国外大学优秀教材 —— 微电子类系列（翻译版）

高级ASIC芯片综合

—— 使用Synopsys® Design Compiler™ Physical Compiler™ 和PrimeTime®

（第2版）

Himanshu Bhatnagar 著
张文俊 译


清华大学出版社
北京
Springer

图书在版编目(CIP)数据

高级 ASIC 芯片综合:使用 Synopsys® Design Compiler™ Physical Compiler™ 和 PrimeTime® (第 2 版)/巴特纳格尔(Bhatnagar, H.)著;张文俊译. —北京:清华大学出版社, 2007.6
书名原文: Advanced ASIC Chip Synthesis: Using Synopsys® Design Compiler™ Physical Compiler™ and PrimeTime®
(国外大学优秀教材——微电子类系列(翻译版))
ISBN 978-7-302-14881-4

Ⅰ. 高… Ⅱ. ①巴… ②张… Ⅲ. 集成电路—电路设计—高等学校—教材
Ⅳ. TN402

中国版本图书馆 CIP 数据核字(2007)第 038435 号

责任编辑: 王敏稚
责任校对: 梁 毅
责任印制: 李红英
出版发行: 清华大学出版社 **地 址:** 北京清华大学学研大厦 A 座
http://www.tup.com.cn **邮 编:** 100084
c—service@tup.tsinghua.edu.cn
社 总 机: 010-62770175 **邮购热线:** 010-62786544
投稿咨询: 010-62772015 **客户服务:** 010-62776969
印 刷 者: 北京四季青印刷厂
装 订 者: 三河市溧源装订厂
经 销: 全国新华书店
开 本: 152×228 **印 张:** 16.5 **字 数:** 273 千字
版 次: 2007 年 6 月第 1 版 **印 次:** 2007 年 11 月第2 次印刷
印 数: 2501～4500
定 价: 33.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770177 转 3103 产品编号:008310—01

译 者 序

微电子技术是21世纪信息时代的关键技术之一，是计算机技术、自动控制、通信技术的基础。现在集成电路的设计和应用已经渗透到各个工程技术领域，几乎每个工程学科的学生和工程技术人员都要了解集成电路设计方面的知识，并且越来越多的人参与到集成电路设计工作当中。因此十分需要一本介绍集成电路设计流程，并且能够指导读者进行设计的教材。美国 Conexant 系统公司 Himanshu Bhatnagar 所著的《高级 ASIC 芯片综合》就是这样一本将技术和工具结合在一起的非常实用的教材。

Himanshu Bhatnagar 是ASIC设计方面的专家。他在使用Synopsys和其他 EDA 工具厂商提供的最新的高性能工具来定义下一代 ASIC 设计流程的方法学方面具有较深的造诣。

本书共 13 章。主要包括三部分内容：第一部分介绍了一个完整的 ASIC 设计流程，并且给出了实际应用的例子。第二部分详细讨论了在 ASIC 设计过程中所采用的各种技术，主要包括 HDL 的编码风格、综合和优化、动态仿真、形式验证、DFT 扫描插入、links to layout（后端集成）、物理综合和静态时序分析等。第三部分详细介绍了 Synopsys 工具所采用的技术和特点以及使用方法，包括 Synopsys 基本的工艺库、Design Compiler、Synopsys Test Compiler、Physical Compiler、PrimeTime 等。贯穿本书各部分的一个主线，就是在讨论设计和相关技术问题的过程中，给读者提供一个应用 Synopsys 工具解决相关问题的方法。这是本书的一大特点，也使本书的实用性更强。

参加本书翻译的还有冯东博士、马君教授、孙月老师和张昕老师，

并得到了清华大学出版社编辑的热心指导和大力支持，得到了清华大学微电子学研究所领导和教师的关心，在此一并深表谢意。

最后，由于译者水平有限，文中定有不当或欠妥之处，望读者批评指正。

张文俊

2006 年 11 月于清华园

序

半导体产业的发展有一条业已证明的轨迹，就是 IC 设计的特征尺寸急速减小。作为一个群体，我们的术语知识很快被指代集成度的术语所充满。例如在 20 世纪 80 年代中期，我们将在一个芯片 (chip) 上集成 1000 个以上晶体管称为“大规模集成”（LSI），然而一两年之后，集成度上升到更先进的 10000 到 100000，称为“超大规模集成”(VLSI)。人们对用新的术语重新定义我们的设计成果做出了少许努力，例如“甚大规模集成”（ULSI）就很幸运地在历史记录中留下一笔。越来越多的远见卓识者认识到摩尔定律的逻辑推演最后必然导致没有恰当的英语词汇来描述最高级的集成度。然而无论如何我们不能拒绝改变头脑中原来对于设计任务的概念，这个概念是基于 20 世纪 90 年代早期到中期新造的短语“片上系统”（System on a Chip），以达到这样一种认识：这种程度的集成度不仅仅允许复杂电子元件的开发，还允许自载信息处理系统。但是我们再一次发现自己在和实际情况斗争，即三四年前“系统”所指代的东西今天却极少能够填满一个中等引脚的计数器件的压焊块环。

因此，我们不应该对此感到惊讶，即设计团体中的一些人正在意识到重新思考和定义设计规则的必要性。设计规则限制了现代 IC 的设计任务。将注意力放在晶体管或者功能的集成度上的做法正在被取代，作为设计产品的规则，设计团队的注意力集中到我们最宝贵的商品和时间上。对于商品和时间，今天的设计任务由机会窗口来定义，机会窗口和设计产出相关，一般是一个不超过 12～18 个月的周期。因此设计团队重点关注一些工具和技术，这些工具和技术可以将设计效率提高到这样的水平：布满硅片的晶体管、功能和子系统模块可以在给定的时间内真正设计出来并表现出特色。当我们的字典开始采用如“六个月规模集成度”（SMSI），“十二个月规模集成度”（TMSI）

这样的术语来定义集成规模时，我们不该太吃惊。

本书完全体现了这种努力，它探讨并描述了工具和技术的集合，这些工具和技术可以戏剧性地减少完成设计任务所需的时间并向市场推出 IC 产品。作者 Bhatnagar 先生收集了一系列现代的生产性 IC 设计工具，并揭示了这些工具应用在进一步的设计流程中的方法。这些技术对设计者构成了挑战，设计者必须超越这样的线性高层次设计流程，即利用 HDL 语言作设计描述，综合得到门级和晶体管级实现以及时序分析。本书探讨了实用技术，通过这些技术，更多的信息不是在设计流程中更快地引入就是更快地反馈，以达到减少反复次数和降低复杂性的目的。最终在这些技术的帮助下，我们在更短的时间内得到了质量更好的产品。

当今的半导体行业是在一个压缩时间、过度竞争的环境下进行的。为了在这样的环境下有效竞争，建议每一个设计者和设计团队，注意不断改善他们的上市时间设计规则。本书将满足从事 VLSI 设计的高年级学生以及经验丰富的业界人士在此方面的需求。

Mr F. Matthew Rhodes
Sr. Vice President and General Manager
Personal Computing Division
Conexant Systems, Inc.

前　言

本书第 2 版描述了使用 Synopsys 工具进行 ASIC 芯片综合、物理综合、形式验证和静态时序分析的最新概念和技术，同时针对超深亚微米（VDSM）工艺的完整 ASIC 设计流程的设计方法学进行了深入的探讨。

本书的重点是使用 Synopsys 工具解决各种 VDSM 问题的实际应用。读者将详细地了解一个有效的处理复杂的亚微米 ASIC 设计方法学，其重点是 HDL 的编码风格、综合和优化、动态仿真、形式验证、DFT 扫描插入、links to layout（后端集成）、物理综合和静态时序分析。在每个步骤中，确定了设计流程中的每一部分的问题，并详细描述了解决的方法。此外，包括与时钟树综合和后端集成（links to layout）等对版图至关重要的问题也进行了详细的讨论。而且为了获得最佳的综合解决方案，本书还对 Synopsys 基本的工艺库、HDL 编码风格进行了深入的探讨。

本书的读者对象是 ASIC 设计工程师和正在学习关于 ASIC 芯片综合以及 DFT 技术的 VLSI 高级课程的硕士研究生。

本书并不是 Synopsys 参考手册的替代品，但希望对 ASIC 设计流程感兴趣的人们有所帮助。对于那些没有版图设计能力，或虽拥有工艺库，但仍依赖于外部厂商进行后端集成和最终器件制造的人们（或公司），本书是非常有用的。由于 VDSM 技术的各种问题，对于传统的向外部厂商提供网表的设计，本书提供了可供选择的设计方法。对于设计者使用来自不同厂商的各种设计工具时所遇到的一些常见问题，本书也提供了解决方案。

在本书的这一版中，已将所有的命令更新到 Design Compiler 的 Tcl 版本，并更改了所有命令以反映 Synopsys 工具的最新版本（2000.11-SP1）。

各章概述

第 1 章概括介绍使用 Synopsys 工具进行 ASIC 设计的不同阶段，并对从概念到芯片的定案下单（tap-out）的完整的设计流程作了简要介绍。本章对不了解芯片设计和集成的完整过程，但想学习完整的 ASIC 设计流程的人们是非常有用的。

第 2 章概略说明第 1 章所描述的 ASIC 设计流程的实际应用。本章可作为初学者的指南，也可作为使用 Synopsys 工具的高级用户的参考。没有使用过 Synopsys 工具进行综合的用户应当跳过本章，在阅读过本书的其他章节后再回来阅读本章。

第 3 章详细地介绍综合的基本概念。这些概念包括在后面章节要用到的综合术语。本章也介绍了 Synopsys 工具及使用环境的基本信息。除了描述每个工具的用途及其设置之外，本章还着重定义了 Design Compiler 使用的对象、变量、属性和编译指示。

第 4 章介绍 Synopsys 工艺库的基础知识。工艺库包含各种不同驱动强度的单元，只要工艺库包含不同驱动强度的多种单元，设计者通常不关心工艺库的全部细节。然而，一个丰富的库常常决定综合的质量。因此，本章试图从设计者的角度描述 Synopsys 工艺库。其重点放在延迟计算方法和可使设计人员更好地利用工艺库的其他技巧，进而提高综合设计的质量。

适当的划分和良好的编码风格是获得好品质的基础。第 5 章对正确划分的各种技术提供指导，以获得最佳的解决方案。此外，在本章的大量实例中，介绍了 HDL 编码风格，并且指导设计人员通过怎样的编码来完成设计以得到较快的逻辑和最小的面积。

第 6 章介绍了用于综合与优化的 Design Compiler 命令。本章对使用 Synopsys 工具的初学者和高级用户都有所帮助。本章重点考虑与理想情况偏离的实际应用，即“不是所有的设计或设计者都能遵循 Synopsys 建议”。本章还列举了大量实例来帮助用户实际使用这些命令。

第 7 章讨论优化技术以满足时序和面积的需求，并比较了 Design Compiler 新旧版本之间差异。本章重点介绍了在 DC 中使用的新的优化技术“TNS”。同时也详细分析了进行逻辑优化的各种方法。此外，对不同的编译策略的优缺点也进行了详细的讨论。

DFT 技术正在逐渐成为 ASIC 设计工程师必须掌握的技术。第 8 章首先概述了目前使用的各种 DFT 技术，然后详细描述了如何使用 Synopsys Test Compiler 将系统设计为可扫描的，并介绍了通过 Design Compiler 插入扫描的命令。为了减少设计中 DFT 扫描插入所引起的相关问题，本章给出了一系列的指导。

第 9 章讨论了 Design Compiler 的后端集成（links to layout）的特性。它描述了前端和后端设计工具之间的接口。本章也为布图后的优化设计提供了不同策略，包括基于优化技术的布局。此外，还介绍了时钟树插入和将时钟树移入到 Design Compiler 所产生的相关问题，并对常见的问题给出了各种解决办法。本章对那些没有版图工具并且愿意学习布局布线过程以及全芯片集成技术的设计者（和公司）是非常有价值的。

Physical Compiler 极大地改变了传统的综合方法。第 10 章详细地描述了这一流程，并描述了使用 Physical Compiler 获得最佳结果的各种方法。为了理解 Physical Compiler 流程，在读本章之前，建议读者阅读关于传统流程的所有章节（尤其是第 9 章）。这将会帮助理解本章与传统流程相关的内容。本章还给出了各种示例脚本来说明这个新工具的用法。

第 11 章标题为“SDF 生成——动态时序仿真”，描述利用 Design Compiler 或 PrimeTime 生成 SDF 文件的过程。本章首先讲述了 SDF 格式句法，然后详细讨论布图前后的 SDF 生成过程。此外，一些创新的思想和建议还可以帮助设计者进行成功的仿真。对那些为了验证设计的功能，喜欢使用动态仿真方法胜过形式验证技术的设计人员，本章也是非常有用的。

第 12 章介绍使用 PrimeTime 进行静态的时序分析的基本方法，包括 PrimeTime 使用的 Tcl 语言的简要介绍。本章还介绍了用于进行静态时序分析以及在出现时序违例（timing violation）时便于设计人员进行调试的 PrimeTime 命令。

对于特定的设计而言，制造工作芯片的关键是能够成功地完成静态时序分析。在整个设计流程中进行静态时序分析是最重要的一步，也是许多设计人员将设计提交给 ASIC 厂商的交付标准。第 13 章讨论了使用 PrimeTime 进行静态时序分析的一些基础和高级的问题，对 ASIC 设计流程中布图前后的 PrimeTime 的用法作了详细的说明。此外，还给出了大量关于各种情况的分析报告和建议的示例。对那些想

要从传统的动态仿真方法转到静态分析设计的设计人员，本章是有益的；同时，对那些想要使用 PrimeTime 对设计进行深入分析的读者，本章也是很有帮助的。

本书使用的约定

所有的 Synopsys 命令以“Arial”字体表示。这包括综合和时序分析脚本的所有示例。

命令行提示以“Courier New”字体表示。例如：

```
dc_shell>和 pt_shell>
```

对于一些带有选项值的命令，选项值放在 < 和 >之间。通常在使用命令之前，需要替换这些值。例如：

```
set_false_path-from<from list> - to <to list>
```

“\” 字符用来表示续行，而“|”字符表示“OR”函数。例如：

```
compile –map_effort low|medium|high\
        -incremental_mapping
```

只要有可能，关键字都用*斜体*。主题或要点需要强调用下划线或**粗体**字来突出表示。

致　谢

在本书的出版过程中，许多人都投入了时间和精力，作者在此向他们表示衷心的感谢。没有他们的帮助，本书是不可能完成的。

首先，特别感谢我的家庭给予了我长久的支持和鼓励，这是我完成这项工作的动力。我的妻子 Nivedita，她容忍我夜里和周末的写作活动，同时花费大量的时间校对原稿并改正我的“工程师英语”。没有她的帮助和理解，我不可能完成这项工作。

我要感谢我的主管 Anil Mankar，他在工作中给予我大力的支持，使我能够完成本书。他的精神支持和创造性的建议对我有很大的帮助。我也非常感谢我在科胜讯（Conexant）的同事。Khosrow Golshan 帮助我设计了书的封面。对后端设计流程，他也提出许多重要的建议。Young Lee，Hoat Nguyen，Vinson Chua，Hien Truong，Songhua Xu，Chilan Nguyen，Randy Kolar，Steve Schulz，Richard Ward，Sameer Rao，Chih-Shun Ding 和 Ravi Ranjan 等都花费了他们的宝贵时间对原稿进行了讨论。

极其幸运的是我遇到一位杰出的评审专家，Kelvin F. Poole（南卡罗来纳州克莱姆森大学）。当写作本书的时候，我得到了认识许多年的普尔博士的指导。他不止逐字地校对了全部文稿（我肯定他咬紧了牙关！），而且还提出了许多有价值的建议，使本书内容更加翔实。谢谢你，普尔博士。

向在 Synopsys 公司工作的 Bill Mullen，Ahsan Bootehsaz，Steve Meier，Russ Segal，Juergen Froessl，Elisabeth Moselery，Kelly Conklin，Bob Moussavi 和 Amanda Hsiao 表示我深深的感谢。他们参与了原稿的讨论，并且提出了许多有价值的建议。Synopsys 公司的 Julie Liedtke 和 Bryn Ekroot 帮助我写必需的商标信息。 特别感谢 Broadcom 公司的 Jeff Echtenkamp，Heratch Avakian，Chung-Jue Chen 和 Chin-Sieh

Lee，他们提供了有价值的反馈和深入的讨论。感谢总是积极给予反馈的 Kameshwar Rao（顾问），Jean-Claude Marin（法国 ST 微电子），Tapan Mohanti（Centillium 通信），Dr. Sudhir Aggarwal（Philips 半导体）和 Abu Horaira（Intel 公司），非常感激他们的鼓励与支持。

在 SNUG 2000 期间，我认识了 Cliff Cummings（Sunburst 设计的总裁兼顾问），作为 Verilog RTL 编码和综合的专家，Cliff 在这一行业中是非常著名的。我请求他帮助我评审本书的某些章节，我非常感谢他提出了许多有价值的建议。我已将这些建议写入了本书的第 5 章。

本书第 2 版的写作比预期花费了更长的时间，主要的原因是引入了 Physical Compiler。我对这本书的内容进行了扩展，但是不能写一些不成熟的内容。在写作本书的过程中，Kluwer 科学出版社的 Carl Harris 理解并且支持我。即使我延迟书的出版，他也表示理解，使我非常感激。

最后，“感谢父母对我永无止境的信任”。

Himanshu Bhatnagar
Conexant Systems, INC.
Newport Beach, California

作者简介

Himanshu Bhatnagar 是位于美国加州新港海滩（Newport Beach）的科胜讯（Conexant）系统公司 ASIC 设计小组的领导。科胜讯系统公司是世界上最大的专门提供半导体通信电子产品的公司。Himanshu 在使用 Synopsys 和其他 EDA 工具厂商提供的最新的高性能工具来定义下一代的 ASIC 设计流程方法学方面具有较深的造诣。

在加入科胜讯之前，Himanshu 在新加坡 ST 微电子工作，该公司的总部位于法国 Grenoble。他在斯旺西（Swansea）大学（英国威尔士）获得了电子与计算机科学专业的学士学位。在克莱姆森大学（美国南卡罗来纳州）获得了超大规模集成电路设计专业的硕士学位。

目　录

第 1 章
ASIC 设计方法学

由于深亚微米半导体几何尺寸的缩小，使用传统的芯片设计方法进行设计已经变得越来越困难。此外要在相同的芯片面积下将晶体管数目增加，会使设计工作变得更加艰巨。而且，在“上市时间”这一理念的压力下，芯片设计周期需要保持原有的水平甚至要不断地缩短。为了克服这些问题，就需要发展新的方法和工具来完善 ASIC 设计方法学。

本章主要讲述在亚微米领域芯片设计各个阶段的最新技术，同时对设计流程的各种改进技术也进行了讨论。

在本书的上一版之后，Synopsys 引进了一个称为 Physical Compiler 的工具。在这个工具中，综合和布局联系得更加紧密，从而使传统的设计流程有了重大的改变。本章强调新技术的重要性，并且解释在设计流程中使用这些新技术缩短整个设计周期以取得最大效益的必要性。考虑到这个工具对 IC 设计界还相当新，而且还未 100% 地被 ASIC 设计界所接受，因此传统的设计流程和新的设计流程都在本书中有所讨论。

本章重点论述基于 ASIC 设计流程方法学，从 RTL 编码到最终定案下单（tape-out）的全部综合过程，对于传统和基于 Physical Compiler 的流程也都进行了讨论。

1.1 传统的设计流程

传统的 ASIC 设计流程（如图 1-1 所示）大致包含如下步骤，后面章节将详细地叙述与综合相关的主题。

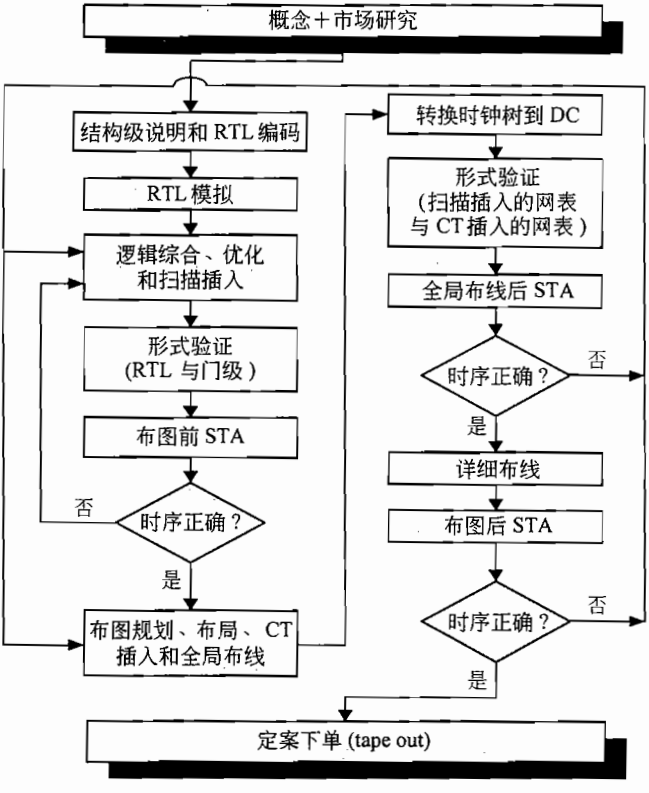


图 1-1 传统的 ASIC 设计流程

1. 结构及电学特性规范。

2. HDL 中的 RTL 编码。

3. 为包含存储单元的设计插入 DFT memory BIST。

4. 为验证设计功能，进行详尽的动态仿真。

5. 设计环境设置，包括将使用的工艺库及其他环境属性。

6. 使用 Design Compiler 对具有扫描插入（和可选择的 JTAG）的设计进行约束和综合设计。

7. 使用 Design Compiler 的内建静态时序分析机进行模块级静态时序分析。

8. 设计的形式验证，使用 Formality 将 RTL 和综合后的网表进行对比。

9. 使用 PrimeTime 进行整个设计布图前的静态时序分析。

10. 对布图工具进行时序约束的前标注。

11. 具有时序驱动单元布局、时钟树插入和全局布线的初始布局划分。

12. 将时钟树转换到驻留在 Design Compiler 中的原始设计（网表）。

13. 在 Design Compiler 中进行设计的布局优化。

14. 使用 Formality 在综合网表和时钟树插入的网表之间进行形式验证。

15. 在全局布线后（第 11 步）从版图提取估计的延时。

16. 从全局布线得到的估计时间数据反标注到 PrimeTime。

17. 使用在全局布线后提取的估计延时数据在 PrimeTime 中进行静态时序分析。

18. 设计的详细布局。

19. 提取来自详细布局设计的实际时间延迟。

20. 实际提取时间数据反标注到 PrimeTime。

21. 使用 PrimeTime 进行布图后的静态时序分析。

22. 布图后的门级功能仿真（如果需要）。

23. 在 LVS 和 DRC 验证之后定案下单（tape out）。

图 1-1 说明了上面讨论的典型的 ASIC 设计流程。缩略词 STA 和 CT 分别表示静态时序分析和时钟树，DC 表示 Design Compiler。

1.1.1　规范和 RTL 编码

芯片设计起始于由市场得出的想法概念，由这些想法进而得出结构和电学特性规范。结构规范定义了芯片的功能并划分为一些能够处理的模块，而电学特性规范通过时序信息定义模块之间的关系。

下一阶段涉及这些规范的实现。过去是利用来自单元库的元件，由人工来完成电路图，这一过程耗时并且不利于设计复用。为了克服这个问题，开发了硬件描述语言 （HDL）。正如它的名称，使用 HDL 可以对设计的功能进行编码。目前有两种常用的硬件描述语言：Verilog 和 VHDL，两种语言具有相同的功能，并都有各自的优缺点。

设计可以用三个抽象层次来表示：行为级、寄存器传输级（RTL）和结构级。行为级编码是在一个较高层次上的抽象，它主要用来将结构规范转换为一个可以仿真的代码，行为编码是完成设计目标的最基本方法。而 RTL 编码描述并推断结构元件和它们之间的连接，这类

编码用来描述设计的功能并且可综合成一个结构网表，这个网表由目标库中的元件和它们相应的连接组成，非常类似于电路图的实现方法。

不论是用 Verilog 还是 VHDL 或是两者混用，设计都是使用 RTL 编码。如果需要，它也能被划分为许多小的模块来形成一个层次，由一个顶层模块连接所有的下层模块。

注：Synopsys 最近引入了 Behavior Compiler，因此有能力进行综合行为级编码。由于这是与本书无关的主题，所以本书只讨论与 RTL 有关的综合。

1.1.2 动态仿真

下一步是通过仿真 RTL 代码以检查设计的功能。所有目前可得到的仿真器都能够仿真行为级及 RTL 级编码。此外，它们也用来仿真映射后的门级设计。

图 1-2 描述一个准备仿真的由测试基环绕的分块设计。这一个测试基通常用行为级 HDL 描述而实际的设计使用 RTL 编码。

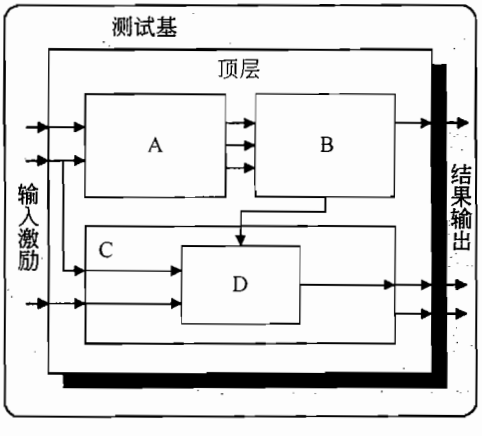


图 1-2 层次设计范例

尽管市场上有一些仿真器有能力仿真一个混合 HDL 设计，但通常仿真器还是只能识别一种语言（Verilog 或 VHDL）。

测试基的目的是为设计提供必需的激励。测试执行的次数和测试基的性能对完成整个设计测试是重要的，这就是为什么完善的测试基对设计极其重要。在 RTL 的仿真期间，不考虑元件（或门）时序。

因此，为了最小化 RTL 仿真和综合后门级仿真之间的差异，通常时序单元的 RTL 源码包含延迟。

1.1.3　约束、综合和扫描插入

在过去很长的一段时间里，硬件描述语言用来进行逻辑验证，设计者需要手工将 HDL 转换为电路图并描述元件间互连来产生一个门级网表。由于综合工具的出现，已经不用手工完成这项工作，工具已经取代了它并且可以完成 RTL 级到门级网表的转换，这一过程被称为综合。

Synopsys 的 DC 是目前在 ASIC 工业中最流行的综合工具和实际标准。

综合设计是一个迭代过程，从为设计中的每个模块定义时序约束开始，这些时序约束定义了每个信号与某个特定模块的时钟输入的相互联系。除了约束外，还需要定义综合环境的文件，这个环境文件详细说明了工艺单元库和 DC 在综合过程中使用的其他相关信息。

DC 读取设计的 RTL 代码并且使用时序约束，综合 RTL 代码到结构级，从而产生一个映射后的门级网表，这个过程如图 1-3 所示。

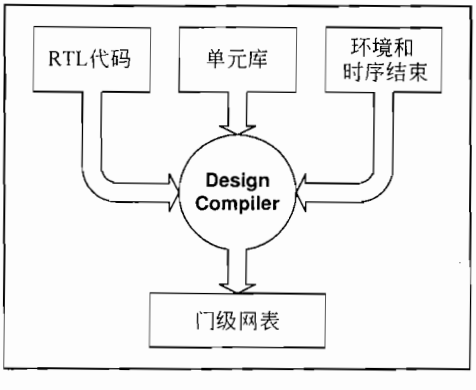


图 1-3　Design Compiler 的输入和输出

通常对于小模块的设计，DC 用内置的静态时序分析来报告综合后设计的时序信息。DC 尽可能地优化设计以满足指定的时序约束。如果时序需求不满足，就必须做进一步的工作。

目前大多数的设计都结合了可测试性设计（DFT）逻辑以便在芯片制造后用来测试它们的功能。DFT 包括逻辑和存储 BIST（内置自

测试)、扫描逻辑和边界扫描逻辑（JTAG）等。

基于控制逻辑并且在综合前合并到设计中的可综合 RTL 构成逻辑存储 BIST。目前在市场上有一些可以用来产生 BIST 控制和周边逻辑的工具，遗憾的是，Synopsys 没有提供这方面的工具。

使用 DC 的测试预编译特性可以执行扫描插入，在扫描链链接它们之前，这一个过程可直接将 RTL 映射到扫描触发器（scan-flop）。使用这一特性的优点是它可以使DC在综合时考虑扫描触发器的时序能力。与对应的非扫描触发器（或正常触发器）相比，扫描触发器可产生不同的延时，因此这项技术是重要的。

JTAG 或边界扫描主要用于测试带有芯片的板连接，JTAG 控制器和周边的逻辑也可以直接由 DC 产生。

1.1.4 形式验证

形式验证的概念对 ASIC 设计领域是相当新的。形式验证技术使用数学方法来确认一个设计，无需考虑工艺因素，如时序和物理效应的影响，它们通过与参考设计对比来检查一个设计的逻辑功能。

许多 EDA 工具厂商都开发了形式验证工具，然而直到最近 Synopsys 才向市场推出形式验证工具，称为 Formality。

形式验证方法和动态仿真之间的主要不同是，形式验证技术通过证明两个设计的结构和功能是逻辑等价的来验证设计；动态仿真方法只能检查那些敏感路径，所以不可能找出其他出现的问题。此外，与动态仿真相比，形式验证方法所需的运行时间是可以忽略不计的。

在设计流程中，形式验证的目标是要验证 RTL 与 RTL、门级网表与 RTL 代码或两个门级网表之间的对应关系是否正确。

RTL 对 RTL 的验证是用来确认新的 RTL 与原来的 RTL 在功能上是否一致。这通常是为了适应因增加的附加性能而需经常修改设计的情况。当这些特性增加到 RTL 源的时候，总是有改变原有正确功能的危险。为了避免这种情况，可以在原来的 RTL 和新的 RTL 之间执行形式验证，以检查原有功能的有效性。

RTL 对门级的验证是用来确定 DC 综合的逻辑是正确的。由于通过动态仿真来验证 RTL 功能正确，所以在 RTL 和有扫描插入的门级网表之间做形式验证，能保证门级也有相同的功能。在这种情况下，如果我们要使用动态仿真方法验证门级，就会花费较长的时间 （数

天和数星期，取决于设计的大小）来验证设计。与动态仿真比较，形式 RTL 方法只用几个小时就可完成一个类似的验证。

最后是门级网表对门级网表的验证。这也是验证过程的一个重要的步骤，因为它主要用来确认版图输入信息与版图输出信息。显然从版图得出的是时钟树插入的网表（平面或层次化的）。这意味着进入布图工具的原来的网表已被修改了。形式验证技术用来确认修改后的网表与原来的网表是逻辑等价的。

1.1.5　使用 PrimeTime 进行静态时序分析

正如前文提到的，DC 可以作模块级静态时序分析。虽然使用上述方法可以完成芯片级静态时序分析，但是这里推荐使用 PrimeTime。PrimeTime 是由 Synopsys 提供的、可独立运行的、能够提高品质的静态时序分析工具，它能够非常快地完成整个芯片级设计的静态时序分析。它提供一个 Tcl 接口，该接口为设计分析和调试提供了一个强大的环境。

从某种程度上说，在整个 ASIC 设计过程中静态时序分析是最重要的步骤。静态时序分析允许用户详细分析设计的所有关键路径并且给出一个有条理的报告。此外，该报告也可包含其他调试信息，如扇出能力或每个线网的容性负载。

对布图（layout）前后的门级网表进行静态时序分析。在布图前，PrimeTime 使用由库指定的线载模型估计线网延时，在这一过程中，先前输入到 DC 的时序约束同样也输入到 PrimeTime 中并详细说明主要的输入输出信号和时钟的关系。如果对于所有关键路径的时序是可接受的，则由 PrimeTime 或 DC 可以得到一个约束文件，目的是为了预标注到布图工具。这个约束文件以 SDF 格式详细描述在布图工具中使用的每个逻辑组之间的时序，以便完成单元的时序驱动布局。

在布图后，实际提取的延迟被反标注到 PrimeTime 以提供真实的延迟计算，这些延迟由连线电容和互连 RC 延迟所组成。

与综合类似，静态时序分析也是一个迭代过程，它与芯片布局布线（placement and routing）的联系非常紧密，这个操作通常需要执行许多次才能满足时序需求。

1.1.6 布局、布线和验证

正如本节标题所示，布图工具完成布局和布线。完成这一步骤有许多方法，本节只讨论与综合相关的内容。

布图规划（floorplan）和布局的质量比实际的布线更重要。最佳的单元布局，不但加速最终的布线，而且在满足时序约束和减少阻塞方面也产生非常好的效果。如前所述，约束文件用来进行时序驱动布局。时序驱动布局方法能够使布图工具根据单元之间的时序关键程度放置单元。

在单元布局后，时钟树通过布图工具插入设计。时钟树插入是可选择的并且只依赖设计需求和用户的偏爱。用户可选择使用较传统的方法对时钟网络进行布线，例如，为了要减少总时间延迟和时钟的倾斜使用 fishbone/spine 结构的时钟网络。当工艺尺寸缩小，由于互连线电阻的增加（从而 RC 延迟），spine 方法实现变得更困难。因此本节（和本书）重点放在时钟树综合方法方面。

在这个阶段，一个附加的步骤对完成时钟树插入是必需的。如上所述，在单元布局后，布图工具将时钟树插入设计，因此，从 DC 产生的最初网表（并且输入到布图工具）缺少时钟树信息（整个时钟树网络，包括缓冲器和线网），所以，时钟树一定要插入到原有的网表中并进行形式验证。一些布图工具通过提供直接的接口给 DC 来完成这一步骤。第 9 章将介绍一些方法，包括传统的和非传统的实现方法。为了简化，假设已经将时钟树插入到原有的网表。

布图工具完成布线通常由两步组成：全局布线和详细布线。在布局后，设计进行全局布线以确定布局的质量和提供估计延迟与在详细布线后实际延时值的近似程度。如果单元布局不是最佳的，与单元布局相比，完成全局布线将花费比较长的时间。不好的布局也会影响整个设计的时序。因此，为使布图综合迭代的次数最小并且改进布局质量，在全局布线后，从版图提取时序信息。尽管这些延迟数据不如在详细布线后提取的数据准确，但它们确实提供了布线后的时序信息。这些估计的延迟被反标注到 PrimeTime 进行分析，并且只有当时序关系满足后，余下的进程才被允许执行。

详细布线是布图工具进行的最后步骤，在详细布线完成后，提取

芯片的实际时间延迟并且输入到 PrimeTime 进行分析。

这些步骤是反复的并且依赖于该设计的时序余量。如果设计不能满足时序需求，在布图后的优化要在进行布图另一次迭代之前运行。如果设计通过静态时序分析，在定案下单（tape-out）前它还要进行 LVS（版图对原理图）和 DRC（设计规则检查）。

值得注意的是，上面讨论的所有步骤也能应用于层次化的布局布线，换言之，在对子模块布局以形成最终版图之前，可以对每个子模块重复上述步骤。

1.1.7　工程改变命令

这个步骤是正常设计流程的一个例外，不要与通常的设计周期混淆。因此，这一个步骤在后面章节将不再加以介绍。

许多设计者认为，工程改变命令（ECO）是在 ASIC 设计流程的每个最后阶段对网表进行必需的改变。例如，当在最后阶段（比如定案下单）后，在设计中遇到的硬件隐藏错误并且需要通过对设计的一个小部分进行再布线来完成一个金属掩模改变时，就运行 ECO。

由于 ECO 的执行结果只对芯片的小部分有作用，为了避免干扰芯片其余部分的布局和布线，因此保留芯片其余部分的时序，只有受影响的部分被修改。可以通过只影响芯片包含的备用的门电路或只对一金属层布线来完成此项操作。这一过程称为金属掩模改变。

通常，这个过程只在对整个芯片（或一个模块，如果做层次化的布局和布线）的修改少于 10%时才执行。如果修改需要超过 10%，那么最好重复整个的过程和重新对整个芯片（或模块）布线。

最新版的 DC 包含 ECO 编译器。它利用数学算法（也应用于形式验证技术），自动地实现需求的变化。利用 ECO 编译器提供设计者手动修改插入网表的方法，可将芯片的完成时间减到最小。

一些布图工具已经将 ECO 算法合并到它们的工具里。布图工具的优点就是不受设计的层次化边界限制。同样，布图工具也受益于知道备用的单元（通常由设计者引入到设计中）放置位置，因此能以最接近备用单元的位置为目标以完成所需的 ECO 改变并获得最小化的布线。

1.2 Physical Compiler 流程

由于半导体几何尺寸的缩小，基于线载模型的综合结果变得很不准确和不可预知。由 Synopsys 开发的一个新工具 Physical Compiler，通过在一个公共引擎中集成综合和布局解决了上述问题，从而可以避免基于线载模型的延迟计算。

Physical Compiler 基本的设计流程一般包含下面列出的一些步骤。图 1-4 所示为与下面描述的设计流程相关的流程图。由于一些步骤在传统流程和基于 Physical Compiler 的流程中都存在，所以这里只说明与 Physical Compiler 流程相关的一些步骤。

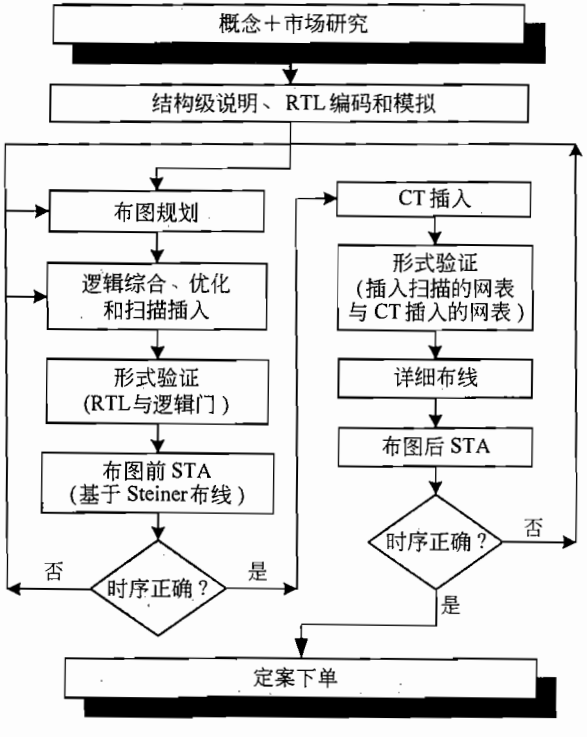


图 1-4 Physical Compiler 流程

1. 设计环境设置，包括要使用的工艺库和物理库以及其他的环境属性。

2. 布图规划设计。

3. 使用 Physical Compiler 进行约束、综合（使用扫描插入）及设计布局。

4. 使用 PrimeTime 进行布图前静态时序分析（基于布局而不是线载模型的时间延迟数据）。

5. 使用 Formality 对设计的 RTL 与综合后的网表进行形式验证。

6. 将网表和布局信息移植到布图工具上。

7. 使用布图工具在设计中插入时钟树。

8. 在插入时钟树的网表与原有的扫描插入的网表之间进行形式验证。

9. 使用布图工具进行详细布线。

10. 从详细布线设计提取实际时间延迟。

11. 将实际提取的数据反标注到 PrimeTime。

12. 使用 PrimeTime 进行布图后的静态时序分析。

13. 利用布图后的时序进行设计的门级功能仿真（如果想要）。

14. 在 LVS 和 DRC 验证后定案下单（tape out）。

1.2.1 物理综合

传统的综合方法是以线载模型为基础的，而线载模型的基本性质是以它们的扇出为基础建立的。换言之，单元的延迟计算是基于单元驱动的扇出数目进行的，这种方法适用于较大的几何尺寸（大于 0.35μm），而不适合于较小的几何尺寸，基于扇出进行延时计算时，线的电阻决定了单元延迟时间，因而是不可靠的并且完全不可预知。

为了解决上述问题，Synopsys 近来以 Physical Compiler（因而称为 PhyC）的形式引入了物理综合概念。PhyC 除保留了原有 DC 的功能外还增加了一些功能，所以 PhyC 是 DC 的一个超集。

PhyC 不使用线载模型，并将以扇出为基础的延迟计算改为以布局为基础的延迟计算。换言之，综合和优化是以单元布局为基础的，并且将扫描链重排功能包含在目前的版本 PhyC（2000.11）中，它确实是极其强大的有用工具。

图 1-4 用一种非常一般的形式来描述这种实现方法。PhyC 有两种模式：RTL 到门级布局（rtl2pg）或门到门级布局（g2pg）。对于前一种模式，PhyC 的输入是 RTL、布图规划信息及包含逻辑库和物理

库的必要设置。PhyC 的输出是 PDEF3.0 格式的结构网表和布局后的门信息。第二种模式（g2pg）在基于布图规划信息的条件下，对一个已存在的门级网表进行优化。在这种情况下，PhyC 的输入替代 RTL 的门级网表，而其余的设置和输入输出文件保持一致。

值得注意的是，目前的 PhyC（2000.11 版）版本没有综合时钟树的能力。Synopsys 最近发布了一个加入到 PhyC 中的有效的时钟树编译工具。没有这个工具的用户只能使用他们的布图工具将时钟树插入到设计数据库中。

进行有效的综合需要完成许多步骤。这些将在后面的章节讨论。然而目前出于解释设计流程的目的，上述论述已足够了。

1.3 小结

本章对包括由超深亚微米（VDSM）技术组成的最新工具和工艺的 ASIC 设计流程进行了介绍。设计流程从定义规范开始，以物理版图结束。重点放在逻辑和物理综合等相关主题。

物理综合是本章引入的一个新概念，它可应用于设计流程中以缩短芯片的设计周期。物理综合的重要性是它能够得到一个较好的对延迟的估计并且能缩短上市时间。

第 2 章
入门指南 静态时序分析与综合

本章针对 Synopsys 工具的初级用户和高级用户。建议以前没有使用 Synopsys 工具的初学者先跳过这一章，在阅读了本书其他部分后再返回这一章。有少量综合经验的初级用户可把本章当作学习使用 Synopsys 工具进行 ASIC 设计流程的起点，高级用户可将本章作为参考。

本章很少甚至没有解释 Synopsys 命令（将在以后章节中进行介绍）。其重点是介绍第 1 章中所描述的以 Synopsys 综合为核心的 ASIC 设计流程的实际应用。这有助于读者将理论概念同实际应用联系起来。

为了描述基于传统和 Physical Compiler 的流程，保留了同前者相关的所有的脚本（将命令改变为 Tcl 格式）。此外还有一个小节介绍了 Physical compiler 流程，用户可挑选最适合自己设计需求的流程。

虽然第 1 章为了强调形式验证方法而跳过门级仿真，但是许多设计人员不愿采用前一种方法。因此，本章也介绍了从 DC 生成用于仿真的 SDF，除了使用 Formality 的形式验证的应用外，本章还介绍了使用 Primetime（PT）的静态时序分析。

可用任何方法进行综合和优化，这只取决于你最喜欢用的和最方便使用的方法。本章采用了最为 Synopsys 用户们广泛使用的一种方法，你可以相当容易地掌握这种方法以满足自己的要求。

为了清晰容易地进行解释，所有的示例和脚本都采用自底向上的编译方法（稍后介绍），这同本章中的综合过程相关。还必须注意的

是，整个 ASIC 流程是反复迭代的过程，用户不应当认为本章介绍的过程满足所有设计。后续章节将详细讨论每个论题，用户可加以调整以适应自己的设计和方法。

2.1 设计示例

学习这个主题的最好方法就是用一个设计示例走一遍整个设计流程。下面所示的是用 Verilog HDL 包含一个层次的 tap 控制器的设计。

tap_controller.v
    tap_bypass.v
    tap_instruction.v
    tap_state.v

顶层设计为 *tap_controller*，其中例化了三个模块，分别为 *tap_bypass*、*tap_instruction* 和 *tap_state*。这一设计包括一个 30MHz 的时钟“tck”和一个复位“trst”。设计时序规范指定所有与“tck”相关的输入信号所需的建立时间为 10ns，而保持时间为 0ns。此外，所有输出信号必须比时钟延迟 10ns。

用于这一设计的工艺为 0.25 微米。由于工艺偏差，为了得到更加精确的结果，使用了两个分别对应最差情况和最佳情况参数的 Synopsys 标准单元工艺库。这两个库分别为 *ex25_worst.db* 和 *ex25_best.db*，还有一个相应的包含电路图表示的符号库 *ex25.sdb*。在 *ex25_worst.db* 库中定义工作条件名为 WORST，而在 *ex25_best.db* 库中定义工作条件名为 BEST。

假设设计的功能已经通过了 RTL 级动态仿真验证。

2.2 初始设置

下一步是综合设计，也就是说要把设计映射到工艺库中的门。在我们开始进行综合前，必须生成如下设置文件：

a）.synopsys_dc.setup 文件，用于 DC 和 PhyC。

b）.synopsys_pt.setup 文件，用于 PT。

第一个文件是用于逻辑综合和物理综合的 DC 和 PhyC 设置文件，而第二个文件同 PT 相关并且定义了静态时序分析所要求的设置。

假设库被保存在目录—/usr/golden/library/std_cells/中，创建具有如下内容的两个文件：

DC & PhyC .synopsys_dc.setup 文件

```
set search_path          [list . /usr/golden/library/std_cells]
set target_library       [list ex25_worst.db]
set link_library         [list {*} ex25_worst.db ex25_best.db]
set symbol_library       [list ex25.sdb]
set physical_library     [list ex25_worst.pdb]

define_name_rules BORG -allowed {A-Za-z0-9_}          \
    -first_restricted "_" -last_restricted "_"        \
    -max_length 30                                    \
    -map {{"*cell*","mycell"},{"*-return","myreturn"}}

set bus_naming_style                    %s[%d]
set verilogout_no_tri                   true
set verilogout_show_unconnected_pins    true
set test_default_scan_style             multiplexed_flip_flop
```

PT .synopsys_pt.setup 文件

```
set search_path          [list . /usr/golden/library/std_cells]
set link_library         [list {*} ex25_worst.db ex25_best.db]
```

2.3　传统流程

下面的步骤描述了传统流程。这里 DC 用于逻辑综合，而布图工具处理包括布局和布线等其他后端问题。

2.3.1　布图前的步骤

以下几个小节介绍在布图前的阶段涉及到的步骤，包括带扫描插

入的逻辑综合、静态时序分析、用于执行门级功能仿真的 SDF 生成以及最终的 RTL 源码与综合后网表之间形式验证的全过程。

2.3.1.1 综合

布图前逻辑综合包括对最大的建立时间优化设计、采用统计的线载模型及 *ex25_worst.db* 工艺库中的最差情况工作条件。为了最大化建立时间，通过定义建立时间的时钟不确定性来约束设计。一般说来，为了最小化综合与布图之间的迭代次数，10%过度约束通常是足够的。

在最初的综合后，如果检测到保持时间（hold-time）违例，它们应该在布图前加以改正。这也有助于减少综合与布图之间的反复。然而，可取的方法是在布图后用反标注的真实延迟来修正次要的保持时间违例。

在本指南中，我们假定存在次要的保持时间违例，因而这些违例可在布图后优化中加以改正。修正保持时间违例涉及将从版图提取的延迟反标注到 DC 中。此外，保持时间修正需要利用 *ex25_best.db* 库中的最佳情况工作条件的用法。

子模块的通用综合脚本

```
set active_design tap_bypass

analyze –format verilog $active_design.v
elaborate $active_design

current_design $active_design
link

uniquify

set_wire_load_model –name SMALL
set_wire_load_mode top
set_operating_conditions WORST

create_clock –period 33 –waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
```

```
set_clock_uncertainty –setup 3.0 [get_clocks tck]
set_clock_transition 0.1 [get_clocks tck]
set_dont_touch_network [list tck trst]

set_driving_cell –cell BUFF1X –pin Z [all_inputs]
set_drive 0 [list tck trst]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

set_max_area 0

set_fix_multiple_port_nets –buffer_constants –all
compile –scan

check_test

remove_unconnected_ports [find –hierarchy cell {"*"}]

change_names –h –rules BORG

set_dont_touch current_design

write –hierarchy –output $active_design.db
write –format verilog –hierarchy        \
                        –output $active_design.sv
```

上述脚本包括用户定义的变量 *active_design*，它对要综合的模块进行命名。这一变量应用于整个脚本，从而使得脚本的其余部分通用。通过将 *active_design* 重新定义到其他子模块（*tap_instruction* 和 *tap_state*），可使同样的脚本用于综合这些子模块。用户可将相同的概念应用到时钟名、时钟周期等以参数化脚本。

假设已经成功综合了三个子模块，命名为 *tap_bypass*,、*tap_instruction* 和 *tap_state*。这一综合过程没有将扫描链串起来，只是将触发器直接映射到扫描触发器。除了在读取 *tap_controller.v* 文件之前必须包括子模块的已映射的“db”文件，我们可以将相同的综合脚本应用于顶层综合，此外，还需要进行扫描插入来把扫描链串起来。为更好地建模互连线，线载模型需要改变为 enclosed。由于子模块

包括 dont_touch 属性，顶层综合不会跨越层次边界进行优化，并可能会违反设计规则约束。为了去除这些违例，必须删除子模块的 dont_touch 属性以重新进行综合/优化。

顶层 DFT 扫描插入是从子模块中除去 dont_touch 属性的另一个原因。这是因为如果子模块包含 dont_touch 属性，就不能在顶层进行 DFT 扫描插入。以下脚本通过执行带有扫描使能的初始综合来说明此过程，在再次编译（compile -only_design_rule）设计前，去除了所有子模块的 dont_touch 属性。

顶层的综合脚本

```
set active_design tap_controller

set sub_modules {tap_bypass tap_instruction tap_state}

foreach module $sub_modules{
        set syn_db $module.db
        read_db syn_db
}

analyze –format verilog $active_design.v
elaborate $active_design

current_design $active_design
link

uniquify

set_wire_load_model –name LARGE
set_wire_load_mode enclosed
set_operating_conditions WORST

create_clock –period 33 –waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_uncertainty –setup 3.0 [get_clocks tck]
set_clock_transition 0.1 [get_clocks tck]
set_dont_touch_network [list tck trst]

set_driving_cell –cell BUFF1X –pin Z [all_inputs]
```

```
set_drive 0 [list tck trst]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

set_max_area 0

set_fix_multiple_port_nets –all –buffer_constants
compile –scan

remove_attribute [find –hierarchy design {"*"}] dont_touch

current_design $active_design
uniquify

check_test
create_test_patterns –sample 10
preview_scan
insert_scan
check_test

compile –noly_design_rule

remove_unconnected_ports [find –hierarchy cell {"*"}]
change_names –hierarchy –rules BORG

set_dont_touch current_design

write –hierarchy –output $active_design.db
write –format verilog –hierarchy        \
                        –output $active_design.sv
```

2.3.1.2　使用 PrimeTime 进行静态时序分析

在综合成功后，对得到的网表必须进行分析以检查时序违例。时序违例可包括建立和/或保持时间违例。

设计在综合过程中以最大化建立时间（setup-time）为重点，因而即使存在建立时间违例，也很少遇到。然而，保持时间违例通常会出现在这一阶段，这是由于数据到达时序单元输入的速度太快（在时

序单元锁存数据前，其值改变），从而违反了保持时间的要求。

如果设计不满足建立时间要求，你别无选择，只能重新综合设计，并以违例路径为目标进行进一步地优化。这涉及到组合违例路径或对具有违例的整个子模块加紧约束。然而如果设计不满足保持时间要求，可在布图阶段前也可推迟到布图后再修正这些违例。许多设计人员对次要的保持时间违例倾向于采用后一种方法（这里也采用此种方法），因为布图前综合和时序分析采用的是统计线载模型，并且在布图前修正保持时间违例可导致同一路径在布图后建立时间违例。必须注意的是，总的保持时间违例应在布图前阶段加以修正，以减少布图后的保持时间修正的数目。

布图前建立时间分析的 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db

current_design $active_design

set_wire_load_model –name large
set_wire_load_mode top

set_operating_conditions WORST

set_load 50.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

create_clock –period 33 –waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 3.0 –setup [get_clocks tck]

set_input_delay 20.0 –clock tck [all_inputs]
set_output_delay 10.0 –clock tck [all_outputs]

report_constraint –all_violators

report_timing –to [all_registers –data_pins]
report_timing –to [all_outputs]
```

```
write_sdf --context verilog –output $active_design.sdf
```

上述 PT 脚本对 *tap_controller* 设计进行静态时序分析。注意到，由于在布图前阶段未插入时钟树，上例中时钟延迟时间和反转时间是固定的，因而有必要设置某一数量的延迟，用于近似与时钟树相关的最终延迟。因为时钟网络具有高扇出，所以还要指定时钟反转时间。高扇出意味着时钟网络驱动许多触发器，每个都有某一数量的引脚电容。这使时钟的输入上升时间更加缓慢。为避免 PT 基于触发器的缓慢输入上升值而产生延迟计算错误，应固定时钟的反转时间（也是最终时钟树的一个近似）。

下面所示为在布图前阶段进行保持时间分析的脚本。为检查保持时间违例，必须通过使用如 *ex25_best.db* 库中指定的最佳情况工作条件进行分析。此外，在 *report_timing* 命令中指定了一个额外的参数（*-delay_type min*），如下所示：

布图前保持时间分析的 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db

current_design $active_design

set_wire_load large
set_wire_load_mode top

set_operating_conditions BEST

set_load 50.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

create_clock –period 33 –waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 0.2 –hold [get_clocks tck]

set_input_delay 0.0 –clock tck [all_inputs]
set_output_delay 0.0 –clock tck [all_outputs]
```

```
report_constraint –all_violators

report_timing –to [all_registers –data_pins]        \
                                        -delay_type min
report_timing –to [all_outputs] –delay_type min

write –sdf –context verilog –output $active_design.sdf
```

2.3.1.3 SDF 生成

为了进行时序仿真，需要将 SDF 文件进行反标注。静态时序分析使用的是 PT，因而如前一脚本所示，用 PT 生成 SDF 文件是稳妥的。而一些设计人员感到用 DC 生成 SDF 文件比较方便，因此本节用 DC 来生成 SDF。

此外，在用于进行设计的时序仿真前，根据设计所得的 SDF 文件需要进行一些调整。调整的原因将在第 11 章详细介绍。

以下脚本用于生成 *tap_controller* 设计的布图前 SDF，该 SDF 文件用于设计的动态时序仿真。此外，这个脚本也生成时序约束文件，虽然该文件也是 SDF 格式，但它主要是为了使用传统方式进行时序驱动布图，而向布图工具中前向标注时序信息。

布图前 SDF 生成的 DC 脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

set_wire_load_model LARGE
set_wire_load_mode top
set_operating_conditions WORST

create_clock –period 33 –waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 3.0 –setup [get_clocks tck]
```

```
set_driving_cell –cell BUFF1X –pin Z [all_inputs]
set_drive 0 [list tck trst]

set_load 50 [all_outputs]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

write_sdf –output $active_design.sdf
write_constraints –format sdf –cover_design   \
                  –output constraints.sdf
```

2.3.1.4　布图规划和布线

布图规划包括单元的布局和时钟树综合，这两步都在布图工具中完成。布局可包括单元的时序驱动布局，这是通过向布图工具标注 *constraints.sdf* 文件（由 DC 生成）完成的。该文件由包含单元到单元时序信息的路径延迟组成。布图工具使用这一信息以时序为主要准则来放置单元，也就是说，布图工具将时序关键单元彼此靠近放置以最小化路径延迟。

假定设计已经进行了布图规划，布图工具也已将时钟树插入到设计中。时钟树插入修改了现有的设计结构，换言之，布图工具中的网表与DC中的初始网表不同。这是因为布图工具中的设计包含时钟树，而 DC 中的初始设计不包含该信息，因此，应将时钟树信息读入 DC 或 PT 中的设计。新的网表（包含时钟树信息）需要对照原来的网表进行形式验证，以确定时钟树转移没有破坏原先逻辑的功能。将时钟树信息转移到设计中的各种方法将在第 9 章中详细讨论。出于简化的原因，此处假定 *tap_controller* 设计中有时钟树信息。

现在可以对设计进行布线了。泛泛地讲，布线分两步完成：全局布线和详细布线。在全局布线中，布线器将版图表面划分为一些分开的区域并进行没有实际放置几何连线的点到点“松弛”布线。最后的布线是由详细布线器完成的，会放置几何连线并在区域内布线。第 9 章将全面介绍这些布线类型。此处假设设计已经进行了全局布线。

下一步是从全局布线后的设计中提取估计的寄生电容和 RC 延迟。这一步减少了综合与布图间的迭代时间，主要是因为单元布局和

全局布线需要的时间比全芯片的详细布线少得多。然而，如果以最小的阻塞优化放置单元，详细布线也很快。在任一情况下，全局布线后提取的延迟（虽然是估计值）提供一个更加接近实际延迟的更快的方法，该延迟值在详细布线后从版图中提取。

使用如下脚本，将估计值反标注到 PT 设计中以进行建立和保持时间的静态时序分析。

使用估计延迟进行建立时间分析的 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db

current_design $active_design
set_operating_conditions WORST

set_load 50.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

source capacitance.pt      #估计的寄生电容
read_sdf rc_delays.sdf     #估计的 RC 延迟

create_clock –period 33 –waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.5 –setup [get_clocks tck]

set_input_delay 20.0 –clock tck [all_inputs]
set_output_delay 10.0 –clock tck [all_outputs]

report_constraint –all_violators

report_timing –to [all_registers –data_pins]
report_timing –to [all_outputs]
```

使用估计延迟进行保持时间分析的 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db
```

```
current_design $active_design

set_operating_conditions BEST

set_load 20.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

source capacitance.pt          #估计的寄生电容
read_sdf rc_delays.sdf         #估计的 RC 延迟

create_clock –period 33 –waveform [0 16.5] tck

set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.05 –hold [get_clocks tck]

set_input_delay 0.0 –clock tck   [all_inputs]
set_output_delay 0.0 –clock tck   [all_outputs]

report_constraint –all_violators

report_timing –to [all_registers  –data_pins]      \
                                  –delay_type min
report_timing –to [all_outputs]   –delay_type min
```

上述脚本反标注了 *capacitance.pt* 和 *rc_delays.sdf* 文件，*capacitance.pt* 文件包括了使用 *set_load* 格式的每条连线的容性负载，而文件 *rc_delays.sdf* 则包含了单个连线的点到点的互连 RC 延迟。DC（和 PT）根据设计中每个单元的输出连线负载和输入反转来计算单元延迟，将在第 9 章中详细解释使用这种方法的原因。

如果设计不满足建立时间要求，可通过调整约束来重新进行综合设计或布图规划设计。如果设计不满足保持时间的要求，根据违例的程度，您可以决定继续使用至最后一步设计的详细布线或通过调整约束重新优化设计。

如果想要重新综合，那么包含实际单元位置的布图规划（放置）信息应反标注到 DC 中。需要这一步是因为到目前为止，DC 都不了解单元的物理放置信息，通过将放置信息标注到 DC 中，会极大地提高 DC 中设计的布图后优化。布图工具产生 PDEF 格式的物理信息，

DC 使用如下命令读取此信息：

```
read_clusters <file name in PDEF format>
```

完成这一任务的脚本除了标注数据和增量编译设计外，与最初的综合脚本是相似的，如下所示：

设计的增量综合脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

source capacitance.dc                    /*估计的寄生电容*/
read_timing –f sdf rc_delays.sdf         /*估计的 RC 延迟*/
read_clusters clusters.pdef              /*物理信息*/

create_wire_load –hierarchy          \
                 –percentile 80      \
                 –output cwlm.txt

create_clock –period 33 –waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 3.0 –setup   [get_clocks tck]

set_dont_touch_network [list tck trst]

set_driving_cell –cell BUFF1X –pin Z [all_inputs]
set_drive 0 [list tck trst]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

set_max_area 0

set_fix_multiple_port_nets –all –buffer_constants
```

```
reoptimize_design –in_place

write –hierarchy –output $active_design.db
write –format verilog –hierarchy        \
                        –output $active_design.sv
```

上述脚本中的 create_wire_load 命令为 tap_controller 设计生成一个自定义的线载模型。最初综合使用的是工艺库中现有的不针对特定设计的线载模型，因此，为了下一次综合迭代取得更高的精确度，应使用针对特定设计的自定义线载模型。

下面命令用于更新DC内存中的工艺库以反映新的自定义线载模型，如

```
dc_shell -t > update_lib ex25_worst.db cwlm.txt
```

假设设计已被再次分析并且满足建立和保持时间的要求。下一步就是对设计进行详细布线。这是一个取决于布图的特性，因而在此不作讨论。

2.3.2　布图后步骤

布图后步骤包括验证带反标注实际延迟的设计时序、设计的功能仿真及最后 LVS 和 DRC 的执行。

假设以最小的阻塞和面积，完成了设计的布线。应从已完成的版图中提取实际的寄生电容和 RC 连线延迟。依据布图工具和提取的类型，RC 连线延迟用 SDF 格式写出，而设计中的每一条连线的寄生信息由一串 set_load 命令产生。此外，如果已经进行了层次化的布局和布线，则也应生成 PDEF 格式的单元的实际布局。

2.3.2.1　使用 PrimeTime 进行布图后静态时序分析

布图后的第一步就是用实际延迟对设计进行静态时序分析。同布局后相似，布线后时序分析也使用同样的命令，只是这次向设计中反标注的是实际延迟。

设计的时序主要依赖于时间的延迟和扭斜。因此一个明智的选择是，在进行全设计分析前分析时钟的扭斜。Synopsys 通过网上在线支持（SolvNET）提供了一个有用的 Tcl 脚本，您可以下载这一脚本并

在执行前对其进行分析。假设时钟延迟（latency）和扭斜（skew）是受限的，下一步就是对设计进行静态时序分析，通过下面的脚本检查建立和保持时间违例（如果存在）。

使用实际延迟的建立时间分析 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db

current_design $active_design

set_operating_conditions WORST

set_load 50.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

source capacitance.pt                    #实际的寄生电容
read_sdf rc_delays.sdf                   #实际的 RC 延迟
read_parasitics clock_info_wrst.spf      #用于时钟等

create_clock –period 33 –waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.5 –setup [get_clocks tck]

set_input_delay 20.0 –clock tck [all_inputs]
set_output_delay 10.0 –clock tck [all_outputs]

report_constraint –all_violators

report_timing –to [all_registers –data_pins]
report_timing –to [all_outputs]
```

实际延迟的保持时间分析 PT 脚本

```
set active_design tap_controller

read_db –netlist_only $active_design.db

current_design $active_design
```

```
set_operating_conditions BEST

source capacitance.pt                          #实际的寄生电容
read_sdf rc_delays.sdf                         #实际的 RC 延迟
read_parasitics clock_info_best.spf            #用于时钟等

set_load 50.0 [all_outputs]
set_driving_cell –cell BUFF1X –pin Z [all_inputs]

create_clock –period 33 –waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.05 –hold [get_clocks tck]

set_input_delay 0.0 –clock tck   [all_inputs]
set_output_delay 0.0 –clock tck   [all_outputs]

report_constraint –all_violators
report_timing –to [all_registers –data_pins]       \
                                    –delay_type min
report_timing –to [all_outputs] –delay_type min
```

2.3.2.2　布图优化

对设计进行布图后优化（PLO）以改进或修正时序要求。DC 通过原位优化（IPO）特性提供了几种修正时序违例的方法。与以前一样，DC 也利用实际布局信息进行基于位置的优化（LBO）。在本例中，我们将使用调整单元尺寸和插入缓冲器的 IPO 特性修正保持时间违例。

2.3.2.2.1　修正保持时间

综合设计以满足最大化建立时间的要求。每一步（综合后、全局布线后）都要验证时序，因此布线后的设计很可能满足建立时间的要求。然而，设计的某些部分在各终点有可能不满足保持时间的要求。

如果设计不能达到保持时间要求，就应当通过增加缓冲器以延迟故障信号相对于时钟的到达时间从而修正违例。假定本设计在多个终点都没能达到保持时间的要求。

有多种方法可以用来修正保持时间违例，第 9 章中将详细讨论这些方法。在本例中，我们将用 dc_shell -t 命令来修正保持时间违例，

如下所示：

修正保持时间违例的 DC 脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

source capacitance.dc                    /*实际的寄生电容*/

read_timing –f sdf rc_delays.sdf         /*实际的 RC 延迟*/
read_clusters clusters.pdef              /*物理层次信息*/

create_clock –period 33 –waveform {0 16.5} tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty –hold 0.05 tck

set_dont_touch_network [list tck trst]

set_driving_cell –cell BUFF1X –pin Z [all_inputs]
set_drive 0 [list tck trst]

set_input_delay –min 0.0 –clock tck –max [all_inputs]
set_output_delay –min 0.0 –clock tck –max [all_outputs]

set_fix_hold tck                         /*修正保持时间违例 w.r.t tck */

reoptimize_design –in_place

write –hierarchy –output $active_design.db
write –format verilog –hierarchy         \
                        –output $active_design.sv
```

上述脚本中，set_fix_hold 命令用来指引 DC 修正与时钟 *tck* 相关的保持时间违例。命令 reoptimize_design 的参数-in_place 为 IPO 命令，它受第 9 章中将介绍到的多种变量的控制。使用这些变量，DC 插入或调整门的大小以修正保持时间违例。LBO 变量有助于在正确

位置插入缓冲器，这样能够降低由其他一些违例路径出发的逻辑路径所产生的影响。

在 IPO 之后，应使用前述的布图后 PT 脚本再次通过 PT 分析设计以确保修正违例。

一旦设计通过了所有的时序要求，如果需要，可为仿真生成布图后 SDF（由 PT 或 DC 生成）。我们可通过下面提供的脚本在 DC 中生成最坏情况下的布图后 SDF，可用相似的脚本生成最佳情况下的 SDF。显然，要从 DC 生成最佳情况下的 SDF，需要反标注由布图工具提取的最佳情况数值。这只与布图工具和方法有关。

最差情况下布图后 SDF 生成的 DC 脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

set_operating_conditions WORST

source capacitance.dc                    /*实际的寄生电容*/
read_timing rc_delays.sdf                /*实际的 RC 延迟*/

create_clock –period 33 –waveform {0 16.5} tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty –setup   0.5 [get_clocks tck]

set_driving_cell –cell BUFF1X –pin Z [all_inputs]
set_drive 0 [list tck trst]

set_load 50 [all_outputs]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

write_sdf –output $active_design.sdf
```

建议在 RTL 源和最终网表之间再次进行形式验证，以检查在整个过程中偶然产生的错误。最后，在定案下单前，设计需进行 LVS

和 DRC 检查。

2.4 Physical Compiler 流程

Physical Compiler（PhyC）流程提供了组合综合和布局的集成方法，它不使用线载模型的传统方法，因而最小化了布图前综合和布图后延迟间的差异。

鉴于这一方法的重要性及其新颖性，将用完整的一章介绍这一流程，在此不再赘述，第 10 章介绍了与这一流程相关的所有脚本。为了理解如何使用 Physical Compiler 以及在我们已经习惯使用的综合和布图方法中的定位，建议读者阅读第 10 章学习使用 Physical Compiler。

2.5 小结

本章以指南的方式着重介绍了 ASIC 设计方法学的实际应用，使用一个设计示例由开始到结束来引导读者，并且每一步都提供有简要的解释和相关的脚本。

本章一开始介绍了设置 Synopsys 环境的基础知识和设计示例的技术规范，接下来的几小节分为布图前、布图规划和布线及最终的布图后步骤。

布图前的步骤包括设计的初始综合和扫描插入、静态时序分析及为动态仿真生成 SDF。为了最小化综合与布图之间的迭代次数，布图规划和布线方法着重介绍了单元的布局，并特别强调了向 DC 反标注从全局布线后的设计中提取的延迟估计值。最后一节使用布图后优化的技巧修正保持时间违例并生成仿真用的最终 SDF。

本章也介绍了使用 Synopsys Formality 的形式验证方法。该部分不包含任何脚本，但读者可注意到形式验证技术的用途及其应用范围。

最后，讨论了物理综合方法，它不需要线载模型并集成了综合和布局。第 10 章将提供同这一方法相关的所有脚本。

第 3 章
基本概念

本章涵盖了使用 Synopsys 系列工具进行综合的相关的基本概念。这些概念向读者介绍了贯穿后面章节的综合术语。这些术语为 Synopsys 综合和静态时序分析提供了一个必要的框架。

虽然本章颇具参考价值，但对于那些已经熟知 Synopsys 术语的高级 Synopsys 工具使用者来说，本章可以跳过。

3.1 Synopsys 产品

本节简要描述与本书相关的 Synopsys 全部相关产品。

a）Library Compiler

b）Design Compiler 和 Design Vision

c）Physical Compiler

d）PrimeTime

e）DFT Compiler

f）Formality

Library Compiler

任何 ASIC 设计的核心都包含一组逻辑单元的工艺库，库可以包含每个单元的功能描述、时序、面积及其他相关信息。在将它转换为可被所有 Synopsys 应用程序使用的格式之前，Library Compiler（LC）会分析文本信息的完整性与正确性。

通过在 UNIX 命令行输入 lc_shell 来启动 Library Compiler，并且 LC 的所有功能在 dc_shell 中也可以使用。

Design Compiler 和 Design Vision

Synopsys Design Compiler（DC）和 Design Vision（DV）构成一套功能强大的逻辑综合工具，根据设计规范和时序约束，提供最佳的门级综合网表。除了高层次综合能力，它还包含一个静态时序分析引擎，提供 FPGA 综合和 links-to-layout（LTL）解决方案。

Design Compiler 是 Synopsys 综合工具的命令行接口，并且通过在 UNIX 命令行里键入 dc_shell 或 dc_shell -t 来调用。dc_shell 是基于 Synopsys 自身语言的原有格式，而 dc_shell -t 使用的是标准 Tcl 语言。本书中只着重于 DC 的 Tcl 版本，因为它与其他 Synopsys 工具具有共性，如 PrimeTime。

Design vision 是 DC 的图形化前端版本，通过键入 design_vision 启动。Design vision 也支持电路原理图的生成，并且通过点对点高亮显示来分析关键路径。

虽然，初学者最初可能更喜欢使用 DV，在较为熟悉 Synopsys 命令之后他们很快就会转向使用 DC。

Physical Compiler

Physical Compiler（或 PhyC）是 Synopsys 的一个新工具，它是 DC 的一个超集。除了包含 DC 全部的综合和优化功能外，它还提供了根据设计的时序和/或面积约束同时优化放置单元的能力。

PhyC 由输入 psyn_shell 调用，也可由输入 psyn_gui 来使用单独的 GUI 版本。虽然它比较慢，但 psyn_gui 为用户提供了在设计的逻辑表示和电路原理图间进行转换的功能。

必须注意的是，PhyC 是 DC 的超集，可在 psyn_shell 中使用所有的 dc_shell 命令。反之却不正确，即不能在 dc_shell 中使用 pysn_shell 命令。

PrimeTime

PrimeTime（PT）是 Synopsys 的签约级、全芯片、门级静态时序分析工具。另外，通常为满足大型设计的需要，它也允许具有较强的建模能力。

PT 比 DC 的内部静态时序分析引擎要快，它也提供了文本及图形的增强分析能力。同 Synopsys 的其他工具相比，这一工具是基于

Tcl 语言的，因此这种语言提供的强大特性提升了设计的分析和调试能力。

PT 是一个独立运行的工具，它既可以命令方式调用，也可以图形方式调用。在 UNIX 窗口中使用命令行方式键入 pt_shell，而使用图形方式键入 Primetime。

DFT Compiler

DFT Compiler（DFTC）是包含在 DC 全套工具中的 Synopsys 测试插入工具。它用于向设计插入 DFT 特性，如扫描插入和边界扫描。所有的 DFTC 命令直接从 dc_shell 或 psyn_shell 调用。

Formality

Formality 是 Synopsys 的形式验证工具，或更准确地讲，它是逻辑等价检测工具。这一工具具有增强的图形调试能力。它包含待验证的逻辑的原理图表示和标注在原理图上作为可能的错误逻辑指示的可视建议。此外，它还提供可能修正设计的建议。

3.2　综合环境

与大多 EDA 产品相同，Synopsys 工具需要一个设置文件以指定工艺库位置和用于综合的其他参数。Synopsys 还定义了自己的格式来存储和处理这些信息。本节着重介绍这些细节。

3.2.1　启动文件

所有 DC 和 PhyC 系列工具都有一个共同的启动文件“.synopsys_dc.setup”，而 PT 需要一个单独的启动文件“.synopsys_pt.setup”，这些文件为 Tcl 格式并且包括工艺库的路径信息和其他环境变量。

PhyC、DC 和 PT 的默认启动文件位于 Synopsys 安装目录中，随着这些工具的启动而自动加载。这些默认文件不包含同设计相关的数据，它们的功能就是加载 Synopsys 与工艺无关的库和其他参数；用户可在启动文件中指定与设计相关的数据，在启动过程中，这些工具以如下顺序读取文件：

1. Synopsys 安装目录。
2. 用户主目录。
3. 项目工作目录。

项目工作目录中的启动文件的设置会覆盖在主目录中指定的相同设置，同样主目录中的设置会覆盖 Synopsys 安装目录中的设置，也就是说，项目工作目录中指定的设置要比其他的设置优先。

用户应在方便的地方保存这些文件，建议在工作目录中保存同设计相关的启动文件。

DC 需要的最少信息为 search_path、target_library、link_library 和 symbol_library。除了与 DC 相关的设置外，PhyC 还需要 physical_library 信息。PT 只需要 search_path 和 link_library 信息。典型启动文件如例 3.1 所示。

例 3.1　设置文件

PhyC & DC .synopsys_dc.setup 文件

```
set search_path          [list . /usr/golden/library/std_cells \
                             / usr/golden/library/pads]
set target_library       [list std_cells_lib.db]
set physical_library     [list std_cells_lib.pdb pad_lib.pdb]
set link_library         [list {*} std_cells_lib.db pad_lib.db]
set symbol_library       [list std_cells_lib.sdb pad_lib.sdb]
```

PT .synopsys_pt.setup 文件

```
set search_path          [list . /usr/golden/library/std_cells \
                             / usr/golden/library/pads]
set link_path            [list {*} std_cells_lib.db pad_lib.db]
```

3.2.2　系统库变量

现在有必要解释 target_library 和 link_library 系统变量间的区别。target_library 指定工艺库的名称，其单元对应于设计人员想让 DC 推断出并最终映射到的库单元。link_library 定义其库单元只用于参考的库名称，也就是 DC 不是使用 link_library 中的单元进行推断。例如，你可能指定一个标准单元工艺库作为 target_library，而在 link_library 列表中则指定压焊块工艺库名称和所有其他的宏单元（RAM、ROM 等）。这就意味着用户将用标准单元库中的单元来综合设计，而同时

连接设计中例化的压焊块和宏单元。如果 target_library 列表中包括了压焊块库，那么 DC 可用压焊块来综合核心逻辑。

如例 3.1 所示，link_library 列表中应包含目标库名，这在 DC 中读取门级网表时是很重要的。如果连接库列表中不包含目标库名，DC 就不能连接网表中的已映射的单元，在这种情况下，DC 生成表示其不能解析网表中单元的警告。

target_library 和 link_library 系统变量允许设计人员更好地控制单元的映射。这些变量也提供了一种有用的方法将门级网表从一种工艺重新映射到另一种工艺。在这种情况下，link_library 可包括旧的工艺库名，而 target_library 可包含新的工艺库名。可通过 dc_shell 中的 translate 命令来完成重新映射。

symbol_library 系统变量是包含工艺库中的单元图形表示的库名称。当使用图形化前端工具 DA 时，它用于表示这些门电路原理图。符号库以扩展名“sdb”为标识。如果设置文件忽略了这一变量，DA 将会使用一个名为“generic.sdb”的通用符号库来生成原理图。通常，所有库厂商提供的工艺库都包含一个相应的符号库。工艺库和符号库间的单元名和引脚名必须准确地匹配。单元中的任何不匹配都可引起 DA 拒绝符号库中的单元而使用通用库中的单元。

如果使用 psyn_shell，除了指定逻辑库，也需要指定物理库。这些库中包含 PhyC 所需的物理信息（如单元的物理尺寸、方位、层信息等）。物理库以扩展名“pdb”为标识，并由 physical_library 系统变量访问，如上例所示。

必须注意的是，DC 使用变量 link_library，而 PT 使用 link_path，除了名称和格式不同外，这两个变量的应用是相同的。既然 PT 是一个门级静态时序分析器，它只用于结构化门级网表。因而，PT 不使用变量 target_library。

3.3　对象、变量和属性

Synopsys 支持大量的对象、变量及属性以使综合过程更有效。使用这些要素，设计人员可以编写出强大的脚本以使综合过程自动化。因此，设计人员有必要熟悉一下这些术语。

3.3.1 设计对象

DC 将设计对象（design object）分为八种不同类型，分别如下：

- **设计**（Design）：它对应完成一定逻辑功能的电路描述。这一设计可以是单独的，也可以包含其他的子设计。虽然子设计是设计的一部分，但 Synopsys 将其视为另一个设计。
- **单元**（Cell）：它是设计中子设计的例化名称。在 Synopsys 术语中，单元与实例无任何区别，均视作单元。
- **引用**（Reference）：它是单元或实例所引用的原始设计的定义。例如，网表中的叶单元必须引用包含单元功能描述的连接库。同样地，例化的子设计（Synopsys 称为单元）必须引用包含例化子设计功能描述的设计。
- **端口**（Port）：它们是设计的原始输入、输出或 IO。
- **引脚**（Pin）：它对应于设计中单元的输入、输出或 IO（注意端口和引脚的区别）。
- **连线**（Net）：这些为信号名，也就是通过将端口接到引脚或将引脚彼此连接起来而使设计连在一起的导线。
- **时钟**（Clock）：作为时钟源的端口或引脚。它可以在库内部确认，也可以用 dc_shell-t 命令来确认。
- **库**（Library）：对应于设计综合的目标或引用连接的特定工艺单元的集合。

3.3.2 变量

变量（variable）是 DC 用作存储信息的占位符，此信息与调整最终网表的指令相关，也可以包含用于综合过程自动化的用户自定义值。DC 预先定义好了一些变量，设计人员可以用这些变量得到存储在这些变量中的当前值。例如，变量“bus_naming_style”对于 DC 有特殊的意义，而“captain_picard”对于 DC 则无意义，后者用于脚本的任意用户定义值。

所有变量都是全局变量且只存在于运行过程中，它们不与设计数据库保存在一起。一旦 dc_shell-t 运行结束，变量值就不存在了。大

多 dc_shell-t 变量都有一个默认值，它们在运行一开始就得到这个值。例如，下一变量用“SYNOPSYS_UNCONNECTED_”作为其默认值，你可以将其改为“MY_DANGLE_”，则 DC 会写出以“MY_DANGLE_”作为所有未连接连线前缀的 verilog 网表。

```
dc_shell-t > set verilogout_unconnected_prefix "MY_DANGLE_"
```

使用如下 DC 命令可以得到所有 DC 变量的列表：

```
dc_shell-t > printvar *
```

为得到某一特定类的变量（如 test 相关的），使用如下命令 DC 将返回所有带有“test”字符串的变量。

```
dc_shell-t > printvar *test*
```

3.3.3　属性

属性（attribute）在本质上同变量相似，两者都用于存储信息，而属性用于存储特定设计对象的信息，如连线、单元或时钟。

总之，属性是预定义的且对于 DC 有特殊的意义，如果需要，设计人员可设置自己的属性。例如，set_dont_touch 是一个预定义属性，它用于在设计中设置 dont_touch，从而阻止 DC 对这个设计进行优化。

使用如下命令设置和取得设计对象的属性：

```
set_attribute <object list>
              <attribute name>
              <attribute value>
get_attribute <object list>
              <attribute name>
```

例如，使用如下命令可得到库（名为“STD_LIB”）中设置的 max_transition 值和库中的一个反相器（INN2X）的 area 属性：

```
dc_shell-t > get_attribute STD_LIB default_max_transition
dc_shell-t > get_attribute STD_LIB/INV2X area
```

使用如下 dc_shell-t 命令可以从 DC 删除属性：

```
remove_attribute <attribute name>
```

3.4 找寻设计对象

DC 和 PT 提供的最有用的命令之一是 get_*命令。有时，为了脚本或自动化综合过程的目的有必要在 dc_shell-t 中定位对象，get_*命令用于在DC中定位设计或库对象列表。DC提供了几类get_*命令，例如：

get_ports、get_nets、get_designs、get_lib_cells、get_cells、get_clocks 等等。

get 命令的完整列表可通过在 dc_shell-t 命令行输入“help get_*”找到。

get_*命令的一些例子如下。

```
dc_shell-t > set_dont_touch [get_designs blockA]
```

☺ 在 blockA 设计中应用 dont_touch 属性。

```
dc_shell-t > remove_attribute [get_designs *] dont_touch
```

☺ 从整个设计移除 dont_touch 属性。

```
dc_shell-t > get_lib_cells stdcells_lib/*
```

☺ 列出库 stdcells_lib 的所有库单元。

```
dc_shell-t > get_pins stdcells_lib/DFF1/*
```

☺ 列出库 stdcells_lib 中单元 DDF1 的所有引脚。

```
dc_shell-t > set_dont_touch_network [get_ports [list clk scan_en] ]
```

☺ 在指定的端口应用 dont_touch_network 属性。

3.5 Synopsys 格式

大多 Synopsys 产品支持和共享一个称为“db”格式的通用内部结构。db 文件是以 RTL 代码编写的、已映射门级设计或 Synopsys 库自身的代表文本数据的二进制编译形式。db 文件也可能包含应用于

设计的任何约束。

此外，所有 Synopsys 工具都能识别下面的 HDL 格式，且 DC 能够读写这些格式。

1. Verilog
2. VHDL
3. EDIF

现在，Verilog 和 VHDL 是用于设计编码的两种主要的 HDL。EDIF（电子设计交换格式）主要用于从一个工具到另一个工具的门级网表交换。过去 EDIF 曾广为使用，而近来 Verilog 被普遍使用，并以其读格式和描述的简单而赢得了主导地位。现在大多 EDA 工具都支持 Verilog 和 EDIF。

VHDL 通常不用于从一个厂商工具到另一个厂商工具的网表交换，因为它需要使用 IEEE 软件包，而且软件包在不同工具间可能还会有所不同。实际上这一语言用于设计编码和系统级验证。

3.6　数据组织

根据格式来组织文件是一个好的做法，这有利于综合过程自动化。使用下面的文件扩展名来组织它们是通常的做法：

脚本文件：	<filename>.scr
RTL Verilog 文件：	<filename>.v
已综合的 Verilog 网表：	<filename>.sv
RTL VHDL 文件：	<filename>.vhd
已综合的 VHDL 网表：	<filename> .svhd
EDIF 文件：	<filename> .edf
Synopsys 数据库文件：	<filename> .db
报告：	<filename> .rpt
日志文件：	<filename> .log

3.7　设计输入

在综合前，设计必须用 RTL 格式（虽然也有其他格式）输入到 DC 中，DC 提供了如下两种设计输入的方法：

a）“read”命令

b）“analyze/elaborate”命令

Synopsys 最初引进了 read 命令，随后有了 analyze/elaborate 命令。较之 read 命令，analyze/elaborate 命令为设计输入提供了一种快速而强有力的方法，因此，推荐 RTL 设计输入使用。

analyze 和 elaborate 命令是两种不同的命令，它使得设计人员可以在建立设计通用逻辑之前先对设计进行语法错误和 RTL 转换分析。通用逻辑或 GTECH 元件是 Synopsys 通用工艺无关库的一部分，它们是布尔函数的未映射表示并且作为工艺相关库的占位符。

analyze 命令也将转换结果保存在指定的，以后会用到的设计库（UNIX 目录）中。例如，一个先前分析过的设计不必再分析就能进行详细描述，从而节省时间。相反的，read 命令执行 analyze 和 elaborate 命令的功能但不保存分析的结果，因此使得过程比较慢。

为了在详细描述设计的过程中传递所需参数，参数化设计（如 VHDL 中使用的 generic 语句）必须使用 analyze 和 elaborate 命令。read 命令用于将编译前设计或网表输入到 DC 中。

表 3-1 从不同类别列出了 read 和 analyze/elaborate 命令间各个类别的主要区别：

表 3-1 read 和 analyze/elaborate 命令的区别

类 别	analyze/elaborate	read
输入格式	Verilog 或 VHDL 的 RTL。	所有格式：Verilog、VHDL、EDIF、db 等。
推荐用法	综合使用 Verilog 或 VHDL 格式的 RTL。	读网表、预编译设计等。
设计库	使用-library 选项来指定设计库，而不是调用 dc_shell 的目录。	不保存分析结果。
类属（在 VHDL 中使用）	在详细描述设计的过程中类属中的参数可以被设置。	不能用来传递参数。
结构（在 VHDL 中）	能指定在 VHDL 中的结构被详细描述。	不能指定在 VHDL 中的结构被详细描述。

与 DC 相反，PT 使用不同的命令进行设计输入。PT 作为静态时序分析器，只用于已映射的结构化网表。第 12 章将介绍 PT 所使用的设计输入命令。

3.8　编译指令

有时有必要从 HDL 源文件本身对综合过程加以控制。因为综合与仿真环境间可能存在差异，所以这一控制从根本上讲是必需的。其他情况下，或引导 DC 映射到某类元件，或直接在 HDL 源码中嵌入约束和属性。

DC 为 Verilog 和 VHDL 设计输入格式提供了许多编译器指令。这些指令提供了直接从 HDL 源码控制综合结果的方法。这些指令在 HDL 代码中作为“注释”声明，但对 DC 有着特殊的含义。这些特殊注释可改变综合过程，而对仿真没有影响。

下面几小节讲解 Verilog 和 VHDL 格式的一些最常用的指令。要得到完整的指令表，建议用户参考 Design Compiler 参考手册。

3.8.1　HDL 编译器指令

HDL 编译器指令用于由 Verilog 格式的 RTL 转换为 Design Compiler 使用的内部格式的过程。如上所述，转换的特定方面由 Verilog 源码中的注释控制。在每个指令的开始通常是 Verilog 注释//或/*，后面为关键字“synopsys”（全部小写），通常用户喜欢用前一种形式来指定 HDL 编译器指令。因此，为了简单起见，本节只讨论//注释风格的 HDL 指令。

假设所有以//synopsys 开始的注释都只包含 HDL 编译器指令。如果在//synopsys 语句后有除 HDL 编译器指令之外的任何语句（别的注释或部分 Verilog 代码），则 DC 将显示出错。

3.8.1.1　translate_off 和 translate_on 指令

translate_off 和 translate_on 指令是最为有用且经常使用的指令。它们提供指示 DC 从“//synopsys translate_off”开始停止 Verilog 源码转换，并在它到达下一指令“//synopsys translate_on”后再次开始转换的方法。这些指令必须成对使用，且以 translate_off 指令开始。

考虑这样一种情况，部分 RTL 源码只是用于动态仿真的目的（或许是利用这些语句来构造测试基准）。例 3.2 描述了这样一种情况：

其包含的 Verilog `*ifdef* 语句易于在仿真时在命令行设置参数。由于 VENDOR__ID 依赖于仿真所指定的模式，因此这样的代码显然是不可综合的。此外，由于 HDL 编译器无法处理这一语句，它将给出一个错误信息，该信息说明，由于“undefined macro `ifdef…”，而无法读取设计。

例 3.2

```
`ifdef MY_COMPANY
    `define VENDOR_ID 16'h0083
`else
    `define VENDOR_ID 16'h0036
`endif
```

translate_off 和 translate_on HDL 指令适用于这一情况以忽略如例 3.3 中说明的 Verilog 代码中的“只仿真”部分。结果逻辑包含只与 MY_COMPANY 相关的 VENDOR__ID 值，为将其变为其他值，用户必须编辑代码并移动 HDL 指令以使其他的 VENDOR_ID 值可见。

例 3.3

```
//synopsys translate_off
    `ifdef MY_COMPANY
// synopsys translate_on
    `define VENDOR_ID 16'h0083
//synopsys translate_off
    `else
    `define VENDOR_ID 16'h0036
    `endif
// synopsys translate_on
```

3.8.2 VHDL 编译器指令

同 HDL 编译器相似，VHDL 编译器指令是影响 VHDL 编译器行为的特殊 VHDL 注释。所有 VHDL 编译器指令由 VHDL 注释（--）开始，然后是 synopsys 或 pragma 语句，这给予编译器以特殊意义且迫使它执行指定任务。

3.8.2.1 translate_off 和 translate_on 指令

这些指令与先前为 HDL 编译器描述的指令的工作方式相似，除

了需要如下 VHDL 注释：

```
-- synopsys translate_off
-- synopsys translate_on
-- pragma translate_off
-- pragma translate_on
```

VHDL 编译器虽然对嵌入的代码要进行语法检查，但它忽略 translate_off/on 指令间的任何 RTL 代码。为了阻止编译器执行语法检查，代码必须完全透明，这可通过设置如下变量为真实现：

```
hdlin_translate_off_skip_text = true
```

这些指令主要用于阻止 VHDL 代码中用于仿真的结构。例如，用户可能在网表中包含了库语句，它指定了包含有网表中门的 VITAL 模型的库名称。这意味着为了仿真，网表中的门引自这个库。DC 读取 VHDL 代码会产生错误，因为该库语句只用于仿真。为了避免这个问题，可用上述指令将库语句包裹起来，以使 DC 完全忽略库语句。

3.8.2.2　synthesis_off 和 synthesis_on 指令

synthesis_off/on 指令同 translate_off/on 指令工作方式相似。synthesis_off/on 指令本身的行为不受 hdlin_translate_off_skip_text 变量值影响。然而，如果上述变量的值设为 false，translate_off/on 指令则执行完全相同的功能。

以上指令是隐藏只用于仿真的结构的更可取的方法。虽然 VHDL 编译器对这些指令中间的代码进行语法检查，但为了综合的目的它会忽略这些代码。

依据语法规则，可使用这些变量如下：

```
-- pragma synthesis_off
     <VHDL code goes here,used only for simulation>
-- pragma synthesis_on
```

3.9　小结

本章向读者介绍了 Synopsys 使用的各种术语和概念。

从 Synopsys 提供的一些工具的简介和目的开始，本章涵盖了

Synopsys 环境，其中包括 PhyC、DC 和 PT 所需的启动文件示例，随后介绍了对象、变量和属性的概念。

本章还简单介绍了 find 命令及其应用，并与设计输入方法一起讨论了不同的 Synopsys 格式。同时，还涉及到使用 read 命令与使用 analyze/elaborate 命令的优缺点。

最后，本章描述了 DC 用于隐藏只用于仿真的结构的一些最有用的指令结束。

贯穿本章的许多示例有助于读者理解上述这些概念。

第 4 章
Synopsys 工艺库

Synopsys 工艺库格式已成为事实上的库标准。它紧凑而富含信息的格式可以充分表示深亚微米工艺。Synopsys 库格式的流行显然是由于这样一个事实，即大多数布局布线工具都提供 Synopsys 库的直接转换，这种转换几乎是在 Synopsys 库的时序模型与布局布线的时序模型之间的一对一的映射。对库格式和延时计算方法的基本理解是成功综合的关键。

只要库里有许多不同驱动能力的单元，设计人员通常不关心工艺库的整个细节。然而为了成功进行优化设计，设计人员有必要对 DC 所使用的延时计算方法、线载模型及单元描述有清晰的理解。因此本章的目的是从设计者的角度描述 Synopsys 工艺库，而不是讨论库的结构和功能语法的细节。

4.1 工艺库

Synopsys 工艺库可以分为两大类：

1. 逻辑库
2. 物理库

4.1.1 逻辑库

逻辑库包含仅与综合过程有关的信息且通过DC用于设计的综合和优化。这一信息包含引脚到引脚的时序、面积、引脚类型和功耗以

及其他 DC 需要的必要数据。逻辑库中没有物理信息。

逻辑库是一个文本文件(通常用扩展名 ".lib"),通过使用 Library Compiler(LC)编译生成带有扩展名 ".db" 的二进制格式。

4.1.2 物理库

物理库包含单元的物理特征及与 Physical Compiler 相关的其他必要信息。这一信息可包含与单元的物理尺寸、层信息及单元方位相关的数据。对于每个逻辑单元,都有一个对应的物理单元。

物理库也是一个文本文件(通常用扩展名 ".plib"),且它可以通过 LC 编译生成带有扩展名 ".pdb" 的二进制格式。Synopsys 提供一个有用的应用程序 "lef2pdb",将标准 LEF(库交换格式)文件和工艺文件(也是 LEF 格式)作为输入并将其转化为 "pdb" 格式。前一个文件包含设计当中的每个单元的物理信息,而工艺文件则包含工艺的特定信息,如层数、间距、电阻、电容等。

下面是这个命令的用法:

```
lc_shell > lef2pdb –t tech.lef –l standard_cells.lef
```

对物理库及其语法的详细解释超出了本书的范畴,如需要了解其细节,建议读者参考物理库参考手册。

4.2 逻辑库基础

逻辑库包含如下信息:

a) 库类

b) 库级属性

c) 环境描述

d) 单元描述

4.2.1 库类

库类语句指定库名,接着是一左大括号。右大括号是库文件的最后一项,大括号里的任何内容都是整个库类描述的一部分。

```
library (ex25) {  /* start of library */
...
<library description>
...
}                    /*end of library*/
```

对于文件名与工艺库名，推荐使用相同的名称。

4.2.2　库级属性

库级属性是作用于整个库的语句，通常包含库特征，如工艺类型、日期、版本和用于整个库的默认值。

```
library (ex25) {
    technology (cmos) ;
    delay_model                    : table_lookup ;
    date                           : "Feb 29,2000" ;
    revision                       : "1.0" ;
    current_unit                   : "1A" ;
    time_unit                      : "1ns" ;
    voltage_unit                   : "1V" ;
    pulling_resistance_unit        : "1kohm" ;
    capacitive_load_unit (1.0,pf) ;
    default_inout_pin_cap          : 1.5 ;
    default_input_pin_cap          : 1.0 ;
    default_outut_pin_cap          : 0.0 ;
    default_max_fanout             : 10.0 ;
    default_max_transition         : 3.0 ;
    default_operating_conditions   :NOMINAL
    in_place_swap_mode             : match_footprint ;
    ......
    ......
}
```

4.2.3　环境描述

库中定义的环境属性用于对温度、电压和制造工艺的偏差建模，它包括比例因子（降低标称值）、时序范围模型和工作条件。此外，环境描述也包含 DC 用于估算连线延迟的线载模型。

4.2.3.1 比例因子

比例因子或 K 因子是乘数，它提供了基于工艺、电压和温度（或简称为 PVT）的偏差减小延迟值的方法。以下只列举了一些 K 因子语句，详细内容请参阅参考手册。

```
k_process_fall_transition                :1.0;
k_process_rise_transition                :1.2;
k_process_fall_propagation               :0.4;
k_process_rise_ propagation              :0.4;
k_temp_fall_transition                   :0.03;
k_ temp _rise_transition                 :0.04;
k_ temp _fall_propagation                :1.2;
k_ temp _rise_ propagation               :1.24;
k_ volt_fall_transition                  :0.02;
k_ volt _rise_transition                 :0.5;
k_ volt _fall_propagation                :0.9;
k_ volt _rise_ propagation               :0.85;
```

4.2.3.2 工作条件

库中定义的工作条件集指定了工艺、温度、电压和 RC 树模型，它们用于设计的综合与时序分析中。一个库使用一套工作条件来表征。在综合和时序分析中，若指定了另一套工作条件，则 DC 根据指定的工作条件用 K 因子减少延迟值。库开发者可在库中定义任意数目的工作条件。下面是在工艺库中工作条件的典型定义：

```
operating_conditions (WORST) {
    process   : 1.3;
    temperature   : 100.0 ;
    voltage   : 2.75 ;
    tree_type : worst_case_tree ;
}
operating_conditions (NOMINAL) {
    process   : 1.0;
    temperature   : 25.0 ;
    voltage   : 3.00 ;
    tree_type : balanced_tree ;
}
operating_conditions (BEST) {
```

```
        process   : 0.7;
        temperature   : 0.0 ;
        voltage   : 3.25 ;
        tree_type : best_case_tree ;
    }
```

前面已解释了工艺、温度和电压属性。tree_type 属性定义了使用的环境互连模型，DC 在计算互连延迟时使用这一属性值来选择合适的公式。worst_case_tree 属性是针对负载引脚在距离驱动最远的连线端的极端情况进行建模。在这种情况下，负载引脚承受整个连线电容和电阻。balanced_tree 模型使用的情况是其中所有的负载引脚在独立的距离驱动相等的连线上。这种情况下负载引脚承受相等的一份连线电容和电阻。best_case_tree 是当负载引脚正好在驱动附近进行建模。在这种情况下负载引脚只承受连线电容而没有任何连线电阻。

4.2.3.3　时序范围模型

timing_range 模型提供了额外的基于指定工作条件的计算信号到达时间的能力，Synopsys 提供这一能力是为了适应优化设计的工作条件的波动。在时序分析中，DC 使用时序范围来计算信号的到达时刻。

```
timing_range (BEST) {
        faster_factor   : 0.5 ;
        slower_factor   : 0.6 ;
}
timing_range (WORST) {
        faster_factor   : 1.2 ;
        slower_factor   : 1.3 ;
}
```

4.2.3.4　线载模型

wire_load（线载）类包含 DC 在设计的布图前阶段用来估计互连线延迟的信息。通常工艺库包括一些适合不同逻辑大小的模型，这些模型定义了 capacitance、resistance 和 area 因子。另外，wire-load 类也为所考虑的逻辑指定了 slope 和 fanout_length。

capacitance、resistance 和 area 因子分别代表每一单位长度的互连线的连线电容、电阻和面积。fanout_length 属性指定了与扇出

数目相关的连线的长度值。与扇出和长度一起，这一属性也可包含其他参数值，如 average_capacitance、standard_deviation 和 number_of_nets。当通过 DC 生成线载模型时，就自动写出这些属性及其值。手动创建时，使用 fanout_length 属性只需扇出值和长度值。对于超过 fanout_length 属性指定的最长长度的连线，斜率用来对已有的 fanout_length 值进行线性插值，从而确定它的值。

```
wire_load (SMALL) {
      resistance          : 0.2 ;
      capacitance         : 1.0 ;
      area                : 0 ;
      slope               : 0.5 ;
      fanout_length (1,0.020) ;
      fanout_length (2,0.042) ;
      fanout_length (3,0.064) ;
fanout_length (4,0.087) ;
      . . . .
      fanout_length (1000,20.0) ;
}
wire_load (MEDIUM) {
      resistance          : 0.2 ;
      capacitance         : 1.0 ;
      area                : 0 ;
      slope               : 1.0 ;
      fanout_length (1,0.022) ;
      fanout_length (2,0.046) ;
      fanout_length (3,0.070) ;
      fanout_length (4,0.095) ;
      . . . .
      fanout_length (1000,30.0) ;
}
wire_load (LARGE) {
      resistance          : 0.2 ;
      capacitance         : 1.0 ;
      area                : 0 ;
      slope               : 1.5 ;
      fanout_length (1,0.025) ;
      fanout_length (2,0.053) ;
      fanout_length (3,0.080) ;
      fanout_length (4,0.110) ;
```

```
    ....
    fanout_length (1000,40.0) ;
}
```

除了 wire_load 类，库中定义的其他属性用于根据逻辑的总单元面积来自动选择合适的 wire_load 类。

```
wire_load_selection (AUTO_WL) {
    wire_load_from_area (0, 5000, "SMALL") ;
    wire_load_from_area (5000, 10000, "MEDIUM") ;
    wire_load_from_area (10000, 15000, "LARGE") ;
}
default_wire_load_selection     : AUTO_WL ;
default_wire_load_mode          : enclosed ;
```

建议将 default_wire_load_mode 值设为“enclosed”或“segmented”而不是“top”。线载模型及其应用将在第 6 章详细论述。

4.2.4　单元描述

库中的每个单元都包含了描述功能、时序和其他与每个单元相关信息的多种属性。下面的例子只展示了对设计者有用的相关属性和信息，而不追求细枝末节，也不描述所有可能的属性：

```
cell(BUFFD0) {
    area   : 5.0 ;
    pin (Z) {
    max_capacitance    : 2.2 ;
    max_fanout    :4.0 ;
    function    : "I" ;
    direction    : output ;
    timing () {
        ...
    }
    timing () {
        ...
    }
    related_pin   : "I" ;
}
pin(I) {
```

```
        direction    : input ;
        capacitance  : 0.04 ;
        fnaout_load  : 2.0 ;
        max_transition  : 1.5 ;
    }
}
```

面积属性将单元面积定义为一无任何单位的浮点数，紧跟着的是引脚描述及其相关时序。

此外，一些设计规则检查（DRC）属性可以同单元的每个引脚相关。它们是：

- 输入引脚的 fanout_load 属性。
- 输出引脚的 max_fanout 属性。
- 输入或输出引脚的 max_transition 属性。
- 输出或输入输出引脚的 max_capacitance 属性。

DRC 条件基于厂商的制造工艺且不得违例。DRC 属性定义库单元安全工作的条件，换言之，在一定条件下（输出负载、输入斜率等）表征单元。设计违反这些条件将对单元的正常工作产生严重影响，从而可导致加工的芯片失效。

虽然前面的例子包含了所有四种属性，但通常只用两种，在大多情况下或者用 fanout_load 和 max_fanout，或者用 max_transition 和 max_capacitance。

fanout_load 和 max_fanout DRC 属性彼此相关，被驱动单元的每个输入引脚的所有 fanout_load 值的总和不能超出驱动器引脚输出的 max_fanout。考虑前面例子所示的单元（BUFFD0），这个单元包含与输出引脚 Z 相关的 max_fanout 值为 4.0，而它输入的 fanout_load 值为 2.0。这一单元不能驱动超过两个的同类单元（BUFFD0），由于

$$\text{max\_fanout(4)=fanout\_load(2) of } 1^{st} \text{ cell + fanout\_load(2) of } 2^{nd} \text{ cell}$$

如果发生 DRC 违例，DC 会用另一个具有更高 max_fanout 值的单元替换驱动单元。

max_transition 属性通常用于输入引脚，而 max_capacitance 则用于输出引脚。两种属性执行与 max_fanout 和 fanout_load 属性类似的功能。区别在于，max_transition 属性定义任何转换时间大于负

载引脚 max_transition 指定值的连线不能连到该引脚。输出引脚的 max_capacitance 指定驱动单元的输出引脚不能和总电容（互连及负载引脚电容）大于或等于输出引脚定义的最大值的连线相连。

如果发生 DRC 违例，DC 则会以另一个具有更高 max_capacitance 值的单元替换驱动单元。

此外，输出引脚包括定义引脚功能的属性和相对输入引脚的延迟值，输入引脚定义其引脚电容和方向。不应将 capacitance 属性同 max_capacitance 属性混淆起来，DC 利用 capacitance 属性只进行延时计算，而如上所述，max_capacitance 则用于设计规则检查。

另外还值得一提的是时序单元，时钟输入引脚使用另一个属性（clock : true）指定输入引脚为时钟类型，详细内容请参阅 Library Compiler 参考手册。

单元的 DRC 属性在单元库中是最有争议的部分。库开发者常常发现无法令每个人都感到满意，且常被指责没有为这些属性提供“正确”的数。这个问题是由于库在某种程度上与编码风格和所选用的方法相关而造成的。某一方法对某个设计可得到很好的结果，但对另一个设计则可能得不到好的结果。因此这一节的目的是，简要介绍为满足设计的需要，设计人员对库进行适当修改的解决方案。

为了适应设计要求，可以在每个单元基础上修改上述 DRC 属性值。然而必须注意的是，库中设置的 DRC 属性只能加紧而不能放松，只有在单元描述中预先指定这些属性才能这样做。用户应当认识到，如果这些属性没有在工艺库中单元的引脚上指定，则从 dc_shell 是不可能将这些属性添加到引脚的。

例如，使用下面的 dc_shell-t 命令将（先前描述的库 ex25）单元 BUFFD0 的引脚 Z 指定的 max_fanout 值由 4.0 改为 2.0：

```
dc_shell-t > set_attribute [get_pins ex25/BUFFD0/Z] max_fanout 2
```

也可在上述命令中运用通配符号涵盖不同的单元，这对于要求全局改变的情况很有用。例如，用户可用如下命令改变工艺库中所有 0 驱动强度单元的 max_fanout 值：

```
dc_shell-t > set_attribute [get_pins ex25/*D0/Z] max_fanout 2
```

类似地，set_attribute 命令用于改变其他 DRC 属性值。为了全局实现，上述命令可在.synopsys_dc.setup 文件中指定。

4.3 延时计算

Synopsys 支持一些延时模型，它们包括 CMOS 通用延时模型、CMOS 分段线性延时模型和 CMOS 非线性查表模型。由于前两种模型不能有效地表示 VDSM 几何尺寸引起的真实延时，因此目前不常用。非线性延时模型在 ASIC 界是最为流行的延时模型。

4.3.1 延时模型

非线性延时模型（NLDM）方法使用各种输入摆率和输出负载电容通过电路模拟器来表征一个单元的晶体管。以输入转换和输出负载电容作为计算最终单元延时的决定因子，结果构成一张表。

图 4-1 描述了最终的延时和摆率插值得到的一个非线性延时模型。模型的准确性要看所选输入摆率和负载电容的精度及范围。

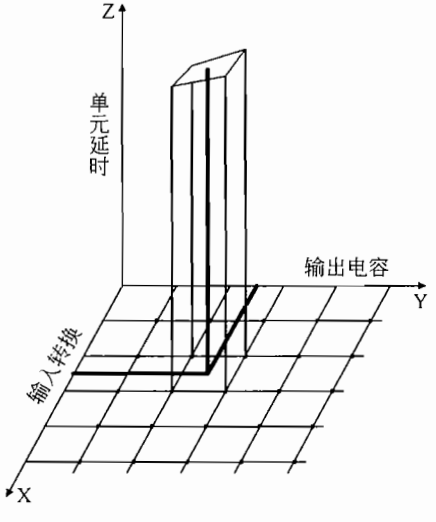


图 4-1 NLDM 表

如果延时数落在方块（库中的表）中，则将使用插值技术计算延时，通过数值方法使用周围四点的值来确定延时值，当任何参数落在表格外时，将会出现问题。DC 使用很好的设计来外推所得的延时，

但经常得到一个极高的值。这不一定是坏事，因为在静态时序分析中高值是很容易看出来的，设计人员有机会改正这种情况。

4.3.2　延时计算问题

进行单元延时计算要用输入转换时间和输出电容负载。单元输入转换时间的估算要以驱动单元（前一单元）转换延迟为基础。如果驱动单元包含多个时序弧，那么使用最坏转换时间作为被驱动单元输入，这直接影响设计的静态时序分析和生成的 SDF 文件。

考虑如图 4-2 所示的逻辑。信号 *reset* 和 *signal_a* 是实例 U1 的输入，假设 *reset* 信号不如 *signal_a* 关键。*reset* 信号是慢信号，因此这一信号的转换时间要比 *signal_a* 长。这就要为单元 U1 计算两个转换延迟（从 A 到 Z 的 2ns 和从 B 到 Z 的 0.3ns）。当生成 SDF 时，将这两个值作为单元 U1 单元延迟的一部分分别写出。然而，问题出现了，DC 该用这两个值中的哪个去计算单元 U2 的输入转换时间呢？DC 用前一个门（U1）的最坏（最大）转换值作为被驱动门（U2）的输入转换时间。由于 *reset* 信号转换时间比 *signal_a* 更长，因此 2ns 值将被用于 U2 的输入转换时间，单元 U2（阴影单元）将计算一个大的延迟值。

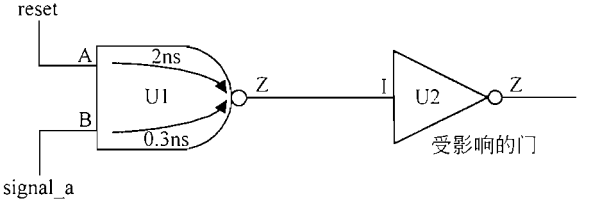


图 4-2　延时计算

为避免这一问题，需告知 DC，不要进行从单元 U1 的引脚 A 到引脚 Z 的时序弧的延迟计算。应在写出 SDF 前进行这一步，如下的 dc_shell 命令可用于这一目的：

```
dc_shell-t > set_disable_timing U1 –from A –to Z
```

不幸的是，在静态时序分析中也会出现这样的问题。未禁止虚假路径的时序计算会导致为被驱动单元计算出大的延迟值。

4.4 何谓好库？

单元库决定综合逻辑的整体性能，一个好的单元库会得到最小面积的高速设计，而一个差的库会恶化最终结果。

过去，单元库是基于电路原理图的。设计人员选择合适的单元并将它们手动连接起来以生成一个设计网表。当自动综合引擎广为使用后，同样的基于电路原理图的库被转化并用于综合。由于综合引擎为了优化而依赖于很多因素，因而这种方法总是得到性能差的综合逻辑。因此，在单元库设计时，兼顾综合实现方法是必要的。

下面的指导方针概述了综合引擎需要的工艺库中特定类型的单元：

a）所有的单元都要有各种驱动强度。

b）反相器和缓冲器要有更多的驱动强度。

c）单元要有平衡的上升和下降延时（用于时钟树缓冲和门的时钟）。

d）同一物理单元有相同逻辑功能及其反相作为独立输出（如或门和或非门作为一个单元），还要有各种驱动程度。

e）相同逻辑功能及其反相作为独立的单元（如与门和与非门作为两个独立的单元），也要有多种驱动强度。

f）复杂单元（如有一个输入反相的与或非、或与非、与非门）要有各种高驱动强度。

g）具有不同范围的驱动强度的高扇入单元（如有六个输入和一个输出的与或非门）

h）有不同驱动强度的各种触发器，包括正沿和负沿触发。

i）每个触发器都有可用的单个或多个输出（如仅有 Q 和仅有 QN 或两者都有），每个都有各种驱动强度。

j）包括 Set 和 Reset 不同输入的触发器（如仅有 Set，仅有 Reset，无 Set 或无 Reset，既有 Set 也有 Reset）。

k）有各种锁存器，既有正边沿使能，又有负边沿使能，每种都有不同的驱动强度。

l）一些延迟单元。当修正保持时间违例时，这些单元很有用。

使用上述指导方针可得到用于综合算法的优化库，这为 DC 提供

了从各种单元中选择实现设计的最好的逻辑方法。

值得注意的是，高扇入单元的使用，虽然有利于减少整个单元面积，却可能引起布线拥塞，这将不经意地导致时序恶化、和/或增加了布线设计的面积。因此，建议谨慎使用这些单元。

一些设计人员喜欢从工艺库中排除高扇入低驱动强度的单元，并再次使用基于布线引擎所用的算法和设计人员所用的布局类型（时序驱动等）。如果布线器没有约束，那么它使用这种方法，在布局单元的同时把权重与设计中每条连线相关联。根据连线的权重，单元被牵引向具有最高权重的源。高扇入单元的输入（由于输入的数目）加权比它们的输出（单一输出）加权更大。因此，布线器将这些单元置于驱动它的门附近，这将导致高扇入单元被拖离它要驱动的单元，导致高扇入单元驱动长的连线。如果高扇入单元不足以驱动长的连线（大电容），那么结果是高扇入单元和被驱动门（由于缓慢的输入转换时间）计算出大的单元延迟。通过从工艺库中剔除高扇入单元的低驱动强度来防止布图后这一问题的出现。

4.5　小结

本章从设计人员的角度介绍了 Synopsys 逻辑库的内容，重点在于正确地使用和理解逻辑库，而不在于仅与库开发者相关的细节上。

本章还简要介绍了 Physical Compiler 使用的物理库，重点没有放在描述这个库的语法和功能上，因为这个论题的讨论超出了本书的范围。

本章由逻辑库基础及库中单独的类开始，详细地解释了各类的相关部分，包括用来完成库任务的所有属性的解释。

特别强调了延迟计算方法、工作条件、线载模型和单元描述。每一步都详细解释了相关问题及其解决办法。

最后，向用户提供了组成综合引擎优化好库的建议，包括把布图工具中布线器的行为考虑在内的有用提示。

第 5 章
划分和编码风格

合理的设计划分和好的 HDL 编码风格对成功的综合影响很大。

逻辑划分是成功综合（和布局布线，如果布图是层次化的）的关键。传统上，设计人员根据每个模块的功能划分设计，而不考虑综合过程。作为错误划分的一个结果，不可变的边界恶化了综合结果，使得优化变得困难。正确地划分设计能显著地增强综合的效果，此外，还能减少编译时间和简化脚本管理。

好的编码风格不仅对综合过程，而且对 HDL 代码的易读性都是必要的。如今，许多设计人员只强调验证设计的功能性。由于时间的限制和/或组员间缺少交流，因此设计人员没有仔细检查 HDL 编码风格的闲情逸致。然而，事实上一个好的编码风格不仅可以减小芯片面积和有助于顶层时序，还可产生更快的逻辑。

5.1 综合划分

划分可视为采用“分而治之”的思想，即把复杂的设计化简为更简单的和易处理的模块。划分设计的一个最重要的优点是它提高设计的复用。

合理划分的设计除了容易满足时序约束，也能方便在组员间分发和管理不同的设计模块。

以下建议可取得最佳综合结果并减少编译时间。

a）相关组合逻辑保持在同一模块中。

b）为设计复用进行划分。

c）根据功能分割模块。

d）结构逻辑与随机逻辑分离。

e）合理限制模块的大小（依据机器内存的容量）。

f）划分顶层（I/O 压焊块、边界扫描和核心逻辑相分离）。

g）不要在顶层添加粘合逻辑。

h）将状态机从其他逻辑中分离出来。

i）避免一个模块有多个时钟。

j）分离用于同步多个时钟的模块。

k）划分时，考虑布图风格。

在设计层次已经由先前编写的 HDL 代码确定后，group 和 ungroup 命令给设计者提供了在 DC 中改变划分的能力。图 5-1 描述了这样的行为。

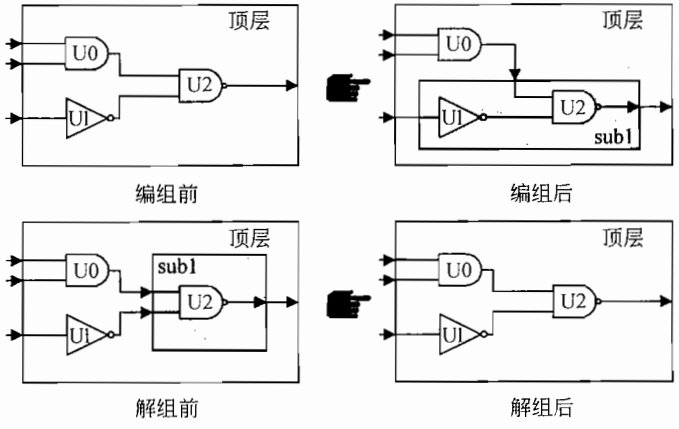


图 5-1　改变划分

group 命令把指定的实例组合成一个单独的模块。在图 5-1 中，使用如下命令将实例 U1 和 U2 组合在一起形成一个名为 sub1 的子模块：

```
dc_shell > current_design top
dc_shell > group {U1 U2} –design_name sub1
```

ungroup 命令进行逆操作。如图 5-1 所示，使用如下命令移除层次：

```
dc_shell > current_design top
dc_shell > ungroup –all
```

设计者也可使用带–flatten 和–all 选项的 ungroup 命令来打平全部

层次，如下所示：

```
dc_shell > ungroup –flatten –all
```

5.2　何谓 RTL?

如今，RTL 或寄存器传输级是高层次设计规范中最流行的形式。设计的 RTL 描述通过从一个寄存器到另一个寄存器的逻辑变换和传输来描述设计。逻辑值被存储在寄存器中，通过一些组合逻辑对其求值，随后将结果存储于下一个寄存器中。

RTL 功能犹如软件和硬件之间的一座桥梁。它是具有强大图形内涵、暗含图形或结构的文本。它是与工艺无关类似于网表的文本结构的描述。

5.2.1　软件与硬件

编写 HDL 代码经常会遇到一个问题就是软件的思维倾向。HDL 是由逻辑网表的表示演化而来的，在其初始形式（寄存器传输级）HDL 是用独立于任何特定工艺库的格式来表示逻辑。HDL 抽象的更高级为行为级，它允许设计同时序和明确的先后顺序无关。

常常期望不论 HDL 是如何写的，综合工具综合 HDL 都能得到最小面积和最高性能。问题就是在高层次完成同样的功能有很多编写代码的方法，例如，可用 case 语句或 if 语句编写条件表达式。逻辑上，这些表达式可执行相同的任务，但考虑到推断的逻辑类型、面积和时序时，综合后却可得到极不相同的结果。给刚开始使用综合的人们的一个明智的告诫——考虑硬件！

5.3　通用指导方针

下面为每个设计人员应当注意的通用指导方针，虽然没有要求设计人员完全遵守这些指导方针，然而遵守它们可极大地提高综合后逻辑的性能，并可得到适合自动化综合过程的更有条理的设计。

5.3.1 工艺无关

HDL 代码应采用与工艺无关的方式编写。库中门的硬编码实例应当最小化。优先选择的是推断而不是例化。这样做的优点是通过重新综合能用任何 ASIC 库和新工艺来实现 RTL 代码。这对于广泛用于设计的可综合 IP 核是尤其重要的。

如果不可避免要布置库中的门，那么将所有例化的门组合起来以形成自己的模块。这有助于设计特定库方面的管理。

5.3.2 时钟相关逻辑

a）包括门控时钟逻辑的时钟逻辑和复位生成应当保持在一个模块内——综合一次并且不再涉及了。这有助于生成简明的时钟约束规范。另一个优势是由时钟逻辑驱动的模块可用理想的时钟规范约束。

b）避免每个模块有多个时钟——尽量保持每个模块有一个时钟。这样的限制有助于避免以后约束包含多个时钟的模块时出现的困难，也有助于在物理级处理时钟扭斜问题。有时这是无法避免的，例如从一个时钟域到另一个时钟域同步信号的同步逻辑。对于这样的情况，建议设计人员把同步逻辑隔离为一个独立的模块单独综合。这包括在主模块中例化同步逻辑之前，给它设置 dont_touch 属性。

c）应给时钟赋以有意义的名称。建议保持时钟名以反映其功能和频率。另一个好习惯是在各个层次上，同一时钟保持相同的名字，也就是说，当时钟穿越层次时它的名字不应改变。这可简化脚本的编写并有助于自动化综合过程。

d）对 DFT 扫描插入，要求从原始输入控制时钟。这涉及在时钟源添加一个可控制的多路选择器。把多路选择器逻辑并入包含所有其他时钟逻辑的模块中，分离时钟逻辑模块有助于在门的尺寸和引用类型方面对综合后的逻辑进行精调。如果需要，这个小模块能容易地调整为适合优化的解决方案。

5.3.3 顶层没有粘合逻辑

顶层只用于连接模块，它不应包含任何组合的粘合逻辑。这一结

构的一个优点是，既不需要进行非常耗时的顶层编译，也不用额外的综合，只需要简单的连接。如果进行层次化的布局布线，那么顶层没有粘合逻辑也便于布图。

5.3.4　模块名与文件名一致

一个好的习惯是保持模块名（或实体名）同文件名一致。不要在一个文件中描述多个模块或实体，一个文件应只包含一个面向综合的模块/实体定义。这在使用脚本语言如 PERL、AWK 等定义一个易懂的方法方面有很大的优点。

5.3.5　压焊块同核心逻辑相分离

将顶层划为“压焊块”和“核心”两个单独的模块。压焊块通常是例化的而不是推断的，因而最好能将其同核心逻辑分离开来。这可同时简化在设计的所有压焊块上设置 dont_touch 属性。通过把 Pad 保持在一个单独的模块，我们把与工艺相关的部分同 RTL 代码分开了。

5.3.6　最小化不必要的层次

不要生成不必要的层次，每个层次设置一个边界。如果生成不必要的层次，则性能会下降。这是因为 DC 不能跨层次进行有效的优化。为取得更佳的结果，在编译设计前，可用 ungroup 命令打散不需要的层次。

5.3.7　寄存所有输出

这是著名的 Synopsys 推荐。模块的输出应直接从寄存器输出。尽管有时是不可行的，但这种编码/设计风格可简化约束规范并且也有助于优化。这一风格可避免组合逻辑跨越模块边界。通过避免这类编译技术很常见的乒乓效应，它也可增加 characterize-write-script 综合方法的有效性。

5.3.8 FSM 综合指导

下面是编写有限状态机的指导，它们有助于优化逻辑。

a）状态名应使用 VHDL 中的“枚举类型”或 Verilog 中的“参数”来描述。

b）计算下一状态的组合逻辑应在与状态寄存器独立的其 *process* 或 *always* 块中。

c）用 *case* 语句实现下一状态组合逻辑。

5.4 逻辑推断

高层次描述语言（HDL）如 VHDL 和 Verilog 是综合的前端。HDL 允许设计用工艺无关的方式来表示。然而，在编写设计的 HDL 描述的方式上，综合加了一定限制。不是所有的 HDL 结构都能被综合，不仅如此，综合希望用特定的方式编写 HDL 代码以得到想要的结果。可以说综合是模板驱动的（如果代码是用综合工具理解和期望的模板写成的），那么结果就是正确的且可预知的。用于综合的模板及其他编码模式称为编码风格。为了得到好的结果，设计人员有必要对编码风格、逻辑推断及 DC 生成的对应逻辑结构进行深入的理解。

5.4.1 不完全敏感信号表

不完全敏感信号表会导致源 RTL 和综合逻辑间的仿真不匹配，这是设计人员最常犯的一个错误。DC 对在 *process* 或 *always* 块中却不在敏感信号表中的信号给出一个警告。这主要是仿真问题，由于当激活时没有触发进程（因为敏感信号表中缺失的信号）。对包含不完全敏感信号表的块，综合后的逻辑在大多数的情况下是正确的。然而，强烈建议设计人员要特别注意敏感信号表并使它完整以消除在综合循环结束后的任何意外情况。

Verilog 样例

```
always @(weekend or go_to_beach or go_to_work)
```

```
begin
    if (weekend)
        action = go_to_beach;
    else if (weekday)
        action = go_to_ work;
```

VHDL 样例

```
process (weekend , go_to_beach , go_to_work)
begin
    if (weekend) then
        action <= go_to_beach;
    else if (weekday) then
        action = go_to_ work;
    end if ;
end process ;
```

上述例子在它们的敏感信号表中没有包含信号“weekday”。综合后逻辑仍是正确的，然而在仿真过程中每次信号“weekday”改变其值时并不会触发进程，这会引起源 RTL 与综合后逻辑仿真结果间的不匹配。

5.4.2　存储元件推断

有两种类型的存储元件——锁存器和触发器。锁存器为电平敏感存储元件，而触发器通常为边沿敏感。只要锁存器的使能是有效的，锁存器就是透明的。一旦锁存器使能无效，它在其 Q 输出保持当时存在于 D 输入的值。另一方面，触发器由时钟的上升或下降沿触发。

锁存器是简单器件，因而与触发器相比占的面积要小。然而，锁存器通常更麻烦，因为在设计中它们的存在使 DFT 扫描插入变得困难，尽管不是不可能。对包含锁存器的设计进行静态时序分析也是很复杂的，这是因为当使能有效时，它们是透明的。由于这个原因，设计人员通常喜欢触发器胜过锁存器。

以下几节将详细介绍如何避免锁存器以及在需要时如何推断它们。

5.4.2.1　锁存器推断

当条件语句没有完全指定时就要推断出锁存器。缺少 *else* 部分的

if 语句就是一个不完全指定条件的例子。下面是一个分别用 Verilog 和 VHDL 描述的例子：

Verilog 样例

```
always @(weekend)
begin
    if (weekend)
        action = go_to_beach;
end
```

VHDL 样例

```
process (weekend)
begin
    if (weekend = '1') then
        action <= go_to_beach;
end process ;
```

上述语句可使 DC 推断一个由信号“weekend”使能的锁存器，上例中，当信号“weekend”为 0 时将不对“action”赋值。为避免无意的锁存器推断总是要覆盖所有的情况，这可通过使用 *else* 语句或在 *if* 分支外使用 *default* 语句实现。

从一个非完全指定的 Verilog Case 语句也可推断出锁存器：

```
`define sunny 2'b00
`define snowy 2'b01
`define windy 2'b10

wire [1:0] weather ;

case (weather)
    sunny : action <= go_motorcycling ;
    snowy : action <= go_skiing ;
    windy : action <= go_paragliding ;
endcase ;
```

上述 *case* 语句中只覆盖了“weather”的 4 个可能值中的 3 个，这导致在信号“action”上推断出一个锁存器。注意，对上例使用 Synopsys full_case 指令来避免锁存器推断，如第 3 章所解释的。下面的例子包含了提供第四个条件的 *default* 语句，因此避免了锁存器

推断。

```
case (weather)
    sunny : action <= go_motorcycling ;
    snowy : action <= go_skiing ;
    windy : action <= go_paragliding ;
    default: action <= go_paragliding ;
endcase ;
```

VHDL 不允许不完全的 case 语句，这意味着必须使用 *others* 子句，从而 VHDL 不会出现上述问题。然而，如果某一个输出信号没在 *case* 语句的每个分支都赋一个值，VHDL 仍可推断锁存器。其推论是输出必须在所有分支都赋一个值以防止 VHDL 中的锁存器推断。

```
case (weather) is
    when sunny   => action <= go_motorcycling ;
    when snowy   => action <= go_skiing ;
    when windy   => action <= go_paragliding ;
    when others => null ;
endcase ;
```

上例虽然包含 other 子句但也会推断锁存器，因为输出信号“action”没有在 others 子句被赋予一个确定的值。为避免这一问题，所有分支应完全指定，如下所示：

```
case (weather) is
    when sunny   => action <= go_motorcycling ;
    when snowy   => action <= go_skiing ;
    when windy   => action <= go_paragliding ;
    when others => action <= go_paragliding ;
endcase ;
```

5.4.2.2　寄存器推断

DC 为寄存器推断提供了各式模板，这是为了支持不同的时钟边沿类型和复位机制。当敏感信号表中指定了一个边沿，就可推断一个寄存器。这个边沿可以是正沿也可以是负沿。

5.4.2.2.1　Verilog 中寄存器推断

在 Verilog 中，当在 *always* 块的敏感信号表中指定了一个边沿，就会推断一个寄存器。为在 *always* 块中赋值的每个变量都推断一个

寄存器，所有不直接依赖时钟沿的变量赋值应在单独的、其敏感信号表不含边沿指定的 *always* 块中进行。

使用如下模板推断简单的正沿触发的 D 触发器：

```
always @(posedge clk)
    reg_out <= data ;
```

为推断出带复位的寄存器，向敏感信号表中添加复位信号并在 *always* 块中编码复位逻辑。以下是一个带异步复位的 D 触发器的例子：

```
always @(posedge clk or posedge reset)
if (reset)
    reg_out <= 1'b0 ;
else
    reg_out <= data ;
```

简单地把“reset”信号从敏感信号表移除，就得到同步复位。在这种情况下，由于块只由时钟沿触发，所以也只在时钟边沿进行复位。

```
always @(posedge clk)
if (reset)
        reg_out <= 1'b0 ;
else
        reg_out <= data ;
```

使用下面的模板可推断出负边沿触发的触发器：

```
always @(negedge clk)
    reg_out <= data ;
```

工艺库中缺乏负边沿触发的触发器会导致DC推断一个带额外反相时钟信号的反相器的正沿触发的触发器。

5.4.2.2.2 VHDL 中寄存器推断

在 VHDL 中，当在 *process* 体中指定一个边沿时，就会推断一个寄存器。下例描述了用于推断 D 触发器的 VHDL 模板：

```
reg1 : process (clk)
begin
```

① 译者注：原书误作 always@(posedge clk or reset)。

```
        if (clk'event and clk = '1') then
            reg_out <=data ;
        end if ;
    end process Reg1 ;
```

由于每当调用函数时，会重新赋值函数中声明的变量，因此 DC 不会为函数中声明的变量推断锁存器。

带异步和同步复位的寄存器的编码风格模板与 5.4.2.2.1 节所示的 Verilog 模板在本质上是类似的。

用以下模板可推断出负边沿触发的触发器：

```
reg1 : process (clk)
begin
    if (clk'event and clk = '0') then
        reg_out <=data ;
    end if ;
end process Reg1 ;
```

缺乏负边沿触发的触发器的工艺库会使DC推断一个带额外反相时钟信号的反相器的正沿触发的触发器。

5.4.3　多路选择器推断

依据设计要求，可用不同的方法进行 HDL 编码来推断使用多路选择器的各种结构。它们包括所有输入到输出有相同延时的单个多路选择器，或使用级联结构多路选择器以优先选择输入信号的优先级编码器。

正确使用 *if* 和 *case* 语句是一个复杂的论题，这超出了本章的范围。应用注解（来自 Synopsys）和其他当前可得到的出版物来理解如何正确使用这些语句，因此本章的目的是引导用户向外界寻求信息资源。下面几小节只提供简要的讨论。

5.4.3.1　使用 case 语句的多路选择器

通常，*if* 语句用于推断锁存器和优先级编码器，而 *case* 语句用于实现多路选择器。推荐只使用 *case* 语句来推断多路选择器。*if* 语句可用于推断锁存器和优先级编码器。它们也可有效地用于优先后达到的信号。这类优先是与实现相关的，它也限制了其复用性。

为避免在 *case* 语句中推断锁存器，*case* 语句的 *default* 部分应当总是被指定。例如，对于状态机，默认行为是 *default* 子句覆盖的所有状态引起到“开始”状态的跳转。带 *default* 子句的 *case* 语句是写 *case* 语句的优先方法，因为它使得 HDL 与综合工具无关。使用像 *full_case* 和 *parallel_case* 使得 RTL 代码依赖于综合工具，应当避免使用这些指令。

如果默认行为是赋无关项，则 RTL 仿真和综合结果间的行为将会产生差异。这是因为 DC 可随机优化无关项导致结果逻辑发生改变。

5.4.3.2 if 语句与 case 语句——优先级事例

有多个分支的多重 *if* 语句生成优先级编码器结构。

```
always @( weather or go_to_ work or go_to_ beach)
    begin
        if (weather[1]) action = go_to_beach ;
        else if (weather[0]) action = go_to_work ;
        else action = 0;
    end
```

上例中，信号“weather”是一个 2bit 输入信号并且用于选择两个输入，“go_to_work”和“go_to_beach”的输出为“action”。综合后将生成级联多路选择器结构的优先级编码器，如图 5-2 所示。

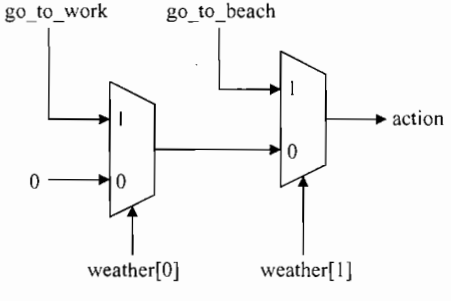


图 5-2 使用多重 *if* 语句进行综合后的结果

如果上例使用 *case* 语句（而不是多重 *if* 语句），其中选择索引的所有可能值被唯一地覆盖，则得到一个单个的多路选择器，如图 5-3 所示。

② 原文为 if (weather [0]) action=go_to_work; if (weather [1]) action=go_to_beach; 与图 5-2 不符。

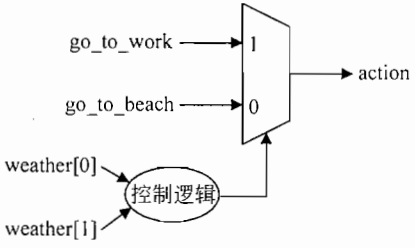


图 5-3　使用 *case* 语句或单一的 *if* 语句进行综合后的结果

如果用一个 *if* 语句和 *elsif* 语句覆盖所有可能的分支，会得到同样的结构（图 5-3）。

5.4.4　三态推断

当高阻（Z）赋给输出时，推断出三态逻辑。通常不提倡随意使用三态逻辑，原因如下：

a）三态逻辑降低可测性。

b）三态逻辑难以被优化，因为它不能被缓冲。这导致 max_fanout 违例和重负载连线。

然而，三态逻辑也有其长处，就是能显著节省面积。

Verilog 实例

```
assign tri_out = enable ? tri_in : 1'bz ;
```

VHDL 实例

```
tri_out <= tri_in when (enable = '1') else 'Z' ;
```

5.5　顺序相关

Verilog 和 VHDL 都提供顺序相关/无关的变量赋值。它们的正确使用可得到需要的结果，而不正确的使用则会引起综合逻辑与源 RTL 行为不同。

5.5.1 Verilog 中阻塞与非阻塞赋值

当进行顺序赋值如几个互斥的数据传输的流水线操作和建模时，使用非阻塞语句是重要的。因为最终结果依赖于赋值的计算顺序，所以在顺序进程中使用阻塞赋值可引起竞争条件。非阻塞赋值是顺序无关的，因此它们同硬件的行为紧密相符。

非阻塞赋值使用“<=”操作符，而“＝”操作符用于阻塞赋值。

```
always @(posedge clk)
    begin
        firstReg        <= data ;
        secondReg       <= firstReg ;
        thirdReg        <= secondReg ;
    end
```

在硬件中，寄存器更新将按如上所示的相反的顺序发生。使用非阻塞赋值可使赋值按与硬件同样的方式发生，也就是说，thirdReg 将会更新为 secondReg 的旧值，并且 secondReg 会更新为 firstReg 的旧值。如在上例中使用阻塞赋值，信号“data”在仿真时会同时传播至 thirdReg。

阻塞赋值通常应用于组合 always 块中。

5.5.2 VHDL 中的信号与变量

同 Verilog 类似，VHDL 通过使用信号和变量也产生顺序相关性。信号赋值可等同于 Verilog 的非阻塞赋值，也就是说，它们是顺序无关的。变量赋值是顺序敏感的且同 Verilog 阻塞赋值相关联。

变量赋值使用“:＝”运算符，而运算符“<=”用于信号赋值。

下例描述了在顺序 process 块中信号赋值的用法。产生的硬件包括三个寄存器，信号“data”由 firstReg 到 secondReg 再到 thirdReg 进行传播。RTL 仿真也将给出同样的结果。

```
process(clk)
begin
    if (clk'event and clk = '1') then
        firstReg <= data ;
```

```
            secondReg   <= firstReg ;
            thirdReg    <= secondReg ;
        end if ;
    end process ;
```

一般的建议是在顺序进程中只用信号赋值而在组合进程中只用变量赋值。

5.6　小结

本章强调了适用于综合的划分和编码风格，提供了各种指导和建议来帮助用户合理划分，并正确地进行 RTL 编码以有效地利用综合引擎。

本章的开始介绍了成功的划分技巧的建议及为什么它们是必要，随后简要讨论了“何谓 RTL”。重点在于编写设计时要“考虑硬件”。

其次，介绍了包括各种建议和技巧的通用指导，虽然这对综合不是必要的，却对成功的优化有很大影响。按照这些建议产生的优化设计适用于自动化综合过程。

描述编码风格是本章重要的一节。在这一节中，提供了大量实例作为推断正确逻辑的模板。这包括锁存器、寄存器、多路选择器和三态逻辑单元的推断。每一步都讨论了优缺点和正确的用法。

最后一节介绍了 Verilog 和 VHDL 语言的顺序相关特性，还讨论了利用两种语言的顺序相关特性的合理编码技巧。

第 6 章
设计约束

本章讨论了指定设计环境及其约束的过程，描述了各种广为使用的 DC 命令和其他可用于综合复杂 ASIC 设计的约束。

请注意本章所介绍的命令只包括最常使用的选项。建议设计人员参考 DC 参考手册以查询特定命令的所有可用选项的列表。

本章包含了对 Synopsys 工具的初学者和高级用户都有用的信息。考虑到与理想情况的偏差，本章着重介绍“实际”应用。换言之，“并非所有的设计或设计人员都遵照 Synopsys 建议”。本章包括大量有用的思想，用“☺”标出以引导读者在实际应用中选择命令。

6.1 环境与约束

为了从 DC 得到最佳结果，设计人员应通过描述设计环境、目标和设计规则来系统地约束其设计。约束可包括时序和/或面积信息，通常由设计规范给出。DC 运用这些约束条件进行综合并且试图优化设计以达到最终目标。

6.1.1 设计环境

到目前为止，假设设计已经完成了划分、编码和仿真，下一步就是要描述设计环境，这一步需要为设计定义工艺参数、I/O 端口属性和统计线载模型。图 6-1 说明了用于描述设计环境的基本 DC 命令。

set_min_library 是 DC98 版本引进的一个新命令，该命令允许用

户同时指定最坏情况和最佳情况的库。这在最初的编译时有用，它能防止在修正保持时间违例时违反建立时间。

```
set_min_library <max library filename>
                -min_version <min library filenames>
dc_shell-t>set_min_library "ex25_worst.db" -min_version"ex25_best.db"
```

☺ 上一命令可在增量编译时用于修正保持时间违例或用于合理优化。在这种情况下，用户应为工作条件设置最小值和最大值。

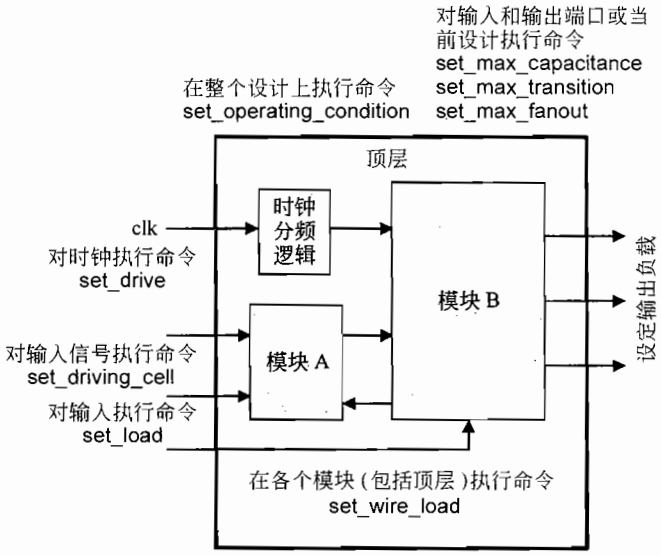


图 6-1 基本设计环境

set_operating_conditions 描述了设计的工艺、电压及温度条件。Synopsys 库包含这些条件的描述，通常描述为 WORST、TYPICAL 和 BEST 情况。工作条件名称是与库相关的，用户应与他们的库厂商核对以做出正确设置。通过改变工作条件命令的值，可覆盖工艺偏差的整个范围。WORST 情况工作条件通常用于布图前综合阶段，因此以最大建立时间优化设计。BEST 情况条件通常用于修正保持时间违例。因为 WORST 和 BEST 情况分析也包括 TYPICAL 情况，所以多数情况下可忽略 TYPICAL 情况。

```
set_operating_conditions <name of operating conditions>
dc_shell -t > set_operating_conditions WORST
```

☺有可能用 WORST 和 BEST 情况同时优化设计。如下所示，在上述命令中使用–min 和–max 选项进行优化。这对修正设计保持时间违例十分有用。

```
dc_shell -t > set_operating_conditions –min BEST –max WORST
```

set_wire_load_model 命令用来为 DC 提供估计的线载信息，反过来 DC 使用线载信息把连线延迟建模为负载的函数。通常 Synopsys 工艺库中列出了许多线载模型，每个模型代表一个特定大小的模块。此外，为了对自己模块的连线进行准确的建模，设计人员也可选择生成定制线载模型。

```
    set_wire_load_model –name <wire-load model>
dc_shell -t > set_wire_load_model –name MEDIUM
```

set_wire_load_mode 定义了三种同建模连线负载相关的模式，分别为 top、enclosed 和 segmented，通常只使用前两种模式。由于 segmented 线载模式依赖于特定连线段的线载模型，因此它很少使用。

top 模式定义层次中的所有连线将继承和顶层模块同样的线载模型。如果打算以后打平子模块进行布图，可选择这种线载模型。如果用户采用自底向上的编译方法来综合设计也可选择这种模式。

第二种模式 enclosed 指定所有连线（属于子模块的）将继承完全包含该子模块的模块线载模型。例如，如果设计者综合完全被模块 A（它完全被顶层所包含）包含的子模块 B 和 C，则子模块 B 和 C 会继承为模块 A 定义的线载模型。

最后一种模式 segmented 用于跨越层次边界的连线。在上例中，子模块 B 和 C 继承特定于他们的线载模型，而子模块 B 和 C 间的连线（在模块 A）会继承为模块 A 指定的线载模型。

```
    set_wire_load_model <top | enclosed | segmented>
dc_shell -t > set_wire_load_mode top
```

☺非常重要的是设计人员要准确地对他们设计的连线负载进行建模。太乐观或太悲观的线载模型会导致综合迭代的次数增加和加大布图后时序收敛的困难。总的来说，在布图前阶段使用稍微悲观的线载模型，这通过提供额外的可布线设计吸收的时序富余量来实现。

set_drive 和 set_driving_cell 用于模块的输入端口。set_drive 命令用于指定输入端口的驱动强度，它主要用于模块建模或芯片端口外驱动电阻。0 值表示最高驱动强度且通常用于时钟端口。相反的，set_driving_cell 用于对输入端口驱动单元的驱动电阻进行建模，这一命令将驱动单元的名称作为其参数并将驱动单元的所有设计规则约束应用于模块的输入端口。

```
set_drive <value> <object list>
set_driving_cell–cell <cell name>
                –pin <pin name> <object list>
dc_shell -t > set_drive 0 {CLK RST}
dc_shell -t > set_driving_cell –cell BUFF1 –pin Z [all_inputs]
```

set_load 将工艺库中定义的单位（通常为皮法，pf）上的容性负载设置到设计的指定连线或端口。它主要在布图前综合过程中设置模块输出端口的容性负载和往连线上反标注布图后提取的电容信息。

```
set_load <value> <object list>
dc_shell -t > set_load 1.5 [all_outputs]
dc_shell -t > set_load 0.3 [get_nets blockA/n1234]
```

设计规则约束或 DRC 由 set_max_transition、set_max_fanout 和 set_max_capacitance 命令组成。这些规则通常在工艺库中设置并且由工艺参数决定，为了得到能工作的芯片，不应违反这些规则。先前的 DC 版本（v97.08 和更早版本）甚至以差的时序为代价也要优先保证 DRC，而最新的 DC98 版本优先考虑时序需求而不是 DRC。

DRC 命令可用于输入端口、输出端口或 current_design。此外，如果在工艺库中设置的值不恰当或太乐观，也可在命令行使用这些命令来控制设计中的缓冲。

```
set_max_transition <value> <object list>
set_max_capacitance <value> <object list>
set_max_fanout <value> <object list>

dc_shell -t > set_max_transition 0.3 current_design

dc_shell -t > set_max_capacitance 1.5 [get_ports out1]
```

```
dc_shell -t > set_max_fanout 3.0 [all_outputs]
```

6.1.2　设计约束

设计约束描述了设计目标，它们可包括时序或面积约束。依据设计是如何约束的，DC 试图满足目标集。因为不实际的规范会导致面积增大、功耗增加和/或时序恶化，所以设计人员必须指定实际的约束。约束设计的基本命令如图 6-2 所示。

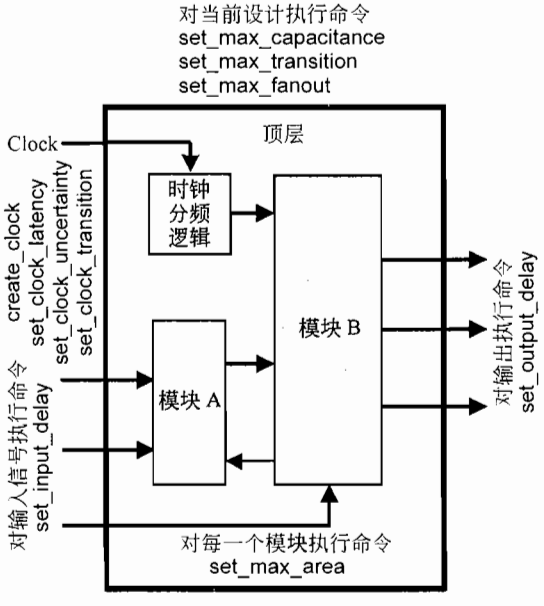


图 6-2　综合的设计约束

create_clock 命令用于定义有特定周期和波形的时钟对象。–period 选项定义时钟周期，而–waveform 选项控制时钟的占空比和起始边沿。这个命令用于引脚或端口对象类型。

下例指定端口 CLK 为“时钟”类型，其周期为 40ns，占空比为 50%。时钟正边沿开始于 0ns，下降边沿发生在 20ns。通过改变下降沿值，可改变时钟的占空比。

```
dc_shell -t > create_clock –period 40 –waveform [list 0 20] CLK
```

☺ 在某些情况下，模块可能只包含组合逻辑。为定义这一模块的延迟约束，可生成一个虚时钟并指定相对于虚时钟的输入

和输出延迟。为生成一个虚时钟，设计人员可将上述命令中的端口名（CLK，在上例）替换为–name <virtual clock name>（虚时钟名）。也可选择使用命令 set_max_delay 或 set_min_delay 来约束这样的模块。这些将在下节中详细讨论。

create_generated_clock 命令用于设计内部生成的时钟。这是一个非常强大的命令，直到最近还只存在于 PrimeTime 中。这一命令可用于描述作为主时钟函数的分频 / 倍频时钟。

```
create_generated_clock   –name <clock name>
                         –source <clock source>
                         –divide_by <factor> |   –multiply_by <factor>
                         …
```

set_dont_touch_network 是一个非常有用的命令，通常用于时钟网络和复位。这个命令用于在时钟引脚或端口上设置 dont_touch 属性。注意设置这一属性也会阻止 DC 为满足 DRC 而缓冲连线。此外，任何与被设置为“dont_touch”的连线相接触的门也将继承 dont_touch 属性。

```
dc_shell-t > set_dont_touch_network {CLK, RST}
```

☺ 假定你有一个时钟作为主时钟输入，并生成第二时钟的模块，如时钟除法器逻辑，在这种情况下，应在模块生成时钟输出端口上应用 set_dont_touch_network，这会帮助阻止 DC 缓冲时钟网络。

☺ 使用 set_dont_touch_network 命令应当谨慎行事。例如，如果一个设计包含门控时钟电路并且在时钟输入设置了 set_dont_touch_network 属性，就会阻止 DC 适当地缓冲门控逻辑，导致时钟信号 DRC 违例。这对门控复位同样成立。

set_dont_touch 用于在 current_design、单元、引用或连线上设置 dont_touch 属性。这一命令经常用于模块的层次化编译过程中，它也能用于阻止 DC 推断工艺库中的某种类型单元。

```
dc_shell -t > set_dont_touch current_design
dc_shell -t > set_dont_touch [get_cells sub1]
dc_shell -t > set_dont_touch [get_nets gated_rst]
```

☺ 例如，这个命令可用于包含备用门的模块。这命令将指示 DC

不要去打乱（或优化）备用门模块的实例。

set_dont_use 命令通常设置在.synopsys_dc.setup 环境文件中，这一命令有助于从工艺库中剔除用户不愿DC推断的某类单元。例如，使用上述命令，能从工艺库中剔除名称以“SDFF”或“RSFF”开头的触发器，如下所示：

```
dc_shell -t > set_dont_use [list mylib/SDFF* mylib/RSFF*]
```

set_input_delay 指定相对于时钟的信号输入到达时间。它用于输入端口，指定在时钟沿后数据稳定所需的时间。设计的时序规范通常包括这样的信息，如输入信号的建立/保持时间要求。如果给定设计的顶层时序规范，也可通过用自顶向下的编译方法或设计预算方法提取出设计子模块的这些信息，在第 7 章详细解释。

```
dc_shell -t > set_input_delay –max 23.0 –clock CLK {datain}
dc_shell -t > set_input_delay –min 0.0 –clock CLK {datain}
```

在图 6-3 中，相对于占空比为 50%周期、为 30ns 的时钟信号 CLK，为信号 datain 指定了 23ns 的最大输入延时约束和 0ns 的最小输入延时约束。换言之，输入信号 datain 的建立时间要求为 7ns，而保持时间要求为 0ns。

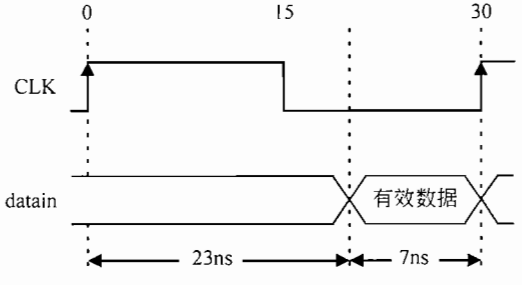


图 6-3　输入延时规范

如果-min 和-max 选项均被省略的话，最大和最小输入延时规范使用相同的值。

set_output_delay 命令用于在输出端口定义在时钟边沿到来之前数据有效所需时间，设计时序规范通常包含这一信息。如果给定设计的顶层时序规范，也可通过用自顶向下的编译方法或设计预算方法提取设计子模块的这些信息，在第 7 章详细解释。

```
dc_shell -t > set_output_delay –max 19.0 –clock CLK {dataout}
```

在图 6-4 中，相对于占空比为 50%、周期为 30ns 的时钟信号 CLK，为信号 dataout 指定了 19ns 的输出延时约束，这意味着在时钟沿后 11ns 数据有效。

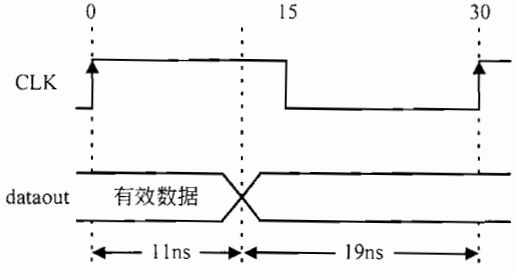


图 6-4 输出延时规范

☺ 在布图前阶段，有时有必要对选择的信号进行过紧约束以最大化建立时间，因此压缩额外时序余量以减少综合-布图迭代次数。为了达到这一目的，可在上述命令中指定过紧的约束值而“愚弄”DC。切记，过分地过紧约束设计将导致不必要的面积增大和功耗增加。

☺ 当布图后利用原位优化设计修正保持时间违例时，也可使用负值（如–0.5）提供额外时序余量，参见第 9 章。

set_clock_latency 命令用于定义在综合时估计的时钟插入延迟，这主要用于布图前综合和时序分析。所估计的延迟值是时钟树网络插入（在布图阶段）产生的延迟的近似值。

```
dc_shell -t > set_clock_latency 3.0 [get_clocks CLK]
```

set_clock_uncertainty 命令让用户定义时钟扭斜信息。基本上此命令用于给时钟的建立和保持时间增加一定的余量。布图前阶段可比布图后阶段增加更多的余量。

```
dc_shell -t > set_clock_uncertainty –setup 0.5 –hold 0.25 [get_clocks CLK]
```

☺ 强烈建议用户在布图前和布图后阶段都指定一定的余量，这样做的主要原因就是使芯片少受制造工艺偏差的影响。

set_clock_transition 命令由于一些原因没有得到应有的重视。然

而这是一个非常有用的命令，用于进行布图前综合和时序分析。这个命令使 DC 对时钟端口或引脚使用指定的转换值（这个值是固定的）。

```
dc_shell -t > set_clock_transition 0.3 [get_clocks CLK]
```

☺ 因为时钟网络伴有大的扇出，所以在布图前为时钟信号的转换时间设定一个固定值是必要的。这个命令使得 DC 能基于指定的时钟信号的转换时间为时钟连线驱动的逻辑计算实际的延时。这将在 6.3 节的“时钟问题”中作进一步的讨论。

set_propagated_clock 用于当设计已完成时钟树网络插入的布图后阶段。在这种情况下，将使用传统的延迟计算方法求出延时。

```
dc_shell -t > set_propagated_clock [get_clocks CLK]
```

6.2　高级约束

6.1 节描述的是常用约束，本节描述的是额外的一些设计约束。这些约束包括指定虚假路径、多周期路径、最大和最小延迟等。此外，本节还讨论了为额外优化而组合时序关键路径的过程。

然而，必须注意的是，过多使用时序例外，如虚假路径和多周期路径会对运行时间产生极大的影响。

set_false_path 用于指示 DC 忽视某一路径的时序或优化。确定设计中的虚假路径是关键。不这样做，会迫使 DC 优化所有路径以减少总的负松弛。结果，由于优化所有路径（也包括虚假路径），关键时序路径可能受到不利的影响。

用于这一命令的有效起点和终点分别是输入端口或时序元件的时钟引脚和输出端口或时序元件的数据引脚。另外，可使用开关 –through 进一步明确某一路径。

```
dc_shell -t > set_false_path –from in1 –through U1/Z –to out1
```

☺ 当时序关键逻辑由于虚假路径没能通过静态时序分析时，就使用这一命令。

set_multicycle_path 用于告知 DC 通过某一路径到达其终点所需的时钟周期数。DC 自动假定所有路径都是单周期路径，同时不必为了获取时序而试图优化多周期段。这将对邻近的路径和面积有直接的

影响。这命令也提供-through 选项来帮助在设计中分离多周期段。

```
dc_shell -t > set_multicycle_path 2    –from U1/Z          \
                                       –through U2/A       \
                                       –to out1
```

set_max_delay 定义某一路径按照时间单位所需的最大延迟。通常它用于只包含组合逻辑的模块。然而，它也用于约束多个具有不同频率时钟驱动的模块。这一命令优先于 DC 得出的时序要求。

☺对只包含组合逻辑的模块，既可生成一个虚时钟并相应地约束这个模块，也可使用这个命令约束从所有输入到所有输出的总延时，如下所示：

```
dc_shell -t > set_max_delay 5 –from [all_inputs] –to [all_outputs]
```

☺虽然 Synopsys 建议每个模块只定义一个时钟，而在某些情况下一个模块可包含多个时钟，每个时钟具有不同的频率。为了约束这种模块，通常可用 create_clock 和 set_dont_touch_network 命令来定义模块中的所有时钟。然而信号相对于各个时钟的输入延时赋值会变得很冗长。为避免这一情况，另一种方法就是用通常的方法定义第一个时钟（最严格的一个），而通过 set_max_delay 命令约束其他时钟，如下所示：

```
dc_shell -t > set_max_delay 0  –from CK2          \
                               –to [all_registers –clock_pins]
```

☺0 值表示在输入端口 CK2 和模块中所有触发器的输入时钟引脚之间需要零延迟值。另外，也可能需要为其他时钟应用 set_dont_touch_network。这一方法适用于包含门控时钟或复位的设计。

set_min_delay 是与 set_max_delay 相对的命令，它用于定义某一路径按照时间单位所需的最小延迟。与 sex_fix_hold 命令（在第 19 章描述）相配合，这个命令会指示 DC 在模块中添加延时以满足指定的最小时间单位。该命令也优先于通过 DC 推导出的时序要求。

```
dc_shell -t > set_min_delay 3 –from [all_inputs] –to [all_outputs]
```

group_path 命令用于将设计中的时序关键路径绑定到一起以进行代价函数计算。组合能使组合路径优先于其他路径。这一命令有不同的选项，包括关键范围和权重的规范。

```
dc_shell -t > group_path –to [list out1 out2] –name grp1
```

☺ 添加太多组合对编译时间有显著的影响，因此，只把它作为最后的手段使用。

☺ 当使用这个命令时须谨慎。用户会发现使用这一命令将增加设计中最差违例路径的延迟。这是由于 DC 使设计中的组合路径优先于其他路径。为改善整个代价函数，DC 将尽力优化组合路径，然而可能恶化另一组的最差违例的时序。

6.3　时钟问题

在任何设计中，综合的最关键部分是时钟的描述，总是有关于布图前后定义的问题。

过去传统上在时钟源旁放置大的缓冲器以驱动整个时钟网络。在版图中使用粗时钟主干以获得时钟网络延时的均匀分布和最小化时钟扭斜。虽然这一方法足以满足亚微米工艺的需求，但是在 VDSM 范围却完全无效。目前互连线 RC 延时为总延时的主要组成部分。这主要是由金属线宽缩小导致电阻增加造成的。用传统的方法对时钟进行建模，即使不是不可能的，也是困难的。

随着能够综合时钟树的复杂布图工具的出现，传统方法发生了很大的改变。由于布图工具具有单元布局信息，它们最适合综合时钟树，因而有必要在 DC 中描述时钟树，这样可效仿最终布图的时钟延迟和扭斜。

6.3.1　布图前

由于上述原因，最好在布图前阶段估计时钟树延时和扭斜。可用如下命令实现：

```
dc_shell -t > create_clock –period 40 –waveform { 0 20} CLK
dc_shell -t > set_clock_latency 2.5 CLK
```

```
dc_shell -t > set_clock_uncertainty –setup 0.5 –hold 0.25 CLK
dc_shell -t > set_clock_transition 0.1 CLK
dc_shell -t > set_dont_touch_network CLK
dc_shell -t > set_drive 0 CLK
```

上例中，指定 2.5ns 的延迟作为时钟信号 CLK 的总延迟。此外，set_clock_uncertainty 命令近似时钟扭斜。如上例所示，使用-setup 和-hold 选项能为建立和保持时间的不确定而指定不同的数。

此外，有必要指定时钟转换，这限定了时钟信号的最大转换值。通过单元的延迟受到输入引脚处信号的斜率和输出引脚处容性负载的影响。时钟网络一般驱动大量的终点。这表明虽然时钟延迟值是固定的，到达终点门的时钟信号的输入转换时间仍将会变慢。这会导致 DC 计算不真实的延时（为终点门），即使实际上布线后的时钟树确保快速地转换。

6.3.2 布图后

由于用户不需要担心时钟延迟和扭斜，定义布图后时钟相对比较容易，它们是所布的时钟树的品质所决定。

一些布图工具为 DC 提供了直接的接口。这为包含时钟树的布线后网表返回 DC 提供了一个平滑的机制。若这个信息不存在，用户应从布图工具中提取时钟延迟和扭斜信息。运用布图前方法（如前所述），这个信息可用于定义时钟延迟和时钟扭斜。然而，如果将网表导入 DC，那么以下命令可用来定义时钟。例如，

```
dc_shell> create_clock –period 40 –waveform [list 0 20] CLK
dc_shell> set_propagated_clock CLK
dc_shell> set_clock_uncertainty –setup 0.25 –hold 0.05 CLK
dc_shell> set_dont_touch_network CLK
dc_shell> set_drive 0 CLK
```

注意：其中没有 set_clock_latency 命令，而包含 set_propagated_clock 命令。由于在网表中插入了时钟树，因此用户传播时钟而不是将其固定为某个值。类似地，由于 DC 基于时钟树计算时钟网络的输入转换值，set_clock_transition 命令也不再需要。此外，也可定义一个小的时钟不确定值，以保证得到一个鲁棒设计。这样即使存在大的工艺偏差，设计仍能正常工作。

有些公司没有自己的布图工具，同时他们依赖外包进行布图。当然情况会随公司的不同有所变化。如果厂商为用户提供包含时钟树的布线后网表，那么就会用到上述方法。有时候，厂商不提供布线后网表，而只提供包含整个时钟网络（及设计）的点对点时序的 SDF 文件。在这种情况下，用户只需在原始的网表中定义时钟，并向原始的网表反标注 SDF 文件，而不用传播时钟。当进行静态时序分析时，时钟扭斜和延迟将由 SDF 文件决定。

6.3.3　生成的时钟

许多复杂的设计包含内部生成的时钟，其中一个例子就是时钟除法器逻辑，它用于从主时钟源生成不同频率的次级时钟。如果指定主时钟为时钟源，那么 DC 的一个限制因素是不能自动为生成的时钟生成一个时钟对象。

考虑图 6-5 中所示的逻辑。模块 clk_div 中的时钟除法器电路二分频主时钟 CLK 并生成驱动模块 A 的分频时钟。主时钟也用于定时模块 B 并在驱动模块 B 之前被内部（在模块 clk_div 中）缓冲。

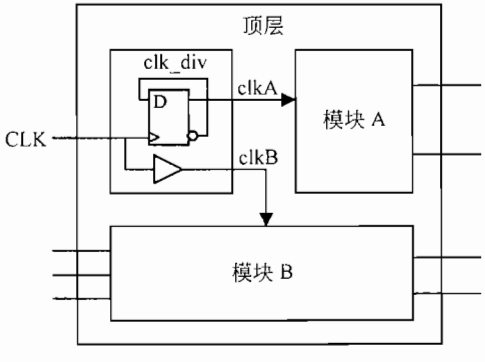


图 6-5　生成时钟描述

对于驱动模块 B 的时钟，通过 create_clock 命令在顶层 CLK 输入的时钟对象的赋值是足够的。这是因为 clkB 连线继承了在主源指定的时钟对象（通过缓冲器）。然而，clkA 不是如此的幸运。DC 无法通过整个连线传播时钟对象，因为在主源 CLK 上的时钟对象规范在寄存器（如阴影触发器所示）停止。为避免这种情况，应在 clk_div 模块的输出端口为 clkA 指定时钟对象。可用如下命令为上例指定时钟：

```
dc_shell > create_clock –period 40 –waveform { 0 20} CLK
dc_shell > create_clock –period 80 –waveform { 0 40}      \
                                    find(port, "clk_div/clkA")
```

或者，也可使用 create_generated_clock 命令来描述时钟，如下所示：

```
dc_shell -t > create_generated_clock –name clkA         \
                                     –source CLK        \
                                     –divide_by 2
```

6.4 综合实例

例 6.1 概述了本章中讲到的一些命令。

例 6.1

```
#______________________________________________________
#Design entry
 analyze –format verilog sub1.v
 analyze –format verilog sub2.v

 analyze –format verilog top_block.v

 elaborate top_block

 current_design top_block
 uniquify
 check_design
#______________________________________________________
#setup operating conditions, wire load, clocks, resets

 set_wire_load_model large_wl
 set_wire_load_mode enclosed
 set_operating_conditions WORST

 create_clock –period 40 –waveform [list 0 20] CLK
 set_clock_latency 2.0 [get_clocks CLK]
 set_clock_uncertainty –setup 1.0 –hold 0.05 [get_clocks CLK]
```

```
set_dont_touch_network [list CLK RESET]

#_____________________________________________________________
#Input drives

set_driving_cell –cell [get_lib_cell buff3] –pin Z [all_inputs]
set_drive 0 [list CLK RST]

#_____________________________________________________________
# Output loads

set_load 0.5 [all_outputs]

#_____________________________________________________________
#Set input & output delays

set_input_delay 10.0 –clock CLK [all_inputs]
set_input_delay –max 19.0 –clock CLK {IN1 IN2}
set_input_delay –min –2.0 –clock CLK IN3

set_output_delay 10.0 –clock CLK [all_outputs]

#_____________________________________________________________
#Advanced constraints

group_path –from IN4 –to OUT2 –name grp1

set_false_path –from IN5 –to sub1/dat_reg*/*

set_multicycle_path 2 –from sub1/addr_reg/CP      \
                      –to sub2/mem_reg/D

#_____________________________________________________________
#Compile and write the database

Compile

current_design    top_block

write –hierarchy –output top_block.db
write –format verilog –hierarchy –output top_block.sv
```

```
#_____________________________________________________
#Create reports

 report_timing –nworst 50
```

6.5 小结

本章介绍了所有DC中使用的基本和高级命令以及增强综合过程的许多技巧，重点放在设计人员在深入 VDSM 工艺时所面对的实际问题上。

本章用单独的一节讨论与时钟相关的问题。此小节描述了在布图前后指定时钟的各种有用的技巧。此外还也包括大多数设计都存在的生成时钟规范的论题。最后，给出指导用户进行复杂和成功综合的DC 脚本示例。

第 7 章
优化设计

在理想情况下，将满足所有时序要求，并且占有面积最小的综合后的设计视为是完全优化的。为达到这一目的，必须理解综合过程的行为。

本章引导读者成功优化设计以获得最佳的结果。

7.1 设计空间探索

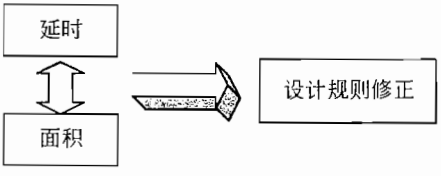
为获得面积最小同时最大化设计的速度需要相当数量的实验和反复综合。分析设计速度和面积，并以最小的面积取得最快的逻辑过程被称为设计空间探索。

多种因素影响优化过程，其中主要是编码风格。编码时，设计人员通常注意的是设计的功能，而没有考虑综合指导，关于这一点，先前已在第 5 章解释过了（这是现实，我们不得不与之共处）。在稍后的阶段对 HDL 代码进行修改以利于综合过程。实际上，HDL 通常是固定的，并且只进行小的改动，因为大变动可能影响其他模块或测试基准。由于这个原因，所以修改 HDL 代码以有助于综合是不值得的。

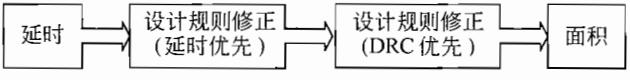
为了设计空间探索，假定 HDL 代码是不变的，则通过综合和优化以最小化面积和满足目标时序要求是设计者的责任。

如图 7-1 所示，从 DC 的 98 版本（或 DC98）开始，先前的编译流程发生了改变，时序优先于面积。DC98 和以前版本的另一个主要区别是，DC98 进行编译以减少“总的负松弛”，而不是“最差负松弛”。

DC98 的这种性能可产生更好的时序结果，但对面积有一些影响。另外，在以前的版本中面积最小化是自动处理的，而 DC98 及以后的版本要求设计人员明确指定面积约束。通常即使没有指定面积约束（即在默认情况下），一些面积清除也会进行，但包括面积约束可得到更好的结果。



在 DC98 之前编译流程



DC98 编译流程

图 7-1　DC98 变化

虽然延迟优先于面积，但给 DC 提供实际的约束是极其重要的。一些设计人员在进行自底向上的编译时没能意识到这一点而过分地约束设计，这导致 DC 为满足不实际的时序目标而增大逻辑。对 DC98 而言更是如此，因为它致力于总的负松弛的减少。约束和面积之间的关系如图 7-2 所示，它强调了加紧约束时面积会显著的增加。

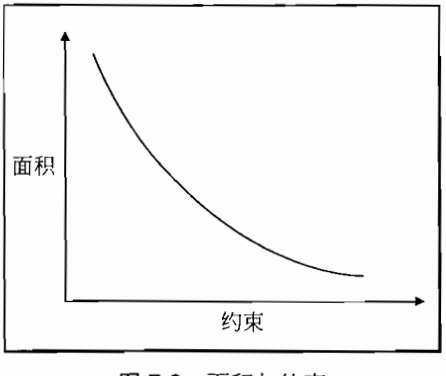


图 7-2　面积与约束

另一个变化约束表示如图 7-3 所示。它描述了约束和通过设计的延迟之间的关系。由图可以看出：加紧约束则逻辑的实际延迟就会减

少；而放松约束，则通过设计的延迟就会增加。左侧线的水平部分表示约束是如此的紧，以致进一步加紧约束并不会减少延迟。类似地，右侧线的水平部分表示完全放松约束，不会进一步增加延迟。

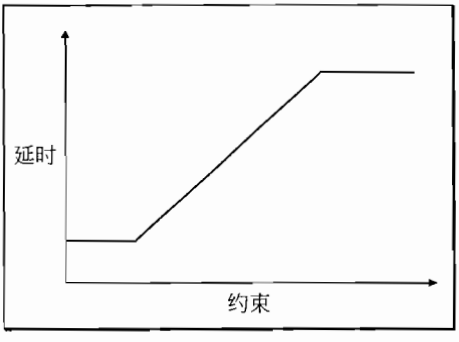


图 7-3　延迟与约束

为进一步解释这一概念，参考示意图 7-4。对过度约束的设计，DC 试图综合“垂直逻辑”以满足紧时序约束。然而，如果不存在时序约束，综合后的设计会产生“水平逻辑”，从而违反了实际的时序规范。

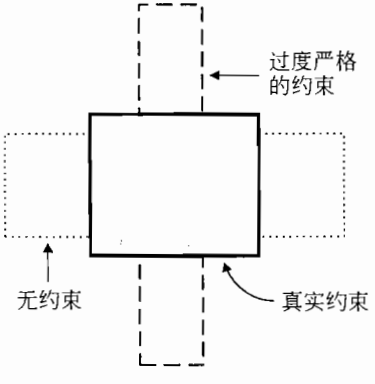


图 7-4　垂直与水平逻辑

这里的想法是通过指定实际的时序约束来找到一个共同点。建议少量地（大约比需要的紧 10%）过度约束设计以避免过多的综合布图迭代。这会得到最小面积的设计，同时仍满足时序规范。因为这个原因，选择下面几节所述的正确编译方法。

7.2 总的负松弛

7.1 节简要介绍了术语“总的负松弛”(或简记为 TNS)。随着 DC98 的出现，它变得非常重要，并且设计人员需要理解这一概念以进行成功的逻辑优化。

在 DC98 版本之前，DC 会基于“最差负松弛”(或 WNS) 优化逻辑。WNS 定义为一个信号从起点到终点穿过某一路径的时序违例(或负松弛)。在编译过程中，为了减少模块总的违例，DC 会一个个地减少 WNS。因为这个原因，组合路径和指定时序关键段的临界范围被认为是必要的。

DC98 不仅使延迟优先于面积，而且还以 TNS 为目标而不是 WNS。为理解 TNS 的概念，考虑如图 7-5 所示的逻辑图，这种情况下 WNS 为–5ns，从 RegA 到 RegB。TNS 是所有每个终点的 WNS 的总和，并且在这种情况下等于–8ns，也就是到 RegA 的 WNS 加上到 RegB 的 WNS。

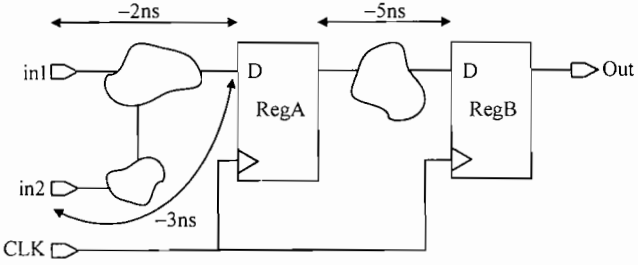


图 7-5 总的负松弛

使用这一方法有一些优点，其中主要是该方法比以前的方法产生更少的时序违例。另一个优点就是，当使用自底向上的编译方法时，子模块的关键路径在顶层不会看做是关键的，减少整个设计的 TNS 能最小化这个影响。提供给时序驱动布图工具的违例路径的数目越少，能使得的综合和布图间的迭代次数也越少。

虽然优先于 WNS 减少 TNS 可产生更少的时序违例，但它确实会对总面积产生影响。建议无论进行何种优化，都设定面积约束。默认情况下，DC98 使 TNS 优先于面积。面积优化只发生在具有正松弛的

路径。为了使面积优先于 TNS，可用如下的命令：

```
dc_shell> set_max_area 0 –ignore_tns
```

7.3　编译策略

Synopsys 建议采用如下的编译策略，它完全依赖于你的设计是如何组织和定义的，用户自己为设计选择最适合的编译策略。

a）自顶向下层次化编译方法。

b）时间预算编译方法。

c）Compile-characterize-write-script-recompile(CCWSR)方法。

d）设计预算方法。

7.3.1　自顶向下层次化编译

在 DC98 发行之前，自顶向下层次化编译方法通常用于综合很小的设计（少于 10K 门）。使用这种方法，读取整个设计来编译。基于设计规范，只在顶层应用约束和属性。虽然这种方法提供了简单的按钮式综合方法，但它需要占用大量内存并只适用于非常小的设计。

DC98 版本给 Synopsys 提供了通过同时处理更大模块（＞100K）而综合百万门设计的能力。对依赖于设计风格（单时钟等）和其他因素的一些设计，这的确是一个可行的方案。通过把子模块组合在一起并打平它们以改善时序，用这个技巧一次可综合更大的模块。

这种方法的优缺点总结如下。

7.3.1.1　优点

a）只需要顶层约束。

b）由于在整个设计上优化而得到更佳的结果。

7.3.1.2　缺点

a）长的编译时间（虽然 DC98 比以前的版本快）。

b）对子模块的增量改变需要完全的重新综合。

c）如果设计包含多个时钟或生成的时钟，它就不能很好地执行。

7.3.2 时间预算编译

综合的第二种编译方法被命名为时间预算策略。如果设计已经按每个模块定义的时序规范进行了适当的划分，也就是说，设计人员已对整个设计进行了时间预算，包括模块间时序需求，这个策略是有用的。

设计者为设计的每个模块手动指定时序需求，从而为每个模块生成多个综合脚本。综合通常自底向上进行，也就是说从最底层开始并上升至设计的最顶层。这种方法以中等至超大规模设计为目标，并且不需要大量的内存。

考虑如图 7-6 所示的设计，顶层模块包括模块 A 和模块 B。这两个模块的规范定义得很明确并且可直接转换为 Synopsys 约束。对这样的设计，时间预算编译策略是最为合适的。

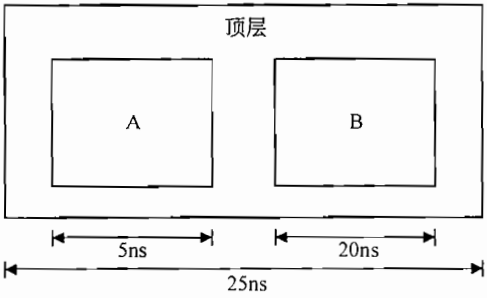


图 7-6 适合时间预算编译策略的设计

这一方法的优缺点如下。

7.3.2.1 优点

a）由于是单个脚本所以更易于控制设计。

b）对子模块的增量改变，不需要对整个设计进行全部的重新综合。

c）不受设计风格的影响，如易于控制多个时钟和生成时钟。

d）由于以单个模块为目标并对其优化的灵活性，因此通常有良好的品质。

7.3.2.2　缺点

a）更新并维护多个脚本是很乏味的。

b）顶层所见的关键路径在底层可能不是关键的。

c）为了修正 DRC，可能需要进行增量编译设计。

图 7-7 描述了适合于这一策略的目录结构和数据组织。为自动化综合过程，使用 makefile（参见附录 B）。makefile 指定了每个模块的依赖关系，并使用用户定义的脚本（script）（保存在 script 目录）来编译整个设计，由最底层开始到最顶层结束。每个模块综合后，结果被自动地移到它们各自的目录中。makefile 中使用的变量定义在用户的.cshrc 文件中，例如$SYNDB 可定义为：/home/project/design/syn/db。

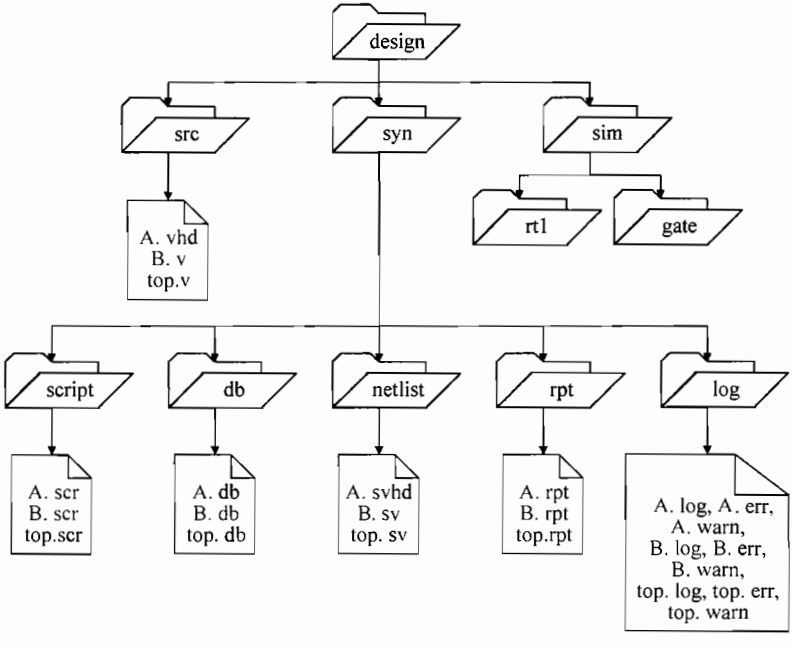


图 7-7　目录结构

7.3.3　Compile-Characterize-Write-Script-Recompile

这种方法对不具有明确的模块间规范的中等至超大规模设计有用。这一方法不受硬件存储的限制且允许在模式间进行时间预算。

这一方法要求在设计的顶层应用约束，并且要预编译每个子模块。然后使用顶层约束来表征子模块。这实际上是从顶层向子模块传播所要求的时序信息。在已刻画的子模块上执行 wirte_script 命令为每个子模块生成约束文件，然后这些约束文件用于重新编译设计的每个模块。

虽然这一方法通常产生好的结果，但仍然建议设计人员使用设计预算方法，该方法将在下一节中介绍。

7.3.3.1 优点

a）较少的内存。

b）由于设计子模块间的优化，因此可产生好的结果。

c）产生可由用户修改的单个脚本。

7.3.3.2 缺点

a）生成的脚本可读性不好。

b）综合遭受乒乓效应。换言之，取得模块间的收敛可能会很困难。

c）在较底层模块的改变通常需要整个设计的重新综合。

d）如果过度约束模块，需要较长的运行时间。

7.3.4 设计预算

到目前为止，对于模块间定义规范不好的设计，这个方法是最适合的编译策略。这种方法自动将顶层设计规范分配到底层模块。设计预算由 DC 或 PT 调用，尽管它也可通过输入 budget_shell 来调用。这一命令窗口使用 Tcl 界面并且不能使用老的 Tcl DC-shell 命令。如果想要利用 PT 的 GUI 界面，建议使用 PT。

不能在 RTL 阶段直接使用这一方法。在预算前，必须首先将设计综合到映射后的门级网表。一旦综合了设计，可在整个设计上运行预算器并为子模块生成脚本。预算的层次级数完全受用户控制，换言之，预算器为用户定义的任何层次级数分配预算。带有准确约束的生成脚本随后用于并行地（减少运行时间）重新综合每一个模块。

这是一个非常强大的门级优化的方法，然而，它是一个迭代过程。用户完全控制脚本及优化的程度。生成的脚本能进一步调整以适应个

体的需要。此外，为了生成更准确的约束，甚至可在布图后使用这种方法。这是通过预算反标注后的设计来完成的。

7.3.4.1　优点

a）提供整个设计的准确约束，因而有更好的 QOR。

b)没有遭受乒乓效应(如果 Characterize-compile 方法中的那种)。

c）通过提供进行并行编译的能力以节省运行时间。

d）提供定制脚本以适合个体需要的能力。

e）脚本可在确立阶段（GTECH 阶段的设计）后使用。

f）较少的内存。

7.3.4.2　缺点

a）不能对 RTL 本身进行预算，设计必须是一个结构化的门级网表。

b）不能使用最佳和最差工作条件进行预算。

c）迭代过程（虽然这不是一个严格的限制)。

说明这一策略的脚本示例如下所示（用黑体强调预算命令)：

```
pt_shell > read_verilog mydesign.sv
pt_shell > source constraints.scr #Top level constraints
pt_shell > allocate_budgets –levels 2 –write_context –format dctcl
```

在上例中，allocate_budgets 命令调用 Design Budgeter，它为当前设计及其以下两个层次的每个子模块分配预算。选项–write_context 指示预算器生成脚本。选项–format 指定生成脚本的格式。其允许值为 ptsh（Primetime Tcl 格式)、dctcl（DC Tcl 格式：dc_shell-t）和 dcsh (DC 格式：dc_shell)。ptsh 格式为默认值。

其他一些命令也可使用这个方法。相关详细信息，建议用户参考 Design Budgeting User Guide（设计预算用户指南)。

7.4　多个实例解析

在进行优化之前，需要解析设计中子模块的多个实例。由于 DC 直到设计中存在的多个实例被解析后才允许编译，所以这是必需的

一步。

为了更好地解释一个模块多次例化的概念，考虑如图 7-8 所示设计的结构。假设已选定时间预算编译策略并且已经单独综合了 moduleA。现在正编译 moduleB，它对 moduleA 进行了两次例化，分别为 U1 和 U2。DC 提示 moduleA 在 moduleB 中例化两次的错误信息并中止编译。有两种解决这个问题的推荐方法，即可在读取 moduleB 之前给 moduleA 赋 dont_touch 属性，也可唯一化（uniquify）moduleB。uniquify 是一个 dc_shell 命令，它实际上生成多个实例的唯一定义。在这一情况下，它将生成 module A_U1 和 moduleA_U2（在 Verilog 中），分别对应实例 U1 和 U2，如图 7-9 所示。

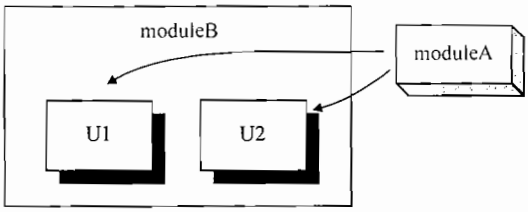


图 7-8 非唯一化设计

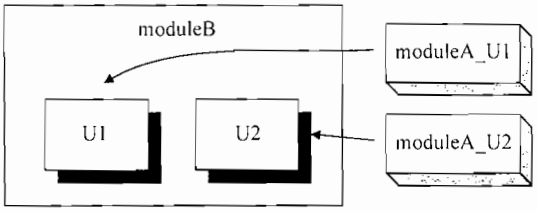


图 7-9 唯一化设计

无论选择何种编译方法，建议总是要进行唯一化设计。当在布图过程中计划进行时钟树综合时，这个建议的原因就变得明显了。这在第 9 章有详细说明。

7.5 优化技巧

本节介绍用于微调设计的各种优化技巧。在开始这个主题之前，了解 DC 使用代价函数来优化设计是重要的。但由于 DC 参考手册完全包括了这个论题，因此本书不涉及这个论题。本书更多地专注于实

际的优化技巧而不是用于计算代价函数的数学算法。只假定 DC 计算基于设计约束和 DRC 的代价函数来优化设计。

7.5.1　编译设计

compile 命令进行设计或模块的编译。这一命令将 HDL 代码映射到指定的目标库的门。DC 为这一命令提供了许多选项以完全控制设计的映射优化。

命令语法及最常用的选项如下所述：

```
compile –map_effort <low | medium | high>
        –incremental_mapping
        –in_place
        –no_design_rule | –only_design_rule
        –scan
```

在默认的情况下，compile 使用–map_effort medium，对大多数设计而言，它通常能得出理想的结果。它在构建和打平属性时使用默认的设置，这将在下一节中介绍。只有当通过编译达不到目标时，才使用–map_effort high。这个选项使得 DC 尽最大努力围绕关键路径进行逻辑的重新构造和重新映射来满足指定的约束。注意，这通常需要很长的编译时间。

由于-incremental_mapping 选项只用在门级，因此它只用在初次编译之后（也就是说，设计已映射到工艺库中的门）。这是一个非常有用和常用的选项，它通常用于改善逻辑的时序和修正 DRC。在增量编译时，DC 进行各种映射优化以改善时序。虽然 Synopsys 声称，在设计约束方面，设计结果不会变糟，且只可能提高；但是在极少数情况下，使用上述选项实际上能恶化时序目标，因此建议用户进行实验并自己判断。虽然如此，当在设计的顶层修正 DRC 时，这个命令的有效性是显然的。为此，可在增量编译时使用-only_design_rule 选项。这避免了 DC 进行映射优化并只专注于修正 DRC。

-no_design_rule 选项不常使用，且如名称所表示，它指示 DC 避免修正 DRC。当不想为最初的几次编译浪费时间修正 DRC 违例时，可使用这一选项。在以后的阶段，生成约束报告并重新增量编译以修正 DRC。这显然是一个令人厌烦的方法，建议用户自己判断。

取得布图后时序收敛，有时有必要调整逻辑大小以修正时序违例。-in_place 选项提供调整门大小的能力。这一选项控制设计人员用于控制逻辑缓冲的各种开关。这一选项的用法将在第 9 章进行详细介绍。

-scan 选项使用 DC 的测试准备编译特性。这一选项指示 DC 直接将设计映射到扫描触发器，而不是在用对应的扫描触发器替换它们之前综合到普通触发器，以形成扫描链。使用这个特性的一个优势是，既然扫描触发器通常具有不同于它们非扫描对应触发器（或普通触发器）的时序，使用这个技巧使得 DC 在综合时考虑到扫描触发器的时序。这个过程产生时序正确的、优化的扫描插入逻辑。

7.5.2 展平和构造

在我们开始讨论之前必须注意，这里使用的“展平”（flattening）术语没有暗含“移除层次”。展平是指将逻辑化简为两级与/或表示的标准术语。DC 用这一方法删除所有中间变量和括号（用布尔分配律）以优化设计。这一选项的默认设置为“false”。

如图 7-10 所示，设计优化分为两个阶段进行。起初通过构造和展平设计来进行逻辑优化，然后使用映射优化技巧将所得的结构映射到门。flatten 和 structure 属性的默认设置如表 7-1 所示。

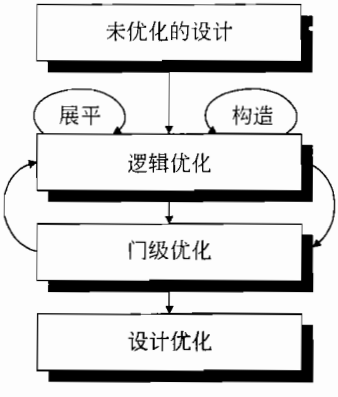


图 7-10 优化步骤

表 7-1 flatten 和 structure 属性的默认设置

属 性	值
flatten	false
structure	true
structure(timing)	true
structure(boolean)	false

如表 7-1 所示，只有当激活时才进行展平（set_flatten true）设计和布尔优化（set_structure –boolean true）。

7.5.2.1　展平

展平对于非结构化设计很有用（如随机逻辑或控制逻辑），因为它除去中间变量并使用布尔分配律移除所有括号。它不适用于包含结构化逻辑的设计，如超前进位加法器或乘法器。

展平得到一个两级（积之和形式）的垂直逻辑，也就是说，在输入和输出间有很少的逻辑层。由于最小化输入和输出间的逻辑层次，因此这通常能得到更快的逻辑。在最终工艺映射优化之前，依据展平设计的形式和使用的 effort 类型能够构造展平的设计。这是推荐的方法，并且应当执行此方法以减小面积，因为展平设计可对设计的面积有重大影响。需要记住的是，如果你用–effort high 选项展平设计，那么 DC 就不能构造设计，因此应谨慎使用这一属性。

通常用默认设置编译设计，因为它们在大多数情况下都能很好的完成。没有通过时序目标的设计可以进行展平，并在第二阶段进行构造（在默认情况下）。如果设计仍没有通过时序目标，就关掉构造只进行展平。也可以尝试反转赋值的相位，它有时产生不错的结果。这通过将 set_flatten 命令的–phase 选项设置为“true”来完成。分别由反转等式和等式的非反转形式产生的逻辑可通过 DC 进行对比。

对于一个层次化设计，flatten 属性只设在 current_design 上。所有的子模块都不继承这一属性。如果你想展平子模块，那么你必须明确地使用-design 选项指明。flatten 属性的语句及最常用的选项为：

```
set_flatten <true | false>
        –design <list of designs>
        –effort <low | medium | high>
        –phase <true | false>
```

7.5.2.2　构造

构造用于包含规则结构逻辑的设计，如超前进位加法器，在默认情况下，它只为时序使能。在构造时，DC 添加可分解的中间变量。这使得逻辑共享，它反过来会导致面积的减少。例如，

构造前：	构造后：
P= ax + ay + c	P = aI + c
Q= x + y + z	Q = I + z
	I = x + y

值得注意的是，构造产生的共享逻辑对逻辑的总体延迟产生影响。没有指明时序约束（或构造时不考虑时序），生成的逻辑通常导致大的跨模块边界延迟。因此，建议除了使用默认设置以外，还要指定实际的约束。

构造分为两种：时序（默认）和布尔优化。后者是减少面积的有用方法，但对时序有更大的影响。布尔类型优化的最佳选择是非关键时序电路，例如随机逻辑结构和有限状态机。顾名思义，这一算法要用到布尔逻辑优化以减少面积。在 v1997.01 版本以前，DC 用一个不同的算法进行布尔优化。此后 Synopsys 引进了另一个更为有效且需要更少运行时间的算法，这个算法在自动测试图形生成（ATPG）方法基础上操作逻辑网络。为了使能这个算法，必须设置如下变量为“true”：

```
compile_new_boolean_structure = true
```

和展平一样，set_structure 命令只应用于 current_design。这一命令的语法及最常使用的选项如下：

```
set_structure <true | false>
              –design <list of designs>
              –boolean <true | false>
              –timing <true |false>
```

通常，用默认设置编译的设计可产生令人满意的结果。然而，如果你的设计是非时序关键的并且只是最小化面积，那就设置面积约束（set_max_area 0）并进行布尔优化。对所有其他情况，构造只与时序相关。

7.5.3 消除层次

在默认情况下，DC 保持设计的原有层次。层次实际上是一个逻辑边界，它防止 DC 跨边界进行优化。许多设计人员不知出于何种原因生成不必要的层次，这不仅使综合过程变得更麻烦，而且也导致综合脚本数目的增加。如前所述，DC 在逻辑边界内进行优化。设计中不必要的层次限制了 DC 在边界内优化而不能跨层次进行优化。

考虑如图 7-11(a)中所示的逻辑。顶层（模块 T）包含两个模块 A

和 B。模块 A 输出端的逻辑和在模块 B 输入端的逻辑被模块边界所分开。模块 A 和 B 的两次单独地优化可能得不到最优解。

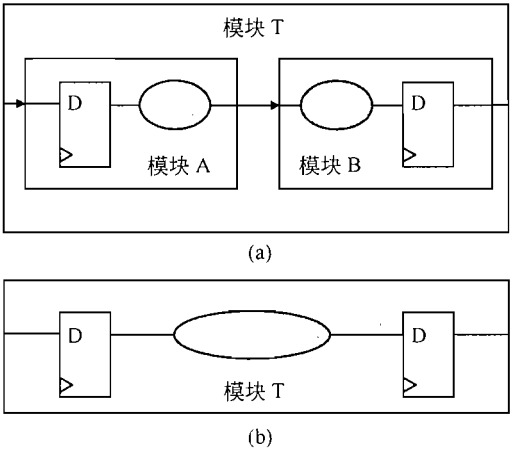


图 7-11　取消设计分组以改善时序

如图 7-11(b)所示的合并模块 A 和模块 B（也就是去除边界），两个逻辑泡可作为一个进行优化，得到一个更优的解。设计人员（而非 Synopsys）称这一过程为“展平”设计。

为完成这个过程，可使用如下命令：

```
dc_shell > current_design BlockT
dc_shell > ungroup –flatten –all
```

7.5.4　优化时钟网络

优化时钟网络是最难执行的操作之一。这是由于当我们向下进入 VDSM 工艺时，金属电阻急剧增加，从而引起由时钟引脚输入到寄存器的巨大延迟。当不需要定时数据时，低功耗设计技术也需要门控时钟以最小化晶体管的翻转。这个技术使用输入为时钟和使能（用于使用或抑制时钟源）的门（如与门）。

以前的方法包括在芯片的顶层靠近时钟源的地方放置一个足以驱动设计中所有寄存器的大的缓冲器。粗大的主干和分支（如鱼骨结构）分布到整个芯片以减少时钟扭斜并最小化 RC 延迟。虽然这个方法对 0.5μm 及以上的工艺是令人满意的，但它却绝对不适用于 VDSM 工艺（0.35μm 及更小）。上述方法也意味着综合–布图迭代次数的

增加。

随着复杂的布图工具的出现，现在在布图工具内综合时钟树已成为可能。时钟树方法最适合 VDSM 工艺，虽然功耗是一个问题，但是时钟延迟和扭斜与大缓冲器方法相比都是最小的。在布图过程中，在单元布局之后，在布线前先进行时钟树综合（CTS）。这使得布图工具知道在布局规划中寄存器的准确布局位置。随后最优地放置缓冲器以最小化时钟扭斜，这对布图工具来说很容易。因为优化时钟是增加综合布图迭代的主要原因，在布图阶段进行 CTS 以减少这一循环。

在布图前的综合阶段中，我们仍必须优化时钟。不能假设布图工具给我们一个神奇的时钟树来解决所有的问题。记住，初始网表优化得越多，从布图工具得到的结果越佳。

那么在综合阶段，我们该如何优化时钟网络呢？通过给时钟引脚设置 set_dont_touch_network，可确信 DC 不会缓冲网络以修正 DRC。这个方法对大多数不包含门控时钟逻辑的设计工作得很好。但是如果存在门控时钟，会怎样呢？如果你在门控的时钟上设置 set_dont_touch_network，DC 就不会增大门的尺寸（假定一个二输入与门）。这是因为 set_dont_touch_network 传遍所有组合逻辑（在这种情况为与门），直到它遇到一个终点（在这种情况下为寄存器输入时钟引脚）。这也引起组合逻辑继承 dont_touch 属性，导致未优化的门控逻辑可能违反 DRC，从而影响整个时序。

例如，假定与门的时钟输出扇出许多寄存器且 DC 为与门推断一个最小的驱动强度，使输入给寄存器的输入转换时间变缓，导致时钟线网可怕的延迟。为避免这一情况，可除去 set_dont_touch_network 属性并进行增量编译。这会增大与门的尺寸并在与门输出和终点间插入额外的缓冲器。虽然这一方法看似理想，却也有一些缺点。首先，它需要很长的时间来完成增量编译并且偶尔会得到次优结果。其次，它需要大量的预见，例如需要在所有其他线网上应用 set_dont_touch_network 属性（不需要时钟树的复位和扫描相关的信号）。

第二种方法是使用 report_net 命令找出设计中所有的高扇出线网并使用 balance_buffer 命令点到点对其进行缓冲（参考 DC 参考手册这一命令的实际语法）。因为 balance_buffer 命令不考虑时钟扭斜，它不应用作时钟树综合的另一选择。

第三种方法是使用 compile-in_place 进行原地优化（IPO），同时

将 compile_ok_to_buffer_during_inplace_opt 开关设为“false”。这可避免 DC 插入额外的缓冲器并且只增大与门的尺寸。

必须注意的是上述方法完全同设计相关。上面提供了可用于时钟网络优化的各种方法。有时，你会发现必须进行上面所有的方法以获得最佳结果，而其他时候一种方法就足够了。

不管使用何种方法，你也应考虑布图过程中想要做什么。对不带门控时钟的设计，较好的办法是在布图阶段做 CTS。对其他有门控时钟的设计，必须仔细分析设计（综合前和综合后）中的时钟，并采取适当的行动。这也可能包含为每组寄存器在与门后插入时钟树（在布图过程中）。大多数布图工具厂商已经意识到这个问题并提供了进行门控时钟树综合的各种方法。

7.5.5　面积优化

在默认情况下，DC 尽力优化设计的时序。非时序关键但面积集中的设计应进行面积优化。这可通过有明确面积要求而无时序约束设计的初次编译来实现。换言之，不使用时序约束，只用面积要求综合设计。

另外，可选择在高驱动强度门上指定 dont_use 属性来排除它们。去除高驱动强度门的原因是它们通常用于加速逻辑以满足时序，然而它们尺寸较大，减少它们的使用可大大减少面积。

一旦设计被映射到门，应再次指定时序和面积约束（正常综合）并增量重新编译设计。增量编译可确保 DC 保持以前的结构并不会不必要地膨胀逻辑。

7.6　小结

优化设计是最耗时且最难的任务，由于它特别依赖于多种因素，如 HDL 编码风格、逻辑类型、约束等。本章介绍了高级优化方法以及它们是如何影响综合过程的。

本章还详细介绍了改变设计约束对时序和面积的影响。反复说明的是，最佳结果是通过提供给 DC 实际的约束取得的。过度约束设计会导致大的面积和次优的结果。

随着 DC98 的引入，优化流程发生了改变，它更多地集中于时序，而不是面积。无论如何建议指定面积约束，在默认情况下，在编译的末尾总是进行面积清除。DC98 通过最小化 TNS 进行时序优化。DC98 之前，以减少每个终点的 WNS 进行时序优化。虽然 TNS 优化对设计的整个面积有一定影响，但它能提供更好的结果。

本章介绍了各种编译策略及自动化这一过程的示例。为了成功地优化设计，你可以选择一种方法或混合使用这些策略以得到所需结果。本章也介绍了所有这些策略自身的优缺点。选择最适合的设计策略。

本章还用一个单独的小节介绍唯一化设计。虽然有些设计人员认为这一步并不需要，但推荐总是进行唯一化设计，其原因将在第 9 章介绍。

最后，讨论了其他优化步骤，着重强调了“怎样生成最佳综合网表”；同时介绍了包括时钟网络优化和优化设计面积的多种技巧以及建议使用的方法。

第 8 章
可测性设计

可测性设计或 DFT 技术在 ASIC 设计人员中日益得到推广。这些技术提供了全面测试加工后器件的品质和覆盖率的方法。

传统上，认为可测性在后面，它只在设计周期的最后才实现。这种方法通常提供最小的覆盖率且往往导致不可预见的问题，从而延长了周期。在设计初期结合可测性的特性是最终的解决方案，称为可测性设计。

8.1 DFT 类型

包括 Synopsys 在内的众多厂商，提供在设计中包括可测性的解决方案。通过包括在 DC 套装工具中的 DFT Compiler（DFTC），Synopsys 为 DC 增加了 DFT 功能。如今主要用到的 DFT 技术如下：

a）扫描插入

b）存储器 BIST 插入

c）逻辑 BIST 插入

d）边界扫描插入

在上面四种技术中，扫描插入和逻辑 BIST 插入是最复杂和最具挑战性的技术，因为它们包括了为完全覆盖设计而需要解决的各种设计问题。

8.1.1 存储器和逻辑 BIST

不幸的是，Synopsys 不提供任何自动存储器或逻辑 BIST（内建

自测试）生成解决方案。由于这个原因，本节不对这两个技术进行详细的介绍。但是由于有些厂商提供关于这两个技术完整的解决方案，所以本节对存储器和逻辑 BIST 的主要功能进行了简单介绍，帮助设计人员了解这些有用的技术。

存储器 BIST 由控制器逻辑组成，它使用各种算法生成输入图形并用于激励设计的存储单元（如 RAM）。基于存储单元的大小和配置自动生成 BIST 逻辑。它通常以可综合的 Verilog 或 VHDL 的形式插入 RTL 源代码并和存储单元连接起来。一经触发，BIST 逻辑就生成基于预定义算法的输入图形（pattern）来完全检测存储单元。输出结果反馈给 BIST 逻辑，在那里使用比较器比较进入的和读出的数据。比较器的输出生成一个通过/失败信号来表示存储单元的真实性。

同存储器 BIST 相似，逻辑 BIST 使用相同的方法但却以设计的逻辑部分为目标。逻辑 BIST 使用随机图形生成器激励设计中的扫描链。输出是与仿真的签名进行比较的压缩的签名。如果被测物（DUT）的签名和仿真的签名相匹配，则被测物通过，否则失败。使用逻辑 BIST 的主要优势是，它为测试工程师免除了生成作为 DUT 输入的大量扫描向量的需要，这节省了大量的测试时间。缺点是，只为了测试而在设计中包含了额外的逻辑（意味着增加了面积）。

8.1.2 边界扫描 DFT

JTAG 或边界扫描主要用于测试电路板的连接，而不必从电路板上拔取芯片。DC 也可以直接生成 JTAG 控制器和周边逻辑。由于整个过程相当简单且大多为自动的，因此边界扫描插入是微不足道的。本章的目的只集中于扫描插入技术及其问题，建议读者参考 Design Compiler 参考手册中的边界扫描插入技术。

8.2 扫描插入

扫描是设计工程师测试芯片缺陷（如固定故障）最广泛使用的 DFT 技术之一。对大多数的设计而言有可能获得非常高的故障覆盖率（通常高于 95%）。

扫描插入技术涉及用包括只用于测试的内建逻辑的特殊的触发

器替换设计中所有的触发器。最普遍使用的结构是多路选择触发器，这种结构类型在 D 触发器的输入端加上一个二输入多路选择器。多路选择器的选择信号决定器件的模式，也就是说，它能使多路选择器或者在正常模式（以正常数据输入的功能模式）或者在测试模式（以扫描数据输入）。这些扫描触发器被连接起来（使用多路选择的扫描数据输入）以形成一条扫描链，其功能就像一个串行移位寄存器。在扫描模式中，一组图案应用在原始输入，并通过扫描链移出。如果正确地完成，那么这一技术为芯片中的所有组合和时序逻辑提供一个非常高的覆盖率。

和多路选择类型触发器一起的其他可用结构为 lssd 结构、定时扫描（clocked scan）结构等。如上所述，最常用的结构为多路选择触发器（multiplexed flip-flop）。因为这个原因，本节重点是 DFT 扫描插入的多路选择触发器类型结构。

扫描也可用于测试 DUT 的任何可能发生的时序违例。为了理解这一点，我们需要深入了解扫描技术的操作。基本上扫描使用两个周期：捕获和移位。扫描数据由原始输入注入到被测物中，并被触发器（穿过逻辑）捕获，移出到原始输出，并同期望的结果相对比。在捕获和移位周期间选择的信号通常称为 scan_enable（扫描使能）信号。此外，还使用另一通常称为 scan_mode（扫描模式）的信号。扫描模式信号用于把 DUT 置于测试条件下。通常会修改设计使得在测试条件下被测物的行为不同于正常的功能行为。为了取得更高的可控性和/或可观性，需要这个修改的。有时简单地遵守严格的 DFT 规则即可。

下面总结了基本的扫描操作：

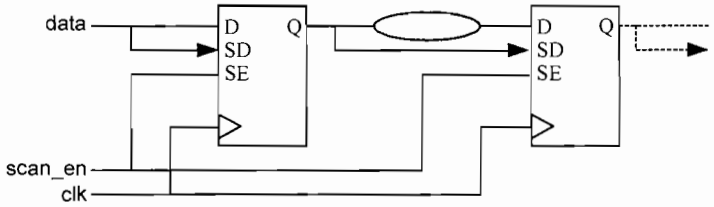
1. 加载/卸载扫描链（移位周期）；
2. 强制原始输入（除时钟外）；
3. 测量原始输出；
4. 激励时钟以捕获功能数据（捕获周期）。

为了理解扫描如何用于测试被测物的时序，对移位周期和捕获周期基本的了解是必要的。

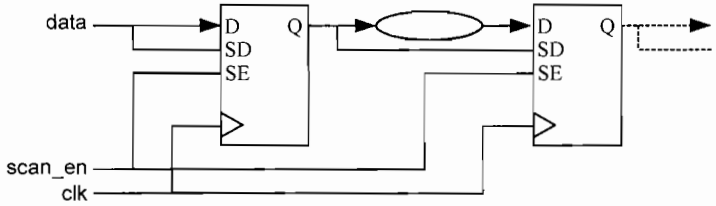
8.2.1　移位周期和捕获周期

在移位周期中，数据通过一个大的寄存器的雏菊链穿过整个设计，这些寄存器连起来像移位寄存器一样（因而得名移位周期）。移

位周期同捕获周期间的主要区别是，捕获周期使用触发器的功能“D”输入，而移位周期使用触发器的“SD”输入（如图 8-1 所示）。翻转 scan_enable 输入可选择所用的输入。因此“捕获”周期捕获到的数据简单地移出给测试仪进行比较。换言之，需要一个时钟周期执行捕获操作，而执行移位操作需要几个时钟周期（由扫描链的长度而定）。



(a) 移位周期 (Shift Cycle)



(b) 捕获周期 (Capture Cycle)

图 8-1　移位和捕获周期

图 8-1 描述了移位周期和捕获周期的行为。数据由原始输入注入被测物并通过触发器的“SD”输入端口移出被测物。假设对于移位操作 scan_en 端口是高有效的。一旦数据从扫描链移出并比较后，则反转 scan_en 信号（变为低）。在再次反转 scan_en（变为高）且移出并比较数据之前，应用一个时钟脉冲捕获通过“D”输入的数据到触发器。

这里值得注意的一个有趣的事情是，数据在捕获周期通过功能路径。换言之，它通过逻辑就像被测物在正常条件下工作一样。因此，如果用于捕获周期的时钟脉冲与功能时钟是同样的频率，则在扫描测试期间也能检查触发器间的所有时序关系。这基本上意味着，如果扫描覆盖率高并且扫描时钟和功能时钟同频率，功能测试就不需要了。

上述情况对大多数设计结构都有效，然而在实际设计中，通常会有功能路径不同于扫描路径的情况，这样的情况如图 8-2 所示。

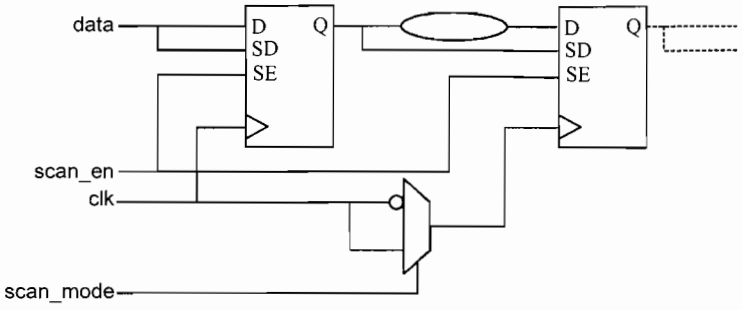


图 8-2　功能周期与捕获周期不同的情况

在图 8-2 中，在功能模式工作时，将一个反相时钟输入给第二个寄存器。然而，为平衡扫描链或使扫描插入简单，大多设计人员都喜欢使用的方法是引进一个称为 scan_mode 的测试信号。在测试期间，scan_mode 选择非反相时钟路径，而在功能模式 scan_mode 下选择反相时钟。在这种情况下，捕获周期不同于功能周期。要测试从第一个触发器的“Q”输出起始到第二个触发器“D”输入终止的路径的时序，不能使用扫描捕获周期。设计人员必须手动写出测试基准以测试这一特殊路径的时序。

8.2.2　RTL 检查

这是 Synopsys 最近引进的一个新特性。它是 DFTC 的一部分且用于检查 RTL 的任何可能的 DFT 规则违例。这是最强大和最有用的特性之一，使得设计人员能够在设计周期的早期检查 RTL 设计的 DFT 规则，它由命令 rtldrc 调用。下面脚本总结了这一特性的使用，与 RTL 检查相关的命令以黑体强调显示。

```
dc_shell -t > analyze –f verilog mydesign.v
dc_shell -t > elaborate mydesign
dc_shell -t > set_scan_configuration –style multiplexed_flip_flop
dc_shell -t > set hdlin_enable_rtldrc_info true
dc_shell -t > create_test_clock –period 100 –waveform {45 55} clk
dc_shell -t > set_test_hold 1 scan_mode
dc_shell -t > set_signal_type test_asynch_inverted reset
dc_shell -t > rtldrc > reportfile
```

设变量 hdlin_enable_rtldrc_info 为“true”，告知 DC 生成一个指

向 RTL 源文件的实际行号（DRC 违例的可能起因）的报告。没有这一变量，报告就不包含任何行号，默认值为“false”。

重要的是，注意上述脚本可作为用于综合的最终脚本的一部分，或由设计人员单独运行以检查设计的 DRC 规则的有效性。

8.2.3 使设计可扫描

Synopsys 通过它的测试准备（或 one pass）编译特性为设计人员提供进行自动扫描插入的能力。这一技术允许设计人员综合设计并将逻辑直接映射到扫描触发器，从而减少插入后调整的潜在要求。

不使用 Synopsys 工具进行扫描插入的公司，依赖其他方法可完成同样的任务。对这种情况，用等价的扫描触发器替换综合后网表中的正常触发器，然后将它们连成扫描链，就完成了扫描插入。强烈建议对扫描插入后的网表再次进行静态时序分析。因为在扫描触发器和它们对应的非扫描（正常）触发器的特征时序间存在着一些差异，若不修正这些差异，则会对设计的总松弛产生不利的影响。为避免这一问题，库开发商通常为正常触发器指定扫描触发器时序。

要运用 Synopsys 测试准备编译特性，应在编译前选择扫描类型。在某一设计上，set_scan_configuration 命令用于指示 DC 如何实现扫描。这一命令有多种可用的选项，可用于控制扫描实现，其中包括时钟混合、扫描链数目和扫描类型的选项。为了对其解释，下面只列出了带任意参数的一些最常用的选项。建议用户参考 Design Compiler 参考手册中的语法和可用的各种选项。

```
dc_shell -t > set_scan_configuration –style multiplexed_flip_flop  \
                                    –methodology full_scan          \
                                    –clock_mixing no_mix            \
                                    –chain_count  2
dc_shell -t > create_test_clock –period 100 –waveform {45 55} clk

dc_shell -t > set_test_hold 1 scan_mode
dc_shell -t > set_scan_signal test_scan_enable   \
                        –port scan_en –hookup pad/scan_en_pad/Z
dc_shell -t > set_scan_signal test_scan_in –port [list PI1 PI2]
dc_shell -t > set_scan_signal test_scan_out –port [list PO1 PO2]
dc_shell -t > compile –scan
```

```
dc_shell -t > preview_scan
```

create_test_clock 用于指定在扫描操作时使用的测试时钟。在上例中测试时钟名为“clk”，周期为 100ns，上升和下降边沿分别在 45ns 和 55ns 处。

set_test_hold 命令用于在测试模式时指定端口的常量，在上面的情况下，扫描时 scan_mode 端口保持逻辑高。

set_scan_signal 命令指出扫描链的扫描输入 / 输出端口和扫描使能信号。此处该命令将名为 scan_en 的端口指定为扫描使能端口并指示 DC 将设计中所有触发器的 SE 端口连到名为 scan_en_pad 的压焊块的 Z 输出。

compile-scan 命令直接将设计编译到扫描触发器而不将它们连成一个扫描链，也就是说不进行扫描插入。设计直接映射到扫描触发器而不是通常的触发器。在这一步设计是功能性正确的，但还不能扫描。

preview-scan 命令用于预览由 set_scan_configuration 命令选择的扫描结构。

强烈建议在编译后使用 check_test 命令检查设计的可测试性相关的规则违例。DC 通过给出警告/错误以标记任何违例。无法修正这些违例总是导致测试覆盖的减小。违例会发生是由于在扫描插入时遇到的各种 DFT 相关的问题。一些问题及其解决方法将在下一节进行讨论。

```
dc_shell -t > check_test
```

修正违例是设计者的责任。这主要通过在“问题”区域周围添加额外逻辑来提供对测试逻辑的控制来实现。为了修正这些问题，建议修改 RTL 源代码而不是网表。这种方法允许 RTL 源代码仍然作为“金”数据库，用做以后阶段的参考。另一方面，如果修改网表，那么在设计被送出后可能遗忘改动从而丢失它。

虽然扫描还未插入，这里额外的一步是通过生成统计 ATPG 测试图形得到设计故障覆盖率的一个估计。这一步有助于在早期量化设计的品质。如果覆盖率较低，那么唯一的选择是找出并修正需进一步提高的区域。然而，如果故障覆盖率较高，那么就可以进行下一步。

必须注意的是故障覆盖率只应视作最佳情况，因为设计可能是更

大层次的一部分，也就是说它可能是子模块。在子模块层次，当这个子模块载入整个设计（顶层）时，输入端口可控性和输出端口可观性可能会不同。对于整个设计，这可能会得到比预期低的故障覆盖率。如下命令用于产生统计测试图形：

```
dc_shell -t > create_test_patterns –sample <n>
```

一旦在 RTL 中找出并修正问题区域，设计就可以进行扫描插入。运用如下命令进行扫描插入：

```
dc_shell -t > insert_scan
```

insert_scan 命令不止将扫描触发器连起来以组成扫描链，它还禁止三态，建立和排列扫描链并且优化它们以除去所有的 DRC。这一命令插入额外测试逻辑以更好地控制设计的某些部分。

在扫描插入后，应再次通过 check_test 命令检查设计的任何规则违例。report_test 命令也可用于生成设计所有与测试相关的信息。有多种选项可控制报告的输出。在 Design Compiler 参考手册中能找到更多详细内容。

8.2.4 现有扫描

具有扫描链的设计需要不同对待。当输入使用 Synopsys DFTC 以外的其他工具扫描插入设计时，就会出现这种情况。在这种情况下不存在“db”文件。DFTC 的输入为扫描插入的结构化网表，因而“db”文件部分的所有扫描属性也不存在。换言之，DFTC 对扫描端口、复位等根本不了解。

扫描属性能再次应用于结构化网表以进行进一步处理（如通过 PhyC 的扫描链排序），这可由以下脚本完成。与原来一通综合方式不同的地方，使用黑体表示。

```
dc_shell -t > set_scan_configuration –style multiplexed_flip_flop \
                                    –methodology full_scan      \
                                    –existing_scan true

dc_shell -t > create_test_clock –period 100 –waveform {45 55} clk
dc_shell -t > set_test_hold 1 scan_mode
```

```
dc_shell -t > set_signal_type test_scan_enable scan_en
dc_shell -t > set_signal_type test_mode scan_mode

#For active low reset, use test_asynch_invert. Active high use test_asynch
dc_shell -t > set_signal_type test_asynch reset

dc_shell -t > set_signal_type test_scan_in [list PI1 PI2]
dc_shell -t > set_signal_type test_scan_out [list PO1 PO2]
```

8.2.5　扫描链排序

扫描链排序有诸多优势，然而也难免会有缺点。

扫描链重新排序的一些优点如下：

1. 减少拥塞，因而改善时序；
2. 更少的总面积（连线长度极大地缩短了）；
3. 由于减小了触发器的负载，因而改善了功能路径的建立时间；
4. 减少负的保持时间（主要的仿真与静态时序分析问题）；
5. 由于更少的总电容，因此改善了时序；
6. 由于驱动更少的连线电容，因此改善了功率消耗；
7. 更好的时钟树（更短的延迟和更少的缓冲器），因而改善了时序和低功耗。

缺点如下：

1. 增加扫描路径保持时间违例的几率；
2. 设计周期中额外的运行时间。

可用 DFTC、PhyC 或使用的布图工具来完成扫描排序。第 10 章将详述基于扫描单元物理邻近的扫描链排序的 PhyC 方法，DFTC 方法同 PhyC 极其相似。不用 physopt 和 insert_scan 命令可使用如下选项在基于扫描单元的物理布局位置基础上更加智能地连接扫描链：

```
dc_shell -t > insert_scan –physical
```

以上方法假定在运行 insert_scan 之前物理信息已被反标注（用 PDEF 格式）到设计中。反标注 PDEF 格式物理信息的命令如下：

```
dc_shell -t > read_pdef<PDEF file>
```

必须记住的是，扫描链排序增加了保持时间违例的几率，这是由于触发器彼此接近，从而为移位周期扫描路径生成一个非常短的路径。换言之，Q 到 SD（如图 8-1 所示）的连线长度非常短，因而扫描数据能比时钟更快到达，从而引起保持时间违例。

有趣的是 DFTC 以一种优美的方式对扫描链排序。它分析源寄存器提供的逻辑并试图找到这样一个单元，它在不改变移位周期的功能条件下，将这一单元的输出连接到目的寄存器的扫描输入端口。

考虑示意图 8-3，图中源寄存器的 Q 输出在遇到其余逻辑之前就提供给了缓冲器。在这种情况下，insert_scan 将缓冲器输出连到目的触发器的 SD 输入。这样，扫描路径上就多了一个缓冲器延迟。此延迟使目的触发器的任何保持时间违例的几率最小化。

必须注意的是它不必一定是缓冲器（如图 8-3 所示）。实际上，只要保持移位周期的功能，它可以是任何单元。例如 insert_scan 可从源单元的 QN 抽头输出并将反相器输出连接到目的触发器的 SD 输入上。换言之，源触发器的 QN 连接到反相器的输入引脚，而反相器的输出连接到目的触发器的 SD 输入。

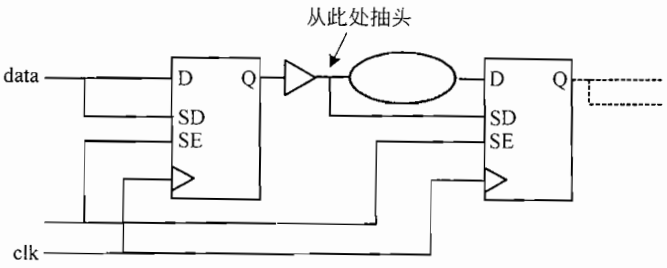


图 8-3 智能连接扫描链排序

如前所述，扫描链排序会增加保持时间违例的几率。使用这种方法可使保持时间违例的可能性最小化。insert_scan 的行为由以下变量控制：

```
dc_shell -t > set test_disable_find_best_scan_out false
```

默认值为“false”意味着 insert_scan 将分析逻辑并找到连接扫描链的最好方法。如果将参量改为“true”，insert-scan 将从 Q 抽头输出并将它连到目的触发器的引脚 SD 上。它不对 Q 提供的逻辑进行分析。

8.2.6　测试图案生成

随着设计扫描插入的完成，使用 TetraMAX 对整个设计生成测试图案。这是一个独立的 ATPG 工具，只用于生成测试图案并为 DC 提供一个无缝接口。它也为分析和调试提供了增强的 GUI 界面。

对于 ATPG 的讨论和 TetraMAX 的用法超出了本书的范围。要了解关于这方面的详细内容，建议读者参考 TetraMAX ATPG 用户指南。

在动态仿真期间，测试图案用作设计的输入激励以使用所有扫描路径。这一步应在全芯片层次上进行，并且最好在布图后。

8.2.7　综合实例

扫描插入是一个复杂的论题且通常有许多方法可使设计进行扫描。为减少混淆，本节提供一个示例脚本以巩固上述的所有信息。

one-pass 扫描综合、插入与排序的脚本

```
dc_shell -t > analyze –f verilog mydesign.v
dc_shell -t > elaborate mydesign
dc_shell -t > set_scan_configuration –style multiplexed_flip_flop
                                    –methodology full_scan      \
                                    –clock_mixing no_mix        \
                                    –chain_count  2
dc_shell -t > set hdlin_enable_rtldrc_info  true
dc_shell -t > create_test_clock –period 100 –waveform {45 55} clk
dc_shell -t > set_test_hold 1 scan_mode
dc_shell -t > set_signal_type test_asynch reset
dc_shell -t > rtldrc
dc_shell -t > source constraints.scr #clocks, iput/output delays etc.
dc_shell -t > set_scan_signal test_scan_enable  \
                    –port scan_en –hookup pad/scan_en_pad/Z
dc_shell -t > set_scan_signal test_scan_in –port [list PI1 PI2]
dc_shell -t > set_scan_signal test_scan_out –port [list PO1 PO2]
dc_shell -t > compile –scan
dc_shell -t > preview_scan
dc_shell -t > check_test
```

```
dc_shell -t > read_pdef mydesign_floorplan.pdef

dc_shell -t > insert_scan –physical
dc_shell -t > check_test
dc_shell -t > write –format verilog –hierarchy –output mydesign.sv
dc_shell -t > write_pdef –v3.0 –output mydesign_scan.pdef
```

排序现有网表的扫描链的脚本

```
dc_shell -t > read_verilog mydesign.sv #Gate level netlist
dc_shell -t > current_design mydesign
dc_shell -t > set_scan_configuration –style multiplexed_flip_flop
                                    –methodology full_scan        \
                                    –clock_mixing no_mix          \
                                    – exiting_scan true           \
                                    –chain_count  2
dc_shell -t > source constraints.scr #clocks, iput/output delays etc.
dc_shell -t > set_signal_type test_scan_enable scan_en
dc_shell -t > set_signal_type test_mode          scan_mode
dc_shell -t > set_signal_type test_asynch        reset
dc_shell -t > set_signal_type test_scan_in       [list PI1 PI2]
dc_shell -t > set_signal_type test_scan_out      [list PO1 PO2]
dc_shell -t > check_test
dc_shell -t > read_pdef mydesign_floorplan.pdef
dc_shell -t > insert_scan –physical
dc_shell -t > check_test
dc_shell -t > write –format verilog –hierarchy –output mydesign.sv
dc_shell -t > write_pdef –v3.0 –output mydesign_scan.pdef
```

注意：上面提供的脚本只使用了 DFTC。PhyC 也提供这种能力且比只使用 DFTC 更好。PhyC 流程将在第 10 章中介绍。

8.3 DFT 指导方针

对于设计，要获得高故障覆盖率依赖于实现的 DFT 逻辑的品质。不是所有的设计都是理想的，大多“现实世界的”设计要受到各种 DFT 相关问题的影响，如果不解决，会降低故障覆盖率。本节指出其中的一些问题并提供克服它们的解决办法。

8.3.1　三态总线竞争

这是 DFT 工具所面对的共同的问题之一。在扫描移位时，总线上的多个驱动器可能同时驱动总线，因而引起竞争。解决这一问题要求在指定时间只有一个有效驱动器。这可通过在设计中增加译码逻辑来完成，它通过 mux 来控制每个三态驱动器的使能输入。mux 用于选择正常信号（在功能模式下）和来自译码器的控制线，且只在扫描模式下选择译码器控制。

译码器输入通常直接由原始输入控制，因而能有选择地打开三态驱动器，从而避免了竞争。

8.3.2　锁存器

应尽量避免使用锁存器。虽然锁存器比触发器少占面积，它们却难以测试。虽然测试困难，但也不是完全不可能的。在扫描模式下使它们透明就可测试它们。这通常意味着要为每个锁存器增加（时钟）控制逻辑。如果一个独立时钟驱动所有锁存器，那么一个测试逻辑模块可用于控制时钟使得在扫描模式下锁存器透明。

8.3.3　门控复位或预置

DFT 要求触发器的复位/预置可控。如果触发器的复位/预置在设计中是功能门控的，那么触发器是不可扫描的。为避免这种情况，复位/预置信号应在扫描模式下旁路门控逻辑。通常使用 mux 来解决这一问题，以外部扫描模式信号作为其选择线，旁路复位/预置信号和原先的门控信号作为其输入。

8.3.4　门控时钟或生成时钟

门控时钟也遭受上面描述的门控复位同样的问题。DFT 要求触发器的时钟输入可控，其解决方法也是通过 mux 旁路门控逻辑，从而使得触发器可控。

这一问题普遍存在于包含有生成分频时钟逻辑的设计中。在扫描

模式下，应旁路用于生成分频时钟的触发器。分频逻辑在这种情况下变得不可扫描，但可由外部控制分频时钟，因而给设计的其余部分提供了覆盖。整个设计所获得的覆盖率增益弥补了分颁逻辑覆盖率的少量损失。

在图 8-4 中，可使用扫描模式下旁路 CLK 信号的 mux 从外部控制 secondary clock。这为设计的其余部分提供了 secondary clock 的可控性。依据使用的分频逻辑的类型，逻辑的某些部分可能不可扫描。如下命令用于告知 DFTC 当扫描插入时排除一列时序单元：

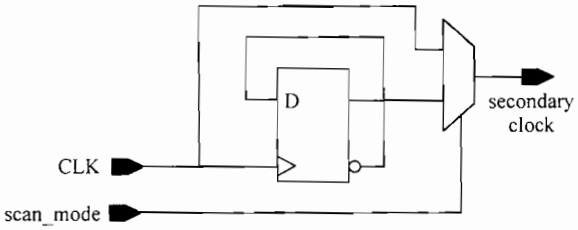


图 8-4 旁路生成时钟

```
dc_shell -t > set_scan_element false <list of cells or designs>
```

8.3.5 使用单时钟沿

大多数设计都用单时钟沿作参考来编码。然而在设计中经常有这样的情况，就是时钟的上升边沿和下降边沿都要用到，由于它无法处理这样的情况，这就为 DFTC 产生了一个问题。可以这样避免问题，即当设计在扫描模式下时，整个设计使用相同的时钟沿。如下面 VHDL 的例子所示：

```
Process (clk, test_mode)
begin
    if (test_mode= '1') then
        muxed_clk_output <= clk;
else
        muxed_clk_output <=not(clk);
    end if;
end process;
```

上面的 VHDL 代码推断出一个二输出 mux。在扫描模式下使用时钟的正沿，而在正常模式下使用时钟的下降沿。

8.3.6　多时钟域

强烈建议设计者为每个时钟域分配单独的扫描链。扫描链中的时钟域混合通常会引起时序问题。这归因于不同时钟域间的时钟扭斜的差异。使用这种技术的一个缺点是它可能导致扫描链变长。

另一个解决办法就是将属于一共同时钟域的所有触发器组合在一起，并将它们串联起来以组成一条扫描链。这就要求时钟域间的时钟扭斜尽量小。时钟源也应从外部（原始输入）可达，使得在测试设备上测试被测物时能由外部控制时序。

还有能解决这一问题的其他方法。其中一个方法就是，在时钟源多路选择时钟，使得在扫描模式下只使用一个时钟。

8.3.7　排序扫描链以最小化时钟扭斜

扫描链中时钟扭斜的存在往往导致保持时间违例。一些设计人员认为既然测试是在比正常工作速度低的条件下进行的，扫描链就不会有任何时序问题。这是一种概念错误。*只有建立时间是频率相关的，而保持时间是频率无关的*。因此极其重要的是，最小化时钟扭斜以避免扫描链中的任何保持时间违例。

扫描链可重新排序，具有较大时钟延迟的触发器更靠近扫描链源，而具有较小时钟延迟的触发器可保持在更远。这有助于减小时钟扭斜，从而最小化保持时间违例的概率。

8.3.8　因存储单元而不可扫描的逻辑

正如前文所述，存储器本身可用存储器 BIST 电路来测试。然而，周围并没有扫描链（通常内建）的存储单元（如 RAM）会导致其输入和输出的组合逻辑的覆盖率的损失。

让我们考虑一下由组合逻辑所驱动的 RAM 的情况。存在于其输入的这个逻辑被 RAM 所屏蔽，因而是不可测的。如果存储单元的输入不是直接来自时序单元，那么存在于时序逻辑和存储单元间的任何组合逻辑都是不可测的。为避免这一情况，在扫描模式下可旁路 RAM。这是通过一个 mux 短接提供给 RAM 的所有输入到 RAM 的输

出得到的。在扫描模式下，mux 激活短接路径并使数据旁路 RAM。

在扫描模式下有代表性的另一个问题是存储单元的输出是未知的。典型地，这会导致“未知”被引入到周围的扫描链，从而使其失效。如上所述，这种情况可使用旁路方法来避免。RAM 生成的“未知”被其输出的 mux 所阻塞。这是因为选择 mux 来旁路 RAM，因而它会阻止“未知”的传播。

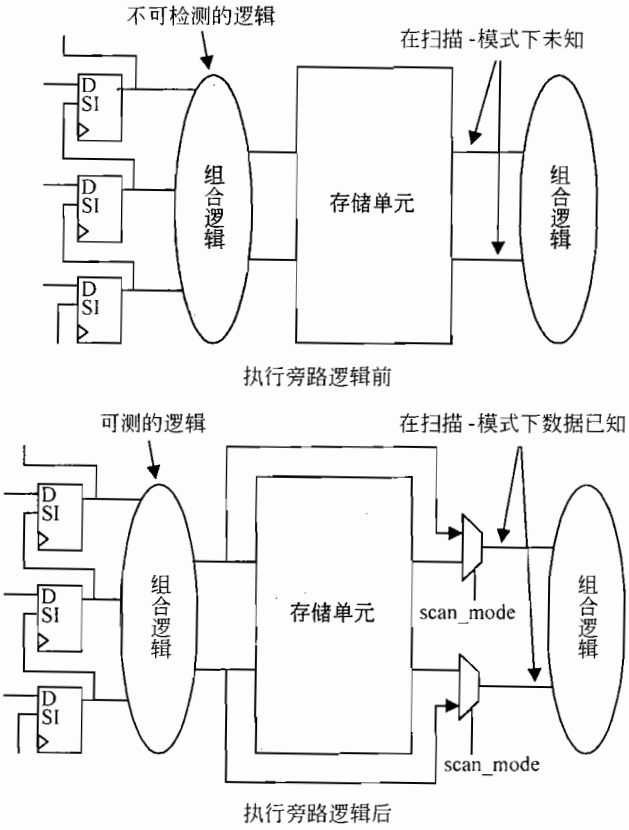


图 8-5 存储器旁路

8.4 小结

DFT 技术对于有效和成功测试加工后的器件是必须的。在设计周期初期实现 DFT 特性，可得到设计的完全测试覆盖，从而缩短在器

件加工后通常花在测试设备上的调试时间。

本章介绍目前所用的基本的可测性技术，包括 Synopsys 还不支持的逻辑及存储器 BIST 的概述。

本章详细介绍了使用 DFT Compiler 的扫描插入 DFT 技术，也提供了各种指导方针和解决方法，以帮助用户确认与这一技术相关的问题。

第 9 章

LINKS TO LAYOUT 和布图后优化

——包括时钟树插入

直到现在，如果合约是以结构化网表的形式提交给 ASIC 厂商进行加工的，那么一座虚拟的墙就会出现在前端和后端的工艺之间。ASIC 厂商负责设计的布图规划和布线，并给前端设计人员提供所得的延迟数据。然而这个过程是低效的，而且常常会导致设计人员和 ASIC 厂之间的网表和版图数据的多次交换。

随着我们更加深入到 VDSM 领域，前端和后端间的虚拟墙就注定要倒塌。这是因为 VDSM 工艺提出了巨大的挑战和困难。为了克服这些困难，ASIC 设计流程需要更大的可控性和灵活性。这要求在综合与布图过程间进行总集成。这意味着现在迫使设计人员自己进行布图。ASIC 厂商现在得到的不再是结构化网表，而是最终制造的物理数据。

合约中的这个转换导致了在综合工具和布局布线（Place & Route）工具（从这以后称作布图工具）之间明确定义的接口。Synopsys 称这个接口为 Links to Layout 或 LTL。

本章描述了 DC 与布图工具间的接口。几乎所有的设计都需要 LTL 接口实施布图后优化（PLO）。本章还提供了用于 PLO 的不同策略。此外，为了成功地完成布图，本章用一节论述布图工具进行的时钟树综合。

假设用户已经综合和优化了一个设计，且该设计满足所有的时序

和面积要求。现在问题产生了，“用于布图前优化的估计线载模型有多么接近从版图实际提取的数据？”寻求这些信息的唯一方式是进行布图规划并对这个设计布线。

随着几何尺寸的缩小，与连线的电容相比，其电阻增加。这导致互连线延迟占总延迟（单元延迟+互连延迟）的大部分。为了减小这个影响，设计人员被迫花费大量的时间对芯片进行布图规划。因此，DC 必须利用物理信息以进行进一步的优化。

使用 LTL，能在 DC 和布图工具间交换相关数据（如时序约束和/或布局信息）。这有助于 DC 改善布局后优化。这也减少了综合与布图间的迭代。

注意：随着 PhyC 的引进，存在两种设计流程。传统流程（在本章讲述）和基于 PhyC 的流程（将在第 10 章讲述）。然而传统流程的某些方面仍与基于 PhyC 的流程相关。因此，为了获得对整个流程的全面理解，强烈建议读者在继续第 10 章之前先阅读本章。

9.1 为布图生成网表

大多数布图工具只接受 Verilog 或 EDIF 网表格式作为输入。许多用 VHDL 编码设计的用户，通过 DC 为布图生成 EDIF 格式的网表。虽然这种格式是通用的，但仍存在缺陷。首先，EDIF 格式可读性差，因而在后面进行 ECO 时，修改网表会很麻烦。其次，EDIF 格式的网表不能仿真。

因此问题是：为什么设计人员必须对不能仿真的网表进行布线呢？如果 DC 由于其 EDIF 转换器中的某些错误生成不正确的网表（坏逻辑），会发生什么呢？使用 EDIF，问题将只能在进行 LVS 的较后阶段才能被发现。因此，建议设计人员使用由 DC 生成的 Verilog 格式的网表作为布图工具的输入。此外，Verilog 格式易于理解，并且倘若需要对设计进行 ECO，它也大大简化修改网表的任务。另外，即使设计的测试基准是 VHDL 格式，仍能利用混合仿真这些语言的仿真器（当前可以得到）来仿真 Verilog 网表。

在将网表（整个设计或单个模块）交给布图前，建议对网表进行如下的步骤，以利于设计从 DC 到布图工具平滑的转移。

a）唯一化网表。
b）通过更改设计中的连线名称简化网表。
c）从整个设计中移去未连接的端口。
d）确定叶单元的所有引脚名可见。
e）检查 *assign* 和 *tran* 语句。
f）检查无意的门控时钟或复位。
g）检查未解析的引用。

9.1.1　唯一化

如前面所提到的，为了在布图时进行时钟树综合，必须唯一化 DC 中的网表。此操作为设计中多次例化的子模块生成唯一的模块/实体定义。看起来这好像是不必要的操作，它减少了网表的可读性并导致网表增大，然而从物理上来看大多数布图工具认为设计是平的。换句话说，多次引用的模块，虽然各方面都理想，但物理上也存在单独的位置。此外，存在于这些模块内的触发器也需要连接到时钟源。显然，为把时钟树连接到这些模块需要单独的时钟连线名。

当时钟树从布图工具转到 DC 时，未唯一化的网表就存在问题。这问题只发生在如果时钟树信息单独转到 DC（通过后面讲述的方法），不包括从布图工具到 DC 的完整网表转移。在这种情况下，对于未唯一化的设计，虽然一个模块/实体定义可有多个例化的模块，但是只有一个模块/实体定义存在于网表中。当时钟树信息转回 DC，也就是修改 DC 中的设计库以包括子模块中的缓冲器和额外的端口时，这会引起问题。问题是两个不同的连线名（时钟树的输出）不能连接到单个模块/实体的同一端口。将设计唯一化解决了上面的问题，但它也会增大网表，因为它为模块的每个实例创建了单独的模块/实体定义。

一些用户更愿意当它们穿越层次达到顶层时唯一化网表，而其他用户从顶层一次性唯一化整个芯片。推荐的方法是：在唯一化网表前，从设计的所有子模块移去 dont_touch 属性。

下面命令可用于在顶层唯一化网表前移除整个设计的 dont_touch 属性：

```
dc_shell -t > remove_attribute [get_designs –hier {*}] dont_touch
```

```
dc_shell -t > uniquify
```

9.1.2 为布图修改网表

一些布图工具难以读取包含有不寻常连线名的 Verilog 网表。例如，有时 DC 产生以“*cell*”或“*–return”结尾或在中间的信号名。另外，用户可能发现一些连线名（或端口名）具有前导或拖尾下划线。大多数布图工具对连线或端口名的最长字符数也有限制。根据特定布图工具所加的限制，在写出网表前，用户能够在 DC 中清理它。DC 的这种能力给布图工具提供了一个平滑的接口，同时满足所有工具的需求。

为避免 DC 生成不合要求的信号名，用户必须首先定义规则，然后指示 DC 在写出网表前遵照这些规则。例如，在“.synopsys_dc.setup”文件中包括如下内容可定义名为“BORG”的规则：

```
define_name_rules BORG  –allowed {A-Za-z0-9_}                \
                        –first_restricted "0-9_\[]"          \
                        –max_length 30                       \
                        –map {{"*cell*","mycell"},{"*–return","myreturn"}}
```

在命令行（或通过脚本）运行如下命令指示 DC 遵照上面的规则（BORG）：

```
dc_shell -t > change_names –hierarchy –rules BORG
```

除了上面的，用户可能也希望改变网表中总线命名风格。DC 提供了一个变量，允许用户修改写出的网表中总线命名风格。此变量也可在设置文件中设置，如下所示：

```
set bus_naming_style {%s[%d]}
```

9.1.3 移除未连接的端口

许多设计都遭受模块端口有意或是由于遗留的原因未连接的问题。虽然这实际上对 DC 生成功能正确的网表没有任何影响，但是一些设计人员更愿在综合时移除这些端口。这通常是一个好习惯，这是

因为，如果未连接，DC 将给出关于未连接端口的警告信息。因为一个设计可能包含许多这样的未连接端口，真正的警告可能在众多的未连接端口警告中被忽略掉。因此，在生成网表之前，移除未连接端口并检查设计是可取的。这可由如下命令完成：

```
dc_shell -t > remove_unconnected_ports [get_cells –hier{*}]
dc_shell -t > check_design
```

9.1.4　可见的端口名

通常，所有综合的设计会导致映射的元件的一个（或多个）输出端口与连线不相连。当 DC 生成 Verilog 网表时，它不会写出这些未连接的端口名。与布图工具相关，物理单元的端口数和网表中同样单元的端口数间可能会出现不匹配。例如，一个 D 触发器包含四个端口，即 D、CLK、Q 和 QN，可连接如下：

```
DFF dff_reg (.D(data), .CLK(clock), .Q(data_out)) ;
```

在上述情形中，DC 没有写出 QN 端口，因为在设计中 QN 反相输出功能并未被利用。物理上，这单元包含所有四个端口，因此，当布图工具读取网表时，就会出现端口数目间的不匹配。在设置文件中设下述变量值为真可避免这种不匹配。

```
set verilogout_show_unconnected_pins true
```

只是根据布图工具的要求才使端口名可见，但是近来一些布图工具厂商已意识到这个限制，并改善了他们的工具，从而取消了上述限制。

9.1.5　Verilog 特殊语句

一些布图工具读取含 *tri* 连线、*tran* 原语及 *assign* 语句的网表有困难。这些是 Verilog 特殊的原语和语句，有许多原因会在网表中生成它们。

DC 为包含“inout”类型端口的设计生成 *tri* 连线。对包含这些端口类型的设计，DC 需要给双向端口赋值，从而生成了 *tri* 连线语句和 *tran* 原语。为避免这种情况，用户可以在设置文件中使用如下 IO 变

量。当设置为真时，所有的三态连线就被声明为 *wire*，而不是 *tri*。

```
set verilogout_no_tri true
```

有几个因素影响 *assign* 语句的生成，其中设计中的穿通就被认为是一个这样的因素。如果模块包含直接连到同一模块输出端口的输入端口，穿通就会出现。这导致 DC 在 Verilog 网表中生成 *assign* 语句。如果输出端口接地，或是由常量（如 1'b0 或 1'b1）所驱动，也会生成 *assign* 语句。当写出 Verilog 格式网表时，DC 发出一个警告，指示正在写出 *assign* 语句。

在穿通的情况下，用户通过在先前连接在一起的输入和输出端口间插入缓冲器，能防止 DC 生在这些语句。这样把输入端口从输出端口隔开，从而打破穿通。为此，在编译设计前使用如下变量：

```
dc_shell -t > set_fix_multiple_port_nets –feedthroughs
```

-buffer_constants 选项也可用于上面变量以缓冲驱动输出端口的常量。然而，由于有许多其他可生成 *assign* 语句的变化，因此使用如下命令进行全面覆盖可能更安全：

```
dc_shell -t > set_fix_multiple_port_nets –all –buffer_constants
```

许多设计人员抱怨说，甚至在上述所有步骤都执行后，在网表中仍会产生 *assign* 语句。在大多数情况下，这是由用户不知道的连线上存在的 dont_touch 属性引起的。用户通过执行 report_net 命令发现这些属性的存在。使用如下命令从连线移除 dont_touch 属性：

```
dc_shell -t > remove_attribute [get_nets <net name>] dont_touch
```

9.1.6 无意的时钟或复位门控

在将网表提交给布局布线之前，反复检查设计中的时钟总是一个好的想法。记住，时钟给所有信号提供参考，也就是说，所有信号直接与时钟相关，并且相对于时钟优化。如果无意地缓冲了时钟（可能你忘记给它应用 set_dont_touch_network 属性），将会影响时钟的延迟和扭斜，从而导致用户不能满足设定的时序目标。

通常，不认为复位同时钟一样重要。然而，由于 set_dont_touch_network 属性也被应用于复位，因此检查它们的缓

冲是明智的。

用户可用如下命令检查无意识的时钟门控：

```
dc_shell -t > report_transitive_fanout –clock_tree
```

可在上面命令中使用–from 选项检查无意识的其他信号（如复位信号）的无意识的门控。例如：

```
dc_shell -t > report_transitive_fanout –from reset
```

显然，时钟应在使用–clock_tree 选项之前被定义，或者也可将–from 选项用于时钟，它不要求时钟先被定义。注意不能同时使用–from 与–clock_tree 选项。

9.1.7　未解析的引用

设计人员应小心并时常检查任何未解析的引用。对于含有模块的实例而没有对应的定义的设计，DC 将给出一个警告。例如，模块 A 是例化子模块 B 的项层模块。如果当你为模块 A 写出网表时，却不能在 DC 中读到模块 B 的定义，DC 将给出一个警告，说模块 A 含有未解析的引用。在实例化单元和其定义间发生端口不匹配的情况下，也给出这个消息。

9.2　布图

有了正确和优化的网表，用户可以用布图工具将设计转化为它的物理形式。虽然布图是一个复杂的过程，但可归纳为如下三个基本步骤：

a）布图规划。

b）时钟树插入。

c）布线。

9.2.1　布图规划

这是整个布图过程中最为关键的一步。首先，对设计进行布图规划是为了在满足时序要求的同时得到最小的面积。其次，进行布图规

划将设计划分为易处理的模块。

广义上，布图规划包括单元和宏单元（如 RAM 和 ROM 或子模块）放置在恰当的位置上。目的是减小连线 RC 延迟和布线电容，从而得到更快的设计。将单元和宏单元放置到恰当的位置也有助于得到最小的面积和减小布线拥塞。

大多数设计都经历布图规划阶段，并且应花时间尽力去寻求正确的单元放置位置。最优的布局改善了设计的全部品质。它也有助于减少综合–布图间的迭代。布图规划对小型和/或慢速设计可能不像对包含大量门（>150K）的大型和/或时序关键设计那样重要。对这样的设计，建议对设计做层次化的布局和布线。例如，子模块已进行了布局和布线，并满足了所有时序和面积要求。这一子模块随后作为一个固定的宏模块引入整个设计，和其余的单元或宏模块一起进行布线。

9.2.1.1 时序驱动布局

找到单元和宏单元的正确位置是十分耗时的，因为每一通都需要全面的时序分析和验证。如果设计不满足时序要求，则重新进行布图规划。这显然是一个耗时并常常失败的方法，为克服这种困难，布局工具厂商引进了时序驱动布局的概念，通常称作时序驱动布图（TDL）。

TDL 方法包括向布图工具前向标注 DC 生成的设计时序信息。在使用这种方法时，单元的物理布局由时序约束指明。布图工具在布局单元时给时序以先后次序，并尽力不违反路径约束。

DC 使用如下命令生成 SDF 格式的时序约束：

```
write_constraints    –format <sdf | sdf –v2.1>
                     –cover_design
                     –from <from list>
                     –to <to list>
                     –through <through list>
                     –output <output file name>
```

上述命令生成 SDF 格式的约束文件，版本 1.0 和 2.1 都支持。如果布图工具不支持 2.1 版本，那么用户总可以通过指定“sdf”替代“sdf–v2.1”来使用默认的 1.0 版本。

除上例所述的选项外，write_constraints 命令还提供了更多的可选项，但使用-cover_design 可选项则更普遍。-cover_design 可选项

使DC输出刚好足够的时序约束以覆盖设计中通过每一驱动负载引脚对的最差路径。至于这一命令及其可选项的更多信息，建议用户参考DC 参考手册。

DC 的-cover_design 可选项生成的 SDF2.1 版本格式的时序约束文件如例 9.1 所示。SDF 文件包含 TIMINGCHECK 域，其中包括设计中所有路径的 PATHCONSTRAINT。PATHCONSTRAINT 的最后一个域 timingcheck 包含定义某一路径段路径延迟的三个数。这三个数，虽然在此例中相同，但对应着最小、典型和最大延迟值。这些数和它们对应的路径控制布图过程中的单元布局。

例 9.1

```
(DELAYFILE
(SDFVERSION "OVI 2.1")
(DESIGN "hello")
(DATE "Mon Jul 20 22:59:49 1998")
(VENDOR "Enterprise")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "1998.02-2")

 (DIVIDER/)
(VOLTAGE 2.70:2.70:2.70)
(PAROCESS "TYPICAL")
(TEMPERATURE 95.00: 95.00: 95.00)
(TIMESCALE 1ns)
(CELL
(CELLTYPE "hello")
(INSTANCE)
(TIMINGCHECK
(PATHCONSTRAINT INPUT1 U751/A3 U751/ZN U754/I1
        U754/ZN REG0/D (1.523: 1.523: 1.523))
(PATHCONSTRAINT INPUT2 U744/A1 U744/Z U745/A1
        U745/ZN REG1/D (1.594: 1.594: 1.594))
(PATHCONSTRAINT REG1/CLK REG1/Q U737/I U737/ZN
        OUTPUT1 (3.000: 3.000: 3.000))
(PATHCONSTRAINT REG2/CLK REG2/Q U1131/A2
        U1131/ZN REG3/D (25.523: 25.523: 25.523))
              ⋮
```

值得注意的是，根据设计的规模，整个设计时序约束的生成花费

大量的时间。选择时序关键路径（使用–from、–to 和–through 选项）生成约束以避免这个问题。用户也可进行层次化布局布线，采用 TDL 方法首先对小的子模块进行布线。层次化布局布线是较好的方法，因为它是基于“分而治之”的技术。将芯片分为小的可操作的模块使得设计人员处理运行时间问题相对简单。

进行 TDL 的另一方法是让布图工具基于设计的边界条件、顶层约束和时序例外生成时序约束。这是一个与工具相关的特性并得到一些布图工具厂商的支持，而另一些不支持。为了单元布局，布图工具使用自己的延迟计算器为设计中每条路径找出时序约束。这种方法更快速，就此意义来说，它优于以前所描述的其他方法。但此方法最大的缺陷在于，用户现在被迫使用并要相信布图工具的延迟计算器。无论如何，用此方法能相对容易地达到时序收敛。

进行 TDL 也会对整个面积产生影响。当使用上述方法时，可能会发现面积增大。然而，这一点存在争议：一些用户坚持说由于 TDL 方法导致了橡皮筋效应(rubber-band effect)，总面积会减小；而另一些用户则持相反的意见。

9.2.1.2 布图规划信息的反标注

为了实现时序和面积收敛，DC 和后端工具的完全集成允许 DC 能更有效地工作。DC 利用几种格式去读取布图信息。对于布图后优化，DC 有必要知道每个子模块的物理位置。使用物理设计交换格式（PDEF）允许 DC 访问相关的信息。PDEF 文件包含了簇（物理分组）信息和版图中的单元的位置。

布局前，网表优化使用的是遍布逻辑层次的线载模型。但物理层次可能不同于逻辑层次。物理上，单元/宏单元可能依据压焊块的位置或一些其他的考虑而组合在一起。因此，为了更有效地优化设计，DC 必须接受物理布局信息。这是以物理层次为基础、对线载在设计上的应用加以调整来实现的。

DC 用如下命令读取布图工具生成的 PDEF 格式的物理布局信息：

```
read_clusters –design <design name> <pdef filename>
```

一旦重新优化网表，物理信息就能通过 PDEF 文件传回布图工具。DC 使用如下命令完成这一功能：

write_clusters –design <design name>　–output　<pdef filename>

9.2.1.3　建议

a）一般来说，对所有类型的设计，TDL 都能很好地进行。然而，对时序关键和/或高速设计一定要使用 TDL 以最小化综合-布图迭化并取得时序收敛。

b）当处理大型设计时，仅对选定的连线生成时序约束。这将节省大量时间。但是如果布图工具能产生自己的时序约束，那么比其他方法更优先选择此法以节省时间。

c）对大型设计进行层次化的布局布线。虽然单调些，但它通常能提供最佳的结果和对整个流程的更好的控制。层次化布局布线也促进了有时在布线完成后需要的手动编辑网表。

d）在 DC 中进行布图后优化时，总是使用 PDEF 格式的物理布局信息，尤其是对大型的层次化设计。

9.2.2　时钟插入

前几章所解释的主要是控制时钟的延迟和扭斜。虽然一些设计实际上利用正扭斜以减少功耗，但大多数设计要求最小的时钟扭斜和时钟延迟。较大值的时钟扭斜会引起竞争条件，它增加了在触发器中锁存错误数据的几率。控制扭斜和延迟需要极大的努力和远见。

如前所述，布图工具进行时钟树综合（CTS）。CTS 在单元布局后立即进行，并且在这些单元的布线之前完成。布图工具由设计者的输入确定最佳的布局和时钟树的类型。一般地，要求设计者提供时钟树层数及每层所用的缓冲器的类型。显然，层数由时钟信号的扇出决定。

广义上，时钟树的层数与时钟树中所用的门的驱动强度成反比。换言之，如果使用低驱动强度的门，那么需要更多的层，而如果使用高驱动强度的门，则层数会减少。

为了使时钟扭斜和时钟延迟最小，设计者会发现如下建议有用。必须注意的是，这些建议并不是一成不变的规则。设计者经常采用混合的技术去解决时钟问题。

a）使用具有最小层数的平衡时钟树结构，尽量不要用过多的层数。层数越多，时钟延迟越大。

b）在大时钟树中，采用高驱动强度的缓冲器。这也有助于减少层数。

c）为减小不同时钟域间的时钟扭斜，尽力平衡层数和每个时钟树中使用的门的类型。例如，一个时钟驱动 50 个触发器，而另一个时钟驱动 500 个触发器，那么在第一个时钟的时钟树中使用低驱动强度的门，而另一个使用高驱动强度的门。这里的想法是，为了匹配两个时钟树的延迟，加速驱动 500 个触发器的时钟，并减慢驱动 50 个触发器的时钟。

d）如果你的库中包含平衡上升下降缓冲器，你可能更愿意使用这些。记住，一般来说，平衡上升下降缓冲器并不总是比通常的缓冲器更快（更小的单元延迟）。一些库提供具有比下降时间更短的信号上升时间的单元延迟的缓冲器。对采用上升沿触发的触发器的设计，这些缓冲器应是一个理想的选择。这个想法是研究库并选择最合适的可用的门。以往的经验也能派得上用场了。

e）为减少时钟延迟，可为两级尽量采用高驱动反相器。这是因为，逻辑上一个单一的缓冲单元由两个连在一起的反相器组成，因此具有两个反相器的单元延迟。使用两个单独的反相器（两级）将实现同样的功能，但会导致整个单元延迟的减少——因为没有对第二级使用另一个缓冲器（更多的两个反相）。仅对不含门控时钟的设计使用这个方法。原因稍后解释（要点 h）。

f）不要限制自己为 CTS 使用同样类型和驱动强度的门。现在的布图工具允许混合并匹配。

g）对一个平衡时钟树（例如 3 级），第一级通常是由压焊块驱动的单个缓冲器。为了减小时钟扭斜，第一级缓冲器被放在靠近芯片中央的位置上，以便它能通过相等的互连线连向下一级的缓冲器。这就生成了一个环状结构：第一个缓冲器在中央，第二个缓冲器（第二级）环绕着它，最后一级又环绕着第二级。这样，第一、二和三级间的距离保持最短。然而这样的安排虽好，却导致第一级缓冲器被放置在离源（压焊块）最远处。使用最小尺寸的连线从压焊块源到第一级缓冲器布时钟网络，这将导致极大的 RC 延迟，从而影响时钟延迟。因此，为了减小连线的电阻，必定要增大（加宽）从压焊块源到缓冲器（第一级）输入的连线尺寸，从而减小整个延迟。依据设计的尺寸和级数，也可能需要在其他级上进行这一操作。

h）为了最小化扭斜，布图工具应具有从时钟树的任一级抽取时

钟信号的能力。这对包含门控时钟的设计尤为重要。如果同样的时钟用于其他的非门控触发器，那么将会导致附加延迟，进而扭斜。如图 9-1（a）所示，如果时钟树在门处结束，附加延迟将导致门控时钟触发器与非门控时钟触发器间的巨大扭斜。因此，有必要为门控时钟触发器从高一级抽取时钟源，而为非门控时钟触发器维持完整的时钟树，如图 9-1（b）所示。然而，如果反相器用于时钟树（要点 e），则上述方法失效。在此情况下，不要将反相器用作时钟树的一部分。

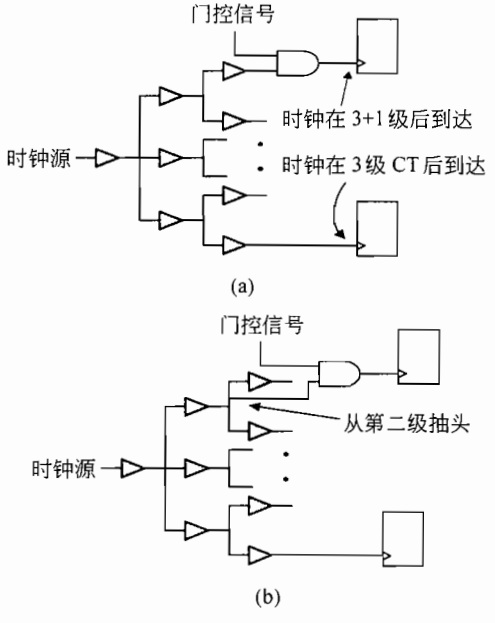


图 9-1　门控时钟的 CTS 与非门控时钟

9.2.3　时钟树到 Design Compiler 的转移

布图工具所做的时钟树综合修改了物理设计（在时钟网络中添加了单元），这个修改在 DC 原有的网表中并不存在。因此，用户有必要将这信息准确地转移到 DC 中，有如下几种实现的方式。

a）通常，所有的布图工具都有能力写出 EDIF 或 Verilog 格式的设计。由于布图工具可展平设计，设计人员可从布图工具得到展平的网表。当然，网表包含时钟树信息，但网表本身的巨大规模是令人畏缩和棘手的。此外，由于缺少原来的设计层次，展平的网表不易读。此法的另一问题是现在迫使用户认定这一网表为“金”网表，即所有

的验证（LVS 等）都必须对照这一网表进行。这样做类似于“自掘坟墓”，因为如果布图工具弄糟了版图，同样的问题就会反映在网表中。当然，LVS 将不出任何错误而通过，由于用户是对照版图生成的网表检查物理版图数据，也就是说，进行的是 LVL 而不是 LVS。另一方法是在原先的层次化网表和布图生成展平网表间进行形式验证。这当然是一个可行的方法，但考虑到设计的大小和复杂性，也有其自身的限制。大多数形式验证工具都受到这种限制，即它们在单一模块的验证上性能很好，但在全芯片验证上却无能为力。这在对照层次化网表验证展平的网表上更是如此。

b）第二种方法是，仅传递从时钟源起始到其终点（触发器的时钟引脚）的点到点时钟延迟信息。布图工具的延迟计算器将执行这一任务，在得到指令后，就给设计者提供 SDF 格式的时钟树的点到点时序信息。设计人员反标注此 SDF 文件到原先的设计以确定时钟延迟和扭斜。这一方法不要求时钟树从布图工具传送到 DC。但这一方法也有其自身的缺陷。首先，这一方法不允许使用来自版图的 SPF 数据反标注到 PT。此外，现在迫使设计者信任布图工具的延迟计算器，也就是说，引入了另一需要检验的变数。由于版图库独立于 Synopsys 库，为了得到同样的延迟值，Synopsys 库中的时序值必须和版图库中的时序值严格匹配。对照版图数据库验证原先的网表的困境仍存在，尤其是因为原先的网表不包含由于时钟树插入的额外的单元和连线。然而能够找到克服的办法并可成功地使用这一方法。

c）所有上述问题的一个解决方案是创造性地传送整个时钟树到 DC，而不改变设计的层次。一些布图工具甚至可生成含有 dc_shell-t 命令（如 disconnect_net、create_cell、create_port 和 connect_net）的 Synopsys 脚本。运行这些命令就在 DC 中原先的设计数据库中插入时钟树，并仍维持设计层次。当然，需要对照原先的网表通过形式验证验证所得的修改后的网表。由于设计层次未被改变，因此形式验证运行得很顺利。

d）另一个解决方案涉及强力修改，通常，布图工具在 CTS 一完成就生成一个对设计所做的所有改变的总结报告。可利用此报告并使用脚本语言（如 Perl 或 Awk）分析它以获得相关信息（如时钟树插入点的名称，缓冲器的类型和名称等）。一旦这些信息被收集起来，可直接修改原先的 Verilog 网表而不必通过 DC。修改后的网表应读回到 DC 中以检查任何语法错误。此外，修改后的网表也应对照原先的

网表进行形式验证。

近来，布图工具厂商认识到这一问题，通过从版图数据库生成层次化网表简化了这一过程。这一网表包含时钟树信息并且应对照原先的网表进行形式验证。一旦验证成功，网表可称为“完美的”。

9.2.4　布线

在时钟树插入之后，最后一步包括布线芯片。广义上，布线分为两个阶段：

1. 全局布线；
2. 详细布线。

第一布线阶段称为全局布线，其中全局布线器为每一连线分配一条通过版图的通用路径。在全局布线中，版图表面被分成几个区域。全局布线器决定在版图表面通过每一区域的最短布线，并不布几何连线。

第二布线阶段称为详细布线。详细布线器利用全局布线收集的信息，在版图表面的每一区域布几何连线。

必须注意的是，如果全局布线的运行时间很长（超过了布局的运行时间），则它表明布局的品质很差。在此情况下，应再次进行布局，重点应放在减少拥塞。

9.2.5　提取

直到现在，综合和优化都是利用线载模型进行的。线载模型是基于统计估算的最终布线电容。由于线载模型的统计特性。与布线后设计的真实延迟值相比，它们可能完全不准确。线载模型与真实延迟值间的差异导致了一个非优化的设计。

提取版图数据库以生成进一步优化设计所必需的延迟值。将这些值反标注到 PT 进行静态时序分析，然后再通过 DC 进一步优化和改善设计。

9.2.5.1　提取什么？

通常，几乎所有的布图工具都有能力使用各种算法提取版图数据库。这些算法定义了提取值的粒度和精度。依据所选的算法和所需的

精度，可提取如下类型的信息：

1. DSPF 或 SPEF 格式的详细寄生参数。
2. RSPF 或 SPEF 格式的约减寄生参数。
3. SDF 格式的连线和单元延迟。
4. SDF 格式的连线延迟+集总寄生电容。

DSPF（详细标准寄生格式）包含布线后网表的每段（多个 R 和 C）的 RC 信息。这是提取的最准确的形式。然而由于整个设计上的提取时间太长，这个方法不实用。这类提取通常限于设计的关键连线和时钟树。

RSPF（约减标准寄生格式）用 π 模型（2 个 C 和 1 个 R）来表示 RC 延迟。因为它不考虑与连线每一段相关的多个 R 和 C，所以此模型的精度不如 DSPF。提取时间可能也很长，从而限制了此类信息的使用。目标应用是设计的关键连线和小模块。

详细和约减寄生参数都能用 OVI（Open Verilog International）的标准寄生交换格式（SPEF）来表示。

最后两个（标号 3 和 4）是设计人员最普遍使用的提取类型。两者都利用 SDF 格式，但两者间有根本的不同。标号 3 使用 SDF 表示单元和连线延迟，而标号 4 使用 SDF 仅表示连线延迟单独生成的集总寄生电容。一些布图工具生成 Synopsys set_load 格式的集总寄生电容，从而可方便直接地反向标注到 DC 或 PT。

值得一提的是，PT 能读取所有 5 种格式（DSPF、RSPF、SPEF、SDF 和 set_load），而 DC 仅能读取 SDF 和 set_load 文件格式。SDF 和 set_load 文件格式不如 DSPF 或 RSPF 提取类型准确，但是，提取版图数据库的时间却明显减少了。对大多数设计而言，这种提取类型提供了足够的准确度和精度。但是，正如所建议的，只有设计中的关键连线和时钟应作为 DSPF 或 RSPF 提取类型的目标。

对生成完整的 SDF（标号 3 的方法）的布图工具，它使用自己的延迟计算器计算基于输出负载和输入信号转换时间的单元延迟。但使用这种方法有一个缺陷。综合使用的是 DC，它使用自己的延迟计算器优化设计。而当选择使用布图工具生成完整的 SDF 时，就引入了另一个需要检验的变数。如何知道布图工具所用的延迟计算器比 DC 使用的更准确呢？另外，即使将完整的 SDF 反标注到 PT，PT 的功能也未能得到充分利用。这是因为单元延迟已在 SDF 文件中固定，即使 SDF 文件中有条件延迟，在 PT 中进行个例分析也不会产生准确

的结果。这将在第 11 章详细讨论。

上述方法还存在另一问题，由于只反标注了单元和连线延迟，DC 不知道与设计中每一连线相关的寄生电容。因此，当进行布图后优化时，DC 仅能利用线载模型对设计做增量改变，从而使反标注的整个目的失效。但是，如果使用第四种方法（SDF 格式的连线延迟+集总寄生电容），DC 在布图后优化过程中（例如增大或缩小门的尺寸）将利用连线负载信息。

为避免这些问题，建议仅从版图数据库提取连线 RC 延迟（也称作互连线延迟）和集总寄生电容。反标注后，DC 或 PT 使用自己的延迟计算器依据反标注的互连 RC 和容性连线负载来计算单元延迟。

总结一下，为了向 DC 反标注以进行布图后优化，建议应由布图工具生成如下信息类型：

a）SDF 格式的连线 RC 延迟。

b）set_load 格式的容性连线负载值。

为使用 PT 进行静态时序分析，可以生成如下信息类型。

a）SDF 格式的连线 RC 延迟。

b）set_load 格式的容性连线负载值。

c）DSPF、RSPF 或 SPEF 文件格式的时钟及其他关键连线的寄生信息。

9.2.5.2　估计的寄生提取

在布线前阶段（全局布线之后）的寄生提取提供了一个更接近最终布线后设计的寄生值的近似值。如果估计值指出了时序问题，在开始进行详细布线之前，快速地重新对这个设计进行布图规划是相当容易的。这种方法减少了综合与布局间选代次数并避免浪费宝贵的时间。

全局布线之后估计的提取延迟值和详细布线之后真实的延迟值之间的差异相当小。相反，布图规划提取与详细布线提取之间的估计延迟值差异可能相当大。因此，当完成布图规划、单元布局和时钟树插入后，在提取估计的延迟值之前，进行全局布线设计是明智的。

完整的提取流程如图 9-2 所示。如果全局布线之后存在大的时序违例，有必要使用反标注的估计延迟在 DC 中重新优化设计。然而，如果时序违例不严重，则重新布图规划设计（和/或重新布局单元）可

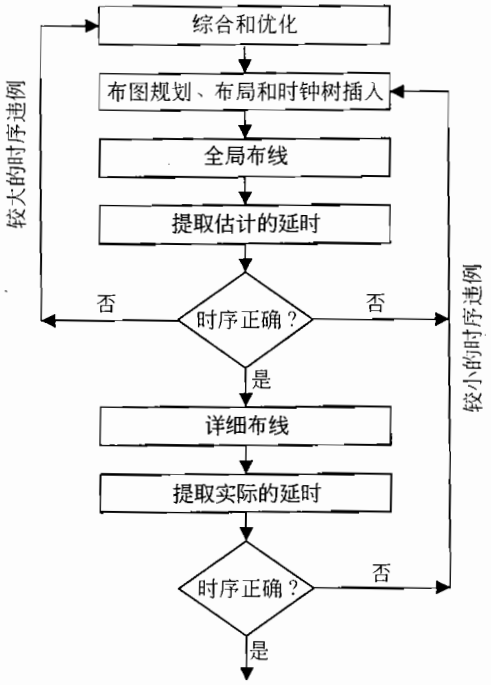


图 9-2 布线和提取流程

取得期望的结果。并且仅在消除了全局布线后产生的所有时序违例后才进行详细布线。

9.2.5.3 实际的寄生提取

在完成全局布线并取得满意结果之后，也就是说，设计没有 DRC 违例并取得所需要的芯片大小之后，进行全提取（实际值，没有估计）。

这显然是整个过程中最关键的部分。如果提取的值不准确，则最终的产品可能不工作。随着工艺缩小到 0.18μm 及以下，布图工具的提取算法需要考虑二阶和三阶的寄生效应。这些值的任何微小的偏差都可导致设计失败。

考虑提取值过分悲观的情况。静态时序分析表明信号满足保持时间要求。然而事实上，这些信号更快地到达导致了真正的保持时间违例，但是，由于悲观的反标注寄生电容，设计通过了静态时序。如果提取值过分乐观，则建立时间的情况是类似的。

9.3　布图后优化

进行布图后优化以进一步的优化和改善设计。这个过程包括将布图工具生成的数据反标注到 DC 中的设计。依据违例的严重性，优化可包括全面综合或通过使用在位优化（IPO）技术进行小的调整。正如 9.2 节所解释的，适于向 DC 反标注的版图相关数据如下：

a）SDF 格式的连线 RC 延迟。

b）包含容性连线负载的 set_load 文件。

c）PDEF 格式的物理布局信息。

9.3.1　反标注和自定义连线负载

下一步分析设计的静态时序。设计人员可选择使用 PT 或 DC 的内部静态时序分析引擎进行这一步。无论如何，布图后优化只能在 DC 中进行，因此版图数据需要被反标注到 DC 和 PT 中。

依据制造工艺，布图工具可生成两个单独的文件，分别对应最坏和最佳情况。如果有两个单独地分属于每个情况的 Synopsys 库，那么向使用最差情况 Synopsys 库的设计反标注最坏情况版图数据，同样地，最佳情况版图数据应被反标注到映射最佳情况 Synopsys 库的设计。

一些厂商只提供一个 Synopsys 库，它包括所有情况，也就是说，库表征的是 TYPICAL 情况，WORST 和 BEST 的情况值是由 TYPICAL 情况（减免）得来的。在这样的情况下，建议设计者将最坏情况的数据反标注到工作条件设为 WORST 的设计以进行最坏情况时序分析。将最佳情况时序数据反标注到工作条件设为 BEST 的设计以进行最佳情况时序分析。

在进行布图后优化之前，使用如下的 dc_shell–t 命令将版图生成的信息反标注到 DC 中的设计。

```
dc_shell -t > current_design    <design name>
dc_shell -t > source            <set_load file name>
dc_shell -t > read_sdf          <RC file name in SDF format>
```

```
dc_shell -t > read_clusters        <cluster file name in PDEF format>
```

在进行静态时序分析之前，使用如下的 pt_shell 命令将版图生成的信息反标注到 PT 中的设计。

```
pt_shell > current_design          <design name>
pt_shell > source                  <set_load file name in PT format>
pt_shell > read_sdf                <RC file name in SDF format>
pt_shell > read_parasitics         <DSPF, RSPF or SPEF file name>
```

在 PT 中反标注之后，如果设计由于相当多的违例未通过静态时序，用户就需要重新进行综合（或者甚至重新对某些模块编码）。因此，使用现有的版图信息进行重新综合是明智的。在重新综合过程中丢弃的版图数据只是浪费了布图所花费的时间和努力。此外，版图数据对微调设计是有帮助的。为获得最大的益处，应使用现有的版图信息通过 DC 生成自定义线载模型。使用自定义的线载模型所得的门级网表提供了更接近布图后时序结果的匹配。使用如下的 dc_shell–t 命令创建自定义线载模型：

```
create_wire_load    –design <design name>
                    –cluster <cluster name>
                    –trim <trim value>
                    –percentile <percentile value>
                    –output <output file name>
```

虽然上述命令有其他可用选项，但通常对大多数设计而言，上面列出的选项已足够。修剪值用于放弃低于某一值的数据，而百分数值用于计算平均值。通过改变百分数值，可在自定义线载模型中添加乐观或悲观。簇显然是在布图过程中用于把单元或模块组合在一起的分组名。

在创建自定义线载模型（CWLM）之后，应更新库以顾及新的 CWLM。这是因为，原来的工艺库仅包含不特定于具体设计的一般线载模型。为使用上面命令生成的 CWLM，库必须进行更新。如下命令可用来更新存在于 DC 内存中的库：

```
update_lib <library name> <CWLM file name>
```

必须注意的是，上述命令没有改变或覆盖源库，它仅更新 DC 内存以包括新的 CWLM。

9.3.2　在位优化

对布图后具有小的时序违例的设计，不需要进行全芯片综合。为了消除这些违例，在位优化或 IPO 是微调设计的一个很好的方法。IPO 的想法就是保持设计的结构不动，而只修改设计的失败的部分，从而对现有版图的影响最小。IPO 通常用于在某些位置添加/交换门以修正建立和/或保持时间问题。

IPO 是依赖于库的并且只限于进行下列：

a）调整单元大小。

b）插入或删除已有的单元（主要是缓冲器）。

通常，所有的 Synopsys 工艺库都具有定义一个使能或禁止 IPO 的属性。使能一个库中 IPO 的属性及其值如下：

```
in_place_swap_mode : match_footprint
```

连同上述的库级属性，Synopsys 库中的所有单元都有 cell_footprint 信息。例如，两个有同样功能，但驱动强度不同的单元可以有同样的 cell_footprint 值。这意味着两个单元具有相同的物理覆盖区域，因此用一个替换另一个不会影响现有的版图，也就是说，相邻单元不用移动。然而，这个限制以及单元尺寸调整、面积优化和缓冲器插入是通过使用下列变量进行控制的：

```
set compile_ignore_footprint_during_inplace_opt     true | false
set compile_ok_to_buffer_during_inplace_opt         true | false
set compile_ignore_area_during_inplace_opt          true | false
set compile_disable_area_opt_during_inplace_opt     true | false
```

使用这些变量允许设计者控制设计中做改变的量。在进行 IPO 之前，在 dc_shell-t 命令行或在 Synopsys 设计文件中给上述变量设置适当值。

使用如下的命令调用 IPO：

```
dc_shell -t > compile –in_place
dc_shell -t > reoptimize_design –in_place
```

两个命令几乎在所有方面都是相似的，也就是说，两者都利用除物理信息外反标注的版图信息。当重新优化设计时，reoptimize_

design 利用物理位置信息。这两个命令间的另一区别是进行 IPO 时“compile –in_place”命令使用库线载模型，而“reoptimize_design – in_place”命令利用自定义线载模型。因此，在进行 IPO 时，必须使用后一个命令。

必须注意的是，当单独使用时，reoptimize_design 会对设计做大的改动。为消除这个可能性，总是使用–in_place 选项。

9.3.3 基于位置的优化

基于位置的优化（LBO）是 IPO 的组成部分，当对包含反标注的 PDEF 格式物理布局位置信息的设计进行 IPO 时，它会被自动调用。由于 LBO 的重要性和额外的能力，因此本章把它从 IPO 中独立出来。

进行带 LBO 的 IPO 改善了设计的整体优化，由于 DC 现在能使用单元布局信息。这允许 DC 在优化期间应用更强大的算法。

考虑起始于原始输入并终止于触发器的路径段。布图后时序分析显示了这个路径的保持时间问题。如图 9-3 所示，LBO 优化会在靠近触发器（终点）处添加缓冲器，而不是在源（起点）处添加它。在源处添加缓冲器可引起源自同一源的另一路径的建立时间失败。

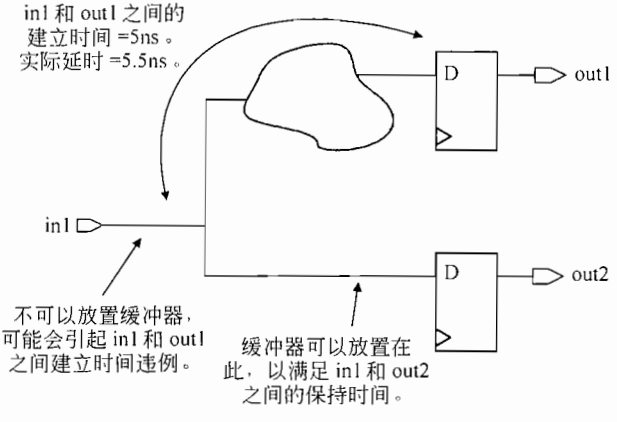


图 9-3 IPO 与 LBO 的对比

除了增强在最佳位置插入缓冲器的能力外，LBO 也提供了簇间连线和为连接插入或删除缓冲器而创建新连线的更好的建模。这是由于 DC 知道单元插入或删除的位置的事实。

为了“reoptimize_design –in_place”进行 LBO，除了与 IPO 相关的变量“compile_ok_to_buffer_during_inplace_opt”外，下列变量需要设为真：

```
set lbo_buffer_removal_enabled true
set lbo_buffer_insertion_enabled true
```

如果设置上述变量的值为假（默认值），则 LBO 不能进行缓冲器插入或移除。对这种情况忽略位置信息，只使用 IPO 算法去处理缓冲器插入或删除。

使用“compile –in_place”或“reoptimize_design –in_place”进行 IPO 和/或 LBO 所做的改动可通过使用如下 dc_shell–t 变量写出到文件：

```
set reoptimize_design_changed_list_file_name <file name>
```

如果文件已经存在，则新的改变集将添加到同一文件中。

9.3.4　修正保持时间违例

几乎每一个设计都会经历修正保持时间违例的过程，尤其是对较快的工艺。设计人员使用各种方法处理这个问题。由于这个原因，因此用单独的一节讨论使用这些方法产生的问题，并在一个主题下把它们统一起来。

大多数设计人员以紧约束进行综合设计以最大化建立时间，所得的结果是一快速逻辑，其数据相对于时钟较快地到达触发器的输入。这可导致保持时间违例，由于数据在被触发器锁存前改变了数值。通常，设计人员喜欢在设计的初次布局和布线之后修正保持时间违例，从而利用更准确的延迟值。

移除保持时间违例包括相对于时钟的延迟数据，以便数据在时钟沿到达之后指定的一段时间（保持时间）不改变。有几种设计人员用来插入适当延迟的方法，其概述如下：

a）使用 Synopsys 方法。

b）手工插入延迟。

c）用蛮力 dc_shell-t 命令自动插入延迟。

9.3.4.1 Synopsys 方法

Synopsys 提供如下 dc_shell-t 命令，它使能 compile 命令修正保持时间违例：

```
set_fix_hold <clock name>
dc_shell -t > set_fix_hold CLK
```

通过同时设置 min/max 库（从 DC98 版本开始），并对 set_input_delay 命令指定 min/max 值，可以在初次编译（或布图后）期间使用上述命令。在同一时间设置 min/max 库的想法是，它消除大多数设计都需要的两通综合（初次综合为最大化建立时间，再次优化为修正保持时间违例）。例 9.2 描述了使用单通综合方法修正布图后保持时间违例的方法。

例 9.2

```
set_min_library "<worst case library name>"          \
                -min_version "<best case library name>"

set_operating_conditions -min BEST -max WORST

source net_delay.set_load
read_timing interconnect.sdf
read_clusters floorplan.pdef

set_input_delay -max 20.0 -clock CLK [list IN1 IN2]
set_input_delay -min -1.0 -clock CLK [list IN1 IN2]

set_output_delay -max 10 -clock CLK [all_outputs]

set_fix_hold CLK

reoptimize_design -in_place
```

作为选择，也可使用最差情况库编译设计以最大化建立时间，接着在布图后重新优化，通过将设计映射到最佳情况库以修正保持时间违例。虽然这个方法使用了 two-pass 综合方法，但由于它稳定的特性仍推荐。大多数设计人员喜欢使用这个方法是因为已经花费了时间并付出了努力来定义这个方法并使其成熟。

必须注意的是，上述命令独立于 IPO 命令。IPO 命令通常以版图信息反标注到设计来修正初次布图后保持时间违例。通过使用“compile–incremental”命令增量编译设计，完成修正布图前保持时间违例。

set_fix_hold 命令指示 DC 在合适的位置插入缓冲器来修正保持时间违例。这也由前面描述的 IPO 相关变量所控制。禁止缓冲器插入，只能交换数据通道中的单元（也就是说，以驱动强度较低的门替代驱动强度较高的门）以增加单元延迟，从而延迟到达触发器输入的数据。

9.3.4.2　延迟的手动插入

如果时序分析显示非常少量的保持时间违例（少于 10～20 处），那么用 set_fix_hold 命令去修正这些违例是不值得的。这种情况下，延迟可以手动插入到网表中。设计者可串联一串缓冲器，以相对于时钟恰好足够地延迟数据，以使它通过保持时间检查。

但值得注意的一点是，缓冲器链可能不能提供充分的延迟，因为延迟依赖于在版图中缓冲器的布局。通常，缓冲器将被放置在彼此非常靠近的地方，因此总的延迟应等于通过每个单元的延迟代数和。由于放置的单元非常接近，互连延迟本身不重要。为克服这个，推荐将高扇入门（如 8 输入与门）的所有输入连在一起，并串联起来形成链。使用这个方法的好处是，高扇入门的输入引脚电容被利用起来了。高扇入门的输入引脚电容通常比单一输入缓冲器的大得多。因此，这个延迟数据的方法提供了一种与在版图中单元布局位置无关的解决方案。这个方法适用于如果工艺库没有延迟单元。如果这些单元存在，那么它们应主要用于修正保持时间违例。

9.3.4.3　蛮力方法

这是一独特的方法，但需要像 Perl 或 Awk 这样的脚本语言方面的知识。

举例来说，如果时序报告显示许多保持时间违例，并且通过 Synopsys 方法修正它们需要大量的运行时间，那么在此情形下，找到失败路径的松弛量（建立时间分析）和对应的违例（保持时间分析）的另一方法是使用脚本语言分析时序报告（最差情况和最佳情况）。用这些数字，用户可以对失败路径生成 dc_shell–t 命令，如

disconnect_net、create_cell 和 connect_net。在 dc_shell–t 中运行这些命令，将迫使 DC 在适当的位置（应在靠近失败路径的终点处）插入和连接缓冲器。这称作自动完成的蛮力方法。

这绝不是一个无瑕疵的方法，但它工作得相当好。同 Synopsys 方法相比较，用这种方法修正保持时间违例所花费的时间是可忽略的。

9.4 小结

Links to layout 是布图工具与 DC 间集成的一个重要组成部分。为了使 DC 更好地进行优化和微调设计，本章重点关注与布图工具交换数据的各个方面。

本章详细解释了由布图工具向 DC 传递时钟树信息的相关问题。对照原来的网表，交叉检查布图工具生成的网表仍为主要瓶颈。为了克服这个问题并选择正确的解决方案，给用户提供了多种选择方案。

从如何从 DC 生成没有错误的网表以最少化布图问题开始，本章包含布局和布图规划、时钟树插入、布线、提取和布图后优化技术，还有各种修正保持时间违例的方法。在每一步，都提供了建议以便于用户选择正确的方向。

第 10 章
物理综合

在设计复杂度增加的同时，上市时间快速地缩短。几何尺寸的缩小进一步恶化了这一问题，迫使 ASIC 设计人员在设计周期的初期考虑时序的同时也考虑功耗和串扰。布图工具和 DC（或 PT）之间的数据交换是低效率的，综合–布图迭代过程所浪费的时间仍是一个主要瓶颈。

综合-布图迭代的主要原因归结为它依靠线载模型综合设计的传统综合方法。线载模型只是对布线后设计的估计。它们可能与从版图表面提取的真实延迟有很大的不同。从布图到综合的迭代解决了这一问题，然而它是以时间为代价的。

为了缓解这一问题，Synopsys 引入了一种不需要线载模型的综合设计的新方法，这一新工具叫做 Physical Compiler(或 PhyC)，它在基于布图规划信息进行综合的同时完成布局。综合与布局联合在一起，这样在综合时就提供了实际互连延迟的准确模型。此外，这一工具也最小化了先前在布图工具和综合工具间传递数据的麻烦。

PhyC 是 DC 的超集，它包含所有的 DC 命令和其自身的一些命令，键入 psyn_shell 来启动它。

10.1 初始化设置

PhyC 使用和 DC 一样的设置文件*.synopsys_dc.setup*。唯一的区别就是，除了逻辑库以外它要求在同样的文件中包含物理库。在第 3 章提供了语法和用法的完整描述。

10.1.1 重要变量

与 DC 一样，PhyC 的行为受变量控制。这些变量可包含在设置文件当中。为得到 PhyC 变量的完整列表，键入如下命令：

```
Psyn_shell> printvar
```

下面描述一些常用的变量：

• physopt_pnet_complete_blockage_layer_names

```
psyn_shell > set physopt_pnet_complete_blockage_layer_names
"metal1 metal2"
```

上述变量定义了 PhyC 的阻挡层电源/地层。通常这是用于布局规划中的电源和地条，意思就是告诉 PhyC 不要在条的下面放置任何单元。以上情况中，使用 metal1 和 metal2 层名。

• physopt_pnet_partial_blockage_layer_names

```
psyn_shell > set physopt_pnet_partial_blockage_layer_names
"metal1 metal2"
```

这一变量允许 PhyC 有限度地在电源/地条下放置单元。只有在单元自己的层不与电源/地条使用的层冲突时，单元才会在条下移动。上述情况中，不包含 metal1 或 metal2 层的单元部分可以在条下面移动。

10.2 作业模式

可用如下的两种模式进行物理综合：

（1）RTL 到布局后的门（或 RTL2PG）

（2）门到布局后的门（或 G2PG）

PhyC 需要 IEEE PDEF3.0 格式的布图规划信息。这一格式同 SDF 格式非常类似。它包括单元的物理坐标、布局阻碍物（如预置 RAM/ROM）、功率条、芯片边界、压焊块/端口位置等等。换言之，这一文件包含 PhyC 进行优化布局所需的所有必要信息。

10.2.1　RTL 到布局后的门

在这一模式中，PhyC 的输入是 RTL 设计、IEEE PDEF 3.0 格式的布图规划信息、I/O 时序约束和物理库；输出为结构化网表和 PDEF 3.0 格式的布局数据，如图 10-1 所示。

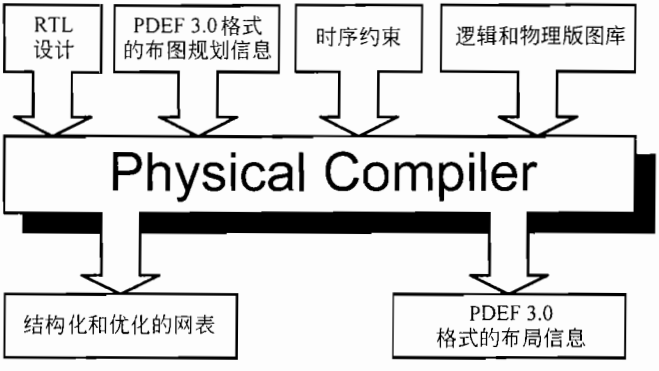


图 10-1　RTL 到布局后的门

同 DC 类似，PhyC 提供一个命令，编译 RTL 以生成优化的网表和布局信息。这个命令称为 compile_physical，具有同 DC 原先 compile 命令类似的选项。下面列出了最常用的选项：

```
compile_physical –congestion –scan
```

–congestion 选项用于调用进一步优化的算法以减少布线拥塞。–scan 选项同 DC 中 compile 命令的–scan 选项相同。这里，它也只是将设计映射到扫描触发器，但并不连接成扫描链。

以下脚本展示了 RTL2PG 物理综合的流程，PhyC 命令以黑体突出显示。

RTL2PG 脚本的实例

```
#Read the source RTL and floorplan information
    read_verilog mydesign.v
read_pdef floorplan.pdef

#Define operating conditions and timing constraints.
#Note the absence of wire-load models.
```

```
#All other steps same as before.
        current_design mydesign
        uniquify
link
set_operating_conditions WORST
set_load 1.0 [all_outputs]
create_clock ...
set_clock_latency ...
set_clock_transition ...
set_dont_touch_network ...
set_input_delay ...
set_output_delay ...

#Define attributes for scan
        set_scan_configuration ...
        create_test_clock ...
        set_test_hold 1 ...
        set_scan_signal test_scan_enable ...
        set_scan_signal test_scan_in ...
        set_scan_signal test_scan_out ...

#Synthesize the design using the physical information.
#Also synthesize to scan flops. No stitching done.
        compile_physical –scan
        check_test
        preview_scan

#Stitch scan chains based on physical location of flops.
        insert_scan –physical
check_test
#Write out structured netlist along with placement information
        write –f verilog –h –o mydesign_placed.sv
        write_pdef –v3.0 –o mydesign_placed.pdef
exit
```

10.2.2 门到布局后的门

在这种模式中，PhyC 的输入是结构化网表而不是 RTL。其余的输入和输出文件同作业模式 RTL2PG 相同。

在这种模式中，PhyC 的输入是一个已经用传统方法（使用 DC

compile 命令并采用线载模型）综合后的结构化网表。它只需要这些门的布局，如图 10-2 所示。

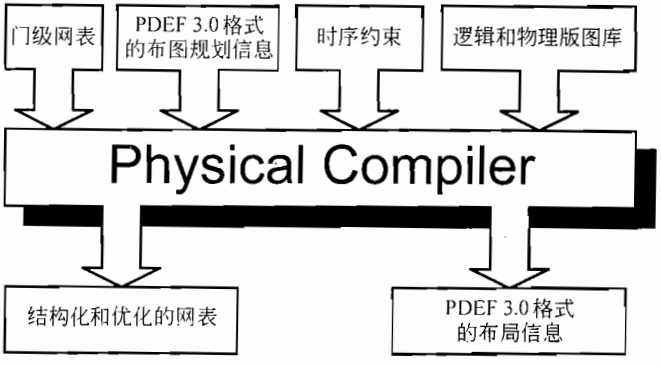


图 10-2　门到布局后的门

PhyC 提供如下命令进行 G2PG 操作：

```
physopt –congestion –timing_driven_congestion –scan_order
```

–congestion 选项与 RTL2PG 方法描述的相同。如果设计约束只有拥塞是最重要的（以面积为中心易于满足时序要求的设计），应使用这一选项。–timing_driven_congestion 应用于时序和拥塞都很重要的设计。这一选项在考虑拥塞的同时，进行时序驱动单元布局。

–scan_order 选项用于根据触发器的物理位置排序扫描链。这也大大有助于减少拥塞。

在 G2PG 模式中，有两种子作业模式分别叫做“双通方法（two pass）”和“集成方法”。这些方法同 physopt 处理链连接的方式相关。

10.2.2.1　two pass 方法

在双通作业模式中，在运行 physopt 之前，使用 insert_scan 命令将扫描链连接起来。physopt 命令随后不仅用于进行布局，而且根据每个触发器的物理位置排序扫描链。

如下脚本描述了双通 G2PG 流程，PhyC 和扫描链连接命令以黑体突出显示。

双通 G2PG 脚本的实例

```
#Read the synthesized gate level netlist in "db" format.
#Assuming "compile -scan" was used to produce the "db" file.
```

```
#In other words, no scan stitching done, only the design has
#been synthesized directly to scan flops.
      read_db mydesign.db
#Read the floorplan information
read_pdef floorplan.pdef

#Define operating conditions and timing constraints.
# Note the absence of wire-load models. The constraints are
# needed by PhyC to perform timing driven placement.
current_design mydesign
      uniquify
link
set_operating_conditions WORST
set_load 1.0 [all_outputs]
create_clock ...
set_clock_latency ...
set_clock_transition ...
set_dont_touch_network ...
set_input_delay ...
set_output_delay ...

#Define attributes for scan
      set_scan_configuration ...
      create_test_clock ...
      set_test_hold 1 ...
      set_scan_signal test_scan_enable ...
      set_scan_signal test_scan_in ...
      set_scan_signal test_scan_out ...
#Stitch scan chains based on physical location of flops.
      insert_scan
check_test
# Perform timing driven placement along with reduced
# congestion. Also order the scan chain based on plysical
# location of each flop.
physopt –timing_driven_congestion –scan_order
check test
#Write out structured netlist along with placement information
      write –f verilog –h –o mydesign_placed.sv
      write_pdef –v3.0 –o mydesign_placed.pdef
exit
```

注意：上述脚本假定 “compile–scan” 及其他扫描属性用于生成起始的 “mydesign.db” 文件。因此，所有扫描相关属性必定已是 “db” 文件的一部分，这些属性因此能够从上述脚本中省略掉。提供它们只是出于解释目的。

10.2.2.2　集成方法

在集成方法这种作业模式中，physopt 命令不仅用于进行布局，而且用于在扫描触发器物理邻近的基础上连接和排序扫描链，它基本上统一了连接和排序功能。换言之，在这种模式中剔除了 insert_scan 命令，从而简化了流程。

如下脚本描述了集成 G2PG 流程，PhyC 命令以黑体突出显示。

集成 G2PG 脚本的实例

```
# Read the synthesized gate level netlist in "db" format.
# Assuming "compile –scan" was used to produce the "db" file.
# In other words, no scan stitching done, only the design has
# been synthesized directly to scan flops.
read_db mydesign.db
#Read the floorplan information
read_pdef floorplan.pdef
# Define operationg conditions and timing constrainst.
# Note the absence of wire-load models. The constraints are
# needed by PhyC to perform timing driven placement.
    current_design mydesign
    uniquify
    link
    set_operating_conditions WORST
    set_load 1.0 [all_outputs]

    create_clock .....
    set_clock_latency .....
    set_clock_transition .....
    set_dont_touch_network .....
    set_input_delay .....
    set_output_delay .....

#Define attributes for scan
    set_scan_configuration .....
```

```
        create_test_clock .....
        set_test_hold 1 .....
        set_scan_signal test_scan_enable .....
        set_scan_signal test_scan_in .....
        set_scan_signal test_scan_out .....
# Perform timing driven placement along with reduced
# congestion. Also stitch and order the scan chain
# based on physical location of each flop.
physopt –timing_driven_congestion –scan_order
check test

#Write out structured netlist along with placement information
        write –f verilog –h –o mydesign_placed.sv
        write_pdef –v3.0 –o mydesign_placed.pdef
exit
```

注意：上述脚本假定“compile -scan”及其他扫描属性用于生成起始的“mydesign.db”文件。因此，所有扫描相关属性必定已是“db”文件的一部分，因此这些属性能够从上述脚本中省略掉。提供它们只是出于解释目的。

10.3 其他 PhyC 命令

与 DC 不同的是，PhyC 没有许多自己的命令。必须注意的是，这些命令的大部分都是在 physopt 或“compile_physical”的庇护下运行的。用户无须显式运行这些命令。它们的意图是，如果需要，可进一步调整版图表面。因此，这里给用户提供这些命令的唯一意图是“那里还有什么”，不提供任何进一步的描述。建议用户参阅 Physical Compiler 用户手册以得到这些命令的完整描述和用法。

下面是一些这样的命令：

```
–create_placement
–legalize_placement
–check_legality
–run_router
–set_congestion_options
–report_congestion
```

```
–set_dont_touch_placement
–remove_dont_touch_placement
```

10.4　Physical Compiler 问题

不幸的是，同大多数的 EDA 工具类似，PhyC 也遇到了一些问题。这些问题都与 PhyC2001-SP1 版本相关。期望以后的版本能够解决其中的一些问题。以下列出了一些比较关键的问题。

1. 当使用 read_verilog 读取门级网表时，PhyC 在最终的 verilog 网表中输出许多 assign 语句。这只发生在进行扫描链排序时，并且即使使用如下隐变量之后仍会发生：

```
set physopt_fix_multiple_port_nets true
```

当读取一个预编译的 db 文件并且执行相同操作（包括使用上面的变量）时，没有 verilog　assign 语句生成。如果将一个门级网表编译为 db 格式（read_verilog 后跟 write–f db），PhyC 仍生成 verilog assign 语句。防止这种情况的唯一办法就是将下面的变量和上面的变量一起使用：

```
set_fix_multiple_port_nets –all –buffer_constants
```

显然这是一件恼人的事，希望 Synopsys 能很快解决这个问题。

2. Synopsys 极力推广的“集成 physopt 流程”并不像广告宣传的那样工作。其想法是通过使用单通扫描综合编译设计，然后使用扫描排序选项运行批 physopt。假设 physopt 进行扫描连接、排序和布局。然而即使通过了 check_test，PhyC 也会中断并抱怨设计没有准备好扫描。在 physopt 运行前使用下面的属性可解决这一问题：

```
set_attribute <design name>　is_test_ready true –type boolean
```

由于某种原因，有必要明确告知 PhyC 设计已准备好扫描。这一问题在 Solv-Net 数据库中有很好的记录并将在最近得到解决。

3. 有时对利用率较低的设计，PhyC 对布局后的单元产生聚集效应。换言之，如果设计是压焊块受限的并且逻辑面积同整个芯片面积相比是非常小的，单元布局不会是最优的。它们在一些簇中聚在一起，

聚集引起了局部布线拥塞问题。在运行 physopt 命令之前，使用如下命令来分散拥塞：

```
set_congestion_options –max_util<number>
```

遗憾的是没有适合所有情况的魔数。这是一种不断尝试的方法。建议用户参阅这个命令的参考页，并根据自己的设计做出自己的明智的决定。

4. 目前的版本不支持多行高单元的布局。这是一个重要的特色，它允许设计人员不仅能布局标准单元，也能自动布局宏单元（如 RAM、ROM、PLL 等）。传统的流程是在运行 physopt 布局前预布局这些宏单元。Synopsys 已声明不久就会给 PhyC 添加这个能力。

10.5 后端流程

Synopsys 最近公布了 PhyC 两个新的附加的选件，它们是 Clock Tree Compiler 和 Route Compiler。激活了这些选件，PhyC 成为唯一提供从 RTL 综合开始到最终 GDSII 完整解决方案的 EDA 工具。整个流程基于一个共同的时序引擎并提供额外的能力，如信号完整性和串扰分析，这个工具变得格外强大。

在写作本书时这些新功能还不可用。因此就没有提供基于这种技术流程的其余部分。没有时钟树和布线编译器的用户可使用自己的布图工具并从时钟树插入阶段（在第 9 章中讲述）继续下去。

10.6 小结

本章描述了 Physical Compiler 的使用和操作。随着这一性能的引入，Synopsys 解决了长期存在的线载模型估计的延迟和最终所得布线设计间差异的问题。

本章还描述了不同的流程和技巧，以及指导用户进行成功的综合、布局和扫描链排序的有用脚本。

本章也讨论了 PhyC 相关的一些问题。虽然这些问题大多能及时

得到解决，但本章的意图是使读者在使用这个版本的 PhyC 时注意到这些问题。

最近公布了能极大增强 Physical Compiler 能力的一些新的附加选件（Clock Tree Compiler 和 Route Compiler）。本章还提到了这些附加选件，然而由于在写作本书时这些选件还不可用，因此就没有描述它们的用法和操作。

第 11 章
SDF 生成——为动态时序仿真

标准延迟格式或 SDF 包括设计中所有单元的时序信息，它为仿真门级网表提供时序信息。

在第 1 章中已经讲到，通过动态仿真验证设计的门级网表不是一个值得推荐的方法。动态仿真方法只在 RTL 级验证设计的功能性。使用动态仿真的门级设计的验证完全依靠测试基准提供的覆盖率，因此不会测到设计中某些没有敏化的路径。相反的，形式验证技术给设计提供了更好的确认。

门级设计验证的动态仿真方法仍是设计人员广泛使用的重要方法。由于这个原因，本章提供了由 DC 和 PT 生成 SDF 文件的简要描述，SDF 文件能用于进行设计的动态时序仿真。此外，提供了一些新的想法和建议以帮助设计人员进行成功的仿真。

请注意本章所介绍的一些内容在前面章节中也有所介绍。由于这是一个重要的论题，因此为了清晰和完整，用完整的一章来介绍 SDF 生成。

11.1 SDF 文件

SDF 文件包括设计中每个单元的时序信息，基本的时序数据由以下几部分组成：

a）IOPATH 延迟。

b）INTERCONNECT 延迟。

c）SETUP 时序检查。

d）HOLD 时序检查。

下面是一个包含两个单元（时序单元输入给一个与门）的时序信息以及两者间连线延迟的 SDF 文件示例：

```
(DELAYFILE
(SDFVERSION "OVI 2.1")
(DESIGN "top_level")
(DATE "Dec 30 1997")
(VENDOR "std_cell_lib")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "1998.08")
(DIVIDER /)
(VOLTAGE 2.70:2.70:2.70)
(PAROCESS "WORST")
(TEMPERATURE 100.00: 100.00: 100.00)
(TIMESCALE 1ns)
(CELL
(CELLTYPE "top_level")
(INSTANCE)
(DELAY
    (ABSOLUTE

        (INTERCONNECT sub1/U1/Q sub1/U2/A1 (0.02:0.03:0.04)
(0.03:0.04:0.05))
        )
    )
)
(CELL
(CELLTYPE "dff1")
(INSTANCE sub1/U1)
(DELAY
    (ABSOLUTE
    (IOPATH CLK Q (0.1:0.2:0.3) (0.1:0.2:0.3))
    )
)
    (TIMINGCHECK
        (SETUP (posedge D) (posedge CLK) (0.5:0.5:0.5))
        (SETUP (negedge D) (posedge CLK) (0.6:0.6:0.6))
```

```
            (HOLD (posedge D) (posedge CLK) (0.001:0.001:0.001))
            (HOLD (negedge D) (posedge CLK) (0.001:0.001:0.001))
        )
    )
    (CELL
    (CELLTYPE "and2")
    (INSTANCE sub1/U2)
    (DELAY
        (ABSOLUTE
            (IOPATH A1 Z (0.16:0.24:0.34) (0.12:0.23:0.32))
            (IOPATH A2 Z (0.11:0.21:0.32) (0.17:0.22:0.34))
            )
        )
    )
    )
```

IOPATH 延迟指定单元延迟，其计算依据输出连线负载和输入信号的转换时间。

INTERCONNECT 延迟是基于路径的点到点的延迟，包括驱动门和被驱动门间的 RC 延迟。它指定了从驱动单元的输出引脚到被驱动单元的输入引脚的连线延迟。

SETUP 和 HOLD 时序检查包括决定每个单元所需的建立和保护时间的数值/时序。这些数字是基于工艺库中的特征值。

11.2　SDF 文件生成

可为布图前或布图后仿真生成 SDF 文件。在将提取的 RC 延迟值和寄生电容反标注到 DC 或 PT 之后，由 DC 或 PT 生成布图后 SDF。布图后的值代表了同设计相关的实际延迟，可用如下命令生成 SDF 文件：

DC 命令

```
write_timing –format sdf-v2.1 –output <filename>
```

PT 命令

```
write_sdf –version [1.0 or 2.1] <filename>
```

注意：PT 默认生成 2.1 版 SDF。

11.2.1 生成布图前 SDF 文件

布图前的数值包括基于线载模型的延迟值，布图前网表也不包括时钟树。因此在生成布图前 SDF 时，有必要近似布线后时钟树的延迟。

为了生成布图前 SDF，下列命令通过定义时钟延迟、扭斜和转换时间来近似布线后时钟树的值：

DC&PT 命令

```
create_clock –period 30 –waveform [list 0 15] [list CLK]
set_clock_latency 2.0 [get_clocks CLK]
set_clock_transition 0.2 [get_clocks CLK]
```

如上所示，通过设置（确定）这些值，设计人员假设所得的 SDF 文件也包含这些值，也就是说，从时钟源到终点（触发器的时钟输入端口）的时钟延迟固定在 2.0，然而情况并非如此。DC 只用上述命令进行静态时序分析，并不将这一信息输出到 SDF 文件。为避免这一问题，设计人员应使 DC 用指定的延迟值而不是自己计算的值。为确保包括 2.0ns 作为时钟延迟，有一个 dc_shell 命令（本节后面会讲到）用于调整所得的 SDF。

在布图前阶段，也应指定时钟转换时间。没能确定时钟的转换时间会导致被驱动的触发器计算的错误值，原因也是在布图前阶段缺少时钟树。缺少时钟树为时钟源强加了高扇出假设，这导致 DC 为整个时钟网络计算出的转换时间较慢。慢的转换时间会影响被驱动的触发器（终点触发器），导致为它们计算大的延迟值。

考虑图 11-1 所示的示意图。虚线描述了布图过程中综合的时钟树缓冲器的布局。在布图前阶段，这些缓冲器不存在。而在时钟源通常有一个缓冲器/单元（如阴影单元所示）。这个单元可能是一个主要用于驱动将来时钟树而由设计人员例化的大驱动器或是一个简单的输入压焊块。假定这是一个输入引脚 A 输出引脚 Z 的输入压焊块（称作 CLKPAD）。在布图前阶段，输出引脚 Z 直接连到所有的终点上。

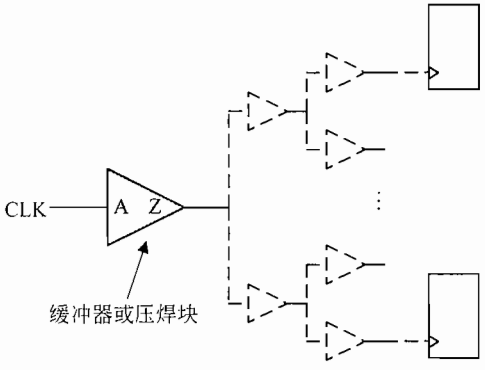


图 11-1　指定时钟树延时

修正 SDF 以反映从源“CLK”到所有终点的 2.0ns 时钟延迟的最容易的方法就是用 2.0ns 代替由 DC 计算的阴影单元（从引脚 A 到引脚 Z）的延迟值。这可通过使用如下 dc_shell 命令来取得：

```
dc_shell -t > set_annotated_delay 2.0 –cell  –from CLKPAD/A –to CLKPAD/Z
```

注意：PT 也有类似的命令。

上述命令以一个指定值 2.0ns 代替了 DC 计算值。这个延迟以单元 CLKPAD 的从引脚 A 到引脚 Z 的 IOPATH 延迟的形式在 SDF 文件中得到反映。

通过修正时钟的延迟值来解决时钟延迟问题，而被驱动的触发器的延迟值会发生什么呢？设计人员作了错误的假定：DC 使用指定的时钟转换只是为了进行静态时序分析，并可能不使用指定的值计算被驱动触发器的延迟。事实并非如此，DC 使用确定的时钟转换值计算被驱动门的延迟。转换值不只用于进行静态时序分析，也用于计算被驱动单元的延迟。因而 SDF 文件含了设计者在布图前阶段近似的延迟值。

11.2.2　生成布图后 SDF 文件

布图后设计包括时钟树信息，因此，布图后 SDF 文件生成不再需要在布图前阶段修正时钟延迟、扭斜和时钟转换时间所需的所有步骤，而是在时钟网络中传播时钟以提供真实的延迟和转换时间。

如第 9 章中所解释的，对于最终的 SDF 生成，只应向 DC 或 PT 反标注提取的寄生电容和 RC 延迟。

当为仿真生成布图后 SDF 文件时，可用如下命令向设计反标注提取的数据并指定时钟信息：

DC&PT 命令

```
read_sdf <interconnect RC's in SDF format>

source <parasitic capacitances in set_load format>

read_parasitics <DSPF, RDPF or SPEF file for clocks + other critical nets>

create_clock –period 30 –waveform [list 0 15] [list CLK]
set_propagated_clock   [get_clocks CLK]
```

11.2.3 时序检查相关问题

有时在仿真过程中，产生的未知值（X）会导致仿真失败。产生这些未知值是由于建立保持时序检查违例造成的。大多数时候，这些违例是真实的，然而也有设计者可能希望忽略同设计的某些部分相关的一些违例，但仍验证其余的情况。通常这是办不到的，因为仿真器无法在选择的基础上关闭 X-生成。

通常对整个设计几乎所有仿真器都提供忽略时序违例的能力，它们不具有忽略设计中一个单元的单独实例的时序违例的能力。由于这个原因，设计人员经常不得不修改仿真库或接受失败的结果。

由于只能对一个单元关闭 X-生成，因此修改仿真库也不是一个可行的方法。这一单元可以在设计中多次例化。关闭这个单元的 X-生成会阻止仿真器为设计中这一个单元的所有实例生成 X。这绝对不是我们想要的，因为它可能掩盖了设计中别处存在的真正的时序问题。

例如，一个设计可能包含多个时钟域并且数据通过同步逻辑从一个时钟域穿到另一个时钟域。虽然这一逻辑在加工了的器件上会工作得很好，但仿真时它可能引起保持时间违例。这会导致设计的仿真失败。

另一个例子同综合方法的类型相关。一些设计人员喜欢只在布图

后修正保持时间违例。没有在布图前 SDF 文件中声称保持时间值为假的或删除它们，可能导致仿真器为违例的触发器生成一个 X（未知值）。X 可传播到逻辑的其余部分，导致整个仿真失败。

为避免这些问题并成功进行仿真，可能需要有选择地对构建在 SDF 文件中的建立和保持时间的值设定为假。SDF 文件是基于实例的（而不是基于单元的），因此容易获得时序检查的选择定位。不用手动从 SDF 文件中删除建立和保持时间结构，更好的方法是仅对违例的触发器在 SDF 文件中把建立和保持时间置零，也就是说，用零来取代现有的建立和保持时间的数值。向仿真器反标注为零值的建立和保持时间，可防止它生成未知值（如果建立和保持时间都是零，就不会有任何违例），从而使得仿真能顺利地运行。可用如下的 dc-shell 命令执行这一步：

```
dc_shell -t > set_annotated_check 0 –setup –hold          \
                                          –from REG1/CLK   \
                                          –to REG1/D
```

注意：PT 也有类似的命令。

11.2.4　虚假延迟计算问题

这一论题在第 4 章中作了介绍，但将其包括在这里是为了完整起见。

单元的延迟计算是基于单元的输入转换时间和输出负载电容，单元的输入转换时间是基于驱动单元（前一单元）的转换延迟求得的。如果驱动单元不只包含一个时序弧，那么将用最坏的转换时间作为被驱动单元的输入。当为了仿真生成 SDF 文件时，这将会引起一个大的问题。

考虑如图 11-2 所示的逻辑，信号 reset 和 signal_a 是实例 U1 的输入。假定信号 reset 不是关键的，而 signal_a 是我们真正感兴趣的。信号 reset 是一个慢信号，因而这个信号的转换时间要比具有较快转换时间的 signal_a 更长。这将为单元 U1 计算出两个转换延迟值（从 A 到 Z 的 2ns 和从 B 到 Z 的 0.3ns）。当生成 SDF 时，这两个值将作为单元 U1 单元延迟的部分单独写出。然而现在问题出现了，DC 使用这两个值中的哪一个为单元 U2 计算输入转换时间？DC 使用前一

个门（U1）的最坏（最大）转换值作为被驱动门（U2）的输入转换时间。由于 reset 信号的转化时间要比 signal_a 的长，因此将使用 2ns 作为 U2 的转入转换时间。这将为单元 U2（阴影单元）计算一个大的延迟值。

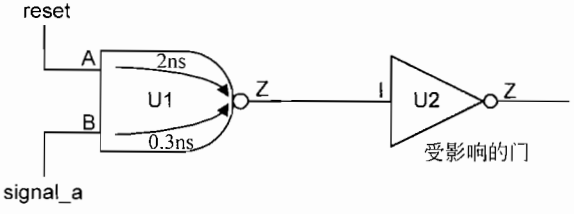


图 11-2 虚假延时计算

为避免这一问题，需要指示 DC 不要为从单元 U1 的引脚 A 到引脚 Z 的时序弧进行延迟计算。这一步应在写出 SDF 之前执行。可用如下 dc_shell 命令执行这一步：

```
dc_shell -t > set_disable_timing U1 –from A –to Z
```

遗憾的是，这个问题在静态时序分析过程中也存在。没有取消虚假路径的时序计算导致为被驱动单元计算大的延迟值。

11.2.5 组合

以下的DC脚本将上面提供的所有信息组合在一起并且用来生成用于设计示例 tap controller 时序仿真的布图前和布图后 SDF。

布图前 SDF 生成的 DC 脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

set_wire_load_model LARGE
set_wire_load_mode top
set_operating_conditions WORST
```

```
create_clock –period 33 –waveform [0 16.5] tck
set_clock_latency 2.0   [get_clocks tck]
set_clock_transition 0.2   [get_clocks tck]

set_driving_cell –cell BUFF1 –pin Z [all_inputs]
set_drive 0 [list tck trst]
set_load 50 [all_outputs]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

# Approximate the clock tree delay
Set_annotated_delay 2.0 –cell –from CLKPAD/A \
                              to CLKPAD/Z
# Assuming, only REG1 flop is violating hold-time
set_annotated_check 2.0 –setup –hold              \
                        –from REG1/CLK –to REG1/D

write_timing   –format sdf-v2.1    \
               –output $active_design.sdf
```

布图后 SDF 生成的 DC 脚本

```
set active_design tap_controller

read_db $active_design.db

current_design $active_design
link

set_operating_conditions BEST

source capacitance.dc     # actual parasitic capacitances
read_timing rc_delays.sdf   # actual RC delays

create_clock –period 33 –waveform [list 0 16.5] tck
set_propagated_clock [get_clocks tck]

set_driving_cell –cell BUFF1 –pin Z [all_inputs]
set_drive 0 [list tck trst]
```

```
set_load 50 [all_outputs]

set_input_delay 20.0 –clock tck –max [all_inputs]
set_output_delay 10.0 –clock tck –max [all_outputs]

# Assuming, only REG1 flop is violating hold-time
set_annotated_check 0 –setup –hold              \
                        -from REG1/CLK –to REG1/D

write_timing    –format sdf-v2.1    \
                –output $active_design.sdf
```

11.3 小结

SDF 文件在 ASIC 领域广泛用于进行动态时序仿真。本章简要总结了与后续讨论相关的 SDF 文件的内容。

本章也讨论了从 DC 和 PT 生成用于布图前和布图后仿真的 SDF 文件的步骤。与命令一起描述的各种有用的技巧来调整 SDF 以进行成功仿真。这些包括在布图前阶段修正时钟延迟和时钟转换，以及为了成功地仿真，避免来自设计的选择逻辑的未知的传播。

最后一节收集了所有信息并以布图前和布图后 SDF 生成的 DC 脚本的形式组合在一起。

第 12 章
PRIMETIME 基础

PrimeTime（PT）是 Synopsys 的签收品质（sign-off quality）的静态时序分析工具。静态时序分析（STA）无疑是设计流程中最重要的一步，它决定了设计是否在所要求的速度下工作，PT 分析设计中的时序延迟并标出必须改正的违例。

与 DC 类似，PT 提供图形用户界面（GUI）和命令行界面。GUI 界面包含帮助图形化分析设计的各种窗口。虽然 GUI 界面是一个不错的起始点，但大多数用户很快地转用命令行界面。因此，本章专注于 PT 的命令行界面。

本章介绍了 PT 的基础知识，其中一节介绍了 PT 使用的 Tcl 语言。本章也选择介绍了一些 PT 命令，它们用于进行成功的 STA，并且也给设计者调试设计中可能的时序违例提供方便。

12.1 导言

PT 是一个不在 DC 工具套件中集成的单独的工具，它是一个和 DC 并行工作的单独的工具。PT 与 DC 有一致的命令，它们生成类似的报告，并支持共同的文件格式。此外 PT 也能生成 DC 用于综合和优化的时序断言，PT 的命令行界面是基于称为 Tcl 的工业标准语言。与 DC 的内部 STA 引擎相比，PT 更快，占用的内存更少，并且还有其他的一些特色。

12.1.1 调用 PT

使用 pt_shell 命令调用命令行模式的 PT，或使用命令 primetime 调用 GUI 模式的 PT。

命令行模式：

```
> pt_shell
```

GUI 模式：

```
> primetime
```

12.1.2 PrimeTime 环境

一经调用，PT 就寻找名为“.synopsys_pt.setup”的文件并默认包括它。它首先在当前目录寻找此文件，若不能找到，则在使用 PT 安装处的默认设置文件之前在用户的主目录中寻找此文件。这个文件包含了定义 PT 使用的设计环境的必要设置变量，示例如下：

```
set search_path   [list . /usr/golden/library/std_cells]
set link_path [list {*} ex25_worst.db, ex25_best.db]
```

变量 search_path 定义了一个列表，它包含了当查找库和设计时需查看的目录。它免除了当引用库和设计时需键入完整文件路径的麻烦。

变量 link_path 定义了一个库列表，它包含用于连接设计的单元。这些库在 search_path 指定的目录中查找。上例中 link_path 变量定义的列表中有三个元。“*”表示内存中所加载的设计，而另两个是与最佳和最差情况相关的标准单元工艺库名。

如果我们不想使用“.synopsys_pt.setup”文件，另一个广泛使用的设置环境的方法是使用 source 命令。source 命令就像 DC 的 include 命令一样工作。它包括并运行文件，就好像它是当前环境中的一个脚本。这个命令在 pt_shell 中调用。例如：

```
pt_shell > source ex25.env
```

12.1.3　自动命令转换

大多数 PT 命令同 DC 命令类似，例外的是，PT 基于 Tcl 使用 Tcl 语言格式。这需要 DC 命令在 PT 能利用它们之前转换为 Tcl 格式。

PT 提供了一个转换脚本，它可将几乎所有的 dc_shell 命令转换到基于 Tcl 的格式 pt_shell。这个脚本称为 transcript 并由 Synopsys 作为单独的应用程序提供。这个脚本由 UNIX 命令解释器执行如下：

```
> transcript  <dc_shell script filename> <pt_shell script filename>
```

12.2　Tcl 基础

Tcl 提供了大部分的基本程序结构——变量、运算符、表达式、控制流、循环和过程等。另外，Tcl 也支持大多数的 UNIX 命令。

Tcl 程序是一个命令列表。命令可以通过称为命令置换的过程嵌入到其他命令中。

使用 set 命令定义变量并给它们赋值。例如：

```
set clock_name clk

set clock_period 20
```

值可以是数字或字符串——Tcl 不区分数值型变量和字符串型变量。在算术场合，它自动使用数字值。上例中，设变量 clock_name 为字符串 CLK，设变量 clock_period 为数值 20。通过在变量名前加$来引用变量。如果在变量名前没有加$，那么 Tcl 将其视为字符串。

```
create_clock $clock_name –period 20 –waveform [0 10]
```

通过 expr 命令进行算术运算。这个有用的技术提供了贯穿于整个脚本的全局参数化的方法。

```
expr $clock_period / 5
```

上述命令返回值为 4。Tcl 提供所有标准算术运算，如*、/、+、–等。

12.2.1 命令置换

命令返回的值可用于其他命令。命令的嵌套可用方括号“[”和“]”完成。当一个命令装入方括号里，它首先求值，然后它的值取代命令的位置。例如：

```
set clock_period 20

set inp_del [expr $clock_period / 5]
```

上述命令首先求方括号内表达式的值，然后用所求得的值替换命令——在此情况下，inp_del 继承了等于 4 的值，命令也可任意深度的嵌套。例如：

```
set clock_period 20

set inp_del [expr [expr $clock_period / 5] + 1 ]
```

上例有两层嵌套。内层命令返回由 clock_period 的值除以 5 得到的等于 4 的值，外层的 expr 命令给内层命令的结果加 1。这样 inp_del 被设为等于 5 的值。

12.2.2 列表

列表表示一个对象的集合——这些对象可以是字符串或是列表。用大括弧包括一组项目的最基本形式形成一个列表。

```
set clk_list {clk1 clk2 clk3}
```

上例中，set 命令创建了一个名为 clk_list 的列表，它包括三个元素：clk1、clk2 和 clk3。

创建列表的另一个方法是用 list 命令，它的典型应用是用在命令置换中。例如，下面的 list 命令创建同前面 set 命令同样的列表：

```
list clk1 clk2 clk3
```

list 命令适于在命令置换中使用，因为它的返回值是一个列表。

set 命令的示例也可写作：

```
set clk_list [list clk1 clk2 clk3]
```

Tcl 提供了一组操作列表的命令—concat、join、lappend、lindex、linsert、list、llength、lrange、lreplace、lsearch、lsort 和 split。例如，concat 将两个列表连接在一起并返回一个新的列表。

```
set new_list [concat [list clk1 clk2] [list clk3 clk4] ]
```

上例中，变量 new_list 是列表 clk1&clk2 和列表 clk3&clk4 连接后得到的结果，每一个列表都是由 list 命令生成的。

从概念上讲，简单的列表是包含由空格分隔的元素的字符串。大多数情况下，下边两个是等价的：

```
{clk1 clk2 clk3}
"clk1 clk2 clk3"
```

某些情况下，第二种表示更为可取，因为它允许变量置换，而第一种表示则不行。例如：

```
set stdlibpath [list "usr/lib/stdlib25" "usr/lib/padlib25"]
set link_path "/project/bigdeal/lib $stdlibpath"
```

其他 list 命令语法的更详细内容，请参考任何的 Tcl 语言标准书。

12.2.3　流控制和循环

如同其他的脚本和程序语言，Tcl 为流控制提供了 *if* 和 *switch* 命令。它也为循环提供了 *for* 和 *while* 循环。*if* 命令可以与 *else* 或 *else if* 语句一起使用，以完整地指定过程流。*if*、*else if* 和 *else* 语句的参数通常是列表，包含在大括弧里以避免任何置换。例如：

```
if {$port == "clk"} {
    create_clock –period 10 –waveform [list 0 5] $port
} elsif {$port == "clkdiv2"} {
    create_generated_clock –divide_by 2 –source clk $port
} else {
    echo "$port is not a clock port"
}
```

12.3 PrimeTime 命令

PT 使用类似于 DC 的命令来执行时序分析和相关的功能。因为所有相关的 dc_shell 命令都在第 6 章作了详细解释，所以本节对所有相关的命令不再详细解释。

12.3.1 设计输入

PT 不像 DC 能用 HDL Compiler 读取 RTL 源文件，它是静态分析引擎，只能读取映射后的设计。这确定了设计输入到 PT 的基础。其中，PT 的输入可以是 db、Verilog、VHDL 或 EDIF 格式的文件。下列每一格式的 pt_shell 命令用于将设计读入到 PT：

```
read_db –netlist_only <design name>.db          #db format

read_verilog <design name>.sv                   #verilog format
read_vhdl   <design name>.svhd                  #vhdl format
read_edif   <design name>.edf                   #EDIF format
```

由于 db 格式的网表也包含约束和/或环境属性（由设计者保存的），read_db 命令使用-netlist_only 选项指示 PT 只加载结构化网表。这可防止 PT 读取与设计相关的约束和/或其他属性。只加载结构化网表。

12.3.2 时钟规范

时钟规范背后的概念与第 6 章为 DC 所描述的一样。由于两者间格式的差异，因此存在细微的语法差异。然而，因为时钟规范变得复杂，尤其是如果有时钟分频内部生成的时钟，本节包括完整的 PT 时钟规范技术和语法。

12.3.2.1 创建时钟

主时钟定义如下：

```
create_clock –period <value>
```

```
                    –waveform {<rising edge><falling edge>}
                      <source list>
pt_shell > create_clock –period 20 –waveform {0 10}    \
                                        [list CLK]
```

上例创建一个名为 CLK 的时钟，它的周期为 20ns，上升和下降边沿分别在 0ns 和 10ns 处。

12.3.2.2　时钟延迟和时钟转换

下列命令用于指定时钟延迟和时钟转换。这些命令主要用于布图前 STA，并在第 13 章进行详细的解释。

```
set_clock_latency      <value> <clock list>

set_clock_transition   <value> <clock list>
```

```
pt_shell > set_clock_latency      2.5 [get_clocks CLK]

pt_shell > set_clock_transition   0.2 [get_clocks CLK]
```

上述命令定义 CLK 端口的时钟延迟为 2.5ns，固定时钟转换值为 0.2ns。

12.3.2.3　传播时钟

在设计中插入时钟树之后布图工具通常接着要进行的是传播时钟，网表被带回 PT 进行 STA。时钟通过网表中的整个时钟树网络传播以确定时钟延迟。换言之，对通过时钟树中每个单元的延迟和单元间的互连线延迟都要加以考虑。

下面的命令指示 PT 通过时钟网络传播时钟：

```
set_propagated_clock <clock list>
```

```
pt_shell > set_propagated_clock   [get_clocks CLK]
```

12.3.2.4　指定时钟扭斜

时钟扭斜或 Synopsys 称作的时钟不确定，是在触发器时钟引脚处时钟到达时间的差异。在同步设计中，数据在一个时钟边沿由触发器开始发送，在另一个时钟边沿（通常为下一时钟沿）由另一触发器

接收。如果两个时钟沿（发送和接收）源自同一个时钟，那么理想的两个边沿间应有准确的一个时钟周期的延迟。时钟扭斜妨碍了这种理想情况。由于布线延迟的差异（或门控时钟情形），接收时钟沿可早到或迟到。早到可能导致建立时间违例，而迟到则可能导致保持时间违例。因此，在布图前阶段必须指定时钟扭斜以生成鲁棒的设计。

通过下面的命令指定时钟扭斜：

```
set_clock_uncertainty <uncertainty value>
                      –from <from clock>
                      –to <to clock>
                      –setup
                      –hold
                      <object list>
```

下例中，0.6ns 用来表示时钟信号（CLK）的建立和保持时间。

```
pt_shell > set_clock_uncertainty 0.6 [get_clocks CLK]
```

选项–setup 用于给建立时间检查应用不确定值，而–hold 选项给保持时间检查应用不确定值。必须注意的是，建立和保持的不同值不能在单个命令中实现。为这一目的，必须使用两个单独的命令。例如：

```
pt_shell > set_clock_uncertainty 0.5 –hold [get_clocks CLK]
pt_shell > set_clock_uncertainty 1.5 –setup [get_clocks CLK]
```

使用-from 和-to 选项也能指定时钟扭斜，这对包括多个时钟域的设计有用，例如：

```
pt_shell > set_clock_uncertainty 0.5 -from [get_clocks CLK1]  \
                                     -to [get_clocks CLK2]
```

12.3.2.5 指定生成的时钟

这是一个 DC 所不具备的重要特色。设计经常包含内部产生的时钟。PT 允许用户通过 create_generated_clock 命令定义生成时钟与源时钟间的关系。这是方便的，因为布图前的脚本能以最少的改变用于布图后。

在布图后时序分析中，插入了时钟树并且通过时钟树缓冲器传播时钟信号来计算时钟延迟。用户选择定义独立于时钟源的分频时钟（通过在分频逻辑子模块的输出引脚上定义时钟）。然而，这个方法

迫使设计人员给从时钟源到分频逻辑模块的时钟延迟手动添加从分频模块到设计的其余部分的时钟树延迟。

通过上述命令建立分频时钟，则这两个时钟在布图前和布图后阶段都保持同步。

```
create_generated_clock   –name <divided clock name>
                         –source <primary clock name>
                         –divide_by <value>
                         <pin name>
```

```
pt_shell > create_generated_clock –name DIV2CLK                \
                                  –source      CLK –divide_by 2 \
                                  blockA/DFF1X/Q
```

上例在属于 blockA 的 DFF1X 单元的引脚 Q 处创建一个生成时钟。生成时钟名为 DIV2CLK，其频率为时钟源 CLK 频率的一半。

12.3.2.6　门控时钟检查

对低功耗应用，设计人员经常在设计中采用门控时钟。这个技术允许设计人员仅当需要时才使能时钟。如果（对门控逻辑）不满足建立和保持时间要求，门控逻辑可产生箍制时钟或毛刺。PT 允许设计人员对门控逻辑指定建立/保持要求，如下所示：

```
set_clock_gating_check   –setup <value>
                         –hold <value>
                         <object list>
```

```
pt_shell > set_clock_gating_check –setup 0. 5 –hold 0.01 CLK   \
```

上例表明 PT CLK 时钟网络中所有门的建立时间和保持时间要求分别是 0.5ns 和 0.01ns。

单个单元的门控检查可通过在对象列表中指定单元名来取得。例如：

```
pt_shell > set_clock_gating_check –setup 0.05 –hold 0.01     \
                                  [get_lib_cell stdcell_lib/BUFF4X]
```

默认情况下，PT 以零值作为建立和保持时间的值来进行门控检查——除非库为用于门控时钟的单元包括特定的建立和保持时间值。

如果门控单元包含建立/保持时间检查，那么门控检查值可自动由 SDF 文件得到。

门控时钟检查仅对组合单元进行，并且门控检查不能在两个时钟间进行。

12.3.3 时序分析命令

本节有选择地描述用于进行 STA 的 PT 命令，并只列举这些命令最常用的选项。

set_disable_timing：这个命令的应用包括禁止一个单元的时序弧以断开组合反馈环，或指示 PT 把一特殊时序弧（进而路径段）排除在分析之外。

```
set_disable_timing   –from <pin name>
                     –to <pin name>
                     <cell name>
pt_shell > set_disable_timing –from A1 –to ZN {INVD2}
```

report_disable_timing：此命令用于显示被用户或 PT 禁止的时序弧。报告用如下标记标识的各个被禁的路径：

标记：　u：被用户禁止时序路径。

l：被 PT 断开的时序环。

c：在情况分析中禁止的时序路径。

set_input_transition：是 set_driving_cell 命令的另一选择。它设置不依赖于连线负载的固定转换时间。在设计的输入/双向端口指定这个命令。

```
set_input_transition <value> <port list>

pt_shell > set_input_transition 0.2 [all_inputs]

pt_shell > set_input_transition 0.4 [list in1 in2]
```

set_timing_derate：用于减免时序报告中所示的延迟值。PT 提供了这一强大的能力，它在给整个设计添加额外的时序余量方面有用。减免的量由用户指定的固定值所控制。在生成时序报告之前，原先的延迟值要与这个值相乘。

```
set_timing_derate –min <value> -max <value>

pt_shell > set_timing_derate –min 0.2 –max 1.2
```

set_case_analysis：此命令进行情况分析，也是 PT 所提供的最有用的特色之一。当进行 STA 时，这个命令用于给端口（或引脚）设置固定的逻辑值。

```
set_case_analysis [0 | 1] <port or pin list>

pt_shell > set_case_analysis 0 scan_mode
```

这个命令的应用包括禁止在特殊工作模式下无效的时序路径。例如，在上例中，scan_mode 端口在功能模式（正常工作）和测试工作模式间切换设计。在 scan_mode 端口设置的零值被传播到被这个端口驱动的所有单元，这导致了与 scan_mode 端口相关的所有单元的某些时序弧失效。由于测试逻辑通常是非时序关键的，失效的非时序关键路径的时序弧促使真正时序关键路径被确认和分析。第 13 章将进一步解释这个命令的用法。

remove_case_analysis：此命令用于删除由上面命令设置的情况分析值。

```
remove_case_analysis <port or pin list>

pt_shell > remove_case_analysis   scan_mode
```

report_case_analysis：此命令用于显示由用户设置的情况分析值。PT 显示一个报告，确认引脚/端口列表以及对应的情况分析值。

```
pt_shell > report_case_analysis
```

report_timing：类似于 DC，这个命令用于生成设计中路径段的时序报告。这个命令被广泛使用并提供了充分的灵活性、这对于明确地关注设计中单个路径或路径集合是有帮助的。

```
report_timing –from <from list> -to <to list>

                    –through <through list>
                    –delay_type <delay type>
                    –nets –capacitance –transition_time
```

```
                                    –max_paths <value> –nworst <value>
```

–from 和–to 选项使用户易于为分析定义路径。因为可以有多条从起点到终点的路径，–through 选项可用于为时序分析进一步分离所需的路径段。

```
pt_shell > report_timing –from [all_inputs]                    \
                         –to [all_registers –data_pins]

pt_shell > report_timing –from in1                             \
                         –to blockA/subB/carry_reg1/D          \
                         –through blockA mux1/A1
```

–delay_type 选项用于指定终点处报告的延迟类型，它可接受的值是 max、min、min_max、max_rise、max_fall、min_rise 和 min_fall。PT 默认使用 max 类型，它报告两点间的最大延迟。min 类型选项用于显示两点间的最小延迟。max 类型用于分析设计的建立时间，而 min 类型用于进行保持时间分析。其他的类型不常用，建议用户参考 PT 用户指南中有关它们用法的完整解释。

```
pt_shell > report_timing –from [all_registers –clock_pins]     \
                         –to [all_registers –data_pins]        \
                         –delay_type min
```

–nets、–capacitance 和–transition_time 选项是 report_timing 命令最有用和最常用的选项。这些选项帮助设计者调试特殊的路径以追踪可能违例的原因。–nets 选项在路径报告中显示每个单元的扇出，而–capacitance 和–transition_time 选项分别报告连线上的集总电容和每个驱动或负载引脚的转换时间（摆率）。没有这些选项将导致时序报告中没有以上提及的信息。

```
pt_shell > report_timing –from in1                             \
                         –to blockA/subB/carry_reg1/D          \
                         –nets –capacitance –transition_time
```

–nworst 选项指定了为每个终点报告的路径数，而–max_paths 选项为不同的终点定义了每个路径组报告的路径数。这两个选项的默认值都是 1。

```
pt_shell > report_timing –from [all_inputs]                    \
```

```
                    –to [all_registers –data_pins]          \
                    –nworst 1000 –max_paths 500
```

report_constraint：类似于 DC，PT 中的这个命令检查由设计者或工艺库定义的 DRC。另外，此命令对确定设计关于建立和保持时间违例的“全面状况”也是有用的。这个命令的语法以及最常用的选项如下：

```
report_constraint    –all_violators –max_delay
                     –max_transition –min_transition
                     –max_capacitance –min_capacitance
                     –max_fanout –min_fanout
                     –max_delay –min_delay
                     –clock_gating_setup –clock_gating_hold
```

选项–all_violators 显示所有约束的违例者。通常，这个选项用于快速地确定设计的全面状况。报告总结了对某一约束从最大到最小的所有违例者。

```
pt_shell > report_constraint –all_violators
```

选择性的报告可用 –max_transition、–min_transtion、–max_capacitance、–min_capacitance、–max_fanout、–min_fanout、–max_delay 和–min_delay 等选项获得。–max_delay 和–min_delay 选项报告所有建立和保持时间违例的总结，而其他的则报告 DRC 违例。–clock_gating_setup 和–clock_gating_hold 选项用于显示门控时钟单元的建立/保持时间报告。此外，这个命令还有其他的可用选项，它们可能会对设计者有用。选项的全面细节可在 PT 用户指南中找到。

```
pt_shell > report_constraint  –max_transition

pt_shell > report_constraint  –min_capacitance

pt_shell > report_constraint  –max_fanout

pt_shell > report_constraint  –max_delay –min_delay

pt_shell > report_constraint  –clock_gating_setup          \
                              –clock_gating_hold
```

☺ 最初使用 report_constraint 命令是为了确定违例的数量和数目。产生的报告提供了设计的全面状况的大体估计。依据违例的严重程度，可能需要重新综合设计。为进一步分离违例的原因，report_timing 命令应用于指向违例路径以显示完整的时序报告。

report-bottleneck：这个命令用于确定对设计中多个违例有贡献的叶单元。例如，设计的一些违例路径段可共有一个共同的叶单元。改变这个叶单元的大小（增大或减小）可以改善所有违例路径段的时序（进而除去违例）。这个命令的语法以及最常用的选项如下：

```
report_bottleneck    –from <from list> -to <to list>
                     –through <through list> –max_cells <value>
                     –max_paths <value> –nworst_paths <value>
```

–from 和–to 选项便于用户为瓶颈分析定义路径。因为可有多条从起点到终点的路径，–through 选项可用于为瓶颈分析进一步分离所需的路径段。

```
pt_shell > report_bottleneck    –from in1                          \
                                –to blockA/subB/carry_reg1/D       \
                                –through blockA/mux1/A1
```

顾名思义，–max_cells 选项指定报告的叶单元的数目，默认值是 20。

选项–nworst_paths 指定为每个终点报告的路径数目，而–max_paths 选项为不同的终点定义每个路径组报告的路径数目，这两个选项的默认值都是 100。

```
pt_shell > report_bottleneck –from in1                             \
                             –to blockA/subB/carry_reg1/D          \
                             –through blockA/mux1/A1               \
                             –max_cells 50                         \
                             –nworst_paths 500 –max_paths 200
```

12.3.4 其他各种命令

write_sdf：此命令生成包含设计中每个实例的延迟和时序检查的 SDF 文件。在布图前阶段 PT 使用线载模型估计单元延迟。在布图后，

当生成 SDF 文件时，PT 使用实际标注的延迟（来自物理版图）。这个命令的语法及最常用的选项为：

```
write_sdf  –version 1.0 | 2.1
           –no_net_delays
           –no_timing_checks
           <sdf output filename>
```

除非明确指出，PT 默认生成 SDF2.1 版本格式的 SDF 文件。

选项 -no_net_delays 指定互连延迟（SDF 文件中的 INTERCONNECT 域）不在 SDF 文件中单独写出。在这种情况下，它们包含在每个单元的 IOPATH 延迟的部分中。这一选项主要用在布图前阶段，因为事实上互连延迟是基于线载模型的。然而，基于布线后设计的互连延迟是真实的。因此，通常当生成布图后 SDF 文件时，应避免这个选项。

```
pt_shell > write_sdf –no_net_delays top_prelayout.sdf
```

```
pt_shell > write_sdf top_postlayout.sdf
```

选项-no_timing_checks 的指定迫使 PT 忽略来自 SDF 文件的时序检查部分（TIMING CHECK 域）。如第 11 章所描述的，时序检查部分包括建立/保持/宽度的时序检查。这个选项对生成用于通过动态仿真只验证设计的功能性而不需要费心检查建立/保持/宽度的时序违例的 SDF 文件是有用的。一旦设计通过了功能验证，即可生成完整的 SDF（没有-no_timing_checks 选项）。

```
pt_shell > write_sdf –no_timing_checks top_prelayout.sdf
```

write_sdf_constraints：这个命令类似于 DC 中的 write_constraints 命令，并且执行同样的功能。它用于生成布图工具以进行时序驱动布图的 SDF 格式的路径时序约束。这个命令的语法及最常用的选项为

```
write_sdf_constraints –version <1.0 | 2.1>
                       –from <from list> –to <to list>
                       –through <through list>
                       –cover_design

                       –slack_lesser_than <value>
                       –max_paths <value> –nworst <value>
```

<constraint filename>

除非明确指出，PT 默认生成 SDF2.1 版本格式的约束文件。–from、–to 和–through 选项便于用户指定写到约束文件中的某一路径。

选项–nworst 为每个终点指定写到约束文件中的路径数，而–max_paths 选项为每个约束组定义要考虑的路径数，这两个选项的默认值都是 1。这些选项的默认设置通常对大多数设计而言是足够的。

```
pt_shell > write_sdf_constraints –from in1                    \
                                –to blockA/subB/carry_reg1/D  \
                                –through blockA/mux1/A1       \
                                 tdl.sdf
```

–cover_design 选项用于生成刚好足够的唯一路径时序约束以覆盖设计中的每个路径段的最差路径。当它被指定时，忽略所有其他选项如–nworst、–to、–from 和–through。虽然 Synopsys 推荐这个选项，但也应审慎地使用它。因为它可引起长的运行时间，尤其是对大型设计。

```
pt_shell > write_sdf_constraints –cover_design tdl.sdf
```

另一方法是使用–slack_lesser_than 选项，它指定忽略具有大于指定松弛值的任何路径。这意味着可认为负松弛值的路径段是最为关键的并具有最高的优先权。这样通过为这个选项指定一个小的值可全部选中所有的关键路径，进而写出到约束文件。忽略所有高松弛值（不太关键的路径）。

```
pt_shell > write_sdf_constraints –slack_lesser_than 1.5 tdl.sdf
```

swap_cell：这个命令可用另一个有同样的引脚的单元去替换设计中已有的单元。

```
swap_cell <cell list to be replaced> <new design>
```

例如，如果一条路径由于保持时间违例而失败，为了修正时序违例，在不改变网表情况下，可通过缩小路径中某一叶单元的大小来观查松弛的影响。在这种情况下，swap_cell 命令用于在命令行中用另一个包括同样引脚的单元替换现有的单元。

```
pt_shell > swap_cell {U1} [get_lib_cell stdcell_lib/AND2X2]
```

上例中，设计中的实例 U1（具有 8X 驱动强度的二输入与门）用来自“stdcell_lib”工艺库的 AND2X2 门（2X 驱动强度）所替换。

12.4　小结

静态时序分析是整个 ASIC 芯片综合流程中最关键步骤之一。本章介绍了 PrimeTime，包括 PrimeTime 调用及其环境设置。

PrimeTime 是一个独立的静态时序分析工具，它基于普遍采用的 EDA 工具语言 Tcl。本章用一小节介绍 PrimeTime 环境中的 Tcl 语言，便于设计者编写 Prime Time 脚本并在此基础上生成复杂的脚本。

最后一节包括了所有相关的 PrimeTime 命令，它们可用于进行静态时序分析，设计调试和写 SDF 格式的延迟信息。此外，本节也包括了布图前和布图后设计输入和时钟规范的讨论。

第 13 章

静态时序分析

——使用 PrimeTime

制造能够工作的芯片的关键通常在于对其设计能够成功地完成静态时序分析。PT 是 Synopsys 用于进行静态时序分析的独立工具。它不仅检查由设计规范定义的设计所需的约束，而且也对设计进行全面分析。这个能力使得 STA 成为整个设计流程中最重要的步骤之一，并且许多设计人员将其用作 ASIC 厂商的签收准则。

本章介绍了设计流程中使用 PT 的部分。它覆盖了 ASIC 设计流程的布图前和布图后两个阶段。

由于 STA 同整个综合流程紧密结合在一起，因而本章的部分内容可能重复本书其他章节的一些内容。

13.1 为何要进行静态时序分析

运用动态仿真分析门级设计的传统方法对大型复杂设计造成了瓶颈。目前的发展趋势是片上系统（SOC），这可导致百万门级的 ASIC。通过动态仿真验证这样一个设计成为设计人员的梦魇，并且由于长的运行时间（通常几天，有时几周），因此证明该方法不可行。此外，动态仿真依赖用于验证的测试基准的品质和覆盖率。只测试敏化的部分逻辑，而设计的剩余部分未能测试到。为了克服这个问题，设计人员现在求助于其他验证方法，如 STA 验证时序，以及形式验证技术对照源 RTL 验证门级网表的功能性。然而，验证源 RTL 的功能性仍需要广泛的测试基准集。这样只有在 RTL 级验证设计的功能

性才需要动态仿真，这导致运行时间的极大减少。

STA 方法比动态仿真要快得多且可验证门级设计的所有时序。由于综合和 STA 引擎的本质相似，静态时序分析极适合验证综合的设计。

13.1.1 分析什么?

通常，对设计进行如下 4 种类型的分析：

（1）从原始输入到设计中所有触发器；

（2）从触发器到触发器；

（3）从触发器到设计的原始输出；

（4）从设计的原始输入到原始输出；

所有这 4 种分析都能用如下命令来完成：

```
pt_shell > report_timing    –from [all_inputs]                    \
                            –to [all_registers –data_pins]

pt_shell > report_timing    –from [all_registers –clock_pins]     \
                            –to [all_registers –data_pins]

pt_shell > report_timing    –from [all_registers –clock_pins]     \
                            –to [all_outputs]

pt_shell > report_timing    –from [all_inputs]                    \
                            –to [all_outputs]
```

虽然，上面所使用的命令是为分析生成报告并成为单个文件的更简洁的方法，然而，PT 要用较长的时间进行每个操作。如果要想 PT 使用更少的时间生成同样的结果，那么使用下面的命令：

```
pt_shell > report_timing –to [all_registers –data_pins]
pt_shell > report_timing –to [all_outputs]
```

13.2 时序例外

在大多数设计中，可能有表现时序例外的路径。比如，逻辑的某些部分可能被设计为多周期路径，而其余的可能只是虚假路径。因此，在分析设计之前，必须使 PT 意识到这些路径表现的特殊行为。如果

没有对它们进行这样的指定，PT 可能会报告多周期路径时序违例。设计中不实际的路径段也必须确认并指定为虚假路径，以防止 PT 为这些路径生成时序报告。

13.2.1　多周期路径

在默认情况下，PT 把设计中所有的路径视为单周期的并据此进行 STA，也就是说，使用第一个时钟沿从驱动触发器获得数据，并由接收触发器使用第二个时钟沿所捕获。这意味着接收触发器必须在一个时钟周期内（单时钟周期）接收数据。在多周期模式下，数据可用不止一个时钟周期到达它的目的地。数据到达其目的地所花费的时间由下面命令中所用的乘数决定：

```
set_multicycle_path  <multiplier value>
                     –from <from list> –to <to list>
```

图 13-1 说明了单周期建立/保持时间关系和多周期建立/保持时间

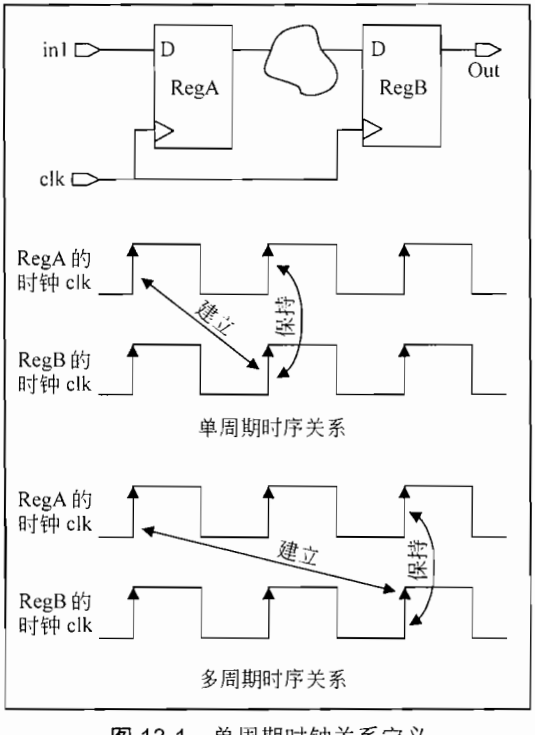


图 13-1　单周期时钟关系定义

关系间的比较。在多周期定义中，乘数 2 用于告知 PT regB 处的锁存数据在一个额外的时钟脉冲之后发生。所用命令如下：

```
pt_shell > set_multicycle_path 2 –from regA –to regB
```

在生成时钟的情况下，PT 不会自动确定原始时钟和导出时钟间的关系，即使使用了 create_generated_clock 命令。单周期的确定与一个时钟是否为生成无关。它是基于第一个时钟的开沿到第二个时钟命令（在这种情况下是生成时钟）的闭沿间的最小的间隔。

对具有不同频率的单个时钟，set_multicycle_path 命令可用于定义这些时钟间的关系。默认情况下，PT 对这些时钟间的建立时间和保持时间关系使用最严格的约束。使用定义这些时钟间准确关系的 set_multicycle_path 命令覆盖这些默认关系。

图 13-2 举例说明了存在于两个单个时钟间的关系，在单周期时序中（默认行为），建立和保持时间关系如图所示。然而，使用下面的命令指定 regA 和 regB 间的多周期路径：

```
pt_shell > set_multicycle_path 2 –setup –from regA/CP –to regB/D
```

上例使用乘数 2 定义两个时钟间建立时间关系，–setup 选项用于定义建立时间关系。然而，这个选项也会影响保持时间关系。PT 使用一组规则（在 PT 用户指南中有详细解释）来确定两个时钟间的保持时间关系的最严格约束。因此，PT 可能采用两个时钟间不正确的保持时间关系（如图 13-2 虚线部分所示）。为避免这一情况，也应定义两个时钟间的保持时间关系。由于通过 set_multicycle_path 命令定义的保持时间关系的规范是非常令人困惑的，因此不作为推荐的方法。建议设计者使用如下命令指定两个触发器间的保持时间关系：

```
pt_shell > set_min_delay 0 –from regA/CP –to regB/D
```

零值将保持时间关系从默认值（图 13-2 虚线处）移到所要的边沿（图 13-2 粗线处）。

13.2.2 虚假路径

一些设计可能含有虚假时序路径。不传播信号的时序路径被认为是虚假路径。虚假路径由如下 pt_shell 命令创建：

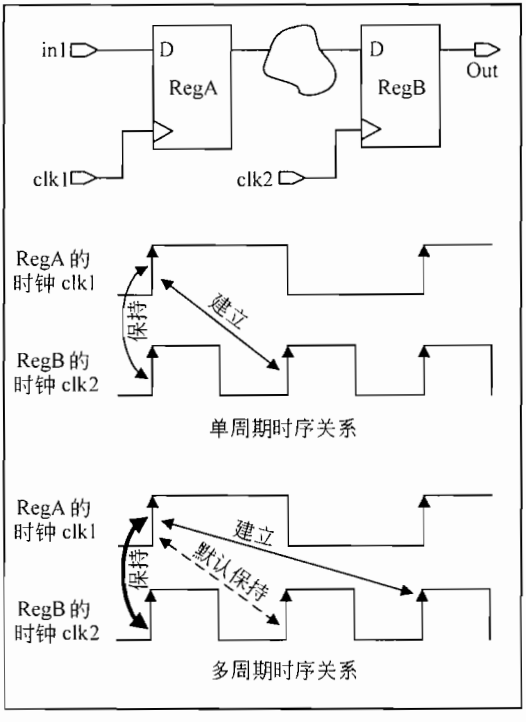


图 13-2　独立时钟间的关系定义

```
set_false_path –from   <from list> –to <to list>
               –through <through list>
```

必须注意的是，上述命令没有禁止任何单元的时序弧，它只移除确认的路径约束。因此，如果在虚假路径上进行时序分析，将会生成未约束的时序报告。

在默认情况下，PT 在所有路径上进行 STA，这导致所有路径段（包括设计中的虚假路径）时序报告的生成。如果虚假路径段以很大程度未通过时序分析，则报告将会掩盖真正的时序路径的违例。当然，这取决用于 report_timing 命令的选项。

假定设计中存在多个虚假路径，并且它们中的部分未能通过保持时间 STA。然而，真正的时序路径只有很少的一部分未能通过。虚假路径未被确认是因为用户认为–nworst 和–max_paths 选项所取的较大值会包含设计中所有的路径（包括真正的时序路径），因此，虚假路径的确认是不必要的。用户可用下述命令分析设计：

```
pt_shell > report_timing –from [all_inputs]                    \
                        –to [all_registers –data_pins]         \
                        –nworst 10000 –max_paths 10000         \
                        –delay_type min
```

上述方法当然是可行的，并且不会过度影响运行时间。然而–nworst 和–max_paths 选项所取的较大值（上例中使用的）使得 PT 生成/显示多个时序报告，包括设计中所有的路径，其中的大部分是虚假路径。只有少数选择的时序报告与真正的时序违例相关。使用这个方法区分真正的时序路径和虚假时序路径将会是令人厌烦的。另外，由于生成大量时序报告，容易错误地忽略违反时序约束的真正时序路径。为避免这种情况，建议在进行 STA 之前确认虚假路径。

此外，设计人员可使用–through 选项进一步分离虚假路径。必须注意的是，–through 选项会极大地影响运行时间，因此，使用时要谨慎，而且尽量少用。一个比较好的替代方法是用 set_disable_timing 命令（本章稍后介绍）禁止–through 列表中单元的时序弧。

13.2.2.1 设置虚假路径的有用提示

时序的异常会影响运行时间。在设计中设置多个虚假路径会造成 PT 更加缓慢地运行。设计人员不考虑没有采用正确方法指定的虚假路径，从而影响运行时间。下面提供的建议有助于设计者正确地定义虚假路径。

（1）当定义虚假路径时避免使用通配符，否则，可导致 PT 生成大量的虚假路径。例如：

```
pt_shell > set_false_path –from ififo_reg*/CP          \
                        –to ofifo_reg*/D
```

在上面的情况，如果 ififo_reg 和 ofifo_reg 是 16 位寄存器组的各个部分，PT 会生成大量不必要的虚假路径。禁止上面路径共享共同单元的时序弧是一个较好的方法。使用 13.3 节介绍的 set_disable_timing 命令可禁止时序弧。

（2）避免对多个虚假路径使用–through 选项，尽量找到一组确认的虚假路径所共享的共同单元，通过 set_disable_timing 命令禁止

这个单元的时序弧。

（3）不要为属于单独的异步时钟域的寄存器定义虚假路径。例如，如果两个异步时钟（如 CLK1 和 CLK2），则应避免使用如下命令：

```
pt_shell > set_false_path –from [all_registers -clock CLK1]        \
                          –to [all_registers -clock CLK2]
```

上面的命令迫使 PT 枚举设计中的每一个寄存器，从而对运行时间造成很大影响。一个更好的替代方法是在时钟自身上而不是在寄存器上设置虚假路径。这样做可阻止 PT 枚举设计中所有的寄存器，因而对运行时间只有很小影响或没有影响。这是在 PT 中定义两个时钟异步行为的首选和有效的方法。例如：

```
pt_shell > set_false_path   –from [get_clocks CLK1]          \
                            –to [get_clocks CLK2]

pt_shell > set_false_path   –from [get_clocks CLK2]          \
                            –to [get_clocks CLK1]
```

13.3　禁止时序弧

在设计中为了完成 STA，PT 自动禁止引起时序环的时序路径。然而，设计人员有时会发现，由于各种原因有必要禁止其他的时序路径，最普遍的是，PT 需要在所有时刻都选择正确的时序路径。可通过分别禁止单元的时序弧或通过对整个设计进行情况分析来禁止时序弧。

13.3.1　分别禁止时序弧

在 STA 期间，有时有必要禁止某个单元的时序弧以阻止 PT 在计算路径延迟时使用这个弧。由于 PT 使用产生最长延迟的时序弧来计算某一单元的延迟。这有时会产生不需要的虚假延迟值，因此需要禁止时序弧，在第 4 章中对此有详细的解释。

禁止个别单元时序弧的另一个原因是防止 PT 选择错误的时序路

径。图 13-3 说明了一种情况，其中控制输入（bist_mode）用于在多路选择器 MUXD1 的输入信号 bist_sig 和 func_sig 间选择。当 bist_mode 信号为低时，选择传播 bist_sig 信号；而当 bist_mode 信号为高时，允许 func_sig 信号通过。在正常模式（功能模式）下，阻塞 bist_sig 信号，而允许 func_sig 信号传播。然而，在测试模式下（如为测试 BIST 逻辑），选择 bist_sig 信号通过，而阻塞 func_sig 信号。这种多路选择器的应用在第 8 章作了详细描述（图 8-5），那里它用于旁路 RAM 的输入信号，从而使先前被 RAM 遮蔽的不可扫描的逻辑能够被扫描。

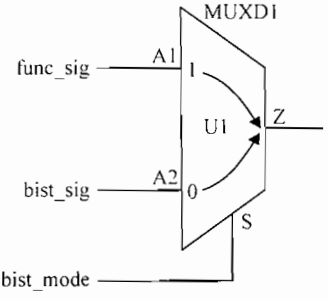


图 13-3 禁止时序弧

这个单元存在 3 条时序弧，分别是从 A1 到 Z、A2 到 Z 和从 S 到 Z。为了清晰，上图只标出了前两条弧。当进行 STA 检查功能模式下的时序时，除非用户使用 report_timing 命令的–through 选项分离出路径，否则 PT 可能选择错误的路径（由 A2 到 Z），从而生成虚假路径延迟时序报告。因此，在功能模式 STA 下，禁止单元 MUXD1（实例名 U1）的由 A2 到 Z 的时序弧是明智的。使用下述 pt_shell 命令执行：

```
pt_shell > set_disable_timing –from A2 –to Z {U1}
```

13.3.2 情况分析

上面情形的另一个解决方案是对设计进行情况分析。通过给 bist_mode 信号设置一个逻辑值，禁止/使能与 bist_mode 信号相关的所有时序弧。在上述情况，使用如下命令禁止由 A2 到 Z 的时序弧：

```
pt_shell > set_case_analysis 1 bist_mode
```

bist_mode 信号的逻辑值 1 迫使 PT 禁止由 A2 到 Z 的时序弧，并使 func_sig 信号能够传播。通过将这个值改变为 0，禁止由 A1 到 Z 的弧并允许 bist_sig 信号传播。

尽管 set_disable_timing 和 set_case_analysis 命令同样有禁止

时序弧的功能，但对许多这样情形的设计，情况分析方法较好。例如，一个命令用于分析在正常模式或测试模式下的整个设计。而当进行 STA 时，set_disable_timing 命令对禁止个别单元的时序弧有用。

13.4　环境与约束

除了细小的语法差别之外，PT 的环境和约束设置与用于 DC 的环境和约束设置相同。下面的命令举例说明了这些设置：

```
pt_shell > set_wire_load_model –name <wire-load model name>

pt_shell > set_wire_load_mode <top | enclosed | segmented>

pt_shell > set_operating_conditions <operating conditions name>

pt_shell > set_load 50 [all_outputs]

pt_shell > set_input_delay   10.0 –clock <clock name> [all_inputs]

pt_shell > set_output_delay 10.0 –clock <clock name> [all_outputs]
```

尽管 PT 为上述命令提供了很多选项，但绝大多数设计者只用到很少一些选项，如上所示。关于上述每个命令另外的可用选项的完整细节，建议用户参看 PT 用户指南。

由于这些命令的行为和功能与用于 DC 的命令相同，这里就不再解释。与上述每个命令相关的 DC 命令在第 6 章作了详细解释。

13.4.1　工作条件——困难的选择

一般来说，使用最坏情况工作条件来分析设计的建立时间违例，而最佳情况工作条件用于分析设计的保持时间违例。

使用最坏情况工作条件进行建立时间分析的原因是，库中每个单元的延迟值描述了在最坏情况条件下（最高温度、低电压和其他最坏情况工艺参数）工作的器件的延迟（通常为大）。大的延迟值使数据流变慢，从而引起某一触发器建立时间失败。

当设计为保持时间 STA 使用最佳工作条件时，对数据流会起到相反的作用。在这种情形下，工艺库中每个单元的延迟值（小）描述了最佳情况工作条件（最低温度、高电压和其他最佳情况工艺参数）。因此，现在数据流到达其目的地遇到了更少的延迟，也就是说数据比以前更快地到达，可能在寄存器的输入引起保持时间违例。

通过在工作条件边界情况下分析设计，可创建一个时间窗——如果器件在两个工作条件定义的范围内工作，器件会成功地工作。

13.5 布图前

在成功综合之后，必须对所得的网表进行静态分析以检查时序违例，时序违例可包含建立和/或保持时间违例。

综合设计的重点在于最大化建立时间，因此会遇到非常少的建立时间违例（如果存在）。然而，保持时间违例通常会在这一阶段发生，这是因为数据相对于时钟到达时序单元的输入太快。

如果设计没达到建立时间的要求，除了以违例路径进一步优化为目标重新综合设计外，别无其他选择。这可包括组合违例路径或过度约束具有违例的整个子模块。然而，如果设计没达到保持时间的要求，在布图前阶段修正这些违例，或推迟这一步直到布图后。很多设计人员对小的保持时间违例倾向于使用后一种方法，由于布图前综合和时序分析使用统计线载模型，并且在布图前阶段修正保持时间违例可导致同一路径布图后建立时间违例。然而，如果线载模型真实地反映了布线后延迟，在这一步修正保持时间违例是明智的。在任何情况下必须注意的是，明显的保持时间违例应在布图前阶段修正，以最小化布图后可导致的保持时间修正的数目。

13.5.1 布图前时钟规范

在布图前阶段，网表中缺少时钟树信息。因此，有必要在布图前阶段在前面估计布线后时钟树延迟，以进行适当的 STA。此外，也应定义估计的时钟转换以阻止 PT 为被驱动的门计算虚假延迟（通常较大）。大延迟的根源通常归结于与时钟网络相关联的高扇出。大扇出导致为驱动终点门的时钟计算出缓慢输入转换时间，进而导致 PT 为

终点门计算出不寻常的大延迟值。为了防止这一情形，建议在源头指定一个固定的时钟转换值。

下面的命令用于在设计的布图前阶段定义时钟：

```
pt_shell > create_clock –period 20 –waveform [list 0 10] [list CLK]

pt_shell > set_clock_latency 2.5 [get_clocks CLK]

pt_shell > set_clock_transition 0.2 [get_clocks CLK]

pt_shell > set_clock_uncertainty 1.2 –setup [get_clocks CLK]

pt_shell > set_clock_uncertainty 0.5 –hold [get_clocks CLK]
```

上述命令指定端口 CLK 为具有周期 20ns 的时钟类型，时钟延迟为 2.5ns，并且固定的时钟转换值为 0.2ns。时钟延迟 2.5ns 表示从输入端口 CLK 到所有终点的时钟延迟固定在 2.5ns。此外，时钟转换值 0.2ns 迫使 PT 使用 0.2ns 值而不是计算它。为建立时间指定的 1.2ns 和保持时间的 0.5ns 近似时钟扭斜。在布图前使用这个方法得出对布图后时钟网络结果的实际的近似。

13.5.2　时序分析

下面的脚本收集了上面提供的所有信息且用于对设计进行建立时间 STA。

布图前建立时间 STA 的 PT 脚本

```
#Define the design and read the netlist only
set active_design <design name>

read_db –netlist_only $active_design.db
#or use the following command to read the Verilog netlist.
# read_verilog $active_design.v
current_design $active_design

set_wire_load_model <wire-load model name>
set_wire_load_mode <top | enclosed | segmented>

set_operating_conditions <worst-case operating conditions>
```

```
#Assuming the 50pf load requirement for all outputs
set_load 50.0 [all_outputs]
# Assuming the clock name is CLK with a period of 30ns.
# The latency and transition are frozen to approximate the
# post-routed values.
create_clock –period 30 –waveform [0 15] CLK
set_clock_latency 3.0 [get_clocks CLK]
set_clock_transition 0.2 [get_clocks CLK]
set_clock_uncertainty 1.5 –setup [get_clocks CLK]
# The input and output delay constraint values are assumed
# to be derived from the design specifications.
set_input_delay 15.0 –clock CLK [all_inputs]
set_output_delay 10.0 –clock CLK [all_outputs]

# Assuming a Tcl variable TESTMODE has been defined.
# This variable is used to switch beween the normal-mode and
# the test-mode for static timing analysis. Case analysis for
# normal-mode is enabled when TESTMODE = 1, while
# case analysis for test-mode is enabled when TESTMODE = 0.
# The bist_mode signal is used from the example illustrated in
# Figure 13-3.

set TESTMODE [getenv TESTMODE]
if {$TESTMODE == 1} {
      set_case_analysis 1 [get_port bist_mode]
}else {
      set_case_analysis 0 [get_port bist_mode]
}

# The following command determines the overall health
# of the design.
report_constraint –all_violators

# Extensive analysis is performed using the following commands.
report_timing –to [all_registers –data_pins]
report_timing –to [all_outputs]
```

为 report_timing 命令的–from 和–to 选项指定的起点和终点也可用于指向选择的路径。此外，使用–through 选项可实现选择的路径的进一步分离。

在默认情况下，PT 进行最大延迟分析，因此为 report_timing 命令的–delay_type 选项指定 max 值是不需要的。然而，为了显示设计的所有时序路径，可利用–nworst 和/或–max_paths 选项。

正如前面章节提到的，report_constraint 命令用于确定设计的整体正确性。这一命令首先用来检查 DRC 违例（max_transition、max_capacitance 及 max_fanout 等）。此外，这个命令也可用于为整个设计生成广泛的建立/保持时间时序报告。注意由 report_constraint 命令生成的时序报告不包含完整路径的时序报告，它只为每个终点的所有违例路径生成一个总结报告（假定使用了 all_violators 选项）。

report_timing 命令用于更加详尽的分析设计，并生成包括从起点到终点完整路径的时序报告。这个命令对分析设计的失败路径段是有用的。例如，利用这个命令的–capacitance 和–net 选项能够缩小失败的范围。

13.6　布图后

布图后步骤包括用反标注的实际延迟分析设计时序，这些延迟由提取版图数据获得。分析是在含有时钟树信息的布线后网表上进行的。有多种方法将时钟树导入 DC 和 PT 中，这些已在第 9 章详细解释了。假定已有 db 格式的修改后的网表。

这一步应对设计进行全面的 STA，包括分析设计的建立和保持时间要求。通常，设计会以绰绰有余的建立时间通过时序，但可能达不到保持时间要求。为了修正保持时间违例，有几种方法可以使用，这些在第 9 章作了解释。在进行保持时间修正后，必须再次分析设计以验证修正的时序。

13.6.1　反标注什么

设计者最常问的问题之一是：我应给 PT 反标注什么，以何种格式反标注？

第 9 章讨论了版图数据库提取的各种类型和相关格式，详细讨论了每种格式的优缺点。建议将布图工具产生的下面的信息类型反标注

到 PT 以进行 STA：

（1）SDF 格式的连线 RC 延迟。

（2）set_load 格式的容性连线负载值。

（3）DSPF、RSPF 或 SPEF 文件格式的时钟及其他关键连线的寄生信息。

下面的 PT 命令用于反标注上面的信息：

- read_sdf：正如其名，该命令用来读取 SDF 文件。例如：

```
pt_shell > read_sdf rc_delays.sdf
```

- source：PT 用这个命令读取 Tcl 格式的外部文件。因此，这个命令可用于反标注 set_load 文件格式的连线电容文件。例如：

```
pt_shell > source capacitance.pt
```

- read_parasitics：PT 利用这个命令反标注 DSPF、RSPF 和 SPEF 格式的寄生参数，不必指定文件的格式，PT 会自动检测它。例如：

```
pt_shell > read_parasitics clock_info.spf
```

13.6.2 布图后时钟规范

除了这次要通过整个时钟网络传播时钟外，布图后时序分析与布图前的类似，都使用同样的命令。这是因为现在时钟网络包含时钟树缓冲器，从而时钟延迟和扭斜依赖于这些缓冲器。因此对布线后时钟规范不再需要修正时钟延迟和转换到一个指定值。下列命令举例说明了布线后时钟规范：

```
pt_shell > create_clock –period 20 –waveform [list 0 10] [list CLK]
pt_shell > set_propagated_clock [get_clocks CLK]
```

正如其名，set_propagated_clock 命令遍布时钟网络传播时钟。由于时钟树信息现在存在于设计中，时钟延迟、扭斜和转换时间由 PT 从组成时钟网络的门计算求得。

13.6.3　时序分析

设计的时序主要依赖于时钟延迟和扭斜，也就是说时钟是设计中所有其他信号的参考。因而在试图分析整个设计之前，进行时钟扭斜分析是明智的。Synopsys 通过其名为 SolvNET 的网上在线支持提供了有用的 Tcl 脚本。可下载这一脚本并在进行之前运行分析。如果没有 Tcl 脚本，设计人员可以编写他们自己的脚本来生成从时钟源点开始到所有终点终止的时钟延迟报告。时钟扭斜和总延迟可通过分析产生的报告确定。

虽然不需要为布图后 STA 设置时钟不确定性，一些设计人员喜欢指定少量的时钟不确定性以产生鲁棒的设计。

假定时钟延迟和扭斜是受限制的，下一步就是对设计进行静态时序分析以检查建立和保持时间违例。建立时间分析同布图前进行的类似，如前所述，差别仅在于时钟规范（传播时钟）。此外，在布线后 STA 期间，将从版图数据库提取的信息反标注到设计中。

下面的脚本描述了对设计进行布线后建立时间 STA 的过程，黑体条目反映了布图前和布图后时序分析间的差别。

布图后建立时间 STA 的 PT 脚本

```
# Define the design and read the netlist only
set active_design <design name>

read_db –netlist_only $active_design.db
# or use the following command to read the Verilog netlist.
# read_verilog $active_design.v
current_design $active_design

set_wire_load_model <wire-load model name>
set_wire_load_mode <top | enclosed | segmented>

#Use worst-case operating conditions for setup-time analysis
set_operating_conditions <worst-case operating conditions>

# Assuming the 50pf load requirement for all outputs
set_load 50.0 [all_outputs]
```

```
# Back annotate the worst-case (extracted) layout information.
source capacitance_wrst.pt # actual parasitic capacitances
read_sdf rc_delays_wrst.sdf # actual RC delays
read_parasitics clock_info_wrst.spf # clock network data

# Assuming the clock name is CLK with a period of 30ns.
# The latency and transition are frozen to approximate the
# post-routed values. A small value of clock uncertainty is
# used for the setup-time.
create_clock –period 30 –waveform [0 15] CLK
set_propagated_clock [get_clocks CLK]

set_clock_uncertainty 0.5 –setup [get_clocks CLK]
# The input and output delay constraint values are assumed
# to be derived from the design specifications.

set_input_delay 15.0 –clock CLK [all_inputs]
set_output_delay 10.0 –clock CLK [all_outputs]

# Assuming a Tcl variable TESTMODE has been defined.
# This variable is used to switch beween the normal-mode and
# the test-mode for static timing analysis. Case analysis for
# normal-mode is enabled when TESTMODE = 1, while
# case analysis for test-mode is enabled when TESTMODE = 0.
# The bist_mode signal is used from the example illustrated in
# Figure 13-3.

set TESTMODE [getenv TESTMODE]
if {$TESTMODE == 1} {
      set_case_analysis 1 [get_port bist_mode]
}else {
      set_case_analysis 0 [get_port bist_mode]
}
# The following command determines the overall health
# of the design.
report_constraint –all_violators

# Extensive analysis is performed using the following commands.
report_timing –to [all_registers –data_pins]
report_timing –to [all_outputs]
```

如上所述，使用最佳情况工作条件来分析设计的保持时间违例。下面的脚本总结了上面提供的所有信息且可用于对设计进行布线后保持时间 STA。黑体条目反映了建立时间和保持时间分析间的差别。

布图后保持时间 STA 的 PT 脚本

```
#Define the design and read the netlist only

set active_design <design name>

read_db –netlist_only $active_design.db
#or use the following command to read the Verilog netlist.
# read_verilog $active_design.v
current_design $active_design

set_wire_load_model <wire-load model name>
set_wire_load_mode <top | enclosed | segmented>

#Use best-case operating conditions for hold-time analysis
set_operating_conditions <best-case operating conditions>

# Assuming the 50pf load requirement for all outputs

set_load 50.0 [all_outputs]
# Back annotate the best-case (extracted) layout information.
source capacitance_best.pt # actual parasitic capacitances
read_sdf rc_delays_best.sdf  # actual RC delays
read_parasitics clock_info_best.spf # clock network data

# Assuming the clock name is CLK with a period of 30ns.
# The latency and transition are frozen to approximate the
# post-routed values.
create_clock –period 30 –waveform [0 15] CLK
set_propagated_clock [get_clocks CLK]
set_clock_uncertainty 0.2 –hold [get_clocks CLK]

# The input and output delay constraint values are assumed
# to be derived from the design specifications.
set_input_delay 15.0 –clock CLK [all_inputs]
set_output_delay 10.0 –clock CLK [all_outputs]
```

```
# Assuming a Tcl variable TESTMODE has been defined.
# This variable is used to switch beween the normal-mode and
# the test-mode for static timing analysis. Case analysis for
# normal-mode is enabled when TESTMODE = 1, while
# case analysis for test-mode is enabled when TESTMODE = 0.
# The bist_mode signal is used from the example illustrated in
# Figure 13-3.

set TESTMODE [getenv TESTMODE]
if {$TESTMODE == 1} {
      set_case_analysis 1 [get_port bist_mode]
}else {
      set_case_analysis 0 [get_port bist_mode]
}

# The following command determines the overall health
# of the design.
report_constraint –all_violators

# Extensive analysis is performed using the following commands.
report_timing –to [all_registers –data_pins]          \
                                    –delay_type min
report_timing –to [all_outputs]  –delay_type min
```

13.7 分析报告

下面几小节描述为进行布图前和布图后分析而由 report_timing 命令生成的时序报告。假定示例设计 *tap_controller* 的时钟 *tck* 的时钟周期为 30ns。

13.7.1 布局前建立时间分析报告

例 13.1 描述在布图前阶段生成的 STA 报告。用布图前时钟规范命令为时钟假定了理想设置。

下面命令用于指示 PT 为由输入端口 tdi 起始到触发器输入引脚结束的最差路径（最大延迟）所显示的一个时序报告:

```
pt_shell>report_timing –from tdi –to [all_registers –data_pins]
```

使用默认设置，也就是说没有指定–delay_type 选项，因此 PT 通过假定–delay_type 选项的 max 设置对设计进行建立时间分析。另外，PT 使用–nworst 和–max_paths 选项的默认值。这确保生成单个最差路径（最小松弛值）时序报告。由于所有其他开始于 tdi 输入端口并终止于其他触发器的路径具有一个较高的松弛值，因此此路径不显示。

例 13.1

```
****************************************
Report     : timing
             –path full
             –delay max
             –max_paths 1
Design     : tap_controller
Version    : 1998.08 –PT2
Date       : Tue Nov 17 11:16:18 1998
****************************************

Startpoint : tdi (input port clocked by tck)
Endpoint : ir_block/ir_reg0
             (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :max

Point                                    Incr       Path
-----------------------------------------------------------------------------
clock tck (rise edge)                    0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                    15.00      15.00  r
tdi (in)                                 0.00      15.00  r
pads/tdi (pads)                          0.00      15.00  r
pads/tdi_pad/Z(PAD1X)                    1.32      16.32  r
pads/tdi_signal (pads)                   0.00      16.32  r
ir_block/tdi (ir_block)                  0.00      16.32  r
ir_block/U1/Z (AND2D4)                   0.28      16.60  r
ir_block/U2/ZN (INV0D2)                  0.33      16.93  r
ir_block/U1234/Z (OR2D0)                 1.82      18.75  r
ir_block/U156/ZN (NOR3D2)                1.05      19.80  r
```

```
ir_block/ir_reg0/D (DFF1X)              0.00     19.80  r
data arrive time                                 19.80

clock tck (rise edge)                  30.00     30.00
clock network delay (ideal)             2.50     32.50
ir_block/ir_reg0/CP (DFF1X)                      32.50  r
library setup time                     -0.76     31.74
data required time                               31.74
--------------------------------------------------------------------------------
data required time                               31.74
data arrival time                               -19.80
--------------------------------------------------------------------------------
slack (MET)                                      11.94
```

由上面的报告可以清楚看出，设计以 11.94ns 松弛值满足所需的建立时间。这意味着在终点触发器建立时间违例之前至少有 11.94ns 的余量。

13.7.2 布图前保持时间分析报告

例 13.2 描述了在布图前阶段生成的 STA 报告。用布图前时钟规范命令为时钟假定了理想设置。

为了进行保持时间 STA，下面的命令用于指示 PT 显示存在于两个触发器间的最小延迟的路径时序报告。

```
pt_shell > report_timing -from [all_registers -clock_pins]   \
                         -to [all_registers -data_pins]      \
                         -delay_type min
```

在上述情况中，选项-delay_type 被指定为 min 值，从而告知 PT 显示最佳情况时序报告，保留所有其他选项的默认值。

例 13.2

```
****************************************
Report     : timing
             -path full
             -delay min
             -max_paths 1
Design     : tap_controller
Version    : 1998.08-PT2
```

```
Date       : Tue Nov 17 11:16:18 1998
*******************************

Startpoint : state_block/st_reg9
             (rising edge-triggered flip-flop clocked by tck)
Endpoint : state_block/bp_reg2
           (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :min

Point                                    Incr          Path
-------------------------------------------------------------------------------
clock tck (rise edge)                    0.00          0.00
clock network delay (ideal)              2.50          2.50
state_block/st_reg9/CP (DFF1X)           0.00          2.50   r
state_block/st_reg9/Q (DFF1X)            0.05          2.55   r

state_block/U15/Z (BUFF4X)               0.15          2.70   r
state_block/bp_reg2/D (DFF1X)            0.10          2.80   r
data arrival time                                      2.80

clock tck (rise edge)                    0.00          0.00
clock network delay (ideal)              2.50          2.50
state_block/bp_reg2/CP (DFF1X)                         2.50   r
library hold time                        0.50          3.00
data required time                                     3.00
-------------------------------------------------------------------------------
data required time                                     3.00
data arrival time                                     -2.80
-------------------------------------------------------------------------------
slack (VIOLATED)                                      -0.20
```

上述报告中的负松弛值意味着终点触发器的保持时间违例为 0.20ns。这是因为相对于时钟而言，数据到达得太快。

为了修正上述路径的保持时间，也应在同一路径上进行建立时间分析以获得总松弛余量。这样就提供了一个能在其中操作数据的时间范围。

对于上例，如果建立时间松弛值很大（比如说为 10ns），那么数据被延迟 0.20ns 或更多（如 1ns），从而在终点触发器处提供充足的

保持时间。然而如果建立时间松弛值比较小（如 0.50ns），那么只有非常窄的 0.30ns(0.50ns–0.20ns)的余量。准确地延迟数据 0.20ns 将生成所要求的结果，留下 0.30ns 作为建立时间。然而，微小的 0.30ns 的时间窗使得设计人员修正时序违例相当困难——刚好满足延迟数据而又不违例建立时间要求。在这种情况下，可能需要重新综合逻辑并以违例路径为目标进一步优化。

13.7.3 布图后建立时间分析报告

用于布图前建立时间 STA 的同样命令也进行布图后建立时间分析。然而，生成的报告略有不同，就是 PT 用星号来指示反标注的延迟。

例 13.3 描述了 PT 生成的布图后时序报告以进行建立时间 STA。以例 13.1（布图前建立时间 STA）中所示的同一路径段为目标说明布图前和布图后时序报告的差别。

例 13.3

```
****************************************
Report     : timing
             -path full
             -delay max
             -max_paths 1
Design     : tap_controller
Version    : 1998.08-PT2
Date       : Wed Nov 18 12:14:18 1998
****************************************

Startpoint : tdi (input port clocked by tck)
Endpoint : ir_block/ir_reg0
             (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :max

Point                                    Incr          Path
---------------------------------------------------------------------------
clock tck (rise edge)                    0.00          0.00
clock network delay (propagated)         0.00          0.00
```

input external delay	15.00	15.00	r
tdi (in)	0.00	15.00	r
pads/tdi (pads)	0.00	15.00	r
pads/tdi_pad/Z(PAD1X)	1.30	16.30	r
pads/tdi_signal (pads)	0.00	16.30	r
ir_block/tdi (ir_block)	0.00	16.30	r
ir_block/U1/Z (AND2D4)	0.22*	16.52	r
ir_block/U2/ZN (INV0D2)	0.24*	16.76	r
ir_block/U1234/Z (OR2D0)	0.56*	17.32	r
ir_block/U156/ZN (NOR3D2)	0.83*	18.15	r
ir_block/ir_reg0/D (DFF1X)	1.03*	19.18	r
data arrival time		19.18	
clock tck (rise edge)	30.00	30.00	
clock network delay (propagated)	2.00	32.00	
ir_block/ir_reg0/CP (DFF1X)		32.00	r
library setup time	–0.76	31.24	
data required time		31.24	
----------			
data required time		31.24	
data arrival time		–19.18	
----------			
slack (MET)		12.06	

经比较，布图后时序结果由松弛值 11.94（例 13.1 中）改善为 12.06。这一变化归结于在布图前 STA 阶段使用的线载模型和由版图实际提取的反标注数据间的差异。在这一情况下，同布线后结果相比，线载模型稍微悲观一点。

布图前和布图后结果间的另一个区别就是时钟的传播。在布图前时序报告中假定一个理想的时钟，然而，在布图后 STA 期间时钟是传播的，因而包含了真实的延迟。这在上面的报告中亦为“clock network delay (propagated)”。

在布图前阶段，假设理想时钟网络延迟为 2.5ns。布线后 STA 结果表明时钟实际上要比预先估计的快，也就是说时钟网络延迟值为 2.0ns 而不是 2.5ns。这提供了布线后时钟网络延迟值的指示。因此，下次（也许是下次迭代）在布线前阶段分析设计，2.0ns 时钟网络延迟值应用于提供对布线后结果更接近的近似。

13.7.4 布图后保持时间分析报告

用于布图前保持时间 STA 的同样命令也用于布图后保持时间分析。然而，生成的报告略有不同，就是 PT 使用星号指示反标注的延迟。另外，时钟网络延迟是传播的，而不是假定的理想延迟。

例 13.4 描述了 PT 生成的布图后时序报告以进行保持时间[①]STA。以例 13.2（布图前保持时间 STA）中所示的同一路径段为目标说明布图前和布图后时序报告的差别。

例 13.4

```
****************************************
Report     : timing
             -path full
             -delay min
             -max_paths 1
Design     : tap_controller
Version    : 1998.08-PT2
Date       : Tue Nov 17 11:16:18 1998
****************************************

Startpoint : state_block/st_reg9
             (rising edge-triggered flip-flop clocked by tck)
Endpoint : state_block/bp_reg2
             (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :min

Point                                    Incr          Path
-------------------------------------------------------------------------------
clock tck (rise edge)                    0.00          0.00
clock network delay (propagated)         1.92          1.92
state_block/st_reg9/CP (DFF1X)           0.00          1.92   r
state_block/st_reg9/Q (DFF1X)            0.18          2.10   r
state_block/U15/Z (BUFF4X)               0.04*         2.14   r
state_block/bp_reg2/D (DFF1X)            0.06*         2.20   r
data arrival time                                      2.20
```

① 原文有误为 setup-time—译者注。

```
clock tck (rise edge)                          0.00          0.00
clock network delay (propagated)               1.54          1.54
state_block/bp_reg2/CP (DFF1X)                               1.54   r
library hold time                              0.50          2.04
data required time                                           2.04
-------------------------------------------------------------------------------
data required time                                           2.04
data arrival time                                           -2.20
-------------------------------------------------------------------------------
slack (MET)                                                  0.16
```

在上例中，以 0.16ns 的余量满足了终点触发器保持时间。注意到时钟延迟在始点触发器（1.92ns）和终点触发器（1.54ns）间的差别。延迟的差异会引起时钟扭斜。通常一个小的时钟扭斜值是可以接受的，而一个大的时钟扭斜可在设计中引起竞争条件。竞争条件使得终点触发器锁存错误的数据。因此最好最小化时钟扭斜以避免这样的问题。

13.8　高级分析

本节为设计者提供了对设计进行高级 STA 的洞察力。依据情形，设计人员可利用以下几节中介绍的概念和方法详细分析设计。

13.8.1　详细的时序报告

设计中的一个路径常常不通过建立和/或保持时间，因此有必要仔细分析设计以找到问题的起因。

考虑例 13.2 中所示的时序报告，可以看出保持时间由于少了 0.20ns 而未通过。为找到问题的原因，使用如下的命令：

```
pt_shell > report_timing –from state_block/st_reg9/CP      \
                        –to state_block/bp_reg2/D          \
                        –delay_type min
                        –nets –capacitance –transition_time
```

在上面命令中，使用了额外的选项，即–nets、–capacitance 和

–transition_time。虽然上述命令同时用到了这 3 个选项，但这些选项也可独立使用。

除了用上述命令生成时序报告包含额外的扇出、负载电容和转换时间信息外，例 13.5 中所示的时序报告同例 13.2 中所示的也是相同的。

例 13.5

```
****************************************
Report     :  timing
           –path full
           –delay min
           –max_paths 1
Design     : tap_controller
Version    : 1998.08–PT2
Date       : Tue Nov 17 11:16:18 1998
****************************************
Startpoint : state_block/st_reg9
           (rising edge-triggered flip-flop clocked by tck)
Endpoint : state_block/bp_reg2
           (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :min

Point                                Fanout   Cap   Trans  Incr   Path
-------------------------------------------------------------------------
clock tck (rise edge)                                0.30   0.00   0.00
clock network delay (ideal)                                 2.50   2.50
state_block/st_reg9/CP (DFF1X)                       0.30   0.00   2.50  r
state_block/st_reg9/Q (DFF1X)                        0.12   0.05   2.55  r
state_block/n1234 (net)              2               0.04
state_block/U15/Z (BUFF4X)                           0.32   0.15   2.70  r
state_block/n2345 (net)              8       2.08

state_block/bp_reg2/D (DFF1X)                0.41           0.10   2.80  r
data arrival time                                                  2.80

clock tck (rise edge)                        0.30           0.00   0.00
clock network delay (ideal)                                 2.50   2.50
state_block/bp_reg2/CP (DFF1X)                                     2.50  r
library hold time                                           0.50   3.00
```

```
data required time                                                    3.00
-------------------------------------------------------------------------
data required time                                                    3.00
data arrival time                                                    -2.80
-------------------------------------------------------------------------
slack (VIOLATED)                                                     -0.20
```

通过分析例 13.5 中所示的时序报告，可看到单元 U15（BUFF4X）的扇出为 8，负载电容为 2.08pF。计算的单元延迟为 0.15ns。如前所述，相对于时钟延迟数据可修正保持时间违例。因此，如果单元 U15 的驱动强度由 4X 减少到 1X。由于转换时间的增加，它会引起单元 U15 的延迟值增加。延迟值的增加有助于减慢整个数据路径，从而消除保持时间违例。所得的时序报告如例 13.6 所示。

例 13.6

```
****************************************
Report    :  timing
          -path full
          -delay min
          -max_paths 1
Design    : tap_controller
Version   : 1998.08-PT2
Date      : Tue Nov 17 11:16:18 1998
****************************************

Startpoint : state_block/st_reg9
          (rising edge-triggered flip-flop clocked by tck)
Endpoint : state_block/bp_reg2
          (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :min

Point                                Fanout   Cap   Trans   Incr   Path
-------------------------------------------------------------------------
clock tck (rise edge)                                0.30   0.00   0.00
clock network delay (ideal)                                 2.50   2.50
state_block/st_reg9/CP (DFF1X)                       0.30   0.00   2.50 r
state_block/st_reg9/Q (DFF1X)                        0.12   0.05   2.55 r
state_block/n1234 (net)              2      0.04
state_block/U15/Z (BUFF4X)                           1.24   0.40   2.95 r
```

```
state_block/n2345 (net)                    8     2.08
state_block/bp_reg2/D (DFF1X)                          1.25   0.10   3.05 r
data arrive time                                                     3.05

clock tck (rise edge)                                  0.30   0.00   0.00
clock network delay (ideal)                                   2.50   2.50
state_block/bp_reg2/CP (DFF1X)                                       2.50 r
library hold time                                             0.50   3.00

data required time                                                   3.00
--------------------------------------------------------------------------
data required time                                                   3.00
data arrival time                                                   -3.05
--------------------------------------------------------------------------
slack (MET)                                                          0.05
```

在上面所示的时序报告中，通过将单元 U15 的驱动强度由 4X 减少到 1X，导致转换时间的增加，从而使门的增量延迟增加。这影响了整个数据路径，得到了正松弛余量 0.05ns，从而消除了终点触发器的保持时间违例。

13.8.2 单元交换

PT 允许交换设计中的单元，只要现有单元的引脚和替换单元的引脚完全相同。这个性能允许设计人员不用离开 pt_shell 就可进行假设和尝试。

例 13.6 中，单元 BUFF1X（具有较低的驱动强度和同 BUFF4X 有相同的引脚）取代了单元 BUFF4X，然而没有讨论取代的过程。完成此过程有两种方法，可在进行 STA 前手动修改网表；或在手动修改网表前，设计人员用 PT 的单元交换性能在违例路径段上进行 STA 假设和尝试。

在进行假设和尝试前手动修改网表当然是一个可行的方法，然而却是费力的。如例 13.6 中所示情况，首先终止 pt_shell，然后手动修改网表（用 BUFF1X 替换 BUFF4X），最后再次调用 pt_shell 以重新分析先前违例的路径段（从 st_reg9 到 bp_reg2）。如果对网表的修改没有生成所需的结果（路径段仍违反时序），那么需要重复整个过程。这种方法当然是冗长的和费时的。

首选的另一方法是用如下的命令以另一个单元替换现有的单元：

```
pt_shell > swap_cell {U15} [get_lib_cell stdcell_lib/BUFF1X]
```

在 pt_shell 中使用上述命令以 BUFF1X（来自“stdcell_lib”工艺库）替换现有的单元 BUFF4X（例化为 U15），并重新分析路径段以观察交换的效果。这提供了在不终止 pt_shell 的情况下进行调试设计并可视化单元交换效果的更快方法。

注意：单元交换只发生在 PT 内存中，物理网表保持不变。如果路径段和设计的其他部分通过了 STA，应通过手动修改网表以在网表中加入这一修改。

13.8.3　瓶颈分析

有时一个设计可包含多条共享一个共同的叶单元的路径段。如果这些路径段未通过时序，那么改变共同的叶单元的驱动强度（增大或减小尺寸）可移除所有路径段的时序违例。PT 提供识别在设计中多条违例路径段共享的共同的叶单元的能力，这称为瓶颈分析且通过使用命令 report_bottleneck 来进行。

例 13.2 中，从 state_block/st_reg9 开始到 state_block/bp_reg2 结束的路径段存在着保持时间违例。然而，从相同起始点（state_block/st_reg9）开始到不同终点 state_block/enc _reg0 结束的路径段也存在着保持时间违例（如例 13.7 所示）。

例 13.7

```
********************************
Report    : timing
            -path full
            -delay min
            -max_paths 1
Design    : tap_controller
Version   : 1998.08-PT2
Date      : Tue Nov 17 11:24:10 1998
********************************

Startpoint : state_block/st_reg9
             (rising edge-triggered flip-flop clocked by tck)
```

```
Endpoint : state_block/enc_reg0
           (rising edge-triggered flip-flop clocked by tck)
Path Group :tck
Path Type :min

Point                                    Incr       Path
-------------------------------------------------------------------------
clock tck (rise edge)                    0.00       0.00
clock network delay (ideal)              2.50       2.50
state_block/st_reg9/CP (DFF1X)           0.00       2.50  r
state_block/st_reg9/Q (DFF1X)            0.05       2.55  r
state_block/U15/Z (BUFF4X)               0.15       2.70  r
state_block/enc_reg0/D (DFF1X)           0.07       2.77  r
data arrival time                                   2.77

clock tck (rise edge)                    0.00       0.00
clock network delay (ideal)              2.50       2.50
state_block/enc_reg0/CP (DFF1X)                     2.50  r
library hold time                        0.50       3.00
data required time                                  3.00
-------------------------------------------------------------------------
data required time                                  3.00
data arrival time                                  -2.77
-------------------------------------------------------------------------
slack (VIOLATED)                                   -0.23
```

在上面所示的时序报告中，保持时间违例为 0.23ns。两个时序报告（例 13.2 和例 13.7）的目测显示一个单元 BUFF4X（例化为 U15）是两个路径段（st_reg9→bp_reg2 和 st_reg9→ enc_reg0）所共有的。因此，减少这一单元的驱动强度可消除两个路径段的保持时间违例。

然而，这一过程包括费力地仔细目测设计中所有路径段，以确定所有违例路径段的起点和终点间共有的叶单元。这种方法对于大量的路径段而言是极度冗长的。

推荐的确定所有违例路径段的起点和终点间共有叶单元的方法就是进行瓶颈分析。对上述情况（例 13.7 中），下面的命令用于确定违例路径段所共享的共同的叶单元：

```
pt_shell > report_bottleneck
```

例 13.8 显示了由 PT 生成的一个报告，确定了单元 U15（BUFF4X）

为上述两个路径段所共享的共同的叶单元。

例 13.8

```
****************************************************
Report          : bottleneck
                  -cost_type path_count
                  -max_cells 20
                  -nworst_paths 100
Design          : tap_controller
Version         : 1998.08-PT2
Date            : Tue Nov 17 12:09:09    1998
****************************************************

Bottleneck Cost = Number of violating paths through cell

Cell                    Reference               Bottleneck Cost
-------------------------------------------------------------------------------
U15                     BUFF4X                  2.00
```

一旦单元被确定了，它可同另一个单元相交换以修正多条路径段的时序违例。应对整个设计再一次进行完整的 STA。应手动地将任何所要求的改动（由于单元交换等）加入到最终的网表。

13.8.4　门控时钟检查

通常，低功耗设计包含仅当需要时由门控逻辑使能的时钟。对于这样的设计，应分析用于门控时钟单元的建立和保持时间违例，以避免时钟的制约。

可以通过第 12 章中所讲到的 set_clock_gating_check 命令来指定建立和保持时间要求。例如：

```
pt_shell > set_clock_gating_check –setup 0.5 –hold 0.02 tck
```

例 13.9 描述了门控时钟报告，它采用了上面为门控时钟“tck”指定的建立和保持时间要求。用如下命令生成报告：

```
pt_shell > report_constraint –clock_gating_setup      \
                             –clock_gating_hold       \
                             –all_violators
```

例 13.9

```
****************************************************
Report          : constraint
                  –all_violators

                  –path slack_only
                  –clock_gating_setup
                  –clock_gating_hold
Design          : tap_controller
Version         : 1998.08–PT2
Date            : Tue Nov 17 12:30:07    1998
****************************************************

clock_gating_setup

Endpoint                                        Slack
-------------------------------------------------------------------------------
state_block/U1789/A1                            –1.02 (VIOLATED)
state_block/U1346/A1                            –0.98 (VIOLATED)

clock_gating_hold
Endpoint                                        Slack
-------------------------------------------------------------------------------
state_block/U1789/A1                            –0.10 (VIOLATED)
state_block/U1450/A1                            –0.02 (VIOLATED)
```

重要的是，注意除了选项–clock_gating_setup 和–clock_gating_hold 以外，应使用选项–all_violators。没有包括–all_violators 选项会导致报告只显示失败的代价函数而不确定失败的门。

出于调试违例的原因以及如何纠正它的目的，也可包括–verbose 选项以显示完整的路径报告。例 13.10 描述了一份这样的报告，它由如下命令生成：

```
pt_shell > report_constraint      –clock_gating_hold      \
                                  –all_violators          \
                                  –verbose
```

例 13.10

```
****************************************************
Report          : constraint
```

```
                -all_violators
                -path slack_only
                -clock_gating_hold
Design          : tap_controller
Version         : 1998.08-PT2
Date            : Tue Nov 17 12:32:10    1998
*******************************************************
Startpoint : state_block/tst_reg11
          (rising edge-triggered flip-flop clocked by tck)
Endpoint : state_block/U1789
          (rising clock gating-check end-point clocked by tck)
Path Group :**clock_gating_default**
Path Type :min

Point                                    Incr       Path
-------------------------------------------------------------------------------
clock tck (rise edge)                    0.00       0.00
clock network delay (propagated)         2.25       2.25
state_block/tst_reg11/CP (DFF1X)         0.00       2.25   r
state_block/tst_reg11/Q (DFF1X)          0.05*      2.30   r
state_block/U1789/A2 (AND4X)             0.12*      2.42   r
data arrival time                                   2.42

clock tck (rise edge)                    0.00       0.00
clock network delay (propagated)         2.50       2.50
state_block/U1789/A2 (AND4X)                        2.50   r
clock gating hold time                   0.02       2.52
data required time                                  2.52
-------------------------------------------------------------------------------
data required time                                  2.52
data arrival time                                  -2.42
-------------------------------------------------------------------------------
slack (VIOLATED)                                   -0.10
```

上例中，门 AND4X[①]用于门控时钟“tck”。这个单元的引脚 A2 和使能信号相连，而时钟驱动这个单元的引脚 A1。根据报告可知，门控逻辑违反了保持时间。为了修正保持时间违例，可减小单元 AND4X[②]的尺寸以减慢数据通道。

① 例中为 AND4X，原文中误为 AND0X。——译者注。
② 同注①

13.9 小结

以能工作的芯片作为最终产品，静态时序是其成功的关键。静态时序不仅验证设计时序，也检查设计中所有的路径段。本章包括了通过静态时序分析全面分析设计的所有必要步骤。

本章是由作为时序验证工具的静态时序分析和动态仿真方法比较开始的。建议以前一种方法作为动态仿真方法的替代方法。接下来是时序例外的详细讨论，包括多周期和虚假路径，提供了指导用户选择最佳方法的有用提示。

本章用单独的一节讨论禁用单元时序弧并进行了情况分析。对于包含许多单元并且时序弧与一共同信号相关的设计，推荐情况分析而不是时序弧的单个禁用。提供了一个 DFT 逻辑情况示例作为情况分析的应用。

另外，本章还详细介绍了布图前和布图后分析设计的过程，其中包括时钟规范和时序分析。

13.8 节为时序报告及时序报告的高级分析，它的内容十分广泛。在每一步分析中都提供了报告示例并详加解释。

附录 A

使用 Physical Compiler 的一个新的时序闭合方法

A New Timing Closure Methodology using Physical Compiler

Himanshu Bhatnagar
Conexant 系统公司
Boston ,SNUG 2001

摘要

商业成功越来越倚重于开发团队交付具有最短的“上市时间”且满足客户要求的产品的能力。为减缓“上市时间”的压力，在过去的几年，工具和技术都有所发展。由于设计复杂度的增加以及对更高速度的需求，所以时序收敛成为设计团队所面临的最大挑战之一。由于这样的原因，因此将“时序闭合”提升到设计周期的最前端。

由于设计复杂性的增加，所以测试也成为一个主要问题。现在设计人员在进行编码设计时不仅要准确地实现设计所需要的功能，还要时时考虑“测试”的需要。

本论文探索了使用 Physical Compiler 的新技术，PhyC 可用于取得早期的时序闭合。本论文还介绍了使用 Physical Compiler 的可测性设计（或 DFT）扫描插入和重排。

引言

时序闭合是目前推动 EDA 厂商的最大动力之一。为了时序收敛，要进行综合的设计陷入综合与布图的无限循环中。前端和后端工具的分离更加剧了这一情况。由前端和后端间的这个人为的障碍加剧了的分离，使得设计工程师和版图工程师“实际上”讲两种不同的语言。当把网表提交给版图工程师时，瓶颈就出现了。负责布图规划、布局布线的版图工程师可能对设计的复杂性没有概念。更糟的是，每个工具（综合和布图）的不同延迟计算器给出不同的结果，增加了时序收敛问题。

为减轻这一问题，Synopsys 引入了 Physical Compiler（或 PhyC）。这个工具位于前端综合和后端布图工具之间。其想法是，使用相同的时序约束、库等并进行设计的单元布局。这个工具用 Steiner 路径作为计算单元延迟的基础，而不是依靠线载模型。这个方法提供了更为准确的延迟计算。

为取得早期时序闭合，本论文解释了使用 PhyC（2000.11sp1 版）的新方法。此外，本论文也详述了扫描插入技术以及 PhyC 的一些缺陷。

设计流程概述

为了缩减从 RTL 编码到最终定案下单的设计周期，我们分析设计周期中的每一步，然后找出缩减每一步的方法。

以下为设计周期的主要步骤：

RTL 编码

RTL 验证

带扫描插入的单通综合

布图规划

布局

时钟树插入

布线

静态时序分析

设计周期中 RTL 编码和仿真这一步需要很强的耐心、人工干预和大量的验证。采用高级技术（如形式验证）运行在具有大量内存的、更快速机器上的最先进的仿真器当然可缩短周期，然而，设计编码和验证过程不是自动的，其他几步可以实现自动化以缩短上市时间。

过去，Synopsys 和其他用户都已制定了各种方针以使综合过程自动化。这些方法提供了设计综合和时序分析的广泛的解决方案。这些方法尽管不够完善，但由于它们采用了线载模型，因此仍能提供一个充分优化的网表。

DFT 扫描插入是另一个需要人工干预的主要瓶颈。设计人员不仅必须完成功能正确的编码，而且在对设计进行编码的过程中也必须遵守 DFT 规则。过去扫描插入是在已综合的网表上进行的。然而，由于不断增加的设计复杂度和时序收敛方面的问题，现在设计在 RTL 源本身就已经考虑要满足 DFT 规则。

将综合后设计交付给版图工程师就进入后端设计的布图规划阶段。通常这一阶段遇到的问题最多，主要是因为版图工程师对设计缺乏了解。如果布图规划不能提供一个好的起点，那么由于拥塞会引起布线问题，因而对设计的时序将产生严重的影响。

取得早期时序闭合的关键

为了缩短时间，我们确定了能够极大地减少设计周期时间的两个领域：

（1）使 RTL 在设计周期中较早地符合 DFT 规则；

（2）进行时序驱动的布图规划。

使设计符合 DFT 规则（2.0 节中约减设计流程中的步骤 3）

最近，Synopsys 引入了一个非常需要的命令，叫作“`rtldrc`”。这个命令在 RTL 源码上运行并确定可能违反 DFT 规则的有问题的测试范围。这个命令的使用极大地减轻了设计工程师掌握所有 DFT 规则的负担。有了这个命令，设计人员可只对设计的功能进行编码，然后通过 rtldrc 运行它。如果违反 DFT 规则，则在进行下一步之前修改设计。这样做，完全消除了后面可检测到的任何意外情况（它们可以

要求改变 RTL 并重新综合）。使用这一方法的优点是，通过减少相关的反复，节省了大量的时间。换言之，不需要修改综合后的网表以使其符合 DFT 规则。

下面是运行 rtldrc 的一个示例：

```
dc_shell -t > analyze –f verilog my_design.v
dc_shell -t > elaborate my_design
dc_shell -t > create_test_clock –p 100 –w {45 55} my_clk
dc_shell -t > set_test_hold 1 my_test_mode
dc_shell -t > set_signal_type test_asynch_inverted my_reset
dc_shell -t > rtldrc
```

在设计通过 rtlrc 检查后，可以继续进行完整的 one-pass 综合并且生成完全优化的网表，它是可能进行扫描的。

时序驱动布图规划（2.0 节中约减设计流程中的步骤 4）

传统的布图规划方法包括用宏单元（如 RAM/ROM 等）布局定义芯片面积以及手动布电源和地线。版图工程师基于如下方法创建布图规划：

1. 布图工具中使用飞线的连通性。
2. 设计人员对模块/宏单元局的建议。

这两种方法可能不会产生最佳的时序结果。设计者对逻辑模块连通性的看法与布图工具所“见”的可能相差非常大。如果布图规划不是最佳的，那么只能在布线后的静态时序分析期间才能认识到设计的时序的品质较差。换言之，在能进行静态时序分析之前，必须对设计进行布局、时钟树插入和布线。如果时序分析未通过，则整个过程要重新开始。这个方法浪费了宝贵的时间。

意识到这一点，我们建立了一个更好的解决方案，它利用了 PhyC 不仅能布局标准单元也能布局宏单元的能力。我们用如下的流程执行这一步：

1. 定义芯片面积/宽长比并布置 IO 压焊块。
2. 修改宏单元的 LEF 文件，因而它们“看起来”同标准单元相同。
3. 使用 lef2pdb 转换应用程序，将 LEF 转换为 pdb（逻辑 db 的物理等价物）。
4. 运行 physopt 并写出 PDEF。

5. 读取 PDEF 到布图工具中。

6. 做一个适当的布图规划（电源线、组合宏单元、添加阻挡物等）。

7. 写出新的 PDEF。

8. 使用新的 PDEF 运行 physopt 并只放置标准单元（使用宏单元的原先的 pdb）。

如下所示，为了进行第 2 步，只需要修改宏单元的 LEF 文件中的一行：

原先的 LEF	修改后的 LEF
CLASS RING	CLASS CORE

使用上述流程的优点是，用 PhyC 的时序驱动布局能力放置标准单元和宏单元。一旦放置好所有的单元，在仍保持其相对位置关系的同时，为了移除较小的重叠（参见下面的注意），少量地移动宏单元。然后，在写出最终布图规划后的 PDEF 前，继续添加围绕宏单元的电源/地环以及阻拦物和电源线。换言之，在时序驱动放置所有单元后，美化版图外观。

使用上述流程，不仅能在一次迭代中收敛时序，而且还大大缩短布图规划芯片所需的时间。

注意：当放置宏单元时，PhyC 会抱怨它不能最优地放置多行高的单元。它会放置它们但布局可能不是最优的。我们会看见一些重叠，然而宏单元的相对位置是不错的。Synopsys 告知我们 PhyC 的这个能力将在即将发布的 2001.08 版中可用。

扫描链排序

扫描链排序的优点很多，然而也难免有缺点。

以下是扫描链重排的一些优点：

1. 减少拥塞，从而改善时序。

2. 较少的总面积（极大地减少了连线长度）。

3. 由于减小了触发器负载，因此改善了功能路径的建立时间。

4. 减少了负保持时间（主要的仿真与静态时序分析问题）。

5. 较少的总电容改善了时序。

6. 由于驱动较少的连线电容，因此改善了功耗。

7. 较好的时钟树（较低的延迟和较少的缓冲器），从而改善了时序以及低功耗。

缺点是：

1. 增加了扫描路径保持时间违例的几率。

2. 在设计周期中，增加了额外的运行时间。

使用时钟扭斜来减少保持时间违例

由于优点远大于使用时钟扭斜来减少保持时间违例的缺点，因此扫描链排序是必须的。然而，我们对增加的保持时间违例能做些什么？一个选择就是插入一个完全平衡的零扭斜时钟树。通常 ASIC 界喜欢使用这种方法并试图找到最小扭斜的时钟树。希望触发器的 Clock-to-Q 延迟和额外的连线 RC 延迟之和会超过时钟扭斜，从而防止保持时间违例。总的说来这个方法工作得最好，并且消除了大多数的保持时间问题。在插入零扭斜时钟树之后，剩余的保持时间问题区域可以通过在扫描路径中添加延迟单元逐一解决。

最小化保持时间违例的另一选择是完全忽略时钟扭斜。这里的座右铭是**“扭斜就是好”**。对此有两个理由。首先，扭斜实际上可防止保持时间问题；其次，扭斜减少了由所有同时翻转的时钟缓冲器引起的功率尖峰。当然，这依赖于设计速度以及设计所能容忍的扭斜的多少。过多的扭斜可导致建立时间的问题。

布局工具的工作方式是将触发器通常聚集在同心圆上（图 A1）。当插入时钟树时，时钟缓冲器被放置到每个圆当中。对于簇中的触发器，这个安排提供了零时钟扭斜。零时钟扭斜对簇中的触发器意味着这些触发器不会有任何保持时间违例。换言之，触发器 Clock-to-Q 延迟本身会防止任何保持时间违例的可能。

当数据信号越过簇边界时，保持时间违例的可能性会增加。单独的时钟缓冲器驱动另一个簇并可导致第一个与第二个簇之间的时钟扭斜。在这种情况下，两个触发器会受到影响。扫描链中的最后一个触发器属于第一簇，扫描链中的第一个触发器属于第二簇。这里，扭斜可以是正的也可以是负的。

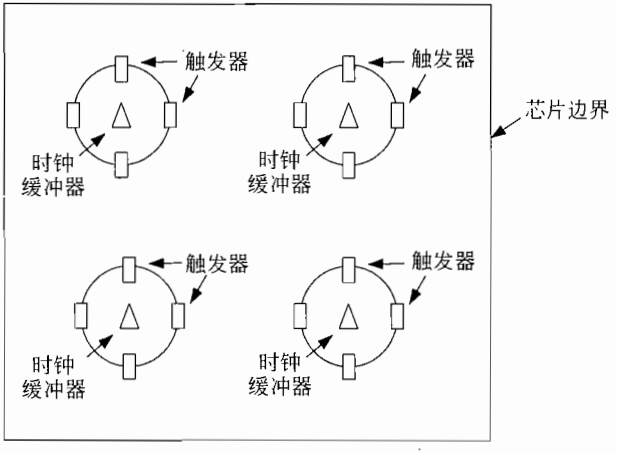


图 A1　触发器簇

图 A2 描述了两个寄存器的情况，它们的时钟在不同的时刻到达。如果 RegB 处的时钟信号的到达时间比 RegA 处时钟信号的到达时间早（负扭斜），则对 RegB 不会有任何保持时间问题；然而如果 RegB 处的时钟信号比 RegA 处的时钟的到达时间晚（正扭斜），可能会发生保持时间违例。

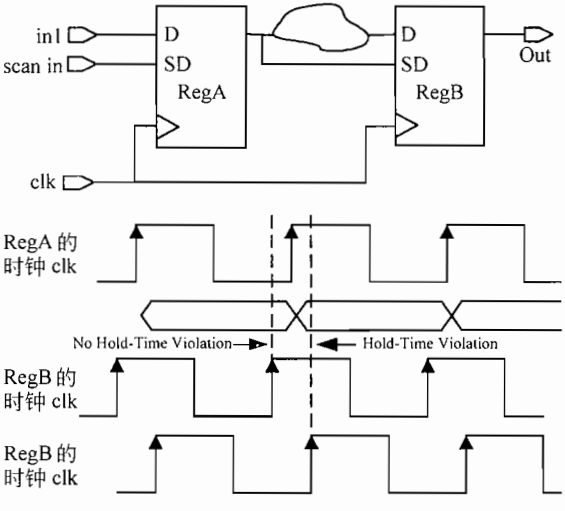


图 A2　正的与负的时钟扭斜

扭斜是无偏的，换言之，它既可以是正的也可以是负的。因而，对于在簇边界上的触发器，50%没有保持时间违例，而其余 50%可能

会有保持时间违例。现在考虑从一个簇到下一个簇的数据通道的延迟。如果这些簇是分散的，那么 RC 延迟本身可能要比簇间的时钟扭斜更大。由于这种布局安排本身就能防止保持时间问题，因此只有很少的触发器容易受到保持时间问题的影响。通过将延迟单元插入它们的扫描路径中，可以逐一修正保持时间违例。

注意：如果你想获得最小扭斜的时钟树，那么平衡上升/下降时间的时钟缓冲器和反相器可极大地减小扭斜。如果你已经门控了时钟树，则可确信门控逻辑也具有平衡的上升/下降时间。

insert_scan 和保持时间

insert_scan 的工作方式是：它尽力找到最佳的扫描链路径以最小化保持时间违例。它考察由触发器驱动的逻辑，如果它找到任何门，如缓冲器或反相器，它将扫描链连接到这个缓冲器的输出，而不是将输出 Q 直接连到下一个触发器的扫描输入端口。这对最小化保持时间违例有极大的帮助。下面的开关控制这一行为：

```
dc_shell -t > set test_disable_find_best_scan_out false
```

通过将上面的变量设置为“false”，insert_scan 会尽力找到最佳的扫描输出。将这个变量设置为“true”会禁止这一行为，它会抽出触发器的输出 Q 并将其连到下一个触发器的扫描输入。这个变量的默认值为“false”。

Physical Compiler 问题

在进行了扫描链排序或使用了隐变量——set physopt_fix_multiple_port_nets true 之后，当使用 read_verilog 读取网表时，PhyC 在最终的 verilog 网表中输出大量的 *assign* 语句。当读取 db 文件并进行相同操作时（包括使用上述变量），不会生成 *assign* 语句。如果 verilog 网表编译成 db 格式，则 PhyC 仍将产生 *assign* 语句。不会生成 *assign* 语句的唯一方法是：当写出 db 时，DFT compiler 已完成扫描插入操作。

Synopsys 极力推广的“集成 physopt 流程”不“像广告”那样地工作。其想法是首先使用单通扫描综合编译设计，然后运行带扫描排

序选项的 physopt。Physopt 应进行扫描连接、排序及布局。然而无论我们做什么，PhyC 都抱怨设计没有准备好扫描。运行 check_test 没有错误，但 physopt 一直抱怨设计没有准备好测试。这个问题被 Syropsys AE（应用工程师）解决（有了 solv-net 文章），他告诉我们必须设置一个属性告知 PhyC 设计准备好测试。设置这一属性之后，就可以成功地运行 physopt 了。

致谢

首先我要感谢 Eric Tan（Conexant）应我的要求做了大量的布图工作。

同时向 Synopsys 公司的 Elisabeth Moseley、Bob Moussavi 和 Kelly Conklin 表示感谢，对我提出的问题，他们极其耐心地作出解答。没有他们的帮助，我不可能完成这篇论文。感谢 Leah Clark 帮助润色本论文。

参考文献

1. Physical Compiler 用户指南
2. DFT Compiler 用户指南

附录 B

Makefile 实例

```
#=============================================
# General Macros
#=============================================
      MV = \mv –f
      RM = \rm –f
      DC_SHELL = dc_shell

      CLEANUP = $(RM) command.log pt_shell_command.log
#=============================================
# Modules for synthesis
#=============================================
top:
          @make $(SYNDB) /top.db
$(SYNDB) /top.db:          $(SCRIPT)/top.scr      \
                           $(SRC)/top.v           \
                           $(SYNDB)/A.db          \
                           $(SYNDB)/B.db
$(DC_SHELL) –f    $(SCRIPT)/top.scr | tee top.log
$(MV) top.log     $(LOG)
$(MV) top.sv      $(NETLIST)
$(MV) top.db      $(SYNDB)
$(CLEANUP)

A:
          @ make $(SYNDB) /A.db
$(SYNDB) /A.db:            $(SCRIPT)/A.scr        \
                           $(SRC)/A.v             \
                           $(SYNDB)/A.db
```

```
$(DC_SHELL) –f $(SCRIPT)/A.scr | tee A.log
$(MV) A.log     $(LOG)
$(MV) A.sv      $(NETLIST)
$(MV) A.db      $(SYNDB)
$(CLEANUP)

B:
        @ make $(SYNDB) /B.db
$(SYNDB) /B.db:         $(SCRIPT)/B.scr         \
                        $(SRC)/B.v              \
                        $(SYNDB)/B.db

$(DC_SHELL) –f  $(SCRIPT)/B.scr | tee B.log
$(MV) B.log     $(LOG)
$(MV) B.sv      $(NETLIST)
$(MV) B.db      $(SYNDB)
$(CLEANUP)
```