期望在 2~5 个时钟周期内信号“b3”出现一个下降沿，下降沿如期在时钟周期 18 出现。在下一个时钟周期，信

号“c3”和“d3”应该为低。由于未检测到信号“d3”为低，检验失败于时钟周期 19。

1.32 嵌套的蕴含

SVA 允许使用嵌套的蕴含。当我们有多个门限条件指向一个最终的后续算子时，这种构造十分有用。

属性 p_nest 检验如果信号“a”有一个下降沿，则是一个有效开始，接着在一个周期后，信号“b”，“c”和“d”应该为低电平有效信号以保持这个有效开始。如果第二个条件匹配，那么在 6 到 10 个周期内期望“free”为真。注意，当且仅当信号“b”，“c”和“d”都匹配时，在后续状况(consequent condition)“free”才会被检验是否为真。

```
`define free (a && b && c && d)

property p_nest;
    @(posedge clk) $fell(a) |->
        ##1 (!b && !c && !d) |->
        ##[6: 10]
        `free;
endproperty

a_nest: assert property(p_nest);
```

同一个属性可以被重写成不使用嵌套蕴含的方式，如下所示。

```
property p_nest1;
    @(posedge clk) $fell(a) ##1 (!b && !c && !d)
                           |-> ##[6:10] `free;
endproperty

a_nest1: assert property(p_nest1);
```

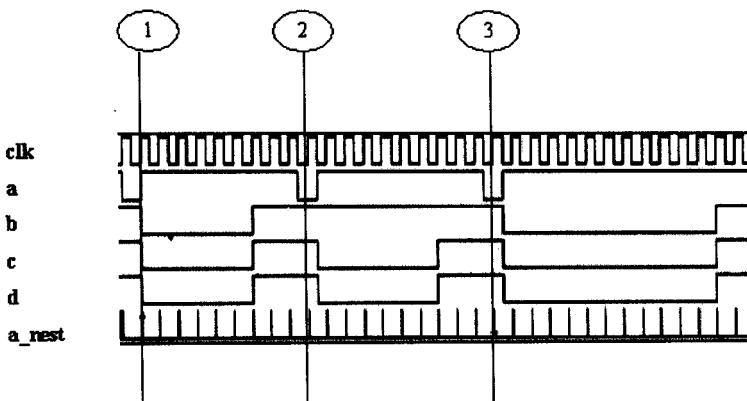


图 1-39 使用嵌套蕴含的 SVA 检验器

注意：

使用嵌套蕴含的属性 `p_nest` 中没有“else”情况，因此属性很容易就能重写成如 `p_nest1` 所示的形式。

图 1-39 显示了断言 `a_nest` 在模拟中的表现。标记 1 显示了检验器的第一次成功。当检测到信号“a”的下降沿时，出现一个有效开始。一个时钟周期后，信号“b”、“c”和“d”如预期地被检测为低。因此检验保持有效，后续算子被检验。在 6 个时钟周期后，检测到状况“free”为真，因此检验成功。

第二个标记指出了下一个有效开始，在此检测到信号“a”的下降沿。一个周期后，检测到信号“c”和“b”为低，但是信号“b”不为低。因此检验没能保持有效，得到一个空成功。

第三个标记也标出了一个有效开始，在此检测到信号“a”的下降沿。一个周期后，如期望的检测到信号“b”，“c”和“d”为低，因此检验保持有效，后续算子被检验。在 6~10 个时钟周期内，没有检验到状况“free”为真，因此检验失败。

1.33 在蕴含中使用 if/else

SVA 允许在使用蕴含的属性的后续算子中使用“if/else”语句。

属性 `p_if_else` 检查如果信号“start”的下降沿被检测到，就是一个有效开始，接着一个时钟周期后，信号“a”或者信号“b”为高。在现行算子成功匹配时，后续算子有两个可能的路径。

1. 如果信号“a”为高，那么信号“c”必须间歇地重复两次为高，且在下一个时钟周期，信号“e”必须为高。
2. 如果信号“a”不为高，那么信号“d”必须间歇地重复两次为高，且在下一个时钟周期，信号“f”必须为高。

注意，在检验信号“a”的后续算子中有优先级。

```
property p_if_else;
  @ (posedge clk)
  ($fell(start) ##1 (a||b)) |->
    if (a)
      (c[~>2] ##1 e)
    else
      (d[~>2] ##1 f);
endproperty

a_if_else: assert property(p_if_else);
```

如果不用“if/else”构造来重写这个属性，需要用三个单独的属性来实现。由于“if/else”有优先级，两个信号导致了如下所示的三种不同的可能结果：

a	b	分支 (Leaf)
1	0	a
0	1	b
1	1	a

可以发现，如果信号“a”和“b”都为高，那么会执行信号“a”的“if”块，因为它的优先级高。重写的三个属性如下所示。

```
property p_if_else_leaf1;
  @ (posedge clk)
  ($fell(start) ##1 a) |->
    (c[~>2] ##1 e);

endproperty
```

```

a_if_else_leaf1:
    assert property(p_if_else_leaf1);

property p_if_else_leaf2;
@(posedge clk)
($fell(start) ##1 b) |->
(d[>2] ##1 f);
endproperty

a_if_else_leaf2:
    assert property(p_if_else_leaf2);

property p_if_else_leaf3;
@(posedge clk)
($fell(start) ##1 (a &&b)) |->
(c[>2] ##1 e);

endproperty

a_if_else_leaf3:
    assert property(p_if_else_leaf3);

```

1.34 SVA 中的多时钟定义

SVA 允许序列或者属性使用多个时钟定义来采样独立的信号或者子序列。SVA 会自动地同步不同信号或子序列使用的时钟域。下面的代码显示了一个序列使用多个时钟的简单例子。

```

sequence s_multiple_clocks;
    @(posedge clk1) a ##1 @(posedge clk2) b;
endsequence

```

序列 `s_multiple_clocks` 检验在时钟 “`clk1`” 的任何上升沿，信号 “`a`” 为高，接着在时钟 “`clk2`” 的上升沿，信号 “`b`” 为高。当信号 “`a`” 在时钟 “`clk1`” 的任意给定上升沿为高时，序列开始匹配。接着 “`##1`” 延迟构造将检验时间移到时钟 “`clk2`” 的最近的

上升沿，检查信号“b”是否为高。当在一个序列中使用了多个时钟信号时，只允许使用“##1”延迟构造。序列 s_multiple_clocks 不能被重写成下面这种形式。

```
sequence s_multiple_clocks_illegal1;
    @ (posedge clk1) a ##0 @ (posedge clk2) b;
endsequence

sequence s_multiple_clocks_illegal2;
    @ (posedge clk1) a ##2 @ (posedge clk2) b;
endsequence
```

使用“##0”会产生混淆，即在信号“a”匹配后究竟哪个时钟信号才是最近的时钟。这将引起竞争，因此不允许使用。使用##2也不允许，因为不可能同步到时钟“clk2”的最近的上升沿。

相似的技术可以用来建立具有多个时钟的属性。如下面的例子所示：

```
property p_multiple_clocks;
    @ (posedge clk1) s1 ##1 @ (posedge clk2) s2;
endproperty
```

它假定序列 s1 没有被时钟驱动，或者它的时钟定义和“clk1”一样。它又假定序列 s2 没有被时钟驱动，或者它的时钟定义和“clk2”一样。同样的，属性可以在序列定义之间使用非交叠蕴含运算符。下面是一个简单的例子：

```
property p_multiple_clocks_implied;
    @ (posedge clk1) s1 |=> @ (posedge clk2) s2;
endproperty
```

禁止在两个不同时钟驱动的序列之间使用交叠蕴含运算符。因为先行算子的结束和后续算子的开始重叠，可能引起竞争的情况，这是非法的。下面的代码显示了这种非法的编码方式：

```
property p_multiple_clocks_implied_illegal;
    @ (posedge clk1) s1 |-> @ (posedge clk2) s2;
endproperty
```

1.35 “matched” 构造

任何时候如果一个序列定义了多个时钟，构造“**matched**”可以用来监测第一个子序列的结束点。序列 s_a 查找信号“a”的上升沿。而信号“a”是根据时钟“clk1”来采样的。序列 s_b 查找信号“b”的上升沿。信号“b”则是根据时钟“clk2”来采样的。属性 p_match 验证在给定的时钟“clk2”的上升沿，如果序列 s_a 匹配，那么在一个周期后，序列 s_b 也必须为真。

```
sequence s_a;
  @(posedge clk1) $rose(a);
endsequence

sequence s_b;
  @(posedge clk2) $rose(b);
endsequence

property p_match;
  @(posedge clk2) s_a.matched |> s_b;
endproperty

a_match: assert property(p_match);
```

图 1-40 显示了断言 a_match 在模拟中的表现。属性在序列 s_a 匹配的时候得到有效开始。注意，虽然序列 s_a 是根据时钟“clk1”来采样的，但我们只在时钟“clk2”的每个上升沿查找这种匹配。

在“clk1”的时钟周期 3，信号“a”有一次有效的上升。这将更新序列 s_a 的匹配值为真。这个值一直被保持到“clk2”的最近的时钟上升沿，也就是“clk2”的时钟周期 2。在这一个时间点，属性被激活，并且在“clk2”的下一个时钟周期，期望序列 s_b 匹配。因此属性的第一次成功开始于“clk2”的时钟周期 2，结束于“clk2”的时钟周期 3。

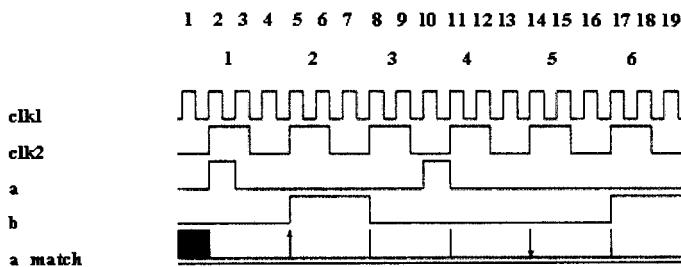


图 1-40 使用“matched”构造的 SVA 检验器

信号“a”的另一个有效上升发生在信号“clk1”的时钟周期 11，并且在“clk2”的时钟周期 5 被属性采样到。属性在这个点被激活，并且期望在“clk2”的时钟周期 6，序列 s_b 匹配。但是这次，信号“b”的上升沿没能出现，因此属性失败。理解“matched”构造的使用方法的关键在于，被采样到的匹配的值一直被保存到另一个序列最近的下一个时钟边沿。

1.36 “expect” 构造

SVA 支持一种叫“expect”的构造，它与 Verilog 中的等待构造相似，关键的区别在于 expect 语句等待的是属性的成功检验。expect 构造后面的代码是作为一个阻塞的语句来执行。expect 构造的语法与 assert 构造很相似。expect 语句允许在一个属性成功或者失败后使用一个执行块(action block)。使用 expect 构造的实例如下所示。

```

initial

begin
  @(posedge clk);
  #2ns cpu_ready = 1'b1;
  expect(@(posedge clk) ##[1: 16]
          memory_ready == 1'b1)
    $display("Hand shake successful\n");
  else
    begin

```

```

$display("Hand shake failed: exiting\n")
$finish();
end

for(i=0; i<64; i++)
begin
    send_packet();
    $display("PACKET %0d sent\n", i);
end

end

```

注意，在信号“cpu_ready”被断言后，expect语句等待信号“memory_ready”在1~16个周期内的任意周期被断言。如果信号“memory_ready”如预期的被断言，那么显示一个成功信息，并且开始执行“for”循环代码。如果信号“memory_ready”没能如预期的被断言，那么显示错误信息，且模拟结束。

1.37 使用局部变量的SVA

在序列或者属性的内部可以局部定义变量，而且可以对这种变量进行赋值。变量接着子序列放置，用逗号隔开。如果子序列匹配，那么变量赋值语句执行。每次序列被尝试匹配时，会产生变量的一个新的备份。

```

property p_local_var1;
int lvar1;
@(posedge clk)
($rose(enable1), lvar1 = a) |->
    ##4 (aa == (lvar1*lvar1*lvar1));
endproperty

a_local_var1: assert property(p_local_var1);

```

属性p_local_var1查找信号“enable1”的上升沿。如果找到，

局部变量“lvar1”保存设计中向量“a”的值。在4个周期后，检查设计的输出向量“aa”是否与局部变量的值的立方相等。属性的后续算子等待设计满足延迟(4个时钟周期)，然后将设计的实际输出和属性局部计算的值比较。图1-41显示了检验在模拟中的响应。

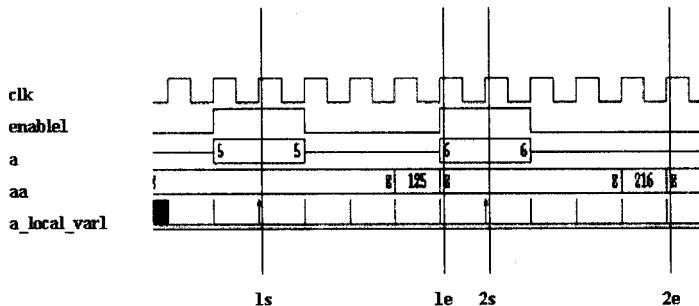


图1-41 使用局部变量的SVA的波形

标记1s显示了信号“enable1”的上升沿被采样到的点，在这个点，向量“a”的值为5，被保存在局部变量“lvar1”中。标记1e标出了输出被采样的点，它在输入值被保存的4个时钟周期之后。在标记1e的点，因为输出值(125)与局部变量“lvar1”的值的立方相等，断言成功。类似地，标记2s显示了下一个输入数据被保存的时刻，标记2e标出了输出被采样并且与局部变量“lvar1”的立方值比较的时间点。

可以在SVA中保存和操作局部变量。

```

property p_lvar_accum;
  int lvar;
  @(posedge clk) $rose(start) |=>
    (enable1 ##2 enable2, lvar = lvar +aa) [*4]
    ##1 (stop && (aout == lvar));
  endproperty

  a_lvar_accum : assert property(p_lvar_accum);

```

属性p_lvar_accum检查下列内容：

- (1) 在任意给定的时钟上升沿，如果检测到信号“start”的上升沿，标志一个有效开始。

- (2) 在一个周期后，寻找一个特定的模型或者子序列。信号“enable1”必须被检测为高，且两个周期后，“enable2”应该被检测为高。这个子序列必须连续重复 4 次。
- (3) 在子序列的每次重复中，向量“aa”的值在序列内部被累加。在重复结束时，局部变量保存着向量“aa”累加 4 次的值。
- (4) 在重复结束的下一个时钟周期，期望信号“stop”为高，且局部变量保存的值与输出向量“aout”的值相等。

图 1-42 显示了检验在模拟中的响应。

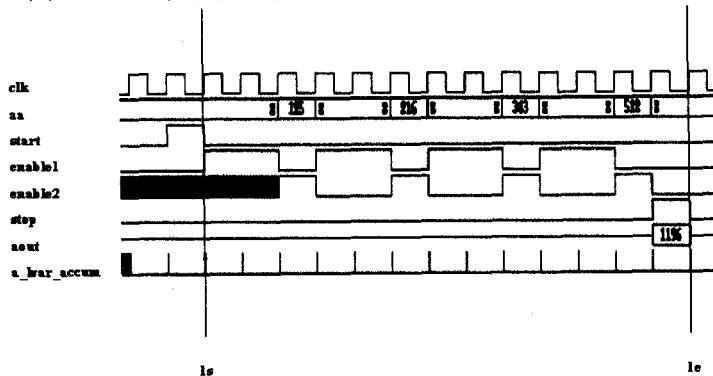


图 1-42 使用局部变量赋值的 SVA

标记 1s 显示了当信号“start”被检测为高时所产生的一个有效开始。标记 1e 显示了检验的结束点。信号“enable*”成功地重复 4 次并且在一个时钟周期后，信号“stop”如期望的被检测为高。局部变量保存的值与输出向量“aout”值相同，因此检验在标记 1e 处成功。

1.38 在序列匹配时调用子程序

SVA 可以在序列每次成功匹配时调用子程序。同一序列中定义的局部变量可以作为参数传给这些子程序。对于序列的每次匹配，子程序调用的执行与它们在序列定义中的顺序相同。

```

sequence s_display1;
@(posedge clk)
($rose(a), $display("Signal a arrived at %t\n",
$time));
endsequence

sequence s_display2;
@(posedge clk)
($rose(b), $display("Signal b arrived at %t\n",
$time));
endsequence

property p_display_window;
@(posedge clk)
s_display1 |-> ##[2: 5] s_display2;
endproperty

a_display_window :
assert property(p_display_window);

```

序列 s_display1 查找信号“a”的上升沿。如果匹配，就执行 display 语句。序列 s_display2 对信号“b”作类似的检查。属性 p_display_window 检验如果序列 s_display1 出现，那么序列 s_display2 必须在 2~5 个时钟周期之间的某个时刻出现。使用 display 语句，用户可以得到精确的信息，了解后续序列经过多少个时钟周期完成。图 1-43 显示了检验在模拟中的响应。

标记 1s 显示了由于检测到信号“a”的上升沿而得到的一个检验器的有效开始。在这一点，SVA 执行序列 s_display1 的 display 语句。标记 1e 显示了信号“b”出现上升沿的点。因为它出现在 3 个时钟周期后，所以检验成功。在这个点上，执行序列 s_display2 的 display 语句。

标记 2s 显示了由于检测到信号“a”的上升沿而得到的检验器的另一个有效开始。在这一点，SVA 执行序列 s_display1 的 display 语句。标记 2e 显示了检验器的结束点。信号“b”的有效上升沿没能在 2~5 个时钟周期内出现，因此检验失败。由于第二个序列没有匹配，序列相关的 display 语句没有执行。SVA 发出一

个默认的出错信息。

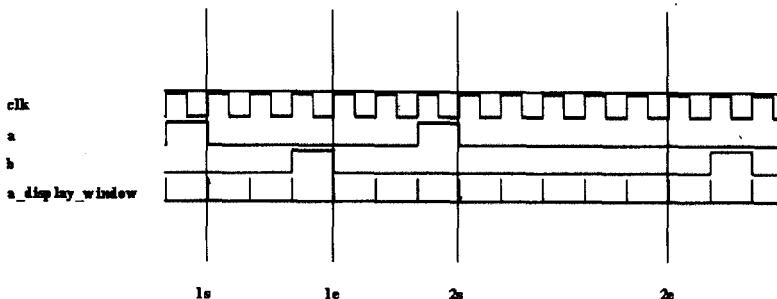


图 1-43 在序列匹配时使用子程序的 SVA

一个模拟日志的实例如下所示。

```

Signal a arrived at      125
Signal b arrived at      275
"sub.v", 45: sub.a_display_window:
started at 125s succeeded at 275s

Signal a arrived at      425
"sub.v", 45: sub.a_display_window:
started at 425s failed at 675s
Offending '$rose(b)'

```

1.39 将 SVA 与设计连接

有两种方法可以将 SVA 检验器连接到设计中。

(1) 在模块(module)定义中内建或者内联检验器。

(2) 将检验器与模块、模块的实例或者一个模块的多个实例绑定。

有的工程师不喜欢在设计中加任何验证代码。在这种情况下，在外部绑定 SVA 检验器是很好的选择。SVA 代码可以内建在模块定义中的任何地方。下面的例子显示了内联在模块中的 SVA。

```

module inline(clk, a, b, d1, d2, d);

input logic clk, a, b;
input logic [7: 0] d1, d2;
output logic [7: 0] d;

always@(posedge clk)
begin
    if(a)
        d <= d1;
    if(b)
        d <= d2;
end

property p_mutex;
    @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule

```

如果用户决定将 SVA 检验器与设计代码分离，那么就需要建立一个独立的检验器模块。定义独立的检验器模块，增强了检验器的可重用性。下面所示的是一个检验器模块的代码实例。

```

module mutex_chk(a, b, clk);

input logic a, b, clk;
property p_mutex;
    @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule

```

注意，定义检验器模块时，它是一个独立的实体。检验器用来检验一组通用的信号，检验器可以与设计中任何的模块(module)或者实例(instance)绑定，绑定的语法如下所示。

```
bind <module_name or instance name>
    <checker name> <checker instance name>
        <design signals>;
```

在上面的检验器例子中，绑定可以用下面的方式实现。

```
bind inline mutex_chk i2 (a, b, clk);
```

在实现绑定时，使用的是设计中的实际信号。

比如，我们有一个如下所示的顶层模块。

```
module top(..);
    inline u1 (clk, a, b, in1, in2, out1);
    inline u2 (clk, c, d, in3, in4, out2);

endmodule
```

检验器 `mutex_chk` 可以用下面的方式与顶层模块中内联 (inline) 的两个模块实例绑定。

```
bind top.u1 mutex_chk i1(a, b, clk);
bind top.u2 mutex_chk i2(c, d, clk);
```

与检验器绑定的设计信号可以包含绑定实例中的任何信号的跨模块引用(cross module reference)。

1.40 SVA 与功能覆盖

功能覆盖是按照设计规范衡量验证状态的一个标准，它可以分成两类。

- a. 协议覆盖。
- b. 测试计划覆盖。

断言可以用来获得有关协议覆盖的穷举信息。SVA 提供了关键词 “`cover`” 来实现这一功能，`cover` 语句的基本语法如下所示。

```
<cover_name> : cover property(property_name)
```

“`cover_name`” 是用户提供的名称，用来标明覆盖语句，

“`property_name`”是用户想获得覆盖信息的属性名。例如，在1.39节定义的检验器“`mutex_chk`”，可以如下所示来检查它的覆盖情况。

```
c_mutex: cover property(p_mutex);
```

`cover`语句的结果包含下面的信息：

- (1) 属性被尝试检验的次数。
- (2) 属性成功的次数。
- (3) 属性失败的次数。
- (4) 属性空成功的次数。

检验器“`mutex_chk`”在一次模拟中的覆盖日志的实例如下所示。

```
c_mutex, 12 attempts, 12 match, 0 vacuous match
```

就像断言(`assert`)语句一样，覆盖(`cover`)语句可以有执行块。在一个覆盖成功匹配时，可以调用一个函数(`function`)或者任务(`task`)，或者更新一个局部变量。



SVA 模拟方法论

在第 1 章中，通过举例详细介绍了 SVA 语言的结构。所有例子都阐明了两个或更多的通用信号间的关系，而没有涉及任何实际设计的详情。在第 2 章中，用一个虚拟的系统表示实际设计的情况。本章将逐步讨论协议的析取和断言的开发过程，并讨论多种能显著提高基于断言的验证(ABV)生产率的模拟方法，本章还将详细讨论功能覆盖和互动测试平台的开发。

2.1 一个被验证的实例系统

这个被验证的实例系统如图 2-1 所示。它有 3 个主控设备(master device)和 2 个目标设备(target device)。主控设备和目标设备之间通过一个中间设备(mediator)相连。在同一时刻，只能有一个主控设备与一个目标设备交互。任何主控设备都可以和任何一个目标设备交互。事务可以是一个读操作或写操作。中间设备有一个仲裁逻辑来决定哪个主控设备管理事务，仲裁器采用一种简单的循环技术。中间设备还包含一个胶合逻辑(glue logic)，实际上将主控设备的信息解码提供给目标设备，反之亦然。胶合逻辑有助于建立一个指定的主控设备和目标设备之间的连接，及成功管理这个事务。

2.1.1 主控设备

图 2-2 显示了有输入和输出端口的主控设备的方框图。主控设备可以执行读和写操作。它支持在单个系统中控制两个目标设备。当主控设备收到指令“ask_for_it”(取信息)，就准备开始一个事务，发出一个低电平有效的脉冲到信号“req”上，然后等待一个“gnt”信号。“gnt”信号是一个低电平有效信号。如果在 2~5 个时钟周期内没有等到“gnt”信号，稍后主控设备将重试这个过程。如果在 2~5 个时钟周期内获得了“gnt”信号，主控设备将立即断言(assert)信号“frame”和“irdy”，用来确认“gnt”信号的到达(“frame”和“irdy”信号是低电平有效信号)。在同一个时钟周期，主控设备也要选择与它建立事务的目标设备。主控设备用输出信号“rsel”来表示，如果信号“rsel”设为 1，则主控设备要与目标设备 1 建立事务，如果信号“rsel”设为 0，则主控设备要与目标设备 0 建立事务。

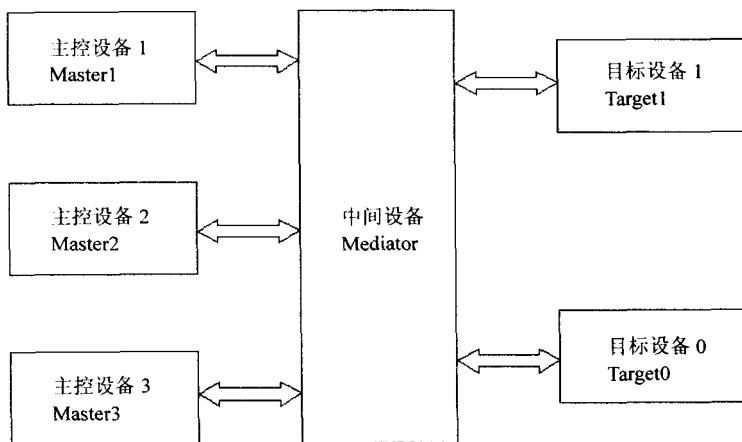


图 2-1 一个实例系统

一旦“rsel”信号被更新，目标设备需要向主控设备确认自己响应了选中。目标设备用信号“trdy”表明它已准备就绪。如果在“rsel”被更新后的 3 个时钟周期内目标设备没有确认它自身，这就是出错的情况。如果目标设备确认了它自身，那么主控设备要

决定是否读或写。主控设备通过“datac”总线发送数据和指令来读或写数据。

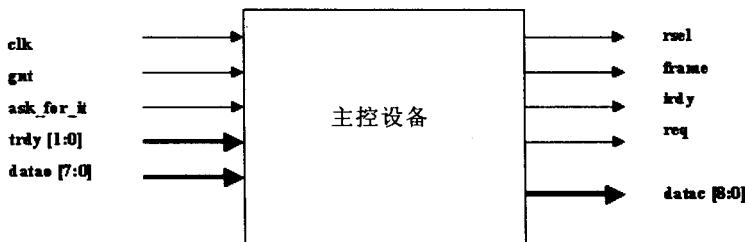


图 2-2 主控设备实例

最高位是指令位(如波形图中的信号“rw”所示),如果指令位为1,则主控设备将执行写操作。如果它为0,则主控设备执行读操作。如果要执行一个写操作,最低的8位数据被写到目标设备。如果要执行一个读操作,则从目标设备读入的数据将会出现在输入总线“datao”上。主控设备的每次事务将会精确地持续8个时钟周期。换言之,在一个事务中主控设备要么读8个字节要么写8个字节。由于没有指定的地址生成方案,主控设备将会写到目标设备中最新更新的写指针所指的地址。同样地,主控设备会从目标设备中最新更新的读指针指向的地址读取。图2-3所示的是主控设备执行写事务的实例波形。图2-4所示的是主控设备执行读事务的实例波形。

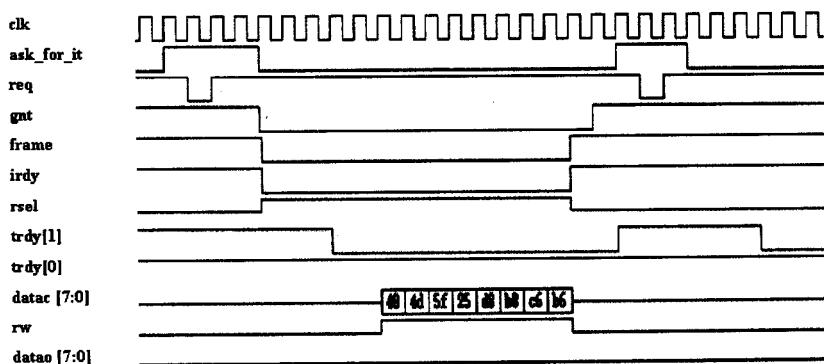


图 2-3 一个主控设备的写事务

一旦读或写操作完成，主控设备在下一个时钟周期将信号“frame”和“irdy”的断言解除以示操作完成，它也把“rsel”信号设为三态(tri-state)。在下一时钟，仲裁器确认“rsel”信号，并将“gnt”信号解除断言。一旦仲裁器清除了“gnt”信号，目标设备通过将“trdy”信号解除断言来确认整个事务的完成。

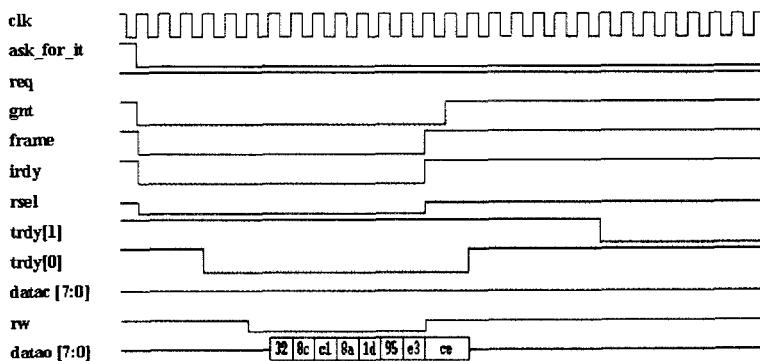


图 2-4 一个主控设备的读事务实例

2.1.2 中间设备

图 2-5 显示了有输入和输出端口的中间设备的方框图。中间设备负责两个重要的任务：

- (1) 提供仲裁逻辑来决定哪个主控设备管理与目标设备之间的事务。
- (2) 建立一个特定的主控设备和一个目标设备之间的连接。在一个给定的时间，许多主控设备可以通过断言各自的“req”信号来请求执行事务。

仲裁器使用轮转(round-robin)算法来决定哪个主控设备获得管理权。当仲裁器做出决定，它会将对应的主控设备的“gnt”信号断言。仲裁器可以在 2~5 个时钟周期内做出决定。仲裁器的内部逻辑可以用一个简单的零有效 one-hot 状态机(zero one-hot state machine)来描述。

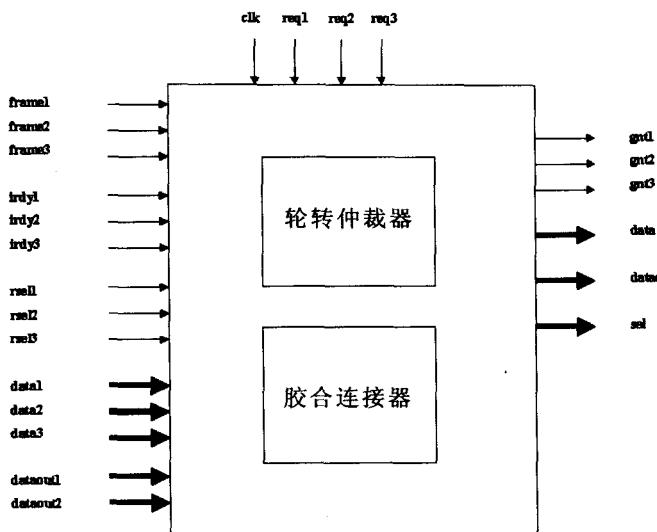


图 2-5 中间设备实例

主控设备选定要与其建立事务的目标设备后，中间设备将提供到指定的目标设备的信息。由于三个主控设备都能与任何一个目标设备建立事务，中间设备就不得不监控三个主控设备的“rsel”信号。在任何给定时间，要么所有三个“rsel”信号都处于三态，要么其中两个处于三态。如果所有三个“rsel”都是三态，那么在此时没有事务请求。如果有一个事务请求，那么其中的一个“rsel”信号的值为 0 或 1，具体的值依赖于选择的目标设备。如果信号“rsel”是 1，那么信号“sel”的最高位(MSB)置为高，表示目标设备 1 被选中，如果信号“rsel”是 0，那么信号“sel”的最低位(LSB)置为高，表示目标设备 0 被选中。

中间设备也为读和写事务选择正确的数据信号。如果是一个写事务，那么中间设备监控哪个主控设备的“rsel”信号是有效的，然后将这个主控设备的数据分配到选定的目标设备输入端。例如，如果主控设备 1 要对目标设备 0 做一个写操作，那么信号“rsel1”将置为低，并且总线“data1”将被分配到中间设备的输出总线“data”，这个输出将被导入选定的目标设备的输入端。在一个读事务中，中间设备则把来自目标设备的正确的输出数据分配到主

控设备。又比如一个读事务要从目标设备 1 读出数据，那么总线“dataout1”将被分配到总线“datao”。图 2-6 显示了中间设备功能的实例波形。

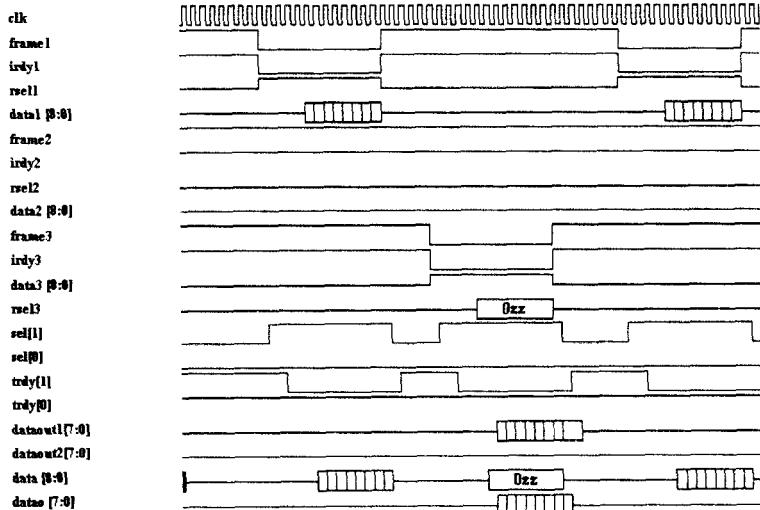


图 2-6 中间设备功能的波形

2.1.3 目标设备

图 2-7 显示了有输入输出端口的目标设备的方框图。目标设备有一个先进先出(FIFO)的内存，可以存储最多 64 字节的数据。

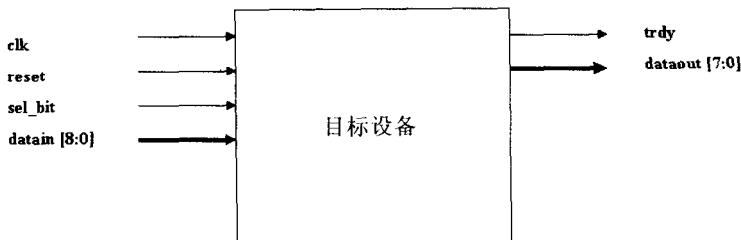


图 2-7 目标设备的实例

目标设备等待信号“sel_bit”的断言。一旦信号“sel_bit”被断言，目标设备要在两个时钟周期后通过断言信号“trdy”来确认它。断言了信号“trdy”后，如果是一个写事务，目标设备等待一

个有效数据和有效的写信号。一旦发现一个有效的写信号，输入的数据从目标设备的写指针寄存器(wi)的最新更新的值所指向的地址开始存储。如果是一个读事务，那么目标设备从当前读指针(ri)指向的内存的位置开始读出 8 个数据。

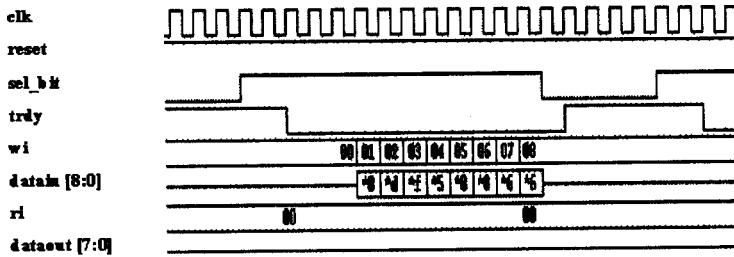


图 2-8 目标设备的写事务

事务的类型由总线“datain”的最高位(MSB)来表示。在一个读事务中，读的数据出现在总线向量“dataout”中。当事务完成，解除对信号“sel_bit”的断言，一个时钟周期后，解除对信号“trdy”的断言。图 2-9 显示了目标设备写操作的实例波形。图 2-9 显示了目标设备读操作的实例波形。

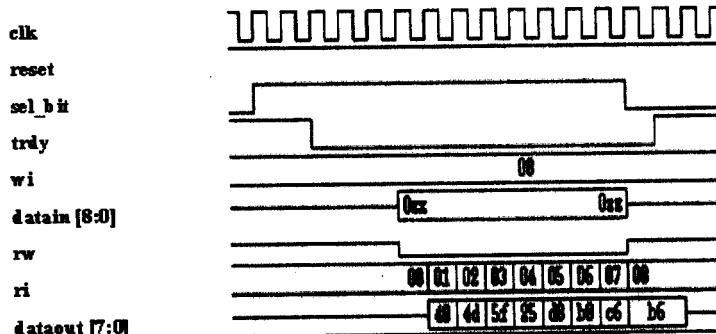


图 2-9 目标设备的读事务

2.2 块级验证

当每个独立设计模块完成后，应该对它们进行彻底的测试。块的穷举验证能提前发现边缘缺陷(corner case bugs)。在系统集成

前必须发现这些缺陷，因为在系统级发现这些缺陷是非常困难的。而且，系统级的出错很难识别定位，在系统级调试边缘缺陷是一项巨大的挑战。SVA 可以很容易地用来有效地测试独立设计块。在模块级，模拟工作量比较少，因此缺陷容易被跟踪并快速修复。在实例系统中有四个独立的设计块需要验证：

- (1) 主控设备
- (2) 目标设备
- (3) 仲裁器
- (4) 胶合(Glue)

还有两个需要彻底测试的块级接口：

- (1) 主控设备和中间设备
- (2) 目标设备和中间设备

2.2.1 SVA 在设计块中的应用

在用 SVA 进行块级验证中推荐下面的一些技巧：

- 所有为块级设计而写的 SVA 检验都应该是内嵌的。块级断言经常要访问设计的内部寄存器，因此把检验内嵌到设计模块更为有效率。
- 模块级的 SVA 检验的引用应该由设计模块内定义的参数来控制。这样，根据模拟的不同需求，可以自由地打开或关闭这些检验。
- 模块级的 SVA 检验的严重级别应该由设计模块内定义的参数来控制。SVA 里默认的严重级别将打印一个出错信息，然后继续模拟。
- 每个模块级的 SVA 检验都应该被执行并覆盖到。所有模块级检验一定要至少有一次真正的成功，这是必需的。

2.2.2 仲裁器的验证

基于 2.1.2 节描述的仲裁器的算法，可以析取出下面的 SVA 检验。在仲裁器的检验中重复使用的一些通用表达式，可以用 assign 语句做如下的定义：

```

assign frame = frame1 && frame2 && frame3;
assign irdy = irdy1 && irdy2 && irdy3;
assign gnt = !gnt1 || !gnt2 || !gnt3;
assign req = !req1 || !req2 || !req3;

```

信号“frame”和“irdy”都是低电平有效信号。每个主控设备有唯一的一个“frame”和“irdy”信号，它们是仲裁器模块的输入。如果主控设备在激活状态，它把信号“frame”和“irdy”都置为低。因此，把几个“frame”信号进行“与”操作，如果结果是低，则我们可以知道总线是激活的。同样地，把所有“irdy”信号进行“与”操作，如果结果是低，则我们可以知道总线是激活的。如果信号“frame”和“irdy”分别进行“与”操作的值都是高，则没有主控设备是激活的。

每个主控设备有唯一的一个“req”信号，用来请求总线，而且仲裁器提供唯一的“gnt”信号。通过把所有的“req”信号进行“或”操作，我们能知道即使只有一个主控设备有效的请求，仲裁器也会判断这个请求。同样地，通过把“gnt”信号进行“或”操作，我们可以知道一个主控设备获得许可。创建这样的中间表达式使SVA检验器具有更好的可读性。

Arb_chk1: 在任何给定的时钟边缘，仲裁器的内部状态变化应该是一个零有效“one-hot”状态机。

```

property p_arb_onehot0;
  @(posedge clk) $onehot0(state);
endproperty

```

Arb_chk2: 一旦主控设备有一个有效的请求，仲裁器应该在2~5个时钟周期内提供一个许可。

```

property p_req_gnt;
  @(posedge clk) $rose (req) |->
    ##[2:5] $rose (gnt);
endproperty

```

Arb_chk3: 一旦授予了许可，主控设备应该在同一时钟周期内通过断言“frame”和“irdy”信号来确认它接受这个许可。

```
property p_gnt_frame;
  @ (posedge clk) $rose (gnt) |->
    $fell (frame && irdy);
endproperty
```

Arb_chk4：一旦主控设备完成这个事务，它要解除对信号“frame”和“irdy”的断言，接着，仲裁器要在下一时钟周期解除对“gnt”信号的断言。

```
property p_frame_gnt;
  @ (posedge clk) $rose(frame && irdy)
    |-> $fell (gnt);
endproperty
```

2.2.3 模拟中针对仲裁器的 SVA 检验

2.2.2 节中所示的四个检验应该内嵌到仲裁器模块。根据需要，应该有一个规则来开关这些特性的检验，以及为这些特性设置严重等级。下面这些代码显示它是怎么实现的。

```
module arbiter(...);

  // port declarations

  parameter arb_sva = 1'b1;
  parameter arb_sva_severity = 1'b1;

  // Arbiter design description
  // SVA property description

  // SVA Checks

  always@ (posedge clk)
  begin
    if(arb_sva)
      begin

        a_arb_onehot0;
        assert property(p_arb_onehot0)
        else if(arb_sva_severity) $fatal;
```

```

a_req_gnt:
    assert property(p_req_gnt)
    else if(arb_sva_severity) $fatal;

a_gnt_frame :
    assert property(p_gnt_frame)
    else if(arb_sva_severity) $fatal;

a_frame_gnt:
    assert property(p_frame_gnt)
    else if(arb_sva_severity) $fatal;

c_arb_onehot0: cover property(p_arb_onehot0);
c_req_gnt: cover property(p_req_gnt);
c_gnt_frame: cover property(p_gnt_frame);
c_frame_gnt: cover property(p_frame_gnt);

end
end

endmodule

```

图 2-10 模拟中仲裁器的检验

要把检验包括进一个模拟中，参数“arb_sva”要设为 1。参数“arb_sva_severity”控制模拟中所做的动作。在这个例子里，如果参数设为 1，则严重度被设为\$fatal。这意味着，只要任何检验失败一次，就要退出模拟。把参数设为 0，检验会使用默认的设定，

即当失败时打印出错信息，然后继续模拟。图 2-10 显示的是实例模拟的一个波形。

2.2.4 主控设备的验证

基于 2.1.1 节中描述的主控设备的协议，可以析取出下面的 SVA 检验。注意每个主控设备只有一个“req”、“gnt”、“frame”和“irdy”信号。在主控设备的检验器中提到的这些信号并不是在仲裁器检验中定义的表达式中的信号。它们只是出现在每个主控设备中的单独的信号。

Master_chk1: 一旦主控设备有一个有效的请求，许可信号要在 2~5 个时钟内到达。如果是这样，并且信号“r_sel”为高，那么在同一时钟周期，主控设备应该断言信号“frame”和“irdy”。三个时钟周期后，目标设备 1 应该通过断言信号“trdy”确认这个选择。

```
property p_master_start1;
  @(posedge clk)
    ($fell (req) ##[2:5] ($fell(gnt) && r_sel)) |->
      (!frame && !irdy) ##3 !trdy[1];
endproperty
```

Master_chk2: 一旦主控设备有一个有效的请求，许可信号要在 2~5 个时钟周期内到达。如果是这样，并且信号“r_sel”为低，那么在同一时钟周期内，主控设备应该断言信号“frame”和“irdy”。三个时钟周期后，目标设备 0 应该通过断言信号“trdy”确认这个选择。

```
property p_master_start2;
  @(posedge clk)
    ($fell (req) ##[2:5] ($fell(gnt) && !r_sel)) |->
      (!frame && !irdy) ##3 !trdy[0];
endproperty
```

Master_chk3: 一旦目标设备确认了它的选择，主控设备应该在 10 个时钟周期内完成这个事务。通过解除对信号“frame”和“irdy”的断言表示这个事务完成。一个时钟周期后，应该解除

对信号“gnt”的断言。

```
property p_master_stop1;
@(posedge clk)
$fell (trdy[1]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty

property p_master_stop2;
@(posedge clk)
$fell (trdy[0]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty
```

注意，我们用两个独立的属性来检验这个事务是否完成，每个目标设备各有一个。

Master_chk4：如果主控设备正处在一个写事务中，那么总线的数据(data_c)不应该是三态，应该是有效的数据。

```
property p_master_data1;
@(posedge clk)
($fell (trdy[1]) ##2 rw) |->
    ($isunknown (data) == 0) [*7];
endproperty

property p_master_data2;
@(posedge clk)
($fell (trdy[0]) ##2 rw) |->
    ($isunknown (data) == 0) [*7];
endproperty
```

- 注意，这两个独立的属性分别对应一个目标设备，检验写事务中数据的正确性。
- 注意，如果信号“rw”为高，那么主控设备正在操作一个写事务。

Master_chk5：如果主控设备处在一个读事务，那么总线的数据(data_o)不应该是三态，应该是有效的数据。

```
property p_master_data01;
@(posedge clk)
($fell (trdy[1]) ##3 !rw) |=>
    ($isunknown (data_o) == 0) [*7];
```

```

endproperty

property p_master_datao2;
  @(posedge clk)
    ($fell (trdy[0]) ##3 !rw) |=>
      ($isunknown (data_o) == 0) [*?];
endproperty

```

- 注意，这两个独立的属性分别对应一个目标设备，检验读事务中数据的正确性。
- 注意，如果信号“rw”为低，那么主控设备正在操作一个读事务。

2.2.5 模拟中针对主控设备的 SVA 检验

2.2.4 节中所示的五个检验应该内嵌到主控设备模块。根据需要，应该有一个规则来控制这些属性，下面这些代码显示它是怎么实现的。

```

module master(...);

  // port declarations

  parameter master_sva = 1'b1;
  parameter master_sva_severity = 1'b1;

  // Master design description

  // SVA property description

  // SVA Checks

  always@ (posedge clk)

  begin
    if (master_sva)
      begin
        a_master_start1:
          assert property(p_master_start1)
          else if(master_sva_severity) $fatal;

```

```
a_master_start2:  
    assert property(p_master_start2)  
    else if(master_sva_severity) $fatal;  
  
a_master_stop1:  
    assert property(p_master_stop1)  
    else if(master_sva_severity) $fatal;  
  
a_master_stop2:  
    assert property(p_master_stop2)  
    else if(master_sva_severity) $fatal;  
  
a_master_data1:  
    assert property(p_master_data1)  
    else if(master_sva_severity) $fatal;  
  
a_master_data2:  
    assert property(p_master_data2)  
    else if(master_sva_severity) $fatal;  
  
a_master_data01:  
    assert property(p_master_data01)  
    else if(master_sva_severity) $fatal;  
  
a_master_data02:  
    assert property(p_master_data02)  
    else if(master_sva_severity) $fatal;  
  
c_master_start1: cover property(p_master_start1);  
c_master_start2: cover property(p_master_start2);  
c_master_stop1: cover property(p_master_stop1);  
c_master_stop2: cover property(p_master_stop2);  
c_master_data1: cover property(p_master_data1);  
c_master_data2: cover property(p_master_data2);  
c_master_data01: cover property(p_master_data01);  
c_master_data02: cover property(p_master_data02);  
  
end  
  
end  
  
endmodule
```

图 2-11 显示的是这些主控设备检验的模拟波形的实例。

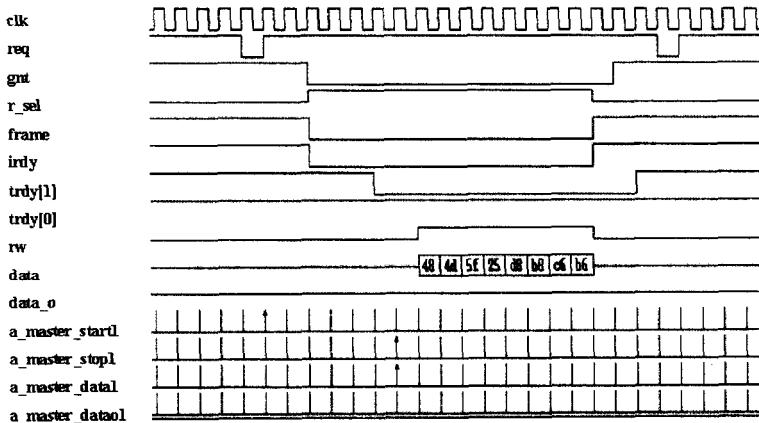


图 2-11 模拟中对目标设备 1 的主控设备的检验

2.2.6 胶合(Glue)的验证

基于 2.1.2 节描述的胶合逻辑(glue logic)的协议，可以析取出下面的 SVA 检验。

Glue_chk1: 如果任何一个主控设备的选择信号“sel1”、“sel2”和“sel3”为高，那么目标设备 1 被选中。

```
property p_sel_1;
  @ (posedge clk)
    (rsel1 || rsel2 || rsel3) |=> sel == 2'b10;
endproperty
```

Glue_chk2: 如果任何一个主控设备的选择信号“sel1”、“sel2”和“sel3”为低，那么目标设备 0 被选中。

```
property p_sel_0;
  @ (posedge clk)
    (!rsel1 || !rsel2 || !rsel3) |=> sel == 2'b01;
endproperty
```

Glue_chk3: 在一个写事务中，如果信号“rsel1”不是三态，那么从主控设备 1 来的数据应该写入相应的目标设备。

```

property p_rsel1_write;
  @(posedge clk)
  ((rsel1 || !rsel1) ##3 ($fell (trdy[1]) || 
    $fell(trdy[0])) ##3 data1[8]) |->
    (data == $past(data1)) [*7];
endproperty

```

- 注意，通过使用总线数据的最高位，我们可以确定事务的种类(读/写)。
- 如果总线数据的最高位是高，那么这是一个写事务。
- 如果总线数据的最高位是低，那么这是一个读事务。
- 在主控设备内，事务的种类由信号“rw”来确定。这个信号复制了总线数据的最高位。信号“rw”是主控设备的局部信号。外部接口应该由总线数据的最高位推断出事务的种类。

Glue_chk4：在一个写事务中，如果信号“rsel2”不是三态，那么从主控设备 2 来的数据应该写到对应的目标设备。

```

property p_rsel2_write;
  @(posedge clk)
  ((rsel2 || !rsel2) ##3 ($fell (trdy[1]) || 
    $fell(trdy[0])) ##3 data2[8]) |->
    (data == $past(data2)) [*7];
endproperty

```

Glue_chk5：在一个写事务中，如果信号“rsel3”不是三态，那么从主控设备 3 来的数据应该写到对应的目标设备。

```

property p_rsel3_write;
  @(posedge clk)
  ((rsel3 || !rsel3) ##3 ($fell (trdy[1]) || 
    $fell(trdy[0])) ##3 data3[8]) |->
    (data == $past(data3)) [*7];
endproperty

```

Glue_chk6：在一个读事务中，如果目标设备 1 被选中，从目标设备 1(dataout1)中读出的数据应该读入到对应的主控设备。

```

property p_read1;
  @(posedge clk)
  ($fell (trdy[1]) ##4 !data[8]) |->
    (dataout1 == datao) [*7];
endproperty

```

Glue_chk7: 在一个读事务中, 如果目标设备 0 被选中, 从目标设备 0(dataout2)中读出的数据应该读入到对应的主控设备。

```

property p_read0;
  @(posedge clk)
  ($fell (trdy[0]) ##4 !data[8]) |->
    (dataout2 == datao) [*7];
endproperty

```

2.2.7 模拟中针对胶合逻辑(glue logic)的 SVA 检验

2.2.6 节中所示的七个检验应该内嵌到胶合模块内。根据需要, 应该有一个规则来控制这些属性。下面这些代码显示它是怎么实现的。

```

module glue(. . .);
  // port declarations

  parameter glue_sva = 1'b1;
  parameter glue_sva_severity = 1'b1;

  // glue design description
  // glue SVA property description

  // SVA Checks

  always@(posedge clk)
  begin
    if(glue_sva)
      begin

        a_sel_1:
          assert property(p_sel_1)
            else if(glue_sva_severity) $fatal;

```

```
a_sel_0:  
    assert property(p_sel_0)  
        else if(glue_sva_severity) $fatal;  
  
a_rsel1_write:  
    assert property(p_rsel1_write)  
        else if(glue_sva_severity) $fatal;  
  
a_rsel2_write:  
    assert property(p_rsel2_write)  
        else if(glue_sva_severity) $fatal;  
  
a_rsel3_write:  
    assert property(p_rsel3_write)  
        else if(glue_sva_severity) $fatal;  
  
a_read1:  
    assert property(p_read1)  
        else if(glue_sva_severity) $fatal;  
  
a_read0:  
    assert property(p_read0)  
        else if(glue_sva_severity) $fatal;  
  
c_sel_1: cover property(p_sel_1);  
c_sel_0: cover property(p_sel_0);  
c_rsel1_write: cover property(p_rsel1_write);  
c_rsel2_write: cover property(p_rsel2_write);  
c_rsel3_write: cover property(p_rsel3_write);  
c_read1: cover property(p_read1);  
c_read0: cover property(p_read0);  
  
end  
end  
  
endmodule
```

图 2-12 显示的是胶合检验的模拟波形的实例。

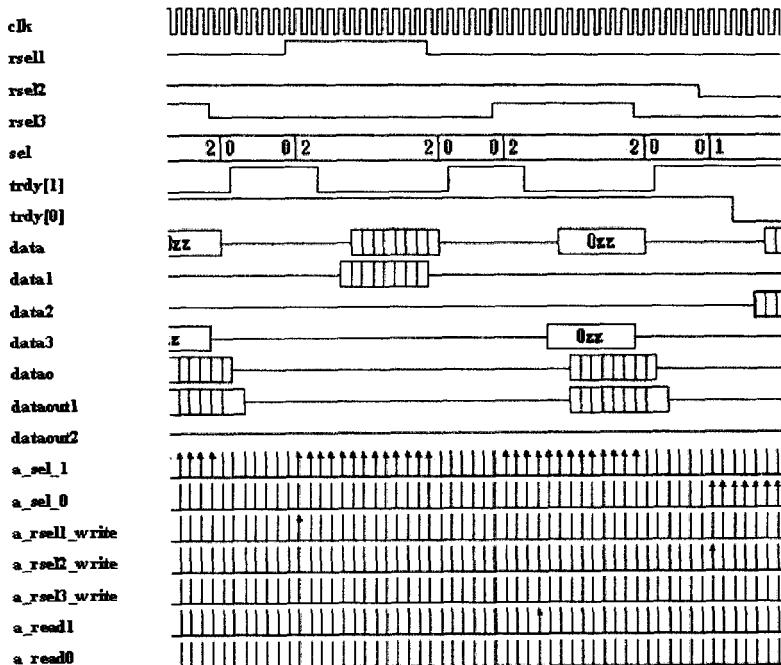


图 2-12 模拟中的胶合检验

2.2.8 目标设备的验证

基于 2.1.3 节中描述的目标设备的协议，可以析取出下面的 SVA 检验。

Target_chk1: 如果选中一个目标设备，那么它应该在两个时钟周期后断言信号“trdy”。

```
property p_sel_trdy_start;
  @ (posedge clk) $rose (sel_bit) |->
    ##1 trdy ##1 !trdy;
endproperty
```

Target_chk2: 在一个事务结束时，信号“sel_bit”被解除断言。一个时钟周期后，信号“trdy”应该被解除断言。

```
property p_sel_trdy_stop;
  @ (posedge clk) $fell (sel_bit) |=> trdy;
endproperty
```

Target_chk3: 在一个写事务中，每次在一个时钟周期后写指针应该加 1，以保证每次有效地写入一个唯一的地址。

```
property p_write;
  @ (posedge clk)
  (datain[8] && sel_bit && (wi != 63)) |>
    (wi == ($past(wi) + 1));
endproperty
```

- 注意，地址指针累加会从 63 回到 0，再从 0 开始累加。因此，如果在一个给定的时钟边缘写指针在 63 的位置，这个检验不适用。
- 可以写一个不同的检验来验证指针总是会正确地从 63 回到 0。

Target_chk4: 在一个读事务中，每次在一个时钟周期内从唯一的地址读出有效的数据完成后，读指针应该加 1。

```
property p_read;
  @ (posedge clk)
  (!datain[8] && sel_bit && (ri != 63)) |=>
    (ri == ($past(ri) + 1));
endproperty
```

- 注意，对于读指针，当指针在 63 的位置，这个检验不适用。
- 读操作有一个时钟周期的延时，因此我们用非重叠蕴含操作符。
- 由于采用非重叠操作符，检验往后移了一个周期，与前一周期的地址作比较。
- 例如，在一个给定的时钟边缘，如果蕴含表达式的先行算子为真，则检验移到下一时钟周期。当指针在 63，如果指针加 1，那么检验会移到指针 0，然后比较 63 和 0 之间的加 1 关系。这是不正确的。因此，在一个给定的时钟边缘，如果读指针的值为 63，不能执行这个检验。
- 可以写一个独立的检验来确认指针能从 63 回到 0。

Target_chk5: 在一个有效的读或写的事务中，从目标设备读出或写入的数据应该是有效的。

```

property p_target_datain;
  @ (posedge clk)
  ($fell (trdy) ##3 (datain[8])) |->
    not ($isunknown (datain)) [*7];
endproperty

property p_target_dataout;
  @ (posedge clk)
  ($fell (trdy) ##3 (!datain[8])) |=>
    not ($isunknown (dataout)) [*7];
endproperty

```

2.2.9 模拟中针对目标设备的 SVA 检验

2.2.8 节中所示的五个检验应该内嵌到目标设备的模块内。根据需要，应该有一个规则来控制这些属性。下面这些代码显示它是怎么实现的。

```

module target(....);

  // port declarations

  parameter target_sva = 1'b1;
  parameter target_sva_severity = 1'b1;

  // target design description
  // target SVA property description
  // SVA Checks

  always@ (posedge clk)
  begin
    if(target_sva)
      begin

        a_sel_trdy_start:
          assert property(p_sel_trdy_start)
          else    if(target_sva_severity) $fatal;
        a_sel_trdy_stop:
          assert property(p_sel_trdy_stop)
          else    if(target_sva_severity) $fatal;

        a_write:
          assert property(p_write)
      end
    end
  end

```

```

    else if(target_sva_severity) $fatal;

a_read:
    assert property(p_read)
else if(target_sva_severity) $fatal;

a_target_datain:
    assert property(p_target_datain)
else if(target_sva_severity) $fatal;

a_target_dataout:
    assert property(p_target_dataout)
else if(target_sva_severity) $fatal;

c_sel_trdy_start:
    cover property(p_sel_trdy_start);
c_sel_trdy_stop: cover property(p_sel_trdy_stop);
c_write: cover property(p_write);
c_read: cover property(p_read);
c_target_datain: cover property(p_target_datain);
c_target_dataout: cover property(p_target_dataout);

end
end
endmodule

```

图 2-13 所示的是针对目标设备的检验做的模拟波形的例子。

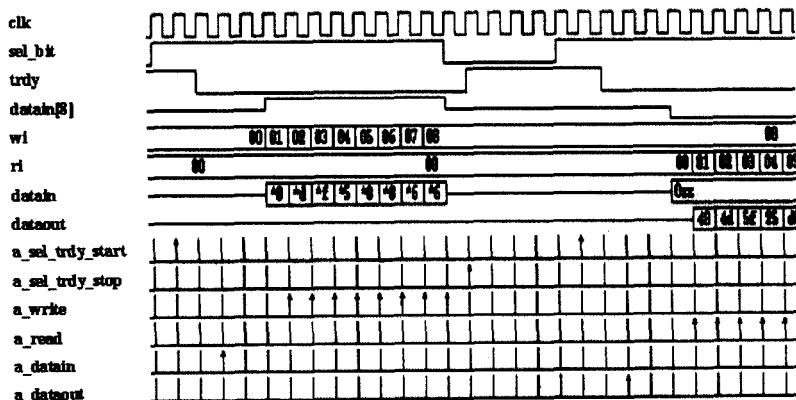


图 2-13 模拟中针对设备的检验

2.3 系统级验证

在这个系统中，一个中间设备实例连着三个主控设备和两个目标设备。系统的顶层连接如下所示。

```
module top(..., ...);
    // port declarations

    master u1 (ask[2], clk, req1, gnt1, frame1,
               irdy1, trdy, data1, rsel1, datao);

    master u2 (ask[1], clk, req2, gnt2, frame2,
               irdy2, trdy, data2, rsel2, datao);

    master u3 (ask[0], clk, req3, gnt3, frame3,
               irdy3, trdy, data3, rsel3, datao);

    arbiteru4(clk, reset, frame, irdy, req1, req2,
              req3, gnt1, gnt2, gnt3);

    glue u5 (clk, frame1, irdy1, frame2, irdy2,
              frame3, irdy3, trdy, rsel1, rsel2, rsel3, data1,
              data2, data3, sel, data, dataout1, dataout2,
              datao);

    target u6 (clk, reset, sel[1], trdy[1], data,
               dataout1);

    target u7 (clk, reset, sel[0], trdy[0], data,
               dataout2);

endmodule
```

在用 SVA 进行系统级验证时推荐下面一些技巧：

- 由于单个块的内部功能已经仔细验证过，所以默认状态下，在系统级验证时不把块级断言包含进去。这么做主要是为了提高性能。

- 如果性能不是瓶颈，默认情况下，在系统级验证中把块级断言包含进去。系统接口提供了更加真实的和一些期望之外的输入情况，而模块级断言要能对此作出正确的反应。
- 如果有断言失败，验证环境要提供机制来激活模块级断言。例如，在我们的实例系统中，如果主控设备 1 和目标设备 0 之间的事务出错，把针对主控设备 1 和目标设备 0 写的块级 SVA 检验包括进去，重跑系统级模拟。
- 在系统级，要写一些新的断言来验证系统的连接，这些检查更应该关注接口规则而不是块内部细节。

针对系统级验证的 SVA 检验

基于系统的连接和协议，针对系统级验证可以写出如下一些检验。

Ss_shk1：在任何给定的时间点，只能断言一个“trdy”信号。换言之，在任何给定的时间，一个事务中只能有一个目标设备。

```
property p_target;
  @ (posedge clk) not (!trdy[0] && !trdy[1]);
endproperty
```

Ss_chk2：在任何给定的时钟周期内，只能断言一对“frame”和“irdy”信号。换言之，在任何给定的时间，一个事务中只能有一个主控设备。

```
property p_frame;
  @ (posedge clk)
    $countones({frame1, frame2, frame3}) >1;
endproperty

property p_irdy;
  @ (posedge clk)
    $countones({irdy1, irdy2, irdy3}) >1;
endproperty
```

Ss_chk3：在任何给定的时间，只能断言一个“gnt”信号。换而言之，仲裁器一次只能让一个主控设备开始事务。

```
property p_gnt;
  @(posedge clk)
    $countones({gnt1, gnt2, gnt3}) > 1;
endproperty
```

Ss_chk4: 在任何给定的时钟周期，只能激活一个“rw”信号，其他的“rw”信号应该是三态(“rw”信号是主控设备数据输出总线的最高位)。

```
property p_rw;
  @(posedge clk)
    ($isunknown(rw1)      &&      $isunknown(rw2)      &&
     $isunknown(rw3) ) ||
    ((rw1==1'b1 || rw1==1'b0) && $isunknown (rw2) &&
     $isunknown(rw3)) ||
    ((rw2==1'b1 || rw2==1'b0) && $isunknown (rw1) &&
     $isunknown(rw3)) ||
    ((rw3==1'b1 || rw3==1'b0) && $isunknown (rw2) &&
     $isunknown(rw2));
endproperty
```

Ss_chk5: 在任何给定的时钟周期，只能激活一个“rsel”信号，其他“rsel”信号应该是三态。

```
property p_rsel;
  @(posedge clk)
    $isunknown(rsell1)      &&      $isunknown(rsell2)      &&
    $isunknown(rsell3) ) ||
    ((rsell1==1'b1 || rsell1==1'b0) && $isunknown (rsell2)
     && $isunknown(rsell3)) ||
    ((rsell2==1'b1 || rsell2==1'b0) && $isunknown (rsell1)
     && $isunknown(rsell3)) ||
    ((rsell3==1'b1 || rsell3==1'b0) && $isunknown (rsell2)
     && $isunknown(rsell1));
endproperty
```

Ss_chk6: 一旦一个主控设备发出一个正确的请求，在2~5个时钟周期内，一个正确的“gnt”应该到达。

```
assign req = !req1 || !req2 || !req3;
assign gnt = !gnt1 || !gnt2 || !gnt3;

property p_req_gnt_w;
```

```

@ (posedge clk)
    $rose (req) |-> ##[2:5] $rose(gnt);
endproperty

```

Ss_chk7: 在任何给定的时钟, 如果一个主控设备的“frame”和“irdy”信号被断言了, 那么3个时钟周期后, 相关的“trdy”信号应该被断言。

```

assign frame_ = !frame1 || !frame2 || !frame3;
assign irdy_ = !irdy1 || !irdy2 || !irdy3;

property p_start_frame;
    @ (posedge clk)
        $rose(frame_ && irdy_) |->##3 $rose(trdy_);
endproperty

```

Ss_chk8: 在任何给定的时钟, 如果主控设备的“frame”和“irdy”信号被解除断言, 那么两个时钟周期后, 相关的“trdy”信号应该被解除断言。

```

assign trdyp = trdy[1] && trdy[0];

property p_end_frame;
    @ (posedge clk)
        $rose(frame && irdy) |->##2 $rose(trdyp);
endproperty

```

Ss_chk9: 在任何给定的时钟, 如果没有有效的事务, 那么总线“data”和“datao”应该是三态。

```

property p_bus_not_in_use;
    @ (posedge clk)
        trdyp |->
            ($isunknown(data) && $isunknown(datao));
endproperty

a_target : assert property(p_target);
a_frame: assert property(p_frame);
a_irdy: assert property(p_irdy);
a_rsel: assert property(p_rsel);
a_rw: assert property(p_rw);
a_gnt: assert property(p_gnt);
a_req_gnt_w : assert property(p_req_gnt_w);

```

```

a_start_frame: assert property(p_start_frame);
a_end_frame: assert property(p_end_frame);
a_bus_in_use: assert property(p_bus_not_in_use);

c_target : cover property(p_target);
c_frame: cover property(p_frame);
c_irdy: cover property(p_irdy);
c_rsel: cover property(p_rsel);
c_rw: cover property(p_rw);
c_gnt: cover property(p_gnt);
c_req_gnt_w : cover property(p_req_gnt_w);
c_start_frame: cover property(p_start_frame);
c_end_frame: cover property(p_end_frame);
c_bus_in_use: cover property(p_bus_not_in_use);

```

在系统级模拟中，最顶层的模块应该配置参数设定，这样可以关闭所有块级断言。在我们的实例系统中，由于每个设计块都有一个参数，如果需要可以把相关的 SVA 检验包括进来。如下所示，我们可以轻松地配置系统级的顶层模块。

```

module top(..., ...);
  // port declarations

  master
  #( .master_sva(1'b0), .master_sva_severity(1'b0))
  u1 (ask[2], clk, req1, gnt1, frame1, irdy1, trdy,
       data1, rsel1, datao);

  master
  #( .master_sva(1'b0), .master_sva_severity(1'b0))
  u2 (ask[1], clk, req2, gnt2, frame2, irdy2, trdy,
       data2, rsel2, datao);

  master
  #( .master_sva(1'b0), .master_sva_severity(1'b0))
  u3 (ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
       data3, rsel3, datao);

  arbiter
  #( .arb_sva(1'b0), .arb_sva_severity(1'b0))
  u4 (clk, reset, frame, irdy, req1, req2, req3,

```

```
gnt1, gnt2, gnt3);  
  
glue  
#(.glue_sva(1'b0), .glue_sva_severity(1'b0))  
u5 (clk, frame1, irdy1, frame2, irdy2, frame3,  
  
      irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,  
      data3, sel, data, dataout1, dataout2, datao);  
  
target  
#(.target_sva(1'b0), .target_sva_severity(1'b0))  
u6 (clk, reset, sel[1], trdy[1], data, dataout1);  
  
target  
#(.target_sva(1'b0), .target_sva_severity(1'b0))  
u7 (clk, reset, sel[0], trdy[0], data, dataout2);  
endmodule
```

注意，每个设计块实例化时，都传入了参数的值。在每个实例化中，第一个参数“*_sva”都被设为 0，这就表示块级断言将不被执行，系统级模拟只做系统级的检验。

假设在系统级模拟时“Ss_chk6”报告了一些失败。这个检验是查找主控设备和仲裁器之间是否存在接口错误。要调试这类错误，可以重新运行模拟并把主控设备和仲裁器相关的块级检验包括进去。这个模拟的顶层模块的配置如下所示：

```
module top(..., ...);  
  
// port declarations  
  
master  
#(.master_sva(1'b1), .master_sva_severity(1'b0))  
u1 (ask[2], clk, req1, gnt1, frame1, irdy1, trdy,  
      data1, rsel1, datao);  
  
master  
#(.master_sva(1'b1), .master_sva_severity(1'b0))  
u2 (ask[1], clk, req2, gnt2, frame2, irdy2, trdy,  
      data2, rsel2, datao);  
  
master
```

```

#.master_sva(1'b1), .master_sva_severity(1'b0))
u3(ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
data3, rsel3, datao);

arbiter
#.arb_sva(1'b1), .arb_sva_severity(1'b0))
u4(clk, reset, frame, irdy, req1, req2, req3,
gnt1, gnt2, gnt3);

glue
#.glue_sva(1'b0), .glue_sva_severity(1'b0))
u5(clk, frame1, irdy1, frame2, irdy2, frame3,
irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,
data3, sel, data, dataout1, dataout2, datao);

target
#.target_sva(1'b0), .target_sva_severity(1'b0))
u6(clk, reset, sel[1], trdy[1], data, dataout1);

target
#.target_sva(1'b0), .target_sva_severity(1'b0))
u7(clk, reset, sel[0], trdy[0], data, dataout2);

endmodule

```

注意，这个配置中，参数“`master_sva`”和“`arb_sva`”被设为1。在基本的设计块中，用“`ifdef - endif`”结构可以有条件地把SVA检验包括进去。通过条件编译SVA的代码，用户可以在模块的所有实例中做这些检验，也可以不做这些检验。这个方法的不足之处是它是一个全局控制机制，会影响所有的模块级检验。通过使用参数，这个不足可以被克服，用户在模拟中可以更加灵活地选择需要的块级检验。

2.4 功能覆盖

到目前为止，编写的系统级检验寻求的是任何可能存在的违反指定的协议的情况。模拟中，确信这些检验至少被执行一次，很大程度上提高了系统功能的可信等级。功能覆盖的另一方面是

从测试平台角度考虑，在模拟中覆盖系统功能的所有可能的情景。在模拟中需要覆盖的情景应该成为测试计划的一部分。

为动态模拟写的 SVA 检验的效果仍然依赖于输入激励。如果输入向量不促使系统执行某些情景，那么它们就不会被测试到。许多测试平台采用随机技术产生模拟的输入激励。一个非常通用的方法是运行许多预先定义的事务，然后测量对某些情景集合的覆盖。通过约束控制输入激励的随机产生，会更加有效地覆盖到各种情景。关键是在最少数量的周期内，达到最大的功能覆盖。从 SVA 收集来的覆盖信息能有效地用来建立反应验证环境 (reactive verification environments)。

2.4.1 实例系统的覆盖率计划

本章中讨论的实例系统有很多重要的功能，应该作为功能验证的一部分被覆盖到。

1. 请求情景(Request Scenario)

“所有可能的请求情景都应该被覆盖到”。

在任何给定的时间，有三个主控设备可以请求访问。这就意味着主控设备的“req”信号有七种可能的组合，如表 2-1 所示。

表 2-1 主控设备请求情景

Req1	Req2	Req3
0	1	1
1	0	1
1	1	0
0	0	1
1	0	0
0	1	0
0	0	0

表中的 0 表示主控设备正在请求总线。测试平台应该在模拟中生成所有这些可能的输入组合。

下面的代码举例说明了如何用功能覆盖的数据来控制模拟环境。所有七种可能的请求组合的属性定义如下。

```

property p_req1; // master 1 requesting
    @(posedge clk) $fell (req1) && req2 && req3;
endproperty

property p_req2; // master 2 requesting
    @(posedge clk) $fell (req2) && req1 && req3;
endproperty

property p_req3; // master 3 requesting
    @(posedge clk) $fell (req3) && req1 && req2;
endproperty

property p_req12; // master 1&2 requesting
    @(posedge clk)
        $fell (req1) && $fell(req2)&& req3;
endproperty

property p_req23; // master 2&3 requesting
    @(posedge clk)
        $fell (req2) && $fell(req3) && req1;
endproperty

property p_req31; // master 1&3 requesting
    @(posedge clk)
        $fell (req3) && $fell(req1) && req2;
endproperty

property p_req123; // master 1&2&3 requesting
    @(posedge clk)
        $fell (req1) && $fell(req2) && $fell(req3);
endproperty

```

每个属性应该有一个相连的如下所示的覆盖(cover)语句。覆盖语句的执行块能用来更新寄存器的标识。对于这种情况，每次覆盖到属性，一个局部的寄存器计数就会加 1。在同一个时钟，我们检验计数器的值是否已经达到 3。如果是，那么与这个属性相连的标识要被断言。换言之，在模拟中，期望每种请求组合会出现三次，当出现三次时，一个与指定的请求组合相连的标识将被断言。

```
c_req1: cover property(p_req1)
begin
    creq1++;
    if(creq1 == 3) creq1_flag = 1'b1;
end

c_req2: cover property(p_req2)
begin
    creq2++;
    if(creq2 == 3) creq2_flag = 1'b1;
end

c_req3: cover property(p_req3)
begin
    creq3++;
    if(creq3 == 3) creq3_flag = 1'b1;
end

c_req12: cover property(p_req12)
begin
    creq12++;
    if(creq12 == 3) creq12_flag = 1'b1;
end

c_req23: cover property(p_req23)
begin
    creq23++;
    if(creq23 == 3) creq23_flag = 1'b1;
end

c_req31: cover property(p_req31)
begin
    creq31++;
    if(creq31 == 3) creq31_flag = 1'b1;
end

c_req123: cover property(p_req123)
begin
    creq123++;
    if(creq123 == 3) creq123_flag = 1'b1;
end
```

这种覆盖信息可以用来有效地控制模拟环境。在实例系统的随机测试平台中，预先确定的许多事务一个接一个地执行，当所

有的事务完成了，模拟也将结束。下面的代码显示如何利用功能覆盖的信息来终止一个模拟。

```

always@(posedge clk)
begin

    if(creq1_flag && creq2_flag && creq3_flag &&
        creq12_flag && creq23_flag && creq31_flag &&
        creq123_flag)

    begin

        $display("FC: All possible request scenarios
covered 3 times each\n");
        $finish();
    end
end

```

在这段代码中，有两种方法来终止一个模拟：

- (1) 随机运行完给定数量的事务，然后退出。
 - (2) 如果所有可能的请求情景每种都覆盖过三次，那么退出。
- 无论哪种情况先发生都将终止模拟。

2. 主控设备到目标设备的事务

每个主控设备应该对每个目标设备执行一个读事务和一次写事务。

系统中有三个主控设备和两个目标设备。这样就生成了 12 种可能的情景，如表 2-2 所示。所有 12 种可能的事务组合的属性定义生成如下所示。

表 2-2 主控设备到目标设备的事务

主控设备	目标设备	事 务
M1	T1	读
M1	T1	写
M1	T0	读
M1	T0	写
M2	T1	读

(续表)

主控设备	目标设备	事 务
M2	T1	写
M2	T0	读
M2	T0	写
M3	T1	读
M3	T1	写
M3	T0	读
M3	T0	写

```

property p_m1t1r;
// master1 reading from target 1
@(posedge clk)
$fell (frame1 && irdy1) |->
##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m1t1w;
// master 1 writing to target 1
@(posedge clk)
$fell (frame1 && irdy1) |->
##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m1t0r;
// master 1 reading from target 0
@(posedge clk)
$fell (frame1 && irdy1) |->
##3 ($fell (trdy[0])) ##3 !data[8];
endproperty

property p_m1t0w;
// master 1 writing to target 0
@(posedge clk)
$fell(frame1 && irdy1) |->
##3 ($fell(trdy[0])) ##3 data[8];
endproperty

property p_m2t1r;
// master 2 reading from target 1
@(posedge clk)

```

```
$fell (frame2 && irdy2) |->
      ##3 ($fell(trdy[1])) ##3 !data[8];
endproperty

property p_m2t1w;
// master 2 writing to target 1
@(posedge clk)
$fell (frame2 && irdy2) |->
      ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m2t0r;
// master 2 reading from target 0
@(posedge clk)
$fell (frame2 && irdy2) |->
      ##3 ($fell(trdy[0])) ##3 !data[8];
endproperty

property p_m2t0w;
// master 2 writing to target 0
@(posedge clk)
$fell (frame2 && irdy2) |->
      ##3 ($fell (trdy[0])) ##3 data[8];
endproperty

property p_m3t1r;
// master 3 reading from target 1
@(posedge clk)
$fell (frame3 && irdy3) |->
      ##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m3t1w;
// master 3 writing to target 1
@(posedge clk)
$fell (frame3 && irdy3) |->
      ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m3t0r;
// master 3 reading from target 0
@(posedge clk)
$fell (frame3 && irdy3) |->
      ##3 ($fell (trdy[0])) ##3 !data[8];
```

```
endproperty

property p_m3t0w;
// master 3 writing to target 0
@(posedge clk)
$fall (frame3 && irdy3) |->
    ##3 ($fall (trdy[0])) ##3 data[8];
endproperty
```

每个属性应该有一个与之相连的如下所示的覆盖语句。采用与前面“请求情景”中的相同技巧来计算情景出现的次数。

```
c_m1t1r: cover property(p_m1t1r)
begin
    m1_t1_r++;
    if(m1_t1_r == 3) m1_t1_r_flag = 1'b1;
end

c_m1t1w: cover property(p_m1t1w)
begin
    m1_t1_w++;
    if(m1_t1_w == 3) m1_t1_w_flag = 1'b1;
end

c_m1t0r: cover property(p_m1t0r)
begin
    m1_t0_r++;
    if(m1_t0_r == 3) m1_t0_r_flag = 1'b1;
end

c_m1t0w: cover property(p_m1t0w)
begin
    m1_t0_w++;
    if(m1_t0_w == 3) m1_t0_w_flag = 1'b1;
end

c_m2t1r: cover property(p_m2t1r)
begin
    m2_t1_r++;
    if(m2_t1_r == 3) m2_t1_r_flag = 1'b1;
end

c_m2t1w: cover property(p_m2t1w)
begin
```

```

        m2_t1_w++;
        if(m2_t1_w == 3) m2_t1_w_flag = 1'b1;
    end

c_m2t0r: cover property(p_m2t0r)
begin
    m2_t0_r++;
    if(m2_t0_r == 3) m2_t0_r_flag = 1'b1;
end

c_m2t0w: cover property(p_m2t0w)
begin
    m2_t0_w++;
    if(m2_t0_w == 3) m2_t0_w_flag = 1'b1;
end

c_m3t1r: cover property(p_m3t1r)
begin
    m3_t1_r++;
    if(m3_t1_r == 3) m3_t1_r_flag = 1'b1;
end

c_m3t1w: cover property(p_m3t1w)
begin
    m3_t1_w++;
    if(m3_t1_w == 3) m3_t1_w_flag = 1'b1;
end

c_m3t0r: cover property(p_m3t0r)
begin
    m3_t0_r++;
    if(m3_t0_r == 3) m3_t0_r_flag = 1'b1;
end

c_m3t0w: cover property(p_m3t0w)
begin
    m3_t0_w++;
    if(m3_t0_w == 3) m3_t0_w_flag = 1'b1;
end

```

“请求情景”和“主控设备到目标设备的事务”中的覆盖信息能用来有效地控制模拟环境。在下面所示的这段代码中，有两种方法可以终止这个模拟：

- (1) 随机运行完预先确定数量的事务，然后退出。
- (2) 如果所有可能的请求情景被覆盖了三次和所有可能的主控设备到目标设备的事务覆盖了三次，那么退出这个模拟。

无论哪种情况先出现都将终止这个模拟。

```
always@(posedge clk)
begin

if(creq1_flag && creq2_flag && creq3_flag &&
creq12_flag && creq23_flag && creq31_flag &&
creq123_flag && m1_t1_r_flag && m1_t1_w_flag
&& m1_t0_r_flag && m1_t0_w_flag && m2_t1_r_flag
&& m2_t1_w_flag && m2_t0_r_flag && m2_t0_w_flag
&& m3_t1_r_flag && m3_t1_w_flag && m3_t0_r_flag
&& m3_t0_w_flag)

begin

$display("FC: All possible request scenarios
covered 3 times\n");

$display("FC: All possible transactions
covered 3 times\n");

$finish();

end
end
```

3. 高级覆盖选项

有另一种数据点可以用来衡量系统的功能覆盖。

“目标设备的每个内存位置都应该被每个主控设备至少写入一次和从中读出一次”。

这个信息需要穷举测试才能获得。每个主控设备应该监控目标设备的每个地址空间的使用情况。在做功能覆盖时 SVA 并不是唯一的选择。如果功能覆盖涉及到穷举测试计划的覆盖点，那么使用支持面向对象编程的测试平台语言，效率会更高。当运行长

的回归测试时，应该使用这样的穷举功能覆盖点。

2.4.2 功能覆盖小结

功能覆盖的衡量保证了对所有必需检验的情景的测试。这种衡量能有效地用来控制模拟环境。一种方法是当达到功能覆盖的目标时终止模拟。在本节实例系统中，可以观察到下面的结果：

- 在测试平台中随机事务的数量默认设置为 500。
- 完成“请求情景”中所示的请求情景，只需要执行 46 个事务就终止了模拟。
- 完成“请求情景”中所示的请求情景和“主控设备到目标设备的事务”中所示的主控设备到目标设备的事务，只执行了 63 个事务就终止了模拟。

获得的功能覆盖的数据也能用来动态重定向测试平台。在随机的测试平台里，可以用一些约束来控制生成的事务的类型。在模拟开始时，为了实现随机分布，给这些约束分配了某些权重。在模拟中根据获得的功能覆盖的信息，动态调整这些权重，可以更快地达到功能覆盖的目标。

2.5 用于创建事务日志的 SVA

SVA 可以用来创建优秀日志文件。模拟中，SVA 检验器侦听任何违背设计属性的情况。如果检验器能记录下它侦听到的信息，那么它也可以被称为“监视器”。在一个复杂的系统中，它对按照时间顺序生成事务的日志是非常有帮助的。在我们的实例系统中，SVA 生成一个关于所有读和写事务的日志，记录下事务在什么设备之间发生和何时发生，是非常重要的调试资源。

SVA 有一个选项可以在检验器范围内使用类似 Verilog 的功能。每个检验器的执行块或者覆盖语句能有效地创建日志文件。创建日志文件的一种方法是针对一个断言的成功或者一个覆盖语句显示信息，另外一种方法是调用一个任务(task)或函数(function)。调用一个任务或函数扩展了 SVA 检验器的功能。除了

在一个任务内显示信息外，也能有效地进行数据检验。下面的代码显示了这个实例系统的事务日志是如何按照时间顺序创建的。

```
// open a file to document transactions

integer h_mt;
initial
begin
    h_mt = $fopen("mt.dat");
end

// calling task for documentation

`ifndef slv_doc

c_m1t1w_doc:
    cover property(p_m1t1w) master_xaction(1, 1);
c_m1t1r_doc:
    cover property(p_m1t1r) master_xaction(1, 1);
c_m1t2w_doc:
    cover property(p_m1t0w) master_xaction(1, 0);
c_m1t2r_doc:
    cover property(p_m1t0r) master_xaction(1, 0);
c_m2t1w_doc:
    cover property(p_m2t1w) master_xaction(2, 1);
c_m2t1r_doc:
    cover property(p_m2t1r) master_xaction(2, 1);
c_m2t2w_doc:
    cover property(p_m2t0w) master_xaction(2, 0);
c_m2t2r_doc:
    cover property(p_m2t0r) master_xaction(2, 0);
c_m3t1w_doc:
    cover property(p_m3t1w) master_xaction(3, 1);
c_m3t1r_doc:
    cover property(p_m3t1r) master_xaction(3, 1);
c_m3t2w_doc:
    cover property(p_m3t0w) master_xaction(3, 0);
c_m3t2r_doc:
    cover property(p_m3t0r) master_xaction(3, 0);

`endif

task master_xaction(
```

```

input int m_identity, input int t_identity);

integer i;

begin

if(data[8])
begin
    for(i=0; i<8; i++)
    begin

        $fwrite(h_mt, "WRITE:
Master %0d writing to Target %0d = %0d at
%0t\n", m_identity, t_identity, data[7:0], $time);

        @(posedge clk);
    end
    end

    if (!data[8])
    begin
        @(posedge clk);
        for(i=0; i<8; i++)
        begin
            $fwrite(h_mt, "READ:
Master %0d reading from Target %0d = %0d at
%0t\n", m_identity, t_identity, datao, $time);

            @(posedge clk);
        end
        end
    end

endtask

```

“主控设备到目标设备的事务”小节中定义的功能覆盖的属性被重用以创建事务日志。如果覆盖语句成功，调用一个名为“master_xaction”的任务。这个任务要有两个输入参数，一个用来确定主控设备，另一个用来确定目标设备。通过使用这些参数，可以创建一个通用的任务来精确记录事务。

事务被记录到一个独立的文件称为“mt.dat”。在模拟开始的

时候，用 \$fopen 语句可以打开这个文件。一旦调用这个任务，这个任务要么执行代码的读出模块或代码的写入模块。由于我们的实例系统按照 8 字节的整数倍进行一次读或写，所以在这个任务中用了一个“for”循环。它循环 8 次，每次循环对应的读或写的数据被 \$fwrite 语句记入到文件“mt.dat”中。下面是用这些代码创建的实例系统的日志的一部分。

```
WRITE: Master 1 writing to Target 1 = 72 at 775
WRITE: Master 1 writing to Target 1 = 77 at 825
WRITE: Master 1 writing to Target 1 = 95 at 875
WRITE: Master 1 writing to Target 1 = 37 at 925
WRITE: Master 1 writing to Target 1 = 216 at 975
WRITE: Master 1 writing to Target 1 = 184 at 1025
WRITE: Master 1 writing to Target 1 = 198 at 1075
WRITE: Master 1 writing to Target 1 = 182 at 1125
READ: Master 3 reading from Target 1 = 72 at 1725
READ: Master 3 reading from Target 1 = 77 at 1775
READ: Master 3 reading from Target 1 = 95 at 1825
READ: Master 3 reading from Target 1 = 37 at 1875
READ:Master 3 reading from Target 1 = 216 at 1925
READ:Master 3 reading from Target 1 = 184 at 1975
READ:Master 3 reading from Target 1 = 198 at 2025
READ:Master 3 reading from Target 1 = 182 at 2075
```

事务的日志可以做得更高级，更有助于调试用户的应用程序。注意这部分代码包含在‘ifdef - ’ endif 块中。在长的回归测试中，不需要这么详细的事务日志，因此应该有条件地包括进去。

2.6 用于 FPGA 原型测试的 SVA

目前存在多种多样的高级验证方法，它们有助于更快地发现缺陷。在这些方法中，约束随机测试平台和断言是其中的重要一块。为了确信所有可能的功能都已经正确测试到而写出成千的测试是很常见的。虽然在 RTL 验证阶段能发现绝大多数缺陷，在已实现了的门级验证中仍发现功能的缺陷也是很常见的。门级模拟一直是一个性能瓶颈，将来也是如此。在门级运行所有在 RTL 验

证阶段开发的测试不是很实际。由于门级的模拟相当慢，越来越多的验证工作组依靠其他的验证方法，如：形式验证(formal verification)、FPGA 原型(prototype)测试等，如图 2-14 所示。在实际的硅片上进行验证，可以很大程度上加快验证过程。它允许在实际的硅片上穷举地运行 RTL 阶段开发的回归测试。

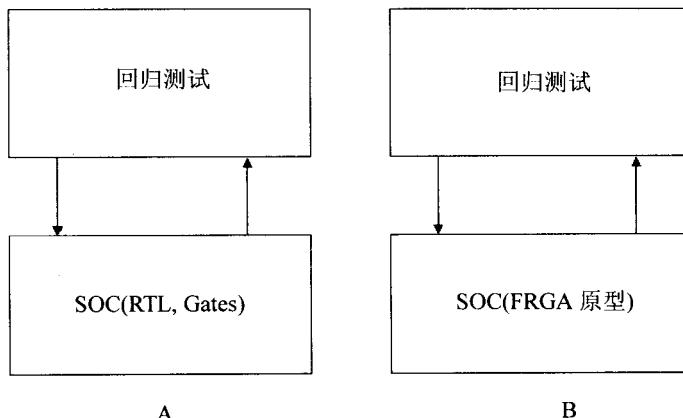


图 2-14 FPGA 原型测试

在实际的硅片原型上运行测试的一个最主要的挑战是调试。在这方面 SVA 能有很大的帮助。通过综合检验器与设计，可以使调试过程更加容易一点。检验器是按照功能规范而写的，有它们监视实际硅片中的设计将使验证受益匪浅。为了适应这些断言，设计需要稍作更改。如果一个断言失败了，设计要用一个输出端口通知外部环境。使用断言的执行块，输出端口的结果就能被更新。在绝大多数实时测试中，可以在这些输出端口设置断点，当这些调试端口的其中一个失败时，可以停止验证，以便进行更深入的分析。实例系统的主控设备如图 2-2，它只包含了设计相关的默认端口。主控设备的 Verilog 代码实例如下。

```

module master (ask_for_it, clk, req, gnt, frame,
 irdy, trdy, data_c, r_sel, data_o);

  input clk, gnt, ask_for_it;
  input [1:0] trdy;

```

```
output req, frame, irdy, r_sel;
output [8:0] data_c;
input [7:0] data_o;

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;

// functional description of master

// Block level SVA checks

endmodule
```

块级断言应该成为设计的一部分，这样有助于 FPGA 的原型测试。每个块级断言都应该有一个与之相连的调试输出端口。如果断言失败，调试输出端口应该被断言。下面描述的代码显示这是如何实现的。

```
module master (ask_for_it, clk, req, gnt, frame,
               irdy, trdy, data_c, r_sel, data_o,
               a_master_start1_flag, a_master_start2_flag,
               a_master_stop1_flag, a_master_stop2_flag,
               a_master_data1_flag, a_master_data2_flag,
               a_master_datao1_flag, a_master_datao2_flag);

    input clk, gnt, ask_for_it;
    input [1:0] trdy;
    output req, frame, irdy, r_sel;
    output [8:0] data_c;
    input [7:0] data_o;

    // debug pins for FPGA prototyping
    output a_master_start1_flag;
    output a_master_start2_flag;
    output a_master_stop1_flag;
    output a_master_stop2_flag;
    output a_master_data1_flag;
    output a_master_data2_flag;
    output a_master_datao1_flag;
    output a_master_datao2_flag;

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;
```

```
// functional description of master

// Block level checks for prototype debugging

`ifdef master_debug

d_a_master_start1:
    assert property(p_master_start1)
else
    a_master_start1_flag = 1'b1;
d_a_master_start2:
    assert property(p_master_start2)
else
    a_master_start2_flag = 1'b1;
d_a_master_stop1:
    assert property(p_master_stop1)
else
    a_master_stop1_flag = 1'b1;
d_a_master_stop2:
    assert property(p_master_stop2)
else
    a_master_stop2_flag = 1'b1;
d_a_master_data1:
    assert property(p_master_data1)
else
    a_master_data1_flag = 1'b1;
d_a_master_data2:
    assert property(p_master_data2)
else
    a_master_data2_flag = 1'b1;
d_a_master_datao1:
    assert property(p_master_datao1)
else
    a_master_datao1_flag = 1'b1;
d_a_master_datao2:
    assert property(p_master_datao2)
else
    a_master_datao2_flag = 1'b1;

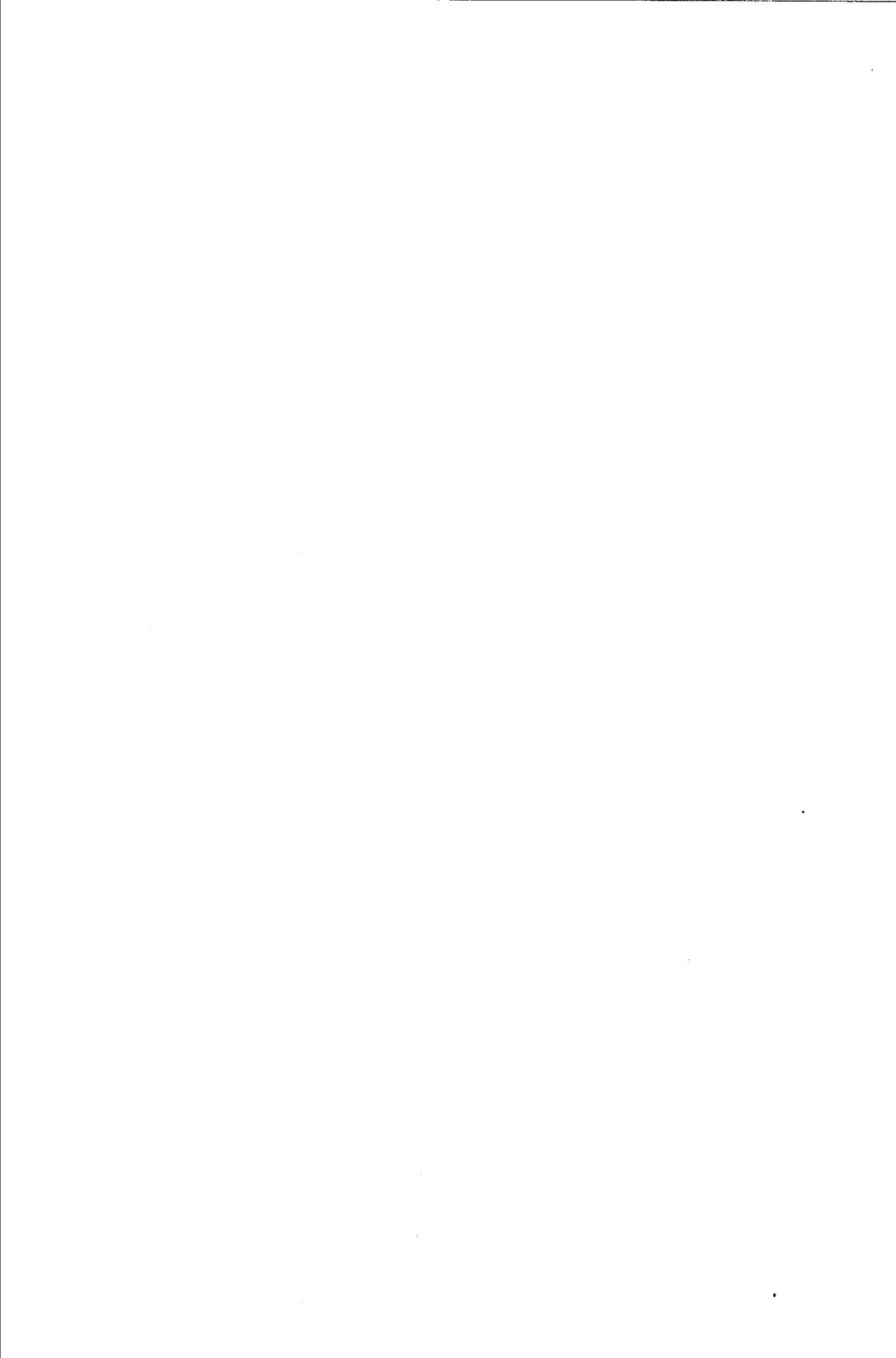
`endif

endmodule
```

注意，当出现失败时，对应的输出端口的标识将被断言。由于这些断言是并发的，它们会在每个时钟边缘寻找有效的起始点。如果在断言失败时硅片测试机制不能提供一种方法设置断点，那么就要求这个失败能被锁存。否则，如果在后续的时钟周期断言成功了，这个失败就会丢失。

2.7 SVA 模拟方法的小结

- SVA 对测试平台环境的补充使动态模拟的效率更高。
- 设计者非常熟悉设计的内部功能，因此，他们应该把 SVA 检验器内嵌到相应的设计模块中。
- 验证工程师负责集成和验证系统，他们应该添加系统级断言，来彻底验证接口协议。
- 验证工程师应该能从验证环境控制或配置块级断言(如果需要，他能打开或关闭断言)。
- 采用 SVA，几乎不费吹灰之力就可以收集到功能覆盖的信息。应该有效利用这些信息来建立反应测试平台。
- 由于 SVA 在模拟全过程中监视设计的协议，它能用来创建信息丰富的日志文件。
- 按照可综合的编码风格编写 SVA 检验器，使 SVA 检验器能成为网表的一部分，可以用于调试原型或混合模拟中的失败。





SVA 在有限状态机中的应用

在任何设计中，有限状态机(FSM)都是主要的控制模块。FSM 通过生成对应的控制信号，有助于设计有序地从一个状态转换到另一个状态。生成控制信号的另一种方法是结合计数器和胶合逻辑(glue logic)，但是这种方法缺乏好的设计结构，而且也难以调试。由于设计的每个状态通常都定义得很清晰，FSM 能为控制信号和调试提供很好的硬件基础。

有限状态机可以分成两类：

摩尔状态机(Moore state machine)——摩尔有限状态机的输出只是当前状态的函数。

米勒状态机(Mealy state machine)——米勒有限状态机的一个或更多的输出是当前状态和一个或更多输入的函数。

我们可以使用不同的编码风格来描述有限状态机的状态。最流行的编码风格是 one-hot 编码，其中每一位的寄存器表示一个状态，已经证明它是最快的结构。如果 FSM 的状态太多，那么 one-hot 编码会被综合成一个相当大的硬件，对于这种情况，二进制编码更为合适。另一种常用来描述 FSM 的编码是格雷码编码(gray coding)。

由于 FSM 控制了整个设计的功能，因此应该对它进行仔细的验证。最常见的检验是确认状态之间的转换正确发生，且不违背任何时序上的要求。SVA 可以有效地用来做这样的检验。

3.1 设计例子——FSM1

在这一节，我们分析一个简单的线性 FSM，它更像一个移位计数器。这个 FSM 线性有序地生成设计的控制信号，因此用 SVA 检验很容易验证。

3.1.1 FSM1 的功能描述

在这个 FSM 中有 16 种状态，把它们进行如下编码：

```
IDLE = 16'd1  
GEN_BLK_ADDR = 16'd2  
WAIT6 = 16'd4  
NEXT_BLK = 16'd8  
WAIT0 = 16'd16  
CNT1 = 16'd32  
WAIT1 = 16'd64  
CNT2 = 16'd128,  
WAIT2 = 16'd256  
CNT3 = 16'd512  
WAIT3 = 16'd1024  
CNT4 = 16'd2048  
WAIT4 = 16'd4096  
CNT5 = 16'd8192  
WAIT5 = 16'd16384  
CNT6 = 16'd32768
```

这个 FSM 是用 one-hot 编码方式进行编码。图 3-1 所示的是 FSM1 的状态图：

一旦重新复位，FSM 移到 IDLE 状态，并在那里等待一个正确的“get_data”信号。

- 一旦得到一个“get_data”信号，FSM 移到 GEN_BLK_ADDR 状态。FSM 呆在这个状态，直到完成 64 个读地址的生成(一个内部计数器跟踪 64 个时钟周期)。
- 64 个时钟周期后，FSM 移到 WAIT0 状态。从这点开始，FSM 保持每个时钟周期前移一个状态。
- CNT 开头的状态(CNT*)是剩余部分设计的输出控制信号产生的地方。
- WAIT 开头的状态(WAIT*)用来在控制信号(latch_en, dp1_en, dp2_en, dp3_en, dp4_en, wr)间生成 2 个周期的间隔。
- 当 FSM 移到 NEXT_BLK 状态，它需要决定移到哪个状态。

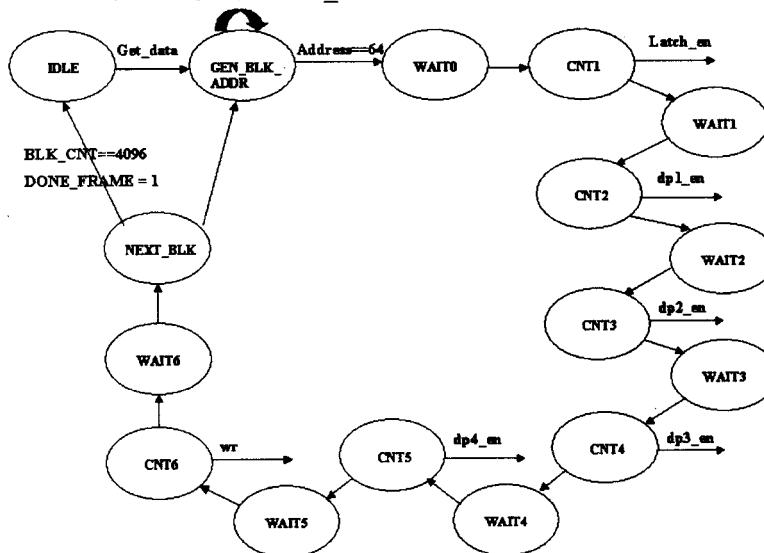


图 3-1 FSM1 的状态图

- 如果内部寄存器“blk_cnt”的值已经达到 4096，FSM 移到 IDLE 状态。这就表示整个数据帧已经处理完，设计等待一个新的帧。当一个新的“get_data”信号到达时，FSM 再次从 GEN_BLK_ADDR 开始相同的状态转换过程。
- 当 FSM 在状态 NEXT_BLK 时，如果内部寄存器“blk_cnt”的值没有达到 4096，FSM 将会回到 GEN_BLK_ADDR 状态。

态，等待 64 个新地址的产生，然后再次移到 CNT* 状态产生控制信号。

例子 3.1 FSM1 的代码实例

```
module fsm (get_data, reset_, clk, rd, rd_addr, data,
done_frame, latch_en, sipo_en, dp1_en, dp2_en, dp3_en,
dp4_en, wr);

    input get_data;
    input reset_;
    input clk;
    input [7:0] data;

    output rd;
    output logic sipo_en, latch_en;
    output logic dp1_en, dp2_en, dp3_en, dp4_en, wr;
    output logic done_frame;
    output [17:0] rd_addr;

    logic [5:0] addr_cnt;
    logic [11:0] blk_cnt;
    logic [3:0] pipeline_cnt;
    logic rd;
    logic [17:0] rd_addr;
    logic enable_cnt, enable_dly_cnt, enable_blk_cnt;

    assign done_frame = (blk_cnt == 4095);
    assign sipo_en = rd;

    enum bit[15:0] {IDLE = 16'd1,
                    GEN_BLK_ADDR = 16'd2,
                    DLY = 16'd4,
                    NEXT_BLK = 16'd8,
                    WAIT0 = 16'd16,
                    CNT1 = 16'd32,
                    WAIT1 = 16'd64,
                    CNT2 = 16'd128,
                    WAIT2 = 16'd256,
                    CNT3 = 16'd512,
                    WAIT3 = 16'd1024,
                    CNT4 = 16'd2048,
                    WAIT4 = 16'd4096,
```

```
CNT5 = 16'd8192,
WAIT5 = 16'd16384,
CNT6 = 16'd32768} n_state, c_state;
// assign the different control signals

assign latch_en = (c_state == CNT1);
assign dp1_en = (c_state == CNT2);
assign dp2_en = (c_state == CNT3);
assign dp3_en = (c_state == CNT4);
assign dp4_en = (c_state == CNT5);
assign wr = (c_state == CNT6);

// 64bit counter to generate read address

always_ff @(posedge clk)
if (!reset_ || !enable_cnt)
    addr_cnt <= 0;
else if (enable_cnt)
    addr_cnt <= addr_cnt + 1;
else
    addr_cnt <= addr_cnt;

// 4096 bit counter

always_ff @(posedge clk)
if (!reset_)
    blk_cnt <= 0;
else if ((c_state == NEXT_BLK) && enable_blk_cnt)
    blk_cnt <= blk_cnt + 1;
else
    blk_cnt <= blk_cnt;

always_ff @(posedge clk)
if (!reset_)
    c_state <= IDLE;
else
    c_state <= n_state;

always @(*)
begin
rd <= 0;
enable_cnt <= 0;
//enable_dly_cnt <= 0;
case (c_state)
```

```
IDLE:  begin
    enable_blk_cnt <= 0;
    if(get_data)
        n_state <= GEN_BLK_ADDR;
    else
        n_state <= IDLE;
end

GEN_BLK_ADDR:  begin
    enable_cnt <= 1;
    rd <= 1;
    rd_addr <= {blk_cnt, addr_cnt};
    if (addr_cnt == 63) begin
        //enable_dly_cnt <= 1;
        n_state <= WAIT0;
    end
    else begin
        n_state <= GEN_BLK_ADDR;
        //pipeline_cnt <= 0;
    end
end

WAIT0:  n_state <= CNT1;
CNT1:  n_state <= WAIT1;
WAIT1:  n_state <= CNT2;
CNT2:  n_state <= WAIT2;
WAIT2:  n_state <= CNT3;
CNT3:  n_state <= WAIT3;
WAIT3:  n_state <= CNT4;
CNT4:  n_state <= WAIT4;
WAIT4:  n_state <= CNT5;
CNT5:  n_state <= WAIT5;
WAIT5:  n_state <= CNT6;
CNT6:  n_state <= DLY;

DLY:  begin
    enable_blk_cnt <= 1;
    n_state <= NEXT_BLK;
end

NEXT_BLK:  begin
    enable_blk_cnt<=1;
    if (blk_cnt == 4095)
```

```

        n_state <= IDLE;
      else
        n_state <= GEN_BLK_ADDR;
      end
    endcase
  end

endmodule

```

图 3-2 所示的是从 GEN_BLK_ADDR 到 CNT*/WAIT* 的状态转换。

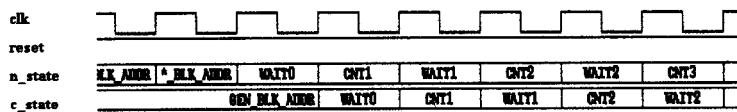


图 3-2 FSM1 的波形 A

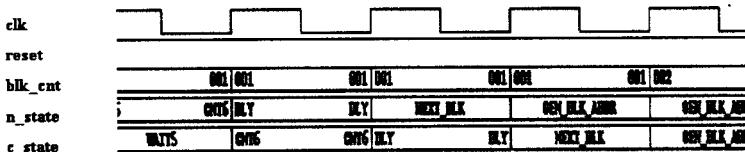


图 3-3 FSM1 的波形 B

图 3-3 显示的是，如果寄存器“blk_cnt”的值没有到达 4096，FSM 将会从 NEXT_BLK 状态循环回 GEN_BLK_ADDR 状态。当“blk_cnt”的值达到 4096，FSM 将会从 NEXT_BLK 状态跳出循环，移到 IDLE 状态。

3.1.2 FSM1 的 SVA 检验器

为了彻底验证 FSM1，需要做下面一些检验。

FSM1_chk1：无论输入情况如何，FSM1 将总是维持 one-hot。

由于 FSM1 是基于 one-hot 编码的，因此它应该总是只有一个状态位被断言。如果不是，那么这个 FSM 就不是真正的 one-hot，而且可能不能生成期望的控制信号。这一点用一个内嵌的任务，即 SVA 语言内部定义的\$countones 或 \$onehot，就可以测试到。

```

property p_onehot;
  @ (posedge clk) (reset_) |->

```

```

    ($countones(n_state) == 1);
endproperty

```

```

a_onehot: assert property(p_onehot);
c_onehot: cover property(p_onehot);

```

FSM1_chk2: 如果当前的状态为“IDLE”，并且“get_data”也被断言，那么下一个状态是“GEN_BLK_ADDR”，64 个周期后，下一个状态应该是“WAIT0”。

基于“IDLE”状态和“get_data”信号，FSM 开始状态转换。当 FSM 达到 GEN_BLK_ADDR，它要在这个状态维持 64 个时钟周期。

```

sequence s_trans1;
  (c_state == IDLE) ##1
  ((c_state == GEN_BLK_ADDR) [*64]) ##1 (c_state
  == WAIT0);
endsequence

```

```

property p_trans;
  @ (posedge clk)
  (reset_ && $rose(get_data)) |->
    (reset_) throughout (s_trans1);
endproperty

a_trans: assert property (p_trans);
c_trans: cover property (p_trans);

```

序列“s_trans1”验证的是：如果 FSM1 当前在 IDLE 状态，那么一个周期后它将转换到 GEN_BLK_ADDR 状态。FSM 将会维持在 GEN_BLK_ADDR 状态 64 个周期(用重复操作符(*)来验证)，并且一个周期后，FSM 将移到 WAIT0 状态。贯穿这个属性时复位(reset)必须处于未激活态。

图 3-4 显示的是属性“a_trans”的结果。当“get_data”信号的上升沿到来时，检验器被激活，并且在波形中的同一点显示出一个匹配的成功点。虽然检验器直到达到 WAIT0 状态都是激活的，只有在检验器的起始点显示了成功。在每个时钟的上升沿，检验器都寻找“get_data”信号的上升沿。如果没有找到，那么检

验器默认假设检验成功，这就是第1章中讨论的空成功。

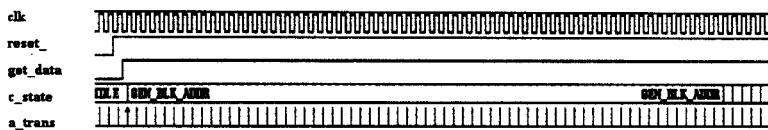


图 3-4 FSM1_chk2 的波形

FSM1_chk3：如果当前的状态是“WAIT0”，那么每一个时钟周期后 FSM 将会有序地从一个状态转换到另一个状态，除非 FSM 复位。

在每个 CNT* 状态间有一个 WAIT* 状态，因此 FSM 从一个 CNT* 状态到另一个 CNT* 状态要两个时钟周期。FSM 从 CNT1 状态以线性方式转换到 CNT6 状态。惟一可能的路径如下：

CNT1 -> CNT2 -> CNT3 -> CNT4 -> CNT5 -> CNT6

```

sequence s_trans3;
  ##1 (c_state == CNT1) ##2 (c_state == CNT2)
  ##2 (c_state == CNT3) ##2 (c_state == CNT4) ##2
  (c_state == CNT5) ##2 (c_state == CNT6);
endsequence

property p_linear_trans;
  @ (posedge clk)
  ((reset_) && (c_state == WAIT0)
  && ($past(c_state) == GEN_BLK_ADDR)) |->
  s_trans3;
endproperty

a_linear_trans: assert property (p_linear_trans);
c_linear_trans: cover property (p_linear_trans);

```

序列“s_trans3”验证的是：如果 FSM 当前在 WAIT0 状态，并且前一周期 FSM 在 GEN_BLK_ADDR 状态，那么一个周期后 FSM 移到 CNT1 状态，一个周期后移到 WAIT1 状态，这样依次转换，直到状态 CNT6。图 3-5 显示的是属性 p_linear_trans 的模拟结果。

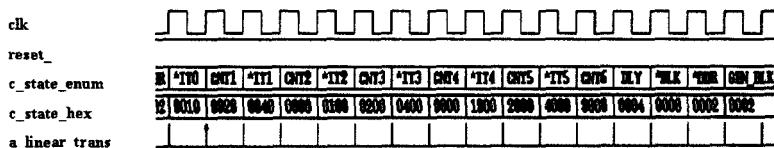


图 3-5 FSM1_chk3 的波形

FSM1_chk4: 确认 FSM1 从 NEXT_BLK 状态分别转换到 IDLE 和 GEN_BLK_ADDR 状态至少一次。

这个检验所做的是功能覆盖的一部分，在输入测试向量的作用下确保 FSM 的所有路径都转换过。

```

sequence s_trans2;
    ##63 (c_state == GEN_BLK_ADDR) ##1
    (c_state == WAIT0);
endsequence

property p_frame;
    @ (posedge clk)
    ((reset_) && (c_state == GEN_BLK_ADDR) &&
    ($past (c_state)== IDLE) || 
    ($past(c_state == NEXT_BLK))) |->
        s_trans2 ##0 s_trans3;
endproperty

a_frame: assert property(p_frame) cnt++;
c_frame: cover property(p_frame);

property p_complete_frame;
    @ (posedge clk)
    ((cnt == 16'd4095) &&reset_ &&
    (c_state==CNT6)) |->
        done_frame;

endproperty

a_complete_frame:
    assert property(p_complete_frame)
    $display ("A complete frame has been transferred
    \n");

```

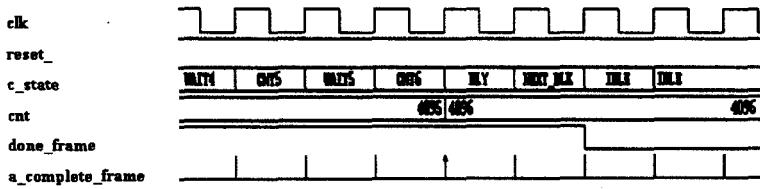


图 3-6 FSM1_chk4 的波形

属性“p_frame”验证 FSM1 从 IDLE 状态开始的所有状态转换过程。在做这个检验时，每次属性“p_frame”成功，执行块里的一个局部变量“cnt”增加 1。当变量“cnt”的值达到 4095 时，数据的所有块都已经处理完，并且断言控制信号“done_frame”。在一帧全部测试结束时，用检验“a_complete_frame”的执行块来显示结果。图 3-6 显示的是属性 p_complete_frame 的模拟结果。

两个单独的属性“p_frame_path1”和“p_frame_path2”用来确信模拟中 FSM1 的所有路径都被覆盖到。

```

property p_frame_path1;
  @ (posedge clk)
    ((reset_) && (c_state == GEN_BLK_ADDR) &&
     ($past(c_state == NEXT_BLK))) |->
      s_trans2 ##0 s_trans3;
endproperty

c_frame_path1 : cover property (p_frame_path1);

property p_frame_path2;
  @ (posedge clk)
    ((reset_) && (c_state == GEN_BLK_ADDR) &&
     ($past(c_state == IDLE))) |->
      s_trans2 ##0 s_trans3;
endproperty

c_frame_path2 : cover property (p_frame_path2);

```

3.2 设计实例——FSM2

这一节讨论稍微复杂点的 FSM。小节 3.1 讨论的 FSM 是线性

的，没有多个路径转换到一个特定状态。FSM2 有较少的状态但是它有更多的路径转换到一个特定状态。在析取需要做的检验时有了一点挑战性。

3.2.1 FSM2 的功能描述

FSM2 起的是一个仲裁器的作用。在任何给定时刻，FSM2 能在三个主控设备间仲裁。所有主设备都能请求授予总线，仲裁器按照轮转(round robin)方式决定哪一个主控设备获得总线。一旦一个主控设备得到许可，它将利用总线来执行一些事务。在事务结束时，主控设备会通知仲裁器，然后释放总线。一旦总线被释放，如果主控设备有挂起的事务，它们可以再次请求总线。关键点是确保仲裁器不会让某个主控设备一直得不到许可。

这个 FSM 有七种可能的状态，显示如下：

```
IDLE = 7'b00000001  
MASTER1 = 7'b00000010  
IDLE1 = 7'b00000100  
MASTER2 = 7'b00001000  
IDLE2 = 7'b00100000  
MASTER3 = 7'b01000000  
IDLE3 = 7'b10000000
```

这个 FSM 是用 one-hot 编码方式编码。图 3-7 显示的是 FSM2 的状态图。

- 当状态机复位，移到 IDLE 状态。
- 当状态机在 IDLE 状态时，它寻找任何一个主控设备请求使用总线的“req”。许可按照一定优先级编码，例如，当 FSM 在 IDLE 状态，如果所有三个主控设备都请求总线，那么“master1”将得到总线。
- 当状态机在 MASTER*状态，状态机要断言相应的主控设备的“gnt”信号。
- 一旦主设备使用完总线，它通过断言“done”信号告知仲裁器，然后 FSM 移到这个主控设备相应的 IDLE*状态。

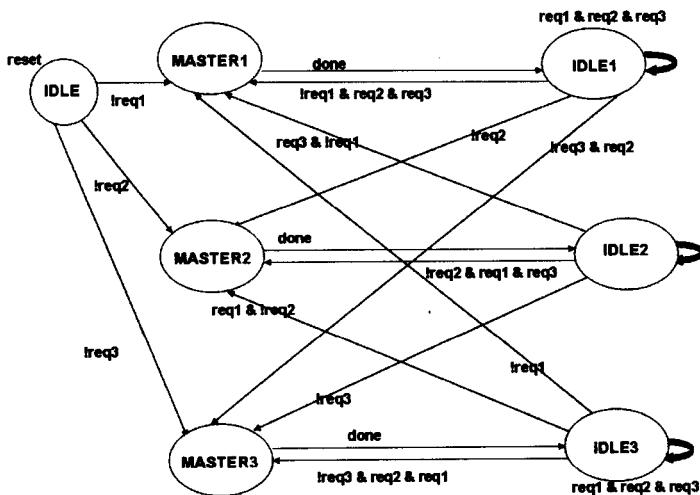


图 3-7 FSM2 的状态图

- 当 FSM 在 IDLE1 状态，它将按照 master2、master3 然后 master1 的顺序寻找“req”。
- 当 FSM 在 IDLE2 状态，它将按照 master3、master1 然后 master2 的顺序寻找“req”。
- 当 FSM 在 IDLE3 状态，它将按照 master1、master2 然后 master3 的顺序寻找“req”。

例子 3.2 FSM2 的代码实例

```

module bus_arbiter(clk, reset, frame, irdy,
req1, req2, req3, gnt1, gnt2, gnt3);

input logic clk, reset, frame, irdy;
input logic req1, req2, req3;

output logic gnt1, gnt2, gnt3;

enum bit [6:0] {IDLE = 7'b00000001,
MASTER1 = 7'b00000010,
IDLE1 = 7'b00000100,
MASTER2 = 7'b00001000,
IDLE2 = 7'b00010000,
MASTER3 = 7'b01000000,
  
```

```
IDLE3 = 7'b1000000} next, state;

logic done, gnt1, gnt2, gnt3;

/* define glue signals */

assign done = frame && irdy;

/* state register code */

always@(posedge clk or negedge reset)
begin
    if(!reset)
        state <= IDLE;
    else
        state <= next;
end

/* next state combinational logic */

always@(*)
begin
    next = IDLE;
    case(state)

        IDLE:
            if (req1 == 1'b0)
                next <= MASTER1;
            else if (req1 == 1'b1 & req2 == 1'b0)
                next <= MASTER2;
            else if (req3 == 1'b0 & req1 == 1'b1)
                next <= MASTER3;
            else
                next <= IDLE;

        MASTER1:
            if(!done)
                next <= MASTER1;
            else
                next <= IDLE1;

        IDLE1:
            if(req2 == 1'b0 )
```

```
        next <= MASTER2;
    else if (req3 == 1'b0 & req2 == 1'b1)
        next <= MASTER3;
    else if (req3 == 1'b1 & req1 == 1'b0 & req2
    == 1'b1)
        next <= MASTER1;
    else
        next <= IDLE1;

MASTER2:
if (!done)

    next <= MASTER2;
else
    next <= IDLE2;

IDLE2:
if (req3 == 1'b0)
    next <= MASTER3;
else if (req3 == 1'b1 & req1 == 1'b0)
    next <= MASTER1;
else if (req1 == 1'b1 & req2 == 1'b0)
    next <= MASTER2;
else
    next <= IDLE2;
MASTER3:
if (!done)
    next <= MASTER3;
else
    next <= IDLE3;

IDLE3:
if (req1 == 1'b0)
    next <= MASTER1;
else if (req1 == 1'b1 & req2 == 1'b0)
    next <= MASTER2;
else if (req2 == 1'b1 & req3 == 1'b0)
    next <= MASTER3;
else
    next <= IDLE3;
endcase

end
```

```

/* output generating statements */

assign gnt1 = ((state == MASTER1)) ? 0 : 1;
assign gnt2 = ((state == MASTER2)) ? 0 : 1;
assign gnt3 = ((state == MASTER3)) ? 0 : 1;

endmodule

```

图 3-8 显示的是一个 FSM2 实例波形。为了方便起见，状态编码同时用枚举的值和十六进制值表示。状态值*1 表示状态是 MASTER1，类似地，*2 表示 MASTER2，*3 表示 MASTER3。在标记 1，master2 和 master3 同时请求授予总线。在这一点，FSM 在 IDLE3 状态，因此授权给 master2。在标记 2，FSM 在 IDLE2 状态，master1 和 master3 请求总线，这次 master3 获得许可。授权给哪个总是由 FSM 当前在哪个 IDLE* 状态决定。

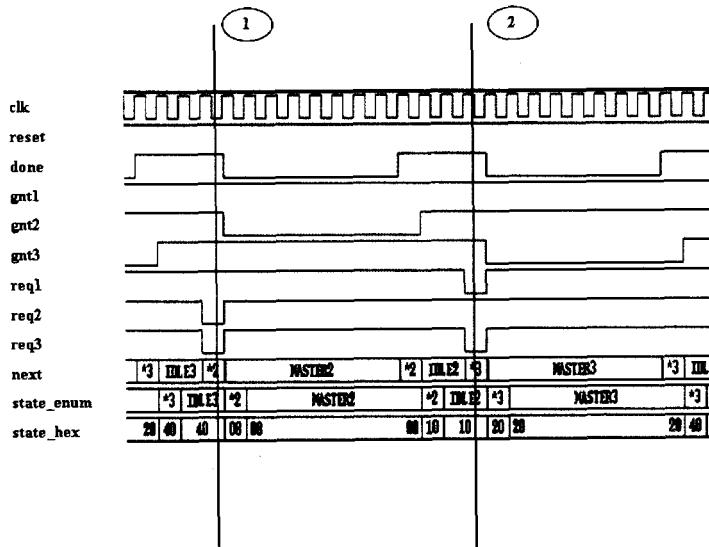


图 3-8 FSM2 的波形

3.2.2 FSM2 的 SVA 检验器

对一个这样的状态机，由于有许多路径可以到达一个特别的状态，因此首先要了解的是可能的合法路径有哪些？这可能是一个很困难的过程，它由状态机的复杂程度决定。第一步，建立一

个矩阵，把所有状态都在 X 轴和 Y 轴表示出来，然后在矩阵上用“Yes”或“No”表示从 X 轴的状态能否转换到 Y 轴的状态。当我们做好一个如上所述的矩阵时，我们可以开始把 SVA 检验分类。

- 基于矩阵，如果一个状态被禁止转换到另一个状态，那么应该用 SVA 检验来验证。
- 测试平台应该覆盖到所有可能的合法状态转换。这一点可以通过在 SVA 属性使用“cover”语句来衡量。从代码覆盖工具可以得到同样的信息。

基于表 3-1 所示的矩阵分析，需要编写下面的检验来彻底验证 FSM2。

表 3-1 FSM2 状态转换的矩阵图

	IDLE	M1	I1	M2	I2	M3	I3
IDLE	Y	Y	N	Y	N	Y	N
M1	Y	Y	Y	N	N	N	N
I1	Y	Y	Y	Y	N	Y	N
M2	Y	N	N	Y	Y	N	N
I2	Y	Y	N	Y	Y	Y	N
M3	Y	N	N	N	N	Y	Y
I3	Y	Y	N	Y	N	Y	Y

FSM2_chk1: FSM2 的行为应该总是一个 one-hot 状态机。

```

property p_fsm2_encoding;

@(posedge clk) $onehot(state);
endproperty

a_fsm2_encoding:
    assert property (p_fsm2_encoding);
c_fsm2_encoding:
    cover property (p_fsm2_encoding);

```

内嵌的函数\$onehot 可以用来确认任何时候状态寄存器只有一位是高位，这样就能证明这个 FSM 总是维持 one-hot。

FSM2_chk2: FSM 不能从 IDLE 转换到 IDLE1、IDLE2 或 IDLE3 状态。

```

property p_forbid_trans1;
@(posedge clk)

(((state == IDLE1) || (state == IDLE2) || 
(state == IDLE3)) && reset) |->
$past ((state == IDLE) == 0);
endproperty

a_forbid_trans1:assert property(p_forbid_trans1);

```

属性 “p_forbid_trans1” 就是用来验证这一点，如果当前状态是 IDLE1、IDLE2 或 IDLE3，那么 FSM 的前一周期的状态不能是 IDLE。

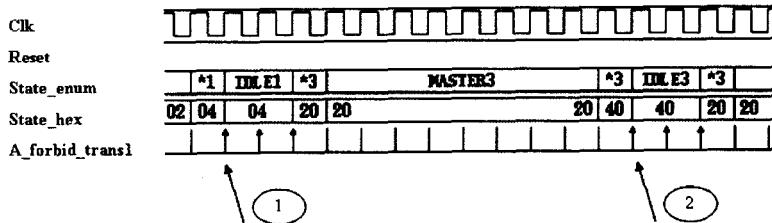


图 3-9 FSM2_chk2 的波形

图 3-9 显示的是检验 “a_forbid_trans1”的结果。在标记 1 显示的点上，FSM 当前是 IDLE1 状态，因为前一周期 FSM 在 MASTER1 状态而不是 IDLE 状态，所以属性通过。类似地，标记 2 显示的点上 FSM 在 IDLE3 状态，因为前一周期 FSM 在 MASTER3 状态而不是 IDLE 状态，所以属性通过。

FSM2_chk3:

FSM 不能从 MASTER1 状态到其他 MASTER 状态或 IDLE2 或 IDLE3。

FSM 不能从 MASTER2 状态到其他 MASTER 状态或 IDLE1 或 IDLE3。

FSM 不能从 MASTER3 状态到其他 MASTER 状态或 IDLE1

或 IDLE2。

这个检验确认，如果 FSM 在某个 MASTER* 状态，那么下一个状态将总是这个的 MASTER 状态对应的 IDLE* 状态。换言之，如果 FSM 当前在 MASTER1 状态，假设 FSM 没有被复位，那么 FSM 的下一个状态只能是 IDLE1，如果它转换到任何其他状态，那就是偏离(violation)。类似地，MASTER2 应该转换到 IDLE2，MASTER3 应该转换到 IDLE3。图 3-10 显示的是检验“a_forbid_trans2a”的结果。

```
property p_forbid_trans2a;
  @ (posedge clk)
    (((state == IDLE2) || (state == IDLE3) ||
      (state == MASTER2) || (state == MASTER3))
     && reset) |->
      $past ((state == MASTER1) == 0);
endproperty

a_forbid_trans2a:
  assert property(p_forbid_trans2a);
c_forbid_trans2a:
  cover property(p_forbid_trans2a);

property p_forbid_trans2b;
  @ (posedge clk)
    (((state == IDLE1) || (state == IDLE3) ||
      (state == MASTER1) || (state == MASTER3))
     && reset) |->
      $past ((state == MASTER2) == 0);
endproperty

a_forbid_trans2b:
  assert property(p_forbid_trans2b);
c_forbid_trans2b:
  cover property(p_forbid_trans2b);

property p_forbid_trans2c;
  @ (posedge clk)
    (((state == IDLE2) || (state == IDLE1) ||
      (state == MASTER2) || (state == MASTER1))
     && reset) |->
      $past (state == MASTER3) == 0);
```

```

endproperty

a_forbid_trans2c:
    assert property(p_forbid_trans2c);
c_forbid_trans2c:
    cover property(p_forbid_trans2c);

```

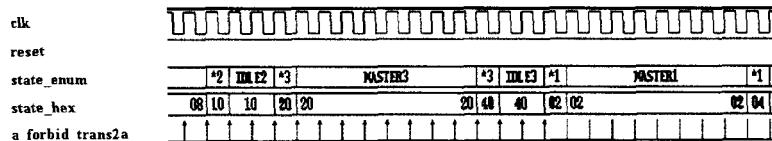


图 3-10 FSM2_chk3 的波形

FSM2_chk4:

FSM 不能从 IDLE1 状态到 IDLE2 或 IDLE3 状态。

FSM 不能从 IDLE2 状态到 IDLE1 或 IDLE3 状态。

FSM 不能从 IDLE3 状态到 IDLE1 或 IDLE2 状态。

这个检验确认，FSM 将总是从 IDLE*状态转换到 MASTER*状态，假设 FSM 在这之间没有复位。如果 FSM 从一个 IDLE*状态转换到另一个 IDLE*状态，那就是偏离。图 3-11 所示的是检验“p_forbid_trans3a”的结果。

```

property p_forbid_trans3a;
    @(posedge clk)
        (((state == IDLE2) || (state == IDLE3))
        && reset) |->
            $past (state== IDLE1) == 0;
endproperty

a_forbid_trans3a:
    assert property(p_forbid_trans3a);
c_forbid_trans3a:
    cover property(p_forbid_trans3a);

property p_forbid_trans3b;
    @(posedge clk)
        (((state == IDLE1) || (state == IDLE3))
        && reset) |->
            $past (state== IDLE2) == 0;

```

```

endproperty

a_forbid_trans3b:
    assert property(p_forbid_trans3b);
c_forbid_trans3b:
    cover property(p_forbid_trans3b);
property p_forbid_trans3c;
    @(posedge clk)
    ((state == IDLE1) || (state == IDLE2))
    && reset) |->
        $past (state== IDLE3) == 0);
endproperty

a_forbid_trans3c:
    assert property(p_forbid_trans3c);
c_forbid_trans3c:
    cover property(p_forbid_trans3c);

```

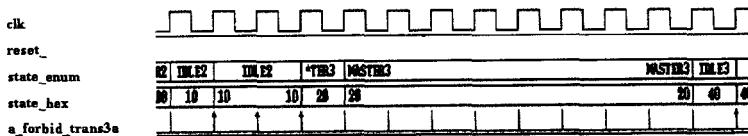


图 3-11 FSM2_chk4 的波形

FSM2_chk5：一旦有发到仲裁器的请求就应该有一个许可。

```

property p_req_gnt;
    @(posedge clk)
    (!req1 || !req2 || !req3) && reset) |->
        ##1 (!gnt1 || !gnt2 || !gnt3);
endproperty

a_req_gnt: assert property(p_req_gnt);
c_req_gnt: cover property(p_req_gnt);

```

属性“p_req_gnt”验证的就是这一点，如果任何主设备请求总线，那么在一个时钟周期内，其中一个“gnt”信号应该被断言。如果在一个周期内，许可没有到达，这是一个致命的错误。

图 3-12 所示的是检验“a_req_gnt”的结果。在波形中标记 1 所示的点，MASTER3 正在请求总线，并且在一个周期内信号“gnt3”被断言，因此检验通过。类似地，标记 2 所示的点，

MASTER2 和 MASTER3 同时请求总线，然后在一个周期内“gnt2”信号被断言，因此检验通过。

FSM2_chk6: 检验仲裁器公平与否。确信所有的主设备得到同等数量的许可机会。

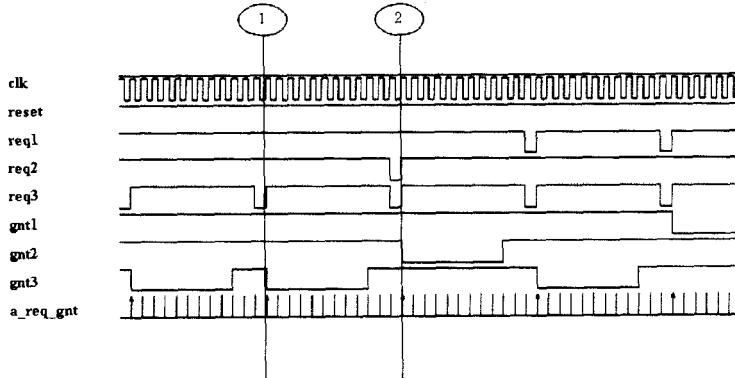


图 3-12 FSM2_chk5 的波形

```

property p_req_gnt_1;
  @(posedge clk) ((!req1 && reset)) |->
    ##1 !gnt1;
endproperty

c_req_gnt_1: cover property(p_req_gnt_1);

property p_req_gnt_2;
  @(posedge clk) ((!req2 && reset)) |->
    ##1 !gnt2;
endproperty

c_req_gnt_2: cover property(p_req_gnt_2);

property p_req_gnt_3;
  @(posedge clk) ((!req3 && reset)) |->
    ##1 !gnt3;
endproperty

c_req_gnt_3: cover property(p_req_gnt_3);

property p_req1;
  @(posedge clk) ($fell(req1) && reset);
endproperty

```

```
c_req1: cover property(p_req1);  
  
property p_req2;  
  @ (posedge clk)  ($fell(req2) && reset);  
endproperty  
  
c_req2: cover property(p_req2);  
  
property p_req3;  
  @ (posedge clk)  ($fell(req3) && reset);  
endproperty  
  
c_req3: cover property(p_req3);
```

这个检验是为了获得功能覆盖信息及验证仲裁器的公平性。三个属性 `p_req_gnt1`、`p_req_gnt2` 和 `p_req_gnt3` 用来计算一个主设备多少次可以真正得到一次许可。接下来的三个属性 `p_req1`、`p_req2` 和 `p_req3` 用来计算每个主控设备实际上请求了几次。通过在属性中使用覆盖语句，按照匹配的数量打印模拟的结果。在一个随机的测试环境的例子中，产生了下面的结果。

```
c_req_gnt_1, 10433 attempts, 288 match  
c_req_gnt_2, 10433 attempts, 290 match  
c_req_gnt_3, 10433 attempts, 291 match  
c_req1, 10433 attempts, 481 match  
c_req2, 10433 attempts, 474 match  
c_req3, 10433 attempts, 505 match
```

注意，每个主控设备请求总线大约 475 次，每个主控设备被授予总线的次数大约是 290 次。这就表明这个仲裁器很公平，没有让任何一个主控设备“挨饿”。

3.2.3 有时序窗口协议的 FSM2

在前一节中，当一个请求收到后的一个时钟周期内，FSM2 断言“gnt”信号。在这一节，假设仲裁器能在 2~5 个时钟周期内产生一个许可。新仲裁器的绝大多数协议的析取过程仍然相同，但一些检验中的时序需要调整。

```

assign req = !req1 || !req2 || !req3;
assign gnt = !gnt1 || !gnt2 || !gnt3;

property p_req_gnt_w;
    @(posedge clk) $rose(req) |>
        ##[2:5] $rose(gnt);
endproperty

a_req_gnt_w : assert property(p_req_gnt_w);

```

属性 `p_req_gnt_w` 寻找 “req” 信号的上升沿。“req” 信号是所有三个请求 “req1”、“req2” 和 “req3”的或操作的输出。当前提条件为 true，这个属性验证在 2~5 个时钟周期内 “gnt” 信号有一个上升沿出现。“gnt” 信号是所有三个 “gnt” 信号 “gnt1”、“gnt2” 和 “gnt3”的或操作的输出。功能覆盖语句同之前检验 `FSM2_chk6` 中所示的类似，这个新的基于时间窗的协议的功能覆盖语句很容易实现。图 3-13 所示的是检验 “a_req_gnt_w”的结果。

标记 “1s” 表示第一个发给仲裁器的有效请求。标记 “1e” 表示 5 个时钟周期后来了一个有效的 “gnt”，因此检验器成功。标记 “2s” 表示第二个发给仲裁器的有效请求，标记 “2e” 表示 2 个时钟周期后来了一个有效的 “gnt”，因此检验器成功。

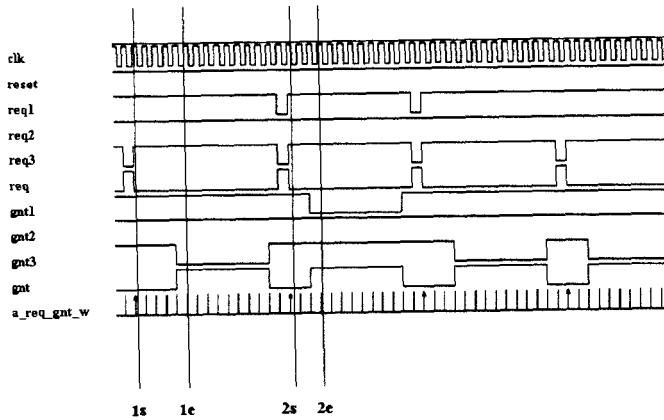


图 3-13 窗口检验波形

用户感兴趣的重要的功能覆盖数据可能是仲裁器的时延。仲裁器花费 2~5 个时钟周期响应主设备中的一个请求。了解仲裁器

对所有三个主设备的平均时延是否相同是很重要的。我们可以用 SVA 的 cover 语句来计算仲裁器对每个主设备的时延。

```
genvar s;
generate
  for (s=2; s<6; s++)
    begin: generic
      c_gnt_generic :
        cover property(@(posedge clk) $rose(gnt) |->
                      ($past(req, s) == 1'b1));
    end
  endgenerate
```

一个 generate 语句可以生成一个 cover 语句的数组。目的是发现仲裁器的平均时延。内嵌函数\$past 用来定义一个有效的请求和许可间的时序。变量“s”用来循环，并为每个可能的时延值(分别是 2、3、4 和 5 个时钟周期)生成 4 个 cover 属性。Cover 语句将对不同的延迟值的情况分别记数。如下所示的是实例的模拟结果。

```
tb.u4.ul.generic[2].c_gnt_generic, 5793 attempts,
112 match, 0 vacuous match
  tb.u4.ul.generic[3].c_gnt_generic, 5793 attempts,
113 match, 0 vacuous match
  tb.u4.ul.generic[4].c_gnt_generic, 5793 attempts,
101 match, 0 vacuous match
  tb.u4.ul.generic[5].c_gnt_generic, 5793 attempts,
104 match, 0 vacuous match
```

从这些结果可以看出仲裁器平均分配时延。我们对前一个例子稍做修改，可以得到指定的主控设备的时延信息。用这种方法我们可以得知仲裁器给指定的主控设备授权上是否花费了更长的时间。

```
assign req_local[3:1] = ({req3, req2, req1});
assign gnt_local[3:1] = ({gnt3, gnt2, gnt1});

genvar j, k;
generate
  for (j=2; j<6; j++)
    begin: generic
      c_gnt_local[j] :
        cover property(@(posedge clk) $rose(gnt_local[j]) |->
                      ($past(req_local[j], k) == 1'b1));
    end
  endgenerate
```

```

begin: latency
  for (k=1; k<4; k++)
    begin: Master
      c_gnt_o :
        cover property(@(posedge clk) $fell(gnt_local[k])
          | -> ($past(req_local[k], j) == 1'b0));
    end
  end
  endgenerate

```

注意，代码中定义了一个名为“gnt_local”的“gnt”信号的向量和一个名为“req_local”的“req”信号的向量。这样一次可以遍历每个主设备。代码使用了两层循环，外层循环“latency”定义了时延的区间，内层循环“Master”定义了主控设备的标识。当发现一个有效的“gnt”信号，属性被激活，在过去的2~5个时钟周期能找到触发这个“gnt”的一个有效的“req”。这样一个覆盖语句的实例模拟结果如下。

```

tb.u4.u1.latency[2].Master[1].c_gnt_o, 5793
attempts, 41 match, 0 vacuous match
tb.u4.u1.latency[2].Master[2].c_gnt_o, 5793
attempts, 37 match, 0 vacuous match
tb.u4.u1.latency[2].Master[3].c_gnt_o, 5793
attempts, 34 match, 0 vacuous match
tb.u4.u1.latency[3].Master[1].c_gnt_o, 5793
attempts, 39 match, 0 vacuous match
tb.u4.u1.latency[3].Master[2].c_gnt_o, 5793
attempts, 34 match, 0 vacuous match
tb.u4.u1.latency[3].Master[3].c_gnt_o, 5793
attempts, 40 match, 0 vacuous match
tb.u4.u1.latency[4].Master[1].c_gnt_o, 5793
attempts, 27 match, 0 vacuous match
tb.u4.u1.latency[4].Master[2].c_gnt_o, 5793
attempts, 36 match, 0 vacuous match
tb.u4.u1.latency[4].Master[3].c_gnt_o, 5793
attempts, 38 match, 0 vacuous match
tb.u4.u1.latency[5].Master[1].c_gnt_o, 5793
attempts, 34 match, 0 vacuous match
tb.u4.u1.latency[5].Master[2].c_gnt_o, 5793
attempts, 29 match, 0 vacuous match

```

```
tb.u4.u1.latency[5].Master[3].c_gnt_o, 5793  
attempts, 41 match, 0 vacuous match
```

3.3 在 FSM 中应用 SVA 的小结

- 有限状态机是任何设计所必需的一部分，需要对它们进行彻底的验证。
- 每个被禁止的转换应该用 SVA 来检验。如果发生了一个被禁止的转换，应该标示为致命错误。
- 测试平台必须覆盖所有可能的合法转换。可以利用功能覆盖的信息建立一个反应的模拟环境。



SVA 用于数据集约型 (DATA iNTENSIVE)的设计

对于任何设计，有两个方面必须完全验证：

控制逻辑是否正确？这些信号控制设计中的数据的流动，并且相互之间有着复杂的时序关系。

输出的数据是否与预期的一致？只有确定了 RTL 的输出数据与黄金模型(Golden Model，通常用 C 构建)的输出一致性，才能确保 RTL 设计在经硬件算法优化后仍与黄金模型一致。

总的说来，基于断言的验证非常适合检验那些有着复杂时序关系或者说控制逻辑的信号。SVA 语言特有的描述性使它更适用于时序检验。虽然断言并没有对数据校验增加额外价值，但它仍可以用于编写有效自检环境。

4.1 简单乘法器的检验

SVA 一大优势是它使用 SystemVerilog 语言自身的大多数数值类型和操作符，这使得它可以灵活地用于编写简单算术校验。

例 4.1 简单乘法器

```
module au (
    input logic [7:0] a, b, c,
    input logic sel,
```

```
        output logic [15:0] d
    );

logic [15:0] e;
logic [15:0] sel_h, sel_l;

// Resource sharing architecture

always_comb
begin
    if(sel) e = b; else e = c;
    d = multiply(a, e);
end

// Functional sva checker

always@(a, b, c, sel)
begin
    sel_h = a*b;
    sel_l = a*c;

    if(sel)
        sel_high : assert (sel_h == d);
    if (!sel)
        sel_low : assert (sel_l == d);

end
endmodule
```

例 4.1 是一个简单乘法器。本例只有一个乘法器并且使用线“sel”来从输入中选择用于乘法的数据。我们可以写两个简单的检验器“sel_high”和“sel_low”来检验乘法器。当“sel”处在高电平，检验器“sel_high”处于激活状态，而当“sel”处在低电平，检验器“sel_low”处于激活状态。用户可以根据自己的环境选择任何一种乘法器。例如，shift/add 乘法器，布斯(Booth)乘法器或者其他乘法器。从验证角度来说，无论使用哪一种乘法器，我们必须保证输出是一致的。图 4-1 显示两个检验器的结果。请注意检验器的激活与否由“sel”状态决定(即时断言)。

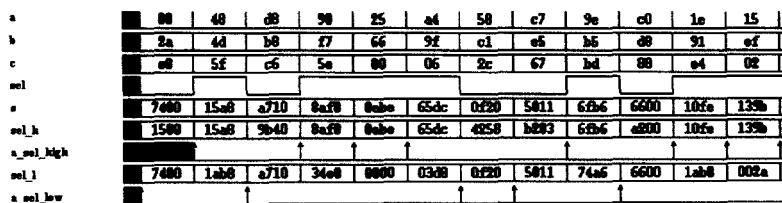


图 4-1 乘法器的检验器波形图

4.2 设计实例——算术单元

在本节，我们讨论一种算术单元(Walsh-Hadamard Transform 变换模块，简称 WHT)。WHT 是一种用于静态图像压缩的常用算法。图像编码前，WHT 将像素从时间域转换到频率域。通常这种算法很容易用 C 或者 Matlab 程序来验证。但当这种算法在被转换成硬件时，它经过深度的优化从而更有硬件效率。通常的做法是给黄金 C 模型和 RTL 模型相同的输入，然后通过对比 RTL 模型的输出与 C 模型的输出，从而对其验证。在本节中，我们将用 SVA 来动态地在模拟过程中产生黄金结果，并且与 RTL 的结果对比。

4.2.1 WHT 算术

WHT 算术是一个 8×8 的矩阵乘法。在图像压缩中，数据被逐块处理。每块是一个 8×8 的矩阵，也就是 64 个数据点。它的目的是对两个 8×8 矩阵进行一次矩阵乘法产生另一个 8×8 矩阵，由于矩阵乘法的可重复性，这种操作一次可得到一个模块。WHT 矩阵按以下定义，由于这种矩阵的换算因子由 -1 或 +1 组成，乘法可简化成加法或减法。

```
WHT [8][8] =  
{  
    {1, 1, 1, 1, 1, 1, 1, 1},  
    {1, 1, 1, 1, -1, -1, -1, -1},  
    {1, 1, -1, -1, -1, -1, 1, 1},  
    {1, 1, -1, -1, 1, 1, -1, -1},  
    {1, 1, 1, 1, -1, -1, 1, -1},  
    {1, 1, 1, -1, 1, -1, 1, 1},  
    {1, 1, -1, 1, 1, 1, 1, -1},  
    {1, -1, 1, 1, 1, -1, -1, 1}}
```

```

{1, -1, -1, 1, 1, -1, -1, 1},
{1, -1, -1, 1, -1, 1, 1, -1},
{1, -1, 1, -1, -1, 1, -1, 1},
{1, -1, 1, -1, 1, -1, 1, -1}

};


```

4.2.2 WHT 硬件的实现

在硬件中，我们可以通过减少所需的加法和减法的次数优化 WHT 算法。这是通过减少对同一组数进行重复的加法和减法操作来实现。每一个算术单元可优化成一个 1×8 与 8×8 矩阵乘法。换句话说就是每次处理一行数(8 个数据点)。用这种简化的算法单元，每得到一行结果需要通过 3 个阶段的加减运算。图 4-2 显示硬件 WHT 算法实现的示意图。

假设 D1, D2, D3, D4, D5, D6, D7 和 D8 组成一行数，这一行数将进行 WHT 矩阵变换。以下列出的是三阶段的优化算法：

阶段 1

$$\begin{aligned} Y1 &= D1 + D2, \quad Y2 = D3 + D4, \quad Y3 = D5 + D6, \quad Y4 = D7 + D8, \\ Y5 &= D1 + D4, \quad Y6 = D5 + D8, \quad Y7 = D2 + D3, \quad Y8 = D6 + D7, \\ Y9 &= D1 + D3, \quad Y10 = D6 + D8, \quad Y11 = D2 + D4, \quad Y12 = D5 + D7 \end{aligned}$$

阶段 2

$$\begin{aligned} Z1 &= Y1 + Y2, \quad Z2 = Y3 + Y4, \quad Z3 = Y1 + Y4, \quad Z4 = Y2 + Y3, \\ Z5 &= Y1 + Y3, \quad Z6 = Y2 + Y4, \quad Z7 = Y5 + Y6, \quad Z8 = Y7 + Y8, \\ Z9 &= Y5 + Y8, \quad Z10 = Y7 + Y6, \quad Z11 = Y9 + Y10, \\ Z12 &= Y11 + Y12, \quad Z13 = Y9 + Y12, \quad Z14 = Y11 + Y10 \end{aligned}$$

阶段 3

$$\begin{aligned} X1 &= Z1 + Z2, \quad X2 = Z1 - Z2, \quad X3 = Z3 - Z4, \quad X4 = Z5 - Z6, \\ X5 &= Z7 - Z8, \quad X6 = Z9 - Z10, \quad X7 = Z11 - Z12, \quad X8 = Z13 - Z14 \end{aligned}$$

X1, Z2, X3, X4, X5, X6, X7 和 X8 组成一行输出结果。

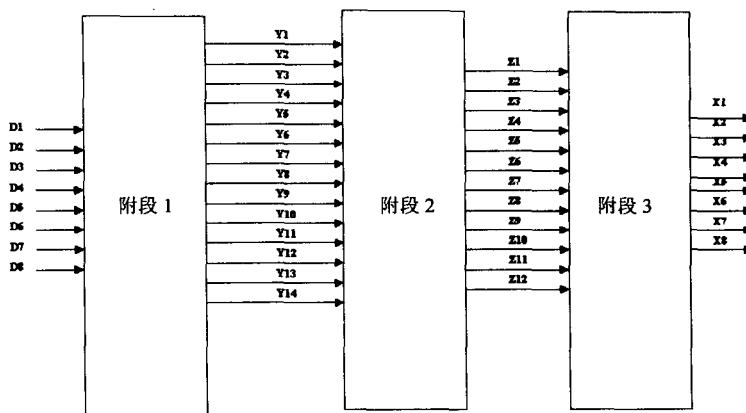


图 4-2 WHT 硬件模块图

4.2.3 WHT 模块的 SVA 检验器

校验器不需要知道模块的具体实现就可以对它进行功能验证，校验器可以产生黄金结果(GOLDEN RESULT)与设计的结果进行对比，SVA 校验器的黄金结果很容易通过矩阵乘法产生，并且与 WHT 模块的输出结果对比。图 4-3 显示一个用于 WHT 模块的简单检验器。

为了得到最稳定的数据，通常把这种组合模块的输出储存在寄存器中，而检验器使用寄存器的使能信号作为触发，检验器产生的结果与 WHT 算法模块产生暂存在寄存器中的输出对比。例 4.2 显示的是 SVA 的检验器。

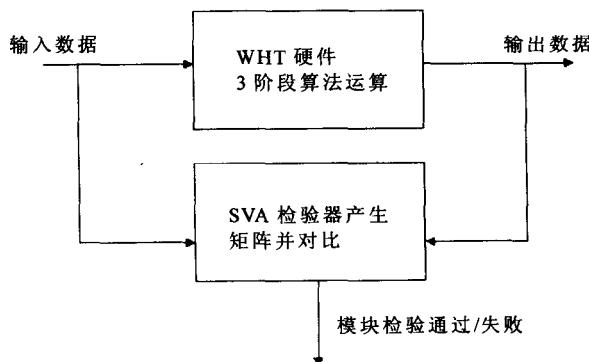


图 4-3 WHT 检验器的设置

例 4.2 WHT 模块的 SVA 检验器

```

module au_comp_chk (
    input logic clk, reset, enable1, enable2,
    input logic signed [15:0]
        d1, d2, d3, d4, d5, d6, d7, d8,
    input logic signed [15:0]
        o1, o2, o3, o4, o5, o6, o7, o8
);

logic signed [15:0] in_local[0:7];
logic signed [15:0] out_orig[0:7];
logic signed [15:0] out_local[0:7];

integer i, k;

integer wh_local[0:7][0:7] =
{
    {1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, -1, -1, -1, -1},
    {1, 1, -1, -1, -1, -1, 1, 1},
    {1, 1, -1, -1, 1, 1, -1, -1},
    {1, -1, -1, 1, 1, -1, -1, 1},
    {1, -1, -1, 1, -1, 1, 1, -1},
    {1, -1, 1, -1, -1, 1, -1, 1},
    {1, -1, 1, -1, 1, -1, 1, -1}
};

always@(o1, o2, o3, o4, o5, o6, o8)
begin
    out_orig[0] <= o1;
    out_orig[1] <= o2;
    out_orig[2] <= o3;
    out_orig[3] <= o4;
    out_orig[4] <= o5;
    out_orig[5] <= o6;
    out_orig[6] <= o7;
    out_orig[7] <= o8;
end

always@(d1, d2, d3, d4, d5, d6, d7, d8)
begin

```

```

for(i=0; i<8; i++)

begin

    out_local[i] <=
    (d1*wh_local[i][0]) + (d2*wh_local[i][1]) +
    (d3*wh_local[i][2]) + (d4*wh_local[i][3]) +
    (d5*wh_local[i][4]) + (d6*wh_local[i][5]) +
    (d7*wh_local[i][6]) + (d8*wh_local[i][7]) ;

end

end

genvar j;
generate

    for(j=0; j<8; j++)
    begin : loop
        a_au_comp_chk_o :
        assert property
            (@(posedge clk) (reset &&enable2) |->
                (out_local[j] == out_orig[j]));
    end

    endgenerate

endmodule

bind au_comp au_comp_chk a1
    (clk, reset, enable1, enable2, d1, d2, d3, d4, d5,
    d6, d7, d8, o1, o2, o3, o4, o5, o6, o7, o8);

```

reset
clk
out_orig[0]
out_local[0]
loop[0].a_au_comp_chk_o

图 4-4 WHT 检验器的波形图

检验器自己算出预期结果并且把结果存入数组“out_local”，而设计的原始输出存在数组“out_orig”中。“generate”这一语句会产生一组属性来验证 8 个数据点，通过参数“genvar”FOR 循

环产生 8 个不同的属性，每一个属性同时验证一个数据点。当激活信号处于高电平并且设计还未被重至时，这些属性会通过对比在“out_local”和“out_orig”中的相应数值来验证，如果不相等则断言失败。图 4-4 显示第一组数据的结果。

4.3 设计实例——JPEG 的数据通路设计

本节我们将讨论 JPEG 模型的验证实例，这个设计模块是 JPEG 解码器的一部份，它用来从存储器中读取数据，然后按照一定的算法对数据转换，转换后的数据再写回存储器用于打包和发送。

JPEG 模型由 3 个主模块组成：数据注入器，数据通路和控制模块。图 4-5 显示的是 JPEG 模型的顶层方块示意图，以下是每个模块的具体功能：

- 控制模块控制握手协议的进行，并且产生控制信号控制数据通路和数据注入器所进行的数据处理。
- 数据注入器每次从存储器读取一组数据，并且把数据传给数据通路去处理。
- 数据通路对这一组数据进行算法操作，然后存入存储器。

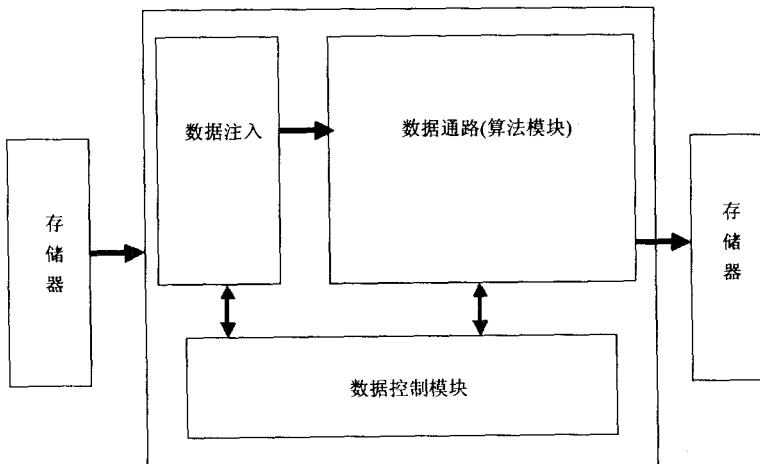


图 4-5 JPEG 模型的方块图

4.3.1 三模块的深入探讨

数据注入器模块

图 4-6 显示的是数据注入器模块的方块图，该模块由两个子模块组成，一个是串行输入并行输出模块(SIPO)，一个是并行输入并行输出模块(PIPO)。SIPO 一次读取 16 位的数据并且并行地输出 64 个 16 位数据。

当 SIPO 被激活(sipo_enable)，它开始把数据载入移位寄存器，当我们有 64 组数据样本后，SIPO 被禁用而 PIPO 将锁住数据，被锁住的数据将用于数据通路的进一步处理，控制模块产生激活信号给 SIPO 和 PIPO，图 4-7 是数据注入器模块功能的波形图。



图 4-6 注入器模块方块图

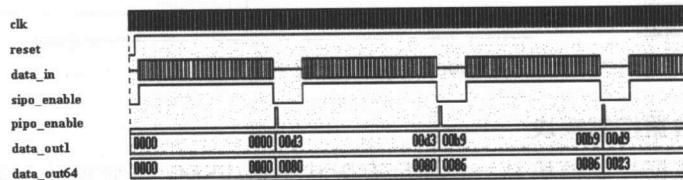


图 4-7 注入器模块波形图

数据通路模块

数据通路模块每次取 64 个 16 位数据进行一定的算法操作，这一过程持续多个时钟周期才能完成所有操作，为了完成这一任务使用了多层流水线。图 4-8 显示执行这一算法的流水线的方块图。

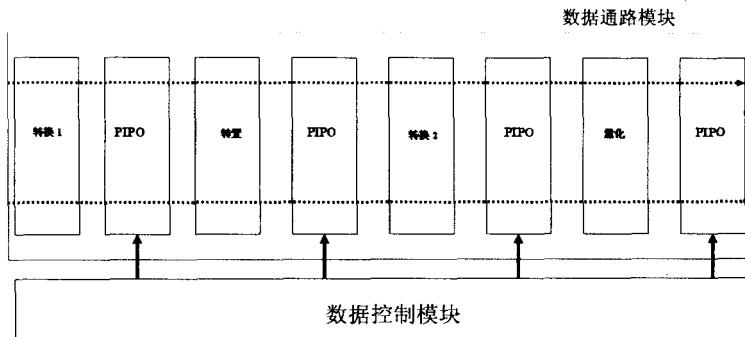


图 4-8 流水线方块图

数据通路模块是一个基于锁存器的流水线，数据需通过 4 个阶段的处理，它们是转换 1，转置，转换 2 和量化。在每一阶段的处理后，稳定的数值通过 PIPO 传到下阶段，PIPO 锁存器由控制模块产生的激活信号所控制。流水线的每一阶段需要 2 个时钟周期来完成。换句话说，控制模块产生 4 个依次间隔 2 个时钟周期的激活信号来锁存各阶段传输。图 4-9 显示控制信号与流水线的关系。

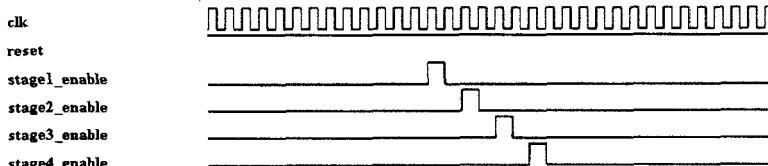


图 4-9 流水线控制的波形图

数据控制模块

数据控制模块是一个简单有限状态机(FSM)，它产生控制所需的信号，使数据在流水线顺利地移动。图 4-10 是控制模块的方块图，图 4-11 是控制信号产生的波形图。

控制模块产生控制信号控制数据注入模块和数据通路模块，一接到外面来的“get_data”的信号，控制模块的状态机就开始工作，它起动计数器产生“read”给储存器和“read_address”帮助每次读取 64 个数据。

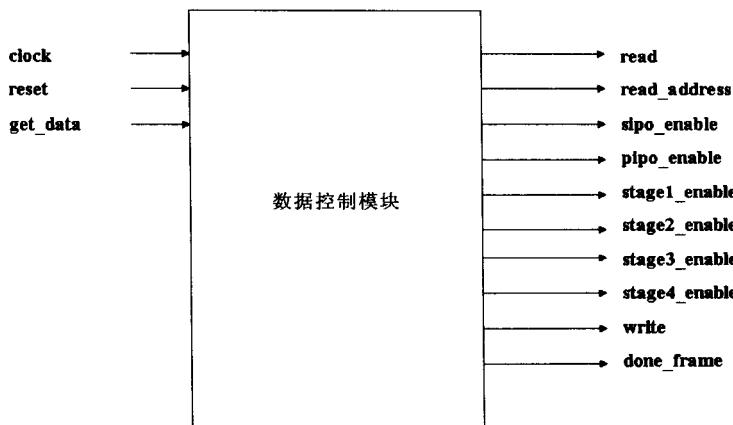


图 4-10 控制通路模块方块图

当 64 个数据读完，“read”信号被禁用，接着产生流水线的激活信号，在这些数据经流水线 4 个阶段的处理后，产生“write”信号把处理完的数据存入储存器，在“write”信号之后，又一新的数据从储存器读出。这一过程持续到储存器内所有的数据都读完为止。本节的 JPEG 实例的储存器可以储存 262144 字节(相当一幅 512×512 的图像)，这意味着为了处理完所有数据，控制信号要产生 4096(262144/64)次。在处理完所有的数据块后，控制模块立即断言“done_frame”并且取消断言“get_data”。

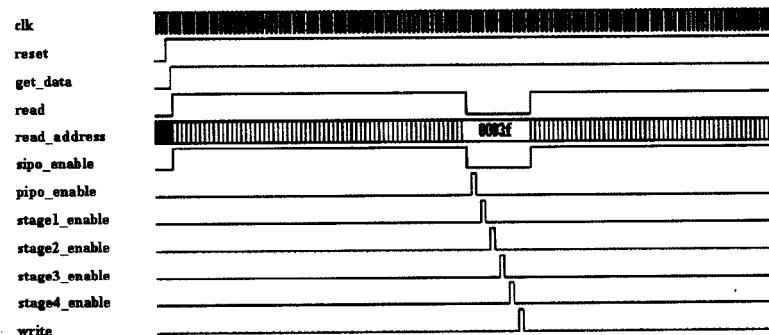


图 4-11 控制模块波形图

需要验证什么？

- 验证数据流动控制信号，它们是否按照时序正确地产生？

- 数据通路是否正确工作，是否产生正确的输出数据，并且存入储存器？

4.3.2 用于 JPEG 设计的 SVA 检验器

基于以上对设计的描述，我们可以用以下检验器来对设计全面地验证。

JPEG_chk1: “get_data” 与 “done_frame” 互斥。

当 “get_data” 信号被触发，该设计开始从储存器里读出数据；在请求和处理数据过程中，“done_frame” 处于低电平。当数据处理完毕，“done_frame” 被设为高电平，同时 “get_data” 被设为低电平，所以这两个信号不会同时被触发。

```
property p_mutex;
  @ (posedge clk) ((reset_) |>
    not (done_frame && get_data));
endproperty
a_mutex: assert property (p_mutex);
```

这种相互排斥的条件用 “not” 来表示。“not” 表明该表达式永远不会成立，检验器在时钟的每个上升沿开始验证，检验器 “a_mutex”的结果如图 4-12 所示：

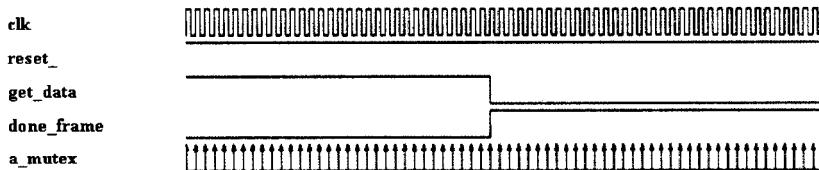


图 4-12 JPEG_chk1 的波形图

JPEG_chk2: “read” 信号连续 64 个时钟保持高电平，而在这一期间，“read_address” 每一个时钟周期增加 1。

本实例一次处理 64 个数据点，也就是说，一个读取区间需 64 个周期才能读完所需的数据。如果我们验证了以上的声明，那么我们就证明该设计每 64 个时钟读取一组数据。

```

sequence s_read;
  rd_addr == $past (rd_addr)+1) [*0:$] ##1
    $fell (rd);
endsequence

property p_read;
  @ (posedge clk)
  ($rose (rd) && reset_) |->
    s_read;
endproperty

a_read: assert property(p_read);

```

\$past 运算符用来检查 “rd_addr” 每周期加 1。本时钟周期 “rd_addr”的值应等于上周期值加 1。这一检查从读取信号(\$rose (rd))的上升沿开始，到下降沿结束(\$fell (rd)) “repeat until [*0:\$]” 用来检查地址的正确性(直到“rd”信号被设为低电平)。“a_read” 检验器的结果如图 4-13 所示。每当读信号上出现上升沿，我们显示一个有效的匹配。该属性本身将持续几个时钟周期，但我们在图上只标示第一次匹配。

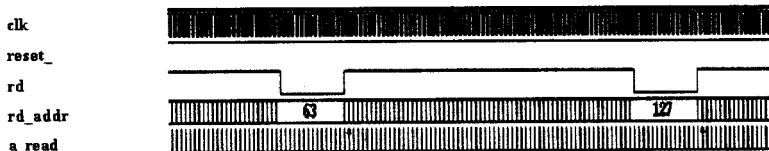


图 4-13 JPEG_chk2 波形图

JPEG_chk3: 在读取数据时，“sipo_en”保持 64 个时钟的高电平，然后被禁用。2 个时钟周期后“pipoe_en”被触发以便锁存住数据通路所要处理的数据。

数据注入模块依赖于“sipo_en”和“pipoe_en”，这两个信号帮助数据注入模块向数据通路模块提供合法数据。

本检验器检验数据注入模块的功能正确与否。

```

sequence s_datafeeder;
  sipo_en[*64] ##1 $fell (sipo_en) ##1
    latch_en ##1 !latch_en;

```

```

endsequence

property p_datafeeder;
  @(posedge clk) ($rose (sipo_en) && reset_) |->
    s_datafeeder;
endproperty

a_datafeeder: assert property(p_datafeeder);

```

该检验器用一个简单的 **repeat operator** (*) 来监视 “sipo_en” 是否保持 64 个时钟的高电平。一旦 “sipo_en” 降为底电平，“latch_en” 脉冲被触发，“a_datafeeder” 检验器的结果如图 4-14 所示：

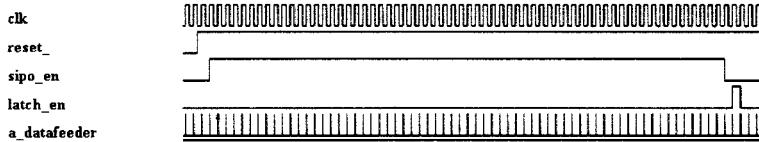


图 4-14 JPEG_chk3 的波形图

“s_datafeeder” 序列可以如下表示：

```

sequence s_datafeeder;
  ##64 $fell (sipo_en) ##2
  latch_en ##1 latch_en;
endsequence

```

在以上 “s_datafeeder”的表示中 “sipo_en”的下降沿是 64 个周期后再检查，这并不能保证 “sipo_en” 保持 64 个周期的高电平。

JPEG_chk4: 在数据通路模块中，每两个时钟激活一个阶段。

在数据通路模块中，每个阶段有两个时钟的时间为下一阶段提供稳定的数据。信号“dp1_enable”，“dp2_enable”，“dp3_enable”和“dp4_enable”帮助每阶段锁存住数据，它们按照每两个时钟的间隔来按顺序地触发，这才能保证数据正确地流动。

```

sequence s_control;
  dp1_en ##1 !dp1_en ##1 dp2_en ##1 !dp2_en ##1
  dp3_en ##1 !dp3_en ##1 dp4_en ##1 !dp4_en ##1
  wr ##1 !wr;
endsequence

```

```

property p_control;
  @(posedge clk) $fell (latch_en) |=> s_control;
endproperty
  a_control: assert property(p_control);

```

PIPO 的每个激活信号都是一个时钟周期的脉冲，它们按 2 个时钟周期的间隔而产生。序列“s_control”用来监视这些控制信号的上升和下降是否遵照顺序串联的方式。该序列当数据被载入数据通路模块(latch_en)时开始，“|=>”用来表示“latch_en”的下降沿是 s_control 被检验的控制条件。当数据通过数据通路模块，处理后的数据写回存储器后(wr)，该序列结束。

JPEG_chk5: 从“get_data”的上升沿到它的下降沿，序列“s_datafeeder”和“s_control”一共重复 4096 次直达该设计被重置。

这保证所有的数据都按正确的顺序处理，同时这也用作功能覆盖来保证储存器中的所有数据都被处理。

```

property p_control_all;
  @(posedge clk) ($rose(sipo_en) && reset_) |=>
    s_datafeeder ##1 s_control;
endproperty

property p_block;
  @(posedge clk)
  $fell (get_data) && $rose(done_frame) |=>
    (block == 4095);
endproperty

a_control_all: assert property(p_control_all);
c_control_all:
  cover property(p_control_all) block++;
a_block: assert property(p_block);

```

为了保证处理完所有的数据序列重复 4096 次，检验器 JPEG_chk3 和 JPEG_chk4 必须是串联的。属性 p_control_all 在数据从储存器读入(sipo_en)时开始，在处理完的数据被写入输出储存器时(wr)结束。属性 p_control_all 可以用“cover”的声明来描述，

这可以提供有关该属性有多少次真正成功和多少次是空成功的信息。每一个真正成功，变量“block”加 1，真正成功总次数应是 4095 次。属性 a_block 用这个变量来检验是否所有的数据块都被检查过。当“done_frame”信号处于上升沿，并且在同一时钟周期“get_data”处于下降沿时，这表明最后一组数据在被处理，变量“block”应是 4095。图 4-15 显示“a_control”的结果，图 4-16 是“a_block”的结果：

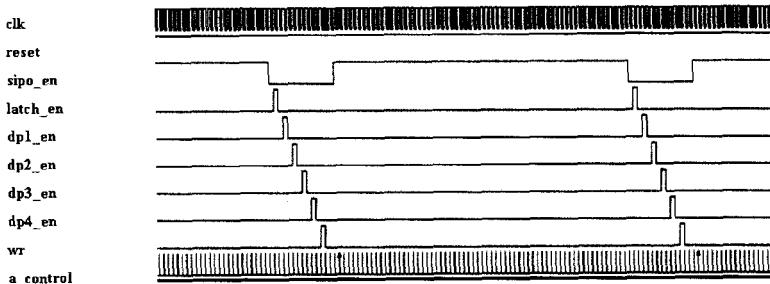


图 4-15 JPEG_chk5 的波形图

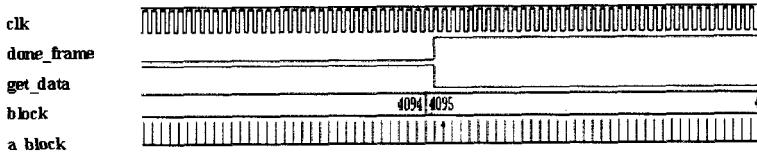


图 4-16 检验 a_block 的波形图

4.3.3 JPEG 模型的数据检验

在 4.3.2 小节中，SVA 检验器用来验证所有的控制信号是否正确地产生。这样才能保证数据在流水线顺利地流动。如果没有检查数据的完整性，而流水线的每一阶段都对数据进行一定变形，这需要被检验。通常的做法是把输出的数据存入一个文件，然后再与黄金模型的输出对比。虽然这种方法是可行的，但它有着以下缺点：

- (1) 需要在跑完整个模拟后才能与预期结果对比，如果结果出错，则浪费了很多模拟时间。

(2) 这种方法不利于查错，因为它无法显示流水线的哪一阶段开始出错。一个简单的解决方法是分阶段把结果存入文件，并且分阶段对比。虽然这有利于查错，但仍无法解决浪费模拟时间的问题。

更有效的检验数据方法是动态地进行对比，它有许多方法来实现，读者可以根据自己的模拟环境来决定选用何种方法。数据的检验是一个重复的过程，在检验了几个数据包后，可以关掉动态对比。换句话说，我们可以设一个检验的目标，一旦检验目的达到后，检验器可以关掉，以提高模拟速度。

JPEG 模型动态数据检验的可能步骤：

- 模拟黄金 C 模型产生如图 4-17 所示流水线各阶段的结果。虽然让黄金 C 模型产生的结果与 RTL 流水线的结果一致不容易，但这还是可能的。
- 在模拟实际的 RTL 前，产生 JPEG 黄金 C 模型的输出数据文件：Wh1.dat (transform1 的输出), Xpose.dat (transpose 的输出), Wh2.dat (transform1 的输出), Quantize.dat (量化输出)。
- 对 RTL 使用相同的输入文件来进行数据验证。
- 如图 4-18，创建一个检验器来载入黄金结果(Golden results)，然后与 RTL 模拟动态地对比。在模拟进行的同时，当遇到相应的触发点，检验器会对比模拟结果与黄金结果，并且报告出错信息，例 4.3 是一个数据检验器的实例。

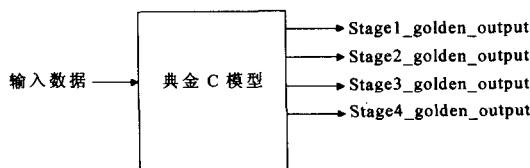


图 4-17 C 模型的黄金输出

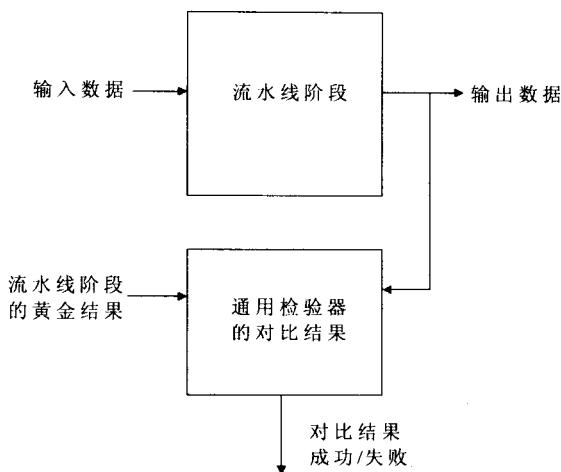


图 4-18 数据通路动态检验

例 4.3 数据通路验证的 SVA 检验器

```

module dp_chk(
    input logic reset, clk, enable,
    input logic [15:0]
        d1, d2, d3, ..., d61, d62, d63, d64 );
parameter data_file = "";
parameter identity = "";
integer i=0, j=0;
integer blk=0;
integer fd, fdl;

logic [31:0] pix_in_temp;
logic [15:0] local_array[0:63];
logic [15:0] pix_in [0:262143];

// use $fopen construct to open the golden
// results file

initial
begin
    fd = $fopen(data_file, "r");
end

// copy design data to a local array

```

```
always@(*)
begin
local_array[0] <= d1;
local_array[1] <= d2;
local_array[2] <= d3;
..
..
local_array[62] <= d63;
local_array[63] <= d64;
end

// load actual results

always@(negedge enable)
begin
if(reset)
$display
  ("\nDATA CHECKING: Block number %0d\n", blk);
for(j=0; j<64; j++)
begin
fdl = $fscanf(fd, " %x", pix_in[j]);
end
blk++;
end

// compare results

genvar k;
generate
for(k=0; k<64; k++)
begin: dchk
a_dp_chk: assert property(
  @ (posedge clk) (reset && $fell(enable)) .|=>
  (pix_in[k] == local_array[k])) else $fatal;
end
endgenerate

endmodule

// check that data is put into blocks of 64 correctly

bind data_feeder dp_chk
#(.data_file("input_image.dat"),
```

```
.identity("INPUT")) dpchk1  
(reset_, clk, latch_en, q0, q1, ....,  
q61, q62, q63);  
  
// check that the output of first wh transform is  
// correct  
  
bind datapath dp_chk  
#(.data_file("wh1.dat"), .identity("WH1"))  
dpchk2  
reset, clk, dp_enable1,  
dw11, dw12, ..., dw161, dw162, dw163, dw164);  
  
// check that the transposed data is correct  
  
bind datapath dp_chk  
#(.data_file("xposed.dat"),  
.identity("TRANSPOSE")) dpchk3  
(reset, clk, dp_enable2, dwlt1, dwlt2, ...,  
dwlt61, dwlt62, dwlt63, dwlt64);  
  
// check that the output of the second wh  
// transform is correct  
  
bind datapath dp_chk  
#(.data_file("wh2.dat"), .identity("WH2"))  
dpchk4  
(reset, clk, dp_enable3, dwltw11, dwltw12, ...,  
dwltw163, dwltw164);  
  
// check that the output of quantization is  
// correct  
  
bind datapath dp_chk  
#(.data_file("quantized.dat"),  
.identity("QUANTIZATION")) dpchk5  
(reset, clk, dp_enable4,  
do1, do2, ....do62, do63, do64);
```

例 4.3 显示的是一个简单 SVA 数据通路检验器和它如何与流水线各阶段一一对应。

- 检验器定义了两个参数来保证其惟一性。参数“`data_file`”定义的是检验器所用的黄金数据文件，参数“`identity`”定义的是它与数据通路的哪一部份相对应。
- 黄金数据存在文件中，该文件用`$fopen`来打开。
- 当被激活信号触发时，原设计的输出结果存入本地检验器`local_array`中。注意：数据通路每次处理 64 个数据点，所以在触发一次时只从黄金数据文件读入 64 个数据点。每次被触发时，我们将一个变量递增用来记录是哪一个图像数据块当前正在被处理。
- 用“`generate`”语句来生成 64 个检验器，每个数据点一个。“`for`”循环语句用来循环检查所有的被触发的 64 个数据点。
- 断言的动作模块使用一个`$fatal` 语句结构。当遇到违例时，它使模拟退出。这防止在查出错误后继续浪费时间在模拟上。
- 通过使用“`bind`”语句结构，检验器可与数据通路的特定点相连接。通过对每一点参数的定义，每个检验器成了惟一的实例。

以下显示一个模拟实例的日志：

```
DATA CHECKING: Block number 1
"adv_datapath.sva", 118:
tb.jpeg_int.datapath_inst.dpchk5.dchk[0].a_dp_chk:
started at 795s failed at 805s
    Offending '(pix_in[0] == local_array[0])'
"adv_datapath.sva", 118:
Fatal: "adv_datapath.sva", 118:
tb.jpeg_int.datapath_inst.dpchk5.dchk[0].a_dp_chk:
at time 805
```

注意，这个验证失败来自与通路中的“QUANTIZATION”模块(实例 `dpchk5`)相连接的检验器，这一失败清楚地指出在何时失败和失败时的数据块，例如，在以上的日志中，数据块 1 的 0 数据点失败。

虽然这是进行动态数据验证的可行方法，但它并不适合所有的设计，每个设计都有着不同的规格和要求。而我们可以将这一验证方法作为模板，根据不同的设计修改成相适应的方法。

4.4 数据集约型设计的小结

- SVA 能够进行算术操作，并支持大多数 SystemVerilog 的数值类型。
- 通过使用 Verilog tasks 和 functions，数据验证可以自动完成，并且可以得到该设计的功能覆盖率。
- 动态的数据通路 SVA 检验器智能地使用时钟周期，无须等到模拟结束就可以发现设计的问题。同时，检验器可以指出错误的确切所在处，从而使查错变得更简单。



SVA 储 存 器

——储存器控制器协议的验证

计算机和家用电子设备都使用大量的存储器来储存多媒体数据。在 ASIC 中，几乎所有芯片都使内嵌式存储器(DRAM、SRAM、ROM 等)。由于存储器的访问速度变得越来越快，这使得控制器能否适应不同厂商的存储器和不同的时序要求变得非常关键。对存储控制器接口进行验证的主要瓶颈是控制信号间的时序检查。这些问题可以通过使用断言有效地解决。一个用于某一特定存储设备的检验器，只要修改时间参数，就可以被多个厂商重复使用。本章将讨论如何开发可用于不同型号存储设备的可重用检验器。

5.1 存储控制系统实例

本实例系统的 CPU 与存储控制器相连接，CPU 可以从多个与存储控制器连接的存储器中读取和写入数据，存储控制器可与不同类型的存储器(如 SDRAM、DDR-SDRAM、SRAM、Flash、ROM 等等)相连接。图 5-1 显示本系统的方块图。

5.1.1 CPU-AHB 接口操作

本实例系统的 CPU 是一款通用处理器，它使用 AHB 总线与存储控制器相连。CPU 发出读和写的命令，同时 CPU 还选择从哪一块存储器中读和写，它支持 SDRAM 和静态存储器的分离和共

享存储地址/数据总线，它还支持 32、64 和 128 位宽的 AHB 数据和 32 位宽的 AHB 地址总线。

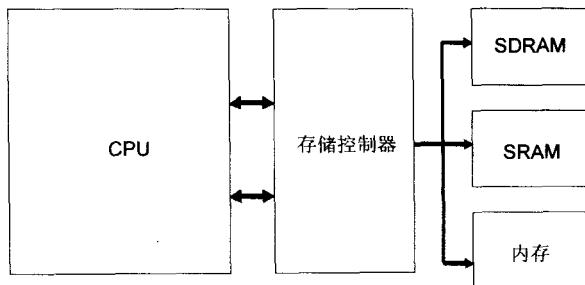


图 5-1 系统方块图

图 5-2 显示的是 CPU-AHB 总线接口与存储控制器之间的信号接口。以下列举的是各个管脚的说明：

- hclk – 总线参考时钟 (由 AHB 主控器 CPU 产生)
- haddr – 地址总线(由 AHB 主控器 CPU 产生的读/写地址通道)
- hwdata – 写入数据总线(由 AHB 主控器 CPU 产生的写入数据通道)
- hrdata – 读取数据总线(由 AHB 主控器 CPU 从存储器读出的数据通道)
- hready – 传输完成响应(由 AHB 主控器 CPU 发出的传输完成信号)
- hready_resp – 传输完成响应确认(由 AHB 随动器存储控制器发出的对 hready 的确认信号)
- hsize – 突发长度定义(由 AHB 主控器 CPU 发出设置从 CPU 到存储器传输数据的大小)
 - 00 – 每次传输 128 比特
 - 01 – 每次传输 64 比特
 - 10 – 每次传输 32 比特
- hburst – 突发模式定义(由 AHB 主控器 CPU 发出定义每次访问多少个地址)

- 000 – single –单字节传输模式
- 001 – INCR –未规定长度的增量模式如 0x40, 0x44 ...
- 010 – WRAP4 –4 字节打包模式如 0x48, 0x4c, 0x40 0x44
- 011 – INCR4 – 4 字节增量模式
- 100 – WRAP8 –8 字节打包模式
- 101 – INCR8 – 8 字节增量模式
- 110 – WRAP16 –8 字节打包模式
- 111 – INCR16 –16 字节增量模式
- sel_mem, sel_reg –对外部存储器/寄存器进行选择。

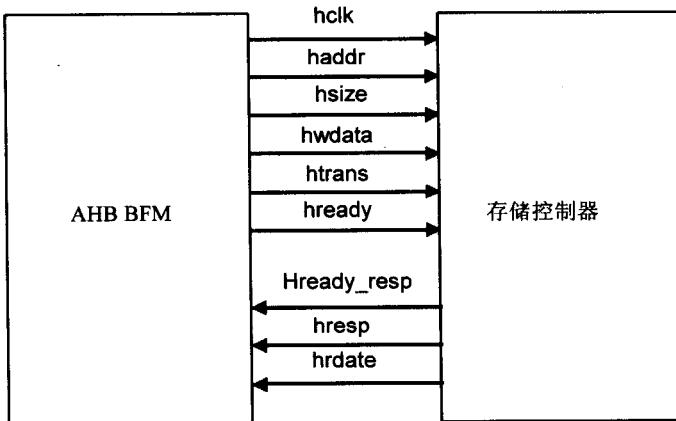


图 5-2 CPU 方块图

图 5-3 显示的是 CPU 对存储控制器发出写操作时的波形图。在标记 1 处 CPU 对存储控制器进行写操作的初始化，突发长度定义(hsize)设为 10，因而传输位宽为 32 比特；模式定义(hburst)设为 010，所以是 WRAP4 型；这种操作将访问地址 0×0、0×4、0×8 和 0×c。那么，这种写操作是 32 位宽，WRAP4 型，并在标记 1 处开始，下图显示的是地址从 80000000 增加到 8000000c 和“hwdata”上的数据被写入 SDRAM 时的波形图。

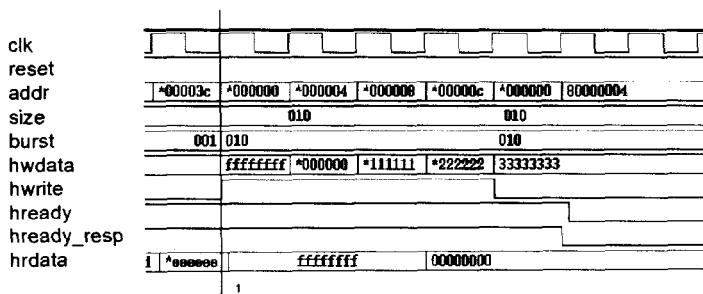


图 5-3 CPU-AHB 写操作

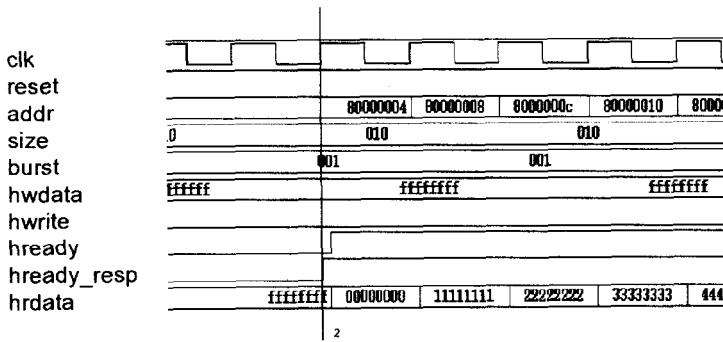


图 5-4 CPU-AHB 读操作

图 5-4 显示的是 CPU 对存储控制器进行读操作的波形图。这一读操作是 32 位宽，是 WRAP 型操作，在标记 2 处 CPU 对存储控制器开始进行读操作，当“ready”和“ready_response”被断言时操作开始，该图显示数据被写入后又被读出。

5.1.2 存储控制器的操作

本实例的存储控制器与 SDRAM、SRAM 和同步闪存设备接口，如图 5-5 所示。

与 SDRAM 接口的存储控制器是通用的。这种接口完全同步，所有的信号都是在时钟的上升沿被触发。对 SDRAM 的读写操作是突发型的，它的起始地址由 AHB 总线来控制并连续读写一定的长度，存储控制器和 SDRAM 是直接连接，之间没有任何胶联，它支持 16 位 SDRAM 地址，行列地址宽度都是可编程的，所有

SDRAM 的时序参数也都是可编程的, 它还支持自动刷新和可编程的刷新间隔。

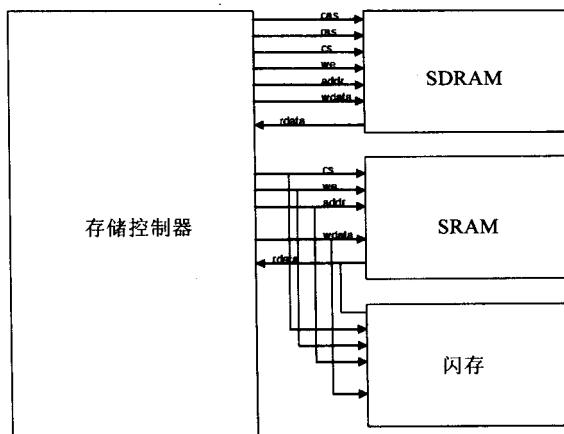


图 5-5 存储控制器方块图

以下列举存储控制器和 SDRAM 的管脚连接和简要说明：

- clk – SDRAM 时钟输入
- ras – 行地址选择
- cas – 列地址选择
- we – 读/写选择(如被断言是写信号, 被取消时是读信号)
- sel – SDRAM 芯片选择
- data – 读/写的双向数据总线
- addr – 读/写地址总线
- bank_sel – SDRAM 模组选择

图 5-6 显示的由存储控制器发出写命令时的波形图, 在下图中, 先执行一个长度为 4 的写操作, 然后再读回。激活命令(ras 和芯片选择信号被断言)先发出后, 然后发出对地址 0x0000 的写命令(cas、we 和 sel 被断言)。图 5-6 中, 在标记 1 处数据由一个长度为 4 的突发写命令写入。图 5-7 显示一个由存储控制器发出的读命令, 数据在 2 个周期的延迟后从存储器读出。

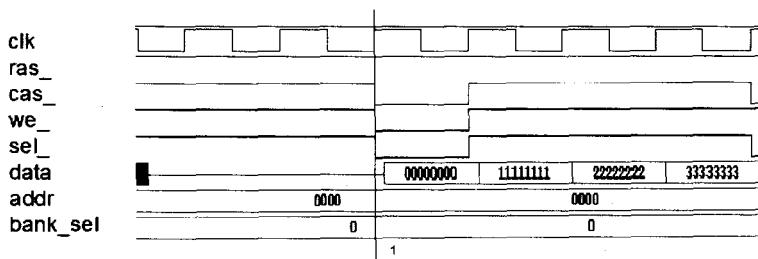


图 5-6 SDRAM 写操作

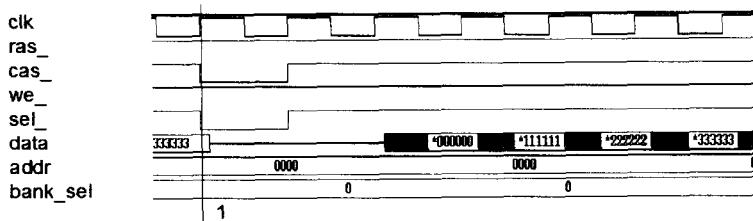


图 5-7 SDRAM 读操作

SRAM/FLASH 接口

SRAM/FLASH 都是静态存储器，它们的接口很类似。存储控制器支持异步 SRAM(asynchronous SRAM)，页面状态闪存(page-mode flash)和 ROM。它们的地址可以设为 32 位宽。存储控制器有“ready”握手信号以支持非 SRAM 型设备，存储数据可设为 8, 16, 32, 64 和 128 比特宽。静态存储器数据的最小值可以是 8 位，而不是标准的 16 位。本例中的闪存有写保护，所以可以保护启动模块中的重要信息。

以下列举的是 SRAM 与存储控制器的管脚连接和简要说明：

- **addr** – 存储控制器到静态存储器的地址管脚
- **data** – 从静态存储器到存储控制器的数据
- **sel_** – 芯片的选择管脚，用于选择相应的静态存储器
- **we_** – 被断言时是写操作信号
- **oe_** – 在读取操作中，被断言时，输出被激活
- **bs_** – 位控制管脚，用于设定不同的位宽。

闪存的接口和 SRAM 类似，比 SRAM 多以下两个信号：

- wp_ -写保护管脚
- rp_ -电源切点时，重设管脚

图 5-8 显示的是存储控制器和 SRAM 接口的波形图。当“we_”和“sel_”被断言(1 处)完成了对 SRAM 的一次写操作。当“sel_”和“oe_”被断言(2 处)而“we_”被取消时，完成了对 SRAM 的一次读操作。图 5-9 显示的是存储控制器和闪存接口的波形图，该图显示来自闪存的一次读操作。当“sel_”和“oe_”被断言时，执行了一次读操作，读取的地址由地址总线(addr)设定。

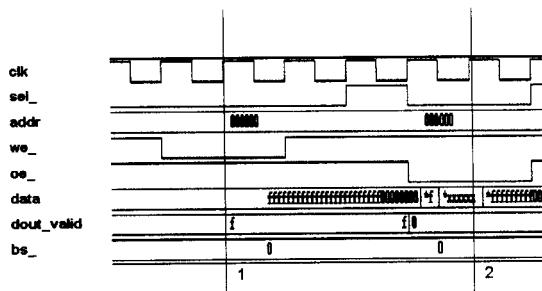


图 5-8 SRAM 接口信号

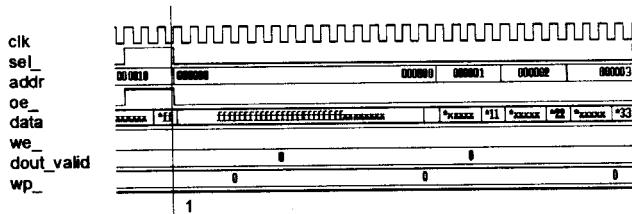


图 5-9 闪存接口信号

5.2 SDRAM 的验证

本节讨论的是如何验证 SDRAM 的控制信号。SDRAM 设备有很多时序参数，用断言可以高效率地验证这些时序是否被违反，本实例的 SDRAM 使用以下设置：

512Mb SDRAM - 8M X 16bit X 4 bank

本实例所要验证的 512 兆比特 SDRAM 有四个模块组(bank)和同步接口。所有的信号是在时钟的正上升沿被触发。每个模块组由 8192 行乘 1024 列乘 16 比特组成。读和写操作都是突发型的。每次访问从选定的地址开始，并按预先设定的值连续读/写一定长度。读写访问总是与紧跟在读写命令之后的激活命令一起开始。激活命令的地址位用来指定行地址和模块组。A0-A11 所指的是地址，BA[1:0]所指的是要访问的模块组，读/写的地址位指明访问的起始列地址(由 A0-A7 所指明)

SDRAM 接口信号 sel_，ras_，cas_ 和 we_ 的不同组合构成不同命令。所有的命令列举在表 5-1 中可以查找。“命令约束”条件用来防止 SDRAM 执行新的命令。已在执行过程中命令不受影响(sel_ = 1, cas_， ras_， we_ = x)。

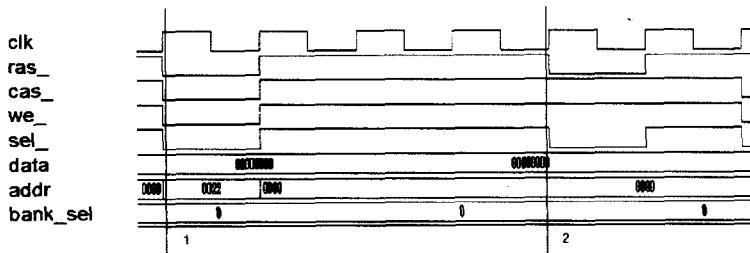


图 5-10 导入模式寄存器/激活命令

- **无操作:** 该命令用来防止在空闲/等待状态执行非所需的命令(sel_ = 0, cas_， ras_， we_ = 1)。
- **导入模式寄存器(Load Mode Register)命令:** 该寄存器由地址总线 (A0-A11)载入，只有当所有模块组都空闲时导入模式寄存器才被设置(sel_， cas_， ras_， we_ = 0)。图 5-10 显示在标记 1 处导入模式寄存器操作和在标记 2 处的“激活”操作。
- **激活命令:** 该命令用于激活/打开一行存储器来读写，地址总线的值是行的值和总线所指的模块组 bank_address 的值(sel_， ras_ = 0; cas_， we_ = 1)。

表 5-1 SDRAM 命令

命 令	Ras	Cas	We	Sel
无操作	H	H	H	L
激活命令	L	H	H	L
读命令	H	L	H	L
写命令	H	L	L	L
突发停止命令	H	H	L	L
载入模寄存器命令	L	L	L	L
预充(Precharge)命令	L	H	L	L
自动刷新命令 Auto-Refresh	L	L	H	L

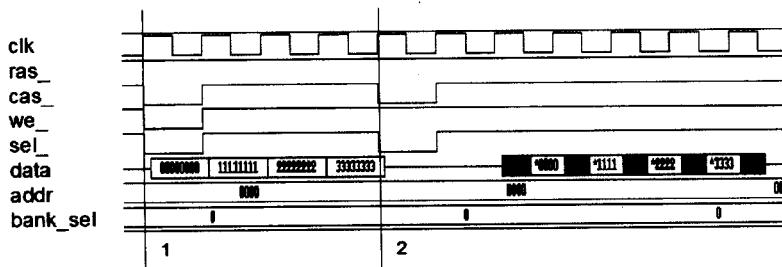


图 5-11 SDRAM 读/写

- **读命令(Read):** 该命令用来对已激活行(active row)进行突发读操作。由地址总线提供的“addr”作为列起始地址(sel_，cas_ = 0, ras_ we_ = 1)。.
- **写命令(Write):** 该命令用来对开放的行(open row)进行突发写操作，由地址总线提供的“addr”作为列起始地址(sel_，cas_，we_ = 0, ras_=1)。图 5-11 显示简单 SDRAM 读/写操作，在标记 1 处执行突发写操作，它的突发大小值是 4，而标记 2 处是突发读操作。
- **预充(Precharge)命令:** 预充用来使行进入禁用状态(sel_，ras_，we_ = 0, cas_=1)。如果 addr[10] 设为 1，那么该模块组所有行都被禁用。

- **自动刷新命令(Auto-refresh)命令:** 这一命令是 SDRAM 的正常操作，每当需要刷新 DRAM 时，这一命令必须发出。对所有激活的模块组必须先预充才能进行自动刷新。
- **突发终止命令(Burst Terminate):** 突发终止命令用于终止突发读/写命令。

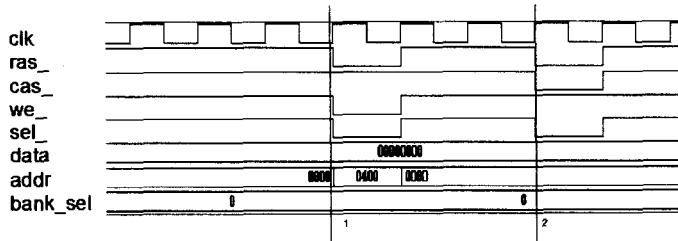


图 5-12 预充/自动刷新

图 5-12 显示预充命令在标记 1 处，自动刷新在标记 2 处。一个完整 SDRAM 读/写操作由图 5-13 中的步骤所构成，以下是简要说明：

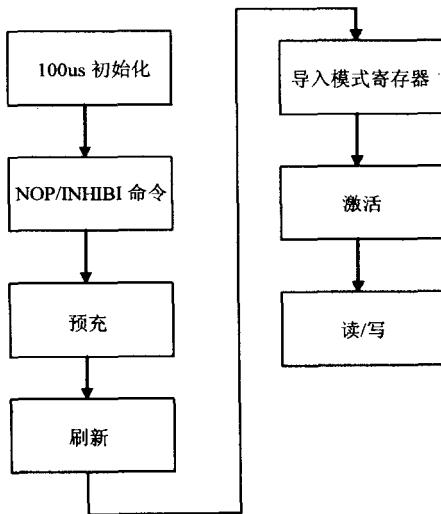


图 5-13 SDRAM 操作流程图

- (1) **初始化:** 当电源打开，SDRAM 需要约 100us 的时间来初始化，之后才能执行其他命令。
- (2) **初始化后，** 需要执行一个 NOP/COMMAND 和 INHIBIT

命令。

- (3) 发出预充(Precharge)命令，禁用所有的行。
- (4) 预充后是刷新命令。
- (5) 导入模式寄存器(设置“cas_”延迟值、突发大小以及其他设置)。
- (6) 发出执行激活命令(激活行)。
- (7) 发出读/写命令。

SDRAM 断言

所有 SDRAM 命令，例如：读，写，突发停止，激活，预充，导入模式寄存器等，都是由四个信号变化而来 ras_，cas_，sel_ 和 we_。而所有这些信号都可以用`define 来定义，这些定义可以在任何需要的 SVA 检验器中重用。

```
`define s_preamble
    (!ras_n && !sel_n[0] && !we_n && cas_n)

`define s_read
    (ras_n && !sel_n[0] && we_n && !cas_n && (burst
     == 3'b000))

`define s_burst_read
    (ras_n && !sel_n[0] && we_n && !cas_n && (burst !=
     3'b000))

`define s_write
    (ras_n && !sel_n[0] && !we_n && !cas_n)

`define s_autorefresh
    (!ras_n && !cas_n && !sel_n[0] && we_n)

`define s_loadmoderegister
    (!ras_n && !cas_n && !sel_n[0] && !we_n)

`define s_active
    (!ras_n && !sel_n[0] && cas_n && we_n)

`define s_write
    (!cas_n && !we_n && !sel_n[0] && ras_n && (burst
```

```

    == 3'b000))
define s_burst_write
  (!cas_n && !we_n && !sel_n[0] && ras_n &&
  (burst != 3'b000))

```

以下是从 SDRAM 的功能里抽取出的一些 SVA 检验器, 表 5-2 列举的是 SDRAM 所必须考虑的时序参数, 这些时序参数有的以时钟周期为单位, 其他的是以纳秒为单位。对于那些以纳秒为单位的值, 它们的时钟周期数由时钟周期长度(tCK)决定。本实例中 tCK 周期长度是 10 纳秒。SDRAM 的时序参数可作以下转换, 例如:

tRCD=18ns

tRCD/tCK =1.8 个时钟周期

所以激活命令与读/写命令之间的时间窗口至少为 2 个时钟周期。

表 5-2 SDRAM 的时序参数

参 数	符 号	Min	Max
导入模式寄存器到激活	tMRD	4 cycles	4 cycles
激活到激活命令期间	tRC	6 cycles (60ns)	-
激活到读/写	tRCD	2 cycles (18ns)	-
读延迟期	tCAS	2 cycles	-
自动刷新期	tRFC	6 cycles (60ns)	-
预充命令期	tRP	2 cycles (18ns)	-
激活模块组 a 到 b	tRRD	2 cycles (12ns)	-
激活到预充命令	tRAS	5 cycles (42ns) (120000ns)	12000 cycles (120000ns)

SDRAM_chk1: 导入模式寄存器到激活命令(tMRD)。

导入模式寄存器用来向 SDRAM 的模式寄存器导入信息, 这

些信息用来设置 SDRAM。在 SDRAM 设置完毕后 4 个时钟周期“tMRD”后应收到激活命令。图 5-14 显示在标记 1 处对导入模式寄存器命令取样，4 个时钟周期后(标记 2 处)再对激活命令取样，图中显示的 tMRD 验证成功。

```

property p_tMRD;
  @ (posedge clk)
    `s_loadmoderegister |->
      ##[tMRD] `s_active;
endproperty

a_tMRD: assert property(p_tMRD);
c_tMRD: cover property(p_tMRD);

```

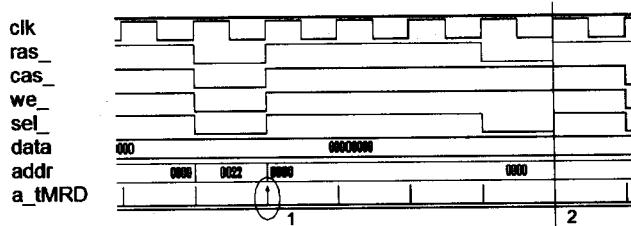


图 5-14 导入模式寄存器到激活命令(tMRD)

SDRAM_chk2: 检查模式寄存器导入的值(022)。

该断言用来检查写入模式寄存器的值，这个值很重要，它决定突发的大小和“cas_”延迟值。当导入模式寄存器命令发出时，地址总线的值被写入导入模式寄存器。Bits [0:2]设置突发的大小，本例的突发大小设为 4 (100)。“cas_”延迟值设为 2，所以模式寄存器写入值为 0x0022。

图 5-14 显示在标记 1 处由存储控制器发出导入模式寄存器命令，此时地址总线的值为 0x0022。因而，我们验证到 a_loadmoderegister 成功。

```

property p_loadmoderegister;
  @ (posedge clk)
    (`s_loadmoderegister) |->
      (addr == 16'h0022);
endproperty

```

```
a_loadmoderegister:  
    assert property(p_loadmoderegister);  
c_loadmoderegister:  
    cover property(p_loadmoderegister);
```

SDRAM_chk3: tCAS, 在读命令发出后, 再延迟 tCAS 后, 所读数据准备就绪。

在本 SDRAM 实例中, 无论何时发出读命令, 数据要延迟 “cas_” 才准备好(列地址选择延迟), 这是由存储器制造商在模式寄存器中预先设置好的。图 5-15 显示了在标记 1 处对读命令取样, 在 tCAS 个周期后, 标记 2 显示该值是合法的。为了验证这一特性, 我们使用蕴含结构和 \$isunknown 结构。

```
property p_read;  
    @ (posedge clk)  
        (`s_read || `s_burst_read) |->  
            ##tCAS ($isunknown(data)==0);  
endproperty  
  
a_read: assert property(p_read);  
c_read: cover property(p_read);
```

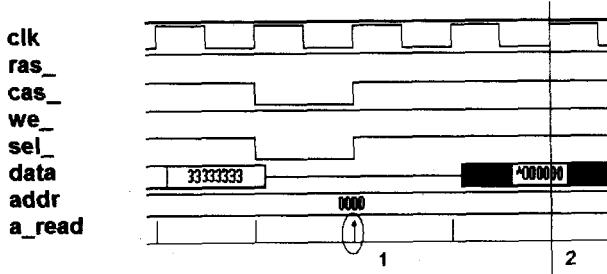


图 5-15 SDRAM 读操作的 tCAS 延迟

SDRAM_chk4: tRCD, 在激活命令后, 必须等 tRCD 个时钟周期后才能读/写。

当存储控制器发出激活命令, 在 “tRCD” 周期内不能发出读/写命令; 本实例还要求在激活命令后, 读/写命令必须在 10 个时钟周期内发出; 所以我们必须测试以下两个条件:

- (1) 当存储控制器发出激活命令, 在 “tRCD” 周期内不能发出读/写命令(这是一个禁止属性)。

- (2) 一旦储控制器发出激活命令后，读/写命令必须在 10 个时钟周期内发出。

```

property p_tRCD_not;
  @(posedge clk)
    `s_active |-> not ##[0: (tRCD - 1)]
      (`s_read || `s_write || `s_burst_read
       || `s_burst_write);
endproperty

property p_tRCD;
  @(posedge clk)
    `s_active |->
      ##[tRCD:10] (`s_read || `s_write
      || `s_burst_read || `s_burst_write);
endproperty

a_tRCD_not: assert property (p_tRCD);
a_tRCD: assert property (p_tRCD);

c_tRCD_not: cover property (p_tRCD);
c_tRCD: cover property (p_tRCD);

```

图 5-16 在标记 1 处有一个激活命令，在标记 2 处，在激活命令后的 2 时钟周期处取样到写命令，所以我们验证到 a_tRCD 是成功。

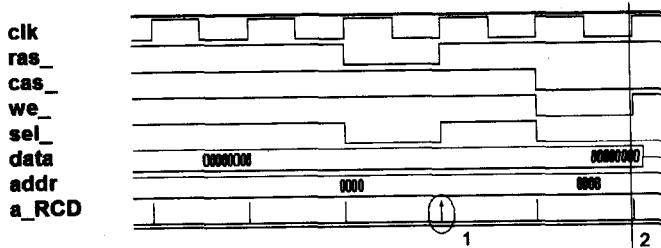


图 5-16 激活到读写命令, tRCD

SDRAM_chk5: tRC, 在一个激活命令后的 tRC 周期内不能再发出另一个激活命令。

在存储控制器发出一个激活命令后, tRC(6 个周期内)它不能再发出另一个激活命令。在本实例中还要求, 如果一个激活命令发出后, 下一个命令必须在 12000 周期内发出。

```

property p_tRC_not;
  @ (posedge clk)
    `s_active |->
      not ##[1: (tRC - 1)] `s_active;
endproperty

property p_tRC;
  @ (posedge clk)
    `s_active |->
      ##[tRC:12000] `s_active;
endproperty

a_tRC_not: assert property (p_tRC_not);
a_tRC: assert property (p_tRC);
c_tRC_not: cover property (p_tRC_not);
c_tRC: cover property (p_tRC);

```

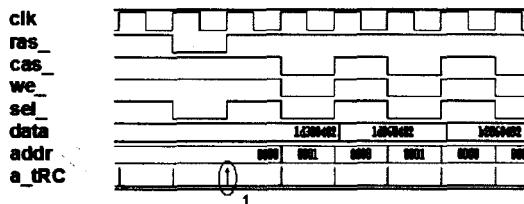


图 5-17 激活到下一激活命令, tRC

图 5-17 中第一个激活命令在标记 1 处, 下一个激活命令在 11,625 个周期抵达(未在图中显示), 所以检验器 a_tRC 成功。

SDRAM_chk6: tRFC, 自动刷新与下一次自动刷新之间必须间隔 tRFC 个周期。

这个属性与前一个非常相似, 在两个连续的自动刷新之间的时间窗口必须大于 tRFC 个周期。在本实例中还要求两个连续的自动刷新之间间隔不大于 12000 个周期。

```

property p_tRFC_not;
  @ (posedge clk)
    `s_autorefresh |->
      not ##[1: (tRFC-1)] `s_autorefresh;
endproperty

property p_tRFC;
  @ (posedge clk)

```

```

`s_autorefresh |->
##[tRFC:12000] `s_autorefresh;
endproperty

a_tRFC_not: assert property (p_tRFC_not);
a_tRFC: assert property (p_tRFC);

c_tRFC_not: cover property (p_tRFC_not);
c_tRFC: cover property (p_tRFC);

clk
ras_
cas_
we_
sel_
data
addr
a_tRFC

```

图 5-18 自动刷新与自动刷新命令, tRFC

图 5-18 在标记 1 处有一个自动刷新命令, 另一个自动刷新命令在 9 个周期后抵达(未在图中显示)。这一时间窗口大于存储器所要求的“tRFC”, 所以验证是成功的。

SDRAM_chk7: 读命令后 tCAS 才能跟写命令。

写命令不能紧接着读命令。我们定义读命令有 2 周期的延迟, 只有满足“cas”延迟的时间窗口后才能发出写命令。

```

property p_rd_wr;
  @ (posedge clk)
    `s_read |->
      not ##[0:tCAS] `s_write;
endproperty

a_rd_wr: assert property (p_rd_wr);
c_rd_wr: cover property (p_rd_wr);

```

SDRAM_chk8: tRP, 预充后必须“tRP”周期后才发出激活命令。

预充命令(禁用行)和激活(激活行)命令不能发生在 2 个周期的时间窗口内, 在本实例中还要求预充命令后, 激活命令必须在 12000 个周期内发出。

```

property p_tRP_not;
  @ (posedge clk)
    `s_preamble | ->
      not ##[0: (tRP - 1)] `s_active;
endproperty

property p_tRP;
  @ (posedge clk)
    `s_preamble | ->
      ##[tRP:12000] `s_active;
endproperty

a_tRP_not: assert property (p_tRP_not);
a_tRP: assert property (p_tRP);

c_trp_not: assert property (p_tRP_not);
c_trp: assert property (p_tRP);

```

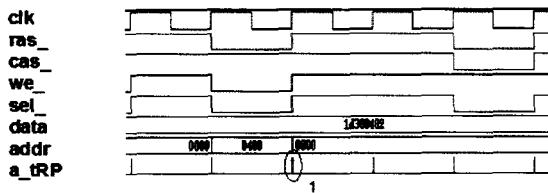


图 5-19 预充到激活命令, tRP

图 5-19 中预充发生在标记 1 处, 激活命令在 12000 个周期内被发出(未在图中显示), 所以, 该验证是成功的。

SDRAM_chk9: tRAS, 激活命令到预充命令必须发生在 tRASmin(5 个周期)到 tRASmax(12000 个周期)内。

激活命令到预充命令不能发生在“tRASmin”个周期内, 必须发生在“tRASmax”个周期内。

```

property p_tRAS_not;
  @ (posedge clk)
    `s_active | ->
      not ##[0: (tRAS_min - 1)] `s_preamble;
endproperty

property p_Tras;
  @ (posedge clk)

```

```

`s_active |->
##[tRAS_min:tRAS_max] `s_preamble;
endproperty

a_tRAS_not: assert property (p_tRAS_not);
a_tRAS: assert property (p_tRAS);
c_tRAS: cover property (p_tRAS);
c_tRAS_not: cover property (p_tRAS_not);

```

SDRAM_chk10: 不允许背靠背写命令(Back to back writes)。

```

property p_wr_wr;
@(posedge clk)
`s_write |->
not ##1 `s_write;
endproperty

a_wr_wr: assert property (p_wr_wr);
c_wr_wr: cover property (p_wr_wr);

```

SDRAM_chk11: 检查在读/写操作中自动预充是否被禁用。

现在大部份 SDRAM 可以在读/写操作中把 addr[10] 设成高电平而实现自动预充，以下的断言用蕴含和逻辑操作定义而成。

```

property p_disable_autoprecharge;
@(posedge clk)
(`s_write || `s_burst_write || 
`s_read || `s_burst_read) |->
addr[10] == 0;
endproperty

a_disable_autoprecharge:
assert property(p_disable_autoprecharge);

```

图 5-20 显示在读/写操作中禁用自动预充的波形图。

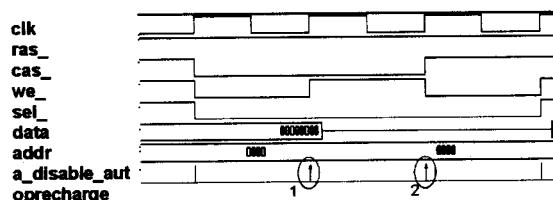


图 5-20 禁用自动预充

在标记 1 处有一写操作，`addr[10]`是 0，在标记 2 处当一个读操作在进行中，`addr[10]`被设为 0。

SDRAM_chk12: tRRD，对不同模块组的激活操作的最小时间间隔由 tRRD (2 个周期) 定义。

SDRAM 通常都有多个模块组，对不同模块组的激活操作有最小时间间隔的限制，本实例有 4 个模块组。

```

property p_tRRD;
  @(posedge clk)
  (<`s_active && bank_addr[1:0] == 0)|->
    not ##[0: tRRD] (`s_active &&
    bank_addr[1:0] != 0));
endproperty

a_tRRD: assert property(p_tRRD);
c_tRRD: cover property (p_tRRD);

```

以下的检验器检验当一个激活命令对模块组 0 发出的“tRRD”周期内，不能对其他模块组(1, 2, 3)发出激活命令。同时这个检验器还需要重复检验 1, 2, 3 模块组是否满足这一要求。我们可以用 generate 语句和“for”循环来实现，详见以下：

```

genvar j;
generate
  for (j=0; j<4; j++)
  begin:loop
    a_generate: assert property(@(posedge clk)
      (`s_active && bank_addr[1:0] == j)
      |-> not ##[1: tRRD] (`s_active &&
        (bank_addr[1:0] != j)));
    c_generate: cover property(@(posedge clk)
      (`s_active && bank_addr[1:0] == j)
      |-> not ##[1: tRRD] (`s_active &&
        (bank_addr[1:0] != j)));
  end
endgenerate

```

SDRAM_chk13: 如果 “`data_size`” 是 128，那么检验掩码运算。

CPU-AHB 总线可以定义数据的大小，因而可以以 128, 64 和

32 位对存储器进行写操作。而最常用的位宽是 32 位。但在使用 128/64 位时，我们使用掩码来把 32 位的数据块写入同一地址来实现。

```
property p_xfer128;
  @(posedge clk)
  ((size == 0) && ((dqm[0] == 0 && (`s_write
    || `s_burst_write))) |-
    ##2 ($fell (dqm[1]) && addr == $past (addr, 2)
  &&
    (`s_write || `s_burst_write))
    ##1 $rose (dqm[1]) ##1 ($fell (dqm[2]))
    && addr == $past (addr, 2)
    && (`s_write || `s_burst_write))
    ##1 $rose (dqm[2]) ##1 ($fell (dqm[3]) && addr
      == $past (addr, 2)
  && (`s_write || `s_burst_write))
    ##1 $rose (dqm[3]);
  endproperty

  a_xfer128: assert property(p_xfer128);
  c_xfer128: cover property(p_xfer128);
```

图 5-21 显示的是 128-比特的数据传输，数据被分成 4 块写入同一地址，每块是 32 比特，标记 1 处是第一块 32 比特被写入地址 0x0021，标记 2 处是第 4 块 32 比特的数据被写入地址 0x0021，掩码 dqm[3:0] 用来控制把数据写入存储器。

当传输的数据是 32 比特，数列 dqm[3:0] 的每一位都设为 0。

当传输的数据是 64 比特，数据被转换成两块，每块 32 比特，当第一块数据被转换，dqm[1:0] 设为 0，当第二块数据被转换，dqm[3:2] 设为 0。

对于 128 比特的传输，当 dqm[0] 被设为 0 时，第一块 32 比特数据被写入存储器，当 dqm[1] 被设为 0 时，第二块数据写入存储器，相同地当 dqm[2] 和 dqm[3] 被设为 0 时，第三和第四块相应地数据写入存储器。

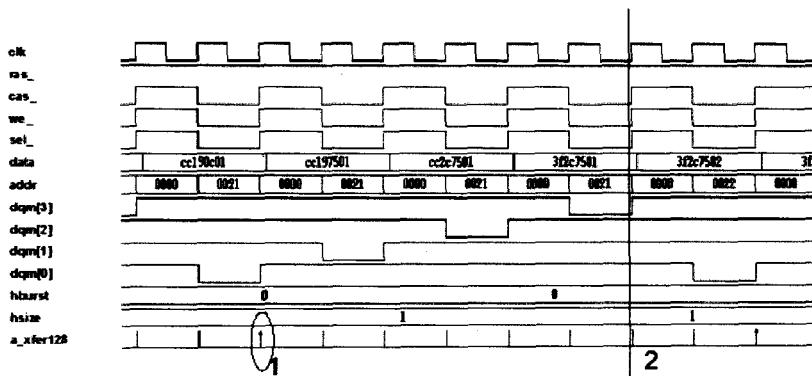


图 5-21 128-位数据传输

SDRAM_chk14: 当“`data_size`”是 64，每个读/写操作需要 2 个周期。

这特性与前一个相似，数据分成两块被写入，每块是 32 比特。当 `dqm[0] == 0` 和 `dqm[1] == 0`，第一块 32 比特数据写入；当 `dqm[2] == 0` 和 `dqm[3] == 0` 时，第二块 32 比特数据被写入。

图 5-22 显示的是 64-位的数据传输，在标记 1 处，第一块 32 比特数据写入地址 0x0103，并且掩码信号 0 和 1 都设为 0；在标记 1 处，第二块 32 比特数据写入相同地址 0x0103，掩码信号 2 和 3 都设为 0。

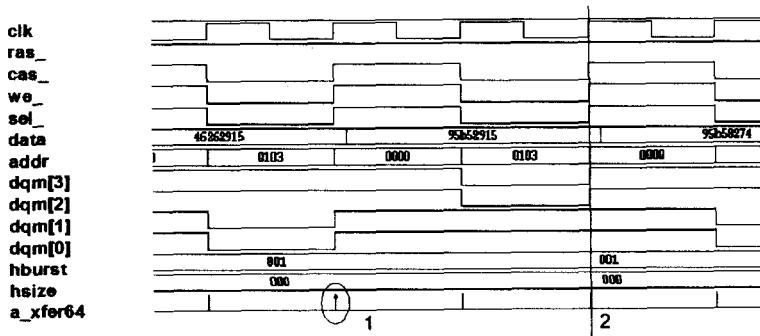


图 5-22 64-位数据传输

```

property p_xfer64;
  @(posedge clk) ((size == 1) && ((dqm[1:0] == 0
  && (`s_write || `s_burst_write))) | ->
  ##2 ($fell (dqm[2] && dqm[3]) && addr == $past (addr,

```

```

2) && (`s_write || `s_burst_write))
##1 $rose (dqm[3] && dqm[2]));
Endproperty

a_xfer64: assert property(p_xfer64);
c_xfer64: cover property(p_xfer64);

```

SDRAM_chk15: 由突发终止来停止读/写操作。

突发终止命令用停止来读/写突发命令，所以在发出突发终止命令的前一周期肯定有一个读/写操作。

```

property p_wr_rd_burstterminate;
@(posedge clk) (s_burstterminate) |->
$past ((`s_burst_write || `s_write || `s_read ||
`s_burst_read), 1);
endproperty

p_wr_rd_burstterminate:
assert property(p_wr_rd_burstterminate);
c_wr_rd_burstterminate:
cover property(p_wr_rd_burstterminate);

```

SDRAM_cover_chk1: 由突发停止结束写操作。

作为验证的一部分，某些情景和属性必须被覆盖。例如，在属性(p_wr_rd_burstterminate)中，已发出了“突发终止”的命令，那么在发出命令的前一周期必须有一个“突发写”或“突发读”的命令。但验证的结果并没有指明是哪个操作(读或写)被突发终止命令所停止，由于用 OR 逻辑所有组合都有可能。为提取这种情景信息，我们把这种属性区分开，用 cover 语句表示。

我们用另外一个属性来检验是否“突发写”被“突发终止”命令所停止。如果这一属性被检验，它有可能由于“突发读”被“突发终止”命令终止而失败。所以为了收集情景覆盖的信息没有必要再断言一个断言。

```

property p_wr_burstterminate;
@(posedge clk)
(s_burstterminate) |->
$past ((`s_burst_write || `s_write), 1);
endproperty

```

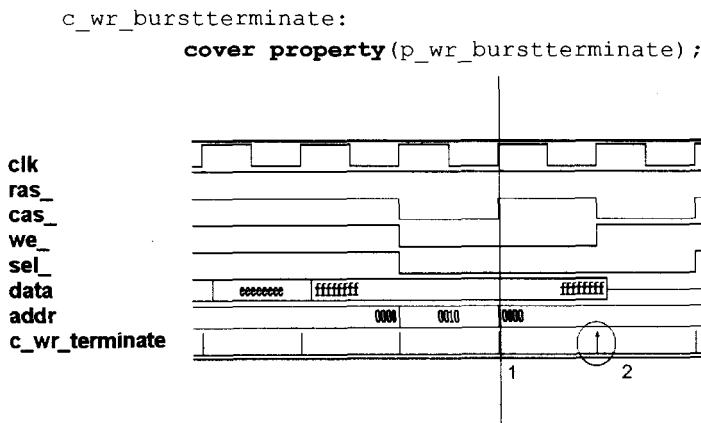


图 5-23 由突发停止结束写操作

图 5-23 在标记 1 处有一写命令，而在标记 2 处是突发终止命令，cover 语句是成功的，因为突发终止命令的前一周期有一写操作。

SDRAM_cover_chk2: 突发终止命令终止读操作。

与前断言相似，检验读操作由突发终止命令所停止，该断言检查如果在当前周期中有突发终止命令，那么前一周期必须有写操作或突发写操作。

图 5-24 在标记 1 处有一读命令由突发终止命令所终止(标记 2)。但由于读命令有“cas”延迟，读命令在发出 2 个周期后才终止，如标记 3 所示。数据未完成之前不能发出其他命令。

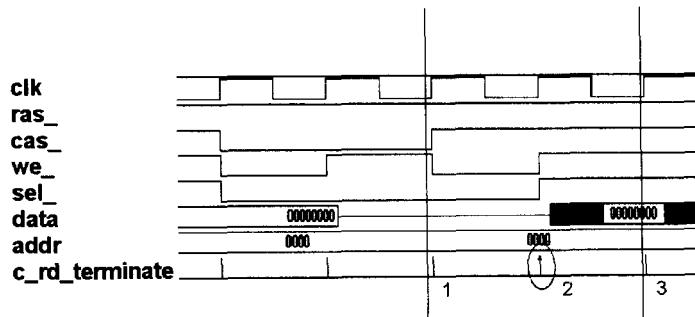


图 5-24 突发终止命令终止读操作

```

property p_rd_burstterminate;
  @(posedge clk) (s_burstterminate) ->
    $past ((`s_burst_read || `s_read), 1);
endproperty

c_rd_burstterminate:
  cover property(p_rd_burstterminate);

```

SDRAM_cover_chk3: 验证写操作被读操作所终止。

写操作可以被读操作所终止，如果一个写操作在进行中，控制器发出一个读操作，则写操作立即被停止。与前例一样，我们无需检验这一属性，由于写命令可能用其他方式终止，或由其他命令跟随，如果对这一属性检验的话，会产成非必要的失败。

图 5-25，在标记 1 处一个写命令被一个读取命令所终止(标记 2)，而读取命令被突发终止命令所停止。本实例中的突发大小是 4，但本实例的读/写命令只操作一次后就被终止了。

```

property p_wr_rdterminate;
  @(posedge clk) (`s_write || `s_burst_write)##1 (`s_read || `s_burst_read);
endproperty

c_wr_rdterminate :
  cover property(p_wr_rdterminate);

```

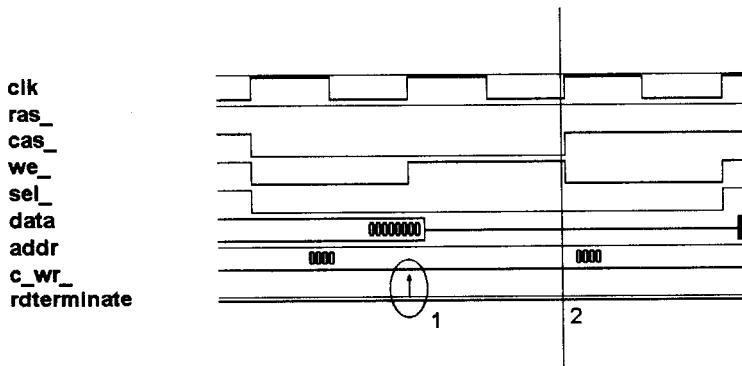


图 5-25 验证写操作被读操作所终止

5.3 SRAM/FLASH 的验证

SRAM(静态 RAM)是一种存储器，只要有电源接通，它就可以一直保存数据而无需外部的刷新。

SRAM 速度比 SDRAM 快很多。

SRAM 价格昂贵且占更大的空间。

由于没有复杂的刷新机制，SRAM/FLASH 的验证非常简单。当“写”和“芯片选择”信号被断言，数据开始往地址总线所指定的位置写入；当“芯片选择”信号被断言，“写”信号被取消而输出激活信号被断言时，数据从存储器中读出。以下是本实例所使用的静态存储器：

SRAM: 256K X 16bit 高速静态 RAM

FLASH: 128Mbit 闪存 (16Mbytes)

表 5-3 是本实例所使用的 SRAM 的时序参数，表 5-4 是本实例所使用的闪存的时序参数。

表 5-3 SRAM 的时序参数

参 数	符 号	Min	Max
写操作周期	tWC	1 个周期(10ns)	-
写脉冲宽度	tWP	2 个周期(20ns)	-
读操作周期	tRC	1 个周期(10ns)	-
芯片选择到输出	tCO	1 个周期(10ns)	-
地址访问时间	tAA	1 个周期(10ns)	-

表 5-4 闪存的时序参数

参 数	符 号	Min	Max
写/读操作周期	tAVAV	15 个周期(150ns)	-
芯片选择到输出滞后	tELQV	-	15 个周期(150ns)
页面地址访问时间	tAPA	-	3 个周期(25ns)

SRAM/FLASH 的断言

SRAM_chk1: 验证写操作周期, tWC。

SRAM 设计规格中要求写操作周期时间必须大于 tWC。写操作周期时, 写周期时间是在芯片选择和写激活信号被断言后地址值保持稳定的时间。

本实例使用 \$stable 和蕴含操作符来实现这个断言, \$stable 用来保证当前时间周期内的地址值与前一周期内的一致。

```
property p_tWC;
@(posedge clk)

($fell (we_n) && !sel_n[2]) |=>

$stable(addr[22:0]);
endproperty
a_tWC: assert property(p_tWC);
c_tWC: cover property(p_tWC);
```

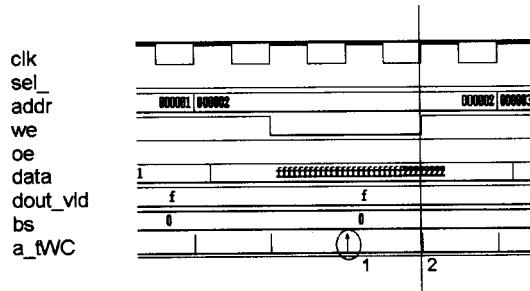


图 5-26 写操作周期, tWC

图 5-26 在标记 1 处是写命令, 断言在标记 2 处成功, 因为在写命令发出后地址至少保持了一个时间周期的稳定值。

SRAM_chk2: 验证写激活脉冲宽度, tWP。

该断言用来检验验证写激活脉冲宽度是否大于设计规格的要求(2 时钟周期)。图 5-27 显示在标记 1 处是写命令的下降沿, 标记 2 处是写操作的上升沿, 它们相距 2 个周期。

```

property p_tWP;
@(posedge clk)
$fell (we_n) |->
    ##tWP $rose (we_n);
endproperty

a_tWP: assert property (p_tWP);
c_tWP: cover property (p_tWP);

```

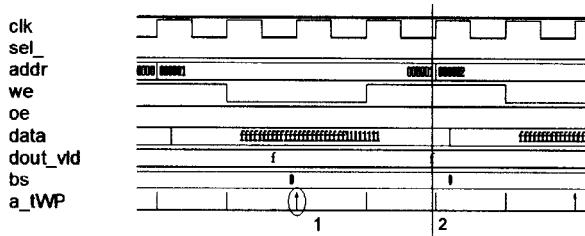


图 5-27 写激活脉冲宽度, tWP

SRAM_chk3: tRC -验证读操作周期。

这个验证和写操作周期验证相类似。写操作周期是在芯片选择和输出信号被断言后地址值保持稳定的时间。图 5-28 在标记 1 处芯片选择和输出信号被断言，地址值在标记 1 和标记 2 之间保持不变。

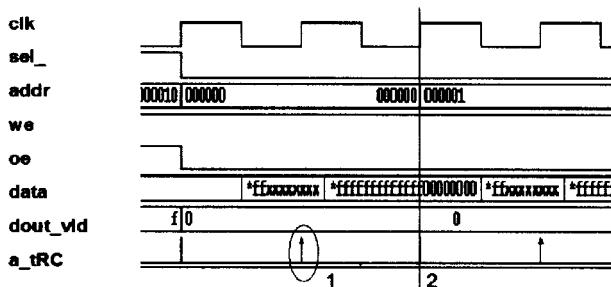


图 5-28 读周期时间, tRC

```

property p_tRC;
@(posedge clk)
(!sel_n[2] && we_n && !oe_n) |=>
    ($stable (addr));

```

endproperty

```
a_tRC: assert property(p_tRC);
c_tRC: cover property(p_tRC);
```

SRAM_chk4: tCO - 验证芯片选择到数据输出的间隔是否正确。

参数“tCO”是从芯片选择被断言到合法数据输出之间的最小时间值。图 5-29 在标记 1 处芯片选择和输出信号被断言，在这一时钟周期中，数据值为“x”。一个周期后，数据值为“1”（标记 2 处）。

```
property p_tCO;
@(posedge clk)
(!sel_n[2] && we_n && !oe_n &&
($isunknown (data))) |=>
($isunknown (data))==0;
endproperty

a_tCO: assert property(p_tCO);
c_tCO: cover property(p_tCO);
```

SRAM_chk5: tAA - 验证合法地址到合法数据的时间间隔。

参数“tAA”是从合法地址后到合法数据之间的最小时间间隔。图 5-30 在标记 1 处有一读命令，在当前周期的地址值必须在下一周期仍保持不变，在标记 2 处数据显示为合法。

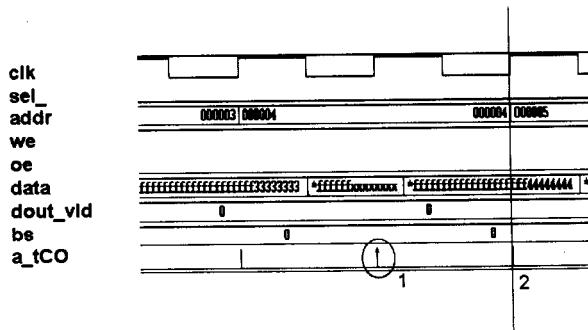


图 5-29 芯片选择到数据输出 tCO

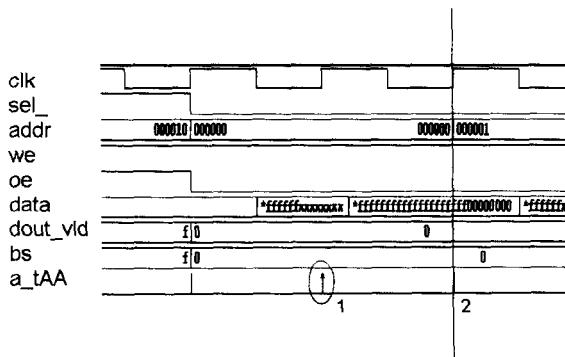


图 5-30 合法地址到合法数据, tAA

```

property p_tAA;
  @(posedge clk) (!sel_n[2] && we_n && !oe_n) |=>
    ((addr == $past(addr, 1)) ##0
     ($isunknown(data))==0);
endproperty
a_tAA: assert property(p_tAA);
c_tAA: cover property(p_tAA);

```

FLASH_Clk1: 验证闪存的写保护。

本实例所使用的闪存是有写保护的，所以每当闪存的芯片选择信号被激活，必须确认保护信号(wp_)已被断言。

```

property p_write_protect;
  @(posedge clk)
  (!sel_n[3]) |->
    wp_n == 0;
endproperty

a_write_protect:
assert property (p_write_protect);
c_write_protect:
cover property (p_write_protect);

```

FLASH_chk2: 验证完成读操作的时钟周期(tAVAV)。

在表 5-4 中完成读操作的最小时钟周期是 tAVAV(15 个时钟周期)。本实例还要求完成读操作的时间不能超过 900 个时钟周期，所以我们用两个检验器来验证最大和最小时序要求。

```

property p_tAVAV_not;
  @(posedge clk)
  (!sel_n[3] && $fall(oe_n)) |->
    not ##[0:15] $rose (oe_n);
endproperty

property p_tAVAV;
  @(posedge clk)
  (!sel_n[3] && $fall (oe_n)) |->
    ##[16:900] $rose (oe_n);
endproperty

a_tAVAV: assert property(p_tAVAV);
a_tAVAV_not: assert property(p_tAVAV_not);

c_tAVAV: cover property(p_tAVAV);
c_tAVAV_not: cover property(p_tAVAV_not);

```

FLASH_chk3: 验证芯片片选和地址值 CS/ADDR 到合法数据的间隔 tELQV。

我们用“tELQV”来定义从芯片选择和地址值稳定到合法数据输出的最长时间，它为 15 个时钟周期。在图 5-31 中，在标记 1 处信号“sel”被断言，15 个时钟周期后(标记 2 处)，第一合法数据出现。

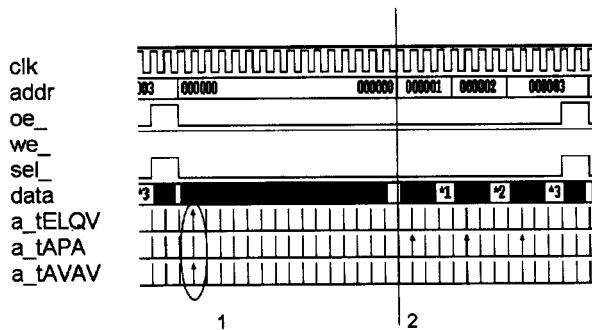


图 5-31 tELQV, tAPA, tAVAV 的波形图

```

property p_tELQV;
  @(posedge clk)
  (!oe_n && $fall (sel_n[3])) |->
    ##14 $isunknown(data)##1 ($isunknown (data)==0);

```

```
endproperty
```

```
a_tELQV: assert property(p_tELQV);
c_tELQV: cover property(p_tELQV);
```

FLASH_chk4: 验证地址到合法数据输出的间隔, tAPA。

参数 tAPA(3 个时钟周期)是地址必须保持稳定后取得合法数据的最长时间值。该参数只对一个突发读取命令中的后续读取操作(subsequent read)有效, 而对突发命令中的第一个读取操作无效。

图 5-32 中显示一个突发读取命令。在这个命令中, 当地址值变化时, tAPA 个时钟周期后, 新数据会被读入。在标记 1 处, 地址由“000000”变为“000001”, 但在这一时间点数据仍是未知的, 只有在 3 个周期后(标记 2 处), 才能取样到合法数据。

```
sequence s_data_trans;
  (!sel_n[3] && !oe_n && ($stable(addr)==0) &&
   $stable(oe_n))##0 $isunknown(data)
  ##3 $isunknown(data)==0;
endsequence

property p_tAPA;
  @ (posedge clk)
    s_data_trans |>
      $stable(addr);
endproperty

a_tAPA: assert property(p_tAPA);
c_tAPA: cover property(p_tAPA);
```

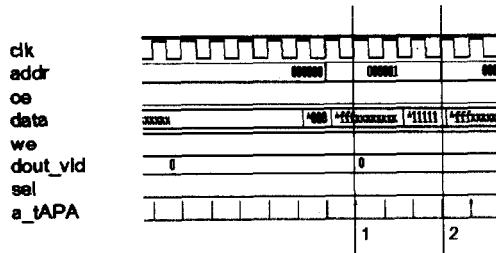


图 5-32 tAPA 波形图

5.4 DDR-SDRAM 的验证

双数据率同步动态随机存儲器(Double Data Rate Synchronous Dynamic Random Access Memory, DDR-SDRAM)与同步存储器类似,但它有着更高的带宽,它在时钟的上升沿和下降沿都可以进行读写。它的操作与 SDRAM 类似,本实例所使用的 DDR-SDRAM 有以下设置:

DDR-SDRAM: 4M 词 × 16 比特 × 4 模块组

DDR-SDRAM 的读和突发读操作与 SDRAM 类似,通过断言“sel_”和“cas_”和保持“ras_”和“we_”高电平来发出突发读命令。地址输入值决定突发操作的起始地址,第一个输出数据出现在读命令发出后再滞后“cas”个时钟周期(按照 DDR-SDRAM 的设计要求是 2 个时钟周期),后续数据出现在数据选通(data strobe)信号的上升和下降沿处。

通过断言“sel_”,“cas_”和“we_”以及在时钟(clk)的上升沿取消“ras_”来发出突发写操作。信号“dqs”的到来有一个时钟周期的滞后,而写命令相对于“dqs”信号没有滞后。

DDR-SDRAM 的断言

DDR_Chk1: 验证 DDR 突发读操作。

DDR 存储器有多个时钟,时钟“clk2x”在数据传输和读取时用,在上升和下降沿都对数据取样。大多数 SDRAM 的检验器可以为 DDR 所复用,我们用一个新定义的断言来验证是否跨越时钟域。

SVA 的关键词 matched 用来同步跨越时钟域的信号,该断言的 cas_, ras_, we_ 和 sel 由时钟“clk”产生,而数据在时钟“clk2x”负下降沿被读取,所以我们必须使用 matched 结构来同步从一个时钟域到另一个时钟域的读取顺序。

```
sequence s_read;
  @ (posedge clk)
```

```

(ras_n && !sel_n[0] && we_n && !cas_n);
endsequence

property p_read;
@(posedge clk2x) s_read.matched |->
##3 ($isunknown(data))
##1 ($isunknown(data)==0);
endproperty

a_read: assert property(p_read);
c_read: cover property(p_read);

```

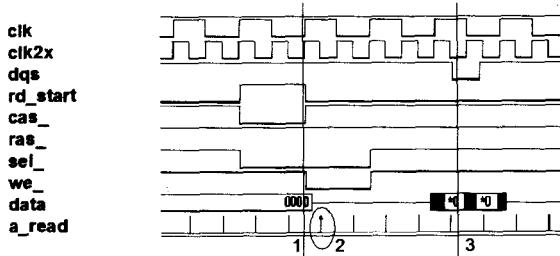


图 5-33 DDR-SDRAM 突发读操作

图 5-33 在标记 1 处有一基于时钟“clk”的读取命令(s_read)，在时钟“clk2x”最近的下降沿处对该操作顺序的匹配值取样(于标记 2 处)，在 4 个时钟周期的延迟后(标记 3 处)数据被读出，数据在信号“dqs”的上升和下降沿被读出，信号“dqs”是基于时钟(clk2x)，所以时钟(clk2x)的下降沿用来对该数据取样。

DDR_Chk2: 验证 DDR 突发写操作。

```

sequence s_write;
@(posedge clk)
(ras_n && !sel_n[0] && !we_n && !cas_n);
endsequence

property p_write;
@(posedge clk2x)s_write.matched
|-> ##1 ($isunknown(data)==0)
##1 ($isunknown(data)==0);
endproperty

a_write: assert property(p_write);
c_write: cover property(p_write);

```

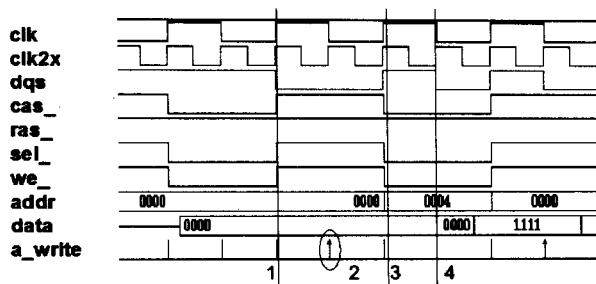


图 5-34 DDR 突发写操作

图 5-34 显示在标记 1 处有一个写命令(基于时钟(clk))，在标记 2 处写命令与 clk2x 的正沿相同步，由于信号“dqs”是基于时钟(clk2x)而产生，时钟(clk2x)的正沿用于对写命令取样。标记 2 处的验证结果是成功的，因为数据在选通信号“dqs”的双沿被写入存储器，如标记 3 和 4 所示。

5.5 存储器 SVA 的小结

可以用断言来方便地验证存储器的时序要求。

所有相关存储设备的时序信息应含可变参数，这样开发用于一种存储器的断言才可以在其他不同厂商的存储器上得到复用。

存储器的断言可以提供情景覆盖率的信息。例如，是否一个写访问由一个读/突发操作所终止？是否一个读访问被一个突发操作所终止？是否执行过背靠背(back to back)写访问？是否一个写访问接着读访问？是否测试覆盖了各种数据位宽(128/64/32 位等)。这有助于我们增加对验证的信心，同时提供一个尺度来测量验证的完整性。



SVA 协议接口

—— PCI 系统的 SVA 检验器

兼容性测试已成为 SOC 设计中一个主要难点。设计通常需要支持某些标准协议，例如，图形芯片可能需要支持标准总线接口，如 PCI/PCIX、USB 或 IEEE1394。这些总线接口让设计能够达到更高的数据传送带宽，同时也为连接多个设备提供一个标准方案。总线协议通常比较复杂，而总线上的各个设备必须符合一系列协议规则。

由于同一系列规则适用于任何支持某特定协议的设备，所以用来测试这种标准协议接口的验证环境通常是可重用的。验证工程师常开发支持某种特定接口协议的总线接口模型(BIM)。该模型并不需要复制设备的内部功能细节，它只需要兼容特定接口的握手协议。这可以帮助验证工程师创建一个包括总线接口模型和被测设计(DUT)的样品系统，继而写出测试来创建该总线接口模型和被测设计之间的事务。在运行测试的同时，编写特定的监视器以确保被测设计完全符合标准协议。大多数验证环境创建总线事务日志。用 SVA 能够非常有效地创建这种总线协议监视器。在本章中，我们利用一个 PCI 系统的实例来展示如何为 PCI 兼容设备创建 SVA 检验器。

6.1 PCI 简介

PCI 局部总线是一种分时复用地址和数据的高性能的 32/64 位总线。该总线用于连接高集成外围控制器元件，外围附加电路板和处理器内存系统。图 6-1 显示了一个 PCI 兼容设备的实例。

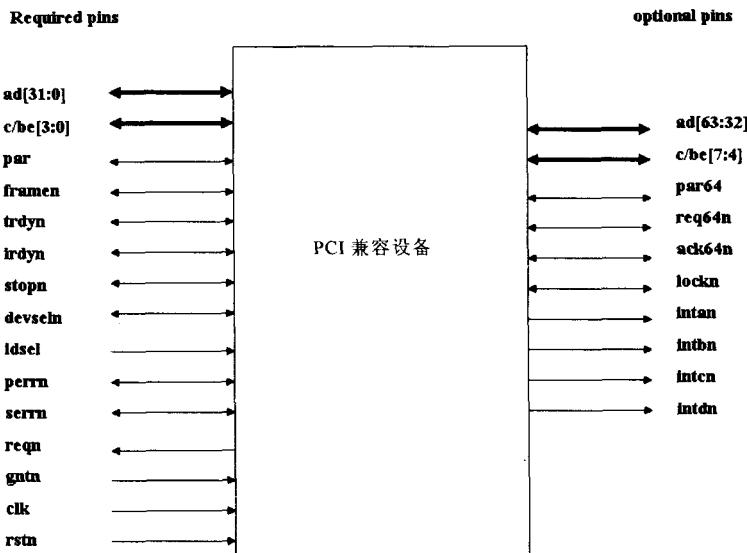


图 6-1 PCI 兼容设备

下面是每一个引脚的简单描述。

ad[31:0] – 地址总线，包含传输数据读出或写入的地址信息。它也是数据总线。

c/be[3:0] – 指令总线，含表 6-1 所示的 12 条指令之一。它也是字节允许总线，定义数据总线中可以传输哪些字节。

par – 奇偶校验位，在 ad, c/be 和 par 中应该一共有偶数个 1 出现。该校验位的值被驱动的时刻是在“ad”总线被驱动时刻的一个时钟周期之后。

framen – 帧信号，被事务主控所断言。当帧信号被断言的同时，主控在“c/be”总线上设置适当的指令来表示事务性质。当主控即将完成最后的数据传输时，该帧信号被取消断言。

表 6-1 PCI 总线指令

C/BE[3:0]	命 令 类 型
0000	中断确认(Interrupt Acknowledge)
0001	特殊周期(Special Cycle)
0010	I/O 读
0011	I/O 写
0100	保留
0101	保留
0110	存储读(Memory Read)
0111	存储写(Memory Write)
1000	保留
1001	保留
1010	配置读(Configuration Read)
1011	配置写(Configuration Write)
1100	多次存储器读(Memory Read Multiple)
1101	双重地址周期(Dual Address cycle)
1110	存储读线
1111	存储写、无效(Memory Write and Invalidate)

trdyn – 目标预备信号，由主控当前所给地址对应的目标设备断言。设备通过断言该信号来告知主控已准备好进行事务的状态。

irdyn – 主控预备信号，由需要做事务的主控断言。

stopn – 停止信号，由需要终止当前事务的目标设备断言。如果目标设备在未进行事务的时刻断言该停止信号，这被称为重试(retry)。如果目标设备再进行了一个或多个数据阶段事务之后断言该停止信号，则被称为断路(disconnect)。

devseln – 设备选择信号，选择目标设备时断言该信号。只有当断言了该信号之后，才断言目标预备信号 trdyn。

idsel – 初始设备选择信号，在进行 PCI 配置读写事务时作为芯片选择信号。

perrn – 校验位错误信号，在主控或从设备发现校验位出错时刻的一个时钟周期后断言。

serrn – 系统错误信号，主控和从设备的输出信号。当致命错误发生时被断言。

reqn – 请求信号，主控设备使用该信号发出 PIC 总线的使用请求。

gntn – 授予信号，表示 PCI 设备有使用 PCI 总线的许可。

ad[63:32] – 64 位数据总线中 32 高位，用于 64 位的事务。

c/be[7:4] – 字节使能总线，定义 64 位事务中高位字节中该传输哪一个字节。

par64 – 奇偶校验位，在 **ad[63:32]**、**c/be[7:4]** 和 **par64** 中应该共有偶数个 1 出现。当设备驱动 “ad” 总线一个时钟后 **par64** 被驱动。

req64n – 请求信号，在 64 位事务中主控设备使用该信号发出使用 PIC 总线的请求。

ack64n – 确认信号，PCI 目标设备确认由主控设备发出的 64 位事务请求。

6.1.1 一个 PCI 读出事务的实例

读出事务由主控设备发起。主控断言 “framen” 信号并对 “ad” 总线写入一个地址，同时也对 “c/be” 总线写入一个读命令。目标设备对地址解码来识别自己。一旦识别自己，它断言 “devseln” 信号。主控设备继续断言 “framen” 信号，但停止驱动地址总线。它断言信号 “irdyn”的同时也在 “c/be” 总线上发字节允许命令。作为响应，目标设备将第一部分数据放在数据总线(ad)上并断言 “trdyn” 信号来确认总线上的数据有效。在多重事务中，被地址选中的目标设备通过递增初始地址来指向后续数据的地址。

在进行事务的过程中，如果目标设备还未准备好将下一个数据阶段写入总线，它将通过取消断言 “trdyn” 信号来创建等待状态。“devseln” 信号将保持被断言的状态，而总线上前一次的数据也保留下。只有当同时断言 “trdyn” 和 “irdyn” 时，主控才读入数据。目标设备再次准备传输时断言 “trdyn” 信号。主控设备通过取消断言 “framen” 信号来表示下一次数据读入将是当前事务中的最后一次。一旦读入最后一次数据，主控设备取消断言

“irdyn”信号，目标设备分别取消断言“trdyn”和“devseln”信号。当“irdyn”和“framen”同时取消断言时，我们称总线处于停滞状态。图6-2显示了一个PCI读入事务的波形图实例。

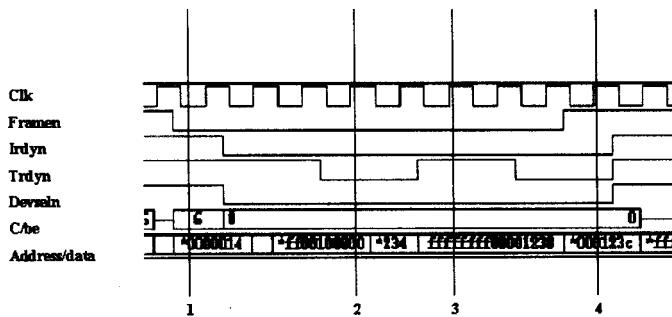


图 6-2 PCI 读入事务实例

标记1显示了主控在“c/be”总线上发出读命令(0110)的时间点。同一个周期内，地址总线载有目标设备的地址。标记2显示了主控读有效数据的时间点。标记3显示目标设备取消断言“trdy”信号来示意未准备好来读入数据。标记4表示“framen”被取消断言指示到达最后数据阶段。在下一个时钟周期，信号“irdyn”，“trdyn”和“devseln”全部被取消断言，表示事务的完成。

6.1.2 PCI 写入事务实例

主控初始化一个写入事务。它断言“framen”信号并向“ad”总线送入一个地址，同时也向“c/be”总线上送入一个写命令。目标设备识别自己并断言“devseln”和“trdyn”信号。主控继续断言“framen”信号。主控通过向“ad”总线上送入数据并断言“irdy”信号来告知从设备总线数据有效。主控也发出命令字节用于识别在同一时钟周期中哪一个字节被写入。如果主控未预备将下一段数据送入总线，它可取消断言“irdyn”信号来创建一个等待状态。在该等待状态中，主控将驱动与前一个时钟周期相同的数据。主控在预备写入最后一段数据前将取消断言“framen”信号。当所有数据写入完成，主控取消断言“irdyn”信号的同时从设备取消

断言“trdyn”和“devseln”信号。图 6-3 显示了 PCI 写入事务的波形图实例。

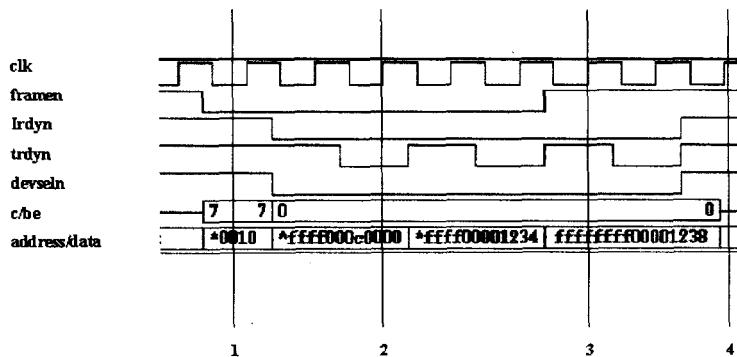


图 6-3 PCI 写入事务实例

标记 1 显示了主控对“c/be”总线发出写信号(0111)。同一时钟周期内，地址总线载有目标设备的地址。标记 2 显示了主控写有效数据的时间点。标记 3 显示了从设备取消断言“trdy”信号来表示它未准备好进行写入操作。在同一个时钟周期内，主控设备取消断言“framen”信号来表示这是最后一个数据阶段。标记 4 显示信号“irdyn”、“trdyn”和“devseln”都被取消断言，表示写入事务完成。

6.2 PCI 系统实例

图 6-4 显示了一个 PCI 系统实例。该图有两个 PCI 主控和两个 PCI 目标设备。用户可能设计一个设备来作为 PCI 主控、PCI 目标设备或兼作两者，而可以利用总线接口模型作为图中实例系统中的其他三个设备来验证该被测设备(DUT)。一共有三种构建 SVA 检验器的可能情形，并作为验证计划一部分。我们将讨论这三种情形。

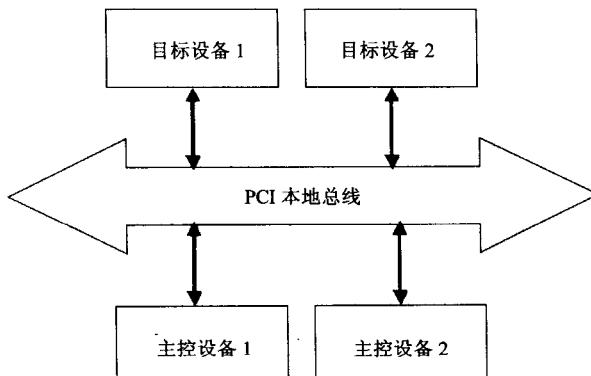


图 6-4 PCI 系统实例

6.3 情形 1——主控 DUT 设备

在本节中，我们假定 PCI 主控设备为待测设备。根据 PCI 局部总线规范，PCI 主控必须符合某些协议达到彻底兼容。在验证环境中常常使用监视器，这些监视器来保证 DUT 不违反协议规范。

监视器也能在日志文件中详细记录主控的所有事务，并用于事后的分析处理。SVA 可以用来构建一系列连接于 PCI 主控的通用检验器。由于 PCI 是一个标准协议，所开发的检验器应该能重用于其他任何 PCI 兼容的主控设备。图 6-5 显示了系统的配置实例。

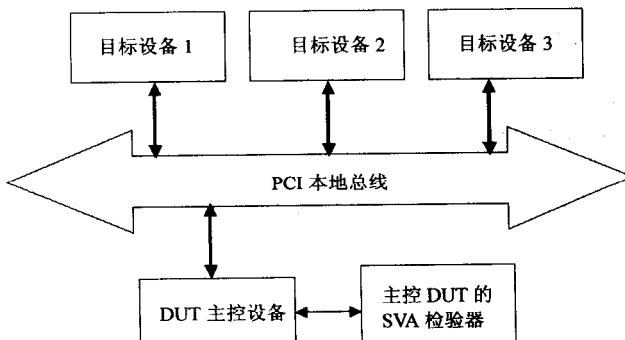


图 6-5 作为 DUT 的 PCI 主控设备的配置实例

PCI 主控的断言

在该节中，我们展示能够验证 PCI 主控功能的几个 SVA 检验器实例。下面定义了一些常用的设计条件便于重用。

```

`define s_IO_READ
    ($fell (framen) && (cxben[3:0] == 4'b0010))
`define s_IO_WRITE
    ($fell (framen) && (cxben[3:0] == 4'b0011))
`define s_MEM_READ
    ($fell (framen) && (cxben[3:0] == 4'b0110))
`define s_MEM_WRITE
    ($fell (framen) && (cxben[3:0] == 4'b0111))
`define s_CONFIG_READ
    ($fell (framen) && (cxben[3:0] == 4'b1010))
`define s_CONFIG_WRITE
    ($fell (framen) && (cxben[3:0] == 4'b1011))
`define s_DUAL_ADDR_CYCLE

    ($fell (framen) && (cxben[3:0] == 4'b1101))
`define s_MEM_READ_LINE
    ($fell (framen) && (cxben[3:0] == 4'b1110))
`define s_MEM_WRITE_INV
    ($fell (framen) && (cxben[3:0] == 4'b1111))
`define s_BUS_IDLE
    (framen && irdyn)

```

Master_chk1: 在给定的时钟周期内，除非“irdyn”一直被断言，否则“framen”将不能被取消断言。

```

property p_mchk1;
  @ (posedge clk)
    $rose (framen) |-> (irdyn == 0);
endproperty

a_mchk1: assert property(p_mchk1);
c_mchk1: cover property(p_mchk1);

```

主控设备在最后数据阶段事务中断言“framen”信号。所以，信号“irdyn”应该在此时保持被断言的状态。否则将是违反规范。图 6-6 显示了在模拟中该检验的一个波形图实例。标记 1、2、3、4 和 5 表示“framen”信号有上升沿以及所有相应的时钟沿的时刻，

在这些时刻，信号“irdyn”总是处于被断言状态。因此，该检验器验证成功。

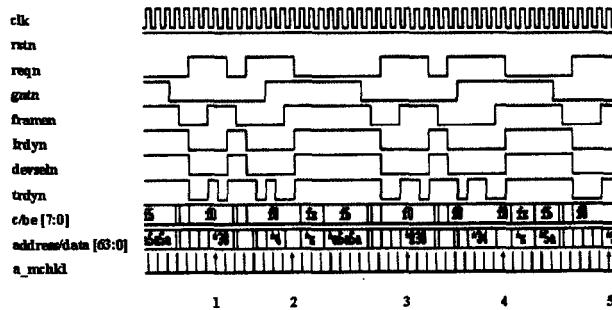


图 6-6 PCI 主控检验 1

Master_chk2: 一旦“framen”被取消断言，它不能在同一事务中被断言。

```
property p_mchk2;
  @(posedge clk) $rose (framen) |->
    framen[*1:8] ##0 $rose (irdyn && trdyn);
endproperty

a_mchk2: assert property(p_mchk2);
c_mchk2: cover property(p_mchk2);
```

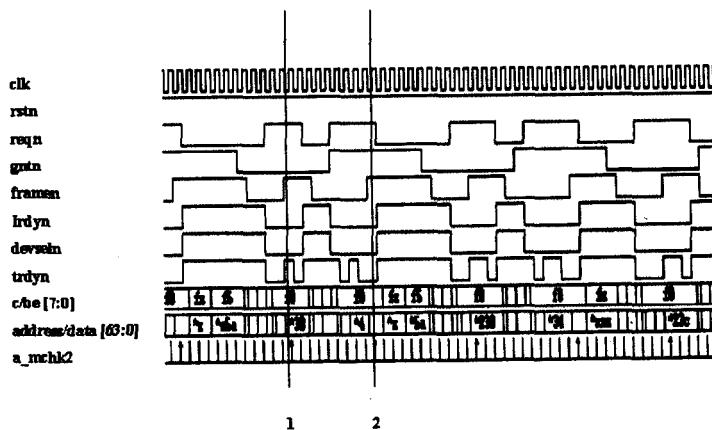


图 6-7 PCI 主控检验 2

一旦信号“framen”被取消断言，主控设备只剩下一个数据阶段。但是它可以用多个周期来完成最后一个数据阶段的事务。例如，如果目标设备未预备接受数据，那么主控会等待以便结束最后的数据阶段。在主控完成最后数据阶段之前，“framen”不能被再次断言。换言之，必须在再次断言“framen”之前先取消断言信号“irdyn”和“trdyn”。图 6-7 显示了在一次模拟中该检验对应的波形图实例。

标记 1 显示了该断言的一次成功。此时，信号“framen”有一次上升沿，所以该检验被激活。注意在同一时钟周期，信号“trdyn”被取消断言而示意目标设备未预备接受数据。在下一个时钟周期，断言信号“trdyn”和“irdyn”，因此最后的数据阶段结束。一个时钟周期之后，信号“irdyn”和“trdyn”被取消断言。标记 2 也显示了一次成功，但此种情形下，当“framen”出现上升沿时，断言信号“irdyn”和“trdyn”，所以最后数据阶段事务完成。在下一个时钟周期内，信号“irdyn”和“trdyn”被取消断言。

Master_chk3: 一旦断言了“irdyn”，主控在当前数据段开始之前不能改变“irdyn”和“framen”。

```

property p_mchk3;
  @(posedge clk)
  $fell (irdyn) ##[0:5]
    !(devseln) ##0 stopn  |->
      (!irdyn) [*0:16] ##0 !trdyn;
  endproperty

  a_mchk3: assert property(p_mchk3);
  c_mchk3: cover property(p_mchk3);

```

当一个主控设备断言信号“irdyn”时，如果假设目标设备不发出停止状态，那么我们期望有效数据段在 16 个时钟周期内将开始。当从设备断言“rdyn”时，该数据段便开始。自此，假设如果没有停止状态，在“irdyn”被断言后，信号“irdyn”则应该保持被断言状态，除非目标设备断言“trdyn”信号。

图 6-8 显示了一次模拟中该检验的一个波形图实例。标记 1 显示该检验的一次成功。此时断言“irdy”和“devseln”信号。一个周期后断言“trdyn”，因而检验成功。

标记 2 显示的情形中，信号“irdyn”和“devseln”都被断言，并在 2 时钟周期后，信号“trdyn”被断言。信号“irdyn”在“trdyn”信号到达之前保持被断言状态，因而该检验失败。

Master_chk4: 主控在“framen”被断言后的 8 个周期之内必须断言“irdyn”。

我们使用一个 **intersect** 结构来控制整个属性的长度。如果在 1~8 个时钟周期内该属性未成功，那么断言失败。

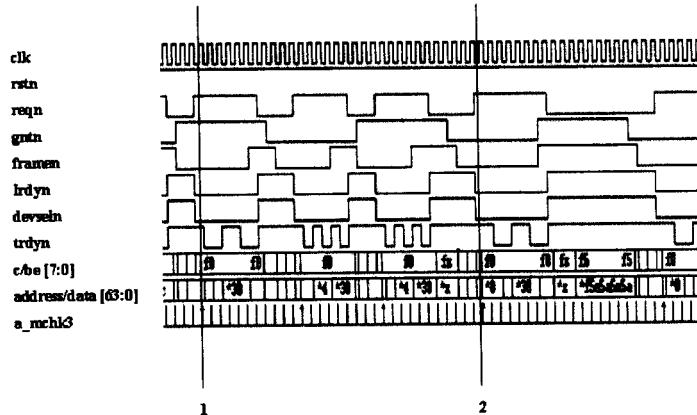


图 6-8 PCI 主控检验 3

```

property p_mchk4;
@(posedge clk)
$fell (framen) |->
    1[*1:8] intersect
        ($fell (framen) ##[1:$] $fell(irdyn));
endproperty

a_mchk4: assert property(p_mchk4);
c_mchk4: cover property(p_mchk4);

```

Master_chk5: 正常终止，当“framen”被取消断言，最后数据段在 8 个时钟周期内完成。

```

property p_mchk5;
@(posedge clk)
$rose (framen) |->
  (##[1:8] ($rose (irdyn && trdyn && devsln)));
endproperty

a_mchk5: assert property(p_mchk5);
c_mchk5: cover property(p_mchk5);

```

Master_chk6: 主控中止，“devseln”应该在“framen”断言的5个周期内被断言。如果“devseln”在5个周期内没有被断言，那么“framen”应该被取消断言，并且一个周期后“irdyn”也被取消断言。

```

sequence s_mchk6;
@(posedge clk)
$fell (framen) ##[1 (devseln)[*5] ##[0 framen;
endsequence

property p_mchk6;
@(posedge clk)
  s_mchk6.ended |-> ##[1 $rose (irdyn);
endproperty

a_mchk6: assert property(p_mchk6);
c_mchk6: cover property(p_mchk6);

```

图 6-9 显示了在一模拟中该检验的一个波形图实例。标记 1 显示了“framen”被检测为被断言状态的时钟边沿。如果信号“devseln”在之后的5个时钟周期仍未到达，那么主控应该中止这次事务。标记 2 显示了“devseln”没有到达的时钟边沿。标记 3 显示在下一个时钟边沿主控设备取消断言信号“irdyn”。由于该属性在序列 s_mchk6 成功结束时开始，标记 2 处表示了成功点。

Master_chk7: 当从设备通过重试(retry)或断连(disconnect)中止主控时，该主控必须在重复事务之前取消断言它发出的请求。该请求应该在总线进入空闲状态的时钟周期内和空闲状态前或后一个时钟周期内被取消断言。

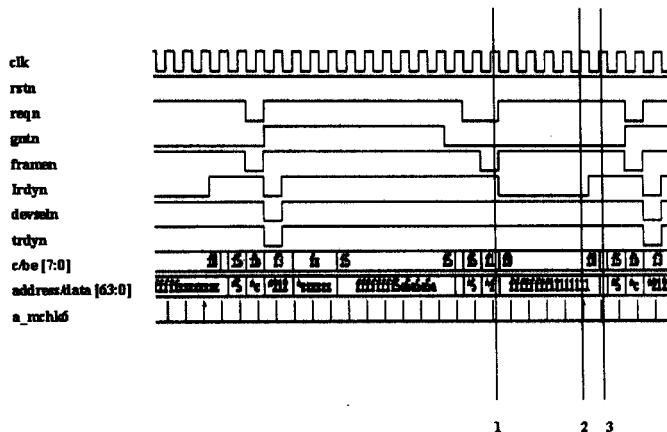


图 6-9 PCI 主控检验 6

```

sequence s_mchk7_before;
@(posedge clk)
    (!devseln && $fell (stopn) && trdyn)
    ##1 reqn ##1 `s_BUS_IDLE;
endsequence

sequence s_mchk7_after;
@(posedge clk)
    (!devseln && $fell (stopn) && trdyn)
    ##1 !reqn ##1 `s_BUS_IDLE;
endsequence

property p_mchk7_before;
@(posedge clk)
    s_mchk7_before.ended |->
        reqn;
endproperty

property p_mchk7_after;
@(posedge clk)
    s_mchk7_after.ended |->
        reqn [*2];
endproperty

a_mchk7_before: assert property(p_mchk7_before);
a_mchk7_after: assert property(p_mchk7_after);

```

```
c_mchk7_before: cover property(p_mchk7_before);
c_mchk7_after: cover property(p_mchk7_after);
```

此检验需要两个属性。主要的要求是：如果从设备发出一个停止状态，主控则在再次请求总线之前让“reqn”信号被取消断言。而主控在停止状态实际到达之前有可能已经取消断言了信号“reqn”。这种情况下，验证以下两个属性：当总线空闲的时钟周期中“reqn”被取消断言，并且“reqn”在前一周期内被取消断言(p_mchk7_before)。

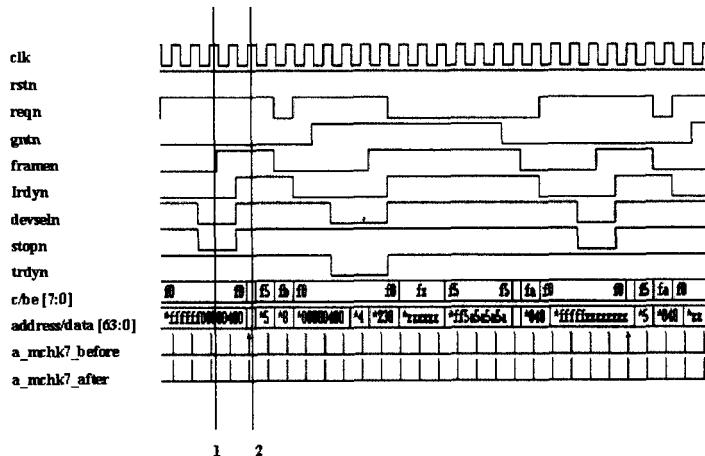


图 6-10 PCI 主控检验 7

如果主控在停止状态到达之前没有取消断言“reqn”信号，那么在总线空闲周期以及其后的一个周期内信号“reqn”被取消断言的属性被验证(p_mchk7_after)。注意，当从设备断言信号“stopn”两个周期之后总线变为空闲。图 6-10 显示在一模拟中该检验的波形实例。

标记 1 显示了目标设备断言“stopn”信号时的时钟沿。标记显示了总线变为空闲的时刻点。注意，信号“reqn”在该时钟周期和前一时钟周期都被取消断言。因此，检验器 a_mchk7_before 成功。

Master_chk8: 当目标设备通过重试(retry)命令来结束一次事务时，主控必须重复该事务直到它结束。

```

sequence s_mchk8a(temp1);
@ (posedge clk)
(((!gntn || $rose (gntn))
&& $fell (framen)), temp1=cxben[3:0])
##[1:2] $fell (irdyn) ##[0:5] $fell (stopn)
&& $fell (devseln) && trdyn;
endsequence

sequence s_mchk8b(temp2);
@ (posedge clk)
$fell (reqn) ##[0:100] !gntn
##[0:5] $fell (framen)
##0 ((cxben[3:0] == temp2));
endsequence

property p_mchk8;
int temp;
@ (posedge clk)
s_mchk8a (temp) |->
##[2:20] s_mchk8b (temp);
endproperty

a_mchk8: assert property(p_mchk8);
c_mchk8: cover property(p_mchk8);

```

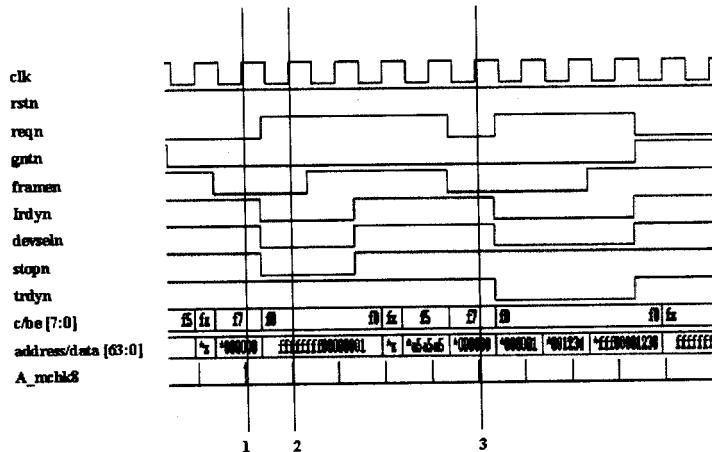


图 6-11 PCI 主控检验 8

我们用两个单独的序列来检验该属性。第一个序列当主控设备断言“framen”信号时开始。当主控断言该帧时，它同时也发

出一个命令。我们用一个临时变量“temp1”来存储主控发出的该命令。该变量应在信号“framen”下降沿匹配时更新值。在之后的几个周期内，如果目标设备通过断言“stopn”来终止事务，那么序列 s_mchk8a 将匹配。属性 p_mchk8 有属性 s_mchk8a 作为先行算子。如果该先行算子为真，那么我们等待主控发出下一个命令。如果主控发出新命令，我们比较本地变量“temp”的值和总线上主控实际发出的命令，从而验证两个命令相同。如果两个命令不同，则是一次违规。图 6-11 显示了一次模拟中的波形图实例。

标记 1 显示了信号“framen”的一个下降沿时间点。此时命令总线上被送入一个“f7”命令。标记 2 显示了目标设备通过断言“stopn”信号来停止事务。主控发出另一个请求得到总线的控制权。标记 3 显示了主控再次断言“framen”信号的时间点。此时命令总线上再次被送入一个“f7”命令，因而检验成功。

Master_chk9: 总线奇偶校验位检验地址阶段的错误(SERR)，它能检验各种不同的事务：存储器读、存储器写、I/O 读、I/O 写等。

```

property p_mchk9;
@(posedge clk)
    $fell (framen) ##1
        (par ^ $past (^ (ad[31:0]^cxben[3:0])) == 1) |-
            ##[1:5] $fell (serrn);
endproperty

a_mchk9: assert property(p_mchk9);
c_mchk9: cover property(p_mchk9);

```

奇偶校验检验是在每次事务的地址阶段进行的。信号“par”、向量“ad”和“c/be”的奇偶校验应该总为偶。这可以通过异或所有信号为来实现。奇偶校验错误通常在下一个时钟周期触发。如果在事务处理中出现奇偶校验错误，那么这表示有系统错误，则应该断言信号“serrn”。图 6-12 显示了模拟中该检验的波形图实例。

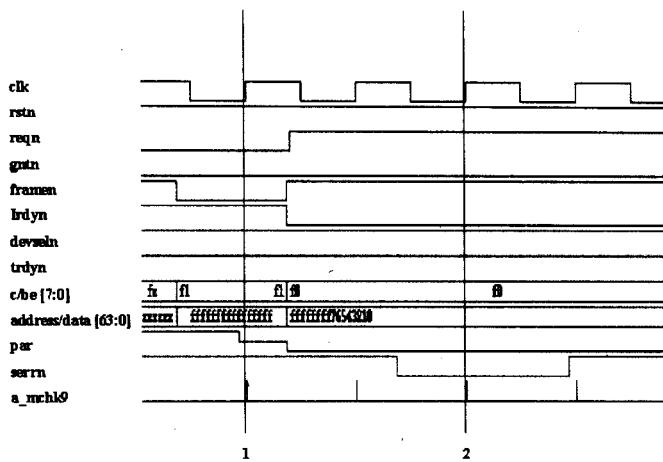


图 6-12 PCI 主控检验 9

标记 1 显示了地址阶段的一个取样点。总线“c/be”、总线“address”和信号“par”的值在该时刻被取样并被异或起来看是否存在偶校验。标记 2 显示偶校验不存在因而断言信号“serrn”。注意，信号“serrn”持续两个时钟周期保持被断言状态。

Master_chk10: 数据阶段的奇偶校验错误(PERR)。

奇偶校验位检查在每次事务的数据阶段进行。信号“par”、向量“ad”和“c/be”的校验应该总为偶。这可以通过异或所有信号位来实现。奇偶校验错误通常在下一个时钟周期被触发。如果事务数据阶段出现奇偶校验错误，那么信号“perrn”因该被断言。图 6-13 显示了模拟中该检验的波形图实例。

```

property p_mchk10;
@(posedge clk)
(!irdyn && !trdyn) ##1
(par ^ $past (^ad[31:0]^cxben[3:0])) == 1) |->
##[1:5] !perrn;
endproperty

a_mchk10: assert property(p_mchk10);
c_mchk10: cover property(p_mchk10);

```

标记 1 显示这个特定事务中多次数据阶段中第一次出现数据

阶段的时间点。在下一个时钟周期，所需的奇偶校验位被置值。其值为前一个时钟周期的数据、命令字节使能取样值的异或结果。如果奇偶校验位错误，信号“perrn”应被断言。对标记 1 来说，数据和命令的异或结果为 1，所以没有校验错误。标记 2 显示的是第二个数据阶段。数据(1234)和命令(0000)的异或为 1，奇偶校验位被置 0。这不符合一个偶校验，所以在下一个时钟周期断言信号“perrn”，正如标记 3 所示。

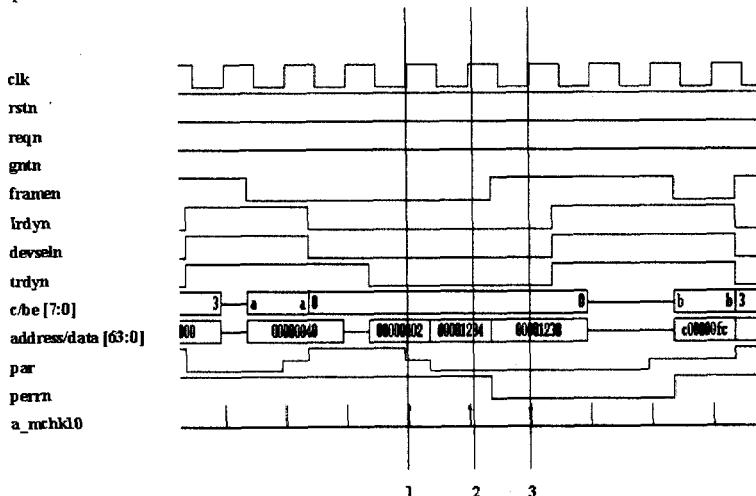


图 6-13 PCI 主控检验 10

Master_chk11: PERR 在特殊周期中不应被断言。

主控可以为特殊周期发出命令。命令总线在特殊周期内的值为“0001”。特殊周期内无论数据总线上数据为何值，奇偶校验错误不被断言。

```

property p_mchk11;
  @(posedge clk)
    ($fell (framen) && (cxben[3:0] == (4'b0001))) |->
      (perrn [*1:$]
        ##0 ($rose (irdyn && trdyn))
        ##1 perrn[*2]);
  endproperty

  a_mchk11: assert property(p_mchk11);
  c_mchk11: cover property(p_mchk11);

```

如果主控在特殊周期内断言“framen”信号，那么“perrn”信号在总线空闲之前不能被断言。注意，我们需要确保即使取消断言“trdyn”和“irdyn”信号，在两个时钟周期内都不会断言信号“perrn”。由于奇偶校验是在数据阶段的下一个周期被触发并通常持续两个周期，所以这个对检验器的额外条件是必要的。图 6-14 显示了模拟中波形图实例。标记 1 显示的是主控断言“framen”信号并在“c/be”总线上发出命令“0001”来标示一个特殊周期。注意，此时无论“par”的值为何，暗示奇偶校验错误的信号“perrn”始终保持被取消断言状态。因此，检验器成功。标记 2 显示了一个类似的特殊周期。

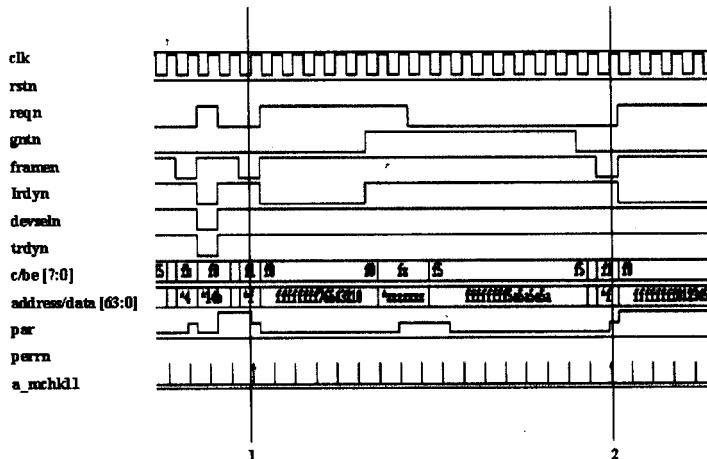


图 6-14 PCI 主控检验 11

Master_Chk12: 双重地址周期。

对于 64 位地址的目标设备，主控需要用两个时钟周期来断言“irdyn”信号。当主控断言“framen”时，它同时为该双重地址周期发出命令并断言“req64n”来告知目标设备它希望进行 64 位事务。

```

property p_mchk12;
  @ (posedge clk)
    `s_DUAL_ADDR_CYCLE && req64n =>
      not $fell (irdyn);
endproperty

```

```
a_mchk12: assert property(p_mchk12);
c_mchk12: cover property(p_mchk12);
```

Master_chk13: 64 位全程事务。

主控设备断言信号“req64n”和“framen”来告知目标设备它要进行 64 位事务。作为应答，目标设备在 1~5 个时钟周期内断言“devseln”和“ack64n”信号。

```
property p_mchk13;
@(posedge clk)
$fell (gntn) ##[1:8]
$fell (framen) && $fell(req64n) |->
##[1:5] $fell (ack64n) && $fell(devseln);
endproperty

a_mchk13: assert property(p_mchk13);
c_mchk13: cover property(p_mchk13);
```

图 6-15 显示了模拟中波形图实例。标记 1 显示了信号“gntn”被断言的时刻点。在下一个时钟周期，主控在记号 2 处断言“framen”和“req64n”信号。这个过程就通知了目标设备主控要进行 64 位处理的意图。而目标设备通过在下一周期断言“ack64n”和“devseln”来应答。主控在下一周期断言“irdyn”信号。注意，在 64 位事务中，主控在断言“framen”信号后需要两个周期来断言“irdyn”信号。

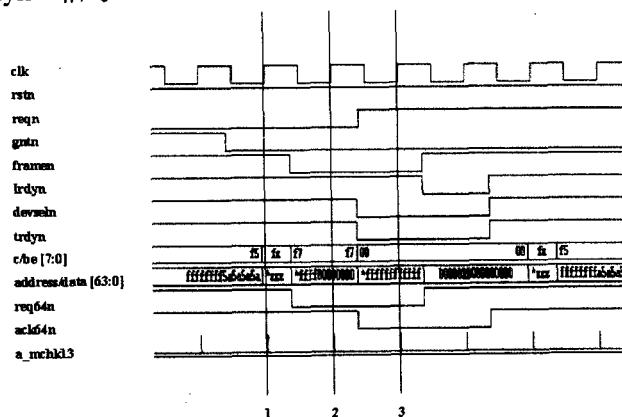


图 6-15 PCI 主控检验 13

Master_Chk14: 检验 par64 信号的有效性。

和 32 位事务类似，64 位事务中也使用一个奇偶校验位。数据总线高 32 位和命令字节使能的高 4 位的位异或值通过“par64”进行偶检验。

```

property p_mchk14;
  @(posedge clk)
    (!ack64n && !irdyn && !trdyn && !devseln) &&
    (^(ad[63:32]^cxben[7:4]) == 1) |=>
      par64;
  endproperty

  a_mchk14: assert property(p_mchk14);
  c_mchk14: cover property(p_mchk14);

```

图 6-16 显示了模拟中波形图实例。标记 1 显示一个有效 64 位数据阶段发生的时刻。数据(00000200)的高 32 位和“c/be”(0000)的高 4 位异或值为 1。因此，为了维持偶校验，在下一个时钟周期内信号“par64”的值应为 1。从标记 2 看到，“par64”的值为 1，因而检验成功。

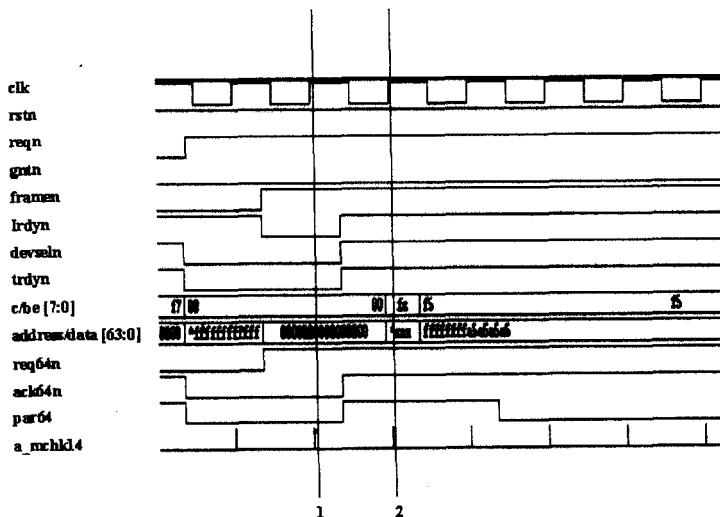


图 6-16 PCI 主控检验 14

Master_chk15: 总线停泊。

总线停泊发生在主控的“reqn”信号被取消断言但仍可访问总线。主控进入空闲状态后驱动稳定的值到数据总线和命令总线来表示它已将总线停泊。当主控进入该空闲状态，“reqn”不应被断言。如果断言了“reqn”，那么可将其认为是一个“背对背事务”(back to back transaction)。

```

sequence s_mchk15;
  @ (posedge clk)
  first_match ($fell (framen) ##[1:$]
    (framen && irdyn && !gntn && reqn));
endsequence

property p_mchk15;
  @ (posedge clk)
    s_mchk15 |->
      ##[1:8] (($stable (ad[31:0]))
      && ($stable (cxben[3:0])))
      ##1 (par ^ $past (^ (ad[31:0]^cxben[3:0])) == 0);
endproperty

a_mchk15: assert property (p_mchk15);
c_mchk15: cover property (p_mchk15);

```

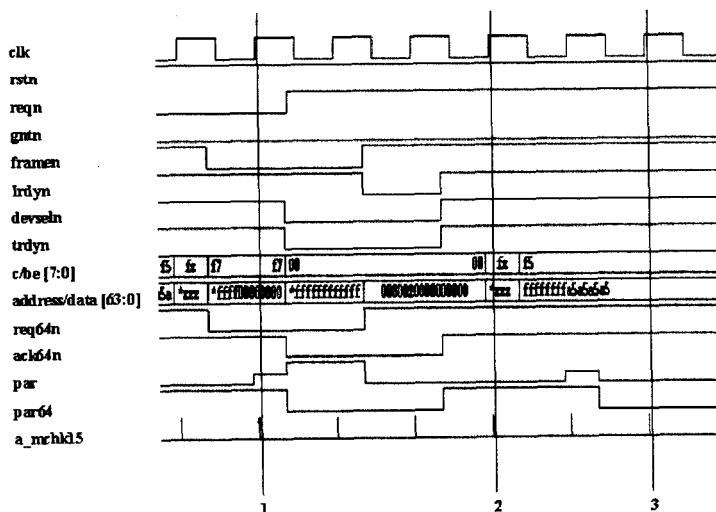


图 6-17 PCI 主控检验 15

我们写一个简单的序列 s_mchk15 来识别一个主控事务的有效完成。在事务完成时，如果主控仍占用总线，那么它会向数据总线和命令总线送入稳定的值，因而停泊 PCI 总线。\$stable 函数就用来监测总线上的值是否达到稳定。我们预计达到稳定的一个周期之后，奇偶校验位会被设为相应的值。这里我们使用异或的方法来检测校验位是否有效。

图 6-17 显示了模拟中该检验的波形图。标记 1 为事务开始时刻，标记 2 为事务结束点。注意在该点信号“reqn”被取消断言但信号“gntn”被断言。因此我们期望主控在 1~8 个周期内向数据总线和命令总线送入稳定值。在标记 3 的时间点主控已停泊了主线。

Master_chk16：快速背靠背事务。

事务完成后紧接的一个时钟周期内，主控可以通过断言“framen”信号来进行快速背靠背事务。这可以发生在单个或者多个数据阶段完成之后。属性 p_mchk16 能捕捉单个数据阶段事务，而属性 p_mchk17 能捕捉到单个和多个数据阶段事务。

```
property p_mchk16;
@(posedge clk)
($rose (framen) && $fell (irdyn))
##1 $fell (framen) |->
                     $rose (irdyn);
endproperty

a_mchk16: assert property(p_mchk16);
c_mchk16: cover property(p_mchk16);

property p_mchk17;
@(posedge clk)
(!irdyn && framen)
##1 $fell (framen) |->
                     $rose (irdyn);
endproperty

a_mchk17: assert property(p_mchk17);
c_mchk17: cover property(p_mchk17);
```

注意，两个属性的主要区别在于取样机制。如果是单一数据

阶段背对背事务，信号“framen”和“irdyn”在它们上升或下降的边沿被取样(“framen”的下降沿和“irdyn”的上升沿)。图 6-18 所示的是模拟中该检验的波形图。

标记 1 和 2 示意了单一数据阶段背靠背事务。标记 3 和 4 示意了多次数据阶段背靠背事务。注意，属性 p_mchk17 足以捕捉到这两种情形。

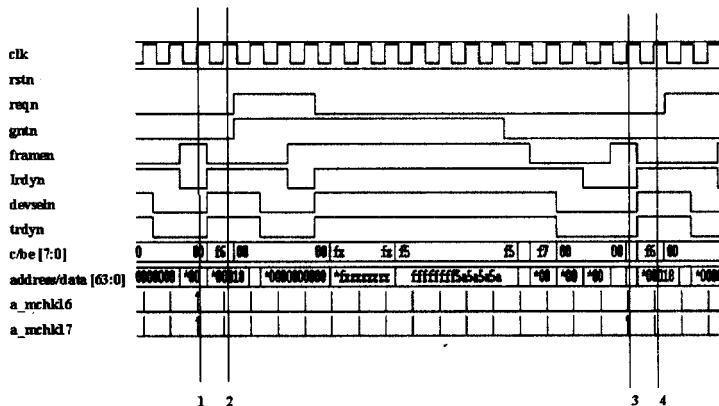


图 6-18 PCI 主控检验 16/17

PCI 主控功能覆盖点的实例

主控中止 – 主控中止可以在下面情形下发生: I/O 读、I/O 写、配置读、配置写、存储器读、存储器写。我们可以用一句覆盖陈述来保证测试模型对所有可能的中止状态至少执行一次。当主控断言“irdyn”后的 5 个周期内，目标设备没有通过断言“devseln”信号来响应，主控将取消断言“irdyn”信号来中止事务。注意，我们在属性中用`define 代码来指定命令。

```

property p_mcov1;
  @ (posedge clk)
    `s_IO_READ ##1 (devseln)[*5] |=>
      $rose (irdyn);
  endproperty

property p_mcov2;
  @ (posedge clk)
    `s_IO_WRITE ##1 (devseln)[*5] |=>
      $rose (irdyn);

```

```
endproperty

property p_mcov3;
@(posedge clk)
`S_MEMORY_READ ##1 (devseln)[*5] |=>
$rose (irdyn);
endproperty

property p_mcov4;
@(posedge clk)
`S_MEMORY_WRITE ##1 (devseln)[*5] |=>
$rose (irdyn);
endproperty

property p_mcov5;
@(posedge clk)
`S_CONFIG_READ ##1 (devseln)[*5] |=>
$rose (irdyn);
endproperty

property p_mcov6;
@(posedge clk)
`S_CONFIG_WRITE ##1 (devseln)[*5] |=>
$rose (irdyn);
endproperty

c_mcov1: cover property(p_mcov1);
c_mcov2: cover property(p_mcov2);
c_mcov3: cover property(p_mcov3);
c_mcov4: cover property(p_mcov4);
c_mcov5: cover property(p_mcov5);
c_mcov6: cover property(p_mcov6);
```

6.4 情形 2——目标 DUT 设备

在本节中，我们假定被测设计是一个 PCI 目标设备，而系统其他部分不变。图 6-19 显示了系统实例。

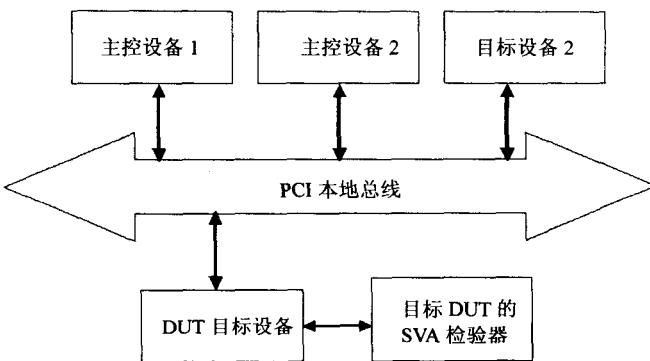


图 6-19 PCI 目标设备作为 DUT 的配置实例

PCI 目标断言

Target_chk1: 一旦目标设备断言信号“stopn”，它将在信号“framen”被取消断言之前保持“stopn”的值。而当“framen”被取消断言后的一个时钟周期后，“stopn”被取消断言。

```

property p_tchk1;
@(posedge clk)
($fell (stopn) && !framen) |->
!stopn [*1:$]
##0 $rose (framen) ##1 $rose(stopn);
endproperty

a_tchk1: assert property(p_tchk1);
c_tchk1: cover property(p_tchk1);

```

注意，该属性使用“repeat until”的结构来确保信号“stopn”直到取消断言“framen”前一直保持被断言状态。图 6-20 显示了模拟中该检验的波形图实例。

标记 1 为信号“stopn”被断言的时间点。在下一个周期内，信号“framen”被取消断言，而在一个周期之后信号“stopn”也被取消断言，如标记 2 所示。

Target_chk2: 一旦目标断言了“trdyn”信号，它在当前数据阶段结束之前不能改变“devseln”和“trdyn”。

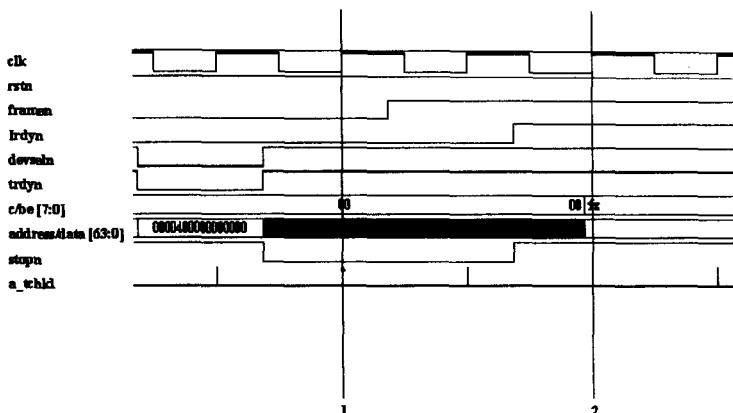


图 6-20 PCI 目标检验 1

当目标断言信号“`trdyn`”，它就确认了其进入接收或发射数据的状态。因此，它不能在未完整一个数据阶段时就取消断言信号“`trdyn`”。

```

property p_tchk2;
@(posedge clk)
$fell (trdyn) |->
    (!trdyn && !devseln) [*0:16] ##0 !irdyn;
endproperty

a_tchk2: assert property(p_tchk2);
c_tchk2: cover property(p_tchk2);

```

属性 `p_tchk2` 在信号“`trdyn`”下降沿时刻激活。该属性确保在断言“`irdyn`”之前“`trdyn`”和“`devseln`”保持被断言的状态。信号“`trdyn`”的延时是 16 个周期。

Target_chk3: 目标设备在断言“`devseln`”之前不能断言“`trdyn`”信号。

```

property p_tchk3;
@(posedge clk)
$fell (trdyn) |->!devseln;
endproperty

a_tchk3: assert property(p_tchk3);
c_tchk3: cover property(p_tchk3);

```

Target_chk5: 与数据断连。

目标设备可以通过同时断言“stopn”和“trdyn”信号来示意它不能继续进行事务。当这种情况发生时，目标设备需要在下一个时钟周期取消断言“trdyn”并保持断言“stopn”。因而，最后的数据阶段没有传输任何数据的情况下就终止了(由于“trdyn”被取消断言)。在 PCI 局部总线规范中，上面情况归类为“断连-B(Disconnect-B)”。

```
property p_tchk5b;
@(posedge clk)
  ($fell (stopn) && !framen && !trdyn && !irdyn)
    |=>
      (framen && trdyn)
      ##1 (stopn && devseln && irdyn);
  endproperty

  a_tchk5b: assert property(p_tchk5b);
  c_tchk5b: cover property(p_tchk5b);
```

图 6-21 显示了模拟中该检验的一个波形图实例。标记 1 为信号“stopn”被断言的时间点。下一个时钟周期内，信号“trdyn”被取消断言。标记 2 显示，在一个时钟周期后，信号“stopn”、“devseln”和“irdyn”全被取消断言，因而事务完成。

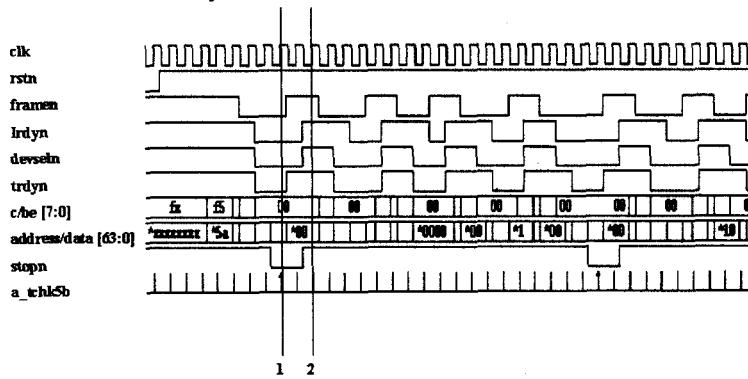


图 6-21 PCI 目标检验 5b

Target_chk6: 无数据终止的断连。

当目标设备不能完成更多的数据阶段，他断言信号“stopn”

并取消断言“trdyn”。直到最后的数据阶段完成，目标设备一直保持信号“stopn”被断言。

注意，像这样复杂的属性应该分成如下较小的序列。

```
sequence s_tchk6a;
@(posedge clk)
(!irdyn && !trdyn && !devseln && !framen);
endsequence

sequence s_tchk6b;
@(posedge clk)
($fell (stopn) && $rose (trdyn) && !framen);
endsequence

sequence s_tchk6c;
@(posedge clk)
$rose (framen) ##[0:8] (!irdyn && !stopn);
endsequence

sequence s_tchk6;
@(posedge clk)
s_tchk6a.ended ##[1:8] s_tchk6b;
endsequence

property p_tchk6;
@(posedge clk)
s_tchk6.ended |=> s_tchk6c;
endproperty

a_tchk6: assert property(p_tchk6);
c_tchk6: cover property(p_tchk6);
```

序列 s_tchk6a 用来识别一个有效的数据阶段。序列 s_tchk6b 识别目标设备断言“stopn”信号的时间点。序列 s_tchk6 是 s_tchk6a 和 s_tchk6b 的连接序列。序列 s_tchk6 在“stopn”发出时进行检验。序列 s_tchk6c 用于寻找信号“irdyn”和“stopn”同时被断言的时刻。我们之所以需要这样做是因为主控在最后数据阶段完成前可能进入等待状态而取消断言信号“irdyn”。

Target_chk6_1: 主控自然终止并且目标设备同时发出中止信号。

这种情况发生在目标设备发出“stopn”信号中止事务的同时主控也正在自然终止事务。这意味着在目标断言“stopn”的同一时钟周期，主控取消断言“framen”信号。

```

sequence s_tchk6_1;
@(posedge clk)
  (!irdyn && !trdyn && !devseln && !framen)
  ##[1:8] ($fell (stopn) && trdyn && framen);
endsequence

property p_tchk6_1;
@(posedge clk)
  s_tchk6_1.ended |=> (irdyn && stopn);
endproperty

a_tchk6_1: assert property(p_tchk6_1);
c_tchk6_1: cover property(p_tchk6_1);

```

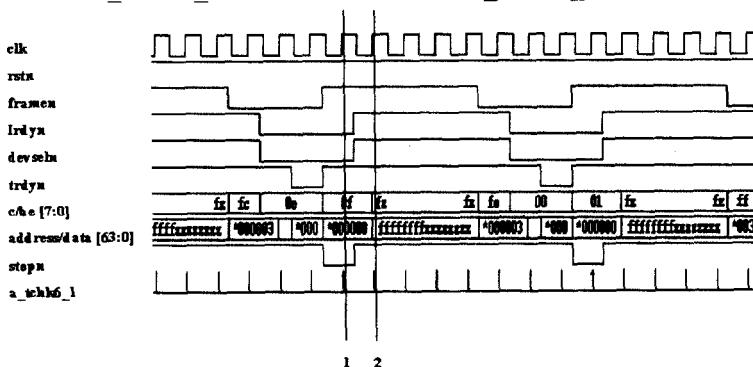


图 6-22 PCI 目标检验 6_1

注意，我们写了一个序列来识别目标和主控同时停止事务。该属性检验如果先行算子匹配，信号“irdyn”则应该在下一个时钟周期和信号“stopn”都被断言。图 6-22 显示了一个模拟中检验的实例。标记 1 显示了信号“framen”被取消断言和信号“stopn”被断言。标记 2 显示了信号“irdyn”和“stopn”被取消断言。

Target_chk7: 重试

如果目标未预备进行事务，它必须在第一个数据阶段发生之前通知主控在以后的时间点再进行事务。也就是目标设备在第一次断言“trdyn”之前就断言信号“stopn”。

```

sequence s_tchk7a;
@(posedge clk)
$fell (framen) ##[1:8] $fell(irdyn);
endsequence

sequence s_tchk7b;
@(posedge clk)
$fell (framen) ##[1:5]
$fell(devseln) && $fell(stopn) && trdyn;
endsequence

sequence s_tchk7;
@(posedge clk)
first_match(s_tchk7a and s_tchk7b);
endsequence

property p_tchk7;
@(posedge clk) s_tchk7.ended |=> framen;
endproperty

a_tchk7: assert property(p_tchk7);
c_tchk7: cover property(p_tchk7);

```

当信号“framn”被断言时序列 s_tchk7 被激活。一旦信号“framen”被断言，我们期望目标设备通过断言“devseln”信号来识别自己。依据目标设备速度的不同，这可能在 1~5 个时钟周期内发生。如果目标设备需要发出重试请求，它需要同时断言“stopn”和“devseln”。此时，信号“trdyn”应该保持取消断言状态。在重试发出后一个周期，主控取消断言“framen”来作为确认。

图 6-23 显示了一个模拟中该检验的实例。标记 1 显示了主控断言“framen”信号的时间点。标记 2 显示了目标请求主控进行重试的时间点。标记 2 后的一个时间周期，主控取消断言“framen”信号并结束事务。

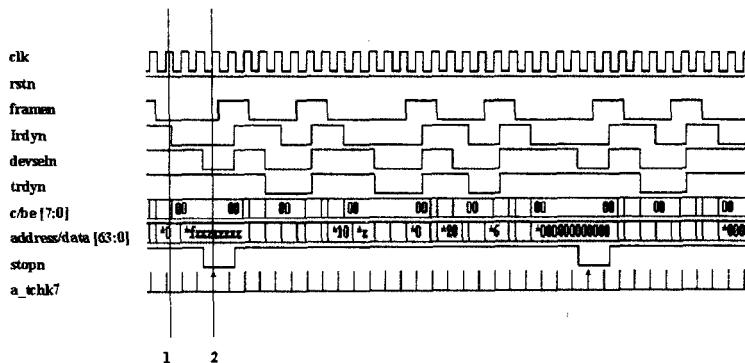


图 6-23 PCI 目标检验 7

Target_chk8: 特殊周期内信号“devseln”不应被断言。

```

property p_tchk8;
  @(posedge clk)
  $fell (framen) && (cben[3:0] == 4'b0001) |->
    devseln [*1:$] ##0 $rose (framen);
endproperty

a_tchk8: assert property(p_tchk8);
c_tchk8: cover property(p_tchk8);

```

在特殊周期内，不应断言信号“devseln”。当“framen”信号被断言并且主控在命令总线上置放一个特殊周期信号时，该属性被激活。该属性确保主控在通过取消断言“framen”信号来完成事务之前，信号“devseln”保持被取消断言状态。

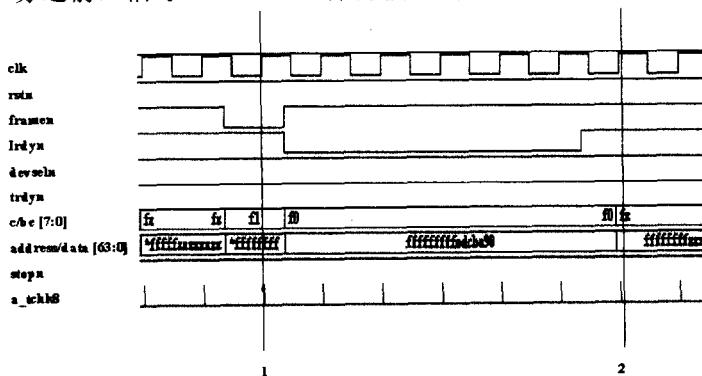


图 6-24 PCI 目标检验 8

图 6-24 显示了一个模拟中该检验的实例。标记 1 为特殊周期命令被探测到的时间点。标记 2 为事务完成点。在标记 1 到标记 2 之间，信号“devseln”保持被取消断言状态。

Target_chk9: 信号“framen”被断言直到第一个数据阶段完成的延迟为 16 个周期。

一旦主控断言信号“framen”，目标设备通过首先断言“devseln”信号来识别自己。根据目标设备的性质不同，这可能在信号“devseln”被断言后 1~5 周期之内的任何时刻发生。譬如，一个高速目标设备只需要一个周期，而中速目标设备需要两个周期。在断言“devseln”信号之后，如果目标设备预备好进行事务，它将断言“trdyn”信号。PCI 局部总线规范所允许的总时延(从主控断言“framen”信号，到数据阶段发生，也就是“trdyn”和“irdyn”都被断言)是 16 个时钟周期。表 6-2 显示对不同性质的目标设备该时延的划分。

我们写两个基本序列来识别一个有效的数据阶段或者一个重试状态。对于每种目标设备类型我们分别定义序列。注意“devseln”被断言的时延是这些序列的惟一差别。

表 6-2 各种目标设备的时延划分

设备类型	Frame -> devsel	Devsel -> (irdy && trdy)
高速(FAST)	1	0:15
中速(MEDIUM)	2	0:14
低速(SLOW)	3	0:13
次低速(SUBTRACTIVE)	4	0:12

```

sequence s_tchk9a;
@ (posedge clk)
(!irdyn && !trdyn);
endsequence

sequence s_tchk9b;
@ (posedge clk)
(!irdyn && !stopn);
endsequence

```

```
sequence s_tchk9_fast;
@(posedge clk)
$fell (framen) ##1 $fell(devseln);
endsequence

sequence s_tchk9_medium;
@(posedge clk)
$fell (framen) ##2 $fell(devseln);
endsequence

sequence s_tchk9_slow;
@(posedge clk)
$fell (framen) ##3 $fell(devseln);
endsequence

sequence s_tchk9_subtractive;
@(posedge clk)
$fell (framen) ##4 $fell(devseln);
endsequence

property p_tchk9_fast;
@(posedge clk)
s_tchk9_fast |-> ##[0:15]
(!devseln) throughout
(s_tchk9a.ended || s_tchk9b.ended);
endproperty

a_tchk9_fast: assert property(p_tchk9_fast);
c_tchk9_fast: cover property(p_tchk9_fast);

property p_tchk9_medium;
@(posedge clk)
s_tchk9_medium |-> ##[0:14]
(!devseln) throughout
(s_tchk9a.ended || s_tchk9b.ended);
endproperty

a_tchk9_medium: assert property(p_tchk9_medium);
c_tchk9_medium: cover property(p_tchk9_medium);

property p_tchk9_slow;
@(posedge clk)
s_tchk9_slow |-> ##[0:13]
(!devseln) throughout
(s_tchk9a.ended || s_tchk9b.ended);
```

```
endproperty
```

```
a_tchk9_slow: assert property(p_tchk9_slow);
c_tchk9_slow: cover property(p_tchk9_slow);
```

```
property p_tchk9_subtractive;
@(posedge clk)
s_tchk9_subtractive |-> ##[0:12]
(!devseln) throughout
(s_tchk9a.ended || s_tchk9b.ended);
endproperty
```

```
a_tchk9_subtractive:
assert property(p_tchk9_subtractive);
c_tchk9_subtractive:
cover property(p_tchk9_subtractive);
```

对每种设备类型，我们分别编写了一个属性。如果在属性先行算子中的序列识别了某种类型的设备，那么其后续算子将根据表 6-2 允许的周期数来匹配。譬如，对于低速类型设备，目标设备需要在 0~13 个时钟周期内完成一个有效的数据阶段或发出重试。

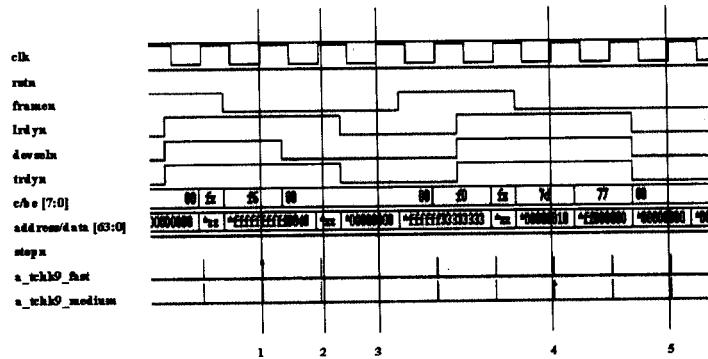


图 6-25 PCI 目标检验 9

图 6-25 显示了一个模拟中该检验的实例。标记 1 显示了属性 p_tchk9_fast 的开端。主控此时断言信号“framen”。一个周期后，在标记 2 处目标设备断言信号“devseln”。在一个周期后，在标

记 3 处断言信号“trdyn”和“irdyn”，因而 check a_tchk9_fast 检验成功。

标记 4 显示了属性 p_tchk9_medium 的开端。主控此时断言信号“framen”。两个周期后，在标记 5 处目标设备断言信号“devseln”。信号“trdyn”和“irdyn”在同一周期内被断言因而 check a_tchk9_medium 检验成功。

Target_chk10: 从前一个数据阶段到后一个数据阶段的时延为 8 个周期。

如果未就绪，主控和目标都可以在事务中发出等待状态。如果在某个给定的时钟边沿，一个数据阶段刚完成一个突发事务(burst transaction)，那么下一个数据阶段应该在 8 个时钟周期内发生。

```
property p_tchk10;
  @ (posedge clk)
    (!irdyn && !trdyn && !devseln && !framen) |>
      ##[1:8] (!irdyn && (!trdyn || !stopn));
  endproperty

  a_tchk10: assert property(p_tchk10);
  c_tchk10: cover property(p_tchk10);
```

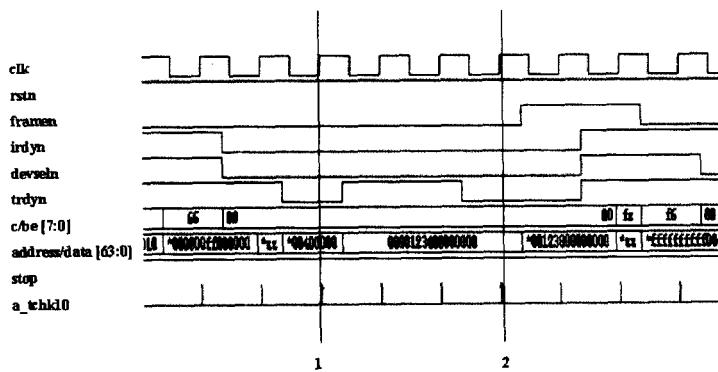


图 6-26 PCI 目标检验器 10

图 6-26 显示了一个模拟中该检验的实例。标记 1 显示了一个有效数据阶段。在下一个时钟周期内，目标设备取消断言“trdyn”信号来发出一个等待状态。该等待状态延续一个周期。一个周期

之后，如标记 2 所示，再次断言信号“trdyn”，因而有效数据阶段接着发生。这种情况下，随后的这个数据阶段发生在 3 个周期内，所以验证成功。

Target_chk11: 读命令引发的第一个数据阶段需要由信号“trdyn”所实施的一个周转周期(turnaround cycle)。

如表 6-1 所示有 4 种可能的读命令，而每个读命令在最低位为“10”。每当有一个读命令，主控必须允许目标设备驱动数据进入总线，因而其间有一个周转周期。在读命令引发的第一个数据阶段的前一个周期，数据总线的数据值应该是未知的。

```
sequence s_tchk11a;
@ (posedge clk)
  ($fell (framen) && (cxben[1:0] == 2'b10));
endsequence

sequence s_tchk11b;
@ (posedge clk)
  first_match($fell (devseln) ##[1:16]
              $fell (trdyn));
endsequence

sequence s_tchk11;
@ (posedge clk)
  s_tchk11a.ended ##[1:5] s_tchk11b;
endsequence

property p_tchk11;
@ (posedge clk)
  s_tchk11.ended |->
    ($isunknown (par)
    && $past ($isunknown(ad[31:0])));
endproperty

a_tchk11: assert property(p_tchk11);
c_tchk11: cover property(p_tchk11);
```

序列 s_tchk11a 探测读命令。序列 s_tchk11b 探测读命令发出后第一个有效数据阶段。属性 p_tchk11 等待第一个数据阶段完成，然后使用 \$past 结构来检查前一个周期的奇偶校验位和数据总线。如果前一个周期值不为“z”，那么它是一个违背。

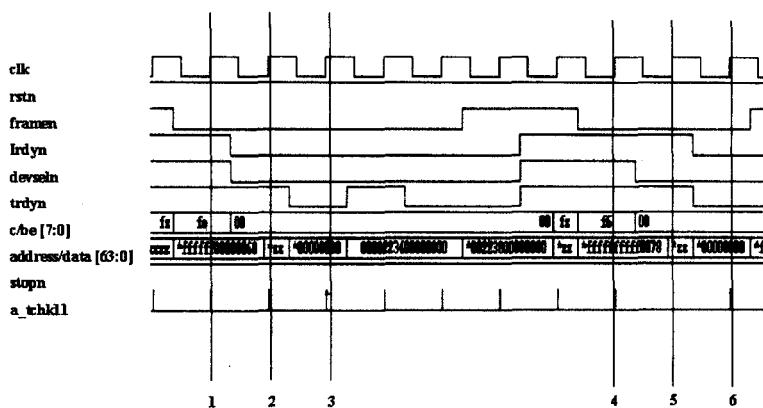


图 6-27 PCI 目标检验 11

图 6-27 显示了一个模拟中该检验的实例。标记 1 为读信号被探测到的时间点(1110 – 储存器读命令)。标记 3 为第一个有效数据阶段发生点。在此一个周期之前，数据线应该为“z”。标记 2 显示数据线的值未知因而检验成功。

标记 4 为读信号被探测到的时间点(0110 – 储存器读命令)。标记 6 为第一个有效数据阶段发生点。在此之前的一个周期，数据线应该为“z”。标记 5 显示数据线的值未知因而检验成功。

Target_chk12: 配置周期 (1)

在有效的配置周期内，地址总线的最低两位被设为“00”或者“01”。当配置命令发出，芯片选择信号“idsel”被断言。目标设备必须断言“devseln”作为应答，而最终当“trdy”信号被断言时配置完成。

```

sequence s_tchk12a;
@ (posedge clk)
(`s_CONFIG_READ || `s_CONFIG_WRITE) &&
((ad[1:0] == 2'b00) || (ad[1:0] == 2'b01)) &&
idsel;
endsequence

sequence s_tchk12b;
@ (posedge clk)
!devseln && stopn;
endsequence

```

```

sequence s_tchk12;
@ (posedge clk)
s_tchk12a ##[1:5] s_tchk12b;
endsequence

property p_tchk12;
@ (posedge clk)
first_match(s_tchk12) |->
##[0:5] $fell (trdyn);
endproperty

a_tchk12: assert property(p_tchk12);
c_tchk12: cover property(p_tchk12);

```

图 6-28 显示了一个模拟中该检验的实例。标记 1 处系统发出配置命令。注意，信号“idsel”被断言。标记 2 处目标设备断言信号“trdyn”。

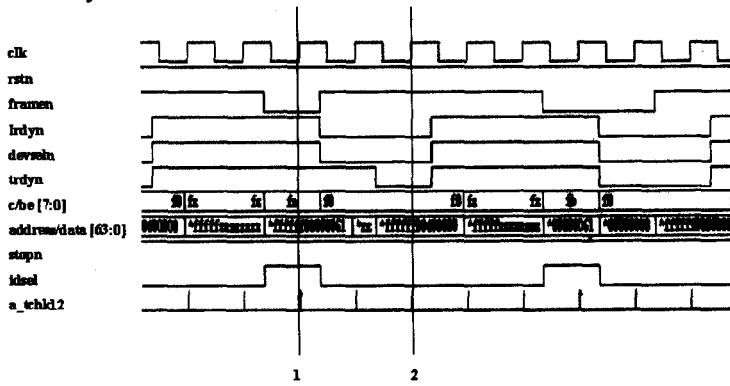


图 6-28 PCI 目标检验 12

Target_chk13: 配置周期 (2)

如果配置命令发出但地址位不正确(“10”或“11”),那么主控应该取消断言“framen”信号予以中止。

```

sequence s_tchk13a;
@ (posedge clk)
(`s_CONFIG_READ || `s_CONFIG_WRITE)
&& ((ad[1:0] == 2'b10) || (ad[1:0] == 2'b11))
&& idsel;
endsequence
sequence s_tchk13b;

```

```

@ (posedge clk)
  (devseln && stopn && trdyn) throughout
    (##[1:5] $rose (framen));
endsequence

property p_tchk13;
  @ (posedge clk)
    s_tchk13a |-> s_tchk13b;
endproperty

a_tchk13: assert property(p_tchk13);
c_tchk13: cover property(p_tchk13);

```

序列 `s_tchk13a` 探测一个无效的配置命令。序列 `s_tchk13b` 保证信号“`devseln`”、“`trdyn`”和“`stopn`”在信号“`framen`”被断言前保持取消断言状态。信号“`framen`”应该在 5 个周期内被取消断言。

图 6-29 显示了一个模拟中该检验的实例。标记 1 处探测到一个无效配置命令。标记 2 处主控取消断言“`framen`”信号来中止事务。注意，信号“`trdyn`”、“`devseln`”和“`stopn`”从标记 1 到标记 2 保持取消断言。

PCI 目标设备的功能覆盖点实例：

保留命令—PCI 保留命令中，不应断言信号“`devseln`”。覆盖属性的先行算子寻找“`framen`”信号被断言时的保留命令。后续算子确保信号“`devseln`”在“`framen`”信号取消断言之前保持取消断言状态。

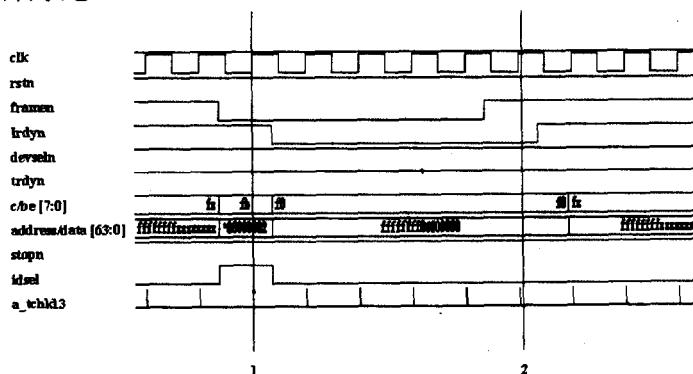


图 6-29 PCI 目标检验 13

```
property p_tcov1;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b0100) |->
    devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov1: cover property(p_tcov1);

property p_tcov2;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b0101) |->
    devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov2: cover property(p_tcov2);

property p_tcov3;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b1000) |->
    devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov3: cover property(p_tcov3);

property p_tcov4;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b1001) |->
    devseln [*1:5] ##1 $rose (framen);
endproperty

c_tcov4: cover property(p_tcov4);
```

6.5 情形 3——系统级断言

本节中，我们展示几个 PCI 仲裁器(arbiter)的检验实例。仲裁器通常是 PCI 总线的一部分。图 6-30 显示了一个系统实例。

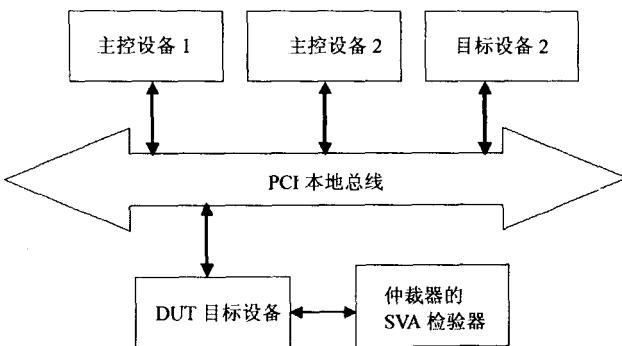


图 6-30 PCI 系统仲裁器检验实例

PCI 仲裁器断言

Arbiter_chk1: 信号“gntn”应该当“framen”断言的时候被断言。

如果信号“gntn”被取消断言，而信号“framen”在同一周期被断言，则仍为有效。

```

property p_schk1;
  @(posedge clk)
  $fell (framen) |->
    !gntn[2] || $rose (gntn[2]);
endproperty

a_schk1: assert property(p_schk1);
c_schk1: cover property(p_schk1);
  
```

Arbiter_chk2: 在一个时钟周期内，只能有一个“gntn”信号被断言。

我们的实例系统中有两个主控，因此仲裁器使用两个“gntn”信号。

```

property p_schk2;
  @(posedge clk)
  $onehot0 (!gntn[3], !gntn[2]);
endproperty

a_schk2: assert property(p_schk2);
c_schk2: cover property(p_schk2);
  
```

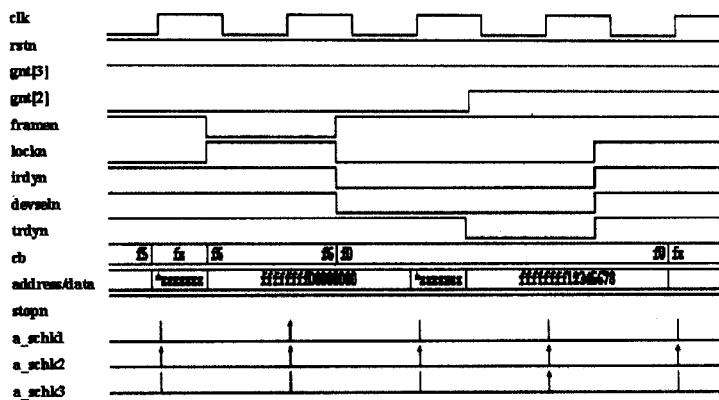


图 6-31 PCI 仲裁器检验 1, 2, 3

因为“**gntn**”信号为低电平有效信号，所以它们被置反并由**zero one-hot**结构来检验。

Arbiter_chk3: 空闲周期除外，在同一周期内不能有一个“**gntn**”信号被取消断言而另一个“**gntn**”信号被断言。

```

property p_schk3;
  @(posedge clk)
  $rose (~gntn[2]) &&
    (!framen || !irdyn) |->
      not $fell (~gntn[3]);
endproperty

a_schk3: assert property(p_schk3);
c_schk3: cover property(p_schk3);

```

图 6-31 显示了一个模拟中 **a_schk1**、**a_schk2** 和 **a_schk3** 的检验实例。

Arbiter_chk4: 信号“**lockn**”应该在整个数据阶段被断言。

```

sequence s_schk4a;
  @(posedge clk)
  first_match($fell (lockn) ##[0:5] !devseln);
endsequence

sequence s_schk4b;
  @(posedge clk)
  framen && !irdyn && (!trdyn || !stopn);

```

```

endsequence

property p_schk4;
@(posedge clk)
s_schk4a |-> !lockn [*1:$] ##0 s_schk4b;
endproperty

a_schk4: assert property(p_schk4);
c_schk4: cover property(p_schk4);

```

Arbiter_chk5: 信号“lockn”应该在地址阶段被取消断言。

```

property p_schk5;
@(posedge clk)
$fell (lockn) |->
    ($past (framen) == 0)
&& ($past(framen, 2) == 1));
endproperty

a_schk5: assert property(p_schk5);
c_schk5: cover property(p_schk5);

```

该属性的先行算子探测信号“lockn”被断言。当“framen”信号被断言时，地址阶段开始。通过检查“framen”信号的下降沿，我们能确认地址阶段的发生。我们使用\$past 算符来得到信号“framen”过去两个周期的值。

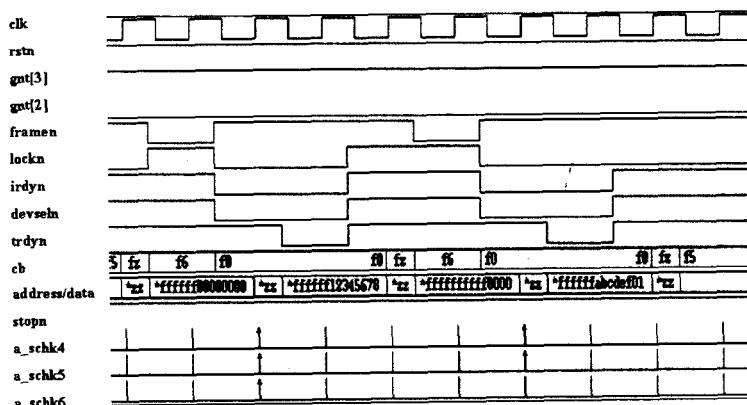


图 6-32 PCI 仲裁器检验 4, 5, 6

Arbiter_chk6: 锁定机制的第一个事务应为读操作。

```
sequence s_schk6;
@(posedge clk)
first_match($fell (gntn[2]) ##[1:8]
            $fell (framen) ##1 $fell(lockn));
endsequence

property p_schk6;
@(posedge clk)
s_schk6.ended |->
    ($past(cxben[1:01] == 2'b10);
endproperty

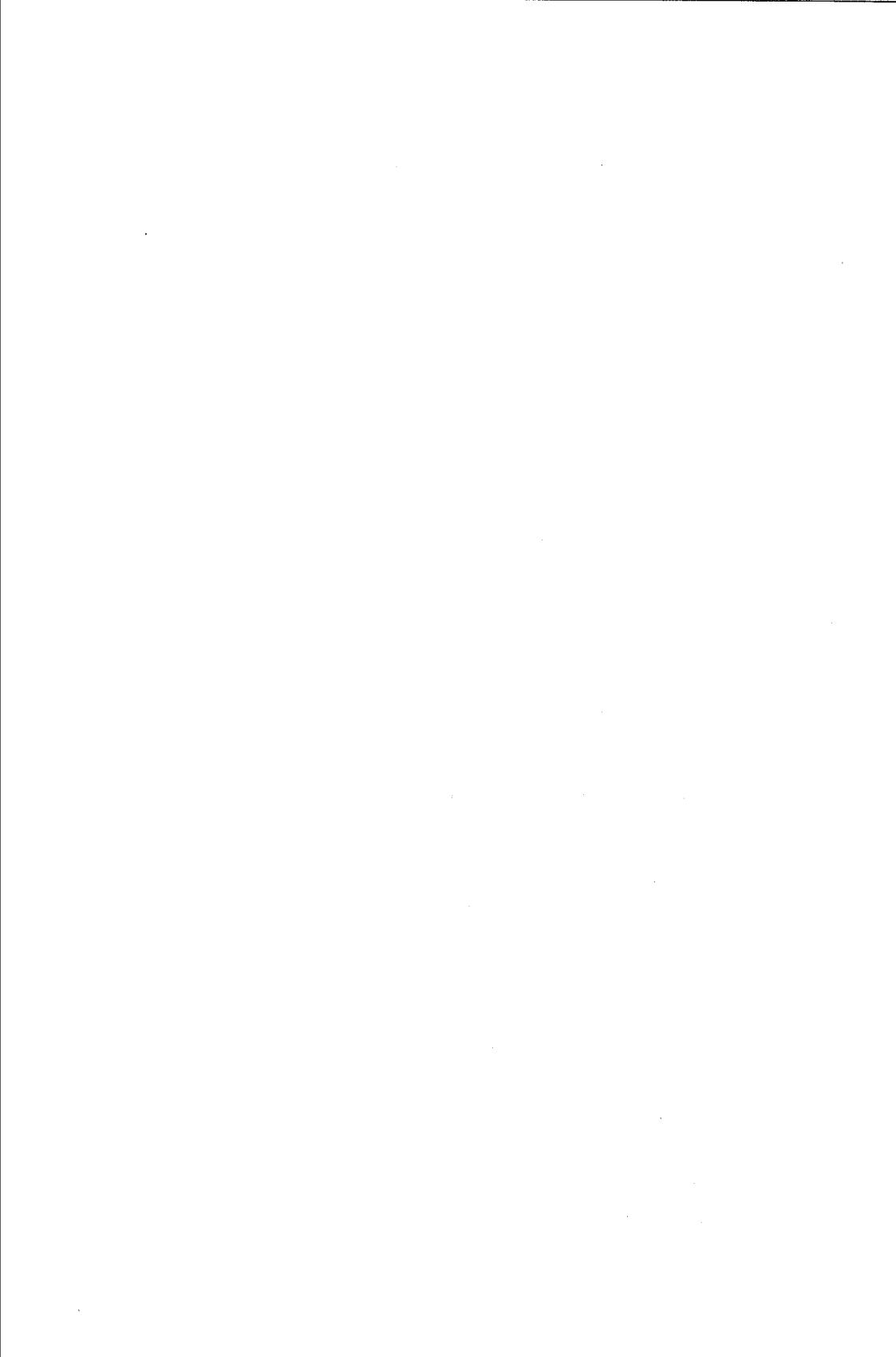
a_schk6: assert property(p_schk6);
c_schk6: cover property(p_schk6);
```

序列 s_schk6 使用“first_match”结构来识别锁定机制。其结尾作为属性的先行算子。而属性的后续算子检验命令的最低两位来确认读命令被发出。

图 6-32 显示了一个模拟中 a_schk4、a_schk5 和 a_schk6 检验的波形图实例。

6.6 用于标准协议的 SVA 小结

- 标准协议非常复杂，需要相当数目的检验器来验证其兼容性。
- 时序规则严格，所以支持这些协议的设备必须符合要求。
- 可为特定的接口开发一组通用的检验器，它们能在支持类似接口的其他设备中得到重用。
- 协议的复杂特性导致大多数属性需要许多先决条件。只有 SVA 提供了各种各样的结构和内建机制可用来定义这些复杂的先决条件。
- SVA 也提供了使用局部变量捕捉总线状态的能力。我们能够有效地使用局部变量和先决条件来编写复杂的时序检验。
- SVA 能有效地为复杂协议创建完善的功能覆盖报表。





对检验器的检验

—— 尽早隔离断言错误

基于断言的验证(ABV)所具有的强大潜力使其能够在验证过程的早期发现设计缺陷。SVA 语言具有的强大内建结构是针对 ABV 的需要而定义。SystemVerilog 3.1a 标准要求在默认情况下将断言失败显示出来，而并不要求显示断言的成功。用户可以利用断言的执行块来显示成功。由于成功的数目可能巨大(大多数断言在每个时钟沿计算)，如果将显示每一个成功定为默认情况，将可能导致巨大的日志文件而减慢模拟。

图 7-1 显示了一个典型的测试配置。它和第 0 章的图 0-2 相同。我们假设用户执行该配置，待模拟结束产生了若干断言错误。用户需要绝对确信这些错误表明的是设计错误。换言之，用户要相信他自己的断言代码正确，而生成的断言失败不是假失败。通过某一个断言来调试整个设计相当困难。如果断言失败是断言代码本身引起的，那么用户可能在验证过程中浪费大量的时间。另一方面，如果模拟中没有任何断言错误，验证工程师也必须绝对确保他的设计将工作正常，否则如果断言本身不正确，它可能捕捉不到设计意图，因而错过一个真正的设计错误。

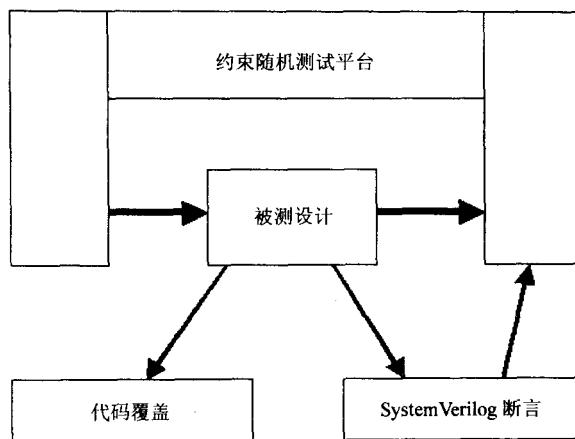


图 7-1 典型模拟配置

SVA 语言的描述特性使 SVA 验证器看起来十分简洁。如果检验写得不好，属性的真实意图可能不能被确切地表达。在将断言的验证代码融入设计中之前，验证代码的功能验证就十分重要。这需要在验证前花一些额外时间，但能够防止用户在调试中走错路。本章为如何检查检验器提供了几点建议。我们将详细讨论一个可配置 testbench 的实例。该实例检验两个信号间的断言。可配置的 testbench 背后的原理可以用来验证更加复杂的协议检验器。断言的检验确认是一个较大的课题，我们在本章中只探讨其中几个基本技巧。

7.1 断言验证

我们可以为验证断言的功能而创建一个简单的 testbench。假定属性没有使用无边界的时序，所以在大多数情况下，断言的输入条件数量有限。一个完全的 testbench 应该测试所有的输入条件。如果涉及到无边界时序，那么将不可能创建所有可能的输入条件。在无边界时序的情况下，检验器虽然能够检测出设计的错误行为，但是如果预期的事件到了模拟结束时也不发生，那么检验器就不会报错。这是因为检验器不能够假设如果模拟跑得更久一些错过

的事件也不会发生。所以，在无边界时序情况下检验器是不完全的。我们还应该注意到，即使只涉及到有限时序，所有可能的输入数目仍可能十分巨大，这取决于检验器的复杂程度。一个断言总是基于两个重要的概念，如图 7-2 所示：

- (1) 逻辑关系
- (2) 时序关系

如果根据一个 n 位的逻辑组合(与、或、异或等)表达式来写断言，那么所有可能输入条件共有 $(2^n - 1)$ 种。考虑下面的逻辑表达式：

```
signal1 && signal2 && signal3
```

该表达式将需要有 8 种可能输入条件被测试来确保正确性。

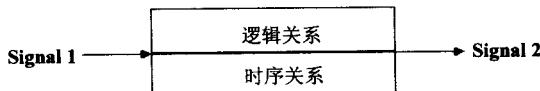


图 7-2 断言关系

我们可以使用最小和最大的时间边界来完全测试两个信号间时序的断言。譬如，考虑下面的情况：

```
signal1 ##[min:max] signal2
```

该表达式在以下两种情形时会失败：

- (1) 如果两信号的时序差小于“min”(这意味着信号 2 在“min”时间前到达而在“min”和“max”时间内没有保持为真)。
- (2) 如果两信号的时序差大于“max”。

该断言必须在时间窗口内一直成功。这意味着信号 1 和信号 2 时差从 $(min-1)$ 到 $(max+1)$ 变化的过程将覆盖所有的成功和至少一个错误条件。如果信号 1 和信号 2 的时序固定，那么“min”和“max”的值同样也是固定的。对于固定时序关系，所有可能成功的情况和至少一个错误条件能够在时间从 $(min-1)$ 到 $(max+1)$ 之间观测到。

7.2 双信号 SVA Assertion Test

Bench(ATB)

本节中，我们展示如何创建一个可配置的 ATB。目的是为双信号的 SVA 代码创建一个能产生激励的 testbench。ATB 最基本的要求是对断言的所有成功和至少一个错误结果正确地响应。双信号的断言包含一个前行信号(LS)和一个尾随信号(TS)。考虑下面的实例：

```
signal1 && signal2  
signal1 |-> signal2  
signal1 ##[1:3] signal2  
signal1 |-> ##2 signal2
```

这些实例中，无论我们是检测逻辑关系还是时序关系，signal1 为前行信号，signal2 为尾随信号。

7.2.1 双信号的逻辑关系

SVA 中双信号的逻辑关系有几种。逻辑关系在每个时钟都被计算。换言之，它们是组合检验。

图 7-3 显示了双信号的逻辑关系树。根据该图，共有 16 种逻辑关系可能。逻辑关系树包含了以下几种可能：

- 电平敏感双信号逻辑关系。
- 边沿敏感双信号逻辑关系。
- 带有交叠蕴含(overlapping implication)的电平敏感双信号逻辑关系。
- 带有交叠蕴含(overlapping implication)的边沿敏感双信号逻辑关系。

任何电平敏感双信号断言的前行信号(LS)和尾随信号(TS)有以下几种可能：

- HH – LS 和 TS 都为高
- HL – LS 为高、TS 为低

c. LH - LS 为低、TS 为高

d. LL - LS 和 TS 都为低

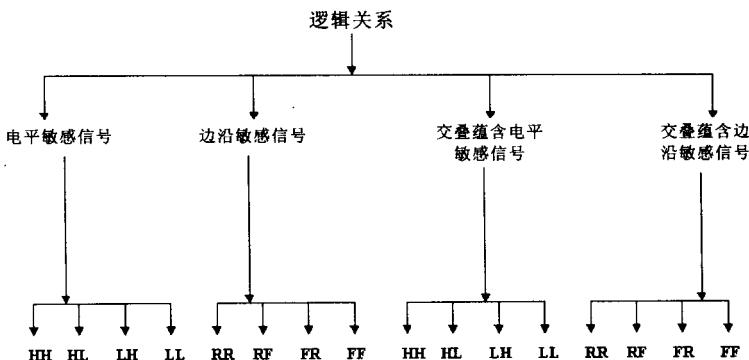


图 7-3 双信号 SVA 逻辑关系树

任何边沿敏感双信号断言的前行信号(LS)和尾随信号(TS)的有以下几种可能：

- RR - LS 和 TS 有上升沿
- RF - LS 有上升沿、TS 有下降沿
- FR - LS 有下降沿、TS 有上升沿
- FF - LS 和 TS 有下降沿

注意逻辑关系树中列有交叠蕴含。如果 LS 和 TS 之间没有时序关系，那么有交叠蕴含的检验器是一个简单的“if”语句。因此，它可以合并到逻辑关系树中。

在同一个断言中可能同时有电平敏感信号和边沿敏感信号。为了简化讨论，我们不把这种组合列在逻辑关系树中。在检验器中，如果前行信号电平敏感，而尾随信号边沿敏感，它就被当作电平敏感检验器。同样道理，如果前行信号边沿敏感，而尾随信号电平敏感，它就被当作边沿敏感检验器。下一节中，我们将展示如何为验证图 7-2 中所示的所有 16 种可能关系产生激励。

7.2.2 电平敏感逻辑关系激励的产生

电平敏感双信号逻辑关系的可能属性如下所列。该实例中我们使用的是逻辑与。它可以被其他任何逻辑运算所替换，而激励

产生保持不变。

```
// On a given clock edge, both leading signal and
// trailing signal are high

property p_1_hh;
  @(posedge clk) a && b;
endproperty

// On a given clock edge, the leading signal is
// high and the trailing signal is low

property p_1_hl;
  @(posedge clk) a && !b;
endproperty

// On a given clock edge, the leading signal is
// low and the trailing signal is high

property p_1_lh;
  @(posedge clk) !a && b;
endproperty

// On a given clock edge, both leading signal and
// trailing signal are low

property p_1_ll;
  @(posedge clk) !a && !b;
endproperty

a_1_hh : assert property(p_1_hh);
a_1_hl : assert property(p_1_hl);
a_1_lh : assert property(p_1_lh);
a_1_ll : assert property(p_1_ll);
```

交叠蕴含和简单逻辑运算符很相似，惟一区别在于先决条件。它对尾随信号隐藏了一个“if”语句。如果前行信号不为真，那么属性默认值是成功。交叠蕴含电平敏感双信号逻辑关系的可能属性如下所列：

```
// on a given clock edge, if the leading signal
// is high, check that the trailing signal is
```

```
// also high

property p4_oli_hh;
  @(posedge clk) a |-> b;
endproperty

// on a given clock edge, if the leading signal
// is high, check that the trailing signal is
// low

property p4_oli_hl;
  @(posedge clk) a |-> !b;
endproperty

// on a given clock edge, if the leading signal
// is low, check that the trailing signal is
// high

property p4_oli_lh;
  @(posedge clk) !a |-> b;
endproperty

// on a given clock edge, if the leading signal
// is low, check that the trailing signal is
// low

property p4_oli_ll;
  @(posedge clk) !a |-> !b;
endproperty

a4_oli_hh: assert property(p4_oli_hh);
a4_oli_hl: assert property(p4_oli_hl);
a4_oli_lh: assert property(p4_oli_lh);
a4_oli_ll: assert property(p4_oli_ll);
```

检验电平敏感信号的逻辑关系共有四种可能的输入条件(00, 01, 10, 11)。通过产生覆盖这四种可能条件的激励，我们可以验证电平敏感双信号间的任何逻辑操作。同样的激励也满足图 7-3 中的 4 种条件(HH, HL, LH, LL)。

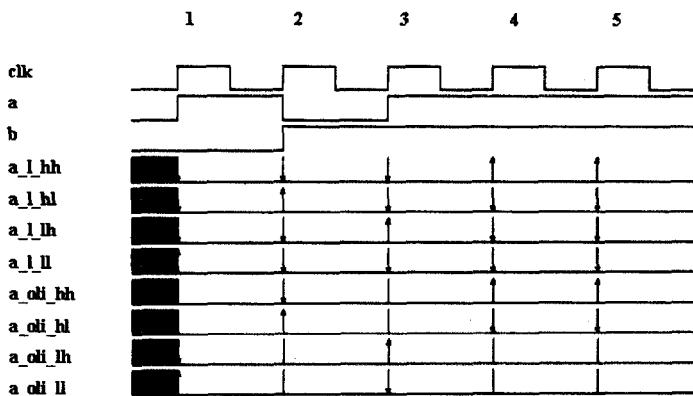


图 7-4 电平敏感双信号逻辑关系波形

下面的 Verilog 测试代码是一个简单两位计数器的实例。计数器的 LSB 驱动前行信号“a”，而 MSB 驱动尾随信号“b”。图 7-4 显示了用该 Verilog 实例生成的激励测试上面属性的结果。图中所使用的激励使每个断言的所有成功和至少一个实际错误都被覆盖到了。

```
// sample test code for logical relationship
// between level sensitive signals

logic [1:0] logical_op_reg;
logical_op_reg = 2'b00;

for(i=0; i<4; i++)
begin
    a <= logical_op_reg[0];
    b <= logical_op_reg[1];
    repeat(1) @ (posedge clk);
    logical_op_reg++;
end
```

7.2.3 边沿敏感逻辑关系激励的产生

边沿敏感双信号逻辑关系的可能属性如下所示：

```
// on a given clock edge the leading signal has a
// falling edge and the trailing signal has a
// falling edge
```

```
property p2_ff;
  @ (posedge clk) $fell(a) && $fell(b);
endproperty

// on a given clock edge the leading signal has a
// falling edge and the trailing signal has a
// rising edge

property p2_fr;
  @ (posedge clk) $fell(a) && $rose(b);
endproperty

// on a given clock edge the leading signal has a
// rising edge and the trailing signal has a
// falling edge

property p2_rf;
  @ (posedge clk) $rose(a) && $fell(b);
endproperty

// on a given clock edge the leading signal has a
// rising edge and the trailing signal has a
// rising edge

property p2_rr;
  @ (posedge clk) $rose(a) && $rose(b);
endproperty

a2_ff: assert property(p2_ff);
a2_fr: assert property(p2_fr);
a2_rf: assert property(p2_rf);
a2_rr: assert property(p2_rr);
```

交叠蕴含电平敏感双信号逻辑关系的可能属性如下所示：

```
// on a given clock edge, if the leading signal
// has a falling edge, then the trailing signal
// must have a falling edge

property p4_oei_ff;
  @ (posedge clk) $fell(a) |-> $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then the trailing signal
```

```

// must have a rising edge

property p4_oei_fr;
  @(posedge clk) $fell(a) |-> $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then the trailing signal
// must have a falling edge

property p4_oei_rf;
  @(posedge clk) $rose(a) |-> $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then the trailing signal
// must have a rising edge

property p4_oei_rr;
  @(posedge clk) $rose(a) |-> $rose(b);
endproperty

a4_oei_ff: assert property(p4_oei_ff);
a4_oei_fr: assert property(p4_oei_fr);
a4_oei_rf: assert property(p4_oei_rf);
a4_oei_rr: assert property(p4_oei_rr);

```

虽然我们只检验逻辑关系，但是由于\$rose 和\$fell 结构的运用使得激励的产生稍微复杂一些。而产生的激励应覆盖各种边沿转换，下面的 Verilog 测试代码是一个两位计数器。它覆盖了下降-下降(FF)和下降-上升(FR)条件下所有成功的可能，如图 7-5 所示。

```

// sample test code for logical relationship
// between edge sensitive signals

for(i=0; i<8; i++)
begin
  a <= logical_op_reg[0];
  b <= logical_op_reg[1];
  repeat(1) @(posedge clk);
  logical_op_reg++;
end

```

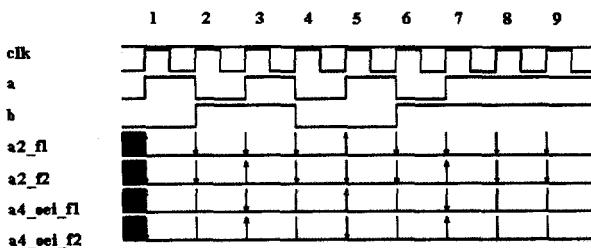


图 7-5 边沿敏感信号逻辑条件- FF, FR

下面的 Verilog 测试代码也是一个两位计数器的实例，但它的计数位是位反。它覆盖了上升-下降(RF)和上升-上升(RR)条件下所有成功的可能，如图 7-6 所示。

```
// sample test code for logical relationship
// between edge sensitive signals
for(i=0; i<8; i++)
begin
    a <= !logical_op_reg[0];
    b <= !logical_op_reg[1];
    repeat(1) @(posedge clk);
    logical_op_reg++;
end
```

由前面的两节可见，满足逻辑关系比较简单。但随着逻辑表达式中信号数目的增加，输入条件的可能性也相应增加。

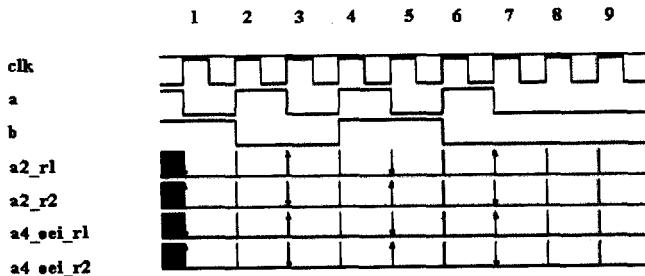


图 7-6 边沿敏感信号逻辑条件- RR, RF

7.2.4 双信号的时序关系

前行信号和尾随信号之间的时序关系可能是固定时延或者可

变时延。其时序关系树和逻辑关系树非常类似，只是可能性的数目增加了一倍(固定时延和可变时延)，正如图 7-7 所示。同时需要注意的是前行信号和尾随信号之间时序关系有非交叠条件。

时序关系树包含以下可能：

- (1) 电平敏感双信号固定时序关系。
- (2) 电平敏感双信号可变时序关系。
- (3) 边沿敏感双信号固定时序关系。
- (4) 边沿敏感双信号可变时序关系。
- (5) 非交叠蕴含电平敏感双信号固定时序关系。
- (6) 非交叠蕴含电平敏感双信号可变时序关系。
- (7) 非交叠蕴含边沿敏感双信号固定时序关系。
- (8) 非交叠蕴含边沿敏感双信号可变时序关系。

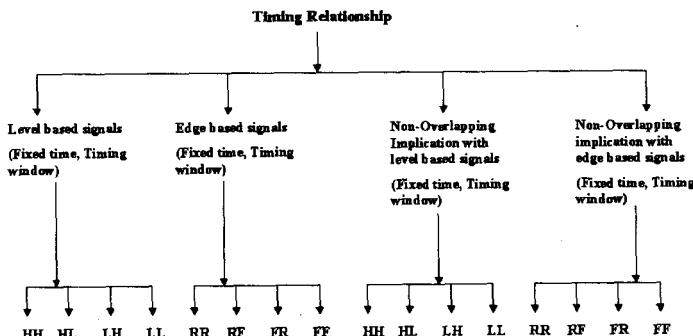


图 7-7 时序关系树

电平敏感信号和边沿敏感信号之间可能有时序关系。为简化时序关系树，这些可能性没有被列出。如 7.2.1 所述，如果前行信号电平敏感而尾随信号边沿敏感，它就被当作电平敏感检验器。同样道理，如果前行信号边沿敏感而尾随信号电平敏感，它就被当作边沿敏感检验器。

7.2.5 时序关系激励的产生

电平敏感双信号固定时序关系的可能属性如下所示：

```
// On a given clock edge, the leading signal is
// high and after "min_time" clock cycles the
// trailing signal is high

property p3_hh;
  @(posedge clk) a ##min_time b;
endproperty

// On a given clock edge, the leading signal is
// high and after "min_time" clock cycles the
// trailing signal is low

property p3_hl;
  @(posedge clk) a ##min_time !b;
endproperty

// On a given clock edge, the leading signal is
// low and after "min_time" clock cycles the
// trailing signal is high

property p3_lh;
  @(posedge clk) !a ##min_time b;
endproperty

// On a given clock edge, the leading signal is
// low and after "min_time" clock cycles the
// trailing signal is low

property p3_ll;
  @(posedge clk) !a ##min_time !b;
endproperty

a3_f1: assert property(p3_hh);
a3_f2: assert property(p3_hl);
a3_f3: assert property(p3_lh);
a3_f4: assert property(p3_ll);
```

电平敏感双信号可变时序关系的可能属性如下所示：

```
// On a given clock edge, the leading signal is
// high and between "min_time" and "max_time"
// clock cycles the trailing signal is high

property p3_w1_hh;
```

```

property p3_w1_hh;
    @(posedge clk) a ## [min_time : max_time] b;
endproperty

// On a given clock edge, the leading signal is
// high and between "min_time" and "max_time"
// clock cycles the trailing signal is low

property p3_w2_hl;
    @(posedge clk) a ## [min_time : max_time] !b;
endproperty

// On a given clock edge, the leading signal is
// low and between "min_time" and "max_time"
// clock cycles the trailing signal is high

property p3_w3_lh;
    @(posedge clk) !a ## [min_time : max_time] b;
endproperty

// On a given clock edge, the leading signal is
// low and between "min_time" and "max_time"
// clock cycles the trailing signal is low

property p3_w4_ll;
    @(posedge clk) !a ## [min_time : max_time] !b;
endproperty

a3_w1: assert property(p3_w1_hh);
a3_w2: assert property(p3_w2_hl);
a3_w3: assert property(p3_w3_lh);
a3_w4: assert property(p3_w4_ll);

```

交叠蕴含电平敏感双信号固定时序关系的可能属性如下所示：

```

// On a given clock edge, if the leading signal
// is high, then after "min_time" clock cycles
// the trailing signal must be high

property p5_f_hh;
    @(posedge clk) a |-> ##min_time b;
endproperty

// On a given clock edge, if the leading signal
// is high, then after "min_time" clock cycles
// the trailing signal must be low

```

```
property p5_f_hl;
  @ (posedge clk) a |-> ##min_time !b;
endproperty

// On a given clock edge, if the leading signal
// is low, then after "min_time" clock cycles
// the trailing signal must be high

property p5_f_lh;
  @ (posedge clk) !a |-> ##min_time b;
endproperty

// On a given clock edge, if the leading signal
// is low, then after "min_time" clock cycles
// the trailing signal must be low

property p5_f_ll;
  @ (posedge clk) !a |-> ##min_time !b;
endproperty

a5_f_hh: assert property(p5_f_hh);
a5_f_hl: assert property(p5_f_hl);
a5_f_lh: assert property(p5_f_lh);
a5_f_ll: assert property(p5_f_ll);
```

交叠蕴含电平敏感双信号可变时序关系的可能属性如下所示：

```
// On a given clock edge, if the leading signal
// is high, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be high

property p5_w_hh;
  @ (posedge clk)
  a |-> ##[min_time : max_time] b;
endproperty

// On a given clock edge, if the leading signal
// is high, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be low

property p5_w_hl;
  @ (posedge clk)
```

```

    a |-> ##[min_time : max_time] !b;
endproperty

// On a given clock edge, if the leading signal
// is low, then between "max_time" and
// "max_time" clock cycles the trailing signal
// must be high

property p5_w_lh;
  @(posedge clk)
  !a |-> ##[min_time : max_time] b;
endproperty

// On a given clock edge, if the leading signal
// is low, then between "max_time" and
// "max_time" clock cycles the trailing signal
// must be low

property p5_w_ll;
  @(posedge clk)
  !a |-> ##[min_time : max_time] !b;
endproperty

a5_w_hh: assert property(p5_w_hh);
a5_w_hl: assert property(p5_w_hl);
a5_w_lh: assert property(p5_w_lh);
a5_w_ll: assert property(p5_w_ll);

```

边沿敏感双信号固定时序关系的可能属性如下所示：

```

// on a given clock edge, the leading signal has
// a falling edge and after "max_time" cycle the
// trailing signal has a falling edge.

property p4_f_ff;
  @(posedge clk) $fell(a) ##min_time $fell(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and after "max_time" cycle the
// trailing signal has a rising edge

property p4_f_rr;
  @(posedge clk) $rose(a) ##min_time $rose(b);

```

```
endproperty

// on a given clock edge, the leading signal has
// a falling edge and after "max_time" cycle the
// trailing signal has a rising edge

property p4_f_fr;
  @ (posedge clk) $fell(a) ##min_time $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and after "max_time" cycles the
// trailing signal has a falling edge

property p4_f_rf;
  @ (posedge clk) $rose(a) ##min_time $fell(b);
endproperty

a4_f_rr: assert property(p4_f_rr);
a4_f_ff: assert property(p4_f_ff);
a4_f_rf: assert property(p4_f_rf);
a4_f_fr: assert property(p4_f_fr);
```

边沿敏感双信号可变时序关系的可能属性如下所示：

```
// on a given clock edge, the leading signal has
// a falling edge and within "max_time" to
// "max_time" cycles the trailing signal has a
// falling edge

property p4_w_ff;
  @ (posedge clk)
  $fell(a) ##[min_time : max_time] $fell(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and within "max_time" to
// "max_time" cycles the trailing signal has a
// rising edge

property p4_w_rr;
  @ (posedge clk)
  $rose(a) ##[min_time : max_time] $rose(b);
endproperty
```

```

    // on a given clock edge, the leading signal has
    // a falling edge and within "max_time" to
    // "max_time" cycles the trailing signal has a
    // rising edge

property p4_w_fr;
    @(posedge clk)
        $fell(a) ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and within "max_time" to
// "max_time" cycles the trailing signal has a
// falling edge

property p4_w_rf;
    @(posedge clk)
        $rose(a) ##[min_time : max_time] $fell(b);
endproperty

a4_w_rr: assert property(p4_w_rr);
a4_w_ff: assert property(p4_w_ff);
a4_w_rf: assert property(p4_w_rf);
a4_w_fr: assert property(p4_w_fr);

```

交叠蕴含边沿敏感双信号固定时序关系的可能属性如下所示：

```

// on a given clock edge, if the leading signal
// has a falling edge, then after "max_time"
// cycles the trailing signal must have a
// falling edge

property p6_f_ff;
    @(posedge clk)
        $fell(a) |-> ##min_time $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then after "max_time"
// cycles the trailing signal must have a
// rising edge

property p6_f_rr;
    @(posedge clk)

```

```
$rose(a) |-> ##min_time $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then after "max_time"
// cycles the trailing signal must have a
// rising edge

property p6_f_fr;
  @(posedge clk)
  $fell(a) |-> ##min_time $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then after "max_time"
// cycles the trailing signal must have a
// falling edge

property p6_f_rf;
  @(posedge clk)
  $rose(a) |-> ##min_time $fell(b);
endproperty

a6_f_rr: assert property(p6_f_rr);
a6_f_ff: assert property(p6_f_ff);
a6_f_rf: assert property(p6_f_rf);
a6_f_fr: assert property(p6_f_fr);
```

交叠蕴含边沿敏感双信号可变时序关系的可能属性如下所示：

```
// on a given clock edge, if the leading signal
// has a falling edge, then within "max_time" to
// "max_time" cycles the trailing signal must
// have a falling edge
property p6_w_ff;
  @(posedge clk)
  $fell(a) |-> ##[min_time : max_time] $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then within "max_time" to
// "max_time" cycles the trailing signal must
// have a rising edge

property p6_w_rr;
```

```

@(posedge clk)
$rose(a) |-> ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then within "max_time" to
// "max_time" cycles the trailing signal must
// have a rising edge

property p6_w_fr;
@(posedge clk)
$fell(a) |-> ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then within "max_time" to
// "max_time" cycles the trailing signal must
// have a falling edge

property p6_w_rf;
@(posedge clk)
$rose(a) |-> ##[min_time : max_time] $fell(b);
endproperty

a6_w_rr: assert property(p6_w_rr);
a6_w_ff: assert property(p6_w_ff);
a6_w_rf: assert property(p6_w_rf);
a6_w_fr: assert property(p6_w_fr);

```

下面 Verilog 测试代码实例能产生除了未知不可测的情况以外的所有双信号时序关系的激励。注意，信号“a”和“b”根据“时序级别”而被初始化。例如，如果“时序级别”设为“10”时，那么测试代码将为“高有效”前行信号和“低有效”尾随信号产生激励。对于一个边沿敏感检验器，如果“时序级别”设为“10”，那么测试代码将为“上升沿”前行信号和“下降沿”尾随信号产生激励。可以用一个简单的“for”循环来产生从(min_time-1)到(max_time+3)的时间窗口。这将能保证覆盖所有可能的成功和至少一个失败。通过以(max_time+1)为上限的时间窗口也能达到同样的结果。但我们这里用(max_time+3)来产生更多的失败条件。如果时间固定，那么“min_time”和“max_time”的值相等。

图 7-8 为电平敏感双信号(前行信号为低有效而尾随信号为高有效)固定时序关系属性的波形图实例。图 7-9 为边沿敏感双信号(前行、尾随信号寻找上升沿)可变时序关系属性的波形图实例。

```
// sample Verilog test code for timing
// relationship between two signals

if(timing_level == 2'b11) begin
  a = 1'b0; b=1'b0; end
if(timing_level == 2'b00) begin
  a = 1'b1; b=1'b1; end
if(timing_level == 2'b01) begin
  a = 1'b1; b=1'b0; end
if(timing_level == 2'b10) begin
  a = 1'b0; b=1'b1; end
for(i=(min_time-1); i<(max_time+3); i++)
begin
  repeat(1) @ (posedge clk);
  a <= ~a;
  if(i == 0)
  begin
    b <= ~b;
    repeat(1) @ (posedge clk);
    a <= ~a; b <= ~b;
  end
  else
  begin
    repeat(1) @ (posedge clk);
    a <= ~a;
    repeat((i-1)) @ (posedge clk);
    b <= ~b;
    repeat(1) @ (posedge clk);
    b <= ~b;
  end
end
end
Fixed timing = 2 cycles
```

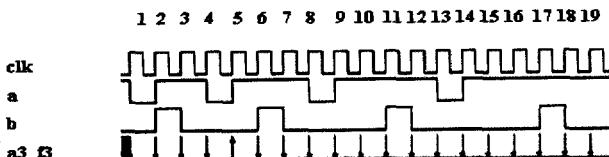


图 7-8 电平敏感信号固定时序

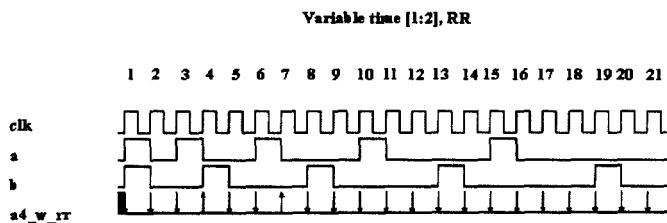


图 7-9 边沿敏感信号可变时序

时序关系可以通过从(min-1)到(max+1)循环来测试。这将确保所有成功和至少一个失败得到测试。信号的重复是时序关系的扩展。重复涉及到时序关系，但它要求前行信号或者尾随信号重复其值达到预定次数。我们将在下面一节中详细讨论重复关系。

7.2.6 双信号的重复关系

双信号之间的重复有两个主要的种类：

- (1) **Repeat after** —— 在前行信号预期沿到达之后，经过时延或者不经过时延，尾随信号被预期重复“n”次。
- (2) **Repeat until** —— 在前行信号预期沿到达之后，前行信号在尾随信号的某个预期值到达之前不断重复。

重复运算符通常涉及边沿敏感信号和电平敏感信号的组合。前行信号通常是边沿敏感信号，而尾随信号通常是电平敏感信号。“repeat until”条件可以把一个边沿敏感信号作为尾随信号。重复属性的命名习惯如下：

RH – LS 有上升沿并且 TS 为高

RL – LS 有上升沿并且 TS 为低

FH – LS 有下降沿并且 TS 为高

FL – LS 有下降沿并且 TS 为低

双信号“repeat after”关系的可能属性如下所示：

```
// on a given clock edge the leading signal has a
// rising edge and after "start_wait" cycles, the
// trailing signal is high "repetition" times
```

```
property p7_c_rpt_rh;
```

```
@(posedge clk)
$rose(a) ##start_wait b[*repetition];
endproperty

// on a given clock edge the leading signal has a
// rising edge and after "start_wait" cycles, the
// trailing signal is low "repetition" times

property p7_c_rpt_rl;
  @(posedge clk)
  $rose(a) ##start_wait !b[*repetition];
endproperty

// on a given clock edge the leading signal has a
// falling edge and after "start_wait" cycles,
// the trailing signal is high "repetition" times

property p7_c_rpt_fh;
  @(posedge clk)
  $fell(a) ##start_wait b[*repetition];
endproperty

// on a given clock edge the leading signal has a
// falling edge and after "start_wait" cycles,
// the trailing signal is low "repetition" times

property p7_c_rpt_fl;
  @(posedge clk)
  $fell(a) ##start_wait !b[*repetition];
endproperty
```

双信号“repeat until”关系的可能属性如下所示：

```
// on a give clock edge, the leading signal has a
// rising edge and stays high until the trailing
// signal is low

property p7_cu_rpt_rl;
  @(posedge clk) $rose(b) ##0 b[*1:$] ##1 !a;
endproperty

// on a given clock edge, the leading signal has
// a falling edge and stays low until the
// trailing signal is low
```

```

property p7_cu_rpt_f1;
  @(posedge clk) $fell(b) ##0 !b[*1:$] ##1 !a;
endproperty

// on a given clock edge, the leading signal has
// a rising edge and stays high until the
// trailing signal is high

property p7_cu_rpt_rh;
  @(posedge clk) $rose(b) ##0 b[*1:$] ##1 a;
endproperty

// on a given clock edge, the leading signal has
// a falling edge and stays high until the
// trailing signal is high

property p7_cu_rpt_fh;
  @(posedge clk) $fell(b) ##0 !b[*1:$] ##1 a;
endproperty

```

下面实例的代码用来产生激励以验证所有“repeat after”和“repeat until”所有可能的属性。这里的激励和前面时序中的激励使用相同的概念。它在(repetition-1)到(repetition+3)之间循环来覆盖所有可能的成功和至少一个失败条件。变量“start_wait”定义了在寻找尾随信号重复之前所等待的周期数(相对于“repeat-after”条件而言)。变量“stop_wait”定义了重设前行信号之前所等待的周期数(相对于“repeat_until”条件而言)。

```

// sample Verilog test code for repetition

logic [1:0] stop_wait;

if(rpt_edge == 2'b11) begin
  a = 1'b0; b=1'b0; end

if(rpt_edge == 2'b00) begin
  a = 1'b1; b=1'b1; end

if(rpt_edge == 2'b01) begin
  a = 1'b1; b=1'b0; end

if(rpt_edge == 2'b10) begin

```

```

a = 1'b0; b=1'b1; end

for(i=(repetition-1); i<(repetition+3); i++)
begin
    repeat(1) @ (posedge clk);
    a <= ~a;
    repeat(start_wait) @ (posedge clk);
    b <= ~b;

    // consecutive repeat condition

    repeat((i)) @ (posedge clk);
    b <= ~b;
    stop_wait <= $random() % 4;
    repeat(stop_wait[0]) @ (posedge clk);
    a <= ~a;
end

```

图 7-10 为“repeat after”条件的波形图实例。图 7-11 为“repeat until”条件的波形图实例。

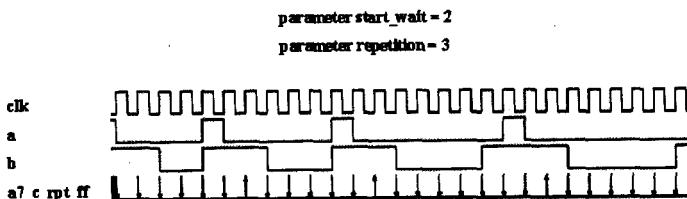


图 7-10 “repeat after” 条件波形图

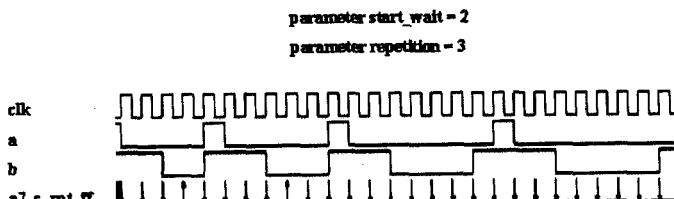


图 7-11 “repeat until” 条件波形图

到现在，我们讨论了几种双信号可能的关系。SVA 结构十分丰富，双信号的可能关系相当多。我们不可能为所有的情况提供解决方案，只是提供了一小部分解答。随着涉及到的信号数量增加，解决方案的难度也相应增加。

7.2.7 双信号 ATB 环境

在前面几节中，我们看到了如何为测试双信号间的不同关系产生完全的激励。也看到若干可以满足不同关系的 Verilog 测试实例代码。在本节中，我们把这些部件整合起来形成一个可配置的测试单元，如图 7-12 所示。

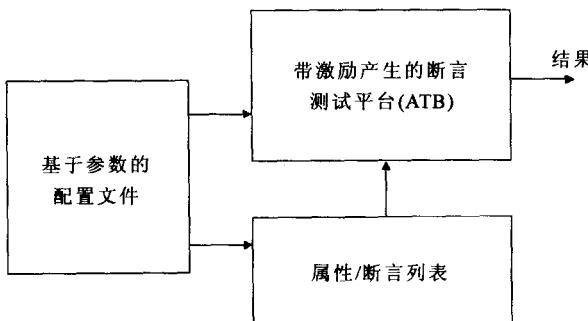


图 7-12 ATB 环境

测试结构包含三部分：

- (1) 一个基于参数的配置文件，用来设定用户希望测试的关系。
- (2) 一个 SVA 列表文件，包含属性定义，和可以根据参数配置选择性断言属性的代码。
- (3) 顶层 Verilog 测试代码，包含前几节讨论的各种激励产生模块。根据参数配置，将执行相应的激励产生模块以彻底地验证当前被测属性。参数定义如表 7-1 所示。

表 7-1 参数定义

参 数	功 能
parameter sig_edge = 0;	定义相关信号是否为边沿敏感，0 为否，1 为是
parameter sig1_edge = 1;	定义前行信号边沿，1 为上升沿，0 为下降沿(仅在逻辑关系中使用)
parameter logic_op = 0;	定义断言是否为逻辑关系，0 为否，1 为是
parameter timing = 1;	定义断言是否为时序关系，0 为否，1 为是

(续表)

参 数	功 能
parameter min_time = 2;	定义时间信息，最大时间应该大于最小时间，
parameter max_time = 2;	最小时间不能为 0，而最大时间应该有界
parameter timing_level = 2'b10;	定义信号电平/边沿，“10”表示电平 HL 和边沿 RF
parameter o_l_implication = 0;	定义电平敏感信号是否交叠蕴含，0 为否，1 为是
parameter o_e_implication = 0;	定义边沿敏感信号是否交叠蕴含，0 为否，1 为是
parameter non_o_implication = 1;	定义非交叠蕴含
parameter rpt_me = 0;	定义有重复
parameter rpt_edge = 2'b00;	定义信号电平/边沿
parameter start_wait = 2;	定义“repeat after”中重复到达前的等待时间
parameter repetition = 3;	定义重复数量。必须大于 1
parameter c_rpt = 0;	定义测试后重复
parameter c_rpt_until = 0;	定义测试前重复

通过设置适当的参数，用户可以为特定关系生成激励。

- 如果参数“logic_op”设为 1 而其他参数设为 0，那么 ATB 将产生双信号逻辑关系激励。
- 如果参数“timing”设为 1 而其他参数设为 0，那么 ATB 将产生双信号时序关系激励。用户可以通过设置“min_time”和“max_time”来指定是否需要固定时延还是可变时延。如果用户将“sig_edge”设为 1，那么这种信号被当作边沿敏感信号来处理。

双信号 SVA 验证的列表文件实例如下所示：

```
module sig_sva (a, b, clk);
  // include the parameter definitions
  `include "config.v"
```

```
input logic a, b, clk;

// List definitions of all properties under test

// code to selectively include assertions

always@(posedge clk)
begin

    // logical relationship between two level
    // sensitive signals

    if(logic_op == 1 && timing == 0 && sig_edge == 0)
    begin
        a_1_hh : assert property(p_1_hh);
        a_1_hl : assert property(p_1_hl);
        a_1_lh : assert property(p_1_lh);
        a_1_ll : assert property(p_1_ll);
    end

    // logical relationship between two edge
    // sensitive signals FF, FR

    if(logic_op == 1 && timing == 0 && sig_edge == 1
    && sig1_edge == 0)
    begin
        a2_ff: assert property(p2_ff);
        a2_fr: assert property(p2_fr);
    end

    // logical relationship between 2 edge sensitive
    // signals RF, RR

    if(logic_op == 1 && timing == 0 && sig_edge == 1
    && sig1_edge == 1)
    begin
        a2_rf: assert property(p2_rf);
        a2_rr: assert property(p2_rr);
    end

    // timing relationship between 2 level sensitive
    // signals

    if(logic_op == 0 && timing == 1 && sig_edge == 0
```

```
&& non_o_implication == 0)
begin
if(min_time == max_time)
begin
if(timing_level == 2'b11)
    a3_hh: assert property(p3_hh);
if(timing_level == 2'b10)
    a3_hl: assert property(p3_hl);
if(timing_level == 2'b01)
    a3_lh: assert property(p3_lh);
if(timing_level == 2'b00)
    a3_ll: assert property(p3_ll);
end
if(min_time != max_time)
begin
if(timing_level == 2'b11)
    a3_w1_hh: assert property(p3_w1_hh);
if(timing_level == 2'b10)
    a3_w2_hl: assert property(p3_w2_hl);
if(timing_level == 2'b01)
    a3_w3_lh: assert property(p3_w3_lh);
if(timing_level == 2'b00)
    a3_w4_ll: assert property(p3_w4_ll);
end
end

// logical relationship between 2 level sensitive
// signals with overlapping implication

if((logic_op == 1 || o_l_implication == 1) &&
timing == 0 && sig_edge == 0)
begin
a4_oli_hh: assert property(p4_oli_hh);
a4_oli_hl: assert property(p4_oli_hl);
a4_oli_lh: assert property(p4_oli_lh);
a4_oli_ll: assert property(p4_oli_ll);
end

// logical relationship between 2 edge sensitive
// signals with overlapping implication FF, FR
if((logic_op == 1 || o_e_implication == 1) &&
timing == 0 && sig1_edge == 1 && sig1_edge == 0)
begin
a4_oei_ll: assert property(p4_oei_ll);

```

```
a4_oei_lh: assert property(p4_oei_lh);
end

// logical relationship between 2 edge sensitive
// signals with overlapping implication RF, RR

if((logic_op == 1 || o_e_implication == 1) &&
   timing == 0 && sig_edge == 1 && sigl_edge == 1)
begin
  a4_oei_hl: assert property(p4_oei_hl);
  a4_oei_hh: assert property(p4_oei_hh);
end

if(logic_op == 0 && timing == 1 && sig_edge == 1
   && non_o_implication == 0)
begin
  if(min_time == max_time)
  begin
    a4_f_rr: assert property(p4_f_rr);
    a4_f_ff: assert property(p4_f_ff);
    a4_f_rf: assert property(p4_f_rf);
    a4_f_fr: assert property(p4_f_fr);
  end
  if(min_time != max_time)
  begin
    a4_w_rr: assert property(p4_w_rr);
    a4_w_ff: assert property(p4_w_ff);
    a4_w_rf: assert property(p4_w_rf);
    a4_w_fr: assert property(p4_w_fr);
  end
end

// timing relation between 2 level sensitive
// signals with non-overlapping implication

if(logic_op == 0 && timing == 1 && sig_edge == 0
   && non_o_implication == 1)
begin
  if(min_time == max_time)
  begin
    if(timing_level == 2'b11)
      a5_f_hh: assert property(p5_f_hh);
    if(timing_level == 2'b10)
      a5_f_hl: assert property(p5_f_hl);
  end
end
```

```
if(timing_level == 2'b01)
    a5_f_lh: assert property(p5_f_lh);
if(timing_level == 2'b00)
    a5_f_ll: assert property(p5_f_ll);
end
if(min_time != max_time)
begin
if(timing_level == 2'b11)
    a5_w_hh: assert property(p5_w_hh);
if(timing_level == 2'b10)
    a5_w_hl: assert property(p5_w_hl);
if(timing_level == 2'b01)
    a5_w_lh: assert property(p5_w_lh);
if(timing_level == 2'b00)
    a5_w_ll: assert property(p5_w_ll);
end
end

// timing relation between 2 edge sensitive
// signals with non-overlapping implication

if(logic_op == 0 && timing == 1 && sig_edge == 1
&& non_o_implication == 1)
begin

if(min_time == max_time)
begin
a6_f_rr: assert property(p6_f_rr);
a6_f_ff: assert property(p6_f_ff);
a6_f_rf: assert property(p6_f_rf);
a6_f_fr: assert property(p6_f_fr);
end
if(min_time != max_time)
begin
a6_w_rr: assert property(p6_w_rr);
a6_w_ff: assert property(p6_w_ff);
a6_w_rf: assert property(p6_w_rf);
a6_w_fr: assert property(p6_w_fr);
end
end

// repetition relationship

if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
```

```
2'b11)
begin
a7_c_rpt_rh: assert property(p7_c_rpt_rh);
a7_cu_rpt_rh: assert property(p7_cu_rpt_rh);
end

if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
2'b10)
begin
a7_c_rpt_rl: assert property(p7_c_rpt_rl);
a7_cu_rpt_rl: assert property(p7_cu_rpt_rl);
end

if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
2'b01)
begin
a7_c_rpt_fh: assert property(p7_c_rpt_fh);
a7_cu_rpt_fh: assert property(p7_cu_rpt_fh);
end

if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
2'b00)
begin
a7_c_rpt_fl: assert property(p7_c_rpt_fl);
a7_cu_rpt_fl: assert property(p7_cu_rpt_fl);
end
end

// Config Parameters illegal values. If
// logical_op is asserted then timing cannot be
// asserted

property config_check1;
@(posedge clk)
(logic_op == 1) |->
(timing == 0);
endproperty
// only one of the implication operators can be
// asserted at any time

property config_check2;
@(posedge clk)
$onehot0({o_l_implication,
o_e_implication, non_o_implication});
```

```
endproperty

// min_time should be atleast 1

property config_check3;
  @ (posedge clk)
    (timing == 1) |-> (min_time >= 1);
endproperty

// repetition should be greater than one

property config_check4;
  @ (posedge clk)
    ((c_rpt == 1) && (rpt_me == 1)) |->
      (repetition > 1);
endproperty

a_check1: assert property(config_check1);
a_check2: assert property(config_check2);
a_check3: assert property(config_check3);
a_check4: assert property(config_check4);

endmodule
```

ATB 根据参数配置来执行。双信号 SVA 验证的 ATB 实例如下所示：

```
module sig_sva_tb;

logic a, b;
logic clk;
logic [1:0] rpt_wait;
logic [1:0] stop_wait;

`include "config.v"

integer i, j;
logic [1:0] logical_op_reg;

initial
begin
  clk = 1'b0; a=1'b0; b=1'b0;
  logical_op_reg = 2'b00;
```

```
//*****
// case 1
// logical operation, overlapping implication
// level sensitive signals
//*****

if((logic_op == 1 || o_l_implication == 1) &&
timing == 0 && sig_edge == 0)
begin
for(i=0; i<4; i++)
begin
    a <= logical_op_reg[0];
    b <= logical_op_reg[1];
    repeat(1) @(posedge clk);
    logical_op_reg++;
end
end

//*****
// case 2
// logical operation, overlapping implication
// edge sensitive signals
//*****
```



```
if((logic_op == 1 || o_e_implication == 1) &&
timing == 0 && sig_edge == 1)
begin

    if(sig1_edge == 0) // ff, fr
    begin
        for(i=0; i<8; i++)
        begin
            a <= logical_op_reg[0];
            b <= logical_op_reg[1];
            repeat(1) @(posedge clk);
            logical_op_reg++;
        end
    end

    if(sig1_edge == 1) // rr, rf
    begin
        for(i=0; i<8; i++)
        begin
            a <= !logical_op_reg[0];
```

```
b <= !logical_op_reg[1];
repeat(1) @ (posedge clk);
logical_op_reg++;
end
end

end

//***** case 3 *****

// timing relation between 2 signals
//***** case 3 *****

if(logic_op == 0 && timing == 1)
begin

if(timing_level == 2'b11) begin
a = 1'b0; b=1'b0;
end

if(timing_level== 2'b00) begin
a = 1'b1; b=1'b1;
end

if(timing_level == 2'b01) begin
a = 1'b1; b=1'b0;
end

if(timing_level == 2'b10) begin
a = 1'b0; b=1'b1;
end
for(i=(min_time-1); i<(max_time+3); i++)
begin
repeat(1) @ (posedge clk);
a <= ~a;
if(i == 0)
begin
b <= ~b;
repeat(1) @ (posedge clk);
a <= ~a; b <= ~b;
end
else
begin
repeat(1) @ (posedge clk);
```

```
a<= ~a;
repeat((i-1)) @ (posedge clk);
b<= ~b;
repeat(1) @ (posedge clk);
b<= ~b;
end
end
end

//***** *****
// case 4
// repetitions
//***** *****

if(rpt_me == 1)
begin

if(rpt_edge == 2'b11) begin
a = 1'b0; b=1'b0;
end

if(rpt_edge == 2'b00) begin
a = 1'b1; b=1'b1;
end

if(rpt_edge == 2'b01) begin
a = 1'b1; b=1'b0;
end

if(rpt_edge == 2'b10) begin
a = 1'b0; b=1'b1;
end

if(c_rpt == 1)
begin
for(i=(repetition-1); i<(repetition+3); i++)
begin
repeat(1) @ (posedge clk);
a <= ~a;
repeat(start_wait) @ (posedge clk);
b <= ~b;

// consecutive repeat condition

```

```

repeat((i)) @ (posedge clk);
b <= ~b;
stop_wait <= $random() % 4;
repeat(stop_wait[0]) @ (posedge clk);
a <= ~a;
end
end
end

repeat(2) @ (posedge clk);
$finish();
end

initial
forever clk = #25 ~clk;

endmodule

bind sig_sva_tb sig_sva il (a, b, clk);

```

注意，配置文件包含在 ATB 中，而 SVA 列表文件和 ATB 模块绑在一起。

7.3 一个 PCI 检验器的 ATB 实例

在本节中，我们用一个复杂的 SVA 检验器来展示如何利用 7.2 节中讨论到的概念进行功能验证。以下是第 6 章讨论的检验器。

信号“framen”被断言直到第一个数据阶段完成的延迟为 16 个周期。

```

sequence s_tchk9a;
@ (posedge clk)
(!irdyn && !trdyn),
endsequence

sequence s_tchk9b;
@ (posedge clk)
(!irdyn && !stopn),
endsequence

```

```

sequence s_tchk9_fast;
@(posedge clk)
$fell(framen) ##1 $fell(devseln);
endsequence
property p_tchk9_fast;
@(posedge clk)
s_tchk9_fast |>
(!framен && !devseln) throughout
(##[1:15] (s_tchk9a.ended || s_tchk9b.ended));
endproperty

a_tchk9_fast: assert property(p_tchk9_fast);
c_tchk9_fast: cover property(p_tchk9_fast);

```

属性 p_tchk9_fast 包含复杂的逻辑和时序关系。当一个有效条件开始时，该属性(s_tchk9_fast)被激活。一旦先行算子匹配，该属性有两种方式成功。如果假设让先行算子匹配的两个信号在计算的过程中保持断言状态，那么 s_tchk9a 或者 s_tchk9b 应该在 1~15 周期内匹配。

我们需要做下面的事情来彻底地检验该检验器：

- (1) 测试 “irdyn”、“trdyn”、“devseln” 和 “stopn” 信号之间所有可能的逻辑关系。
- (2) 测试先行算子和后续算子之间所有的时序关系。

根据 7.2 节的理论，4 个信号的所有逻辑组合有 16 个，正如表 7-2 所示。在先行算子成功匹配后的 1~15 个时钟周期内，表 7-2 中每个成功条件都应该发生。通过从(min-1)到(max+1)，即 1~16，循环该检验器，用户可以覆盖所有时延条件下可能的成功和至少一个错误条件。当循环到一个特定的时延，应该执行所有 16 种逻辑关系。因此，16 个可能的时间点和 16 种可能的逻辑条件组成 256 种情况。该检验在从 1~15 的每个时延值内处理 3 个条件，所以一共有 45 个真成功。换言之，这个检验能在 45 种不同条件下成功。我们能够用属性覆盖语句来对其进行交叉检验。

表 7-2 PCI 检验逻辑条件

irdyn	trdyn	devseln	stopn	状态
0	0	0	0	成功 (s_tchk9a && s_tchk9b)
0	0	0	1	成功 (s_tchk9a)
0	0	1	0	失败
0	0	1	1	失败
0	1	0	0	成功 (s_tchk9b)
0	1	0	1	失败
0	1	1	0	失败
0	1	1	1	失败
1	0	0	0	失败
1	0	0	1	失败
1	0	1	0	失败
1	0	1	1	失败
1	1	0	0	失败
1	1	0	1	失败
1	1	1	0	失败
1	1	1	1	失败

以下的 Verilog 代码实例可以产生激励来满足这 256 种不同情形。

```
module ctc_complex;

logic irdyn, trdyn, devseln, stopn, framen;
integer i, j;
logic clk;
logic [3:0] test_expr;

assign irdyn = test_expr[3];
assign trdyn = test_expr[2];
assign devseln = test_expr[1];
assign stopn = test_expr[0];

initial begin
clk = 1'b0; test_expr = 4'd15;
```

```

repeat(2) @ (posedge clk);

for(i=1; i<17; i++) // timing loop
begin
  for(j=0; j<16; j++) // logical loop
  begin
    framen = 1'b0;
    repeat(1) @ (posedge clk);
    test_expr = 4'b1101;
    repeat(i) @ (posedge clk);
    test_expr = j;
    repeat(1) @ (posedge clk);
    framen = 1'b1;
    repeat(1) @ (posedge clk);
    test_expr = 4'b1111;
    repeat(1) @ (posedge clk);
  end
end

repeat(2) @ (posedge clk);
$finish;
end

initial forever clk = #25 ~clk;

endmodule

```

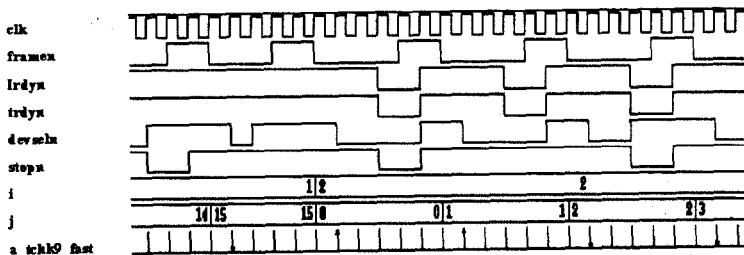


图 7-13 PCI 检验验证

注意，这里时序约束在外层循环，而逻辑约束在内层循环。该检验器的测试代码应该能生成 211 个失败和 45 个成功。这就确保了检验器对所有可能输入条件都响应正确。图 7-13 显示了该测试的波形图实例。

7.4 检验器检验小结

- 双信号关系的可能数目能够在 SVA 域内呈指数级增长。
- 通过探讨双信号之间的 3 种关系(逻辑、时序和重复)，我们写了 56 种断言语句。当 SVA 定义的信号数目增加时，可能的断言语句将呈指数级增加。
- 我们需要一种自动化的方法来验证这些断言。通过简单的 Verilog 语言，我们能够创建一些十分有效的激励生成方法来对这些断言进行彻底测试。
- 同样的激励生成方法被应用于实际的 PCI 检验器来验证其正确性。
- 当断言越来越复杂，可以利用约束性随机 testbench 的高级特性来有效地检验这些检验器。而自我检验机制能够用来分析断言验证的结果。
- 虽然没有检验检验器的自动方法，用户仍然可以把它当作任何一种设计模块用 testbench 来验证。
- 如果没有“检验器检验”的方法，用户无法知道设计是否工作还是失败，也不知道检验器是否写得正确。
- 该过程需要在验证前期投入时间，但能够为验证后期节省大量时间。