网站支持

EDA 先锋工作室是与人民邮电出版社紧密合作的一支电子设计领域专业书籍创作队伍。该工作室的成员都是电子、通信、半导体行业的资深研发人员。

为了配合本书的学习，EDA 先锋工作室在“EDA 专业论坛”（http://www.edacn.net）上开办了《设计与验证——Verilog HDL》的讨论园地。利用该园地，作者联合业界专业人士长期在论坛上为读者答疑解惑，讨论 EDA 工程经验与设计技巧，并对书中所述问题加以引申，借此与读者共同切磋，互相提高。网站提供了本书光盘中所附资料的下载服务，并介绍了 EDA 先锋工作室所编图书的出版动态。

EDA 先锋工作室非常重视您的批评和建议，您可以通过电子函件以及网站反馈您的信息。

电子信箱 altera_book@edacn.net。

EDA 先锋工作室

主　编：王　诚

副主编：薛小刚　钟信潮

编　委：李　楠　吴继华　庞　健　由武军　袁　园
周海涛　侯小辉　寿开宇　范丽珍　薛　宁
路　远　梁晓明　伊贵业　吴义涛　张世卓
张伟平　王书松　吴　蕾　胡安琪　吴卫旋
董振东　于春华

关 于 本 书

内容和特点

Verilog HDL 作为两大硬件描述语言之一，拥有很大的用户群。据调查，目前美国有 90%左右的 IC 设计人员使用 Verilog，在中国，大概有 50%左右的人在使用 Verilog。当前数字芯片设计行业正处于强劲上升时期，风头盖过了几年前的软件设计业，已经成为电子和 IT 类的高薪行业。大量高校毕业生和部分软件设计人员正在不断涌入这个领域。要想尽快在 IC 设计领域站稳脚跟，就必须要尽快掌握 HDL 语言的设计方法。

现在市场上关于 Verilog 的书籍大多数是介绍语法和建模的，没有真正体现出理论性与实用性的结合。针对这种情况，本工作室创作了本书。

全书共分 9 章，各章内容简要介绍如下。

- 第 1 章：介绍 HDL 的设计方法，Verilog 与 VHDL、C 等语言的区别，以及 HDL 语言的设计与验证流程。
- 第 2 章：介绍 Verilog 的语言基础。
- 第 3 章：重点介绍 Verilog 的 3 种描述方法和不同的设计层次。
- 第 4 章：介绍 RTL 建模的概念和一些常用电路的 Verilog 设计方法，最后引申出 Verilog 语言的可综合子集。
- 第 5 章：总结了常用的 RTL 同步设计原则，逐一介绍了设计模块的划分、设计组合逻辑和时序逻辑时应该注意的问题，以及优化 RTL 代码的方法等内容。
- 第 6 章：介绍状态机的设计方法和技巧。
- 第 7 章：介绍如何搭建测试平台，对设计进行验证。
- 第 8 章：详细描述了 Verilog 语言的语义和仿真原理，是 Verilog 语言的精髓所在。
- 第 9 章：总结并展望 HDL 和 HVL 的发展趋势。

读者对象

本书可作为高等院校通信工程、电子工程、计算机、微电子和半导体等相关专业的教材，也可作为硬件工程师和 IC 工程师的参考书。

光盘使用说明

本书配套光盘中提供了书中示例的工程文件、设计源文件和说明文件，示例按照章节编号和出现的先后顺序排列，例如“Example-2-1”表示第 2 章中的第 1 个示例。

工程示例文件夹中包含该工程的项目文件、源文件、报告文件和生成结果等文件。

对于一些相对复杂的示例，说明文件中给出了示例的详细信息和操作指南，而对于一些简单的实例，则只给出了源代码。

另外，为了配合读者进一步学习，光盘中还提供了 Verilog 1995 和 Verilog 2001 这两个版本的 IEEE 标准文献，读者可以从中查阅 Verilog 的语法细节。

本书约定

为了方便读者阅读，本书还设计了 4 个小图标，这些图标的含义如下。

 行家指点：用于介绍使用经验和心得，或罗列一些重要的概念。

 注意事项：用于提醒读者应该注意的问题。

 多学一招：用于介绍实现同一功能的不同方法。

 操作实例：用于引出一个操作题目和相应的一组操作步骤。

本书第 1、4、5、6 章由王诚编写，第 2、3、7、8、9 章由吴继华编写。由于作者水平有限，书中难免会有疏漏，敬请读者批评指正。

感谢您选择了本书，也请您把对本书的意见和建议告诉我们。

EDA 先锋工作室网站 http://www.EDACN.net。

EDA 先锋工作室

2006 年 7 月

目 录

第1章 HDL设计方法简介

本章重点介绍数字系统的建模和 HDL 语言的基本概念，并引入了主流的设计和验证流程。

本章主要内容如下：

- 设计方法的变迁；
- Verilog 语言的特点；
- HDL 的设计与验证流程。

1.1 设计方法的变迁

随着微电子设计技术的发展，数字集成电路已经从电子管、晶体管、中小规模集成电路、超大规模集成电路（VLSIC）逐步发展到今天的专用集成电路（ASIC）。人们在工作和生活中用到的一些产品，如计算机、手机、数字电视等都运用了复杂的专用数字集成电路，而数字逻辑器件也从简单的逻辑门发展到了复杂的 SOC（System On Chip，片上系统），提供了对复杂系统的灵活支撑。

随着数字电路系统的不断发展，系统的逻辑复杂度与规模日益增加，数字系统的设计方法也随之不断演进。在早期简单的门逻辑设计阶段，电子辅助设计（EDA）工具的应用范围十分有限，工程师们习惯于使用卡诺图简化设计，然后通过面包板等实验系统验证设计；在系统相对复杂以后，工程师们又开始借助 EDA 工具通过原理图描述数字系统，原理图由元件库中的元件构成，使用 EDA 工具可以对原理图进行仿真并分析其性能；当数字系统发展到 ASIC 与可编程逻辑器件（PLD）设计阶段后，原理图不利于移植，维护起来费时费力等缺点逐步显现，这时一种抽象度更高、运用起来更灵活的设计方式——硬件描述语言（HDL， Hardware Description Language）应运而生。

使用 HDL 语言可以从算法、系统级（System Level）、功能模块级（Function Model Level）、行为级（Behavior Level）、寄存器传输级（RTL，Register Transfer Level）、门级（Gate Level）和开关级（Switch Level）等不同层次描述数字电路系统，然后通过 EDA 工具综合、仿真并实现该系统。可以说 HDL 语言的出现是数字系统设计方法的一个重大飞跃。

由于 EDA 工具的不断推陈出新，又引发了数字电路系统设计方法的另一个重大飞跃。当数字系统发展到 ASIC 和 PLD 设计阶段之后，人们需要直接描述 CMOS 的开关电路或门级电路。这种电路设计量庞大，仿真速度也非常慢，如果用开关级或门级方法描述当今系统门数量为千万门级的 FPGA（Field Programmable Gate Array，现场可编程门阵列），结果是不可想象的。这时人们就希望能够使用 HDL 语言直接从更高的层次描述电路，然后使用

EDA 工具自动将高层次的 HDL 电路描述解析到门级，从而大大缩短了设计与仿真的时间。这种通过 EDA 工具将高层次的电路描述解析到门级等低层次的电路描述的过程就叫做“综合”（Synthesize），或者称为逻辑综合。综合工具能将高层次的 HDL 语言、原理图等设计描述翻译成由与、或、非门等基本逻辑单元组成的门级连接（网表），并根据设计目标与要求（约束条件）优化所生成的逻辑连接，输出门级网表文件。目前最成熟的综合工具是 RTL 级综合工具，它能将 RTL 级描述翻译并优化为门级网表。综合工具的产生实现了数字电路系统设计方法的又一次伟大飞跃。

1.2 Verilog 语言的特点

本节综述 Verilog HDL 语言的特点，并将其与其他语言进行比较。

1.2.1 Verilog 的由来

Verilog 是 Verilog HDL 的简称。Verilog 语言最初于 1983 年由 Gateway Design Automation 公司开发，于 1995 年被认证为 IEEE 标准。Verilog 语言不仅定义了语法，而且还对每个语法结构都清晰定义了仿真语义，从而便于仿真调试。Verilog 语言继承了 C 语言的很多操作符和语法结构，对初学者而言易学易用。另外 Verilog 语言具有很强的扩展性，最新的 Verilog 2001 标准大大扩展了 Verilog 的应用灵活性。

另外一种流行的 HDL 语言是 VHDL（Very High Speed Integrated Circuit HDL，超高速集成电路硬件描述语言），其发展初期得到了美国国防部的支持，并于 1987 年成为 IEEE 标准。VHDL 语言的特点是描述严谨。

为了加深读者对 Verilog 语言的理解，下面将其与其他几种电路描述方法进行对比。

1.2.2 HDL 与原理图

HDL 和原理图是两种最常用的数字硬件电路描述方法。原理图设计输入法在早期应用得比较广泛，它会根据设计要求，选用器件，绘制原理图，完成输入过程。这种方法的优点是直观，便于理解，元件库资源丰富；但是在大型设计中，这种方法的可维护性较差，不利于模块建设与重用，更主要的缺点是，当所选用的芯片升级换代后，所有的原理图都要作相应的改动。

目前进行大型工程设计时，最常用的设计方法是 HDL 设计输入法，其中影响最为广泛的 HDL 语言是 Verilog HDL 和 VHDL。它们的共同特点是利于自顶向下的设计，利于模块的划分与重用，可移植性好，通用性好，设计不因芯片工艺和结构的变化而变化，更利于向 ASIC 移植。

波形输入和状态机输入方法是两种常用的辅助设计输入方法。使用波形输入法时，只要绘制出激励波形和输出波形，EDA 软件就能自动根据响应关系进行设计或仿真。而使用状态机输入法时，设计者只需画出状态转移图，EDA 软件就能生成相应的 HDL 代码或者原理图。这两种设计方法往往与某种特定的设计工具相关，应用起来容易受到局限，而且效率和可维护性不高，仅仅在某些场合作为辅助的设计描述手段使用。

因此推荐初学者在描述和仿真数字电路时首选 HDL 语言方式，而在某些要求使用图形描述设计顶层的情况下才使用原理图，不要在设计顶层以外的其他层次使用原理图。另外不要依赖波形设计工具，因为简单的信号虽然用波形描述起来十分方便，但是复杂的测试激励几乎无法使用波形工具进行有效描述。

1.2.3 Verilog 和 VHDL

Verilog 和 VHDL 作为最流行的 HDL 语言，从设计能力上而言都能胜任数字电路系统的设计任务。

VHDL 最初被用作文档来描述数字硬件的行为，因此 VHDL 的描述性和抽象性更强，也就是说 VHDL 更适合描述更高层次（如行为级、系统级等）的硬件电路。

Verilog 最初是为更简捷、更有效地描述数字硬件电路和仿真而设计的，它的许多关键字和语法都继承了 C 语言的传统，因此易学易懂。

前面已经提到最流行的 HDL 语言是 Verilog 和 VHDL，后来在其基础上又发展出了许多抽象程度更高的硬件描述语言，如 SystemVerilog、Superlog、SystemC 和 CoWare C 等。这些高级 HDL 语言的语法结构更加丰富，更适合用于系统级、功能级等高层次的设计描述和仿真。HDL 语言适用层次示意图如图 1-1 所示，其中实线框表示适用程度较高，虚线框表示适用程度较低。

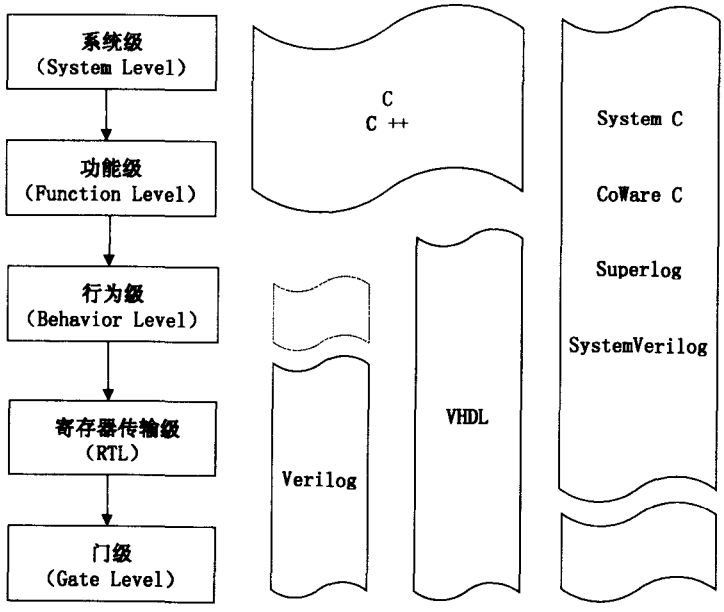

图1-1 HDL 语言适用层次示意图

本书推荐初学者先从 Verilog 学起。Verilog 语法的寄存器和线网两种数据类型定义清楚，时序与组合电路描述简洁，能够帮助初学者快速了解硬件设计的基本概念，非常容易上手，这也是 Verilog 成为最受欢迎的 HDL 语言的主要原因。

但是请读者明确，Verilog 与 VHDL 语言本身并没有什么优劣之分，而是各有所长。使用 HDL 语言描述数字硬件电路，其本质是将硬件电路抽象为语言这种表达形式，因此可以

说 HDL 是实际硬件电路与 EDA 工具之间的桥梁。选择何种语言作为桥梁本身并不重要，关键是如何有效地为真实电路建模，因此最重要的是建模的方法与思想。

1.2.4 Verilog 和 C 语言

Verilog 语言是根据 C 语言发明而来的，因此 Verilog 语言具备了 C 语言简洁易用的特点。Verilog 从 C 语言中借鉴了许多语法，例如预编译指令和一些高级编程语言结构等。

一、C 语言与 Verilog 的最大区别

C 语言与 Verilog 的最大区别在于 C 语言缺乏硬件描述的 3 个基本概念。

- 互连（connectivity）：在硬件系统中，互连是一个非常重要的组成部分，而在 C 语言中，并没有直接可以用来表示模块间互连的变量；而 Verilog 的 wire 型变量配合一些驱动结构能有效地描述出网线的互连。
- 并发（concurrency）：C 语言天生是串行的，不能描述硬件之间的并发特性，C 语言编译后，其机器指令在 CPU 的高速缓冲队列中基本是顺序执行的；而 Verilog 可以有效地描述并行的硬件系统。
- 时间（time）：运行 C 程序时，没有一个严格的时间概念，程序运行时间的长短主要取决于处理器本身的性能；而 Verilog 语言本身定义了绝对和相对的时间度量，在仿真时可以通过时间度量与周期关系描述信号之间的时间关系。

二、HDL 语言的本质

读者必须明确一点，就是硬件描述语言（HDL）同软件语言（如 C、C++等）是有本质区别的。Verilog 作为硬件描述语言，它的本质作用在于描述硬件。Verilog 虽然采用了 C 语言的形式，但是它的最终描述结果是芯片内部的实际电路。所以评判一段 HDL 代码优劣的最终标准是其描述并实现的硬件电路的性能（包括面积和速度两个方面）。评价一个设计的代码水平较高，仅仅是说这个设计由硬件向 HDL 代码这种表现形式转换得更流畅、更合理，而一个设计的最终性能，在很大程度上取决于设计工程师所构想的硬件实现方案的合理性。

初学者，特别是从软件设计转行的初学者，片面追求代码的整洁、简短是错误的，是与评价 HDL 的标准背道而驰的。正确的编码方法是，首先要对所要实现的硬件电路有一个清楚的认识，对该部分硬件的结构与连接了解得十分透彻，然后再用适当的 HDL 语句将其表达出来。

三、Verilog 与 C 的结合

Verilog 毕竟是硬件描述语言，它在抽象程度上比 C 语言要差一些，语法不如 C 灵活，在文件的输入与输出方面功能也明显不如 C。为了克服这些缺陷，Verilog 的设计者们发明了编程语言接口，也叫做 PLI。通过 PLI，可以在仿真器中实现 C 语言程序和 Verilog 程序间的互相通信，或者是在 Verilog 中调用 C 语言的函数库，从而大大扩展了 Verilog 语言的灵活性和高层次抽象的能力。开发时，一方面硬件设计者使用 Verilog 进行硬件建模，另一方面验证工程师却常常使用 C 语言来编写测试向量，然后通过 Verilog 的编程语言接口（PLI）将 Verilog 和 C 联系起来。

1.3 HDL 的设计与验证流程

HDL 的基本功能就是有效地描述并仿真硬件系统。这节我们抛开具体的 PLD 或 ASIC 设计流程，从 HDL 语言层次入手，分析典型的 HDL 设计与验证流程。HDL 的设计与仿真流程如图 1-2 所示，其中虚线框里的步骤可以依据项目的复杂程度而省略，实线框里的步骤为必须执行的步骤。

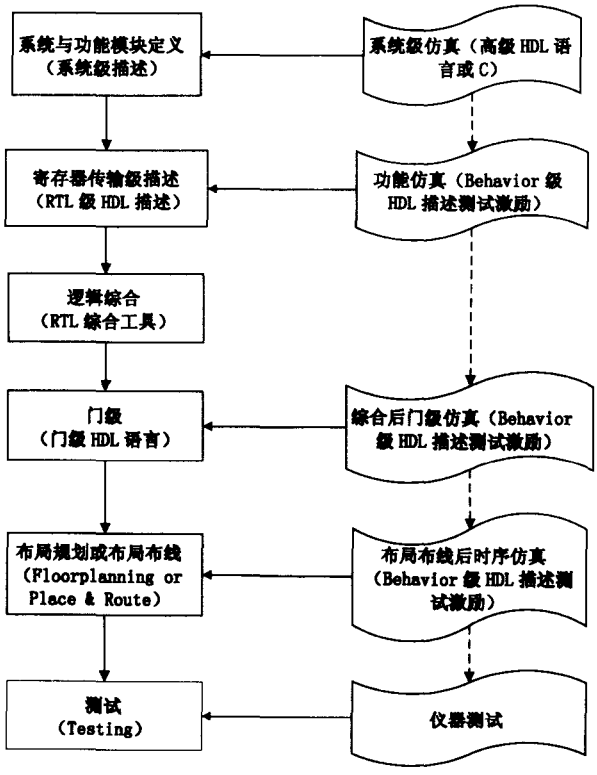

图1-2 HDL 的设计与仿真流程

下面具体讲一些关键的设计步骤与概念。

- **系统与功能模块定义（系统与功能模块级）**

  在大型系统的设计与实现中，首先要进行详细的系统规划和描述。此时 HDL 描述侧重于对整体系统的规划和实现。系统级仿真的主要目标是对系统的整体功能和性能指标进行衡量。系统级设计与仿真多采用高级描述语言，如 C/C++、System C 和 System Verilog 等。系统级描述完成后，应该进一步将系统功能划分为可实现的具体功能模块，大致确定模块间的接口，如时钟、读写信号、数据流和控制信号等，并根据系统要求描述出每个模块或进程的时序约束，这个细化的过程被称为功能模块级设计。功能模块级仿真主要是考察每个子模块或进程的功能与基本时序。在系统级与功能模块级设计层次，必须整体权衡多种实现方案之间孰优孰劣，根据系统性能指标要求，从整体上优化实现方案，从而更有效地满足设计需求。

- **行为级描述测试激励（Behavior Level）**

  行为级模块描述的最大特点是必须明确每个模块间的所有接口和边界。此时模块内部的功能已经明确，模块间的所有接口、顶层的输入和输出信号等在行为级已经被清晰地描述出来。在 PLD 和 ASIC 设计流程中，常用行为级描述方式编写测试激励。延时描述、监视描述等命令都是在编写测试激励的过程中常用的行为级语法。行为级描述常使用 HDL 语言，如 Verilog 和 VHDL 等。

- **寄存器传输级（RTL，Register Transfer Level）**

  寄存器传输级指不关注寄存器和组合逻辑的细节（如使用了多少逻辑门，逻辑门之间的连接拓扑结构等），通过描述寄存器到寄存器之间的逻辑功能描述电路的 HDL 层次。RTL 级是比门级更高的抽象层次，使用 RTL 级语言描述硬件电路一般比门级要简单、高效得多。寄存器传输级描述的最大特点是可以直接用综合工具将其综合为门级网表。RTL 设计直接决定着设计的功能和效率。好的 RTL 设计能在满足逻辑功能的前提下，使设计的速度和面积达到一种平衡。RTL 级描述最常用的 HDL 语言是 Verilog 和 VHDL 语言。

- **对 RTL 级描述进行功能仿真**

  一般来说需要对 RTL 级设计进行功能仿真，仿真的目的是验证 RTL 级描述是否与设计意图一致。为了提高效率，功能仿真的测试激励一般使用行为级的 HDL 语言描述。

- **逻辑综合（使用 RTL 级 EDA 工具）**

  RTL 级综合指将 RTL 级 HDL 语言翻译成由与、或、非门等基本逻辑单元组成的门级连接（网表），并根据设计目标与要求（约束条件）优化所生成的逻辑连接，输出门级网表文件。随着综合工具的不断智能化，使用 RTL 级语言描述硬件电路越来越方便，特别是在可编程逻辑器件（PLD，主要指 FPGA 和 CPLD）设计领域，最重要的代码设计层次就是 RTL 级。

- **门级（Gate Level）**

  目前大多数的 FPGA 设计都依靠专业综合工具完成从 RTL 级代码向门级代码的转换，设计者直接用 HDL 语言描述门级模型的情况越来越少，高效的综合工具将设计者从复杂烦琐的门级描述中彻底解放出来。目前要直接使用门级描述的情况一般是 ASIC 和 FPGA 设计中有面积或时序要求较高的模块。门级描述的特点是整个设计用逻辑门实现，通过逻辑门的组合显化描述设计的引脚、功能和时钟周期等信息。

- **综合后门级仿真**

  综合完成后如果需要检查综合结果是否与原设计一致，就需要进行综合后仿真。在仿真时，把综合生成的标准延时文件反标注到综合仿真模型中去，可估计门延时所带来的影响。综合后仿真虽然比功能仿真精确一些，但是只能估算门延时，不能估算线延时，仿真结果与布线后的实际情况还有一定的差距，并不十分准确。这种仿真的主要目的在于检查综合结果是否与原设计一致。目

前主流综合工具日益成熟，对于一般性设计而言，如果设计者确信自己的表述准确，不会产生歧义，则可以省略综合后仿真这一步骤。一般情况下，综合后仿真与功能仿真的仿真激励相同。

- **布局规划与布局布线**

  综合的门级结果最终要映射到目标库（如 ASIC 设计）或目标器件（如 PLD 设计）中。由于本书的重点为 HDL 设计，因此这里不再深究 ASIC 与 PLD 设计的相关流程。

- **布局布线后的时序仿真与验证**

  将最终布局规划或布局布线的延时信息反标注到设计网表中所进行的仿真就叫时序仿真或布局规划与布局布线后仿真，简称后仿真。布局规划与布局布线之后生成的仿真延时文件包含的延时信息最全，不仅包含门延时，而且还包含实际的布线延时，所以时序仿真最准确。它能较好地反映芯片的实际工作情况，建议进行时序仿真，通过时序仿真可以检查设计时序与芯片的实际运行情况是否一致，确保设计的可靠性和稳定性。时序仿真的主要目的在于发现时序违规（Timing Violation），即不满足时序约束条件或者器件固有时序规则（建立时间、保持时间等）的情况。

综上所述，HDL 语言中有两个非常重要的问题，一个是如何使用 HDL 语言在 RTL 级高效地描述电路，这个问题将在本书第 4、5、6 章中集中讨论；另一个是如何使用 HDL 语言在行为级描述测试激励，这个问题将在本书第 7、8 章中重点讨论。

1.4 问题与思考

1. Verilog 与 C 语言的最大区别是什么？
2. HDL 语言的本质是什么？
3. 基于 HDL 的设计验证流程包含哪些主要步骤？


读书笔记

第2章 Verilog 语言基础

Verilog HDL 曾经是一种私有语言，现在它已发展成为 ASIC 和 FPGA 设计领域中应用最为广泛的硬件描述语言。可以这样说，Verilog 在硬件设计领域中的地位甚至超过了 C 语言在软件编程领域中的地位。

Verilog 能发展到今天，与其本身的优越性有着很大的关系。它简单易学，语法更贴近硬件行为，同时还借鉴了许多 C 语言中的高级语句，支持多种层次、多种方式的描述，大大提高了工程师的设计效率。

想要全面掌握 Verilog 语言，首先得要从语法基础说起，同时还需要了解该语言与其他语言的相似点和不同点。

本章主要内容如下：

- Top-Down 和 Bottom-Up;
- 从一个实例开始;
- 基本词法;
- 模块和端口;
- 编译指令;
- 逻辑值与常量;
- 变量类型;
- 参数;
- Verilog 中的并发与顺序;
- 操作数、操作符和表达式;
- 系统任务和系统函数。

2.1 Top-Down 和 Bottom-Up

在传统意义上讲，设计硬件电路主要采用的是自底向上（Bottom-Up）的设计方法。工程师们总是从最底层的逻辑门开始，逐渐搭建出较大的模块，然后再将这些模块组成更大的模块，最后完成整个设计。

本书第 1 章已经介绍过，随着 HDL 和逻辑综合技术的进步，工程师们正逐步开始使用自顶向下（Top-Down）的方法来设计硬件。工程师们首先关注于设计的规格（Specification），然后将规格分解为一个个的模块，再分解为更小的模块，接着采用 HDL 的可综合子集直接描述硬件的行为，由逻辑综合工具自动完成从 HDL 到门级电路的转换。

最近几年 IP 核市场逐渐兴起，许多设计者逐渐意识到利用现有的 IP 核可以帮助他们节约设计成本，减少设计周期，许多人甚至希望所有的设计模块都使用现成的，自己仅仅开发

一些简单的粘合逻辑。这有点像电路板设计，工程师将各种芯片集成到一块电路板上，而自己只需完成这些芯片间的互连和一些简单的 CPLD 逻辑设计，以及对微处理器的编程即可。使用现有的 IP 来搭建系统，实际上也是一种自底向上的设计方法。

Verilog HDL 完全支持这两种设计方法。在门级的设计中，用户可以直接实例化 Verilog 语言中的门级原语构建系统。如果需要描述硬件行为，可以使用 Verilog 的行为级描述功能；如果要使用 IP 核，只要在设计中直接实例化 IP 核即可。

EDA 行业的先行者们发明了 Verilog 硬件描述语言，其最根本的目的就是用 Verilog 来描述硬件的行为，但是有的描述是不需要实现为硬件电路的。

硬件电路最大的特点是它由一个个模块组成，模块之间使用互连线，各个模块独立并行工作，同时各个模块会通过输入和输出端口与其相邻的模块互相沟通。每个硬件单元都有相应的延时特性，硬件的延时也是设计的目标之一。

2.2 Verilog 的 3 种描述方法

本节将通过一个实例引申出 Verilog 的 3 种描述方法。

2.2.1 实例

图 2-1 所示是一个简单的电路——HelloVlog，它既可以是一个独立的设计，也可以是更大系统的一个组成部分。

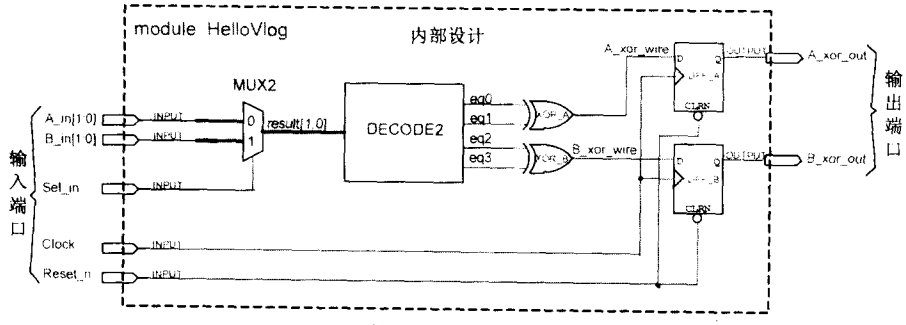

图2-1　HelloVlog 模块

该电路首先由 Sel_in 信号在两个 2 比特的输入数据 A_in[1:0]和 B_in[1:0]之间作二选一，后面是一个 2-4 译码电路，将输入的信号 result[1:0]解析成 eq0、eq1、eq2 和 eq3 4 个信号，它们同时只有一个为 1，其余 3 个信号为 0。

将 eq0 和 eq1 相异或（xor），eq2 和 eq3 相异或，然后将两个异或的结果 A_xor_wire 和 B_xor_wire 分别寄存输出给 A_xor_out 和 B_xor_out。两个输出寄存器带有低有效（active low）的异步复位端。

这些电路单元是并行工作的，相互之间并没有先后的顺序关系，这一点与软件程序设计不同。

下面使用 Verilog 语言描述图 2-1 中所示的电路，示例代码详见随书光盘中“Example-2-1”目录下的相关内容。

```
//以下是编译指令，用于定义时间单位和时间精度
`timescale 1ns / 100ps //时间单位是 1ns，仿真精度是 100ps

//以下是 module 名称和端口列表
module HelloVlog ( Clock, Reset_n, A_in, B_in, Sel_in, A_xor_out, B_xor_out );

//以下是输入和输出端口声明
input Clock;
input Reset_n;
input [1:0] A_in; //注意总线的标识方法，其中 A_in 是 2 比特输入总线
input [1:0] B_in;
input Sel_in;
output A_xor_out;
output B_xor_out;

//以下是线网和寄存器声明
wire A_xor_wire;  //模块内部的 wire 型变量
wire B_xor_wire;
wire [1:0] result;
reg eq0, eq1, eq2, eq3; //模块内部的 reg 型变量
reg A_xor_out; //将输出信号 A_xor_out 声明为 reg 型变量
reg B_xor_out;

//行为描述，DFF_A
always @ (posedge Clock or negedge Reset_n)
   if (~Reset_n)
      A_xor_out <= 0;
   else
      A_xor_out <= A_xor_wire;

//行为描述，DFF_B
always @ (posedge Clock or negedge Reset_n)
   if (~Reset_n)
      B_xor_out <= 0;
   else
      B_xor_out <= B_xor_wire;

//数据流描述，XOR_A
assign #1 A_xor_wire = eq0 ^ eq1 ;
```

```
//结构化描述，XOR_B，直接使用 Verilog 自带的 xor 门原语
xor #1 XOR_B ( B_xor_wire, eq2, eq3 );

//数据流描述，MUX2
assign #3 result = (Sel_in) ? B_in : A_in;

//行为描述，DECODE2，一个 2-4 译码器
always @ ( result )
begin
   case ( result )
   2'b00 : begin
           {eq3, eq2, eq1, eq0} = #2 4'b0001 ;
           $display ("At time %t - ", $time, "eq0 = 1");
      end
   2'b01 : begin
           {eq3, eq2, eq1, eq0} = #2 4'b0010 ;
           $display ("At time %t - ", $time, "eq1 = 1");
      end
   2'b10 : begin
           {eq3, eq2, eq1, eq0} = #2 4'b0100 ;
           $display ("At time %t - ", $time, "eq2 = 1");
      end
   2'b11 : begin
           {eq3, eq2, eq1, eq0} = #2 4'b1000 ;
           $display ("At time %t - ", $time, "eq3 = 1");
      end
   default : ;
   endcase
end

//module 结束
endmodule
```

对于 MUX2，二选一的电路，一般采用如下的数据流描述（assign 语句）：

```
assign #3 result = (Sel_in) ? B_in : A_in;
```

“assign”是 Verilog 中的关键词，用它赋值的语言称为连续赋值语句。

如果 Sel_in 为真（1），则选择 B_in，否则将选择 A_in。而“#3”表示经过 3 个延时单位，再将选择结果赋值给 result，这也模拟了组合逻辑的延时。由于例 2-1 中已经使用了 Verilog 的编译指令将延时单位定义为 1ns（`timescale 1ns/100ps），因此这里的#3 代表 3ns 的

延时。

关于实现 2-4 译码器，则采用了如下的描述：

```
always @ ( result )
begin
    case ( result )
    2'b00 : begin
            {eq3, eq2, eq1, eq0} = #2 4'b0001 ;
            $display ("At time %t - ", $time, "eq0 = 1");
        end
    ...
    default : ;
    endcase
end
```

很显然，这里采用了另一种描述方式，即用 always 语言来描述电路的行为。通过 case 判断 result 的值来决定 eq0～eq3 的值。同样这里也采用了“#2”来模拟组合逻辑的延时。“always”和“case”也是 Verilog 中的关键词，而这样的描述方法通常被称为行为描述，它侧重于描述电路的行为。

```
{eq3, eq2, eq1, eq0} = #2 4'b0001 ;
```

该语句表示将 4'b0001 赋值给 eq3～eq0 合并成的 4 位变量。“{}”是 Verilog 的合并符号。

对于 XOR_A 和 XOR_B 两个完全一样的异或门，也采用了两种不同的描述方式：

```
assign #1 A_xor_wire = eq0 ^ eq1 ;
```

和

```
xor #1 XOR_B ( B_xor_wire, eq2, eq3 );
```

xor 是 Verilog 中自带的基本逻辑门原语，这里相当于调用了该 xor 门，而 B_xor_wire、eq2 和 eq3 则是代入到 xor 门中的参数。eq2 和 eq3 是输入，B_xor_wire 是输出。这里 B_xor_wire 是 eq2 和 eq3 相异或的结果。

在 Verilog 中，通常将调用其他功能模块（包括 Verilog 的内嵌基本逻辑门）的行为称为“实例化（Instantiate）”。实例化类似于软件设计中的调用，但又不能简单理解为软件中的调用。软件中的调用过程是顺序执行的，而实例化的硬件电路在设计中是独立于其他功能块并行运行的。

这种在模块中实例化其他功能模块的描述方式称为结构化描述。

2.2.2 3 种描述方法

从以上描述可以看出 Verilog 语言有 3 种基本的描述方法。

- 数据流描述：采用 assign 语句，该语句被称为连续赋值语句。
- 行为描述：使用 always 或 initial 语句块，其中出现的语句被称为过程赋值语句。

- 结构化描述：实例化已有的功能模块。

所有的 Verilog 功能模块都是由这 3 种方式来描述的，其中结构化描述又包括 3 种形式。

- Module 实例化：实例化已有的 module。
- 门实例化：实例化基本的门电路原语。
- 用户定义原语（UDP）实例化：实例化用户定义的原语。

2.3 基本词法

Verilog HDL 是一种对大小写非常敏感的语言，也就是说同一个名称，用大写和用小写就代表了两种不同的符号，这一点与 VHDL 不同，因此在书写的时候要格外注意。

在 Verilog 语言中，所有的关键字（又叫保留字）都为小写。完整的 Verilog 关键字请参考本书附录 A。Verilog 的内部信号名（又称标识符）使用大写和小写都可以。标识符可以是字母、数字、$（美元符号）和_（下划线）的任意组合，只要第一个字符是字母或者下划线即可。

读者可能已经注意到，在例 2-1 中使用了双反斜线“//”来表示注释，除此之外，还有一种注释方式，即用“/*.....*/”来表示，所不同的是，前者为单行注释，后者则将“/*”和“*/”之间的内容全部看作注释内容。

通常，注释的内容只是设计者为了增强代码的可读性而增加的内容，对整个代码的功能没有任何影响。不过在一些工具，尤其是逻辑综合工具中还定义了一些特殊的指令，用于控制工具的编译过程，这些指令也是以注释的方式出现的，例如：

```
module bl_box(out,data,clk) /* synthesis syn_black_box */;
```

在 module bl_box 的声明处有一行注释，用“/*.....*/”表示。它看起来像是一个注释，但实际上却是综合工具 Synplify 中的一个指令，指示 Synplify 将该模块看作一个黑盒（black_box），不处理模块内部的描述。

在 Verilog 中，通常使用空格符、跳格符和换行符作为间隔。在书写代码的时候，适当运用间隔符可以提高代码的可读性。比如在声明 4 个 reg 型数据 eq0～eq3 时，可以采用例 2-1 中的方法，如：

```
reg eq0, eq1, eq2, eq3;
```

也可以用换行符将其分开，如：

```
reg eq0;
reg eq1;
reg eq2;
reg eq3;
```

在 Verilog 中还有一些转义字符，比如“\n”表示换行符，“\t”表示 Tab 键，为了防止引起歧义，就用“\\”表示“\”符号本身等。

这里不再对 Verilog 的词法做过多的阐述。读者如果想要了解更多的 Verilog 词法细节，可以参考本书所附光盘中的 Verilog IEEE 国际标准（IEEE Std 1364-1995）或其他文献资料。

2.4 模块和端口

大型设计往往是由一个个的模块构成的。

实际上，模块可大可小，大到一个复杂的微处理器系统，小到一个基本的晶体管，都可以作为一个模块来设计，例如 2.2.1 节中描述的 HelloVlog 就是一个模块。

在 Verilog 中，模块（module）是基本的组成单位。

通常情况下建议在一个 Verilog 文件中只放一个 module 定义，而且要使文件名称与 module 名称一致。

以下是 Verilog 中 module 的基本语法。

```
module 模块名称 (端口列表) ;
//声明:
   reg, wire, parameter,
   input, output, inout,
   function, task, ...
//语句:
   initial 语句
   always 语句
   module 实例化
   门实例化
   用户定义原语 (UDP) 实例化
   连续赋值 (Continuous assignment)
endmodule
```

首先需要有一个名称来标识这个 module。

通常 module 具有输入和输出端口。在 module 名称后面的括号中列出了所有输入、输出和双向端口的名称。

有些 module 不包含端口，例如在仿真平台的顶层模块中，其内部已经实例化了所有的设计模块和激励模块，是一个封闭的系统，没有输入和输出。这种没有端口的模块通常都是用于仿真的，不用作实际电路。

在 module 内部的声明部分，需要声明端口的方向（input、output 和 inout）和位宽。按照 Verilog 的习惯，高位写在左边，低位写在右边，比如“`input [1:0] A_in;`”就表示两位的总线。

模块内部使用的 reg（寄存器类型中的一种）、wire（线网类型中的一种）、参数、函数以及任务等，都将在 module 中声明。

一般来说，module 的 input 缺省定义为 wire 类型，output 信号可以是 wire 类型，也可以是 reg 类型（如果在 always 或 initial 语句块中被赋值），而 inout 是双向信号，一般将其设为 tri 类型，表示其有多个驱动源，如无驱动时则为三态。

虽然变量声明只要出现在其被使用的相应语句之间即可，但是还是建议将声明放在所有

的语句之前，这样具有较好的可读性。

在声明之后就应该是语句了，共有如下 6 种语句：

- initial 语句；
- always 语句；
- 其他子 module 实例化；
- 门实例化；
- 用户定义原语（UDP）实例化；
- 连续赋值（Continuous assignment）。

在 Verilog 中，所有的功能描述都是通过以上几种描述方式进行的。

初学者需要格外注意的是，以上几种语句如果出现在同一个 module 中，其相互之间是没有任何顺序关系的，它们在 module 中出现顺序的改变不会改变 module 的功能，这正是硬件的一大特点。有硬件电路原理图设计经验的读者可以想象一下画原理图的过程，先画哪个器件，后画哪个器件根本没有任何关系。

2.5 编译指令

Verilog 语言中提供了一些编译指令，例如定义宏、文件包含、条件编译、时间单位和精度定义等。这些编译指令都是从 C 语言中的“预处理指令”演变得来的。

这里列出了一些常用的编译指令：

- `timescale;
- `define、`undef;
- `ifdef、`else、`endif;
- `include;
- `resetall。

与 C 语言中使用“#”不同，Verilog 中使用反引号“`”来标识编译指令。编译器一旦遇到某个编译指令，则该指令将在整个编译过程中有效，直到编译器遇到另一个相同的编译指令为止。

比如在每个 module 文件前面加上`timescale 编译指令，就可以保证该文件中的延时信息受其自身文件中的`timescale 编译指令指导，否则在编译过程中，该模块将沿用上一个`timescale 的值，或者使用缺省值。

例 2-1 中的 HelloVlog 模块就使用了一个“`timescale 1ns/100ps”编译指令，其中 1ns 表示延时单位，100ps 表示时间精度，也就是编译器所能接收的最小仿真时间粒度。“`timescale”编译指令在模块外部出现，并且会影响后面模块中的所有延时值，直到遇到下一个“`timescale”或“`resetall”指令为止。

比如语句：

```
assign #1.16 A_xor_wire = eq0 ^ eq1 ;
```

如果采用`timescale 1ns/100ps 编译指令，由于延时单位是 1ns，最小时间粒度是 100ps，即 0.1ns，那么根据四舍五入的规则，1.16ns 实际上对应 1.2ns 延时。如果采用`timescale 1ns/10ps 编译指令，由于延时单位是 1ns，最小时间粒度是 10ps，即 0.01ns，那么 1.16ns 实

际上还是对应 1.16ns 延时。

`define 用于定义宏。例如可以首先定义一个总线宽度的宏为 16，然后利用这个宏定义一个宽度为 16 的 reg 类型数据 Data，方法如下：

```
`define BUS_WIDTH 16
...
reg [`BUS_WIDTH - 1 : 0] Data ;
```

在一个文件中出现的`define 可以被多个文件使用，也就是说`define 是一种全局性定义，这是`define 与 parameter 定义的最大区别。

`define 指令被编译以后，将在整个编译过程中有效，直到遇到`undef 指令为止，比如：

```
`undef BUS_WIDTH
```

当遇到该编译指令后，先前的`define 指令将会失效。

接着再来看看如下的条件编译指令。

```
`ifdef NARROW
   parameter BUS_WIDTH = 16;
`else
   parameter BUS_WIDTH = 32;
`endif
```

在这个条件编译指令中，如果先前已经定义了 NARROW 宏，那么参数 BUS_WIDTH 将被设置为 16，否则将被设置为 32。`else 指令对于`ifdef 来说是可选的，也就是说`ifdef 可以单独使用。

在 Verilog 中，可以使用`include 指令嵌入某个文件中的内容，比如：

```
`include "HEADFILE.h"
```

那么在编译的时候，就将使用 HEADFILE.h 文件中的内容完全替换这一行语句，而双引号中则可以通过指明相对路径或绝对路径，说明被引用文件的存储位置，其缺省值存储在当前路径下。

“`resetall”编译指令会将其他编译指令重新设置为缺省值，因此要谨慎使用。

Verilog 语言中的编译指令不止这几条，其他一些不常用的指令这里就不再一一介绍了，有兴趣的读者可以参考其他文献资料。

2.6 逻辑值与常量

2.6.1 逻辑值

学过数字电路的读者都知道，在二进制计数中，单比特逻辑值只有“1”和“0”两种状态，而在 Verilog 语言中，为了对电路进行精确建模，又增加了两种逻辑状态，即“X”和“Z”。

- 当“X”用作信号状态时表示未知，当用作条件判断时（在 casex 或 casez 中）表示不关心；

- “Z” 表示高阻状态，也就是没有任何驱动，通常用来对三态总线进行建模。

但是在综合工具眼中，或者说在实际实现的电路中，并没有什么 X 值，只存在 0、1 和 Z 3 种状态。在实际电路中还可能出现亚稳态，它既不是 0，也不是 1，而是一种不稳定状态。

Verilog 语言中的所有数据都是由以上描述的 4 种基本逻辑值 “0”、“1”、“X” 和 “Z” 构成的，同时，“X” 和 “Z” 是不区分大小写的，例如 0z1x 和 0Z1X 表示同一个数据。

2.6.2 常量

常量是 Verilog 中不变的数值，如例 2-1 中的 4'b0001 就表示一个 4 比特的二进制整数常量 0001。

Verilog 中的常量有以下 3 种类型：

(1) 整数型；

(2) 实数型；

(3) 字符串型。

用户可以使用简单的十进制表示一个整数型常量，例如：

- 16 表示十进制的 16；
- –15 表示十进制的–15，用二进制补码表示至少需要 5 位，即 10001，最高一位为符号位，如果用 6 位表示，则为 110001，同样最高一位为符号位。

整数型常量也可以采用基数表示法表示，例如：

- 8'haa 表示 8 位的 16 进制数，换算成二进制是 1010_1010；
- 6'o33 表示 6 位的 8 进制数，换算成二进制是 011_011；
- 4'b1011 表示 4 比特的二进制数 1011；
- 3'd7 表示 3 比特十进制的 7。

基数表示法的格式如下：

[长度]'数值符号 数字

其中长度可有可无，数值符号中 h 表示 16 进制，o 表示 8 进制，b 表示二进制，d 表示十进制。如果长度比后面数字的实际位数多，则自动在数字的左边补足 0，如果位数少，则自动截断数字左边超出的位数。

如果将数字写成 “'haa”，那么这个 16 进制数的长度就决定于数字本身的长度。

在基数表示法中如果遇到 X，则在 16 进制数中表示 4 个 X，在 8 进制数中表示 3 个 X。

另外，数字中的下划线没有任何意义，只不过是为了增强可读性，例如 4'b1011 和 4'b10_11。

Verilog 语言中的实数型变量可以采用十进制，也可以采用科学计数法，例如：

13_2.18e2 表示 13218。

字符串是指双引号中的字符序列，是 8 位 ASCII 码值的序列，例如 “Hello World”，该字符串包含 11 个 ASCII 符号（两个单词共 10 个符号，单词之间的空格为一个符号，共 11 个 ASCII 符号），因此需要用 11 个字节存储，方法如下：

```
reg [1: 8*11] Message;
...
Message = "Hello World" ;
```

这样即可将字符串常量存入到 Message 变量中。

2.7 变量类型

Verilog 语言中有以下两种变量类型。

- 线网型：表示电路间的物理连线。
- 寄存器型：Verilog 中一个抽象的存储数据单元。

凡是在 always 或 initial 语句中赋值的变量，一定是寄存器变量；凡是在 assign 语句中赋值的变量，一定是线网变量。

2.7.1 线网类型

线网类型下又包括几种子类型，它们具有线网的共性。

- wire、tri：表示电路间的物理连线，tri 主要用于多驱动源建模。
- wor、trior：表示该连线具有“线或”功能。
- wand、triand：表示该连线具有“线与”功能。
- trireg：表示该连线具有总线保持功能。
- tri1、tri0：表示当无驱动时，连线状态为 1（tri1）和 0（tri0）。
- supply1、supply0：分别表示电源和地信号。

在以上描述的线网类型中，除了 trireg 未初始化时的值为“X”以外，其余子类型未初始化时的值均为“Z”。

线网类型主要用在连续赋值语句中，并可作为模块之间的互连信号。

2.7.2 寄存器类型

寄存器类型的变量在 Verilog 语言中通常表示一个存储数据的空间，在 Verilog 仿真器中，寄存器类型的变量通常要占据一个仿真内存空间。

- reg：是最常用的寄存器类型数据，可以是一位、多位或二维数组（存储器）。
- integer：整数型数据，存储一个至少 32 位的整数。
- time：时间类型，存储一个至少 64 位的时间值。
- real、realtime：实数和实数时间寄存器。

reg 类型用来定义一位或者多位寄存器。

```
reg AB; //定义一个名为 AB 的一位寄存器
reg [3:0] ABC; //定义一个名为 ABC 的 4 位寄存器
```

在多位寄存器中可以进行位选择或部分选择，例如：

```
ABC [3] = 1;//将 ABC 的第 3 位赋值为 1
```

```
ABC [0] = 0;//将 ABC 的第 0 位赋值为 0
ABC [2:1] = 2'b01;//将 ABC 的第 1、2 位赋值为 1 和 0
```

这样整个 ABC 变量的值将为 4'b1010。

使用 reg 类型还可以定义二维寄存器数组，这种结构通常用于描述存储器（Memory）结构。

```
reg [3:0] MEMABC [0:7]; //定义一个存储器，地址为 0～7，每个存储单元都是 4 比特
```

与一维的 reg 变量不同的是，不能再对存储器中的存储单元进行位选择或部分选择，但是可以为每个单元单独赋值，比如：

```
MEMABC [1] = 4'b0101;//为 MEMABC 中的第 1 个存储单元赋值 4'b0101
```

在 Verilog 中，不存在一条语句可以对整个存储器赋值，必须对每个单元单独赋值，除非使用$readmemb 或$readmemh 系统任务从文件中读入整个或者部分存储器的数据。

本书第 7 章将讨论如何从文件中读入数据，给存储器赋值。

integer 变量通常用于高层次建模，也常用在 for 语句的索引中，例如：

```
initial
begin : ACCESS
   integer i ; //定义一个整数变量 i
   for ( i=0; i<= 7; i=i+1 )//遍历 0～7 地址
   begin
      MEMABC [i] = i;
   End
end
```

time 变量用于存储和处理系统时间，real 和 realtime 用来存储实数和实数时间。

2.7.3 变量的物理含义

线网变量可以理解为电路模块中的连线，但寄存器变量并不严格对应于电路上的存储单元，如触发器（flip-flop）和锁存器（latch）等。从纯粹的语言表达角度来说，寄存器变量的值将一个一个被保存下来，并且不会在仿真过程中丢失。

实际上，在使用 Verilog 仿真工具进行仿真的时候，寄存器类型的变量是占用仿真环境的物理内存的，这与 C 语言中的变量类似。寄存器变量在被赋值之后，便一直保存在内存中，直到再次对该寄存器变量进行赋值。而线网变量是不占用仿真内存的，它的值由当前所有驱动该线网的其他变量（可以是寄存器变量或线网变量）决定，这是寄存器变量和线网变量的最大区别，也是当初 Verilog 的发明者定义线网和寄存器变量的根本动机。

下一节中将引入“驱动”和“赋值”两个概念，深入探讨这两种变量的含义。

2.7.4 驱动和赋值

为了更清楚地描述寄存器变量和线网变量的含义，这里将引入 Verilog 语言中两个重要的概念——驱动（Driving）和赋值（Assigning）。

- 线网是被驱动的，该值不被存储，在任意一个仿真步进上都需要重新计算。

- 寄存器是被赋值的，且该值将在仿真过程中被保存，直到再次对该变量进行赋值。

例 2-1 中定义了一个 A_xor_wire 的 wire，它是 eq0 和 eq1 相异或的结果。这里采用如下方式进行描述：

```
assign #1 A_xor_wire = eq0 ^ eq1;
```

实际上，也可以采用另一种方式进行描述：

```
always @ (eq0 or eq1)
   A_xor_wire = #1 eq0 ^ eq1;
```

当然还需要在 module 的声明处将 A_xor_wire 首先定义成 reg 变量，而不是 wire 变量：

```
reg A_xor_wire;
```

这两种表述方式所对应的实际硬件电路是完全相同的，都是一个异或门，如图 2-2 所示。

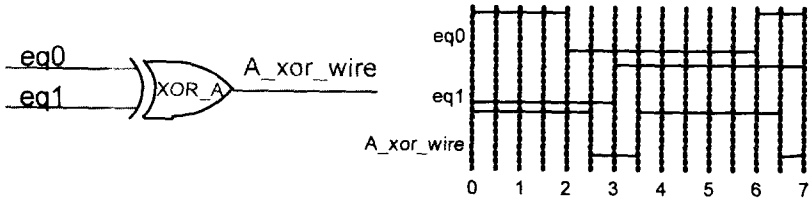

图2-2 异或门

那么从语义上来说，这两种描述方式有什么不同呢？

第一种描述方式使用“assign”语句，Verilog 中将其称为连续赋值语句（Continously Assignment），实际上就是连续驱动的过程，也就是说在任意一个仿真时刻，当前时刻 eq0 和 eq1 相异或的结果决定了 1ns 以后（语句#1 的延时控制）线网变量 A_xor_wire 的值，不管 eq0 和 eq1 变化与否，这个驱动过程一直存在，因此称为连续驱动。要知道，在仿真器中，线网变量是不占用仿真内存空间的，如图 2-2 所示，这个驱动过程在任意时刻（包括 0、1、2 ...7 等）都存在。

第二种描述方式使用了“always”语句，后面紧跟一个敏感列表@ (eq0 or eq1)，该语句只有在 eq0 或 eq1 发生变化后才会执行。图 2-2 中，在时刻 2、3 和 6，该语句都将被执行，将 eq0 和 eq1 赋值的结果延时 1ns 以后赋值给 A_xor_wire 变量，而在其他时刻，A_xor_wire 变量将保持不变。因此从仿真语义上讲，需要一个存储单元，也可以说是寄存器，来保存 A_xor_wire 变量的中间值，这就是 Verilog 语言中寄存器类型变量的来历，而这个 A_xor_wire 变量需要被定义为 reg 类型。

不管采用哪种方式，其所对应的硬件电路都是完全相同的组合逻辑电路。这里要特别注意第二种描述，虽然其在语法上被定义为 reg 类型，但并没有被实现为硬件上的触发器（flip-flop），所以说对于被定义为 reg 型的变量不能一概而论，需要具体分析其实现的硬件电路。有时 reg 型变量仅仅在仿真语义上被理解为寄存器概念，而在实际电路实现时却被实现为纯组合逻辑。

但是在对实际电路中的 D 触发器进行建模时，必须采用 reg 型的变量。图 2-3 所示为 D 触发器模型。

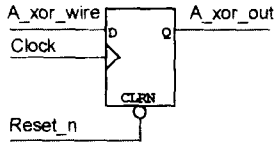

图2-3　D 触发器模型

例 2-1 中相应的描述如下：

```
always @ (posedge Clock or negedge Reset_n)
   if (~Reset_n)
      A_xor_out <= 0;
   else
      A_xor_out <= A_xor_wire;
```

D 触发器只对时钟和复位（置位）敏感，因此在敏感列表中列出了 Clock 的上升沿和 Reset_n 的下降沿。如果 Reset_n 为 0，触发器就会被复位，否则将会在 Clock 的时钟上升沿有效时，将 A_xor_wire 信号寄存到触发器的输出端 A_xor_out。

这样的代码精确地描述了一个 D 触发器的行为，这里的 reg 变量对应硬件中的 D 触发器。

在叙述时为了简单起见，常将“驱动”和“赋值”统一说成赋值，但是读者一定要清楚其本质。

2.8 参数

参数是一种常量，通常出现在 module 内部，常被用于定义状态机的状态、数据位宽和延时大小等，例如：

```
parameter and_delay = 2;
parameter xor_delay = 4;
```

可以在编译时修改参数的值，因此它又常被用于一些参数可调的模块中，使用户在实例化模块时，可以根据需要配置参数。

前面介绍的`define 是一种全局定义，而 parameter 是出现在模块内部的局部定义，而且可以被灵活改变，这是 parameter 的一个重要特征。

2.9 Verilog 中的并发与顺序

并行是硬件中一个非常重要的概念，是初学者或软件工程师需要重点掌握的知识。

与在处理器上运行的软件不同，硬件电路之间是并行工作的。为了描述硬件的并行性，Verilog 语言本身就具有并发的特性。在 Verilog 语言的 module 中，所有描述语句（包括连续赋值语句、行为语句块 always 和 initial 以及模块实例化等）都是并行发生的。

在语句块（always 和 initial 语句块）内部可以存在两种语句组。

- begin...end: 顺序语句组。
- fork...join: 并行语句组。

在 begin...end 中存在的语句应该是顺序执行的，而在 fork...join 中存在的语句则是并行执行的。

相对于顺序运行的事物来说，并行的事物比较难以理解，同时用于仿真的计算机是串行执行的，而 Verilog 语言本生的语义是用计算机进行模拟的语义，也就是用一种串行的语义来模拟并行的硬件。

Verilog 仿真器用来模拟硬件并行行为的方式类似于软件中的多任务操作系统，在某个时刻只能执行一个任务。

2.10 操作数、操作符和表达式

本节将讨论 Verilog 语言中的操作数、操作符及表达式等知识。

2.10.1 操作符

操作符是操作数之间的运算符号。在介绍操作数之前，先来了解一下 Verilog 中的操作符。Verilog 语言中的操作符如图 2-4 所示。

+	一元加	>>	右移
-	一元减	<	小于
!	一元逻辑非	<=	小于等于
~	一元按位求反	>	大于
&	归约与	>=	大于等于
~&	归约与非	==	逻辑相等
^	归约异或	!=	逻辑不等
^~或~^	归约异或非	===	全等
|	归约或	!==	非全等
~|	归约或非	&	按位与
*	乘	^	按位异或
/	除	^~或~^	按位异或非
%	取模	|	按位或
+	二元加	&&	逻辑与
-	二元减	||	逻辑或
<<	左移	?:	条件操作符

图2-4 操作符

其中，一元操作表示仅有一个操作数，二元操作表示有两个操作数。归约操作也是只有一个操作数，它是该操作数中所有比特之间的计算。

下面分类介绍 Verilog 中的操作符。

一、算术操作符

算术操作符		
+	m + n	将 n 与 m 相加

-	m - n	用 m 减去 n
-	-m	m 取反（二进制补码）
*	m * n	将 m 与 n 相乘
/	m / n	用 m 除以 n
%	m % n	对 m / n 求模

二、按位操作符

按位操作符		
~	~m	将 m 的每个比特取反
&	m & n	将 m 的每个比特与 n 的相应比特相与
|	m | n	将 m 的每个比特与 n 的相应比特相或
^	m ^ n	将 m 的每个比特与 n 的相应比特相异或
~^ ^~	m ~^ n m ^~ n	将 m 的每个比特与 n 的相应比特相异或非

三、归约操作符

归约操作符		
&	&m	将 m 中的所有比特相与（1 比特结果）
~&	~&m	将 m 中的所有比特相与非（1 比特结果）
|	|m	将 m 中的所有比特相或（1 比特结果）
~|	~|m	将 m 中的所有比特相或非（1 比特结果）
^	^m	将 m 中的所有比特相异或（1 比特结果）
~^ ^~	~^m ^~m	将 m 中的所有比特相异或非（1 比特结果）

四、逻辑操作符

逻辑操作符		
!	!m	m 是否不为真？（1 比特 真/假 结果）
&&	m && n	m 和 n 是否都为真？（1 比特 真/假 结果）
||	m || n	m 或者 n 是否为真？（1 比特 真/假 结果）

五、相等操作符

相等操作符（仅比较逻辑 1 和 0）		
==	m == n	m 和 n 相等吗？（1-bit 正确/错误结果）
!=	m != n	m 和 n 不相等吗？（1-bit 正确/错误结果）

六、全等操作符

全等操作符（比较逻辑 0、1、 x 和 z）		
===	m === n	m 和 n 全等吗？（1-bit 正确/错误结果）
!==	m !== n	m 和 n 全不等吗？（1-bit 正确/错误结果）

七、关系操作符

关系操作符		
<	m < n	m 小于 n？（1-bit 正确/错误结果）
>	m > n	m 大于 n？（1-bit 正确/错误结果）
<=	m <= n	m 小于或等于 n？（1-bit 正确/错误结果）
>=	m >= n	m 大于或等于 n？（1-bit 正确/错误结果）

八、逻辑移位操作符

逻辑移位操作符		
<<	m << n	将 m 左移 n 位

>>	m >> n	将 m 右移 n 位

九、条件操作符

? :	sel?m:n	如果 sel 为真，选择 m，否则选择 n

十、连接复制操作符

{}	{m,n}	将 m 和 n 连接起来，产生更大的向量
{{}}	{n{m}}	将 m 重复 n 次

以上描述的操作符之间有优先级之分，如表 2-1 所示。

表 2-1　　操作符优先级

操作符优先级	
! ~ + - (一元) * / % + - (二元) << >> < <= > >= == != === !== & ~& ^ ~^ | ~| && || ?:	最高优先级 最低优先级

例如 A + B & C + D 就表示（A+B）&（C+D），而不是 A+（B&C）+D。

2.10.2　二进制数值

在讨论操作数之前，先来看看二进制数中是如何表示有符号数和无符号数的。

在一个 6 比特二进制整形变量中：

- 无符号数能表示的范围是 0 ~ 63;
- 有符号数采用二进制补码（Two's complement）方式，能表示的范围是–32 ~ 31。其中二进制的最高位表示符号，最高位为 1 表示该数是负数，为 0 表示该数是正数。

这里不对具体的编码方式做过多的介绍，但要求读者必须掌握二进制中无符号数和有符号数的表示方法，以及计算的机制。

2.10.3　操作数

在 Verilog 语言中有以下几种操作数：

- 常数;
- 参数;
- 线网;
- 寄存器;
- 向量的位选择;
- 向量的部分选择;
- 存储器单元;
- 系统函数或用户自定义函数调用的返回值。

在选择操作数时需要注意操作数的极性。在 Verilog 中，无符号数通常以如下 3 种形式存在：

- 线网变量;
- 一般寄存器变量;
- 基数格式表示形式的整数常数。

而有符号数则以如下形式存在：

- 整型寄存器变量;
- 十进制形式的整型常量。

首先讨论常量，如果采用基数格式表示一个数，例如– 4'd12，其二进制表示方式则为 1111_1111_1111_1111_1111_1111_1111_0100（1100 的补码），由于基数格式的整数为无符号数，因此– 4'd12 的值就是十进制的 4294967284。

当采用普通十进制数来表示–12 的时候，虽然它的二进制表示方式与上面的数相同，但–12 是一个有符号数，因此在运算时就表示十进制的–12。

这里定义两个变量，一个是无符号的 reg 型变量，另一个是有符号的整型变量：

```
reg [4:0] Opreg; //一个 5 位的 reg 型变量，存储无符号数
integer Opint; //一个 32 位的 integer 型变量，存储有符号数
```

接下来进行如下运算：

```
Opreg = - 4'd12 / 4;
```

此时 Opreg 被赋值为 29，即（-4'd12 / 4）的最低 5 位，这是因为十进制数“-3”的 5 比特补码是“11101”，换算为无符号的十进制数则为“29”。

```
Opint = - 4'd12 / 4;
```

此时 Opint 被赋值为 1073741821，因为 Opint 被定义为 32 位的整数，按照前面的描述，– 4'12/4 这种表示形式虽然在表面上看是负数，但其实它的整数常数是 Verilog 的一种无符号数，这一点需要读者格外注意。

```
Opreg = - 12 / 4; //Opreg 被赋值为 29，(-12 / 4) 的最低 5 位
Opint = - 12 / 4; //Opint 被赋值为-3，采用 32 位的二进制补码表示方式
```

通过以上计算可以看出，无符号数和有符号数的算术运算有很大的不同，用户在设计常量和变量并用它们进行计算的时候，一定要搞清楚它们中哪些表示有符号数，哪些表示无符号数，这很重要。

2.11　系统任务和系统函数

Verilog 语言中预先定义了一些任务和函数，用于完成一些特殊的功能，它们被称为系统任务和系统函数。Verilog 能提供的系统任务和系统函数如下：

- 显示任务（display task）；
- 文件输入/输出任务（file I/O task）；
- 时间标度任务（timescale task）；
- 模拟控制任务（simulation control task）；
- 时序验证任务（timing check task）；
- PLA 建模任务（PLA modeling task）；
- 随机建模任务（stochastic modeling task）；
- 实数变换函数（conversion functions for real）；
- 概率分布函数（probabilistic distribution function）。

Verilog 的系统任务和系统函数种类很多，这里将重点介绍一些常用的内容，希望读者能够迅速掌握，灵活使用。

2.11.1　显示任务

$display 是显示任务，通常用来显示变量值、字符串和仿真时间等信息。

我们在例 2-1 中使用了这样的系统任务。

```
$display ("At time %t - ", $time, "eq0 = 1");//显示时间
```

其中，双引号中的是字符串，%t 是时间格式，$time 是产生模拟时间的系统函数，它的返回值显示在字符串中的%t 位置。

再如：

```
$display ("The value of ABC is %d", ABC);//显示当前 ABC 变量的值
```

%d 表示十进制数，ABC 的值显示在字符串中的%d 位置。

2.11.2　文件输入/输出任务

系统函数$fopen 用于打开一个文件，并返回一个整数的文件指针。然后，$fdisplay 就可以使用这个文件指针在文件中写入信息。写完后，则可以使用$fclose 系统关闭这个文件。例如：

```
integer Write_Out_File; //定义一个文件指针
Write_Out_File = $fopen("Write_Out_File.txt");
$fdisplay (Write_Out_File, "@%h\n%h", Mpi_addr, Data_in);
$fclose (Write_Out_File);
```

以上语法将“Mpi_addr”和“Data_in”分别显示在“@%h\n%h”中的两个%h 位置，并写入 Write_Out_File 指针所指的文件 Write_Out_File.txt 中。

用户可以通过$readmemb 或者$readmemh 从文件中读入数据，该文件中的数据格式是一

定的，例如：

```
reg [7:0] DataSource [0:47];
$readmemh ( "Read_In_File.txt", DataSource );
```

就是将 Read_In_File 文件中的数据读入到 DataSource 数组中，然后就可以直接使用这些数据了。Read_In_File 数据文件的格式如下：

```
@2f
24
@2e
81
...
```

其中@2f 表示地址，是 16 进制的；24 表示该地址的数据，以此类推。

2.11.3 其他系统任务和系统函数

一、仿真控制任务

Verilog 还提供了一些仿真控制任务，例如：

$finish 表示使仿真器退出；

$stop 表示使仿真挂起。

二、时序验证任务和仿真时间函数

Verilog 仿真器也可以用于检查设计时序及返回当前仿真时间等，例如：

$setup 系统任务用来检查建立时间；

$hold 系统任务用来检查保持时间；

$time 系统函数用来返回一个 64 位的模拟时间。

三、概率分布函数

$random 系统函数可以用来返回一个 32 位的有符号整型随机数。

初学者常常会犯一个错误，就是认为系统任务/系统函数可以在综合工具或者布线工具中运行。实际上，系统任务/函数只可以在 Verilog 仿真器中运行，仅仅对代码仿真有意义，综合工具和布线工具将忽略所有的系统任务和函数。

除了这些预先定义的系统任务和系统函数之外，Verilog 还允许用户自己定义任务和函数。关于自定义任务和函数的方法，请参考本书第 7 章中的相关内容。

2.12 小结

本章以一个实例为出发点，系统地介绍了 Verilog 的基本语法，为读者深入学习 Verilog 语言打下坚实的基础。

2.13 问题与思考

1. 一个模块必须有输入、输出端口吗？

2. 什么叫作编译指令？
3. Verilog 有哪两种变量类型？
4. 寄存器类型变量是否对应于硬件电路中的寄存器？
5. Verilog 的 3 种描述方式是什么？

第3章　描述方式和设计层次

Verilog 语言可以有多种方式来描述硬件，同时，使用这些描述方式，又可以在多个抽象层次上设计硬件，这是 Verilog 语言的重要特征。

本章主要内容如下：

- 描述方式；
- 数据流描述；
- 行为描述；
- 结构化描述；
- 设计层次；
- 实例。

3.1 描述方式

本书第 2 章中已经介绍过，在 Verilog 语言中，有以下 3 种最基本的描述方式。

- 数据流描述：采用 assign 连续赋值语句。
- 行为描述：使用 always 语句或 initial 语句块中的过程赋值语句。
- 结构化描述：实例化已有的功能模块或原语。

下面逐一对这些描述方式进行介绍。

3.2 数据流描述

3.2.1 数据流

在数字电路中，信号经过组合逻辑时会类似于数据流动，即信号从输入流向输出，并不会在其中存储。当输入发生变化时，总会在一定时间以后体现在输出端。同样，我们可以模拟数字电路的这一特性，对其进行建模，这种建模方式通常被称为数据流建模。数据流描述中最基本的语句是 assign 连续赋值语句。

3.2.2 连续赋值语句

图 3-1 中的模型可以用如下语句来描述：

```
assign #1 A_xor_wire = eq0 ^ eq1;
```

在任意一个时刻，A_xor_wire 线网的值都是由 eq0 和 eq1 决定的，也可以说是由它们驱

动的。

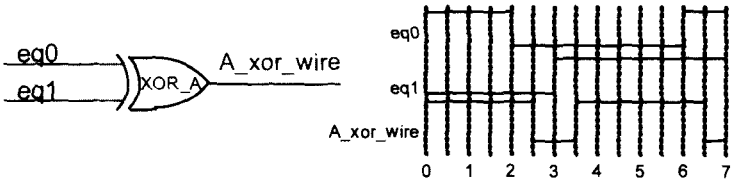

图3-1　一个电路模型

下面对连续赋值语句的特点进行说明。

一、连续驱动

连续赋值语句是连续驱动的，也就是说只要输入发生变化，都会导致该语句的重新计算。

二、只有线网类型的变量才能在 assign 语句中被赋值

由于连续赋值语句中被赋值的变量在仿真器中不会存储其值，因此该变量是线网类型（Net）的，而不是寄存器类型的。

另外，线网类型的变量可以被多重驱动，也就是说可以在多个连续赋值语句中驱动同一个线网。

但是寄存器变量就不同了，它不能被不同的行为进程（例如 always 语句块）驱动。

三、使用 assign 对组合逻辑建模

建议使用 assign 对组合逻辑建模，这是因为 assign 语句的连续驱动特点与组合逻辑的行为非常相似，而且在 assign 语句中加延时可以非常精确地模拟组合逻辑的惯性延时。

四、并行性

assign 语句与行为语句块（always 和 initial）、其他连续赋值语句、门级模型之间是并行的。一个连续赋值语句是一个独立的进程，进程之间是并发的，同时也是交织的。

五、实例

图 3-2 所示是一个由两个半加器和一个或门组成的全加器，下面使用连续赋值语句描述这个电路。

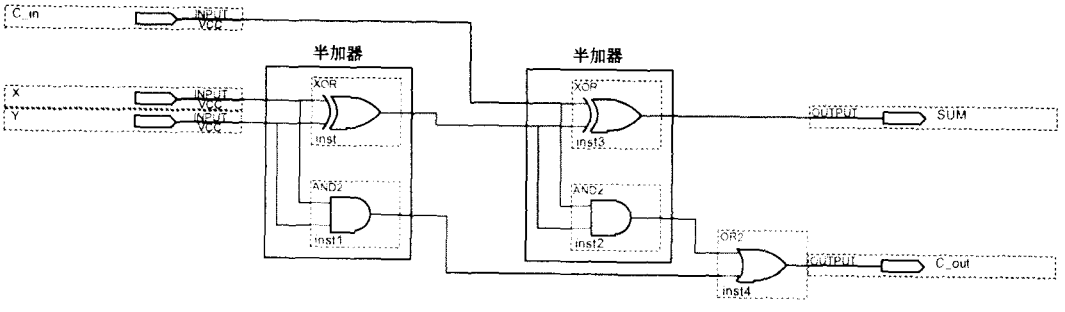

图3-2　两个半加器和一个或门组成的全加器

【例 3-1】　全加器实例，示例代码详见随书光盘中“Example-3-1”目录下的相关内容。

```
module HalfAdd (X, Y, SUM, C_out);//半加器模块
```

```
input X;
input Y;
output SUM;
output C_out;
assign SUM = X ^ Y ;
assign C_out = X & Y ;
endmodule

module FullAdd (X, Y, C_in, SUM, C_out);//全加器模块
input X;
input Y;
input C_in;
output SUM;
output C_out;

wire HalfAdd_A_SUM;
wire HalfAdd_A_COUT;
wire HalfAdd_B_COUT;

assign C_out = HalfAdd_A_COUT | HalfAdd_B_COUT ;

HalfAdd u_HalfAdd_A (  //半加器实例A
    .X     (X),
    .Y     (Y),
    .SUM   (HalfAdd_A_SUM),
    .C_out (HalfAdd_A_COUT) );

HalfAdd u_HalfAdd_B (  //半加器实例B
    .X     (C_in),
    .Y     (HalfAdd_A_SUM),
    .SUM   (SUM),
    .C_out (HalfAdd_B_COUT) );
endmodule
```

在 HalfAdd 模块中，两个 assign 语句之间是独立并行的，它们的顺序与逻辑功能无关。同样，在 FullAdd 模块中，两个 HalfAdd 的实例与“或”门的 assign 语句之间也是独立的。

3.2.3 延时

在连续赋值语句中，可以对电路的延时进行建模，当然也可以没有延时。

比如：

```
assign #1 A_xor_wire = eq0 ^ eq1; //`timescale 1ns/1ns
```

这个语句就表示该异或门的延时为 1ns，也就是说从输入端信号变化到输出端体现变化需要 1ns 的时间。

实际上，电路对不同的信号跳变表现出的延时往往并不一致，这些延时模型包括上升沿延时（输出变为 1）、下降沿延时（输出变为 0）、关闭延时（输出变成 Z，高阻态）和输出变成 X 的延时。

比如：

```
assign #(1,2) A_xor_wire = eq0 ^ eq1;
assign #(1,2,3) A_xor_wire = eq0 ^ eq1;
```

第一句表示上升延时为 1ns，下降延时为 2ns，关闭延时和传递到 X 的延时取两者中最小的，即 1ns。

第二句表示上升延时为 1ns，下降延时为 2ns，关闭延时为 3，传递到 X 的延时取 1、2、3 中最小的，即 1ns。

在一些电路模型中，延时分为最大、典型和最小 3 种情况。连续赋值语句中的延时也可以采用 min:typ:max 的格式来表示。

如：

```
assign #(4:5:6, 3:4:5) A_xor_wire = eq0 ^ eq1;
```

表示上升延时的 min:typ:max 为 4:5:6；下降延时的 min:typ:max 为 3:4:5。

需要注意的是，连续赋值语句中的延时具有硬件电路中惯性延时的特性，也就是说任何小于其延时的信号变化脉冲将被滤除掉，不会体现在输出端口上。关于这部分内容将在本书第 8 章中详细介绍。

另外，assign 语句中的延时特性通常是被逻辑综合工具忽略的，因为综合工具要将 Verilog 语言模型综合成逻辑电路，而逻辑电路的延时又是由基本的单元库和走线延时决定的。用户无法对逻辑单元指定延时，但是可以在综合和实现工具中添加时序约束，让工具尽量满足设计的时序要求。

3.2.4　多驱动源线网

下面描述当线网具有多重驱动源时的情况。

一、多重驱动 wire，错误用法

```
module WS (A, B, C, D, WireShort);
input A, B, C, D;
output WireShort;
wire WireShort;  //显式定义为 wire 类型
assign WireShort = A ^ B ;
assign WireShort = C & D ;
endmodule
```

在以上代码中，由于 WireShort 为 wire 类型，同时它有多重驱动源，因此仿真时

WireShort 的值将为 X，也就是不定态。

但是在综合工具眼中，该语法是错误的。

二、线与、线或功能

可以使用 wor 线网类型将不同的输出“线或”在一起。

```
module WO(A, B, C, D, WireOr);
input A, B, C, D;
output WireOr;
wor WireOr; //显式定义为 wor 类型
assign WireOr = A ^ B ;
assign WireOr = C & D ;
endmodule
```

逻辑综合以后，其对应的逻辑电路如图 3-3 所示。

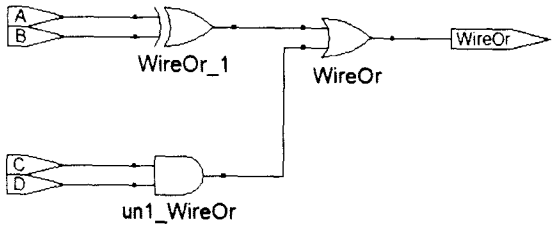

图3-3 线或功能

同样，可以使用 wand 线网类型将不同的输出“线与”在一起。

```
module WA(A, B, C, D, WireAnd);
input A, B, C, D;
output WireAnd;
wand WireAnd; //显式定义为 wand 类型
assign WireAnd = A ^ B ;
assign WireAnd = C & D ;
endmodule
```

其对应的逻辑电路如图 3-4 所示。

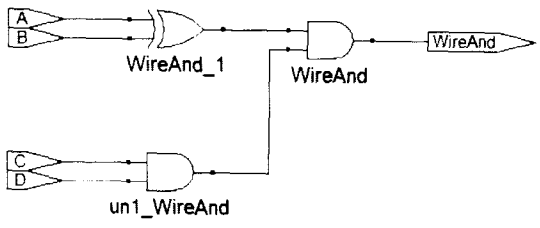

图3-4 线与功能

三、三态总线功能

如果要实现多个三态总线相连，可以采用 tri 型线网。

```
module WT (A, B, C, D, WireTri, En1_n, En2_n);
```

```
input A, B, C, D, En1_n, En2_n;
output WireTri;
tri WireTri;  //显式定义为 tri 类型
assign WireTri = (En1_n) ? 1'bZ : (A ^ B) ;
assign WireTri = (En2_n) ? 1'bZ : (C & D) ;
endmodule
```

其对应的电路如图 3-5 所示。

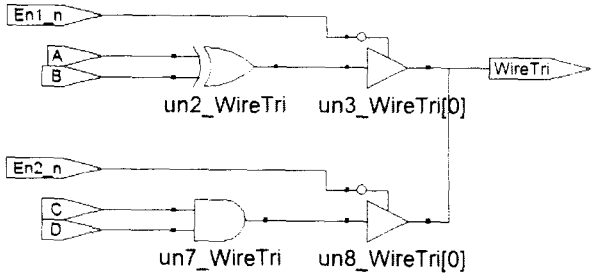

图3-5　三态驱动总线

3.3 行为描述

所谓行为描述是指用语言描述电路的行为。行为描述的语句有两种，即 initial 和 always 语句。

3.3.1 行为描述的语句格式

在 initial 和 always 的后面一般要跟语句或语句组（statement group）。

语句可以是非阻塞过程赋值、阻塞过程赋值、连续过程赋值或高级编程语句。

一、initial 或 always 过程块（procedual block）

initial 语句在 0 仿真时间执行，而且只执行一次；always 语句同样在 0 仿真时间开始执行，但是它将一直循环执行，这样的特点单单从它们的命名上就能看得出来：一个是 initial，就是初始化一次的意思；一个是 always，就是总在运行的意思。

下面使用 initial 和 always 语句各自的特点，产生一个时钟发生器的模型。

```
`timescale 1ns/1ns
module ClockGen (Clock);
output Clock;
reg Clock;

initial //将 Clock 初始化为 0;
   Clock = 0;

always  //每隔 5ns 将 Clock 翻转一次
```

```
    #5 Clock = ~ Clock;
endmodule
```

在 0 时刻，initial 和 always 语句同时执行，顺序随机。假设先运行 initial 语句，那么 Clock 变量将被赋值为 0，这时 initial 语句进程将被永远挂起，再也不会执行。

然后开始运行 always 语句。该 always 语句每隔 5ns 会将 Clock 信号翻转一次，并一直不停地运行，这样就产生了一个周期是 10ns 的时钟信号。假设先执行的是 always 语句，那么 Clock 就不会被初始化为 0 了。

二、过程块中的语句种类

过程块中的语句可以是非阻塞过程赋值、阻塞过程赋值、连续过程赋值或高级编程语句，语句组可以是 begin...end 和 fork...join 两种。

语句组中可以有其他几种语句类型，而高级编程语句中也可以有语句组，它们可以互相嵌套，完成非常复杂的逻辑功能描述。

以下是 always 过程块中直接跟阻塞赋值语句的情况。

```
always  //每隔 5ns 将 Clock 翻转一次
    #5 Clock = ~ Clock;
```

下面的代码体现了语句组和高级编程语句间的互相嵌套。

```
always @(posedge Clock or negedge Rst_n)
begin  //语句组
    if (~Rst_n) //高级编程语句
   begin //语句组
        Reg_A <= 0 ;  //非阻塞赋值语句
        Reg_B <= 0 ;  //非阻塞赋值语句
    end
    else //高级编程语句
   begin //语句组
        Reg_A <= Input_A ;  //非阻塞赋值语句
        Reg_B <= Input_B ;  //非阻塞赋值语句
    end
end
```

三、时序控制（Timing Control）

在行为描述中，可以采用 3 种方式对设计模型进行时序控制，分别为事件语句（“@”）、延时语句（“#”）和等待语句。

当执行 initial 或 always 语句块时遇到一个事件语句（“@”）、延时语句（“#”）或表达式值为假（false）的等待语句后，语句块（或称为进程）将被挂起（suspended），直到发生该事件、已经过了指定延时的时间单位数，或者等待语句表达式变为真（ture）时，才能重新执行 initial 或 always 语句块，这个过程就是时序控制。Verilog 的行为描述正是利用这几种时序控制语句来实现各种各样的逻辑功能。

下面先来分析一下事件语句（@）的用法。

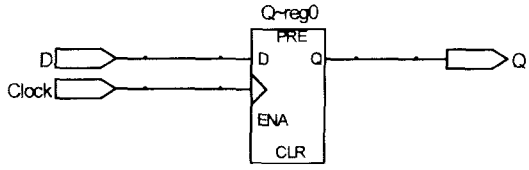

图3-6　D 触发器模型

要实现图 3-6 中所示的这样一个 D 触发器，通常采用以下代码：

```
module TYP_DFF (Clock, D, Q ) ;
input Clock, D;
output Q;
reg Q;
always @ (posedge Clock)
begin
   Q <= D;
end
endmodule
```

在 0 仿真时刻，always 语句块开始执行。当遇到@(posedge Clock)语句时，该进程将被挂起，直到 Clock 的上升沿到来，才能重新激活该进程。当 Clock 的上升沿出现后，将 D 的值赋给 Q，always 语句块执行完成。

基于 always 语句的特点，always 语句会马上开始重新执行，当遇到@(posedge Clock)语句时，进程再一次被挂起，直到 Clock 的上升沿到来，才能继续往下执行。

这样通过在 always 语句中使用@事件语句，即可很好地模拟触发器的行为，综合工具会马上将上述代码映射成一个如图 3-6 所示的 D 触发器。

同样的道理，采用以下代码也可以得到相同的 D 触发器。

```
module TYP_DFF (Clock, D, Q ) ;
input Clock, D;
output Q;
reg Q;
always
begin
   @ (posedge Clock)
   Q <= D;
end
endmodule
```

当有多个条件语句时，一般将它们用“or”分隔开。例如要实现一个带异步复位端的 D 触发器，可以采用如下代码：

```
module TYP_DFF (Clock, D, Q, Rst ) ;
input Clock, D, Rst;
```

```
output Q;
reg Q;
always @ (posedge Clock or posedge Rst)
begin
    if (Rst)
        Q <= 0;
   else
        Q <= D;
end
endmodule
```

当出现 Clock 或 Rst 的下降沿时，才会触发 always 语句。

下面再来看看如何使用延时语句进行时序控制。

```
always  //每隔 5ns 将 Clock 翻转一次
    #5 Clock = ~ Clock;
```

当 always 语句开始执行时，马上会遇到#5，此时 always 语句块将被挂起，直到 5ns 以后才恢复执行，这时将 Clock 信号的值取反。当再次执行 always 时，动作与上一次完全一致。这里模拟了一个周期为 10ns 的时钟。

需要强调的是，上面这种写法一般仅用于生成仿真激励，其综合实现结果并不是时钟发生器电路。事实上，综合工具在综合时会将延时语句“#5”忽略，所以上述代码无法综合成一个 10ns 周期的时钟发生器电路，而仅能在仿真时产生测试激励。

以下代码利用延时语句产生一个复位信号。

```
initial
begin
    Rst_n = 1;
    #5 Rst_n = 0;
    #100 Rst_n = 1;
end
```

当 initial 语句开始运行时，首先将 Rst_n 赋值为 1。当遇到#5 时，该 initial 的执行过程将被暂时挂起，等待 5ns 后才能恢复执行，Rst_n 被置为 0，处于复位状态。当遇到#100 时，该过程要等待 100ns 以后才能恢复执行，Rst_n 被置为 1。这时，initial 语句块将被永远挂起，再也不会执行，于是就产生了一个 100ns 的复位信号。

下面分析一下如何使用等待语句进行时序控制。

```
module MY_LATCH (Strobe, D, Q ) ;
input Strobe, D;
output Q;
reg Q;
always
begin
wait (Strobe == 1);
```

```
    Q = D;
    end
    endmodule
```

该语句表示，当 always 语句开始执行后遇到 wait()语句时，如果括号内的变量不为真，则该进程将被挂起，直到(Strobe == 1)为真后，always 才继续往下执行，将 D 的值赋值给 Q，这样就模拟了一个电平敏感的锁存器。

需要注意的是，目前多数综合工具还不支持 wait 语句，因此这个锁存器只能在仿真时使用，不能实现为具体的电路。

3.3.2 过程赋值语句

所谓过程赋值语句就是在 initial 和 always 语句块中的赋值语句。赋值对象只能是寄存器变量。右边的表达式可以是任意操作符的表达式。

一、阻塞赋值

阻塞赋值的语法如下：

寄存器变量 = 表达式 ；

所谓阻塞赋值实际上有两层含义：

第一，右边表达式的计算和对左边寄存器变量的赋值是一个统一的原子操作中的两个动作，这两个动作之间不能再插入其他任何动作；

第二，如果有多个阻塞赋值语句顺序出现在 begin...end 语句中，则前面的语句在执行时将完全阻塞后面的语句，直到前面语句的赋值完成以后，才会执行下一句中右边表达式的计算。例如在“begin m=n; n=m; end”语句中，只有当 m 被完全赋值以后，才开始执行 n=m，将 m 的新值赋予 n。这样执行的结果就是 n 的初始值不变，而且 m 与 n 相等。

基于这一特点，建议在为组合逻辑建模时采用阻塞赋值，比如：

```
wire A_in, B_in, C_in;
reg Temp, D_out;
…
always @ (A_in or B_in or C_in)
begin
   Temp = A_in & B_in; //阻塞赋值
   D_out = Temp | C_in; //阻塞赋值
end
```

上述代码所对应的电路如图 3-7 所示。

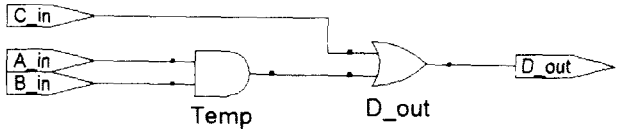

图3-7　一个组合逻辑

首先，任何一个输入发生变化，D_out 必然发生变化，因此在 always 的敏感列表中包

括 A_in、B_in 和 C_in。在进行内部计算时，首先将 A_in 和 B_in 相与，得到一个中间结果 Temp。等 Temp 被完全赋值后，才开始执行下一个语句，即将 Temp 的新值与 C_in 相或，得到 D_out 的值。

另外，有一点会令初学者产生较大的疑问，就是为什么 Verilog 规定只有寄存器（register）类型的变量才能够在过程赋值语句中被赋值。有时候在 Verilog 中定义的寄存器变量，在综合时并不一定会映射成一个实实在在的触发器硬件。比如在上面的例子中，Temp 和 D_out 被定义成 reg 变量，而综合结果却还是组合逻辑，而不是存储单元。

在 Verilog 语言中，寄存器变量的特点是，它需要在仿真运行器件中保存其值，也就是说这个变量在仿真时需要占据内存空间。

在上面的 always 实例中，always 语句块只对 A_in 等 3 个变量的输入变化敏感。如果没有这 3 个变量的变化事件，则 Temp 和 D_out 变量将需要保存其值，因此它们必须被定义为寄存器类型的变量。但是在综合之后，并不对应硬件锁存器或者触发器。

二、非阻塞赋值

非阻塞赋值的语法如下：

寄存器变量 <= 表达式 ；

在执行该语句时，首先计算右边的表达式，而且并不立刻为左边的变量赋值。由于这个赋值操作在当前仿真时间事件队列中的优先级比较低，因此将赋值操作推迟到当前仿真时刻的后期执行。有关这方面的知识请参考本书第 8 章中的内容。

与阻塞赋值不同的是，如果有多个非阻塞赋值语句顺序出现在 begin...end 语句中，那么前面语句的执行并不会阻塞后面语句的执行。例如“begin m<=n; n<=m; end”语句，其最后的结果是将 m 的值与 n 的值互换。

如果采用以下代码来描述图 3-7 中所示的电路功能：

```
wire A_in, B_in, C_in;
reg Temp, D_out;
...
always @ (A_in or B_in or C_in)
begin
   Temp <= A_in & B_in;  //非阻塞赋值
   D_out <= Temp | C_in; //非阻塞赋值
end
```

则可以发现，该功能并不是所需的功能。比如当 A_in 发生变化时执行 always 语句，其中“Temp <= A_in & B_in;”这句话并没有立刻对 Temp 赋值，而是放在当前仿真时刻的后期才开始执行对 Tamp 信号的更新。这样当执行“D_out <= Temp | C_in;”表达式的右式计算事件时，Temp 的值还是旧值，因此 D_out 并不会发生变化。

读者可以尝试将如上代码用综合工具综合一下，可能也会得到如图 3-7 所示的电路，这是由于一些综合工具可以容忍这些代码缺陷，从而造成了 RTL 仿真和综合结果不一致的现象发生。

RTL 仿真器严格按照 Verilog 的仿真语义执行 RTL 的仿真过程，而综合工具通常只是根

据代码推断设计者的意图，然后生成相应的电路结构，因此，综合的过程有一定的主观推断性，并不严格遵守 Verilog 语义，另外不同综合工具的判决标准也不一样。

如果仿真和综合结果不一致，就说明源代码中很可能存在隐患，不符合 Verilog 的语义，会错过许多 bug，增加设计的不稳定性，这种情况是每个设计都应该尽量避免的。

一般情况下常利用非阻塞赋值的特点来对时序逻辑进行建模，比如：

```
always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
end
```

该代码实现的结果如图 3-8 所示。

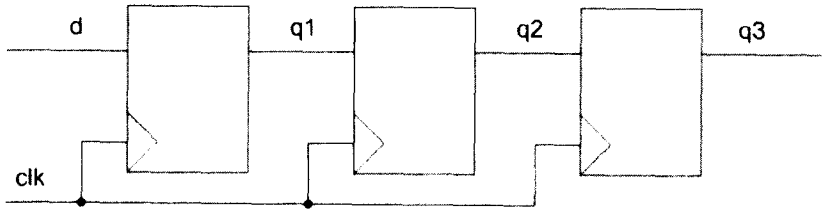

图3-8　三级流水寄存器

三、过程连续赋值

在 Verilog 语言中还有一种过程赋值语句，叫作“过程连续赋值”，它们也出现在 always 和 initial 语句块中。

过程连续赋值的类型主要有以下两种。

- assign 与 deassign：在过程语句块中强制为寄存器变量赋值并释放；
- force 与 release：在过程语句块中对寄存器和线网进行强制赋值和释放。

下面举一个例子，该实例使用 assign 和 deassign 描述了一个带异步清零端的 D 触发器。

```
module DEF(D , Clr , Clk , Q) ;
input D , Clr , Clk;
output Q;
reg Q;
always @ (Clr) begin
   if( !Clr)
      assign Q = 0; // D 的值对 Q 无效，将 Q 强制为 0
   else
      deassign Q; // 将强制的 Q 值释放
end
always @ (negedge Clk)
   Q = D;
endmodule
```

3.3.3 语句组

语句组是两条以上语句的组合，它们看起来像一个独立的语句。语句组也出现在 initial 和 always 过程块中。

根据语句的执行顺序，语句组可以分为“顺序语句组”和“并行语句组”两类。

一、顺序语句组 begin...end

在顺序语句组中各语句是一条一条顺序执行的，比如下面的语句：

```
always @ (A_in or B_in or C_in)
begin
    Temp = A_in & B_in; //阻塞赋值
    D_out = Temp | C_in; //阻塞赋值
end
```

首先执行第一句，将 A_in 和 B_in 相与，然后将结果赋给 Temp 变量；再执行第二句，将新的 Temp 值与 C_in 相或，结果会立刻被赋予 D_out。

当然，这里是同时利用了 begin...end 语句组和阻塞赋值的特点，才实现了用户想要的逻辑功能。

再比如要产生一个值序列：

```
initial
begin
DataBin = 0;
# 6 DataBin = 0;
# 4 DataBin = 1;
# 2 DataBin = 0;
end
```

由于语句是顺序执行的，因此产生的波形如图 3-9 所示。

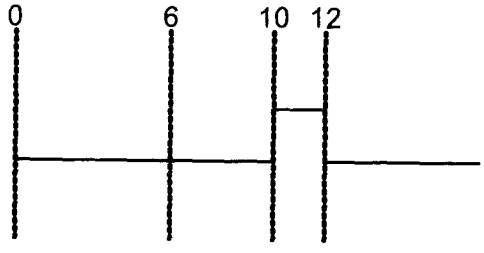

图3-9 顺序语句块产生的波形

二、并行语句组 fork...join

在 fork...join 语句组中语句是并行执行的。将上面的代码改写为：

```
initial
fork
DataBin = 0;
```

```
    # 6 DataBin = 0;
    # 4 DataBin = 1;
    # 2 DataBin = 0;
    join
```

由于所有语句是并行执行的，也就是说以上 4 条语句都是从 0 时刻开始同时执行的，因此产生的波形如图 3-10 所示。

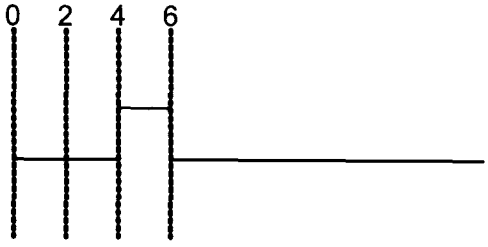

图3-10　并行语句块产生的波形

三、语句组的标识符

语句组既可以有标识符，也可以没有标识符。

当一个语句组有标识符时，在语句组内部可以定义局部变量，而不会传递到语句组的外部。然而在仿真语义上，这个变量是静态变量，它的值在整个仿真运行周期中是不变的，而且也不会与其他语句组中同一个名称的变量发生冲突。

例如：

```
integer i; //always 语句以外的 i 变量
always @ (...)
begin: SORT
...
integer i; //语句组内部的 i 变量
for (i=0;i<=7;i=i+1)
...
end
```

在 always 以外定义的 i 变量和在 always 里面定义的 i 变量属于两个不同的变量，并不冲突。在仿真的时候它们将占用两块不同的内存，类似于 C 语言中的静态局部变量。

3.3.4　高级编程语句

一、为什么需要编程语句

Verilog 作为硬件描述语言，最重要的特性就是其设计层次比较高，不仅仅停留在晶体管级和门级，而是可以在更高的层次，如 RTL 级甚至是行为级描述硬件系统的行为，或者编写测试激励。

为了达到提高描述能力和抽象层次的目的，Verilog 从 C 语言等编程语言中借鉴了一些

语句，同时也创造了一些语句，例如 if、case、while、for、repeat 和 forever 等，通常称这些语句为高级编程语句。有了这些语句，Verilog 才可以描述比较复杂的电路行为。

编程语句只能出现在 initial 和 always 的过程块中。编程语句中还可以嵌套其他语句，如过程赋值语句等。

可以将高级编程语句分为以下 3 大类：

- if-else 语句；
- case 语句；
- 循环语句，如 forever、repeat、while 和 for 等。

二、if-else 语句

if 语句后面跟语句或语句组（begin...end 或 fork...join），常和 else 搭配来实现不同条件下的各种情况。if 也可以单独使用。

请参考如下代码：

```
always @(sela or selb or a or b or c)
begin
   if (sela)
       q = a;
   else if (selb)
      q = b;
   else
      q = c;
end
```

其所要实现的逻辑如图 3-11 所示。

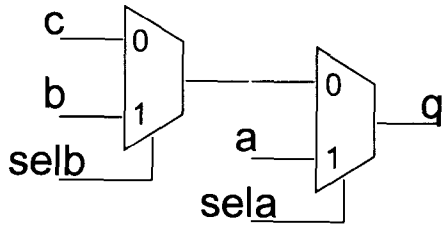

图3-11 if-else 语句

在 if-else 语句中，条件是从上到下逐条检查的。因此当满足一个条件时，就会执行其后的语句，跳过 else 后面的语句。当所有条件都不满足时，便执行最后一条 else 后面的语句。可以说 if-else 语句是有优先级顺序的。

在上例中实际上使用了 if-else 优先级编码的特点，sela 的判断优先级最高，因此在逻辑中的级数要明显少一些，参考图 3-11。如果 sela 为关键路径的话，就可以利用这样的优先级编码提高设计的性能。

在使用 if...else 语句时，尤其是当该语句被用在组合逻辑中时，需要注意不要引入 Latch 电路。来看一下如下代码：

```
always @(sel or a or b or c)
begin
   if (sel == 2'b00)
       q = a;
   else if (sel == 2'b01)
      q = b;
   else if (sel == 2'b10)
      q = c;
end
```

最后一个条件 sel==2'b11 的语句没有被写出，言下之意是，当 sel 为 2'b11 时，q 值需要保持不变。这个代码在综合时会产生锁存器，如图 3-12 所示。

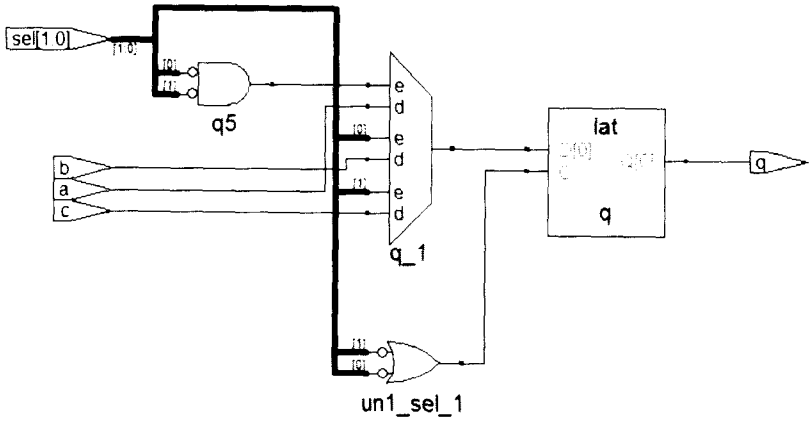

图3-12　产生锁存器

在数字同步逻辑设计中应该尽量避免产生锁存器，因为锁存器容易引起竞争冒险，同时静态时序分析工具也很难分析穿过锁存器的路径。

在下面的代码中已经明确写出当 sel 为 2'b11 时，q 值不关心，被赋值为 x。

```
always @(sel or a or b or c)
begin
   if (sel == 2'b00)
       q = a;
   else if (sel == 2'b01)
      q = b;
   else if (sel == 2'b10)
      q = c;
   else
      q = 1'bx; //当 sel 为 2'b11 时，q 值不关心
end
```

既然不关心 sel 为 2'b11 时 q 的值，那么有的综合工具就会顺手将 sel 等于 2'b11 时 q 的值也赋值为 c，这样就避免了锁存器的产生，实现的电路如图 3-13 所示，这正是我们想要的电路。

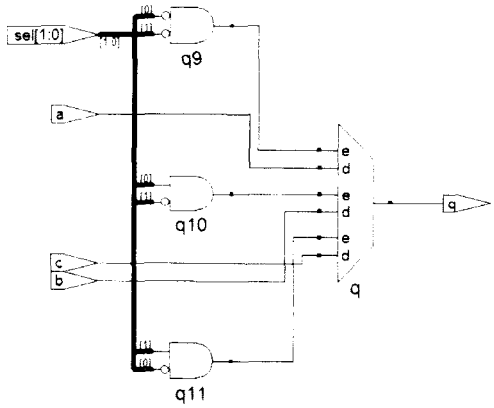

图3-13 无锁存器的组合逻辑电路

然而，在描述时序逻辑时，我们通常将利用 if 语句的隐式条件对带时钟使能的 D 触发器建模。

例如：

```
always @(posedge Clock or negedge Rst_n)
begin
    if ( ~Rst_n )
        Sum <= 0;
    else if ( En )
        Sum <= a + b;
end
```

以上语句表示在时钟正沿来临时，如果 En 为 1，则将 a+b 的值付给 Sum。言下之意是，如果 En 为 0，那么 Sum 保持原值不变。这里综合工具会把代码综合成一个带时钟使能的 D 触发器，如图 3-14 所示。

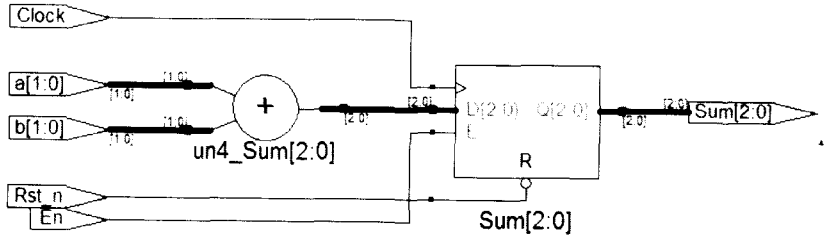

图3-14 带时钟使能的 D 触发器

其中 En 信号是 D 触发器的时钟使能端，Rst_n 是 D 触发器的异步清零信号。

三、case 语句

case 语句的功能同 if-else 类似，但是又有很大的不同。它后面也可以跟语句或语句组（begin...end 或 fork...join）。

例如：

```
always @(sel or a or b or c)
```

```
begin
   case (sel)
   2'b00: q = a;
   2'b01: q = b;
   2'b10: q = c;
   default: q = 1'bx;
   endcase
end
```

在 case 语句中，“default:”一条语句描述了所有没有明确说明的其他可能情况。比如这里的 default 就包含了 sel 为 2'b11、2'bzz 和 2'bxx 等时的情况。

以上代码将实现同图 3-13 所示一样的电路。

与 if-else 语句不同的是，在 case 语句中，所有被判断的分支条件都具有一样的优先级。

与 if-else 语句类似的是，case 语句同样需要考虑所有可能的情况，否则将会产生出不想要的锁存器，如果将代码改为：

```
always @(sel or a or b or c)
begin
   case (sel)
   2'b00: q = a;
   2'b01: q = b;
   2'b10: q = c;
   endcase
end
```

这样将会产生与图 3-12 所示一样的锁存器，这当然是设计者所不愿意看到的。

对于 case 语句，有两个派生语句：

- casez ；
- casex 。

casez 语句将分支条件中所有的 z 都看作“不关心”的值，而不看作任何逻辑值。条件中的 z 可以改写为?，例如：

```
casez (encoder)
   4'b1???  :  high_lvl = 3;
   4'b01??  :  high_lvl = 2;
   4'b001?  :  high_lvl = 1;
   4'b0001  :  high_lvl = 0;
   default  :  high_lvl = 0;
endcase
```

这里，如果 encoder 为 4'b1zzz，则 high_lvl 取值为 3。

casex 语句将分支条件中所有的 x 和 z 都看作“不关心”的值，而不看作任何逻辑值，例如：

```
casex (encoder)
```

```
    4'b1xxx  :  high_lvl = 3;
    4'b01xx  :  high_lvl = 2;
    4'b001x  :  high_lvl = 1;
    4'b0001  :  high_lvl = 0;
    default  :  high_lvl = 0;
endcase
```

这里，如果 encoder 为 4'b1xzx，则 high_lvl 取值为 3。

四、循环语句

循环语句一般用于重复的操作中。

循环语句后面可以跟语句或语句组（begin...end 或 fork...join）。

- **forever** 循环：永远执行

```
initial
begin
clk = 0;
forever  #25  clk = ~clk;
end
```

以上语句将产生一个 50 个时间单位的时钟。

- **repeat** 循环：执行固定的次数

```
if (rotate == 1)
    repeat (8)
        begin
          tmp = data[15];
          data = {data << 1, temp};
        end
```

当 rotate 为 1 时，重复对 data 数据进行 8 次左移。

- **while** 循环：当表达式为真时执行

```
initial
begin
    count = 0;
    while (count < 101)
   begin
       $display ("Count = %d", count);
       count = count + 1;
   end
end
```

在 while 语句中，只要后面的条件满足，就会持续执行该语句，直到条件不满足为止。这里，将 count 从 0 递增到 101，逐步打印出来。

- **for** 循环：从初始值开始，如果表达式为真就执行

```
integer  i;  // 为 for 循环声明索引
always @(inp or cnt)
begin
    result[7:4] = 0;
    result[3:0] = inp;
    if (cnt == 1)
   begin
    for (i = 4; i <= 7; i = i + 1)
        begin
           result[i] = result[i-4];
        end
    result[3:0] = 0;
  end
end
```

for 语句开始执行，直到 i 大于 7 时跳出循环。以上代码实现了一个 4 比特的左移器。

3.4 结构化描述

结构化描述就是说在设计中实例化已有的功能模块，这些功能模块包括门原语、用户自定义原语（UDP）和其他模块（module）。以下是结构化描述的 3 种实例类型：

- 实例化其他模块；
- 实例化门；
- 实例化 UDP。

下例是由两个半加器组成的全加器模型，其中所有模块都采用了结构化描述方式。

【例 3-2】 全加器实例，示例代码详见随书光盘中“Example-3-2”目录下的相关内容。

```
module HalfAdd (X, Y, SUM, C_out);//半加器模块
input X;
input Y;
output SUM;
output C_out;
//assign SUM = X ^ Y ;
//assign C_out = X & Y ;
xor u_xor (SUM, X, Y); //门级原语实例
and u_and (C_out, X, Y); //门级原语实例
endmodule


module FullAdd (X, Y, C_in, SUM, C_out);//全加器模块
input X;
input Y;
```

```
input C_in;
output SUM;
output C_out;

wire HalfAdd_A_SUM;
wire HalfAdd_A_COUT;
wire HalfAdd_B_COUT;

//assign C_out = HalfAdd_A_COUT | HalfAdd_B_COUT ;
or u_or (C_out, HalfAdd_A_COUT, HalfAdd_B_COUT);// 门级原语实例
HalfAdd u_HalfAdd_A (  //半加器实例 A
    .X     (X),
    .Y     (Y),
    .SUM   (HalfAdd_A_SUM),
    .C_out (HalfAdd_A_COUT) );

HalfAdd u_HalfAdd_B (  //半加器实例 B
    .X     (C_in),
    .Y     (HalfAdd_A_SUM),
    .SUM   (SUM),
    .C_out (HalfAdd_B_COUT) );
endmodule
```

这里已将例 3-1 半加器代码中的 assign 语句改为门原语的实例化。在全加器的模块中，有两个半加器模块的实例和一个 or 门原语的实例。

在以上代码中，实例化的 or 门原语是 Verilog 语言自带的电路，实例化的半加器模块则是用户自己设计的模块。

实现的电路如图 3-15 所示。

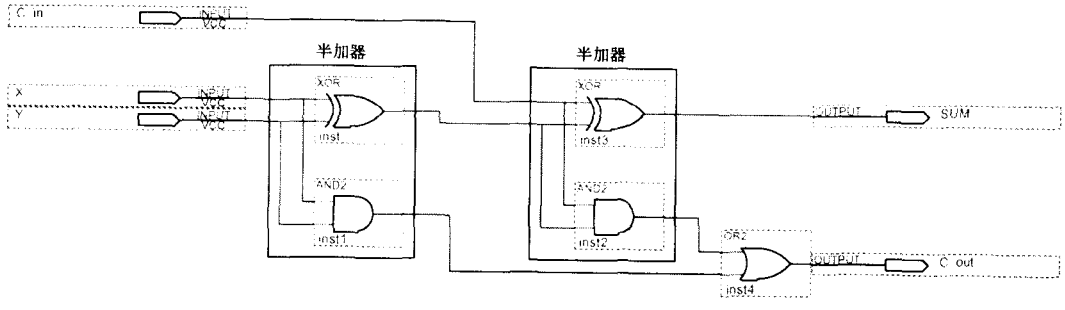

图3-15　全加器

在以上内容中，我们提到了实例化基本门和其他模块。

关于 UDP，它是用户自定义的原语，由于应用不广泛，其功能基本上可以由模块替代，因此这里不再介绍。

3.4.1　实例化模块的方法

在结构化描述中，需要将模块实例与外部信号相连接。下面谈谈模块实例的端口连接规则。

先来看看一个模块内部输入/输出/双向端口的属性。

- Input：在模块内部缺省为一个线网类型。
- Output：在模块内部是一个寄存器（在过程赋值语句中被赋值）或者线网类型。
- Inout：在模块内部缺省为一个线网类型，是双向信号，一般定义为 tri。

当这个模块被实例化时，与之相连的信号类型如下。

- 与模块 input 端口相连：可以是一个线网或者寄存器。
- 与模块 output 端口相连：一定是驱动到一个线网。
- 与模块 inout 端口相连：输入时从一个线网驱动来，输出时驱动到一个线网。

这里，初学者经常会犯这样一个错误，他们用寄存器变量驱动 inout 端口，导致编译出错。要切记，只有线网类型可以驱动 inout 端口。

图 3-16 清楚地说明了模块端口在内部和外部的类型。

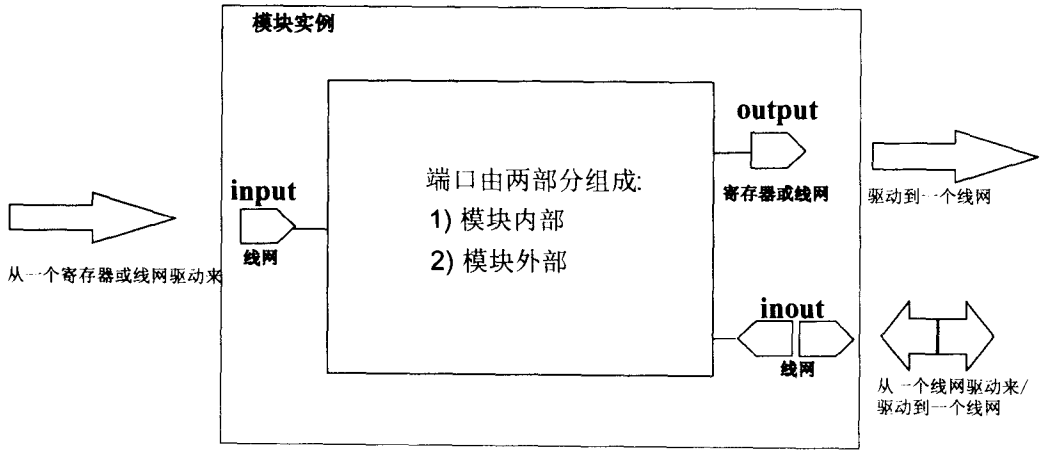

图3-16　模块实例的端口连接规则

下面举例说明模块内部和外部的端口。

```
HalfAdd u_HalfAdd_A (  //半加器实例 A
    .X     (X),
    .Y     (Y),
    .SUM   (HalfAdd_A_SUM),
    .C_out (HalfAdd_A_COUT) );
```

点“.”后面紧跟的信号是 HalfAdd 内的端口名称，而括号中的信号是上一层 FullAdd 模块中的驱动源或被驱动信号。

模块实例的端口对应方式有以下两种：

- 名称对应；
- 位置对应。

所谓名称对应是指将模块实例外部的信号直接对应于模块的端口名称。我们在实例化 HalfAdd 时就采用了名称对应的方式。

在这种端口对应方式下，端口对应的顺序可以是任意的。在没有对应外部信号的时候，可以将端口后面的括号空着，例如：

```
模块 实例名称 (
    .模块端口名称      (实例外部信号),
    .模块端口名称      (实例外部信号),
    .模块端口名称      (), //无对应信号
  . ... );
```

所谓位置对应是指在实例化模块的时候，外部的信号需要按照该模块端口声明的顺序一一对应，例如：

```
module HalfAdd (X, Y, SUM, C_out);
...
endmodule
```

我们在实例化该模块的时候可以使用：

```
HalfAdd u_HalfAdd (E_X, E_Y, E_SUM, E_C_out);
```

其中 E_X、E_Y、E_SUM 和 E_C_out 分别对应 HalfAdd 的端口 X、Y、SUM 和 C_out，严格按照了 HalfAdd 模块的端口位置顺序。在没有对应外部信号的时候，可以将位置留空，例如：

```
HalfAdd u_HalfAdd (E_X, E_Y, , E_C_out);
```

其中 E_X、E_Y 和 E_C_out 分别对应 HalfAdd 的端口 X、Y 和 C_out，模块端口 SUM 没有可对应的外部信号。

3.4.2 参数化模块

本节讨论可参数化的模块。

一、参数定义

module 中的参数一般是用作定义其中常量的工具。

以下代码定义了半加器的“与门”和“异或门”的延时分别为 2 和 4 个时间单位。

```
module half_adder (co, sum, a, b);
output co, sum;
input a, b;
parameter and_delay = 2;
parameter xor_delay = 4;
and #and_delay u1(co, a, b);
xor #xor_delay u2(sum, a, b);
endmodule
```

实际上在 Verilog 语言中，当实例化模块时，用户可以修改模块中的参数，用来实现不同的特性。这个定制过程是通过“新参数直接代入”或“参数重定义”来完成的。

Verilog 模块参数的这一特性非常有用，用户可以定义一个通用的模块，该模块具有缺省的参数值，通过改变参数可将其做成不同的实例模块。

例如可以设计一个通用的 RAM 模块，将其位宽和地址深度定义为参数。在具体使用时，如果需要用到不同的位宽和深度，则可以改变模块中的参数。

二、参数的定制

参数的用户定制方法有以下两种：

1. 通过 defparam 关键字重新定义模块中的参数；
2. 直接在实例化模块时代入参数。

有意思的是，两家最大的 PLD 供应商 Altera 和 Xilinx 的通用模块定制恰好分别采用了这两种方法。

比如在 Altera 的 Quartus II 开发环境中用 MegaWiard 工具定制一个宽度为 8bit，深度为 32bit 的单口 RAM 时，将产生以下代码：

```
module ram_w8_d32 (address, clock, data, wren, q);
input   [4:0] address;
input    clock;
input   [7:0] data;
input    wren;
output  [7:0] q;

altsyncram altsyncram_component (
            .wren_a (wren),
            .clock0 (clock),
            .address_a (address),
            .data_a (data),
            .q_a (q),
            .aclr0 (1'b0),
            .aclr1 (1'b0),
            .q_b (),
            .clocken1 (1'b1),
            .clocken0 (1'b1),
            .data_b (1'b1),
            .rden_b (1'b1),
            .address_b (1'b1),
            .wren_b (1'b0),
            .byteena_b (1'b1),
            .addressstall_a (1'b0),
            .byteena_a (1'b1),
            .addressstall_b (1'b0),
```

```
            .clock1 (1'b1));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.intended_device_family = "Stratix II",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32,//32 个字
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M4K",
        altsyncram_component.widthad_a = 5,//5 位地址
        altsyncram_component.width_a = 8,//8 位宽的数据
        altsyncram_component.width_byteena_a = 1;
    endmodule
```

其中，altsyncram 是 Altera 的块 RAM 的通用模型，可以在 Altera 的仿真文件“altera_mf.v”中找到。

altsyncram_component 是实例名称，“.”后面是参数的名称，“=”后面是被重新定义的参数值，有的是字符串，有的是整数。

defparam 关键字后面是参数重定义语句：

```
altsyncram_component.width_a = 8,//8 位宽的数据
```

其结果是将“altsyncram_component”中的参数“width_a”重新定义为 8。

使用 defparam 的方法重新定义参数时，可以根据需要重新定义部分参数，而其他参数将保留缺省值。

接下来使用 Xilinx 设计工具 ISE 中的 CORE Generator 产生一个单端口 8 位宽、32 位深的 RAM，注意观察 RAM 的参数是如何代入的。

```
module ram_w8_d32 (addr, clk, din, dout, we);
input [4 : 0] addr;
input clk;
input [7 : 0] din;
output [7 : 0] dout;
input we;
// synopsys translate_off
BLKMEMSP_V6_1 #(
    5,  // c_addr_width
    "0",    // c_default_data
    32, // c_depth
```

```
        0,  // c_enable_rlocs
        1,  // c_has_default_data
        1,  // c_has_din
        0,  // c_has_en
        0,  // c_has_limit_data_pitch
        0,  // c_has_nd
        0,  // c_has_rdy
        0,  // c_has_rfd
        0,  // c_has_sinit
        1,  // c_has_we
        18, // c_limit_data_pitch
        "mif_file_16_1",    // c_mem_init_file
        0,  // c_pipe_stages
        0,  // c_reg_inputs
        "0",    // c_sinit_value
        8,  // c_width
        0,  // c_write_mode
        "0",    // c_ybottom_addr
        1,  // c_yclk_is_rising
        1,  // c_yen_is_high
        "hierarchy1",   // c_yhierarchy
        0,  // c_ymake_bmm
        "32kx1",    // c_yprimitive_type
        1,  // c_ysinit_is_high
        "1024", // c_ytop_addr
        0,  // c_yuse_single_primitive
        1,  // c_ywe_is_high
        1)  // c_yydisable_warnings
    inst (
        .ADDR(addr),
        .CLK(clk),
        .DIN(din),
        .DOUT(dout),
        .WE(we),
        .EN(),
        .ND(),
        .RFD(),
        .RDY(),
        .SINIT());
```

```
// synopsys translate_on
endmodule
```

其中“BLKMEMSP_V6_1”是 Xilinx 的块状 RAM 的通用模型，“inst”是用户实例名称。用户需要将具体参数代入到通用的 RAM 模型中，实现不同类型的 RAM。使用 #(..., ..., ...) 语法可以实现参数的逐个替代。

使用上面这种参数直接代入法时需要注意一点，即所有的参数都要按顺序列出来，不能遗漏，也不能颠倒顺序，否则就会对应不上。

3.5 设计层次

3.5.1 系统级和行为级

Verilog 语言作为一种用户工具，提供给用户许多描述硬件的手段，如前面所述的数据流描述、行为描述（always 和 initial 语句）、结构化描述等。用户可以根据自己的需要，在不同的抽象层次上对硬件进行描述，如图 3-17 所示。

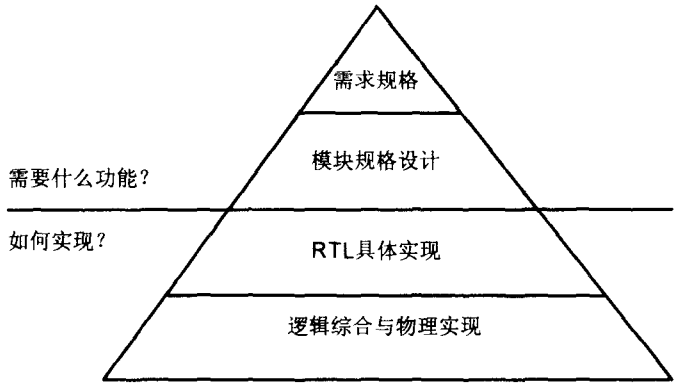

图3-17 设计层次金字塔

下面简单介绍几种不同角色的工作特点，以及他们所处的设计层次。

系统构架师：在目前业界主流的设计方法学中，系统构架师（System Architect）通常用高级语言，例如 System C，来描述一个系统的规格，仿真整个系统的功能和性能等。这种早期的设计和探索往往不涉及具体的实现细节，甚至软件和硬件的划分都没有开始。系统构架师也可以采用 Verilog 来描述系统的功能，他们往往不考虑硬件实现的细节。我们称这种设计层次为系统级或算法级。

逻辑设计工程师：他们利用前面所讲的各种 Verilog 描述手段设计 RTL 级的代码，精确到时钟周期。逻辑设计工程师的代码通过综合工具的综合，可以转换为 Verilog 的门级网表，其中所有的功能块都是由基本的门单元组成的。

物理设计工程师：他们对这些门级网表进行布局和布线，将其做成实际的芯片。

验证工程师：他们负责对设计好的电路进行验证，他们编写的代码主要用来产生激励，这些激励大部分需要抽象层次更高，以使仿真效率更高。然后在工具中对电路进行仿真，检

查响应结果。这些代码不会实现为具体硬件，有些并不需要精确到时钟周期，而只是在软件的仿真工具中运行，实现一定的功能即可，我们称这种描述层次为行为级。

这里的行为级描述不同于 3.3 节中介绍的“行为描述方式”，这里特指一种描述的抽象层次。

下面通过一个实例说明 RTL 级和行为级的区别，所要实现的状态机如图 3-18 所示。

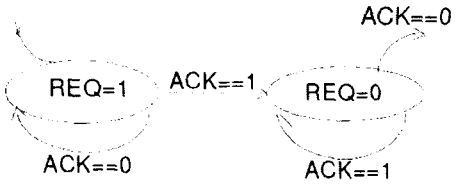

图3-18　一个简单的握手协议状态机

图中只有两个状态，当处于左边的状态时，REQ 输出 1；当处于右边的状态时，REQ 输出 0。ACK 是状态机的输入信号，决定了状态的跳转。

如果让一个 RTL 设计工程师来设计，他马上就会将逻辑功能与硬件实现的细节联系起来，很可能会使用如下代码：

```
/*****************************
RTL 描述
*****************************/
reg [1:0] Curr_sm ;
reg [1:0] Next_sm ;//定义状态寄存器
parameter ... ,
State0 = 2'b01 ,
State1 = 2'b10 ;//状态名称
//state registers
always @(posedge Clk or posedge Rst)
begin
if (Rst)
Curr_sm <= ... ;
else
Curr_sm <= Next_sm ;
end
//next state logic
always @(Curr_sm or Ack)
begin
case(Curr_sm)
...
State0 : begin
REQ = 1 ;//request
```

```
if (Ack==1)
Next_sm = State1 ;
else
Next_sm = State0 ;
end
State1 : begin
REQ = 0 ;//release
if (Ack==0)
Next_sm = ... ;
else
Next_sm = State1 ;
end
...
endcase
end
```

然而，如果让一个逻辑验证工程师来设计这个状态机，他就不会考虑硬件实现的细节，只需要在语义上满足要求即可，甚至可以不出现状态寄存器，例如：

```
/*****************************
行为描述
*****************************/
initial
begin
...
REQ = 1 ;
wait ( ACK == 1 ) ;
REQ = 0 ;
wait ( ACK == 0 ) ;
...
end
```

读者在上例中可以看出，逻辑验证工程师的设计方式十分简单，仅仅利用了 begin...end 语句组的顺序执行特性，同时利用了 wait 语句来实现状态的转移。这是典型的行为级设计风格，也是逻辑验证工程师们所追求的思维方式。

3.5.2 RTL 级

所谓寄存器传输级（RTL 级）就是在描述电路的时候，只需要关注寄存器本身，以及寄存器到寄存器之间的逻辑功能，而不用关心寄存器和组合逻辑的实现细节（具体用了多少逻辑门等）。

随着逻辑综合工具的兴起，工程师已经可以从 RTL 级进行电路设计了，而不需要用传

统的设计方法从门级电路搭起。它们的 RTL 设计代码将直接通过逻辑综合工具，综合成门级的设计网表，这些网表通常是由基本的门单元组成的。逻辑综合是 EDA 流程中的一个重要组成部分。

Verilog 设计中最常用的设计层次就是 RTL 级。在 RTL 描述时，设计者需要关注寄存器的行为，其中保存着数据；同时需要关注寄存器和寄存器之间的组合逻辑功能是否能满足功能需求和时序需求。RTL 级模型是严格精确到时钟周期的模型。

本书第 4、5、6 章将重点介绍 RTL 设计的方法和技巧。

硬件中的 RTL 级描述类似于在软件中使用 C 编程语言对处理器编程，逻辑综合工具则类似于 C 编译器。

3.5.3 门级

在 Verilog 语义中，使用一些基本的门原语可以直接描述电路的门级功能，例如：

```
module (X, Y, SUM, C_out);//半加器模块
input X;
input Y;
output SUM;
output C_out;
xor u_xor (SUM, X, Y);
and u_and (C_out, X, Y);
endmodule
```

其中直接调用了 xor 和 and 的两个 Verilog 门原语。

门级设计就是指在逻辑门一级将电路搭出来。对于大型设计来说，这种设计非常耗时，效率低下，同时容易出错。但是对于一些逻辑容量较小，性能和面积要求非常高的设计来说，采用门级设计可以满足一些特殊的需求。

门级设计类似于软件中的汇编语言设计，非常精确，但是耗时耗力。

3.5.4 晶体管级

逻辑门是由一个个的晶体管组成的。在 Verilog 语言中，有用于直接描述 NMOS 和 PMOS 的原语。

3.5.5 混合描述

Verilog 支持不同设计层次的混合描述。实际上，这些描述层次之间没有严格的界限。

3.6 实例：CRC 计算与校验电路

循环冗余编码（CRC）是在二进制通信系统中常用的差错检测方法，通过在原始数据后

加冗余校验码来检测差错，采用的冗余位越多，检测出误码的机率越大。

CRC 分为计算和校验两个部分，在信号的发送端需要计算 CRC 码，将冗余码附在数据的后面。在信号接收端则对接收到的数据重新计算 CRC 码，将其与代入的 CRC 进行比较，如果两者一致，则说明在信号传输过程中没有误码。

CRC10 是一种冗余码为 10 比特的 CRC 校验码，其多项式为：

$G(x) = x^{10} + x^9 + x^5 + x^4 + x + 1$

CRC10 的功能示意图如图 3-19 所示。

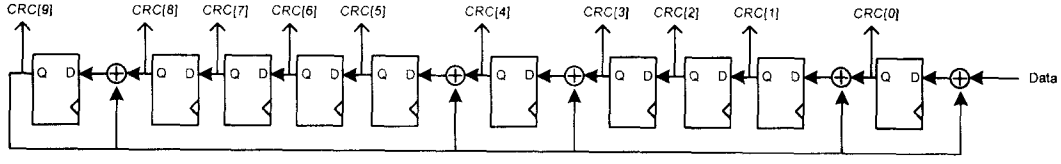

图3-19　串行 CRC10 的功能示意图

这里，我们将对 48 字节的数据进行计算和校验。48 个字节的数据为 12 组 32 比特的数据。

计算一个 48 字节的 CRC10，必须将 CRC10 的 10 比特寄存器清零，然后再将 12×32 比特的数据一位一位移入到 CRC10 的 10 比特寄存器中，当最后一比特数据移入后，CRC10 中存在的数据就是 CRC 的计算结果。

下面将根据这一原理说明如何用行为级和 RTL 级描述这个设计。

3.6.1 CRC10 校验，行为级

当我们从行为建模的角度来考虑 CRC10 的实现方式时，由于不用考虑代码的硬件实现面积和性能等参数，因此可以将 12×32 比特的数据一次性移入到 CRC10 寄存器中。

以下是行为级的描述代码：

```
...
reg [31:0] Data [0:11]; //需要计算的数据
reg [ 9:0] CRC_Reg ;
reg [ 9:0] CRC_Out ; //计算结果
integer j ; //字索引
integer i ; //比特索引
... //以下代码通过两层索引将 12×32 比特的数据移入到 CRC10 寄存器中
for ( j = 0; j <= 11; j = j + 1 )//字索引
begin
   Data_In = Data [ j ];
   for ( i = 31; i >= 0; i = i - 1 )//比特索引
   begin
      crcfb     = CRC_Reg[9] ;
      CRC_Reg[9] = CRC_Reg[8]^crcfb ;
```

```
            CRC_Reg[8] = CRC_Reg[7] ;
            CRC_Reg[7] = CRC_Reg[6] ;
            CRC_Reg[6] = CRC_Reg[5] ;
            CRC_Reg[5] = CRC_Reg[4]^crcfb ;
            CRC_Reg[4] = CRC_Reg[3]^crcfb ;
            CRC_Reg[3] = CRC_Reg[2] ;
            CRC_Reg[2] = CRC_Reg[1] ;
            CRC_Reg[1] = CRC_Reg[0]^crcfb ;
            CRC_Reg[0] = Data_In[i]^crcfb ;
        end
    end
    CRC_Out = CRC_Reg;
    ...
```

在以上的实例中，我们不会考虑代码的硬件实现结果，而是采用最方便、最简洁的方式来实现 CRC10 的计算。

3.6.2 CRC10 计算电路，RTL 级

在 3.6.1 节中，我们采用了两层索引方式，将 12×32 比特的数据同时移入到 CRC10 寄存器中，并不考虑实现该电路需要多少时钟周期。但是，如果我们需要设计的是电路，也就是设计 RTL 的代码，那么就要考虑需要多少时钟周期，寄存器和寄存器之间有多少逻辑级数等。

实现电路的性能与采用的实现工艺关系很大，这里介绍一种比较常用的 CRC10 电路 RTL 设计方法。

如果将 12×32 比特的数据在一个时钟周期内一次性移入到 CRC10 寄存器中，组合逻辑级数显然过高。因此对于 12 组 32 比特的数据来说，可以分为 12 个时钟周期将其移入到 CRC10 寄存器中，每个时钟周期处理 32 比特，这样就可以保证 CRC10 电路的性能。

实现时序如图 3-20 所示。

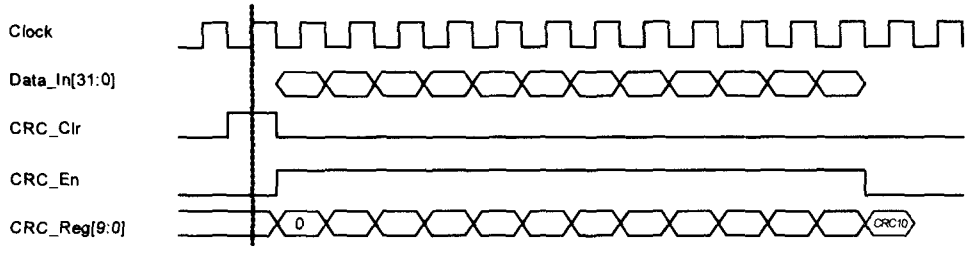

图3-20　分为 12 个时钟周期计算的 12×32 比特的数据 CRC10

其中，在 CRC_Clr 为 1 时，将 CRC 寄存器 CRC_Reg 清零，然后在 CRC_En 为 1 时，将 32 位的 Data_In 移入到 CRC 寄存器中，直到第 12 个 32 比特数据移入以后，CRC_Reg 上的数据就是 CRC10 的计算结果。

笔者按照图 3-20 中所示的时序，设计了一个 RTL 级的 CRC10 模块，代码如下。

【例 3-3】 RTL 级 CRC10 模块实例，示例代码详见随书光盘中“Example-3-3”目录下的相关内容。

```
module CRCCYC (Clock, Data_In, CRC_En, CRC_Clr, CRC_Out);
input Clock, CRC_En, CRC_Clr;
input [31:0] Data_In;
output [9:0] CRC_Out;

reg crcfb;
reg [9:0] CRC_Reg;
integer i;
always @ (posedge Clock) //单周期的 CRC10 计算代码
begin
   if (CRC_Clr)
      CRC_Reg = 0;
   else if (CRC_En)
   begin
      for (i=31;i>=0;i=i-1)
       begin
             crcfb     =CRC_Reg[9];
            CRC_Reg[9]=CRC_Reg[8]^crcfb;
            CRC_Reg[8]=CRC_Reg[7];
            CRC_Reg[7]=CRC_Reg[6];
            CRC_Reg[6]=CRC_Reg[5];
            CRC_Reg[5]=CRC_Reg[4]^crcfb;
            CRC_Reg[4]=CRC_Reg[3]^crcfb;
            CRC_Reg[3]=CRC_Reg[2];
            CRC_Reg[2]=CRC_Reg[1];
            CRC_Reg[1]=CRC_Reg[0]^crcfb;
            CRC_Reg[0]=Data_In[i]^crcfb;
      end
   end
end
endmodule
```

以上代码所实现的电路如图 3-21 所示。

通过以上分析，可以清楚地感受到行为级描述和 RTL 级描述的差别。总的说来，行为级关心的是代码的行为，RTL 级关心的是电路的实现方式和性能。时钟周期和 D 触发器是 RTL 代码的两个设计重点。

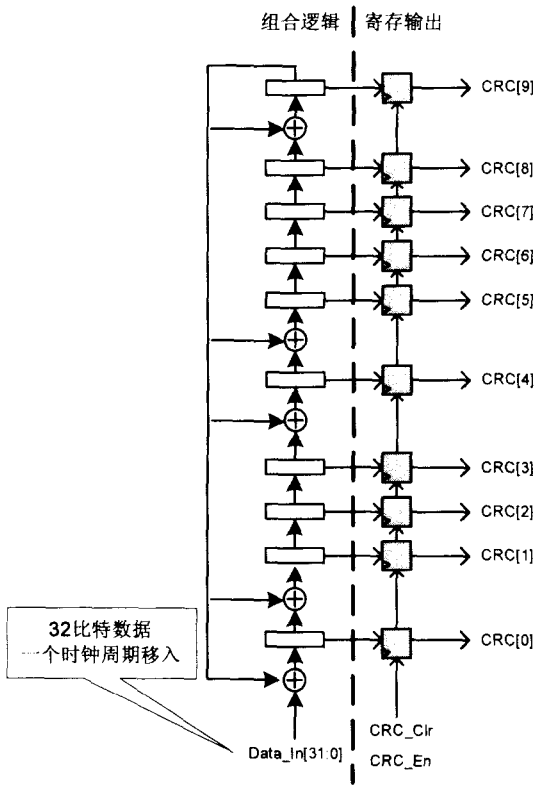

图3-21　实现结果示意图

3.7 小结

本章重点介绍了 Verilog 的 3 种描述方式，以及 Verilog 可以应用的设计层次。

3.8 问题与思考

1. 数据流描述方式采用什么语句？
2. 驱动和赋值的概念有什么区别？
3. 行为描述主要使用哪两种语言？
4. 什么是过程赋值语句？
5. 模块实例的端口对应方式有哪些？
6. 简述 RTL 级设计和行为级设计的区别？

第4章 RTL概念与RTL级建模

Verilog HDL的基本功能之一是描述可综合的硬件电路。如何合理使用Verilog HDL描述高性能的可综合电路是本书的主旨之一，也是第4、5、6章所要重点讨论的问题。本书第4章安排了一些常用的设计实例，使读者对RTL级建模有一个感性的认识；第5章重点讨论RTL级编码风格以及各种设计原则；第6章讨论RTL级设计中一个非常重要的内容——有限状态机（FSM）的描述技巧。读者可将这3章看作一个整体进行学习。

本章主要内容如下：

- RTL与综合的概念；
- RTL级设计的基本要素和步骤；
- 常用的RTL级建模；
- 实例。

4.1 RTL与综合的概念

通过前面章节的学习可知，HDL语言是分层次的，最常用的层次概念有系统级（System Level）、功能模块级（Function Model Level）、行为级（Behavior Level）、寄存器传输级（RTL，Register Transfer Level）和门级（Gate Level）等，其中寄存器传输级（RTL，Register Transfer Level）指不关注寄存器和组合逻辑的细节，通过描述寄存器到寄存器之间的逻辑功能描述电路的HDL层次。RTL级是比门级更高的抽象层次，使用RTL级语言描述硬件电路一般比用门级描述电路简单、高效得多。

RTL级语言最重要的特性就是RTL级描述是可综合的描述。所谓综合（Synthesize）是指将HDL语言、原理图等设计输入翻译成由与、或、非门等基本逻辑单元组成的门级连接，并根据设计目标和要求优化所生成的逻辑连接，输出门级网表文件。而RTL级综合则是指将RTL级源代码翻译并优化为门级网表。

随着综合工具的不断智能化，使用RTL级语言描述硬件电路越来越方便。目前在可编程逻辑器件（PLD，主要指FPGA和CPLD）设计领域，最重要的代码设计层次就是RTL级。

4.2 RTL级设计的基本要素和步骤

典型的RTL级设计包含以下3个部分：

- **时钟域描述**

  描述设计中使用的所有时钟、时钟之间的主从与派生关系以及时钟域之间

的转换。

- **时序逻辑描述（寄存器描述）**

  根据时钟沿的变换，描述寄存器之间的数据传输方式。

- **组合逻辑描述**

  描述电平敏感信号的逻辑组合方式和逻辑功能。

在 RTL 描述中，时序逻辑、组合逻辑的连接关系和拓扑结构决定了设计的性能，如何调整时序逻辑、组合逻辑的连接关系和拓扑结构以达到最佳性能，是后面所要讨论的各种编码风格（Coding Style）的核心。

设计 RTL 级代码的顺序有很多种，笔者推荐的设计步骤如下：

- **功能定义与模块划分**

  根据系统功能的定义和模块划分准则（详见第 5 章 5.4.2 小节）划分各个功能模块。

- **定义所有模块的接口**

  定义每个模块的接口，完成每个模块的信号列表，这种思路与 Modular Design（模块化设计方法）一致，便于模块的重用、调试和修改。

- **设计时钟域**

  根据设计的时钟复杂程度定义时钟之间的派生关系，分析设计中有哪些时钟域，是否存在异步时钟域之间的数据交换，对于 PLD 设计，还需确认全局时钟和是否使用 PLL/DLL 完成时钟的分频、倍频、移相等功能，哪些时钟使用全局时钟资源布线，哪些时钟使用第二全局时钟资源布线。全局时钟资源的特点是几乎没有 Clock Skew（时钟偏斜），有一定的 Clock Delay（时钟延时），驱动能力最强；第二全局时钟资源的特点是有较小的 Clock Skew 和 Clock Delay，时钟驱动能力较强。

- **考虑设计的关键路径**

  所谓关键路径是指设计中时序要求最难以满足的路径。设计的时序要求主要体现在频率、建立时间、保持时间等时序指标上。在设计初期，设计者可以根据系统的频率要求，粗略地分析出设计的时序难点（如最高频率的路径、计数器的最低 bit 和包含复杂组合逻辑的时序路径等），通过一些时序优化手段（如 Pipeline、Retiming 和逻辑复制等，这些时序优化手段将在第 5 章 5.7 节中详细介绍）从代码上缓解设计的时序压力，这种方法比单纯依靠综合和布局布线工具的自动优化功能有效得多。

- **顶层设计**

  常用的设计方法有两种，一种是自顶而下的设计方法，即先描述设计的顶层，然后描述设计的每个子模块；另一种是由底向上的设计方法，即首先描述设计的子模块，最后定义设计的顶层。RTL 设计推荐使用自顶而下的设计方法，因为这种设计方法与模块规划的顺序一致，而且更利于进行 Modular Design（模块化设计方法），并行开展设计工作，提高模块的重用率。

- **FSM 设计**

  有限状态机（FSM）是逻辑设计中最重要的内容之一。

- **时序逻辑设计**

  首先根据时钟域规划好寄存器组，然后描述各个寄存器组之间的数据传输方式。

- **组合逻辑设计**

  一般来说，大段的组合逻辑最好与时序逻辑分开描述，这样更利于时序约束和时序分析，使综合器和布局布线器得到更好的优化。

4.3 常用的 RTL 级建模

4.3.1 阻塞赋值、非阻塞赋值和连续赋值

阻塞赋值和非阻塞赋值的概念，在前面已经介绍过了，为了避免因不当使用阻塞赋值和非阻塞赋值而造成的种种歧义和错误，笔者推荐：

- 对于时序逻辑，即 always 模块的敏感表为沿敏感信号（多为时钟或复位的正沿或负沿），统一使用非阻塞赋值“<=”。

【例 4-1】 时序逻辑中使用非阻塞赋值，代码参见随书光盘中“Example-4-1\source”目录下的相关内容。

```
reg [3:0] cnt_out;
always @ (posedge clock)
        cnt_out <= cnt_out + 1;
```

- 对于 always 模块的敏感表为电平敏感信号的组合逻辑，统一使用阻塞赋值“=”。

【例 4-2】 请读者注意，此例中“cnt_out_plus”虽然被指定为 reg 型，但是实现时却是纯组合逻辑，代码参见随书光盘中“Example-4-1\source”目录下的相关内容。

```
reg [3:0] cnt_out_plus;
always @ (cnt_out)
        cnt_out_plus = cnt_out + 1;
```

- 对于 assign 关键字描述的组合逻辑（通常称之为连续赋值语句），统一使用“=”，变量被定义为 wire 型信号。

【例 4-3】 本例中“cnt_out_plus”被定义为 wire 型信号，代码参见随书光盘中“Example-4-1\source”目录下的相关内容。

```
wire [3:0] cnt_out_plus;
assign  cnt_out_plus = cnt_out + 1;
```

例 4-2 和例 4-3 中之所以不采用类似例 4-1 中的“cnt_out <= cnt_out + 1”，是因为这样会产生组合逻辑环。组合逻辑环是同步时序逻辑设计中要尽量避免的设计方式，它会使得时序路径无法被工具所分析，不同芯片的延时不同，会造成逻辑功能不稳定。有些已经完成很久的设计，在换了芯片批次后，逻辑功能不正确，大多数都是由组合逻辑环造成的。

4.3.2 寄存器电路建模

寄存器和组合逻辑是数字逻辑电路中的两大基本要素。寄存器一般和同步时序逻辑关联，其特点为仅当时钟的沿（上升沿或下降沿）到达时，才有可能发生输出的改变。实现的目标不同，寄存器的建模结构也略有不同，需要注意以下几点。

- 寄存器信号声明：寄存器被定义为 reg 型。但是请读者注意，这个命题的反命题不一定成立。某些信号虽然被定义为 reg 型，但是最终综合实现的结果并不是寄存器，如例 4-2 中的“cnt_out_plus”虽然被指定为 reg 型，但是实现时却是纯组合逻辑。只有当信号被定义为 reg 型，且处理该信号的 always 敏感表为 posedge 或 negedge 沿敏感时，该信号才会被实现为寄存器。
- 时钟输入：在每个时钟的正沿或负沿对数据进行处理。时钟的正沿有效还是负沿有效，是由 always 敏感表中的 posedge 或 negedge 决定的。

【例 4-4】 指定寄存器的触发沿是时钟的下降沿，代码参见随书光盘中“Example-4-4\source”目录下的相关内容。

```
reg [3:0] cnt_out;
always @ (negedge clock)
        cnt_out <= cnt_out + 1;
```

- 异步复位/置位：绝大多数目标器件的寄存器模型都包含异步复位/置位端。所谓异步复位/置位，是指无论时钟沿是否有效，当复位/置位信号的有效沿到达时，复位/置位会立即发挥功能。指定异步复位/置位时，只需在 always 的敏感表中加入复位/置位信号的有效沿即可。下面描述的异步复位电路是最常用的寄存器复位形式之一。

【例 4-5】 设异步复位信号“reset_”是低有效信号（即下降沿开始复位）。用户可以在 begin/end 结构之间填入复杂的用户逻辑，代码参见随书光盘中“Example-4-4\source”目录下的相关内容。

```
reg [3:0] cnt_reg;
always @ (posedge clock or negedge reset_)
  if (!reset_)
    cnt_reg <= 4'b0000;
  else
   begin
     ...
   end
```

- 同步复位/置位：任何寄存器都可以实现同步复位/置位功能。指定同步复位/置

位时， always 的敏感表中仅有时钟沿信号，当同步复位/置位信号发生变化时，同步复位/置位并不立即发生，仅当时钟沿采到同步复位/置位的有效电平时，才会在时钟沿到达时刻进行复位/置位操作。

【例 4-6】 常用的同步复位电路，设同步复位信号“reset_”为低电平有效信号，代码参见随书光盘中“Example-4-4\source”目录下的相关内容。

```
reg [3:0] cnt_reg;
always @ (posedge clock)
  if (!reset_)
    cnt_reg <= 4'b0000;
  else
    begin
      ...
    end
```

同步复位和异步复位的优缺点将在本章 4.3.9 小节中详细介绍。

- 同时使用时钟上升沿和下降沿的问题：有时因为数据采样或调整数据相位等需求，设计者会在一个 always 的敏感表中同时使用时钟的 posedge 和 negedge，或者在两个 always 的敏感表中分别使用时钟的 posedge 和 negedge，对某些寄存器电路进行操作（在这两种描述下，当时钟上升沿或时钟下降沿到达时，该寄存器电路都会做相应的操作。这个双沿电路往往可以等同于使用了原时钟的倍频时钟的单沿操作电路）。对于 PLD 设计而言，不推荐同时使用时钟的上升沿、下降沿，因为 PLD 内嵌的 PLL/DLL 和一些时钟电路往往只能对时钟的一个沿保证非常好的指标，而另一个沿的抖动、偏斜、斜率等指标不见得非常优化，有时同时使用时钟的正、负沿会因时钟的抖动、偏斜、占空比、斜率等问题而造成一定的性能恶化。因此笔者推荐的做法是，将原时钟通过 PLL/DLL 倍频，然后使用倍频时钟的单沿（如上升沿）进行操作。

【例 4-7】 某电路使用 50MHz 的时钟双沿操作相当于使用同相位 100MHz 的倍频时钟操作，代码参见随书光盘中“Example-4-7 \source”目录下的相关内容。

```
reg   [3:0] cnt_temp1, cnt_temp2;
wire  [3:0] cnt1;
always @ (posedge clk_50M or negedge rst_ )
  if (!rst_)
    cnt_temp1 <= 4'b0000;
  else
    cnt_temp1 <= cnt_temp2 + 1;
always @ (negedge clk_50M or negedge rst_ )
  if (!rst_)
```

```
        cnt_temp2 <= 4'b0000;
    else
        cnt_temp2 <= cnt_temp1 + 1;
assign cnt1 = (clk_50M)? cnt_temp2 : cnt_temp1;
```

这里相当于使用与 50MHz 时钟 clk_50M 同相位的 2 倍频时钟 clk_100M 进行累加计算，等效描述如下：

```
reg    [3:0] cnt2;
always @ (posedge clk_100M or negedge rst_ )
  if (!rst_)
     cnt2 <= 4'b0000;
  else
     cnt2 <= cnt2 + 1;
```

以上两段代码综合后所对应的电路结构如图 4-1 所示，RTL 仿真波形如图 4-2 所示。

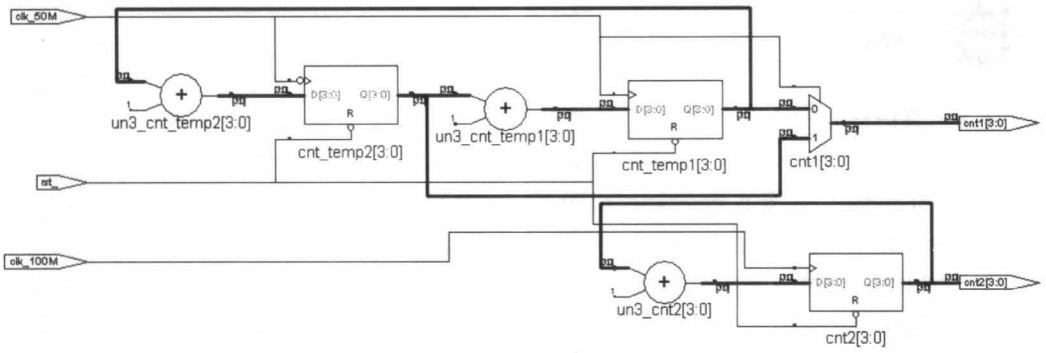

图4-1　某双沿加法电路与等价倍频时钟单沿加法电路结构示意图

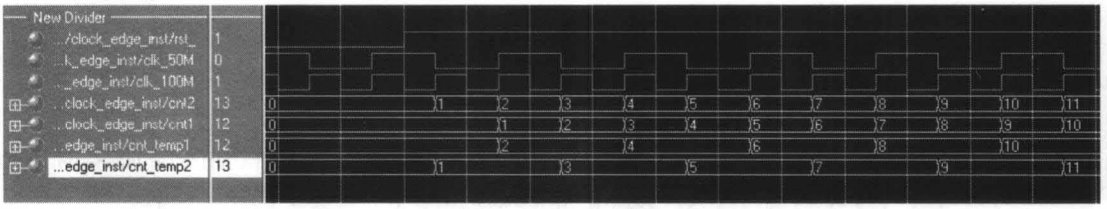

图4-2　某双沿加法电路与等价倍频时钟单沿加法仿真波形图

通过这个例子说明某些使用双沿操作的电路等价于使用倍频时钟的单沿电路，也就是说，这些电路使用了时钟的双沿触发寄存器，则相当于提高了设计频率，提高了设计要求的时序难度。对于这些电路，在附加时序约束和进行时序分析时一定要考虑充分。

4.3.3　组合逻辑建模

组合逻辑是逻辑电路设计中另一个重要的组成部分。组合逻辑的特点是输出的变化仅仅和输入的电平相关，而与时钟沿无关。常用的 RTL 级组合逻辑模有两种，第一种是 always 模块的敏感表为电平敏感信号的电路；第二种是 assign 等关键字描述的组合逻辑电路。

- **always 模块的敏感表为电平敏感信号的组合逻辑电路**

这种形式的组合逻辑电路应用非常广泛，如果不考虑代码的复杂性，几乎任何组合逻辑电路都可以用这种方式建模。always 模块的敏感表为所有判定条件和输入信号，请读者在使用这种结构描述组合逻辑时，一定要将敏感表写完整。在 always 模块中可以使用 if...else...、case、for 循环等各种 RTL 关键字结构。下面以某组合逻辑译码电路为例，说明这种组合逻辑建模方式。如前所述，在 always 模块中推荐使用阻塞赋值“=”，虽然信号被定义为 reg 型，但是最终综合实现的结果并不是寄存器，而是组合逻辑，将信号定义为 reg 型只是为了满足语法要求而已。

【例 4-8】 某组合逻辑译码电路，代码参见随书光盘中“Example-4-8 \source”目录下的相关内容。

```
reg    cs1, cs2, cs3, cs4;
   always @ (CS or addr)
     if (CS)
       {cs1, cs2, cs3, cs4} = 4'b 1111;
     else
       begin
         case (addr[7:6])
           chip1_decode: {cs1, cs2, cs3, cs4} = 4'b 0111;
           chip2_decode: {cs1, cs2, cs3, cs4} = 4'b 1011;
           chip3_decode: {cs1, cs2, cs3, cs4} = 4'b 1101;
           chip4_decode: {cs1, cs2, cs3, cs4} = 4'b 1110;
         endcase
       end
```

图 4-3 所示为该例对应的电路结构示意图。

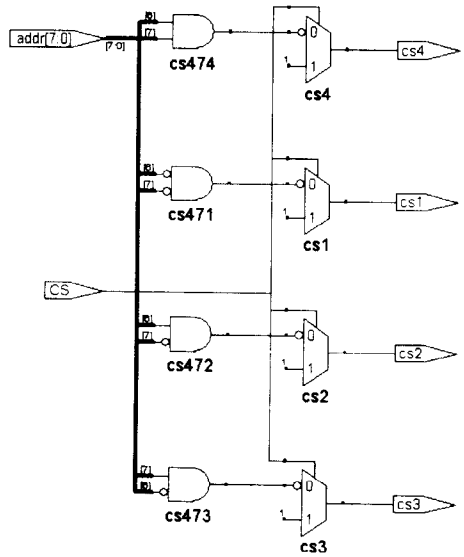

图4-3 本例组合逻辑译码电路结构示意图

- **assign 等语句描述的组合逻辑电路**

这种形式的组合逻辑电路适用于描述那些相对简单的组合逻辑，信号一般被定义为 wire 型，常用的 assign 结构除了直接赋值逻辑表达式外，还可以使用？语句。

【例 4-9】　使用 assign 语句描述该组合逻辑译码电路，代码参见随书光盘中“Example-4-8 \source”目录下的相关内容。

```
wire    cs1, cs2, cs3, cs4;
assign cs1 = (!CS && (addr[7:6] == chip1_decode))? 0 : 1 ;
assign cs2 = (!CS && (addr[7:6] == chip2_decode))? 0 : 1 ;
assign cs3 = (!CS && (addr[7:6] == chip3_decode))? 0 : 1 ;
assign cs4 = (!CS && (addr[7:6] == chip4_decode))? 0 : 1 ;
```

该描述方式实现的电路参见图 4-3。

从上例可以看出，简单的组合逻辑使用 assign 和？语句描述比较清晰，但是如果所描述的组合逻辑过于复杂，则需要很多条 assign 语句或者嵌套？语句来描述，不易解读，此时推荐使用第一种组合逻辑建模方法。

4.3.4　双向端口与三态信号建模

前面谈到的双向总线（既做输入又做输出的总线）应该在顶层模块中实例化三态信号，而不能在顶层以外的其他子层次中实例化三态信号。某些早期的 EDA 软件和器件支持在子模块中定义双向总线，实例化三态信号。其实从理论上讲，任何子模块中定义的三态信号都可以迁移到顶层中。实际上，很多综合工具也默认将子模块中定义的三态信号综合为选择器，并将子模块中三态信号的实例化迁移到顶层模块中去。

为了避免仿真和综合实现的结果不一致，并且为了便于维护，笔者强烈建议仅在顶层定义双向总线和实例化的三态信号，禁止在除顶层以外的其他层次赋值高阻态“Z”，在顶层将双向信号分为输入信号和输出信号两种类型，然后根据需要分别传递到不同的子模块中。这样做的另一个好处在于便于描述仿真激励。

【例 4-10】　几种典型的双向信号和三态信号的描述方法，代码参见随书光盘中“Example-4-10”目录下的相关内容。

本例在顶层将一个双向总线分成输入和输出两条总线，然后分别将其引入子模块。参见随书光盘中“Example-4-10 \bibus”目录下的 bibus.v 程序，将双向总线 data_bus 分为输入总线 data_in 和输出总线 data_out，然后分别使用。

```
inout [7:0] data_bus;
wire [7:0] data_in, data_out;
assign data_in = data_bus;
assign data_bus = (sel) ? data_out : 8'bZ;
```

简单的三态信号用 assign 语法描述，如 bibus.v 代码，描述三态总线如下：

```
assign data_bus = (sel) ? data_out : 8'bZ;
```

如果三态总线的使能关系比较复杂，不是单一信号，此时可以使用嵌套的问号表达式，

或者使用 case 语句进行描述（当然也可以用 if...else 结构）。读者可参考随书光盘中“Example-4-10 \complex_bibus”目录下的 complex_bibus.v 和 complex_bibus2.v 两个程序。该例中的双向总线 data_bus 的输出由 3 个使能信号 sel1, sel2, sel3 共同确定，在 complex_bibus2.v 中，使用嵌套的问号表达式描述了该三态总线的使能选择。

```
inout [7:0] data_bus;
assign data_bus = (sel1)? decode_out : ((sel2)? cnt_out : ((sel3)?
8'b11111111: 8'bZZZZZZZZ));
```

笔者认为，如果使能情况较为复杂，则不宜使用 complex_bibus2.v 中嵌套的问号表达式。更清晰的描述方法是使用 complex_bibus.v 例子中的 case 语句，通过 case 语句可以清晰地罗列出每种使能组合情况下的输出情况。这种描述需要用到组合逻辑中的 always 模块，此时需要引入中间变量 data_out，并将其定义为 reg 型，相关代码如下：

```
inout [7:0] data_bus;
wire [7:0] data_in;
reg  [7:0] data_out; //use reg type, but not registers
wire [7:0] decode_out;
wire [7:0] cnt_out;
always @ (decode_out or cnt_out or sel1 or sel2 or sel3)
   begin
      case ({sel1, sel2, sel3})
        3'b100:  data_out = decode_out;
        3'b010:  data_out = cnt_out;
        3'b001:  data_out = 8'b11111111;
        default: data_out = 8'bZZZZZZZZ;
      endcase
   end
assign data_bus = data_out;
```

读者注意，虽然输出的中间变量 data_out 被定义为 reg 型，但是在物理实现时它并不是寄存器，而是纯组合逻辑。引入这个变量的原因是 inout 类型的信号只能被定义为 wire 或 tri 型，不能在组合逻辑的 always 模块中直接被赋值。

4.3.5 Mux 建模

Mux 也是一种组合逻辑电路，它的常用建模方式也有两种：对于简单的 Mux，可以直接用 assign 和？表达式建模；对于复杂的 Mux，则需要使用 always 和 if...else..., case 等条件判断语句建模。

- 简单的 Mux 用？表达式建模，信号被定义为 wire 型，使用？表达式的判断条件描述 Mux 选择端的逻辑关系。

【例 4-11】 使用？表达式描述一个 2 选 1 的 Mux，代码参见随书光盘中“Example-4-11”目录下的相关内容。

```
wire mux_out;
assign mux_out = (en)? a : b; //当 en 为真时选择 a，否则选择 b。
```

- 复杂的 Mux 用 case 或嵌套的 if...else 建模，信号被定义为 reg 型，case 或 if...else 的每个条件分支均分别对应 Mux 的某路选择输出。

【例 4-12】 使用 case 描述一个 4 选 1 的 Mux，代码参见随书光盘中“Example-4-11”目录下的相关内容。

```
reg mux_out;
 always @ (en or a or b or c or d)
     case(en)
       2'b00:  mux_out = a;
       2'b01:  mux_out = b;
       2'b10:  mux_out = c;
       2'b11:  mux_out = d;
     endcase
```

4.3.6 存储器建模

逻辑电路设计会经常使用一些单口 RAM、双口 RAM 和 ROM 等类型的存储器。Verilog 语法中基本的存储单元定义格式如下。

```
reg [datawidth] MemoryName [addresswidth];
```

例如定义一个数据位宽为 8bit，地址为 63 位的 RAM8x64，则可定义为：

```
reg [7 : 0] RAM8x64 [0 : 63];
```

在使用存储单元时，不能直接引用存储器某地址的某比特位值，如想对地址为“32”的第 2bit 和高 2bit 的值进行操作，则下面这两种描述都是错误的。

```
RAM8x64 [32] [2]
RAM8x64 [32] [6:7]
```

正确的操作方法是，先将存储单元赋值给某个寄存器，然后在对该寄存器的某位进行相关操作，如下例所示。

【例 4-13】 一个简单的 8bit 位宽、64 位地址 RAM 的读写电路，读的时候，先将“RAM8x64”某地址的数据读到“mem_data”寄存器中，然后即可对寄存器的任意 bit 位进行相关操作。代码参见随书光盘中“Example-4-13\ram_basic”目录下的相关内容。

```
reg [7:0] RAM8x64 [0:63];
reg [7:0] mem_data;
     always @ (posedge clk)
       if (WR && CS) //WRite
         RAM8x64 [addr] <= data_in [7:0];
       else if (~WR && CS ) // read
         mem_data <= RAM8x64 [addr];
```

上面讲解的仅仅是 Verilog 语法建模存储单元的一般方法。而对于 PLD 设计而言，由于几乎所有的 FPGA 都内嵌有 RAM 资源，所以并不推荐使用 Verilog 直接建模 RAM。FPGA 内嵌的 RAM 资源大致分为两类：块 RAM（Block RAM）资源和分布式 RAM 资源（Distributed RAM，是一种基于特殊底层逻辑单元，通过查找表和触发器实现的 RAM 结构）。在 PLD 中使用存储结构的基本方法如下。

- 第一种方法：通过器件商的开发平台中内嵌的 IP 生成器，在图形化界面中直接选择存储器类型（如双口 RAM、单口 RAM、ROM 和分布式 RAM 等），配置存储器参数，生产相应 IP，然后在用户逻辑中直接调用该 IP。这种设计方法是 PLD 设计中推荐的方法，因为器件商最了解 PLD 的底层硬件结构，通过 IP 生成器，可以自动地选择使用 PLD 内嵌的 RAM 资源，并生成存储器的粘合逻辑（glue logic），方便、高效、可靠。
- 第二种方法：直接根据上面的描述用 Verilog 语言建模存储器，由综合器根据代码描述类推并优化存储器结构，调用器件内嵌的硬件存储器资源。这种方法有两个问题，一是要清晰合理地在代码中描述存储器，有一定的设计难度；二是最终实现的结果在很大的程度上取决于综合器的类推算法，有一定的不确定性。这种方法经常用在两个场合，一是 PLD 本身没有块 RAM 或分布式 RAM 等专用存储单元（如 CPLD 等）；二是用户非常熟悉综合器的类推算法，并能通过综合器的相关约束属性，指定所需使用的底层硬件 RAM 资源。

4.3.7 简单的时钟分频电路

时钟电路是 PLD 设计的核心。第 5 章将介绍同步时序电路的相关知识，并介绍一些常用时钟的电路设计。

对于 PLD 设计而言，由于大多数 PLD（特别是 FPGA）都内嵌有专用 PLL/DLL 模块，通过这些内嵌的 PLL 或 DLL，设计者可以实现灵活的分频/倍频（一般可实现小数分频倍频）、移相等调整与运算。所以这类 PLD 设计中，时钟电路的设计方法如上节一样，都推荐使用器件商开发平台中内嵌的 IP 生成器，在图形化界面中直接配置 PLL/DLL 的参数，生产相应的 IP，然后在用户逻辑中直接调用该 IP。

这里介绍的一般时钟分频电路建模方法，适用于没有上述内嵌 PLL/DLL 时钟电路（如 CPLD、ASIC 设计等）中，或内嵌 PLL/DLL 资源不能满足所需时钟关系时的一些处理中。

一般来说，PLD 中的主要时钟处理为分频和移相。偶数分频十分简单，只需用高速时钟做一个同步计数器，然后在相应的 bit 位抽头即可。奇数分频电路相对复杂一些。移相的基本方法是通过高速时钟调整相位，或者通过时钟反向调整相位。

【例 4-14】 将一个 200kHz 时钟做 2 分频、4 分频、8 分频，要求分频后的 3 个时钟同相，而且与源时钟近似同相。在这个设计中，因为输入时钟速率很低，仅有 200kHz，而一般 PLD 内嵌的 PLL 的输入频率下限都在 MHz 级，所以无法使用 PLL 完成分频与相位调整要求。另外对于低速时钟的分频，使用计数器既能满足时序要求，也比较节约器件资源，代码参见随书光盘中“Example-4-14\clk_div_phase”目录下的相关内容。

```
reg [2:0] cnt;
always @ (posedge clk_200K or negedge rst)
  if (!rst)
    cnt <= 3'b000;
  else
    cnt <= cnt + 1;
assign clk_100K = ~cnt [0];
assign clk_50K  = ~cnt [1];
assign clk_25K  = ~cnt [2];
```

这个设计的难点在于如何调整所有时钟的相位关系。本例巧妙地通过对计数器每个 bit 的反向处理，完成了所有分频后时钟的相位调整，保证了 3 个分频后时钟的相位严格同相。这 3 个派生时钟与源时钟相比有一个非常小的相位差，这个相位差是由寄存器的固有 Tco（Clock to Output 延时）和计数器累加的组合逻辑造成的。一般来说在 PLD 中寄存器固有 Tco 的典型值为 1～2ns，而简单加法运算的组合逻辑门延时也约为 ns 级，这两个延时的总和与时钟周期相比微乎其微。如果忽略这个 ns 级的延时，则可以认为通过每个分频时钟的反向，使 3 个分频时钟与源时钟同相，也就是说这 4 个时钟拥有共同的上升沿。

【例 4-15】 对源时钟做 3 分频，要求 3 分频时钟占空比为 50%，代码参见随书光盘中“Example-4-14\clk_3div”目录下的相关内容。

```
reg[1:0] state;
reg clk1;
  always @(posedge clk or negedge reset)
    if(!reset)
      state<=2'b00;
    else
      case(state)
        2'b00:state<=2'b01;
        2'b01:state<=2'b11;
        2'b11:state<=2'b00;
        default:state<=2'b00;
      endcase
  always @(negedge clk or negedge reset)
    if(!reset)
      clk1<=1'b0;
    else
      clk1<=state[0];
  assign clk_out=state[0]&clk1;
```

3 分频、5 分频等奇数分频时钟可以使用 case 结构或简单的状态机（FSM）描述，设计难点在于如何通过组合逻辑调整分频时钟的占空比。

4.3.8 串并转换建模

数据流串并转换的实现方法多种多样，根据数据的排序和数量的要求，可以选用移位寄存器、RAM 等来实现。对于数据量比较小的设计来说，可以使用移位寄存器完成串并转换；对于排列顺序有规定的串并转换，可以用 case 语句判断实现；对于复杂的串并转换，还可以用状态机实现。

【例 4-16】 简单的串行到并行转换方法，数据排列顺序是高位在前，代码参见随书光盘中“Example-4-16”目录下的相关内容。

```
reg    [7:0] pal_out;
always @ (posedge clk or negedge rst)
  if (!rst)
    pal_out <= 8'b0;
  else
    pal_out <= {pal_out,srl_in};
```

4.3.9 同步复位和异步复位

复位电路是每个数字逻辑电路中最重要的组成部分之一。复位电路的工作目的有两个方面：第一是仿真的时候使电路进入初始状态或者其他预知状态；第二是对于综合实现的真实电路，通过复位使电路进入初始状态或者其他预知状态。一般来说，逻辑电路的任何一个寄存器、存储器结构和其他逻辑单元都必须要附加复位逻辑电路，以保证电路能够从错误状态中恢复，可靠地工作。

常用的复位信号为低电平有效信号，在应用时外部引脚接上拉电阻，这样能增加复位电路的抗干扰性能。

复位方式大致分为两类，即同步复位和异步复位。这两种复位方式各有优缺点，其应用场合也各不相同。

一、同步复位电路建模

所谓同步复位是指当复位信号发生变化时，并不立即生效，只有当有效时钟沿采样到已变化的复位信号后，才对所有寄存器复位。同步复位的应用要点如下。

- 指定同步复位时，always 的敏感表中仅有时钟沿信号，仅仅当时钟沿采到同步复位的有效电平时，才会在时钟沿到达时刻进行复位操作。如果目标器件或可用库中的触发器本身包含同步复位端口，则在实现同步复位电路时可以直接调用同步复位端。然而很多目标器件（如 PLD）和 ASIC 库的触发器本身并不包含同步复位端口，这样复位信号与输入信号组成某种组合逻辑（比如复位低电平有效，只需复位与输入信号两者相与即可），然后将其输入到寄存器的输入端。为了提高复位电路的优先级，一般在电路描述时使用带有优先级的 if...else 结构，复位电路在第一个 if 下描述，其他电路在 else 或 else...if 分支中描述。

【例 4-17】　同步复位电路建模，代码参见随书光盘中“Example-4-17\syn_rst”目录下的相关内容。

```
always @ (posedge clk)
  if (!rst_)
    begin
      ...
    end
  else
    begin
      ...
    end
```

- 很多目标器件（如 FPGA 和 CPLD）和 ASIC 库的触发器本身并不包含同步复位端口，这时同步复位会被实现为如图 4-4 所示的结构。

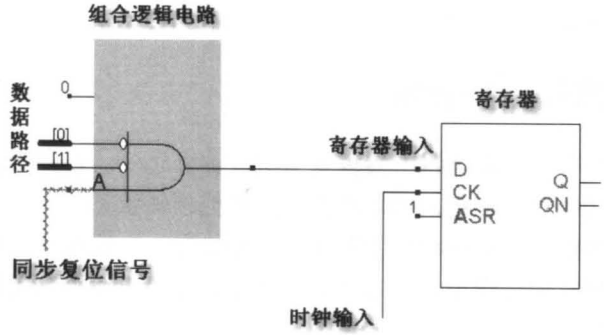

图4-4　同步复位实现结构示意图

同步复位的优点如下：

- 同步复位利于基于周期机制的仿真器进行仿真；
- 使用同步复位可以设计 100％的同步时序电路，有利于时序分析，其综合结果的频率往往较高；
- 同步复位仅在时钟的有效沿生效，可以有效地避免因复位电路毛刺造成的亚稳态和错误。同步复位在进行复位和释放复位信号时，都是仅当时钟沿采到复位信号电平变化时才进行相关操作，如果复位信号树的组合逻辑出现了某种毛刺，此时时钟沿采样到毛刺的概率非常低，这样通过时钟沿采样，可以十分有效地过滤复位电路组合逻辑产生的毛刺，增强了电路稳定性。

同步时序电路的概念可查阅本书 5.2.1 小节中的内容；亚稳态的概念可查阅本书 5.2.2 小节中的内容。

同步复位的缺点如下：

- 很多目标器件（如 FPGA 和 CPLD）和 ASIC 库的触发器本身并不包含同步复位端口，使用同步复位会增加更多逻辑资源；

- 同步复位的最大问题在于必须保证复位信号的有效时间足够长，这样才能保证所有触发器都能有效地复位。由于同步复位仅当时钟沿采样到复位信号时才会进行复位操作，所以其信号的持续时间起码要大于设计的最长时钟周期，以保证所有时钟的有效沿都能采样到同步复位信号。事实上，仅仅保证同步复位信号的持续时间大于最慢的时钟周期还是不够的，设计中还要考虑到同步复位信号树通过所有相关组合逻辑路径时的延时，以及由于时钟布线产生的偏斜（skew）。这样，只有同步复位大于时钟最大周期，加上同步信号穿过的组合逻辑路径延时，再加上时钟偏斜时，才能保证同步复位可靠、彻底。如图 4-5 所示，假设同步复位逻辑树组合逻辑的延时为 $t1$，复位信号传播路径的最大延时为 $t2$，最慢时钟的周期为 Period max，时钟的 skew 为 clk2 - clk1，则同步复位的周期 Tsyn_rst 应该满足如下公式。

```
Tsyn_rst > Period max + (clk2-clk1) + t1 + t2;
```

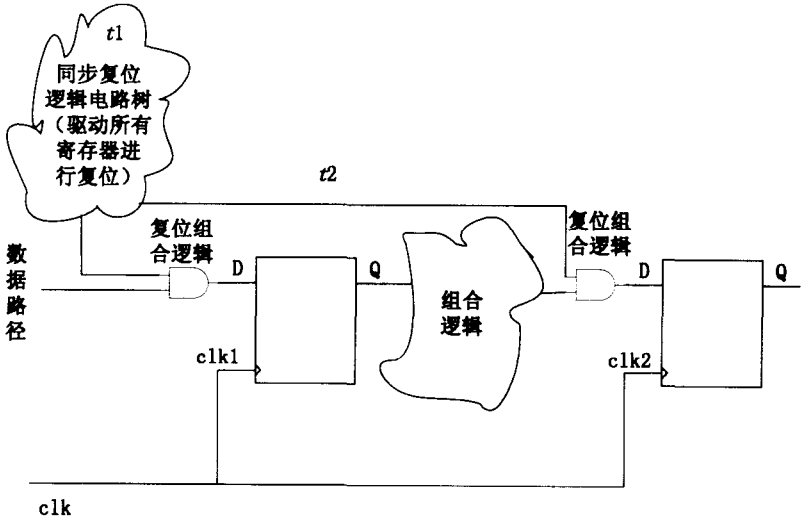

图4-5 同步复位实现结构示意图

二、异步复位电路建模

所谓异步复位是指当复位信号有效沿到达时，无论时钟沿是否有效，都会立即对目标（如寄存器、RAM 等）复位。异步复位的应用要点如下。

- 指定异步复位时，只需在 always 的敏感表中加入复位信号的有效沿即可，当复位信号有效沿到达时，无论时钟沿是否有效，复位都会立即发挥其功能。

【例 4-18】 异步复位电路建模，代码参见随书光盘中“Example-4-17\asyn_rst”目录下的相关内容。

```
always @ (posedge clk or negedge rst_)
  if (!rst_)
    begin
      ...
    end
```

```
else
  begin
    ...
  end
```

- 大多数目标器件（如 FPGA 和 CPLD）和 ASIC 库的触发器都包含异步复位端口，异步复位会被直接接到触发器的异步复位端口，如图 4-6 所示。

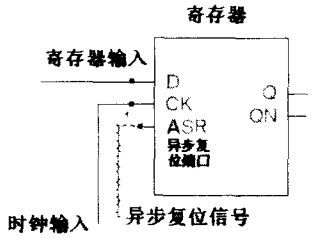

图4-6　异步复位实现结构示意图

异步复位的优点如下：

- 由于多数目标器件（如 FPGA 和 CPLD）和 ASIC 库的触发器都包含异步复位端口，异步复位会节约逻辑资源；
- 异步复位设计简单；
- 对于大多数 FPGA，都有专用的全局异步复位/置位资源（GSR, Global Set Reset），使用 GSR 资源，异步复位到达所有寄存器的偏斜（skew）最小。

异步复位的缺点如下：

- 异步复位的作用和释放与时钟沿没有直接关系，异步复位生效时问题并不明显；但是当释放异步复位时，如果异步复位信号释放时间和时钟的有效沿到达时间几乎一致，则容易造成触发器输出为亚稳态，形成逻辑错误；
- 如果异步复位逻辑树的组合逻辑产生了毛刺，则毛刺的有效沿会使触发器误复位，造成逻辑错误。

三、推荐的复位电路设计方式

推荐的复位电路设计方式是异步复位、同步释放。这种方式，可以有效地继承异步复位设计简单的优势，并克服异步复位的上述风险与缺陷。在 FPGA 和 CPLD 等可编程逻辑器件设计中，使用异步复位、同步释放可以节约器件资源，并获得稳定可靠的复位效果。

【例 4-19】 异步复位、同步释放电路建模。异步复位、同步释放的具体设计方法很多，关键是如何保证同步地释放复位信号。本例的设计方法是在复位信号释放时，用系统时钟采样，然后将复位信号送到寄存器的异步复位端，代码参见随书光盘中“Example-4-17\asyn_rst_syn_release”目录下的相关内容。

```
// reset release circuit
reg reset_reg;
```

```
always @ (posedge clk)
    reset_reg <= rst_;
always @ (posedge clk or negedge reset_reg)
  if (!reset_reg)
    begin
      ...
    end
  else
    begin
      ...
    end
```

上例使用时钟将外部输入的异步复位信号寄存一个节拍后，再送到触发器异步复位端口的设计方法的另一个好处在于，做 STA（静态时序分析）分析时，时序工具会自动检查同步后的异步复位信号和时钟的到达（Recovery）/撤销（Removal）时间关系，如果因布线造成的 skew 导致该到达/撤销时间不能满足，STA 工具会上报该路径，帮助设计者进一步分析问题，如图 4-7 所示。

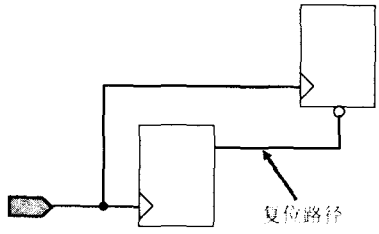

图4-7 异步部位同步化

4.3.10 使用 case 和 if...else 语句建模

一般来说，case 语句是“平行”（Balance，Parallel）的结构，即每个 case 分支的条件判断和执行都是并行的，没有“优先级（Prior）”。if...else if...else if...语句也可以建模无优先级的判断结构；而 if...if...if...结构可以建模具有优先级的判断结构。一般来说，建立优先级结构（优先级树）会消耗组合逻辑资源，如果非设计需要，推荐使用 case 或 if...else 建立无优先级的判断结构。但是某些设计中，有些信号要求先到达（如关键使能信号、选择信号等），而有些信号需要后到达（如慢速信号、有效时间较长的信号等），此时则需要使用 if...if...结构建立具有优先级的判断结构。

目前综合工具的优化能力越来越强，大多数情况下它可以将不必要的优先级树优化掉。这样综合结果是否具有优先级，很大的程度上取决于综合工具的类型、版本、目标器件（目标库）的固有硬件结构。

为了帮助读者形象地理解优先级判断结构建模的方法，下面使用几个简单的例子，分别介绍业界最流行的两个综合工具 Synplify Pro 和 Precision RTL，分析其综合结果的 RTL 视图和结构视图（初学者不用关心这两个综合工具的使用方法，而需要重点观察综合结果的

RTL 视图和结构视图，以分析不同语句建模的区别）。

【例 4-20】　分析条件判断语句建模的结构。本例对一个简单的片选电路分别使用 casex，if...else if...else if... ，if... if... if...语句建模，然后分别使用 Synplify Pro 和 Precision RTL 这两款业界流行的综合工具综合结果，并分析其 RTL 视图和结构视图。代码参见随书光盘中“Example-4-20”目录下的相关内容。

使用 casex 语句建模的代码如下。

```
always @(a or b or c or d or sel0, sel1, sel2, sel3)
  begin
    casex ({sel0, sel1, sel2, sel3})
      4'b1xxx: z = d;
      4'bx1xx: z = c;
      4'bxx1x: z = b;
      4'bxxx1: z = a;
      default: z = 1'b0;
    endcase
  end
```

使用单 if 语句（if...else if...else if...）建模的代码如下。

```
always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)
  begin
    z = 0;
    if (sel3)
       z = d;
    else if (sel2)
       z = c;
    else if (sel1)
       z = b;
    else if (sel0)
       z = a;
  end
```

使用多 if 语句（if... if... if...）建模的代码如下。

```
always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)
  begin
    z = 0; //must add the default value
    if (sel0) z = a;
    if (sel1) z = b;
    if (sel2) z = c;
    if (sel3) z = d;
  end
```

上面的描述如果在 Synopsys 公司的 Design compiler 或 FPGA Compiler 等综合工具的较早版本下综合，则多 if 语句（if... if... if...）的综合结构如图 4-8 所示，而单 if 语句（if...else if...else if...）和 casex 语句的综合结构如图 4-9 所示。从这两幅图中可以看到，使用多 if 语句建模时其结果带有优先级，这时最后一条 if 语句对应的 sel3 和 d 的优先级最高，而使用单 if 语句和 casex 语句建模时则没有建立优先级。

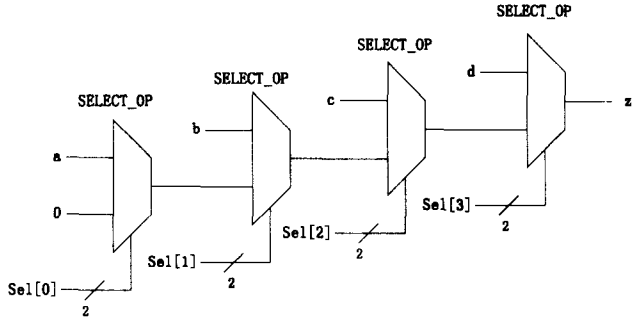

图4-8　多 if 语句 Design compiler 综合结构视图

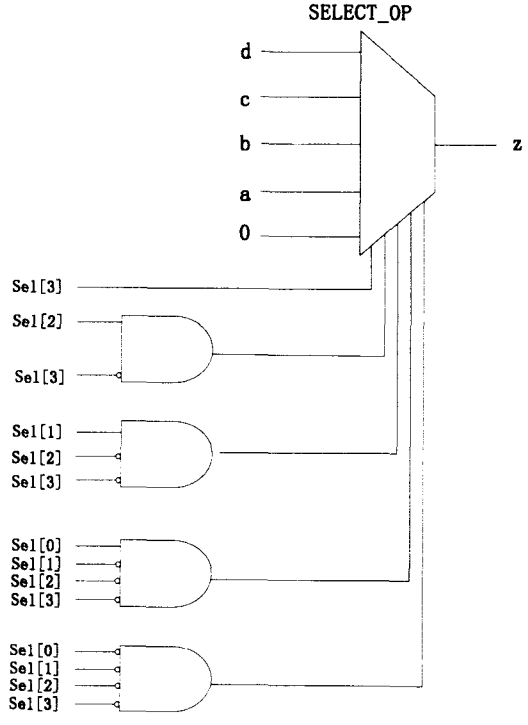

图4-9　单 if 语句和 casex 语句建模使用 Design compiler 综合结构视图

但是由于综合工具的综合优化策略不同，即使对于相同的代码，其综合结果也不尽相同。为了加深理解，这里使用业界最流行的 PLD 综合工具 Synplify Pro 和 Precision RTL 分别对多 **if** 语句（**if... if... if...**）、单 **if** 语句（**if...else if...else if...**）和 **casex** 语句进行综合，并将综合结果及其对应的 RTL 视图、结构视图分别保存在“Example-4-10\if_mult”、“Example-4-10\if_single”和“Example-4-10\case”目录下。其中多 **if** 语句的 Synplify Pro 综合结果所对应的 RTL 视图和工艺结构视图分别如图 4-10、图 4-11 所示；多 **if** 语句的

Precision RTL 综合结果所对应的 RTL 视图和工艺结构视图分别如图 4-12、图 4-13 所示。读者简单地分析一下即可发现，这 3 种语句的 Synplify Pro 和 Precision RTL 综合结果基本一致，都没有明显的优先级结构，这是因为 Synplify Pro 和 Precision RTL 这两种综合工具为了节约硬件资源，根据其优化算法，均优化掉了冗余的优先级判断结构。

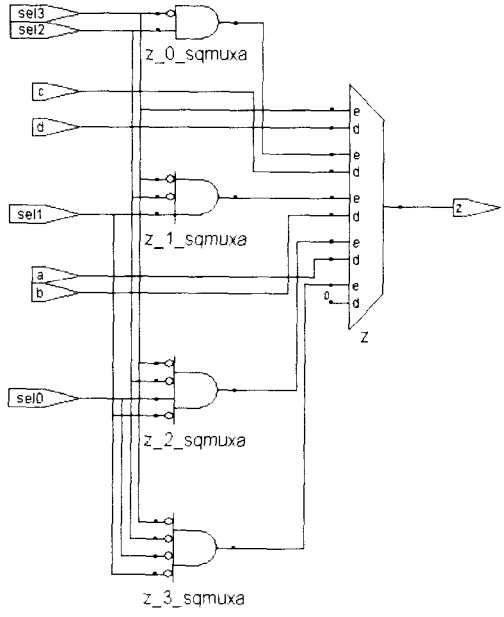

图4-10　多 if 语句的 Synplify Pro 综合结果 RTL 视图

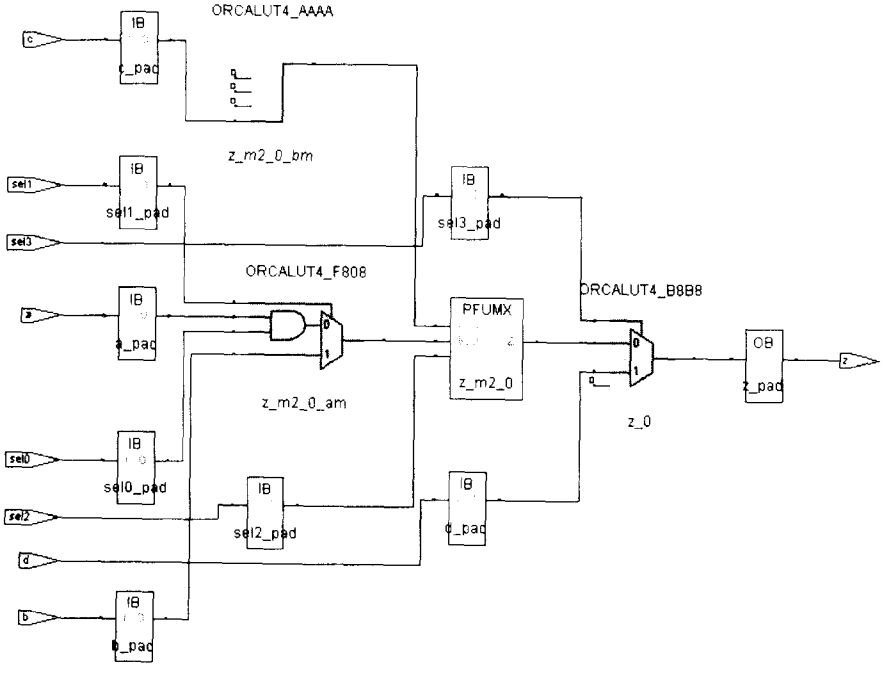

图4-11　多 if 语句的 Synplify Pro 综合结果工艺结构视图

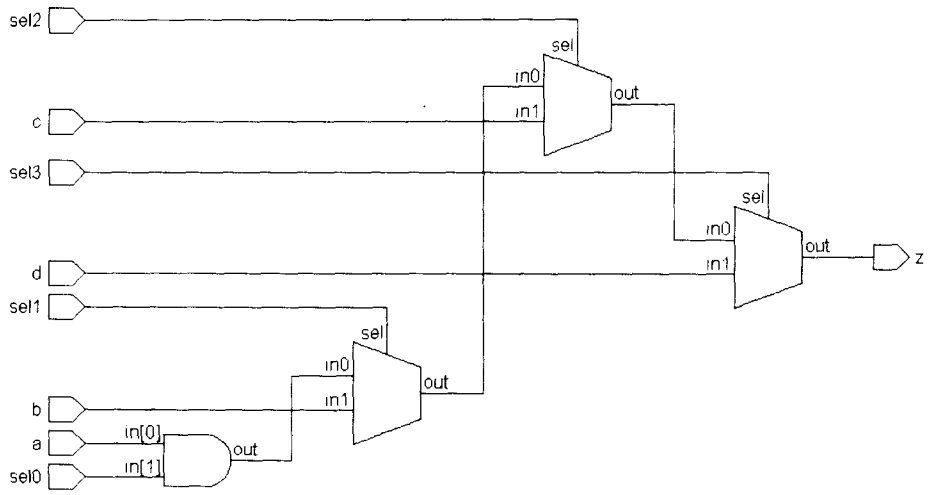

图4-12　多 if 语句的 Precision RTL 综合结果 RTL 视图

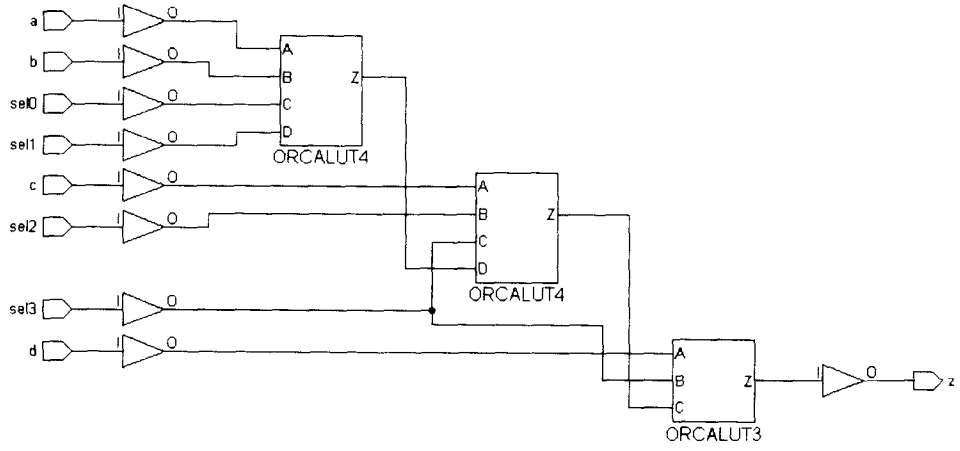

图4-13　多 if 语句的 Precision RTL 综合结果工艺结构视图

所以从语法上讲，多 if 语句（if... if... if...）可以建模具有优先级的条件判断结构；而单 if 语句（if...else if...else if...）和 case 语句可用于建模不带优先级的条件判断。但是随着综合工具优化能力的不断增强，新型的综合工具大多时候会自动优化掉优先级结构，以减少芯片面积，提高时序性能。另外，条件结构的综合结果是否带有优先级不但取决于综合工具的类型和版本，还和目标器件或目标库有直接关系。

推荐初学者尽量使用 case 或单 if 语句（if...else if...else if...）建模判断结构，这样无论使用何种综合工具，一般情况下都不会产生不必要的优先级结构。使用多 if 结构，如果没有为所有的 if 指定默认的输出，则会生成 Latch（锁存器），例如删除上例代码中的“z = 0”这一默认输出，而改为下面的描述，则会生成 Latch。使用 Synplify Pro 综合的综合结果 RTL 视图和结构视图分别如图 4-14 和图 4-15 所示。

```
always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)
  begin
```

```
    if (sel0) z = a;
    if (sel1) z = b;
    if (sel2) z = c;
    if (sel3) z = d;
end
```

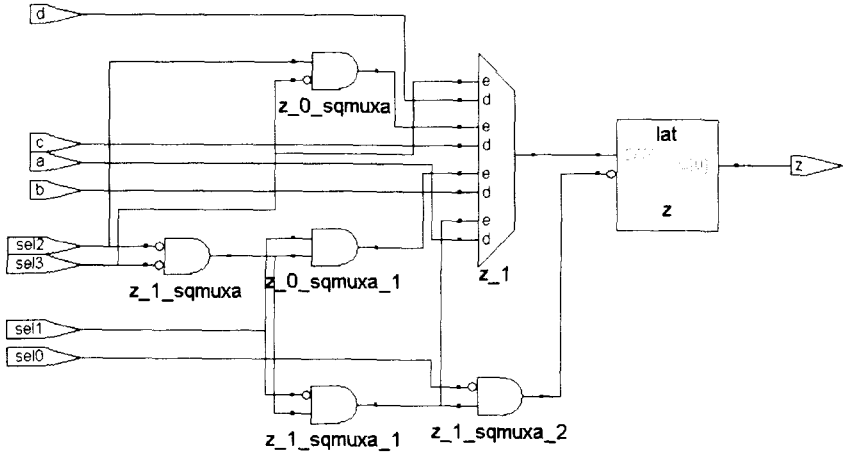

图4-14　多 if 语句无默认输出时 Synplify Pro 综合结果 RTL 视图

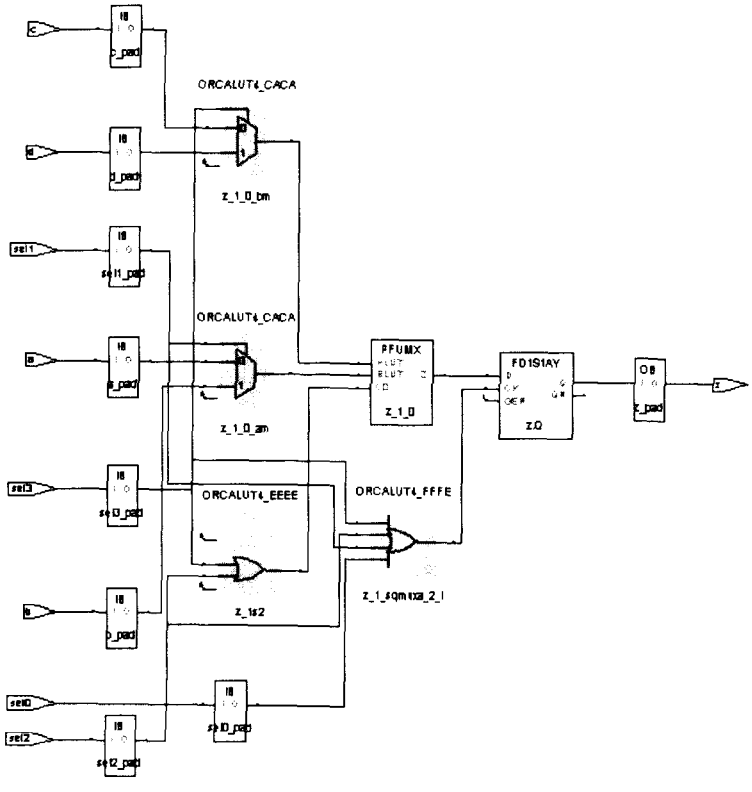

图4-15　多 if 语句无默认输出时 Synplify Pro 综合结果结构视图

如果生成的 Latch 并非设计者意愿，则会造成与设计意图的偏离，甚至是错误。而使用完整的 if...else 或 case（全译码或加有 default 关键字）语句，则可以有效地避免无意之中生成的 Latch。本书 3.3.4 小节中也有防止产生不必要 Latch 的相关描述，请读者参考。

4.3.11 可综合的 Verilog 语法子集

通过上节的建模的知识讲解，读者也许会发现，在 RTL 级建模时，使用的可综合的 Verilog 语法是整个 Verilog 语法的一个非常小的子集。其实，可综合的 Verilog 常用的关键字非常有限，这恰恰体现了 Verilog 语言是硬件描述语言的本质。Verilog HDL 作为硬件描述语言，其本质在于把硬件电路流畅、合理地转换为语言形式，同时使用较少的关键字就可以有效地将电路转换到可综合的 RTL 语言结构。

常用的 RTL 语法结构如下。

- 模块声明：module...endmodule。
- 端口声明：input、output、 inout。
- 信号类型：wire、reg、tri 等，integer 通常用于 for 语句中。
- 参数定义：parameter。
- 运算操作符：各种逻辑操作、移位操作、算术操作符（可参考本书第 2 章 2.10 节中的相关内容）。
- 比较判断：case [default] endcase (casex/casez)，if...else...。
- 连续赋值：assign，问号表达式。
- always 模块：建模时序和组合逻辑（敏感表为电平、posedge 或 negedge 的沿信号）。
- begin...end：语法分割符。
- 任务定义：tast...endtask。
- 循环语句：for。

这些关键字的语法在本书第 2 章和第 3 章中都有详细的介绍，请读者参考。

4.4 设计实例：CPU 读写 PLD 寄存器接口

本节通过一个 CPU 读写 PLD 寄存器的设计实例，综合讲解前面提到的组合逻辑、寄存器、Mux、双向总线及译码电路的建模方法，希望通过实例，能够加深读者对 RTL 级建模的感性理解。

在通信系统中，常常需要单板的 CPU（或 MCU）对单板上的 PLD 进行配置，并读取一些参数，这些参数一般存储在 PLD 的寄存器中。这类设计被称为 CPU 读写 PLD 寄存器设计。不同的 CPU 读写总线其结构和时序也不同，这里仅举一个非常简单的例子。

假设 CPU 与 PLD 的读写总线接口如图 4-16 所示，CPU 为主设备，PLD 为从设备，所有读写操作均由 CPU 发起，那么 CPU 读写时序则如图 4-17 所示，其中：

- CS 为片选信号，低电平有效，对 PLD 而言为输入信号；
- OE 为输出使能信号，低有效，用于指示数据总线上的数据是否有效，对 PLD

而言为输入信号；

- WR 为读、写指示信号，低电平指示读数据；高电平指示写数据；对 PLD 而言为输入信号；
- Address 为地址总线，假设地址总线为 8bit，对 PLD 而言为输入信号；
- Data 为数据总线，假设数据总线为 8bit，对 PLD 而言为双向总线。

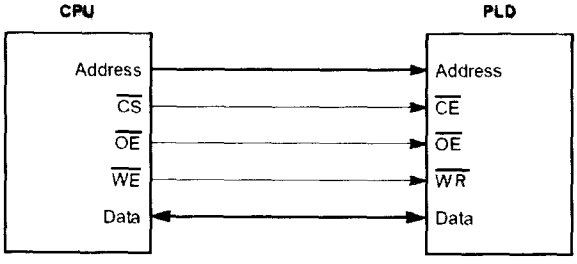

图4-16　某 CPU 的读写总线接口

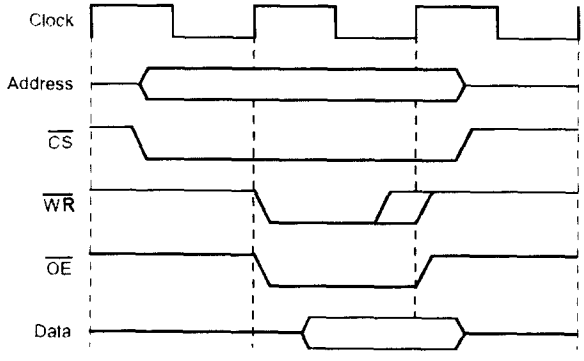

图4-17　CPU 读写时序

一般来说，CPU 读写寄存器设计由 4 部分组成。

- 地址译码电路：根据 CPU 的片选、地址、使能、读写等信号译码出对哪个寄存器进行何种操作。
- 读寄存器操作：根据 CPU 的片选、地址、使能、读写等信号读某个寄存器的值，并将读到的值送到 CPU 数据总线上。
- 写寄存器操作：根据 CPU 的片选、地址、使能、读写等信号将数据总线上的数据写入某个寄存器。
- 顶层：连接上述各个模块，并实例化 CPU 的数据总线为双向总线。

常见的 CPU 读写寄存器的方法有 3 种：使用 CPU 的读写时钟同步方式读写 PLD 寄存器；使用 OE 或 WR 的沿读写寄存器；使用 OE 或 WR 的电平异步方式读写寄存器。下面分别解析每种做法的优劣。

【例 4-21】 使用 CPU 读写时钟同步方式读写 PLD 寄存器设计，代码参见随书光盘中“\Example-4-21\syn_wr”目录下的相关内容。一般来说，CPU 的读写时钟会引入到 PLD 中，笔者利用 CPU 的读写时钟实现同步读写寄存器，提高设计的可靠性。因此这种建模方式是推荐的 CPU 读写 PLD 寄存器建模方式。

首先分析译码电路，代码参见随书光盘中“Example-4-21\syn_wr”目录下的相关内容。译码电路可以设计为组合逻辑译码，也可以在译码电路后插入一级寄存器，以确保译码结果不是亚稳态。这里笔者使用了组合逻辑译码，虽然片选、地址、使能、读写等信号有稳定周期，组合逻辑译码输出容易产生毛刺，但是在读写寄存器时使用 CPU 的读写时钟，可以有效地过滤掉组合逻辑的毛刺，提高电路的可靠性。

```
reg         CS_reg1, CS_reg2, CS_reg3;
assign my_wr = (!WR_) && (!CS_) && (!OE_);
assign my_rd = (WR_)  && (!CS_) && (!OE_);
always @ (Addr or CS_)
  if (!CS_)
    begin
      case (Addr)
        8'b 11110000: CS_reg1 <= 1'b1;
        8'b 00001111: CS_reg2 <= 1'b1;
        8'b 10100010: CS_reg3 <= 1'b1;
        default:    begin
                      CS_reg1 <= 1'b0;
                      CS_reg2 <= 1'b0;
                      CS_reg3 <= 1'b0;
                    end
      endcase
    end
```

使用 CPU 时钟同步写寄存器的模块参见随书光盘中“Example-4-21\syn_wr”目录下的相关内容。代码使用译码得出的“my_wr”信号作为判断条件，在 CPU 读写时钟的沿上进行写 PLD 寄存器操作，过滤掉了译码产生的写信号“my_wr”的毛刺，提高了写寄存器的可靠性。

```
reg   [7:0] reg1, reg2, reg3;
  always @ (posedge clk or negedge rst)
     if (!rst)
       begin
           reg1 <= 8'b0;
           reg2 <= 8'b0;
           reg3 <= 8'b0;
       end
     else
       begin
         if (my_wr)
            begin
                if (CS_reg1)
```

```
                reg1 <= data_in;
            else if (CS_reg2)
                reg2 <= data_in;
            else if (CS_reg3)
                reg3 <= data_in;
        end
    else
        begin
            reg1 <= reg1;
            reg2 <= reg2;
            reg3 <= reg3;
        end
end
```

使用 CPU 时钟同步读寄存器的模块参见随书光盘中“Example-4-21\syn_wr”目录下的相关内容。代码中使用译码得出的“my_rd”信号作为判断条件，在 CPU 读写时钟的沿上进行读 PLD 寄存器操作，过滤掉了译码产生的写信号“my_ rd”的毛刺，提高了读寄存器的可靠性。

```
reg    [7:0] data_out;
 always @ (posedge clk or negedge rst)
    if (!rst)
      data_out <= 8'b0;
    else
      begin
        if (my_rd)
          begin
            if (CS_reg1)
              data_out <= reg1;
            else if (CS_reg2)
              data_out <= reg2;
            else if (CS_reg3)
              data_out <= reg3;
          end
        else
          data_out <= 8'b0;
      end
```

顶层模块参见随书光盘中“Example-4-21\syn_wr”目录下的相关内容。在顶层中，笔者连接各个子模块并实例化 CPU 的数据总线为双向端口。实例化双向端口的方法参见本章 4.3.4 小节中的内容。另外为了提高设计的可靠性，读者可以在顶层使用 CPU 的输入时钟对片选、地址、使能、读写等信号先用寄存器打一个节拍，以防止因 CPU 总线时序不稳定，

在后续同步读写时产生亚稳态。关于亚稳态的详细论述请读者参考本书 5.2.2 小节中的内容。

【例 4-22】 使用 OE 或 WR 的沿读写 PLD 寄存器，代码参见随书光盘中“Example-4-21\ oe_edge”目录下的相关内容。如果 CPU 的读写时序能够保证其 OE 或 WR 信号的某个沿可以有效地采样数据，即该沿采样数据总线时可以满足 Setup 和 Hold 时间，则可以使用 OE 或 WR 的沿读写寄存器。假设本例中 OE 的上升沿可以有效地采样数据总线，即 OE 的上升沿采样数据总线时，Setup 和 Hold 时间都能保证满足；而且 WR 和 CS 信号都比 OE 信号宽，则设计可以做如下修改。

打开随书光盘中“Example-4-21\ oe_edge”目录下的 decode.v，对组合逻辑译码电路做如下修改。

```
assign my_wr = (!WR_) && (!CS_);
assign my_rd = (WR_)  && (!CS_);
```

这里做了一个假设，即 WR 和 CS 信号都比 OE 信号宽，也就是说当 OE 上升沿读写寄存器时，CS 和 WR 信号始终保持有效，只有这样，才能保证译码是正确的。

CPU 写寄存器电路代码参见随书光盘中“Example-4-21\ oe_edge”目录下的相关内容；CPU 读寄存器电路代码参见随书光盘中“Example-4-21\ oe_edge”目录下的相关内容。这两段代码的最大修改之处在于将描述读写的 always 模块的触发沿修改为 OE 信号的上升沿。同理，使用 OE 的上升沿也可以有效地滤除组合逻辑译码电路带来的毛刺。

```
always @ (posedge OE_ or negedge rst)
```

顶层模块参见随书光盘中“Example-4-21\ oe_edge”目录下的相关内容，在其中去掉了 CPU 的读写时钟。

使用 OE 或 WR 的沿读写寄存器的描述，看起来比前面介绍的使用 CPU 时钟同步读写寄存器的描述简单，但是读者必须明确，这种方式正常工作的前提条件有如下两个：

- OE 的上升沿可以有效地采样数据总线，即 OE 的上升沿采样数据总线时 Setup 和 Hold 都能被保证满足；
- WR 和 CS 信号都比 OE 信号宽，即 OE 上升沿读写寄存器时，CS 和 WR 信号始终保持有效。

只有在这两个条件同时满足的前提下，才能保证使用 OE 的沿读写 PLD 寄存器电路是可靠的。

【例 4-23】 使用 OE 或 WR 的电平异步方式读写寄存器，代码参见随书光盘中“Example-4-21\ asyn_bad”目录下的相关内容。首先读者必须明确，这种方式通常是不可靠的，是必须摒弃的。有很多初学者习惯使用译码结果作为电平敏感的组合逻辑，实现读写寄存器操作，这是非常危险的。前面提到过，如果译码电路是组合逻辑，则其译码结果就有可能带有毛刺，另外由于 CPU 总线的时序在电压、温度、环境变化的情况下可能遭到破坏，造成 OE、WR、CS 等信号的时序余量恶化，如果此时将译码得到的电平作为电平敏感的 always 模块进行读写寄存器操作（参见随书光盘中“Example-4-21\ asyn_bad”目录

下的 read_reg.v 和 write_reg.v)，则会因为毛刺和错误电平造成读写错误，所以这种设计方法必须摒弃。

4.5 小结

本章首先介绍了 RTL 和综合的基本概念，然后通过一个个具体范例，力图使初学者逐步掌握综合 RTL 子集的概念。希望读者认真琢磨常用电路结构的建模方法，在实践中掌握 RTL 级设计的基本技巧。

4.6 问题与思考

1. 逻辑综合的含义是什么？
2. RTL 级设计的基本要素和步骤是什么？
3. 如何设计简单和复杂的双向端口与三态总线？
4. 常用的复位方式有哪些？为什么推荐使用异步复位、同步释放方式？

第5章 RTL 设计与编码指导

通过第 4 章的学习，相信读者已经对 RTL 级描述有了一些感性的认识，本章将更为深入地探讨 RTL 级设计的基本规律。RTL 级设计涉及到的规律与方法很多，在此不可能面面俱到，希望通过本章的介绍，能引起读者的重视。在日后的工作实践中，读者还要有意识地积累一些设计原则、设计思想，以作为设计指导。

本章主要内容如下：

- 一般性指导原则；
- 同步设计原则和多时钟处理；
- 代码风格；
- 结构层次设计和模块划分；
- 组合逻辑的注意事项；
- 时钟设计的注意事项；
- RTL 代码优化技巧。

5.1 一般性指导原则

RTL 级设计的评判标准很多，如时序性能、所占面积、可测试性、可重用性、功耗、时钟域的分配、复位信号设计以及是否与所用 EDA 工具匹配等。如果设计目标是在 FPGA 或 CPLD 等可编程逻辑器件上实现，则还需考虑是否能发挥这些 PLD 的结构特点等。根据上述这些目标的组合和优先级设置，可以派生出很多不同的设计原则。这里仅讨论一般意义上的指导原则。

这里抛砖引玉地提出 4 个基本设计原则，这些指导原则涉及范畴非常广，希望读者不仅要学习它们，更重要的是理解它们，并在今后的工作实践中充实、完善它们。

(1) 面积和速度的平衡与互换原则。

面积和速度的平衡与互换原则提出了 RTL 设计的两个基本目标，并探讨了这两个目标对立统一的矛盾关系。

(2) 硬件原则。

硬件原则重点在于提醒读者注意弱化软件设计的思路，理解 HDL 语言设计的本质。

(3) 系统原则。

系统原则希望读者能够在整体上把握设计，从而提高设计质量，优化设计效果。

(4) 同步设计原则。

同步设计原则是设计稳定时序的基本要求，也是高速 RTL 设计的通用法则。

5.1.1 面积和速度的平衡与互换原则

这里的“面积”是指一个设计所消耗的目标器件（如 FPGA、CPLD 和 ASIC 等）的硬件资源数量。对于 FPGA 来说，可以用所消耗的触发器（FF）和查找表（LUT）数量来衡量；对于 CPLD 来说，常用宏单元（MC）衡量；对于 ASIC 来说，则可以用设计的系统门衡量。“速度”指设计在芯片上稳定运行时所能够达到的最高频率，这个频率由设计的时序状况决定，与设计满足的时钟周期、PAD to PAD Time、Clock Setup Time、Clock Hold Time 和 Clock-to-Output Delay 等众多时序特征量密切相关。面积（Area）和速度（Speed）这两个指标贯穿于 RTL 设计的始终，是衡量设计质量的终极标准。这里讨论一下设计中关于面积和速度的基本原则，即面积和速度的平衡与互换原则。

面积和速度是一对对立统一的矛盾体。要求一个设计同时具备设计面积最小，运行频率最高的特点是不现实的。科学的设计目标应该是在满足设计时序要求（包含对设计最高频率的要求）的前提下，占用最小的芯片面积，或者在所规定的面积下，使设计的时序余量更大，频率更高。这两种目标充分体现了面积和速度的平衡思想。关于面积和速度的要求，读者不应该简单地理解为工程师水平的提高和设计完美性的追求，而应该认识到它们是和产品的质量、成本直接相关的。如果设计的时序余量比较大，运行的频率比较高，则意味着设计的健壮性更强，整个系统的质量更有保证；另一方面，设计所消耗的面积更小，则意味着在单位芯片上实现的功能模块更多，需要的芯片数量更少，整个系统的成本也随之大幅度削减。

作为矛盾的两个方面，面积和速度的地位是不一样的。相比之下，满足时序、工作频率的要求更重要一些，所以当两者发生冲突时，应采用速度优先的原则。

面积和速度的互换是 RTL 设计的一个重要思想。从理论上讲，一个设计如果时序余量较大，所能跑的频率远远高于设计要求，那么就能通过功能模块复用减少整个设计所消耗的芯片面积，也就是用速度的优势对换面积的节约。反之，如果一个设计的时序要求很高，普通方法达不到设计频率，那么一般可以通过将数据流串并转换，并行复制多个操作模块，对整个设计采取“乒乓操作”和“串并转换”的思想进行处理，在芯片输出模块处再对数据进行“并串转换”。从宏观上看，整个芯片满足了处理速度的要求，这相当于用面积复制换取速度的提高。面积和速度互换的操作技巧很多，比如模块复用、“乒乓操作”、“串并转换”等，这些技巧需要读者在今后的工作中不断积累。下面举例说明如何进行“速度换面积”和“面积换速度”的操作。

【例 5-1】 进行“用速度的优势换取面积的节约”操作，代码参见随书光盘中“Example-5-1\soure”目录下的相关内容。

在 WCDMA（宽带码分多址）系统中用到了快速哈达码（FHT）运算，FHT 由 4 步相同的算法完成，如图 5-1 所示。

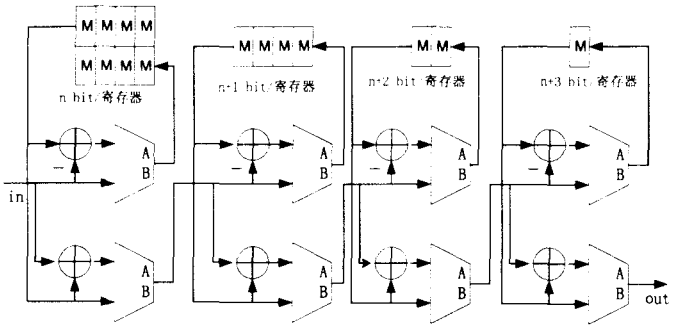

图5-1 FHT 原理图

FHT 的单步算法如下。

$$Out[2i] = In[2i] + In[2i+8]; i = 0-7;$$
$$Out[2i+1] = In[2i+1] - In[2i+1+8]; i = 0-7$$

考虑流水线式数据处理的要求，最自然的设计方法就是设计不同端口宽度的 4 个单步 FHT，并将这 4 个单步模块串联起来，从而完成数据流的流水线处理。该 FHT 的实现代码如下。

```
    //该模块是 FHT 的顶层，调用 4 个不同端口宽度的单步 FHT 模块，完成整个 FHT 算法
    module
fhtpart(Clk,Reset,FhtStarOne,FhtStarTwo,FhtStarThree,FhtStarFour,
            I0,I1,I2,I3,I4,I5,I6,I7,I8,
            I9,I10,I11,I12,I13,I14,I15,
            Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
            Out9,Out10,Out11,Out12,Out13,Out14,Out15);
    input Clk;    //设计的主时钟
    input Reset;   //异步复位
    input FhtStarOne,FhtStarTwo,FhtStarThree,FhtStarFour; //4 个单步算法的时
序控制信号
    input [11:0] I0,I1,I2,I3,I4,I5,I6,I7,I8;
    input [11:0] I9,I10,I11,I12,I13,I14,I15;                    //FHT 的 16 个输入
    output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
    output [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15;  //FHT 的
16 个输出

    //第 1 次 FHT 单步运算的输出
    wire [12:0] m0,m1,m2,m3,m4,m5,m6,m7,m8,m9;
    wire [12:0] m10,m11,m12,m13,m14,m15;

    //第 2 次 FHT 单步运算的输出
    wire [13:0] mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,mm9;
    wire [13:0] mm10,mm11,mm12,mm13,mm14,mm15;
```

```
//第 3 次 FHT 单步运算的输出
wire [14:0] mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,mmm9;
wire [14:0] mmm10,mmm11,mmm12,mmm13,mmm14,mmm15;

//第 4 次 FHT 单步运算的输出
wire [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9;
wire [15:0] Out10,Out11,Out12,Out13,Out14,Out15;
//第 1 次 FHT 单步运算
fht_unit1 fht_unit1(Clk,Reset,FhtStarOne,
          I0,I1,I2,I3,I4,I5,I6,I7,I8,
          I9,I10,I11,I12,I13,I14,I15,
          m0,m1,m2,m3,m4,m5,m6,m7,m8,
          m9,m10,m11,m12,m13,m14,m15
          );

//第 2 次 FHT 单步运算
fht_unit2 fht_unit2(Clk,Reset,FhtStarTwo,
          m0,m1,m2,m3,m4,m5,m6,m7,m8,
          m9,m10,m11,m12,m13,m14,m15,
          mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,
          mm9,mm10,mm11,mm12,mm13,mm14,mm15
          );
//第 3 次 FHT 单步运算
fht_unit3 fht_unit3(Clk,Reset,FhtStarThree,
          mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,
          mm9,mm10,mm11,mm12,mm13,mm14,mm15,
          mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,
          mmm9,mmm10,mmm11,mmm12,mmm13,mmm14,mmm15
          );
//第 4 次 FHT 单步运算
fht_unit4 fht_unit4(Clk,Reset,FhtStarFour,
          mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,
          mmm9,mmm10,mmm11,mmm12,mmm13,mmm14,mmm15,
          Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
          Out9,Out10,Out11,Out12,Out13,Out14,Out15
          );
endmodule
```

单步 FHT 运算如下（仅以第 4 步的模块为例）。

```
module fht_unit4(Clk,Reset,FhtStar,
          In0,In1,In2,In3,In4,In5,In6,In7,In8,
          In9,In10,In11,In12,In13,In14,In15,
          Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
          Out9,Out10,Out11,Out12,Out13,Out14,Out15
          );

input Clk;          //设计的主时钟
input Reset;        //异步复位
input FhtStar;      //单步 FHT 运算控制信号
input [14:0] In0,In1,In2,In3,In4,In5,In6,In7,In8,In9;
input [14:0] In10,In11,In12,In13,In14,In15;          //单步 FHT 运算输入
output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9;
output [15:0] Out10,Out11,Out12,Out13,Out14,Out15;  //单步 FHT 运算输出

//Single FHT calculation
reg [15:0] Out0,Out1,Out2,Out3,Out4,Out5;
reg [15:0] Out6,Out7,Out8,Out9,Out10,Out11;
reg [15:0] Out12,Out13,Out14,Out15;
//补码运算
wire [14:0] In8Co =~In8+1;
wire [14:0] In9Co =~In9+1;
wire [14:0] In10Co=~In10+1;
wire [14:0] In11Co=~In11+1;
wire [14:0] In12Co=~In12+1;
wire [14:0] In13Co=~In13+1;
wire [14:0] In14Co=~In14+1;
wire [14:0] In15Co=~In15+1;

always @(posedge Clk or negedge Reset)
begin
  if(!Reset)
  begin
    Out0<=0;Out1<=0;Out2<=0;Out3<=0;
    Out4<=0;Out5<=0;Out6<=0;Out7<=0;
    Out8<=0;Out9<=0;Out10<=0;Out11<=0;
    Out12<=0;Out13<=0;Out14<=0;Out15<=0;
  end
  else
```

```
    begin
      if(FhtStar)
      begin
      Out0<={In0[14],In0 }+{In8[14],In8 };
      Out1<={In0[14],In0 }+{In8Co[14],In8Co };
      Out2<={In1[14],In1 }+{In9[14],In9 };
      Out3<={In1[14],In1 }+{In9Co[14],In9Co };
      Out4<={In2[14],In2 }+{In10[14],In10 };
      Out5<={In2[14],In2 }+{In10Co[14],In10Co };
      Out6<={In3[14],In3 }+{In11[14],In11 };
      Out7<={In3[14],In3 }+{In11Co[14],In11Co };
      Out8<={In4[14],In4 }+{In12[14],In12 };
      Out9<={In4[14],In4 }+{In12Co[14],In12Co };
      Out10<={In5[14],In5 }+{In13[14],In13 };
      Out11<={In5[14],In5 }+{In13Co[14],In13Co };
      Out12<={In6[14],In6 }+{In14[14],In14 };
      Out13<={In6[14],In6 }+{In14Co[14],In14Co };
      Out14<={In7[14],In7 }+{In15[14],In15 };
      Out15<={In7[14],In7 }+{In15Co[14],In15Co };
      end
    end
  end
  endmodule
```

评估一下系统的流水线时间余量就会发现，处理整个流水线需要 16 个时钟周期，而 FHT 模块的运算速度较快，其中加法操作本身仅消耗 1 个时钟周期，加上数据的选择和分配所消耗的时间，也完全能满足系统的频率要求，所以将单步 FHT 运算复用 4 次，就能在很大程度上减少资源的消耗。这种复用单步算法的 FHT 结构图如图 5-2 所示，由输入选择寄存、单步 FHT 模块，由输出选择寄存和计数器构成。

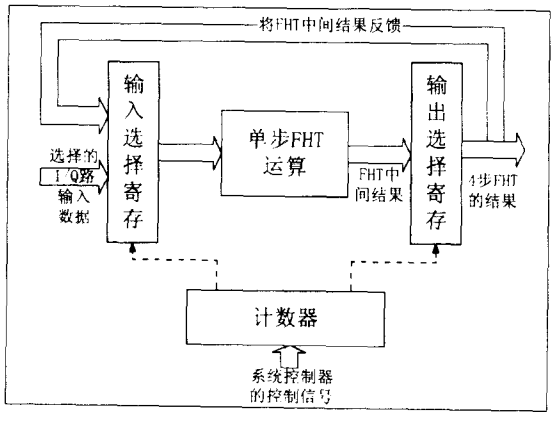

图5-2　FHT 运算复用结构图

代码如下。

```
//复用单步算法的 FHT 运算模块
module wch_fht(Clk,Reset,
          PreFhtStar,
          In0,In1,In2,In3,In4,In5,In6,In7,
          In8,In9,In10,In11,In12,In13,In14,In15,
          Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
          Out9,Out10,Out11,Out12,Out13,Out14,Out15
          );
input Clk;           //设计的主时钟
input Reset;         //异步复位信号
input PreFhtStar;    //FHT 运算指示信号，与上级模块运算相关联
input [11:0] In0,In1,In2,In3,In4,In5,In6,In7;
input [11:0] In8,In9,In10,In11,In12,In13,In14,In15;    //FHT 的 16 个输入
output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
output [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15; //FHT 的
16 个输出

//FHT 输出寄存信号
reg [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
reg [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15;
//FHT 的中间结果
wire [15:0] Temp0,Temp1,Temp2,Temp3,Temp4,Temp5,Temp6,Temp7;
wire [15:0] Temp8,Temp9,Temp10,Temp11,Temp12,Temp13,Temp14,Temp15;

//FHT 运算控制计数器，与前一级流水线模块配合
reg [2:0] Cnt3;//count from 0 to 4,when Reset Cnt3=7;
reg FhtEn;//Enable fht culculate

always @(posedge Clk or negedge Reset)
begin
  if (!Reset)
    Cnt3<= #1 3'b111;
  else
  begin
     if(PreFhtStar)
     Cnt3<= #1 3'b100;
     else
     Cnt3<= #1 Cnt3-1;
```

```
    end
end
always @(posedge Clk or negedge Reset)
if (!Reset)
   FhtEn<= #1 0;
else
begin
    if (PreFhtStar)
    FhtEn<= #1 1;
    if  (Cnt3==1)
    FhtEn<= #1 0;
end

//补码运算，复制符号位
      assign Temp0=(Cnt3==4)?{4{In0[11]},In0}:Out0;
      assign Temp1=(Cnt3==4)?{4{In1[11]},In1}:Out1;
      assign Temp2=(Cnt3==4)?{4{In2[11]},In2}:Out2;
      assign Temp3=(Cnt3==4)?{4{In3[11]},In3}:Out3;
      assign Temp4=(Cnt3==4)?{4{In4[11]},In4}:Out4;
      assign Temp5=(Cnt3==4)?{4{In5[11]},In5}:Out5;
      assign Temp6=(Cnt3==4)?{4{In6[11]},In6}:Out6;
      assign Temp7=(Cnt3==4)?{4{In7[11]},In7}:Out7;
      assign Temp8=(Cnt3==4)?{4{In8[11]},In8}:Out8;
      assign Temp9=(Cnt3==4)?{4{In9[11]},In9}:Out9;
      assign Temp10=(Cnt3==4)?{4{In10[11]},In10}:Out10;
      assign Temp11=(Cnt3==4)?{4{In11[11]},In11}:Out11;
      assign Temp12=(Cnt3==4)?{4{In12[11]},In12}:Out12;
      assign Temp13=(Cnt3==4)?{4{In13[11]},In13}:Out13;
      assign Temp14=(Cnt3==4)?{4{In14[11]},In14}:Out14;
      assign Temp15=(Cnt3==4)?{4{In15[11]},In15}:Out15;

always @(posedge Clk or negedge Reset)
begin
if (!Reset)
begin
   Out0<=0;Out1<=0;Out2<=0;Out3<=0;
   Out4<=0;Out5<=0;Out6<=0;Out7<=0;
   Out8<=0;Out9<=0;Out10<=0;Out11<=0;
   Out12<=0;Out13<=0;Out14<=0;Out15<=0;
```

```
    end
    else
    begin
      if ((Cnt3<=4) && Cnt3>=0 && FhtEn)
        begin
        Out0[15:0]<= #1 Temp0[15:0]+Temp8[15:0];
        Out1[15:0]<= #1 Temp0[15:0]-Temp8[15:0];
        Out2[15:0]<= #1 Temp1[15:0]+Temp9[15:0];
        Out3[15:0]<= #1 Temp1[15:0]-Temp9[15:0];
        Out4[15:0]<= #1 Temp2[15:0]+Temp10[15:0];
        Out5[15:0]<= #1 Temp2[15:0]-Temp10[15:0];
        Out6[15:0]<= #1 Temp3[15:0]+Temp11[15:0];
        Out7[15:0]<= #1 Temp3[15:0]-Temp11[15:0];
        Out8[15:0]<= #1 Temp4[15:0]+Temp12[15:0];
        Out9[15:0]<= #1 Temp4[15:0]-Temp12[15:0];
        Out10[15:0]<= #1 Temp5[15:0]+Temp13[15:0];
        Out11[15:0]<= #1 Temp5[15:0]-Temp13[15:0];
        Out12[15:0]<= #1 Temp6[15:0]+Temp14[15:0];
        Out13[15:0]<= #1 Temp6[15:0]-Temp14[15:0];
        Out14[15:0]<= #1 Temp7[15:0]+Temp15[15:0];
        Out15[15:0]<= #1 Temp7[15:0]-Temp15[15:0];
        end

    end
    end
    endmodule
```

为了便于对比两种实现方式的资源消耗，在 Synplify Pro 中对两种实现方法分别做了综合。两次综合选用的参数完全一致，以便于考察设计所消耗的寄存器和逻辑资源，Enable “Disable I/O Insertion”选项，不插入 IO，取消 Synplify Pro 中诸如“FSM Compiler”、“FSM Explorer”、“Resource Sharing”、“Retiming”、“Pipelining”等综合优化选项。两次综合的结果如图 5-3 和图 5-4 所示。

Log Parameter	rev_2
fhtpart I/O primitives	Not Available
fhtpart I/O Register bits	0
fhtpart Register bits (Non I	928 (38%)
fhtpart Total Luts	1328 (55%)

未复用的FHT实现方案占用的资源。取消了所有综合优化选项，并“Disable I/O Insertion”

图5-3 未采用复用方案的“fhtpart”模块综合所消耗的资源

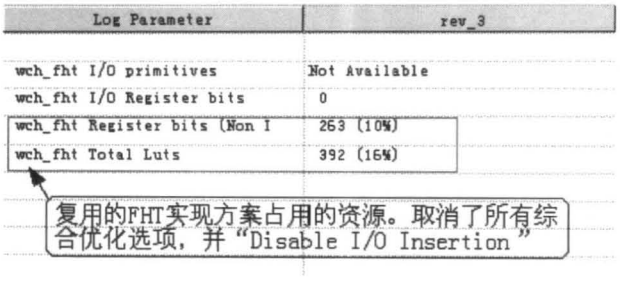

图5-4　采用复用方案的“wch_fht”模块综合所消耗的资源

通过对比可以清晰地看到，采用复用实现方案所占用的面积约为原方案的 1/4，而得到这个好处的代价是，完成整个 FHT 运算的周期为原来的 4 倍。这个例子用运算周期的加长换取了芯片面积的减少，是前面所述的用频率换面积的一种体现。本例所述“频率换面积”的前提是，FHT 模块频率较高，运算周期的余量较大，采用 4 步复用后，仍然能够满足系统流水线设计的要求。其实，如果流水线时序允许，FHT 运算甚至可以采用 1bit 全串行方案实现，该方案所消耗的芯片面积资源更少。

【例 5-2】　如何进行“面积换速度”的操作？

本例是一个路由器设计实例。假设输入数据流的速率是 450Mbit/s，而在 FPGA 上设计的数据处理模块的处理速度最大为 150Mbit/s，由于处理模块的数据吞吐量满足不了要求，因此直接在 FPGA 上实现是一个“不可能完成的任务”。在这种情况下，就应该利用“面积换速度”的思想，至少复制 3 个处理模块。首先将输入数据进行串并转换，然后利用这 3 个模块并行处理分配的数据，最后将处理结果“并串转换”，以满足数据速率的要求。在处理模块的两端，其数据速率是 450Mbit/s，而在 FPGA 的内部，每个子模块处理的数据速率是 150Mbit/s。其实整个数据吞吐量的保障是依赖于 3 个子模块的并行处理来完成的，也就是说通过占用更多的芯片面积来实现高速处理。设计示意图如图 5-5 所示。

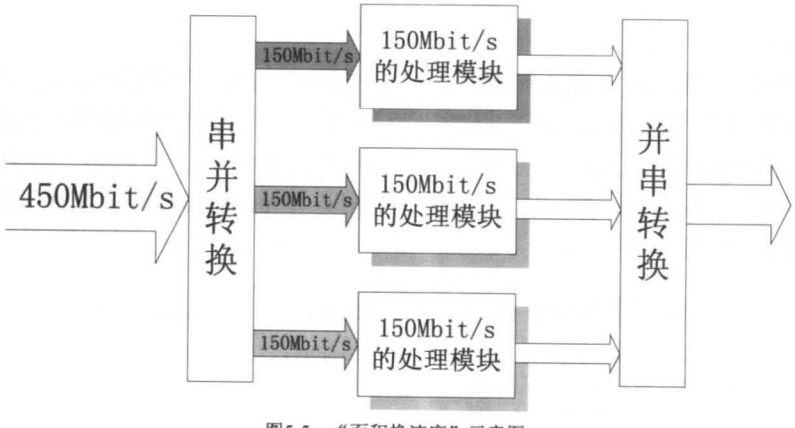

图5-5　“面积换速度”示意图

上面仅仅是对“面积换速度”思想的一个简单例举，其实在具体操作过程中还会涉及到很多的方法和技巧，例如，对高速数据流进行串并转换，采用“乒乓操作”方法提高数据处理速率等。希望读者通过平时的练习进一步掌握各种技巧。

5.1.2 硬件原则

硬件原则主要针对 HDL 代码编写而言。首先应该明确，FPGA/CPLD、ASIC 的逻辑设计所采用的硬件描述语言（HDL）同软件语言（如 C、C++等）是有本质区别的。以 Verilog 语言为例，虽然 Verilog 的很多语法规则与 C 语言相似，但是 Verilog 作为硬件描述语言，它的本质作用在于描述硬件，应该认识到 Verilog 是采用了 C 语言形式的硬件的描述抽象，它的最终实现结果是芯片内部的实际电路。

所以评判一段 HDL 代码优劣的最终标准是其描述并实现的硬件电路的性能（包括面积和速度两个方面）。如果说一个设计的代码水平较高，仅仅是说这个设计由硬件向 HDL 代码这种表现形式转换得更流畅、更合理。而一个设计的最终性能，在更大程度上取决于设计工程师所构想的硬件实现方案的效率以及合理性。

初学者，特别是由软件转行的初学者，片面追求代码的整洁、简短是错误的，是与评价 HDL 的标准背道而驰的。正确的编码方法是，首先要做到对所需实现的硬件电路“胸有成竹”，对该部分硬件的结构与连接十分清晰，然后再用适当的 HDL 语句将其表达出来。

前面已经讨论过，HDL 语言与 C 等软件语言相比，其最显著的区别在于 HDL 语言便于描述“互联”、“并发”、“时间”这 3 个硬件设计的基本概念。

- 互连（connectivity）：互连是硬件电路的一个基本要素，在 C 语言中，并没有可以直接用来表示模块间互连的变量；而 HDL 的网线型变量则专为模块互连而设计，用它描述电路连接清晰明确。如 Verilog 的 wire 型变量配合一些驱动结构就能有效地描述各个模块之间的端口与网线连接关系。
- 并发（concurrency）：C 语言天生是串行的，不能描述硬件之间并发的特性。C 语言编译后，其机器指令在 CPU 的高速缓冲队列中基本上是顺序执行的，而 Verilog 则可以有效地描述并行的硬件系统。硬件系统比软件系统速度快、实时性高的一个重要原因就是硬件系统中各个单元的运算是独立的，信号流是并行的。所以在使用 HDL 建模时，应该充分理解硬件系统并行处理的特点，合理安排数据流的时序，提高整个设计的效率。
- 时间（time）：C 程序运行的时候，没有一个严格的时间概念，程序运行时间的长短，取决于处理器本身的性能，而 HDL 语言本身定义了绝对和相对的时间度量，在仿真时可以通过时间度量与周期关系描述信号之间的时间关系。

Verilog 作为一种 HDL 语言，对系统行为的建模方式是分层次的。比较重要的层次有系统级（System）、算法级（Algorithm）、寄存器传输级（RTL）、逻辑级（Logic）、门级（Gate）和电路开关级（Switch）等。系统级和算法级与 C 语言更相似，可用的语法和表现形式也更丰富。自 RTL 级以后，HDL 语言的功能就越来越侧重于硬件电路的描述，可用的语法和表现形式的局限性也越大。相比之下，C 语言与系统级和算法级 Verilog 描述更相近一些，而与 RTL 级、Gate 级、Switch 级描述从描述目标到表现形式上都有较大的差异。

【例 5-3】 举例说明 RTL 级 Verilog 描述语法和 C 语言描述语法的区别。

在 C 语言的描述中，为了使代码执行效率高，表述简洁，经常用到下面的 for 循环语句。

```
for (i=0; i<16; i++)
```

```
    DoSomething();
```

但是在实际工作中，除了描述仿真测试激励（Testbench）时使用 for 循环语句外，RTL 级编码中必须要慎用 for 循环。这是因为 for 循环会被综合器展开为所有变量依次执行的语句，每个变量独立占用寄存器资源，有些情况下不能有效地复用硬件逻辑资源，会造成资源的浪费。当在 RTL 硬件描述中遇到循环算法时，通常的做法是先搞清楚设计的时序要求，做一个 reg 型计数器，在每个时钟沿累加，并在每个时钟沿处判断计数器情况，做出相应的处理，能复用的处理模块尽量复用，即使所有操作不易复用，也可以采用 case 语句展开描述，代码如下。

```
reg [3:0] counter;
always @ (posedge clk)
if (syn_rst)
    counter <= 4'b0;
else
    counter <= counter+1;

always @ (posedge clk)
    begin
        case (counter)
            4'b0000:
            4'b0001:
            ...
            default:
        endcase
    end
```

另外，在 C 语句描述中还有 if...else 和 switch 条件判断语句，其语法如下。

```
if (flag)    // 表示 flag 为真
...
else
...
```

switch 语句的基本格式如下。

```
switch (variable)
{
case value1 :   ...
break;
case value2 :   ...
break;
...
default :   ...
break;
```

```
}
```

两者之间的区别主要在于 switch 是多分支选择语句，而 if 语句只有两个分支可供选择。虽然可以用嵌套的 if 语句来实现多分支选择，但那样的话程序将会冗长难读。

Verilog 也有 if...else 语句和 case 语句，if 语句的语法与 C 语言相似，case 语句的语法如下。

```
case (var)
    var_value1:
    var_value1:
    ...
    default:
endcase
```

通过第 4 章 4.3.10 小节的学习，读者会发现 case 语句、if...else if...else if...语句以及 if...if...if...语句建模时可以建立无优先级和有优先级的判断结构。使用 HDL 语言建模的关键在于选择适当的建模方法与描述，保证该描述对应的综合实现结果的硬件结构满足设计要求。

下面进一步讨论 Verilog 的 for 循环。前面讲过 Verilog 语言是分层次的，即使是同一个语法关键字，在不同的应用层次也有不同的理解，for 循环就是一个非常好的例子。

- for 循环在行为级描述测试激励时的应用：前面介绍过，本书推荐使用 Behavior 级方式描述测试激励，在描述测试激励时，推荐使用 for 循环。好处主要有两个，一是描述简单，代码清晰；二是仿真器会对 for 循环开放一片内存，提高代码执行效率，加快仿真进程。本书第 7 章 7.2.1 小节中论述的就是如何使用 for 语句实现遍历测试。
- for 循环在 RTL 级描述硬件电路时的应用：在 RTL 级描述硬件时，一定要慎用 for 循环。前面已经介绍过，for 循环在硬件实现时会被综合器展开，不利于硬件资源的复用，如果应用不当，还会造成资源浪费。但是任何问题都不是绝对的，如果用户非常清晰 for 循环会被综合器展开这一基本原则，则可以逆向思维，将某些硬件上无法复用的展开结构抽象为 for 循环描述，提高代码的可读性。例如本书第 8 章 8.6.2 小节中的“案例分析”就分别列举了使用展开结构和使用 for 循环语句对某“冒泡排序”寻找最小值电路的建模方法，读者可以仔细琢磨。

5.1.3 系统原则

系统原则包含两个层次的含义，一是实现的目标器件本身可以看做一个系统，需要充分有效地发挥该系统每个单元的功效。如果设计的实现目标为 FPGA，因为当代 FPGA 内嵌了很多固有的硬件资源（如可编程输入/输出单元、基本可编程逻辑单元、嵌入式块 RAM、丰富的布线资源、底层嵌入功能单元和内嵌专用硬核等），如何合理地使用这些硬件资源，对设计的全局有个宏观上的合理安排，比如合理安排时钟域、模块复用、约束、面积和速度等问题，就显得至关重要。如果实现目标是 SOC，则需要分析什么样的算法和功能适合放在硬件系统里面实现；什么样的算法和功能适合放在微处理器系统（如 DSP、CPU 等）里面实现，并需要进一步合理划分软、硬件之间的数据交换。

从更高层面上看，任何一个硬件系统如何进行模块划分与任务分配，什么样的算法和功能适合放在可编程逻辑器件或 ASIC 里面实现，什么样的算法和功能适合放在 DSP、CPU 等微处理器中实现，如何划分软、硬件功能、安排模块接口设计等问题都非常重要。要知道在系统上复用模块节省的面积远比在代码上小打小闹来的实惠得多。

图 5-6 所示是 FPGA 设计的系统规划流程图，从该图中可以发现，从整体上对设计进行模块复用应该在定义完系统功能之后就充分考虑，并指导模块的具体划分。模块划分非常重要，除了关系到是否能最大程度上发挥项目成员的协同设计能力，还直接决定着设计的综合、实现效果和相关的操作时间。模块划分的具体方法可参考本章 5.4.2 小节中关于模块划分技巧的相关论述。

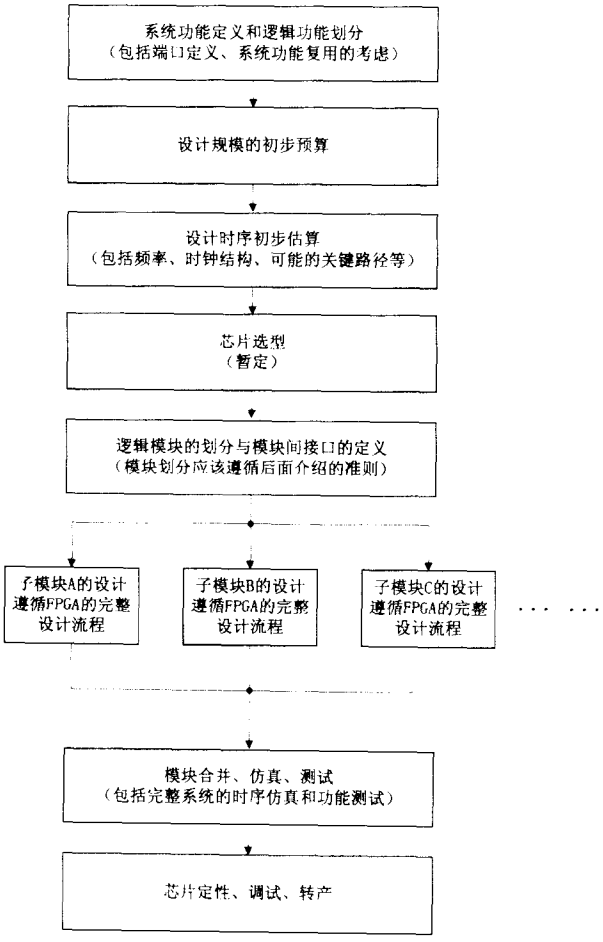

图5-6　系统规划流程图

下面简单谈谈模块化设计方法。模块化设计是系统原则的一个很好的体现，它不仅仅是一种设计工具，更是一种设计思路、设计方法。它是由顶向下、模块划分、分工协作设计思路的集中体现，是当代大型复杂系统的推荐设计方法。目前很多 EDA 厂商都使用这类工具划分每个模块的设计区域，然后单独设计和优化每个模块，最后将每个模块融合到顶层设计中，从而实现了团队协作、并行设计的模块化设计思想。合理使用模块化设计，能在最大程

度上继承以往的设计成果，并行分工协作，有效利用开发资源，缩短开发周期。

【例 5-4】 在系统层次复用模块。

利用“可编程匹配滤波器”实现 WCDMA 基站的方案，其核心是在合理规划系统的基础上，通过合理划分模块并安排操作时序，提高单元模块的复用率，从而大大降低硬件消耗，其设计思想是系统原则的集中体现。可编程匹配滤波器工作原理框图如图 5-7 所示。

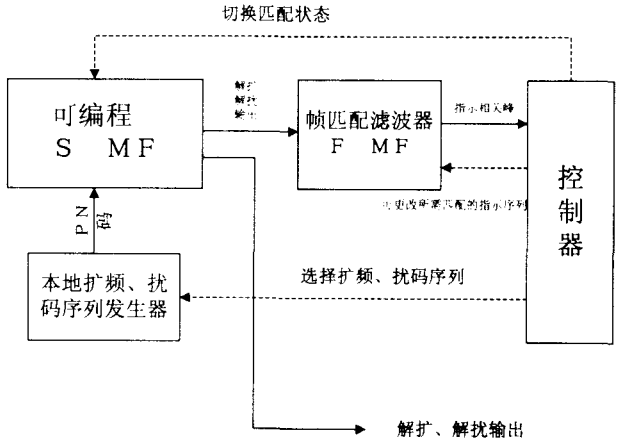

图5-7 可编程匹配滤波器工作原理框图

该原理图体现的设计思想是，利用信道固有的特点（如信道 pilot 导频、信道结构等），应用现代可编程数字信号处理技术（如 DSP、FPGA 等），采取反馈与控制匹配滤波方式，实现对某信道已扩信息的自动解扩解扰。该可编程匹配滤波器的主要组成部分为本地码发生器、可编程信号 MF（S MF）、帧匹配滤波器（FRAME MF）和控制器等。本地码发生器可生成各种所需的扩频、加扰序列，可接收控制器的指示脉冲，产生规定的本地解扩、解扰序列，作为 S MF 的参考序列。S MF 是完成匹配滤波的主体，可接收控制器的指示脉冲，将自己的匹配状态切换到下一匹配状态；F MF 完成对导频信号等特殊信号（信息比特待选集有限）的检测，生成指示相关峰值，通知控制器将 S MF 切换到下一匹配状态；控制器统一协调各部分工作。这种可编程滤波器可以在越区切换、同步方面、CPCH 收发信机等各领域中应用，如果为其适当安排时序流程，可以节约硬件资源。

5.2 同步设计原则和多时钟处理

本节重点阐述同步设计原则和多时钟设计技术。

5.2.1 同步设计原则

同步设计是 PLD 和 ASIC 设计的最重要原则。本节首先阐释为什么在 PLD 设计中要采用同步时序设计，然后重点论述同步时序设计的要点。

一、异步时序设计与同步时序设计

简单比较一下异步电路和同步电路的异同。

1. 异步电路

- 电路的核心逻辑用组合电路实现，比如异步的 FIFO/RAM 读写信号、地址译码等电路。
- 电路的主要信号、输出信号等并不依赖于任何一个时钟性信号，不是由时钟信号驱动触发器（FF）产生的。
- 异步时序电路的最大缺点是容易产生毛刺。在布局布线后仿真和用高分辨率逻辑分析仪观测实际信号时，这种毛刺尤其明显。
- 不利于器件移植，包括 FPGA 器件族之间的移植和从 FPGA 向结构化 ASIC 的移植。
- 不利于静态时序分析（STA）和验证设计时序的性能。

2. 同步时序电路

- 电路的核心逻辑用各种各样的触发器实现。
- 电路的主要信号、输出信号等都是由某个时钟沿驱动触发器产生的。
- 同步时序电路可以很好地避免毛刺。布局布线后仿真和用高速逻辑分析仪采样实际工作信号时皆无毛刺。
- 有利于器件移植，包括 FPGA 器件族之间的移植和从 FPGA 向结构化 ASIC 的移植。
- 有利于静态时序分析（STA）和验证设计时序的性能。

早期 PLD 设计经常使用行波计数器（Ripple Counters）或者异步脉冲生成器等典型的异步逻辑设计方式，以节约设计所消耗的面积资源。但是异步逻辑设计的时序正确与否完全依赖于每个逻辑元件和布线的延时，所以其时序约束相对繁杂而困难，并且极易产生亚稳态、毛刺，造成设计稳定性下降和设计频率不高等问题。随着数字逻辑的不断经济化，器件资源已经不再成为设计的主要矛盾，而同步时序电路对全面提高设计的频率和稳定性至关重要，从这个层面上讲，同步时序电路更为重要。

另一方面，随着 PLD 和 ASIC 逻辑规模的不断扩大，在芯片中完成复杂且质量优良的异步时序设计过于费时费力，其所需调整的时序路径和需要附加的相关约束相当繁琐，异步时序方法是与可编程设计理念背道而驰的。

随着 EDA 工具的发展，大规模设计综合、实现工具的优化效果越来越强。但是目前大多数综合、实现等 EDA 工具都是基于时序驱动（Timing Driven）优化策略的。异步时序电路增加了时序分析的难度，确定最佳时序路径所需的计算量难以想象，所需的时序约束相当繁琐，而且很多综合、实现工具的编译会带来歧义。而同步时序设计则恰恰相反，其时序路径清晰，相关时序约束简单明了，综合、实现工具优化容易，布局布线计算量小。所以推荐使用同步时序设计。

综上所述，现代数字芯片设计推荐采用同步时序设计方式。

二、同步时序设计

同步时序设计的基本原则是使用时钟沿触发所有的操作。如果所有寄存器的时序要求（Setup、Hold 时间等指标）都能够满足，则同步时序设计与异步时序设计相比，在不同的 PVT（工艺、电压、温度）条件下能获得更佳的系统稳定性与可靠性。

同步设计中，稳定可靠的数据采样必须遵从以下两个基本原则：

- 在有效时钟沿到达前，数据输入至少已经稳定了采样寄存器的 Setup 时间之久，这条原则简称为满足 Setup 时间原则；
- 在有效时钟沿到达后，数据输入至少还将稳定保持采样寄存器的 Hold 时间之久，这条原则简称为满足 Hold 时间原则。

同步时序设计有以下几个注意事项。

- 异步时钟域的数据转换，详见本章 5.2.3 小节“异步时钟域数据同步”。
- 组合逻辑电路的设计方法，详见本章 5.5 节“组合逻辑的注意事项”。
- 同步时序电路的时钟设计，详见本章 5.6 节“时钟设计的注意事项”。
- 同步时序电路的延时。最常用的设计方法是用分频、倍频的时钟或者同步计数器完成所需延时。换句话说，同步时序电路的延时被当做一个电路逻辑来设计。对于比较大的和有特殊定时要求的延时来说，一般用高速时钟产生一个计数器，根据计数器的计数控制延时；对于比较小的延时，则可以用 D 触发器打一下，这种做法不仅使信号延时了一个时钟周期，而且还完成了信号与时钟的初次同步，该方法通常在输入信号采样或增加时序约束余量时使用。另外，还有许多初学者用行为级（Behavior Level）方法描述延时，如“#5 a<=4'b0101;”这种行为级描述方法常用于仿真测试激励，但是在电路综合时将会被忽略，并不能起到延时的作用。

5.2.2 亚稳态

异步时钟域数据转换的核心就是要保证下级时钟对上级数据采样的 Setup 时间和 Hold 时间。如果触发器的 Setup 时间或者 Hold 时间不能得到满足，则有可能会产生亚稳态，此时触发器输出端 Q 在有效时钟沿到达之后比较长的一段时间内将处于不确定的状态。在这段时间内 Q 端将会产生毛刺并不断振荡，最终固定在某一电压值上，此电压值并不一定等于原来数据输入端 D 的数值，这段时间称为决断时间（Resolution time）。经过决断时间之后，Q 端将稳定到 0 或 1 上，但是究竟是 0 还是 1，这是随机的，与输入没有必然的关系，如图 5-8 所示。

亚稳态的危害主要体现在破坏系统的稳定性上。由于输出在稳定下来之前可能是毛刺、振荡、固定的某一电压值，因此亚稳态将导致逻辑误判，严重情况下输出 0～1 之间的中间电压值还会使下一级产生亚稳态，即导致亚稳态的传播。逻辑误判将导致功能性错误，而亚稳态的传播则扩大了故障面。另外，在亚稳态状态下，任何诸如环境噪声、电源干扰等细微的扰动都将导致更恶劣的不稳定状态，这时这个系统的传输延时将会增大，状态输出错误，在某些情况下甚至会使寄存器在两个有效判定门限（VoL、VoH）之间长时间的振荡。

只要系统中有异步元件存在，亚稳态就无法避免，因此在设计电路时，首先要减少由亚稳态而导致的错误；其次要使系统对产生的错误不敏感。前者要靠同步设计来实现，而后者则根据不同的设计应用采用不同的处理方法来解决。

使用两级以上寄存器采样可以有效地降低亚稳态继续传播的概率。在图 5-9 中，左边所示为异步输入端，经过两级触发器采样，使右边的输出与 bclk 同步，而且该输出基本不存

在亚稳态。其原理是即使第一个触发器的输出端存在亚稳态，但经过一个 Clk 周期后，第二个触发器 D 端的电平仍未稳定的概率会非常小，因此第二个触发器 Q 端基本不会产生亚稳态。如果再添加一级寄存器，使同步采样达到 3 级，则末级输出为亚稳态的概率几乎为 0。

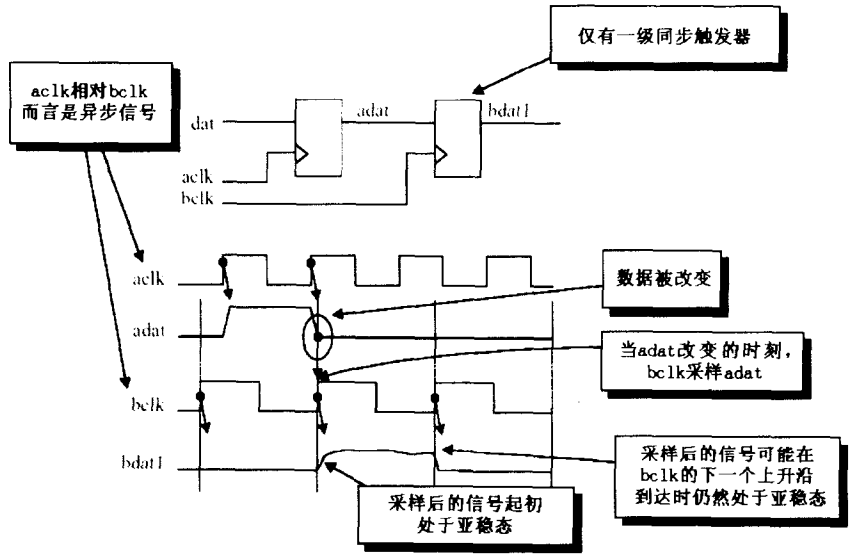

图5-8　亚稳态示意图

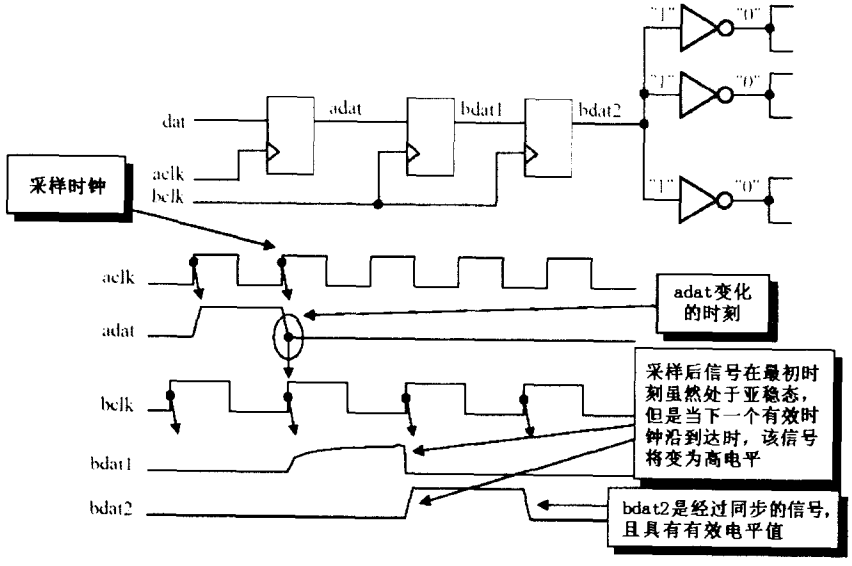

图5-9　两级寄存器采样可以降低亚稳态继续传播的概率

使用如图 5-9 所示的两级寄存器采样仅能降低亚稳态的概率，但是并不能保证第二级输出的稳态电平就是正确电平。前面说过，经过决断时间之后，寄存器输出的电平是一个不确定的稳态值，也就是说这种处理方法并不能排除采样错误的产生，这时就要求所设计的系统对采样错误要有一定的容忍度。有些应用本身就对采样错误不敏感，如一帧图像编码和一段话音编码等；而有些系统则对错误采样比较敏感。这类由于亚稳态而造成的采样错误是一些

突发性错误，所以可以采用一些纠错编码完成对错误的纠正。

5.2.3 异步时钟域数据同步

异步时钟域数据同步是芯片设计中一个常见的问题，该问题既是一个重点，也是一个难点。很多设计的不稳定都源于异步时钟域数据同步的不稳定。

一、两类异步时钟域同步的表现形式

异步时钟域数据同步也称为数据接口同步，顾名思义，是指如何在两个时钟不同步的数据域之间可靠地进行数据交换。数据时钟域不同步的情况主要有以下两种：

- 两个域的时钟频率相同，但是相位差不固定，或者相位差固定，但是不可测，简称为同频异相问题；
- 两个时钟域频率不同，简称为异频问题。

二、两种不推荐的异步时钟域操作方法

首先讨论两种在设计中不推荐使用的异步时钟域转换方法。一种是通过增加 Buffer 或者其他门延时调整采样；另一种是盲目使用时钟正负沿调整数据采样。

(1) 通过 Buffer 等组合逻辑延时线调整采样时间。

在早期的逻辑电路图设计阶段，有些设计者养成了手工加入 Buffer 或者非门调整数据延时的习惯，以保证本级模块的时钟对上级模块数据的建立及保持时间的要求。这些做法目前主要应用于两种场合，一种是使用分立逻辑元件（如 74 系列）搭建数字逻辑电路；另一种是在 ASIC 设计领域。使用分立逻辑元件搭建数字逻辑电路的场合一般为系统复杂度相对较低，系统灵活性要求不高的场合。使用分立逻辑元件设计数字逻辑电路时，由于可以使用的延时调整手段较少，而且一般设计频率较低，时序余量较大，因此采用插入 Buffer、非门等单元调整延时的手段是可以接受的。在 ASIC 设计领域中采用这种方法是以严格的仿真和约束条件作为支持的，而在大多数数字逻辑设计，特别是在 FPGA/CPLD 等可编程逻辑设计中，这种方法是应该坚决避免的，其原因在于，Buffer 和非门等单元是组合逻辑，正如本章 5.2.1 小节“同步设计原则”所述，使用组合逻辑方法产生延时，容易产生毛刺，而且这种设计方法的时序余量较差，一旦外界条件变换（环境试验，特别是高低温试验），采样时序就有可能完全紊乱，造成电路瘫痪。另外，一旦芯片更新换代，或者被移植到其他器件族的芯片上，就必须对采样延时重新进行调整，电路的可维护性和继承性都很差。

(2) 盲目使用时钟正负沿调整数据采样。

很多初学者习惯随意使用时钟的正负沿来调整采样，甚至还会通过产生一系列不同相位或不同占空比的时钟来使用其正负沿调整数据，这种做法是不推荐的，原因如下。

- 第一，如果在一个时钟周期内使用时钟的双沿同时进行操作，则使用该时钟的同相倍频时钟也能实现相同的功能。换句话说，一个时钟周期内，使用时钟的双沿同时操作，相当于使用了一个同相的倍频时钟。此时由于设计的时钟频率提升，所有相关的使用约束都会变得更紧，不利于可靠实现。
- 第二，芯片中的 PLL 和 DLL 一般都能较好地保证某个时钟沿的 Jitter、Skew 和占空比等各种参数指标，而对于另一个时钟沿的指标控制则并不是那么严

格。特别是对于综合、实现等 EDA 软件来说，如果没有明确对另外一个沿进行约束，那么这个沿的时序分析就不一定完善，其综合或实现结果就不一定能严格满足用户的时序要求（比如 Setup、Hold 时间等），这样往往会产生在该沿操作不稳定的结果。

总结这两点，如果设计者并不十分清楚同一周期内使用时钟双沿的注意事项，则同时使用上下沿，还不如直接使用同相倍频时钟更加简单、明确、可靠。但是如果设计者十分清楚同一周期内使用双沿的注意事项，并附加了相应的约束，则这种做法并非不可。

这里还要补充以下两点。

- 虽然使用了同一个时钟的两个沿，但是如果没有在同一个周期内同时使用双沿，则不会增加时钟频率。
- DDR、QDR 本身就是利用了上下沿采样的原理。随着存储器件的高速发展，时钟速度已经成为存取器件的瓶颈，所以可用时钟上下沿操作缓解对单沿 RAM 时钟振荡器的要求。但是必须清楚，硬件的 DDR、QDR 电路（包括 ASIC 的 DDR 和 QDR、FPGA 内嵌的 DDR 和 QDR 电路）是专用的高速设计电路，对时钟正沿、负沿的 Jitter、Skew 和占空比等指标都有明确的要求，这一点与普通逻辑设计，特别是 FPGA 中的设计是截然不同的，希望读者加以区分。

三、异步时钟域数据同步问题的解决方法

下面分别介绍本章 5.2.3 小节中提出的两大异步时钟域数据同步问题的解决方法。

(1) 同频异相问题

同频异相问题的简单解决方法是用后级时钟对前级数据采样两次，即通常所述的用寄存器打两次，如图 5-10 所示。这种做法可以有效地减少亚稳态的传播，使后级电路数据都是有效电平值。但是这种做法并不能保证两级寄存器采样后的数据是正确的电平值，因为一旦 Setup 或 Hold 时间不满足，采样发生亚稳态，则经决断时间后，还是可能判决到错误的电平值。所以这种方法仅仅适用于对少量错误不敏感的功能单元。

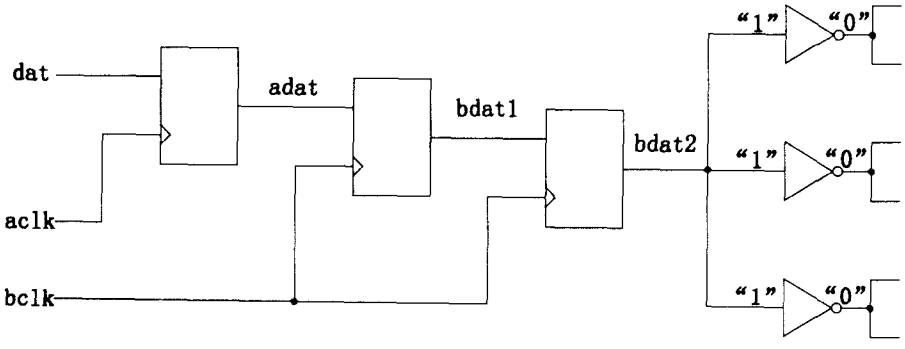

图5-10　数据同步

可靠的做法是用 DPRAM、FIFO 或者一段寄存器 Buffer 完成异步时钟域的数据转换。将数据存入 DPRAM 或 FIFO 的方法是，将上级芯片提供的数据随路时钟作为写信号，将数据写入 DPRAM 或者 FIFO，然后使用本级的采样时钟

（一般是数据处理的主时钟）将数据读出。由于时钟频率相同，DPRAM 或 FIFO 两端的数据吞吐率一致，因此实现起来相对简单一些。

(2) 异频问题

解决异频问题的可靠方法就是使用 DPRAM 或 FIFO。其实现思路与前面所述一致，用上级随路时钟写入上级数据，然后用本级时钟读出数据。但是由于时钟频率不同，所以两个端口的数据吞吐率不一致，设计时一定要开好缓冲区，并通过监控（Full、Half、Empty 等指示）确保数据流不会溢出。

5.3 代码风格

代码风格即 Coding Style，主要分为一般性代码风格和针对综合工具、实现工具、器件类型的代码风格两种。

5.3.1 代码风格的分类

所谓一般性的代码风格是指不依赖于综合、实现工具和器件类型的代码风格。不同的综合实现工具对一些语法细节的阐释略有不同，特别是那些关于优先级、实现的先后顺序等内容，所以不同的综合工具在个别细节上对代码风格的解释有一定的差异。本章所述的 Coding Style，如果没有特别声明，一般都是指不依赖于 EDA 工具类型，适用于不同目标器件的一般意义上的 Coding Style。

还有一类代码风格是针对某种综合工具，或者特定器件结构的，根据器件硬件结构，正确地实例化底层单元模块，合理地使用其内嵌的硬件电路，以达到最优化的设计效果。另外需要特别声明一点，一般 ASIC 设计的代码风格和 PLD（主要指 FPGA 和 CPLD）设计的代码风格有明显差异，这主要是因为 PLD 设计是基于固有的硬件结构（如逻辑单元、块 RAM、PLL/DLL、时钟资源等）的。而 ASIC 设计结构灵活，目标多样，特别是在功耗、速度、时序等要求上，与 PLD 设计有明显差异。例如 ASIC 设计中根据要求会有意识地采样某些组合逻辑、门控时钟等，以降低功耗或提高速度。本文中所述的代码风格和基本原则如不特殊声明，均基于 PLD 设计要求。

另外，本章所述代码风格主要是基于 RTL（寄存器传输级）级而言的，并非其他描述层次。所以诸如业界炒的非常热的结构化设计方法（Architectural-based Design）代码风格与本章无关，它的很多原则和方法是与本章所述背道而驰的。

5.3.2 代码风格的重要性

当代的可编程逻辑设计日趋复杂，其系统复杂度和设计频率要求与 5 年前不可同日而语。良好的代码风格对于设计的工作频率、所消耗的芯片面积，甚至是整个系统的稳定性都非常重要，良好、规范的代码风格将有利于设计的移植。

随着 EDA 技术的不断发展，综合、实现工具的优化能力越来越强，可以自动完成许多复杂设计的面积和时序方面的优化，并且其优化效果日趋先进。但是如果读者盲目依赖综合、实现等 EDA 工具的优化，而忽略了自己设计的代码风格，就大错特错了。代码风格对

综合、实现等 EDA 工具的优化结果所产生的影响可以用这样一句话来概括：“好的代码风格会使综合、实现等优化事半功倍，达到最优化的结果；不良的代码风格会使综合、实现等优化南辕北辙，甚至产生错误的结果。”所以读者必须明确，综合、实现等 EDA 工具的优化能力和正确性最终取决于所设计的代码风格的优劣。

5.4 结构层次设计和模块划分

5.4.1 结构层次化编码（Hierarchical Coding）

结构层次化编码是模块化设计思想的一种体现。目前大型设计中必须采用结构层次化编码风格，以提高代码的可读性。它易于模块划分，易于分工协作，易于设计仿真测试激励。最基本的结构化层次是由一个顶层模块和若干个子模块构成的，每个子模块根据需要还可以包含自己的子模块。结构层次化编码示意图如图 5-11 所示。

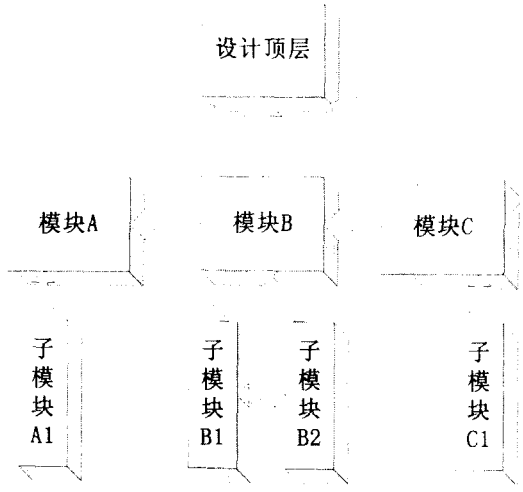

图5-11　结构层次化编码示意图

结构层次化编码有以下几条注意事项。

- 结构的层次不易太深，一般为 3～5 层即可。在综合时，一般综合工具为了获得更好的综合效果，特别是为了使综合结果所占用的面积更小，会默认将 RTL 代码的层次打平（Flatten）。有时为了在综合后仿真和布局布线后的时序仿真中能够较为方便地找出一些中间信号，比如子模块之间的接口信号等，还可以在综合工具中设置保留结构层次，以便于仿真信号的查找和观察。
- 顶层模块最好仅仅包含对所有模块的组织和调用，而不应该完成比较复杂的逻辑功能。较为合理的顶层模块由输入输出管脚声明、模块的调用与实例化、全局时钟资源、全局置位/复位、三态缓冲和一些简单的组合逻辑等构成。
- 所有的 I/O 信号，如输入、输出、双向信号等的描述都在顶层模块中完成。
- 子模块之间也可以有接口，但是最好不要建立子模块间跨层次的接口，例如上

图中模块 A1 到模块 B1 之间不宜直接连接，两者需要交换的信号可以通过模块 A、模块 B 的接口传递，这样做的好处是增加了设计的可读性和可维护性。

- 应该综合考虑子模块的功能、结构、时序、复杂度等多方面因素对子模块进行合理划分。

结构层次化编码的本质不应该简单地理解为一种具体的设计手段，而应该认识到它其实更是一种系统层次的设计方法。很多初学者都有这样一个疑问，一个设计完全可以在同一个模块内完整描述，为什么还要将其中的时序逻辑、组合逻辑、不同优化目标、不同功能的电路分成多级层次，使用多个模块描述呢？模块划分增加了内部接口描述的工作量，这不是“自讨苦吃”么？虽然在理论上，任何设计都可以在同一个模块中完成，但是如果将不同功能、不同层次、不同类型的电路混淆在同一个模块中，则不是一种好的系统设计方法，对于比较复杂的设计来说，将会导致整个设计杂乱无章，不利于设计的阅读与维护，也会给综合和实现过程带来许多麻烦。

5.4.2 模块划分的技巧（Design Partitioning）

结构层次化设计方法的第一个要点就是模块划分，模块划分非常重要，关系到能否最大程度上发挥项目成员协同设计的能力，更重要的是它直接决定着设计的综合、实现步骤所消耗的时间及其效率。模块划分的基本原则如下。

(1) 对每个同步时序设计的子模块的输出使用寄存器（Registering）。

这种做法有时也被称为用寄存器分割同步时序模块原则。使用寄存器分割同步时序单元的好处是，便于综合工具比较所分割的子模块中的组合电路和同步时序电路，从而达到更好的时序优化效果，这种模块划分符合时序约束的习惯，便于利用约束属性进行时序约束。

(2) 将相关的逻辑或者可以复用的逻辑划分在同一模块内。

这种做法有时也被称为呼应系统原则。这样做的好处是，一方面将相关的逻辑和可以复用的逻辑划分在同一模块内，以便在最大程度上复用资源，减少设计所消耗的面积；另一方面有利于综合工具优化某个具体功能的时序关键路径。其原因是，传统的综合工具只能同时优化某一部分的逻辑，而它所能同时优化的逻辑的基本单元就是模块，所以将相关功能划分在同一模块内，可以在时序和面积上获得更好的综合优化效果。

(3) 将不同优化目标的逻辑分开。

在介绍速度和面积的平衡与互换原则时，谈到合理的设计目标应该综合考虑面积最小和频率最高两个指标。好的设计，在规划阶段设计者就应该初步规划了设计的规模和时序关键路径，并对设计的优化目标有一个整体上的把握。对于时序紧张的部分，应该将其独立划分为一个模块，其优化目标为“速度”，这种划分方法便于设计者进行时序约束，也便于使用综合和实现工具进行优化。目前很多综合与实现工具都支持物理区域位置约束，以模块为单元进行物理区域约束，从而优化关键路径时序，以达到更高的系统工作频率。另一类情况是，设计的矛盾主要集中在芯片的资源消耗上。这时应该将资源消耗过大的部

分划分为独立的模块，这类模块的优化目标应该定为“面积”。同理，将它们规划到一起，更有利于区域布局与约束。这种根据优化目标进行优化的方法其最大的好处是，对于某个模块综合器仅仅需要考虑一种优化目标和策略，从而比较容易达到较好的优化效果。相反，如果同时考虑两种优化目标，会使综合器陷入互相制约的困境。

(4) 将时序约束较松的逻辑归入同一模块。

有些逻辑的时序非常宽松，不需要较高的时序约束，这样可以将这类逻辑归入同一模块，如多周期路径（Multi-cycle Path）等。将这些模块归类，并指定松约束，则可以帮助综合器尽量节省面积资源。

(5) 将存储逻辑独立划分成模块。

应该将 RAM、ROM、CAM 和 FIFO 等存储单元划分成独立的模块。这样做的一个好处是便于利用综合约束属性显化指定这些存储单元的结构和所使用的资源类型，也便于综合器将这些存储单元自动类推为指定器件的硬件原语。另一个好处是在仿真时消耗的内存也会少一些，便于提高仿真速度。这是因为大多数仿真器对大面积的 RAM 都有独特的内存管理方式，以提高仿真效率。

(6) 合适的模块规模。

从理论上讲，模块的规模越大，越利于模块资源共享（Resource Sharing）。但是庞大的模块会要求综合器同时处理更多的逻辑结构。这对综合器的处理能力和计算机的配置提出了较高的要求。另外，庞大的模块划分不利于发挥目前非常流行的增量综合与实现技术的优势。

5.5 组合逻辑的注意事项

相对复杂一些的设计都是由两部分组成的，这两个部分分别为时序逻辑（Sequential Logic）和组合逻辑（Combination Logic）。同步时序设计系统中并不是不包含组合逻辑，而是要更加合理地设计、划分组合逻辑。下面几节将介绍组合逻辑设计的一些问题。

5.5.1 always 组合逻辑信号敏感表

几乎所有的编码指导手册都有关于信号敏感表的论述。时序逻辑的信号敏感表比较好写，只要在信号敏感表中写明时钟信号的正负触发沿即可。信号敏感表的主要问题集中在组合逻辑信号敏感表的写法上。

- 正确的信号敏感表设计方法是将 **always** 模块中使用到的所有输入信号和条件判断信号都列在信号敏感表中。
- 希望通过信号敏感表的增减完成某项逻辑功能是大错特错的。
- 不完整的信号敏感表有时会造成综合前的仿真结果与综合后仿真、布局布线后仿真的结果不一致。
- 一般综合工具对于不完整的信号敏感表的默认做法是，将处理进程中用到的所有输入和判断条件信号都默认添加到综合结果的信号敏感表中，并发出警告

（warning）信息，提示用户该进程的信号敏感表不完整。

有些初学者发现在信号敏感表中增减一些信号，会得到不同的仿真结果，于是企图依靠修改信号敏感表来完成某些逻辑设计，这种做法是大错特错的。其实一般综合工具的默认操作都是将 **always** 模块中使用到的所有输入信号和条件判断信号当做触发信号，综合到信号敏感表中，所以增减信号敏感表后得到的综合结果其实是完全一致的。之所以在增减信号敏感表后得到不同的仿真结果，是因为仿真器的工作机制所致，大多数仿真器是由数据流和时钟周期驱动的，如果信号敏感表中没有某个信号，则无法触发和该信号相关的仿真进程，从而得到不同的仿真结果。

5.5.2 组合逻辑反馈环路

组合逻辑反馈环路是数字同步逻辑设计的大忌，它最容易因振荡、毛刺、时序违规等问题引起整个系统的不稳定和不可靠。

图 5-12 所示即为一个典型的组合逻辑反馈环路，寄存器的 Q 输出端直接通过组合逻辑反馈到寄存器的异步复位端，如果 Q 输出为 0，经组合逻辑运算后为异步复位端有效，则电路将会进入不断清零的死循环。

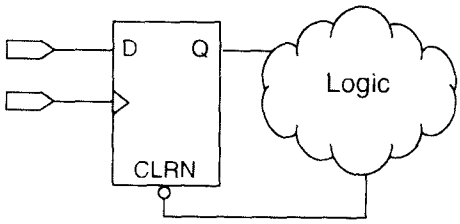

图5-12　组合逻辑反馈环路示意图

图 5-13 所示是一个比较常见的鉴相电路，用于配合 VCO 实现锁相环功能。这是一个典型的组合逻辑环电路。但是，如果这个电路是在 PLD 中实现的，则常常会引起一些非常特殊的问题，甚至是在一个已经非常成熟的设计中，由于芯片批次的改变，都有可能使其不能工作。因此，笔者强烈建议尽量不要在同步数字设计中出现这样的电路。

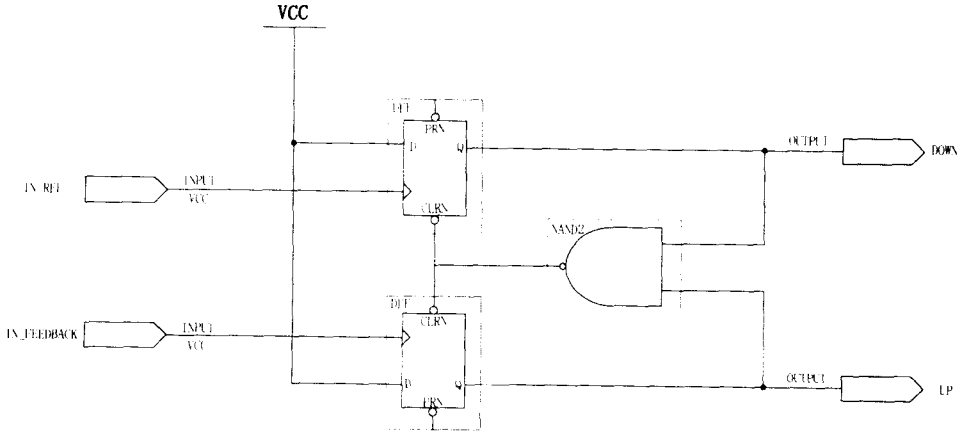

图5-13　鉴相逻辑

组合逻辑反馈环路是一种高风险的设计方式，主要原因如下。

- 组合反馈环的逻辑功能完全依赖于其反馈环路上组合逻辑的门延时和布线延时等，如果这些传播延时有任何改变，则该组合反馈环单元的整体逻辑功能将彻底改变，而且改变后的逻辑功能很难确定。
- 组合反馈环的时序分析是无穷循环的时序计算，综合、实现等 EDA 工具迫不得已一般必须主动割断其时序路径，以完成相关的时序计算。而不同的 EDA 工具对组合反馈环的处理方法各不相同，所以组合反馈环的最终实现结果有很多不确定因素。

同步时序系统中应该避免使用组合逻辑反馈环路，具体操作方法主要有以下两种。

- 牢记任何反馈环路必须包含寄存器。
- 检查综合、实现报告的 Warning 信息，发现 Combinational Loops 后立即进行相应的修改。

5.5.3 脉冲产生器

在异步时序设计中，常用延时链（Delay Chains）产生脉冲。常用的异步脉冲产生方法如图 5-14 所示。

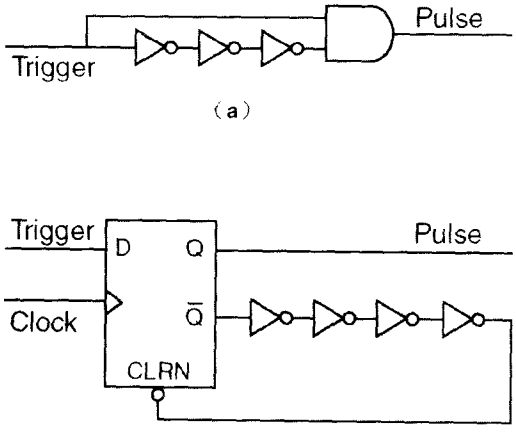

图5-14　常用的异步脉冲产生方法示意图

这类异步方法设计的脉冲产生电路的脉冲宽度取决于 Delay Chains 的门延时和线延时。而在 FPGA/CPLD 中，大多数 Timing Driven 的综合、布线工具无法保证其布线延时恒定。另外，PLD 器件本身在不同的 PVT（工艺、电压、温度）环境下其延时参数也有微小的波动，所以脉冲宽度无法确定。而且 STA（静态时序分析）工具也无法准确分析脉冲的特性，这为时序仿真和验证带来了很多不确定性。

异步脉冲序列产生电路（Multi-vibrators）又称为毛刺生成器（Glitch Generator），它利用组合反馈环路振荡而不断产生毛刺。正如前面所述，组合反馈环是同步时序必须避免的，这类基于组合反馈环的 Multi-vibrator 也会给设计带来稳定性、可靠性等方面的问题，必须避免使用。

常用的同步脉冲产生方法如图 5-15 所示。该设计的脉冲宽度不因器件改变或设计移植而改变，恒等于时钟周期，避免了异步设计的诸多不确定因素，其时序路径便于计算、STA 分析和仿真验证。

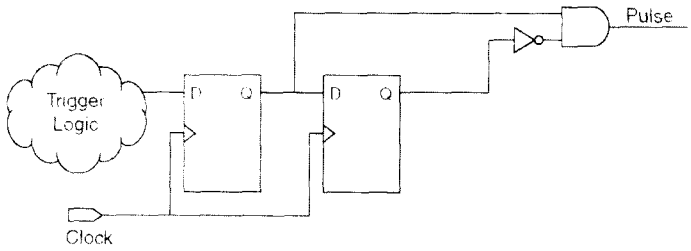

图5-15 常用的同步脉冲产生方法示意图

5.5.4 慎用锁存器（Latch）

同步时序设计要尽量慎用 Latch，特别要注意，如果综合出与设计意图不一致的 Latch，则将导致仿真和设计错误。对于一定要使用 Latch 的特定设计，设计者必须明确该 Latch 是否为有意设计的。综合出与设计意图不吻合的 Latch 结构的主要原因在于，在设计组合逻辑时使用不完全的条件判断语句，如有 if 而没有 else，使用不完整的 case 语句（这仅仅是一种可能，并不一定生成 Latch），或者设计中有组合逻辑的反馈环路（combinatorial feedback loops）等异步逻辑。

【例 5-5】 因 if 语句不完整而生产 Latch 的典型情况，代码参见随书光盘中“Example-5-5\source”目录下的相关内容。

典型的生成 Latch 的 Verilog 描述如下。

```
reg data_out;
always @(cond_1 or data_in)
  begin
    if (cond_1)
        data_out <= data_in;
  end
```

在上述描述中由于未指定在条件“cond_1”等于“0”时的动作，因此会生成如图 5-16 所示的 Latch 结构。读者可以打开随书光盘中的 Example-5-5 目录，分析并综合 latch.v 文件。

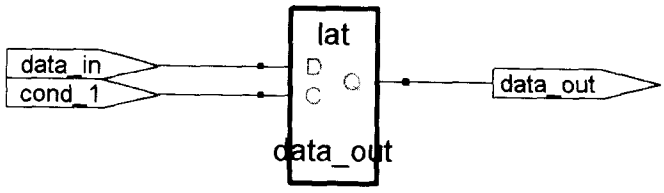

图5-16 Latch 的 RTL 示意图

防止产生非目的性 Latch 的措施如下。

- 使用完备的 **if...else** 语句。
- 检查设计中是否含有组合逻辑反馈环路。
- 为每个输入条件设计输出操作，为 **case** 语句设置 **default** 操作。特别是在状态机设计中，最好有一个 **default** 状态转移，而且每个状态最好也都有一个 **default** 操作。
- 在使用 **case** 语句，特别是在设计状态机时，尽量附加综合约束属性，综合为完全条件 **case** 语句（full case）。目前，大多数综合工具都支持 full case 的综合约束属性，具体的语法可参考综合工具的约束属性指南。

仔细检查综合器的综合报告，目前大多数的综合器对所综合出的 Latch 都会报“warning”，通过综合报告可以较为方便地找出无意中生成的 Latch。

5.6 时钟设计的注意事项

时钟是同步设计的基础，在同步设计中，所有操作都是基于时钟沿触发的，所以时钟的设计对于同步时序电路来说非常重要。在 PLD 设计中，通常推荐使用 FPGA 内嵌的 PLL 或 DLL 做时钟的频率与相位变化，并用全局时钟和专用时钟选择器进行时钟布线。而在 ASIC 设计中，常会用到各种各样的组合逻辑所产生的时钟，但是如果将这些设计直接移植到同步时序电路中，则将带来各种各样的问题。本节主要辨析一些常用时钟电路的优劣。

5.6.1 内部逻辑产生的时钟

如果需要使用内部逻辑产生时钟，则必须要在组合逻辑产生的时钟后插入寄存器，如图 5-17 所示。如果直接使用组合逻辑产生的信号作为时钟信号或者异步置位/复位信号，则将使设计不稳定。这是因为组合逻辑难免会产生毛刺，如果这些毛刺仅仅存在于同步时序的数据路径中，则寄存器采样会对毛刺产生过滤效应，一般来说负面影响并不大。但是当带有毛刺的信号作为时钟信号或者异步置位/复位信号时，如果毛刺的宽度足以驱动寄存器的时钟端或者异步置位/复位端，则必将产生错误的逻辑操作，即使毛刺的宽度不足以驱动时钟端或异步置位/复位端，也会引发寄存器的不稳定，甚至激发寄存器产生亚稳态。所以对于时钟路径来说，必须插入寄存器以过滤毛刺。

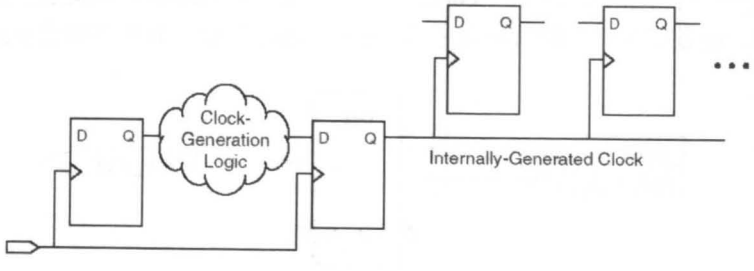

图5-17　内部时钟设计必须插入寄存器

另一方面，组合逻辑产生的时钟还会带来另外一个问题，组合逻辑电路的 Jitter 和 Skew

都比较大，如果时钟产生逻辑的延时比数据路径的延时更大，则会带来负的 Skew。负的 Skew 对于同步逻辑设计而言是灾难性的。所以使用组合逻辑来产生内部时钟仅仅适用于时钟频率较低、时钟精度要求不高的情况。另外，这类时钟应该使用快速布线资源布线，而且需要为组合逻辑电路附加一定的约束条件，以确保时钟质量。

5.6.2 Ripple Counter

所谓 Ripple Counter 即行波计数器，其结构是一组寄存器级连，每个寄存器的输出端都会接到下一级寄存器的时钟管脚。这种计数器常常用于异步分频电路。早期的 PLD 设计经常使用 Ripple Counter 以节约芯片资源。但是由于 Ripple Counter 是一种典型的异步时序逻辑，正如前面的“同步设计原则”所述，异步时序逻辑会带来各种各样的时序问题，所以在同步时序电路设计中必须严格避免使用 Ripple Counter。

5.6.3 时钟选择

在通信系统中，为了适应不同的数据速率要求，经常要进行时钟切换。有时为了节约功耗，可能也会把高速时钟切换为低速时钟，或者进行时钟休眠操作。切换时钟的最佳途径是使用芯片内部专用的 Clock MUX。这些 MUX 的反应速度快，锁定时间短，切换瞬间带来的冲击和抖动小。

如果所需器件没有专用的 Clock MUX，那么在进行时钟切换时应注意以下几点。

- 时钟切换控制逻辑在配置后将不再改变。
- 切换时钟后，将所有相关电路复位，以保证所有寄存器、状态机和 RAM 等电路的状态不会锁死或进入死循环。
- 设计系统对时钟切换过程中出现的短暂错误不敏感。

5.6.4 门控时钟

门控时钟即 Gated Clock，如图 5-18 所示，是 IC 设计中一种常用的减少功耗的手段。设计者通过对 Gating Logic 的控制，可以控制门后端的所有寄存器不再翻转，从而有效地节约功耗。

但是 Gated Clock 不是同步时序电路，其 Gating Logic（门控逻辑）会污染 Clock 的质量，产生毛刺，并使时钟的 Skew（偏斜）、Jitter（延时）等指标恶化。正如 “同步设计原则”一节所述，在同步时序电路中，应该尽量不使用 Gated Clock。

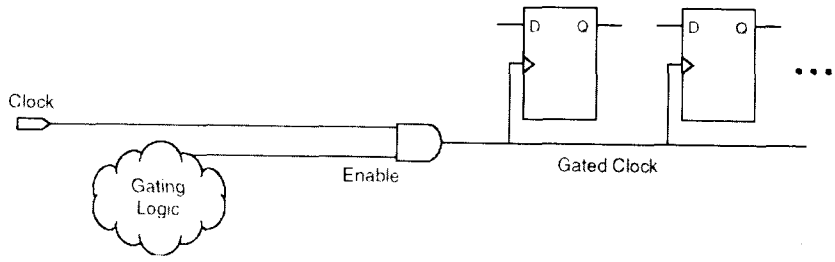

图5-18 门控时钟

虽然有些书籍中指出当功耗成为主要矛盾时，可以使用如图 5-19 所示的电路完成类似门控时钟的功能，但是笔者仍强烈建议在 PLD 设计中不要使用该图所示的门控时钟改进电路。虽然这个改进电路已经在较大程度上解决了门控电路产生毛刺的问题，但是请读者注意，这个电路工作的前提是时钟源 Clock 的占空比（Duty Cycle）是非常理想的 50%，如果时钟的占空比不能保证为 50%，则会产生许多有规律的毛刺信号。另外这个电路的使用还有一个前提，那就是 Clock 与 Enable 信号的布线 Skew 为 0，否则也会产生毛刺。

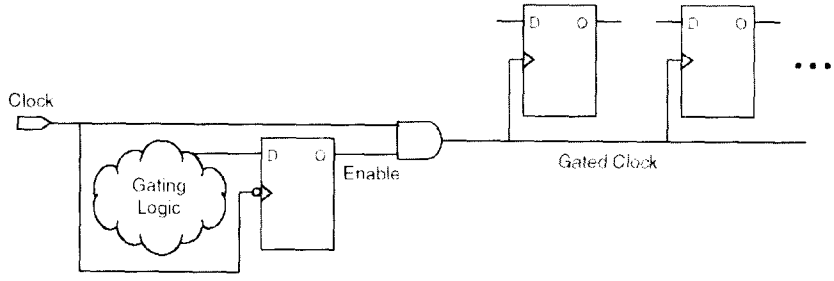

图5-19　门控时钟改进电路

如果功耗真的成为 PLD 设计的首要问题，那么建议采用其他方法减少功耗，如最近发展起来的低核电压芯片（Core 电压为 1.0V）、芯片休眠功能和 Clock MUX 等。

5.6.5　时钟同步使能端

大多数像寄存器这样的同步单元都支持时钟的同步使能（Synchronous Clock Enable）。需要注意的是，虽然使能无效时这些单元无输出，但是这种方法并不能像 Gated Clock 一样减少功耗。不过 Synchronous Clock Enable 能够非常方便地完成一些逻辑功能，通过使用同步时钟使能端完成某些逻辑功能，有时可以节约芯片面积并提高设计频率。

图 5-20 上半部分所示的同步使能功能，在目前大多数的器件上可以直接将使能信号连接到芯片的同步使能端实现，如图 5-20 下半部分所示。

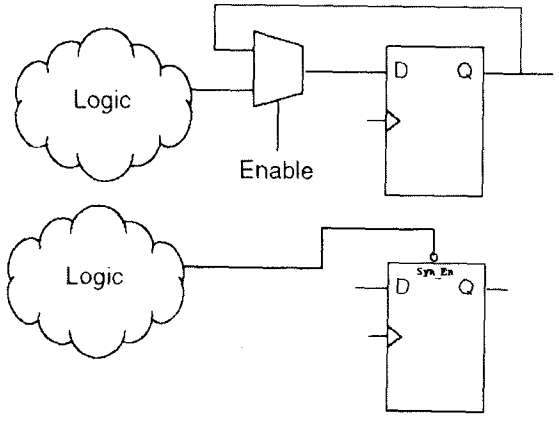

图5-20　同步使能端

5.7 RTL 代码优化技巧

5.7.1 使用 Pipelining 技术优化时序

Pipelining，即流水线时序优化方法，其本质是调整一个较长的组合逻辑路径中的寄存器位置，用寄存器合理分割该组合逻辑路径，从而降低对路径 Clock-To-Output 和 Setup 等时间参数的要求，达到提高设计频率的目的。但是必须要注意的是，使用 Pipelining 优化技术只能合理地调整寄存器的位置，而不应该凭空增加寄存器的级数，所以 Pipelining 有时也被称为 Register Balance（寄存器平衡）优化技术。

目前一些先进的综合工具能根据用户的参数配置，自动运用 Pipelining 技术，通过使用寄存器平衡设计中较长的组合路径（Register Balance），在一定程度上提高设计的工作频率。这种时序优化手段对乘法器、ROM 等单元效果显著。

5.7.2 模块复用与资源共享

本节的模块复用和资源共享（Resource Sharing）主要是站在微观的角度观察节约面积的问题。为了便于理解，下面举两个例子。

【例 5-6】 资源共享实例，一个补码平方器，代码参见随书光盘中“Example-5-6\source”目录下的相关内容。

这是一个补码平方器的例子，输入是 8bit 补码，求其平方和。由于输入是补码，所以当最高位是 1 时，表示原值是负数，需要按位取反，加 1 后再平方；当最高位是 0 时，表示原值是正数，直接求平方。

下面是两种描述方式，读者可判断一下优劣，并体会 Resource Sharing 的含义。

第一种实现方式：

```
module resource_share (data_in, square);
input [7: 0]   data_in;     //输入是补码
output [15: 0]  square;
wire  [7: 0]   data_bar;

assign data_bar = ~data_in + 1;
assign square=(data_in[7])? (data_bar*data_bar) :  (data_in*data_in);
endmodule
```

第二种实现方式：

```
module resource_share (data_in, square);
input [7: 0]   data_in;       //输入是补码
output [15: 0]  square;
wire  [7: 0]   data_tmp;
```

```
assign data_tmp = (data_in[7])? (~data_in + 1) :  data_in;
assign square = data_tmp * data_tmp;
endmodule
```

仔细观察一下可以发现，第一种实现方式需要两个 16bit 乘法器，同时平方，然后根据输入补码符号选择输出结果，其关键在于使用了两个乘法器，选择器在乘法器之后；第二种实现方法首先根据输入补码的符号将补码换算为正数，然后做平方，其关键在于选择器在乘法器之前，仅仅使用了一个乘法器，节约了资源。第二种实现方式与第一种实现方式相比其节约的资源有两部分，第一部分节约了一个 16bit 乘法器；第二部分，后者的选择器是 1bit 判断、8bit 输出，而前者的选择器是 1bit 判断、16bit 输出。

两种代码的硬件结构示意图如图 5-21 和图 5-22 所示。

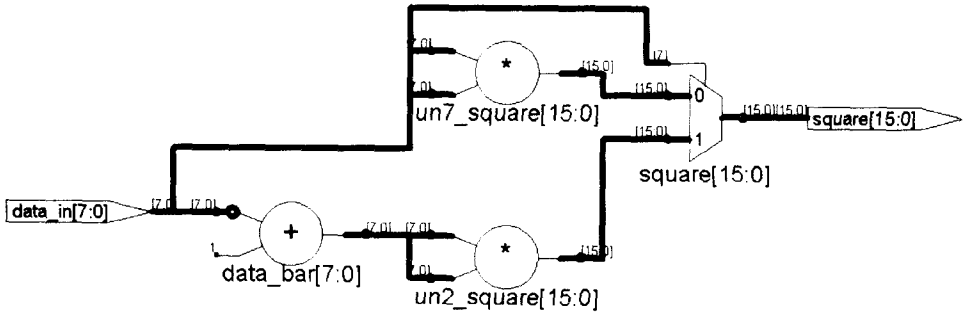

图5-21　两个乘法器的实现方案

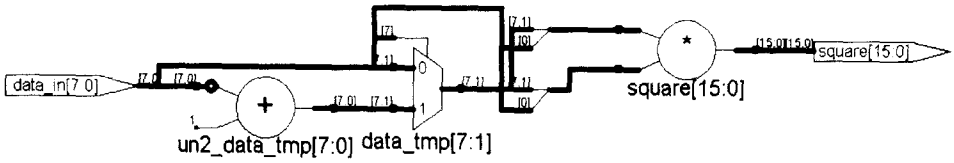

图5-22　一个乘法器的实现方案

细心的读者也许已经注意到，本例所示的结构示意图是 Synplify Pro 综合后提取的 RTL 视图。这里需要强调以下几点。

(1) 本例综合选用的工具是 Synplify Pro，目标器件为 Altera Cyclone EP1C3 –6 TC100，并关闭了“Resource Sharing”等所有优化参数。第一种实现方法所占用的逻辑资源为 213 个 LUTs；第二种实现方法所占用的逻辑资源为 76 个 LUTs，两者所占用的资源相差约 2 倍。

(2) 上例资源共享的单元是乘法器，通过 Resource Sharing，节省了一个乘法器和一些选择器占用的资源。其实如果拓展一下思维，将乘法器换成加法器、除法器等，甚至推广到任意逻辑模块，如果后续结构含有选择器，则都可以使用本例的设计思想，通过 Resource Sharing 成倍地节省前级模块所消耗的资源。

(3) 不同的综合工具、同一综合工具的不同版本、不同的优化参数、不同厂商的目标器件、同一厂商的不同器件族等因素都可能造成不同的综合结果。

(4) 目前很多综合工具都有“Resource Sharing”之类的优化参数，选择该参数，综合工具就会自动考察设计中是否有可以资源共享的单元，并在保证逻辑功能不变的情况下进行 Resource Sharing，以获得面积更小的综合结果。例如上例中打开 Synplify Pro 的“Resource Sharing”综合优化参数，Synplify Pro 就会自动运用资源共享的优化算法，其综合结果将与第二种代码描述的综合结果完全一致。

(5) 最后需要强调的是，不能因为综合工具优化能力的增强而片面依靠综合工具，放松对自己 Coding Style 的要求。这是因为第一，综合工具的优化力度毕竟有限，很多情况下不能智能地发现需要 Resource Sharing 的逻辑；第二，前面已经说过，“不同的综合工具、同一综合工具的不同版本、不同的优化参数、不同厂商的目标器件、同一厂商的不同器件族等因素”都会直接影响综合工具的优化能力和效果，所以综合工具的优化能力并不十分可靠；第三，在 ASIC 设计中，综合工具非常忠实于用户的意图，这时代码风格更加重要。所以逻辑工程师必须注意自己在代码风格方面的修养。

5.7.3 逻辑复制

逻辑复制是一种通过增加面积而改善时序条件的优化手段。逻辑复制最常使用的场合是调整信号的扇出。如果某个信号需要驱动后级的很多单元，换句话说，也就是其扇出非常大，那么为了增加这个信号的驱动能力，就必须插入很多级 Buffer，这样就在一定程度上增加了这个信号路径的延时。这时可以复制生成这个信号的逻辑，用多路同频同相的信号驱动后续电路，使平均到每路的扇出变低，这样不需要插入 Buffer 就能满足驱动能力增加的要求，从而节约了该信号的路径延时，如图 5-23 所示。

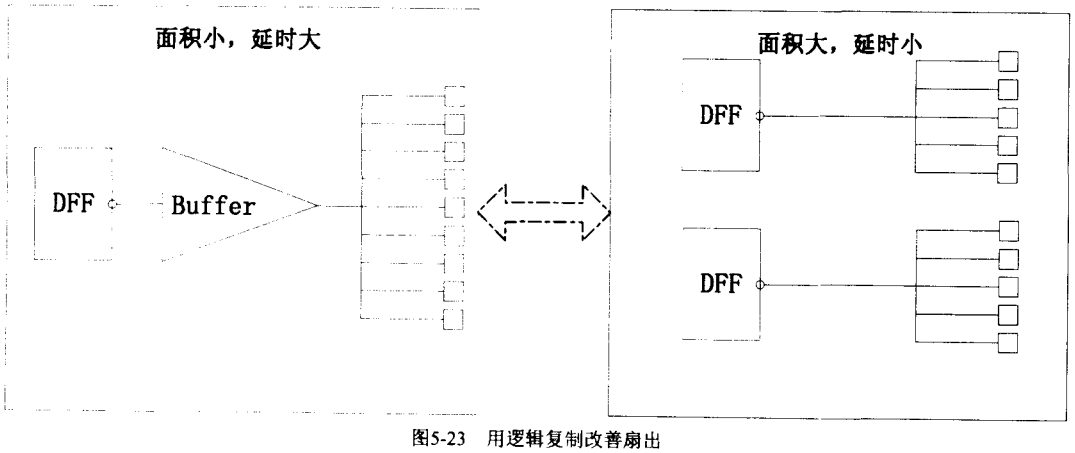

图5-23 用逻辑复制改善扇出

需要说明的是，现在很多综合工具都可以自动设置最大扇出值（Max Fanout），如果某个信号的扇出值大于最大扇出值，则该信号将会自动被综合工具复制。最大扇出值和器件的工艺密切相关，其合理值应该根据器件手册的声明和工程经验设置。这里举例用逻辑复制手段调整扇出，达到优化路径延时仅仅是为了讲述逻辑复制的概念，其实逻辑复制还有很多其

他形式，例如香农扩展（Shannon Expansion）等时序优化技术。

有的读者会有疑问，逻辑复制和资源共享是两个矛盾的概念，既然使用了资源共享优化技术，为什么还要做逻辑复制呢？

其实这个问题的本质还是面积与速度的平衡。逻辑复制与前面的资源共享是两个对立统一的概念。资源共享的目的是为了节省面积资源，而逻辑复制的目的是为了提高工作频率。当使用逻辑复制手段提高工作频率时，必然会增加面积资源，这是与资源共享相对立的方面；但是正如前面介绍的面积与速度的对立统一一样，逻辑复制和资源共享都是要达到设计目标的手段，一个侧重于速度目标，一个侧重于面积目标，两者存在一种转换与平衡的关系，所以两者又是统一的。

先来看下面一个例子。

【例 5-7】 一个加法器的资源共享实例，代码参见随书光盘中“Example-5-7\source”目录下的相关内容。

这个例子和前面乘法器的例子非常相似，只是将平方器换成了加法器。实现这个加法器也有两种代码写法，分别对应两种不同的硬件结构，如图 5-24 所示。

第一种写法对应左边的 RTL 结构示意图。

```
assign data_out= (sel)? (a+b) :  (c+d) ;
```

第二种写法对应右边的 RTL 结构示意图。

```
wire temp1, temp2;
assign temp1 = (sel)? (a) : (c) ;
assign temp2 = (sel)? (b) : (d) ;
assign data_out = temp1 + temp2;
```

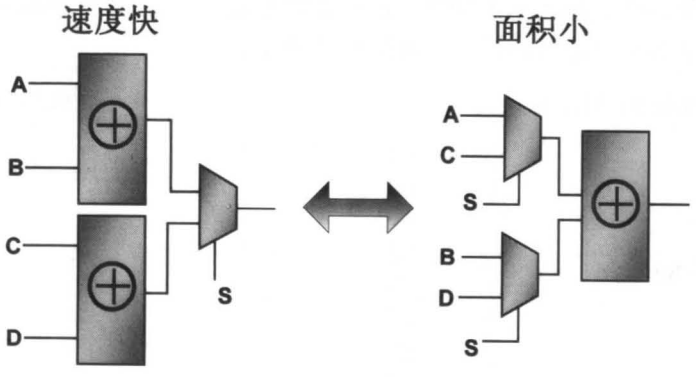

图5-24　资源共享示例

第一种写法比第二种写法省了一个加法器。但是严格来讲，第一种写法比第二种写法耗时略少一些，这在本例中还不算十分明显，当运算模块不是加法器而是一些较为复杂的逻辑时，会比较明显。当设计的时序满足要求，或者设计的面积紧张时，一般会采用资源共享的优化方法，将第一种设计转换为第二种设计。但是在某些特殊情况下，则要反其道而行之，将第二种设计转换为第一种设计，以便于调整组合逻辑信号的到达时间，提高该加法选择器的工作频率。

5.7.4 香农扩展运算

前面已经讲到，香农扩展（Shannon Expansion）也是一种逻辑复制、增加面积、提高频率的时序优化手段，其定义如下：

$$F(a,b,c) = aF(1,b,c) + \bar{a}F(0,b,c)$$

从上面的定义可以看出，香农扩展即布尔逻辑扩展，是卡诺逻辑化简的反向运算。香农扩展相当于逻辑复制，提高频率；而卡诺逻辑化简相当于资源共享，节约面积。

香农扩展通过增加 MUX 来缩短某个优先级高、但是组合路径长的信号的路径延时，从而提高了该关键路径的工作频率。通过下面的例子，读者会对香农扩展有一个更全面的理解。

【例 5-8】 使用香农扩展优化组合逻辑时序，代码参见随书光盘中“Example-5-8\source”目录下的相关内容。

设需要运算的逻辑表达式如下：

F = ((({8{late}} | in0) + in1) = = in2) & en;

其中信号 in0、in1 和 in2 都是 8bit 的数据，信号 late 和信号 en 是控制信号，信号 late 是本逻辑运算的关键路径信号，延时最大。

使用香农扩展：

F = late.F(late=1) + ~late.F(late=0)

= late.[((({8{1'b1}} | in0) + in1) = = in2) & en] +

~late.[((({8{1'b0}} | in0) + in1) = = in2) & en]

= late.[(8'b1+in1) = = in2] & en] + ~late[((in0 + in1) = = in2) & en]

这相当于一个以 late 为选择信号，以[(8'b 1+in1) = = in2] & en]和[((in0 + in1) = = in2) & en]为两个输入信号的二选一的 MUX，因此 late 信号的优先级被提高，其信号路径的延时降低，但是其代价是设计的面积增加了，并且需要两个比较运算符。

未使用香农扩展的 Verilog 代码如下。

```
module un_shannon (in0,in1,in2,late,en,out);
input [7: 0] in0,in1,in2;
input late,en;
output out;
  assign out = ((({8{late}} | in0) + in1) == in2) & en;
endmodule
```

使用 Synplify Pro 综合，关闭优化参数，选择目标器件 Altera Cyclone EP1C3T100C6，得到如图 5-25 所示的 RTL 视图。

从图中可以清晰地看到，未使用香农扩展时，输入 PAD late 到输出 PAD out 之间共有 4 个逻辑单元，5 段路径。其综合结果使用了一个 8bit 或门、一个 8bit 输入加法器、一个 8bit 比较器和一个 2 输入与门。不选择任何优化参数，用 Synplify Pro 在上述目标器件上综合，并用 Quartus II 实现，其实现结果是共使用了 22 个 LE，最差的 tpd 时间约为 13ns。

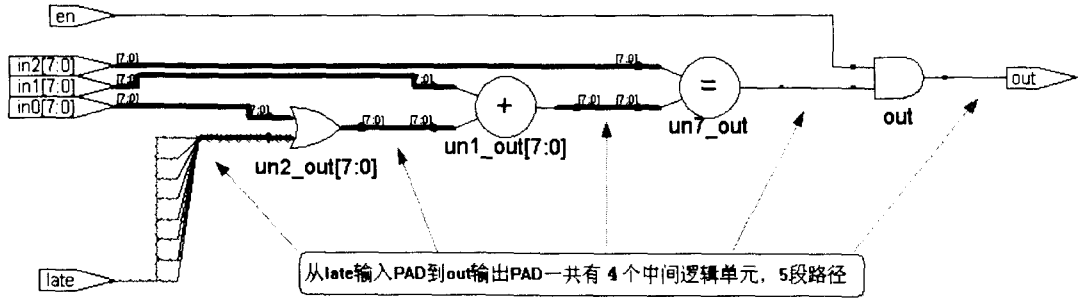

图5-25　未使用香农扩展前的逻辑表达式所对应的 RTL 视图

使用香农扩展的 Verilog 代码如下。

```
module shannon_fast (in0,in1,in2,late,en,out);
input [7: 0] in0,in1,in2;
input late,en;
output out;
wire late_eq_0,late_eq_1;
assign late_eq_0 = ((in0+in1) == in2) & en;
//assign late_eq_0 = ((({8{1'b0}} | in0) + in1) == in2) & en;
assign late_eq_1 = ((8'b1+in1) == in2) & en;
//assign late_eq_1 = ((({8{1'b1}} | in0) + in1) == in2) & en;
assign out = (late) ? late_eq_1 : late_eq_0;
endmodule
```

同样使用 Synplify Pro 综合，选择相同的目标器件 Cyclone EP1C3T100C6，得到如图 5-26 所示的 RTL 视图。

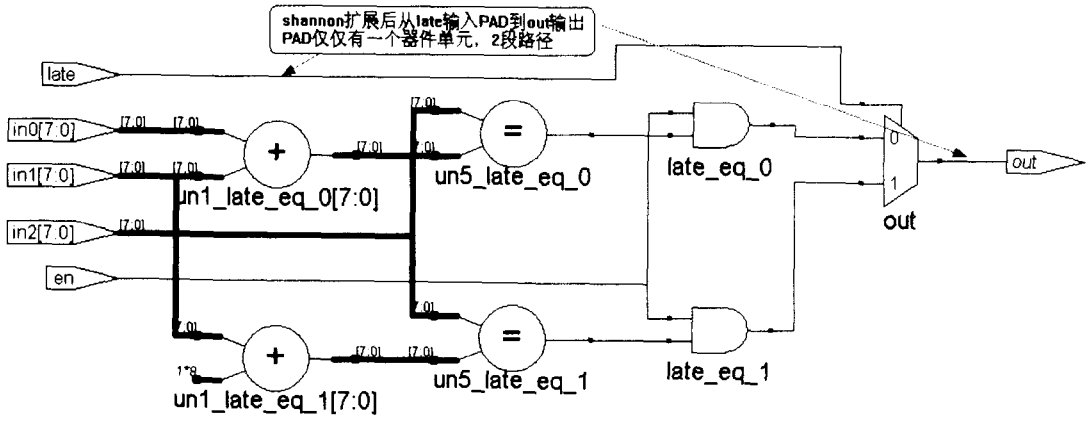

图5-26　使用香农扩展后的逻辑表达式所对应的 RTL 视图

在图中可以清晰地看到，使用香农扩展后，输入 PAD late 到输出 PAD out 之间共有一个逻辑单元，两段路径。其综合结果是使用了两个 8bit 输入加法器、两个 8bit 比较器、两

个输入与门和一个 2 输入选择器。采用默认参数，用 Synplify Pro 在上述目标器件上综合，并用 Quartus II 实现，其结果是共使用了 27 个 LE，最差的 tpd 时间约为 12.7ns。

虽然使用不同的器件族，综合结果的工作频率不一致，但是从 RTL 视图中可以清晰地看到，采用香农扩展后，对于 late 信号这一关键路径，消除了 3 个逻辑层次，从而在一定程度上提高了设计的工作频率。作为提高工作频率的代价，多用了一个加法器和选择器，消耗了更多的面积。由于本例十分简单，所以多消耗的 LUT 和缩短的路径延时都不十分显著，如果在复杂设计中运用香农扩展，就会取得更加显著的效果。

正如前面反复强调的面积和速度的平衡关系，是否使用香农扩展时序优化手段，关键要看被优化对象的优化目标是面积还是速度。

5.8 小结

图 5-27 所示为 RTL 代码的设计目标，包括以下几项内容。

- 时序（Timing）：要求设计满足预期的时序约束条件，满足预期频率要求。
- 面积（Area）：要求设计所消耗的资源满足所规划的面积要求。
- 时钟（Clocks）：要求设计中的时钟质量满足规划要求。
- 验证（Verification）：要求设计通过仿真验证的测试。
- 可测试性（DFT，Design For Test）：要求设计时充分考量设计的可测试性。
- 可重用性（Reuse）：要求设计便于被重用，便于继承设计成果。
- 功耗（Power）：要求设计的最大功耗在规定范围之内。
- 原厂策略（Vendor Policies）：这里主要指 EDA 工具的优化测量和 PLD、ASIC 厂方的设计规格与软硬件需求。

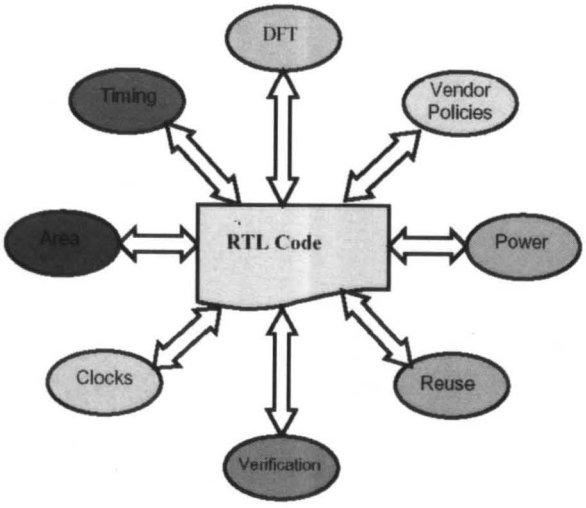

图5-27 RTL 代码的设计目标

对于一个 RTL 设计而言，要求它同时满足上述所有目标是非常困难的，设计者应该根据 RTL 设计的实现载体和具体应用，分析上述要求对本设计的重要性，然后综合考虑以上因素。

希望读者有意识地将本章所述的一般性指导原则和代码风格运用在设计实践中，不断总结并完善这些设计原则，从而迅速提高自己的设计水平。

5.9 问题与思考

1. 简述同步设计和异步设计的特点。
2. 进行模块划分时应该考虑哪些问题？
3. 组合逻辑设计有哪些需要考虑的问题？
4. 说出几种 RTL 代码优化的技巧。

第6章　如何写好状态机

状态机是逻辑设计的重要内容，状态机的设计水平直接反应了工程师的逻辑功底，所以在许多公司的硬件和逻辑工程师面试中，状态机设计几乎是必选题目。本章在引入状态机设计思想的基础上，重点讨论如何写好状态机。

本章主要内容如下：

- 状态机的基本概念；
- 如何写好状态机；
- 使用 Synplify Pro 分析 FSM。

6.1 状态机的基本概念

本节的重点在于帮助读者理解状态机的基本概念和应用场合。

6.1.1 状态机是一种思想方法

相信大多数工科学生在学习数字电路时都学习过状态机这一基本概念，了解一些使用状态机描述时序电路的基本方法。但是，笔者希望读者能扩展思维，认识到状态机不仅仅是一种时序电路设计工具，而且还是一种思想方法。

首先看下面一个简单的例子。在大学生活中，某学生的在校生活可以简单地概括为宿舍、教室、食堂之间的周而复始，用图 6-1 就可以形象地表现出来。这里画这张图，并不是要讨论这个学生是否是一个“乖乖”类型学生，请大家注意，如果将图中的“地点”认为是“状态”，将“功能”认为是状态的“输出”，那么这张图就是一张标准的状态转移图，也就是说，我们用状态机的方式清晰地描述了这个学生的在校生活。

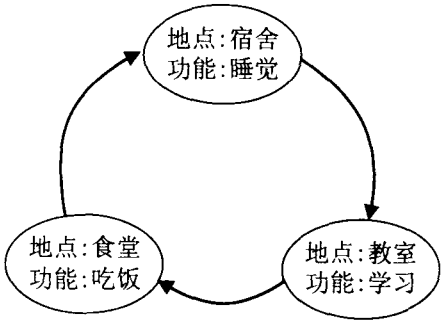

图6-1　某学生在校生活状态转移图

如果读者认为这张图描述的学生生活过于单调而怀疑状态机描述方法的功能，那么再来看看另一位生活丰富多彩的学生的在校生活，他（她）的在校生活方式可以用图 6-2 表示。

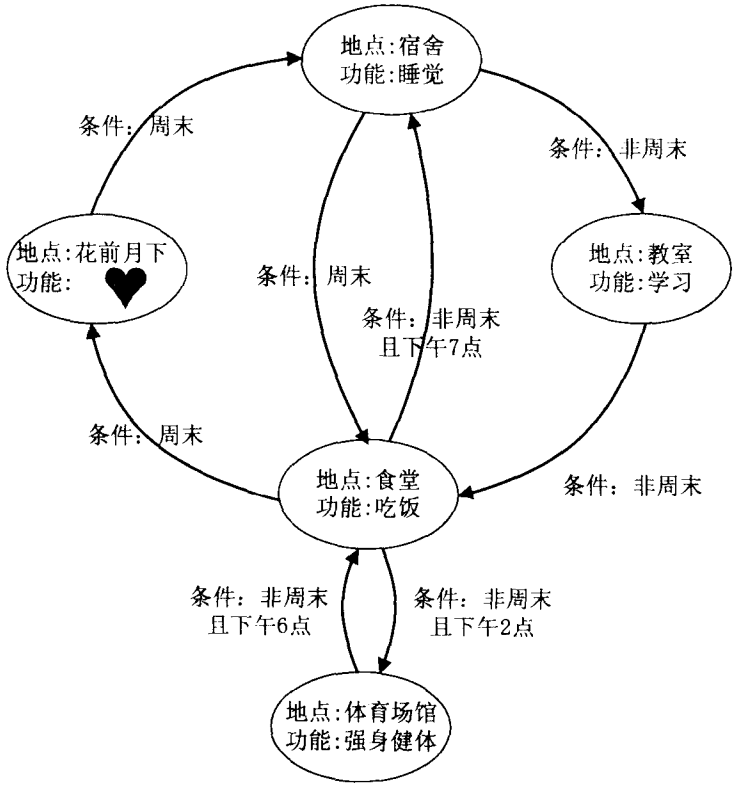

图6-2　另一位学生在校生活状态转移图

同样如果将图中的“地点”认为是“状态”，将“功能”认为是状态的“输出”，将“条件”认为是状态转移的“输入条件”，那么图 6-2 也是一张标准的状态转移图，通过状态机的方式我们再次清晰地描述了另一个学生的在校生活方式。

事实上，使用状态机方式可以细致入微地描述任何一个学生的在校生活方式。通过前面两个简单的例子不难发现，状态机特别适合描述那些发生有先后顺序或者有逻辑规律的事情，其实这就是状态机的本质。状态机就是对具有逻辑顺序或时序规律的事件进行描述的一种方法。这个论断中最重要的两个词就是“逻辑顺序”和“时序规律”，这两点就是状态机所要描述的核心和强项，换言之，所有具有逻辑顺序和时序规律的事情都适合用状态机描述。

很多初学者不知道何时应用状态机，这里介绍两种应用思路，第一种思路是从状态变量入手。如果一个电路具有时序规律或者逻辑顺序，我们就可以自然而然地规划出状态，从这些状态入手，分析每个状态的输入、状态转移和输出，从而完成电路功能；第二种思路是首先明确电路的输出关系，这些输出相当于状态的输出，回溯规划每个状态、状态转移条件及状态输入等。无论哪种思路，其目的都是要控制某部分电路，完成某种具有逻辑顺序或时序规律的电路设计。

其实对于逻辑电路而言，小到一个简单的时序逻辑，大到复杂的微处理器，都适合用状

态机进行描述。

6.1.2 状态机的基本要素及分类

状态机的基本要素有 3 个，分别是状态、输出和输入。

- 状态：也叫状态变量。在逻辑设计中，使用状态划分逻辑顺序和时序规律。比如设计伪随机码发生器时，可以用移位寄存器序列作为状态；在设计电机控制电路时，可以将电机的不同转速作为状态；在设计通信系统时，可以将信令的状态作为状态变量等。
- 输出：输出指在某一个状态时特定发生的事件。如设计电机控制电路时，如果电机转速过高，则输出为转速过高报警，也可以伴随减速指令或降温措施等。
- 输入：指状态机中进入每个状态的条件，有的状态机没有输入条件，其中的状态转移较为简单，有的状态机有输入条件，当某个输入条件存在时才能转移到相应的状态。

根据状态机的输出是否与输入条件相关，可将状态机分为两大类，即摩尔（Moore）型状态机和米勒（Mealy）型状态机。

- 摩尔型状态机：摩尔型状态机的输出仅仅依赖于当前状态，而与输入条件无关。例如图 6-1 所示的例子，将图中的“地点”认为是“状态”，将“功能”认为是状态的“输出”，则每个输出仅与状态相关，所以它是一个摩尔型状态机。
- 米勒型状态机：米勒型状态机的输出不仅依赖于当前状态，而且取决于该状态的输入条件。例如图 6-2 所示的例子，将图中的“地点”认为是“状态”，将“功能”认为是状态的“输出”，将“条件”认为是状态转移的“输入条件”，可以发现，该学生到达什么地方，做什么事情都是由当前状态和输入条件共同决定的，所以它是一个米勒型状态机。

根据状态机的数量是否为有限个，可将状态机分为有限状态机（Finite State Machine，FSM）和无限状态机（Infinite State Machine，ISM）。逻辑设计中所涉及到的状态一般都是有限的，所以以后我们所说的状态机都指有限状态机，用 FSM 表示。

6.1.3 状态机的基本描述方式

在逻辑设计中，状态机的基本描述方式有 3 种，分别是状态转移图、状态转移列表和 HDL 语言描述。

- **状态转移图**

  状态转移图是状态机描述中最自然的方式，如本章第一节中的图 6-1 和图 6-2 都使用了状态转移图这一描述方式。状态转移图经常在设计规划阶段定义逻辑功能时使用，也可以在分析已有源代码中的状态机时使用，这种图形化的描述方式非常有助于理解设计意图。

另外值得一提的是，目前有一些 EDA 工具支持状态转移图作为逻辑设计的输入，例如 StateCAD，在该工具中设计者只要画出状态转移图就可以了，StateCAD 能自动将状态转移图翻译成 HDL 语言代码，而且翻译出来的代码规范，可读性较好，可综合，易维护。StateCAD 还能自动检测状态机的完备性和正确性，当状态转移图中存在冗余状态、自锁状态、歧义转移条件和不完备状态机等隐含的错误时都会报警，并协助设计者更正错误。StateCAD 还会自动生成设计的测试激励，并调用仿真程序，验证状态机的正确性，这个测试激励甚至可在后仿真中使用。总之，StateCAD 提供了状态机的输入、翻译、检测、优化和测试等一条龙的服务，使状态机的设计变得安全、可靠、快速、便捷。这类能将状态转移图自动转换为 HDL 源代码的辅助设计工具对设计、分析一些规模较小的状态机非常有效，但是由于其自动翻译的代码过于程式化，因此效率不是很高，对于较大规模的逻辑设计而言，还是推荐使用 HDL 语言进行描述。

使用 Synplify Pro 的 RTL 视图配合 FSM Viewer 可以将源代码中描述的 FSM 用状态转移图显示出来，使用图形化的界面帮助用户分析理解状态机。关于使用 FSM Viewer 分析状态机的方法将在本章 6.3 节中详细介绍。

- **状态转移列表**

  状态转移列表是用列表的方式描述状态机，是数字逻辑电路常用的设计方法之一，经常被用于状态化简，对于可编程逻辑设计而言，由于可用逻辑资源比较丰富，而且状态编码要考虑设计的稳定性、安全性等因素，所以并不经常使用状态转移列表优化状态。

- **HDL 语言描述状态机**

  如何使用 HDL 语言描述状态机是本章所要讨论的重点，使用 HDL 语言描述状态机应具有一定的灵活性，但是决不是天马行空，而是有章可循的。通过使用一些规范的描述方法，可以使 HDL 语言描述状态机更安全、更稳定、更高效、更易于维护。

6.2 如何写好状态机

本节重点讨论如何在 RTL 级描述安全、高效的 FSM。

6.2.1 评判 FSM 的标准

评判 FSM 的标准很多，这里拣选几个最重要的方面进行讨论。好的 RTL 级 FSM 的评判标准如下。

- **FSM 要安全，稳定性高**

  所谓 FSM 安全是指 FSM 不会进入死循环，特别是不会进入非预知的状态，而且即使由于某些扰动进入非设计状态，也能很快恢复到正常的状态循环中来。这里面有两层含义，第一，要求该 FSM 的综合实现结果无毛刺等异常扰动；

第二，要求状态机要完备，即使受到异常扰动进入非设计状态，也能很快恢复到正常状态。

- **FSM 速度快，满足设计的频率要求**
  任何 RTL 设计都应该满足设计的频率要求。
- **FSM 面积小，满足设计的面积要求**
  任何 RTL 设计都应该满足设计的面积要求。
- **FSM 设计要清晰易懂、易维护**
  不规范的 FSM 写法很难让其他人解读，过一段时间后设计者会发现其很难维护。

需要说明的是以上所列的各项标准，特别是前 3 项标准绝不是割裂的，它们之间有着紧密的内在联系。如果读者读过本工作室的其他书籍，应该记得其中花了相当长的篇幅论述了评判 FPGA/CPLD 设计的两个基本标准，即面积和速度。这里的“面积”是指一个设计所消耗 FPGA/CPLD 的逻辑资源数量；“速度”指设计在芯片上稳定运行所能够达到的最高频率。两者是对立统一的矛盾体，要求一个设计同时具备设计面积最小、运行频率最高的特性，这是不现实的。科学的设计目标应该是在满足设计时序要求（包含对设计最高频率的要求）的前提下，占用最小的芯片面积，或者在所规定的面积下，使设计的时序余量更大，频率更高。

另外，如果要求 FSM 安全，那么在很多时候都需要使用“full case”的编码方式，即状态转移变量的所有向量组合情况都在 FSM 中有相应的处理，这势必意味着要耗费更多的设计资源，有时也会影响 FSM 的频率。

所以各条标准要综合考虑，根据设计的要求来权衡。当各条评判标准发生冲突时，请按照标准的罗列顺序考虑，所谓罗列顺序就是指根据这些标准在设计中的重要性进行排列，也就是说第一条“FSM 要安全，稳定性高”的优先级最高，最重要，第四条“FSM 设计要清晰易懂、易维护”的优先级最低，是相对次要的标准。

6.2.2 RTL 级状态机描述常用的语法

本书第 2、3 章中介绍了 Verilog 的基本语法和常用关键字，其中在 RTL 级设计可综合的 FSM 时常用的关键字如下：

- **wire 、reg 等**
  这里不对 **wire** 、**reg** 等变量、向量的定义做过多的阐述，需要补充的是状态编码时（也就是用某种编码描述各个状态时）一般都要使用 **reg** 寄存器型向量。
- **parameter**
  用于描述状态名称，增强源代码的可读性，简化描述。

例如某状态机使用初始值为“0”的独热码（one-hot）编码方式定义的 4bit 宽度的状态变量 NS（代表 Next State，下一状态）和 CS（代表 Current State，当前状态），且状态机包含 5 个具体状态 IDLE（空闲状态）、S1（工作状态 1）、S2（工作状态 2）、S3（工作状态

3）和ERROR（告警状态），则代码如下：

```
reg [3:0] NS,CS;
parameter [3:0]  //one hot with zero initial
IDLE  = 3'b0000,
S1    = 3'b0001,
S2    = 3'b0010,
S3    = 3'b0100,
ERROR = 3'b1000;
```

- **always**

在FSM设计中有3种**always**的使用方法，第一种用法是根据主时钟沿完成同步时序的状态迁移。

例如某状态机从当前状态CS迁移到下一个状态NS可以如下表述：

```
//sequential state transition
always @ (posedge clk or negedge nrst)
    if (!nrst)
      CS <= IDLE;
    else
      CS <=NS;
```

always的第二种用法是根据信号敏感表完成组合逻辑的输出。

always的第三种用法是根据时钟沿完成同步时序逻辑的输出。

- **case/endcase**

case/endcase是FSM描述中最重要的语法关键字，其语法结构如下：

```
case (case_expression)
case_item1 : case_item_statement1;
case_item2 : case_item_statement2;
case_item3 : case_item_statement3;
case_item4 : case_item_statement4;
default : case_item_statement5;
endcase
```

其中，case_expression是case的判断条件表达式，在FSM描述中一般为当前状态寄存器；每个case_item是case语句的分支列表，在FSM描述中一般为FSM中所有状态的罗列，从中还可以分析出状态的编码方式；case_item_statement为进入每个case_item的对应操作，在FSM描述中即为每个状态对应的状态转移或者输出，如果case_item_statement包含的操作不只一条，可以用**begin/end**嵌套多条操作；**default**是个可选的关键字，用以指明当所列的所有case_item与case_expression都不匹配时的操作，在FSM设计中，为了提高设计的安全性，避免所设计的FSM进入死循环，一般要求加上**default**关键字来描述FSM所需状态的补集状态下的操作。另外Verilog还支

持 **casex** 和 **casez** 等不同关键字，但是由于综合器对这两个关键字的支持情况略有差异，因此笔者建议初学者使用完整的 **case** 结构，而不使用 **casex** 或 **casez**。

例如某 FSM 的状态转移用 case/endcase 结构描述如下：

```
case (CS)
    IDLE:    begin
                 IDLE_out;
                 if (~i1)          NS = IDLE;
                 if (i1 && i2)      NS = S1;
                 if (i1 && ~i2)     NS = ERROR;
             end
    S1:      begin
                 S1_out;
                 if (~i2)          NS = S1;
                 if (i2 && i1)      NS = S2;
                 if (i2 && (~i1))   NS = ERROR;
             end
    S2:      begin
                 S2_out;
                 if (i2)           NS = S2;
     if (~i2 && i1)     NS = IDLE;
                 if (~i2 && (~i1))  NS = ERROR;
             end
    ERROR:   begin
                 ERROR_out;
                 if (i1)           NS = ERROR;
                 if (~i1)          NS = IDLE;
             end
    default: begin
                 Default_out;
                 NS = ERROR;
             end
endcase
```

Verilog 的 **case** 结构虽然与 C 等高级语言的 **case** 结构形式相似，但是本质不同。Verilog 的 **case** 结构对应并行判断的硬件结构，而且当 case_expression 与任意一个 case_item 匹配后，将忽略对其他 case_item 的判断，执行完匹配的 case_item_statement 后直接跳出 **case** 结构。

- **task/endtask**

task/endtask 在描述状态机时的主要用途是将不同状态所对应的输出用

task/endtask 封装，增强了代码的可维护性和可读性。

例如某状态机 IDLE 状态的输出可以用 task/endtask 封装为“IDEL_out”任务：

```
task IDLE_out;
    begin
    {w_o1,w_o2,w_err} = 3'b000;
    end
```

当然，在描述状态机时也会使用到其他一些常用的 RTL 级语法，如 if/else 和 assign 等，它们的功能与一般的 RTL 描述方法一致，这里就不再赘述了。

6.2.3 推荐的状态机描述方法

描述状态机时关键是要描述清楚前面提到的几个状态机要素，即如何进行状态转移，每个状态的输出是什么，状态转移是否与输入条件相关等。描述这些要素的方法多种多样，有的设计者习惯将整个状态机写到一个 **always** 模块里，在该模块中既描述状态转移，又描述状态的输入和输出，这种写法一般被称为一段式 FSM 描述方法；还有一种写法是使用两个 **always** 模块，其中一个 **always** 模块采用同步时序的方式描述状态转移，而另一个模块采用组合逻辑的方式判断状态转移条件，描述状态转移规律，这种写法被称为两段式 FSM 描述方法；还有一种写法是在两段式描述方法的基础上发展而来的，这种写法使用 3 个 **always** 模块，一个 **always** 模块采用同步时序的方式描述状态转移，一个采用组合逻辑的方式判断状态转移条件，描述状态转移规律，第三个 **always** 模块使用同步时序电路描述每个状态的输出，这种写法本书称为三段式写法。

一般而言，推荐使用后两种 FSM 描述方法，即两段式和三段式 FSM 描述方法。其原因是 FSM 和其他设计一样，最好使用同步时序方式设计，以提高设计的稳定性，消除毛刺。两段式之所以比一段式编码合理，就在于两段式编码将同步时序和组合逻辑分别放到不同的 **always** 程序块中实现。这样做的好处不仅仅是便于阅读、理解、维护，更重要的是利于综合器优化代码，利于用户添加合适的时序约束条件，利于布局布线器实现设计。而一段式 FSM 描述不利于时序约束、功能更改及调试等，而且不能很好地表示米勒 FSM 的输出，容易写出 Latches，导致逻辑功能错误。

在两段式描述中，为了便于描述当前状态的输出，很多设计者习惯将当前状态的输出用组合逻辑实现。但是这种组合逻辑仍然有产生毛刺的可能性，而且不利于约束，不利于综合器和布局布线器实现高性能的设计。因此如果设计允许插入一个额外的时钟节拍（latency），则要求尽量对状态机的输出用寄存器寄存一拍。但是很多情况不允许插入一个寄存节拍，此时则可以采用三段式描述方法。三段式描述方法与两段式相比，其优势在于能够根据状态转移规律，在上一状态根据输入条件判断出当前状态的输出，从而在不插入额外时钟节拍的前提下实现寄存器输出。

为了便于理解，下面通过一个实例说明这 3 种不同的写法。

【例 6-1】 使用不同的 FSM 描述方法描述状态机，参考示例详见随书光盘中“Example-6-1”目录下的相关内容。

在这个范例中我们将用一段式、两段式、三段式分别描述图 6-3 中所示的状态机。这里选用了一个非常典型的米勒型状态机，共有 4 种状态，即 IDEL、S1、S2 和 ERROR，输入信号包括时钟“clk”，低电平异步复位信号“nrst”和信号“i1”、“i2”，输出信号为“o1”、“o2”和“err”，状态关系如图 6-3 所示。状态的输出如下：

IDLE 状态的输出为{o1,o2,err} = 3'b000;

S1 状态的输出为{o1,o2,err} = 3'b100;

S2 状态的输出为{o1,o2,err} = 3'b010;

ERROR 状态的输出为{o1,o2,err} = 3'b111。

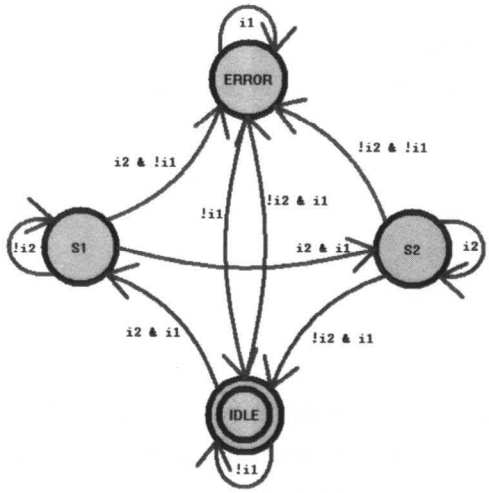

图6-3　实例的状态转移图

一、一段式状态机描述方法（应该避免的写法）

该例的一段式描述代码如下：

```
//1-paragraph method to describe FSM
//Describe state transition, state output, input condition in 1 always block
module state1 ( nrst,clk,
                i1,i2,
                o1,o2,
                err
               );

input          nrst,clk;
input          i1,i2;
output         o1,o2,err;
reg           o1,o2,err;
reg    [2:0]   NS; //NextState
parameter [2:0]      //one hot with zero idle
```

```
    IDLE  = 3'b000,
    S1    = 3'b001,
    S2    = 3'b010,
    ERROR = 3'b100;

//1 always block to describe state transition, state output, input condition
always @ (posedge clk or negedge nrst)
 if (!nrst)
   begin
     NS          <= IDLE;
    {o1,o2,err} <= 3'b000;
   end
 else
   begin
     NS          <=  3'bx;
    {o1,o2,err} <=  3'b000;
    case (NS)
     IDLE:  begin
             if (~i1)         begin{o1,o2,err}<=3'b000;NS  <= IDLE; end
             if (i1 && i2)    begin{o1,o2,err}<=3'b100;NS <= S1;   end
             if (i1 && ~i2)   begin{o1,o2,err}<=3'b111;NS <= ERROR;end
           end
     S1:    begin
             if (~i2)         begin{o1,o2,err}<=3'b100;NS <= S1;   end
             if (i2 && i1)    begin{o1,o2,err}<=3'b010;NS <= S2;   end
             if (i2 && (~i1)) begin{o1,o2,err}<=3'b111;NS <= ERROR;end
           end
     S2:    begin
             if (i2)          begin{o1,o2,err}<=3'b010;NS <= S2;   end
             if (~i2 && i1)   begin{o1,o2,err}<=3'b000;NS <= IDLE; end
             if (~i2 && (~i1))begin{o1,o2,err}<=3'b111;NS <= ERROR;end
           end
     ERROR: begin
             if (i1)          begin{o1,o2,err}<=3'b111;NS <= ERROR;end
             if (~i1)         begin{o1,o2,err}<=3'b000;NS <= IDLE; end
           end
    endcase
  end
endmodule
```

如前面所说，一段式写法就是将状态的同步转移、输出和输入条件都写在一个 **always** 模块中，一段式写法可以概括为如图 6-4 所示的结构。

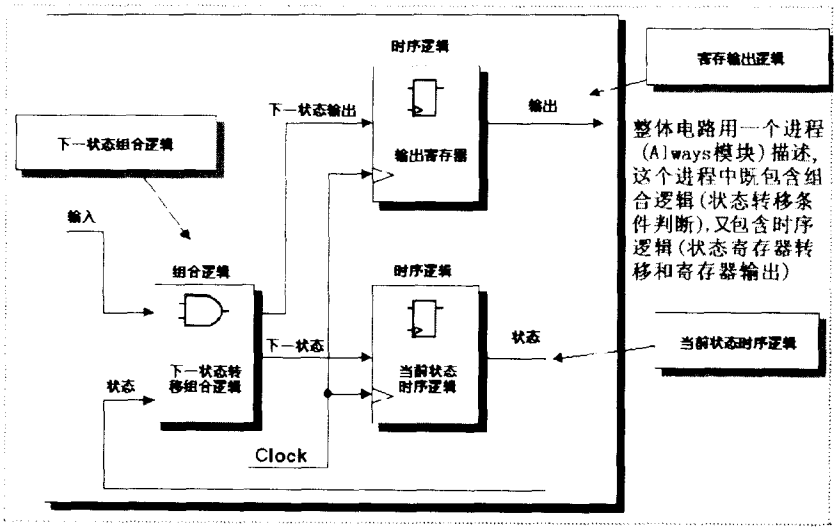

图6-4　一段式 FSM 描述结构图

一段式描述方法将状态转移判断的组合逻辑和状态寄存器转移的时序逻辑混写在同一个 **always** 模块中，不符合将时序和组合逻辑分开描述的 Coding Style（代码风格），而且在描述当前状态时还要考虑下一个状态的输出，整个代码不清晰，不利于维护和修改，不利于附加约束，不利于综合器和布局布线器对设计的优化。

另外，这种描述相对于两段式描述来讲比较冗长。本例为了便于初学者掌握，选择了一个非常简单的米勒型状态机，不能很好地反应一段式描述冗长的缺点，如果状态机相对复杂一些的话，一段式代码的长度会比两段式代码的长度长大约 80%～150%左右。所以不推荐使用一段式描述方式。

二、两段式状态机描述方法（推荐写法）

为了使 FSM 描述清晰简洁，易于维护，易于附加时序约束，便于综合器和布局布线器更好地优化设计，推荐使用两段式 FSM 描述方法。

本例的两段式描述代码如下：

```
//2-paragraph method to describe FSM
//Describe sequential state transition in 1 sequential always block
//State transition conditions in the other combinational always block
//Package state output by task. Then register the output
module state2 ( nrst,clk,
                i1,i2,
                o1,o2,
                err
              );
input           nrst,clk;
```

```
input         i1,i2;
output        o1,o2,err;
reg           o1,o2,err;
reg    [2:0]  NS,CS;
parameter [2:0]      //one hot with zero idle
     IDLE   = 3'b000,
     S1     = 3'b001,
     S2     = 3'b010,
     ERROR  = 3'b100;
//sequential state transition
always @ (posedge clk or negedge nrst)
     if (!nrst)
       CS <= IDLE;
     else
       CS <=NS;
//combinational condition judgment
always @ (CS or i1 or i2)
        begin
            NS = 3'bx;
            ERROR_out;
            case (CS)
                IDLE:    begin
                             IDLE_out;
                             if (~i1)          NS = IDLE;
                             if (i1 && i2)      NS = S1;
                             if (i1 && ~i2)     NS = ERROR;
                         end
                S1:      begin
                             S1_out;
                             if (~i2)          NS = S1;
                             if (i2 && i1)      NS = S2;
                             if (i2 && (~i1))   NS = ERROR;
                         end
                S2:      begin
                             S2_out;
                             if (i2)           NS = S2;
                 if (~i2 && i1)     NS = IDLE;
                             if (~i2 && (~i1))  NS = ERROR;
                         end
```

```
                ERROR:    begin
                              ERROR_out;
                              if (i1)          NS = ERROR;
                              if (~i1)         NS = IDLE;
                          end
            endcase
        end
//output task
task IDLE_out;
    {o1,o2,err} = 3'b000;
endtask
task S1_out;
    {o1,o2,err} = 3'b100;
endtask
task S2_out;
    {o1,o2,err} = 3'b010;
endtask
task ERROR_out;
    {o1,o2,err} = 3'b111;
endtask
endmodule
```

两段式写法是推荐的 FSM 描述方法之一，在此仔细讨论一下代码的结构。两段式描述的核心思想是，一个 **always** 模块采用同步时序方式描述状态转移；另一个模块采用组合逻辑方式判断状态转移条件，描述状态转移规律。两段式写法可以概括为如图 6-5 所示的结构。

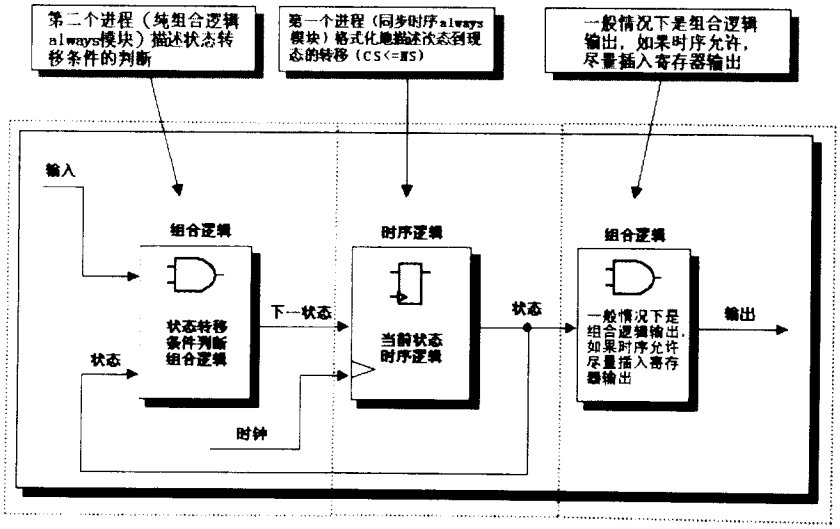

图6-5　两段式 FSM 描述结构图

本例中，同步时序描述状态转移的 **always** 模块代码如下：

```
always @ (posedge clk or negedge nrst)
    if (!nrst)
      CS <= IDLE;
    else
      CS <=NS;
```

其实这是一种程式化的描述结构，无论具体到何种 FSM 设计，都可以定义两个状态寄存器“CS”和“NS”，它们分别代表当前状态和下一状态，然后根据所需的复位方式（同步复位或异步复位），在时钟沿到达时将 NS 赋给 CS。需要注意的是，这个同步时序模块的赋值要采用非阻塞赋值“<=”。

本例中，另一个采用组合逻辑判断状态转移条件的 **always** 模块代码如下：

```
//combinational condition judgment
always @ (nrst or CS or i1 or i2)
        begin
            NS = 3'bx;
            ERROR_out;
            case (CS)
                IDLE:    begin
                             IDLE_out;
                             if (~i1)          NS = IDLE;
                             if (i1 && i2)     NS = S1;
                             if (i1 && ~i2)    NS = ERROR;
                         end
                S1:      begin
                             S1_out;
                             if (~i2)          NS = S1;
                             if (i2 && i1)     NS = S2;
                             if (i2 && (~i1))  NS = ERROR;
                         end
                S2:      begin
                             S2_out;
                             if (i2)           NS = S2;
                 if (~i2 && i1)    NS = IDLE;
                             if (~i2 && (~i1)) NS = ERROR;
                         end
                ERROR:   begin
                             ERROR_out;
                             if (i1)           NS = ERROR;
```

```
                    if (~i1)          NS = IDLE;
                end
            endcase
        end
```

这个使用组合逻辑判断状态转移条件的 **always** 模块也可以采用格式化的结构书写。其中 **always** 的敏感列表为当前状态“CS”、复位信号和输入条件（如果是米勒状态机，则必须有输入条件；如果是摩尔状态机，则敏感表和后续逻辑判定没有输入），请读者注意，电平敏感表必须列完整。本例中的这段电平敏感列表为：

```
always @ (nrst or CS or i1 or i2)
```

一般来说，要先在这个组合 **always** 敏感表下写出默认的下一状态“NS”的描述，然后根据实际的状态转移条件由内部的 **case** 或者 **if...else** 条件判断确定正确的转移，如本例中下面这段代码：

```
...
begin
    NS = ERROR;
    ERROR_out;
    case (CS)
...
```

推荐将敏感表后描述的默认状态设计为不定状态 X，这样描述的好处有两个，第一个好处是在仿真时可以很好地考察所设计的 FSM 的完备性，如果所设计的 FSM 不完备，则会进入任意状态，仿真时很容易发现；第二个好处是综合器对不定态 X 的处理是“Don't Care”，即任何没有定义的状态寄存器向量都会被忽略。这里赋值不定态的效果和使用 **casez** 或 **casex** 替代 **case** 的效果非常相似。

每个 case 模块的内部结构也非常相似，都是先描述当前状态的组合逻辑输出，然后根据输入条件（米勒 FSM）判定下一个状态。

该组合逻辑模块中所有的赋值推荐采用阻塞赋值“=”。

请读者注意，虽然下一状态寄存器 NS 为寄存器类型，但是在两段式 FSM 判断状态转移条件的 **always** 模块中，实际上对应的真实硬件电路是纯组合逻辑电路。

每个输出一般都用组合逻辑描述，比较简便的方法就是用 **task/endtask** 将输出封装起来，这样做的好处不仅仅是写法简单，而且利于复用共同的输出。例如本例中 S1 状态的输出被封装为 S1_out，在组合逻辑 **always** 模块中直接调用即可。

```
task S1_out;
    {o1,o2,err} = 3'b100;
endtask
```

组合逻辑容易产生毛刺，因此如果时序允许，请尽量对组合逻辑的输出插入一个寄存器节拍，这样可以更好地保证输出信号的稳定性。

三、三段式状态机描述方法（推荐写法）

两段式 FSM 描述方法虽然有很多好处，但是它有一个明显的弱点，就是其输出一般使用组合逻辑描述，而组合逻辑易产生毛刺等不稳定因素，并且在 FPGA/CPLD 等逻辑器件中过多的组合逻辑会影响实现的速率（这点与 ASIC 设计不同），所以在上节中特别提到了在两段式 FSM 描述方法中，如果时序允许插入一个额外的时钟节拍，则尽量在后级电路对 FSM 的组合逻辑输出用寄存器寄存一个节拍，这样可以有效地消除毛刺。但是在很多情况下，设计并不允许插入额外的节拍（Latency），此时就需要采用三段式 FSM 描述方法。三段式描述方法与两段式描述方法相比，关键在于使用同步时序逻辑寄存 FSM 的输出。

本例的三段式描述代码如下：

```
//3-paragraph method to describe FSM
//Describe sequential state transition in the 1st sequential always block
//State transition conditions in the 2nd combinational always block
//Describe the FSM out in the 3rd sequential always block
    module state2 ( nrst,clk,
                    i1,i2,
                    o1,o2,
                    err
                   );
    input          nrst,clk;
    input          i1,i2;
    output         o1,o2,err;
    reg           o1,o2,err;
    reg    [2:0]   NS,CS;
    parameter [2:0]      //one hot with zero idle
         IDLE   = 3'b000,
         S1     = 3'b001,
         S2     = 3'b010,
         ERROR  = 3'b100;
    //1st always block, sequential state transition
    always @ (posedge clk or negedge nrst)
         if (!nrst)
           CS <= IDLE;
         else
           CS <=NS;
    //2nd always block, combinational condition judgment
    always @ (nrst or CS or i1 or i2)
            begin
                NS = 3'bx;
```

```
            case (CS)
                IDLE:    begin
                             if (~i1)          NS = IDLE;
                             if (i1 && i2)     NS = S1;
                             if (i1 && ~i2)    NS = ERROR;
                         end
                S1:      begin
                             if (~i2)          NS = S1;
                             if (i2 && i1)     NS = S2;
                             if (i2 && (~i1))  NS = ERROR;
                         end
                S2:      begin
                             if (i2)           NS = S2;
                             if (~i2 && i1)    NS = IDLE;
                             if (~i2 && (~i1)) NS = ERROR;
                         end
                ERROR:   begin
                             if (i1)           NS = ERROR;
                             if (~i1)          NS = IDLE;
                         end
            endcase
        end
    //3rd always block, the sequential FSM output
    always @ (posedge clk or negedge nrst)
     if (!nrst)
         {o1,o2,err} <= 3'b000;
     else
       begin
          {o1,o2,err} <=  3'b000;
          case (NS)
             IDLE:  {o1,o2,err}<=3'b000;
             S1:    {o1,o2,err}<=3'b100;
             S2:    {o1,o2,err}<=3'b010;
             ERROR: {o1,o2,err}<=3'b111;
          endcase
       end
    endmodule
```

三段式写法可以概括为如图 6-6 所示的结构。

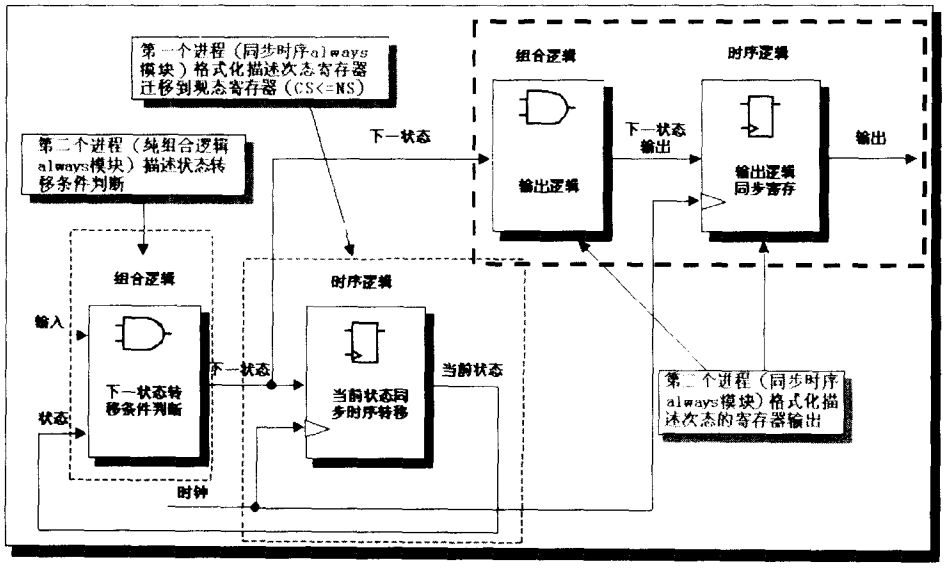

图6-6　三段式 FSM 描述结构图

对比上节两段式 FSM 的描述，可以清晰地发现三段式 FSM 描述与两段式 FSM 描述的最大区别在于两段式采用了组合逻辑输出，而三段式则巧妙地根据对下一状态的判断，用同步时序逻辑寄存 FSM 的输出，例如本例中的下面一段代码：

```
always @ (posedge clk or negedge nrst)
 if (!nrst)
     {o1,o2,err} <= 3'b000;
 else
    begin
      {o1,o2,err} <=  3'b000;
      case (NS)
         IDLE:  {o1,o2,err}<=3'b000;
         S1:    {o1,o2,err}<=3'b100;
         S2:    {o1,o2,err}<=3'b010;
         ERROR: {o1,o2,err}<=3'b111;
      endcase
    end
```

有的读者可能会问，一段式写法也是用寄存器同步了 FSM 的输出，为什么说一段式的输出代码容易混淆，不利于维护呢？请读者对比一下 6.2.3 节中这段一段式输出的代码：

```
          ...
      case (NS)
        IDLE:  begin
               if (~i1)        begin{o1,o2,err}<=3'b000;NS  <= IDLE; end
               if (i1 && i2)    begin{o1,o2,err}<=3'b100;NS <= S1;   end
               if (i1 && ~i2)   begin{o1,o2,err}<=3'b111;NS <= ERROR;end
```

```
        end
    ...
```

通过对比可以清晰地看到，使用一段式建模 FSM 的寄存器输出时，必须要综合考虑现态在何种状态转移条件下会进入哪些次态，然后在每个现态的 case 分支下分别描述每个次态的输出，这显然不符合思维习惯；而三段式建模描述 FSM 的状态机输出时，只需指定 **case** 敏感表为次态寄存器，然后直接在每个次态的 case 分支中描述该状态的输出即可，根本不用考虑状态转移条件。本例的 FSM 很简单，如果设计的 FSM 复杂一些的话，三段式描述的优势就会凸显出来。

另一方面，三段式描述方法与两段式描述相比，虽然代码结构复杂了一些，但是换来的却是使 FSM 做到了同步寄存器输出，消除了组合逻辑输出的不稳定性，而且更利于时序路径分组，一般来说，其在 FPGA/CPLD 等可编程逻辑器件上的综合与布局布线效果更佳。

请读者注意，在三段式 FSM 描述方法中，判断状态转移的 always 模块的 case 语句判断的条件是当前状态 "CS"，而在同步时序 FSM 输出的 always 模块的 case 语句判断的条件是下一状态 "NS"。

四、3 种描述方法与状态机建模问题的引申

可以说合理的状态机描述与状态机的建模技巧是本章的重中之重。这里需要引申讨论几个问题。

- ***n* 段式描述方法和 always 语法块的个数**

  通过学习可知标准的一段式、两段式、三段式 FSM 描述方法分别使用了 1、2、3 个 always 语法块。但是请读者注意，这个命题的反命题不成立，不能说一段 FSM 的描述中使用了 *n* 个 always 语法块，就是 *n* 段式描述方法。这是因为我们特指的一段式、两段式、三段式 FSM 描述方法中每个 always 语法块都有固定的描述内容和格式化的结构，其实也就是通过这些特定的描述内容和格式化的结构，确立了 3 种 FSM 建模方式。例如两段式写法中，第一个 always 模块格式化地使用同步时序电路描述次态寄存器到现态寄存器的转移，而第二个 always 模块格式化地使用纯组合逻辑描述状态转移条件。也就是说本书所指的两段式建模方法所对应的硬件电路结构就是图 6-5 所示的电路结构。其实从语法角度上说，总可以将一个 always 模块拆分成多个 always 模块，或者反之将多个 always 模块合并为一个 always 模块。所以请读者注意，*n* 段式 FSM 描述方法强调的是一种建模思路，绝不是简单的 always 语法块个数。

- **FSM 的建模方式**

  这里反复强调的 *n* 段式描述方法其实就是 FSM 的 3 种建模方式。回顾一下图 6-3 中描述的 FSM，在学习本节之前，读者可能会有各种不同的描述思路，通过本节的学习，希望读者能够自然而然地想到用图 6-5（对应两段式思路）和图 6-6（对应三段式思路）所示的结构建模。其实对于绝大多数 FSM 来说，都可以采用图 6-4、图 6-5 或图 6-6 所示的结构建模。一般来说，笔者推荐使用后两种结构建模，这是因为采用两段式思路建模结构清晰，描述简洁，便于约束，而且如果输出逻辑允许插入一个时钟节拍，就可以通过插入输出寄

存器来改善输出逻辑的时序，并在一定程度上避免了组合逻辑产生毛刺，而采用三段式思路建模结构清晰，格式化的结构解决了在不改变时序要求的前提下用寄存器做状态输出的问题。

- **一段式建模和三段式建模的关系**

请读者比较一下图 6-4 和图 6-6，如果将图 6-4 中所示的两部分组合逻辑合并起来，则三段式建模电路与一段式建模电路的结构就完全一致了，如图 6-7 所示。一段式建模与三段式建模的区别前面已经介绍过了，因此这里不再赘述。

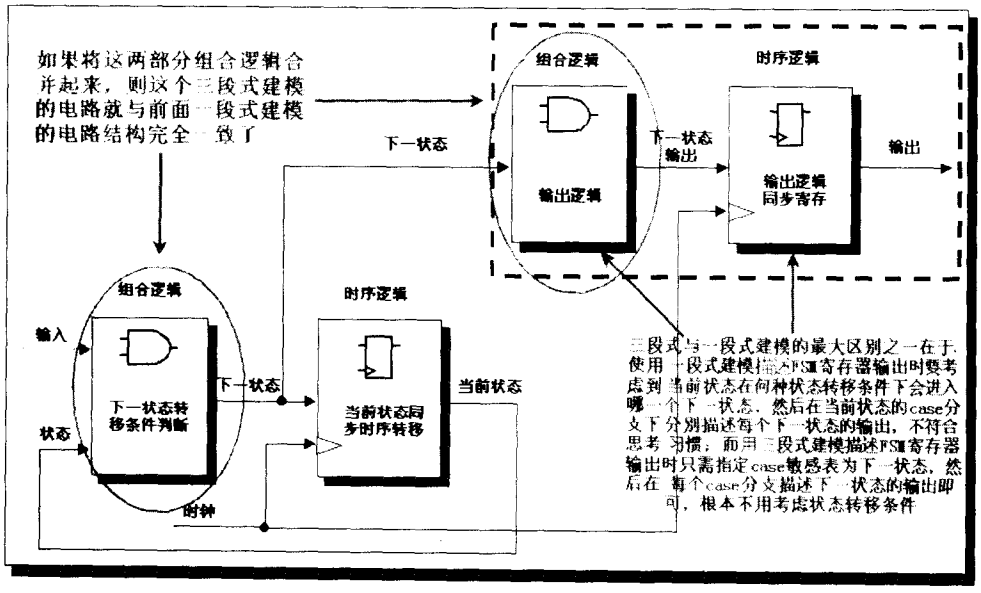

图6-7　三段式建模结构与一段式建模结构的关系图

- **两段式建模和三段式建模的关系**

从代码上看，三段式建模的前两段与两段式建模完全相同，仅仅多了一段寄存器 FSM 输出。一般来说，使用寄存器输出不仅可以改善输出的时序条件，而且还能避免出现组合电路的毛刺，但是电路设计不是一成不变的，在某些情况下，两段式结构比三段式结构更有优势。分析一下图 6-5 和图 6-6 中所示的结构，细心的读者就会发现，两段式用状态寄存器分割了两部分组合逻辑（状态转移条件组合逻辑和输出组合逻辑），而三段式结构中，从输入到寄存器状态输出的路径上要经过两部分组合逻辑（状态转移条件组合逻辑和输出组合逻辑），从时序上看，这两部分组合逻辑完全可以看为一体。这样这条路径的组合逻辑就会比较繁杂，该路径的时序相对紧张。也就是说，两段式建模中用状态寄存器分割了组合逻辑，而三段式将寄存器移到组合逻辑的最后端。如果寄存器前的组合逻辑过于复杂，势必会成为整个设计的关键路径，此时就不宜再使用三段式建模，而要使用两段式建模了。解决两段式建模组合逻辑输出产生毛刺的方法是，额外在 FSM 后级电路中插入寄存器，调整时序，完成功能。

- **3 种描述 FSM 方法的比较**

一般来说，这 3 种 FSM 描述方法可以从表 6-1 中所示的几个方面进行比较。但是请读者注意，任何一种描述的优劣只是一般规律，而不是绝对性规律。例如一般来说不推荐使用一段式描述，但是如果 FSM 的结构十分简单，状态很少，状态转移条件和状态输出都十分简化，那么使用一段式建模的效率则会很高，这些经验需要读者逐步积累。

表 6-1　　3 种 FSM 描述方法的比较

比较项目	一段式描述方法	两段式描述方法	三段式描述方法
推荐等级	不推荐	推荐	最优推荐
代码简洁程度（对于相对复杂的 FSM 而言）	冗长	最简洁	简洁
always 模块个数	1	2	3
是否利于时序约束	不利于	利于	利于
是否有组合逻辑输出	可以无组合逻辑输出	多数情况下有组合逻辑输出	无组合逻辑输出
是否利于综合与布局布线	不利于	利于	利于
代码的可靠性与可维护度	低	高	最好
代码风格的规范性	低，任意度较大	格式化，规范	格式化，规范

6.2.4 状态机设计的其他技巧

本节讨论 FSM 设计的其他技巧。

- **FSM 的编码**

Binary（二进制编码）、gray-code（格雷码）编码使用最少的触发器，较多的组合逻辑，而 one-hot（独热码）编码则与之相反。one-hot 编码的最大优势在于状态跃迁时仅需要改变状态变量的某一个 bit，在一定程度上简化了比较逻辑，从而减少了毛刺产生的概率。相比之下，CPLD 的组合逻辑资源较为丰富，而 FPGA 的触发器资源较为丰富，所以 CPLD 多使用 gray-code 编码，而 FPGA 多使用 one-hot 编码。另一方面，对于小型设计而言使用 gray-code 和 binary 编码更有效，而对于大型状态机而言则使用 one-hot 编码更为高效。

在代码中添加综合器的综合约束属性或者在图形界面下设置综合约束属性可以较为方便地改变状态的编码。需要注意的是，Synplicity、Synopsys 和 Exemplar 等综合工具关于 FSM 综合约束属性的语法格式各不相同。

- **FSM 初始化状态**

一个完备的状态机（健壮性强）应该具备初始化状态和默认状态。当芯片加电或者复位后，状态机应该能够自动将所有判断条件复位，并进入初始化状态。需要注意的是，大多数 FPGA 都有 GSR（Global Set/Reset）信号，当 FPGA 加电后，GSR 信号默认对所有的寄存器、RAM 等单元置位或复位，这

时配置于 FPGA 的逻辑并未生效，所以不能保证正确地进入初始化状态，也就是说使用 GSR 进入 FPGA 的初始化状态，常常会产生种种不必要的麻烦。常规的做法是采用异步复位信号，当然也可以使用同步复位，但是要注意同步复位逻辑的设计。解决这个问题的另一种方法是将默认的初始状态的编码设为全零，这样当 GSR 复位后，状态机将自动进入初始状态。本章 6.2.3 小节中介绍的编码方法就是初始状态全零的 one－hot 编码方式。

- **FSM 状态编码定义**

状态机可以用 parameter 定义，但是不推荐使用`define 宏定义的方式，因为`define 宏定义在编译时会自动替换整个设计中所定义的宏，而 parameter 仅仅定义模块内部的参数，定义的参数不会与模块外的其他状态机混淆。例如一个工程里面有两个 module，它们各包含一个 FSM，如果设计时都有 IDLE 这一名称的状态，那么使用`define 宏定义就会发生混淆，而使用 parameter 则不会造成任何不良影响。

- **FSM 输出**

如果使用两段式 FSM 描述 Mealy 状态机，输出逻辑可以用"? 语句"描述，或者使用 case 语句判断转移条件与输入信号。如果输出条件比较复杂，而且多个状态共用某些输出，则建议使用 task/endtask 将输出封装起来，达到模块复用的目的。

- **阻塞和非阻塞赋值**

为了避免不必要的竞争，不论是做两段式还是三段式 FSM 描述，都必须遵循时序逻辑 always 模块使用非阻塞赋值“<=”的原则，即当前状态向下一状态时序转移和寄存 FSM 输出等 always 模块中都要使用非阻塞赋值；而组合逻辑 always 模块则使用阻塞赋值“=”，即状态转移条件判断和组合逻辑输出等 always 模块中都要使用阻塞赋值。

- **FSM 的默认状态**

完整的状态机应该包含一个默认（default）状态，当转移条件不满足或者状态发生突变时，要能保证逻辑不会陷入“死循环”，这是对状态机健壮性的一个重要要求，也就是常说的要具备“自恢复”功能。对应于编码上就是对 case 和 if...else 语句要特别注意，尽量使用完备的条件判断语句。Verilog 中，使用 case 语句的时候要用 default 建立默认状态。读者可能已经注意到，在上节实例中的 case 语句中，没有用 default 建立默认状态，其实我们可以将其中一个状态不编码，指定其为 default 默认状态，则任何与所列状态机不匹配的状态都会转到 default 状态，从而增强了 FSM 的健壮性，另外也可以添加一个额外的 default 状态，电路一旦运行到这个状态，就会自动跳转到 IDLE 状态，从新启动状态机，这样做也能增强状态机的健壮性。

```
case (CS)
    IDLE:   begin
                IDLE_out;
```

```
                    if (~i1)            NS = IDLE;
                    if (i1 && i2)       NS = S1;
                    if (i1 && ~i2)      NS = ERROR;
                end
        S1:     begin
                    S1_out;
                    if (~i2)             NS = S1;
                    if (i2 && i1)        NS = S2;
                    if (i2 && (~i1))     NS = ERROR;
                end
        S2:     begin
                    S2_out;
                    if (i2)              NS = S2;
                     if (~i2 && i1)      NS = IDLE;
                    if (~i2 && (~i1))    NS = ERROR;
                end
        ERROR:  begin
                    ERROR_out;
                    if (i1)             NS = ERROR;
                    if (~i1)            NS = IDLE;
                end
        default: begin
                    IDLE_out;
                    NS = IDLE;
                end
    endcase
end
```

在 case 语句结构中增加 **default** 默认状态是推荐的代码风格。

- **Full Case 与 Parallel Case 综合属性**

所谓 Full Case 是指 FSM 的所有编码向量都可以与 case 结构的某个分支或 default 默认情况匹配起来。如果一个 FSM 的状态编码是 8bit，则对应的 256 个状态编码（全状态编码是 2^n 个）都可以与 case 的某个分支或者 default 映射起来。

所谓 Parallel Case 是指在 **case** 结构中，每个 **case** 的判断条件表达式（即本章 6.2.2 小节中描述的 case_expression）有且仅有惟一一个 **case** 语句的分支（即本章 6.2.2 小节中描述的每个 case_item）与之对应，即两者关系是一一对

应关系。

目前知名的综合器（如 Synplify Pro、Precision RTL 和 Synopys 等）都支持“ synthesis full_case”和“ synthesis parallel_case”这些综合约束属性，合理使用 Full Case 约束属性，可以增强设计的安全性；合理使用 Parallel Case 约束属性，可以改善状态机译码逻辑。但是设计者必须具体情况具体分析，对于某些设计来说，不当使用这两条语句，会占用大量的逻辑资源，并恶化 FSM 的时序表现。

6.3 使用 Synplify Pro 分析 FSM

代码走读时分析 FSM 是一件比较耗时的事情，如果代码不符合本章 6.2 节所述的两段式或三段式 FSM 描述规范，走读他人代码则是一件异常痛苦的事情。这里笔者以 Synplify Pro 为例，介绍一下如何利于 EDA 工具分析、综合 FSM，提高 FSM 性能。

Synplify Pro 提供了 3 个有限状态机设计工具，分别为 FSM Compiler、FSM Explorer 和 FSM Viewer，灵活地使用这 3 个有限状态机工具分析、编译、优化 FSM，可使 FSM 的综合结果达到最优，下面逐一讨论它们的使用方法。

(1) 有限状态机编译器（FSM Compiler）。

一般的综合工具将 FSM 按照普通逻辑综合，而 Synplify Pro 与之不同。Synplify Pro 首先使用 FSM Compiler 将 FSM 编译为类似状态转移图的连接图，然后对 FSM 重新编码、优化，以达到更好的综合效果。

FSM Compiler 适用于有以下需求的场合：需要优化 FSM 设计，达到更好的综合效果；使用 FSM Viewer 调试状态机；使用 FSM Explorer 进一步优化有限状态机。

FSM Complier 的使用非常灵活，既可以对整个设计进行优化，也可以只对指定的状态机进行优化。使用 FSM Compiler 对整个设计进行优化，只需在主界面上的重要综合优化参数中选择【FSM Compiler】选项，或者在【综合优化参数设置】对话框中选中【FSM Compiler】选项即可。如果觉得对其他设计的 FSM 已经满意，而仅对某个 FSM 不满意，则可以在源代码或综合约束文件中手动添加综合属性，指定对某个状态机单独进行编译与优化。FSM Compiler 综合属性如表 6-2 所示。

表 6-2　FSM Compiler 综合属性

语法 / 语言	禁止优化所指定的 FSM	优化所指定的 FSM
Verilog	reg [3:0] curstate /* synthesis syn_state_machine=0 */;	reg [3:0] curstate /* synthesis syn_state_machine=1 */

(2) 有限状态机探测器（FSM Explorer）。

FSM Explorer 使用 FSM Compiler 的编译结果，鳞选不同的编码方式进行状态机编码试探，从而达到对 FSM 编码的最佳优化。与 FSM Compiler 相比，FSM Explorer 的优化效果往往更好，编译优化所花费的时间也更长。

对设计使用 FSM Explorer 的方法也有两种，第一种是对整个设计的所有 FSM 自动运用 FSM Explorer；第二种是对设计中特定的 FSM 使用 FSM Explorer。第一种方法可以在

Synplify Pro 主界面上的重要综合优化参数中选择【FSM Compiler】选项，或者在【综合优化参数设置】对话框中选中【FSM Explorer】选项；第二种方法需要在源代码或者综合约束文件中添加使用 FSM Explorer 的综合属性声明。FSM Explorer 综合属性如表 6-3 所示。

(3) 有限状态机观察器（FSM Viewer）。

在 Synplify Pro 中除了可以使用 RTL 视图和结构视图观察、分析 FSM 外，还可以使用专用的 FSM 观察器（FSM Viewer）分析 FSM。FSM Viewer 用状态转移图显示经编译优化的 FSM，非常直观，便于对 FSM 进行分析。

表 6-3　　FSM Explorer 综合属性

语法 / 语言	手动指定是否优化 FSM	手动指定 FSM 编码方式
Verilog	reg [3:0] curstate /* synthesis state_machine */ ;	reg [3:0] curstate /* synthesis syn_encoding "gray"*/

下面以 6.2.3 节中列举的 FSM 为例，介绍 FSM Viewer 的使用方法。

【例 6-2】 使用 FSM Viewer 分析有限状态机，参考示例详见随书光盘中“Example-6-1”目录下的相关内容。

启动 Synplify Pro，单击 Open Project... 按钮打开“Example-6-1\FSM\state2”目录下的“state2.prj”，单击 ⊕ 按钮启动 RTL 视图，选择状态机模块“statemachine”，单击 ⇅ 按钮进入状态机层次结构，或者单击鼠标右键，在弹出的命令菜单中选择【View FSM】命令，如图 6-8 所示。

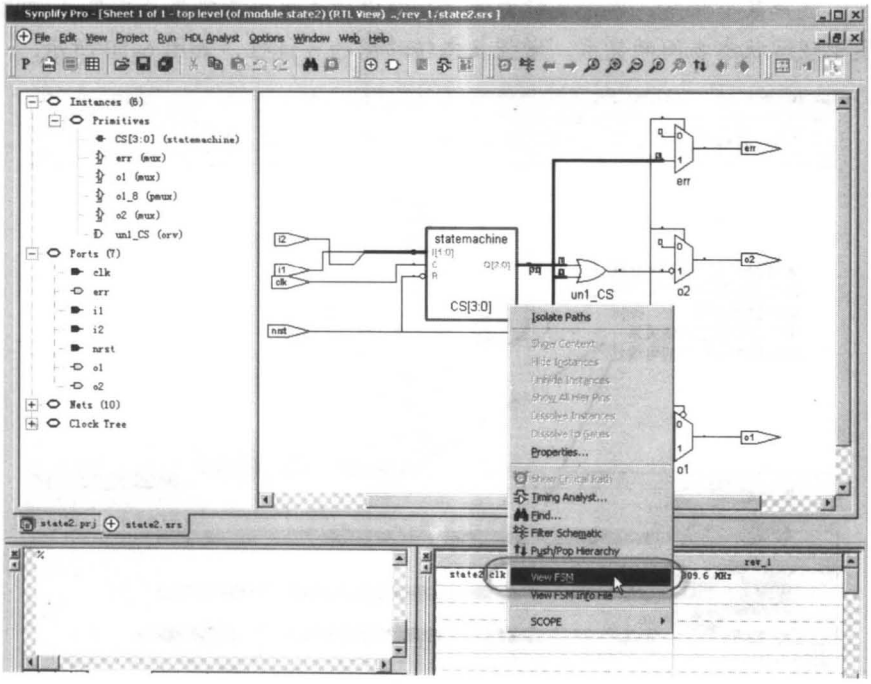

图6-8　启动 FSM Viewer 分析有限状态机

FSM Viewer 的主界面主要由状态转移图和 FSM 信息显示选项卡组成。状态转移图为源代码经过编译再现的状态机，FSM 信息显示包含转移条件（Transitions）、寄存器传输级状态编码（RTL Encodings）和映射后状态编码（Mapped Encodings）等 3 个选项卡，如图 6-9 所示。

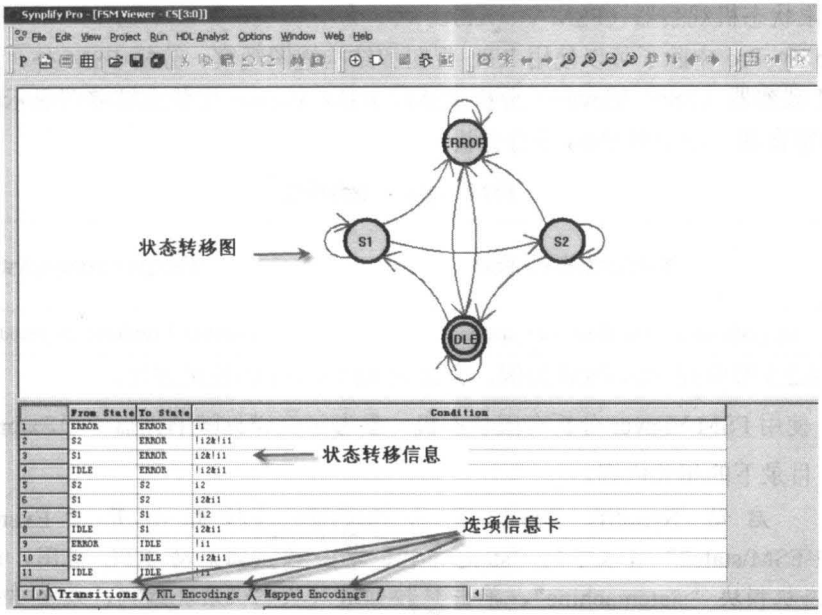

图6-9　FSM Viewer 主界面

选择某个状态，单击鼠标右键，在弹出的菜单中选择显示或屏蔽该状态，便于理解状态之间的关系，增强状态转移图的可读性，如图 6-10 所示。

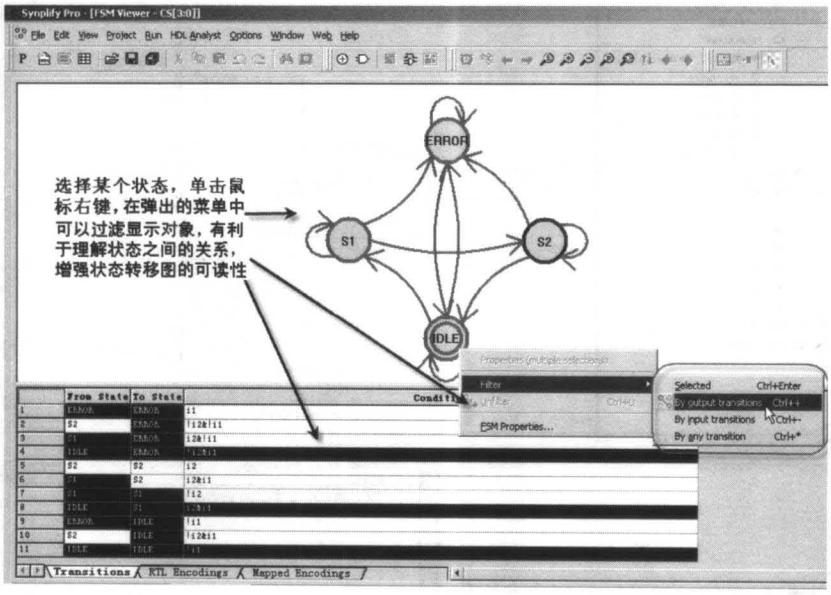

图6-10　在 FSM Viewer 中过滤显示对象

6.4 小结

状态机不仅仅是一种时序电路设计工具，它更是一种思想方法。状态机的本质就是对具有逻辑顺序或时序规律的事件进行描述。这个论断中最重要的两个词就是“逻辑顺序”和“时序规律”，这两点就是状态机所要描述的核心和强项，换言之，所有具有逻辑顺序和时序规律的事情都适合用状态机描述。本章在引入状态机设计思想的基础上，重点介绍了写好状态机的方法，希望读者认真学习。

6.5 问题与思考

1. 简述状态机的本质和适用的场合。
2. 状态机的基本要素有哪些？
3. 两段式、三段式 FSM 描述方法的基本结构如何？
4. FSM 描述何时使用阻塞赋值，何时使用非阻塞赋值？
5. 三段式 FSM 描述的两个 case 结构中，判断表达式与当前状态和下一个状态寄存器的对应关系如何？


读书笔记

第7章　逻辑验证与 Testbench 编写

验证是芯片设计过程中非常重要的一个环节。无缺陷的芯片不是设计出来的，而是验证出来的。验证过程是否准确与完备，在一定程度上决定了一个芯片的命运。

在芯片设计中，功能验证的方法主要有 3 种，分别为仿真、形式验证和硬件加速。本章将重点介绍仿真的概念、仿真平台的搭建，以及如何利用高效的仿真平台来验证设计等内容。

本章主要内容如下。

- 概述。
- 建立 Testbench，仿真设计。
- 实例：CPU 接口仿真。
- 结构化 Testbench 思想。
- 实例：结构化 Testbench 的编写。
- 扩展 Verilog 的高层建模能力。

7.1　概述

7.1.1　仿真和验证

验证是保证设计在功能上正确的一个过程。通常，设计和验证都有一个起点和一个终点。

设计的过程实际上是从一种形式到另一种形式的转换，比如从设计规格（也就是通常所讲的 Specification 或 SPEC）到 RTL 代码，从 RTL 代码到门级网表，从网表到版图（Layout）等。验证则是要保证每一步的设计转换过程准确无误。图 7-1 所示即为设计与验证的关系。

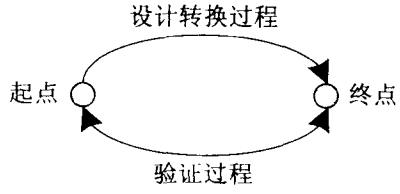

图7-1　设计与验证的关系

图 7-2 所示说明了从设计规格到门级网表的转换和验证过程。其中，从设计规格到 RTL 代码是由设计工程师手动完成的，而从 RTL 代码到门级网表则是由逻辑综合工具自动完

成的。

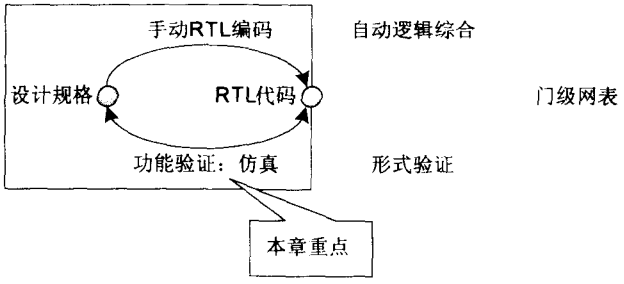

图7-2　设计与验证

相对应的是，要验证从设计规格到 RTL 代码的过程，就需要进行功能验证。功能验证一般是指验证 RTL 代码是否符合原始的设计需求和规格，这也是本章讨论的重点。

仿真的一般性含义是使用 EDA 工具，通过对实际情况的模拟，验证设计的正确性。由此可见，仿真的重点在于使用 EDA 软件工具模拟设计的实际工作情况。在 FPGA/CPLD 设计领域，最常用的仿真工具是 ModelSim。

目前，业界主流的功能验证方法是对 RTL 级代码的仿真。给设计增加一定的激励，观察响应结果。当然，这些仿真激励必须能够完整地体现设计规格，验证的覆盖率要尽可能高。

在传统的 ASIC 设计领域，验证是最费时耗力的一个环节，而对于 FPGA/CPLD 等可编程逻辑器件来说，验证的问题就相对简单一些。可以使用如 ModelSim 或 Active-HDL 等 HDL 仿真工具对设计进行功能上的仿真，也可以将一些仿真硬件与仿真工具相结合，通过软硬件联合仿真，加快仿真速度。我们还可以在硬件上直接使用逻辑分析仪、示波器等测量手段直接观察设计的工作情况。

本书的重点将放在 Verilog 语言本身的仿真上，至于门级网表和布线结果的功能（时序）仿真，以及硬件加速、逻辑分析仪等验证手段，请参考其他文献。

7.1.2　什么是 Testbench

Testbench，顾名思义就是测试平台的意思。简单来说，在仿真的时候 Testbench 用来产生测试激励给待验证设计（DUV），或者称为待测设计（DUT），同时检查 DUV 的输出是否与预期的一致，达到验证设计功能的目的，如图 7-3 所示。

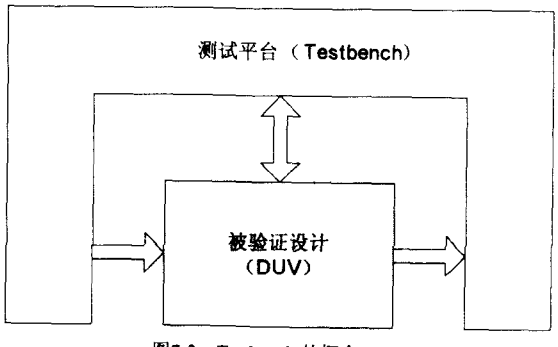

图7-3　Testbench 的概念

Testbench 概念的提出为我们提供了一个很好的验证芯片的平台。

仿真因 EDA 工具和设计复杂度的不同而略有不同，对于简单的设计，特别是一些小规模的 CPLD 设计来说，一般可以直接使用开发工具内嵌的仿真波形工具绘制激励，然后进行功能仿真。另外一种较为常用的方式是，使用 HDL（硬件描述语言）编制 Testbench（仿真文件），通过波形或自动比较工具，分析设计的正确性，并分析 Testbench 自身的覆盖率和正确性。

基于 Testbench 的仿真流程如图 7-4 所示。

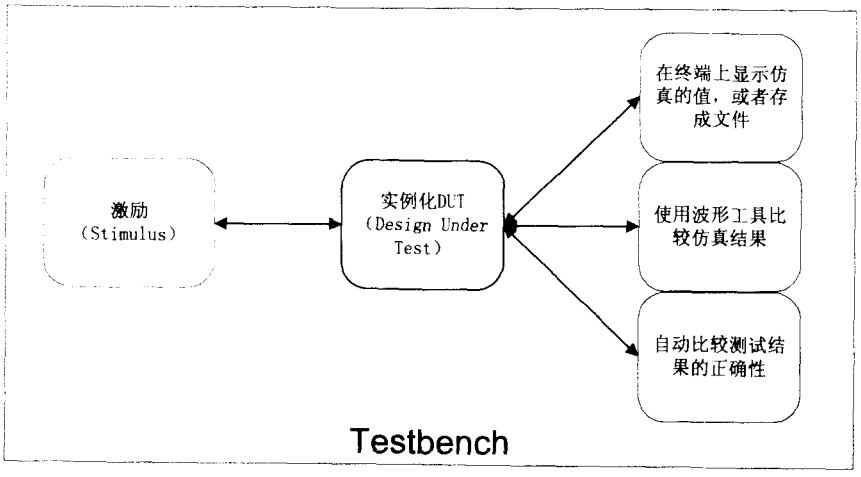

图7-4 基于 Testbench 的仿真流程

从图中可以清晰地看出 Testbench 的主要功能：

- 为 DUT（Design Under Test，待测试设计）提供激励信号；
- 正确实例化 DUT；
- 将仿真数据显示在终端或者存为文件，也可以显示在波形窗口中以供分析检查；
- 复杂设计可以使用 EDA 工具，或者通过用户接口自动比较仿真结果与理想值，实现结果的自动检查。

前两点功能主要和 Testbench 的编写方法或 Coding Style 相关，后两点功能主要和仿真工具的功能特性及支持的用户接口相关。如何编写规范、高效、合理的测试激励是本章所要论述的重点问题。

另外，一个 Testbench 设计好以后，可以为芯片设计的各个阶段服务。比如在对 RTL 代码、综合网表和布线之后的网表进行仿真的时候，都可以采用同一个 Testbench。

7.2 建立 Testbench，仿真设计

本章前一节已经叙述过，要仿真设计的功能，必须为其建立一个 Testbench，也叫仿真平台。

为了验证设计模块功能的正确性，我们通常需要在 Testbench 中编写一些激励给设计模块，同时观察这些激励在设计模块（DUV）中的响应是否与我们的期望值一致。

要充分验证一个设计，需要模拟各种外部的可能情况，特别是一些边界情况（corner

cases)，因为往往是这些边界情况最容易出问题。

图 7-5 中显示了一个 DUV。我们不仅需要产生时钟信号和复位信号，而且还需要编写一系列的仿真向量，同时观察 DUV 的响应，确认仿真结果。

关于这个设计的验证将在后面章节中重点介绍。

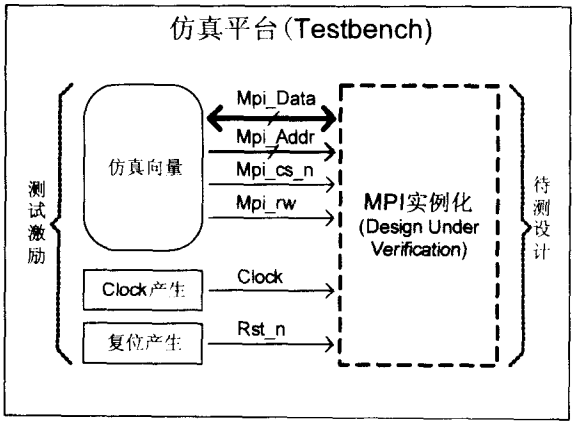

图7-5　用户验证 MPI 接口功能的仿真平台

7.2.1　编写仿真激励

对于初学者来说，迅速掌握一些测试激励的写法是非常重要的，这样可以有效地提高代码的质量，减少错误的产生。

一、仿真激励与被测对象的连接

本书第 3 章 3.4.1 小节介绍了结构化描述方法中模块实例的端口连接方法。

同样，在 Testbench 中也需要实例化被测试模块（DUV），DUV 的端口与 Testbench 中的信号互连也要遵循同样的规则，如图 7-6 所示。

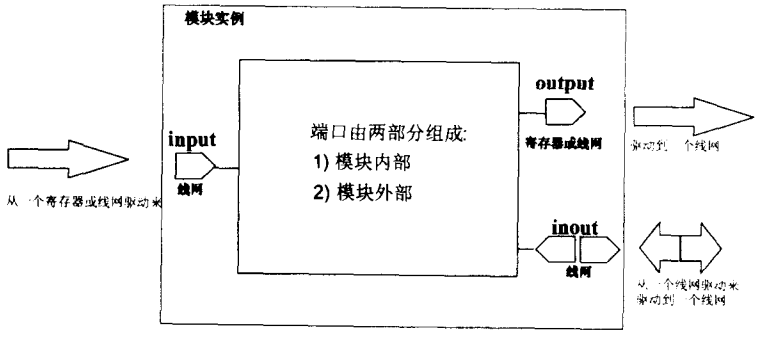

图7-6　模块实例端口连接规则

这里需要提醒读者注意的是，驱动双向信号的也一定是一个三态的线网，而不能是寄存器类型，否则就会发生冲突。

二、使用 initial 语句和 always 语句

initial 和 always 是两种基本的过程结构语句，在仿真开始时就相互并行执行。通常来

说，被动地检测响应时使用 always 语句，而主动地产生激励时则使用 initail 语句。

initial 和 always 的区别是 initial 语句只执行一次，而 always 语句不断地重复执行。但是，如果希望在 initial 里多次运行一个语句块，可以在 initial 里嵌入循环语句（如 while、repeat、for 和 forever 等），比如：

```
initial
begin
     forever   /* 永远执行*/
     begin
       ...
     end
end
```

而 always 语句通常只有在一些条件发生时才能执行，比如：

```
always @ (posedge Clock)
begin
   SigA = Sig B;
   ...
end
```

当发生 Clock 上升沿时，执行 always 操作，begin...end 中的语句顺序执行。

三、时钟、复位的写法

1. 普通时钟信号

用 initial 语句产生时钟的方法如下：

```
//产生一个周期为 10 的时钟
parameter FAST_PERIOD = 10 ;
reg Clock ;
initial
begin
   Clock = 0;
   forever
      # (FAST_PERIOD/2) Clock = ~ Clock ;
end
```

用 always 语句产生时钟的方法如下：

```
//用 always 语句产生一个周期为 10 的时钟
parameter FAST_PERIOD = 10 ;
reg Clock ;

initial
   Clock = 0;//将 Clock 初始化为 0
```

```
always
   # (FAST_PERIOD/2) Clock = ~ Clock;
```

以上写法所产生的波形如图 7-7 所示。

图7-7　产生的时钟测试激励

2. 非 50%占空比时钟信号

有时在设计中会用到占空比不是 50%的时钟，比如可以用 always 语句实现占空比为 40%的时钟，代码如下：

```
//占空比为 40%的时钟
parameter Hi_Time = 5,
          Lo_Time = 10 ;
reg Clock ;
always
begin
      # Hi_Time Clock = 0;
      # Lo_Time Clock = 1;
end
```

以上代码所产生的时钟波形如图 7-8 所示。

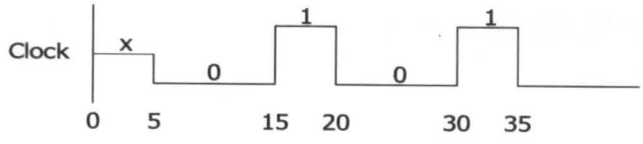

图7-8　占空比不是 50%的时钟

由于 Clock 在 0 时刻没有被初始化，而且 Clock 是寄存器类型的变量，因此在该信号的前 5ns，Clock 在仿真器中的值为 x。

当然也可以在 initial 语句中使用 forever 语句来描述同样的波形。

3. 固定数目时钟信号

如果需要产生固定数目的时钟脉冲，可以在 initail 语句中使用 repeat 语句来实现，代码如下：

```
//两个高脉冲的时钟
parameter PulseCount = 4, FAST_PERIOD = 10 ;
reg Clock ;
initial
```

```
begin
   Clock = 0;
   repeat (PulseCount)
      # (FAST_PERIOD/2) Clock = ~ Clock ;
end
```

以上代码产生了有两个高脉冲的时钟。

4. 相移时钟信号

另外一种应用较广的时钟是相移时钟，代码如下：

```
// 相移时钟
parameter HI_TIME = 5, LO_TIME = 10, PHASE_SHIFT = 2 ;
reg Absolute_clock ; //寄存器变量
wire Derived_clock ; //线网变量
always
begin
      # HI_TIME Absolute_clock = 0;
      # LO_TIME Absolute_clock = 1;
End
assign # PHASE_SHIFT Derived_clock = Absolute_clock;
```

这里首先使用 always 语句产生了一个 Absolute_clock 时钟，然后用 assign 语句将该时钟延时，产生了一个相移 2 的 Derived_clock 时钟，波形图如图 7-9 所示。

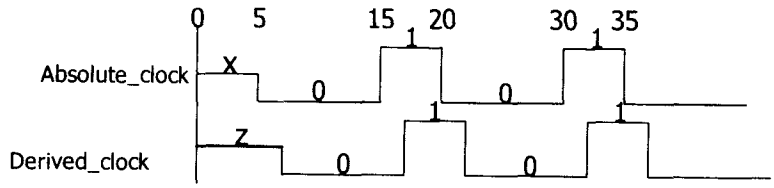

图7-9 Absolute_clock 和 Derived_clock

值得注意的是，图中的 Absolute_clock 为 register（寄存器）型变量，初始值为 x；而 Derived_clock 为 net（线网）型变量，初始值为 z。

5. 异步复位信号

复位信号不是周期信号，因此可以使用 initial 语句产生一个值序列。

```
//异步复位信号
parameter PERIOD = 10 ;
reg Rst_n ;
initial
begin
Rst_n = 1;
# PERIOD Rst_n = 0;
# (5* PERIOD) Rst_n = 1;
```

```
end
```

Rst_n 为低有效，以上代码在 10ns 时开始复位，复位持续时间为 50ns。

6. 同步复位信号

同步复位信号的实现代码如下：

```
//同步复位信号
initial
begin
    Rst_n = 1 ;
    @( negedge Clock) ;    // 等待时钟下降沿
    Rst_n = 0 ;
    # 30 ;
    @( negedge Clock) ;    // 等待时钟下降沿
    Rst_n = 1 ;
end
```

该代码首先将 Rst_n 初始化为 1，然后在第一个 Clock 的下降沿处开始复位。再延时 30ns，然后在下一个时钟下降沿处撤销复位。这样，复位的产生和撤销都避开了时钟的有效上升沿，因此这种复位可以认为是同步复位，如图 7-10 所示。

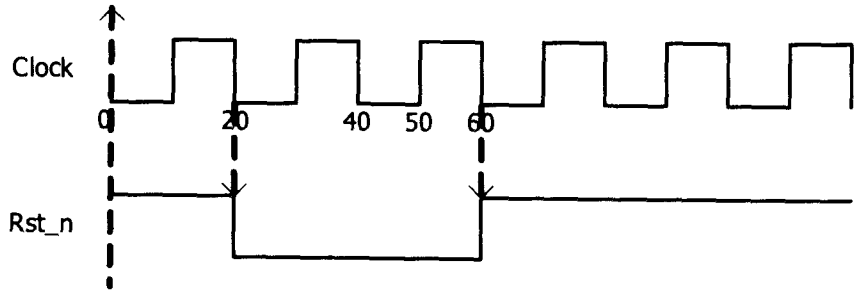

图7-10　同步复位

另一种同步复位信号的实现方法如下：

```
//同步复位信号
initial
begin
    Rst_n = 1 ;
    @( negedge Clock) ;    // 等待时钟下降沿
    Rst_n = 0 ; // 复位开始
    repeat  ( 3 )  @( negedge Clock) ;    // 经过 3 个时钟下降沿
    Rst_n = 1 ; // 复位撤销
end
```

首先将 Rst_n 初始化为 1，在第一个 Clock 的下降沿处开始复位，然后经过 3 个时钟下降沿，在第 3 个时钟下降沿处撤销复位信号 Rst_n，如图 7-11 所示。

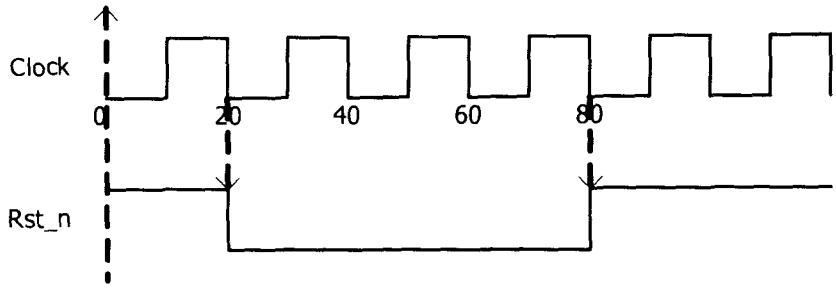

图7-11　另一种同步复位

四、产生值序列

下面使用 Verilog 语言描述图 7-12 中所示的一个波形。

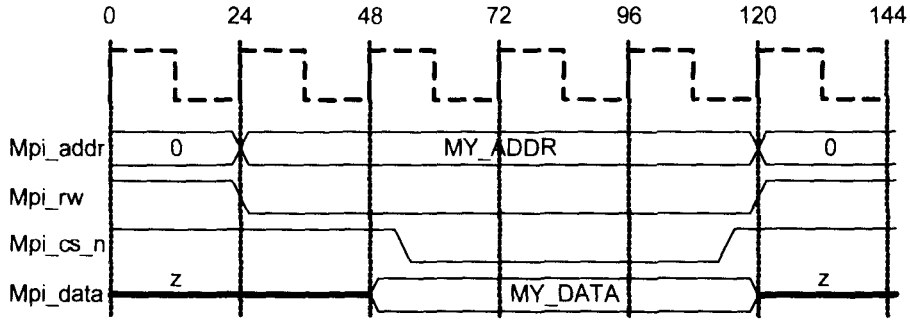

图7-12　一个波形图

该时序图是以 24ns 为刻度的，MY_ADDR 和 MY_DATA 都代表参数值。

使用 initial 语句块产生这样一组值序列，代码如下：

```
//值序列模型
Parameter SLOW_PERIOD = 24;
reg [5:0] Mpi_addr;
reg Mpi_cs_n;
reg Mpi_rw;
reg Mpi_oe;
tri [7:0] Mpi_data;
reg [7:0] Data_out;
assign Mpi_data = ( Mpi_oe )? Data_out : 8'bz ;
initial //在 initial 语句块中产生值序列
begin
      Mpi_addr = 0 ;
      Mpi_cs_n = 1 ;
      Mpi_rw = 1 ;
      Mpi_oe = 0 ;
```

```
        Data_out = 0 ;
        # SLOW_PERIOD;
        Data_out = MY_DATA;
        Mpi_addr = MY_ADDR;
        Mpi_rw = 0 ;
        # SLOW_PERIOD;
        Mpi_oe = 1 ;
        # (SLOW_PERIOD/4);
        Mpi_cs_n = 0 ;
        # ((2*SLOW_PERIOD) + (SLOW_PERIOD/2)) ;
        Mpi_cs_n = 1 ;
        # (SLOW_PERIOD/4);
        Mpi_addr = 0 ;
        Mpi_rw = 1 ;
        Mpi_oe = 0 ;
        # SLOW_PERIOD;
end
```

这里采用了阻塞赋值的方法来产生一系列的值变化。

值得注意的是，在输出 Mpi_data 的时候，并没有采用在 initial 中赋值后直接输出的方式，而是采用了输出使能控制的方式，用一个三态门输出。这样，我们将 Mpi_data 定义成一个 tri 类型的变量（Net 中的一种），以便于和被验证模块中的三态双向总线互连。

五、利用系统函数和系统任务

在编写 Testbench 时，一些系统函数和系统任务可以帮助我们产生测试激励，显示调试信息，协助定位。

比如使用 display 语句在仿真器中打印出地址和数据：

```
$display ("Addr: %b -> DataWrite: %d", Mpi_addr, Data_out);
```

同时也可以利用时序检查的系统任务来检查时序，例如：

```
$setup (Sig_D, posedge Clock, 1);//如果在 Clock 上升沿到达之前的 1ns 时间内 Sig_D 发生跳变，则将给出建立时间违反告警
$hold(posedge Clock, Sig_D, 0.1); //如果在 Clock 上升沿到达之后的 0.1ns 时间内 Sig_D 发生跳变，则将给出保持时间违反告警
```

另外，也可以利用$random()系统函数来产生测试激励数据，比如：

```
Data_out = {$random} % 256; //产生 0~255 的数据
```

$time 系统函数可以用来返回当前的仿真时间，协助仿真。

能够用于 Testbench 中的系统任务和函数有很多，它们的使用方法大同小异，非常简单。感兴趣的读者可以参考其他 Verilog 的语法资料。

六、从文本文件中读出和写入数据

在编写测试激励时，往往需要从已有的文件中读入数据，或者把数据写入到文件中，以

便做进一步分析。那么在 Verilog 语言中这是如何实现的呢？

先来看看如下代码：

```
reg [7:0] DataSource [0:47];//定义一个二维数组
$readmemh ( "Read_In_File.txt", DataSource );
```

该代码的含义是将 Read_In_File 文件中的数据读入到 DataSource 数组中，然后就可以直接使用这些数据了。

向文件中写入数据的代码如下：

```
integer Write_Out_File;//定义一个整数的文件指针
Write_Out_File = $fopen("Write_Out_File.txt");//打开文件
$fdisplay (Write_Out_File, "@%h\n%h", Mpi_addr, Data_in);//往文件中写入内容
$fclose (Write_Out_File); //关闭文件
```

七、并行激励

如果希望在仿真的某一时刻同时启动多个任务，可以采用 fork...join 语法结构。

例如在仿真开始 100 ns 后，如果希望同时启动发送和接收任务，而不是发送完毕后再进行接收，可以采用如下代码：

```
//并行激励
initial
begin
        #100 ;
        fork  //并行操作
             Send_task ;
             Receive_task ;
        join
end
```

这是产生并行激励的一个好办法。

八、利用 For 语句实现遍历测试

如果设计的工作模式很多，那么就需要对各种模式进行遍历测试，而遍历测试的工作量又非常大。

在很多时候，各种模式之间仅仅是部分寄存器配置的内容不同，而各种模式之间的测试都是一样的。有什么方法可以减轻这种遍历测试的工作量呢？我们可以采用 for 循环语句，用循环索引来传递各种模式的配置值，从而减少工作量，而且不会漏掉任何一种模式。

代码如下：

```
initial
begin
    for ( i = 0 ;  i < m ; i = i + 1 ) //遍历模式 1 至 m-1
        for  ( j = 0 ; j < n ; j = j +1 ) //遍历子模式 1 至 n-1
        begin
```

```
            case  ( j ) //设置每种模块的配置值
                0 : Conf_Value = a ;
                1 : Conf_Value = b ;
                2 : Conf_Value = c ;
                ...
             endcase
             //共用的测试向量
        end
end
```

读者在设计 Verilog 代码时灵活使用 for 语句，可以使代码更简洁。

九、高级语句 force 和 release

顾名思义，force 就是强制性为变量赋予确定的值，而 release 就是解除 force 的作用，将变量恢复为驱动源的值，例如：

```
//force 和 release
wire  a  ;
assign   a  =  1'b0 ;
initial
begin
    #10 ;
    force    a  =  1'b1 ;
    #10
    release  a ;
end
```

在 10 ns 时，a 的值由 0 被强制为 1，在 20ns 时，a 的值又恢复为 0 。

force 和 release 并不常用。有时可以利用它们和仿真工具进行简单的交互操作。例如，在 VerilogXL 的图形界面可以很方便地将一个变量 force 为 0 或 1。而在 Testbench 里，可以检测变量是否被 force 为固定的值，当被 force 为固定的值时就执行预定的操作，从而实现了简单的交互操作。

十、封装功能模块

与 C 语言类似，在编写 Testbench 的时候，可以将一些固定的操作封装成任务或者函数。

1. 任务（Task）

任务的格式如下：

```
task 任务名称;
    输入、输出声明;
    语句;
endtask
```

例如一个读 CPU 接口的任务：

```
task Read ;
output [ 7:0] Rtask_Data ;//data read out
input  [ 5:0] Rtask_Addr ;
begin
    uP_rw = 0 ;
    # SLOW_PERIOD;
    uP_addr = Rtask_Addr ;
    uP_rw = 1 ;
    # SLOW_PERIOD;
    …
end
endtask
```

其中，input 是任务的输入，可以用 output 来向任务外部传递计算结果。当然，也可以通过在任务中直接修改仿真中的全局变量来传递数值。调用 task 时的代码如下：

```
Read (Data, Addr);//Addr 是需要读的地址，而 Data 是读出的数据
```

2. 函数（Function）

函数的格式如下：

```
function [BITWIDTH - 1 : 0] 函数名称;
    输入声明;
    语句;
endfunction
```

与任务不同的是，函数将返回一个值。以上代码会返回一个 BITWIDTH 宽度的值。

例如：

```
function [7:0] Product ;
input [3:0] Sig_A;
input [3:0] Sig_B;
begin
   Product = Sig_A * Sig_B;
end
endfunction
```

调用的格式如下：

```
ProductResult = Product (A, B);//将 A 和 B 的乘积赋值给 ProductResult 变量
```

十一、波形编辑器

在一些工具中，可以使用波形编辑器来产生仿真激励。Quartus II 开发软件中的波形编辑器如图 7-13 所示。

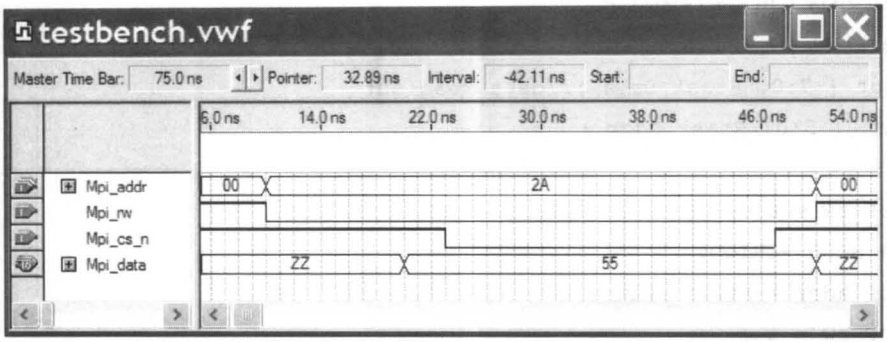

图7-13　Quartus II 工具中的波形编辑器

7.2.2　搭建仿真环境

通常，我们会为一个设计建立一个仿真平台，将这个设计在该平台中实例化，然后将在平台中产生的测试激励输入给设计模块，再观察 DUV 的响应是否与期望值相同。

下面将以图 7-14 中所示的 Testbench 为例，介绍端口的连接关系。

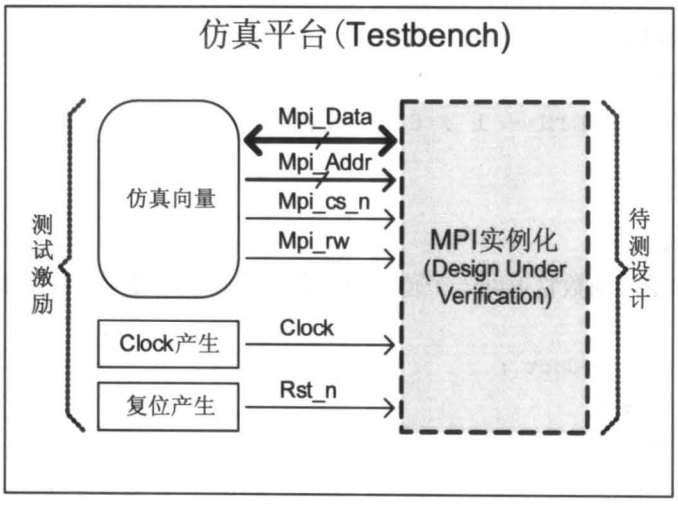

图7-14　搭建仿真环境

图中的仿真平台是单顶层的 Testbench，在后面还将介绍多顶层的 Testbench。

下面的部分代码是介绍如何搭建 Testbench 的。

```
module Testbench ;//测试平台顶层
//时钟激励产生
initial
begin
    ...
end
```

```
//复位激励产生
initial
begin
    ...
end

//各种测试用例
initial
begin
   ...
end

// 设计模块实例
MPI u_MPI (
    .Clock      (Clock),
    .Rst_n      (Rst_n),
    .Mpi_data   (Mpi_data),
    .Mpi_addr   (Mpi_addr),
    .Mpi_cs_n   (Mpi_cs_n),
    .Mpi_rw     (Mpi_rw)
    ) ;
endmodule
```

另外，在 Verilog 语言中也支持多顶层结构。关于多顶层的 Testbench，请参考本章 7.4.5 小节中的内容。

7.2.3 确认仿真结果

一、直接观察波形

最简单的确认仿真结果的方法就是用眼睛观察输出波形。

二、观察文本输出

也可以依靠一些系统任务打印的信息协助查看仿真结果，例如$display，直接输出到标准输出设备；$monitor，监控参数的变化；$fdisplay，输出到文件等。

三、自动检查仿真结果

对于一些大型设计而言，其测试向量成千上万，每条都用手动方式检查很不现实，这时就必须借助仿真软件接口进行自动比较。常用的自动比较方法有如下 3 种。

- 数据库比较法。首先需要生成一个标准向量数据库（Golden Vector Database），该数据库中存储的是期望得到的仿真结果，是比较的基础。然后自动的将每条仿真输出的响应向量与标准向量进行比较，并记录下不一致向量的位置和内容。这种方法的

优点是简单易行，其主要缺点是，根据输出的响应向量回溯并定位输入激励不是十分方便，也不够直观。

- 波形比较法。与数据库比较法的思路基本一致，只是比较的对象是仿真输出波形。首先存储标准波形文件（Golden Wave File），然后通过仿真软件手动或者自动将仿真的输出波形与标准波形文件进行比较，用图标（marker）定位比较结果相异的地方。这种方法的优点是直观明了。ModelSim 等仿真工具通常都支持波形比较（Wave Compare）功能。
- 动态自检测法。前面两种比较法的本质都是将仿真结果与事先存储好的标准向量（Golden Vector）进行比较，这两种方法统称为静态分析方法。对于复杂设计而言，仿真系统的输出不仅仅和当前输入相关，还和历史输入甚至反馈值相关，使用前面两种方法，即使发现了输出的响应向量和标准向量不一致，要定位造成不一致的原因，特别是追溯哪些输入造成的输出不一致也是比较困难的。与之相反，动态分析法就能实时地定位哪些激励造成响应不一致，其基本思路如图 7-15 所示。首先可以在不同的抽象层次（如行为级或者混合层次）描述出与 DUT 功能一致的仿真模型，然后读入测试激励向量（Test Vectors），将测试激励向量同时送到实例化的 DUT 和前面提到的仿真模型中，实时地观察、判断、存储两者的输出响应，比较输出结果。这样一旦发现输出响应不一致，即可暂停仿真过程，观察 DUT 和仿真模型的每个中间状态值，记录输入的激励向量，定位设计错误。

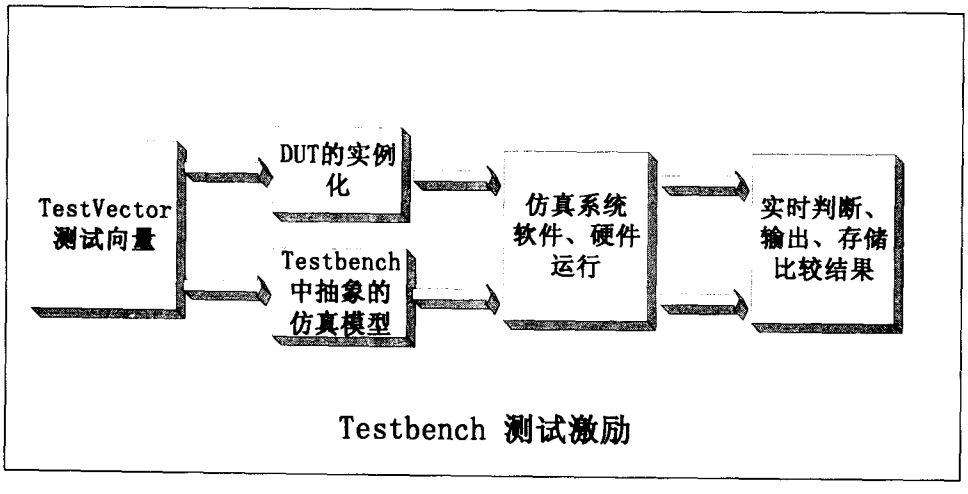

图7-15　动态自检测法

由于自动比较法通常用在非常庞大的设计验证中，因此不作为本书的重点。

四、使用 VCD 文件

VCD 文件是一种标准格式的波形记录文件，该文件只记录发生变化的波形。

可以将仿真器中的仿真结果输出成一个 VCD 文件，然后将该 VCD 文件输入给其他第三方分析工具进行分析。图 7-16 所示是 VCD 文件的使用流程。

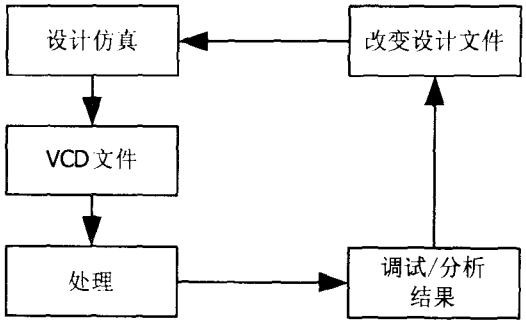

图7-16　VCD 文件调试和分析仿真过程

7.2.4　编写 Testbench 时需要注意的问题

一、Testbench 不是硬件

本书前面章节中曾经提到过，设计硬件时要尽量使用硬件的思维方式，时刻记住是在设计硬件，每一句语句都有明确的硬件定义，可以被综合工具理解。

需要注意的是，在编写 Testbench 的时候，情况就大不相同了。

通常，Testbench 不会被实现成具体的电路，不需要有可综合性，只要它能在仿真器中模拟出相应的功能即可。

因此，在编写 Testbench 的时候，需要尽量使用抽象层次较高的语句，这样不但可以提高编写效率，而且也可以提高仿真效率。

二、使用行为级描述方式描述 Testbench

读者必须明确，可综合的硬件电路一般要求用 RTL（寄存器传输级）方法描述，而 Testbench 则需要用行为级甚至更高层次的 HDL 语言描述。在讲述行为级描述的优点之前，首先引入 HDL 语言的层次概念，如图 7-17 所示。

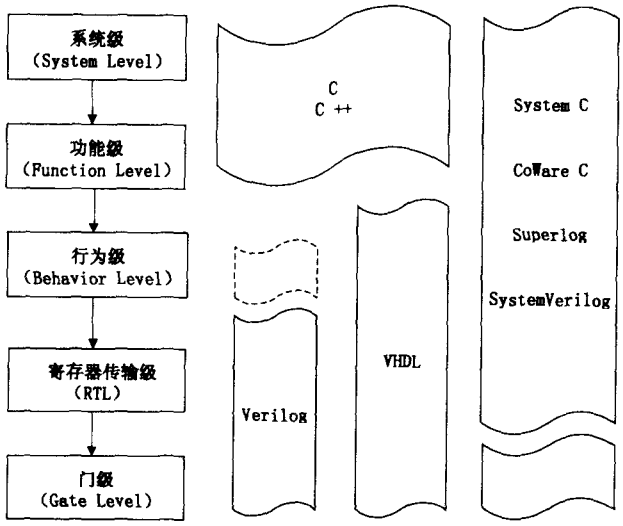

图7-17　HDL 语言的适用层次示意图

上图说明了不同的 HDL 语言所对应的 HDL 描述层次，图中的实线框表示适用程度较高，虚线框表示适用程度较低。常用的 HDL 描述层次有门级、寄存器传输级和行为级等。

使用行为级或更高层次描述方式的好处如下所述。

- 编写 Testbench 时仅需要关注电路的功能，而不需要理解电路的结构和实现方式，从而降低了设计 Testbench 的难度，节约了设计时间。
- 可以使用高级数据结构和运算。在行为级描述中，比较容易将某种运算封装起来，便于调用。另外如果使用可编程用户接口（PLI, Programmable Language Interface），还可以在 Testbench 中嵌入 C 和 C++语言。越来越多的仿真工具支持诸如 SystemVerilog、Superlog、System C 和 CoWare C 等高级语言。C、C++、SystemVerilog、Superlog、System C 和 CoWare C 等高级语言的引入，强有力地支持了用户自定义的数据结构，通过对象的封装，支持进程与事件之间通信，有效地提高了 Testbench 设计效率，并加强了 Testbench 的安全性。
- 行为级描述便于根据需要从不同层次抽象设计。与第 2 点相似，使用行为级或者更高层次的描述方式，可以将设计抽象到不同层次，在高层次描述设计会更加简便、高效，只有当需要解析某个部分的详细结构时，才使用低层次的详见描述，这样可以有效节约设计时间，提高仿真效率。
- 行为级仿真速度更快。行为级仿真速度更快的原因有两个，一是仿真工具支持某些高级算法，编译和运行速度快，二是行为级描述的抽象层次高，本身就是对运算处理的一种简化，例如在 RTL 级描述一个 32bit 乘法器时，需要反复进行选择、移位、与/或/非等运算；而在行为级描述这个 32bit 乘法器时，只需直接写“A * B”即可。仿真工具在仿真时也可以直接得到乘法结果，从而大大提高了效率，节约了时间。

三、设计高效的 Testbench

上一节主要从宏观的角度对行为级描述方法进行论述。具体到代码编写层次，希望读者平时注意积累一些标准、规范、高效的 Testbench 描述方法。要想设计出高效的 Testbench，需要注意以下几点：

- 避免使用无限循环。一般来说，Testbench 里面的每个事件都应该是可控制和有限的，否则会增加仿真器的 CPU 和 Memory 资源消耗，降低仿真速度。这条原则的一个特例是时钟产生电路，例如使用“forever”或无条件的“always”语句产生周期性时钟信号等；
- 使用逻辑模块划分激励。在 Testbench 中，所有的 initial、always 及 assign 等语法块是并行执行的，其中描述的每个事件都是基于时间“0”点安排的，这样通过这些语法结构将不同的激励划分开，将有利于设计维护测试激励；
- 避免不必要的输出显示。一些常用的仿真工具都支持将信息显示在终端上或者存储在文件中，这种功能对分析仿真结构十分有用。但是对于复杂设计而言，一定要避免不必要的输出显示，因为这类进程非常耗费 CPU 和 Memory 资源，会极大地降低仿真速度；
- 掌握程式化的仿真结构描述方法。诸如产生时钟信号，仿真双向总线，仿真 CPU 读写寄存器，定义事件的延时与顺序及 RAM 等常用模块的初始化、读、写过程等，都

是常用的仿真结构，大家已经形成了比较程式化的标准写法，初学者多读一些好的仿真代码，积累这些程式化的描述方法，可以有效地提高自己设计 Testbench 的能力。

7.3 实例：CPU 接口仿真

下面将通过一个常用的实例，引领读者深入学习如何编写 Testbench，如何验证设计。

7.3.1 设计简介

图 7-18 所示是一个 PowerPC 和 FPGA 的接口模型。FPGA 被当成处理器的一个简单异步外设处理。

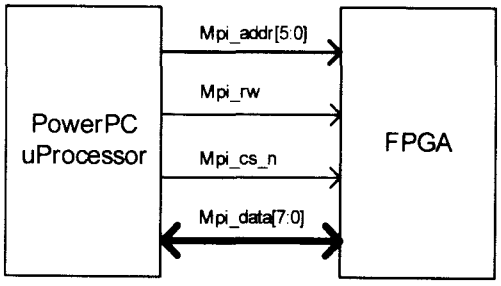

图7-18 微处理器和 FPGA 接口示意图

所有的总线操作都由 PowerPC 发起，共分为两种总线操作方式，即读和写。

写操作的时序如图 7-19 所示。

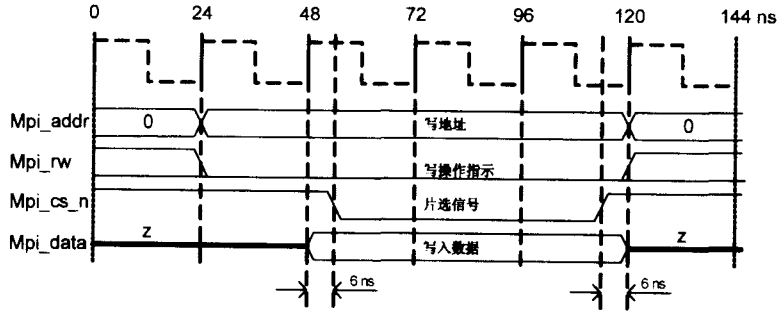

图7-19 写操作时序

读操作的时序如图 7-20 所示。

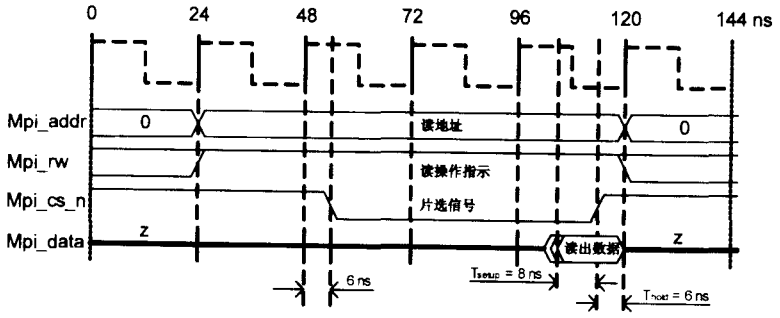

图7-20 读操作时序

需要注意的是，PowerPC 的读写时序与许多配置参数有关，而且处理器使用的主频不同，时序也会不太一样。这里为了简单明了，采用了特定的设置和主频。如果读者在实际的系统中采用不同的设置，则需要参考 PowerPC 的数据手册。

假设已经存在一个设计模块，其名称为 MPI，对应图 7-18 中所示的 FPGA，它的接口定义如下：

```
`timescale 1ns/100ps
module MPI( Clock, Rst_n, Mpi_data, Mpi_addr, Mpi_cs_n, Mpi_rw);
input           Clock ;
input           Rst_n ;
inout  [7:0]Mpi_data ;
input  [5:0]Mpi_addr ;
input           Mpi_cs_n ; //Chip Select
input           Mpi_rw ;   // 1:read; 0:write
endmodule
```

详细的设计代码参见随书光盘中“Example-7-1\Proj”目录下的相关内容。

该设计采用一个全局时钟 Clock 来处理 FPGA 与 PowerPC 之间的接口，将异步接口同步化。这是一种非常可靠的设计方法，希望读者能够领会其中的奥妙。

该设计中，输入地址采用 6 位，地址 0～31 为设计中的块状 RAM（8 位宽，32 个字节深度），地址 32～47 为设计中 D 触发器实现的 8 位寄存器，地址 48～63 保留未使用。

7.3.2　一种 Testbench

下面介绍一种 Testbench，该 Testbench 使用$random 产生激励，用$display 输出仿真结果。

要想验证上一节中介绍的 MPI 模块功能，首先需要为其产生激励，将数据写入到指定的地址，然后将该地址上的数据读出，并与写入的数据进行比较，如果一致，则说明设计正确，如果不一致，则说明设计有缺陷。

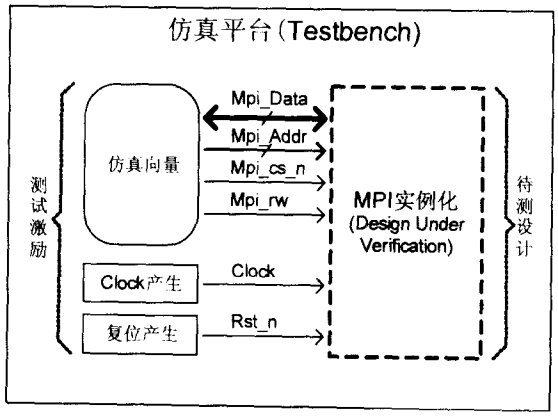

图7-21　Testbench 结构

根据图 7-21 中所示的 Testbench 结构设计 Testbench 顶层如下：

```
module NorTestBench ;//testbench 顶层模块
...
//时钟激励产生
initial
begin
Clock = 0;
 Forever
   # (FAST_PERIOD/2) Clock = ~ Clock ;
end
...
//复位激励产生
initial
begin
Rst_n = 1;
# FAST_PERIOD Rst_n = 0;
# (5*FAST_PERIOD) Rst_n = 1;
end

//输出三态 Buffer，用于和 MPI 接口的数据总线相连
assign Mpi_data = ( Mpi_oe )? Data_out : 8'bz ;

//仿真向量产生
initial
begin : ACCESS
//根据前面介绍的方法产生读写序列
for ( i=6'b101111; i>= 0; i=i-1 )//遍历 47～0 地址
begin
...
//用$random 系统函数产生写入的数据
Data_out = {$random} % 256; //data between 0~255
//打印出写入的地址数据信息
$display ("Addr: %b -> DataWrite: %d", Mpi_addr, Data_out);
...
//打印出读出的地址数据信息
$display ("Addr: %b -> DataRead: %d", Mpi_addr, Data_in);
...
$stop; //仿真停止
end
...
```

```
    $stop;
    end

    // 设计模块实例
    MPI u_MPI (
        .Clock      (Clock), //Clock 是寄存器型变量
        .Rst_n      (Rst_n), //Rst_n 是寄存器型变量
        .Mpi_data   (Mpi_data), // Mpi_data 是 tri 型变量
        .Mpi_addr   (Mpi_addr), // Mpi_addr 是寄存器型变量
        .Mpi_cs_n   (Mpi_cs_n), // Mpi_cs_n 是寄存器型变量
        .Mpi_rw     (Mpi_rw) // Mpi_rw 是寄存器型变量
        ) ;
    endmodule
```

以上是 Testbench 顶层结构示意，完整的代码请参考随书光盘中“Example-7-1\Proj”目录下的相关内容。

在该 Testbench 中，使用系统函数$random 产生 0～255 之间的随机数，然后从地址 47～0 写入，并在同一地址读出。

将写入的地址和数据以及读出的地址和数据用系统函数$display 输出到仿真标准输出设备中进行手动比较。

有了 Testbench 和设计模块后，即可对设计进行仿真验证了。

使用 Testbench 仿真，参考示例详见随书光盘中“Example-7-1”目录下的相关内容

1. 进入工程目录。

   将随书光盘中“Example-7-1”目录下的内容拷贝到本地硬盘中，例如可以拷贝到 E 盘的根目录下。运行 ModelSim 仿真器，在 ModelSim 仿真器的菜单中选择【File】/【Change Directory...】，然后选择目录 E:\Example-7-1\Proj，或者直接在 ModelSim>提示符后输入 cd E:/Example-7-1/Proj，回车。这样就将目录切换到了工程目录下。这里采用的是 ModelSim Altera 6.0c 版本，其他版本一样使用。

2. 运行仿真。

   在 ModelSim>提示符后输入 do sim.do，然后回车，开始进行仿真。

   “do”是 ModelSim 中的命令，而 sim.do 文件是笔者事先编写好的 ModelSim 自动运行脚本，其中包含了编译库文件、编译设计文件、载入仿真、开始运行仿真等命令，使得整个仿真过程自动完成，如图 7-22 所示。

3. 查看仿真结果。

   由于在这个 Testbench 中采用随机数作为写入数据，将写入和读出的数据都打印到标准的输出设备上，因此读者可以在 ModelSim 窗口中手动查看仿真结果。在仿真时，Testbench 中$display 系统函数的显示结果如图 7-23 所示。读

者只需查看其中的内容，即可检查读写数据是否一致。另外，还可以通过查看具体的波形来进行调试。

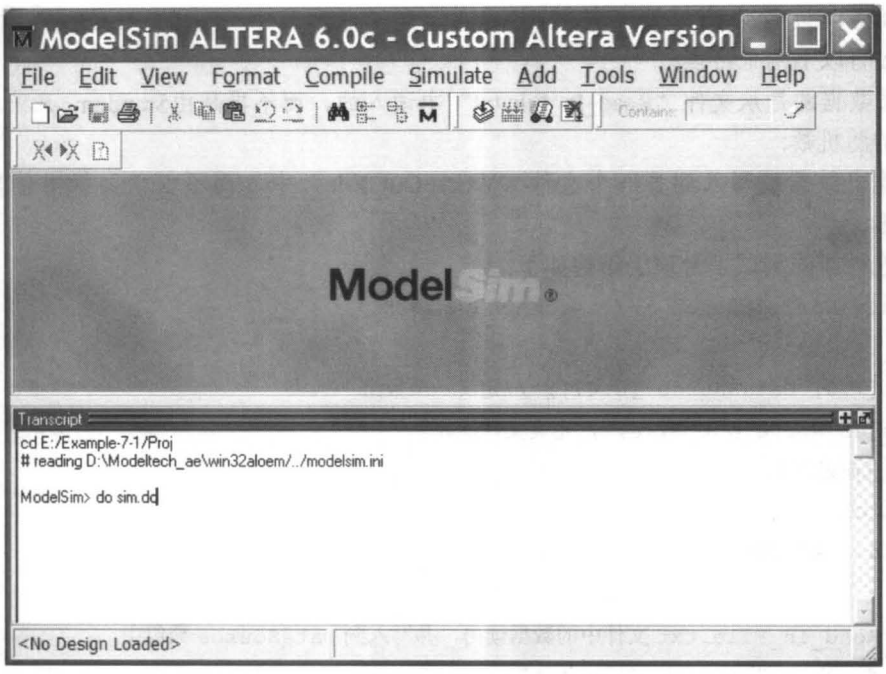

图7-22　ModelSim 窗口（1）

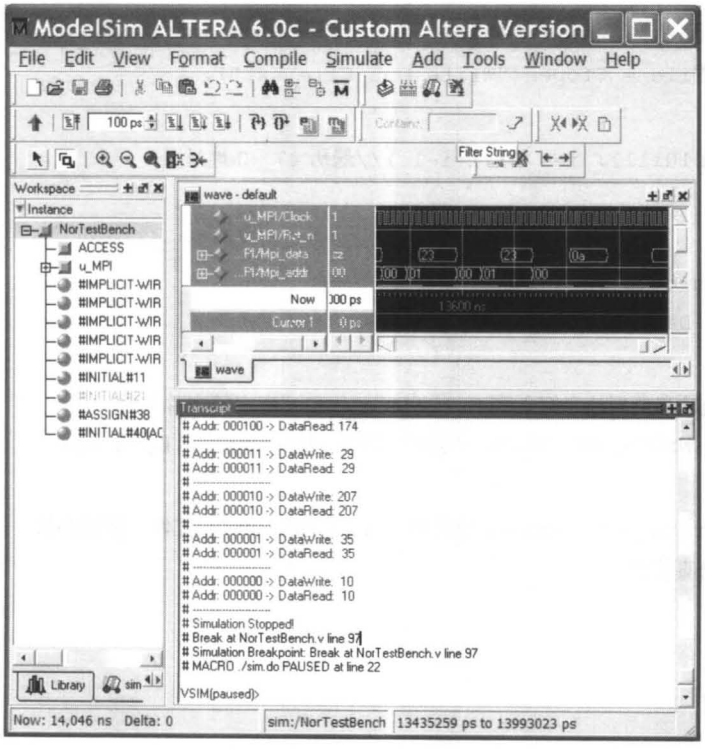

图7-23　ModelSim 窗口（2）

7.3.3　另外一种 Testbench

本节将向读者介绍另外一种 Testbench。与 7.3.2 节中介绍的 Testbench 相比，该 Testbench 有以下两个特点：

- 写数据源是从文件“Read_In_File.txt”中读入的，而不再是由$random 系统函数产生的随机数；
- 读出数据被写入到另一个文件“Write_Out_File”中，而不仅仅向标准输出设备输出。

这种新的测试平台顶层结构如下：

```
module NorTestBench ;//testbench 顶层模块
...
reg [7:0] DataSource [0:47]; //定义一个数组
integer Write_Out_File;// 定义文件指针
//仿真向量产生
initial
begin : ACCESS
...
//将 Read_In_File.txt 文件中的数据读出，并写入到 DataSource 数组中
$readmemh ( "Read_In_File.txt", DataSource );

//将 Write_Out_File.txt 文件打开，并将文件指针赋给 Write_Out_file
Write_Out_File = $fopen("Write_Out_File.txt");

for ( i=6'b101111; i>= 0; i=i-1 )//遍历 47～0 地址
begin
...
//从 DataSource 数组中取数据源
Data_out = DataSource[i] ;
...
//将读出的地址和数据信息写入到 Write_Out_File 指定的文件中
$fdisplay (Write_Out_File, "@%h\n%h", Mpi_addr, Data_in);
...
$fclose (Write_Out_File );//关闭 Write_Out_File 文件，释放指针
$stop; //仿真停止
end
...
$stop;
end
```

```
...
endmodule
```

在 Read_In_File.txt 文件中，根据 Verilog 的语法，存储如下数据：

```
@2f
24
@2e
81
@2d
09
...
```

地址由 2f～0 递减。

读出的地址和数据用系统函数$display 输出到仿真标准输出设备中，同时将输出数据和地址按照与 Read_In_File.txt 文件中一样的格式写入到文件 Write_Out_File.txt 中，以便于比较。运行仿真以后，在仿真目录下会生成一个“Write_Out_File.txt”文件，用来存储读出的地址和数据。此时只要将这两个文件打开，再手动进行比较，或者利用一些自动比较工具进行比较，就可以知道仿真结果正确与否。

具体的仿真步骤请参考 7.3.2 节中的操作步骤。工程文件在随书光盘中的“Example-7-2\Proj”目录下。

运行完仿真后，在工程目录下会生成一个文件 Write_Out_File.txt。可以将它与文件“Read_In_File.txt”进行比较，如果两者一致，则说明仿真结果正确，否则结果错误。此外读者还可以手动查看仿真波形来进行调试。

7.4 结构化 Testbench

在 7.3 节中介绍了两种验证 MPI 模块的 Testbench。实际上，在 Testbench 中，对 MPI 接口进行的操作主要有两种，即写操作和读操作。

如果将产生这两种操作的时序功能模块作为一种标准功能模块，而将要操作的地址和数据等作为参数去调用这种总线功能模块，会有什么好处呢？

这样的话，无论对 MPI 进行什么样的读写操作，只要调用这种通用的总线模块（就是后面将要介绍的 BFM），同时将读写的地址和数据代入即可，操作简单，代码容易维护，读与写的功能模块也得到了重用，这就是要结构化 Testbench 的原因。

结构化 Testbench 不仅使得 BFM 和测试用例分离，而且还将测试套具和测试用例分离开，不同的测试用例之间也是相互独立的，如图 7-24 所示。

结构化 Testbench 的好处是：

- 使总线功能模块（BFM）得到重用；
- 结构清晰，易于设计，减少了工作量；
- 提高了 Testcase 的抽象程度，而无需关心底层细节；
- 适用于复杂的设计模块。

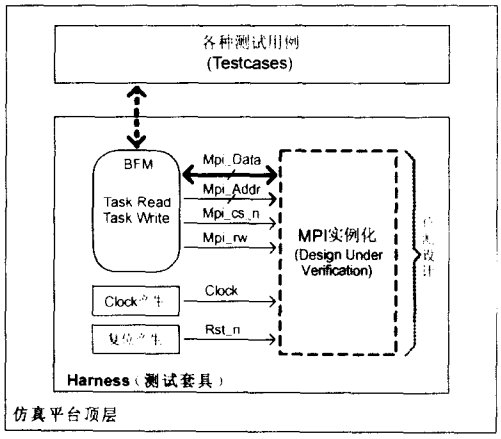

图7-24　结构化 Testbench 示意图

7.4.1　任务和函数

本章 7.2.1 小节中介绍了 Verilog 中的两种功能封装方法，即任务（Task）和函数（Function），在下一节中的 BFM 设计部分，正是利用了这种功能封装的概念。

7.4.2　总线功能模型（BFM）

Bus Functional Model（BFM），总线功能模型，是一种将物理的接口时序操作转化成更高抽象层次接口的总线模型。

以本章 7.3 节中的设计为例，为了验证 FPGA 中 MPI 接口的功能，需要给 MPI 接口添加各种各样的激励，仿真激励的种类越多，仿真越完备。

这里需要在 Testbench 中模拟 PowerPC 接口的时序和功能。

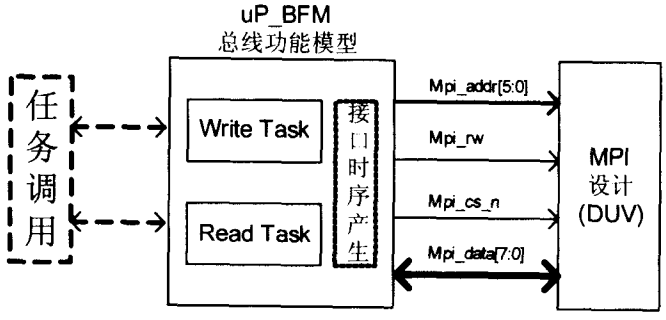

图7-25　BFM 示意图

在图 7-25 中，uP_BFM 是作为一个 Verilog HDL 中的 module 出现的。其中，uP_BFM 模拟了 PowerPC 访问异步外设的时序，与 MPI 设计模块（DUV）对接，该接口具有精确的延时特性。在该模型的内部有两个用户定义的任务，即 write 和 read，供其他模块（如测试激励）调用。

这样，测试激励（Testcases）就不需要关心接口时序，而只需能调用 write 和 read 两个任务即可，实现底层时序的功能则交给了 uP_BFM。

7.4.3 测试套具（Harness）

所谓测试套具（Harness），从系统测试的角度来说，就是将被测模块固定，以方便测试，而从验证 Verilog 代码的角度来说，就是将被测试模块封装起来，留出简单易用的访问接口，以便于各种测试用例调用及测试设计模块等。

如图 7-26 所示，Harness 中实例化了 uP_BFM、DUV，以及一些基本的激励，如 Clock 和复位的产生。对外，测试激励可以通过 uP_BFM 中的任务，来对 DUV 施加各种激励。

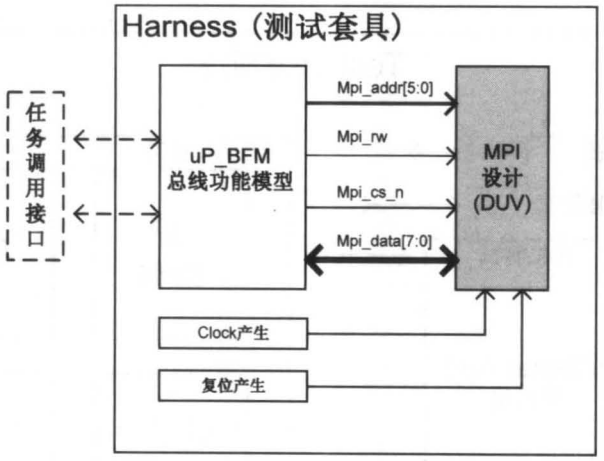

图7-26 Harness 测试套具

7.4.4 测试用例（Testcase）

在上一节中介绍了测试套具，即 Harness。当把被测模块固定好以后，就需要利用各种各样的测试用例来进行测试，通常将这些测试用例称为 Testcase，如图 7-27 所示。

Testcase 中的各种用例可以通过图中的虚线箭头来调用 Harness 中的任务。关于测试用例和 Harness 之间的层次关系，请参考下一节中的内容。

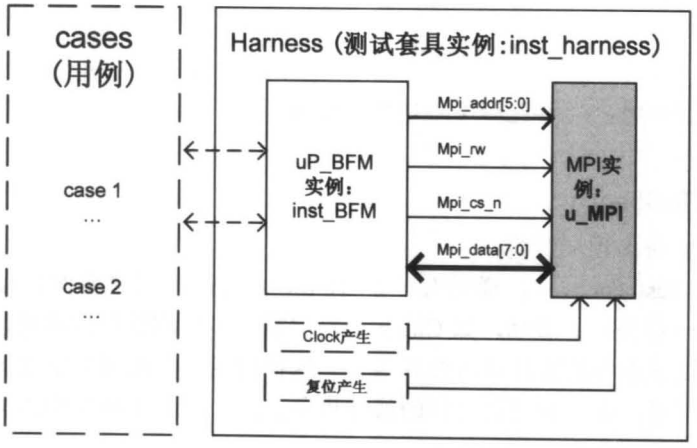

图7-27 Testcase 测试用例

另外，设计测试用例时还需要考虑测试用例的可扩展性和独立性。

7.4.5　结构化 Testbench

在这一节中将讨论测试用例和 Harness 的层次关系。

一、单顶层 Testbench

在单顶层的 Testbench 中只有一个顶层，Harness 的实例以及各种测试用例都在顶层中，如图 7-28 所示。

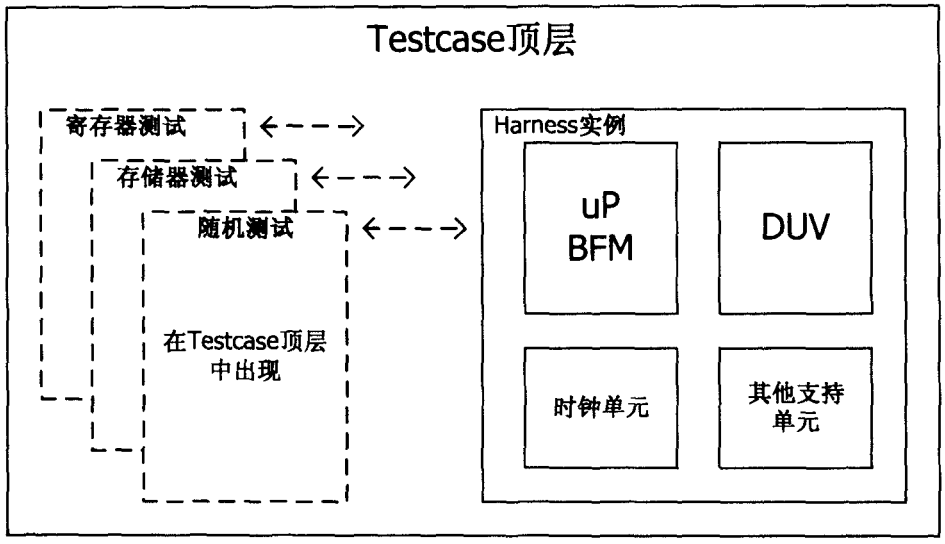

图7-28　单顶层的 Testbench 结构

假设 Harness 在 Testbench 顶层中的实例名为 inst_harness，Harness 实例化进来的模块 uP_BFM 里面有一个任务 SEND_DATA，该任务可以产生激励输入到 DUV，则在 Testcase 里调用该任务就可写为：

```
initial
begin
   ...
   inst_harness . uP_BFM . SEND_DATA ( ... ) ;
end
```

二、多顶层 Testbench

Verilog 语言支持多顶层结构。

在多顶层的 Testbench 中，通常有一个 Harness 顶层，用来实例化设计模块，实例化 BFM，以及提供一些基本的激励，如 Clock 和复位等。多个测试用例都可以作为顶层，不同的测试用例用来测试不同的特性或者边界条件。这样的话，如果要增加或减少用例，只需要增加或减少文件即可，而且对现有文件的修改也非常少，有利于防止引入人为的错误，如图 7-29 所示。

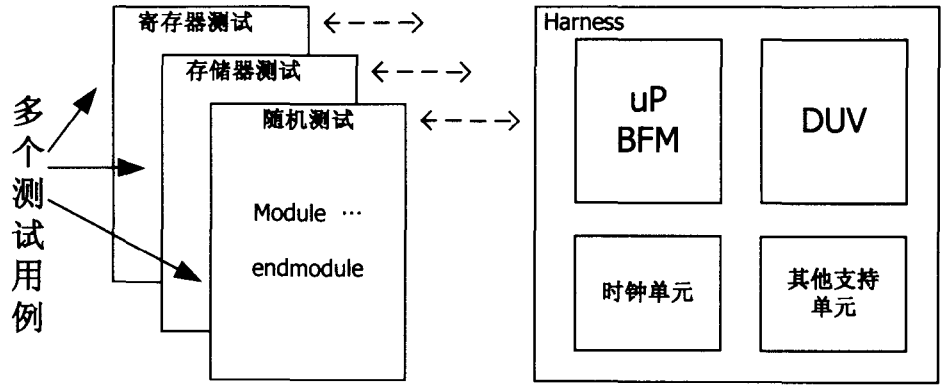

图7-29 多顶层的 Testbench 结构

Harness 顶层是由 DUV 和一些接口模型（BFM）构成的一个狭义上的测试平台，其他测试用例模块可以调用 BFM 里面的 Task、Function 等，向 DUV 施加激励。

注意，这些顶层之间是没有端口映射的，它们之间的互相调用和访问是通过层次路径名的方式实现的，图 7-29 中的虚线表示层次路径名的访问。

Verilog HDL 的顶层类似于 C 的结构体，而实例化的模块、任务、函数、变量等就是结构体里的成员，可以通过点（ . ）隔开的方式访问结构体里面的每一个成员。

顶层 Harness 实例化进来的模块 uP_BFM 里有一个任务 SEND_DATA，该任务可以产生激励输入到 DUV，在每个 Testcase 的 module 文件中调用该任务则可写为：

```
initial
begin
   ...
   Harness . uP_BFM . SEND_DATA ( ... ) ;
end
```

多顶层结构的可扩展性和重用性要比单顶层结构强得多。层次路径的访问方式非常有用，希望读者能够掌握它的用法。

三、Testbench 设计思想

验证的完备性对于 Testbench 的设计来说意义非常重大，在设计 Testbench 时需要清楚要验证什么特性，怎么才能将各种设计的特性转化为设计的激励。

以下是设计一个 Testbench 的基本思想和步骤，供读者参考。

- **从需求（SPEC）规格到特性（Features）：从设计的规格提取设计的特性。**
- **从特性到用例（Testcase）：根据设计特性，编写出相应的测试用例来验证该特性。**
- **从测试用例到 Testbench：根据精心设计的测试用例搭建出测试平台（Testbench）。**

Testbench 不是凭空设计的，它的根本起点是设计的需求规格。验证工程师的基本工作，正是要验证逻辑设计工程师的设计是否满足需求规格。

7.5 实例：结构化 Testbench 的编写

前面主要介绍了结构化 Testbench 的思想、基本组成部分和优点，这里将通过两个实例，带领读者进入实战阶段。

7.5.1 单顶层 Testbench

单顶层 Testbench 的结构如图 7-30 所示。

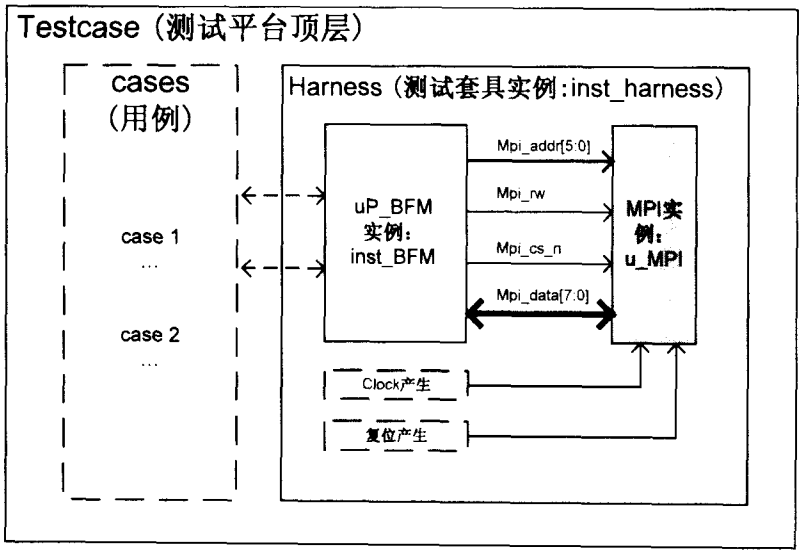

图7-30　单顶层 Testbench 的结构

在单顶层 Testbench 中实例化 Harness，同时写了两个测试用例来调用 uP_BFM 实例中的读写任务。

其中 Harness 中包括 uP_BFM 和 MPI 设计的实例化模块，以及一些基本的激励。

Testcase 顶层代码如下：

```
`timescale 1ns/100ps
module testcase ;
harness inst_harness (); //harness 实例，没有端口
reg    [7:0] Data_out;
reg    [7:0] Data_in;
reg    [7:0] DataSource [0:47];
integer      Write_Out_File;
initial
begin: MYCASE
   integer i;
   # 222 ;
   // testcase 1:
```

```
    for ( i=6'b101111; i>= 0; i=i-1 )
        begin
            Data_out = {$random} % 256; //data between 0~255
            inst_harness.inst_BFM.Write(Data_out, i);
            $display ("case1, Addr: %h -> DataWrite: %h", i, Data_out);
            inst_harness.inst_BFM.Read(Data_in, i);
            $display ("case1, Addr: %h -> DataRead: %h", i, Data_in);
            $display ("------------------------");
        end

    $readmemh ( "Read_In_File.txt", DataSource );
    Write_Out_File = $fopen("Write_Out_File.txt");
    // testcase 2
    for ( i=6'b101111; i>= 0; i=i-1 )
        begin
            Data_out = DataSource[i] ;
            inst_harness.inst_BFM.Write(Data_out, i);
            $display ("case2, Addr: %h -> DataWrite: %h", i, Data_out);
            inst_harness.inst_BFM.Read(Data_in, i);
            $display ("case2, Addr: %h -> DataRead: %h", i, Data_in);
            $display ("------------------------");
            $fdisplay (Write_Out_File, "@%h\n%h", i, Data_in);
        end
    $fclose ( Write_Out_File );
    $display ("Simulation Finished!");
    $stop;
end
endmodule
```

Harness 的模块结构如下：

```
`timescale 1ns/100ps
module harness ;
// Clock Stimulus generation
...
// Reset Stimulus generation
...
tri  [7:0] Mpi_data;
wire [5:0] Mpi_addr ;
wire       Mpi_cs_n;
wire       Mpi_rw;
```

```
// BFM 模块实例
uP_BFM inst_BFM (
    .uP_data    (Mpi_data),
    .uP_addr    (Mpi_addr),
    .uP_cs_n    (Mpi_cs_n),
    .uP_rw      (Mpi_rw) );
// MPI 设计模块实例
MPI u_MPI (
    .Clock      (Clock),
    .Rst_n      (Rst_n),
    .Mpi_data   (Mpi_data),
    .Mpi_addr   (Mpi_addr),
    .Mpi_cs_n   (Mpi_cs_n),
    .Mpi_rw     (Mpi_rw) ) ;
endmodule
```

在 Testcase 顶层中，Harness 的实例名为 inst_harness，而在 Harness 中，uP_BFM 的实例名为 inst_BFM。因此，当测试用例调用 uP_BFM 中的 Write 和 Read 任务时，采用如下方法：

```
inst_harness.inst_BFM.Write(Data_out, i); //Data_out 为写数据，i 为写地址
inst_harness.inst_BFM.Read(Data_in, i); //Data_in 为读数据，i 为读地址
```

具体的仿真步骤请参考 7.3.2 节中的操作步骤。工程文件在随书光盘中的“Example-7-3\Proj”目录下。仿真运行结果如下：

```
# case1, Addr: 0000002f -> DataWrite: 24
# case1, Addr: 0000002f -> DataRead: 24
# ----------------------
# case1, Addr: 0000002e -> DataWrite: 81
# case1, Addr: 0000002e -> DataRead: 81
# ----------------------
...
# case2, Addr: 0000002f -> DataWrite: 24
# case2, Addr: 0000002f -> DataRead: 24
# ----------------------
# case2, Addr: 0000002e -> DataWrite: 81
# case2, Addr: 0000002e -> DataRead: 81
# ----------------------
...
```

这里的 case1 延用的是 7.3.2 节中的测试激励，数据源为随机数，而 case2 则为 7.3.3 节中的测试激励，数据源是从文件“Read_In_File.txt”中读出的。将两种数据的测试用例合并到一个文件的不同用例下，检查仿真结果的方法相同。

7.5.2 多顶层 Testbench

多顶层 Testbench 的结构如图 7-31 所示，多个 Testcase 文件都可以调用 Harness 中的读写任务。

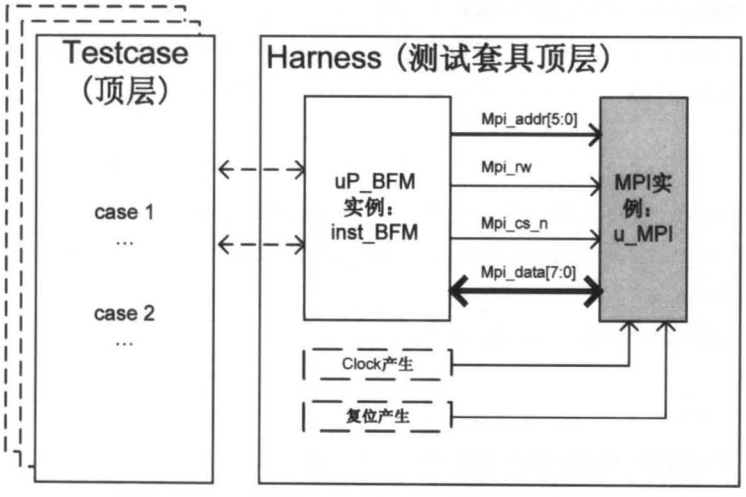

图7-31 多顶层 Testbench 的结构

在 Testcase 代码中如果需要调用 Harness 下 uP_BFM 实例中的读写任务，则需要在 Testcase 中写出如下代码：

```
harness.inst_BFM.Write(Data_out, i); //Data_out 为写数据，i 为写地址
harness.inst_BFM.Read(Data_in, i); //Data_in 为读数据，i 为读地址
```

ModelSim 中调用多个顶层仿真的命令为 vsim -L altera_mf testcase harness。

在多顶层结构中，Harness 代码是不变的，只有 Testcase 需要作相应的修改。代码修改如下：

```
`timescale 1ns/100ps
module testcase ;
reg    [7:0] Data_out;
reg    [7:0] Data_in;
reg    [7:0] DataSource [0:47];
integer      Write_Out_File;
initial
begin: MYCASE
   integer i;
   # 222 ;
   // testcase 1:
   for ( i=6'b101111; i>= 0; i=i-1 )
```

```
        begin
            Data_out = {$random} % 256; //data between 0~255
            harness.inst_BFM.Write(Data_out, i);
            $display ("case1, Addr: %h -> DataWrite: %h", i, Data_out);
            harness.inst_BFM.Read(Data_in, i);
            $display ("case1, Addr: %h -> DataRead: %h", i, Data_in);
            $display ("------------------------");
        end

    $readmemh ( "Read_In_File.txt", DataSource );
    Write_Out_File = $fopen("Write_Out_File.txt");
    // testcase 2
    for ( i=6'b101111; i>= 0; i=i-1 )
        begin
            Data_out = DataSource[i] ;
            harness.inst_BFM.Write(Data_out, i);
            $display ("case2, Addr: %h -> DataWrite: %h", i, Data_out);
            harness.inst_BFM.Read(Data_in, i);
            $display ("case2, Addr: %h -> DataRead: %h", i, Data_in);
            $display ("------------------------");
            $fdisplay (Write_Out_File, "@%h\n%h", i, Data_in);
        end
    $fclose ( Write_Out_File );
    $display ("Simulation Finished!");
    $stop;
end
endmodule
```

细心的读者也许已经发现，本例中 Testcase 调用 Harness 中任务的格式如下：

```
harness.inst_BFM.Write(Data_out, i);//直接引用 harness 模块名
```

而单顶层 Testbench 中的格式如下：

```
inst_harness.inst_BFM.Write(Data_out, i);//引用 harness 在 testcase 中的实例名 inst_harness
```

具体的仿真步骤请参考 7.3.2 节中的操作步骤。工程文件在随书光盘中的“Example-7-4\Proj”目录下。

7.6 扩展 Verilog 的高层建模能力

一些 Verilog 的高级用户在使用 Verilog 编写 Testbench 时常常感到它的文件输入输出操作，以及高级抽象的数学功能比 C 语言要弱，毕竟 Verilog 是一种硬件描述语言。

不过，Verilog 还是开放了一个接口叫做 PLI（编程语言接口），专门用来和 C 语言的程序通信。这样借助 PLI，Verilog 高层建模功能上的不足就可以得到弥补。

目前，业界绝大部分的仿真器都支持 PLI。

那么 PLI 是如何工作的呢？这里简单说明一下。例如，在 PC 上运行了一个 Verilog 的仿真器，同时还运行着一个用 C 语言编写的应用程序。该应用程序可以通过 PC 的一个内存空间与 Verilog 仿真器交换数据。应用程序产生的激励可以放在内存中，等待仿真器取走数据并将结果写回后，应用程序再将写回的结果取走，这样就完成了一个由 C 应用程序扩展 Verilog 语言能力的过程。

关于 PLI，这里不再详细阐述，感兴趣的读者可以参考其他文献。

7.7 小结

本章主要介绍了验证和仿真的基本概念，重点讲解了如何建立 Testbench，如何搭建仿真环境，以及如何确认仿真结果等内容。

另外需要补充一点，Testbench 是可以继承的。虽然本章将重点放在 RTL 级的代码仿真上，但同样的 Testbench 在门级仿真或者后仿真阶段也可以使用。

7.8 问题与思考

1. 什么是验证？
2. Testbench 是测试激励吗？
3. 如果需要设计并行激励，应使用什么语言元素？
4. 在 Verilog 中如何从文件中读入数据？
5. 简述单顶层 Testbench 和多顶层 Testbench 的联系和区别。


读书笔记

第8章 Verilog 语义和仿真原理

我们平时介绍 Verilog 的时候，总喜欢使用“简单易学”这个词。的确，Verilog 语言与其他硬件描述语言相比，更简洁，更贴近硬件特性，更容易被初学者理解。

但是任何事物都具有两面性，Verilog 也经常有让用户摸不着头脑的时候。比如初学者经常会遇到一些奇怪的问题：

同一个设计，在不同的 Verilog 仿真器中仿真，结果却不一致；

一个设计模块如果没有加上延时单位，仿真结果就不正确。

同时，我们也经常听到一些简单化的设计经验：

尽量使用阻塞赋值描述组合逻辑；

在设计模型时建议加入延时，防止冲突发生；

……

实际上，上述这些问题都与 Verilog 的仿真原理密切相关。掌握 Verilog 的语义（Semantics）和仿真原理，是充分掌握这门语言的关键。

本章主要内容如下：

- 从一个问题说起；
- 电路与仿真；
- 仿真原理；
- 分层事件队列与仿真参考模型；
- 时序模型与延时；
- 再谈阻塞与非阻塞赋值；
- 如何提高代码的仿真效率；
- 如何防止仿真与综合结果不一致的现象发生。

8.1 从一个问题说起

如果对如下代码进行 RTL 级仿真：

```
always @ (A_in or B_in or C_in)
begin
   Temp = A_in & B_in; //阻塞赋值
   D_out = Temp | C_in; //阻塞赋值
end
```

可以得到如图 8-1 所示的电路行为。

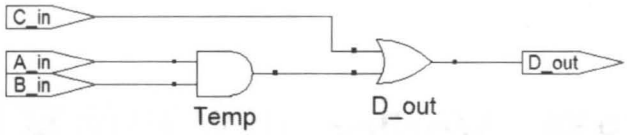

图8-1　电路行为

如果将其中两句赋值语句的顺序颠倒一下：

```
always @ (A_in or B_in or C_in)
begin
   D_out = Temp | C_in; //阻塞赋值
   Temp = A_in & B_in; //阻塞赋值
end
```

在仿真的时候将会发现，当 A_in 和 B_in 信号发生变化时，D_out 根本不会发生任何变化。

仅仅是语句顺序发生了变化，为什么功能会有如此大的不同呢？

如果再对代码进行修改，仅在敏感表中加入 Temp 变量：

```
always @ (A_in or B_in or C_in or Temp)
begin
   D_out = Temp | C_in; //阻塞赋值
   Temp = A_in & B_in; //阻塞赋值
end
```

那么仿真时的电路行为就与图 8-1 中所示的电路行为完全一致了。这又是为什么呢？

在本章中我们将一起来分析 Verilog 语言的仿真原理，希望读者能从中找到一些门道。

需要注意的是，以上所说的仿真行为是在 HDL 仿真器中进行 RTL 仿真的结果，而不是对综合之后的电路进行门级仿真。

8.2 电路与仿真

本节主要介绍 Verilog 语言和电路特性的关系，以及如何使用 Verilog 仿真硬件的行为。

8.2.1 电路是并行的

做过硬件设计的人都知道，与软件程序不同的是，硬件的运行是并行的，也就是说硬件系统中的模块是相互独立、并行运行的。在独立运行的同时，各个硬件模块之间也通过输入和输出接口进行交互，实现相互间的控制和通信。

如图 8-2 所示，在一个 PCB 板上，每个芯片都是独立并行工作的，同时它们又通过 I/O 相互通信，共同完成需要的功能。

同样，芯片内部的模块之间也是这种关系，只不过一个是在芯片内部，一个是在单板上。

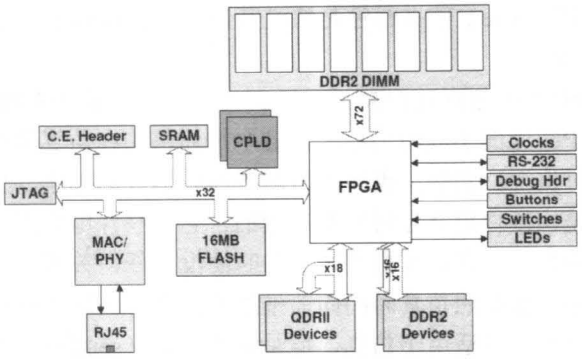

图8-2　并行 PCB 电路

8.2.2　Verilog 是并行语言

本书前面介绍过 Verilog 语言的基本语法结构，包括 RTL 级电路设计的基本方法和技巧。

Verilog 语言作为硬件描述语言，其本身是用来描述并行硬件结构的，因此它的一个重要特性就是并行性。

图 8-3 所示是本书第 2 章中用来说明 Verilog 的一个电路，其中用以描述 MUX2、DECODE2、XOR_A、XOR_B、DFF_A 和 DFF_B 的 Verilog 语言之间都是并行执行的。这些语言结构包括 always 语句、连续赋值语句（assign）、门实例（如 XOR），以及仿真中常用的 initial 语句等。

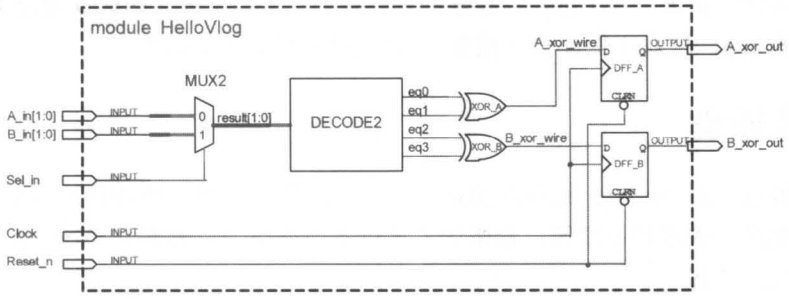

图8-3　并行描述的 Verilog 语言

这些并行语言结构之间是通过相应的信号相互影响和通信的，这些信号变量可以是 wire 或 reg 等。正是这些相互影响又相互联系的功能模块组成了一个个的硬件数字系统。

8.2.3　Verilog 仿真语义

众所周知，在软件世界中，程序是串行执行的，即便在多任务操作系统中也是如此。比如在我们使用的 PC 中，CPU 在同一时刻也只能执行一个任务，多个任务是分时执行的。

因此 EDA 工具厂商所设计的仿真器也必然是串行，或称之为顺序执行的。

那么顺序执行的仿真器怎么仿真并行的 Verilog 语言呢？

实际上，Verilog 语言的发明者们已经考虑到这个问题了。Verilog 本身是一个描述硬件的并行语言，而仿真的时候，它也是按照一定的顺序一条一条语句执行的，从而完成电路的

仿真，因此在仿真器中 Verilog 的并行性是通过其语义（Semantics）仿真出来的。所谓语义是指 Verilog 的语言含义。

在这一点上，Verilog 的仿真过程类似于 PC 中的多任务操作系统。Verilog 中的不同硬件电路描述语句对应 PC 上的不同应用程序。各个描述语句在仿真器中分时执行，类似于应用程序在 CPU 上分时执行。当然，也不能将两者简单地等同起来，仿真器运行有自己的特点。比方说仿真时间是由硬件电路延时参数决定的，它是硬件电路实际工作的时间。需要注意的是，仿真时间与仿真软件在计算机上运行的时间没有任何关系。

所有仿真工作都是严格按照仿真时间向前推进的，在什么时间执行什么操作。

虽然 Verilog 语言不仅仅适用于仿真，但是它的语义（semantics）却是专门为仿真定义的，其他所有的东西都是根据这一基本定义抽象得到的。所以在用 Verilog HDL 建模时，必需充分考虑代码在仿真上的语义，保证得到准确、真实的仿真结果，这是设计可靠电路所迈出的第一步。

有相当多的设计师并不关心 Verilog 的仿真语义，认为只要写出来的代码可以综合实现，然后再利用综合以后的门级模块去仿真，确保电路功能正确就行了，这样的想法实际上是非常危险的。

如果设计出的 Verilog 源代码不符合 Verilog 的仿真语义，那么对源代码进行仿真，就可能会出现仿真歧义，也就是说代码仿真结果会与综合后实现的门级网表功能不一致，这样很可能会遗漏一些关键的 bug。有的设计者想通过门级仿真保证电路正确，大家都知道，门级仿真的效率是相当低的，而且想找出设计 bug 非常困难。实际上，越是到了设计的后期，想找出 bug 也就越困难。

因此，确保写出来的 Verilog 代码符合仿真语义，尽量在设计的前期通过 RTL 级代码仿真找出设计的 bug，这对于保证设计质量和设计进度来说非常关键。

8.3 仿真原理

本节将重点介绍 Verilog 语言在仿真时的执行过程，介绍一些仿真的关键元素，如仿真时间、事件驱动、进程和调度等。希望读者通过本节的学习，能够对 Verilog 的仿真过程有一个更为深入的了解。

8.3.1 Verilog 的仿真过程

为了将原理讲述清楚，减轻读者的负担，这里采用了一个非常简单的电路和激励来描述 Verilog 的仿真过程，设计电路如图 8-4 所示。

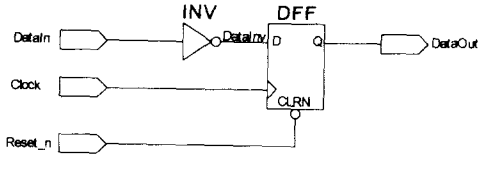

图8-4　设计电路

虽然后面的描述稍微显得有点琐碎，但是充分理解这个简单电路的仿真过程对深入掌握Verilog语言将会起到事半功倍的效果，希望读者认真体会。

设计代码INV_DFF和仿真激励TB如下。

【例 8-1】 INV_DFF 和 TB 的设计实例，代码参见随书光盘中“Example-8-1\sim”目录下的相关内容。

```
//以下是module名称和端口列表
`timescale 1ns/100ps
module TB ;
reg Ck, Rst_n, Din;
wire Dout;

//Clock generation
initial
begin
   Ck = 0;
   forever
     #10 Ck =  ~Ck; //don't add delay on RHS of blocking assignments
end

//Reset generation
initial
begin
   Rst_n = 1;
   #5  Rst_n = 0;
   #55 Rst_n = 1;
end

//Data Input Generation
initial
begin
   Din = 0;
   #80 Din = 1;
   #40 Din = 0;
end

//实例化被验证的电路
INV_DFF u_INV_DFF (
   .Clock   (Ck),
   .Reset_n (Rst_n),
```

```
    .DataIn  (Din),
    .DataOut (Dout)
);
endmodule

module INV_DFF (Clock, Reset_n, DataIn, DataOut);
input Clock;
input Reset_n;
input DataIn;
output DataOut;
reg DataOut;
wire DataInv;
always @(posedge Clock or negedge Reset_n)
begin
    if (~Reset_n)
        DataOut <= 1'b0;
    else
        DataOut <= DataInv;
end
assign #3 DataInv = ~ DataIn ;//更新事件在 3ns 以后执行
endmodule
```

本例的仿真时序和波形如图 8-5 所示。

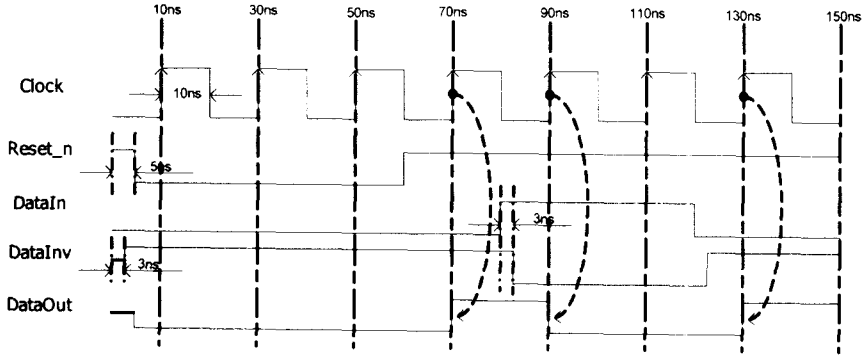

图8-5　仿真时序和波形

仿真器在 0 时刻开始工作，下面按照时间轴顺序进行描述。

1. 0 仿真时刻

- 在仿真顶层 TB 中，4 个进程同时在 0 时刻执行，包括 3 个 initial 进程和一个 INV_DFF 实例 u_INV_DFF。
- 同时执行的进程其顺序是不固定的，跟其在代码中出现的顺序无关，但是有些仿真器将它们出现在代码中的顺序作为执行的顺序。

- 首先执行 Clock Generation 中的语句 Ck=0，然后执行 forever #10 语句，当仿真器遇到#10 时将该进程挂起，转而执行其他进程，这是因为后面的语句是 10ns 仿真时刻所需执行的语句，而目前的仿真时刻还是 0，没有向前前进。
- 然后执行 Reset generation 中的语句 Rst_n = 1，当仿真器遇到#5 时将该进程挂起。
- 再执行 Data Input Generation 中的语句 Din=0，当仿真器遇到#80 时将该进程挂起。
- 在执行实例 n_INV_DFF 时，虽然 always 语句也是在 0 时刻开始执行的，但它只对 Clock 的上升沿和 Reset_n 的下降沿敏感，而在这时（0 时刻），并没有这些事件发生，所以也就没有执行该 always 语句。
- assign #3 DataInv = ~ DataIn 语句需要分成两部分进行分析，即计算事件和更新事件。在 0 时刻首先计算 DataInv=~DataIn，但是真正更新 DataInv 变量的事件却被调到 3ns 以后处理。这就是为什么在仿真波形上看 DataInv，3ns 以前是“X”，3ns 以后才更新成 1 的原因。
- 至此，0 仿真时刻的语句全部执行完成，仿真时间轴将向前推进。由于在仿真时间 3ns 以前已经没有需要执行的任务了，因此仿真器会将时间轴推进到 3ns 时刻。

2. 3ns 仿真时刻

- 在 3ns 仿真时刻只有一个任务需要执行，就是将 DataInv 更新为 1。3ns 仿真时刻的语句全部执行完成后仿真时间轴将向前推进。由于在仿真时间 3ns 以前已经没有需要执行的任务了，因此仿真器会将时间轴推进到 5ns 时刻。

3. 5ns 仿真时刻

- 在 5ns 仿真时刻，在 0 时刻被挂起的 Reset Generation 进程将被重新唤醒，开始执行“Rst_n = 0;”语句。当遇到#55 后，该进程将再次被挂起，等待“5+55=60ns”时刻，该进程将再次被唤醒。
- 这时，由于 Rst_n 被置位于 0，这一 Rst_n 的下降沿将触发 u_INV_DFF 中的 always 语句。执行“if (~Reset_n) DataOut <= 1'b0;”语句，这样，DataOut 变量将从初始的“x”状态被复位为 0。always 语句的进程将被挂起。
- 至此，5ns 仿真时刻的语句执行完成，仿真时间轴将向前推进。由于在仿真时间 3ns 以前已经没有需要执行的任务了，因此仿真器会将时间轴推进到 10ns 时刻。

4. 10ns 仿真时刻

- 在 10ns 仿真时刻，在 0 时刻被挂起的 Clock Generation 进程将被重新唤醒，执行“Ck = ~Ck;”语句。由于原来的 Ck 是 0，因此这里 Ck 出现了一个上升沿。这时，该进程的执行过程还没有完成。由于 forever 是一个永远循环的语句，因此仿真器继续返回到 forever #10 语句，直到遇到#10 以后，该进程才被挂起，执行其他 10ns 时刻的进程语句。
- Ck 的上升沿必然触发 u_INV_DFF 中的 always 语句，但是这时的 Reset_n 还是 0，因此依然执行“if (~Reset_n) DataOut <= 1'b0;”语句，DataOut 的值并没有发生变化。这实际上模拟了 D 触发器的特性：当触发器处于复位状态时，无论是否有时钟沿出现，输出都不会改变。

- 在这以后的时刻，如 30ns、50ns 等，虽然会有时钟沿连续出现，但 DataOut 仍然为 1。仿真时间推进到 60ns 时刻后，将唤醒 “Reset generation” 进程，Rst_n 被重新置位为 1。撤销对 D 触发器的复位，同时该 initial 进程执行完成，并将永远被挂起。

5. 70ns 仿真时刻

- 这时，前面带有 “//Clock Generation” 注释的 initial 语句的执行将再次产生一个 Clock 的上升沿。这一事件将触发 u_INV_DFF 中的 always 语句。由于这时候的 Reset_n 是 1，因此执行语句 “DataOut <= DataInv”。这样 DataOut 的值在 70ns 时刻将更新为 1。

6. 80ns 仿真时刻

- 在 80ns 仿真时刻，“Data Input Generation” 进程将被唤醒，执行 “Din = 1;” 语句。这一事件会继续触发 u_INV_DFF 进程中 “assign #3 DataInv = ~ DataIn” 语句的执行。首先计算 DataInv 的新值 0，然后将 DataInv 的更新事件调度到 3ns 以后的 83ns 时刻执行。于是在 83ns 仿真时刻，DataInv 将被更新为 0。

7. 90ns 仿真时刻

- 在 90ns 仿真时刻将再次出现 Ck 的上升沿。这一事件将触发 u_INV_DFF 中的 always 进程语句，执行语句 “DataOut <= DataInv”。这样 DataOut 的值在 90ns 时刻将更新为 0。

8. 120ns 仿真时刻

- 在 120ns 仿真时刻，“Data Input Generation” 进程将被唤醒，执行 “Din = 0;” 语句。同时，该 initial 进程执行完成，将永远被挂起！这一事件继续触发 u_INV_DFF 进程中 “assign #3 DataInv = ~ DataIn” 语句的执行。首先计算 DataInv 的新值 1，然后将 DataInv 的更新事件调度到 3ns 以后的 123ns 时刻执行。于是在 123ns 仿真时刻，DataInv 将被更新为 1。

9. 130ns 仿真时刻

- 在 130ns 仿真时刻将再次出现 Ck 的上升沿。这一事件将触发 u_INV_DFF 中的 always 进程语句，执行语句 “DataOut <= DataInv”。这样 DataOut 的值在 130ns 时刻将更新为 1。

通过以上描述，相信读者已经对 Verilog 语言的仿真过程有了一个感性的认识。下面将通过对一些重要概念的介绍，深入剖析 Verilog 仿真原理，这正是 Verilog 语言的精髓所在。

8.3.2 仿真时间

“什么时间做什么事” 是 Verillog 语言仿真的原则。

读者可能已经注意到，在前一节描述仿真过程时是按照时间轴顺序进行介绍的，我们称这个时间为仿真时间，Verilog 仿真将按照仿真时间的时间轴严格向前推进。

仿真时间是指由仿真器维护的时间值，用来对仿真电路所用的真实时间进行建模。比如仿真器处在 0 时刻，刚刚开始仿真，此时刻就被称为当前仿真时间，而以后的任何时刻都被

称为将来仿真时间。同样，当仿真时间推进到某一个时间点时，该时间点就被称为当前仿真时间，而以后的任何时刻都被称为将来仿真时间。

如果在当前的仿真时刻有多个事件需要执行，那么首先需要根据它们之间的优先级（在事件队列中的优先级）来判定谁先谁后。如果优先级相同，则执行顺序随机，不同的仿真器的行为有可能有所不同。比如在例 8-1 中，其中 4 个进程同时并行执行，而有些仿真器如 ModelSim，将按照它们在代码中出现的顺序执行。这种情况，如果处理不当，或不理解这一特性，很容易出问题。

8.3.3 事件驱动

功能仿真是一种事件驱动的仿真。Verilog 语言用来对数字系统的功能和时序进行建模，这些模型的仿真过程是围绕事件来组织的。

事件是指在特定时刻，模型中数值的变化。Verilog 语言的语义规定了一个事件导致其他事件及时发生的方式。

在被仿真的电路中，线网或寄存器值的任何改变被认为是一个更新事件（Update event）。

进程（Process）对更新事件敏感。当一个更新事件被执行后，所有对该事件敏感的进程都将以随机顺序计算（evaluated）。进程（门或行为模型）的计算也是一个事件，叫做计算事件（Evaluation event）。实际上，更新事件会导致计算事件发生，计算事件也会导致更新事件发生。

在例 8-1 中，在 70ns 仿真时刻，Ck 的上升沿事件将导致语句“DataOut <= DataInv”中左式计算事件的发生，以及右式 DataOut 更新事件的发生。

计算事件和更新事件之间循环往复的互相触发，推动了仿真时间的前进。按照这些事件的顺序，仿真得以执行，仿真时间（Simulation time）得以向前推进（Advance）。

在本章 8.4 节中将会用到一个 Verilog 仿真参考模型，所有兼容的仿真器都应遵守该模型中的规则，但是不同的仿真器在执行的某些细节上可能有很大的差别。另外，不同厂商可以自由地采用不同的算法，Verilog 规范并没有规定。

8.3.4 进程

进程是 Verilog 语言中的独立执行单元，用 Verilog 描述的数字系统正是由一个一个进程组成的。

进程包括原语（Primitives）、模块（Modules）、Initial 和 always 过程块、连续赋值语句、异步任务以及过程赋值语句等。

比如在例 8-1 中，顶层模块 TB 中有 4 个进程，包括 3 个 initial 语句和一个 INV_DFF 的实例 u_INV_DFF，而 u_INV_DFF 中又包含两个子进程，即 always 语句块和 assign 连续赋值语句。

进程可以被激活（Activate）或者被挂起（Suspend）。仿真器总是在处理被激活的一个进程，而其他所有的进程则处于挂起状态。

在仿真的时候，所有的进程都是并行执行的，它们之间的顺序随机，仿真工具可以根据

其自身的原则安排它们之间的先后执行关系。比如在例 8-1 中，所有的进程都是按照它们在源代码中出现的顺序来先后执行的。但是，这并不违背 Verilog 语言的并发性，因为在执行完所有的进程之前，仿真时间是不会向前推进的，仍然是 0 时刻。

执行一个进程时，如果遇到一个事件语句（“@”）、延时语句（“#”）或其表达式值为 FALSE 的等待语句（wait），则进程将被挂起，直到发生该事件、已经过延时中的时间单位数或等待语句表达式变为真时，该进程才会重新被激活。

从例 8-1 中可以明显地看出这样一个现象，即当执行 initial 进程的过程中遇到“#10”时，该进程将被挂起，转而执行其他进程，直到延时 10ns 的仿真时刻到达后，该 initial 进程才会重新被执行。

读者也许会发现，在 Verilog 语言中，进程之间是并行独立执行的，但它们在执行的过程中又交织在一起。一个进程被挂起，另一个进程开始执行；该进程被挂起，另一个进程又开始执行，这就是 Verilog 语言在仿真时的一大特点。这样的执行过程类似于多任务操作系统，操作系统也是在多个任务中间分配执行的时间的。在操作系统中有进程锁死的现象，而在 Verilog 仿真时也存在类似的问题。

比如将例 8-1 中的 Clock Generation 进程简单修改一下，将#10 去掉，代码如下：

```
//Clock generation
initial
begin
   Ck = 0;
   forever
     Ck =  ~Ck;
end
```

这样，在执行仿真的时候，读者将发现仿真器的仿真时间将一直停留在 0 时刻，不管让仿真器运行多久，仿真时间永远不会向前推进，这是由于该 initial 的进程已经被锁死。它永远在执行 Ck=~Ck，因为没有延时语句，所以该 Ck=~Ck 语句将在 0 时刻一直将 Ck 取反。该进程永远不会被挂起，其他进程也就无法执行，仿真时间就无法向前推进。

8.3.5 调度

在 Verilog 的仿真过程中，另一个重要的概念就是调度。

调度实际上就是安排事件执行的顺序。

对于 assign #3 DataInv = ~ DataIn 语句在 0 时刻的执行，首先需要执行一个计算事件，即 DataInv=~DataIn，然后把 DataInv 的新值安排到 3ns 以后再更新，这就是事件的调度。

在 Verilog 中，同一个仿真时刻也许有很多的事件需要执行，不同的事件类型之间有一个先后的优先级关系，而事件的执行也将产生新的当前时刻事件和将来时刻事件，这些事件需要被合理地调度和管理。

有关调度方面的知识，将在 8.4 节的内容中介绍。

8.3.6 时序控制（Timing Control）

8.3.4 小节中讲解进程的时候曾经提到过，在 Verilog 仿真时，主要通过以下 3 种方法来进行时序控制，即事件语句（“@”）、延时语句（“#”）和等待语句。

执行一个进程时如果遇到一个事件语句（“@”）、延时语句（“#”）或其表达式值为 FALSE 的等待语句（wait），那么该进程将被挂起，直到发生该事件、已经过延时中的时间单位数或等待语句表达式变为真时，该进程才会重新被激活。时序控制总是伴随着进程的激活和挂起。

8.3.7 进程、事件和仿真时间的关系

事件会在不同的时间发生。为了跟踪事件，使其以正确的顺序处理，事件将被保存在事件队列（Event queue）中，由仿真时间来排序。特定时刻的所有事件都保存在一起，仿真器调度器（Simulator Scheduler）将跟踪新事件的发生并维护事件列表。调度器可以通过将事件插入到事件列表中来把事件调度到将来的某个时刻，也可以把事件从事件列表中删除，从而达到取消调度的目的。

把一个事件放入事件队列被称为调度一个事件（Scheduling an event），而一个事件弹出事件队列则被称为执行一个事件（Execution of an event）。事件的弹出将会激活相应的进程。

当执行一个事件语句（“@”）、延时语句（“#”）或其表达式值为 FALSE 的等待语句时，进程（比如 initial 或 always 语句）将被挂起（suspended），直到发生该事件、已经过延时中的时间单位数或等待语句表达式变为真时，才重新执行 initial 或 always 语句块。

Verilog 是一种并行语言，允许描述多个同时发生的动作。但执行这些操作时要求将它们序列化，因为计算机不同于所建模的硬件，它并不是并行的。

8.3.8 Verilog 语言的不确定性

在 Verilog 仿真时，有两个不确定的行为来源，一个是在零时刻时的任意执行顺序，一个是进程之间语句的随意交叉。

仿真器执行被调度到同一时刻的一组事件时也许需要几个仿真周期来完成，因为一个事件可能产生本时刻的其他事件。这里说的执行同一时刻的事件是指在长度为零的时间内执行它们，这并不是说执行这些事件不需要时间，而是说所有事件的发生并不会促使仿真时间向前推进（Do not advance simulation time）。这种零长度时间内事件的任意执行顺序是仿真语言不确定性的根源。

Verilog 语言的第二个不确定因素是不同行为进程语句的潜在交织。Verilog 语言规范并没有规定进程交织的顺序，因此各仿真器厂家的仿真器其实现方法都有可能不同。

8.4 分层事件队列与仿真参考模型

8.4.1 分层事件队列

为了更好地理解仿真器的事件调度，下面将 Verilog 事件队列分为 5 个不同的区域，如图 8-6 所示。需要注意的是，事件是按照一定的规则被加入到 5 个区域中任意一个区域的，但是只从其中的“活跃事件”区域出队，出队之后该事件将会立刻执行。

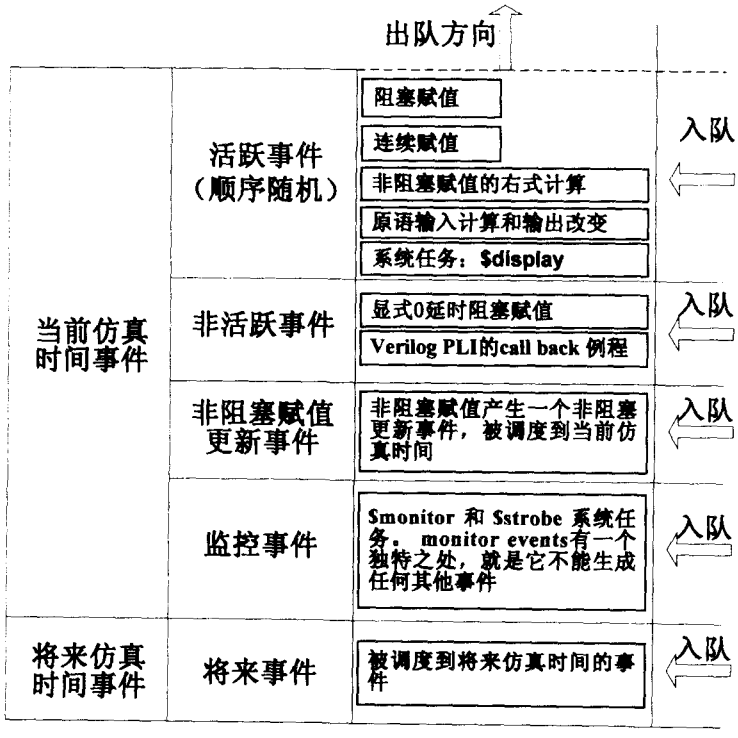

图8-6　分层事件队列

仿真器首先按照仿真时间对事件进行排序，然后再在当前仿真时间里按照事件的优先级顺序进行排序。

活跃事件是优先级最高的事件。在活跃事件之间，它们的执行顺序是随机的。阻塞赋值（=）、连续赋值（assign）以及非阻塞赋值的右式计算等都属于活跃事件。

将来仿真时间里的所有事件都将暂时存放在将来事件队列中。当仿真器前进到某个时刻后，该时刻的所有事件也会被分类到当前仿真时间事件队列中，而仿真时刻未到的事件则仍然留在将来事件队列中。

8.4.2 仿真参考模型

IEEE Std 1364–1995 国际标准中定义了 Verilog 仿真的参考模型。在下面的代码中，“T”代表当前仿真时间，所有事件都保存在事件队列中，由仿真时间排序。

仿真参考模型的代码如下：

```
while ( 队列中有事件需要被处理 ) {
    //以下是激活事件的过程
if ( 没有活跃事件 )
{
        if ( 有非活跃事件 )
{激活所有的非活跃事件 ; }
        else if ( 有非阻塞赋值的更新事件 )
            {激活所有非阻塞赋值的更新事件; }
        else if (有 monitor 事件 )
            {激活所有 monitor 事件}
        else
{将 T 推进到下一个事件时间 ;
 激活时间 T 上的所有非活跃事件 ;}
        }
    //以下是执行事件的过程
E = 任何活跃事件 ;
    if ( E 是一个更新事件 )
{更新需修改的对象 ;
         将产生的计算事件增加到事件队列中;}
else
{ /* 应该是一个计算事件 */
        计算这个进程 ;
        将计算产生的更新事件加入到事件队列中 ;}
}//end while
```

对当前仿真时间里所有 Active events 的处理都被称为仿真周期（Simulation cycle）。如果当前 Active events 输出的更新事件也是当前时刻的事件，则这个事件将被插入到事件列表中，在下一个仿真周期进行处理，这样仿真器执行被调度到同一时刻的一组事件也许需要几个仿真周期来完成。

8.5 时序模型与延时

本节重点介绍 Verilog 中的时序模型和正确的延时设计方法。

8.5.1 仿真模型（Simulation Model）

在 Verilog 语言中，我们将数字硬件的模型称为仿真模型，比如 module 块和其中的 always 语句块都可以被称作仿真模型。

8.5.2　时序模型（Timing Model）

时序模型是仿真器的时间推进模型，它反映了推进仿真时间和调度事件的方式。

时序模型分为门级时序模型和过程时序模型两种。

在解释两种时序模型之前，先了解一下敏感表和扇出表的区别。敏感表是仿真模型的输入表，它由接收新值的元素组成，它告诉我们当输入发生变化时，哪些输入变化会导致仿真模型的执行。扇出表由将产生新值的元素组成，它告诉我们当一个事件发生时需要计算哪些元素。

一、门级时序模型

Verilog 门级时序模型主要适用于分析所有的连续赋值语句、过程连续赋值语句、门原语和用户自定义原语等。该模型的特点是，任意时刻、任意输入发生变化，门示例都将重新计算其输出，如果输出有变化，那么以后很可能会产生一个新的事件。所有的仿真模型都对输入变化敏感，而这种变化又将导致仿真模型的执行。

门级时序模型的另一个特征是关于新事件的调度。假设存在一个门级时序模型，同时该模型产生的一个事件已被调度但尚未执行，如果事件的结果将导致一个新事件产生，那么仿真调度器就会撤销对先前事件的调度，转而调度新事件。所以 Verilog 的门级模型具有惯性延时的特性，它刚好比传输延时要长。

门级时序模型非常精确地模拟了电路中的惯性延时。

二、过程时序模型

Verilog 过程时序模型并不像门级时序模型那样，它不是对任何时刻的任何变化都敏感，它的敏感性依赖于控制的上下文（Context）。一般来说，initial 和 always 语句只对输入的一个子集敏感，这种敏感性是随着仿真执行的时间而改变的。因此，这些敏感元素是根据行为模型的当前执行部分来确定的。比如，always 语句对@符号后面括号中的变量敏感。

过程时序模型的另一个特征是关于事件调度方式的。假设一个寄存器的一个更新事件已经被调度，如果再调度同一个寄存器的另一个更新事件，即使在同一时刻，前一个事件也不会被取消。因此，一个实体（如寄存器）的事件列表中可能有多个事件，如果同时有几个更新事件，那么它们的执行顺序是不确定的。注意，这与门级时序模型不同，在门级时序模型中，针对同一个实体，其输出的新事件将取代先前已调度但尚未执行的事件。

模型仿真时可以有交织，这种交织可以通过使用两种 Verilog 时序模型来建立。事实上，根据输入敏感元（敏感表中的变量），过程时序模型可以建立门级时序模型所能建立的模型。一般来说，这两种时序模型定义了语言中的两大类元件，门级时序模型主要用于对组合逻辑建模，而过程时序模型主要用于对时序逻辑建模。

8.5.3　案例分析

下面通过一个实例，进一步说明时序模型的概念。

笔者在仿真一个逻辑的过程中，为了模拟 E1 信号在长距离传输过程中的延时，在 Testbench 的顶层链接中使用了如下语法：

```
assign #8000000 Ck_delay = Ck ;
```

时间单位是 1ns，使用该语法的原意是将 Ck 信号延时 8ms 赋给 Ck_delay。Ck 信号是周期为 488ns 的时钟信号，远远小于所需的延时值 8ms。

本来想借助 assign 语句模拟信号的传导延时，然而出乎意料的是，在仿真结果波形中，Ck_delay 并不是 Ck 信号延时 8ms 后的信号，而是一直保持未初始化的状态，而且只有延时值小于 244ns 时，比如 assign #240 Ck_delay = Ck，Ck_delay 才会正常输出。

在该语法中，无论何时 Ck 信号发生变化，都会立即产生并执行一个计算事件，计算等号右边的表达式，同时产生一个更新事件（把计算值赋给 Ck_delay），但该更新事件并没有立即执行，而是被调度到当前仿真时间以后的 8ms 时刻才去执行。

当然，根据门级时序模型的特征，在上一更新事件执行以前还会有新的更新事件产生，因为 Ck 是一个周期为 488ns 的时钟信号，它每隔 244ns 变化一次。这样，在每个 244ns 以后，Verilog 调度器就会撤销先前已调度的事件，而调度新的事件。所以更新事件不断产生，又不断被撤销，最后导致没有一条更新事件被真正执行，于是出现了 Ck_delay 信号一直处于未初始化状态的现象。

而当延时小于 244ns 时，如 assign #240 Ck_delay = Ck，Ck_delay 的更新事件将被调度到 240ns 以后执行，而 Ck 每隔 244ns 才翻转一次，因此在新的更新事件执行以前，上一次被调度的事件已经被执行，这样就可以正确输出需要的延时了。

后来我们又试了下面几种语法，如：

```
always @( Ck ) begin
      $display("%d",$time);
     #8000000 Ck_delay <= Ck ;
end
```

和

```
always @( Ck ) begin
     $display("%d",$time);
     #8000000  Ck_delay =  Ck ;
end
```

以及

```
always @( Ck ) begin
     $display("%d",$time);
    Ck_delay = #8000000  Ck ;
end
```

这些方法虽然都使用的是过程时序模型，不存在惯性延时的问题，但是它们都需要等待延时完成之后，才会对 Ck 信号的新变化敏感（可以通过以上描述中的$display 系统任务来观察 always 块的触发时间），这也不是我们想要得到的结果。

最后我们发现以下描述可以很好地满足我们的需要：

```
always @( Ck ) begin
     $display("%d",$time);
     Ck_delay <= #8000000 Ck ;
```

```
end
```

在产生一个 8ms 以后的更新事件后，本次触发执行完成，可以由 Ck 变化再次引起触发，其间进程并没有被挂起。前面已经介绍过，过程时序模型的特点是，在寄存器中的一个更新事件已经被调度的情况下，如果再调度同一个寄存器中的另一个更新事件，即使在同一时刻，前一个事件也不会被取消，因此这个语句非常好地模拟了电路中的传导延时，这正是我们需要的电路行为。

8.5.4　在 Verilog 语言中增加延时

延时在 Verilog 语言中经常被错误地使用，希望通过下面的介绍，读者能够理解延时的真正含义。

一、电路中的两种延时

在电路中存在两种延时，即惯性延时（Inertial Delay）和传导延时（Transport delay）。

例如在一个与非门电路中，门延时为 5ns，那么任何小于这个延时值的输入变化都不会对输出造成影响，如图 8-7 所示。其中，A 信号输入的 4ns 的高脉冲并没有在 B 端体现出来，而 13ns 的脉冲经过 5ns 门延时以后在 B 端输出反相信号。这种由于输入变化过快，小于门本身的延时值，而导致输出无响应的特性被称为电路的惯性延时，也就是说电路具有一定的惯性，或者说是惰性。

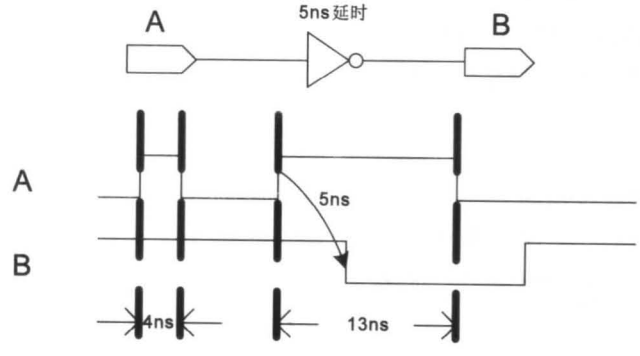

图8-7　惯性延时

本章 8.5.2 小节中曾经介绍过 Verilog 中的门级时序模型，包括连续赋值语句、过程连续赋值语句、门原语和用户自定义原语等。一般来说，Verilog 的门级时序模型是具有纯惯性延时的模型。

Verilog 中也有一些命令行开关用于改变门级模型仿真时的行为。例如可以将门级时序模型改变为具有传导延时的模型，或者是当输入变化小于门延时的时候，输出不确定值（x）。这里不再过多地讨论这个问题，感兴趣的读者可以参考其他文献资料。

接着再来看看另外一种延时情况。信号 A 经过延时 5ns 的传输线到达另外一端 B，信号 A 的任何变化都将在 5ns 以后体现在 B 端，如图 8-8 所示，这种延时通常被称为传导延时。

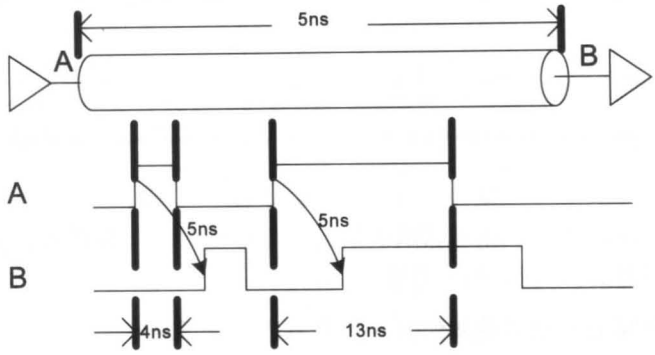

图8-8　传导延时

二、在阻塞赋值语句中增加延时（不推荐）

不少工程师习惯在阻塞赋值语句的左式（LHS）或者右式（RHS）中增加延时，例如：

```
always @( a or b ) begin //语句A，不推荐
    #5 sum = a + b ;
end
```

和

```
always @( a or b ) begin //语句B，不推荐
    sum = #5 a + b ;
end
```

下面分析一下该描述方法，由于 sum=a+b 是阻塞赋值，因此计算和更新这两个事件是原子操作。对于语句 A，在某个 T 时刻时 a 发生了变化，导致 always 语句开始执行，当遇到#5 后，该 always 进程将立刻被挂起。等待 5ns 以后，再将 T+5ns 时刻的 a 和 b 相加，并将结果赋予 sum。这样，在 T 到 T+5ns 的时间之内，a 和 b 上的所有变化就都被忽略了。

如图 8-9 所示，信号 a 在 5ns 时变成 16 进制的 4'ha，致使 always 语句被激活，但是直到 10ns 以后，sum 的值才变为当前时刻的 a+b 的值，也就是 4'hf+4'h1=5'h10，而 10ns 以前的任何变化却都被忽略了。代码参见随书光盘中“Example-8-2\Blocking_LHS_Delay”目录下的相关内容。

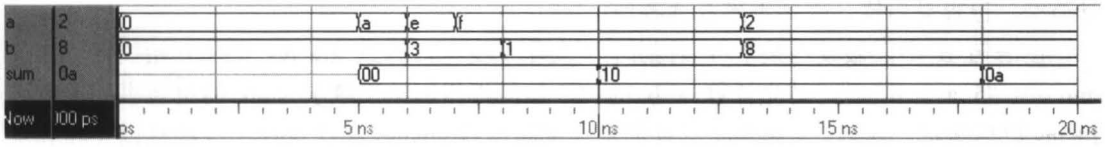

图8-9　阻塞赋值左式延时仿真波形（错误）

对于语句 B，在某个 T 时刻时 a 发生了变化，导致 always 语句开始执行，先将 T 时刻变化后的 a 和 b 相加，然后将该 always 进程挂起，等待 5ns 以后，再将 T 时刻的 a+b 的结果赋予 sum，也就是说在 T 到 T+5ns 的时间之内，a 和 b 上的所有变化也都被忽略了。

如图 8-10 所示，信号 a 在 5ns 时变成 16 进制的 4'ha，致使 always 语句被激活，但是直到 10ns 以后，sum 的值才变为 5ns 时刻的 a+b 的值，也就是 4'ha+0=5'h0a，而 10ns 以前的

任何变化却都被忽略了。代码参见随书光盘中“Example-8-2\Blocking_RHS_Delay”目录下的相关内容。

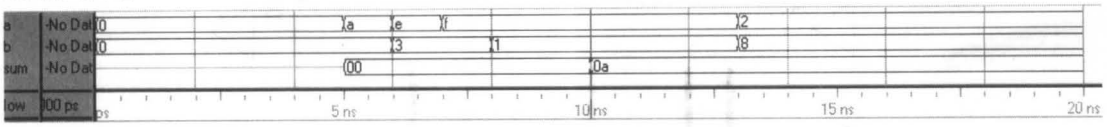

图8-10　阻塞赋值右式延时仿真波形（错误）

通过以上分析可以看出，在阻塞赋值语句中增加延时，不能模拟任何电路中的延时类型（惯性延时和传导延时），因此不推荐读者使用。

三、在非阻塞赋值中语句增加延时（只在右式中有意义）

不少工程师习惯在非阻塞赋值语句的左式（LHS）或者右式（RHS）中增加延时，例如：

```
always @( a or b ) begin //语句A，不推荐
    #5 sum <= a + b ;
end
```

和

```
always @( a or b ) begin //语句B，可用于传导延时建模
    sum <= #5 a + b ;
end
```

在语句 A 中，假设当某个 T 时刻时 a 发生了变化，导致 always 语句开始执行，当遇到 #5 后，该 always 进程将立刻被挂起。等待 5ns 以后，always 语句才重新被激活，将 T+5ns 时刻的 a 和 b 相加，并将结果赋予 sum。这样，在 T 到 T+5ns 的时间之内，a 和 b 上的任何变化都将被忽略，仿真波形如图 8-11 所示。代码参见随书光盘中“Example-8-2\NonBlocking_LHS_Delay”目录下的相关内容。

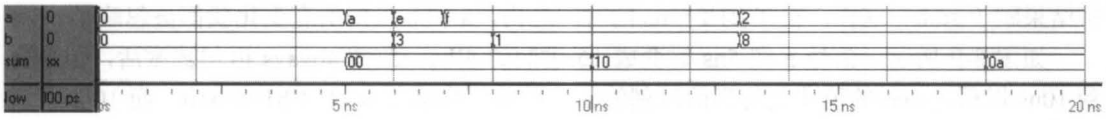

图8-11　非阻塞赋值左式延时仿真波形（错误）

通过以上分析可以看出，在非阻塞赋值语句的左边增加延时，不能模拟任何电路中的延时类型，也容易出错，因此不推荐读者使用。

在语句 B 中，假设当某个 T 时刻时 a 发生了变化，导致 always 语句开始执行，执行 sum <= #5 a + b; 时，首先计算 a+b 的值，然后将相加结果赋予 sum 的更新事件调度到 T+5ns 以后执行，此后，a 和 b 上的任何变化都将导致 always 再次执行，也就是说 a 和 b 上的任何变化都不会被忽略，而总是在 5ns 以后体现在 sum 上，仿真波形如图 8-12 所示。代码参见随书光盘中“Example-8-2\NonBlocking_RHS_Delay”目录下的相关内容。

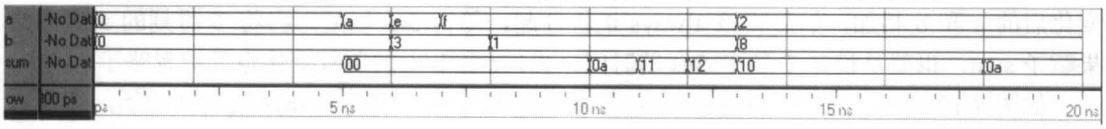

图8-12　非阻塞赋值右式延时仿真波形（正确）

因此，在非阻塞赋值语句的右式（RHS）中增加延时，可以精确地模拟电路中的传导延时，是一种有意义的、健康的代码风格。

四、在连续赋值语句中增加延时

在连续赋值语句中，只有一种延时语法是合法的，如下：

```
assign #5  B = ~ A ;
```

这种延时精确地模拟了电路中的惯性延时。A 信号上任何小于 5ns 的变化都将被过滤掉，而不会反映到 B 信号上。

8.6 再谈阻塞与非阻塞赋值

有了以上的分析，接下来再来深入理解阻塞和非阻塞赋值的本质。

8.6.1 本质

一、语义本质

首先看看阻塞赋值和非阻塞赋值在语义上的含义。

代码如下：

```
always @(posedge clk) begin
   q1 = d;
   q2 = q1;
   q3 = q2;
end
```

always 语句块对 clk 的上升沿敏感，当发生 clk 0～1 的跳变时，执行该 always 语句。

在 begin...end 语句块中所有语句是顺序执行的，而且最关键的是，阻塞赋值是在本语句中“右式计算”和“左式更新”完全完成之后，才开始执行下一条语句的。

在本例中， d 的值赋给 q1 以后，再执行 q2=q1；同样在 q2 的值更新以后，才执行 q3=q2。这样，最终的计算结果就是 q3=d。

所有的语句执行完以后，该 always 语句等待 clk 的上升沿，从而再一次触发 begin...end 语句。

接下来再来看看非阻塞赋值的情况。

非阻塞赋值，顾名思义，就是指当前语句的执行不会阻塞下一语句的执行。要理解这一点，首先得了解非阻塞赋值的语句结构。

```
{CO, SUM} <=   A+B;
```

将上式中的 A+B 称为右式（RHS）计算事件，{CO, SUM}称为左式（LHS）更新事件。当然，右式和左式也可能是单纯的变量，而不是表达式。

在分层事件队列一节中曾经提到过，非阻塞赋值的右式计算属于活跃事件，需要优先执行，因此首先执行 A+B，然后产生一个更新事件，把计算结果更新到{CO, SUM}中，将其放入事件队列中，但是并不马上执行，即使这个更新事件属于当前仿真时间的事件。因为非阻塞赋值的更新事件优先级较低，需要执行完其他当前仿真时间的活跃事件和非活跃事件之

后，才会执行当前时间的非阻塞赋值更新事件。

如果在非阻塞赋值的右式中有延时参数，例如：

```
{CO, SUM} <= #5  A+B; //time unit is ns.
```

那么 A+B 完成以后，会将{CO, SUM}的更新事件放入事件队列中，并在当前仿真时间的 5ns 以后才会执行该事件。当仿真时间向前推进 5ns 以后，只有执行完该仿真时刻上的所有活跃事件和非活跃事件，才会执行{CO, SUM}更新事件。

下面分析如下代码的执行过程，请读者仔细推敲其与上一段代码的不同之处。

```
always @(posedge clk) begin
   q1 <= d;
   q2 <= q1;
   q3 <= q2;
end
```

首先执行 q1<=d，产生一个更新事件，将 d 的当前值赋给 q1，但是这个赋值过程并没有立刻执行，而是在事件队列中处于等待状态。

然后执行 q2<=q1，同样产生一个更新事件，将 q1 的当前值（注意上一语句中将 d 值赋给 q1 的过程并没有完成）赋给 q2，这个赋值事件也将在事件队列中处于等待状态。

再执行 q3<=q2，产生一个更新事件，将 q2 的当前值赋给 q3，这个赋值事件也将在事件队列中等待执行。

这时 always 语句块执行完成，开始对下一个 clk 上升沿敏感。

那么什么时候才执行那 3 个在事件队列中等待的事件呢？通过前面章节的描述可知，只有当当前仿真时间内的所有活跃事件和非活跃事件执行完成之后，才开始执行这些非阻塞赋值的更新事件。这样就相当于将 d、q1 和 q2 的值同时赋给了 q1、q2 和 q3。

二、电路本质

接着再来看看非阻塞赋值在电路上的含义。

```
always @(posedge clk or negedge rst_n) begin
   if (~rst_n)
      A <= 0;
   else
      A <= #2 B; //time unit is ns.
end
```

以上描述非常精确地模拟了一个 D 触发器的行为，如图 8-13 所示。

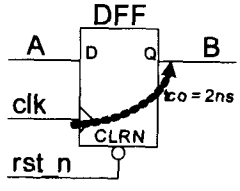

图8-13　D 触发器模型

A 的值在时钟上升沿处被采样，在 2ns 以后将 B 更新，也就是 DFF 的固有时钟到输出延时（tco）为 2ns。

与此类似，上一节中的描述：

```
always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
end
```

其实现结果如图 8-14 所示。

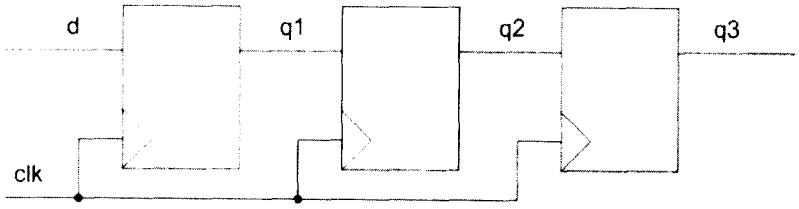

图8-14 三级流水的寄存器

所实现的电路与其中 3 条语句在 begin...end 中出现的顺序无关。

使用阻塞赋值的代码如下：

```
always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
end
```

实现的电路结果如图 8-15 所示。中间变量 q1 和 q2 被忽略，对它们的赋值没有任何意义。

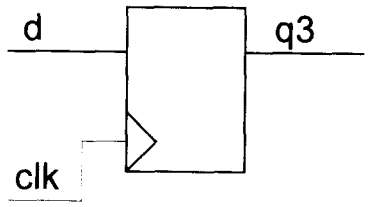

图8-15 一级寄存器

在一些参考文献中，为了使用户减少代码错误，会建议在描述纯组合逻辑时采用阻塞赋值，而描述时序逻辑行为时采用非阻塞赋值。这个粗略的规则有一定的好处，也为初学者减少了许多麻烦，但有时根据用户所要实现的电路的要求，也会在描述组合和时序混合的电路结构中采用阻塞赋值。

比如本书第 3 章 3.6.2 小节中的 CRC10 实例，为了实现一个 32 位数据移入的暂存，采用了阻塞赋值，将组合逻辑和寄存器在一个 always 语句中实现。

下面通过典型案例，进一步说明阻塞赋值和非阻塞赋值的区别。

8.6.2　案例分析

这里有一个数组：Data[0]、Data[1]、Data[2]和 Data[3]，它们都是 4 比特的数据。我们需要在它们当中找到一个最小的数据，同时将该数据的索引输出到 LidMin 中，这个算法有点类似于“冒泡排序”的过程，而且需要在一个时钟周期内完成。例如，如果这 4 个数据中 Data[2]最小，那么 LidMin 的值则为 2。

开始先写出如下代码：

```
reg [1:0] LidMin;
reg [3:0] Data [0:3];
always @( posedge clk or negedge rst_n )
begin
   if ( ~ rst_n )
         LidMin <= 0 ;
   else begin
        if ( Data[0] <= Data[LidMin] ) //"<="表示小于等于
         LidMin <= 0 ;  //"<="表示非阻塞赋值
        if ( Data[1] <= Data[LidMin] )
         LidMin <= 1 ;
        if ( Data[2] <= Data[LidMin] )
         LidMin <= 2 ;
        if ( Data[3] <= Data[LidMin] )
         LidMin <= 3 ;
   end
end
```

我们的原意是首先将 LidMin 设置为一个初始值（任意值都可以），然后将 Data[0]～Data[3]与 Data[LidMin]进行比较，每比较一个数，就将较小的索引暂存在 LidMin 中，然后再进行下一次比较。当 4 组数据比较完成之后，最小的数据索引就会保留在 LidMin 中。

我们在以上代码中使用了非阻塞赋值，结果发现，仿真波形根本不是我们所需要的功能，如图 8-16 所示。图中的 Data[0]～Data[3]分别为 11、3、10 和 12，LidMin 的初始值为 0。按道理来说，LidMin 的计算结果应该为 1，因为 Data[1]最小，但仿真波形却为 2。

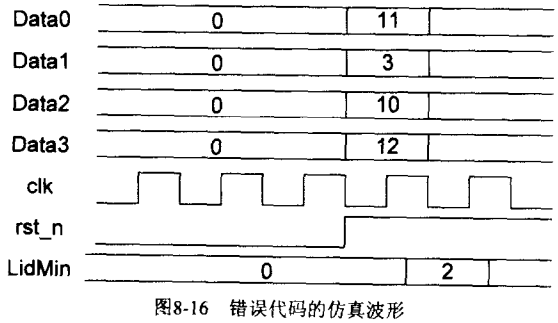

图8-16　错误代码的仿真波形

为什么会得出这样的结果呢？

在时钟上升沿到来以后，且 rst_n 信号无效时开始执行以下 4 个语句，假设这时候的 LidMin 是 0，Data[0]～Data[3]分别为 11、3、10 和 12：

```
if ( Data[0] <= Data[LidMin] ) //"<="表示小于等于
  LidMin <= 0 ;  //"<="表示非阻塞赋值
 if ( Data[1] <= Data[LidMin] )
  LidMin <= 1 ;
 if ( Data[2] <= Data[LidMin] )
  LidMin <= 2 ;
 if ( Data[3] <= Data[LidMin] )
  LidMin <= 3 ;
```

第一句的 if 为真，因此执行 LidMin <= 0，而这时候，LidMin 并没有立刻被赋值，而是调度到事件队列中等待执行，这是非阻塞赋值的特点。

第二句的 if 为真，因此执行 LidMin <= 1，这时 LidMin 也没有立刻被赋值为 1，而是调度到事件队列中等待执行。当前的 LidMin 还是 0，没有发生任何变化。

同样，第三句的 if 也为真，因此执行 LidMin <= 2，将更新事件调度到事件队列中等待执行。当前的 LidMin 还是 0。

而第四句的 if 为假，因此直接跳过 LidMin <= 3 不执行，这时跳出 always 语句，等待下一个时钟上升沿。

在以上的 always 语句执行完成以后，仿真时间没有前进。这时存在于事件队列中当前仿真时间上的 3 个被调度的非阻塞更新事件开始执行，它们分别将 LidMin 更新为 0、1 和 2。

按照 Verilog 语言的规范，这 3 个更新事件属于同一仿真时间内的事件，它们之间的执行顺序随机，这就产生了不确定性。一般的仿真器在实现的时候是根据它们被调度的先后顺序执行的，事件队列就像一个存放事件的 FIFO，它是分层事件队列的一部分，如图 8-17 所示。

图8-17 更新赋值队列

这 3 个事件在同一仿真时间被一一执行，而真正起作用的是最后一个更新事件，因此在仿真的时候得到的最终结果是 LidMin 为 2。

然而我们想要得到的结果是，在每个 if 语句判断并执行完成以后，LidMin 先暂存这个中间值，再进行下一次比较，也就是说在进行下一次比较之前，这个 LidMin 必须被更新，而这一点也正是阻塞赋值的特点，因此我们将代码作如下更改：

```
always @( posedge clk or negedge rst_n )
begin
   if ( ~ rst_n )
```

```
            LidMin = 0 ;
    else begin
          if ( Data[0] <= Data[LidMin]  ) //"<="表示小于等于
           LidMin = 0 ;  //"="表示阻塞赋值
          if ( Data[1] <= Data[LidMin]  )
           LidMin = 1 ;
          if ( Data[2] <= Data[LidMin]  )
           LidMin = 2 ;
          if ( Data[3] <= Data[LidMin]  )
           LidMin = 3 ;
    end
end
```

其仿真波形如图 8-18 所示。

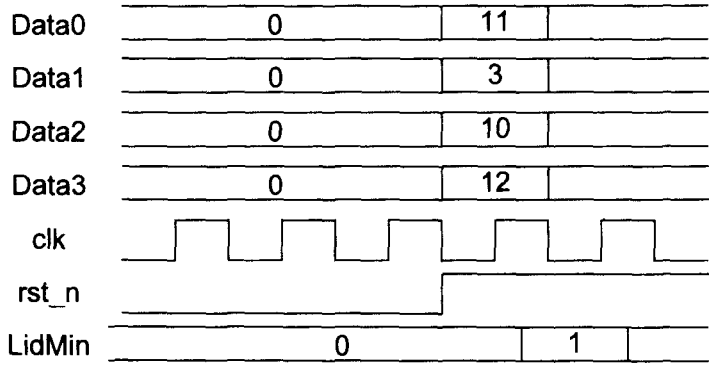

图8-18　正确代码的仿真波形

在代码仿真过程中，第二句的 if 为真，执行 LidMin = 1，根据阻塞赋值的特点，LidMin 被立刻赋值为 1。在执行第三句 if 的时候，if (Data[2] <= Data[LidMin])为假，直接跳过 LidMin=2 不执行，同样也跳过 LidMin=3 不执行。LidMin 被最终赋值为 1，这正是我们想要的结果。

另外，为了使代码看起来更简洁，我们使用 for 语句改写了代码：

```
always @( posedge clk or negedge rst_n )
begin : COM_PROC //定义语句块名称
    integer i ;//局部变量
    if ( ~ rst_n )
          LidMin = 0 ;
    else begin
       for ( i = 0 ; i <= 3 ; i = i + 1 ) begin
          if ( Data[i] <= Data[LidMin] )
             LidMin = i ;
       end
    end
```

```
end
```

这种写法与前面展开的写法完全等效，功能完全一致。如果读者今后在读代码时发现带有 for 语句的电路功能比较难理解，可以将这些语句展开，增强代码的可读性。

8.7 如何提高代码的仿真效率

对于一个复杂的设计而言，仿真效率往往是芯片验证的瓶颈。本节将介绍几个要点，帮助 Verilog 用户提高 RTL 代码的仿真效率。

一、仿真精度越高，仿真效率越低

仿真时采用`timescale 1ns/1ns 比采用`timescale 1ns/100ps 的仿真效率高。

二、减少层次结构

在设计中层次结构越少，仿真速度越快，这是因为参数在 module 之间通过端口传递会消耗仿真器的时间。

三、进程越少，仿真效率越高

语言中出现的进程越少，仿真越快。比如实现一个状态机，采用两个 always 结构就比使用一个 always 结构仿真慢，因为仿真器在多个进程之间切换也需要时间。

四、减少门级原语的使用，尽量采用行为描述

在描述同样的功能时尽量采用行为描述，少用 Verilog 的门级原语。建模的抽象层次越高，仿真效率越高。

五、尽量使用 case 语句，而不是 if...else 语句

如果使用 case 语句和 if...else 语句能够实现相同的电路，则应尽量使用 case 语句，以提高仿真效率。

六、减少 begin...end 语句块的使用

在语言不发生歧义的情况下尽量少用 begin...end 语句块，以提高仿真效率。

七、减少仿真器的输出显示

过多使用仿真器的输出显示系统任务，如语句$display、$fdisplay 等，会降低仿真器执行的速度。

最后要提醒读者的是，以上几点只是用来提高仿真效率的建议，并不是说设计代码一定要按照这几点来做，而是建议读者在保证代码的可读性、可维护性和安全性的前提下，尽量采用能提高仿真效率的设计方法，节约仿真和调试的时间，毕竟代码的可读性、可维护性和安全性才是最重要的。

8.8 防止仿真和综合结果不一致

本章 8.2.3 小节中提到过，保证 Verilog 代码符合仿真语义是一项非常重要、非常有意义的工作，是成功的第一步。

只有符合仿真语义、没有歧义的代码，才能被综合工具综合成想要的电路，否则综合结

果将与仿真结果不一致，这是相当危险的。下面讨论一些常见的问题。

一、不完全敏感表

不完全的 always 语句敏感表可能不会对综合工具产生什么影响，但是对仿真工具来说却是致命的，因为在仿真时，always 语句是完全依靠敏感表来触发执行的。

二、case 语句

在使用 case 语句时，通常在最后一句使用 default 语句，使得所有情况都可以有一个初始值。但是有时候综合工具会将最后一个 default 语句忽略。例如在一个状态机中，一个用 case 语句写出的次态逻辑在仿真时由于有 default 语句，因此看起来像是一个安全状态机。但是经过综合工具的综合，实现的电路并没有将所有的 default 情况都包含在其中，因此一旦状态机进入非法状态，就无法再恢复了。

三、初始化

在 Verilog 语言仿真时，寄存器变量缺省值为 X，线网变量缺省值为 Z，但是在实际综合后的芯片内部不存在 X 这个变量值，也很少有 Z 这个值，所以在综合以后的电路中并不一定也是这样的值。

在 Verilog 语句中可以对变量进行初始化，但是这个初始化对综合工具不起任何作用，仅在仿真器中有效。

四、代码中的延时参数

为了便于仿真，我们常常会在代码中加入延时参数，需要注意的是，这些延时参数将被综合工具忽略，导致仿真结果与综合的电路不一致。

8.9 小结

本章围绕着 Verilog 的仿真语义，深入详细地讲解了 Verilog 的仿真过程和原理，介绍了 Verilog 中的分层事件队列、仿真参考模型、时序模型、延时、阻塞赋值及非阻塞赋值等内容。本章所讲的内容是 Verilog 语言中最难理解的部分，希望读者能够认真学习。

8.10 问题与思考

1. 简述电路与软件处理流程的区别。
2. 简述仿真时间的概念。
3. 简述进程和调度的含义。
4. 什么是分层事件队列？
5. 阻塞赋值与非阻塞赋值最大的区别是什么？
6. 举例说明如何提高仿真效率。

第9章　设计与验证语言的发展趋势

在最近 20 年中，数字系统的设计方法（methodology）发生了革命性的变化，由主流的自底向上的设计方法（从基本的逻辑门开始，逐渐构造较为复杂的数字系统）逐渐转变为硬件描述语言（HDL）的自顶向下的设计方法，而不再过多地关心数字逻辑的门级实现细节，大大提高了设计的抽象层次和生产率（productivity）。

硬件描述语言在 RTL 设计中扮演着重要的角色，经过历史的选择，VHDL 和 Verilog 两种硬件描述语言被广大设计者所接受，它们也先后成为 IEEE 的标准。同时，各大 EDA 厂商相继推出基于 VHDL 和 Verilog 的仿真器和 RTL 综合器，进一步促进了这一设计方法的推广。

随着设计复杂度（complexity）的逐渐增加，设计和验证几十万、甚至几百万门级的 ASIC 和 FPGA 面临着前所未有的挑战。业界的领导者们纷纷寻求新的设计和验证方法，以解决他们在工作中遇到的麻烦，因此在设计和验证领域出现了群雄四起的局面。这时候，国际标准化组织 Accellera 和 IEEE 也开始着手制定这一领域的标准。

本章主要分析当今硬件描述语言（HDL）和硬件验证语言（HVL）的发展方向，供读者参考。

本章主要内容如下：

- 设计与验证语言的发展历程；
- 硬件设计语言的发展现状和走向；
- 验证语言的发展现状和走向；
- 总结和展望。

9.1　设计与验证语言的发展历程

本节将按照时间顺序来介绍设计与验证语言的发展历程。

9.1.1　HDL 语言

在硬件描述语言刚刚被人们采用的时候，设计的规模并不大，有的设计者认为不需要进行仿真，即使进行仿真，也是利用 PLD 供应商的工具自带仿真器手动为输入信号增加激励，手动检查各激励的输出。

慢慢地，设计者开始用 HDL 语言构建 Testbench。由于 Testbench 的编写并不受综合工具的限制，因此可以使用 HDL 中层次较高的描述语法和数据类型。在这一阶段，设计和验证语言可以认为是统一的。随着设计复杂度的进一步提高，验证一个设计通常需要编写大量的测试向量，验证过程逐渐成为缩短上市时间的瓶颈，人们需要一种更灵活、描述能力更强

的语言出现。一般来说，VHDL 的描述抽象层次要高于 Verilog，因此也有设计者使用 VHDL 的 Testbench 来验证 Verilog 的 RTL 设计。但是由于受当时大多数仿真器的限制，两种语言引擎混合仿真的效率要低得多。

9.1.2 C/C++和私有的验证语言

后来，很多 EDA 供应商开始考虑使用 C/C++来进行设计和验证。C/C++确实比 HDL 具有更高层次的描述能力，但其本身并不具有 HDL 的并行性，所以产生了很多类似的工具，使 C/C++看起来更像 Verilog，如 SynApps 等。由于 C/C++是独立工作的，因此它们需要通过 HDL 语言的编程语言接口（PLI）和 HDL 的仿真器连接起来。很多人对此提出批评，他们认为这样会严重影响仿真器的工作性能。另外，一些业界的领导者还开发了自己私有的 HVL。这些 HVL 通常继承了 Verilog 和 C/C++的特性，增加了一些用于验证的重要功能，如断言检查（Assertions Checking）等，这些语言主要包括 Intel 的 ForSpec、IBM 的 Sugar、Verisity 的“e”、Synoposys 的 OpenVera 和 Motorola 的 CBV 等。

9.1.3 Accellera 和 IEEE 的标准化工作

从设计的抽象层次和可验证性方面考虑，业界需要一种比 Verilog 更新的设计语言，而从标准化硬件验证语言方面考虑，则需要形成 HVL 标准，因为基于语法的竞争毫无意义。由 OVI（Open Verilog International）和 VI（VHDL International）合并成的标准化组织 Accellera 正在努力为业界制定新的设计和验证语言标准，其手下有 3 份工作正同时进行。

- VEV（Verilog Formal Verification）委员会已经在 2002 年 4 月将 IBM 的 Sugar 语言定为 Formal Property Language 的标准。
- 2002 年 6 月，Accellera 的 HDL+委员会又宣布推出下一代 Verilog 语言的标准 SystemVerilog。
- OVL（Open Verification Library）正在制定一种可用于 Verilog 和 VHDL 的 monitor 机制，用于仿真验证。

“天下大势，合久必分，分久必合”这句古语仍适用于形容 HDL 和 HVL 的发展方向。笔者认为 HDL 和 HVL 从统一走向分离后，将会再一次走向统一。HDL 和 HVL 的预期发展趋势如图 9-1 所示。

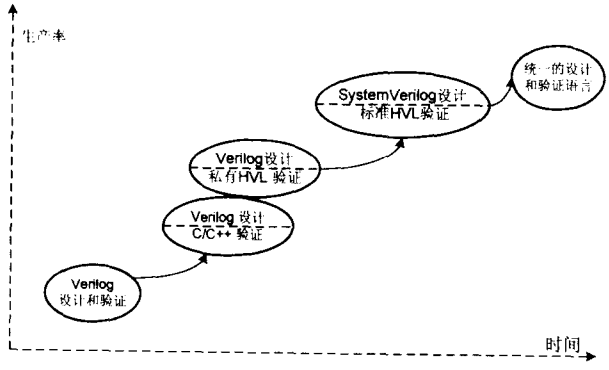

图9-1　HDL 和 HVL 的预期发展趋势

9.2 硬件设计语言的发展现状和走向

9.2.1 HDL 的竞争

设计语言的发展过程可以从最具代表性的 Verilog 语言的发展过程去分析。随着 Verilog 自身的发展和其他验证技术的进步，VHDL 逐渐被多数人认为是鸡肋，因为其相对于 Verilog 抽象层次较高的优越性已逐渐被 C/C++和其他私有的验证语言所取代，而 VHDL 本身固有的缺陷却无法消除（如其门级 VITAL 库的仿真效率等）。

Verilog 95 为我们提供了一些最基本的建模能力，如硬件并发行、时序控制和混合的 RTL/gate/switch 等描述方法。Verilog 2001 虽然在此基础上又增加了一些新的特性，但并没有使 Verilog 产生质的变化。一般来说，在 RTL 建模方面，Verilog 仍然是最有效、最准确的建模语言，虽然其在某些细节上的表现差强人意。

9.2.2 一些尝试

随着设计规模和复杂度的进一步增加，Verilog 的一些缺点正逐渐突显，主要表现在以下几个方面：

- 描述的抽象层次比较低；
- 数据类型呆板；
- 模块间的互连线复杂，容易出错；
- 不具有断言检查（Assertion Checking）的能力。

面对上面这些缺陷，很多人想到了 C/C++。不可否认，C/C++确实是一种非常优秀的语言，具有 Verilog 所不具有的许多优点，如数据类型灵活，抽象层次高，可以基于对象建模等。但是它毕竟不是为设计硬件而发明的语言，在描述硬件时也有其固有的不足，需要在许多方面作层层改装，才可以用于硬件建模。另外，由于几家 EDA 巨头对开发 C/C++建模综合工具并不热心，所以并没有引起人们的太多兴趣。

最重要的问题是，现在 Verilog 拥有众多的使用者，有不计其数的 Verilog 代码正在被使用，而且很多硬件工程师并不是很熟悉 C。如果设计师都离开了 HDL 语言，那么他们会在一定程度上丧失对硬件建模的能力。正确的解决方案是扩展那些正在起作用的东西，而不是用一个全新的东西去完全取代它。

9.2.3 下一代的 Verilog 语言

基于此，Accellera 在 2002 年 6 月推出了下一代 Verilog 标准 SystemVerilog。SystemVerilog 几乎可以弥补现有 Verilog 的所有缺陷，如采用隐式的模块间互连和更高层次的数据类型等。它混合了 Verilog、C/C++和 Co-Design Automation 公司 Superlog 语言的一个断言能力（Assertion Capability），用以提供更高的抽象层次。所谓断言检查就是在仿真过程中检查某个自己定义的事件是否发生，这一点非常重要。断言检查能力使得自底向上的可验证实现过程变为可能，而且自底向上的验证过程改善了基于仿真的方法学（Simulation-

Based Methodology），为通向形式验证（Formal Verification）提供了一条无缝之路。

SystemVerilog 最根本的意图是为 Verilog 提供一个新的建模抽象层次，并且扩展其验证大型设计的能力。目前硬件描述语言 SystemVerilog 标准已获 IEEE 通过，成为 IEEE 的正式标准。EDA 业界的巨头们已经开始在工具中支持 SystemVerilog 了。

SystemVerilog 和 Verilog、C 语言的关系如图 9-2 所示。

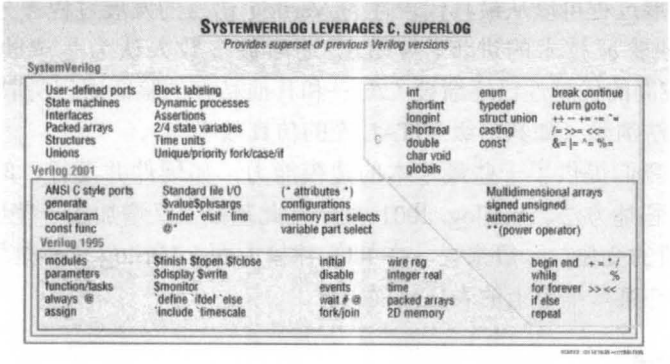

图9-2　SystemVerilog 和 Verilog、C 语言的关系

SystemVerilog 实际上是 Co-design Automation 公司 Superlog 的 ESS （Extended Synthesizable Subset） 部分。Superlog 语言与其他语言的关系如图 9-3 所示。

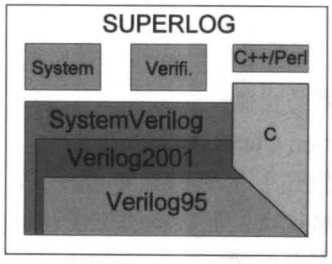

图9-3　Superlog 语言和其他语言的关系

Superlog 语言是一种比较理想的、集设计和验证于一体的语言，但是或许是基于 EDA 业界各家供应商利益分配因素的考虑，或出于其他技术因素的考虑，Superlog 并没有被 Accellera 直接作为标准，而是用它的一个子集（ESS）作为 SystemVerilog 的蓝本，其 Formal Property Language 标准则采用了 IBM 的 Sugar 语言。

9.2.4　SystemC

这里值得一提的是 Synoposys 公司推出的新一代系统建模语言 SystemC。SystemC 在推出的初期被人认为是一种 HDL 的替代品，人们怀疑 SystemVerilog 的出现是否会影响 SystemC 的生存。

然而，SystemC 的支持者却认为两者定位的用户群不一样。他们强调，SystemC 的基本目标是定位在那些已经使用了 C/C++的系统设计者，而不是 RTL 的硬件设计者。Open SystemC Initiative 的主席 Stan Krolikoski 表示，“在 RTL 设计的人们需要提高他们的设计抽象层次，所以像 SystemVerilog 这种东西将会对他们非常有意义。”言下之意，SystemC 不是

为 RTL 工程师们设计的语言，而是为那些熟悉 C/C++软件的系统工程师们设计的。

目前 IEEE 也启动了 SystemC 的标准化工作，相信它在系统建模领域将赢得一席之地。

9.3 验证语言的发展现状和走向

9.3.1 验证方法

在现今的 EDA 业界，各家 EDA 工具厂商正纷纷采用自己的方法来帮助验证设计。

从工具方面来说主要采用两种工具进行验证，一种是使用基于 C、C++、TCL 或 Perl 语言的工具，如 CynApps 等，这些工具使 C/C++看起来更像 Verilog；另一种是使用基于私有验证语言的工具，如 OpeaVera、Specman E 和 Superlog 等，在这两类工具中，后者逐渐成为主流。

从验证方法学方面来说，主要有传统的基于仿真的验证方法学和形式验证方法学两种。

这里讲的形式验证是指模型检查，也就是回答下面的问题："我的实现结果满足设计规格吗？"这就要求必须把规格描述成一个需要被验证的特性的集合，要求验证语言拥有强大的断言（assertion）机制来支持。虽然在短期内 RTL 设计工程师不会使用专用的验证语言，但这些验证语言很快会被构架设计师和验证工程师所采用。

另外随着 Accellera 标准化进程的推进，RTL 工程师将可以在自己的 HDL 设计中指定 RTL 实现的特性，通过使用 SystemVerilog 语言和 Open Verification Library，来改善基于仿真的验证方法学。

9.3.2 HVL 标准化进程

2002 年 4 月，Accellera 宣布将 IBM 的 Sugar 语言作为其 Formal Property Language 的标准，称它将推动基于断言的验证，并与不久前 Synoposys 和 Intel 联合发布的 OpenVera2.0 展开竞争。

正当人们担心验证语言领域将出现激烈对抗的时候，在 2002 年 6 月 11 日召开的设计自动化大会（DAC）上，Synoposys 宣布将 OpenVera2.0 的 Assertion 部分捐献给 Accellera，从而避免了两者的正面冲突。

在 SystemVerilog 成为 IEEE 正式标准以后，许多验证语言已经显得较为黯淡，但 e 语言却受到了较大的关注。

e 硬件验证语言曾是一种专属语言，是 Verisity 公司 Specman 产品的一部分。Verisity 携同 Specman 一起被 Cadence 收购，而 Cadence 同时也是 SystemVerilog 的强劲支持者，因此引发了关于该公司如何计划以处置两种语言的疑问。

开始很多观察家认为，被 IEEE 批准为标准的 SystemVerilog 将置 e 于死地。根据年初 EDA 工具用户网站 Deepchip.com 的 John Cooley 进行的调查发现，78%的受访者认为功能验证语言 e 和 Vera 将在 5 年内死亡。

不过 e 语言已经有了比较广泛的用户群，目前 e 语言的标准化工作也正在进行，IEEE 发起者关于将 e 硬件验证语言标准化的投票以压倒性的票数获得通过。

9.3.3 HVL 的新需求

目前验证语言已经逐渐被许多人所接受。一种 HVL 语言的基本需求是支持高层次的数据类型，面向对象功能，并行控制和设计的可观察性。正是这些基本需求使得相当一部分用户转向使用 C/C++来实现 Testbench。C++加上 PLI 很容易满足这些基本需求。但是这些基本需求只可以用来写定向的、自检测的、事务级的 Testbench，并不能满足验证实现上的一些基本转变。

为了适应下一代的百万门设计，HVL 必须拥有 3 种互补的工具来提供一种新的功能验证方法，这 3 种工具分别为：

- 可约束的随机发生器；
- 时间断言；
- 功能覆盖率检查。

可约束的随机发生器允许使用者增加适当的约束，使随机发生器向着需要的方向产生激励。

关于时间断言，本章前面也有描述，主要用于检查 Properties 的正确性。

另外，可以通过功能覆盖率检查工具获知设计验证是否完毕。代码覆盖率会提问“Testbench 是否忘记了执行这行代码？”，而功能覆盖率会提问“在所有可能的数据值顺序下，设计都工作吗？”代码覆盖率可以帮助检查现存代码中的错误，而功能覆盖率则可以帮助检查未实现功能的错误。

9.4 总结和展望

笔者相信，在硬件设计领域，SystemVerilog 将是业界将来的领导者，而在硬件设计领域，SystemVerilog 也具有不错的表现，同时，e 语言也将成为一种受到广泛关注的硬件验证语言。

HDL 和 HVL 是 EDA 业界的重要基础。先进的语言必须和先进的方法学一起使用，否则根本不能体现出语言的优越性。另外必须记住，即使语言具有种种强大的功能，也不一定能充分利用之，学会一门语言容易，而学精一门语言并不容易，这需要付出很多的努力。

9.5 小结

本章主要介绍了 HDL 和 HVL 的发展，以及新的验证思想和技术。

9.6 问题与思考

1. 简述 HDL 的发展趋势。
2. SystemVerilog 与 Verilog 相比有什么优势？
3. 验证语言需要有哪些新特性？

附录　Verilog 关键字列表

关键字是 Verilog HDL 中预定义的、非转义的标识符，用来定义 Verilog 的语言结构。转义字符不能被看作关键字，而且所有的关键字必须都是小写的。

以下是以字母顺序排列的 Verilog 关键字列表。

always	ifnone	rnmos
and	incdir	rpmos
assign	include	rtran
automatic	initial	rtranif0
begin	inout	rtranif1
buf	input	scalared
bufif0	instance	showcancelled
bufif1	integer	signed
case	join	small
casex	large	specify
casez	liblist	specparam
cell	library	strong0
cmos	localparam	strong1
config	macromodule	supply0
deassign	medium	supply1
default	module	table
defparam	nand	task
design	negedge	time
disable	nmos	tran
edge	nor	tranif0
else	noshowcancelled	tranif1
end	not	tri
endcase	notif0	tri0
endconfig	notif1	tri1
endfunction	or	triand
endgenerate	output	trior
endmodule	parameter	trireg
endprimitive	pmos	unsigned

续表

endspecify	posedge	use
endtable	primitive	vectored
endtask	pull0	wait
event	pull1	wand
for	pulldown	weak0
force	pullup	weak1
forever	pulsestyle_onevent	while
fork	pulsestyle_ondetect	wire
function	rcmos	wor
generate	real	xnor
genvar	realtime	xor
highz0	reg	
highz1	release	
if	repeat	