Verilog HDL 硬件描述语言
程序设计与实践教程

云创工作室

人民邮电出版社

前言

目前，EDA 技术已经成为现代电子设计领域的基本手段，涵盖印制电路板（PCB）设计、可编程逻辑芯片开发、专用集成芯片设计以及系统验证等诸多领域。硬件描述语言（HDL）是 EDA 技术中的一个重要组成部分，可应用于除 PCB 设计外的各个领域。

Verilog HDL 语言为两个主流 HDL 语言之一（另一个是 VHDL 语言），在全球范围具有广泛的用户群，具有超过 80%的行业覆盖率。在美国，使用 Verilog HDL 进行设计的工程师大约有 l0 多万人，占 HDL 设计行业工程师的 90%左右，并有超过 200 多所大学开设有 Verilog HDL 语言的相关课程，包括斯坦福大学、卡梅隆大学这样的著名高校。在中国，业界大约有一半的工程师在使用 Verilog HDL 语言，并且这一比例还在快速上升；在教育界有以夏宇闻老师为代表的各位前辈进行了初期推广，目前已具备较大的应用规模。其实就学习难度而言，Verilog HDL 语言要比 VHDL 简单的多，且和 C 语言语法风格类似，更容易被在校大学生和初学者接受。

Verilog HDL 语言的学习是一个实践性很强的过程，直接上手就去完成芯片设计，需要大量的资金和昂贵的成本，并面临极大的风险，对于大多数在校学生、工程师和企业而言都是无法承受的，因此可编程逻辑器件（CPLD、FPGA）就成为最好的学习和创新平台。为了让更多的在校学生和初学者更好地切近行业需求，同时为了改善高校教学内容，Xilinx 大学计划（Xilinx 公司是全球最大的可编程逻辑器件提供商）和上海智翔信息服务技术有限公司推出了一系列教材和相关课程，本书也是其中的一本，通过 Xilinx 公司的 FPGA 平台来介绍基于 Verilog HDL 语言开发的相关知识。

本书按照开门见山、自顶向下的方式来组织内容，在介绍相关知识点之前，先告诉读者其出现的背景、本质特征以及如何语法，让读者不仅掌握基本语法，并能够获得深层次理解。全书内容分为 13 章，第 1 章为 EDA 设计与 Verilog HDL 语言概述；第 2 章　Verilog HDL 基础与开发平台操作指南；第 3 章为 Verilog HDL 程序结构；第 4 章为 Verilog HDL 语言基本要素；第 5 章为面向综合的描述语句；第 6 章为面向验证和仿真的描述语句；第 7 章为系统任务和编译预处理语句；第 8 章为 Verilog HDL 难点解析；第 9 章为高级逻辑设计思想与代码风格；第 10 章为可综合状态机开发实例；第 11 章为常用逻辑的 Verilog HDL 实现；第 12 章为 Xilinx 硬核模块的 Verilog HDL 调用；第 13 章为 RS232 串口总线接口模块的设计。本章均附有一定的思考题，供读者练习和自我检查。

本书语言简洁，层次清晰，以 Verilog HDL 语言的各方面开发为主线，遵照硬件应用系统开发的基本步骤和思路进行详细讲解，并穿插 ISE 开发工具的操作技巧与注意事项，具备很强的可读性、指导性和实用性。本书在成书过程中，参考了多篇文献，包括书籍和论文，这里向其作者表示感谢。本书是云创工作室的团队合作的成果，在这里感谢给予我们支持、帮助的师长和朋友们。

本书主要面向通信、电子、计算机等相关专业大学生，也适合从事电子设计和开发人员阅读，特别是希望系统学习 Verilog HDL 语言的工程师。

目录

第 1 章　EDA 设计与 Verilog HDL 语言概述

从 20 世纪 70 年代中期以来，数字电路技术获得了突飞猛进的发展，最初的数字集成电路只有几个门逻辑，到目前为止已经出现了千万门以上的大规模集成电路。这使得电路设计变得非常复杂，已经超出设计人员脑力运算的能力，必须通过计算机的自动化设计来完成。事实上，电子设计自动化技术在 20 世纪 60 年代就出现雏形，历史比数字集成电路的发展还悠久，涵盖了电路板设计以及芯片设计的各个领域。随着片上系统（System on Chip，SoC）和片上可编程系统（System on Programable Chip，SoPC）潮流的兴起，EDA 软件已经能完成系统级的自动化设计。Verilog HDL 语言是 EDA 软件的主要输入方式，在可编程逻辑器件和专用集成电路设计中有着广泛的应用。本章主要对 EDA 技术、Verilog HDL 语言及其描述层次进行简要说明，为后续章节的学习奠定基础。

1.1 EDA 设计概述

1.1.1　EDA 技术简介

电子设计自动化（EDA，Electronic Design Automation）是指利用计算机完成电子系统的设计，以计算机和微电子技术为先导，汇集了计算机图形学、逻辑学、微电子工艺和结构学以及计算数学等多种计算机应用学科最新成果的先进技术。简言之，EDA 技术就是利用软件程序和工具来设计并实现硬件产品。

从 20 世纪 60 年代中期开始，人们就不断在研究各种计算机辅助设计（Computer Aided Design，CAD）工具来提高电子设计人员的效率，主要是一些单独的印制电路板（Printed Circuit Board，PCB）软件，用于布线设计、电路模拟、逻辑模拟、版图的绘制等，利用计算机的计算功能，将设计人员从大量繁琐重复的计算和绘图工作中解脱出来。

20 世纪 80 年代初，随着集成电路规模的快速发展，CAD 技术发展到计算机辅助工程（Computer Aided Engineering，CAE）技术，主要表现为设计工具和单元库完备，具备原理图输入、编译和连接、逻辑模拟、测试代码生成、版图自动布局等功能。CAE 软件要针对产品开发，按照设计、分析、生产、测试等划分为多个阶段，不同阶段使用不同的软件，每个软件完成其中的一项工作，通过顺序循环使用这些软件，可完成设计的全过程。这一阶段的重大事件还包括：CPLD、FPGA 芯片的面市以及 HDL 语言的出现。

尽管 CAD/CAE 技术取得了巨大的成功，但并没有把设计人员从繁重的设计工作中彻底解放出来。在整个设计过程中，自动化和智能化程度还不高，各种软件界面千差万别，学习使用困难，并且互不兼容，直接影响到设计环节间的衔接。基于以上不足，人们开始追求将整个设计过程的自动化，也就是电子系统设计自动化（Electronic System Design Automation，ESDA）。此外，进入 20 世纪的 90 年代中期后，微电子技术以惊人的速度发展，在单芯片内可集成数百万乃至上千万只晶体管，工作速度达到 GHz 以上，对 EDA 技术提出了更高要求，也促进了 EDA 技术在系统设计领域的飞速发展。从上可以看出，EDA 技术的每一次进步，都引起了设计层次上的一个飞跃，如图 1-1 所示，其中的物理级设计主要指版图设计。

目前的 EDA 技术以高级语言描述、系统级仿真和综合技术为特征，自动将用户以高级语言描述的功能需求转化为基础门电路，在应用数量较少时可以将设计封装到 CPLD/FPGA 芯片中，在大规模应用时再将设计制作成 ASIC 芯片，极大地提高了系统的设计效率，使得设计人员摆脱了大量的辅助性和基础性工作，将精力集中于创造性的方案与算法和系统的构思上。

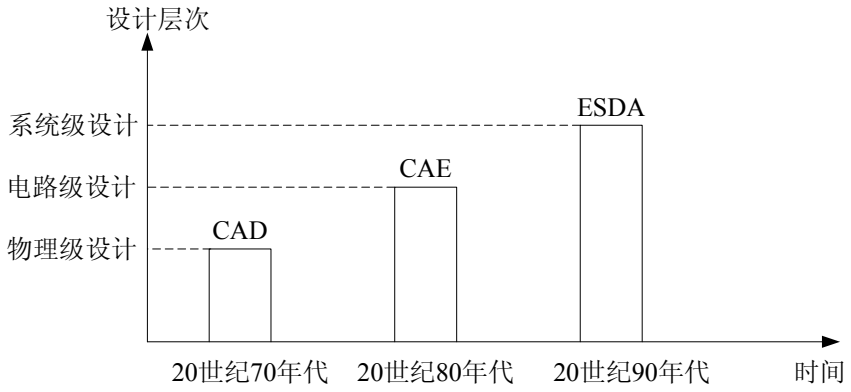


图 1-1　EDA 设计层次示意图

这一阶段的 EDA 工具种类多、系统集成度高，大体可以分为两类：一类是专业的 EDA 软件公司产品，比较著名的有 Menter Graphic 公司系列产品（ModelSim、PADS、WG 等）、Cadence 公司系列产品（Incisive、Encounter 以及 Allegro 等软件）、Synopsys 公司系列产品（Galaxy Design Platform 等平台）、Altium 公司系列产品（Protel DXP 等）；另一类是半导体器件厂家开发的专门用于自身产品开发的 EDA 工具，比较著名的包括 Xilinx 公司的 ISE 软件、TI 公司的 CCS 软件等。

EDA 专业公司的软件独立于半导体器件厂商，其 EDA 工具具有较好的标准化和兼容性，追求技术上的先进性，一般来讲适合专业的 IC 设计公司和深层次的学术研究人员使用。半导体器件厂商开发的软件都针对自己芯片的工艺特点做了优化设计，尽可能高地提高资源利用率、降低功耗、改善性能，非常适合产品开发和高校教学科研。在实际中，专业 EDA 公司和半导体提供商的合作非常紧密，两类公司经常在先进技术开发、器件生产、产品服务以及 IP 模块等方面结成战略联盟。

1.1.2　EDA 与传统电子系统设计方法

1．设计模式的比较

（1）传统自底向上的设计模式

传统的电子系统设计模式通常是自底向上的，即首先确定系统最底层的电路模块或元件的结构和功能，然后根据系统的功能要求，将元件组成更大的功能块，使得功能块的结构和功能满足上层系统的要求，依次类推，直到完成整个目标系统的设计。

例如，利用传统电子系统设计方法设计一个音频处理系统，首先必须决定使用器件的类别和规格，如 CPU/FPGA 等控制器件、RAM/ROM 等存储器件、专用的音频处理芯片、数据采集芯片等；然后构成多个功能模块，通过控制器件将数据采集芯片、专用音频处理芯片、RAM/ROM 等存储器件逐级连接起来，直到最后完成整个系统的设计。

上述设计方法的特点是：必须首先关注并致力于解决最底层硬件单元的可实现性，包括功能特性和可获得性；其次，在逐级设计和测试过程中，必须始终考虑所有目标器件的技术

细节，直到最后，才能发现底层器件的技术参数不满足总体要求。在开发过程的最后阶段才发现错误，必将导致以前所有的工作都前功尽弃。由此可见，自底向上的设计方法是一种低效、高风险且低可靠性的设计方法。在目前的超大规模集成电路时代，这种方法已被彻底放弃。

（2）EDA 时代自顶向上的设计模式

只有 EDA 工具具备强大的功能后，在电子设计领域应用自顶向下的设计模式才成为可能。所谓自顶向下的开发模式就是将设计流程中的各个环节逐步分解、求精的过程，首先从系统设计入手，在顶层进行功能方框图的划分和结构设计。在方框图级进行仿真、纠错，并用硬件描述语言对高层次的系统行为进行描述，在系统级进行验证。然后用综合优化工具生成具体门电路的网表，其对应的物理实现级可以是印刷电路板或专用集成电路。由于设计的主要仿真和调试过程是在高层次上完成的，这一方面有利于早期发现结构设计上的错误，避免设计工作的浪费，同时也减少了逻辑功能仿真的工作量，提高了设计的一次成功率。这样的设计模式被称为高层次的电子设计模式。

高层次设计是一种“概念驱动式”设计，设计人员无需通过门级原理图描述电路，而是针对设计目标进行功能描述。由于摆脱了底层细节的束缚，设计人员可以把精力集中于创造性的方案与概念构思上，一旦这些概念构思以高层次描述的形式输入计算机后，EDA 工具就能以规则驱动的方式自动完成整个设计。这样，新的概念得以迅速有效地成为产品，大大缩短了产品的研制周期。不仅如此，高层次设计只是定义系统的行为特性，可以不涉及实现工艺，在厂家综合库的支持下，利用综合优化工具可以将高层次描述转换成针对某种工艺优化的网表，工艺转化变得轻松容易。

例如，利用自顶向下的模式实现一个音频处理系统，首先对整个系统进行方案设计和功能划分，系统的数据采集电路用一片模数转换器的专用集成电路（ASIC）实现，然后分解系统，通过硬件描述语言（HDL）完成系统级和算法级的设计，最后通过综合器和适配器生成最终的目标器件。

2．设计方法的比较

（1）传统设计方法

在传统的电子系统设计方法中，手工设计占了绝对比例。首先需要根据电路功能需求进行功能划分，然后分析每个子模化并画出其相应的真值表，用卡诺图进行手工逻辑简化，写出布尔表达式，得到相应的逻辑线路图，然后再根据此选择元器件，设计电路板，最后进行调试与测试。其缺点如下：

- 复杂电路的设计和调试十分困难；
- 不易修改，也不便于寻找错误；
- 设计过程中产生的文档不易管理；
- 设计的移植性和可重用性差；
- 只有设计出样机才能进行测试，一旦发现问题，则需要再次手工设计，之前所有的设计都将被抛弃，风险大。

（2）基于 EDA 的设计方法

EDA 设计主要采用硬件描述语言作为设计输入，包括抽象行为与功能描述，甚至到内部的具体线路结构；再借助于计算机将设计自动转化为底层逻辑。这一自动编译转换过程不需要人工参与，并且可以在系统设计的各个阶段、各个层次进行计算机模拟验证，保证设计过

程的正确性。这样，不仅提高了设计人员的效率，还极大地降低了风险。其特点如下：

- 强大的系统建模和电路仿真功能。EDA 技术和传统设计方法最大的区别就在于自动将功能描述转化为系统底层设计的能力，设计人员只需输入功能和行为描述语句。此外，EDA 仿真测试技术只需要通过计算机，就能对所设计的电子系统从各种不同层次的系统性能特点出发完成一系列准确的测试与仿真操作。即使在完成设备后，还可以通过修改 HDL 代码对系统进行修改。
- 引入了“库”的概念。“库”本身就是专门存放预先编译好的程序包（package）。这样就可以在其他设计中被调用，它实际上对应一个目录，预编译程序包的文件就放在此目录中。EDA 工具之所以能够完成各种自动设计过程，关键就是各类库的支持，如逻辑仿真时序库、综合库、布局布线的版图库等。一般来讲，库都是 EDA 设计公司和半导体生产商紧密合作、共同开发的。
- 保护了设计人员的知识产权。不管传统的电子系统设计的多么完美，完成了多么先进的功能，但系统对于其设计者来讲没有任何自主产权而言，因为系统采用的数字锁相环、单片机以及其他特定功能的 IC，对于别人而言一目了然，且系统的关键器件并非出自设计者之手，这将导致系统的应用直接受到限制。而采用 EDA 设计的系统，是通过 HDL 语言实现用户特定功能的，并封装在通用的 CPLD/FPGA 芯片或者制作成 ASIC 芯片，设计者拥有完全的自主权。
- 提高了设计性能、可靠性和可重用性。传统的电子设计方法至今没有任何标准规范加以约束，因此效率低、系统性能差、开发成本高、竞争力小且可重用性低。以单片机/DSP 开发为例，每一次新的开发，必须选用性价比更高的处理器，但由于处理器的结构不同，语言和硬件特性也有很大差异，因此设计者每次都需要重新了解处理器的详细结构和电气特性，重新编写功能需求代码。但利用 EDA 技术则完全不同，其设计语言是标准化的，不会由于硬件平台的不同而改变，因此其设计结果是通用性的，具有规范的接口协议、良好的可抑制和可测试性，为高效、高质、高可靠的系统开发提供给出。
- 有效地管理了文档。由于所有设计是通过 HDL 代码实现的，其本身就是文档型的语言，极大地简化了设计文档的管理。
- 降低了对设计人员的硬件知识和经验的要求。传统的电子设计手段对设计人员有较高的要求，需要丰富的硬件经验和熟练的软件操作，掌握系统各个方面的设计技巧，这显然不符合现代电子技术快速更新换代的特点，也难以满足设计人员的需求。EDA 技术的标准化和设计平台对具体硬件的无关性，使得设计人员将精力和创造力主要集中在项目性能的提高和优化上，不必去了解更底层的东西。

1.1.3 可编程逻辑器件对 EDA 技术的要求

可编程逻辑器件（Programmable Logic Device，PLD）起源于 20 世纪 70 年代，是在专用集成电路（ASIC）的基础上发展起来的一种新型逻辑器件。目前一般以 CPLD 和 FPGA 器件为主，是当今数字系统设计的主要硬件平台，其主要特点就是完全由用户通过软件进行配置和编程，从而完成某种特定的功能，且可以反复擦写。在修改和升级设计时，不需额外地改变 PCB 电路板，只需要通过 EDA 开发工具修改和更新程序，使硬件设计工作成为软件开发工作，缩短了系统设计的周期，提高了实现的灵活性并降低了成本，因此获得了广大硬件工程师的青睐，形成了巨大的产业规模。这也是本书内容以 FPGA 为硬件平台的主要原因。

基于 CPLD 和 FPGA 的开发，完全由 EDA 工具来完成，包括“HDL 语言的逻辑编译、化简、分割、综合及优化、布局布线、仿真以及对于特定目标芯片的适配编译和编程下载等工作。典型的 EDA 工具中必须包含特殊的综合器软件包，其功能就是将设计者在 EDA 平台上完成的针对某个系统项目的 HDL、原理图或状态图形描述，针对给定的硬件系统组件，进行编译、优化、转换和综合。

随着开发规模的级数性增长，就必须减短 CPLD/FPGA 开发软件的编译时间、并提高其编译性能以及提供丰富的知识产权（IP）核资源供设计人员调用。此外，PLD 开发界面的友好性以及操作的复杂程度也是评价其性能的重要因素。目前在 PLD 产业领域中，各个芯片提供商的 PLD 开发工具已成为影响其成败的核心成分。只有全面做到芯片技术领先、文档完整和 PLD 开发软件优秀，芯片提供商才能获得客户的认可。一个完美的 PLD 开发软件应当具备下面 5 点：

- 准确地将用户设计转换为电路模块；
- 能够高效地利用器件资源；
- 能够快速地完成编译和综合；
- 提供丰富的 IP 资源；
- 用户界面友好、操作简单。

Xilinx 公司的 ISE 集成开发环境是业界公认的优秀集成 PLD 开发软件。此外综合软件 Synplify、仿真软件 ModelSim 以及软件辅助工具 Matlab 等诸多第三方 EDA 开发软件也满足上述要求。

1.2 Verilog HDL 语言简介

1.2.1 硬件描述语言说明

1．硬件描述语言的概念

硬件描述语言（Hardware Discription Language，HDL）语言以文本形式来描述数字系统硬件结构和行为，是一种用形式化方法来描述数字电路和系统的语言，可以从上层到下层来逐层描述自己的设计思想。即用一系列分层次的模块来表示复杂的数字系统，并逐层进行验证仿真，再把具体的模块组合由综合工具转化成门级网表，接下去再利用布局布线工具把网表转化为具体电路结构的实现。目前，这种自顶向下的方法已被广泛使用。概括地讲，HDL 语言包含以下主要特征：

- HDL 语言既包含一些高级程序设计语言的结构形式，同时也兼顾描述硬件线路连接的具体结构。
- 通过使用结构级行为描述，可以在不同的抽象层次描述设计。HDL 语言采用自顶向下的数字电路设计方法，主要包括 3 个领域 5 个抽象层次。
- HDL 语言是并行处理的，具有同一时刻执行多任务的能力。这和一般高级设计语言（例如 C 语言等）串行执行的特征是不同的。
- HDL 语言具有时序的概念。一般的高级编程语言是没有时序概念的，但在硬件电路中从输入到输出总是有延时存在的，为了描述这一特征，需要引入时延的概念。HDL 语言不仅可以描述硬件电路的功能，还可以描述电路的时序。

Verilog HDL 和 VHDL 是目前世界上最流行的两种硬件描述语言（HDL：Hardware

Description Language），均为 IEEE 标准，被广泛地应用于基于可编程逻辑器件的项目开发。二者都是在 20 世纪 80 年代中期开发出来的，前者由 Gateway Design Automation 公司（该公司于 1989 年被 Cadence 公司收购）开发，后者由美国军方研发。

2．HDL 语言的优势

传统的数字逻辑硬件电路的描述方式是基于原理图设计的，根据设计要求选择器件，绘制原理图，完成输入过程。这种方法在早期应用中得到了广泛应用，其优点是直观、便于理解；但在大型设计中，其维护性很差，不利于设计建设和复用。此外，原理图设计法有一个最致命的缺点：当所用芯片停产或升级换代后，相关的设计都需要做出改动甚至是重新开始。

HDL 以文本形式来描述数字系统硬件结构和行为的语言，不仅可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

随着人们对数十万门、数百万门乃至数千万门电路设计需求的增加，依靠传统的原理图输入已经不能满足设计人员的要求，和原理图设计方法相比，Verilog HDL 语言设计法利用高级的设计方法，有利于将系统划分为子模块，便于团队开发；且与芯片的工艺和结构无关，通用性和可移植强

1.2.2 Verilog HDL 语言的历史

1983 年，Gateway Design Automation（GDA）硬件描述语言公司的 Philip Moorby 首创了 Verilog HDL。后来 Moorby 成为 Verilog HDL-XL 的主要设计者和 Cadence 公司的第一合伙人。1984 至 1986 年，Moorby 设计出第一个关于 Verilog HDL 的仿真器，并提出了用于快速门级仿真的 XL 算法，使 Verilog HDL 语言得到迅速发展。1987 年 Synonsys 公司开始使用 Verilog HDL 行为语言作为综合工具的输入。1989 年 Cadence 公司收购了 Gateway 公司，Verilog HDL 成为 Cadence 公司的私有财产。1990 年初，Cadence 公司把 Verilog HDL 和 Verilog HDL-XL 分开，并公开发布了 Verilog HDL。随后成立的 OVI（Open Verilog HDL International）组织负责 Verilog HDL 的发展并制定有关标准，OVI 由 Verilog HDL 的使用者和 CAE 供应商组成。1993 年，几乎所有 ASIC 厂商都开始支持 Verilog HDL，并且认为 Verilog HDL-XL 是最好的仿真器。同时，OVI 推出 2.0 版本的 Verilong HDL 规范，IEEE 则将 OVI 的 Verilog HDL2.0 作为 IEEE 标准的提案。1995 年 12 月，IEEE 制定了 Verilog HDL 的标准 IEEE1364-1995。目前，最新的 Verilog 语言版本是 2000 年 IEEE 公布的 Verilog 2001 标准，其大幅度地提高了系统级和可综合性能。

1.2.3 Verilog HDL 语言的能力

Verilog HDL 既是一种行为描述语言，也是一种结构描述语言。按照一定的规则和风格编写代码，就可以将功能行为模块通过工具自动转化为门级互连的结构模块。这意味着利用 Verilog 语言所提供的功能，可以构造一个模块间的清晰结构来描述复杂的大型设计，并对所需的逻辑电路进行严格的设计。下面列出的是 Verilog 语言的主要特点：

- 可描述顺序执行或并行执行的程序结构；
- 用延迟表示式或事件表达式来明确地控制过程的启动时间；
- 通过命名的事件来触发其他过程里的激活行为或停止行为；
- 提供了可带参数且非零延续时间的任务程序结构；
- 提供了可定义新的操作符的函数结构；

- 提供了用于建立表达式的算术运算符、逻辑运算符和位运算符；
- 提供了一套完整的表示组合逻辑基本元件的原语；
- 提供了双向通路和电阻器件的描述；
- 可建立 MOS 器件的电荷分享和衰减模型；
- 可以通过构造性语句精确地建立信号模型。
- 在行为级描述中，Verilog HDL 不仅能够在 RTL 级上进行设计描述，而且能够在体系结构级描述及其算法级行为上进行设计描述。
- 能够使用门和模块实例化语句在结构级进行结构描述。
- 对高级编程语言结构，例如条件语句、情况语句和循环语句，语言中都可以使用。
- 可以显式地对并发和定时进行建模。
- 提供强有力的文件读写能力。
- 语言在特定情况下是非确定性的，即在不同的模拟器上模型可以产生不同的结果；例如，事件队列上的事件顺序在标准中没有定义。

此外，Verilog HDL 语言还有一个重要特征就是：和 C 语言风格有很多的相似之处，学习起来比较容易。

1.2.4 Verilog HDL 和 VHDL 语言的比较

1．相同点

Verilog HDL 和 VHDL 都是用于数字逻辑设计的硬件描述语言，二者相同点在于：都能形式化地抽象表示电路的行为和结构；支持逻辑设计中层次与范围的描述；可以简化电路行为的描述；具有电路仿真和验证机制；支持电路描述由高层到低层的综合转换；与实现工艺无关；便于管理和设计重用。

2．不同点

Verilog HDL 和 VHDL 最大的差别在语法上，Verilog HDL 是一种类 C 语言，而 VHDL 是一种类 ADA 语言。由于 C 语言简单宜用且应用广泛，因此也使得 Verilog HDL 语言容易学习，如果具有 C 语言学习的基础，很快就能够掌握；相比之下，VHDL 语句较为晦涩，使用难度较大，一般需要半年以上的专业培训才能够掌握。

此外，Verilog HDL 和 VHDL 又有各自的特点，由于 Verilog HDL 推出较早，因而拥有更广泛的客户群体、更丰富的资源。传统观点认为 Verilog HDL 在系统级抽象方面较弱，不太适合特大型的系统。大多数业界学者和工程师认为：VHDL 侧重于系统级描述，从而更多地为系统级设计人员所采用；Verilog HDL 侧重于电路级描述，从而更多地为电路级设计人员所采用，其在行为级抽象上的覆盖范围如图 1-2 所示。但这两种语言也仍处于不断完善的过程中，都在朝着更高级、更强大描述语言的方向前进；其中，经过 IEEE Verilog 2001 标准的补充之后，Verilog HDL 语言的系统级表述性能和可综合性能有了大幅度提高。

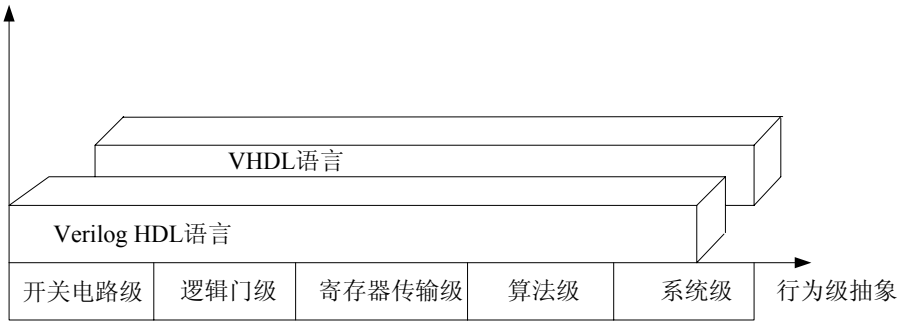


图 1-2　Verilog HDL 和 VHDL 建模能力比较

综上所述，Verilog HDL 语言作为学习 HDL 设计方法入门和基础是非常合适的。掌握了 Verilog HDL 语言建模、综合和仿真技术，不仅可以增加读者对数字电路设计技术的深入了解，还可以为后续阶段的高级学习打好基础，包括数字信号处理和无线通信的 FPGA 实现、IC 设计等领域。

1.2.5　Verilog HDL 和 C 语言的比较

虽然 Verilog 的某些语法与 C 语言接近，但存在本质上的区别。Verilog 是一种硬件语言，最终是为了产生实际的硬件电路或对硬件电路进行仿真；C 语言是一种软件语言，是控制硬件来实现某些功能。利用 Verilog 编程时，要时刻记得 Verilog 是硬件语言，要时刻将 Verilog 与硬件电路对应起来。二者的异同点如下：

（1）C 语言是由函数组成的，而 Verilog HDL 则是由称之为 module 的模块组成的。

（2）C 语言中的函数调用通过函数名相关联，函数之间的传值是通过端口变量实现的。相应地，Verilog HDL 中的模块调用也通过模块名相关联，模块之间的联系同样通过端口之间的连接实现，和 C 语言中端口变量所不同的是，模块间连接反映的是硬件之间的实际物理连接。

（3）C 语言中，整个程序的执行从 main 函数开始。Verilog HDL 没有相应的专门命名模块，每一个 module 模块都是等价的，但必定存在一个顶层模块，它的端口中包含了芯片系统与外界的所有 I/O 信号。这个顶层模块从程序的组织结构上讲，类似于 C 语言中的 main 函数，但 Verilog HDL 中所有 module 模块都是并发运行的，这一点必须从本质上与 C 语言加以区别。

（4）C 语言是运行在 CPU 平台上的，是串行执行的。Verilog HDL 语言用于 CPLD/FPGA 开发，或者 IC 设计，对应着门逻辑，所有模块是并行执行的。

（5）Verilog HDL 中对注释语句的定义与 C 语言类似。

1.3 Verilog HDL 语言的描述层次说明

1.3.1　Verilog HDL 语言描述能力综述

使用 Verilog HDL 语言可以从 3 个描述级别的 5 个抽象层次等不同层次描述数字典路系统，具体包括系统级、算法级、寄存器传输级（Register Transfer Level，RTL）、门级和开关级，如表 1-1 所列。

表 1-1　　Verilog HDL 语言设计层次总结

描述级别	抽象级别	功能描述	物理模型
行为级	系统级	用语言提供的高级结构能够实现所设计模块外部性能的模型	芯片、电路板和物理划分的子模块
	算法级	用语言提供的高级功能能够实现算法运行的模型	部件之间的物理连接，电路板
	RTL 级	描述数据如何在寄存器之间流动和如何处理、控制这些数据流动的模型	芯片、宏单元
逻辑级	门级	描述逻辑门和逻辑门之间连接的模型	标准单元布图
电路级	开关级	描述器件中三极管和存储节点以及它们之间连接的模型	晶体管布图

1.3.2　系统级和算法级建模

系统级建模和算法级建模常用来从功能上描述系统的规格、仿真系统或核心算法的功能和特性。这类描述一般不涉及具体的实现细节，只是用 Verilog HDL 语言描述系统的功能，但不考虑是否能通过 EDA 工具将设计转化成硬件设计，因此往往将其称为系统级或者算法级描述。

虽然 Verilog HDL 语言具备系统级和算法级描述能力，但和 MATLAB 或 C++等高级语言相比仍存在很大的差距，因此在实际开发中，设计人员很少应用 Verilog HDL 语言的系统级和算法级建模能力。

1.3.3　RTL 级建模

RTL 级建模，也属于行为级描述范畴，在描述电路的时候只关注寄存器本身，以及寄存器到寄存器之间的逻辑功能，而不在意寄存器和组合逻辑的实现细节。RTL 级描述最大的特点就在于 RTL 级描述是目前最高层次的可综合描述语言（关于可综合特性的讨论将在 1.5 节展开）。在 EDA 工具的帮助下，设计人员可以直接在 RTL 级进行电路设计，而无须从逻辑门电路（与门、或门和非门）的较低层次来设计电路。

在最终实现时，所有的设计都需要映射到门级电路上，RTL 级代码也不例外，只不过 RTL 代码通过 EDA 软件中的逻辑综合工具转化成设计网表，网表基本上由门电路组成。目前，RTL 级设计代码是 Verilog HDL 程序设计中最常用的设计层次，因此逻辑综合是设计中必不可缺的一部分。本书主要内容都是在围绕着 RTL 级层次的代码设计和验证展开的。

1.3.4　门级和开关级建模

门级建模和开关级建模都属于结构描述范畴，都是对电路结构的具体描述，分别把需要的逻辑门单元和 MOS 晶体管调出来，再用连线把这些基本单元连接起来构成电路。这两种结构化描述方式是简单且严格的，因为一方面只需要说明“某一个门电路或 MOS 管的某个端口”与“另一个门电路或 MOS 管的某个端口”相连；但另一方面这种建模方式要求设计者必须对基本门电路和 MOS 管的功能及连接方式熟悉，否则只要一个端口连错就会使整个模块无法工作。因此这两个层次的描述类似于汇编语言和机器语言，虽然精确，但十分耗时耗力。

目前，可编程逻辑门数已达百万、千万门，对于大规模设计在这两类较低层次上设计电

路，效率低下且非常容易出错；只有对于小规模的设计，特别是对性能要求非常高的设计，采用门级电路和开关级电路可以满足一些特殊要求。在大多数 Verilog HDL 程序开发中，基于这两个层次的设计方法已被彻底抛弃。

1.4 基于 Verilog HDL 语言的 CPLD/FPGA 开发流程

基于 Verilog HDL 的 CPLD/FPGA 的设计流程就是利用 EDA 开发软件和编程工具对 FPGA 芯片进行开发的过程，和基于 VHDL 语言的开发流程一样，如图 1-3 所示，包括电路设计、设计输入、功能仿真、综合优化、综合后仿真、实现、布线后仿真、板级仿真以及芯片编程与调试等主要步骤。

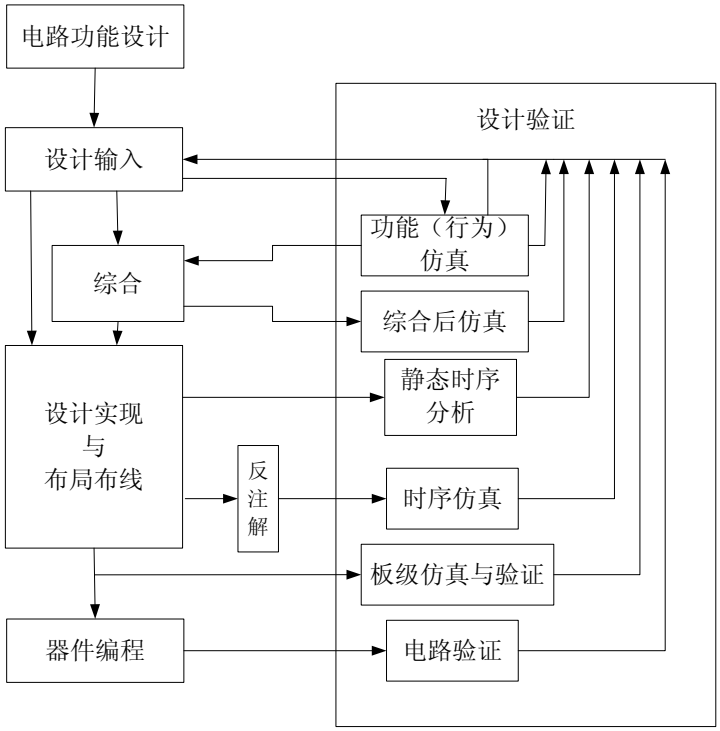


图 1-3　FPGA 开发的一般流程

1．电路功能设计

在系统设计之前，首先要进行的是方案论证、系统设计和 FPGA 芯片选择等准备工作。系统工程师根据任务要求，如系统的指标和复杂度，对工作速度和芯片本身的各种资源、成本等方面进行权衡，选择合理的设计方案和合适的器件类型。一般都采用自顶向下的设计方法，把系统分成若干个基本单元，然后再把每个基本单元划分为下一层次的基本单元，一直这样做下去，直到可以直接使用 EDA 元件库为止。

2．设计输入

设计输入是将所设计的系统或电路以开发软件要求的某种形式表示出来，并输入给 EDA 工具的过程。常用的方法有硬件描述语言（HDL）和原理图输入方法等。原理图输入方式是一种最直接的描述方式，在可编程芯片发展的早期应用比较广泛，它将所需的器件从元件库中调出来，画出原理图。这种方法虽然直观并易于仿真，但效率很低，且不易维护，不利于模块构造和重用。更主要的缺点是可移植性差，当芯片升级后，所有的原理图都需要作一定的改动。目前，在实际开发中应用最广的就是 HDL 语言输入法，利用文本描述设计，可以

分为普通 HDL 和行为 HDL。普通 HDL 有 ABEL、CUR 等，支持逻辑方程、真值表和状态机等表达方式，主要用于简单的小型设计。而在中大型工程中，主要使用行为 HDL，其主流语言是 Verilog HDL 和 VHDL。这两种语言都是美国电气与电子工程师协会（IEEE）的标准，其共同的突出特点有：语言与芯片工艺无关，利于自顶向下设计，便于模块的划分与移植，可移植性好，具有很强的逻辑描述和仿真功能，而且输入效率很高。

3．功能仿真

功能仿真，也称为前仿真，是在编译之前对用户所设计的电路进行逻辑功能验证，此时的仿真没有延迟信息，仅对初步的功能进行检测。仿真前，要先利用波形编辑器和 HDL 等建立波形文件和测试向量（即将所关心的输入信号组合成序列），仿真结果将会生成报告文件和输出信号波形，从中便可以观察各个节点信号的变化。如果发现错误，则返回设计修改逻辑设计。常用的工具有 Model Tech 公司的 ModelSim、Sysnopsys 公司的 VCS 和 Cadence 公司的 NC-Verilog 以及 NC-VHDL 等软件。虽然功能仿真不是 FPGA 开发过程中的必需步骤，但却是系统设计中最关键的一步。

4．综合

所谓综合就是将较高级抽象层次的描述转化成较低层次的描述。综合优化根据目标与要求优化所生成的逻辑连接，使层次设计平面化，供 FPGA 布局布线软件进行实现。就目前的层次来看，综合优化（Synthesis）是指将设计输入编译成由与门、或门、非门、RAM、触发器等基本逻辑单元组成的逻辑连接网表，而并非真实的门级电路。真实具体的门级电路需要利用 FPGA 制造商的布局布线功能，根据综合后生成的标准门级结构网表来产生。为了能转换成标准的门级结构网表，HDL 程序的编写必须符合特定综合器所要求的风格。由于门级结构、RTL 级的 HDL 程序的综合是很成熟的技术，所有的综合器都可以支持到这一级别的综合。常用的综合工具有 Synplicity 公司的 Synplify/Synplify Pro 软件以及各个 FPGA 厂家自己推出的综合开发工具。

5．综合后仿真

综合后仿真检查综合结果是否和原设计一致。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，可估计门延时带来的影响。但这一步骤不能估计线延时，因此和布线后的实际情况还有一定的差距，并不十分准确。目前的综合工具较为成熟，对于一般的设计可以省略这一步，但如果在布局布线后发现电路结构和设计意图不符，则需要回溯到综合后仿真来确认问题之所在。在功能仿真中介绍的软件工具一般都支持综合后仿真。

6．实现与布局布线

实现是将综合生成的逻辑网表配置到具体的 FPGA 芯片上，布局布线是其中最重要的过程。布局将逻辑网表中的硬件原语和底层单元合理地配置到芯片内部的固有硬件结构上，并且往往需要在速度最优和面积最优之间作出选择。布线根据布局的拓扑结构，利用芯片内部的各种连线资源，合理正确地连接各个元件。目前，FPGA 的结构非常复杂，特别是在有时序约束条件时，需要利用时序驱动的引擎进行布局布线。布线结束后，软件工具会自动生成报告，提供有关设计中各部分资源的使用情况。由于只有 FPGA 芯片生产商对芯片结构最为了解，所以布局布线必须选择芯片开发商提供的工具。

7．时序仿真与验证

时序仿真，也称为后仿真，是指将布局布线的延时信息反标注到设计网表中来检测有无时序违规（即不满足时序约束条件或器件固有的时序规则，如建立时间、保持时间等）现象。

时序仿真包含的延迟信息最全，也最精确，能较好地反映芯片的实际工作情况。由于不同芯片的内部延时不一样，不同的布局布线方案也给延时带来不同的影响。因此在布局布线后，通过对系统和各个模块进行时序仿真，分析其时序关系，估计系统性能，以及检查和消除竞争冒险是非常有必要的。在功能仿真中介绍的软件工具一般都支持综合后仿真。

8．板级仿真与验证

板级仿真主要应用于高速电路设计中，对高速系统的信号完整性、电磁干扰等特征进行分析，一般都以第三方工具进行仿真和验证。

9．芯片编程与调试

设计的最后一步就是芯片编程与调试。芯片编程是指产生使用的数据文件（位数据流文件，Bitstream Generation），然后将编程数据下载到 FPGA 芯片中。其中，芯片编程需要满足一定的条件，如编程电压、编程时序和编程算法等方面。逻辑分析仪（Logic Analyzer，LA）是 FPGA 设计的主要调试工具，但需要引出大量的测试管脚，且 LA 价格昂贵。目前，主流的可编程器件提供商都提供了内嵌的在线逻辑分析仪（如 Xilinx ISE 中的 ChipScope）来解决上述矛盾，它们只需要占用芯片少量的逻辑资源，具有很高的实用价值。

1.5 Verilog HDL 语言的可综合与仿真特性

说明：本节内容是全书的难点之一，也是 Verilog 设计的关键核心之一。正因为如此重要，本书将其安排在 Verilog HDL 内容之前，让读者首先从概念上对其进行了解。第一次阅读可能不能完全体会本节内容的含义，因此需要在学习过程中多次返回完成重复阅读。关于语句是否可综合的深入讨论将在第 5 章展开。

1.5.1 Verilog HDL 语句的可综合性说明

1．语句可综合的概念

综合就是将 HDL 语言设计转化为由与门、或门和非门等基本逻辑单元组成的门级连接。因此，可综合语句就是能够通过 EDA 工具自动转化成硬件逻辑的语句。

读者首先须要明确的是，HDL 语言并不是针对硬件设计而开发的语言，只不过目前被设计人员用来设计硬件。这是因为 HDL 语言只是硬件描述语言，并不是“硬件设计语言(Hardware Design Language)”，换句话说任何符合 HDL 语法标准的代码都是对硬件行为的一种描述，但不一定是可直接对应成电路的设计信息。

如 1.4 节所述，行为描述可以基于不同的层次，如系统级，算法级，寄存器传输级(RTL)、门级等等。以目前大部分 EDA 软件的综合能力来说，只有 RTL 或更低层次的行为描述才能保证是可综合的。

例如要实现两个变量相除的运算时，在代码中写下 C=A/B 这样的语句，可以发现在功能中，该句话可以正确执行，但任何 EDA 软件都不能将其综合成硬件电路。读者可以思考其中的原因，在计算除法时，需要从高位到低位逐次试除、求余、移位，需要经过多次运算才能得到最终结果；此外试除和求余需要减法器，商数和余数的中间结果必须有寄存器存储；显然这么多的计算不可能在一个时钟周期里完成，需要经过多次反复操作才能完成。因此，C=A/B 这样的语句相对于 EDA 软件的能力，显得太抽象，因而无法将其转化为硬件逻辑。

2．如何判断语句是否可综合？

在实际设计中，EDA 工具的综合结果以及相应的综合报告是判断语句是否可综合的最根本标准。但在操作中，不可能每句代码编写后都通过运行 EDA 工具来检测是否可综合。因此需要设计者在设计时就要判断所书写的语句是否可综合。

其实在综合判断方面，设计者的判断是要远远强于 EDA 工具的，也就是说计算机永远没有人聪明。对于一段代码，如果设计者本身都不能想象出一个较直观的硬件实现方法，那 EDA 软件肯定也不行。例如加法器、多路选择器是大家都很熟悉的电路，所以类似 A+B-C，(A>B)?C:D 这样的运算一定可以综合。而除法、求平方根、对数以及三角函数等较复杂的运算，必须通过一定的算法实现，并没有直观简单的实现电路，因而可以判断那些计算式是不能综合的，必须按其相应的算法写出更具体的代码才能实现。此外，硬件无法支持的行为描述，当然也不能被综合（例如在 FPGA 内部实现 DDR SDRAM 存储器那样的双延触发逻辑就是不可综合的）。

当然，这样直观的判断有时候是不准确的，因此最终都需要经过 EDA 工具的检验。不过正确判断代码是否可综合是 Verilog HDL 开发人员必须掌握的一项基本功能。当读者通过本书的学习后可以较准确判断代码可综合性的时候，说明已基本理解并掌握了 Verilog HDL 程序设计的本质。

1.5.2 Verilog HDL 语句的仿真特性说明

语句的仿真特性是相对语句的可综合特性而言的，但二者并不独立和相对。二者的区别在于可综合语句可用于仿真，而专门面向仿真语句虽然可以描述任何电路行为，但却是不可综合的。由于 Verilog HDL 语言最初就是为了完成仿真而发明的，从语法数量讲，可综合的语句只是 Verilog HDL 语言中的一个较小的子集。不过从用户开发上讲，可综合设计是最重要的，只有可综合设计才能将用户的“idea”最终实现在硬件平台上，所有仿真语句都是为了验证可综合设计而存在的。

从概念上讲，用最精简的语句描述最复杂的硬件是硬件描述语言的本质，但不可综合的仿真语句同样重要。在实际中利用 Verilog HDL 语言完成开发时，就是为了得到一个满足要求的、可综合的程序。但如何得到一个满足功能的正确程序呢？在设计阶段只能通过仿真测试才能得到。因此，设计人员在开发功能模块时，需要了解外围电路特点以及二者之间是否能够协调工作？这就要求开发外围电路的 Verilog HDL 代码。然而外围电路的 Verilog HDL 代码和功能模块的 Verilog HDL 代码有着本质区别，这是因为在实际中，外围电路是客观存在的，无须重新设计，设计人员只要将其模拟出来完成功能模块的测试即可，因此可以用不可综合的语句（专门用于仿真的语句）来实现，而不必管它是否能在硬件平台上实现。

在目前的电子开发行业中，Verilog HDL 开发工作也因上述原因分为两类：一类是可综合的功能模块开发；另一类就是专门用于测试的仿真模块开发，都有着广阔的应用领域。对于单个设计人员来讲，在编写 Verilog HDL 程序时，首先应该明确代码是用于仿真的还是综合的，要是用来综合的话，就必须要严格地使用可综合的语句；要是代码仅用来完成仿真测试，则非常灵活，不必在意硬件实现，可以使用 Verilog HDL 语言的所有语句，只要达到所要求的行为即可。.

1.6 本章小结

本章作为全书的起始，主要阐述目前先进的设计方法以及 Verilog HDL 语言的开发优势，共分为 5 个部分。首先介绍了 EDA 技术，该方法以计算机的软、硬件为基本工作平台，通过专用的应用软件帮助设计人员完成电路功能设计、逻辑设计、性能分析、时序测试以及印刷电路板（PCB）的制作，包括了数字系统的各个方面；其基本特征就是采用硬件描述语言来设计硬件系统。其次，介绍了 Verilog HDL 语言的历史和能力，并和 VHDL 和 C 语言等其他的软、硬件语言进行比较，得出 Verilog HDL 语言功能强大、简单好学等特点。第三，说明了 Verilog HDL 语言的描述层级，从上往下依次包括：系统级、算法级、RTL 级、门级和开关级，其中 RTL 级描述层次是使用最广泛的层次。第四，阐述了基于 Verilog HDL 语言的可编程逻辑器件开发方法，依次为设计输入、功能仿真、综合、实现、时序仿真（布局布线后仿真）、器件编程以及板级调试等步骤。最后介绍了硬件描述语言的仿真和综合特点，只有可综合设计才能开发硬件设计；而仿真语句只是用于验证可综合的设计的。

1.7 思考题

1．什么是 EDA 技术？ EDA 技术的基本特征是什么？常用的 EDA 软件包括哪些？
2．什么是可编程逻辑器件？其和传统的专用集成电路相比有什么特点？
3．硬件描述语言开发方法的优势包括哪些？
4．目前主流的硬件描述语言包括哪些，各自有什么特点？
5．Verilog HDL 语言包括几级描述层次，各层次的特点是什么？
6．简要描述基于 Verilog HDL 语言开发可编程逻辑器件的流程？
7．Verilog HDL 语言的可综合特点的具体含义是什么？
8. Verilog HDL 语言的可仿真特点的具体含义是什么？与可综合设计相比，各自有什么区别？

第 2 章　Verilog HDL 基础与开发平台操作指南

Verilog HDL 语言作为一种用于开发数字系统的硬件描述语言，其理论基础就是逻辑代数，其开发过程是一个从抽象到实际的转化。目前的设计规模已达到数百万乃至千万门，其复杂度已远非一个设计人员独立完成，在此情况下设计模式和设计方法是至关重要的。本书内容围绕着 Xilinx 可编程逻辑器件开发展开，因此本章以 Spartan 3E 系列 FPGA 为例介绍 Xilinx FPGA 的结构，给出 Xilinx 公司的 EDA 软件工具 ISE 10.1 版本以及 Mentor Garaphic 公司的验证工具 ModelSim 软件的快速入门操作。

2.1 Verilog HDL 程序开发的必备知识

2.1.1　数字的表示形式

在数字逻辑系统中，只存在高电平和低电平，因此用其表示数字只有整数形式，并存在 3 种表示方法，即：原码表示法（符号加绝对值）、反码表示法（符号加反码）和补码表示法（符号加补码）。这三种在 FPGA 开发中都有着广泛的应用，下面分别讨论。

1．原码表示法

原码表示法是机器数的一种简单的表示法，采用符号位级联绝对值的方法来表示数字。其最高位为符号位，用 0 表示正数，1 表示复数；其余部分为绝对数值部分。原码一般用二进制形式表示，如式 2.1 所示。

$$x = (-1)^{a_0} \sum_{i=1}^{B} \alpha_i 2^{-i} \tag{2-1}$$

例如，X1= +1010110，X2= -1001010，则其原码分别为：01010110 和 11001010

原码表示数的范围与二进制位数有关。当用 8 位二进制来表示小数原码时，其表示范围：最大值为 0.1111111，其真值约为 10 进制中的 0.99；最小值为 1.1111111，其真值约为十进制种的-0.99。当用 8 位二进制来表示整数原码时，其表示范围：最大值为 01111111，其真值为十进制的 127；最小值为 11111111，其真值为十进制的-127。

在原码表示法中，对 0 有两种表示形式，分别记为+0 和-0，以 8 比特数据为例，其相应的表示为：+0=00000000、-0=10000000。

2．反码表示法

反码可由原码得到。如果数字是正数，则其反码与原码一样；如果数字是负数，则其反码是对它的原码（符号位除外）各位取反而得到的（除符号位外，所有的 0 改为 1，所有的 1 改为 0）。

例如：X1=+1010110，X2= -1001010，则其相应的反码为 01010110、10110101。

3．补码表示法

补码表示法是实际中应用最广泛的数字表示法，其表示规则如下：若是正数，补码、反码和原码的表示是一样的；若是负数，补码、反码和原码的表示都是不一样的。补码的十进制可以表示为：

$$x = -a_0 + \sum_{1}^{B} \alpha_i 2^{-i} \tag{2-2}$$

实现负数的补码表示有两步：

- 取负数的绝对值，按照原码表示为 $\hat{x}$ 。
- 从 $\hat{x}$ 的最右位向左开始，找出二进制码为“1”的第一位，从第 1 位（不含）向左余下的位数取其补即可得到补码。

可以经过推导得到负数的反码和原码之间的简单换算关系：负数的补码等于其反码在最低位加 1，等效于直接用 2^{N+1} 减去其绝对值，因此补码也被称为“2”的补。

4．各类表示方法小结

原码的优点就是乘除运算方便，不论正负数，乘除运算都一样，并以符号位决定结果的正负号；若做加法则需要判断两数符号是否相同；若作减法，还需要判断两数绝对值的大小，以使大数减小数。

补码的优点是，加法运算方便，不论数的正负都可直接相加，而符号位同样参加运算，如果符号位发生进位，把进位的 1 去掉，余下的即为结果。

例 2-1：给出各类码字表示法的基本加法运算实例，并说明各自特点。

（1）首先给出原码的运算示例，其中$()_{10}$代表十进制数。首先给出一个原码的减法计算实例，完成“1+（-1）=0”的操作。

$$(1)_{10} + (-1)_{10} = (0)_{10} \tag{2-3}$$

如果读者直接利用原码来完成上式的计算，会就发现用带符号位的原码进行在加减运算的时候就出现了问题，将数据以 8 比特的表示形式为例，如式（2-4）所示：

$$(00000001)原 + (10000001)原 = (10000010)原 = (-2)_{10} \tag{2-4}$$

式（2-4）的计算结果是不对的，问题在于两点：首先，负数的符号位直接改变了计算结果的符号；其次，绝对值部分计算也不正确。这说明原码无法直接完成正数和负数的加法。事实上，对于原码表示的两个正数，其计算结果也会变成负数，这一特性，希望读者自行验证。

（2）既然，原码不能完成正、负数相加，那么反码形式可以完成此操作吗？仍然以“1+（-1）=0”为例，其相应的反码表达式如式（2-5）所示。

$$(00000001)\ 反 + (11111110)反 = (11111111)反 = (\ -0\ )_{10} \tag{2-5}$$

则发现问题出现在(+0)和(-0)上，因为在实际的计算中的零是没有正负之分的。但式（2-5）标明了反码完成正、负数相加后，其绝对值部分是正确的，因此能正确完成正、负数相加的表达形式必定包含反码的特性。

（3）最后给出补码的相关特性说明，负数的补码就是对反码加一，而正数不变。以八比特数据为例，通过 $(-128)_{10}$代替了$(-0)_{10}$，所以其表示范围为[-128,127]。从直观上，补码消除了(+0)和(-0)，并且具备反码特点，那么究竟其能否完成正、负数的加法运算呢？答案是肯定的，下面给出具体实例，如式（2-6）所示。

$$(00000001)补 + (11111111)补 = (00000000)补 = (0)_{10} \tag{2-6}$$

基于以上讨论，可以得到一个基本结论：只有补码才能正确完成正、负数的加法运算，并将减法运算转化为加法运算，从而简化运算规则。

但对于乘法操作，则以原码形式计算是最为方便的，请读者自行验证。在实际的系统设计中，经常需要完成数字各类表达形式的转化，是数字系统的设计基础，读者一定要掌握本

部分内容。

2.1.2 常用术语解释

1．时钟

在电子技术中，如果脉冲信号是一个按一定电压幅度，一定时间间隔连续发出的脉冲信号，则该脉冲信号被称为周期信号。脉冲信号之间的时间间隔称为周期；而将在单位时间（如 1 秒）内所产生的脉冲个数称为频率。频率是描述周期性循环信号（包括脉冲信号）在单位时间内所出现的脉冲数量多少的计量名称；频率的标准计量单位是 Hz（赫兹）。时钟信号本身就是一种周期信号，只是专门用来驱动电路设计。常见的时钟信号波形包括：方波、锯齿波波和正弦波等，其典型的时域波形如图 2-1 所示。

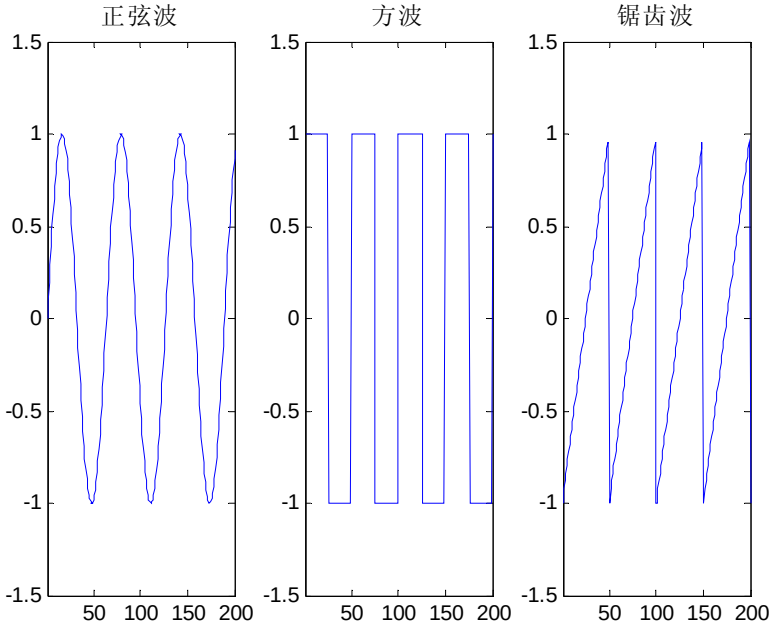


图 2-1 常见的时钟信号波形图

时钟占空比（Duty Cycle）的含义如下：在一串理想的脉冲序列中（如方波），正脉冲的持续时间与脉冲总周期的比值。在图 2-2 所示的时钟信号波形中，其中脉冲宽度 1us，信号周期 4us，则其占空比为 0.25。

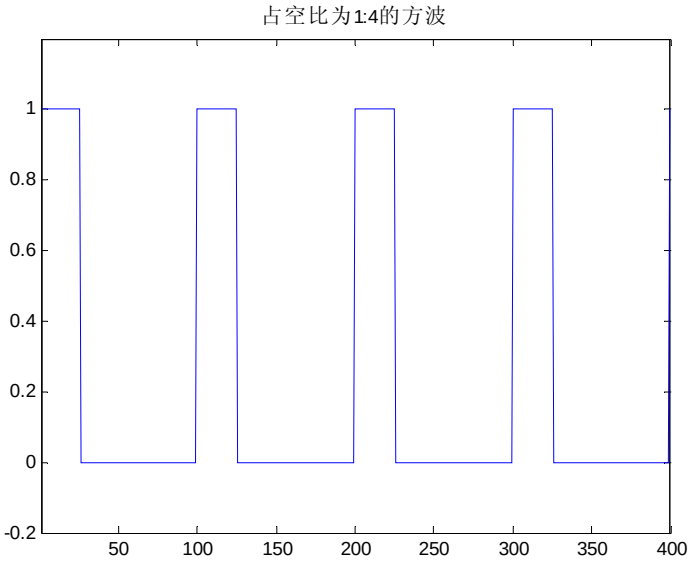


图 2-2　1:4 的占空比时钟示意图

2．资源

Verilog HDL 语言是用来在硬件平台上（PLD 或者 ASIC）来完成数字系统开发的，所谓的资源指的是所用到的硬件平台的规模大小，最通用的评断标准就是逻辑门数（MOS 管的数目）。例如，“目前电路规模越来越大，已达千万门”就意味着该芯片内部集成了超过一千万个 COMS 管。

不同的可编程逻辑器件厂家对各自芯片的资源定义也是不一样的，一般有两种方法对其进程粗略评估。（1）第一，把 FPGA 基本单元（如 LUT+FF、ESB/BRAM）和实现相同功能的标准门阵列比较，门阵列中包含的门数即为该 FPGA 基本单元的等效门数，然后乘以基本单元的数目就可以得到 FPGA 门数估计值；（2）第二，分别用 FPGA 和标准门阵列实现相同的功能，从中统计出 FPGA 的等效门数，这种方法比较多地依赖于经验数据。详细的比较可以查阅相关集成开发的 Help 菜单。但门数是一种等效概念，大约需要 7 个门来实现一个 D 触发器，而一个门即可实现一个 2 输入与非门。其它 RAM 等都可以进行等效。但是准确地评估规模不用门数，而用基本可编程配置单元的数量。对于 Xilinx 器件，一个底层可编程单元 Slice 包含 2 个触发器（FF）和 2 个查找表（LUT）；对于 Altera 器件，一个底层可编程单元 LE 包含 1 个触发器（FF）和 1 个查找表（LUT）。所以对于这两个厂家的器件，如果 Xilinx 有 1 万个 Slice，其规模就相当于 Altera 有 2 万个 LE 的芯片。

2.1.2　Verilog HDL 程序的优劣判断指标

Verilog HDL 程序设计首要指标是功能的完备性，达到设计要求，这是任何设计都必须完成的。其次，还包括“面积”、“速度”和功耗指标，是设计的深层次要求。从实用角度来讲，后者的重要性并不亚于功能完整性。在设计中，“面积”、“速度”和功耗之间并不是相互独立的，可以相互转换。下面对上述三个指标进行简单介绍。

1．面积性能

这里的“面积”主要是指设计所占用的 FPGA 逻辑资源数目，利用所消耗的触发器（FF）、查找表（LUT）以及各类嵌入式硬核来衡量。由于 FPGA 芯片的逻辑资源数量是有

限的，只有设计所需的各类逻辑资源都小于芯片的最大值，才能将其运行在 FPGA 芯片中。因此，面积性能是 FPGA 设计的最基本指标。

2．时序性能

“速度”是指在芯片上稳定运行时所能够达到的最高频率。不同设计在同一芯片上所能运行的最高频率是不一样的。

面积和速度是一对对立和统一的矛盾体。一方面，要提高速度，就需要消耗更多的资源，即需要更大的面积；另一方面，为了减少面积，就需要降低处理速度。所以既要提高速度，又要减少面积，是不可能同时实现的。但在实际中，总是存在二者之间的平衡，也意味着二者可以互换。面积和速度互换的具体操作很多，比如模块复用、乒乓操作、串并变换以及流水线操作。在 Xilinx 公司的设计软件 ISE 中，提供了多类辅助工具来帮助用户在面积和速度之间达到最佳的平衡。面积和速度这两个指标始终贯穿着 FPGA 的设计，是评价设计质量的最终标准。

3．功耗性能

设计的功耗由两部分组成：静态功耗和动态功耗。前者是由静态电流引起的；后者是电路工作时消耗的功率。

其中，静态功耗主要由晶体管的泄漏电流引起，即晶体管即使在逻辑上被关断时，也会从源极“泄漏”到漏极或通过栅氧“泄漏”的小电流。这与设计代码无关，主要取决于芯片型号。

动态功耗是电路工作功耗，是当电路中的电压由于激励信号发生变化时消耗的功率，可分为两部分：翻转功耗和内部功耗。翻转功耗是指一个驱动元件在对负载电容进行充放电时消耗的功率，电路电压翻转越频繁，功耗就越大；内部功耗是在芯片内部晶体管电压发生翻转时由于瞬间导通而产生的功率，对于翻转率较慢的电路，这部分功耗会很显著。

动态功耗估算的基本方法是：（1）计算每个设计单元的功耗。（2）累加各个设计单元的功耗，常用以下计算公式：

$$P = \sum C \times V^2 \times E \times f \times 1000$$

式中，P 表示功耗，单位是 mW；C 表示电容，单位是 F；V 表示电压，单位是 V；E 表示翻转频率，指每个时钟周期的翻转次数；f 表示工作频率，单位是 Hz。可以看出工作频率越高，设计所造成的功耗就越大。

2.2 Verilog HDL 程序设计模式

2.2.1 自顶向下的设计模式

目前可编程逻辑器件已经发展到单芯片集成系统（System On Chip，SOC）阶段，相对于集成电路（IC）的设计思想有着革命性的变化。SOC 是一个复杂的系统，它将一个完整产品的功能集成在一个芯片上，包括核心处理器、存储单元、硬件加速单元以及众多的外部设备接口等，具有设计周期长、实现成本高等特点，因此其设计方法必然是自顶向下的从系统级到功能模块的软、硬件协同设计，达到软、硬件的无缝结合。

具体到 Verilog HDL 程序设计时，首先从系统级开始，把系统划分为若干个二级单元，并对其接口和资源进行评估，编制出相应的行为或结构模型；再将各个二级单元划分为下一

层次的基本单元，一直做下去，直到可以直接用可综合的 Verilog HDL 语句来实现为止，如图 2-3 所示。这就允许多个设计者同时设计一个硬件系统中的不同模块，并为自己所设计的模块负责。同时，也要求设计人员在开发之前，应当对设计的全貌有一定的预见性。

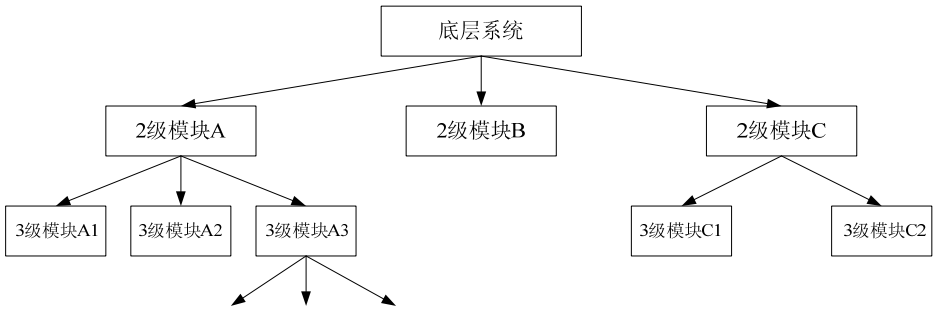


图 2-3　自顶向下的 FPGA 设计开发流程

目前，主流的可编程开发工具都提供了自顶向下的管理功能，可以有效地梳理错综复杂的层次，能够方便地查看某一层次模块的源代码以修改错误。

2.2.2　层次、模块化模式

在自顶向下的设计模式中，隐含着对硬件系统的层次化、结构化设计。在设计中的每一层都会有若干个功能相对独立的模块，该层次的硬件结构就由这些模块的互连描述得到。从图 2-3 中也可以看到，层次化就是纵向的系统划分，模块化则对应着横向的划分。

在工程实践中，还存在 EDA 软件编译时间过长的问题。由于大型设计包含多个复杂的功能模块，其时序收敛与仿真验证复杂度很高，为了满足时序指标的要求，往往需要反复修改源文件，再对所修改的新版本进行重新编译，直到满足要求为止。这里面存在两个问题：首先，对于大规模系统设计，EDA 软件编译一次需要长达数小时甚至数周的时间，这是设计人员所不能容忍的；其次，每次重新编译和布局布线后结果差异很大，会将已满足时序的电路破坏。因此必须提出一种能有效提高设计性能，继承已有结果，便于团队化设计的软件工具。因此 EDA 公司也意识到这类层次化设计的工具需求，并开发出了相应的逻辑锁定和增量设计的软件工具。例如，Xilinx 公司就推出了层次化设计辅助工具 PlanAhead。

PlanAhead 组件允许高层设计者为不同的模块划分相应 CPLD/FPGA 芯片区域，并允许底层设计者在在所给定的区域内独立地进行设计、实现和优化，等各个模块都正确后，再进行设计整合。如果在设计整合中出现错误，单独修改即可，不会影响到其它模块。PlanAhead 将结构化设计方法、团队化合作设计方法以及重用继承设计方法三者完美地结合在一起，有效地提高了设计效率，缩短了设计周期。

不过从以上描述可以看出，新型的设计方法对系统顶层设计师有很高的要求。在设计初期，他们不仅要评估每个子模块所消耗的资源，还需要给出相应的时序关系；在设计后期，需要根据底层模块的实现情况完成相应的修订。

2.2.3　IP 核的重用

1．IP 核的基本概念

IP（Intelligent Property）核是具有知识产权核的集成电路芯核总称，是经过反复验证过的、具有特定功能的宏模块，且该模块与芯片制造工艺无关，可以移植到不同的半导体工艺

中。到了 SOC 阶段，向用户提供 IP 核服务已成为可编程逻辑器件提供商的重要任务。对于可编程逻辑器件提供商来讲，其提供的 IP 核越丰富，用户的设计就越方便，其市场占用率就越高。目前，IP 核已经变成系统设计的基本单元，并作为独立设计成果被交换、转让和销售。

从 IP 核的提供方式上，通常将其分为软核、硬核和固核这 3 类。从完成 IP 核所花费的成本来讲，硬核代价最大；从使用灵活性来讲，软核的可复用使用性最高。

（1）软核

软核在 EDA 设计领域指的是综合之前的寄存器传输级（RTL）模型；具体在 FPGA 设计中指的是对电路的硬件语言描述，包括逻辑描述、网表和帮助文档等。软核只经过功能仿真，需要经过综合以及布局布线才能使用。其优点是灵活性高、可移植性强，允许用户自配置；缺点是对模块的预测性较低，在后续设计中存在发生错误的可能性，有一定的设计风险。软核是 IP 核应用最广泛的形式。

（2）固核

固核在 EDA 设计领域指的是带有平面规划信息的网表；具体在 FPGA 设计中可以看做带有布局规划的软核，通常以 RTL 代码和对应具体工艺网表的混合形式提供。将 RTL 描述结合具体标准单元库进行综合优化设计，形成门级网表，再通过布局布线工具即可使用。和软核相比，固核的设计灵活性稍差，但在可靠性上有较大提高。目前，固核也是 IP 核的主流形式之一。

（3）硬核

硬核在 EDA 设计领域指经过验证的设计版图；具体在 FPGA 设计中指布局和工艺固定、经过前端和后端验证的设计，设计人员不能对其修改。不能修改的原因有两个：首先是系统设计对各个模块的时序要求很严格，不允许打乱已有的物理版图；其次是保护知识产权的要求，不允许设计人员对其有任何改动。虽然，IP 硬核不允许用户修改特点对其复用造成了一定的困难，但 IP 硬核的性能最高，且不占用可编程芯片的逻辑资源，因此在实际中也获得了广泛使用。

2．IP 核重用的优势

IP 核是自下而上设计方法学的基础，也是大规模设计的构造基础。IP 的可重用特性不仅可以缩短 SoC 芯片的设计时间外，还能大大降低设计和制造的成本，提高可靠性。此外，随着电子设计种类的级数性增长，单个设计工程师，包括一个设计团队，都无法独立完成各类设计，从项目需求上，迫切需要各类丰富的 IP 核来满足系统设计的要求。目前，IP 核已形成了相当大的产业规模，读者应对该技术的原理和发展进行跟踪。本书 2.4.4 节将详细介绍如何在 ISE 中调用 Xilinx 公司提供的 IP 核。

2.3 Xilinx Spartan 3E 系列 FPGA 简介

由于 Spartan 3E 系列 FPGA 芯片既具备较高的性能，成本还比较低，性价比高，特别适合初学者作为入门练习的硬件平台。本书所有的实例均在 Xilinx 公司大学计划官方推荐的 Spartan 3E Starter 开发板上执行过。因此本节主要介绍 Spartan 3E 系列 FPGA 芯片的硬件结构，为后续的深入学习以及动手实践做好准备。

2.3.1 Spartan-3E 系列 FPGA 简介

Spartan-3E 是目前 Spartan 系列最新的中低端产品，具有多款芯片，系统门数范围从 10 万到 160 万。Spartan-3E 是在 Spartan-3 成功的基础上进一步改进的产品，提供了比 Spartan-3 更多的 I/O 端口和更低的单位成本，是 Xilinx 公司性价比最高的 FPGA 芯片。由于更好地利用了 90 *nm* 技术，在单位成本上实现了更多的功能和处理带宽，是 Xilinx 公司新的低成本产品代表，主要面向消费电子应用，如宽带无线接入、家庭网络接入以及数字电视设备等。其主要特点如下：

- 采用 90 *nm* 工艺；
- 大量用户 I/O 端口，最多可支持 376 个 I/O 端口或者 156 对差分端口；
- 端口电压为 3.3V、2.5Ⅴ、1.8V、1.5V、1.2V ；
- 单端端口的传输速率可以达到 622 Mbit/s，支持 DDR 接口；
- 最多可达 36 个18×18 的专用乘法器、648 kbits 块 RAM、231 kbits 分布式 RAM；
- 宽的时钟频率 5MHz~300MHz 以及多个专用片上数字时钟管理（DCM）模块。

Spartan-3E 系列产品的主要技术特征如表 2-1 所示。

表 2-1　　Spartan-3E 系列 FPGA 主要技术特征

型号	系统门数	SLICE 数目	分布式 RAM 容量	块 RAM 容量	专用乘法器数	DCM 数目	最大可用 I/O 数	最大差分 I/O 对数
XC3S100E	100k	960	15k	72k	4	2	108	40
XC3S250E	250k	2448	38k	216k	12	4	172	68
XC3S500E	500k	4656	73k	360k	20	4	232	92
XC3S1200E	1200k	8672	136k	504k	28	8	304	124
XC3S1600E	1500k	14752	231k	648k	36	8	376	156

2.3.2 Spartan-3E 系列 FPGA 结构说明

图 2-4 为 Xilinx 公司 Spartan-3E 系列 FPGA 的内部结构示意图（注：图 2-2 只是一个示意图，实际上每一个系列的 FPGA 都有其相应的内部结构，由于应用场合不同，所以内部结构会有局部不同），可以看出主要包括 IOBs、CLBs、DCM、BRAM、硬核乘法器以及连线资源。下面对其各功能模块进行简要介绍。

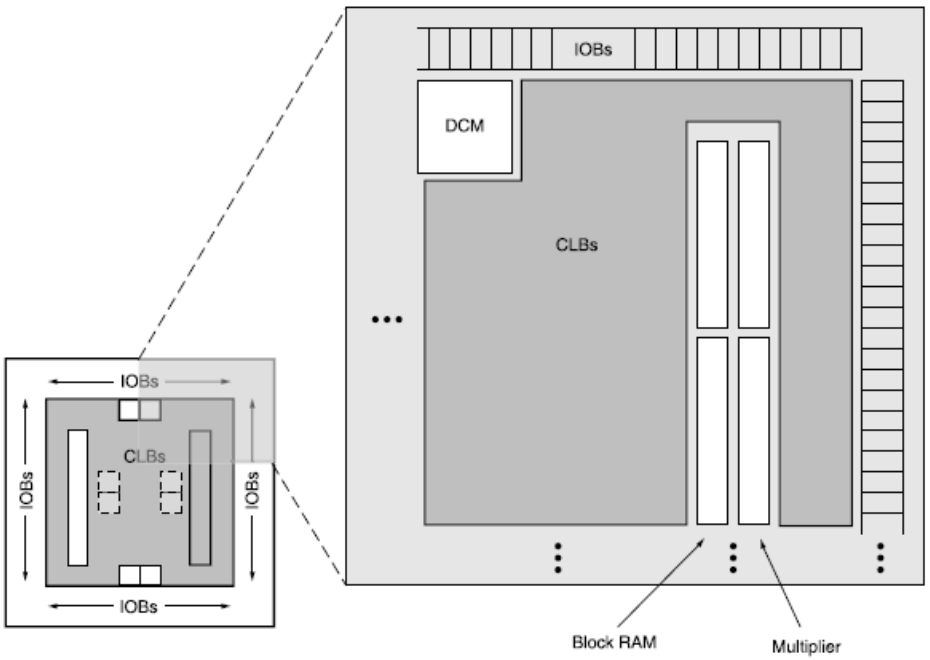


图 2-4　Spartan-3E 系列 FPGA 内部结构示意图

1．可编程输入输出单元（IOB）

可编程输入/输出单元简称 I/O 单元，是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配要求，其示意结构如图 2-5 所示。FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性，可以调整驱动电流的大小，可以改变上、下拉电阻。目前，I/O 口的频率也越来越高，一些高端的 FPGA 通过 DDR 寄存器技术可以支持高达 2Gbps 的数据速率。

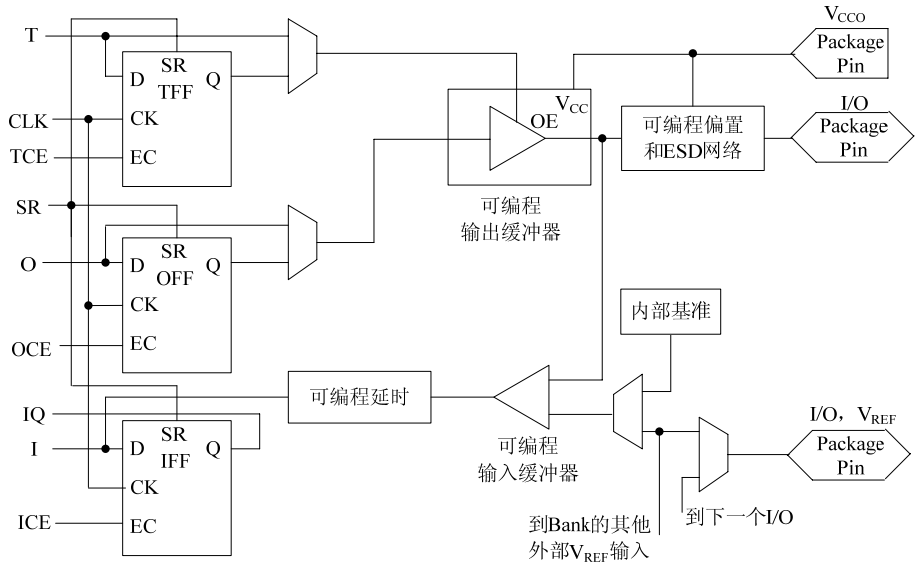


图 2-5　典型的 IOB 内部结构示意图

外部输入信号可以通过 IOB 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。当外部输入信号经过 IOB 模块的存储单元输入到 FPGA 内部时，其保持时间（Hold Time）的要求可以降低，通常默认为 0。

为了便于管理和适应多种电器标准，FPGA 的 IOB 被划分为若干个组（bank），每个 bank 的接口标准由其接口电压 V_{CCO} 决定，一个 bank 只能有一种 V_{CCO}，但不同 bank 的 V_{CCO} 可以相同。只有相同电气标准的端口才能连接在一起，V_{CCO} 电压相同是接口标准的基本条件。

2． 可配置逻辑块（CLB）

CLB 是 FPGA 内的基本逻辑单元。CLB 的实际数量和特性会依器件的不同而不同，但是每个 CLB 都包含一个可配置开关矩阵，此矩阵由 4 或 6 个输入、一些选型电路（多路复用器等）和触发器组成。开关矩阵是高度灵活的，可以对其进行配置以便处理组合逻辑、移位寄存器或 RAM。

Spartan3E 的每个 CLB 包含 4 个 Slice，如图 2-6 所示。其中，左边的一对 Slice 被称为 SliceEM，包含分布式 RAM 和逻辑；右边的一对被称为 SliceL，仅包含逻辑。CLB 具有输入函数配置，每个 CLB 可实现 16:1 多路复用器。CLB 内部通过开关矩阵进行的普通布线，实现了局部布线最优化，再加上不同 CLB 之间的快速直接布线，可实现全局布线的最优化。

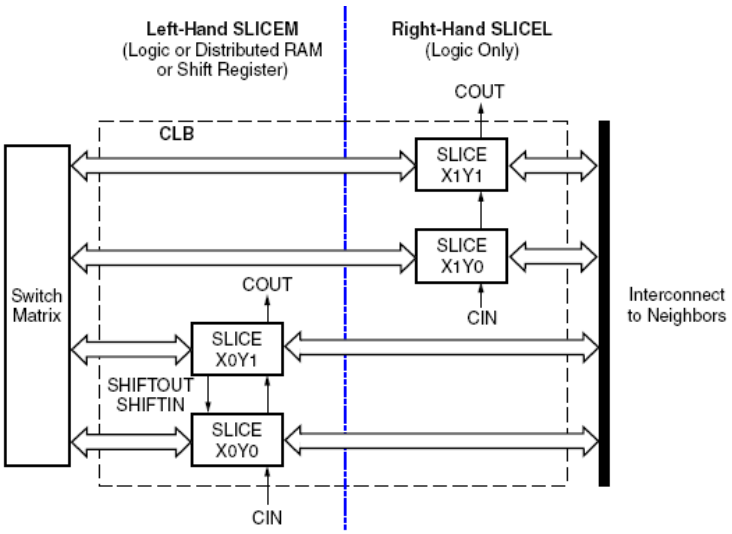


图 2-6　Spartan3E 芯片的 CLB 内部结构

Slice 是 Xilinx 公司定义的基本逻辑单位，其内部结构如图 2-7 所示，一个 Slice 包含两个 4 输入的 LUT、两个存储逻辑以及附加逻辑（进位链、多路选择器）等。算术逻辑包括一个异或门（XORG）和一个专用与门（MULTAND），一个异或门可以使一个 Slice 实现 2bit 全加操作，专用与门用于提高乘法器的效率；进位逻辑由专用进位信号和函数复用器（MUXC）组成，用于实现快速的算术加减法操作；4 输入函数发生器用于实现 4 输入 LUT、分布式 RAM 或 16 比特移位寄存器（目前，基于 65*nm* 工艺的 FPGA 一般都采用 6 输入查找表，可以实现 6 输入 LUT 或 64 比特移位寄存器）；进位逻辑包括两条快速进位链，用于提高 CLB 模块的处理速度。

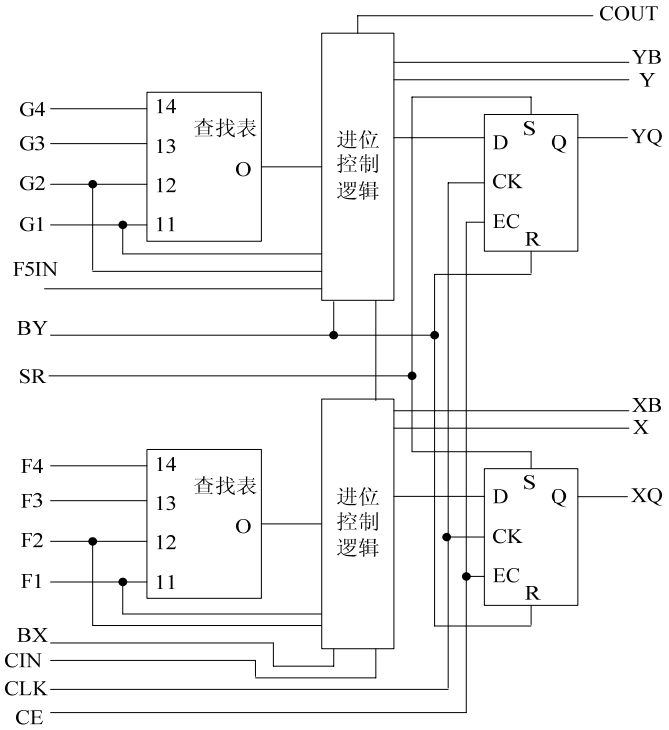


图 2-7　典型的 4 输入 Slice 结构示意图

3．数字时钟管理模块（DCM）

业内大多数 FPGA 均提供数字时钟管理（Xilinx 的全部 FPGA 均具有这种特性）。Xilinx 推出最先进的 FPGA 提供数字时钟管理和相位环路锁定。相位环路锁定能够提供精确的时钟综合，且能够降低抖动，并实现过滤功能。

4．嵌入式块 RAM（BRAM）

大多数 FPGA 都具有内嵌的块 RAM，这大大拓展了 FPGA 的应用范围和灵活性。块 RAM 可被配置为单端口 RAM、双端口 RAM、内容地址存储器（CAM）以及 FIFO 等常用存储结构。RAM、FIFO 是比较普及的概念，在此就不冗述。CAM 存储器在其内部的每个存储单元中都有一个比较逻辑，写入 CAM 中的数据会和内部的每一个数据进行比较，并返回与端口数据相同的所有数据的地址，因而在路由的地址交换器中有广泛的应用。除了块 RAM，还可以将 FPGA 中的 LUT 灵活地配置成 RAM、ROM 和 FIFO 等结构。在实际应用中，芯片内部块 RAM 的数量也是选择芯片的一个重要因素。

单片块 RAM 的容量为 18*k* 比特，即位宽为 18 比特、深度为 1024，可以根据需要改变其位宽和深度，但要满足两个原则：首先，修改后的容量（位宽×深度）不能大于 18*k* 比特；其次，位宽最大不能超过 36 比特。当然，可以将多片块 RAM 级联起来形成更大的 RAM，此时只受限于芯片内块 RAM 的数量，而不再受上面两条原则约束。

5．硬核乘法器

硬核是相对底层嵌入的软核而言的，每一个硬核本身就等效于一个 ASIC 电路。为了提高 FPGA 的处理能力并降低功耗，Xilinx 公司在 Spartan 3E 内部集成了数目不等的硬核乘法器。

6．丰富的布线资源

布线资源连通 FPGA 内部的所有单元，而连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。FPGA 芯片内部有着丰富的布线资源，根据工艺、长度、宽度和分布位置的不同而划分为 4 类不同的类别。第一类是全局布线资源，用于芯片内部全局时钟和全局复位/置位的布线；第二类是长线资源，用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线；第三类是短线资源，用于完成基本逻辑单元之间的逻辑互连和布线；第四类是分布式的布线资源，用于专有时钟、复位等控制信号线。

在实际中设计者不需要直接选择布线资源，布局布线器可自动地根据输入逻辑网表的拓扑结构和约束条件选择布线资源来连通各个模块单元。从本质上讲，布线资源的使用方法和设计的结果有密切、直接的关系。

2.4 ISE 快速入门

2.4.1 ISE 操作基础

1．ISE 简介

Xilinx 是全球领先的可编程逻辑完整解决方案的供应商，研发、制造并销售应用范围广泛的高级集成电路、软件设计工具以及定义系统级功能的 IP（Intellectual Property）核，长期以来一直推动着可编程逻辑器件和 EDA 技术的发展。Xilinx 的开发工具也在不断地升级，由早期的 Foundation 系列逐步发展到目前的 ISE 11.x 系列，集成了 EDA 开发需要的所有功能，其主要特点有：

- 全面支持 Virtex-5 系列器件；
- 提供了 Xilinx 新型 SmartCompile 技术，可以将实现时间缩减 2.5 倍，能在最短的时间内提供最高的性能，提供了一个功能强大的设计收敛环境；
- SmartXplorer 技术可并行完成 FPGA 设计，利用分布式处理和多层实施策略来提高设计性能；
- 添加了基于策略的设计方法；
- 映射、布局布线过程都支持两种模式：Performance Evaluation Mode 和 Non Timing Driven Mode；
- 在一个工程中可以添加多个用户约束文件（UCF），可为不同等级的各个模块灵活添加各类约束；
- 自动生成 Tcl 脚本。

ISE Foundation 具有界面友好、操作简单的特点，再加上 Xilinx 的 CPLD/FPGA 芯片占有很大的市场，使其成为非常通用的 EDA 工具软件。此外，ISE 作为接口开放的 EDA 设计工具集合，与第三方软件扬长补短，使软件功能越来越强大，为用户提供了更加丰富的 Xilinx 开发平台。

2．ISE 安装

ISE Foundation 10.1 软件支持 Microsoft Windows XP、Microsoft Windows Vista、Red Hat Enterprise Linux 4/5 以及 SUSE Linux Enterprise 10 等多类操作系统。下面以 Windows 系统为例介绍其具体安装过程：

（1）将光盘放进 DVD 光驱，等待其自动运行（如果没有自动运行，直接执行光盘目录下的 Setup.exe 文件程序即可），会弹出图 2-8 所示的欢迎界面，点击“Next”进入下一页。

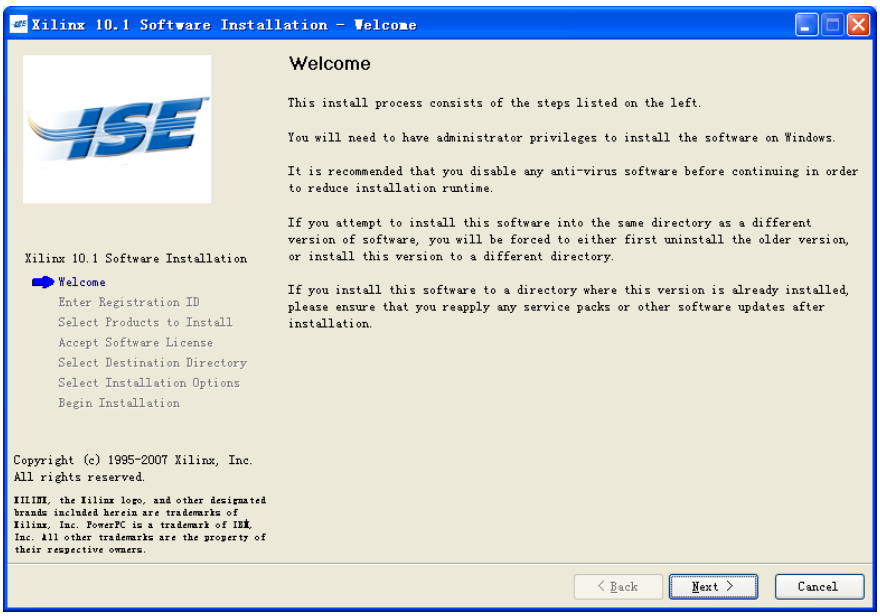


图 2-8　ISE10.1 安装过程的欢迎界面

（2）接着进入注册码获取、输入对话框，如图 2-9 所示。注册码可以通过网站、邮件和传真方式申请注册码。如果已有注册码，输入后单击“Next”按键后继续。

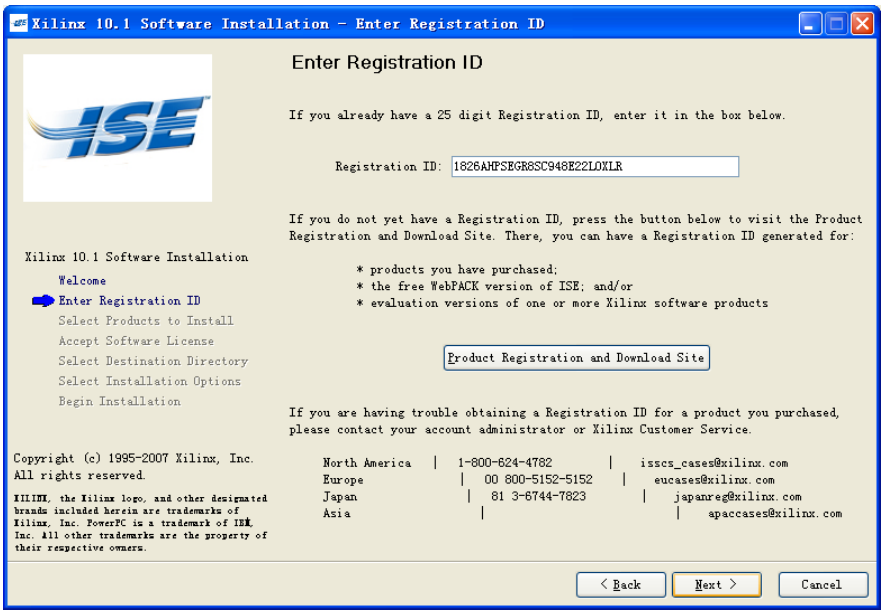


图 2-9　ISE 10.1 安装程序的注册码输入界面

购买了正版软件后，最常用的方法就是通过网站注册获取安装所需的注册码。首先在 Xilinx 的官方主页 www.xilinx.com 上建立自己的帐号，点击图 2-7 中的“Product Registration and Download Site”登陆，输入 CD 盒上的产品序列号，就会自动生成 25 位的注册码，直接记录下来即可，同时 Xilinx 网站会将注册码的详细信息发送到帐号所对应的邮箱中。如果安装试用版，无需产品序列号，直接申请注册码即可，并可全功能试用 60 天。

（3）接下来的三个对话框是 Xilinx 软件的产品列表和授权声明对话框，直接选中“I accept the terms of this software license”，单击“Next”后进入安装路径选择界面，如图 2-10

所示。单击“Browse”按键后选择自定义安装路径，单击“Next”按键继续。

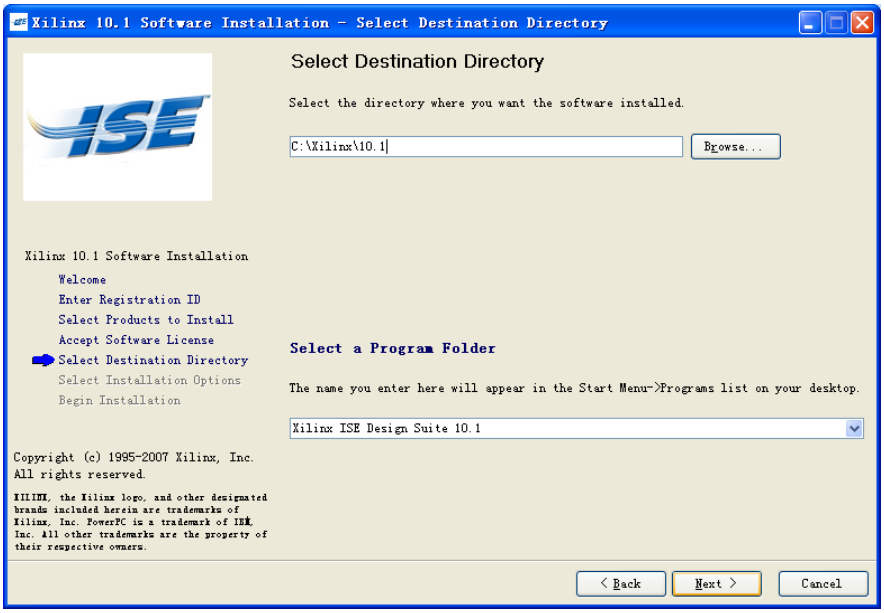


图 2-10　ISE 软件安装路径选择对话框

（4）接下来的几个对话框分别是安装组件选择，如图 2-11 所示，用户需要选择自己使用的芯片所对应的模块，这样才能在开发中使用这些模块。在计算机硬盘资源不紧张的情况下，通常选择“Select All”。

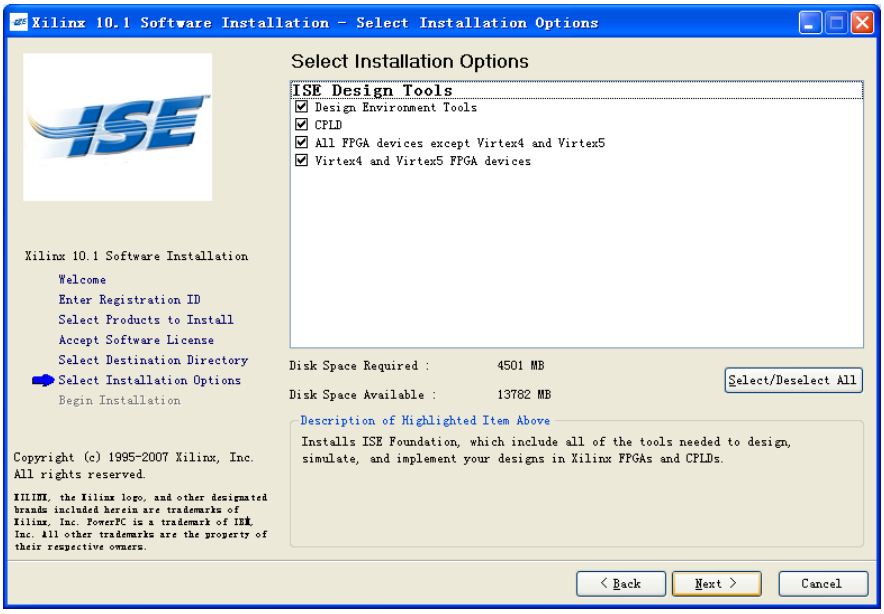


图 2-11　ISE 安装组件选择界面

（5）随后进入设置环境变量页面，保持默认即可。如果环境变量设置错误，则安装后不能正常启动 ISE。选择默认选项，安装完成后，在“我的电脑”上单击右键，选择“属性→环境变量”，可看到名为“Xilinx”的环境变量，其值为安装路径。最后进入安装确认对话框，单击“Install”按钮，即可按照用户的设置自动安装 ISE，如图 2-12 所示。

图 2-12　ISE 安装进程示意图

（6）安装完成后，会弹出图 2-13 所示的对话框，点击“OK”按键完成安装。安装程序会在桌面以及程序菜单中添加 Project Navigator 的快捷方式，双击即可进入 ISE 集成开发环境。

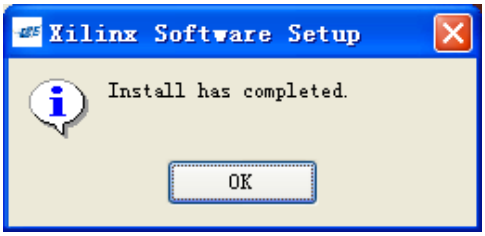


图 2-13　ISE 安装完成提示框

3．ISE 用户界面

ISE10.1i 的界面如图 2-14 所示，由上到下主要分为标题栏、菜单栏、工具栏、工程管理区、源文件编辑区、过程管理区、信息显示区、状态栏等 8 部分。

- 标题栏：主要显示当前工程的路径、名称以及当前打开的文件名称。
- 菜单栏：主要包括文件（File）、编辑（Edit）、视图（View）、工程（Project）、源文件（Source）、操作（Process）、窗口（Window）和帮助（Help）等 8 个下拉菜单。其使用方法和常用的 Windows 软件类似。

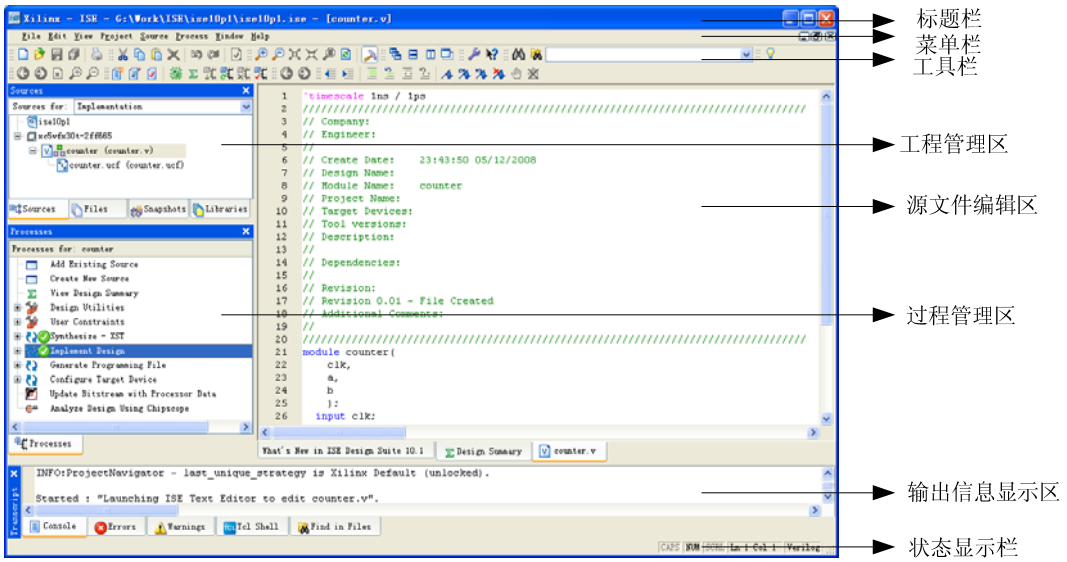


图 2-14　ISE 的主界面

● 工具栏：主要包含了常用命令的快捷按钮。灵活运用工具栏可以极大地方便用户在 ISE 中的操作。在工程管理中，此工具栏的运用极为频繁。

● 工程管理区：提供工程及其相关文件的显示和管理功能，主要包括源代码页面（Sources），文件页面（Files）、快照视图（Snapshots）和库视图（Librarys）。其中源代码视图比较常用，显示了源代码的层次关系。文件页面是 10.1 新添加的，列出了工程包含的所有文件。快照是当前工程的备份，设计人员可以随时备份，也可以将当前工程随时恢复到某个备份状态；快照视图用于查看当前工程的快照，执行该功能的方法是选择菜单项“Project | Take Snapshot”。库视图则显示了工程中用户产生的库内容。

● 源文件编辑区：源文件编辑区提供了源代码的编辑功能。

● 过程管理区：本窗口显示的内容取决于工程管理区中所选定的文件。相关操作和 FPGA 设计流程紧密相关，包括设计输入、综合、仿真、实现和生成配置文件等。对某个文件进行了相应的处理后，在处理步骤的前面会出现一个图标来表示该步骤的状态。

● 信息显示区：显示 ISE 中的处理信息，如操作步骤信息、警告信息和错误信息等。信息显示区的下脚有两个标签，分别对应控制台信息区（Console）和文件查找区（Find in Files）。如果设计出现了警告和错误，双击信息显示区的警告和错误标志，就能自动切换到源代码出错的地方。

● 状态显示栏：显示相关命令和操作的信息，并指示 ISE 软件目前所处的状态。

2.4.2　新建工程

首先打开 ISE，每次启动时 ISE 都会默认恢复到最近使用过的工程界面。当第一次使用时，由于此时还没有过去的工程记录，所以工程管理区显示空白。选择“File | New Project”选项，在弹出的新建工程对话框中的工程名称中输入“mycounter”。在工程路径中单击 Browse 按键，当工程放到指定目录，如图 2-15 所示。

然后点击“Next”进入下一页，选择所使用的芯片类型以及综合、仿真工具。计算机上所安装的所有用于仿真和综合的第三方 EDA 工具都可以在下拉菜单中找到，如图 2-16 所示。

在图中，我们选用了 Virtex 5-FX30T 芯片，并且指定综合工具为 XST（VHDL/Verilog），仿真工具为 ISE Simulator（VHDL/Verilog）。

再点击“Next”进入下一页，可以选择新建源代码文件，也可以直接跳过，进入下一页。第 4 页用于添加已有的代码，如果没有源代码，点击“Next”，进入最后一页，单击确认后，就可以建立一个完整的工程。

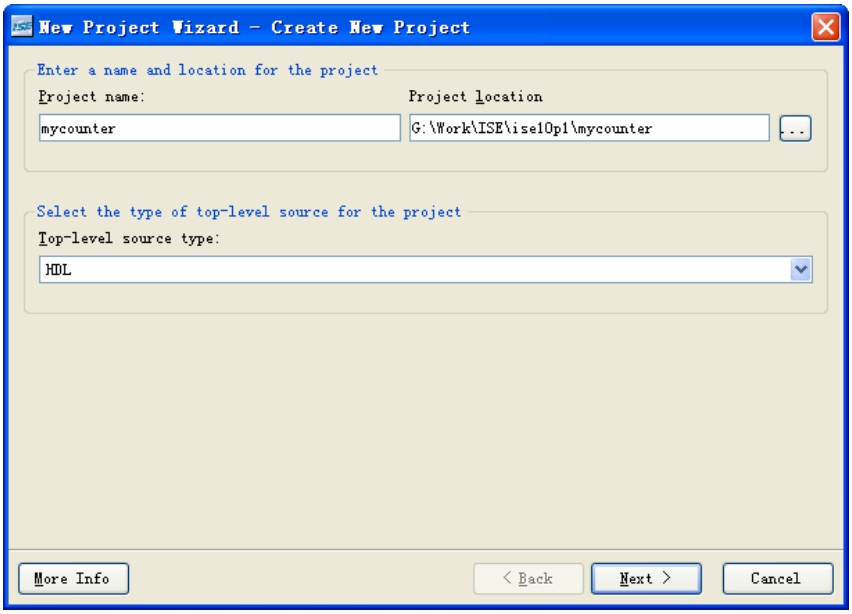


图 2-15　利用 ISE 新建工程的示意图

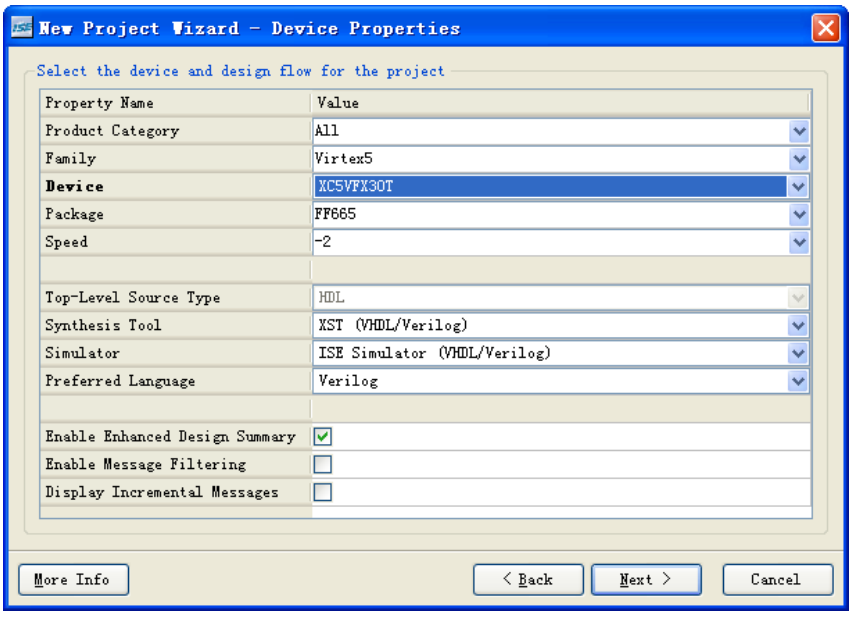


图 2-16　新建工程器件属性配置表

2.4.3　Verilog HDL 代码的输入与功能仿真

在工程管理区任意位置单击鼠标右键，在弹出的菜单中选择“New Source”命令，会弹出如图 2-17 所示的 New Source 对话框。对于逻辑设计，最常用的输入方式就是 HDL 代码输入法（Verilog Module、VHDL Module）、状态机输入法（State Diagram）和原理图输入法

（Schematic）。

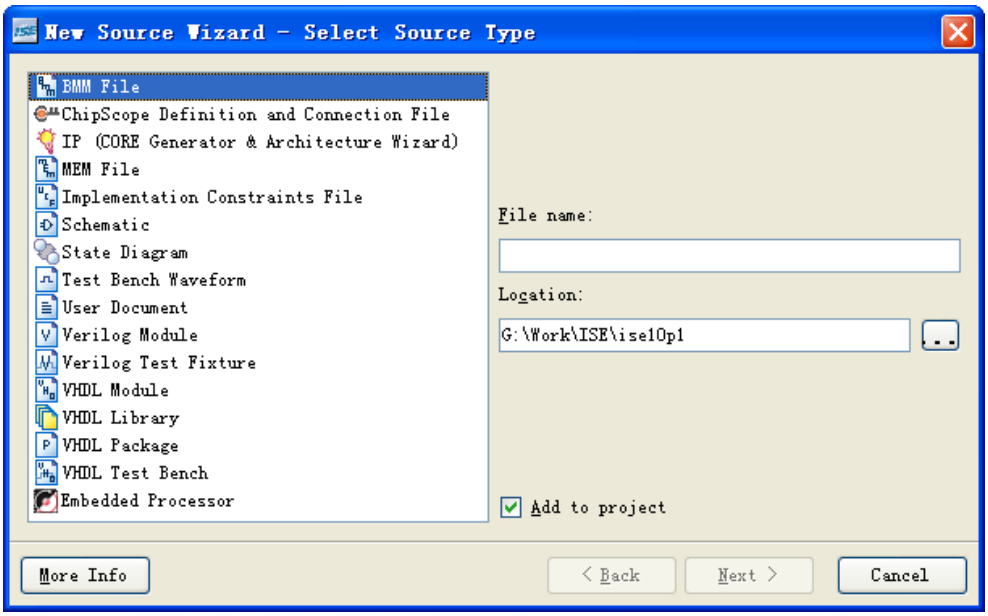


图 2-17　新建源代码对话框

左侧的列表用于选择代码的类型，各项的意义如下所示：

【BMM File】：块存储器映射（Block Memory Map）文件，用于将单个的块 RAM 连成一个更大容量的存储逻辑单元。

【ChipScope Definition and Connection File】：在线逻辑分析仪 ChipScope 文件类型，具有独特的优势和强大的功能，将在第 5 章进行讨论。

【IP (Coregen & Architecture Wizard)】：由 ISE 的 IP Core 生成工具快速生成可靠的源代码，这是目前最流行、最快速的一种设计方法。

【MEM File】：存储器定义（Memory Define）文件，用于定义 RAMB4 和 RAMB16 存储单元的内容。注意：一个工程只能包含一个 MEM 文件。

【Implementation Constraints File】：约束文件类型，可添加时序和位置约束。

【State Disgram】：状态图类型。

【Test Bench Wavaform】：测试波形类型。

【User Document】：用户文档类型。

【Verilog Module】：Verilog 模块类型，用于编写 Verilog 代码。

【Verilog Test Fixture】：Verilog 测试模块类型，专门编写 Verilog 测试代码。

【VHDL Module】：VHDL 模块类型，用于编写 VHDL 代码。

【VHDL Library】：VHDL 库类型，用于制作 VHDL 库。

【VHDL Packet】：VHDL 包类型，用于制作 VHDL 包。

【VHDL Test Bench】：VHDL 测试模块类型，专门编写 VHDL 测试代码。

由以上参数可以看出，HDL 代码输入法是目前最主要的 FPGA 设计方法，拥有广泛的用户群，主要包括 Verilog HDL 和 VHDL 两大类。二者同属 HDL 语言范畴，且可相互调用。本书所有实例以 Verilog HDL 语言为主进行说明。

单击“New Source”命令，在代码类型中选择 Verilog Module 选项，在 File Name 文本

框中输入“mycounter”，单击“Next”进入端口定义对话框，如图 2-18 所示。

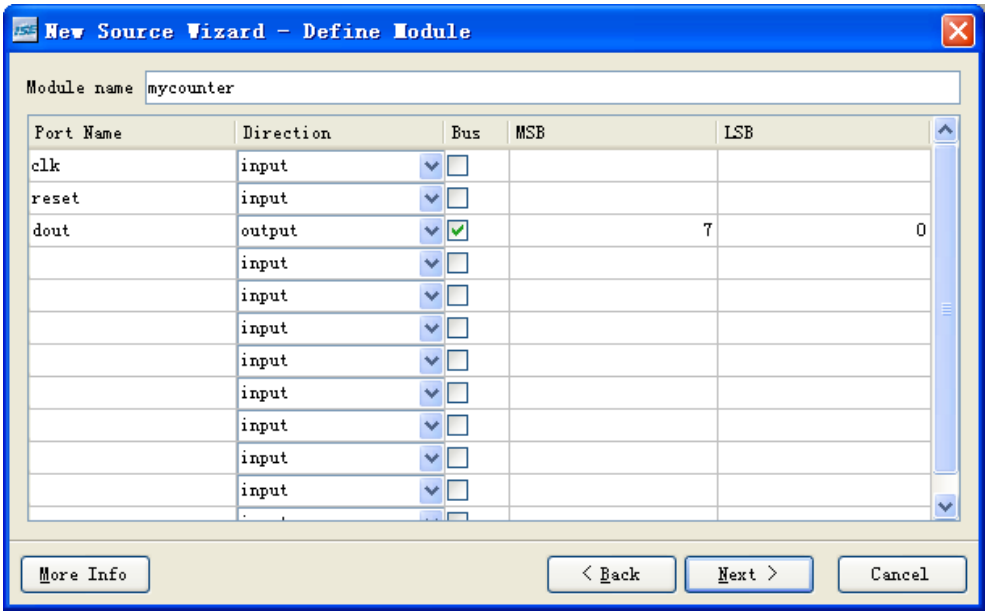


图 2-18　Verilog 模块端口定义对话框

其中 Module Name 就是输入的“mycounter”，下面的列表框用于对端口的定义。“Port Name”表示端口名称，“Direction”表示端口方向（可以选择为 input、output 或 inout），MSB 表示信号的最高位，LSB 表示信号的最低位，对于单位信号的 MSB 和 LSB 不用填写。

定义了模块端口后，单击“Next”进入下一步，点击“Finish”按键完成创建。这样，ISE 会自动创建一个 Verilog 模块的例子，并且在源代码编辑区内打开。简单的注释、模块和端口定义已经自动生成，所剩余的工作就是在模块中实现代码。

例 2-2：利用 Verilog 代码实现 8 比特计数器。

```
module mycounter(
    clk, reset, dout
    );
    input clk;
    input reset;
    output [7:0] dout;

    // 以下为手工添加的代码
    reg     [7:0] dout;
     always @(posedge clk) begin
        if (reset == 0)
            dout <= 0;
         else
            dout <= dout + 1;
     end
```

endmodule

在 ISE10.1 版本中，增强了代码编辑器的功能，将鼠标光标移动到一对“( )”或者“[]”任一部分的后面，则该对括号以红色显示，便于用户查看是否配对。

3．测试代码输入

下面介绍基于 Verilog 语言建立测试平台的方法。首先在工程管理区将“Sources for”设置为 Behavioral Simulation，在任意位置单击鼠标右键，并在弹出的菜单中选择“New Source”命令，然后选中“Verilog Test Fixture”类型，输入文件名为“tb_ mycounter”，再点击“Next”进入下一页。这时，工程中所有 Verilog Module 的名称都会显示出来，设计人员需要选择要进行测试的模块。用鼠标选中 mycounter，点击“Next”后进入下一页，直接点击“Finish”按键，ISE 会在源代码编辑区自动显示测试模块的代码：

```
`timescale 1ns / 1ps
module tb_ mycounter;

    // 定义输入变量
    reg   clk;
    reg   reset;
    // Outputs
    wire [7:0] dout;

    // 例化被测试模块(UUT)
    mycounter uut (
        .clk(clk),
        . reset (reset),
        .dout(dout)
    );

    initial begin
        // 初始化输入变量
        clk = 0;
        reset = 0;
        //全局复位 100 个仿真时间单位
        #100;
        // Add stimulus here
        reset = 1;
    end

    // 产生仿真所需的时钟信号，其周期为 10 个仿真时间单位
    always #5 clk = ~ clk;

endmodule
```

其中测试平台的整体架构是由 ISE 自动生成的，包括所需信号、端口声明以及模块调用的完成。所需的工作就是在 initial…end 模块中的“// Add stimulus here”后面添加测试向量生成代码。本例在其中添加了产生时钟信号的 always 语句。

完成测试平台后。在工程管理区将“Sources for”选项设置为 Behavioral Simulation，这时在过程管理区会显示与仿真有关的进程，如图 2-19 所示。

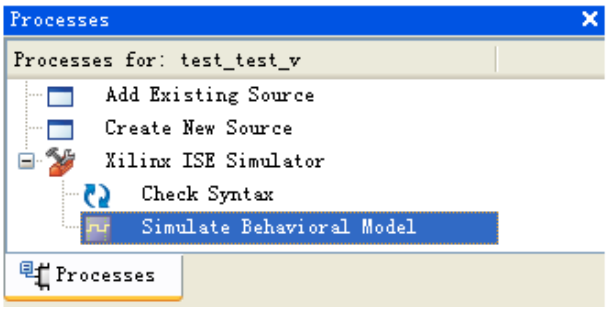


图 2-19　仿真过程示意图

选中图 2-19 中 Xilinx ISE Simulator 下的 Simulate Behavioral Model 项，点击鼠标右键，选择弹出菜单的 Properties 项，会弹出如图 2-20 所示的属性设置对话框，最后一行的 Simulation Run Time 就是仿真时间的设置，可将其修改为任意时长，本例采用默认值。

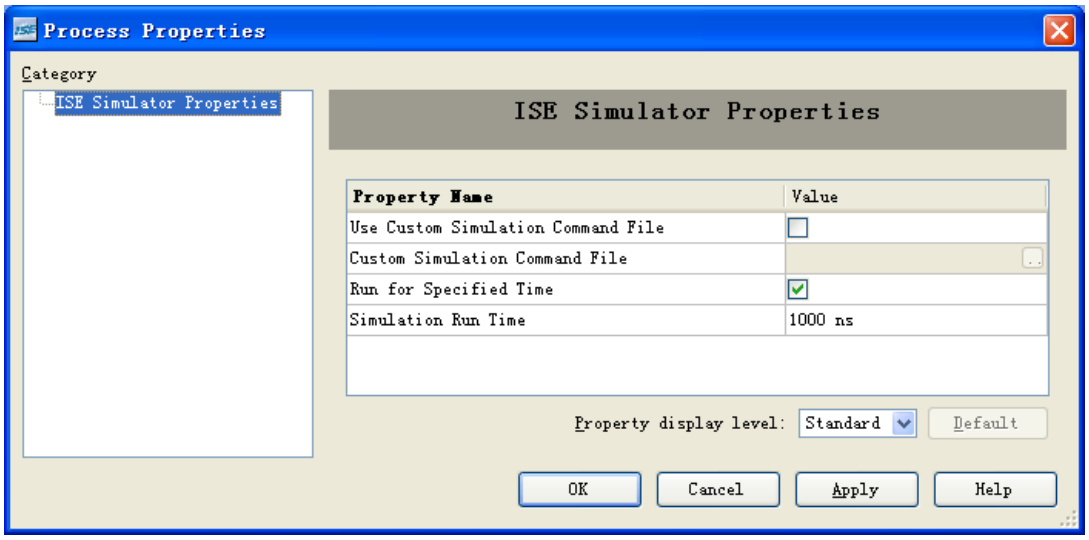


图 2-20　仿真属性设置对话框

仿真参数设置完后，就可以进行仿真。首先在工程管理区选中测试代码，然后在过程管理区双击 ISE Simulator 软件中的 Simulate Behavioral Model，则 ISE 会自动启动 ISE Simulator 软件，并得到如图 2-21 所示的仿真结果，从中可以看到设计达到了预计目标。

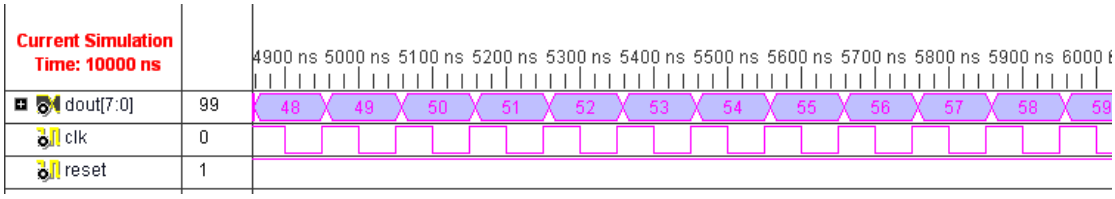


图 2-21　test 模块的仿真结果

2.4.4　Xilinx IP 核的使用

1．Xilinx IP Core 简介

IP Core 就是预先设计好、经过严格测试和优化过的电路功能模块，如乘法器、FIR 滤波器、PCI 接口等，并且一般采用参数可配置的结构，方便用户根据实际情况来调用这些模块。随着 FPGA 规模的增加，使用 IP Core 完成设计成为发展趋势。

IP Core 生成器（Core Generator）是基于 Xilinx 平台的 Verilog HDL 设计中的一个重要工具，提供了大量成熟的、高效的 IP Core 为用户所用，涵盖了汽车工业、基本单元、通信和网络、数字信号处理、FPGA 特点和设计、数学函数、记忆和存储单元、标准总线接口等 8 大类，从简单的基本设计模块到复杂的处理器一应俱全。配合 Xilinx 网站的 IP 中心使用，能够大幅度减轻设计人员的工作量，提高设计可靠性。

Core Generator 最重要的配置文件的后缀是.xco，既可以是输出文件又可以是输入文件，包含了当前工程的属性和 IP Core 的参数信息。

2．调用 IP core 的基本操作

启动 Core Generator 有两种方法，一种是在 ISE 中新建 IP 类型的源文件，另一种是双击运行“开始 → 程序 → Xilinx ISE Design Suit 10.1 → ISE → Accessories → Core Generator”。

Xilinx 公司提供了大量的、丰富的 IP Core 资源，究其本质可以分为两类：一是面向应用的，和芯片无关；还有一种用于调用 FPGA 底层的宏单元，和芯片型号密切相关。本节以调用和芯片结构无关的比较器为例来介绍第一种方法。

例 2-3：生成比较器的 IP 核，实现“>=”的逻辑判断，并通过 Verilog HDL 代码调用该 IP 核以及完成功能仿真。

（1）在 ISE 中新建工程，然后在工程管理区，单击右键，选择“New Source”命令，在文件类型中选择“IP（CORE Generator & Architecture Wizard）”，并在右端的“File Name”文本框输入 compare_demo，如图 2-22 所示。然后单击“Next”按键进入下一页。

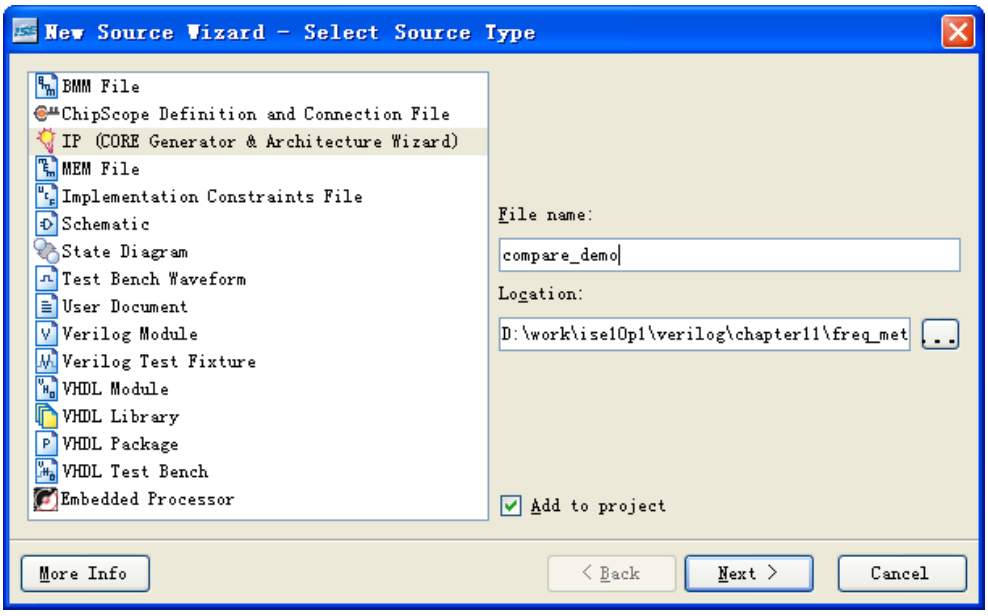


图 2-22　IP 核新建向导页面

（2）然后在弹出的“Select IP”页面，选择“Math Functions”类别下“Comparators”

分类中的“Comparator v9.0”，如图 2-23 所示。然后单击“Next”按键进入 IP 核向导的小结页面，列出了 IP 核的存储路径以及类型，如图 2-24 所示。如果确认无误，可以单击“Finish”按键，完成 IP 核的生成向导过程；如果参数有误，可单击“Back”按键，返回相应页面进行修改。

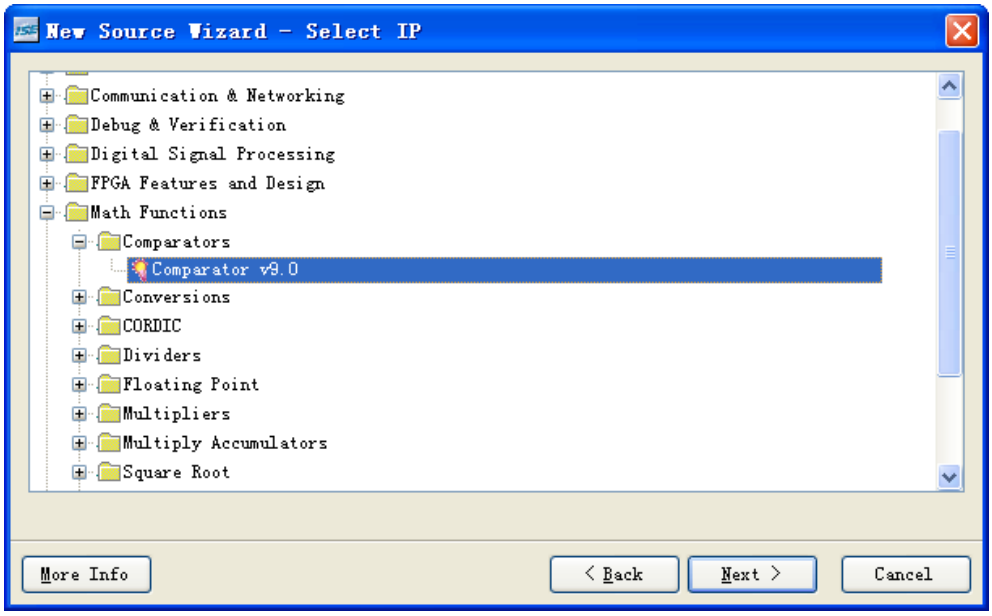


图 2-23　IP 核类型选择页面

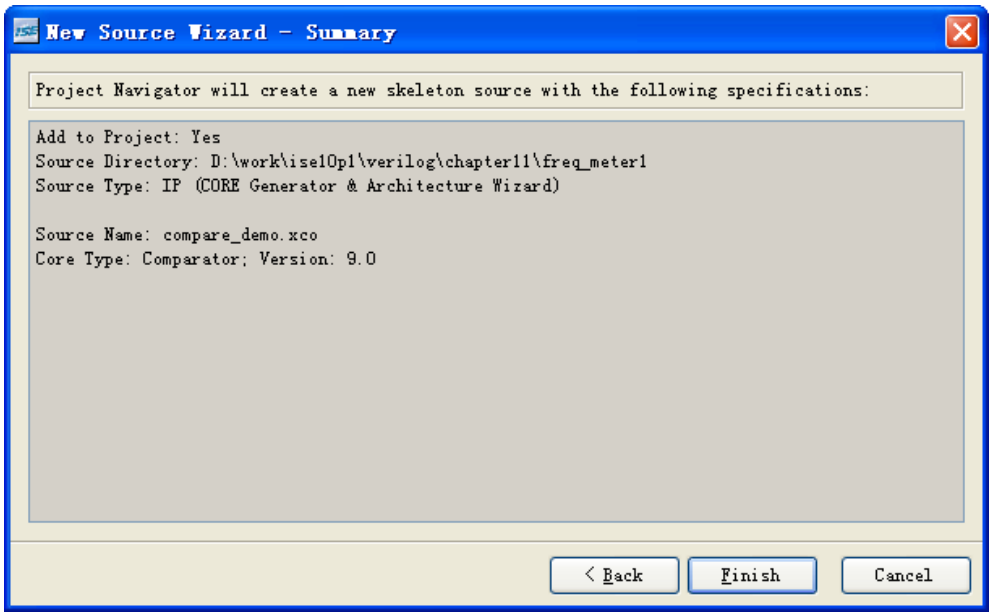


图 2-24　IP 核向导小结页面

（3）在弹出的确认页面中，点击“Finish”按键，ISE 会自动完成 IP 核文件生成过程，进入 IP 核配置页面，如图 2-25 所示。其中，“Component Name”栏列出了 IP 核的名称，就是用户在图 2-22 页面所输入的名称，这里不能再对其进行修改。“Operation”区域列出了比较器的所有操作，包括“=”、“<”、“<=”、“<>”、“>”、“>=”等 6 个操作，其中“<>”代表着不等判断逻辑。“Input Options”区域列出了两个输入端口 A、B 的数据类型，存在 Signed

和 Unsigned 两种选择，其中 Signed 意味着补码形式。“Input Width”区域用于设定输入数据的位宽，数值范围为 2~256；也可以将 B 端口的数据设置为常数。“Output Options”区域用于配置输出数据类型，有“Registered Output”和 “Non-Registered Output”两种选择，分别代表着比较结果输出寄存和直接输出。

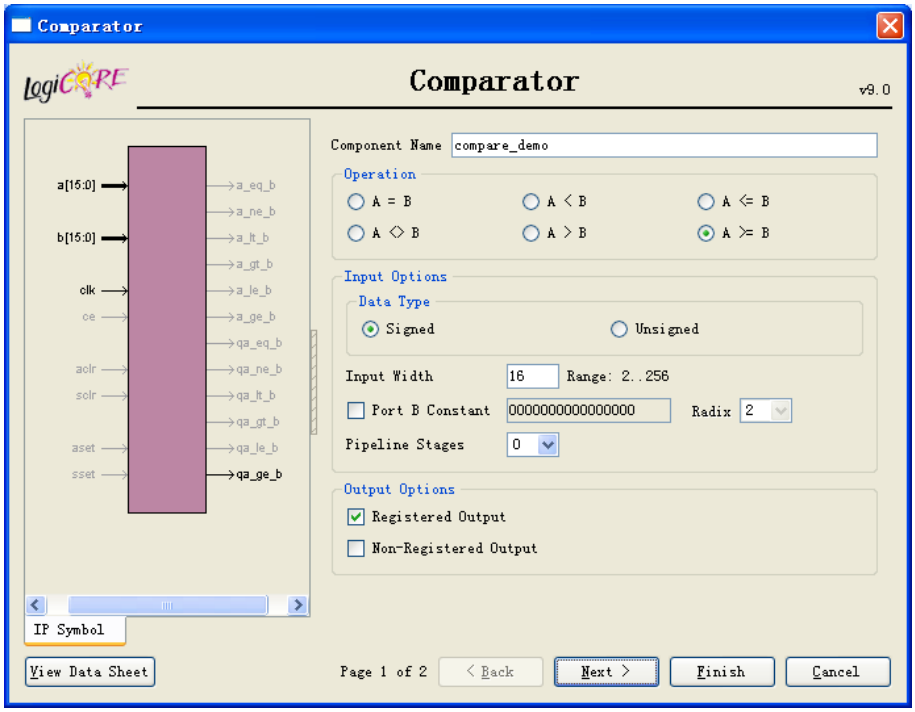


图 2-25　比较器 IP 核配置页面（1）

本例配置为“A>=B”，输入数据为有符号数，其位宽为 16 比特，并且将比较器结果输出寄存。配置完成后，单击“Next”按键进入下一页。

（4）比较器输出结果的第二页面为寄存器配置界面。其中“Clock Enable”区域用于添加时钟使能信号 CE，并且可以调整 CE 和同步控制信号的优先级。“Asynchronous Settings”区域用于设置异步参数，包括置位（Set）和清空（Clear）功能，以及上电后的复位电平。“Synchronous Settings” 区域用于设置同步参数，包括置位（Set）和清空（Clear）功能。

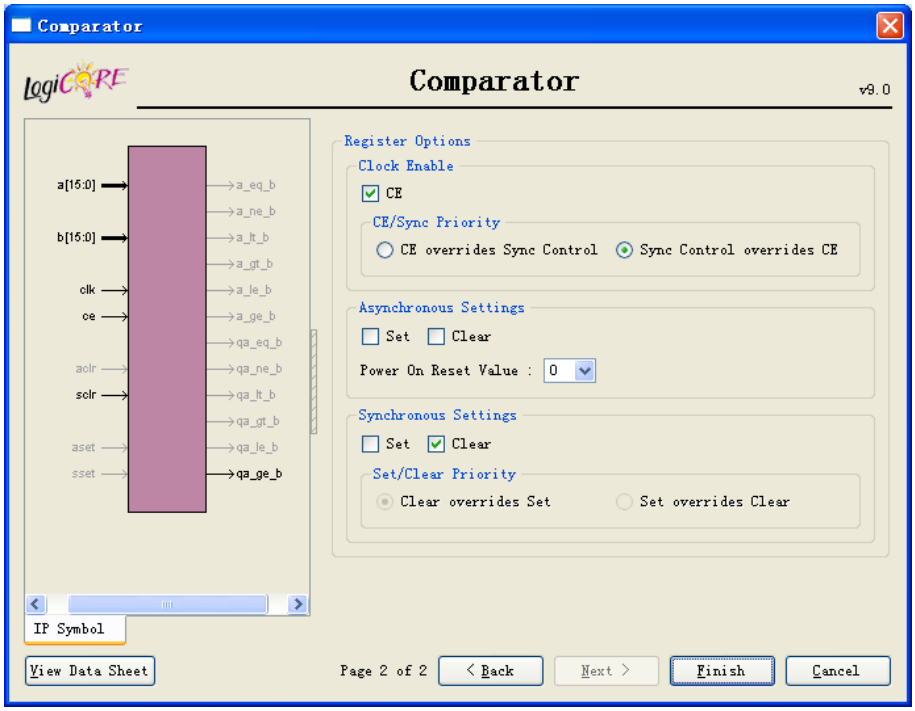


图 2-26　比较器 IP 核配置页面（2）

本例配置为添加时钟使能信号，以及同步清空功能。配置完成后，单击“Finish”按键，完成 IP 核的配置，ISE 会在输出信息显示区输出各类指示信息，包括 IP 核生成的指示信息，如下所列。

Generating IP...

WARNING:sim:217 - The chosen IP does not support a Verilog behavioral model, generating a Verilog structural model instead.Generating Implementation files.

Generating ISE symbol file...

WARNING:coreutil -　Default charset GBK not supported, using ISO-8859-1 instea

Generating NGC file.

Generating Verilog structural model.

Finished Generating.

Successfully generated compare_demo.

（5）比较器的 IP 核生成后，可以在工程管理区点击选中生成的比较器 IP 核，然后双击过程管理区“Core Generator”栏下的“View HDL Functional Model”命令，如图 2-27 所示，则可以在源文件 编辑区察看比较器的代码，如下所列。

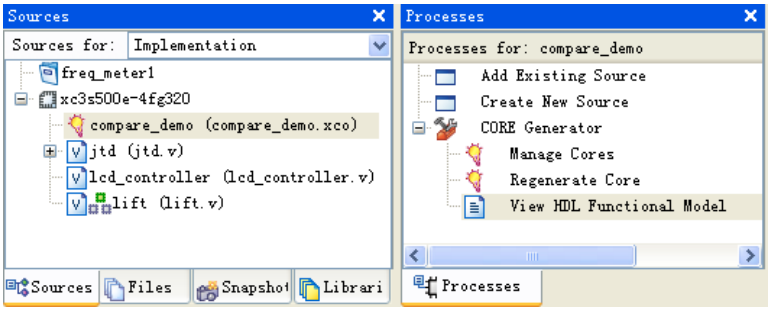


图 2-27　察看比较器 IP 核源代码的操作示意图

比较器的 IP 核代码如下，读者只需要关注其模块头的端口信号声明既可，这是因为在模块例化时需要用到其模块端口声明。

```
////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
////////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version: K.37
//  \   \         Application: netgen
//  /   /         Filename: compare_demo.v
// /___/   /\     Timestamp: Mon Oct 27 19:53:47 2008
// \   \  /  \
//  \___\/\___\
//
// Command   : -intstyle ise -w -sim -ofmt verilog
//D:\work\ise10p1\verilog\chapter11\freq_meter1\tmp\_cg\compare_demo.ngc
//D:\work\ise10p1\verilog\chapter11\freq_meter1\tmp\_cg\compare_demo.v
// Device  : 3s500efg320-4
// Input file    : D:/work/ise10p1/verilog/chapter11/freq_meter1/tmp/_cg/compare_demo.ngc
// Output file   : D:/work/ise10p1/verilog/chapter11/freq_meter1/tmp/_cg/compare_demo.v
// # of Modules     : 1
// Design Name     : compare_demo
// Xilinx          : E:\Xilinx\10.1\ISE
//
// Purpose:
//     This verilog netlist is a verification model and uses simulation
//     primitives which may not represent the true implementation of the
//     device, however the netlist is functionally correct and should not
//     be modified. This file cannot be synthesized and should only be used
//     with supported simulation tools.
//
// Reference:
//     Development System Reference Guide, Chapter 23 and Synthesis and Simulation
Design Guide, Chapter 6
//
////////////////////////////////////////////////////////////////////////////////
`timescale 1 ns/1 ps
module compare_demo (
```

```
    sclr, qa_ge_b, ce, clk, a, b
    );
      input sclr;
      output qa_ge_b;
      input ce;
      input clk;
      input [15 : 0] a;
      input [15 : 0] b;

      // synthesis translate_off
      //  下面的代码省略
      …
endmodule
```

（6）在工程中新建 mycompare.v 的 Verilog HDL 源文件，添加以下的代码：

```
module mycompare(
    sclr, qa_ge_b, ce, clk, a, b);
    //声明模块端口
    input sclr;
    output qa_ge_b;
    input ce;
    input clk;
    input [15 : 0] a;
    input [15 : 0] b;

    //  例化比较器的 IP 核
    compare_demo inst_compare_demo(
      .sclr(sclr),
      .qa_ge_b(qa_ge_b),
      .ce(ce),
      .clk(clk),
      .a(a),
      .b(b)
      );

endmodule
```

为了测试 IP 核，还需要在工程中新建一个测试代码（Verilog Test Fixture），命名为 tb_mycompare.v，其代码如下所列：

```
`timescale 1ns / 1ps
module tb_mycompare;
```

```
//声明输入信号
reg sclr;
reg ce;
reg clk;
reg [15:0] a;
reg [15:0] b;

//声明输出信号
wire qa_ge_b;

//  例化被测试模块(UUT)
mycompare uut (
        .sclr(sclr),
        .qa_ge_b(qa_ge_b),
        .ce(ce),
        .clk(clk),
        .a(a),
        .b(b)
);

initial begin
        //初始化所有的输入信号
        sclr = 0;
        ce = 0;
        clk = 0;
        a = 0;
        b = 0;
        //全局复位 100 个仿真时间单位
        #100;
        //添加相应的仿真输入激励
        ce = 1;
           #100;
        sclr = 1;
           #100;
        sclr = 0;
end

//产生时钟信号
always #5 clk = ~clk;
```

```
//产生测试数据
always #10 begin
    a = a + 3000;
     b = b + 6000;
end

endmodule
```

在 ISE Simulator 中运行上述程序，可以得到图 2-28 所示的仿真结果。当 sclr 为高时，同步清空有效，不管输入如何，比较器输出信号 qa_ge_b 为 0；当 sclr 且 ce 信号为高时，当 a 的值大于 b 时，qa_ge_b 为高，否则为 0。可以看出，比较器 IP 核正常工作，完成了 A>=B 的逻辑判断功能。

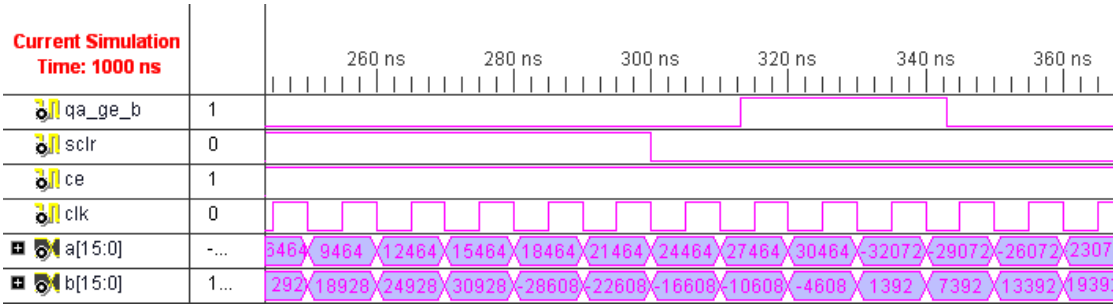


图 2-28　比较器 IP 核的仿真结果示意图

2.4.5　用户约束输入

用户约束是 Verilog HDL 设计所不可缺少的，包括管脚约束、时序约束以及区域约束等多类约束，对于一般的设计（非高速设计），只有管脚约束是必备的，因此本节首先介绍约束文件，然后详细介绍添加管脚约束的方法。

1．约束文件综述

（1）约束文件简介

在 ISE 中有多种用户约束，可指定设计各个方面的设计要求，如管脚位置约束、区域约束、时序约束以及电平约束等。其中，管脚约束将模块的端口和 FPGA 的管脚对应起来；时序约束保证了设计在高速时钟下的工作可靠性等等。由于用户约束文件（UCF）操作简便且功能强大，获得了广大设计人员的青睐。

FPGA设计中的约束文件有3类：用户设计文件（.UCF）、网表约束文件（.NCF）以及物理约束文件（.PCF），可以完成时序约束、管脚约束以及区域约束。3类约束文件的关系为：用户在设计输入阶段编写UCF文件，然后UCF文件和设计综合后生成NCF文件，最后再经过实现后生成PCF文件。本书只涉及UCF文件的使用方法。

UCF文件是ASCII码文件，描述了逻辑设计的约束，可以用文本编辑器和Xilinx约束文件编辑器进行编辑。NCF约束文件的语法和UCF文件相同，二者的区别在于：UCF文件由用户输入，NCF文件由综合工具自动生成。当二者发生冲突时，以UCF文件为准，这是因为UCF的优先级最高。PCF文件可以分为两个部分：一部分是映射产生的物理约束，另一部分是用户输入的约束，同样用户约束输入的优先级最高。一般情况下，用户约束都应在UCF文件中完成，不建议直接修改NCF文件和PCF文件。

（2）创建约束文件

约束文件的后缀是.ucf，所以一般也被称为 UCF 文件。创建约束文件的过程为：首先，新建一个源文件，在代码类型中选取“Implementation Constrains File”，在“File Name”中输入约束文件的名称；其次，单击“Next”按键进入模块选择对话框，选择要添加约束的模块，然后单击“Next”进入下一页；最后，单击“Finish”按键完成约束文件的创建。

（3）编辑约束文件

在工程管理区中，将“Source for”设置为“Synthesis/Implementation”，并用鼠标选中约束文件，然后双击过程管理区中“User Constrains”下的“Edit Constraints (Text)”就可以打开约束文件编辑器，如图 2-29 所示。

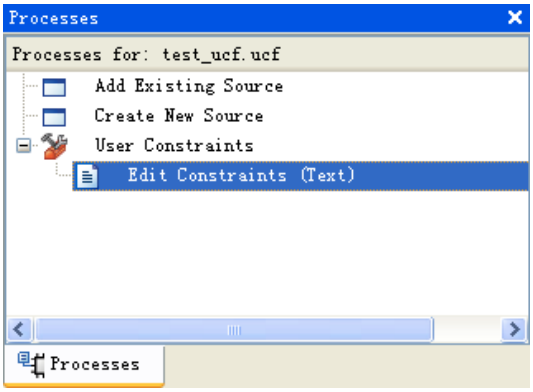


图 2-29　用户约束管理窗口

UCF 文件编辑器和普通文本编辑器的操作方法一致，用户可直接输入时序、位置以及电平等约束语句。需要注意的是，UCF 文件是大小敏感的，端口名称必须和源代码中的名字一致，且端口名字不能和关键字一样。但是关键字 NET 不区分大小写。

2．UCF 语法说明

（1）语法

UCF 文件的语法为：

{NET|INST|PIN} "signal_name" Attribute;

其中，“signal_name”是指所约束对象的名字，包含了对象所在层次的描述；“Attribute”为约束的具体描述；语句必须以分号“;”结束。可以用“#”或“/*　*/”添加注释。需要注意的是：UCF 文件是大小写敏感的，信号名必须和设计中保持大小写一致，但约束的关键字可以是大写、小写甚至大小写混合。例如：

NET "CLK" LOC = P30;

“CLK”就是所约束信号名，“LOC = P30;”是约束具体的含义，将 CLK 信号分配到 FPGA 的 P30 管脚上。

对于所有的约束文件，使用与约束关键字或设计环境保留字相同的信号名会产生错误信息，除非将其用" "括起来，因此在输入约束文件时，最好用" "将所有的信号名括起来。

（2）通配符

在 UCF 文件中，通配符指的是“*”和“?”。“*”可以代表任何字符串以及空，“?”则代表一个字符。在编辑约束文件时，使用通配符可以快速选择一组信号，当然这些信号都要

包含部分共有的字符串。例如：

NET "*CLK?" FAST;

将包含“CLK”字符并以一个字符结尾的所有信号，并提高其速率。

在位置约束中，可以在行号和列号中使用通配符。例如：

INST "/CLK_logic/*" LOC = CLB_r*c7;

把 CLK_logic 层次中所有的实例放在第 7 列的 CLB 中。

3．管脚约束

管脚约束是 Verilog HDL 设计所不可缺少的，它将模块的端口和可编程逻辑芯片的管脚对应起来。在 ISE 中有多种管脚分配的方法，其中基于约束文件的方法获得了设计人员的青睐。下面主要介绍基于约束编辑器的管脚分配方法，其它分配管脚的方法可以参阅 ISE 的在线帮助。

使用 LOC 完成端口定义时，其语法如下：

NET "Top_Module_PORT" LOC = "Chip_Port";

其中，“Top_Module_PORT”为用户设计中顶层模块的信号端口，“Chip_Port”为FPGA芯片的管脚名。“LOC”约束是FPGA设计中最基本的布局约束和综合约束，能够定义基本设计单元在芯片中的位置，可实现绝对定位、范围定位以及区域定位。

LOC语句中是存在优先级的，当同时指定LOC端口和其端口连线时，对其连线约束的优先级是最高的。例如，在图2-30中，LOC=11的优先级高于LOC=38。

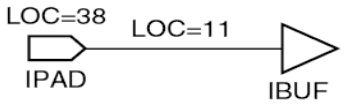


图 2-30　LOC 优先级示意图

LOC 语句通过加载不同的属性可以约束管脚位置、CLB、Slice、TBUF、块 RAM、硬核乘法器、全局时钟、数字锁相环（DLL）以及 DCM 模块等资源，基本涵盖了 FPGA 芯片中所有类型的资源。由此可见，LOC 语句功能十分强大，表 2-2 列出了用于管脚约束的 LOC 属性。

表 2-2　　LOC 语句常用属性列表

约束类型	可用属性示例	属性含义
IO 管脚约束	P12	将信号置于由芯片引脚号定位的端口上
	A12	将信号置于由芯片引脚阵列号定位的端口上
	B, L, T, R	将信号定位到芯片特定边界中（从物理位置上划分的上、下、左、右 4 部分）的端口上。
	LB, RB, LT, RT, BR, TR, BL, TL	将信号定位到芯片特定边界上的一半（从物理位置上划分的上左 、上右 、下左、下右、左上、坐下、右上以及右下 8 部分）位置中的端口上。
	Bank0, Bank1, Bank2, Bank3, ...	将信号置于特定管脚分组中的端口上。

2.4.6 综合与实现

1．综合

所谓综合，就是将HDL语言、原理图等设计输入翻译成由与、或、非门和RAM、触发器等基本逻辑单元的逻辑连接（网表），并根据目标和要求（约束条件）优化所生成的逻辑连接，生成NGC、NCR以及LOG文件，如图2-31所示。XST内嵌在ISE 3以后的版本中，并且在不断完善。此外，由于XST是Xilinx公司自己的综合工具，对于部分Xilinx芯片独有的结构具有更好的融合性。

完成了输入和仿真后就可以进行综合了。在过程管理区双击 Synthesize-XST 即可开始综合过程，如图 2-32 所示。此外，在 ISE10.1 中，管脚分配可以在综合之前完成，也可以在综合之后完成，对于一般使用差异不大，取决于用户选择；在大规模或高速设计中，建议还是先使用区域约束，然后根据约束情况再来分配管脚。

综合可能有 3 种结果：如果综合后完全正确，则在 Synthesize-XST 前面有一个打钩的绿色小圈圈；如果有警告，则出现一个带感叹号的黄色小圆圈；如果有错误，则出现一个带叉的红色小圈圈。如果综合步骤没有语法错误，XST 能够给出初步的资源消耗情况，点击过程管理区的“View Design Summary”，即可查看，如图 2-33 所示。

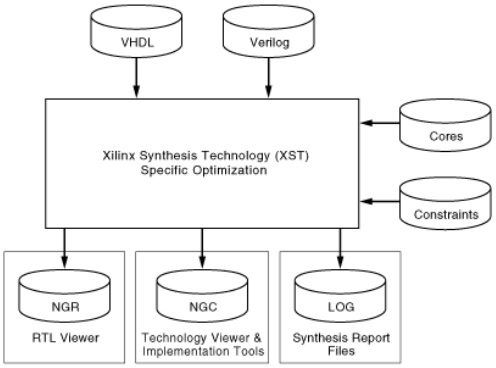


图 2-31　综合工具 XST 的功能示意图

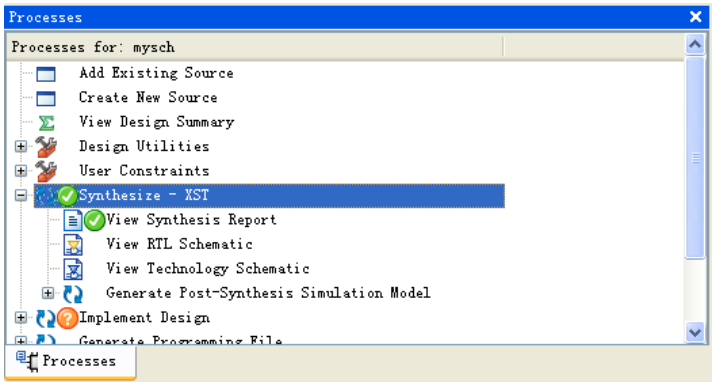


图 2-32　综合操作窗口示意图

mycounter Project Status (05/26/2008 - 12:54:52)			
Project File:	mycounter.ise	Current State:	Synthesized
Module Name:	mysch	• Errors:	No Errors
Target Device:	xc4vlx15-12sf363	• Warnings:	No Warnings
Product Version:	ISE 10.1 - Foundation Simulator	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

mycounter Partition Summary	[-]
No partition information was found.	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	Used	Available	Utilization
Number of Slices	4	6144	0%
Number of Slice Flip Flops	8	12288	0%
Number of 4 input LUTs	9	12288	0%
Number of bonded IOBs	10	240	4%
Number of GCLKs	1	32	3%

图 2-33　综合结果报告

综合完成之后，可以通过双击 View RTL Schematics 来查看 RTL 级结构图，察看综合结构是否按照设计意图来实现电路。ISE 会自动调用原理图编辑器 ECS 来浏览 RTL 结构。对于例 2-2，所得到的 RTL 结构图如图 2-34 所示，综合结果符合设计者的意图，调用了加法器和寄存器来完成逻辑。

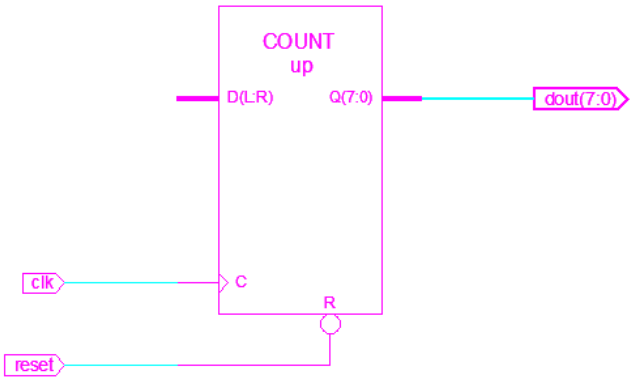


图 2-34　例 2-2 的综合后的 RTL 结构示意图

2．实现

所谓实现（Implement）是将综合输出的逻辑网表翻译成所选器件的底层模块与硬件原语，将设计映射到器件结构上，进行布局布线，达到在选定器件上实现设计的目的。实现主要分为3个步骤：翻译（Translate）逻辑网表，映射（Map）到器件单元与布局布线（Place & Route）。在ISE中，执行实现过程，会自动执行翻译、映射和布局布线过程；也可以单独执行。

翻译的主要作用是将综合输出的逻辑网表翻译为Xilinx特定器件的底层结构和硬件原语。映射的主要作用是将设计映射到具体型号的器件上（LUT、FF、Carry等）。布局布线步骤调用Xilinx布局布线器，根据用户约束和物理约束，对设计模块进行实际的布局，并根据设计连接，对布局后的模块进行布线，产生FPGA/CPLD配置文件。

在过程管理区双击“Implement Design”选项，就可以自动完成实现的 3 个步骤，如图 2-35 所示。如果设计没有经过综合，会启动 XST 完成综合，在综合后再完成实现过程。

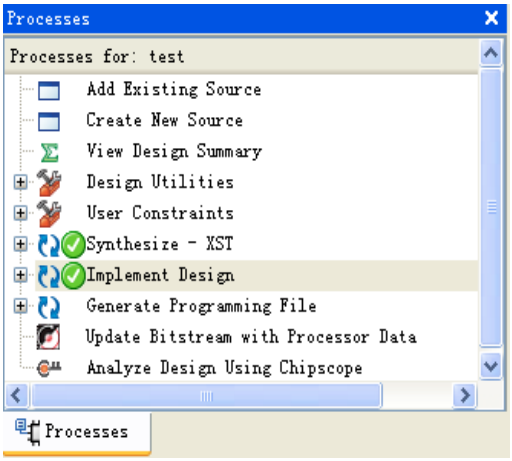


图 2-35　设计实现窗口

经过实现后能够得到精确的资源占用情况，以例 2-2 的计数器设计为例，双击“Place & Router”下的“Place & Route Report”即可查阅最终的资源报告，如图 2-36 所示。

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    1
Logic Utilization:
  Number of 4 input LUTs:                8 out of  30,720    1%
Logic Distribution:
  Number of occupied Slices:                             5 out of  15,360    1%
    Number of Slices containing only related logic:      5 out of       5  100%
    Number of Slices containing unrelated logic:         0 out of       5    0%
      *See NOTES below for an explanation of the effects of unrelated logic
Total Number of 4 input LUTs:              9 out of  30,720    1%
  Number used as logic:                    8
  Number used as a route-thru:             1
  Number of bonded IOBs:                  17 out of     448    3%
  Number of BUFG/BUFGCTRLs:                1 out of      32    3%
    Number used as BUFGs:                  1
    Number used as BUFGCTRLs:              0
```

图 2-36　实现后的资源统计结果

2.4.7　器件配置

1．FPGA 配置

FPGA 配置主要用于调试阶段快速、多次验证功能，断电后芯片内的逻辑立刻消失，每次上电都需要重新配置。该操作比较简单，首先，在配置启动参数中选择配置时钟为 JTAG CLK，否则会产生警告，配置过程容易出错；其次，只需在过程管理区中双击 Generate Programming File 选项即可完成，完成后则该选项前面会出现一个打钩的圆圈，如图 2-37 所示；并弹出图 2-38 所示的报告信息对话框，直接关掉即可。生成的编程文件放在 ISE 工程目录下，是一个扩展名为<顶层文件名.bit>的二进制比特流文件。

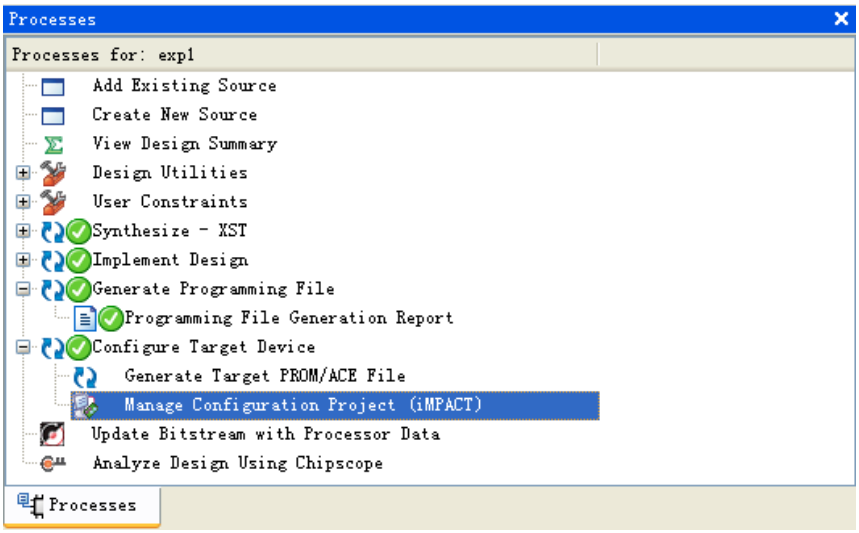


图 2-37　生成编程文件的窗口

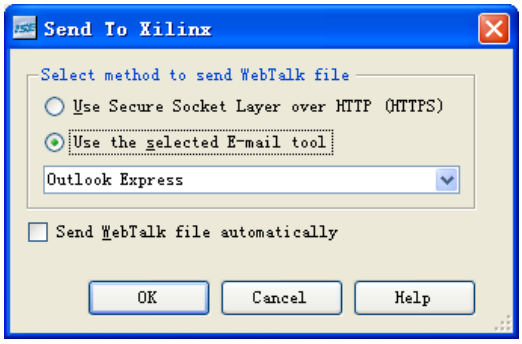


图 2-38　比特文件报告对话框

到此，只剩下完成设计的最后一步——下载。双击过程管理区的“Configure Target Device”栏目下的“Manage Configure Project (iMPACT)”，然后在弹出的 iMPACT 配置对话框中选取“Configure device using Boundary-Scan (JTAG)”，如图 2-39 所示。

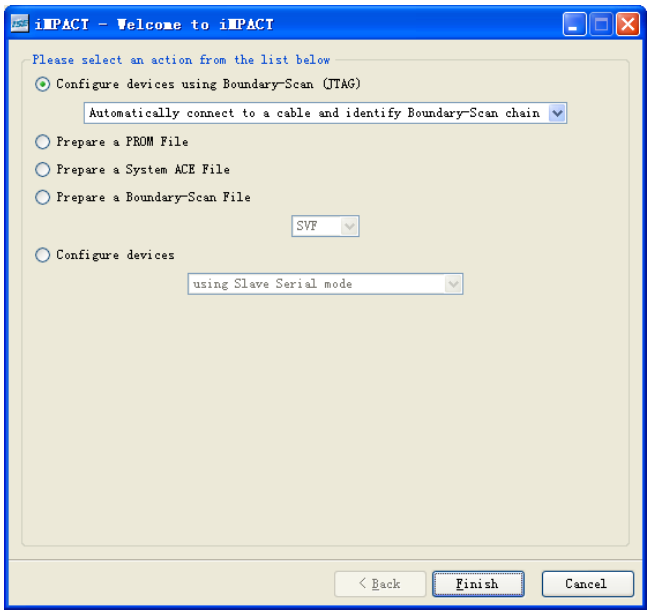


图 2-39　iMPACT 配置对话框

点击“OK”按键后，ISE 会自动连接 FPGA 设备。成功检测到设备后，会显示出 JTAG 链上所有芯片，其典型示意如图 2-40 所示。从中可以看出，链上的所有芯片构成了一个 TDI 到 TDO 的完整回路，这是电路设计时必须要保证的。

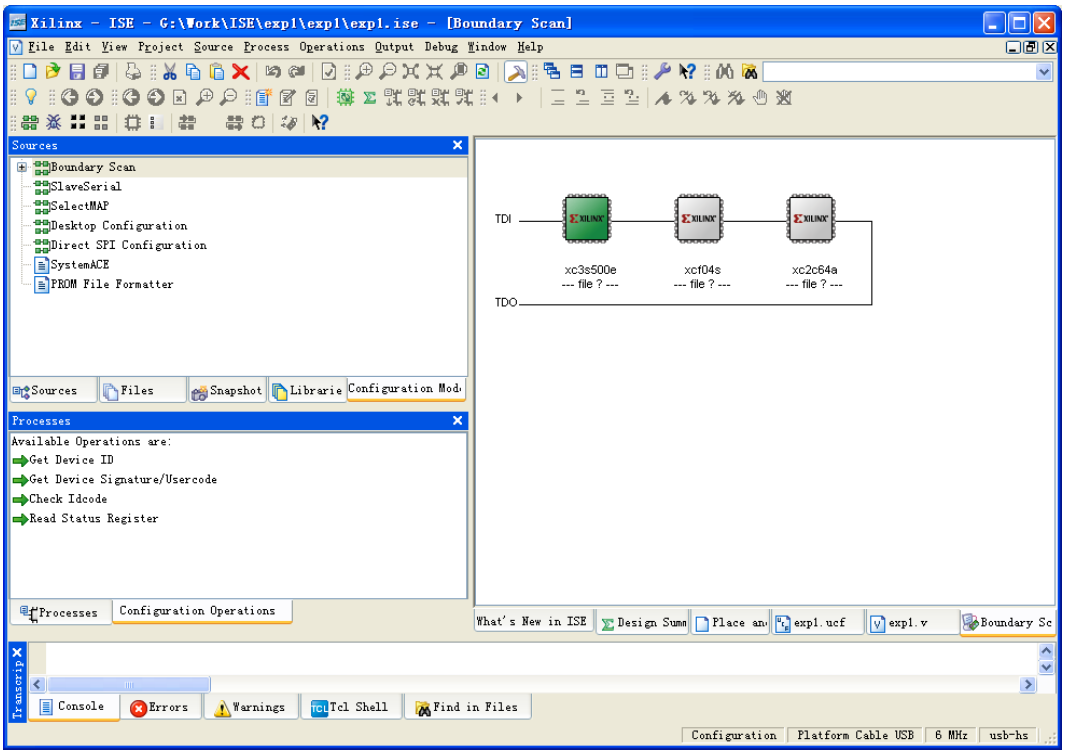


图 2-40　JTAG 链边界扫描结果示意图

如果 JTAG 链检测失败，其弹出的对话框如图 2-41 所示。在配置 FPGA 器件时，经常会出现扫描失败的情况，常用的解决方法有：首先，检查电源是否正确；其次，采用质量更好的并口配置电缆（Parallel Cable-IV）或信号质量更好的 USB 配置电缆，排除下载线的问题；第三，检查所有芯片的 TCK、TMS 管脚是否和 JTAG 接口的 TCK、TMS 连接在一起；最后，检查配置电路的 JTAG 链路是否完整，从 JTAG 接口的 TDI 到链首芯片的 TDO、……、再到链尾芯片的 TDO 是否连接到 JTAG 接口的 TDO。

图 2-41　JTAG 链扫描失败对话框

JTAG 链检测正确后，在 FPGA 芯片的图标上双击，或点击右键并在弹出的菜单中选择“Assign New Configuration File”，会弹出图 2-42 的窗口，让用户选择后缀为.bit 的二进制比特流文件，然后单击“Open”按键完成，则会在 iMPACT 的主界面会出现一个芯片模型以及位流文件的标志。

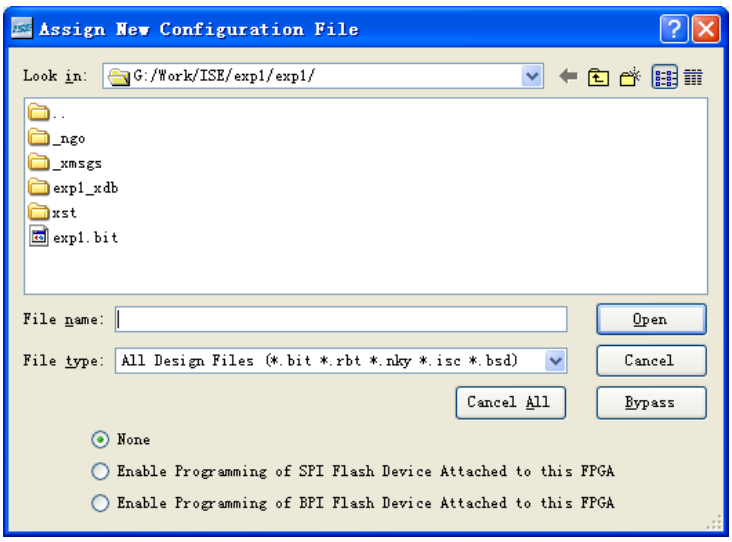


图 2-42　选择位流文件

在 FPGA 芯片标志上单击右键，在弹出的对话框中选择 Program 选项，就可以对 FPGA 设备进行编程，如图 2-43 所示。

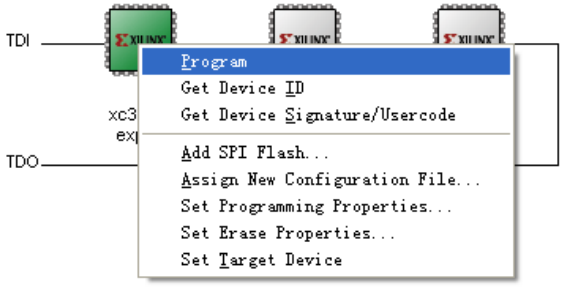


图 2-43　对 FPGA 设备进行编程示意图

配置成功后，会弹出配置成功的界面，如图 2-44 所示。

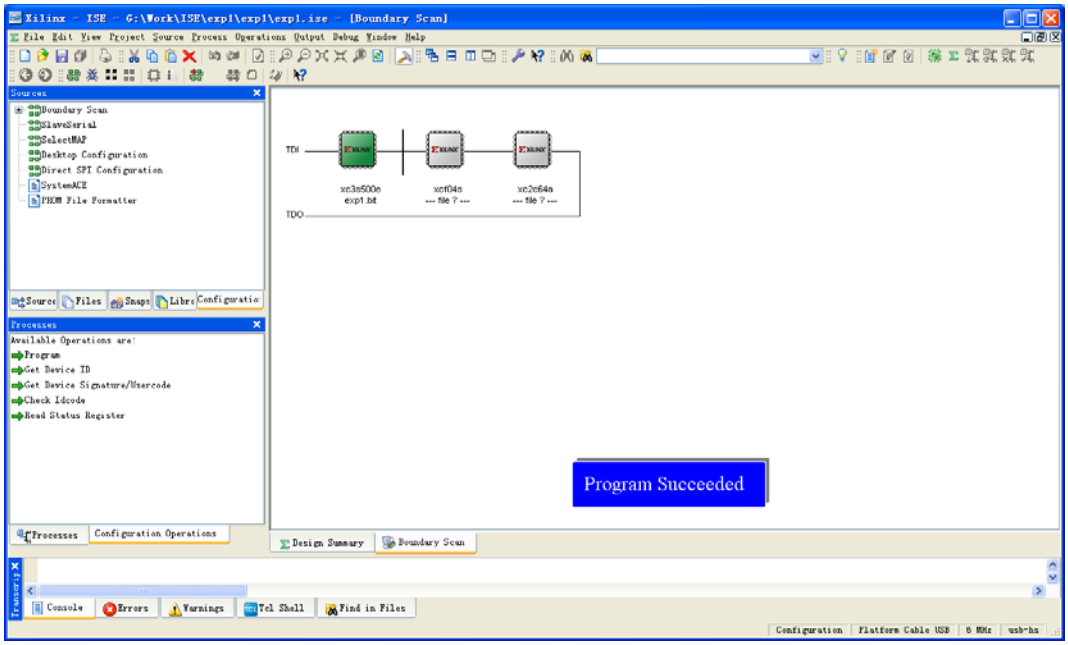


图 2-44　FPGA 配置成功指示界面

2．PROM 配置

一个设计经过综合、实现之后，需要为不同器件生成不同类型的编程文件。ISE 中内嵌了比特流生成器，可生成 FPGA 以及 PROM 格式文件，从而实现动态配置，并验证数据是否正确。只有生成 PROM 文件并下载 PROM 芯片中，才能保证 FPGA 上电后自动加载逻辑并正常工作。和生成 FPGA 配置文件（.bit）相比，生成 PROM 配置文件的操作较为复杂，下面对其进行详细说明。

（1）将设计经过前仿、综合、实现以及后仿，确保设计无误。双击过程管理区的“Configure Target Device”栏目下的“Manage Configure Project (iMPACT)”，然后在弹出的 iMPACT 配置对话框中选取“Prepare a PROM File”，如图 2-45 所示。

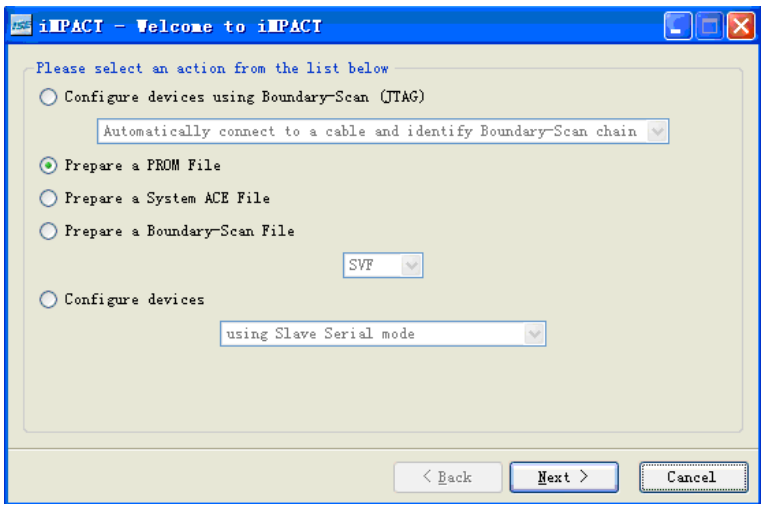


图 2-45　iMPACT 配置对话框

（2）单击“Next”按钮，进入 PROM 器件选择界面，如图 2-46 所示。下面以 Xilinx PROM 为例进行说明。选中 Xilinx PROM，在文件格式“PROM File Format”中选择 EXO，可在“PROM File Name”文本框修改 PROM 配置文件名称，在“Location”栏输入 PROM 文件的存放路径。

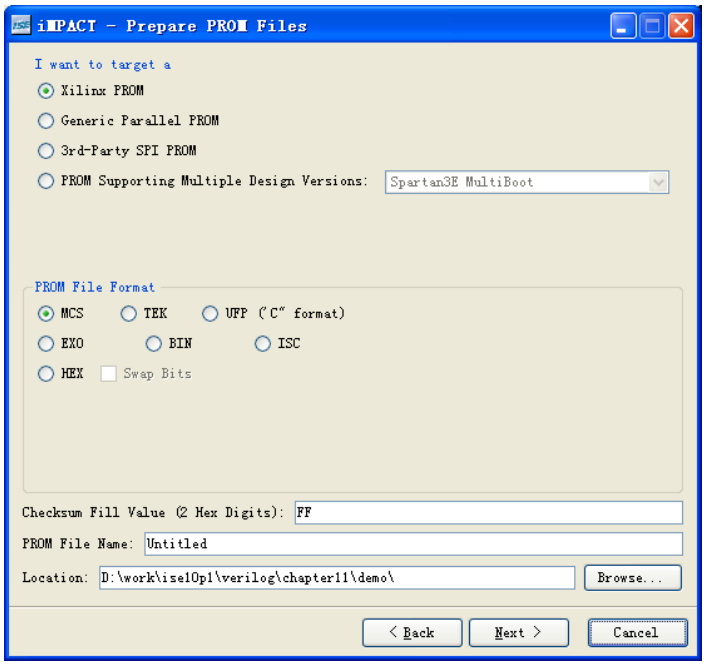


图 2-46　选择 PROM 芯片的类型和文件格式

（3）单击“Next”按钮，进入 PROM 模式配置界面，如图 2-47 所示，支持单片串行、并行以及多片 PROM 的级联配置，这里选择串行，因为 XCF04S 只支持串行配置。

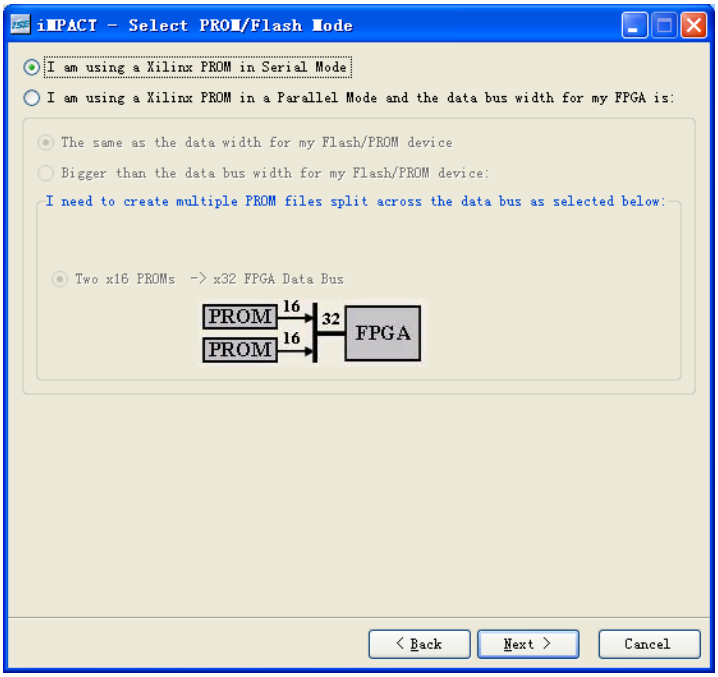


图 2-47　PROM 芯片配置模式选择界面

（4）点击“Next”按钮，选择 PROM 器件的型号，如图 2-48 所示。可以选中“Auto Select PROM”选项，由 iMPACT 自动选择合适的 PROM 芯片，也可以手动在“Select a PROM”选项的下拉框中选择合适的 PROM 芯片，然后单击“Add”按钮添加选中的器件。可根据需要反复多次，添加多个 PROM 芯片。此外，对于 XCF08P 以上的批 ROM 芯片，还可以使能修改和压缩功能。

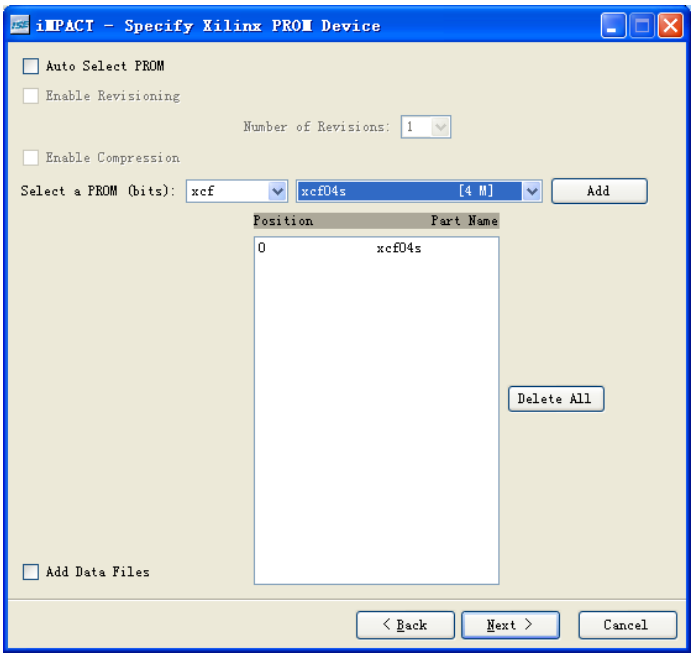


图 2-48　PROM 芯片型号选择界面

（5）单击图 2-48 中的“Next”按键，进入 PROM 文件综合信息显示窗口，如图 2-49 所示。如果确认信息无误，单击“Finish”，进入后续步骤；否则返回前面进行修改。

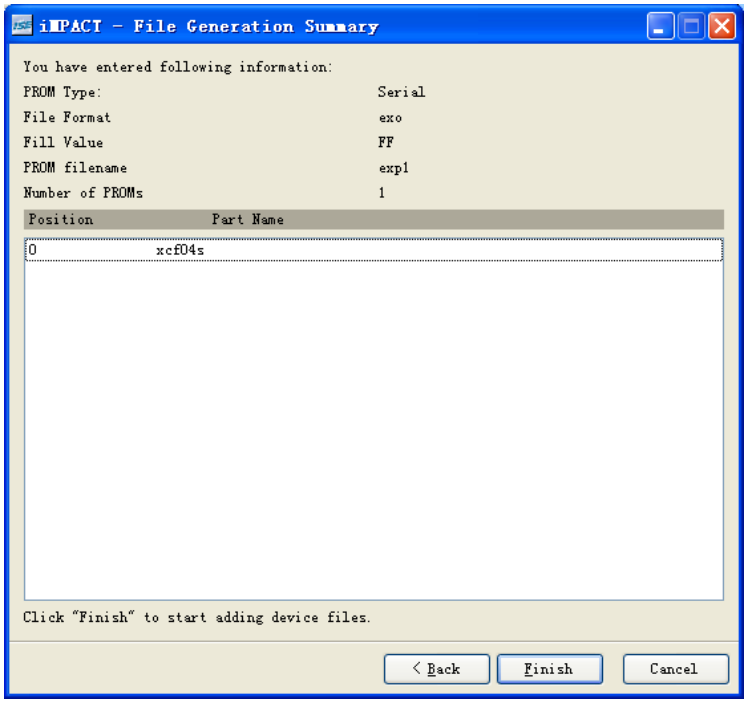


图 2-49　PROM 芯片综合信息显示界面

（6）单击图 2-49 中的“Finish”按键，弹出的比特文件加载窗口如图 2-50 所示，点击弹出“Add Device”对话框的“OK”按键，可打开.bit 文件选择界面，如图 2-51 所示。

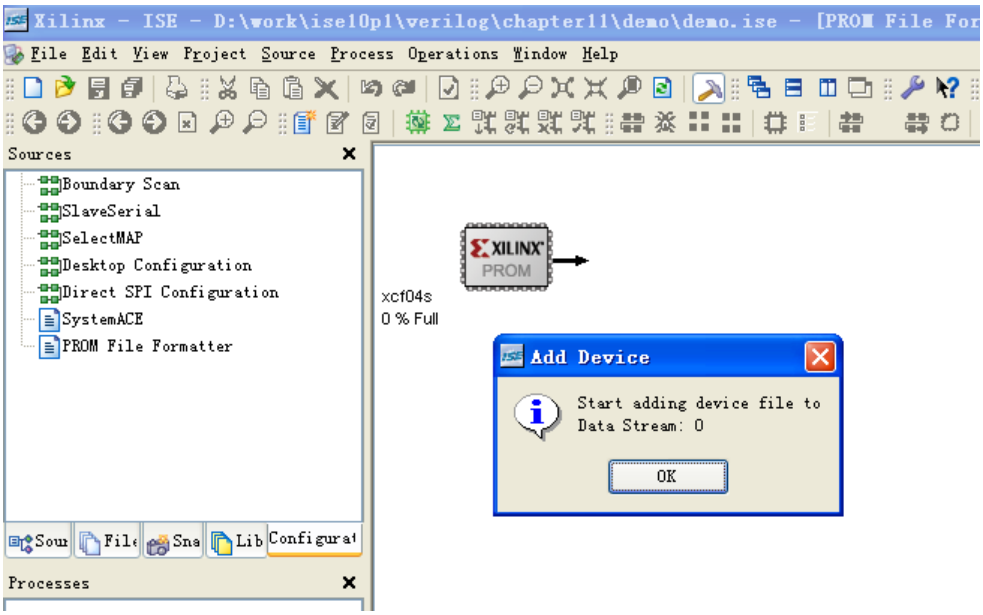


图 2-50　比特文件加载窗口

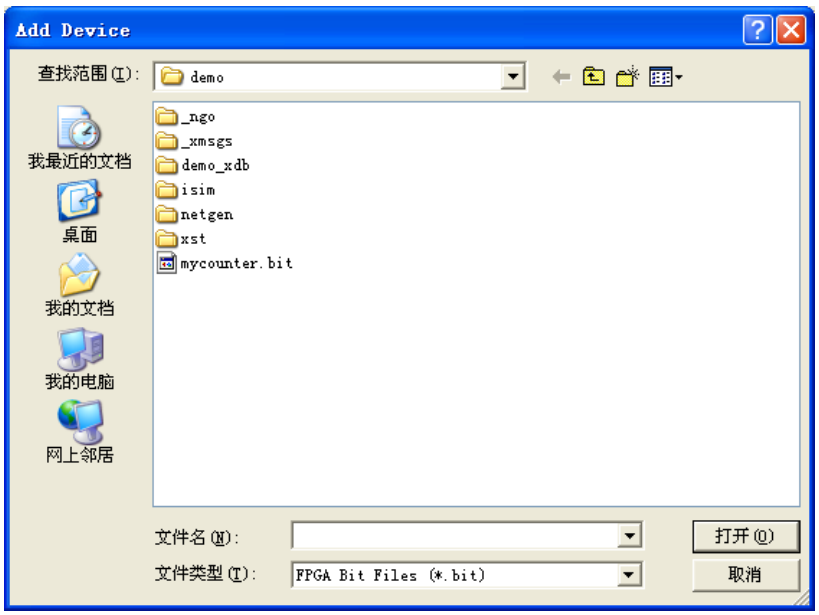


图 2-51　比特文件选择界面

选择加载的比特文件后，单击图 2-51 中的“打开(O)”按键后，iMPACT 会提示用户是否再添加比特文件，如图 2-52 所示。这是因为只要容量允许，一片 PROM 可配置多个 FPGA。如果还有 FPGA 配置文件，可点击“Yes”继续添加，否则可点击“No”按键，iMPACT 会弹出加载完成对话框，如图 2-53 所示，点击“OK”完成比特文件加载。

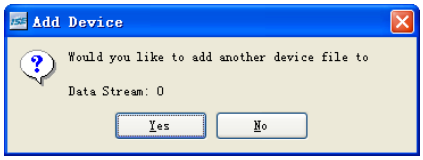


图 2-52　比特文件选择界面图

图 2-53　比特文件选择界面

（7）此时，iMPACT 会根据加载的 bit 文件所对应的 FPGA 芯片计算 PROM 的容量，如果 PROM 容量不够，会主动提醒用户修改 PROM 型号或者添加更多的 PROM 芯片；如果容量富裕，则会给出 PROM 的容量利用率，如图 2-54 所示。例如，图中给出的设计使用了 54.13%的 PROM 容量。此时，还可以在 PROM、FPGA 器件的图标上点击右键分别更新芯片型号和相应的.bit 文件。

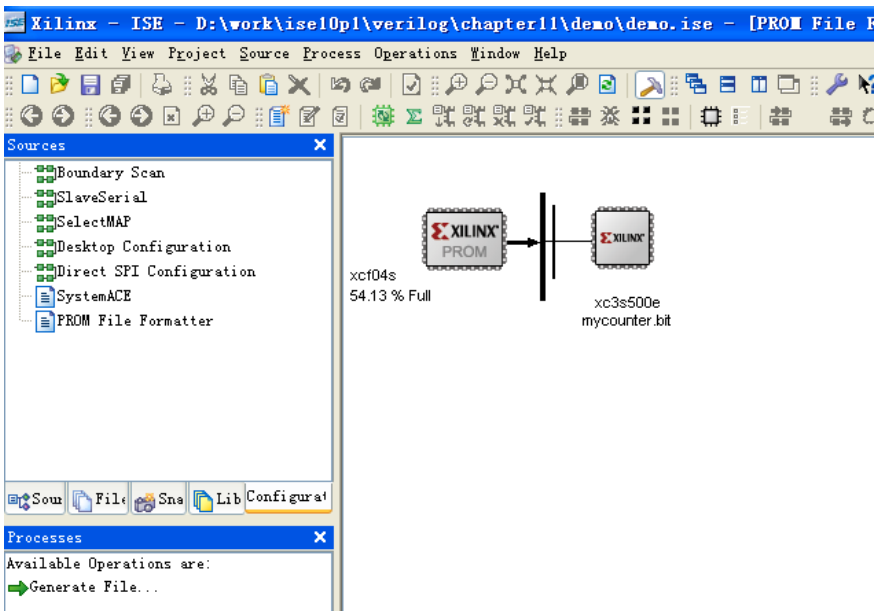


图 2-54　PROM 容量显示界面

在 iMPACT 的过程管理窗口，双击“Generate File”，iMPACT 会自动创建 PROM 配置文件，并在 iMPACT 界面上显示“PROM File Generation Succeeded”，如图 2-55 所示。

图 2-55　PROM 配置文件创建成功提示界面

（8）单击工具栏的“ ”按键，初始化 JTAG 链，并在 PROM 芯片上双击，可弹出 PROM 配置文件选择界面，选择刚才生成的 MCS 文件。这一过程和为 FPGA 芯片选择 bit 文件是一致的。

（9）在 PROM 芯片上点击右键，选择“Pragram”命令，会弹出图 2-56 所示的对话框，点击其中的“OK”按键，即开始配置 PROM 芯片，并弹出图 2-57 所示的进度条。

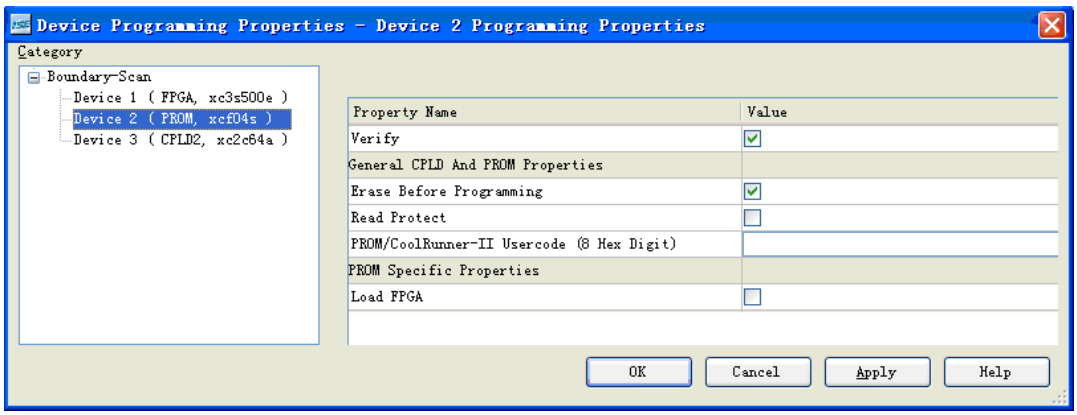


图 2-56　PROM 配置设置对话框

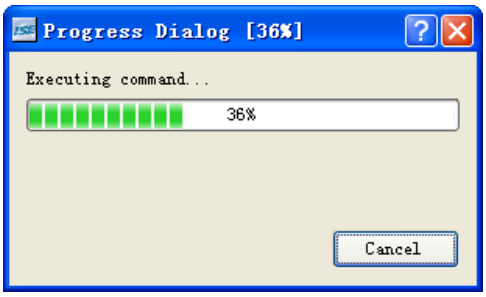


图 2-57　PROM 配置进度条

进度条达到 100%后，完成 PROM 配置后，iMPACT 给显示“Program Succeeded”，如图 2-58 所示。

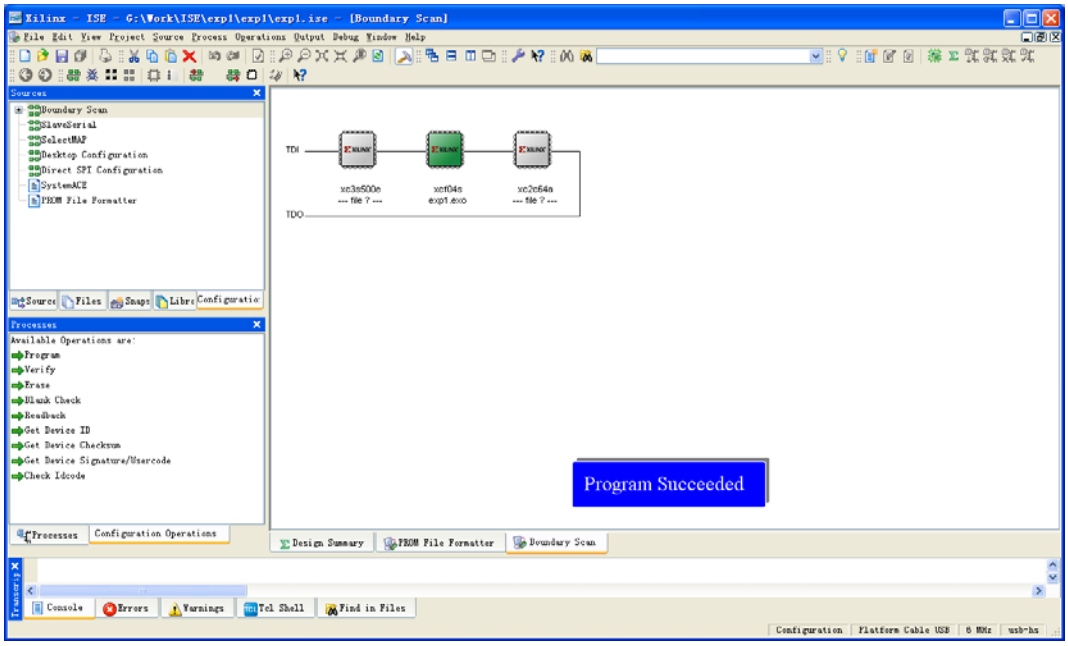


图 2-58　PROM 配置成功示意图

至此，就完成了一个完整的 FPGA 设计流程。当然，ISE 的功能十分强大，以上介绍只是其中最基本的操作，更多的内容和操作需要读者通过阅读 ISE 在线帮助来了解，在大量的实际实践中来熟悉。

2.5 ModelSim 快速入门

ModelSim 软件是一款强大的仿真软件，具有速度快、精度高和便于操作的特点，此外还具有代码分析能力，可以看出不同代码段消耗资源的情况。其功能侧重于编译和仿真，不能制定编译的器件和下载配置的能力，所只能作为第三方软件配套 ISE 使用。

2.5.1　ModelSim 仿真软件的安装

下面介绍一下 ModelSim 的安装步骤。

（1）运行安装程序后，出现图 2-59 所示的界面，如果拥有有效的 License，可以选择完全版（Full Product）安装，反之，则应当选择评估版（Evaluation Edition）安装。

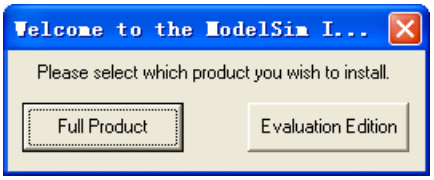


图 2-59　ModelSim 版本选择窗口

（2）选择完安装类型之后，下一个步骤就是设定安装路径，如图 2-60 所示。

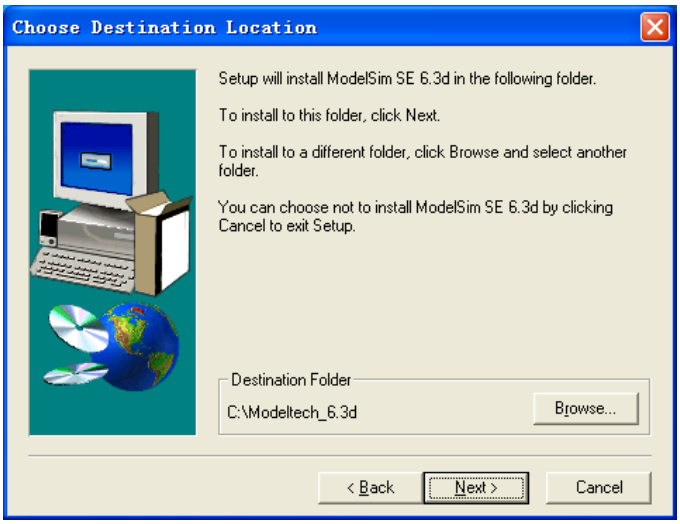


图 2-60　ModelSim 安装路径选择窗口

（3）如果选择的是完全版本，安装之后会出现 Liscense Wizard 对话框（或者通过“开始 → 所有程序 → ModelSim SE 6.3d → Liscense Wizard”打开 Liscense Wizard），如图 2-61 所示。

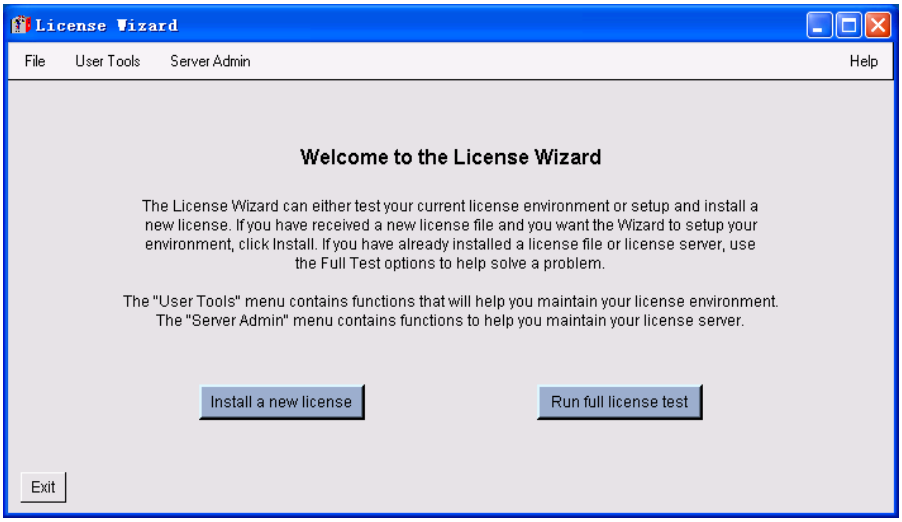


图 2-61　ModelSim 软件 License 管理向导

（4）选择“Install a new license”选项，出现图 2-62 界面，点击“Browse”，选中 License 所在的路径。

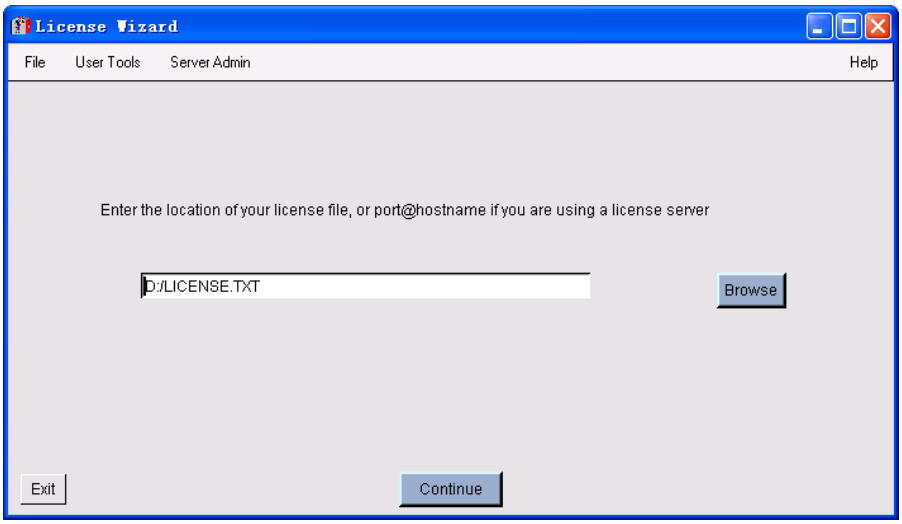


图 2-62　Liscense Wizard 导入 LICENSE 文件

（5）点击“Continue”，如果出现下图 2-63 所示的界面，则完成 License Wizard 配置验证操作，只有合法的 License 文件，才能使 ModelSim 正常工作。

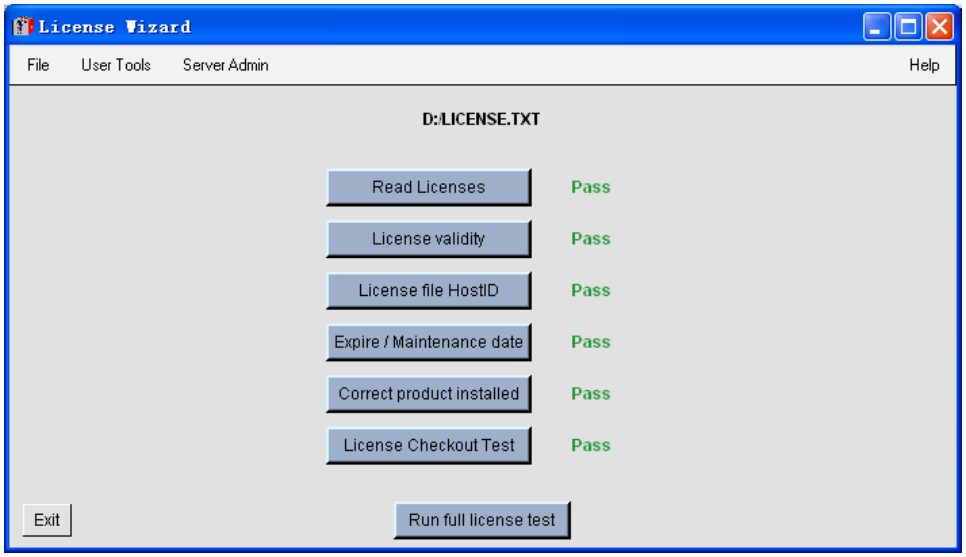


图 2-63　Liscense Wizard 完成 LICENSE 导入

2．关联 ISE 和 ModelSim

完成了 ModelSim 安装后，需要将其和 ISE 软件关联后才能使用 ModelSim 进行仿真。运行 ISE 软件，在主界面中选择“Edit|Preference”菜单项，如图 2-64 所示。

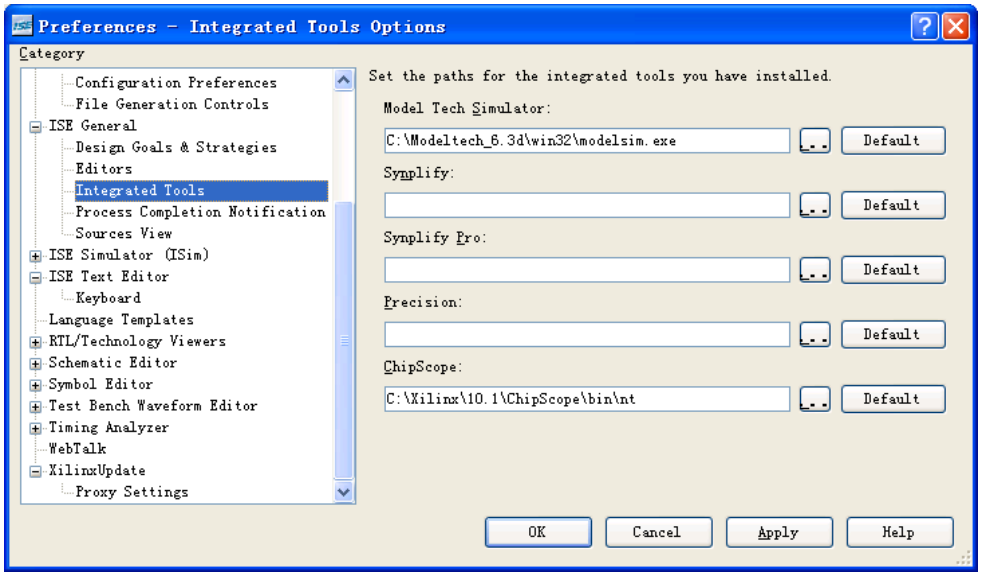


图 2-64　ISE 中“Preference”界面

在弹出的“Preference”对话框中选择“Integrated Tools”选项卡。该选项卡用于设定与 ISE 集成的软件的路径，第一项的“Model Tech Simulator”就用于设定 ModelSim 仿真软件的路径，如图 2-65 所示。单击“Model Tech Simulator”文本框后面的按钮，会弹出一个文件选择对话框，选择 ModelSim 安装路径下 win32 目录下的“modelsim.exe”文件即可。

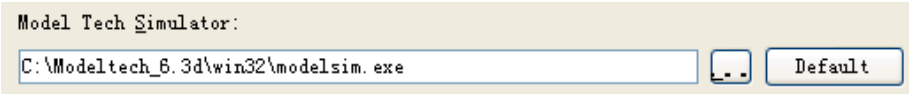


图 2-65　选中 modelsim 仿真软件

2.5.2　在 ModelSim 中指定 Xilinx 的仿真库

ModelSim SE 版在发行时是不带任何 FPGA 厂家的仿真库，因此用户必须手动编译这些库，由此面临的一个问题就是怎样建立各 FPGA 器件的仿真库，目前各 FPGA 厂家都支持用户编译库，所以实现比较简单。

在 ModelSim 中编译 Xilinx 仿真库有很多方法，下面介绍一种比较常用的方法，分为 3 步来完成。

（1）点击“开始/运行”按钮，执行下面的命令：

“compxlib -s mti_se -f all -l all –o c:\Modeltech _6.3d\xilinx_libs -p c:\ Modeltech _6.3d\win32”

其中，“c:\Modeltech _6.3d”为 ModelSim 软件的安装目录，用户可根据自己的目录来替换。等待上述命令运行完毕，其运行时间较长，用户不要中途中断。

（2）在 Xilinx 本地库编辑成功后，在相应的目录下，会自动生成 Modelsim.ini 的文件，用任何一个文本编辑器将该文件中[Library]目录下（除 others 以外）的内容添加到硬盘上相应的另外的 ModelSim 安装目录下同名“modelsim.ini”文件中的相应[Library]位置。

（3）最后进入 ISE 主界面，点击“Edit”下拉菜单按纽，选中下拉菜单中的“Prefrence”选项，再选中“Intergal Tools”页面，重新指定 ModelSim 可执行文件即可。退出所有软件，以后再对 Xilinx 的设计进行仿真都不需要进行库的处理了。

2.5.3 ModelSim 的基本操作

本节主要简单介绍 ModelSim SE 6.3d 的使用方法，主要包括建立工程和基本 Verilog 仿真，更多的操作方法需要在实际应用中熟悉并掌握。

1．建立工程

使用 ModelSim 建立工程主要包括 5 个基本步骤：

（1）启动 ModelSim，选择菜单“File → New → Poject”，会打开“Creat Project”对话框，如图 2-66 所示。在“Creat Project”对话框中填写“Project Name”为“test”，然后在“Project Location”栏中选择 Project 文件的存储目录，保留“Default Library Name”的设置为 work。点击 OK 按键确认，在 ModelSim 软件主窗口的工作区中即增加了一个空的 Project 标签，同时弹出一个“Add items to the Project”对话框，如图 2-67 所示。

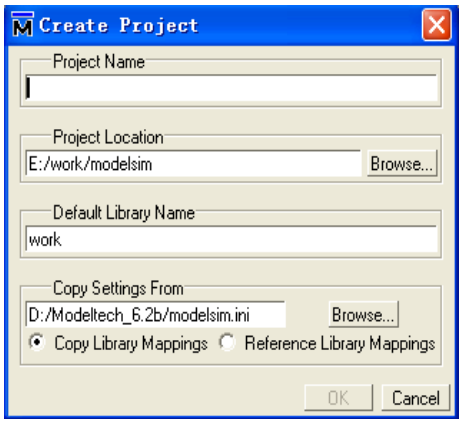


图 2-66　ModelSim 新建工程窗口

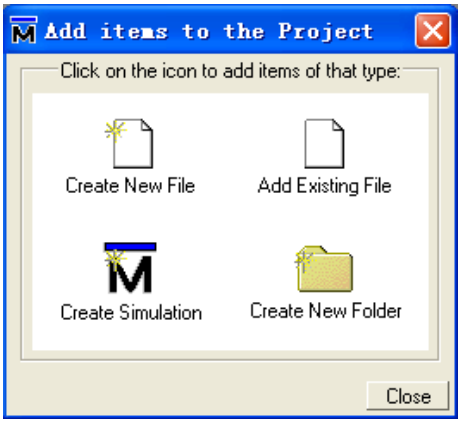


图 2-67　添加文件到工程向导示意图

（2）添加包含设计单元的文件。直接点击 Add items to the Project 对话框以后，在对话框中利用“Add Existing File”或“Create New File”选项，可以在工程中加入已经存在的文件或建立新文件。本节我们选择“Add Existing File”，弹出“Add file to the Project”对话框，如图 2-68 所示。点击对话框中的 Browse 按键，打开 ModelSim 安装路径中的“examples/tutorials/verilog/compare/”目录，选取 sm.v 和 sm.v 文件（选中多个文件时，只需要一直按住 Ctrl 按键，用鼠标点击即可），再选中对话框下面的“Reference from current location”选项，然后点击 OK 按键确认。

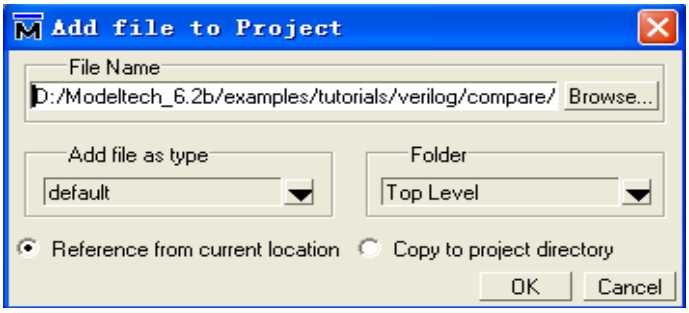


图 2-68　ModelSIm 添加文件选项示意图

（3）在工作区中的 Project 标签页中可以看到新加入的文件，单击右键，选取“Compile → Compile All”命令对加入的文件进行编译，如图 2-69 所示。

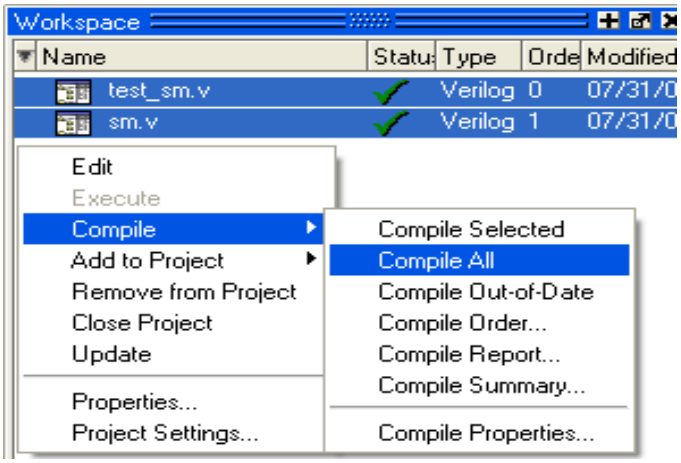


图 2-69　ModelSim 软件中的工程编译窗口

（4）两个文件编译完后，用鼠标点击“Library”标签栏。在标签栏中用鼠标点击 work 库前面的“+”，展开 work 库，就会看到两个编译了的设计单元，如图 2-70 所示。

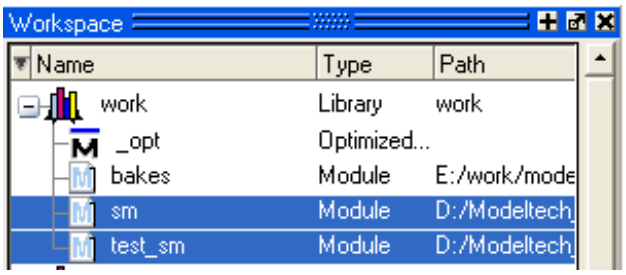


图 2-70　编译后的设计单元示意图

（5）导入设计单元。双击 Library 标签页中的“test_sm”，在工作区中将会出现 sim 标签，并在右边的对象窗口列出了 test_sm 单元所用到的信号，如图 2-71 所示。

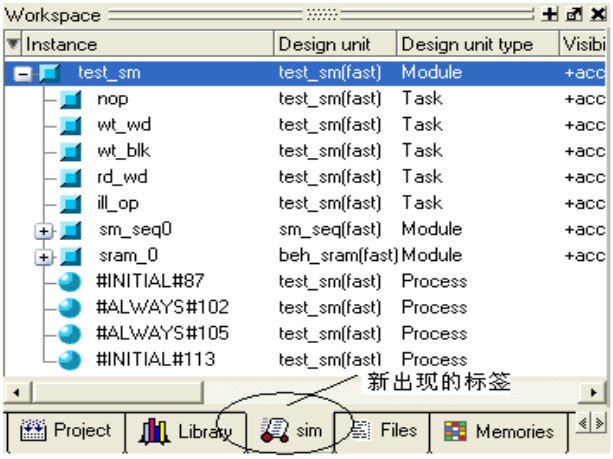


图 2-71　将 test_sm 模块加入工作区示意图

到此，一个工程就已经建立好了，接下来的就是开始运行仿真、分析和设计调试了。选择“File → Close → Poject”可以关闭当前目录。

2．基本 Verilog 仿真

在准备仿真的时候，需要完成上述建立工程中的所有步骤。然后继续进行下面的步骤：

（1）通过选择“View → <窗口名>”调出 signal、list 和 wave 等窗口。也可以通过在主

窗口命令行操作区的 VSIM 提示符下输入下面的命令：view signals list wave（回车）。

（2）向 wave 窗口添加信号。在 signal 窗口中，单击右键，在弹出的菜单中选择“Add to Wave”选项中的“Signal in design”，将设计中用到的所有信号都列在 Wave 窗口中，如图 2-72 所示。

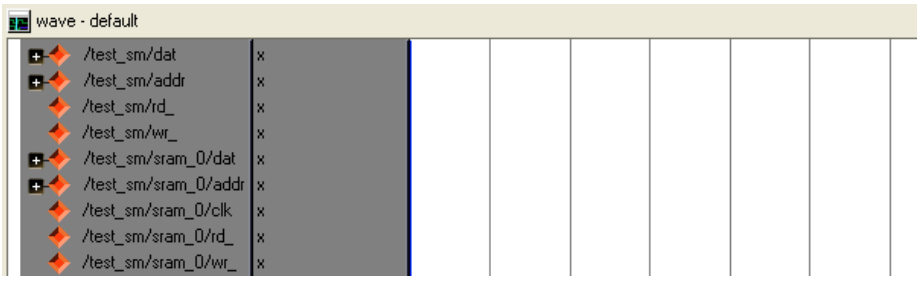


图 2-72　在 Wave 窗口中添加信号

（3）导入设计的时候，会在工作区打开一个新的 sim 标签，点击“+”展开设计层次结构，可以看到实例 test_sm、sm 等模块。点击 sim 标签中的顶层行保证 test_sm 模块显示在 source 窗口中。

（4）点击主窗口工具条的 Run 启动仿真，默认仿真长度为 100ns。或者在命令行输入 run。可以在仿真途中点击“Break”中断运行，在 source 窗口中查看中断时执行的语句。

（5）仿真完成，观察仿真波形如图 2-73 所示。确认无误后退出仿真，如果有错则返回 source 区域修改代码。

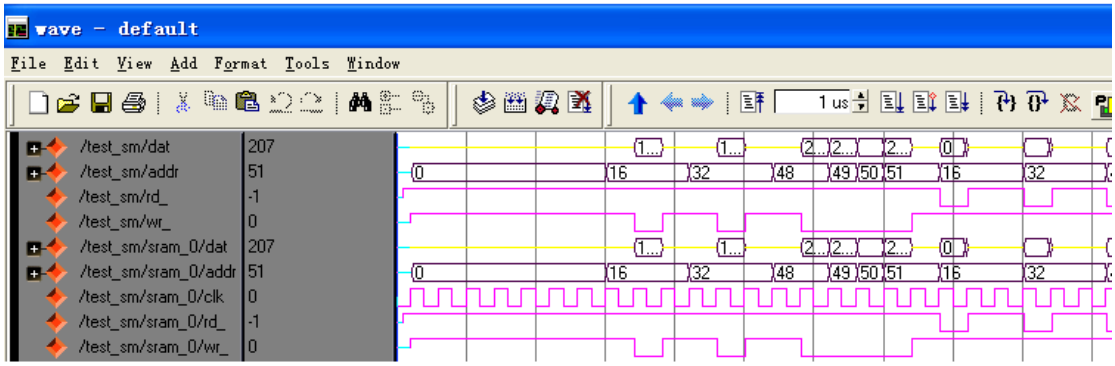


图 2-73　test_sm 模块的仿真结果示意图

2.6 本章小结

本章首先介绍了 Verilog HDL 代码开发的必备基础以及程序设计模式，指出传统的自下向上的开发模式已经不能适用大规模系统开发，并介绍了自顶向下的、层次化、模块化开发模式：从系统开始，把系统划分为若干个基本单元，再把每个基本单元划分为下一层次的基本单元，如此往下，直到可以直接利用 Verilog HDL 描述为止。此外，为了提高开发效率和质量，部分模块可通过知识产权核（IP）来完成设计，并且基于 IP 的设计方法已成为一种主流的设计模式。最后，介绍了 Xilinx 公司的 Spartan 3E 系列 FPGA 结构、ISE 和 ModelSim 软件的快速入门操作，为后续章节的学习做好必要的准备工作。

2.7 思考题

1．整数的二进制表示形式有哪些，之间怎样完成相互转化？

2．什么是信号的周期和占空比？

3．一个优秀 Verilog HDL 程序的评断标准是什么？

4．自顶向下设计模式的特点是什么，和传统的自低向上的设计模式相比有什么优势？

5．什么是 IP 核？具有什么设计优势？

6．Spartan 3E 系列的 FPGA 器件具有什么特点？

7．ISE 软件的主要功能包括哪些？

8．如何在 ISE 中完成设计输入、功能仿真以及 IP 核的调用？

9．什么是综合、实现过程？如何在 ISE 软件中完成？

10．为什么要配置 RPOM 芯片？如何在 ISE 软件中完成 CPLD/FPGA 以及 PROM 芯片的配置？

11．ModelSim 软件的特点是什么？如何在 ModelSim 软件中指定 Xilinx 仿真库，并完成基本的仿真功能？

第 3 章　Verilog HDL 程序结构

在传统的 Verilog HDL 书籍中，往往先讲述 Verilog HDL 语言的基本语法，直到最后才介绍 Verilog HDL 的程序结构，造成读者的学习没有明确的目的性，不知道掌握诸多的语法如何使用，不宜掌握知识点。因此本书以 Verilog HDL 程序结构的说明和层次化设计方法为首开启 Verilog HDL 语言的大门，使读者首先从宏观上了解 Verilog HDL，再深入到语法细节，明白各条语句出现的背景和使用方法，不仅可以避免出现“只见树林，不见山峰”的缺陷，还在后续的学习中通过实例快速验证语法，真切感受 Verilog HDL 的硬件描述魅力。

3.1 程序模块说明

3.1.1　Verilog HDL 模块的概念

模块（module）是 Verilog HDL 最基本的概念，也是最常用的基本单元，用于描述某个设计的功能或结构以及相同通信的接口。模块的实际意义是代表硬件电路上的逻辑实体，每个模块都实现特定的功能。

例如一个实现了 2 输入加法器的模块就对应着一个 2 输入加法电路，同样可以被所有需要实现 2 输入加法的模块调用。从而可见，模块对应着的硬件电路，其之间是并行运行的，也是分层的，高层模块通过调用、连接低层模块的实例来实现复杂的功能。如果要将所有的的功能模块连接成一个完整系统，则需要一个模块将所有的子模块连接起来，这一模块也被称为顶层模块（Top Module）。

在读者所熟悉的在 C 语言中，一个.c 文件中可以实现多个函数。和此类似，一个 Verilog HDL 文件（.v）也可以实现多个模块，但为了便于管理，一般建议一个.v 文件实现一个模块。需要注意的是，无论是面向综合的程序，还是面向仿真的程序，都需要以模块的形式给出，且模块的结构都是一致的，只存在语句上的差别。

3.1.2　模块的基本结构

一个 Verilog HDL 模块的完整结构如下所示：

```
module module_name (port_list)
    //声明各种变量、信号
    reg //寄存器
    wire//线网
    parameter//参数
    input//输入信号
    output/输出信号
    inout//输入输出信号
    function//函数
    task//任务
    ……
    //程序代码
    initial assignment
    always assignment
```

```
        module assignment
        gate assignment
        UDP assignment
        continous assignment
    endmodule
```

说明部分用于定义不同的项，例如模块描述中使用的寄存器和参数。语句用于定义设计的功能和结构。说明部分可以分散于模块的任何地方，但是变量、寄存器、线网和参数等的说明必须在使用前出现。

在实际中，一个 Verilog HDL 模块并不需要具备所有的结构特征，基本的模块结构已经能够满足大多数设计。下面给出模块的基本结构。

```
module <模块名> (<端口列表>)
    <定义>
    <模块条目>
endmodule
```

其中，<模块名>是模块唯一性的标志符；<端口列表>定义了和其余模块进行通信连接的信号，根据数据流方向，可以分为输入、输出和双向端口等三类；<定义>用来指定数据对象为寄存器型、存储器型、线型以及过程块。<模块条目>可以是 initial 结构、always 结构、连续赋值或模块实例。

下面给出一个简单的 Verilog HDL 模块，其实现了 3 线~8 线译码功能。

例 3-1：3 线~8 线译码器的 Verilog HDL 实现。

```
module decoder3to8(
    din, dout
    );
  input   [2:0] din;
  output [7:0] dout;
  reg      [7:0] dout;

  always @ (din) begin
    case(din)
        3'b000: dout <= 8'b0000_0001;
        3'b001: dout <= 8'b0000_0010;
        3'b010: dout <= 8'b0000_0100;
        3'b011: dout <= 8'b0000_1000;
        3'b100: dout <= 8'b0001_0000;
        3'b101: dout <= 8'b0010_0000;
        3'b110: dout <= 8'b0100_0000;
        3'b111: dout <= 8'b1000_0000;
      endcase
  end

endmodule
```

3.1.3 端口说明

模块端口是指模块与外界交互信息的接口，包括 3 种类型：

（1）input: 输入端口，模块从外界读取数据的接口，在模块内不可写。

（2）output：输出端口，模块往外界送出数据的接口，在模块内不可读。

（3）inout：输入输出端口，也成为双向端口，可读取数据也可以送出数据，数据可双向流动。

上述三类端口中，input 端口只能为线网型数据类型；output 端口可以为线网型，也可以为寄存器数据类型；而对于 inout 端口由于具备输入端口特点，所以也只能声明为线网型数据类型。（注：这里只是给出端口的简要说明，各类数据类型将在 4.3 节进行介绍）。

端口位宽由[M:N]来定义，如果 M>N，则其为降序排列，[M]位是有效数据的最高比特，[N]是有效数据的最低比特，其等效位宽为 M-N+1；如果 M<N，则其为升序排列，[M]位是有效数据的最低比特，[N]是有效数据的最高比特，其等效位宽为 N-M+1。下面给出一些常用的端口表示实例，例如：

```
output [15:0] crc_reg;
input [17 : 0] din;
input wr_en;
output [ 0: 17] dout;
```

3.2 Verilog HDL 的层次化设计

通过 2.2.2 节可以看出，层次化设计的核心思想有两个：一是模块化，二是模块例化（也就是程序调用）。本节主要介绍如何在 Verilog HDL 程序中实现模块调用和系统的层次化设计，并给出在 ISE 中如何与图形化设计结合的方法。

3.2.1 Verilog HDL 层次化设计的表现形式

层次化设计，简单来讲就是在利用 Verilog HDL 语言来编写程序实现相应功能时，不需要把所有的程序写在一个模块中。如果将系统中所有功能都放在一个模块中，那么其错误的检查、功能验证和调试的难度和复杂度将是无法想象的。

层次化设计方法的基本思想就是分模块、分层次地进行设计描述。描述系统总功能的设计为顶层设计，描述系统中较小单元的设计为底层设计。整个设计过程可理解为从硬件的顶层抽象描述向最底层结构描述的一系列转换过程，直到最后得到可实现的硬件单元描述为止。层次化的设计中所用的模块有两种：一是预先设计好的标准模块；二是由用户设计的具有特定应用功能的模块。

在基于 Verilog HDL 的层次化中，其最明显的表现特征就是具备多个不同级别的 Verilog HDL 代码模块，除顶层模块外，每个模块完成一项较为独立的功能。

3.2.2 模块例化

Verilog HDL 的模块例化也称作程序调用，指将已存在的 Verilog HDL 模块作为当前设计的一个组件，设计人员将其作为黑盒子看待，直接送给输入即可得到相应的输出信号。通过程序例化，可在顶层模块中，将各底层元件用 Verilog HDL 语言连接起来即可；逐次封装，形成最终的顶层文件，满足系统要求。

1．程序例化语法

在 Verilog HDL 语言中，有三种模块调用的方法：位置映射法、信号名映射法以及二者的混合映射法。

（1）位置映射法

位置映射法严格按照模块定义的端口顺序来连接，不用注明原模块定义时规定的端口名，其语法为：

模块名 例化名 (连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名,…);

下面给出一个位置映射法的例化实例。

例 3-2：给出利用位置映射法的 Verilog HDL 调用实例。

下面给出一个 4 输入的相等比较器，假设系统具有 4 个输入端口，分别为 a0、a1、b0 和 b1，要求判断 a0 和 b0 是否相等，a1 和 b1 是否相等。

通过分析可以发现，a0、b0 和 a1、b1 的比较是相同操作，因此只要实现一个相等比较器，然后调用两次即可达到设计目的。这样在验证时只需要验证比较器这一个子模块，从而达到简化设计的目的。

首先，给出相等比较器的 Verilog HDL 代码。

```
module compare_core(
      result, a , b
      );
 input    [7:0] a, b;
 output           result;

 //判断两输入是否相等，相等输出 1，否则输出 0
 assign result = (a == b) ? 1 : 0;

endmodule
```

其次，在应用的顶层模块中两次调用比较器子模块，其代码如下所列。

```
module compare_app0(
      result0, a0 , b0,
      result1, a1 , b1
      );
 input     [7:0] a0, b0, a1, b1;
 output           result0, result1;

 // 第 1 次调用比较器子模块，利用位置映射法
 compare_core inst_compare_core0(
      result0, a0,b0
      );

 // 第 2 次调用比较器子模块，利用位置映射法
 compare_core inst_compare_core1(
      result1, a1,b1
      );

endmodule
```

顶层模块在 ISE 软件中综合后的 RTL 级结构图如图 3-1 所示，可以看出，其两次调用了比较器子模块来达到设计目的。

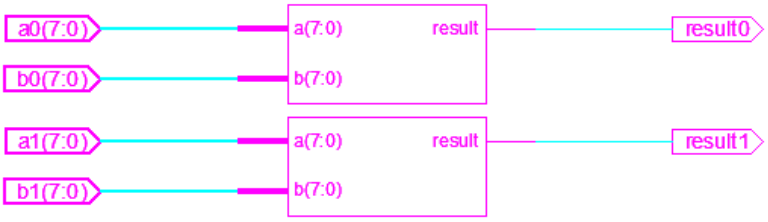


图 3-1　采用位置映射法的比较器应用模块 RTL 结构示意图

（2）信号名称映射法

信号名称映射法，即利用“.”符号，表明原模块定义时的端口名，其语法为：

模块名 例化名
(.端口 1 信号名(连接端口 1 信号名),
.端口 2 信号名(连接端口 2 信号名),
.端口 3 信号名(连接端口 3 信号名),…
);

显然，信号映射法同时将信号名和被引用端口名列出来，不必严格遵守端口顺序，不仅降低了代码易错性，还提高了程序的可读性和可移植性。因此，在良好的代码中，严禁使用位置映射法，全部采用信号映射法。

例 3-3：将例 3-2 的模块调用通过信号映射法实现。

```
module compare_app1(
      result0, a0 , b0,
      result1, a1 , b1
      );
 input    [7:0] a0, b0, a1, b1;
 output           result0, result1;

 //第 1 次调用比较器子模块，利用信号映射法
 compare_core inst_compare_core0(
      .result(result0),
      .a(a0),
      .b(b0)
      );

 //第 2 次调用比较器子模块，利用信号映射法
 compare_core inst_compare_core1(
      .a(a1),
      .b(b1),
      .result(result1)
      );

endmodule
```

上述程序在 ISE 中综合后的 RTL 级结构图如图 3-2 所示，可以看出其和图 3-1 是一致的，说明例 3-2 代码和例 3-1 代码是等效的。

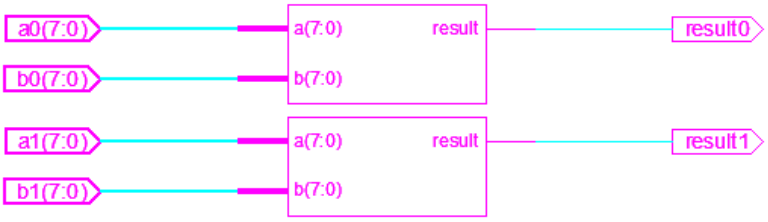


图 3-2 采用信号名映射法的比较器应用模块 RTL 结构示意图

2．特殊处理说明

例 3-1 和例 3-2 给出了基本完整的模块例化方法，但在实际中，还有大量的异常情况需要处理，包括部分输入、输出端口不用，某一端口位宽不匹配等异常情况。本节对这些特殊情况应用进行说明。

（1）悬空端口处理

在我们的实例化中，被调用模块的某些管脚可能不需要使用，因此要在例化时进行特殊处理。首先需要说明的，悬空例化模块端口只能在信号名映射法中来完成，其方法有两种：其一，在模块例化时，相应的端口映射中采用空白处理，如下列代码所示：

```
DFF d1 (
   .Q(QS),
   .Qbar ( ), //  该管脚悬空
   .Data (D ) ,
   .Preset ( ), //  该管脚悬空
   .Clock (CK)
); //信号名映射法
```

另一方法就是直接在例化时，不调用该端口，其示例代码如下：

```
DFF d1 (
   .Q(QS),
   //.Qbar ( ), //  该管脚悬空
   .Data (D ) ,
   //.Preset ( ), //  该管脚悬空
   .Clock (CK)
); //信号名映射法
```

需要说明的：在模块例化时，如果将输入管脚悬空，则该管脚输入为高阻 Z；如果将输出管脚悬空，则该输出管脚废弃不用。

（2）不同端口位宽的处理

模块例化的另一大类异常就是端口的位宽匹配问题，其处理原则如下：当模块例化端口和被例化模块端口的位宽不同时，端口通过无符号数的右对齐截断方式进行匹配。下面通过一个实例来说明上述处理原则。

例 3-4：给出 Verilog HDL 模块例化时端口位宽不匹配的处理实例。

被调用的子模块 Child 代码如下：

```
module Child (Pba, Ppy) ;
      input    [5:0] Pba;
      output [2:0] Ppy;

      assign Ppy[2] = Pba[5] | Pba[4];
      assign Ppy[1] = Pba[3] && Pba[2];
```

```
    assign Ppy[0] = Pba[1] | Pba[0];

endmodule
```

顶层模块 Top 的代码如下：

```
module Top(Bdl, Mpr);
    input   [1:2] Bdl;
    output [2:6] Mpr;

    //采用位置映射法例化模块 Child
    Child C1 (Bdl, Mpr) ;

endmodule
```

在对 Child 模块的实例中，根据位宽不匹配的异常处理原则：Bdl[2]连接到 Pba[0]，Bdl[1]连接到 Pba[1]，余下的输入端口 Pba[5]、Pba[4]和 Pba[3]悬空，因此为高阻态 z。与之相似，Mpr[6]连接到 Ppy[0]，Mpr[5]连接到 Ppy[1]，Mpr[4]连接到 Ppy[2]，如图 3-3 所示。

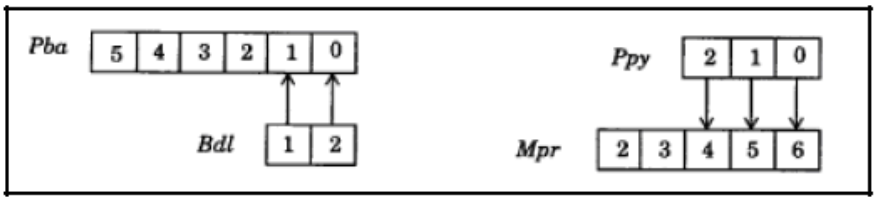


图 3-3　例 3-4 端口匹配示意图

那么在 EDA 软件操作中，究竟是什么样子呢？将上述代码添加到 ISE 中，将 Top.v 设置成顶层模块，综合后的 RTL 结构图如图 3-4 所示，单击选中 Child 模块的输入、输出端口信号线时，可在 ECS 页面查阅相应的信号线名称。例如，选中图 3-4 中的 Ppy(2:0)线网时，其指示信息如图 3-5 所示，可以看出其和图 3-3 的分析是一致的。

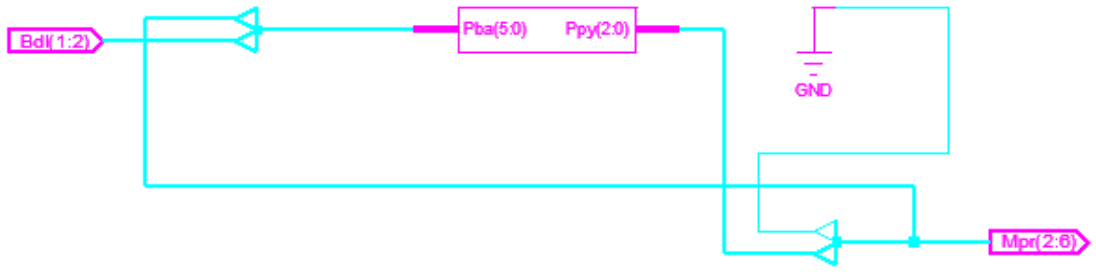


图 3-4　例 3-4 综合后的 RTL 级结构示意图

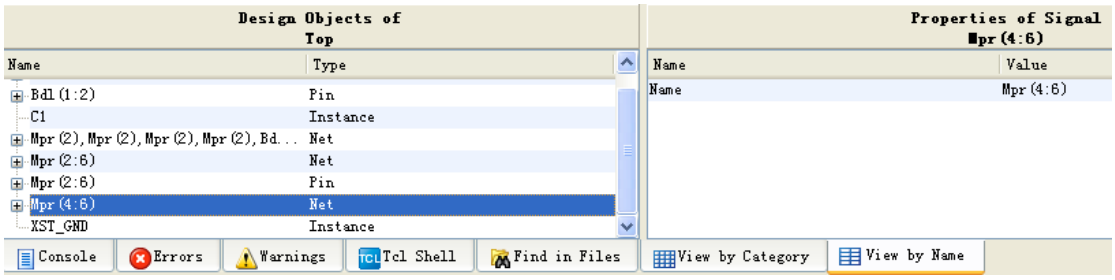


图 3-5　子模块输出网线信息示意图

3.2.3 参数映射

参数映射的功能就是实现参数化元件。所谓的“参数化元件”就是指元件的某些参数是可调的，通过调整这些参数从而可实现一类结构类似而功能不同的电路。在应用中，很多电路都可采用参数映射来达到统一设计，如计数器、分频器、不同位宽的加法器以及不同刷新频率的 VGA 视频接口驱动电路等。

（1）参数定义

在 Verilog HDL 中用 parameter 来定义参数，即用 parameter 来定义一个标志符表示一个固定的参数。采用该类型可以提高程序的可读性和可维护性。parameter 型信号的定义格式如下：

```
parameter  参数名 1 =  数据名 1;
```

下面给出几个例子：

```
parameter s1 = 1;
parameter [3:0] S0=4'h0,
                S1=4'h1,
                S2=4'h2,
                S3=4'h3,
                S4=4'h4;
```

参数值的作用域为声明所在的整个.v 文件，其数值可以在编译时被改变。改变参数值可以使用参数定义语句或通过在模块初始化语句中定义参数值。

（2）参数传递

参数传递就是在编译时对参数重新赋值而改变其值。传递的参数是子模块中定义的 parameter，其传递方法有下面两种。

- 使用“#”符号

在同一模块中使用“#”符号。参数赋值的顺序必须与原始模块中进行参数定义的顺序相同，并不是一定要给所有的参数都赋予新值，但不允许跳过任何一个参数，即使是保持不变的值也要写在相应的位置。格式如下：

```
module_name #( parameter1, parameter2) inst_name( port_map);
module_name #( .parameter_name(para_value), .parameter_name(para_value))
        inst_name (port map);
```

例 3-5：通过“#”字符实现一个模值可调的加 1 计数器。

顶层模块的代码如下：

```
module param_counter(
            clk_in, reset, cnt_out
    );
    input               clk_in;
    input               reset;
    output   [15:0] cnt_out;

    //参数化调用，利用#符号将计数器的模值 10 传入被调用模块
    cnt #(10) inst_cnt(
            .clk_in(clk_in),
            .reset(reset),
```

```
            .cnt_out(cnt_out)
    );

endmodule
```

被例化的参数化计数器的代码如下：

```
module cnt(
            clk_in, reset, cnt_out
    );
    //定义参数化变量
     parameter [15:0]Cmax = 1024;

     input              clk_in;
     input              reset;
     output   [15:0] cnt_out;

     reg        [15:0] cnt_out;

     //完成模值可控的计数器
     always @(posedge clk_in) begin
         if(!reset)
              cnt_out <= 0;
          else
              if(cnt_out == Cmax)
                   cnt_out <= 0;
               else
                   cnt_out <= cnt_out + 1;
     end
endmodule
```

整个程序实现不同模值的计数器，从结构上分为两部分：一部分就是计数器本身的功能实现部分，另一部分则是元件定义和参数化调用部分。计数器的实现包含 parameter 语句，在元件定义和调用时，通过“#”符号来传递参数。程序的仿真结果如图 3-6 所示，从中可以看出，计数器的最大值为 10，达到了参数化调用的要求。

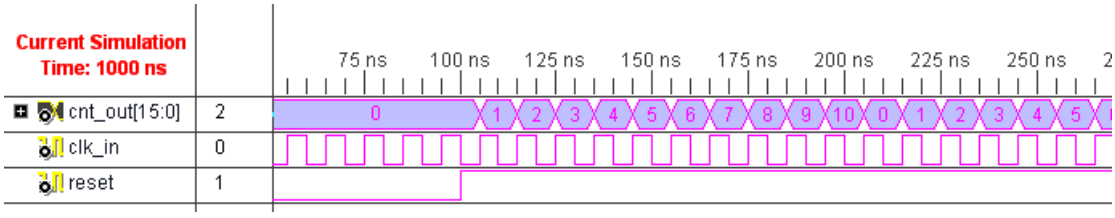


图 3-6　利用“#”符号的加 1 计数器仿真结果

- 使用 defparam 关键字

defparam 关键字可以在上层模块去直接修改下层模块的参数值，从而实现参数化调用，其语法格式如下：

```
defparam heirarchy_path.parameter_name = value;
```

这种方法与例化分开，参数需要写绝对路径来指定。参数传递时各个参数值的排列次序必须与被调用模块中各个参数的次序保持一致，并且参数值和参数的个数也必须相同。如

果只希望对被调用模块内的个别参数进行更改，所有不需要更改的参数值也必须按对应参数的顺序在参数值列表中全部列出（原值拷贝）。

使用 defparam 语句进行重新赋值时必须参照原参数的名字生成分级参数名。

例 3-6：通过“defparam”实现一个模值可调的加 1 计数器，要求其功能和例 3-5 一致。

```
module param_counter(
        clk_in, reset, cnt_out
);
input           clk_in;
input           reset;
output  [15:0] cnt_out;

//调用计数器子模块
cnt inst_cnt(
        .clk_in(clk_in),
        .reset(reset),
        .cnt_out(cnt_out)
);

//通过 defparam 参数指定例化模块的内部参数
defparam inst_cnt.Cmax = 12;

endmodule
```

程序的仿真结果如图 3-7 所示，从中可以看出，计数器的最大值为 12，达到了参数化调用的要求。

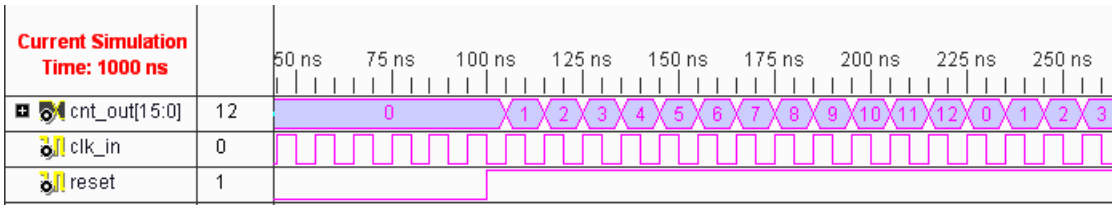


图 3-7　利用“defparam”符号的加 1 计数器仿真结果

3.2.4　在 ISE 中通过图形化方式实现层次化设计

随着设计规模的增大，原理图输入法已不太适合单独应用于 FPGA 设计中，但由于其具备直观清晰的特点，常和 HDL 设计混合使用，即通过 HDL 语言设计底层的复杂功能模块，而用其来构建顶层模块，类似于利用原理图绘制软件来设计整个系统。因此下面介绍如何通过原理图输入法来建立顶层模块。

1．建立用户设计的图形化表示符号

只有将 HDL 模块转化成图形化符号才能在原理图输入法中调用，ISE10.1 提供了上述转换的功能。在工程中建立 HDL 模块，完成 HDL 仿真测试以及综合后，用鼠标选中该模块，在过程管理区点击“Design Utilities”前面的“+”号，双击“Creat Schematic Symbol”命令，即可生成该模块的图形化符号，如图 3-8 所示。

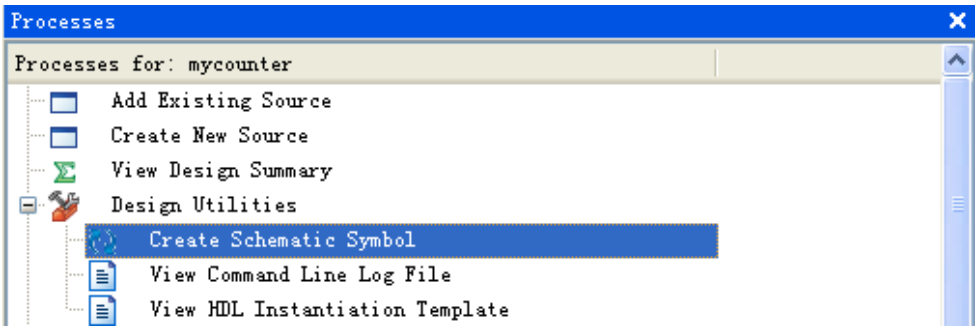


图 3-8　模块图形化符号生成操作示意图

2．利用原理图法构建顶层模块

（1）原理图设计法的输入界面

在当前工程中，新建“Schematic”（原理图）类型的文件，然后在工程区的“Source”页面双击该文件，可打开原理图设计页面，如图 3-9 所示。

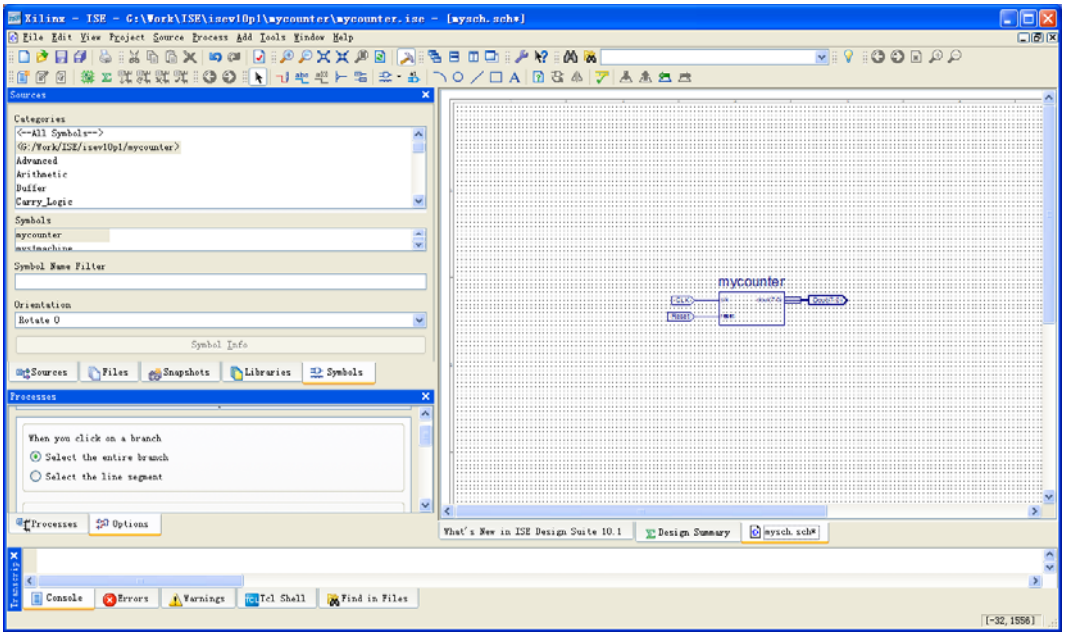


图 3-9　原理图设计页面

（2）原理图设计法的基本操作

原理图设计的元件库中包含了固有的图形化组件和用户自定义的图形化模块组件，前者包含了数字电路中所有的基本单元和 Xilinx 系列 FPGA 中集成的硬核模块，如与门、非门、加法器、复用器、乘法器、块 RAM 和 PowerPC 处理器等。在原理图混合设计中，最常用且不可缺少的是 I/O 端口组件，因为用户自定义的图形化模块是不包括 I/O 逻辑的，因此需要添加元件库中的 I/O 单元，才能构成完整的电路。

① 添加用户自定义模块

在“Categories”栏选择当前工程路径条目，则在“Symbols”栏会列出当前工程中用户自定义的所有图形化模块组件，单击鼠标左键选中目标，然后移动鼠标到设计区，会发现鼠标变成“+”，且附带着图形化单元，只需要在合适的位置点击鼠标左键，即可添加一个组件。添加完后，点击鼠标右键或按下键盘的“Esc”键，可将鼠标恢复正常。

② 添加 I/O 单元

添加 I/O 单元的方法和添加用户自定义模块的方法是类似的，只是 I/O 模块需要点击工

具栏的“ ”图标得到，且 I/O 单元必须在其余功能元件之后才能添加。当移动 I/O 单元到功能元件的管脚处出现“ ”图标时才能点击左键添加。

添加的 I/O 单元会自动根据元件管脚的方向属性调整为输入/输出，同时也会自动调整位宽。当然，在添加后，可通过双击 I/O 单元来修改管脚名称，相应的编辑界面如图 3-10 所示。

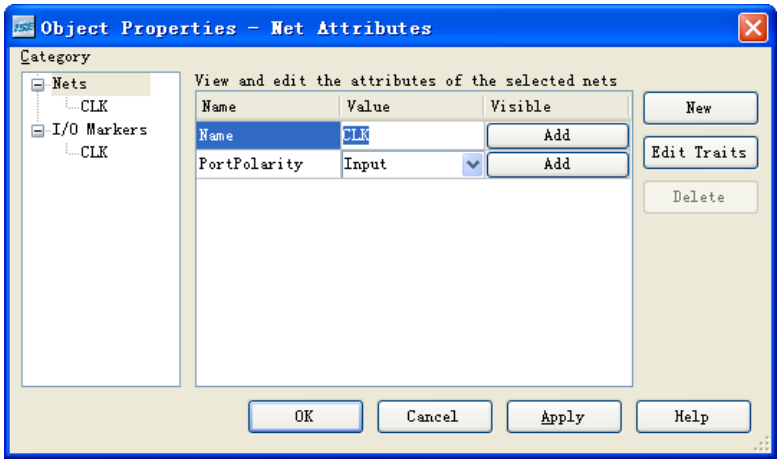


图 3-10　I/O 单元编辑界面

（3）应用实例

例 3-7：在原理图设计中调用例 3-5 的 16 比特计数器，形成完整设计。

① 在例 2-2 的基础上，完成综合，并在过程管理区选择“Design Utilities”下的“Creat Schematic Symbol”命令，生成计数器的图形化符号。

② 在 ISE 工程管理区的 Source 页面，新建“Schematic”源文件，并命名为 mysch。

③ 双击 mysch，进入原理图编辑页面，在“Categories”栏选择“G:/Work/ISE/isev10p1/mycounter”，然后在“Symbols”选择 mycounter，并将鼠标移动到原理图编辑区，点击左键添加。

④ 在工具栏选择“ ”，在 mycounter 模块的管脚上添加 3 个 I/O 单元。添加完毕后，双击 I/O 单元修改命令，如图 3-11 所示。

图 3-11　计数器的原理图设计

⑤ 保存原理图设计，点击右上角“ ”按键，关闭原理图编辑页面，返回工程管理区的“Source”页面。在 mysch 上单击右键，选择“Set as Top Module”命令，将其设置为顶层模块。

3.3 Verilog HDL 语言的描述形式

Verilog HDL 可以完成实际电路不同抽象级别的建模，具体而言有 3 种描述形式：如果从电路结构的角度来描述电路模块，则称为结构描述形式；如果对线型变量进行操作，就是数据流描述形式；如果只从功能和行为的角度来描述一个实际电路，就成为行为级描述形式。如前所述，电路具有 5 种不同模型（系统级、算法级、RTL 级、门级和开关级）。系统级、

算法级、RTL 级属于行为描述；门级属于结构描述；开关级涉及模拟电路，在数字电路中一般不考虑，其分类关系如图 3-12 所示。

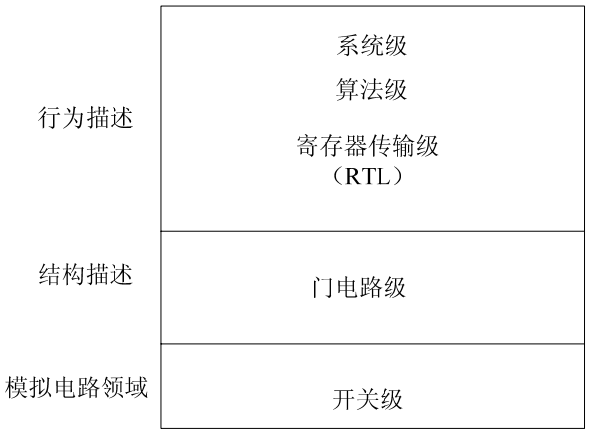


图 3-12　Verilog HDL 描述层次示意图

3.3.1　结构描述形式

Verilog HDL 中定义了 26 个有关门级的关键字，实现了各类简单的门逻辑。结构化描述形式通过门级模块进行描述的方法，将 Verilog HDL 预先定义的基本单元实例嵌入到代码中，通过有机组合形成功能完备的设计实体。在实际工程中，简单的逻辑电路由少数逻辑门和开关组成，通过门元语可以直观地描述其结构，类似于传统的手工设计模式。

Verilog HDL 语言提供了 12 个门级原语，分为多输入门、多输出门以及三态门三大类，如表 3-1 所列。

表 3-1　　门原语关键字说明列表

门级单元		
多输入门	多输出门	三态门
and	buf	bufif0
nand	not	bufif1
or		notif0
nor		notif1
xor		
xnor		

结构描述的每一句话都是模块例化语句，门原语是 Verilog HDL 本身提供的功能模块。其最常用的调用格式为：

门类型 <实例名> (输出，输入 1，输入 2，……,输入 N)

例如：

nand na01(na_out, a, b, c);

表示一个名字为 na01 的与非门，输出为 na_out，输入为 a, b, c。

1．多输入门原语

（1）and 门

and 门是二输入的与门，其逻辑结构如图 3-13 所示。

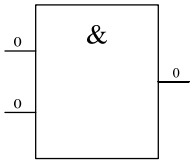


图 3-13　and 门的逻辑结构示意图

and 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-2 所列。

表 3-2　　and 门原语真值表

and		a			
		0	1	X	Z
b	0	0	0	0	0
	1	0	1	X	X
	X	0	X	X	X
	Z	0	X	X	X

（2）nand 门

nand 门是二输入的与非门，其逻辑结构如图 3-14 所示。

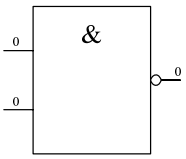


图 3-14　nand 门的逻辑结构示意图

nand 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-3 所列。

表 3-3　　nand 门原语真值表

nand		a			
		0	1	X	Z
b	0	1	1	1	1
	1	1	0	X	X
	X	1	X	X	X
	Z	1	X	X	X

（3）or 门

or 门是二输入的或门，其逻辑结构如图 3-15 所示。

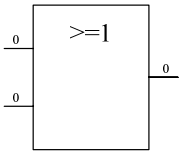


图 3-15　or 门的逻辑结构示意图

or 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-4 所列。

表 3-4　　or 门原语真值表

or		a			
		0	1	X	Z
b	0	0	1	X	X
	1	1	1	1	1
	X	X	1	X	X
	Z	X	1	X	X

（4）nor 门

nor 门是二输入的或非门，其逻辑结构如图 3-16 所示。

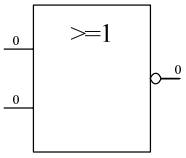


图 3-16　nor 门的逻辑结构示意图

nor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-5 所列。

表 3-5　　nor 门原语真值表

nor		a			
		0	1	X	Z
b	0	1	0	X	X
	1	0	0	0	0
	X	X	0	X	X
	Z	X	0	X	X

（5）xor 门

xor 门是二输入的异或门，其逻辑结构如图 3-17 所示。

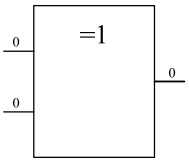


图 3-17　xor 门的逻辑结构示意图

xor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-6 所列。

表 3-6　　xor 门原语真值表

xor		a			
		0	1	X	Z
	0	0	1	X	X

b	1	1	0	X	X
	X	X	X	X	X
	Z	X	X	X	X

（6）xnor 门

xnor 门是二输入的异或非门，其逻辑结构如图 3-18 所示。

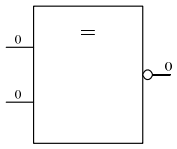


图 3-18　xnor 门的逻辑结构示意图

xnor 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-7 所列。

表 3-7　　xnor 门原语真值表

xnor		a			
		0	1	X	Z
b	0	1	0	X	X
	1	0	1	X	X
	X	X	X	X	X
	Z	X	X	X	X

2．多输出门原语

（1）buf 门

buf 门是单输入的数据延迟门，其逻辑结构如图 3-19 所示。

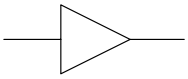


图 3-19　buf 门的逻辑结构示意图

buf 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-8 所列。

表 3-8　　buf 门原语真值表

输入	输出
0	0
1	1
X	X
Z	Z

（2）not 门

not 门是单输入的反相器，其逻辑结构如图 3-20 所示。

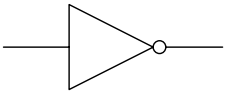


图 3-20　not 门的逻辑结构示意图

not 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-9 所列。

表 3-9　　not 门原语真值表

输入	输出
0	1
1	0
X	X
Z	Z

3．三态门原语

（1）bufif0 门（或门）

bufif0 门是单输入的三态门，控制端低有效，其其逻辑结构如图 3-21 所示。

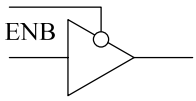


图 3-21　bufif0 门的逻辑结构示意图

bufif0 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-10 所列。

表 3-10　　bufif0 门原语真值表

bufif0		ctrl			
		0	1	X	Z
in	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

（2）bufif1 门（或门）

bufif1 门是单输入的三态门，控制端高有效，其逻辑结构如图 3-22 所示。

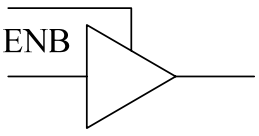


图 3-22　bufif1 门的逻辑结构示意图

bufif1 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-11 所列。

表 3-11　　bufif1 门原语真值表

bufif1		ctrl			
		0	1	X	Z
	0	Z	0	L	L

in	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

（3）notif0 门（或门）

notif0 门是单输入的三态反相器，控制端低有效，其其逻辑结构如图 3-23 所示

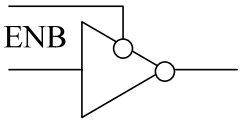


图 3-23　notif0 门的逻辑结构示意图

notif0 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-12 所列。

表 3-12　　notif0 门原语真值表

notif0		ctrl			
		0	1	X	Z
in	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

（4）notif1 门（或门）

notif1 门是单输入的三态反相器，控制端高有效，其其逻辑结构如图 3-24 所示。

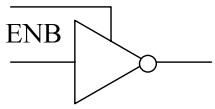


图 3-24　notif1 门的逻辑结构示意图

notif1 门的输入端口是对等的，无位置、优先级等区别，假设其名称分别为 a、b，则其真值表如表 3-13 所列。

表 3-13　　notif1 门原语真值表

notif1		ctrl			
		0	1	X	Z
in	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

基于门原语的设计，要求设计者首先将电路功能转化成逻辑组合，再搭建门原语来实现，是数字电路中最底层的设计手段。下面给出一个基于门原语的全加器设计实例。

例 3-8：利用 Verilog HDL 的一个单比特全加器。

```
module ADD(A, B, Cin, Sum, Cout);
```

```
    input A, B, Cin;
    output Sum, Cout;
    // 声明变量
    wire S1, T1, T2, T3;

    //调用两个或非门
    xor X1 (S1, A, B),
        X2 (Sum, S1, Cin);

    //调用 3 个与门
    and A1 (T3, A, B),
        A2 (T2, B, Cin),
        A3 (T1, A, Cin);

    //调用一个或门
    or O1 (Cout, T1, T2, T3);

endmodule
```

在这一实例中，模块包含门的实例语句，也就是包含内置门 xor、and 和 or 的实例语句。图 3-25 给出全加器的连接结构示意图，可以看出门实例由线网型变量 S1、T1、T2 和 T3 互连，和代码语句结构相同。由于未指定顺序，门实例语句可以以任何顺序出现。

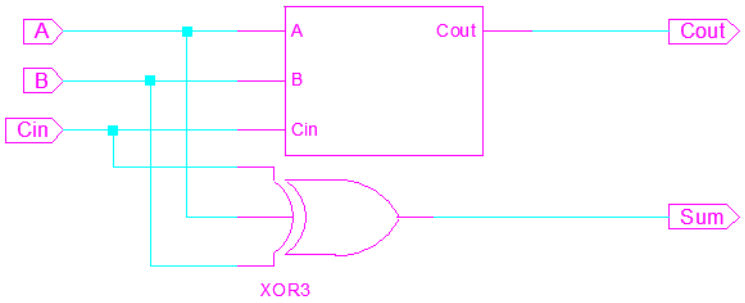


图 3-25　全加器的连接结构示意图

其中，双击完成 A、B 和 Cin 相加的模块，可以看到图 3-26 所示的内部结构。

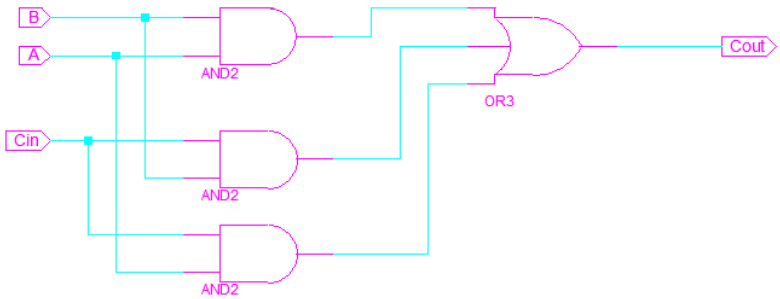


图 3-26　全加器的结构示意图

在 ISE Simulator 中运行上述程序，得到的仿真结果如图 3-27 所示，从中可以其功能是正确的。为什么上述门原语的组合能实现一个单比特的全加器，请读者自行分析其原理。

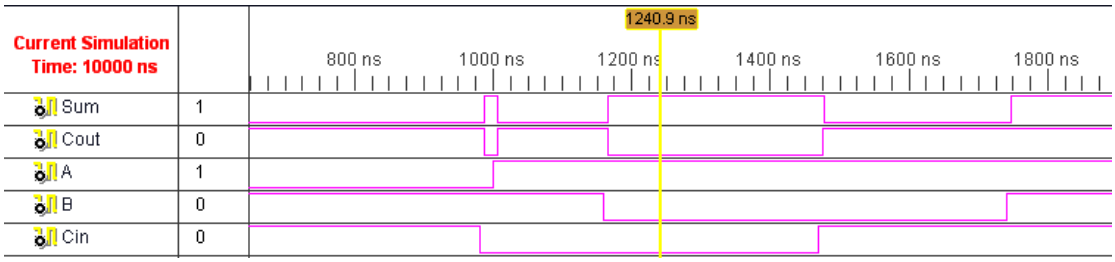


图 3-27　基于门结构的全加器仿真结果

门级描述本质上也是一种结构网表，具备较高的设计性能（资源、速度性能）。读者在实际中的使用方法为：先使用门逻辑构成常用的触发器、选择器、加法器等模块，再利用已经设计的模块构成更高一层的模块，依次重复几次，便可以构成一些结构复杂的电路。其缺点是：不易管理，难度较大且需要一定的资源积累。

在 90 年代中期以前，由于可编程逻辑器件的资源很少，大都为目前主流器件的千分之一到百分之一左右的规模；再加上相关的设计都比较简单，因此结构化描述有着广泛的应用。至此以后，特别是近十年，由于半导体器件规模和系统设计规模的飞速增长，这一传统且古老的设计方式已被彻底弃用。事实上，通过本书后续内容的学习，没有人还愿意书写或阅读类似于例 3-8 的代码。

3.3.2　行为描述形式

行为型描述主要包括语句/语句块、过程结构、时序控制、流控制等 4 个方面，是目前 Verilog HDL 中最重要的描述形式。

1．语句块

语句就是各条 Verilog HDL 代码。语句块就是位于 begin……end/fork……join 块定义语句之间的一组行为语句，将满足某一条件下的多条语句标记出来，类似于 C 语言中“{}”符号中的内容。

语句块可以有独立的名字，写在块定义语句的第一个关键字之后，即 begin 或 fork 之后，可以唯一地标识出某一语句块。如果有了块名字，则该语句块被称为一个有名块。在有名块内部可以定义内部寄存器变量，且可以使用“disable”中断语句中断。块名提供了唯一标识寄存器的一种方法，如例 3-9 所示。

例 3-9：语句块使用例子。

```
always @ (a or b )
begin : adder1     //adder1 为语句块说明语句
        c = a + b;
end
```

定义了一个名为 adder1 的语句块，实现输入数据的相加。

语句块按照界定不同分为两种：

（1）串行 begin…end

begin…end 块用来组合需要顺序执行的语句，因此被称为串行块。例如下面的语句：

```
reg[7:0] r;
begin                          //由一系列延迟产生的波形
      r = 8'h35  ;          //语句 1
      r = 8'hE2  ;          //语句 2
      r = 8'h00  ;          //语句 3
      r = 8'hF7  ;          //语句 4
```

end

其执行顺序就是，首先执行语句 1，将 8'h35 赋给变量 r；再执行语句 2，将 8'hE2 再次赋给变量 r，覆盖语句 1 所赋的值；…；最后，将 8'hF7 赋给变量 r，形成其最终的值。

串行块的执行特点如下：

- 串行块内的各条语句是按它们在块内的语句逐次逐条顺序执行的，当前一条执行完之后，才能执行下一条。例如，上例中语句 1 至语句 4 是顺序执行的。
- 块内每一条语句中的延时控制都是相对于前一条语句结束时刻的延时控制。例如，上例中语句 2 的时延为 2d。
- 在进行仿真时，整个语句块总的执行时间等于所有语句执行时间之和。如上例中语句块中总的执行时间为 4d。
- 在可综合语句中，begin……end 块内的语句在时序逻辑中本质上是并行执行的，和语句的书写顺序无关，其中原因将在 8.8 节进行介绍。不过，读者可以从 EDA 设计的本质去简单理解，可综合 Verilog HDL 语句描述的是硬件电路，数字电路的各个硬件组成部分就是并列工作的（类似于 PC 机的声卡和显卡就是同时工作的，用户可以同时听到声音，并欣赏图像）。

（2）并行 fork…join

fork…join 用来组合需要并行执行的语句，被称为并行块。例如：

```
parameter d = 50;
reg[7:0] r1, r2, r3, r4;
fork                          //由一系列延迟产生的波形
    r1 = ' h35  ;             //语句 1
    r2 = ' hE2  ;             //语句 2
    r3 = ' h00  ;             //语句 3
    r4 = ' hF7  ;             //语句 4
join
```

并行块的执行特点为：

- 并行语句块内各条语句是各自独立地同时开始执行的，各条语句的起始执行时间都等于程序流程进入该语句块的时间。如上例中语句 2 并不需要等语句 1 执行完才开始执行，它与语句 1 是同时开始的。
- 块内每一条语句中的延时控制都是相对于程序流程进入该语句块的时间而言的。
- 在进行仿真时，整个语句块总的执行时间等于执行时间最长的那条语句所需要的执行时间。

需要说明的是：其中 begin...end 块是可综合语句，其串行执行特点，是从语法结构上讲的。在实际电路中，各条语句之间并不全是串行的，这一点是 Verilog HDL 设计思想的难点之一，本书将在 8.8.1 节对其进行介绍。至于 fork…join，则是不可综合的，更多的用于仿真代码中。

2．过程结构

过程结构采用下面 4 种过程模块来实现，具有强的通用型和有效性。

- initial 模块
- always 模块
- 任务（task）模块
- 函数（function）模块

一个程序可以有多个 initial 模块、always 模块、task 模块和 function 模块。initial 模块和 always 模块都是同时并行执行的，区别在于 initial 模块只执行一次，而 always 模块则是

不断重复地运行。initial 模块是不可综合的，常用于仿真代码的变量初始化中；always 模块则是可综合的。

下面给出 initial 模块和 always 模块的说明，任务和函数将在 5.4 节进行介绍。

（1）initial 模块

在进行仿真时，一个 initial 模块从模拟 0 时刻开始执行，且在仿真过程中只执行一次，在执行完一次后，该 initial 就被挂起，不再执行。如果仿真中有两个 initial 模块，则同时从 0 时刻开始并行执行。

initial 模块是面向仿真的，是不可综合的，通常被用来描述测试模块的初始化、监视、波形生成等功能。其格式为：

```
initial begin/fork
    块内变量说明
    时序控制 1    行为语句 1；
    ……
    时序控制 n    行为语句 n；
end/join
```

其中，begin……end 块定义语句中的语句是串行执行的，而 fork……join 块语句中的语句定义是并行执行的。当块内只有一条语句且不需要定义局部变量时，可以省略 begin……end/fork……join。

例 3-10：下面给出一个 initial 模块的实例。

```
initial begin
// 初始化输入向量
    clk = 0;
    ar = 0;
    ai = 0;
    br = 0;
    bi = 0;
    // 等待 100 个仿真单位，全局 reset 信号有效
    // 其中#为延迟控制语句，其使用方法见 6.2 节
    #100;
    ar = 20;
    ai = 10;
    br = 10;
    bi = 10;
end
```

（2）always 模块

和 initial 模块不同，always 模块是一直重复执行的，并且可被综合。always 过程块由 always 过程语句和语句块组成的，其格式为：

```
always @ (敏感事件列表) begin/fork
    块内变量说明
    时序控制 1    行为语句 1；
    ……
    时序控制 n    行为语句 n；
end/join
```

其中，begin……end/fork……join 的使用方法和 initial 模块中的一样。敏感事件列表是

可选项，但在实际工程中却很常用，而且是比较容易出错的地方。敏感事件表的目的就是触发 always 模块的运行，而 initial 后面是不允许有敏感事件表的。

敏感事件表由一个或多个事件表达式构成，事件表达式就是模块启动的条件。当存在多个事件表达式时，要使用关键词 or 将多个触发条件结合起来。Verilog HDL 的语法规定：对于这些表达式所代表的多个触发条件，只要有一个成立，就可以启动块内语句的执行。例如，在语句

```
always@ (a or b or c) begin
    ……
end
```

中，always 过程块的多个事件表达式所代表的触发条件是：只要 a、b、c 信号的电平有任意一个发生变化，begin……end 语句就会被触发。

always 模块主要是对硬件功能的的行为进行描述，可以实现锁存器和触发器等基本数字处理单元，也可以用来实现各类大规模设计。

例 3-11：下例给出一个 always 模块的应用示例。

```
module and3(f, a, b, c);
    input a, b, c;
    output f;
    reg f;

    always @(a or b )begin
        f = a & b & c;
    end
endmodule
```

3．时序控制

Verilog HDL 提供了两种类型的显示时序控制，一种是延迟控制，在这种类型的时序控制中通过表达式定义开始遇到这一语句和真正执行这一语句之间的延迟时间。另外一种是事件控制，这种时序控制是通过表达式来完成的，只有当某一事件发生时才允许语句继续向下执行。

一般来讲，延时控制语句是不可综合的，常用于仿真，第 6.3 节将给出其具体使用说明；而事件控制是可综合，通过 always 语句来实现，分为电平触发和信号跳变沿触发两类。其描述方式将在 8.1 节详细介绍。

4．流控制

流控制描述一般都采用 assign 连续赋值语句来实现，主要用于完成简单的组合逻辑功能。连续赋值语句右边所有的变量受持续监控，只要这些变量有一个发生变化，整个表达式被重新赋值给左端。其语法格式如下：

```
assign   L_s = R_s;
```

例 3-12：一个利用数据流描述的移位器。

```
module mlshift2(a, b);
    input a;
    output b;

    //数据流描述语句，使用 assign 关键字
    assign b = a<<2;
```

```
endmodule
```

在上述模块中，只要 a 的值发生变化，b 就会被重新赋值，所赋值为 a 左移两位后的值。

3.3.4 混合设计模式

在 Verilog HDL 模块中，结构描述、行为描述可以自由混合。也就是说，模块描述中可以包括实例化的门、模块实例化语句、连续赋值语句以及行为描述语句的混合，它们之间可以相互包含。使用 always 语句和 initial 语句（切记只有寄存器类型数据才可以在这两个模块中赋值）来驱动门和开关，而来自于门或连续赋值语句（只能驱动线网型）的输出能够反过来用于触发 always 语句和 initial 语句。

例 3-13：利用 Verilog HDL 语言完成一个与非门混合设计。

```
module hunhe_demo(
    A, B, C
    );
     input    A, B;
     output C;

     //定义中间变量
     wire     T;

     //调用结构化与门
     and A1 (T, A, B);

     //通过数据流形式对与门输出求反，得到最终的与非门结果
     assign C = ~T;

endmodule
```

3.4 本章小结

本章主要介绍了 Verilog HDL 的程序结构，包括模块说明、端口说明，读者需要掌握其中哪些因素是必不可少的。其次，介绍了模块例化的两种方法，分别为信号名映射法和位置映射法，以及如何通过“#”和“defparam”传递参数到模块中。在实际中，自顶向下的层次化设计模式就是通过模块例化完成的。此外，ISE 提供了将 Verilog HDL 模块生成图形化模块的方法，便于和图形化输入相结合，具有直观、清楚的特点。最后，本章初步介绍了 Verilog HDL 的两大类描述形式：结构化描述和行为化描述，其中行为化描述具备更高的设计效率，是最常用的，也是 RTL 级编码的主要手段，读者应该加强行为描述的学习。

3.5 思考题

1．一个完整的 Verilog HDL 程序包含哪些部分，其中哪些部分是必须的？
2．解释 Verilog HDL 的模块（module）的概念，指明其与传统的软件函数的且别。
3．在 Verilog HDL 模块中，端口可以分为哪几类？各有什么特点？
4．如何通过 Verilog HDL 语言完成模块例化，有哪几种方法？

5．有哪几种方法可以实现 Verilog HDL 子模块的参数化配置？
6．在 ISE 软件中，如何生成一个模块的图形化表示符号？
7．Verilog HDL 有哪几种描述形式？
8．什么是结构化描述方式？有什么特点，给出一个结构化描述的简单示例。
9．什么是行为化描述方式？有什么特点，给出一个结构化描述的简单示例。
10．结构化描述能和行为化描述混合使用吗？

第 4 章　Verilog HDL 语言基本要素

Verilog HDL 语言是一种严格的数据类型化语言，规定每一个变量和表达式都要有唯一的数据类型。数据类型需要静态确定，一旦确定不能再在设计中改变。和其他语言一样，Verilog HDL 语言具有多种丰富的数据类型。此外，Verilog HDL 语言具有大量的运算操作符，给设计带来很大的灵活性。本章主要介绍 Verilog HDL 语言的基本要素，包括标志符、数字、数据类型、运算符和表达式等。虽然，这些基本要素和 C 语言具有相同之处，但也有 Verilog HDL 作为一种硬件描述语言所特有的地方，如数据类型 wire、reg 等。帮助读者掌握并理解 HDL 语言与软件描述语言（C 语言等）的区别是本书的核心思想之一。

4.1 标志符与注释

4.1.1　标志符

标志符是赋给对象的唯一名称，通过标志符可以提及相应的对象。标志符可以是一组字母、数字、_下划线和$符号的组合，且标志符的第一个字符必须是字母或者下划线。另外，标志符是区别大小写的。下面给出标志符的几个例子：

```
Clk_100MHz
diag_state
_ce
P_o1_02
```

转义标识符（Escaped identifier）可以在一条标识符中包含任何可打印字符。转义标识符以\ (反斜线)符号开头，以空白结尾（空白可以是一个空格、一个制表字符或换行符）。下面例举了几个转义标识符：

```
\fg00
\.*.$
\{*+5***}
\~Q
\Verilog                    //与 Verilog 相同。
```

最后一个例子说明了在一条转义标志符中，反斜线和结束空格并不是转义标志符的一部分，因此转义标志符\Verilog 和标志符 Verilog 是恒等的。

Verilog HDL 语言预定义了一系列非转义标志符的保留字来说明语言结构，叫作关键字，附录 1 给出了 Verilog HDL 语言中所有的关键字。需要注意的是：标志符不能和关键字重复。只有小写的关键字才是保留字，因此在实际开发中，建议将不确定是否是保留字的标志符首字母大写。例如：标志符 if（关键字）与标志符 IF 是不同的。需要注意的是，转义标志符与关键字是不同的，例如标志符\initial（非关键词）与标志符 initial（关键词）就是不同的。

4.1.2　注释

在 Verilog HDL 中有两种形式的注释：

（1）一种是以“/*”符号开始，“*/”结束，在两个符号之间的语句都是注释语句，因

此可扩展到多行。如：

/*第一种形式：
可以扩展至
多行 */

以上 3 行语句都是注释语句。

（2）另一种是以“//”开头的语句，它表示以“//”开始到本行结束都属于注释语句。如：

//第二种形式：在本行结束

4.2 数字与逻辑数值

4.2.1 逻辑数值

Verilog HDL 有下列 4 种基本的逻辑数值：

0：逻辑 0 或“假”；
1：逻辑 1 或“真”；
x：未知；
z：高阻。

其中 x、z 是不区分大小写的。Verilog HDL 中的数字由这四类基本数值表示。在 Verilog HDL 语言中，表达式和逻辑门输入中的'z'通常解释为'x'。

4.2.2 常量

Verilog HDL 中的常量分为 3 类：整数型、实数型以及字符串型。下划线符号“_”可以随意用在整数和实数中，没有实际意义，只是为了提高可读性。例如：56 等效于 5_6。

1．整数

整数型在 Verilog HDL 语言设计中最常用的一类常量，可以按如下两种方式书写：简单的十进制数格式以及基数格式。

（1）简单的十进制格式

简单的十进制数格式的整数定义为带有一个“+”或“-”操作符的数字序列。下面是这种简易十进制形式整数的例子。

45　　十进制数 45
-46　　十进制数-46

简单的十进制数格式的整数值代表一个有符号的数，其中负数可使用两种补码形式表示。例如，32 在 6 位二进制形式中表示为 100000，在 7 位二进制形式中为 0100000，这里最高位 0 表示符号位；-15 在 5 位二进制中的形式为 10001，最高位 1 表示符号位，在 6 位二进制中为 110001，最高位 1 为符号扩展位。

（2）基数表示格式

基数格式的整数格式为：

[长度] '基数 数值

长度是常量的位长，基数可以是二进制、十进制、十六进制之一。数值是基于基数的数字序列，且数值不能为负数。下面是一些具体实例：

6'b9　　6 位二进制数
5'o9　　5 位八进制数
9'd6　　9 位十进制数

2．实数
实数可以用下列两种形式定义：
（1）十进制计数法，例如：

2.0
16539.236

（2）科学计数法
这种形式的实数举例如下，其中 e 与 E 相同。

235.12e2　　其值为 23512
5e-4　　其值为 0.0005

根据 Verilog 语言的定义，实数通过四舍五入隐式地转换为最相近的整数。
3．字符串
字符串是双引号内的字符序列。字符串不能分成多行书写。例如：

"counter"

用 8 位 ASCII 值表示的字符可看作是无符号整数，因此字符串是 8 位 ASCII 值的序列。为存储字符串“counter”，变量需要 8×7 位。

reg [1: 8*7] Char;
Char = "counter";
需要注意的是，ISE 软件是不支持字符串的，因此本书后续内容中不会出现字符串的有关应用。

4.2.3　参数

参数是一个特殊的常量，经常用于定义时延和变量的宽度。使用参数说明的参数只被赋值一次。参数说明形式如下：
parameter param1 = const_expr1, param2 = const_expr2, . . . ,
paramN = const_exprN;
下面给出一些具体实例：
parameter LINELENGTH = 132, ALL_X_S = 16'bx;
parameter BIT = 1, BYTE = 8, PI = 3.14;
parameter STROBE_DELAY = (BYTE + BIT) / 2;
parameter TQ_FILE = " /home/bhasker/TEST/add.tq";
参数值也可以在编译时被改变。改变参数值可以使用参数定义语句或通过在模块初始化语句中定义参数值。

4.3 数据类型

4.3.1　数据类型综述

数据类型用来表示数字电路硬件中的数据存储和传送元素。Verilog HDL 中总共有两大

类数据类型：线网类型和寄存器类型。

线网类型主要表示Verilog HDL中结构化元件之间的物理连线，其数值由驱动元件决定；如果没有驱动元件连接到线网上，则其缺省值为高阻'z'。寄存器类型主要表示数据的存储单元，其缺省值为不定'x'。二者最大的区别在于：寄存器类型数据保持最后一次的赋值，而线网型数据则需要持续的驱动。

4.3.2 线网类型

线网型数据常用来表示以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输入、输出信号类型默认为 wire 型。wire 型信号可以用做方程式的输入，也可以用做“assign”语句或者实例元件的输出。

线网数据类型包含下述不同种类的线网子类型，其中只有 wire、tri、supply0 和 supply1 是可综合的，其余都是不可综合的，只能用于仿真语句。需要特别指出的是，wire 是最常用的线网型变量。

- wire：标准连线（缺省为该类型）；
- tri： 具备高阻状态的标准连线；
- wor：线或类型驱动；
- trior：三态线或特性的连线；
- wand 线与类型驱动
- triand：三态线与特性的连线；
- trireg：具有电荷保持特性的连线；
- tri1：上拉电阻（pullup）；
- tri0：下拉电阻（pulldown）；
- supply0：地线，逻辑 0；
- supply1：电源线，逻辑 1。

线网数据类型的通用说明语法为：

net_kind [msb:lsb] net1, net2, . . . , netN;

其中，net_kind 是上述线网类型的一种。msb 和 lsb 是用于定义线网范围的常量表达式；范围定义是可选的；如果没有定义范围，缺省的线网类型为 1 位。下面给出一些线网类型说明实例：

```
wire [7:0] data1, data2;      //两个 8 比特位宽的线网
wire       ce;                //1 个 1 比特位宽的线网
```

线网类型变量的赋值（也就是驱动）只能通过数据流“assign”操作来完成，不能用于 always 语句中，如：

```
assign ce = 1'b1;
```

1．wire 线网

wire 线网是最常用的线网型数据类型，Verilog HDL 模块（module）的输入/输出端口的缺省值就是 wire 型。wire 型信号可以作为任何表达式的输入，也可以用作“assign”语句和模块例化的输出。wire 型信号的取值可以为 0、1、x、z。

根据 Verilog HDL 语法，wire 型变量可以有多个驱动源。表 4-1 给出了两个驱动源驱动同一根 wire 线网的真值表。

表 4-1 两驱动线网的真值表

wire	0	1	x	z
0	0	x	x	0
1	x	1	x	1

x	x	x	x	x
z	0	1	x	z

下面给出一个 wire 型变量多驱动的应用实例。

例 4-1： wire 型变量两驱动演示实例。

```
module net_demo(
        a, b, c
       );
        input    a, b;
        output c;

        wire      temp;
        assign temp = a;
        assign temp = b;
        assign c = temp;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 4-1 所示。在这个实例中，temp 有两个驱动源。两个驱动源的值（右侧表达式的值）用于在表 4-2 中索引，以便决定 temp 的有效值。从图 4-1 中可以看出，其相应的仿真结果和表 4-2 的结果实=是一致的。

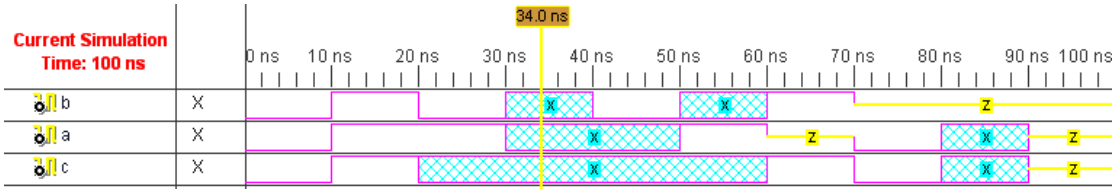


图 4-1　例 4-1 的仿真结果示意图

虽然，Verilog HDL 语法规定可以对 wire 型变量有多个驱动，但其仅用于仿真程序中。在实际电路，对任何信号有多个驱动都会造成一些不确定的后果，因此在面向综合的设计中，对任何变量连接多个驱动都是错误。如果在 ISE 中对例 4-1 的程序进行综合，则会给出图 4-2 所示的错误。因此，在面向综合的程序中，不要出现任何形式的多驱动代码。

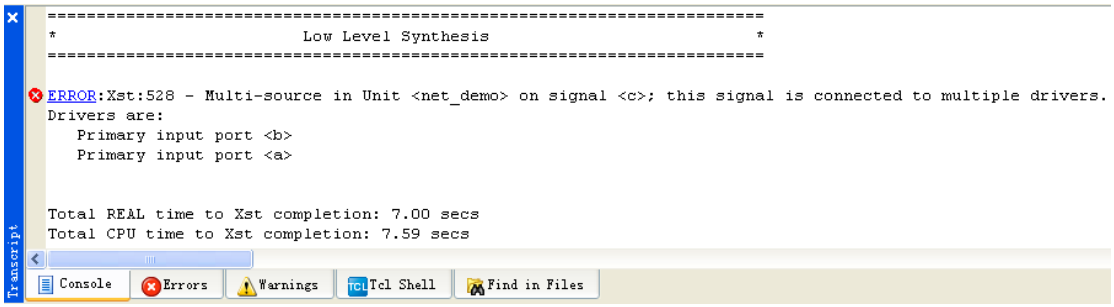


图 4-2　例 4-1 的综合提示错误

2．tri 线网

在 Verilog HDL 语言的定义中，tri 与 wire 的功能是完全一致的，唯一的差别就是名称书写上的不同。提供这两种不同名称的作用只是为了增加可读性。例如，为了强调总线具有高阻态的特征，将其命名为 tri 型，以便与普通的 wire 连线加以区别。事实上，wire 型也具备描述信号的高阻特征。如：

tri [7 : 0] Addr;

同样，tri 信号也可以有多个驱动源，其真值表和 wire 类型一致，参见表 4-1。下面给出一个 tri 线网应用实例。

例 4-2：tri 线网应用实例。

```
module Tristate (in, oe, out);
    input    in, oe;
    output   out;
    tri      out;

    bufif1   b1(out, in, oe);

endmodule
```

3．wor 和 trior 线网

wor 线网和 trior 线网专门用于单信号多驱动，如果某个驱动源为 1，那么线网的值也为 1，因此 wor 被称为线或类型，trior 被称为三态线或类型，二者在语法和功能上是一致的，其关系类似于 wire 和 tri，并无本质区别，仅仅为了增加可读性，trior 用于表征高阻状态。其定义示例如：

wor [10 : 4] A;

trior [3: 0] B, C, D;

如果多个驱动源驱动 wor 线网和 trior 线网，其有效值由表 4-2 决定。

表 4-2　　wor 线网和 trior 线网驱动的真值表

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

wor 线网和 trior 线网只能用于仿真，而不能用于综合代码。

4．wand 线网和 triand 线网

wand 线网和 triand 线网也是专门用于多驱动源情况，如果某个驱动源为 0，那么线网的值为 0，因此 wand 线网被称为线与类型，triand 线网被称为三态线与类型。二者语法和功能上是一致的。triand 仅用于从名称上表征高阻特征。其定义示例如：

wand [-7 : 0] Dbus;

triand Reset, Clk;

如果 wand 线网和 triand 线网存在多个驱动源，其有效值由表 4-3 决定。

表 4-3　　wand 线网和 triand 线网驱动的真值表

wor/trior	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

同样，wand 线网和 triand 线网只能用于仿真，而不能用于综合代码。

5．trireg 线网

trireg 线网用于存储数值（类似于寄存器）以及电容节点的建模。当三态寄存器（trireg）

的所有驱动源都处于高阻态，也就是说输入值为 z 时，三态寄存器线网将保存作用在线网上的最后一个值。此外，三态寄存器线网的缺省初始值为 x。

trireg [1:8] Dbus, Abus;

trireg 线网只能用于仿真，而不能用于综合代码。下面给出 trireg 线网的应用示例。

例 4-3：给出 Verilog HDL 语言中 trireg 线网的应用示例。

```
module tb_trireg;
    trireg   [7:0] data;
    reg      [1:0] flag;

    initial begin
       flag = 1;
       #200;
       flag = 0;
       #200;
       flag = 3;
       #200;
       flag = 0;
       #200;
       flag = 2;
       #200;
       flag = 0;
   end

   assign data = (flag==1) ? 10 : (flag == 0) ? 8'hzz : (flag ==3) ? 30 : 255;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 4-3 所示，可以看出 trireg 类型的变量在仿真时可以保持上一次非高阻驱动的数值。

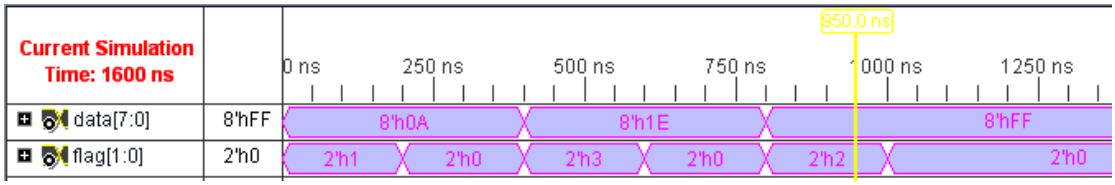


图 4-3　例 4-3 仿真结果

6．tri0 线网和 tri1 线网

tri0 线网和 tri1 线网用于线逻辑的建模，即线网有多于一个驱动源。tri0（tri1）线网的特征是，若无驱动源驱动，它的值为 0（tri1 的值为 1）。其示例定义语句为：

tri0 [0:3] D;

tri1 [0:5] B, C, A;

表 4-4 给出了在多个驱动源情况下 tri0 线网或 tri1 线网的有效值。

表 4-4　　wand 线网和 triand 线网驱动的真值表

tri0(tri1)	0	1	x	z
0	0	x	x	0

1	x	1	x	1
x	x	x	x	x
z	0	1	x	0(1)

7．supply0 和 supply1 线网

supply0 用于对“地”建模，即低电平 0；supply1 网用于对电源建模，即高电平 1。其声明示例为:

```
supply0 Gnd_FPGA;
supply1 [2:0] Vcc_Bank;
```

4.3.3 寄存器类型

寄存器型变量，都有“寄存”性，即在接受下一次赋值前，将保持原值不变。寄存器型变量没有强度之分，且所有寄存器类变量都必须明确给出类型说明（无缺省状态）。寄存器数据类型包含下列 4 类数据类型。

- reg：常用的寄存器型变量。用于行为描述中对寄存器类的说明，由过程赋值语句赋值；
- integer： 32 位带符号整型变量；
- time： 64 位无符号时间变量；
- real： 64 位浮点、双精度、带符号实型变量。
- realtime：其特征和 real 型变量一致。

1．reg 寄存器类型

寄存器数据类型 reg 是最常见的数据类型。reg 类型使用保留字 reg 加以说明，形式如下：

```
reg [ msb: lsb] reg1, reg2, . . . regN;
```

其中 msb 和 lsb 定义了范围，并且均为常数值表达式。范围定义是可选的；如果没有定义范围，缺省值为 1 位寄存器。例如：

```
reg [3:0] Sat;     //Sat 为 4  位寄存器。
reg Cnt;           //1 位寄存器。
reg [1:32] Kisp, Pisp, Lisp;
```

寄存器可以取任意长度。reg 型数据的缺省值是未知的，reg 型数据可以为正值或负值。但当一个 reg 型数据是一个表达式中的操作数时，它的值被当作无符号值，即正值。如果一个 4 位的 reg 型数据被写入-1，在表达式中运算时，其值被认为是+15。例如：

```
reg [3:0] Comb;
. . .
Comb = -2;     //Comb  的值为 14（1110），1110 是 2 的补码。
Comb = 5;      //Comb 的值为 15（0101）。
```

2．integer 寄存器类型

整数寄存器包含整数值。整数寄存器可以作为普通寄存器使用，典型应用为高层次行为建模。使用整数型说明形式如下：

```
integer integer1, integer2,. . . intergerN [msb:1sb] ;
```

其中，msb 和 lsb 是定义整数数组界限的常量表达式，数组界限的定义是可选的。一个整数最少容纳 32 位，但是具体实现可提供更多的位。下面是整数说明的实例：

```
integer A, B, C; //三个整数型寄存器。
integer Hist [3:6]; //一组四个寄存器。
```

一个整数型寄存器可存储有符号数，并且算术操作符提供 2 的补码运算结果。整数不能作为位向量访问。例如，对于上面的整数 B 的说明，B[6]和 B[20:10]是非法的。一种截取位值的方法是将整数赋值给一般的 reg 类型变量，然后从中选取相应的位，如下所示：

```
reg [31:0] Breg;
integer Bint;
...
//Bint[6]和 Bint[20:10]是不允许的。
...
Breg = Bint;
/*现在，Breg[6]和 Breg[20:10]是允许的，并且从整数 Bint 获取相应的位值。*/
```

上例说明了如何通过简单的赋值将整数转换为位向量。类型转换自动完成，不必使用特定的函数。从位向量到整数的转换也可以通过赋值完成。例如：

```
integer J;
reg [3:0] Bcq;
J = 6; //J 的值为 32'b0000...00110。
Bcq = J; //Bcq 的值为 4'b0110。
Bcq = 4'b0101;
J = Bcq; //J 的值为 32'b0000...00101。
J = -6; //J 的值为 32'b1111...11010。
Bcq = J; //Bcq 的值为 4'b1010。
```

注意赋值总是从最右端的位向最左边的位进行；任何多余的位被截断。读者注意由于整数是作为 2 的补码位向量表示的，因而可得到这里的类型转换。

3．time 类型

time 类型的寄存器用于存储和处理时间。time 类型的寄存器使用下述方式加以说明。

```
time time_id1, time_id2, . . . ,time_idN [ msb:1sb];
```

其中，msb 和 lsb 是表明范围界限的常量表达式。如果未定义界限，每个标识符存储一个至少 64 位的时间值。时间类型的寄存器只存储无符号数。例如：

```
time Events [0:31]; //时间值数组。
time CurrTime; //CurrTime  存储一个时间值。
```

4．real 类型

实数寄存器（或实数时间寄存器）使用如下方式说明：

```
//实数说明：
real real_reg1, real_reg2, . . ., real_regN;
//实数时间说明：
realtime realtime_reg1, realtime_reg2, . . . ,realtime_regN;
```

realtime 与 real 类型完全相同。例如:

```
real Swing, Top;
realtime CurrTime;
```

real 说明的变量缺省值为 0。不允许对 real 声明值域、位界限或字节界限。当将值 x 和

z 赋予 real 类型寄存器时，这些值作 0 处理。例如：

```
real RamCnt;
...
RamCnt = 'b01x1Z;
```

RamCnt 在赋值后的值为'b01010。

5．realtime 类型

realtime 用于定义实数时间寄存器，其使用方式和 real 型变量一致，这里就不再介绍。

6．reg 的扩展类型——memory 型

Verilog 通过对 reg 型变量建立数组来对存储器建模，可以描述 RAM、ROM 存储器和寄存器数组。数组中的每一个单元通过一个整数索引进行寻址。memory 型通过扩展 reg 型数据的地址范围来达到二维数组的效果，其定义的格式如下：

```
reg [n-1:0]  存储器名  [m-1:0];
```

其中，reg [n-1:0]定义了存储器中每一个存储单元的大小，即该存储器单元是一个 n 位位宽的寄存器；存储器后面的[m-1:0]则定义了存储器的大小，即该存储器中有多少个这样的寄存器。注意存储器属于寄存器数组类型。线网数据类型没有相应的存储器类型。

例如：

```
reg [15:0] ROMA [7:0];
```

这个例子定义了一个存储位宽为 16 位，存储深度为 8 的一个存储器。该存储器的地址范围是 0 到 8。

存储器数组的维数不能大于 2。单个寄存器说明既能够用于说明寄存器类型，也可以用于说明存储器类型，如：

```
parameter ADDR_SIZE = 16 , WORD_SIZE = 8;
reg [1: WORD_SIZE] RamPar [ADDR_SIZE－1 : 0], DataReg;
```

其中 RamPar 是存储器，是 16 个 8 位寄存器数组，而 DataReg 是 8 位寄存器。需要注意的是：对存储器进行地址索引的表达式必须是常数表达式。

尽管 memory 型和 reg 型数据的定义比较接近，但二者还是有很大区别的。例如，一个由 n 个 1 位寄存器构成的存储器是不同于一个 n 位寄存器的。

```
reg [n-1 : 0] rega;        // 一个 n 位的寄存器
reg memb [n-1 : 0];     // 一个由 n 个 1 位寄存器构成的存储器组
```

如果要对 memory 型存储单元进行读写必须要指定地址。例如：

```
memb[0] = 1;          //   将 memeb 中的第 0 个单元赋值为 1。
reg [3:0] Xrom [4:1];
Xrom[1] = 4'h0;
Xrom[2] = 4'ha;
Xrom[3] = 4'h9;
Xrom[4] = 4'hf;
```

在赋值语句中需要注意如下区别：存储器赋值不能在一条赋值语句中完成，但是寄存器可以。如一个 n 位的寄存器可以在一条赋值语句中直接进行赋值，而一个完整的存储器则不行。

```
rega = 0;                    // 合法赋值
memb = 0;                    // 非法赋值
```

在存储器被赋值时，需要定义一个索引。下例说明它们之间的不同。

```
reg [1:5] Dig; //Dig 为 5 位寄存器。
. . .
Dig = 5'b11011;
```

上述赋值都是正确的, 但下述赋值不正确：

```
reg BOg[1:5]; //Bog 为 5 个 1 位寄存器的存储器。
. . .
Bog = 5'b11011;
```

一种存储器赋值的方法是分别对存储器中的每个字赋值。例如：

```
reg [0:3] Xrom [1:4]
. . .
Xrom[1] = 4'hA;
Xrom[2] = 4'h8;
Xrom[3] = 4'hF;
Xrom[4] = 4'h2;
```

另外一种就是通过系统任务读取计算机上的初始化文件来完成，例如：$readmemb 用于加载二进制值，$readmemb 用于加载十六进制值，本书 7.1.2 节将对这两个系统任务进行详细说明。

4.4 运算符和表达式

在 Verilog HDL 语言中运算符所带的操作数是不同的，按其所带操作数的个数可以分为 3 种：

- 单目运算符：带一个操作数，且放在运算符的右边。
- 双目运算符：带两个操作数，且放在运算符的两边。
- 三目运算符：带三个操作数，且被运算符间隔开。

Verilog HDL 语言参考了 C 语言中大多数算符的语法和句义，运算范围很广，其运算符按其功能分为 9 类，在下面各节分别进行介绍。需要注意的是，运算符和表达式即可以用于数据流语句“assign”中，也可用于 always 块语句中；除了“/”和“%”这两个算术操作符受限外，所有的运算符都是可综合的。

4.4.1 赋值运算符

赋值运算分为连续赋值和过程赋值两种。

1．连续赋值

连续赋值语句和过程块一样也是一种行为描述语句，有的文献中将其称为数据流描述形式，但本书将其视为一种行为描述语句。

连续赋值语句只能用来对线网型变量进行赋值，而不能对寄存器变量进行赋值，其基本的语法格式为：

```
线网型变量类型 [线网型变量位宽] 线网型变量名；
```

assign #(延时量) 线网型变量名 = 赋值表达式;

例如：

```
wire a;
assign a = 1'b1;
```

一个线网型变量一旦被连续赋值语句赋值之后，赋值语句右端赋值表达式的值将持续对被赋值变量产生连续驱动。只要右端表达式任一个操作数的值发生变化，就会立即触发对被赋值变量的更新操作。

在实际使用中，连续赋值语句有下列几种应用：

- 对标量线网型赋值

```
wire a, b;
assign a = b;
```

- 对矢量线网型赋值

```
wire [7:0] a, b;
assign a = b;
```

- 对矢量线网型中的某一位赋值

```
wire [7:0] a, b;
assign a[3] = b[1];
```

- 对矢量线网型中的某几位赋值

```
wire [7:0] a, b;
assign a[3:0] = b[3:0];
```

- 对任意拼接的线网型赋值

```
wire a, b;
wire [1:0] c;
assign c ={a ,b};
```

2．过程赋值

过程赋值主要用于两种结构化模块（initial 模块和 always 模块）中的赋值语句。在过程块中只能使用过程赋值语句（不能在过程块中出现连续赋值语句），同时过程赋值语句也只能用在过程赋值模块中。

过程赋值语句的基本格式为：

<被赋值变量><赋值操作符><赋值表达式>

其中，<赋值操作符>是“=”或“<=”，它分别代表了阻塞赋值和非阻塞赋值类型。

在硬件中，过程赋值语句表示用赋值语句右端表达式所推导出的逻辑来驱动该赋值语句左边表达式的变量。过程赋值语句只能出现在 always 语句和 initial 语句中。在 Verilog HDL 语法中有阻塞赋值和非阻塞赋值这两种过程赋值语语句。

（1）阻塞赋值语句

阻塞赋值由符号“=”来完成，“阻塞赋值”由其赋值操作行为而得名：“阻塞”即是说在当前的赋值完成前阻塞其他类型的赋值任务，但是如果右端表达式中含有延时语句，则在延时没结束前不会阻塞其他赋值任务。

（2）非阻塞赋值语句

非阻塞赋值由符号“<=”来完成，“非阻塞赋值”也由其赋值操作行为而得名：在一个时间步（time step）的开始估计右端表达式的值，并在这个时间步（time step）结束时用等式右边的值更新取代左端表达式。在估算右端表达式和更新左端表达式的中间时间段，其他的对左端表达式的非阻塞赋值可以被执行。即“非阻塞赋值”从估计右端开始并不阻碍执

行其他的赋值任务。

过程赋值语句只能对寄存器类型的变量（reg、integer、real 和 time）进行操作，经过赋值后，上面这些变量的取值将保持不变，直到另一条赋值语句对变量重新赋值为止。过程赋值操作的具体目标可以是：

- reg、integer、real 和 time 型变量（矢量和标量）；
- 上述变量的一位或几位；
- 存储器类型，只能对指定地址单元的整个字进行赋值，不能对其中某些位单独赋值。

阻塞赋值和非阻塞赋值是 Verilog HDL 语言学习中的难点，以至于部分资深工程师虽然明白其应用规则，但却很难表达出其本质，因此本书在 8.3 节将对阻塞赋值和非阻塞赋值操作进行详细解释。

例 4-4：给出一个过程赋值的例子。

```
reg c;
always @(a)
begin
    c = 1'b0;
end
```

4.4.2 算术运算符

1．运算符说明

在 Verilog HDL 中，算术运算符又称为二进制运算符，有下列 5 种：

- + 加法运算符或正值运算符，如：s1+s2; +5;
- - 减法运算符或负值运算符，如：s1-s2; -5;
- * 乘法运算符，如 s1*5;
- / 除法运算符，如 s1/5;
- % 模运算符，如 s1%2;

上述操作符中，+、-、*三种操作符都是可综合的，而对于/和%这两种操作只有当除数或者模值为 2 的幂次方的时候（2、4、8、16…）才是可综合的，其余情况都是不可综合的。

在进行整数除法时，结果值要略去小数部分。在取模运算时，结果的符号位和模运算第一个操作数的符号位保持一致。例如：

运算表达式	结果	说明
12.5/3	4	结果为 4，小数部分省去
12%4	0	整除，余数为 0
-15%2	-1	结果取第一个数的符号，所以余数为-1
13/-3	1	结果取第一个数的符号，所以余数为 1

注意：在进行基本算术运算时，如果某一操作数有不确定的值 X，则运算结果也是不确定值 X。

2．算术操作结果的位宽

算术表达式结果的位宽由位宽最大的操作数决定。在赋值语句下，算术操作结果的位宽由操作符左端目标位宽决定。下面通过一个实例来说明上述特征。

例 4-5：以加法操作符为例，说明算术操作符操作结果的位宽保留规则。

```
module opea_demo(
    a_in, b_in, q0_out, q1_out
    );
```

```
input   [3:0] a_in, b_in;
output [3:0] q0_out;
output [4:0] q1_out;

//同位宽操作，会造成数据溢出
assign q0_out = a_in + b_in;

//扩位操作，可保证计算结果正确
assign q1_out = a_in + b_in;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 4-4 所示。可以发现，如果加法两端的数据位宽相同，可能会造成溢出，因此在实际中应当让赋值语句左端和的数据位宽比右端加数的位宽大一位，如程序中的 q1_out。

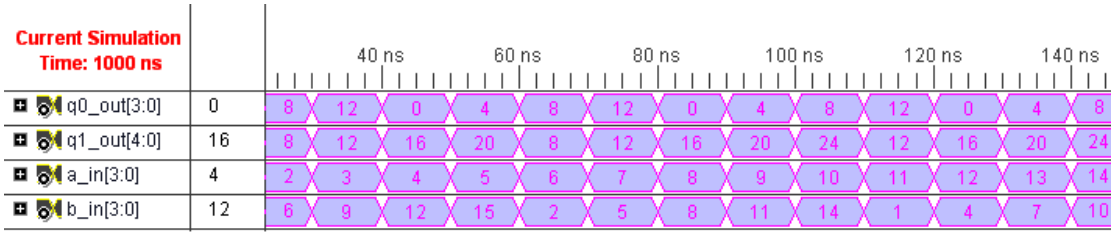


图 4-4　例 4-5 的仿真结果示意图

从图 4-4 可以看出，例 4-5 中第一个加法的结果位宽由 a_in、b_in 和 q0_out 位宽决定，位宽为 4 位。第二个加法操作的位宽同样由 q1_out 的位宽决定（q1_out 的位宽最大），位宽为 6 位。在第一个赋值中，加法操作的溢出部分被丢弃，因此在 40ns 标度处，4 和 12 相加的结果 10000 的最高位被丢弃，q0_out 的数值为 0。而在第二个赋值中，由于任何溢出的位存储在结果位 q1_out[4]中，在 40ns 标度处，4 和 12 相加的结果为 16。

那么在较长的表达式中，中间结果的位宽如何确定？在 Verilog HDL 中定义了如下规则：表达式中的所有中间结果应取最大操作数的位宽（赋值时，此规则也包括左端目标）。下面给出相应的演示实例。

例 4-6：以加法操作符为例说明算术操作符中间结果的位宽保留规则。

```
wire [4:1] Box, Drt;
wire [5:1] Cfg;
wire [6:1] Peg;
wire [8:1] Adt;
. . .
assign Adt = (Box + Cfg) + (Drt + Peg) ;
```

表达式右端的操作数最大位宽为 6，但是将左端包含在内时，最大位宽为 8，所以所有的加操作使用 8 位进行。例如：Box 和 Cfg 相加的结果位宽为 8 位。

3．从有符号数、无符号数衍生的高级讨论话题

在设计中，所有的算数运算符都是按照无符号数进行的。如果要完成有符号数计算，对于加、减操作通过补码处理即可用无符号加法完成。对于乘法操作，无符号数直接采用“*”运算符，有符号数运算可通过定义输出为 signed 来处理。

例 4-7：通过“*”运算符完成有符号数的乘法运算。

```
module signed_mult (out, clk, a, b);
```

```
output          [15:0]     out;
input                      clk;

//通过 signed 关键字定义输入端口的数据类型为有符号数
input      signed    [7:0]      a;
input      signed    [7:0]      b;

//通过 signed 关键字定义寄存器的数据类型为有符号数
reg   signed    [7:0]      a_reg;
reg   signed    [7:0]      b_reg;
reg   signed    [15:0]     out;

wire       signed    [15:0]     mult_out;

//调用*运算符完成有符号数乘法
assign mult_out = a_reg * b_reg;

always@(posedge clk)
begin
      a_reg <= a;
      b_reg <= b;
      out <= mult_out;
end

endmodule
```

上述程序在 ISE 中的综合结果如图 4-5 所示，从其 RTL 结构图可以看到乘法器标注为“signed”，为有符号数乘法器。

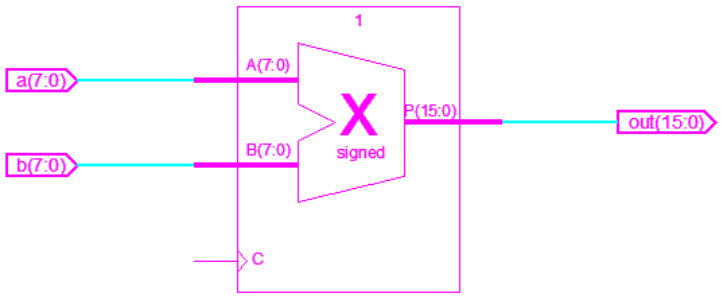


图 4-5　例 4-7 的 RTL 结构图

细心的读者会发现，会什么在图 4-5 所示 RTL 结构图中，乘法器模块有闲置的时钟输入端口 c？事实上，在 ISE 中查阅其逻辑资源占用报告，发现其并未占用逻辑资源，而是调用了硬核乘法器，如图 4-6 所示。由于硬核乘法器具备时钟管脚，因此图 4-5 中的乘法器具有时钟管脚。之所以时钟管脚闲置，则是因为乘法运算符是在“assign”语句中以数据流的方式使用的。

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	0	4656	0%
Number of bonded IOBs	33	232	14%
Number of MULT18X18SIOs	1	20	5%
Number of GCLKs	1	24	4%

图 4-6　例 4-7 的逻辑资源占用表

那么究竟是什么原因让“*”运算符调用了硬核乘法器呢？如果在早期的芯片中没有硬核乘法器，上述程序又会是什么结果呢？其实这正是 ISE 软件综合功能强大的体现，ISE 综合器发现程序中有乘法操作，加上用户所选的 FPGA 芯片型号又包含硬核乘法器，便自动调用速度更快、功耗更低的硬核乘法器来完成代码中的乘法操作。当然读者也可以通过修改 ISE 的综合配置，将乘法的实现变更为由传统的逻辑资源来实现。具体方法如下：在 ISE 过程管理区的“Synthesize-XST”图标上单击右键，在弹出的对话框中选中“HDL Options”选项，然后将右侧的“Multiplier Style”更改为“LUT”（其缺省值为 Auto），如图 4-7 所示。这样，代码中的乘法器则通过传统的逻辑资源来实现。

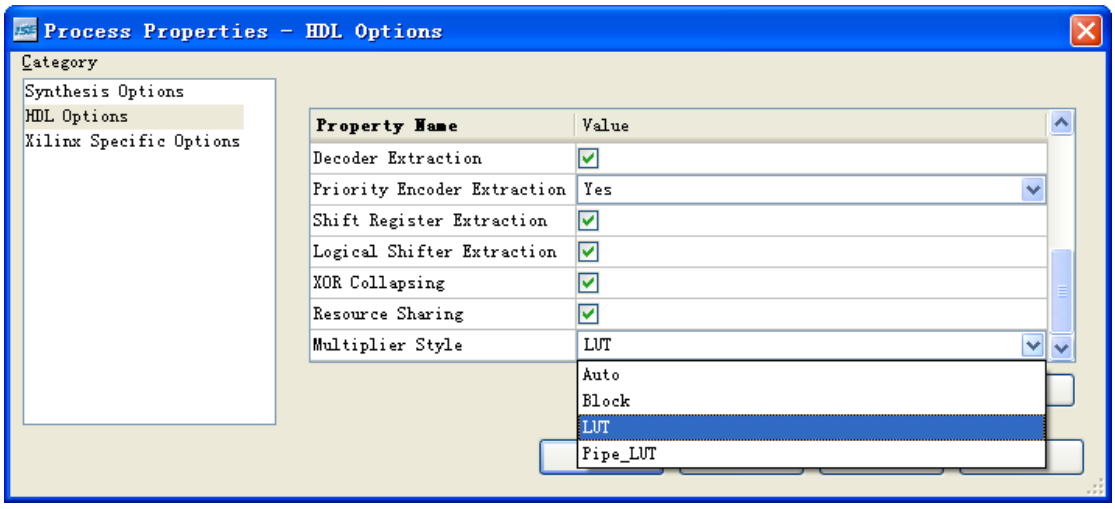


图 4-7　乘法的实现变更

图 4-8 给出了两输入的 8 比特乘法器所需的逻辑资源，大约需要 32 个触发器，83 个 LUT 单元，总共需要 56 个 Slice 单元。

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	56	4656	1%
Number of Slice Flip Flops	32	9312	0%
Number of 4 input LUTs	83	9312	0%
Number of bonded IOBs	33	232	14%
Number of GCLKs	1	24	4%

图 4-8　8 比特乘法器所占逻辑资源示意图

当然，不管采取怎样的方法，最终都能满足设计者的要求。对于例 4-7，无论综合选项怎么设置，其仿真结果都是一致的，如图 4-9 所示。可以正确实现乘法的功能。

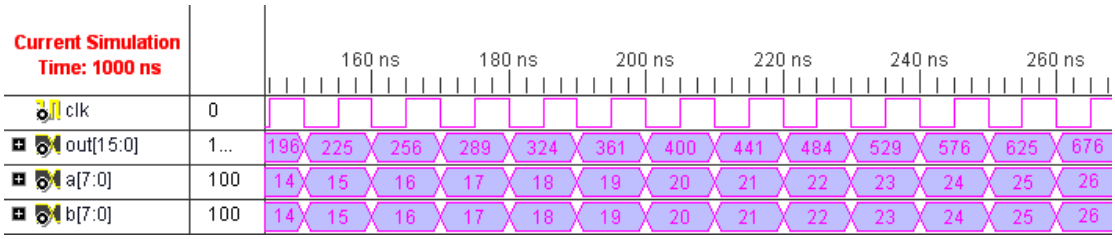


图 4-9　例 4-7 的仿真结果

在实际开发中，不建议使用“*”运算符来直接完成乘法器的开发，因为其不能完全开发乘法器的优势，既不能达到工作的最高性能，又不能根据需求对乘法器进行裁减（串行、并行等结构）。因此，乘法器需要利用 IP 核或者专门的算法模块来实现，本书第 12 章将详细介绍利用 IP 核来配置硬核乘法器的方法。

4.4.3　逻辑运算符

Verilog HDL 中有 3 类逻辑运算符：

- &&　　逻辑与
- ||　　逻辑或
- !　　逻辑非

其中“&&”和“||”是二目运算符，要求有两个操作数；而“!”是单目运算符，只要求一个操作数。“&&”和“||”的优先级高于算术运算符。逻辑运算符的真值表如下表所示：

表 4-5　　逻辑运算符的真值表

a	b	!a	!b	a&&b	a||b
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0
x	x				
x	z				
z	z				

下面给出逻辑运算符的应用示例。

例 4-8：逻辑运算符的应用示例。

```
module logic_demo(
        a_in, b_in, q0_out, q1_out, q2_out
      );

        input     a_in, b_in;
        output    q0_out, q1_out, q2_out;
        reg       q0_out, q1_out, q2_out;

        always @(a_in or b_in) begin
              q0_out = !a_in;
              q1_out = a_in && b_in;
              q2_out = a_in ||b_in;
        end
```

endmodule

上述程序在 ISE 综合后的 RTL 结构图如图 4-10 所示，其和代码设计意图一致，说明了代码的正确性。

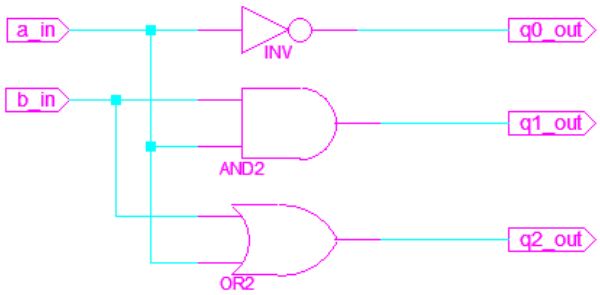


图 4-10　逻辑运算符示例综合结果示意图

4.4.4　关系运算符

关系运算符总共有以下 8 种：

- >　　大于
- >=　　大于等于
- <　　小于
- <=　　小于等于
- ==　　逻辑相等
- !=　　逻辑不相等
- ===　　全等
- !==　　全不等

在进行关系运算符时，如果操作数之间的关系成立，返回值为 1；关系不成立，则返回值为 0；若某一个操作数的值不定，则关系是模糊的，返回的是不定值 X。

实例算子“===”和“!==”可以比较含有 X 和 Z 的操作数，在模块的功能仿真中有着广泛的应用。所有的关系运算符有着相同优先级，但低于算术运算符的优先级。

下面给出关系运算符的应用示例。

例 4-9：关系运算符的应用示例

```
module rela_demo(
    a_in, b_in, q_out
    );

    input   [7:0] a_in, b_in;
    output [7:0] q_out;
    reg      [7:0] q_out;

    always @(a_in or b_in) begin
        q_out[0] = (a_in > b_in) ? 1 : 0;
        q_out[1] = (a_in >= b_in) ? 1 : 0;
        q_out[2] = (a_in < b_in) ? 1 : 0;
        q_out[3] = (a_in <= b_in) ? 1 : 0;
        q_out[4] = (a_in != b_in) ? 1 : 0;
```

```
        q_out[5] = (a_in == b_in) ? 1 : 0;
        q_out[6] = (a_in === b_in) ? 1 : 0;
        q_out[7] = (a_in !== b_in) ? 1 : 0;
    end

endmodule
```

上述程序的仿真结果如图 4-11 所示，可以看出，其运算规则和传统的 C 语言等软件语言是一致的。

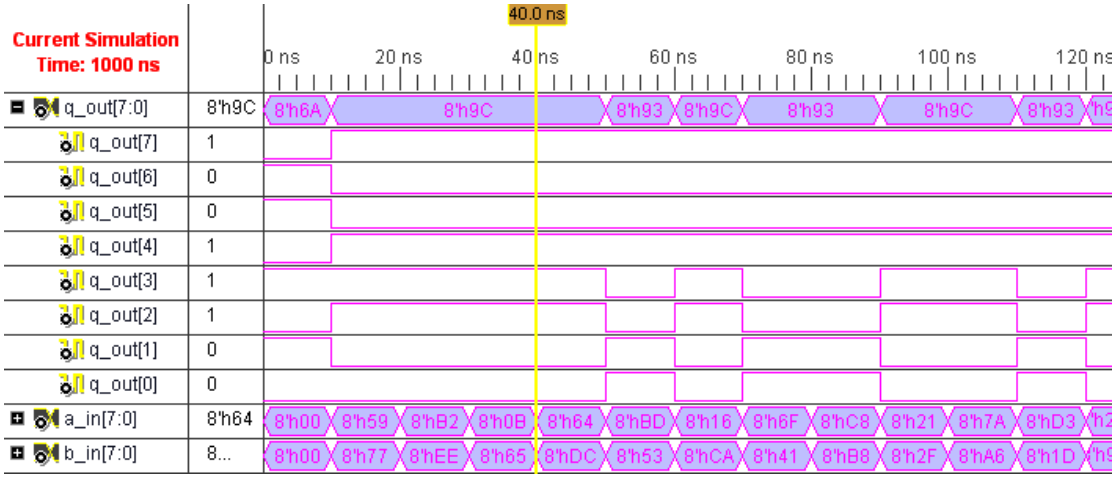


图 4-11　例 4-9 的仿真结果

其中，“===”和“!==”仅用于仿真，在综合时将其按照“==”和“!=”来对待，这是因为在实际硬件系统中不存在“x”和“z”状态。为了便于说明，直接将例 4-9 中的“===”和“!==”语句单独提取出来，放到下面的实例代码中：

```
module alleq(
    a_in, b_in, q_out
);
  input    [7:0] a_in, b_in;
  output [7:0] q_out;
  reg       [7:6] q_out;

  always @(a_in or b_in) begin
      q_out[6] = (a_in === b_in) ? 1 : 0;
      q_out[7] = (a_in !== b_in) ? 1 : 0;
  end

endmodule
```

上述程序在 ISE 中完成综合后，会得到下面的警告提示信息，但仍可以正确综合。

```
=========================================================================
*                            HDL Analysis                               *
=========================================================================
Analyzing top module <alleq>.
WARNING:Xst:1464 - "alleq.v" line 29: Exactly equal expression will be synthesized as an equal expression, simulation mismatch is possible.
```

WARNING:Xst:1465 - "alleq.v" line 30: Exactly not equal expression will be synthesized as a not equal expression, simulation mismatch is possible.

Module <alleq> is correct for synthesis.

其综合后的 RTL 级结构如图 4-12 所示，从中可以看出，全等和全不等被普通的“=”和“!=”所代替。

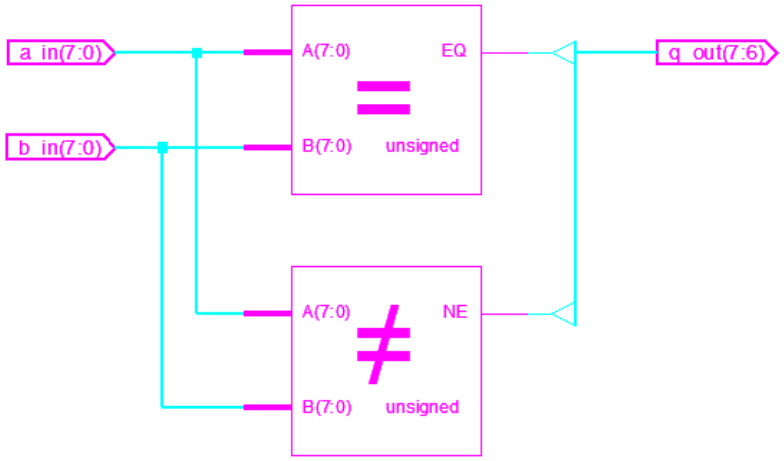


图 4-12　全等和全不等示例代码的 RTL 结构图

4.4.5　条件运算符

条件运算符“？　:”有三个操作数，第一个操作数是 TRUE，算子返回第二个操作数，否则返回第三个操作数，条件算子可以用来实现一个选择器的格式如下：

y = x ? a : b;

如果第一个操作数 y = x 是 True，算子返回第二个操作数 a，否则返回第三个操作数 b。条件运算符“？　:”可以应用于数据流描述形式中，例如：

```
wire y;
assign y = (s1 == 1) ? a : b;
```

此外，“？　:”可通过嵌套的操作实现多路选择。如：

```
wire [1:0] s;
assign s = (a >=2 ) ? 1 : (a < 0) ? 2: 0;
//当 a >=2 时，s=1；当 a <0 时，s=2；在其余情况，s=0。
```

同样，“？　:”可以用于 always 块中，下面给出完整的示例。

例 4-10：条件运算符的示例。

```
module conditional(x, y);
    input [2:0] x;
    output [2:0] y;

    reg [2:0] y;
    parameter xzero = 3'b000;
    parameter xout = 3'b111;

    always @(x)
      y = (x != xout) ? x +1 : xzero;
```

endmodule

从程序代码中可以看出其实现了一个加 1 的限模加法器。程序在 ISE 软件中综合后的 RTL 结构如图 4-13 所示，也是一个加法器，和设计意图一致。

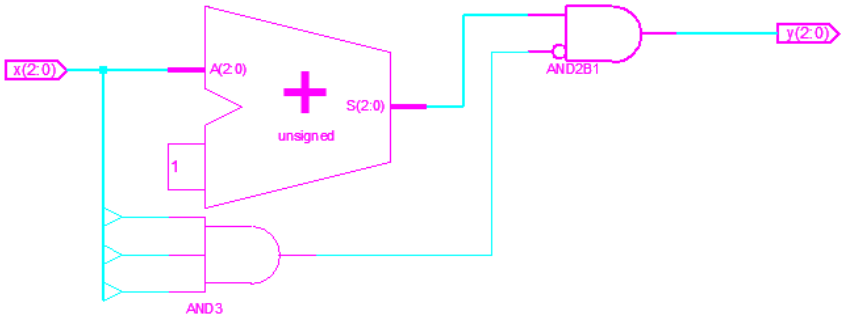


图 4-13　例 4-10 的 RTL 结构

上述程序的 ISE Simulator 中的仿真结果如图 4-14 所示，可以看出正确实现了加 1 加法器，且模值为 8。

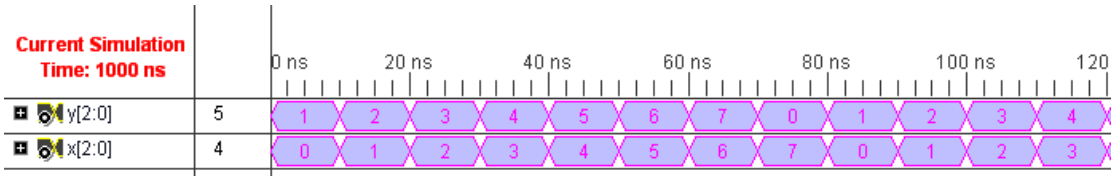


图 4-14　例 4-10 的仿真结果

4.4.6　位运算符

作为一种针对数字电路的硬件描述语言，Verilog HDL 用位运算来描述电路信号中的与、或以及非操作，总共有 5 种位逻辑运算符：

- ~　　非
- &　　与
- |　　或
- ^　　异或
- ^~　　同或

位运算符中除了“~”，都是二目运算符，其操作实例如下所示：

```
s1 = ~s1;
var = ce1 & ce2;
```

位运算对其自变量的每一位进行操作，例如：s1&s2 的含义就是 s1 和 s2 的对应位相与。如果两个操作数的长度不相等的话，将会对较短的数高位补零，然后进行对应位运算，使输出结果的长度与位宽较长的操作数长度保持一致。下面给出说明实例。

例 4-11：位运算符的示例。

```
module bit_demo(
    a, b, c1, c2, c3, c4, c5
    );

  input   [1:0] a, b;
  output [1:0] c1, c2, c3, c4, c5;
```

```
  assign c1 =      ~a;
assign c2 = a & b;
assign c3 = a | b;
assign c4 = a ^ b;
assign c5 = a ^~ b;

endmodule
```

在 ISE 完成综合后，所得的 RTL 级结构图如图 4-15 所示，每个比特运算符都由基本的逻辑单元实现。

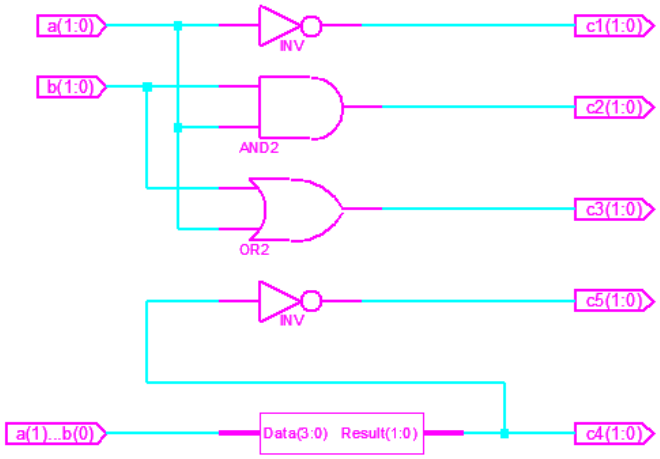


图 4-15　例 4-11 的 RTL 结构

上述程序的 ISE Simulator 中的仿真结果如图 4-16 所示，请读者自行验证仿真结果的正确性。

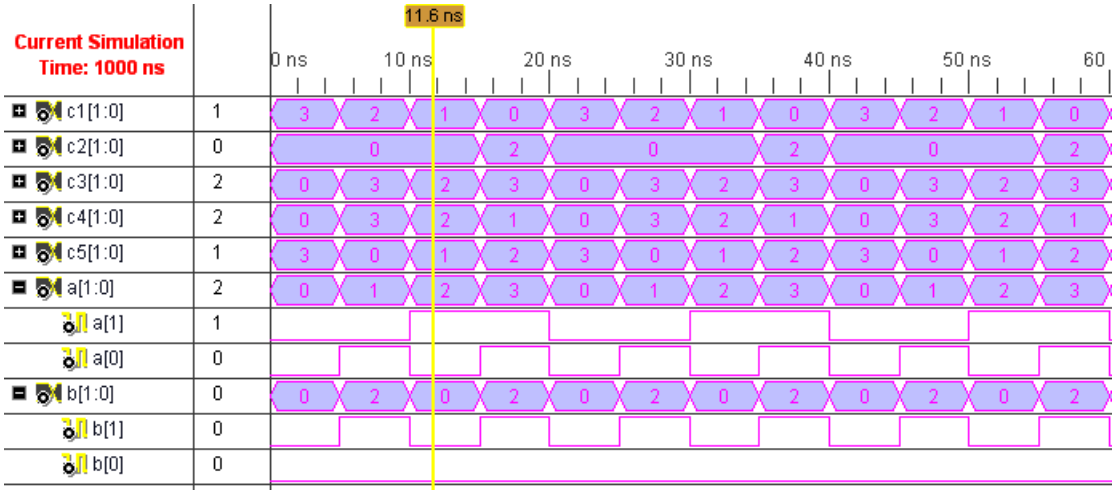


图 4-16　例 4-11 的仿真结果

4.4.7　拼接运算符

拼接运算符可以将两个或更多个信号的某些位并接起来进行运算操作。其使用格式为：

{s1, s2, … , sn}

将某些信号的某些位详细地列出来，中间用逗号隔开，最后用一个大括号表示一个整体信号。

在工程实际中，拼接运算受到了广泛使用，特别是在描述移位寄存器时。

例 4-12：拼接符的 Verilog 实例。

```
reg [15:0] shiftreg;
always @( posedge clk)
    shiftreg [15:0] <= {shiftreg [14:0], data_in};
```

此外，在 Verilog 语言中还有一种重复操作符{{}}，即将一个表达式放入双重花括号中，复制因子放在第一层括号中，为复制一个常量或变量提供一种简便记法。例如，{3{2'b01}} = 6'b010101。

4.4.8 移位运算符

移位运算符只有两种：“<<”（左移）和“>>”（右移），左移一位相当于乘 2，右移一位相当于除 2。其使用格式为：

s1 << N; 或 s1 >>N

其含义是将第一个操作数 s1 向左（右）移位，所移动的位数由第二个操作数 N 来决定，且都用 0 来填补移出的空位。进行移位运算时应注意移位前后变量的位数，下面给出几个例子：

4’b1001<<1 = 5’b10010；　　4’b1001<<2 = 6’b100100；

1<<6 = 32’b1000000；　　4’b1001>> 1= 4’b0100；

4’b1001>>4 = 4’b0000；

在实际运算中，经常通过不同移位数的组合来计算简单的乘法和除法。例如 s1*20，因为 20=16+4，所以可以通过 s1<<4+s1<<2 来实现。下面给出示例代码。

例 4-13：通过移位运算符实现将输入数据放大 19 倍。

```
module amp19(
    clk, din, dout
    );
     input              clk;
     input   [7:0]   din;
     output [11:0] dout;

     reg       [11:0] dint16;
     reg       [11:0] dint2;
     reg       [11:0] dint;

     //将放大倍数 19 分解为 16+2+1
     always @(posedge clk) begin
          dint16 <= din << 4;
          dint2   <= din << 1;
          dint     <= din;
     end

     //将 2 的各次幂值加起来
     assign dout = dint16 + dint2 + dint;    = {din, 4'b0} + {din, 1'b0} + din
```

endmodule

上述程序在 ISE 综合后的 RTL 级结构图如图 4-17 所示，其中 D 触发器用于移位放大数据，两级放大器用于实现 3 输入加法器。

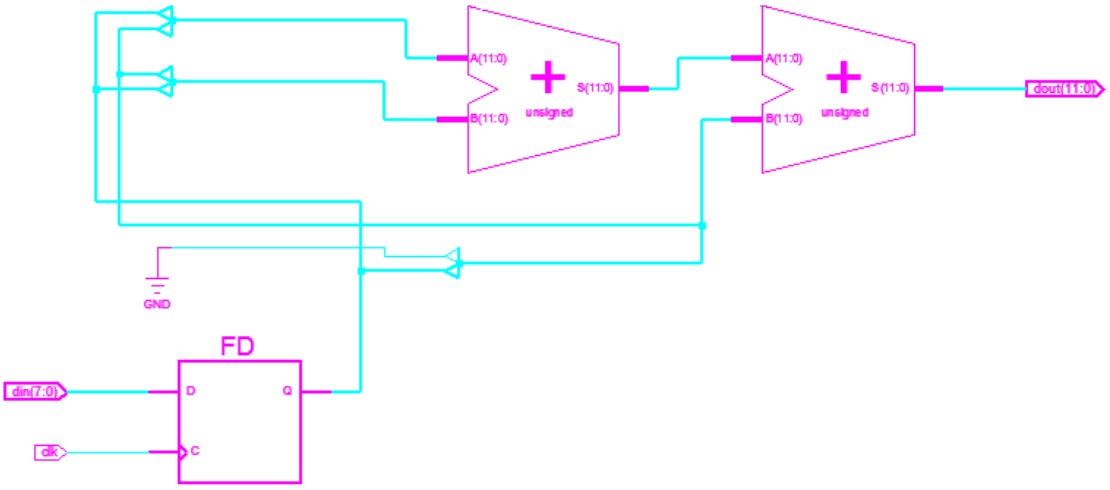


图 4-17　例 4-13 的仿真结果

上述程序在 ISE Simulator 中的仿真结果如图 4-18 所示，可以看出，例 4-13 的代码成功实现了将输入数据放大 19 倍的功能。

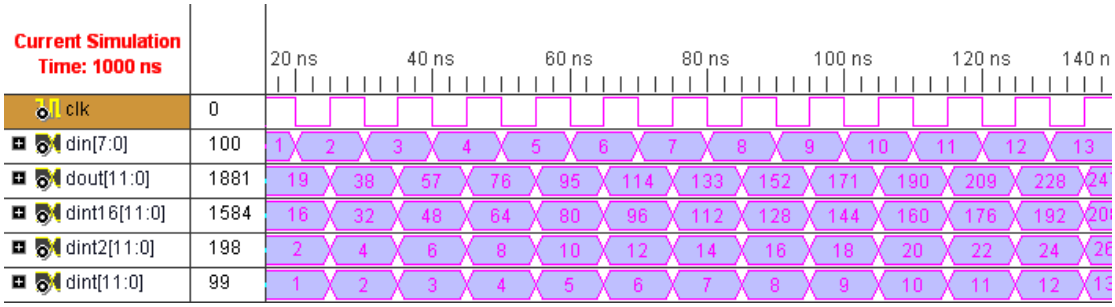


图 4-18　例 4-13 的仿真结果

4.4.9　一元约简运算符

一元约简运算符是单目运算符，其运算规则类似于位运算符中的与、或、非，但其运算过程不同。约简运算符对单个操作数进行运算，最后返回一位数，其运算过程为：首先将操作数的第一位和第二位进行与、或、非运算；然后再将运算结果和第三位进行与、或、非运算；依次类推直至最后一位。

常用的约简运算符与位操作符关键字一样，仅仅由单目运算和双目运算的区别。下面给出示例代码。

例 4-14：给出包含所有一元简约运算符的 Verilog HDL 代码实例。

```
module reduction(a, out1, out2, out3, out4, out5, out6);
    input [3:0] a;
    output out1, out2, out3, out4, out5, out6;

    reg out1, out2, out3, out4, out5, out6;

    always @ (a) begin
      out1 = & a;            //与约简运算
```

```
    out2 = | a;              //或约简运算
    out3 = ~& a;            //与非约简运算
    out4 = ~| a;            //或非约简运算
    out5 = ^ a;             //异或约简运算
    out6 = ~^ a;            //同或约简运算
  end

endmodule
```

这段程序比较简单，这就就不再给出其 RTL 级结构图和仿真结果。读者可以自己完成其综合和仿真实验。

4.5 本章小结

本章主要介绍 Verilog HDL 语言的基本要素，包括用户自定义符号和关键字、关键符号两大类。用户自定义符号主要包括标志符和注释；关键字、关键符号是事先定义好的确认符，作为组织语言的基本要素。所有的关键字都是通过小写字母定义的，因此在代码书写时要注意关键字的大小写，避免出现错误。读者需要注意的是，除了“/”和“%”外，所有的运算符和表达式都是可综合的。

4.6 思考题

1．定义标注符需要遵守什么原则？
2．在 Verilog HDL 语言有哪几种注释的方法？
3．在 Verilog HDL 语言可以使用的逻辑数值包括哪几类？
4．常用的数据类型包括哪几类？哪些是可以综合的，哪些是不可综合的？
5．线网型变量和寄存器变量的区别有哪些？
6．Verilog HDL 中的赋值操作包括哪几类？各自有什么特点？
7．目前，算术运算符中哪些运算符只有在特定场合才能达到可综合的目的？
8．逻辑运算符和位运算符有什么不同，各自在什么场合下使用？
9．拼接符的作用是什么？其物理意义是什么？
10．移位运算符的作用是什么？

第 5 章　面向综合的行为描述语句

如前所述，行为描述级 Verilog HDL 语言中最常用的描述层次，是 Verilog HDL 语言的一个子集。面向综合的代码语句是行为描述语句的两大子集之一，另外一个子集就是面向仿真的代码语句。VerilogHDL 语言是一种硬件开发语言，完成硬件系统设计是其最主要的功能，仿真子集也是为了快速、准确地完成硬件设计而存在。只有可综合的语句才能最终被 EDA 工具转化成硬件设计。本章主要介绍可综合的触发事件控制语句、条件语句、循环语句以及任务与函数。当然，除了“/”和“%”等算术运算受限外，所有的操作符都是可综合的。

5.1 触发事件控制

5.1.1 信号电平事件语句

电平敏感事件是指指定信号的电平发生变化时发生指定的行为。下面是电平触发事件控制的语法和实例：

第一种：@(<电平触发事件>) 行为语句；
第二种：@(<电平触发事件 1> or <电平触发事件 2> or …… or <电平触发事件 n>) 行为语句；

例 5-1：电平沿触发计数器的实例。

```
module counter1(
    clk, reset, cnt);
  input          clk, reset;
  output [4:0] cnt;
  reg [4:0] cnt;
  always @(reset or clk) begin
      if (reset)
          cnt = 0;
      else
          cnt = cnt +1;
  end

endmodule
```

其中，只要 a 信号的电平有变化，信号 cnt 的值就会加 1，这可以用于记录信号 a 的变化的次数。关键字“or”表明事件之间是“或”的关系，在 Verilog HDL2001 规范中，可以用标号“,”来表示或的关系，其相应的语法格式为：

always @(<电平触发事件 1> , <电平触发事件 2> , …,<电平触发事件 n>) 行为语句；

因此也可以将例 5-1 修改为：

```
reg [4:0] cnt;
always @(reset, clk) begin
    if (reset)
```

```
        cnt = 0;
    else
        cnt = cnt +1;
end
```

程序在 ISE 中综合后的 RTL 级结构图如图 5-1 所示。

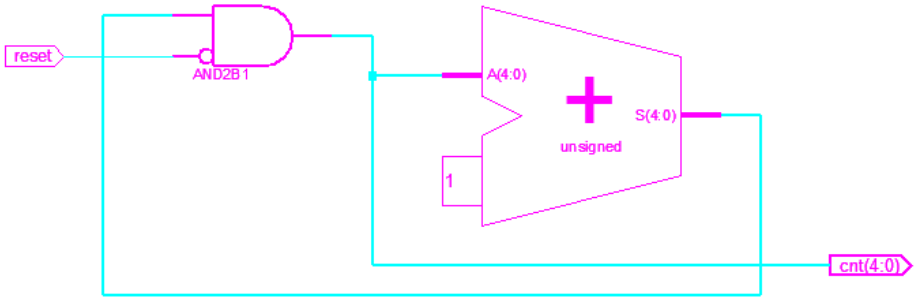


图 5-1　组合逻辑计数器的 RTL 结构示意图

上述程序的仿真结果如图 5-2 所示，在 reset 信号电平为低时，只要 clk 的电平发生变化，cnt 的数值就会累加 1。这也表明了电平触发事件以触发信号的电平变化为触发事件，无论信号电平由高变低，还是由低变高，一律同等对待。

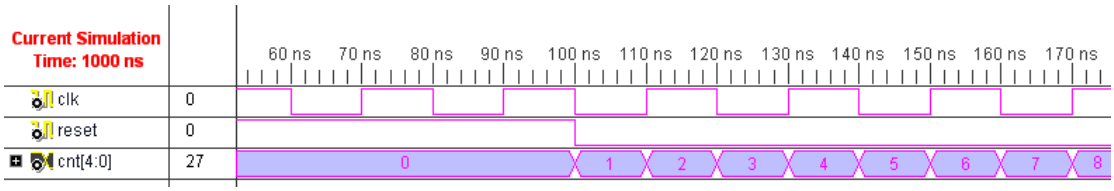


图 5-2　组合逻辑计数器的仿真结果

5.1.2　信号跳变沿事件语句

边沿触发事件是指指定信号的边沿信号跳变时发生指定的行为，分为信号的上升沿（x-->1 or z-->1 or 0-->1）和下降沿（x-->0 or z-->0 or 1-->0）控制。上升沿用 posedge 关键字来描述，下降沿用 negedge 关键字描述。边沿触发事件控制的语法格式为：

第一种：@(<边沿触发事件>) 行为语句；

第二种：@(<边沿触发事件 1> or <边沿触发事件 2> or …… or <边沿触发事件 n>) 行为语句；

例 5-2：基于边沿触发事件的加 1 计数器。

```
module counter2(
   clk, reset, cnt
   );
   input          clk, reset;
   output [4:0] cnt;
   reg     [4:0] cnt;
   always @(negedge clk) begin
       if (reset)
           cnt <= 0;
       else
```

```
            cnt <= cnt +1;
    end
endmodule
```

上面这个例子表明：只要 clk 信号出现下降沿，那么 cnt 信号就会加 1，完成计数的功能。这种边沿计数器在同步分频电路中有着广泛的应用。同样，在 Verilog HDL2001 规范中，信号沿跳变事件语句中的“or”也可以用标号“,”来代替。

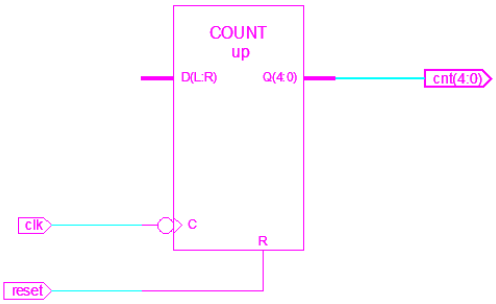


图 5-3　时序逻辑计数器的 RTL 结构示意图

上述程序的仿真结果如图 5-4 所示。通过和例 5-1 的仿真结果比较，可以看出，信号跳变沿触发电路对信号的某一跳变沿敏感，不过在一个时钟周期内，只有一个下降沿和一个跳变沿，因此的计算结果在一个周期内保持不变，而电平触发电路则会引起数据在一个时钟周期内变化一次或多次。

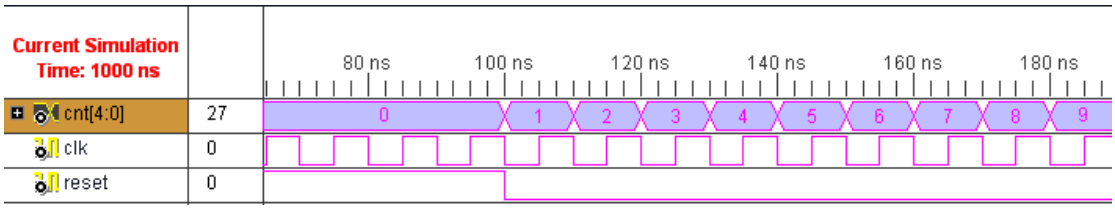


图 5-4　时序逻辑计数器的仿真结果

5.2 条件语句

Verilog HDL 语言含有丰富的条件语句，包括 if 语句和 case 语句，在语法上与 C 语言相似。读者需要注意的是，条件语句只能用于过程块中，包括 initial 结构块和 always 结构块这两类。由于 initial 语句块主要面向仿真应用，因此本节主要介绍在 always 块中的应用。

5.2.1　IF 语句

Verilog HDL 语言中的 if 语句与 C 语言的十分相似，使用起来也很简单，，其使用方法有以下三种：

```
（1）if (条件 1)          语句块 1;
（2）if (条件 1)          语句块 1;
     else  语句块 2;
（3）if (条件 1)          语句块 1;
     else if (条件 2)     语句块 2;
     ……
     else if (条件 n)     语句块 n;
     else  语句块 n+1;
```

在上面三种方式中，“条件”一般为逻辑表达是或者关系表达式，也可以是一位的变量。

如果表达式的值出现 0、x、z，则全部按照“假”处理，若为 1，则按“真”处理。对于第三种形式，如果条件 1 的表达式为真（或非 0 值），那么语句块 1 被执行，否则语句块不被执行，然后依次判断条件 2 至条件 n 是否满足，如果满足就执行相应的语句块，最后跳出 if 语句，整个模块结束。如果所有的条件都不满足，则执行最后一个 else 分支。语句块若为单句，直接书写即可；若为多句，则需要使用“begin end”块将其括起来。建议读者，无论是单句还是多句，都通过“begin end”块括起来，这样便于检查 if 和 else 的匹配，特别是在多重 if 语句嵌套的情况下。

在应用中，else if 分支的语句数目由实际情况决定；else 分支也可以缺省，但在组合逻辑中会产生一些不可预料的逻辑单元，导致设计功能失败，因此应该尽量保持 if 语句分支的完整性。下面给出一个 if 语句的应用实例。

例 5-3：通过 if 语句实现一个多路数据选择器。

```
module sel(
    sel_in, a_in, b_in, c_in, d_in, q_out
    );

     input   [1:0] sel_in;
     input   [4:0] a_in, b_in, c_in, d_in;
     output [4:0] q_out;

     reg      [4:0] q_out;

     always @(a_in or b_in or c_in or d_in or sel_in) begin
        if(sel_in == 2'b00)
              q_out <= a_in;
         else if (sel_in == 2'b01)
              q_out <= b_in;
         else if (sel_in == 2'b10)
              q_out <= c_in;
         else
             q_out <= d_in;
     end

endmodule
```

当 sel_in 的值为 2'b00 时，将 a_in 的值赋给 q_out；当 sel_in 的值为 2'b01 时，将 b_in 的值赋给 q_out；当 sel_in 的值为 2'b10 时，将 c_in 的值赋给 q_out；当 sel_in 的值为 2'b11 时，将 d_in 的值赋给 q_out。上述程序在 ISE Simulator 中的仿真结果如图 5-5 所示，验证了设计的正确性。

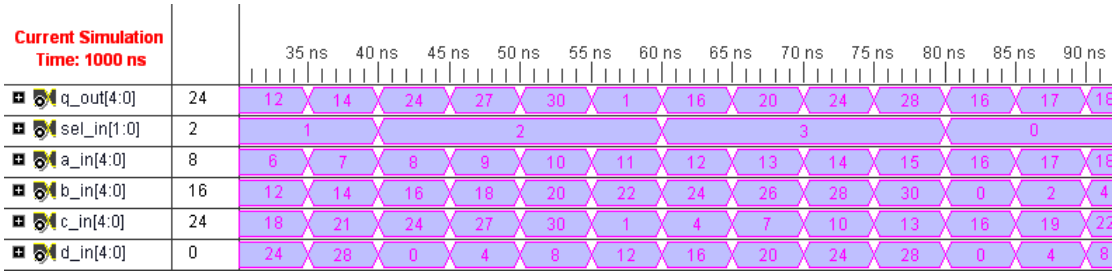


图 5-5　例 5-3 的仿真结果

5.2.2　CASE 语句

case 语句是一个多路条件分支形式，常用于多路译码、状态机以及微处理器的指令译码等场合，有 case、casez 和 casex 这三种形式。

1．case 语句

case 语句的语法格式为：

```
case   (<条件表达式>)
    <分支 1>：<语句块 1>;
    <分支 2>：<语句块 1>;
    ...
    default：<语句块 n>
endcase
```

其中的<分支 n>通常都是一些常量表达式。case 语句首先对条件表达式求值，然后同时并行对各分支项求值并进行比较，这是 if 语句最大的不同。比较完成后，与条件表达式值相匹配的分支中的语句被执行。可以在 1 个分支中定义多个分支项，但这些值需要互斥，否则会出现逻辑矛盾。缺省分支 default 将覆盖所有没有被分支表达式覆盖的其他分支。此外，当 case 语句跳转到某一分支后，控制指针将转移到 endcase 语句之后，其余分支将不再遍历比较，因此不需要类似 C 语言中的 break 语句。

如果几个分支都对应着同一操作，则可以通过逗号将这个不同分支的取值隔开，再将这些情况下需要执行的语句放在这几个分支值之后，其格式为：

```
<分支 1>, <分支 2>, ..., <分支 n>;
        <语句块>;
```

下面给出一个 case 语句的例子，随着 cnt 的取值，q 和不同的数相加，其功能等效于 q=cnt+q+1。

```
reg [2:0] cnt;
case (cnt)
    3'b000: q = q + 1;
    3'b001: q = q + 2;
    3'b010: q = q + 3;
    3'b011: q = q + 4;
    3'b100: q = q + 5;
    3'b101: q = q + 6;
    3'b110: q = q + 7;
    3'b111: q = q + 8;
    default: q <= q + 1;
endcase
```

需要指出的是，case 语句的 default 分支虽然可以缺省，但是一般不要缺省，否则在组合逻辑中，会和 if 语句中缺少 else 分支一样，生成锁存器。

case 语句在执行时，条件表达式和分支之间进行的比较是一种按位进行的全等比较，也就是说，只有在分支项表达式和条件表达式的每一位都彼此相等的情况下，才会认为二者是“相等”的。在进行对应比特的比较时，x、z 这两种逻辑状态也作为合法状态参与比较。各逻辑值在比较时的真值表如表 5-1 所列，其中 Ture 表示比较结果相等，False 表示比较结

果不等。

表 5-1　　case 语句的比较规则

case	0	1	x	z
0	Ture	False	False	False
1	False	Ture	False	False
x	False	False	Ture	False
z	False	False	False	Ture

由于 case 语句有按位进行全等比较的特点，因此 case 语句的条件表示式和分支值必须具备同样的位宽，只有这样才能进行对应位的比较。当各分支取值以常数形式给出时，必须显式地表明其位宽，否则 Verilog HDL 编译器会默认其具有与 PC 机字长相等的位宽。例 5-4 给出了一个实现操作码译码的实例。

例 5-4：使用 case 语句实现操作码译码。

```
module decode_opmode(
    a_in, b_in, opmode, q_out
    );

      input    [7:0] a_in;
      input    [7:0] b_in;
      input    [1:0] opmode;
      output   [7:0] q_out;

      reg      [7:0] q_out;

      always @(a_in or b_in or opmode) begin
        case(opmode)
          2'b00: q_out = a_in + b_in;
          2'b01: q_out = a_in + b_in;
          2'b10: q_out = (~a_in) + 1;
          2'b11: q_out = (~b_in) + 1;
        endcase
      end

endmodule
```

上述程序中的输入信号 opmode 是宽度为两位的操作码，用于指定输入 a_in 和 b_in 执行的运算类型。当操作码为 2'b00，则取值为 a_in 和 b_in 的和；当操作码为 2'b01，则取值为 a_in 和 b_in 的差；当操作码为 2'b10，则取值为 a_in 的补码；当操作码为 2'b11，则取值为 b_in 的补码。上述程序在 ISE Simulator 中的仿真结果如图 5-6 所示，验证了设计的正确性。

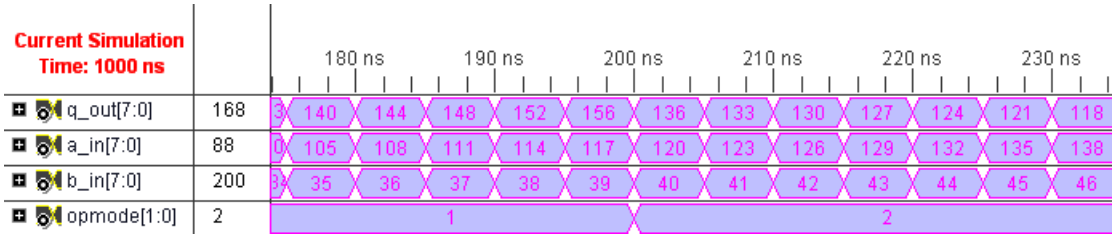


图 5-6　例 5-4 的仿真结果

2．casez 和 casex 语句

casez 和 casex 语句是 case 语句的变体。在 casez 语句中，如果分支取值的某些位为高阻 z，则这些位的比较就不予以考虑，只关注其他位的比较结果；casex 语句则把这种处理方式扩展到对 x 的处理，即如果比较双方有一方的某些位为 x 或 z，那么这些位的比较就不予以考虑。表 5-2、5-3 分别给出 casez 和 casex 比较时的真值表。

表 5-2　　casez 语句的比较规则

casez	0	1	x	z
0	Ture	False	False	Ture
1	False	Ture	False	Ture
x	False	False	Ture	Ture
z	Ture	Ture	Ture	Ture

表 5-3　　casex 语句的比较规则

casex	0	1	x	z
0	Ture	False	Ture	Ture
1	False	Ture	Ture	Ture
x	Ture	Ture	Ture	Ture
z	Ture	Ture	Ture	Ture

在 casez 和 casex 语句中，分支取值的 z 也可以用符号“?”代替，例如：

```
reg [1:0] a, b;
casez(b)
   2'b1? : a = 2'b00;
   2'b?1 : a = 2'b11;
endcase
```

其与下面的代码是等效的，只要 b 的高比特为 1，则 a 的值为 2'b00；b 的低比特为 1，则 a 的值为 2'b11。

```
reg [1:0] a, b;
casez(b)
   2'b1z : a = 2'b00;
   2'bz1 : a = 2'b11;
endcase
```

从上述内容可以看出，casez 和 casex 的唯一不同就在于对 x 逻辑的处理，其语法规则是完全一致的。下面以 casex 为例，给出例 5-5 所实现的操作码译码器。

例 5-5：使用 case 语句实现操作码译码。

```
module decode_opmodex(
      a_in, b_in, opmode, q_out
      );

       input     [7:0] a_in;
       input     [7:0] b_in;
       input     [3:0] opmode;
       output    [7:0] q_out;
```

```
reg        [7:0] q_out;

always @(a_in or b_in or opmode) begin
    casex(opmode)
        4'b0001: q_out = a_in + b_in;
        4'b001x: q_out = a_in - b_in;
        4'b01xx: q_out = (~a_in) + 1;
        4'b1zx?: q_out = (~b_in) + 1;
      default: q_out = a_in + b_in;
      endcase
end

endmodule
```

上述代码在比较 opmode 和 casex 分支数值时，将分别忽略其中取值为 z、x 以及?的位。只要操作码 opmode 的取值为 4'b0001，q_out 的数值为两个数相加的和；高三位取值为 1，q_out 的值为 a_in - b_in；高两位为 2'b01，q_out 的值为 a_in 的补码；最高位为 1 时，q_out 的值为 b_in 的补码。上述程序在 ISE Simulator 中的仿真结果如图 5-7 所示，验证了设计的正确性。

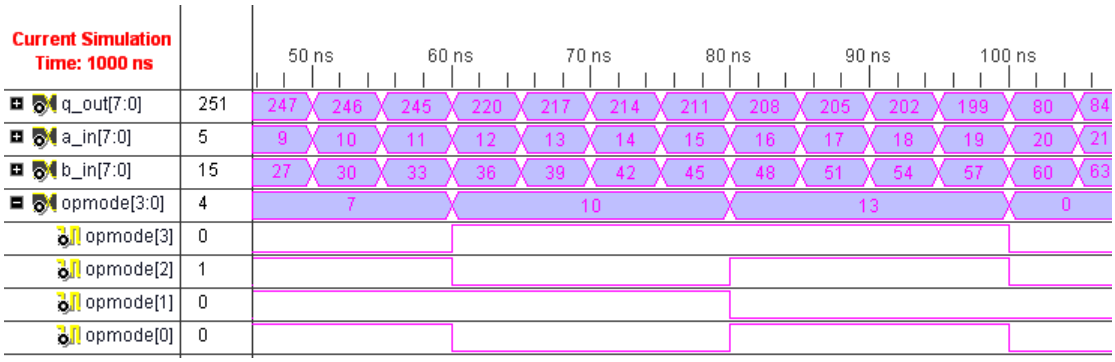


图 5-7　例 5-5 的仿真结果

5.2.3　条件语句的深入理解

1．if 和 case 语句的区别

if 语句指定了一个有优先级的编码逻辑，而 case 语句生成的逻辑是并行的，不具有优先级。if 语句可以包含一系列不同的表达式，而 case 语句比较的是一个公共的控制表达式。通常 if-else 结构速度较慢，但占用的面积小，如果对速度没有特殊要求而对面积有较高要求，则可用 if-else 语句完成编解码。case 结构速度较快，但占用面积较大，所以用 case 语句实现对速度要求较高的编解码电路。嵌套的 if 语句如果使用不当，就会导致设计的更大延时，为了避免较大的路径延时，最好不要使用特别长的嵌套 if 结构。如想利用 if 语句来实现那些对延时要求苛刻的路径时，应将最高优先级给最迟到达的关键信号。有时为了兼顾面积和速度，可以将 if 和 case 语句合用。

2．if 语句和 case 语句的实例剖析

下面分别给出两个使用 if 和 case 语句的实例，希望读者从中体会到二者的不同。

例 5-6：使用 if 语句实现一个 4 选 1 的数据通路选择器。

```
module sdata_if(clk, reset, x, s, y);
   input clk;
```

```
    input reset;
    input [3:0] x;
    input [1:0] s;
    output y;

    reg y;
    always @(posedge clk) begin
        if(!reset) begin
            y <= 0 ;
       end
       else begin
            if(s == 2'b00)
                 y <= x[0];
             else if (s == 2'b01)
                 y <= x[1];
             else if (s == 2'b10)
                 y <= x[2];
             else
                 y <= x[3];
       end
    end

endmodule
```

上述程序经过综合后，其中数据选择部分的 RTL 级结构如图 5-8 所示。从中可以看出状态变量 s[1:0]通过 y13、y14、y15 以及 y16 是串行输入到复用器中的，且具有严格的逻辑顺序，其逻辑级数为 4。

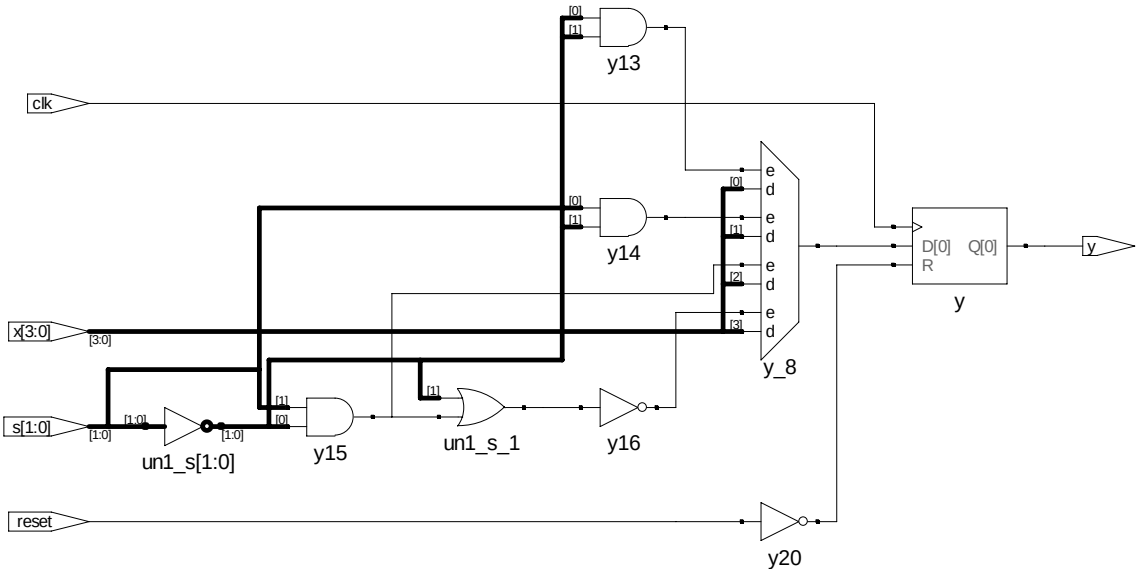


图 5-8　用 if 语句实现的 4 选 1 选择器 RTL 级结构图

例 5-7：使用 case 语句实现一个 4 选 1 的 8 位数据选择器。

```
module sdata_case(clk, reset, x, s, y);
```

```
input clk;
input reset;
input [3:0] x;
input [1:0] s;
output y;

reg y;
always @(posedge clk) begin
    if(!reset) begin
       y <= 0 ;
   end
   else begin
       case(s)
           2'b00: y <= x[0];
           2'b01: y <= x[1];
           2'b10: y <= x[2];
           2'b11: y <= x[3];
        endcase
   end
end

endmodule
```

上述程序经过综合后，其中数据选择部分的 RTL 级结构如图 5-9 所示。从中可以看出状态变量 s[1:0]通过 y13、y14、y15 以及 y16 是并行输入到复用器中的，因此逻辑级数只有 1 级。

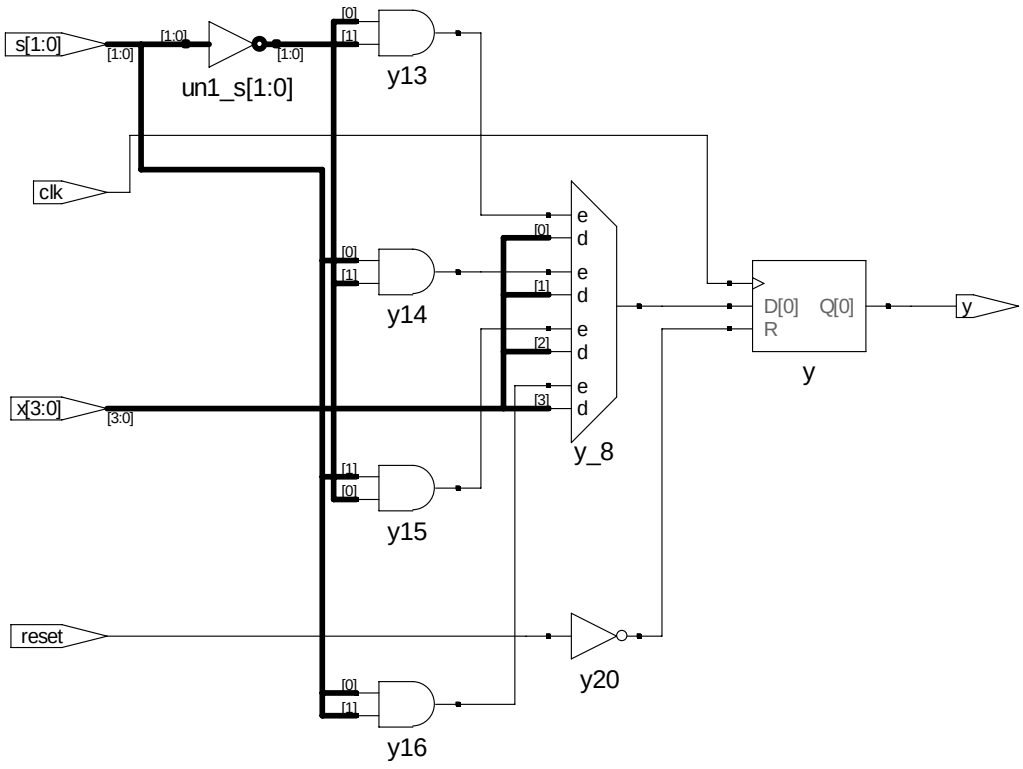


图 5-9　用 case 语句实现的 4 选 1 选择器 RTL 级结构图

5.3 循环语句

Verilog HDL 中提供了 4 种循环语句，可用于控制语句的执行次数，分别为：

- for 循环：执行给定的循环次数；
- while 循环：执行语句直到某个条件不满足；
- repeat 循环：连续执行语句 N 次；
- forever 循环：连续执行某条语句。

其中，for、while 以及 repeat 是“可综合”的，但循环的次数需要在编译之前就确定，动态改变循环次数的语句则是不可综合的；forever 语句是不可综合的，常用于产生各类仿真激励。因此，本节主要介绍 for、while 和 repeat 语句，forever 语句将在 6.4 节介绍。

5.3.1 REPEAT 语句

repeat 循环语句执行指定循环数，如果循环计数表达式的值不确定，即为 x 或 z 时，那么循环次数按 0 处理。repeat 循环语句的语法为

```
repeat(循环次数表达式) begin
    语句块;
end
```

其中，“循环次数表达式”用于指定循环次数，可以是一个整数、变量或者数值表达式。如果是变量或者数值表达式，其数值只在第一次循环时得到计算，从而得以事先确定循环次数；“语句块”为重复执行的循环体。

在可综合设计中，“循环次数表达式”必须在程序编译过程中保持不变。下面给出一个利用 repeat 语句实现两个 8 比特数据的乘法。

例 5-8：利用 repeat 语句实现两个 8 比特数据的乘法。

```
module mult_8b_repeat(
    a, b, q
    );

      parameter bsize = 8;
      input    [bsize-1 : 0] a, b;
      output [2*bsize-1 : 0] q;

      reg [2*bsize-1 : 0] q, a_t;
      reg [bsize-1 : 0] b_t;

      always @(a or b) begin
          q = 0;
          a_t = {{bsize{0}},a};
          b_t = b;

          repeat(bsize) begin
                if (b_t[0]) begin
                    q = q + a_t;
```

```
            end
            else begin
                q = q;
            end
            a_t = a_t << 1;
            b_t = b_t >> 1;
        end
    end

endmodule
```

在程序中，repeat 语句中指定循环次数的是参数“bsize”，其数值为 8，因此循环体将被重复执行 8 次；循环体部分由一个 begin…end 语句块组成，每执行一次就进行一次移位相加操作，在重复执行 8 次后就完成了两个 8 位输入数据的相乘运算。图 5-10 给出了乘法器的仿真结果，从中可以看出，利用 repeat 语句正确地实现了乘法功能。

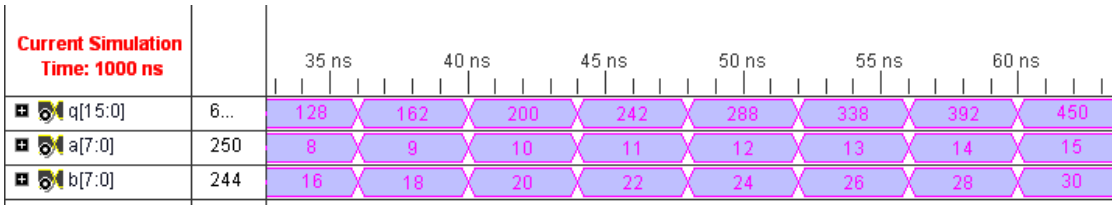


图 5-10　基于 repeat 语句乘法器仿真结果示意图

5.3.2　WHILE 语句

while 循环语句实现的是一种“条件循环”，只有在指定的循环条件为真时才会重复执行循环体，如果表达式条件在开始不为真（包括假、x 以及 z），那么过程语句将永远不会被执行。while 循环的语法为：

```
while (循环执行条件表达式) begin
    语句块
end
```

在上述格式中，“循环执行条件表达式”代表了循环体得到继续重复执行时必须满足的条件，通常是一个逻辑表达式。在每一次执行循环体之前，都需要对这个表达式是否成立进行判断。“语句块”代表了被重复执行的部分，可以为单句或多句。

While 语句在执行时，首先判断循环执行条件表达式是否为真，如果真，执行后面的语句块，然后再重新判断循环执行条件表达式是否为真，为真的话，再执行一遍后面的语句块，如此不断，直到条件表达式不为真。因此，在执行语句中，必须有改变循环执行条件表达式的值的语句，否则循环就变成死循环。

下面通过 while 语句实现例 5-8 的功能，完成两个 8 比特无符号数相乘的功能。

例 5-9：使用 while 语句实现两输入 8 位无符号数据的乘法。

```
module mult_8b_while(
        a, b, q
    );

    parameter bsize = 8;
    input   [bsize-1 : 0] a, b;
```

```
output [2*bsize-1 : 0] q;

reg [2*bsize-1 : 0] q, a_t;
reg [bsize-1 : 0] b_t;
reg [bsize-1 : 0] cnt;

always @(a or b) begin
    q = 0;
     a_t = {{bsize{0}},a};
     b_t = b;
     cnt = bsize;

     while(cnt > 0) begin
         if (b_t[0]) begin
              q = q + a_t;
          end
          else begin
              q = q;
          end
          cnt = cnt - 1;
          a_t = a_t << 1;
          b_t = b_t >> 1;
     end
end

endmodule
```

上述程序中，while 语句开始执行时，cnt 的初始值为 8，条件表达式成立，循环体语句开始执行，并将 cnt 的值减 1；再次判断执行条件，执行循环语句，直到经过 8 次循环后，cnt 的值为 0，这时条件表达式不再成立，循环结束。图 5-11 给出了相应的仿真结果，从中可以看出，while 循环正确地实现了两个无符号数相乘的功能。

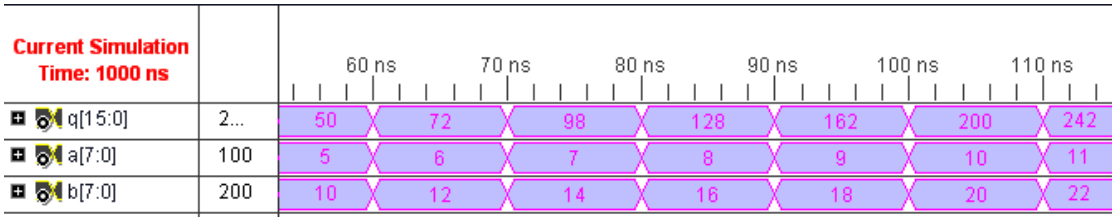


图 5-11　基于 while 语句的乘法器仿真结果

5.3.3　FOR 语句

和 while 循环语句一样，for 循环语句实现的循环也是一种“条件循环”，按照指定的次数重复执行过程赋值语句，其语法格式为：

for(表达式 1; 表达式 2; 表达式 3) 语句块;

for 循环语句最简单的应用形式是很容易理解的，其形式为：

```
for(循环变量赋初值; 循环执行条件; 循环变量增值)
  循环体语句的语句块;
```

其中，“循环变量赋初值”和“循环变量增值”语句是两条过程赋值语句；“循环执行条件”代表着循环继续执行的条件，通常是一个逻辑表达式，在每一次执行循环体之前都要对这个条件表达式是否成立进行判断；“循环体语句的语句块”是要被重复执行的循环体部分，如果超过多条语句，需要使用“begin…end”语句块将循环体语句括起来。

for 循环语句的执行过程可以分为以下几步：

（1）执行“循环变量赋初值”语句；

（2）执行“循环执行条件”语句，判断循环变量的值是否满足循环执行条件。若结果为真，执行循环体语句，然后继续执行下面的第（3）步；否则，结束循环语句。

（3）执行“循环变量增值”语句，并跳转到第（2）步。

从上面说明可以看出，“循环变量赋初值”语句只在第一次循环开始之前被执行一次，“循环执行条件”在每次循环开始之前都会被执行，而“循环变量增值”语句在每次循环结束之后被执行。读者可以发现，如果“循环变量增值”语句不改变循环变量的值，则 for 语句会进入无限次循环的死循环状态，这种情况在程序设计中是要避免的。

事实上，for 语句等价于由 while 循环语句构建的如下循环结构：

```
begin
  循环变量赋初值;
  while(循环执行条件) begin
      循环体语句的语句块;
      循环变量增值;
  end
end
```

这里需要强调的是，虽然从表面上看来，while 语句需要 3 条语句才能完成一个循环控制，for 循环制需要一条语句就可以实现，但二者对应的逻辑本质是一样的。在代码书写时，由于 for 语句的表述比 while 语句更清晰、简洁，便于阅读，因此推荐使用 for 语句。

目前，大多数 EDA 工具都支持 for 语句的综合，包括 Xilinx 的 ISE 工具，但要求“循环结束条件”是个常量。下面通过计算数据零比特个数的程序来说明 for 语句的使用。

例 5-10：使用 for 语句统计输入数据中所包含零比特的个数。

```
module countzeros (a, Count);
    input [7:0] a;
    output [2:0] Count;
    reg [2:0] Count;
    reg [2:0] Count_Aux;
    integer i;
    always @(a) begin
        Count_Aux = 3'b0;
        for (i = 0; i < 8; i = i+1) begin
            if (!a[i])
                Count_Aux = Count_Aux+1;
        end
        Count = Count_Aux;
    end
endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 5-12 所示，可以看出该段程序正确实现了统计输入数据中零比特的个数，达到了设计目的。

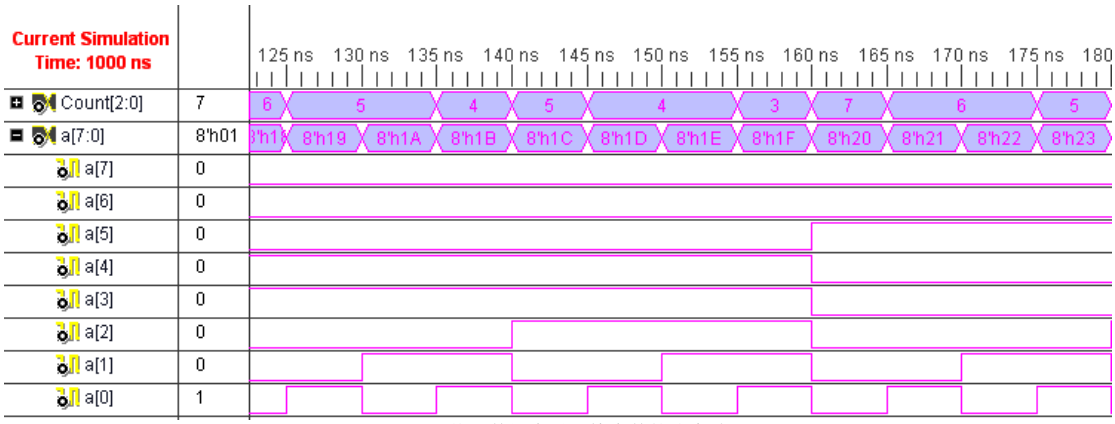


图 5-12　统计数据中零比特个数的仿真结果

在应用时，repeat、while 以及 for 语句三者之间是可以相互转化的，例如对于一个简单的 5 次循环，分别用 repeat、while 以及 for 书写如下：

```
for( i = 0; i <= 4; i=i+1)
begin
   …
end
```

```
repeat(5)
begin
      …
end
```

```
i = 0;
while(i<5)
begin
   …
   i = i + 1;
end
```

为了说明几种循环语句之间可以互换，下面给出用 for 循环实现两个 8 位无符号数相乘的实例。

例 5-11：使用 for 语句实现两输入 8 位无符号数据的乘法。

```
module mult_8b_for(
      a, b, q
      );

        parameter bsize = 8;
        input    [bsize-1 : 0] a, b;
        output [2*bsize-1 : 0] q;

        reg [2*bsize-1 : 0] q, a_t;
        reg [bsize-1 : 0] b_t;
        reg [bsize-1 : 0] cnt;

        always @(a or b) begin
            q = 0;
             a_t = {{bsize{0}},a};
             b_t = b;
             cnt = bsize;

             for(cnt = bsize; cnt>0; cnt = cnt-1)begin
```

```
            if (b_t[0]) begin
                q = q + a_t;
            end
            else begin
                q = q;
            end
            a_t = a_t << 1;
            b_t = b_t >> 1;
        end
    end
endmodule
```

图 5-13 给出了程序在 ISE Simulator 中的仿仿真结果，从中可以看出，for 循环语句正确地实现了两个无符号数相乘的功能。

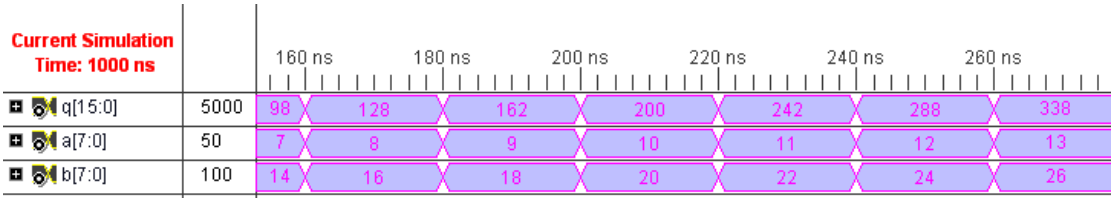


图 5-13　基于 for 语句的乘法器的功能仿真结果

读者需要特别注意的是：在 Verilog 中不支持 C/C++语言中的“++”和“—”运算，因此必须通过完整的语句“i = i + 1”和“i = i -1”来实现类似功能。

5.3.4　循环语句的深入理解

1．循环语句应用指南

Verilog HDL 是一门硬件描述语言，如果期望代码在硬件中实现，则需要经过 EDA 工具将其最终翻译成基本的门逻辑。而在硬件电路中并没有循环电路的原型，因此在使用循环语句时要十分小心，必须时刻在意其可综合性。

这里向读者再次介绍一点硬件设计思想，在硬件系统中，任何 RTL 级的描述都是需要占用资源的。因此必须确保循环是一个有限循环，否则设计将是不可综合的，因为任何硬件实现平台（FPGA、ASIC）的资源都是有限的，这也是 forever 语句是不可综合的原因。当然，循环语句的优势也是明显的，代码简捷明了，便于维护和管理，具有高级语言的普遍特征。

根据作者的设计经验，硬件里的 for 语句不会像软件程序那样频繁的使用，一方面是因为 for 语句需要占用一定的硬件资源，另一方面是因为在设计中 for 循环可通过计数器来代替。所以，在 Verilog HDL 程序开发中，使用循环语句一定要谨慎，毕竟描述层次越抽象，将其转化成硬件实现的难度就越大，性能就越差，并且占用资源越多。

2．实例剖析

对于初学者来讲，上述应用指南可能比较难懂，下面通过实例阐述循环语句的各类特点。本质上，repeat、while 以及 for 语句都是等效的，因此本节以形式简洁易懂的 for 语句为例进行说明。

首先对例 5-11 中的 for 语句添加 50MHz 的时序约束（a、b 输入信号的更新频率），并完成布局布线后时序仿真，其结果如图 5-14 所示。可以看出，输出结果存在很大的抖动，这对系统级应用是致命的。

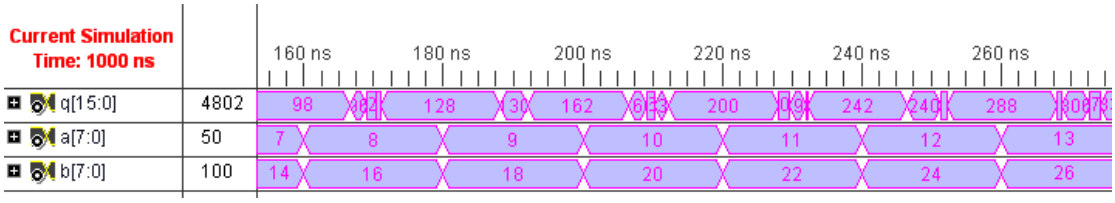


图 5-14　基于 for 语句的乘法器的布局布线后仿真结果

当然，在组合逻辑中经常会出现图 5-15 所示的毛刺和抖动，那么从时序仿真结果无法直接确定 for 语句的特点。为此，将例 5-11 的代码改写成时序逻辑，即同步于工作时钟，相应的代码如例 5-12 所列。

例 5-12：将例 5-11 的代码改写成时序逻辑，将其和传统的计数器实现代码比较，并在 ISE 10.x 工具上得到时序、资源特性的结论，硬件平台选择 XC3S500E-FG320-4。

（1）例 5-11 对应的时序逻辑如下：

```
module mult_8b_for(
    clk, a, b, q
    );

    parameter bsize = 8;
    input                    clk;
    input   [bsize-1 : 0]    a, b;
    output [2*bsize-1 : 0] q;

    reg [2*bsize-1 : 0] q, a_t;
    reg [bsize-1 : 0] b_t;
    reg [bsize-1 : 0] cnt;

    always @(posedge clk) begin
        q = 0;                          //一定要使用阻塞赋值
        a_t = {{bsize{0}},a};
        b_t = b;
        cnt = bsize;

        for(cnt = bsize; cnt>0; cnt = cnt-1)begin
            if (b_t[0]) begin
                q = q + a_t;          //一定要使用阻塞赋值
            end
            a_t = a_t << 1;          //一定要使用阻塞赋值
            b_t = b_t >> 1;         //一定要使用阻塞赋值
        end
    end
endmodule
```

其中，在循环语句中出现的变量都采用了阻塞赋值，这是因为在 always 语句中使用非阻塞赋值“<=”时，只有在 always 结束后才会把右端的值赋给左边的寄存器。如果采用非阻塞赋值，则会造成循环语句只执行一次，这一点请读者自行验证。

上述程序的功能仿真结果如图 5-15 所示，从中可以看出，在时钟的上升沿，加载输入，

同时就能得到相应的输出，没有任何延迟。那么，在时序电路中，电路运行同步于时钟的特征如何体现呢？其实这是由于循环语句本质上还是由组合逻辑实现的，虽然模块整体基于时序逻辑，但其中循环部分却是组合逻辑。同样，对于循环体内的变量，在仿真过程中是无法观测的，读者可以自己动手将例 5-12 中的 cnt 加入仿真结果观测列表中，但并不能观察到其从 8 到 0 的递减过程，只能看到循环体执行后的数值 0。

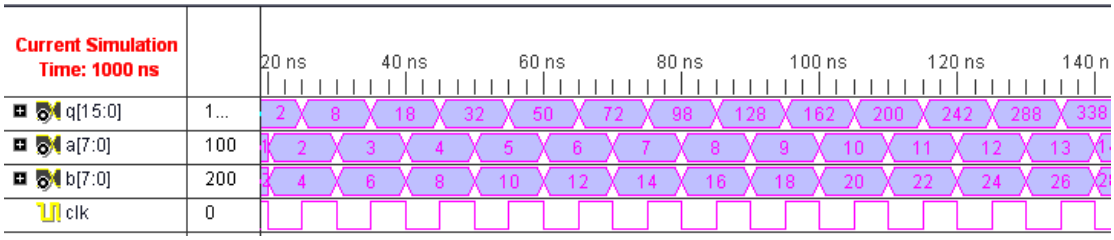


图 5-15　时序循环电路功能仿真示意图

虽然循环体通过组合逻辑设计实现，其时序性能不会太高，但基于时序逻辑能够达到 50MHz 的时序要求呢？

（2）下面对例 5-12 的 clk 信号加入 50MHz 的频率约束，其布局布线后仿真如图 5-16 所示。可以看出，其结果和功能仿真结果一致，表明在 50MHz 的频率下，硬件平台还能在 20ns（1/50MHz）时间内完成一个 8 循环。那么如果将例 5-12 中的位宽参数 bsize 修改成 16，那么还能在 20ns 完成一个 16 循环吗？结论是否定的，读者可自行动手验证。

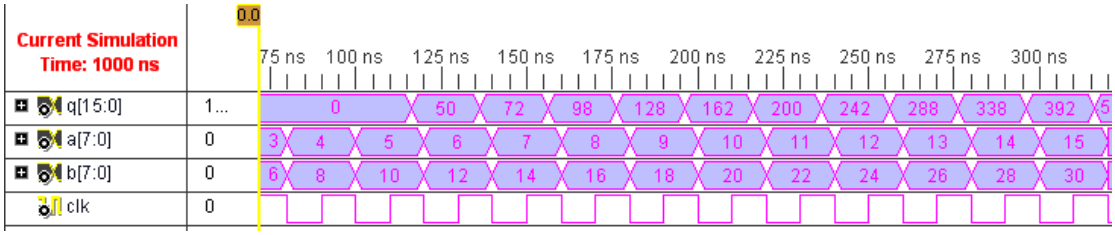


图 5-16　时序循环电路在 50MHz 时钟约束下的布局布线后仿真示意图

再进一步将 clk 的时序约束提高到 100MHz，完成布局布线后仿真，其结果如图 5-17 所示。从中可以看出，计算结果已经错误，乘法器不能正常工作。这意味着在硬件平台上，无法在 10ns 内完成一个 8 循环，自然也就无法在 20ns 的时间内完成一个 16 循环。

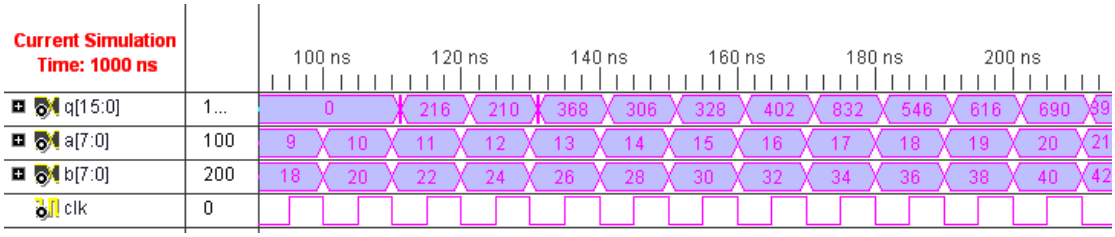


图 5-17　时钟约束 100MHz 的布局布线后仿真示意图

经过布局布线后，可以得到其所占用的逻辑资源，需要 70 个 Slice。

（3）为了便于比较，下面给出基于计数器实现乘法功能的 Verilog HDL 代码：

```
module mult_demo(
        clk, a, b, q
    );
      input              clk;
      input   [7 : 0]    a, b;
      output [15: 0]     q;
```

```
reg       [15:0]        tmp0, tmp1, tmp2, tmp3,
                          tmp4, tmp5, tmp6, tmp7;
reg       [15:0]        ad0,   ad1,   ad2;

always @(posedge clk) begin
    if(b[0] == 1'b1) begin
        tmp0 <= a;
     end
     else begin
        tmp0 <= 0;
     end

    if(b[1] == 1'b1) begin
        tmp1 <= a << 1;
     end
     else begin
        tmp1 <= 0;
     end

    if(b[2] == 1'b1) begin
        tmp2 <= a << 2;
     end
     else begin
        tmp2 <= 0;
     end

    if(b[3] == 1'b1) begin
        tmp3 <= a << 3;
     end
     else begin
        tmp3 <= 0;
     end

    if(b[4] == 1'b1) begin
        tmp4 <= a << 4;
     end
     else begin
        tmp4 <= 0;
     end

    if(b[5] == 1'b1) begin
        tmp5 <= a << 5;
     end
```

```
        else begin
            tmp5 <= 0;
        end

        if(b[6] == 1'b1) begin
            tmp6 <= a << 6;
        end
        else begin
            tmp6 <= 0;
        end

        if(b[7] == 1'b1) begin
            tmp7 <= a << 7;
        end
        else begin
            tmp7 <= 0;
        end

    ad0 <= tmp0 + tmp1 + tmp2 + tmp3;
    ad1 <= tmp4 + tmp5 + tmp6 + tmp7;

    ad2 <= ad0 + ad1;

end

assign    q = ad2;

endmodule
```

上述程序的功能仿真结果如图 5-18 所示，可以正确完成输入数据的乘法操作，从而达到设计要求。

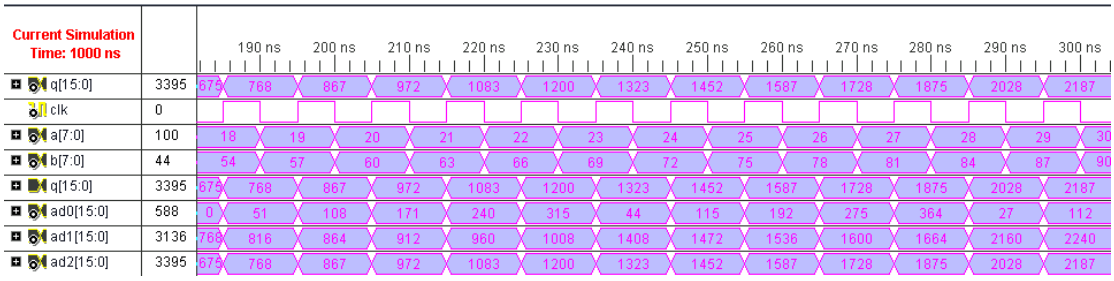


图 5-18　传统方式乘法器的功能仿真示意图

上例程序的时序性能报告如图 5-19 所示，可以看出，要求目标设计达到 50MHz 的设计性能（20ns），经过布局布线后的最差时延为 11.653ns，意味着将近 90MHz 的性能。其次，需要 72 个 Slice，和基于循环结构的程序相比，其所占逻辑资源并没有增加。

```
------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------------------------------
  TS_clk = PERIOD TIMEGRP "clk" 20 ns HIGH  | SETUP   |    11.653ns|     8.347ns|       0|           0
  50%                                       | HOLD    |     1.020ns|            |       0|           0
------------------------------------------------------------------------------------------------------


All constraints were met.
```

图 5-19　时序仿真结果示意图

（4）通过本例，可以得到一个基本结论：虽然基于循环语句的 Verilog HDL 设计显得相对精简，阅读起来也比较容易；但面向硬件的设计和软件设计的关注点是不同的，硬件设计并不追求代码的短小，而是设计的时序和面积性能等特征。读者在面向综合的设计中使用循环语句要慎重。

5.4 任务与函数

如果程序中如果有一段语句需要执行多次，则重复性的语句非常多，代码会变的冗长且难懂，维护难度也很大。任务和函数具备将重复性语句聚合起来的能力，类似于 C 语言的子程序。通过任务和函数来替代重复性语句，也有效简化程序结构，增加代码的可读性。此外，verilog 的 task 和 function 是可以综合的，不过综合出来的都是组合电路。

5.4.1　任务（TASK）语句

任务就是一段封装在“task-endtask”之间的程序。任务是通过调用来执行的，而且只有在调用时才执行，如果定义了任务，但是在整个过程中都没有调用它，那么这个任务是不会执行的。调用某个任务时可能需要它处理某些数据并返回操作结果，所以任务应当有接收数据的输入端和返回数据的输出端。另外，任务可以彼此调用，而且任务内还可以调用函数。

1．任务定义

任务定义的形式如下：

```
task task_id;
    [declaration]
    procedural_statement
endtask
```

其中，关键词 task 和 endtask 将它们之间的内容标志成一个任务定义，task 标志着一个任务定义结构的开始；task_id 是任务名；可选项 declaration 是端口声明语句和变量声明语句，任务接收输入值和返回输出值就是通过此处声明的端口进行的；procedural_statement 是一段用来完成这个任务操作的过程语句，如果过程语句多于一条，应将其放在语句块内；endtask 为任务定义结构体结束标志。下面给出一个任务定义的实例。

例 5-13：定义一个任务。

```
task task_demo;                    //任务定义结构开头，命名为 task_demo
    input   [7:0] x,y;             //输入端口说明
    output [7:0] tmp;              //输出端口说明

    if(x>y)                        //给出任务定义的描述语句
      tmp = x;
     else
       tmp = y;
```

endtask

上述代码定义了一个名为“task_demo”的任务，求取两个数的最大值。在定义任务时，有下列六点需要注意：

（1）在第一行“task”语句中不能列出端口名称；

（2）任务的输入、输出端口和双向端口数量不受限制，甚至可以没有输入、输出以及双向端口。

（3）在任务定义的描述语句中，可以使用/出现不可综合操作符合语句（使用最为频繁的就是延迟控制语句），但这样会造成该任务不可综合。

（4）在任务中可以调用其他的任务或函数，也可以调用自身。

（5）在任务定义结构内不能出现 initial 和 always 过程块。

（6）在任务定义中可以出现“disable 中止语句”，将中断正在执行的任务，但其是不可综合的。当任务被中断后，程序流程将返回到调用任务的地方继续向下执行。

2．任务调用

虽然任务中不能出现 initial 语句和 always 语句语句，但任务调用语句可以在 initial 语句和 always 语句中使用，其语法形式如下：

task_id[(端口 1, 端口 2, ..., 端口 N)];

其中 task_id 是要调用的任务名，端口 1、端口 2，…是参数列表。参数列表给出传入任务的数据（进入任务的输入端）和接收返回结果的变量（从任务的输出端接收返回结果）。任务调用语句中，参数列表的顺序必须与任务定义中的端口声明顺序相同。任务调用语句是过程性语句，所以任务调用中接收返回数据的变量必须是寄存器类型。下面给出一个任务调用实例。

例 5-14：通过 Verilog HDL 的任务调用实现一个 4 比特全加器。

```
module EXAMPLE (A, B, CIN, S, COUT);

  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;

  reg [3:0] S;
  reg COUT;
  reg [1:0] S0, S1, S2, S3;

  task ADD;

    input A, B, CIN;
    output [1:0] C;

    reg [1:0] C;
    reg S, COUT;

    begin
```

```
        S = A ^ B ^ CIN;
        COUT = (A&B) | (A&CIN) | (B&CIN);
        C = {COUT, S};
    end
  endtask

  always @(A or B or CIN) begin
    ADD (A[0], B[0], CIN, S0);
    ADD (A[1], B[1], S0[1], S1);
    ADD (A[2], B[2], S1[1], S2);
    ADD (A[3], B[3], S2[1], S3);
    S = {S3[0], S2[0], S1[0], S0[0]};
    COUT = S3[1];
  end
endmodule
```

上述代码在 ISE Simulator 中的仿真结果如图 5-20 所示，正确实现了加法器的功能，达到了设计目的。

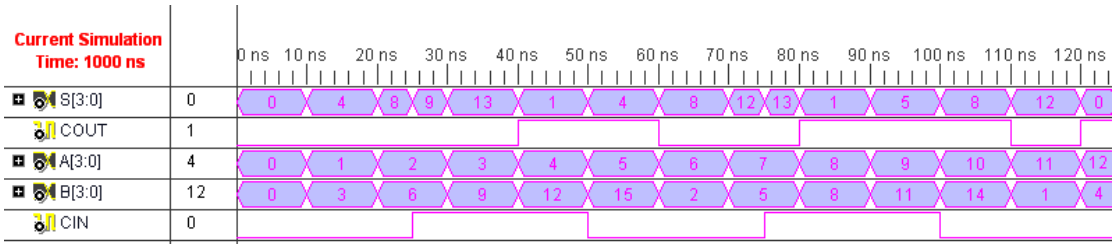


图 5-20　基于任务调用加法器的仿真结果

在调用任务时，需要注意以下几点：

（1）任务调用语句只能出现在过程块内；

（2）任务调用语句和一条普通的行为描述语句的处理方法一致；

（3）当被调用输入、输出或双向端口时，任务调用语句必须包含端口名列表，且信号端口顺序和类型必须和任务定义结构中的顺序和类型一致。需要说明的是，任务的输出端口必须和寄存器类型的数据变量对应。

（4）可综合任务只能实现组合逻辑，也就是说调用可综合任务的时间为“0”。而在面向仿真的任务中可以带有时序控制，如时延，因此面向仿真的任务的调用时间不为“0”。

5.4.2　函数（FUNCTION）语句

函数的功能和任务的功能类似，但二者还存在很大的不同。在 Verilog HDL 语法中也存在函数的定义和调用。

1．函数的定义

函数通过关键词 function 和 endfunction 定义，不允许输出端口声明（包括输出和双向端口），但可以有多个输入端口。函数定义的语法如下：

```
function [range] function_id;
    input_declaration
    other_declarations
    procedural_statement
endfunction
```

其中，function 语句标志着函数定义结构的开始；[range]参数指定函数返回值的类型或位宽，是一个可选项，若没有指定，默认缺省值为 1 比特的寄存器数据；function_id 为所定义函数的名称，对函数的调用也是通过函数名完成的，并在函数结构体内部代表一个内部变量，函数调用的返回值就是通过函数名变量传递给调用语句；input_declaration 用于对函数各个输入端口的位宽和类型进行说明，在函数定义中至少要有一个输入端口；endfunction 为函数结构体结束标志。下面给出一个函数定义实例。

例 5-15：定义函数实例。

```
function    AND;
  //定义输入变量
  input A, B;
  //定义函数体
  begin
      AND = A    && B;
  end
endfunction
```

函数定义在函数内部会隐式定义一个寄存器变量，该寄存器变量和函数同名并且位宽也一致。函数通过在函数定义中对该寄存器的显式赋值来返回函数计算结果。此外，还有下列几点需要注意：

（1）函数定义只能在模块中完成，不能出现在过程块中；

（2）函数至少要有一个输入端口；不能包含输出端口和双向端口；

（3）在函数结构中，不能使用任何形式的时间控制语句（#、wait 等），也不能使用 disable 中止语句；

（4）函数定义结构体中不能出现过程块语句（always 语句）；

（5）函数内部可以调用函数，但不能调用任务。

2．函数调用

和任务一样，函数也是在被调用时才被执行的，调用函数的语句形式如下：

```
func_id(expr1, expr2, ....., exprN)
```

其中，func_id 是要调用的函数名，expr1, expr2, ...exprN 是传递给函数的输入参数列表，该输入参数列表的顺序必须与函数定义时声明其输入的顺序相同。下面给出一个函数调用实例。

例 5-16：函数调用实例。

```
module comb15 (A, B, CIN, S, COUT);

  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;

  wire [1:0] S0, S1, S2, S3;

  function signed [1:0] ADD;

    input A, B, CIN;
```

```
    reg S, COUT;

    begin
      S = A ^ B ^ CIN;
      COUT = (A&B) | (A&CIN) | (B&CIN);
      ADD = {COUT, S};
    end
  endfunction

  assign S0 = ADD (A[0], B[0], CIN),
         S1 = ADD (A[1], B[1], S0[1]),
         S2 = ADD (A[2], B[2], S1[1]),
         S3 = ADD (A[3], B[3], S2[1]),
         S = {S3[0], S2[0], S1[0], S0[0]},
         COUT = S3[1];
endmodule
```

上述程序的仿真结果如图 5-21 所示，正确实现了加法器的功能，达到了设计目的。

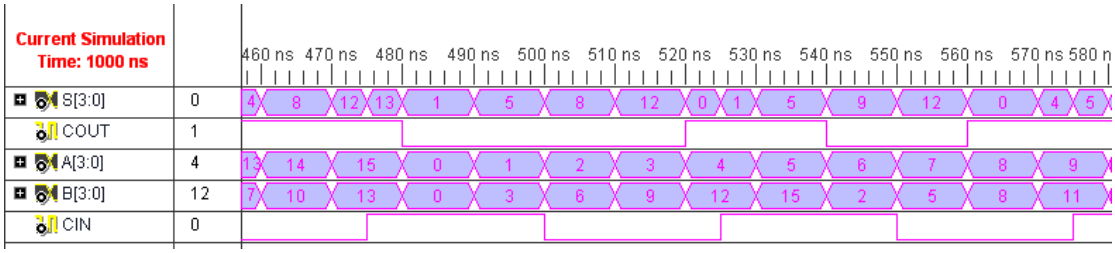


图 5-21　基于函数调用加法器的仿真结果

在函数调用中，有下列几点需要注意：

（1）函数调用可以在过程块中完成，也可以在 assign 这样的连续赋值语句中出现。

（2）函数调用语句不能单独作为一条语句出现，只能作为赋值语句的右端操作数。

5.4.3　任务和函数的深入理解

通过任务和函数可以将较大的行为级设计划分为较小的代码段，允许 Verilog HDL 程序开发人员将在多个地方使用的相同代码提取出来，简化程序结构，提高代码可读性。一般的综合器都是支持了 task 和 function 语句的。

1．关于 task 语句的深入说明

根据 Verilog HDL 语言标准上看来，task 比 always 低 1 个等级，即 task 必须在 always 里面调用，task 本身可以调用 task，但不能调用 Verilog HDL 模块（module）。module 的调用是与 always、assign 语句并列的，所以在这些语句中均不能直接调用 module，只能采用和 module 端口交互数据的方法达到调用的功能。

task 语句是可综合的，但其中不能包含 always 语句，因此也只能实现组合逻辑。顺序调用 task 对于电路设计来说，就是复制电路功能单元。多次调用 task 语句就是多次复制电路，因此资源会成倍增加，不能达到电路复用的目的；同时用 task 封装的纯逻辑代码会使得电路的处理时间变长，最高频率降低，不能应用于高速场合。

综上所述，可以看出 task 语句的功能就是将代码中重复的组合逻辑封装起来简化程序结构，具备组合逻辑设计的所有优点和缺点；而对于时序设计，task 语句则无法处理，只能通过 Verilog HDL 语言中的层次化设计方法，将其封装成 module，通过端口交换数据达到化

简程序结构的目的。

2．关于 function 语句的深入说明

在面向综合的设计中，function 语句是可综合的，但由于 function 语句中不支持使用 always 语句，因此无法捕获信号跳变沿，所以不可能实现时序逻辑。和 task 语句一样，function 语句具有组合逻辑电路的所有优点和缺点，这里就不再对其进行过多说明。

3．task 语句和 function 语句的比较

task 语句和 function 语句都必须在模块内部定义，除了参数个数不同外，还可以定义内部变量，包括寄存器、时间变量、整型等，但是不能定义线网型变量。此外，二者都只能出现在行为描述中，并且在 task 语句和 function 语句内部不能包含 always 和 initial 语句。

表 5-4 列出了 task 和 function 语句的不同点。

表 5-4　　任务和函数的区别

比较点	任务	函数
输入、输出	可以有任意多个各种类型的参数	至少有一个输入，不能有输出端口，包括 inout 端口
调用	任务只能在过程语句中调用，而不能在连续赋值语句 assign 中调用	函数可作为赋值操作的表达式，用于过程赋值和连续赋值语句
触发事件控制	任务不能出现 always 语句；可以包含延迟控制语句（#），但只能面向仿真，不可综合。	函数种不能出现（always、#）这样的语句，要保证函数的执行在零时间内完成。
调用其他函数和任务	任务可以调用其它任务和函数	函数只能调用函数，但不能调用任务
返回值	任务没有返回值	函数向调用它的表达式返回一个值
其他说明	任务调用语句可以作为一条完整的语句出现	函数调用语句只能作为赋值操作的表达式，不能作为一条独立的语句出现

5.5 本章小结

本章主要介绍 Verilog HDL 语言中可综合语句的用法。首先介绍了触发事件的控制语句，包括电平事件语句和跳变沿事件语句两大类。其次介绍了 if 和 case 这两类条件语句，需要明白 if 语句和 case 虽然功能类似，但本质是不同的，前者是串行执行的，后者是并行执行的。第三，介绍了循环语句，其中循环变量的增加不能采用“++”和“--”这两类操作符。最后，着重说明了任务和函数语句的用法。这些语句虽然在形式上和 C 语言很类似，语法等各方面比较容易理解，但读者要注意的是它们表示的不是一个直接的计算过程，而是逻辑电路硬件的行为，语句间细微的差别可能导致其对应的硬件有很大的变化。因此本章给出了大量的应用实例，希望读者认真理解这些语句的本质，包括各类细节，才能设计出符合要求的逻辑。

5.6 思考题

1．如何理解 always 语句引导的过程块是不断活动的？
2．Verilog HDL 的触发事件可以分为哪几类？如何通过 Verilog HDL 语言实现？
3．if 语句有什么特点？其与 case 语句有什么区别和联系？
4．说明 case、casex 和 casez 语句之间的不同？
5．可综合的循环语句包括哪些？
6．什么是任务，有什么特点？

7．什么是函数，有什么特点？其与任务有什么不同？
8．任务和函数是可综合的吗？二者能否用于实现时序逻辑？

第 6 章　面向验证和仿真的行为描述语句

随着设计规模的不断增大，验证任务在设计中所占的比例越来越大，已成为 Verilog HDL 设计流程中非常关键的一个环节，传统的验证手段已无法满足需求。事实上，Verilog HDL 语言有着非常强的行为建模能力，可以方便地写出高效、简洁的测试代码。验证包含了功能验证、时序验证以及形式验证等诸多内容，其中，对于大多数急于可编程逻辑器件的应用来讲，不存在后端处理，因此功能验证占据了验证的绝大部分工作。本章首先介绍关于验证的一些基本概念，然后重点说明 Verilog HDL 仿真语句的使用方法。通过本章的学习，读者可真切感受到 Verilog HDL 语言强大的行为级建模能力。

6.1 验证与仿真概述

6.1.1 代码验证与仿真概述

1．验证的基本概念

在 Verilog HDL 语言设计中，整个流程的各个环节都离不开验证，一般分为四个阶段：功能验证、综合后验证、时序验证和板级验证。其中前三个阶段只能在 PC 上借助 EDA 软件，通过仿真手段完成；第四个步骤则将设计真正地运行在硬件平台（FPGA、ASIC 等）上，即可借助传统的调试工具（示波器、逻辑分析仪等）来验证系统功能，也可以通过灵活、先进的软件调试工具（在线逻辑分析仪，例如 Xilinx 公司的 Chipscope Pro 软件）来直接调试硬件。

仿真，也就是模拟，是对所设计电路或系统输入测试信号，然后根据其输出信号和期望值是否一致得到设计正确与否的结论。由于综合后验证主要通过察看 RTL 结构来检查设计，因此常用的仿真包括功能仿真和时序仿真。Verilog HDL 语言不仅可以描述设计，还能提供对激励、控制、存储响应和设计验证的建模能力。Verilog HDL 测试代码主要用于产生测试激励波形以及输出响应数据的收集。

要对设计进行仿真验证，必须有仿真软件的支持。按照对设计代码的处理方式，可将仿真工具分为编译型仿真软件和解释型仿真软件两大类。编译型仿真软件的速度相对较快，但需要预处理，因此不能即时修改；解释型仿真器的速度较慢，但可以随时修改仿真环境和条件。按照 HDL 语言类型，可将仿真软件分为 Verilog HDL 仿真器、VHDL 仿真器和混合仿真器三大类。

常用的仿真工具有 Mentor Graphic 公司的 ModelSim、Cadence 公司的 NC-Verilog 和 Verilog-XL 以及 Xilinx 公司的 ISE-Simulator 等，都能提供 Verilog HDL 和 VHDL 的混合仿真。其中，ModelSim 和 ISE-Simulator 属于基于编译的仿真软件，能快速完成功能和时序仿真；而 Verilog-XL 则是基于解释的仿真软件，速度相对较慢。

验证与仿真是否准确与完备，在一定程序上决定了所设计系统的命运，可以说无缺陷的系统不是设计出来的，而是验证出来的。因此在大型系统设计中，验证和仿真所占用的时间往往是设计阶段所有时间的数倍。

2．收敛模型

在编写验证代码时，首先要清楚地是：验证对象是什么？验证目的是什么？为了回答这个问题，需要首先给出收敛模型的概念（基于 Verilog HDL 的数字系统应用设计）。

收敛模型是验证过程的抽象描述，主要包括两方面内容：首先给出验证任务的说明；其

次通过检查任务验证和转换是否收敛于共同的起点来证明转换的正确性。这里的转换是一个广义的概念，对于整个设计而言是指从设计需求说明到最终硬件系统的转换，对于功能验证而言是指从需求说明到可综合的 Verilog HDL 代码的转换。对一个转换的验证只能通过同一个起点的另外一条收敛路径去完成对一个转换的验证，如图 6-1 所示[1]。

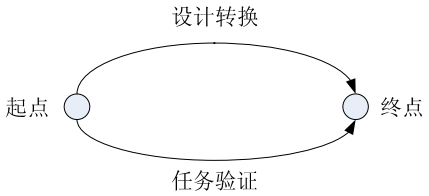


图 6-1　转换和验证原理

因此，对于功能验证来讲，通过验证要保证可综合代码正确实现了设计需求。这就意味着验证和设计需要有共同的起点（这个起点就是设计需求说明书），否则，验证和设计也就没有共同的收敛点，相当于实际上没有做验证。

此外，在实际设计操作中，大多数读者处于收敛模型中的一个误区：代码设计者自己验证自己的设计。如果设计者集设计和验证任务于一身，其验证的起点是设计者自己对需求说明书的理解，而不是需求说明书本身，如图 6-2 所示。这就使得设计者只能验证自己是否正确地将自己对说明书的理解转换成了 Verilog HDL 代码，而不能验证自己是否正确理解了需求说明书。一旦出现理解错误，将不能被检查出来。

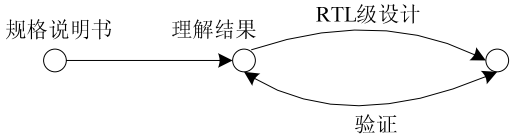


图 6-2　设计和验计合并的收敛模型

为了避免上述误区，实际中要求设计和验证相互独立，分别由不同的团队完成。设计者完成代码设计以及模块级的验证，验证人员完成系统级的测试。这样，设计和验证的起点都是设计需求说明书，如图 6-3 所示。这样，理解错误的概率由原来的 x%降低为$(x\%)^2$，从而减少甚至消除由于主观理解不正确而引起的设计错误。

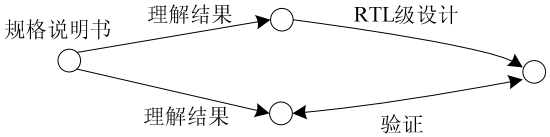


图 6-3　设计和验证独立的收敛模型

6.1.2　测试平台说明

1．测试平台综述

当完成所需硬件模块的 Verilog HDL 语言程序后，验证其实现的功能和性能与设计规范是否相吻合，就成为设计人员的首要任务。

一般来讲，完成设计的硬件都有一个顶层模块，该模块定义了系统中所有的外部接口，例化（调用）了各底层模块并完成正确的连接，以实现层次化开发。要对所设计的硬件进行功能验证，就要对顶层模块的各个对外接口提供符合设计规范要求的测试输入（也称为测试激励），然后观察其输出和中间结构是否满足要求。在实践中，往往通过测试平台（Testbench）来为顶层模块输入激励，并例化待验证设计（DUT），且监视 DUT 的输出，如图 6-4 所示。

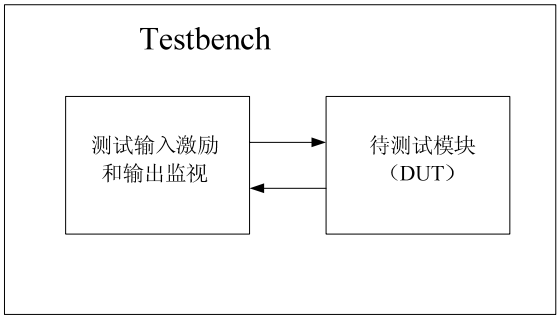


图 6-4　Testbench 示意图

对于简单的设计，特别是一些面向 CPLD 器件的开发，直接利用仿真工具内嵌的波形编辑工具绘制激励，然后进行仿真验证；对于 FPGA 的一般设计，特别是大型设计，则适合通过 Verilog HDL 语言编写 Testbench，通过软件工具比较结果，分析设计的正确性以及 Testbench 自身的覆盖率，发现问题及时修改。

2．Testbench 模型

Testbench 的概念为设计人员提出了一个高效、灵活的设计验证平台，其主要思想就是在不需要硬件外设的前提下，采用模块化的方法完成代码验证。Verilog HDL 还可以用来描述变化的测试信号，它可以对任何一个 DUT 模块进行动态的全面测试。此外，Testbench 设计好以后，可应用于各类验证，例如功能验证和时序验证就采用同一个 Testbench。因此，如何高效、规范、完备地编写测试激励是本章的重点。

（1）传统模型

传统的 Testbench 模型如图 6-5 所示，直接起测试 DUT 模块的输出值。从中可以看出，Testbench 最主要的任务就是提供完备的测试激励以及例化 DUT 模块。后端的比较、检查任务则依赖 EDA 工具。

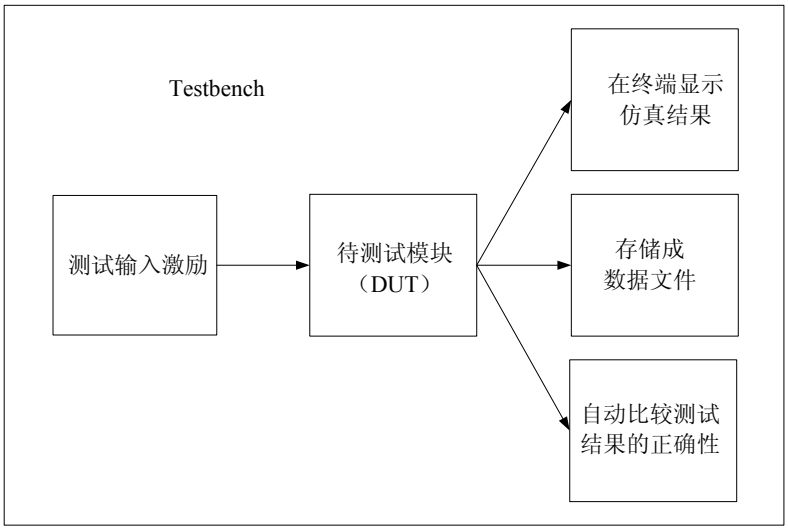


图 6-5　Testbench 的传统模型示意图

传统模型的优点是直观准确，能有效覆盖设计的全部功能。其缺点有两点：首先，需要事先计算期望输出，当数据通道比较复杂时，需要消耗很多时间去计算输出，从而难以使用随机测试信号，存在验证漏洞；其次，验证代码的可重用性很差。

（2）参考模型

参考模型如图 6-6 所示，不仅要例化 DUT 模块，还要实现一个参考模块，然后为二者提供同样的输入，直接比较输出结果是否一致，得到验证结论。

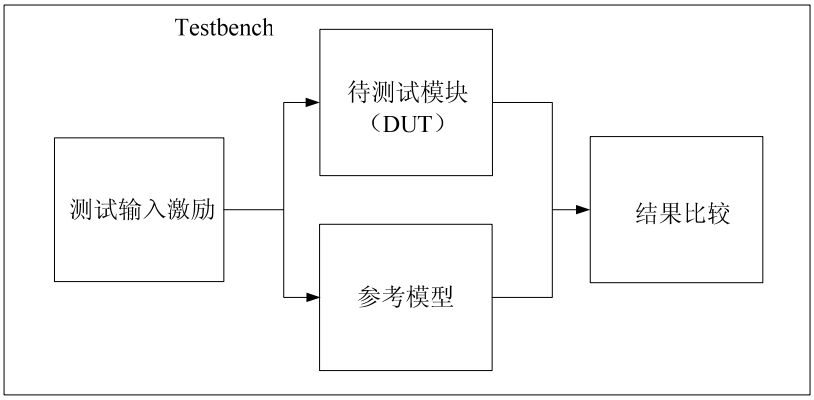


图 6-6　Testbench 参考模型示意图

参考模型的优点是具备良好的可重性，并且可以方便地使用随机测试向量。其缺点是，需要对被测对象建立参考模型，从而使得前期的工作量非常大。因此，对于小型设计，效率反而不高，但适合于大型或复杂设计，特别是与数字信号处理有关的设计。

Verilog HDL 语言中所有语句和关键字操作，包括面向综合、面向仿真以及系统级任务都可用于 Testbench 的书写，产生测试激励。

6.1.3　验证测试方法论

了解了如何利用 Testbench 来进行验证后，接着来介绍基本的验证测试方法，只有这样才能在最短的时间内发现尽可能多的错误，并少走弯路，提高测试效率。当一个大规模的系统设计完成后，将不可避免地出现各种各样的错误，再加上随着硬件复杂度的级数级增加，验证已成为硬件设计的瓶颈。高效、完备的测试成为必需要求。

其中，高效是指尽快发现错误，这是由越来越短的上市时间决定的，需要设计人员利用多种 EDA 工具生成各类测试向量，以在尽可能短的时间内完成验证。完备则指发现全部错误，要求硬件测试要达到一定的覆盖率，包括代码的覆盖率和功能的覆盖率。

1．功能验证方法

目前的功能验证方法有很多种，下面主要介绍黑盒测试法、白盒测试法以及灰盒测试法这三类主要验证方法。

（1）黑盒测试法

对于 Verilog HDL 设计，从代码角度来看，可以把一个设计模块看作是一个构件；从硬件的角度来看，可以把一个设计模块看作一个集成块。但不论怎样，都可以把它看作一个黑盒，从而可以运用黑盒测试的有关理论和方法对它进行测试验证。

黑盒测试是把 Verilog HDL 设计看作一个“黑盒子”，不考虑程序内部结构和特性，在程序接口进行测试。测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。黑盒测试方法主要有等价类划分、边值分析、因果图、错误推测等，主要用于功能测试。

黑盒测试方法在代码接口上进行测试，目的是发现以下几类错误：

- 是否有不正确或遗漏了的功能。
- 在接口上，输入能否正确地接受，输出正确的结果。
- 是否有数据格式错误或外部信息访问错误。
- 性能上是否能够满足要求。
- 是否有初始化或终止性错误。

用黑盒测试发现程序中的错误，必须在所有可能的输入条件和输出条件中确定测试数据，来检查程序是否都能产生正确的输出。

黑盒测试法的优点在于两点：首先，简单，验证人员无须了解程序的细节，只需要根据设计需求说明书搭建测试代码；其次，便于达到设计和验证分离的目的，保证测试人员不会受到设计代码的影响。其缺点则是可观性差，由于验证人员对内部的实现细节不太了解，无法对错误进行快速定位，在大规模设计中很难跟踪错误的来源。所以，黑盒测试法一般用于中、小规模设计。

（2）白盒测试法

和黑盒测试相反，白盒测试要求验证人员对 Verilog HDL 设计内部的细节完成细致性检查。这种方法首先要求验证人员对设计熟悉，从而将测试对象看做一个打开的盒子，利用程序内部的逻辑结构及有关信息、设计或选择测试用例，对程序所有逻辑路径进行测试。通过在不同点检查程序状态，确定实际状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。白盒测试主要是想对程序模块进行如下检查：

- 对程序模块的所有独立的执行路径至少测试一遍。
- 对所有的逻辑判定，取“真”与取“假”的两种情况都能至少测一遍。

白盒测试法的优点在于容易观察和控制验证的进展情况，可以通过事先设置的观测点，在错误出现后很快定位问题的根源。其缺点则是需要耗费很长的时间去了解设计代码，且很难做到设计和验证分离，从而使得验证人员深受设计影响，从而无法全面验证设计功能的正确性。

（3）灰盒测试法

灰盒测试是介于白盒测试与黑盒测试之间的一种测试方法。可以这样理解，灰盒测试关注输出对于输入的正确性，同时也关注内部表现，但这种关注不象白盒那样详细、完整，只是通过一些表征性的现象、事件、标志来判断内部的运行状态。在很多测试中，经常会出现输出正确、内部错误的情况，如果每次都通过白盒测试来操作，效率会很低，因此需要采取灰盒测试法。灰盒测试法的优缺点介于黑盒和白盒之间。

在实际应用中，验证人员经常在 Verilog HDL 代码之间插入测试点，以快速定位问题。下面给出一个实例说明黑盒测试和灰盒测试的区别。

例 6-1：黑盒测试、白盒测试以及灰盒测试说明实例。

下面通过一个流程分为 5 步的设计进行各类测试方法的深入讨论。

（1）黑盒测试法的方法只送进不同组合的最原始端输入，然后直接在 5步流程的后的输出端收集数据，判断其是否正确。其特点就是从宏观整体入手，而不进入被测试模块。

（2）白盒测试法，则采用分布式的方法，首先理解全部代码，然后测试 5 步流程中的每一个细节，依次往上，完成每步流程的单独测试，最后再从第一个流程开始，依次级联下一步流程，完成测试。其特点是容易观察并控制验证，但需要耗费大量的时间去理解程序。

（3）灰盒测试法，则在整体设计的关键处插入观测点，以每步流程为起点，单独测试；成功后再完成整体测试。这样，不仅可以快速定位错误，也减少了测试的工作量。目前，大部分测试都基于灰盒测试。

本书的验证代码都是基于灰盒测试思想的，如果设计复杂，则加入关键信号的波形分析；否则，直接观测设计的最终输出。

2．时序验证方法

（1）时序验证说明

在以往的小规模设计中，验证环节通常只需要做动态的门级时序仿真，就可同时完成对 DUT（Design Under Test，被测试设计）的逻辑功能验证和时序验证。随着设计规模和速度的不断提高，要得到较高的测试覆盖率，就必须编写大量的测试向量，这使得完成一次门级时序仿真的时间越来越长。为了提高验证效率，有必要将 DUT 的逻辑功能验证和时序验证分开，分别采用不同的验证手段加以验证。

首先，电路逻辑功能的正确性，可以由 RTL 级的功能仿真来保证；其次，电路时序是否满足，则通过 STA（Static Timing Analysis，静态时序分析）得到。两种验证手段相辅相成，确保验证工作高效可靠地完成。时序分析的主要作用就是察看 FPGA 内部逻辑和布线的延时，验证其是否满足设计者的约束。在工程实践中，主要体现在以下 3 点：

- 确定芯片最高工作频率

更高的工作频率意味着更强的处理能力，通过时序分析可以控制工程的综合、映射、布局布线等关键环节，减少逻辑和布线延迟，从而尽可能提高工作频率。一般情况下，当处理时钟高于 100MHz 时，必须添加合理的时序约束文件以通过相应的时序分析。

- 检查时序约束是否满足

可以通过时序分析来察看目标模块是否满足约束，如果不能满足，可以通过时序分析器来定位程序中不满足约束的部分，并给出具体原因。然后，设计人员依此修改程序，直到满足时序约束为止。

- 分析时钟质量

时钟是数字系统的动力系统，但存在抖动、偏移和占空比失真等 3 大类不可避免的缺陷。要验证其对目标模块的影响有多大，必须通过时序分析。当采用了全局时钟等优质资源后，如果仍然是时钟造成目标模块不满足约束，则需要降低所约束的时钟频率。

- 确定分配管脚特性

FPGA 的可编程特性使电路板设计加工和 FPGA 设计可以同时进行，而不必等 FPGA 引脚位置完全确定后再进行，从而节省了系统开发时间。通过时序分析可以指定 I/O 引脚所支持的接口标准、接口速率和其它电气特性。

（2）静态时序分析说明

早期的电路设计通常采用动态时序验证的方法来测试设计的正确性。但是随着 FPGA 工艺向着深亚微米技术的发展，动态时序验证所需要的输入向量将随着规模增大以指数增长，导致验证时间占据整个芯片开发周期的很大比重。此外，动态验证还会忽略测试向量没有覆盖的逻辑电路。因此静态时序分析（STA，Static Timing Analysis）应运而生，它不需要测试向量，即使没有仿真条件也能快速地分析电路中的所有时序路径是否满足约束要求。STA 的目的就是要保证 DUT 中所有路径满足内部时序单元对建立时间和保持时间的要求。信号可以及时地从任一时序路径的起点传递到终点，同时要求在电路正常工作所需的时间内保持恒定。整体上讲，静态时序分析具有不需要外部测试激励、效率高和全覆盖的优点，但其精确度不高。

STA 是通过穷举法抽取整个设计电路的所有时序路径，按照约束条件分析电路中是否有违反设计规则的问题，并计算出设计的最高频率。和动态时序分析不同，STA 仅着重于时序性能的分析，并不涉及逻辑功能。STA 是基于时序路径的，它将 DUT 分解为 4 种主要的时序路径，如图 6-7 所示。每条路径包含一个起点和一个终点，时序路径的起点只能是设计的基本输入端口或内部寄存器的时钟输入端，终点则只能是内部寄存器的数据输入端或设计的基本输出端口。

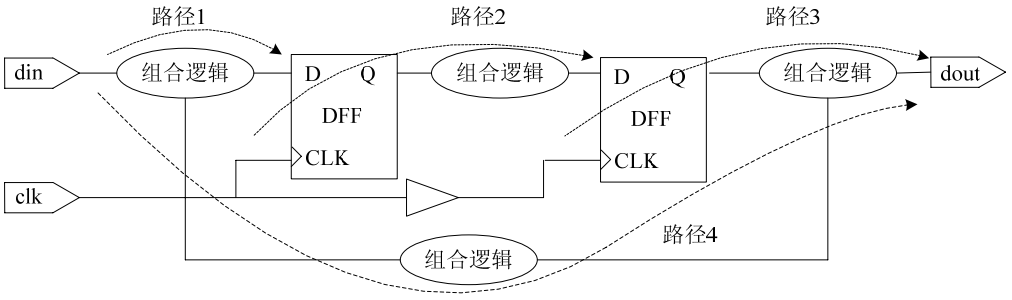


图 6-7　静态时序分析的基本路径

STA 的四类基本时序电路为：

- 从输入端口到触发器的数据 D 端；
- 从触发器的时钟 CLK 端到触发器的数据 D 端；
- 从触发器的时钟 CLK 端到输出端口；
- 从输入端口到输出端口。

静态时序分析在分析过程中计算时序路径上数据信号的到达时间和要求时间的差值，以判断是否存在违反设计规则的错误。数据的到达时间指的是：数据沿路从起点到终点经过的所有器件和连线延迟时间之和。要求时间是根据约束条件（包括工艺库和 STA 过程中设置的设计约束）计算出的从起点到达终点的理论时间，默认的参考值是一个时钟周期。如果数据能够在要求时间内到达终点，那么可以说这条路径是符合设计规则的。其计算公式如式(6-1)所示：

$$Slack = Trequired\_time - Tarrival\_time \tag{6-1}$$

其中 $Trequired\_time$ 为约束时长，$Tarrival\_time$ 为实际时延，$Slack$ 为时序裕量标志，正值表示满足时序，负值表示不满足时序。$Tarrival\_time$ 的具体计算见下节。STA 把式(6-1)作为理论依据，分析设计电路中的所有时序路径。如果得到的 STA 报告中 $Slack$ 为负值，那么此时序路径存在时序问题，是一条影响整个设计电路工作性能的关键路径。在逻辑综合、整体规划、时钟树插入、布局布线等阶段进行静态时序分析，就能及时发现并修改关键路径上存在的时序问题，达到修正错误、优化设计的目的。

3．覆盖率检查

覆盖率表征一个设计的验证所进行的程度，主要根据仿真时统计代码的执行情况，可以按陈述句、信号拴、状态机、可达状态、可触态、条件分支、通路和信号等进行统计分析，以提高设计可信度。

覆盖率一般表示一个设计的验证进行到什么程度，也是一个决定功能验证是否完成的重要量化标准之一。覆盖主要指的是代码覆盖和功能覆盖。

（1）代码覆盖

代码覆盖可以在仿真时由仿真器直接给出，主要用来检查 RTL 代码哪些没有被执行到。使用代码覆盖可以有效地找出冗余代码，但是并不能很方便地找出功能上的缺陷。

（2）功能覆盖

使用功能覆盖则可以帮助设计人员找出功能上的缺陷。一般说来，对一个设计覆盖点的定义和条件约束是在验证计划中提前定义好的，然后在验证环境中具体编程实现，把功能验证应用在约束随机环境中可以有效检查是否所有需要出现的情况都已经遍历。功能验证与面向对象编程技术结合可以在验证过程中有效地增减覆盖点。这些覆盖点既可以是接口上的信号，也可以是模块内部的信号，因此既可以用在黑盒验证也可以用在白盒验证中。通过在验证程序中定义错误状态可以很方便地找出功能上的缺陷。下面通过实例说明代码覆盖和功能覆盖的区别。

例 6-2：条件语句的覆盖率测试说明实例。

下面给出一段基于 if 语句的互斥条件语句，其代码如下：

```
if(cnt <3 && cnt > 5) begin //互斥条件
    x = 1;   //语句 1
end
else begin
    x = 0;   //语句 2
end
```

代码覆盖率检查就是测试代码哪部分被执行了，哪部分没有执行，从而找出错误进一步修改测试条件。由于语句 1 的条件是互斥的，x=1 这条语句用于不会执行，因此无论加入什么测试向量，上段代码在测试中，x 的值一直是 0，这样便会去寻找为什么 x 不输出 1 的原因，从而发现 if 语句的互斥条件，从而发现错误并修改。

6.1.4 Testbench 结构说明

Testbench 模块没有输入输出，在 Testbench 模块内例化待测设计的顶层模块，并把测试行为的代码封装在内，直接对待测系统提供测试激励。下面给出了一个基本的 Testbench 结构模板。

```
module testbench;
  //数据类型声明
  //对被测试模块实例化
  //产生测试激励
  //对输出响应进行收集
endmodule
```

一般来讲，在数据类型声明时，和被测模块的输入端口相连的信号定义为 reg 类型，这样便于在 initial 语句和 always 语句块中对其进行赋值；和被测模块输出端口相连的信号定义为 wire 类型，便于进行检测。可以看出，除了没有输入输出端口，testbench 模块和普通的 Verilog HDL 模块没有区别。Testbench 模块最重要的任务就是利用各种合法的语句，产生适当的时序和数据，以完成测试，并达到覆盖率要求。

下面给出一些在编写 Testbench 时需要注意的问题：

（1）Testbench 代码不需要可综合

Testbench 代码只是硬件行为描述而不是硬件设计。第 5 章所介绍的语句全部面向硬件设计，必须是可综合语句，每一条代码都对应着明确的硬件结构，能被 EDA 工具所理解。而 Testbench 只用于在仿真软件中模拟硬件功能，不会被实现成电路，也不需要具备可综合性。因此，在编写 Testbench 的时候，需要尽量使用抽象层次高的语句，不仅具备高的代码书写效率，而且准确、仿真效率高。

（2）行为级描述优先

如前所述，Verilog HDL 语言具备 5 个描述层次，分别为开关级、门级、RTL 行为级、算法级和系统级。虽然所有的 Verilog HDL 语言都可用于 Testbench 中，但是其中行为级描述代码具有以下显著优势：

- 降低了测试代码的书写难度，使得设计人员不需要理解电路的结构和实现方式，从而节约了测试代码开发时间。
- 行为级描述便于根据需要从不同的层次进行抽象设计。在高层描述中，设计会更加简单、高效，只有需要解析某个模块的详细结构时，才需要使用低层析的详细描述。
- 行为级仿真速度快。首先，各 EDA 工具本身就支持 Testbench 中的高级数据结构和运算，其编译和运行速度快；其次，高层次的设计本身就是对电路处理的一种简化。

基于上述原因，推荐在书写 Testbench 代码时使用行为级描述。

（3）掌握结构化、程式化的描述方法

结构化的描述有利于设计维护，由于在 Testbench 中，所有的 initial、always 以及 assign 语句都是同时执行的，其中每个描述事件都是基于时间“0”点开始的，因此可通过这些语句将不同的测试激励划分开来。一般不要将所有的测试都放在一个语句块中。

其次，对于常用的 Verilog HDL 测试代码，诸如时钟信号、CPU 读写寄存器、RAM 以及用户自定义事件的延迟和顺序等应用，已经形成了程式化的标准写法，因此应当大量阅读

这些优秀的仿真代码，积累程式化的描述方法，可有效提高设计 Testbench 的能力。

6.2 仿真程序执行原理

仿真程序执行原理从根本上说明了计算机的串行操作如何去模拟硬件电路的并行特征，以及可综合语句的执行过程，是真切理解 Verilog HDL 的基础。本部分内容根据文献 M、M、M ，再加上作者自己的理解构成，在此特别说明。希望读者能通过本章内容对 Verilog HDL 的仿真语句有一定的深入了解。

6.2.1 Verilog HDL 语义简介

由于 Verilog HDL 是用于硬件设计的，因此可综合语句都对应着实实在在的硬件电路，本质上是一种并行语言。而 EDA 软件都是运行在 PC 机上的，PC 上所有的程序都是串行执行的，CPU 在同一时刻执行执行一个任务，因此 EDA 软件（包括仿真器）也必然是串行的。因而这里就产生一个疑惑：顺序执行的仿真器怎么完成并行语言的仿真验证？

在仿真中，Verilog HDL 语句也是串行执行的，其面向硬件的并行特性则通过其语言含义（语义）来实现的。关于 Verilog HDL 语义的理解和建模是一个深入的课题，本书限于篇幅不对其进行专门介绍。读者只需明白，虽然在仿真中所有代码是串行执行的，但由于语法语义的存在，并不会丢失代码的并行含义和特征。

从不同角度（面向综合的应用以及面向仿真的应用）来讲，深入理解 Verilog HDL 语义都可以大幅提高设计人员的编码能力。由于在 Verilog HDL 语言的 IEEE 标准中，其语义采用非形式化的描述方法，因此不同厂家的仿真工具、综合语句的后台策略肯定存在差异，同一段代码在不同仿真软件的运行结果可能是不同的，也会导致设计人员对程序的理解产生偏差等问题。

对于普通用户，在一般设计中无需考虑上述差异。文献[2]对 Verilog HDL 的形式化语义有着深入的研究，有兴趣的读者，可以阅读文献[2]。

6.2.2 Verilog HDL 仿真原理

这一节的仿真原理主要介绍仿真的关键元素，包括仿真时间、事件驱动、进程以及调度等。下面分别对其进行说明。

1．仿真时间

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。0 时刻为仿真起始时刻。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未来仿真时间。

读者需要注意的是，仿真时间只是对电路行为的一个时间标记，和仿真程序在 PC 机上的运行时间没有关系。对于一个很复杂的程序，尽管只需要很短的仿真时间，也需要仿真器运行较长的时间；而对于简单的程序，即时仿真很长时间，也只需要短的运行时间。在 ISE 仿真工具 Simulator 的仿真结果中，最上面一栏的以“ns”为单位的数字就是仿真时间，如图 6-8 所示。本质上，仿真时间是没有单位的，之所以会出现图 6-8 所示的“**ns”，则是由于`timescale 语句的定义导致，7.2.4 节将对其进行专门介绍。

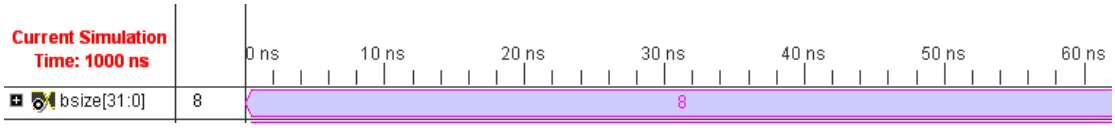


图 6-8　ISE Simulator 仿真时间示意图

所有的仿真事件都是严格按照仿真时间向前推进的，也就是说在恰当的时间执行恰当的

操作。如果在同一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来判定谁先执行。如果优先级相同，则不同仿真器的执行方式是不同的，有可能随机，也有肯定按照代码出现的顺序来执行。大多数仿真器采用后一种方法。

2．事件驱动

如果没有事件驱动，控制仿真时间将不会前进。仿真时间只能被下列事件中的一种来推进：

- 定义过的门级或线传输延迟
- 更新事件
- 由“#”关键字引入的延迟控制
- 由“always”关键字引入的事件控制
- 由“wait”关键字引入的等待语句

其中第 1 种形式是由门级器件来决定的，无须讨论。更新事件指线网、寄存器数值的任何改变。本章后续内容会对后三种形式以及路径延迟的定义分别进行讲述。事实上，上述事件都是循环、相互触发，来共同推动仿真时间的前进。

3．事件队列与调度

Verilog 具有离散事件时间仿真器的特性，也就是说在离散的时间点，预先安排好各个事件，并将它们按照时间顺序排成事件等待队列。最先发生的事件排在等待队列的最前面，而较迟发生的事件依次放在其后。仿真器总是为当前仿真时间移动整个事件队列，并启动相应的进程。在运行的过程中，有可能为后续进程生成更多的事件放置在队列中适当的位置。只有当前时刻所有的事件都运行结束后，仿真器才将仿真时间向前推进，去运行排在事件队列最前面的下一个事件。

在 Verilog 中，事件队列可以划分为 5 个不同的区域，不同的事件根据规定放在不同的区域内，按照优先级的高低决定执行的先后顺序，图 6-9 就列出了部分 Verilog 分层事件队列。其中，活跃事件的优先级最高（最先执行），而监控事件的优先级最低，而且在活跃事件中的各事件的执行顺序是随机的（注：为方便起见，在一般的仿真器中，对同一区域的不同事件是按照调度的先后关系执行的）。

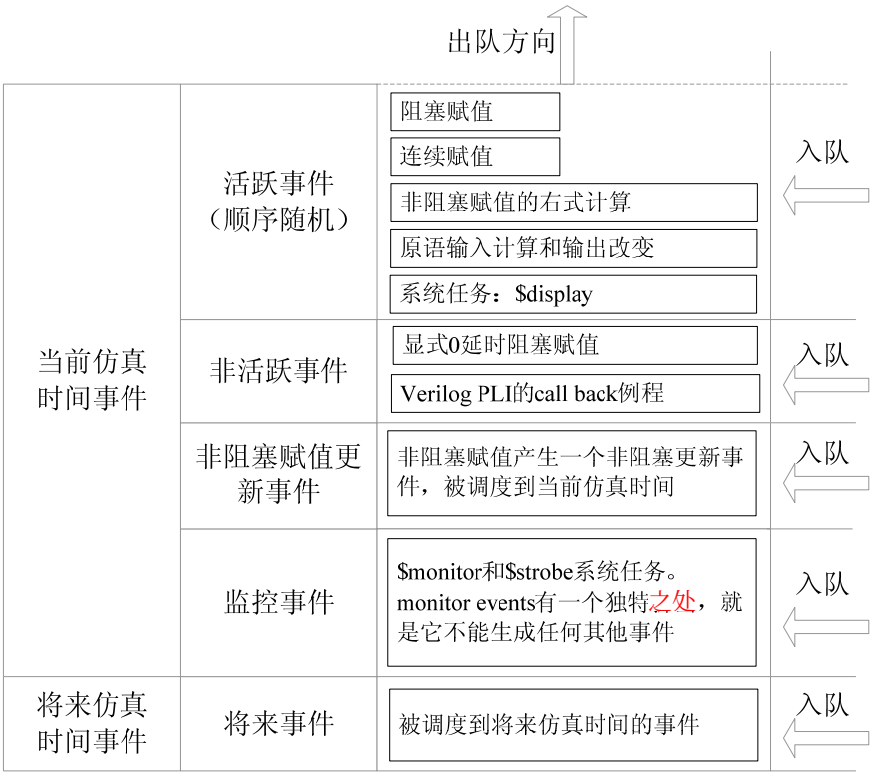


图 6-9　Verilog HDL 分层事件队列

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，非活跃事件的优先级次之，非阻塞赋值的优先级为第三，监控事件的优先级第四；将来事件的优先级最低。将来仿真时间内的所有事件都将暂存到将来事件队列中，当仿真进程推进到某个时刻后，该时刻所有的事件都会被加入当前仿真事件队列内。

由图 6-9 可以知道，阻塞赋值属于活跃事件，会立刻执行，这就是阻塞赋值“计算完毕，立刻更新”的原因。此外，由于在分层事件队列中，只有将活跃事件中排在前面的事件调出，并执行完毕后，才能够执行下面的事件。

虽然 IEEE Verilog HDL 标准定义了上述的层次化事件队列，但文献[3]指出，EDA 仿真软件的制造商如何来实现上述事件队列，由于关系到仿真器的效率，被视为商业机密，因此本书也不涉及该内容。

6.3　延时控制语句

6.3.1　延时控制的语法说明

延迟语句用于对各条语句的执行时间进行控制，从而快速满足用户的时序要求。Verilog HDL 语言中延时控制的语法格式有两类：

（1）# <延迟时间> 行为语句;

（2）#<延迟时间>;

其中，符号“#”是延迟控制的关键字符，“<延迟时间>”可以是直接指定的延迟时间量，并以多少个仿真时间单位的形式给出。在仿真过程中，所有时延都根据时间单位定义。下面是带时延的连续赋值语句示例：

assign #2 Sum = A ^ B;

#2 指 2 个时间单位。

使用编译指令将时间单位与物理时间相关联。这样的编译器指令需在模块描述前定义，如下所示：

` timescale 1ns /100ps

此语句说明时延时间单位为 1ns 并且时间精度为 100ps（时间精度是指所有的时延必须被限定在 0.1ns 内）。如果此编译器指令所在的模块包含上面的连续赋值语句，#2 代表 2ns。如果没有这样的编译器指令，Verilog HDL 模拟器会指定一个缺省时间单位，IEEE Verilog HDL 标准中没有规定缺省时间单位，因此由各 EDA 工具厂家自行设定。在 Xilinx 公司的 ISE 工具中，默认时间单位为 ns。

6.3.2 延时控制应用实例

在实际的仿真测试中，延迟控制语句可以出现在任何赋值语句中，主要有下列三类应用方式。

1．# <延迟时间常量> 行为语句;

在这种方式中，<延时时间常量>后面直接跟着一条行为语句。仿真进程遇到这条语句后，并不会立即执行行为语句指定的操作，而是要等到<延迟时间值>所指定的时间过去之后，才开始执行行为语句的操作。下面给出一个操作实例。

例 6-3：“#”语句的应用实例 1。

```
`timescale 1ns / 1ps
module delay_demo1(q0_out, q1_out, q2_out);
    output [7:0]   q0_out, q1_out, q2_out;
    reg    [7:0]   q0_out, q1_out, q2_out;

    initial begin
        q0_out = 0;
        //循环体 1
        repeat(100) begin
            #5 q0_out = 1;   //延迟语句 1
            #5 q0_out = 2;   //延迟语句 2
        end
    end

    initial fork
        //循环体 2
        repeat(100) begin
            #5 q1_out = 3;   //延迟语句 3
            #5 q1_out = 4;   //延迟语句 4
        end

        //循环体 3
        repeat(100) begin
            #5 q2_out = 5;   //延迟语句 5
            #5 q2_out = 6;   //延迟语句 6
        end
```

```
    join

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-10 所示。总共有 6 条延迟控制语句。在仿真启动后，同时进入语句 1、语句 3 以及语句 5，都延迟 5 个仿真时间单位后，同时执行语句 2、语句 4 以及语句 6。这是因为在串行语句块“begin…end”中，语句是串行执行的，并行语句块“fork…join”却是并行执行的，因此 3 个循环体是同时并行执行的，而执行每个循环体都需要 10 个仿真时间单位。

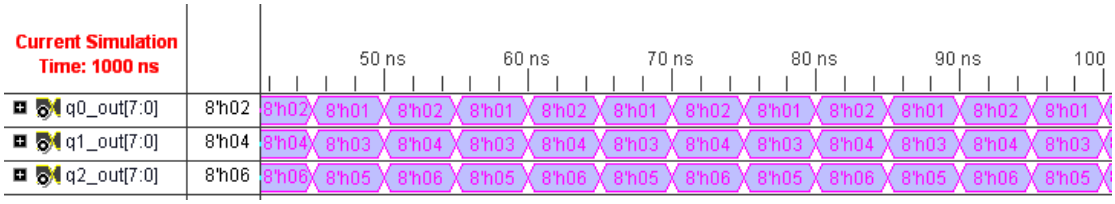


图 6-10　例 6-3 仿真结果示意图

2．#<延迟时间常量>;

在这种方式中，<延迟时间常量>后面没有出现任何行为语句，仿真进程遇到该语句后，也不执行任何操作，而是进入等到状态；等过了延迟时间后，再继续执行后续语句。由于在并行 fork…join 语句块和串行 begin…end 语句块进入仿真等待状态的影响是不同的，因此其在两类语句块中产生的作用也是不一样的，下面给出说明实例。

例 6-4：“#”语句的应用实例 2。

```
`timescale 1ns / 1ps
module delay_demo2(q0_out, q1_out);
    output   [7:0] q0_out, q1_out;
    reg      [7:0] q0_out, q1_out;

    initial begin
      q0_out = 0;
       #100 q0_out = 1;   //延迟语句 1
       #100;              //延迟语句 2
       #100 q0_out = 10; //延迟语句 3
       #300 q0_out = 20; //延迟语句 4
    end

    initial fork
      q1_out = 0;
       #100 q1_out = 1;   //延迟语句 5
       #100;              //延迟语句 6
       #200 q1_out = 10; //延迟语句 7
       #300 q1_out = 20; //延迟语句 8
    join

endmodule
```

上述程序总共有 8 条延迟控制语句，其中前 4 条在串行语句块执行，后 4 条在并行语句块中执行；延迟语句 2、延迟语句 6 为第二种用法。其在 ISE Simulator 中的仿真结果如图

6-11 所示，可以看出在串行块中，延迟语句 2 将下一条语句的执行延迟了指定的时间量。而在并行语句块中，语句 5、6、7、8 都在 0ns 时刻被执行，赋值操作分别在 100ns、100ns、200ns、300ns 处完成，语句 6 的操作不会对仿真结果产生任何影响，程序流控制在执行时间最长的语句 8 执行后结束。

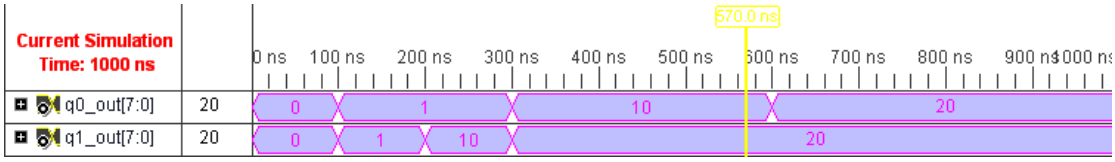


图 6-11　例 6-4 仿真结果示意图

3．#<延迟表达式> 行为语句;

在这种方式中，延迟时间是一个表达式或变量，不必将其局限于一个常量，极大地增加了仿真程序的可移植性。由于延迟时间为表达式或变量，因此有可能在其对应的值出现负值以及'z'或'x'。对于这种情况，Verilog HDL 语法规定，如果在延迟时间的变量或表达式中'z'或'x'比特，将其按照“0”来处理；如果代表延迟时间的变量或表达式的计算值为负值，则其实际的延时为零时延。下面给出一个应用实例。

例 6-5：“#”语句的应用实例 3。

```
`timescale 1ns / 1ps
module delay_demo3(q0_out, q1_out);
    output   [7:0] q0_out, q1_out;
    reg      [7:0] q0_out, q1_out;

    parameter delay_time = 100;
    initial begin
      q0_out = 0;
       #delay_time q0_out = 1;            //延迟语句 1
       #(delay_time/2);                   // 延迟语句 2
       #(delay_time*2) q0_out = 10;   //延迟语句 3
       #300 q0_out = 20;                 //延迟语句 4
    end

    initial begin
      q1_out = 0;
       #100;                                    //延迟语句 5
       #(delay_time-5'bxxxxx) q1_out = 1; //延迟语句 6
       #100;                                    //延迟语句 7
       #100 q1_out = 10;                    //延迟语句 8
       #50;                                     //延迟语句 9
       #(delay_time - 200) q1_out = 20;     //延迟语句 10
    end

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-12 所示。可以看出，在延迟控制语句的延迟表达式中出现 x、z 以及负值后，其延迟值全部按照 0 来对待。

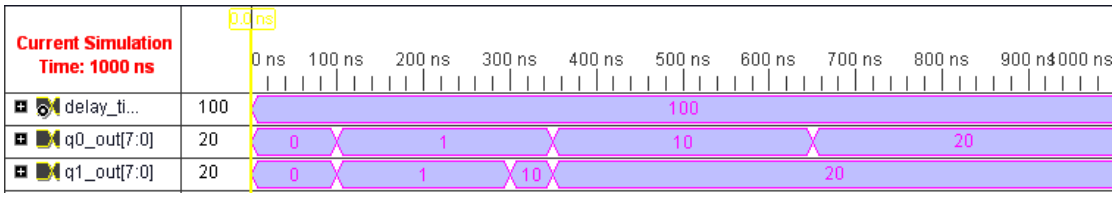


图 6-12　例 6-5 仿真结果示意图

6.4 常用的行为仿真描述语句

虽然所有的 Verilgo HDL 语句都可以在仿真代码中使用，但并不是每条语句都高效、已用。本节介绍一些在 Testbench 中经常使用的行为描述语句，基于这些语句，可以设计出高效、规范的测试代码。

6.4.1　循环语句

在 5.3 节中，已向读者说明：循环语句中的 forever 语句不可综合，只能用于行为仿真；也介绍了可综合的循环语句在 RTL 级硬件设计方面的不足。但在功能仿真代码中，所有的循环语句都具有非常重要的地位。

1．forever 语句

forever 循环语句连续执行过程语句。为跳出这样的循环，中止语句可以与过程语句共同使用。同时，在过程语句中必须使用某种形式的时序控制，否则 forever 循环将永远循环下去。forever 语句必须写在 initial 模块中，主要用于产生周期性波形。

forever 循环的语法为：

```
forever begin
    ……
end
```

例：forever 语句的应用实例。

```
initial
forever begin
    if(d) a = b + c;
    else a= 0;
end
```

需要说明的一点是，在很多情况下要避免使用 forever 语句，因为只有可控制和有限的事件才是高效的，否则会增加 PC 机 CPU 和内存的的资源消耗，从而降低仿真速度。但也有一个特例，就是时钟产生电路。这是因为时钟本身就是周期性的，但由于时钟只是单比特信号，因此不会对仿真速度造成太大影响。

2．利用循环语句完成遍历

for、while 语句常用于完成遍历测试。当设计代码包含了多个工作模式，那么就需要对各种模式都进行遍历测试，如果手动完成每种模式的测试，则将造成非常大的工作量。利用 for 循环，通过循环下标来传递各种模式的配置，不仅可以有效减少工作量，还能保证验证的完备性，不会漏掉任何一种模式。其典型的应用模版如下：

```
parameter    mode_num = 5;
initial begin
```

```
    //各种模式不同的参数配置部分
    for (i = 0; i < (mode_num - 1); i = i + 1) begin
        case (i)
            0 : begin
                    ...
                end
            1 : begin
                    ...
                end
          ...
        endcase
    end

    //各种模式共同的测试参数

end
```

由于仿真语句并不追求电路的可综合性，因此推荐在多分枝情况下使用 for 循环来简化代码的编写难度，并降低其出错概率。

3．利用

repeat 语句主要用于实现有次数控制的事件，其典型示例如下：

```
initial begin
    //初始化
    in_data = 0;
    wr = 0;

    //利用 repeat 语句将下面的代码执行 10 次
    repeat(10) begin
      wr = 1;
      in_data = in_data + 1;
      #10;
      wr = 0;
      #200;
    end
end
```

4．循环语句的异常处理

通常，循环语句都会有一个“正常”的出口，比如当循环次数达到了循环计数器所指定的次数或 while 表示式不再为真。然后，使用 disable 语句可以退出任何循环，能够终止任何 begin…end 块的执行，从紧接这个块的下一条语句继续执行。

disable 语句的典型示例如下：

```
begin:one_branch
    for(i = 0; i < n; i = i +1) begin:two_branch
        if (a == 0)
          disable one_branch;
        if (a == b)
```

```
            disable two_branch;
        end
    end
```

6.4.2 FORCE 和 RELEASE 语句

force/release 语句可以用来跨越进程对一个寄存器或一个电路网络的赋值。该结构一般用于强制特定的设计的行为。一旦一个强制值被释放，这个信号保持它的状态直到新的值被进程赋值。

force 语句可为寄存器类型和线网型变量强制赋值，当应用于寄存器时，寄存器当前值被 force 覆盖；当 release 语句应用于寄存器时寄存器当前值将保持不变，直到被重新赋值。当用于线网时，数值立即被 force 覆盖；当 release 语句应用于线网时，线网数值立即恢复到原来的驱动值。下面给出一个 force/release 语句的应用实例。

例 6-6：force/release 语句的应用实例。

```
`timescale 1ns / 1ps
module tb_force;
    reg     [7:0] q0_out;
    wire    [7:0] q1_out;

     initial begin
         q0_out = 0;
         #100;
         force q0_out = 0;
         #100;
         release q0_out;
     end

     always #10 q0_out = q0_out + 1;

     initial begin
         #100;
          force q1_out = 0;
          #100;
          release q1_out;
     end
    assign   q1_out =   127;

endmodule
```

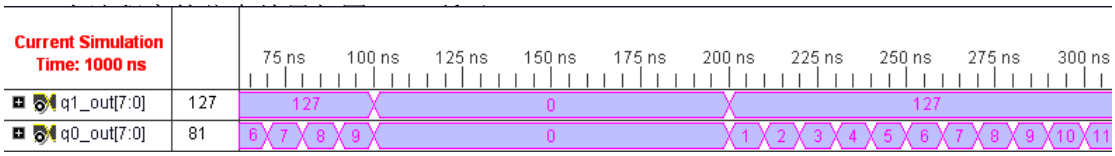


图 6-13　force/release 语句仿真结果示意图

6.4.3 WAIT 语句

wait 语句是一种不可综合的电平触发事件控制语句，有如下两种形式：

（1）wait (条件表达式)　语句/语句块;

（2）wait (条件表达式) ;

对于第一种形式，语句块可以是串行块（begin...end）或并行块（fork…join）。当条件表达式为“真（逻辑 1）”时，语句块立即得到执行，否则语句块要等到条件表达式为真再开始执行。例如：

```
wait(rst == 0) begin
    a = b
end
```

所实现的功能就是等待复位信号 rst 变低后，将信号 b 的值赋给 a。如果在仿真进程中 rst 信号不为低，那么就暂停进程并等待。

在第二种形式中，没有包含执行的语句块。当仿真执行到 wait 语句，如果条件表达式为真，那么立即结束该 wait 语句的执行，仿真进程继续往下进行；如果 wait 条件表达式不为真，则仿真进程进入等待状态，直到条件表达式为真。下面给出一个 wait 语句的开发实例。

例 6-7：wait 语句的实例。

```
module tb_wait;
    reg   [7:0] q0_out;
    reg         flag;

    //initial 初始化语句块 1
    initial begin
      flag = 0;
       #100 flag = 1;
       #100 flag = 0;
    end

    //initial 初始化语句块 2
    initial begin
       q0_out = 0;
       wait( flag == 1) begin    //wait 语句
            q0_out = 100;
             # 100;
       end
       q0_out = 255;
    end

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-14 所示，可以看出 initial 初始化语句块 2 中的 wait 语句直到 100ns 后，等待到 flag 信号为 1 后，才将 q0_out 的数值赋为 100，在此之前一直阻塞 initial 初始化语句块 2 的执行。

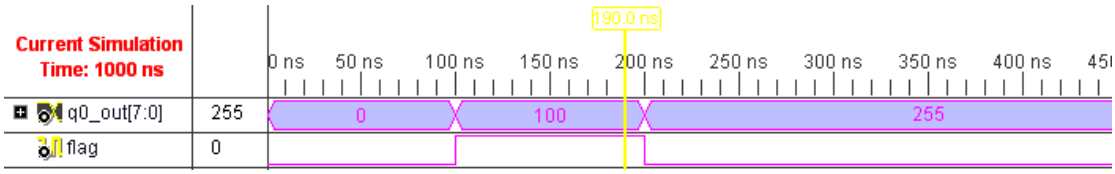


图 6-14　wait 语句开发实例

6.4.4　事件控制语句

在仿真进程中也存在电平触发和信号跳变沿触发两大类，描述方法和 5.1 节的方法相同。此外，在仿真程序中，还可通过“@(事件表达式)”事件来完成单次事件触发。例如下面的实例。

例 6-8：利用“@(事件表达式)”事件来完成单次事件触发。

```
`timescale 1ns / 1ps
module tb_event;
    reg     [7:0]   cnt0, cnt1;
     reg             clk;

     initial begin
         forever begin
                 clk = 0;
                 #5;
                 clk = 1;
                 #5;
          end
     end

     //捕获信号上升沿
     initial begin
         cnt0 = 0;
         forever begin
              @(posedge clk)     //捕获脉冲沿事件
                  cnt0 = cnt0 + 1;
          end
     end

     //捕获信号电平
     initial begin
         cnt1 = 0;
         forever begin
              @(clk) begin       //捕获电平事件
                  if(clk == 1)
                      cnt1 = cnt1 + 1;
                end
          end
     end
```

endmodule

如前所述，用于综合的语句完全可以用于仿真应用中，因此仿真代码中信号跳变沿的捕获和电平事件捕获方法和面向综合的设计完全一致。上述程序在 ISE Simulator 中的仿真结果如图 6-15 所示，可以看出其正确完成了事件控制。

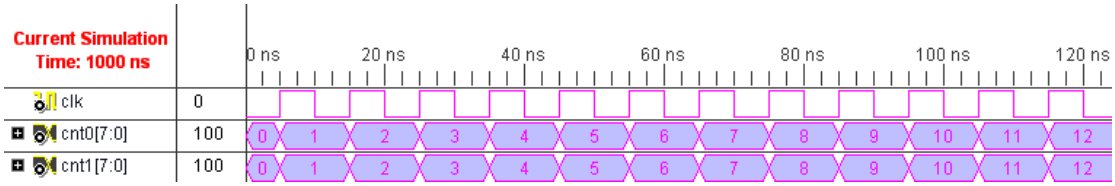


图 6-15　例 6-8 的仿真结果示意图

6.4.5　TASK 和 FUNCTION 语句

task 语句和 function 语句在仿真程序中发挥了最大优势，可以将固定操作封装起来，配合延时控制语句，可精确模拟大多数常用的功能模块，具备良好的可重用性。下面给出一个 task 语句在 Verilog HDL 仿真代码中的演示实例。

例 6-9：基于 task 的 3 次方模块演示实例。

```
`timescale 1ns / 1ps
module tb_tri;
    parameter    bsize = 8;
    parameter    clk_period = 2;
    parameter    cac_delay    = 6;
    reg    [(bsize -1):0]         din;
    reg    [(3*bsize -1) :0]    dout;

     //定义完成 3 次方运算的 task
     task tri_demo;
         input      [(bsize -1):0]         din;
         output    [(3*bsize -1) :0]      dout;
          # cac_delay dout =    din*din*din;
     endtask

     //在串行语句块中调用完成 3 次方运算的 task
     initial begin
         din = 0;
     end

     always # clk_period begin
             din = din + 10;
             //任务调用语句
             tri_demo(din, dout);
     end

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-16 所示，正确地计算出输入数据的 3 次方，达到了设计要求。

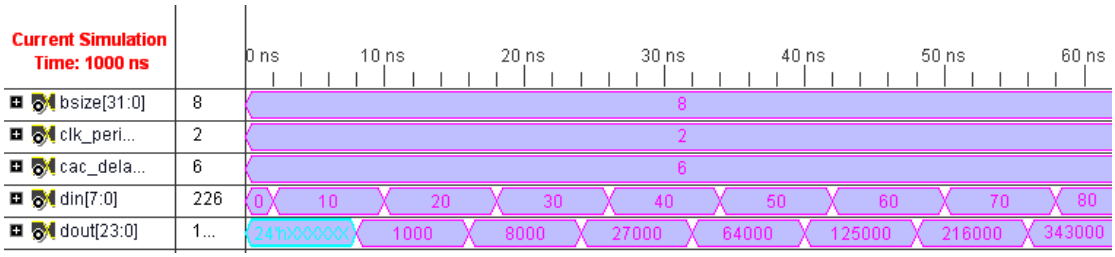


图 6-16　task 语句仿真结果示意图

实质上，在仿真程序中，task 和 function 就完全是 C 语言中的内联函数，主要用于简化代码结构。

6.4.6　串行激励与并行激励语句

与可综合语句一样，begin…end 语句用于启动串行激励，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。fork…join 的语法格式如下：

```
fork : <>
    时间控制 1    行为语句 1;
    …
    时间控制 n    行为语句 n;
join
```

其中，fork…join 块内被赋值的语句必须为寄存器型变量。其主要特点如下：

（1）并行块内语句是同时开始执行的，当仿真进程进入到并行块之后，块内各条语句同时、独立地开始执行。

（2）并行块语句中指定的延时控制都是相对于程序流程进入并行块的时刻的延时。

（3）当并行块所有语句都执行完后，仿真程序进程才跳出并行块。整个并行块的执行时间等于执行时间最长的那条语句所执行的时间。

（4）并行块可以和串行块混合嵌套使用。内层语句块可以看成外层语句块中的一条普通语句，内层语句块在什么时候得到执行由外层语句块的规则决定；而在内层语句块开始执行后，其内部各条语句的执行要遵守内层语句块的规则。

例如，在仿真进程开始 100 个时间单位后，希望同时启动发送和接收任务，可以采用并行语句块“fork..join”，这样可以避免在发送完毕后再启动接收任务，造成数据丢失的现象。

```
initial begin
    #100;
    fork
        send_task;
        receive_task;
    join
    …
end
```

其中，fork…join 块被包含在 begin..end 块之内，其等效于单条赋值语句，在“#100”语句之后开始执行，内部的两个 task 是并行执行的，等两个任务都执行完毕后，跳出 fork…join 块，顺序执行后续语句。

上述例子将并行块包含在串行块中，同样，也可以将串行块包含在并行块中，其执行分析过程与上述说明类似。

6.5 用户自定义元件

Verilog HDL 语言提供了一种扩展基元的方法，允许用户自己定义元件（User Defined Primitives，UDP）。通过 UDP，可以把一块组合逻辑电路或时序逻辑电路封装在一个 UDP 内，并把这个 UDP 作为一个基本门元件来使用。读者需要注意的是，UDP 是不能综合的，只能用于仿真。

6.5.1 UDP 的定义与调用

1．UDP 的定义

在定义语法上，UDP 定义和模块定义类似的，但由于 UDP 和模块属于同级设计，所以 UDP 定义不能出现在模块之内。UDP 定义可以单独出现在一个 Verilog 文件中或与模块定义同时处于某个文件中。

模块定义使用一对关键词“primitive-endprimitive”封装起来的一段代码，这段代码定义来该 UDP 的功能。这种功能的定义是通过表来实现的，即在这段代码中有一段处于关键词“table-endtable”之间的表，用户可以通过设置这个表来规定 UDP 的功能。

UDP 的定义格式如下：

```
primitive UDP_name(OutputName, List_of_inputs)
    Output_declaration
    List_of_input_declarations
    [Reg_declaration]
    [Initial_statement]
    table
        List_of_table_entries
    endtable
endprimitive
```

和 Verilog HDL 中的模块（module）相比，UDP 具备以下特点：

（1）UDP 的输出端口只能有一个，且必须位于端口列表的第一项。只有输出端口能定义为 reg 类型。

（2）UDP 的输入端口可有多个，一般时序电路 UDP 的输入端口最多至 9 个，组合电路 UDP 的输入端口可多至 10 个。

（3）所有端口变量的位宽必须是 1 比特。

（4）在 table 表项中，只能出现 0、1、x 三种状态，z 将被认为 x 状态。

根据 UDP 包含的基本逻辑功能，可以将 UDP 分为组合电路 UDP 和时序电路 UDP，这两类 UDP 的差别主要体现在 table 表项的描述上。

2．UDP 的调用

UDP 的调用和 Verilog HDL 中模块的调用方法相似，通过位置映射，其语法格式如下所列：

```
UDP 名  例化名  (连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名,…);
```

位置映射法严格按照 UDP 中定义的端口顺序来连接，第一个连接端口为输出端口。

6.5.2 UDP 应用实例

1．组合电路 UDP 元件

组合逻辑电路的功能列表类似真值表，就是规定了不同的输入值和对应的输出值，表中

每一行的形式是“Output, Input1, Input2, …”，排列顺序和端口列表中的顺序相同。如果某个输入组合没有定义的输出，那么就把这种情况的输出置为 x。下面给出一个单比特乘法器的 UDP 开发实例。

例 6-10：单比特乘法器的 UDP 开发实例。

```
primitive MUX2x1 (Z, Hab, Bay, Sel) ;
   output Z;
   input Hab,Bay, Sel;
   table
   // Hab Bay Sel : Z  注：本行仅作为注释。
   0 ? 1 : 0 ;
   1 ? 1 : 1 ;
   ? 0 0 : 0 ;
   ? 1 0 : 1 ;
   0 0 x : 0 ;
   1 1 x : 1 ;
   endtable
endprimitive
```

其中，字符?代表不必关心相应变量的具体值，即它可以是 0、1 或 x。此外，Verilog HDL 语言标准规定，如果 UDP 输入端口出现的 z 值按照 x 处理。表 6-1 列出了 UDP 原语中的可用选项。可以看出，其直接通过真值表来描述电路功能，和 FPGA 的工作原理是一致的，但遗憾的是，UDP 并不能用于可综合设计。

表 6-1　　所有能够用于 UDP 原语中表项的可能值

符号	意义	符号	意义
0	逻辑 0	(AB)	由 A 变到 B
1	逻辑 1	*	与(??)相同
x	未知的值	r	上跳变沿，与(01)相同
?	0、1 或 x 中的任一个	f	下跳变沿，与(10)相同
b	0 或 1 中任选一个	p	(01)、(0x)和(x1)的任一种
—	输出保持	n	(10)、(1x)和(x0)的任一种

2．时序电路 UPD 元件

UDP 除了可以描述组合电路外，还可以描述具有电平触发和边沿触发特性的时序电路。时序电路拥有内部状态状态序列，其内部状态必须用寄存器变量进行建模，该寄存器的值就是时序电路的当前状态，它的下一个状态是由放在基元功能列表中的状态转换表决定的，而且寄存器的下一个状态就是这个时序电路 UDP 的输出值。所以，时序电路 UDP 由两部分组成——状态寄存器和状态列表。定义时序 UDP 的工作也分为两部分——初始化状态寄存器和描述状态列表。

在时序电路的 UDP 描述中，[01, 0x, x1]就代表着信号的上升沿。下面给出一个上升沿 D 触发器的 UDP 开发实例。

例 6-11：通过 Verilog HDL 语言给出 D 触发器的 UDP 描述，并在模块中调用 UDP 组件，给出仿真结果

```
primitive D_Edge_FF(Q, Clk, Data) ;
      output Q ;
      reg Q ;
      input Data, Clk;
```

```
    initial Q = 0;
    table
      // Clk Data Q (State)    Q(next )
      (01) 0 : ? : 0 ;
      (01) 1 : ? : 1 ;
      (0x) 1 : 1 : 1 ;
      (0x) 0 : 0 : 0 ;
      //  忽略时钟负边沿：
      (?0) ? : ? : - ;
      //  忽略在稳定时钟上的数据变化:
      ? (??): ? : - ;
    endtable

endprimitive
```

表项（01)表示从 0 转换到 1，表项（0x)表示从 0 转换到 x，表项（?0）表示从任意值（0 、 1 或 x)转换到 0，表项(??)表示任意转换。对任意未定义的转换，输出缺省为 x。假定 D_Edge_FF 为 UDP 定义，它现在就能够象基本门一样在模块中使用，如下面的 4 位寄存器所示。下面给出 D_Edge_FF 用户自定义元件的调用实例，用来实现一个 4 比特数据的寄存。

```
module Reg4 (Clk, Din, Dout) ;
    input Clk ;
    input [0:3] Din;
    output [0:3] Dout;

    //例化调用 UDP
    D_Edge_FF DLAB0 (Dout[0],Clk, Din[0]),
                DLAB1 (Dout[1],Clk, Din[1]),
                DLAB2 (Dout[2],Clk, Din[2]),
                DLAB3 (Dout[3],Clk, Din[3]);

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-17 所示，可以看出，在 CLk 上升沿将 Din 数据加载到 Dout 中。

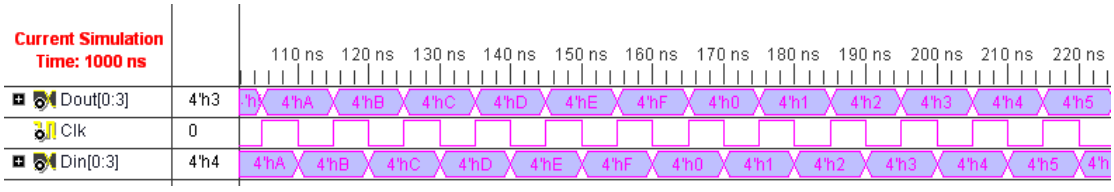


图 6-17　UDP 调用模块语句仿真结果示意图

3．混合电路 UPD 元件

在同一个表中能够混合电平触发和边沿触发项。在这种情况下，边沿变化在电平触发之前处理，即电平触发项覆盖边沿触发项。下面给出一段带异步清空的 D 触发器的 UDP 描述。

例 6-12：利用 Verilog HDL 语言完成异步清零 D 触发器的 UDP 描述。

```
primitive D_Async_FF (Q, Clk, Clr, Data) ;
    output Q;
    reg Q;
    input Clr, Data, Clk;

    //定义混合 UDP 元件
    table
    // Clk Clr Data ( SQtate) Q( next )
    (01) 0 0 : ? : 0 ;
    (01) 0 1 : ? : 1 ;
    (0x) 0 1 : 1 : 1 ;
    (0x) 0 0 : 0 : 0 ;
    //  忽略时钟负边沿：
    (?0) 0 ? : ? : - ;
    (??) 1 ? : ? : 0 ;
    ?   1 ? : ? : 0;
    endtable

endprimitive
```

上述的代码的功能和例 6-11 比较类似，这里就不再详细介绍。

6.6 仿真激励的产生

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，因为其最容易出问题。目前，主要有三种产生激励的方法：

（1）直接编辑测试激励波形。

（2）用 Verilog HDL 测试代码的时序控制功能，产生测试激励。

（3）利用 Verilog HDL 语言的读文件功能，从文本文件中读取数据（该数据可以通过 C/C++、MATLAB 等软件语言生成）。

其中，方法（1）和 EDA 工具有关，将在 6.8.1 节进行介绍；方法（3）涉及到系统任务的调用，第 7 章会有详细介绍；本节主要介绍第（2）中方法。

6.6.1 变量初始化

在 Verilog HDL 语言中，有两种方法可以初始化变量：一种是利用初始化变量；另外一种就是在定义变量时直接赋值初始化。这两种初始化任务是不可综合的，在硬件平台中没有任何意义，但对于仿真过程却是必须要掌握的。

1．变量初始化的必要性

由于 Verilog HDL 语言规定了 1、0、x 以及 z 这 4 类逻辑数值，对于 Testbench 中的变量，如果不进行初始化，会按照“x”来对待。这样，基于未初始化信号的累加以及各类判断全部以“x”来完成，造成仿真错误。下面给出一个计数器设计由于未初始化而使得仿真失败的实例。

例 6-13；通过计数器演示由于未初始化而造成的仿真失败。

```
module counter_demo(
    clk, cnt
    );
```

```
    input              clk;
    output   [3:0]   cnt;

    reg        [3:0] temp ;
    always @(posedge clk) begin
        temp <= temp + 1;
    end

    assign cnt = temp;
endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 6-18 所示。可以看出，计数器输出全部为 x，但上述代码在硬件中可以正确实现计数器，只是验证程序不能从功能上验证其正确性。这是因为寄存器变量 temp 没有经过初始化，其数值为不定态 x，“temp <= temp + 1”操作结果也是不定态，从而无法得到正确的仿真结果。

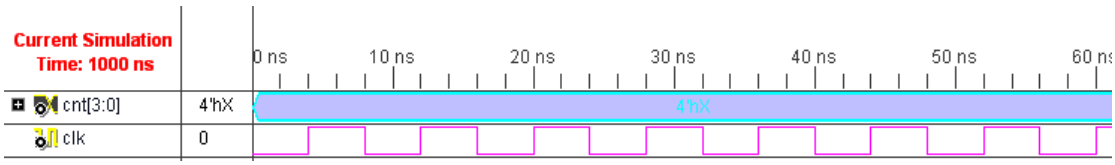


图 6-18　例 6-13 仿真结果示意图

2．变量初始化的方法

通过上例可以看出，要想通过验证代码完成设计的功能测试，必须要完成变量的初始化。初始化工作既可以在 Testbench 中完成，也可以在面向综合的设计代码中完成。对于后者，所有的初始化在综合时会被忽略，不会影响代码的综合结果。因此基本原则为：可综合代码中完成内部变量的初始化，Testbench 中完成可综合代码所需的各类接口信号的初始化。

初始化的方法有两种，一种是通过 initial 语句块初始化，另外一种是在定义时直接初始化，下面对其分别介绍。

（1）initial 初始化

在大多数情况下，Testbench 中变量初始化的工作通过 initial 过程块来完成，可以产生丰富的仿真激励；此外，也可用于可综合代码中。

initial 语句只执行一次，即在设计被开始模拟执行时开始（0 时刻），专门用于对输入信号进行初始化和产生特定的信号波形。一个 Testbench 可以包含多个 initial 语句块，所有的 initial block 都同时执行。需要注意的是：initial 语句中的变量必须为 reg 类型的。

当 initial 语句块中有多条语句时，需要用 begin…end 或者 fork…join 语句将其括起来。begin..end 中的语句为串行执行，而 fork…join 为并行执行。由于 fork..join 语句的控制难度较大，因此不推荐使用。建议读者尽量使用 begin…end 语句，如果信号较多，且处理冲突，可以使用多个 initial 语句块。本章会给出很多 initial 块在 Testbench 中的开发实例，下面先给出一个 initial 语句块在可综合代码中的应用。

例 6-14：利用 initial 语句完成例 6-11 中代码的初始化。

```
module counter_demo2(
clk, cnt
);
  input              clk;
  output   [3:0]   cnt;
```

```
    reg         [3:0] temp ;

    initial begin
        temp = 0;
    end

    always @(posedge clk) begin
        temp <= temp + 1;
    end

    assign cnt = temp;

endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 6-19 所示。可以看出，虽然例 6-12 中的代码添加了

```
initial begin
    temp = 0;
end
```

语句段，但其并不会改变程序的逻辑结构，没有添加硬件结构上的初始化电路，仍然是一个标准的计数器。当然，加上上述语句后，在代码仿真时，才能获得正确的结果。

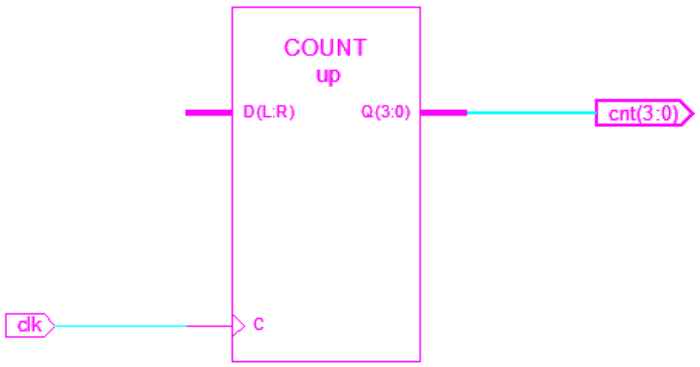


图 6-19　例 6-12 综合结果示意图

上述程序在 ISE Simulator 中的仿者结果如图 6-20 所示，计数器的初始值为 0，表明 initial 语句达到了预期目标。

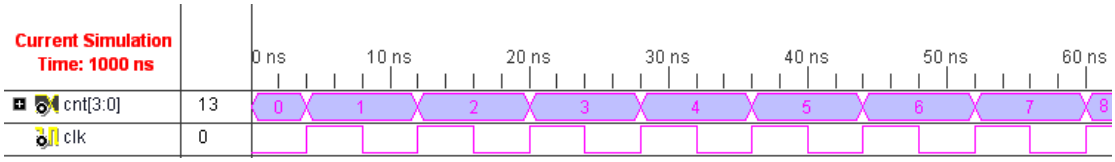


图 6-20　例 6-14 仿真结果示意图

（2）定义变量时初始化

在定义变量时初始化的语法非常简单，直接用“=”在变量右端赋初值即可，如：

```
reg [7:0] cnt = 8'b00000000;
```

就将 8 比特的寄存器变量 cnt 初始化为全 0 比特。

和 initial 语句比较起来，定义时初始化的方法功能比较单一，但使用方便。常用于可综合代码书写中，其目的就是保证设计的硬件实现和软件仿真达到一致。例如要完成例 6-12 中的变量 temp 初始化，将语句“reg [3:0] temp;”修改为下列语句：

```
reg       [3:0] temp = 0;
```

3．硬件系统中的初始化

通过上面的介绍，读者肯定会有疑惑，在硬件平台上，存在变量初始化吗？如果存在初始化工作是怎样完成的呢？答案是肯定的，硬件平台中当然存在初始化，但该操作必须通过可综合语句来实现，不能通过上述两种方法来完成。

（1）EDA 工具设置

在硬件平台中，当系统上电工作后，信号电平不是“1”就是“0”，不会存在“x”，因此对于例 6-12 所示的计数器，在没有初始化处理（仿真语句的初始化没有实际意义）的情况下，依然会正常工作。但在不同的平台上，其初始状态也是不确定的，存在两种可能的初始状态“0000”或者“1111”。在 Xilinx 公司的 CPLD/FPGA 平台上，默认为“0”。

（2）通过外部复位信号

虽然，硬件器件具有本身的初始化电平，但不具备通用特征，且完全依赖默认电平，会使得程序不可控。例如例 6-12 中的计数寄存器 temp，其当前数值就是设计人员所无法控制的。通过复位信号会达到上述的双重目的，即完成寄存器初始化，又使其可控。下面给出一个应用实例。

例 6-15：对例 6-14 添加复位信号，并给出仿真结果。

```
module counter_demo(
      clk, reset, cnt
      );
       input              clk, reset;
       output   [3:0]   cnt;

       reg        [3:0] temp ;
       always @(posedge clk) begin
           //通过复位信号来初始化计数器
           if (!reset)
              temp <= 3'b000;
           else
              temp <= temp + 1;
       end

       assign cnt = temp;
endmodule
```

这种设计模式从硬件上确保设计和仿真一致，不论在什么平台上，都可以达到预期效果。程序在 ISE 中综合后的 RTL 结构图如图 6-21 所示，可以看出其添加了复位信号。

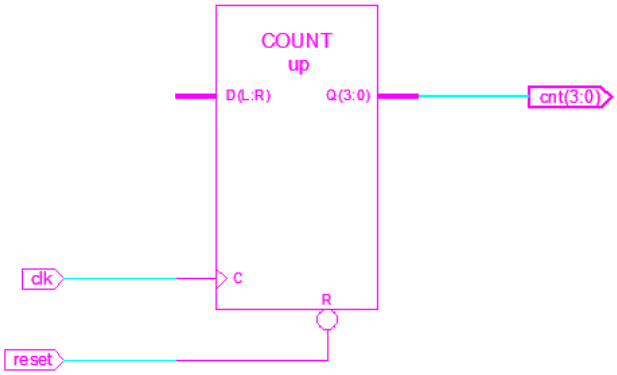


图 6-21　例 6-15 综合结果示意图

上述程序在 ISE Simulator 中的仿真结果如图 6-22 所示。可以看出，经过 reset 复位后，计数器从 0 开始，正常工作。

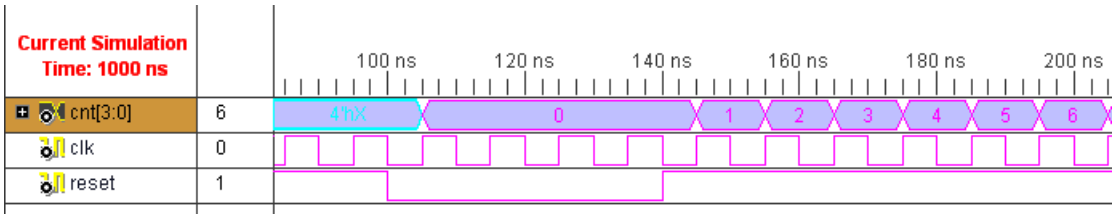


图 6-22　例 6-15 综合结果示意图

通过上例可以发现，通过复位信号，不管有没有变量初始化语句，Testbench 能够完全控制被测试代码，是一种优秀的代码书写风格。

6.6.2　时钟信号的产生

时钟是时序电路设计最关键的参数，而时序电路又获得了广泛应用，因此本节专门介绍如何产生仿真验证过程所需要的各类时钟信号。

1．普通时钟信号

所谓的普通时钟信号就指的是占空比为 50%的时钟信号，也是最常用的时钟信号，其波形如图 6-23 所示。

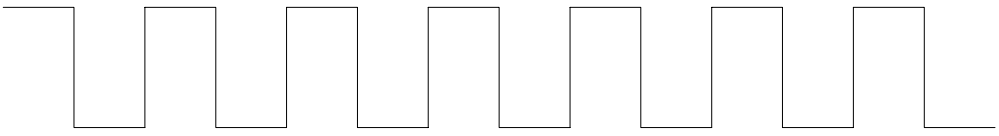


图 6-23　占空比为 50%的时钟信号

普通时钟信号可通过 initial 语句和 always 语句产生，其方法如下。

（1）基于 initial 语句的方法

```
parameter clk_period = 10;
reg clk;
initial begin
    clk = 0;
    forever
      # (clk_period/2) clk = ~clk;
end
```

（2）基于 always 语句的方法

```
parameter clk_period = 10;
reg clk;
initial
    clk = 0;
always #   (clk_period/2) clk = ~clk;
```

initial 语句用于初始化 clk 信号，否则就会出现对未知信号取反的情况，因而造成 clk 信号在整个仿真阶段都为未知状态。

2．自定义占空比的时钟信号

自定义占空比信号通过 always 模块可以快速实现，下面给出一个占空比为 20%的时钟信号代码：

```
parameter High_time = 5,
          Low_time = 20;
//占空比为 High_time/( High_time+ Low_time)
reg     clk;
always begin
    clk = 1;
    #High_time;
    clk = 0;
    #Low_time;
end
```

这里由于直接对 clk 信号赋值，所以不需要 initial 语句初始化 clk 信号。当然，这种方法可以用于产生普通时钟信号，只是代码行数较多而已。

3．相位偏移的时钟信号

相位偏移是两个时钟信号之间的相对概念，如图 6-24 所示，其中 clk_a 为参考信号，clk_b 为偏移信号。

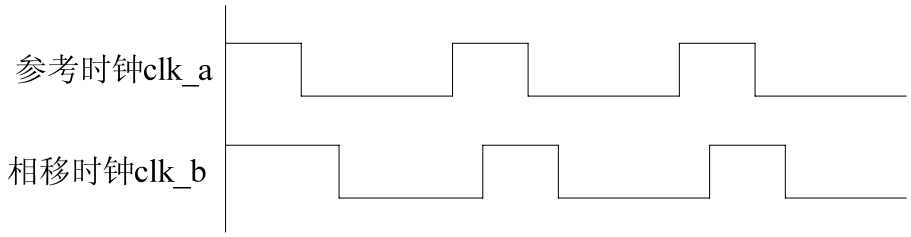


图 6-24　相位偏移时钟示意图

产生相移时钟的代码为：

```
parameter High_time = 5, Low_time = 5, pshift_time = 2;
reg     clk_a;
wire    clk_b;
always    begin
    clk_a = 1;
    # High_time;
    Clk_b = 0;
    # Low_time;
end
assign    # pshift_time clk_b = clk_a;
```

首先通过一个 always 模块产生参考时钟 clk_a，然后通过延迟赋值得到 clk_b 信号，其偏移的相位可通过 360*pshift_time%（High_time+Low_time）来计算，其中%为取模运算。上述代码的相位偏移为 72 度。

4．固定数目的时钟信号

上述语句产生的时钟信号都是无限个周期的，也可以通过 repeat 语句来产生固定个数的时钟脉冲，其代码如下：

```
parameter clk_cnt = 5, clk_period = 2;
reg     clk;
initial begin
    clk = 0;
    repeat (clk_cnt)
        # clk_period/2 clk = ~clk;
end
```

上述代码的产生了 5 个周期的时钟。

6.6.3 复位信号的产生

复位信号不是周期信号，通常通过 initial 语句产生的值序列来描述。下面分别介绍同步和异步复位信号。

1．异步复位信号

异步复位信号的实现代码如下：

```
parameter rst_repiod = 100;
reg rst_n;
initial begin
  rst_n = 0;
  # rst_repiod;
  rst_n = 1;
end
```

上述代码将产生低有效的复位信号 rst_n，其复位时间为 100 个仿真时间单位。

2．同步复位信号

同步复位信号的实现代码如下：

```
parameter rst_repiod = 100;
reg rst_n;
initial begin
   rst_n = 1;
   @( posedge clk);
      rst_n = 0;
   # rst_repiod;
   @( posedge clk);
      rst_n = 1;
end
```

上述代码首先将复位信号 rst_n 初始化为 1，然后等待时钟信号 clk 的上升沿，将 rst_n 拉低，进入有效复位状态；然后经过 100 个仿真周期，等待下一个上升沿到来后，将复位信号置为 1。在仿真代码中，是不存在逻辑延迟的，因此在上升沿对 rst_n 的赋值，能在同一个沿送到测试代码逻辑中。

在需要复位时间为时钟周期的整数倍时，可以将 rst_repiod 修改为时钟周期的 3 倍来实现，也可以通过下面的代码来完成。

```
parameter rst_num = 5;
initial begin
   rst_n = 1;
   @(posedge clk);
   rst_n = 0;
   repeat(rst_num) @(posedge clk);
   rst_n = 1;
end
```

上述代码在 clk 的第一个上升沿开始复位，然后经过 5 个时钟上升沿后，在第 5 个时钟上升沿撤销复位信号，进入有效工作状态。

6.6.4 数据信号的产生

如前所述，数据信号既可以通过 Verilog HDL 语言的时序控制功能（#、initial、always 语句）来产生各类验证数据，也可以通过系统任务来读取计算机上已存在的数据文件。本小节主要介绍第一种方法，读取文件的方式将在 7.1.2 节进行介绍。

数据信号的产生主要有两种形式：其一就是初始化和产生都在单个 initial 块中产生；其二就是初始化在 initial 语句中完成，而产生却在 always 语句块中完成。前者适合不规则数据序列，并且长度较短；后者适合具有一定规律的数据序列，长度不限。下面分别通过实例进行说明。

例 6-16：产生位宽为 4 的质数序列{1、2、3、5、7、11、13}，并且重复 2 次，其中样值间隔为 4 个仿真时间单位。

由于该序列无明显规律，因此利用 initial 语句最为合适，代码如下：

```
`timescale 1ns / 1ps
module tb_xulie1;
      reg          [3:0]          q_out;
      parameter sample_period = 4;
      parameter queue_num     = 2;

      initial begin
          q_out     = 0;
           repeat(queue_num) begin
             # sample_period q_out = 1;
             # sample_period q_out = 2;
             # sample_period q_out = 3;
             # sample_period q_out = 5;
             # sample_period q_out = 7;
             # sample_period q_out = 11;
             # sample_period q_out = 13;
         end
      end

endmodule
```

上述程序的仿真结果如图 6-25 所示。

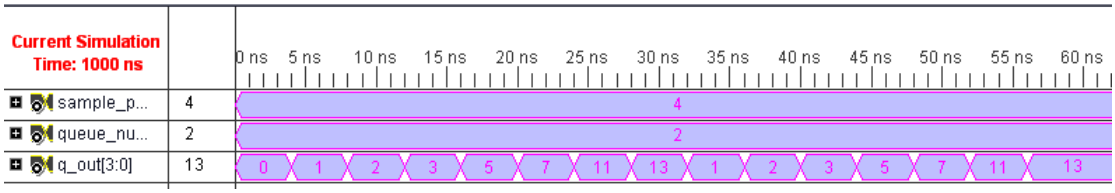


图 6-25　质数序列仿真结果示意图

例 6-17：产生位宽为 4 的偶数序列，并重复多次。

由于该序列规律明显，因此利用 always 语句最为方便，代码如下：

```
module tb_xulie2;
reg          [3:0]          q_out;
parameter sample_period = 4;

initial
    q_out   = 0;

always # sample_period
    q_out   = q_out + 2;

endmodule
```

上述程序的仿真结果如图 6-26 所示，达到了设计要求。

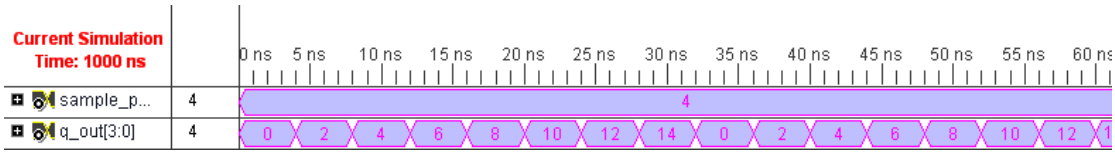


图 6-26　偶数序列仿真结果示意图

6.6.5　典型测试平台实例

下面给出测试模块的代码编写风格。

```
timescale 1ns / 1ps
module cmult_v;
    // 输入信号向量
    reg clk;
    reg [15:0] ar, ai, br ,bi ;

    //输出信号向量
    wire [31:0] qr, qi;

    // 实例化待测的模块单元  (UUT)
    cmultip uut (
        .clk(clk), .ar(ar), .ai(ai),  .qr(qr), .br(br), .bi(bi),    .qi(qi)
    );
```

```
    initial begin
        // 初始化输入向量
        clk = 0;   ar = 0;   ai = 0;   br = 0; bi = 0;

        #100;   //等待 100ns 后，全局 reset 信号有效
        ar = 20;   ai = 10;   br = 10; bi = 10;
    end

    always #5 clk = ~clk;
    always # 10 ar = ar + 1;
    always # 10 ai = ai + 1;
    always # 10 br = br + 1;
    always # 10 bi = bi + 1;

endmodule
```

在测试模块中，测试向量的产生是测试问题中的一个重要部分，只有测试向量产生的完备，分析测试结果才有意义。如果有方法产生出期望的结果，可以用 Verilog 或者其他工具自动地比较期望值和实际值。如果没有简易的方法产生期望的结果，那么明智地选择测试向量，可以简化仿真的结果。当然，测试向量的产生是个在繁琐中追求特殊的情况。所以需要根据实际情况来选择测试向量。

6.6.6 关于仿真效率的说明

和 C/C++等软件语言相比，Verilog HDL 行为级仿真代码的执行时间比较长，其主要原因就是因为要通过串行软件代码完成并行语义的转化。随着代码的增加，会使得仿真验证过程非常漫长，从而导致仿真效率的降低，成为整体设计的瓶颈。即便如此，不同的设计代码其仿真执行效率也是不同的，下面列出几个注意点，可以帮助设计人员提高 Verilog HDL 代码的仿真代码执行时间[1]。

1．减少层次结构

仿真代码的层次越少，执行时间就越短。这主要是由于参数在模块端口之间传递需要消耗仿真器的执行时间。

2．减少门级代码的使用

由于门级建模属于结构级建模，自身参数建模已经比较复杂了，还需要通过模块调用的方式来实现，因此建议仿真代码尽量使用行为级语句，建模层次越抽象，执行时间就越短。引申一点，在行为级代码中，尽量使用面向仿真的语句。例如，延迟两个仿真时间单位，最好通过“#2”来实现，而不是通过深度为 2 的移位寄存器来实现。

3．仿真精度越高，效率越低

例如包含`timescale 1ns / 1ps 定义的代码执行时间就比包含`timescale 1ns / 1ns 定义的代码执行时间长。

4．进程越少，效率越高

代码中的语句块越少仿真越快，例如将相同的逻辑功能分布在两个 always 语句块中，其仿真执行时间就比利用一个 always 语句来实现的代码短。这是因为仿真器在不同进程之间进行切换也需要时间。

5．减少仿真器的输出显示

Verilog HDL 语言包含一些系统任务，可以在仿真器的控制台显示窗口输出一些提示信

息。虽然其对于软件调试是非常有用的，但会降低仿真器的执行效率。因此，在代码中这一类系统任务不能随意使用。本书第 7.1 节会详细介绍各类系统任务。

本质上来讲，减少代码执行时间并不一定会提高代码的验证效率，因此上述建议需要和仿真代码的可读性、可维护性以及验证覆盖率等多方面结合起来考虑。

6.8 Xilinx 仿真工具 ISE Simulator

ISE Simulator 是 ISE 集成开发环境中的一个仿真组件，是完整的 PC 硬件描述语言（HDL）仿真和调试工具，主要有两个版本。ISE Simulator Lite 随所有 ISE 版本免费提供，为 HDL 源代码不超过 1 万行的 CPLD 和低密度 FPGA 设计提供了一个理想的解决方案。ISE Simulator 完全版支持所有设计密度。

6.8.1 基于波形测试法的仿真

在 ISE 中创建 testbench 波形，可通过 HDL Bencher 修改，再将其和仿真器连接起来，然后验证设计功能是否正确。首先在工程管理区将“Sources for”设置为 Behavioral Simulation，然后在任意位置单击鼠标右键，在弹出的菜单中选择“New Source”命令，然后选中“Test Bench WaveForm”类型，输入文件名为“test_bench”，点击“Next”进入下一页。这时，工程中所有 Verilog Module 的名称都会显示出来，设计人员需要选择要进行测试的模块，这里选择“test”，如图 6-27 所示。

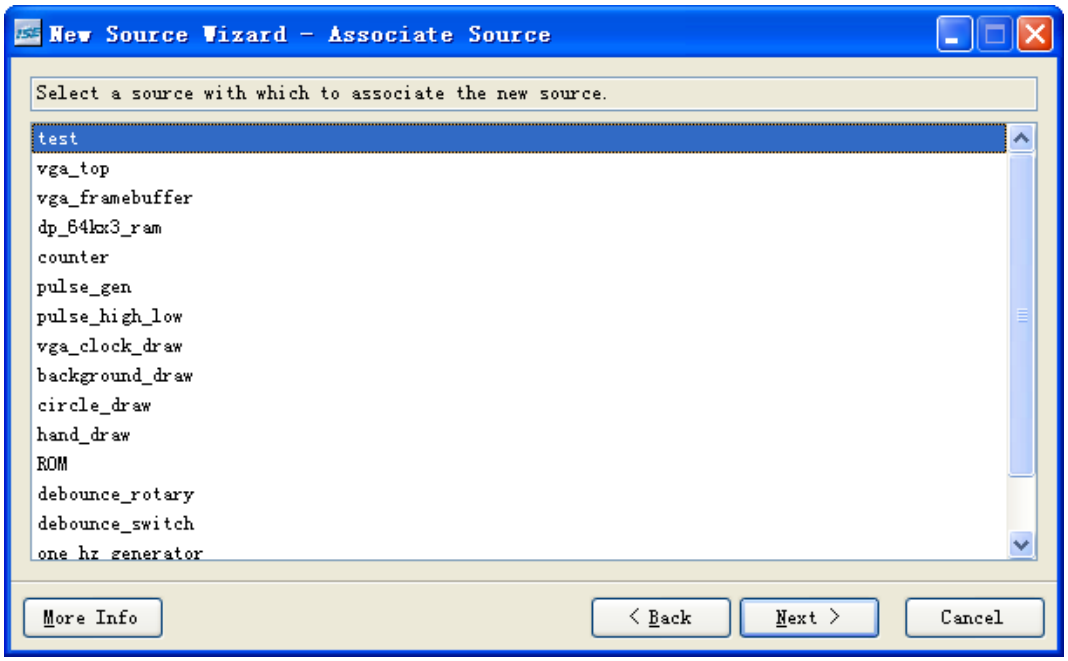


图 6-27　选择待测模块对话框

用鼠标选中 test，点击“Next”后进入下一页，直接点击“Finish”按键。此时 HDL Bencher 程序自动启动，等待用户输入所需的时序要求，如图 6-28 所示。

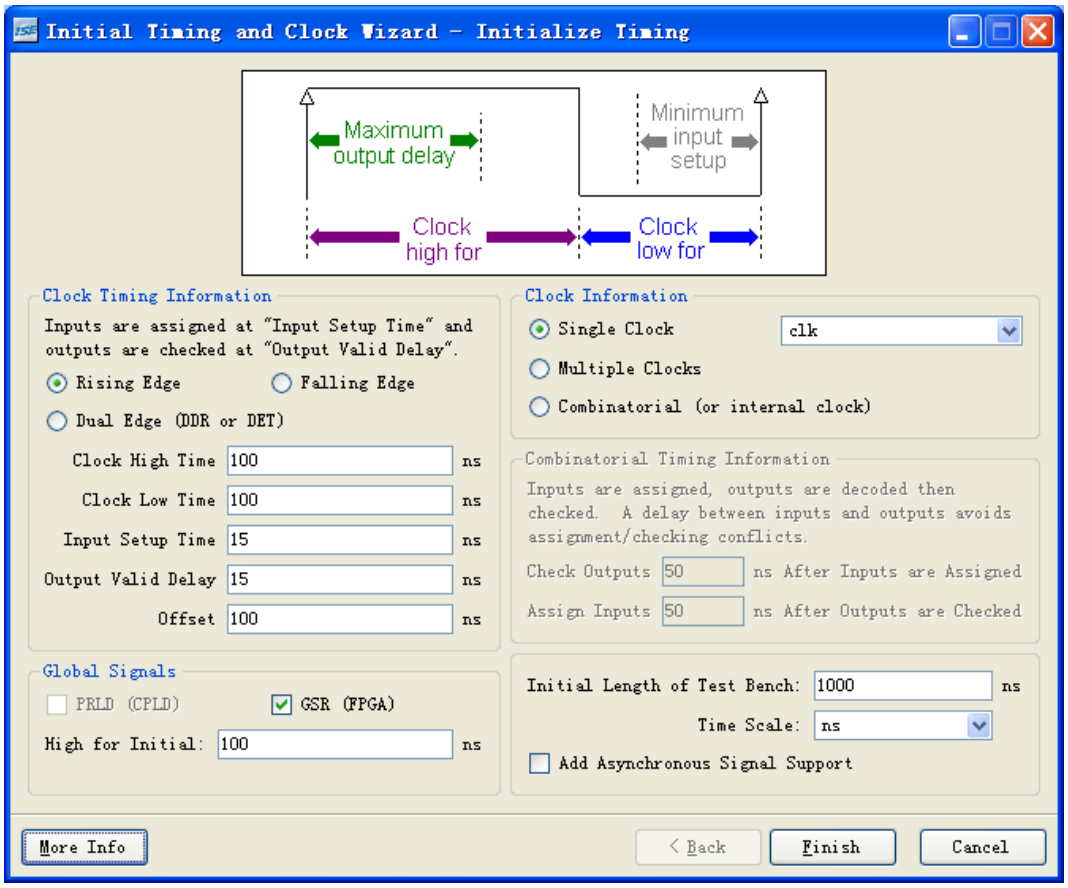


图 6-28　时序初始化窗口

时钟高电平时间和时钟低电平时间一起定义了设计操作必须达到的时钟周期，输入建立时间定义了输入在什么时候必须有效；输出有效延时定义了有效时钟延时到达后多久必须输出有效数据。默认的初始化时间设置如下：

- 时钟高电平时间（Clock High Time）：100ns
- 时钟低电平时间（Clock Low Time）：100ns
- 输入建立时间（Input Setup）：15ns
- 输出有效时间（Output Valid）：15ns
- 偏移时间（Offset）：100ns

单击“Finish”按钮，接受默认的时间设定。测试矢量波形显示如图 6-29 所示。

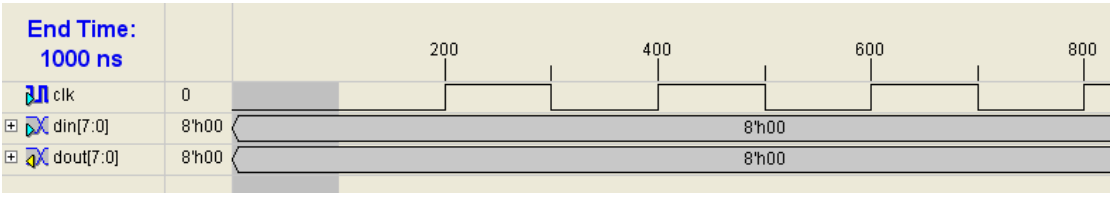


图 6-29　测试矢量波形

接下来，初始化输入（注：灰色的部分不允许用户修改），修改的方法为：选中信号，在其波形上单击，从该点所在周期开始，在往后所有的时间单元内该信号电平反相。点击 din 信号前面的“+”号，在 din[7]的第 2 个时钟周期内单击，使其变高；在 din[6]的第 3 个时钟周期内单击，使其变高；同样的方法修改 din[5]~din[0]信号，使其如图 6-30 所示。

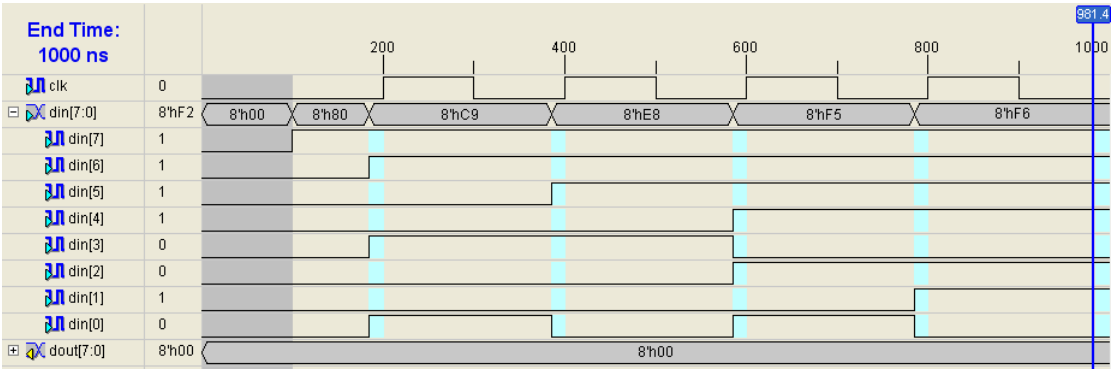


图 6-30　初始化输入

然后将 testbench 文件存盘，ISE 会自动将其加入到仿真的分层结构中，在代码管理区会列出刚生成的测试文件 test_bench.tbw，如图 6-31 所示。

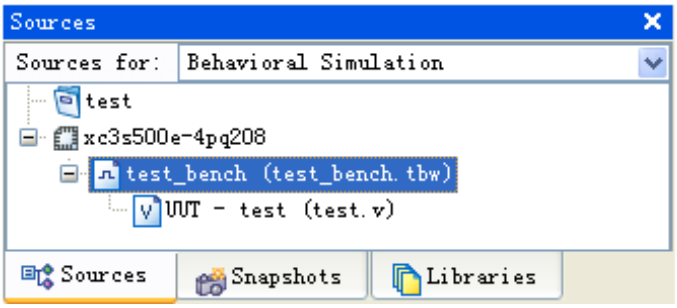


图 6-31　测试文件列表

选中 test_bench.tbw 文件，然后双击过程管理区的“Simulate Behavioral Model”，可完成功能仿真。同样，可在“Simulate Behavioral Model”选项上单击右键，设置仿真时间等。其仿真结果如图 6-32 所示。从图中可以看出，dout 信号等于 din 信号加 1，功能正确。

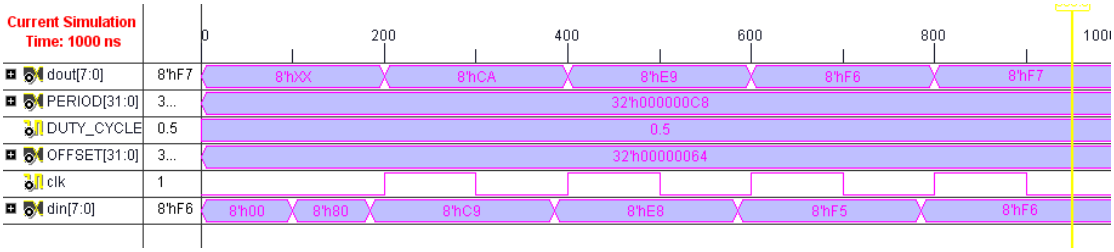


图 6-32　功能仿真结果

6.8.2　基于 Verilog HDL 测试平台的仿真

仿真过程是正确实现设计的关键环节，用来验证设计者的设计思想是否正确，及在设计实现过程中各种分布参数引入后，其设计的功能是否依然正确无误。仿真主要分为功能仿真和时序仿真。功能仿真是在设计输入后进行； 时序仿真是在逻辑综合后或布局布线后进行。基于 Verilog HDL 语言的行为级仿真操作（功能仿真）方法已经在 2.4.3 节进行了详细介绍，这里就不再对其进行说明。

时序仿真使用布局布线后器件给出的模块和连线的延时信息，在最坏的情况下对电路的行为作出实际地估价。时序仿真使用的仿真器和功能仿真使用的仿真器是相同的，所需的流程和激励也是相同的；惟一的差别是时序仿真加载到仿真器的设计包括基于实际布局布线设计的最坏情况的布局布线延时，并且在仿真结果波形图中，时序仿真后的信号加载了时延，而功能仿真没有。

因此，完成时序仿真的过程分为三个步骤：首先，添加设计的时序约束文件；其次，生成时序仿真模型；第三，完成时序仿真。

1．时序约束

（1）时序约束语法

一般来讲，添加约束的原则为先附加全局约束，再补充局部约束，而且局部约束比较宽松。其目的是在可能的地方尽量放松约束，提高布线成功概率，减少 ISE 布局布线时间。最常用的全局约束就是周期约束。

周期约束是附加在时钟网路上的基本时序约束，以保证时钟区域内所有同步组件的时序满足要求。在分析时序时，周期约束能自动处理寄存器时钟端的反相问题，如果相邻的同步元件时钟相位相反，则其延迟会被自动限制为周期约束值的一半，这其实相当于降低了时钟周期约束的数值，所以在实际中一般不要同时使用时钟信号的上升沿和下降沿。

硬件设计电路所能工作的最高频率取决于芯片内部元件本身固有的建立保持时间，以及同步元件之间的逻辑和布线延迟。所以电路最高频率由代码和芯片两部分共同决定，相同的程序，在速度等级高的芯片上能达到更高的最高工作频率；同样，在同一芯片内，经过速度优化的代码具有更高的工作频率，在实际中往往取二者的平衡。

在添加时钟周期之前，需要对电路的期望时钟周期有一个合理的估计，这样才不会附加过松或过紧的周期约束，过松的约束不能达到性能要求，过紧的约束会增加布局布线的难度，实现的结果也不一定理想。常用的工程策略是：附加的时钟周期约束的时长为期望值的 90%，即约束的最高频率是实际工作频率的 110%左右。

附加时钟周期约束的方法有两个：一是简易方法，二是推荐方法。简易方式是直接将周期约束附加到寄存器时钟网线上，其语法如下所示：

[约束信号] PERIOD =　{周期长度}　{HIGH | LOW}　[脉冲持续时间];

其中，[]内的内容为可选项，{}中的内容为必选项，“|”表示选择项。[约束信号]可为“Net net_name”或“TIMEGRP group_name”，前者表示周期约束作用到线网所驱动的同步元件上，后者表示约束到 TIMEGRP 所定义的信号分组上（如触发器、锁存器以及 RAM 等）。{周期长度}为要求的时钟周期，可选用 *ms*、s、*ns* 以及 *ps* 等单位，默认值为 *ns*，对单位不区分大小写。{HIGH | LOW}用于指定周期内第一个脉冲是高电平还是低电平。[脉冲持续时间]用于指定第一个脉冲的持续时间，可选用 *ms*、s、*ns* 以及 *ps* 等单位，默认值为 *ns*，如果缺省该项，则默认为 50%的占空比。如语句：

Net “clk_100MHz” period = 10ns High 5ns;

指定了信号 clk_100MHz 的周期为 10ns，高电平持续的时间为 5ns，该约束将被添加到信号 clk_100MHz 所驱动的元件上。

推荐方法常用于约束具有复杂派生关系的时钟网络，其基本语法为：

TIMESPEC “TS_identifier” = PERIOD “TNM_reference” {周期长度}
{HIGH | LOW}　[脉冲持续时间];

其中，TIMESPEC 是一个基本时序相关约束，用于标志时序规范。“TS_identifier”由关键字 TS 和用户定义的 identifier 表示，二者共同构成一种时序规范，称为 TS 属性定义，可在约束文件中任意引用，大大地丰富了派生时钟的定义。在使用时，首先要定义时钟分组，然后再添加相应的约束，如：

NET “clk_50MHz” = “syn_clk”;
TIMESPECT “TS_syn_clk” = PERIOD “syn_clk”　20ns HIGN 10ns;

同样，High|Low 后面的时间也可以通过占空比来表示，例如上面的约束等效于：

NET “clk_50MHz” = “syn_clk”;
TIMESPECT “TS_syn_clk” = PERIOD “syn_clk” 20ns HIGN 50%;

TIMESPEC 利用识别符定义派生时钟的语法为：

TIMESPEC “TS_identifier2” = PERIOD “timegroup_name” “TS_identifier1”
[* | /] 倍数因子 [+| -] phasevalue [单位]

其中，TS_identifier2 是要派生定义的时钟，TS_identifier1 为已定义的时钟，“倍数因子”用于给出二者周期的倍数关系，phasevalue 给出二者之间的相位关系。如：

定义系统时钟 clk_syn：

TIMESPEC “clk_syn” = PERIOD “clk” 5ns;

下面给出其反相时钟 clk_syn_180 以及 2 分频时钟 clk_syn_half：

TIMESPEC “clk_syn_180” = PERIOD “clk_180” clk_syn PHASE + 2.5ns;
TIMESPEC “clk_syn_180” = PERIOD “clk_half” clk_syn / 2;

（2）在 ISE 中添加时序约束的操作

和管脚分配一样，时序约束也通过 UCF 文件完成。如果设计中已经存在 UCF 文件，然后将时序约束添加即可；如果没有 UCF 文件，则需要新建 UCF 文件，然后加入时序约束。在 ISE 中，所有的约束都是添加在同一个 UCF 文件中的，包括管脚约束、时序约束以及区域约束等。

2．时序仿真操作实例

下面通过实例介绍如何基于 ISE Simulator 完成时序仿真（布局布线后仿真）。

例 6-18：在例 6-17 的基础上完成其相应的时序仿真。

（1）时序仿真的第一步就是添加时序约束，即通知 ISE，要求程序中的最高时钟为多少。本例添加的时序约束如下：

NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;

（2）在 ISE 中完成程序的综合、实现，并在实现过程完成后，单击过程管理区“Implement Design”选项前的扩展信号“+”，然后双击其中的“Generate Post-Place & Route Simulation Model”选项，生成时序仿真所需的时序模型，如图 6-33 所示。

在执行该操作时，ISE 会在信息显示区输出下列内容，最后一条信息表明仿真模型已经生成。该步骤完成后，“Generate Post-Place & Route Simulation Model”选项前会出现绿色的小圈，表示操作已经完成。

Started : "Generate Post-Place & Route Simulation Model".
INFO:NetListWriters:633 - The generated Verilog netlist contains Xilinx SIMPRIM simulation primitives and has to be used with SIMPRIM simulation library for correct compilation and simulation.
INFO:NetListWriters - Setup Simulation - To perform a setup simulation, specify values in the Maximum (MAX) field with the following command line modifier: -SDFMAX
INFO:NetListWriters - Hold Simulation - To perform the most accurate hold simulation, specify values in the Minimum (MIN) field with the following

command line modifier: -SDFMIN
INFO:NetListWriters:665 - For more information on how to pass the SDF switches
to the simulator, see your Simulator tool documentation.

Process "Generate Post-Place & Route Simulation Model" completed successfully

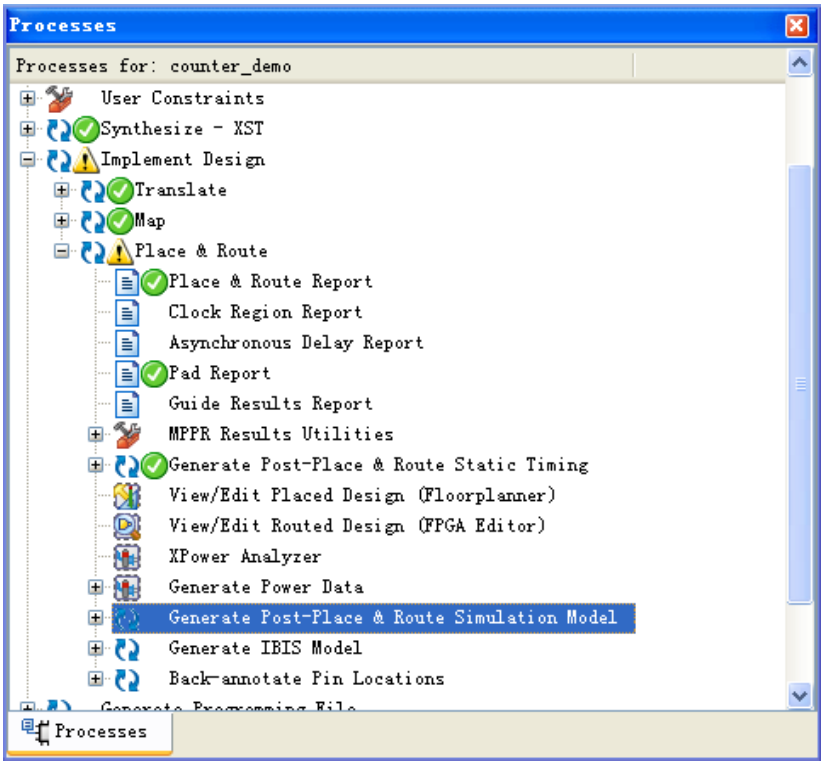


图 6-33　生成时序仿真模型操作示意图

同时，可以双击图 6-33 中的“Place & Route”栏目下的“Place & Route Report”，察看布局布线报告，从中得到设计的时序性能。本例的时序报告如图 6-34 所示，最高性能可达 307MHz（3.247ns），最差性能为 59.69MHz（16.753ns），评估时应该以最差性能为准。但这并不意味着该设计只能最高运行在 59.69MHz。因此 ISE 布局布线时采用“最大努力（Best effort）”，只要达到了时序约束后，便会停止更优布局布线方案的追求。如果，我们将时序约束修改为 500MHz，其时序性能一定不会满足，但肯定高于 59.69MHz。

```
Timing Score: 0

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------------------------------
  TS_clk = PERIOD TIMEGRP "clk" 20 ns HIGH  | SETUP   |    16.753ns|     3.247ns|       0|           0
  50%                                       | HOLD    |     1.325ns|            |       0|           0
------------------------------------------------------------------------------------------------------


All constraints were met.
```

图 6-34　设计的时序性能报告

既然这样，读者可能想？那么每次将约束设的很高，不就可以满足需求了吗？这样便忽略了一个问题，当时序约束很高时，ISE 需要更多的时间来寻找最优路径，会使得布局布线

时间非常长，超过设计人员的想象。因此，建议读者一般将时钟约束为所需频率的 1.1 倍。

（3）在工程管理窗口中，将“Source for”下拉框中的选项设置为“Post-Router Simulation”，并用鼠标单击选中对应的代码测试文件，然后在过程管理区双击“Xilinx ISE Simulation”栏目下的“Simulate Behavioral Model”选项，完成布局布线仿真。需要说明的是，时序仿真和功能仿真使用同一个仿真测试文件，不同在于：对于功能仿真，其仿真时间单位是没有物理意义的，时间单位为 1us 和 1ns 不会导致仿真结果发生变化，而在时序仿真中，时间单位具备相应的物理意义。如果假设时间单位为 1ns，则下面的时钟仿真代码产生的时钟，对于时序仿真模型而言是真正的 50MHz 时钟。

```
`timescale 1ns / 1ps
always #5 clk = ~clk;
```

在测试时，本例对应的测试文件内容如下所列：

```
`timescale 1ns / 1ps
module tb_counter_demo;

    //定义输入信号
    reg clk;
    reg reset;

    //定义输出信号
    wire [3:0] cnt;

    // 例化被测试模块(UUT)
    counter_demo uut (
        .clk(clk),
        .reset(reset),
        .cnt(cnt)
    );

    initial begin
        //初始化输入信号
        clk = 0;
        reset = 1;

        //全局复位 100ns
        #100;
        reset = 0;
        #40;
        reset = 1;
    end

    //生成 50MHz 的时钟
    always #10 clk = ~clk;

endmodule
```

在 ISE 执行上述代码，其相应的仿真结果如图 6-35 所示，可以看出在时序仿真中，即使未经过初始化，计数器的初始值便为 0，这表明时序仿真所用的模型就对应着芯片本身。当然，计数器值 1 和 2 之间跳变时，会产生毛刺，但毛刺并不在时钟信号上升沿，因此对设计的应用不会有影响。

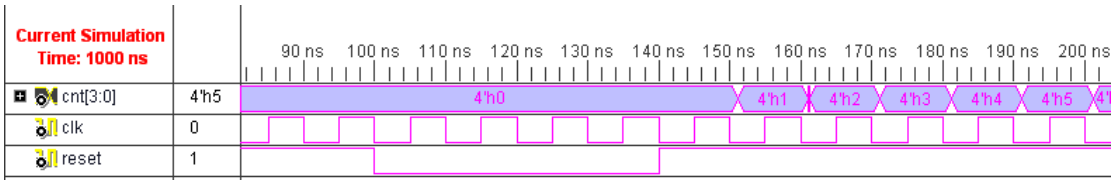


图 6-35　50MHz 的时序仿真结果

如果将测试代码中的时钟产生语句替换为：always #5 clk = ~clk;，则其时序仿真结果如图 6-36 所示，在复位过程中，计数器的值就发生错误，表明时序已经错乱，无法工作在相应的时钟。

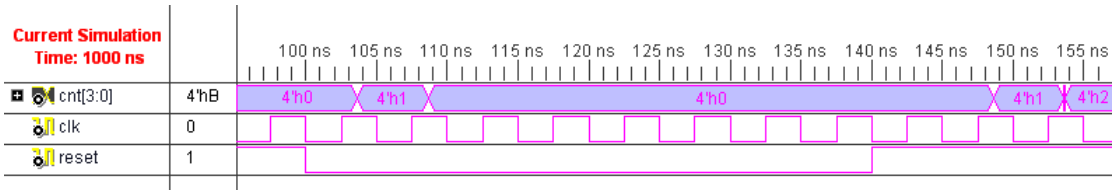


图 6-36　100MHz 的时序仿真结果

6.9 Xilinx 系统验证工具 ChipScope Pro

ChipScope Pro 调试和验证工具支持对在 Xilinx FPGA 芯片上实现的设计进行片上实时验证和调试，且在调试过程中，芯片还可以和软件端观测工具交互。与传统调试方法相比，可以使验证周期缩短 50%。此外，ChipScope Pro 还可以直接与 Agilent 逻辑分析仪配合使用，实现更强大、深入的 FPGA 信号分析。本节主要介绍 ChipScope Pro 的基本使用方法。

6.9.1　ChipScope Pro 工具简介

ChipScope Pro 可以分析任何内部 FPGA 信号，包括嵌入式处理器总线；在设计采集或综合之后，插入小型的、可配置的软件核，将引脚影响降至最低；在板上以达到或接近目标工程运行的速度验证 FPGA 设计；利用 FPGA 的可重编程性能，可以在几分钟或几小时内确定设计问题并修改设计；内置的软件逻辑分析器可以用来识别设计问题并进行调试，包括高级触发、过滤和显示选项，无需重新综合即可改变探针指向；可利用远程调试（从办公室到实验室，或在全球范围内）通过互联网连接进行调试；此外还包括 Agilent 科技推出的、用于实现功能强大的验证功能的逻辑分析器可选配件，可以探测包括从 FPGA 内部到板上任何地方的交叉互联信号。其典型的工作模式如图 6-37 所示。

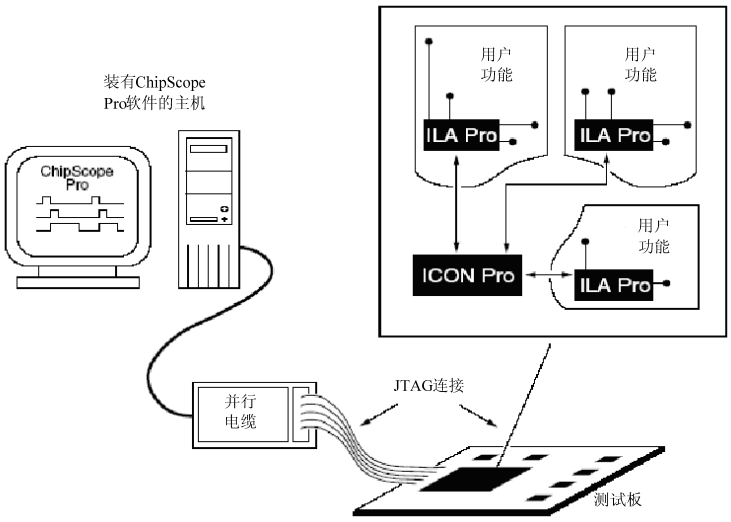


图 6-37　ChipScope Pro 的典型工作模式示意图

ChipScope Pro 为用户提供方便和稳定的逻辑分析解决方案，支持 Spartan 和 Virtex 全系列 FPGA 芯片，但对 PC 和芯片之间的 JTAG 通信电缆有一定的要求，目前支持下面四类：

- Platform Cable USB
- Parallel Cable IV
- MultiPRO (JTAG mode only)

ChipScope Pro 软件由 ChipScope Pro 核生成器（ChipScope Pro Core Generator）、ChipScope Pro 核插入器（ChipScope Pro Core Inserter）、ChipScope Pro 分析仪（ChipScope Pro Analyzer）以及 ChipScope Tcl 脚本接口（ChipScope Tcl Scripting Interface）四个组件组成，支持普通 FPGA 设计以及基于 FPGA 的嵌入式、SOC 系统，其具体功能如表 6-2 所示。

表 6-2　　ChipScope Pro 组件的功能简介

组件名称	功能描述
核生成器	提供下列网表和实例模版： • 集成控制核（Integrated Controller Pro core，ICON）； • 集成逻辑分析仪核（Integrated Logic Analyzer Pro cores，ILA）； • 适用于处理器外设总线的集成总线分析核（On-Chip Peripheral Bus core，OPB/IBA）； • 使用于处理器本地总线的集成总线分析核（Processor Local Bus core，PLB/IBA）； • 虚拟输入、输出核（Virtual Input Output core，VIO）； • 安捷伦跟踪核（Agilent Trace Core 2 ，ATC2）； • 集成的误比特率测试核（Integrated Bit Error Ratio Tester core ，IBERT）；
核插入器	自动将ICON、ILA以及ATC2等核插入到用户经过综合的设计中
分析仪	完成ILA、IBA/OPB、IBA/PLB、VIO以及IBER等核的芯片配置、触发设置以及跟踪显示等功能。其中不同的核提供不同的触发、控制以及跟踪捕获能力，例如：ICON核就能完成和专用边界扫描管脚的通信。
Tcl脚本接口	通过Tcl脚本语言和JTAG链，完成与芯片的交互通信。

在使用时，直接将 ICON、ILA、以及 ATC2 等核直接插入到设计的综合网表中，然后通过实现工具完成布局布线，将生成的比特文件下载到芯片中，从而实现在线逻辑分析器。

6.9.2 ChipScope Pro 开发实例

在 Xilinx 软件设计工具中，ISE 可集成 Xilinx 公司的所有工具和程序。ChipScope Pro 也不例外，在 ISE 中将其作为一类源文件，和 HDL 源文件、IP Core 以及嵌入式系统的地位是等同的。本节在 Xilinx Spartan3E-D 开发板上实现一个计数器模块，基于该模块详细介绍如何在 ISE 中新建 ChipScope 应用以及观察、分析数据的详细操作。

例 6-19：在 ISE 中实现一个 8 比特计数器，利用 ChipScope 分析其逻辑输出。

（1）新建用户工程，添加 mycounter.v 的源文件，其内容如下所列：

```
module mycounter(clk, reset, dout);
    input clk;
    input reset;
    output [7:0] dout;

    reg [7:0] dout;

    always @(posedge clk) begin
      if (reset == 0)
            dout <= 0;
       else
            dout <= dout + 1;
    end

endmodule
```

然后根据电路连接，添加相应的管脚约束。需要注意的是，虽然 ChipScope Pro 用于采集 FPGA/CPLD 内部信号，但其时钟信号所来自于外部输入，因此时钟信号 clk 信号必须分配相应的管脚约束。如果读者要测试对输入的相应，相关的输入信号也需要分配管脚，所有的输出信号可以不用对其分配管脚。

（2）综合工程，然后在 ISE 工程管理区，单击右键，选择“Add New Source”命令，在弹出的对话框中选择“ChipScope Definition and Connection File”类型，并在“File Name”栏输入 ChipScope 设计名称 mychipscope，如图 6-38 所示。

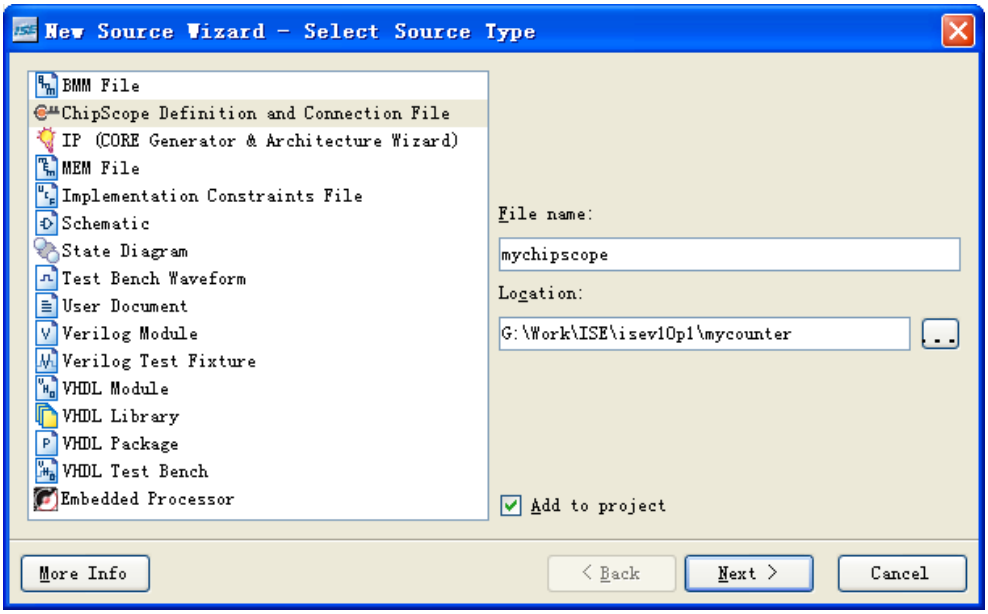


图 6-38　添加 ChipScope 设计示意图

单击“Next”按键，进入分析文件选择界面，这里会将该文件夹里所有的 HDL 设计、原理图设计都罗列出来（包括顶层模块和全部底层模块），供用户挑选，用鼠标单击即可选中，本例选择 mycounter，如图 6-39 所示。单击“Next”按键进入小结页面，单击“Finish”按键完成添加。

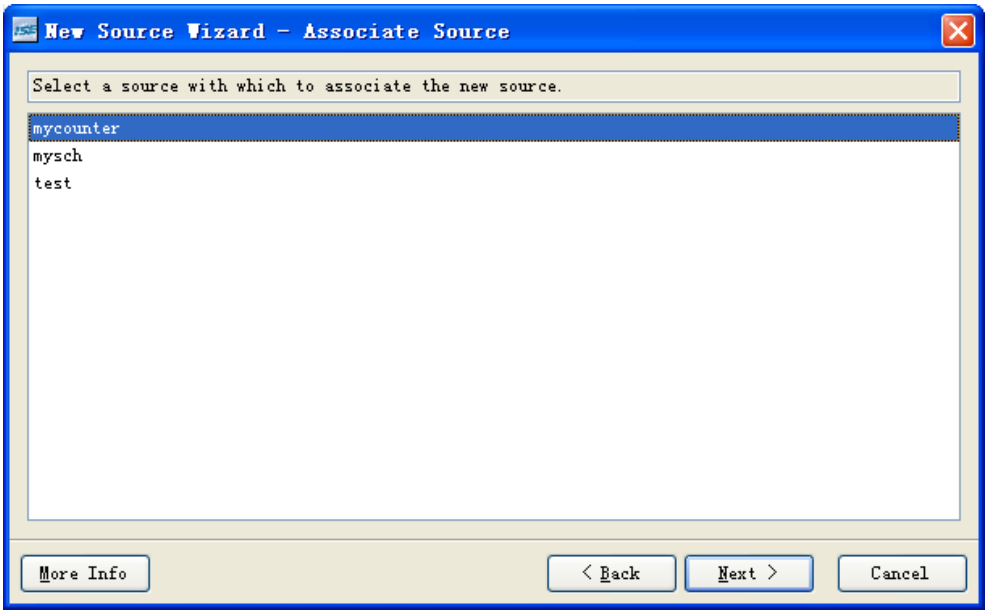


图 6-39　测试模块选择界面

（3）双击工程区 mycounter.v 下的子模块 mychipscope.cdc，可自动打开 Chipscope Pro Core Insterser 软件，添加触发单元和触发位宽。其中触发类型选为 Basic，位宽为 8 比特；设置采样深度为 4096，各步骤如图 6-40 到图 6-43 所示。

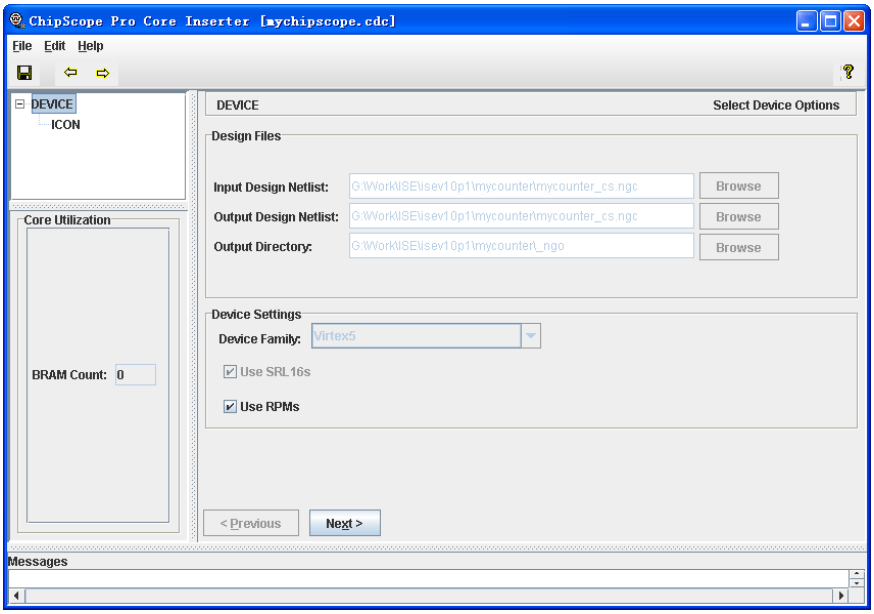


图 6-40　调试工程配置界面

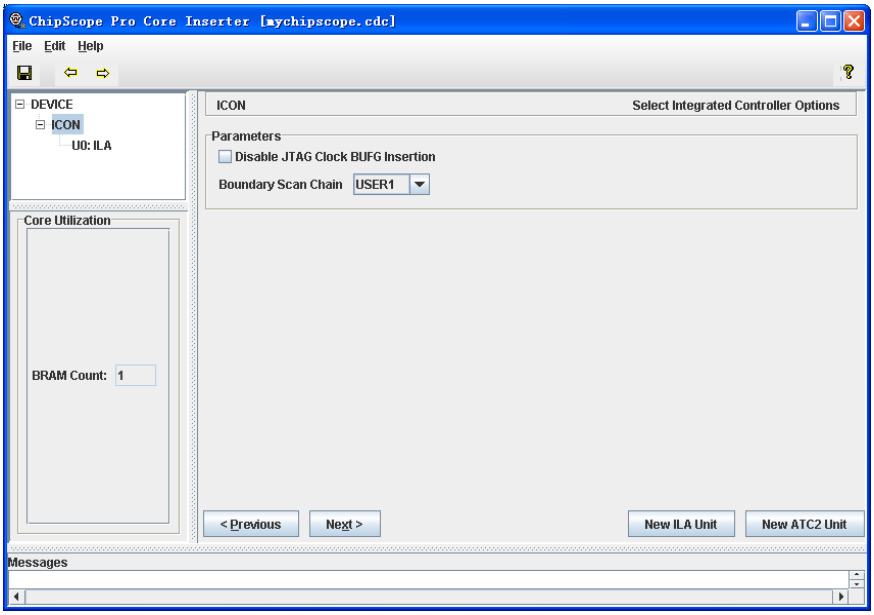


图 6-41　ICON 核配置界面

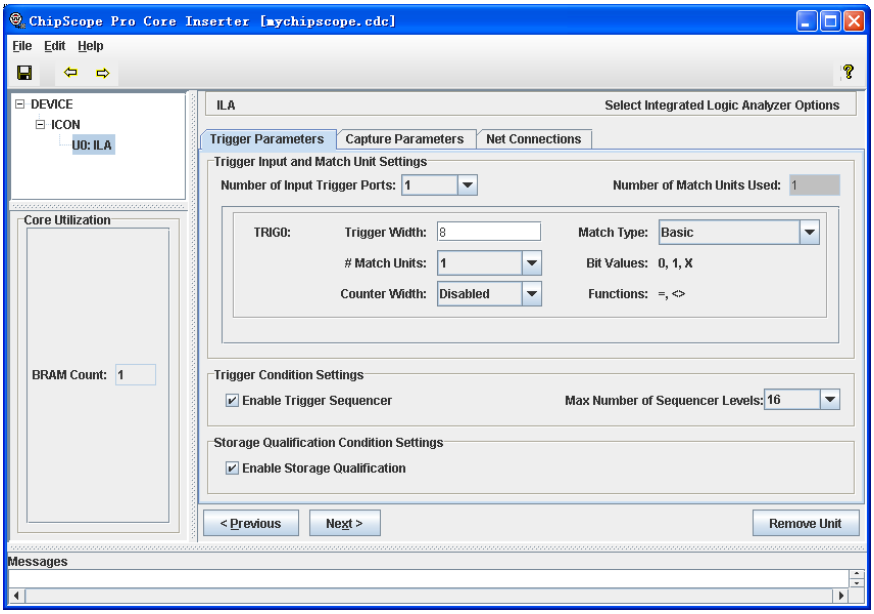


图 6-42　触发信号配置界面

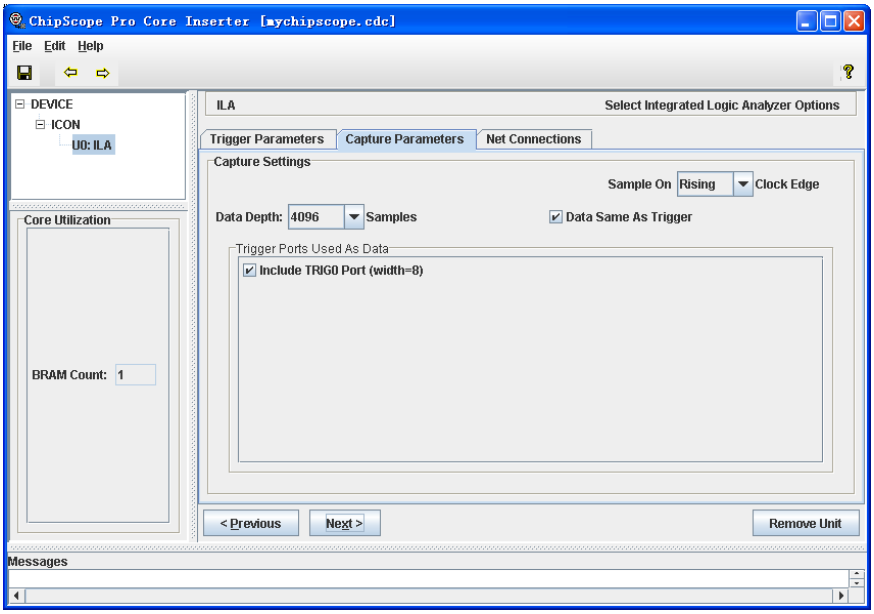


图 6-43　采集深度配置界面

（4）点击“Next”进入网表连接显示页面，如图 6-44 所示。其中如果用户定义的触发和时钟信号线有未连接的情况，则图中“UNIT”、“CLOCKPORT”以及“TRIGGERPORTS”等字样以红色显示；正确完成连接后则变成黑色。

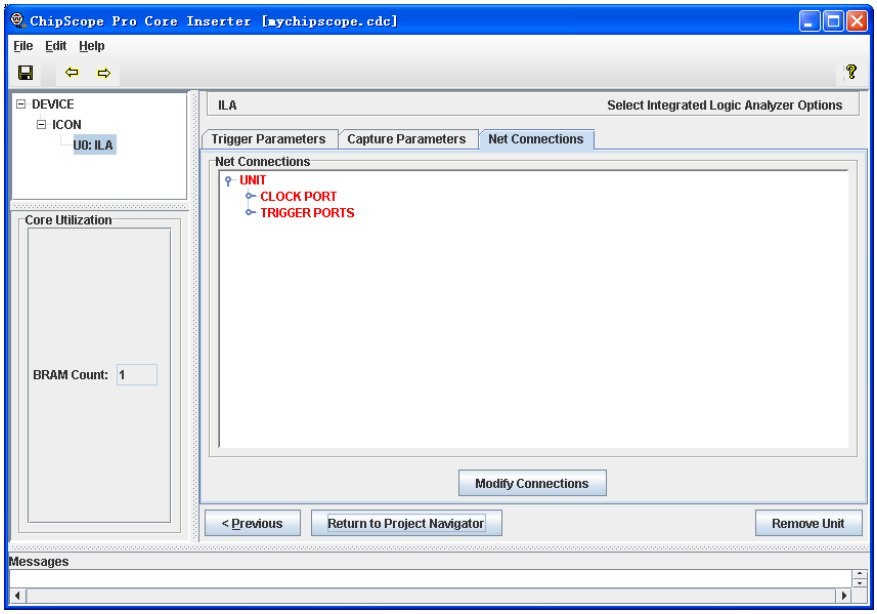


图 6-44　网表连接提示界面

单击图 6-44 中“Modify Connection”的按键，进入连接页面，时钟和数据的连接如图 6-45、图 6-46 所示。需要注意的是，ChipScope Pro 只能分析 FPGA 设计的内部信号，因此不能直接连接输入信号的网表，所以输入信号网表全部以灰色显示。如果要采样输入信号，可通过连接其输入缓冲信号来实现，时钟信号选择相应的 BUFGP，普通信号选择相应的 IBUF。如图 6-45 中所示，选择采样时钟时，选择了 CLK_BUFGP。

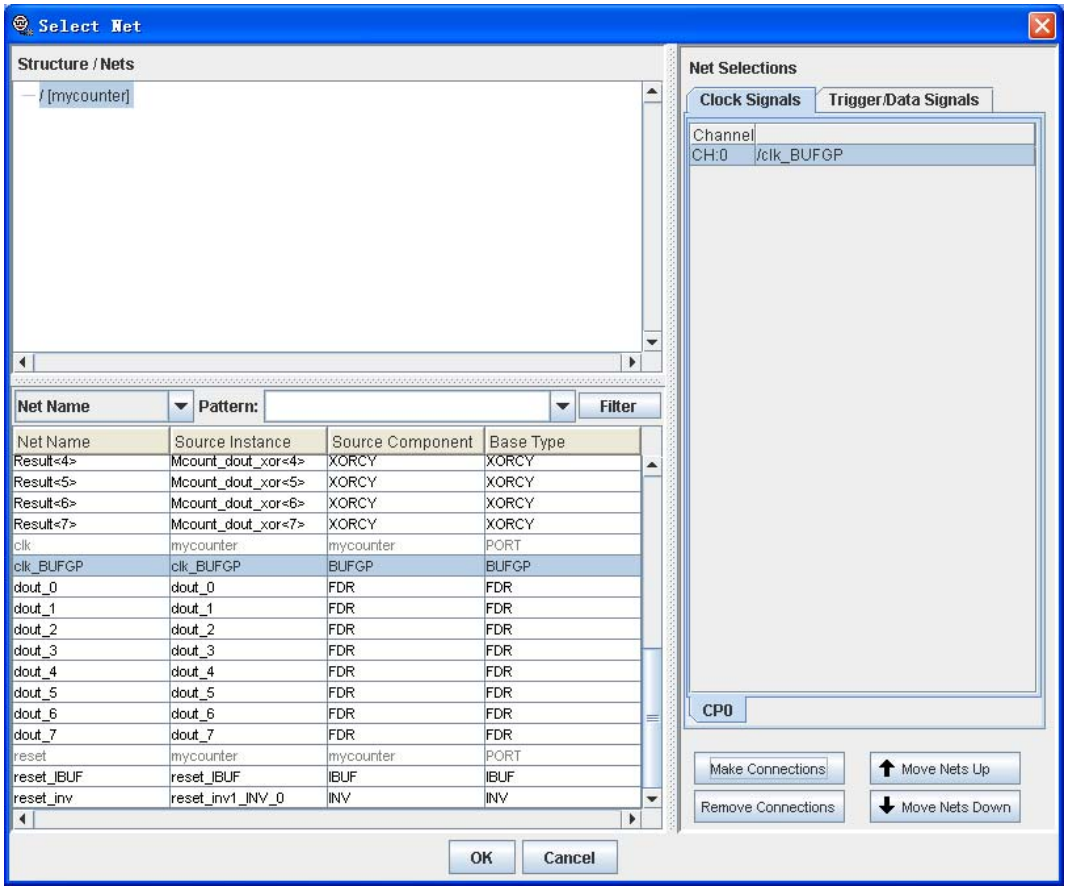


图 6-45　时钟网表连接界面

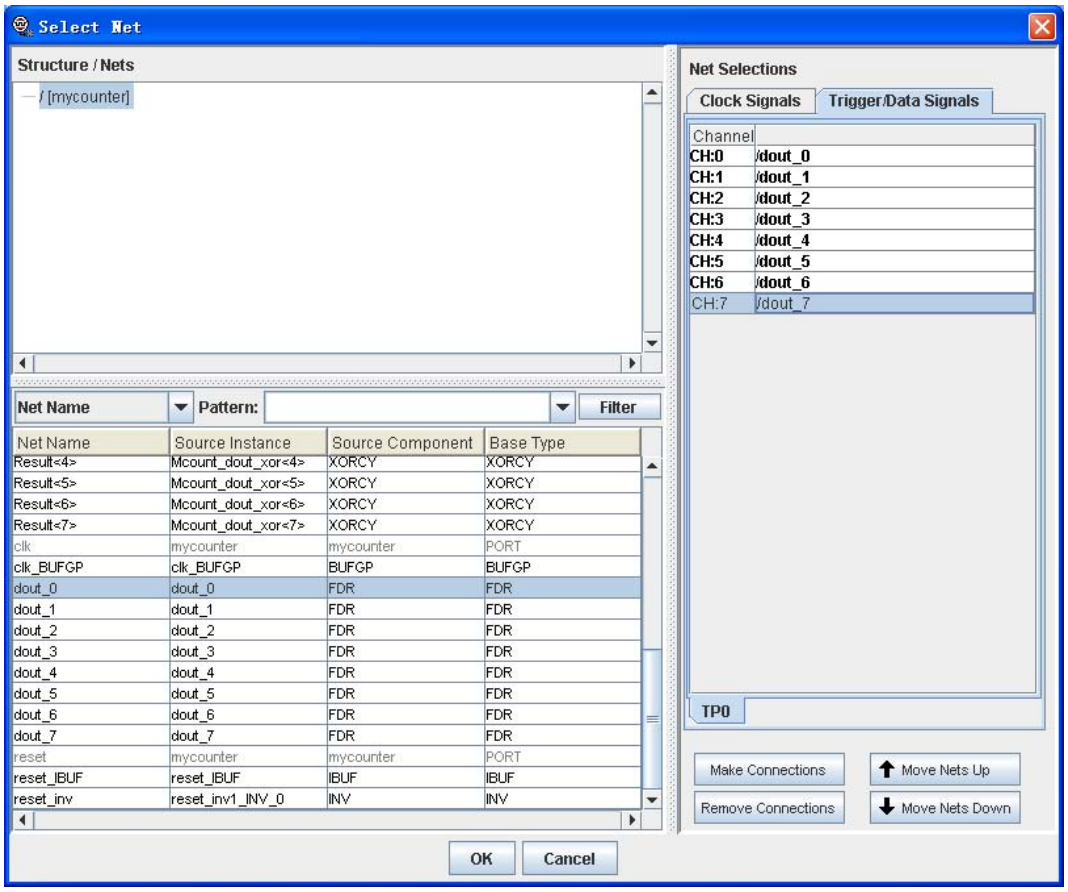


图 6-46　触发网表连接界面

连接完成后，单击“OK”按键返回连接显示界面，发现所有提示字符“UNIT”、“CLOCKPORT”以及“TRIGGERPORTS”没有红色，则单击“Return Project Navigator”，退出 Chipscope，返回到 ISE 中。否则需要再次点击“Modify Connection”按键重新连接。

（5）在工程中加入 UCF 文件，约束时钟、数据管脚位置。为了简化也可以只添加 clk 和 reset 这两个控制信号的管脚约束，其内容如下：

```
NET "clk" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
# Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
NET " clk " PERIOD = 20.0ns HIGH 40%;
NET "reset" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN ;
```

（6）在 ISE 过程控制区中双击“Implement Design”和“Generate Programming File”，可以完成实现以及生成可编程文件，并将设计人员插入的各类核也将被包含在比特文件中。生成配置文件后，双击图 6-47 所示的“Analyze Design Using Chipscope”图标，可自动打开 Chipscope Pro Analyzer 软件。

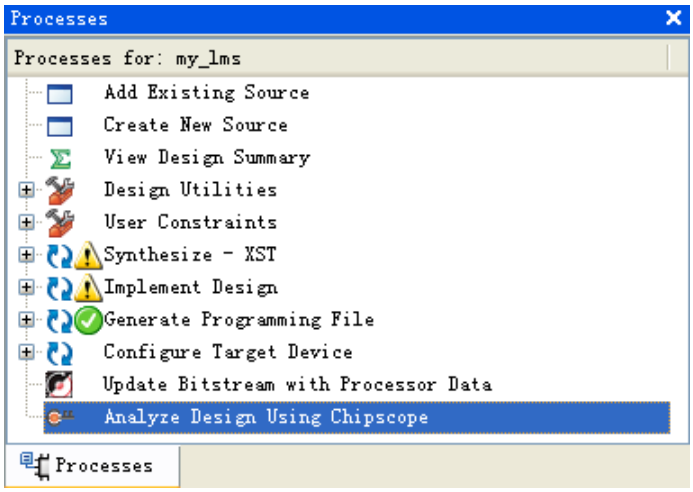


图 6-47　Chipscope Pro Analyzer 启动操作示意图

（7）在 Chipscope Analyzer 用户界面上点击工具栏上图标“”，初始化边界扫描链。等扫描完成后，单击“Device”菜单下“DEV: 0 My Device0(XC3S500E) → Configure”命令选择.bit 文件配置 FPGA。

（8）芯片配置完成后，选择“File”菜单的“Import”命令，可弹出 CDC 文件加载页面，选择相应的 CDC 文件，将会把所有以“Dataport<n>”的名称修改为综合后的线网名称。

（9）组合 cnt 总线信号。可按住“Ctrl”键，选择多个总线信号，单击右键，选择“Add to Bus”命令，将其组合成相应的总线信号，如图 6-48 所示。

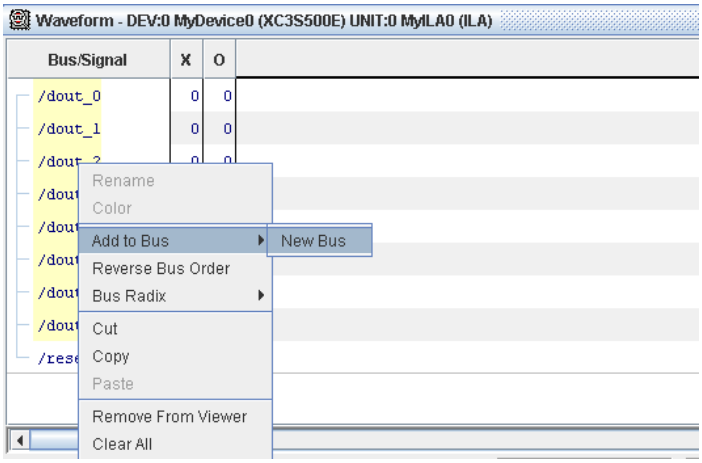


图 6-48　添加总线操作示意图

（10）不设定触发条件采集数据。点击工具栏的“”图标，开始采集数据。整体结果如图 6-49 所示，单击工具栏的“”按键，可放大信号，局部结果如图 6-50 所示。从分析结果可以看出，本设计在 FPGA 中成功地完成了 8 比特计数器的功能。

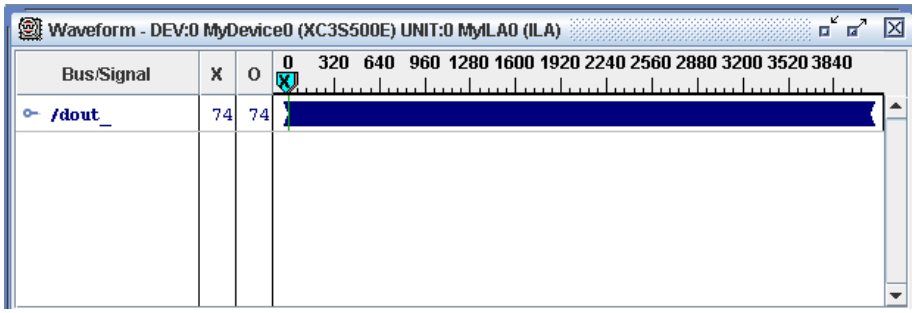


图 6-49　Analyzer 分析结果整体示意图

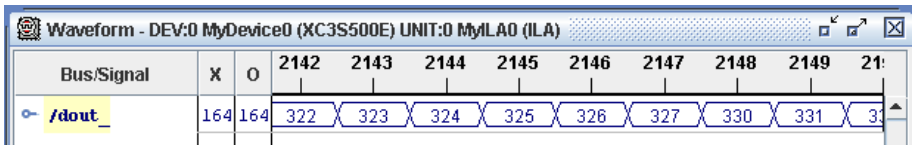


图 6-50　Analyzer 分析结果局部示意图

（11）设定触发条件采集数据。在“Trigger Setup”栏 Match 区域的“M0: Trigger Port0”行的 Value 列输入触发条件“0000_0000”，如图 6-51 所示。

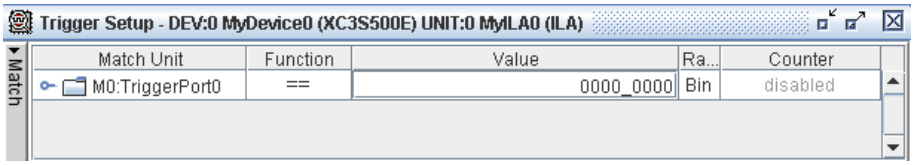


图 6-51　触发条件设置界面

点击工具栏的“▶”图标，开始采集数据，可以看到，采集结果的第一个数为 0，如图 6-52 所示。当然，用户可以根据需要设置更复杂的触发条件。

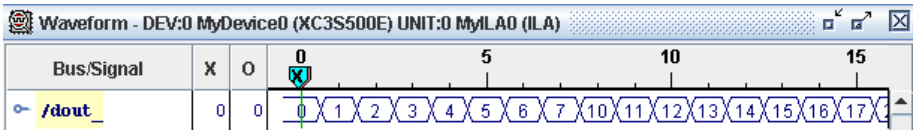


图 6-52　触发条件设置界面

（12）利用 Bus Plot 功能绘制输出信号波形。在工程区双击“Bus Plot”命令，然后在弹出窗口的“Bus Selection”区域选中“dout”，则会将采集数据以图形方式显示出来，如图 6-53 所示。由于本设计是 8 比特加 1 计数器，因此其波形就是幅度为 0 到 255 的锯齿波。

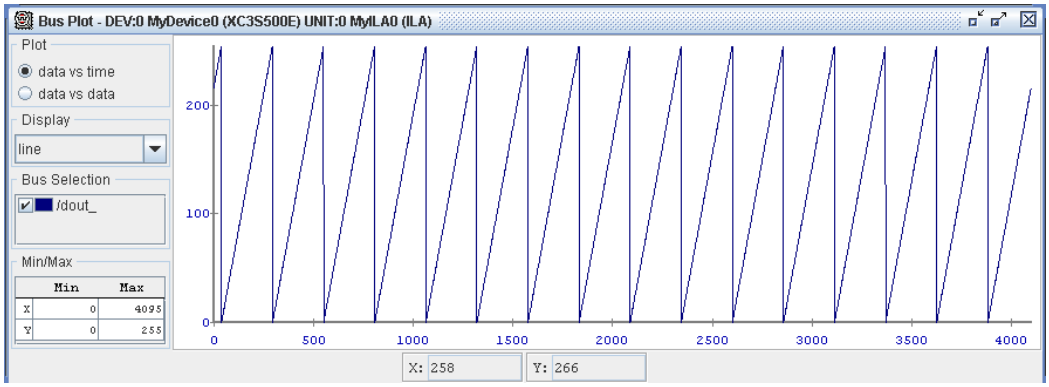


图 6-53　8 计数器的波形示意图

6.10　本章小结

本章主要介绍了 Verilog HDL 语言的验证与仿真的本质、原理以及常用语句，并重点说明了利用 Verilog HDL 语言编写仿真激励的方法以及 ISE Simulator 软件的使用和 ChipScope Pro 软件的使用。首先介绍了验证和仿真程序的概述以及方法论，以及 Verilog HDL 语言的语义和仿真原理，读者应该掌握事件队列与调度原则。其次，说明了 Verilog HDL 语言中的延迟控制语句和常用的行为仿真语句，包括循环语句、事件控制语句、Task 和 Function 语句、串行激励和并行激励语句等基本语句。第三，重点阐述了 UDP 语句，其虽然只能用于仿真，但说明了数字电路的本质，望读者能够切实理解。第四，介绍了仿真激励语句，包括

变量初始化、时钟信号、复位信号以及典型的数据信号的产生，并对仿真代码的效率进行了深入说明。最后，介绍了 Xilinx 的仿真工具 ISE Simulator 以及在线调试工具 ChipScope Pro 的使用方法，可完成所有语句的验证。

6.11 思考题

1．代码验证的原理和作用是什么？
2．简要描述测试平台的架构。
3．在 Verilog HDL 语言中，有哪几类测试方法？
4．简要说明 Testbench 的结构。
5．利用自己的语言说明 Verilog HDL 语义和仿真原理，特别是对事件队列的调度原理。
6．延迟控制在并行块和串行块中有什么区别？
7．循环语句在仿真过程中能够非正常中断退出
8．在 Testbench 中如何完成串行激励和并行激励？
9．简要描述 UDP 的使用方法？
10．常用的仿真激励有哪些？如何生成？
11．简要描述 ISE Simulator 的时序仿真操作流程，并说明其和功能仿真的区别。
12．简要描述 ChipScope 的操作流程。

第 7 章　系统任务和编译预处理语句

Verilog HDL 语言为常用的屏幕显示、线网值动态监视、暂停和结束仿真等操作提供了标准的系统任务（也叫系统函数）。编译预处理语句的作用在于通知综合器，哪些部分需要编译，哪些部分不需要进入编译。编译预处理语句和系统任务配合起来，可以通过一套程序就达到不同参数设定的目的，读取不同的测试输入，并在关键分支上插入显示的系统任务，快速完成大规模测试平台的建立，并有效提高设计效率。读者需要明白的：系统任务语句是不可综合的，只能用于仿真代码中；而编译预处理语句则是可用于可综合逻辑中，并且是层次化设计中常用的代码语句。本章主要介绍 Verilog HDL 语言中系统任务和编译预处理语句的使用方法。

7.1 系统任务语句

在 Verilog HDL 语言中，以“$”字符开始的标识符表示系统任务或系统函数。系统任务和函数即在语言中预定义的任务和函数。和用户自定义任务和函数类似，系统任务可以返回 0 个或多个值，且系统任务可以带有延迟。系统任务的功能非常强大，主要分为以下几类：

- 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）；
- 仿真控制任务（simulation control task）；
- 时序验证任务（timing check task）；
- 仿真时间函数 (simulation time function)
- 实数变换函数（conversion functions for real）；
- 概率分布函数（probabilistic distribution function）。

7.1.1　输出显示任务

输出显示任务指将各类信息输出到 ISE 信息显示区域以提示设计人员的系统任务，分为显示任务、探测监控任务以及连续监控任务三大类，下面分别对其进行介绍。

1．显示任务

（1）语法说明

显示系统任务用于信息显示和输出。这些系统任务进一步分为：显示和写入任务、探测监控任务以及连续监控任务。

显示和写入任务都能将信息显示出来，其区别在于显示任务将特定信息输出到标准输出设备，并且带有行结束字符，即自动换行；而写入任务输出特定信息时，不自动换行。显示任务的语法格式为：

```
task_name (format_specification1, argument_list1,
    format_specification2, argument_list2,
    ...,
    format_specificationN, argument_listN) ;
```

上述的 task_name 是指下面编译指令的一种：

$display，$displayb，$displayh，$displayo；

$write，$writeb，$writeh，$writeo。

其中$display、$displayb、$displayh、$displayo 为显示任务，$write、$writeb、$writeh 以及$writeo 为写入任务；“format_specification1”为格式控制参数，“argument_list1”为输出列表，包含期望显示的内容，既可以是数据，也可以是表达式。如果没有特定的参数格式说明，各任务的缺省值如下：

- $display 与$write：十进制数
- $displayb 与$writeb：二进制数
- $displayo 与$writeo：八进制数
- $displayh 与$writeh：十六进制数

格式控制参数“format_specification1”一般是用双引号括起来的字符串，包含两类信息，分别如下所示：

- 由“%”+格式字符组成，用于将输出的数据转换成指定的格式输出。表 7-1 给出了常用的几种输出模式。

表 7-1　　常用的输出格式

输出格式	简要说明
%h 或%H	将数据以 16 进制数的形式输出
%d 或%D	将数据以 10 进制数的形式输出
%o 或%O	将数据以 8 进制数的形式输出
%b 或%B	将数据以 2 进制数的形式输出
%c 或%C	将数据以 ASCII 码形式输出
%v 或%V	输出网络型数据信号强度
%m 或%M	输出等级层次的名字
%s 或%S	将数据以字符串的形式输出
%t 或%T	将数据以当前的时间格式输出
%e 或%E	将实数数据以指数的形式输出
%f 或%F	将实数数据以 10 进制的形式输出
%g 或%G	将数据以 10 进制数或指数的形式输出，无论何种格式都以最短的格式输出

- 特殊字符的显示。表 7-2 给出了一些特殊字符，用于输出转换序列和特殊字符。

表 7-2　　常用的输出格式

换码序列	简要说明
\n	换行
\t	横向调格，跳到下一个输出区
\\	反斜杠字符\
\”	反斜杠字符”
\o	1~3 位八进制数代表的字符
%%	百分符号%

输出列表中的数据显示位宽是自动根据输出格式进行调整的。这样，在显示输出数据时，经过格式转换后，总是用最小的位宽来显示表达式的当前值。如果输出列表中的表达式含有不确定值，则遵循下列处理原则：

如果在输出格式为十进制的情况下，若所有比特的逻辑数值为不定或高阻，则其输出为小写的'x'、'z'；若其部分比特为不定或高阻，则其输出为大写的'X'、'Z'。

如果输出格式为十六进制或八进制，在每 4/3 比特代表着一个有效数据，若每位数字中

所有比特的逻辑数值为不定或高阻，则该数字对应的输出为小写'x'、'z'；若其部分比特为不定或高阻，则其输出为大写的'X'、'Z'。

如果输出格式为二进制，则每个比特都可以输出 1、0、x、z。

（2）应用实例

在 ISE Simulator 中，显示语句的信息将在图 2-14 中的信息显示区显示。下面给出一些$display 语句和$write 语句的应用实例来具体说明其使用方法。

- $display 语句

例 7-1：$display 使用实例。

```
module system_display;
    initial begin
      $display (" 5 +10 = ", 5 +10);
      $display (" 5- 10 = %d", 5 -10);
      $displayb (" 10 -5 = ", 10 -5);
      $displayo (" 10 *5 = ", 10*5);
      $displayh (" 10 /5 = ", 10/5);
      $display (" 10 /-5 = %d", 10/-5);
      $display (" 10 %%3 = %d", 10%3);
      $display (" +5 = %d", +5);
      $display (" 4'b1001 = %d", 4'b1001);
      $display (" 4'b100x = %d", 4'b100x);
      $display (" 4'bxxxx = %d", 4'bxxxx);
      $display (" 4'b100z = %d", 4'b100z);
      $display (" 4'bzzzz = %d", 4'bzzzz);
    end
endmodule
```

上述程序在ISE Simulator中的仿真结果如图7-1所示。可以看出在输出格式为十进制时，如果表达式中的所有位为不定逻辑值，则输出结果为'x'；如果表达式中的部分位为不定逻辑值，则输出结果为大写的'X'；当表达式中的所有位为高阻逻辑值时，则输出结果为'z'；当表达式中的部分位为高阻逻辑值时，则输出结果为大写的'Z'。

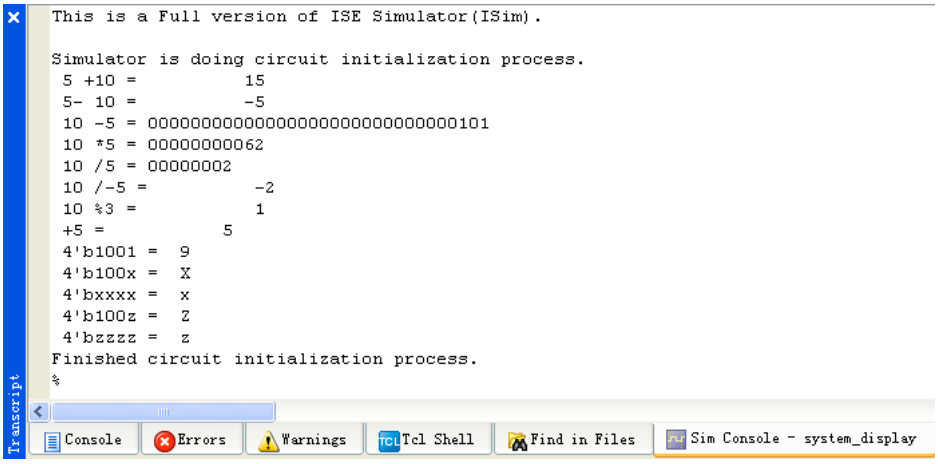


图 7-1　例 7-1 的仿真结果

在输出格式为 2 进制时，表达式的每一位都可以显示 1、0、x、z。当输出格式为 16 进制或 8 进制时，每 4 位或 3 位二进制数代表一位 16 进制数或 8 进制数；如果一位 16 进制数或 8 进制数所对应的 4 个或 3 个比特中，有一个为不定、高阻，则该 16 进制数或 8 进制数

的输出为大写的'X'、'Z'；如果都为不定、高阻，则其输出为'x'、'z'。

- $write 语句

$write 语句和$display 最大的区别就在于$write 输出时不换行，因此在使用时要注意加入换行符“\n”，确保输出内容便于区分。

例 7-2：$write 使用实例。

```
module system_write;
initial begin
   $write (" 5 +10 = ", 5 +10);
   $writeb (" 5- 10 = %d", 5 -10);
   $write (" 10 -5 = ", 10 -5);
   $write("\n");
   $write (" 5 +10 = %d \n", 5 +10);
   $writeb (" 5- 10 = ", 5 -10, "\n");
   $write (" 10 -5 = ", 10 -5, "\n");
end
endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 7-2 所示。

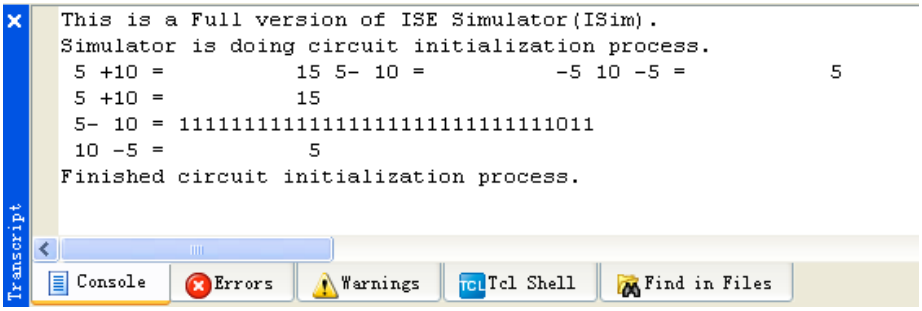


图 7-2　例 7-2 的仿真结果

2．探测监控任务

探测任务用于在某时刻所有事件处理完后，在这个时间步的结尾输出一行格式化的文本。常用的系统任务如下：

$strobe，$strobeb，$strobeh，$strobeo

这些系统任务在指定时间显示模拟数据，但这种任务的执行是在该特定时间步结束时才显示模拟数据。“时间步结束”意味着对于指定时间步内的所有事件都已经处理了。探测监控任务的语法如下：

```
$strobe(<functions_or_signals>);
$strobe ("<string_and/or_variables>", <functions_or_signals>);
```

其中，

这些系统任务的参数定义语法和$display 任务一样，但是$strobe 任务在被调用的时刻所有的赋值语句都完成了，才输出相应的文字信息。因此$strobe 任务提供了另一钟数据显示机制，可以保证数据只在所有赋值语句被执行完毕后才被显示。

例 7-3：给出$strobe 系统任务的演示实例。

```
module strobe_demo;
    reg a, b;
```

```
//initial 语句块 1
initial begin
    a = 0;
    $display("a by dispaly is :", a);              // displays 0
    $strobe("a by strobe is :", a);                // displays 1 ...
    a = 1;
end

//initial 语句块 2
initial begin
    b <= 0;
    $display("b by dispaly is :", b);              // displays x
    $strobe("b by strobe is :",b);                 // displays   0
    #5;
    $display("#5 b by dispaly is :", b);           // displays 0
    $display("#5 b by strobe is :", b);            // displays 1
    b <= 1;
end
```

endmodule

在 ISE Simulator 中执行上述程序，会得到图 7-3 所示的结果。可以看出，在第一个 initial 语句块中，由于采用阻塞赋值，在 0 时刻，a 的值就为 0，因此$display 语句输出 0；而在 0 时刻还有 a = 1 赋值操作，因此$strobe 语句输出为赋值完成后的 1。在第二个 initial 语句块中，赋值操作全部为非阻塞赋值，其赋值操作分为两步执行，因此在 0 时刻刚进入语句块时，b 的值并不是 0，因此$display 语句执行后输出为 x，然后 b 的值变为 0，$strobe 输出赋值完成后的 1。在当仿真时间再往前推进到 5 时，根据同样的原因，$display 输出 0，$strobe 输出 1。

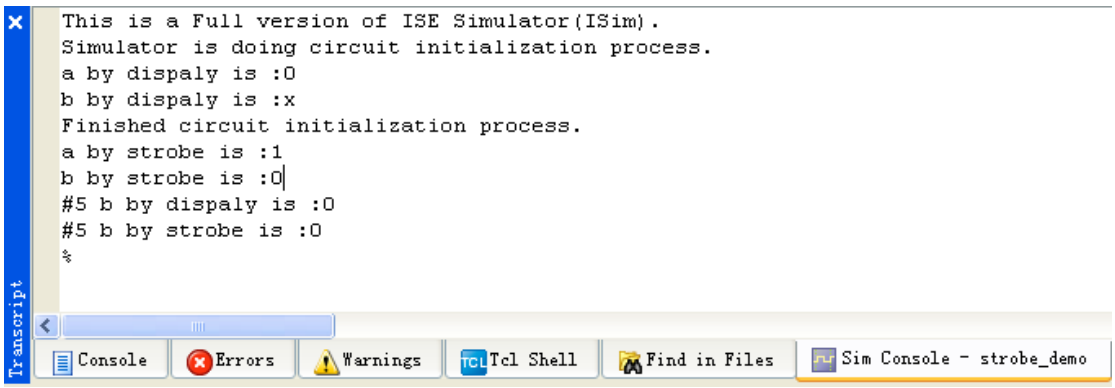


图 7-3　$strobe 实例仿真结果

3．连续监控任务

连续监控任务提供了监控和输出参数列表中表达式或变量值的功能，当一个或多个指定的线网或寄存器数值改变时，就输出一行文本。用于在测试设备中监控仿真行为。常用的连续监控任务关键字包括：

$monitor，$monitorb，$monitorh，$monitoro，$monitoron，$monitoroff

其相应的语法格式为：

```
$monitor(
    format_specification1, argument_list1,
    format_specification2, argument_list2,
    ...,
    format_specificationN, argument_listN);
$monitoron;
$monitoroff;
```

其中，format_specification1 以及 argument_list1 和$display 语句中的一样，包括各类细节说明。只要输出列表中的数值有一个发生变化，就会启动该$monitor 任务，整个输出列表中的所有变量或者表达式的值都会显示。如果在同一仿真时刻，有多个（两个或两个以上）参数表达式的值发生变化，则在该时刻只输出显示一次。典型的示例如下：

```
$monitor("a=%b, b=%b, out=%b\n", a, b, out);
```

$monitoron 和$monitoroff 这两个任务的作用是通过打开和关闭监控标志来控制、监控任务$monitor 的启动和停止，这样使得程序员可以很容易控制$monitor 的发生时间。在缺省情况下，监控任务在仿真的起始时刻就自动打开。这样，在多模块调试的情况下，会有多个模块调用$monitor，但是在任意时刻都只能有一个$monitor 任务被启动，因此就需要使用$monitoron 和$monitoroff 在特定时刻启动需要检测的模块，关闭其他模块，并在检测完毕后及时关闭，以便把$monitor 任务让给其他模块使用。

需要注意的是，$monitor 语句一旦出现，便会不间断地对被检测信号进行监视输出，不会停止下来，因此最好不要用其来观测循环的周期信号。

此外，$monitor 语句的输出参数还可以是$time 系统函数，其格式如下：

```
$monitor($time,"d=%b", d);
```

其中，$time 会列出当前仿真时间，可位于任何输出列表之前或之后，因此上述示例和下面的语句都可以输出当前仿真时间，只是仿真时间的显示位置不同。

```
$monitor("d=%b", d, $time);
```

下面给出一个$monitor 的应用实例。

例 7-4：$monitor 的应用实例。

```
module system_monitor;
    reg   [3:0] a, b;
    reg         clk;

     initial begin
         $monitor("Simulation time", $time," ns:", "a=%b, b=%b", a, b);
     end

     initial begin
          a = 0;
          b = 0;
          clk = 0;
     end
```

```
always #4 clk = ~clk;

always @(posedge clk) begin
    if(a == 15) begin
        b <= b + 1;
        a <= 0;
     end
     else begin
        a <= a + 1;
    end
end

endmodule
```

在 ISE Simulator 中完成上述程序的功能仿真时，就会产生图 7-4 所示的输出结果，可以看出，只要 a、b 信号的数值发生变化，$monitor 语句就会在信息显示窗口其对应的数值，达到了连续监测的功能。

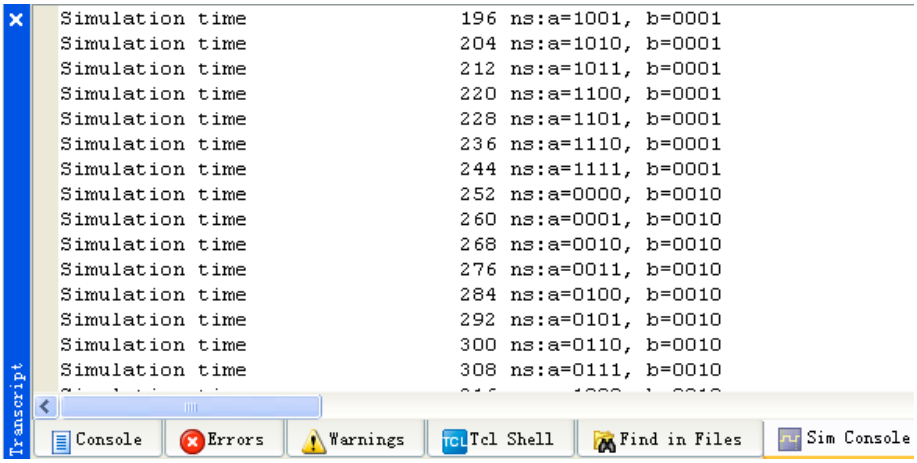


图 7-4　例 7-4 的仿真结果

7.1.2　文件输入输出任务

为什么要使用 Verilog HDL 语言读取/写入文件呢？主要有下列 3 个优点：首先，将数据准备和分析的工作从 Testbench 中隔离出来，便于协同工作；其次，可通过其他软件工具 C/C++、MATLAB 等快速产生数据；第三，将数据写入文档后，可通过 C/C++、Excel 以及 MATLAB 工具分析。因此，在测试代码中完成文件输入输出操作，是测试大型设计的必备手段。

1．文件操作语法

Verilog HDL 语言的文件操作和 C/C++语言类似，首先需要打开文件，然后对文件进行读/写操作，最后关闭文件。

（1）文件的打开和关闭

- 打开文件

系统函数$fopen 用于打开一个文件，将文件和 integer 指针关联起来，其语法格式如下：

```
integer file_pointer = $fopen(file_name);
//系统函数$fopen 返回一个关于文件的整数（指针）。
```

此外，在 IEEE Verilog HDL-2001 标准中，还提供了下面三个独立功能的系统任务：

file = $fopenr("filename");　　//以只读模式打开数据文件

file = $fopenw("filename");　　//以只写模式打开数据文件

file = $fopena("filename");　　//以读、模式打开数据文件，等效于$fopen

不过 ISE Simulator 10.1 版本只支持$fopen 任务，还不支持上面这 3 个独立功能的系统任务。

- 关闭文件

系统任务$fclose 可用于关闭一个文件，格式如下：

$fclose(file_pointer)；

（2）输出到文件

显示、写入、探测和监控系统任务都有一个用于向文件输出的相应副本，该副本可用于将信息写入文件。这些系统任务如下：

$fdisplay，$fdisplayb，$fdisplayh，$fdisplayo

$fwrite，$fwriteb，$fwriteh，$fwriteo

$fstrobe，$fstrobeb，$fstrobeh，$fstrobeo

$fmonitor，$fmonitorb，$fmonitorh，$fmonitoro

所有这些任务的第一个参数是文件指针，其余的所有参数是带有参数表的格式定义序列，含义和相应的不带字符'f'的系统任务相同。

（3）从文件中读取数据

Verilog HDL 语言中从文本读取数据有两大类方法：第一类为$fscanf 系统任务；第二类为$readmemb 和$readmemh 系统任务。上述两个任务都可以从文本文件中读取数据，并将数据加载到存储器。被读取的文本文件可以包含空白空间、注释和二进制（对于$readmemb）或十六进制（对于$readmemh）数字，每个数字由空白空间隔离。

- $fscanf 系统任务

$fscanf 系统任务和 C 语言中的 fscanf()函数语法和功能相同，从一个流中执行格式化输入，其在 Verilog HDL 中的用法如下所列：

integer file, count;

count = $fscanf(file, format, args);

$fscanf 任务从与 file 关联的文件中接受输入并根据指定的 format 来解释输入，解析后的值会被装入数组 args 返回。如果读写数据错误，则返回值 count 为-1

- $readmemb 和$readmemh 系统任务

$readmemb 和$readmemh 是 Verilog HDL 语言中专门用于读取数据的系统任务，其使用格式主要有下面的 6 种：

$readmemb("<数据文件名>", <存储器名>);

$readmemb("<数据文件名>", <存储器名>, <起始地址>);

$readmemb("<数据文件名>", <存储器名>, <起始地址>, <结束地址>);

$readmemh("<数据文件名>", <存储器名>);

$readmemh("<数据文件名>", <存储器名>, <起始地址>);

$readmemh("<数据文件名>", <存储器名>, <起始地址>, <结束地址>);

这两个任务用语从指定文件中读取数据并载入到指定存储器中，这两个任务可以在仿真时间任何时刻执行，被读取的文件只能包含下面的内容：

空格、换行、制表符（tab 键）和换页

注释

二进制或十六进制数字

对于$readmemb 而言，每个数字都为二进制形式；而对于$readmemh，数字将表示为十

六进制。对于未知值（x、X）高阻值（z、Z）以及下划线（_），在 Verilog HDL 语言中都可以用来指定一个数字。空格符和注释可以用来区分这些数字。

当文件被读取时，遇到的每个数字都在存储器中分配到一个连续字单元，可通过在系统任务中设定起始地址或结束地址来进行寻址，也可以在数据文件中设定地址。

如果使用数据文件中的地址，可以在@字符后面跟一个十六进制数，可以大小写混合形式表示，但不允许存在空格，例如：

@hhh…h

如果在系统任务中定义了数据的起始地址和结束地址，则数据文件里的数据按照该起始地址开始存放到存储器中，直到该结束地址，而忽略存储器定义语句的起始地址和结束地址。如果在系统任务中只给出起始地址而没有结束地址，可以从所给的地址载入数据，将存储器中说明的最右侧作为结束地址。如果在系统任务中没有定义地址，并且在数据文件中没有地址说明，则将存储器声明中的最左侧地址作为缺省的起始地址。如果在系统任务和存储器定义中都定义了地址信息，则数据文件里的地址必须在系统任务中地址参数声明的范围之内，否则将提示错误信息，并且装载数据到存储器中的操作会被中断。

下面给出下$readmemb 和$readmemh 的应用示例。

```
reg [0:3] Mem_A [0:63];
initial
$readmemb( "ones_and_zero.vec ", Mem_A) ;
//读入的每个数字都被指派给从 0 开始到 63 的存储器单元。
```

显式的地址可以在系统任务调用中可选地指定，例如：

```
$readmemb( "rx. vex ", Mem_A, 15, 30 ) ;
//从文件“rx.vec”中读取的第一个数字被存储在地址 15 中，下一个存储在地址
// 16，并以此类推直到地址 30。
```

2．文件操作实例

下面首先给出一个利用$fscanf 任务完成文件读取的实例。由于$fscanf 每次只能读取一个数据，因此需要通过循环语句来控制。

例 7-5：利用$fscanf 任务读取文本，并将读取内容写入输出文本中，文件依次存入了 1~9 包括 0 这 10 个数据，每个数据通过空格间隔。glbl.v is needed to run under modelsim.

```
`timescale 1ns / 1ps
`define NULL 0

module file_scanf;
        integer fp_r, fp_w;    //file_point
        integer flag;
        reg [3:0] bin;

        reg [15:0] data_in;
        reg [15:0] cnt = 10;

        initial begin : file_fscanf

            fp_r = $fopen("data_in.txt", "r");
            fp_w = $fopen("data_out.txt", "w");
```

```
        if (fp_r == `NULL) // If error opening file
            disable file_fscanf; // Just quit

        if (fp_w == `NULL) // If error opening file
            disable file_fscanf; // Just quit

        while (cnt > 0) begin
                flag = $fscanf(fp_r, "%d", data_in);
                cnt = cnt - 1;
                $write("%d", data_in, " ,");
                $fwrite(fp_w, "%d\n", data_in );
            #5;
        end

        $fclose(fp_r);
        $fclose(fp_w);
    end

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 7-5、7-6 所示。图 7-5 给出了程序运行后信息显示区的输出信息，可以看到显示信息和文本文件中一致，表明程序代码的正确性。

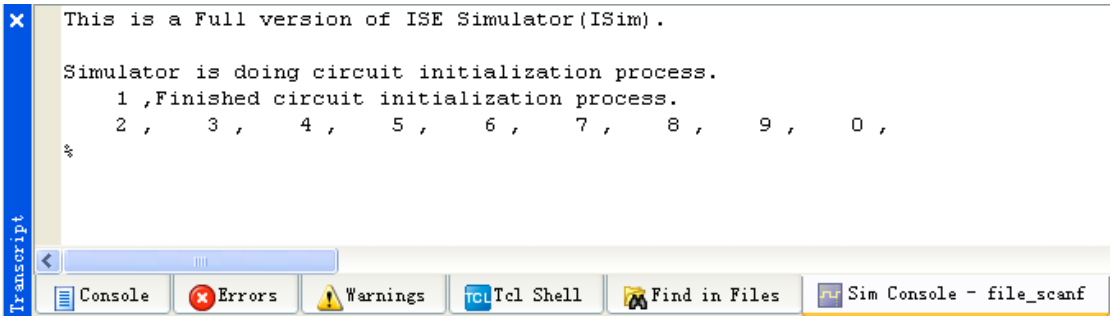


图 7-5　例 7-5 的信息显示区输出

利用系统任务读取数据的主要目的是用于完成代码仿真，因此最重要的为 Testbench 提供波形激励，因此还需要在波形图中观察是否正确。图 7-6 给出了代码的波形仿真结果，可以看出 data_in 信号的波形确实是数据文件的 1~9 包括 0。

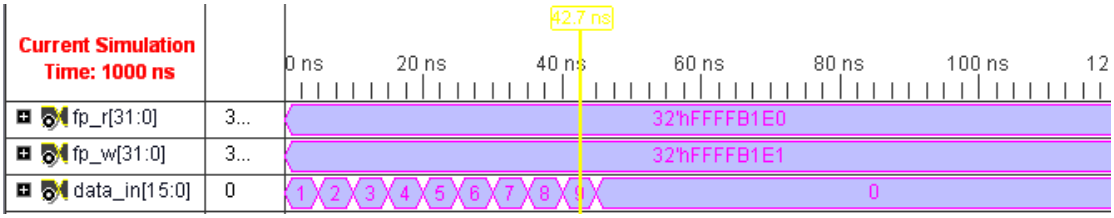


图 7-6　例 7-5 的仿真结果

由于$readmemb 或$readmemh 任务一次性将所有数据全部读入寄存器中，因此首先需要一个二维数组来存取数据，然后再将数组中数据依次传递给仿真输入。下面给出一个利用

$readmemb 或$readmemh 来读取数据的实例。

例 7-6：利用$readmemb 或$readmemh 任务读取文本，并将读取内容写入输出文本中。

```
`timescale 1ns / 1ps
module readmemh_demo;

        parameter data_period = 4;
        parameter data_num = 10;
        // Declare memory array that is ten words of 32-bits each
        reg [31:0] Mem [0:data_num - 1];
        reg [31:0] data;

        // Fill the memory with values taken from a data file
        initial $readmemh("data_in.txt",Mem);      //put      data      into      array

        // Display the contents of memory
        integer k;
        initial begin
            #data_period;
            $display("Contents of Mem after reading data file:");
            for (k=0; k< data_num; k=k+1) begin
               data = Mem[k];
               #data_period;
               $display("%d:%h",k,Mem[k]);
            end
        end

endmodule
```

上述程序的仿真结果分为波形图和控制台输出两部分，分别如图 7-7、7-8 所示。图 7-7 给出了程序运行后信息显示区的输出信息，可以看到显示信息和文本文件中一致，表明程序代码的正确性。

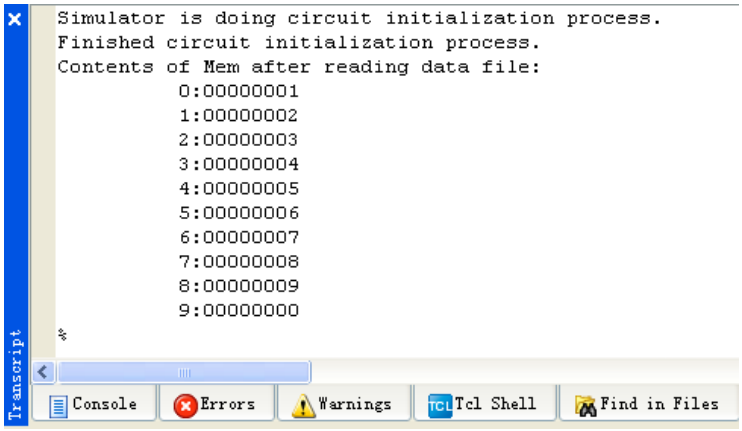


图 7-7　例 7-6 的控制台输出

图 7-8 给出了代码的波形仿真结果，可以看出 data_in 信号的波形确实是数据文件的 1~9 包括 0。

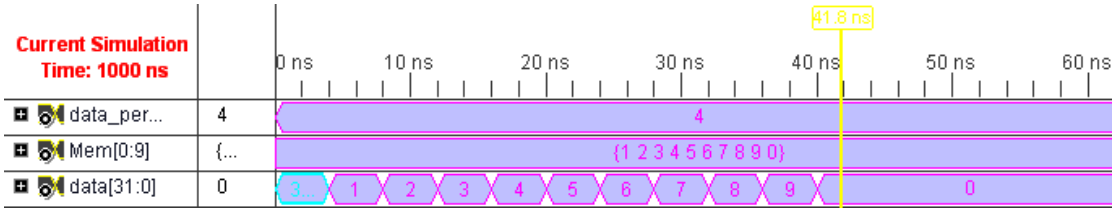


图 7-8　例 7-6 的波形输出

7.1.3　时间标度任务

1．系统任务$printtimescale

该任务给出打印仿真代码中的的仿真时间单位和仿真时间最小可分辨精度，其语法格式分别如下：

$ printtimescale;

$ printtimescale (hier_path_to_module) ;

若$printtimescale 任务没有指定参数，则打印包含该任务调用的仿真模块的时间单位和精度，即包含改系统任务的模块。如果指定了模块的层次路径名为参数，则系统任务输出指定模块的时间单位和精度。下面给出$printtimescale 任务的应用实例。

例 7-7：给出$printtimescale 的应用实例。

```
`timescale 1ns / 1ps
module printtimescale_demo;

    initial begin
      $printtimescale(printtimescale_demo);
    end

endmodule
```

上述代码在 ISE Simulator 中的仿真结果如图 7-9 所示，可以看出$printtimescale 任务输出了 printtimescale_demo 模块的仿真时间单位和最小分辨间隔。

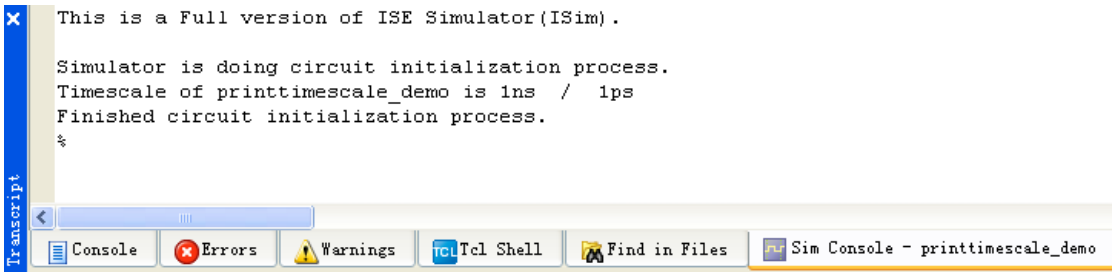


图 7-9　例 7-7 仿真结果示意图

2．系统任务$timeformat

$timeformat 任务用于设定当前的时间格式信息，使得$time 系统按照预定格式输出，其任务语法如下：

$timeformat(units_number, precision, suffix, numeric_field_width) ;

其中 units_number 是 0 到-15 之间的整数值，表示打印的时间单位，其含义如表 7-3 所列；precision 是在小数点后面要打印的小数位数；suffix 是在时间值后面打印的一个字符串；numeric_field_width 是打印的最小数量字符包括前面的空格，如果要求更多字符那么打印的

字符更多。如果没有指定变量，默认地使用下面的值 units_number：仿真精度；precision 为 0；suffix 为空字符串；numeric_field_width 为 20 个字符。

表 7-3　　units_number 的含义列表

数值	时间含义	数值	时间含义
0	1 s	-1	100 ms
-2	10 ms	-3	1 ms
-4	100 us	-5	10 us
-6	1 us	-7	100 ns
-8	10 ns	-9	1ns
-10	100 ps	-11	10 ps
-12	1 ps	-13	100 fs
-14	10 fs	-15	1 fs

用`timescale、$timeformat 和$realtime 带%t 指定和显示仿真时间；用$display、$monitor 或其他显示任务

下面给出时间表度任务的调用实例。

例 7-8：$timeformat 任务的调用实例。

```
`timescale 1ns / 1ps
module tb_test;
    initial begin
            $display("Current simulation time is %t",$time);
            $timeformat(-10, 2, " x100ps", 20); // 20.12 x100ps
            $display("Current simulation time is %t",$time);
     end
endmodule
```

在 ISE Simulator 中执行上述代码，将会在控制台输出窗口显示$display 任务中% t 说明符的值，其结果如图 7-10 所列。如果没有指定$timeformat，%t 按照源代码中所有时间标度的最小精度输出。

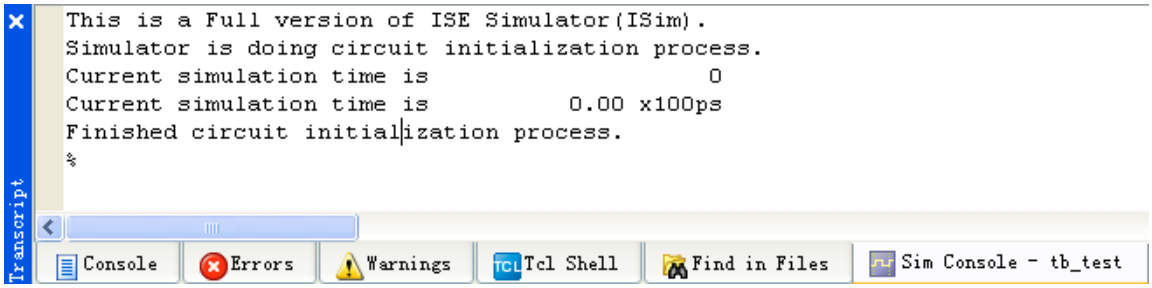


图 7-10　仿真时间格式控制语句的仿真结果示意图

7.1.4　仿真控制任务

仿真控制任务包括仿真完成和暂停这两类操作，分别对应着$finish 和$stop 任务，下面对其进行介绍。

1．$finish 任务

$finish 任务的作用时退出仿真器，并将控制返回到操作系统，有两种用法：

$finish;

$finish(n);

其中，$finish 可以带参数（0、1、2），根据参数的值输出不同的特征信息，各参数的值

为表 7-4 所列。如果不带参数，默认$finish 的参数值为 1。

表 7-4　　$finish 参数值说明列表

参数值	功能说明
0	结束任务，不输出任何信息
1	结束任务，输出当前仿真时刻和位置
2	结束任务，输出当前仿真时刻、位置以及仿真过程中所用的内存和 CPU 时间的统计

例如，在 ISE Simulator 中执行“$finish(1);”语句，则会在控制台输出如下所示的类似信息：

Stopped at time : 1000 ns : File "D:/work/ise10p1/verilog//demo2/system_display.v" Line 40

2．$stop 任务

$stop 任务的作用是将仿真器置城暂停模式，使仿真进程被挂起。在这一阶段，交互命令可能被发送到模拟器。下面是该命令使用方法的例子。

initial #500 $stop;

500 个时间单位后，模拟停止。

下面给出$finish 和$stop 任务的典型应用模版，例如：

```
if (…) begin
   $ stop;                //在某一条件下中断仿真
end
```

又如：

```
# Ntime $finish;       //在某一特定时刻，仿真结束
```

7.1.5　仿真时间函数

在 Verilog HDL 语言中，有两种类型的仿真时间函数$time、$realtime，通过这两个系统任务可以输出当前的仿真时刻。

其中：

$time：　　返回一个 64 位的整数来表示当前的仿真时刻值。

$realtime：返回实型数来表示当前的仿真时刻值。

1．系统任务$time

$time 任务返回一个 64 位的整形模拟时间值，其数值由调用模块中的`timescale 语句指定。下面给出一个$time 任务应用实例。

例 7-9：$time 任务调用实例。

```
`timescale 10ns /1ns
module time_demo;

    reg   tmp;

    parameter p = 1.7;

    initial begin
        $monitor ($time, "tmp= ", tmp);
        #p tmp = 0;
        #p tmp = 1;
        #p tmp = 0;
    end
```

endmodule

在 time_demo 模块中，time_demo 原本要在 17ns 时刻处将寄存器 tmp 的数值修改为 0，在时刻 34ns 时刻处将 tmp 数值设置为 1，在 51ns 时刻处再将 tmp 设置为 0。程序在 ISE Simulator 中的仿真结果如图 7-11 所示，可以看出，$time 按模块 time_demo 的时间单位比例返回值和预想存在差异，其原因是因为$time 输出的时刻总是仿真时间单位的整数倍，所有的小数都需要取整。其中，仿真时间最小分辨率并不影响输出数值的取整操作。

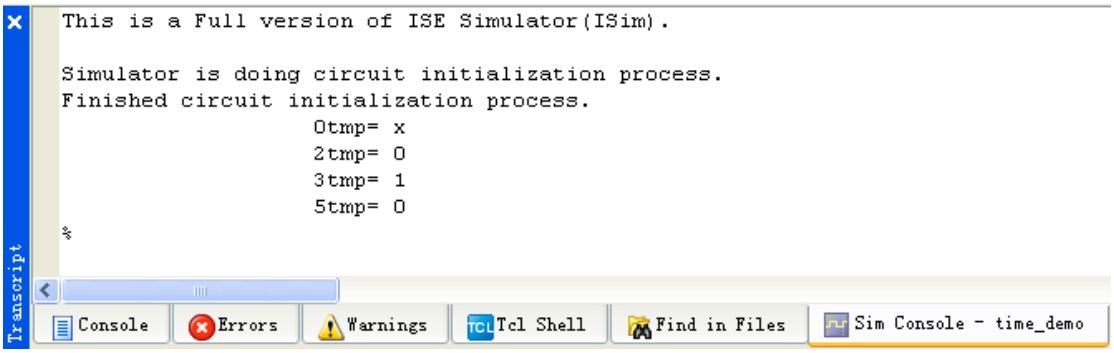


图 7-11　$time 系统任务仿真结果示意图

2．系统任务$realtime

$realtime 和$time 的作用是一样的，只是$realtime 返回的时间数字是一个实型数，当然该数字也是以时间尺度为基准的。下面给出一个$realtime 应用实例。

例M：$realtime 任务调用实例

```
`timescale 10ns /1ns
module realtime_demo;

    reg    tmp;

    parameter p = 1.7;

    initial begin
        $monitor ($realtime, "tmp= ", tmp);
        #p tmp = 0;
        #p tmp = 1;
        #p tmp = 0;
    end

endmodule
```

程序和例 M 相比，只是将$time 替换成$realtime，其在 ISE Simulator 中的仿真结果如图 7-12 所示，可以看出，$realtime 将仿真时间经过尺度变换后输出，没有取整操作，返回的是一个实数。

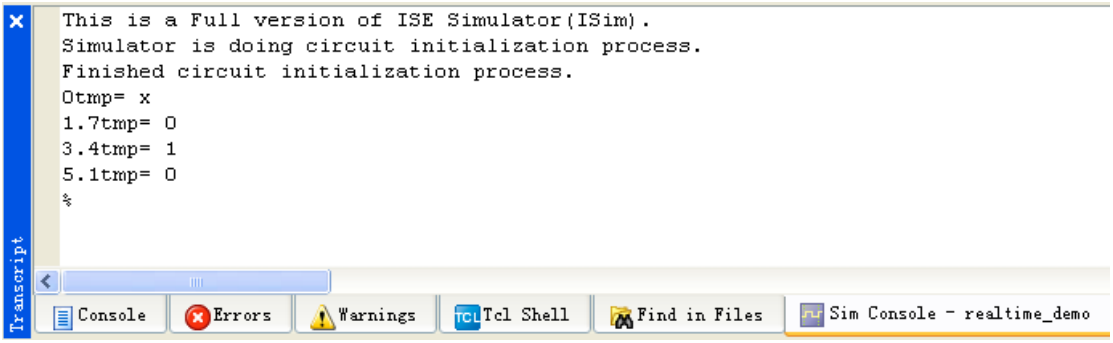


图 7-12　$realtime 系统任务仿真结果

7.1.6　数字类型变换函数

下列系统函数是数字类型变换的功能函数：

- $rtoi(real_value)：　通过截断小数值将实数变换为整数。
- $itor(integer_value)：将整数变换为实数。
- $realtobits(real_value)：将实数变换为 64 位的实数向量表示法（实数的 IEEE 745 表示法）
- $bitstoreal(bit_value)：将位模式变换为实数（与$realtobits 相反）。

下面给出一个数字类型变化的应用示例。

例 7-10：数字类型变换任务调用实例。

```
module zhuanhuan_demo;
    reg   [63:0] a , b, c, d;

     initial begin
        $monitor ("a= ", a, "\n",
                  "b= ", b, "\n",
                  "c= ", c, "\n",
                  "d= ", d, "\n"
                      );

        a =   $rtoi(3.14);
        b =   $itor(a);
        c =   $realtobits(3.14);
        d =   $bitstoreal(c);

     end

endmodule
```

上述程序在 ISE Simulator 中的执行结果如图 7-13 所示，可以看出，由于$rtoi 任务有截断操作，因此其输出数据经过$itor 不能恢复；由于$realtobits 和$bitstoreal 只是完成数据类型转换，因此其二者互为逆变换。

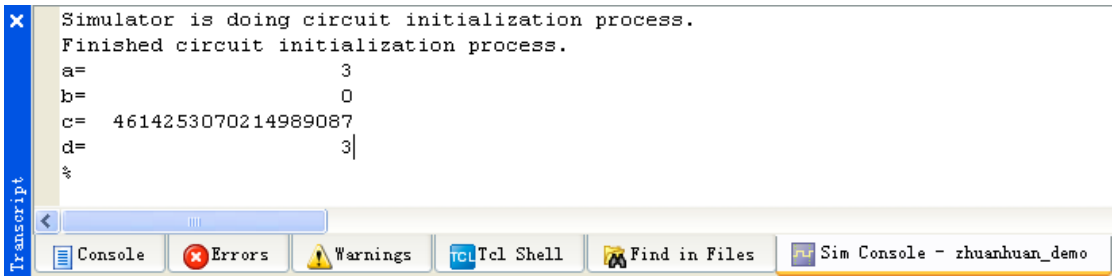


图 7-13　数据类型转换任务仿真结果

7.1.7　概率分布函数

Verilog HDL 语言也提供了随机数的系统函数$random，其语法如下所述：

$random [(seed)]

$random 任务根据种子变量（seed）的取值按 32 位的有符号整数形式返回一个随机数，其中种子变量（必须是寄存器、整数或时间寄存器类型）控制函数的返回值，即不同的种子将产生不同的随机数。如果没有指定种子，每次$random 函数被调用时根据缺省种子产生随机数。$random 任务典型的语法如下所示：，

```
integer Seed, Rnum;
wire clk ;
initialSeed = 12;
always @ (clk)
   Rnum= $random (Seed) ;
```

上述代码在 clk 的每个边沿（包括上升沿和下降沿），$random 被调用并返回一个 32 位有符号整型随机数。如果数字在取值范围内，下述模运算符可产生-10～+10 之间的数字。

```
Rnum = $random(Seed) % 11;
```

如果未显式指定$random 任务的种子，则将随机选取任务种子。例如：

```
Rnum = $random ;
```

这就表明种子变量是可选的，注意数字产生的顺序是伪随机排序的，即对于一个初始种子值产生相同的数字序列。

```
Rnum    = {$random} % 11
```

则产生 0 ～10 之间的一个随机数，其中并置操作符（{ }）将$rando m 函数返回的有符号整数变换为无符号数。

下列函数根据在函数名中指定的概率函数产生伪随机数，其代表着概率分布可通过函数名来鉴别，这里就不再深入讨论。其中所有函数的参数都必须是整数。

```
$dist_uniform (seed, start , end)
$dist_normal (seed , mean , standard_deviation, upper)
$dist_exponential (seed, mean)
$dust_poisson (seed , mean)
$dist_chi_square (seed , degree_of_freedom)
$dist_t (seed, degree_of_freedom)
$dist_erland(seed,k_stage,mean)
```

下面给出一个$random 任务的应用实例。

例 7-11：通过$random 任务生成随机宽度的脉冲序列。

```
`timescale 1ns / 1ps
module random_demo;
```

```
reg dout;

 integer delay1, delay2, num;

 initial begin
     #10    dout    = 0;
      num = 100;
      while (num > 0) begin
          num = num - 1;
            delay1 = {$random} % 10;
            delay2 = {$random} % 20;
       dout = 1;
       #delay1;
       dout = 0;
       #delay2;
      end
end

endmodule
```

上例在 ISE Simulator 中的仿真结果如图 7-14 所示，其有效完成了随机宽度的脉冲信号，达到了设计要求。其中，num 为循环的次数，delay1 为每次循环中高电平持续的时间；delay2 为每次循环中低电平持续的时间。

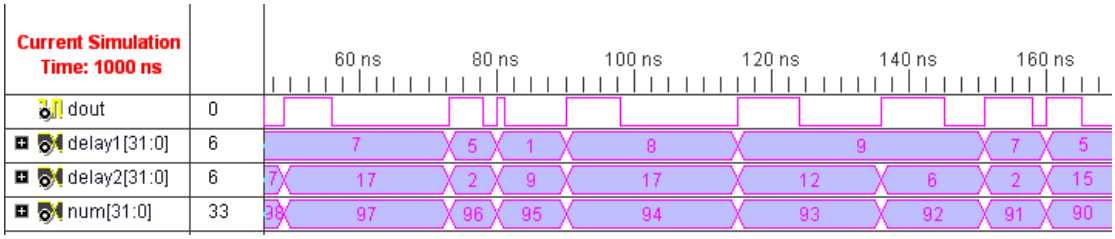


图 7-14　随机位宽脉冲设计仿真结果示意图

7.2 编译预处理语句

编译预处理是 Verilog HDL 编译系统的一个组成部分，指编译系统会对一些特殊命令进行预处理，然后将预处理结果和源程序一起再进行通常的编译处理。以“`”（反引号）开始的某些标识符是编译预处理语句。在 Verilog HDL 语言编译时，特定的编译器指令在整个编译过程中有效（编译过程可跨越多个文件），直到遇到其它的不同编译程序指令。常用编译预处理语句如下：

- `define, `undef
- `ifdef, `else, `endif
- `default_nettype
- `include
- `resetall
- `timescale
- `unconnected_drive, `nounconnected_drive
- `celldefine, `endcelldefine

7.2.1 宏定义`define 语句

1．`define 指令说明

`define 指令是一个宏定义命令，通过一个指定的标志符来代表一个字符串，可以增加Verilog 代码的可读性和可维护性，找出参数或函数不正确或不允许的地方。

`define 指令像 C 语言中的#define 指令，可以在模块的内部或外部定义，编译器在编译过程中，遇到该语句将把宏文本替换为宏的名字。`define 的声明语法格式如下：

`define <macro_name> <Text>

对于已声明的语法，在代码中的应用格式如下所示，不要漏掉宏名称前的“`”。

` macro_name

例如：

```
`define MAX_BUS_SIZE 32
 …
 reg [ `MAX_BUS_SIZE - 1:0 ] AddReg;
```

一旦 `define 指令被编译，其在整个编译过程中都有效。例如，通过另一个文件中的`define 指令 MAX_BUS_SIZE 能被多个文件使用。`undef 指令取消前面定义的宏。例如：

```
`define WORD 16 // 建立一个文本宏替代。
...
wire [ `WORD : 1] Bus;
...
`undef WORD
// 在 `undef 编译指令后，WORD 的宏定义不再有效 7
```

关于宏定义指令，有下面 8 条规则需要注意：

（1）宏定义的名称可以是大写，也可以是小写，但要注意不要和变量名重复。

（2）和所有编译器伪指令一样，宏定义在超过单个文件边界的时仍有效（对工程中的其他源文件），除非被后面的`define `undef 或 `resetall 伪指令覆盖，否则`define 不受范围限制。

（3）当用变量定义宏时，变量可以在宏正文使用，并且在使用宏的时候可，以用实际的变量表达式代替。

（4）通过用反斜杠“\”转义中间换行符，宏定义可以跨越几行，新的行是宏正文的一部分。

（5）宏定义行末不需要添加分号“;”结束。

（6）宏正文不能分离以下的语言记号：注释、数字、字符串、保留的关键字、运算符。

（7）编译器伪指令不允许作为宏的名字。

（8）宏定义中的本文也可以是一个表达式，并不仅用于变量名称替换。

2．`define 和 parameter 的区别

`define 和 parameter 都可以用于完成文本替换的功能，但其存在本质上的不同，前者是编译之前就预处理，而后者是在正常编译过程中完成替换的。此外，`define 和 parameter 存在下列两点不同之处。

（1）作用域不同

parameter 作用于声明的那个文件；`define 从编译器读到这条指令开始到编译结束都有效，或者遇到`undef 命令使之失效，可以应用于整个工程。如果要让 parameter 作用于整个项目，可以将如下声明写于单独文件，并用`include 让每个文件都包含声明文件：

`define 也可以写在代码的任何位置，而 parameter 则必须在应用之前定义。通常编译器都可以定义编译顺序，或者从最底层模块开始编译。因此写在最底层就可以了。

（2）传递功能不同

parameter 可以用作模块例化时的参数传递，实现参数化调用；`define 语句则没有此作用。`define 语句可以定义表达式，而 parameter 只能用于定义变量。

3．`define 开发实例

下面给出一个典型的宏定义开发实例。

例 7-12：宏定义应用实例。

```
module define_demo(clk, a, b, c, d, q);
    `define bsize 9
    `define c   a + b

     input            clk;
     input   [`bsize:0] a, b, c, d;
     output [`bsize:0] q;

     reg      [`bsize:0] q;

     always @(posedge clk) begin
         q <= `c + d;
     end

endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 7-15 所示，其中`c 就意味着左端的加法器，是表达式的宏定义；`bsize 定义了位宽，是变量的定义。

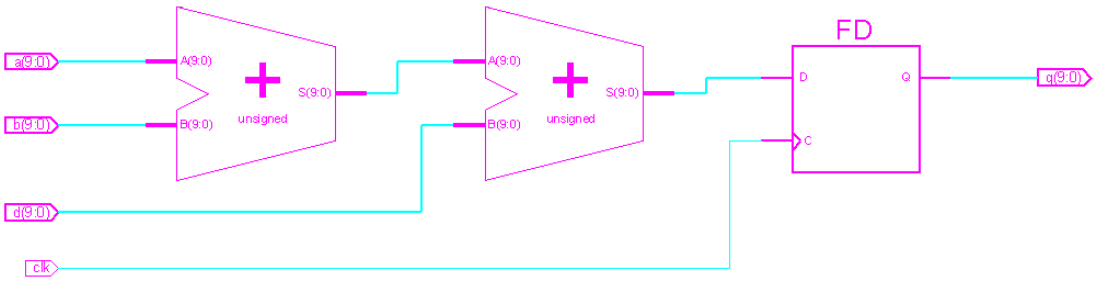


图 7-15　例 7-12 综合后 RTL 结构图

上述程序在 ISE Simulator 中的仿真结果如图 7-16 所示，计算结果正确，从而验证了设计的正确性。

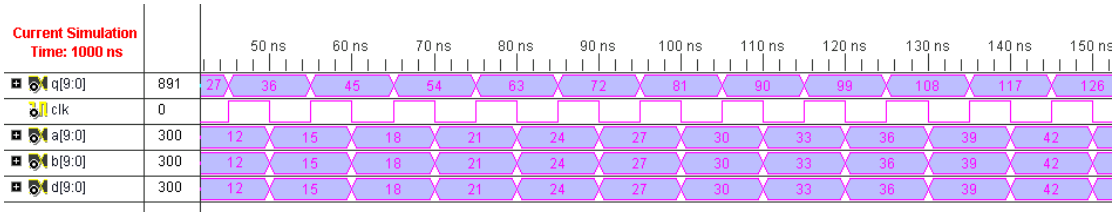


图 7-16　例 7-12 综合结果示意图

7.2.2　条件编译命令`if 语句

条件编译指令包括`ifdef、 `else 和 `endif，其中`ifdef 定义指定的宏是否决定条件编译 Verilog HDL 代码，其应用语法格式有下列两类：

（1）`ifdef MacroName

```
        语句块;
    `endif
（2）`ifdef MacroName
        语句块 1;
    `else
        语句块 2;
    `endif
```

可以看出，`else 程序指令对于 `ifdef 指令是可选的。条件编译语句可以在程序的任何地方调用，其规则如下：

（1）如果宏的名字已经用了`define 定义, 那么只编译 Verilog 代码的第一个块；

（2）如果没有定义宏的名字而且出现`else 伪指令那么只编译第二个块；

（3）这些伪指令可以嵌套；

（4）不被编译的代码都应是有效的 Verilog 代码。

条件编译的简单实例如下：

```
`ifdef WINDOWS
    parameter WORD_SIZE = 16
`else
    parameter WORD_SIZE = 32
endif
```

在编译过程中，如果已定义了名字为 WINDOWS 的文本宏，就选择第一种参数声明，否则选择第二种参数说明。

7.2.3 文件包含`include 语句

`include 编译器指令用于嵌入内嵌文件的内容，可在一个源文件 A 中将另外一个源文件 B 的全部内容包含进来，使得 A 可以使用 B 中的模块，如图 7-17 所示。

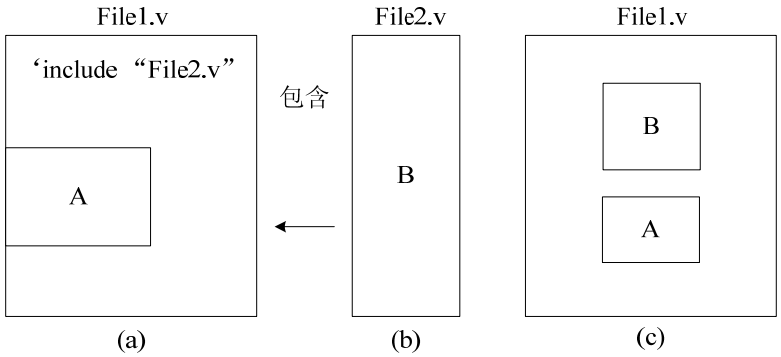


图 7-17 文件包含关系示意图

文件既可以用相对路径名定义，也可以用全路径名定义，例如：

`include " . . / . . /primitives.v"

编译时，这一行由文件 “../../primitives.v” 的内容替代。在实际开发中，`include 命令是很有用的，可以节省设计人员的重复劳动。关于文件说明，有下面 5 点需要注意：

（1）一个`include 指令只能指定一个被包含的文件。如果要完成 N 个文件的包含，则需要调用 N 个`include 指令。

（2）可以将多个`include 指令写在同一行，在`include 命令行只能出现空格和注释。如

下面的写法是合格的。

`include "a.v"　　`include "b.v"

（3）如果文件 A 包含了文件 B 和文件 C，则文件 C 可以直接利用文件 B 的内容，同样文件 B 也可以直接利用文件 C 的内容。

下面给出一个`include 指令的应用实例。

例 7-13：给出一个 Verilog HDL 中`include 语句调用实例

在 D 盘根目录下创建一个.v 文件，并命名为 Onebit_adder，其中包含的内容如下：

```
module Onebit_adder(A,B,Cin,Sum,Cout);
    input A,B,Cin;
    output Sum,Cout;
    wire S1,T1,T2,T3;

    xor X1(S1,A,B);
    xor X2(Sum,S1,Cin);

    and A1(T3,A,B),
        A2(T2,B,Cin),
        A3(T1,A,Cin);

    or o1(Cout,T1,T2,T3);
endmodule
```

然后在任意的目录下，创建一个 Fourbit_adder.v 的文件，作者本人在 F 盘创建了工程，并新建了 Fourbit_adder.v 文件，其内容如下：

```
`include "D:/Onebit_adder.v"
module Fourbit_adder(FA,FB,FCin,FSum,FCout);
    parameter size=4;
    input [size:1]FA,FB;
    output[size:1]FSum;
    input FCin;
    output FCout;
    wire [1:size-1]FTemp;

    Onebit_adder
        FA1(.A(FA[1]),.B(FB[1]),.Cin(FCin),.Sum(FSum[1]),.Cout(FTemp[1])),
       FA2(.A(FA[2]),.B(FB[2]),.Cin(FTemp[1]),.Sum(FSum[2]),.Cout(FTemp[2])),
       FA3(FA[3],FB[3],FTemp[2],FSum[3],FTemp[3]),
       FA4(FA[4],FB[4],FTemp[3],FSum[4],FCout);
endmodule
```

如果注释掉第一句`include "D:/Onebit_adder.v"，综合 Fourbit_adder.v 的程序，ISE 会给出图 7-18 所示的错误提示，表明 Onebit_adder 模块对于 Fourbit_adder 来讲是未知的。

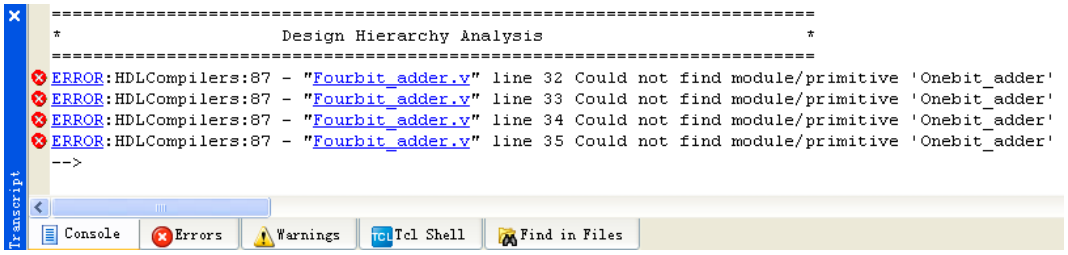


图 7-18　错误提示信息

添加了`include "D:/Onebit_adder.v"后，在编译时，Fourbit_adder.v 中的内容等效为包含了两个模块，如下所列：

```
module Onebit_adder(A,B,Cin,Sum,Cout);
    input A,B,Cin;
    output Sum,Cout;
    wire S1,T1,T2,T3;

    xor X1(S1,A,B);
    xor X2(Sum,S1,Cin);

    and A1(T3,A,B),
        A2(T2,B,Cin),
        A3(T1,A,Cin);

    or o1(Cout,T1,T2,T3);
endmodule

module Fourbit_adder(FA,FB,FCin,FSum,FCout);
    parameter size=4;
    input [size:1]FA,FB;
    output[size:1]FSum;
    input FCin;
    output FCout;
    wire [1:size-1]FTemp;

    Onebit_adder
        FA1(.A(FA[1]),.B(FB[1]),.Cin(FCin),.Sum(FSum[1]),.Cout(FTemp[1])),
       FA2(.A(FA[2]),.B(FB[2]),.Cin(FTemp[1]),.Sum(FSum[2]),.Cout(FTemp[2])),
       FA3(FA[3],FB[3],FTemp[2],FSum[3],FTemp[3]),
       FA4(FA[4],FB[4],FTemp[3],FSum[4],FCout);
endmodule
```

这样，Onebit_adder 对 Fourbit_adder 模块可见，可作为后者的子模块，完成层次化调用。Fourbit_adder.v 在 ISE 中可以正确综合，其仿真结果如图 7-19 所示。可以看出，Fourbit_adder 正确实现了一个两输入的 4 比特加法器。

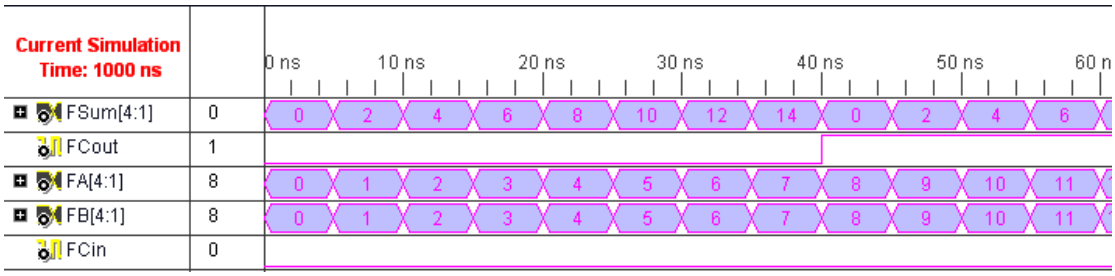


图 7-19　4 比特加法器仿真结果示意图

7.2.4　时间尺度`timescale 语句

在 Verilog HDL 模型中，所有时延都用单位时间表述。使用 `timescale 编译器指令将时间单位与实际时间相关联。该指令用于定义时延的单位和时延精度。 `timescale 编译器指令格式为：

`timescale time_unit / time_precision

其中 time_unit 是一个用于测量的单位时间。time_precision 决定延时应该达到的精度，为仿真设置单位步距。time_unit 和 time_precision 由值 1、10、和 100 以及单位 s、 ms、us、ns、ps 和 fs 组成。例如：

`timescale 1ns/100ps

表示时延单位为 1ns, 时延精度为 100ps。`timescale 编译器指令在模块说明外部出现，并且影响后面所有的时延值。例如 ：

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
    output Z;
    input A, B;

    and # (5.22, 6.17 ) Al (Z, A, B);
    // 规定了上升及下降时延值。
endmodule
```

编译器指令定义时延以 ns 为单位，并且时延精度为 1/10ns（100ps）。因此，时延值 5.22 对应 5.2ns，时延 6.17 对应 6.2ns。如果用如下的 `timescale 程序指令代替上例中的编译器指令：

`timescale 10ns/1ns

那么 5.22 对应 52ns，6.17 对应 62ns。在编译过程中，`timescale 指令影响这一编译器指令后面所有模块中的时延值，直至遇到另一个`timescale 指令或`resetall 指令。当一个设计中的多个模块带有自身的`timescale 编译指令时将发生什么？在这种情况下，模拟器总是定位在所有模块的最小时延精度上，并且所有时延都相应地换算为最小时延精度。例如，

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
    output Z;
    input A, B;

    and # (5.22, 6.17 ) Al (Z, A, B);
endmodule
```

```
`timescale 10ns/ 1ns
module TB;
    reg PutA, PutB;
    wire GetO;

    initial
      begin
      PutA = 0;
      PutB = 0;
      #5.21 PutB = 1;
      #10.4 PutA = 1;
      #15 PutB = 0;
   end
   AndFunc AF1(GetO, PutA, PutB);
endmodule
```

在这个例子中，每个模块都有自身的`timescale 编译器指令。`timescale 编译器指令第一次应用于时延。因此，在第一个模块中，5.22 对应 5.2ns，6.17 对应 6.2ns；在第二个模块中 5.21 对应 52ns，10.4 对应 104ns，15 对应 150ns。如果仿真模块 TB，设计中的所有模块最小时间精度 100ps。因此，所有延迟（特别是模块 TB 中的延迟）将换算成精度为 100ps。延迟 52 ns 现在对应 520*100ps，104 对应 1040*100ps，150 对应 1500*100ps。更重要的是，仿真使用 100ps 为时间精度。如果仿真模块 AndFunc，由于模块 TB 不是模块 AddFunc 的子模块，模块 TB 中的`timescale 程序指令将不再有效。

7.2.5 其他语句

除上述常用的编译预处理语句外，Verilog HDL 语言还包括下列预处理语句，由于其应用范围并不广泛，因此只对其进行简单介绍。

1．`default_nettype 语句

`default_nettype 用于为隐式线网指定线网类型。也就是将那些没有被说明的连线定义线网类型。

`default_nettype wand

该实例定义的缺省的线网为线与类型。因此，如果在此指令后面的任何模块中没有说明的连线，那么该线网被假定为线与类型。

2．`resetall 语句

`resetall 编译器指令将所有的编译指令重新设置为缺省值。

例如：

`resetall

该指令使得缺省连线类型为线网类型。

3．`unconnected_drive 语句

`unconnected_drive 和 `nounconnected_drive 在模块实例化中，出现在这两个编译器指令间的任何未连接的输入端口或者为正偏电路状态或者为反偏电路状态。

```
`unconnected_drive pull1
...
/* 在这两个程序指令间的所有未连接的输入端口为正偏电路状态（连接到高电平）*/
`nounconnected_drive
```

```
`unconnected_drive pull0
...
/* 在这两个程序指令间的所有未连接的输入端口为反偏电路状态（连接到低电平） */
nounconnected_drive
```

4．`celldefine 语句

`celldefine 和 `endcelldefine 这两个程序指令用于将模块标记为单元模块。它们表示包含模块定义，如下例所示。

```
`celldefine
module FD1S3AX (D, CK, Z) ;
    ...
endmodule
`endcelldefine
```

7.3 本章小结

本章主要介绍了 Verilog HDL 语句中最常用的一些系统任务和编译预处理语句，这些语句是非常重要的，是编写调试模块所必需的。读者需要重点掌握的有文件的输入输出任务，这是大规模验证中最快速的数据读取方法。其中，在多文件系统中，$monitor 需要配合$monitoron 和$monitoroff 使用。对于这些系统级的任务和处理语句，需要读者明白其语句含义、使用场合以及在程序中的使用位置，将其和测试代码的书写联系起来，了解有关语法的实质，只能在设计中灵活运用，达到事半功倍的效果。

7.4 思考题

1．什么是系统任务？其有什么特征？
2．输出显示任务包括哪些系统任务？简要描述输出显示任务的使用方法。
3．利用 Verilog HDL 语言完成文件读取，假设在文件中存储了 1~65535 这 65535 个数据。
4．利用概率分布函数实现了一个输出为 1~100 之间的随机数发生器。
5．什么是编译预处理任务？
6．简要说明宏定义 define 语句的使用方法？
7．给出一个`timescale 语句的应用实例，说明其对仿真结果的影响。

第 8 章　Verilog HDL 可综合设计的难点解析

通过前面各章的学习可知，Verilog HDL 语言分为面向综合和面向仿真的两大类语句，且可综合语句远少于仿真语句，读者可能会有可综合设计相对简单的感觉。然而事实刚好与此相反，这是因为：首先，可综合设计是用来构建硬件平台的，因此对设计的指标要求很高，包括资源、频率和功耗，这都需要通过代码来体现；其次，在实际开发中要利用基本 Verilog HDL 语句完成种类繁多的硬件开发，给设计人员带来了很大的挑战。所有的仿真语句只是为了可综合设计的验证而存在。为了让读者深入地理解可综合设计、灵活运用已学内容，本章将可可综合设计中的基本知识点和难点提取出来，融入 Verilog HDL 语法以及开发工具等诸多方面，以深入浅出的方式向读者说明设计中的难点本质。

8.1 组合逻辑和时序逻辑

数字电路根据逻辑功能的不同特点，可以分成两大类，一类叫组合逻辑电路（简称组合电路），另一类叫做时序逻辑电路（简称时序电路）。掌握组合逻辑和时序逻辑的区分手段与实现方法是数字系统设计的基本要求。

8.1.1　组合逻辑设计

1．组合逻辑概念

组合逻辑是 Verilog HDL 设计中的一个重要组成部分。从电路本质上讲，组合逻辑电路的特点是输出信号只是当前时刻输入信号的函数，与其它时刻的输入状态无关。无存储电路，也没有反馈电路，其典型结构如图 8-1 所示。从电路行为上看，其特征就是输出信号的变化仅仅与输入信号的电平有关，不涉及对信号跳变沿的处理。

图 8-1　组合逻辑结构示意图

尽管组合电路在功能上千差万别，可是其分析方法却有很多相似之处。给定逻辑电路后，得到其输入与输出的直接表达式，将输入组合全部带入表达式中计算得到输出结果，并以真值表的形式表达出来，最后根据真值表说明电路功能。

组合逻辑电路的设计就是在给定逻辑功能的前提下，通过某种设计渠道，得到满足功能要求、且最简单的逻辑电路。基于 HDL 语言和 EDA 工具的组合逻辑电路的设计流程如图 8-2 所示。

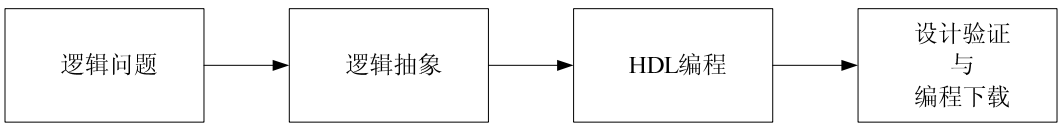


图 8-2　组合逻辑结构示意图

其中逻辑抽象和 HDL 编程是重点环节。在很多情况下，逻辑问题都是通过文字描述的，逻辑抽象就是对设计对象的输入与输出信号间的因果关系，用逻辑函数的方法表示出来。HDL 语言编程就是直接通过语句来实现抽象结果。

2．组合逻辑的 Verilog HDL 描述

根据组合逻辑的电路行为，可以得到两种常用的 RTL 级描述方式。第一种是 always 模块的触发事件为电平敏感信号列表；第二种就是用 assign 关键字描述的数据流赋值语句。

（1）always 模块的敏感表为电平敏感信号的电路

这种方式的组合电路应用非常广泛，几乎可以完成对所有组合逻辑电路的建模。always 模块的敏感列表为所有判断条件信号和输入信号，但一定要注意敏感列表的完整性。在 always 模块中可以使用 if、case 和 for 等各种 RTL 关键字结构。由于赋值语句有阻塞赋值和非阻塞赋值两类，建议读者使用阻塞赋值语句“=”，详细原因将在 8.3.1 节进行说明。

always 模块中的信号必须定义为 reg 型，不过最终的实现结果中并没有寄存器。这是由于在组合逻辑电路描述中，将信号定义为 reg 型，只是为了满足语法要求。下面给出一个组合逻辑实例。

例 8-1：通过 Verilog HDL 语言实现一个两输入比较器，输入分别为 d1、d2，输出分别为 f1（d1>d2 时为高电平）、f2（d1=d2 时为高电平）、f3（d1<d2 时为高电平）。

```
module compare_demo(
    d1, d2, f1, f2, f3
    );

     input [7:0] d1, d2;
     output      f1, f2, f3;
     reg         f1, f2, f3;

     always @(d1, d2) begin
         if (d1 > d2)
             f1 = 1;
        else
             f1 = 0;

         if (d1 == d2)
             f2 = 1;
         else
             f2 = 0;

         if (d1 < d2)
             f3 = 1;
         else
             f3 = 0;
    end

endmodule
```

可以看出，组合逻辑就对应着电平触发事件电路。上述程序在 ISE 中综合后的 RTL 级结构图如图 8-3 所示，可以看出，虽然将输出信号 f1、f2 以及 f3 声明为寄存器变量，并且

在 always 模块中进行赋值操作，但在组合逻辑设计中，并没有综合成 D 触发器。

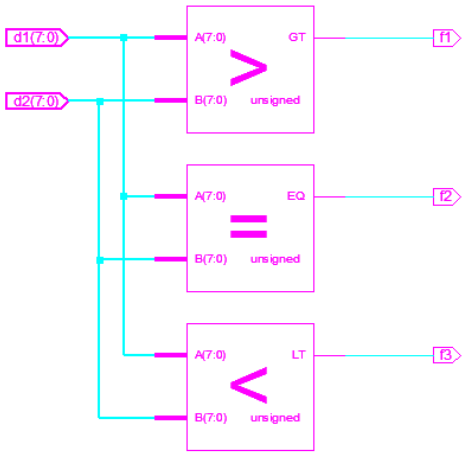


图 8-3　比较器 RTL 结构示意图

上述程序在 ISE Simulator 中的仿真结果如图 8-4 所示，只要敏感信号电平发生变化，always 语句块中所有语句都会被重新执行一次。

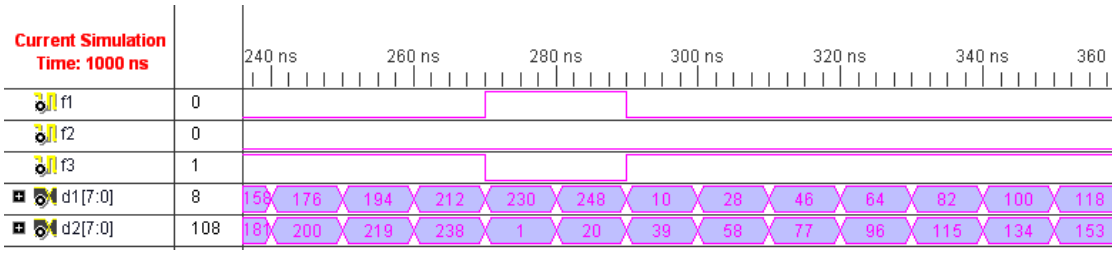


图 8-4　比较器组合逻辑的仿真结果示意图

（2）assign 语句描述的电路

利用条件符“?”可以描述一些相对简单的组合逻辑电路，信号只能被定义为 wire 型。当组合逻辑比较复杂时，需要很多条语句 assign 语句或者多重嵌套“？”，使得代码可读性极差，因此此时推荐第一种组合逻辑建模方式。下面给出一个由 assign 关键字描述的组合逻辑实例。

例 8-2：通过 assign 语句实现例 8-1 的比较器。

```
module compare_demo2(
    d1, d2, f1, f2, f3
    );

     input [7:0] d1, d2;
     output        f1, f2, f3;

     assign f1 = (d1 > d2) ? 1 : 0;
     assign f2 = (d1 ==d2) ? 1 : 0;
     assign f3 = (d1 < d2) ? 1 : 0;

endmodule
```

在 ISE 中查看其综合后的 RTL 级结构示意图，可以发现和图 8-3 一样，其仿真结果也和例 8-1 的一致。

3．组合逻辑电路的注意事项

（1）敏感信号列表

在组合逻辑设计中，读者必须重点对待敏感信号列表。敏感信号列表出现在 always 块中，其典型行为级的含义为：只要敏感信号列表内的信号发生电平变化，则 always 模块中的语句就执行一次，因此设计人员必须将所有的输入信号和条件判断信号都列在信号列表中。有时，不完整的信号列表会造成不同的仿真和综合结果，因此需要保证敏感信号的完备性。在实际的 PLD 器件开发中，EDA 工具都会默认将所有的输入信号和条件判断语句作为触发信号，增减敏感信号列表中的信号不会对最终的执行结果产生影响，因此读者如果期望在设计中通过修改敏感信号来得到不同的逻辑，那就大错特错了。当敏感信号不完备时，会使得仿真结果不一样，这是因为仿真器在工作时不会自动补充敏感信号表。如果缺少信号，则无法触发和该信号相关的仿真进程，从而得不到正确的仿真结果。

因此，为了确保仿真和最终实现结果一致，必须要保证组合逻辑电路 always 敏感信号列表的完备性。如果设计人员在设计中，认为列举信号麻烦，则采用下面的语句：

```
always @(*) begin
    …
end
```

此时，综合工具和仿真工具会自动将所有的敏感信号自动加入敏感信号列表。ISE 也支持这一用法。

（2）不要在组合逻辑中引入环路

在组合逻辑中引入环路会导致电路产生振荡、毛刺以及冲突等问题，从而降低设计的稳定性和可靠性，因此要彻底避免环路。

图 8-5 给出一个简单的环路设计，把一个寄存器输出通过组合逻辑后，再次通过两级组合逻辑处理反馈给该组合逻辑的引脚时，就会产生组合环路，要避免该组合回路可以采用图 8-6 所示的逻辑设计示意图，不仅功能结构一致，还取消了组合逻辑环路。

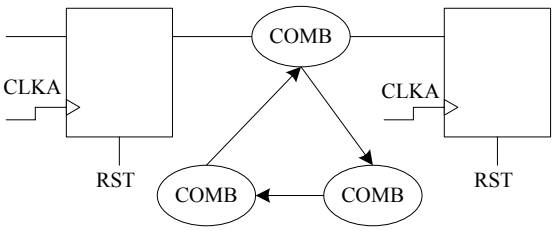


图 8-5　组合逻辑中的环路示意图

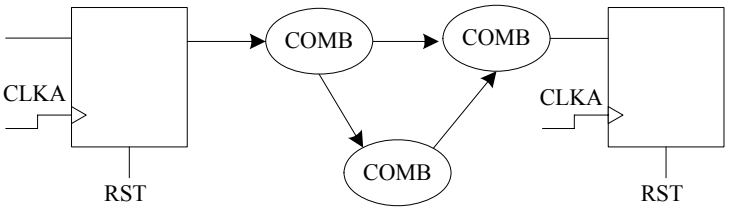


图 8-6　组合逻辑中无环路的示意图

之所以称逻辑环路是一种高风险设计，其原因如下：

- 首先，环回逻辑的延时完全依靠组合逻辑门延迟和布线延迟。一旦这些传播时延有所变化，则环路的整体逻辑将彻底失效。
- 其次，环路的时序分析是个死循环过程。目前的 EDA 开发工具为了计算环路的时序逻辑都会主动割断时序路径，引入许多不确定的因素。

目前的综合工具都会给出逻辑环路的警告（Combinational Loops），因此设计人员必须

对软件工具的此类报告特别在意。如果一定要实现环路，则需要通过时序逻辑的寄存器来完成。

8.1.2 时序逻辑设计

1．时序逻辑电路的基本知识

时序逻辑是 Verilog HDL 设计中另一类重要应用。从电路特征上看来，其特点为任意时刻的输出不仅取决于该时刻的输入，而且还和电路原来的状态有关。电路里面有存储元件（各类触发器，在 FPGA 芯片结构中只有 D 触发器）来记忆信息，如图 8-7 所示。从电路行为上讲，不管输入如何变化，仅当时钟的沿（上升沿或下降沿）到达时，才有可能使输出发生变化。

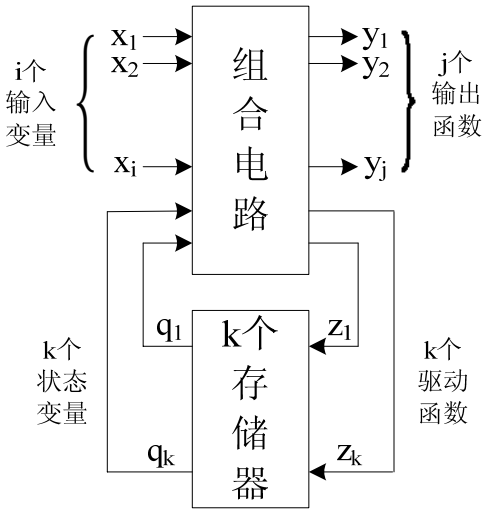


图 8-7　时序电路结构图

从图 8-7 可以看出，时序逻辑电路由组合逻辑电路和存储电路这两部分组成，其中存储电路由各类触发器（JK 触发器、D 触发器以及 T 触发器等类型）构成，并将组合逻辑的部分输出反馈到输入逻辑的输入端口。

时序电路可通过表达式（电路输出端的输出逻辑表达式、存储电路触发器输出端的驱动或激励表达式，以及表示触发器状态的状态方程）、状态转移表、状态转移图、时序图以及 HDL 语言行为描述等多种描述方法。若将输入变量和各级触发器状态的全部组合列出，分别代入各级触发器的状态方程和电路的输出方程，则可以计算出各级触发器的次态值和当前输出值，把相应的计算结果列成真值表就可得到状态转移表。对于读者最关心的 HDL 行为描述代码，可在时序图的基础上快速得到。

分析一个时序电路，就是要找出给定时序电路的逻辑功能。具体地说，就是要求找出电路的状态和输出状态（一般指进位输出、借位输出等）在输入变量和时钟信号作用下的变化规律。为了直观地说明上述方法，下面给出一个简单的时序逻辑电路，通过分析得到其常用的描述形式。

例 8-3：完成图 8-8 所示的简单时序逻辑电路的分析。

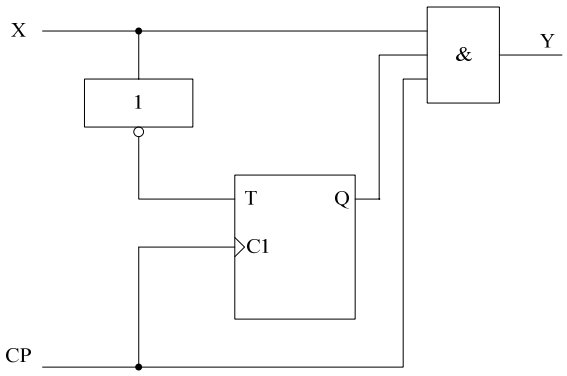


图 8-8　简单时序电路的示意图

- 首先，其输出方程、驱动方程分别如下所列：

$$Y = XQ^nCP \tag{8-1}$$

$$T = \overline{X} \tag{8-2}$$

由于电路采用 T 型触发器，因此其特征方程为：

$$Q^{n+1} = T \cdot \overline{Q}^n + \overline{T} \cdot Q^n \tag{8-3}$$

将驱动方程代入特征方程，可以得到式(8-4)所示的状态方程。

$$Q^{n+1} = \overline{X} \cdot \overline{Q}^n + X \cdot Q^n \tag{8-4}$$

- 计算并列出状态转移表

图示电路有一个输入 X 和 1 级触发器，因此输入与触发器初态的取值组合只有 4 组，即 00、01、10 和 11。把这些取值带入式(8-4)所示的状态方程组和式(8-1)所示的输出方程，可计算出触发器的次态和电路的输出值，其相应的状态转移表如表 8-1 所列。

表 8-1　　例 8-3 的状态转移表

X	Q^n	Q^{n+1}	Y
0	0	1	0
0	1	0	0
1	0	0	0
1	1	1	1

- 画出状态转移图和时序图

状态转移图直观、形象地显示出时序逻辑电路的特点和逻辑功能，本例的状态转移图如图 8-9 所示。其中，圆圈内的数字表示电路的状态，箭头表示状态转换的方向，箭头旁注明了状态转换的输入条件和输出结果，输入条件为斜线上方，而输出结果位于斜线下方。

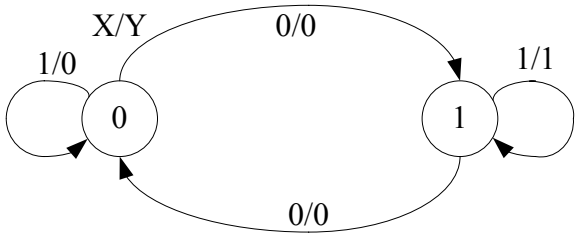
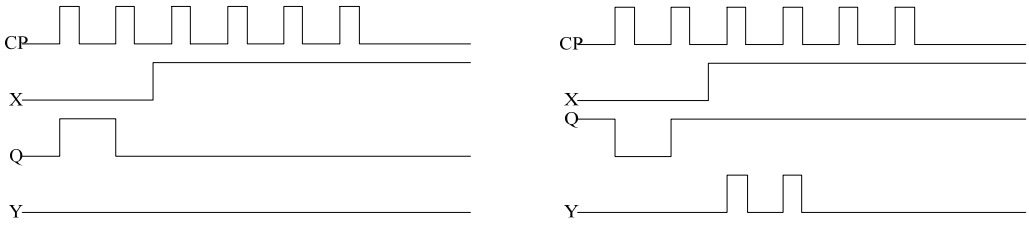


图 8-9　例 8-3 的状态转移图

时序图就是通过数字信号波形直观表示时序逻辑电路的特点和逻辑功能，可根据状态方

程、状态转移表等多方面得到，用于判断设计结果的正确性。图 8-10（a）、（b）图分别给出触发器初始状态为 0 和 1 的时序图。



（a）触发器初始状态为 0 的时序图　　（b）触发器初始状态为 1 的时序图

图 8-10　例 8-3 的时序图

从中可以看出以下两点：首先，时序电路的输出信号不仅取决于电路当时的输入，还取决于电路原来的状态，体现了“记忆”特性。其次，在同步时序逻辑电路中，触发器由时钟信号 CP 来触发，控制其翻转时刻，而对触发器翻转到何种状态并无影响。

2．时序逻辑的 Verilog HDL 描述

时序电路的行为决定了其只能通过 always 块语句实现，通过关键词“posedge”和“negedge”来捕获时钟信号的上升沿和下降沿。在 always 语句块中可以使用任何可综合的标志符。下面首先以 D 触发器为例，给出基本单元触发器的 Verilog HDL 实例，读者可自行完成其余常用触发器（RS 触发器、JK 触发器以及 T 触发器等）的 Verilog HDL 实现。

例 8-4：通过 Verilog HDL 实现 D 触发器。

同步 D 触发器的功能为：D 输入只能在时序信号 clk 的沿变化时才能被写入到存储器中，替换以前的值，常用于数据延迟以及数据存储模块中。由于 D 触发器只有一个输入端，在许多情况下，可使触发器之间的连接变得非常简单，因此使用十分广泛。

```
module sy_d_ff(clk, d, q, qb);
    input clk, d;
    output q, qb;
    reg q;

    assign qb = ~q;

    //实现 D 触发器
    always @(posedge clk) begin
        q <= d;
    end

endmodule
```

上述程序在 ISE 中综合后的 RTL 级结构如图 8-11 所示。

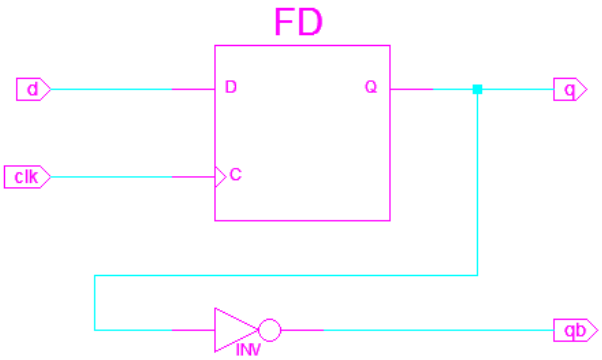


图 8-11　同步 D 触发器的 RTL 结构图

上述程序的仿真结果如图 8-12 所示。从中可以看出，在时钟上升沿，D 触发器都将输入数据接收并寄存。

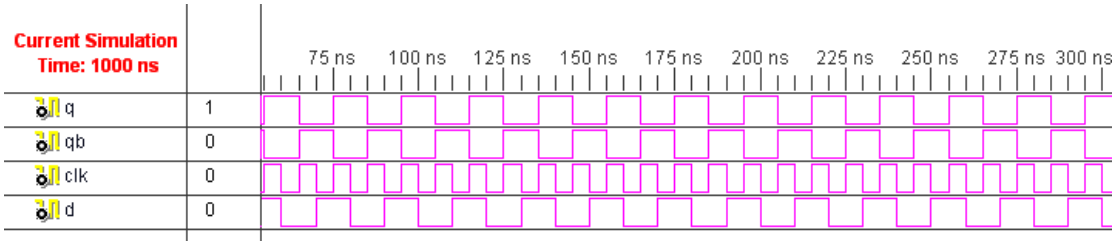


图 8-12　同步 D 触发器的仿真结果示意图

在给出时序逻辑设计最基本的电路后，下面给出图 8-8 所示电路的 Verilog HDL 实现，和例 8-3 的描述方法进行比较，。

例 8-5：通过 Verilog HDL 语言实现例 8-3 所示电路。

```
module tl_demo(
    clk,   x, y
    );

     input    clk,   x;
     output   y;

     reg       x_d;
     reg       d_q;

     always @(posedge clk) begin
           x_d <= ~x;
     end

     always @(posedge clk) begin
           d_q <= x;
     end

     assign   y = d_q && x && clk;

endmodule
```

程序在 ISE 综合后的 RTL 结构图如图 8-13 所示，可以看出，其和图 8-8 是一致的，达

到了设计的目的。

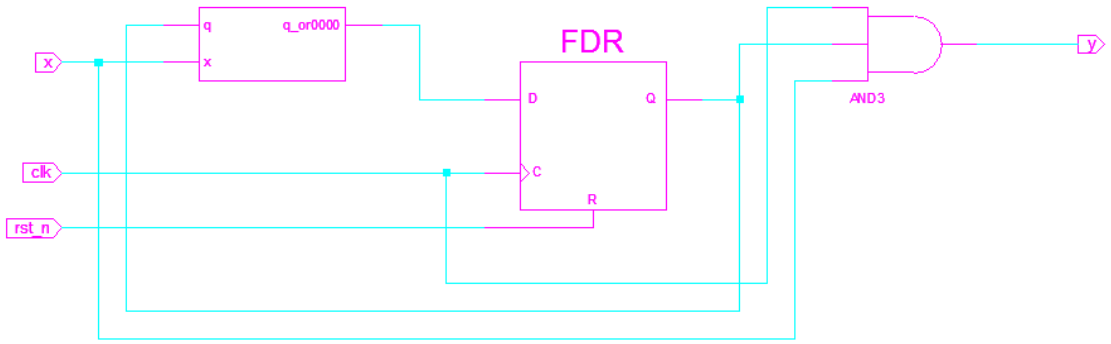


图 8-13　例 8-5RTL 结构示意图

上述程序的仿真结果如图 8-14 所示，验证了程序的正确性。

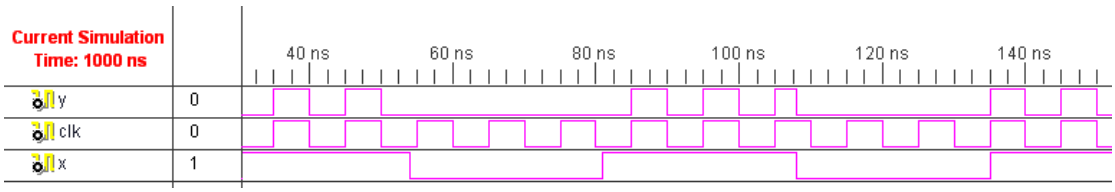


图 8-14　例 8-5 的仿真结果示意图

在利用 Verilog HDL 描述时序电路时有下面几个问题需要注意：

（1）在描述时序电路的 always 块中的 reg 型信号都会被综合成寄存器，这是和组合逻辑电路所不同的。

（2）时序逻辑中推荐使用非阻塞赋值“<=”，原因将在 8.3 节详细说明。

（3）时序逻辑的敏感信号列表只需要加入所用的时钟触发沿即可，其余所有的输入和条件判断信号都不用加入，这是因为时序逻辑是通过时钟信号的跳变沿来控制的。

8.1.3　组合逻辑电路中的竞争与冒险

1．什么是竞争与冒险

信号在组合逻辑电路内部通过连线和逻辑单元时，都有一定的延时。延时的大小与连线的长短和逻辑单元的数目有关，同时还受器件的制造工艺、工作电压、温度等条件的影响。此外，信号的高低电平转换也需要一定的过渡时间。由于存在这两方面因素，多路信号的电平值发生变化时，在信号变化的瞬间，组合逻辑的输出有先后顺序，并不是同时变化，往往会出现一些不正确的尖峰信号，这些尖峰信号称为“毛刺”，如图 8-15 所示。如果一个组合逻辑电路中有“毛刺”出现，就说明该电路存在“冒险”。

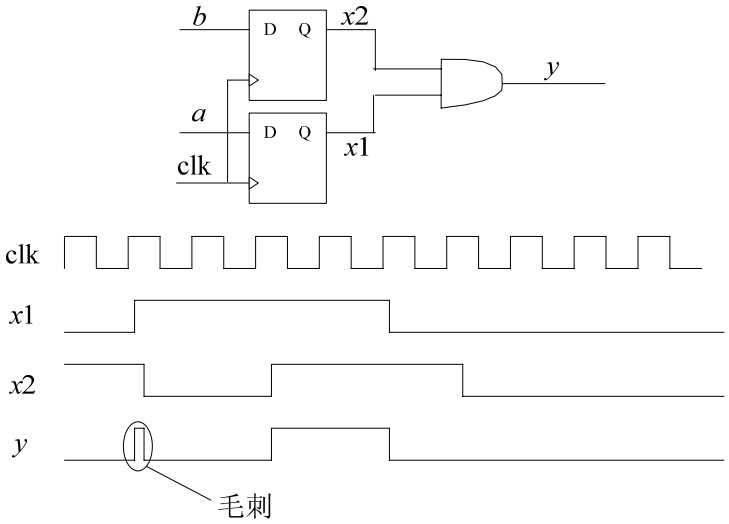


图 8-15　冒险毛刺示意图

需要注意的是，冒险信号的脉冲宽度很小，常常只有数纳秒或数十纳秒，其频带带宽可达数百兆赫兹或更宽。在板级调试时，如果示波器的上限频率较低，会将幅度较大的毛刺显示为幅度较小的毛刺，甚至不易被察觉。这都是在实际开发中捕获毛刺应该注意的问题。

冒险按产生形式的不同可以分为静态冒险和动态冒险两大类。静态冒险是指输入有变化，而输出不应变化时产生的单个窄脉冲；动态冒险则指的是输入变化时，输出也应变化时产生的冒险。文献[4]指出，动态冒险是由静态冒险引起的，因此存在动态冒险的电路也存在静态冒险。

静态冒险根据产生条件的不同，分为功能冒险和逻辑冒险两大类。当有两个或两个以上输入信号同时产生变化时，在输出端产生毛刺，这种冒险称为功能冒险。如果只有一个变量产生变化时出现的冒险则是逻辑冒险。

冒险往往会影响到逻辑电路的稳定性。清零和置位端口对毛刺信号十分敏感，任何一点毛刺都可能会使系统出错，因此判断逻辑电路中是否存在冒险以及如何避免冒险是设计人员必须要考虑的问题。

2．冒险产生的原因

由于动态冒险主要由静态冒险引起的，消除了静态冒险，动态冒险也就自然消除，因此下面介绍静态冒险的检查和消除。

判断一个逻辑电路在某些输入信号发生变化时是否会产生冒险，首先要判断信号是否会同时变化，然后判断在信号同时变化的时候，是否会产生冒险，这可以通过逻辑函数的卡诺图或逻辑函数表达式来进行判断。

（1）功能冒险的检查

功能冒险是由电路的逻辑功能引起的，只要输入信号不是按照循环码的规律变化，组合逻辑就可能产生功能冒险，且不能通过修改设计加以消除，只能通过对输出采用时钟采样来消除。

（2）逻辑冒险的检查

检查电路是否产生逻辑冒险的方法有两种：代数法和卡诺图法。

- 代数法

如果一个组合逻辑函数表达式 F，在某些条件下能化简成 $F = A + \overline{A}$ 或 $F = A\overline{A}$ 的形式，在 A 产生变化时，就可能产生静态逻辑冒险。

- 卡诺图法

在组合逻辑的卡诺图中，若存在素项圈相切，则可能会产生逻辑冒险。如图 8-16 所示的卡诺图，AC 和 $\bar{A}B$ 两个素项环相切，在 $B=C=1$ 时，A 由 1 变为 0 时，将产生逻辑冒险。

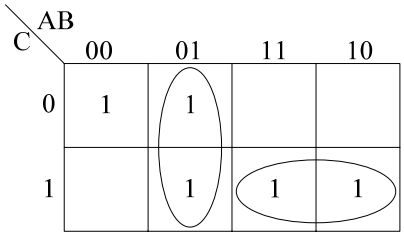


图 8-16　产生冒险的组合逻辑的卡诺图示意图

3．竞争与冒险在 Verilog HDL 设计中的体现

例 8-6：详细分析 Verilog HDL 程序内在的逻辑冒险现象。

首先，给出一段示例代码，如下所列：

```
module maoxian(
      A, B, C, D, Out
      );
       input    A, B, C, D;
       output Out;

       wire       aandb, candd;
       assign    aandb = A && B;
       assign    candd = C && D;
       assign    Out     = aandb || candd;

endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 8-17 所示。

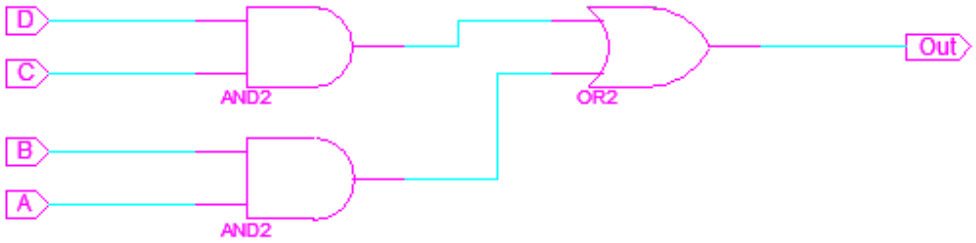


图 8-17　综合结果示意图

上述程序在 ISE Simulator 中的仿真结果如图 8-18 所示。从图 8-18 的仿真波形可以看出，由于“A、B、C、D”四个输入信号高低电平变换不是同时发生的，这导致输出信号“Out”出现了毛刺。由于无法保证所有连线的长度一致，所以即使四个输入信号在输入端同时变化，但经过可编程逻辑器件内部的走线，到达或门的时间也是不一样的，毛刺必然产生。简单地说，只要输入信号不是同时变化，组合逻辑必将产生毛刺。因而，将它们的输出直接连接到时钟输入端、清零或置位端口的设计方法是错误的，这可能会导致严重的后果。所以我们必须检查设计中所有时钟、清零和置位等对毛刺敏感的输入端口，确保输入不会含有任何毛刺。

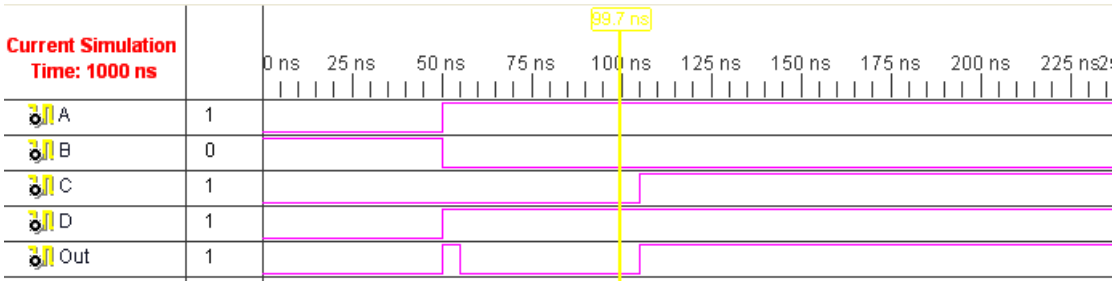


图 8-18　逻辑电路的毛刺示意图

4．毛刺的消除

毛刺并不是对所有的输入都有危害，例如 D 触发器的 D 输入端，只要毛刺不出现在时钟的上升沿并且满足数据的建立和保持时间，就不会对系统造成危害，我们可以说 D 触发器的 D 输入端对毛刺不敏感。根据这个特性，我们应当在系统中尽可能采用同步电路，这是因为同步电路信号的变化都发生在时钟沿，只要毛刺不出现在时钟的沿口并且不满足数据的建立和保持时间，就不会对系统造成危害。（由于毛刺很短，多为几纳秒，基本上都不可能满足数据的建立和保持时间）

因此我们可以通过改变设计，破坏毛刺产生的条件，来减少毛刺的发生。例如，在数字电路设计中，常常采用格雷码计数器取代普通的二进制计数器，这是因为格雷码计数器的输出每次只有一位跳变，消除了竞争冒险的发生条件，避免了毛刺的产生。目前，主要有两种基本的采样方法：脉冲选择法和时序逻辑保持法。

（1）脉冲选择法

该方法在输出信号的保持时间内，用一定宽度的高电平脉冲与输出信号做逻辑“与”运算，由此获取输出信号的电平值。例 8-7 说明了这种方法，采样脉冲信号从输入引脚“Sample”引入。

例 8-7：通过脉冲选择法修改例 8-6 电路中的毛刺。

```
module maoxian(
    A, B, C, D, Sample, Out, Tout
    );
      input   A, B, C, D, Sample;
      output Out, Tout;

      wire     aandb, candd, tmp;
      assign   aandb = A && B;
      assign   candd = C && D;
      assign   tmp    = aandb || candd;
      assign   Out    = Sample && tmp;
      assign   Tout   = tmp;
endmodule
```

程序的综合结果如图 8-19 所示，其中“Tout_bmp”模块封装了除脉冲选择以外的全部电路，脉冲选择通过与门来实现。

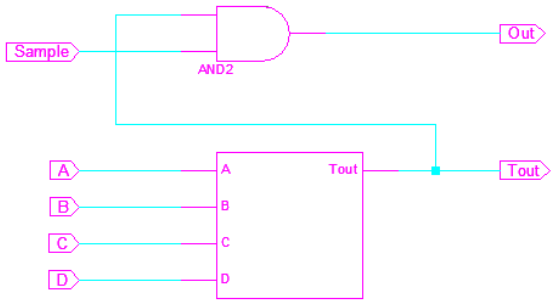


图 8-19　综合结果示意图

程序的仿真结果如图 8-20 所示，从其仿真波形上可以看出，毛刺信号出现在“Tout”引脚上，而“Out”引脚上的毛刺已被消除了。

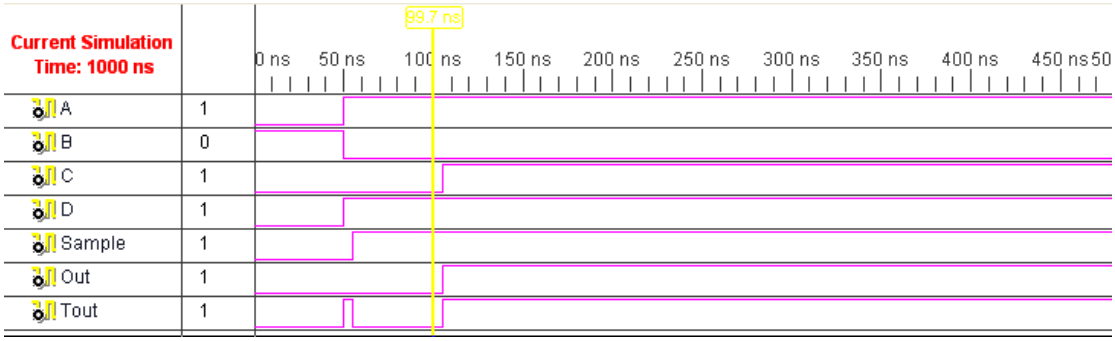


图 8-20　例 8-7 的仿真结果示意图

以上方法可以大大减少毛刺，但它并不能完全消除毛刺，有时，我们必须手工修改电路来去除毛刺。我们通常使用“采样”的方法。一般说来，冒险出现在信号发生电平转换的时刻，也就是说在输出信号的建立时间内会发生冒险，而在输出信号的保持时间内是不会有毛刺信号出现的。如果在输出信号的保持时间内对其进行“采样”，就可以消除毛刺信号的影响。

（2）时序逻辑保持法

脉冲选择法的一个缺点是必须人为地保证 sample 信号必须在合适的时间中产生，另一种更常见的方法即为时序逻辑保持法，其利用 D 触发器的 D 输入端对毛刺信号不敏感的特点，在输出信号的保持时间内，用触发器读取组合逻辑的输出信号，这种方法类似于将异步电路转化为同步电路。下面给出一个应用实例。

例 8-8：通过 D 触发器来消除例 8-6 所示电路的冒险现象。

```
module maoxian(
    clk, A, B, C, D, Out, Tout
    );
     input    clk;
     input    A, B, C, D;
     output Out, Tout;

     reg       Out;
     wire       aandb, candd, tmp;
     assign    aandb = A && B;
     assign    candd = C && D;
     assign    tmp      = aandb || candd;
```

```
    always @(posedge clk) begin
        Out <= tmp;
    end

    assign   Tout   = tmp;
endmodule
```

上述程序综合后的 RTL 级结构如图 8-21 所示，同样其中“Tout_bmp”模块封装了除脉冲选择以外的全部电路，时序逻辑保持通过 D 触发器来实现。

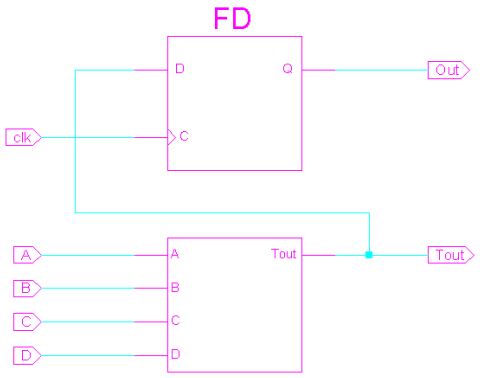


图 8-21　例 8-8 代码的 RTL 结构示意图

程序的仿真结果如图 8-22 所示，从其仿真波形上可以看出，毛刺信号出现在“Tout”引脚上，而“Out”引脚上的毛刺已被消除了，无需外部输入采样控制信号。

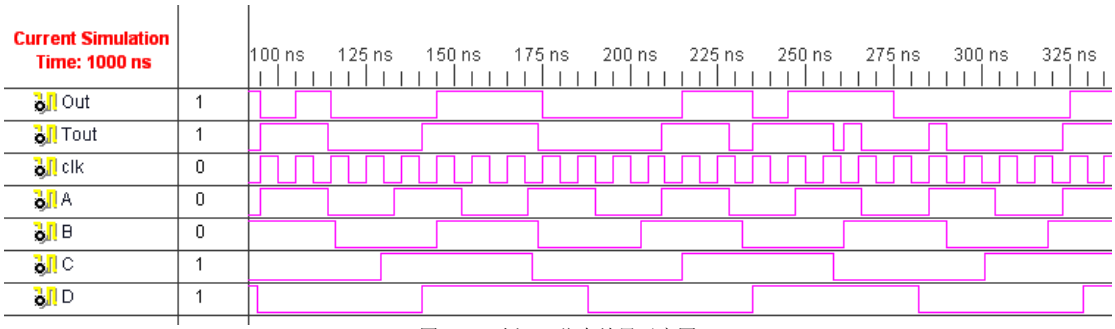


图 8-22　例 8-8 仿真结果示意图

8.1.4　时序逻辑的时钟选择策略

从 8.1.2 节可知，时钟对于时序逻辑是至关重要的，时钟的任何一个随机的抖动都会导致电路重新执行一次，使得逻辑时序发生错误跳转，因此时钟的质量对时序逻辑是至关重要的。设计时序电路的第一步就是选择性能优良的时钟信号。

1．Xilinx FPGA 时钟资源说明

全局时钟和第二全局时钟资源是 FPGA 芯片的重要资源之一，合理地利用该资源可以改善设计的综合和实现效果；如果使用不当，不但会影响设计的工作频率和稳定性等，甚至会导致设计的综合、实现过程出错。

（1）全局时钟

在 Xilinx 系列 FPGA 产品中，全局时钟网络是一种全局布线资源，它可以保证时钟信号到达各个目标逻辑单元的时延基本相同。其时钟分配树结构如图 8-23 所示。

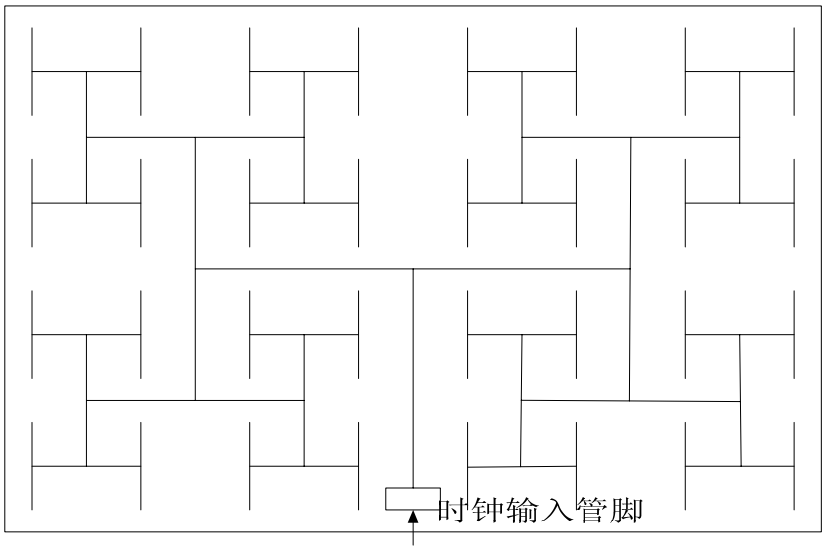


图 8-23　Xilinx FPGA 全局时钟分配树结构

针对不同类型的器件，Xilinx 公司提供的全局时钟网络在数量、性能等方面略有区别。但一般以全铜工艺实现，差分全局时钟控制技术将歪斜与抖动降至最低；再加上专用时钟缓冲与驱动结构，从而可使全局时钟到达芯片内部所有的逻辑可配置单元，且 I/O 单元以及块 RAM 的时延和抖动最小，以可满足高速同步电路对时钟触发沿的苛刻需求。

在 FPGA 设计中，FPGA 全局时钟路径需要专用的时钟缓冲和驱动，具有最小偏移和最大扇出能力，因此最好的时钟方案是由专用的全局时钟输入引脚驱动的单个主时钟，去钟控设计项目中的每一个触发器。只要可能就应尽量在设计项目中采用全局时钟，因为对于一个设计项目来说，全局时钟是最简单和最可预测的时钟。

（2）第二全局时钟

第二全局时钟属于长线资源，分布于芯片内部的行、列栅栏（Bank）上，其长度和驱动能力仅次于全局时钟资源，也可驱动芯片内部任何一个逻辑资源，其抖动和时延指标仅次于全局时钟信号。在设计中，一般将高频率、高扇出的时钟使能信号以及高速路经上的关键信号指定为全局第二时钟信号。

读者需要注意的是第二全局时钟资源和全局时钟资源的区别：使用全局时钟资源并不占用逻辑资源，也不会影响其他布线资源；而第二全局时钟资源占用的是芯片内部的资源，需要占用部分逻辑资源，各个部分的布线会相互影响，因此建议在逻辑设计占用资源不超过芯片资源 70%时使用。

2．全局时钟的使用

Xilinx FPGA 中的全局时钟采用全铜工艺实现，并设计了专用时钟缓冲与驱动结构，可到达芯片内部任何一个逻辑单元，包括可配置逻辑单元（CLB）、I/O 管脚、内嵌的块 RAM 以及硬核乘法器等模块，且时延和抖动都最小。因此，对于 FPGA 设计而言，全局时钟是最简单和最可预测的时钟。最好的时钟方案就是：由专用的全局时钟输入引脚驱动单个全局时钟，并用后者去控制设计中的每一个触发器，如图 8-24 所示。

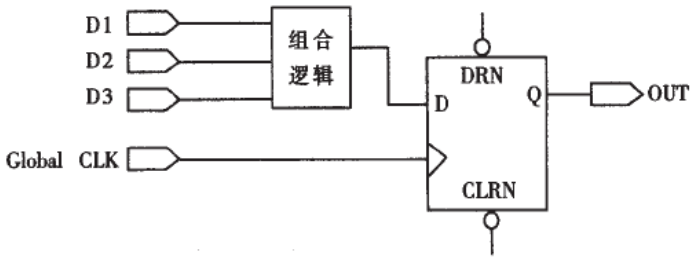


图 8-24 全局时钟应用策略

目前的主流芯片都集成了专用时钟资源与数字延迟锁相环，且数目众多。如面向中低端应用的 Spartan-3E 系列 FPGA，就可最多可提供 16 个全局时钟输入端口和 8 个数字时钟管理模块（DCM）；面向高端的 Virtex-4/5 系列芯片，可以提供多达数十个全局时钟输入端口和 DCM 模块。全局时钟主要有下面 2 种组合：

（1）IBUFG/ IBUFGDS +BUFG 的方法

IBUFG 后面连接 BUFG 的方法是最基本的全局时钟资源使用方法，由于 IBUFG 组合 BUFG 相当于 BUFGP，所以在这种使用方法也称为 BUFGP 方法。其相应的语法为：

```
IBUFG CLKIN_IBUFG_INST (.I(CLKIN_IN),
                        .O(CLKIN_IBUFG));

//调用 BUFG 原语，将 CLKIN_IBUFGDS 转成最终输出，其典型使用方法如下所述：
BUFG CLK_BUFG_INST (.I(CLKIN_IBUFGDS),
                    .O(CLK_OUT));
```

当输入时钟信号为差分信号时，需要使用 IBUFGDS 代替 IBUFG，相应的语法为：

```
//调用 IBUFGDS 原语，将差分时钟转换成单端输出 CLKIN_IBUFGDS
IBUFGDS CLKIN_IBUFGDS_INST (.I(CLKIN_P_IN),
                            .IB(CLKIN_N_IN),
                            .O(CLKIN_IBUFGDS));

//调用 BUFG 原语，将 CLKIN_IBUFGDS 转成最终输出
BUFG CLK_BUFG_INST (.I(CLKIN_IBUFGDS),
                    .O(CLK_OUT));
//示例结束
```

需要注意的是，当信号从全局时钟管脚输入，不论其是否为时钟信号，都必须使用 IBUFG 或 IBUFGDS；反之，如果对信号使用了 IBUFG 或 IBUFGDS 硬件原语，则该信号一定要从全局时钟管脚输入，否则在布局布线时会报错。IBUFG 和 IBUFGDS 的输入端仅仅与芯片的专用全局时钟输入管脚有物理连接，与普通 IO 和其它内部 CLB 等没有物理连接。

（2）LOGIC+BUFG 的方法

BUFG 不但可以驱动 IBUFG 的输出，还可以驱动其它普通信号的输出。当某个信号（时钟、使能、快速路径）的扇出非常大，并且要求抖动延迟最小时，可以使用 BUFG 驱动该信号，使该信号利用全局时钟资源。但需要注意的是，普通 IO 的输入或普通片内信号进入全局时钟布线层需要一个固有的延时，一般在 10ns 左右，即普通 IO 和普通片内信号从输入到 BUFG 输出有一个约 10ns 左右的固有延时，但是 BUFG 的输出到片内所有单元（IOB、CLB、选择性块 RAM）的延时可以忽略不计，认为其为“0”ns[5]。

相应的调用语法为：

```
//调用 BUFG 原语，将内部输入 CLK_IN 转成最终输出 CLK_OUT
BUFG CLK_BUFG_INST (.I(CLK_IN),
                    .O(CLK_OUT));
//示例结束
```

在软件代码中，可通过调用原语 IBUFGP 来使用全局时钟。IBUFGP 的基本用法是：

```
IBUFGP U1(.I(clk_in), .O(clk_out));
```

全局时钟网络对 FPGA 设计性能的影响很大，所以建议设计中的主时钟全部走全局时钟网络。

3．第二全局时钟的使用

第二全局时钟信号的驱动能力和时钟抖动延迟等指标仅次于全局时钟信号。Xilinx 的 FPGA 中一般有比较丰富的第二全局时钟资源（很多器件有 24 个第二全局时钟资源），以满足高速、复杂时序逻辑设计的需要。在设计中，一般将频率较高，扇出数目较多的时钟、使能、高速路径信号指定为第二全局时钟信号[5]。

第二全局时钟资源的使用方法比较简单，可通过在约束编辑器的专用约束（Misc）选项卡中，指定所选信号使用低抖动延迟资源“Low Skew”来制定。但最直接的方法是直接在指导 Xilinx 实现步骤的用户约束文件（UCF）中添加“USELOWSKEWLINES”约束命令。在约束编辑器中的操作等效于在用户约束文件中添加如下内容。

```
NET "s1" USELOWSKEWLINES;
NET "s2" USELOWSKEWLINES;
NET "s3" USELOWSKEWLINES;
```

8.2 同步时序电路和异步时序电路

触发器是构成时序逻辑电路的基本元件，根据电路中各级触发器时钟端的连接方式，可以将时序逻辑电路分为同步时序电路和异步时序电路。在同步时序电路中，各触发器的时钟端全部连接到同一个时钟源上，统一受系统时钟的控制，因此各级触发器的状态变化是同时的。在异步时序逻辑电路中，各触发器的时钟信号是分散连接的，因此触发器的状态变化不是同时进行的。

8.2.1 同步时序电路设计

1．同步时序电路原理说明

从构成方式上讲，同步时序电路所有操作都是在同一时钟严格的控制下步调一致地完成的。从电路行为上，同步电路的时序电路共用同一个时钟，而所有的状态变化都是在时钟的上升沿（或下降沿）完成的。例如，基本的 D 触发器就是同步电路，当时钟上升沿到来时，寄存器把 D 端的电平传到 Q 输出端；在上升沿没有到来时，即使 D 端数据发生变化，也不会立即将变化后的数据传到输出端 Q，需要等到下一个时钟上升沿。换句话说，同步时序电路中的只有一个时钟信号。

2．同步电路的 Verilog HDL 描述

同步逻辑是时钟之间存在固定因果关系的逻辑，所有时序逻辑都在同源时钟的控制下运行。注意，在 Verilog HDL 实现时并不要求同一时钟，而是同源时钟。所谓的同源时钟是指同一个时钟源衍生频率比值为 2 的幂次方，且初相位相同的时钟。例如，clk 信号和其同初相的 2 分频时钟、4 分频就是同源时钟。

（1）典型的同步描述

在 Verilog HDL 设计中，同步时序电路要求在程序中所有 always 块的 posedge/negedge 关键字后，只能出现同一个信号名称（包括同源的信号），并且只能使用一个信号跳变沿。下面给出一个同步时序电路的描述实例。

例 8-9：通过 Verilog HDL 给出一个同步的与门。

```
module syn_andgate(
      clk, a_in, b_in, y_out
      );
       input     clk, a_in, b_in;
       output    y_out;

       reg        y_out;
       always @(posedge clk) begin
            y_out <= a_in && b_in;
       end

endmodule
```

上述程序比较简单，这里就不给出其仿真结果。

（2）同步复位的描述

同步复位，顾名思义，就是指复位信号只有在时钟上升沿为有效电平时，才能达到复位的效果。否则，无法完成对系统的复位工作。同步复位的 Verilog 描述模板如下：

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

下面给出一个同步复位的应用实例。

例 8-10：给例 8-9 的同步与门添加一个同步复位功能。

```
module synrstn_andgate(
      clk, rst_n, a_in, b_in, y_out
      );
       input     clk, a_in, b_in, rst_n;
       output    y_out;

       reg        y_out;
       always @(posedge clk) begin
            if(!rst_n)
                y_out <= 0;
             else
                y_out <= a_in && b_in;
       end

endmodule
```

在 ISE 中的综合结果如图 8-25 所示，可以看出，复位信号 rst_n 通过 D 触发器的控制端来实现。

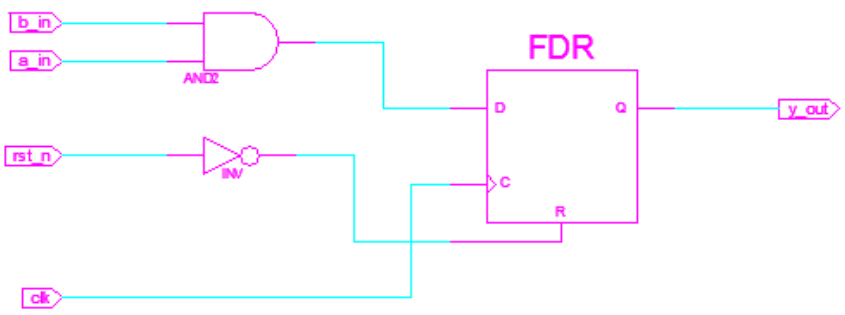


图 8-25　例 8-10 的 RTL 结构图

上述程序在 ISE 中的仿真结果如图 8-26 所示，复位信号并不是立即变高后，与门逻辑就开始工作，而要等到时钟信号 clk 的上升沿采样到 rst_n 信号变高后，与门逻辑才会对 clk 上升沿采样到的输入进行与运算。

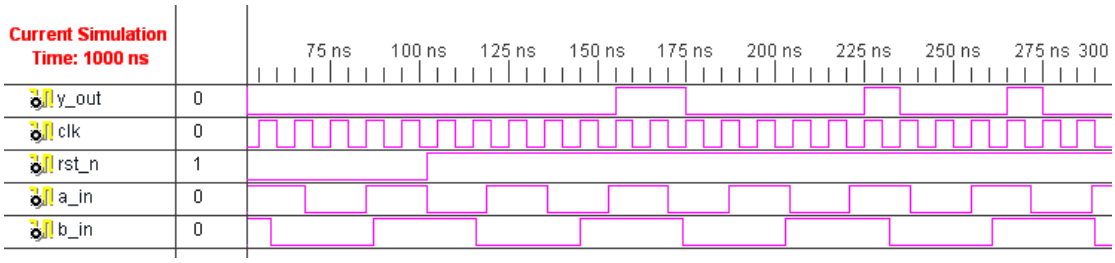


图 8-26　例 8-10 的仿真结果示意图

3．同步电路的准则

（1）单时钟策略、单时钟沿策略

尽量在设计中使用单时钟，在单时钟设计中，很容易就将整个设计同步于驱动时钟，使设计得到简化。尽量避免使用混合时钟沿来采样数据或驱动电路。使用混合时钟沿将会使静态时序分析复杂，并导致电路工作频率降低。下面给出混合时钟沿采样数据而降低系统工作时钟的实例。

在时序设计中，有时会因为数据采样或调整数据相位等需求，需要同时使用时钟的上升沿和下降沿对寄存器完成操作，设计人员很可能会想到下列两类写法，这两类做法在语法上是正确的，也可被综合，但在设计中不建议出现类似代码。

- 一个 always 模块

```
always @ (posedge clk or negedge clk) begin
    …
end
```

- 二个或多个 always 模块

```
always @ (negedge clk) begin
    …
end
…
always @ (posedge clk) begin
    …
end
```

上述两种方式都会使得在时钟上升沿和下降沿都对寄存器操作，其功能等同于使用了原来时钟的二倍频单信号沿来驱动电路。但对于可编程逻辑器件，不推荐使用同时使用同一

信号的两个沿。这是因为可编程逻辑器件内部的时钟处理电路，只能保证时钟的一个沿具有非常好的指标，而另外一个沿的抖动、偏斜以及过渡时间等指标都不保证，因此同时采用两个沿会造成时钟性能的恶化。因此在可编程逻辑的设计中，在这种情况下，推荐首先将原时钟倍频，然后利用单沿对电路进行操作。

此外，即使在 ASIC 设计中，同时利用上升沿和下降沿，意味着时序延迟折半，不利用后端做电路的时钟树综合的工作，并且也会对自动测试向量产生带来不利影响[6]。下面给出一个混合时钟沿采样的实例。

例 8-11：利用混合时钟先后完成输入数据的下降沿和上升沿采样，并级联输出。

```
module hunhe(clk, din, d1, dout);
        input    clk;
        input    [7:0] din;
        output [7:0] d1;
        output [7:0] dout;

        reg [7:0] d1, dout;

        always @(negedge clk) begin
            d1 <= din;
        end

        always @(posedge clk) begin
            dout <= d1;
        end

endmodule
```

程序在 ISE 中综合后的 RTL 级结构图如图 8-27 所示，比较两个 D 触发器就会发现：左端 D 触发器的时钟输入端有一个对时钟取反的操作。

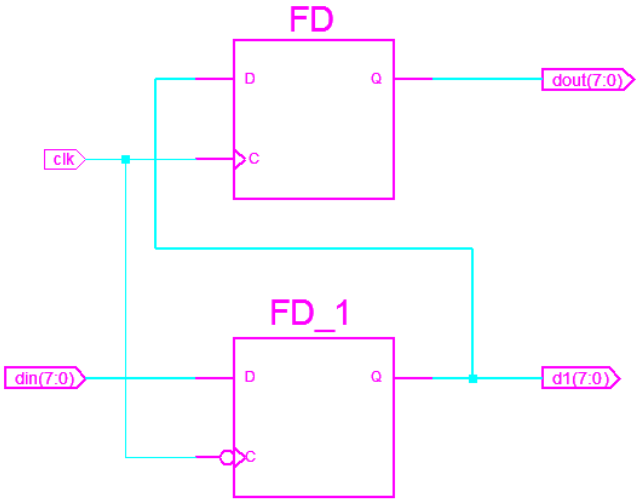


图 8-27　混合时钟采样实例的 RTL 级结构示意图

上述程序在 ISE Simulator 中的功能仿真结果如图 8-28 所示，可以看出其到达了设计目标。这说明，混合时钟沿电路并不影响功能仿真结果。

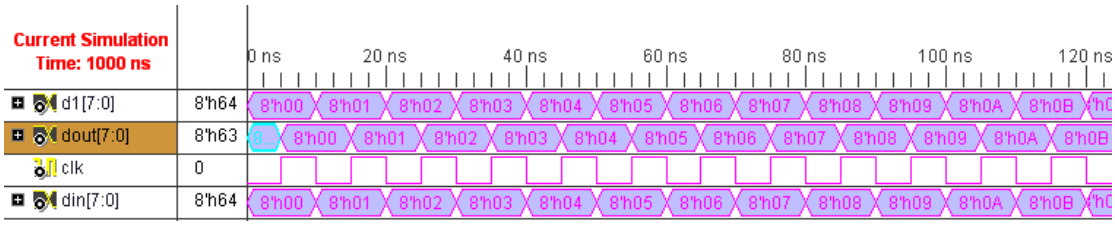


图 8-28　混合时钟采样的功能仿真结果

如前所述，混合电路影响的是电路的时序性能，所以其对于电路的真实影响需要通过时序仿真而得到。本例添加了 50MHz 的时序约束，然后对设计进行了时序仿真，结果如图 8-29 所示。可以看出，信号 d1 对数据采样错误，存在顺序颠倒的现象，意味着电路时序性能很差，无法正常工作在给定的时钟频率。

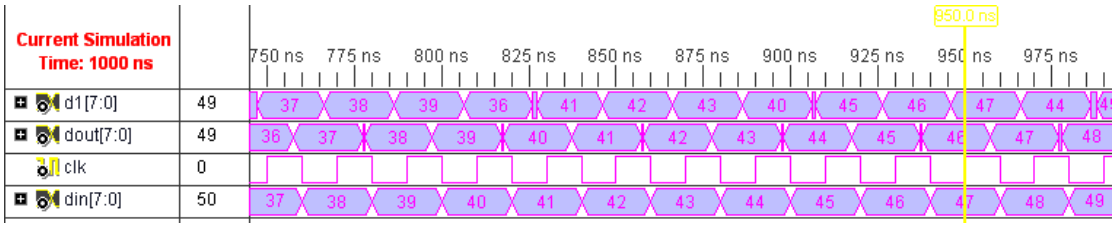


图 8-29 混合时钟采样的时序仿真结果

（2）避免使用门控时钟

如果一个时钟节点由组合逻辑驱动，那么就形成了门控时钟，如图 8-30 所示。门控时钟常用来减少功耗，但其相关的逻辑不是同步电路，即可能带有毛刺，而任何的一点点小毛刺都可以造成 D 触发器误翻转；此外，门控逻辑会污染时钟质量，产生毛刺，并恶化偏移和抖动等指标。所以门控时钟对设计可靠性有很大影响，应尽可能避免。不要为了节省功耗去使用门控时钟，最近发展起来的用于减少功耗的方法是：低核电压 FPGA、FPGA 休眠技术以及动态部分重构技术等，有兴趣的读者可以深入阅读文献[7]。

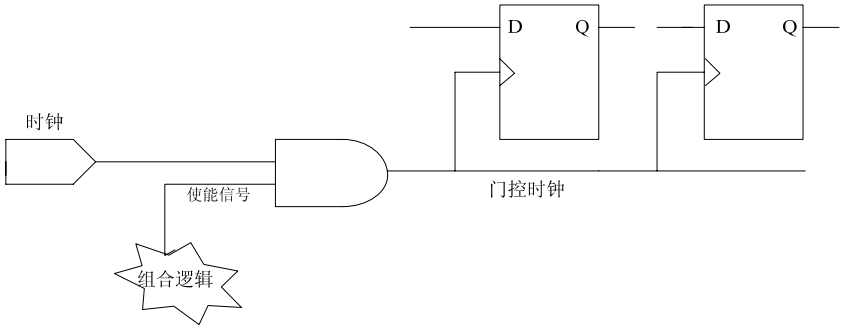


图 8-30 门控时钟示意图

（3）不要在子模块内部使用计数器分频产生所需时钟

各个模块内部各自分频会导致时钟管理混乱，不仅使得时序分析变得复杂，产生较大的时钟漂移，并且浪费了宝贵的时序裕量，降低了设计可靠性。推荐的方式是由一个专门的子模块来管理系统时钟，产生其他模块所需的各个时钟信号。

8.2.2　异步时序电路设计

1．异步时序电路原理说明

异步时序电路，顾名思义就是电路的工作节奏不一致，不存在单一的主控时钟，主要是用于产生地址译码器、FIFO 和异步 RAM 的读写控制信号脉冲。除可以使用带时钟的触发

器外，还可以使用不带时钟的触发器和延迟元件作为存储元件；电路状态的改变由外部输入的变化直接引起。由于异步电路没有统一的时钟，状态变化的时刻是不稳定的，通常输入信号只在电路处于稳定状态时才发生变化。也就是说一个时刻允许一个输入发生变化，以避免输入信号之间造成的竞争冒险。

2．异步电路的 Verilog HDL 描述

异步电路不使用时钟信号对系统逻辑进行同步，但仍需要对各子系统进行控制，因此采用预先规定的“开始”和“完成”信号或者状态完成逻辑控制。因此异步电路具有下列优点：无时钟歪斜问题、低电源消耗等。

（1）典型的异步描述

电子抢答器是一种典型的异步时序电路。假设一个电子抢答器具有 1 个主持人和 4 个抢答按钮。只有在主持人按下按钮后，才能开始抢答，当最先抢答的选手按下按钮后，其余选手的抢答按键失效，并将抢答成功的按钮序号显示出来。

根据设计要求，由于各个按钮不可能同时按下，因此电路内部肯定是异步执行的，例 8-12 给出相应的 Verilog HDL 实现代码。

例 8-12：利用 Verilog HDL 语言实现 4 个抢答按键的抢答器。

```
module jdq_demo(
    emcee, actor1, actor2, actor3, actor4, num
    );
     input              emcee, actor1, actor2, actor3, actor4;
     output   [2:0]     num;

     reg   [2:0]       num       = 0;
     reg   [3:0]       flag      = 0;
     reg               enable    = 0;
     reg               cnt       = 0;

     //产生抢答器的控制信号
     always @(emcee or actor1 or actor2 or actor3 or actor4) begin
         if(emcee == 1'b0) begin
            enable = 1'b1;
               cnt= 1'b1;
          end
          else begin
                // enable 为高后的第一次按键的抢答的人胜出
                enable = cnt && actor1 && actor2 && actor3 && actor4;
               cnt       = 0;
          end
     end

     //检测主持人和抢答人 1
     always @(negedge emcee or negedge actor1) begin //典型的异步设计
          if (!emcee) begin       //响应主持人按键
               flag[0] <= 1'b0;
          end
```

```
        else begin                  //响应抢答人 1 的按键，并做出是否有效的判断
            if(enable)              //抢答有效
                flag[0] <= 1'b1;
             else                   //抢答无效
                flag[0] <= flag[0];
        end
end

//检测主持人和抢答人 2
always @(negedge emcee or negedge actor2) begin
    if (!emcee) begin
            flag[1] <= 1'b0;
        end
        else begin
            if(enable)
                flag[1] <= 1'b1;
             else
                flag[1] <= flag[1];
        end
end

//检测主持人和抢答人 3
always @(negedge emcee or negedge actor3) begin
    if (!emcee) begin
            flag[2] <= 1'b0;
        end
        else begin
            if(enable)
                flag[2] <= 1'b1;
             else
                flag[2] <= flag[2];
        end
end

//检测主持人和抢答人4
always @(negedge emcee or negedge actor4) begin
    if (!emcee) begin
            flag[3] <= 1'b0;
        end
        else begin
            if(enable)
                flag[3] <= 1'b1;
             else
                flag[3] <= flag[3];
```

```
        end
    end

    //显示电路，显示成功抢答人的序号
    always @(flag) begin
        case(flag)
            4'b0000 : num = 3'b000;
            4'b0001 : num = 3'b001;
            4'b0010 : num = 3'b010;
            4'b0100 : num = 3'b011;
            4'b1000 : num = 3'b100;
            default : num = 3'b000;
        endcase
    end

endmodule
```

上述程序的仿真结果如图 8-31 所示，可以看出，3 号在主持人按键后第一个抢答成功，num 正确显示出其序号，达到了设计的目的。

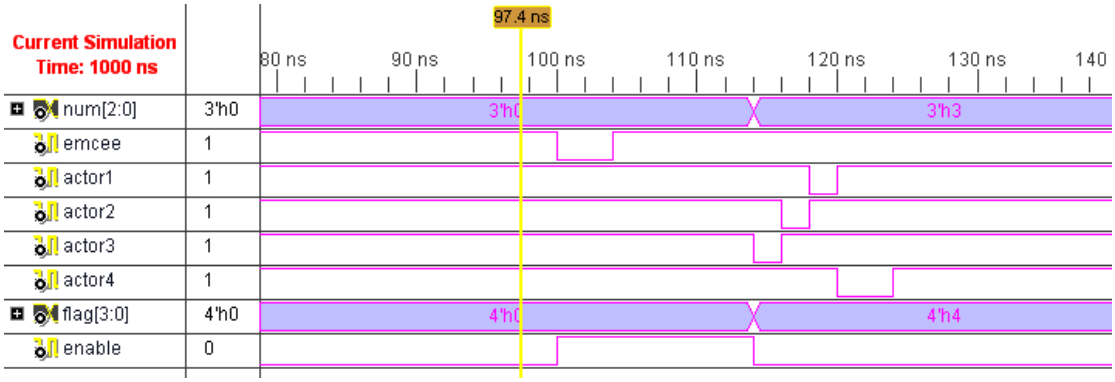


图 8-31　抢答器程序的仿真结果示意图

（2）异步复位的描述

异步复位是指无论时钟沿是否到来，只要复位信号有效，就对系统进行复位，其相应得 Verilog HDL 描述如下：

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

下面给出一个实例，将例 8-10 所示的同步复位与门电路转化成异步复位与门。

例 8-13：通过 Verilog HDL 语言实现一个异步复位与门。

```
module asynrstn_andgate(
      clk, rst_n, a_in, b_in, y_out
      );
        input     clk, a_in, b_in, rst_n;
        output    y_out;

        reg       y_out;
```

```
//异步描述
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
        y_out <= 0;
     else
        y_out <= a_in && b_in;
end

endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 8-32 所示，对比图 8-25 可以发现，异步复位的功能是通过 D 触发器的清零信号来实现的，从而达到复位信号随时有效的功能。

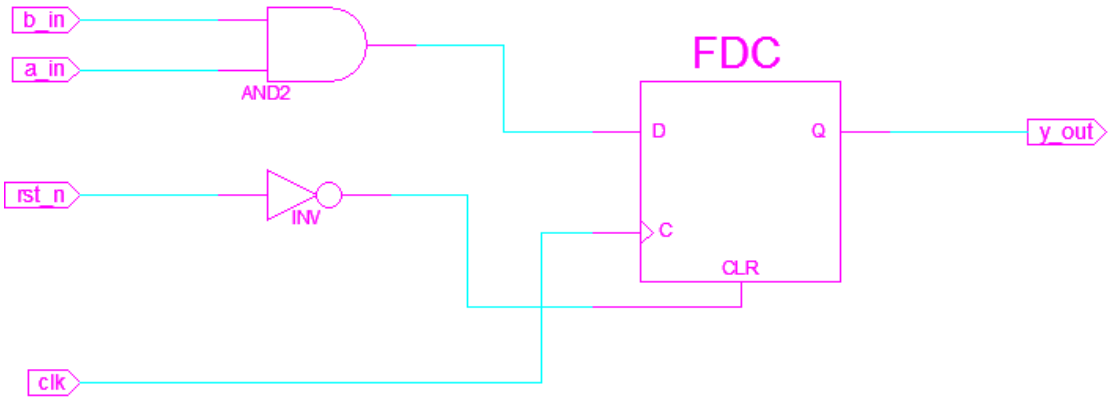


图 8-32　异步复位 D 触发器的 RTL 结构图

上述程序在 ISE Simulator 中的仿真结果如图 8-33 所示，复位信号只要有效，与门电路功能立即失效，输出零电平。

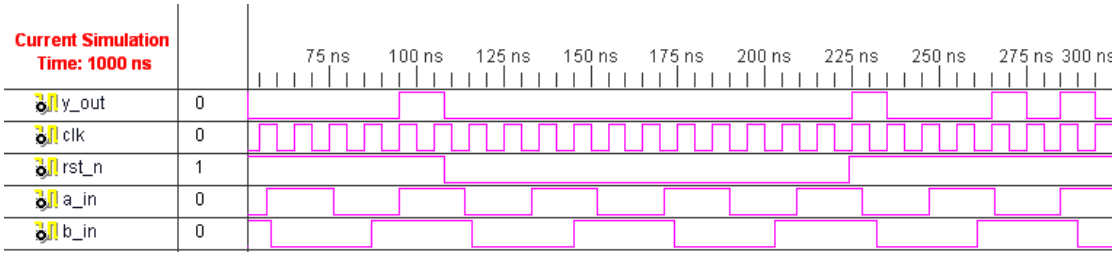


图 8-33　异步复位与门的仿真结果示意图

8.2.3　异步电路和同步电路的比较

同步电路在目前数字电路系统中占绝对优势，和异步电路相比具有下列优势：对温度、电压、生产过程等外部参数的适应性更强；可移植性更高；可以消除毛刺和内部歪斜的数据，能将设计频率提升到吉赫兹（GHz）。但是，同步电路也有缺点，因为需要时序器件，因此和异步电路相比，需要更多的逻辑资源，且由于所有动作都在时钟控制下，过高的信号翻转率使得设计功耗远大于异步电路功耗。

1．同步时序电路的优点

同步设计主要有以下三个优点：

（1）可以有效避免毛刺的影响，提高设计可靠性。毛刺是数字电路的天敌，只要有逻辑电路就会有毛刺发生，是永远存在的。因此，优秀的设计都必须从如何避免毛刺对设计的

不良影响入手，提高设计稳定性。同步设计是避免毛刺影响的最简单方法。

（2）可以简化时序分析过程。时序分析是高速数字设计的重要话题，参考文献[8]对其进行详细讨论。

（3）可以减少工作环境对设计的影响。异步电路受工作温度、电压等影响，器件时延变化较大，异步电路时序将变得更加苛刻，会导致芯片无法正常工作。同步电路只要求时钟和数据沿相对稳定，时序要求较为宽松，因此对环境的依赖性较小。

2．同步时序电路的缺点

同步逻辑也有两个主要的缺点：

（1）时钟信号必须要分布到电路上的每一个正反器。而时钟通常都是高频率的讯号，这会导致功率的消耗，也就是产生热量。即使每个正反器没有做任何的事情，也会消耗少量的能量，因此会导致废热产生。

（2）最大的可能时钟频率是由电路中最慢的逻辑路径决定，也就是关键路径。意思就是说每个逻辑的运算，从最简单的到最复杂的，都要在每一个时脉的周期中完成。一种用来消除这种限制的方法，是将复杂的运算分开成为数个简单的运算，这种技术称为“pipelining”。这种技术在微处理器中的作用显著，可以用来帮助提升现今处理器的时钟频率

3．应用小结

从延迟设计方面考虑，异步电路的延时靠门延时来实现，比较难预测；同步电路使用计数器或触发器实现延时。从资源使用方面考虑，虽然在 ASIC 设计中同步电路比异步电路占用的面积大，但是在 FPGA 中，是以逻辑单元衡量电路面积的，所以同步设计和异步设计相比，也不会浪费太多资源，加上目前的 FPGA 门数都比较大，在不是万不得已之际，不要使用异步设计。

同步设计时钟信号的质量和稳定性决定了同步时序电路的性能，FPGA 内部有专用的时钟资源，如全局时钟布线资源、专用的时钟管理模块 DUL、PLL 等。目前商用的 FPGA 都是面向同步的电路设计而优化的，同步时序电路可以很好地避免毛刺，提倡在设计中全部使用同步逻辑电路。特别注意，不同时钟域的接口需要进行同步。

当然，上述讨论只是一般建议，如果设计本身就具备异步背景，则异步电路肯定是首选。

8.3 阻塞赋值与非阻塞赋值

在 4.4.1 节已经对阻塞赋值与非阻塞赋值操作符进行了简单说明：组合逻辑使用阻塞赋值，时序逻辑使用非阻塞赋值。但仅此简单介绍无法让读者掌握其本质，不利于编写优秀的代码。本节将详细地阐述阻塞与非阻塞过程赋值的功能和执行过程，并通过一些具体的例子来分析它们之间的差异和本质特点。

8.3.1 阻塞赋值与非阻塞过程的深入理解

1．概念说明

阻塞与非阻塞过程赋值的误用不仅在仿真时会产生一些逻辑错误，而且会造成仿真与综合的不一致，更为严重的是往往这种错误不易被发现。为解决这一问题，必须深刻理解阻塞与非阻塞过程赋值的功能和执行过程的本质区别。并在此基础上运用一些可以产生可综合逻辑并能避免仿真错误的重要编码风格，才可以有效地避免阻塞与非阻塞过程赋值的误用。

在硬件中过程赋值语句表示的是：用赋值语句 RHS（Right Hand Side，表示右边表达式和变量）表达式所推导出的逻辑来驱动该赋值语句 LHS（Left Hand Side，表示左边表达式和变量）的变量，且只能出现在 always 语句和 initial 语句中。

阻塞赋值由符号“=”来完成，“阻塞赋值”由其赋值操作行为而得名：“阻塞”即是说

在当前的赋值完成前阻塞其他类型的赋值任务。

非阻塞赋值由符号“<=”来完成，“非阻塞赋值”在一个时间步的开始估计 RHS 表达式的值，并在这个时间步结束时用等式右边的值更新取代 LHS。在估算 RHS 表达式和更新 LHS 表达式的中间时间段，其他的对 LHS 表达式的非阻塞赋值可以被执行。即“非阻塞赋值”从估计 RHS 开始并不阻碍执行其他的赋值任务。

2．执行过程

在分析阻塞与非阻塞过程赋值语句的执行过程之前，必须了解 IEEE Verilog HDL 标准中对层次化事件队列的划分，本书在 6.2.2 节进行了专门介绍，如图 6-9 所示。其中，活跃事件列发生在当前仿真时刻的事件，列中的事件可以以任意的顺序执行。非活跃事件列也发生在当前仿真时刻的事件，但是必须等所有的活跃事件执行完后才执行。

阻塞过程赋值和非阻塞过程赋值中，对右端表达式求值都属于活跃事件，但阻塞赋值操作属于活跃事件，而非阻塞赋值操作属于非活跃事件。因此，从层次事件列表不难看出阻塞与非阻塞过程赋值在执行过程上的区别：

- 阻塞式赋值一步完成；
- 非阻塞式赋值分两步完成，首先完成右端表达式计算，再完成赋值操作。

8.3.2 组合逻辑中的阻塞与非阻塞

下面将通过一个具体的例子来理解阻塞与非阻塞在执行过程中的差别。

例 8-14：分别给出组合逻辑中 Verilog HDL 语言的阻塞赋值和非阻塞赋值语句的应用实例。

```
module ex1 ( out, a, b, c, d)
    input a, b, c, d;
    output out;
    reg t1, t2;
    always @ (a or b or c or d) begin
      t1 = a & b;     //阻塞赋值
      t2 = c & d;     //阻塞赋值
      out = t1 | t2;  //阻塞赋值
    end
endmodule

module ex2 ( out, a, b, c, d)
    input a, b, c, d;
    output out;
    reg t1, t2;
    always @ (a or b or c or d) begin
      t1 <= a & b;     //非阻塞赋值
      t2 < = c & d;    //非阻塞赋值
      out < = t1 | t2; //非阻塞赋值
    end
endmodule
```

这个例子清楚地展示了阻塞赋值和非阻塞赋值的差异。当输入信号 a、b、c、d 的值发生如下变化：a、b、c、d 都从 0 变到 1 时，在模块 ex1 中，采用阻塞赋值语句，所得的结果是 t1 变为 1，t2 变为 1，out 变为 1；在模块 ex2 中，采用非阻塞赋值语句所得的结果是

t1 变为 1，t2 变为 1，out 仍为 0。

从上述语句可以看出从阻塞语句与非阻塞赋值语句的最大区别：阻塞赋值是一步完成，而且一条语句执行的同时会阻止其他阻塞赋值语句的执行，所以执行“out = t1| t2;”语句时所用的 t1、t2 的值是更新过的值；非阻塞赋值是两步完成，而且一条语句地执行的同时不会阻止其他非阻塞赋值语句的执行，首先执行 RHS 的估值，此时 t1，t2 都没有被更新，使用的都是没更新的旧值，然后执行 LHS 的更新，所以 out 的值不变。

对于上例，如果仍然想利用非阻塞赋值，可以把@(a or b or c or d) 改为@ (a or b or c or d or t1 or t2)，这样每当 t1，t2 发生变化时，always 语句就会被重新计算，并最终得到正确的 out 值，但是这样会降低仿真器性能，由此发现非阻塞赋值存在以下两个问题：

- 非阻塞赋值不反映逻辑流；
- 需要将所有赋值对象都列入事件表中。

如果用阻塞式赋值就很容易避免这两个问题。

由8.1.1节可以知道，通过Verilog HDL可以有很多种方法为组合逻辑建模，但是当使用always块来为组合逻辑建模时，应该使用阻塞赋值。如果在某个always块里面只有一个赋值（表达），那么使用阻塞或者非阻塞赋值都可以正确工作。但是如果要完成高质量的HDL编码，还是要“总是用阻塞赋值对组合逻辑建模”。

8.3.3 时序逻辑中的阻塞与非阻塞

1．基本的时序逻辑

首先给出阻塞赋值语句在 D 触发器中的应用实例。

例 8-15：下面给出一个基于阻塞赋值和非阻塞赋值的多级触发器级联实例，要求将输入数据延迟 3 个时钟周期再输出，并给出对应的 RTL 级结构图和仿真结果。

（1）基于 D 触发器的阻塞赋值语句代码如下：

```
module pipeb1 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q1 = d;
        q2 = q1;
        q3 = q2;
    end
endmodule
```

上述代码综合后能得到所期望的逻辑电路吗？答案是否定的，根据阻塞赋值语句的执行过程可以得到执行后的结果是 q1 = d；q2 = d。实际只会综合出一个寄存器，如图 8-33 所示，并列出下面的警告信息，而不是所期望的三个。其中的主要原因就是采用了阻塞赋值，首先将 d 的值赋给 q1，再将 q1 的值赋给 q2，依次到 q3，但是 q1、q2、q3 的值在赋值前其数值已经全部被修改为当前时刻的 d 值，因此上述语句等效于 q3=d，这和图 8-33 所示的 RTL 结构是一致的。

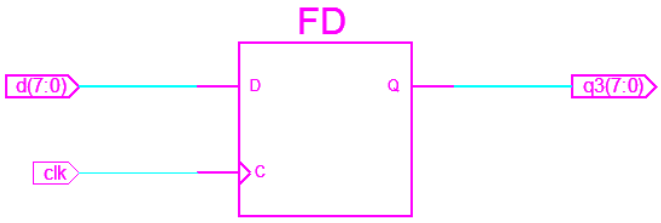


图 8-33　采用阻塞赋值的综合结果示意图

Synthesizing Unit <pipen1>.

Related source file is "pipen1.v".

WARNING:Xst:646 - Signal <q2> is assigned but never used. This unconnected signal will be trimmed during the optimization process.

WARNING:Xst:646 - Signal <q1> is assigned but never used. This unconnected signal will be trimmed during the optimization process.

Found 8-bit register for signal <q3>.

Summary:

inferred　　8 D-type flip-flop(s).

Unit <pipen1> synthesized.

图 8-34 给出了上述程序在 ISE Simulator 中的仿真结果，可以看出，其确实只能将数据沿迟一个时钟周期，没有达到预先的设计要求。

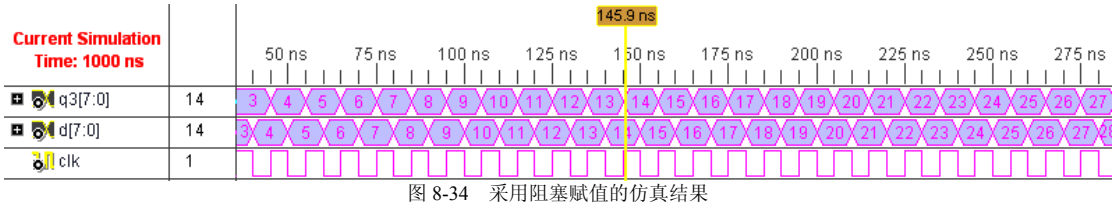


图 8-34　采用阻塞赋值的仿真结果

（2）如何才能得到所需要的电路呢？如果把 always 块中的两个赋值语句的次序颠倒后再进行分析：先把 q2 的值赋于 q3、再把 q1 的值赋于 q2，最后把 d 赋于 q1。这样在先赋值再修改，可以使得 q2，q3 的值都不再是 d 的当前值。修改后的代码如下所列。

```
module pipeb2 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q3 = q2;
        q2 = q1;
        q1 = d;
    end
endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 8-35 所示。

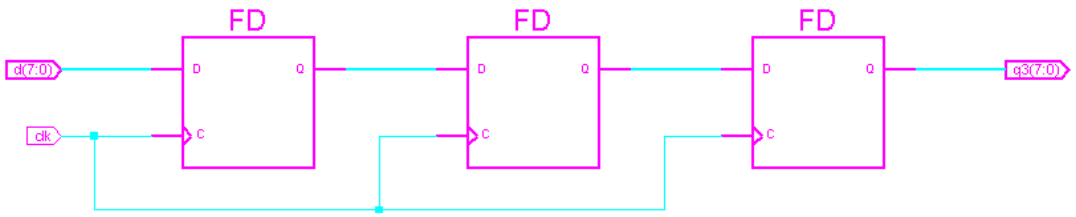


图 8-35　修改后阻塞赋值代码综合结果示意图

图 8-36 给出了上述程序在 ISE Simulator 中的仿真结果，可以看出，其正确将数据沿迟 3 个时钟周期，达到了设计要求。

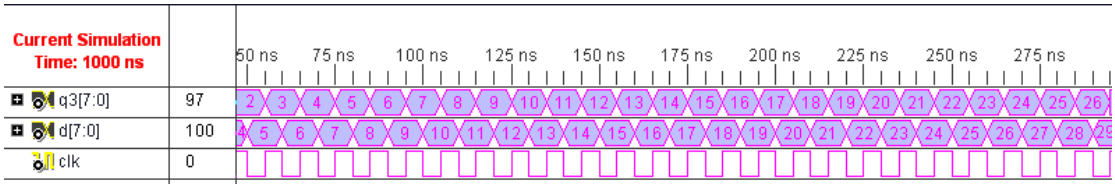


图 8-36　修改后阻塞赋值代码仿真结果

（3）保持和（1）中的赋值顺序一致，利用非阻塞赋值语句完成相应的设计，其代码如下所列。

```
module pipeb3(q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q1 <= d;
        q2 <= q1;
        q3 <= q2;
    end
endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 8-37 所示，发现即使和（1）中赋值顺序一致，仍然生成了 3 级 D 触发器，这是由于非阻塞赋值在 clk 上升沿首先得到所有“<=”符号右端的值，然后统一再将右端的值赋给左值，q2、q3 分别对应着上一时钟沿的 q1、q2，因此其需要 3 级触发器来实现。

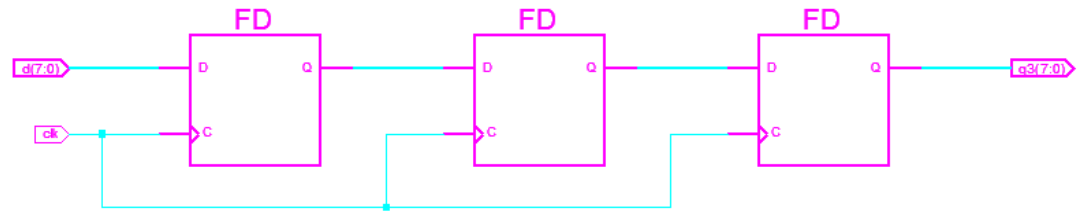


图 8-37　采用非阻塞赋值的综合结果示意图

图 8-38 给出了上述程序在 ISE Simulator 中的仿真结果，可以看出，输出数据和输入数据相差 3 个时钟周期，满足设计需求。

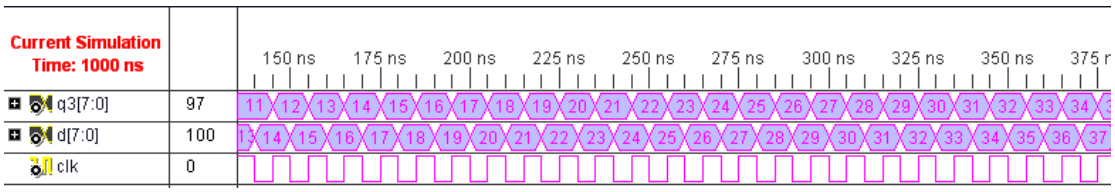


图 8-38　采用非阻塞赋值的仿真结果

（4）如果调换非阻塞赋值的语句顺序，使得其和（2）代码中的赋值顺序一致，其代码如下所列。

```
module pipeb4(q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q3 <= q2;
        q2 <= q1;
        q1 <= d;
    end
endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 8-39 所示，发现即使调整了赋值顺序，代码的 RTL 结构没有发生变化。因为所有的非阻塞赋值都是先得到右端的值，再统一赋给左端，加上其当前操作并不“阻塞”后续操作，因此非阻塞赋值结果与语句出现顺序无关。

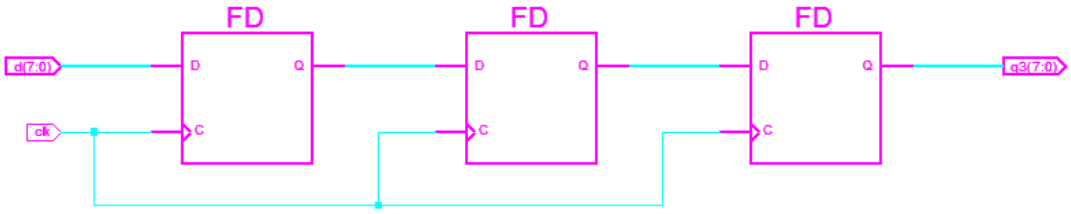


图 8-39　修改后的 RTL 结构图

图 8-40 给出了上述程序在 ISE Simulator 中的仿真结果。可以看出，输出数据和输入数据相差 3 个时钟周期，满足设计需求。

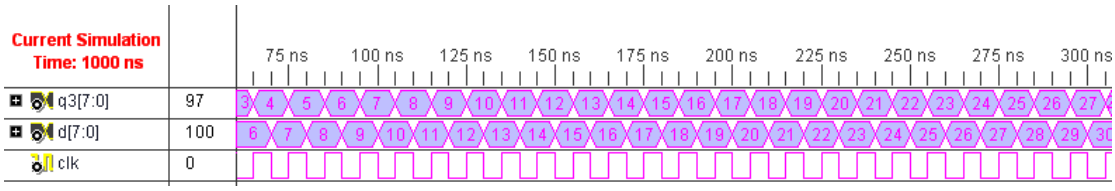


图 8-40　修改后的仿真结果

如果用非阻塞赋值来实现，就会发现不论两条语句的次序如何都能满足要求。如果把寄存器从 2 个变为 3，4，…，n 个，语句的次序就会更多，不同的次序对阻塞赋值会有不同的结果，但非阻塞赋值语句的结果都是一样的，所以一个好的编程风格就是：对时序逻辑建模采用非阻塞式赋值。

8.3.4 编码建议

1．结论

通过实践表明，遵循以下好的编码风格可以大大减少设计中的错误和提高设计效率：

- 对组合逻辑建模采用阻塞式赋值；
- 对时序逻辑建模采用非阻塞式赋值；
- 用多个 always 块分别对组合和时序逻辑建模；
- 尽量不要在同一个 always 块里面混合使用“阻塞赋值”和“非阻塞赋值”；
- 如果在同一个 always 块里面既为组合逻辑又为时序逻辑建模，应使用“非阻塞赋值”，不要在同一个 always 块里面混合使用“阻塞赋值”和“非阻塞赋值”；
- 当为锁存器（latch）建模，使用“非阻塞赋值”。

2．阻塞赋值和非阻塞赋值语句段各自独立

有时候为了方便会把时序逻辑和简单的组合逻辑放在一个 always 块，此时需要使用非阻塞赋值为这种混合逻辑建模。Verilog HDL 标准允许在一个 always 块里面自由混合“阻塞”与“非阻塞”赋值，但这是一种特别不好的代码风格。因此，在 ISE 中综合类似这样的代码会出错。下面给出一个应用实例。

例 8-16：下面给出一个混合赋值语句的实例。

```
module ba_nba2 (q, a, b, clk, rst_n);
    output q;
    input a, b, rst_n;
    input clk;
    reg q;

    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            q <= 1'b0;      //非阻塞赋值
        end
        else begin
              q = a & b;    //阻塞赋值
        end
    end

endmodule
```

上述程序在 ISE 综合时，会给出图 8-41 所示的错误提示，可以看出该程序不能正确地完成综合过程。

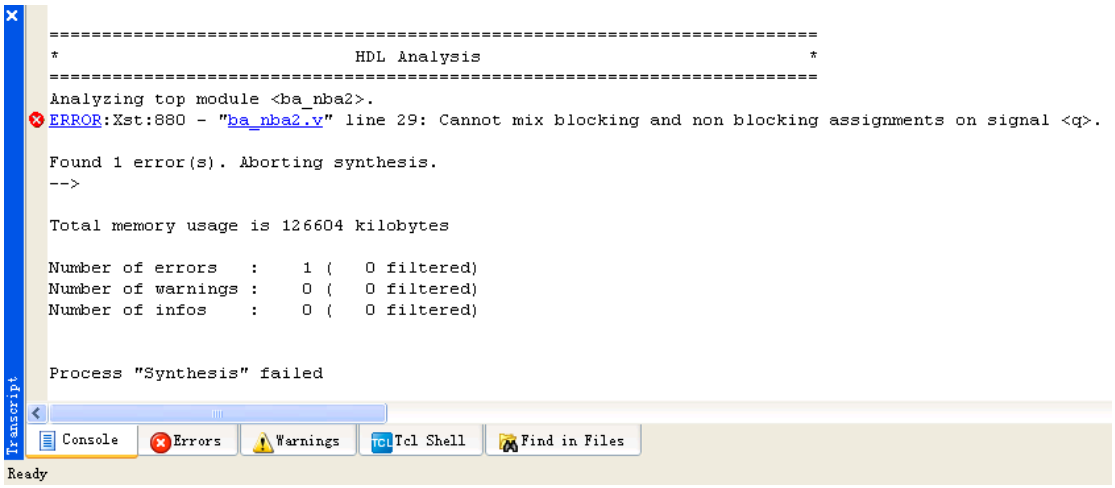


图 8-41　例 8-16 的综合错误提示信息示意图

8.4 双向端口

8.4.1 双向端口简介

三态缓冲器也称三态门，其典型应用是双向端口，常用于双向数据总线的构建。在数字电路中，逻辑输出有两个正常态：低电平状态（对应逻辑 0）和高电平状态（对应逻辑 1）；此外，电路还有不属于 0 和 1 状态的高组态（对应于逻辑 Z）。所谓高阻，即输出端属于浮空状态，只有很小的漏电流流动，其电平随外部电平高低而定，门电平放弃对输出电路的控制。或者可以理解为输出与电路是断开的。最基本的三态缓冲器的逻辑符号如图 8-42 所示。

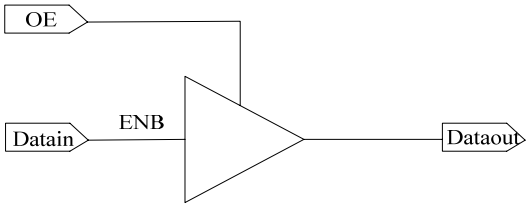


图 8-42　三态缓冲器的逻辑符号图

当 OE 为高电平时，Dataout 与 Datain 相连；而 OE 为低时，Dataout 为高阻态，相当于和 Datain 之间的连线断开。

在应用代码中，Verilog HDL 程序模块首先要进行 I/O 端口（input：输入端口；output：输出端口；inout：双向端口，同时具有输入/输出功能的端口）的定义，然后是逻辑功能的描述。在 Verilog HDL 中，output 端口信号可定义成寄存器型变量，并在 always 块内可以被赋值使用，而 inout 型双向端口信号不能被定义成 reg 型变量，因此在 always 块内不能被直接赋值使用，这一点与 VHDL 中双向端口的使用方法不同。

由于现在 FPGA 设计和外部存储器或 CPU 数据交换的频繁运用，以及引脚资源有限，使用双向端口设计可以成倍地节省数据引脚线，所以利用 Verilog HDL 实现双向端口至关重要。在设计双向端口时应注意两点：其一，要用三态门的控制来处理实现双向端口；其二，要分别指定双向端口作为输出口和输入口时，对外部对象的数据操作。

8.4.2 双向端口应用实例

1．双向端口的 Verilog HDL 描述

假设输入输出数据的位宽都为 16，那么如果数据输入口和输出口分别设计则共需要 32 根数据线，而用双向端口来设计，只需要 16 根数据线，这样就节省了 16 根数据线引脚。下面

给出应用实例。

例 8-17：双向端口的 Verilog 实例。

```
module dinout(din,z,clk,dout,dinout);
    input [7:0] din;
    input z;
    input clk;
    output[7:0] dout;
    inout [7:0] dinout;

    reg [7:0] dout;
    reg [7:0] din_reg;

    assign dinout= (!z)?din_reg:8'bz;

    always @ (posedge clk) begin
        if(!z)
            din_reg=din;
        else
            dout=dinout;
    end

endmodule
```

程序在 ISE 中综合后的 RTL 级结构图如图 8-43 所示。dinout 定义为双向端口，即可作为输入端口，又可作为输出端口；当双向端口 dinout 作为输出口时，从输入端口 din 输入数据到模块中，让数据从 dinout 端口输出；当双向端口 dinout 作为输入口时，数据从 dinout 口输入，从输出端口 dout 输出。z 为三态门选通信号，当 z=1 时，把三态门置为高阻态，这时 dinout 作为输入口；当 z=0 时，开通三态门，dinout 作为输出端口。

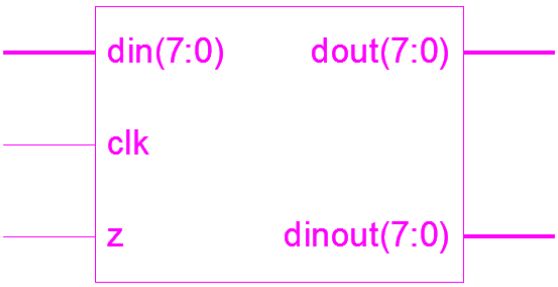


图 8-43　例 8-17 的综合结果示意图

2．双向端口的仿真

编写测试模块时，对于 inout 类型的端口，需要定义成 wire 类型变量，而其它输入端口都定义成 reg 类型，这两者是有区别的[9]。此外，对于双向端口本身，仿真其输出端口和输入端口的语法是不同的。下面分别给出例 8-19 中双向端口的输入、输出特性仿真。

（1）输出端口特性仿真

当双向端口作为输出口时，不需要对其进行初始化，只要开通三态门即可。例 8-17 的输出端口特性仿真代码如例 8-18 所示。假设在 100ns 后，让数据 10、11、12、13、14、15、16、17、18、19 以及 20 依次从 din 口输入，然后用 10ns 的采样时钟从双向端口 dinout 输出。

例 8-18：例 8-17 中双向端口 dinout 的输出特性仿真实例。

```
`timescale 1ns/1ps
module tb_dout;
    reg [7:0] din;
    reg z;
    reg clk;
    wire [7:0] dout;
    wire [7:0] dinout;
    integer i;

    dinout uut(
        .din(din),
        .z(z),
        .clk(clk),
        .dout(dout),
        .dinout(dinout)
    );

    always #5 clk= ~clk;

    initial begin
        din = 0;
        z = 0;
        clk = 0;
        #100 din=10;
        for(i=0; i<10; i=i+1)
            #10 din=din+1;
        end
endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 8-44 所示。可以看出这时双向端口 dinout 作为输出端口，输出从 din 端口输入到模块中的数据，达到了设计目的。

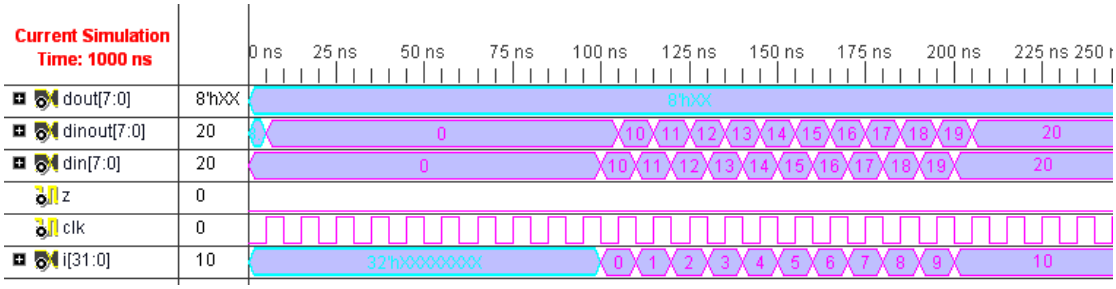


图 8-44　双向端口输出特性仿真结果

（2）输入端口特性仿真

当双向端口 dinout 作为输入口时，需要对它进行初始化赋值并关闭三态门。而如果把它跟一般的输入口一样直接初始化赋值，则会出错，这是因为在定义它的时候是 wire 型的数据变量，而不是 reg 型数据类型。因此，这里需要用到 force 命令，用来强制给 dinout 赋值。同样，下面给出一个 Verilog HDL 仿真实例，设定在 100ns 后，让数据 20、19、18、17、

16、15、14、13、12、11 以及 10 从 dinout 口输入模块，然后用 10ns 的采样时钟从输出口 dout 输出。

例 8-19：例 8-17 中双向端口 dinout 的输入特性仿真实例。

```
module tb_din;
    reg [7:0]din;
    reg z;
    reg clk;
    wire [7:0] dout;
    wire [7:0] dinout;

    integer i;

    dinout uut(
    .din(din),
    .z(z),
    .clk(clk),
    .dout(dout),
    .dinout(dinout)
    );

    always #5 clk = ~clk;

    initial begin
    z = 1;
    clk = 0;
    force dinout=20;
    #100;
    for (i=0;i<10;i=i+1)
        #10 force dinout=dinout-1;
    end

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 8-45 所示，可以看出这时双向端口 dinout 作为输入端口，再将数据从端口 dout 完整输出，达到了预期目的。

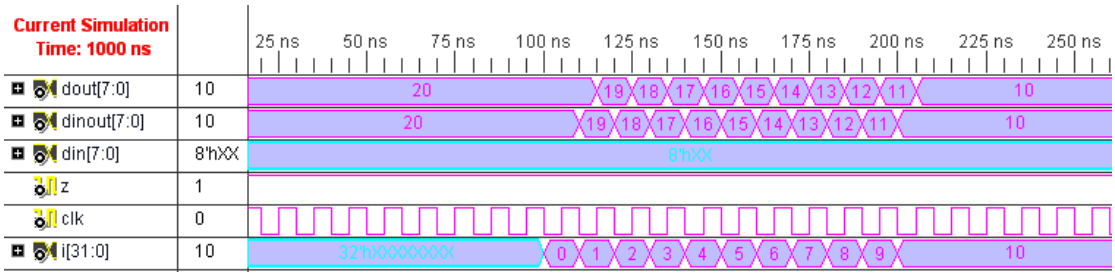


图 8-45　双向端口输入特性仿真结果

8.6 锁存器

锁存器是一种对脉冲电平敏感的存储单元电路，可以在特定输入脉冲电平作用下改变状态，其本身也是一类常用的逻辑单元，有着特定的需求。但由于在实际的代码书写中，很容易产生设计人员不期望的锁存器，使得设计功能出错，因此很多 Verilog HDL 书籍都对其“闻虎色变”。但实质上，锁存器只是一种基本单元，完全受代码控制。本章主要说明为什么会生成锁存器以及如何在设计中合理地应用锁存器。

8.6.1 锁存器本质说明

1．锁存器的基本概念

锁存器是一种在异步时序逻辑电路系统中，对输入信号电平敏感的单元，用来储存信息。一个锁存器可以储存一个比特的信息。通常，锁存器会多个一起出现，因此会有 4 位锁存器（可以储存四个比特）以及 8 位锁存器（可以储存八个比特）等诸多名称。

锁存器在数据未锁存时，输出端的信号随输入信号变化，就像信号通过一个缓冲器一样，一旦锁存信号有效，则数据被锁住，输入信号不起作用。因此，锁存器也被称为透明锁存器，指的是不锁存时输出对于输入是透明的。

2．锁存器和寄存器的区别

锁存器和寄存器都是数字电路的基本存储单元，但锁存器是电平触发的存储器，触发器是边沿触发的存储器。

本质上，锁存器和 D 触发器的逻辑功能是基本相同的，都可存储数据，且锁存器所需的门逻辑更少，具备更高的集成度。但锁存器具备下列三个缺点：

（1）对毛刺敏感，不能异步复位，因此在上电后处于不确定的状态。

（2）锁存器会使静态时序分析变得非常复杂，不具备可重用性。

（3）在 PLD 芯片中，基本的单元是由查找表和触发器组成的，若生成锁存器反而需要更多的资源。

根据锁存器的特点可以看出，在电路设计中，要对锁存器特别谨慎，如果设计经过综合后产生出和设计意图不一致的锁存器，则将导致设计错误，包括仿真和综合。因此，在设计中需要避免产生意想不到的锁存器。

8.6.2 锁存器的产生原因和处理策略

1．锁存器的产生原因

如果组合逻辑的语句完全不使用 always 语句块，就可以保证综合器不会综合出锁存器，例如：

```
assign a = din ? x : y;
```

上述语句不需要保持信号 a 的前一个状态，因此肯定不会产生锁存器。在基于 always 的组合逻辑描述语句中，可能产生锁存器的情况具体可分为两种：其一是在 if 语句中，另一种是在 case 语句中。下面将对 if 和 case 语句造成的锁存器分别进行分析。

（1）if 语句造成的锁存器

例 8-20 给出了在 always 块中使用 if 语句，但缺乏 else 分支而造成锁存器的情况。

例 8-20：表 8-2 给出由于 if 语句不完整而生成意外的锁存器示例。

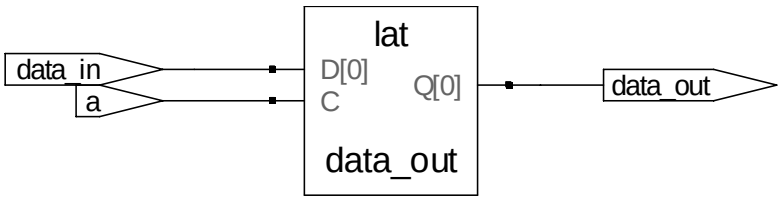
表 8-2　　if 语句不完整的情况

生成意想不到的锁存器	无锁存器
always@(a or data_in) begin if(a) begin	always@(a or data_in) begin if(a) begin

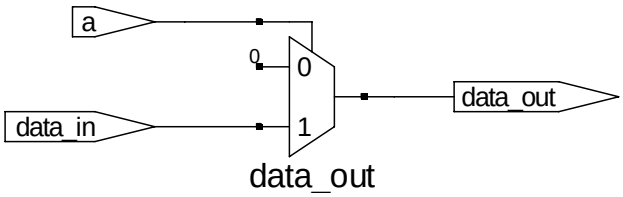
```
        data_out = data_in;
    end
end
```

```
        data_out = data_in;
    end
    else begin
      data_out = 0;
    end
end
```

上述两段程序经过综合后，其 RTL 视图分别如图 8-46（a）以及（b）所示。



(a) 存在锁存器的 RTL 结构



(b) 不存在锁存器的 RTL 结构

图 8-46 if 语句生成锁存器的 RTL 示意图

可以看出，左边的语句块只有在 a 的值为 1 的情况下，data_in 的值才能传递给 data_out，但没有指定在 a 的值为 0 的情况下 data_out 的取值。这样在 always 语句块中，如果没有改变变量的赋值，变量值将保持不变，生成锁存器。如果希望在 a=0 时，data_out 值为 0，那么程序就如右例所示。本例说明利用 else 分支就不会生成锁存器。

（2）case 语句造成的锁存器

例 8-21 给出了在 always 块中使用 case 语句，由于缺乏 default 分支而造成锁存器的情况。

例 8-21：表 8-3 给出由于 case 语句不完整而生成意外的锁存器示例。

表 8-3 **case 语句不完整的情况**

生成意想不到的锁存器	无锁存器
always@(a[1:0] or data_in1 or data_in2) begin case(a) 2'00: data_out = data_in1; 2'01: data_out = data_in2; endcase end	always@(a[1:0] or data_in1 or data_in2) begin case(a) 2'00: data_out = data_in1; 2'01: data_out = data_in2; default: data_out = 0; endcase end

左边的例子中，当 a[1:0]的值为 2'00、2'01 时，分别将 data_in1 或 data_in2 赋给 data_out，在 a 为其余值的时候就生成了锁存器，data_out 保持上一次的赋值保持不变；右边的例子比较明确，在 a 的值不等于 2'00、2'01 时，data_out 的值为 0，不会生成锁存器。

以上两个例子给出了如何避免生成意外锁存器。即如果用到 if 语句，最好有 else 分支；如果用到 else 语句，最好有 default 语句。即使需要锁存器，也通过 else 分支或 default 分支来显式说明。按照上面的建议，可以避免意想不到的错误，提高程序的稳健性和可读性。

8.6.3 锁存器的应用规则

1．锁存器的本质

锁存器作为一种电路单元，必然有其存在的理由以及应用场景，并不像目前的很多书籍简单地将锁存器列为“头等敌人”。其实在实际中，有些设计是不可避免地要用到锁存器，特别是在总线应用上，锁存器能提高驱动能力、隔离前后级。例如，常见的应用包括地址锁存器、数据锁存器以及复位信号锁存器等。但在更多的情况下，很容易在代码中产生未预料到的锁存器，使得逻辑功能不满足要求，浪费了大量的调试时间，从而使得大多数设计人员“闻虎色变”。因此较好的应用规则是：要学会分析是否需要锁存器以及代码是否会产生意想不到的锁存器。只有这样才能灵活运用锁存器。下面通过实例来给予说明。

例 8-22：通过 Verilog HDL 语言实现序列最大值的搜寻程序，并保持检测到的最大值。

```
module two_max(
    a, reset_n, abmax
   );
    input   [7:0] a;
    input         reset_n;
    output [7:0] abmax;

    reg      [7:0] abmax_tmp;

    always   @(a or reset_n) begin
      if (reset_n)
           abmax_tmp = 8'h00;
       else
           //判断最大值，并利用锁存器锁存
           if (a > abmax_tmp)
               abmax_tmp = a;
    end

    assign abmax = abmax_tmp;

endmodule
```

上述代码在 ISE 中的综合结果会给出，设计中包含锁存器所产生时序问题的警告信息，如下所示：

WARNING:Xst:737 - Found 8-bit latch for signal <abmax_tmp>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.

但实际上，信号 abmax_tmp 的锁存器正是我们需要的。因此，虽然综合工具给出警告信息，但锁存器是设计所需要的，因此上述代码的设计是没有问题的。读者可能会反问，综合工具是不是因为没有将 if 语句的 else 分支补充完整而出现的警告呢？可以将例 8-22 中的 if 语句补充完整，如下所列：

```
if (a > abmax_tmp)
    abmax_tmp = a;
else
```

```
        abmax_tmp = abmax_tmp;
```

经过综合后，仍然会有锁存器的警告信息。至此，读者就明白了，无论锁存器是否是用户需要的，ISE 都会给出相关的提示信息，其主要原因就是因为从锁存器会对整个设计的时序性能有较大的影响。所以，在设计中要尽量避免出现锁存器；如果确实要使用锁存器，对 EDA 软件的提示信息也不要“惧怕”。

2．“不期望”锁存器应用示例

所谓“非期望”锁存器，指的是其和设计人员的设计意图不匹配，但实际上，锁存器是一种固定电路单元，所以主要问题在于设计人员没有合理利用 Verilog HDL 语言，常见的原因就是对条件语句（if 语句、case 语句）的分支描述不完整。例如，设计一个两输入（信号 a、b）电路，如果 a 大于 b，则输出高电平，否则输出低电平。如果在代码编写时，只有 a 大于 b 分支，缺少 a 小于 b 分支，则综合器为了满足寄存器保持数据的特征，会产生锁存器，从而使得输出一直为高，造成其逻辑功能和用户意图不符，出现“不期望”的情况。

下面再给出典型的“非期望”锁存器开发实例。

例 8-23：利用 Verilog HDL 语言实现一个锁存器，当输入数据大于 127 时，将输入数据送到输出端口，否则输出 0。

（1）产生锁存器的代码示例

```
module latch_demo(
    din, dout
    );

     input    [7:0] din;
     output [7:0] dout;

    reg       [7:0] dout;

    always @(din)begin
        if(din > 127)
              dout = din;
    end

endmodule
```

程序在 ISE 综合后的 RTL 级代码如图 8-47 所示，可以看出，其在比较器后面级联了锁存器 Gate，这就是由于缺少 if 语句的 else 分支造成的。

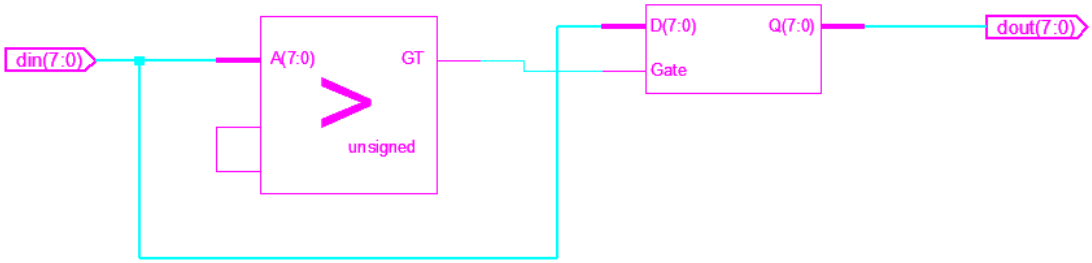


图 8-47　带锁存器的比较器 RTL 结构示意图

上述程序在 ISE Simulator 中的仿真结果如图 8-48 所示，可以看出，当 a 小于 b 时，不能输出 0，而是保持了前一次大于 127 的数值，没有达到设计要求。

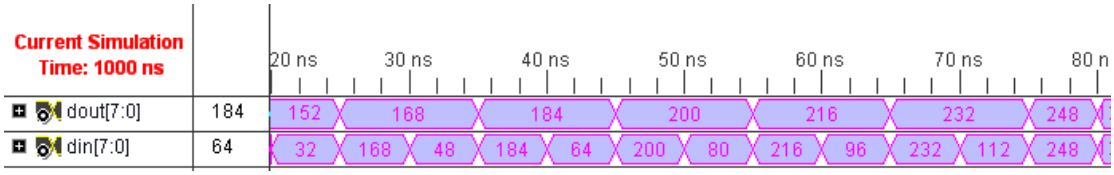


图 8-48　带锁存器的比较器仿真结果示意图

（2）上述程序的修改非常简单，只要添加完整 else 分支即可，其修改后的代码如下。

```
module latch_demo(
      din, dout
      );

       input    [7:0] din;
       output [7:0] dout;

       reg       [7:0] dout;

       always @(din)begin
          if(din > 127)
                dout = din;
            else
                dout = 0;
       end

endmodule
```

程序在 ISE 综合后的 RTL 级代码如图 8-49 所示，可以看出，补充完整 else 分支后，其在比较器后面级联了与门，替代了原有的锁存器。

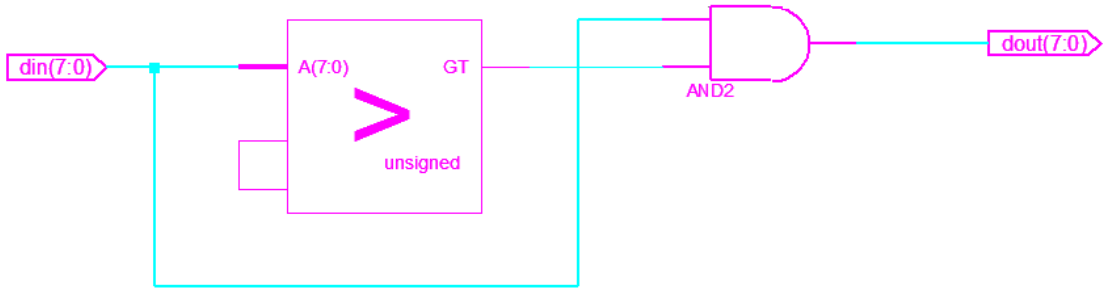


图 8-49　无锁存器的比较器 RTL 结构示意图

上述程序在 ISE Simulator 中的仿真结果如图 8-50 所示，可以看出，其与设计要求符合，当 a 大于 b 时，输出 a；当 a 小于 b 时，不能输出 0，达到了设计要求。

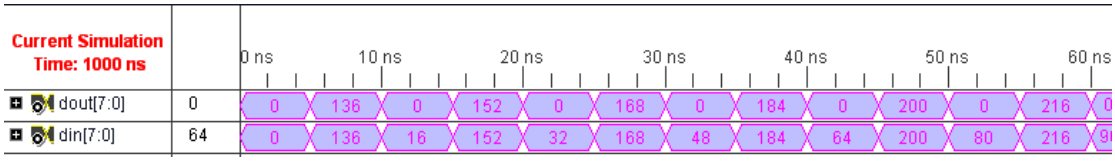


图 8-50　无锁存器的比较器仿真结果示意图

至此，读者可以得到一个基本结论：锁存器是一种基本电路单元，会影响到电路的时序性能，应该尽量避免使用，但出现锁存器造成设计和原始意图不符的情况，则是由于设计人

员代码输入不正确造成的。

8.7 消除不确定输入的电路设计

8.7.1 初始值不确定态的消除

关于变量初始值的讨论在 6.6.1 节进行了一定的讨论，主要针对仿真、EDA 软件设置以及复位信号的应用这三个方面来介绍。但实际上，对于一个实际的电子系统，其复位信号不是随便可以产生的，因此子模块的信号初始化需要内部自动完成，达到硬件可控的目的。下面通过一个简单的计数器来说明如何完成初始值的自动设定。

例 8-24：通过 Verilog HDL 语言实现一个初值可控的计数器。

```
module auto_set_demo(
        clk, cnt_out
    );
      input          clk;
      output [7:0] cnt_out;
      reg     [7:0] cnt_out;

      reg            flag;

    always @ (posedge clk) begin
        if (flag != 1'b1) begin
            flag <= 1'b1;
             //完成初始值的设定，本例设为 8'haa
             cnt_out <= 8'haa;
          end
          else    begin
              cnt_out <= cnt_out + 1;
          end

      end
endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 8-51 所示，从中可以看出 flag 信号由低到高的跳变可以完成计数器的初始化，无须外部控制信号。

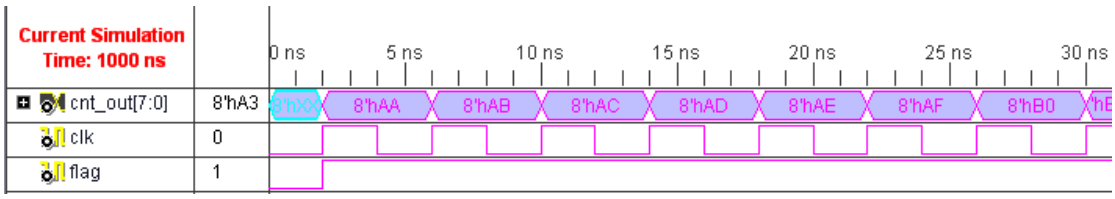


图 8-51　例 8-24 仿真结果示意图

8.7.2 逻辑运算不确定态的消除

有些电路，尽管它的某些输入是不确定值，但其输出却是个确定值。这种最简单例子是与门的一个输入是不确定值 x，另一个输入却是 0。在这种情况下，Verilog HDL 可以识别出门的输出一定是 0。此外，还有更复杂的例子，如实际中的 2 选 1 选择器，如果选择器的两

个输入都是 0，而输入的控制信号是 x，那么不管控制信号是什么，输出确定是 0。但对于这种情况下，Verilog HDL 却不能认识到这种情况，相反会把 x 值传播到输出口。所以，这就需要设计者自行设计一个电路，使其在所有条件下都能展示出期望行为的选择器。

在 Verilog HDL 中，有两种不同的原因可能导致信号值为 x。第一种原因是，有两个不同的信号源用相同的强度驱使同一个节点，并试图驱动成不同的逻辑值，这一般是由设计错误造成的。第二种原因是信号值没有初始化。所以在设计组合逻辑时，需要将不确定的输入转化成确定输入，然后再完成组合逻辑。这种结构如图 8-52 所示：

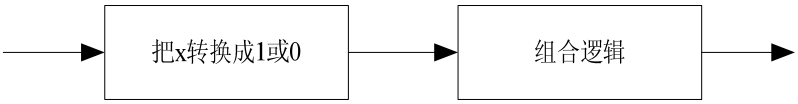


图 8-52　输入的不确定值转换成确定值

例 8-25：一个转换不确定输入为确定输出的模块

```
module x2one (in, out);
    input in;
    output out;
    assign out = (in==1) ? 1: 0;
endmodule
```

可以看出，程序非常简单，其本质就是一个直接赋值语句，在 ISE 综合后的 RTL 结构图如图 8-53 所示，其中的 ALIAS 模块就是一个“Out <= >In”的赋值操作模块。虽然，最终的硬件的没有不确定态，但该操作可以保证硬件和软件仿真一致。

图 8-53　不确定输入转换模块的 RTL 结构图

上述程序在 ISE Simulator 中的仿真结果如图 8-54 所示，可以看出，这样在仿真中就消除了 x 以及 z 状态，只有 1 和 0，从而使得仿真和硬件操作一致。

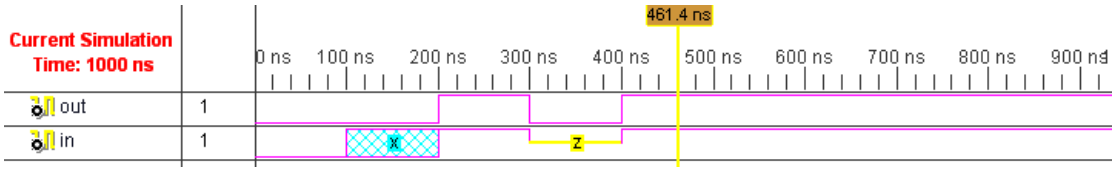


图 8-54　不确定输入转换模块的仿真结果

8.8 面向硬件的设计思维

本书内容从第 1 章开始就在讲解 Verilog HDL 是一种硬件描述语句，反复强调可综合设计代码和仿真代码的区别。在讲述了大部分可综合设计难点后，本节简要总结面向硬件的设计思维，包括硬件设计模式、程序的执行顺序以及时钟是电路控制者等内容。

8.8.1　基本的硬件设计模式

当阅读到这里时，读者应该强烈地意识到 Verilog HDL 语言同软件语言（如 C，C++等）有本质区别。虽然 Verilog HDL 语言采用了 C 语言形式的硬件抽象，但是它的最终实现结果是芯片内部的实际电路。所以评判一个 Verilog HDL 设计优劣的最终标准是：电路功能以及所达到的性能（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的更流畅、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。

因此，在 Verilog HDL 代码编写中不需要追求代码的整洁、简短。合理的编码方法是：首先理解要实现的硬件电路，对该部分硬件的结构与连接十分清晰，然后再用可综合的 HDL 语句表达出来即可。而不能凭空地去写代码，只有存在的电路才是物理可实现的。

在没有 HDL 语言之前，所有的设计都需要通过原理图输入的方式来完成，必须要求先构思电路雏形，然后才能去实现。Verilog HDL 语言的出现只是为我们增加了一种设计工具，可以从更高的层次来描述，但是要设计的电路本质并没有改变。因此，并不是说有了 HDL 语言就可以放弃硬件电路的基本知识。相反，只有深入地理解了电路本身，才能通过 HDL 语言完成高质量的设计。

例如，在实际中没有只工作两次就停下来的 D 触发器，因此下面的语句就是不能综合的，不可用于设计硬件。

```
repeat(2) @(possedge clk)
    d <= x;
```

至此可以得到一个基本结论：面向硬件的设计模式就是要从电路本身的特征和行为来编写 Verilog HDL 代码。

8.8.2 程序执行顺序

和 C/C++等顺序编程语言不同，HDL 语言用于电路描述，其代表着门电路和触发器电路。在任何时刻，只要 FPGA 上电，芯片内部各个模块将同时工作，不会因为各个模块之间的先后不同而存在执行顺序的差异。这需要设计人员以并行思维来考虑算法结构。

模块内部的执行顺序比较复杂，在优秀的设计中模块内部是并行执行的，即使在“begin … end”语句内部。这里，不少读者可能存在疑问，“begin … end”语句不是串行执行的模块吗？下面对此进行详细解释。

“begin … end”之间存在阻塞赋值和非阻塞赋值两种赋值方式，如果使用不当会存在冒险和竞争现象，必须按照下面两条准则[3]：（1）在描述组合逻辑的 always 块中使用阻塞赋值，则综合组合逻辑的电路结构；（2）在描述时序逻辑的 always 块中使用非阻塞赋值，则综合时序逻辑的电路结构。在组合逻辑中，阻塞赋值只与电平有关，往往和触发沿没有关系，可以将其看成并行执行的；在时序逻辑中，非阻塞赋值是并行执行的；因此，优秀的 HDL 设计，其内部语句也是并行执行的。为了便于读者理解，给出例 8-26 予以说明。

例 8-26：给出 Verilog HDL 语句并行执行特点的实例说明。

```
module   para_demo(clk, reset, a, b);
    input          clk;
    input          reset;
    input    [3:0] a;
    output   [3:0] b;

    reg      [3:0] tempa1, tempa2, b;

  always @(posedge clk) begin
     if(!reset) begin
            tempa1 <= 0;
            tempa2 <= 0;
            b <= 0;
```

```
        end
        else begin
                tempa1 <= a + 1'b1;
                tempa2 <= tempa1 + 1'b1;
                b <= tempa2 + 1'b1;
        end
    end

endmodule
```

上述程序在 ISE 中综合后的 RTL 级结构图如图 8-55 所示，可以看出，3 级累加操作采用 3 个乘法器来实现，每级累加都是并行执行的，只存在数据流达到先后的差异。

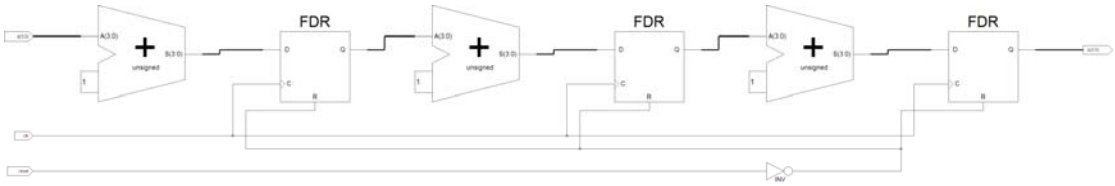


图 8-55　非阻塞赋值实例的 RTL 级结构示意图

在 ISE Simulator 中完成仿真，其结果如图 8-56 所示。其中，信号 tempa2 并不是等到本次“a+1”的执行结果赋给 tempa1 后才执行“tempa1+1”，而是在执行“a+1”的同时执行“tempa1+1”，其中的 tempa1 是上一次“a+1”的结果。同样的过程还可以在信号 b 的赋值中看到。因此，当 a 的值为 9 时，b 的值为 10，而不是 12。

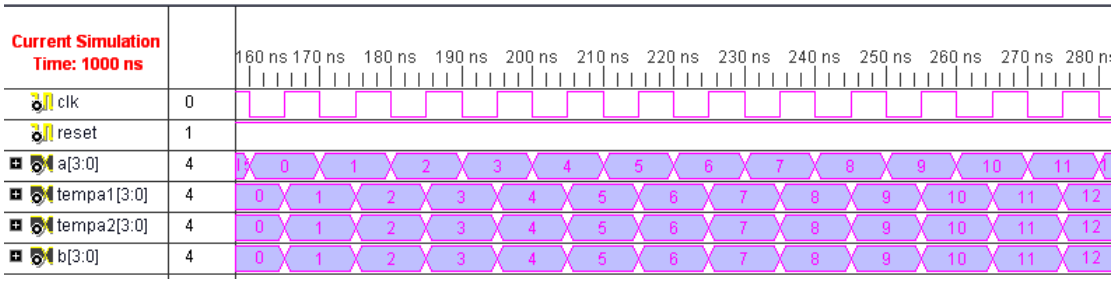


图 8-56　非阻塞赋值实例的仿真结果

8.8.3　时钟是时序电路的控制者

上文解释了 HDL 程序的并行性，但在设计中需要像 C/C++语言的串行控制功能，如先接收外部配置指定，然后接收数据并完成模块内部配置，再将配置结果反馈到外部，这需要通过时间的精确定位来获取严格的先后关系。那么怎么来实现呢？

其实很简单，假设全部事件需要 5 个时钟周期，那么利用一个周期为 5 的循环计数器来实现。在计数器为 1 的时候，完成事件 1；在计数器为 2 的时候，完成事件 2；……如此循环即可。总结起来就是按照时钟节拍来完成串行控制。当然，这样的电路在 FPGA 资源的利用上是存在浪费的，因为在执行事件 1，用于执行事件 2，3，4，5 的逻辑处于等待状态，但其却始终占用着逻辑资源。

例 8-27：时钟执行控制器的实例。

```
module clk_model(clk, reset, a1, y1, y2);
  input clk;
```

```
    input reset;
    input [7:0] a1;
    output [7:0] y1, y2;

    reg [7:0] y1, y2;
    reg [2:0] cnt;
    always @(posedge clk) begin
        if(!reset) begin
                cnt <= 0;
                y1 <= 0;
                y2 <= 0;
        end
        else begin
              if(cnt == 2'b11)
                 cnt <= 2'b00;
              else
                 cnt <= cnt + 1'b1;
             case(cnt)
                  2'b00: begin
                            y1 <= a1 + 1'b1;
                            y2 <= y2;
                  end
                  2'b01: begin
                            y1 <= y1;
                            y2 <= y1 + 1'b1;
                  end
                  default: begin
                            y1 <= y1;
                            y2 <= y2;
                  end
              endcase
       end
   end

endmodule
```

上述程序在 ISE 中综合后，其 RTL 级结构图如图 8-57 所示。可以看到生成了 3 个加法器，即为变量 y1、y2 以及 cnt 的计算都生成了加法器，但 y1 与 y2 的加法器却在时钟信号 clk 的控制下工作，其工作时间是不连续的，每 4 个周期计算 1 次，且 y1 的计算永远在 y2 的计算之前。

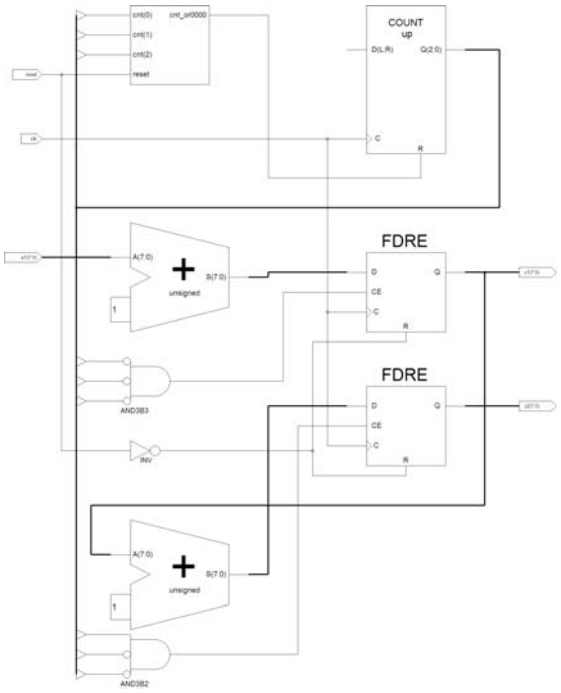


图 8-57　时钟控制实例的 RTL 级结构示意图

上述程序在 ISE Simulator 中的仿真结果如图 8-58 所示。从中可以看出，y1 和 y2 的周期是时钟的 4 倍，y1 的值等于 cnt 为 0 时所采样的输入信号 a1 的值加 1，y2 的值等于 y1 的值加 1，且比 y1 的值延迟了一个时钟周期。

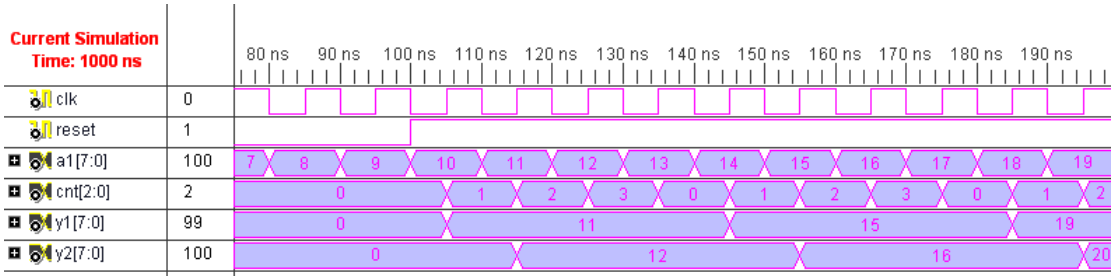


图 8-58　时钟控制实例的仿真结果

8.9 本章小结

本章主要介绍 Verilog HDL 可综合设计的难点。首先说明了综合逻辑、时序逻辑的电路特征以及 Verilog HDL 语言描述，并说明了时序电路的时钟选择策略。其次，介绍了时序电路中同步电路和异步电路的特点，以及其相应的 Verilog HDL 描述；第三，给出了阻塞赋值和非阻塞赋值的详细说明，说明了在组合逻辑中采用阻塞赋值以及在时序逻辑采用非阻塞赋值的本质。接下来，介绍了双向端口、锁存器以及消除电路中的不确定输入的 Verilog HDL 实现。最后详细介绍了面向硬件的设计思维。通过本章的阅读，读者可体会到硬件开发和软件开发的不同，并在实际操作中能够时刻做到“胸有成竹”。本章是本书的重点，也是难点之一，希望读者仔细阅读，反复体会。

8.10 思考题

1．什么是组合逻辑电路？如何用 Verilog HDL 语言来完成组合逻辑电路？

2．解释组合逻辑中竞争和冒险现象以及产生原因？如何在 Verilog HDL 语言设计的过程中避免组合逻辑设计的竞争和冒险？
3．什么是时序逻辑电路？如何用 Verilog HDL 语言来完成时序逻辑电路？
4．简要说明时序电路中时钟信号的选择策略。
5．分别给出一个同步时序电路和异步时序电路的 Verilog HDL 描述实例。
6．什么是时序电路的同步复位和异步复位？
7．阻塞赋值语句和非阻塞赋值语句的应用原则是什么？
8．什么是双向端口？如何通过 Verilog HDL 完成相关仿真？
9．什么是锁存器？为什么人们在设计中对锁存器“闻虎色变”？
10．为什么要消除设计中的不确定性输入？如何通过 Verilog HDL 语言完成？
11．简要回答 Verilog HDL 语言的执行顺序？
12．什么是时序电路的控制器？

第 9 章　高级逻辑设计思想与代码风格

在前面介绍了基本的 Verilog HDL 语法之后，本章主要讨论一些实际应用中的相对高级且必要的设计规律。当然，设计规律是一个面很宽的话题，需要设计者更多地在实际操作中积累经验。但在设计中存在很多内在规律，总结并掌握这些规律对于深刻理解 Verilog HDL 语言编程有着重要的意义。本章内容主要列出一些常用的设计方法，以提纲式的方式为读者后续的实践列出关键点，更多的需要读者自行实践，有意积累。

9.1 通用指导原则

9.1.1　面积和速度的互换原则

这里的“面积”主要是指设计所占用的 FPGA 逻辑资源数目，即利用所消耗的触发器（FF）和查找表（LUT）来衡量。“速度”是指在芯片上稳定运行时所能够达到的最高频率。面积和速度这两个指标始终贯穿着 FPGA 的设计，使设计质量评价的最终标准。本节主要讨论一个基本原则：面积和速度的互换。

面积和速度是一对对立和统一的矛盾体。一方面，要提高速度，就需要消耗更多的资源，即需要更大的面积；另一方面，为了减少面积，就需要降低处理速度。所以既要提高速度，又要减少面积，是不可能同时实现的。但在实际中，总是存在二者之间的平衡，也意味着二者可以互换。

面积和速度互换的具体操作很多，比如模块复用、乒乓操作、串并转换以及流水线操作。本节对面积和速度互换只做一个简略介绍，具体应用及方法将在后面的章节中逐步提及。其中，流水线的操作比较重要，将在 9.3.1 节详细介绍。这里主要说明了串并转换的应用利用这一互换原理。

例 9-1：使用串并转换乘法器来说明“面积换速度”的思路。

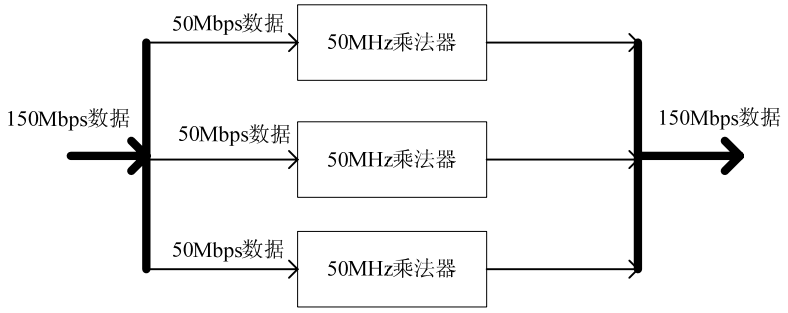


图 9-1　串并转换的示意图

如图 9-1 所示，假设数据速率是乘法器模块处理速度的 3 倍，那么由于乘法器模块的数据吞吐量满足不了要求，在这种情况下，就利用面积换速度的思想，复制 3 个乘法器模块。首先将输入数据进行串并转换，然后利用这 3 个模块并行处理所分配到的数据，最后将处理结果并串转换，达到数据速率的要求。上述例子只是对面积换速度思想的一个简单举例，在具体操作过程中，还涉及很多方法和技巧，需要读者从大量实例中自己体会。在后续章节中，还会逐步地应用这一互换思想。

9.1.2 模块划分原则

自顶向下的层次化设计方法中最关键的工作就是模块划分，将一个很大的工程合理地划分为一系列功能独立的部分，且具备良好的协同设计能力，以便快速地实现整个设计。此外，模块划分直接影响到所需的逻辑资源、时序要求以及实现效率。其基本的原则如下所示：

1．信息隐蔽、抽象原则

上一层模块只负责为下一层模块的工作提供原则和依据，并不规定下层模块的具体行为，以保证各个模块的相对独立性和内部结构的合理性，使得模块之间层次分明，易于理解、实施和维护。

2．明确性原则

每个模块必须功能明确，接口含义明确，禁止多重功能和无用接口，整个设计过程中应具有统一的命名规范。

3．模块时钟域区分原则

在设计中，经常采用多时钟设计，必然存在亚稳态，如果处理不当，将会给设计的可靠性带来极大的隐患。这里需要通过异步 FIFO 以及双口 RAM 来建立接口，尽量避免让信号直接跨越不同时钟域。此外，由于时钟频率不同，其时序约束需求也不同，可以将低频率时钟域划分到同一模块，如多时钟路径等，则可以让综合器尽量节约面积。

4．资源复用原则

在 HDL 设计中，要将可以复用的逻辑或者相关逻辑尽量放在同一模块，不仅可以节省硬件资源，还有利于优化关键路径。但在实际中，不能为了资源复用而将存储器逻辑混用。因为 FPGA 芯片生产商提供了各类存储器的硬件原语，尽量使用原语，而不是使用查找表和寄存器来实现原语的功能，才能将设计所需资源最小化。从概念上讲，模块越大越利于资源共享和复用，但庞大的模块在仿真验证时需要较长的时间和较高的 PC 机配置，不利于修改，无法使用增量设计模式。

5．同步时序模块的寄存器划分原则

即在设计时，尽量将模块中的同步时序逻辑输出信号以寄存器的形式送出，便于综合工具区分时序和组合逻辑；并且时序输出打寄存器符合流水线设计思想，能工作在更高的频率，极大地提高模块吞吐量。

9.2 代码风格

目前，FPGA 的规模越来越大，HDL 代码的功能越来越复杂，规模也越来越大，代码的可移植性以及时序、资源等指标的要求也越来越高，并且设计的稳定性也是越来越被关注的方面。与此同时，EDA 也越来越智能化，同一个程序经过不同的工具分析后可能会产生不同的结果。因此，代码的书写风格极大地影响着设计。优秀的代码风格可以减少错误，提高电路性能，达到事半功倍的效果；如果使用不当则会有南辕北辙之后果。

9.2.1 代码风格的含义

代码风格有两层含义：其一是 Verilog 的代码书写习惯，包括模块、变量命名以及换行格式等，主要为了提高代码的可读性；另一个则是设计风格，对于一特定电路，用哪一种形式的语言描述，才能将电路描述得更准确，综合以后产生的电路更为合理。

代码设计风格有通用设计风格和专用设计风格两大类，前者指不依赖于 FPGA 开发的 EDA 软件工具和 FPGA 芯片类型，仅仅是从 HDL 语言出发的代码风格；后者指和开发软件以及硬件芯片密切相关的代码风格，不仅需要关注 EDA 软件在语法细节上的差异，还要紧密依赖于固有的硬件结构。显然，前者具有较好的通用性，但性能未必最优。在使用时，如

果有后续的进一步开发，建议使用通用风格；否则就可采用后者，以便极大地挖掘芯片潜力。

9.2.2 通用的代码设计风格

代码设计风格主要是针对 RTL 级代码而言的，不包括其他的描述层次。RTL 级代码的评判标准很多，包括时序性能、面积、可重用性、功耗以及所有的 EDA 工具等诸多方面。此外，还需考虑要运行设计的目标器件的特点是否可以通过程序发挥出来。根据上述这些目标的组合和优先级设计，可以派生很多不同的设计原则，包括面积换速度、速度换面积、流水线、以及逻辑合并与拆分等。当然，这些原则是 RTL 级代码设计的灵魂，也是常用的设计思想，本书将在 9.3 节详细介绍。

此外，在 RTL 代码设计中，也有许多细节处理技巧，其重要性不比上述设计原则低，特别是在要求较高的代码设计中，包括信号的同步设计优先、反相操作、触发器资源分配技术等，下面对其进行介绍。

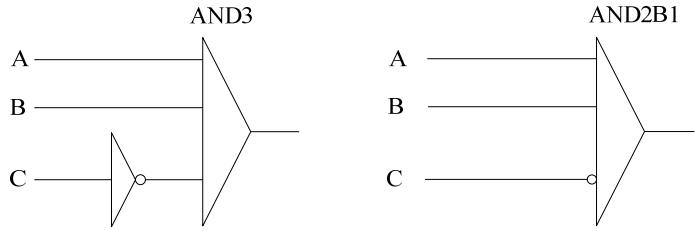
1．同步设计优先

同步设计的优点和实现方法已经在8.2节进行了详细讨论，这里就不再重复描述。

2．信号反相的处理策略

在处理反相信号时，设计时应尽可能地遵从分散反相原则。即应使用多个反相器分别反相，每个反相器驱动一个负载，这个原则无论对时钟信号还是对其它信号都是适用的。因为在FPGA设计中，反相是被吸收到CLB或IOB中的，使用多个反相器并不占用更多的资源，而使用一个反相器将信号反相后驱动多个负载却往往会多占资源，而且延迟也增加了。

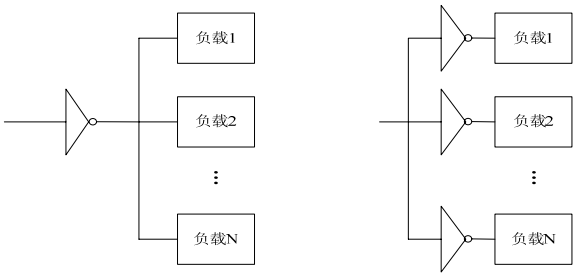
首先，如果输入信号需要反相，则应尽可能地调用输入带反相功能的符号，而不是用分离的反相器对输入信号进行反相。例如，如图9-2所示电路是对逻辑 $Y= AB\overline{C}$ 的两种设计电路。两者对比，最优的方案为采用如图(b)所示电路，即直接调用AND2B1，而不要用如图(a)所示的电路，用分离的非门对输入信号C反相后，再连接到AND3的输入。因为在前一种作法中，由于函数发生器用查表方式实现逻辑，C的反相操作是不占资源的，也没有额外延迟；而后一种作法中，C的反相操作与AND3操作可能会被分割到不同的逻辑单元中实现，从而消耗额外的资源，增加额外的延迟。



(a) 非优化反相设计　　(b) 优化反相设计

图 9-2　两种反相设计电路比较

其次，如果一个信号反相后驱动了多个负载，则应将反相功能分散到各个负载中实现，如图9-3(b)图所示。而不能采用传统TTL电路设计，采用集中反相驱动多个负载来减少所用的器件的数量，如图9-3(a)图所示。因为在FPGA设计中，集中反相驱动多个负载往往会多占一个逻辑块或半个逻辑块,，而且延迟也增加了。分散信号的反相往往可以与其它逻辑在同一单元内完成而不消耗额外的逻辑资源。



(a) 非优化集中反相　　(b) 优化分散反相

图 9-3　两种反相驱动电路比较

3．触发器资源的分配技术

由于FPGA是一种触发器密集型可编程器件，因此系统的逻辑设计就应该充分利用触发器资源，尽可能降低每个组合逻辑操作的复杂度。

首先，应尽量使用库中的触发器资源。因为FPGA触发器资源丰富，而且开发系统在划分逻辑块时，对D触发器等元件直接利用CLB中的触发器；而对自建触发器则认为是组合电路，需要使用CLB中的组合逻辑电路构成，这样既占用更多的CLB，又浪费了CLB的触发器资源。将两种方法进行比较，每使用一个自建D触发器比使用库中D触发器的电路多占用二至三个CLB。

其次，在设计状态机时，应该尽量使用ONE-HOT状态编码方案，不用二进制状态编码方案。ONE-HOT状态编码方案是表示每个状态由1位触发器来表示，而二进制状态编码方案是用LgN/Lg2位触发器来表示N个状态。由于二进制状态编码的稳定度较低，ONE-HOT状态编码方案对于触发器资源丰富的FPGA芯片十分适用。

9.2.3　通用的代码书写风格

1．信号命名规则

信号命名规则在团队开发中占据着重要地位，统一、有序的命名能大幅减少设计人员之间的冗余工作，还可便于团队成员代码的查错和验证。比较著名的信号命名规则当推 Microsoft 公司的“匈牙利”法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。例如所有的字符变量均以 ch 为前缀，若是常数变量则追加前缀 c。信号命名的整体要求为：命名字符具有一定的意义，直白易懂，且项目命名规则唯一。对于 HDL 设计，设计人员还需要注意以下命名规则。

（1）系统级信号的命名

系统级信号指复位信号，置位信号，时钟信号等需要输送到各个模块的全局信号。系统信号以字符串 sys 或 syn 开头；时钟信号以 clk 开头，并在后面添加相应的频率值；复位信号一般以 rst 或 reset 开头；置位信号为 st 或 set 开头。典型的信号命名方式如下所示：

```
wire [7:0] sys_dout, sys_din;
wire clk_32p768MHz;
wire reset;
wire st_counter;
```

（2）低电平有效的信号命名

低电平有效的信号后一律加下划线和字母 n。如：

```
wire SysRst_n;
wire FifoFull_n;
```

（3）经过锁存器锁存后的信号

经过锁存器锁存后的信号，后加下划线和字母 r，与锁存前的信号区别。如：

信号 CpuRamRd 信号，经锁存后应命名为 CpuRamRd_r。

低电平有效的信号经过锁存器锁存后，其命名应在_n 后加 r。如：

CpuRamRd_n 信号，经锁存后应命名为 CpuRamRd_nr

多级锁存的信号，可多加 r 以标明。如：

CpuRamRd 信号，经两级触发器锁存后，应命名为 CpuRamRd_rr。

2．模块命名规则

HDL 语言的模块类似于 C 语言中的函数，可采用 C 语言函数的大多数规则。模块的命名应该尽量用英文表达出其完成的功能。遵循动宾结构的命名法则，函数名中动词在前，并在命名前加入函数的前缀，函数名的长度一般不少于 2 个字母。HDL 模块的命名还需要考虑以下情况：

（1）模块的命名规则

在系统设计阶段应该为每个模块进行命名。命名的方法是，将模块英文名称的各个单词首字母组合起来，形成 3 到 5 个字符的缩写。若模块的英文名只有一个单词，可取该单词的前 3 个字母。各模块的命名以 3 个字母为宜。例如：

Arithmatic Logical Unit 模块，命名为 ALU。

Data Memory Interface 模块，命名为 DMI。

Decoder 模块，命名为 DEC。

（2）模块之间接口信号的命名

所有变量命名分为两个部分：第一部分表明数据方向，其中数据发出方在前，数据接收方在后；第二部分为数据名称。两部分之间用下划线隔离开。第一部分全部大写，第二部分所有具有明确意义的英文名全部拼写或缩写的第一个字母大写，其余部分小写。举例：

wire CPUMMU_WrReq;

下划线左边是第一部分，代表数据方向是从 CPU 模块发向存储器管理单元模块（MMU）。下划线右边 Wr 为 Write 的缩写，Req 是 Request 的缩写。两个缩写的第一个字母都大写，便于理解。整个变量连起来的意思就是 CPU 发送给 MMU 的写请求信号。模块上下层次间信号的命名也遵循本规定。若某个信号从一个模块传递到多个模块，其命名应视信号的主要路径而定。

（3）模块内部信号

模块内部的信号由几个单词连接而成，缩写要求能基本表明本单词的含义；单词除常用的缩写方法外（如：Clock->Clk，Write->Wr，Read->Rd 等），一律取该单词的前几个字母（如：Frequency->Freq，Variable->Var 等）；每个缩写单词的第一个字母大写；若遇两个大写字母相邻，中间添加一个下划线（如 DivN_Cntr）；举例：

SdramWrEn_n;

FlashAddrLatchEn;

3．代码格式规范

（1）分节书写格式

各节之间加 1 到多行空格。如每个 always，initial 语句都是一节。每节基本上完成一个特定的功能，即用于描述某几个信号的产生。在每节之前有几行注释对该节代码加以描述，至少列出本节中所描述信号的含义。

行首不要使用空格来对齐，而是用 Tab 键，Tab 键的宽度设为 4 个字符宽度。行尾不要有多余的空格。

（2）注释的规范

使用//进行的注释行以分号结束；使用/* */进行的注释，/*和*/各占用一行，并且顶头；例如：

```
// Edge detector used to synchronize the input signal；
```

对于函数，应该从“功能”，“参数”，“返回值”、“主要思路”、“调用方法”、“日期”六个方面用如下格式注释：

```
// 程序说明开始
// ==========================================================================//
// 功能：      完成两个输入数的相加。
// 参数：      strByDelete,strToDelete
// 输入参数
// 输出参数
// 主要思路：本算法主要采用 2 级流水线完成相加
// 日期：起始日期，如：2008/8/21.9:40--2008/8/23.21:45
// 版本：
// 程序编写人员：
// 程序调试记录：
// ==========================================================================//
// 模块说明结束
```

此外，在注释说明中，需要注意以下细节：

- 在注释中应该详细说明模块的主要实现思路，特别要注明自己的一些想法，如果有必要则应该写明对想法产生的来由。
- 在注释中详细注明函数的适用方法，对于输入参数的要求以及输出数据的格式。
- 在注释中要强调调用时的危险方面，可能出错的地方。
- 对日期的注释要求记录从开始编写模块到模块测试结束之间的日期。
- 对模块注释开始到模块命名之间应该有一组用来标识的特殊字符串。如果算法比较复杂，或算法中的变量定义与位置有关，则要求对变量的定义进行图解。对难以理解的算法能图解尽量图解。

（3）空格的使用

不同变量，以及变量与符号、变量与括号之间都应当保留一个空格。Verilog HDL 语言关键字与其它任何字符串之间都应当保留一个空格。如：

```
always @ ( ..... )
```

使用大括号和小括号时，前括号的后边和后括号的前边应当留有一个空格。逻辑运算符、算术运算符、比较运算符等运算符的两侧各留一个空格，与变量分隔开来；单操作数运算符例外，直接位于操作数前，不使用空格。使用//进行的注释，在//后应当有一个空格；注释行的末尾不要有多余的空格。例：

```
assign SramAddrBus = { AddrBus[31:24], AddrBus[7:0] };
assign DivCntr[3:0] = DivCntr[3:0] + 4'b0001;
assign Result = ~Operand;
```

（4）begin…end 的书写规范

同一个层次的所有语句左端对齐；initial、always 等语句块的 begin 关键词跟在本行的末尾，相应的 end 关键词与 initial、always 对齐，并且在 end 后面添加注释标明结束；这样做的好处是避免因 begin 独占一行而造成行数太多；如：

```
always @ ( posedge SysClk or negedge SysRst ) begin
    if( !SysRst ) DataOut <= 4'b0000;
    else if( LdEn ) begin
        DataOut <= DataIn;
        End
    else
        DataOut <= DataOut + 4'b0001;
end   //end always 模块
```

不同层次之间的语句使用 Tab 键进行缩进，每加深一层缩进一个 Tab；在 endmodule，endtask，endcase 等标记一个代码块结束的关键词后面要加上一行注释说明这个代码块的名称。

4．模块调用规范

如 3.2.2 节所述，在 Verilog 中，有两种模块调用的方法，一种是位置映射法，严格按照模块定义的端口顺序来连接，不用注明原模块定义时规定的端口名，其语法为：

```
被调用模块名 用户自定义调用名
(连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名,…);
```

另一种为信号映射法，即利用“.”符号，表明原模块定义时的端口名，其语法为：

```
被调用模块名 用户自定义调用名
        (.端口 1 信号名(连接端口 1 信号名),
        .端口 2 信号名(连接端口 2 信号名),
        .端口 3 信号名(连接端口 3 信号名),…);
```

显然，信号映射法同时将信号名和被引用端口名列出来，不必严格遵守端口顺序，不仅降低了代码易错性，还提高了程序的可读性和可移植性。因此，在良好的代码中，严禁使用位置调用法，全部采用信号映射法。

9.2.3 Xilinx 专用代码设计风格

专用代码风格是指从FPGA器件特征角度考虑，尽可能利用芯片结构以及内嵌的底层宏单元，以取得最佳的综合和实现效果。对于同一个设计，使用适合于FPGA体系结构特点的优化设计方法，可以大大提高芯片利用率和设计实现速度。

1. 时钟信号的分配策略

时钟分配网络是FPGA芯片中的特殊布线资源，由特定的引脚和特定的驱动器驱动，只能驱动芯片上触发器的时钟输入端或除了时钟输入端外有限的一些负载，其目的是为设计提供小延迟偏差和扭曲可忽略的时钟信号。

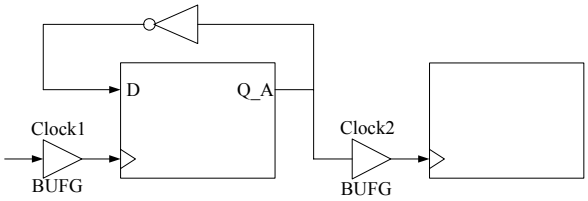
首先，使用全局时钟，可为信号提供最短的延时和可忽略的扭曲。全局网线由全局缓冲器BUFG来驱动，使用BUFG时，时钟信号经BUFG驱动后通过长线同时接到每个触发器的时钟端，减少传输延迟。如不使用BUFG，时钟信号按一般布线连接到不同CLB，时钟信号到达各触发器的延迟不一致，使同步时序电路出现不同步的现象。

其次，FPGA特别适合于同步电路设计，尽可能减少使用的时钟信号种类。如TTL电路

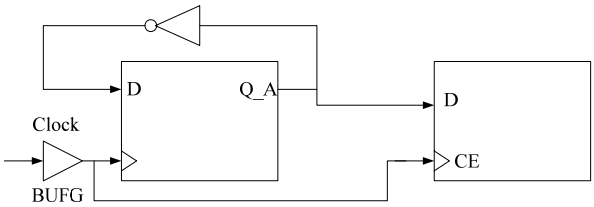
设计中，经常采用的由组合逻辑生成多个时钟分别驱动多个触发器的设计方法对FPGA的设计不适用。因为这样做使得时钟种类很多，不能利用专用的时钟驱动器和专用的时钟布线资源，时钟信号只能由通用的布线资源拼凑而成，各个负载点上的时钟延迟偏差很大，会引起数据保持时间问题，降低工作速度。

第三，减小时钟摆率的另一种更有效方法是使用一个时钟信号，生成多个时钟使能信号，分别驱动触发器的时钟使能端，所有触发器的数据装入都由同一个时钟控制，但只有时钟使能信号有效的触发器才会装入数据，时钟使能信号无效的触发器则保持数据。这种方法充分发挥了FPGA器件体系结构的优势。

图9-4所示电路为分频器的一般设计和优化设计的对比。两种设计方法相比较，图(a)电路使用两个全局缓冲，实现两个触发器的异步控制。图(b)电路将异步控制转化为同步控制，只占用一个全局缓冲，利用时钟使能信号CE控制触发器的动作。因此图(b)电路占用更少的全局时钟资源，而且使用一种时钟信号更利用同步控制和减小时钟摆率。



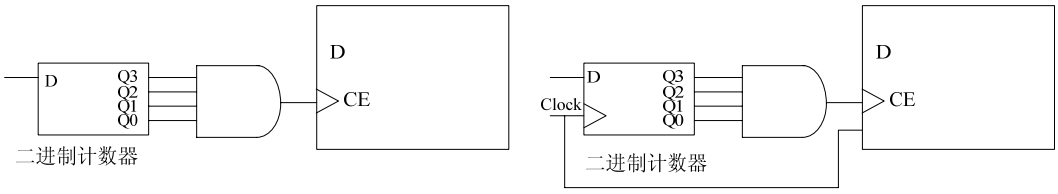
(a) 分频器的普通设计



(b) 分频器的优化设计

图 9-4　分频器两种设计电路的比较

第四，要避免时钟信号毛刺。由图9-5所示电路可知，当二进制计数器从0111向1000变化时，必然会出现一个1111的过渡过程，D触发器的时钟就会产生毛刺，毛刺出现的时间很短，但对于高速处理来讲，足以使触发器误动作。图(b)所示电路就是使用同步时钟，并使用时钟使能CE端避免时钟信号毛刺的设计电路。



(a) 会产生毛刺的传统设计　　(b) 能消除毛刺的优化电路

图 9-5　两种设计电路的比较

2．硬核资源优先原则

硬核资源是相对于基于逻辑资源的软核而言的，为了提高FPGA性能，芯片生产商在芯片内部集成了一些专用的硬核，等效于ASIC电路芯片。Xilinx的主流芯片都集成了硬核的块RAM、硬核乘加器，在高端产品还集成了Power PC系列CPU，还内嵌了吉比特收发器（MGT）模块。这些硬件资源的使用，不占用任何逻辑资源，并且具备更高的时序性能和更低的功耗，

因此在设计中首先考虑要使用硬核资源。

3．SRL16的使用

SRL16是一种基于查找表（LUT）的移位寄存器，可用于构建高密度DSP结构（如滤波器），能够大幅削减硬件资源。在Virtex-5芯片中为6输入的查找表，在其余系列芯片中都为4输入查找表。其位宽（B）和深度（D）可以任意配置，最大的特点就是占用Slice资源特别少。其占用Slice资源 M 的计算公式为：

$$M = B(R[D/16]+1)$$

其中，R[]函数的意义是取整。从上式可以看出，深度为 1 的移位寄存器和深度为 16 的移位寄存器所占用的 Slice 资源是一样的。

例 9-2：使用 SRL16 生成位宽为 8，深度为 16 的移位寄存器。

```
module lut_ram(clk, d, q);
   input clk;
   input [7 : 0] d;
   output [7 : 0] q;

   srl16_based_ram srl16_based_ram(
   .clk(clk),
   .d(d),
   .q(q)
   );
endmodule
```

其中 srl16_based_ram 例化了 Xilinx 提供的 RAM_Based_Shiftreg 的 IP Core，使用 SRL16 结构完成了移位寄存器。上述程序在 ISE 中综合后，得到的 RTL 结构如图 9-6 所示。

图 9-6　SRL16 结构移位寄存器的 RTL 结构示意图

在 ISE Simulator 中完成仿真，其结果如图 9-7 所示，，可以看出输入、输出数据相差 16。表明 lut_ram 将输入数据延迟 16 个周期。

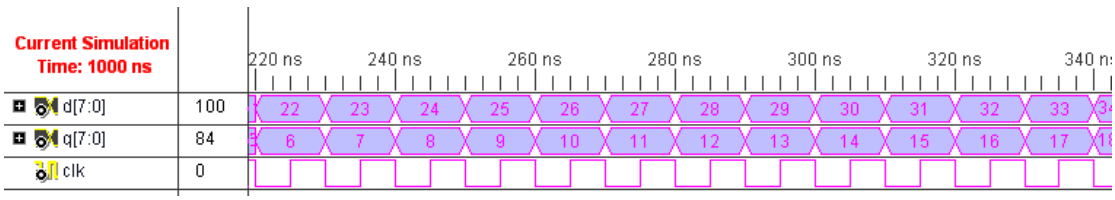


图 9-7　SRL16 结构移位寄存器的仿真结果示意图

9.3 常用的设计思想与代码设计风格

9.3.1　流水线技术原理和 Verilog HDL 实现

1．流水线原理

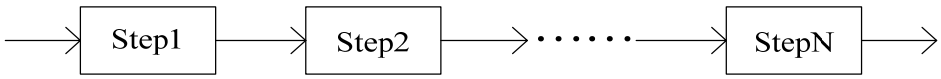
在高速通信系统设计中，如何提高系统的工作速度是系统设计成败的关键问题。在通常情况下，提高系统的工作速度有两种方法：其一是采用并行方式设计。传统上，设计方式常采用串行的方式，而利用串行方式设计的电路系统的运行速度与每个模块之间的延时是直接相关的。为了减少模块间的延时，就要采用并行方式来设计电路。其二是采用流水线式设计方式。

本节主要介绍流水线设计方法及其应用，并对流水线式设计方法和普通方法进行了比较。所谓流水线处理，如同生产装配线一样，将操作执行工作量分成若干个时间上均衡的操作段，从流水线的起点连续地输入，流水线的各操作段以重叠方式执行。这使得操作执行速度只与流水线输入的速度有关，而与处理所需的时间无关。这样，在理想的流水操作状态下，其运行效率很高。

如果某个设计的处理流程分为若干的步骤，而且整个数据处理是单流向的，即没有反馈或者迭代运算，前一个步骤的输出是下一个步骤地输入，则可以采用流水线设计方法来提高系统的工作频率。

流水线设计的结构示意图如 9-8 所示。

(a) 没有流水线的结构示意图



(b) 流水线的结构示意图

图 9-8 有（无）流水线设计的结构示意图

如果不采用流水线，输入数据需要等到前一个数据经过处理单元中所有运算后才能进入处理模块；而采用流水线模块后，输入数据只需要经过一级流水线处理后，就可以进入下一个流水线处理模块，数据处理时间减小了 N 倍。如果原来数据处理单元的吞吐量是 T，采用 N 级流水线后总的吞吐量为 NT。

流水线的基本结构是将适当划分的 N 个操作步骤串联起来。流水线操作的最大特点是数据流在各个步骤的处理，从时间上看是连续的；其操作的关键在于时序设计的合理安排、前后级接口间数据的匹配。如果前级操作的时间等于后级操作的时间，直接输入即可；如果前级操作时间小于后级操作时间，则需要对前级数据进行缓存，才能输入到后级；如果前级操作时间大于后者，则需要串并转换等方法进行数据分流，然后再输入到下一级。常用的流水线设计时序示意图如图 9-9 所示。

流水线处理方式之所以频率较高，是因为拆分、复制了处理模块，是面积换取速度思想的又一种具体体现。

2．流水线应用实例

在本节中，用 8 位全加法器作为实例，分别列举了非流水线方法、二级流水线方法和四级流水线方法。实现时，在每个全加法器之间加了一个锁存器，整个系统用同一个时钟。每一级的执行在时间上重叠起来。这样，整个系统的执行速度就加快了。

（1）非流水线实现方式

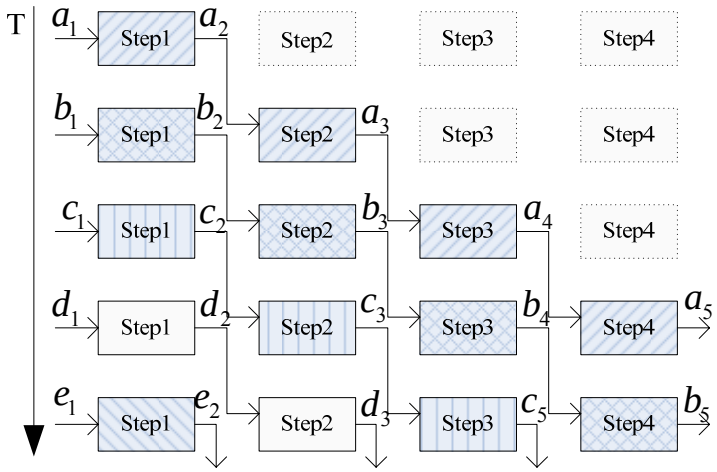


图 9-9　流水线设计时序

例 9-3：利用 Verilog HDL 语言实现一个 8 比特非流水线加法器。

```
module adder8 (cout ,sum ,clk ,cina ,cinb ,cin);
    input [7 :0 ]cina ,cinb;
    input clk ,cin;
    output [7 :0 ] sum;
    output cout;

    reg[7 :0 ]sum;
    reg cout ;

    always @(posedge clk)    begin //  时钟上升沿有效;
        {cout ,sum} = cina + cinb + cin ; // 8  位相加;
    end

endmodule
```

图 9-10 给出了程序在 ISE 中综合后的 RTL 结构图，其中只有一个 8 位加法器，没有额外的辅助电路，其运算时延是比较大的，只能用于低速设计。

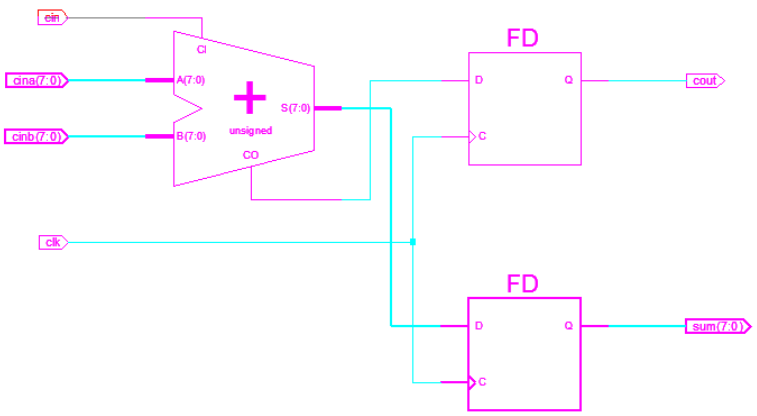


图 9-10　没有流水线结构的加法器 RTL 图

上述程序在 ISE Simulator 中的仿真结果如图 9-11 所示，可以看出，其正确完成了加法的功能，并且只有一个时钟的延迟。

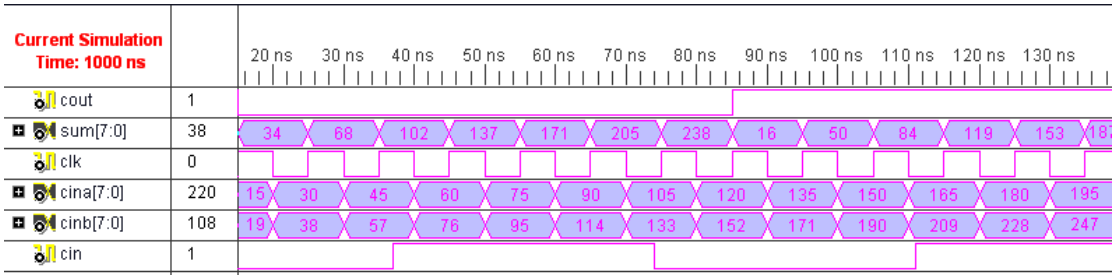


图 9-11　没有流水线结构加法器的仿真结果

（2）二级流水线实现方式

例 9-4：加法器的二级非流水线实现。

```
module adder8_2(cout ,sum ,clk ,cina ,cinb ,cin) ;
    input [7 :0]cina ,cinb;
    input clk ,cin;
    output [7 :0] sum;
    output cout;
    reg cout ;
    reg cout1 ;
    reg[3 :0]sum1;
    reg[7 :0]sum;

    always @(posedge clk) begin //LSB 4  位相加;
        {cout1 , sum1} = cina [3 : 0] + cinb [3 : 0] + cin ;
    end
    always @(posedge clk) begin    //高 4  位相加,并且将 8  位拼接起来;
        {cout ,sum} = {{cina[7],cina [7 :4]} +{cinb[7], cinb[7 :4] + cout1} ,sum1};
    end

endmodule
```

图 9-12 给出了程序在 ISE 中综合后的 RTL 结构图，其中只有 2 个 4 比特加法器，添加了较少的辅助逻辑，使加法的运算时延降低到对 4 个比特的处理，可以在中速应用中使用。

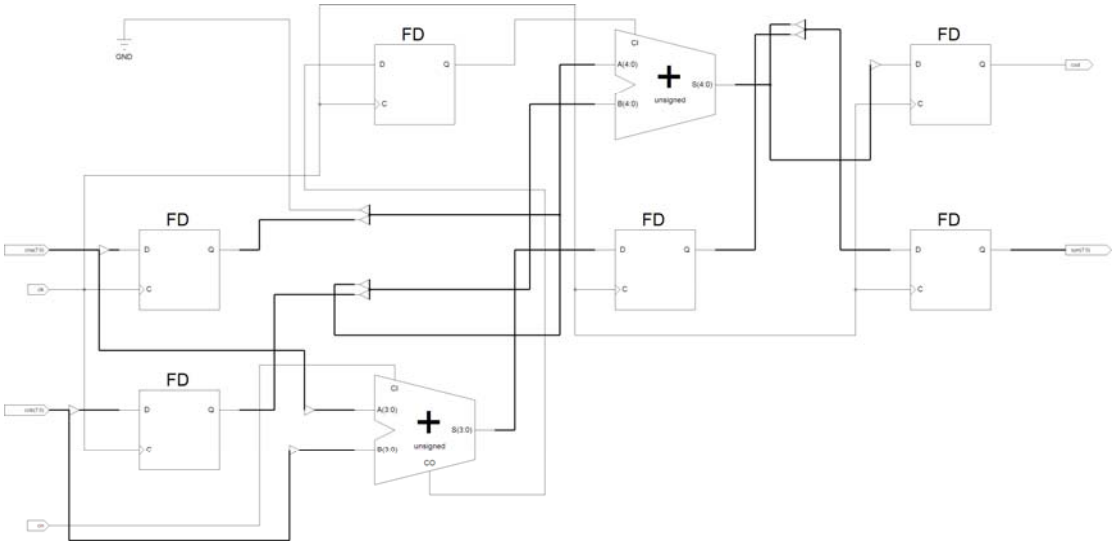


图 9-12　具有 2 级流水线结构加法器的 RTL 图

上述程序在 ISE Simulator 中的仿真结果如图 9-13 所示，可以看出，其正确完成了加法的功能，并且只有一个时钟的延迟。

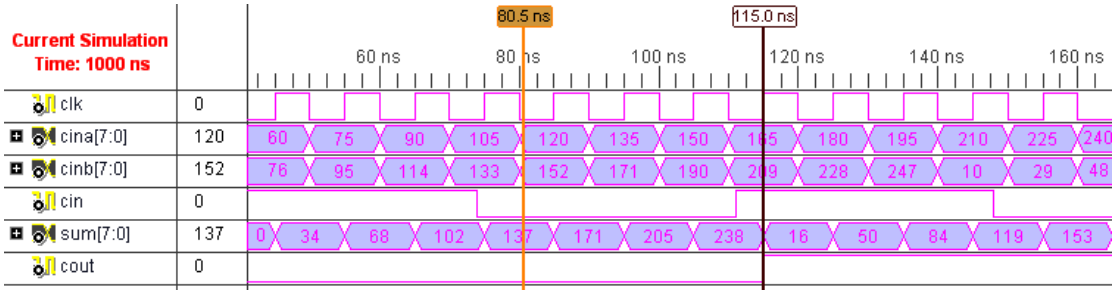


图 9-13　具有 2 级流水线结构加法器的仿真结果

（3）四级流水线实现方式

例 9-5：加法器的四级非流水线实现。

```
module adder8_4(cout ,sum ,clk ,cina ,cinb ,cin);
        input   [7:0]cina ,cinb;
        input   clk ,cin;
        output [7:0] sum;
        output cout;
        reg     [7:0] sum;
        reg     cout;

        reg     cout1, cout2, cout3;
        reg     [1:0] sum1;
        reg     [3:0] sum2;
        reg     [5:0] sum3;


        reg     [7:0] cina_t1, cinb_t1, cina_t2, cinb_t2, cina_t3, cinb_t3;

        always @(posedge clk) begin //  低二位相加
```

```
        {cout1, sum1} <= cina[1:0] + cinb[1:0] + cin;
         cina_t1 <= cina;
         cinb_t1 <= cinb;
end

always @(posedge clk) begin    //相加,并且将低 4  位拼接起来;
        {cout2, sum2} <= {{1'b0,cina_t1[3:2]} + {1'b0,cinb_t1[3:2]} +
                          {2'b00,cout1}, sum1};
         cina_t2 <= cina_t1;
         cinb_t2 <= cinb_t1;
end

always @(posedge clk) begin //相加,并且将低六位拼接起来;
        {cout3, sum3} <= {{1'b0,cina_t2[5:4]} + {1'b0,cinb_t2[5:4]} +
                          {2'b00,cout2}, sum2};
         cina_t3 <= cina_t2;
         cinb_t3 <= cinb_t2;
end

always @(posedge clk) begin //高 2 位相加,并且将 8 位拼接起来;
        {cout, sum} = {{1'b0,cina_t3[7:6]} + {1'b0,cinb_t3[7:6]} +
                       {2'b00,cout3}, sum3};
end

endmodule
```

图 9-14 给出了程序在 ISE 综合后的 RTL 结构图，其中有 4 个 2 位加法器，添加了较多的辅助逻辑，使加法的运算时延降低到对 2 个比特的处理，从而可以应用在高速设计中。

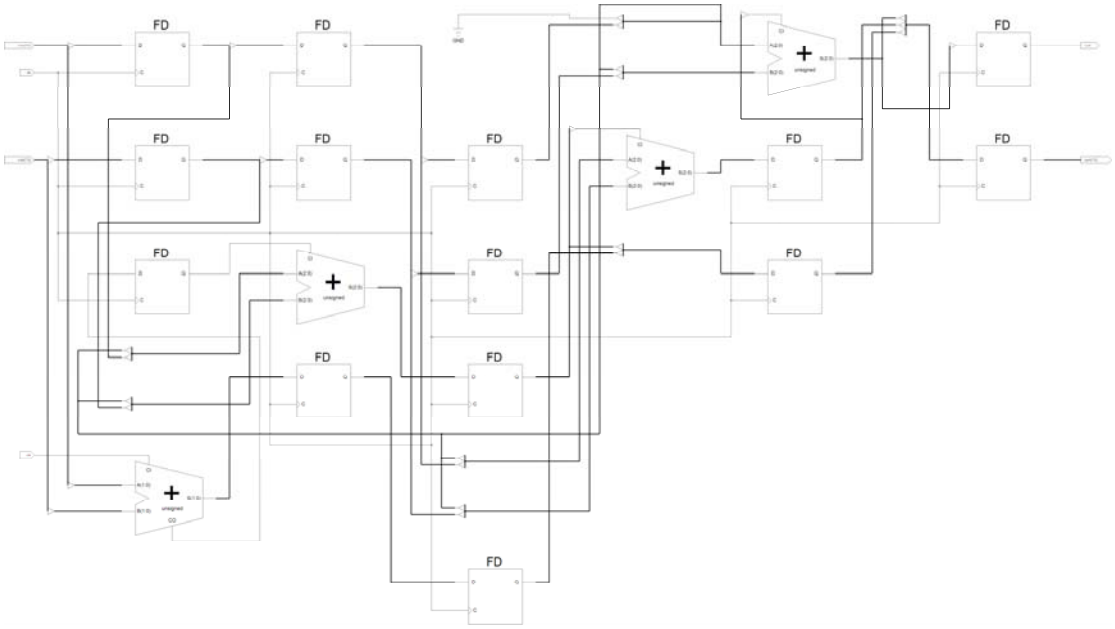


图 9-14　具有 4 级流水线结构加法器的 RTL 图

上述程序在 ISE Simulator 中的仿真结果如图 9-15 所示，可以看出，其正确完成了加法的功能，并且只有一个时钟的延迟。

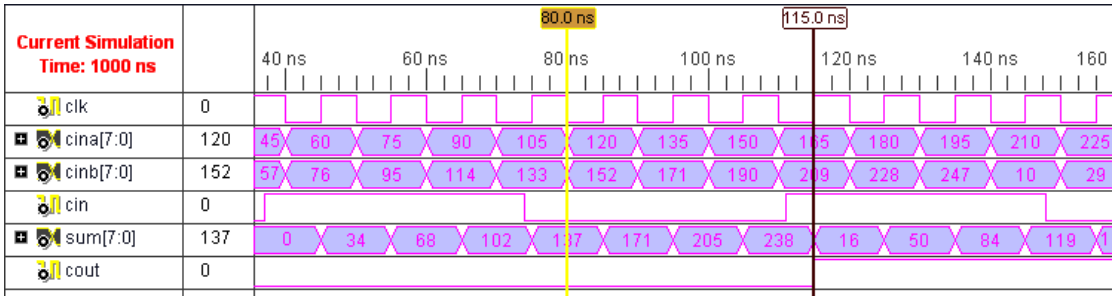


图 9-15　具有 2 级流水线结构加法器的仿真结果

3．结论

利用流水线式的设计方法，可大大提高系统的工作速度。这种方法可广泛应用于各种设计，特别是大型的、对速度要求较高的系统设计。虽然采用流水线会增大资源的使用，但是它可降低寄存器间的传播延时，保证系统维持高的系统时钟速度。在实际应用中，考虑到资源的使用和速度的要求，可以根据实际情况来选择流水线的级数以满足设计需要。

9.3.2　逻辑复用与逻辑复制原理和 Verilog HDL 实现

前文已经介绍过 FPGA 设计中面积和速度的转化关系，该理念始终贯穿 HDL 代码设计。二者典型的转换方式有逻辑复用和逻辑复制。前者通过速度换面积，后者通过面积换取速度，两者各有相应的应用范围。

1．逻辑复用——速度换面积

逻辑复用是通过提高工作频率来节省面积的优化方法，经常用于存在多个资源可共享单元的设计中，是大规模 FPGA 设计的核心思想。为了便于理解，首先给出一个例子。

例 9-6：幅度到功率转化模块中的逻辑复用。假设系统输入信号为 1、2 两路，每路速率为 5Mbps、位宽为 16 比特，分别给出普通实现和使用逻辑复用的实现代码。

功能分析：计算功率需要先将 1、2 输入信号经过平方计算模块，然后再将 2 个平方和相加输出即可，每次累加 4000 个数据长度。

普通实现方式：

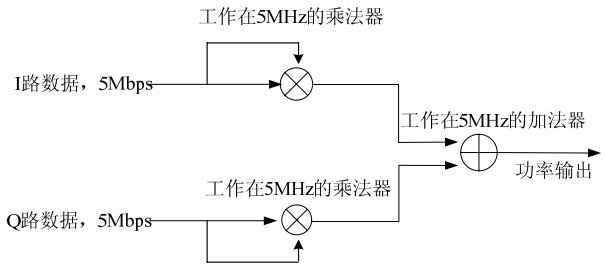


图 9-16　功率统计模块的一般实现方式

逻辑复用方式：

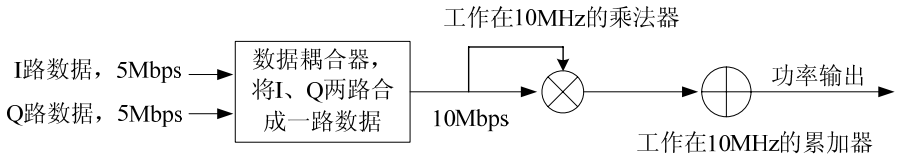


图 9-17　采用逻辑复用方式实现功率统计模块

经过比较可以发现，第 1 种实现方式需要两个 16 比特乘法器，第 2 种实现方式只需要 1 个乘法器，基本上将资源缩减到第 1 种方式的一半。目前，虽然有很多综合工具可以提供逻辑复用的选项，但不能因为此而放松对代码编程的要求。主要原因有两点：首先 EDA 工具的能力毕竟有限，很多情况下不能智能发现可以复用的逻辑；其次，不同综合工具的优化参数以及能力并不相同，所以综合结果对于设计者是不确定的。因此，逻辑复用主要还是通过代码体现。下面分别给出有、无功率复用的 Verilog HDL 代码实例。

例 9-7：利用 Verilog HDL 语言实现图 9-13 所示的普通功率统计模块。

```
module glj_v1(
    clk_5MHz, din_i, din_q, power_out
    );

     input              clk_5MHz;
     input   [15:0] din_i, din_q;
     output [45:0] power_out;

     reg      [45:0] power_out;
     reg      [45:0] power_out_tmpi = 0;
     reg      [45:0] power_out_tmpq = 0;
     wire     [31:0] pout_i, pout_q;
     reg      [13:0] cnt = 0;

     always @(posedge clk_5MHz) begin
        if (cnt == 4095) begin
                cnt                 <= 0;
                //将 I、Q 路各自的功率相加
                power_out        <= power_out_tmpi + power_out_tmpq;
                power_out_tmpi <= 0;
                power_out_tmpq <= 0;
           end
           else begin
               cnt                 <= cnt + 1;
               //完成 I、Q 路功率的累积求和计算
               power_out_tmpi <= power_out_tmpi + {{14{pout_i[31]}},pout_i[31:0]};
               power_out_tmpq  <=  power_out_tmpq  +  {{14{pout_q[31]}},pout_q[31:0]};

           end
    end

    //计算 I 路数据的平方
     mult inst_mult_i(
        .clk(clk_5MHz),
         .a(din_i),
         .b(din_i),
         .p(pout_i)
```

```
);

//计算 Q 路数据的平方
mult inst_mult_q(
   .clk(clk_5MHz),
    .a(din_q),
    .b(din_q),
    .p(pout_q)
);

endmodule
```

其中，mult 采用 ISE 中提供的乘法器 IP 核来实现，其输入为 16 比特位宽，输出为 32 比特位宽。上述程序利用了两个宽比特的乘法器，其在实现中要占用较多的 Slice 资源，占用的逻辑资源如图 9-18 所示。

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	401	4656	8%
Number of Slice Flip Flops	220	9312	2%
Number of 4 input LUTs	725	9312	7%
Number of bonded IOBs	79	232	34%
Number of GCLKs	1	24	4%

图 9-18　传统方式功率计所占资源

例 9-8：利用 Verilog HDL 语言实现图 9-17 所示的逻辑复用功率统计模块。

```
module glj_v2(
    clk_10MHz, din_i, din_q, power_out
    );

    input           clk_10MHz;
    input  [15:0] din_i, din_q;
    output [45:0] power_out;

    reg    [45:0] power_out;
    reg    [45:0] power_out_tmp = 0;
    wire   [31:0] pout;
    reg    [13:0] cnt   = 0;
    reg           flag = 0;

    reg    [15:0] a, b;

    always @(posedge clk_10MHz) begin
      if (cnt == 4095) begin
            cnt           <= 0;
            power_out     <= power_out_tmp;
            power_out_tmp <= 0;
        end
```

```
        else begin
            cnt                 <= cnt + 1;
            power_out_tmp <= power_out_tmp + {{14{pout[31]}},pout[31:0]};
        end
    end

    //利用一个乘法器分时完成两路数据的平方计算
     always @(posedge clk_10MHz) begin
        flag    <= ~flag;
         case(flag)
            1'b0: begin
                    a <= din_i;
                    b <= din_i;
                 end
            1'b1: begin
                    a <= din_q;
                    b <= din_q;
                 end
        endcase
     end

     mult inst_mult(
       .clk(clk_10MHz),
        .a(a),
        .b(b),
        .p(pout)
     );

endmodule
```

上述程序完全实现了例 9-7 的功能，并且通过图 9-19 所示的资源统计列表可以看出：通过逻辑复用，可以将所需的资源数降低到原来的一半左右。之所以没达到准确的一半，是因为在通过高速时钟片选数据时，需要付出几个额外的寄存器资源。

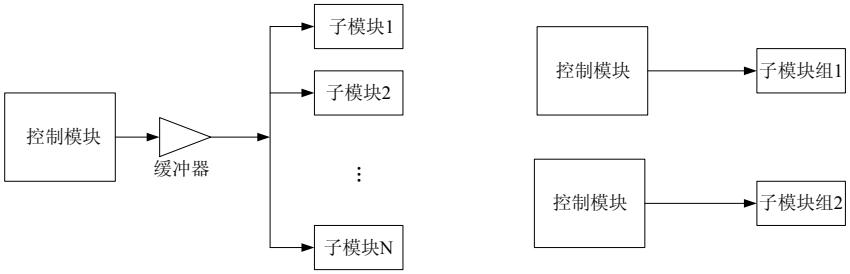
Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	223	4656	4%
Number of Slice Flip Flops	172	9312	1%
Number of 4 input LUTs	365	9312	3%
Number of bonded IOBs	79	232	34%
Number of GCLKs	1	24	4%

图 9-19　逻辑复用功率计所占资源

2．逻辑复制——面积换速度

逻辑复制是通过增加面积而改善设计时序的优化方法，经常用于调整信号的扇出。如果信号具有高的扇出（如：时钟和复位信号等），即要驱动很多后续电路，则要添加缓存器来增强驱动能力，但这会增大信号的时延。通过逻辑复制，使用多个相同的信号来分担驱动

任务。这样，每路信号的扇出就会变低，就不需要额外的缓冲器来增强驱动，即可减少信号的路径延迟。例如用于产生控制信号的监控模块一般都有高的扇出，这时就往往需要考虑逻辑复制这一功能。图 9-20(a)给出未采用逻辑复制的设计思路，其占用资源较少，但延迟大，容易出错；而采用逻辑复制的设计如图 9-20(b)所示，延迟小，但占用的资源多。



(a) 未采用逻辑复制的设计模式　　(b) 采用逻辑复制的设计模式

图 9-20　逻辑复制实例示例

逻辑复用和逻辑复制是资源与速度的对立统一，目的都是为了提高设计性能、达到设计目标，但一个是为了节省面积来提高速度，另一个却是为了提高速度来占用额外的面积，两者之间存在转换和平衡的关系。在实际工程中，经常可以看到这两种方法的应用。在 Xilinx ISE 的综合工具 XST 中，用户可以设定最大扇出数，当某信号的扇出超过最大扇出值时，该信号会自动被综合工具复制，以降低扇出。

9.3.3　关键路径提取原理和 Verilog HDL 实现

在 Verilog 设计中，经常会遇到由于信号路径过长或信号来的比较晚，从而造成建立时间不够的情况。这种引起电路建立时间不够的信号路径就称为关键路径。在复杂电路设计中必须有效地处理关键信号，尽量减少其时延，提高电路的工作频率。

1．简单组合电路关键路径的提取方法

简单组合电路的关键路径提取方法就是拆分逻辑，将复杂逻辑变成多个简单组合电路的进一步组合，缩减关键信号的逻辑级数，如例 9-9 所示。

例 9-9：简单关键电路的提取实例。对于语句：

```
assign y = a & b & c | d & e & b;
```

从中可以看出，信号 b 为关键信号。现将其简单路径计算，再经过关键路径逻辑。

```
assign temp = a & b & c & d;
assign y = b & temp;
```

assign temp=a &c | d &e

通过关键路径提取，将信号 b 的路径由 2 级变成 1 级。拆分逻辑的方法就是布尔逻辑扩展，也被称为香农扩展，其原理如下式所示：

$$F(x,y,z) = xF(1,y,z) + \overline{x}F(0,y,z)$$

可以看出，拆分逻辑可通过复制逻辑，缩短那些组合路径长的关键信号的路径延迟，从而提高工作频率。

2．复杂 always 块中关键路径的提取方法

对于 always 模块中时间要求非常紧的信号，需要通过分布提取方法，让关键路径先行，保证改写后的描述与原 always 块逻辑等效。例 9-10 给出了提取并改善 always 模块中关键信号的实例。

例 9-10：always 块中关键路径的提取和优化实例。

```
always@(w or x or y or z or in1 or in2) begin
```

```
    if(!w) begin
        if(x&&(! (y&&z)))
            out = in1;
        else
            out = in2;
    else if(y&&z)
        out =in1;
    end
    else begin
        out = out;
    end
end
```

其中，若 z=0，则原代码等效于 if (!w) out = in1; else out = out;。若 z=1，则源代码等效于 if (!w && x && !y) out = in1; else if(!w && x && y) out = in2; else out = out;。对于信号 y 也有类似的分析结果，因此 y、z 都是关键信号。因此通过优先首先计算关键路径，改写为：

```
always@(w or x or y or z or in1 or in2) begin
    temp = y && z;
    if(!temp) begin
        if(x&&!w)
            out = in1;
        else
            out = in2;
    else if(temp)
        out =in1;
    end
    else begin
        out = out;
    end
end
```

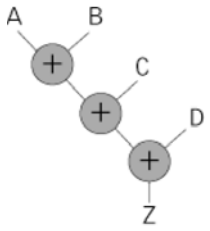
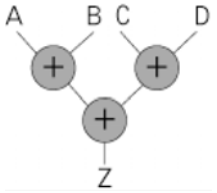
9.3.4 逻辑合并与拆分原理和 Verilog HDL 实现

逻辑合并和拆分操作主要用于修改逻辑结构。数字电路中的逻辑结构主要分为链状结构（Chain Architecture）和树状结构（Tree Architecture）。一般来讲，链状结构具有较大的时延，后者具有较小的时延。所谓的链状结构主要指程序是串行执行的，树状结构是串并结合的模式，具体如例 9-11 所示。

例 9-11：表 9-1 给出具有链状结构和树状结构的 4 输入加法器的实现实例。

表 9-1　　4 输入加法器的实现

	链状结构	树状结构
程 序 描述	Z <= A + B + C + D;	Z <= (A + B) + (C + D);

RTL 综合图		
性能	具有 3 个延迟	在占用同等数量的资源下，具有 2 个延迟

从上例可以明显看出树状结构的优势，它能够在同等资源的情况下，缩减运算时延，从而提高电路吞吐量以节省面积。下面给出一个逻辑合并的 Verilog HDL 设计实例。

例 9-12：利用 Verilog HDL 语言实现一个 4 输入的树形加法器。

```
module adder_tree(
    dina, dinb, dinc, dind, dout
    );

    input   [7:0] dina, dinb, dinc, dind;
    output [9:0] dout;

    wire    [8:0] sum1, sum2;

    assign sum1   = dina + dinb;
    assign sum2   = dinc + dind;
    assign dout   = sum1 + sum2;

endmodule
```

上述程序在 ISE 中综合后的 RTL 级结构如图 9-21 所示，可以看出利用 3 个加法器实现了 4 输入加法的并行结构。

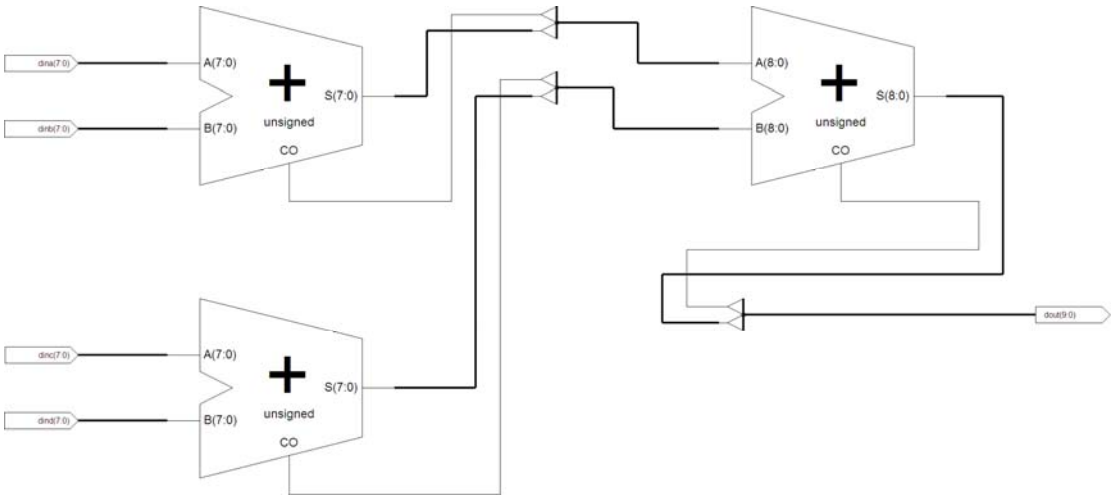


图 9-21　4 输入加法器树形结构示意图

9.3.5　多时钟域接口设计技巧

1．多时钟域介绍

在设计中，经常采用多时钟设计，必然存在亚稳态（当触发器的建立时间和保持时间要求没有得到满足时，触发器就会进入一个界于逻辑 1 和逻辑 0 之间的第三种状态，即亚稳态）。

理想的触发器是在时钟边沿到达的那个时刻采样数据。但是在实际电路中，时钟的跳变具有一定的斜率，电路采样、保存、传递数据也需要一定的时间。如果在数据还未稳定时进行采样，就可能会导致亚稳态的发生，如果处理不当，将会给设计的可靠性带来极大的隐患。

一个良好的 FPGA 设计是尽量让所有模块都处在高速时钟的运行下，减少硬件资源的使用。多速率时采用使能信号（Clock Enable）配合系统时钟实现，但是这种设计的缺点是功耗高、并且很难做到高速的使能管理；所以在某些特定情况下，使用多时钟处理便成为一种很好的设计方案。多时钟设计的一个典型应用是 FPGA 与不同速率的外设接口连接。这些外设接口包括很多，比如：（1）I/O 缓冲（buffer）与微处理器接口，微处理器使用处理器内部时钟采数据，与 I/O 缓冲数据速率不同，这就需要处理两个不同时钟域的数据通信。（2）数字信号送往 DA 转换器接口，该 DA 的时钟与数据速率不同，也同样需要处理不同时钟域的数据通信。可见，当 FPGA 与外设的处理时钟不同时，都需要使用多时钟设计。

为了保证各时钟域内部仍然是同步设计，并且使得时钟边沿位于数据的中间，则需要在两个时钟域之间添加同步接口设计，将相对本时钟域的异步数据转化成同步数据。常用的方法包括基于 D 触发器、异步 FIFO（双口 RAM）来建立接口，尽量避免让信号直接跨越不同时钟域。下面分别对这两种接口进行介绍。

2．基于 D 触发器的同步接口

同步器的功能是采样异步输入信号，并使产生的输出信号满足同步系统的建立时间和保持时间的要求。简单的同步器一般采用单级 D 触发器来构成，即在不同的时钟域端添加一个 D 触发器，如图 9-22 所示。D 触发器的每一个时钟触发沿采样异步输入信号，并产生一个同步后的输出信号。要构建更好的同步器可以采用更快速的触发器，减小采样保持时间，使器件能够更快地采样到信号。

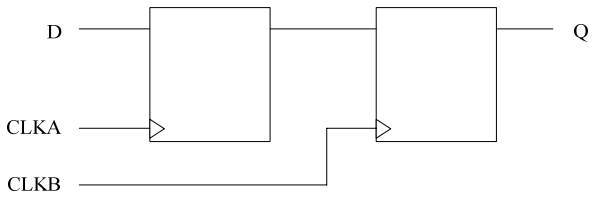


图 9-22　单级 D 触发器构成的同步接口

D 触发器工作过程中存在数据建立与保持时间的约束，如果这种约束得不到满足，触发器就会进入某个不确定状态——亚稳态。亚稳态的存在可能导致连锁反应，以致引起整个系统功能混乱。在单时钟域电路设计中由于不存在时钟之间的延迟和错位，所以建立条件和保持条件的时间约束容易满足。而在多时钟域里由于各个模块的非同步性，则必须考虑亚稳态的发生，如图 9-23 所示。

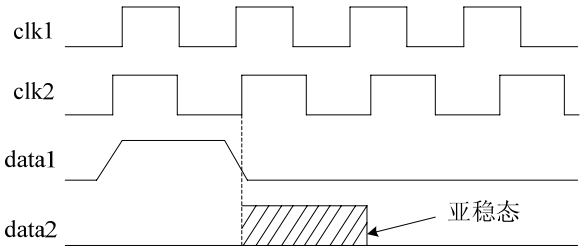


图 9-23　多时钟域亚稳态的产生

图 9-24 为最常用的一种双寄存器同步法，即在一个信号进入另一个时钟域前，将该信号用两个寄存器连续寄存两次，最后得到的采样结果就可以消除亚稳态。使用两级是因为时钟周期由于技术进步正在日益减少，使得单个触发器的亚稳定性不可能在单个周期内解决，

两级同步能够更可靠地避免亚稳态的出现，而三级以上同步器的效果并不能提高多少。

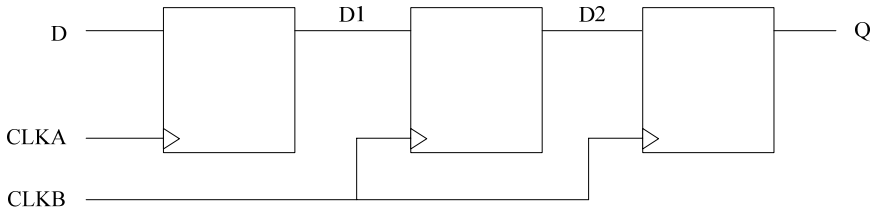


图 9-24　两级 D 触发器构成的同步接口

3．基于异步 FIFO 的同步接口

FIFO 是数据传输系统中极其重要的一环，特别是在两个处在不同时钟域的系统接口部分，FIFO 的合理设计，将不但能使接口处数据传输的输入输出速率进行有效的匹配，不使数据发生复写、丢失和读入无效数据的情况，而且还将有效地提高系统中数据的传输效率。FIFO 存储器主要分为基于移位寄存器型和基于 RAM 型。而后者又有单口 RAM 和双口 RAM 之分，目前来说用的较为广泛的是基于双端口 RAM 的 FIFO。因为时钟的同步和异步之分，FIFO 又可分为同步 FIFO 和异步 FIFO。多时钟域情况下要求 FIFO 的读写时钟为异步的，因此本节主要介绍基于双口 RAM 的异步 FIFO。

异步 FIFO 由存储器块和控制逻辑构成，控制逻辑包括读写指针，读写指针分别位于不同的时钟域中，其内部结构如图 9-25 所示。当 FIFO 中有数据而非空时，read 信号用来控制读数据，所读数据来自读指针所指的存储单元，并且读指针加一。当读指针追上写指针时，FIFO 为空并产生一个 empty 信号，empty 同步于读时钟。当 FIFO 中有空间而非满时，write 信号用来控制写数据，所写数据写向写指针所指的存储单元，并且写指针加一。当写指针追上读指针时，FIFO 为满并产生一个 full 信号，full 信号同步于写时钟。

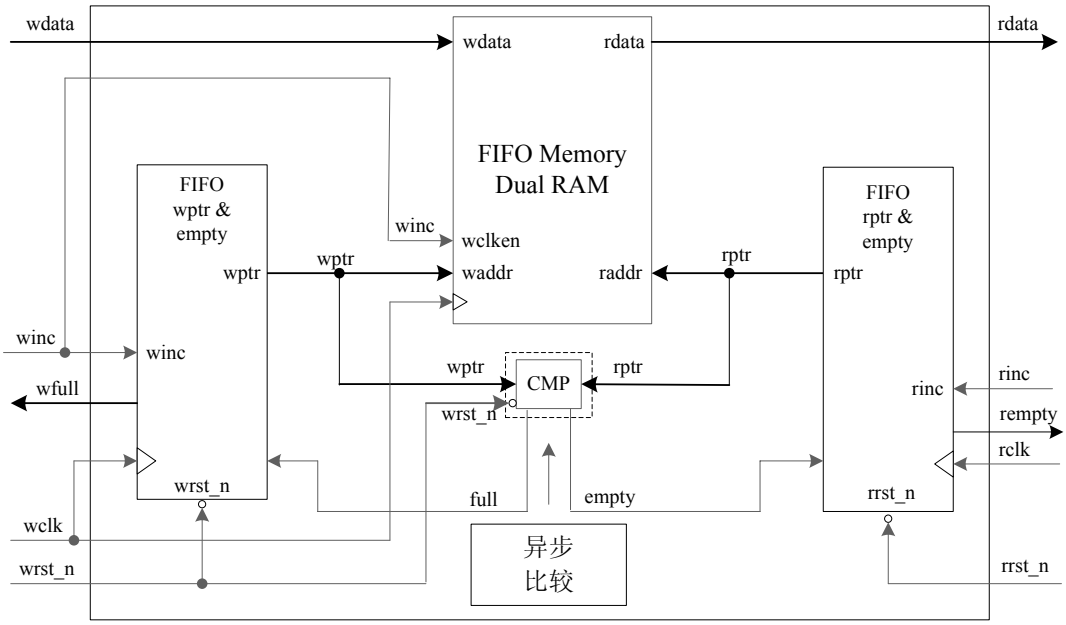


图 9-25　基于双口 RAM 的异步 FIFO 结构图

异步 FIFO 在多时钟域的接口方法如图 9-25 所示，其中发时钟域控制 FIFO 的写数据端，收时钟域控制 FIFO 的读数据端，一条单向的通信链路就需要一个异步 FIFO。在每次读、写之前都需要判断 FIFO 的 empty 和 full 信号。在基于 Xilinx 的 ISE 平台设计中，FIFO 可通过 IP 核的方式直接调用，下面给出一个异步 FIFO 的应用实例。

例 9-13：在 ISE 中调用异步 FIFO 的 IP 核，并完成多时钟域的功能仿真。

（1）在 ISE 新建工程，然后在工程管理区，单击右键，选择“New Source”命令，在文件类型中选择“IP（CORE Generator & Architecture Wizard）”，并在右端的“File Name”文本框输入 fifo_demo，然后单击“Next”按键进入下一页，如图 9-26 所示。

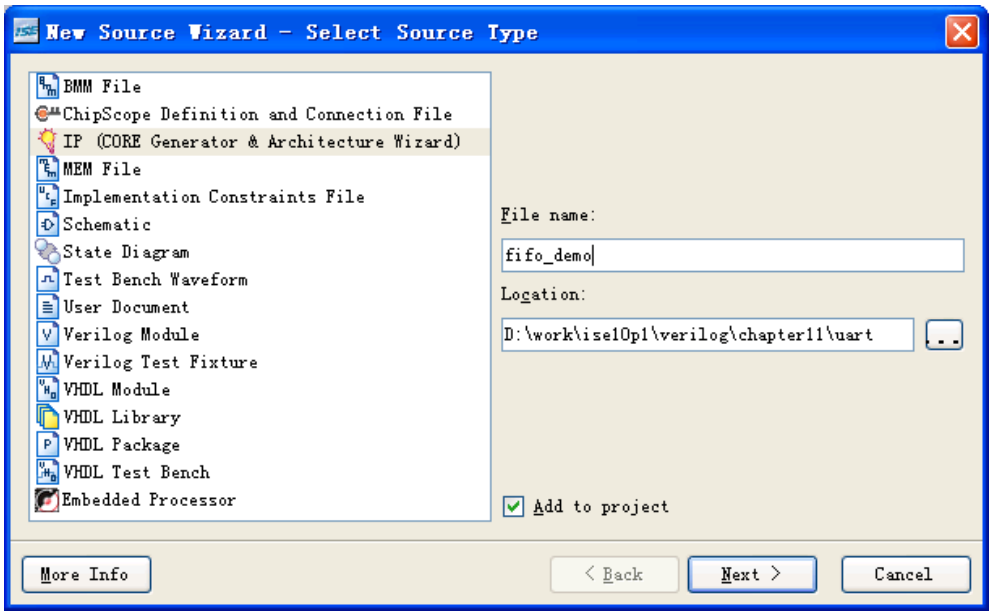


图 9-26　新建 FIFO 的 IP 核向导（1）

（2）然后在弹出的“Select IP”页面，选择“Memories & Storage Elements”类别下“FIFOs”分类中的“Fifo Generator v4.3”，如图 9-27 所示。然后单击“Next”按键进入下一页。

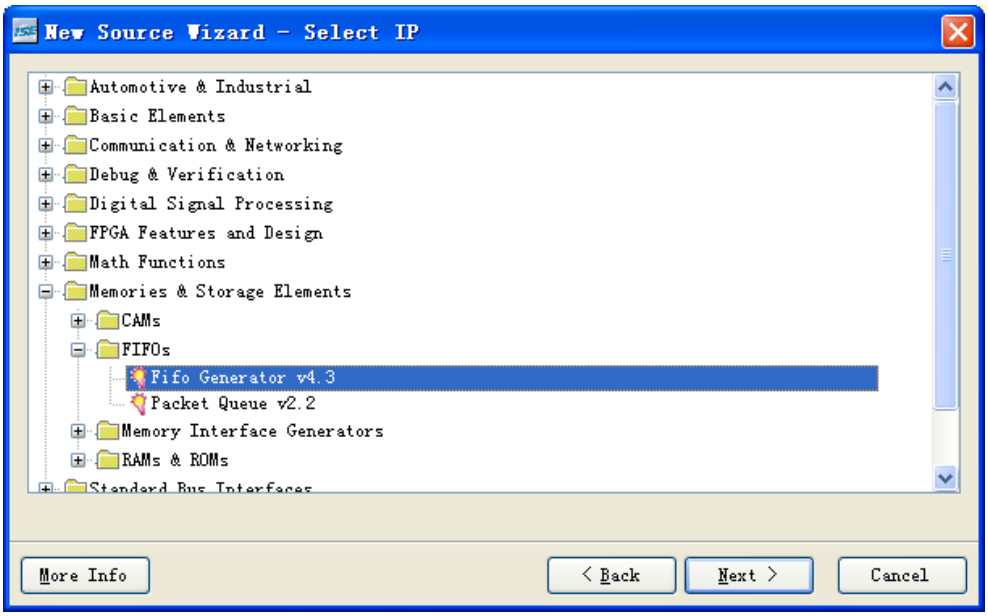


图 9-27　新建 FIFO 的 IP 核向导（2）

（3）在弹出的确认页面中，点击“Finish”按键，完成 IP 核文件生成过程，进入 IP 核配置页面，如图 9-28 所示。其中，“Read/Write Clock Domains”说明了 FIFO 的读写时钟域，“Common Clock(CLK)”表明读、写采用同一个时钟，适合同步系统中应用；“Independent Clocks(RD_Clk, WR_Clk)”选项表明读、写时钟分离，可用于不同时钟域的数据交换，本例就需要选择这种读、写时钟分离的 FIFO。“Memory Type”用于说明 FIFO 的实现细节，“Block RAM”表明 FIFO 基于 FPGA 内部的硬核块 RAM 来搭建，主要用于大存储深度和高速情况

下；“Distributed RAM”类型通过 LUT 单元来搭建，用于小存储深度和普通速度的应用场合。

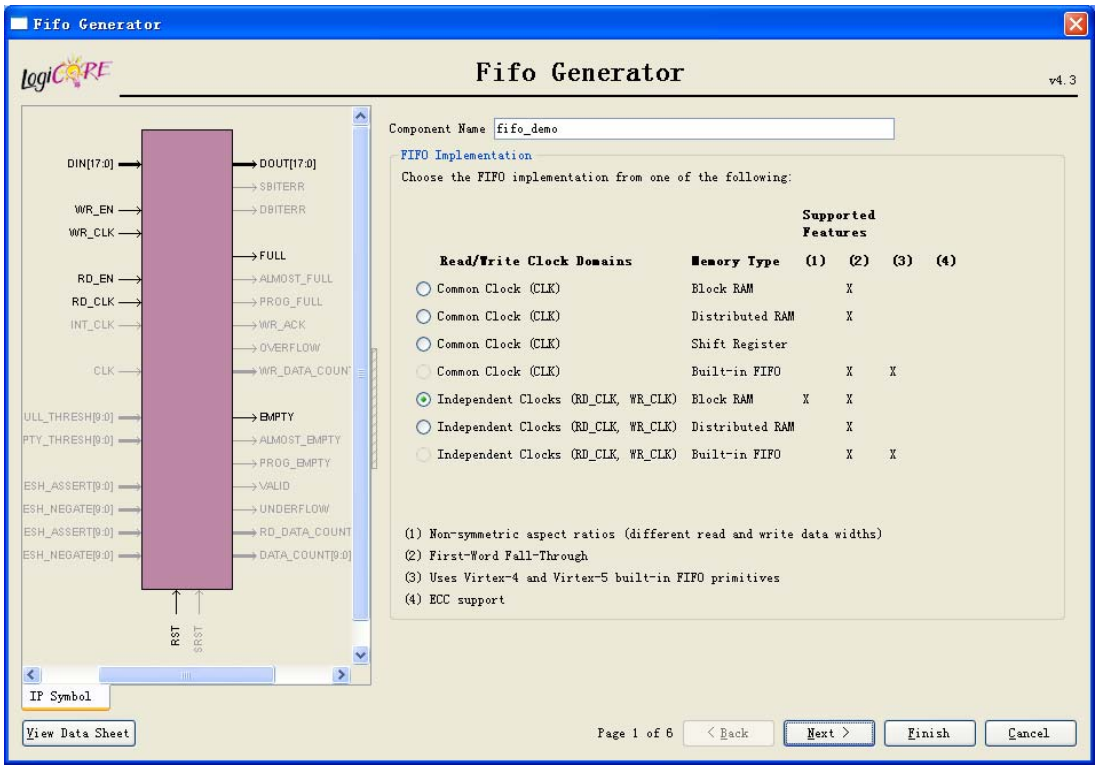


图 9-28　新建 FIFO 的 IP 核向导（3）

此外，IP 核还有 4 类其它特征，包括有：（1）非对称的读、写位宽；（2）首字直接通过；（3）利用 Virtex 4 或者 Virtex 5 中的内建 FIFO；（4）支持 ECC 功能。在图 9-28 中，“X”表示不支持。

（4）在弹出的 FIFO IP 核配置窗口中，“Read Mode”用于配置 FIFO 类型，其中“Standard FIFO”选项表明其具备 FIFO 的一般特征，而“First-Word Fall-Thtough”则意味着第一个数据直通，没有延迟。“Build-in FIFO Options”选项用于选择读、写时钟的相对数值大小。“Data Port Parameter”栏用于选择读写端口的位宽和深度，其中读、写的位宽是独立的，但是读端口的深度由写端口决定。“Implementation Options”栏用于控制 FIFO 的实现细节，包括校验编码以及是否使用其内在的寄存器。本例选择标准 FIFO，数据深度选择 1024、读写位宽都为 18 比特，确认后点击“Next”按键，进入下一配置页面。

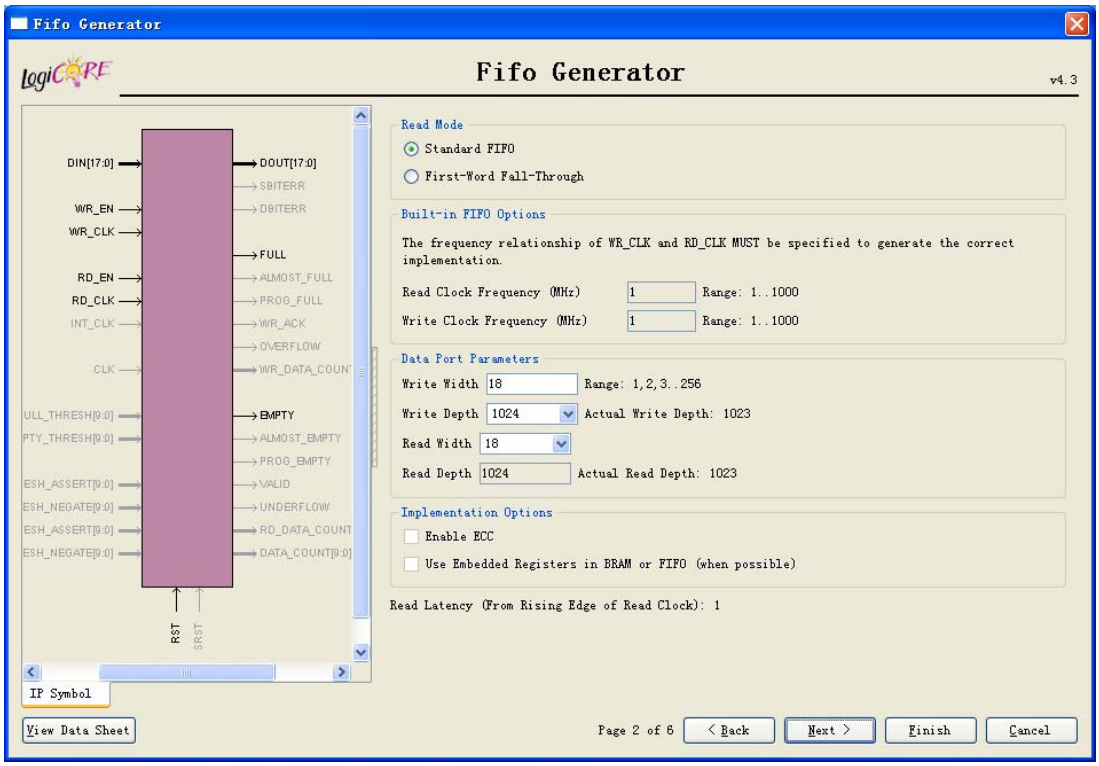


图 9-29　新建 FIFO 的 IP 核向导（4）

（5）接下来的页面用于配置 FIFO 的标志信号和握手信号。“Optional Flags”区域用于给出 FIFO 的“Almost Full”和“Almost Empty”指示，高有效，所谓的“Almost”就是“接近”的意思，其和“真正”的满、空差一个数据。“Handshaking Options”区域用于添加握手信号，主要包括读、写操作的 ACK 信号（表明数据读写已完成）和读、写操作的溢出信号（FIFO 已满还在继续写入数或者 FIFO 已空还在继续读取数），并且可以配置这些握手信号信号的有效电平，设计人员可根据需求灵活选择。本例添加了读、写溢出信号，并设置为高有效，如图 9-30 所示。配置完成后，单击“Next”按键，进入下一配置页面。

（6）该页面用于设置 FIFO 初始化信号以及可编程标志信号。其中“Initialization”区域用于配置是否需要复位管脚、复位形式（同步复位还是异步复位）以及复位后数据断口输出端（Dout）信号的数值（在“Use Dout Reset Value”文本框中直接以 16 进制形式输入）。“Programmable Flags”用于设置可编程的空、满深度，并最多可以设置两个用户定义的空、满深度（该深度并不对应物理上的空、满），共有 No Programmable Full/Empty Threshold、Single Programmable Full/Empty Threshold Constant、Multiple Programmable Full/Empty Threshold Constants、Single Programmable Full/Empty Threshold input 以及 Multiple Programmable Full/Empty Threshold inputs 等 5 类选择，其中，第 1 类选择采用默认值，满、空深度数值分别为 1021 和 2；第 2 类选择，可以在 Full/Empty Threshold Assert Value 本文框中输入；第 3 类选择，可以在 Full/Empty Threshold Assert Value、Full/Empty Threshold Negate Value 这两个文本框中输入；后两类选择通过外部输入端口输入。本例选择如图 9-31 所示，配置完成后，单击“Next”按键进入下一页。

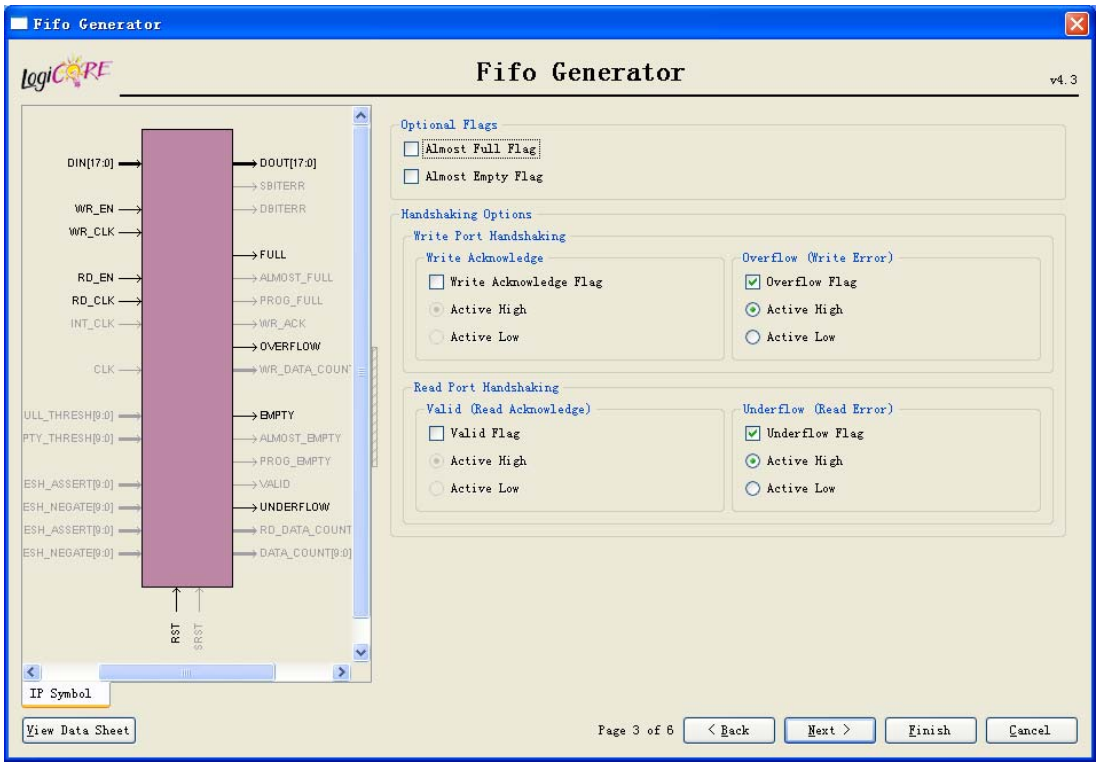


图 9-30　新建 FIFO 的 IP 核向导（5）

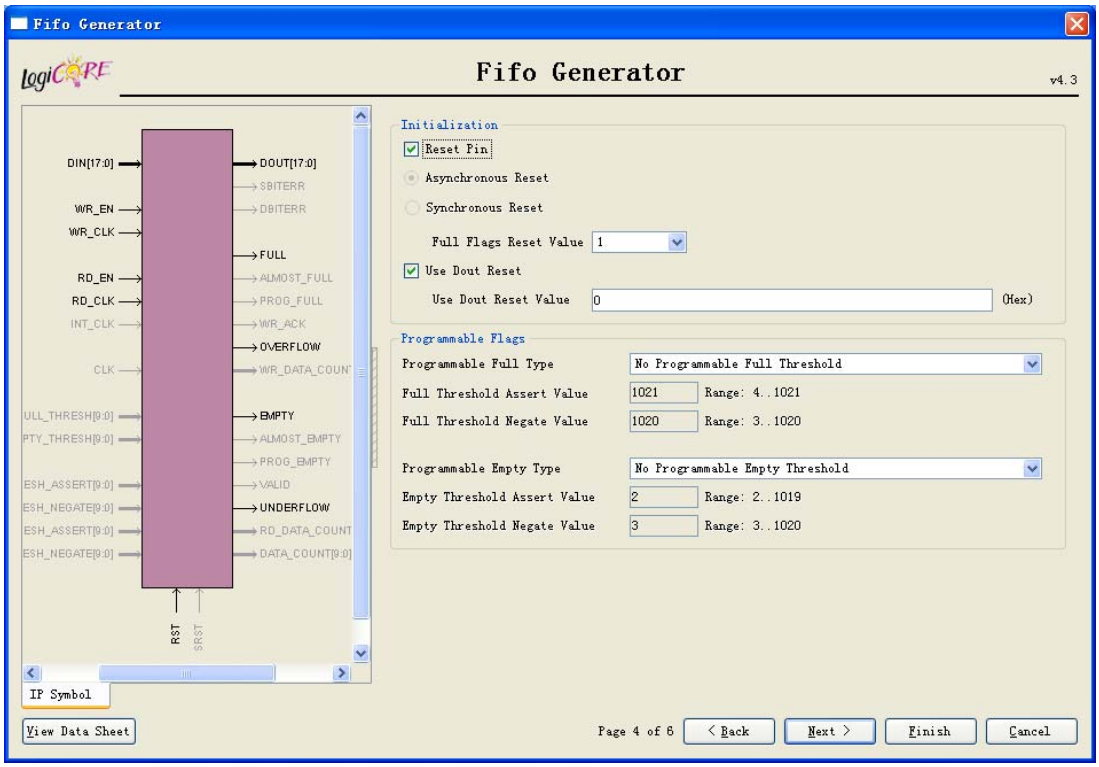


图 9-31　新建 FIFO 的 IP 核向导（6）

（7）该配置界面用于给出数据计数器选项，选中 Write Data Count*和 Read Data Count*，可分别在写、读时钟上升沿送出 FIFO IP 核中的数据个数，并且可以在 Write/Read Data Count Width 文本框中输入个数计数器的位宽，有效值为 1~10。本例选择如图 9-32 所示，确认后

单击“Next”按键进入下一页。

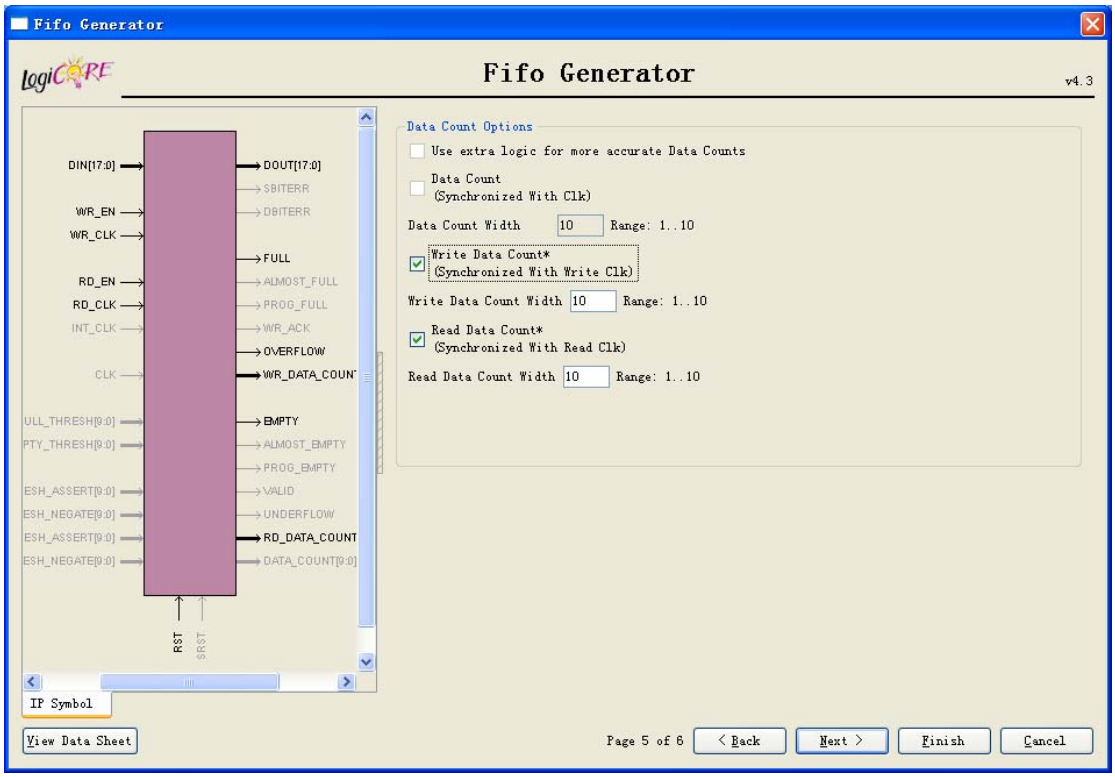


图 9-32　新建 FIFO 的 IP 核向导（7）

（8）本页面是确认页面，是 IP 核配置的最后一页，分别列出了选择的 FIFO 类型、FIFO 的实现方式（块 RAM 还是分布式 RAM）、读写数据位宽以及深度、块 RAM 使用数量以及各类握手信号，用户确认无误后，单击“Finish”按键，完成 IP 核的生成；如果存在参数和期望输入不一致，单击“Back”按键返回前一页面进行修改。本例的参数如图 9-33 所示，单击“Finish”按键，完成 IP 核的配置，ISE 会自动生成相应的各类文件，并输入下面的指示信息。

Customizing IP...

10.1.02 - Xilinx CORE Generator IP GUI Launcher K.37 (nt)

Copyright (c) 1995-2008 Xilinx, Inc.　All rights reserved.

WARNING:coreutil - sim:254 - Unable to launch pdf viewerWARNING:coreutil - sim:254 - Unable to launch pdf viewerFinished Customizing.

Generating IP...

Generating Implementation files.

Generating ISE symbol file...

WARNING:coreutil -　Default charset GBK not supported, using ISO-8859-1 instea

Generating NGC file.

Finished Generating.

Successfully generated fifo_demo.

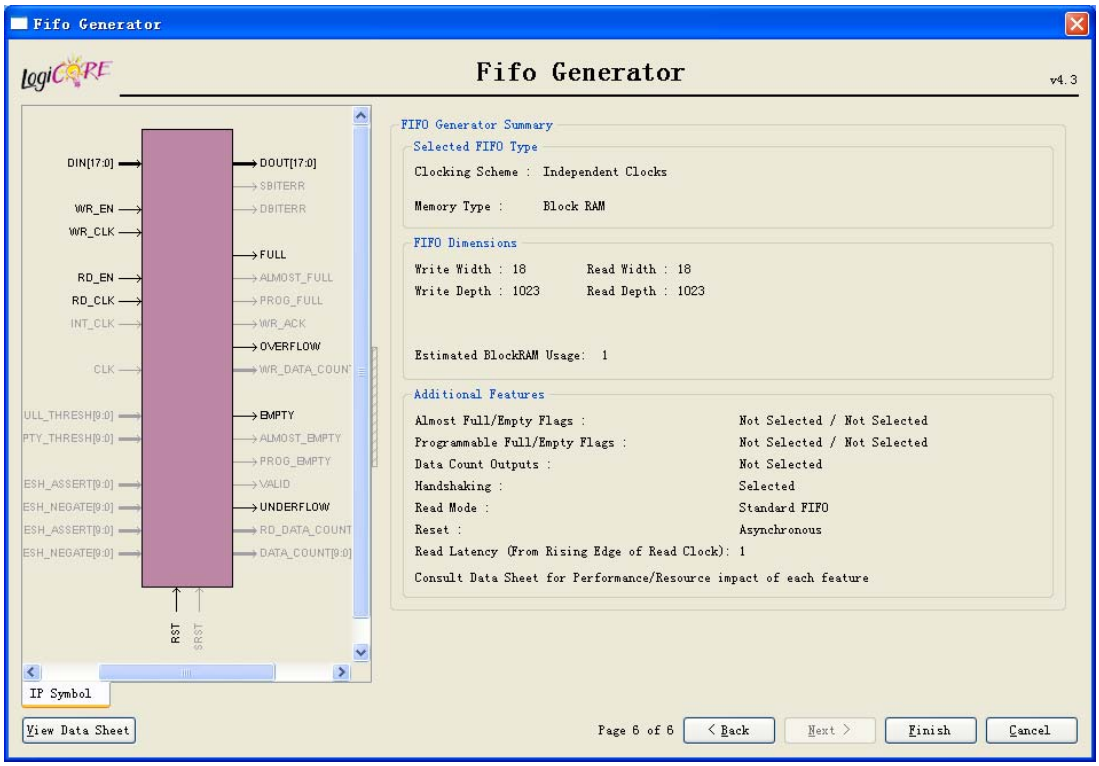


图 9-33　新建 FIFO 的 IP 核向导（8）

（9）在工程管理区，单击鼠标左键选中生成的 FIFO IP 核，然后在过程管理区，双击“View HDL Functional Model”，即可在代码编辑区查阅 IP 核的端口说明，如图 9-34 所示。

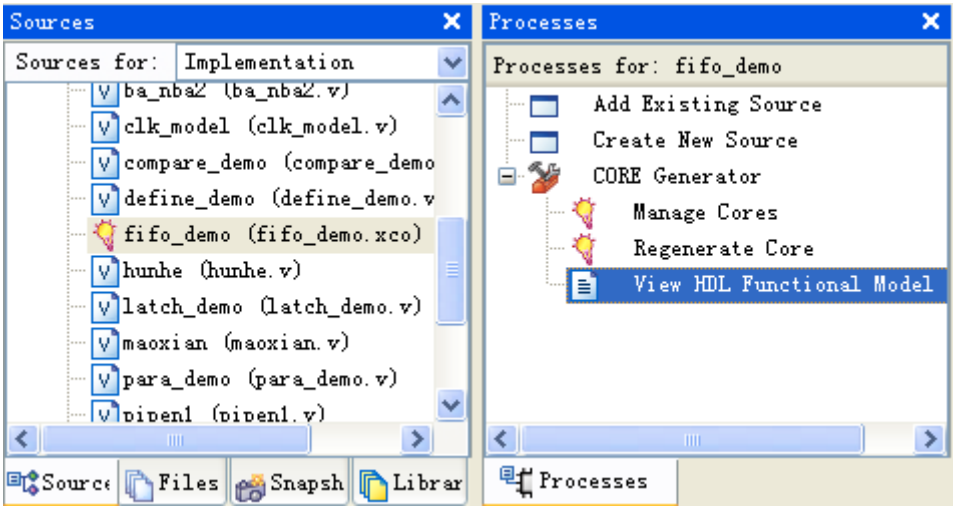


图 9-34　查看 FIFO IP 核代码的操作示意图

FIFO IP 核在代码编辑窗口显示的代码头部端如图 9-35 所示，在 Verilog HDL 代码中例化该 IP 核时，直接通过 fifo_demo 完成例化操作即可。

```
40  module fifo_demo(
41     din,
42     rd_clk,
43     rd_en,
44     rst,
45     wr_clk,
46     wr_en,
47     dout,
48     empty,
49     full,
50     overflow,
51     rd_data_count,
52     wr_data_count);
53
54
55  input [17 : 0] din;
56  input rd_clk;
57  input rd_en;
58  input rst;
59  input wr_clk;
60  input wr_en;
61  output [17 : 0] dout;
62  output empty;
63  output full;
64  output overflow;
65  output [9 : 0] rd_data_count;
66  output [9 : 0] wr_data_count;
67
68  // synthesis translate_off
```

图 9-35　FIFO IP 核端口代码示意图

（10）在工程中新建 fifo_test 的 Verilog HDL 源代码文件，在其中例化生成的 IP 核，其内容如下所列：

```
module fifo_test(
    din, rd_clk, rd_en, rst, wr_clk, wr_en, dout, empty, full, overflow,
rd_data_count, wr_data_count);

    input [17 : 0] din;
    input rd_clk;
    input rd_en;
    input rst;
    input wr_clk;
    input wr_en;
    output [17 : 0] dout;
    output empty;
    output full;
    output overflow;
    output [9 : 0] rd_data_count;
    output [9 : 0] wr_data_count;

    //例化 fifi_demo IP 核
    fifo_demo inst_fifo_demo(
        .din(din),
        .rd_clk(rd_clk),
        .rd_en(rd_en),
```

```
.rst(rst),
.wr_clk(wr_clk),
.wr_en(wr_en),
.dout(dout),
.empty(empty),
.full(full),
.overflow(overflow),
.rd_data_count(rd_data_count),
.wr_data_count(wr_data_count)
);

endmodule
```

在 ISE 中综合上述程序，可以得到其占用的逻辑资源表，如图 9-36 所示，可以看出，该 FIFO 模块使用了一个块 RAM 外，还使用了 100 个 Slice，这部分逻辑单元主要用于将基本的块 RAM 转化成 FIFO，完成图 9-25 中，除块 RAM 之外的功能。

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	100	4656	2%
Number of Slice Flip Flops	166	9312	1%
Number of 4 input LUTs	112	9312	1%
Number of bonded IOBs	64	232	27%
Number of BRAMs	1	20	5%
Number of GCLKs	2	24	8%

图 9-36　FIFO 模块的资源统计结果

其 RTL 级结构如图 9-37 所示，可以看出，其中“fifo_demo”模块就是生成的 FIFO IP 核，fifo_test 模块对其例化，只是简单的端口连接，并没有添加任何逻辑。

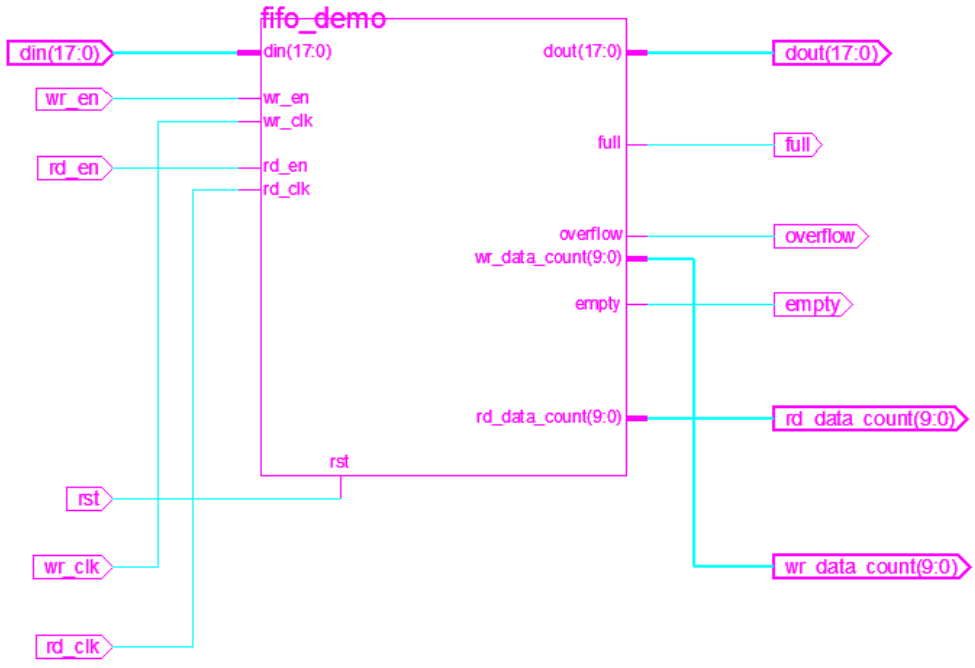


图 9-37　fifo_test 模块的 RTL 结构图示意图

（11）为了验证该 FIFO 核，编写的测试代码如下所列。

```
module tb_fifo_test;

    // Inputs
    reg [17:0] din;
    reg rd_clk;
    reg rd_en;
    reg rst;
    reg wr_clk;
    reg wr_en;

    // Outputs
    wire [17:0] dout;
    wire empty;
    wire full;
    wire overflow;
    wire [9:0] rd_data_count;
    wire [9:0] wr_data_count;

    // Instantiate the Unit Under Test (UUT)
    fifo_test uut (
        .din(din),
        .rd_clk(rd_clk),
        .rd_en(rd_en),
        .rst(rst),
        .wr_clk(wr_clk),
        .wr_en(wr_en),
        .dout(dout),
        .empty(empty),
        .full(full),
        .overflow(overflow),
        .rd_data_count(rd_data_count),
        .wr_data_count(wr_data_count)
    );

    initial begin
        // Initialize Inputs
        din = 0;
        rd_clk = 0;
        rd_en = 0;
        rst = 0;
        wr_clk = 0;
        wr_en = 1;
```

```
        // Wait 100 ns for global reset to finish
        #100;
      rst = 1;
end

always #5 begin
    rd_clk = ~rd_clk;
     wr_clk = ~wr_clk;
end

always @(posedge wr_clk) begin
    if(wr_en)
        din = din + 1;
end

always #12000 begin
    rd_en   = ~rd_en;
     wr_en   = ~wr_en;
end

endmodule
```

从仿真代码中可以看出，在写入和读取数据交替进行，单个操作的数据个数都大于 FIFO 的深度，因此肯定会交替出现 full、overflow 信号和 empty 信号交替变高的波形。在 ISE Simulator 中运行上述程序，并将仿真时间设为 100000ns，可得到图 9-38 所示的仿真结果，整体上符合仿真代码的测试意图。

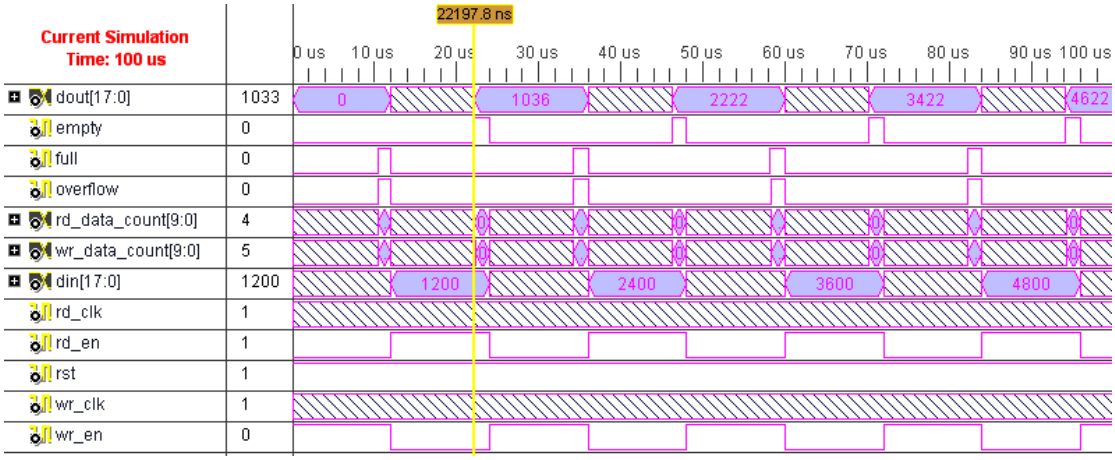


图 9-38 FIFO 模块的仿真结果

下面方法图 9-38 中数据 fifo 第一次被写满时刻的时序图，如 9-39 所示，可以看出当 fifo 中的数据个数超过 1021 时，就会将 full 信号拉高，此时再写入数据，就会产生溢出，overflow 信号变高。这与图 9-31 中的 full 状态门限值得的设置是一致的。

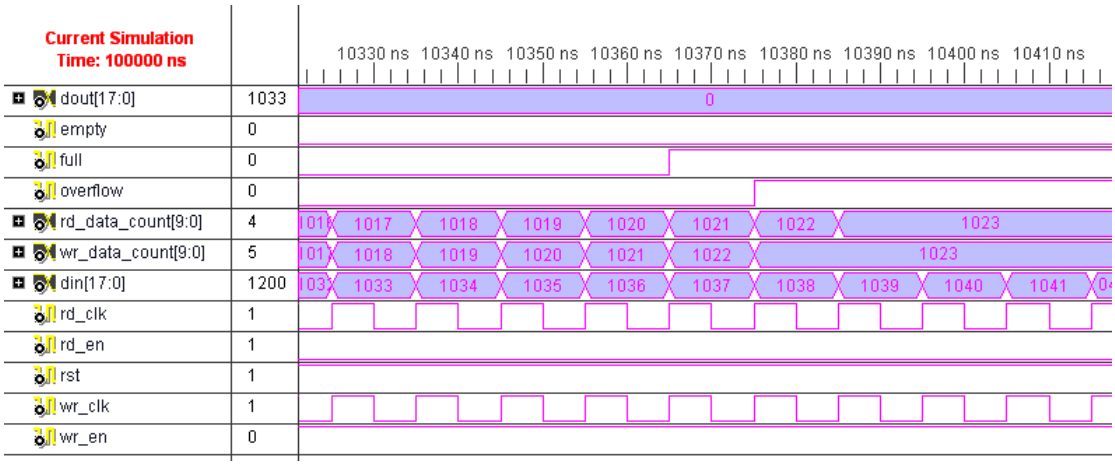


图 9-39 FIFO 满时刻仿真结果

fifo 第一次开始读的时序如图 9-40 所示，同时由于写信号无效，此时 overflow 信号就由高变低，每读出一个，fifo 中的数据个数就少一个，当其等于 1021 时，full 信号由高变低。

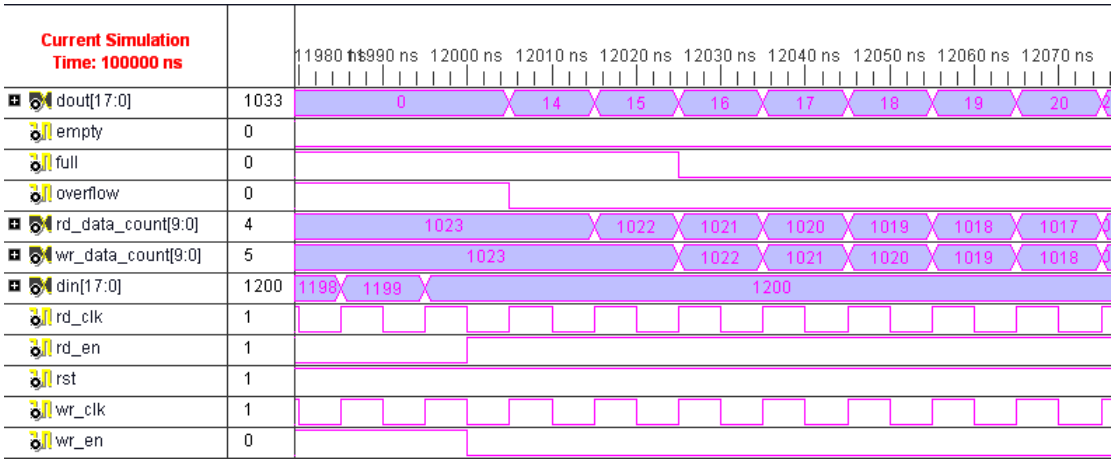


图 9-40 FIFO 第一次读操作的仿真结果

图 9-41 给出 fifo 第一次读空时的时序图，可以看出当 fifo 中数据少于 2 个时，empty 信号变高，这与图 9-31 中所配置的 fifo 下限是一致的。

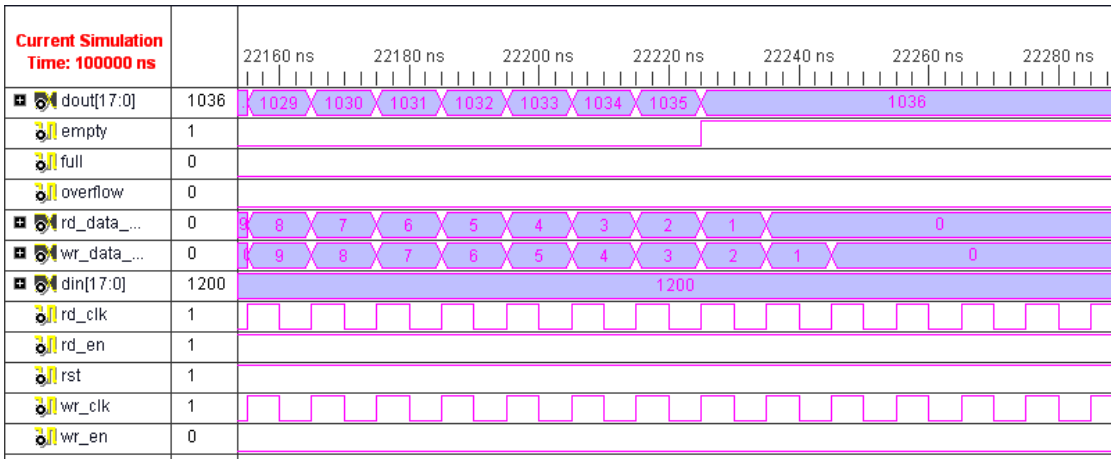


图 9-41 FIFO 空时刻的仿真结果

9.4 本章小结

本章主要讨论了 Verilog HDL 设计中的基本内在规律。首先介绍了面积和速度互换原则

和模块划分原则等基本指导思想。其次，说明了代码风格，指出代码风格包括代码书写风格和代码设计风格，并给出了针对 Xilinx 器件的专用代码设计风格。最后介绍了常用的代码思想及其 Verilog HDL 代码示例，包括流水线技术、逻辑复用和逻辑复制技术、关键路径提取技术、逻辑合并和逻辑拆分技术以及多时钟域接口设计的技巧。

9.5 思考题

1．什么是面积和速度互换原则？试举出一个简单实例。
2．简要说明模块划分原则？
3．代码风格分为哪几重含义？在实践中哪些最容易被忽略？
4．针对 Xilinx 器件的专用代码设计风格包括哪些？
5．什么是流水线技术？编写一个流水线程序的实例。
6．用自己的语言描述一下逻辑复用和逻辑复制技术？
7．为什么要提取设计中的关键路径？有什么具体的实现方法？
8．逻辑合并和逻辑拆分技术的优势是什么？
9．在多时钟域设计中，有哪几种接口的实现方案？

第 10 章　可综合状态机开发实例

状态机是数字逻辑设计的重要内容，经常出现在各种逻样设计中。本章主要介绍常用状态机的几种不同编码方式和描述风格，并从稳定性、可读性、速度和面积等方面比较了不同实现方式的利弊，并分析了采用不同描述风格所得的综合结果。最后，以交通灯的实例来说明 Moore 状态机的实现方法，以电梯控制器的实例来说明 Mealy 状态机的实现方法。在数字系统中，状态机的设计对系统的可靠性、稳定性具有决定性的作用，基于超高速集成电路硬件的有限状态机的设计和优化是完成数字系统设计的重要环节。

10.1 状态机基本概念

10.1.1　状态机工作原理以及分类

1．状态机工作原理基础

状态机是组合逻辑和寄存器逻辑的特殊组合，一般包括两个部分：组合逻辑部分和寄存器逻辑部分；寄存器用于存储状态，组合电路用于状态译码和产生输出信号。状态机的下一个状态及输出不仅与输入信号有关，还与寄存器当前状态有关，其基本要素有三个：状态、输出和输入。

（1）状态

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。例如要设计一个交通灯控制器，可以用允许通行、慢行和禁止通行作为状态；设计一个电梯控制器，每层就是一个状态等等。

（2）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。例如，交通灯控制器就没有输入条件，状态随着时间的改变自动跳转；电梯控制器是存在输入的，每一层的上、下按键以及电梯内的层数选择按键都是输入，会对电梯的下一个状态产生影响。

（3）输出

输出指在某一个状态时特定发生的事件。例如，交通灯控制器在允许通行状态输出绿色，缓行状态输出黄色，禁止通行状态输出红色；电梯控制器在运行时一直会输出当前所在层数以及当前运行方向（上升或下降）。

2．Moore 型和 Mealy 型状态机

根据输出是否与输入信号有关，状态机可以划分为 Mealy 型和 Moore 型状态机；根据输出是否与输入信号同步，状态机可以划分为异步和同步状态机。由于目前电路设计以同步设计为主，因此本书主要介绍同步的 Mealy 型状态机和 Moore 型状态机。

（1）Moore 型状态机

Moore 型状态机的输出仅仅依赖于当前状态（Current State），其逻辑结构如图 10-1 所示。组合逻辑块将输入和当前状态映射为适当的次态（Next State），作为触发器的输入，并在下一个时钟周期的上升沿覆盖当前状态，使得状态机状态发生变化。输出是通过组合逻辑块计算得到的，本质上是当前状态的函数。其中，输出的变化和状态的变化都与时钟信号变化沿保持同步。在实际应用中，大多数 Moore 状态机的输出逻辑都非常简单。

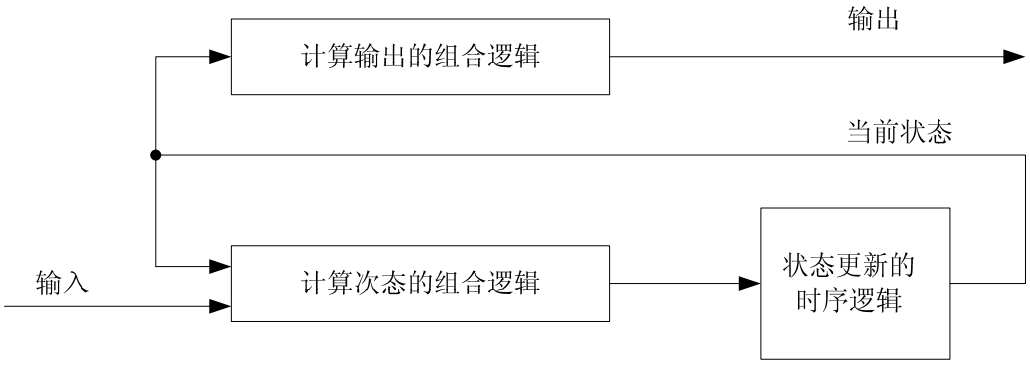


图 10-1　Moore 型状态机结构示意图

（2）Mealy 型状态机

Mealy 状态机的输出同时依赖于当前状态和输入信号，其结构如图 10-2 所示。输出可以在输入发生改变之后立即改变，而和时钟信号无关。因此 Mealy 型状态机具有异步输出。在实际中，Mealy 型状态机应用更加广泛，该类型常常能够减少状态机的状态数。

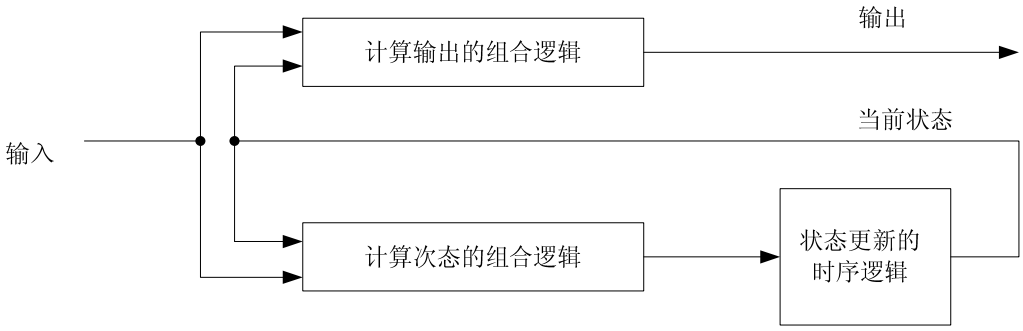


图 10-2　Mealy 型状态机结构示意图

由于 Mealy 型状态机的输出和状态转换有关，因此和 Moore 型状态机相比，只需要更少的状态就可以产生同样的输出序列。此外，还需要注意 Mealy 型状态机的输出和时钟是异步的，而 Moore 型状态机输出却保持同步。

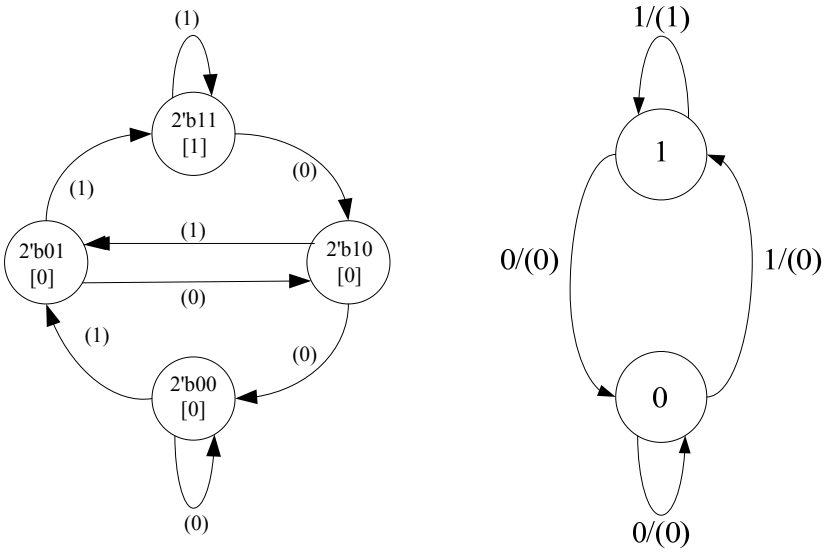
10.1.2　状态机描述方式

状态机有三种表示方法：状态转移图、状态转移表和编程语言描述，这三种表示方法是等价的，相互之间可以转换。

1．状态图表示法

状态转移图是状态机描述的最自然的方式。状态转移图经常在设计规划阶段中定义逻辑功能时使用，也可以在分析代码中状态机的时候使用，通过图形化的方式非常有助于理解设计意图。

下面考虑这样的有限状态机，它具有单个输出，并且任何时候只要输入序列中含有连续的两个 1 时，输出为 1，否则输出为 0。则其最简单的 Moore 型和 Mealy 型状态机分别如图 10-3 所示。其中圆圈表示状态；状态之间的箭头连线表示转移方向，也称作分支；对于 Moore 型状态机，分支上括号内的数字表示输入，状态内“[]”中的数值表示输出；对于 Mealy 型状态机，分支上的数字表示由一个状态转移到另外一个状态的输出信号，而括号中的数字表示相应的输入信号。



（a）Moore 型状态机　　（b）Mealy 型状态机

图 10-3　不同状态机的状态转移图

对于 Moore 型转态机，以当前接收到的比特和上一比特的组合作为状态，共有 4 个状态，在不同状态输出不同的数值。而对于 Mealy 型状态机，由于可以直接利用输入信号来产生输出信号，则直接以上一次接收到的比特为状态，以当前接收到的比特为输入，则只需要两个状态。

值得一提的是，Xilinx 的开发工具 ISE 内嵌的状态机开发工具 StateCAD 就支持以状态转移图作为逻辑设计输入。设计者只要在其中画出状态转移图，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。关于 StateCAD 的详细用法将在 10.4 节进行说明。

2．状态转移表

状态转移列表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，经常用于对状态化简。从表面上看来，状态转移表类似于真值表，下面给出一个简单的状态转移表表述实例。表 10-1 给出了和图 10-3 中 Moore 型状态机对应的状态转移表，读者可以自己完成 Mealy 型状态机的状态转移表。

表 10-1　　Moore 状态机的状态转移表

当前状态	输入	次状态	输出
2'b00	0	2'b00	0
2'b00	1	2'b01	0
2'b01	0	2'b10	0
2'b01	1	2'b11	0
2'b10	0	2'b00	0
2'b10	1	2'b01	0
2'b11	0	2'b10	1
2'b11	1	2'b11	1

基于 EDA 的 Verilog HDL 语言程序设计，主要采用 RTL 级的行为建模，且目前的主流 PLD 器件的可用逻辑资源比较丰富，再加上设计效率、稳定性以及安全性等方面的考虑，所以并不需要通过状态转移表来手工简化、优化状态。

3．编程语言描述

常见的编程语言都可以实现状态机，要将其列为状态机的描述方式可能有些牵强，因为程序大都建立在设计人员已得到状态转移图的基础上才完成的。但这些语言确实也对状态机进行了完整描述，因此本书将其也算作状态机的一种描述方式。如何通过 Verilog HDL 语言描述高质量的状态机就是本章的核心内容。

10.1.3 状态机设计思想

状态机是一类简单的电路，在数字电路以及逻辑设计等课程中属于必修内容，因此大多数读者都了解其概念。但本小节内容想向读者说明的是，状态机不仅仅是一种电路，而且是一种设计思想，贯穿于数字系统设计中。

从电路上讲，状态机可以说是一个广义时序电路，触发器、计数器、移位寄存器都算是它的特殊功能的一种。从功能上讲，状态机可以有效管理系统的各个步骤，类似于 PC 机中的 CPU，包括实现一些非常先进的设计理念。例如，第 9 章介绍的流水线技术、逻辑复用技术等处理方法在实现时都是基于状态机设计思想来完成的。以逻辑复用为例，假设系统处理时钟是数据速率的 4 倍，那么其中就有一个具备四状态的状态机。在每个状态，设计人员都将不同的数据送给复用模块。简而言之，状态机就是数字设计的“大脑”。

事实上，很多初学者不明白如何应用状态机，这是因为不理解状态机的本质。其实状态机就是一种能够描述具有逻辑顺序和时序顺序的事件的方法，特别适合描述那些存在先后顺序以及其它规律性事件。对于要解决的问题，首先按照事件逻辑关系划分出状态；其次，明确各状态的输入、输出以及其相互之间的关系；第三，得到系统的抽象状态转移图，并通过 Verilog HDL 语言实现。所有步骤的目的就是根据需求控制电路。

对于基于 Verilog HDL 语言的设计而言，小到一个简单的时序逻辑，大到整个系统设计都适合用状态机描述。此外，对于设计者而言，状态机的设计水平直接反映了逻辑设计功底，因此读者应该在阅读以及实践过程中注意状态机的设计思想。

10.2 可综合状态机设计原则

状态机作为数字系统的控制器，其设计代码必须面向综合。本节主要介绍状态机开发的方法和常用的设计指标。

10.2.1 状态机开发流程

目前，无论是教育界还是工业界，都在状态机的设计方面积累了丰富的经验。本书推荐的开发流程如下：

（1）理解问题背景。有限状态机的需求常常通过文字描述，准确理解这些描述是明白状态机行为规范的基础。例如，对于最简单的状态机——计数器，简单的枚举状态序列就足够了；而对于复杂的状态机，如无人自动售货机，则需要理解人机交易的所有细节，以及可能出现的种种问题。

（2）得到状态机的抽象表达。一旦读者了解了问题，必须将其变成实现有限状态机过程中更容易处理的抽象形式。最好通过状态转移图将状态机表达出来。这一步是状态机设计的关键。

（3）进行状态简化。步骤（2）得到的抽象表达往往具备很多冗余状态，其中某些特定的状态变化路径可以被清除掉。其中冗余与否的判断准则是：输入/输出行为和其他功能等价的变化路径是否重复。

（4）状态分配。在简单的状态机中，例如计数器，其输出和状态是等价的，因此不需

要在对状态编码进行讨论。但对于一般的状态机，输出并不直接就是状态值，而等同于存储在状态触发器中的比特位（也有可能是某些输入值），因此选择良好的状态编码可以有更改好的性能。10.2.2 节将给出状态编码的详细讨论。

（5）有限状态机的 Verilog HDL 语言实现。这一步骤是最后一步，也有很多优秀的设计方法，本书将在 10.3 节进行介绍。

10.2.2 状态编码原则

状态编码又称状态分配。通常有多种编码方法，编码方案选择得当，设计的电路可以简单；反之，电路会占用过多的逻辑或速度降低。设计时，须综合考虑电路复杂度和电路性能这两个因素。下面主要介绍二进制编码、格雷编码和独热码。

1．二进制编码

二进制编码和格雷码都是压缩状态编码。二进制编码的优点是使用的状态向量最少，但从一个状态转换到相邻状态时，可能有多个比特位发生变化，瞬变次数多，易产生毛刺。二进制编码的表示形式比较通用，这里就不再给出。

2．格雷码

格雷码在相邻状态的转换中，每次只有1个比特位发生变化，虽减少了产生毛刺和一些暂态的可能，但不适用于有很多状态跳转的情况。表10-2给出了十进制数字0~9的格雷的表示形式。

表 10-2　　格雷码数据列表

10进制数字码	格雷码	10进制数字码	格雷码
0	0010	5	1100
1	0110	6	1101
2	0111	7	1111
3	0101	8	1110
4	0100	9	1010

由于在有限状态机中，输出信号经常是通过状态的组合逻辑电路驱动，因此有可能由于输入信号的不同时到达而产生毛刺。如果状态机的所有状态是一个顺序序列，则可通过格雷码编码来消除毛刺，但对于时序逻辑状态机中的复杂分支，格雷编码也不能达到消除毛刺的目的。

3．独热码（One Hot）

独热码是指对任意给定的状态，状态向量中只有1位为1，其余位都为0。*n*状态的状态机需要*n*个触发器。这种状态机的速度与状态的数量无关，仅取决于到某特定状态的转移数量，速度很快。当状态机的状态增加时，如果使用二进制编码，那么状态机速度会明显下降。而采用独热码，虽然多用了触发器，但由于状态译码简单，节省和简化了组合逻辑电路。独热编码还具有设计简单、修改灵活、易于综合和调试等优点。表10-3给出了十进制数字0~9的独热码表示形式。

表 10-3　　独热码数据列表

10进制数字码	独热码	10进制数字码	独热码
0	000_0000_00	5	000_0100_00
1	000_0000_01	6	000_1000_00
2	000_0000_10	7	001_0000_00
3	000_0001_00	8	010_0000_00
4	000_0010_00	9	100_0000_00

对于寄存器数量多、而门逻辑相对缺乏的 FPGA 器件，采用独热编码可以有效提高电路的速度和可靠性，也有利于提高器件资源的利用率。独热编码有很多无效状态，应该确保状态机一旦进入无效状态时，可以立即跳转到确定的已知状态。

10.2.3 状态机的容错处理

在状态机设计中，不可避免地会出现大量剩余状态，所谓的容错处理就是对剩余状态进行处理。若不对剩余状态进行合理的处理，状态机可能进入不可预测的状态（毛刺以及外界环境的不确定性所致），出现短暂失控或者始终无法摆脱剩余状态以至于失去正常功能。因此，状态机中的容错技术是设计人员应该考虑的问题。

当然，对剩余状态的处理要不同程度地耗用逻辑资源，因此设计人员需要在状态机结构、状态编码方式、容错技术及系统的工作速度与资源利用率等诸多方面进行权衡，以得到最佳的状态机。常用的剩余状态处理方法如下所列：

（1）转入空闲状态，等待下一个工作任务的到来；

（2）转入指定的状态，去执行特定任务；

（3）转入预定义的专门处理错误的状态，如预警状态。

在程序编写时，如果通过 if 语句来实现状态调转或者下一状态的计算，不要漏掉 else 分支；如果使用 case 语句则不要漏掉 default 分支。

10.2.4 常用的设计准则

1．基本的设计要求

评价状态机设计的标准很多，下面给出其中最关键的几条准则：

（1）状态机设计要稳定

所谓稳定就是指状态机不会进入死循环，不会进入一些未知状态，即使由于某些不可抗拒原因（系统故障、干扰等）进入不正常状态，也能够很快恢复正常。

（2）工作速度快

由于在设计中，状态机大都面向电路级设计，因此状态机必须满足电路的频率要求。可以尽可能通过 case 语句来代替 if 语句。

（3）所占资源少

在满足工作频率要求的前提下，使用尽可能少的逻辑资源。

（4）代码清晰易懂、易维护

这里面有两个层次的要求：首先，代码书写要规范；其次，要做好文档维护，注重注释语句的添加。

需要说明的，（1）~（3）项要求，不是绝对独立的，它们之间存在相互转化的关系。例如，安全性高就意味着必须处理所有条件判断的分支语句，但这必然导致所用逻辑资源加多；至于面积和速度，二者的互换更是逻辑设计的关键思想。因此，各条要求要综合考虑，但无论如何，稳定性总是第一位的。

2．设计的注意事项

有限状态机的设计准则很多，下面给出常用的注意事项：

（1）单独用一个 Verilog HDL 模块来描述一个有限状态机。这样不仅可以简化状态的定义、修改和调试，还可以利用 EDA 工具（如 ISE 等）来进行优化和综合，以达到更优的效果。

（2）使用代表状态名的参数 parameter 来给状态赋值，而不是用宏定义（`define）。因为宏定义产生的是一个全局的定义，而参数则定义了一个模块内的局部常量。这样当一个设计具有多个有重复状态名的状态机时也不会发生冲突!

（3）在组合 always 块中使用阻塞赋值，在时序 always 块中使用非阻塞赋值。这样可以使软件仿真的结果和真实硬件的结果相一致。

10.3 状态机的 Verilog HDL 实现

在 Verilog HDL 设计中，状态机的代码实现具有一定的开发模板。根据该模板写出来的状态机有着较高的性能和完全的可综合性，能提高设计效率。本节主要介绍两类状态机的经典实现模板。

10.3.1 状态机实现综述

基于 Verilog HDL 语言的状态机设计方法非常灵活，按代码描述方法的不同，可分为一段式描述、二段式描述和三段式描述等。不同的描述所对应的电路是不同的，因此最终的性能也是不同的。为了保证代码的规范性与可靠性，提高代码可读性，下面介绍 3 种常用的描述模板。

1.“一段式”模板

这种方式是将当前状态向量和输出向量用同一个时序 always 块来进行描述，其结构如图 10-4 所示。这样，由于是寄存器输出，输出向量不会产生毛刺，也有利于综合。但是，这种方式有很多缺点，如代码冗长，不易修改和调试、可维护性差且占用资源多；通过 case 语句对输出向量的赋值应是下一个状态的输出，这点较易出错。状态向量和输出向量都由寄存器逻辑实现，面积较大；不能实现异步 Mealy 有限状态机。

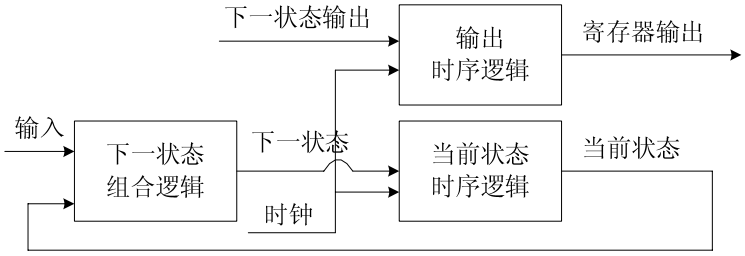


图 10-4　单进程式 Moore 型 FSM 描述结构

“一段式”状态机的 Verilog HDL 代码模板如下：

```
always @(posedge clk) begin
    if (!rst_n) begin
        //
          state <=
          //
          out1 <=
          out2 <=
          ...
      end
      else begin
          case(state)
             s0: begin
                  //
                  state <=
                  //
```

```
            out1 <=
            out2 <=
            ...
        end
        s1: begin
            //
            state <=
            //
            out1 <=
            out2 <=
            ...
        end
        ...
    endcase
  end
end
```

2．“两段式”模板

这种方式中，一个时序 always 块给当前状态向量赋值，一个组合 always 块给下一状态和输出向量赋值，通常用于描述组合输出的 Moore 状态机或异步 Mealy 状态机，其结构如图 10-5 所示。

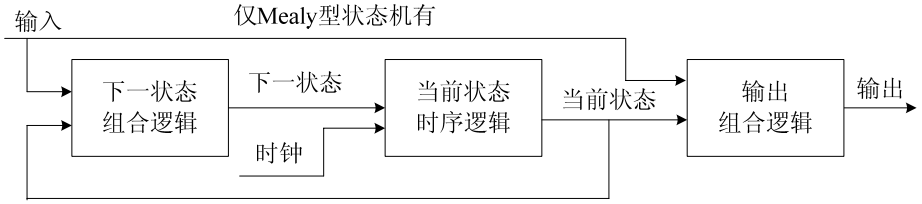


图 10-5　两段式有限状态机结构

与 1 个 always 模板和 3 个 always 块模板相比，这种方式具有最优的面积和时序性能，但缺点是其输出为当前状态的组合函数，因此存在以下几个问题：

（1）组合逻辑输出会使输出向量产生毛刺。一般情况下，输出向量的毛刺对电路的影响可以忽略不计。但是，当输出向量作为三态使能控制或者时钟信号使用的时候，就必须要消除毛刺，否则会对后面的电路产生致命的影响。

（2）从速度角度而言，由于这种状态机的输出向量必须由状态向量经译码得到，因此加大了从状态向量到输出向量的延时。

（3）从综合角度而言，组合输出消耗了一部分时钟周期，即增加了由它驱动的下一个模块的输入延时。这样不利于综合脚本的编写和综合优化算法的实现。综合的基本技巧是将一个设计划分成只有寄存器输出，且所有的组合逻辑仅存在于模块输入端以及内部寄存器之间的各个子模块。这样不仅能在综合脚本中使用统一的输入延时，还能得到更优化的综合结果。

“二段式”状态机的 Verilog HDL 代码模板如下：

```
//状态调转
always @(posedge clk) begin
    if (!rst_n)
        state <= idle;
    else
```

```
            state <= next_state;
    end

    //下一状态的计算以及输出逻辑
    always @(state) begin
        case(state)
            s0: begin
                //
                  next_state = ;
                  //
                  out1       = ;
                  out2       = ;
                  ..
             end
             s1: begin
                //
                  next_state = ;
                  //
                  out1       = ;
                  out2       = ;
             end
             ...

         endcase
    end
```

3．“三段式”模板

“三段式”式用两个时序 always 模块分别用于产生当前状态向量和输出向量，一个组合 always 用于产生下一状态向量，其结构如图 10-6 所示。

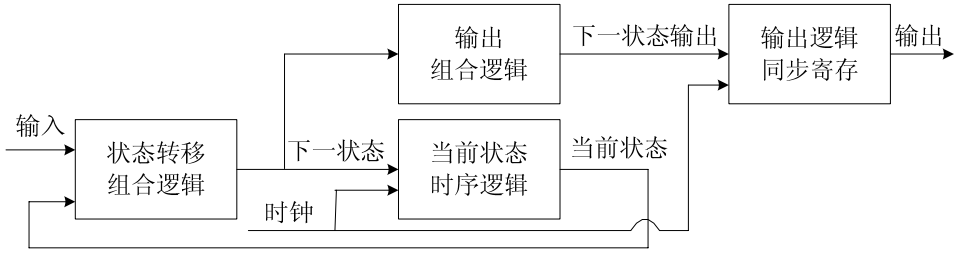


图 10-6　三进程式 Moore 型 FSM 描述结构

“三段式”代码主要包括以下 3 个部分：

（1）状态转移部分

这部分定义了基于时钟的寄存器，对状态值进行不同的编码可以得到不同的寄存器组类型。例如：独热码可以简化输出组合逻辑，但是消耗了更多寄存器资源；而采用格雷码，由于每次状态变化时只改变一个比特位，因此可以降低功耗。

（2）状态转移条件部分

此部分是纯组合逻辑，实现了状态转移的条件判断。在这部分中，如果某一状态下通过不同条件进入不同状态，则应仔细考虑这些条件间的优先级。

（3）输出逻辑部分

根据不同需要可以有多种方式实现。比如处于某状态时，输出一个时钟周期的信号，或多个时钟周期的信号；也可以在进入某状态的时候，输出一个或多个时钟周期的信号。

这个模板与 1 个 always 块模板风格相比，同样是寄存器输出，但面积较小，代码可读性强；与 2 个 always 块模板风格相比面积稍大，但具有无毛刺的输出且有利于综合，因此推荐读者使用 3 个 always 块模板。

“三段式”状态机的 Verilog HDL 代码模板如下：

```
//状态调转
always @(posedge clk) begin
    if (!rst_n)
        state <= idle;
     else
        state <= next_state;
end

//下一状态的计算
always @(state) begin
    case(state)
        s0: next_state = ;
        s1: next_state = ;
          ...
     endcase
end

//输出逻辑的处理
always @(posedge clk) begin
  case(state)
      s0: begin
            out1 <= ;
              out2 <= ;
              ...
       end
      s1: begin
            out1 <= ;
              out2 <= ;
              ...
       end
      ...
  end
end
```

10.3.2 Moore 状态机开发实例

本节给出一个典型的 Moore 状态机的应用实例——交通灯控制的完整开发。其基本要求如下所列：

（1）交通灯控制器工作在十字路口交叉处，如图 10-7 所示。由于南北通路为人行道，

东西为动车道，因此只需要考虑南北方向和东西方向的指示灯，不涉及南北通道和东西通路之间的交叉转向指示。

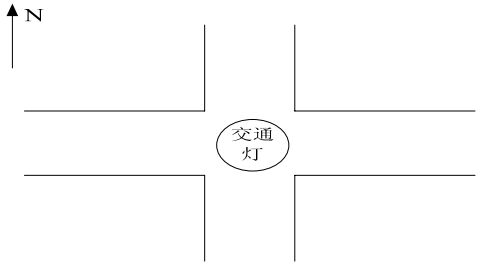


图 10-7　交通灯工作示意图

（2）其中每条通路的红灯、绿灯的持续时间都为 15s。

例 10-1：使用 Verilog HDL 语言实现上述交通灯控制器，并给出功能仿真结果。

```
module jtd(
      clk_1Hz, rst_n,
       red_ew, green_ew,
       red_ns, green_ns
      );

       input     clk_1Hz, rst_n;
       output    red_ew,green_ew;
       output    red_ns, green_ns;

       reg        red_ew, green_ew;
       reg        red_ns, green_ns;

       reg        state, next_state = 0;
       reg    [4:0] cnt;

       //
       always @(posedge clk_1Hz) begin
           if(!rst_n)
                 cnt <= 0;
          else
                 if(cnt == 29)
                       cnt <= 0;
                  else
                       cnt <= cnt + 1;
       end

       always @(posedge clk_1Hz) begin
           if(!rst_n)
                 state <= 1'b0;
          else
                 state <= next_state;
```

```
    end

    always @(state, cnt) begin
      case(state)
            1'b0: begin
                  if(cnt == 14)
                        next_state = 1'b1;
                  else
                        next_state = 1'b0;
             end
            1'b1: begin
                  if(cnt == 29)
                        next_state = 1'b0;
                  else
                        next_state = 1'b1;
             end

         endcase
    end

    always @(posedge clk_1Hz) begin
        case(next_state)
              1'b0: begin
                    red_ew     <= 1;
                     green_ew <= 0;
                     red_ns     <= 0;
                     green_ns <= 1;
               end
               1'b1: begin
                    red_ew     <= 0;
                     green_ew <= 1;
                     red_ns     <= 1;
                     green_ns <= 0;
               end
         endcase
    end

endmodule
```

为了完成上述程序的仿真，读者可在 ISE 中新建“Verilog Test Fixture”类型的源文件来创建 Testbench，并添加下列内容。

```
module tb_jtd;

    // Inputs
    reg clk_1Hz;
```

```
reg rst_n;

// Outputs
wire red_ew;
wire green_ew;
wire red_ns;
wire green_ns;

// Instantiate the Unit Under Test (UUT)
jtd uut (
        .clk_1Hz(clk_1Hz),
        .rst_n(rst_n),
        .red_ew(red_ew),
        .green_ew(green_ew),
        .red_ns(red_ns),
        .green_ns(green_ns)
);

initial begin
        // Initialize Inputs
        clk_1Hz = 0;
        rst_n = 0;
        // Wait 100 ns for global reset to finish
        #100;
        rst_n = 1;
end

always    #1 clk_1Hz = !clk_1Hz;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 10-8 所示，可以看出实现了成对红、绿灯的 15s 周期翻转，达到了设计要求。

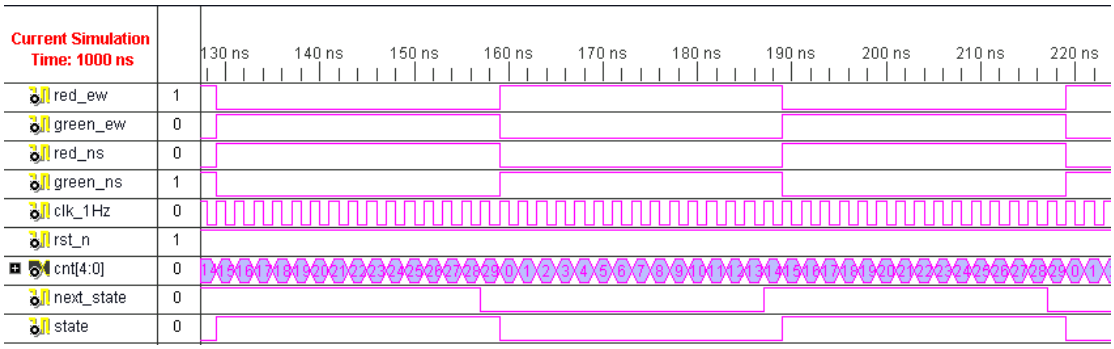


图 10-8　交通灯控制器的仿真结果

10.3.3 Mealy 状态机开发实例

例 10-2：利用 Verilog HDL 语言实现一个基于一段式 Mealy 状态机的序列检测器，当输入数据依次为 10010 时，输出一个脉冲。

本例的实现代码如下：

```
module xljcq(clk,reset,din,signalout);
    input clk,din,reset;
    output signalout;
    reg [2:0] state;

    parameter
        idle = 3'd0,
        a = 3'd1, //5'b1xxxx
        b = 3'd2, //5'b10xxx
        c = 3'd3, //5'b100xx
        d = 3'd4, //5'b1001x
        e = 3'd5; //5'b10010

    //根据状态机判断输出
    assign signalout = (state == e)?1:0;

    always@(posedge clk)
        if(!reset)
            begin
                state <= idle;
            end
        else
            begin
                casex(state)
                    idle:
                        begin
                            if(din == 1)
                                state <= a;
                            else
                                state <= idle;
                        end
                    a:
                        begin
                            if(din == 0)
                                state <= b;
                            else
                                state <= a;
                        end
                    b:
                        begin
```

```
                    if(din == 0)
                        state <= c;
                    else
                        state <= a;
                end
            c:
                begin
                    if(din == 1)
                        state <= d;
                    else
                        state <= idle;
                end
            d:
                begin
                    if(din == 0)
                        state <= e;
                    else
                        state <= a;
                end
            e:
                begin
                    if(din == 0)
                        state <= c;
                    else
                        state <= a;
                end
            default:
                state <= idle;
        endcase
    end
endmodule
```

上述程序在综合时，ISE 会在信息显示区输出状态机的编码信息，如图 10-9 所示，表明本例采用格雷码来实现状态编码。

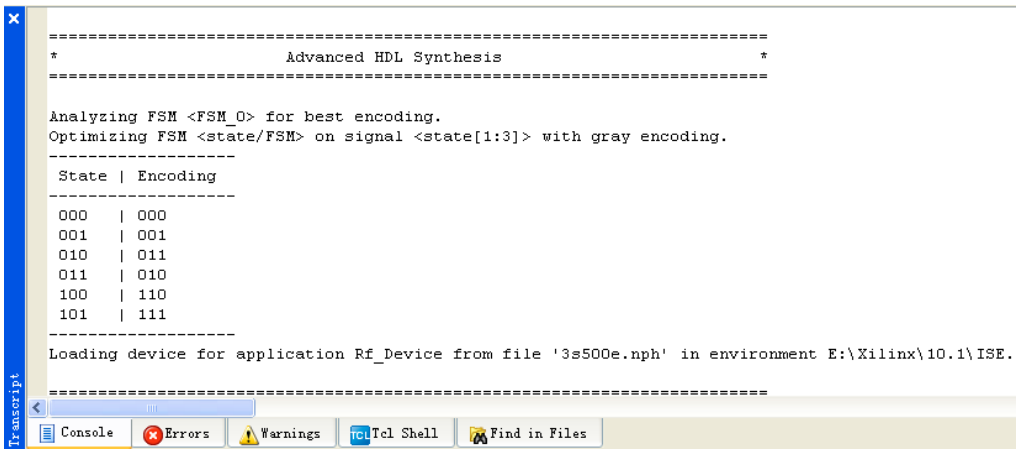


图 10-9　状态编码指示信息示意图

上述程序在 ISE Simulator 中的仿真结果如图 10-10 所示，表明序列检测器工作正常，可满足设计要求。

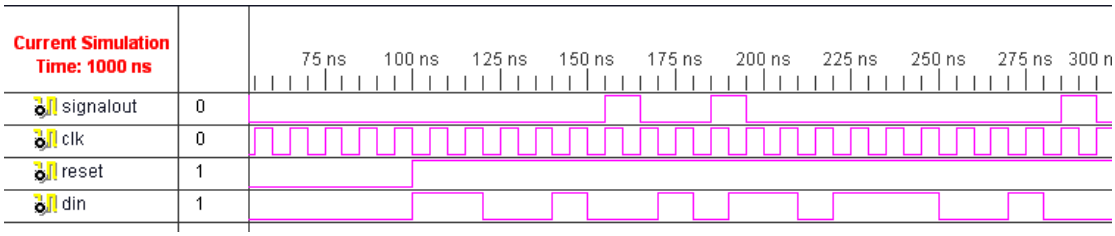


图 10-10　自动售货机的仿真结果

10.4 Xilinx 状态机设计工具 StateCAD

由于状态机是系统各部分正常工作的指挥棒，在设计中具有非常重要的地位，必须具备良好的鲁棒性和强的容错性处理。因此，ISE 中自带了状态机设计工具 StateCAD。StateCAD 使状态机的设计变得简单、高效且可靠，只需要设计出状态转移图，StateCAD 会自动将其翻译成 HDL 代码，包括 VHDL、Verilog 以及 ABEL 语言。此外，其还能根据用户所选择的芯片型号和综合器类型对可综合代码进行优化，并生成相应的测试激励来验证状态机的正确性。

10.4.1　StateCAD 基础介绍

StateCAD 能自动检测状态机的完备性和正确性，对状态转移图中的冗余状态、自锁状态、歧义转移条件和不完备状态机等隐含错误都会报警，并协助设计者更正错误。最后 StateCAD 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、优化和测试等完整开发流程，使状态机的设计变得安全、可靠、快速、便捷。这类自动转换状态转移图为 HDL 源代码的工具，对于设计、分析状态机非常有效。

StateCAD 是独立于 ISE 的第三方设计软件，有两种启动方式：一种是单独启动 StateCAD，直接点击“开始 → 程序 → Xilinx ISE Design Suit 10.1 → ISE 10.1 → Accessories → StateCAD”命令；二是在 ISE 中单击“New Source”命令，在代码类型中选择 State Machine 选项，在 File Name 文本框中输入“mystmachine”，单击“Next”进入状态机编辑界面，如图 10-11 所示。

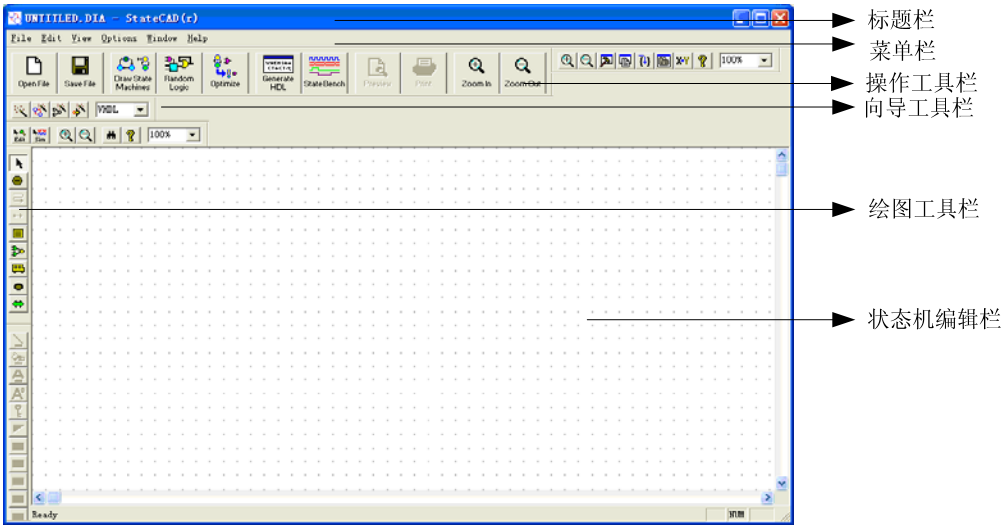


图 10-11　状态机编辑界面示意图

状态机的编辑界面比较简单，所有操作可通过操作工具栏按键快速完成，如图 10-12 所示。下面对操作工具栏的常用操作进行简单介绍。

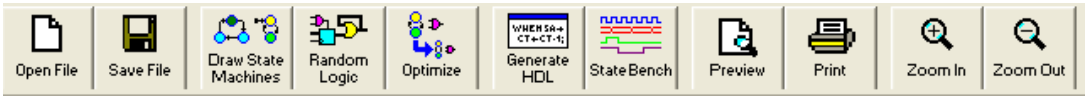


图 10-12　操作工具栏示意图

【Open File】：打开状态机设计文件，其后缀为.DIA。

【Save File】：保存当前设计。

【Draw State Machines】：启动状态机设计向导，是状态机设计的第一步。

【Random Logic】：启动逻辑设计向导，用于辅助设计数据流的逻辑规则。

【Optimize】：启动优化向导，使得设计在速度、面积、I/O 端口以及综合结果等方面达到整体优化；此外，还能设置 HDL 语言类型。

【Generator HDL】：编译状态图设计文件，并生成 HDL 代码。

【State Bench】：设计状态机的测试激励，通过功能仿真来测试生成逻辑的正确性。

【Preview】：打印预览。

【Print】：打印。

【Zoom In】：放大视图。

【Zoom Out】：缩小视图。

由此可以看出，通过 StateCAD 设计状态机的流程为：编辑状态机、优化状态机并生成 HDL 代码以及测试状态机。

10.4.2　编辑状态机

下面通过一个实例来介绍如何编辑状态机。

例 10-3：通过 StateCAD 设计一个具有 3 个转移状态的同步状态机。当系统上电后，进入“IDLE”状态，使信号 Reset_OK 为低；然后无条件进入“CONFIGURE”状态，将 Reset_OK 信号拉高、Configure_OK 信号拉低，经过 16 个时钟后，拉高 Configure_OK 信号，进入“Work 状态”；在“Work”状态，拉高 Work_OK 信号，经过 1024 个时钟周期后，进入到“IDLE”状态。

① 点击“”（Draw State Machine）按钮，进入状态机编辑向导，如图 10-13 所示，有列状转移图（Column）、多列状转移图（Muti-Column）、行状转移图（Row）和几何转移图（Geometric）等 4 类转移图。其中几何转移图最适用。所以选择“Geometric”选项，并将“Number of states”修改为 3。

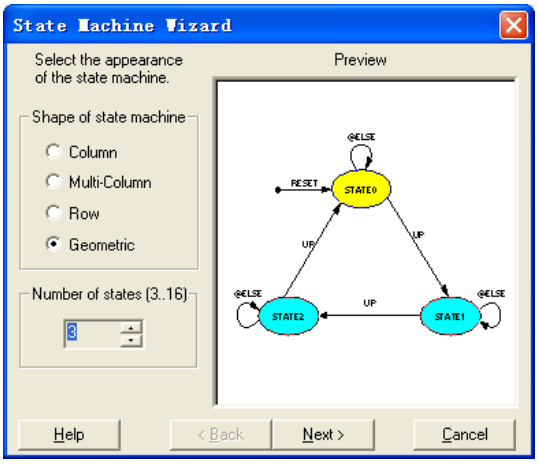


图 10-13　状态机转移类型和状态数选择示意图

② 点击“Next”按键后，进入状态机复位机制选择界面，如图 10-14 所示，有同步复位（Asynchronous）和异步复位（synchronous）两种方式。一般为了可靠性，需要选择同步复位模式。

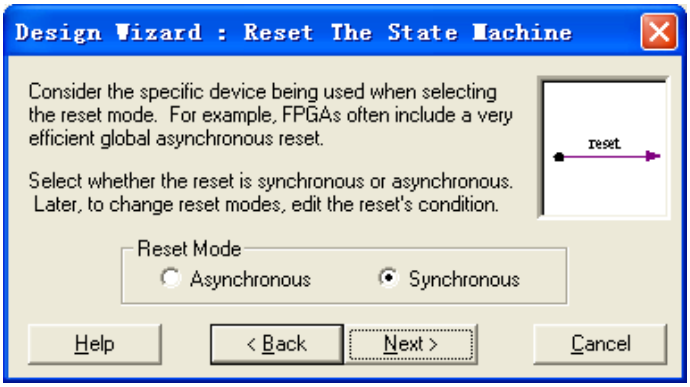


图 10-14　状态机复位模式选择示意图

③ 点击“Next”按键，进入状态机转移方式，如图 10-15 所示，分为自循环（Loop back）、下行（Next）和上行（Previous）三种模式。一般常用的是下行模式。不过三种模式可以同时复选，如本例就选择了自循环和下行模式。

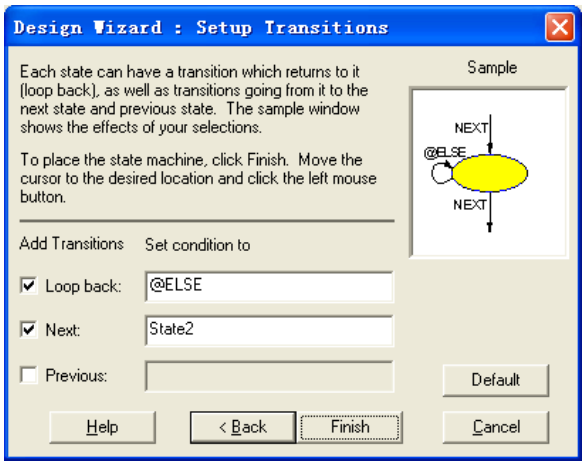


图 10-15　状态机转移模式选择示意图

④ 单击“Finish”按键，完成状态机配置向导，进入用户修改界面，如图 10-16 所示。

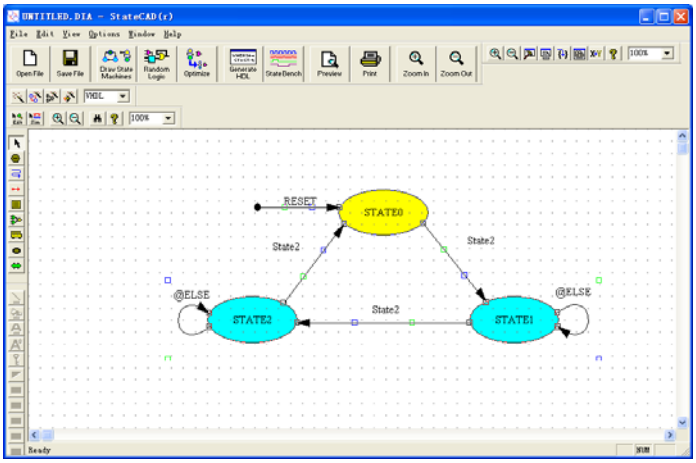
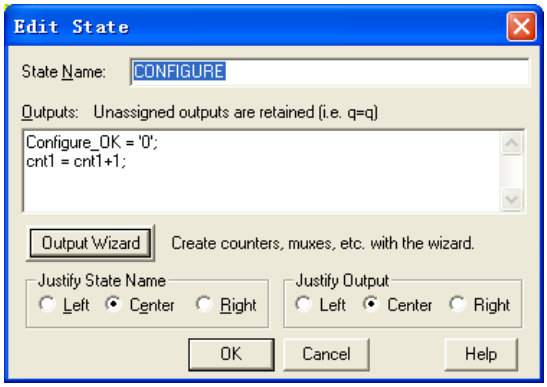


图 10-16　自动向导生成的状态转移示意图

双击其中任何一个状态都可以进入状态编辑窗，如图 10-17 所示，可修改其状态名、输出信号和输出逻辑。状态名可直接在“State Name”栏修改；输出信号和输出逻辑可以在“Outputs”栏输入，也可通过点击“Output Wizard”按键，进入输出逻辑向导，选择更丰富的逻辑资源，包括常数、D 触发器、逻辑操作、移位、复用器以及计数器等类型的单元模块，如图 10-18 所示。如本例中，在“CONFIGURE”状态需要一个模为 16 的 4 比特计数器，在“Work”状态需要一个模为 1024 的 10 比特计数器，都可通过选择“Counter UP”类型的逻辑单元来实现，添加合适的信号名后，并将相应的位宽设置为 4 和 10，单击“OK”即可完成输出逻辑的设置。



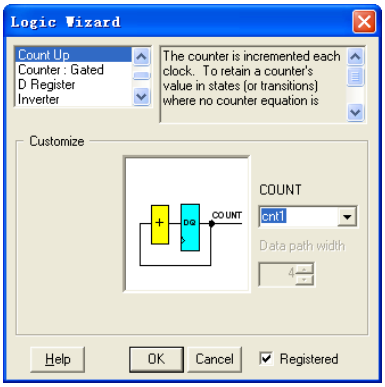


图 10-17　状态编辑窗口示意图　　图 10-18　输出逻辑选择窗口示意图

按照上述方法，将 State0、State1 以及 State2 的名字分别修改为 IDLE、CONFIGURE 以及 WORK，依次添加输出 Reset_OK、Configure_OK、Work_OK 信号，并全部拉低，最后分别在 State1 以及 State2 状态内添加 4 比特计数器 cnt1 和 16 比特计数器 cnt2。

⑤ 添加状态转移逻辑。双击任意两个状态之间的转移路径，即可进入状态条件编辑框，如图 10-19 所示。其输出逻辑的添加方法和状态输出逻辑的添加方法是一样的。

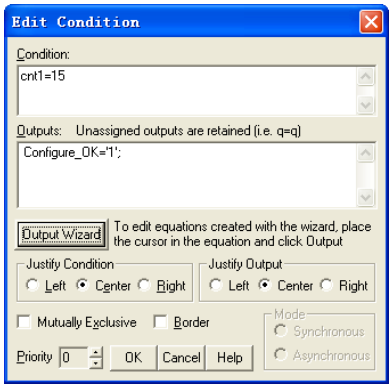


图 10-19　状态转移条件编辑窗口示意图

在本例中，从 IDLE 状态到 CONFIGURE 状态无转移条件，输出 Reset_OK='1'；从 CONFIGURE 到 WORK 的状态转移条件为 cnt1=15，输出 Configure_OK='1'；从 WORK 到 IDLE 状态的转移条件为 cnt2=1023，输出 Work_OK='0'。至此，状态机的设计步骤完成，最终的设计结果如图 10-20 所示。

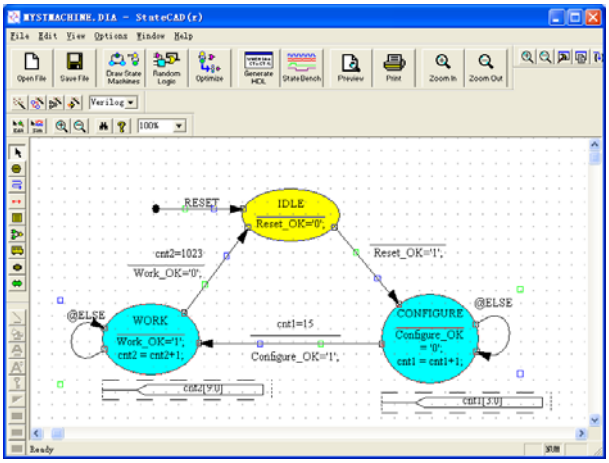


图 10-20　最终的状态机设计示意图

10.4.3　状态机优化以及 HDL 代码生成

设计完状态机后，就可以进入优化和代码生成阶段。

例 10-4：完成例 10-3 的优化，并生成相应的 Verilog HDL 代码。

① 在例 10-3 的基础上，单击操作工具栏的“Optimize”按键，进入优化向导，如图 10-21 所示，说明了优化操化设置的五个步骤。

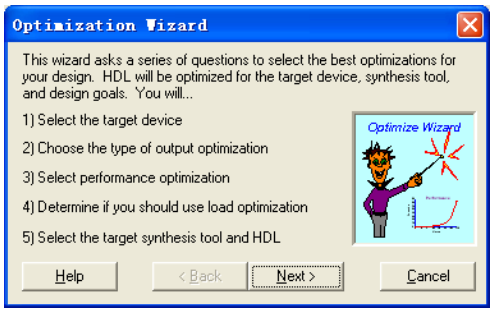


图 10-21　优化向导界面示意图

② 点击“Next”按键，进入五步优化的第一步，选择目标器件，有“FPGA”、“CPLD/PAL”以及“Simulation on”三种选项，本例选择“FPGA”，如图 10-22 所示。

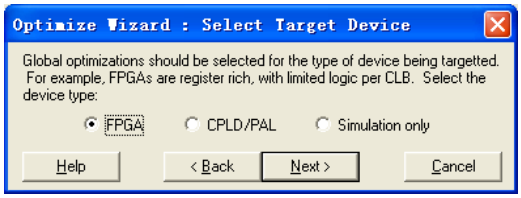


图 10-22　目标器件选择界面示意图

③ 点击“Next”按键，进入五步优化的第二步，优化目标选择界面，有 Manual、Speed 和 Area 三种选项。FPGA 设计优化就是时序性能和面积资源的平衡。一般来讲，为了保证状态机的稳定性，大多牺牲面积来换取性能，本例选择“Speed”，如图 10-23 所示。

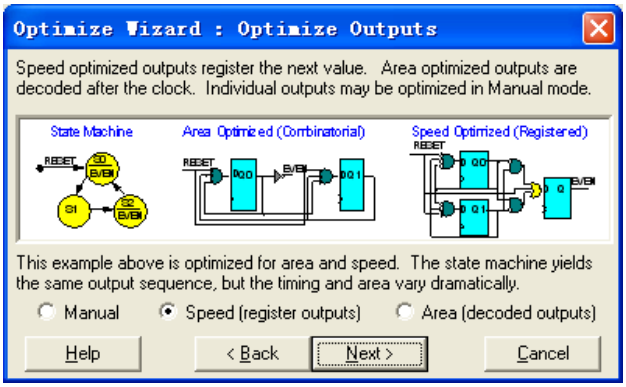


图 10-23　设计性能选择界面示意图

④ 点击“Next”按键，进入五步优化的第三步，提高性能界面，有“Guarantee coverage”和“Maximize speed, reduce area”两类选项。前者在设计中添加了对转移条件的完整逻辑判断，即会添加转移条件所有分支的判断处理语句；而后者只有转移条件的处理分支，逻辑判断级数相对较少，速率快且面积少，但容错性极差。一般为了保证状态机的稳定性，需选择前者，本例的选择也是前者，如图 10-24 所示。

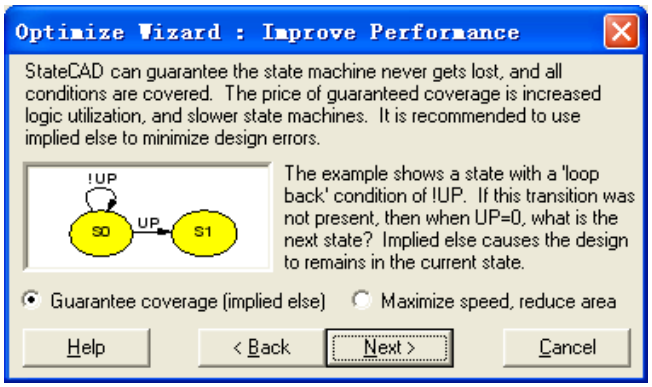


图 10-24　提高性能界面示意图

⑤ 点击“Next”按键，进入五步优化的第四步，加载类型选择界面，可添加反馈信号缓冲，从而提高设计性能。本例选择默认值，如图 10-25 所示。

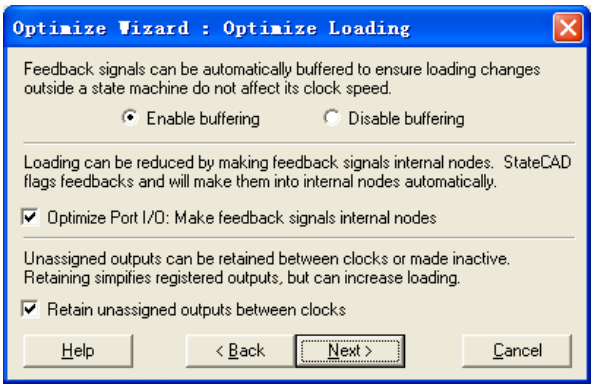


图 10-25　加载类型选择界面示意图

⑥ 点击“Next”按键，进入五步优化的第五步，选择 HDL 语言类型和综合工具，本例选择 Verilog 语言，并且选择 Xilinx XST 作为综合工具，如图 10-26 和图 10-27 所示。

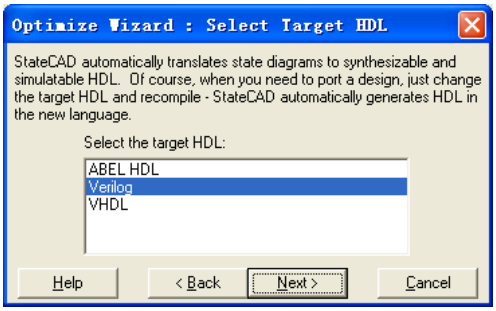


图 10-26　HDL 语言类型选择界面示意图

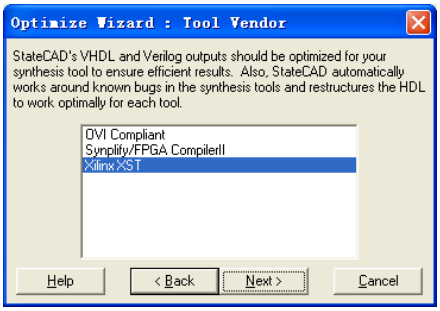


图 10-27　综合工具选择界面示意图

⑦ 点击“Next”按键完成设计优化过程。点击“Generator HDL”按键，编译设计并生成 HDL 代码，则弹出输出信号选择界面，如图 10-28 所示，点击去掉信号前面的“×”即可将其设置为输出信号，再单击“Optimize”按键，可完成状态机的编译，正确编译结果如图 10-29 所示。

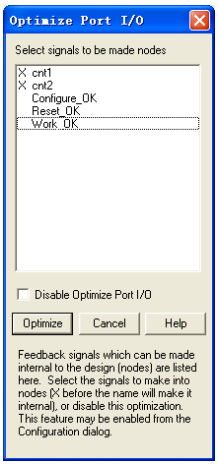


图 10-28　端口选择界面示意图

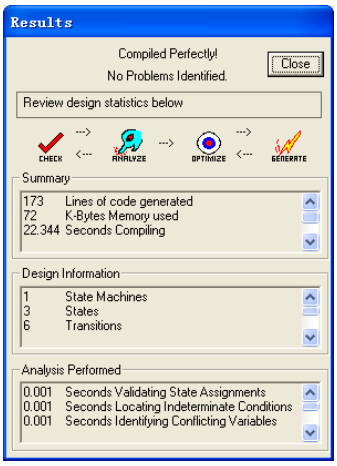


图 10-29　编译结果界面示意图

点击图 10-29 中的“Close”按键关闭编译结果，即可看到生成的 HDL 代码，如图 10-30 所示。该代码可作为普通 Verilog 模块，被设计者任意例化调用。至此，已完成状态机的优化、编译和代码生成。

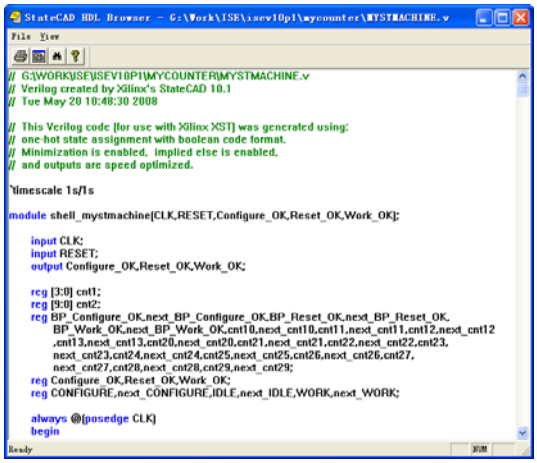


图 10-30　所生成 HDL 代码示意图

10.4.4　测试状态机

行为级测试是状态机设计的最后一步。下面通过实例介绍 StateCAD 的测试机制。

例 10-5：在例 10-4 的基础上利用 StateCAD 完成测试。

① 单击“State Bench”按键，可进入仿真测试对话框，如图 10-31 所示。从中可以看出，有状态转移显示页面和测试激励输入页面上下两部分。

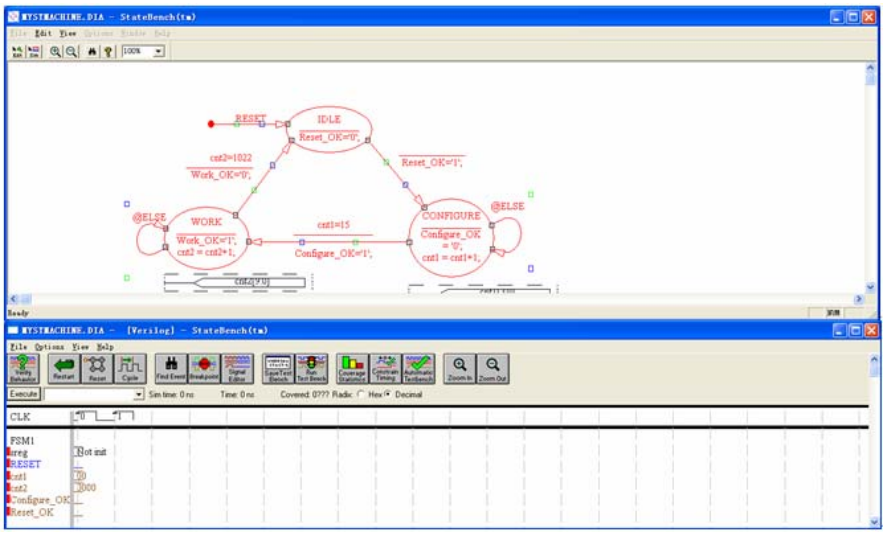


图 10-31　状态机测试界面示意图

② 点击测试激励页面的“Verify Bench”按键，即可进入仿真设置对话框，包括初始化条件、测试目标和测试信号，如图 10-32 所示。

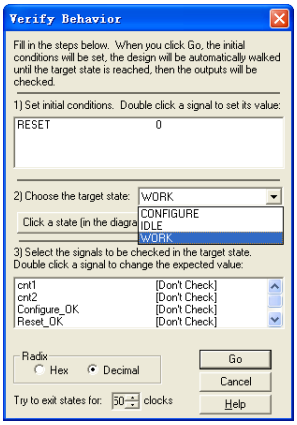


图 10-32　状态机测试激励配置页面示意图

③ 单击“OK”按键可运行测试激励，测试结果如图 10-33 所示，同时状态跳转测试页面会将当前所处状态高亮显示。从中可以看出，例 10-3~例 10-5 实现了预定目标。

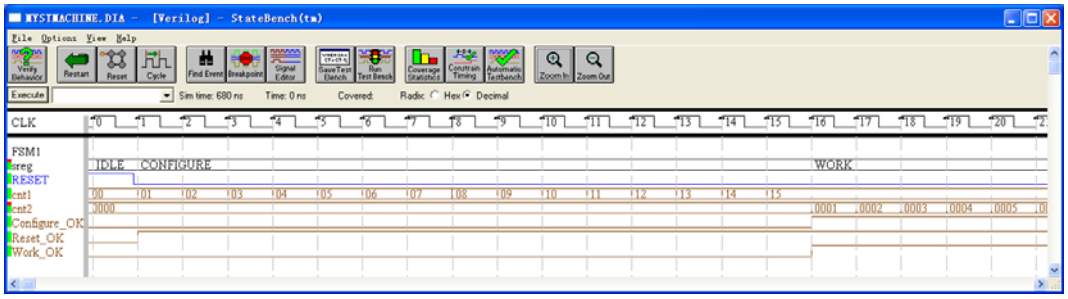


图 10-33　状态机测试结果示意图

10.5 本章小结

本章主要围绕着数字系统中状态机的设计展开。首先，介绍了状态机的工作原理、分类、描述方式以及设计思想，指出状态机主要分为 Moore 型和 Mealy 型两大类，是现代电子系统中最关键的基本单元，是系统的控制中枢。其次，介绍了可综合状态机的设计原则，包括状态机开发流程、状态编码原则、状态机的复位与容错处理，并给出常用的设计准则，为状

态机的 Verilog HDL 实现做好准备。第三，是本章的重点，详细阐述状态机的实现方法，给出 3 类实现模板，并比较了其优、缺点，推荐读者采用三段式描述方法；针对 Moore 型和 Mealy 型状态机，分别给出交通灯控制器和序列检测器的开发实例，希望读者从阅读中加深对状态机实现的理解。最后，介绍了 ISE 自带的状态机设计工具 StateCAD 组件的使用方法，并给出完整的开发实例。

10.6 思考题

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.

第 11 章　常用逻辑的 Verilog HDL 实现

Verilog HDL 语言作为一种通用硬件描述语言，既可以完成数字系统核心处理，又可以完成外围控制芯片、存储芯片和显示器件的接口。加上可编程逻辑器件应用的日益广泛，有些基本模块的使用频率会相对较高。为了让读者更好掌握 Verilog HDL 语言，并学以致用，本章给出了一些难度适中的常用的 Verilog HDL 开发实例，包括时钟电路、数学运算、数码管、按键、CRC 编解码、存储器以及 SPI 接口协议等。

11.1　时钟处理电路的 Verilog HDL 实现

在 CPLD/FPGA 设计中，时钟可以算作系统的“血液”。在时序电路设计中，几乎所有的信号都需要依靠时钟向前传递，因此在进行 VHDL 开发之前需要确定所需的时钟频率。如果是同步设计，还需要对异步输入信号进行同步整形处理。本节主要介绍分频电路、DCM/PLL 模块以及同步整形电路的原理和实现。

11.1.1　整数分频模块

分频器是一种基本电路，通常用来对某个给定频率进行分频，以得到所需的频率。整数分频是数字逻辑开发中最基本的应用。

1．偶数分频模块

偶数倍分频是最简单的一种分频模式，可通过计数器计数实现，有多种实现方法。下面介绍一种最常用的方法，如要进行 N 倍偶数分频，那么可由待分频的时钟触发计数器计数，当计数器从 0 计数到 N/2-1 时，输出时钟进行翻转，并给计数器一个复位信号，使得下一个时钟从零开始计数，以此循环下去。这种方法可以实现任意的偶数分频。例 11-1 给出的是一个 16 分频电路，其它倍数的分频电路可通过修改计数器的上限值得到。

例 11-1：使用 Verilog HDL 程序完成 16 分频电路。

```
module clk_div_16(
    clk_in, rst_n, clk_out
    );

     input     clk_in;
     input     rst_n;
     output    clk_out;

     reg        [2:0] cnt;
     reg        clk_out_t;

     always @(posedge clk_in) begin
        if (!rst_n) begin
            cnt <= 0;
             clk_out_t <= 0;
       end
       else begin
```

```
            if (cnt == 3'b111) begin
                cnt <= 3'b000;
                 clk_out_t <= ~clk_out_t;
              end
              else begin
                cnt <= cnt    + 3'b001;
              end
        end
    end

    assign clk_out = clk_out_t;

endmodule
```

上述程序的仿真结果如图 11-1 所示，从中可以看出，在复位信号 rst 为高的情况下，一个 clk_out 的周期内有 16 个 clk_in 信号，表明 clk_out 的频率是 clk_in 的 1/16，达到了 16 分频的目的。

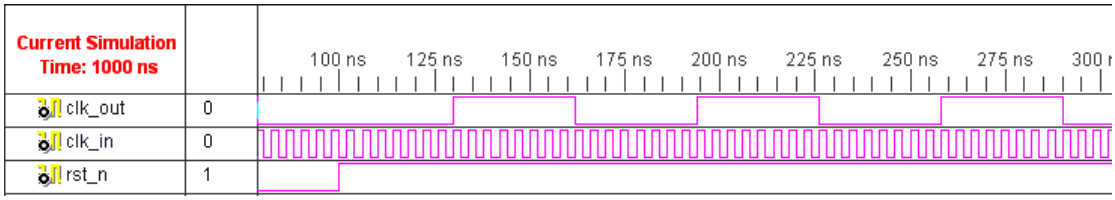


图 11-1　16 分频电路仿真结果示意图

2．奇数分频模块

奇数倍分频有多种实现方法，下面介绍常用的错位“异或”法的原理。如进行三分频，通过待分频时钟上升沿触发计数器进行模三计数，当计数器计数到邻近值进行两次翻转。比如在计数器计数到 1 时，输出时钟进行翻转，计数到 2 时再次进行翻转，即在邻近的 1 和 2 时刻进行两次翻转。这样实现的三分频占空比为 1/3 或者 2/3。如果要实现占空比为 50%的三分频时钟，可以通过待分频时钟下降沿触发计数，和上升沿同样的方法计数进行三分频，然后将下降沿产生的三分频时钟和上升沿产生的时钟进行相或运算，即可得到占空比为 50%的三分频时钟。

这种错位“异或”法可以推广实现任意的奇数分频：对于实现占空比为 50%的 N 倍奇数分频，首先进行上升沿触发的模 N 计数，计数到某一选定值时进行输出时钟翻转，然后经过(N-1)/2 再次进行翻转得到一个占空比非 50%奇数 N 分频时钟。再者同时进行下降沿触发的模 N 计数，到和上升沿触发输出时钟翻转选定值相同值时，进行输出时钟时钟翻转，同样经过(N-1)/2 时，输出时钟再次翻转生成占空比非 50%的奇数 N 分频时钟。两个占空比非 50%的 N 分频时钟相或运算，得到占空比为 50%的奇数 N 分频时钟。

例 11-2：使用 Verilog HDL 语言实现 3 分频电路，要求占空比达到 50%。

```
module clk_div_3(clk_in, rst_n, clk_out);
   input clk_in;
   input rst_n;
   output clk_out;

   reg [1:0] cnt, cnt1;
   reg       clk_1to3p, clk_1to3n;
```

```
    always @(posedge clk_in) begin    //上升沿 3 分频，占空比为 1:2
        if(!rst_n) begin
            cnt <= 0;
             clk_1to3p <= 0;
        end
        else begin
            if(cnt == 2'b10) begin
                cnt <= 0;
                 clk_1to3p <= clk_1to3p;
            end
            else begin
                cnt <= cnt + 1;
                 clk_1to3p <= !clk_1to3p;
            end
        end
    end

    always @(negedge clk_in) begin //下降沿 3 分频，占空比为 1:2
        if(!rst_n) begin
            cnt1 <= 0;
             clk_1to3n <= 0;
        end
        else begin
            if(cnt1 == 2'b10) begin
                cnt1 <= 0;
                 clk_1to3n <= clk_1to3n;
            end
            else begin
                cnt1 <= cnt1 + 1;
              clk_1to3n <= !clk_1to3n;
            end
        end
    end

    assign clk_out = clk_1to3p | clk_1to3n; //错位相或

endmodule
```

上述程序的仿真结果如图 11-2 所示，从中可以看出，在复位信号 rst 为高的情况下，一个 clk_out 的周期内有 3 个 clk_in 信号，表明 clk_out 的频率是 clk_in 的 1/3，达到了 3 分频的目的。需要注意的是：经过错位“异或”法得到的分频时钟，其相位和输入时钟是不对齐的。

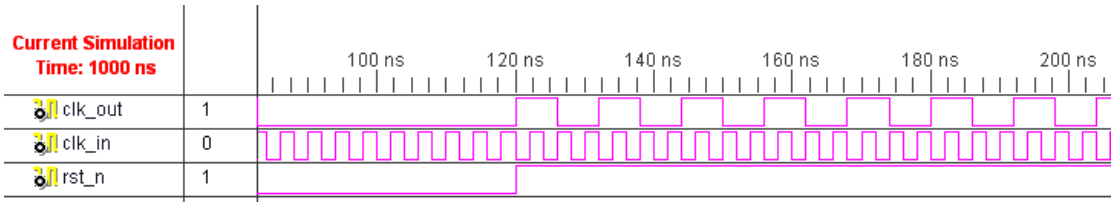


图 11-2　3 分频电路仿真结果示意图

11.1.2　非整数分频模块

非整数分频模块有两种实现方法，分别为分频比交错法和累加器分频法。下面分别进行介绍。

1．分频比交错法

分频比交错法，顾名思义就是在一定时间间隔 T 内，由不同的分频比电路交叉着对输入信号进行分频，从而在 T 时间内达到小数分频的目的。假设要实现 8.666666 分频，则可以以 6 次分频为一个周期，每个周期内进行 2 次 9 分频和 4 次 8 分频，这样，输出 F_OUT 平均为 F_IN 的 8.666666 分频，该类分频器的程序结构如图 11-3 所示。在这种方法中，为使分频输出信号的占空比尽可能均匀，8 分频和 9 分频应交替进行。

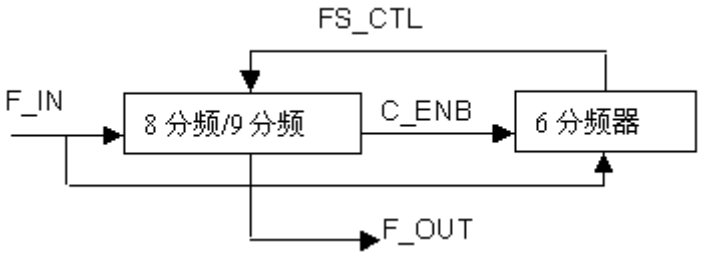


图 11-3　分频比交错法电路结构示意图

例 11-3：利用 Verilog HDL 语言通过分频比交错法完成 8.66 分频的电路。

```
module clk_div_8p6(
    clk_in, rst_n, clk_out
    );

    input     clk_in;
    input     rst_n;
    output    clk_out;

    reg    [3:0]    cnt1;
    reg    [2:0]    cnt2;
    reg             clk_out_t;
    reg             fs_ctl;

    assign clk_out = clk_out_t;

    always @(posedge clk_in) begin
        if(!rst_n) begin
            cnt1 <= 4'b0000;
            cnt2 <= 3'b000;
```

```
            clk_out_t <= 1'b0;
        end
        else begin
            case(fs_ctl)
                1'b1: begin //9 分频模块处理电路
                    if (cnt1 == 4'b1000) begin
                        cnt1 <= 4'b000;
                        clk_out_t <= 1'b1;
                        if(cnt2 == 3'b101)    //控制占空比
                            cnt2 <= 3'b000;
                        else
                            cnt2 <= cnt2 + 3'b001;
                    end
                    else if(cnt1 < 4'b0101) begin
                        cnt1 <= cnt1 + 4'b0001;
                        clk_out_t <= 1'b0;
                    end
                    else begin
                        cnt1 <= cnt1 + 4'b0001;
                        clk_out_t <= 1'b1;
                    end
                end
                1'b0: begin //8 分频模块处理电路
                    if (cnt1 == 4'b0111) begin //控制占空比
                        cnt1 <= 4'b000;
                        if(cnt2 == 3'b101)
                            cnt2 <= 3'b000;
                        else
                            cnt2 <= cnt2 + 3'b001;
                        clk_out_t <= 1'b1;
                    end
                    else if(cnt1 < 4'b0100) begin
                        cnt1 <= cnt1 + 4'b0001;
                        clk_out_t <= 1'b0;
                    end
                    else begin
                        cnt1 <= cnt1 + 4'b0001;
                        clk_out_t <= 1'b1;
                    end
                end
            endcase
        end
    end
```

```
always @(posedge clk_in) begin
    if(!rst_n) begin
        fs_ctl <= 1'b0;
    end
    else begin
        case(cnt2)   //完成 8、9 分频的选择，达到 8.66 分频的目的
            3'b000,3'b011: fs_ctl <= 1'b1;
            default: fs_ctl <= 1'b0;
        endcase
    end
end

endmodule
```

上述程序的仿真结果如图 11-4 所示，从中可以看出，在 6 个分频间隔中有 2 个 9 分频、4 个 8 分频，整体上实现了 8.66 分频。

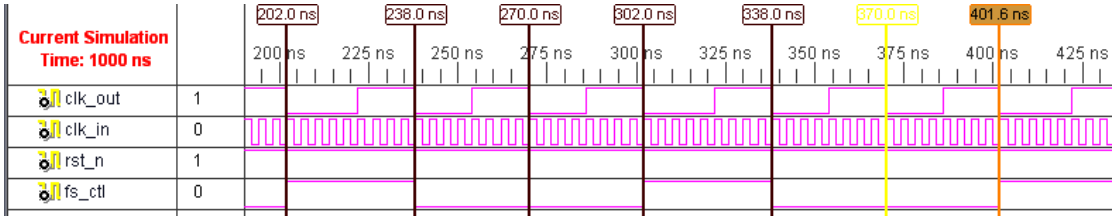


图 11-4　分频交错法结构示意图

2．累加器分频法

累加器分频法的结构如图 11-5 所示，通过调整步长 STEP 的值来实现不同的分频比。

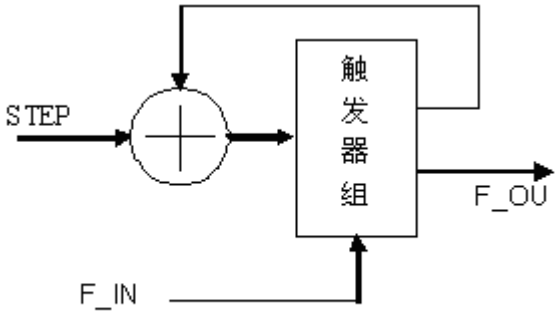


图 11-5　累加器分频法结构示意图

假设累加器位数为 8，则累加器的模值 N 为 2^8=256。若 STEP=27，则分频比 K 为：

$$K = \frac{N}{\text{STEP}} = \frac{256}{27} = 9.84148148\cdots\cdots$$

类似地，通过改变模值 N 和步长 STEP 就可以以任意精度逼近某个预定的分频比。

例 11-4：利用累加器法完成 8.66 分频的电路。

首先，假设模值为 65536，计算出相应的步长：

N=[65536/8.66] = [7567.667]=7568，其中“[]”为四舍五入操作。

其次，得到的程序如下：

```
module clk_div_8p62(
```

```
    clk_in, rst_n, clk_out
);

  input       clk_in;
  input       rst_n;
  output      clk_out;

  reg   [15:0] cnt;

  always @(posedge clk_in) begin
      if(!rst_n) begin
          cnt <= 16'h0000;
        end
        else begin     //完成计数器的相加
          cnt <= cnt + 7568;
        end
  end

assign    clk_out = cnt[15];

endmodule
```

上述程序在 ISE Somulator 中的仿真结果如图 11-6 所示。从中可以看出，clk_out 的每个周期包括 8 个或 9 个 clk_in 信号的周期，整体而言达到 8.66 分频的目的，

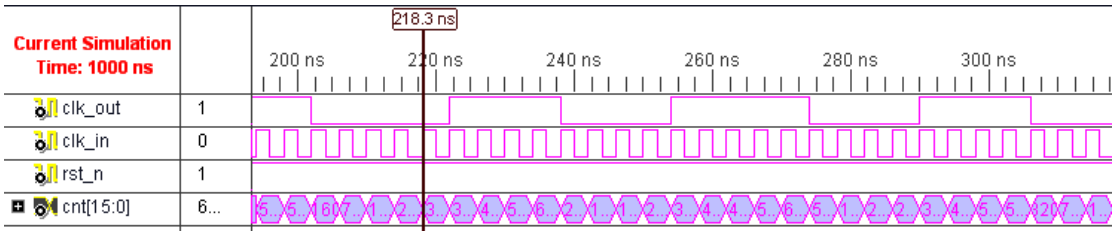


图 11-6　分频器的仿真结果

当然，由分频比交错法和累加法分频法得到的分频时钟，其占空比基本达不到 50%、相位不可控，且每个分频时钟都不是精确的非整数分频，只是从一定时间的统计意义上到达了非整数分频。要想在 CPLD/FPGA 做到真正意义上的非整数分频，且相位和占空比可控，必须利用其内部的时钟处理硬核模块（Xilinx 公司的 DCM 模块），其使用方法将在 12.2 节中进行介绍。

11.1.3　同步整形电路

在 CPLD/FPGA 同步电路具备最稳定的工作状态和工作性能，因此经常需要将外部输入的异步信号进行同步处理（与系统时钟同步）和整形（将输入信号由不规则波形提取为具备一个时钟周期长的脉冲信号）。图 11-7 给出了将异步信号同步到时钟上升沿的示意图，同样也可以将信号同步到时钟的下降沿。

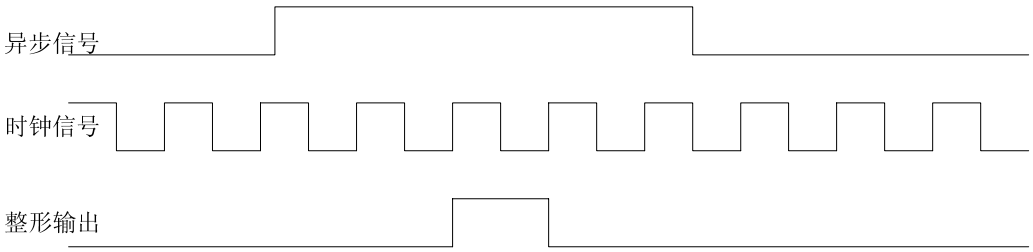


图 11-7　异步信号的同步采样示意图

同步整形的基本方法就是通过时钟对异步信号连续采样得到同步信号，然后由前后两次的同步采样进行逻辑组合得到整形输出。下面给出一个利用上升沿完成信号同步整形的设计。虽然同步整形的原理比较简单，但代码的实现却需要读者具备一定的时序电路设计能力。

例 11-5：利用 Verilog HDL 实现上升沿同步整形电路。

```
module syn_posedge(
    clk, rst_n, din, dout
    );
     input      clk;
     input      rst_n;
     input      din;
     output     dout;

     reg         dtmp1, dtmp2;

     always @(posedge clk) begin
       if(!rst_n)begin
            dtmp1 <= 0;
            dtmp2 <= 0;
        end
        else begin        //对信号进行两次采样
            dtmp1 <= din;
            dtmp2 <= dtmp1;
        end
    end

     assign   dout = dtmp1 && (~dtmp2); //通过两次信号的逻辑运算，得到脉冲信号

endmodule
```

程序在 ISE 中综合后的 RTL 结构图如图 11-8 所示。可以看到其结构非常简单，就是将输入信号经过两个 D 触发器进行二次采样，然后将一次采样结果取反和二次采样结果相与得到输出信号。

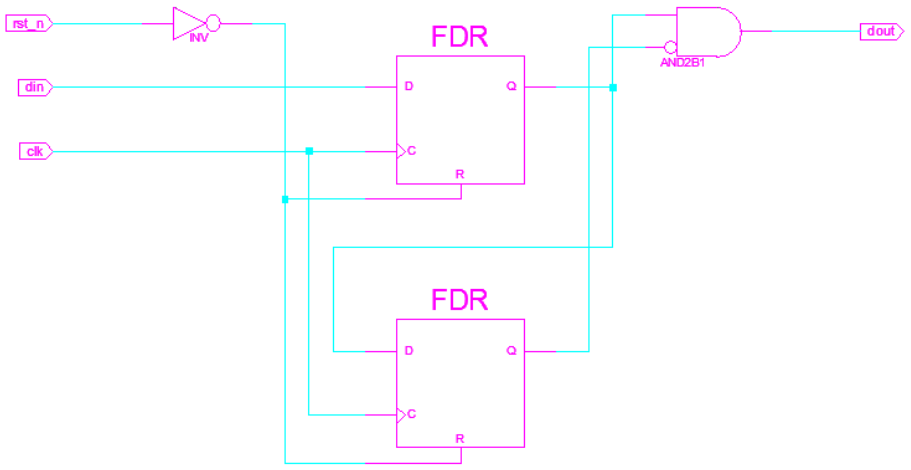


图 11-8　上升沿同步整形电路的 RTL 级结构示意图

下面给出上述处理的详细说明。首先，通过 D 触发器采样输入信号，将其数值宽度划分为一个一个时钟周期宽度的数值。其次，设定两次采样，则是结合后面的处理电路，将输出信号同步到一个时钟宽度。如果要将输入信号同步到 N 个时钟宽度，则需要 N+1 级 D 触发器级联，然后将第 1 级和第 N+1 级触发器的值送入后续处理逻辑。第三，对 D 触发采样值的处理逻辑，就是一个简单的判断上升沿的组合逻辑，其真值表如表 11-1 所列，可以直观得到，其逻辑为 dout = dtmp1 && (~dtmp2)。

表 11-1　　上升沿检测电路的真值表

采样值 dtmp1	采样值 dtmp2	对应脉冲形式	输出 dout
0	0	无变化	0
0	1	下降沿	0
1	0	上升沿	1
1	1	无变化	0

上升沿同步程序在 ISE Simulator 中的仿真结果如图 11-9 所示。可以看出：如果在时钟的上升沿 din="1"，则 x=1，y=0，dout=x and （not y）=1；如果 din="1"超过一个时钟宽度，则 x=1，y=1，dout=x and (not y）=0。即使 din 在时钟周期内出现抖动，也不会影响输出结果，还是被整形成一个时钟宽度。所以不管是长周期信号还是短周期信号，经过同步采样后，有效高电平宽度都等于时钟周期。

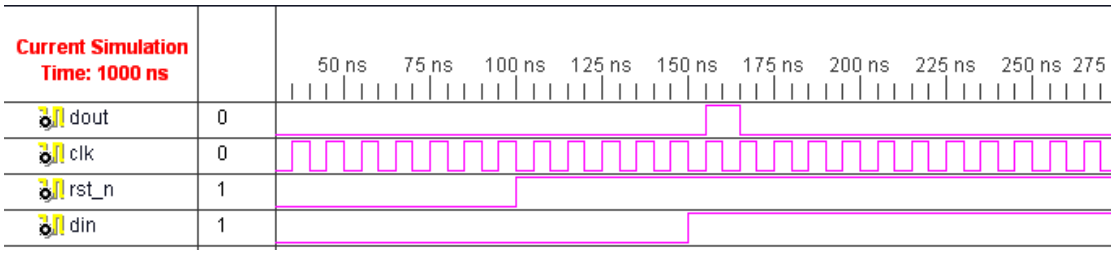


图 11-9　上升沿同步整形电路的仿真结果

11.2　乘加运算的 Verilog HDL 实现

11.2.1　加法器的 Verilog HDL 实现

1．加法器工作原理

加法器按照操作原理可以分为串行加法器和并行加法器两大类，下面分别对其进行介

绍。

（1）串行加法器

组合逻辑的加法器可以利用真值表，通过与门和非门简单地实现。假设 x_i 和 y_i 表示两个加数，S_i 表示和，C_{i-1} 表示来自低位的进位，C_i 表示向高位的进位。每个全加器都执行如下的逻辑表达式：

$$S_i = T_i \oplus C_i \qquad \text{其中 } T_i = x_i \oplus y_i \qquad (11\text{-}1)$$

$$C_i = T_i \cdot C_{i-1} + G_i \qquad \text{其中 } G_i = x_i \cdot y_i \qquad (11\text{-}2)$$

这样可以得到加法器的一种串行结构。因此，式(11-1)所示的加法器也被称为串行加法器。如图 11-10 给出了一个 4 位串行加法器的结构示意图。

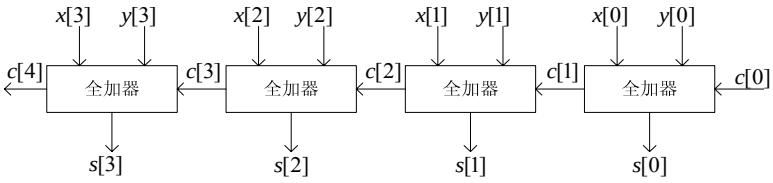


图 11-10　串行加法器的结构示意图

例 11-6：通过 Verilog HDL 语言实现一个 1 比特全加器。

```
module adder1_demo(
    a, b, ci, so, co
    );
      input    a, b, ci;
      output so, co;

      assign so=(~a&&~b&&ci)||(~a&&b&&~ci)||(a&&~b&&~ci)||(a&&b&&ci);
      assign co=(~a&&b&&ci)||(a&&~b&&ci)||(a&&b&&~ci)||(a&&b&&ci);

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 11-11 所示。

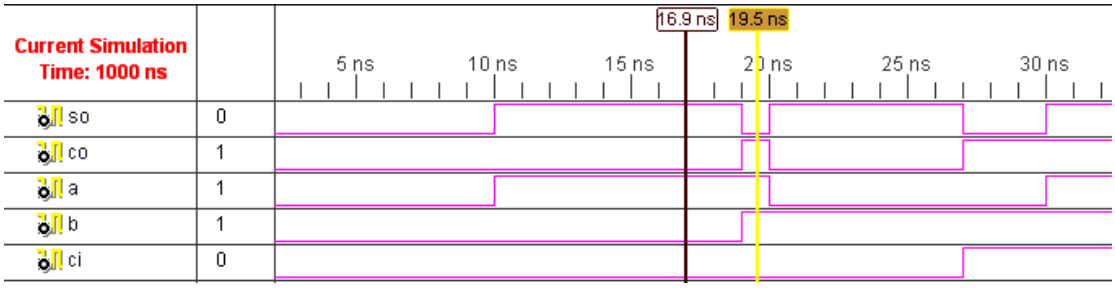


图 11-11　加法器仿真结果示意图

对于多位宽的串行加法器，只需要将多个 1 比特全加器按照图 11-10 级联起来即可，这里就不再赘述。

（2）超前进位加法器

在实时信号处理中，常常要用到多位数字量的加法运算，但串行加法器速度较慢，超前进位加法器则能满足要求，且结构并不复杂。现在普遍使用的并行加法器是超前进位加法器，只是在几个全加器的基础上增加了一个超前进位形成逻辑，以减少由于逐步进位信号的传递所造成的时延。图 11-12 给出了一个 4 位并行加法器的结构示意图。在 4 位并行加法器

的基础上，可以递推出 16 位、32 位和 64 位的快速并行加法器。

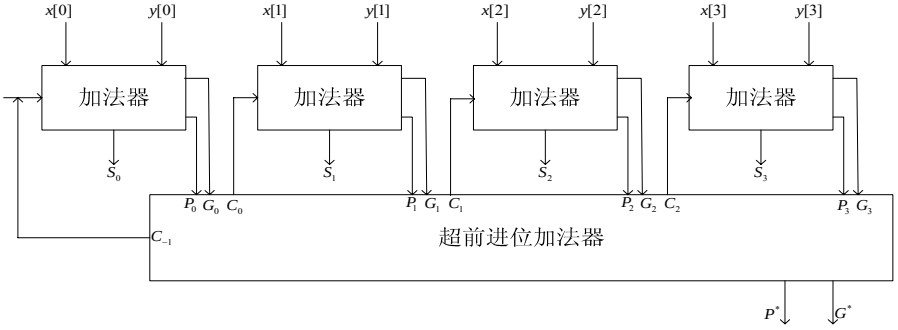


图 11-12　串行加法器的示意图

下面给出 4 比特超前进位链的推导。首先，对于 1 比特全加器，

$$S_i = x_i \oplus y_i \oplus C_i \tag{11-3}$$

$$C_i = x_i y_i + (x_i + y_i) C_{i-1} \tag{11-4}$$

从中可以看出，如果两个输入 x、y 都为 1，则进位输出肯定为 1；如果 x、y 有一个为 1，则进位输入等于下一级的进位输入。定义 P=x+y，G= xy，则 4 比特超前进位链的逻辑如下所列：

$C_0 = C_{in}$

$C_1 = C_0 + P_0 G_0 = C_0 + P_0 G_{in}$

$C_2 = C_1 + P_1 G_1 = C_1 + P_1 (G_0 + P_0 G_{in}) = G_1 + P_1 G_0 + P_1 P_0 G_{in}$

$C_3 = C_2 + P_2 G_2 = C_2 + P_2 (G_1 + P_1 G_1) = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 G_{in}$

$C_4 = C_3 + P_3 G_3 = C_3 + P_3 (G_2 + P_2 G_2) = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 G_{in}$

$C_{out} = C_4 + P_4 G_4$

可以看出，各个进位彼此独立产生，将进位级联传播的串行特性取消掉，因此通过超前进位链可以减少进位所产生的延迟。

例 11-7：通过 Verilog HDL 语言实现一个 4 比特的超前进位加法器。

```
module adder2_demo(
    a, b, cin, so, co
    );
    input    [3:0] a;
    input    [3:0] b;
     input            cin;
     output [3:0] so;
     output           co;

     wire     [3:0] G, P;
     wire     [3:0] C;

     assign G[0]    = a[0] & b[0];
     assign P[0]    = a[0] | b[0];
     assign C[0]    = cin;
     assign so[0] = G[0] ^ P[0] ^ C[0];
```

```
    assign G[1]   = a[1] & b[1];
    assign P[1]   = a[1] | b[1];
    assign C[1]   = G[0] |(P[0] & cin);
    assign so[1] = G[1] ^ P[1] ^ C[1];

    assign G[2]   = a[2] & b[2];
    assign P[2]   = a[2] | b[2];
    assign C[2]   = G[1] |(P[1] & C[1]);
    assign so[2] = G[2] ^ P[2] ^ C[2];

    assign G[3]   = a[3] & b[3];
    assign P[3]   = a[3] | b[3];
    assign C[3]   = G[2] |(P[2] & C[2]);
    assign so[3] = G[3] ^ P[3] ^ C[3];

  assign co    = G[3] |(P[3] & C[3]);

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 11-13 所示，可以看出正确实现了加法器的功能。

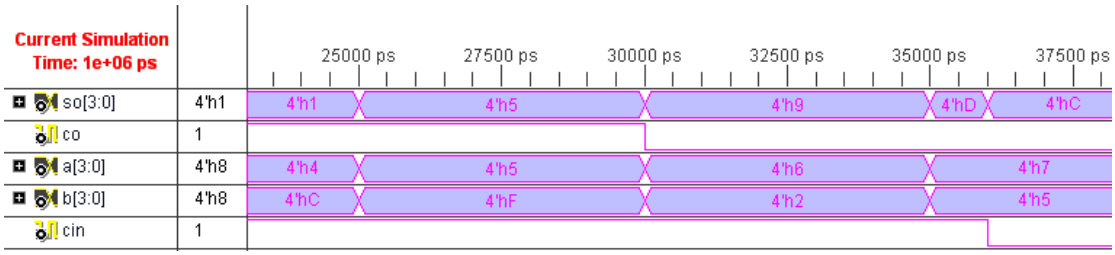


图 11-13　超前进位加法器仿真结果

2．加法运算符“+”的应用实例

由真值表推导设计电路的输出表达式后，再通过 Verilog HDL 语言建模是设计全加器的一种方法，但这并不是最好的方式。最简便的方法是利用运算符“+”，其本质上是一种并行加法器。下面给出一个应用实例。

例 11-8：一个加法运算符“+”的应用实例。

```
//-----------用加法运算符实现 8 位加法----------------------
module add_4 (x, y, sum, C);
    input [3:0] x, y;
    output [3:0] sum;
    output C;
    assign {C, Sum} = x + y;
endmodule
```

读者需要注意的是，上述程序中的“+”运算符实质上是并行加法运算符，其位宽由操作数决定，因此应该保证“+”号两边的数位宽一致。

11.2.2 乘法器的 Verilog HDL 实现

1．串行乘法器

两个 N 位二进制数 x 、y 的乘积用简单的方法计算就是利用移位操作来实现，用公式可以表示为：

$$P = xy = \sum_{k=0}^{N-1} X_k 2^k y \tag{11-5}$$

其中，这样输入量随着 k 的位置连续地变化，然后累加 $2^k y$ 。

例 11-9：用 Verilog HDL 语言实现一个 8 位串行乘法器

```
module ade (clk, x, y, p);
    input     clk;
    input     [7:0]   x, y;
    output    [15:0] p;
    reg       [15:0] p;

    parameter s0=0, s1=1, s2=2;
    reg [2:0] count = 0;
    reg [1:0] state = 0;
    reg   [15:0] p1, t;            // 比特位加倍
    reg   [7:0] y_reg;

    always @(posedge clk) begin
        case (state)
          s0 : begin               // 初始化
            y_reg <= y;
            state <= s1;
            count = 0;
            p1 <= 0;
            t <= {{8{x[7]}},x};
          end
          s1 : begin                // 处理步骤
            if (count == 7)     //判断是否处理结束
               state <= s2;
            else begin
               if (y_reg[0] == 1)
                  p1 <= p1 + t;
                  y_reg <= y_reg >> 1;     //移位
                  t <= t << 1;
                  count <= count + 1;
                  state <= s1;
            end
          end
          s2 : begin
```

```
            p <= p1;
            state <= s0;
          end
        endcase
    end
endmodule
```

图 11-14 给出了上述程序在 ISE Simulator 中的仿真结果。乘法功能是正确的，但计算一次乘法需要 8 个周期。因此可以看出串行乘法器速度比较慢、时延大，但这种乘法器的优点是所占用的资源是所有类型乘法器中最少的，在低速的信号处理中有着广泛的应用。

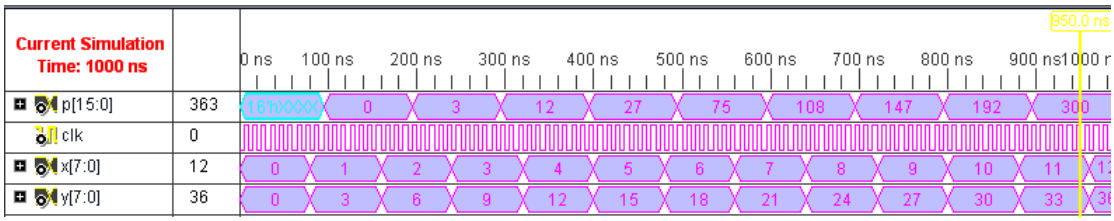


图 11-14　串行乘法器的局部仿真结果示意图

2．流水线乘法器

一般的快速乘法器通常采用逐位并行的迭代阵列结构，将每个操作数的 N 位都并行地提交给乘法器。但是一般对于 FPGA 来讲，进位的速度快于加法的速度，这种阵列结构并不是最优的。所以可以采用多级流水线的形式，将相邻的两个部分乘积结果再加到最终的输出乘积上，即排成一个二叉树形式的结构，这样对于 N 位乘法器的需要 $\log_2(N)$级来实现。一个 8 位乘法器，如图 11-15 所示。

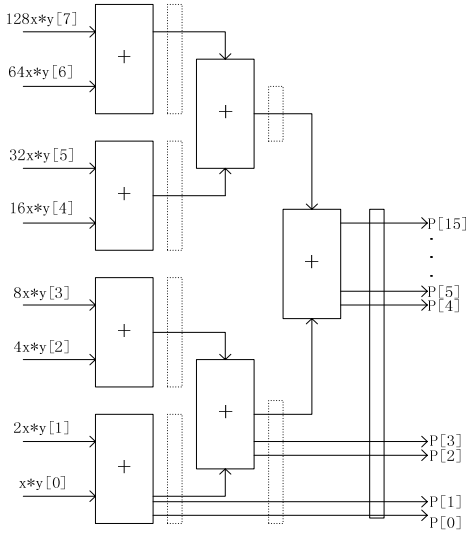


图 11-15　流水线乘法器结构图

例 11-10：用 Verilog HDL 实现一个 4 位的流水线乘法器

```
module mul_addtree(mul_a, mul_b, mul_out, clk, rst_n);
    parameter MUL_WIDTH = 4;
    parameter MUL_RESULT = 8;
```

```
    input [MUL_WIDTH-1 : 0] mul_a;
    input [MUL_WIDTH-1 : 0] mul_b;
    input clk;
    input rst_n;
    output [MUL_RESULT-1 : 0] mul_out;
    reg [MUL_RESULT-1 : 0] mul_out;
    reg [MUL_RESULT-1 : 0] stored0;
    reg [MUL_RESULT-1 : 0] stored1;
    reg [MUL_RESULT-1 : 0] stored2;
    reg [MUL_RESULT-1 : 0] stored3;
    reg [MUL_RESULT-1 : 0] add01;
    reg [MUL_RESULT-1 : 0] add23;

    always @ (posedge clk or negedge rst_n)
    begin
        if(!rst_n)
            begin   //初始化寄存器变量
                mul_out <= 8'b0000_0000;
                stored0 <= 8'b0000_0000;
                stored1 <= 8'b0000_0000;
                stored2 <= 8'b0000_0000;
                stored3 <= 8'b0000_0000;
                add01 <= 8'b0000_0000;
                add23 <= 8'b0000_0000;
            end
        else
            begin   //实现移位相加
                stored3 <= mul_b[3]?{1'b0,mul_a,3'b0}: 8'b0;
                stored2 <= mul_b[2]?{2'b0,mul_a,2'b0}: 8'b0;
                stored1 <= mul_b[1]?{3'b0,mul_a,1'b0}: 8'b0;
                stored0 <= mul_b[0]?{4'b0,mul_a}: 8'b0;
                add01 <= stored1 + stored0;
                add23 <= stored3 + stored2;
                mul_out <= add01 + add23;
            end
    end

endmodule
```

程序在 ISE 综合后，其 RTL 级结构图如图 11-16 所示。

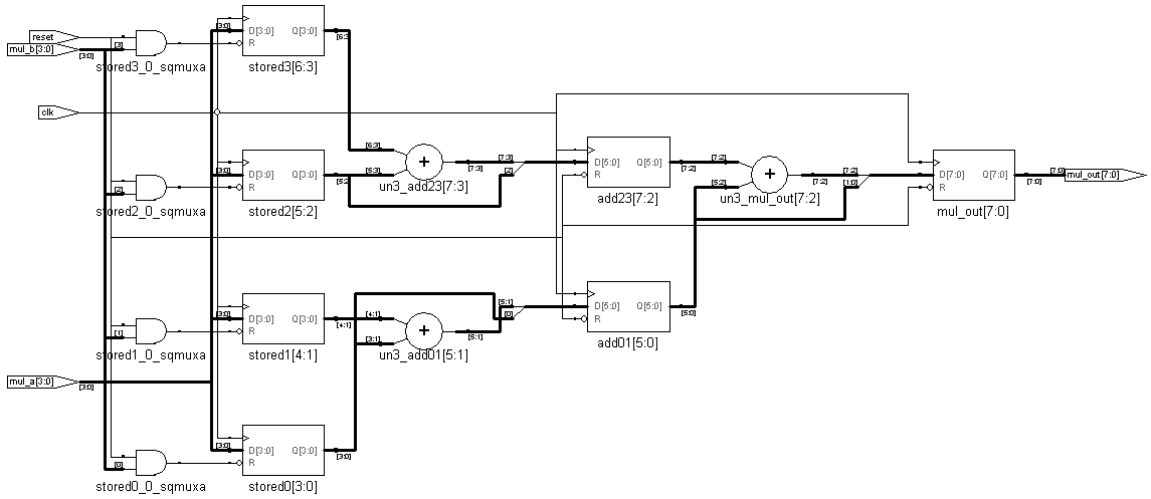


图 11-16　流水线加法树乘法器的 RTL 结构示意图

图 11-17 给出了流水线乘法器模块在 ISE Simulator 中的仿真结果，可以看出结果是正确的，虽然输出比输入延迟了 3 个时钟周期，但输入数据的速率和时钟频率大小是一致的，有效地提高了吞吐量，这就是流水线的主要功能。

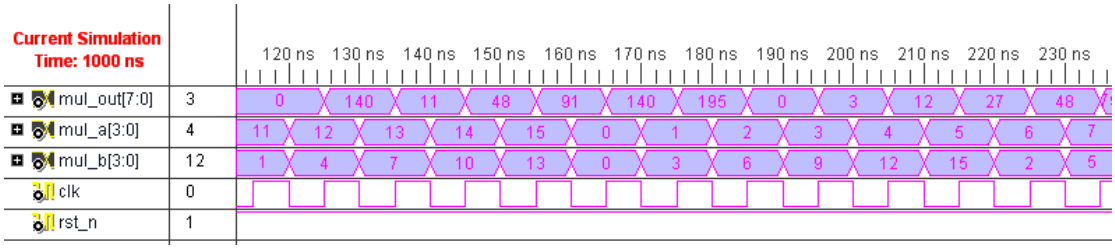


图 11-17　流水线乘法器的局部仿真结果示意图

从仿真结果可以看出，上述流水线乘法器比串行加法器的速度快很多，在非高速的信号处理中有着广泛的应用。至于高速信号的乘法一般需要利用 FPGA 芯片中内嵌的硬核 DSP 单元来实现。

3．复数乘法器

一般来讲，需要 4 个实数乘法器才能完成组合成一个复数乘法器，但由于在 FPGA 中乘法器占用的资源是比较多的，应当设法尽可能减少这一资源。为了达到这一目的，下面给出只需要 3 个实数乘法器就可以完成复数乘法的方法。

复数 p， a 和 b，且 $p=ab$，

$$p = p_r + p_i = ab = (a_r + a_i)(b_r + b_i) \tag{11-6}$$

其中：　$p_r = a_r b_r - a_i b_i = a_r(b_r + b_i) - (a_r + a_i)b_i$

$$p_i = a_r b_i + a_i b_r = a_r(b_r + b_i) + (a_i - a_r)br$$

如式(11-6)所示，通过增加一个加法，节省了一个乘法器，其原理框图如 11-18 所示。

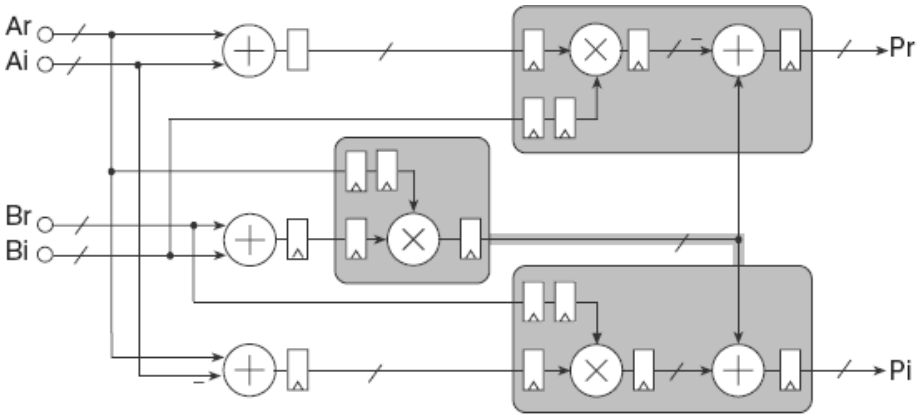


图 11-18　只需要 3 个乘法的复数乘法示意图

例 11-11：用 Verilog 实现节省乘法器的 16 位复数乘法

```
module cmultip(clk, ar, ai, qr, br, bi, qi);
   input clk;
   input [15 : 0] ar;
   input [15 : 0] br;
   output [31 : 0] qr;
   input [15 : 0] ai;
   input [15 : 0] bi;
   output [31 : 0] qi;

   wire [15 : 0] br_add_bi;
   wire [15 : 0] ar_add_ai;
   wire [15 : 0] ai_sub_ar;

   reg [31 : 0] arbrbiout;
   reg [31 : 0] araibiout;
   reg [31 : 0] aiarbrout;

   //完成加法预处理
   assign      br_add_bi = br + bi;
   assign      ar_add_ai = ar + ai;
   assign      ai_sub_br = ai - ar;

   //调用乘法器模块
   always @(posedge clk) begin
         arbrbiout <= ar * br_add_bi;
         araibiout <= bi * ar_add_ai;
         aiarbrout <= br * ai_sub_ar;
   end

   // 完成加法后处理
   assign qr = arbrbiout - araibiout;
```

```
    assign qi = arbrbiout + aiarbrout;

endmodule
```

上述程序经过综合后 RTL 级结构如图 11-19 所示，可以看出其和图 11-18 的结构是一致的，只用了 3 个乘法器。其中，“*”运算符可以用任何一种乘法器来替代，包括 Xilinx 公司的实数乘法器 IP Core 生成。

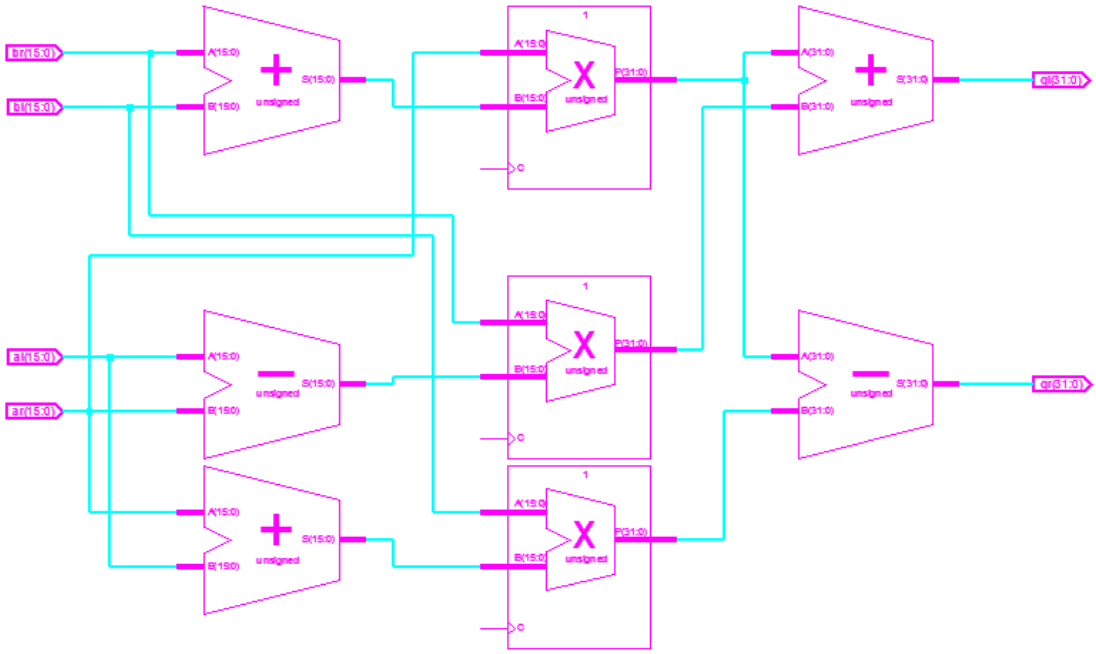


图 11-19　复数乘法器的 RTL 结构图

图 11-20 给出了复数乘法器模块在 ISE Simulator 中的仿真结果，可以看出该模块的功能是正确的

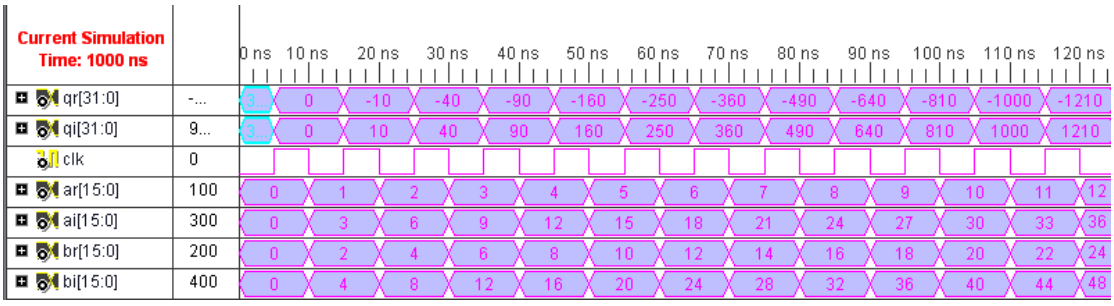


图 11-20　复数乘法器的局部功能仿真结果

11.2.3　数据的截位与扩位

1．扩位操作

在定点计算中，经过加法和乘法运算后，输出结果的位宽会增加。但如果继续使用和输入操作数同等位宽的数来表示结果，就会丢失有用的比特信息，造成输出结果错误。例如，在有限字长的情况下，若两个 M 位的数相加，其结果最高可能为 $M+1$ 位；若两个 M 位的数相乘，其结果最多可为 $2M$ 位。在下面的例 11-12 中给出扩位现象的例子。

例 11-12：展示 4 比特加法运算中的扩位现象。

下面从有符号数和无符号数两类情况来说明：

（1）无符号数的加法：4'b1111 + 4'b1111 = 5'11110，但是由于加数是 4 位，在 Verilog 语言中只保留低 4 位，就会得到 4'b1111 + 4'b1111 = 4'1110 的结果，这样就会造成计算错误。

（2）有符号数的加法：4'b0101 和 4'b0111 分别对应着+5 和+7，二者相加后本应为+12，即 5'b01100。但由于位宽限制，如不扩位，只能保留低 4 位，即 4'b1100，对应着-4，造成严重的计算错误。类似的错误还会造成负数相加变成正数。

从上面可以看出，对于数学运算需要考虑位宽效率，否则会造成严重的计算错误。

2．截位操作

在有限字长的情况下，若两个 M 位的数相加，其结果就是 $M+1$ 位；若两个 M 位的数相乘，其结果就是 $2M$ 位。但在实际的操作过程中，考虑到资源的问题，不能任由相加、相乘操作来增加操作数的位宽，必须进行截断。例如，两个 16 位数相乘后，其结果为 32 位，如再和一个 16 位数相乘，结果就变为 48 位，这样下去，用不了几个乘法操作就会使操作数的位宽剧增，所占用的硬件资源也会很多。因此，需要将乘积结果进行截位，寄存在 M 位的寄存器中。

扩位和截取都是按照定点仿真的结果来定的，下面依次给出加法扩位操作、加保护截取操作和移位操作的书写规范。

（1）加法实现规范，扩展符号位后相加。

```
reg[12:0] Adder_Out;
reg[11:0] Adder_In1,Adder_In2;

Adder_Out <= {Adder_In1[11],Adder_In1} + {Adder_In2[11],Adder_In2};
```

（2）对于截取乘法的结果，需要加溢出保护的截取规范。例如要截取 12 比特输出的第 6 位到第 2 位，其实现代码为：

```
if((addRakeOut[11:6] == 0) || (addRakeOut[11:6] == 63))
    tmptraffic <= addRakeOut[6:2];
else
    tmptraffic <= (addRakeOut[11] == 1) ? 16 : 15;
```

或者：

```
if((addRakeOut[11:6] == 6'b000000) || (addRakeOut[11:6] == 6'b111111))
    tmptraffic <= addRakeOut[6:2];
else
    tmptraffic <= (addRakeOut[11] == 1) ? 5'b10000 : 5'b01111;
```

（3）移位操作规范

移位操作的截位和加法器的截位规则类似，下面给出 16 比特数据的左、右 1 比特移位示例。

```
reg[15:0] Data;

//左移一位
```

```
if((Data[15:14] == 2'b00) || (Data[15:14] == 2'b11))
    Data <= {Data[14:0],1'b0};
else
    Data <= (Data[15]) ? 16'b1000_0000_0000_0000 : 16'b0111_1111_1111_1111;
//右移一位
Data <= {Data[15],Data[15:1]};
```

11.3 数码管接口电路的 Verilog HDL 实现

11.3.1 数码管简介

数码管是电路中常见的显示元件，按段数分为七段数码管和八段数码管，八段数码管比七段数码管多一个发光二极管单元（多一个小数点显示）；按照显示“8”的个数，可分为 1 位、2 位、4 位等数码管，如图 11-21 所示的为一个 8 段 4 位数码管；按发光二极管单元连接方式分为共阳极数码管和共阴极数码管。共阳数码管是指将所有发光二极管的阳极接到一起形成公共阳极的数码管。共阳数码管在应用时应将公共极接到+5V 或+3.3V，当某一字段发光二极管的阴极为低电平时，相应字段就点亮。当某一字段的阴极为高电平时，相应字段就不亮。共阴数码管是指将所有发光二极管的阴极接到一起形成公共阴极（COM）的数码管。共阴数码管在应用时应将公共极 COM 接到地线 GND 上，当某一字段发光二极管的阳极为高电平时，相应字段就点亮。当某一字段的阳极为低电平时，相应字段就不亮。

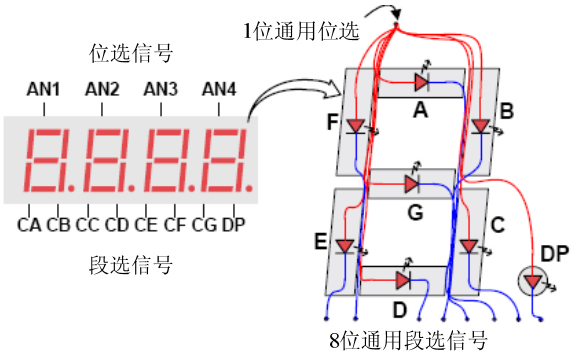


图 11-21　8 段 4 位数码管

2．数码管连接方式

任何一个 7 段码管都有 128 种显示模式，而其中的数字 0-9 是最为有用也是最常见的。通过控制共阳极（共阴极）数码管的阴极（阳极），可以显示数字 0-9，图 11-22 给出共阳极和共阴极数码管各自的连接关系。对于多位数码管而言，实际中为了简化电路，常常需要将所有共阴极数码管的阳极接到一起，所有共阳极数码管的阴极接到一起，用多个独立的位选和 7 个（或 8 个）公共段选控制所有的数码管。

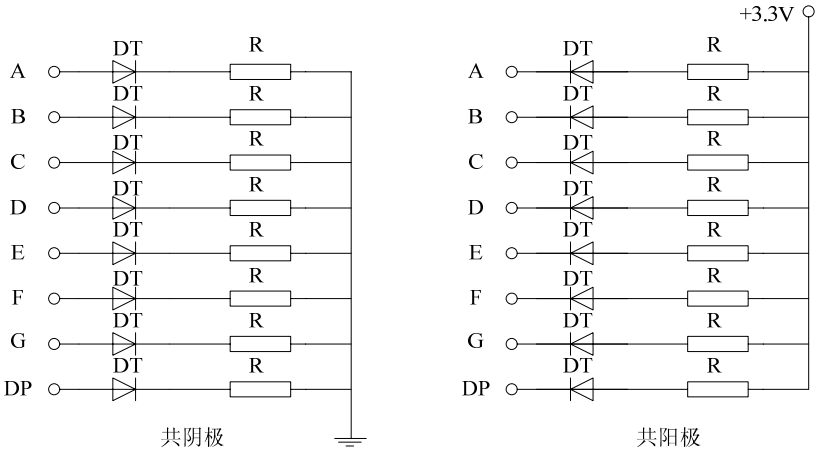


图 11-22　共阴极、共阳极数码管的连接关系示意图

由于所有数码管共用段选，为了独立显示每位数码管，只能用段选来区分不同的数码管。具体来说就是每次只将某一位数码管的位选置为有效，其它数码管的位选都无效。此时的段选决定了该位数码的显示。然后在下一个时刻，置下一位数码管的位选有效，其它数码管的位选都无效。依次类推，循环往复。如果总共有 4 位数码管，则用于显示的控制时序如图 11-23 所示。

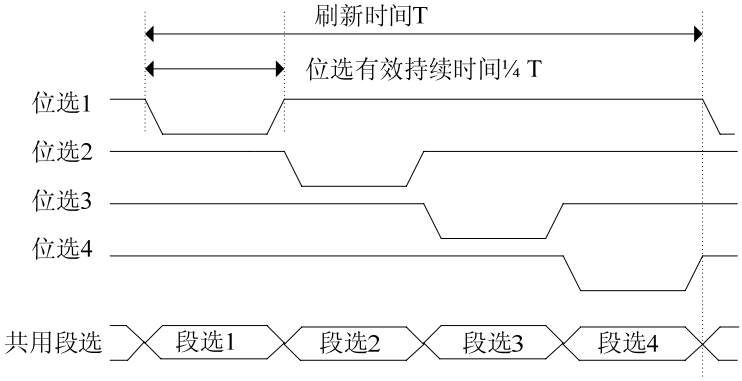


图 11-23　四位数码管控制时序

位选 1 为低时（其它位选都为高），第一位数码管被选中，此时的共用段选用于第一位数码管的显示；位选 2 为低时（其它位选都为高），第二位数码管被选中，此时的共用段选用于第二位数码管的显示，三、四位数码管的显示依次类推。在一个刷新周期 T 内，每位数码管都有 1/4T 周期的时间被刷新。为了保证所有 4 位数码管的显示不闪烁，一般刷新频率要大于 45Hz。在一个周期中，虽然每位数码管会有 3/4T 的时间不被点亮，但位选刷新的速度较快，同时由于数码管自身的余辉特性，每位数码管在变暗之前就又会被重新刷新，因此人眼无法感觉到数码管变暗。如果刷新的频率小于一定值（如 45Hz），则人眼就会感觉到数码管的闪烁。一般刷新频率在 60Hz 到 1KHz 之间时，多位数码管显示得比较理想。

11.3.2　数码管显示电路的 Verilog HDL 实现

例 11-13：假设位选信号为低有效，当位选有效时，段选为 0 时对应的二极管段被点亮。则用 FPGA 控制 4 位 8 段数码管分别显示数字 1、2、3、4 的程序如下：

```
module LED_Display(clk,seg,dp,an);
    input clk;                  //输入时钟
```

```
output[6:0] seg;            //7 个公共段选信号，从低到高对应七段数码管的 ABCDEFG
output dp;                  //小数点段选信号 DP
output[3:0] an;     //4 位数码管的位选信号

reg [15:0] count_for_clk = 0; //分频计数器,65536 分频
reg [3:0]    an_reg = 0;
reg [6:0]    seg_reg = 0;

assign seg=seg_reg;         //7 个段选赋值
assign dp=1;                //小数点段选赋值
assign an=an_reg;           //4 个位选赋值

parameter           //七段数码管显示数字 0-9 的段选值
      zero = 7'b100_0000,
      one    = 7'b111_1001,
      two    = 7'b010_0100,
      three= 7'b011_0000,
      four = 7'b001_1001,
      five = 7'b001_0010,
      six    = 7'b000_0010,
      seven= 7'b111_1000,
      eight= 7'b000_0000,
      nine = 7'b001_0000;

//分频计数器
always@(posedge clk) begin
      count_for_clk<=count_for_clk+1;
end

//段选寄存器赋值，4 位数码管分时复用
always@(posedge clk) begin
      case(count_for_clk[15:14])
            0:    seg_reg<=one;       //数码管 1 段选
            1:      seg_reg<=two;         //数码管 2 段选
            2:    seg_reg<=three;     //数码管 3 段选
            3:    seg_reg<=four;//数码管 4 段选
      endcase
end

//位选寄存器赋值，每次只选通一位数码管
always@(posedge clk) begin
      case(count_for_clk[15:14])
            0:     an_reg<=4'b0111;   //选通数码管 1
            1:     an_reg<=4'b1011;   //选通数码管 2
```

```
            2:    an_reg<=4'b1101;  //选通数码管 3
            3:    an_reg<=4'b1110;  //选通数码管 4
        endcase
    end

endmodule
```

程序在 ISE 中综合后，其 RTL 级结如图 11-24 所示。

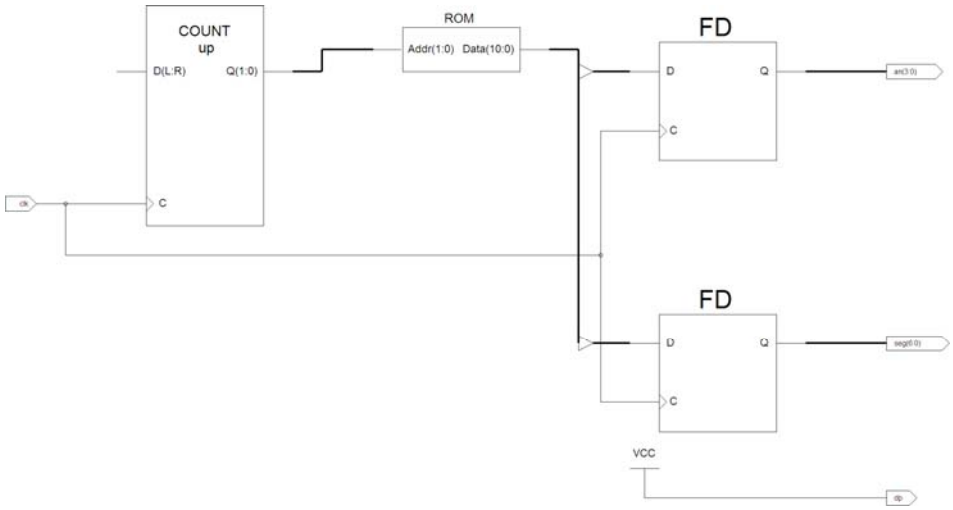


图 11-24　LED 的 RTL 结构图

上述程序在 ISE Simulator 中的仿真结果如图 11-25 所示：

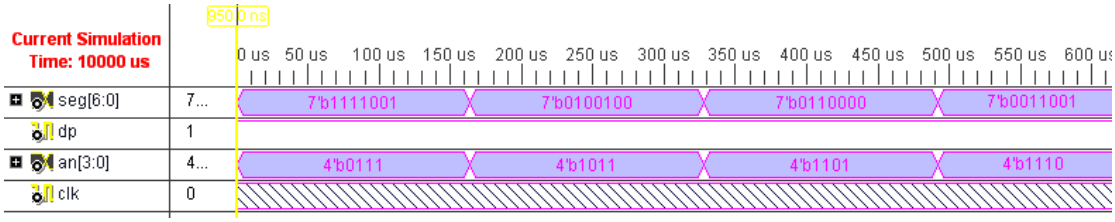


图 11-25　LED 显示仿真结果

由图 11-25 可知，小数点段选信号一直为高，对应的段熄灭。计数器 count_for_clk[15:14]控制位选信号 an 和段选信号 seg。在 count_for_clk[15:14]由 0-3 变化的过程中，位选信号 an 依次为 0111、1011、1101、1110，分别选中 1-4 位数码管；相应的段选信号 seg 分别对应数字 1、2、3、4。count_for_clk[15:14]大约 5ms 循环一次，因此刷新的频率为 200Hz。

11.4 按键接口电路的 Verilog HDL 实现

按键在数字系统中最简单的交互设备，不论是单片机、DSP、ARM 或 FPGA 的开发板，按键都是不可缺少的，如常用的系统复位按键。本节主要介绍按键扫描和防抖电路的 VHDL 实现。

11.4.1　按键扫描电路的 Verilog HDL 实现

1．按键接口的连接方式

按键接口分为独立式和矩阵式两大类。独立式键盘的每个按键都有一个信号线与

CPLD/FPGA 相连，所有按键有一个公共地或公共正端，每个键相互独立互不影响，如图 11-26 所示。当按下键 1 时，无论其它键是否按下，键 1 的信号线就由 1 变 0；当松开键 1 时，无论其它键是否按下，键 1 的信号线就由 0 变 1。矩阵式键盘的按键触点接于由行、列母线构成的矩阵电路的交叉处，每当一个按键按下时通过该键将相应的行、列母线连通，其连接原理如图 11-27 所示。

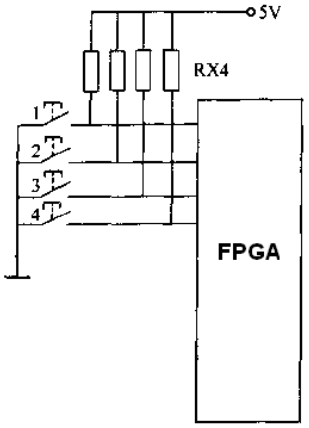


图 11-26　独立式按键原理图

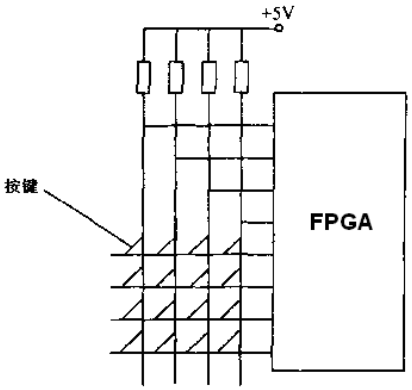


图 11-27　矩阵式按键原理图

从中可以看出，矩阵式按键接口操作要比独立式按键接口复杂，但可以节省大量 I/O 接口。当矩阵按键有 m 行 n 列时，则可连接（mn）个按键，却只占用（m+n）个 I/O 接口，特别适合应用于多按键场景。

2．按键扫描电路

对于独立式按键，不用特殊的扫描电路，直接判断 I/O 管脚的输入电平即可。按键扫描电路主要针对矩阵式连接。CPLD/FPGA 系统的矩阵式按键接法有多种，但本质上都是利用行、列扫描定位的原理。本节主要介绍图 11-28 所示的典型示例，其他形式都可从中变化得到。

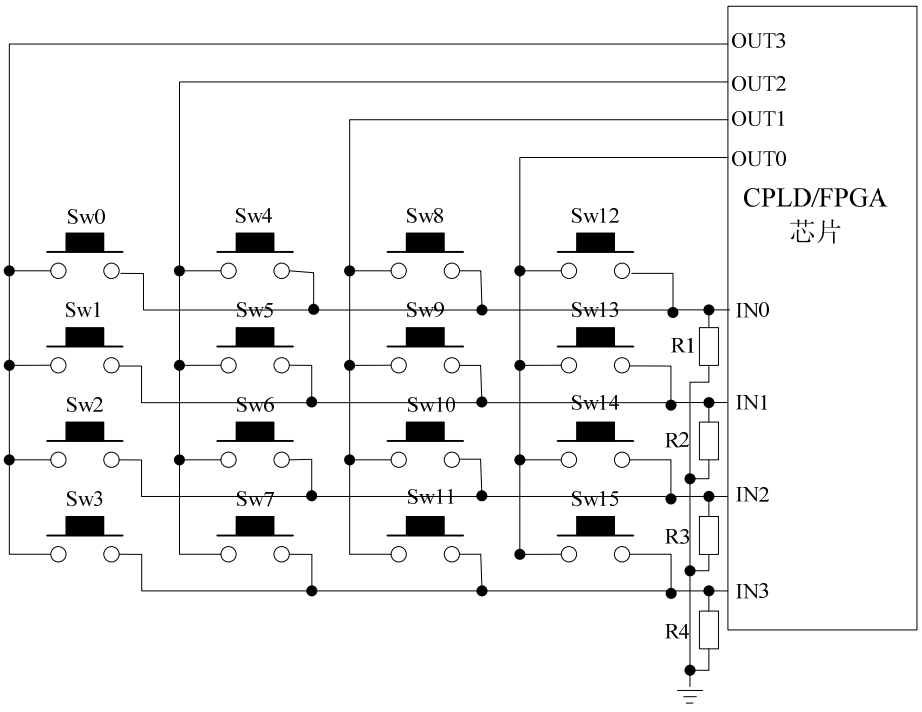


图 11-28　典型的按键矩阵连接电路

（1）按键扫描原理

在图 11-27 所示的连接方式中，每个按键都连接着一个行线（OUT）和一个列线（IN），OUT 信号为 CPLD/FPGA 芯片的输出信号，IN 信号为其输入信号。下面以 Sw0 为例进行说明。当 OUT3~OUT0 输出为“1000”时，如果只有按键 Sw0 被按下，则 IN0 为 1，IN2~IN4 由于下拉电阻的原因为低电平，因此 IN0~IN3 的输入管脚电平为“1000”；如果没有按键按下，则 IN0~IN3 的输出为“0000”。

矩阵键盘的扫描是通过输出不同的 OUT3~OUT0，然后检测输入信号 IN3~IN0，从而得到按键号。所以，可初步得到以下结论：当某一按键的 OUT 端信号为 1 时，如果此按键被按下，则其 IN 端信号也为 1，否则为 0。依次类推，可得到表 11-2 所列的按键与 IN[3:0]、OUT[3:0]信号电平的真值表。

表 11-2　　扫描信号电平和按键号的真值表

OUT3~OUT0（输出信号）	IN3~IN0（输出信号）	按键号
1000	0001	Sw0
1000	0010	Sw1
1000	0100	Sw2
1000	1000	Sw3
0100	0001	Sw4
0100	0010	Sw5
0100	0100	Sw6
0100	1000	Sw7
0010	0001	Sw8
0010	0010	Sw9
0010	0100	Sw10
0010	1000	Sw11
0001	0001	Sw12
0001	0010	Sw13
0001	0100	Sw14
0001	1000	Sw15

从表中可以看出，输出的扫描信号 OUT3~OUT0 在任意时刻只能有一个比特为高电平，若同时拉高多个信号，则无法准确判断按键号。例如，当 OUT3 和 OUT2 被同时赋为高电平，按键 SW0 和 SW4 被按下时，IN3~IN0 的值都为 0001，无法确定是 SW0 还是 SW4 倍按下。因此，在完成按键扫描电路时，需要保证输出的扫描信号 OUT 为 1000 的循环移位序列，0100、0010、0100 以及 1000。

（2）按键扫描接口的 Verilog HDL 实现

例 11-14：使用 Verilog HDL 实现图 11-28 所示的 4*4 矩阵键盘的接口扫描模块。

```
module button_scan(
    clk, in_s, out_s, num
    );
     //定义模块端口信息
     input            clk;
     input   [3:0]   in_s;
     output [3:0]   out_s;
     output [4:0]   num;
```

```
//定义输出信号类型以及局部变量
reg       [4:0]    num;
reg       [1:0]    cnt = 0;
reg       [1:0]    tmp = 0;
reg       [3:0]    out_st = 0;
wire      [7:0]    dsample;

//将扫描输出和输入信号级联，得到矩阵扫描结果
assign dsample = {out_st, in_s};
assign out_s = out_st;

//产生按键矩阵的列扫描信号
always @(posedge clk) begin
    cnt <= cnt + 1;
     case(cnt)
         2'b00: out_st <= 4'b1000;
         2'b01: out_st <= 4'b0100;
         2'b10: out_st <= 4'b0010;
         2'b11: out_st <= 4'b0001;
     endcase
end

//根据按键的列扫描信号和行输入信号判断按键是否被按下
always @(posedge clk) begin
  //如果无按键按下，定义 num=16 为无效态
  if (in_s == 4'b0000) begin
       if(tmp == 3) begin
            num <= 16;
            tmp <= 0;
        end
        else begin
            num <= num;
            tmp <= tmp + 1;
        end
    end
    else begin
       tmp <= 0;
        case(dsample)
            //第一列扫描结果
            8'b1000_0001 : num <= 0;
            8'b1000_0010 : num <= 1;
            8'b1000_0100 : num <= 2;
            8'b1000_1000 : num <= 3;
```

```
                //第二列扫描结果
                8'b0100_0001 : num <= 4;
                8'b0100_0010 : num <= 5;
                8'b0100_0100 : num <= 6;
                8'b0100_1000 : num <= 7;
                //第三列扫描结果
                8'b0010_0001 : num <= 8;
                8'b0010_0010 : num <= 9;
                8'b0010_0100 : num <= 10;
                8'b0010_1000 : num <= 11;
                //第四列扫描结果
                8'b0001_0001 : num <= 12;
                8'b0001_0010 : num <= 13;
                8'b0001_0100 : num <= 14;
                8'b0001_1000 : num <= 15;
            endcase
        end
    end

endmodule
```

上述程序在 ISE Simualtor 中的仿真结果如图 11-29 所示，表明该模块能正确完成外部输入的检测，通过矩阵模式有效设计所需的管脚数。

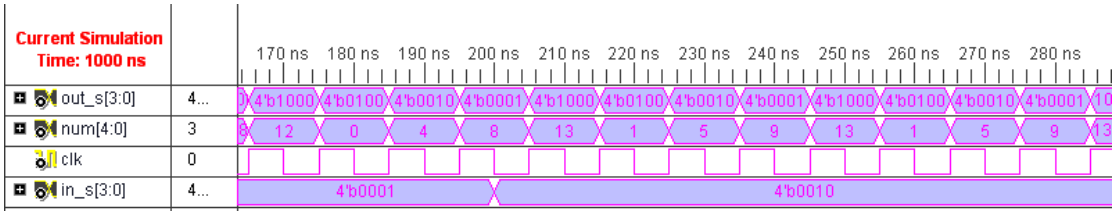


图 11-29　按键扫描模块仿真结果

11.4.2　按键防抖电路的 Verilog HDL 实现

1．键盘防抖简介

按键大多是机械式开关结构，一个按键开关在闭合时不会马上接通，在断开时也不会一下在断开，而是会产生一系列的抖动现象。如图 11-30 所示，释放按键后，按键信号稳定前出现了多个段脉冲，如果将这样的信号直接送到计数器之类的时序电路，结果将可能发生计数超过一次以上的误动作，从而误以为键盘按了多次。因此，必须加上抖动消除电路，除去短脉冲。

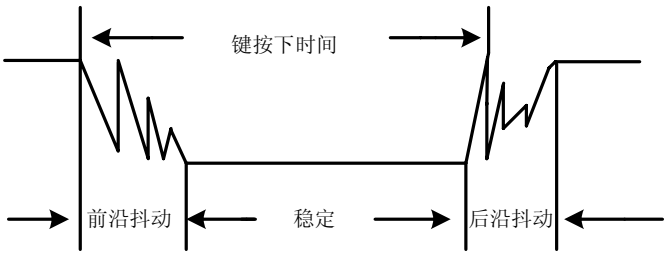


图 11-30　按键抖动现象示意图

常见的消抖方法可分为软件和硬件方式两大类。软件消抖通过计数器对键值进行判断，当某一键值保持一段时间不变时，才确认其为有效值。简单的硬件消抖主要通过电容的充放电来消除按键的毛刺。对于稳定性要求比较高的应用则需要借助专门的防抖芯片来实现消抖。不同开关的最长抖动时间也不同。抖动时间的长短和机械开关特性有关，一般为 5ms 到 10ms。但是，某些开关的抖动时间长达 20ms，甚至更长。一般按键按下去的时间由操作人员决定，通常为零点几秒至数秒。

2．键盘防抖的 Verilog HDL 实现

在实现时需要注意计数器门限的取值，如果门限值太大，即采样时间过长，将漏掉正确的信号；如果计数器的门限值过小，即采样时间太短，则会将毛刺误认为是正确信号。一般我们认为按键按下的时间在 100ms 以上，假设稳定时间的占空比为 50%，也就是说在图 11-28 所示场景中，前沿抖动的时间和稳定的时间各为 50ms。因此稳定时间大于 50ms 的信号为正确信号，而稳定时间小于 50ms 的为毛刺。因此计数器的门限值 n 为：50ms/系统采样时钟周期。在上例中，系统的采样时钟为 100MHz，所以计数器的门限值为 50ms*100MHz =24'h4C4B40。

例 11-15：使用 Verilog HDL 语言实现按键防抖电路。

```
module fangdou(Clk_100MHz,PB_UP,PB_Out,count_sel);
    input Clk_100MHz;                   //模块时钟 100MHz
    input PB_UP;                        //按键输入
    output PB_Out;                      //去抖后的按键输出
    output[1:0] count_sel;              //计数器输出

    reg [23:0] count_high = 0;          //按键输入高电平计数器
    reg [23:0] count_low= 0;            //按键输入低电平计数器
    reg PB_reg= 0;
    reg[1:0] count_sel_reg= 0;

    assign PB_Out=PB_reg;
    assign count_sel=count_sel_reg;

    //对输入进行采样，计数
    always @(posedge Clk_100MHz )
    if(PB_UP==1'b0)
        count_low<=count_low+1;//对低电平计数
    else
        count_low<=24'h000000;

    always @(posedge Clk_100MHz )
    if(PB_UP==1'b1)
        count_high<=count_high+1;//对高电平计数
    else
        count_high<=24'h000000;

    //防抖输出
    always@(posedge Clk_100MHz)
```

```
    if(count_high==24'h4C4B40) //判断高电平信号是否符合输出条件
        PB_reg<=1'b1; //如果符合条件，则防抖输出高电平
    else
        if(count_low==24'h4C4B40) //判断低电平信号是否符合输出条件
            PB_reg<=1'b0; //如果符合条件，则防抖输出低电平
        else
            PB_reg<=PB_reg;

    //使用去抖输出 PB_reg 控制 count_sel 计数
    always@(posedge PB_reg)
        count_sel_reg<=count_sel_reg+1;

endmodule
```

上述程序的仿真结果如图 11-31 所示，从中可以看出由于按键高电平、低电平的计数器数值都未达到预设的阀值，因此 PB_Out 的逻辑电平是不会发生变化的。

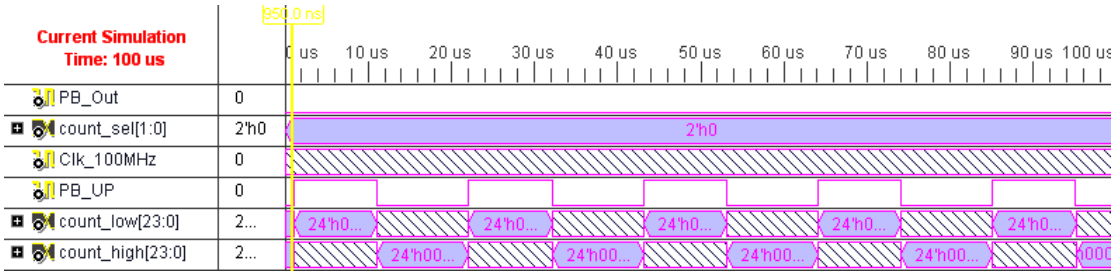


图 11-31　按键防抖电路的仿真结果

11.5　CRC 编码器的 Verilog HDL 实现

随着技术的不断进步，各种数据通信的应用越来越广泛。由于传输距离、现场状况、干扰等诸多因素的影响，设备之间的通信数据常会发生一些无法预测的错误。为了降低错误所带来的影响，常在通信时采用数据校验的办法，而循环冗余码校验是常用的重要校验方法之一。循环冗余校验码（CRC）特别适合于检测错误，这是由于它既具有很强的检测能力，同时实现起来也比较简单。

11.5.1　CRC 校验码的原理

循环冗余校验码的基本思想是利用线性编码理论，在发送端根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的 r 位监督码（即 CRC 码），并附在信息位后边，构成一个新的共（$n=k+r$）位的二进制码序列，最后发送出去。这种编码又叫 $(n,\ k)$ 码。对于一个给定的 $(n,\ k)$ 码，可以证明存在一个最高次幂为 r 的多项式 $G(x)$。根据 $G(x)$ 可以生成 k 位信息的校验码，而 $G(x)$ 叫做这个 CRC 码的生成多项式。

校验码的具体生成过程为：假设发送信息用信息多项式 $C(x)$ 表示，将 $C(x)$ 左移 r 位，则可表示成 $C(x)*2^r$，这样 $C(x)$ 的右边就会空出 r 位，这就是校验码的位置。通过 $C(x)*2^r$ 除以生成多项式 $G(x)$ 得到的余数就是校验码。

接收方将接收到的二进制序列数（包括信息码和 CRC 码）除以多项式，如果余数为 0，则说明传输中无错误发生，否则说明传输有误。

CRC 用于检错，一般能检测如下错误：突发长度小于 $n-k+1$ 的突发错误；或者大部分

突发长度等于$n-k+1$的错误，其中不可检出错误的仅占$2^{-(n-k-1)}$；或者大部分突发长度大于$n-k+1$的错误，其中不可检出错误的仅占$2^{-(n-k)}$；或者所有与许用码组码距小于$d_{\min}-1$的错误以及所有奇数个错误。

下面举个例子说明 CRC 的编码过程。

例 11-16：假设生成多项式是$G(x)=x^3+x+1$。4 位的原始报文为 1010，求编码后的报文。

首先，将生成多项式$G(x)=x^3+x+1$转换成对应的二进制除数 1011；其次，由于生成多项式有 4 位（$r+1$），要把原始报文$C(x)$左移 3（r）位变成 1010000；再次，用生成多项式对应的二进制数对左移 4 位后的原始报文进行模 2 除；最后，得到余数 011，即为校验位；最后得到编码后的报文（CRC 码）为 1010011。

CRC 编码完成后，将码字添加在原始数据比特流后面，然后再将整体发送。在接收端，则对整个数据进行 CRC 译码，只有所有接收到的比特（包括原始数据比特和 CRC 码字）都经过译码后，才能给出有无错误的标志。若校验结果为 0，则说明传送时没有发现错误；否则意味着传输错误，需要采取一定的措施。

11.5.2 CRC16 编码器的 Verilog HDL 实现

根据应用环境的不同，CRC 校验码可以分为几种不同的标准，如 CRC-12 码、CRC-16 码、CRC-CCITT 码等。CRC-12 码通常用来传送 6 比特字符串，CRC-16 码和 CRC-CCITT 码则用来传送 8 比特字符串。其中 CRC-16 码为美国采用，而 CRC-CCITT 码则为欧洲采用。

下面介绍 CRC-16 码的串行编码实现。CRC-16 码采用的生成多项式为$G(x)=x^{16}+x^{15}+x^2+1$，其逻辑实现结构如图 11-32 所示。初始化时，每一位寄存器都清零，然后每输入一个数据，16 级移位寄存器按照异或逻辑由低到高移动一位，直到一组校验数据结束。此时，16 级移位寄存器的内容就是该组数据的 CRC-16 的校验位。

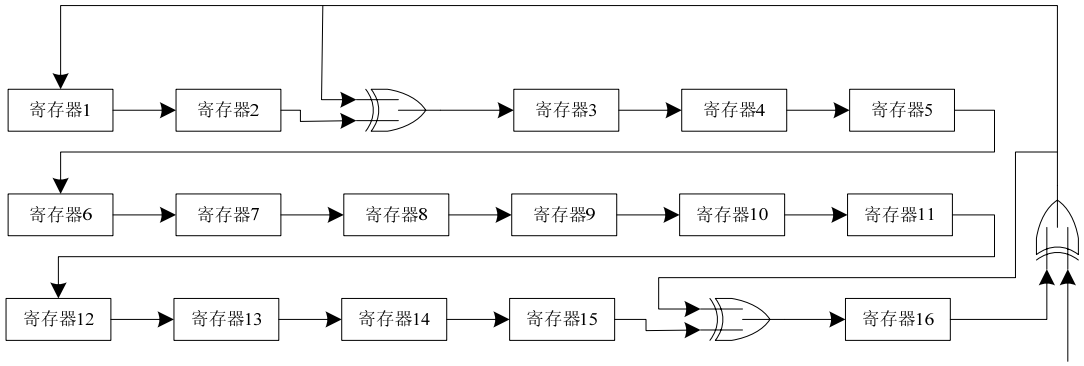


图 11-32 CRC-16 校验码的逻辑结构图

例 11-17：用 Verilog 实现一个串行 CRC-16 编码器。

```
module crc_16(clk, reset, x, crc_reg, crc_s);
        input clk;                  //系统归工作时钟
        input reset;                //  复位信号
        input x;                    //串行输入数据
        output [15:0] crc_reg; //CRC 编码输出
        output crc_s;     //CRC 同步信号，标志着一帧编码的结束

        reg [15:0] crc_reg;
```

```
reg crc_s;
reg [3:0] cnt;
wire [15:0] crc_enc;

always @(posedge clk) begin
   if(!reset) begin
         crc_reg <= 0;
         cnt <= 0;
   end
   else begin
        crc_reg <= crc_enc;
         cnt <= cnt +1;
         if(cnt == 0)
            crc_s <= 0;
         else
            crc_s <= 1;
   end
end

assign crc_enc[0] = crc_reg[15]^x;
assign crc_enc[1] = crc_reg[0];
assign crc_enc[2] = crc_reg[1]^crc_reg[15]^x;
assign crc_enc[14:3] = crc_reg[13:2];
assign crc_enc[15] = crc_reg[14]^crc_reg[15]^x;

endmodule
```

程序在 ISE 中综合后的 RTL 级结构图如图 11-33 所示，其基本结构和图 11-32 所示的 CRC16 编码器结构是一致的。

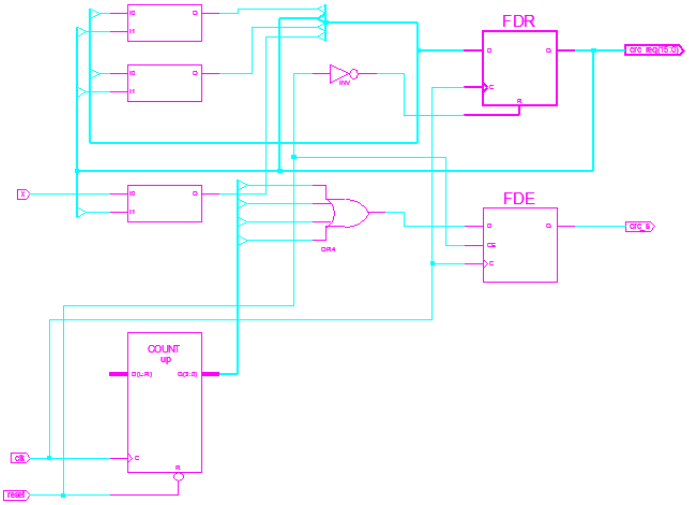


图 11-33　CRC-16 编码器的 RTL 结构图

上述程序在 ISE Simualtor 中的仿真结果如图 11-34 所示，从而验证了设计的正确性。

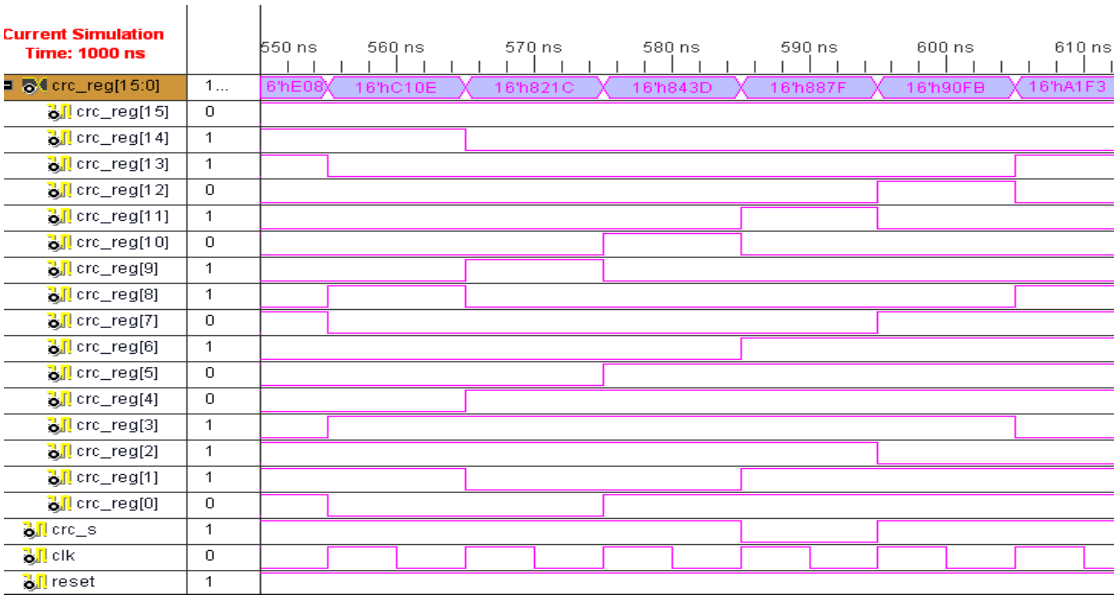


图 11-34　CRC-16 编码器模块的局部仿真结果

11.6　片内存储器的 Verilog HDL 实现

片内存储器分为 RAM 和 ROM 两大类。RAM（Random Access Memory）的全名为随机存取记忆体，可在任何时候完成读写操作，掉电后丢失数据。ROM 是 Read Only Memory 的意思，也就是说这种存储器只能读，不能写。本节主要介绍各类 RAM 和 ROM 模块的 Verilog HDL 实现。

11.6.1　RAM 的 Verilog HDL 实现

1．RAM 简介

RAM 作为一种矩阵存储单元，其内部结构如图 11-35 所示。工作时可以随时从任何一个指定地址中读出数据，也可以随时将数据写入任何一个指定的存储单元中。

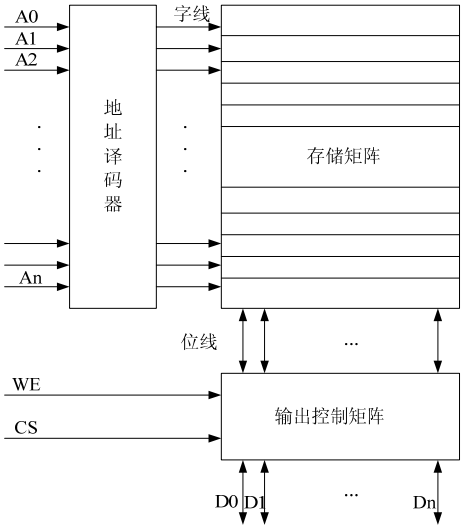


图 11-35　RAM 内部结构示意图

RAM 单元根据地址总线、数据总线以及读写控制线的数目可以分为单口 RAM、双口

RAM 两大类。单口 RAM 只有一套数据总线、地址总线和读写控制线，因此当多个外设需要访问同一块单口 RAM 时，需要通过仲裁电路来判断。双口 RAM 具有两套完全独立的数据线、地址线和读写控制线，从而实现了大量数据的高速访问以及不同时钟域的数据交换。

2．RAM 的 Verilog HDL 实现

在 Verilog HDL 中，若干个相同宽度的向量构成数组，其中 reg 型数组变量就代表着存储器。例如：

```
reg [7:0] memory [1023:0];
```

该语句定义了 1024 个字的存储器变量 memory，每个字的字长为 8 位，经过定义后的 memory 型变量可以用下面的语句对存储器单元赋值：

```
memory [7] = 90; //存储器 memory 的第 7 个字被赋值为 90
```

存储器单元中的数据也可以读出，因此存储器型变量相当于一个 RAM。由于存储器由逻辑资源产生，因此存储容量越大，所需要的逻辑资源就越多。

（1）单口 RAM 单元

单口 RAM，只有一套地址总线，读和写是分开（至少不能在同一个周期内完成）。下面给出一个 8×8 位 RAM 的设计实例。

例 11-18：利用 Verilog HDL 语言设计一个单口 RAM 模块。

```
module ram_single(
    clk, addm, cs_n, we_n, din, dout
    );
     input            clk;
     input   [2:0] addm;
     input            cs_n;
     input            we_n;
     input   [7:0] din;
     output [7:0] dout;

     reg      [7:0] dout;
     reg      [7:0] ram1 [7:0];

     always @(posedge clk) begin
       if(cs_n)
            dout <= 8'b0000_0000;
        else
            if(we_n)
                dout <= ram1[addm];
             else
                ram1[addm] <= din;
    end

endmodule
```

程序在 ISE 中的综合结果如图 11-36 所示，从中可以看出，综合工具自动将寄存器类型的 ram1 转化成标准的 RAM 模块。其中，addm 为 3 比特地址线，可以实现 8 个存储单元的

寻址；cs_n 为片选信号，低有效，当 cs_n 为低时，存储器处于工作状态（可以读或写）；当 cs_n 为高时，存储器处于禁止状态（强制输出 0）。we_n 为写使能信号，低有效，当 we_n 为高时，存储器处于读状态，否则处于写状态。dout 为存储器的输出端口，din 为存储器的输入端口。

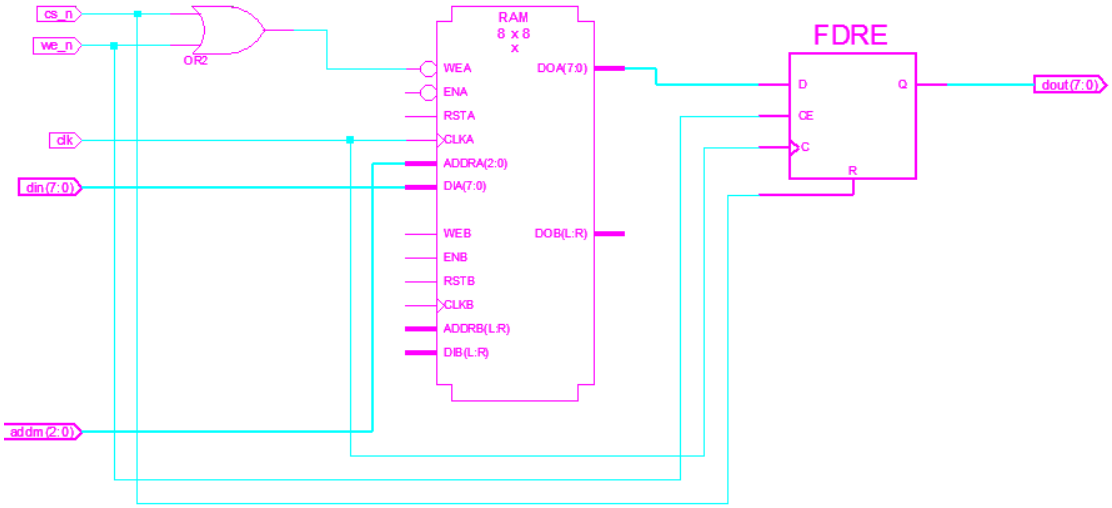


图 11-36　单口 RAM 程序综合后的 RTL 结构图

上述程序在 ISE Simulator 中的仿真结果如图 11-37 所示，根据控制输入端 cs_n 和 we_n 信号的不同组合，将仿真波形分为 3 个阶段。在仿真的第一阶段，cs_n 的电平为高，表明 RAM 并未使能，因此输出端 dout 为程序中设定的输出 0。第二阶段为写入阶段，cs_n 和 we_n 的电平都为低，共有 8 个周期，分别将 10~17 这 8 个数依次写入 RAM，其实地址为 2 号地址。第三阶段为读出阶段，cs_n 电平为低，而 we_n 电平为高，将写入阶段写入 RAM 的数据循环读出，起始地址为 2 号，数据依次为 10~17，表明了设计的正确性。

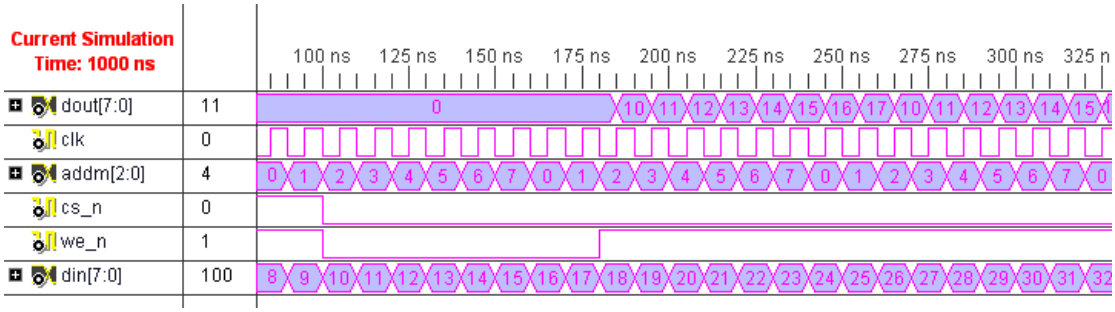


图 11-37　单口 RAM 程序仿真结果示意图

（2）双口同步 RAM 单元

例 11-19：双口同步 RAM 具有两套地址总线，一套用于读数据，一套用于写数据，二者可分别独立操作。下面给出一个 128×8 位双口 RAM 的 Verilog HDL 设计实例。

```
module ram_dual(q, addr_in, addr_out, d, we, clk1, clk2);
    output[7:0] q;
    input [7:0] d;
    input [6:0] addr_in;
    input [6:0] addr_out;
    input we, clk1, clk2;
```

```
reg [6:0] addr_out_reg;
reg [7:0] q;
reg [7:0] mem [127:0];

always @(posedge clk1)
begin
    if (we)
        mem[addr_in] <= d;
end

always @(posedge clk2) begin
    q <= mem[addr_out_reg];
    addr_out_reg <= addr_out;
end

endmodule
```

程序在 ISE 中的综合结果如图 11-38 所示，和图 11-36 对比后可以发现，读、写端口从物理上是分开的，也就意味着可以同时完成读、写操作。

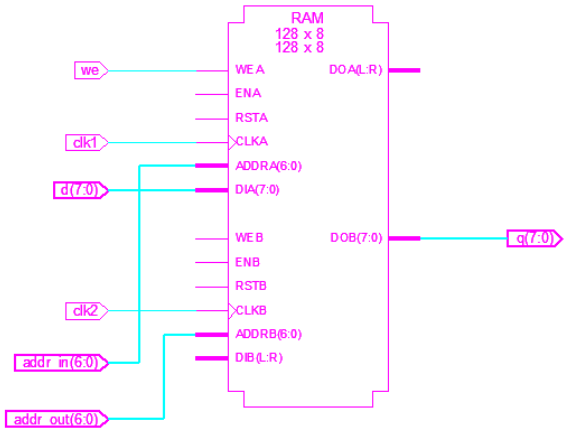


图 11-38　双口 RAM 综合后的 RTL 级结构示意图

上述程序在 ISE Simulator 中的仿真结果如图 11-39 所示，通过和例 11-18 相比可以看出，双口 RAM 可以同时完成读、写功能。在测试时，将读地址比写地址慢一个时钟周期，则可以看出，RAM 读出的数据都是上一个周期写入的数据，从而验证了设计的正确性。

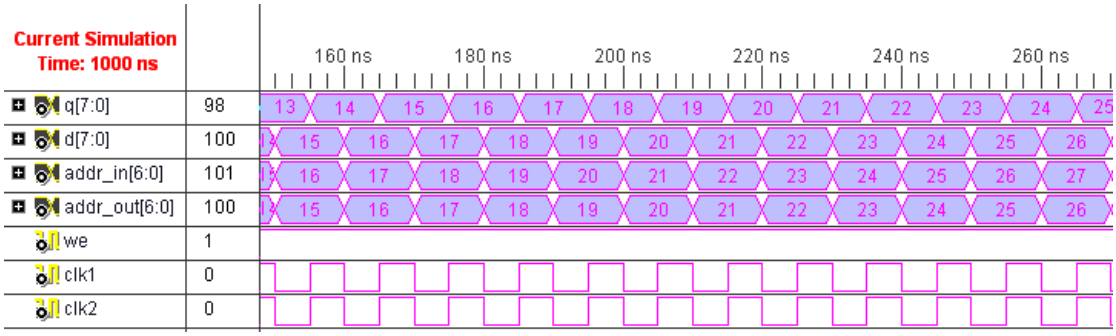


图 11-39　双口 RAM 的仿真结果

（3）ROM 单元

在数字系统中，由于 ROM 掉电后数据不会丢失，因此 ROM 单元也有着更广泛的应用。对于容量不大的 ROM，在 Verilog HDL 中可以通过 case 语句来实现。下面给出一个 8×8 位的 ROM 设计实例。

例 11-20：利用 Verilog HDL 语言实现一个 8×8 位的 ROM 模块。

```
module rom_demo(
    clk, addm, cs_n, dout
    );
     input            clk;
     input   [2:0] addm;
     input            cs_n;
     output [7:0] dout;

     reg       [7:0] dout;

     always @(posedge clk) begin
       if(cs_n)
            dout <= 8'b0000_0000;
        else
         case(addm)
                3'b000: dout <= 1;
                3'b001: dout <= 2;
                3'b010: dout <= 4;
                3'b011: dout <= 8;
                3'b100: dout <= 16;
                3'b101: dout <= 32;
                3'b110: dout <= 64;
                3'b111: dout <= 128;
            endcase
    end

endmodule
```

在应用中，case 语句中的数值可以根据实际需要修改，其中 addm 为地址输入信号，cs_n 为片选信号。程序在 ISE Similator 中的仿真结果如图 11-40 所示，该结果和代码中的数据是一致的，从而验证了设计的正确性。

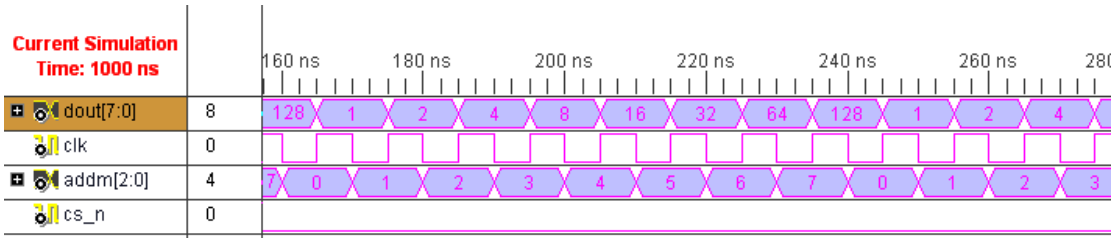


图 11-40 双口 ROM 的仿真结果

11.6.2 移位寄存器的 Verilog HDL 实现

寄存器是计算机和其他数字系统中用来存储代码或数据的逻辑部件。它的主要组成部分

是触发器。一个触发器能存储 1 位二进制代码，所以要存储 n 位二进制代码的寄存器就需要用 n 个触发器组成。移位寄存器应用很广，可构成移位寄存型计数器；顺序脉冲发生器；串行累加器；可用作数据转换，即把串行数据转换为并行数据，或把并行数据转换为串行数据等。

1．移位寄存器的工作原理

把若干个触发器串接起来，就可以构成一个移位寄存器。由 4 个边沿 D 触发器构成的 4 位移位寄存器逻辑电路如图 11-41 所示。数据从串行输入端 D1 输入。左边触发器的输出作为右邻触发器的数据输入。假设移位寄存器的初始状态为 0000，现将数码 D3D2D1D0(1101)从高位(D3)至低位依次送到 D1 端，经过第一个时钟脉冲后，Q0＝D3。由于跟随数码 D3 后面的数码是 D2，则经过第二个时钟脉冲后，触发器 FF0 的状态移入触发器 FF1，而 FF0 变为新的状态，即 Q1＝D3，Q0＝D2。依此类推，可得 4 位右向移位寄存器的状态，如表 11-3 所示。

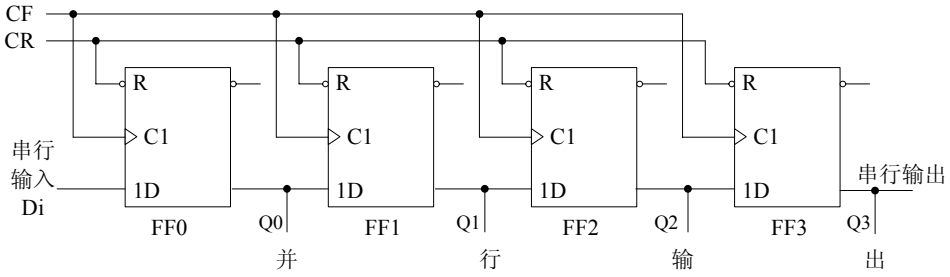


图 11-41 用边沿 D 触发器构成的 4 位移位寄存器

表 11-3　　图 11-41 电路的状态表

CP	Q0	Q1	Q2	Q3
0	0	0	0	0
1	D3	0	0	0
2	D2	D3	0	0
3	D1	D2	D3	0
4	D0	D1	D2	D3

由表 11-3 可知，输入数码依次地由低位触发器移到高位触发器，作右向移动。经过 4 个时钟脉冲后，4 个触发器的输出状态 Q3Q2Q1Q0 与输入数码 D3D2D1D0 相对应。这样，就可将串行输入（从 D1 端输入）的数码转换为并行输出（从 Q3、Q2、Q1、Q0 端输出）的数码。

2．移位寄存器的 Verilog HDL 实现

在 Verilog HDL 语言实现移位寄存器只需要从行为级描述即可，下面给出两个应用实例。首先将图 11-41 所示的移位寄存器实现出来，再给出一个多位宽的移位寄存器描述实例。

例 11-21：通过 Verilog HDL 语言完成图 11-41 中的移位寄存器。

```
module onebit_shift(
    clk, din, dout
    );
     input                clk;
     input                din;
     output   [3:0]       dout;

     reg      d0, d1, d2, d3;
```

```
    always @(posedge clk) begin
        d0 <=   din;
         d1 <=   d0;
         d2 <=   d1;
         d3 <=   d2;
    end

    assign dout = d3;

endmodule
```

上述程序在 ISE 中综合后的 RTL 级如图 11-42 所示，可以看出，其和图 11-41 所示的移位寄存器设计是一致的。

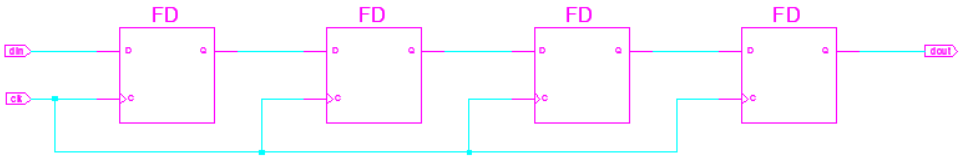


图 11-42 单比特移位寄存器的 RTL 级结构示意图

由于在实际中，移位寄存器有着广泛的应用，下面给出一些更复杂的移位寄存器设计实例，其中深度和位宽都大于 1。

例 11-22：通过 Verilog HDL 实现 8 比特位宽、64 深度的移位寄存器。

```
module shift_8x64_taps(
    clk,   shift,   sr_in,   sr_out
    );

     input              clk, shift;
     input   [7:0]   sr_in;
     output [7:0]     sr_out;

     reg       [7:0] sr [63:0];
     integer n;

     always@(posedge clk)
     begin
          if (shift == 1'b1)
          begin
               for (n = 63; n>0; n = n-1)
               begin
                    sr[n] <= sr[n-1];
               end

               sr[0] <= sr_in;
          end
     end
```

```
assign sr_out = sr[63];

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 11-43 所示，可以看出，输出数据的值比输入小 64，正是移位寄存器的长度，从而表明了设计的正确性。

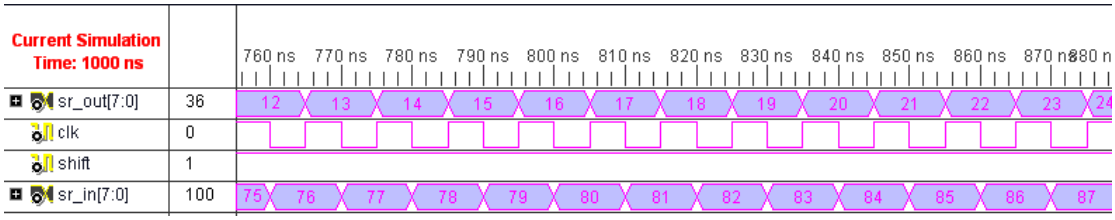


图 11-43　移位寄存器的仿真结果

11.7　SPI 接口协议的 Verilog HDL 实现

串行外设接口 SPI（Serial Peripheral Interface）是一种由 Motorola 公司推出的一种同步串行接口，得到了广泛应用。SPI 接口可以共享，便于组成带多个 SPI 接口器件的系统，且传送速率可编程，连接线少，具有良好的扩展性，是一种优秀的同步时序电路。

11.7.1　SPI 通信协议

1．SPI 总线介绍

SPI，顾名思义就是串行外围设备接口，只需 4 条线就可以完成主、从与各种外围器件全双工同步通讯。4 根接口线分别是：串行时钟线（SCK）、主机输入/从机输出数据线（MISO）、主机输出/从机输入数据线（MOSI）、低电平有效从机选择线 CS。SPI 总线主要特点有：

- 可以同时发出和接收串行数据;
- 可以当作主机或从机工作;
- 提供频率可编程时钟;
- 发送结束中断标志;
- 写冲突保护;
- 总线竞争保护等。

SPI 系统可分为主机设备和从机设备两类设备，其中主机提供 SPI 时钟信号和片选信号；从机是接收 SPI 信号的任何集成电路，包括简单的 TTL 移位寄存器，复杂的 LCD 显示驱动器，A/D、D/A 转换子系统或其他的 MCU。当 SPI 工作时，在移位寄存器中的数据逐位从输出引脚（MOSI）输出，同时从输入引脚（MISO）逐位接收数据。发送和接收操作都受控于 SPI 主设备的时钟信号（SCK），从而保证同步，因此只能有一个主设备，但从设备则可以有多个，通过不同的片选信号（CS_n）选中可同时选中一个或多个从设备，其典型统连接关系如图 11-44 所示。

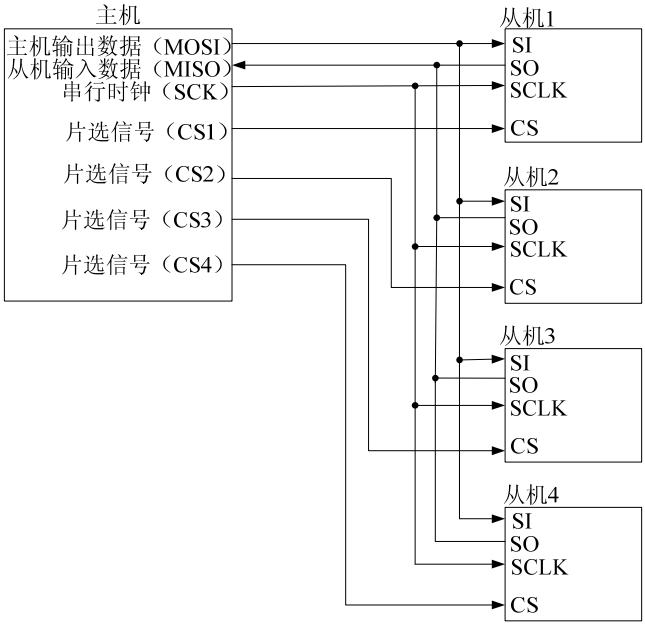


图 11-44 典型的 SPI 总线系统连接方式

2．SPI 接口的总线时序

SPI 的工作模式分为主模式和从模式，二者都需要在 SCK 的作用下才能工作；但主模式不需要 CS 信号，而从模式必须在 CS 信号有效的情况下才能完成。不论是在主模式下还是在从模式下，都要在时钟极性（CPOL）和时钟相位（CPHA）的配合下才能有效地完成一次数据传输。其中，时钟极性表示时钟信号在空闲时的电平；时钟相位决定数据是在 SCK 的上升沿采样还是下降沿采样。根据时钟极性和时钟相位的不同组合，可以得到 SPI 总线的 4 种工作模式，如图 11-45 所示。

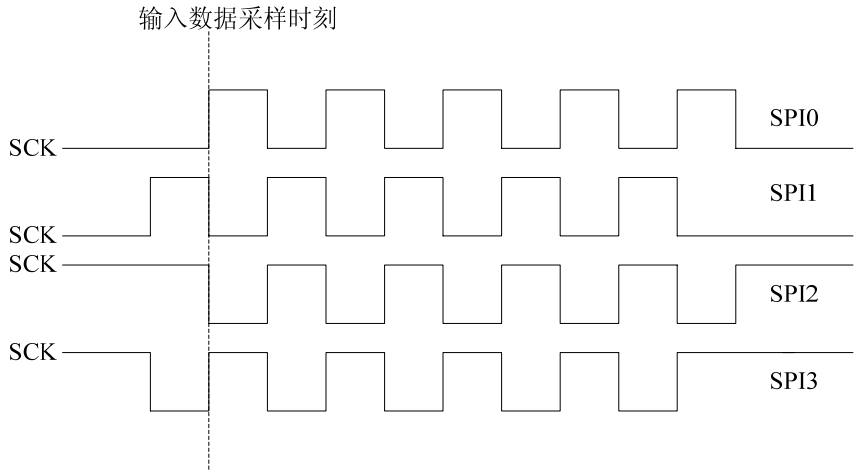


图 11-45　SPI 总线四种工作方式

- SPI0 模式下的 CPOL 为 0，SCK 的空闲电平为低；CPHA 为 0，数据在串行同步时钟的第一个跳变沿（由于 CPOL 为低，因此第 1 个跳变沿只能为上升沿）时数据被采样。
- SPI1 模式下的 CPOL 也为 0，SCK 的空闲电平为低；但是 CPHA 为 1，数据在串行同步时钟的第二个跳变沿（由于 CPOL 为低，因此第 2 个跳变沿只能为下降沿）时数据被采样。
- SPI2 模式下的 CPOL 为 1，SCK 的空闲电平为高；CPHA 为 0，数据在串行同步时钟的第 1 个跳变沿（由于 CPOL 为高，因此第 1 个跳变沿只能为下降沿）时数据被采样。

● SPI3 模式下的 CPOL 为 1，SCK 的空闲电平为高；CPHA 为 1，数据在串行同步时钟的第 2 个跳变沿（由于 CPOL 为高，因此第 1 个跳变沿只能为上升沿）时数据被采样。

在上述 4 种模式中，使用的最为广泛的是 SPI0 和 SPI3 方式。由于每一种模式都与其他三种不兼容，因此为了完成主、从设备间的通讯，主、从设备的 CPOL 和 CPHA 必须有相同的设置。读者需要注意的是：如果主设备/从设备在 SCK 上升沿发送数据，则从设备/主设备最好在下降沿采样数据；如果主设备/从设备在 SCK 下降沿发送数据，则从设备/主设备最好在 SCK 上升沿采样数据。

在 CPLD/FPGA 芯片内实现 SPI 接口，除了需要根据外设工作要求来调整 CPOL 和 CPHA，使得主、从设备设置一样外，还需要注意字节在串行数据线上的传送顺序，即 MOSI 和 MISO 上是高比特首先传输还是低比特首先传输。下面以高比特在前的传输为例给出 4 类 SPI 模式完整的 SPI 接口时序，分别如图 11-46、图 11-47 所示。

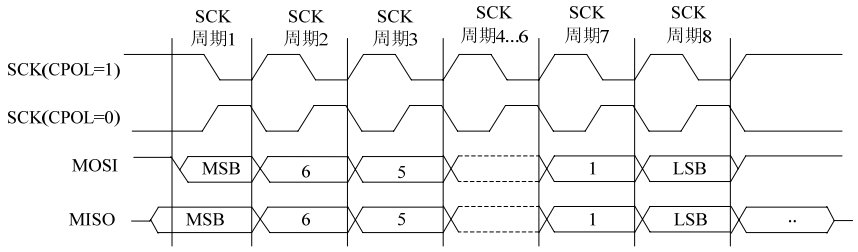


图 11-46　CPHA=0 时 SPI 总线数据传输时序

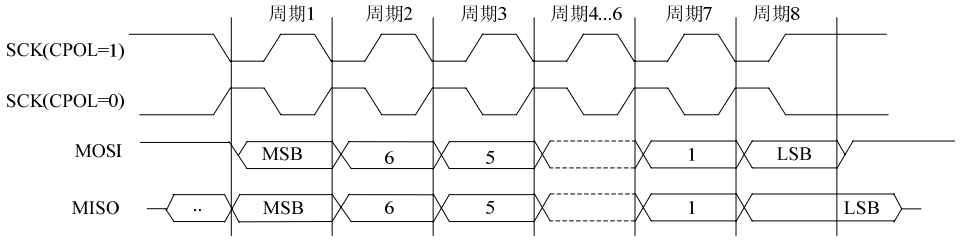


图 11-47　CPHA=1 时 SPI 总线数据传输时序

11.7.2　SPI 协议的 Verilog HDL 实现

在通过 HDL 语言实现 SPI 接口协议完成通信或者对具有 SPI 接口的芯片进行编程以及功能配置时，需要注意以下几个问题：

（1）确认接口芯片读入或送出数据发生在时钟信号的上升沿或是下降沿，并在数据保持稳定后再进行数据的读写操作；

（2）数据需保持的最短有效时间（一般而言查阅接口芯片的数据手册就可以得到），避免在接口芯片未完成读写数据时即进行下一次的操作；

（3）对于从节点主动寻求主动节点服务的接口芯片，应注意 SPI 接口芯片发出中断数据请示信号后，所需的响应时间，以避免出现接口芯片发出请示服务信号后长时间处于等待状态而致使数据信息丢失等现象的出现。

（4）在进行数据通信时，确认通信字节位传输的顺序是以 MSB→LSB 方式进行，还是以 LSB→MSB 的方式进行。

例 11-23：使用 Verilog HDL 语言实现 SPI0 模式的 SPI 主模式，其中读、写操作都是低字节在前，高字节在后，每次传送 1 个字节。

```
module spi_master(addr, in_data, out_data, rd, wr, cs, clk, miso, mosi, sclk);
    input wire [1:0] addr;
    input wire [7:0] in_data;
```

```
output reg [7:0] out_data;
input wire rd;
input wire wr;
input wire cs;
input wire clk;
inout miso;
inout mosi;
inout sclk;

reg sclk_buffer = 0;
reg mosi_buffer = 0;
reg busy = 0;

reg [7:0] in_buffer = 0;
reg [7:0] out_buffer = 0;
reg [7:0] clkcount = 0;
reg [7:0] clkdiv = 0;
reg [4:0] count = 0;

always@(cs or rd or addr or out_buffer or busy or clkdiv) begin
    out_data = 8'bx;
    //完成读操作
    if(cs && rd)    begin
        case(addr)
            2'b00:    begin
                        out_data = out_buffer;
            end
            2'b01:    begin
                        out_data = {7'b0, busy};
            end
            2'b10:    begin
                        out_data = clkdiv;
            end
        endcase
    end
end

always@(posedge clk)   begin
    if(!busy) begin
        //完成写操作
        if(cs && wr)
        begin
            case(addr)
            2'b00: begin
```

```
                        in_buffer = in_data;
                            busy = 1'b1;
                    end
                    2'b10: begin
                            clkdiv = in_data;
                    end
                    endcase
                end
            end
            else begin
                clkcount = clkcount + 1;
                if(clkcount >= clkdiv) begin
                    clkcount = 0;

                    if((count % 2) == 0)       begin
                        mosi_buffer = in_buffer[7];
                        in_buffer = in_buffer << 1;
                    end

                    if(count > 0 && count < 17) begin
                        sclk_buffer = ~sclk_buffer;
                    end

                    count = count + 1;

                    if(count > 17) begin
                        count = 0;
                        busy = 1'b0;
                    end
                end
            end
        end

        always@(posedge sclk_buffer) begin
            out_buffer = out_buffer << 1;
            out_buffer[0] = miso;
        end

        assign sclk = sclk_buffer;
        assign mosi = mosi_buffer;

    endmodule
```

本设计的读、写功能都是正确的，下面给出写功能的测试文件，而将读功能的验证作为一个练习题留给读者完成。完成写功能测试的代码如下：

```
module tb_spi_master;

    // 定义输入测试激励
    reg [1:0] addr;
    reg [7:0] in_data;
    reg rd;
    reg wr;
    reg cs;
    reg clk;

    // 定义输出测试激励
    wire [7:0] out_data;

    // Bidirs
    wire miso;
    wire mosi;
    wire sclk;

    // 例化被测试模块(UUT)
    spi_master uut (
        .addr(addr),
        .in_data(in_data),
        .out_data(out_data),
        .rd(rd),
        .wr(wr),
        .cs(cs),
        .clk(clk),
        .miso(miso),
        .mosi(mosi),
        .sclk(sclk)
    );

    initial begin
        // 初始化输入变量
        addr = 0;
        in_data = 0;
        rd = 0;
        wr = 0;
        cs = 0;
        clk = 0;

        // 复位信号持续 80 个仿真时间单位
        #80;
        cs = 1;
```

```
        rd = 0;
        // 下面为仿真的测试激励
        repeat(10) begin
            wr = 1;
            in_data = in_data + 1;
            #10;
            wr = 0;
            #200;
        end
    end

    //产生仿真所需的时钟信号
    always #5 begin
        clk = ~clk;
    end

endmodule
```

上述在 ISE Simulator 中的仿真结果如图 11-48 所示，可以看出 spi_master 模块正确实现了数据发送，达到了设计要求。

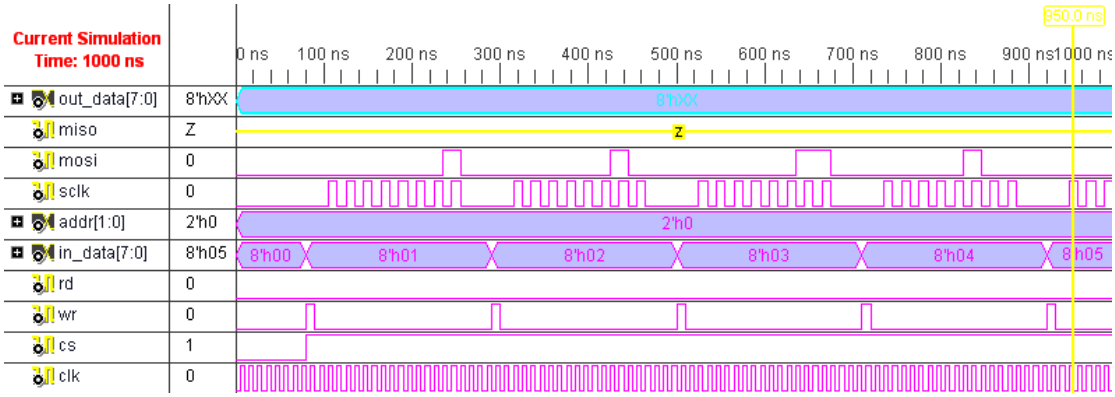


图 11-48　SPI 写操作的仿真结果

11.8　本章小结

本章主要介绍了基本的、常用的 Verilog HDL 语言设计，从最简单的时钟处理电路开始到 SPI 串口通信协议的开发为止，总共有 7 大类实例。首先，介绍了整数分频电路、非整数分频电路以及同步整形电路，其中同步整形电路的应用非常广泛，但难度较大，读者需要仔细体会。其次，介绍了基本的乘加单元，从用户角度可以认为在 Verilog HDL 中，只有加法操作和乘法操作是最基本的，但实质上其也是由典型的与非门来实现的。接下来，介绍了 7 段数码管和按键的响应接口，并给出其 Verilog HDL 实现。最后，介绍了 CRC 编码器、RAM、FIFO 以及 SPI 口等典型的接口电路，具有较强的实用性。

11.9　思考题

1．以二分频、三分频电路为例，简要描述整数分频电路的核心处理思想。
2．什么是同步整形电路，具有什么功能？

3．通过 Verilog HDL 实现小数分频有哪几种方法，各有什么特点？
4．Verilog HDL 中最基本的算术操作是哪一种数学运算？
5．乘加运算后的数据截尾和扩位规则是什么？
6．数码管电路有哪些注意事项？
7．矩阵键盘的扫描原理是什么？如何完成键盘的防抖操作？
8．CRC16 的编码的原理是什么？如何通过 Verilog HDL 实现？
9．片内存储器包括哪些种类？如何通过 Verilog HDL 语言实现？
10．SPI 接口协议有什么特点？如何通过 Verilog HDL 语言实现？
11．完成例 11-23 中，SPI 接口模块接收功能的仿真。

第 12 章　Xilinx 硬核模块的 Verilog HDL 调用

Xilinx 系列 FPGA 具有丰富的专用硬件模块，性能高且不占用 Slice 资源，属于 FPGA 芯片中关键的硬件资源，在工程中具有重要意义。底层单元可通过硬件原语来调用，也可以通过 Core Generator 调用，二者的关系类似于 DOS 程序和 Windows 界面程序的区别。Xilinx 的底层单元包括全局时钟网络、DLL 模块、DCM 模块、内嵌的块存储单元、硬核乘法器、高速收发器以及嵌入式处理器等，都经过 ASIC 设计验证，可工作在芯片允许的最高频率。需要注意的是，不同系列芯片底层单元的属性是不同的，一般来讲 Virtex 系列的底层单元性能高于 Spartan 系列的性能。本章主要介绍以 IP Core 或原语的方式来调用 FPGA 底层单元的使用方法。

12.1 差分 I/O 对管脚的 Verilog HDL 调用

12.1.1 差分 I/O 对管脚结构说明

1．可编程逻辑器件管脚说明

Xilinx 可编程逻辑器件（CPLD/FPGA）的管脚，可以分为电源、地、时钟输入管脚、配置管脚、JTAG 专用管脚、输入/输出管脚等 6 大类。可编程逻辑器件经常需要和不同接口电平连接，因此，可编程逻辑器件的管脚经常分为若干个分组（Bank），每个分组支持一种电平标准，根据芯片逻辑门数和管脚的不同，分组数以及分组内管脚的名称也是不同的。在 Xilinx 公司的标准中，分组是由封装确定的。例如，所有 VQ100 封装的 Xilinx FPGA 芯片的 100 个管脚都分为 4 个分组，每组 25 个管脚，详细分类如图 12-1 所示。

● 用户自定义专用输入/输出管脚（IO）

用户自定义专用输入/输出管脚是 CPLD/FPGA 主要管脚类型，可被配置为输入、输出以及双向端口，也可以将其按对组合，形成差分端口。其接口电平、驱动能力以及逻辑处理都可用户自定义。可以说 IO 端口可编程性诠释了可编程逻辑器件本质优点，也是与 DSP 等可编程器件的主要区别之一。

● 专用输入管脚（IP）

专用输入管脚（IP）没有输出结构，只能用于信号输入。当然，IP 管脚的接口电平和逻辑也能用户自定义。此外，IP 管脚也可成对组成差分输入接口。其命名规则为 IP_Lxxy_#，各个字符的含义和 IO 管脚中的含义一样。

● JTAG 专用管脚（JTAG）

每款 FPGA 芯片都具有 4 个专用的 JTAG 电压（TCK、TMS、TDI 以及 TDO），它们由 VCCAUX 供电，用于完成芯片和 PC 机的通信，可实现芯片配置以及检测等功能。JTAG 管脚不能用于 IO 管脚。

● 专用配置管脚（CONFIG）

FPGA 配置过程中需要数目较多的管脚，其中大多管脚都可复用，只有 DONE 和 PROG_B 两个管脚为专用配置管脚，不能复用。其中 DONE 和 PROG_B 管脚也有参考电压 VCCAUX 供电。

● 专用内核电压管脚（VCCINT）

内核电压管脚也为专用管脚，为 FPGA 内部逻辑提供电源。不同型号以及同一型号不同封装的内核电压管脚数量、电压值都不一样，但一般都以 VCCINT 命名。例如，Spartan 3E

系列芯片的 VCCINT 要求 1.2V，而 Spartan 2 系列芯片的 VCCINT 为 2.5V。VCCINT 对电源的要求很高，且电流也较大。在设计中，要对每一个 VCCINT 的输入管脚信号进行滤波，以保证芯片正常工作。

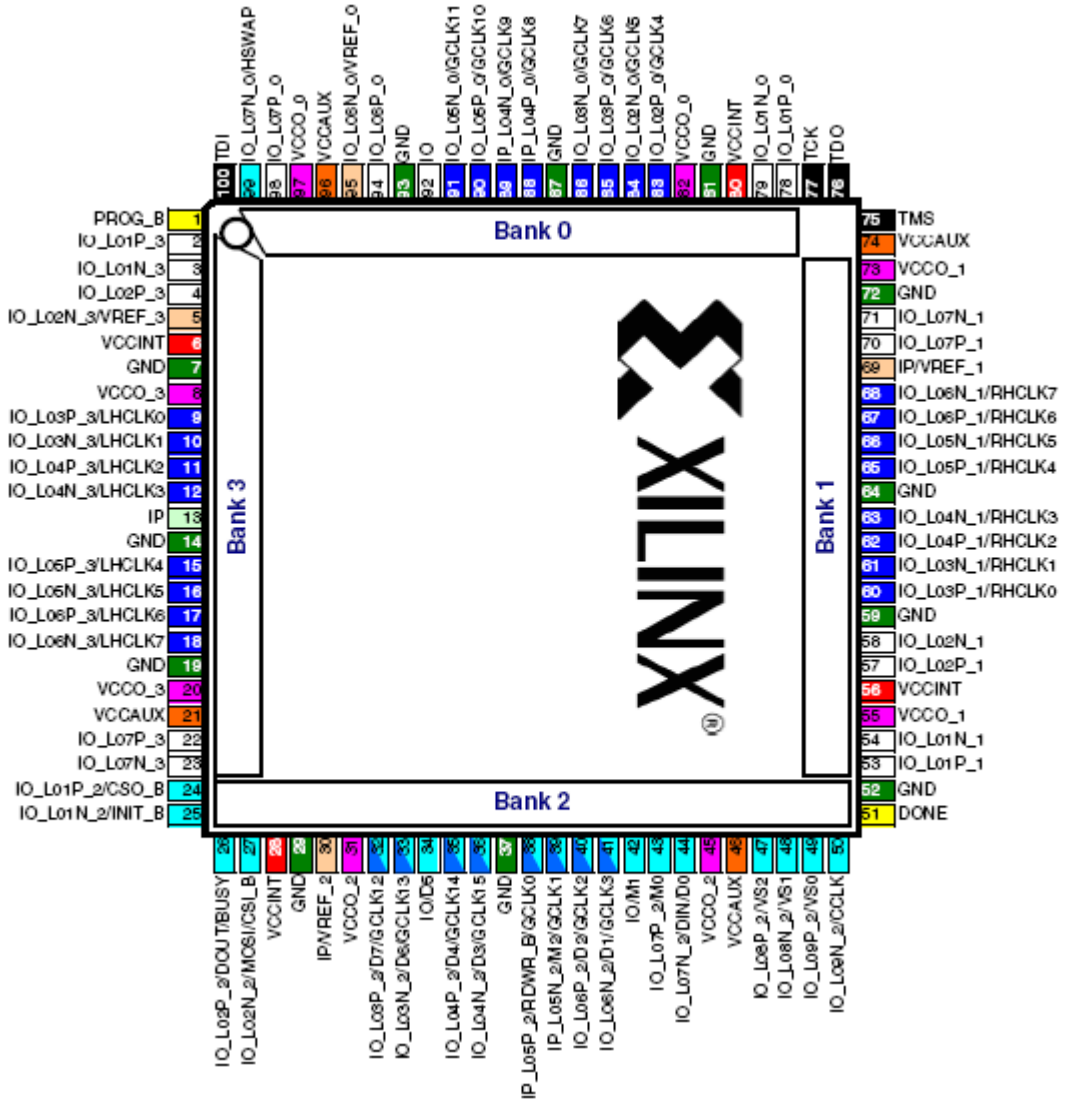


图 12-1　Xilinx FPGA VQ100 封装的管脚分组示意图

● 专用辅助电压管脚（VCCAUX）

辅助电压管脚也为专用管脚，为配置管脚和JTAG管脚提供电压，以VCCAUX命名。不同型号以及同一型号不同封装的内核电压管脚数量、电压值也是不一样的。需要注意的是，在早期的FPGA芯片中，并没有该管脚。

VCCAUX 对电源的要求不是很高，但在设计中最好也对其输入进行滤波，提高电源信号质量。

● 专用 I/O 电压管脚（VCCO）

IO电压管脚是为IO管脚供电的专用管脚，其数量由芯片的管脚分组（BANK）数决定。每个分组的IO电压都和该组的VCCO保持一致。VCCO的命名规则为VCCO_#，其中的“#”为管脚分组号。

VCCO对电源的要求较多，每个输入电压都需要进行滤波处理。

● 接地管脚（GND）

接地管脚是一组专用管脚，以 GND 命名，其数量由芯片型号的封装类型同时决定。所

有的 GND 管脚必须都接地，否则会造成 FPGA 芯片不能正常工作。

● 无连接管脚（N.C.）

无连接管脚在 FPGA 内部没有任何连接，以 N.C.命名。所有连接到该管脚的信号相当于悬空。一直在设计中直接悬空即可。并不是每一款 FPGA 都具有无连接管脚。

2．差分管脚说明

Xilinx 差分 IO 管脚的具有统一的命名规则，下面以 Spartan 3E 系列芯片为例，I/O 端口一般以 IO_Lxxy_#命名，其中的“L”表明具有差分驱动能力；“xx”为两个 10 进制的整数，表明了差分对序号；“y”用于指定差分对中的电平属性，“p”为正，“n”为负；“#”给出了端口所在的分组号（Bank 序号）。典型实例如图 12-2 所示。

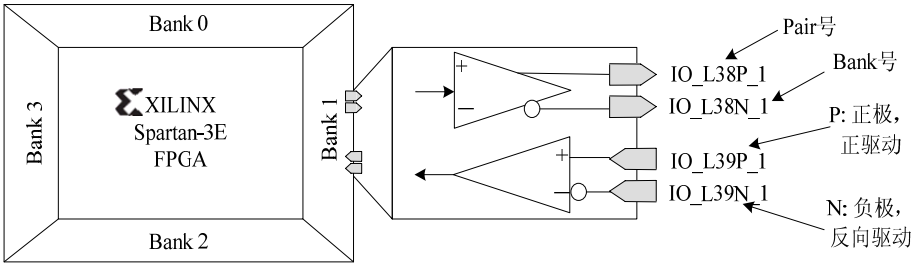


图 12-2　Spartan-3E 系列芯片的端口命名示例

3．在 ISE 中查阅差分管脚对

在 ISE10.1 中，保留了以前版本的引脚和区域约束编辑器 PACE（Pinout and Area Constraints Editor），并添加了布局规划器（Floorplanner）、FPGA 底层编辑器（FPGA Editor）功能的部分功能，形成了新的管脚和区域约束工具 Floorplan Editor。在 ISE10.1 中，Virtex-4、Virtex-5 以及 Spartan-3A 系列 FPGA 的管脚和区域约束都通过新的 Floorplanner Editor 来完成；其余系列芯片的类似约束还是通过 PACE 来完成的。

Floorplanner Editor 的启动方法有两种：一种是单独启动 PACE，直接点击“开始→程序→Xilinx ISE Design Suit 10.1→ISE 10.1→Accessories→PACE”，再选择 ISE 工程文件，启动 ISE 进入 PACE 工作界面；另一种是在工程经过综合后，在过程管理区双击“User Constraints→Floorplan IO-Pre-Synthesis”或“Floorplan Area/IO/Logic- Post- Synthesis”来打开 Floorplanner Editor，并自动加载当前工程。

在 FPGA 的差分应用中，差分对是固定的，需要匹配使用，由于目前 FPGA 芯片具有大量差分管脚，为了简化差分信息的记忆难度，PACE 提供了差分对匹配示意功能。点击“IOBs”菜单下的“Show Differential IO Pairs”命令，可在芯片管脚封装视图区列出所有的差分对，如图 12-3 所示。

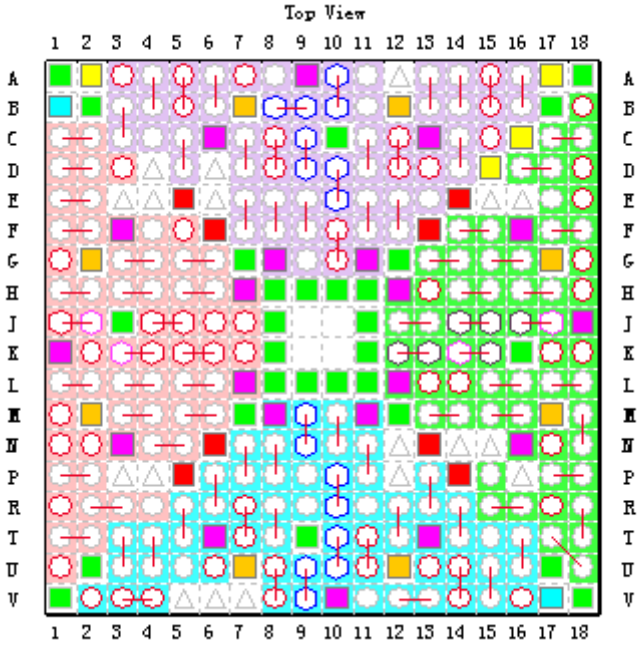


图 12-3　PACE 差分对示意图

Floorplan Editor 也可以成对标记差分管脚，选择“View”菜单下的“Differential IOs”命令，即可在 Package 页面用红色单端线标记出差分对，其中有端点端为差分对的“P”端。

12.1.2　调用差分 I/O 的参考设计

1．差分 I/O 转单端 I/O

该功能可利用 IBUFDS 原语完成。IBUFDS 原语用于将差分输入信号转化成标准单端信号，且可加入可选延迟。在 IBUFDS 原语中，输入信号为 I、IB，一个为主，一个为从，二者相位相反。

IBUFDS 的逻辑真值表如表 12-1 所列，其中“-*”表示输出维持上一次的输出值，保持不变。

表 12-1　　IBUFDS 原语的输入、输出真值表

输入		输出
I	IB	O
0	0	-*
0	1	0
1	0	1
1	1	-*

IBUFDS原语的例化代码模板如下所示：

```
// IBUFDS:  差分输入缓冲器（Differential Input Buffer）
//  适用芯片：Virtex-II/II-Pro/4, Spartan-3/3E
// Xilinx HDL 库向导版本，ISE 10.1
IBUFDS #(
.DIFF_TERM("FALSE"),
//  差分终端，可设置为True/Flase
.IOSTANDARD("DEFAULT")
```

```
// 指定输入端口的电平标准，如果不确定，可设为DEFAULT
) IBUFDS_inst (
.O(O),  // 缓冲单端输出信号
.I(I),   // 差分时钟的正端输入，需要和顶层模块的端口直接连接
.IB(IB) // 差分时钟的负端输入，需要和顶层模块的端口直接连接
);
```

在综合结果分析时，IBUFDS的RTL结构如图12-4所示。

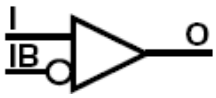


图 12-4　IBUFDS 原语的 RTL 结构图

2．单端 I/O 转差分 I/O

在 Spartan 3E 系列 FPGA 中可通过原语 OBUFDS 将单端信号转化成差分信号输出，其真值表如表 12-2 所示。

表 12-2　　OBUFDS 原语的输入、输出真值表

输入	输出	
I	O	OB
0	0	1
1	1	0

OBUFDS 原语的例化代码模板如下所示：

```
// OBUFDS: 将单端信号转换为差分信号
// 适用芯片：Virtex-II/II-Pro/4, Spartan-3/3E
// Xilinx HDL 库向导版本，ISE 10.1
OBUFDS #(
.IOSTANDARD("DEFAULT")
//指定输入端口的电平标准，如果不确定，可设为 DEFAULT
) OBUFDS_inst (
.O(O),      //差分信号的正端
.OB(OB),   //差分信号的负端
.I(I)       //输入信号
);
```

在综合结果分析时，IBUFDS的RTL结构如图12-5所示。

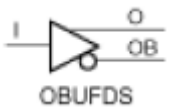


图 12-5　IBUFDS 原语的 RTL 结构图

12.2 DCM 模块的 Verilog HDL 调用

12.2.1　DCM 模块的说明

1．DCM 模块简介

数字时钟管理模块（Digital Clock Manager，DCM）是基于 Xilinx 的其他系列器件所采用的数字延迟锁相环（DLL，Delay Locked Loop）模块。在时钟的管理与控制方面，DCM

与 DLL 相比，功能更强大，使用更灵活。DCM 的功能包括消除时钟的延时、频率的合成、时钟相位的调整等系统方面的需求。DCM 的主要优点在于：①实现零时钟偏移（Skew），消除时钟分配延迟，并实现时钟闭环控制；②时钟可以映射到 PCB 上，用于同步外部芯片，这样就减少了对外部芯片的要求，将芯片内外的时钟控制一体化，以利于系统设计。对于 DCM 模块来说，其关键参数为输入时钟频率范围、输出时钟频率范围、输入/输出时钟允许抖动范围等。

2．DCM 模块的组成结构

DCM 共由四部分组成，如图 12-6 所示。其中最底层仍采用成熟的 DLL 模块；其次分别为数字频率合成器（DFS，Digital Frequency Synthesizer）、数字移相器（DPS，Digital Phase Shifter）和数字频谱扩展器（DSS，Digital Spread Spectrum）。不同芯片模块的 DCM 输入频率范围是不同的，例如：Virtex -4SX 系列芯片，低输入模式的外范围为 1~210MHz，高输入模式的范围为 50~350MHz；而 Spartan 3E 系列低、高两种模式的范围都只能是 0.2~333MHz。下面对 DCM 的四个部分分别进行介绍。

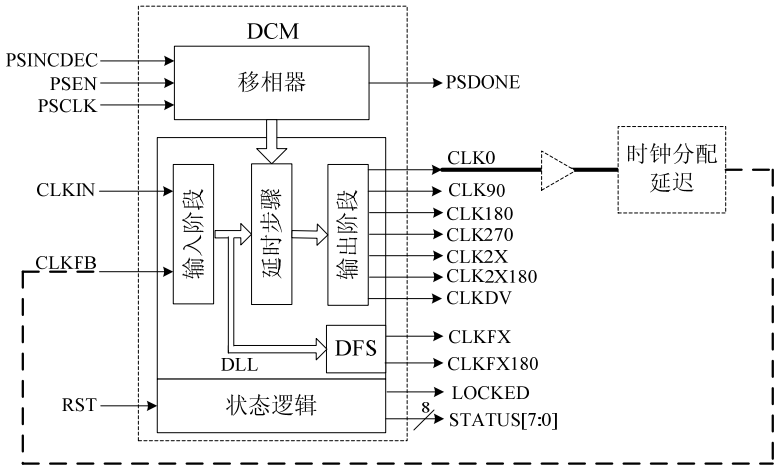


图 12-6　DCM 功能块和相应的信号

（1）DLL 模块

DLL 主要由一个延时线和控制逻辑组成。延时线对时钟输入端 CLKIN 产生一个延时，时钟分布网线将该时钟分配到器件内的各个寄存器和时钟反馈端 CLKFB；控制逻辑在反馈时钟到达时采样输入时钟以调整二者之间的偏差，实现输入和输出的零延时，如图 12-7 所示。具体工作原理是：控制逻辑在比较输入时钟和反馈时钟的偏差后，调整延时线参数，在输入时钟后不停地插入延时，直到输入时钟和反馈时钟的上升沿同步，锁定环路进入“锁定”状态，只要输入时钟不发生变化，输入时钟和反馈时钟就保持同步。DLL 可以被用来实现一些电路以完善和简化系统级设计，如提供零传播延迟，低时钟相位差和高级时钟区域控制等。

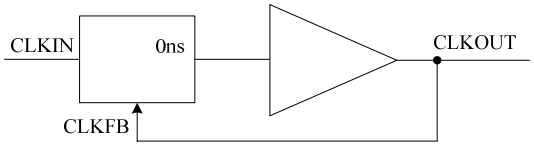


图 12-7　DLL 简单模型示意图

在 Xilinx 芯片中，典型的 DLL 标准原型如图 12-8 所示，其管脚分别说明如下：

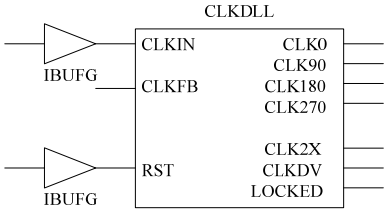


图 12-8　Xilinx DLL 的典型模型示意图

- CLKIN（源时钟输入）：DLL 输入时钟信号，通常来自 IBUFG 或 BUFG。
- CLKFB（反馈时钟输入）：DLL 时钟反馈信号，该反馈信号必须源自 CL K0 或 CL K2X，并通过 IBUFG 或 BUFG 相连。
- RST（复位）：控制 DLL 的初始化，通常接地。
- CLK0（同频信号输出）：与 CL KIN 无相位偏移；CL K90 与 CL KIN 有 90 度相位偏移；CL K180 与 CL KIN 有 180 度相位偏移；CL K270 与 CL KIN 有 270 度相位偏移。
- CLKDV（分频输出）：DLL 输出时钟信号，是 CLKIN 的分频时钟信号。DLL 支持的分频系数为 1.5，2，2.5，3，4，5，8 和 16。
- CLK2X（两倍信号输出）：CLKIN 的 2 倍频时钟信号。
- LOCKED（输出锁存）：为了完成锁存，DLL 可能要检测上千个时钟周期。当 DLL 完成锁存之后，LOCKED 有效。

在 FPGA 设计中，消除时钟的传输延迟，实现高扇出最简单的方法就是用 DLL，把 CLK0 与 CLKFB 相连即可。利用一个 DLL 可以实现 2 倍频输出，如图 12-9 所示。利用两个 DLL 就可以实现 4 倍频输出，如图 12-10 所示。

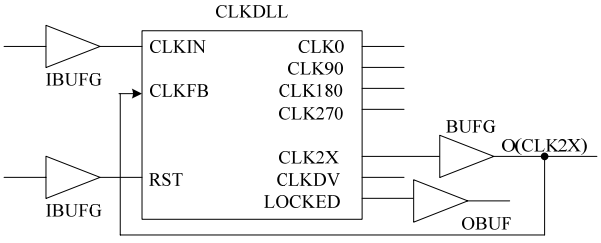


图 12-9　Xilinx DLL 2 倍频典型模型示意图

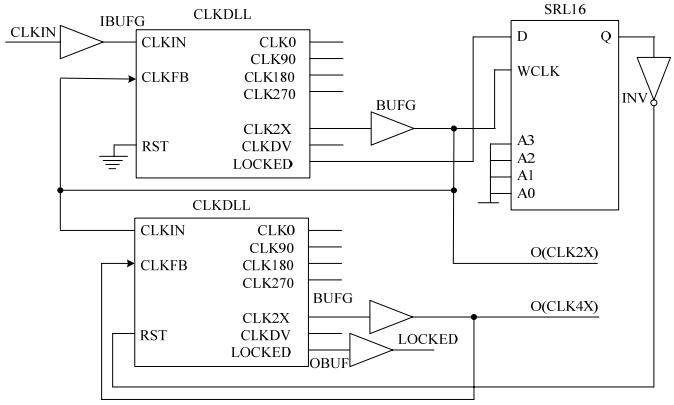


图 12-10　Xilinx DLL 4 倍频典型模型示意图

（2）数字频率合成器

DFS 可以为系统产生丰富的频率合成时钟信号，输出信号为 CLKFB 和 CLKFX180，可提供输入时钟频率分数倍或整数倍的时钟输出频率方案，输出频率范围为 1.5～320 MHz（不同芯片的输出频率范围是不同的）。这些频率基于用户自定义的两个整数比值，一个是乘因子（CLKFX_ MULTIPLY），另外一个是除因子（CLKFX_ DIVIDE），输入频率和输出频率之间的关系为：

$$F_{CLKFX} = F_{CLKIN} \times \frac{CLKFX\_MULTIPLY}{CLKFX\_DIVIDE}$$

比如取 CLKFX_MULTIPLY = 3，CLKFX_DIVIDE = 1，PCB 上源时钟为 100 MHz，通过 DCM 3 倍频后，就能驱动时钟频率在 300 MHz 的 FPGA，从而减少了板上的时钟路径，简化板子的设计，提供更好的信号完整性。

（3）数字移相器

DCM 具有移动时钟信号相位的能力，因此能够调整 I/O 信号的建立和保持时间，能支持对其输出时钟进行 0 度、90 度、180 度、270 度的相移粗调和相移细调。其中，相移细调对相位的控制可以达到 1%输入时钟周期的精度（或者 50ps），并且具有补偿电压和温度漂移的动态相位调节能力。对 DCM 输出时钟的相位调整需要通过属性控制 PHASE_SHIFT 来设置。PS 设置范围为-255 到+255，比如输入时钟为 200MHz，需要将输出时钟调整+ 0.9 *ns* 的话，PS =（0.9ns/ 5ns）×256 = 46。如果 PHASE_ SHIFT 值是一个负数，则表示时钟输出应该相对于 CLKIN 向后进行相位移动；如果 PHASE_SHIFT 是一个正值，则表示时钟输出应该相对于 CLKIN 向前进行相位移动。

移相用法的原理图与倍频用法的原理图很类似，只用把 CLK2X 输出端的输出缓存移到 CLK90、CLK180 或者 CLK270 端即可。利用原时钟和移相时钟与计数器相配合也可以产生相应的倍频。

（4）数字频谱合成器

Xilinx 公司第一个提出利用创新的扩频时钟技术来减少电磁干扰（EMI）噪声辐射的可编程解决方案。最先在 FPGA 中实现电磁兼容的 EMIControl 技术，是利用数字扩频技术（DSS）通过扩展输出时钟频率的频谱来降低电磁干扰，减少用户在电磁屏蔽上的投资。数字扩频（DSS）技术通过展宽输出时钟的频谱，来减少 EMI 和达到 FCC 要求。这一特点使设计者可极大地降低系统成本，使电路板重新设计的可能性降到最小，并不再需要昂贵的屏蔽，从而缩短了设计周期。

12.2.2 调用 DCM 模块的参考设计

例 12-1：在 ISE 中调用 DCM 模块，完成 61.44MHz 时钟信号到 40.96MHz 时钟信号的转换，二者的分频比为 3/2。

（1）新建工程，在源文件进程中双击“Create New Source”，然后在弹出的源文件窗口中选择“IP (CoreGen & Architecture Wizard)”，输入文件名“my_dcm”；再点击“Next”，在选择类型窗口中，“FPGA Features and Design → Clocking → Spartan-3E, Spartan-3A”，然后选择“Single DCM ADV v9.1i”，如图 12-11 所示。

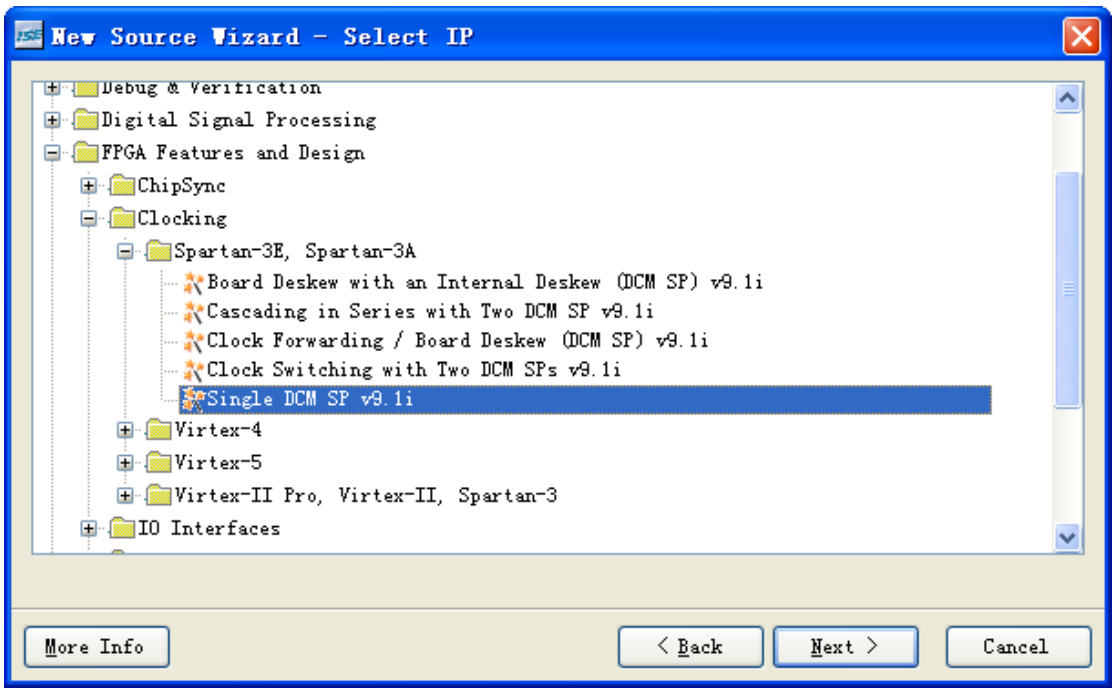


图 12-11　新建 DCM 模块 IP Core 向导示意图

（2）点击“Next”，“Finish”进入 Xilinx 时钟向导的建立窗口，如图 12-12 所示。ISE 默认选中 CLK0 和 LOCKED 这两个信号，用户根据自己需求添加输出时钟。在“Input Clock Frequency”输入栏中敲入输入时钟的频率或周期，单位分别是 MHz 和 ns，其余配置保留默认值。为了演示，这里添加了 CLKFX 信号，并设定输入时钟信号为单端信号，频率为 61.44MHz，其余选项保持默认值。

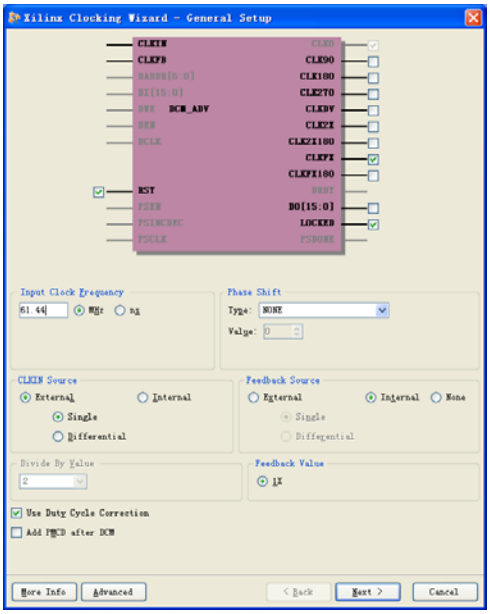


图 12-12　DCM 模块配置向导界面

（3）点击“Next”，进入时钟缓存窗口，如图 12-13 所示。默认配置为 DCM 输出添加全局时钟缓存以保证良好的时钟特性。如果设计全局时钟资源，用户亦可选择“Customize buffers”自行编辑输出缓存。一般选择默认配置即可。

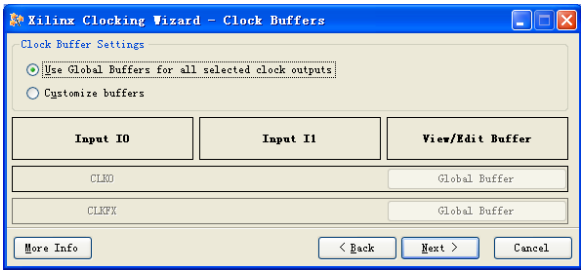


图 12-13　DCM 模块时钟缓存配置向导界面

（4）点击“Next”，进入时钟频率配置窗口，如图 12-14 所示。键入输出频率的数值，或者将手动计算的分频比输入。最后点击“Next”按键进入下一页，然后单击“Finish”按键完成 DCM 模块 IP Core 的全部配置。本例直接键入输出频率为 40.96MHz 即可。

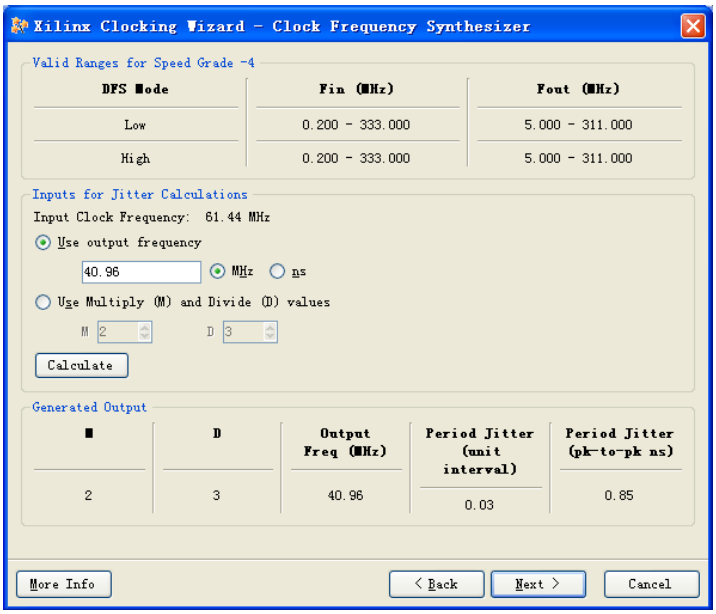


图 12-14　指定 DCM 模块的输出频率

（5）经过上述步骤，可在源文件进程中看到“*my_dcm.xaw*”文件，剩余的工作就是在设计中调用该 DCM IP Core。新建 HDL 源文件，其内容如下所示：

```
module dcm_top(
                CLKIN_IN,
                RST_IN,
                CLKFX_OUT,
                CLKIN_IBUFG_OUT,
                CLK0_OUT,
                LOCKED_OUT);

  input CLKIN_IN;
input RST_IN;
output CLKFX_OUT;
output CLKIN_IBUFG_OUT;
output CLK0_OUT;
output LOCKED_OUT;
```

```
mydcm dcm1(
.CLKIN_IN(CLKIN_IN),
.RST_IN(RST_IN),
.CLKFX_OUT(CLKFX_OUT),
.CLKIN_IBUFG_OUT(CLKIN_IBUFG_OUT),
.CLK0_OUT(CLK0_OUT),
.LOCKED_OUT(LOCKED_OUT)
);

endmodule
```

（6）上述代码经过综合后，其 RTL 级结构图如图 12-15 所示。

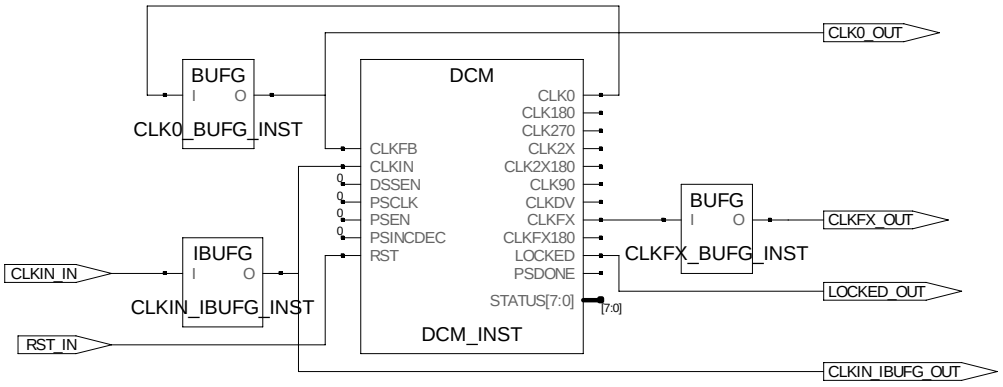


图 12-15　DCM 模块的 RTL 结构示意图

（7）DCM 模块的软件仿真。在 ISE 中新建仿真文件，并将其内容修改为如下代码：

```
module tb_mydcm;
    // Inputs
    reg CLKIN_IN;
    reg RST_IN;

    // Outputs
    wire CLKFX_OUT;
    wire CLKIN_IBUFG_OUT;
    wire CLK0_OUT;
    wire LOCKED_OUT;

    // Instantiate the Unit Under Test (UUT)
    cs_mydcm uut (
        .CLKIN_IN(CLKIN_IN),
        .RST_IN(RST_IN),
        .CLKFX_OUT(CLKFX_OUT),
        .CLKIN_IBUFG_OUT(CLKIN_IBUFG_OUT),
        .CLK0_OUT(CLK0_OUT),
        .LOCKED_OUT(LOCKED_OUT)
```

```
    );

    initial begin
        // Initialize Inputs
        CLKIN_IN = 0;
        RST_IN = 1;
        // Wait 100 ns for global reset to finish
        #100;
        RST_IN = 0;
    end

    //产生 61.44MHz 的时钟
    always # 8.1 CLKIN_IN = !CLKIN_IN;

endmodule
```

上述代码经过 ISE Simulater 仿真后，其局部仿真结果如图 12-16 所示。从中可以看出，当 LOCKED_OUT 信号变高时，DCM 模块稳定工作，输出时钟频率 CLKFX_OUT 为输入时钟 CLK_IN 频率的 2/3，完成了预定功能。需要注意的是，复位信号 RST_IN 是高有效。

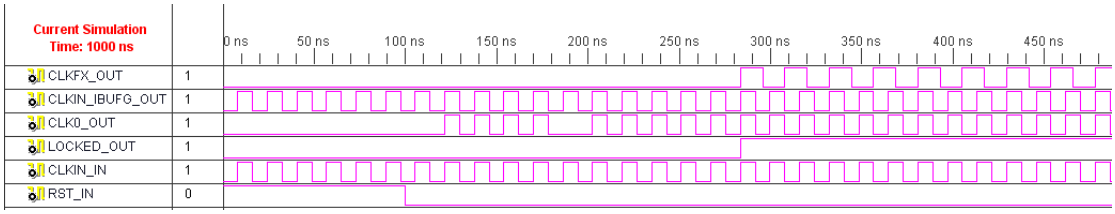


图 12-16　DCM 的仿真结果示意图

12.3 硬核乘法器的 Verilog HDL 调用

12.3.1 硬核乘法器结构说明

硬核乘法器是 Xilinx XtremeDSP 解决方案的核心组成部分，从而可以独立实现 500MHz 的性能，或在整合到一列中时实现 DSP 功能，支持 40 多个动态控制的操作模式，包括乘法器、乘法器-累加器、乘法器-加法器/减法器、三输入加法器、桶形移位器、宽总线多路复用器或宽计数器，其组成结构如图 12-17 所示。此外，级联硬核乘加器，无需使用 FPGA 逻辑和路由资源。

图 7-31 中的 OPMODE 是乘法器工作模式配置输入，可在 ISE 中通过软件方式指定。。不同系列芯片中乘法器的特点略有不同。

Spartan3E 系列最多可包含 104 个 18×18 的硬核乘法器，支持 18 位有符号或 17 位无符号应用，并支持级联，以实现更宽位的乘法操作。整体来讲，Spartan 3E 系列乘法器可用于实现简单的算法和算术功能，以及提供超过 3300 亿/秒的乘加运算功能。

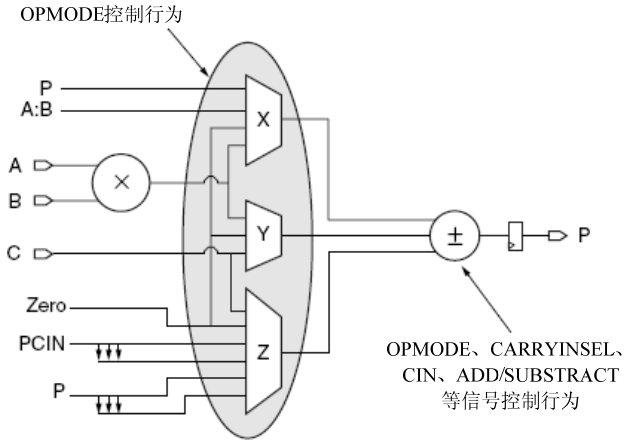


图 12-17　硬核乘加器的组成结构

12.3.2　基于 IP 核调用硬核乘法器

硬核乘法器 IP Core 可以完成有符号数以及无符号数的乘法，还能够完成输出数据的位宽截取，支持流水线操作，功能强大。其用户操作界面如图 12-18 所示，点击“Next”按键进入下一页，可以让用户选择是使用 FPGA 芯片上的硬乘法器（Use Mults），还是用 Slice 来构建乘法器（Use LUTs）。

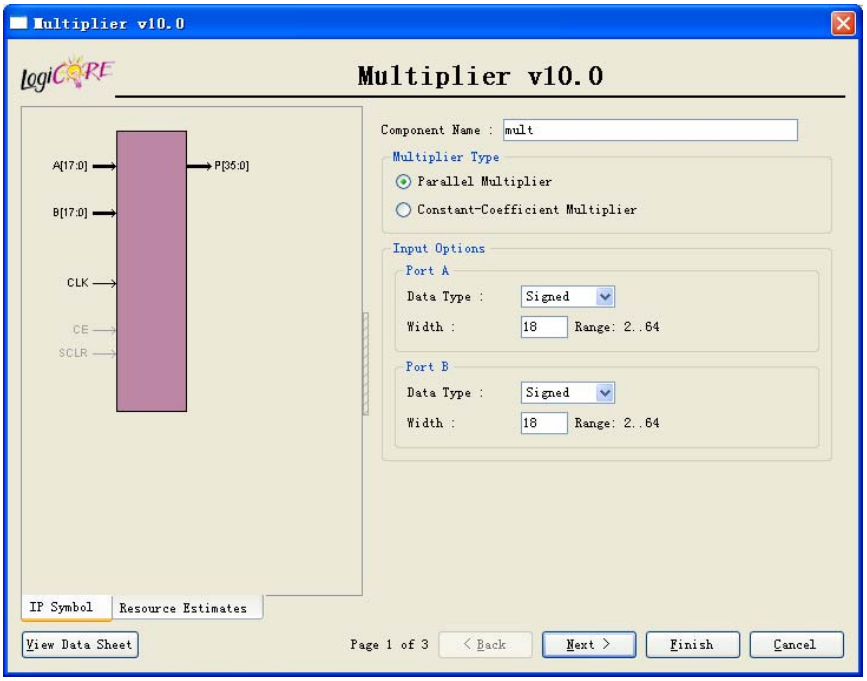


图 12-18　乘法器 IP core 用户操作界面

乘法器具有丰富的控制信号，说明如下:

【A】：乘法器的一个输入操作数，在使用时应当确定其位宽，其位宽可以在右边的端口位宽编辑框可以输入，并且可以有符号数、无符号数的两种选择。

【B】：乘法器的一个输入操作数，在使用时应当确定其位宽，其位宽可以在右边的端口位宽编辑框可以输入，而且可以与 A 口的输入操作数位宽不同。可以有符号数、无符号数的两种选择。（注：乘法器可以只接收 A 口的输入信号，而与一常数相乘，此常数可以为

固定的或可重导入的。对于可重导入数据（RCCM）情况中，此常数可以在 B 口重新导入而得到改变。）

【CLK】：乘法器的工作时钟。上升沿有效。

【CE】：输入信号，指示时钟是否有效。

【ND】：握手信号。

【ACLR】：异步清零信号。

【SCLR】：同步清零信号。

【LOADB】：当 RCCM 时才有效。当此信号为高时，B 端口新的输入可以重新写入计算模块的存储单元。

【SWAPB】：当 RCCM 时才有效。当此信号为高时，在计算模块的存储单元已存有的多个常数中进行选择。

【RDY】：输出的握手信号，当其变高时表明数据有效。

【RFD】：握手信号，在其变高后的下一个时钟上升沿数据有效。

【O】：异步输出信号，位宽根据输入信号的位宽而定。

【Q】：同步输出信号，位宽根据输入信号的位宽而定。

【LOAD_DONE】：当 RCCM 时才有效，指示重新导入数据的过程完成。

例 12-2：使用 IP Core 实例化一个 18 比特×18 比特的硬核乘法器，并完成相关软件测试和硬件仿真。

（1）新建工程，调用“Math Function→ Multiplies”目录下的“Multiplier 10.1”IP Core，并命名为 multiply。

（2）在第一页配置对话框中，将输入 A、B 端口数据类型配置为“Signed”、位宽为 18；在第二页的“Multiplier Construction”下拉框中选择“Use Mults”，分别如图 12-19、12-20 所示。其余参数保持默认值。单击“Finish”按键完成配置，生成硬核乘法器 IP Core。

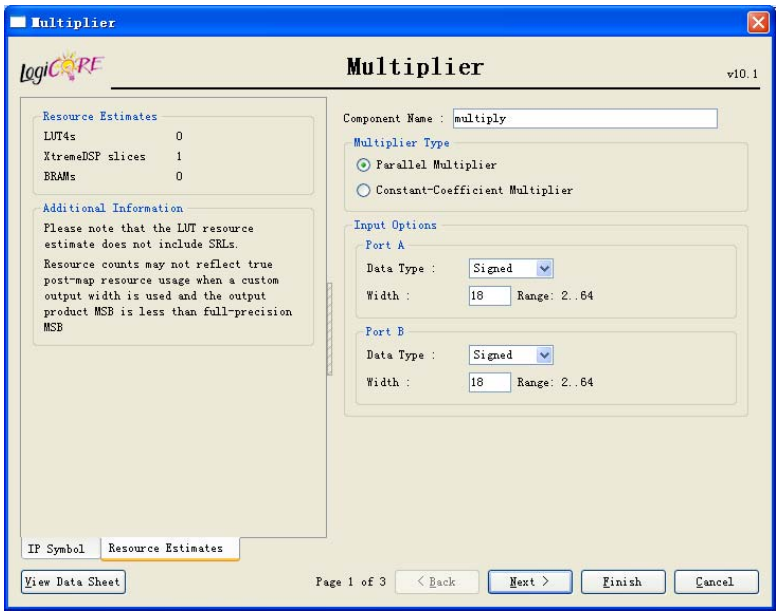


图 12-19　端口数据类型配置

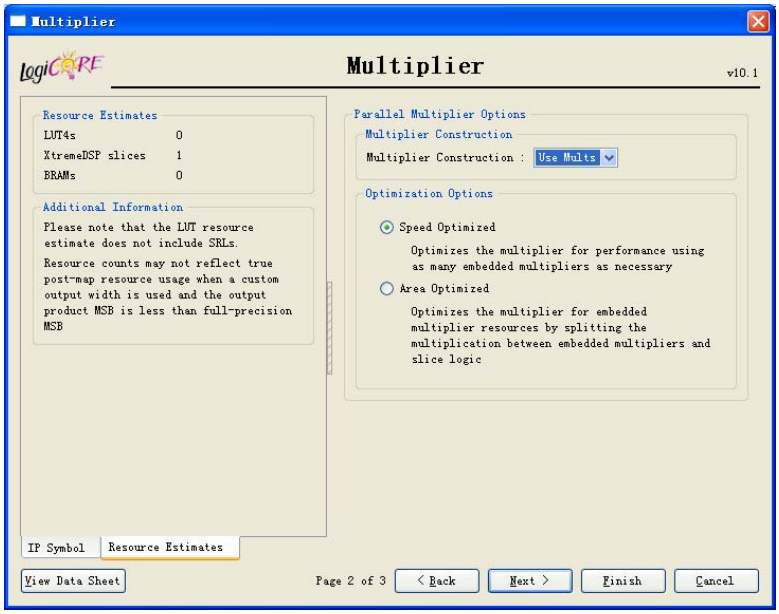


图 12-20　乘法器实现类型选择界面

（3）添加 Verilog 源文件，其内容如下所列：

```
module multiply1 (clk, a, b, q);
   input              clk;
   input [17 : 0] a, b;
   output [33 : 0] q;

   //调用硬核乘法器 IP Core
   multiply multiply1(.clk(clk),.a(a), .b(b), .p(q));

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 12-21 所示。从中可以看出，硬核乘法器只要一个时钟即可计算出结果。当然，用户也可以在 IP Core 配置中添加流水线，可选有 1~32 级，最佳选择为 3 级流水线。读者需要注意的是，添加 N 级流水线，计算结果则需要往后延迟 N 拍时钟，但可极大地提高吞吐量。

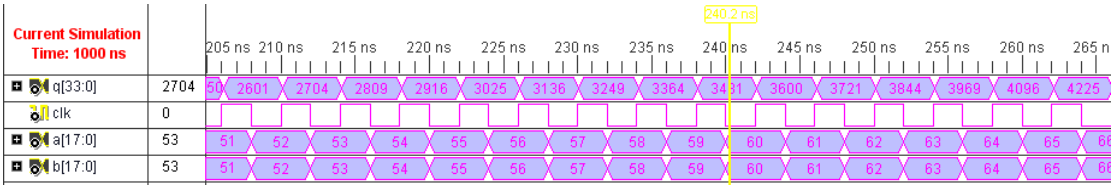


图 12-21　乘法器 IP core 的仿真波形

12.4 块 RAM 的 Verilog HDL 调用

12.4.1　块 RAM 结构说明

1．块存储器简介

Xilinx 公司提供了大量的存储器资源，包括了内嵌的块存储器、分布式存储器以及 16 位的移位寄存器。利用这些资源可以生成深度、位宽可配置的 RAM、ROM、FIFO 以及移

位寄存器等存储逻辑。其中，块存储器是硬件存储器，不占用任何逻辑资源，其余两类都是 Xilinx 专有的存储结构，由 FPGA 芯片的查找表和触发器资源构建的，每个查找表可构成 16×1 位的分布式存储器或移位寄存器。一般来讲，块存储器是宝贵的资源，通常用于大数据量的应用场合，而其余两类用于小数据量环境。

在 Xilinx FPGA 中，块 RAM 是按照列来排列的，这样保证了每个 CLB 单元周围都有比较接近的块 RAM 用于存储和交换数据。与块 RAM 接近的是硬核乘加单元，这样不仅有利于提高乘法的运算速度，还能形成微处理器的雏形，在数字信号处理领域非常实用。例如，在 Spartan 3E 系列芯片中，块 RAM 分布于整个芯片的边缘，其外部一般有两列 CLB，如图 12-22 所示，可直接对输入数据进行大规模缓存以及数据同步操作，便于实现各种逻辑操作。

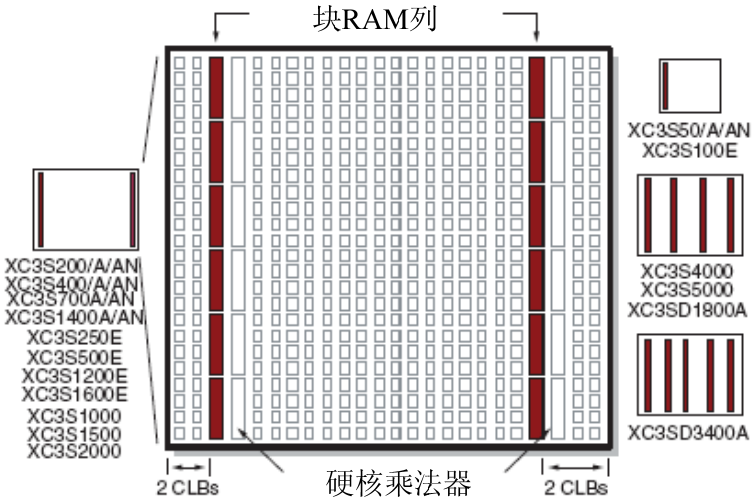


图 12-22　Spartan3E 系统芯片中块 RAM 的分布图

2．块 RAM 的应用类型

块 RAM 几乎是 FPGA 器件中除了逻辑资源之外用得最多的功能块，Xilinx 的主流 FPGA 芯片内部都集成了数量不等的块 RAM 硬核资源，速度可以达到数百兆赫兹，不会占用额外的 CLB 资源，而且可以在 ISE 环境的 IP 核生成器中灵活地对 RAM 进行配置，构成单端口 RAM、简单双口 RAM、真正双口 RAM、ROM（在 RAM 中存入初值）和 FIFO 等应用模式，如图 12-23 所示。同时，还可以将多个块 RAM 通过同步端口连接起来构成容量更大的块 RAM。

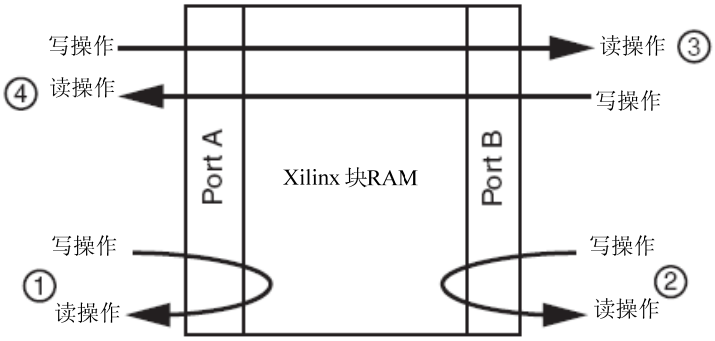


图 12-23　块 RAM 组合操作示意图

（1）单端口 RAM 模式

单端口 RAM 的模型如图 12-24 所示，只有一个时钟源 CLK，WE 为写使能信号，EN 为单口 RAM 使能信号，SSR 为清零信号，ADDR 为地址信号，DI 和 DO 分别为写入和读

出数据信号。

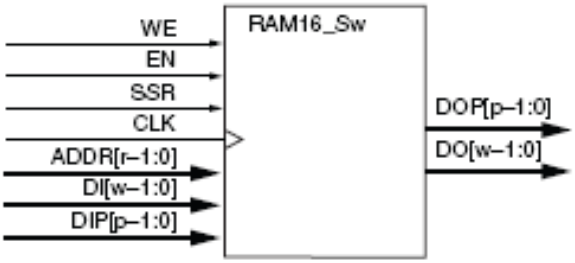


图 12-24　Xilinx 单端块 RAM 的示意模型

单端口 RAM 模式支持非同时的读写操作。同时每个块 RAM 可以被分为两部分，分别实现两个独立的单端口 RAM。需要注意的是，当要实现两个独立的单端口 RAM 模块时，首先要保证每个模块所占用的存储空间小于块 RAM 存储空间的 1/2。在单端口 RAM 配置中，输出只在 read-during-write 模式有效，即只有在写操作有效时，写入到 RAM 的数据才能被读出。当输出寄存器被旁路时，新数据在其被写入时的时钟上升沿有效。

（2）简单的双端口 RAM

简单双端口 RAM 模型如图 12-25 所示，图中上边的端口只写，下边的端口只读，因此这种 RAM 也被称为伪双端口 RAM（Pseudo Dual Port RAM）。这种简单双端口 RAM 模式也支持同时的读写操作。

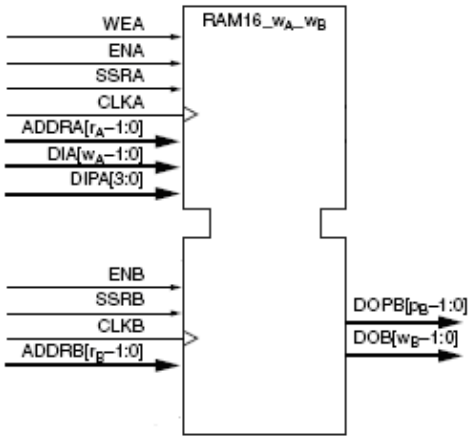


图 12-25　Xilinx 简单双端口块 RAM 的示意模型

块 RAM 支持不同的端口宽度设置，允许读端口宽度与写端口宽度不同。这一特性有着广泛地应用，例如：不同总线宽度的并串转换器等。在简单双端口 RAM 模式中，块 RAM 具有一个写使能信号 wren 和一个读使能信号 rden，当 rden 为高电平时，读操作有效。当读使能信号无效时，当前数据被保存在输出端口。当读操作和写操作同时对同一个地址单元时，简单双口 RAM 的输出或者是不确定值，或者是存储在此地址单元的原来的数据。

（3）真正双端口 RAM 模式

真正双端口 RAM 模型如图 12-26 所示，图中上边的端口 A 和下边的端口 B 都支持读写操作，WEA、WEB 信号为高时进行写操作，低为读操作。同时它支持两个端口读写操作的任何组合：两个同时读操作、两个端口同时写操作或者在两个不同的时钟下一个端口执行写操作，另一个端口执行读操作。

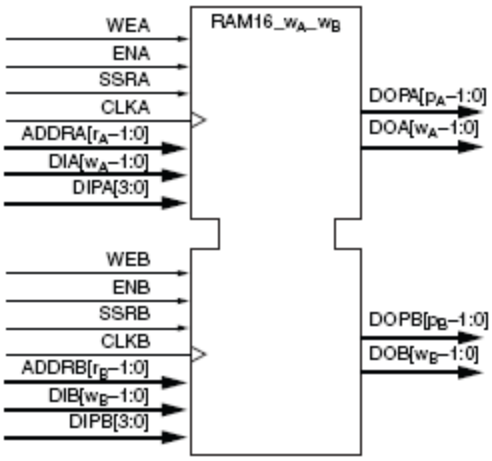


图 12-26　Xilinx 真正双端口块 RAM 的示意模型

真正双端口 RAM 模式在很多应用中可以增加存储带宽。例如，在包含嵌入式处理器 MiroBlaze 和 DMA 控制器系统中，采用真正双端口 RAM 模式会很方便；相反，如果在这样的一个系统中，采用简单双端口 RAM 模式，当处理器和 DMA 控制器同时访问 RAM 时，就会出现问题。真正双端口 RAM 模式支持处理器和 DMA 控制器同时访问，这个特性避免了采用仲裁的麻烦，同时极大地提高了系统的带宽。

一般来讲，在单个块 RAM 实现的真正双端口 RAM 模式中，能达到的最宽数据位为 36 比特*512，但可以采用级联多个块 RAM 的方式实现更宽数据位的双端口 RAM。当两个端口同时向同一个地址单元写入数据时，写冲突将会发生，这样存入该地址单元的信息将是未知的。要实现有效地向同一个地址单元写入数据，A 端口和 B 端口时钟上升沿的到来之间必须满足一个最小写周期时间间隔。因为在写时钟的下降沿，数据被写入块 RAM 中，所以 A 端口时钟的上升沿要比 B 端口时钟的上升沿晚到来 1/2 个最小写时钟周期，如果不满足这个时间要求，则存入此地址单元的数据无效。

（4）ROM 模式

块 RAM 还可以配置成 ROM，可以使用存储器初始化文件（.coe）对 ROM 进行初始化，在上电后使其内部的内容保持不变，即实现了 ROM 功能。

（5）FIFO 模式

FIFO 即先入先出，其模型如图 12-27 所示。在 FIFO 具体实现时，数据存储的部分是采用简单双端口模式操作的，一个端口只写数据而另一个端口只读数据，另外在 RAM（块 RAM 和分布式 RAM）周围加一些控制电路来输出指示信息。FIFO 最重要的特征是具备“满（FULL）”和“空（EMPTY）”的指示信号，当 FULL 信号有效时（一般为高电平），就不能再往 FIFO 中写入数据，否则会造成数据丢失；当 EMPTY 信号有效时（一般为高电平），就不能再从 FIFO 中读取数据，此时输出端口处于高阻态。

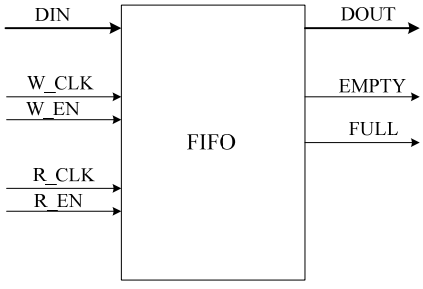


图 12-27　Xilinx FIFO 模块的示意模型

12.4.3　基于 IP 核调用块 RAM 单元

例 12-3：利用块 RAM 实现 a、b 两路数据的延迟，其中 a、b 两路数据的位宽都为 32 比特，速率都为 61.44Mbps，要求 a 路延迟 16 个数据时钟周期，b 路延迟 8 个数据时钟周期。

（1） 新建工程，添加 IP Core 类型源文件，并命名为 bram_16，并选择 IP 核列表中“Memories& Storage&Elements → RAMS&ROMS”目录下的“Block Memory Generator v2.7”，并选择“Ture Dual Port RAM”，如图 7-19 所示。点击“Next”按键，进入下一页。

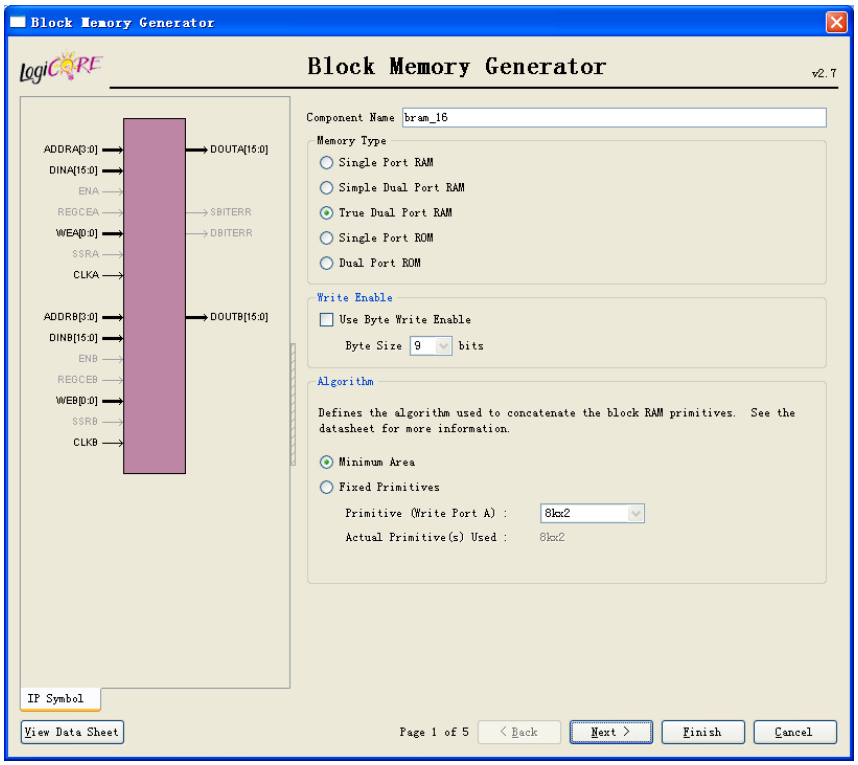


图 7-19　块存储器的 IP Core

（2）接下来的两页配置 A、B 端口参数，其配置参数一样：32 比特为宽、64 深度、具备读、写功能，其中 A 端口的配置参数如图 7-20 所示。点击“Next”按键进入下一页。

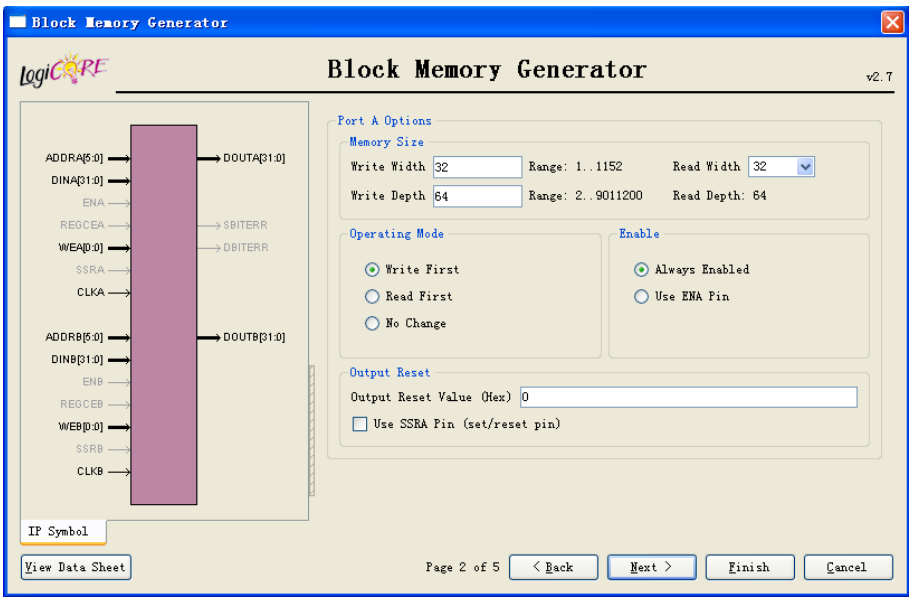


图 7-20　A 端口配置参数示意图

（3）此后所有页的配置参数都采用缺省值，在第 5 页单击“Finish”按键，生成块 RAM IP Core。用户可在源文件管理区看到文件。用鼠标选中该 IP Core，在过程管理区双击“Core Generator”栏目下的“View HDL Functional Model”，可查看其对应的 HDL 源码。

（4）新建 Verilog 源文件，并输入以下内容：

```
module bram_delay(clk_122p88MHz, a, b, a_delay, b_delay);
        input              clk_122p88MHz;
        input   [31:0]   a;
        input   [31:0]   b;
        output [31:0]   a_delay;
        output [31:0]   b_delay;

        reg      [31:0]   a_delay;
        reg      [31:0]   b_delay;
        wire     [5:0]    addra, addrb;
        wire     [31:0]   douta, doutb;
        reg      [5:0]    addra1 = 0;
        reg      [5:0]    addra2 = 0;
        reg      [5:0]    addrb1 = 32;
        reg      [5:0]    addrb2 = 32;
        reg                wea = 0;
        reg                web = 0;
        reg                flag = 0;

        always @(posedge clk_122p88MHz) begin
            flag <= !flag;
            if(flag == 1'b1) begin
                a_delay <= a_delay;
                b_delay <= b_delay;
                wea <= 1'b1;
                web <= 1'b1;
                addra2 <= addra2;
                addrb2 <= addrb2;
                if(addra1 == 31)
                    addra1 <= 0;
                else
                    addra1 <= addra1 + 1'b1;
                if(addrb1 == 63)
                    addrb1 <= 32;
                else
                    addrb1 <= addrb1 + 1'b1;
            end
            else begin
                wea <= 1'b0;
```

```
                web <= 1'b0;
                a_delay <= douta;
                b_delay <= doutb;
                addra1 <= addra1;
                addrb1 <= addrb1;
                if(addra1 <=15)
                    addra2 <= addra1 + 16;
                else
                    addra2 <= addra1 - 16;
                if(addrb1 <=39)
                    addrb2 <= addrb1 + 24;
                else
                    addrb2 <= addrb1 - 8;
            end
        end

        assign addra = !flag ? addra1 : addra2;
        assign addrb = !flag ? addrb1 : addrb2;

    bram_16 bram_16(
        .clka(clk_122p88MHz),
        .dina(a),
        .addra(addra),
        .wea(wea),
        .douta(douta),
        .clkb(clk_122p88MHz),
        .dinb(b),
        .addrb(addrb),
        .web(wea),
        .doutb(doutb)
    );

endmodule
```

（5）添加 HDL 仿真文件（Verilog Test Fixture），相关的测试代码如下：

```
module tb_bram_delay;

    // Inputs
    reg clk_122p88MHz;
    reg [31:0] a;
    reg [31:0] b;

    // Outputs
    wire [31:0] a_delay;
    wire [31:0] b_delay;
```

```
// Instantiate the Unit Under Test (UUT)
bram_delay uut (
        .clk_122p88MHz(clk_122p88MHz),
        .a(a),
        .b(b),
        .a_delay(a_delay),
        .b_delay(b_delay)
);

initial begin
        // Initialize Inputs
        clk_122p88MHz = 0;
        a = 0;
        b = 255;
end

always #4 clk_122p88MHz = !clk_122p88MHz;
always #16 a = a + 1;
always #16 b = b - 1;

endmodule
```

将源文件工程区切换到“Behavioral Simulation”，利用 ISE Simulator 完成代码测试，得到的测试结果如图 7-21 所示。可以看出，a 路数据延迟了 16 个数据周期，b 路数据延迟了 8 个时钟周期，完成了预期功能。

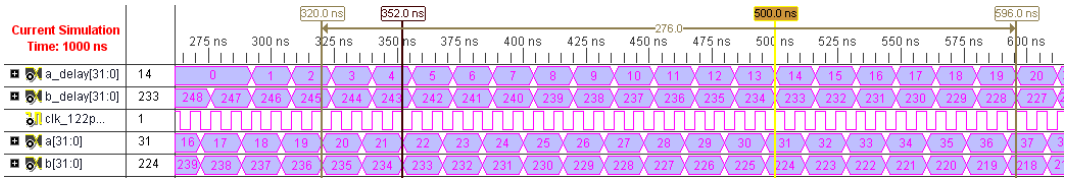


图 7-21　数据延迟模块仿真结果示意图

2．ROM 开发实例

对于 ROM 模块，主要是生成相应的.coe 文件。下面以一个实例介绍如何借助 MATLAB 生成 ROM 的.coe 文件，并在设计中如何通过 ROM 模块完成逻辑功能设计。

例 12-4：利用块 ROM 实现一个两输入的 4 比特乘法器。

实现思路如下：首先，例化一个块 ROM，其地址位宽为 8 比特，存储的数据位宽为 8 比特。然后将地址的高 4 位作为乘法器的一个输入，低 4 位作为另一个输入，而将对应的地址中存放其地址高 4 比特和低 4 比特所代表数据的乘积，这样只要通过地址查表，就可以实现乘法的功能。

（1）利用 MATLAB 依次计算出 8 比特地址线按照高 4 位和低 4 位区分后的乘积值，量化为 8 比特后将其写入文本文件。

```
%%在区间[0,6.28]之间等间隔地取 1024 个点
x= linspace(0,6.28,1024);
```

```
y1=sin(x); %%计算相应的正余弦值
%%由于正余弦波形的值在[0,1]之间，需要量化成 16 比特，先将数值放大
y1=y1*32678;
%%再将放大的浮点值量化，并写到存放在 C 盘的文本中
fid = fopen('c:/sin.txt', 'wt');
fprintf(fid, '%16.0f\n', y1);   %%在写文件的时候量化成 16 比特
fclose(fid)
```

（2）生成 coe 文件。在 C 盘根目录下，将 sin_coe.txt 文件的后缀改成.coe，打开文件，把每一行之间的空格用文本的替换功能换成逗号“,”，并在最后一行添加一个分号“;”。最后在文件的最开始添加下面两行：

```
memory_initialization_radix=10;
memory_initialization_vector =
```

然后保存文件退出。

（3）将 coe 文件加载到 BLOCKROM 所生成的 ROM 中。调用“Memories & Storage Elements → RAMs & ROMS → Block Memory Generator v2.7”IP Core，命名为 rom16，在第一页选择 Single Port Rom，在第二页选择位宽为 8、深度为 256，在第三页下载 coe 文件，如图 12-34 所示，然后双击“Finish”，完成 IP core 的生成。如果 coe 文件生成的不对，图 12-34 中用椭圆标志之处是红色的，coe 文件错误的类型主要有数据基数不对和数据的长度不对这两类。设计人员可点击“Show”按键，查看初始化数据，如图 12-35 所示，确认无误后，点击“Finish”按键。

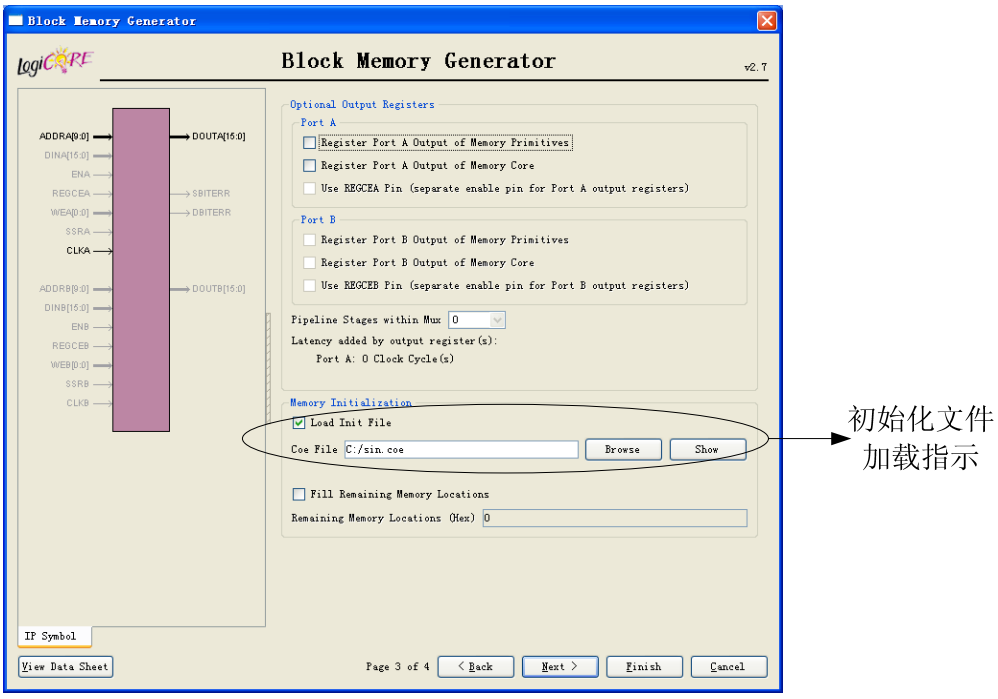


图 12-34　块 ROM 加载 coe 文件的用户配置界面

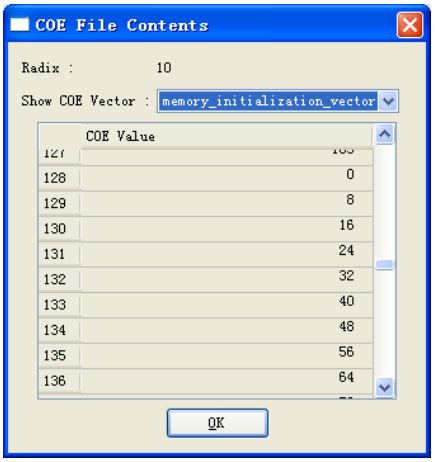


图 12-35　块 ROM 初始化数据查看页面

（4）在工程中新建源文件，命名为 mult_rom_demo 并添加下面的代码：

```
module mult_rom_demo(
    clk, dina, dinb, doutq
    );
     input          clk;
     input  [3:0] dina;
     input  [3:0] dinb;
    output  [7:0] doutq;

   rom16 inst_rom16(
        .clka(clk),
        .addra({dina, dinb}),
        .douta(doutq)
        );

endmodule
```

为了测试 mult_rom_demo 模块，还需要一个完整的测试文件 tb_mult_rom，其代码如下所列：

```
module tb_mult_rom_demo;

    // Inputs
    reg clk;
    reg [3:0] dina;
    reg [3:0] dinb;

    // Outputs
    wire [7:0] doutq;

    // Instantiate the Unit Under Test (UUT)
    mult_rom_demo uut (
        .clk(clk),
        .dina(dina),
```

```
        .dinb(dinb),
        .doutq(doutq)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        dina = 0;
        dinb = 0;
    end

    always #5 clk = ~clk;
    always #10 begin
        dina = dina + 1;
        dinb = dinb + 3;
    end

endmodule
```

在 ISE Simulator 中的仿真结果如图 12-36 所示，通过比较 COE 文件，可以看出功能仿真是正确的。

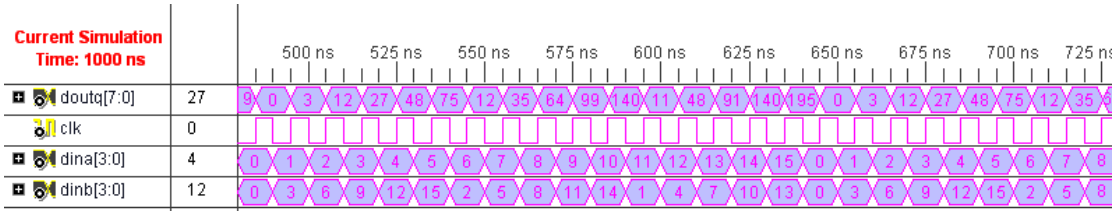


图 12-36　正弦波发生器仿真结果示意图

在实际开发中，这种查找表的形式来实现复杂逻辑的思路是经常用到的，不仅可以实现例 11-24 所示的乘法器，还包括除法、三角函数以及开根号等超越函数的运算。

12.5 本章小结

Xilinx 主流 FPGA 芯片内嵌了大量的底层硬核单元，可用于完成时钟管理、数据存储、数字信号处理、高速连接以及嵌入式处理。本章介绍了 Xilinx FPGA 中常用的硬核组件，包括差分时钟、DCM 模块、硬核乘法器以及块 RAM 的实例，并给出在 Spartan 3E 系列 XC3S500E 平台上的实现结果。通过这些实例可将基于 ISE 逻辑设计的相关内容有机衔接起来，便于读者在短时间内了解并掌握基于 FPGA 开发方法。

12.6 思考题

1．什么是硬核模块，在 Xilinx FPGA 中有哪些硬核模块？
2．什么是差分管脚？如果通过 Verilog HDL 程序调用？
3．DCM 模块的全称是什么？具有什么功能？
4．硬核乘法器和软核乘法器有什么区别？如何通过 Verilog HDL 程序调用？
5．块 RAM 的最大特点是什么？基于块 RAM 可以完成哪些主流操作？

第 13 章　串口接口的 Verilog HDL 设计

RS-232 接口（又称 EIA RS-232-C）是目前最常用的一种串行通讯接口。它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标准。本章主要介绍 RS232 接口的原理介绍、实现以及完整的软、硬件调试流程。

13.1　串口以及串口通信协议简介

13.1.1　串口接口

串口即串行数据接口，是一种常用的数据接口。在 PC 一般都有两个串行口（COM1 和 COM2），其形状如图 13-1 所示，是典型的 9 针 D 形接口，也称为 DB9。由于串口多采用 RS-232-C 传输协议并长达数十年，因此也常被称为 RS-232 接口。

图 13-1　串口的物理形状示意图

串口主要用于网管控制或主业务数据的传输，支持数据的双向传输，速率从 9600-115200bps，即可以完成和 PC 的通信，也可以完成与带有标准串口的外设相连，其典型的连接方式如图 13-2 所示。其中串口接口分为带插孔和带插针的两种，其中插针端称为 DCE，插孔端称为 DTE。

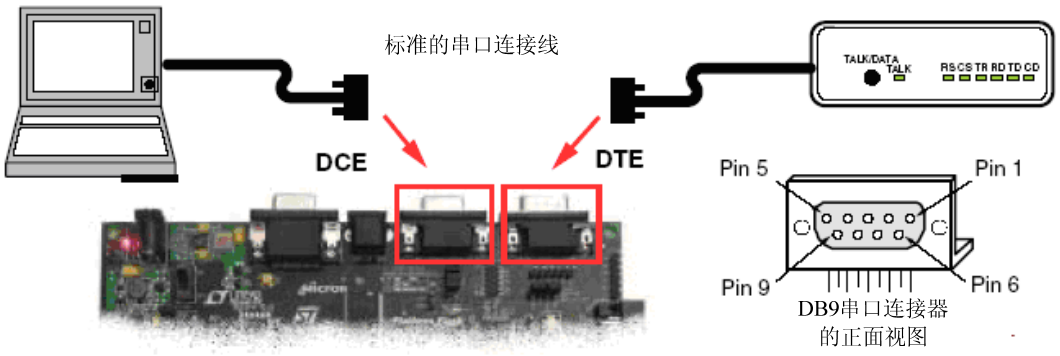


图 13-2　串口连接的示意图

13.1.2　RS-232 通信协议

RS-232 全名是“数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准”，该标准规定采用一 25 个脚的串口连接器（DB25），不仅对连接器的每个

引脚的信号内容加以规定，还规定了各种信号的电平。

标准的 RS-232-C 协议具有 25 根信号线，其中有 4 根为数据线、11 根控制线、3 根定时线和 7 根备用线。通常情况下，使用其中的 9 根线就可以实现 RS-232 串口通信。

通常 DB-25 也可以像 DB-9 那样使用，但是它们的针号和针孔之间的连接关系是不一样的，下面给出常用的 25 针 DB-25 和 DB-9 的对照表以及管脚说明，如表 13-1 所示。

表 13-1　　RS-232-C 接口引脚定义

DB25	DB9 芯	信号源	信号名	信号功能描述
2	3	DCE	TXD	发送数据，终端通过此将信号发给调制器
3	2	DTE	RXD	接收数据，终端通过此从调制器接收数据
4	7	DCE	RTS	请求发送，当终端需要发送数据时，使能该信号，控制调制器进入发送状态
5	8	DTE	CTS	允许发送，当调制器准备好接收数据时，使能该信号，通知终端开始发送数据
6	6	DTE	DSR	数据就绪状态，当其状态有效时，表明调制器处于可用状态
7	5	GND	GND	信号地，信号地是所有信号的参考电平
8	1	DTE	DCD	载波检测，当信号有效时，表明调制器已经接通了通信链路，终端可以准备接收数据
20	4	DCE	DTR	数据终端准备，当其信号有效时，表明数据终端可以使用
22	9	DTE	RI	响铃指示，当本地调制器收到交换台发送的振铃呼叫信号时，使能该信号，通知终端

最为简单且常用的 RS-232-C 连接方法就是三线连接法，即地、接收数据和发送数据三脚相连。这是因为收、发数据是时分的，二者不会同时传输。对于 DB-9 和 DB-25，常用的 3 线连接法的原则如表 13-2 所示。

表 13-2　　DB-9、DB-25 常用的 3 线连接法

DB9~DB9		DB25~DB-25		DB9~DB25	
3 线连接管脚名					
2	3	3	2	2	2
3	2	2	3	3	3
5	5	7	7	5	7

RS-232-C 的串行总线在空闲的时候保持为逻辑“1”状态，即串行连接线上的电平为-3~15V。当需要传送一个字符时，首先会发送一个逻辑为“0”的起始位，表示开始发送数据；之后就逐个发送数据位、奇偶校验位和停止位（逻辑“1”），每一次传输 1 个字符（可以设成 5~8 个比特）。由于任意两个字符对应瞬间的时间间隔是可变的，因此也被称为异步格式。典型的传输时序如图 13-3 所示。

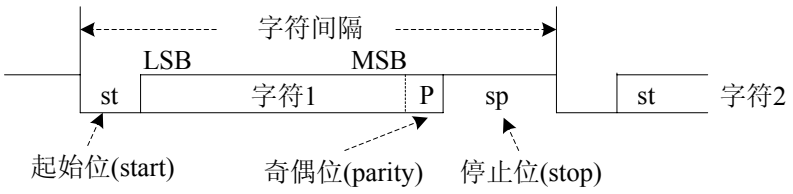


图 13-3　RS232 协议的传输时序

在 RS-232-C 协议中，可以设置数据的传输速率。除此之外，字符的位宽、奇偶校验位、停止位都可以被设置。字符可以被设置成 5~8 比特，奇偶校验位可以被去除；停止位可以设置成 1 位、2 位甚至 1.5 位。每秒传输的比特数也被称为二进制波特率，即位周期的倒数，简称为比特率，用于衡量 RS-232 的传输速率。RS-232-C 有一系列波特率标准：110bps、300bps、600bps、1200bps、1800bps、2400 bps、4800bps、9600bps、14.4kbps、19.2kbps、28.8kbps、33.6kbps 和 56kbps 等。有两点注意的是：首先，在设置波特率时，必须同时通知通信双方；其次，波特率的计算包括了起始位、字符、校验位、停止位在内所有的比特，而不是仅针对字符。

目前，大部分处理器都集成了支持 RS-232-C 的通用异步收发器（Universal Asynchronous Receiver/Transmitter，UART），辅助处理器和串行设备之间通信，设计人员只需要对其进行配置即可完成下列工作：

- 完成处理器内部的并行数据到串行数据的转化以及外部串行数据到并行数据的转化；
- 完成输入数据的奇偶校验，以及在输出数据中插入奇偶校验比特；
- 完成数据传输和停止位的检测，并从中提取符号数据；
- 完成外部设备串口设备的管理和响应。

13.2 串口通信控制器的 Verilog HDL 实现

本设计采用分层设计思想，主要由顶层模块、波特率发生器、接收模块和发送模块这 4 个模块组成，强调功能划分明确，便于系统设计和调试。

13.2.1　系统功能说明

本系统要求在 Xilinx Spartan 3E Starter 开发板上实现波特率为 9600，停止位为 1 比特、不带校验位并且具备复位功能的串口通信控制器，并要求和 PC 机通过超级终端完成双向通信。不仅要求将板极发送数据显示在 PC 机的超级终端上，还要求用 PC 发送数据的 ASCII 码来驱动电路板的 8 个 LED 灯。为了便于测试，要求当按下开发板上的 button_s 时，板级发送的数值恢复到 48，对应着字符 0（字符 0 的 ASCII 码为 48），然后按下一次 button_n，发送数据加 1。

因此，在系统实现时，不仅要包括完整的串口通信模块，还需要有相应的按键处理模块。这是因为按键按下的持续时间很长，对发送模块来讲，是一个电平信号，而不是脉冲信号，因此需要利用 11.1.3 节的同步整形电路，将其处理成单时钟周期宽度的脉冲信号。

13.2.2　顶层模块的组成结构和 Verilog HDL 实现

RS-232 通信控制器的顶层模块（uart_top.v）主要由三个子模块实现，包括波特率发生器、

接收模块以及发送模块，其结构如图 13-4 所示。

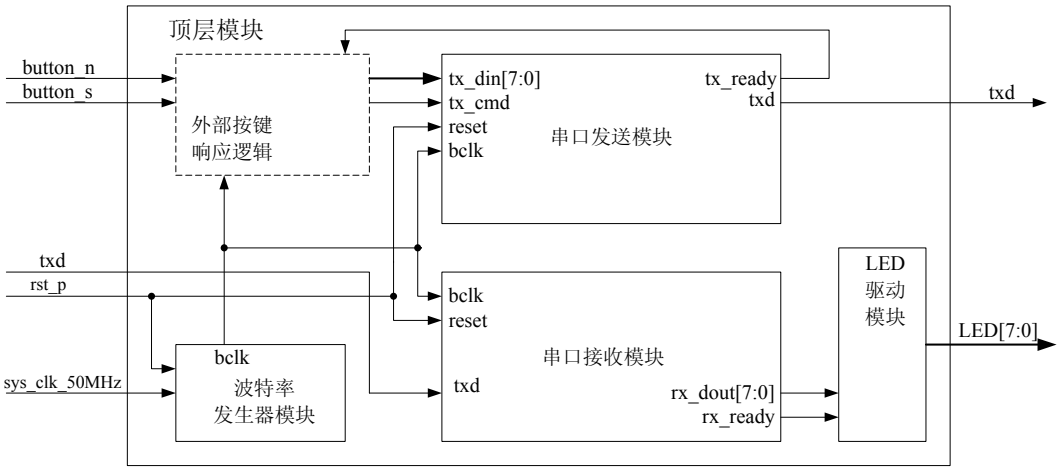


图 13-4　通信控制器顶层模块的子模块

顶层模块作为设计的主干，用于例化各个模块以及响应外部按键，不包含串口收发模块的处理代码。串口通信控制器的顶层模块 uart_top 的代码如下所列。

```
module uart_top(
    sys_clk_50MHz, rst_p, txd, rxd, LED, button_n, button_s
    );
     //系统工作时钟
     input    sys_clk_50MHz;
     //系统复位信号，高有效
     input    rst_p;
     //RS232 接口接收到数据，是设计的输入数据
     input    rxd;
     //用户按键，button_s 恢复初始值，button_n 将发送数据加 1，高有效
     input    button_n, button_s;
     // RS232 接口的发送数据，是设计的输出数据
     output txd;
     //8 个 LED 小灯的驱动信号
     output [7:0] LED;

     //将输出的 LED 小灯驱动信号定义为寄存器类型
     reg      [7:0] LED;

     wire     bclk;
     wire     [7:0] rx_dout;
     wire     rx_ready;
     reg      [7:0] din;
     wire     tx_ready;
```

```
 reg       tx_cmd;

 //例化波特率发生器模块
 baud_gen inst_baud_gen(
       .clk_50MHz(sys_clk_50MHz),
       .rst_p(rst_p),
       .bclk(bclk)
       );

//1.用于从用户操作按键的电平信号中提取响应脉冲
//2.从串口接收模块的接收完成电平信号中提取接收数据指示脉冲
reg    [2:0] bv1, bv2;
wire            bv1_posedge, bv2_posedge;

always @(posedge bclk) begin
      bv1 <= {bv1[1:0], button_n};
      bv2 <= {bv2[1:0], rx_ready};
end
//完成同户按键脉冲的提取，其原理和 11.1.3 节的同步整形电路一致
assign bv1_posedge = (!bv1[2]) & bv1[1];

//给出接收完成响应脉冲的提取，其原理和 11.1.3 节的同步整形电路一致
assign bv2_posedge = (!bv2[2]) & bv2[1];

always @(posedge bclk) begin
      if(button_s == 1'b1) begin
                  din    <= 48;
                  tx_cmd <= 1'b0;
      end
      else begin
            //用户每按下一次，发送数据加 1
            if(bv1_posedge == 1'b1) begin
                  din       <= din + 1;
                  tx_cmd <= 1'b1 & tx_ready;
            end
            else begin
                  din       <= din;
                  tx_cmd <= 1'b0;
            end
   end
```

```
    end

    always @(posedge bclk) begin
        //每接收一个字节数据后，用其驱动 8 个 LED 灯
        if (bv2_posedge == 1'b1) begin
            LED <= rx_dout;
        end
    end

    //例化串口发送模块
    uart_tx inst_uart_tx(
            .bclk(bclk),
            .reset(rst_p),
            .tx_din(din),
            .tx_cmd(tx_cmd),
            .tx_ready(tx_ready),
            .txd(txd)
        );

    //例化串口接收模块
    uart_rx inst_uart_rx(
            .bclk(bclk),
            .reset(rst_p),
            .rxd(rxd),
            .rx_ready(rx_ready),
            .rx_dout(rx_dout)
          );

endmodule
```

13.2.3 波特率发生器模块的 Verilog HDL 实现

波特率发生器实际上是一个分频器，从给定的系统时钟频率得到要求的波特率。一般来讲，为了提高系统的容错性处理，要求波特率发生器的输出时钟为实际串口数据波特率的 N 倍，N 可以取值为 8、16、32、64 等。在本设计中，取 N 为 16，因此波特率发生器的输出信号频率应改为 9600*16=153.6kbps。

由于串口速率较低，其 16 倍频率值也不高，因此在设计中，可以不要求波特率发生器输出信号的占空比为 50%，在本例中，其占空比为 1：325。设计中的波特率发生器的代码（baud_gen.v）如下所列。

```
module baud_gen(
clk_50MHz, rst_p, bclk
```

```
);
//输入的系统时钟，将其分频为 153.60kbps
 input       clk_50MHz;
//复位信号，低有效
 input       rst_p;
 // 分频输出信号，其大小为 9600*16=153.60kbps
 output      bclk;
 reg          bclk;

 reg          [8:0] cnt;
 //利用同步计数器完成时钟分频
 always @(posedge clk_50MHz) begin
     if(rst_p) begin
          cnt <= 0;
           bclk <= 0;
       end
       else begin
          // 50_000_000 /153600 = 325.5
          if(cnt > 324) begin
               cnt <= 0;
               bclk <= 1;
           end
           else begin
               cnt <= cnt + 1;
               bclk <= 0;
           end
       end
 end

endmodule
```

为了测试上述程序，编写了 tb_uart_tx 的测试代码用于检查发送逻辑有无错误，其内容如下所列：

```
module tb_buad_gen;
//定义输入变量
reg clk_50MHz;
reg rst_p;

//定义输出变量
wire bclk;
```

```
// 例化被测试模块(UUT)
baud_gen uut (
        .clk_50MHz(clk_50MHz),
        .rst_p(rst_p),
        .bclk(bclk)
);

initial begin
        // 初始化输入信号值
        clk_50MHz = 0;
        rst_p = 1;
        // 使得全局复位达到 100 个仿真时间单位
        #100;
          rst_p = 0;
end

        //模拟产生输入时钟
always #10 clk_50MHz = ~clk_50MHz;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 13-5 所示。可以看出，bclk 信号仅在计数器 cnt 的值达到 325 后变高一个时钟周期，在其余的 0~324 个数值中都为低电平，达到了设计目标，可产生 153.60kHz 的工作时钟。

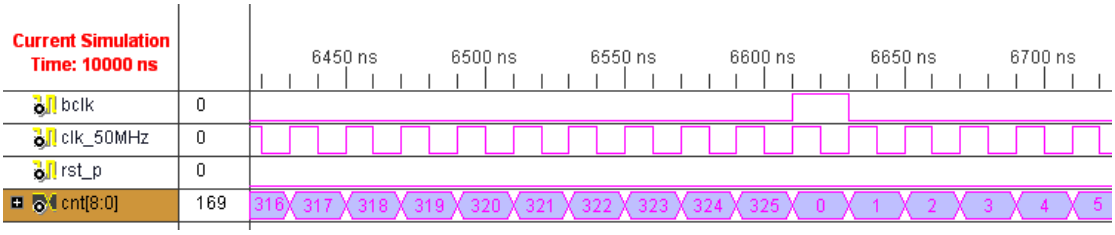


图 13-5　波特率发生器模块的仿真结果

13.2.4　发送模块的 Verilog HDL 实现

由于波特率发生器产生的时钟信号 bclk 的频率为 9600Hz 的 16 倍，因此在发送器中，每 16 个 bclk 周期发送一个有效比特，发送数据格式严格按照图 13-3 所示的串口数据帧来完成：首先是起始位（发送端口 txd 从逻辑 1 转化为逻辑 0，并持续 1/9600s），其次是 8 个有效数据比特（低位在前，高位在后），最后是一位停止位（没有添加校验位）。

整个发送模块的状态机包含 5 个状态：s_idle、s_start、s_wait、s_shift 以及 s_stop，其状态转移图如图 13-6 所示。

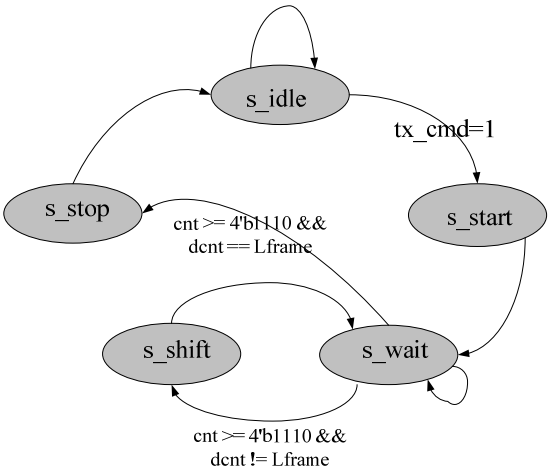


图 13-6　串口发送模块的状态机转移图

其中，s_idle 为空闲状态，当复位信号有效或者发送任务已完成时，发送模块就处于 s_idle 状态，等待下一个发送指令（tx_cmd）的到来。在 s_idle 中，发送完成指示 tx_ready 为高电平，表明随时可以接收外部的发送指令。tx_cmd 信号高有效，且持续时间为一个 bclk 信号的周期，其由顶层模块根据外部按键响应同步整形得到。当 tx_cmd 有效时，发送模块的下一状态为 s_start。

s_start 为发送模块的起始状态，拉低 tx_ready 信号，表明发送模块正处于工作中，并拉低发送比特线 txd，给出起始位，然后跳转到 s_wait 状态。需要注意的是，s_start 状态仅持续一个 bclk 周期，完成相关信号值的改变后，无条件进入 s_wait 状态。

s_wait 为发送模块的等待状态，保持所有信号值不变。当发送模块处于这一状态时，等待计满 16 个 bclk 后，判断 8 个有效数据比特是否发送完毕，如果发送完毕跳转到 s_stop，结束有效数据的发送；否则，跳转到 s_shift 状态，发送下一个有效比特。

s_shift 为数据移位状态，发送模块在这一状态将下一个要发送的数据移动到发送端口上，然后直接跳转到 s_wait 状态。

s_stop 状态完成停止位的发送，当有效数据发送完成后，发送模块进入该状态，发送一个停止位，发送完成后自动进入 s_idle 状态，并且将 tx_ready 信号拉高。在实际设计中，如果读者需要实现 1.5 位或者 2 位停止码，直接修改计数器的数值即可。

发送模块的 Verilog HDL 代码如下所列。

```
module uart_tx(
    bclk, reset, tx_din, tx_cmd, tx_ready, txd
);
//发送模块的工作时钟
input   bclk;
//发送模块的复位信号，高有效
input   reset;
//发送控制信号，高有效
input   tx_cmd;
//期望发送的字节
```

```
input    [7:0] tx_din;
//发送完成指示信号
output tx_ready;
//串口发送数据
output txd;

reg              tx_ready;

//定义串口参数
parameter [3:0] Lframe    = 8;
parameter [2:0] s_idle    = 3'b000;
parameter [2:0] s_start    = 3'b001;
parameter [2:0] s_wait    = 3'b010;
parameter [2:0] s_shift    = 3'b011;
parameter [2:0] s_stop    = 3'b100;

//定义所需的中间变量
//发送模块状态机的状态寄存器
reg    [2:0] state = s_idle;
reg    [3:0] cnt     = 0;
reg    [3:0] dcnt    = 0;
reg              txdt;

assign         txd = txdt;

always @(posedge bclk or posedge reset) begin
    if(reset) begin
            state       <= s_idle;
            cnt          <= 0;
            tx_ready <= 0;
            txdt         <= 1;
      end
      else begin
          case(state)
                //空闲状态，检测发送指示信令
                s_idle: begin
                      tx_ready <= 1;
                      cnt <= 0;
                      txdt <= 1'b1;
                      if(tx_cmd == 1'b1)
```

```
            state <= s_start;
        else
            state <= s_idle;
end

//发送模块的开始状态
s_start: begin
        tx_ready <= 0;
        txdt <= 1'b0;
        state <= s_wait;
end

//延迟等待状态，使每个有效数据保持 16 个 bclk 周期，
//从而得到 9600 的波特率
s_wait : begin
        tx_ready <= 0;
        if(cnt >= 4'b1110) begin
                cnt <= 0;
                if(dcnt == Lframe) begin
                        state <= s_stop;
                        dcnt    <= 0;
                        txdt <= 1'b1;
                end
                else begin
                        state <= s_shift;
                        txdt    <= txdt;
                end
        end
        else begin
            state <= s_wait;
            cnt <= cnt + 1;
        end
end

//移位状态，每进入一次就将下一个发送比特移动到发送数据端口上
s_shift : begin
        tx_ready <= 0;
        txdt <= tx_din[dcnt];
        dcnt <= dcnt + 1;
        state <= s_wait;
```

```
                end

                //停止状态，发送停止位
                s_stop    : begin
                    txdt         <= 1'b1;
                    if(cnt > 4'b1110) begin
                       state <= s_idle;
                       cnt      <= 0;
                       tx_ready <= 1;
                    end
                    else begin
                       state <= s_stop;
                       cnt <= cnt + 1;
                    end
                end
            endcase
        end
    end

endmodule
```

为了测试上述程序，编写了 tb_uart_tx 的测试代码用于检查发送逻辑有无错误，其内容如下所列：

```
module tb_uart_tx2;

    // 定义各输入变量
    reg bclk;
    reg reset;
    reg [7:0] tx_din;
    reg tx_cmd;

    // 定义输出变量
    wire tx_ready;
    wire txd;

    // 例化被测试模块(UUT)
    uart_tx uut (
        .bclk(bclk),
        .reset(reset),
        .tx_din(tx_din),
        .tx_cmd(tx_cmd),
```

```
        .tx_ready(tx_ready),
        .txd(txd)
    );

    initial begin
        // 初始化输入变量
        bclk = 0;
        reset = 1;
        tx_din = 0;
        tx_cmd = 0;

        // 全局复位等待 100 个仿真时间单位
        #100;
      reset = 0;
        #20;
      //设置发送数据为 10，对应成 16 进制就是 8'h0A
        tx_din = 10;
        tx_cmd = 1;
        // 添加仿真条件
        #20;
      // 禁止发送控制信号 tx_cmd，整个仿真过程，完成一次数据发送任务
        tx_cmd = 0;
    end
     always #10 bclk = !bclk;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 13-7 所示。Txd 信号从大约 175ns 开始发送起始信号，然后依次发出 0101_0000（逆序的 8'h0A），紧接着是停止位，满足串口数据标准，达到了预期的设计目的。

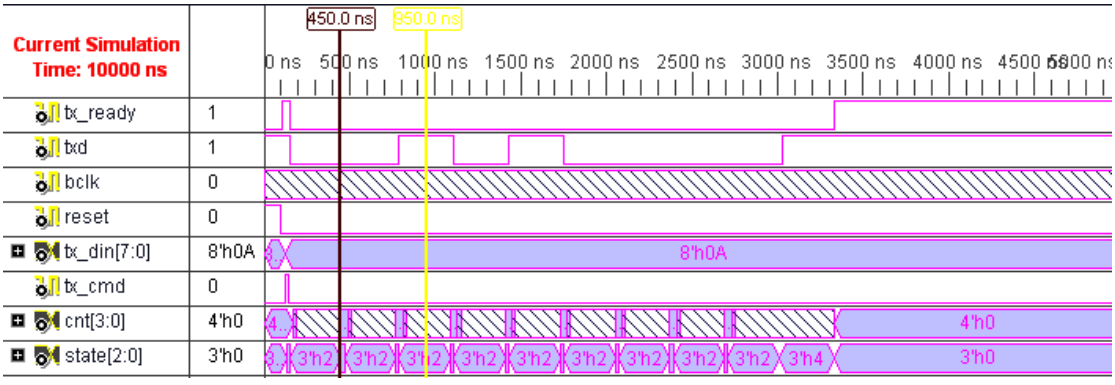


图 13-7　串口发送模块的功能仿真结果

13.2.5 接收模块的 Verilog HDL 实现

但凡涉及到双方通信的系统，接收机的复杂度往往都是高于发送机的，对于串口通信系统也如此。在接收系统中，起始状态和数据都需要依靠接收端检测得到，为了避免毛刺影响，能够得到正确的起始信号和有效数据，需要完成一个简单的最大似然判决，其方法如下：由于 bclk 信号的频率为 9600Hz 的 16 倍，则对于每个数据都会有 16 个样值，最终的采样比特值为出现次数超过 8 次的电平逻辑值。

整个接收模块的状态机包含 3 个状态：s_idle、s_sample 以及 s_stop，其状态转移图如图 13-8 所示。

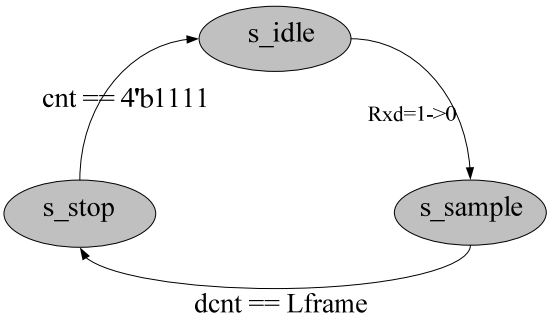


图 13-8　接收模块的状态转移图

其中，s_idle 状态为空闲状态，用于检测接收数据链路上的起始信号。系统复位后，接收模块就处于这一状态，一直检测 rxd 数据是否从 1 跳变为 0，一个起始位代表着新的一帧数据。一旦检测到起始位，立刻进入 s_sample 状态，采集有效数据。在此状态下，rx_ready 信号的值为 1。

s_sample 为数据采样状态，在此状态下，接收模块连续采样数据，并对每 16 个采样样值进行最大似然判决，判决得到相应的逻辑值，这一过程要重复 8 次，并依次完成串并转换，直到接收完 8 个数据比特后，直接进入 s_stop 状态。在这一状态下，rx_ready 信号的值为 0。

s_stop 状态用于检测停止位，为了使得接收模块的使用范围更广，本程序在这一状态等待一定的时间后，直接跳转到 s_idle 状态，无论停止位是 1、1.5 还是 2 位，也不对其数值进行采样判断。这是因为没有添加校验位，根据串口的传输协议，8 个有效数据后肯定是停止位，但停止位所占的时间却是要补偿的，对于不同位宽的停止位，需要修改计数器的模值。

接收模块的 Verilog HDL 代码如下所列。

```
module uart_rx(
    bclk, reset, rxd, rx_ready, rx_dout);
    //接收模块的工作时钟
    input       bclk;
    //接收模块的复位信号
    input       reset;
    //串口接收数据
    input       rxd;
    //接收完成指示信号
    output      rx_ready;
```

```
//接收到的并行数据，以字节形式输出
output [7:0] rx_dout;

//定义串口接收参量和状态数值
parameter [3:0] Lframe      = 8;
parameter [2:0] s_idle      = 3'b000;
parameter [2:0] s_sample      = 3'b010;
parameter [2:0] s_stop      = 3'b100;

reg               rx_ready;
reg    [2:0] state = s_idle;
reg    [3:0] cnt      = 0;
reg    [3:0] num      = 0;
reg    [3:0] dcnt    = 0;
 reg    [7:0] rx_doutmp      = 0;

assign         rx_dout = rx_doutmp;

//接收模块核心处理代码
always @ ( posedge bclk or posedge reset) begin
      if (reset == 1'b1) begin
            state      <= s_idle;
            cnt         <= 0;
            dcnt        <= 0;
            num          <= 0;
            rx_doutmp <= 0;
            rx_ready <= 0;
      end
      else begin
          case(state)
                //检测起始条件
                s_idle: begin
                      rx_doutmp <= 0;
                      dcnt        <= 0;
                      rx_ready <= 1;
                      if(cnt == 4'b1111) begin
                            cnt <= 0;
                            if(num > 7) begin
                                state <= s_sample;
                                num <= 0;
```

```
                end
                else begin
                    state <= s_idle;
                  num <= 0;
              end
          end
          else begin
              cnt <= cnt + 1;
              if(rxd == 1'b0) begin
                  num <= num + 1;
              end
              else begin
                  num <= num;
              end
          end
end

//接收模块数据采样、判决以及串并转换模块
s_sample: begin
          rx_ready <= 1'b0;
          if(dcnt == Lframe) begin
              state <= s_stop;
          end
            else begin
              state <= s_sample;
              if(cnt == 4'b1111) begin
                  dcnt <= dcnt + 1;
                  cnt   <= 0;
                  if(num > 7) begin
                      num <= 0;
                      rx_doutmp[dcnt] <= 1;
                  end
                  else begin
                      rx_doutmp[dcnt] <= 0;
                      num <= 0;
                  end
              end
              else begin
                  cnt <= cnt + 1;
                  if(rxd == 1'b1) begin
```

```
                        num <= num + 1;
                    end
                    else begin
                        num <= num;
                    end
                end
            end
        end

        //接收模块的停止状态
        s_stop: begin
            rx_ready <= 1'b1;
            if(cnt == 4'b1111) begin
                cnt <= 0;
                state <= s_idle;
            end
            else begin
                cnt <= cnt + 1;
            end
        end
    endcase
  end
end

endmodule
```

为了测试上述程序，编写了 tb_uart_rx 的测试代码用于检查发送逻辑有无错误，其内容如下所列：

```
module tb_uart_rx;

    // 输入信号参数
    reg bclk;
    reg reset;
    reg rxd;

    // 输出信号
    wire rx_ready;
    wire [7:0] rx_dout;

    // 例化被测试模块(UUT)
    uart_rx uut (
```

```
        .bclk(bclk),
        .reset(reset),
        .rxd(rxd),
        .rx_ready(rx_ready),
        .rx_dout(rx_dout)
    );

    initial begin
        //初始化输入参数
        bclk = 0;
        reset = 1;
        rxd = 1;

        //全局复位等待 100 个仿真时间单位
        #100;
        reset = 0;
        rxd = 0;
        #640;
        rxd = 1;
    end

    //产生接收时钟信号
    always #10 bclk = !bclk;

endmodule
```

上述程序在 ISE Simulator 中的仿真结果如图 13-9 所示。可以看到串口输入数据依次为“0111_1111”，低位在前，对应着字节 8'hFE，rx_dout 的数据在 rx_ready 为低电平时完成数据的串并转换，并且在 rx_ready 的上升沿输出最终的数据，且保持 16 个 bclk 宽度。通过仿真结果可以看到，上述代码正确实现了串口接收的功能。

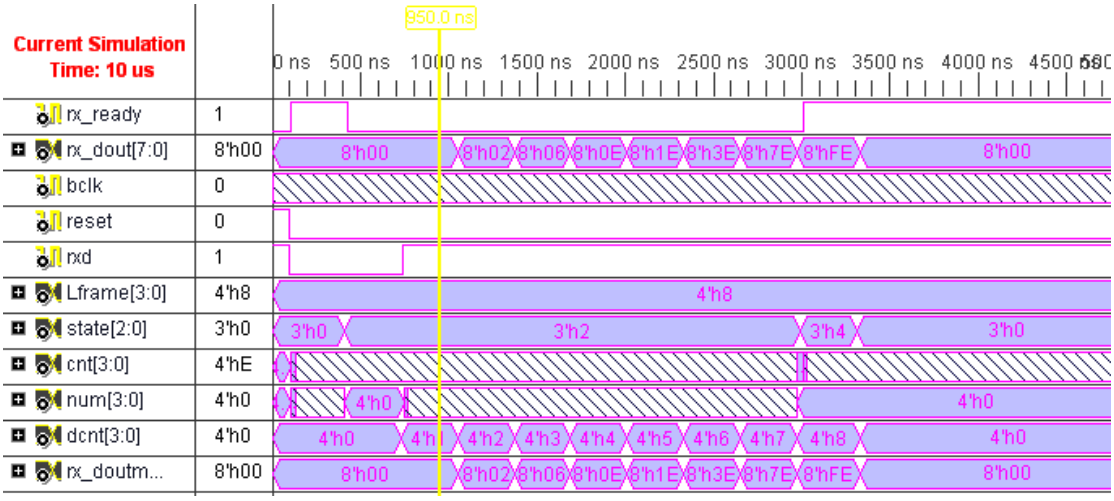


图 13-9　串口接收模块的仿真结果

13.3　RS232 设计板级调试

前面章节已经完成 RS232 接口程序的编写和调试，但其能够完成串口接口的功能，还需要通过板级验证才能达到最终可用的目的。本节主要介绍 13.2 节的程序在 Xilinx 公司的 Spartan 3E 开发板上的调试方法、步骤以及结果。

13.3.1　板级调试说明

1．板级调试场景

板级调试场景由 PC 主机和 Spartan 3E Starter 板组成，二者通过 DB9 串口连接线组成，如图 M 所示。在电路板上按下一次按键就将按照特定规律递增的数据发送到 PC 上，由超级终端来捕获;超级终端发出的数据被电路板接收到到后,用于驱动电路板的 LED 灯作为标志。同时，在设计中加入 ChipScope Pro 核完成有效数据的抓取，通过时序逻辑来判断设计是否正确。

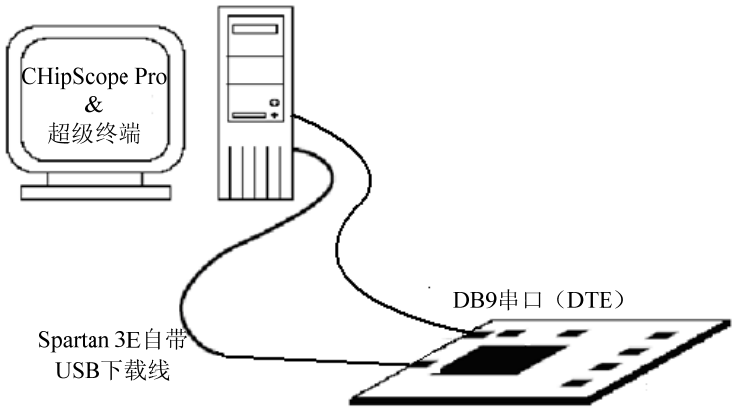


图 13-10　板级调试场景

2．添加管脚约束文件

要将程序在 FPGA 中运行，在完成了功能仿真和综合之后，需要添加管脚约束，然后再完成设计的实现，并生成最终的二进制配置文件，通过 iMPACT 组件下载到 FPGA 芯片或者 PROM 芯片中。因此，第一步就是添加管脚约束文件。

读者阅读 Xilinx UG230 文档后，可以找到 DCT 和 DCE 详细的管脚说明。这里，我们使用了 DCT 接口，并将相应的信号名修改成代码中的信号名，如下所列：

```
NET "rxd" LOC = "U8" | IOSTANDARD = LVTTL ;
NET "txd" LOC = "M13" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW ;
NET "sys_clk_50MHz" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "rst_p" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "button_n" LOC = "V4" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "button_s" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN ;

NET "LED<7>" LOC = "F9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
```

```
NET "LED<6>" LOC = "E9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
```

在实际开发中，在进行板级调试第一步之前，都需要根据原理图或者 PCB 图得到完整、正确的管脚约束信息。然后，在工程中新建 UCF 文件，将上述管脚约束粘贴进去，完成整个工程完整的约束文件。此时，可以实现整个工程，察看设计所占资源，因为此时的资源评估是最准确的，这本身也是一个良好的设计习惯。双击“Place & Router”下的“Place & Route Report”即可查阅最终的资源报告，如图 11-31 所示。

```
Design Summary Report:

 Number of External IOBs                          14 out of 232     6%

   Number of External Input IOBs                  5

      Number of External Input IBUFs              5
        Number of LOCed External Input IBUFs      5 out of 5     100%


   Number of External Output IOBs                 9

      Number of External Output IOBs              9
        Number of LOCed External Output IOBs      9 out of 9     100%


   Number of External Bidir IOBs                  0


   Number of BUFGMUXs                        2 out of 24      8%
   Number of Slices                         80 out of 4656    1%
      Number of SLICEMs                      1 out of 2328    1%
```

图 11-31 实现后的资源统计结果

13.3.2 配置超级终端

1．超级终端说明

超级终端是Windows操作系统中一个通用的串行交互软件，大多数嵌入式系统都具有相应的交互程序，可通过这些程序，使超级终端成为嵌入式系统的“显示器”。超级终端的原理并不复杂，它是将用户输入随时发向串口（采用TCP协议时是发往网口），但并不显示输入。它显示的是从串口接收到的字符。所以，嵌入式系统的相应程序应该完成的任务便是：首先，将自己的启动信息、过程信息主动发到运行有超级终端的主机；其次，将接收到的字符返回到主机，同时发送需要显示的字符（如命令的响应等）到主机。

2．超级终端的配置

（1）从“开始→所有程序→附件→通讯→超级终端”，打开超级终端，会弹出图13-10所示的对话框，在名称的编辑栏中输入“SP3E_UART”，点击“确定”按键，进入下一页。

图 13-10　超级终端新建页面

（2）在图13-11的“连接时使用”的下拉框中选择“Com1”，即计算机上的串口，点击“确定”按键进入串口配置页面，设置波特率为9600，数据位为8，无奇偶校验，停止位为1，如图13-12所示，点击“确定”按键。其中，串口的各个参数必须和串口外设IP Core的配置相同，否则无法完成PC机和FPGA芯片的交互通信。

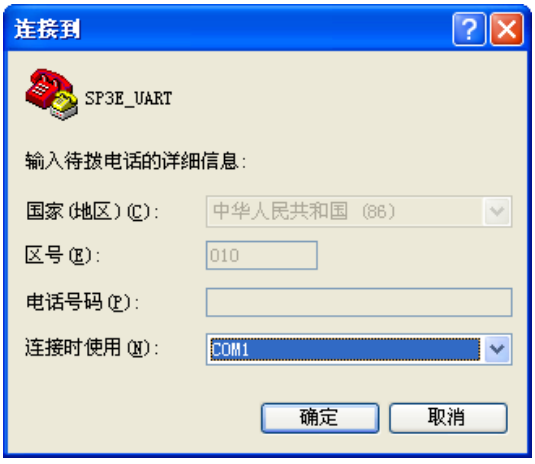


图 13-11　连接时使用串口

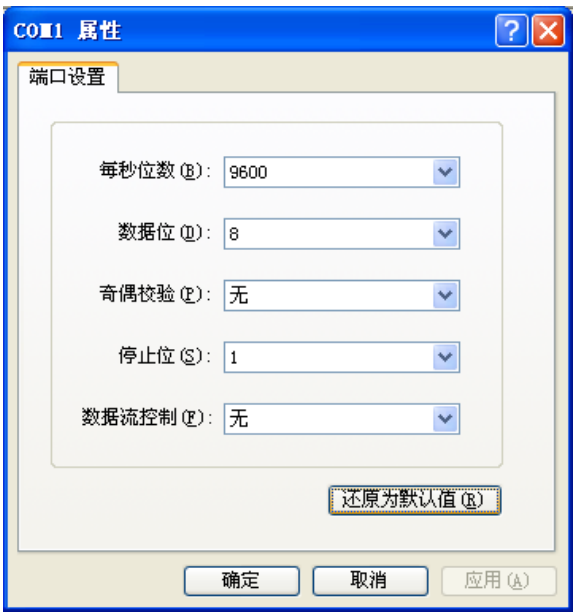


图 13-12　COM1 属性设置

（3）上述配置完成后，点击“确定”按键，即可进入图 13-13 所示的超级终端操作界面。为了更好地完成演示实验，需要进行进一步配置。点击“文件”菜单下的“属性”命令，进入图 13-14 所示的对话框。

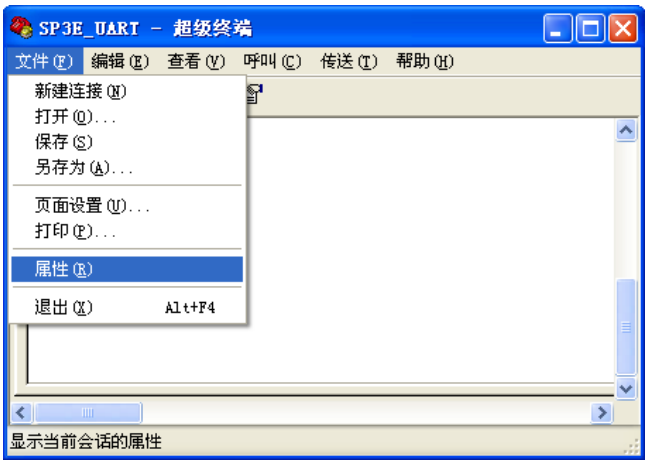


图 13-13　超级终端的主操作界面

（4）单击超级终端属性页面的“设置”页面，进入图13-14所示的设置页面，然后点击其中的“ASCII 码设计（A）…”按键，进入图13-15所示的配置页面，将其中的所有“ASCII码发送”和“ASCII码接收”选项都选中，然后单击“确定”按键，完成超级终端设置。只有这样，才能在超级终端上查看用户已发送的数据。

图 13-14　超级终端属性设置页面

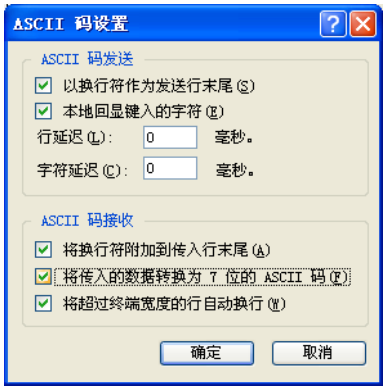


图 13-15　超级终端 ASCII 码设置页面

提示：使用超级终端时需要注意当前状态，如果处于断开状态一定要重新连接好才可以接收到数据。另外，当输出用户无法识别的符号时，可能是对应的ASCII码设置错误，或者波特率不正确，也可能需要保存文本。

13.3.3　添加 ChipScope Pro 核

按照例 6.9.2 节 ChipScope Pro 的开发方法在设计中添加 ILA 核，并命名为 cs_uart，其相关的信号连接如图 13-16~图 13-21 所示。

图 13-16 给出了观测信号的触发端口信息，共使用了 3 个触发端口，每个端口的信号数分别为 14、12 和 5。其中 TRIG0 通道用于观测发送模块信号，TRIG1 通道用于观测接收模块信号，TRIG1 通道用于观测按键响应模块信号。

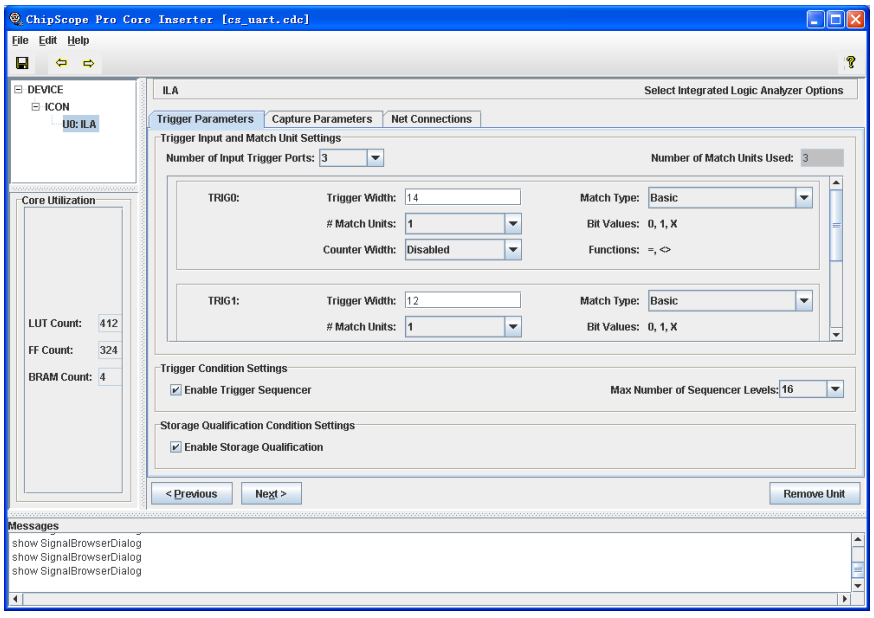


图 13-16　ChipScope Pro 触发端口设置参数示意图

图 13-17 为信号采样参数配置窗口，设置信号采集深度为 2048，数据在时钟上升沿采样，且选中“Data Same As Trigger”选项，保证数据和触发端口一致，从而节省信号端口数和存储空间。

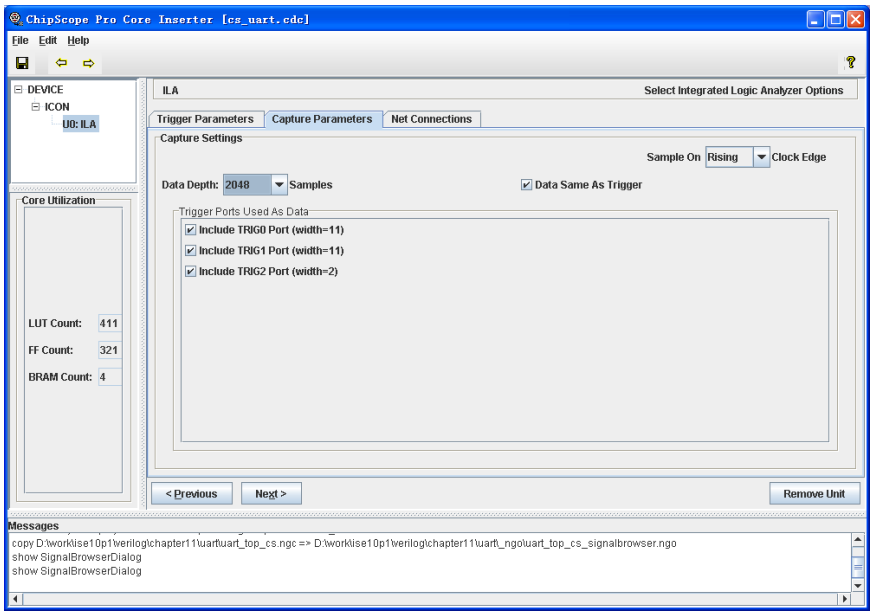


图 13-17　ChipScope Pro 信号采样参数示意图

图 13-18 为采样时钟配置页面，在设计中选择波特率发生器模块的 bclk 信号为采样时钟 。在一般应用中，采样时钟的频率和数据速度的比值越大，就越能得到信号的细节特征，但也需要消耗较大的片内内存（块 RAM），因此需要设计人员在调试中，和采样深度、采样信号位宽以及芯片中可利用的资源联合考虑，作出最佳的权衡，

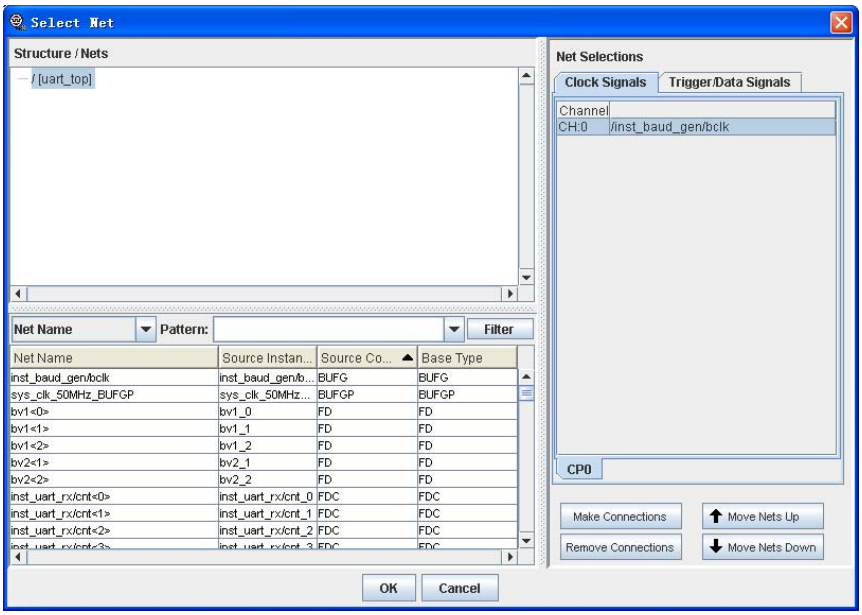


图 13-18　ChipScope Pro 信号采样参数示意图

图 13-19 为触发端口 1 中发送模块中信号连接关系，包括串口的输出端口 txdt、串口发送完成指示信号 tx_ready，发送命令 tx_cmd，串口发送模块状态机 state_FSM_FFd1、state_FSM_FFd2、state_FSM_FFd3，以及待发送的字节 din[7:0]，共 14 个信号。

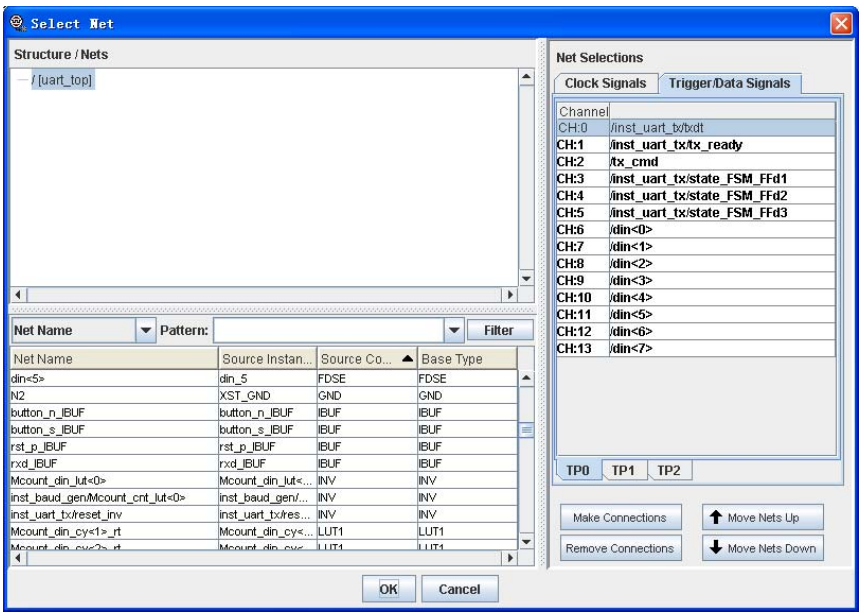


图 13-19　触发端口 1 的信号连接关系

图 13-20 为触发端口 2 中接收模块中信号连接关系，包括串口的输入端口 rxdt、串接收完成指示信号 rx_ready，串口发送模块状态机 state_FSM_FFd1、state_FSM_FFd2，以及接收到的字节 doutmp[7:0]，共 12 个信号。

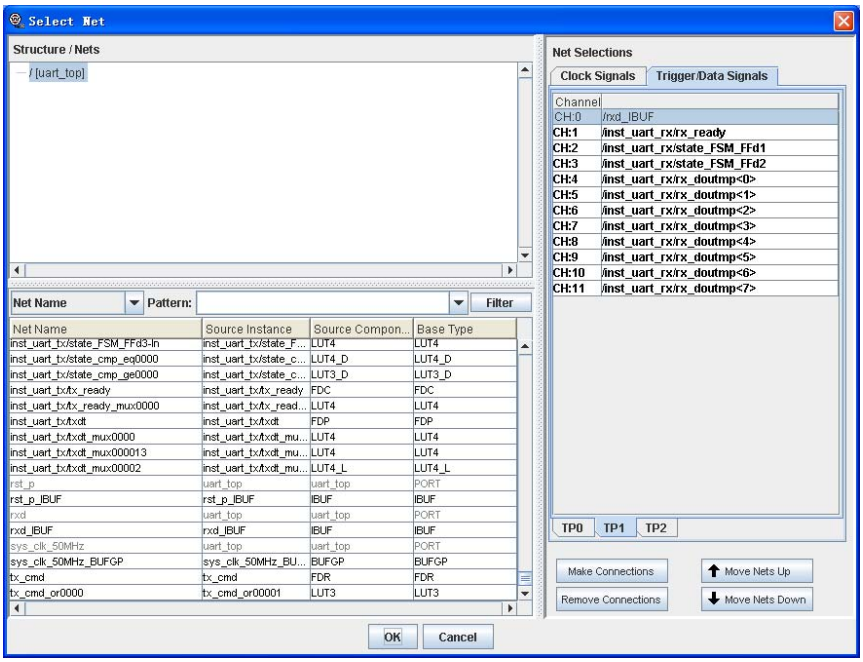


图 13-20　触发端口 2 的信号连接关系

图 13-21 为触发端口 3 中接收模块中信号连接关系，包括 bv1_posedge、bv2_posedge、button_n_IBUF、button_s_IBUF 以及 rst_p_IBUF，共五个信号。期望从中观测到，前两个信号分别是 button_n_IBUF 和 button_s_IBUF 输入采样，其脉冲宽度为 bclk 的一个周期。

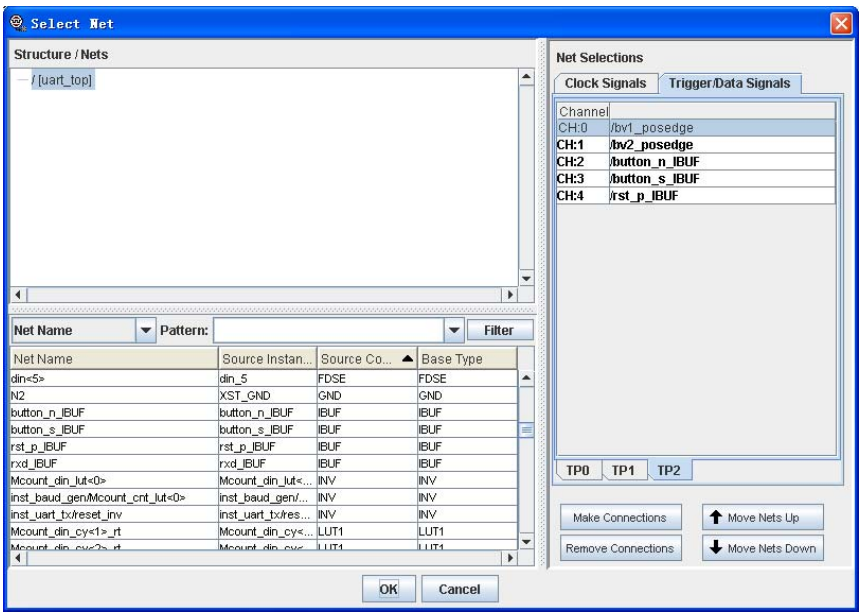


图 13-21　触发端口 3 的信号连接关系

完成上述配置后，单击“OK”返回 Chipscope Pro 主界面，再返回 ISE 操作界面，然后完成整个工程的重新实现，图 13-22 给出了综合后的资源占用情况，可以看出，采集和存储数据使用了 4 个块 RAM，345（425-80）个 Slice。

```
Design Summary Report:

 Number of External IOBs                          14 out of 232     6%

   Number of External Input IOBs                  5

      Number of External Input IBUFs              5
        Number of LOCed External Input IBUFs      5 out of 5     100%


   Number of External Output IOBs                 9

      Number of External Output IOBs              9
        Number of LOCed External Output IOBs      9 out of 9     100%


   Number of External Bidir IOBs                  0


   Number of BSCANs                           1 out of 1     100%
   Number of BUFGMUXs                         3 out of 24     12%
   Number of RAMB16s                          4 out of 20     20%
   Number of Slices                         425 out of 4656    9%
      Number of SLICEMs                      81 out of 2328    3%
```

图 13-22　添加 Chipscope 核后的逻辑资源占用情况

上一步完成后，双击 ISE 过程管理区的“Generate Programming File”选项，生成可配置的比特文件。

13.3.4　系统调试结果

接下来的步骤就是将生成的 bit 文件下载到开发板上的 FPGA 中。这里有两种方法：一种利用 iMPACT 配置 FPGA/PROM，但无法使用 ChipScope 组件实时观测；另一种通过 Chipscope Analyze 来分配置，但无法将设计配置 PROM。本例由于主要用于完成板级调试，因此使用后一种方法。

双击 ISE 过程管理区的“Analyze Design Using Chipscope”选项，可以打开 Chipscope Analyze，如图 13-23 所示。后续的 JTAG 链扫描、比特文件配置以及 CDC 文件导入等操作和例 6-19 中的步骤一致，这里就不再重复描述。

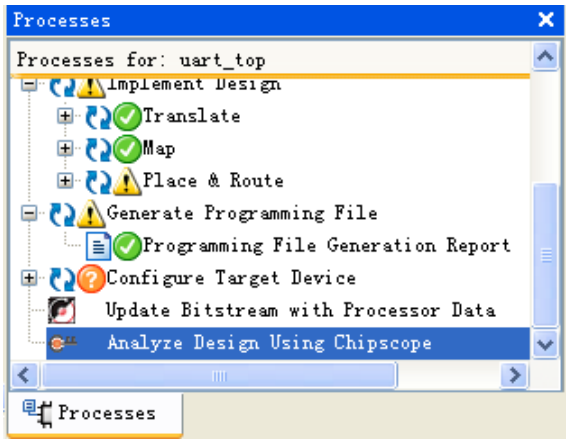


图 13-23　打开 Chipscope Analyzer 组件操作示意图

在配置完 FPGA 芯片后，打开 PC 端已建好的“SP3E_UART”的超级终端，以便于软硬件联合测试。

1．观测串口发送模块

为了观测串口发送模块，首先需要设置触发条件，检测到“tx_cmd”为高时，开始采集数据，相关的触发参数配置如图 13-24 所示。

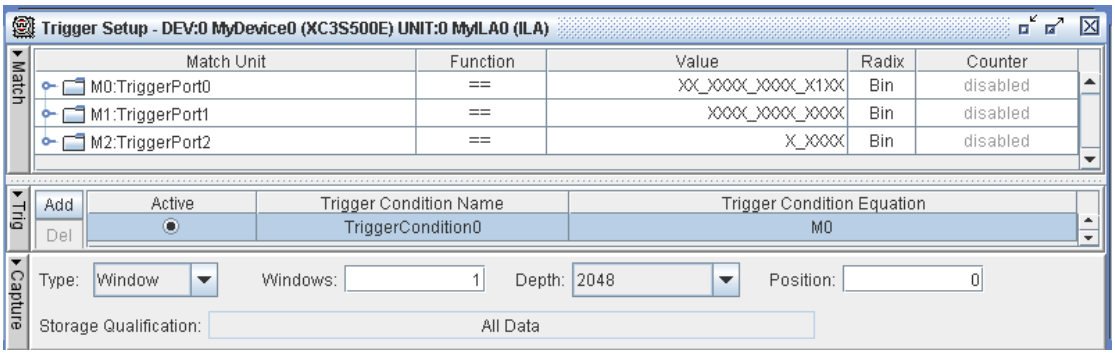


图 13-24　串口发送模块的触发参数配置页面

然后，按下开发板上的“BTN South”按键，完成系统复位，将初始值设置为 30。然后再按下“BTN North”按键，发送“31”到串口端口，其板级采样数据如图 13-25 所示，符合串口接口时序，从而验证了设计的正确性。

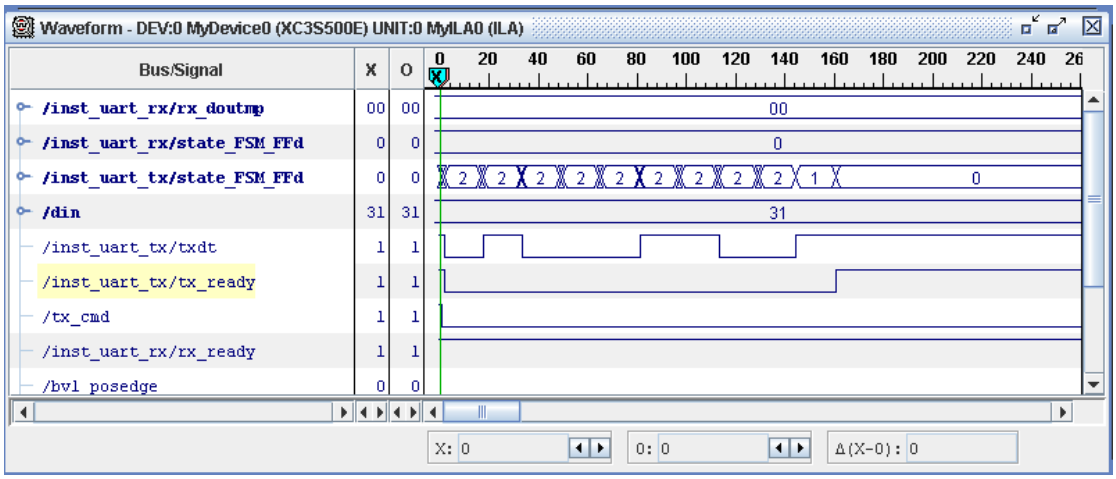


图 13-25　串口发送模块时序分析结果示意图

连续按下三次后，可以在超级终端上看到图 13-26 的采集结果，进一步验证了设计的正确性。此外，这也是开发板和 PC 机交换数据的一种方式。

图 13-26　超级终端接收数据显示结果

2．观测串口接收模块

测试完发送模块，接着测试接收模块。首先需要设置触发条件，检测到“rxdt”由高变低后，开始采集数据，相关的触发参数配置如图 13-27 所示。

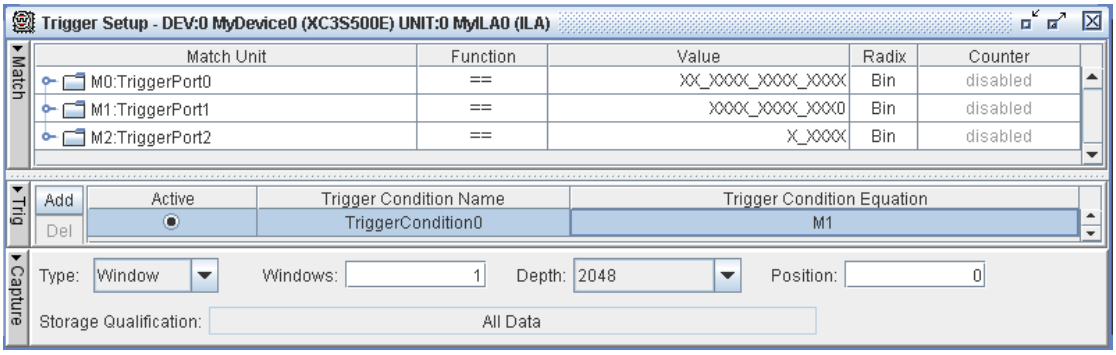


图 13-27 串口接收模块的触发参数配置页面

然后在 PC 上当超级终端设置为当前窗口，然后在键盘上键入“a”，可以在超级终端上看到字母“a”，并且在 Chipscope 中看到图 13-28 所示的时序图，其中，在 tx_ready 的上升沿送出 61，其对应者字母“a”的 ASCII 码。同时也可以看到开发上的 8 个 LED 灯，依次亮、灭的状态为 00111101（0 代表灭、1 代表亮），其对应着 61 的二进制表示形式。

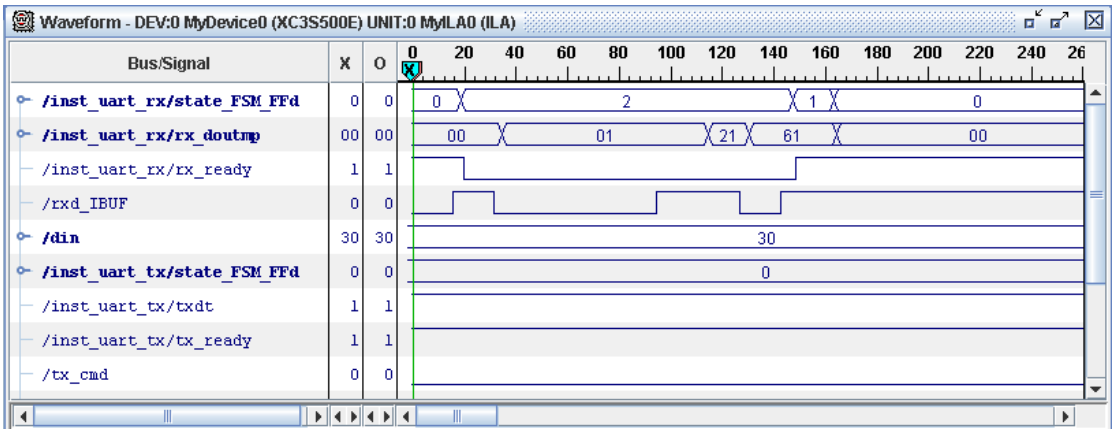


图 13-28 串口接收模块时序分析结果示意图

通过上述测试，表明设计的接收模块和发送模块都可以正常工作，说明设计是正确的。

13.4 本章小结

本章作为全书的总结给出一个串口通信的开发实例。首先介绍了串口硬件接口模块以及相应的 RS232 通信协议。其次重点讨论了串口通信控制器的 Verilog HDL 实现。整个设计在自顶向下、逐层分割的层次化设计思想的指导下完成；顶层模块设计是整个设计的关键，对系统给出一个全面的、宏观的规划，并调用核心的处理子模块，包括波特率发生模块、信号接收模块以及信号发送模块。最后，说明了整体设计在 Spartan 3E Starter 开发板上完成板级调试的过程，介绍了如何添加约束文件和辅助调试工具超级终端的配置方法，并给出了最终的调试结果，表明设计的正确性。通过本章的学习，读者要明白在基于 Verilog HDL 语言的电子系统设计环节中，功能的设计是第一位的，但并不是全部，代码的验证以及最终实物调试也是非常重要的环节。

13.5 思考题

1．异步接收怎样实现数据比特同步？
2．异步接收有哪些优点？
3．为什么异步接收的时钟为数据率的 16 倍？
4．应如何估计异步数据传输下发/收端的最大允许频率偏差？
5．异步数据传输有哪些缺点？
6．在串口接收和发送模块中，状态机有什么用处？
7．如何实现从发送缓冲器到发送移存器的正确传送？
8．如何实现数据从接收移存器到接收缓冲器的正确传送的？应该怎么考虑该过程？
9．Windows 操作系统的超级终端具有什么样的功能？
10．在串口设计中，如果要实现其他波特率的串口接口，该如何修改？

参考文献

1. 吴继华、王诚，设计与验证 Verilog HDL，北京：人民邮电出版社.
2. Mike Gordon. The semantic of Verilog HDL . P roc Tench A nnual IEEE Sympo sium on Logic in Computer Science .IEEE Computer Society Press, 1995. 1362145.
3. 夏宇闻，Verilog 数字系统设计教程，北京：北京航空航天大学出版社，2003.
4. 徐惠民、安德宁，数字逻辑设计与 VHDL 描述，北京：机械工业出版社，2002.
5. 王诚等，FPGA/CPLD 设计工具 Xilinx ISE 使用详解，北京：人民邮电出版社，2005.
6. 简弘伦，精通 Verilog HDL：IC 设计核心技术实例详解，北京：电子工业出版社.，2005.
7. 谷銮等，FPGA 动态可重构理论及其研究进展，计算机测量与控制，2007.15，pp1415-1418.
8. 田耘，徐文波，Xilinx ISE Design Suite 10.x FPGA 开发指南——逻辑设计篇，北京：人民邮电出版社，2008.
9. 莫海永、张申科，FPGA 中双向端口 I/O 的研究
10. 王钿，卓兴旺，基于 Verilog HDL 的数字系统应用设计，北京：国防工业出版社，2006.
11. 刘秋云、王佳，Verilog HDL 设计实践与指导，北京：机械工业出版社，2005.
12. 江国强，数字系统的 Verilog HDL 设计，北京：机械工业出版社，2007.
13. 黄智伟等，FPGA 系统设计与实践，北京：电子工业出版社，2005.