SPECIALLY DESIGNED FOR ENGINEERS AND TECHNICIANS OF ELECTRONICS

ELECTRONIC ENGINEER

# FPGA Design Based on Verilog

# 基于Verilog的FPGA设计基础

杜慧敏　李宥谋　赵全良　编著

西安电子科技大学出版社
http://www.xduph.com

国家自然科学基金资助

# 基于 Verilog 的 FPGA 设计基础

杜慧敏　李宥谋　赵全良　编著

西安电子科技大学出版社

2006

## 内 容 简 介

本书简要介绍了 FPGA 的编程技术，详细讨论了以 Altera FPGA 为代表的可编程器件的结构和特点、Altera Quartus II 集成环境的使用以及目前工业界最常用的仿真工具 Modelsim 的使用。重点讲授了 FPGA 设计流程中的基本概念、所采用的步骤和应该遵循的原则，包括模块划分原则、可综合 Verilog 编码风格、验证程序的编写方法和静态时序分析等。另外，本书结合 Altera 公司的 Nios II 软核，简单介绍了基于 SOPC 的系统设计方法以及 Altera SOPC Builder 软件的使用方法。

本书可作为从事数字集成电路设计及相关工程技术人员的参考书，也可作为大专院校电子信息、自动控制等专业高年级本科生及研究生的教学用书。

**图书在版编目（CIP）数据**

**基于 Verilog 的 FPGA 设计基础** / 杜慧敏等编著. —西安：西安电子科技大学出版社，2006.2
(国家自然科学基金资助)
ISBN 7–5606–1626–7
Ⅰ. 基… Ⅱ. 杜… Ⅲ. ① 硬件描述语言—程序设计 ② 现场可编程门阵列—系统设计
Ⅳ. ① TP312 ② TP332.1

**中国版本图书馆 CIP 数据核字（2005）第 155182 号**

责任编辑 王晓杰 云立实 臧延新
出版发行 西安电子科技大学出版社(西安市太白南路 2 号)
电 话 (029)88242885 88201467 邮 编 710071
http: //www.xduph.com E-mail: xdupfxb@pub.xaonline.com
经 销 新华书店
印刷单位 西安文化彩印厂
版 次 2006 年 2 月第 1 版 2006 年 2 月第 1 次印刷
开 本 787 毫米×1092 毫米 1/16 印张 24.375
字 数 578 千字
印 数 1～4000 册
定 价 35.00 元
ISBN 7–5606–1626–7/TP・0931
**XDUP 1918001-1**

# 前　言

FPGA(现场可编程门阵列)是一种大规模可编程逻辑器件，自 1984 年第一片 FPGA 问世至今，FPGA 已经历了 20 年的发展历史。在这 20 年的发展过程中，FPGA 从最初的 1200 门，发展到现在的几百万门，器件的集成度不断提高。以 Altera、Xilinx 等为代表的 FPGA 厂家不断更新、优化产品架构和生产工艺，降低了 FPGA 的功耗和系统成本，推出了很多高性能、低价位的解决方案，将 FPGA 应用从传统的高端通信产品扩展到汽车和消费类电子产品中。

随着低成本 FPGA 的推广，越来越多的科技工作者开始应用 FPGA 解决实际问题，也有越来越多的高等学校开始在本科生中开设基于 FPGA 进行系统设计方面的课程，以满足社会对这些方面人才的需要。本书正是在这个背景下编写的，其目的是为工程技术人员和在校学生提供基于 FPGA 设计的基本方法。

在以 FPGA 为载体的设计过程中，必须遵循一定的设计流程、设计原则和方法。本书涵盖了基于 FPGA 设计全过程的主要内容，包括系统规范的制定、模块的设计划分、可综合代码的编写原则、仿真程序的编写和后端验证方法等，并给出了一些较为复杂的设计实例。除了设计方法之外，本书以 Altera 的 FPGA 为例，对 FPGA 内部结构做了深入的分析，介绍了世界著名的 FPGA 公司的产品及其特点。作为 FPGA 设计过程中必不可少的开发环境，本书介绍了 Altera 公司的 Quartus II 开发环境以及 MIT 的 Modelsim 仿真软件。另外，对于目前非常流行的 SOPC 设计也做了简要的介绍，可供嵌入式系统的设计人员参考。

本书的三位作者在基于 FPGA 设计方面有着丰富的经验，书中的部分内容也是作者多年工作的总结，希望能对读者有所帮助。

本书分为 8 章，其中第 1 章、第 3 章和第 4 章由杜慧敏执笔，第 2 章和第 7 章由李宥谋执笔，第 5 章、第 6 章和第 8 章由赵全良执笔。另外，在本书的编写过程中，得到了韩俊刚教授、蒋林教授和谢庆胜硕士的大力支持，他们为本书提出了许多宝贵意见。邢立冬和张阿宁同志验证了本书中的设计实例。在此一并表示感谢。

本书可作为大专院校 FPGA 设计基础课程的教材。书中所有例子均是用 Verilog 语言描述的，因此，使用本教材的学生应先学习 Verilog 语言和数字电路设计两门课程。另外，本书也可以作为相关领域工程技术人员的参考资料。

由于编者的水平有限，书中难免存在缺点和错误，恳请各位专家和读者批评指正。作者的联系方式为：fv@xiyou.edu.cn，lym@xiyou.edu.cn，zql@xiyou.edu.cn。

编　者

2005 年 11 月

# 目　　录

# 第 1 章 绪　论

现场可编程门阵列 FPGA(Field Programmable Gate Array)是 20 世纪 80 年代中期由美国 Xilinx 公司首先推出来的。随着半导体加工工艺的不断发展，FPGA 在结构、速度、工艺、集成度和性能等方面都有了极大的改进和提高，与之相对应的设计方法学和自动化设计工具也得到迅速的发展。其中，设计工具的自动化为 FPGA 的应用和发展起到了推波助澜的作用。本章将简要回顾一下 FPGA 器件和自动化设计工具的发展过程，并简要介绍基于 FPGA 设计的流程。

## 1.1　FPGA 概 述

### 1.1.1　FPGA 发展的简要回顾

简单地讲，现场可编程门阵列 FPGA 是一种可以编程的数字集成电路 IC(Integrated Circuit)，它包含了可配置的逻辑块以及逻辑块之间的互连线。所谓的现场可编程是指设计人员可以通过在工作现场(如实验室、调试现场等)完成对这些逻辑块和连线的配置，以实现或改变复杂的电子系统的功能。

为了对 FPGA 有一个较全面的认识，我们简要回顾一下与 FPGA 发展相关的技术。如图 1.1 所示，图中的白色区域表示已开始研究该技术的时间，但是由于种种原因没有得到很好的推广。

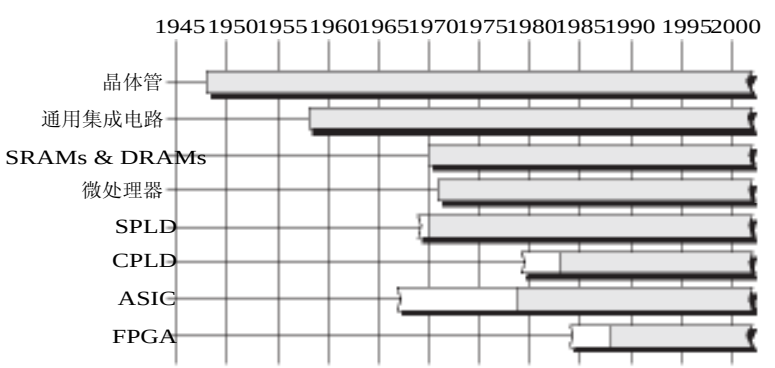



图 1.1　FPGA 技术发展过程

1) 晶体管

图 1.2　第一只晶体管

1947 年 12 月 23 日，美国物理学家 William Shockley，Walter Brattain 和 John Bardeen 在贝尔实验室首先制造出第一只点接触式锗晶体管，如图 1.2 所示。他们三人也因对半导体物理做出的杰出贡献在 1956 年获得诺贝尔物理学奖。1950 年制造出的双极型晶体管，也称 BJT(Bipolar Junction Transistor)。

20 世纪 50 年代后期可以用硅片生产晶体管。由于硅比锗要便宜的多，因此 BJT 工艺的晶体管得到了广泛的应用。根据连接关系的不同，BJT 工艺的晶体管可以构建出 TTL (Transistor-Transistor Logic)类型的数字逻辑门，也可以构造 ECL(Emitter-Coupled Logic)类型的数字逻辑门。ECL 门的速度比 TTL 门的快，但是功耗比 TTL 门大。

1962 年，美国科学家 Steven Hofstein 和 Fredric Heiman 在普林斯顿的 RCA 研究试验室发明了一种被称为金属氧化物场效应管 MOSFET(Metal-Oxide Semiconductor Field-Effect Transistors)工艺的元件，简称 FET。虽然当时的 FET 比 BJT 的速度慢，但是它们具有廉价、体积小和功耗小等特点。有两种类型的 FET，一种称为 NMOS，另一种称为 PMOS。由 PMOS 和 NMOS 可以构造出 CMOS 逻辑门(Complementary Metal-Oxide Semiconductor)，这种方式实现的逻辑门速度上比 TTL 要稍稍慢一些，但是两者的工艺是一致的。CMOS 工艺的逻辑门的最大优点就是它的静态功耗非常小。CMOS 工艺是现代超大规模集成电路的主流工艺。

2) 通用集成电路(IC)

第一个晶体管是以分立元件的形式出现的，它可以用小的金属外壳独立封装。在这个基础上，人们开始想到在一个半导体上生产出一个完整的电路。1952 年 5 月，英国雷达专家 G.W.A.Dummer 首先提出集成电路的概念。时隔 6 年之后，1958 年 9 月，美国德州仪器 TI(Texas Instruments)Jack Kilby 成功地生产出第一个由五个元件构成的简单晶体振荡器集成电路，如图 1.3 所示。2000 年，Jack Kilby 因发明集成电路而和其他物理学家一起获得了物理学的诺贝尔奖，他是诺贝尔历史上少有的获此殊荣的工程师。

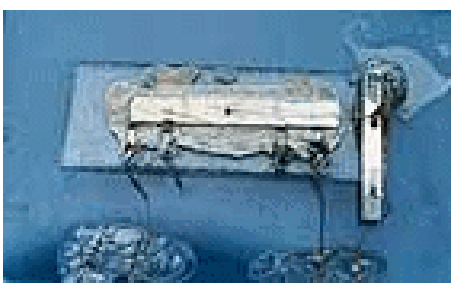


图 1.3　第一块集成电路

瑞士物理学家 Jean Hoerni 和美国物理学家 Robert 在 Kilby 的基础上发明了现代 IC 生产中用到的光刻技术(Optical Lithographic Techniques)，奠定了现代 IC 生产的基础。20 世纪 60 年代中期，TI 开始为军用和商用生产 5400 系列和 7400 系列的 TTL IC。这些 IC 完成非

常简单的逻辑门。例如，2 输入 NAND 门 7400，2 输入 NOR 7402 等。有时，把这些逻辑也称为粘和逻辑(英文称为 Jelly-bean Logic)。

3) SRAM，DRAM 和微处理器

20 世纪 60 年代后期到 70 年代前期，数字 IC 得到了飞速的发展。1970 年 Intel 公司宣布 1024 bit 的 DRAM 诞生了，当时著名的仙童(Fairchild)半导体公司生产出第一个 256 bit 的 SRAM，它是 FPGA 的基础。1971 年，Intel 公司生产出第一个包含 2300 个晶体管，每秒钟完成 60 000 个操作的微处理器。

4) SPLD 和 CPLD

第一个可编程的 IC 是在 1970 年以 PROM 的形式出现的，通常被称为可编程逻辑元件 PLD(Programmable Logic Devices)。刚开始的 PLD 结构比较简单，只能完成简单的功能，把这种 PLD 称为简单的 PLD，即 SPLD。到 20 世纪 70 年代末以后，出现了规模更大，集成度更高的 PLD，可以完成非常复杂的功能，我们把这类 PLD 称为复杂 PLD，即 CPLD。1984 年，Altera 公司推出了结合 CMOS 和 EPROM 工艺的 CPLD，这种 CPLD 可以实现较复杂的功能，功耗相对又比较小，将 CPLD 元件的性能提高了一大步。

5) ASIC

在 FPGA 的发展史上，专用集成电路 ASIC 的实现方法对 FPGA/CPLD 的发展有着重要的影响。按实现方法划分，ASIC 可以被分为全定制 ASIC 和半定制 ASIC。半定制 ASIC 中又分为门阵 ASIC、标准单元 ASIC 和结构化 ASIC。

在数字 IC 发展的早期，除了存储器之外，主要有两大类 IC，一类是由 TI 和 Fairchild 公司生产的功能相对简单的数字 IC，它们是以现货方式提供给用户。另外一类就是根据用户特殊需求设计生产的全定制的 IC，称为专用集成电路 ASIC(Application Specific Integrated Circuit)。在全定制的 ASIC 中，设计工程师完成用于芯片加工的每个掩膜层的控制，ASIC 厂商并不在硅片上预先放置任何元件，也不提供任何预定义的逻辑门和功能，而是借助于特殊的工具，由工程师手工完成每个晶体管的尺寸设计，然后根据这些晶体管创建高层的功能。例如，如果工程师需要一个速度快的逻辑门，那么他可以改变构造该门的晶体管的尺寸。全定制的设计非常复杂，同时也非常耗时，但是生产的芯片在面积、功耗、速度等方面具有优势。

6) Micromatrix 和 Micromosaic

20 世纪 60 年代中期，Fairchild 半导体公司引入了被称为 Micromatrix 的元件，该元件中包含了 100 个左右的未连接的裸晶体管。为了使这种元件能完成特定的功能，设计工程师通过手工的方式绘制这些晶体管之间的连线。虽然这种方式费时，且易出错，但是却允许工程师在一定的时间内，用较高的代价形成一个定制的元件。

到了 20 世纪 60 年代的后期，Fairchild 半导体公司生产了 Micromosaic 元件。它包含了几百个没有连接的晶体管，这些晶体管可以在以后连接并实现 150 个左右的与/或非门。Micromosaic 的主要特性就是设计工程师可以通过布尔表达式说明元件要完成的功能，然后通过计算机程序决定哪些晶体管进行互联，同时计算机程序也构造用于生产元件的光掩模。这在 IC 历史上具有重要的意义，它是现代 ASIC 中门阵的先驱。

7) 门阵

门阵的概念最早由 IBM 和富士通等公司在 20 世纪 60 年代后期提出的，然而直到 20

世纪 70 年代中期，基于 CMOS 的门阵工艺成熟后该项技术才得以推广。门阵的基本思想是预先在硅片上生产一些没有连接关系的称为基本单元的晶体管和电阻，上面几层作为晶体管互连的金属层用全定制掩模的方式确定。门阵实现方式的缺点是大多数设计可能不能完全使用门阵的内部资源，造成一些浪费。另外，门阵的布局和布线优化程度也不高。

8) 标准单元

为了克服门阵的缺点，在 20 世纪 80 年代初，推出了基于标准单元的 ASIC。标准单元有许多地方与门阵非常相似，ASIC 厂家定义了设计工程师用到的单元库，同时也提供了软宏和硬宏的库。这些库包括处理器、通信接口以及 ROM 和 RAM 等，设计人员可以根据需要考虑是否购买和重用这些被称为知识产权 IP(Intelligence Property)的标准单元。与门阵不同的是，标准单元并不预先在芯片上生产元件，而是通过软件工具决定网表中逻辑门之间的互连，并产生用于生产的每层定制的光掩膜。

9) FPGA

在 20 世纪 80 年代，在数字 IC 和可编程逻辑元件之间存在着一个空挡，即 SPLD 和 CPLD 具有易修改和设计周期短等特点，但由于结构和资源等方面的限制，不能实现非常复杂的功能。另外一方面是 ASIC，它可以实现非常复杂的功能，但开发周期长并且不可修改。为了填补这个空挡，1984 年，Xilinx 公司开始在市场上推出一种现场可编程门阵列器件，即 FPGA。FPGA 结构类似于门阵列 ASIC，但它是可编程的。很多时候，FPGA 被称为是一种可编程的 ASIC。

10) 结构化 ASIC

结构化 ASIC 是在 20 世纪 90 年代初提出的概念，但是这一概念直到 10 年之后才得到 ASIC 厂商的重视。2003 年，Altera 首先推出了 Hard Copy Structured ASIC。结构化 ASIC 类似于门阵列，底层单元都已经做好了，掩膜也是现成的。用户只需要对几层的金属连线和通孔进行编程，大大节约了掩膜层的成本。结构化 ASIC 主要适合中规模产量需要，单芯片成本低于 FPGA，但高于标准单元 ASIC。结构化 ASIC 的主要特点是和其本身 FPGA 100%兼容(包括功能、管脚和时序等)，用户验证过的 FPGA 原型设计，可直接由厂家在几周内转换成 ASIC。由于结构化 ASIC 成本低，同时生产周期比较短，因此，一些 ASIC 厂家现在正在积极推广结构化 ASIC 器件。

### 1.1.2 FPGA 与 ASIC

IC 的种类非常多，从完成简单逻辑功能的 IC 到完成复杂系统功能的系统芯片应有尽有。我们感兴趣的两类芯片是可编程逻辑器件 PLD 和专用集成电路 ASIC，其中可编程逻辑器件按其规模划分为低密度可编程逻辑器件和高密度可编程逻辑器件，FPGA 是高密度可编程逻辑器件。

与通用 IC 不同的是，这两类芯片都可以根据用户的需要实现特殊功能。其中，ASIC 是为用户定制的芯片，需要经过 ASIC 厂家生产，它可以完成非常复杂的系统功能，芯片的规模也可以非常大。与通用集成电路相比，ASIC 在构成电子系统时具有以下几个方面的优越性：

(1) 缩小系统的体积，减轻系统的重量，降低系统的功耗和提高系统的性能。

(2) 提高可靠性。用 ASIC 芯片进行系统集成后，外部连线减少，因而可靠性明显提高。

(3) 可增强保密性。电子产品中的 ASIC 芯片对用户来说相当于一个“黑匣子”，难以仿造。

(4) 在大批量应用时，可显著降低成本。

而 PLD 也可以根据用户的需要完成特殊的功能，其中低密度可编程逻辑器件只能完成简单的逻辑功能，而高密度逻辑可编程器件(如 CPLD 和 FPGA)则可以实现非常复杂的系统功能。

与 ASIC 不同的是，PLD 是在市面上可以购买的，其实现功能可以在现场进行修改，而 ASIC 一旦生产就不能修改了。

FPGA 的主要用途有两个方面：

(1) 作为 ASIC 设计的快速原型系统。生产 ASIC 的费用非常昂贵，这其中包含了两方面的费用，一是设计 ASIC 的工具费用，另外就是 ASIC 中不可回归的工程费用，即通常所言的 NRE(Nonrecurring Engineering)费用。正如前面所述，一旦 ASIC 产生，就不能修改，设计中的任何微小的错误，都可能导致 ASIC 的失败，如果修改后重新投片，需要向 ASIC 厂家再支付一笔 NRE。因此，许多 ASIC 设计人员在流片之前，先用 FPGA 系统验证 ASIC 设计。与流片费用相比，购买 FPGA 的价格要低得多。另外，如果购买了某个厂家的 FPGA，FPGA 的供应商会提供相应的开发系统。从经济的角度讲，FPGA 的开发费用要小得多。但是，如果 ASIC 用量非常大，NRE 费用平摊到每个芯片上时，ASIC 单片价格就比购买 FPGA 的价格要低，因此，在大批量使用时，还是考虑用 ASIC 而不是 FPGA。

(2) 验证新算法的物理实现。很多应用场合，设计人员提出一些新的算法，为了验证算法硬件的可实现性和算法的正确性，通常也用 FPGA 作为实现的一种载体。

随着半导体工艺的进步，FPGA 厂家也在生产一些比较廉价的 FPGA，因此在使用数量不多的时候，也可以考虑用 FPGA 而不用 ASIC。此外，由于电子产品更新换代的速度加快，许多产品为了快速占领市场，也在大量使用 FPGA。

## 1.2 可编程逻辑器件的基本概念

FPGA 区别于 ASIC 的一个重要特点就是 FPGA 是现场可编程的。本小节简单介绍可编程器件的一些相关技术。

### 1．熔丝链技术

第一个在可编程逻辑器件中用到的技术是熔丝链技术。出厂时元件之间是通过熔丝连接的，如图 1.4(a)所示。图中所示的熔丝类似日常生活中所用的保险丝，当电流非常大时，熔丝便熔断。当熔丝熔断后，与熔丝相连接的输入被断开，由于上拉电阻的作用，输入端的逻辑值为逻辑高电平。设计工程师可以根据设计需要通过施加规定宽度和幅度的脉冲电流，熔断熔丝，完成相应的功能。例如当图 1.4(b)中 $F_{bt}$ 和 $F_{af}$ 熔断后，实现的逻辑功能是：y=a &!b。熔丝一旦熔断，便不可再恢复，因此，编程是一次性的。虽然熔丝链技术不是 FPGA 的编程技术，但它是 FPGA 编程技术发展的起点。

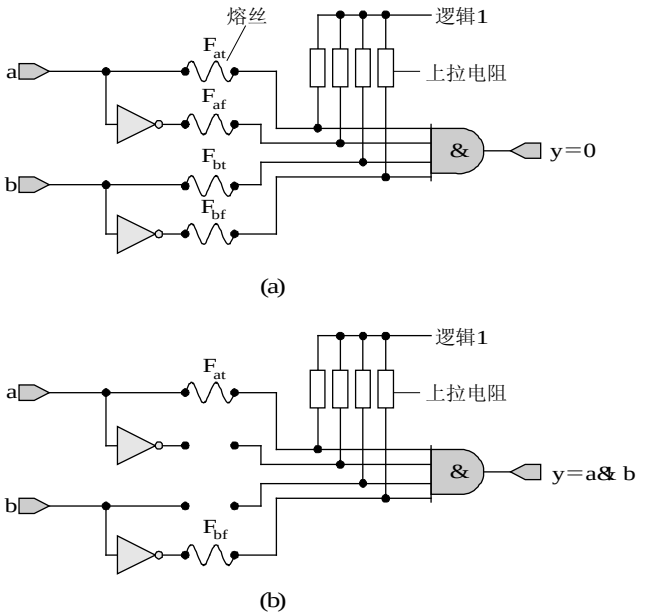



图 1.4　熔丝电路

(a) 通过熔丝连接的电路；(b) 熔丝熔断后的电路

### 2．反熔丝技术

一个普通的熔丝一般是构成一个连接，直到有一个过量的电流通过熔丝并烧断为止。反熔丝则正好相反，它在施加电压之前是断开的，而在施加了电压后形成导体。图 1.5 说明了反熔丝电路的形成。

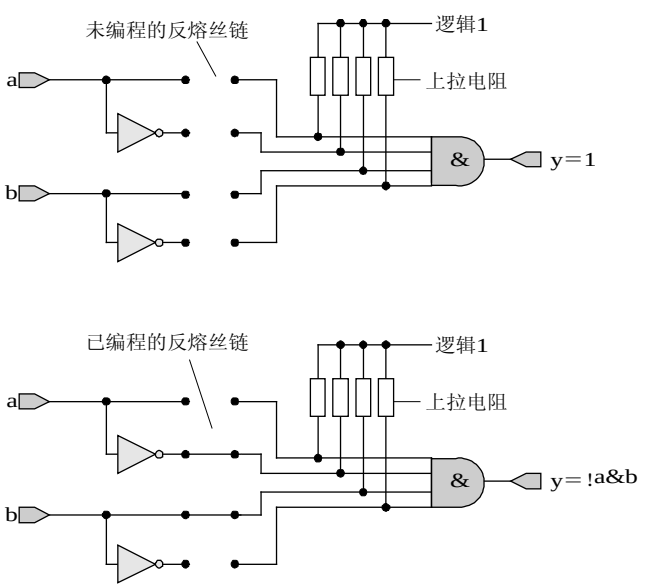



图 1.5　反熔丝电路

反熔丝技术的基本原理是在绝缘体分开的两个导体之间跨接一个反熔丝链，当在这个反熔丝链上施加一个大电压时，反熔丝链熔化，从而在两个导体之间产生电流。反熔丝隔开了 FPGA 上的互连线，而编程器烧断反熔丝后形成了永久的连接，这个过程不可逆转。因此，基于反熔丝的 FPGA 是不可再编程的。

反熔丝的 FPGA 器件的主要优点是：速度快、功耗低、非易失、抗辐射性好和保密性好。因此，它被应用到航天和军事系统中。它的缺点是不可重复设计，要求有编程器及专用的插座，封装都是 BGA；另外，由于反熔丝，所以它需要一定的编程电流。

### 3. 可编程只读存储器 PROM(Programmable Read Only Memory)

ROM 数据的存储是靠生产厂家特定的掩膜将数据写入存储器，如果用户需修改已存储的数据，则需要由生产厂家重新生产一套新的掩膜，经过加工测试然后再交回用户，这个周期比较长，对用户来讲，非常不方便。PROM 的出现解决了这一问题。PROM 允许通过专用的编程器将数据“烧录”到存储器中，这个过程叫做“编程”，烧录后的数据同样能保持断电后不丢失。

一次性 PROM 单元是由熔丝和二极管或三极管构成的，如图 1.6 所示。当大电流通过熔丝时，熔丝断开，从而切断原来的连接。PROM 产品在出厂时，所有存储单元均被加工成同一状态“0”(或“1”)。用户对 PROM 编程是逐字逐位进行的，根据需要写入的信息，按字线和位线选择某个存储单元，通过脉冲电流将该三极管的熔丝熔断，使该存储单元的状态被改变成与原状态相反的状态。熔丝一旦熔断，便不可再恢复，因此，编程是一次性的，PROM 的这种特性影响了它的使用，目前这种器件很少使用。

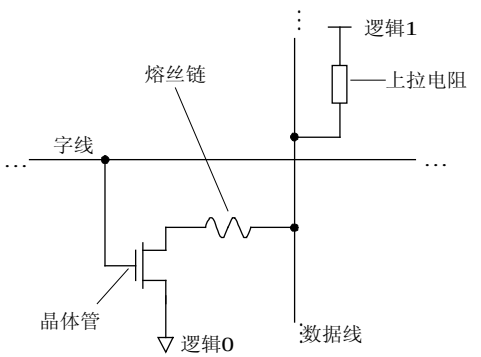



图 1.6　PROM 结构

### 4. EPROM 技术

由于 PROM 是一次性的，一旦编程完毕，其内容便不能再改变，如果数据烧录错误，那么 PROM 只能报废。Intel 公司在 1971 年首次开发了可擦除可编程的只读存储器 EPROM(Erasable PROM)，这种器件由于允许用户利用编码器对器件反复编程、擦除，因而得到了广泛的应用。这种器件通过施加高压信号进行编程，将器件置于紫外线下就可以擦除其内容。

EPROM 型的晶体管与标准的 MOS 管具有相同的结构，而与 MOS 不同的是，它增加了另外一个被称为浮栅的多晶硅，有时把这个晶体管称为浮栅雪崩注入 MOS 管，即 FAMOS 管。FAMOS 管的栅极全部被二氧化硅绝缘层包着，没有引出线，呈悬浮状，所以称作浮栅。图 1.7 给出了这两种晶体管的结构。

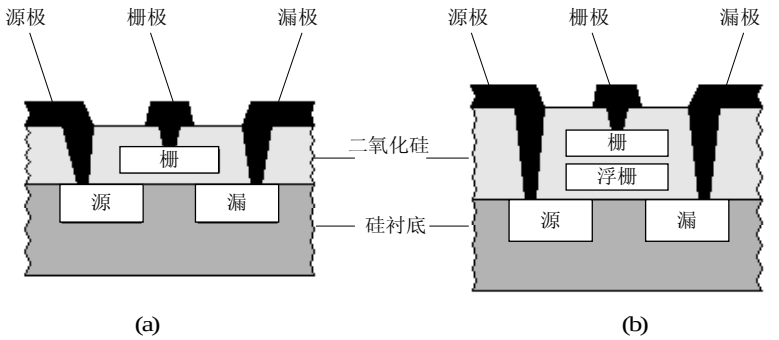



图 1.7　标准的晶体管和 EPROM 晶体管

(a) 标准 MOS 管；(b) EPROM 晶体管

在未编程时，浮栅管没有充电，因此，不影响晶体管的正常操作。为了编程，在栅极和漏极之间需要施加一个相对高的电压，使漏极与衬底之间的 PN 结击穿，雪崩击穿产生的高能电子穿过氧化物堆积在浮栅上，使浮栅管导通。对应 FAMOS 管的截止或导通，可使位线上输出的电平为高电平或低电平，以表示该存储单元存放的信息是“1”或“0”。由于浮栅被绝缘的二氧化硅包着，编程时堆积的电子没有放电回路，故电荷不会消失，信息能够长期保存。如果用紫外线照射 FAMOS 管，则浮栅上积累的电子由于吸收了足够的能量形成光电流而泄放，从而导电沟道消失，管子又恢复截止状态。为了使编程后能进行擦除和重写，在芯片的封装外壳装有透明的石英窗口。对编程好的 EPROM 要用不透光的胶纸将受光窗口封住，以免信息丢失。EPROM 的优点是其内容可以擦除后重新写入数据，即使写错了也无所谓，但其缺点是重新改写时需将器件拆下来在专门的编程器中进行改写。

### 5. $E^2$PROM 技术

EPROM 虽然具有可反复编程的优点，但 EPROM 只能整体擦除，不能将存储单元逐个独立地擦除，擦除操作也比较麻烦。而电可擦除可编程只读存储器 $E^2$PROM (Electronically-Erasable Programmable Read-Only Memory)克服了 EPROM 的这一不足之处。$E^2$PROM 的结构与 EPROM 相似，$E^2$PROM 的内部电路与 EPROM 电路类似，但其 FAMOS 的结构进行了一些调整，在浮栅上增加了一个遂道二极管(实际上是在浮栅与 N 型的衬底间形成一层薄薄的氧化层后形成的)，这个二极管用于电擦除存储单元。因此，与 EPROM 相比，$E^2$PROM 的面积比 EPROM 的大 2.5 倍。图 1.8 给出了 $E^2$PROM 元件的示意图。

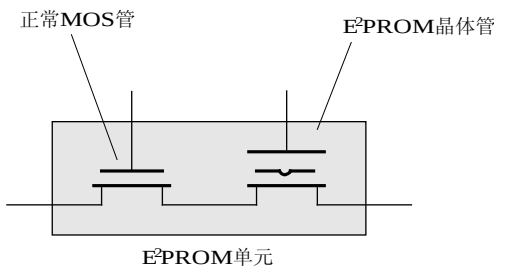



图 1.8　$E^2$PROM 的元件

$E^2PROM$ 不需要紫外光激发放电，即擦除和编程只需加电就可以完成了，且写入的电流很小。这种器件不仅工作电流小，擦除速度快，而且允许改写的次数大大高于 EPROM。

6. Flash 技术

Flash 是 $E^2PROM$ 的一种形式，也是由浮栅型场效应管构成的。但是它可以在极短的时间内完成多个存储单元的擦除和写入，而正常的 $E^2PROM$ 一次只允许对一个存储单元的内容进行擦除或写入，因此 Flash 的速度更快，效率更高。

7．静态 RAM 技术

常用的 RAM 有两类，一类是动态的 RAM，即 DRAM；另外一类是静态的 RAM，即 SRAM。在 DRAM 中，每个单元信息的存储是基于晶体管电容的充放电实现的。随着时间的推移，电容上存储的电荷会逐渐消失，因此，DRAM 需要周期性进行刷新操作以保持信息不丢失，DRAM 技术与 PLD 器件关系不大。

在 PLD 器件中，很多是采用 SRAM 技术编程的，SRAM 的主要特点是一旦某个值存储在 SRAM 单元中，则该值保持不变，直到外部对该单元进行了操作或者系统掉电为止。基于 SRAM 的 FPGA 在掉电后, 必须重新配置，因此需要外加一片专用配置芯片，在上电的时候，由这个专用配置芯片把数据加载到 FPGA 中。

在上面介绍的编程技术中，简单的 PLD 一般采用的是 EPROM/$E^2PROM$，复杂的 PLD 多采用的是 EPROM/$E^2PROM$ 和 Flash 技术，而 FPGA 采用各种编程技术。下面将各种技术在速度、密度、功耗和保密性等各个方面做一简单比较，见表 1.1。

表 1.1　各种技术性能的比较

| 分类 / 性能 | SRAM | Antifuse | Flash | EPROM/$E^2PROM$ |
|---|---|---|---|---|
| 速度 | 差 | 最好 | 差 | 中等 |
| 功耗 | 可变 | 好 | 最好 | 差 |
| 密度 | 中等 | 次好 | 最好 | 差 |
| 抗辐射 | 差 | 最好 | 中等 | 中等 |
| 重复编程 | 可以 | 不可以 | 可以 | 可以 |
| 保密性 | 中等 | 好 | 好 | 好 |
| 非易失性 | 差 | 好 | 好 | 好 |

## 1.3　简单可编程器件(SPLD)的结构

本节主要介绍简单可编程器件(SPLD)的结构。SPLD 包括 PROM、PLA、PAL 和 GAL。其结构由“与”阵列和“或”阵列组成，可以有效地实现“与或”形式的布尔函数。

1．PROM

PROM 是最早出现的 SPLD，它是由固定的 AND 阵列和一个可编程的 OR 阵列组成的。例如图 1.9 给出了一个三输入，三输出的 PROM 阵列构成示意图(不代表实际的电路图)。其

中，(a)图的左边实点表示的是固定连接，右边阵列的“*”表示可编程连接，这些可编程的连接可以通过基于熔丝连接、基于 EPROM 或基于 $E^2$PROM 的技术实现。与阵列的输入信号为互补缓冲输入，通过交叉点上的连接加到函数的与或表达式的乘积项中。或阵列的输入由与阵列的输出提供。这个与或阵列可以实现三输入的任意组合的逻辑函数。就图 1.9(b)而言，经过编程之后，它实现的逻辑函数是 w=(a&b), x=!(a&b)和 y=(a&b)^c。

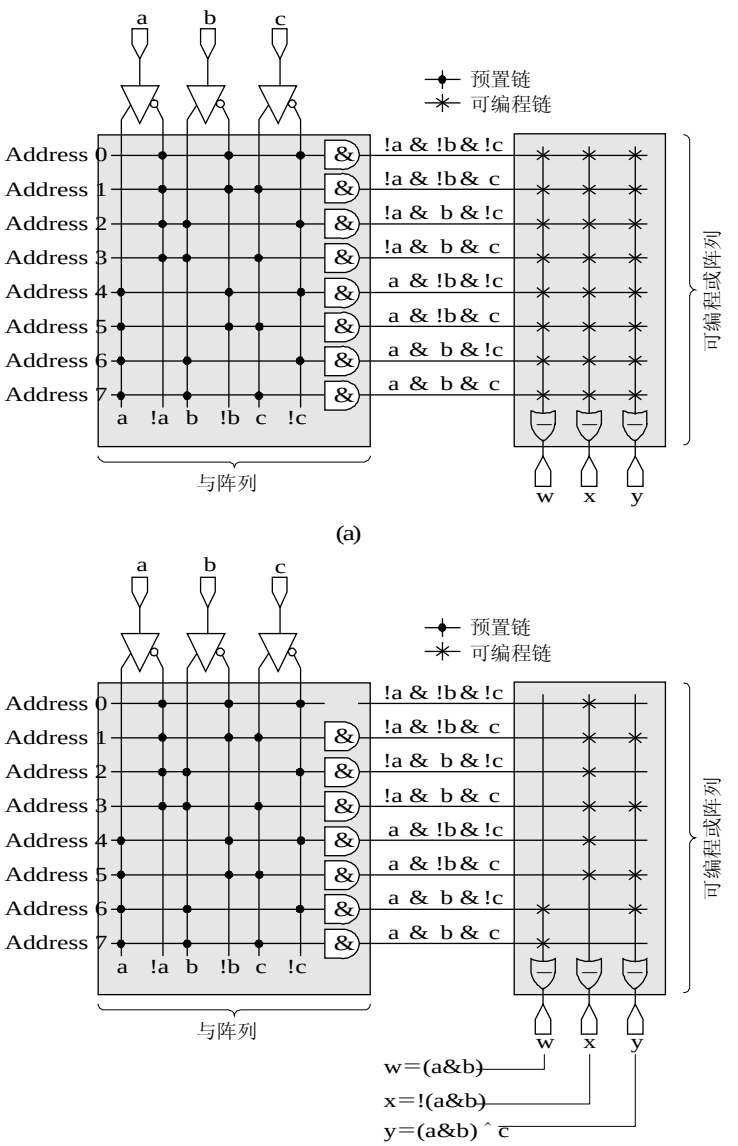



图 1.9　PROM 阵列

(a) 未编程的 PROM 阵列；(b) 实现特定函数的 PROM 阵列

### 2．可编程逻辑阵列

可编程逻辑阵列 PLA(Programmable Logic Array)是 20 世纪 70 年代中期出现的一种可

编程逻辑器件，它的出现解决了当时 PROM 在速度和输入方面的问题。PLA 是由大量的可编程与阵列和或阵列组成的。在与阵列中，通过对不同的连接点编程，可以对不同的信号进行与操作，经与阵列输出再连接到或阵列上；在或阵列中，与阵列输出的各项按不同的方式或起来，最后形成或阵列的输出。与阵列的输入端和输出端都有反相器，可以得到逻辑非的输出。与 PROM 不同的是，它们不能实现所有输入信号的逻辑组合。通过对与逻辑和或逻辑的编程，可以实现特定的逻辑电路。图 1.10(a)、(b)分别给出了三输入三输出的未编程和已编程的 PLA 结构。图 1.10(b)实现的逻辑函数为：w=(a&c)|(!b&!c)，x=(a&b&c)|(!b&!c)和 y=(a&b&c)。由于 PLA 器件的价格比较昂贵，编程复杂，资源利用率低等缺陷，因此没有得到广泛的应用。

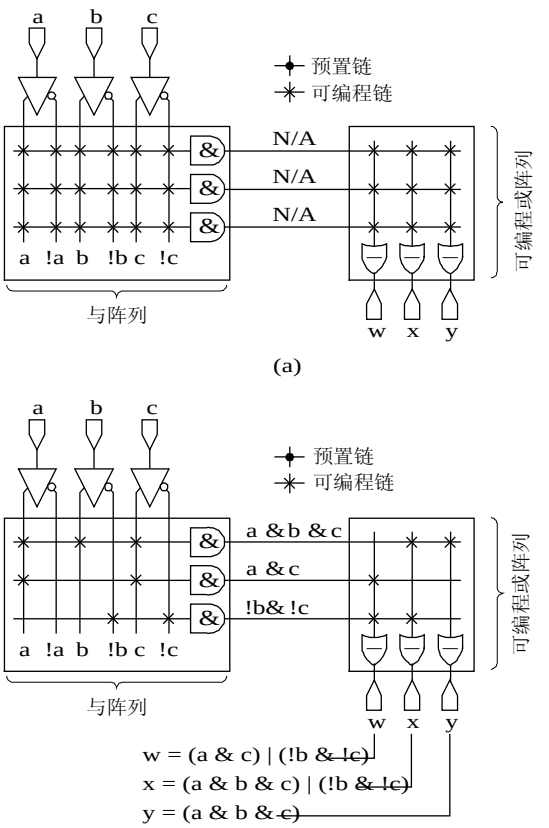



图 1.10　PLA 结构

(a) 编程前的 PLA；(b) 编程后的 PLA

### 3．可编程阵列逻辑和通用阵列逻辑

可编程阵列逻辑 PAL(Programmable Array Logic)是 20 世纪 70 年代末推出的一种器件。它是 PLA 的一种变形，与 PLA 不同的是它的或阵列是固定的。PAL 中的与阵列对输入信号进行与操作，与阵列中每一个交叉点为编程元素，使得直线与水平线要么连接，要么断开。因此，这就产生了一个“乘积项”，然后再把乘积项相或。通过这些组合，实现不同的逻辑。图 1.11 给出了编程前后三输入/输出的 PAL 阵列。图 1.11 实现的函数是：x= (a&c)|(!b&!c)，y= (a&b&c)|(!b&!c)和 z = (a&b&c)。

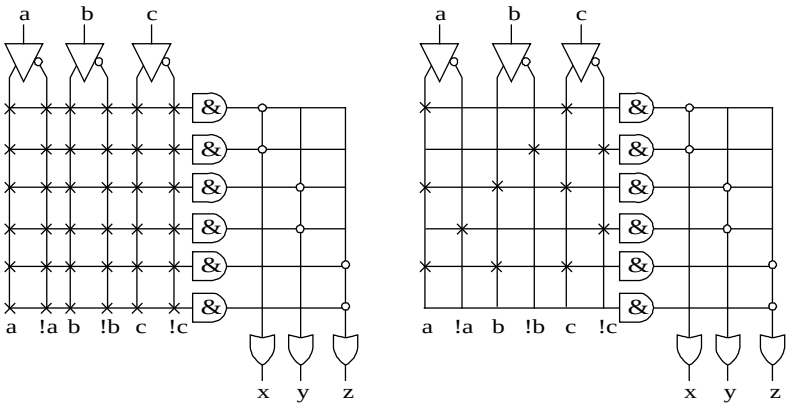



图 1.11　PAL 结构

但 PAL 器件也存在一些缺点，一是 PAL 采用熔丝工艺，一旦编程后便不能改写；二是 PAL 的输出是固定的，不能编程，芯片的型号一旦选定，输出结构也就固定了；三是型号太多(一种输出结构就是一种型号)，设计者要根据需要选择不同输出结构的器件。这些缺点给用 PAL 设计逻辑的用户带来了不便。

通用阵列逻辑 GAL(Generic Array Logic)器件是 20 世纪 80 年代中期面世的。它是在 PAL 基础上发展起来的一种可编程器件。它采用了高速电可擦 CMOS 工艺，具有可电擦写、可重复编程和可设置加密位等特点。GAL 与 PAL 的最大差别是 GAL 的输出结构可以由用户定义，是一种可编程的输出结构。一种型号的 GAL 器件可以对几十种 PAL 器件做到全兼容，GAL 的器件几乎完全取代了 PAL，并可以取代大部分中小规模的数字集成电路，因而获得了广泛的应用。

## 1.4　高密度可编程逻辑器件

前面介绍的低密度可编程逻辑器件，结构简单，对开发软件要求低，但是由于它们规模较小，其寄存器资源、I/O 管脚数目和时钟资源都有限，因此只能实现较简单的设计。随着半导体工艺的快速发展，可编程逻辑器件在密度、速度和体系结构上都较低密度可编程逻辑器件有了较大的提高。FPGA 和 CPLD 是 PAL 和门阵之间的桥梁，它们具有门阵和 PAL 的所有优点。CPLD 的工作速度与 PAL 一样，但是比 PAL 复杂，可以实现复杂的逻辑功能。而 FPGA 在规模和结构上都接近于门阵，但是它是可编程的。

### 1.4.1　复杂可编程逻辑器件 CPLD

虽然各个生产厂家在生产 CPLD 时有所不同，但是它们有三个主要的组成部分：输入/输出功能模块、宏单元和互连矩阵。图 1.12 是 Altera MAX7000 的结构。在这个结构中，通过可编程互连阵列 PIA(Programmable Interconnect Array)将多个逻辑阵列模块 LAB(Logic Array Block)连接在一起，每个 LAB 包含 16 个宏单元，全局总线可以直接连接到专用的 I/O 管脚和宏模块上。

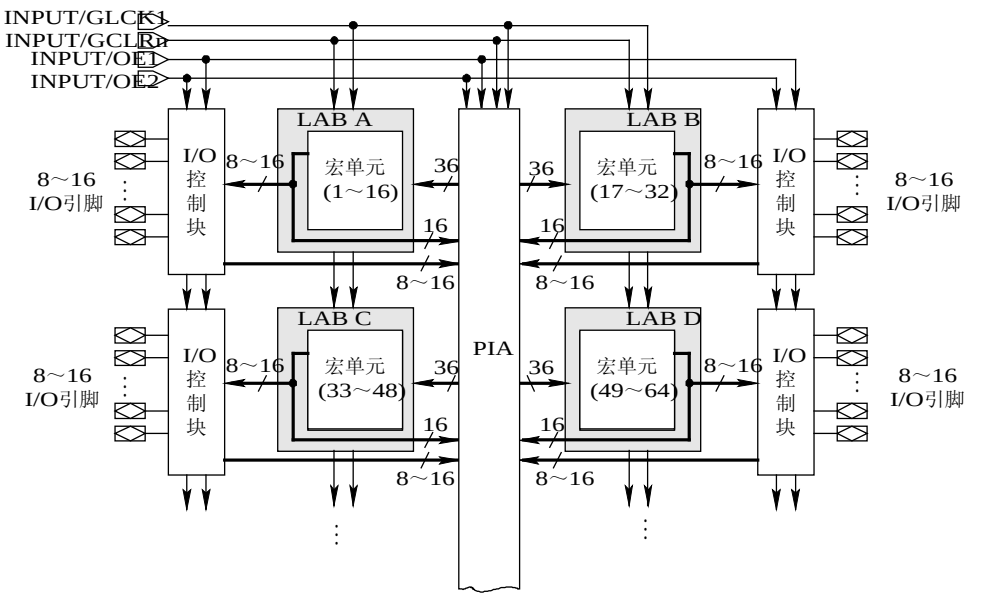



图 1.12　一种 CPLD 结构示意图

### 1．宏模块

宏模块的基本结构与 PLD 类似，由它来实现基本的逻辑功能。一个典型的宏模块如图 1.13 所示。与 PAL 和 GAL 类似，宏模块也是通过与或逻辑阵列实现组合逻辑的。与阵列由图中左边相互交叉的连线表示，与阵列的输入可以来自 I/O 模块或另外一个功能模块或者同一模块的反馈。每一个交叉点都是一个可编程的熔丝，如果导通就实现与逻辑。后面的乘积项是一个逻辑阵列，两者一起组成一个组合逻辑。图右侧是一个可编程的 D 触发器，

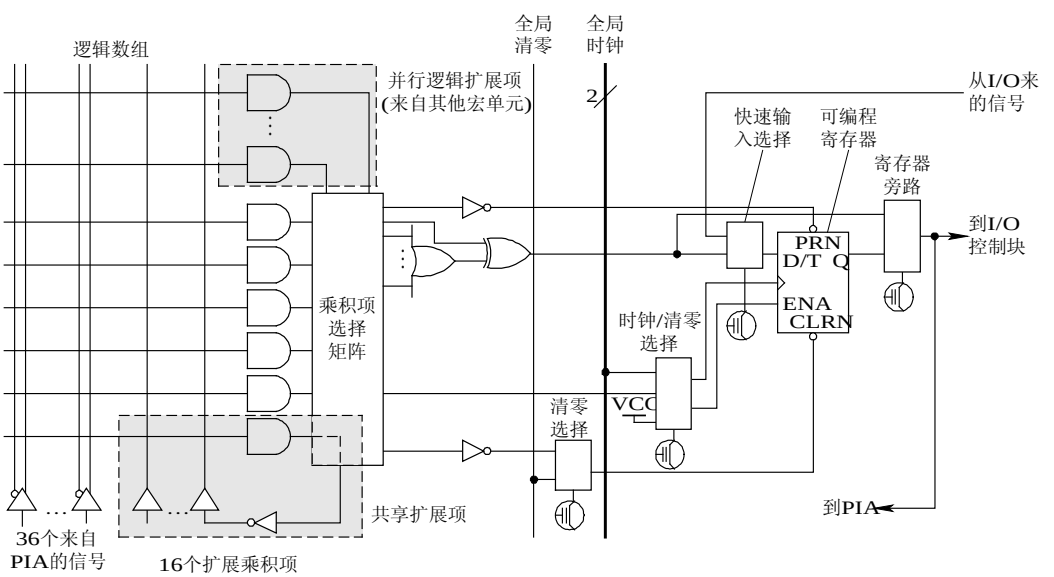



图 1.13　宏模块内部结构(MAX7000 系列)

它的时钟、清零端都是可选择的。可以使用专用的全局清零和全局时钟信号，也可以使用内部逻辑(乘积项阵列)产生的时钟和清零信号。如果不需要触发器，也可以将此触发器旁路，信号直接输给可编程互连阵列 PIA 或输出到 I/O 脚。

2．互连矩阵 PIA

CPLD 内部有一个规模很大的可编程开关矩阵 PIA。PIA 可以对 CPLD 中的任何信号进行互连，但只有 LAB 需要的信号才被实际地从 PIA 连接到 LAB 上。CPLD 开关矩阵的一个重要优点就是通过芯片的延时是固定的，设计人员通过计算 I/O 模块、功能模块和矩阵开关的延时就可以确定任何信号的延时。

3．I/O 模块

I/O 模块的功能是用合适的电平(如 TTL/ECL/CMOS/PECL)把内部的信号驱动到 CPLD 的外部引脚。I/O 控制块可以把一个 I/O 管脚配置成输入、输出或双向类型。

## 1.4.2 现场可编程门阵列 FPGA

FPGA 器件的结构非常类似于门阵 ASIC，但是，FPGA 芯片没有任何定制的掩膜层，设计人员可以对其进行设计输入和仿真，最后用专用软件将设计转换成一串二进制比特，形成配置文件。这个配置文件描述了需要完成设计的 FPGA 芯片的连接关系。最后，通过计算机将配置文件下载到 FPGA 或配置芯片上对其进行配置。

每一个 FPGA 生产厂都有自己的 FPGA 体系结构，但是所有厂家的 FPGA 结构中都包含了如图 1.14 所示的三个基本块，即可配置的逻辑块 CLB、可配置的 I/O 模块和可编程互连资源。另外，在 FPGA 中有一个时钟电路用于驱动时钟信号到每个 CLB 中的触发器，还有一些其他逻辑资源如存储器、译码器等。随着 FPGA 的发展，各个厂家所提供的资源越来越丰富，如数字信号处理器 DSP、锁相环 PLL 等。

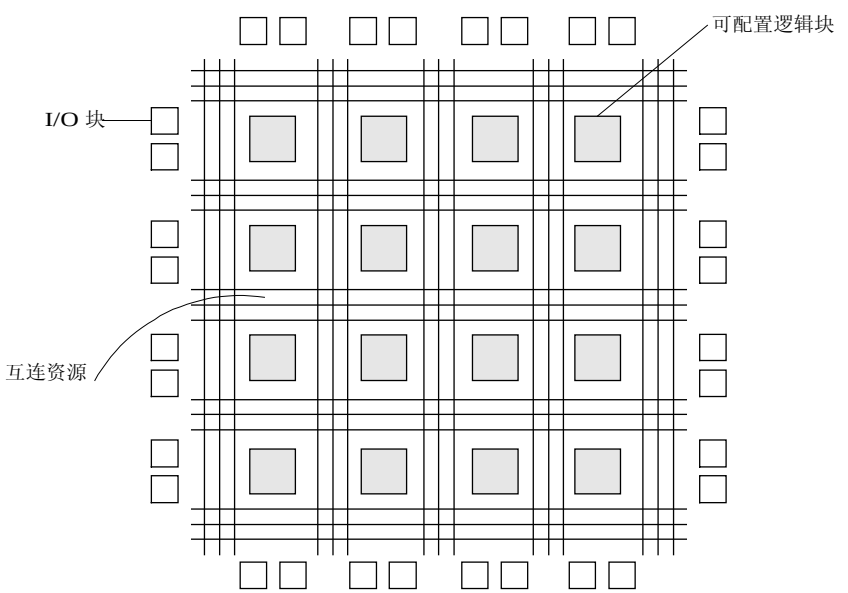
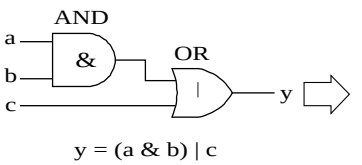



图 1.14　FPGA 的结构

### 1. 可编程逻辑块

基本的可编程逻辑块有两种，一种是基于多路选择器的可编程逻辑块，另外一种是基于查找表结构的可编程逻辑块。在今天的 FPGA 市场上，几乎都是基于查找表结构的 FPGA。

查找表是一种按某种方式排列的真值表，其输入信号用于作为查找表的索引。查找表的各个单元包含了不同输入信号组合所计算的布尔函数值。例如一个三输入函数 y = (a&b)|c 对应的查找表如图 1.15 所示。




真值表

| a | b | c | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

图 1.15　查找表

第一个查找表的 FPGA 是三输入的，后来人们尝试过四输入，五输入和六输入查找表或几种输入相结合的 FPGA。但是由于逻辑综合工具的现状，目前大多数的 FPGA 采用四输入查找表结构。

可配置逻辑块 CLB 包含了 FPGA 的可编程逻辑。各个厂家的可编程逻辑块名称都不相同。例如，Xilinx 的可编程逻辑块被称为可配置逻辑块 CLB，而 Altera 的则称为逻辑阵列块 LAB，其他的厂家也各有其名称。它们在结构上略有不同，但在概念上是类似的。

### 2. 可配置输入/输出模块

可配置的输入/输出模块 IOB 为芯片外部封装管脚和内部逻辑提供连接接口。每个 IOB 控制一个封装管脚，可配置成输入、输出或双向口。

### 3. 可编程的互连资源

通过可编程资源可以将 CLB 和 CLB，CLB 和 I/O 相互连接起来。在 FPGA 中，一般有三类连线资源。第一类为直线或短线，通过直线每个 CLB 可连接到与它相邻的 CLB 上。另外一类连线资源是长线，这些长线可以连接物理位置上彼此相距非常远的 CLB，一般是从一个 CLB 模块，直接连接到另外一个 CLB 模块中，而中间并不与开关矩阵相连接。第三类资源由经纬分段连线组成，这些连线到达开关矩阵之前经过了多个 CLB。开关矩阵允许信号从一个开关矩阵到达一个或多个开关矩阵，最后到达 CLB。可编程的互连资源结构如图 1.16 所示。

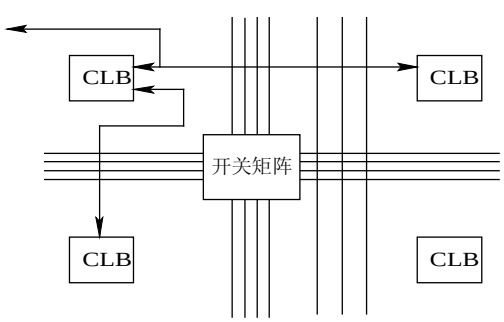



图 1.16　FPGA 的可编程互连资源

除了基本结构之外，FPGA 中还提供各种可用的嵌入式资源，主要有：

(1) 嵌入式 RAM。在实际应用中需要使用大量的存储单元，所以现在的 FPGA 中内嵌了许多 RAM 块，根据 FPGA 结构的不同，这些 RAM 块可能被放置在芯片相对独立的位置，或者以列的形式排列。不同的 FPGA 内部的 RAM 的大小也不一样，一般包含了多块 RAM，这些 RAM 可以组合使用，形成更大的 RAM。RAM 通常可以实现单端口、双端口、FIFO 等存储功能，也可以实现算术逻辑和自动机等功能。

(2) 嵌入式的乘法器和加法器等。在一些应用中(如图像、数字信号处理等)，经常要用到一些算术操作，这些操作如果用可编程逻辑块经过互连实现，则速度非常慢。因此，有些 FPGA 中包含了嵌入式硬核如乘法器、加法器等，这些模块一般在嵌入式 RAM 的周围，如图 1.17 所示。

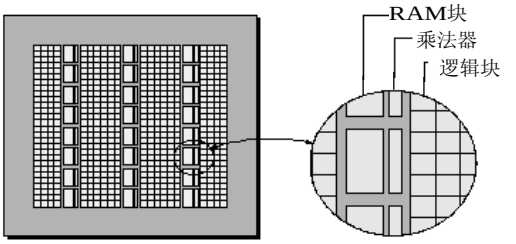



图 1.17　FPGA 中嵌入式资源

(3) 嵌入式处理器核。微处理器在电子系统中扮演着重要的角色，随着 FPGA 密度和速度的提高，许多高端的 FPGA 中都包含有嵌入式处理器核。在 FPGA 中存在两种嵌入式的处理器：一种是微处理器核以相对独立模块的形式放置在 FPGA 中，也可以是微处理器和 FPGA 两个芯片，最后封装成一个芯片。另外一种方式是微处理器核直接嵌入在 FPGA 中，与其他部分融合在一起。上面说的这两种处理器一般都是厂家物理上实现好的，另外一种是通过配置一组可编程逻辑形成微处理器，我们把这种微处理器称为软核，软核比硬核的速度慢。例如 Altera Cyclone 系统提供微处理器软核 Nios Ⅱ。关于 Nios Ⅱ，本书有专门的章节进行介绍。嵌入式的微处理器为系统设计工程师带来许多好处，可以简化系统的设计，减小系统板的体积并提高性能。

### 1.4.3　CPLD 和 FPGA 的区别

CPLD 和 FPGA 都是由可编程的逻辑单元、I/O 块和互连资源三个部分组成的。I/O 块的功能基本相同，而其他两个部分则有所区别。

除了 Actel 的 FPGA，其他的 FPGA 和 CPLD 的逻辑单元的结构都是由与阵列、或阵列和可配置的输出宏单元组成。FPGA 的逻辑单元是小单元，每个单元只有 1 到 2 个触发器，其输入变量通常只有几个，采用查表的结构方式。这样的结构占用的芯片面积小、速度高，每个 FPGA 的芯片上能集成的单元数目多，但是每个逻辑单元实现的功能少，因此，我们把 FPGA 也称为细粒度结构。实现一个复杂的逻辑函数时，需要用到多个逻辑单元，输入到输出的延时大，互连关系比较复杂。

CPLD 的逻辑单元是大单元，通常其输入变量的数目可以是 20～28 个，我们称之为粗粒度结构。因为变量多，所以只能采用 PAL 的结构方式。这样一个单元内可以实现复杂的逻辑功能，因此实现复杂的逻辑函数时，CPLD 的互连关系比较简单，一般通过总线就可以实现互连。CPLD 的大单元使用的互连矩阵，总线上任意一对输入端之间的延时相等，因此，其延时是可预测的。而 FPGA 的小单元使用直接连接、长线连接和分段连接等不同类型的互连方式，互连结构复杂，延时不易确定。

如何在 CPLD 和 FPGA 之间进行选择呢？实际上主要还是取决于设计项目的需要。下面对 FPGA 和 CPLD 的一些主要特性做一简要的比较，以供参考(见表 1.2)。

表 1.2　CPLD 和 FPGA 的比较

| 主要特性 | CPLD | FPGA |
|---|---|---|
| 结构 | 类似 PAL | 类似门阵 |
| 速度 | 快、可预测 | 取决于应用 |
| 密度 | 低等到中等 | 中等到高密度 |
| 互连 | 纵横连接方式 | 路径选择方式 |
| 功耗 | 高 | 低 |

### 1.4.4　FPGA/CPLD 厂家简介

FPGA 由于开发周期短、功能强、可靠性高和保密性好等特点广泛地应用在各个领域。FPGA 应用领域的不断扩大和半导体加工工艺的不断进步，都促使了 FPGA 的快速发展，其中 Altera 和 Xilinx 公司的产品占到整个 FPGA/CPLD 市场的 80%。Actel 虽然规模较小，但是由于它提供了反熔丝 FPGA，保密性和可靠性非常好，因此，在航空和军品领域占有很大的市场。

(1) Altera 公司：它是世界上最大的 CPLD/FPGA 供应厂家之一，是结构化 ASIC 的首创者。其产品包括 FPGA 系列、CPLD 系列和结构化 ASIC 系列。FPGA 系列有：Stratix Ⅱ，Stratix，Cyclone Ⅱ，Cyclone，StratixGX，APEX Ⅱ，APEX20K，Mercury，FLEX10K，ACEX1K，FLEX 6000；CPLD 系列包括 MAX7000，MAX3000A；结构化 ASIC 包括 hardcopy Stratix 系列和 hardcopy Flex20K 系列。Altera 的开发集成环境是 MAX+PLUS Ⅱ 和 Quartus Ⅱ，其中 Quartus Ⅱ 是 Altera 最新推出的集成环境，可与第三方软件工具无缝连接，支持 Altera 所有产品的开发。

(2) Xilinx 公司：它是 FPGA 的发明者。其产品种类较多，主要有 XC9500/4000，Coolrunner(XPLA3)，Spartan，Virtex 等系列。其中 2002 年推出的 Virtex-Ⅱ Pro 系列是 Xilinx 公司自 1984 年发明 FPGA 以来所推出的最重要的产品之一，支持芯片到芯片、板到板、机箱到机箱，以及芯片到光纤的应用，将可编程技术的使用模式从逻辑器件层次提升到系统一级。Xilinx 的软件集成环境是 Foundation 和 ISE，其中 ISE 是最新推出的，它将逐步取代 Foundation。另外，Xilinx 公司还提供免费的开发软件 IEWEBPACK，其功能比 ISE 少一些，可直接从网上下载。

(3) Actel 公司：其产品包括反熔丝和 Flash 两类 FPGA。其中 Flash 产品包括 ProASIC$^{plus}$ 和 ProASIC；基于反熔丝的产品包括：Axcelerator SX-A/SX EX 和 MX。

Actel 的产品由于具有抗辐射、耐高温、功耗低、速度快、保密性强等特点，因此被广泛应用在军品和宇航领域。Actel 软件集成环境是 Libero，集成了针对 FPGA 结构而开发的 Syncity 软件，综合效率非常高。

(4) Lattice 公司：Lattice 是 ISP(In-System Programmability)技术的发明者，ISP 技术极大地促进了 PLD 产品的发展，与 Altera 和 Xilinx 相比，其开发工具比 Altera 和 Xilinx 略逊一筹。该公司的中小规模 PLD 比较有特色，大规模 PLD 的竞争力还不够强(Lattice 没有基于查找表技术的大规模 FPGA)，主要产品有 ISPLSI2000/5000/8000，MACH4/5 等。

## 1.5 基于 FPGA 的设计流程与设计方法

### 1.5.1 基于 FPGA 的设计流程

什么是基于 FPGA 的设计？本书所讲的基于 FPGA 的设计是指用 FPGA 器件做载体，借助于 EDA(Electronic Design Automation，设计自动化)软件工具，实现有限功能的数字系统设计。FPGA 的设计过程就是从系统功能到具体实现之间若干次变换的过程。FPGA 设计需要按照一定的设计流程进行，在流程的某些环节，需要遵循一定的原则和规定。为了对基于 FPGA 的设计有一个初步的认识，我们简要介绍一下通用的 FPGA 设计流程。

FPGA 的设计流程大体上分为系统规范、模块设计、设计输入、功能仿真(前仿真)、综合、布局布线、时序验证(后仿真)、配置下载等七个步骤，设计流程如图 1.18 所示。下面分别介绍各个设计阶段的主要任务。

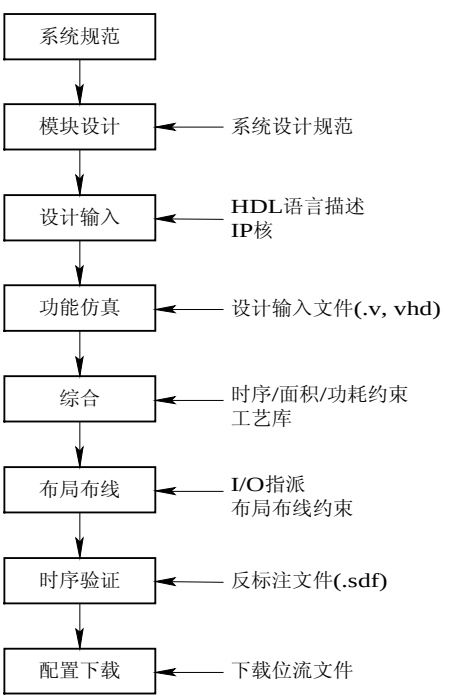



图 1.18 FPGA 的设计流程

### 1．系统规范

系统规范阶段是整个项目最有创造性的阶段。它描述项目完成的功能，确定设计的总体方案，平衡各个方面的因素，对整个项目有一个初步的规划。在系统设计阶段，根据对设计面积、功耗、I/O 和 IP 核使用等的估算，确定所使用的目标芯片和设计工具。

### 2. 模块设计

在制定完系统规范后，根据系统功能，采用自顶向下的方法，逐步细化，将系统划分为可实现的设计模块。这些模块之间存在着一定的层次关系，每个模块完成相对独立的功能。

### 3. 设计输入

设计输入是指将模块设计阶段定义好的模块借助于一定的设计输入手段转换为 EDA 工具能接受的信息格式。目前主要的设计输入手段有：高级硬件描述语言 HDL(包括 Verilog/VHDL)和原理图。HDL 语言支持不同层次的描述，不依赖于 FPGA 厂家的工艺器件，便于修改。它可以用任意的文本编辑器作为输入平台，在状态机、控制逻辑、总线功能方面较强。原理图输入法具有图形化强、直观等特点。

### 4. 功能仿真

设计输入后，经 HDL 编译器检查没有语法错误后，就可以对设计进行验证了。这里的验证是指通过仿真软件验证其功能是否符合由步骤 1 所制订的规范，称这一阶段的验证为功能仿真或行为仿真。目前，仿真工具比较多，其中 Cadence 公司的 NC-verilog，Synopsys 公司的 VCS 和 Mentor 公司的 Modelsim 都是业界广泛应用的仿真工具。

### 5．综合

综合实际上是根据设计功能和实现该设计的约束条件(如面积、速度、功耗和成本等)，将设计描述(如 HDL 文件、原理图等)变换成满足要求的电路设计方案，该方案必须同时满足预期的功能和约束条件。对于综合来说，满足要求的方案可能有多个，综合器将产生一个最优的或接近最优的结果。因此，综合的过程也就是设计目标的优化过程，最后获得的结构与综合器的性能有关。这个阶段产生网表，供布局布线使用，网表中包含了目标器件中的逻辑元件和互连的信息。FPGA 综合工具有 Synopsys 公司的 CompilerⅡ FPGA，Synplicity 公司的 Synplify 等。

### 6. 布局布线

这一步骤就是要完成实现方案(网表)到实际目标器件(FPGA 器件)的变换。根据设计者指定的约束条件(如面积、延时、时钟等)、目标器件的结构资源和工艺特征，将电路方案中的逻辑元件分解布局，用作拓扑目标器件的连线资源，实现布线连接。在布局布线过程中，时序信息形成产生反标注文件，供给后续的时序仿真使用，同时还产生 FPGA 配置时需要的位流文件。FPGA 设计中的布局布线工具主要由 FPGA 厂商提供，种类比较多，本书介绍 Altera 公司的 Quartus Ⅱ集成环境中自带的布局布线工具。

### 7．时序验证

在布局布线后，提取有关的器件延迟、连线延时等时序参数(这些信息在反标注文件中)，

在此基础上进行的仿真称为后仿真，也称时序验证，它是接近真实器件运行的仿真。时序验证的目的是为了检查设计中是否有时序上的违规。FPGA 中同步电路的验证采用静态时序分析实现，异步电路的验证则需要运行特殊仿真激励确认。仿真工具可以用前仿真所用的工具，而静态时序分析工具一般也由各个 FPGA 厂家的 FPGA 集成环境自带。

**8．配置下载**

配置下载是在功能仿真与时序仿真正确的前提下，将布局布线后形成的位流文件通过下载工具下载到具体的 FPGA 芯片中，这个过程也叫 FPGA 编程(配置)。将位流文件下载到 FPGA 器件内部后，就可以将 FPGA 和其他芯片构成的系统进行物理测试，当得到正确的测试结果后就证明了设计的正确性。下载软件也是由各个 FPGA 厂家提供。

### 1.5.2 自顶向下和自底向上的设计方法学

随着微电子技术的快速发展，深亚微米的工艺可以使一个芯片上集成数以千万乃至上亿只的晶体管，单片上就可以实现复杂系统，即所谓的片上系统。在这种情况下，传统的自底向上的设计方法学已经不可能适应现在的设计要求，而自顶向下的设计方法学已经成为设计界的主流设计方法学。

在 EDA 工具出现以前，人们采用自底向上的设计方法设计集成电路。在这种设计方法学中，功能设计是自顶向下的，即提出所设计电路要完成的功能，然后进行行为级描述，RTL 级设计、逻辑设计和版图设计。具体的实现过程则正好相反，从最底层的版图开始，然后是逻辑设计，直到完成所需实现的功能。

这种设计方法的缺点是：效率低，设计周期长，设计质量难以保证，适用于小规模的电路设计。

自顶向下的设计方法学是和 EDA 工具同步发展起来的，借助于 EDA 工具可以实现从高层次到低层次的变换，无论是功能设计和具体实现都是自顶向下的。FPGA 设计流程就是典型的自顶向下设计方法学(见图 1.18)的一个体现。在这个设计流程中，设计人员从制订系统的规范开始，依次进行系统级设计和验证、模块级设计和验证、设计综合和验证、布局布线和时序验证，最终在载体上实现所设计的系统。

自顶向下的设计方法学的优点是显而易见的，在整个设计过程中，借助于 EDA 仿真工具可以及时发现每个设计环节的错误，进行修正，最大限度地不把错误带入到后续的设计环节中。另外，由于在自顶向下的设计方法学中用硬件描述语言作为设计输入，改变了传统的电路设计方法，是 EDA 技术的一次巨大进步。它可以在系统级、行为级、寄存器传输级、逻辑级和开关级等五个不同的抽象层次描述一个设计，设计人员可以在较高的层次寄存器传输级描述设计，不必在门级原理图层次上描述电路。由于摆脱了门级电路实现细节的束缚，因而设计人员可以把精力集中于系统的设计与实现方案上，一旦方案成熟，那么就可以以较高层次描述的形式输入计算机，由 EDA 工具自动完成整个设计。这种方法大大缩短了产品的研制周期，极大地提高了设计的效率和产品的可靠性。

### 1.5.3 基于 IP 核的设计

由于芯片的集成度变得越来越高，因而设计的难度也变得越来越大。设计成本事实上

主导了芯片的价格。如何提高设计效率，最大限度地缩短设计周期，使产品快速上市，这给设计人员提出了非常高的要求。采用他人的成功设计是解决这个问题的有效方法。

所谓设计重用实际上包含两个方面的内容：设计资料重用和生成可被他人重用的设计资料。前者通常被称为 IP 重用(IP Reuse)，而后者则涉及到如何去生成 IP 核。设计资料内不仅仅包含一些物理功能和技术特性，更重要的是包含了设计者的创造性思维，具有很强的知识内涵。这些资料因而也被称为具有知识产权的内核(Intellectual Property Core)，简称 IP 核，它们通常可以实现比较复杂的功能，且已经经过验证，可以被设计人员直接采用。

一般来讲，IP 核有三种表现形式：软核(Soft-Core)、固核(Firm-Core)和硬核(Hard-Core)。

(1) 软核：它以硬件描述语言 Verilog 或 VHDL 语言代码的形式存在，软核功能的验证通常是通过时序模拟。软核不依赖于任何实现工艺或实现技术，具有很大的灵活性。设计者可以方便地将其映射到自己所使用的工艺上去，可重用性很高。

(2) 硬核：它以集成电路版图(Layout)的形式提交，并经过实际工艺流片验证。显然，硬核强烈地依赖于某一个特定的实现工艺，而且在具体的物理尺寸，物理形态及性能上具有不可更改性。

(3) 固核：处于软核和硬核之间的固核以电路网表(Netlist)的形式提交，并且通常采用硬件进行验证。硬件验证的方式有很多种，比如可以采用可编程器件(如 FPGA，EPLD)进行验证，采用硬件仿真器(Hardware Emulator)进行验证等。

不同的 FPGA 厂商在其不同的 FPGA 系列中都具有嵌入式的 IP 核，这些核可能是硬核(如锁相环)，也可能是可配置的软核。用户可以根据设计需求，直接使用这些 IP 核，借助于这些 IP 核，用户可以加快设计进度，提高设计效率和设计可靠性。

## 1.6 EDA 技术简介

在上面介绍的 FPGA 流程中，许多步骤都是借助于 EDA 工具完成的，如综合、布局布线、仿真等阶段都有相应的 EDA 工具。FPGA 的广泛应用和推广与 EDA 工具的迅速发展密不可分。

电子设计自动化(EDA)技术是指以计算机为基本工作平台完成电子系统自动设计的技术。EDA 工具是融合了图形学、电子学、计算机科学、拓扑学、逻辑学和优化理论等多学科的研究成果而开发出来的软件系统。借助于 EDA 工具，电子设计工程师可以利用计算机完成包括产品规范定义、电路设计和验证、性能分析、IC 版图或 PCB 版图在内的整个电子产品的开发过程。EDA 工具的发展极大地改变了电子产品的设计方法、验证方法、设计手段，大幅度地提高了电子产品的设计效率和可靠性。

EDA 工具最早是在 20 世纪 70 年代初出现的，那时的集成电路也刚出现不久。当时的集成电路比较简单，只能完成简单的逻辑功能，如前面所提及的 TI 公司的 7400 系列。这些 IC 从功能设计到最后版图设计的整个过程都是通过手工设计完成的。最大的问题就是人们无法对非线性元件的行为进行精确的预测。因此，在设计规模增大后，往往第一个原型芯片不能很好地工作，需要对设计进行多次修改，直到设计出的 IC 完全符合要求为止。为了解决这个问题，加州 Berkeley 大学推出了计算机仿真程序 SPICE，这个程序可以说是 EDA

技术的基础。SPICE 是非常重要的仿真工具，现在还是模拟电路设计中不可缺少的工具之一。SPICE 的出现极大地提高了电路设计的效率，它可以仿真包括非线性元件在内的电路网络，并可预测电路随时间变化的频率特性。

CAD(Computer Aided Design，计算机辅助设计)工具最初是为机械和结构工程而开发的，但是很快人们便发现这些工具可用于任意的几何设计。利用 CAD 工具，设计人员可以方便地输入、修改和存储多边形数据，然后通过机械光系统或电子束将这些多边形数据转换成物理图像(即所谓的掩模)。

20 世纪 70 年代，除了仿真工具外，其他比较重要的 EDA 工具是用于检查版图几何尺寸的设计规则检验(DRC)工具和版图参数提取工具，这些物理设计工具的出现将设计人员从繁琐而费时的后端设计中解放出来，极大地提高了 IC 设计的效率。

20 世纪 80 年代，半导体技术发展很快，已经可以在一个芯片上集成上万门的电路，70 年代的 EDA 工具已不能适应这么大规模的 IC 设计。所幸的是这个时期的计算机技术也有很大的发展，高性能的工作站和软件图形界面的开发为 EDA 工具的发展奠定了很好的基础。这个阶段主要的 EDA 工具有以下几种：

(1) 原理图编辑器：最初人们用网表描述一个设计，网表中包含了一个设计所有的元件和元件之间的互连关系。由于网表的数据量小但却包含了设计的所有信息，因此非常适合于存储，但是网表描述形式不利于设计人员对电路的理解。20 世纪 80 年代推出了原理图编辑器，这种编辑器一经推出，便由于其直观、易于理解而受到设计人员的欢迎。

(2) 自动布局布线工具：指自动确定芯片上元件的位置和元件之间互连的工具，该工具的出现极大地提高了布线的效率。

(3) 逻辑仿真工具：这类仿真器将信号离散化，内建延时模型，根据电路自动计算延时，其仿真的速度远远高于 SPICE。

这个时期的其他 EDA 工具包括逻辑综合工具(允许用户将网表映射到不同的工艺库中)、印刷电路板布图等工具，使得设计自动化的程度进一步提高, 实现从设计输入到版图输出的全设计流程的自动化。

20 世纪 80 年代，一些研究人员提出从设计描述开始，如布尔表达式或寄存器传输级的描述，自动完成集成电路设计过程中的所有步骤，直到最后生成版图的设想。少数的几所大学在逻辑设计自动化的算法方面做了大量研究。但是这个设想一开始并没有取得很好的效果，直到在硬件描述语言标准化之后，一些 EDA 厂家在描述语言(如 Verilog 和 VHDL 语言)的基础上开发了实现设计自动变换(即从设计输入到网表变换)的逻辑综合工具，才真正地实现了这个目标。

目前比较成功的 IC 综合工具是 Synopsys 公司的设计编译器 DC(Design Compiler)，早期的 DC 综合出的电路性能不是非常优化，存在不少的缺点，综合效率也比较低。经过不断的改进，DC 目前已经普遍被工业界所接受。主要的原因是 20 世纪 90 年代中、后期，各个高校相继开设了 Verilog 和 VHDL 语言的课程，新一代的设计工程师习惯用语言而不是电路图描述电路；另外一个原因是半导体工艺快速发展，设计规模变得非常大，功能也非常复杂，传统的电路图的方法已经不可能适应当代的设计要求。自动综合工具开发无疑是 EDA 工具历史上一次非常重要的革命，它彻底地改变了人们的设计方法，极大地提高了设计效率。

随着 FPGA 的迅速发展，针对具体 FPGA 结构特点的综合工具也有不少面世，其中 Synplicity 就是一个典型的代表。Synplicity 是专门针对 FPGA 的综合工具，它可以根据 FPGA 的特点，产生最佳的综合效果。目前已经有多个 FPGA 厂家将该工具集成到其开发环境中。

除了综合工具，验证工具也在 20 世纪 90 年代后得到了迅猛的发展。系统建模工具、静态时序分析工具，等价性检验、模型检验等形式化工具也成为了设计工程师完成设计的重要辅助手段。

简言之，EDA 工具经过 30 多年的发展，已经成为硬件设计工程师必不可少的设计手段。随着各个学科的不断进步，EDA 工具将有更大的发展。

# 第 2 章 可编程逻辑器件

30 多年来，可编程逻辑器件 PLD 厂商不断优化其产品结构，采用更先进的设计和生产工艺，使得 PLD 的逻辑单元越来越多，性能越来越高，而单位成本和功耗却不断地降低。不同的可编程逻辑器件厂家的产品结构和性能都各有特色，以满足不同系统对性能和价格的要求。以 Altera、Xilinx、Lattice 和 Actel 等公司为代表的世界知名 PLD 生产商，其产品结构非常具有代表性，占据了绝大部分市场。本章将简单介绍这些厂家典型的可编程逻辑器件产品的结构和性能。

## 2.1 Altera 器件概述

Altera 公司是最大的可编程逻辑器件供应商之一。主要产品有：MAX3000/7000、FLEX10K、APEX20K、ACEX1K、Stratix/Ⅱ/GX、Cyclone/Ⅱ、HardCopy/Ⅱ等，开发软件为 MaxplusⅡ和 QuartusⅡ。近年来，Altera 公司发展很快，推出了一批新型可编程逻辑器件(PLD)，不仅提高了系统设计的灵活性，同时也加快了通信、电脑外围设备及工业领域内的公司进入市场的步伐。

Altera 公司提供的软件开发工具，可以让用户轻松地创建可编程芯片系统(SOPC)方案，将嵌入式处理器、存储器和其他复杂逻辑相结合的 IP 核集成在单个高性能 PLD 芯片上。工具软件 QuartusⅡ提供了诸如板级编辑、逻辑映射、布局布线及增强时序分析等功能，支持 Altera 公司的全线产品。

### 2.1.1 FPGA 系列简介

Altera 公司的 FPGA 器件种类丰富、使用方便，根据该器件的性能和发展过程，可以将其分为以下三类：

(1) 第一类为功能简单的早期 FPGA 器件，主要包含 FLEX10K、ACEX1K、FLEX6000 和 FLEX8000 系列。这些系列的结构相似，由逻辑块、嵌入式 RAM、输入/输出(I/O)单元和快速通道组成。其特点是容量小、功能简单、价格便宜。它属于 20 世纪 90 年代初的产品，现在已经逐步淘汰。

(2) 第二类为复杂的 FPGA 器件，包括 APEXⅡ、APEX 20K、Mercury 和 Excalibur 等，这些器件内部除嵌入了 RAM 之外，还嵌入了锁相环(PLL)、CPU 和高速收发器等，兼容较多的 I/O 标准，并具有容量大、集成度高、功能强大、应用广泛等特点。

(3) 第三类为新型的 FPGA 器件，由 Stratix、Stratix Ⅱ、Stratix GX、Cyclone Ⅱ和 Cyclone 五种系列组成。这些器件内嵌了许多的功能模块，如多种容量的 RAM、数字信号处理 DSP、高性能锁相环、高速收发器和各种 I/O 接口等硬核。此外，它能很方便地嵌入 IP 软核，尤其是 Altera 公司提供的 NIOS 软核。

表 2.1 给出了常用 FPGA 器件系列的主要性能。

**表 2.1　Altera 的 FPGA 器件系列主要性能表**

| 性　能<br>器　件 | 逻辑单元 LE | RAM 块/bit | 乘法器块<br>(18×18 位) | 锁相环<br>(PLL) | 用户 I/O<br>引脚 |
|---|---|---|---|---|---|
| FLEX6000 | 880～1960 | | | | 71～218 |
| FLEX8000 | 208～1296 | | | | 68～208 |
| FLEX10K | 576～2880 | 6144～20 480 | | | 66～470 |
| ACEX1K | 576～4992 | 12 288～49 152 | | | 66～333 |
| APEX20K/C/E | 1200～51 840 | 24 576～442 368 | | 1～4 | 88～808 |
| APEX Ⅱ | 16 640～67 200 | 425 984～1 146 880 | | 4＋8 | 492～1060 |
| Excalibur | 4160～38 400 和<br>一个 ARM 核 | 单口 32～256 K<br>双口 16～128 K | | | 186～711 |
| Mercury | 4800～14 400 | 49 125～114 688 | | 8～18 | 303～486 |
| Cyclone | 2910～20 060 | 59 904～294 912 | | 1～2 | 65～301 |
| Cyclone Ⅱ | 4608～68 416 | 119 808～1 152 000 | 13～150 | 2～4 | 85～622 |
| Stratix | 10 570～114 140 | 799 000～8 783 000 | 6×4～22×4 | 6～12 | 345～1203 |
| Stratix Ⅱ | 等效<br>15 600～179 400 | 419 328～9 383 040 | 12×4～96×4 | 6～12 | 308～1170 |
| Stratix GX | 10 570～41 250 | 920 448～3 423 744 | 6×4～14×4 | 4～8 | 330～544 |

## 2.1.2　EPLD 系列简介

Altera 的 EPLD(Electrically Programmable Logic Device)器件是一种可擦除可编程逻辑器件，它将多个可编程阵列逻辑(PAL)器件集成到一个芯片上，具有类似 PAL 的结构，因此也称为复杂可编程逻辑器件(CPLD)。主要系列有 MAX3000、MAX5000、MAX7000、MAX9000、MAX Ⅱ和 Classic 器件。

(1) MAX5000 和 Classic 是 Altera 公司的早期产品，基于 EPROM 工艺，编程信息不易丢失，需要用紫外线进行擦除，使用不方便，而且集成度低，现在已经退出市场。

(2) MAX3000A、MAX7000、MAX9000 是 Altera 公司推出的低价格 $E^2$PROM 工艺 PLD，容量为 32～560 个宏单元，满足了不同的用户需要，属于 Altera 公司的主流器件。

(3) MAX Ⅱ器件系列基于 0.18 μm 六层金属 Flash 工艺，采用查找表(LUT)的方式，继承 CPLD 器件的特点，但成本降低了一半，功耗降低了 90%，密度增加了四倍，而性能提高了两倍。

表 2.2 给出了 EPLD 器件的主要性能。

表 2.2　Altera 的 EPLD 器件的主要性能表

| 性　能<br>器　件 | 宏单元 | 逻辑块 | 用户 I/O 引脚 | 典型可用门 |
|---|---|---|---|---|
| MAXⅡ | 等效 192～1700 | 240～2210 逻辑单元 LE | 80～272 | |
| MAX7000S/AE | 32～512 | 2～32 | 34～212 | 6000～10 000 |
| MAX3000A | 32～512 | 2～32 | 34～158 | 600～10 000 |
| MAX9000 | 320～560 | 20～35 | 59～216 | 10 000～12 000 |
| Classic | — | — | 22～68 | 300～900 |

### 2.1.3　结构化 ASIC 器件

Altera 的 HardCopy 结构化 ASIC 是一种新的 ASIC 解决方案，HardCopy 器件与等价的 FPGA 原型器件功能相同，而且工艺处理技术和处理电压也相同，只是在结构组成上有所不同。为用户的专用设计保留底层和顶层两个金属层。通过定制金属层替代可编程布线和逻辑配置层，同时可以去掉不需要的可编程单元和一些硬 IP 核，实现了更小的管芯尺寸，大大降低了功耗，实现了大批量、低成本的要求。

设计人员可通过以下步骤实现 FPGA 向 HardCopy 上的移植，而避开传统 ASIC 开发所需的耗时和巨大投入：

(1) 以 Altera FPGA 为原型进行设计。

(2) 系统测试、调试在 FPGA 原型上进行。

(3) 将 FPGA 原型上所有功能无缝移植到 HardCopy 结构化 ASIC 器件上。

在为用户保留的两个金属层中只有顶层金属层需要生成，因此 Altera 能够在 HardCopy 系列器件上实现快速的批量生产，满足用户在性能、价格和时间上的要求。在 Altera 公司将用户的设计无缝移植到引脚兼容、功能等价的 HardCopy 器件之前，用户可以使用 Altera 的 FPGA 来开发、验证并完成系统设计。由于在 FPGA 器件上完成了前端设计，降低了 Altera 结构化 ASIC 的使用风险。表 2.3 给出了与 Hardcopy 器件等价的 FPGA 原型器件以及工艺处理技术。

表 2.3　HardCopy 系列工艺处理技术

| 处理工艺<br>器　件 | HardCopy<br>工艺处理技术 | 定制层数量 | 电压<br>(与 FPGA 相同) | FPGA<br>工艺处理技术 |
|---|---|---|---|---|
| HardCopyⅡ | 90 nm | 2 | 1.2 | 90 μm |
| HardCopy Stratix | 0.13 μm | 2 | 1.5 | 0.13 μm |
| HardCopy APEX 20KC | 0.18 μm | 3 | 1.8 | 0.15 μm |
| HardCopy APEX 20KE | 0.18 μm | 3 | 1.8 | 0.18 μm |

### 2.1.4　FPGA 器件的配置芯片

一般 FPGA 器件采用 SRAM 工艺实现，在掉电后 FPGA 内部保存的配置数据全部丢失，每次加电时都需要重新写入配置数据，因此需要外部提供非易失性 ROM 来保存 FPGA 的

配置数据，并且外部 ROM 与对应的 FPGA 要有良好的接口。所以各个厂商在提供 FPGA 器件的同时也提供对应的配置芯片 ROM。

Altera 公司的配置芯片为所有 Altera FPGA 产品提供了解决方案，包括 Stratix Ⅱ、Stratix、Stratix GX、Cyclone Ⅱ、Cyclone、APEX Ⅱ、APEX 20K、APEX 20KE、APEX 20KC、Excalibur 和 Mercury 器件系列。由于下一代无线、通信和存储系统中对于高密度 FPGA 的需求不断上升，使得更大容量更快速度的配置芯片器件的需求量也不断增加。

Altera 公司的配置器件有两类：

(1) 标准型配置器件，包括 EPC2、EPC1、EPC1441、EPC1213、EPC1064 和 EPC1064V，配合低密度 FPGA 的使用。

(2) 增强型配置器件，包括 EPC4、EPC8、 EPC16、EPCS4、EPCS16、EPCS64 器件，拥有高达 64 Mb 的配置存储器，为大容量 FPGA 提供单器件一站式(只用一个配置芯片可以满足大容量的要求)的解决方案。表 2.4 列出了配置器件芯片的性能。

**表 2.4　配置器件的性能表**

| 型　号 | 容量 | 适用型号<br>(详细内容请参阅数据手册) | 电压/V | 常用封装 |
|---|---|---|---|---|
| EPC1441 (不可擦写) | 441 kb | FEX 6000/8000/10K | 3.3/5 | 8 脚 DIP |
| EPC1 (不可擦写) | 1 Mb | FEX 6000/8000/10K，更大芯片要多片级连 | 3.3/5 | 8 脚 DIP |
| EPC2(可重复擦写) | 2 Mb | 所有型号，更大芯片要多片级连 | 3.3/5 | 20 脚 PLCC |
| EPC8(可重复擦写) | 8 Mb | 所有型号 | 3.3 | 100 脚 PQFP |
| EPC16(可重复擦写) | 16 Mb | 所有型号 | 3.3 | 88 脚 BGA |
| EPC1213 | 208 Kb | FLEX 8000 | 5 | 8 脚 PDIP<br>20 脚 PLCC |
| EPC1064V | 64 Kb | FLEX 8000 | 3 | 8 脚 PDIP<br>20 脚 PLCC |
| EPCS1 | 1 Mb | EP1C3、EP1C4、EP1C6 | 3 | 8 脚 SOIC |
| EPCS4 | 4 Mb | 所有 Cyclone 型号 | 3 | 8 脚 SOIC |
| EPCS16 | 16 Mb | 所有 Cyclone 型号 | 3 | 8 脚 SOIC |
| EPCS64 | 64 Mb | 所有 Cyclone Ⅱ 和 Stratix Ⅱ 型号 | 3 | 8 脚 SOIC |

## 2.2　Altera 的 EPLD 器件系列

### 2.2.1　EPLD 器件的特性

Altera 公司的 EPLD 器件，采用 CMOS $E^2$PROM 技术制造，具有非易失和在线可编程等特性。器件 MAX3000/5000/7000/9000 系列，在结构上每个逻辑阵列块由 16 个宏单元组成，逻辑阵列块之间由快速通道互连。MAX Ⅱ 系列采用了基于查找表的新型 EPLD 架构，这种基于查找表的架构在最小的 I/O 焊盘约束的空间内提供了更多的逻辑容量。具有成本

低、功耗小和密度高的特点。

### 1．MAX3000 系列

Altera 在 1999 年推出了 MAX3000 系列的 $E^2$PROM 可编程器件，有 32～512 个宏单元，34～208 个可用引脚，兼容 PCI 接口，支持在系统可编程(ISP)、JTAG 边界扫描、热插拔和多电压接口等功能，其主要性能如表 2.5 所示。

表 2.5　MAX3000 器件性能表

| 器件 / 性能 | EPM3032A | EPM3064A | EPM3128A | EPM3256A | EPM3512A |
|---|---|---|---|---|---|
| 可用门 | 600 | 1250 | 2500 | 5000 | 10 000 |
| 宏单元 | 32 | 64 | 128 | 256 | 512 |
| 逻辑阵列块 | 2 | 4 | 8 | 16 | 32 |
| 用户可用 I/O 引脚 | 34 | 66 | 96 | 158 | 208 |
| 可达到计数频率 fCNT/MHz | 227.3 | 222.2 | 192.3 | 126.6 | 116.3 |

### 2．MAX5000 系列

MAX5000 系列是 Altera 公司的第一代 MAX 器件。它采用 EPROM 技术制造的非易失 PLD，可用紫外线进行擦除，编程信息不易丢失。虽然它价格低，但集成度低，I/O 引脚少，信息擦除不方便，属于早期产品，已经退出市场。

### 3．MAX7000 系列

MAX7000(包含 MAX7000AE、MAX7000B 和 MAX7000S)系列器件，采用 CMOS $E^2$PROM 工艺，集成度为 600～10 000 个可用门，32～512 个宏单元，计数器的工作频率可达 303 MHz，支持先进的 I/O 标准(包括 SSTL-2、SSTL-3 和 GTL+)和多电压接口，与 PCI 接口兼容，内嵌 JTAG 边界扫描电路，支持热插拔和 ISP 功能。MAX7000 系列的主要性能如表 2.6 所示。

表 2.6　MAX7000 系列性能表

| 5 V | 3.3 V | 2.5 V | 宏单元 | 可用门 |
|---|---|---|---|---|
| EPM7032S | EPM7032AE | EPM7032B | 32 | 600 |
| EPM7064S | EPM7064AE | EPM7064B | 64 | 1250 |
| EPM7128S | EPM7128AE | EPM7128B | 128 | 2500 |
| EPM7256S | EPM7256AE | EPM7256B | 256 | 5000 |
| — | EPM7512AE | EPM7512B | 512 | 10 000 |

### 4．MAX9000 系列

MAX9000 系列基于第二代 MAX 结构，并将高效宏单元结构与 FLEX 的高性能、延时可预测的快速通道互连结构结合在一起，实现了 EPLD 器件的高密度和高性能。该器件提供 6000～12 000 个可用门，引脚之间的延时只有 10 ns，计数器速度可达 144 MHz，内嵌

JTAG 边界扫描测试(BST)电路。表 2.7 给出了 MAX9000 系列器件的性能。

表 2.7　MAX9000 系列性能表

| 器　件<br>性　能 | EPM9320<br>EPM9320A | EPM9400 | EPM9480 | EPM9560<br>EPM9560A |
|---|---|---|---|---|
| 宏单元 | 320 | 400 | 480 | 560 |
| 逻辑阵列块 | 20 | 25 | 30 | 35 |
| 可用门 | 6000 | 8000 | 10 000 | 12 000 |
| 最大用户 I/O 引脚 | 168 | 159 | 175 | 216 |

### 5．MAXⅡ系列

MAXⅡ系列采用先进的 0.18 μm 6 层金属 Flash 工艺，继承了 EPLD 器件的可擦除可编程的优点，是接口桥接、I/O 扩展、器件配置等方面应用的理想器件，也能实现 FPGA 和标准逻辑器件中所能实现的逻辑功能。MAXⅡ器件的密度范围是 240～2210 个逻辑单元(LE)，最多可达 272 个用户 I/O 管脚。表 2.8 列出了 MAXⅡ器件的性能。

表 2.8　MAXⅡ器件性能表

| 器　件<br>性　能 | EPM240 | EPM570 | EPM1270 | EPM2210 |
|---|---|---|---|---|
| 逻辑单元数(LE) | 240 | 570 | 1270 | 2210 |
| 等效典型宏单元数 | 192 | 440 | 980 | 1700 |
| 最大用户 I/O 管脚 | 80 | 160 | 212 | 272 |
| 用户 Flash 存储器比特 | 8192 | 8192 | 8192 | 8192 |
| 引脚之间的延时/ns | 3.6～4.5 | 3.6～5.5 | 3.6～6 | 3.6～7.1 |

MAXⅡ器件的主要特点有以下几个方面：

(1) 基于新型 EPLD 的架构，使芯片的密度提高了四倍，未封装的裸片面积减小，单个 I/O 管脚成本降低。与上一代 MAX 器件相比价格降低了一半。

(2) 采用 1.8 V 内核电压以减小功耗，与 3.3 V MAX 器件相比，功耗降低了 90%，而性能提高了两倍，内部的时钟频率可以达到 300 MHz。

(3) 内置用户非易失性 Flash 存储器块，使用户的使用更加灵活，取代 ROM 存储器件，减少芯片数量，降低设计电路板的复杂性。

(4) 提供实时在系统可编程能力，器件在工作状态时能够下载第二个设计，降低远程现场升级的成本。

(5) 灵活的多电压内核，片内有电压调整器，支持 3.3 V、2.5 V 或 1.8 V 的电源输入，减少电源电压种类，简化单板设计。

(6) JTAG 翻译器。通过访问 JTAG 状态机例化用户功能，提高单板上不兼容 JTAG 协议的 Flash 器件的配置效率。

(7) 多电压 I/O 能力。多电压能力提供和外部器件在 1.5 V、1.8 V、2.5 V 或 3.3 V 逻辑级的接口。

### 2.2.2 MAX9000 器件的结构

MAX9000、MAX7000 和 MAX3000 系列器件，都采用 CMOS $E^2$PROM 技术制造，组成功能的构成类似，本节以 MAX9000 器件结构的组成为例作一简要介绍。

MAX9000 系列器件的结构由逻辑阵列块、宏单元、扩展乘积项、快速通道(Fast Track)互连、专用输入和 I/O 单元组成。如图 2.1 所示。

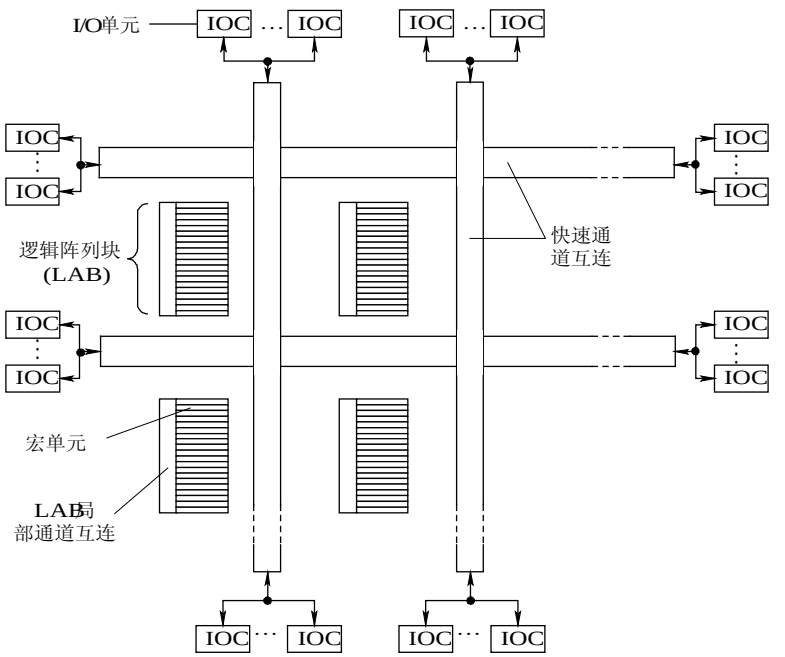



图 2.1 MAX9000 结构框图

#### 1．逻辑阵列块(LAB)

每个 LAB(Logic Array Block)由 16 个宏单元组成，各宏单元之间通过 LAB 局部通道互连阵列连接。多个 LAB 通过行和列快速通道(Fast Track)连接在一起，快速通道是贯穿器件全部长度和宽度的一系列快速互连通道，快速通道的两端都连接到 I/O 单元(IOC)上。MAX9000 每个行快速通道有 96 个，列快速通道有 48 个，如图 2.2 所示，还有三组数据选择器，两组为 16 个三选一，一组为 16 个四选一。

LAB 局部通道互连阵列由行快速通道送来的 33 个输入信号和 16 个宏单元送来的 16 个局部反馈信号，加上它们各自的反相信号共有 98((33＋16)×2＝98)个信号，再加上 16 个共享扩展信号，总共 114 个信号组成。其中有两个全局时钟和一个全局清除信号送给 LAB 中的每个宏单元，作为 16 个宏单元中寄存器的控制信号。LAB 的 16 个宏单元的输出信号直接送给行和列快速通道。一旦信号送到行和列快速通道，便会迅速地将信号送到其他 LAB 或 I/O 单元。

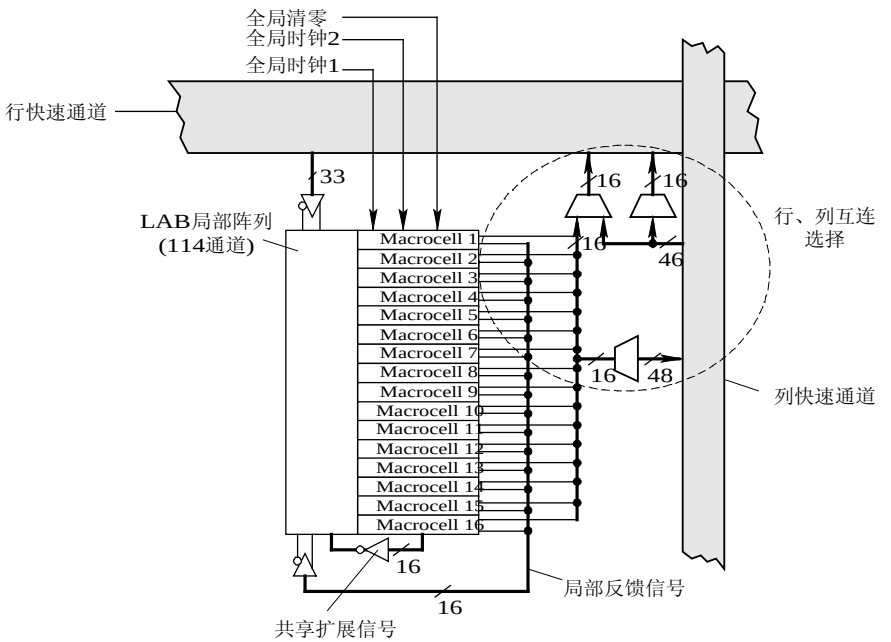



图 2.2　MAX9000 逻辑阵列块

宏单元和 I/O 引脚之间的连线是由快速通道直接相连的，这种全局的布线结构对于复杂的设计也能够预测其性能，但限制了布线的灵活性。

### 2．宏单元

MAX9000 的宏单元可以实现组合逻辑和时序逻辑功能，它由乘积项、乘积项选择矩阵和可编程的寄存器组成，如图 2.3 所示。左边的 LAB 局部互连为每个宏单元提供了五个乘积项，并作为乘积项选择矩阵的输入，以实现组合逻辑功能；组合逻辑的输出可以直接送到行或列快速通道，或可编程寄存器；乘积项的输出也可作为可编程寄存器的清除、置位、使能和时钟控制信号。16 个共享扩展项的反相信号直接送到逻辑阵列，并联扩展项可以与邻近的宏单元组合成更大的乘积项，这两种扩展乘积项可以有效利用逻辑资源，并提供组合逻辑功能。设计人员可以通过开发软件自动地分配和优化乘积项。

可编程寄存器可配置为 D 型、T 型、JK 或 RS 触发器，支持异步清除和异步置位，由乘积项选择矩阵控制这些操作，寄存器的清除也可单独由全局清除信号控制。

每个宏单元都有两个输出，一个是去行或列快速通道，另一个是反馈回 LAB 局部互连阵列。当乘积项的一部分组合逻辑通过寄存器输出或直接输出到快速通道，而其余的乘积项可以实现与之无关的组合逻辑，或者通过局部反馈信号将其送到 LAB 局部互连阵列，与其他宏单元组合实现其他逻辑功能。

一个宏单元可以实现五个输入的逻辑函数功能，而多于五个逻辑函数时需要附加乘积项。为了提高资源的利用率，MAX9000 结构中具有共享扩展和并联扩展乘积项，作为附加的乘积项直接送到同一个 LAB 中的任意宏单元中，这样既节省了资源，又提高了速度。

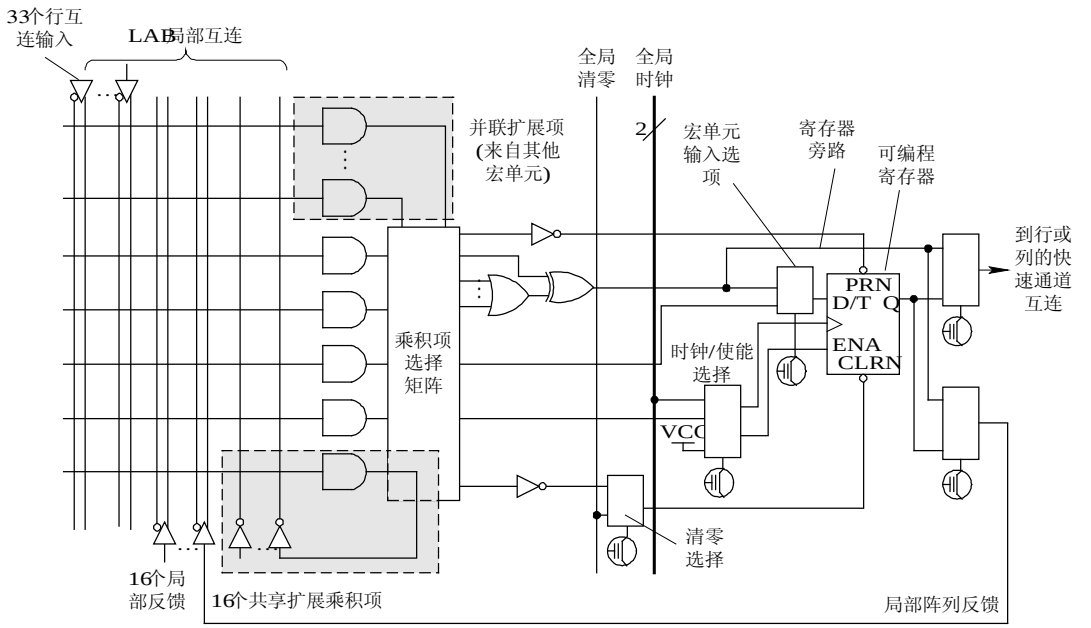



图 2.3　MAX9000 宏单元组成

图 2.4 是 MAX9000 共享扩展项逻辑电路。其中，每个宏单元有一个乘积项可以反馈到 LAB 局部互连阵列，而 LAB 中的 16 个共享扩展项信号可以被同一个 LAB 块中的任意一个宏单元使用，实现复杂的逻辑功能。

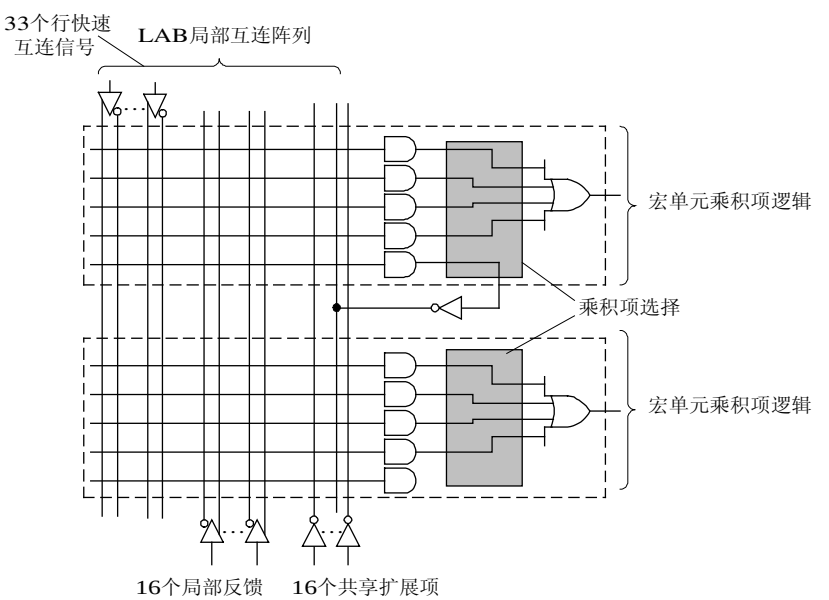



图 2.4　MAX9000 共享扩展逻辑电路

并行扩展是将若干个宏单元级连起来，实现快速复杂的逻辑功能。并行扩展可以达到20个乘积项，其中五个由宏单元本身提供，15个由扩展的乘积项提供，如图2.5所示。

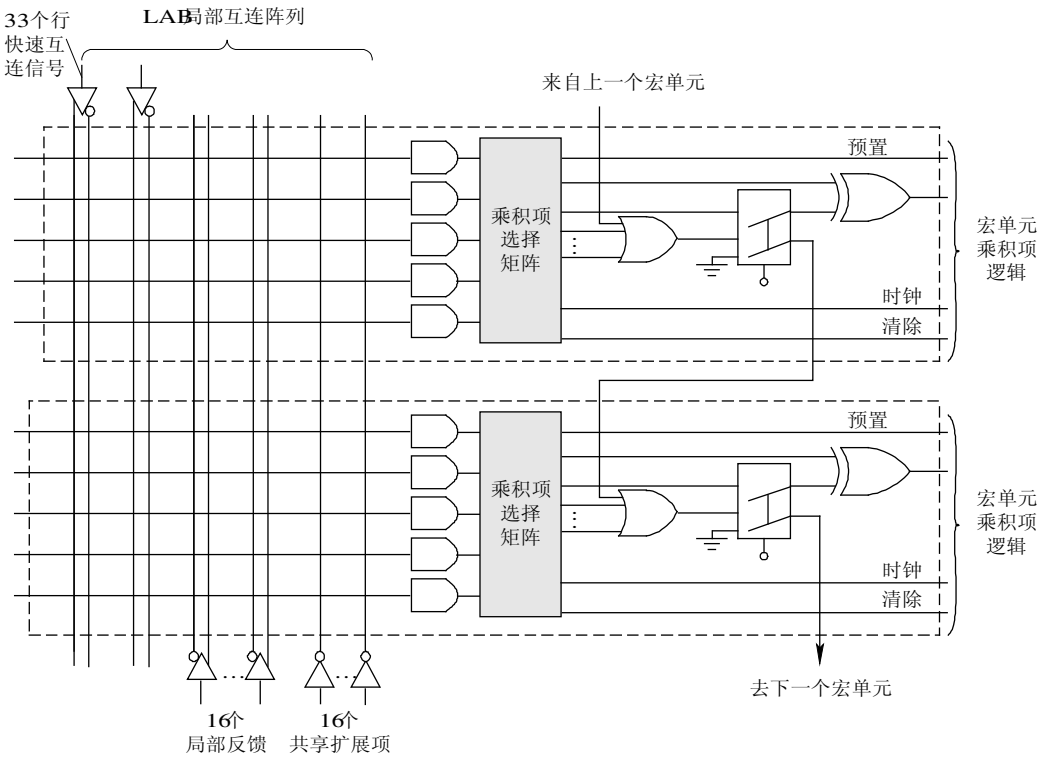



图2.5　MAX9000并联扩展逻辑电路

在一个LAB块中，将宏单元分为两组。宏单元1～8为一组，宏单元9～16为另一组，组成两个并行扩展项的链，并行扩展只能由小到大按顺序级连，例如宏单元8可以从宏单元7或宏单元7和6、或宏单元7、6和5借用并联扩展项。因此每组宏单元的最小编号仅能出借给并联扩展项，而最大编号宏单元仅能借用并联扩展项。

### 3．I/O单元

I/O单元由I/O寄存器、输出缓存器和一些多路选择器组成，如图2.6所示。I/O引脚可作为输入、输出或双向引脚。I/O寄存器也可作为输入寄存器或输出寄存器。寄存器的时钟信号、清除信号、使能信号和输出缓存器控制信号由周边扩展总线提供。周边扩展总线由八个输出缓存器控制信号、四个时钟信号、六个寄存器使能信号和两个清除信号组成，这些信号由专用引脚和内部电路提供。

输出缓存器提供了可编程的压摆率控制。低压摆率可以降低系统噪声，但缓存器的延时会增加，相反，快压摆率可以缩小缓存器的延时，而增加系统噪声。设计人员可根据各个引脚信号的特性，来指定各个引脚的电压摆率。

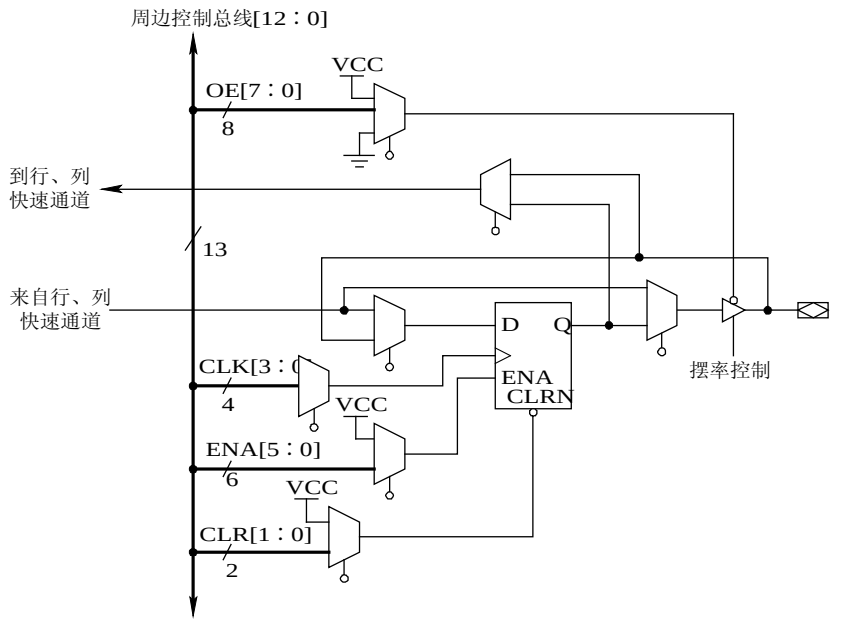



图 2.6　MAX9000 I/O 单元

### 2.2.3　MAX Ⅱ器件的结构

MAX Ⅱ架构由基于查找表的逻辑阵列块(LAB)、I/O 块、多轨互连线、JTAG 控制电路以及用户闪存和配置闪存组成，如图 2.7 所示。采用多轨互连线设计，提高了逻辑输入到输出之间连线的效率，从而获得了高性能、低功耗的特性。

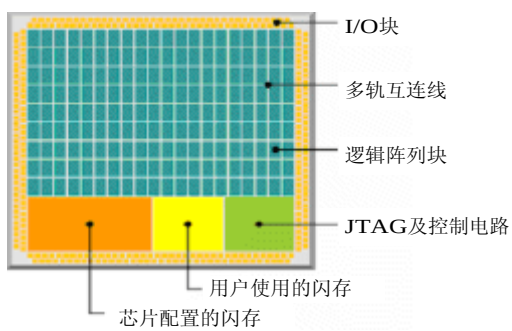



图 2.7　MAX Ⅱ结构框图

#### 1．逻辑阵列块

逻辑阵列块 LAB 由 10 个逻辑单元 LE 和局部的互连通道组成，每个 LAB 块可以与邻近的 LAB 块、I/O 引脚、行互连通道和列互连通道直接相连，逻辑阵列块的配置非常灵活。

#### 2．用户 Flash 存储器

MAX Ⅱ的用户 Flash 存储器是一个 8 Kb 的、用户可以使用的可编程 Flash 存储器块，用于存储用户定义的数据。这个功能允许在一个单一的 MAX Ⅱ器件内集成分立的非易失性的存储器，以减少系统芯片的数量和成本。

用户 Flash 存储器与 JTAG 电路及内核逻辑之间都有接口，用户可以灵活地采用各种方法对存储器进行读/写操作。如果用户想将其连接到一个标准总线上，如串行外设接口(SPI)、并口等，则可以使用 Quartus Ⅱ 软件提供的宏功能(megafunction)自动创建接口。

### 3．实时在系统可编程(ISP)

MAXⅡ器件嵌入了一个芯片配置 Flash 存储器(CFM)，将 Flash 配置块和可编程逻辑块分离，可实现实时 ISP 功能。新的设计能够直接下载到器件中，也可以等到下一次上电循环的时候再加载。有了实时 ISP 功能升级时就不需要停止系统运行，而可以在现场或远程直接快速升级，节省了因为系统停止运行或派遣人员去现场升级而产生的费用，降低了维护成本。

### 4．JATG 翻译器

MAXⅡ器件具有一种独有的 JATG 翻译器特性。这种特性允许通过 MAXⅡ器件执行定制的 JTAG 指令，配置单板上不兼容 JTAG 协议的器件(例如标准 Flash 存储器件)，从而简化了单板管理。有了 MAXⅡ器件内的 JTAG 翻译器，通过定制的指令，就可以用一个专用的 I/O 扫描链来编程和验证 Flash 器件。这种翻译器使用 JTAG 状态机访问 MAXⅡ器件内的可编程逻辑，执行 Flash 存储器驱动程序和译码功能。编程指令经过所连接的 I/O 引脚可以直接下载给 Flash 器件。QuartusⅡ软件以宏功能的形式支持这种应用。JTAG 翻译器可实现的功能如下：

(1) Flash 存储器的下载，编程标准 Flash 存储器件。

(2) 上电复位(POR)，用一个状态寄存器作为上电诊断。

(3) 内置自测功能(BIST)，内部包含一个向量发生状态机和 CRC 寄存器。

(4) 事件日志，通过 JTAG 接口访问系统事件日志。

(5) JTAG 接口到串口或并口桥接，实现从 JTAG 协议端口到任何串行或并行协议端口的桥接。

### 5．快速 I/O 连接

图 2.8 是 MAXⅡ器件中的 I/O 单元的逻辑框图。在 I/O 管脚和与之相邻的逻辑单元(LE)之间，MAXⅡ I/O 单元提供一个专用连线，缩短了 $t_{PD}$(引脚之间的延时)和 $t_{CO}$(全局时钟到输出的延时)。QuartusⅡ软件提供自动选用这个专用连线的功能，用于加速 I/O 单元的速度。

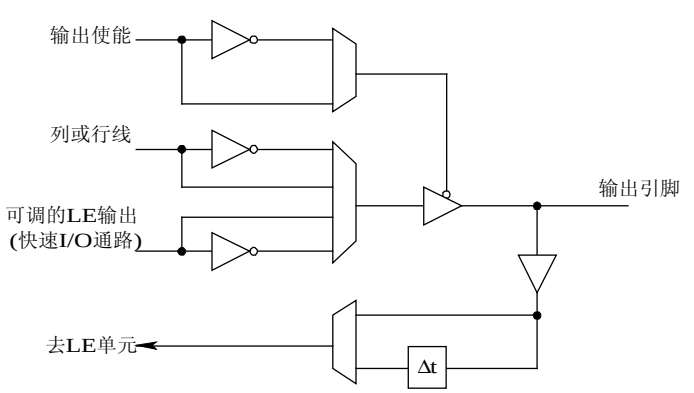



图 2.8　MAXⅡ的 I/O 单元的逻辑框图

6．灵活的多种电压

MAXⅡ架构支持多电压内核，该内核允许器件在 1.8 V、2.5 V 或 3.3 V 电源电压环境下工作，使设计者减少电源电压种类数量，简化板级设计。MAXⅡ器件还支持多电压 I/O 接口特性，能够与其他器件保持 1.5 V、1.8 V、2.5 V 或 3.3 V 逻辑级的无缝连接。EPM 240 和 EPM 570 器件包含两个 I/O 区，EPM 1270 和 EPM 2210 包含四个 I/O 区，每个 I/O 区有其自己的 $V_{CC}$ I/O 管脚，可以被独立地配置成支持 1.5 V、1.8 V、2.5 V 或 3.3 V 的接口。

# 2.3 Altera 的 FPGA 器件

## 2.3.1 简单 FPGA 器件

Altera 公司的 FLEX 系列和 ACEX1K 系列的 FPGA 器件，功能比较简单，属于较早推出的 FPGA，采用可重构的 SRAM 工艺和基于查找表(LUT)的结构，具有实现一般门阵列宏功能需要的所有特点。主要有 FLEX6000 系列、FLEX8000 系列和 FLEX10K 系列，其中 FLEX10K 系列是第一个嵌入 RAM 的 FPGA 器件，随后推出的 ACEX1K 系列，采用 2.5 V SRAM 的工艺，其结构与 FLEX10K 非常类似，也带有嵌入式 RAM 块。

1. 简介

1) FLEX6000 系列

FLEX6000 系列是一种小容量、低成本的 FPGA 器件，采用 0.3 μm 的 SRAM 工艺。该器件由逻辑阵列块构成，一个阵列块有 10 个逻辑单元 LE。一个 LE 由一个四输入的查找表和一个寄存器组成。该器件的集成度范围为 10 000～24 000 个可用门、808～1960 个 LE 单元。其主要性能如表 2.9 所示。

表 2.9 FLEX6000 系列性能

| 器件 / 性能 | EPF6010A | EPF6016 | EPF6016A | EPF6024A |
|---|---|---|---|---|
| 典型门 | 10 000 | 16 000 | 16 000 | 24 000 |
| 逻辑单元 | 808 | 1320 | 1320 | 1960 |
| 用户 I/O 引脚 | 120 | 204 | 171 | 218 |
| 工作电压 | 3.3 V | 5.0 V | 3.3 V | 3.3 V |

2) FLEX8000 系列

FLEX8000 系列器件结合了 FPGA 和 CPLD 的优点，具有 FPGA 的精细结构和较多个寄存器的特点，又具有快速 CPLD 和可预知连线延迟的优点。可满足各种应用，如数字信号处理、数据变换、总线接口和高速控制器等方面的需要。该系列通过查找表和可编程寄存器实现各种逻辑功能，集成度范围为 2500～16 000 个可用门，208～1296 个逻辑单元 LE，寄存器为 282～1500 个，并且有较多的 I/O 引脚和多电压的 I/O 接口。表 2.10 列出了 FLEX8000 系列器件的性能。

表 2.10　FLEX8000 系列器件性能

| 器件 / 性能 | EPF8282A<br>EPF8282AV | EPF8452A | EPF8636A | EPF8820A | EPF81188A | EPF81500A |
|---|---|---|---|---|---|---|
| 可用门 | 2500 | 4000 | 6000 | 8000 | 12 000 | 16 000 |
| 触发器 | 282 | 452 | 636 | 820 | 1188 | 1500 |
| 逻辑单元 | 208 | 336 | 504 | 672 | 1008 | 196 |
| 最大用户 I/O 引脚 | 78 | 120 | 136 | 152 | 184 | 208 |
| JTAG 边界扫描测试(BST)电路 | 有 | 无 | 有 | 有 | 无 | 有 |

3) FLEX10K 系列

FLEX10K 系列器件包含了一个嵌入阵列和一个逻辑阵列。嵌入阵列由容量为 2048 bit 的 RAM 块组成，可实现存储器和复杂的逻辑功能，如信号处理、数据传输、控制等；逻辑阵列由查找表和可编程寄存器组成，可实现各种组合逻辑和时序逻辑功能。这种嵌入阵列和逻辑阵列结合的特性，使设计人员可以实现可编程单芯片系统(System on a Programmable Chip，SOPC)。该系列器件的密度可达 25 万个可用门，40 960 位嵌入式 RAM 和 12 160 个逻辑单元，兼容 PCI 总线，支持 JTAG 边界扫描测试和 I/O 的多电压接口。芯片有多种封装，最大的用户 I/O 引脚可达 470 个，表 2.11 列出了 FLEX10K 系列常用的几种器件的性能。

表 2.11　FLEX10K 系列常用的几种器件的性能

| 器件 / 性能 | EPF10K30<br>EPF10K30A<br>EPF10K30E | EPF10K50<br>EPF10K50V<br>EPF10K50E<br>EPF10K50S | EPF10<br>K70 | EPF10K100<br>EPF10K100A<br>EPF10K100B<br>EPF10K100E | EPF10<br>K130V<br>EPF10<br>K130E | EPF10<br>K200E<br>EPF10<br>K200S | EPF10K<br>250A |
|---|---|---|---|---|---|---|---|
| 典型可用门(1) | 30 000 | 50 000 | 70 000 | 100 000 | 130 000 | 200 000 | 250 000 |
| 逻辑单元(LE) | 1728 | 2880 | 3744 | 4992 | 6656 | 9984 | 12 160 |
| 逻辑阵列块(LAB) | 216 | 360 | 468 | 624 | 832 | 1248 | 1520 |
| 嵌入式阵列块 | 6 | 10 | 9 | 12 | 16 | 24 | 20 |
| 总 RAM 位数 | 12 288<br>24 576(2) | 20 480<br>40 960(2) | 18 432 | 24 576<br>49 152 (2) | 32 768<br>65 536 (2) | 98 304 | 40 960 |
| 最大用户 I/O 引脚 | 246 | 310 | 358 | 406 | 470 | 470 | 470 |

注：(1) 边界扫描测试 JTAG 电路需要 31 250 附加门；(2) FLEX10KE 器件大于这些数量。

4) ACEX1K 系列

ACEX1K 器件也带有高性能的嵌入式存储块(EAB)，每个 EAB 块提供 4096 位，除实现一般 RAM 功能外，还可以实现双口 RAM、ROM，先进先出(FIFO)等其他一些逻辑功能，可以达到 64 位、66 MHz 的高性能的带宽。逻辑阵列由若干个 LAB 块组成，每个 LAB 块有八个逻辑单元 LE 和一个局部的快速通道组成，一个 LE 由一个四输入 LUT、一个可编程

的触发器、进位链、级连链和互连信号组成。一部分型号的器件还带有锁相环(PLL)功能，I/O 单元有多电压的功能，可以配置成 5 V、3 V 和 2.5 V 的电压。器件的密度为 576～4992 个逻辑单元，可以满足比较复杂的设计要求。表 2.12 列出了 ACEX1K 器件的性能。

表 2.12　ACEX1K 器件的性能

| 器　件<br>性　能 | EP1K10 | EP1K30 | EP1K50 | EP1K100 |
|---|---|---|---|---|
| 典型可用门 | 10 000 | 30 000 | 50 000 | 100 000 |
| 最大器件门数 | 56 000 | 119 000 | 199 000 | 257 000 |
| 逻辑单元 | 576 | 1728 | 2880 | 4992 |
| 嵌入块 | 3 | 6 | 10 | 12 |
| 总 RAM 位 | 12 288 | 24 576 | 40 960 | 49 152 |
| 最大用户 I/O 引脚 | 136 | 171 | 249 | 333 |

### 2. FLEX10K 器件结构

Altera 公司的简单 FPGA 器件系列结构组成相似，我们以 FLEX10K 系列为例来说明 FPGA 器件的组成。FLEX10K 由嵌入式阵列块、逻辑阵列块、快速通道和 I/O 单元组成，如图 2.9 所示。由图可以看出 LAB 排列成行和列，每一行包含了一个嵌入块 EAB，LAB 和 EAB 之间通过快速互连通道连接，I/O 块位于行快速互连通道和列快速互连通道的两端。

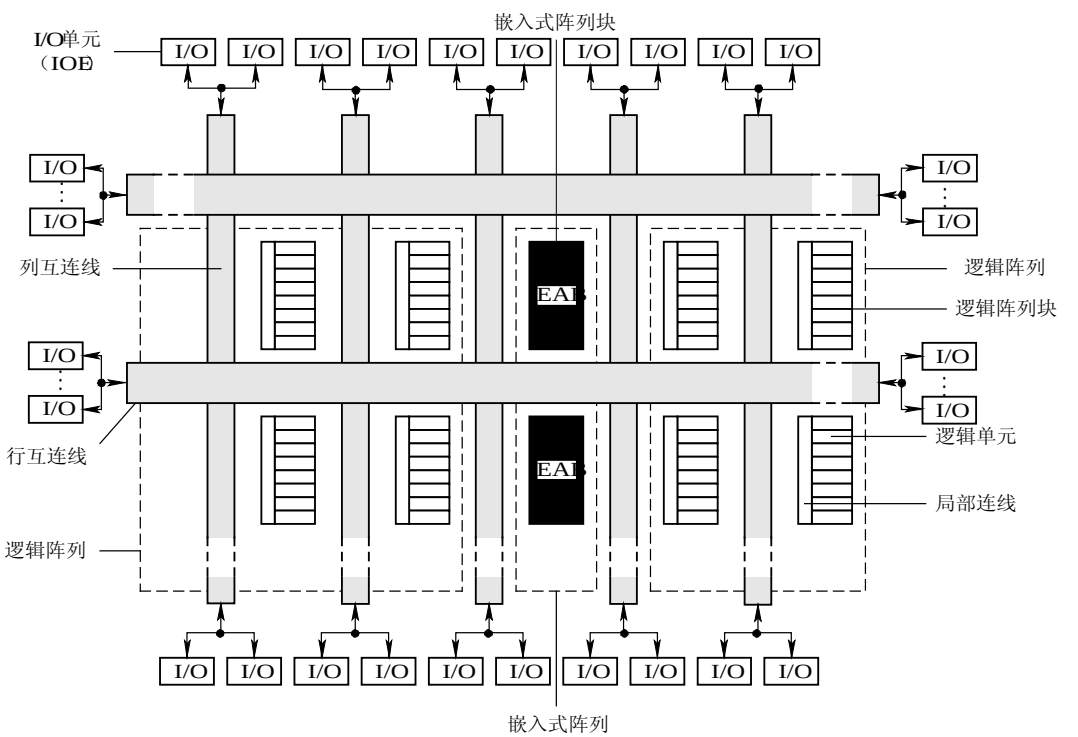
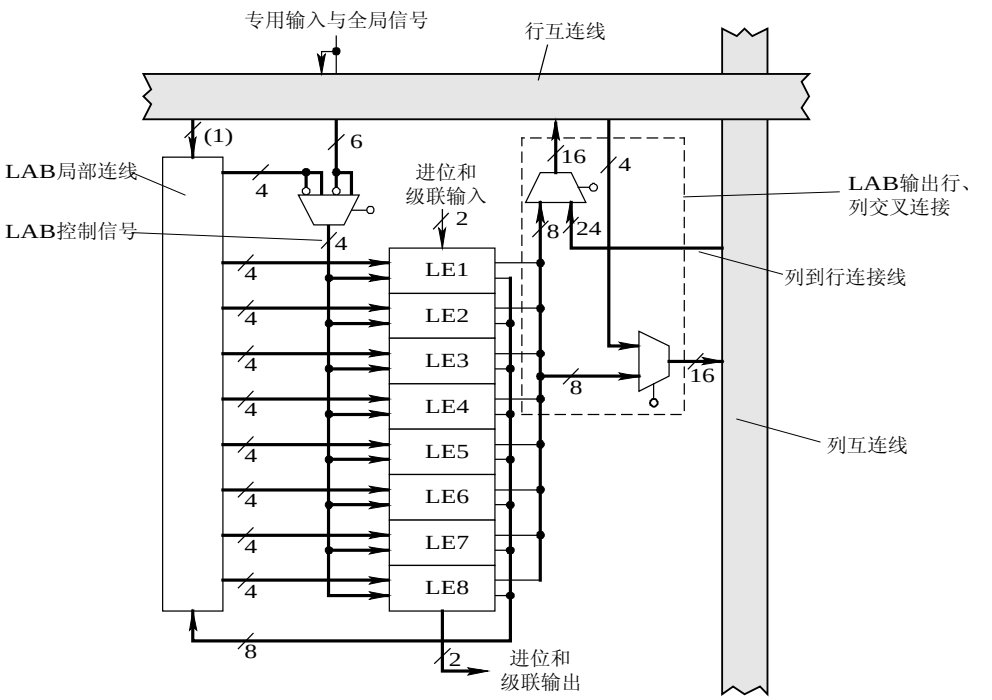



图 2.9　FLEX10K 器件结构框图

FLEX10K 器件提供四个全局信号，采用专用的布线支路，以便有比快速通道更短的延时和更小的偏移。全局信号可由四个专用的输入引脚驱动或内部的逻辑驱动，作为全局时钟信号和全局清除信号，用来驱动触发器的控制端，以确保控制信号高速、低偏移、有效地分配。

1) 逻辑阵列块(LAB)

每个 LAB 由八个 LE 和局部互连线组成，每个 LE 含有一个四输入查找表(LUT)、一个可编程触发器、进位链和级连链。八个 LE 可以构成一个中规模的逻辑块，如八位计数器、地址译码器和状态机。多个 LAB 可以组合为更大的逻辑功能块，实现更为复杂的逻辑功能。每个 LAB 块大约需要 96 个可用逻辑门，构成一个逻辑功能块，为 FLEX10K 器件提供粗颗粒的互连结构。这样以便于更容易实现高速布线，提高器件性能和器件资源的利用率。图 2.10 给出了 FLEX10K 器件的 LAB 结构。




注：图中(1)器件 10K10～10K50 有 22 个行互连线进入 LAB 局部互连阵列，器件 10K70～10K250 有 26 个行互连线进入 LAB 局部互连阵列。

图 2.10　FLEX10K 器件的 LAB 结构

每个 LAB 为八个 LE 提供四个控制信号，其中的两个用作时钟信号，另外两个用作清除/置位信号。LAB 四个控制信号可以由器件的六个专用输入引脚、全局信号或 LAB 局部互连的内部信号经过选择器选择后直接驱动。LAB 块有八个输出同时送给行互连和列互连，另外八个输出送到 LAB 局部互连，每个 LAB 块还分别有一个进位输入和级连输入，一个进位输出和级连输出。

2) 逻辑单元(LE)

LE 是 FLEX10K 结构中的基本单元，每个 LE 含有一个四输入查找表(LUT)、一个带有同步使能的可编程触发器、一个进位链和一个级连链。LUT 可以快速实现任何四输入的函数功能。每个 LE 都能驱动局部互连和快速通道(FastTrack)互连。图 2.11 给出了 FLEX10K 器件的 LE 逻辑单元。

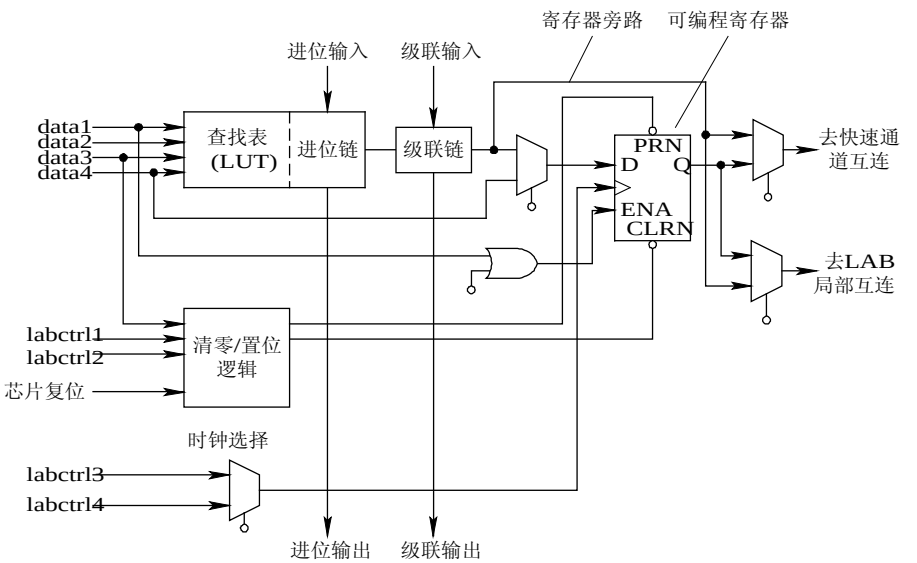



图 2.11　FLEX10K 器件的 LE 逻辑单元

LE 中的可编程触发器可配置成 D 型、T 型、JK 和 RS 触发器。触发器的时钟通过时钟选择器从 LAB 块的控制信号 labctrl3 和 labctrl4 中选择、清除和置位控制信号，这些控制信号可由 LAB 块的控制信号 labctrl1、labctrl2、data3 和芯片级复位信号组合产生。对于组合逻辑的输出可以旁路输出，也可以通过寄存器输出。

LE 的输出信号有两路，一路用于驱动 LAB 局部互连通道，另一路用于驱动行和列的快速通道，这两路输出信号可以被分别控制。例如，旁路输出驱动行和列快速通道，而寄存器输出驱动局部互连通道。由于寄存器和 LUT 可以分别独立使用，因而能够有效提高 LE 的利用率。

LE 之间有两路专用高速数据通道，即进位链(Carry Chain)和级连链(Cascade Chain)，将相邻 LE 连接，不经过局部互连通道。进位链支持高速计数器和加法器，级连链可以提高多个 LE 级连的速度。进位链和级连链可将同一 LAB 中的所有 LE 级连，也能将同一行中所有 LAB 级连。级连可以提高器件的速度，但大量使用进位链和级连链却会降低布局布线的灵活性，因此，只有速度要求高的关键部分才使用级连链。

(1) 进位链。进位链逻辑可以由工具软件 MAX+PLUS 或 Quartus 编译器在设计处理时自动生成，或者由设计人员在设计输入期间手工建立。进位链提供 LE 之间非常快(小于 0.2 ns)的向前进位功能。低位进位信号通过进位链向前进到高位，即进位到 LE 相邻的下一级。这种快速的进位功能可实现高速计数器、加法器和任意位数的比较器。

多个 LAB 块之间进位链互连时，为了提高适配率，进位链可以在一行 LAB 块中跳跃

交替互连，即长度超过一个 LAB 的进位链，要么从偶数 LAB 跨接到偶数 LAB，要么从奇数 LAB 跨接到奇数 LAB。例如，在一行中第一个 LAB 的最后一个 LE 进位输出到该行中第三个 LAB 的第一个 LE 上，到达中间的 EAB 时进位链终止，不能跨越 EAB 块。图 2.12 中给出了利用进位链将 n＋1 个 LE 互连来实现 n 位全加器的方法。

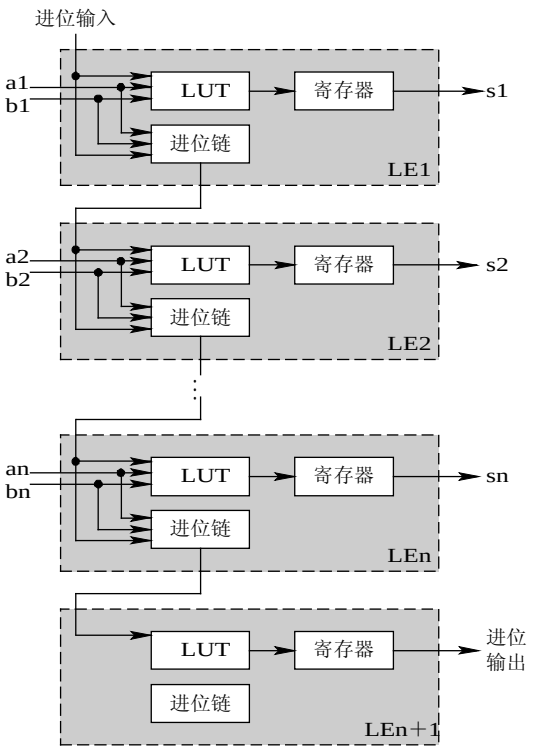



图 2.12　n 位全加器的进位链电路

LUT 完成两个输入信号相加，并将结果送到 LE 输出端。对于一般的加法器，可将寄存器旁路，若相加结果需要保存，就要使用寄存器。进位链逻辑产生进位输出信号，并直接送到高一级的进位输入。最后一个进位输出信号送到 LE 的 LUT 输入端，通过 LE 输出。

(2) 级连链。通过级连链组合更多输入的复杂逻辑功能，相邻的 LUT 通过级连链串接起来。级连链具有“逻辑与”或者“逻辑或”的功能，如图 2.13 所示。每增加一个 LE，函数功能的有效输入增加四个，其延时增加不到 0.7 ns。级连链可以通过编译器自动产生或由设计人员手工建立。

多个 LAB 块的级连链与进位链相似，采用跳跃交替互连，即长度超过一个 LAB 的级连链，要么从偶数 LAB 跨接到偶数 LAB，要么从奇数 LAB 跨接到奇数 LAB。例如，在一行中第一个 LAB 的最后一个 LE 级连输出到该行中第三个 LAB 的第一个 LE 级连输入，级连链与进位链一样也不能跨越 EAB 块。

图 2.13 是 4n 个输入函数的级连链电路，采用与级连和或级连的两种不同方式，可以实现不同的逻辑功能。级连链的延时小于 0.7 ns，LE 的延时小于 1.6 ns，使用级连链对一个 16 位地址译码，需要 3.7 ns 的时间。

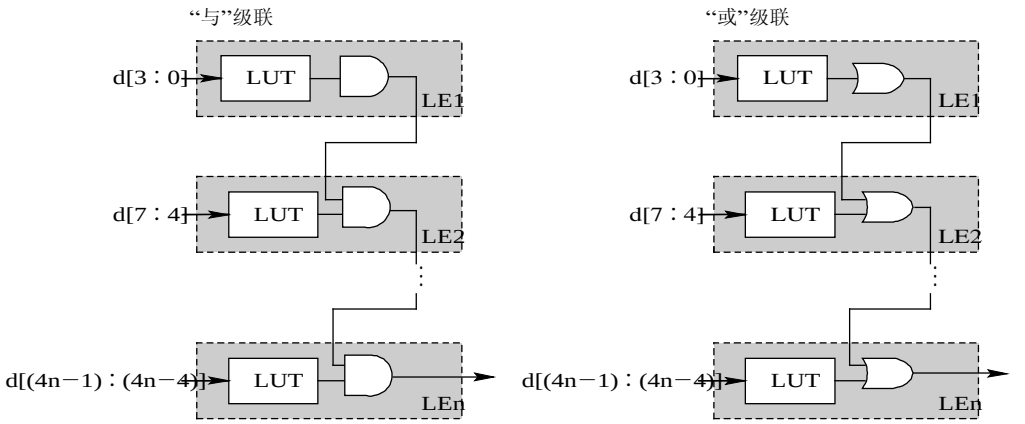



图 2.13　4n 个输入函数的级连链电路

(3) LE 的工作模式。FLEX10K 器件的逻辑单元 LE 有四种模式：正常模式、算术模式、加/减计数模式和可清除计数模式，如图 2.14 所示。这些模式使用 LE 的资源有所不同，其中输入信号有七个，四个来自 LAB 局部互连，另外三个信号分别是可编程寄存器的反馈信号、进位输入信号和级连输入信号。除此之外，还有三个寄存器控制信号，分别为时钟信号、清除信号和置位信号。Altera 的开发工具能够自动地选择合适的 LE 模式，实现计数、加法和乘法等功能，如果需要，设计人员也可指定 LE 的工作模式，来优化器件性能，实现高效的特定功能。

① 正常模式。如图 2.14(a)所示，正常模式用于一般的逻辑功能和各种译码功能以发挥 LE 级连链的优势。在这种模式下，从 LAB 局部互连来的四个数据输入和一个进位输入，作为四输入 LUT 的输入信号。编译器能够自动地从进位输入和 data3 中选择一个作为 LUT 的输入信号，LUT 的输出信号可以与级连输入信号相与后，通过级连输出信号形成级连链。寄存器或 LUT 输出可以同时用来驱动局部互连和快速通道互连。

LE 中的 LUT 和寄存器能够独立使用，这一特性称为寄存器包装。为了支持寄存器包装，LE 有两个输出：一个用于驱动局部互连，另一个用于驱动快速通道互连。data4 信号能够直接驱动寄存器，允许 LUT 具有独立于寄存器的逻辑功能。查找表 LUT 的第四个输入信号还可被寄存器保存。换句话说，LE 不但能产生一个四输入逻辑函数，而且其中一个输入能够用来驱动寄存器。被包装的寄存器仍然能够使用 LE 中的时钟使能、清除和置位信号。LE 中的寄存器包装后，寄存器能够驱动快速通道互连，而 LUT 能够驱动局部互连，反之亦然。

② 算术模式。该模式下，LE 可以被配置成两个三输入 LUT，分别是实现加法器、累加器和比较器理想模式。其中一个 LUT 实现三输入逻辑函数，另一个产生进位输出。如图 2.14(b)所示，第一个 LUT 将进位输入信号和两个来自 LAB 局部互连的输入信号进行逻辑组合产生一个(直接或寄存器)输出。例如，在加法器里，输入信号是 data1、data2 和进位输入三个信号之和。第二个 LUT 用这相同的三个输入信号产生进位输出信号，从而形成一个进位链。而级连链也可以同时使用。

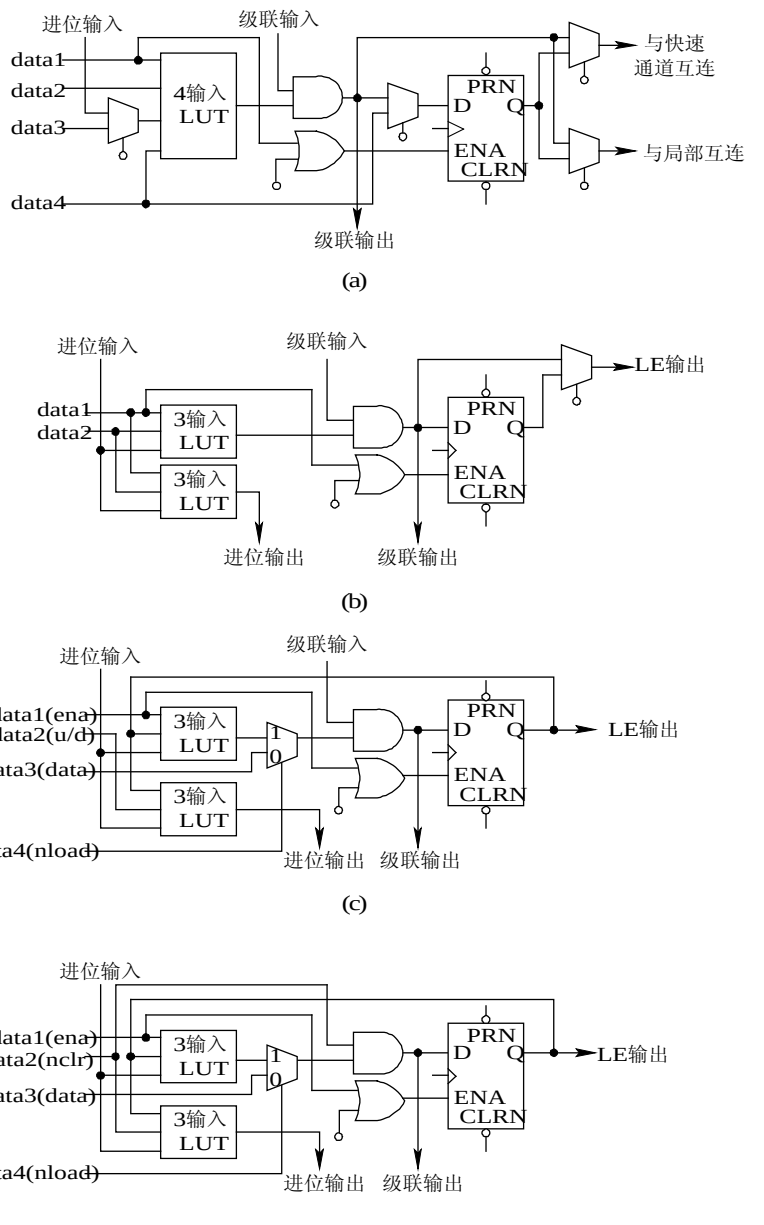



图 2.14　LE 的工作模式

(a) 正常模式；(b) 运算模式；(c) 加/减计数模式；(d) 可清除计数模式

③ 加/减计数模式。该模式也有两个三输入 LUT，其中一个产生计数数据，另一个产生快速进位，如图 2.14(c)所示。该模式提供计数器使能、时钟使能、加/减控制和数据加载选择等信号，这些控制信号可以是 LAB 局部互连的输入信号、进位输入信号或寄存器输出的反馈信号。二选一多路选择器可以提供同步数据加载，也可以不用 LUT 资源，而借助于寄存器的清除和置位控制信号异步加载数据。

④ 可清除计数模式。该模式类似于加/减计数器模式，但它支持同步清除而不是加减控制，用级连输入信号作为清除信号，可清除计数模式的两个三输入 LUT。一个 LUT 用作计数，另一个 LUT 产生快速进位信号。二选一多路选择器可以提供同步数据加载，多路选择器输出和同步清除信号进行逻辑与之后，作为级连输出信号并同时送给寄存器。

(4) 寄存器清除/置位。LE 的输入信号 data3，labctrl1 和 labctrl2 可作为寄存器的清除和置位控制信号。清除和置位信号控制 LE 异步加载数据到寄存器，可用 labctrl1 或 labctrl2 控制异步清除，或者 labctrl1 控制寄存器置位，来实现异步加载，加载的数据由 data3 确定。当 labctrl1 有效时，data3 将被加载到寄存器。

Altera 软件在编译时，能自动选择合适的控制信号实现清除/置位功能。由于清除和置位信号是低电平有效，所以编译器自动将未使用的清除和置位信号配置为高电平。

在设计中实现逻辑的异步清除与置位，有六种模式可供选择。它们是异步清除，异步置位，异步清除与置位，带清除的异步加载，带置位的异步加载，无清除或置位的异步加载，这六种模式如图 2.15 所示。

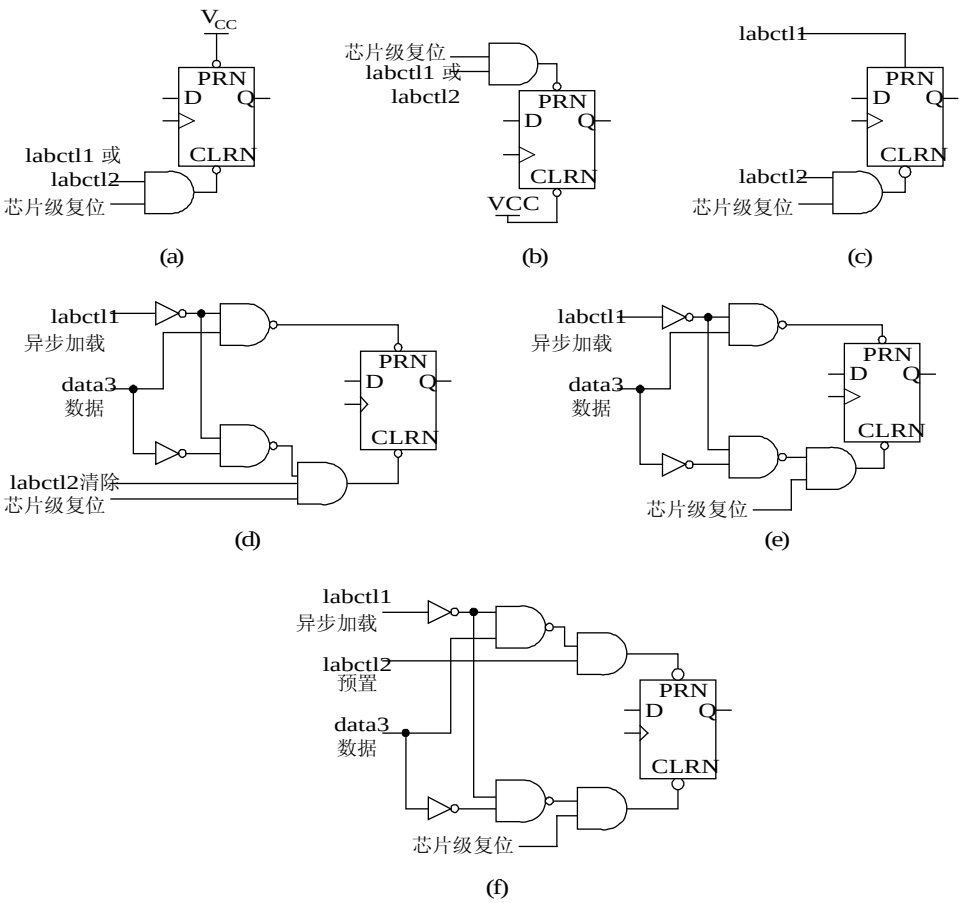



图 2.15　寄存器清除/置位

(a) 异步清除；(b) 异步置位；(c) 异步清除与置位；(d) 带有清除的异步加载；(e) 不带有清除和置位的异步加载；(f) 带有置位的异步加载

另外，FLEX10K 器件还提供了一个芯片级复位引脚，可使所有寄存器复位，在设计输入时可指定该特性。在清除与置位所有模式中，芯片级复位信号优先于其他信号。在异步置位模式下，当芯片级复位信号起作用时，寄存器被置位，因而也可以用复位信号实现异步置位。

3) 嵌入式 RAM 块(EAB)

嵌入式 RAM 块是构成嵌入式阵列的基本单元。每个 EAB 块作为存储器时可提供 2048 位，实现 RAM、ROM、FIFO 或双口 RAM 等功能。当用来实现乘法器、控制器、状态机以及 DSP 等复杂逻辑时，每个 EAB 可作为 100～600 个逻辑门使用。EAB 块可以单独使用，也可与其他模块组合使用。

图 2.16 是嵌入式 RAM 块的结构。其中，每个嵌入式 RAM 块的输入口和输出口都带有寄存器，可以实现一般阵列的复杂功能，也可实现乘法器、矢量运算、调整电路等功能，将这些功能组合起来，可实现更为复杂(如数字滤波器和微控制器等)的功能。

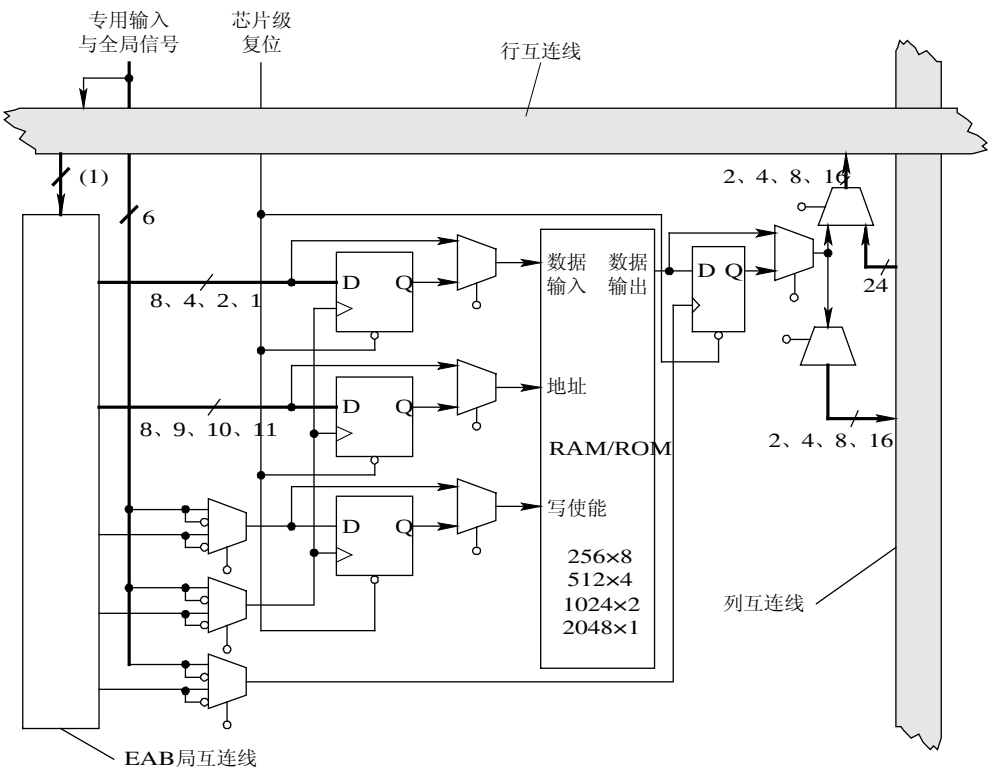



图 2.16　FLEX10K 嵌入式 RAM 块

嵌入式 RAM 块可以作为一个大型查找表 LUT，实现多输入的函数功能。在配置时，根据函数要求将嵌入式 RAM 块编程后，作为只读模式使用。EAB 的大容量使得设计人员能够实现复杂的逻辑功能，相对多个 LE 或 FPGA 的 RAM 块互连来说，不存在布线延时，因此使用查找表实现组合逻辑要比一般算法快。例如，单个 EAB 可以实现一个带有八输入和八输出的 4×4 乘法器。

当 EAB 用作 RAM 时，每个 EAB 能配置成 256×8，512×4，1024×2 或 2048×1 模式

使用，更大的 RAM 可由多个 EAB 进行组合。例如，两个 256×8 的 RAM 块可组成一个 256×16 的 RAM，两个 512×4 的 RAM 可组成一个 512×8 的 RAM，如图 2.17(a)所示。如果需要，一个器件中所有的 EAB 可级连成一个单一的 RAM，Altera 软件能自动组合 EAB 块，达到设计人员所需要的 RAM 格式，图 2.17(b)给出了 RAM 的组合实例。

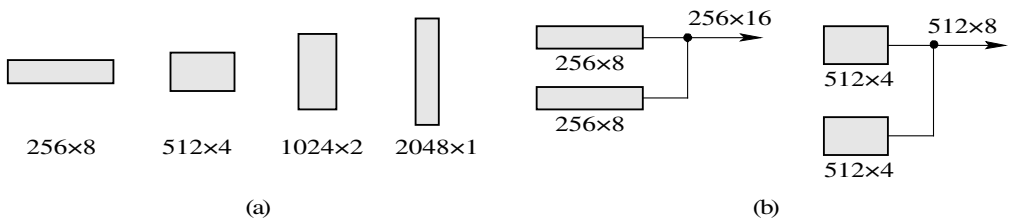



图 2.17　嵌入式 RAM 配置举例

(a) 嵌入式 RAM 块配置；(b) 嵌入式 RAM 块组合

EAB 块的驱动和时钟信号选项很灵活。EAB 的输入和输出可以选择不同的时钟。EAB 的数据输入、数据输出、地址输入和写使能信号 WE 都可使用寄存器，或不使用寄存器。EAB 时钟信号可用全局信号、专用时钟引脚信号或 EAB 局部互连来的信号。因为 LE 输出可以反馈到 EAB 局部互连，所以 LE 的输出可以控制 WE 信号或 EAB 时钟信号。

每个 EAB 块的输入信号来自行互连通道，它的输出可同时驱动行互连通道和列互连通道，没有使用的通道可由其他 LE 使用。这样可以增加器件布线资源的利用率。

4) 快速通道

器件内部信号的互连和器件引脚之间的信号互连由快速通道连线提供，快速通道互连是一系列贯通器件行、列的快速连接通道。这种全局布线结构，即使对于复杂的设计也可预测其性能。

图 2.18 画出了快速通道互连的结构。快速通道由贯穿整个器件的行快速互连和列快速互联组成，每条行快速互连承载进出这一行中 LAB 的信号。行和列快速互连都可以驱动 I/O 引脚，也可将信号送给器件中的相应行列中的 LAB 块。每个行通道可通过多路选择器选择 LE 输出和三个列通道中的一个信号。每个 LE 通过两个四选一的多路选择器驱动两个行通道，一个 LAB 中的所有八个 LE 都可以驱动行通道。

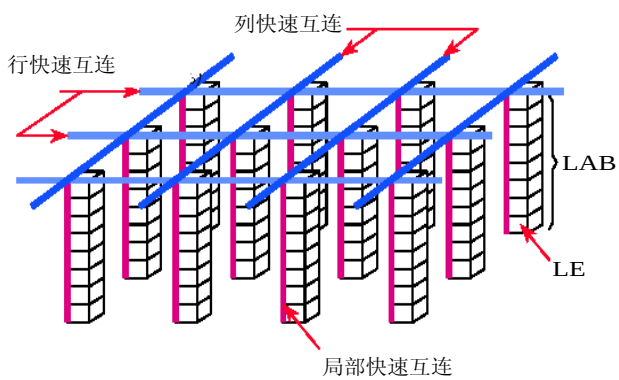



图 2.18　快速通道互连结构

来自列通道的信号，可能是 LE 的输出，也可能是 I/O 引脚的输入。在将列互连信号送到另一个 LAB 块或 EAB 块之前，必须先将其传送到行互连通道。由 I/O 单元(I/O)或 EAB 驱动的每一个行通道信号都可驱动一个特定的列通道，如图 2.19 所示。

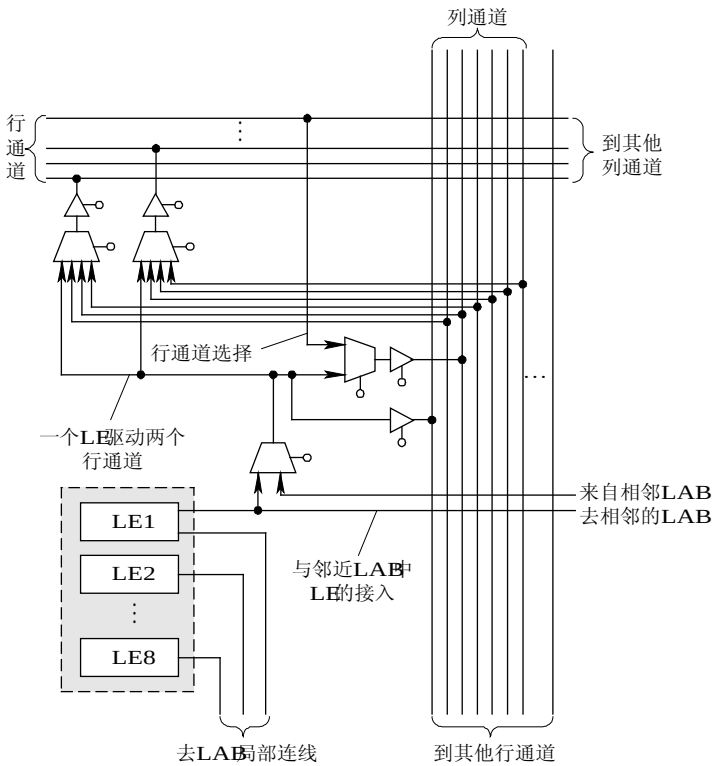



图 2.19　LAB 块与行、列通道的互连

为了提高布通率，行通道由全长和半长通道组成，全长通道与一行中所有的 LAB 块相连，而半长通道仅连接一行中一半的 LAB。EAB 可以连到左半长通道，也可连到全长通道，EAB 输出能驱动全长通道。此外，还提供了一个可预测的行宽(Row-Wide)互连通道，增加了布线资源。两个相邻的 LAB 可以使用一个半长通道连接，而另一个半长通道能够用作其他行通道连接。

5) I/O 单元

图 2.20 是 I/O 单元的逻辑框图。它由一个输出缓存器、三个寄存器以及一些逻辑电路组成。三个寄存器分别为输入寄存器、输出寄存器和输出使能寄存器(OE 寄存器)。寄存器既可用作需要快速建立时间的外部数据输入，也可作为快速的数据输出，在某些情况下，用 LE 的寄存器作为输入寄存器，比 I/O 输入寄存器在建立时间方面更快。I/O 引脚可配置为输入、输出或双向引脚。如果来自行、列的互连信号需要反相，可使用编译器的反向选项功能，自动地将其信号反相。

I/O 控制信号网络可以有 12 个控制信号，每个 I/O 控制信号可由任意一个专用输入引脚驱动，也可以由一个特定行中的第一个 LE 驱动。为了减小信号的失真，使用了高速驱动器，向每个 I/O 单元提供时钟信号、清除信号、时钟使能信号和输出使能信号，这些信号

可以控制八个输出缓存器使能、六个时钟使能、两个时钟信号和两个清除信号。如果需要多于六个时钟使能信号和八个输出使能信号，可配置一些专用 LE 单元作为时钟使能信号或输出使能信号，实现对器件中每个 I/O 的控制。I/O 中的三种寄存器可以被芯片级复位信号复位，而且芯片级复位优于其他控制信号。

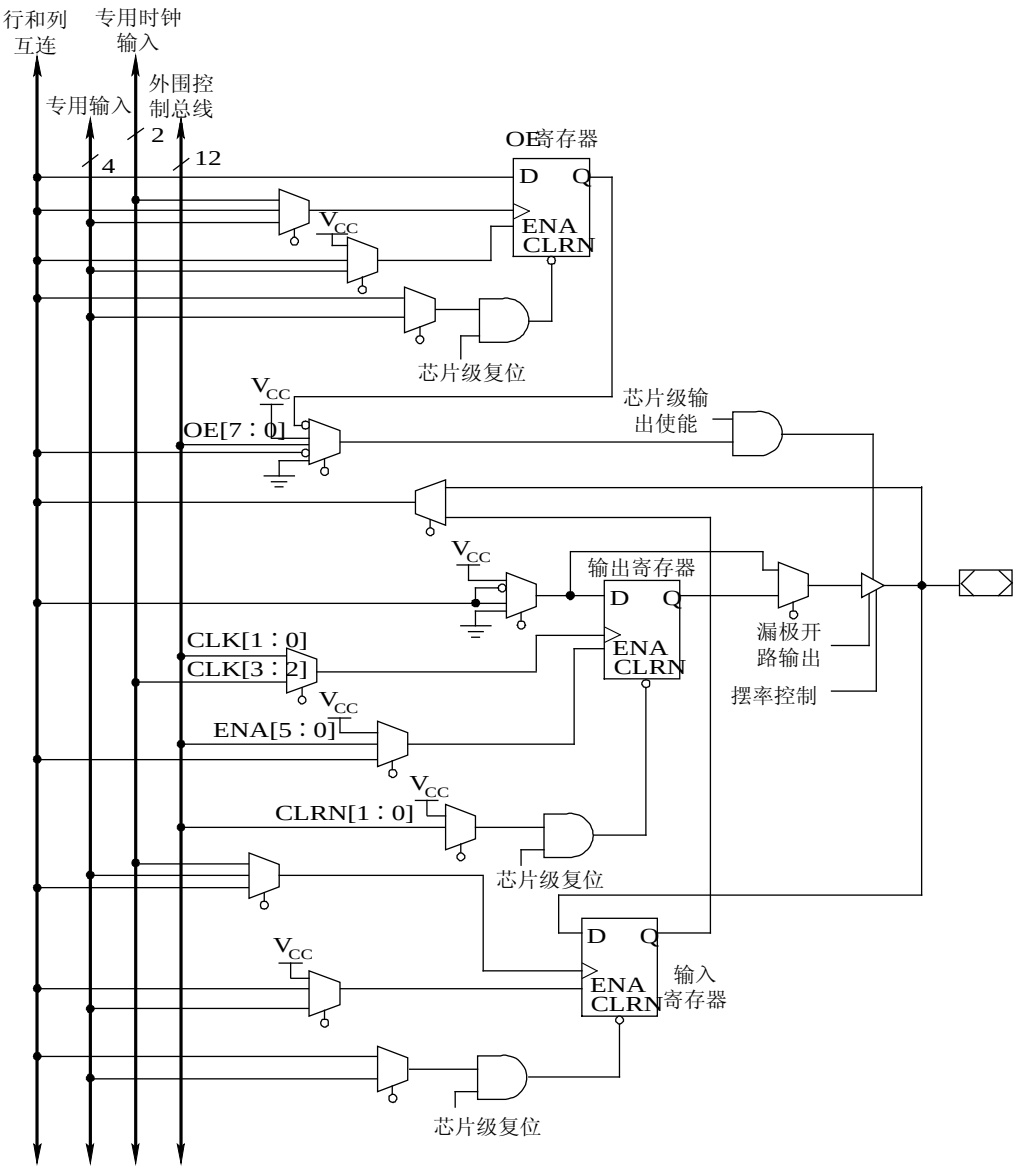



图 2.20　I/O 单元的逻辑框图

I/O 单元的输出缓存器具有可编程的漏极开路输出和可编程输出压摆率控制，I/O 输入信号可直接进入行、列通道，也可输入到输入寄存器，通过时钟信号加载到行、列通道上。来自各个 LAB 块的信号，经行通道或列通道直接或通过输出寄存器输出。

四个“专用输入”信号可作为输入寄存器和输出使能寄存器的控制信号，两个“专用

时钟输入”信号可用作三个寄存器的时钟信号。

FLEX10K 器件提供了六个专用输入引脚。这些专用输入信号分布到整个器件中，可用作全局时钟、清除、置位、外围输出使能和时钟使能控制信号，同时还可用作器件内所有 LAB 和 I/O 的控制信号。这些信号可以来自每个 LAB 局部互连，因此一般数据也可作为这种专用的输入通道，但这种专用输入信号进入到控制信号网络时会产生附加延时。

### 2.3.2 复杂 FPGA 器件

复杂的 FPGA 器件除具有简单的 FPGA 功能之外，片内还嵌入了其他专用电路，如锁相环、DSP、CPU、收发器等专用模块，而且器件的密度高、容量大，能应用在一些复杂的系统设计中。本节主要介绍 Altera 的 APEX20K 系列、APEX Ⅱ系列、Mercury 系列和 Excalibur 系列，近几年推出的新型 FPGA 器件将在下一节中介绍。

#### 1．APEX20K 器件系列

APEX20K/20KE 系列是 Altera 公司较大规模的 2.5 V/1.8 V SRAM 工艺 的 FPGA 器件，集成度从 72.8～239.2 万个门电路，芯片内嵌入有 RAM 块、锁相环(PLL)、JTAG 边界扫描等电路，支持先进的 I/O 标准、多电压 I/O 接口和按内容寻址的存储器(CAM)，兼容 32/64 位、33 MHz PCI 总线。该系列的主要性能见表 2.13。

表 2.13 APEX20K/20KE 器件的主要性能

| 2.5 V | 1.8 V | 逻辑单元(LE)数量 | 嵌入式 RAM 块 | 锁相环(PLL) |
|---|---|---|---|---|
| — | EP20K60E | 2560 | 16×2K 位 | 2 |
| EP20K100 | EP20K100E | 4160 | 26×2K 位 | 2(1) |
| EP20K200 | EP20K200E | 8320 | 52×2K 位 | 2(1) |
| — | EP20K300E | 11 520 | 72×2K 位 | 4 |
| EP20K400 | EP20K400E | 16 640 | 104×2K 位 | 4(1) |
| — | EP20K600E | 24 320 | 152×2K 位 | 4 |
| — | EP20K1000E | 38 400 | 160×2K 位 | 4 |
| — | EP20K1500E | 51 840 | 216×2K 位 | 4 |

注：表中(1) EP20K100、EP20K200、EP20K400 有一个锁相环(PLL)。

APEX20K 系列是一种具有多核结构的 FPGA 器件，这种多核结构把乘积项逻辑、查找表逻辑和嵌入式存储器组合在同一器件中。使用查找表实现数学运算、数字信号处理(DSP)、数据通道以及增强型寄存器等功能；使用乘积项实现高速控制逻辑和状态机；使用嵌入式系统块(ESB)实现多种存储功能，如双口 RAM、FIFO、CAM、ROM 以及 RAM 等。

APEX20K 器件结构如图 2.21 所示，由四输入 LUT、乘积项、存储器、时钟电路、I/O 和一系列纵横贯穿整个行、列的快速通道组成。为了实现大容量、高密度和资源的有效利用，采用了分层的多级互连方式，第一层通过快速通道把宏逻辑阵列块(MegaLAB)、I/O、时钟电路互连在一起，实现粗颗粒快速互连；第二层是通过 MegaLAB 互连通道把一组逻辑阵列块 LAB(LAB 数量 16～24 个，不同的器件每组 LAB 块个数不同)和一个 ESB 块快速地互连在一起；第三层是 LAB 局部互连把 10 个逻辑单元 LE 互连在一起，组成一个 LAB

块，并且把乘积项和存储器组成一个嵌入式系统块 ESB；ESB 和 LAB 之间连接通过 LAB 局部互连线实现。图 2.22 给出了 MegaLAB 的组成。

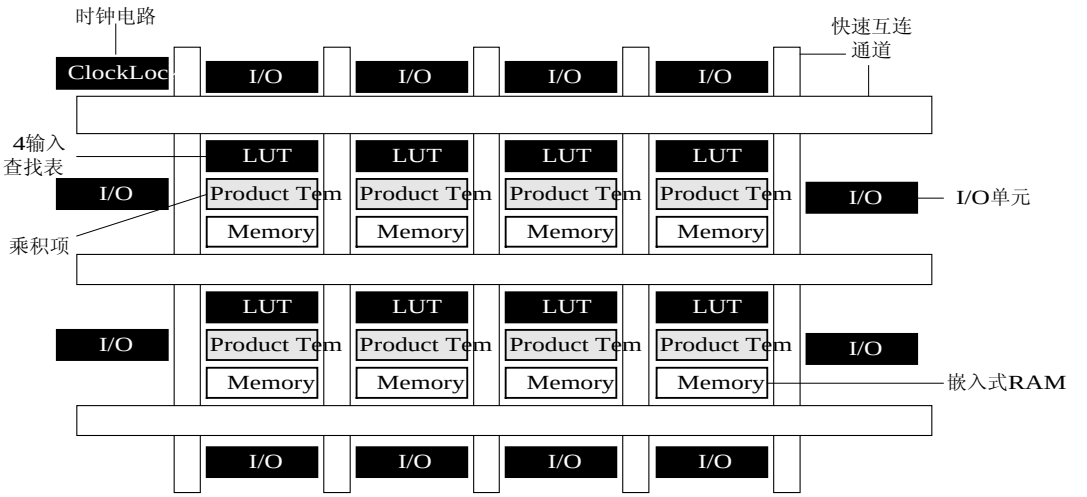



图 2.21　APEX20K 器件结构框图

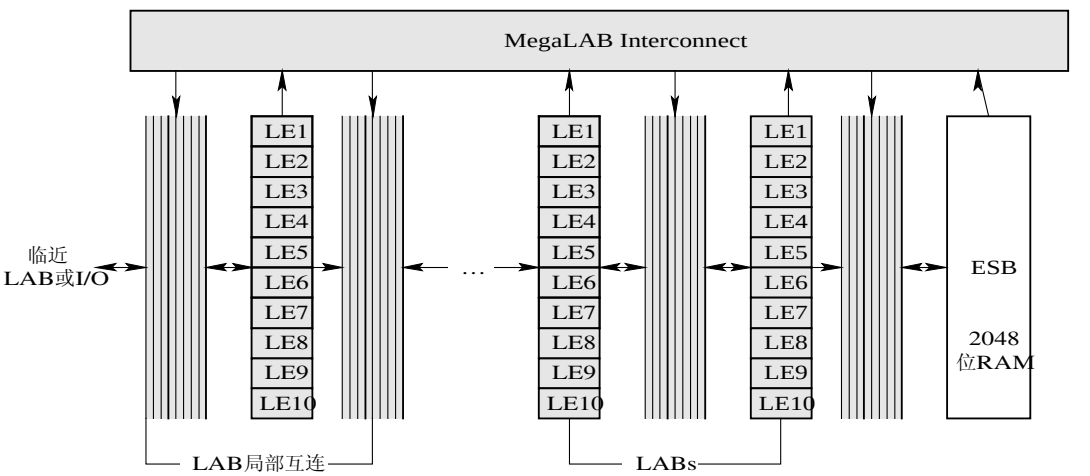



图 2.22　APEX20K 器件的 MegaLAB 的组成

APEX20K 器件 LE 结构和功能与 FLEX10K 系列类似，包含一个四输入的查找表、可编程的寄存器、一个进位链和一个级连链。每个 LE 都能驱动 LAB 局部互连，MegaLAB 互连通道和快速通道。当使用快速互连时，每个 LE、ESB、I/O 能驱动任何其他的 LE、ESB、I/O，实现了资源间的快速互连。另外还有多达八路全局时钟信号和灵活的锁相环管理电路，提供时钟的分频、倍频和锁定功能，锁定功能可减少时钟的延时和相差。

APEX20K 器件的乘积项包含宏单元和并联扩展项，其结构和功能与 MAX9000 系列中的宏单元和并联扩展类似。

在行、列快速通道的两端是 I/O 单元，如图 2.21 所示。每个 I/O 单元包含一个输出缓存器、一个输入寄存器、一个输出寄存器、一个输出使能寄存器和一些辅助电路。它们支

持 PCI 总线标准、JTAG 边界扫描、输出电压的摆率控制、多电压的 I/O 接口以及先进的 I/O 标准。

### 2．APEXⅡ器件系列

APEXⅡ器件是在 APEX20K 基础上开发更大规模的 FPGA 器件，把 LUT 逻辑、乘积项逻辑、存储器和高速 I/O 标准集成在同一器件中。支持 LVDS 接口、PLL 和 CAM 功能，有四个锁相环和八个锁相环输出，适用于高密度设计，器件的密度从 16 640～67 200 个 LE 单元，RAM 达到 416～1120 Kb。表 2.14 列出了 APEXⅡ器件的主要性能。

表 2.14　APEXⅡ器件的主要性能

| 器　件<br>性　能 | EP2A15 | EP2A25 | EP2A40 | EP2A70 |
|---|---|---|---|---|
| 器件门数 | 1 900 000 | 2 750 000 | 3 000 000 | 5 250 000 |
| 典型门 | 600 000 | 900 000 | 1 500 000 | 3 000 000 |
| 逻辑单元(LE) | 16 640 | 24 320 | 38 400 | 67 200 |
| RAM 的 EBS | 104 | 152 | 160 | 280 |
| 普通锁相环(PLL) | 4 | 4 | 4 | 4 |
| 最多 RAM bit 数 | 425 984 | 622 592 | 655 360 | 1 146 880 |
| True-LVDS™ 通道(发送/接收) | 36/36 | 36/36 | 36/36 | 36/36 |
| Flexible-LVDS 通道(发送/接收) | 56/56 | 56/56 | 88/88 | 88/88 |
| 用户 I/O 引脚最多 | 492 | 607 | 735 | 1060 |

### 3．Excalibur 器件系列

Excalibur 器件由嵌入式微处理器部分和 PLD 部分组成。嵌入式微处理器包含了 ARM 公司的 32 位 RISC 处理器 ARM922T 和存储器，PLD 部分由逻辑单元和宏单元组成。表 2.15 列出了 Excalibur 器件的主要性能。

表 2.15　Excalibur 器件的主要性能

| 器　件<br>性　能 | EPXA1 | EPXA4 | EPXA10 |
|---|---|---|---|
| APEX 器件相同结构 | EP20K100E | EP20K400E | EP20K1000E |
| 最大系统门 | 263 000 | 1 052 000 | 1 772 000 |
| 典型门 | 100 000 | 400 000 | 1 000 000 |
| 逻辑单元 | 4160 | 16 640 | 38 400 |
| 嵌入 RAM 块(ESB) | 26 | 104 | 160 |
| 总 RAM 位 | 53 248 | 212 992 | 327 680 |
| 宏单元 | 416 | 1664 | 2560 |
| 最大用户 I/O 引脚 | 178 | 360 | 521 |
| 单口 SRAM/Kb | 32 | 128 | 256 |
| 双口 SRAM/Kb | 1×16 | 2×32 | 2×64 |

4．Mercury 器件系列

Mercury 器件采用先进的 SRAM 工艺，对器件内核性能进行优化，提高了器件的布线能力和速度。与 APEX 和 FLEX 系列相比，该器件主要在内部的布线结构上变化较大，另外集成了 18 个 1.25 Gb/s 含有时钟数据恢复 CDR 的高速差分收发器，并支持多种 I/O 协议，如千兆以太网、光纤 SONET/SDH 协议等。表 2.16 给出了 Mercury 器件的主要性能。

表 2.16　Mercury 器件的主要性能

| 器　件<br>性　能 | EP1M120 | EP1M350 |
|---|---|---|
| 典型门 | 120 000 | 350 000 |
| 逻辑单元 | 4800 | 14 400 |
| CDR 通道 | 8 | 18 |
| RAM 嵌入块(ESB) | 12 | 28 |
| 总 RAM 位 | 49 152 | 114 688 |
| 最大用户 I/O 引脚 | 303 | 486 |
| 封装 | 484 脚 BGA | 780 脚 BGA |

## 2.3.3　新型 FPGA 器件

Altera 的新型 FPGA 器件是指 2002 年之后推出的。它采用先进的 90 nm 或 130 nm SRAM 工艺制造，器件密度高、功能强，并且嵌入了许多专用硬核，具有高性能、模块化的结构。它能灵活地植入各种 IP 软核，尤其是 Altera 可编程 NIOS 处理器软核，可以很容易地实现各种可编程片上系统(SOCP)，极大地满足网络、电信、DSP、海量存储和其他大带宽系统的应用需求。另外新型 FPGA 器件支持 HardCopy 结构化的 ASIC，为低成本大批量的 FPGA 应用找到了一种新的途径。

1．Stratix 器件系列

Stratix 器件从结构上进行了优化，增加了许多专用功能，使器件内核性能、存储能力和架构效率有了很大的提高，满足了高带宽系统的需求，这些专用功能用于时钟管理和数字信号处理(DSP)。Stratix 器件支持差分和单端口 I/O 标准，还具有片内端口匹配和远程系统升级能力，能较好地实现可编程芯片系统(SOPC)。如果需要更高性能、更大容量可以选用功能更强的 StratixⅡ器件。

Stratix 器件采用 1.5 V、0.13 μm 全铜 SRAM 工艺，容量从 10 570～79 040 个逻辑单元(LE)，嵌入 RAM 多达 7 Mb，嵌入乘法器(9×9)多达 176 个，可以组合成 22 个 DSP 块。它具有 True-LVDS 电路，支持 LVDS、LVPECL、PCML 和 HyperTransport 差分 I/O 电气标准及高速通信接口，具有比较完整的时钟管理功能和层次化的时钟结构，可提供多达 12 个锁相环(PLL)。在大批量应用时，Stratix FPGA 设计能很容易地移植到掩码编程的 HardCopy Stratix 器件上以降低设计成本。表 2.17 列出了 Stratix 器件系列产品性能。

表 2.17　Stratix 器件系列产品性能

| 性　能 | 器　件 | | | | | | |
|---|---|---|---|---|---|---|---|
| | EP1S10 | EP1S20 | EP1S25 | EP1S30 | EP1S40 | EP1S60 | EP1S80 |
| 逻辑单元(LE) | 10 570 | 18 460 | 25 660 | 32 470 | 41 250 | 57 120 | 79 040 |
| M512 RAM 块<br>(512 bit＋奇偶校验) | 94 | 194 | 224 | 295 | 384 | 574 | 767 |
| M4K RAM 块<br>(4 Kb＋奇偶校验) | 60 | 82 | 138 | 171 | 183 | 292 | 364 |
| M-RAM 块<br>(512 Kb＋奇偶校验) | 1 | 2 | 2 | 4 | 4 | 6 | 9 |
| RAM 总数 | 920 448 | 1 669 248 | 1 944 576 | 3 317 184 | 3 423 744 | 5 215 104 | 7 427 520 |
| DSP 块 | 6 | 10 | 10 | 12 | 14 | 18 | 22 |
| 嵌入乘法器 | 48 | 80 | 80 | 96 | 112 | 144 | 176 |
| 锁相环 PLL | 6 | 6 | 6 | 10 | 12 | 12 | 12 |
| 最大用户 I/O 管脚数 | 426 | 586 | 706 | 726 | 822 | 1022 | 1203 |

下面介绍 Stratix 器件的具体功能。

1) 高性能的模块化设计

Stratix 器件结构由纵向直列逻辑阵列块(LAB)、TriMatrix 存储块、数字信号处理块(DSP)、锁相环(PLL)和环绕四周的 I/O 单元构成，如图 2.23 所示。这种结构提供快速的互连线和低偏移时钟网络，实现器件内部时钟和数据信号的连接。

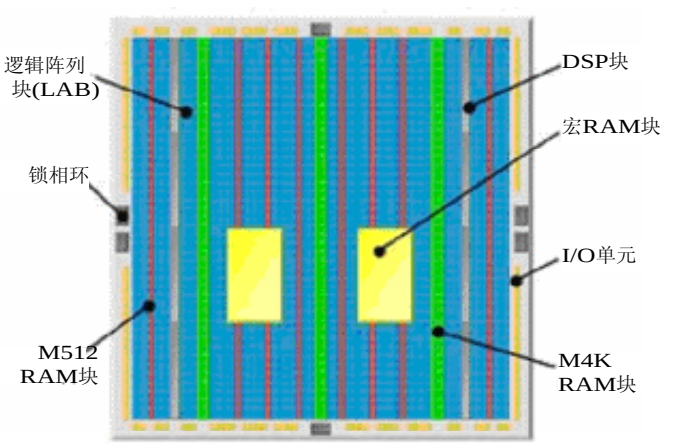



图 2.23　Stratix 器件结构框图

Stratix 器件采用了具有直接驱动(DirectDrive)技术的多轨(MultiTrack)互连线。它由不同长度的布线组成，实现不同设计模块之间的互通。DirectDrive 技术能够保证无论在器件中的什么位置都有一致的布线资源，避免了设计改变引起的系统重新优化过程，简化了模块设计的系统集成过程。使设计者可以自由添加、修改和移动设计的不同部分，而不用担心设计性能下降。

MultiTrack 互连结构在低偏移时钟网配合下，在器件内进行时钟分配，能够在每个区域内访问多达 22 个时钟域。每个 Stratix 器件有多达 16 个全局时钟网，供所有的模块使用。全局时钟可以由内部逻辑、锁相环(PLL)输出或器件输入引脚驱动，也能作为其他大扇出的全局信号，例如异步清除、时钟使能信号等。

2) TriMatrix 存储器

Stratix 器件具有 TriMatrix 存储结构，它包括三种大小的嵌入式 RAM 块。一种是 512 位的 M512 RAM 块，另一种是 4 Kb 的 M4K RAM 块和 512 Kb 的宏 RAM 块(M-RAM 块)，每个块都可实现各种存储特性。TriMatrix 存储器结构提供了多达 7 Mb 的 RAM 和高达 4 Tb/s 的器件存储带宽，满足了大容量、高带宽的存储器设计需求。在应用中 M512 RAM 块常用作 FIFO 缓存器实现时钟域缓冲，宏 RAM 块用作大缓存器，例如互联网的 IP 包缓冲、系统高速缓存等；M4K RAM 块是中等大小缓存的理想选择，比如异步传输模式(ATM)信元处理。表 2.18 列出了高性能 TriMatrix 存储结构中的 RAM 块特性。

**表 2.18　TriMatrix 存储器特性**

| 存储特性 | M512 块<br>512 bit＋奇偶校验 | M4K 块<br>4 Kb＋奇偶校验 | M-RAM 块<br>512 Kb＋奇偶校验 |
|---|---|---|---|
| 最大性能 | 319 MHz | 290 MHz | 287 MHz |
| 真双口存储器 | — | √ | √ |
| 单双口存储器 | √ | √ | √ |
| 单口存储器 | √ | √ | √ |
| 字节使能 | — | √ | √ |
| 奇偶校验位 | √ | √ | √ |
| 移位寄存器 | √ | √ | √ |
| 混合时钟模式 | √ | √ | √ |
| 配置 | 512×1<br>256×2<br>128×4<br>64×8<br>64×9<br>32×16<br>32×18 | 4 K×1<br>2 K×2<br>1 K×4<br>512×8<br>512×9<br>256×16<br>256×18<br>128×32<br>128×36 | 64 K×8<br>64 K×9<br>32 K×16<br>32 K×18<br>16 K×32<br>16 K×36<br>8 K×64<br>8 K×72<br>4 K×128<br>4 K×144 |

注：M512 RAM 块有 64 bit 的奇偶校验位，M4K RAM 块 512 bit 的奇偶校验位，M RAM 块 64 Kb 的奇偶校验位。

3) 支持各种外部存储器接口

Stratix 器件以两种方式满足不断增加的带宽需求。一种是提供 TriMatrix 存储结构，为客户提供丰富的内部存储资源；另一种提供外部存储接口，以增加片外存储资源。用户可以直接将 Stratix 器件同其他厂商的存储器件相连接，如 Micron Technology、Integrated Device

Technology、Samsung Electronics 等厂商的存储器。也可利用第三方解决方案定制的 IP，将大容量的存储器件集成到复杂的系统设计中，采用这种方式不会降低数据存取的性能。表 2.19 给出了 Stratix 器件支持的外部存储器接口。

**表 2.19　Stratix 器件支持的外部存储器接口**

| 外部存储器件 | 最大数据传输速率/(Mb/s) | 存储时钟速度/(MHz) |
|---|---|---|
| 单数据率(SDR) SDRAM | 200 | 200 |
| 双数据率(DDR) SDRAM | 400 | 200 |
| DDR FCRAM | 400 | 200 |
| 零总线转换(ZBT) SRAM | 200 | 200 |
| 四数据率(QDR) SRAM | 668 | 167 |
| QDRⅡ SRAM | 668 | 167 |

4) 数字信号处理(DSP)

Stratix 器件的 DSP 块适用于 Rake 接收机、VOIP 网关、正交频分复用(OFDM)收/发器、图像处理、多媒体系统等方面的应用。在 333 MHz 的速度下，Stratix 器件中 DSP 块的数据吞吐量可达到每个 DSP 块 26.7 亿 MACS(Multiply-accumulates)，并且布线阻塞很小。例如，EP1S80 器件中有 22 个 DSP 块，能够实现高达 586 亿 MACS 的吞吐量。

Stratix DSP 块由硬件乘法器、加法器、减法器、累加器和流水线寄存器组成，如图 2.24 所示。每个 DSP 块能够实现四个 18×18 位乘法，也能够根据不同的应用，选用不同的 DSP 块工作模式，配置成八个 9×9 位乘法，或一个 36×36 位乘法。当 DSP 块配置为 36×36 位模式时，它可以进行浮点运算。DSP 块中的乘法电路支持有符号和无符号的乘法操作，能够在不降低精度的情况下在二者之间切换。

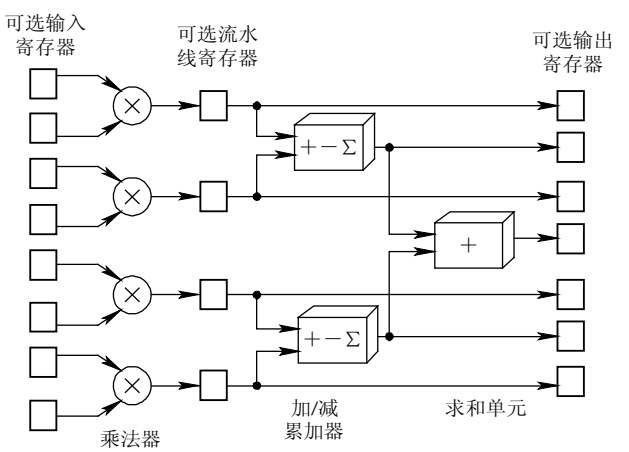



图 2.24　Stratix DSP 组成

加法器/减法器/累加器单元可以根据工作模式配置为一个加法器、一个减法器或一个累加器。这个单元能够自动地在加法器和减法器功能之间切换，根据需求可配置为 9 位、18 位或 36 位加法器。

5) 多功能的 I/O 接口

Stratix 器件支持各种差分和单端标准，很容易同背板、主处理器、总线、存储器件和 3D 图形控制器之间连接。Stratix 器件为设计者提供了多达 116 个高速差分 I/O 通道，其中有多达 80 个通道可达到 840 Mb/s 的速度。每个 I/O 通道都有专用的串行器/解串行器(SERDES)电路，便于实现高速接口标准，如 POS-PHY Level 4 (SPI-4 Phase 2)、SFI-4、FlexBus Level 4、HyperTransport、RapidIO、10 Gbit 以太网(XSBI)和 UTOPIA Level 4，见表 2.20。

表 2.20　Stratix 支持的 I/O 标准

| 特　性 | 单端 I/O 标准 | 差分 I/O 标准 | 外部存储接口 |
|---|---|---|---|
| 电气标准 | LVTTL<br>LVCMOS<br>SSTL<br>HSTL<br>GTL+<br>CTT<br>AGP | LVDS<br>LVPECL<br>PCML<br>HyperTransport | SSTL-2<br>SSTL-3<br>SSTL-18<br>HSTL Class I & Ⅱ<br>Differential SSTL<br>Differential HSTL |
| 专用电路 | Terminator 技术：<br>① 串行匹配<br>② 并行匹配<br>③ PCI 钳位二极管 | True-LVDS 电路：<br>① 专用 SERDES 电路<br>② 差分 I/O 缓冲<br>③ 数据重对齐 | 专用 DDR 电路：<br>① 专用 DQS 电路<br>② DDR 时序电路<br>③ 多个 I/O 寄存器 |
| 相关 IP 核和参考设计 | PCI-X<br>32-/64-bit PCI<br>CSIX<br>DMA 控制器 | POS-PHY Level 4 (SPI-4 Phase2)<br>Flexbus Level 4<br>HyperTransport<br>RapidIO<br>10Gb 以太网(XSBI)<br>Utopia Level 4 | DDR SDRAM 控制器<br>SDR SDRAM 控制器<br>DDR FCRAM 控制器<br>QDR SRAM 控制器<br>ZBT SRAM 控制器 |

Stratix 器件采用专用电路，可以满足高速接口标准的时序要求。该器件中的 True-LVDS 功能具有 SERDES 电路、差分 I/O 缓存器、数据重对齐电路和锁相环 PLL，可进行快速和准确的数据传送。数据重对齐电路紧靠着高速 I/O 缓存器，对齐每个通道上数据字节的边界。数据重对齐功能提供了通道之间或者 Stratix 器件与其他器件之间的数据同步。片内 PLL 为多个时钟域提供了灵活的定时以满足不同的接口规范。PLL 输入时钟可达 645 MHz，支持高速接口标准(如 SFI-4 和 XSBI)。

Stratix 器件差分接口 True-LVDS，支持 LVDS、LVPECL、PCML 和 HyperTransport 差分 I/O 标准。这些差分 I/O 标准有更高的性能、更好的噪声容限、更低的电磁干扰(EMI)和功耗，可支持高速接口标准所需的大数据吞吐量。

Stratix 器件支持的单端 I/O 标准有 LVTTL、LVCMOS、SSTL、HSTL、GTL、GTL+、PCI-X、AGP 和 CTT 等，也支持电路板上其他器件接口。单端系统比差分 I/O 标准提供更大的电流驱动能力，适用于连接高级存储器件，如 DDR SDRAM 和 ZBT SRAM 器件。

6) 系统时钟管理

Stratix 器件提供多达 12 个锁相环(PLL)和 48 个独立系统时钟，可以作为中央时钟管理器

以满足系统时序的需求。它具有高端分立 PLL 器件所具备的 PLL 特性，如扩频时钟、时钟切换、频率合成、可编程相移、可编程延迟、外部反馈和可编程带宽。Stratix 器件还提供 PLL 重配置功能，允许用户改变 PLL 的配置，而无需重新对整个器件编程。如图 2.25 所示。

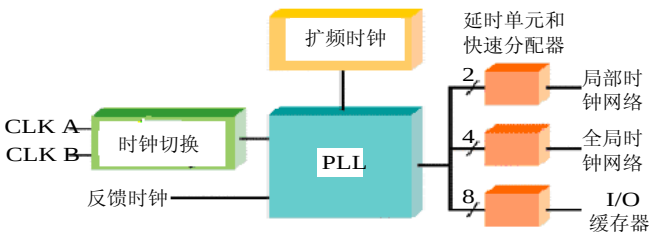



图 2.25　Stratix PLL 工作原理框图

Stratix 器件有两类通用 PLL：增强 PLL 和快速 PLL。增强 PLL 是功能丰富的通用 PLL，支持外部反馈、时钟切换、PLL 重配置、扩频时钟和可编程带宽等特性。见表 2.21。

**表 2.21　Stratix 器件的增强和快速 PLL 特性**

| 特　性 | 增强 PLL | 快速 PLL |
|---|---|---|
| 输入频率范围 | 3～462 MHz | 30～644.5 MHz |
| 输出频率范围 | 0.6～462 MHz | 9～644.5 MHz |
| 可编程相移 | 160 ps | 160 ps |
| 可编程延迟 | 250 ps 增量(1) | — |
| 时钟切换 | √ | — |
| PLL 重配置 | √ | — |
| 可编程带宽 | √ | — |
| 扩频时钟 | √ | — |
| 专用外部差分时钟输出数 | 8(2) | (3) |
| 反馈时钟输入数 | 4(4) | — |
| 每个器件的 PLL 数 | 多达四个 | 多达八个 |

注：

(1) 任何两个输出之间－3.0～3.0 ns 范围内以 250 ps 增量调整。

(2) 每个 Stratix 器件有两个具有八个外部单端或四个外部差分输出的增强 PLL。EP1S40、EP1S60、EP1S80 和 EP1S120 器件中另外两个 PLL 都有一个单端外部输出。

(3) 每个 Stratix GX 器件有两个具有一个外部单端或外部差分反馈输入的增强 PLL。

(4) 快速 PLL 通过高速差分 I/O 管脚驱动输出差分时钟。

Stratix 器件有多达 16 个高性能、低偏移的时钟作为高性能或全局时钟。此外，每个区域有六个本地(区域)时钟，可将任一区域的时钟总数增加到 22 个。Stratix 器件还有两个专用输出的 PLL 能够管理板级系统时序。它提供多达 16 个单端或八个差分输出，这些输出可作为系统中其他器件的时钟源。

7) 远程系统升级

Stratix 器件能从远程进行实时系统升级，可以使用任何通信网络传输远程系统升级数据。在 Stratix 器件中有专用恢复电路保证安全和可靠的远程更新。这种专用电路确保无论

何时发生错误，不论在数据传送还是器件配置期间，均能恢复到能正常工作的状态，确保“永远可操作”的功能，并能将差错的详细情况发送给控制器。图 2.26 是远程系统升级的简单步骤，即为以下三步：

(1) 从开发地点通过网络将升级数据发送给 Stratix 器件。

(2) 将升级数据存放在存储器中。

(3) 用新的数据升级 Stratix 器件。

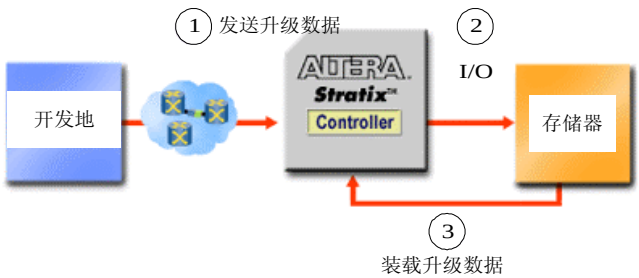



图 2.26　Stratix 远程系统升级步骤

8) 嵌入式处理器核

Stratix 器件的结构特性结合 Nios Ⅱ 嵌入式处理器的处理能力，可满足网络、电信、DSP 应用、海量存储和其他大带宽系统的需求。Nios Ⅱ 处理器为了满足市场的不同应用，提供了三种内核：快速的(Nios Ⅱ /f)、经济的(Nios Ⅱ /e)和标准的(Nios Ⅱ /s)，每种都对应不同的性能范围和成本。它们可完成下列功能：

(1) 实现复杂的状态机。

(2) 分担已有处理器的任务。

(3) 执行 I/O 和数据处理任务。

(4) 远程配置 FPGA。

(5) 加速数字信号处理(DSP)算法。

设计者使用 Altera 的 Quartus Ⅱ 软件、自动系统开发工具(SOPC Builder)以及 Nios Ⅱ 集成开发环境(Nios Ⅱ IDE)，就可以轻松地将 Nios Ⅱ 处理器嵌入到 Stratix 器件中了。SOPC Builder 为设计者提供了一个强大的平台，将一些通用的系统组件(如处理器、外设和存储器接口)组建成一个基于总线的系统。Stratix 器件支持 SOPC Builder 工具实现系统生成。设计者可以通过 SOPC Builder 来添加系统组件，而不会对系统性能造成影响。

SOPC Builder 提供了许多可以定制的外设，例如中断控制器、直接存储存取(DMA)、并行 I/O 块、串行接口和存储器接口。Nios Ⅱ IDE 是一个完整的软件开发环境，可以完成所有的软件开发任务，如源程序编辑、编译和调试。

### 2．Stratix Ⅱ 器件系列

Stratix Ⅱ 器件采用新的逻辑结构和先进的 90 nm 技术，使其容量更大、效率更高。多达 180 K 等效逻辑单元(LE)和 9 Mb 的嵌入存储器，和上一代 Stratix FPGA 相比，平均性能提高了 50%。Stratix Ⅱ 器件结构除了具有创新性逻辑结构之外，还在互连结构和时钟网络方面作了进一步改进，使得连接 LE、TriMatrix 存储块、DSP 块、锁相环(PLL)和 I/O 单元之间

的系统性能更高。表 2.22 给出了 Stratix Ⅱ器件的主要特性。

表 2.22　Stratix Ⅱ器件的主要特性

| 特　性 | 器　件 | | | | | |
|---|---|---|---|---|---|---|
| | EP2S15 | EP2S30 | EP2S60 | EP2S90 | EP2S130 | EP2S180 |
| 自适应逻辑模块(ALM) | 6240 | 13 552 | 24 176 | 36 384 | 53 016 | 71 760 |
| 等效逻辑单元(LE) | 15 600 | 33 880 | 60 440 | 90 960 | 132 540 | 179 400 |
| M512 RAM 块<br>(512 位＋奇偶校验) | 104 | 202 | 329 | 488 | 699 | 930 |
| M4K RAM 块<br>(4 Kb＋奇偶校验) | 78 | 144 | 255 | 408 | 609 | 768 |
| M-RAM 块<br>(512 Kb＋奇偶校验) | 0 | 1 | 2 | 4 | 6 | 9 |
| RAM 总容量 | 419 328 | 1 369 728 | 2 544 192 | 4 520 448 | 6 747 840 | 9 383 040 |
| DSP 块 | 12 | 16 | 36 | 48 | 63 | 96 |
| 嵌入乘法器 | 48 | 64 | 144 | 192 | 252 | 384 |
| PLL | 6 | 6 | 12 | 12 | 12 | 12 |
| 最大用户 I/O 管脚数 | 366 | 500 | 718 | 902 | 1126 | 1170 |

Stratix Ⅱ器件由 LAB 块、DSP 块、M512 RAM 块、M4K RAM 块、M-RAM 块、多功能的 I/O 单元、通用 I/O 单元以及时钟网络和高速互连通道组成，见图 2.27。

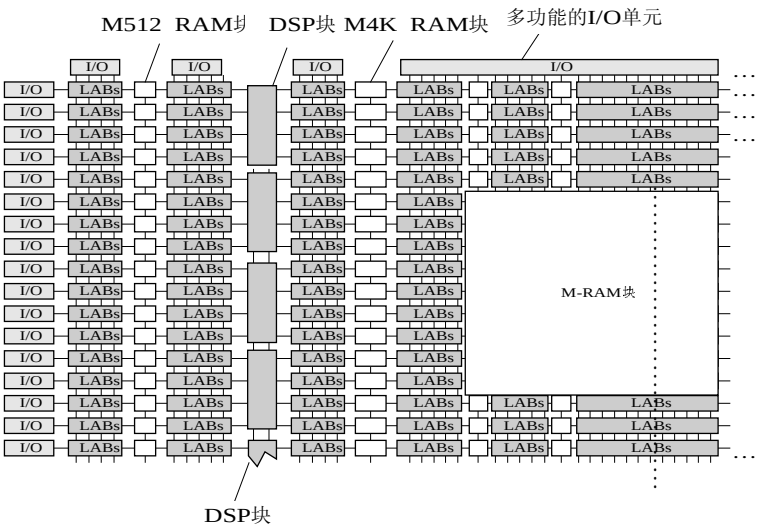



图 2.27　Stratix Ⅱ的结构框图

1) 逻辑块 LAB

LAB 块由八个可调逻辑块(ALM)、进位链、共享算术链、LAB 控制信号、局部互连部分和寄存器链路组成。局部互连部分是 ALM 之间的连接线，寄存器链路是 ALM 寄存器输出到邻近的 ALM 之间的连接。使用 Quartus Ⅱ编译器选择局部互连、共享算术链、寄存器链路等互连功能，可将同一 LAB 或邻近的 LAB 中的逻辑功能综合在一起，实现性能和资源的优化。图 2.28 是 Stratix Ⅱ LAB 的结构图。

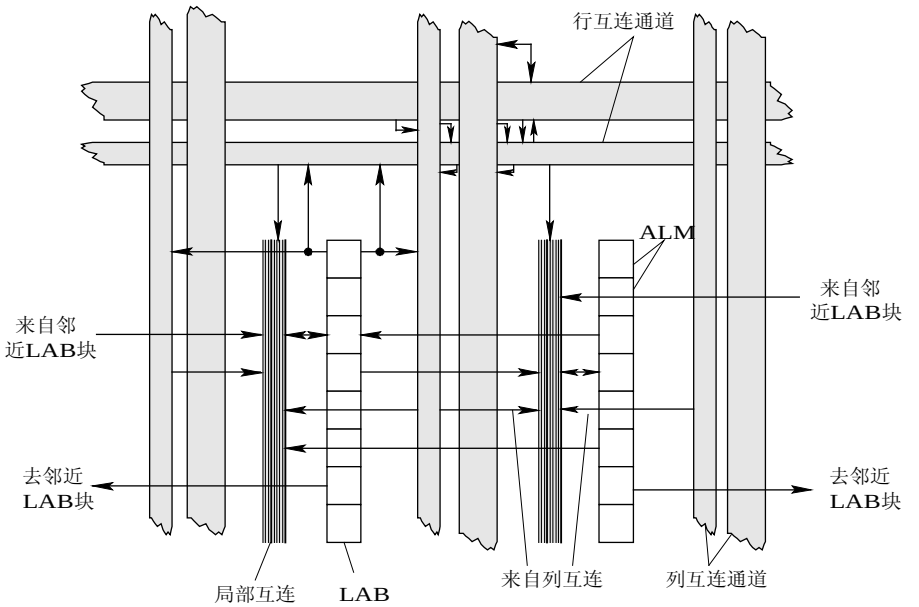



图 2.28　StratixⅡ LAB 的结构图

2) 可调逻辑模块

StratixⅡ结构中基本的逻辑单元是可调逻辑块(ALM)。一个 ALM 基于多个查找表，可以实现任何六输入的查找表 LUT 或某些七输入查找表 LUT 的功能。一个 ALM 包含两个可编程的寄存器、两个全加器、一个进位链、一个算术链和寄存器链。通过这些资源可以实现各种算术运算和寄存器移位，而且可以进行多种互连，见图 2.29。ALM 的工作方式有四种：一般模式、扩展的 LUT 模式、算术模式和共享的算术模式。

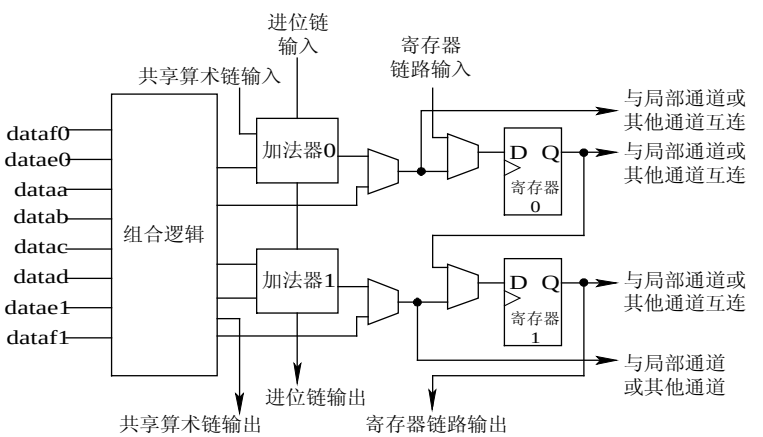



图 2.29　StratixⅡ的 ALM 逻辑结构

3) TriMatrix 存储器

TriMatrix 存储器由三种类型的存储器组成，它们是 M512、M4K 和 M-RAM 存储器。虽然这些存储器块是不同的，但它们能实现带校验位或不带校验位的各种类型的存储器功能，包括真正的双口 RAM、简单双口 RAM、单口 RAM、ROM 和 FIFO。这三种大小不同

的 RAM 有效地满足了不同的应用，通过 Quartus Ⅱ 工具软件能自动划分用户定义的存储器，也能由设计人员手工指定存储器块的大小。

4) 数字信号处理模块 DSP

DSP 用于滤波、快速傅立叶变换、正交余弦变换等，所有这些功能都可用乘法器和一些辅助电路实现，一个 DSP 可以有三种配置：八个 9×9 位的乘法器、四个 18×18 位的乘法器或一个 36×36 位的乘法器。

5) 锁相环(PLL)和时钟网络

Stratix Ⅱ 器件提供分层时钟结构和多种功能的 PLL，可以生成大数量的时钟源和快速的 PLL。有 16 个全局时钟和 32 个局部时钟，这些时钟可形成分层的时钟结构，最多有 48 个独立的时钟域，16 个指定的时钟引脚，可以任意地驱动全局时钟或局部时钟。

Stratix Ⅱ 器件有强大的时钟管理功能和四个增强型 PLL、八个快速的 PLL。锁相环提供先进的时钟接口和时钟频率合成，如时钟转换、带宽调整，相位控制等。器件中嵌入 PLL 为设计者提供了时钟控制和系统定时。快速 PLL 支持高速的差分 I/O 接口，增强型和快速 PLL 一起用于快速 I/O 和时钟网络，以提高系统的性能和带宽。增强型和快速 PLL 的工作原理如图 2.30 和 2.31 所示。

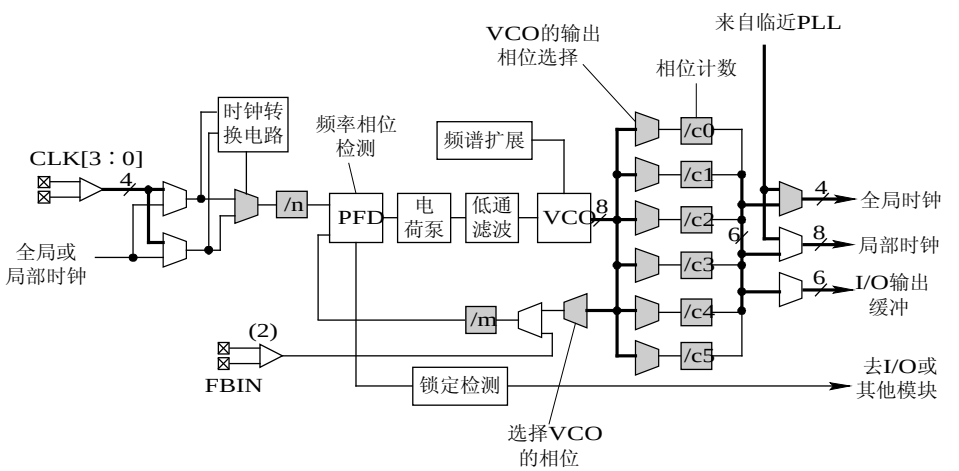



图 2.30　增强型 PLL 的工作原理

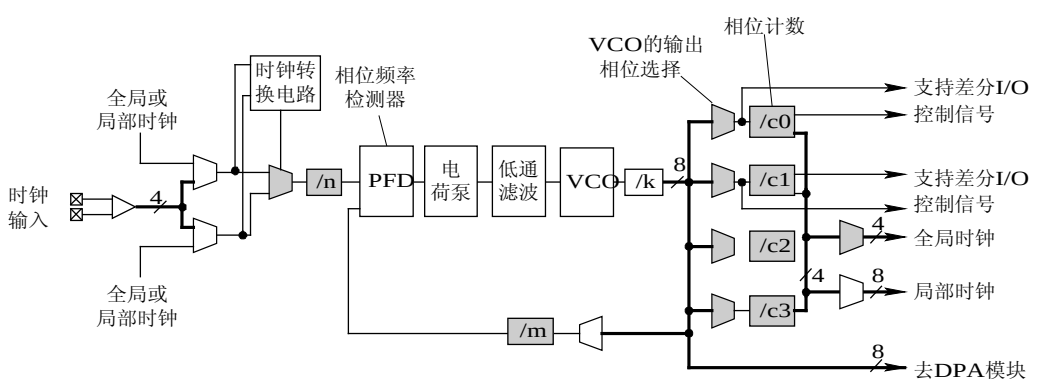



图 2.31　快速 PLL 的工作原理

6) I/O 单元

Stratix Ⅱ的 I/O 由双向的 I/O 缓存器、六个寄存器和一个锁存器组成，如图 2.32 所示。其中有两个输入寄存器，两个输出寄存器和两个使能寄存器。一般使用两个输入寄存器和一个锁存器实现 DDR 输入，用两个输出寄存器驱动 DDR 输出。使能寄存器能够快速地定时，负沿时钟 OE 寄存器用于 DDR SDRAM 接口，Quartus Ⅱ工具软件能自动产生 OE 寄存器实现多路输出和双向输出的功能。

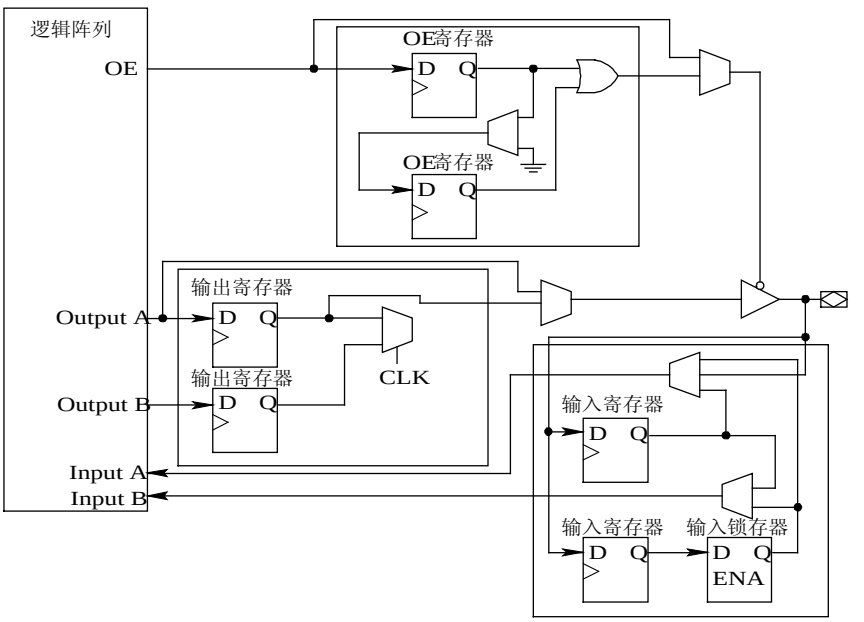



图 2.32 Stratix Ⅱ输入/输出单元

I/O 单元的功能分别如下：

(1) 差分和单端的 I/O 缓冲。

(2) 3.3 V、64 bit、66 MHz PCI 兼容。

(3) 3.3 V、64 bit、133 MHz PCI-X 1.0 兼容。

(4) 支持 JTAG 边界扫描测试。

(5) 输出驱动能力控制。

(6) 三态缓冲。

(7) 总线保持电路。

(8) 可编程的上拉电阻。

(9) 可编程的输入、输出延时。

(10) 漏极开路输出。

(11) 双倍数据速率(DDR)寄存器。

### 3．Stratix GX 器件系列

Stratix GX 器件基于 Stratix 体系。它增加了高性能的数千兆位收发器模块，具有多达 20 个高达 3.125 Gb/s 的全双工收发器通道，满足了高速背板和芯片至芯片通信的需求。该器件还具有：动态可编程预加重、动态可编程接收均衡、动态可编程驱动强度、片内发射

和接收终结功能。

Stratix GX 器件采用 1.5 V、0.13 μm 全铜 SRAM 工艺，容量从 10 570～41 250 个逻辑单元和 3 Mbit 的 RAM。能够实现 1 Gb/s 的源同步差分 I/O 信号，支持 LVDS、LVPECL、3.3 V PCML 和 HyperTransport 差分 I/O 电气标准。也支持不同的高速协议，包括 SerialLite、10 Gbit 以太网(XAUI 和 XSBI)、SONET/SDH、千兆以太网、1 G、2 G 和 10 Gb/s 光纤通道以及串行 RapidIO、SFI-4、POS-PHY Level 4(SPI-4 Phase 2)、HyperTransport、RapidIO、PCI Express、HD-SDI 和 UTOPIA IV 标准。它具有层次时钟结构和多达八个锁相环(PLL)，能提供完整的时钟管理方案。另外还有多达 112 个(9×9 位)乘法器的嵌入式 DSP 块。表 2.23 列出了 Stratix GX 器件的特性。

表 2.23　Stratix GX 系列产品特性

| 器　件<br>特　性 | EP1SGX10C | EP1SGX10D | EP1SGX 25C | EP1SGX 25D | EP1SGX25F | EP1SGX40D | EP1SGX40G |
|---|---|---|---|---|---|---|---|
| 逻辑单元(LE) | 10 570 | 10 570 | 25 660 | 25 660 | 25 660 | 41 250 | 41 250 |
| 全双工收发器通道 | 4 | 8 | 4 | 8 | 16 | 8 | 20 |
| 源同步通道 | 22 | 22 | 39 | 39 | 39 | 45 | 45 |
| M512 RAM 块(512 位＋校验) | 94 | 94 | 224 | 224 | 224 | 384 | 384 |
| M4K RAM 块(4 Kb＋校验) | 60 | 60 | 138 | 138 | 138 | 183 | 183 |
| M-RAM 块(512 Kb＋校验) | 1 | 1 | 2 | 2 | 2 | 4 | 4 |
| 总 RAM 容量 | 920 448 | 920 448 | 1 944 576 | 1 944 576 | 1 944 576 | 3 423 744 | 3 423 744 |
| DSP 块 | 6 | 6 | 10 | 10 | 10 | 14 | 14 |
| 嵌入式乘法器(1) | 48 | 48 | 80 | 80 | 80 | 112 | 112 |
| PLL (2) | 4 | 4 | 4 | 4 | 4 | 8 | 8 |

注：(1) 9×9 位乘法器总数。

(2) 包括增强 PLL 和快速 PLL。

StratixGX 收发器功能块的主要特点如下：

(1) 支持从 622 Mb/s 至 3.125 Gb/s 区间的所有频率。

(2) 支持 3.1875 Gb/s 的 10 Gb 光纤通道。

(3) 集成 SERDES、时钟数据恢复(CDR)、模式检测、字对齐、8 b/10 b 编/解码器和同步功能。

(4) 功耗低，每四通道千兆位收发器块只有 450 mW(单个通道是 150 mW，包括数千兆收发器功能块的功耗)。

(5) 支持动态可编程预加重、均衡和 I/O 缓存器上的差分输出电压(VOD)。

(6) 支持精简的点对点 SerialLite 协议。

(7) 对差分信号有片内匹配电路。

(8) 具有 10 Gb 以太网(XAUI)物理介质接入层(PMA)和物理编码子层(PCS)功能。

(9) 支持灵活的时钟拓扑，包括每个收发器模块中有一个专用发送器锁相环(PLL)和四个接收器 PLL。

(10) 采用 1.5 V、0.13 μm 全铜 CMOS 工艺技术制造，支持 1.5 V PCML I/O 标准。

(11) 具有独立的发送器和接收器节电功能，在不工作时可关闭以减小功耗。

(12) 有内建自检(BIST)功能，包括嵌入伪随机二进制序列(PRBS)图案生成和验证。

(13) 有四个独立的环回路径用于系统验证。

### 4．Cyclone 器件系列

Cyclone FPGA 器件基于全铜 0.13 μm、1.5 V SRAM 工艺，容量从 2910～20 060 个逻辑单元，具有多达 294 912 bit 嵌入 RAM，见表 2.24。Cyclone 支持各种单端 I/O 标准(如 LVTTL、LVCMOS、PCI 和 SSTL-2/3)和差分 I/O 标准(如 LVDS、RSDS 标准)，每个 LVDS 通道可达 640 Mb/s。Cyclone 器件有专用电路实现双数据速率(DDR)SDRAM 和快速周期 RAM(FCRAM)，还有两个锁相环(PLL)提供六个输出时钟和层次时钟结构，以及时钟管理电路。

表 2.24　Cyclone 系列产品性能

| 器　件<br>性　能 | EP1C3 | EP1C4 | EP1C6 | EP1C12 | EP1C20 |
|---|---|---|---|---|---|
| 逻辑单元(LE) | 2910 | 4000 | 5980 | 12 060 | 20 060 |
| M4K RAM 块(4 Kb＋奇偶校验) | 13 | 17 | 20 | 52 | 64 |
| RAM 总量 | 59 904 | 78 336 | 92 160 | 239 616 | 294 912 |
| PLLs | 1 | 2 | 2 | 2 | 2 |
| 最大用户 I/O 数 | 104 | 301 | 185 | 249 | 301 |
| 差分通道 | 34 | 129 | 72 | 103 | 129 |

Cyclone 器件为大批量低成本的应用系统提供了经过优化的功能集。对于需要更大容量和更多功能的设计者，可选用 CycloneⅡ系列。目前 Cyclone 器件的批量价格是每千个 LE 不超过 1.50 美元，这种低成本结构和 Cyclone FPGA 丰富的资源相结合，降低了实现完整的可编程芯片系统(SOPC)的成本，推动了 SOPC 的大批量应用。

Cyclone FPGA 具有以下特性：

(1) 新的可编程构架实现了低成本。

(2) 嵌入式存储资源支持各种存储器应用和数字信号处理(DSP)。

(3) 专用外部存储接口电路，集成了 DDR FCRAM 和 SDRAM 器件以及 SDR SDRAM 存储器件接口。

(4) 支持串行、总线和网络接口及各种通信协议。

(5) 使用 PLL 管理片内和片外系统时序。

(6) 支持常用单端 I/O 标准和差分 I/O 标准。

(7) 支持 Nios Ⅱ 系列嵌入式处理器。

(8) 采用新的串行配置器件，降低了器件配置成本。

#### 5．Cyclone Ⅱ 器件系列

Cyclone Ⅱ 采用全铜层、低 K 值、90 nm、1.2 V SRAM 工艺设计，提供 4608～68 416 个逻辑单元(LE)，嵌入了 18×18 位的乘法器、专用外部存储器接口电路、4 Kb 存储器块、锁相环(PLL)和高速差分 I/O 接口，如表 2.25 所示。

表 2.25　Cyclone Ⅱ 系列器件性能

| 器　件<br>性　能 | EP2C5 | EP2C8 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
|---|---|---|---|---|---|---|
| 逻辑单元 | 4608 | 8256 | 18 752 | 33 216 | 50 528 | 68 416 |
| M4K RAM 块<br>(4 Kb＋512 校验比特) | 26 | 36 | 52 | 105 | 129 | 250 |
| 总比特数 | 119 808 | 165 888 | 239 616 | 483 840 | 594 432 | 1 152 000 |
| 嵌入式 18×18 位乘法器 | 13 | 18 | 26 | 35 | 86 | 150 |
| PLLs | 2 | 2 | 4 | 4 | 4 | 4 |
| 最多用户 I/O 引脚 | 142 | 182 | 315 | 475 | 450 | 622 |
| 差分通道 | 58 | 77 | 132 | 205 | 193 | 262 |

Cyclone Ⅱ 是 Cyclone 系列低成本 FPGA 中的新产品。Cyclone Ⅱ 在 Cyclone 的基础上，采用相同的方法在尽可能小的裸片面积下构建了 Cyclone Ⅱ 系列。与 Cyclone FPGA 系列相比具有更大的密度和更强的功能，进一步降低了单个 LE 的平均成本。

## 2.4　Xilinx 公司产品简介

Xilinx 公司成立于 1984 年，是现场可编程门阵列(FPGA)的发明者，于 1985 年首次推出了商业化的 FPGA 产品。1999 年 Xilinx 收购了 Philips 的 PLD 部门，增强了复杂可编程逻辑器件(CPLD)的研发实力。Xilinx 的主要产品有 FPGA 和 CPLD 集成电路，软件设计工具以及作为预定义系统级功能的 IP 核，是全球最大的可编程逻辑器件供应商之一。

### 2.4.1　Xilinx CPLD 器件

#### 1. XC9500 系列

Xilinx 公司的 CPLD 器件有两个系列：XC9500 系列和 CoolRunne 系列。XC9500 系列采用快闪存技术(Fastflash)，宏单元数可达 288 个，引脚到引脚的逻辑延时可达 3.5 ns，引脚符合 PCI 总线规范，含有 JTAG 测试电路，具有在系统可编程 ISP 功能。XC9500 系列有三种类型，分别为 5 V 的 XC9500、3.3 V 的 XC9500XL 和 2.5 V 的 XC9500XV，常见型号有 XC9536，XC9572 和 XC95144。表 2.26 是 XC9500 系列器件宏单元对照表。

表 2.26　XC9500 系列宏单元对照表

| 5 V | 3.3 V | 2.5 V | 宏单元 |
|---|---|---|---|
| XC9536 | XC9536XL | XC9536XV | 36 |
| XC9572 | XC9572XL | XC9572XV | 72 |
| XC95108 | XC95108XL | XC95108XV | 108 |
| XC95144 | XC95144XL | XC95144XV | 144 |
| XC95288 | XC95288XL | XC95288XV | 288 |

XC9500 系列器件的基本结构相同，每个器件由功能块(FB)、输入/输出块(IOB)和一个开关矩阵(Fastconnect)组成。每个 FB 提供了 36 个输入和 18 个输出的可编程逻辑，IOB 提供了输入和输出的缓冲，开关矩阵将 IOB 与 FB 互连。

每个功能块 FB 由 18 个独立的宏单元组成，每个宏单元可实现一个组合电路或寄存器的功能。FB 除了接收 Fastconnect 的输入外，还接收全局时钟、输出使能和复位/置位信号。FB 的逻辑是利用可编程与阵列来实现。它将来自 Fastconnect 的信号和 FB 块内的互连信号形成 90 个乘积项，通过乘积项分配器分配到每个宏单元。每个宏单元由一个寄存器和一些组合电路构成，寄存器可以配置成 D 触发器或 T 触发器，也可以被旁路。

输入/输出块 IOB 提供内部逻辑电路到 I/O 引脚之间的接口，每个 IOB 包括一个输入缓存器、输出驱动器、输出使能、用户可编程接地和可编程上拉电阻。输入缓存器兼容标准的 5 V 和 3.3 V 信号电压，每个输出有独立的输出压摆率控制，通过编程可以使输出沿变得缓慢以减少系统噪声。

2．CoolRunner 系列

CoolRunner(XPLA3)原是 Philips 的 PLD 产品，1999 年被 Xilinx 收购，其产品的特点是功耗很低，可以用于电池供电系统，供电有 5 V、3.3 V 和 2.5 V 三种，现在被新一代的 CoolRunner Ⅱ取代。CoolRunner Ⅱ是 2002 年推出 1.8 V 低功耗 PLD 产品，它集高性能、低功耗和低成本于一身，适合用于电池供电系统。它还采用了 100%全数字核、性能可达 333 MHz，静态电流小于 100 μA。表 2.27 为 CoolRunner 的宏单元对照。

表 2.27　CoolRunner 系列宏单元对照表

| 5 V | 3.3 V | 2.5 V | 1.8 V | 宏单元 |
|---|---|---|---|---|
| XCR5032 | XCR3032 | XCR3032XL | XC2C32 | 32 |
| XCR5064 | XCR3064 | XCR3064XL | XC2C64 | 64 |
| XCR5128 | XCR3128 | XCR3128XL | XC2C128 | 128 |
| — | — | — | XC2C256 | 256 |
| — | — | — | XC2C384 | 384 |
| — | — | — | XC2C512 | 512 |

CoolRunner 系列器件的组成与 XC9500 系列类似，每个器件由逻辑功能块、输入/输出块和连接功能块的互连矩阵组成。每个逻辑功能块包含一个可编程逻辑阵列 PLA 和 16 个

宏单元。每个宏单元均可在上电时复位或置位，可配置成 D 触发器、T 触发器、锁存器或组合逻辑功能模块。触发器的时钟可以有八个选择，其中有两个全局时钟、一个通用时钟、一个乘积项时钟和四个本地控制信号。输入/输出单元有一个压摆率控制位可以有效地减少电磁干扰，输出与 3.3 V PCI 电气兼容。

### 2.4.2 Xilinx FPGA 器件的特性

Xilinx 自从 1984 年成立以来，推出了一代又一代性能更高，而价格更低的 FPGA 器件，从早期的 XC3000、XC4000 到近年来的 Spartan 和 Virtex FPGA，Xilinx 一直领跑着 FPGA 的发展。目前，Xilinx 公司 FPGA 器件的种类较多，可满足不同用户的需求。

1．SPARTAN 系列

SPARTAN 系列的结构与早期的 XC4000 系列相同，目前 XC4000 主要有 XC4000E(5 V)、XC400XL/XLA(3.3 V)、XC4000XV(2.5 V)，容量从 64～8464 个可配置逻辑块(CLB)，这些早期的产品都可以由相同规模的 SPARTAN 替代。SPARTAN 器件采用 SRAM 工艺，中等规模，结构简单，SPARTAN 系列包含的 CLB 块数见表 2.28。

表 2.28　SPARTAN 系列的 CLB 数

| 5 V | 3.3 V | CLB |
|---|---|---|
| XCS05 | XCS05XL | 100 |
| XCS10 | XCS10XL | 196 |
| XCS20 | XCS20XL | 400 |
| XCS30 | XCS30XL | 576 |
| XCS40 | XCS40XL | 784 |

SPARTANⅡ是 SPARTAN 的升级产品，采用 0.18 μm、2.5 V SRAM 工艺。它在结构上进行了改进，嵌入了 4 K 位的 RAM 存储器块多达 14 个，还嵌入了四个延迟锁相环(DLL)，系统门数最多 200 K，逻辑单元有 432～5292 个，完全兼容标准的 PCI 总线。SPARTANⅡ主要性能如表 2.29 所示。

表 2.29　SPARTANⅡ系列主要性能

| 器　件 / 性　能 | XC2S15 | XC2S30 | XC2S50 | XC2S100 | XC2S150 | XC2S200 |
|---|---|---|---|---|---|---|
| 系统门 | 15 K | 30 K | 50 K | 100 K | 150 K | 200 K |
| 逻辑单元 | 432 | 972 | 1728 | 2700 | 3888 | 5292 |
| BRAM/kbit | 16 | 24 | 32 | 40 | 48 | 56 |
| 延迟锁相环(DLL) | 4 | 4 | 4 | 4 | 4 | 4 |
| 最大分布式 RAM/Kb | 6 | 13.5 | 24 | 37.5 | 54 | 73.5 |
| 最大可用用户 I/O 引脚 | 86 | 132 | 176 | 196 | 260 | 284 |

SpartanⅡE 在 Virtex-E 结构的基础上进行了改进，采用先进的 0.15 μm，1.8 V SRAM 工艺，是 Virtex-E 的低价格版本，中等规模 FPGA 器件。其主要性能如表 2.30 所示。

表 2.30　Spartan-ⅡE 系列主要性能

| 产品 / 性能 | XC2S50E | XC2S100E | XC2S150E | XC2S200E | XC2S300E | XC2S400E | XC2S600E |
|---|---|---|---|---|---|---|---|
| 系统门 | 50 K | 100 K | 150 K | 200 K | 300 K | 400 K | 600 K |
| 逻辑单元 | 1728 | 2700 | 3888 | 5292 | 6912 | 10 800 | 15 552 |
| BRAM/kbit | 32 | 40 | 48 | 56 | 64 | 160 | 288 |
| 延迟锁相环(DLL) | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 最大分布式 RAM/kbit | 24 | 37.5 | 54 | 73.5 | 96 | 150 | 216 |
| 最大用户可用 I/O 引脚 | 182 | 202 | 265 | 289 | 329 | 410 | 514 |

Spartan-3/L 是 2003 年推出的 FPGA 新器件，支持多达 23 种 I/O 标准(包括 LVDS)，嵌入式块 RAM 存储器、范围广泛的 IP(包括 DSP 和处理器内核)以及可同时用于片上和板级时钟管理的数字锁相环。该器件是 Xilinx 公司的一种低成本、高密度的 FPGA 器件。主要性能如表 2.31 所示。

表 2.31　Spartan-3/L 系列主要性能

| 器件型号 / 性能 | XC 3S50 | XC 3S200 | XC 3S400 | XC 3S1000 | XC 3S1500 | XC 3S2000 | XC 3S4000 | XC 3S5000 |
|---|---|---|---|---|---|---|---|---|
| Spartan-3/L | — | — | — | XC 3S1000L | XC 3S1500L | — | XC 3S4000L | — |
| 系统门 | 50 K | 200 K | 400 K | 1000 K | 1500 K | 2000 K | 4000 K | 5000 K |
| 逻辑单元 | 1728 | 4320 | 8064 | 17 280 | 29 952 | 46 080 | 62 208 | 74 880 |
| 18×18 位乘法器 | 4 | 12 | 16 | 24 | 32 | 40 | 96 | 104 |
| 块 RAM/Kb | 72 | 216 | 288 | 432 | 576 | 720 | 1728 | 1872 |
| 分布式 RAM/Kb | 12 | 30 | 56 | 120 | 208 | 320 | 432 | 520 |
| DCM | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| I/O 标准 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 |
| 最大差分 I/O 口 | 56 | 76 | 116 | 175 | 221 | 270 | 312 | 344 |
| 最大单端 I/O 口 | 124 | 173 | 264 | 391 | 487 | 565 | 712 | 784 |

### 2．Virtex 类系列

Xilinx Virtex 类系列从结构上重新定义了可编程逻辑产品，与传统 FPGA 相比，其集成度和功能都达到了一个新的水平，可以解决高性能系统的设计问题。在高端 FPGA 器件中，嵌入了微处理器、高密度片上存储器、串行收发器、数字时钟管理器以及更多其他功能，简化了设计人员的电路设计。

(1) Virtex/Virtex-E：集成密度从 58 K～4 M 的系统门，四输入 LUT 的速度可达 130 MHz，具有高性能的外部 RAM 接口，支持多达 20 种高性能接口和差分 I/O 接口，还

嵌入有全数字锁相环(DLL)和多达 832 kbit 的 RAM 等 IP 硬核。表 2.32 列出了 Virtex/Virtex-E 器件的主要性能。

表 2.32　Virtex /Virtex-E 系列主要性能

| Virtex (2.5 V) | Virtex-E(1.8 V) | 逻辑单元 | RAM 块 | 备　注 |
|---|---|---|---|---|
| XCV50 | XCV50E | 768 | 8/16 | 每个 RAM 块容量是 4 Kb |
| XCV100 | XCV100E | 1200 | 10/20 | |
| XCV150 | — | 1728 | 12 | |
| XCV200 | XCV200E | 2352 | 14/28 | |
| XCV300 | — | 3072 | 16/32 | |
| XCV400 | XCV400E | 4800 | 20/40 | |
| — | XCV600E | 6912 | 24/72 | |
| XCV800 | — | 9408 | 28 | |
| XCV1000 | XCV1000E | 12 288 | 32/96 | |
| — | XCV1600E | 15 552 | 144 | |
| — | XCV2000E | 19 200 | 160 | |
| — | XCV2600E | 25 396 | 184 | |
| — | XCV3200E | 32 448 | 208 | |

(2) Virtex Ⅱ平台 FPGA 系列属于高端器件，支持多种 IP 核，有增强的系统存储器和快速的 DSP，密度范围从 4～800 万系统门。该器件中包含了多达 168 个 18×18 位嵌入式乘法器(可提供高达 0.5 Tera MAC/s 的性能)，多达 12 个 DCM(digital clock manager，提供 16 个全局时钟)和 32 位软处理器。表 2.33 给出了 Virtex Ⅱ系列主要性能。

表 2.33　Virtex Ⅱ系列主要性能

| 器　件 / 性　能 | XC 2V40 | XC 2V80 | XC 2V250 | XC 2V500 | XC 2V1000 | XC 2V1500 | XC 2V2000 | XC | XC 2V4000 | XC 2V6000 | XC 2V8000 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 逻辑单元 | 576 | 1152 | 3456 | 6912 | 11 520 | 17 280 | 24 192 | 32 256 | 51 840 | 76 032 | 104 882 |
| BRAM/Kb | 72 | 144 | 432 | 576 | 720 | 864 | 1008 | 1728 | 2160 | 2592 | 3024 |
| 18×18 乘法器 | 4 | 8 | 24 | 32 | 40 | 48 | 56 | 96 | 120 | 144 | 168 |
| 数字时钟管理块 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 12 | 12 | 12 | 12 |
| 最大分布式 RAM/Kb | 8 | 16 | 48 | 96 | 160 | 240 | 336 | 448 | 720 | 1056 | 1456 |
| 最大用户可用 I/O 引脚 | 88 | 120 | 200 | 264 | 432 | 528 | 624 | 720 | 912 | 1104 | 1108 |

(3) Virtex Ⅱ Pro FPGA 采用 130 nm、9 层铜金属工艺技术，拥有两个嵌入式 IBM PowerPC(TM)405 处理器和多达 16 个工作在 3.125 Gb/s 速率的高速收发器。Xilinx Rocket I/O 收发器是一个完全的串行接口，支持带有 XAUI 的 10 Gb 以太网、PCI Express 和 SerialATA。Virtex Ⅱ Pro FPGA 中嵌入的每个 IBM PowerPC 可以运行在 400 MHz 时钟下，提供 600 DMI/s(Dhrystone Million Instructions executed Per Second)的性能，并支持 IBM CoreConnect 总线技术。Virtex Ⅱ Pro 器件的密度范围从 3～90 K 逻辑单元，速度可达到 400 MHz 时钟速率，包含了多达 444 个 18×18 位嵌入式乘法器，性能如表 2.34 所示。

表 2.34　Virtex Ⅱ Pro 性能对照表

| 器　件<br>性　能 | XC 2VP2 | XC 2VP4 | XC 2VP7 | XC 2VP20 | XC 2VPX20 | XC 2VP30 | XC 2VP40 | XC 2VP50 | XC 2VP70 | XC 2VPX70 | XC 2VP100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 逻辑单元 | 3168 | 6768 | 11 088 | 20 880 | 22 032 | 30 816 | 46 632 | 53 136 | 74 448 | 74 448 | 99 216 |
| BRAM/Kb | 216 | 504 | 792 | 1584 | 1584 | 2448 | 3456 | 4176 | 5904 | 5544 | 7992 |
| 18×18 位乘法器 | 12 | 28 | 44 | 88 | 88 | 136 | 192 | 232 | 328 | 308 | 444 |
| 数字时钟管理块 | 4 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 12 |
| 配置/Mbit | 1.31 | 3.01 | 4.49 | 8.21 | 8.21 | 11.36 | 15.56 | 19.02 | 26.1 | 26.1 | 33.65 |
| PowerPC 处理器 | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| 最大可用 3.125 Gb/s Rocket I/O 收发器* | 4 | 4 | 8 | 8 | 0 | 8 | 12* | 16* | 20 | 0 | 20* |
| 最大可用 10.3125 Gb/s Rocket I/O X 收发器 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 20 | 0 |
| 最大可用用户 I/O 引脚 | 204 | 348 | 396 | 564 | 552 | 644 | 804 | 852 | 996 | 992 | 1164 |

### 3．EasyPath FPGA 系列

Xilinx EasyPath FPGA 是针对大批量生产用户采用的一种解决方案，是 Xilinx 传统 FPGA 产品的延伸。一旦用户认为他们的设计已经确定，客户就可使用 EasyPath 专用的 FPGA 实现批量生产，降低器件的单元成本。用户采用通用的 FPGA 器件进行设计，设计完成后将其对应到相应的 EasyPath FPGA 器件中，设计原型中的 FPGA 器件和 EasyPath FPGA 器件对应关系见表 2.35。

表 2.35　EasyPath FPGA 器件对应表

| EasyPath FPGA | Spartan-3 | Spartan-3L | Virtex-Ⅱ | Virtex-Ⅱ Pro |
|---|---|---|---|---|
| XCE3S1500 | XC3S1500 | XC3S1500L | — | — |
| XCE3S2000 | XC3S2000 | — | — | — |
| XCE3S4000 | XC3S4000 | XC3S4000L | — | — |
| XC3S5000 | XC3S5000 | — | — | — |
| XCE2V3000 | — | — | XC2V3000 | — |
| XCE2V4000 | — | — | XC2V4000 | — |
| XCE2V6000 | — | — | XC2V6000 | — |
| XCE2V8000 | — | — | XC2V8000 | — |
| XCE2VP30 | — | — | — | XC2VP30 |
| XCE2VP40 | — | — | — | XC2VP40 |
| XCE2VP50 | — | — | — | XC2VP50 |
| XCE2VP70 | — | — | — | XC2VP70 |
| XCE2VPX70 | — | — | — | XC2VPX70 |

### 2.4.3 Xilinx FPGA 器件的结构

Xilinx 公司 1984 年提出的 CLB 结构，是可编程器件发展史上的一个里程碑。CLB 中采用 16×1 位的存储器查找表 LUT 的思想，实现四输入函数发生器的功能，再由若干个四输入的函数发生器组合复杂的逻辑电路。将两个四输入的函数发生器、一个三输入的函数发生器、两个寄存器和一些逻辑电路组成一个 CLB 块，作为 FPGA 的基本单元，通过可编程的互连矩阵将 CLB 块互连起来。这种思想解决了可编程器件规模增大，而电路的复杂度成倍增长和资源利用率下降的矛盾，为可编程器件找到了新的发展之路。

FPGA 器件通过加载配置数据到内部的 SRAM 存储器中，实现器件配置的功能，器件可以无数次的重新配置。配置数据可存在外部 PROM 中(串行主模式)，上电后加载到器件内部的 SRAM 中，也可以通过外部器件(串行从模式)将配置设计加载到内部的 SRAM 中。

从 Xilinx 公司 FPGA 的发展来看，FPGA 结构的内部组成在不断地改进，但利用 LUT 实现函数发生器的思想没有改变。为了进一步提高密度，1998 年推出的 Virtex FPGA 重新定义了可配置逻辑块。下面以 Spartan 器件为例，描述 Xilinx 公司的 FPGA 器件结构。

早期的 XC4000 系列和 Spartan 系列器件结构基本相同，如图 2.33 所示。主要由可配置逻辑块 CLB 阵列、输入/输出 IOB 块和高性能的分层布线以及可编程开关矩阵组成，它有丰富的布线资源，能实现 CLB 和 IOB 之间的各种互连。

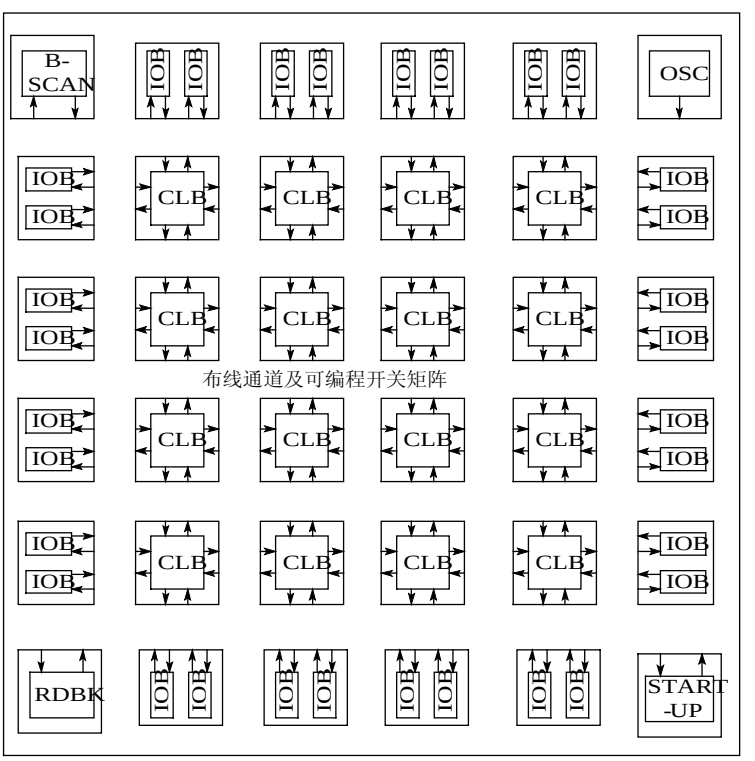



图 2.33　Spartan 系列器件的结构

#### 1．可配置逻辑块 CLB 块

CLB 块阵列主要实现用户的逻辑功能。CLB 块是 FPGA 中可编程的基本单元，每个

CLB 块包含两个四输入的函数发生器、一个三输入的函数发生器、两个寄存器和两组数据选择器。如图 2.34 所示。

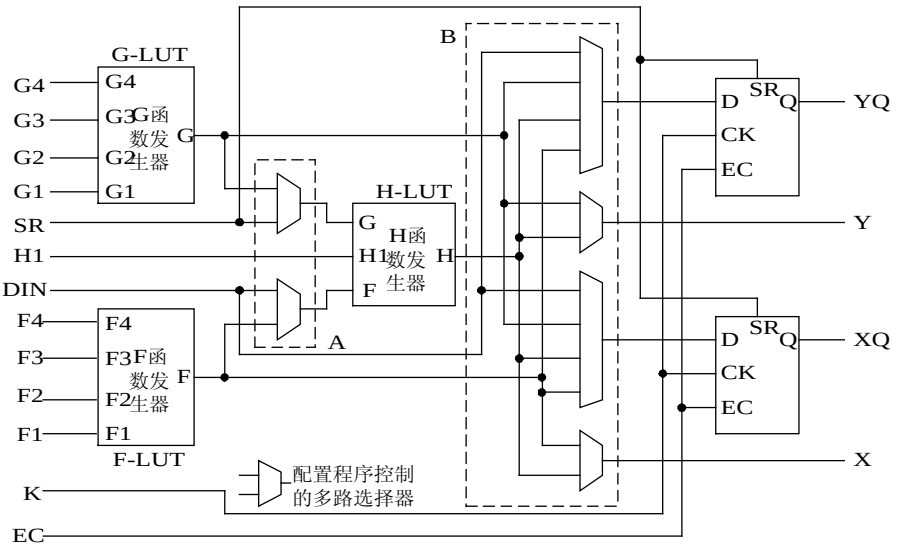



图 2.34　Spartan 系列器件的 CLB 逻辑图

1) 函数发生器

在 CLB 块中有三个函数发生器，F-LUT 和 G-LUT 是两个四输入的查找表。四输入的查找表是一个 16×1 位的存储器，可以实现四输入的任何布尔函数。由于采用 LUT 的方式，因此传输时延与实现的逻辑功能无关，主要由 LUT 的速度确定。H-LUT 是三输入的 LUT，它能实现任何三输入的布尔函数。其中两个输入可以分别是 F-LUT 和 G-LUT 的输出，第三个是 CLB 块的一个固定输入 H1，因此 CLB 块可实现九个变量的函数，也可以将三个 LUT 组合成一个五输入的 LUT，实现任意 5 输入的布尔函数。

2) 寄存器

在图 2.34 中的两个寄存器配置成两个 D 触发器，可以单独使用，也可以存储 F-LUT 和 G-LUT 的输出信号，CLB 的输入 DIN 可直接连到两个触发器的输入端，H1 也可通过 H-LUT 连接到两个触发器中的任意一个。两个沿触发器有共同的时钟(CK)，使能(EC)和置位/复位(SR)输入，并受一个全局初始信号 GSR 的控制(图 2.34 中没有画出此信号)。如果是 SpartanXL 器件，则 CLB 中的寄存器还可以配置成锁存器，两个锁存器有共同的时钟(CK)和使能(EC)输入。

3) CLB 中 RAM 的配置

CLB 阵列中的每个 LUT 都可配置成 RAM 存储器，这种分布式的存储器结构，不但存取速度比片外快得多，而且使用灵活。配置的模式有两种：单端口 RAM 和双端口 RAM。

CLB 单端口 RAM 有三种配置方式：两个 16×1 位的 RAM，一个 16×2 位的 RAM 和一个 32×1 位的 RAM。CLB 双端口 RAM 可以配置成一个 16×1 位的双端口 RAM。

4) 快速进位逻辑

为了提高算术逻辑的速度，CLB 的 F-LUT 和 G-LUT 函数发生器增加了快速进位链，

它与正常的布线资源是相互独立的。这种专用的快速进位逻辑极大地增强了 CLB 实现加法器、减法器、累加器、比较器和计数器的效率和性能，也为许多新应用如数字信号处理和高速偏移地址计算等提供了支持。

图 2.35 为 CLB 块的进位逻辑电路图。图中 F 进位和 F-LUT 可以组成一个带进位的一位全加器，同样 G 进位电路和 G-LUT 也可以实现同样的功能。

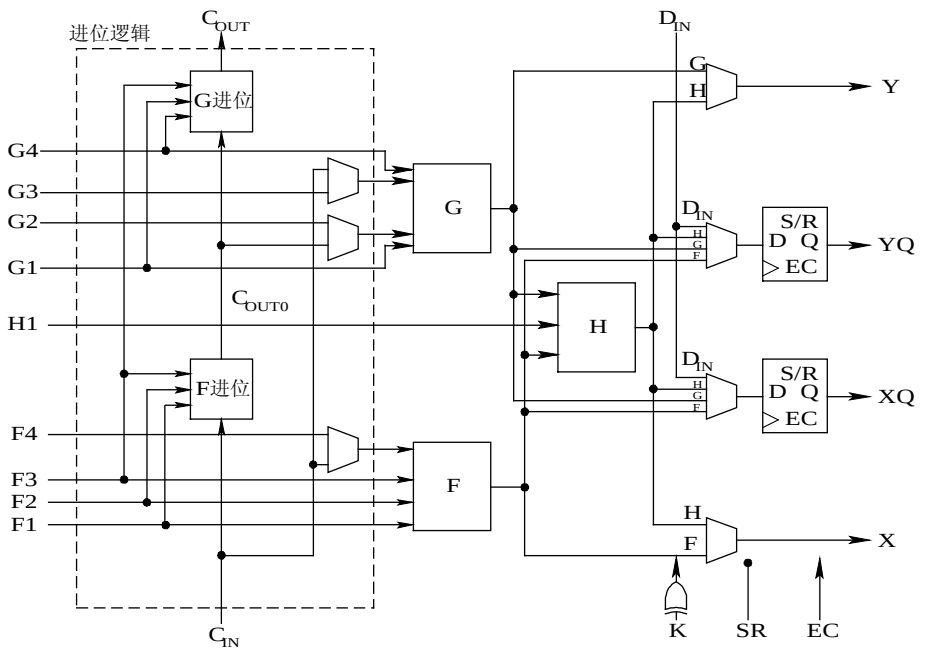



图 2.35　CLB 块的进位逻辑图

CLB 的两个四输入 LUT 和相应的进位链可配置为一个带进位的 2 位加法器，进位链通过组合可扩展到任意长度，可组合成带进位的 8 位加法器、16 位加法器和 32 位加法器等。

### 2．输入/输出块(IOB)

IOB 是 FPGA 器件内部逻辑和外部封装引脚之间的接口，每个接口可配置成输入、输出或双向信号，如图 2.36 所示。图中 EC 信号为输入寄存器使能信号，CLK 为输入寄存器时钟信号，OK 为输出寄存器时钟信号。当 IOB 配置为输入接口时，来自引脚的信号通过输入缓存器进入到输入寄存器，或者通过选择器直接连接到布线通道 I1 和 I2。输入寄存器可作为触发器或锁存器使用。Spartan IOB 输入路径有一个 1 拍延时单元，可以将输入信号配置为 1 拍延时或无时延，SpartanXL IOB 输入路径有一个 2 拍延时单元，可以将输入信号配置为 2 拍延时、1 拍延时或无延时。在器件设计时，可为触发器添加一个 NODELAY 属性，以获得较短的建立时间。

IOB 作为输出接口时，内部的输出信号 O 可在 IOB 中取反，输出信号可直接连到输出缓存器，也可先存放到输出寄存器中，再送到输出缓存器。输出缓存器是一个三态门的缓存器，三态门的控制端由全局初始信号 GTS 和布线通道来的逻辑信号 T 控制，输出三态缓存器实现了接口的双向数据传输功能。输出信号的摆率可编程，以便降低干扰。还有可编程的上拉/下拉电阻，使用户的使用更加方便。

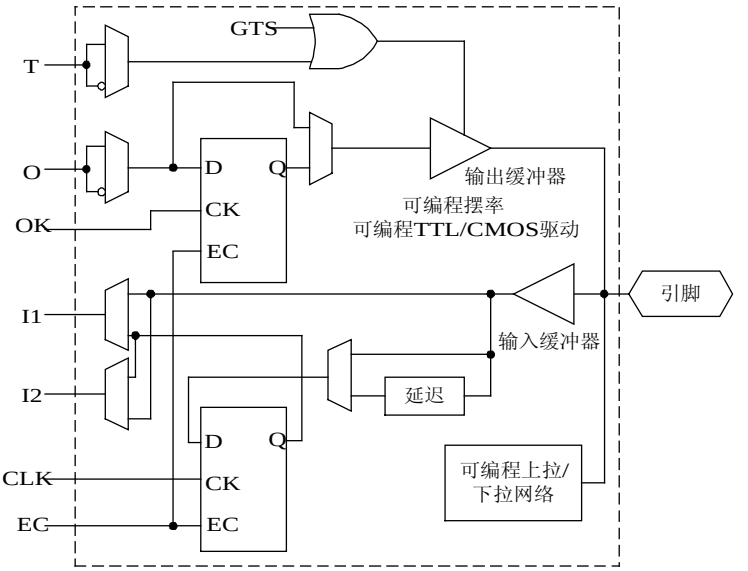



图 2.36　Spartan 器件 IOB 块框图

### 3．布线通道(Routing Channel)

布线通道由可编程的开关矩阵和带有可编程连接点的通道线构成，是一种分层的布线通道矩阵结构，通过布线工具软件为用户提供了高效率的自动布线。如图 2.37 所示。

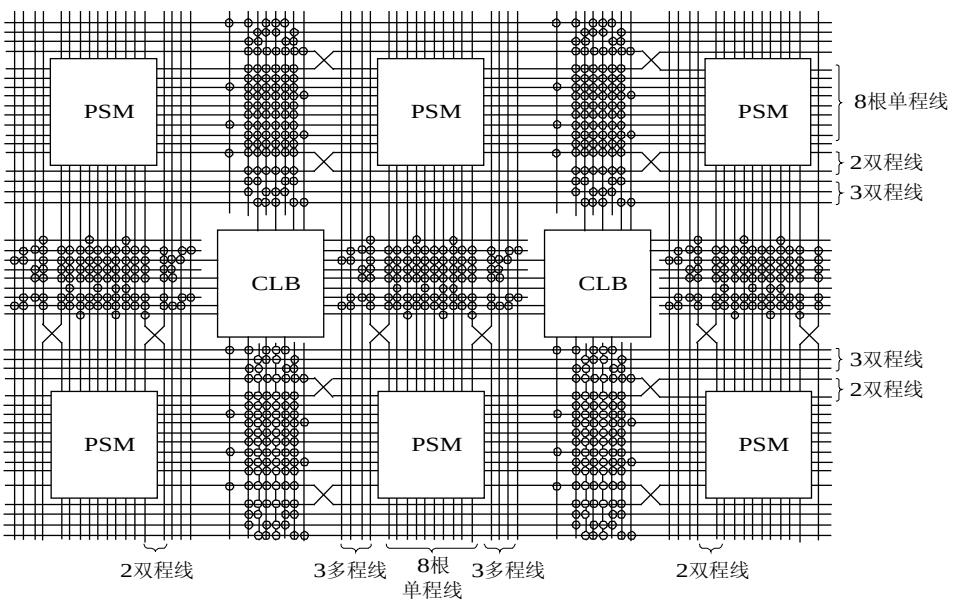



图 2.37　Spartan 布线通道结构图

CLB 的布线通道有三种通道线：单程线(Single-length)、双程线(Double-length)和多程线(Longline)，每个布线通道块的行和列各有八根单程线、四根双程线和六根多程线。单程线

是贯穿 CLB 之间八条垂直和水平的金属线，这些金属线段的交叉处是可编程的开关矩阵(PSM)，通过编程可把 CLB 与其他的 CLB 或 IOB 连接在一起。双程线是四根垂直和水平金属线，其长度是单程线的两倍，要贯穿两个 CLB 之后，才与 PSM 相连。因此利用双程线可使两个相间的 CLB 连接在一起。多程线是六根垂直和水平的金属线，它贯穿整个 CLB 矩阵的行和列，这些多程线不经过开关矩阵 PSM，信号的延时小，用于全局信号或一些关键信号的传输，多程线上有可编程的分离开关，使多程线分成若干个独立的连续通路。单程线和双程线也可通过其线上的编程开关与多程线相连。

图 2.38 是可编程开关矩阵 PSM 的结构图。每个连接点上有六个选通晶体管，从四个不同方向进入节点的信号，可与任何方向的通路互连。

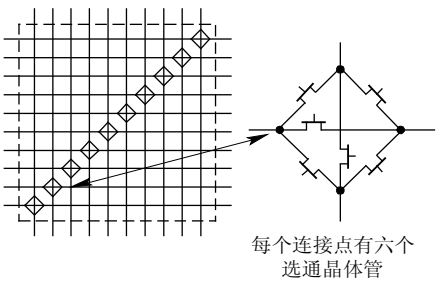



图 2.38　PSM 开关矩阵连接图

## 2.5　Lattice 公司产品简介

Lattice 公司是 ISP 技术的发明者，ISP 技术极大地促进了 PLD 产品的发展，20 世纪 80 年代和 90 年代初是其黄金时期，但很快被 Xilinx 和 Altera 超过。与 Xilinx 和 Altera 相比，其开发工具略逊一筹。中小规模的 PLD 比较有特色，种类齐全。1999 年收购了 Vantis(原 AMD 子公司)，2002 年并购了 Agere 公司的 FPGA 部门，是世界第三大可编程逻辑器件供应商。

PLD 器件通常采用易失的 SRAM 工艺或非易失的 $E^2$CMOS 工艺。易失 SRAM 工艺支持无限重构，非易失 $E^2$CMOS 支持反复编程但不支持无限重构。所以，用户不论选择哪种工艺，都不得不牺牲某一方面的特性。Lattice 公司 2002 年推出了 XP 技术，它基于内嵌 Flash 工艺，是一种能让可编程逻辑器件同时具备非易失性和无限可重构性的技术。

Lattice 公司目前可提供现场可编程门阵列(FPGA)、现场可编程系统芯片(FPSC)和高性能 ISP 可编程逻辑器件(PLD)，包括复杂的可编程逻辑器件(CPLD)、可编程模拟芯片(PAC®)和可编程数字互连器件(GDX®)。

### 2.5.1　Lattice CPLD 器件系列

Lattice 主导产品是高密度 PLD 产品系列。它拥有市场上门类齐全的 CPLD 产品，向用户提供 ispLSI 系列 、ispMACH 系列和 ispXPLD 5000MV/B/C 器件。这些产品的主要性能如表 2.36 所示。

表 2.36　CPLD 系列器件的性能表

| 性能 / 器件 | 工作电压/V | 最高速度/MHz | 最短传播延迟/ns | 逻辑(宏单元) | 引脚数 |
|---|---|---|---|---|---|
| ispMACH 4000V/B/C | 3.3/2.5/1.8 | 400 | 2.5 | 32～512 | 30～208 |
| ispMACH 5000VG/B | 3.3/2.5 | 275 | 3.0 | 128～1024 | 92～384 |
| ispMACH 4000Z | 1.8 | 265 | 3.5 | 32～256 | 32～128 |
| ispMACH 4A5 | 5.0 | 182 | 5 | 32～256 | 32～128 |
| ispLSI 5000VE | 3.3 | 180 | 5 | 128～512 | 72～256 |
| ispXPLD 5000MV/B/C | 3.3/2.5/1.8 | 300 | 4 | 75～300 | 208～672 |

ispMACH 4000Z 系列采用了一种经过优化的器件结构，保证了极低的器件功耗，适用于手提及掌控设备，典型情况下待机电流仅为 10～15 μA。ispXPLD 5000MV/B/C 采用了新的构建模块——多功能块(Multi-Function Block，MFB)，这些 MFB 可以根据用户的应用需要，被分别配置成超宽(SuperWIDE，有 136 个输入)逻辑、单口或双口 RAM、FIFO 或 CAM，并且支持 LVDS、HSTL 和 SSTL 等 I/O 接口标准以及 LVCMOS 标准。此外 sysCLOCK PLL 电路可实现时钟的倍频、分频和时钟移相等功能。

### 2.5.2　Lattice FPGA 产品系列

Lattice 从 2002 年并购 Agere 公司的 FPGA 部门开始，进入了 FPGA 市场，并推出了自主开发的系列产品。目前提供四种 FPGA 产品系列。

1) ORCA FPGA

ORCA FPGA 是在可重构单元阵列(ORCA)结构的基础上开发出来的，它具备了许多先前的 FPGA 所不具备的功能和特点。ORCA FPGA 采用非常灵活的基于 SRAM 的可编程逻辑，具备强大的系统级特性以及丰富的布线层次和互连资源，符合多种接口标准，能够满足功能复杂、性能要求高的设计应用。它的主要性能如表 2.37 所示。

表 2.37　ORCA FPGA 器件的性能表

| 性能 / 器件 | 工作电压/V | 逻辑/LUT | 逻辑/门 | 最大 RAM/Kb | 引脚数 |
|---|---|---|---|---|---|
| ORCA 2 | 5.0/3.3 | 400～3600 | 5～100 K | 58 | 44～128 |
| ORCA 3 | 5.0/3.3/2.5 | 1152～11 552 | 18～340 K | 185 | 44～208 |
| ORCA 4 | 1.5 | 4992～16 192 | 260～1.1 M | 404 | 128～388 |

2) spXPGA 器件

spXPGA 器件能够同时实现非易失性与无限可重构性，通过芯片内的 $E^2$PROM 单元实现微秒级的瞬时上电加载，在几毫秒内重构基于 SRAM 的逻辑，ISP 无需外部的配置存储单元。spXPGA 包括多达 1.25 M 系统门，496 个 I/O 和 414 kbit 的内嵌存储单元，具有可变长度的互连布线、时钟管理 sysCLOCK PLL 电路、用于高性能接口连接的 sysIO 和针对 850 Mb/s 串行通讯的 sysHSI SERDES。如表 2.38 所示。

表 2.38　spXPGA 器件的性能表

| 性能 / 器件 | FPGA 系统门/K | SysHSI 通道* | LUT-4/K | 逻辑触发器/K | 块 RAM/K | 分布式 RAM/K | 最多用户 I/O 数 |
|---|---|---|---|---|---|---|---|
| ispXPGA 125/E | 139 | 4 | 1.9 | 3.8 | 92 | 30 | 176 |
| ispXPGA 200/E | 210 | 8 | 2.7 | 5.4 | 111 | 43 | 208 |
| ispXPGA 500/E | 476 | 12 | 7.1 | 14.1 | 184 | 112 | 336 |
| ispXPGA 1200/E | 1250 | 20 | 15.4 | 30.8 | 414 | 246 | 496 |

注：E 系列不支持 sysHSI (High Speed Serial Interface)。

3) 高容量 FPGA

LatticeECP-DSP(EConomyPlus-DSP)FPGA 器件中的嵌入式 sysDSP 块具有 LatticeEC 结构和 DSP 功能，LatticeEC (EConomy)是一个高容量的 FPGA 器件。主要性能见表 2.39。

表 2.39　LatticeCP/ECP 器件的性能表

| 性能 / 器件 | sys DSP 块* | 18×18 乘法器* | LUT-4/K | 分布式 RAM/K | EBR 块 SRAM/K | EBR SRAM 块 | 最多用户 I/O 数 | PLL |
|---|---|---|---|---|---|---|---|---|
| EC1 | — | — | 1.5 | 6 | 18 | 2 | 112 | 2 |
| EC3 | — | — | 3.1 | 12 | 55 | 6 | 160 | 2 |
| ECP6/EC6 | 4 | 16 | 6.1 | 25 | 92 | 10 | 224 | 2 |
| ECP10/EC10 | 5 | 20 | 10.2 | 41 | 277 | 30 | 288 | 4 |
| ECP15/EC15 | 6 | 24 | 15.4 | 61 | 350 | 38 | 352 | 4 |
| ECP20/EC20 | 7 | 28 | 19.7 | 79 | 424 | 46 | 400 | 4 |
| ECP40/EC40 | 10 | 40 | 41 | 164 | 645 | 70 | 576 | 4 |

* 仅对 ECP 器件。

### 2.5.3　FPSC 产品系列

单片现场可编程系统(FPSC)是将 ASIC 宏单元和 FPGA 集成在同一个芯片内的技术，具有广泛的应用范围，表 2.40 给出了 FPSC 器件的性能。嵌入式宏单元包含了工业标准 IP 核，如 PCI、高速接口和高速收发器。当这些宏单元与成千上万的可编程门结合起来时，可实现各种不同的复杂系统设计。

表 2.40　FPSC 器件的性能表

| 性能 / 器件 | 嵌入核的功能 | 采用 FPGA 技术 | 可配置功能块(PFU) | FPGA 系统门数 | 嵌入式 RAM 位数 | 最大可用 I/O 数 | SERDES | 可用 PLL 数 |
|---|---|---|---|---|---|---|---|---|
| ORLI10G | OIF 标准(OIF 99.102.5) | ORCA 系列 4 | 1296 | 333～643 K | 111 K | 316 | — | 4 |
| ORT82G5/42G5 | 有八个通道的背板收发器 | ORCA 系列 4 | 1296 | 333～643 K | 111 K | 372 | √ | 4 |
| ORT8850L/H | 有八个通道的背板收发器 | ORCA 系列 4 | 624/2024 | (201～397 K)/(471～899 K) | 74 K/148 K | 278/297 | √ | 4 |
| ORSO82G5 | 有八个通道的 SONET 背板收发器 | ORCA 系列 4 | 1296 | 333～643 K | 111 K | 372 | √ | 4 |

ORLI10G 器件嵌入了 OIF 标准(OIF 99.102.5)，它符合 XSBI 10 Gb/s 发送和 10 Gb/s 接收线接口。ORT82G5/42G5 器件的每个通道可在 3.7 Gb/s 的速度下工作，带有内置时钟和数据恢复(CDR)的全双工同步接口。ORT8850L/H 器件的每个通道可在高达 850 Mbit/s(当全部八个通道都被使用时为 6.8 Gb/s)的速度下工作，带有内置时钟和数据恢复(CDR)的全双工同步接口。ORSO82G5 器件的每个通道可在高达 2.7 Gbit/s 的速度下工作，带有内置时钟和数据恢复(CDR)的全双工同步接口。

### 2.5.4 低密度 PLD 产品系列

Lattice 提供门类齐全的低密度 CMOS PLD 产品，又叫 SPLD。18 个系列的 GA 产品提供 200 多种速度、电压、封装和温度范围不同的产品，逻辑门数从 200～1000，典型封装形式为 20、24 和 28 引脚的标准双列直插以及 20 和 28 引脚的标准 PLCC 封装。此外，还提供几个扩展结构的器件，ISP22V10、6001/2、16VP8、16V8Z、18V10、20VP8、20V8Z、20RA10、20XV10 和 26V12，每种结构都针对特定的应用进行了优化。在 3.3 V 满量程的标准结构中，ISP22LV10、16LV8、20LV8、22LV10 和 26CLV12 有多种速度等级，其传播速度延迟低至 3.5 ns，器件的性能良好。

### 2.5.5 其他产品

Lattice 公司还提供了可编程模拟和混合信号产品 ISPPAC，设计人员可以快速、简便地设置电阻值、电容值、增益、信号极性和电路互连以实现多种功能。它们主要应用在电源管理、滤波器和信号调理方面，可以替代许多离散的模拟元器件。用户可以在 PC 机上使用 Lattice 的 PAC-Designer 软件开发工具，实现 ISPPAC 的设计，并对器件进行编程。

Lattice ISPGDX 采用创新的数字交叉点开关结构，将 ISP 技术拓展到了 PCB 板级，其最小传输延迟为 3.0 ns，最多有 256 个输入/输出引脚，完全支持引脚至引脚的信号布线。ISPGDX 广泛应用在数字信号互连和接口方面。

## 2.6 Actel 公司产品简介

Actel 于 1985 年成立，提供多种基于反熔丝及 Flash 技术的 FPGA、高性能的 IP 核、软件开发工具以及设计服务。Actel 产品主要针对高速通信、专用集成电路(ASIC)替代品和航天军品市场。

### 2.6.1 Flash FPGA 器件

Actel 公司的 Flash FPGA 器件经历了三代，第一代是 ProASIC 系列；第二代是 ProASIC$^{PLUS}$ 系列，芯片密度为 75 K～1 M 系统门；第三代是 ProASIC3/E 系列，可提供 30 K～3 M 的系统门。它们均基于非易失的 Flash ROM 技术，在价格、密度和性能方面得到了用户的认可。

#### 1. ProASIC3/E 系列

ProASIC3/E 系列是 Actel 公司的新产品，ProASIC3 器件的密度可达到 3～100 万系统门，有 18～108 K 位真正的双口 SRAM(除了 A3P030)，内核电压为 1.5 V，支持 3.3 V、64

位 66 MHz PCI，81～288 个用户 I/O 引脚和 1.5 V、1.8 V、2.5 V 和 3.3 V 四种电压的 I/O 引脚，I/O 电压的选择可达四个区域。ProASIC3/E 器件结构与 ProASIC3 器件的相同，只是器件的密度更高，可从 60～300 万系统门，有 108～504 K 位真正的双口 SRAM，最多达到 616 个用户 I/O。详见表 2.41 器件性能表。

表 2.41　ProASIC3/E 器件的性能表

| 器件 / 性能 | A3P030 | A3P060 | A3P125 | A3P250 | A3P400 | A3P600 | A3P1000 | A3PE600 | A3PE1500 | A3PE3000 |
|---|---|---|---|---|---|---|---|---|---|---|
| 系统门 | 30 K | 60 K | 125 K | 250 K | 400 K | 600 K | 1 M | 600 K | 1.5 M | 3 M |
| 寄存器 (D 触发器) | 768 | 1536 | 3072 | 6144 | 9216 | 13 824 | 24 576 | 13 824 | 38 400 | 75 264 |
| RAM Kb (1024 bit) | — | 18 | 36 | 36 | 54 | 108 | 144 | 108 | 270 | 504 |
| 4608 bit 块 | — | 4 | 8 | 8 | 12 | 24 | 32 | 24 | 60 | 112 |
| FlashROM (FROM)/Kb | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 安全 (AES) ISP | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 锁相环 PLLs | — | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 |
| 支持 I/O 标准接口 | Std. & Hot Swap | Std.+ | Std.+ | Std.+/ LVDS | Std.+/ LVDS | Std.+/ LVDS | Std.+/ LVDS | Pro | Pro | Pro |

图 2.39 是 ProASIC3/E 器件结构组成。它由通用逻辑块(VersaTile)、嵌入容量为 4608 bit 的 RAM 块、I/O 单元、时钟电路、用户使用的闪存 FROM、加密电路和电荷泵组成。

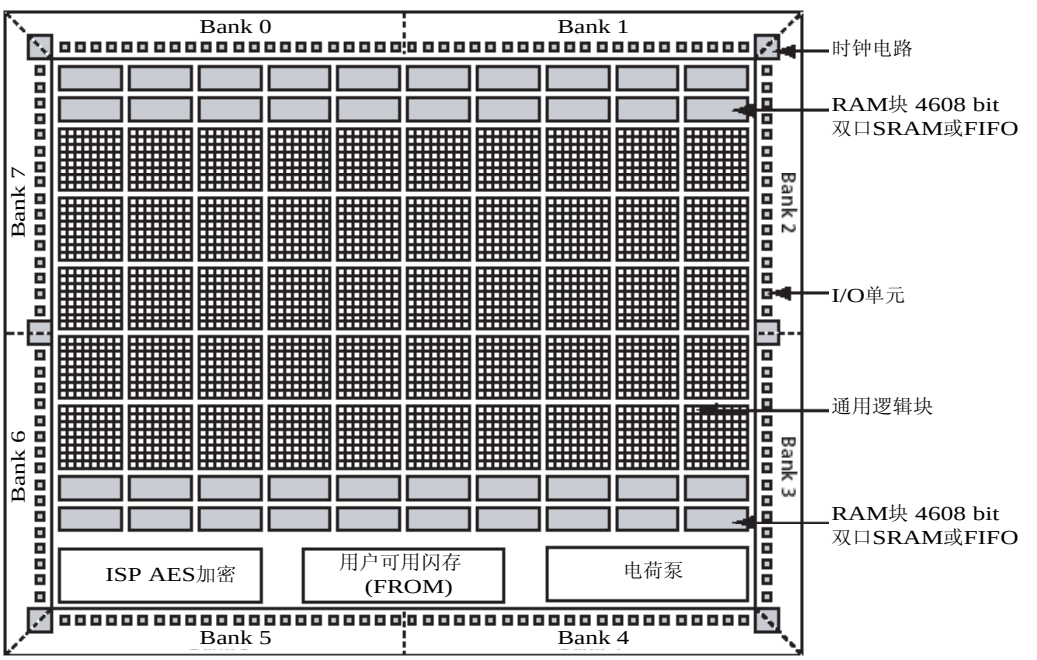



图 2.39　ProASIC3/E 器件结构组成

ProASIC3/E 器件有下列主要特性：

(1) 重复编程的 Flash 技术，采用 130 nm、7 层金属(6 层铜)，基于 CMOS 的处理工艺，断电后信号不丢失，上电时不需加载配置数据，单片结构，使用方便。

(2) 高容量：最多可达 3 M 系统门，504 Kb 的真双口 RAM 和 616 个用户引脚。

(3) 为用户提供了非易失的存储器，用户可使用 1 Kb 的闪存 FROM。

(4) 高性能：片内系统速度可达 150 MHz，64 bit 的 PCI 可达 66 MHz。

(5) ISP 和安全性：使用片内 128 bit 的 AES 加密技术，通过 JTAG 接口，实现安全的在系统编程。采用 Flash 锁定技术保护 FPGA 的内容。

(6) 低功耗：内核采用 1.5 V 电压和低阻抗的 Flash 开关，降低了芯片功耗。

(7) 多层布线：采用分段、分层布线和时钟结构，有超速的局部线和长线的布线网，增强型高速和超长线的布线网和高性能低偏移的全局布线网，使资源利用率和布线效率很高。

(8) 时钟电路和锁相环：有六个时钟电路 CCC(Clock Conditioning Circuit)，内部集成了锁相环 PLL，可实现灵活的时钟倍频、分频和相移功能，输入频率范围从 1.5～300 MHz。

(9) SRAM 和 FIFO 存储器：RAM 块位数配置灵活，可配置为×1、×2、×4、×9 和×18 形式。实现真双口 RAM(除×18 之外)和 FIFO 速度可达 350 MHz。

(10) I/O 接口性能如下：

① 兼容 700 Mb/s DDR、LVDS 的 I/O 接口。

② 支持 1.5 V、1.8 V、2.5 V 和 3.3 V 混合电压工作模式。

③ 有多个 I/O 接口电压选择区域。

④ 支持单端口 I/O 标准：LVTTL、LVCMOS 3.3 V/2.5 V/1.8 V/1.5 V、3.3 V PCI/3.3 V PCI-X 和 LVCMOS2.5 V/5.0 V 输入。

⑤ 支持差分 I/O 标准：LVPECL 和 LVDS。

⑥ 支持参考电压 I/O 标准：GTL+2.5 V/3.3 V、GTL 2.5 V/3.3 V、HSTL、SSTL2 和 SSTL3。

⑦ I/O 单元的寄存器有输入寄存器、输出寄存器和输出三态控制寄存器。

⑧ I/O 接口支持热插拔。

⑨ 输出信号的压摆率和驱动能力可编程。

⑩ 输入信号延时、上拉电阻和下拉电阻均可编程。

⑪ 在单端输入时可选择施密特触发器模式。

⑫ 支持 JTAG 边界扫描功能。

### 2．ProASIC$^{PLUS}$ 系列

ProASIC$^{PLUS}$ 器件采用 0.22 μm、四层金属的 Flash 技术，密度为 75 K～1 M 系统门，包含容量为 256×9 bit 的内嵌 RAM 块，主要性能见表 2.42。

### 3．ProASIC 500K 系列

ProASIC 500K 系列采用 0.25 μm，标准的 Flash/CMOS 工艺，具有低价格、低功耗、非易失和无限重构的特性，提供了一种可预测的、高效的布线结构，与基于 SRAM 的 FPGA 相比有许多优点。表 2.43 是 ProASIC 500K 系列器件的性能。

表 2.42　ProASIC$^{PLUS}$ 器件性能表

| 器件 / 性能 | APA075 | APA150 | APA300 | APA450 | APA600 | APA750 | APA1000 |
|---|---|---|---|---|---|---|---|
| 系统门 | 75 000 | 150 000 | 300 000 | 450 000 | 600 000 | 750 000 | 1 000 000 |
| 寄存器 | 3072 | 6144 | 8192 | 12 288 | 21 504 | 32 768 | 56 320 |
| 嵌入RAM块总位数 | 27 K | 36 K | 72 K | 108 K | 126 K | 144 K | 198 K |
| RAM 块数 (256×9) | 12 | 16 | 32 | 48 | 56 | 64 | 88 |
| LVPECL | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 锁相环 PLL | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 最多用户I/O数 | 158 | 242 | 290 | 344 | 454 | 562 | 712 |

表 2.43　ProASIC 500K 系列器件的性能

| 器件 / 性能 | A500K050 | A500K130 | A500K180 | A500K270 |
|---|---|---|---|---|
| 系统门 | 100 000 | 290 000 | 370 000 | 475 000 |
| 最多 I/O 引脚 | 204 | 306 | 362 | 440 |
| 嵌入 RAM | 14 K | 45 K | 54 K | 63 K |
| 触发器 | 5376 | 12 800 | 18 432 | 26 880 |

## 2.6.2　反熔丝 FPGA 器件

Actel 公司的反熔丝 FPGA 器件是一次性编程器件，由于反熔丝配置一旦编程后便不能改变，因此不会存在固件错误。而基于 SRAM 的 FPGA 器件，会在大气层产生的高能量中子的冲击下，使 SRAM 配置单元内容改变，发生固件错误。器件采用熔丝锁定技术，保障设计人员避免遇到常见的安全性问题，包括克隆、反工程和拒绝服务。

### 1．Axcelerator 系列

Axcelerator 系列器件，采用 0.15 μm、7 层金属反熔丝工艺，基于 Actel 的 AX 架构，提供优于 500 MHz 的内部工作频率和高达 100%的资源利用率。此外，Actel 的上电运行、单芯片 Axcelerator FPGA 可避免涌入电流尖峰，能够简化系统电源设计，与同类型器件比较有更低的待机和动态功耗。

表 2.44 是 Axcelerator 系列器件的性能表。

表 2.44　Axcelerator 系列器件的性能表

| 器件 / 性能 | AX125 | AX250 | AX500 | AX1000 | AX2000 |
|---|---|---|---|---|---|
| 典型门 | 82 000 | 154 000 | 286 000 | 612 000 | 1 060 000 |
| RAM 总位数 | 18 432 | 55 296 | 73 728 | 165 888 | 294 912 |
| 最多寄存器 | 1344 | 2816 | 5376 | 12 096 | 21 504 |
| 模块总数 | 2016 | 4224 | 8064 | 18 144 | 32 256 |
| 专用寄存器 | 672 | 1408 | 2688 | 6048 | 10 752 |
| RAM 块 | 4 | 12 | 16 | 36 | 64 |
| LVDS 差分对 | 84 | 124 | 168 | 258 | 342 |
| 锁相环 PLLs | 8 | 8 | 8 | 8 | 8 |
| 用户 I/O 引脚 | 168 | 248 | 336 | 516 | 684 |

Axcelerator 器件的主要特点如下：

(1) 单片非易失的 FPGA 器件。

(2) 可达到 100%的资源利用率。

(3) 低功耗，内核电压为 1.5 V。

(4) 灵活的多种 I/O 标准。

(5) 片内嵌入 RAM/FIFO 块，配置灵活。

(6) 具有独特的在系统诊断和调试功能。

(7) 支持 JTAG(IEEE1149.1 标准)边界扫描。

(8) 可编程的安全技术，避免了应用过程和设计中的偷窃。

(9) 350 MHz 的系统频率，500 MHz 的内部工作频率。

(10) 兼容 LVDS 700 Mb/s 速率的 I/O 接口。

### 2．SX-A/SX 系列

SX-A/SX 系列器件密度为 12～108 K 系统门，内部工作时钟可达 350 MHz，支持 66 MHz、64 bit、3.3 V/5 V 的 PCI 接口，能够带电热插拔，支持 JTAG 边界扫描。适用在 ATM、IP、WDM、DBE 和 SONET 领域的应用，支持 SONET 标准中 OC3～OC192 的数据速率。表 2.45 是 SX-A/SX 系列器件性能表。

表 2.45　SX-A/SX 系列器件性能表

| 器件 / 性能 | SX08/08A | SX16/16A | SX16P | SX32/32A | SX72A |
|---|---|---|---|---|---|
| 典型门 | 8000 | 16 000 | 16 000 | 32 000 | 72 000 |
| 系统门 | 12 000 | 24 000 | 24 000 | 48 000 | 108 000 |
| 触发器 | 256 | 528 | 528 | 1080 | 2012 |
| 最多 I/O 引脚 | 130 | 177 | 177 | 249 | 360 |
| 逻辑模块 | 768 | 1452 | 1452 | 2880 | 6036 |

### 3．eX 系列和 MX 系列

eX 系列是低功耗小封装的反熔丝器件，芯片密度从 3000～12 000 个系统门不等。容量小、价格低，有三种型号，如表 2.46 所示。MX 系列器件工作在 5 V 的电压下，有很好的性价比，安全可靠，兼容 3.3 V 和 5 V 的 PCI，表 2.46 是 eX 系列和 MX 系列的主要性能表。

表 2.46　Ex 系列和 MX 系列器件性能表

| 器件 / 性能 | MX02 | MX04 | MX09 | MX16 | MX24 | MX36 | eX64 | eX128 | eX256 |
|---|---|---|---|---|---|---|---|---|---|
| 系统门 | 3000 | 6000 | 13 500 | 24 000 | 36 000 | 54 000 | 3000 | 6000 | 12 000 |
| 逻辑模块 | 295 | 547 | 684 | 1232 | 1890 | 2438 | 128 | 256 | 512 |
| 寄存器 | 147 | 273 | 516 | 928 | 1410 | 1822 | 64 | 128 | 256 |
| 最多 I/O | 57 | 69 | 104 | 140 | 176 | 202 | 84 | 100 | 132 |

## 2.6.3　航空航天和军用器件

Actel 公司还提供了高可靠性的器件，主要应用在航空航天和军用领域。根据应用要求的不同将产品分为三类，高强抗辐射(Rad Hard，RH)、抗辐射(Rad Tolerant，RT)和军用及航空。器件在特殊的环境下，经过严格的筛选优化。采用专用的反熔丝技术，为军用航空提供非常高的使用安全性，减少用户风险，降低器件价格。

Actel 公司用于军用航空的器件种类较多，如 RTAX-S 抗辐射系列、RTSX-SU 抗辐射系列、HiRel SX-A 系列、Mil/Aero ProASIC$^{PLUS}$ Flash 系列、Legacy RH/RT/HiRel 器件、54SX 抗辐射系列等产品，详细资料请查找 Actel 公司的数据手册。

# 第 3 章 FPGA 设计入门

由于硬件描述语言 HDL 的标准化和自动化综合工具的不断成熟，基于 FPGA 设计的效率大幅度提高，EDA 界普遍认为有效的建模风格是控制综合结果的最有力的手段。设计(建模)风格的不同，最终得到的综合效果也不同。本章介绍用 Verilog 进行 FPGA 设计时，一些常用设计方法和编写可综合 Verilog 代码规则。

## 3.1 系统的抽象层次与高级硬件描述语言 Verilog

一个系统可以在不同的抽象层次上进行描述，也可以从不同的观点(行为/结构/物理)进行描述，如图 3.1 所示。在 Y-图中有三个轴，每个轴代表一个域，三个轴代表的三个域分别是行为域、结构域和物理/几何设计域。在每个域中，又可以在不同的抽象层次描述一个系统，在 Y-图中这些抽象层次在各自不同的轴上被表示成点。行为域说明一个特定的系统完成什么功能，结构域说明不同的实体之间是如何连接的，物理域则说明如何构造出一个实际的器件。

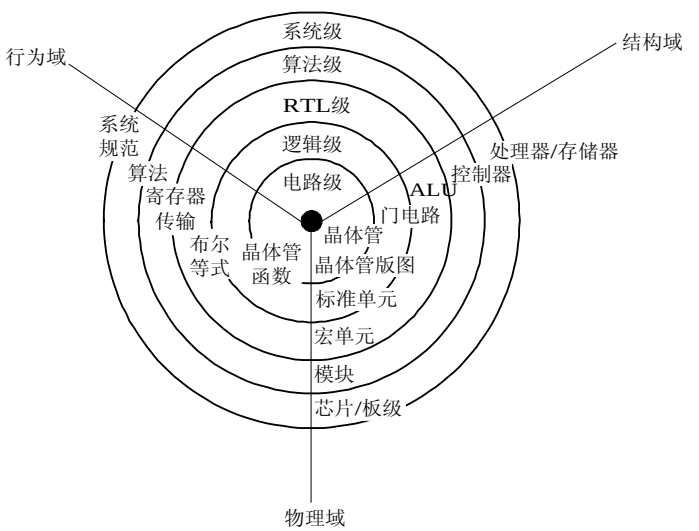



图 3.1 Y-图

从 Y-图上看出，可以在五个层次上描述一个系统，它们分别是：系统级、算法级、RTL(Register Transfer Level，寄存器传输)级、逻辑级和电路级，其中系统级的抽象层次最高，而电路级的抽象层次最低。抽象层次越高，所包含信息就越少，抽象层次越低，所包

含的细节就越多。例如，假设我们要完成一个 10 位的加法器，在系统级我们只要说明设计完成的功能和相关的约束，而不必关心加法器的实现方式，如行波加法器或快速进位加法器，也没有必要关心为了达到相应的速度，采用了几级流水线，或关心为了节约资源而采用的实现方式。

高级硬件描述语言 HDL(Verilog/VHDL) 可以在五个不同的抽象层次上描述一个系统，建立相应的模型。在系统级或体系结构级建立的模型，我们称为行为模型，行为模型可以是系统级模型也可能是算法级的模型。行为模型与具体的硬件实现没有任何关系，它们只是表述被描述对象实现什么样的功能。在 RTL/门级/开关级建立的系统模型我们称为是结构模型。设计人员可以采用自顶向下的设计方法，建立系统的行为模型并验证，然后再用结构模型的源代码替换行为模型的源代码，并对结构模型验证，用同一种语言完成各个设计阶段的任务。

例如，下面是 4 位加法器 Verilog 的行为模型。代码只描述了 4 位二进制加法的行为，但是并没有限定是什么方式实现。示例如下：

```
module adder(a，b，c0，c);
input [3:0] a，b;
input c0;
output [3:0] sum;
output c;
wire [4:0] mid_res;
assign mid_res = a + b + c0;
assign sum = mid_res[3:0];
assign c     = mid_res[4];
endmodule
```

设计初期设计人员更关心系统的性能和特性，在高抽象层次行为级建立系统的模型，可以使设计人员将注意力集中在系统的行为模型的建立和验证上，而忽略具体的实现方式。一旦系统模型正确，可以在行为模型的基础上，进一步细化行为模型，得到结构模型并实现。实际上，除了用 HDL 语言建立系统模型外，设计人员也可以用 C++/C，SystemC 或其他专用的系统建模工具建立系统的模型或验证算法的正确性。

Verilog 建模的最大优点就是与工艺没有关系，便于设计人员的修改设计，极大地提高了设计效率。通常我们用 Verilog 语言在 RTL 级描述一个设计，借助于自动综合工具，设计人员可以将 RTL 级代码快速地变换成逻辑级描述。行为级综合有时也称为高层次综合(High-level Synthesis)，目前，除了在某些特定的应用系统，如数字信号处理，高层次综合工具对行为级描述电路的综合效果没有对 RTL 级描述的电路那么好，而我们大部分的电路是在 RTL 级进行描述的。那么什么是 RTL？

(1) R(Register)：是指存储元件(如锁存器，触发器和存储器等)，这些存储元件接收输入逻辑状态，在定时信号的作用下保存这些逻辑状态。定时信号可以是时钟沿/电平。

(2) T(Transfer)：是指传输，即输入到寄存器、寄存器到寄存器和寄存器到输出的变换。

(3) L(Level)：是指描述设计的抽象级别。

根据上面的解释，一个典型的 RTL 级的模型如图 3.2 所示。

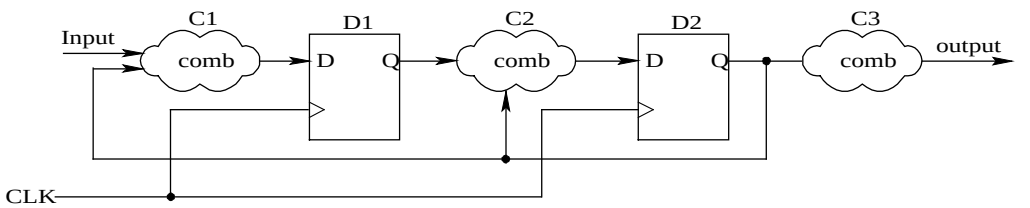



图 3.2　RTL 级描述的典型模型

图 3.2 中的 D1 和 D2 表示存储元件(如锁存器、触发器和存储器等)，C1、C2 和 C3 表示组合电路，相当于信号的传输变换过程。RTL 按照存储器之间的数据流描述一个数字电路，在描述中说明了存储什么样的信息、存储在哪里以及在电路的工作过程中，信息是如何通过电路传递的。

4 位二进制加法器的 RTL 描述如下，它采用快速进位的方式实现加法。示例如下：

```
module   lookahead_adder(a，b，c0，clk，sum，c);
input [3:0] a，b;
input c0;                              //低级进位信号
input clk;
output [3:0] sum;
output c;
wire c0，c1，c2，c3，c4;               //加法进位信号
wire g0，g1，g2，g3;                   //进位产生信号
wire p0，p1，p2，p3;                   //进位传递信号
wire [3:0] sum;
   assign g1 = a[0] & b[0];
   assign g2 = a[1] & b[1];
   assign g3 = a[2] & b[2];
   assign g4 = a[3] & b[3];
   assign p1 = a[0] ^ b[0];
   assign p2 = a[1] ^ b[1];
   assign p3 = a[2] ^ b[2];
   assign p4 = a[3] ^ b[3];
   assign c1 = g0 | c0 & p0;
   assign c2 = g1 | g0 & p1 | c0 & p0 & p1;
   assign c3 = g2 | g1 & p2 | g0 & p1 & p1 | c0 & p0 & p1 & p2;
   assign c4 = g3 | g2 & p3 | g1 & p2 & p3 | g0 & p1 & p2 & p3 | c0 & p0 & p1 & p2 & p3;
   always @(posedge clk)
   begin
      sum <= {p4，p3，p2，p1};
      c    <= c4;
   end
endmodule
```

另外一种结构模型是通过逻辑门及其互连线描述电路，我们把这种级别上描述的电路称为门级描述。在 Verilog 语言中可以通过三种实例化语句描述结构模型(更详细的内容请查阅 Verilog 教程)，它们分别是：

(1) Verilog 支持的内建的基本逻辑门，如与门(and)，与非门(nand)，或非门(nor)，或门(or)，异或门(xor)，异或非门(xnor)，缓存器(buf)和非门(not)等。

(2) 用户定义的原语 UPD。

(3) module 实例语句。

例如，通过内建的逻辑门描述的一位的半加器(见图 3.3)的 Verilog 代码如下：

```
module mux_2_1(a，b，c，sum);
    input a，b;
    output c，sum;
    xor sum_Z (sum，a，b);
    and  c_Z  (c，a，b);
endmodule
```

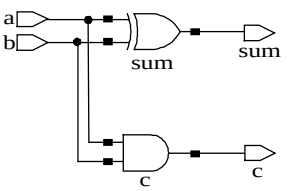



图 3.3　半加器

最低级别的模型是开关模型，一个开关级模型是通过晶体管描述的，而这些晶体管构成了门电路。在 FPGA 设计中不涉及到这个层次，因此不在此讨论。

理解 HDL 语言的抽象层次，对于初学者比较重要。原因在于，我们在进行 FPGA 设计时，从系统的行为模型出发，将设计分解成可以实现的 RTL 模型，良好的 RTL 模型可以使得综合工具更合理地优化你的设计。

## 3.2　用 Verilog 语言建立数字电路模型

设计的最终目标是为了将用 Verilog 所描述的电路设计能通过 EDA 工具映射到具体的物理器件上。Verilog 是功能强大的仿真语言，其中只有一部分子集描述的电路是可以通过 EDA 工具综合的，我们把这部分子集称为可综合子集。EDA 界普遍认为有效的建模风格是控制综合结果最有力的手段，设计(建模)风格的不同，最终会影响到不同的综合效果。为了帮助 FPGA 初学者编写良好的可综合代码，我们给出一些编写 Verilog 可综合代码的建议。

### 3.2.1　代码的书写风格

为了提高源代码的易读性，便于其他人员更快更方便的理解设计，同时减少在工作中由于人为原因造成的信号定义、多位向量顺序颠倒和命名错误等引起的仿真错误，在编写源代码的时候注意以下几点：

(1) 代码的顶部要有版权说明(COPYRIGHT)、文件名(File Name)、文件版本号(File Revision)、发布信息(Release Information)、修改记录(Revise record)等说明。下面以某公司的外部总线接口模块(EBI)源代码为例进行说明，示例如下：

```
//================================================
//      This confidential and proprietary software may be used only as
//      authorised by a licensing agreement from YYY Limited
//        (C) COPYRIGHT 1999 YYY Limited
//              ALL RIGHTS RESERVED
//      The entire notice above must be reproduced on all authorised
//      copies and copies may only be made to the extent permitted
//      by a licensing agreement from YYY Limited.
//------------------------------------------------------------------
//      Version   and   Release Control Information:
//      File Name                  : Ebi.v，v
//      File Revision              : 1.2
//      File Revise                : ** month ** day : Add **signal for C module
//      Release Information        : PL090-REL1v0
//------------------------------------------------------------------
```

(2) 源代码简要功能说明。示例如下：

```
//-------------------------------------------------------------------------
//   Purpose : This block implements the External Bus Interface (EBI)
//-------------------------------------------------------------------------
```

① 定义模块端口信号的时候根据信号的方向分成输入、输出和双向三大部分，用于测试的信号另归为一部分。示例如下：

```
module Ebi1 (// Inputs
               HCLK，
               HRST_Nn，
               BusReq1，
               …
             // Scan Signals
               SCANENABLE，
               SCANIN，
               SCANOUT，
            // Outputs
              BusGnt1，
              BusGnt2，
              BusGnt3，
              …
            // Inouts
              Exp);
```

② 在进行 I/O 说明的时候，根据信号的方向进行分类，并对信号的功能进行简要的注释。示例如下：

```
// Inputs
input        HCLK;        // Bus Clock
input        HRST_Nn;     // Bus Rst_n
input        BusReq1;     // Bus Request 1
…
// Outputs
output       BusGnt1;     // Bus Grant 1
output       BusGnt2;     // Bus Grant 2
output       BusGnt3;     // Bus Grant 3
…
// Inouts
inout Exp;                // Example
```

③ 源代码功能概述，在这部分最好能结合代码中的主要功能块的功能进行描述。示例如下：

```
//-------------------------------------------------------------------
//          Ebi
//-------------------------------------------------------------------
// Overview
//-------------------------------------------------------------------
// The External Bus Interface implements the following functions:
// # Multiplexes the Address and Data lines from 3 separate controllers
//     on to the common Address and Data pins of the chip.
// # Implements the Arbiter state machine to regulate access requests from
//     the Memory / Tic controllers.
// # Re-circulates the data input from the pads to minimize toggling of nets
//     to reduce power-dissipation.
//-------------------------------------------------------------------
```

④ 定义参数，并对该参数的意义和功能进行说明。示例如下：

```
//-------------------------------------------------------------------
// Parameters
//-------------------------------------------------------------------
    parameter CLOCKEDGNT = 0;
// This parameter controls the smallest delay between BusReq* assertion and
// BusGnt* assertion
//-------------------------------------------------------------------
// Constant declarations
//-------------------------------------------------------------------
//-------------------------------------------------------------------
// State encoding for the EBI1 state machine
//-------------------------------------------------------------------
 'define ST_EBI1_IDLE   4'b0001        // Idle state
```

```
'define ST_EBI1_GNT1 4'b0010          // Grant 1 state
'define ST_EBI1_GNT2   4'b0100        // Grant 2 state
'define ST_EBI1_GNT3   4'b1000        // Grant 3 state
```

⑤ 在进行变量声明的时候，根据变量的类型进行分类说明。如果是端口信号要标出信号的方向(输入、输出和双向)，并将同一类方向的信号放在一起。示例如下：

```
//------------------------------------------------------------------
// Integer declarations
//------------------------------------------------------------------
//------------------------------------------------------------------
 // Wire declarations
//------------------------------------------------------------------
   wire              HCLK;
// Bus Clock    (Module input)
wire              HRST_Nn;
// Bus Rst_n (Module input)
wire              BusReq1;
// Bus Request 1 (Module input)
…
//------------------------------------------------------------------
// Register declarations
//------------------------------------------------------------------
reg   BusGnt1;
// Bus Grant 1    (Module output)
reg     BusGnt2;
// Bus Grant 2     (Module output)
reg       BusGnt3;
// Bus Grant 3    (Module output)
reg [31:0] LatchedDataIn;
// Latched ExtDataIn
```

⑥ 在主程序体开始的地方给出标注。示例如下：

```
//------------------------------------------------------------------
// Main body of code
//------------------------------------------------------------------
```

⑦ 注意标识符的命名：在写程序的时候，不要给信号取一些毫无意义的标识符，如 a、b 和 c 等，要根据信号的功能来对信号进行命名。示例如下：

- 信号为高表示正在进行存储器写操作：MemWrite。
- 对上述信号进行锁存后的信号：MemWritelat。
- 总线仲裁状态机信号：ArbState[4:0]。
- 上述状态机的后状态信号：NextArbState[4:0]。
- 表示寄存器写状态：ST_REGWR。

⑧ 模块与模块之间的信号互连方式：由于模块与模块之间，通常需要大量信号相互交汇，因此顶层模块的信号互连工作十分繁重，且极易出错，这种情况对后续工作(仿真和综合)影响很大。Verilog 语言提供显式和隐式两种关联方式，建议使用显式关联方式关联模块之间的信号。

### 3.2.2 可综合代码的编码风格

#### 1. 正确使用阻塞赋值和非阻塞赋值

在 Verilog 中有两种类型的赋值语句：阻塞赋值语句和非阻塞赋值语句。正确地使用这两种赋值语句对于 Verilog 设计和仿真非常重要。下面我们以例子说明阻塞赋值和非阻塞赋值的区别。

**【例 3.1】** 三级移位寄存器的设计。本例将给出移位寄存器三种不同的 Verilog 代码描述，其综合结果分别是图 3.4、3.5 和 3.6。

实现 1：

```
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
```

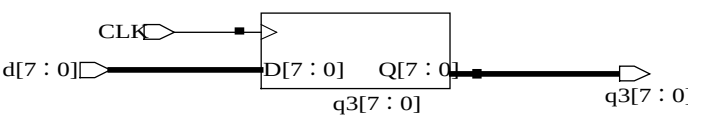



图 3.4 8 位 D 触发器

实现 2：

```
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
```

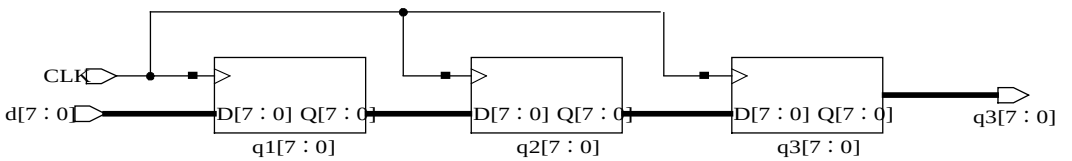



图 3.5　移位寄存器

实现 3：

```
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
    q3 = q2;
    q2 = q1;
    q1 = d;
end
endmodule
```

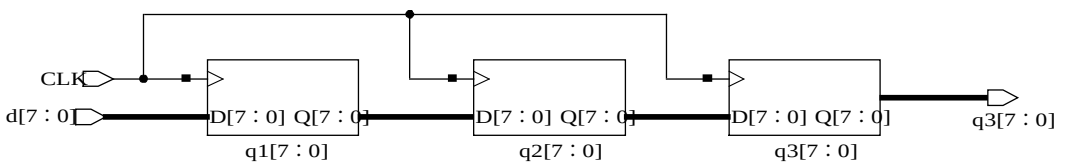



图 3.6　移位寄存器

从这个例子中，我们可以看出，在阻塞赋值语句中，赋值的次序非常重要，而在非阻塞赋值语句中，赋值的次序并不重要。实现 1 和实现 2 使用了相同的赋值次序，但是结果却不同，区别在于它们使用了不同的赋值方式。而实现 3 得到的结果与实现 2 的相同，实现 3 使用了阻塞赋值，但是在实现 3 中明确了移位寄存器的移位的次序。

让我们再回顾一下阻塞赋值和非阻塞赋值的含义。在 always 语句中的“=”赋值我们称为阻塞性过程赋值，在下一语句执行前该赋值语句完成。因此，实现 1 和实现 3 虽然都是阻塞赋值，但是得到的结果却不同。而非阻塞赋值语句被执行时，计算表达式的右端的值赋给左端，并继续执行下一条语句，在当前的时间步结束时或时钟的有效沿到来时候，更新左端的值。在本例中，当时钟的上升沿到达时，更新表达式左端的值。非阻塞语句的执行可以归纳成以下两点：

(1) 在仿真周期的开始，计算赋值号右边表达式(RHS)的值。

(2) 在仿真周期的结束，更新赋值号左边变量(LHS)的值。

为了更清楚地了解两种赋值的区别，我们再举一个例子加以说明。

【例 3.2】

实现 1：

```
module fbosc2 (rst_n，clk ，y1，y2);
input rst_n，clk;
output y1，y2;
reg y1，y2;
  always @(posedge clk or negedge rst)
    if (~rst) y1 = 0; // reset
    else     y1 = y2;

   always @(posedge clk or negedge rst)
     if (~rst) y2 = 1; // reset
     else     y2 = y1;
endmodule
```

实现 2：

```
module fbosc2 (rst_n，clk ，y1，y2);
input rst_n，clk;
output y1，y2;
reg y1，y2;
  always @(posedge clk or posedge rst)
    if (~rst) y1<= 0; // reset
    else       y1 <= y2;

   always @(posedge clk or posedge rst)
     if (~rst) y2 <= 1; // preset
     else     y2 <= y1;
endmodule
```

【例 3.2】中的实现 1 和实现 2 的差别只是使用了不同的赋值语句，其仿真结果却大相径庭。实现 1 的结果与两个 always 语句执行的顺序有关，如果 rst_n 结束后第一个 always 语句先执行，那么 y1、y2 的值均为 0；如果第二个 always 先执行，那么 y1、y2 的值均为 1；其后保持不变。实现 2 的结果是 y1 和 y2 是时钟信号 CLK 的两分频，y1 和 y2 相位相差 180°。

关于阻塞语句和非阻塞语句，有以下的使用建议：

(1) 在描述组合电路时，使用阻塞赋值语句。当在 always 过程中建立组合电路时，许多人也喜欢用非阻塞赋值。如果在 always 语句中只有一个赋值语句，那么使用阻塞赋值和非阻塞赋值的结果是一样的。但是如果使用多条赋值语句，写法不当，可能会导致仿真结果不正确。

【例 3.3】 本例是一个组合电路，但是仿真结果是不正确的。因为非阻塞赋值在更新 LHS 之前计算 RHS 的值，因此，tmp1 和 tmp2 使用的是 a 和 b 进入 module 时的旧值而不是在放置结束时的新值。

```
module ao4 (a，b，c，d，y);
input a，b，c，d;
output y;
reg y，tmp1，tmp2;
always @(a or b or c or d)
begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  y      <= tmp1 | tmp2;
end
endmodule
```

要想使上面的电路正确工作，必须把 tmp1 和 tmp2 也加入到敏感变量表中。由于 tmp1 和 tmp2 的变换引起 always 语句的重新计算，因此结果正确，但这样导致仿真器多次对 always 过程进行计算，会降低仿真器的性能。示例如下：

```
module ao5 (a，b，c，d，y);
input a，b，c，d;
output y;
reg y，tmp1，tmp2;
always @(a or b or c or d or temp1 or temp2)
begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  y <= tmp1 | tmp2;
end
endmodule
```

但是如果 ao4 用阻塞赋值语句，那么结果就是正确的。这是因为 tmp1 和 tmp2 在当前时间步的更新导致了 y 的计算。示例如下：

```
module ao6 (a，b，c，d，y);
input a，b，c，d;
output y;
reg y，tmp1，tmp2;
always @(a or b or c or d) begin
  tmp1 = a & b;
  tmp2 = c & d;
  y = tmp1 | tmp2;
end
ehdmodule
```

(2) 用一个过程(always)描述时序电路时，使用非阻塞赋值语句。当组合电路和非组合电路在一个过程中描述时，也使用非阻塞赋值语句。示例如下：

```
module nbex1 (rst_n，clk，a，b，q);
input rst_n ，clk;
input a，b;
output q;
reg q;
always @(posedge clk or negedge rst_n)
if   (!rst_n) q <= 1'b0;
     else      q <= a ^ b;
endmodule
```

更好的一种代码风格是将组合电路和非组合电路分别描述。示例如下：

```
module nbex1 (rst_n，clk，a，b，q);
input rst_n ，clk;
input a，b;
output q;
reg q;
wire y;
assign y = a ^ b;
always @(posedge clk or negedge rst_n)
  if (!rst_n)   q <= 1'b0;
  else            q <= y;
endmodule
```

### 2．组合电路设计

1) 可综合组合电路的描述形式

用 Verilog 语言可以有多种方式描述组合电路，但是有些综合工具并不支持每一种描述方式。大多数综合工具可综合下面三种形式描述的组合电路。

(1) 通过结构原语描述电路。

**【例 3.4】** 下面的代码是用原语描述的一个组合电路。

```
module stru_des(y_out1，y_out2，a，b，c，d);
output y_out1，y_out2;
input a，b，c，d;
and (y1，a，c)
and (y2，a，d);
and (y3，a，e);
or   (y4，y1，y2);
or   (y_out1，y3，y4);
and (y5，b，c);
```

```
and (y6，b，d);
and (y7，b，e);
or   (y8，y5，y6);
or (y_out2，y7，y8);
endmodule
```

上面描述的电路设计如图 3.7(a)所示，但是经过综合器综合后，删除了冗余的逻辑，电路简化成图 3.7(b)。对于大多数设计者而言，简化一个这样的电路是比较困难的，但综合器可以保证逻辑设计的最小化。

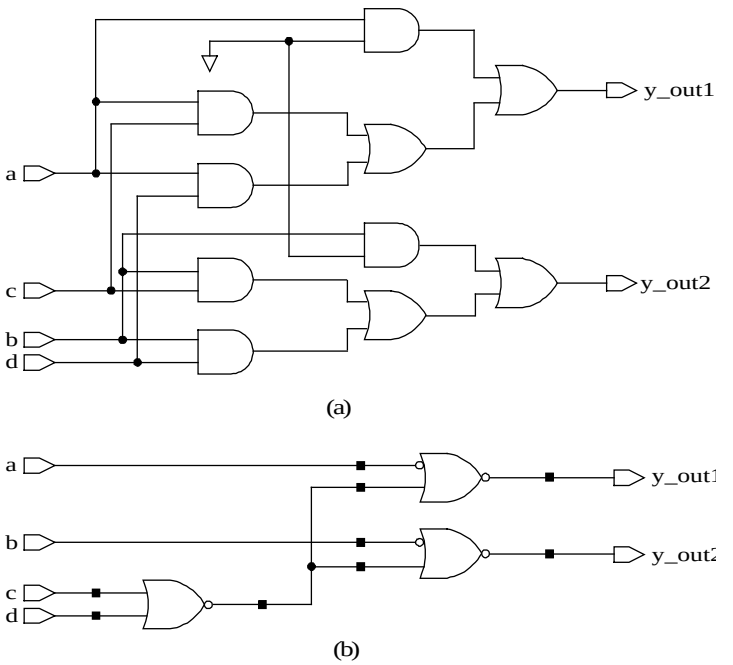



图 3.7

(a) 原语表示的电路；(b) 综合后的电路

(2) 通过连续赋值的方式描述一个组合电路。综合器可以把用连续赋值描述的电路翻译成布尔等式并优化。

**【例 3.5】** 四选一的多路选择器。

```
module   mux_4_1 (a，b，c，d，sel，out);
input a，b，c，d;
input [3:0] sel;
output out;
assign out = (sel[0]) ? a : (sel[1]) ? b :(sel[2]) ? c    : sel[3]) ? d : 1'b0;
endmodule
```

(3) 用 always 语句描述组合逻辑。值得注意的是在这种描述中，对每种输入组合输出都必须有一个值与之对应，否则会导致锁存器产生。

【例 3.6】 4 位的比较电路，给定两个 4 位输入 a、b。如果 a＞b，则 a_gt_b 输出为高电平；如果 a＜b，则 a_lt_b 输出为高电平；a＝b，则 a_eq_b 输出为高电平。

```
module comp_4 (a_gt_b，a_lt_b，a_eq_b，a，b);
input[3:0] a，b;
output a_gt_b，a_lt_b，a_eq_b；
reg a_gt_b，a_lt_b，a_eq_b;
always @(a or b)
begin
    a_gt_b=0;
    a_lt_b = 0;
    a_eq_b = 0;
    if  ( a==b)   a_eq_b = 1;
    if  ( a> b)    a_gt_b = 1;
    if  (a < b)    a_lt_b = 1;
end
endmodule
```

本例可以用另外一种方式描述:

```
module comp_4 (a_gt_b，a_lt_b，a_eq_b，a，b);
input[3:0] a，b;
output a_gt_b，a_lt_b，a_eq_b；
reg a_gt_b，a_lt_b，a_eq_b;
always @(a or b)
begin
    if  ( a==b)   a_eq_b = 1;
    else a_eq_b = 0;
    if  ( a> b)    a_gt_b = 1;
    else a_gt_b = 0;
    if  (a < b)    a_lt_b = 1;
    else a_lt_b = 0;
end
endmodule
```

在上面的描述中，包含了“>”、“<”、“==”、“？”等操作符，在 Verilog 语言中包含了很多类似的操作符，这些操作符中的一部分可以通过综合器直接映射成工艺库中的某些电路。不同的综合器支持可综合子集的大小不一样，具体哪些操作符可以被综合需要看综合器的参考手册。上述电路也可以用 for 循环结构描述，具体如下：

```
module comp_for_4 (a_gt_b，a_lt_b，a_eq_b，a，b);
output a_gt_b，a_lt_b，a_eq_b；
input [3:0] a，b;
parameter size = 4;
```

```
reg a_gt_b，a_lt_b，a_eq_b;
integer k;
always @(a or b) begin : compare_loop
   for (k=size-1; k>=0 ; k = k-1) begin
    if (a[k] !=b[k]) begin
    a_gt_b = a[k];
    a_lt_b = ~a[k];
    a_eq_b = 1'b0;
    disable compare_loop;
    end
      a_gt_b = 1'b0;
      a_lt_b = 1'b0;
      a_eq_b = 1'b1;
 end
endmodule
```

如果 a、b 对应位置上的值相同，那么 a＝b，否则就用 a[k]的值决定 a_gt_b 和 a_lt_b 的值。描述中 disable 的目的是为了在 a≠b 时，强制进入下一次的循环。有些综合器可以支持有固定循环次数的循环结构。

2) 简单组合电路设计举例

在数字电路中，组合电路的一般手工设计步骤如下：

(1) 根据实际问题，定义输入逻辑变量和输出逻辑变量，列出真值表。

(2) 根据真值表，写出布尔表达式，借助于卡诺图简化布尔表达式，得到最后的与或表达式。

(3) 根据与或表达式，画出电路图。

(4) 最后通过实验验证设计。

在用 Verilog 语言描述一个组合电路时，只需要根据真值表写出相应的布尔表达式，逻辑综合工具就可以将 Verilog 语言描述的设计变换成电路，自动地优化电路，并以门级网表形式存储。设计人员可以根据软件仿真的方法确认设计结果是否正确，最后将设计下载到 FPGA 器件中，通过 FPGA 器件和其他器件构成的实际系统确认设计是否正确。

**【例 3.7】** 用 Verilog 描述一个半加器(见图 3.8)。该半加器有两个输入 a、b (加数和被加数)和两个输出 c、s (进位与加法和)，其真值表如表 3.1 所示。

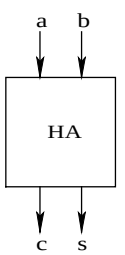



图 3.8 半加器符号

**表 3.1 半加器真值表**

| a | b | c | s |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

根据真值表，我们可以编写如下的 Verilog 代码。对应于这个真值表可以有几种不同的 Verilog 写法。

实现 1：

```
module   HA(a，b，c，s)
input a，b;
output c，s;
reg c，s;
always @(a or b)
  begin
     c = 0;
     case ({a，b}) begin
     2'b 00 : s = 0;
     2'b 01 : s = 1;
     2'b 10 : s = 1;
     2'b 11 : begin s =0 ;
                    c = 1;
     end
   endcase
end
endmodule
```

实现 2：

```
module   HA_1(a，b，c，s)
    input a，b;
    output c，s;
    wire c，s;
    assign c = a & b;
    assign s = a ^ b;
endmodule
```

实现 3：

```
module   HA_2 (a，b，c，s);
    input a，b;
    output c，s;
    wire c，s;
    wire [1:0] adder;
    assign adder = a + b;
    assign s =adder[0];
    assign c = adder[1];
  endmodule
```

【例 3.8】　用 Verilog 语言描述一个全加器。全加器有三个输入 a、b、carry_in，其中 a 和 b 是加数，而 carry_in 是从低位来的进位，两个输出 sum 和 carry_out。其真值表如表 3.2 所示。

**表 3.2　全加器真值表**

| a | b | carry_in | carry_out | sum |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 |

用 Verilog 描述全加器的方式有多种。

(1) 根据全加器的结构描述，一个全加器由两个半加器构成。如图 3.9 所示。

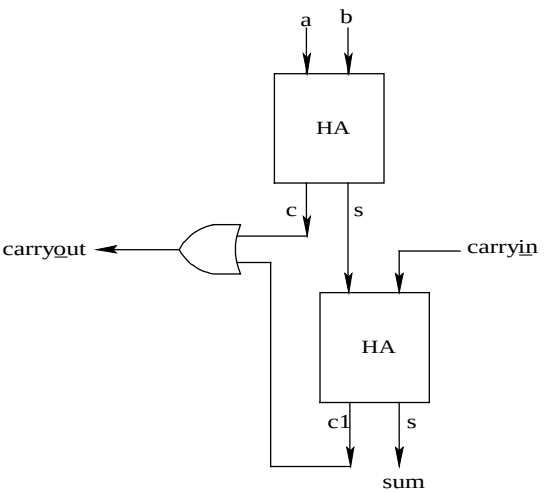



图 3.9　由半加器构成的全加器

```
module   FA_1(a，b，carry_in，sum，carry_out)    //结构描述
    input a，b，carry_in;
    output sum，carry_out;
    wire c，s，c1;
    HA HA_1 (.a(a)，
             .b(b)，
             .c(c)，
             .s(s));
```

```
    HA HA_2(.a(s)，
            . b (carry_in)，
            .c(c1)，
            .s(sum));
  assign carry_out = c ^c1;
endmodule
```

(2) 根据全加器功能描述。

```
module   FA_2(a，b，carry_in，sum，carry_out) ;      //功能描述
input a，b，carry_in;
output sum，carry_out;
wire [1:0] full_adder;                                  //2 bit 变量，高位为进位，低位为和
assign full_adder = a + b + carry_in;
assign sum = full_adder[0];
assign carry_out = full_adder[1];
endmodule
```

(3) 根据真值表，化简，写出与或项，然后再用 Verilog 语言进行描述。

```
module   FA_3 (a，b，carry_in，sum，carry_out) ;     //手工设计
input a，b，carry_in;
output sum，carry_out;
assign   sum        = a&b&carry_in | a&~b&~carry_in | ~a & ~b & carry_in | ~a & b & carry_in;
assign   carry_out = a & b &carry_in | a&b & ~carry_in | b & carry_in & ~a | a & ~b & carry_in;
endmodule
```

(4) 直接根据真值表描述。

```
module   FA_4(a，b，carry_in，sum，carry_out);       //真值表描述
input a，b，carry_in;
output sum，carry_out;
reg sum;
reg carry_out;

always @(a or b or carry_in)
  case ({a，b，carry_in})
    3'b 000 : begin
                sum = 1'b0;
                carry_out = 1'b0;
              end
    3'b 001 : begin
                sum = 1'b1;
                carry_out = 1'b0;
                end
```

```
        3'b 010 : begin
                  sum = 1'b1;
                  carry_out = 1'b0;
                  end
        3'b 011 : begin
                  sum = 1'b0;
                  carry_out = 1'b1;
                end
        3'b 100 : begin
                  sum = 1'b1;
                  carry_out = 1'b0;
                  end
        3'b 101 : begin
                  sum = 1'b0;
                  carry_out = 1'b1;
                  end
        3'b 110 : begin
                  sum = 1'b0;
                  carry_out = 1'b1;
                  end
        default: begin
                  sum = 1'b1;
                  carry_out = 1'b1;
                  end
        endcase
    endmodule
```

若干位的二进制加法器可以用多个全加器构成，即所谓的行波加法器，如图 3.10 所示。读者可以试着自己写出多位二进制加法器。

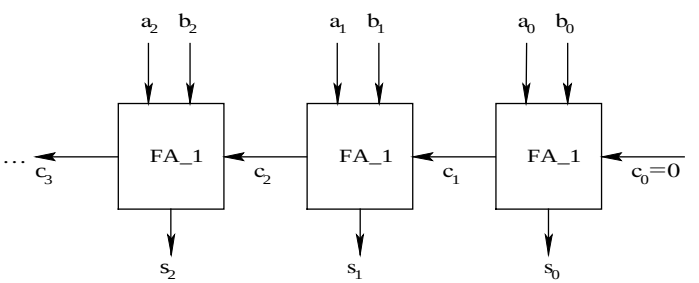



图 3.10　多位二进制加法器的实现

从上述例子看出，Verilog 语言只是一个描述工具，可以用多种不同的风格描述一个设计。如在全加器的设计中，采用了结构描述 FA_1，行为描述 FA_2，真值表的描述 FA_3 和 FA_4，其中 FA_1 是人为地写出了与或式形式，但是没有简化，而 FA_3 就是真值表的直接翻译，这些写法都可以综合出实现全加器功能的电路。

事实上，经过训练后，按照某种风格，就可写出可综合的 Verilog 代码，这些代码可以被综合器变换成合理的电路设计。设计的关键还是在于对问题的理解和建模上。

**【例 3.9】** 设计一个带使能端的 3-8 译码器。

译码器是最常见的组合电路之一。3 位二进制信号可以译出八个不同的信号。所谓的带使能端就是当一个使能信号位有效时，译码器工作；信号位无效时，译码器不工作。定义使能信号名称 decode_en 高电平有效，3 位二进制信号用一组向量 binary_in[2:0]表示，8 位输出信号用 decoder_out[7:0]表示。译码器的功能定义如表 3.3 所示。

**表 3.3 译码器真值表**

| decoder_en | binary_in[2:0] | decoder_out[7:0] |
|---|---|---|
| 0 | ××× | 0000_0000 |
| 1 | 000 | 0000_0001 |
| 1 | 001 | 0000_0010 |
| 1 | 010 | 0000_0100 |
| 1 | 011 | 0000_1000 |
| 1 | 100 | 0001_0000 |
| 1 | 101 | 0010_0000 |
| 1 | 110 | 0100_0000 |
| 1 | 111 | 1000_0000 |

下面我们用不同的编码风格对其进行描述。

```
module decoder_using_case (binary_in,      //3 bit 输入
                           enable,         //译码使能
                           decoder_out,    //8 bit 输出
                           enable;         //译码使能

input [2:0] binary_in ;
input enable ;
output [7:0] decoder_out ;
reg [7:0] decoder_out ;
always @ (enable or binary_in)
begin
   decoder_out = 0;
   if (enable) begin
```

```
        case (binary_in)
            3'h0 : decoder_out = 8'h01;
            3'h1 : decoder_out = 8'h02;
            3'h2 : decoder_out = 8'h04;
            3'h3 : decoder_out = 8'h08;
            3'h4 : decoder_out = 8'h10;
            3'h5 : decoder_out = 8'h20;
            3'h6 : decoder_out = 8'h40;
            3'h7 : decoder_out = 8'h80;
        endcase
    end
end
endmodule

//使用 Assign 语句描述
module decoder_using_assign (
binary_in，        //3 bit 输入
decoder_out，      //8 bit 输出
enable             //译码使能
);
input [2:0] binary_in;
input    enable ;
output [7:0] decoder_out ;

wire [7:0] decoder_out ;
assign decoder_out = (enable) ? (1 << binary_in) : 8'b0 ;  //左移 7 位
endmodule

//使用 for 语句描述
module decoder_using_for (
binary_in，        //4 bit 输入
decoder_out，      //16 bit 输出
enable );          //译码使能
input [2:0] binary_in;
input    enable ;
output [7:0] decoder_out ;

reg [7:0] decoder_out ;
```

```
integer i ;

always @ (enable or binary_in)
begin
   decoder_out = 0;
   if (enable) begin
      for (i = 0; i < 8; i = i +1 ) begin
         decoder_out = (i == binary_in) ? i + 1 : 8'h0;
      end
   end
end
endmodule
```

【例 3.10】　多路选择器的设计。

多路选择器可以通过 CASE 结构与 IF-THEN-ELSE 结构实现。这两种结构都能实现组合电路，但是综合器在综合时有着不同的处理。IF-THEN-ELSE 结构具有优先级特征，而 case 结构描述的电路不具有优先级的特征。

```
module single_if(a，b，c，d，sel，z);
input a，b，c，d;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or sel)
begin
      if (sel[3])
            z = d
      else if (sel[2])
            z   = c;
      else if (sel[1])
            z   = b;
      else if(sel[0])
            z   = a;
      else
            z =0 ;
end
endmodule
```

这种描述方式综合出来的电路具有优先级，综合器会综合出具有优先级的电路，图 3.11 为 Synopsys 综合器综合的结果。sel[3]和 d 具有最小的延时。

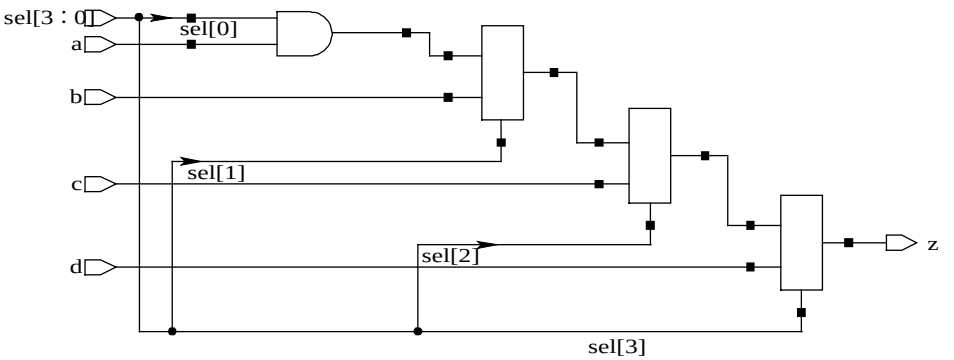



图 3.11　IF-THEN-ELSE 结构的多路选择器

使用 case 语句设计不具有优先级的多路选择器，图 3.12 中 Synopsys 综合器综合用 case 结构描述多路选择器的结果。

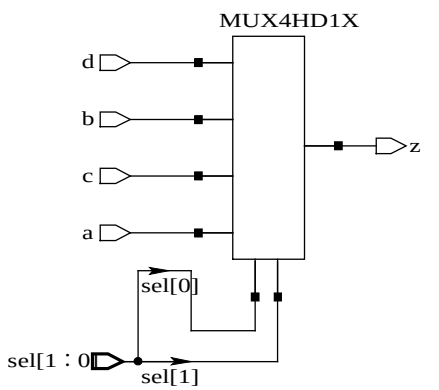



图 3.12　case 结构的多路选择器

```
module case1(a，b，c，d，sel，z);
input a，b，c，d;
input [1:0] sel;
output z;
reg z;
always @(a or b or c or d or sel) begin
casex (sel)
    2'b00: z = d;
    2'b01: z = c;
    2'b10: z = b;
    2'b11: z = a;
    default: z = 1'b0;
endcase
end
endmodule
```

【例 3.11】　设计一个 8–3 的编码器。由于篇幅的关系，省略了编码器的真值表。

```
module encoder(data，code);
input [7:0] data;
output [2:0] code;
always   @(data)
    case (data)
      8'h 01 : code = 0;
      8'h 02: code = 1;
      8'h 04 : code = 2;
```

```
            8'h 08 : code = 3;
            8'h 10 : code = 4;
            8'h 20 : code = 5;
            8'h 40 : code = 6;
            8'h 80 : code = 7;
            default : code = 3'bx;
        endcase
endmodule
```

3) 组合电路设计中应注意的问题

(1) 避免组合逻辑反馈。初学者容易在设计组合电路时，产生组合环，在电路中使用组合环会引起包含时序分析不正确等众多问题，应该避免使用。有组合环的结构如图 3.13(a)所示，没有组合环的结构如图 3.13(b)所示。

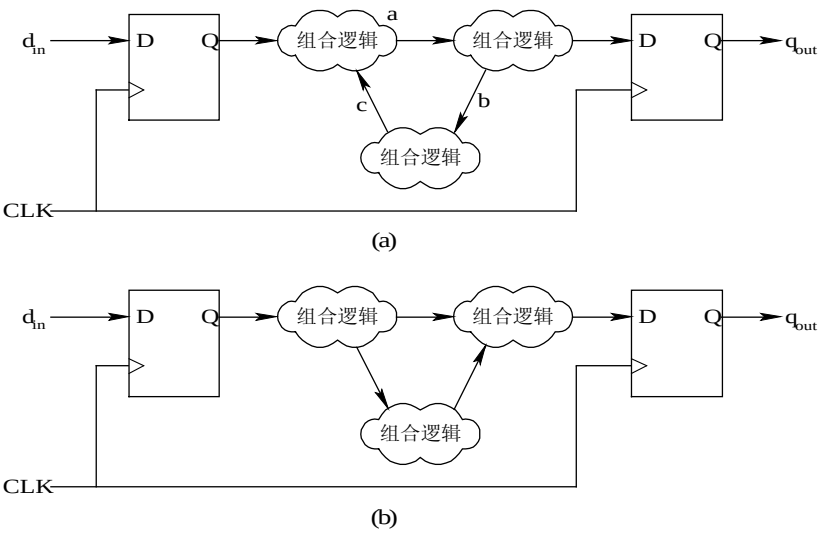



图 3.13　有组合环与无组合环的逻辑结构

(a) 有组合环结构；(b) 无组合环结构

在图 3.13(a)中，如果信号 a 是通过信号 c 和其他信号组合产生的，而信号 b 是通过信号 a 和其他信号组合产生的，信号 c 是通过信号 b 和其他信号组合产生的，那么就会产生组合环。

(2) 在敏感变量表中列出所有的敏感信号。组合电路设计中，在每个 always 中应该给出完整的敏感信号列表。如果没有完整的敏感信号列表，综合前的设计与综合后网表之间的仿真结果就会有差异。对于组合逻辑模块(不包含寄存器或锁存器)而言，敏感信号列表应该包含那些在进程中读取的信号，也就是说，出现在赋值运算符号右边以及出现在条件表达式中的信号都应当在敏感信号列表中出现。

**【例 3.12】**　本例给出了敏感列表不全的代码，综合前后的仿真结果如图 3.14 所示。

```
always@ (a)
    c <= a or b;
```

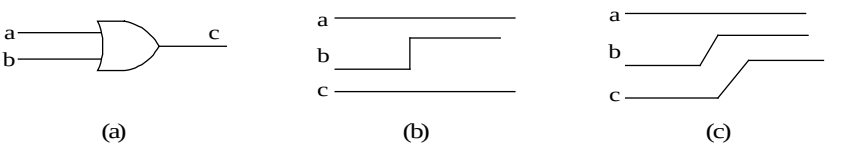



图 3.14　敏感变量不全电路的综合前后仿真

(a) 电路设计；(b) 综合前仿真结果；(c) 综合后仿真结果

(3) 避免设计锁存器。在组合电路设计中，由于疏忽，非常容易写出锁存器。锁存器一般会导致特殊的时序关系，在设计中应该避免使用锁存器。

【例 3.13】　下面的设计将产生锁存器，综合结果如图 3.15 所示。

```
module mux_latch(sel_a，sel_b，data_a，data_b ，y_out);
input sel_a;
input sel_b;
input data_a;
input data_b;
output y_out;
always @(sel_a or
                sel_b or b
                data_a or
                data_b)
    case({sel_a，sel_b})
      2'b 10 : y_out = data_a;
      2'b 01 : y_out = data_b;
    endcase
endmodule
```

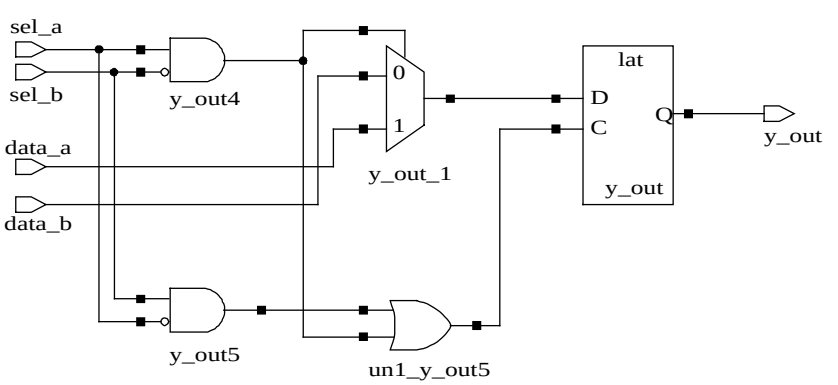



图 3.15　带锁存器的多路选择器

为避免编码中隐含锁存器，如果使用 case 语句，则在列举完所有的情况后，增加 default

语句即可避免锁存器的产生，如图 3.16 所示。如果使用 if 语句，则在最后分支中应该使用 else 语句。

```
module mux_latch(sel_a，sel_b，data_a，data_b ，y_out);
input sel_a;
input sel_b;
input data_a;
input data_b;
output y_out;
    always @(sel_a or
                sel_b or
                data_a or
                data_b)
     case({sel_a，sel_b})
       2'b 10 : y_out = data_a;
       2'b 01 : y_out = data_b;
       default : y_out = 2'b 00;
    endcase
endmodule
```

利用 if-then-else 设计组合电路，代码如下：

```
combinational_proc : always @ (decode or a or b)
 begin
    if (decode == 1'b0)
        c = a;
    else
        c = b;
 end
```

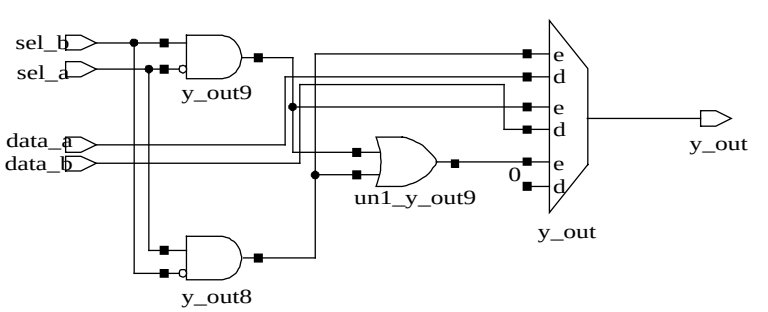



图 3.16　没有锁存的多路选择器

### 3. 三态和总线接口的设计

在很多设计中，会涉及到三态总线的接口，典型的应用如 CPU 接口。三态的设计使用连续赋值语句“=”，其中一个分支被赋值成高阻 z 状态。

1) 单向三态总线

```
moduleuni_dir_bus(core_logic_to_bus，bus_enable，data_to_bus);
input bus_enable;
input [3:0] core_logic_to_bus;
output [3:0] data_to_bus;
wire [3:0] core_logic_to_bus;
assign data_to_bus = (bus_enable) ? core_logic_to_bus : 8'hz;
endmodule
```

2) 双向三态总线

下面是一个典型的微处理器接口的例子，地址总线为 2 位，数据总线为 8 位。

```
module bi_dir_bus(rst_n，
                  wr_clk，
                  wr_en，
                  rd_en，
                  data_to_from_bus );
input rst_n;                        //复位信号
input wr_clk;                       //写数据时钟
input wr_en;                        //写使能
input rd_en；                       //读使能
input [1:0] addr;                   //地址
inout [7:0] data_to_from_bus;       //双向数据总线

reg [7:0] reg1，reg2，reg3，reg4;
wire [7:0]core_logic;
assign data_to_from_bus     = (rd_en) ? core_logic: 8'hz;

always @(negedge rst_n or posedge wr_clk)
  if (!rst_n)
     reg1 <= 8'h 00;
  else if (wr_en & addr == 2'b 00)
     reg1 <= data_to_from_bus;

always @(negedge rst_n or posedge wr_clk)
  if (!rst_n)
     reg2 <= 8'h 00;
  else if (wr_en & addr == 2'b 01)
    reg2 <= data_to_from_bus;

always @(negedge rst_n or posedge wr_clk)
```

```
  if (!rst_n)
      reg3 <= 8'h 00;
  else if (wr_en & addr == 2'b 10)
     reg3 <= data_to_from_bus;

always @(negedge rst_n or posedge wr_clk)
  if (!rst_n)
      reg4 <= 8'h 00;
  else if (wr_en & addr == 2'b 11)
     reg4 <= data_to_from_bus;

assign core_logic =    (addr == 2'b 00) ? reg1 :
                       (addr == 2'b 01) ? reg2:
                       (addr == 2'b 10) ? reg3 : reg4 ;
endmodule
```

### 3.2.3 时序电路的设计

#### 1. 时序电路的基本概念

1) 时序电路的模型

与组合电路不同，时序电路的输出不仅与当前的输入有关，而且与电路过去的历史状态也有关系。一般的时序电路模型见图 3.17，一个时序电路由存储元件和组合电路组成，其中($x_1$，…，$x_n$)是输入信号，($z_1$，…，$z_m$)是输出信号，($y_1$，…，$y_n$)表示当前状态，($Y_1$，…，$Y_n$)表示下一个状态，输出变量 $z_i$ 和下一个状态变量 $Y_i$ 是($x_1$，…，$x_n$)和当前状态($y_1$，…，$y_n$)的函数。在 FPGA 中，一般使用锁存器和 D 触发器作为存储元件，数字电路中的其他存储元件(T 触发器和 JK 触发器)可以从 D 触发器中构造出来。D 触发器是边沿触发的，它在有效沿(上升沿或下降沿)到来时，将输入端的数据锁存。锁存器是电平敏感的存储元件，时钟为高电平时，输入透明传输到输出，在时钟为低电平时，输出数据维持上一次高电平的输入数据不变。

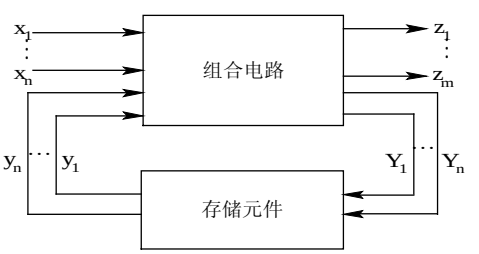



图 3.17　时序电路

2) 触发器的建立和保持时间

建立时间(Setup Time)是指在触发器的时钟信号有效沿到来以前，数据稳定不变的时间，如果建立时间不够，在这个时钟沿锁入触发器的数据可能不正确；保持时间(Hold Time)是

指在触发器的时钟信号有效沿到来以后，数据保持稳定不变的时间，如果保持时间不够，在这个时钟沿锁入触发器的数据同样可能不正确。数据稳定传输必须满足建立和保持时间的要求，在 FPGA 器件手册中对建立和保持时间都有规定。建立时间和保持时间如图 3.18 所示。

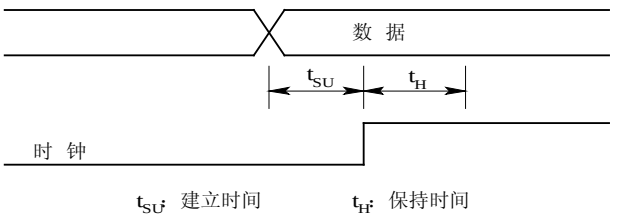



图 3.18　建立时间和保持时间

3) 同步电路和异步电路

时序电路可以分成“同步”和“异步”两种。一个时序电路，如果其中的所有 D 触发器的时钟端(锁存器除外)都与同一个时钟相连接，则称为同步时序电路；否则，则称为异步时序电路。

在 FPGA 设计中，最重要的一个概念就是同步设计。虽然，异步电路在功耗、面积上都比较有优势，但是，一般而言，异步电路的设计难度比较大，时序也难以控制。每修改一次设计，都要花费很长的时间去调整电路的时序，每次布局布线的结果都会对电路时序造成比较大的影响。甚至温度、电压或者加工器件的方法发生一些变化，都会导致异步电路中信号的时序发生变化，造成电路设计不可靠。

同步电路设计有一套完整的设计方法，设计相对简单，设计出的电路可靠性高。另外，目前大部分 EDA 工具都是针对同步电路开发的，因此，在 FPGA 设计中，提倡同步电路设计。一个系统最好只有一个时钟，但是事实上，有些系统不可避免地要用到多个时钟源，在这种情况下，需要对两个不同时钟域的信号进行同步。在 FPGA 设计中，如果要使用多个时钟，首先要考虑所选择的 FPGA 中能提供的全局时钟资源，根据这个资源和电路设计的需要，决定哪些信号使用全局时钟，使用几个时钟。因此，时钟的使用没有一个统一的标准，要取决于设计者对系统所做的分析，取得一个合理的平衡。虽然如此，我们还是提倡尽量少用多个时钟，采用同步设计。

4) 时钟树

为了保证 FPGA 能可靠的工作，一般设计方法是同步设计。在同步设计中要求时钟信号必须在同一时间到达电路中每个寄存器的时钟输入端，而且时钟信号经过输入管脚到达触发器的路径具有很小的延时。在 FPGA 中，给专用的 I/O 模块配置了速度非常快的时钟驱动缓存器，这些缓存器驱动输入时钟信号到芯片内部的时钟树上。之所以用时钟树是因为其结构像一个树，而且它的每个分支都驱动固定数目的触发器的时钟输入端。设计这种树型结构的目的是为了把各个时钟信号到达整个芯片各个触发器的延时差异减少到最小，时钟树的每个分支都具有相同的长度。图 3.19 给出了时钟树的示意图，其中，(a)表示时钟分布网络的结构，而(b)表示对应的时钟树。

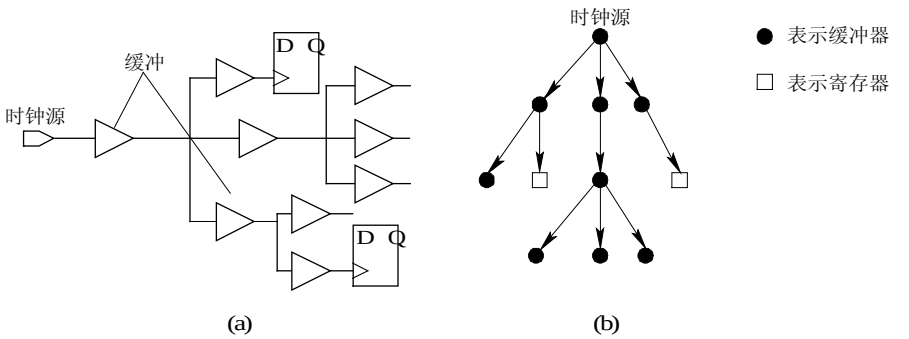



图 3.19　时钟树示意图

(a) 时钟分布网络的结构；(b) 时钟树

5) 时钟类型

时钟的类型有四种：全局时钟、门控时钟、多级逻辑时钟和行波时钟。

(1) 全局时钟：正如前面介绍的在任意一个厂家提供的 FPGA 中，都有专门的全局时钟资源，这些全局资源有专用的全局时钟管脚，可直接连接到器件中的每个寄存器的时钟端(见图 3.20)，提供时钟到输出的最短延时和最小的时钟歪斜。这就意味着时钟到每个 D 触发器的时间基本相同，从而一个 D 触发器的输出能被下一级 D 触发器正确采样。

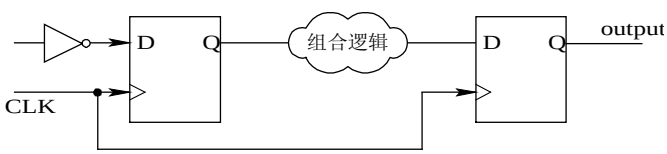



图 3.20　全局时钟示例

(2) 门控时钟：所谓的门控时钟(见图 3.21)就是由逻辑门和时钟进行逻辑操作后产生的时钟。设计不当的门控时钟往往容易产生毛刺，从而影响电路的可靠性。即使产生的时钟没有毛刺，如果门控时钟不进入时钟网络，则时钟到达 D 触发器的输入端的延时也可能会较大，由于布局布线的原因，使用门控时钟存储元件可能不能正确地锁存数据。

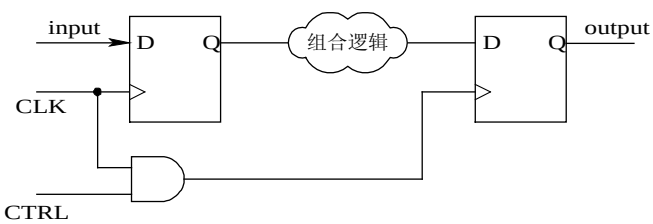



图 3.21　门控时钟电路

(3) 行波时钟：用一个 D 触发器的输出作另一个触发器的时钟输入是数字电路设计中经常用到的一种设计方案。行波时钟不产生任何毛刺，可以跟全局时钟一样地可靠工作。然而，行波时钟使得与电路有关的时序计算变得很复杂。行波时钟到行波链上各触发器的时钟之间可能产生较大的时间偏移，并且会超出最坏情况下的建立时间、保持时间和电路

中时钟到输出的延时，使系统的工作不可靠。图 3.22 是用 CLK 二分频后的时钟做下一级 D 触发器的时钟。

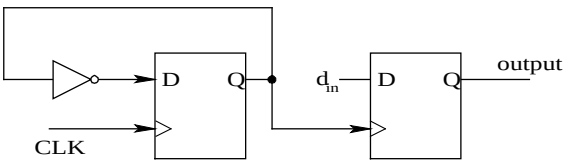



图 3.22　行波时钟

(4) 多级逻辑时钟：当产生门控时钟的组合逻辑超过一级(即超过单个的“与”门或“或”门)时，电路的可靠性变得很难控制。即使仿真结果没有显示出冒险竞争的现象，但实际上仍然可能存在着危险。

6) 时钟策略

同步设计对 FPGA 来讲非常重要，这种设计方法可以保证时序的正确性，减少调试电路时间。因此对于时钟的应用要非常小心，下面给出多时钟设计电路中的一些时钟使用策略。

(1) 如果在电路中一定要用到门控时钟或行波时钟，那么应当在顶层的时钟模块完成时钟的分频或产生门控时钟，时钟的反相也在顶层模块完成。

(2) 如果时钟需要倍频，使用 FPGA 中的锁相环 PLL 实现。

(3) 在顶层的时钟模块完成时钟元件(分频器、PLL)的实例化。

(4) 时钟模块的输出端口应该直接连接到核心模块存储元件的输入端，在时钟模块外，不应再有其他地方产生时钟。

(5) 不要用多级逻辑产生的时钟，这样的时钟容易有毛刺，导致存储元件不能正确保存数据。

(6) 有的 FPGA 可以根据时钟驱动电路的大小，自动将时钟驱动到全局时钟资源网络上，而有的 FPGA 则需要通过特殊的时钟缓冲元件才能驱动时钟到全局时钟资源网络上，在时钟模块完成时钟缓冲元件的实例化。

**【例 3.14】** 图 3.23 给出某设计中的时钟模块的一种结构。在本例中，一共需要四个时钟 m1_clk，m2_clk，m3_clk 和 m4_clk，这四个时钟分别是信号 CLK1，CLK2 和 CLK3 经过分频或者直接得到的。这四个时钟应该分布到全局时钟资源网络上，使得这四个时钟所管辖的设计区域的 D 触发器的时钟具有较好的一致性。

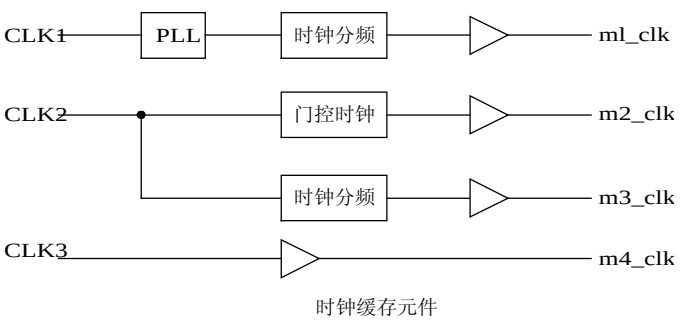



图 3.23　时钟模块的结构

## 2. 时序电路的建模

1) 基本 D 触发器的建模风格

(1) 有异步复位的 D 触发器，如图 3.24 所示，其代码如下：

```
module Dflip(rst_n，
             clk，
             din，
             dout)；
input rst_n;
input clk;
input   din;
output dout;
reg   dout;

always @(negedge rst_n or posedge clk)
    if ( !rst_n)
        dout <= 1'b0;
    else
        dout <= din;
endmodule
```

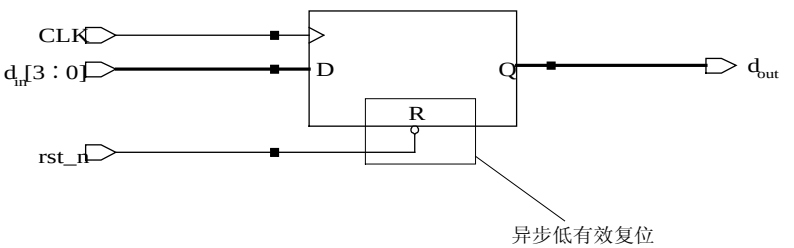



图 3.24　异步复位 D 触发器

上述代码是一个简单的具有异步复位的 D 触发器的模型，综合结果如图 3.24 所示。注意其 always 中复位信号和时钟信号的写法，我们看到，rst_n 信号出现在敏感变量表中，并出现在第一个 if 语句中，而 CLK 虽然也出现在敏感变量表中，但是在整个分支语句中，对 CLK 没有任何的编码信息。从这种写法中，综合器可以推测出 D 触发器的结构。rst_n 表示一个低有效的复位电路。无论什么时候，只要 rst_n 为低电平，且低电平具有一定的宽度时，D 触发器的输出端变成 0。在 rst_n 为高电平，且时钟的上升沿到来时，$d_{out}$ 由 $d_{in}$ 更新。

(2) 有同步复位功能的 D 触发器，如图 3.25 所示，其代码如下：

```
module Dflip(rst_n，
             clk，
             din，
             dout)
input rst_n;
```

```
input clk;
input din;
output dout;
reg dout;
always @( posedge clk )
    if (!rst_n)
        dout <=1'b0;
    else
        dout <= din;
endmodule
```

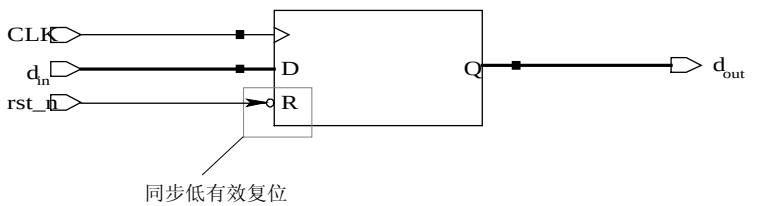



图 3.25　同步复位 D 触发器

上面是一个简单的具有同步复位的 D 触发器的模型，综合结果如图 3.25 所示。注意其 always 中已经没有复位信号 rst_n，同样在本例中没有对 CLK 有任何编码。在上述描述的电路中，rst_n 的优先级高于 $d_{in}$。如果在时钟 CLK 的上升沿到来时，rst_n 为低电平，$d_{out}$ 变成低电平。所谓的同步复位是指复位信号只有在时钟有效沿到来时且复位信号有效时，才完成复位。

(3) 有使能端的 D 触发器。

```
module dflip(rst_n，
             clk，
             din_en，
             din，
             dout)
input rst_n;
input clk;
input din;
input din_en;
output dout;
reg dout;
always @(negedge rst_n or posedge clk)
    if (!rst_n)
        dout <= 1'b0;
    else if (din_en)
       dout <= din;
endmodule
```

本段描述的是一个简单的具有异步复位和使能端的 D 触发器的模型。在 rst_n 为高电平且使能端 din_en 为高电平时，在时钟的上升沿，$d_{out}$ 由 $d_{in}$ 更新。

2) 状态机建模

有限状态机是建立系统模型最为有效的手段，有着广泛的应用。综合工具可以非常有效地将 HDL 语言描述的状态机行为转换成门级电路。下面首先介绍一下状态机的基本概念和设计方法。

(1) 有限状态机的基本概念。为了描述不同的系统和行为，有限状态机有非常多的变形，研究人员作了大量的理论研究。在数字电路设计中，我们一般采用被称为 Mealy 机和 Moore 机(见图 3.26)这两类状态机描述电路的行为。Mealy 机和 Moore 机都由三个部分构成：存储当前状态的寄存器(存储元件)，决定下一个状态的组合电路和输出组合电路。不同的是，Mealy 机的输出不但与当前状态有关，还与输入有关，而 Moore 机的输出只与当前状态有关。Mealy 机和 Moore 机可以用下面两个抽象结构表示，如图 3.26 所示。

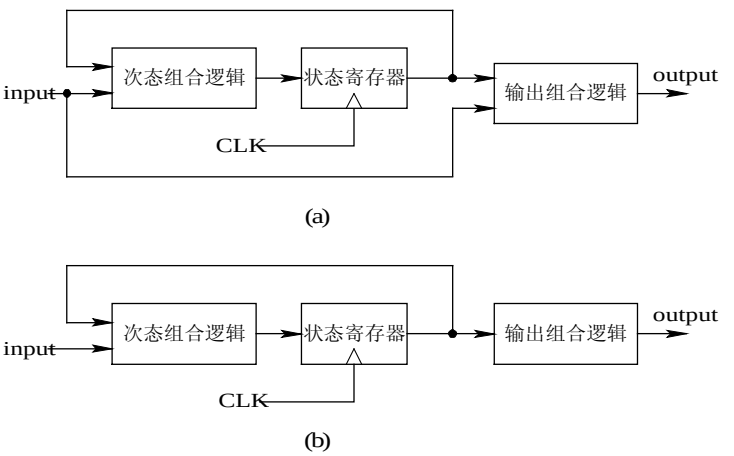



图 3.26　两种有限状态机模型

(a) Mealy 机；(b) Moore 机

Moore 机和 Mealy 机的一般手工设计方法步骤如下:

① 根据功能要求，确定电路的状态数目。

② 定义状态转移表。

③ 选择状态赋值。

④ 编码状态和输出表。

⑤ 状态化简和输出。

⑥ 根据状态转移图完成电路的实现。

(2) 用 Verilog 语言描述有限状态机。与手工设计不同的是，状态的化简和设计实现是由综合器自动实现的。根据图 3.26 所示的 Mealy 机和 Moore 机的抽象结构，在用 Verilog 描述状态机时，将状态机的描述划分成以下两个进程：

① 一个进程用电平敏感的组合逻辑描述次态和输出。

② 一个进程用边沿敏感的行为描述同步更新的时序逻辑(状态)。

【例 3.15】　设计 BCD 码到余 3 码的转换电路，BCD 和余 3 码之间的关系如表 3.4 所示。

表 3.4　BCD 到余 3 码的转换

| 序　列 | BCD | 余 3 码 |
|---|---|---|
| 0 | 0000 | 0011 |
| 1 | 0001 | 0100 |
| 2 | 0010 | 0101 |
| 3 | 0011 | 0110 |
| 4 | 0100 | 0111 |
| 5 | 0101 | 1000 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1010 |
| 8 | 1000 | 1011 |
| 9 | 1001 | 1100 |

BCD 码和余 3 码之间有如下关系：

① BCD 码加 3 即为余 3 码。

② 余 3 码是一个自补码，即对于任意一个余 3 码 a，存在另外一个 a'，使得 a＋a'＝9。

假设码流以串行流的形式进入转换电路，并以串行流的方式进行发送，BCD 码的低位先进入本转换电路，余 3 码的低位先发送。

设计 BCD 和余 3 码之间转换的电路有多种，这里考虑用 Mealy 机来实现。根据 BCD 和余 3 码之间的关系，可以画出的状态转移图如图 3.27 所示。

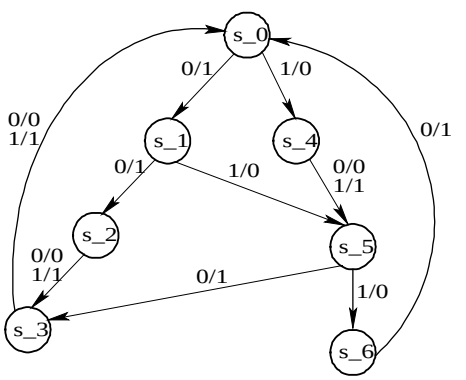



图 3.27　BCD 到余 3 码转换的状态图

```
module bcd_to_Excess_3(clk，
                       rst_n，
                       b_in，
                       b_out);
input clk;
```

```
input rst_n;
input b_in;
output b_out;

parameter s_0 = 3'b 000，
          s_1= 3''b 001，
          s_2= 3'b 010，
          s_3= 3'b 011，
          s_4 = 3'b 100，
          s_5 = 3'b 101，
          s_6 = 3'b 110;
reg [2:0] curr_st，next_st;
reg b_out;


//过程 1：用组合逻辑计算次态
always @(curr_st or b_in)
begin
 b_out = 0;
case(curr_st)
 s_0 : if   (~b_in) begin
       next_st = s_1;
       b_out   = 1'b1;
    end
  else   next_st = s_4;
 s_1 : if   (~b_in) begin
          next_st = s_2;
          b_out   = 1'b1;
        end
       else   next_st = s_5;
s_2 :   begin
       next_st = s_3;
       b_out   = b_in;
       end
s_3 : begin
       next_st = s_0;
       b_out   = b_in;
       end
s_4 : begin
```

```
            next_st = s_5;
            b_out   = b_in;
          end
    s_5 : if (~b_in) begin
            next_st = s_3;
            b_out   = 1'b1;
          end
          else
            next_st = s_6;
    s_6: begin
     next_st = s_0;
     b_out = 1;
    end
    default : begin
        next_st   = s_0;
        b_out     = 0;
    end
    endcase
    end

    //过程 2：更新当前状态
    always @(negedge rst_n or posedge clk)
      if (!rst_n)
        curr_st <= s_0;
      else
        curr_st <= next_st;
    endmodule
```

在用 Verilog 建立一个状态机的模型时，需要注意:

① 用 parameter 说明符号状态名，如 s_0，s_1，用符号名定义状态使得 Verilog 代码更易读，并且在重新修改状态时变得简单；也可以用 'define 定义状态，但是 define 定义的是全局变量，而 parameter 则为局部定义，这样可以在一个设计中，定义多个名称相同的状态。例如，parameter　IDLE=3'd0，S1=3'd1，S2=3'd2，S3 = 3'd3，ERROR=3'd4。

② case 语句中对所有的状态进行编码。对 case 语句中没有列举的状态，用 default 进行说明，以便避免综合出锁存器，使电路工作不正常。一般有三种类型的缺省次态赋值：

- 次态赋值成不定态(x)，初值赋值成 x 时，如果系统状态没有列举全，那么仿真时可能出现不确定的状态。但是，综合工具将 x 视为无关项进行优化。
- 次态被设置成预定义的恢复状态，如 IDLE，这样在一些不确定的情况下，电路可自动恢复到正常工作状态。
- 次态被赋值成状态寄存器的值，即当前状态。

③ 时序部分只用组合电路计算出的新状态更新当前状态。

④ 状态机的输出可以用连续赋值，如

```
assign b_out = (curr_st == s_0) | (curr_st == s1);
```

也可以像本例一样在计算次态的 always 块中赋值。

⑤ 状态机设计时，要求用一组 D 触发器表示一组状态，并要求每个状态应该用一个二进制编码惟一的表示。状态编码直接影响 D 触发器的数目、计算次态的组合电路和输出组合电路实现的复杂度。读者可以参看数字电路设计方面的书籍了解一些基本的编码规则，这些规则有可能减少组合电路的复杂度。

⑥ 在设计电路时，必须保证表示系统状态的 D 触发器数目是足够的，表示系统状态的 D 触发器数目与表示状态的编码有关。例如，如果采用 BCD 编码，则八个状态需要用三个 D 触发器，而采用独热码则需要八个 D 触发器；有些综合器，可以自动将 BCD 编码转换成独热码。

一般常用的编码有：二进制编码(Binary Code)、格雷码(Gray Code)、独热码(One-hot Code)、约翰逊编码(Johnson Code)，表 3.5 列出了这四种编码。

**表 3.5　常见的状态编码**

| 十位数 | 二进制码 | 格雷码 | 独热码 | 约翰逊码 |
|---|---|---|---|---|
| 0 | 0000 | 0000 | 0000_0000_0000_0001 | 0000_0000 |
| 1 | 0001 | 0001 | 0000_0000_0000_0010 | 0000_0001 |
| 2 | 0010 | 0011 | 0000_0000_0000_0100 | 0000_0011 |
| 3 | 0011 | 0010 | 0000_0000_0000_1000 | 0000_0111 |
| 4 | 0100 | 0110 | 0000_0000_0001_0000 | 0000_1111 |
| 5 | 0101 | 0111 | 0000_0000_0010_0000 | 0001_1111 |
| 6 | 0110 | 0101 | 0000_0000_0100_0000 | 0011_1111 |
| 7 | 0111 | 0100 | 0000_0000_1000_0000 | 0111_1111 |
| 8 | 1000 | 1100 | 0000_0001_0000_0000 | 1111_1111 |
| 9 | 1001 | 1101 | 0000_0010_0000_0000 | 1111_1110 |
| 10 | 1010 | 1111 | 0000_0100_0000_0000 | 111_11100 |
| 11 | 1011 | 1110 | 0000_1000_0000_0000 | 111_11000 |
| 12 | 1100 | 1011 | 0001_0000_0000_0000 | 111_10000 |
| 13 | 1101 | 1010 | 0010_0000_0000_0000 | 111_00000 |
| 14 | 1110 | 1001 | 0100_0000_0000_0000 | 110_00000 |
| 15 | 1111 | 1000 | 1000_0000_0000_0000 | 100_00000 |

在这四种编码中，二进制码和格雷码所用的 D 触发器最少。与二进制码不同的是格雷码相邻的编码之间只有一位不同，在相邻状态发生转移的时候，可以减少同时翻转的 D 触发器数目，因此也可以减少电路的噪声。利用这个特性，可以简化一些电路的设计。约翰逊码也具有这样的特性，但是所用的 D 触发器更多一些。独热码是目前流行的一种编码方

式，每个状态一个 D 触发器，这种编码方式虽然需要的 D 触发器比较多，但是计算次态的组合电路却比较小，电路的速度和可靠性有比较明显的提高。

⑦ 状态机的状态数目不宜过多，否则会导致电路规模太大。

**【例 3.16】** 约翰逊计数器的设计。

约翰逊计数器是一种移位计数器，它把最高位触发器的输出取反后送入到最低位触发器。约翰逊计数器在每个时钟周期只有一位输出发生变换，这样，对约翰逊计数器输出进行译码时，不会产生毛刺。另外，约翰逊计数器中，D 触发器和 D 触发器之间没有组合电路，因此，它的速度比较快。约翰逊计数器是 Moore 类型的计数器，它需要 n 个 D 触发器表示 2n 种循环状态。

```
module johnson1(rst_n，clock，count_out);
  input rst_n，clk;
  output ［3：0］ count_out;
  reg [3:0] count_out;
always @(posedge clk or negedge rst_n)
   if (!rst_n)
     count_count <= 4'b 0000;
   else begin
     count_count [3:1] <= count_out[2:0];
     count_out[0] <= count_out[3];
   end
endmodule
```

上面的 Verilog 描述的是不具有错误恢复功能的约翰逊计数器，也就是说一旦由于干扰而使得计数器进入到无效的状态后，只有重新复位，计数器才能恢复到正常。在实际的工作中，要求电路具有一定的抗干扰能力。如果增加一些逻辑，对无效状态进行自动检测和恢复，虽然电路的复杂度增加了，但是，电路的抗干扰能力却增强了。有兴趣的读者可以写出具有错误恢复能力的约翰逊计数器。

(3) 用 Verilog 语言描述隐含状态机。有些时序电路并不是通过显式的状态机来描述的，而是通过隐含的状态机进行描述，其状态被隐含地定义在行为的描述中。与显式状态机不同的是，隐含状态机中的每个状态只能从一个状态进入，状态机的状态是由每个周期的行为所决定的。每个周期都具有相同行为的时序电路是一个状态机，它的行为可以通过电路的运行状态描述。最简单的状态机是我们前面讲过的 D 触发器，还有移位寄存器、二进制计数器等。

例如，下面的代码描述了一个模 4 的计数器。

```
module counter_4(rst_n，clk，dout);
  input rst_n;
   input clk;
   output [1:0] dout;
   reg [1:0] dout;
     always @(negedge rst_n or posedge clk)
```

```
        if (!rst_n)
            dout <= 2'b 00;
        else
            dout <= dout + 1;
    endmodule
```

(4) 时序电路设计应该注意的问题。

① 避免使用门控时钟和多级时钟，应把这些时钟转换成使能端使用。

```
clk_p1=clk & p1_gate;                //门控时钟
always@(posedge clk_p1)
    begin
      …
  end
```

可以改为：

```
always @(posedge clk)
begin
    if (p1_gate==1'b1)
      begin
      …
      end
    else
      begin
        …
      end
end
```

② 不要随意使用行波时钟。由于 FPGA 的时钟资源有限，因此不要随意使用分频时钟。在布局布线时，它们不能被布线到全局时钟资源线上，这会对电路的时序造成影响，可以用同步预置的方式实现行波时钟的功能。

例如：下面代码试图用独热码的三分频输出，作为一个模 4 计数器的时钟。

```
…
always@(posedge clk)
    3'b 000 : count <= 3'b 001;    //one-hot code;
    …
always @(posedge count [1])
    …
```

修改为：

```
always @(posedge clk)
  if (count [1])
    cnt_4 <= cnt_4 + 1;
```

③ 敏感变量：对于时序逻辑模块而言，敏感信号列表应该只包含时钟和异步复位信号(如果需要)。

④ 异步复位信号的产生：如果异步复位信号是由多个信号产生的，那么应该在复位模块，用 assign 语句产生。注意，产生的复位信号应该没有毛刺。

```
…
wire rst_n;
assign rst_n = a & !b;                    //将 rst_n 连到其他 D 触发器的复位端
```

⑤ 如果计数器进位链非常长，那么可用若干个进位链较短的计数器实现。

例如，下面的例子说明如何用三个 8 位的计数器实现一个 16 位的计数器。

```
 module cntlength_16(   rst_n，
                        clk，
                        cnt_val);
input rst_n;
input clk;
output [15:0] cnt_val;
reg [7:0] cnt_1;
reg [7:0] cnt_2;
reg carry;
   always @(negedge rst_n or posedge clk)
      if (~rst_n)
        cnt_1 <= 8'd0;
      else
        cnt_1 <= cnt_1 + 1;

   always (@(negedge rst_n or posedge clk)
      if (~rst_n)
         carry <= 1'b0;
      else
         carry <= cnt_1 == 8'd254;

   always @(negedge rst_n or posedge clk)
      if (~rst_n)
        cnt_2 <= 8'd0;
      else if (carry)              //当低 8 位计数器有进位时，高位计数器加 1
        cnt_2 <= cnt_2 + 1;

   assign cnt_val = {cnt_2，cnt_1};
endmodule
```

本例是用两个 8 位计数器实现一个 16 位计数器的功能，当计数器位数太多时，组合电

路的逻辑将增大，另外会影响到电路的速度。在本例中使用了 carry 这个变量，作为 cnt1 加到 255 进位指示信号，当 carry 为高电平的时候，计数器 cnt_2 才加 1。在这个例子中，carry 是 CLK 把 cnt1 等于 254 时的信号延时一个节拍后的信号，这样第二个计数器 cnt2 的速度与第一个计数器 cnt1 的组合电路的延时没有关系，由此提高电路速度。

### 3. 亚稳态及其解决方法

什么是信号的亚稳态？亚稳态就是不稳定的状态，介于低电平 0 和高电平 1 之间，或是经过振荡到达 1 或 0 的稳态。如果时钟和 D 触发器的输入信号之间的关系是随机的，用时钟去采样 D 触发器的输入信号时，那么输入信号可能会变得与有效时钟沿之间太近，从而不能满足建立保持时间的要求，不可避免地导致输出状态的不确定。

例如，图 3.28 就是一个可能出现亚稳态的电路，q2 的输出在某些时候可能是不稳定的。

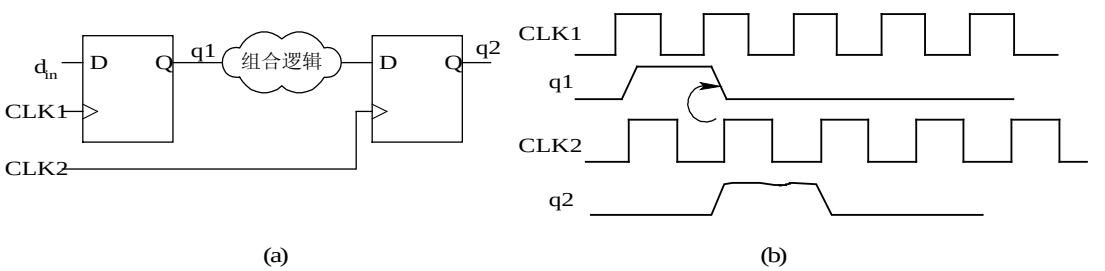



图 3.28　亚稳态电路及其波形

(a) 亚稳态电路；(b) 亚稳态电路波形

在设计中的每个触发器都有一个特定的最小建立时间和保持时间，也就是说在时钟有效沿的前后，输入数据必须保持足够的稳定时间。如果这个稳定的时间不够，那么会导致输出变成亚稳态。

异步电路的设计会导致亚稳态现象的出现，也就是说信号在不同的时钟域中传递时，会有不稳定的信号产生。那么如何消除这些亚稳定状态呢？如果一个电路中包含了多个时钟，在设计时将具有多个时钟的模块独立出来，而其他每个模块只有一个时钟。在时钟的模块中，用一个时钟同步另外一个时钟域中的信号，用这种方法可以消除亚稳态。如图 3.29 所示。

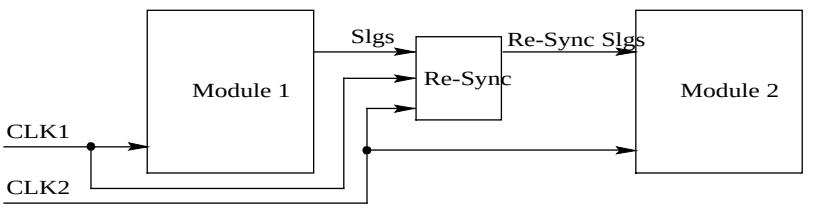



图 3.29　异步电路的划分示意

同步异步信号的电路有以下两种：

(1) 如果一个被同步信号的宽度大于同步时钟的周期，那么可以采用图 3.30 所示的同步电路。

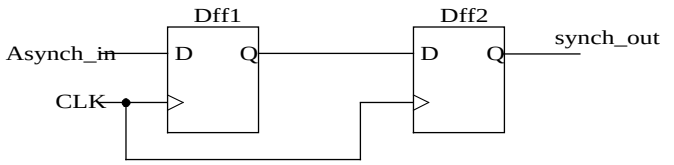



图 3.30　不同时钟域同步电路 I

让我们分析一下上述电路的工作过程：如果 Asynch_in 信号在 Dff1 的建立时间之前稳定了，那么在两个 CLK 周期后，synch_out 从 Dff2 送出，这样 synch_out 与 CLK 同步。如果异步信号在 Dff1 的建立时间之前稳定了，那么两个 CLK 周期之后，synch_out 到达稳定状态。如果 Asynch_in 信号在 Dff1 建立时间之前不稳定，假设不稳定的信号被采样为 0，但是最后到达 1，那么这个 1 将在三个周期后出现在 Dff2 的输出端 synch_out。如果信号最后稳定为 0，那么这个 0 将在两个周期后出现在输出端 synch_out。不稳定的信号只出现第一个 D 触发器，第二个 D 触发器采样到的数据是稳定的。

(2) 如果被同步的信号，脉冲宽度小于用于同步的时钟时，应该采用图 3.31 所示的电路，该电路包括了三个触发器的同步电路。在第一个 Dff0 中，$V_{CC}$ 接到数据的输入端，而用异步 Asynch_in 信号做 Dff0 的时钟，这样 Asynch_in 上的一个窄脉冲将 Dff1 驱动到 1，这个值在两个时钟脉冲后，传送到 Dff2 的输出。当 Dff2 的输出变成 1 时，接到 Dff0 异步复位端的信号 CLR 使得 Dff0 变成 0。

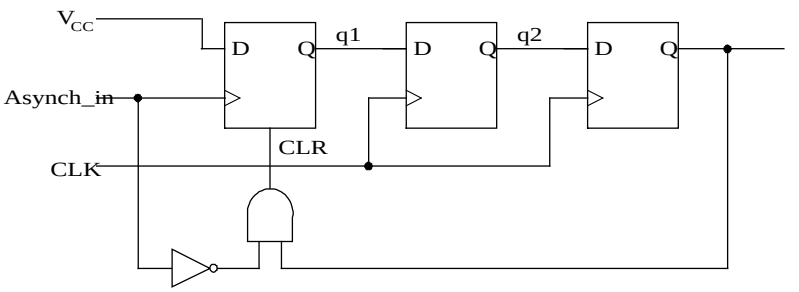



图 3.31　不同时钟域同步电路 II

(3) 同步多个信号时，最好使用异步 FIFO 结构。一个异步的 FIFO 设计可以按照图 3.32 的结构实现。由于异步 FIFO 的读/写时钟不同，因此，将读地址、写地址分别用两个模块实现，这两个模块中分别只有一个时钟。FIFO 用双端口 RAM 实现。根据读写地址判断的空满条件模块 CMP，包括了读/写两个时钟。

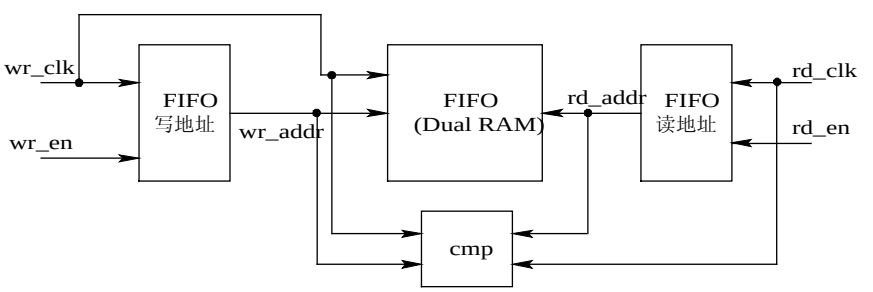



图 3.32　一种异步 FIFO 的结构

#### 4. 存储器的设计

大部分的 FPGA 中都提供了内嵌式存储器(ROM 或 RAM)，因此，存储器的设计与目标器件密切相关。在使用存储器之前，首先应该清楚你所使用的存储器的类型(双端口，还是单端口)、大小是否够用、速度是否满足设计要求、是否需要读出时钟等等。

在综合前，可以先用 Verilog 语言描述一个存储器的功能作为仿真的时候使用，等电路设计验证正确后，再用 FPGA 中嵌入存储器替换语言描述的存储器。

例如，下面是一个 128×8 的 RAM 设计，这是一个双端口的 RAM，具有读/写时钟，在读使能和写使能控制下，进行 RAM 的读/写操作。

```
module ram_128*8(wr_clk，wr_en，wr_addr，wr_dat8，rd_clk，rd_en，rd_addr，rd_dat8)
input wr_clk，wr_en，rd_clk，rd_en;
input [7:0] wr_dat8;
input [6:0] wr_addr;
output [7:0] rd_dat8;
output [6:0] rd_addr;

reg [7:0] rd_dat8;
reg [6:0] rd_addr;
reg [7:0] ram[127:0] ;

always @(posedge wr_clk)
if (wr_en)
    ram[wr_addr] <= wr_dat8 ;

always @(posedge rd_clk)
if (rd_en)
    rd_dat8 <= ram[rd_addr] ;
endmodule
```

该电路用 synplify 综合器可以得到一个标准的双口 RAM，然而使用 ISE/QUARTUS 等工具中自带的综合器却会综合成 D 触发器堆，耗费大量资源。因而在使用那些综合器时，必须将 RAM 模块注释成黑盒子(black_box)，然后在布线时用实例化的 RAM 块替代之。

## 3.3 模 块 设 计

设计描述阶段包括顶层设计和详细设计两个阶段。根据系统规范的要求，将系统划分成若干个模块，形成顶层模块图，顶层模块完成系统定义的全部功能。在顶层设计完成之后，定义各个模块的功能和接口并以原理图的形式画出各个子模块之间的连接关系。在顶层模块的基础上，将子模块进一步划分成更小的模块，重复这个过程，直到所细分的模块能实现相对单一的功能为止。设计描述阶段，应该定义每个模块的功能、本模块和其他模

块之间的接口。对于包含一些特定算法的模块，应该说明算法的原理，实现细节等。模块划分多大、如何划分取决于设计人员对所设计系统的理解和设计经验，没有一个非常严格的规则，但大体上有一些基本原则，这些原则最初是针对 IC 设计总结的，但是我们认为对 FPGA 设计也同样适合，使用这些规则可以清晰地划分电路，形成较为合理的电路结构，帮助设计者形成良好的习惯。下面我们简单地列举一些模块划分的原则：

(1) 分离特殊逻辑和核心逻辑。在芯片级应该把特殊功能的逻辑如存储器模块、I/O 模块、时钟模块和复位模块等从核心逻辑中分离出来。一种比较合理的顶层划分图如图 3.33 所示。I/O 模块包含了所有 I/O 缓存器，时钟模块包含了所有核心模块用到的时钟，并且每个时钟应该通过一定的方式进入到 FPGA 全局时钟网络中，复位模块包含了核心逻辑模块中用到的所有的复位信号，核心逻辑模块包含了系统的主要功能。

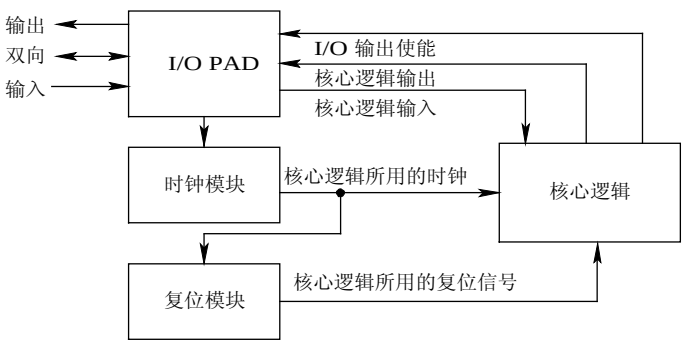



图 3.33　顶层划分

(2) 不要在模块之间使用粘合逻辑。一个设计应该只在层次结构中的最底层模块中包含门电路的实例。例如，在图 3.34(a)中的两个二级模块中，存在一个异或门，综合编译器不可能将异或门与模块 B 中的组合逻辑合并，因此限制了逻辑优化。图 3.34(b)中，将粘合逻辑合并到模块 B 的组合逻辑中。

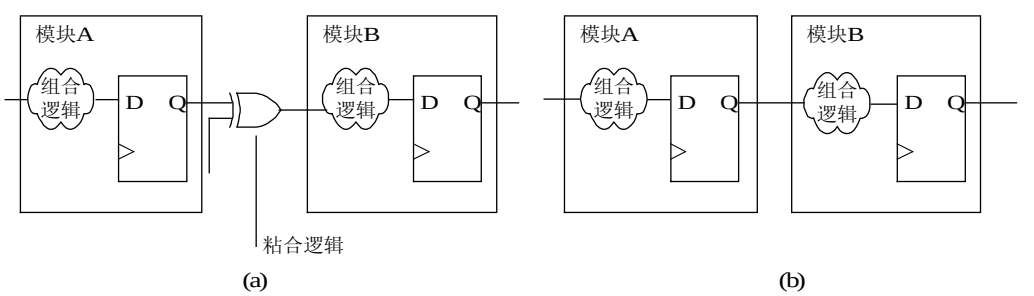



图 3.34　有粘合逻辑与无粘合逻辑的结构示意图

(a) 模块间存在粘合逻辑；(b) 模块间没有粘合逻辑

(3) 一个模块内只使用一个时钟(除了时钟处理模块外)。如果一个设计中包含了多个时钟，按时钟管辖的范围划分模块。将具有多个时钟的设计，划分成若干个模块，一个时钟管理一个模块。这样做的目的是为了便于时序分析，实施综合约束。

(4) 相关的组合逻辑应该放到同一模块中。当相关的组合逻辑被划分在一个模块内时，综合编译器可以灵活地优化这些组合逻辑。一般情况下综合编译器不能将一个模块的组合逻辑搬到另外一个模块中，除非在编译之前将不同的模块展平。图 3.35 给出了一个例子，在这个例子中，相关的逻辑被划分在三个不同的模块中，应该改成图 3.35(b)的划分才较为合理。

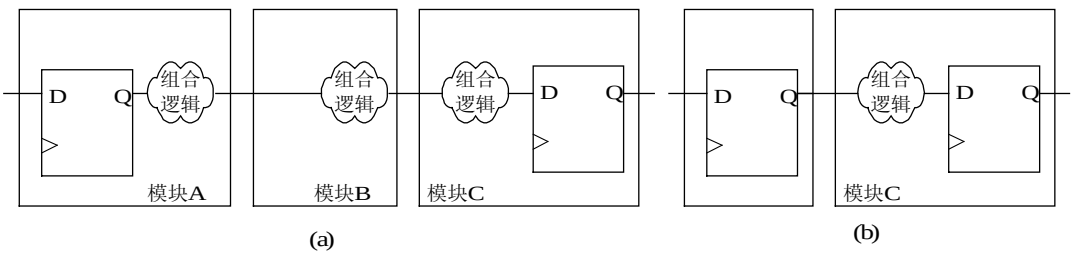



图 3.35　逻辑划分示意图

(a) 不合理划分；(b) 较合理的划分逻辑

(5) 按照不同的设计目标划分模块。它将影响到系统工作速度的关键路径模块从非关键路径的模块中分离出来，这样，编译器可以对关键路径按速度优化关键路径，而对非关键路径则按面积优化。如图 3.36 所示。

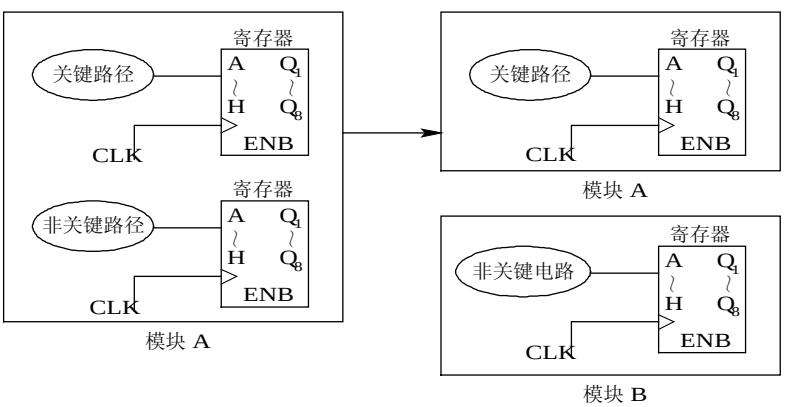



图 3.36　关键路径与非关键路径分离

(6) 寄存所有输出。每个模块经过 D 触发器寄存后再输出，如图 3.37 所示。这个原则实际上就是说寄存器和输出端口之间没有组合逻辑。

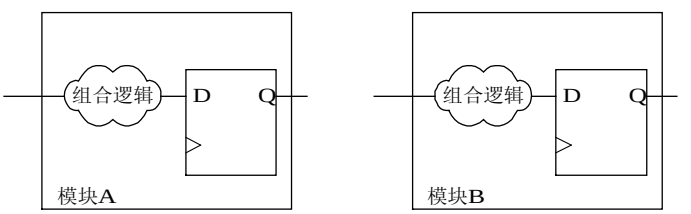



图 3.37　锁存所有输出

(7) 独立异步逻辑。在 FPGA 的设计中，应该尽力避免异步逻辑设计。如果必须使用异步逻辑，将异步逻辑放在一个独立的模块中。这样可以更方便地检查代码、功能和时序。

(8) 时序逻辑使用 D 触发器而不是锁存器。

## 3.4 系 统 规 范

### 3.4.1 系统规范的内容

为了叙述方便，让我们再回忆一下 FPGA 的设计流程：系统规范定义、模块设计、设计输入、功能仿真(前仿真)、综合、布局布线、时序仿真(后仿真)、配置下载。

FPGA 系统的设计是从系统规范定义开始的，它是非常重要的工作。如果系统的规范定义的不正确，或有二义性，那么其他的工作都是没有意义的。一个 FPGA 设计的系统规范定义至少应该包含以下的内容。

(1) FPGA 完成的功能：详细描述所设计的 FPGA 完成的功能和拟达到的性能指标，功能规范最好是使用可执行的规范语言描述，以便整个功能定义没有二义性。制定无二义性功能是整个设计的核心，是后继 FPGA 实现的依据。

(2) 所设计的 FPGA 一些典型应用：所设计的 FPGA 是如何与外部的其他器件一起构成系统的，这些典型设计可以作为使用 FPGA 的 PCB 设计人员的参考设计。

(3) I/O 管脚的描述：在管脚描述中应该包括输入/输出的驱动能力、输入/输出的阈值电平(即 TTL/CMOS/PECL/LVDS)、I/O 管脚数目、时序等。

(4) FPGA 设计规模的大小估计：虽然在设计没有完成前，无法准确给出设计的大小，但是应该有一个初步的预估，以便选择合适的 FPGA 器件。

(5) 封装形式：封装不同 FPGA 的价格也不一样，需要了解不同的 FPGA 厂家的封装，根据要求选择合适的封装。

(6) 目标功耗：提出功耗要求，以便在设计中采用合适的设计和算法使得 FPGA 功耗达到要求。

(7) 可能使用的第三方 IP 核：需要从其他公司了解 IP 核，分析它们的规范、了解价格、评估它们对本项目的影响，以决定是否采用。如果采用，是否需要作微小的修改，或以什么样的形式提交给本项目。

(8) 构想数个总体实现方案：一个项目往往可以包含多个实现方案，在系统规范阶段，可以提出多种方案，然后分析各种方案的可行性、根据资源、难易程度、软硬件等综合考虑，挑选一个合理的方案。

(9) FPGA 的验证和测试：在规范中还应该明确如何验证和测试 FPGA 功能的正确性，包括 FPGA 设计的前端仿真方案、PCB 板设计、测试程序方案和软件等。

(10) 说明关键模块：关键模块往往是一个项目能否顺利完成的核心。关键模块需要安排合适的人提前进行设计和验证，以保证它们不会影响整个项目进度。

(11) 拟选用的 FPGA 类型：综合考虑 FPGA 价格、FPGA 的保密性、设计规模、I/O 管脚的数目和使用第三方 IP 等因素，选择合适类型的 FPGA 作为系统实现的载体。

系统规范阶段除了要制定详细的系统设计规范文档外，还应该形成详细的项目规划文档，包括项目的进度、资源的需求、各个阶段所使用的工具、技术和方法等。另外，还应选择合适的人形成设计团队、制定培训计划等项目管理方面的文档。

在制定规范阶段，应该请所有相关人员对系统的规范进行评估，确定系统规范的可行性。这个评估非常重要，它是整个芯片设计的基础，不同的人员可能会从不同的角度对系统规范提出意见或指出疏漏。根据评估的意见，修改系统规范，以便设计可以进入第二阶段。

### 3.4.2 选择 FPGA

在选择 FPGA 器件时，应该考虑到以下几个问题。

(1) 可配置逻辑块：虽然大多数的 FPGA 有类似的逻辑模块，但是它们之间有一些区别。根据设计需要选择合适结构的 FPGA。

(2) 可配置逻辑块的数目：它决定了所能容纳的设计的逻辑门数。

(3) I/O 管脚的数量和类型：根据设计需要，选择合适数目的 I/O 管脚，了解有多少是通用的 I/O 管脚，有多少特殊用途的 I/O 管脚，如全局时钟输入、复位信号、下载管脚信号等。

(4) 嵌入式 IP 核：所选择的器件中是否包含了你设计中需要的 IP 核，如是否包含了锁相环核、DSP 核、SDRAM 控制器，有多大的 RAM 等。充分使用这些 IP 核可以提高设计效率。

(5) FPGA 器件的编程方式：选择反熔丝、Flash 还是 SRAM？综合评定设计使用需求，如是否需要安全、低功耗、非易失性，根据需要选择合适的 FPGA。

(6) FPGA 的工作温度：所设计的 FPGA 工作温度满足的标准是工业标准、军用标准还是商业标准。

(7) FPGA 的工作速度：每个 FPGA 厂家在同一种 FPGA 中，提供不同速度的 FPGA 型号，速度越高，价格越贵。因此，在选定了某个类型的 FPGA 后，还需要考虑同一 FPGA 中选择哪一种速度的 FPGA。

对于一个 FPGA 设计工程而言，项目管理也是非常重要的任务，良好的项目管理，可以加快整个设计的完成。有兴趣的读者可以参考一些相关的文献。

# 第 4 章

# 设 计 验 证

随着 IC 工艺的不断发展，设计变得越来越复杂，SOC(片上系统)已成为 ASIC/FPGA 设计的一个重要趋势。EDA 业内人士普遍认为验证是产品到市场的一个瓶颈问题。百万门设计并不困难，而验证百万门的设计是一件非常难的事情。据估计，目前在一个 SOC 设计中，验证工程师的人数是设计工程师的 2 倍左右，验证工作占到整个设计的 60%～70%，而验证代码，则占到了全部代码的 70%～80%左右。显然，验证已经成为集成电路设计中一个非常重要的环节。本章将介绍验证的概念、基本方法和验证程序的写法。

## 4.1 验 证 综 述

### 4.1.1 验证的概念

图 4.1 是 Janick Bergeron 提出的表示验证过程的重复收敛模型(Reconvergence)。验证过程是证明设计正确的过程，验证的目的是为了保证设计实现与设计规范是一致的，保证从设计规范开始，经过一系列变换后得到的网表与最初的规范是一致的，即整个变换的过程是正确的。

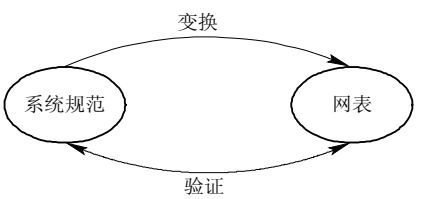



图 4.1 验证过程模型

图 4.1 中的变换可以理解成根据输入而产生输出的任何过程。从规范到网表之间可能包含了多个变换，比如一个 FPGA 设计可能包含以下几个变换：

(1) 从自然语言表述的系统规范变换成完整的、可验证的和无二义性的系统规范。

(2) 从系统规范变换成可实现的模块设计规范。

(3) 从模块设计规范变换成 RTL 及代码描述。

(4) 从 RTL 代码通过综合工具变换成门级网表。

(5) 从门级网表通过后端布局布线工具变换成具有延时信息的网表。

从规范到网表之间的变换包含了许多问题，如设计规范是否正确，有无矛盾之处；设计人员是否正确理解了设计规范；模块设计是否正确地反映了其功能；模块之间的接口是否正确；包含有延时信息的网表的时序是否满足要求。这些问题都是验证过程需要解决的问题。验证过程是为了开始的系统规范和最后的结果一致，如果验证过程和变换过程没有共同的开始点，那么就不会存在验证。验证是一个多次重复的过程，是一个不断向期望结果靠近的过程。

### 4.1.2 验证和测试

验证(Verification)和测试(Test)这两个概念通常被人们所混淆，它们实际上是 ASIC 设计流程中两个不同的环节。测试的目的是为了确认生产后的设计产品是正确的，而验证的目的则是为了确认设计符合设计规范。目前，验证一般是通过仿真实现的。在本章中，验证和仿真不加区分。设计和验证之间的关系可以用图 4.2 表示。

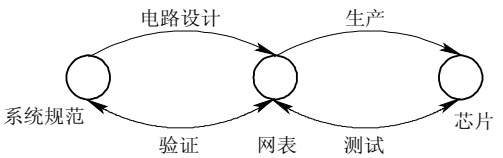



图 4.2 设计和验证

测试是通过测试向量实现的，一般由加工厂家或封装测试厂家完成。测试不是为了检查设计功能是否正确，而是为了检查生产后物理器件是否能完成正常的 0 到 1 或 1 到 0 的翻转。有大量关于测试的理论和方法研究，它们不属于本书的讨论范围，有兴趣的读者可以参看相应的参考资料。

### 4.1.3 自顶向下和自底向上的验证方法

硬件开发过程的发展和软件的非常类似。在 20 世纪 80 年代，硬件设计主要依赖手工画原理图，设计速度和规模都不是非常大。在 20 世纪 90 年代后，由于 EDA 工具的快速发展，越来越多的设计公司依赖于高层的 RTL 描述，借助于综合工具实现它们的设计。无论设计的复杂度还是设计的开发周期都比以前有了较大的提高。为了设计和验证更复杂的系统，硬件设计工程师在硬件设计中借助于软件工程师的经验和研究方法，形成了适合硬件系统设计和验证的方法。

#### 1．自顶向下的验证方法

在自顶向下的验证方法中，验证分成四个阶段。

(1) 系统级验证：在大中型设计项目中，验证方案往往与系统规范同步开始，在系统规范签收(Sign-off)完成之后，就开始了系统级的验证。根据系统规范对系统进行建模，并对建立的模型进行验证。用于系统级的建模工具有很多，如通用的语言 C、C++、HDL、SystemC，也可用专门的验证语言如 Sugar、Vera 和 Specman Elite，还可以是形式化的语言。

(2) 功能验证：它用于验证一个设计的 RTL 代码是否符合系统的规范。功能仿真是目前功能验证的主要方法。此外，形式化验证技术可以作为辅助的手段完成一些关键模块的

功能验证。

(3) 门级网表验证：通过功能仿真或形式化工具(如 E-CHECK 或 FORMALITY)检验 RTL 代码和综合后网表是否等价。

(4) 时序验证：它用于验证门级网表到含有延时信息的网表变换后，时序是否满足规范中关于时序的要求。同步设计的时序，一般通过静态时序分析工具完成验证。目前，各主要的 FPGA 厂家都有内嵌的静态时序分析工具。

**2．自底向上的验证方法**

自底向上的验证方法可以用图 4.3 表示，目前这种方法仍被大多数设计厂家所使用。该验证方法的解释如下：

(1) 设计文件通过词法扫描器(包括 HDL)确认没有语法错误，以保证设计文件是特定验证工具所能接收的语法子集。同时，使用 lint 检查工具验证设计代码中没有句法违规错误。

(2) 0 层验证：它用于独立地验证每个设计元件/块。这层的验证需要穷举模块的各种情况，保证每个单元的设计质量。直接仿真、随机仿真和模型检验等技术都可以用于 0 层验证。

(3) 1 层验证：它用于验证设计内部模块设计之间的接口和系统存储映射是否正确。验证的内容包括通过片上处理器或片外的处理器对各个模块中的寄存器的读/写操作，各个接口之间的配合是否正确等。

(4) 2 层验证：它是系统级验证，目标是为了验证整个设计的功能。本层验证主要集中在设计和外部环境之间能否协调工作，包括一些极端情况、边界条件和错误处理等。

图 4.3　自底向上的验证方法

(5) 门级网表验证和时序验证同自顶向下的验证方法中的是一致的。

### 4.1.4　主要验证技术

验证技术目前主要分为两类：基于形式化的验证和基于 Testbench(验证程序)的技术。

**1．形式化方法**

形式化验证技术通过数学的方法证明设计是否与规范一致。一般而言，如果用形式化工具证明设计的某个特性是正确的，那么验证人员可以不必再用 testbench 去仿真这些特性。工业界常用的形式化方法主要包括以下两种。

(1) 等价性检验：它主要是用于验证两个设计是否完全等价。等价性检验主要用于两个方面，一个是两个网表的比较，其目的是保证一个经过修改、插入扫描链、时钟树综合或手工修改后的网表与原有的网表功能是一致的；另一个是验证网表是否正确地实现了 RTL 代码。如果设计者完全相信综合工具是正确的，这个验证就可以省略。等价性检验可以用图 4.4 形象地表示。

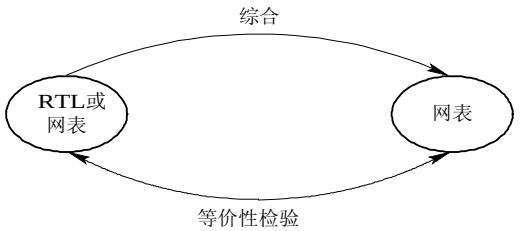



图 4.4　等价性检验

(2) 模型检验：该方法经过十几年理论和实践的探索，目前已经逐步应用到工业界。模型检验的主要思想是根据设计人员的 RTL 代码，提取有限状态机并穷举搜索设计的状态空间，验证用户定义的特性。如果要使验证的特性不成立，验证工具应产生一条从初态到失败状态之间完整的路径，设计人员根据这条路径找到错误状态。Cadence 公司推出的 FormalCheck 工具实现了模型检验技术。另外，IBM 公司的 Sugar，Sypopsys 公司的 Vera 和其他公司的专用验证语言吸收了模型检验的有关思想，使复杂的逻辑公式能以简洁的方式表达，极大地推广了模型检验的应用。目前，Sugar 语言已成为设计验证方面的工业标准。

其他形式化方法，如定理证明系统 HOL 和 PVS 等，目前还处在研究阶段。

### 2．基于 testbench 的验证

虽然模型检验已开始用于工业界，但是模型检验还有其局限性，一是能验证设计的规模和复杂度都是有限的；二是模型检验所能描述的特性有限。因此，目前确认功能是否正确的主要方法还是基于 Testbench 验证的验证方法。Testbench 在本书中的意思是利用 HDL 语言编写的用于产生设计输入序列的代码，也就是验证程序。基于 Testbench 验证主要有以下三种方法。

(1) 黑盒验证法。在黑盒验证法中，设计被当成一个黑盒子，对设计人员而言不知道内部设计细节，根据设计规范，验证设计是否符合规范。在这种验证方法中，验证与设计相分离，验证方案与电路的设计方案完全不挂钩，验证人员只关注规范，列出需要验证的特性，然后组织适当的用例(Testcase)来验证这些特性。黑盒验证模型如图 4.5 所示。

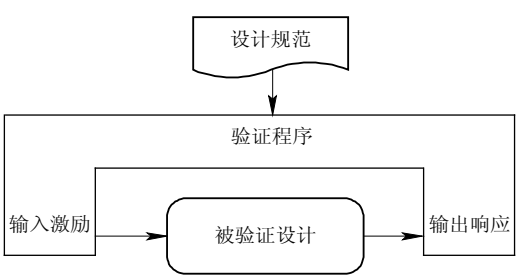



图 4.5　黑盒验证模型

黑盒验证法可以发现下面类型的错误：

① 初始化和中止错误。

② 接口错误。

③ 性能错误。

④ 未实现的或实现不正确的功能。

由于缺乏可观测性和可控性，黑盒验证法很难发现隐藏在设计内部的错误。

(2) 白盒验证法：这种方法为验证人员提供了很好的可控性和可观测性，这种方法有时可称为结构验证法。由于知道设计的内部细节，因此，很容易产生特殊情况的激励，检测内部设计的错误。验证环境的建立相对明确、简单，具有较强的针对性，结果检查相对来说也简单一些。这种方法被广泛应用于设计验证中。白盒验证法的缺点就是验证人员要知道设计内部的细节。白盒验证模型见图 4.6。

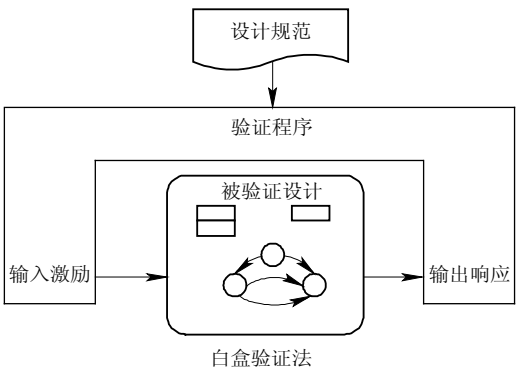



图 4.6　白盒验证模型

(3) 灰盒验证法：灰盒验证法是介于白盒和黑盒验证法之间的一种验证方法。验证人员在关心规范需求的同时又关心电路的详细设计方案，需要依据两者制定验证方案。如同黑盒验证法，灰盒验证法通过顶层接口控制和观察整个设计，但是又需要验证一些重要的特定的设计细节。

在一个设计中，通常是这三种方法结合起来一起使用。

### 4.1.5　验证工具的介绍

提高验证可靠性和有效性的一个主要机制是自动化。本节介绍验证过程中涉及到的一些主要验证工具。

#### 1．Lint 工具

Lint 工具是对设计代码进行静态的检查，以验证句法的正确性。因此，Lint 工具只能发现初始化的变量、接口不匹配和不支持的结构等句法错误，而不能发现设计错误。大部分仿真器和综合器都带有 Lint 检查工具。Lint 工具是静态的验证工具，它不需要任何附加的信息和用户要求的动作。

#### 2．仿真工具

仿真器是验证中最常用的工具。仿真不是一个项目的目标，所有硬件设计的最终目标是能在市场上销售并产生效益的真实硬件实现。仿真器试图创建一个能够模拟真实设计的人工环境，使设计工程师和设计进行交互，在设计生产之前发现设计错误，以减少损失。之所以称为仿真器，是因为它们是真实状态的一种近似。例如，一个数字仿真器假设一个信号只有 0，1，X(未知)和 Z(高阻)四种状态，而实际上信号是连续的，它具有无数多值。

仿真器是一个动态的验证工具，它要求验证人员提供一个能使设计正常工作的环境信息(或输入激励)，这个环境信息就是通过 Testbench 提供的。仿真器通过一定方式和设计人员交互，将设计的输出状态随设计环境变化的信息反映给设计人员。仿真器分为两种：事件驱动的仿真器和基于周期的仿真器。

1) 事件驱动的仿真器

只有在输入发生变化时，仿真器才去计算电路的模型，计算与输入相关的输出或中间信号的状态，这类仿真器我们称之为事件驱动仿真器。在这类仿真器中，输入的任何变化被定义成一个事件，该事件被传递到设计的各个部分。在一个周期中，由于输入的不同到达时间和信号的反馈，一个设计元件可能被计算几次。事件驱动的仿真器提供了非常精确的仿真环境，但是仿真的速度将随设计规模的增大而降低。事件驱动仿真器支持以下描述方式的设计：用 HDL 描述的行为设计、RTL 设计、门级和晶体管级设计。

目前工业界比较流行的事件驱动仿真器包括以下两种类型：

(1) 代码编译型的事件驱动仿真器：它接受用 HDL 语言描述设计，将设计编译成数据结构并执行。常用的仿真器有 Cadence 公司的 NC-Verilog 和 Synosys 公司的 VCS 仿真器(Verilog Compiled Simulation)。

(2) 代码解释型的事件驱动仿真器：它接受用 HDL 语言描述设计，逐行解释代码并运行。如 Cadence 公司的 Verilog-XL。

2) 基于周期的仿真器

在每个周期结束时计算电路的稳定状态，这种仿真器称为基于周期的仿真器。由于在一个周期内，仿真模型只计算一次，因此这类仿真器的速度比较快。另外，有些基于周期的仿真器只计算 0 和 1 两种状态，而不考虑 X(未知)和 Z(高阻)状态，因此可以进一步提高仿真速度。然而，基于周期的仿真器只能仿真同步电路，而对于包含异步输入、锁存器和多时钟的设计，靠这类仿真器不能得到正确的结果。

### 3．波形观察工具

波形观察器是最常见的和仿真器一起使用的验证工具。通过波形观察器的图形界面，设计人员可以直观地观察随时间变化的信号以及信号之间的相互关系，可以非常容易地定位设计错误或测试文件的错误。

### 4．代码覆盖分析工具

当一个设计的所有验证程序仿真都正确，设计中是否还存在某些功能或功能组没有得到验证呢？哪些设计没有被验证到？覆盖分析工具可以回答这个问题。覆盖分析技术最早源于软件测试，在 IC 验证中引入该技术的目的是为了找出仿真用例(Testcase)集合没有覆盖到的 HDL 代码，创建附加的仿真用例以提高代码覆盖率，从而提高设计的质量。在许多工程中，验证是否结束是以覆盖率是否达到规定的覆盖率要求为标准的，代码覆盖分析主要包括以下几个方面：

(1) 语句覆盖(Statement Coverage)分析：分析一个验证程序能覆盖代码的全部行数。分析工具可以让用户快速地浏览源代码并快速标识没有被执行的设计代码。

(2) 路径覆盖(Path Coverage)：分析一个验证程序通过 if…else 或 case 结构的所有可能的路径。

(3) 表达式覆盖(Expression Coverage)：分析说明哪一个 if…else…分支或 case 分支已被执行过。

(4) 触发覆盖(Triggering Coverage)：分析敏感变量中的信号是否惟一触发一个过程。

(5) 自动机覆盖(FSM Coverage)：分析仿真用例是否覆盖了所有的状态，所有的状态是否都 100％可达。

使用代码覆盖技术必须非常了解设计细节，通过代码覆盖分析工具了解哪些路径已经被执行，哪些表达式已经被执行，哪些过程没有被触发等等，然后修改验证程序，提高代码覆盖率。

工业界常用的仿真工具有：Synopsys 的 VCS，Cadence 的 NC-sim 以及 TransEDA 的 Verification navigator 等，它们都嵌有代码覆盖工具。

### 4.1.6 验证计划和流程

随着设计规模的不断加大，验证在整个设计周期中所占的比例越来越大，制定验证计划是功能验证过程中的一个重要环节，验证计划可以提高验证效率，减少验证的盲目性。一个典型的验证流程如图 4.7 所示。验证计划是在设计规范结束(Sign Off)之后开始的，验证工程师应该和总体设计师以及设计人员一起讨论整个设计功能，详尽理解设计规范以及和 DUV(被验证的设计)相连接的接口信号等。在此基础上制定验证计划以确定设计需要验证的所有特性，确定验证策略，规划验证环境和验证程序的开发，确定整个验证所需的验证人员的数目，资源和时间等等。

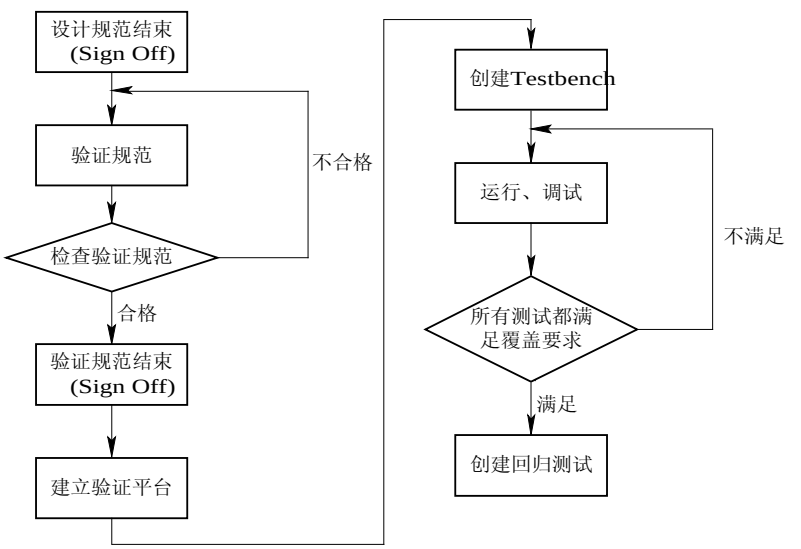



图 4.7 典型的验证流程

验证计划中需要说明以下问题：

(1) 确定设计需要验证的特性。从理解验证规范入手，和总体设计师以及设计人员认真讨论，确定设计需要验证的特性。

(2) 确定验证方法。确定验证过程中采用的方法，如前面所介绍的自底向上或自顶向下

的方法或其他验证方法。

(3) 确定验证策略，采用什么样的策略验证一个设计，主要包括下面几个方面：

① 确定实施验证的抽象层次和验证策略，不同的层次，采用的验证策略是不一样的。如果在模块级验证，可能采用白盒验证法；而为了测试接口，可能采用黑盒验证法；而系统功能验证，则有可能采用灰盒验证法。

② 激励产生策略，直接仿真激励或随机仿真激励。

③ 验证响应，响应的验证一般采用三种方法，观察法、记录法和自检查(Self-Checking)法。根据验证的内容，在这三种方法之间作一种折衷。

④ 确定验证的质量标准，如功能覆盖率、代码覆盖率等。

⑤ 根据验证的质量标准，制定相应的验证方案。

⑥ 确定验证资源和其他的相关问题，包括人力资源、机器资源和软件资源等，也包括验证过程的质量跟踪等方面的问题。

一个典型的验证流程如下：

(1) 确定验证规范。和总体设计师以及设计人员认真讨论，确定要设计、要验证的特性。对这些特性做一些简短的描述，最好和设计规范有一个对应关系，以便能没有疏漏地列出所有需要验证的特性。同时，说明这些特性的验证是在哪一个层次上进行的，如系统级、子系统级还是模块级等。由验证特性制定仿真用例，将列出的验证特性按它们的重要性进行优先级划分，有些特性对设计的成功有非常重要的影响，而有些特性只是锦上添花，许多用户可能都不用，验证应该更关注重要特性。如果说重要的特性验证得比较充分，那么设计的成功率就比较高，对于那些用户不用的特性，即便有些 bug，也不会影响芯片的使用。

另外，对于具有相同配置的特性或相关性比较密切的特性，可以将它们归入到同一个测试用例中。

(2) 根据提交的验证方案和设计人员一起讨论，检查是否有疏忽，是否存在不合理之处。如果有，则修改验证规范，如果没有问题，则验证规范就可以结束了。

(3) 根据验证规范的要求，建立相应的验证平台。验证平台可以借助于已有的验证平台，增加新的内容，节约开发时间和投入。如果没有相关的验证平台，则需要根据项目的功能，进行新的开发。

(4) 在验证平台的基础上，验证人员根据验证规范列出的验证特性优先级和制定的仿真用例，编写验证程序。

(5) 在验证平台上，运行验证程序，发现设计错误。

(6) 如果所有的验证用例都满足验证规范制定的覆盖率，那么可以进入步骤(7)的回归测试。如果验证用例没有达到验证规范制定的覆盖率，则采用随机测试或基于约束的验证用例。

(7) 如果设计修改，则进行回归测试。否则，验证结束。

回归测试是软件测试术语，也是电路功能验证时常用到的一种方法。它的基本意思是对修复好了的缺陷再重新进行测试，目的在于验证以前出现过但已经修复好的缺陷。在修正缺陷时必须更改源代码，这有可能影响这部分源代码所控制的功能，所以在验证修正的缺陷时不仅要按照缺陷原来出现时的步骤重新测试，而且还要测试有可能受影响的所有功能。

# 4.2 功 能 验 证

## 4.2.1 验证程序(Testbench)的组成

验证程序一般是指描述一个设计确定的输入序列和期望输出响应的代码集合，也可以包括外部数据文件或 C 语言程序。仿真程序提供设计的输入激励并监控设计的响应。

图 4.8 说明了仿真程序与被验证设计 DUV(Device Under Verification)之间的关系。注意，这个 Testbench 是完全封闭的，没有输入也没有输出。

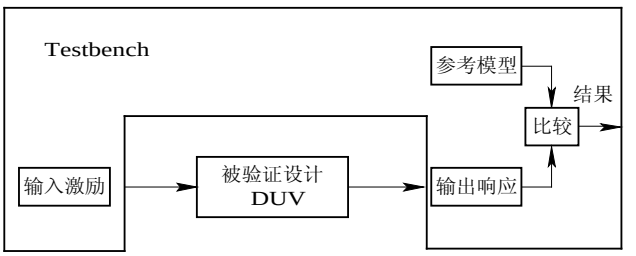



图 4.8 测试程序的构成

一个典型的 Testbench 应该由下面六个部分构成：

(1) DUV：它可以是 RTL 代码，也可以是网表。

(2) 输入激励：就是能使 DUV 工作的输入激励。

(3) 时序控制模块：用于产生仿真电路和 DUV 所需要的时钟信号。

(4) 参考模型：用于和 DUV 进行比较用的设计，参考模型可以是行为模型，也可以是已经验证过的设计。

(5) 诊断记录：在验证过程中，记录被验证设计中相关信号的变化情况。设计人员可以利用记录的信息找到错误。一种比较好的方法就是所谓的自检查方法，用期望的激励和被验证设计的输出的响应进行比较。如果结果不正确，那么报告出错，及时停止仿真。

(6) 断言检查器：断言是一种白盒验证法，可以通过它来检查机制，以发现设计内部的错误。关于断言的使用，将在后面的小节中详细介绍。

有了这六个构件，就不难理解为什么图 4.8 是一个既没有输入也没有输出的封闭系统了。下面通过一个例子说明 Testbench 的构成。

**【例 4.1】** 编码器设计的验证程序框架，如图 4.9 所示。

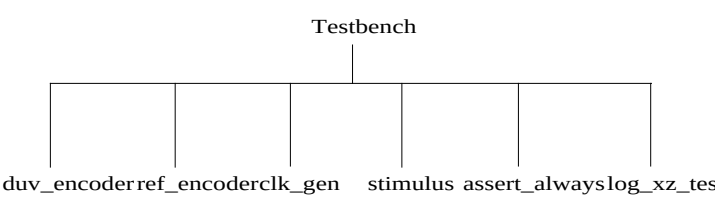



图 4.9 编码器验证程序构成

顶层的验证程序包括以下六个模块：

(1) 被验证设计 duv_encoder：该模块中包含了一个具有优先级编码的设计。

(2) 参考设计 ref_encoder：该模块中包含了一个用行为级代码描述的优先级编码设计。

(3) 时序控制模块 clk_gen：该模块用于产生本例中所需要的时钟。

(4) 输入激励产生模块 stimulus：这个模块通过一个计数器产生 0～31 之间的数值，作为编码器的输入。

(5) 诊断日志文件模块 log_xz_test：本模块在仿真的过程中建立一个文件，用于记录仿真过程中，需要记录的关键信息，同时该信息也显示在屏幕上。

(6) 断言检测模块 assert_always：这是一个断言模块，这个模块用于监控仿真过程中不正确的信息。如果参考设计的结果和实际设计的结果不一致，那么暂时停止仿真。

```
//=====================================//
//        以下是顶层仿真程序                    //
//=====================================//
'define EVENT1 1'b1
'define EVENT2 1'b0

module testbench;
wire clk;
wire [4:0] c_stimulus;
wire [2:0] c_codex;
wire [2:0] c_codez;
reg rst_n;
wire finish;

clk_gen ck_gen(.clk(clk));                          //时序控制模块
stimulus inst_sti(.clk(clk)，                       //仿真激励产生模块
                  .rst_n(rst_n)，
                  .c_stimulus(c_stimulus)，
                  .finish(finish));
ref_encoder inst_ref(.c_error_vector(c_stimulus)，  //参考设计模块
                  .c_code(c_codex));
duv_encoder inst_duv (.code(c_stimulus)，           //设计模块
                  .encoder(c_codez));
log_xz_test xz_test(.clk(clk)，                     //日志记录模块
                  .c_stim(c_stimulus)，
                  .c_codez(c_codez)，
                  .c_codex(c_codex)，
                  .finish(finish));
```

```
assert_always safety(.clk(clk)，                          //断言模块
                     .event_trig_1(`EVENT1)，
                     .test(c_codex==c_codez)，
                     .event_trig_2(`EVENT2));;
        initial
        begin
              rst_n = 1'b0;
            # 100 rst_n =1'b1;
          end
        endmodule

//==================//
//DUV//
//==================//
module duv_encoder(code，encoder);
input [4:0] code;                                  //5 位输入信号
output [2:0] encoder;                              //编码输出
assign encoder[0] = code[4] |
                    ~|code[4:1] & code[0] |
                    ~code[4] & ~code[3] & code[2];
assign encoder[1] = ~code[4] & code[3] | ~code[4] & ~code[3] & code[2];
assign encoder[2] = ~code[4]& ~code[3] & ~code[2] & ~code[1] & code[0] |
                          ~code[4]& ~code[3] & ~code[2] & code[1];
endmodule

//==================//
//参考设计模块//
//==================//
module ref_encoder(c_error_vector，c_code);
input [4:0] c_error_vector;
output [2:0] c_code;
reg [2:0] c_code;
always @(c_error_vector)
      begin
        casex(c_error_vector)
          5'b 1???? : c_code = 3'h1;
          5'b 01??? : c_code = 3'h2;
          5'b 001?? : c_code = 3'h3;
```

```
            5'b 0001? : c_code = 3'h4;
            5'b 00001 : c_code = 3'h5;
            default :
               c_code = 3'h0;
        endcase
    end
endmodule

//==================//
//激励产生模块//
//==================//
module stimulus(rst_n，clk，c_stimulus，finish);
  input rst_n;                    //全局复位，低有效模块
  input clk;                      //时钟
  output [4:0] c_stimulus;        //输入该参考设计和被验证设计的输入激励
  output finish;                  //文件关闭指示信号，当该信号为高电平时，关闭打开文件
  reg [4:0] c_stimulus;
  reg [5:0] r_counter;
  reg [5:0] c_counter;
  reg finish;
  initial begin
     finish = 1'b0;              //初始文件关闭信号为低电平
  end
  always @(r_counter) begin
       c_counter = r_counter + 6'd1;      //模 32 的加法器，用于编码器的输入
     if (r_counter == 6'h 20)             //32 个数据测试结束
   begin
            finish = 1'b1;                //文件关闭信号为高
            $finish();                    //仿真结束
  end
     c_stimulus = c_counter[4:0];
  end
always @(posedge clk or negedge rst_n)
    if (~rst_n)
      r_counter <= 6'd0;
    else
            r_counter <=c_counter;
endmodule
```

```
//==================//
//日志模块//
//==================//
'define DELAY_LOGGING #1
module log_xz_test(clk，c_stim，c_codex，c_codez，finish);
input clk;
input [4:0] c_stim;
input [2:0] c_codex，c_codez;
input finish;

integer file;
   always @(posedge clk) begin
    'DELAY_LOGGING
      $display("%t   %b   %h   %h"，$time，c_stim，c_codex，c_codez);
   end

initial
begin
       file = $fopen("encoder.log");       //在工程目录下，建立一个日志文件
       if (finish)                         //仿真结束，关闭日志文件
           $fclose(file);
    end

initial
begin
    $fdisplay(file，"time     c_stim     c_codex     c_codez");  //日志文件的头信息
    end
always @(posedge clk) begin              //在每个时钟周期记录仿真结果
    'DELAY_LOGGING
$fdisplay(file，"%t   %b   %h   %h"，$time，c_stim，c_codex，c_codez);
 end
endmodule
//==================//
//断言模块//
//==================//
'define DELAY_ASSERT #2;
module assert_always(clk，
          event_trig_1，
```

```
            test，
            event_trig_2);

    input clk;
    input event_trig_1;
    input test;
    input event_trig_2;

    reg test_state;
    initial test_state = 1'b0;
    always @(event_trig_1 or event_trig_2)        //触发监控
      if (event_trig_1 || event_trig_2)
        test_state = ~event_trig_2 && (event_trig_1 || test_state);

    always @(posedge clk)                         //监控 test 事件是否发生
    begin
      'DELAY_ASSERT
        if ((test_state == 1'b1) && (test !=1'b1)) begin
          $display ("ASSERT ERROR %t   %b: %m ",  $time,  test);
          $stop;                                  //test 时间发生，暂时停止仿真
        end
      end
    endmodule
```

在顶层设计中，定义了两个事件 EVENT1 和 EVENT2，它们用于断言监控模块中是否触发监控。本例中监控总是被触发。

### 4.2.2 实用构造 Testbench 技术

#### 1. 使用行为级代码描述验证模型

所有有经验的硬件设计工程师都习惯于编写第 3 章中讨论的可综合代码模型，他们在编写 Verilog 代码时，无论是设计代码还是用于验证的代码，往往从实现的角度出发，写出的代码都是可综合的。实际上，用于验证的代码没有必要考虑到内部的实现，只需要按规范描述出一个设计的功能就可以了，也就是说只要建立一个设计模型就可以了。

**【例 4.2】** 考虑一个简单的设计。在 SDHSTM-0 帧结构中，需要根据帧定位 A1 的位置确定出指针 H1 和 H2 的位置。也就是说，在 A1 信号变高后的 270 个周期后，H1 便保持高电平一个周期，H2 在 H1 变高后保持一个周期的高电平，假设 A1 信号每 810 个时钟周期来一次。

设计工程师一般用计数器的方法，靠统计 A1 后周期的个数计算出 H1 的信号，他们的代码往往是 RTL 可综合风格的。而验证工程师则从功能的角度出发，他们的代码往往是描述行为的。RTL 和行为代码分别见图 4.10。

```
// rtl_module
….
reg [8:0] cnt_270;
wire h1_pos;
assign h1_pos ;
always @(clock clk)
if (a1_pos)
   cnt_270 <= 9'd 0;
else
   cnt_270 <= cnt_270 + 1;
assign h1_pos = (cnt_270 == 269)
…
```

```
//behavior;
reg h1_pos;
initial
begin
 h1_pos = 1'b0;
 forever begin
 if (a1_pos) = 1'b0
    h1_pos = 1'b0;
 else begin
      # (269 * cycle) h1_pos = 1'b0
      #cycle          h1_pos = 1'b1;
  end
```

图 4.10　RTL 和行为代码

这两种写法都能根据 A1 信号，正确产生地 H1 和 H2 信号，但是它们之间是有区别的。

在 RTL_model 描述中，由于 always 对时钟 CLK 敏感，因此在每个时钟周期，仿真器都重复计算并更新推断出的寄存器的值。而行为模型并不随时钟而同步改变状态，只是在需要的时候才计算，因此行为模型的仿真速度比 RTL 级的速度要快。另外，为了观察波形，RTL_model 记录的信息量要比行为模型记录的信息量要大的多。假设我们仿真 1000 ns，在 Cadance 仿真环境下，记录.shm 文件，在仿真结束后，比较两个 shm 文件，结果 rtl_model.shm 文件比 behavioral_model.shm 大 100 倍之多。

在许多情况下，行为模型代码的描述往往比 RTL 代码的描述简单的多。【例 4.3】给出了一个自动机模型。我们给出行为代码和 RTL 级代码的描述。

**【例 4.3】**　图 4.11 是一个握手协议的自动机模型。这个模型的含义是：一旦检测到确认信号(ACK)变为高电平，则将请求信号(REQ)设置成低电平。

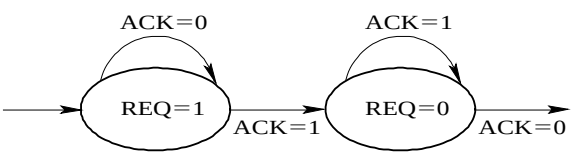



图 4.11　握手协议

```
//=======================================//
// Verilog RTL code for handshaking protocal          //
//=======================================//
parameter make_req = 1'b 0;
parameter release     = 1'b 1;
reg next_st，curr_st;
always @(ack or curr_st)
   case (curr_st)
```

```
begin
  make_req :    req <= 1'b1;
            if (ack)
               next_st <= release;
             else
               next_st <= make_req;
            end
  release : begin req <= 1'b0;
          if (ack)
             next_st <= release;
          else
             next_st <= …
…
endcase

//behavior description
always
  begin
     @(posedge ack) req <= 1’b0;
     @(negedge ack) req <= 1’b1;
  end
```

显然，行为代码要简单的多，可读性也较可综合风格代码要好。

设计工程师从物理实现上考虑 Verilog 代码的写法，受综合工具的影响，他们编写的代码需要遵循一定的规则，代码的好坏，划分是否合理直接影响到综合的结果。而验证工程师则不关心物理实现，他们关心的是验证代码是否真实地描述了规范。因此，他们写出的代码可以是不可综合的。验证工程师不必像设计工程师那样遵循特定的编码风格和划分模块的原则，他们往往可以根据规范的要求，按照功能来划分模块。

由于目前 Verilog 语言的可综合子集支持的数据结构非常少，因此硬件设计工程师描述他们的复杂设计时必须把复杂的结构映射为二值逻辑，用一维向量或存储器类型结构实现。但是行为描述则不局限于此，验证工程师可以灵活地使用实数、多维数组、记录和链表等丰富的数据结构描述一个功能。

据估计，在大规模的 IC 设计中，验证文件的代码量占到了整个设计代码量的 80%左右，验证代码与设计代码一样需要调试。如果验证代码写成行为模型风格的 Verilog 代码，由于描述同样规范的行为代码比 RTL 代码要简单，代码越简单，越容易查错，调试的工作量就越少。在行为代码中，大部分代码是顺序执行的代码，它们比并发型代码更容易纠错，因为它们不牵扯到并行代码之间的同步问题和复杂的数据交换问题。

无论是可综合的子集还是不可综合的子集都可以用来编写 Testbench。可综合的子集编写的仿真代码有以下优点：

(1) 便于移植到基于周期的仿真器上。

(2) 仿真电路的代码没有冒险和竞争。

但是可综合代码写仿真电路的缺点在于：

(1) 代码比较长，可读性较差。

(2) 从仿真器的角度来看，可综合 RTL 代码的仿真性能比较低。

(3) 只能处理可综合实现的数据类型：比特、向量比特和整数。而其他一些数据结构如实数、多维数组、记录等无法实现。

### 2．使用抽象数据类型

行为级代码可以不受可综合代码的约束，在更高的层次上实现数据的抽象，使得验证的层次与设计层次相对应。下面主要讨论用 Verilog 语言实现一些抽象数据结构的方法。

1) 实数的实现

在 Verilog 语言中，实数不可以通过接口进行传递，函数可以返回一个实数值，但是，实数不能作为函数或任务的输入变量。为了能让实数作为任务或函数的输入变量，可以通过调用系统函数$realtobits 和$bitstoreal 将一个实数转换成 64 比特向量或将 64 比特向量转换成实数。

**【例 4.4】** 用 Verilog 实现一个滤波器函数：

$$y(n)=a_0x(n)+a_1x(n-1)+a_2x(n-2)+b_1y(n-1)+b_2y(n-2)$$

在上面的表达式中，y(n)的值是通过上一次的函数值 y(n－1)，y(n－2)，x(n)，x(n－1)和 x(n－2)计算得到的，在计算过程中，需要保留每一次的计算结果。这个表达式可以通过函数调用来实现。在 Verilog 语言中所有的寄存器变量都是静态的，它们在编译的时候就被分配好了，仿真过程中在内存中一直有变量的备份。

```
module test_real;
parameter a0= 0.50000，a1=1.125987，a2=-0.097743，b1=-0.1009373，b2=0.009672;
real y;
function real yn;
input [63:0] xn;
real xn_1，xn_2，yn_1，yn_2;        //寄存器变量
begin
   yn = a0 * $bitstoreal(xn) + a1 * xn_1 + a2 * xn_2 + b1 * yn_1 + b2 * yn_2 ;
   xn_2 = xn_1;
   xn_1 = $bitstoreal(xn);
   yn_2 = yn_1; yn_1 = yn;
   $display("%f %f %f"，yn，yn_1，yn_2);
   $display("%f %f %f"，xn，xn_1，xn_2);
end
endfunction
initial
          begin
           //初始化滤波器参数
```

```
        yn.xn_1 = 0.0;   yn.xn_2 = 0.0;   yn.yn_1 = 0.0;   yn.yn_2 = 0.0;
        y = yn ($realtobits(1.0));
        repeat (10) begin
          y = yn($realtobits(0.0));
          $display("%f"，y);
        end
    end
    endmodule
```

2) 记录

记录是一种抽象的数据结构，可以由不同类型的信息组成，也可以方便地表示具有一定结构的数据。例如，ATM 中的信元，就可以用记录表示，又如 SDH 中的帧结构也可以用记录实现。Verilog 语言本身并不支持记录结构，但是可以通过一些方法来模拟记录的实现。模拟的基本方法是: 创建一个没有参数的 module，内部的所有变量都用寄存器类型声明。当模块实例化后，用模块中定义的变量表示记录中的域。【例 4.5】是 Verilog 语言产生 ATM 信元发送和接收的例子。

**【例 4.5】** 用 Verilog 语言实现图 4.12 所示的 ATM 信元结构。其中有 12 位的 VPI，16 位的 VCI，2 的位 PT，1 位的 CLP，其余是 48 个字节的净负荷(PAYLOAD)。

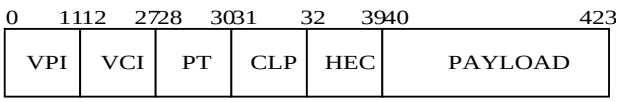



图 4.12 ATM 信元结构

```
module atm_cell_type;

//定义 ATM 信元结构
reg [11 : 0] vpi;
reg [15 : 0] vci;
reg [2 : 0]   pt;
reg clp;
reg [7 : 0] hec;
reg [7 : 0] payload [0:47];

//定义变量
integer bit_count;
reg is_vpi，is_vci，is_pt，is_clp，is_hec，is_payload;
reg [15 : 0]   temp;

//形成一维向量
function [0 : 423] ToBits;
input dummy;
begin
```

```
    ToBits = {vpi，vci，pt，clp，hec，
        payload[0]，  payload[1]，  payload[2]，  payload[3]，  payload[4]，  payload[5]，
        payload[6]，  payload[7]，  payload[8]，  payload[9]，  payload[10]，  payload[11]，
        payload[12]，  payload[13]，  payload[14]，  payload[15]，  payload[16]，  payload[17]，
        payload[18]，  payload[19]，  payload[20]，  payload[21]，  payload[22]，  payload[23]，
        payload[24]，  payload[25]，  payload[26]，  payload[27]，  payload[28]，  payload[29]，
        payload[30]，  payload[31]，  payload[32]，  payload[33]，  payload[34]，  payload[35]，
        payload[36]，  payload[37]，  payload[38]，  payload[39]，  payload[40]，  payload[41]，
        payload[42]，  payload[43]，  payload[44]，  payload[45]，  payload[46]，  payload[47] };
end
endfunction

//计算不同的域在一维向量中的位置，并取出各个比特
function Send_Bit;
input dummy;
begin
    is_vpi        = bit_count<12;
    is_vci        =(bit_count<(12 + 16)) & (bit_count >= 12);
    is_pt         =(bit_count<(12 + 16 + 3)) & (bit_count >= (12 + 16));
    is_clp        =(bit_count<(12 + 16 + 3 + 1))& (bit_count >= (12 + 16 + 3));
    is_hec        =(bit_count<(12 + 16 + 3 + 1 + 8))&(bit_count >= (12 + 16 + 3 + 1));
    is_payload = bit_count>=(12 + 16 + 3 + 1 + 8 );
    if (is_vpi)
        Send_Bit = vpi[11- bit_count];
    else if (is_vci)
        Send_Bit = vci[15 - (bit_count - 12)];
    else if (is_pt)
        Send_Bit = pt[2 - (bit_count - 12 - 16)];
    else if (is_clp)
        Send_Bit = clp;
    else if (is_hec)
        Send_Bit = hec[7 - (bit_count - 12 - 16 - 3 - 1)];
    else
    begin
        temp[7:0] = payload[(bit_count - 12 - 16 - 3 - 1 - 8) / 8];
        Send_Bit = temp[7 - ((bit_count - 12 - 16 - 3 - 1 - 8) % 8)];
    end
        bit_count    = bit_count + 1;
end
```

```
endfunction
endmodule

//测试程序
module tb_ATM;

atm_cell_type cell_Send();                    //实例化一个 ATM 信元

reg clk;
integer i;
reg atmdata;
reg [0 : 423] serialBits;
reg isequal;

initial
begin
  /* 给 ATM 信元的各个比特赋值 */
    cell_Send.vpi        = 12'h 090;
    cell_Send.vci        = 16'h ffff;
    cell_Send.pt         = 3'h 3;
    cell_Send.clp        = 1'b0;
    cell_Send.hec        = 8'h 09;
    for (i =0; i <=48; i = i+ 1)
        cell_Send.payload[i] = 8'h 5a;
    cell_Send.bit_count = 0;
    serialBits                = 424'h0;
end

initial
begin
  clk = 0;
end

always #5 clk = ~clk;

always @(posedge clk)                        //按比特串行发送该信元
    atmdata = cell_Send.Send_Bit(0);

always @(negedge clk)
begin                    //接收信元
    serialBits = {serialBits[1:423]，atmdata};
```

```
        if (serialBits == cell_Send.ToBits(0))     //比较发送和接收信元是否相等
            isequal = 1'b1;
        else
            isequal = 1'b0;
    end
    endmodule
```

在上面这个例子中，定义了没有任何输入和输出的模块 atm_cell_type，在该模块中定义了图 4.12 所示的 ATM 信元的各个字段。由于这个记录不是一个真正意义上的记录，因此它们不能进行直接比较和直接赋值等相关操作，所以，需要将记录转换成等价的一维数组，一旦变成了数组，就可以进行比较和赋值等操作。tb_ATM 是一个验证程序，该模块首先实例化 atm_cell_type，实际上就是形成了一个 ATM 信元的记录结构，然后给这个结构中的不同域进行赋值。用一个时钟驱动形成串行的比特流 atmdata 进行发送，atmdata 应该连接到被验证的 ATM 设计上，在这里只是为了说明如何形成 ATM 数据流，省略了设计。在这个程序中还有一个接收过程，这个接收过程将接收的 ATM 流和发送码通过一维数组形式进行比较。

3) 多维数组

二维数组是一种常用的数据结构，在实际的设计中，常常用于对 RAM 等数据结构的建模。对于仿真而言，二维数组提供了构造较复杂的数据结构的一种简单方法。有些情况下，测试激励需要构造有固定格式的循环反复的数据，使用二维数组是一种较好的方法。下面举例说明之。

**【例 4.6】** 使用二维数组产生以太网帧，以太网帧格式如图 4.13 所示。

| Preamble&SFD | DA | SA | TYPE/LEN | Payload | Padding | FCS |
|---|---|---|---|---|---|---|

图 4.13 以太网帧结构

图 4.13 中各项分别如下：

(1) Preamble&SFD 是以太网的前导码，通常由 7 个字节的“55”数字和一个字节的“d5”数字组成，在接收以太网帧时，从 55→d5 的跳变确定以太网帧的开始位置。

(2) DA 是以太网帧的目的地址，即哪个站点应当接收该以太网帧。该字段占用 6 个字节。

(3) SA 是以太网帧的源地址，即哪个站点发出了该以太网帧。该字段占用 6 个字节。

(4) TYPE/LEN 字段有双重定义，该字段占用 2 个字节。当 TYPE/LEN 字段值大于 1536 时，它表示以太网帧中承载的净荷是哪种上层协议数据，例如 0800H 表示是 IP 包；当 TYPE/LEN 字段值小于等于 1536 时，表示以太网帧的长度。

(5) Payload 字段为以太网帧装载的上层协议的净荷，如 IP 协议数据。

(6) Padding 字段为可选的附加字段。以太网帧的帧长是 64～1518 字节，当净荷 Payload 字段填入后帧长仍然小于 64 字节，则填入 Padding 字段，以满足最小 64 字节的要求。

(7) FCS 为 CRC32 校验字段，占用 4 个字节。它对从 DA 到 Padding 的所有数据计算从而产生 CRC32，接收方检测该字段以判断帧是否正确。

```
module ethernet_data(rst_n，clk，tx_en，tx_dat8);
output rst_n，clk，tx_en ; reg rst_n，clk，tx_en ;
output [7:0] tx_dat8 ;
reg [7:0] tx_dat8 ;
reg [7:0] tx_dat_defined [63:0];

//首先定义以太网帧
initial
begin
    tx_dat_defined[00] = 8'H00 ;   //DA
    tx_dat_defined[01] = 8'H07 ;
    tx_dat_defined[02] = 8'H95 ;
    tx_dat_defined[03] = 8'HE8 ;
    tx_dat_defined[04] = 8'H79 ;
    tx_dat_defined[05] = 8'H82 ;
    tx_dat_defined[06] = 8'H00 ;        //SA
    tx_dat_defined[07] = 8'H0A ;
    tx_dat_defined[08] = 8'HE6 ;
    tx_dat_defined[09] = 8'HE3 ;
    tx_dat_defined[10] = 8'H5C ;
    tx_dat_defined[11] = 8'H4E ;
    tx_dat_defined[12] = 8'H08 ;        //类型
    tx_dat_defined[13] = 8'H00 ;
    tx_dat_defined[14] = 8'H45 ;        //IP 版本号为 4，头长度为 20
    tx_dat_defined[15] = 8'H00 ;        //服务类型为 0
    tx_dat_defined[16] = 8'H00 ;        //全部长度
    tx_dat_defined[17] = 8'H3C ;
    tx_dat_defined[18] = 8'H6C ;        //标识符
    tx_dat_defined[19] = 8'H63 ;
    tx_dat_defined[20] = 8'H00 ;        //标志为 0
    tx_dat_defined[21] = 8'H00 ;        //偏移量为 00
    tx_dat_defined[22] = 8'H80 ;        //存储时间
    tx_dat_defined[23] = 8'H01 ;        //协议为 ICMP
    tx_dat_defined[24] = 8'H4B ;        //头校验和
    tx_dat_defined[25] = 8'H8A ;
    tx_dat_defined[26] = 8'HC0 ;        //源 IP 地址为 192.168.0.220
    tx_dat_defined[27] = 8'HA8 ;
    tx_dat_defined[28] = 8'H00 ;
    tx_dat_defined[29] = 8'HDC ;
    tx_dat_defined[30] = 8'HC0 ;        //目标 IP 地址为 192.168.0.167
```

```
        tx_dat_defined[31] = 8'HA8 ;
        tx_dat_defined[32] = 8'H00 ;
        tx_dat_defined[33] = 8'HA7 ;
        tx_dat_defined[34] = 8'H08 ;        //ICMP 类型为 8
        tx_dat_defined[35] = 8'H00 ;        //ICMP 代码为 0
        tx_dat_defined[36] = 8'H00 ;        //ICMP 校验为 00 84
        tx_dat_defined[37] = 8'H84 ;
        tx_dat_defined[38] = 8'H02 ;        //ICMP 标识
        tx_dat_defined[39] = 8'H00 ;
        tx_dat_defined[40] = 8'H4A ;        //序列号
        tx_dat_defined[41] = 8'HD8 ;
        tx_dat_defined[42] = 8'H00 ;        //填充值
        tx_dat_defined[43] = 8'H00 ;
        tx_dat_defined[44] = 8'H00 ;
        tx_dat_defined[45] = 8'H00 ;
        tx_dat_defined[46] = 8'H00 ;
        tx_dat_defined[47] = 8'H00 ;
        tx_dat_defined[48] = 8'H00 ;
        tx_dat_defined[49] = 8'H00 ;
        tx_dat_defined[50] = 8'H00 ;
        tx_dat_defined[51] = 8'H00 ;
        tx_dat_defined[52] = 8'H00 ;
        tx_dat_defined[53] = 8'H00 ;
        tx_dat_defined[54] = 8'H00 ;
        tx_dat_defined[55] = 8'H00 ;
        tx_dat_defined[56] = 8'H00 ;
        tx_dat_defined[57] = 8'H00 ;
        tx_dat_defined[58] = 8'H00 ;
        tx_dat_defined[59] = 8'H00 ;
        tx_dat_defined[60] = 8'H56 ;        //FCS
        tx_dat_defined[61] = 8'HCF ;
        tx_dat_defined[62] = 8'H2B ;
        tx_dat_defined[63] = 8'HB0 ;
    end
    parameter clk_width = 40;               //时钟 25 MHz
    integer   i ;
    initial
    begin
        rst_n = 1 ;
        clk   = 0 ;
```

```
#1 rst_n = 0 ;
#1 rst_n = 1 ;                          //产生复位信号
end
always #(clk_width/2) clk = ~clk ;      //产生时钟

//产生以太网帧数据
initial
begin
    forever                             //循环发送数据
    begin
        #(clk_width*24);                //产生以太网帧间隔
        txen = 1;                       //以太网数据使能
        for(i=0;i<=63;i=i+1)            //连续发送 64 个字节以太网数据
        begin
            tx_dat8=tx_dat_defined[i];
            #clk_width;
        end
        txen = 0 ;                      //发送完成，清除发送使能
        tx_dat8 = 0 ;                   //清零发送数据
     end
end
endmodule
```

在本例中，首先定义了发送的以太网数据帧，在一帧发送结束后，将发送使能清零，再进行下一次发送。有兴趣的读者，可以按照【例 4.5】将本例修改成记录格式，以便发送不同的以太网帧数据。

对于三维以上的数组应用，可以将三维数组变换成二维数组再实现。

### 3．编写有结构的仿真代码

从代码可维护的角度看，行为代码通常按功能和需求划分结构。如果功能非常复杂，应该把功能划分为若干个子功能，然后编写行为代码实现这些子功能。在 Verilog 中，可以用 module，function 和 task 实现仿真代码的结构化。

封装是实现结构化仿真编码的主要手段。封装的主要思想是将实现的细节隐藏起来，将功能和它的实现完全分离开，只要封装的接口不变，实现的修改和优化将不影响用户的使用。这也是仿真代码可重用的基本出发点。下面介绍实现封装的几种方法。

1) 变量局部化

方法一：变量声明。最简单的封装是尽可能地将变量的声明局部化，这种方法避免了局部变量与其他模块相互作用产生不正确的结果。

**【例 4.7】** 在下面的两个循环句子中，I 是公用的，在执行时可能会导致意想不到的结果。

```
integer I;
always begin
    for (I = 0; I<=32；I = I + 1) begin
    …
end

always begin
 for (I=5; I>0；I = I - 1) begin
   …
  end
```

对上述结构稍加修改，使得变量说明局部化。将每个 always 块命名一个名称，为了重复执行的次数 I 不影响其他模块的执行，将变量 I 的说明放在每个模块的内部。程序修改如下：

```
always begin ： block_1
      Integer I;
 for (I=0; I<=32; I = I+1) begin
 …
end

always begin : block_2
integer I;
 for (I=5; I>0l; I=I-1) begin
 …
end
```

通过正确的封装，这些局部变量就不会被其他 always 和 initial 块所访问而产生不良的效果。

方法二：用 task 和 function 使变量局部化。在 Verilog 语言中，用 task 和 function 也可以使说明局部化。

**【例 4.8】** sin 函数可以用 function 来实现，在 function 中定义的 real x，x1，y，y2，y3，y5，y7，sum，sign 都是局部变量。

```
function real sin;
  input x;
  real x;
  real x1，y，y2，y3，y5，y7，sum，sign;
  begin
   …
  end
endfunction
```

2) 封装子程序

有些子程序在整个项目中或不同的项目都非常有用，有三种方法可以实现子程序的封装。

方法一：在使用子程序时，将它们复制到验证程序中，这种方法的缺点是代码可维护性差，增加代码量。

方法二：将这些公用的代码放在一个文件中，然后通过’include 命令将它们包含在需要它们的验证程序中。

**【例 4.9】** 下面的一段代码是常用的显示错误信息的代码，它以任务的形式构造，放在 msg.v 文件中。其他模块用到错误信息显示时，用 include 将代码包含进来。

```
//FILE msg.v
task write_error;
input[14:0] addr;
begin
      $display ("read register %h doesn't equal write value，  -ERROR-"，  addr);
endtask

//invoke msg.v
   …
'include "msg.v"
if (…) error(14'h 0090);
   …
   ….
if (…) error(14'h 0010)
```

但是这种实现方法的缺点是：

① 每个包含任务的模块要编译 task，因而 task 任务不可能包含全局变量。

② 不能将编码封装在所需要的模块中，因为 task 没有包含在一个模块中。

方法三：将任务放在仿真模块中，但是不在任何使用任务的模块中实例化。通过一个绝对的层次路径访问该模块。

**【例 4.10】** 本例将任务独立地封装在模块中，然后在仿真程序中调用这些任务。

```
module syslog;
integer warings;
integer errors;
initial
begin
   warning = 0;
   errors   = 0;
end

task warn;
   input [80:0] msg;
   begin
      $write("warning at %t : %s"，$time，msg);
```

```
            warnings = warnings + 1;
        end
endtask
…
endmodule

module testcase
initial
begin
    …
     if (…) syslog.warn("Unexpected respones")
    …
 end
```

3) 总线功能模型 BFM(Bus Function Module)

(1) BFM 简介。目前，EDA 界广泛用总线功能模型 BFM，有时也称为事务处理程序(Transactions)描述模块的功能。所谓 BFM 就是 DUV 和 Testbench 之间的一种抽象。它是任务的集合，集合中的每个任务完成一个特定的事务，事务可以是非常简单的操作，如内存的一次读/写，也可以是非常复杂的操作，如通信中有结构的数据包。BFM 被直接连接到 DUV 上。图 4.14 中给出了 BFM，Testbench 和 DUV 之间的关系。

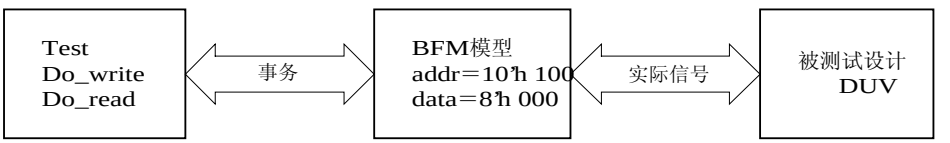



图 4.14　事务和事务处理程序之间的关系

(2) 实例。下面我们通过一个 CPU 接口的例子说明 BFM 的概念。

**【例 4.11】** 在 CPU 接口应用中，我们通常需要对某个寄存器的特定位进行设置。为了完成这个任务，首先是根据地址读出寄存器的值，然后将改写位(设置位)的值和不改写位的值一起再回写到该寄存器中。

我们可以将 CPU 接口抽象成图 4.15 所示的形式。根据预定义的协议，由 CPU BFM 产生 CPU 接口所需要的实际物理信号，如图 4.15 右侧所示，而左侧接口用特定的数据初始化一个事务，根据不同的事务，CPU BFM 产生不同的物理信号，把左侧的接口称为过程接口。

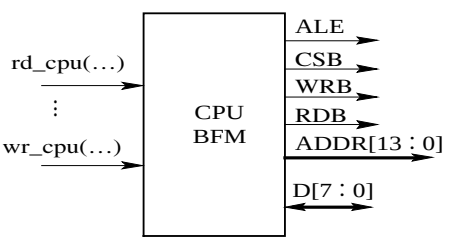



图 4.15　CPU 接口模型

我们可以根据 CPU 接口的时序，应用 task 产生一次 CPU 读写操作所需要的 CPU 地址、读/写信号、片选信号等。假设这个 CPU 接口的地址总线是 14 位，数据总线是 8 位。

```
/* 定义 CPU 接口所需要的信号 */
module cpu_interface( A，          //地址总线
                      D，          //数据总线
                      ALE，        //ALE 地址使能信号
                      WRB，        //CPU 写信号
                      RDB，        //CPU 读信号
                      CSB);        //CPU 片选信号
output [13:0] A;
inout [7:0] D;
output ALE;
output WRB;
output RDB;
output CSB;
reg [13:0] A;
reg ALE;
reg WRB;
reg RDB;
reg CSB;
//定义 CPU 读/写信号的所需要的时序参数
parameter tSalr = 10，   tSlr = 5，  Prd = 30，  tHrd = 10，  tHar = 10，
          tVl = 5，  tSalw = 10，  tVwr = 40，  tSdw = 20，  tSlr = 5；
parameter data_width = 8，  addr_width = 14;
reg [data_width :0] work_reg;

task rd_cpu;                        //读操作所需要的时序
input [addr_width:0] addr;
#10    A   = addr;
       CSB = 1'b0;
#(tSalr-tVl) ALE= 1'b0;
#tVl         ALE=1'b1;
#tSlr        RDB = 1'b0;
#Prd         work_reg = D;
#tHar        A = 14'h 0000;
             CSB = 1'b1;
end
endtask

task wr_cpu ;                  //写 CPU 操作
```

```
input [addr_width:0] addr;
input [data_width:0] write_value;
begin
#10 A = addr;
    CSB = 1'b0;
#(tSalw-tVl) ALE = 1'b1;
#tVl            ALE = 1'b0;
#tSlw           ALE = 1'b0;
#(tVwr-tSdw) D = write_value;
                wrb = 1'b1;
#tHdw           release D;
                A = 14'h 0000;
                CSB = 1'b0;
end
endtask

//设置 CPU 所写值
task set_value;
input [addr_width-1:0] addr;              //进行 CPU 操作的地址
input [data_width-1:0] expect_bits;       //8 位，如果在对那一位进行操作，则把该位设置成 1
input[data_width-1:0] expect_value;       //8 位，设置的值
reg [data_width-1:0] written_value;
begin
rd_cpu(addr);   //调用读任务，从相应的地址中读出相应的值，放在寄存器变量 work_reg 中
written_value = ~expect_bits & work_reg | expect_bits & expect_value;
                                          //将需要设置的值写入到对应位
wr_cpu(addr，  written_value);
//写入对应位
rd_cpu(addr);                             //读出 addr 的内容
if (~(work_reg == written_value)) begin              //写入值是否正确
  $display("－ERROR－，  register %h write wrong !"，addr);
  $stop; //如果写入值错误，那么仿真暂停
end
end
endtask
endmodule
```

上面的例子包含了三个任务。第一个任务 rd_cpu 是读指定地址的寄存器内容；第二个任务 wr_cpu 是将指定的值写入到指定的寄存器中；第三个任务 set_value 是在前两个任务的基础上构造而成的，首先读出指定寄存器的内容，然后将读出寄存器内容用指定位(expect_bits)的内容(expect_value)替换，其他位的值保持不变，得到一个值放到 written_value

中，将它的值再写入到指定寄存器(addr)中。最后将读出写入的寄存器的内容和 written_value 比较。如果结果正确，说明所指定的值正确地写入到了指定的寄存器。否则给出错误提示，同时仿真停止。

另外上面的例子包含了一些参数定义，这些参数可以根据不同类型 CPU 的读/写时序进行定义。本例在任务调用的时候，省略了 CPU 时序参数的传递。

4) BFM 的调用

下面介绍验证程序调用 cpu_interface 接口中的任务 rd_cpu，wr_cpu，set_value 的方法。

方法一：在测试程序中，通过实例化 cpu_interface 模块，直接将 cpu_interface 的实例和 DUV 连接，通过层次关系调用 cpu_interface 中的任务。

【例 4.12】 调用 cpu-interface 接口。

```
module testcase;
  …
  …
cpu_interface cpu(.A(A),
                  .D(D),
                  .ALE(ALE),
                  .WRB(WRB),
                  .RDB(RDB),
                  .CSB(CSB));
DUV   DUV_inst( /* other signal */
                  …
                  /* cpu signal */
                  .A(A),
                  .D(D),
                  .ALE(ALE),
                  .WRB(WRB),
                  .RDB(RDB),
                  .CSB(CSB),
                  /* other signal */
                  …      );
initial
begin
#300   cpu.set_value(14'h 0010，8'b0010_1000，8'h0000_1000);
…;            //其他事务
#1000 cpu.set_value(14'h 0001，8'b0011_1100，8'h0010_1100);

… ;           //其他事务
end
endmodule
```

cpu_interface 的信号可以直接连接到 DUV 上。Initial 中，有两次对寄存器的设置，一次在开始后的 300 时间单位，通过调用 cpu_interface 中的 set_value 将寄存器 14'h0010 的比特 3 位设置成 1，比特 5 设置成 0；第二次调用 set_value 设置寄存器 14'h0001 的比特 2，3，4，5 为 1101。

方法二：将测试程序中所有与 DUV 有关的模块分离出来，形成另外一个层次，它在测试文件和 BFM 中间。

```
module harness;
…
cpu_interface   cpu_inst (.A(A),
                          .D(D),
                          .ALE(ALE),
                          .WRB(WRB),
                          .RDB(RDB),
                          .CSB(CSB));
DUV     DUV(…;            //其他信号
        .A(A),
        .D(D),
        .ALE(ALE),
        .WRB(WRB),
        .RDB(RDB),
        .CSB(CSB),
        …       );      //其他信号

    …
endmodule
//在测试程序中调用 haress 模块
module testcase;
  …
harness th();
initial
begin
#300   th.cpu.set_value(14'h0010，8'b0010_1000，8'h0000_1000);
…;                  //其他事务
#1000 th.cpu.set_value(14'h0001，8'b0011_1100，8'h0010_1100);

…;                          //其他事务
end

endmodule
```

由于 harness 没有和其他模块连接的任何信号，因此它们可以不用实例化。它们可以形成附加的仿真顶层模块，与 testbench 和 DUV 同时运行，可以通过绝对名称访问 harness 中的任务和函数。在 Modelsim 中，使用下面的命令完成上述功能：

```
vlog testbench.v harness.v cpu.v dut.v
vsim testcase harness cpu duv
```

从上面的例子中，我们可以看出引入 BFM 的优点在于：

(1) 有利于验证重用：验证程序可以直接应用到功能相同，但不具有相同接口的设计中，而事务处理程序则可以重用在具有相同接口的不同设计中。

(2) 由于事务处理程序封装了接口的实现细节，因此可以极大地提高验证程序的开发效率。

(3) 提高了仿真代码的可读性。

**4．编写具有层次结构的仿真代码**

从上面的 CPU 接口的仿真代码例子中，我们可以看出，其中包含了一定的层次结构。验证工程师在最顶层，只需要写出特定的完成某项功能验证的事务序列，而这些序列中的事务去调用不同的事务处理程序，这些不同的事务处理程序又可以去调用更低层的事务处理程序，以产生 CPU 接口所需要的物理信号。这种层次化结构，可以为验证工程师提供一个良好的可操作环境，使他们更关注于 DUV 的验证而不是注意如何产生 DUV 接口信号。

可以将验证代码的构成划分成如图 4.16 所示的四个层次的结构，低层为它的高层提供一定的服务，而高层事务通过 BFM 将所处理的事务传递给低层。

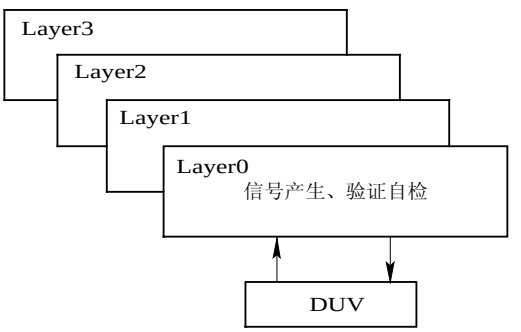



图 4.16　测试环境的逻辑层次结构

(1) 第 0 层用于信号的产生和验证自检查。0 层直接连接到 DUV 上，这层根据更高层所处理的事务，产生各类 DUV 所需要的正常数据和异常数据，同时 DUV 的输出也直接连接到本层，以便自动检查输出激励是否和预期值一致。在这层中，可以按功能划分模块以提高验证代码的可重用性。

(2) 第 1 层为低层 BFM 层。这层的 BFM 将调用第 0 层的事务形成 DUV 所需要的信息，同时为其上层提供功能调用。

(3) 第 2 层为高层 BFM 层。这层的 BFM 利用第 1 层和第 0 层提供的低级 BFM，构造更高级的 BFM。第 1 层和第 2 层之间没有明确的界限，可根据不同的设计项目确定。

(4) 第 3 层面向应用层。这层由验证工程师们实现，根据被验证设计的验证方案，调用第 1 层和第 2 层提供的 BFM 以产生不同约束的验证序列。

在上面的层次结构中，第 0 层由熟悉被验证芯片时序的工程师完成。因为第 0 层的 BFM 直接产生 DUV 所需要的信号，而这些信号的具体时序只有设计工程师清楚。而其他层 BFM 可由验证工程师完成，验证工程师可以不必关心仿真信号时序，也不必关心第 0 层的实现细节。有了这样一个层次结构，验证工程师通过图 4-16 所示的 4 层结构产生 DUV 所需要的激励或进行自检。

### 5．编写具有自检查功能的仿真程序

设计的有效性必须通过设计对激励响应的结果得以体现，有几种方式可以检查设计响应是否正确。

(1) 方法一：通过人工观测 DUV 输出波形的结果是否正确是常用的一种方法，这种方法简单、直观。在出错时，通过人工干预，可以立即停止仿真。但是，其缺点是工作量大，易出错。

(2) 方法二：通过日志的方式，将一些结果输出到文件中，在仿真结束后，分析日志文件的结果。这种方式的主要缺点是结果分析需要等到仿真结束。

(3) 方法三：自检查方式，它是在仿真过程中，自动将预期的结果和仿真输出的结果进行比较，一旦出错，仿真自动停止。这种方式的优点是能在仿真的过程中，并行地自动检查设计的正确性。

理想情况下，预期的结果通过参考模型体现，图 4.17 给出了参考模型、激励、DUV 之间的关系。通常，建立一个功能非常完善的参考模型比较困难，需要大量的时间，对于功能比较复杂的设计，纯用 Verilog 或 VHDL 语言比较难以实现，可以使用一些专用的验证语言，如 Syopsys 公司的 Vera、VCS 或 Cadence 公司的 testbuilder 等，也可以使用 C/C++等高级语言构造参考模型。

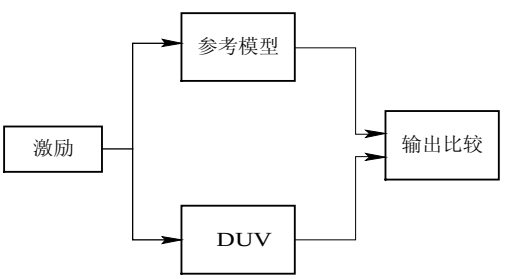



图 4.17　参考模型、激励和 DUV 之间的关系

然而，对于一些比较特殊的设计，如果你能知道特定激励下的期望响应，那么就可以采用一些特殊的方式来实现自检查的设计。例如，对于 SDH 设计和以太网等与输入激励有密切关系的输出，可以通过 FIFO 对输入进行缓存，在特定的时刻将输入和输出进行比较，实现自检查功能。

### 6．编写可重用的验证代码

大规模 FPGA/ASIC 设计一般由多个层次构成，设计人员必须对各个层次上的子模块逐一验证，然后将这些验证过的模块连接在一起形成高层规模较大的设计。为这些不同的子模块开发不同的验证环境实际上需要花费大量的时间和精力，验证重用是解决这一问题的有效方法，设计人员根据大量的可重用验证模块构造出不同层次模块的验证环境。

验证可重用有两种形式，一种是同一个芯片设计中验证重用，另一种是不同芯片设计之间的重用。在同一个芯片设计的重用是指在验证的不同周期或设计的不同阶段验证代码的重用。好的验证代码可以在子模块级和系统级验证时均可重用。不同芯片设计验证重用是指验证代码可以用于同一芯片的更新换代上，或用于一个包含许多标准设计模块的芯片或与以前设计有相同性的新设计中。

一般而言，需要重用的模块越多，所考虑的事项和投入的精力就越多。需要在验证重用所取得的效果和投入重用的资源之间做一些平衡。资源控制，项目领导和安排以及重用所取得的效果都影响开发和对源代码的改变策略。

为了优化重用，验证代码应该与公用的总线和 I/O 端口的功能块一致。

通常，仿真程序被划分成两个主要的部分：可重用的验证代码与专用的验证代码。两者之间的关系如图 4.18 所示。图中的 VIP 是指经过验证的仿真模块，可以重用。

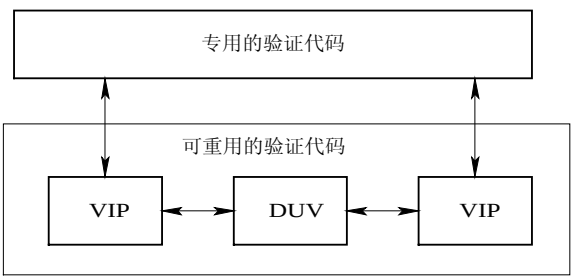



图 4.18　验证程序的构成

将验证程序分成两个部分，验证工程师可以利用 VIP 快速地构造适应于本项目的验证程序。如果本项目的 DUV 所有信号都已产生，那么，验证工程师只需要关注 DUV 的验证，专注地编写专用的验证代码即可。

VIP 的构造通常是有层次结构的，一旦底层的 VIP 被验证后，这些 VIP 可以被高层 VIP 所共享，也可以为其他的仿真用例或项目所用。

## 4.3　基于断言的验证

实际上在软件设计中，断言已经得到了广泛的应用，它可以帮助软件工程师在软件开发及测试过程中更早更快地发现并定位出软件中可能存在的错误，是一种非常高效的调试方法。现在这种方法被引入在集成电路设计的验证中，成为一种非常有效的调试电路的方法。

基于断言的验证是一种有效的白盒验证法，它是在 RTL 设计的源程序中监视系统的关键行为，特别是在某些特殊情况下的行为。通过断言的方法，可以增加观测点，在仿真过程中及时发现设计错误。

为了说明基于断言的验证方法，首先介绍一些断言中用到的概念。

(1) 特性(Property)：用于刻划设计特性的一些通用的行为属性。特性可以是高阶属性，如进入或退出网络的数据包的一些特性，也可以是低阶属性，如 FIFO 的空满等。

(2) 事件(Event)：事件在验证过程，一种希望出现的行为。例如，如果内存访问过程出错，我们希望能有一个合适的错误处理函数。作为验证的一部分，观测事件的目的是为了保证验证的完整性。确定事件出现的数目可以得到一些特殊的极端情形(Corner Case)量化信息，并指示其他已验证特性。事件的统计信息形成了功能覆盖度量(Function Coverage Metric)

(3) 断言(Assertion)：断言是设计中希望特定性质为真的描述语句，断言的目的是捕获不希望在设计中出现的行为。断言是用于监控或检查施加在设计上的一些规则和假设的一种机制。

用户可以用各种硬件描述语言如 Verilog、VHDL 或 SystemC 等形成断言以监视设计在仿真过程中的行为，也可以利用已有的断言库，将断言直接加入到设计中。此外，一些专用的硬件特性描述的语言(Property Specification Language)和断言语言已经称为工业标准，并开始推向市场。例如 IBM 公司开发的 Suger 语言已被 Accellera 组织接受，成为 PSL 工业的标准，而 Synopsys 的 OVA 则是另一个经过实际设计验证的断言描述语言。

断言可以有多种实现方法，其中最常用也最简单的方法是所谓的叙述性的实现方法，即在设计结构中描述断言，断言和设计中的其他结构一起并发地计算。限于书的篇幅，我们简单介绍用 Verilog 语言实现叙述性断言的方法。

叙述性断言实际上是一些代码，这些代码中一般需要包含三部分：一是断言的条件，二是报告信息，三是错误的严重程度以及相关的处理。

例如，不变性断言：

assert_always [#(severity_level，options，msg)] inst_name (clk，reset_n，test_expr)

其中：assert_always 为断言的名称；inst_name 为断言的实例化名称；test_expr 为断言的条件，断言在每个时钟的上升沿检查表达式 test_expr，如果 test_expr 为假，也就是在设计中检测到错误，则激活断言；[#(severity_level，options，msg)]为断言的参数，severity_level 表示错误的严重等级，根据不同的错误等级，进行相应的处理。另外一个是消息 msg，用于表示某个性质不成立时要显示的信息。如果在模拟的过程中，违背了设定的性质，那么就会触发监视器。另外一个是可选的信息 options。

下面是 Assert_always 断言的代码。

```
module assert_always (ck，reset_n，test_expr);
input ck，reset_n，test_expr;

parameter severity_level = 0;
parameter msg="ASSERT ALWAYS VIOLATION";

 'ifdef ASSERT_ON
 integer error_count;
 initial error_count = 0;

 always @(posedge ck) begin
   'ifdef ASSERT_GLOBAL_RESET
     if ('ASSERT_GLOBAL_RESET != 1'b0) begin
```

```
        'else
          if (reset_n != 1'b0) begin
        'endif
          if (test_expr != 1'b1) begin
            error_count = error_count + 1;
          'ifdef ASSERT_MAX_REPORT_ERROR
              if (error_count <= 'ASSERT_MAX_REPORT_ERROR)
          'endif
            $display("%s : severity %0d : time %0t : %m"，msg， severity_level， $time);
          if (severity_level == 0) $finish;
          end
        end
      end // always
      'endif
    endmodule // assert_always
```

上述的断言是用于检测某个表达式是否永远为真，如果 test_expr 表达式不为真，那么错误计数器计数不为真的次数，如果错误计算器的值小于用户定义的错误次数，那么显示错误信息。如果定义错误等级为 0，则退出仿真。

从上面的实现，我们可以看到，一个断言实际上就是一段 Verilog 代码，用模块的形式将其封装起来。因此，叙述性断言的用法非常简单，直接采用实例化的形式把断言嵌入在设计中就可以了，当测试条件不成立的时候，触发该断言。【例 4.13】是说明如何使用 assert_always 断言的。该例子是一个模 9 的计数器，如果计数器的值不在 0～9 之间，那么则启动该断言中的监控机制，并报告错误信息。如果错误等级定义在 0，那么在出现错误后仿真结束。

【例 4.13】 模 9 计数器中使用 always 断言。

```
module counter_0_to_9(reset_n,clk);
input reset_n,clk;
reg [3:0] count;
always @(posedge clk)
begin
    if   (reset_n == 0 || count >= 9) count = 1'b0;
    else count = count + 1;
end
assert_always #(0,0,"error: count not within 0 and 9")       //always 断言
        valid_count (clk，reset_n，(count >= 4'b0000) && (count <= 4'b1001));
endmodule
```

从上面的例子我们可以看到，用 Verilog 叙述方法实现的断言可以直接嵌入到设计的源代码中，说明静态和时序断言，提供统一的信息报告机制。

利用断言的优点有：

(1) 可以节约仿真时间，在仿真的过程中动态地检查断言可以及时发现设计中不希望的行为，一旦出现了仿真错误，可以立即停止仿真。

(2) 增加了设计的可观察性。

(3) 减少设计的错误定位时间，可以准确而快速定位设计错误。

(4) 提供了一种捕获并确认接口约束的手段。

Accellera(www.accellera.com)推出采用了断言思想的验证库 OVL(Open Verification Library)，该库中用 HDL 语言(VHDL 和 Verilog)定义和实现了一些非常常用的属性声明。这个库资源是免费的，设计人员可以在设计里面直接使用这些属性声明来检测设计是否遵从了相应的设计属性，也可以对其进行修改用于不同的设计中，本节的例子就源于 OVL 库。

【例 4.14】 利用 OVL 中的 assert_never 监视 FIFO 的溢出情况。

```
module guarded_fifo (clk，reset_n，read，write，data_in，data_out);
input clk，reset_n，read，write;
input [15:0] data_in;
output [15:0] data_out;
  wire fifo_full，fifo_empty;
  fifo fifo (clk，reset_n，read，write，data_in，data_out，fifo_full，fifo_empty);
  assert_never #(0，0，“Fifo overflow”) fifo_overflow (clk，reset_n，fifo_full && write);
  assert_never #(0，0，“Fifo underflow”) fifo_underflow (clk，reset_n，fifo_empty && read);
endmodule
```

# 4.4 时 序 验 证

时序验证的目的是为了确认物理实现的电路时序是否满足时序规范要求。时序规范用于约束一个电路的接口信号和周围环境之间的时序关系或约束电路内部的延时。时序验证的方法分为动态验证和静态验证两种，本节简单地介绍静态时序分析中涉及到的一些基本概念和动态时序验证中用到的一些系统函数，静态时序分析涉及到很多算法，有兴趣的读者可以参阅相关文献。在后续章节中，详细介绍 Altera 静态时序分析工具的使用和动态时序仿真的方法。

## 4.4.1 静态时序分析概述

### 1. 静态时序分析与动态时序分析

时序验证用两种方法实现：一是动态时序分析，即根据电路中提取的延时参数，通过仿真软件动态地仿真电路以验证时序是否满足要求。二是静态时序分析，即通过分析设计中所有可能的信号路径以确定时序约束是否满足时序规范。动态时序分析的时序确认通过仿真实现，分析的结果完全依赖于验证工程师所提供的激励。不同激励分析的路径不同，也许有些路径(比如关键路径)不能覆盖到，当设计规模很大时，动态分析所需要的时间、占

用的资源也变得越来越大。与动态时序分析相比较，静态时序分析根据一定的模型从网表中创建无向图，计算路径延迟的总和，如果所有的路径都满足时序约束和规范，那么认为电路设计满足时序约束规范。静态时序分析的方法不依赖于激励，且可以穷尽所有路径，运行速度很快，占用内存很少。它完全克服了动态时序验证的缺陷，适合大规模的电路设计验证。因此，静态时序分析不需要激励便可完成。对于同步设计电路，可以借助于静态时序分析工具完成时序验证的任务。

静态时序分析主要完成的功能包括：

(1) 建立时间/保持时间违规检查。

(2) 恢复时间/移除时间检查(包括反向建立/保持)。

(3) 检查最小和最大跳变。

(4) 检查时钟脉冲宽度和时钟畸变。

(5) 门级时钟的瞬时脉冲检测。

(6) 检查总线竞争与总线悬浮错误等。

(7) 对关键路径、约束性冲突、异步时钟域、组合环、假路径和某些瓶颈逻辑进行识别与分类。

有不少的 EDA 厂家都提供静态时序分析的工具，Synopsys 公司的 Primetime 和 Mentor Graphic 公司的 SST Velocity 是比较有影响的用于全定制 IC 时序分析的工具。FPGA 供应商如 Altera，Xilinx 和 Actel 也在其集成环境中嵌入了静态时序分析工具。

### 2. 时序路径

一般情况，在一个电路设计中存在四种基本时序路径(见图 4.19)：

(1) 路径 1(Path1)：从输入管脚到 D 触发器的输入。

(2) 路径 2(Path2)：从 D 触发器的输入到下一个 D 触发器的输入。

(3) 路径 3(Path3)：从 D 触发器的输入到输出管脚。

(4) 路径 4(Path4)：从输入到输出。

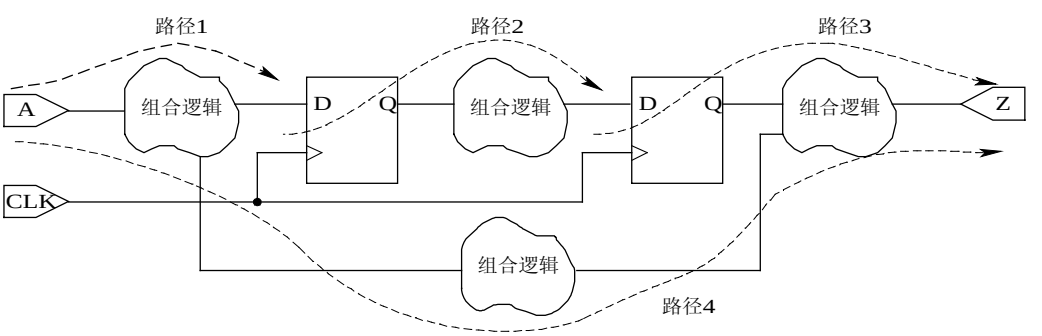



图 4.19　数据路径

组合逻辑可能有多条路径，静态时序分析工具用最长的路径计算最大延时，用最短的路径计算最小延时，如图 4.20 所示。

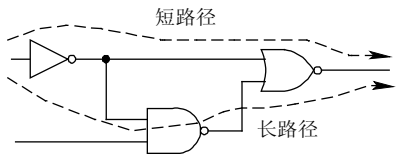



图 4.20　路径延时

除了上面的四类路径，还可能有如下的路径(见图 4.21)：

(1) 时钟路径(Clock Path)：从一个时钟的输入通过一个或多个缓存器或反相器到达一个存储元件的时钟管脚的路径。

(2) 门控时钟路径(Clock-gating)：为检查建立、保持时间而设计门控时钟路径(从输入端口到门控时钟元件)。

(3) 异步路径：为检查恢复时钟和扇出而设置异步路径(从输入端口到一个存储元件的异步复位、置位端)。

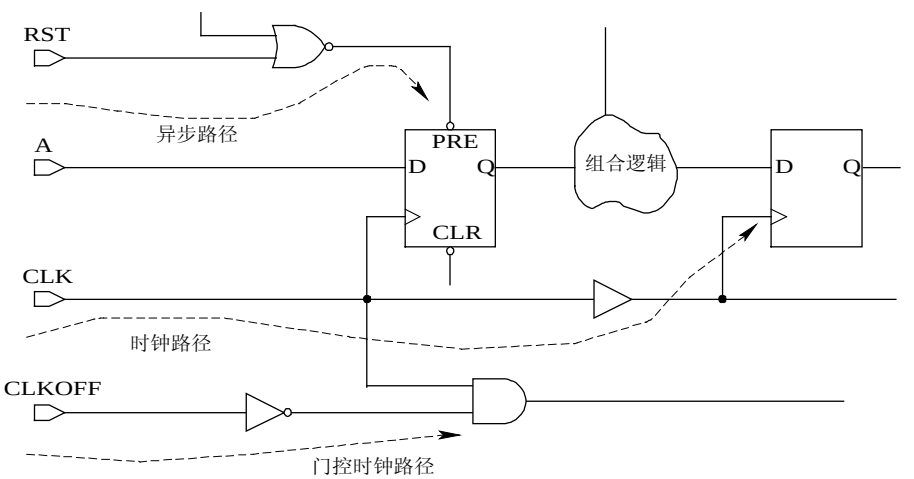



图 4.21　其他路径

如果一个电路具有多个独立的时钟，即不是由某个时钟经过分频或门控后得到的时钟，那么静态时序分析工具将每个不同的时钟管辖的路径划分到不同的组中进行时序分析，而对于那些终点不是存储元件输入的路径划分到缺省的组中进行分析。如图 4.22 所示。图中，路径 1 被划分到 CLK1 组中，而路径 2 被划分到 CLK2 所管辖的组中，而路径 3 和 4 则在缺省的组中。

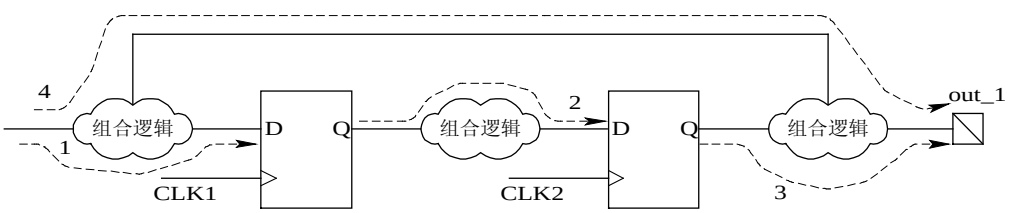



图 4.22　多时钟路径

在时序分析中，禁止组合环的存在，要求所有的反馈路径都可以在时钟边界被打断。静态分析工具通过反向跟踪路径终点到起点上升沿或下降沿的跳变来计算传输延时，并累加路径上的传输延时。

一条路径的延时等于在该条路径上所有元件和连线延时之和，如图 4.23 所示。元件延时是一个门的输入到输出之间的延时。连线延时是时序分析路径上一个元件的输出到下一个元件输入之间的路径延时。两个元件的连线之间的寄生电容，线电阻和驱动线上的有限驱动强度等都会引起延时。

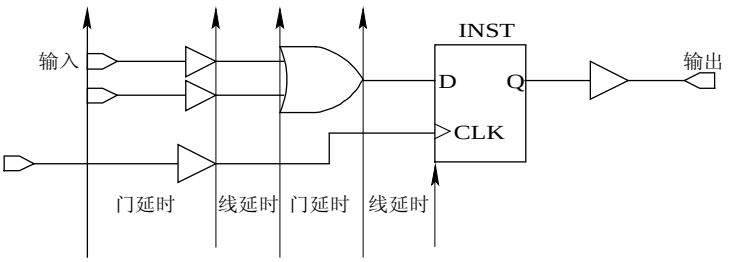



图 4.23　数据路径的延时

图 4.23 给出从输入端到寄存器 INST 之间的路径延时计算方法，这条路径包含了两个门延时和两个线延时。

一条路径的最长延时是由该路径的组合电路、存储元件、路径上门的扇出负载、信号之间的互连线负载、时钟的歪斜率、时钟抖动和信号的压摆率等所决定的(相关的概念将在后面介绍)。

## 4.4.2　静态时序分析中的基本概念

### 1. 扇入和扇出

一个逻辑门的扇入是指连接到该门输入的数目，一个逻辑门的扇出是指连接到该门输出的负载门的数目，如图 4.24 所示。扇出越多，延时越大。

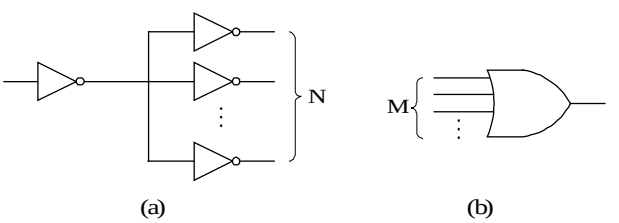



图 4.24　扇入和扇出

(a) 扇出；(b) 扇入

### 2．压摆率(Slew Rate)

电压变化的速度，工程上一般把压摆率定义为

$$\frac{dV}{dt}=\frac{(V_{OH}-V_{OL})\times 80\%}{t_R\,(t_F)}$$

式中，$V_{OH}$为输出电平为逻辑 1 时的最大输出电压；$V_{OL}$为输出电平为逻辑 0 时的最小输出电压；上升时间($t_R$)为输出电压从 $0.1V_{CC}$上升到 $0.9V_{CC}$所需要的时间(见图 4.25)；下降时间($t_F$)为输出电压从 $0.9V_{CC}$下降到 $0.1V_{CC}$所需要的时间；延时时间($t_{PD}$)为输出电压从 0 上升到 $0.5V_{CC}$所需要的时间(见图 4.25)。

信号的压摆率对门的延时有影响。压摆率越大，延时越小，压摆率越小，延时越大。

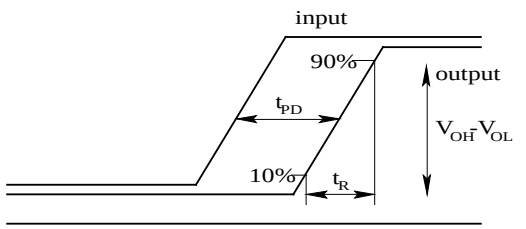



图 4.25　时序说明图

### 3．时钟歪斜(Clock Skew)

时钟在经过时钟路径后，到达存储元件的时间存在差别，这种时间差称为时钟歪斜。由于在时钟网络上，各条时钟路径的长度不一样，因此会出现时钟歪斜。时序上相邻的寄存器在时钟歪斜较大的电路中，可能在同一时钟沿上出现时间违规或不能正确保存数据的现象，所谓的时序相邻寄存器是指两个寄存器之间只有组合逻辑和它们之间的互连线，如图 4.26 所示。

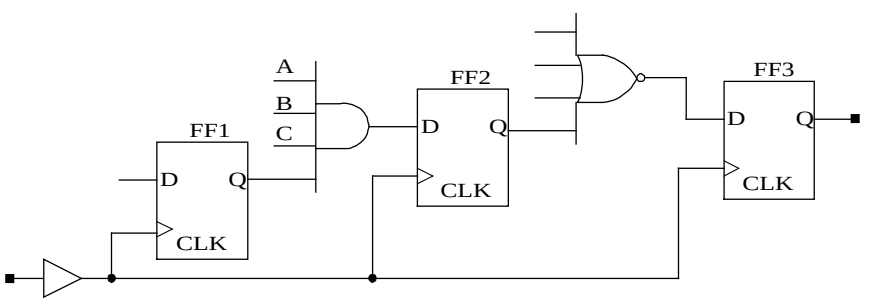



图 4.26　时序相邻的寄存器

因此，在图 4.26 中只有 FF1 和 FF2，FF2 和 FF3 之间的时钟歪斜才有意义，而 FF1 和 FF3 之间的时钟歪斜是没有意义的。给定时序相邻的两个寄存器 Ri 和 Rj 以及一个时钟网络，Ri 和 Rj 之间时钟歪斜定义为：

$$tskew(i，j)＝t_{ci}－t_{cj}$$

其中 $t_{ci}$和 $t_{cj}$分别表示从源时钟到达寄存器 Ri 和 Rj 的时钟延时。

### 4．寄存器的建立和保持时间

寄存器的建立和保持时间的验证是静态时序分析最重要的一个功能。所谓的建立时间是指一个数据信号在有效的时钟沿到达前必须稳定的最小时间，如图 4.27 和 4.28 所示。

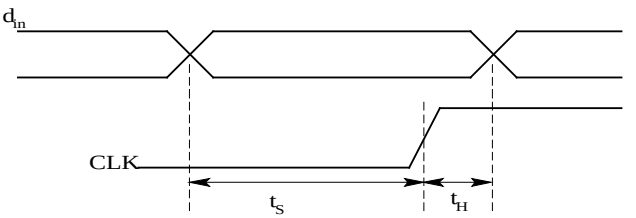



图 4.27　建立时间和保持时间

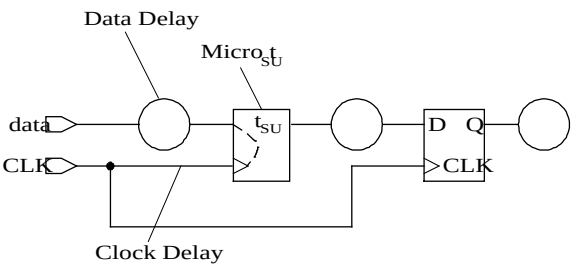



图 4.28　电路示意图

数据的建立时间计算公式为：

数据的建立时间＝最长的数据延时－最短的时钟延时＋Micro $t_{SU}$

其中，Micro $t_{SU}$ 是 D 触发器内部固有的要求建立时间，不受外部信号的影响。

保持时间是指一个数据信号在有效时钟沿结束后必须稳定的最短时间，如图 4.27 和 4.29 所示。保持时间的计算公式为：

保持时间＝最长的时钟延时－最短的数据延时＋Micro $t_H$

式中，$t_H$ 为寄存器内部要求的保持时间。

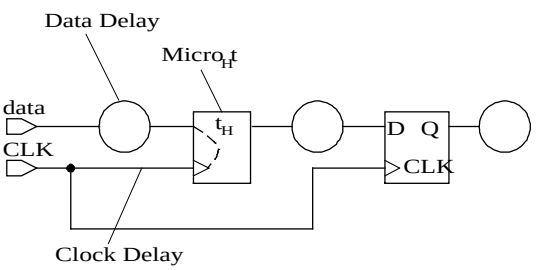



图 4.29　保持时间电路示意图

建立和保持时间都是相对于某个时钟沿而言的，例如，相对于时钟的上升沿。如果系统中寄存器元件的建立时间或保持时间存在违规，那么系统将不能正常工作。

### 5．时钟到输出的延时

时钟到输出的延时是指信号通过寄存器传播到输出管脚后，在输出管脚上获得稳定有效的数据所需要的最大时间，如图 4.30 所示。延时计算公式为

延时＝最长的时钟延时＋最长的数据延时＋Micro $t_{CO}$

其中，Micro $t_{CO}$ 为 D 触发器内部要求的时钟到输出的延时。

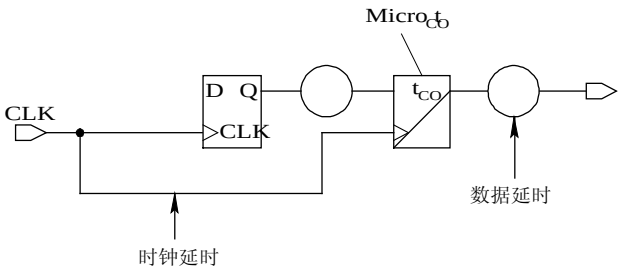



图 4.30　时钟到输出的延时

### 6. 输入延时与输出延时

一个 FPGA 设计总是和其他外围电路一起工作的，输入延时表示从 FPGA 外部的寄存器到 FPGA 一个特定输入管脚的延时，等于外部寄存器的时钟到其输出延时加实际 PCB 板的延时如图 4.31 所示。输出延时表示从 FPGA 设计一个管脚到外部寄存器的延时，这个值是外部寄存器的建立时间加实际 PCB 板的延时。

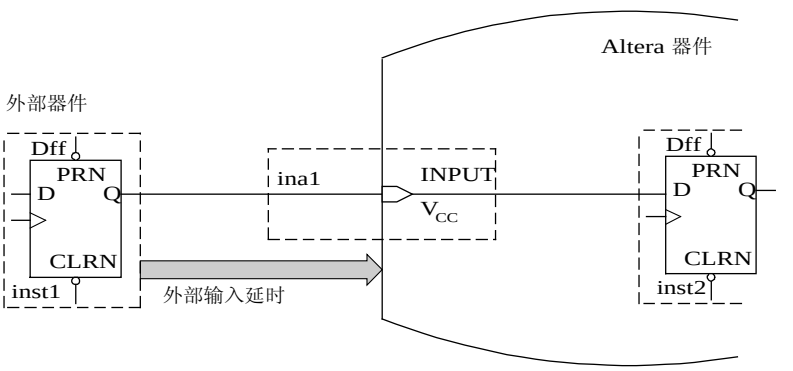



图 4.31　输入延时示意图

### 7. 恢复(Recovery)数据/撤销(Removal)数据时间

在时钟有效沿跳变前，异步控制输入信号(如 Reset、Clear)必须稳定的最小时间称为恢复时间。在时钟有效沿跳变后，异步控制输入信号(如 Reset、Clear)必须稳定的最小时间称为撤销时间。如图 4.32 所示。如果时钟有效沿和异步复位信号的结束之间的时间太短，寄存器无法判断是继续保持复位值，还是该由时钟沿打入新的数据，从而导致寄存器的内容不确定。

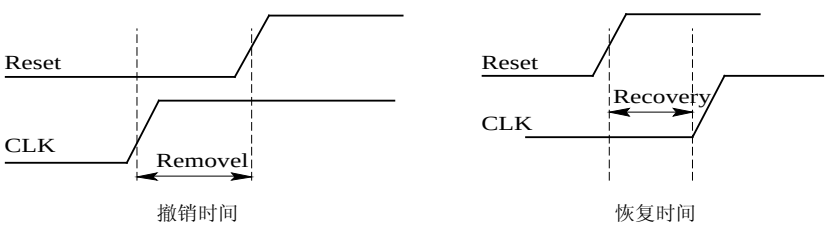



图 4.32　恢复时间和撤销时间

### 8．时钟脉冲宽度

时钟脉冲宽度定义为一个时钟周期的高电平或低电平的最小宽度。如果脉冲宽度过小，那么存储元件将不能正确保存数据，如图 4.33 所示。

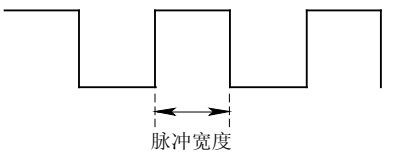



图 4.33　脉冲宽度

### 9．最大时钟频率

最大时钟频率是指在不违背内部要求的建立和保持时间前提下，电路工作的最快速度。频率和最大的时钟周期计算公式如下(见图 4.34)：

$$\text{频率}=\frac{1}{\text{最大的时钟周期}}$$

最大的时钟周期＝时钟到输出的时间＋数据延时＋建立时间－时钟的歪斜

$$=t_{CO}+B+t_{SU}-(E-C)$$

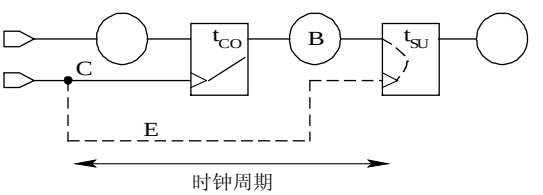



图 4.34　时钟周期

### 10．裕度(Slack)

裕度是时序要求与实际时序之间的差值，它反映了时序是否满足要求。正的裕度表示设计满足时序要求，而负的裕度表示设计不满足时序要求，图 4.35 是裕度的一种示意图。裕度的计算公式如下：

裕度＝要求的时间－实际的时间

＝裕度时钟周期(slack clock period)－数据延时(data delay)－$t_{CO}$－$t_{SU}$

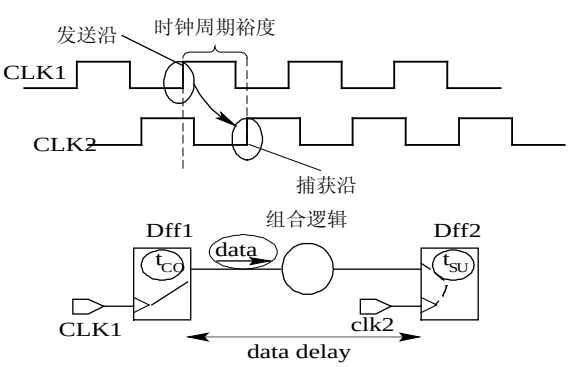



图 4.35　裕度

### 4.4.3 假路径和多周期路径

#### 1．假路径

静态时序分析工具对于一些我们称之为假路径的路径不能正确分析。那么什么是假路径？回答这个问题之前，我们首先了解一些相关的概念。

1) 逻辑门的控制值和非控制值

如果一个逻辑门的输出值只取决于一个输入值，这个输入值就是该逻辑门的控制值，而其他值则为非控制值。例如，与门的控制值是 0，非控制值是 1，而或门的控制值是 1，0 为非控制值，如图 4.36 所示。

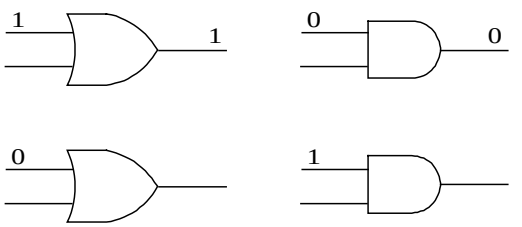



图 4.36　逻辑门的控制值和非控制值

2) 路径敏化

(1) 静态可敏化：对于一条路径，如果存在一组输入向量使得该路径上逻辑门的输入都被设置成非控制值，则这条路径称为是可敏化路径。

例如图 4.37 中，假设不考虑互连线的延时，每个门的延时只有 1 个时间单位，虚线标出的路径(a−c−d−y−z)是不可静态敏化的，这条路径的长度为 4。从图中我们可以看出，当 b＝0 时，e＝1，输出 z 为 1。当 b＝1 时，e＝0，y＝0，输出 z 的值为 0。也就是说从输入 a 到 z 不能传递任何信号的跳变，这条路径不可敏化。由于这条路径不可敏化，因此，报告的最长路径为 3(a−c−d−y)。

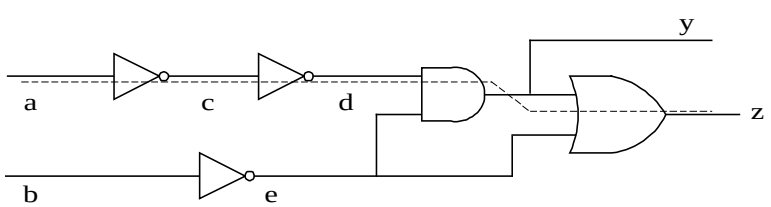



图 4.37　不可静态敏化路径

如果一条路径是不可静态敏化的，那么这条路径对于延时分析是没有贡献的，把这种路径称为是假路径。

(2) 动态可敏化：如果在一条路径上，在不同的时间可以找到一组输入，使得这条路径可以传输信号的跳变，这种敏化方式称为动态可敏化。动态可敏化路径见图 4.38 中的虚线路径。

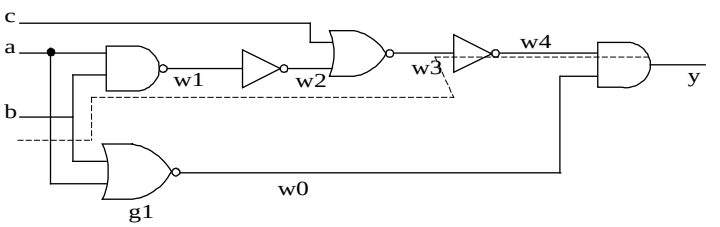



图 4.38　动态不可敏化路径

敏化路径 b−w1−w2−w3−w4−y 要求 a＝1(非控制值)，在这种情况下，a＝1 变成了门 g1 的控制值，g1 的输出为 0，导致 y 的输出为 0，这样 b−w1−w2−w3−w4−y 这条路径是不可静态敏化的。但是，如果先让 b 取 1 驱动 w1，然后在切换到 0，这样就可以使得 b−w1−w2−w3−w4−y 传递信号的跳变，其时序图如图 4.39 所示。

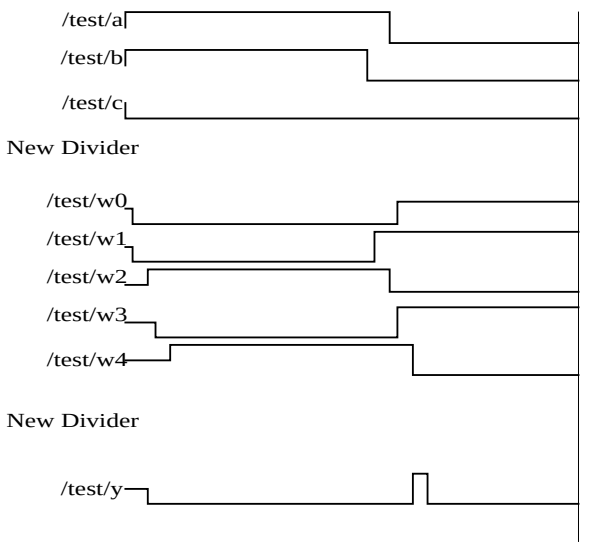



图 4.39　图 4.38 的时序图

### 2．多周期路径

图 4.40 说明了多周期路径的概念，其上半部分路径包含了多于两级的寄存器，因此需要一个以上周期输入才会在输出有效。图的下半部分说明另外一种多周期路径，它只包含了两级寄存器，但是，这个路径上组合电路的延时比较大，要求多于一个周期的时间完成组合电路的计算，以便保证输出的有效性。

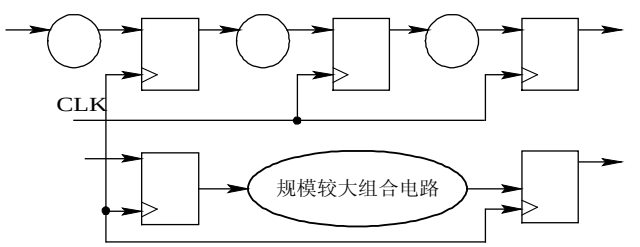



图 4.40　多周期路径

如果用户设计中包含了这些类型的多周期路径或者假路径，应该在时序检查时，进行设置，这样这些路径将不被布局布线工具所限制，时序分析工具将不报告这些路径违规。

静态时序分析由FPGA开发系统中的静态时序分析工具自动完成，用户根据电路的特点，向静态时序分析工具提交时序约束文件，静态时序分析工具计算所有的可能的路径，检查电路设计是否满足时序要求，并给出详细的分析报告。用户根据报告，对电路进行修改。

### 4.4.4 时序验证中的系统任务

在Verilog语言中提供了一些内建的系统函数，用于在动态仿真的时候检查设计的时序。这些任务可以自动监控仿真的行为，监测时序并报告时序违规。

(1) 时序检查：如果一个D触发器的建立时间和保持时间不合乎要求，那么D触发器将不能正确工作。在动态仿真过程中，可以通过setup和hold两个函数进行动态的检测。

① $setup(data_event，reference_event，limit)：用于检查data_event在与reference_event相关的时间limit内是否出现建立违规，其中参数limit约束了建立时间的长短。例如，$setup(signal，posedge clk，5)检查在时钟上升沿到达前的五个时间单位，signal是否出现建立时间违规。

② $hold(data_event，reference_event，limit)：用于检查data_event在与reference_event相关的时间limit内是否出现保持违规，其中参数limit约束了保持时间的长短。保持时间违规主要由于数据路径到D触发器的时间太短，在数据路径上开始点D触发器数据的变化传递到结束点D触发器的速度太快，结束点D触发器的数据输出还没有稳定，新的一次数据变换又到了。

(2) 脉冲宽度检查：在时序器件中必须限制最小的脉冲宽度，D触发器的时钟必须保证一定的宽度，以保证D触发器的正常工作。用$width(reference_event，limit)来检测时钟的宽度，limit限制时钟的宽度。例如，$width(posedge clock，10)用于检测时钟clock的上升沿和下降沿之间的脉冲宽度是否小于10。这个系统函数可以用于检测时钟信号上可能出现的毛刺和时钟信号的降级。

(3) 时钟歪斜检查：时钟歪斜太大可能会影响到系统工作，如果一个时钟信号没有进入到时钟网络，往往会产生较大的时钟歪斜。$skew (reference_event，data_event，limit)在仿真过程中检查reference_event和data_event之间的距离是否大于limit，如大于则报告时钟歪斜的错误。例如，$skew(posedge clk1，posedge clk2，3)，检查两个时钟上升沿之间的间隔是否大于3。

(4) 时钟周期检查：系统函数$period(reference_event，limit)用于检查设计的工作周期，该函数连续监测reference_event时间，如果两个连续的reference_event之间的间隔小于limit，则报告错误。

(5) 恢复时间检查：系统函数$recovery(reference_event，data_event，limit)用于检查异步复位(reference_event)无效后，下一个有效时钟沿(data_event)达到的最小时间(limit)，如果这个时间小于limit，则报告错误。例如，$recovery(posedge reset，posedge clock，4)检查在复位信号reset无效后，第一个时钟上升沿和reset无效之间的时间是否小于4。

时序检查一般与工艺库有关，内建时序检查函数都在工艺库的仿真模型中，工艺库中的每个元件都有一个仿真模型。

# 第 5 章 ModelSim 工具介绍

第 4 章中我们介绍了设计验证的基本概念、验证流程和方法，本章将介绍 Mentor 公司的子公司 Model Tech 开发的 ModelSim 仿真工具的使用方法。ModelSim 支持 VHDL、Verilog 以及混合语言设计的仿真，既可应用于设计的前仿真，也可以在 FPGA 器件库的支持下进行时序仿真。

在 ModelSim 中进行仿真有两种工作方式:

(1) 工程仿真流程——该方式使用工程建立仿真环境，简单、易学、易用，可以方便地借助 GUI 进行对话式操作，其许多工作交由工程工具完成。

(2) 基本仿真流程——直接建立仿真库，并以仿真库为基础直接进行仿真，此种工作方式便于生成 DO 文件脚本内容，便于批命令自动执行。

本章主要以“工程仿真流程”为主讨论 ModelSim 的使用，并对 DO 文件及自动批执行作以简单介绍。

## 5.1 ModelSim 概述

### 5.1.1 基本仿真流程

ModelSim 仿真的基本流程如下：

(1) 创建一个工作库。在 ModelSim 中，所有的以 VHDL/Verilog 或混编形式存在的设计必须被编译到一个库中。ModelSim 默认创建一个名称为“work”的工作库，启动一个新仿真。名称为“work”的库是编译器编译设计单元默认的目标单元。

(2) 编译设计文件。工作库创建完成后，将各设计单元编译到其内。ModelSim 库格式文件对所有的可支持平台都适用，所以在不同的平台上进行仿真时，可以直接使用该库来仿真设计，不必再重新编译设计。

(3) 运行仿真。设计编译完成后，我们就可以针对顶层模块(Verilog)、结构、实体加载仿真器。当设计装载成功，仿真时间设置到 0 位置时，我们可以输入一个运行命令开始仿真。

(4) 调试结果。如果仿真结果不是预期的，则可以使用 ModelSim 的调试工具去跟踪问题的缘由。

### 5.1.2 工程仿真流程

ModelSim 中所谓的“工程”包含了设计或验证所需的全部实体。在一个最小化的工程中应该包含一个工程根目录下，一个工作库和一个工程环境配置文件(.mpf 文件)。该 .mpf 文件位于工程根目录下，用于保存编译开关设置、编译顺序和编译映射等信息，.mpf 文件定义了工作环境所需的完备文件集合，其更新和存储只能在工程中完成。此外，一个工程还包含如下内容：

(1) HDL 源设计文件和源设计文件参考。

(2) 其他的工程文档文件。

(3) 本地库文件。

(4) 通用库参考文件。

(5) 仿真配置文件。

(6) 应用文件夹等。

工程仿真流程具有以下优点：

(1) 与 ModelSim 交互更简便，用户不必了解复杂的编译开关和库映射等设置。

(2) 编译顺序在工程中维护。

(3) 不必重新建立已有编译开关和设置。

(4) 允许用户共享库，不必将文件拷贝到本地目录，用户可以建立存储于其他目录或本地目录的源文件。

(5) 允许用户分别对多个文件改变参量。

(6) 每次打开工程时，重新加载 .ini 环境变量设置。

ModelSim 工程仿真流程一般包括以下四个步骤：

(1) 创建一个工程，即创建 .mpf 文件和工作库。

(2) 向工程添加有效的设计单元，包括设计源文件、ModelSim 管理文件夹、仿真环境设置等；可以将这些文件拷贝到工程目录，也可以简单地将它们映射到本地。

(3) 编译文件。进行语法检查并建立 ModelSim 仿真伪机器码。

(4) 仿真设计。对指定的设计单元进行仿真并在主窗口中打开结构标签页。

图 5.1 给出了在 ModelSim 工程中仿真一个设计的基本流程。该流程很类似于基础仿真流程，然而，也有以下两点重要的区别：

(1) 在工程流程图 5.1 中，不再需要创建工作库，在建立工程时自动创建工作库。

(2) 工程是连续的，每当调入 ModelSim 时工程都会自动打开，除非特别关闭它们。

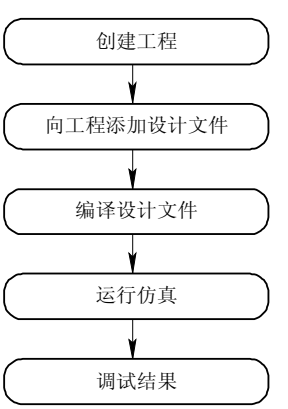



图 5.1 ModelSim 工程基本仿真流程

### 5.1.3 多数据库仿真流程

ModelSim 以下面两种方式使用设计数据库(关于设计库请参见 5.3 节)：

(1) 用作一个本地工作库，它包含了被编译的设计编译结果。

(2) 用作一个资源库。工作库的内容将随着设计的更新和编译而修改，而资源库被用于设计构成的一部分，是静态的。用户可以创建自己的资源库，资源库也可以由其他设计组或第三方来提供，如厂家或专业公司等。

在编译时，可以指定使用的资源库以及库搜寻规则，既使用工作库又使用资源库的一个常见的例子就是将一个设计和验证程序都编译在工作库中，而设计中调用的一些参考模块在一个第三方提供的资源库中。图 5.2 给出了多数据库使用流程图。

也可以在一个工程内进行资源库连接。如果使用了工程，图 5.2 中的第一个步骤就由下述步骤代替：创建工程，将验证程序、设计加入到工程中。

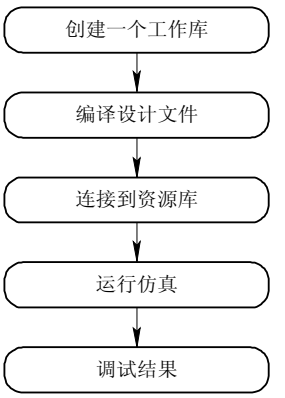



图 5.2　多数据库使用流程

### 5.1.4　调试工具

ModelSim 为调试、分析设计提供了众多的工具。本章将介绍以下几种常用工具：

(1) 设置断点与源代码单步执行。

(2) 波形查看与时间测量。

(3) 管理设计中的“物理”连通性。

(4) 查看与初始化存储器。

(5) 分析仿真性能。

(6) 测试代码覆盖情况。

(7) 波形比较。

(8) 批处理。

## 5.2　ModelSim 工程

### 5.2.1　创建一个新工程

本节使用的实例 tcounter.v 和 counter.v 存在于 <modelsim 安装目录>/example/目录下。counter.v 是一个简单的计数器设计，tcounter.v 是针对该设计的测试激励，本节以它们为例创建并组织一个 ModelSim 工程，如图 5.3、5.4、5.5 所示。

(1) 启动 ModelSim。

(2) 创建一个新工程。

① 在欢迎对话框中选择“Create a Project”，或在主菜单窗口的主菜单中选择“File→New→Project”，如图 5.3 所示。在打开的窗口中，输入项目名、项目位置(目录)以及默认库名称。默认库用于存储设计单元的编译结果。

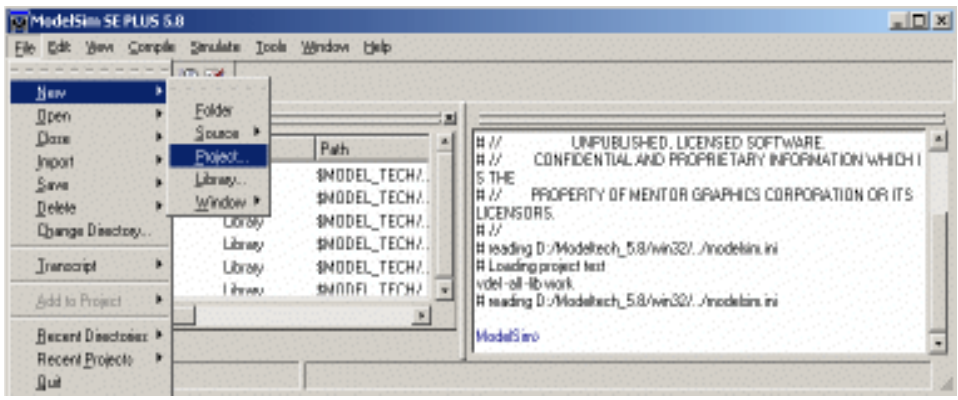



图 5.3　创建工程

② 在项目名称域中输入工程名称(如 test)。

③ 单击 Browse 按钮选择工程文件存储的目录。

④ 确认默认库名称为 work，单击 OK 按钮，此步骤后 .mpf 文件被创建并存储于该目录下。如图 5.4 所示。

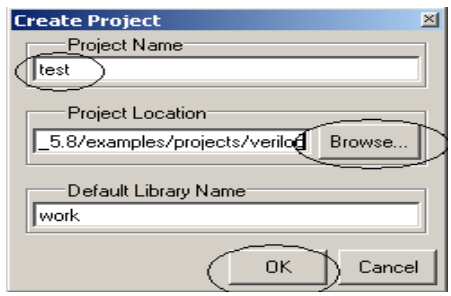



图 5.4　设定工程信息

(3) 向工程添加项目：单击 OK 按钮接受工程设置后，在主窗口的工作区中将出现一个工程标签，同时弹出向工程添加项目对话框。在该对话框中，可以创建新设计的文件，添加已存在的文件，添加用于管理的文件夹或建立仿真结构。

① 单击“Add Existing File”，启动 Add file to Project 对话框，使用 Browse 按钮选择添加两个已存在的文件 tcounter.v 和 counter.v。当一个工程包含多个文件时，可在按下 Ctrl 键的同时复选需要添加的文件。如图 5.5 所示。

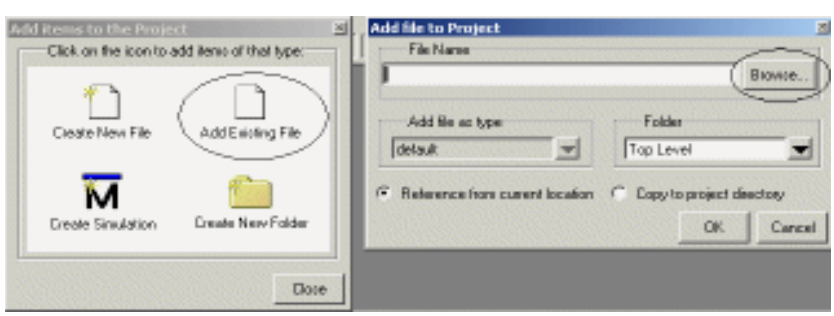



图 5.5　向工程添加文件

其中 Add file to Project (向工程添加文件)窗口的 Folder 选项用于将文件加入到指定的文件夹中。“Copy to project directory”和“Reference from current location”选项用于指定参考当前文件建立工程还是将文件拷贝到工程目录。

② 依次单击 Open 和 OK 按钮完成文件的添加。

③ 单击 Close 按钮退出向工程添加项目对话框。

此时，可在主窗口 Project 标签页，看到两个未编译的文件。

### 5.2.2 编译和加载设计

编译和加载设计的具体操作步骤如下：

(1) 在主窗口中选择 “Compile→Compile All”完成工程的编译。对于 ModelSim 正确编译的设计文件，都打上“√”标志；对于编译失败的情况，打上“×”标志，此时可在右侧的脚本状态窗中查看出错信息，修正后再编译。

(2) 完成工程的正确编译后，可以在主窗口中单击 Library 标签，进入编译库页，并在其中打开 work 库，可以查看编译后的设计单元、类型以及源文件路径等。

(3) 在 Library 标签页下，双击测试向量单元 test_counter，加载测试设计单元。在主窗口工作空间新出现 Sim 标签，指示仿真加载情况；同时出现 files 标签，指示源设计文件的相关信息。

(4) 仿真结束时，在主菜单中选择“Simulate→End Simulate”，结束仿真。

### 5.2.3 利用文件夹组织工程

如果在一个工程中有许多文件，则也可以用文件夹来组织管理它们。可以在将这些文件加入到工程前(或加入到工程后)创建文件夹，将不同用途、分类的文件分别组织到不同的文件夹。如果是在向工程中加入文件之前创建的文件夹，则可以在向工程中加入文件的同时，指定该文件组织到指定的文件夹中。如果是在向工程中加入文件之后创建了一个文件夹，则可以编辑文件属性，将文件移动组织到选定的目标文件夹中，文件夹还可以嵌套。该文件夹仅用于 ModelSim 工程，不能在操作系统中创建使用。

利用文件夹组织工程文件的具体步骤如下：

(1) 添加文件夹。打开图 5.6 所示的窗口，输入文件夹名等选项并确认。

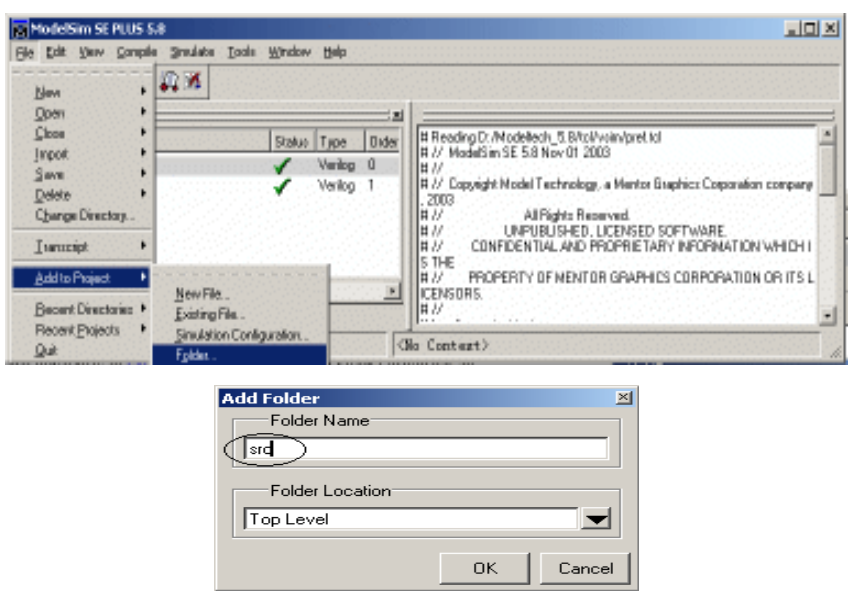



图 5.6 创建 ModelSim 管理文件夹

(2) 添加文件。

① 在向工程添加文件时，单击 Browse 按钮找到目标文件，在 Folder 下拉选项框中选定目标文件夹，再单击 OK 按钮将文件添加到目标文件夹中。如图 5.7 所示。

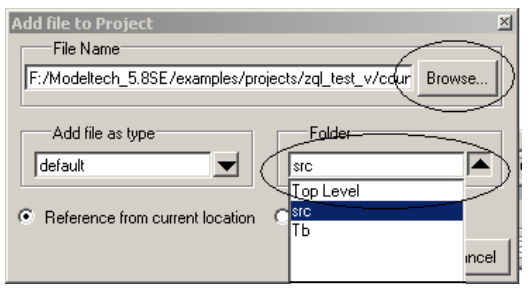



图 5.7　组织文件到 ModelSim 管理文件夹

② 将已在工程中的文件添加到目标文件夹。在图 5.8 的上图中选择文件，右键选择“Properties”打开 Project Compiler Settings(属性设置)窗口，在 General 选项卡的“Place in Folder”下拉选项框中选择目标目录，单击 OK 按钮完成设置。如图 5.8 所示。

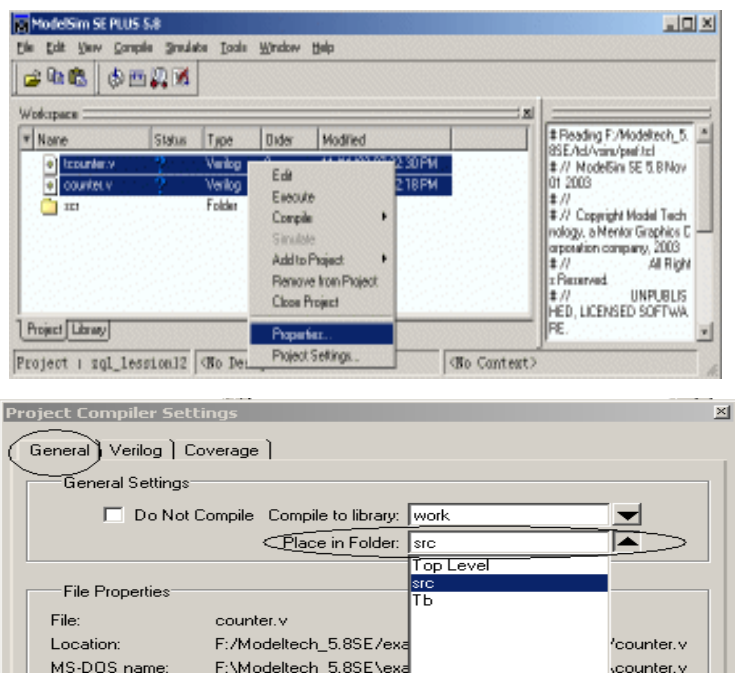



图 5.8　将工程中的文件组织到文件夹

## 5.2.4　在工程中进行仿真配置

仿真配置是指与设计单元和它的仿真相关的选项。通常情况下，加载并仿真一个设计时都要指定仿真分辨率、仿真类型和 SDF 时序文件等，这需要每次加载仿真时进行指定。使用仿真配置文件，可以将这些设置存储到仿真配置文件中，该配置文件列在了工程标签页中，在工程中使用。双击该配置文件即可按照配置文件的设置加载仿真。例如，在每次

调入 tcounter.v 时都希望将仿真器设置成分辨率为 ps，并且使能事件顺序冲突检查功能。一般情况，我们会在每次加载设计仿真时都要指定这些选项；利用仿真配置，则可以为设计指定相应的选项，并存储到一个“配置”文件中，该配置会被列在 Project 标签页中，以后双击它即可将 tcounter.v 以及其相应的选项设置并加载。

1) 创建一个新的仿真配置

(1) 在主窗口中选择“File→Add to Project→Simulation Configuration”打开仿真对话框。

(2) 在“Simulation Configuration Name”域输入仿真配置文件名，可为任意名称，例如 zql_count_TB。

(3) 在 Place in Folder 下拉选项框中选择仿真目标目录。

(4) 打开 work 库的“+”图标，并选择 test_counter。

(5) 打开 Resolution 下拉选框，并选择 ps。

(6) 对于 Verilog 设计，打开 Verilog 选项卡，并在其中选择“Enable Hazard Checking”。

(7) 单击 OK 按钮。

此时，在工程选项卡上显示了一个名称为 zql_count_TB 的仿真配置。如图 5.9 所示。

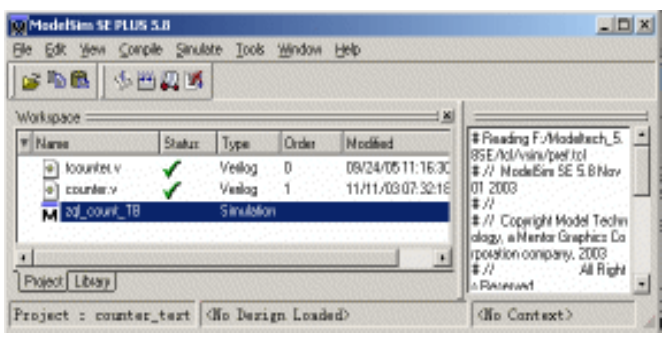



图 5.9　创建 ModelSim 仿真配置文件

2) 加载仿真配置

在工程选项卡中双击仿真配置 zql_count_TB。仿真器将按照设定加载仿真文件系统，同时在主窗口的脚本窗格中出现与之对应的命令行：vsim -hazards -t ps work.test_counter。也可以在主窗口的脚本窗格中直接输入命令：vsim -hazards -t ps work.test_counter。如图 5.10 所示。

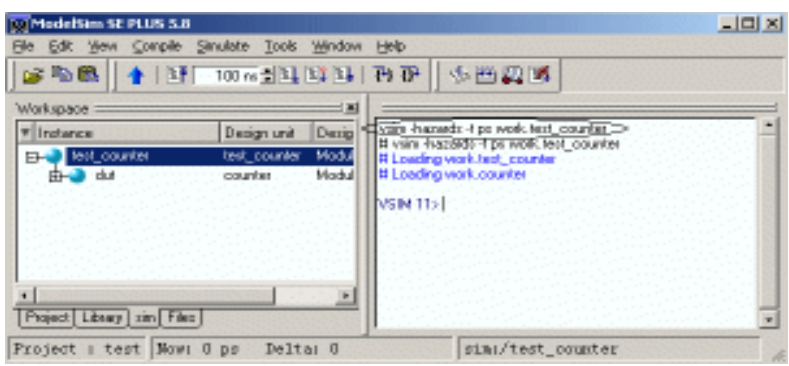



图 5.10　执行仿真配置

### 5.2.5 关于工程的其他基本操作

除了上面所介绍的工程操作外，还有如下所述一些常用的基本操作：

(1) 可以在主菜单中选择“File→Open→Project”，打开一个已存在的工程。同一时刻只能有一个工程被打开，当打开一个新工程时，原先打开的工程会自动关闭。

(2) 可以在主菜单中选择“File→Close→Project”，关闭当前已打开的工程，该操作将关闭 Workspace 中的 Project 标签，但仍遗留 Library 标签。注意，当工程正在仿真时不能关闭。

(3) 可以在主菜单中选择“File→Delete→Project”，删除一个已存在的工程，但不能删除一个当前正打开的工程。

### 5.2.6 Project 标签页及菜单简介

当一个工程建立完成后，在 ModelSim 的 Workspace 窗格中会新增一个 Project 标签页。默认情况下，该页有五列，如图 5.11 所示。

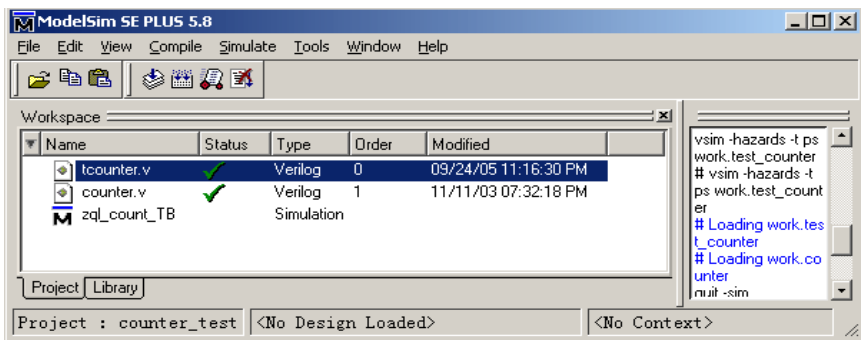



图 5.11 ModelSim 工程标签页介绍

图 5.11 中，Name 为文件或工程的名称；Status 表示源文件是否被成功编译；Type 为加入到工程中的文件的类型，如 VHDL、Verilog 源文件、子目录、库等；Order 为完全编译时的编译顺序；Modified 为最后修改的时间。

Project 标签页用鼠标的右键单击可以弹出如表 5.1 所示的菜单项。

表 5.1 Project 标签页的菜单项

| 菜 单 名 称 | 含 义 |
|---|---|
| Edit | 使用系统默认的文本编辑器打开选中文件 |
| Compile | 编译文件、修改设定编译顺序或查看编译信息 |
| Simulate | 加载设计单元并从仿真配置文件中加载相关的仿真选项配置 |
| Add to Project | 向工程添加项目(新文件、已存在的文件、文件夹和仿真配置等) |
| Remove from Project | 将选定的项目从工程中移除 |
| Close Project | 关闭当前活动的工程 |
| Properties | 查看、修改选定文件的编译设置 |
| Project Settings | 修改工程设定 |

### 5.2.7 指定文件属性和工程设置

在工程中有两种类型的属性设置：文件编译属性设置和工程设置。文件编译属性分别对各个文件有效，而工程设置作用于整个工程。

#### 1．文件编译属性设置

Verilog 编译器(Vlog)有多个可选项用于控制设计的编译和随后的仿真，用户可按文件和文件分组分别设置这些属性。无论是使用命令行、GUI，还是使用修改 modelsim.ini 这些非工程内的设定操作来修改编译属性，都不会对工程内的文件属性造成影响。

设定文件属性的方法是：在 Project 标签中选择文件，右击文件名并选择“Properties”，打开 Project Compiler Settings(属性设置)窗口(该窗口会随选中的文件个数、类型有所差别)。如果选择的是多个 .v 文件，则结果如图 5.12 所示。

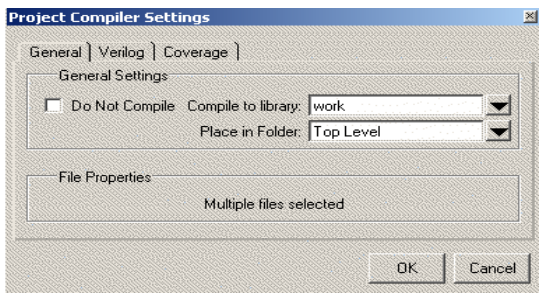



图 5.12　编辑文件属性

图 5.12 中，General 标签页包含的选项分别如下：

(1) Do Not Compile：确定选中的文件是否编译。

(2) Compile to library：确定选中的文件编译到哪个库中。

(3) Place in Folder：确定选中的文件放置到哪个目录中。

(4) File Properties：该子栏只对一个文件进行属性设置时有效，它显示了文件的类型、大小、路径等信息。

#### 2．工程设置

在 Project 标签页，单击鼠标右键并选择“Project Settings”打开相应的窗口，如图 5.13 所示。

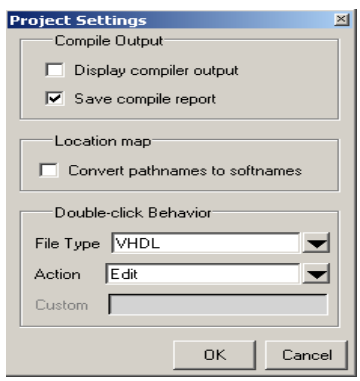



图 5.13　工程设置(1)

图 5.13 中各项含义如下：

(1) Display compiler output：将编译的详细结果输出到脚本子窗口，默认值为仅输出到编译报告中。

(2) Save compiler report：保存编译报告到磁盘。我们可以右键单击文件名并选择“Compile→Compile Report”访问该报告。

(3) Double-click Behavior：指定双击一个文件时的操作。如果设定某“FileType”指定的文件类型对应的“Action”是“Custom”，则可以在下面的 Custom 文本框中指定一条 TCL 命令。在该 TCL 命令中可以使用%f 宏代换文件名。例如，我们期望双击某 TCL 文件名时，用“记事本”打开它，就要在“Custom”文本框中输入“notepad %f”，系统会自动地用双击的文件名来替代%f 执行 notepad。如图 5.14 所示。

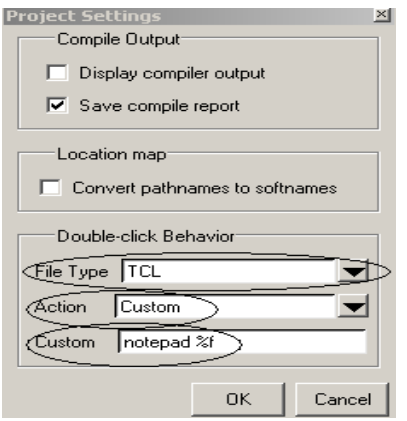



图 5.14　工程设置(2)

# 5.3 设　计　库

## 5.3.1　设计库简介

Verilog 设计文件中的所有模块和 UDPs(用户定义元语)必须被编译到一个或多个设计库中。Modelsim 系统中设计库含有以下信息：可重指定执行的代码，调试信息和从属信息等。该设计库分为两类：一类是资源库，另外一类是工作库。资源库是一个典型的静态库，用于存储第三方提供的已编译过的参考设计，可用作设计源文件的一部分。用户可以创建自己的资源库，也可以直接使用其他设计组或第三方(如器件厂家)提供的资源库。

设计库是一个目录或存档文件，用于存储当前设计单元的编译结果，当更新设计并重编译时，工作库内容即被修改。工作库中的设计单元由 Verilog 模块、UDPs 和 System C 模块等组成。默认情况下，设计库以目录结构来存储，其内的每个设计单元存储为一个子目录，也可以使用 vlib 命令的 -archive 参数创建存档文件，将设计库配置成一个存档文件方式。这种情况下，每个设计单元被存储为它的存档文件。这种用法不多见，例如编译时出现以下错误：

mkdir: cannot creat directory '65534': Too many links

系统指示超出了系统支持的子目录数时，需要使用此方式进行重编译。

**注意**：Unix 和 Linux 支持的最大子目录数为 65 533。

通常情况下，一个库可以容下一个简单的设计，也可以将一个复杂的设计的各模块分别编译到不同的库中。当在设计中使用有相同名称的不同模块时，则必须将这些模块编译到不同的库中，因为同一个库中的单元名必须惟一。

由于实例化绑定不是在编译时决定的，故当加载设计时，必须指导仿真器进行库搜寻。除非打开模块时使用“library”选项，顶层模块默认从 work 库加载，所有其他的 Verilog 实例按以下规则搜寻：

(1) 按照 vsim 命令行中的 -Lf(或 -L)参数指定的各库逻辑名在命令行中的先后顺序，进行库搜寻。

(2) 搜寻 work 库。

(3) 搜寻在实例名中专门指定的库。

Modelsim 当前工程中只能有一个库是工作库，在编译时其他的库都是资源库。我们可以指定编译时要用到的资源库和库搜寻的顺序规则。例如，有一个设计和一个相应的验证程序编译到了工作库中，而该设计中的一些模块在资源库中，该例就是一个既使用一个工作库，又使用一个资源库的常见例子。在 ModelSim 中名称为 work 的库具有特别的属性，work 库在编译器中被预定义为默认的工作库。

## 5.3.2 使用设计库工作

### 1．创建一个设计库

创建一个工程时，ModelSim 会自动地创建一个工作设计库。如果不创建工程，也可以在运行编译器前使用命令行方法和 ModelSim 图形界面法创建设计库。

使用命令行方式创建设计库时可直接使用命令：vlib <目录路径名>。

使用 ModelSim 图形界面法创建设计库时需要在 ModelSim 主窗口中选择“File→New→Library”。

### 2．创建资源库(关于第三方资源库的连接使用详见 5.5 节)

(1) 创建一个用于保存目标资源库的目录，将 counter.v 文件拷贝到该目录中。本书中该目录名为 parts_lib。

(2) 创建验证程序目录，该目录用于保存验证程序和工程文件。将文件 tcounter.v 拷贝到该目录中。我们在此创建了两个目录，来模拟收到第三方的资源库的情况。

(3) 启动 ModelSim，并选择“File→Change Directory”，将前面创建的“资源目录”作为当前目录。

(4) 在 ModelSim 中创建资源库。

① 在主菜单中选择“File→New→Library”，启动 Create a New Library(创建新目录)窗口。

② 在 Library Name(库名称)栏内输入库名称(本例中输入 parts_lib)，单击 OK 按钮创建该库。同时 ModelSim 将相关信息写入到 modelsim.ini 文件中。如图 5.15 所示。

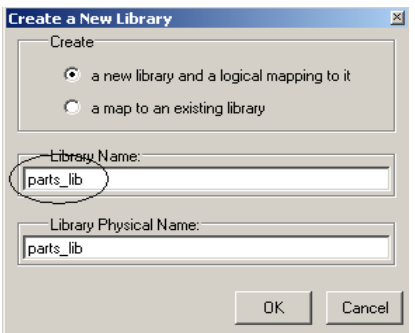



图 5.15　创建设计库

(5) 将 counter 编译到资源库中。

① 在菜单或图标中启动执行编译(Compile)，打开编译设定窗口，如图 5.16 所示。

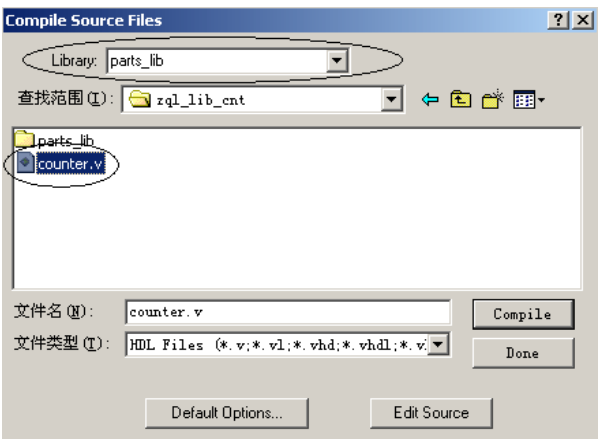



图 5.16　将设计编译到设计库中

② 在库下拉列表中选定目标库，在文件窗中选择目标文件。

③ 单击 Done 完成设置和编译。这样就完成了资源库的建立，以后 parts_lib 即可用作资源库。相应的命令行分别如下：

cd<资源目录全路径名>：指定当前工作目录。

vlib parts_lib：创建新库。

vmap parts_lib parts_lib：完成物理库到逻辑库的映射。

vlog –work parts_lib counter.v：将 counter.v 编译到 parts_lib 库中。

(6) 在主菜单中选择“File→Change Directory”，将“验证目录”设置为 ModelSim 的当前目录。

### 3．管理设计库内容

对设计库的内容进行浏览、删除、重编译和编辑等操作可以命令行方式和 GUI 方式进行。

主窗口中的 Library 标签页提供了设计库中可访问的设计单元，这些列表按层次组织，并且在 Type 栏列出了各单元的类型。

在 Library 标签页单击鼠标右键，可以弹出相应的操作菜单，其中包含以下命令：

(1) Simulate：加载选中的设计单元，相当于执行 vsim 命令。

(2) Edit：在源代码编辑窗口打开选中的设计单元。

(3) Refresh：刷新选中的库映射，相当于执行带有 -refresh 参数的 Vlog 命令。

(4) Recompile：重编译选中的设计单元，相当于执行 Vlog 命令。

(5) Optimize：优化 Verilog 设计单元，相当于执行带有+opt 参数的 Vlog 命令。

(6) Update：可用的库和设计单元的显示更新。

(7) Delete：删除选定的设计单元，相当于执行 Vdel 命令。

(8) New：创建一个新库。

(9) Properties：显示选定的设计库或设计单元的属性。

### 4. 为设计库指定逻辑名

默认情况下，ModelSim 在当前目录下查找设计库。对于位于其他目录的设计库查找，需要映射逻辑库名到设计库的全路径。我们可以使用 GUI、命令行或工程等方式对一个设计库指定逻辑名。

1) 使用 GUI 进行库映射

在主窗口的 Library 标签页中选择设计库，右键单击鼠标并在弹出的菜单中选择 Edit，打开“Edit Library Mapping”窗口，编辑设计库映射。如图 5.17 所示。

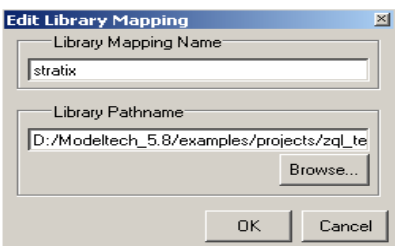



图 5.17　在 GUI 中进行库映射

图 5.16 所示的窗口选项含义如下：

① Library Mapping Name：设计库的逻辑名称。

② Library Pathname：逻辑库全路径名称。

2) 使用命令行进行库映射

在 ModelSim 中使用命令行“vmap <库逻辑名>　<库物理位置全路径名>”可实现逻辑库名和目录间的映射。该命令将在 modelsim.ini 的“library section”中添加相应的映射信息，也可以直接用文本编辑器在该文件的相应位置处直接加入“<库逻辑名> = <库物理位置全路径名>”实现同样的功能。

3) 设计库搜寻规则

系统首先搜寻 modelsim.ini 文件。如果未找到该文件，或该文件中未包含指定的逻辑名，系统将在当前工作目录下按逻辑名搜寻匹配的子目录。如果还未搜寻到映射的逻辑名，则在编译时由编译器产生一个出错信息。

### 5. 设计库的移动

设计库中的个别设计单元不能移动，但使用操作系统命令可以使设计库整体进行移动。