



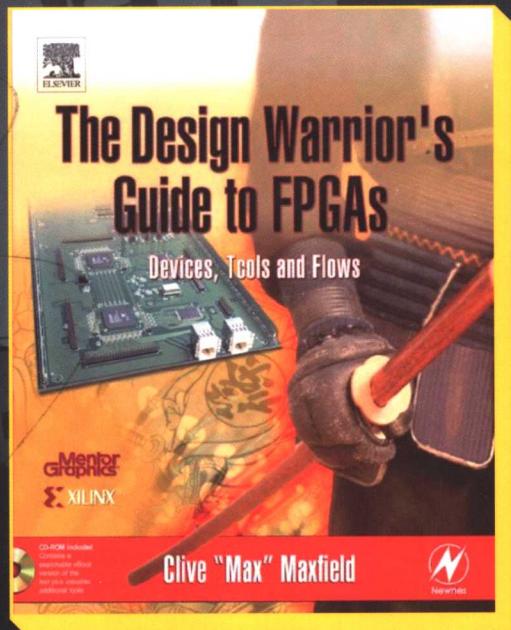
图灵电子与电气工程丛书



# FPGA设计指南 器件、工具和流程

The Design Warrior's Guide to FPGAs  
Devices, Tools and Flows

[美] Clive "Max" Maxfield 著  
杜生海 邢闻 译



人民邮电出版社  
POSTS & TELECOM PRESS



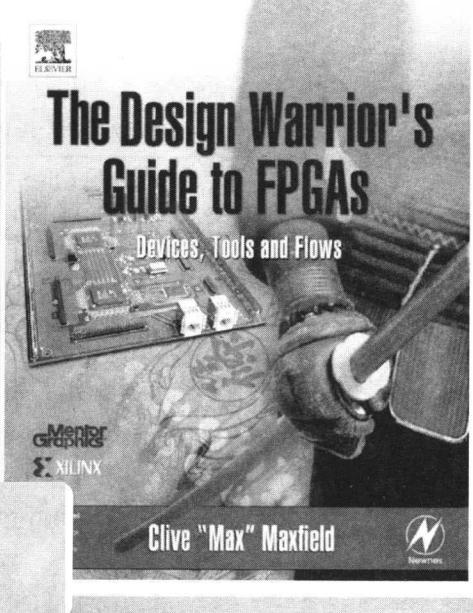
图灵电子与电气工程丛书

# FPGA设计指南

## 器件、工具和流程

The Design Warrior's Guide to FPGAs  
Devices, Tools and Flows

[美] Clive "Max" Maxfield 著  
杜生海 邢闻 译



人民邮电出版社  
北京

## 图书在版编目（CIP）数据

FPGA 设计指南：器件、工具和流程 / （美）马克斯菲尔德  
(Maxfield, C.) 著；杜生海，邢闻译。—北京：人民邮电出版社，2007.12

（图灵电子与电气工程丛书）

ISBN 978-7-115-16862-7

I . F… II . ①马…②杜…③邢… III. 可编程序编辑器  
件—系统设计 IV. TP332.1

中国版本图书馆 CIP 数据核字（2007）第 146283 号

## 内 容 提 要

本书用简洁的语言向读者展示了什么是FPGA、FPGA如何工作、如何对FPGA编程以及FPGA设计中遇到的各种概念、器件和工具，如传统的基于HDL/RTL的仿真和逻辑综合、最新的纯C/C++设计捕获和综合技术以及基于DSP的设计流程。另外，本书还涉及大量丰富的、工程师所需的技术细节。

本书适用于使用FPGA进行设计的工程师、进行嵌入式应用任务开发的软件工程师以及高等院校电气工程专业的师生。

图灵电子与电气工程丛书

## FPGA 设计指南：器件、工具和流程

- 
- ◆ 著 [美] Clive “Max” Maxfield
  - 译 杜生海 邢 闻
  - 责任编辑 朱 巍
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
  - 邮编 100061 电子函件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 北京铭成印刷有限公司印刷
  - 新华书店总店北京发行所经销
  - ◆ 开本：700×1000 1/16
  - 印张：22
  - 字数：432 千字 2007 年 12 月第 1 版
  - 印数：1~4 000 册 2007 年 12 月北京第 1 次印刷

著作权合同登记号 图字：01-2006-2739 号

ISBN 978-7-115-16862-7/TN

定价：49.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223

# 版 权 声 明

*The Design Warrior's Guide to FPGAs*

by Clive "Max" Maxfield

ISBN 0-7506-7604-3

Copyright © 2004, Mentor Graphics Corporation and Xilinx, Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 981-259-632-1

Copyright © 2007 by Elsevier (Singapore) Pte Ltd. All rights reserved.

**Elsevier (Singapore) Pte Ltd.**

3 Killiney Road

#08-01 Winsland House I

Singapore 239519

Tel: (65)6349-0200

Fax: (65)6733-1817

First Published 2007

2007年初版

Printed in China by POSTS & TELECOM PRESS under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan.

Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由Elsevier (Singapore) Pte Ltd.授权人民邮电出版社在中华人民共和国境内出版发行。

本版仅限于在中华人民共和国境内（不包括中国香港特别行政区和台湾地区）  
及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

## 致 谢

很早以前我就想写一本FPGA方面的书了，所以当Elsevier公司（据我所知，它是世界上最大的英语图书出版商之一）的出版人Carol Lewis给予我这样的机会时，我很高兴。

然而，有个小问题，我已经把生命中近10年的大部分时间用于工作了，现在还要把晚间和周末都消磨在写书中。某种程度上，这减少了我陪伴家人和朋友的时间。因此，我很高兴Mentor Graphics和Xilinx公司能够为本书的创作提供赞助，这使得我能够在白天写作，而晚间和周末仍然属于自己。

作为一名专业工程师，我也不喜欢看实际上是介绍某种专门技术的书籍。所以令我高兴的是，两个赞助者都明确了本书的内容不应以Mentor或Xilinx为中心，而应该包括所有我认为有用的信息（不计个人好恶的）。

如果不与人协作，一个人是无法完成这样一本书的。在此期间，我接受了大量的帮助和建议，无法在此一一提及。但是，我想在此感谢所有帮助我的在Mentor和Xilinx公司工作的人，他们为我花费了很多时间，并给我提供了大量信息。还要感谢Gartner DataQuest公司的Gray Smith 和Daya Nadamuni，以及EETimes杂志的Richard Goering，他们一直在花时间回复我的电子邮件（这些邮件都带着令人惧怕的标题“Just one more little question...”）。

在此，我还想感谢以下公司对我提供帮助的人，这些公司是0-In、AccelChip、Actel、Aldec、Altera、Altium、Axis、Cadence、Carbon、Celoxica、Elanix、InTime、Magma、picoChip、QuickLogic、QuickSilver、Synopsys、Synplicity、The MathWorks、Hier Design和Verisity。<sup>①</sup>我还要感谢Launchbird Design Systems的Tom Hawkins给予我的热情帮助，他在开源设计工具方面的洞察力使我受益匪浅。同样，GigaTest实验室的Eric Bogatin与我分享了他在电路板级信号完整性方面的知识。

最后，当然也很重要的一点，再次感谢我的出版人——Elsevier公司的Carol Lewis，他允许我从我的*Designus Maximus Unleashed*一书中摘取出附录B的内容，而且允许我从我的另一本书*Bebop to the Boolean Boogie Second Edition*中摘取出附录C的内容。

---

<sup>①</sup>如果有遗漏，我将十分抱歉（请通知我，我会将其加入到本书的下一版中）。

## 前　　言

本书内容新颖奇特，但并不是一本典型的技术流派的书（作为作者，我知道这一点）。我之所以这么说，是因为本书旨在满足非常广泛的、各类读者的兴趣需要。本书的主要读者是目前正在使用现场可编程门阵列（FPGA）进行设计的合格的工程师，或计划在不久的将来这么做的人。本书涉及大量丰富的、工程师所喜好的技术细节，如多种不同的设计流程、工具和概念。此外本书还涵盖了一系列技术层次相对低的主题，如基本概念。

我之所以如此编排本书，是因为目前人们对FPGA产生了浓厚的兴趣，尤其是还没有使用过或认真考虑过FPGA的人。最早的FPGA器件在它们所支持的等效逻辑门数量和所提供的性能方面相对有限，所以任何“严肃”（巨大、复杂、高性能）的设计都是由专用集成电路（ASIC）或专用标准部件（ASSP）自动实现的。然而，设计和制造ASIC和ASSP极其费时且造价高昂，并且最终设计是固定的，不开发器件的新版本就无法对其修改。

相比之下，创建一个FPGA设计的成本要比设计ASIC或ASSP低得多，在FPGA中实现设计改动更为容易，这样这种设计的上市速度就更快。特别令人感兴趣的是，最近面世的新FPGA架构中包括了数百万计的等效逻辑门、嵌入式处理器和超高速互连。这些器件允许FPGA被用于那些目前还仅是ASIC和ASSP的应用。

与FPGA器件中嵌入式处理器有关的设计，需要硬件工程师和软件工程师的协作。很多情况下，软件工程师可能不太熟悉这些器件中的一些与硬件有关的基本设计考虑。因此，除了硬件工程师，本书也旨在满足为这些器件承担开发嵌入式应用任务的软件工程师们的需求。

本书还适用于高等院校电气工程专业的学生，为EDA和FPGA公司工作的销售人员、市场人员，以及市场分析师和杂志编辑等相关读者。这些读者可以从基本概念和附录介绍的材料中发现对自己有帮助的底层技术细节。

最后，我要把它写成我最喜欢读的那类书。（此时此刻，我很愿意阅读本书——在开始这项工作时我泰然自若——因为那时我已经有一些关于如何写作本书的线索……如果你领会我的意思。）说实话，我自己也不喜欢阅读技术书籍，因为它们往往令人头痛。因此，我在写作时，更愿意把复杂的主题分散到基本概念（“它

们从何而来”和“我们为什么这样做”)以及使人们产生兴趣的技术细节之中。相比从前精力旺盛的时候,这样做增添了我的优势,同时我还可以通过阅读自己的著作(总能有些美好的期待)从中获得惊喜和快乐。

Clive “Max” Maxfield

# 目 录

<b>第1章 概论 .....</b>	1
1.1 什么是FPGA .....	1
1.2 FPGA为什么令人感兴趣 .....	1
1.3 FPGA的用途 .....	2
1.4 本书内容 .....	3
1.5 本书不包括什么 .....	4
1.6 读者对象 .....	4
<b>第2章 基本概念 .....</b>	5
2.1 FPGA的核心 .....	5
2.2 简单的可编程功能 .....	5
2.3 熔丝连接技术 .....	5
2.4 反熔丝技术 .....	7
2.5 掩模编程器件 .....	8
2.6 PROM .....	9
2.7 基于EPROM的技术 .....	10
2.8 基于EEPROM的技术 .....	12
2.9 基于闪存的技术 .....	12
2.10 基于SRAM的技术 .....	12
2.11 小结 .....	13
<b>第3章 FPGA的起源 .....</b>	15
3.1 相关的技术 .....	15
3.2 晶体管 .....	15
3.3 集成电路 .....	16
3.4 SRAM/DRAM和微处理器 .....	16
3.5 SPLD和CPLD .....	17
3.5.1 PROM .....	18
3.5.2 PLA .....	20
3.5.3 PAL和GAL .....	22
3.5.4 其他可编程选择 .....	22
3.5.5 CPLD .....	23
3.5.6 ABEL、CUPL、PALASM、JEDEC等 .....	24
3.6 专用集成电路（门阵列等） .....	25
3.6.1 全定制 .....	26
3.6.2 Micromatrix和Micromosaic .....	26
3.6.3 门阵列 .....	27
3.6.4 标准单元器件 .....	28
3.6.5 结构化ASIC .....	29
3.7 FPGA .....	30
3.7.1 FPGA平台 .....	32
3.7.2 FPGA-ASIC 混合 .....	33
3.7.3 FPGA厂商如何设计芯片 .....	34
<b>第4章 FPGA结构的比较 .....</b>	35
4.1 一点提醒 .....	35
4.2 一些背景信息 .....	35
4.3 反熔丝与SRAM与其他 .....	36
4.3.1 基于SRAM的器件 .....	36
4.3.2 以SRAM为基础器件的安全问题和解决方案 .....	37
4.3.3 基于反熔丝的器件 .....	37
4.3.4 基于EPROM的器件 .....	39
4.3.5 基于E <sup>2</sup> PROM/FLASH的器件 .....	39
4.3.6 FLASH-SRAM混合器件 .....	39
4.3.7 小结 .....	40
4.4 细粒、中等颗粒和粗粒结构 .....	40
4.5 MUX与基于LUT的逻辑块 .....	41
4.5.1 基于MUX的结构 .....	41
4.5.2 基于LUT的结构 .....	42
4.5.3 基于MUX还是基于LUT .....	43
4.5.4 3、4、5或6输入LUT .....	44
4.5.5 LUT与分布RAM与SR .....	44
4.6 CLB、LAB与slices .....	45
4.6.1 Xilinx 逻辑单元 .....	45
4.6.2 Altera逻辑部件 .....	46
4.6.3 slicing和dicing .....	46
4.6.4 CLB和LAB .....	47
4.6.5 分布RAM和移位寄存器 .....	47

4.7 快速进位链 .....	48	6.5 专业FPGA和独立EDA提供商 .....	74
4.8 内嵌RAM .....	48	6.6 使用专门工具的FPGA设计 顾问 .....	75
4.9 内嵌乘法器、加法器、MAC等 .....	49	6.7 开源、免费和低成本的设计 工具 .....	75
4.10 内嵌处理器核（硬的和软的） .....	50	<b>第7章 FPGA与ASIC设计风格 .....</b>	77
4.10.1 硬微处理器核 .....	50	7.1 引言 .....	77
4.10.2 软微处理器核 .....	52	7.2 编码风格 .....	77
4.11 时钟树和时间管理器 .....	52	7.3 流水线和逻辑层次 .....	77
4.11.1 时钟树 .....	52	7.3.1 什么是流水线 .....	77
4.11.2 时钟管理器 .....	53	7.3.2 电子系统中的流水线 .....	78
4.12 通用I/O .....	55	7.3.3 逻辑层次 .....	79
4.12.1 可配置I/O标准 .....	56	7.4 异步设计实践 .....	80
4.12.2 可配置I/O阻抗 .....	56	7.4.1 异步结构 .....	80
4.12.3 核与I/O电压 .....	57	7.4.2 组合回路 .....	80
4.13 吉比特传输 .....	57	7.4.3 延迟链 .....	81
4.14 硬IP、软IP和固IP .....	58	7.5 时钟考虑 .....	81
4.15 系统门与实际的门 .....	59	7.5.1 时钟域 .....	81
4.16 FPGA年 .....	60	7.5.2 时钟平衡 .....	81
<b>第5章 FPGA编程（配置） .....</b>	62	7.5.3 门控时钟与使能时钟 .....	81
5.1 引言 .....	62	7.5.4 PLL和时钟调节电路 .....	82
5.2 配置文件 .....	62	7.5.5 跨时钟域数据传输的 可靠性 .....	82
5.3 配置单元 .....	62	7.6 寄存器和锁存器考虑 .....	82
5.4 基于反熔丝的FPGA .....	63	7.6.1 锁存器 .....	82
5.5 基于SRAM的FPGA .....	64	7.6.2 具有“置位”和“复位” 输入的触发器 .....	82
5.5.1 迅速的过程欺骗了眼睛 .....	65	7.6.3 全局复位和初始化条件 .....	83
5.5.2 对嵌入式（块）RAM、分布 RAM编程 .....	65	7.7 资源共享（时分复用） .....	83
5.5.3 多编程链 .....	66	7.7.1 使用它或者放弃它 .....	83
5.5.4 器件的快速重新初始化 .....	66	7.7.2 其他内容 .....	83
5.6 使用配置端口 .....	66	7.8 状态机编码 .....	84
5.6.1 FPGA作为主设备串行下载 .....	67	7.9 测试方法学 .....	84
5.6.2 FPGA作为主设备并行下载 .....	68	<b>第8章 基于原理图的设计流程 .....</b>	85
5.6.3 FPGA作为从设备并行下载 .....	69	8.1 往昔的时光 .....	85
5.6.4 FPGA作为从设备串行下载 .....	70	8.2 EDA初期 .....	86
5.7 使用JTAG端口 .....	70	8.2.1 前端工具，如逻辑仿真 .....	86
5.8 使用嵌入式处理器 .....	71	8.2.2 后端工具如版图设计 .....	89
<b>第6章 谁在参与游戏 .....</b>	73	8.2.3 CAE + CAD = EDA .....	90
6.1 引言 .....	73	8.3 简单的原理图驱动ASIC设计 流程 .....	90
6.2 FPGA和FPAA提供商 .....	73		
6.3 FPNA 提供商 .....	73		
6.4 全线EDA提供商 .....	74		

8.4 简单（早期）的原理图驱动FPGA	114
设计流程 ..... 91	
8.4.1 映射 ..... 92	
8.4.2 包装 ..... 93	
8.4.3 布局和布线 ..... 94	
8.4.4 时序分析和布局布线后仿真 ..... 94	
8.5 平坦的原理图与分层次的	
原理图 ..... 95	
8.5.1 沉闷的扁平原理图 ..... 95	
8.5.2 分等级（基于模块）的	
原理图 ..... 96	
8.6 今天的原理图驱动设计流程 ..... 97	
<b>第9章 基于HDL的设计流程 ..... 98</b>	
9.1 基于原理图流程的问题 ..... 98	
9.2 基于HDL设计流程的出现 ..... 98	
9.2.1 不同的抽象层次 ..... 98	
9.2.2 早期基于HDL的ASIC设计	
流程 ..... 99	
9.2.3 早期基于HDL的FPGA设计	
流程 ..... 101	
9.2.4 知道结构的FPGA流程 ..... 102	
9.2.5 逻辑综合与基于物理的	
综合 ..... 102	
9.3 图形设计输入的生活 ..... 103	
9.4 绝对过剩的HDL ..... 104	
9.4.1 Verilog HDL ..... 104	
9.4.2 VHDL和VITAL ..... 106	
9.4.3 混合语言设计 ..... 108	
9.4.4 UDL/I ..... 108	
9.4.5 Superlog 和 SystemVerilog ..... 108	
9.4.6 SystemC ..... 109	
9.5 值得深思的事 ..... 110	
9.5.1 担心，非常担心 ..... 110	
9.5.2 串行与并行多路复用器 ..... 110	
9.5.3 小心锁存器 ..... 111	
9.5.4 聪明地使用常量 ..... 111	
9.5.5 资源共用考虑 ..... 112	
9.5.6 还有一些不可忽视的内容 ..... 113	
<b>第10章 FPGA设计中的硅虚拟原型 ..... 114</b>	
10.1 什么是硅虚拟原型 ..... 114	
10.2 基于ASIC的SVP方法 ..... 114	
10.2.1 门级SVP（由快速综合	
产生） ..... 115	
10.2.2 门级SVP（由基于增益的	
综合产生） ..... 115	
10.2.3 团簇SVP ..... 117	
10.2.4 基于RTL的SVP ..... 117	
10.3 基于FPGA的SVP ..... 119	
10.3.1 交互式操作 ..... 120	
10.3.2 增量式布局布线 ..... 121	
10.3.3 基于RTL的FPGA SVP ..... 121	
<b>第11章 基于C/C++等语言的设计</b>	
流程 ..... 122	
11.1 传统的HDL设计流程存在的	
问题 ..... 122	
11.2 C对C++与并行执行对顺序	
执行 ..... 124	
11.3 基于SystemC的设计流程 ..... 125	
11.3.1 什么是SystemC以及它从	
哪里来 ..... 125	
11.3.2 SystemC 1.0 ..... 125	
11.3.3 SystemC 2.0 ..... 126	
11.3.4 抽象级 ..... 127	
11.3.5 基于SystemC设计流程的	
可选方案 ..... 127	
11.3.6 要么喜爱它，要么讨厌它 ..... 129	
11.4 基于增强型C/C++的设计流程 ..... 129	
11.4.1 什么是增强型C/C++ ..... 129	
11.4.2 可选择的增强型C/C++设计	
流程 ..... 131	
11.5 基于纯C/C++的设计流程 ..... 132	
11.6 综合的不同抽象级别 ..... 134	
11.7 混合语言设计和验证环境 ..... 136	
<b>第12章 基于DSP的设计流程 ..... 138</b>	
12.1 DSP简介 ..... 138	
12.2 可选择的DSP实现方案 ..... 139	
12.2.1 随便选一个器件，不过不要	
让我看到是哪种器件 ..... 139	
12.2.2 系统级评估和算法验证 ..... 139	
12.2.3 在DSP内核中运行的软件 ..... 140	

12.2.4 专用DSP硬件 .....	141	14.2.1 模块化设计 .....	169
12.2.5 与DSP相关的嵌入式		14.2.2 增量设计 .....	169
FPGA资源 .....	143	14.2.3 存在的问题 .....	170
12.3 针对DSP的以FPGA为中心的		14.3 总有其他办法 .....	171
设计流程 .....	144		
12.3.1 专用领域语言 .....	144		
12.3.2 系统级设计和仿真环境 .....	145		
12.3.3 浮点与定点表示 .....	146		
12.3.4 系统/算法级向RTL的转换			
(手工转换) .....	146	15.4.1 高速设计 .....	175
12.3.5 系统/算法级向RTL的转换		15.4.2 信号完整性分析 .....	175
(自动生成) .....	147	15.4.3 SPICE与IBIS .....	176
12.3.6 系统/算法级向C/C++的		15.4.4 起动功率 .....	176
转换 .....	148	15.4.5 使用内部末端阻抗 .....	176
12.3.7 模块级IP环境 .....	150	15.4.6 串行或并行处理数据 .....	177
12.3.8 别忘了测试平台 .....	150		
12.4 DSP与VHDL/Verilog混合			
设计环境 .....	151		
<b>第13章 基于嵌入式处理器的</b>			
设计流程 .....	153		
13.1 引言 .....	153		
13.2 硬核与软核 .....	154		
13.2.1 硬核 .....	154	16.1 缺乏可见性 .....	178
13.2.2 微处理器软核 .....	156	16.2 使用多路复用技术 .....	179
13.3 将设计划分为硬件和		16.3 专用调试电路 .....	180
软件部分 .....	157	16.4 虚拟逻辑分析仪 .....	180
13.4 硬件和软件的世界观 .....	159	16.5 虚拟线路 .....	181
13.5 利用FPGA作为自身的		16.5.1 问题描述 .....	181
开发环境 .....	160	16.5.2 虚拟线路解决方案 .....	183
13.6 增强设计的可见性 .....	161		
13.7 其他一些混合验证方法 .....	161		
13.7.1 RTL (VHDL或Verilog) .....	162	<b>第17章 IP</b> .....	185
13.7.2 C/C++、SystemC等 .....	162	17.1 IP的来源 .....	185
13.7.3 硬件模拟器中的物理芯片 .....	163	17.2 人工优化的IP .....	185
13.7.4 指令集仿真器 .....	163	17.2.1 未加密的RTL级IP .....	186
13.8 一个相当巧妙的设计环境 .....	165	17.2.2 加密的RTL级IP .....	186
<b>第14章 模块化设计和增量设计</b> .....	167	17.2.3 未经布局布线的网表级IP .....	186
14.1 将设计作为一个大的模块		17.2.4 布局布线后的网表级IP .....	186
进行处理 .....	167	17.3 IP核生成器 .....	187
14.2 将设计划分为更小的模块 .....	168	17.4 综合资料 .....	187
<b>第15章 高速设计与其他PCB设计</b>			
注意事项 .....	172		
15.1 开始之前 .....	172		
15.2 我们都很年轻,因此 .....	172		
15.3 变革的时代 .....	173		
15.4 其他注意事项 .....	175		
15.4.1 高速设计 .....	175		
15.4.2 信号完整性分析 .....	175		
15.4.3 SPICE与IBIS .....	176		
15.4.4 起动功率 .....	176		
15.4.5 使用内部末端阻抗 .....	176		
15.4.6 串行或并行处理数据 .....	177		
<b>第16章 观察FPGA的内部节点</b> .....	178		
16.1 缺乏可见性 .....	178		
16.2 使用多路复用技术 .....	179		
16.3 专用调试电路 .....	180		
16.4 虚拟逻辑分析仪 .....	180		
16.5 虚拟线路 .....	181		
16.5.1 问题描述 .....	181		
16.5.2 虚拟线路解决方案 .....	183		
<b>第17章 IP</b> .....	185		
17.1 IP的来源 .....	185		
17.2 人工优化的IP .....	185		
17.2.1 未加密的RTL级IP .....	186		
17.2.2 加密的RTL级IP .....	186		
17.2.3 未经布局布线的网表级IP .....	186		
17.2.4 布局布线后的网表级IP .....	186		
17.3 IP核生成器 .....	187		
17.4 综合资料 .....	187		
<b>第18章 ASIC设计与FPGA设计</b>			
之间的移植 .....	189		
18.1 可供选择的设计方法 .....	189		
18.1.1 只做FPGA设计 .....	189		
18.1.2 FPGA之间的转换 .....	189		
18.1.3 FPGA到ASIC的转换 .....	190		

18.1.4 ASIC到FPGA的转换 .....	191	19.7 混合设计 .....	218
<b>第19章 仿真、综合、验证等设计</b>		19.7.1 HDL语言到C语言的转换 .....	218
<b>工具</b> .....	193	19.7.2 代码覆盖率 .....	219
19.1 引言 .....	193	19.7.3 性能分析 .....	220
19.2 仿真（基于周期、事件 驱动等） .....	193	<b>第20章 选择合适的器件</b> .....	221
19.2.1 什么是事件驱动逻辑 仿真器 .....	193	20.1 丰富的选择 .....	221
19.2.2 事件驱动逻辑仿真器发展 过程简述 .....	195	20.2 要是有选型工具就好了 .....	221
19.2.3 逻辑值与不同逻辑值系统 .....	196	20.3 工艺 .....	222
19.2.4 混合语言仿真 .....	197	20.4 基本资源和封装 .....	223
19.2.5 其他延迟格式 .....	198	20.5 通用I/O接口 .....	223
19.2.6 基于周期的仿真器 .....	201	20.6 嵌入式乘法器、RAM等 .....	224
19.2.7 选择世界上最好的逻辑 仿真器 .....	202	20.7 嵌入式处理器核 .....	224
19.3 综合（逻辑/HDL综合与 物理综合） .....	203	20.8 吉比特I/O能力 .....	224
19.3.1 逻辑/HDL综合技术 .....	203	20.9 可用的IP .....	224
19.3.2 物理综合技术 .....	203	20.10 速度等级 .....	225
19.3.3 时序重调、复制及二次 综合 .....	204	20.11 轻松的注解 .....	226
19.3.4 选择世界上最好的综合 工具 .....	206	<b>第21章 吉比特收发器</b> .....	227
19.4 时序分析（静态与动态） .....	206	21.1 引言 .....	227
19.4.1 静态时序分析 .....	206	21.2 差分对 .....	228
19.4.2 统计静态时序分析 .....	207	21.3 多种多样的标准 .....	229
19.4.3 动态时序分析 .....	207	21.4 8bit/10bit编码等 .....	230
19.5 一般验证 .....	208	21.5 深入收发器模块内部 .....	231
19.5.1 验证IP .....	208	21.6 组合多个收发器 .....	233
19.5.2 验证环境和创建testbench .....	210	21.7 可配置资源 .....	234
19.5.3 分析仿真结果 .....	211	21.7.1 逗号检测 .....	234
19.6 形式验证 .....	211	21.7.2 差分输出摆幅 .....	234
19.6.1 形式验证的不同种类 .....	212	21.7.3 片内末端电阻 .....	234
19.6.2 形式验证究竟是什么 .....	212	21.7.4 预加重 .....	234
19.6.3 术语及定义 .....	213	21.7.5 均衡化 .....	235
19.6.4 其他可选的断言/属性 规范技术 .....	214	21.8 时钟恢复、抖动和眼图 .....	236
19.6.5 静态形式验证和动态形式 验证 .....	216	21.8.1 时钟恢复 .....	236
19.6.6 各种语言的总结 .....	217	21.8.2 抖动和眼图 .....	237

23.3.1 一个理想的picoArray应用： 无线基站 .....	246
23.3.2 picoArray设计环境 .....	247
23.4 QuickSilver公司的ACM技术 .....	247
23.4.1 设计混合节点 .....	249
23.4.2 系统控制器节点、输入输出 节点及其他节点 .....	249
23.4.3 空间与时间分割 .....	250
23.4.4 在ACM上创建和运行程序 .....	251
23.4.5 还有更多的内容 .....	252
23.5 这就是硅，但与我们知道的 并不相同 .....	252
<b>第24章 独立的设计工具 .....</b>	<b>253</b>
24.1 引言 .....	253
24.2 ParaCore Architect .....	253
24.2.1 产生浮点处理功能模块 .....	254
24.2.2 产生FFT功能模块 .....	254
24.2.3 基于网络的接口 .....	255
24.3 Confluence系统设计语言 .....	256
24.3.1 一个简单的例子 .....	256
24.3.2 还有更多的功能 .....	258
24.3.3 免费评估版本 .....	259
24.4 你是否具有这种工具 .....	259
<b>第25章 创建基于开源的设计流程 .....</b>	<b>260</b>
25.1 如何白手起家创办一家FPGA 设计工作室 .....	260
25.2 开发平台：Linux .....	260
25.3 验证环境 .....	262
25.3.1 Icarus Verilog .....	263
25.3.2 DinoTrace和GTKWave .....	263
25.3.3 Covered代码覆盖率工具 .....	263
25.3.4 Verilator .....	263
25.3.5 Python .....	264
25.4 形式验证 .....	264
25.4.1 开源模型检查 .....	265
25.4.2 基于开源的自动推断 .....	265
25.4.3 真正的问题是什么 .....	266
25.5 访问公共IP元件 .....	266
25.5.1 OpenCores .....	266
25.5.2 OVL .....	267
25.6 综合与实现工具 .....	267
25.7 FPGA开发板 .....	267
25.8 综合材料 .....	267
<b>第26章 FPGA未来的发展 .....</b>	<b>269</b>
26.1 一种担忧 .....	269
26.2 下一代结构和技术 .....	269
26.2.1 十亿晶体管级器件 .....	269
26.2.2 超快速I/O .....	270
26.2.3 超快速配置 .....	270
26.2.4 更多的硬IP .....	271
26.2.5 模拟与混合信号器件 .....	271
26.2.6 ASMBL与其他结构 .....	272
26.2.7 不同的结构粒度 .....	272
26.2.8 ASIC结构中的嵌入式 FPGA内核 .....	273
26.2.9 ASIC和FPGA结构中嵌入 FPNA内核或者相反 .....	273
26.2.10 基于MRAM的器件 .....	273
26.3 设计工具 .....	273
26.4 期待意外的发生 .....	274
<b>附录A 信号完整性简介 .....</b>	<b>275</b>
<b>附录B 深亚微米延迟效应 .....</b>	<b>285</b>
<b>附录C 线性移位寄存器 .....</b>	<b>299</b>
<b>术语表 .....</b>	<b>310</b>
<b>索引 .....</b>	<b>326</b>

# 第1章 概论

## 1.1 什么是FPGA

现场可编程门阵列（FPGA）是由可配置（可编程）逻辑块组成的数字集成电路（IC），这些逻辑块之间用可配置的互连资源。设计工程师可以对这类器件进行（编程）配置来完成各种各样的任务。

由于实现方式不同，有些FPGA只能编程一次，而有些FPGA可以重复多次编程。自然，只能编程一次的器件被称为一次性可编程（OTP）器件。

FPGA名称中的“现场可编程”是指编程“在现场”进行。（与那些内部功能已被制造商固化的器件正相反。）这意味着FPGA可能是在实验室中配置的，也可能是在对已应用于外部世界的电子系统中某个设备功能的改进。如果一个器件能够在某个高层系统有编程的能力，它就是在系统可编程（ISP）器件。

## 1.2 FPGA为什么令人感兴趣

数字集成电路（IC）有很多种类型，包括所谓的“jelly-bean”逻辑（指由一些简单、固定的逻辑功能组成的小元件）、存储器件和微处理器。但是，我们特别感兴趣的是可编程逻辑器件（PLD）、专用集成电路（ASIC）、专用标准部件（ASSP），当然，还有FPGA。

就我们讨论的这部分内容而言，PLD这个术语包括简单可编程逻辑器件（SPLD）和复杂可编程逻辑器件（CPLD）。

第2章和第3章将更详细地讨论不同类型的PLD、ASIC和ASSP，眼下我们只需要了解，PLD是一种内部架构已经由制造商确定、但可由工程师通过在现场配置（编程）来实现多种不同功能的器件。但是，和FPGA相比，这种器件包含的逻辑门相对有限，它们可以用来实现的功能也更少、更简单。

与PLD对应的是ASIC和ASSP，它们拥有数百万计的逻辑门，可以用来产生令人难以置信的强大复杂功能。ASIC和ASSP基于相同的设计流程和制造工艺，都是定制设计来满足专门的应用。它们之间仅有的不同就在于，ASIC是专为某个特定公司的使用而设计制造的，而ASSP的目的是出售给多个顾客。（今后，在我们使用ASIC一词时，它可能是指ASSP，除非另外指出或此处的解释不符合上下文。）

尽管ASIC在尺寸（晶体管的数量）、复杂性和性能实现上都是最佳的，但是设

计和制造它是一个很耗费时间和代价昂贵的过程，还有一个不足，就是最终设计是固定的，不开发新版本，就无法调整。

**2** FPGA因而弥补了PLD和ASIC之间的空白。因为它可以像PLD那样在现场编程，同时具有数百万的逻辑门<sup>①</sup>，可以用来实现很大、很复杂的功能，而这是从前只有使用ASIC才可以实现的。

FPGA设计的成本要大大低于ASIC设计的成本（尽管大规模生产时ASIC部件肯定要比FPGA便宜很多），同时，在FPGA中对设计修改要容易很多，而且这种设计的产品上市时间会早很多。FPGA催生了许多富有创新意识的小公司（当然FPGA也为很多大的系统设计公司所使用），因为它给“小规模开发”提供了便利。这意味着某个工程师或小工程团队可以在基于FPGA的测试平台上实现他们的软件或硬件想法，而不必承担数额巨大的不可重现工程（NRE）成本或采购昂贵的ASIC软件开发工具。因此，据估计在2003年只有1500家到4000家ASIC设计公司<sup>②</sup>和5000家ASSP设计公司（这些数字近年来正在显著下降）；相反，同年估计有大约45万家FPGA设计公司<sup>③</sup>。

### 1.3 FPGA的用途

20世纪80年代中期，FPGA刚出现时，大部分用来实现粘合逻辑<sup>④</sup>、中等复杂度的状态机和相对有限的数据处理任务。在20世纪90年代早期，FPGA的规模和复杂度开始增加，那时它们的主要市场在通信和网络领域，这都涉及大量数据的处理。后来，到了20世纪90年代末，FPGA在消费、汽车和工业领域的应用经历了爆炸式增长。

FPGA经常用于制作ASIC设计的原型或是提供一个硬件平台来验证一个新算法的物理层实现。但是，FPGA的低开发成本和短上市时间意味着越来越多公司使用它制造最终的产品。（实际上一些主要的FPGA厂商就拥有直接与ASIC在某些特定市场上竞争的器件。）

到了21世纪早期，已经可以买到有数百万门容量的高性能FPGA。这些器件中有

<sup>①</sup> 在FPGA概念中，哪些器件包含“逻辑门”这个概念有些模糊。这将在第4章详细讨论。

<sup>②</sup> 该数字不太确切，这取决于所指的具体对象。

<sup>③</sup> 这些数字难以确定的另外一个原因是，人们在到底什么是“设计开始”上不容易达成一致。例如，在ASIC设计中，“设计开始”应该包括中途取消的设计，还是仅仅考虑那些最终完成流片的设计？对于FPGA，由于FPGA的可重配置的特性，这个问题几乎要变得没有意义了。或许还应该讲一讲下面的事，在给我介绍了一个以FPGA为主的工业评论网站之后，一名来自FPGA厂商的代表补充说：“其实这里给出的数字不是非常准确。”当我问为什么时，他露出了狡猾的笑容：“主要是因为我们不能给他提供很好的数据！”

<sup>④</sup> 粘合逻辑是指用来连接（“粘合”）大逻辑块、功能块或器件的相对少量的简单逻辑（以及作为它们之间的接口）。

的内嵌了微处理器核、高速输入/输出（I/O）接口和类似的模块。这样，今天的FPGA几乎可以用来实现任何东西，包括通信设备和由软件定义无线电；雷达、影像和其他数字信号处理（DSP）的应用；直至包含硬件和软件的片上系统（SoC）<sup>①</sup>。

说得更具体一些，FPGA正在蚕食4个主要的市场：ASIC和定制硅、DSP、嵌入式微处理器应用和物理层通信芯片。另外，FPGA还创建一个属于自己的新市场：可重配置计算（RC）技术。

- ASIC和定制硅：正如前面部分提到的，今天的FPGA正逐渐用来实现各种设计，而从前这些设计仅能由ASIC和定制硅来完成。
- 数字信号处理：高速DSP传统上是用数字信号处理器来实现的，实际上这是一种特殊剪裁的微处理器。但是，今天的FPGA可以包含内嵌的乘法器、专用计算例程和大量的片上RAM等所有DSP操作所需的特性。这些特性再加上FPGA提供的并行性，其结果就是比最快的DSP芯片还要快上500倍乃至更多。
- 嵌入式微处理器：小的控制功能传统上是用称作微控制器的专用嵌入式微处理器处理的。这些低成本器件由围绕在处理器核外的片上程序和指令存储器、定时器和I/O组成。但是，随着FPGA的价格不断下降，甚至最小的器件都足以实现一个集成有可选定制I/O功能的软处理器核。结果就是FPGA对嵌入控制应用越来越具有吸引力。
- 物理层通信：FPGA长期以来用于实现物理层通信芯片和网络协议层互连的粘合逻辑。今天的高端FPGA拥有了多种高速收发器，这意味着通信和网络功能可以合并到同一设备中。
- 可重配置计算技术：这是指由FPGA提供的固有的并行性和可重配置性来实现软件算法的“硬件加速”。许多公司正在建立大型的以FPGA为基础的可重配置计算引擎，来完成从硬件仿真到密码分析，再到开发新药物的任务。

## 1.4 本书内容

任何从事过电子设计或电子设计自动化领域的人都知道，近年来这些领域正变得越来越复杂，FPGA也不例外。

早些时候（在20世纪80年代中期）还不那么复杂，当时FPGA才刚刚出现。最早的FPGA只包含数百个简单的逻辑门，设计这些元件的流程——主要是基于原理图——也容易理解和使用。相比之下，今天的FPGA极其复杂，设计工具、流程、工艺多得令人眼花缭乱。

<sup>①</sup> 尽管片上系统（SoC）一词倾向于表示在单个器件上的完整电子学系统，现状是人们无一例外地还需要额外的元件。因此，更准确的表达可能是片上子系统（SSoC）或片上部分系统（PoaSoC）。

本书的开篇介绍了基本的概念和目前常见的几种FPGA的架构和器件。然后研究各种设计工具和流程，具体应用哪种取决于设计工程师的需要。接着，除了了解FPGA的内部组成之外，本书也考虑了将器件与同一个电路板内的系统其余部分集成时的各种问题，包括对最近才出现的吉比特接口的讨论。

## 1.5 本书不包括什么

本书内容不针对特定的FPGA厂商或特殊的FPGA器件，因为新的特性和芯片类型出现得很快，以至于在本书出版之前（有时是在我撰写完相关文字以前），这里的相关内容已经过时了。

与之类似，本书尽可能地（在这样做有意义的范围内）不提及具体EDA厂商或他们的工具名称，因为这些厂商经常彼此并购，改变其公司名字——换句话说，让这些名字变得奇形怪状——或者变更他们的设计和分析工具的名称。类似地，事情发展得如此迅速，这个行业很少说“*A*工具有这个特性，而*B*工具没有”，因为可能仅仅几个月，*B*工具已经得到增强，而*A*工具已经被淘汰了。

由于上述原因，本书主要介绍不同风格的FPGA器件和多种设计工具的概念以及流程，但不涉及哪个FPGA厂商支持哪种架构，哪个EDA厂商和工具支持哪种专门特性的问题，这留给读者去研究（在第6章列出了有用的Web站点）。

## 1.6 读者对象

本书面向大范围的读者，包括小型FPGA设计顾问、大系统机构中的硬件和软件设计工程师、向FPGA领域转型的ASIC设计者、开始使用FPGA的DSP设计者、高等院校相关专业学生、EDA和FPGA公司的销售人员、市场人员和其他人员、评论专栏和杂志社的编辑们。

# 第2章 基本概念

## 2.1 FPGA的核心

FPGA与ASIC的关键区别，也就是它们存在的主要原因，这在它们的名称当中得到了很好的体现：

Field Programmable Gate Array  
现场可编程门阵列

关键在于，为了达到可编程的目的，我们需要某种机制，来允许我们配置（编程）一个预制的硅芯片。

## 2.2 简单的可编程功能

在进一步讨论之前，我们先来考虑一个很简单的可编程功能，它有两个输入 $a$ 和 $b$ ，还有一个输出 $y$ （图2-1）。

9

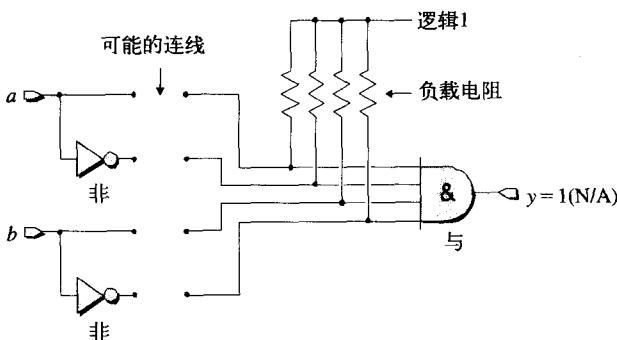


图2-1 简单的可编程功能

反相器（非门）与输入相连，意味着每个输入都可以获得它的原值（未改变的）和补码（反相）形式。观察潜在连线的位置。在这些潜在连线的空白处，与门的所有输入通过负载电阻与逻辑1相连，其值被置为逻辑1。这意味着输出 $y$ 将总被驱动为逻辑值1，这使得这个电路很无聊地保持着当前的状态。为了使这个功能更加有趣，我们需要一些灵巧的机构来帮助建立一个或更多可能的连线。

## 2.3 熔丝连接技术

最早为人们所知的一种允许用户对他们自己的器件进行编程的技术是熔丝连

接技术。在这种情况下，器件被制造成带有可熔的连接线路，这些连接线路被称为熔丝（图2-2）。

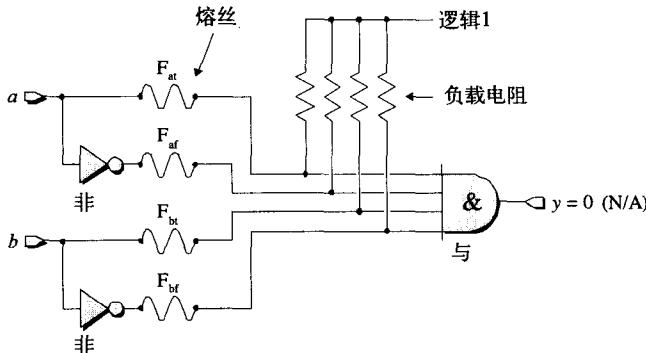


图2-2 用尚未编程的可熔连接来扩充器件

这些熔丝在概念上与你在家用产品中（比如电视）可以找到的保险丝类似。当  
10 有任何危险情况（如电视变得功率过大）发生时，它的保险丝将会熔断。结果是电路开路，使剩下的部分免于受损。显然，在硅片中的熔丝是通过与产生片上创建晶体管和导线相同的过程形成的，因此它们的尺度小到在显微镜下才可以看到。

当一名工程师买到一个基于熔丝连接技术的可编程器件时，所有的熔丝都没有动过。这就是说，处于它们的未编程状态，从示例电路中输出来的将一直是逻辑0。（任何时候与门输入一个0将会使它的输出为0，所以如果输入a是0，与门的输出将是0。相反，如果输入a是1，那么它的非门的输出——我们称之为 $\neg a$ ——将是0，与门的输出会又一次是0。类似的情形也发生在输入b上。）

此处的关键在于，设计者可以通过对器件的输入施以相对大的电流和电压的脉冲，有选择地除去不需要的熔丝。例如，考虑移除熔丝 $F_{at}$ 和 $F_{bt}$ 后将发生什么（图2-3）。

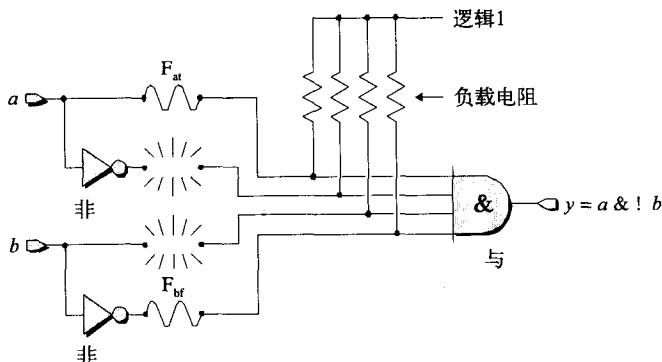


图2-3 移除熔丝连接（编程）

移除这些熔丝使输入  $a$  的补码和输入  $b$  的原值断开了和与门的连接（与这些信号相连的负载电阻使它们对与门的输入逻辑值成为1）。这样器件表现为新的功能，即  $y = a \& !b$ （等式中的符号“ $\&$ ”表示“与”，而符号“ $!$ ”表示“非”，这种语法将在第3章中详细讨论）。像这样把熔丝熔断一般是指对器件进行编程的过程，但是也可能是指熔断熔丝或烧毁器件。

基于熔丝连接技术的器件被称为一次性可编程器件，即OTP，因为一旦熔丝熔断，将不能被替代或恢复原状。

显然，现代FPGA基于一系列广泛的可编程技术，但熔丝连接方法并不包含在其中。在这里要提到它，是因为它为后续内容提供了背景，而且它在第3章将要提到的早期器件技术中还是适用的。

公元前2500年，美索不达米亚人发明了焊接，来连接金片。

## 2.4 反熔丝技术

与熔丝连接技术相反，我们还有反熔丝技术，即在每个可配置路径上都有称为反熔丝的连接。当处于未编程状态时，反熔丝的电阻是如此之高以至于可视为开路，如图2-4所示。

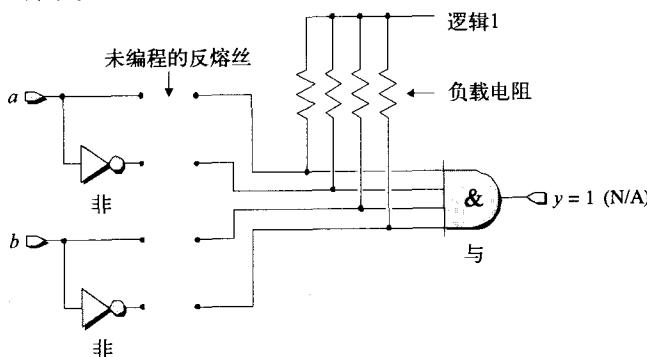


图2-4 未编程的反熔丝连接

这就是器件刚买回来时的状态。但是，通过对器件的输入加以相当大的电压和电流的脉冲有选择地使反熔丝“变长”（编程）。例如，如果我们加在输入  $a$  的补码和输入  $b$  的原值所对应的反熔丝上，器件将表现为函数  $y = !a \& b$ （图2-5）。

反熔丝开始时是连接两个金属连线的微型非晶硅柱。在未编程时，非晶硅表现为一个电阻超过  $10^9\Omega$  的绝缘体（图2-6a）。

对这个特殊元件的编程行为有效地“形成”了连接，实际上，通过将绝缘的非晶硅转化成导电的多晶硅而产生了通孔（via）（图2-6b）。

11

12

13

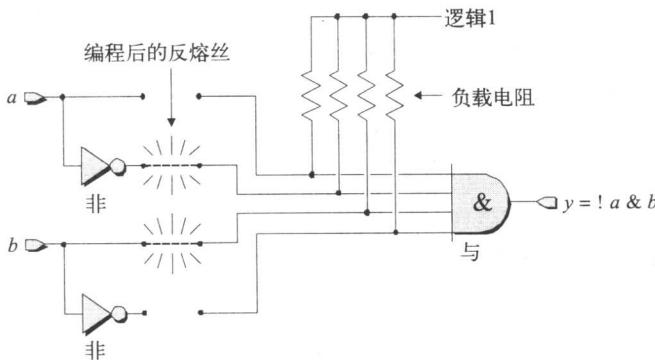


图2-5 编程后的反熔丝连接

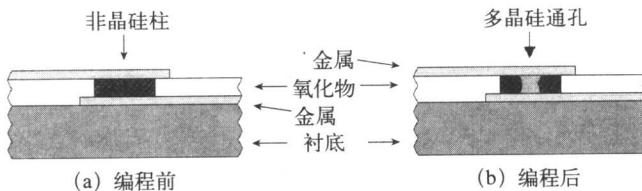


图2-6 反熔丝的编程

自然，基于反熔丝技术的器件也是一次性编程器件，因为一旦反熔丝生成，它将不能被移除，因而你的想法也不能再改变了。

## 2.5 掩模编程器件

深入讨论之前，了解一些背景可以帮助我们理解将要遇到的名词术语。电子系统（特别是计算机）一般利用两种主要的存储设备：只读存储器（ROM）和随机存取存储器（RAM）。

ROM是非易失的，因为系统断电后它们的数据仍然保留着。系统中的其他组成设备可以从ROM中读取数据，但是不能写入新的数据。相比之下，数据不但可以读出而且可以写入RAM设备中。由于系统断电时数据将会丢失，所以RAM是为易失的。

基本的ROM也是掩模可编程的，因为存储的所有数据都固化在它的结构中，这是通过光掩模的方法产生出晶体管和金属走线（称为金属化层）并把它们连接到硅芯片时进行。例如，一个以晶体管为基础的ROM单元保存了一位数据（图2-7）。

光掩模的概念和在制造硅芯片中的使用的更多细节可以参考*Bebop to the Boolean Boogie (An Unconventional Guide to Electronics)*, ISBN 0-7506-7543-8。

整个ROM包括了大量的行（字）和列（数据），形成了一个阵列，每一列都有负载电阻使之保持弱逻辑1值，每个行列的交叉有一个关联晶体管和一个（可能的）掩模编程连接。

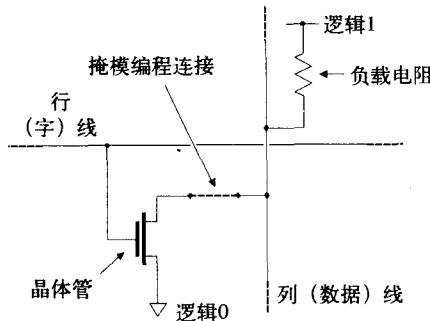


图2-7 以晶体管为基础的掩模编程ROM单元

大部分ROM可以预制造，同样的基础架构可以用于不同的用户。当为了特定客户的需求而定制这种器件时，使用一个单独的光掩模来确定哪些单元包括有掩模编程连接，哪些单元制造出来后没有这样的连接。

现在设想一下，当中的一行进入工作状态将会发生什么，这样它会试图激活所有与这一行相连的晶体管。当一个单元包括掩模编程连接时，激活这个单元的晶体管将把列线通过晶体管连接到逻辑0，在外界看来这个列显示的值将是0。相反，当一个单元不包括掩模编程连接时，单元的晶体管不发生作用，相连的负载电阻使列线保持在逻辑1，这就是向外提供的值。

## 2.6 PROM

掩模编程器件的问题是，除非生产量相当大，否则代价将非常高昂。而且，如果开发过程中经常修改里面的内容，这种元件几乎就毫无用处了。

由于这个原因，1970年Harris半导体公司开发出了第一个可编程只读存储器(PROM)，这种器件中引入使用了以镍为基础的熔丝连接技术。作为一般例子，设想一个以晶体管—熔丝为基础的简化PROM单元（图2-8）。

在厂商提供的未编程状态下，器件上所有的熔丝都完好保留着。在这种情况下，当行（字）线处于工作状态时，所有与之相连的晶体管都会导通，这样列（数据）线将通过各自的晶体管被下拉到逻辑0。但是，正如我们先前所讨论的，设计者对器件的输入加以大的电流和电压脉冲就可以有选择地擦除不需要的熔丝。当熔丝被擦除后，相应的单元就显示为逻辑1。

这些器件最初主要作为存储器来存放计算机程序和常数值（因此它们有了“只读存储器”的称号）。但是工程师们发现它们也可以实现简单的逻辑功能，如

16 查找表和状态机。因为PROM相对低廉，所以它可以用来简单地烧写后插入系统中，来修正错误或测试新的设备。

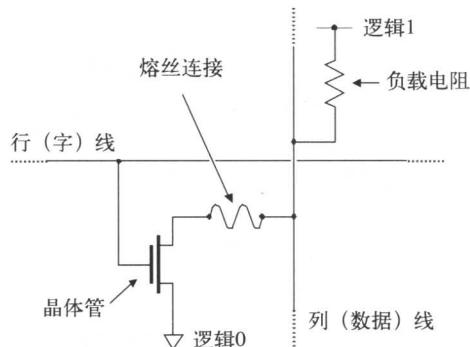


图2-8 以晶体管—熔丝连接为基础PROM的单元

此后，各种基于熔丝连接技术和反熔丝技术的通用PLD也纷纷面世了（详情参见第3章）。

## 2.7 基于EPROM的技术

前面提到，基于熔丝连接技术和反熔丝技术的器件只能编程一次——如果你擦除（或生成）了熔丝，设计想法就来不及改变了。（某些情况下，可以增量修改设计，即擦除或生成额外的熔丝，但是这需要运气。）因此，人们开始想，如果有某种方法可以制造能够编程、擦除和用新的数据重编程的器件，将会多么方便。

一个备选方案是称为可擦除可编程只读存储器（EPROM）的技术，Intel在1971年引入了首个这样的器件（1702）。EPROM晶体管和标准的MOS晶体管具有相同的基本结构，但是多了由氧化层绝缘的多晶硅浮置栅（图2-9）。

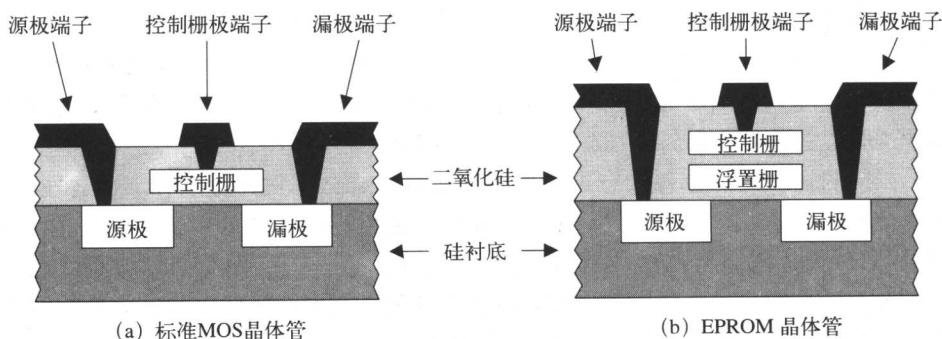


图2-9 标准MOS晶体管与 EPROM 晶体管

当它处于未编程状态时，浮置栅不带电，也不影响控制栅的一般操作。为了对晶体管编程，要在控制栅和漏极端子之间加大电压（数量级为12V）。晶体管在高压之下，高能电子穿越氧化层进入浮置栅，这一过程称为热（高能）电子注入。当编程信号撤销后，负电荷储存在浮置栅中。这些电荷将非常稳定，在正常运行状态下可以保持10年以上。浮置栅里储存的电荷阻止了控制栅的正常操作，这样，就可以把已经编程的单元和没有编程的单元区分开。这意味着我们可以使用这样的晶体管来形成存储单元（图2-10）。

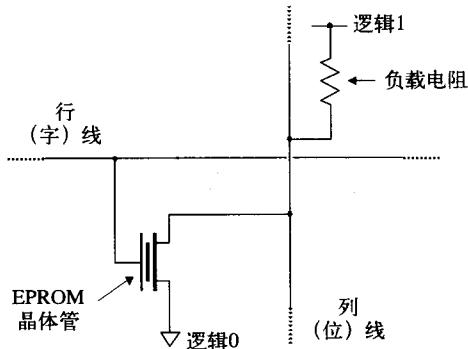


图2-10 一个以EPROM晶体管为基础的存储单元

可见，这个单元不再要求熔丝、反熔丝或掩模可编程连接。在制造商预留的未编程状态下，EPROM晶体管中所有的浮置栅不带电荷。这种情况下，字线激活后把所有与字线相连的晶体管接通，于是所有的位线通过相应的晶体管下拉到逻辑0。为了对器件编程，工程师有选择地使晶体管浮置栅带电，使晶体管失效，这些单元将显示逻辑1值。

因为EPROM单元比熔丝小一个数量级，所以它们非常高效。但是，它们的名气主要还是因为可以被擦除和重编程。一个EPROM单元可以通过将浮置栅放电来擦除，放电所需的能量由一个紫外（UV）射线源提供。交付时的EPROM器件是在陶瓷或塑料封装里，顶部开着一个小晶体窗口，这个窗口通常用不透明的胶带盖着。在器件需要擦除时，把它从电路板中取下来，把晶体窗口上的带子揭开，放进包含强烈紫外线源的容器里。

EPROM器件的主要问题是，它们带有晶体窗口的封装太昂贵，而且擦除它们所花费的时间多达20min。可以预见到未来的器件会存在与制造工艺的进步而使晶体管越来越小有关的问题。由于器件的内部结构越来越精细，器件的密度（晶体管和互连的数量）增加，芯片表面的大部分被金属覆盖。这使EPROM单元吸收紫外线照射更加困难，要求的暴露时间也增加了。

这些器件最初同样是用于可编程存储器（因此它们名字中包含PROM）。但此

后类似技术被用于更通用的PLD，因此这样的PLD后来被称为可擦除PLD (EPLD)。

## 2.8 基于EEPROM的技术

下一个技术进步是电可擦除可编程只读存储器的形式 (EEPROM或E<sup>2</sup>PROM)。

19 一个E<sup>2</sup>PROM单元大约是同等EPROM单元的2.5倍，因为它包括两个晶体管和这两个晶体管之间的距离 (见图2-11)。

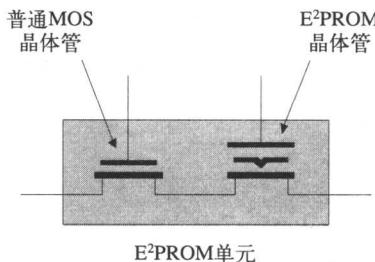


图2-11 E<sup>2</sup>PROM单元

E<sup>2</sup>PROM晶体管与EPROM晶体管相似，也包含浮置栅，但是围绕这个栅的绝缘氧化层非常薄。第二个晶体管可以用来擦除单元的电荷。

E<sup>2</sup>PROM最初问世时是作为计算机存储器，然而同样的技术继而用在了PLD上，所以称为电可擦除PLD (EEPLD或E<sup>2</sup>PLD)。

## 2.9 基于闪存的技术

一种名为闪存的技术进展可以追溯到EPROM和E<sup>2</sup>PROM技术。名称“闪”(FLASH)准确反映出这种技术的删除时间要比EPROM快得多。基于闪存的元件可以采用多种结构。一方面与EPROM单元类似的具有一个浮置栅晶体管单元，但是另一方面还具有E<sup>2</sup>PROM器件的薄氧化层特性。因此，这些器件尽管是电可擦除的，但是只能整个清除或是擦除大部分。其他方面的结构性质与拥有两个晶体管的E<sup>2</sup>PROM单元相似，从而能够在以字为顺序的基础上擦除和重编程。

20 早期的闪存每个单元只能存储一位数据。但是到了2002年，工艺专家们已经实验了大量的不同方法来增加这个容量。一种工艺涉及在闪存晶体管中存储一定量水平的电荷来表现每个单元两位。另一个可能的途径涉及在低于栅的层创造两个隔离的结点，作为每个单元中两位的载体。

## 2.10 基于SRAM的技术

半导体RAM有两种主要类型：动态RAM (DRAM) 和静态RAM (SRAM)。

对于DRAM，每个单元由一个晶体管—电容对构成，非常节省硅片面积。使用修饰词“动态”是因为电容经过一段时间会丢失电荷，所以如果需要保持它的数据，每个单元必须周期性地补充电荷。这种操作——被称为刷新——比较复杂，需要大量额外的电路。当这个刷新电路的成本摊到DRAM存储设备的数千万位时，这种办法就变得非常物有所值。但是，DRAM技术对于可编程逻辑来说仍然缺乏吸引力。

相应的，与SRAM关联的修饰词“静态”用在这里是因为：一旦SRAM单元把值载入，它将保持不放电，除非被专门修改或者整个系统断电。考虑这个以SRAM为基础的可编程单元的示意图（见图2-12）。

整个单元包括一个多晶体管SRAM存储元件，此元件的输出驱动着一个额外的控制晶体管。取决于存储元件的内容（逻辑0或逻辑1），控制晶体管将会是关（OFF）或开（ON）。

基于SRAM单元的可编程器件有一个缺点，就是每个单元消耗大量的硅片面积，因为这些单元是由4个或6个晶体管配置成一个锁存器而形成的。另一个缺点是器件配置的数据（编程后状态）在系统断电后将会丢失。这意味着这种器件在系统上电时需要被重新编程。然而这种器件又具有相应的优点：可按需迅速和反复地编程。

以SRAM为基础的FPGA是如何使用这些单元的，将在后面的章节里进一步讨论。在这里对于我们来说，只需能够在概念上替换图2-2示例电路中的熔丝、图2-4中的反熔丝或图2-7里ROM单元有关的晶体管和有关掩模可编程连接（当然，对于后者，以SRAM为基础的ROM是无意义的）即可。

## 2.11 小结

表2-1显示了各种器件以及与之相关的各种主要的可编程技术。

另外，我们不应该忘记新的技术仍然在不断涌现。有的如昙花一现，然后在你还未注意它时就消失了，有的突然出现在台前，快得让你不知道它们是从哪里来的。

例如，一种叫做磁性RAM（MRAM）的技术近期吸引了大量的目光。这种技术可追溯到1974年，当时IBM开发了一种叫做磁性隧道结（MJT）的元件。它包括一个夹层结构，是用一个薄绝缘层隔开两个铁磁层构成的。两个沟道（称为字线和位线）之间是MJT夹层结构，沟道的交叉就可以产生一个MRAM单元。

MRAM单元具有结合SRAM的高速、DRAM的存储能力和闪存的非易失性的潜力，当然这些都需要消耗一定量的电能。2005年，以MRAM为基础的存储芯片

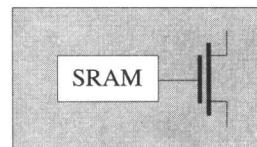


图2-12 SRAM为基础的可编程单元

21

22

23 投入商用。预测以MRAM为基础的FPGA等器件可能会在不远的将来出现。

表2-1 可编程技术总结

技 术	符 号	与之相关的主要器件
熔丝	—~—	SPLD
反熔丝	—□—	FPGA
可擦除PROM	—↑—	SPLD 和 CPLD
电可擦除PROM / 闪存	—↑—	SPLD和 CPLD (和一些FPGA)
静态随机存储器		FPGA (和一些CPLD)

# 第3章 FPGA的起源

## 3.1 相关的技术

为了对FPGA的形成及其原因有更好的理解，将其置于其他相关技术的背景中是有益的（图3-1）。

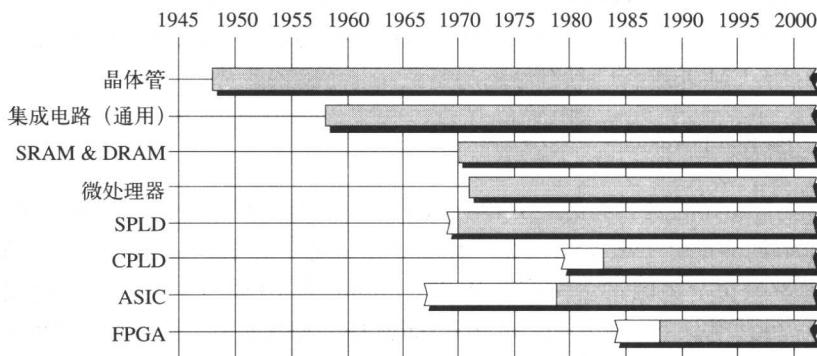


图3-1 技术发展时间路线图

在图3-1中，时间条的白色部分表明，尽管这些技术在早期已经出现了，但是出于种种原因，还不能够被这一时期的主流工程师们接受。比如，尽管Xilinx在1984年就发明了第一个FPGA，但设计工程师们并没有真正开始使用这些小玩意，而是将其扔在一旁，直到20世纪90年代早期才开始使用。

25

## 3.2 晶体管

1947年12月23日，在美国贝尔实验室工作的物理学家William Shockley、Walter Brattain和John Bardeen成功地创造出了第一个晶体管：一个锗（化学符号Ge）制造的点接触器件。

1950年，出现了更加复杂的、称为双极型晶体管（BJT）的器件。它的制造更为容易，造价低廉，具有更高的可靠性。到了20世纪50年代末期，晶体管的制造已经更多地使用硅（化学符号Si）而不是锗。虽然锗在电性能上具有明显优势，但是硅更为廉价而且更易于处理。

如果把双极型晶体管以特定方式相连，形成的数字逻辑门被称为晶体管—晶

体管逻辑（TTL）。同样的晶体管以另外的方式相连后，就形成发射级耦合逻辑（ECL）。TTL构成的逻辑门速度快而且具有很强的驱动能力，但是它消耗的电能也比较大。ECL构成的逻辑门比TTL构成的逻辑门要快很多，但是它要消耗更多的电能。

1962年，Steven Hofstein 和 Fredric Heiman 在新泽西州普林斯顿的RCA研究实验室，发明了一系列新的器件，叫做金属氧化物半导体场效应晶体管（MOSFET），通常简称为FET。早期的FET在某种程度上要比它们的双极型兄弟慢一些，但是它们更廉价、尺寸更小而且耗电更低。

**26** FET有两种主要类型，叫做NMOS和PMOS。由NMOS和PMOS晶体管以互补方式构成的逻辑门就是我们所熟知的互补金属氧化物半导体（CMOS）。用CMOS实现的逻辑门比TTL实现的速度要慢一些，但是这两种技术在相当程度上是等价的。然而，CMOS逻辑门具有静态功率消耗非常低的优势。

### 3.3 集成电路

第一个晶体管是作为分立元件提供的，独立封装在一个小金属外壳中。后来，人们开始觉得把整个电路制造在一片半导体上是个好主意。这个想法的第一次公开讨论要归功于英国雷达专家G.W.A.Dummer在1952年发表的一篇论文。直到1958年，德州仪器公司（TI）的Jack Kilby成功地把包括5个元件的移相振荡器制作在同一片半导体上。

大约与Kilby开发工作的同一时期，两位Fairchild半导体公司的创建者（瑞士物理学家Jean Hoerni和美国物理学家Robert Noyce）发明了衬底光学印刷技术，现在这种技术广泛用于制造现代IC中的晶体管、绝缘层和互连线。

在20世纪60年代中期，TI公司设计制造了大量的基本设计组件IC，称为54xx和74xx系列，它们分别面向军事和商业领域。这些小器件，一般约为 $3/4\text{in}^{\circ}$ 长、 $3/8\text{in}$ 宽，有14或16个管脚，每个包含少量的简单逻辑（准确地说，这种器件还可能更长、更大、具有更多的管脚）。例如，7400器件包括4个二输入NAND门，7402包括4个二输入NOR门，7404包括6个NOT（反相）门。

**27** TI公司的54xx和74xx系列是用TTL实现的。相应地，1968年RCA开发了等效的以CMOS为基础的器件——称为4000系列。

### 3.4 SRAM/DRAM和微处理器

20世纪60年代末期和70年代早期是数字IC领域大发展的时期。例如，1970年Intel宣布第一个1024位DRAM（1103），与此同时，Fairchild公司开发了第一个

<sup>①</sup>  $1\text{in} \approx 2.54\text{cm}$ 。——编者注

256位SRAM（4103）。

一年后的1971年，Intel公司开发了世界上第一个微处理器4004，是由Marcian “Ted” Hoff、Stan Mazor和Federico Faggin设想并开发的。它也被称为“片上计算机”，4004包含有大约2300个晶体管，每秒可以执行60 000次操作。

实际上，尽管4004被大多数文献记录为第一个微处理器，但存在争议。1968年2月，International Research Corporation开发了一种他们称为“片上计算机”的体系结构。并且在1970年12月，4004面世的一年前，一个名叫Gilbert Hyatt的人申请了名为“单片集成电路计算机体系结构”的专利（关于这个专利的争论持续到现在）。不管争执的结果怎样，但是，事实上4004是第一个在物理上实现的、投入市场并且实际应用的微处理器。

我们在这里对SRAM和微处理器技术感兴趣的原因是，今天FPGA主要是基于SRAM技术的，而且目前某些高端的器件内嵌了微处理器核（这两个主题将在第4章做进一步的讨论）。

## 3.5 SPLD和CPLD

第一种可编程IC一般被称为可编程逻辑器件（PLD）。这种器件最初于1970年以PROM的形式进入人们视野，当时还相当简单。仅仅到了20世纪70年代的末期，复杂高级得多的产品就已经实用化了。为了把它们与并不复杂精密的前辈相区别，这些新器件被命名为复杂可编程逻辑器件（CPLD），并沿用至今。毫不奇怪，它的前辈通常被称为简单可编程逻辑器件（SPLD）。

有的人把PLD和SPLD理解为同义词，而有的人则把PLD视为既包括SPLD也包括CPLD的超集。除非特别指出，我们这里采用后一种解释。

工程师们喜欢使用相同的缩写来表示不同的事，或者是用不同的缩写来表示相同的事（听一群工程师彼此愉快地谈话甚至会让最坚强的人“变得怒不可遏”）。例如，对于SPLD，基础结构有许多种，而其中不少都有3个或4个字母的不同组合缩写（图3-2）。

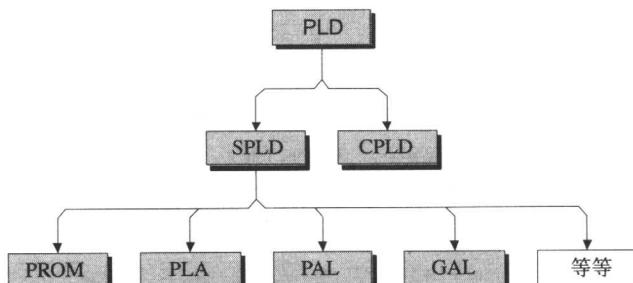


图3-2 PLD的谱系

29

当然，这些器件中还会有EPLD、E<sup>2</sup>PLD和闪存的版本——比如，EPROM和E<sup>2</sup>PROM——但是为了简洁起见，它们在图3-2中被忽略了（这些概念在第2章有介绍）。

### 3.5.1 PROM

第一个简单可编程逻辑器件是PROM，出现于1970年。对于这些器件是如何完成它们的魔术，一种直观的办法就是把它看作由AND阵列函数驱动可编程的OR阵列函数。例如，考虑一个3输入、3输出PROM（图3-3）。

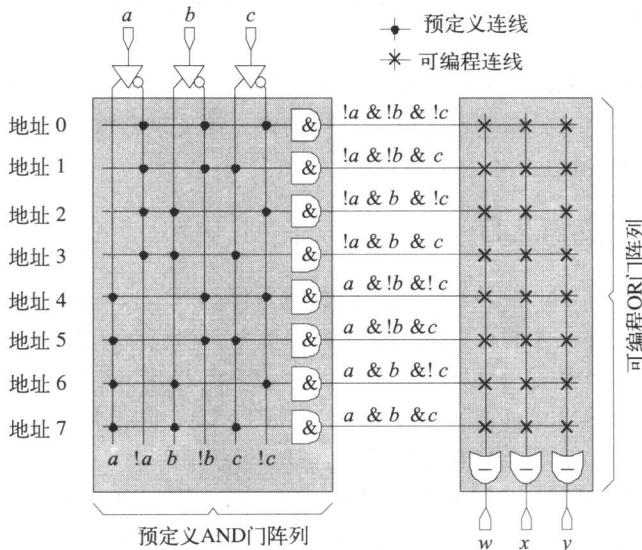


图3-3 未编程的PROM（预定义的AND门阵列，可编程OR门阵列）

30

在OR门阵列中的可编程连线可以用熔丝，或分别使用EPROM和E<sup>2</sup>PROM器件中的EPROM晶体管和E<sup>2</sup>PROM单元来实现。请注意，此图仅仅是为了了解我们提到的实例器件如何工作，它并不代表实际的电路图。在现实中，AND门阵列中的每个AND函数都有3个输入，分别是器件输入 $a$ 、 $b$ 和 $c$ 的适当原值或补码。类似地，OR门阵列的每个OR函数有8个输入，分别由AND门阵列提供。

正如前面所提到的，PROM最初作为计算机存储器来存储程序指令和常量数据。然而，设计师们也使用它们来实现简单的逻辑功能，例如查找表和状态机。实际上，PROM可以用来实现任何组合逻辑块，只要它没有太多的输入和输出。例如，图3-3所示简单的3输入、3输出PROM可以实现任何不超过3个输入和3个输出的组合逻辑。为了理解这是如何实现的，思考一下图3-4所示的小逻辑块（这个电路显然没有超出本例的设想）。

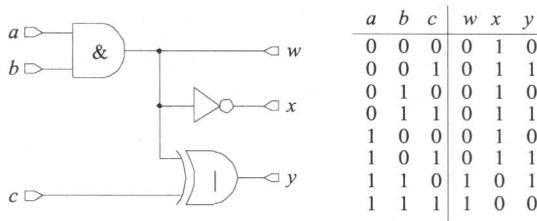


图3-4 简单的组合逻辑块

我们可以用3输入、3输出PROM来替换这个逻辑块。只需要对OR门阵列中适当的连线编程（图3-5）。

注意这个图中的等式，“ $\&$ ”表示AND，“ $\mid$ ”表示OR，“ $\wedge$ ”表示XOR，而“ $!$ ”表示NOT。这种语法在PLD的早期非常普遍，因为它允许使用标准计算机键盘在文本文件中简单且精确的表达逻辑等式。

当然，上面的例子非常简单。真正的PROM拥有很多的输入和输出，因而可以实现很大的组合逻辑块。从20世纪60年代中期到20世纪80年代中期（或更晚一些），组合逻辑普遍采用TI 74xx系列器件那样的小IC来实现的。31

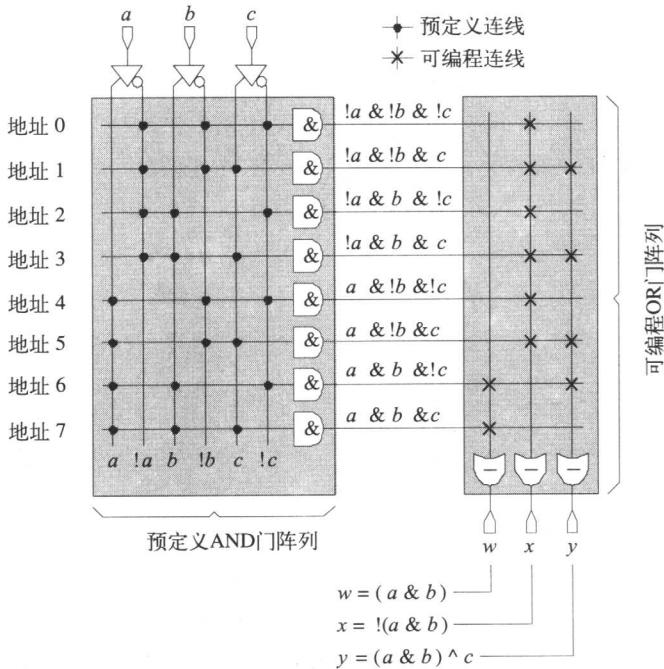


图3-5 编程后的PROM

实际上，这些小IC芯片中的大部分都可以用一个单一的PROM来代替，这样

电路板会更小、更轻、更便宜且更不易出错（每个电路板上的焊点都可能失效）。而且，如果后来发现设计中的这个部分出现逻辑错误（例如，设计者可能粗心地把AND函数当做了NAND），那么这个过失很容易地以通过烧写一个新的PROM来弥补（或者对EPROM和E<sup>2</sup>PROM擦除和重编程）。这比基于小IC的电路板必须先修正错误更为可取。这包括在电路板上增加新器件，用刀来切出线道，以及为其余电路手工添加与新器件的连线。

在逻辑中，众所周知AND（“&”）表示逻辑相乘或积，而OR（“|”）表示逻辑相加或和。而且，如果以下逻辑式成立，

$$y = (a \& !b \& c) | (!a \& b \& c) | (a \& !b \& !c) | (a \& !b \& c)$$

则文字表示变量的原值或补码（ $a$ 、 $!a$ 、 $b$ 、 $!b$ 等），用“&”操作符相连的一组文字（操作数）代表一个乘积项。这样，乘积项（ $a \& !b \& c$ ）包括三个文字 $a$ 、 $!b$ 和 $c$ ，以上的等式被称为乘积和形式。

当使用它们来实现图3-4或图3-5所示的组合逻辑时，PROM可以满足方程式所要求大量的乘积项，而且每个输入组合都可以解码和使用，所以它们的输入相对要少。

### 3.5.2 PLA

由于PROM结构对地址的限制，可编程逻辑器件（PLD）的下一步演化是可编程逻辑阵列（PLA）。PLA于1975年首次投入使用。PLA是简单可编程逻辑器件中用户可配置性最好的器件，因为它的AND和OR阵列都是可配置的。我们首先考虑一个未编程的简单的3输入、3输出PLA（图3-6）。

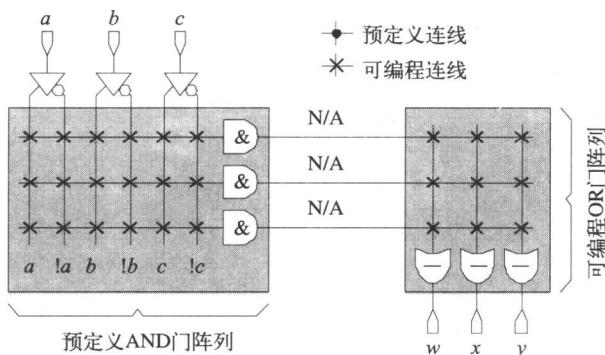


图3-6 未编程的PLA（可编程的AND和OR阵列）

与PROM不同，在AND阵列中的AND函数的数目是独立于器件的输入数目的。只要引入更多的行就可以在阵列中形成额外的AND函数。

类似的，OR阵列中OR函数的数目也是独立于AND阵列的AND函数和器件的输入数。引入更多的列就可以在阵列中形成额外的OR函数。

现在假设我们希望例子中的PLA要实现前面所述的3个公式。可以按照图3-7所示对适当的连线进行编程，就可以达到这个目的。

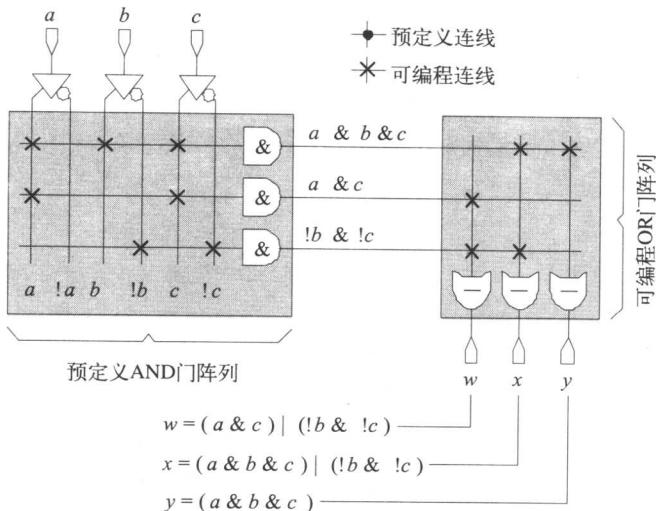


图3-7 编程后的PLA

$$w = (a \& c) | (!b \& !c)$$

$$x = (a \& b \& c) | (!b \& !c)$$

$$y = (a \& b \& c)$$

34

仿佛是它的宿命，PLA一直没有在市场上占据显要的地位，尽管不少提供商试着让这种器件去满足不同的需要。例如，PLA并不是必须要让AND阵列接入OR阵列，而别的与之类似的结构，比如AND阵列连接OR阵列偶尔也能大显身手。然而，在理论上OR-AND、NAND-OR和NAND-NOR等现场结构虽然可能，但这些变体相对很少或不存在。这些器件经常可转为AND-OR<sup>①</sup>（和AND-NOR）架构，原因之一是经常用于指定逻辑式的乘积项形式可以直接映射为后者。其他形式的逻辑式如和的乘积能够使用标准代数方法转化（这通常用软件实现）。

PLA据称对于大型设计非常有用，因为它的逻辑式有这样的特点：具有一组大量公共的乘积项，可用于多个输出；例如，在图3-7中，乘积项  $(!b \& !c)$  可用于  $w$  也可用于  $x$  的输出。这个特点可以实现对乘积项的共享。

不利的一面，与预定制的对应技术正相反，信号通过可编程连线要花费相对更

<sup>①</sup> 实际上在我写这段文字之前，一位设计者和我谈到他的团队发明了一种NOT-NOR-NOR-NOT结构（显然，这提供了一些速度优势），但是他们告诉用户这是一种（在外界看来更像是）AND-OR的结构，因为“这就是他们所期望的”。甚至到了现在，厂商所说的他们正在制造的器件和他们实际在做的器件都不一定完全相同。

35 长的时间。这样，AND和OR阵列都可编程就意味着PLA的速度要比PROM慢得多。

### 3.5.3 PAL和GAL

为了解决PLA的速度问题，一种叫做可编程阵列逻辑（PAL）的新器件于20世纪70年代末期出现。在概念上，PAL几乎与PROM正相反，因为它有一个可编程AND阵列和一个预定制的OR阵列。作为一个例子，考虑未编程状态的简单3输入、3输出PAL（图3-8）。

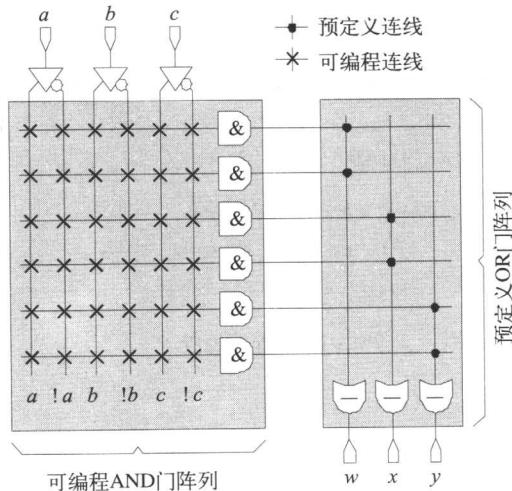


图3-8 未编程的PAL（可编程AND阵列，预定制OR阵列）

PAL的优势在于（与PLA相比）它的阵列只有一个可编程，所以速度要快得多。但是PAL有更多的限制，因为它只允许有限数量的乘积项相或（但是工程师们是聪明的，我们仍然有各种办法解决这种问题）。

### 3.5.4 其他可编程选择

以上展示的PLA和PAL例子为了简明起见，所以都比较小而且是初步的。除了36 将会大得多（有更多的输入、输出和内部信号）之外，真正的器件还能够提供一系列的可编程选择，如反相和三态输出等。

此外，有的器件支持寄存器或锁存器输出（与可编程多路复用有关，使用户可以指明在管脚到管脚的基础上输出是寄存还是非寄存）。而且有的器件提供配置特定管脚作为输出或额外输入的能力，类似的例子还能列出很多。

这里的问题是，不同的器件可能提供多种选择的不同子集，这使为了特定应用而选择最优的器件也构成一个挑战。对于这个问题，工程师们一般是把自己限定在有限的备选器件中，然后裁减自己的设计以适合这个器件，或者在应用基础

上，使用软件来帮助它们决定哪种器件最适合要求。

### 3.5.5 CPLD

在电子领域一个不言而喻的现象就是，每个人都在向往更强大（就实现功能的能力而言）、更小（就物理尺寸而言）、更快、更有力和更廉价——当然这是不必说的，不是吗？于是，在20世纪70年代末和20世纪80年代早期，更为复杂精密的PLD器件出现了，我们称之为复杂可编程逻辑器件（CPLD）。

领导这个潮流的是早期PAL器件的发明者MMI (Monolithic Memories Inc.) 公司的工程师们，他们称开发的组件为MegaPAL。这是一个具有84个管脚的器件，基本上是由4个标准的PAL及把它们连在一起的互连线构成的。然而，MegaPAL消耗了不相称的电能，普遍认为它与使用4个独立的器件相比，没有提供太多好处。

质变发生在1984年，新成立的Altera公司发明了基于CMOS和EPROM技术组合的CPLD。使用CMOS使Altera公司在消耗功率较低的情况下，获得极高的使用密度和复杂度。这些器件基于EPROM单元的可编程能力使它们在开发和原型验证环境下使用非常理想。37

Altera获得的名望不仅仅是由于把CMOS和EPROM结合起来。当工程师们开始把简单可编程逻辑器件SPLD发展为类似MegaPAL的更大型的器件时，总是设想成中央互连阵列（也就是可编程互连矩阵）连接独立的SPLD块，每个块要求百分之百的连接到输入和输出。问题是SPLD块尺寸上的加倍（要求双倍的输入和双倍的输出），结果就是互连阵列的尺寸增加到四倍。于是，造成了连接速度的降低、更高的功耗和部件成本的增加。

Altera推动了观念的更新，开始使用低于百分之百连接的中央互连阵列（看看与图3-10有关的讨论，有这一观念的更多信息）。这增加了软件设计工具的复杂性，但是它保证了速度、功耗和器件的成本。

尽管每个CPLD制造商都有他们自己的独特结构，一个普通的器件会包括一定数量的SPLD块（典型的如PAL）分享一个公共的可编程互连矩阵（图3-9）。

除编程单独的SPLD块之外，块之间的连接通过可编程互连矩阵来编程。

当然，图3-9是一个高层表示。在实际中，所有这些结构都在同一硅片上，这里没有展示其他多种多样的特点。例如，可编程互连矩阵可以包括大量的连线（比如说100），但是这超出了单独SPLD块能够应付的范围，它可能只适应一定量的信号（比如说30）。这样，SPLD块与互连矩阵的界面要使用某种形式的可编程多路复用器（图3-10）。38

CPLD的可编程开关可能基于EPROM、E<sup>2</sup>PROM、闪存或SRAM单元，这取决于制造商和器件的系列。在SRAM为基础的情况下，一些衍生器件允许SRAM单元与每个SPLD块联系，可以作为可编程开关使用或作为实际的存储块来使用，增强

39

了其功能。

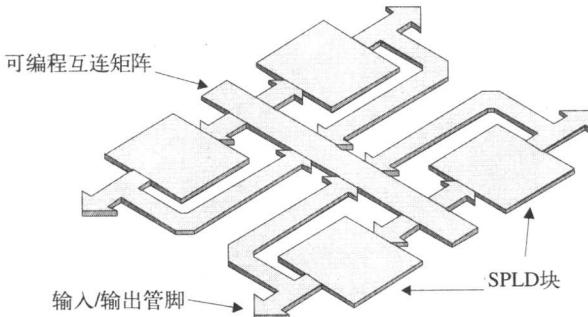


图3-9 一个典型的CPLD结构

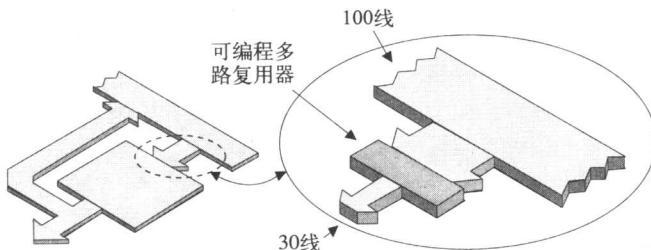


图3-10 使用可编程多路复用器

### 3.5.6 ABEL、CUPL、PALASM、JEDEC等

在许多方面，PLD的早期就像是设计工程师的黑暗时代。新器件刚开始的规格说明是原理图（或状态机）的形式。这些图是用铅笔在纸上画出来的，因为今天我们所熟知的计算机辅助电子设计记录工具，那时候还根本不存在。

当设计以图的形式记录下来以后，它又被手工转换成等价的图表，随后转成文本文件。这个文本文件会定义哪个熔丝将被熔断或是哪个反熔丝将要生成之类的事情。在那时，这个文本文件被直接输入到被称为“器件编程器”的特殊盒子，然后对芯片的编程。然而随着时代的进步，在一个主机上创建这个程序变得平常起来，于是要求在主机上把文件下载到编程器中，并且控制编程器（图3-11）。

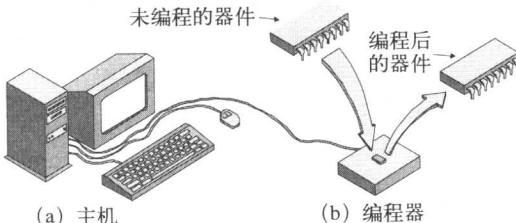


图3-11 编程PLD的物理过程

创建这个编程文件要求工程师对器件的内部互连和编程器使用的文件格式有深入的了解。可是，每种PLD的发明者都发展了属于自己的文件格式，也仅能用于他们自己的器件。很明显，这对定位和改正错误是不利的，而很显然每个人都关心设计是否会花费时间，或者是否会出错。

在1980年，EIA的成员JEDEC（联合电子器件工程师协会）发布了一种PLD编程文件的标准格式。过了不久，所有的编程器都改为能够接受这种格式。

大约在同时，John Birkner（他设想出最初的PAL器件并管理了它的开发）开发了PAL Assembler (PALASM)。PALASM既指一种基本的硬件描述语言 (HDL)，也可以看作应用软件。作为HDL时，PALASM允许设计工程师用源文本文件的形式来详细指明电路的功能，这种文本文件包含有乘积和形式的逻辑式。作为应用软件时（我们现在称这种软件为EDA工具），PALASM（只用了6页的FORTRAN<sup>①</sup>代码编写）能够读出源文本文件并且自动产生一个可用于编程器的编程文本文件。

在那个时代，PALASM是一个巨大的进步，但是最初的版本仅仅支持MMI的PAL器件，而且它没有执行任何化简或是优化。为了解决这些问题，1983年DATA I/O公司发布了它的先进布尔表达式语言 (ABEL)。大约在同时，Assisted技术公司发布了可编程逻辑通用工具 (CUPL)。ABEL 和 CUPL 既是HDL也是应用软件，并且支持状态机结构和自动逻辑化简算法。它们都可以用于不同厂商的多种类型PLD。

尽管PALASM、ABEL 和 CUPL 是早期HDL中最知名的，实际上还有许多其他的HDL，比如来自Signetics公司的AMAZE (Automated Map and Zap of Equations)。这些简单的语言和相应的工具为现在ASIC和FPGA设计使用的高层次硬件描述语言（如Verilog和VHDL）和工具（比如逻辑综合）铺平了道路。

## 3.6 专用集成电路（门阵列等）

在写作本书时，有4类专用集成电路（ASIC）不可不提。以复杂程度排序，它们是门阵列、结构化ASIC、标准单元器件和全定制芯片（图3-12）。

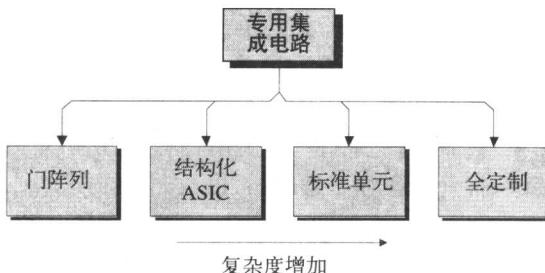


图3-12 不同类型的ASIC

<sup>①</sup> FORTRAN是IBM在20世纪50年代中期开发的，意为“FORmula TRANslation”，即公式转换语言，是第一种比汇编更高级的计算机编程语言。

尽管可以按照这个图所反映的复杂度增加顺序介绍这些ASIC，实际上按照它们的出现顺序来描述这些器件可以让我们加深对它们的认识，即全定制芯片，接着是门阵列，然后是编准单元器件，最后是结构化ASIC（注意结构化ASIC比传统的门阵列更加复杂还是更简单还有争论）。

### 3.6.1 全定制

在数字集成电路的早期，实际上只有两类器件（除了存储芯片之外）。第一个是相对简单的组件型器件，由TI和Fairchild等公司设计制造并作为标准元件销售给那些使用它们的客户。第二种是类似微处理器那样的全定制ASIC，由专门公司设计制造。  
[42]

在全定制器件的情况下，制造硅芯片的每一个掩模层都由设计工程师完全支配。ASIC的销售商没有在硅中预制任何组件，也没有提供任何预定义逻辑门和函数的库。

通过使用合适的工具，工程师可以深入到单个晶体管的尺寸，在这些元件的基础上创造出高层次的功能。例如，如果工程师需要快一点的逻辑门，他们可以改动晶体管的尺寸来构建这个逻辑门。用于全定制器件的设计工具通常是工程师们自己开发出来的。

1642年 Blaise Pascal 发明了一种叫做算法机的机械式计算器。

全定制器件的设计是高度复杂和很费时的，但是获得的芯片能够以最小的硅芯片面积实现最大量的逻辑。

### 3.6.2 Micromatrix和Micromosaic

在20世纪60年代中期的某个时候，Fairchild半导体开发了一种叫做Micromatrix的器件，它包括很有限（大约100）的无互连晶体管。为了使这种器件执行有用的功能，设计工程师手工喷镀金属层来连接在两个塑料薄片上的晶体管。

第一张薄片（用绿钢笔画的）表示Y轴（南—北）的轨迹在金属层1种实现，而第二张薄片（用红钢笔画）表示X轴（东—西）的轨迹在金属层2种实现。（额外的薄片用来画连接1层和晶体管的通孔（穿透的圆柱）以及连接1层和2层的通孔）。

这种方式记录一个设计是很痛苦很费时的，而且容易出错，但是至少困难的、代价高昂的、很费时的工作——创造晶体管——已经完成了。这意味着Micromatrix允许设计者能够以合理（尽管仍然很高昂）的花费和合理（尽管仍然很长）的时间开发出全定制器件。  
[43]

几年以后，在1967年，Fairchild开发了名为Micromosaic的器件，包含数以百计的无互连的晶体管。这些晶体管随后可以相连来实现大约150个AND、OR和

NOT门。Micromosaic的关键特点是设计者可以使用包含布尔（逻辑）方程的文本文件和一个决定必要的晶体管互连和光掩模结构的计算机程序，来完成器件以获得所需要的功能。在那个时候，这是革命性的，于是Micromosaic现在被视为现代ASIC形式门阵列的先驱，也是最早应用的计算机辅助设计（CAD）。

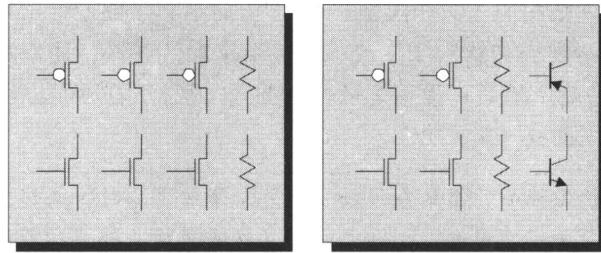
### 3.6.3 门阵列

门阵列的概念最早在20世纪60年代末期成型于IBM和富士通这样的公司里。然而，这些早期的器件只能在内部使用，直到20世纪70年代中期以CMOS为基础的门阵列技术实用化以后，普通人才能够买到它。

早期的门阵列层有时被称为无约束逻辑阵列 (ULA)，但是很快这个词就不再用了。

门阵列是基于基本单元的思想，基本单元是由可选择的未连接晶体管和电阻组成的。每个ASIC销售商来考虑在它的特定基本单元里提供什么样的组件才是最优的（图3-13）。

ASIC厂商开始把包含这些基本单元的阵列预定制在硅芯片里。在沟道门阵列中，基本单元呈现为单排或双排阵列，阵列之间的空间就是沟道（图3-14）。



(a) 纯CMOS基本单元 (b) 双极CMOS基本单元

图3-13 简单门阵列基本单元的例子

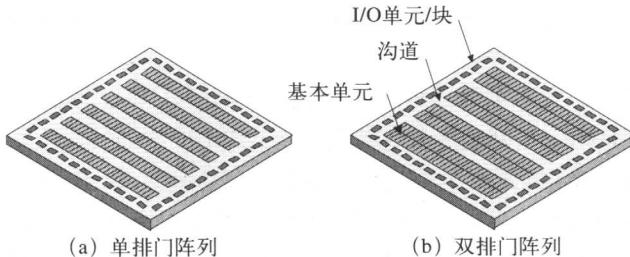


图3-14 沟道门阵列结构

**44** 相比之下，在少沟道或空闲沟道器件中，基本单元呈现为单个大阵列。器件的表面被基本单元的“海”所覆盖，而且没有明显的空间用于内部互连。于是，这些器件俗称为“门海”（sea-of-gates）或“单元海”（sea-of-cells）。

ASIC厂商定义了一组逻辑功能，比如原语门、多路复用器和寄存器供设计者使用。这些模块功能中的每一个都作为一个单元（不要与基本单元混淆），这些由ASIC厂商支持的一组功能就是单元库。

**45** 这些ASIC的实际设计方法超出了本书的讨论范围。简单来说，设计师最后得到一个门级网表，描述了他们所使用的逻辑门及其连接关系。使用专门的映射、布局和布线软件工具来把逻辑门指定到特定的基本单元，并且定义这些单元是如何连接在一起的。得到的结果用于产生喷镀金属层所需的光掩模，喷镀的金属层作用是连接基本单元内的元件、基本单元的彼此相连以及与器件的输入输出的连接。

门阵列提供了显著的成本优势，它内部的晶体管和其他元件是预定制的，只有金属层需要定制。它的不足在于大部分设计都有大量内部资源没有利用，内部的布置受约束，布线远远达不到最优。这些因素对设计的运行和功耗产生了负面影响。

### 3.6.4 标准单元器件

为了解决门阵列的这些问题，到了20世纪80年代的早期标准单元器件出现了。这种器件与门阵列很相似。又一次地，ASIC厂商定义了供设计工程师们使用的单元库。厂商也支持硬宏和软宏库，这些库包括诸如处理器、通信功能以及可选择的RAM和ROM功能。最后（但不是最不重要），设计工程师可以决定复用以前的设计或者购买知识产权（IP）块。

当一个电子工程师的团队在设计一个复杂集成电路时，与完全重新设计相比，他们可能会决定使用别人已经开发好的功能模块。这些功能模块就是知识产权，或简称IP。IP模块范围很广，有的具有复杂通信功能和微处理器，更复杂的功能像微处理器，被称为“核”。

最终，设计工程师又发明了门级网表。门级网表描述了要使用的逻辑门和逻辑门之间的连接。

**46** 与门阵列不同，标准单元器件没有使用基本单元的概念，在芯片中没有预定制元件。专门工具用来在网表中独立放置每个逻辑门，决定布线（门之间的连线）的最佳方式。结果就是在器件的制造中要为每一层创建定制的光掩模。

根据标准单元思想，可以使用最少量的晶体管创建每个逻辑功能单元，没有基本元件，而且每个功能单元的位置可以调整也有利于单元之间的布线。因此，标准单元器件对硅片的利用比门阵列更接近于最优化。

### 3.6.5 结构化ASIC

自从引入了标准单元器件以后，产业观察家们曾经预言门阵列的淡出，但是它们仍然占有一席之地，而且近些年还看到某种程度的回热。

结构化ASIC（尽管在那时还不这么称呼）在20世纪90年代开始的时候突然出现，没精打采地过了一段时间后，又回到了它原来的地方。10年之后（在2001年~2002年）ASIC制造商大多开始寻求创新以减少ASIC设计成本和开发时间。为了与传统的门阵列相区别，当2003年中期的某一时刻有人想到结构化ASIC这个非正式名称时，大家都很乐意地接受了。

通常（当然）每个厂商有他自己的合适架构，所以我们只讨论这些器件的公共属性。每个器件都会有基本单元，有的人称之为“模块”，还有的人称为“砖”。这些元件可能包含一个预定制的通用逻辑的混合（既可以实现逻辑门、多路复用器，也可以实现查找表），一个或几个寄存器，还可能有一些局部RAM（图3-15）。

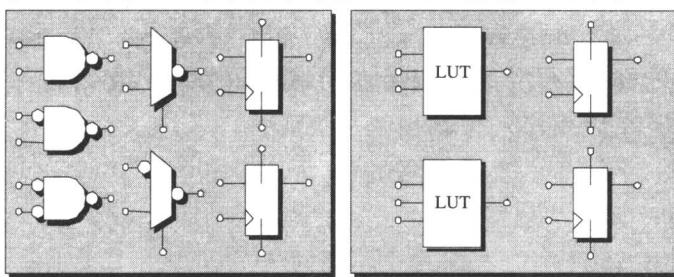


图3-15 结构化ASIC“砖”的例子

这样的一个单元阵列（海）就预制在芯片的表面上。或者，其他一些结构的基础单元（或基础模块或基础“砖”）只包括预制逻辑门、多路复用器或查找表形式的通用逻辑块。一个基础单元（如 $4 \times 4$ 、 $8 \times 8$ 或 $16 \times 16$ ）的阵列——连接着包含一些专门寄存器的单元、小存储单元和其他逻辑——构成一个主单元（或主模块或主“砖”等）。这些主单元的阵列（海）也是预定制在芯片的表面上。

一些功能块如RAM块、时钟产生器、边界扫描逻辑等也预定制在上面（一般在器件的边缘）（图3-16）。

它的主要思想是器件可以只用金属层来定制（就像一个标准门阵列）。不同之处在于，由于结构化ASIC的单元复杂许多，大部分金属层也预定制了。

这样，许多结构化ASIC只要求定制2或3个金属层（有的情况下，只需要定制单一通孔层）。这样就戏剧性地减少了为完成器件剩余部分所需要的光掩模而耗费的成本和时间。

47

48

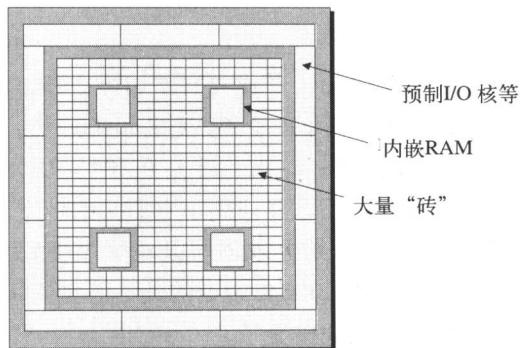


图3-16 通用结构化ASIC

尽管很难指出它的确切价值，预定义和预定制逻辑使结构化ASIC在功耗、使用效果和硅面积使用效率上都领先于标准单元器件。早期的结果显示结构化ASIC在执行相同的功能时需要的面积和功率分别是标准单元器件的三倍和两倍。实际上，这个结果会因结构而异，而且不同类型的设计也会产生不同的结构。很遗憾，在写作本书的时候，还没有能够涵盖所有出现的结构化ASIC结构的、以产业标准的参考设计为基础的估计。

### 3.7 FPGA

大约在20世纪80年代早期，数字IC环节中存在的缺陷已经很明显了。一端是可编程逻辑器件如PLD和CPLD，具有高可配置性和较短的设计和调试时间，但是不能支持大的、复杂的功能。

另一端是ASIC，这些器件可以支持极为大规模和复杂的功能，但是它们的设计很耗时、代价巨大。因而，一旦设计已经实现为一个ASIC，它就已经固化在硅片上了（图3-17）。

49

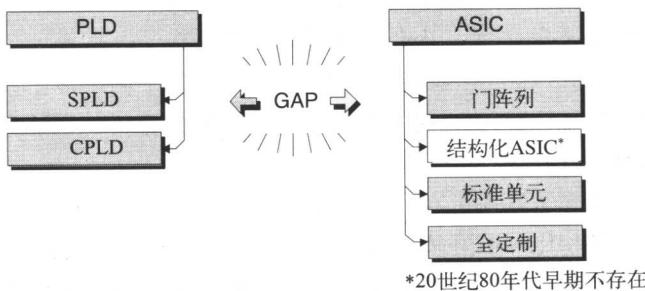


图3-17 PLD和ASIC之间的缺环

为了弥补这一缺环，Xilinx开发了一种新级别的IC，名叫现场可编程门阵列，

即FPGA，于1984年投入市场。

今天能够获的多种类型FGPA在第4章讨论细节。眼下我们只需要注意到第一个FPGA是基于CMOS，并且为了可配置使用了SRAM单元。尽管这些早期器件用今天的标准来看相对简单，包含的逻辑门数（或等效逻辑门数）也很少，但是底层架构的许多方面仍然沿用到了现在。

早期器件建立在可编程逻辑块的思想上，这种块包含一个3输入查找表（LUT）、一个可以用作触发器或锁存器的寄存器和一个多路复用器，还有这里不感兴趣的别的一些元件也和在它们一起。图3-18展示了一个很简单的可编程逻辑块（在新型FPGA中的可编程逻辑块要复杂的多——更多的细节见第4章）。

每个FPGA拥有大量下面所讨论的这种可编程逻辑块。通过适当的SRAM可编程单元，器件中的每个逻辑块能够配置以执行不同的功能。每个寄存器能够配置初值为逻辑0或逻辑1并作为触发器（如图3-18所示）或锁存器。如果选择为触发器，寄存器可以配置为是时钟正沿或负沿触发（时钟是所有逻辑块共用的）。多路复用器可以配置成接受LUT的输出，也可以配置成接受分离的逻辑块输出，然后复用器将信号送入寄存器，而LUT可以配置为表示任何3输入逻辑函数。

50

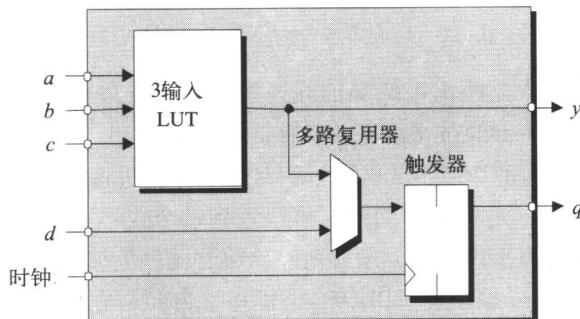


图3-18 构成简单可编程逻辑块的关键部件

例如，假设查找表LUT需要执行函数功能

$$y = (a \& b) \mid \! c$$

可以通过使查找表LUT输出适当的值来完成（图3-19）。

51

注意在图3-19中使用以8:1—多路复用器为基础的LUT是经过简化的；更逼真的实现内容在第4章里。

完整的FPGA包括大量的可编程逻辑块“岛”，周围是可编程互连线“海”（图3-20）。

通常，这些高层次描述仅仅是一种抽象表示。在现实中，所有这些晶体管和互连线都使用标准IC制造技术在同一片硅上实现。

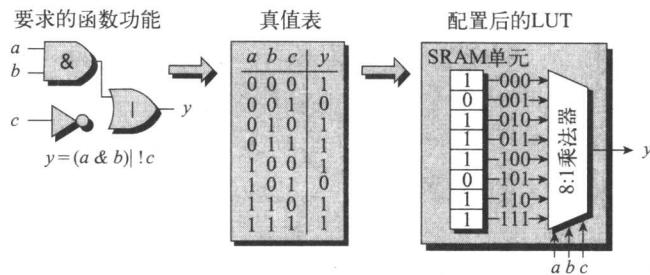


图3-19 配置查找表 LUT

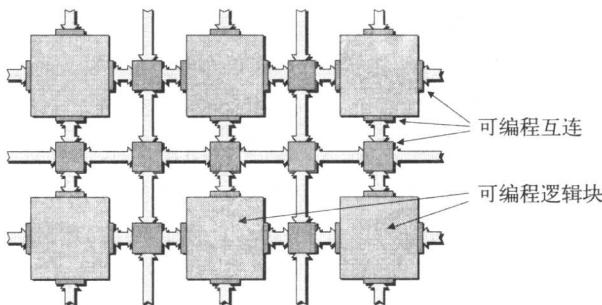


图3-20 从上向下看的简单、通用FPGA结构

除了在图3-20中反映出来的局部互连之外，还有全局（高速）互连线可以在芯片中传输信号而不必通过多路开关元件。

器件也具有基本的I/O管脚（这里没有显示）。通过使用自己的SRAM单元，内部互连线可以编程，这样器件的输入可以连到一个或多个可编程逻辑块，任何一个可编程逻辑块的输出也可以驱动其他逻辑块和器件的输出。

这样，FPGA成功地整合了PLD和ASIC。一方面，它具有类似PLD的高度可配置性，设计和调试很快。另一方面，它可以用来实现巨大和复杂的功能，这是以前只有ASIC才能完成的。（实际上，巨大、复杂、高性能设计仍然要求使用ASIC，但是由于FPGA复杂度的增加，它们开始越来越接近ASIC的设计空间。）

### 3.7.1 FPGA平台

参照设计或平台设计的思想在电路板极设计中已经存在了很久了。这是指创建一个基础设计轮廓可以用来派生出不同的产品。

除了大量可编程逻辑，今天高端FPGA的特征是具有内嵌（块）RAM、内嵌处理器核、高速I/O模块等。而且，设计者能够选取大范围的IP。这样就产生了FPGA平台的思想。一个公司可以使用一个FPGA设计平台作为公司内多种产品的基础，或者可以提供一个初始设计给其他的公司，让他们来做定制和差异化。

### 3.7.2 FPGA-ASIC 混合

在FPGA中嵌入ASIC将不会有任何意义，因为这样创造出来的器件将会面临ASIC设计流程产生的所有典型问题（不可回收成本高、开发时间长）。然而，已经有很多把一个或多个FPGA核作为标准单元ASIC设计的一部分的案例了。

在ASIC中嵌入FPGA的一个原因是，有利于使用设计平台的思想。在这种情况下平台是ASIC，嵌入的FPGA可以形成一种定制和差异化子设计的机制。

另一个原因是，最近几年已经能够看到用FPGA来增强ASIC设计的比率在增加。在这种情况下，大规模的、复杂的ASIC会在接近电路板的位置拥有一个附加的FPGA（图3-21）。

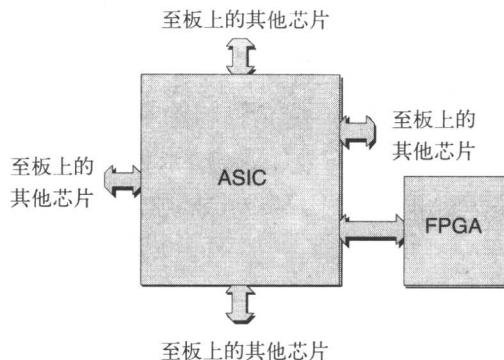


图3-21 使用FPGA完善ASIC设计

产生这种情况的原因是在ASIC中修改一个错误或由于初始设计条件的变化而调整功能时，要花费难以估量的时间和成本。然而，如果ASIC是正确的，它附加的FPGA就可以用来实现下游的调整和增强。这种办法的一个问题是ASIC和FPGA之间穿梭信号所消耗的时间。解决方案是把FPGA核嵌入到ASIC内部，结果就是一个FPGA-ASIC混合芯片。

然而，这种混合所面临的一个首要问题是，ASIC和FPGA设计工具和流程底层的显著不同。例如，ASIC被称为“细粒度”的，因为它们（最终）是在逻辑门原语层次实现的。这意味着传统的比如逻辑综合设计技术和布局布线实现技术也要转向“细粒”结构。

相比之下，FPGA被称为“中等粒度”（或“粗粒度”，这取决于谈话对象）的，因为它们的物理实现使用的是高层次模块，比如本章中前面所介绍的可编程逻辑块。这种情况下，在使用FPGA专用的综合和布局布线技术时实现的最好设计结果，是就这些高层次模块的角度而言的。

FPGA-ASIC混合的一个令人感兴趣方面是结构化ASIC，因为它更多地是以

块为基础考虑的。这意味着，当提到设计工具时，结构化ASIC的厂商与FPGA提供商谈综合和布局布线技术时比他们的传统ASIC伙伴更为投机。反过来，这意味着FPGA-ASIC混合基于将自动趋向于统一的工具和设计流程，因为同样以块为基础的综合和布局布线算法引擎能够通用于设计ASIC和FPGA部分。

### 3.7.3 FPGA厂商如何设计芯片

最后（但绝非不重要），通常会问一个问题，但是在关于FGPA的书中极少提到——FPGA厂商在实际当中是如何设计新一代器件的。

换句话说，他们是手工放置每个晶体管和轨道，使用的流程类似于全定制ASIC，还是他们创建一个RTL描述，把它综合成一个门级网表，并且使用布局布线软件，沿用着典型的ASIC（门阵列或标准单元）设计流程（这些工具背后的思想在第二部分讨论进一步的细节）。

简洁的回答是“都是”，稍长一些的回答是，器件的某些部分比如可编程逻辑块和基本连线结构，是FPGA厂商为每平方微米和每一纳秒时延都要仔细考虑的，这些部分的设计是使用全定制ASIC技术来手工放置晶体管和轨道的。  
55

另一方面，设计的这些部分都相当小且高度重复，所以一旦创建出来以后就沿着芯片表面复制几千次。

器件的辅助部分，比如配置控制电路，每个器件只有需要一个，而且尺寸和  
56 性能要求不那么苛刻，设计的这些部分是使用标准单元ASIC技术创建的。

# 第4章 FPGA结构的比较

## 4.1 一点提醒

这一章我们介绍FPGA多种多样的结构特点。某些结构（比如使用反熔丝还是SRAM配置单元）是无法共存的。有的FPGA厂商专注于其中的一种或几种，还有的厂商提供基于这些不同技术的多个器件系列。（除非另外注明，这里主要讨论基于SRAM的器件。）

对于不同的嵌入式模块，比如乘法器、加法器、存储器和微处理器核，不同的厂商有不同原料的“食谱”来满足不同“口味”。（就像不同商标的巧克力，巧克力块有大有小，某些FPGA器件系列也会有大型/中型/小型的嵌入式RAM块，或者其他的比例更多的乘法器、支持更多的I/O标准等。）

问题是每个厂商和每个器件系列所支持的特性几乎每天都在变化。这意味着一旦你决定所需要的特性，就需要做一些研究来看看哪个厂商当前提供的器件最接近于满足要求。

## 4.2 一些背景信息

在进入本章的主要部分之前，我们需要澄清一些概念，以便能够在统一的鼓点下前进。例如，你将看到“结构”（fabric）一词贯穿全书，在硅芯片的背景下，它是指器件的基础结构（有点像短语“文明社会的结构”）。 57

当你首次听到有的人这样的使用“结构”一词，可能觉得听上去有点傲气或自负（实际上，有的工程师认为这个词仅仅是一个由ASIC和FPGA厂商推广的市场概念，使他们的器件听上去比实际更加复杂精密）。然而，说真的，一旦你使用了之后，它真是很有用的词汇。

在我们谈到IC的“几何学”，我们是指建造在芯片上独立结构的面积——比如场效应管（FET）的沟道。这些结构令人难以置信地小。在20世纪80年代的早期到中期，器件是基于 $3\mu\text{m}$ 的几何结构，这意味着它们的最小结构在尺寸上是 $1\mu\text{m}$ 的百万分之三（反之，我们可以说，“这个IC基于 $3\mu\text{m}$ 技术”）。

每个新的几何结构被称为“技术节点”。在20世纪90年代，基于 $1\mu\text{m}$ 几何结构的器件开始出现，而且特征尺寸在这个十年的过程中持续垂直下落。到了21世纪，

高性能IC已经具有 $0.18\mu\text{m}$ 的几何结构了。到2002年，已经发展到 $0.13\mu\text{m}$ ，而2003年，基于 $0.09\mu\text{m}$ 的器件开始出现了。

所有小于 $0.5\mu\text{m}$ 的几何结构被划分为深亚微米（DSM）。按照某些还没有定义好的观点（或者那些多种定义，取决于谈话对象），我们在向超深亚微米（UDSM）领域迈进。

当几何结构降低到 $1\mu\text{m}$ 以下时，事情开始变得有点笨拙、不方便，因为我们不得不痛苦地说“ $0.13\mu\text{m}$ ”之类的话。由于这个原因，开始转换成使用“nm”，由于 $1\text{nm}$ 等于 $1\mu\text{m}$ 的千分之一，也就是百万分之一米的千分之一。这样，不必那么绕口的说“ $0.09\mu\text{m}$ ”，我们可以直接说“ $90\text{nm}$ ”。当然，这都是在说同一回事，但是如果你想在这些话题上和你的朋友们聊得很愉快，最好还是使用这些约定俗成的术语，跟上现在的潮流，而不是表现得唠唠叨叨的、好像是来自千禧年之前的人。

## 4.3 反熔丝与SRAM与其他

### 4.3.1 基于SRAM的器件

大部分FPGA是基于SRAM配置单元的，这意味着它们可以一次又一次地配置。这种技术的主要优势是，新的设计思想可以很快的实现并验证，这样，能够相对容易的容纳标准和协议的变化。或更进一步地，当系统第一次上电时，FPGA一开始可以被编程为一种自检测或是板级/系统检测，然后它可以再被编程为执行它的主要任务。

基于SRAM的另一个大好处是，这些器件能够站在技术的最前列。FPGA厂商可以利用许多别的致力于存储设备的公司在这一领域投入的庞大研发资源。而且，SRAM单元的制造使用与其他器件相同的CMOS技术，在创建这些元件时并不要求别的专门处理。

过去，存储器件经常用来证明与新技术节点相关的制造流程是否合格。最近，与最新一代FPGA有关的尺寸、复杂度和规律性的混合导致这些器件用于这个任务。使用FPGA代替存储器件的来评估制造工序流程的优点是，如果某处有缺陷的话，FPGA的结构使得确认和定位问题更容易（也就是说，估计错误是什么、在哪里）。  
[59] 例如，当IBM和UMC开发出他们的 $0.09\mu\text{m}$ （ $90\text{nm}$ ）制造流程时，来自Xilinx的FPGA就像是比赛中第一个冲出来的选手。

然而，世上没有免费的午餐。以SRAM为基础的器件的不利之处是它们在系统每次上电时都需要重新配置。这也要求使用一个专用的外部存储器件（这带来了与板级有关的花费和面积消耗）或者一个单板微处理器（或这些技术的一些变

动——见第5章)。

### 4.3.2 以SRAM为基础器件的安全问题和解决方案

关于以SRAM为基础器件的另一个应当考虑的问题是，你的智力产权很难得到保护。这是因为用来对器件编程的配置文件是存储在某种形式的外部存储器中。

目前，还没有商业工具来能够读取配置文件的内容并产生相应的原理图和网表描述。可以这么说，理解配置文件并提取出逻辑，虽然不是个琐碎的任务，但也并不超出聪明的人们的能力范围，在使用目前可获得的计算能力的情况下。

不要忘了世界上有大量的公司做逆向工程，专门从事复原“设计知识产权”。所以，如果设计是高收益的项目，你可以打赌一定有很多人渴望并准备在你不注意的时候复制它。

实际上，真正的问题不是某些人通过对配置文件内容做逆向来偷窃你的知识产权，而是他们可以克隆你的设计，不论他们是否理解内部运行的情况。很轻松地使用能够得到的技术，相对容易地做一个电路板，把它置于“天真儿童”般的测试状态，可以很快提取出完整的网表。这个网表可以接着用来复制电路。现在那些坏蛋剩下的任务就是复制你的FPGA配置文件到他的boot PROM(或EPROM、E<sup>2</sup>PROM，或其他什么)中，于是他们就拥有了完整设计的复制品。

好的一面是，今天某些基于SRAM的FPGA支持比特流加密的概念。这种情况下，最终的配置数据经过加密后存入外部存储器件。密钥本身经过FPGA的JTAG端口载入一个FPGA内部专用的基于SRAM的寄存器。与一些相关逻辑配合，密钥在加密配置比特流载入器件时对比特流解密。

载入加密比特流的命令/过程自动的禁用FPGA的回读能力。这意味着在开发期间你一般使用未加密配置数据(此时需要回读)，当你开始制造产品时才用加密数据(你可以在任何时候载入未加密的比特流，所以你能够容易地载入测试配置，然后重新载入加密版本)。

这个方案的主要缺点是，在电路板上你需要一组备用电池，当系统断电时它们保持FPGA中的密钥寄存器的内容。这个组将有一年或十年的生命期，因为它在器件里只需要保持一个简单的寄存器。但是它确实增加了电路板的尺寸、重量、复杂性和成本。

### 4.3.3 基于反熔丝的器件

不同于在系统编程的SRAM为基础的器件，基于反熔丝的器件使用专门的器件编程器是在机外编程的。

基于反熔丝器件的支持者为其多种多样(确实有一定价值)的优点自豪。首

先，这些器件是非易失性的（它们的配置数据在系统断电时仍能保持），这意味着它们在系统上电时可以立刻使用。由于它们的非易失性，这些器件不需要外部的存储芯片来存放配置数据，这样就节约了额外部件的成本，也节省了电路板的面积。

反熔丝器件的一个值得注意的优点是，它的内部互连结构是天生“防辐射的”，也就是说，它相对不受电磁辐射的影响。这对军事和宇航应用具有特别的吸引力，因为在这些环境下基于SRAM元件中的配置单元被射线击中时可能会发生“翻转”（外层空间中有大量的射线）。相比之下，当反熔丝器件编程以后，它不会以这种方式改变。还应当指出，这些器件中的任何触发器都对射线敏感，所以用于高辐射环境下的芯片必须使用三倍冗余设计来保护它们的触发器。这是指每个寄存器有三个副本来进行多数投票（在理想情况下，这三个寄存器将保持完全相同的值，但是如果有一个“翻转”导致出现两个寄存器说0而第三个说1，那么说0的占多数具有决定权，或者反过来，两个说1第三个说0同理）。

然而，或许基于反熔丝FPGA的最重要的优点是配置数据深深嵌入它的内部。在默认情况下，器件编程器读出数据是可能的，因为这实际上就是编程器在如何编程。由于每个反熔丝都被处理过，器件编程器持续测试它来确定元件何时已经完全编程，接着开始处理下一个反熔丝。此外，器件编程器可以用来自动验证配置过程是否成功执行了（当你用到包含五千万以上编程单元的器件时，这是很有用的）。为了做到这一点，器件编程器需要能够读出反熔丝的状态，并把它们与配置文件中定义的状态做比较。

然而，当器件已经被编程序，可能要设置（生长）一个专门的反熔丝，来防止随后从器件中读出任何配置数据（以反熔丝存在或者不存在的形式）。甚至当器件被破坏（它的顶部被除去），编程和未编程的反熔丝显示为完全一样，实际上所有的反熔丝都嵌入内部的金属层，这让逆向工程设计变得几乎不可能。

基于反熔丝FPGA的厂商可能也推出一些其他的优点，比如功耗和速度，但是，如果你没注意，它可能会像敏捷的手一样蒙蔽你的眼睛。举例来说，反熔丝器件只消耗等价的基于SRAM器件备用功率的百分之二十（大约），他们可能用这一事实来说服你，它们的运行功耗要低很多，而且内部互连延迟也小得多。同样，他们多半会偶尔提及反熔丝要小得多，这样就比同等的SRAM单元（尽管他们疏忽了反熔丝也需要额外的编程电路，包括每个反熔丝的巨大的编程晶体管）节约了大量的芯片面积。当你有了一个包含一千万配置单元的器件时，他们就会说使用反熔丝能够使静止逻辑更加密集。这会起到减少互连延迟的作用，从而使这些器件的速度超过了其SRAM对手。

以上这些观点会是对的——如果在同样技术节点实现的两种器件之间做比较

的话。但是事实上存在一个错误，因为在主要的制造过程已经合格以后，反熔丝工艺还需要使用三个额外的处理步骤。由于这个（相关的）原因，反熔丝器件总是落后于基于SRAM的元件至少一代——通常是几代（技术节点），而这显然会让人忽略其任何可能令人感兴趣的速度或是功耗优势。

当然，反熔丝工艺器件的主要不足在于它们是OTP，所以一旦你已经对其中的一个编程完毕，它的功能就会坚如磐石般不可更改。这将导致这些器件在开发或原型环境中几乎没有机会。

#### 4.3.4 基于EPROM的器件

这部分内容目前还没有人制造（或有计划制造）基于EPROM的FPGA。

#### 4.3.5 基于E<sup>2</sup>PROM/FLASH的器件

基于E<sup>2</sup>PROM或FLASH的FPGA与它们的SRAM对手很像，它们的配置单元是用一个长的移位寄存器型链联在一起的。这些器件可以使用一个器件编程器脱机配置。另外，有些版本是在系统或ISP编程的，但是它们的编程时间大约是基于SRAM器件的3倍。

编程之后，它们内部的数据容易丢失，所以在系统第一次加电时，这些器件将会“瞬时”运行。关于保护，这些器件中有的使用了多位密钥，范围可以从50比特到几百比特。当你对这种器件编程后，你可以载入你的用户定义密钥（比特模式）来确保配置数据的安全。载入密钥以后，从器件中读出数据或写入新数据的唯一途径，就是通过JTAG端口（这个端口在本章稍后和第5章中讨论）载入一个你的密钥的副本。事实上，目前器件的JTAG端口速度大约是20MHz，这意味着，将所有可能的值穷举一遍来破解密钥要用十亿年。

双晶体管E<sup>2</sup>PROM和FLASH单元的尺寸大约是它们的单晶体管EPROM兄弟的2.5倍，但是它们仍然比它们的SRAM对手小。这意味着它们的内部逻辑更紧密，这会减少互连延迟。

不利的方面是，这些器件在标准CMOS工艺的基础上要求大约5个额外的处理步骤，结果，它们落后于基于SRAM工艺的器件一代或几代（工艺技术节点）。最后，这些器件的静态功耗更高，以维持大量的内部负载电阻。

#### 4.3.6 FLASH-SRAM混合器件

总有些人想在烹饪锅里加上更多的原料。在FPGA中，有的厂商提供可编程技术的冷僻组合。例如，认真考虑一种器件，它的每个配置单元是由一个FLASH（或E<sup>2</sup>PROM）单元和一个相关的SRAM单元组合形成的。

这种情况下，FLASH元件就可以提前编程。这样一来，当系统上电时，FLASH单元的内容以大规模的并行方式复制到其相应的SRAM单元里。这种技术使你拥有了与反熔丝器件有关的非易失性，这意味着当系统上电以后器件可以立刻运行。但是与反熔丝器件不同，当器件保持在系统中时，可以继续使用SRAM单元来重配置它。或者，可以在系统中或用器件编程器用脱机方式使用FLASH单元重配置器件。

### 4.3.7 小结

表4-1简单总结了与上述不同的可编程技术工艺有关的关键点。

性 质	SRAM	反 熔 丝	E <sup>2</sup> PROM/FLASH
技术节点	最新	落后一代或者更多代	落后一代或者更多代
可重编程	有（系统内）	无	有（系统内或者离线）
重编程速度	快		是SRAM的三分之一
易失性（上电时必须编程）	有	无	无（要求的话可以）
要求外部配置文件	是	否	否
适合原型阶段设计	是（非常好）	否	有（合理）
瞬时性	无	有	有
IP安全性	可接受	很好	很好
配置单元的尺寸	大（6个晶体管）	非常小	有些小（2个晶体管）
功耗	适中	低	适中
固化	无	有	基本没有

## 4.4 细粒、中等微粒和粗粒结构

把FPGA分类为细粒或粗粒是很常见的。为了理解这样说的含义，我们首先回忆一下把FPGA与其他器件区分开的主要特点，它们的基本结构主要由大量的相对较小的可编程逻辑块“岛”嵌入在可编程互连“海”里构成（图4-1）。

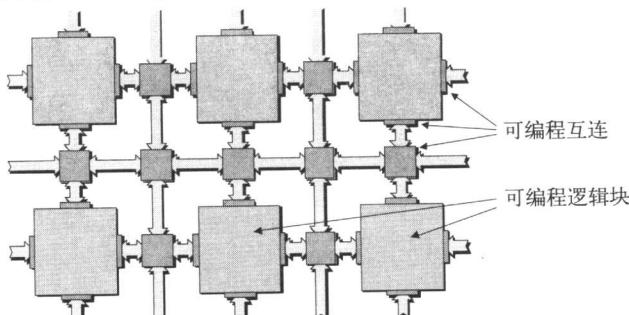


图4-1 FPGA基础结构

66

在细粒结构的情况下，每个逻辑块可以实现一个非常简单的逻辑。举例来说，可以把一个块配置为任何3输入函数，比如基本逻辑门（与、或、异或等）或存储元件（D型触发器、D型锁存器等）。

除了实现粘合逻辑和不规则结构像状态机，细粒结构在执行收缩算法（非常适合大规模并行实现的函数功能）时特别有效。这种结构还提供了一个优点适合于主要面向细粒ASIC结构的传统逻辑综合技术。

20世纪90年代兴起了对细粒FPGA结构的研究热潮，但是不久这个热潮退去了，剩下的只有粗粒结构。对于粗粒结构，它的每个逻辑块包含的逻辑数量比细粒的要多很多。例如，一个逻辑块可能会包含四个4输入LUT，四个多路复用器，四个D型触发器和一些快速进位逻辑（更多细节稍后会介绍）。

在谈到结构粒度时的一个重要考虑是，相比这些块能支持的功能数量来说，细粒实现需要相对更大数量的互连来从每个块输入和输出。在块的粒度增加到中等规模粒度和更高时，相比它们能支持的功能数量，与块相连的互连数量减少。这一点的重要性在于，可编程块互连引起的延迟占信号在FPGA内部传播延迟的大部分。

67

很多公司最近开始发展由结点阵列组成的真正的粗粒器件结构，它的每个结点都是一个高复杂度处理元件，包括了从某种算法比如快速傅里叶变换（FFT）到完整的通用微处理核（第6章和第23章也有相关内容）。尽管这些器件没有归类为FPGA，但它们确实涉足了这一领域。由于这个原因，基于查找表LUT的FPGA结构常常被分类为中等粒度，这样，粗粒度的称号就赋予这些新的基于节点的器件。

## 4.5 MUX 与基于LUT的逻辑块

在前面的部分中指出，用于组成中等粒度结构的可编程逻辑块有两种基本形态：基于MUX（多路复用器）和基于LUT（查找表）。

### 4.5.1 基于MUX的结构

作为基于MUX结构的一个例子，能够使用只包含多路复用器的模块来实现3输入函数 $y = (a \& b) | c$ （图4-2）。

68

器件可以这样编程，模块的每个输入呈现一个逻辑0、一个逻辑1，或者一个来自别的模块或器件基本输入的信号（这个例子中为 $a$ 、 $b$ 或 $c$ ）的真值或补值。这就允许每个模块可以用成千上万种方式配置来实现多种可能的功能（在图4-2中，多路复用器中心的 $x$ 说明不必在意这个输入是与0还是与1相连）。

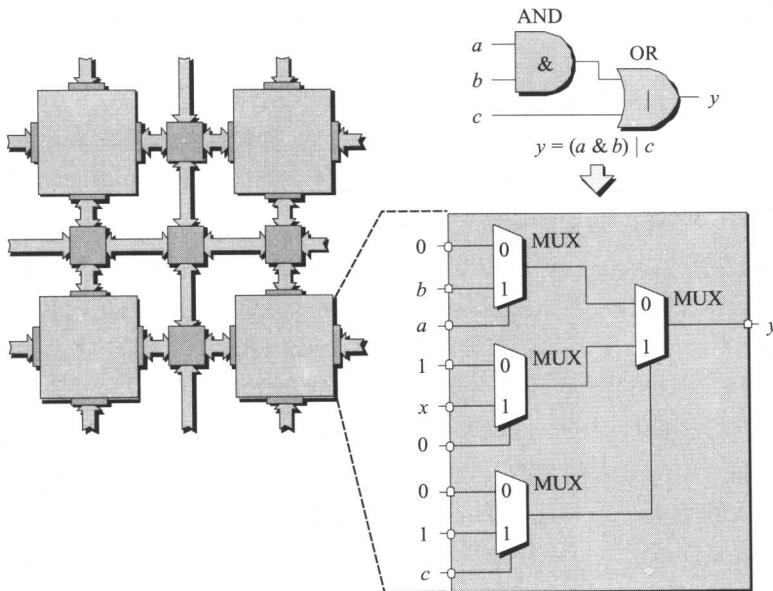


图4-2 基于MUX的逻辑块

#### 4.5.2 基于LUT的结构

在LUT背后的基本概念器很简单。一组输入信号用一个索引（指针）连接到一个查找表中。这个表的内容从单元结点到包括预期值的每个输入组合。例如，我们希望实现这个功能：

69

$$y = (a \& b) | c$$

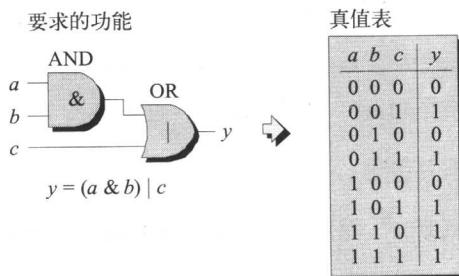


图4-3 要求的功能和相应的真值表

这可以通过将合适的值载入一个3输入查找表LUT来实现。我们假设LUT是由SRAM单元形成的（但是也可以用本章前面所讨论过的反熔丝、E<sup>2</sup>PROM或FLASH单元来形成LUT）。一种经常使用的技术是使用输入来选择期望的值，如图

4-4所示使用一个传输门的串联（注意，SRAM单元也连在一个用于配置的链上，即为了给它们载入要求的值。但是为了简洁，这个图中略去了这些连接。）

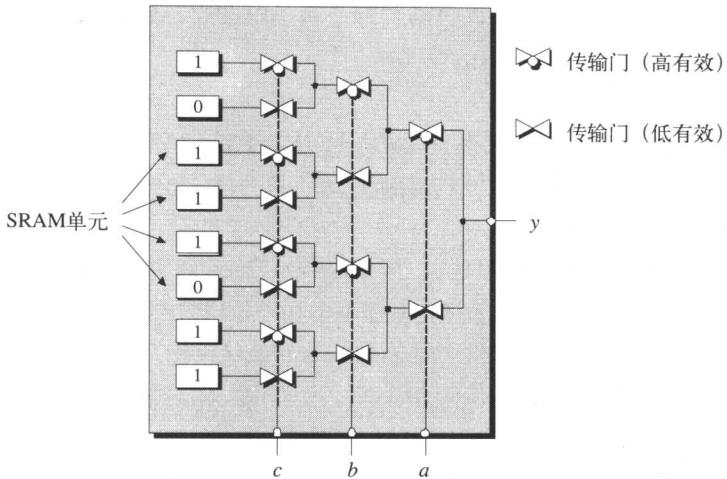


图4-4 基于传输门的LUT（这里为简洁略去了编程链）

如果一个传输门被使能（有效），它把信号从输入传递到输出。如果门被断开，它的输出和它所驱动的线也断开了连接。

传输门标识符的一个小圆圈（叫做bubble或bubble）表明这些门将在它们的控制信号为逻辑0时有效。如果没有这个bubble的标志，则表明这些传输门将在逻辑1时有效。基于这种理解，很容易看出如何使用不同的输入组合来选择多种SRAM单元的内容。

#### 4.5.3 基于MUX还是基于LUT

在今天的复杂CAD工具出现以前，工程师还在手工处理他们的电路，那时有人说使用基于MUX的结构能够达到最好的结果。（然而，他们通常不能精确解释这些结果如何更好，所以与我们的想象有很大的距离。）还有，在实现“如果一个输入是真并且另一个输入是假，那么输出为真……”<sup>①</sup>这样的控制逻辑时，基于MUX结构具有优势。然而，有的没有提供高速进位逻辑链，因此它们的LUT对手在与算术处理有关的领域处于领先地位。

在20世纪90年代的大部分日子里，FPGA广泛应用于远程通信和网络市场。这

- ① 某些基于MUX的结构——比如QuickLogic公司某些产品的架构——特点是，逻辑块包括由类似AND原语逻辑门组成的多层MUX。这样它们具有大量的扇入能力，在地址解码和状态机解码应用方面具有很大的优势。

两个领域都涉及大量的数据传输，基于LUT的结构具有更大的实用价值。此外，随着设计(和器件容量)越来越大和综合技术越来越复杂，手工处理大规模电路成为历史。其结果就是今天主流的FPGA结构都是基于LUT的，如下文所示。

#### 4.5.4 3、4、5或6输入LUT

**[71]** 对于一个 $n$ 输入查找表来说很重要的一点是，它能够实现任何可能的 $n$ 输入组合函数功能。增加更多的输入可以得到更复杂的功能，但是每当增加一个输入，所使用的SRAM单元就将加倍。

第1个FPGA是基于3输入LUT的。后来FPGA厂商和研究所又研究了3、4、5甚至6输入LUT的潜在价值(无论怎样，你都不要和一伙FPGA设计师在会议中争论)。主流意见一致认为4输入LUT提供了性能和成本的最优平衡。

过去，有的器件混合使用不同的LUT，比如3输入和4输入，这为优化器件的使用率提供了保证。然而，设计工程师的百宝箱中一个主要工具是逻辑综合，而一致性和规律性能让综合工具工作得更好。这样，当前真正成功的结构都是基于仅使用4输入LUT的(这并不是说混合LUT结构在未来也不会随着设计软件复杂度的持续增加而重新出现)。

#### 4.5.5 LUT与分布RAM与SR

一个拥有大量SRAM单元的基于SRAM器件中还包含LUT核，这一事实提供了大量令人关注的可能性。除了它最初作为查找表的角色，一些厂商允许单元组成LUT来作为一个小块的RAM(例如，16个单元形成一个4输入LUT，能够扮演一个 $16 \times 1$  RAM的角色)。因为LUT分布在芯片的表面，而且不同于大块的块RAM(本章的后面将会介绍)，所以称之为分布RAM。

**[72]** 还有，所有的FPGA配置单元，包括那些组成LUT的，实际上串在一起成为一个长链(图4-5)。

有关结构的细节将在第5章进一步讨论。这里要指出的是，一旦器件被编程，有的厂商允许SRAM单元形成一个LUT来对待链中的独立主体，并且以一个移位寄存器的形式来使用。这样，每个LUT可能被理解为多面的(图4-6)。

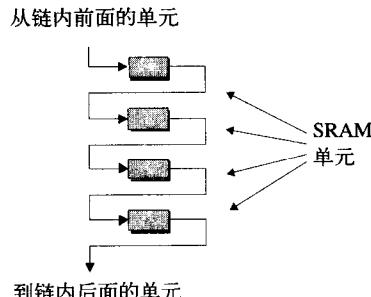


图4-5 配置单元连在一条长链里

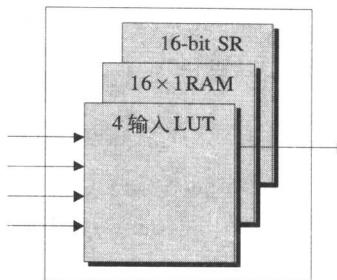


图4-6 多面的LUT

## 4.6 CLB、LAB与slices

“人们的生活中不能只有LUT”，如果Bard偶然被当作一个FPGA设计者的代表，他一定会这么说。基于这个原因，除了一个或几个LUT之外，一个可编程逻辑块将包括其他元件，比如多路复用器和寄存器。但是在涉及这个主题之前，首先需要给我们的大脑增加一些术语。73

### 4.6.1 Xilinx 逻辑单元

在谈到FPGA时会比较令人烦恼，即不同厂商对于相同的东西都独立命名。例如，Xilinx的现代FPGA中的核心块叫做逻辑单元(logic cell, LC)。除此之外，一个LC包括一个4输入LUT（也可以作为一个 $16 \times 1$ RAM或一个16比特移位寄存器）、一个多路复用器和一个寄存器（图4-7）。

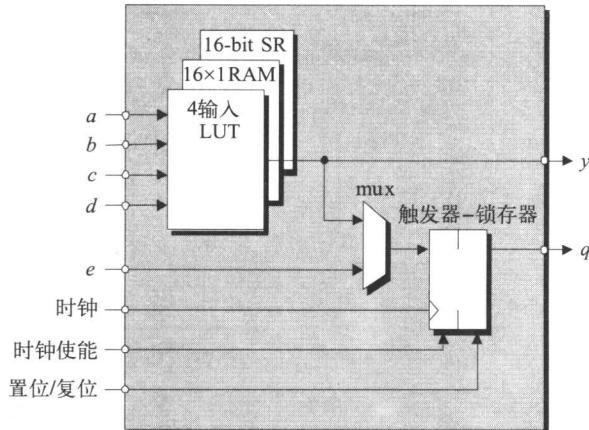


图4-7 一个Xilinx LC的简单视图

必须指出的是，图4-7的描述是经过简化的，但是它满足了我们的意愿。寄存

器可以配置为一个触发器或一个锁存器，如图4-7所示。时钟的极性（上升沿触发或下降沿触发）可以配置，时钟使能和置位/复位信号（高有效/低有效）的极性同样也可以。

74 除了LUT、MUX和寄存器之外，LC也包括其他元件，如一些用于算术操作的专用快速进位逻辑（稍后将讨论更多的细节）。

### 4.6.2 Altera逻辑部件

为了便于参考，在Altera的FPGA中核心块叫作逻辑部件（logic element，LE）。Xilinx的LC和Altera的LE有很大不同，但是总的来说是很类似的。

### 4.6.3 slicing和dicing

等级中的下一级就是Xilinx的slice（Altera和其他厂商也有相应的名字）。为什么是slice（切片）？好的，他们必须称它为某种东西，而且，不论你如何看待它，slice（切片）一词都是“某种东西”。在本书写作时，一个slice包括两个逻辑单元LC（图4-8）。

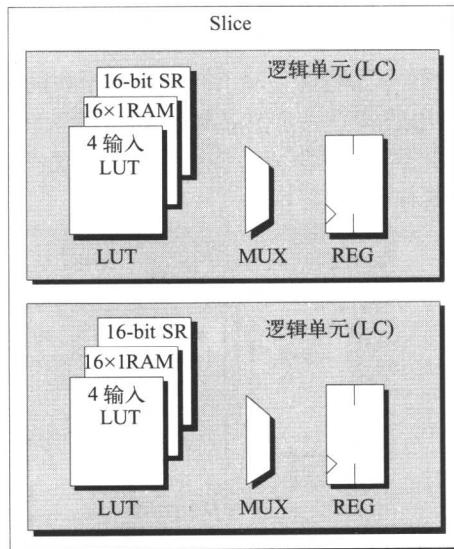


图4-8 一个slice包括两个逻辑单元LC

75 用“在本书写作时”修饰的原因是这些定义能够，而且确实随着时间改变。为了简洁在图中省略了内部的连线，然而，应当指出，尽管每个逻辑单元的LUT、MUX和寄存器有它们自己的数据输入和输出，slice有一套时钟、时钟使能和置位/复位信号共用于两个逻辑单元。

#### 4.6.4 CLB和LAB

随着我们在等级结构中又上了一级，我们到达了Xilinx称为可配置逻辑块（CLB）而Altera称为逻辑阵列块（LAB）的地方（毫无疑问，对这里的每个主题，其他FPGA厂商都有自己相应的名称，但是只有当你实际用到他们的器件时，这些才会令人感兴趣）。

使用CLB来作为例子，一些Xilinx FPGA在每个CLB中有2个slice，而另一些有4个。在写下这些的时候，一个CLB等同于我们原先描述的在可编程互连“海”中的可编程逻辑“岛”的一个逻辑块（图4-9）。

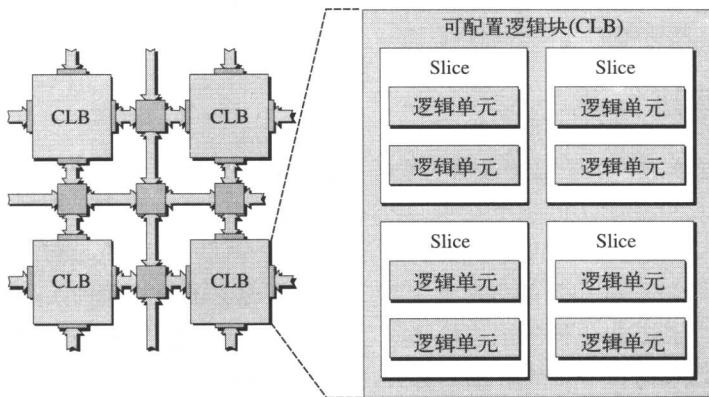


图4-9 一个CLB包括四个slice (slice的数目与FPGA的器件系列有关)

在CLB中也有一些快速可编程互连。这些互连（为了清晰。在图4-9中没有显示）用于连接相邻的slice。

76

造成这种类型的逻辑块等级——LC→Slice（包括2个LC）→CLB（包括4个slice）——的原因是，它由在互连中的同一等级对应部分相补充而组成的。这样，在同一slice的LC之间有快速互连，在同一CLB中的slice之间稍慢些，接下来是CLB之间的互连。这样做可以比较容易地把它们彼此连在一起，同时也不会增加太多的互连延迟，达到优化平衡。

#### 4.6.5 分布RAM和移位寄存器

我们前面提到每个4比特LUT可以当作一个 $16 \times 1$  RAM来使用。而这一切在变得越来越好因为，设想一下在图4-9中描绘的每个CLB包括4个slice的配置，一个CLB中的所有LUT都配置在一起就可以实现如下：

- 单口  $16 \times 8$  比特 RAM
- 单口  $32 \times 4$  比特 RAM

- 单口  $64 \times 2$  比特 RAM
- 单口  $128 \times 1$  比特 RAM
- 双口  $16 \times 4$  比特 RAM
- 双口  $32 \times 2$  比特 RAM
- 双口  $64 \times 1$  比特 RAM

同样地，每个4比特LUT可以用作一个16比特移位寄存器。这时，在slice中的逻辑单元之间有专用连接，而且在slice之间允许一个移位寄存器的最高比特连接到另一个的最低比特，而不使用通常的LUT输出（它可以用来观察在16比特寄存器内部被选中比特的内容）。这样，一个CLB内部的LUT配置在一起就可以实现一个容量多达128比特的移位寄存器。

## 4.7 快速进位链

现代FPGA的一个关键特性是，它们拥有实现快速进位链所要求的专门逻辑和互连。77 在前面介绍CLB的上下文中，每个LC包含专门进位逻辑。它还由每个slice的2个LC之间、每个CLB的slice之间和CLB之间的专门互连补充配套。

这些专门进位链和专用路由大大提升了计数器逻辑功能和加法器的算术功能的性能。这些快速进位链的工作效率，这与LUT形成的移位寄存器（前面讨论过）和嵌入式乘法器等（下面将介绍）有关，为FPGA在数字信号处理上的应用提供了保证。

## 4.8 内嵌RAM

大量的应用要求使用存储器，所以FPGA现在包括很多块内嵌RAM，这叫作78 内嵌RAM或块RAM。取决于元件的结构，这些块可能位于器件的外围、孤立分散在芯片的表面或是组织成列，如图4-10所示。

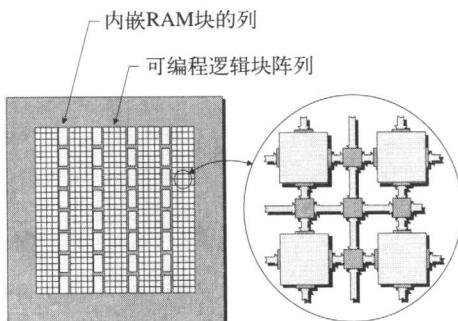


图4-10 带有内嵌RAM块的芯片鸟瞰

取决于器件，这样一个RAM可能能够保存几千到上万比特。此外，一个器件可能包括从几十到数百个这样的RAM块，这样可提供从几百K到几M的完整的存储能力。

每个RAM块可以单独使用，或多个块可以连在一起形成一个更大的块。这些块可以用于多种用途，比如实现标准的单口或双口RAM、先入先出（FIFO）功能、状态机，等等。

## 4.9 内嵌乘法器、加法器、MAC等

有的功能，像乘法器，如果用大量的可编程逻辑块连在一起来实现会速度很慢。由于大量的应用都要求这些功能，许多FPGA固化了专门的硬线乘法器块。它们的位置一般紧邻前面所提到的内嵌RAM块，因为这些功能往往被同时使用（图4-11）。

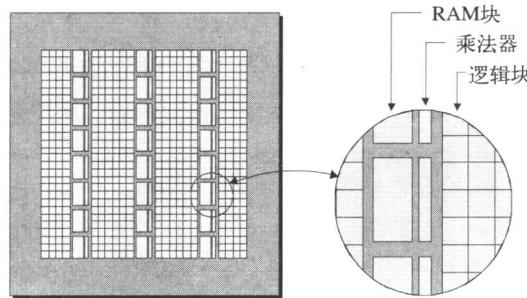


图4-11 包含有内嵌乘法器和RAM块阵列的芯片鸟瞰

类似地，有的FPGA提供了专用的加法块。一种在DSP型的应用里很常见的操作叫做乘累加（MAC）（图4-12）。顾名思义，这个功能把两个数相乘再把结果累

79

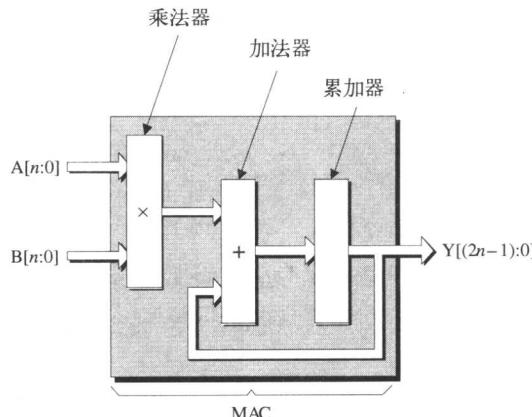


图4-12 形成MAC的功能

加到总和当中。

如果你所使用的FPGA仅支持内嵌乘法器，你将不得不把乘法器和大量可编程逻辑块形成的加法器连接起来实现这个功能，而结果将存入相关的触发器、一个块RAM或大量的分布RAM。如果FPGA也提供内嵌加法器，并且有的FPGA提供完整的MAC作为内嵌的功能，我们的工作会更加轻松。

## 4.10 内嵌处理器核（硬的和软的）

一个电子设计的任何部分几乎都可以实现为硬件（使用逻辑门和寄存器等）或软件（在微处理器中按照指令执行）。把它们区分开的标准是，你希望这些功能在执行它们的任务时有多快。

- 皮秒和纳秒逻辑：**运行时疯狂地快，必须用硬件（在FPGA结构里）实现。
- 微秒逻辑：**运行时相当快，并且既可以用硬件实现也可以用软件实现（这个类型的逻辑就需要你花费大块时间来决定采用哪种方式完成）。
- 毫秒逻辑：**这是用来实现接口功能的，如读取切换位置和闪烁发光二极管（LED）。硬件在实现这种功能时会很痛苦（比如，需要产生大量的计数器来产生时延）。这样，这些任务用在微处理器上运行的代码来实现会更好（与专用硬件相比，处理器能够给你糟糕的速度，但是极好的复杂度）。

事实上，设计的大部分都在使用这种或那种形式的微处理器。直到前不久，它们还呈现为电路板上的离散设备。最近以来，高端的FPGA已经开始提供一个或更多的内嵌微处理器，一般称为微处理器核。这时，似乎可以把所有由外部微处理器执行的任务都移到内部核来执行。这样做有很大的好处，不仅仅节约了两个设备的花费，还大量减少了电路板的走线、焊盘和管脚，而且使电路板更小更轻。

### 4.10.1 硬微处理器核

硬微处理器核被实现为专用的、预定义的块。这样的核集成到FPGA中有两种主要的途径。第1种是把它置于在FPGA主结构一侧的窄带（实际上叫作The Stripe）（图4-13）。

这种情况下，所有的元件被制造到同一硅片上，尽管它们也可以被制造成两个芯片并且封装为多芯片组件（MCM）。FPGA主结构也包含内嵌RAM块、乘法器和前面介绍过的类似元件。但是为了简洁我们把它们忽略了。

这种实现的一个优势在于，对于器件来说FPGA主结构是完全相同的，没有内嵌处理器核，使工程师使用设计工具时更加容易。另一个优点是FPGA厂商可以把所有附加功能装入窄带来辅助微处理器核，比如存储器、专用的外围设备等。

另一种方式就是把内嵌的一个或多个微处理器核直接放入FPGA主结构。目前，

主流的实现包含有1个、2个甚至4个核（图4-14）。

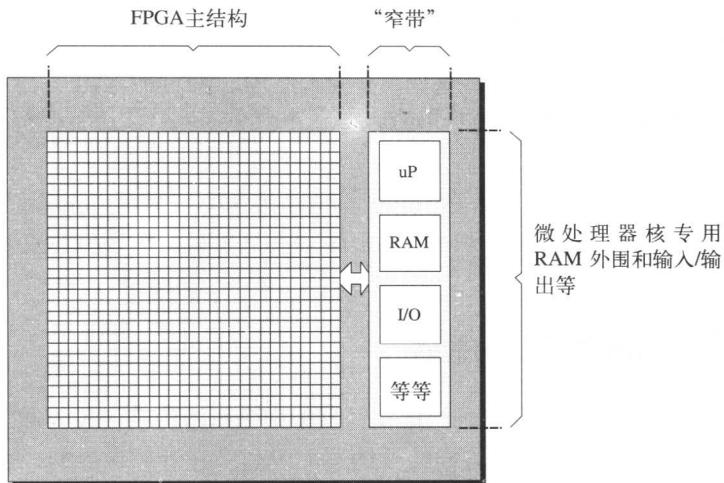


图4-13 在主结构之外具有内嵌核的芯片鸟瞰

同样，FPGA主结构也将包括内嵌RAM块、乘法器和前面介绍过的类似元件。82  
为了保持简单我们在这个插图中也忽略了它们。

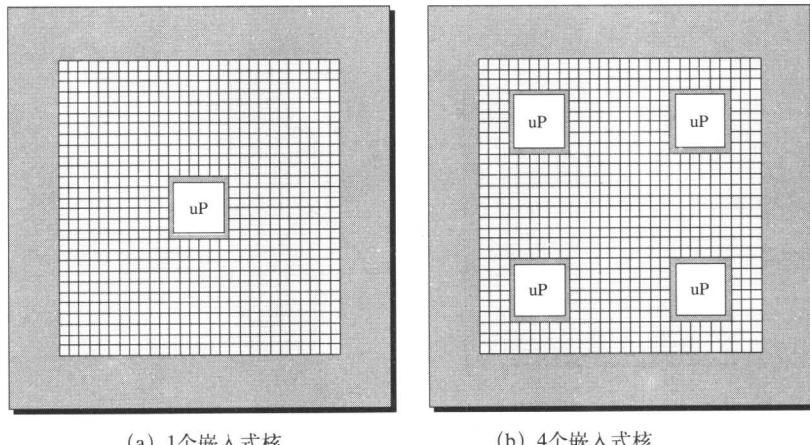


图4-14 嵌入式核在主结构内部的芯片鸟瞰

这种情况下，设计工具能够考虑到这些块在主结构内部的存在，这些核使用的存储器由内嵌RAM块实现，而其他附属功能使用通用可变成逻辑块实现。这种方案的支持者称由于微处理器核与FPGA主结构紧密相邻，能够获得内在的速度优势。

### 4.10.2 软微处理器核

与内嵌微处理器核物理上置于芯片结构内部相反，还可以把一组可编程逻辑块配置为一个微处理器，这通常被称为软核。还有更精细的分类，“软”或“固”取决于微处理器功能映射到逻辑块的方式（在本章后面的硬IP、软IP和固IP的主题中也可以看到有关讨论）。

83 软核更简单（更原始）且比它们的硬核伙伴要慢<sup>①</sup>。然而，它们具有自己的优势，当你需要它时你只需要实现一个核，而且你可以根据需要例化更多的核直到用尽可编程逻辑块资源。

## 4.11 时钟树和时间管理器

FPGA内部所有的同步部件——例如，可编程逻辑块内被配置为触发器的寄存器——需要时钟信号来驱动。这样的时钟源一般是来自外部世界的，通过专用时钟输入管脚进入FPGA，接着传送到整个器件并连接到适当的寄存器。

### 4.11.1 时钟树

84 考虑一个忽略了可编程逻辑块，只显示时钟树及其相连的寄存器简化的表示（图4-15）。

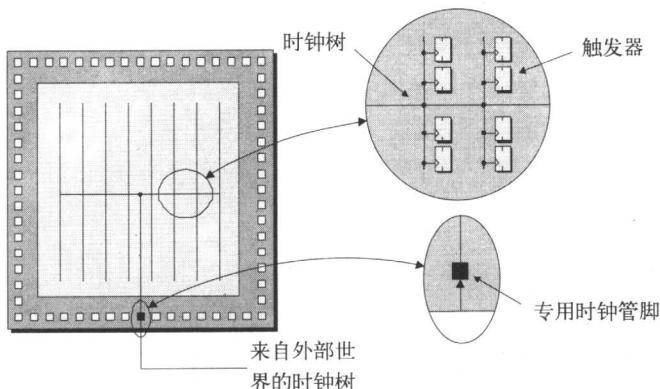


图4-15 简单的时钟树

之所以作为时钟树是因为主时钟信号一次又一次的分支（触发器可以视为在分支末端的“叶子”）。使用这种结构来保证所有触发器接收到的信号尽可能一致。如果时钟使用一条长的走线来一个接一个地驱动触发器，那么最接近时钟管脚的触发器接收的信号看上去将比位于链条最末端所接收到的快很多。这被称为抖动，

<sup>①</sup> 软核的运行速度一般是硬核的30%~50%。

并且会带来很多问题（甚至使用时钟树时，在分支上的寄存器之间和分支本身之间也存在一定数量的抖动）。

时钟树使用专门的走线，与通用可编程互连相分离。以上反映的情况其实是非常简单的。在实际情况中，可以有多个时钟管脚（不使用的时钟管脚可以作为通用I/O管脚），而在器件内部可以有多个时钟域（时钟树）。

## 4.11.2 时钟管理器

如果在配置时不把时钟管脚直接连入内部的时钟树，可以把管脚用于驱动一个叫作时钟管理的专用硬线功能（块），产生一定数量的子时钟（图4-16）。

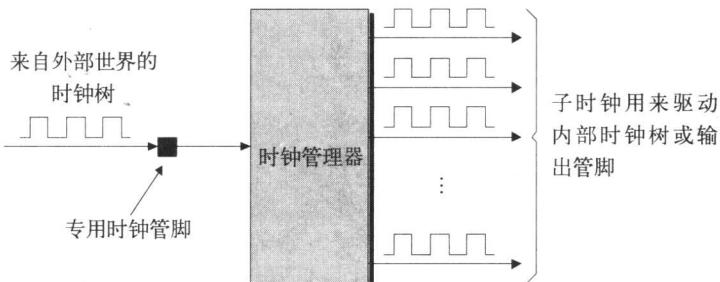


图4-16 一个时钟管理器产生子时钟

这些子时钟可以用来驱动内部时钟树，或者驱动输出管脚为主板上其他器件提供时钟。每个系列的FPGA有它们自己的时钟管理器类型（一个器件里可能有多个时钟管理模块），不同的时钟管理器可能会支持下列特性的一个子集。

**消除抖动：**作为一个简单例子，设想一个频率为1MHz的时钟信号（当然，在现实中会高得多）。在理想环境下来自外部世界的每个时钟沿将会精确地在前一个的百万分之一秒后到达。然而，在真实世界中，时钟沿可能会来得早一些或晚一些。

为了使这种影响，也就是抖动更直观，设想一下把多个时钟沿重叠在一起；结果将是一个“模糊”的时钟（图4-17）。

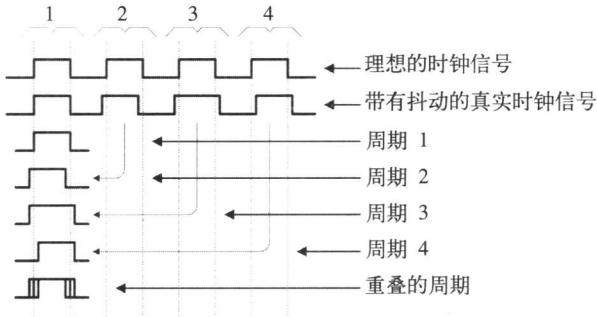


图4-17 抖动将导致一个模糊的时钟

FPGA的时钟管理器可以检测并纠正抖动，提供一个“干净”的子时钟信号给器件内部使用（图4-18）。

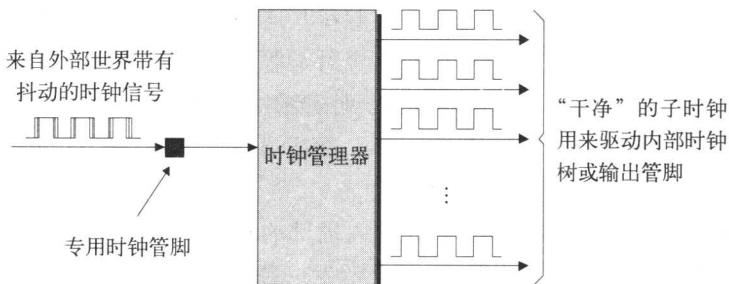


图4-18 时钟管理器能够消除抖动

**86** 频率综合：有的时候，来自外部世界的时钟信号的频率并不符合设计者的期望。这时，时钟管理器可以把源时钟信号倍频或分频来得到子时钟。

作为一个实际中的简单例子，考虑3个子时钟信号：第1个的频率等于源时钟，第2个的频率是源时钟的两倍，而第3个的频率是原时钟的一半（图4-19）。

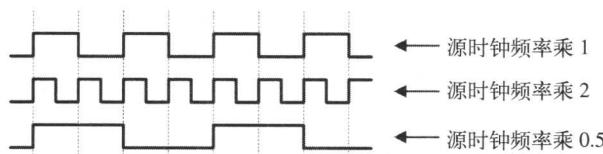


图4-19 使用时间管理器完成频率综合

图4-19反映的是非常简单的例子。在真实世界里，人们可以综合各类内部时钟，比如输出为源时钟频率的五分之四。

**相位调整：**某些设计要求使用彼此相位可以调整（延迟）的时钟。一些时钟管理器允许你在固定的相位调整值比如 $120^\circ$ 和 $240^\circ$ （用于三相时钟方案）或 $90^\circ$ ， $180^\circ$ 和 $270^\circ$ （如果要求的是四相时钟方案）中选择。还有的FPGA允许你对设计中要求的每个子时钟配置精确的相位调整值。

**87** 例如，我们设想一下从一个主时钟得到4个内部时钟，第1个相位与源时钟相同，第2个的相位改变了 $90^\circ$ ，第3个相位改变 $180^\circ$ ，依此类推（图4-20）。

**自动偏移校正：**为简便起见，设想我们正在谈论一个子时钟，它被配置为频率和相位都与输入FPGA的主时钟信号相同的。然而，时钟管理器会给信号带来一定的时延。而且，在分配时钟的互连线和驱动门上会产生更大的时延。结果就是，如果没有任何校正措施的话，子时钟将比输入时钟落后一些。这两个信号的差叫做偏移。

取决于子时钟和主时钟在FPGA（和电路板的其他部分）中的使用方式，偏移

会产生一系列的问题。于是，时钟管理器允许一个用于子时钟的专门输入。这时，时钟管理器将比较两个信号，并给子时钟加额外的延迟使它重新与主时钟对准（图4-21）。

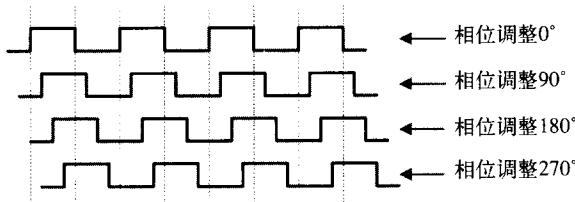


图4-20 使用时钟管理期调整子时钟的相位

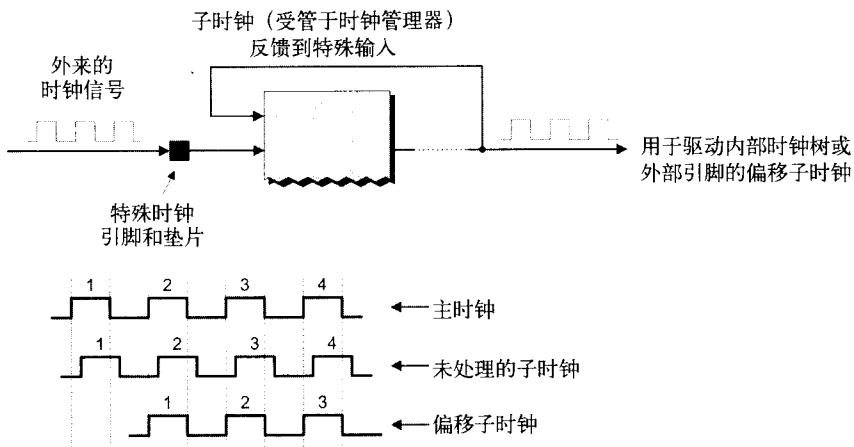


图4-21 以主时钟为参考偏移

只有基本的（零相位）子时钟使用这种方式处理，所有其他子时钟将在相位上与基本子时钟对准。

有的FPGA时钟管理器基于锁相环（PLL），还有的基于数字延迟锁相环（DLL）。PLL从1940年代起就开始用于模拟技术实现，但是目前对数字方式的强调使它用于匹配数字信号的相位。PLL既可以用模拟技术实现也可以用数字技术实现，而DLL通常被认为是数字技术。DLL的支持者说它们提供了精确度、稳定性、功率管理、对噪声不敏感和抖动表现方面的优势。

88

## 4.12 通用I/O

今天的FPGA能够拥有1千甚至更多的管脚，它们在封装底部排成一个阵列。类似地，当它们进入封装内部的硅芯片时，倒装芯片封装方法允许电源、地、时钟和I/O管脚位于芯片的表面。但是，如果纯粹为了这些讨论（和描述）的目的，

89

我们假设所有这些芯片的连接位于器件周边的环形上，事情将变得简单。事实上，过去很多年它们确实是在那里。

### 4.12.1 可配置I/O标准

让我们从结构和工程师设计电路板的视角考虑一会儿电子产品。取决于他们想做什么，他们所使用的器件，电路板工作的环境，等等。这些工程师将选择一个特殊的标准来传输一个数据信号（在这里，“标准”指的是信号的电特性，比如逻辑1和逻辑0的电平）。

问题是这类的标准有很多，制造一个特殊的FPGA来适应每一个标准是很痛苦的。由于这个原因，FPGA的通用I/O接收和产生的信号能够适合各种标准。这些通用I/O信号被分割为大量的簇（bank），我们设为从0~7的8个簇（图4-22）。

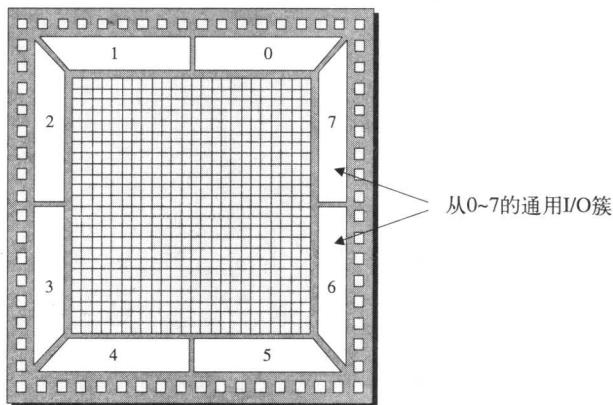


图4-22 显示通用I/O簇的芯片鸟瞰

令人感兴趣的是，每个簇可以被独立的配置为支持一种特殊的I/O标准。这样，除了允许FPGA与使用不同I/O标准的器件协同工作外，还能够使FPGA作为不同I/O标准之间的接口（也可作为不同协议之间的转换，这些协议可能基于专门的电气标准）。

90

### 4.12.2 可配置I/O阻抗

在今天的电路板上，用来连接器件的信号往往具有很高的边沿速率（这是指信号在不同逻辑值切换时所花费的时间）。为了阻止信号反射（边界反弹），很有必要为FPGA的输入和输出引脚设置适当的终端电阻。

过去，这些电阻是位于FPGA外部电路板上的离散的元件，然而，随着引脚数目的增加和间距的缩小，这种技术变得问题很多。因此，今天的FPGA能够使用内部的终端电阻，它的阻值可以由使用者根据不同的电路板环境和I/O标准来调整。

### 4.12.3 核与I/O电压

在过去的日子里，1965年至1995年，大部分数字集成电路使用0V的地线和1个5V的输入电压。而且，它们的I/O信号也在0V（逻辑0）和+5V（逻辑1）之间切换。

后来，硅芯片的几何结构变得越来越小，因为更小的晶体管具有更小的成本、更高的速度和更低的功耗。但是，这就要求更低的输入电压，这个电压近年来在不断降低（表4-2）。

这些电源输入（实际上是大量的电源和地引脚）用于FPGA的内部逻辑，所以称之为核心电压。然而，不同的I/O标准可能使用明显不同于核心电压的电平，所以通用I/O的每个簇具有自身额外的电源输入引脚。

表4-2 输入电压与工艺节点

年	输入（内核电压）	工艺节点
1998	3.3	350
1999	2.5	250
2000	1.8	180
2001	1.5	150
2003	1.2	130

有一点很有趣，从350nm节点开始核心电压与工艺具有正比例关系。然而，由于物理原因，这个规律不能持续到1V以下（这些原因与晶体管输入变换阈值和电压降有关），所以这个“电压阶梯”在不远的将来将不会再使用了。

## 4.13 吉比特传输

在器件之间传输大量数据的传统方法是使用一个总线，一个承载相似数据并且执行一个通用功能的信号集合（图4-23）。

早期基于微处理器的系统使用8比特总线来传输数据。由于需要传输更多的数据，而且需要它们传输的速度更快，总线增加为16比特宽，然后是32比特、64比特等。问题是这样要求在器件上的大量引脚和连接引线。随着电路板复杂度的增加，把这些引线布得具有相同的长度和阻抗变得非常困难。此外，当你面对大量的总线时，信号完整性问题（比如对噪声的敏感度）的处理也越来越困难。

由于这个原因，今天的高端FPGA包括特殊的硬线吉比特传输模块。这些模块

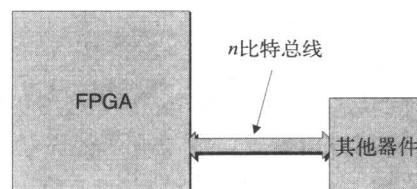


图4-23 在器件之间使用一个总线通信

使用一差分信号对（这对信号总是传送相反的逻辑值）来传输（TX）数据，另一对来接收（RX）数据（图4-24）。

这些传输器工作于极高的速度，每秒能够传输和接收数以十亿计的比特。此外，每个模块实际上能够支持一定数量（比如4个）的传输器，而一个FPGA可以包含很多这样的传输模块。

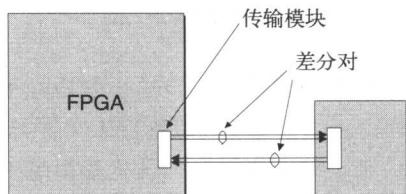


图4-24 在器件之间使用高速传输器通信

## 4.14 硬IP、软IP和固IP

每个FPGA制造商提供他们自己的硬IP、软IP和固IP选择方案。硬IP提供了预先实现的模块，比如微处理器核、千兆比特接口、乘法器、加法器、MAC功能，等等。在功耗、硅面积和性能一定的情况下，这些模块设计得尽可能高效。这些模块的不同组合，与不同数量的可编程逻辑块组合在一起就构成了一个FPGA系列。

93

在谱系的另一端，软IP指一个可以包括在用户设计中的高性能源代码库。这些功能一般使用硬件描述语言（或HDL），如Verilog或VHDL在寄存器传输级（RTL）的抽象层次上表示。设计者决定使用的任何软IP都要插入到设计的主体中，也是寄存器传输级描述，接下来综合为一组可编程逻辑块（可能包括一些像乘法器那样的硬IP）。

位于中间位置的就是固IP，它也是一些高层次功能某种形式的库。然而不像软核，这些功能已经被优化映射、布局布线成为一组可编程逻辑块（可能包含一些硬IP如乘法器等）。在设计中可以按照要求例化（调用）一个或多个这种预制固IP块的副本。

问题是，在这些使用硬IP实现效果最佳的功能和那些应该使用软IP或固IP（使用大量的通用可编程逻辑块）实现的功能之间很难画出一道分界线。对于像乘法器、加法器和MAC这些本章前面讨论过的功能，在广泛的领域中有普遍的应用。另一方面，一些FPGA包含专用的模块来处理特殊的接口协议，比如PCI标准。这样，如果这就是你希望的器件与电路板连接的方式，一切就简单了很多。但是，如果你决定使用一些其他接口，专用的PCI模块只能在你的芯片中浪费空间、增加功耗。

一般说来，一旦FPGA制造商在他们的器件中加入了类似的功能，他们实质上已经把这个元件放到一个相应的环境中了。有时为了达到预期的性能，你必须这么做，这是一个经典的问题，因为下一代的出现往往很快，可以使用它的主体

94

(可编程) 结构来执行这个功能。

## 4.15 系统门与实际的门

在ASIC世界里器件容量的常用度量衡是等效门。这是因为不同的制造商在单元库里提供了不同的功能模块，而每个功能模块的实现都要求不同数量的晶体管。这样在两个器件之间比较容量和复杂度就很困难。

解决办法就是给每个功能赋予一个等效门数值，就比如“A功能模块等于5个等效门，B功能模块等于3个等效门……”下一步就是统计每个功能模块，把它们转换成相应的等效门值，把这些值相加，然后就可以自豪地公布，“我的ASIC包括一千万个等效门，这要比你的ASIC大多了！”

但是，事情没有那么简单，不同的厂商对等效门实际结构的定义是不同的。通常情况下，一个2输入NAND功能表示一个等效门。也有一些厂商定义一个等效门等于一个任意数目的晶体管。还有的厂商定义一个ECL等效门为“实现一个单比特全加器所要求最小逻辑的十一分之一”。(这到底是谁想出来的？)通常，最好的办法是在投资之前先确信大家在谈论同样的事。

我们回到FPGA来，FPGA制造商遇到了一个问题，他们试图建立一个基础用于他们的器件和ASIC作比较。例如，如果说有的人有一个现成的包含500 000个等效门的ASIC设计，他想把这个设计变为用FPGA实现，他应该怎样描述这个设计需要的FPGA呢。事实上每个4输入LUT能够表示从1~20个2输入基本逻辑门所能表示的任何功能，所以这样的比较相当微妙。

为了解决这个问题，FPGA制造商在20世纪90年代早期开始讨论系统门。有人说这是个代价高昂的想法，在ASIC设计中才会涉及这种专门术语。而另外一些人说这纯粹是一个市场策略，没有给任何人带来好处。95

与此同时，似乎没有清晰的定义来解释什么是系统门。在FPGA实质上只包含LUT或寄存器形式的通用可编程逻辑资源时，这很令人尴尬。在那时甚至很难界定一个包含 $x$ 个等效门的专门ASIC设计是否能够用一个包含有 $y$ 个系统门的FPGA来代替。这是因为有的ASIC设计者可能在组合逻辑方面具有优势，而另外一些则可能更偏重使用寄存器。这两种情况得到的结果可能是一个在FPGA上的次最优映射。

在FPGA开始包含内嵌的RAM块，因为有的功能使用RAM实现要比通用逻辑实现效率更高。而且，事实上LUT可以当作分布RAM来使用，例如有的厂商系统门计算值现在包括一个定语，“假设20%~30%的LUT是作为RAM来使用的”。当然，在我们开始认为FPGA包含内嵌处理器核和类似功能时，这个问题更加严重了。于是，有的厂商现在说：“系统门数值没有计入这些元件”。

96

到底有没有简单的规则来把系统门转换成等效门呢？其实有很多。有的人认为如果你感觉乐观，你应当把系统门数值除以3（比如三百万FPGA系统门应该等于一百万ASIC等效门）。或者如果你看到更多的是悲观的那一面，你可以把系统门除以5（这样三百万系统门将会等于600 000D等效门）。

然而其他人认为，只有在你假定系统门数值包括了所有能使用通用可编程逻辑和块RAM实现的功能时，以上规律才是正确的。这些人会接着说，如果你把块RAM从方程中去掉，你就必须把系统门数值除以10（这时，三百万系统门就只能等于300 000等效门），但是这时你仍然可以使用块RAM。

最后，这个问题陷入了这样一个泥潭，甚至FPGA制造商都不愿再谈论系统门。对于新出现在人们视野的FPGA，人们很惬意地想像着等效门，而且方便用LUT（查找表）、SLICE等考虑设计；然而，大量的FPGA设计者更习惯于用FPGA的名词。由于这个原因，有的人仍然保留了传统的习惯，我更愿意看到的是FPGA的具体规定，而且使用简单的计算来比较。

- 逻辑单元或逻辑元素或其他的什么（等于4输入LUT和相应的寄存器/锁存器）的数目；
- 内嵌块RAM的数目（和大小）；
- 内嵌乘法器的数目（和大小）；
- 内嵌加法器的数目（和大小）；
- 内嵌MAC的数目（和大小）；
- 等等。

97

为什么会这么困难？对一个真实世界里的ASIC设计实例的全面描述，给出它们的等效门，包括细节如它们的寄存器/锁存器、原语门和其他更复杂的功能，是很有用处的。这些设计实例在FPGA中实现所要求LUT和寄存器/锁存器的数量，还有内嵌RAM和其他内嵌功能的数目就与此有关。

尽管现在还不理想，当然，因为在FPGA和ASIC中人们的理解毕竟不一样的，但是总会有一个开始。

## 4.16 FPGA年

我们都听说过狗的1年相当于人的7年，这个意思就是说一条10岁的狗就相当于人类的70岁。这样说其实没有任何意义，另一方面，在你不能对长远的某些东西保持判断时，这确实提供了一个有用的参考框架。你可以说：“好吧，这只是个期望，因为这个可怜的家伙快100岁了”（或别的什么）。

类似地对于FPGA，我们或许可以这么认为，他们的一年大约相当于人类的15年。这样，如果你使用的是在上一年进入市场的FPGA，你应该视其为十几岁。一

方面，如果你对未来抱有很高的希望，他或她可能最后成长为诺贝尔和平奖得主或美国的总统。另一方面，要实现你的目标将会有一些困难，你必须适应它，学习与之相关的一些知识。

一个FPGA到了上市两年的时候（相当于一个30岁的人了），你可以把它当作一个很成熟的人，而且它的能力尖峰也变得有些圆滑。经过3年以后（45岁了），FPGA开始有些沉稳就像一个中年人，而到了第4年（60岁），你应当尊重他，而且一定不要试图让他像拉车的马一样工作。

# 第5章 FPGA编程（配置）

## 5.1 引言

先说点耍滑头的话。（谚语云：雄鹰可翱游在天，滑头的黄鼠狼却不会被卷入天上飞机的引擎。）

每个FPGA厂商都有自己独特的术语和自己的技术和协议，不同器件族的FPGA编程细节是不一样的。由于这个原因，后面的讨论仅限于这个主题的一般介绍。

## 5.2 配置文件

本书的第2个单元描述了一系列的工具和流程，可以用来实现FPGA设计。这些处理的最终结果是一个配置文件（有时候称之为比特文件），包含着将要下载到FPGA的信息，这些信息用于对FPGA编程使之执行特定功能。

对于以SRAM为基础的FPGA，配置文件包括配置数据（用来直接定义可编程逻辑单元状态的比特）和配置命令（告诉设备如何处理配置数据的指令）。在配置文件载入器件的过程中，这些信息转换成我们所说的配置比特流。

以E<sup>2</sup>和FLASH为基础的器件编程时采用的方式与他们的SRAM表亲类似。相比之下，对于反熔丝为基础的FPGA，配置文件主要包括生成反熔丝所使用的配置数据。

## 5.3 配置单元

对FPGA进行编程（配置文件载入器件）所涉及的基础概念是相对简单的。然而，这个过程有关各个方面内容能够如细流般包围人的大脑，所以我们将从基础部分开始我们的行程。在一开始，假设有一个非常简单的器件，只包括一个简单的可编程逻辑块，周围被可编程互连围绕着（见图5-1）。

任何一种能够被编程的器件都需要利用专门的配置单元。大部分FPGA都是基于使用SRAM单元，但是有些使用FLASH单元，而其他的使用反熔丝。

不考虑基本的技术工艺，器件的内部互连具有大量的相关单元能够配置，用于器件的基本输入和输出与可编程逻辑块的连接，以及这些逻辑块彼此的互连。（对于器件基本输入输出，在图5-1中没有显示出来，这些输入输出都有大量的相

关单元，可以配置它们来适应特殊的I/O标准等。)

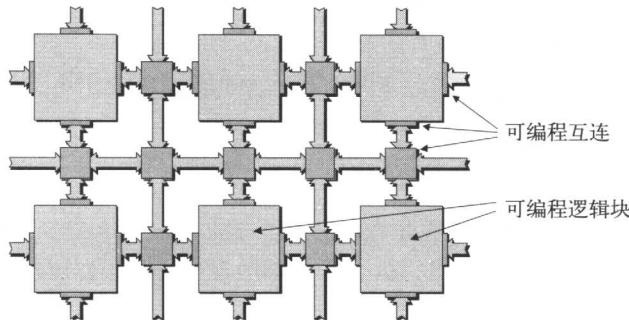


图5-1 简单FPGA结构的自顶向下视图

对于我们所讨论的部分，我们应该设想每个可编程逻辑块只包括一个4输入LUT、一个多路复用器和一个寄存器（图5-2）。多路复用器要求一个相应的配置单元来确定选择的是哪个输入。寄存器要求相关的单元来确定它是作为边沿触发器（如图5-2所示）还是一个电平敏感的锁存器，以及它是时钟的上升沿触发还是下降沿触发（在触发器的情况下），或者是低有效使能还是高有效使能（如果寄存器被当作一个锁存器使用），还有它初始化为逻辑0还是逻辑1。同时，4输入LUT本身就是基于一个16配置单元。

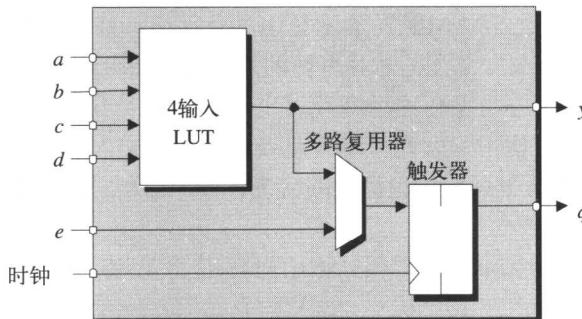


图5-2 一个非常简单的可编程逻辑块

## 5.4 基于反熔丝的FPGA

在反熔丝为基础的FPGA里，我们可以把反熔丝单元形象地看成散布在器件表面上必要位置上。器件被置于专门的器件编程器中，配置文件从宿主计算机上传到器件编程器中，器件编程器根据这个文件产生较大的电压和电流脉冲输入选择的引脚，按顺序熔断每个熔丝。101

一个非常简单的方法是令每个反熔丝都具有一个在芯片表面“虚拟”的 $x$ - $y$ 坐

标，这些 $x$ - $y$ 值被定义为整数。在这个背景下，我们可以形象地使用一组I/O引脚来代表与一个特定反熔丝有关的 $X$ 值，而另一组引脚代表 $Y$ 值。（在实际情况中要复杂得多，但这是个绝妙的方法。）

当所有这些熔丝都被处理后，FPGA从器件编程器中取出来，然后连接到电路板中。当然，反熔丝器件是一次性可编程器件（OTP），因为一旦你开始了配置过程，再改变你的想法已经来不及了。

**基于FLASH（和E<sup>2</sup>）的器件的编程方式一般与基于SRAM的器件类似。**

与基于SRAM的器件不同，基于FLASH的器件是非易失性的，当系统断电时可以保持它们的配置，而且它们不需要在系统再次上电时重编程（尽管如果有要求的时候，它们也可以这么做）。

同样，基于FLASH的器件也可以在系统（在电路板上）编程，或者通过器件编程器在系统外编程。

## 5.5 基于SRAM的FPGA

接下来要考虑的只有基于SRAM的FPGA了。我们还记得这些器件是易失性的，这意味着它们必须在系统（在电路板上）编程，而且在每次系统上电时都需要重新编程。

我们可以形象地把SRAM配置单元看作单个（长）移位寄存器。考虑一个只显示I/O引脚和SRAM配置单元的芯片表面的鸟瞰图（图5-3）。

在一开始，我们应当设想这个寄存器链的开始和结尾可以由外界直接访问。然而，这只是适合于使用配置端口编程机制，结合FPGA作为主设备串行下载或FPGA作为从设备串行下载的编程模式的情况，如下面所讨论的那样，认识到这一点很重要。

102

也要指出，图5-3显示的配置数据输出引脚/信号只用于有多个FPGA串行（菊

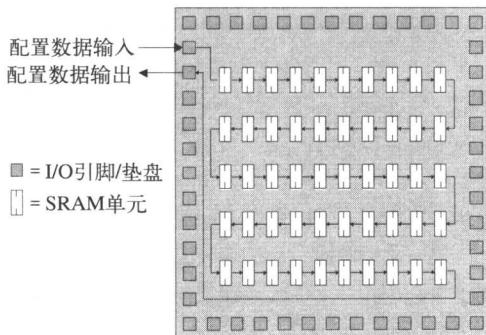


图5-3 把SRAM单元视为一个长移位寄存器

花链)方式配置时,或者器件由于某种原因需要能够读出配置数据的情况。

### 5.5.1 迅速的过程欺骗了眼睛

了解这些内容并不是很有必要,所以如果你着急,你可以直接进入下一节。然而这部分其实很有趣,我想你也会发现这一点的。如图5-3所示,对SRAM编程单元的内部结构最简单的直观认识就是一个长移位寄存器。如果确实是这种情况,那么每个单元都可以被实现为一个触发器,并且链内所有的触发器都将由同一个时钟驱动。

问题是一个FPGA包括大量的配置单元。比如,到2003年一个高端的器件很容易包括大约2500万个单元!一个触发器的核要求8个晶体管,而一个锁存器的核只要求4个晶体管。所以,基于SRAM的FPGA中的配置单元是由锁存器形成的。(我们举例的器件有2500万个配置单元,就会节约1亿个晶体管,这是很可观的。)

FPGA编程需要大量的时间。想象一下,一个比较高端的器件包括2500万个基于SRAM的配置单元。使用串行模式和一个25MHz的时钟对这样一个器件编程将需要一秒的时间。这在你第一次对系统加电时还不算坏,但是这意味着在器件运行时,你确实不希望经常对FPGA重编程。

问题在于你不能从锁存器中创造出移位寄存器(好吧,实际上你能,正如本章后面部分所讨论的那样,但是长度不是2500万个单元)。FPGA制造商们的对此的解决办法是一组触发器,或者1024个共用一个时钟并且配置成一个标准的移位寄存器。这个组叫做帧。

器件实例中的2500万个配置单元/锁存器也被分成很多帧,作为触发器帧它们具有相同的长度。在外部世界看来,2500万比特的配置数据在单一时钟作用下进入器件。但是,在器件内部,当第一个1024比特串行载入触发器帧时,专门的内部电路自动地并行载入/复制这个数据到第1个锁存器帧。当下一个1024比特载入触发器帧时,它们自动地载入/复制到第2个锁存器帧,剩下的部分依此类推。(数据从器件读出的过程正好相反。)

### 5.5.2 对嵌入式(块)RAM、分布RAM编程

在FPGA包括大块嵌入式(块)RAM的情况下,这些块的核心是用SRAM锁存器实现的,而且这些锁存器就是形成一部分我们假想中寄存器链的配置单元(正如前面部分所讨论的)。

有趣的一点是每个4输入LUT(见图5-2)能够配置成一个LUT、一个小( $16 \times 1$ )块分布RAM或一个16比特移位寄存器。所有这些表现形式可以用相同的16位SRAM锁存器实现,这些锁存器也形成了我们假想寄存器链的一部分。

[104] “到底16比特移位寄存器是什么东西？”你快要喊出来了，“难道它不需要用真正的触发器来实现吗？”很好，这是个好问题，很高兴你提出这样的问题。实际上，技巧性的电路采用了电容性锁存器概念来防止典型竞争条件（在20世纪60年代早期，设计者大致上以同样的方式用分立的晶体管、电阻和电容来构建触发器）。

### 5.5.3 多编程链

图5-3显示的配置单元呈现为单个编程链。由于配置单元数量达上千万，这个链确实非常长。有的FPGA的被设计为配置部分实际上驱动大量的短链。这样允许配置器件的个别部分，而且对于一系列概念如模块化和增量设计都是有帮助的（在第2部分讨论这些概念的细节）。

### 5.5.4 器件的快速重新初始化

如前所述，可编程逻辑块中的寄存器有个相应的配置单元，确定寄存器初始化为逻辑0还是逻辑1。一般的FPGA系列都会提供某种机制，如初始化引脚，使能这个引脚时，所有的寄存器都将恢复为初始值（这个机制不能恢复任何嵌入式[块]或分布RAM）。

## 5.6 使用配置端口

早期的FPGA利用了配置端口。甚至到今天，可以使用更复杂的技术（如本章稍后要讨论的JTAG接口），配置接口仍在广泛的使用，因为它相对简单，可以被FPGA的拥护者很好地理解。

我们以一组专门配置模式引脚作为开始，这些引脚用于向器件表明将使用哪一种配制模式。在早期，仅仅用2个引脚来提供4种模式，如表5-1所示。

[105] 注意在表中显示的模式名称——还有在模式引脚上的编号之间和模式之间的关系——表明这只是作为一个例子使用的。实际的编号和模式名称因制造商而异。

表5-1 4种早期的配置模式

模式引脚	模 式
0 0	FPGA作为主设备串行下载
0 1	FPGA作为从设备串行下载
1 0	FPGA作为主设备并行下载
1 1	FPGA作为从设备并行下载

模式引脚一般在电路板级硬线连接为需要的逻辑0或逻辑1（这些引脚可以被其他逻辑驱动，允许调整编程模式，但是在实践当中很少这样做）。

除了硬线的模式引脚，有一个额外的引脚用于指示FPGA实际配置过程的开始，而另一个引脚被器件用来报告何时这个过程结束（也用于确定是否这个过程中有错误发生）。这意味着除了系统初次上电配置FPGA的时候，如果有需要，器件也可以使用原始配置数据重新初始化。

配置端口也利用了额外的引脚来控制数据的下载和数据本身的输入。这些引脚的数量取决于所选择的配置模式。值得指出的是，当配置结束以后，这些引脚中的大部分可以接着作为通用I/O引脚来使用（稍候我们会返回这个话题）。

### 5.6.1 FPGA作为主设备串行下载

这或许是最简单的编程模式。在早期，它需要使用一个外部的PROM。后来被EPROM取代，再后来是E<sup>2</sup>PROM，而到了现在——最常用的，是一个基于FLASH的设备。这个专用存储元件有一个数据输出引脚，连接到FPGA上的配置数据输入引脚（图5-4）。

106

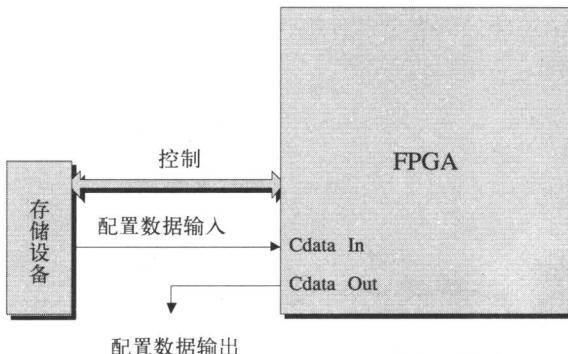


图5-4 FPGA作为主设备串行下载

107

FPGA也使用几个比特来控制外围存储设备，比如，当FPGA准备开始读数据时，要发出一个复位信号和数据输出使用的时钟信号。

这个模式的思想是FPGA不需要为外部存储设备提供一系列地址。它用简单的脉冲信号来表明自身希望开始从起始端读取数据，接着它发送一系列的时钟脉冲使配置数据从存储器件中读出。

只有在由于某些原因需要从器件中读出配置数据时，来自FPGA的配置数据输出信号才需要连接起来。这一般发生在电路板上有多个FPGA的情况下。此时，每个FPGA都有自己专用的外部存储设备，能够被单独配置，如图5-4所示。这些FPGA也可以级联（菊花链）在一起，共用一个外部存储器（图5-5）。

这种情形下，链中的第一个FPGA（直接与外部存储器相连的那个）将使用串行主模式来配置，而其他的将使用串行从模式，本章后面将讨论这个内容。

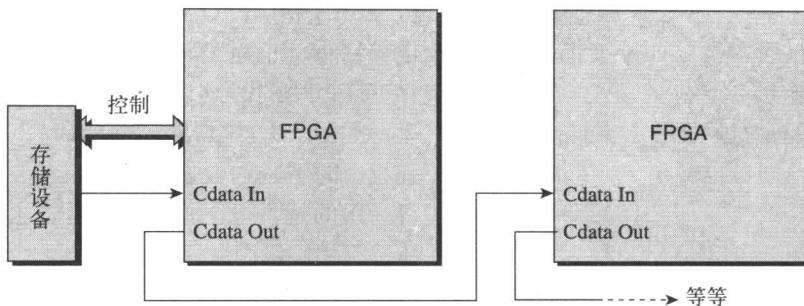


图5-5 菊花链中的FPGA

### 5.6.2 FPGA作为主设备并行下载

这种模式在很多方面都类似于前面的模式，除了数据是以8比特的块从存储器件的8个输出引脚中读出的。8比特是非常普通的，一般称为字节。除了提供控制信号，最初的FPGA还为外部存储器件提供一个地址，用于指出接下来载入的是哪个配置数据（图5-6）。

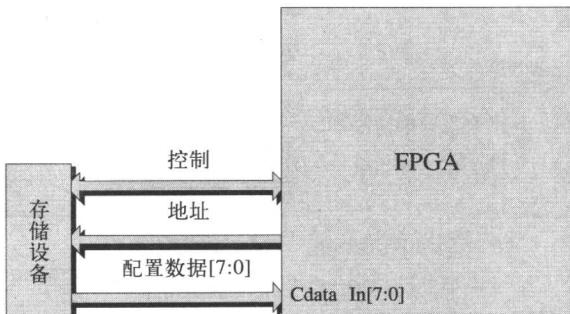


图5-6 FPGA作为主设备并行下载（早期的技术）

这种工作方式下，FPGA具有一个内部计数器用于为外部存储器产生地址（早期的FPGA有24比特的计数器，允许它们对16兆比特的数据选址）。在配置序列的起始位置，这个计数器将初始化为零。计数器所指向的数据字节被读取以后，计数器将递增指向下一个字节的数据。这个过程将持续下去直到所有的数据都被载入。

当电子设备和计算机最早出现时，对它们的定义是相当随意的。造成的结果就是不同的公司对于字节之类都有它们自己的名称。而且经常可以看到5、6、7、8甚至9比特的字节。这种情况延续了很久，直到一致同意使用8比特的字节。

很容易看出，并行下载技术具有速度优势，然而它并不是为了一些时间。这是因为，在早期的器件里，在一个字节的数据读入器件时，在时钟节拍控制下以串行的方式进入内部配置移位寄存器。幸好，这一点在现代的FPGA中已经修正过了。另一方面，尽管这8个引脚在配置数据载入后可以作为通用I/O引脚使用，实际上这种做法仅停留在纸面上。这是因为这些引脚仍然由引线与外部存储器件相连，这会造成一些信号完整性问题。

这种往日的技术还如此流行的主要原因是，在串行配置FPGA主模式下使用的专用存储器件相当昂贵。相比之下，并行技术允许设计者使用相当便宜的架外存储器件。

1846年，德国 Gustav Kirchhoff 定义了电路的基尔霍夫定律。

说了这么多，用于FPGA的专用存储器件现在相对还不昂贵（作为基于FLASH的器件，也可以重复使用）。这样，目前现代FPGA使用了一种新改进的并行下载技术。这种情况下，外部存储器是专用器件，不要求外部地址，这样FPGA就不需要专门的内部计数器（图5-7）。

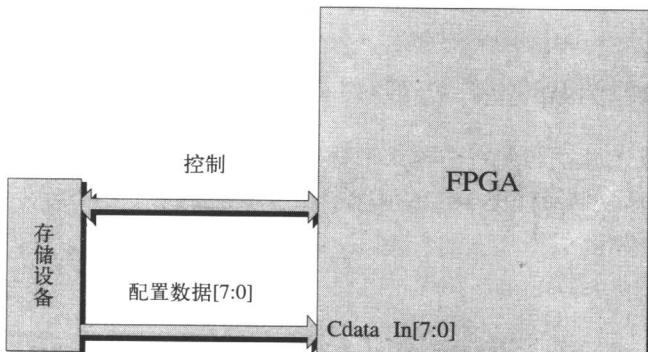


图5-7 FPGA作为主设备并行下载（现代技术）

由于前面讨论过的串行模式，FPGA给外部存储器件发出单脉冲复位信号以表明它要开始从头读取数据，接着它发出一系列时钟脉冲作存储器件输出配置数据的时钟信号。109

### 5.6.3 FPGA作为从设备并行下载

以上所讨论的FPGA作为主设备模式，是很吸引人的，因其具有固有的简单性，而且只需要FPGA自身，连同一个简单的外部存储器件。

但是，大量的电路板包括一个微处理器，使之完成大量的事务管理任务，设计者可能会使用微处理器来对FPGA进行载入（图5-8）。110

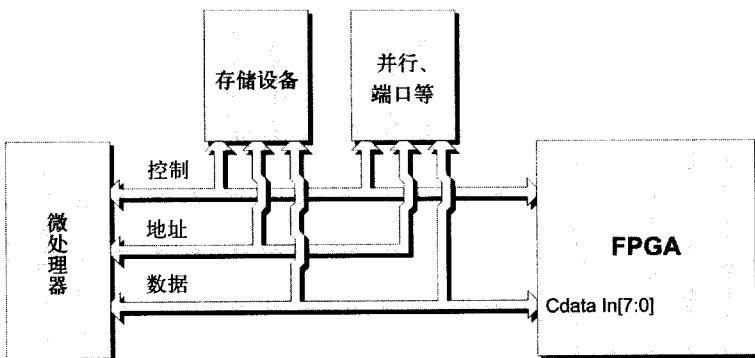


图5-8 FPGA作为从设备并行下载

这种方法是使微处理器作为控制者。微处理器通知FPGA何时开始配置过程。接着从适当的存储器（外围或别的什么）中读取一个字节的数据，把数据写到FPGA中，从存储器件中读下一个字节，再把这个字节写到FPGA里，如此重复直到配置过程结束。

这种方式具有传送量大的优势，而且除此以外，微处理器核还可以查询它周围系统所处的环境，接着选择相应的配置数据载入FPGA。

#### 5.6.4 FPGA作为从设备串行下载

这种模式几乎与并行主模式一样，除了载入FPGA的数据是单个的比特（微处理器仍然从存储器件中一次读取一个字节，但是接着把这些数据转换为串行比特流写入FPGA）。

这种方式的主要优点是它使用更少FPGA的I/O引脚。这意味着，在配置过程之后，只有一个单独I/O引脚具有与微处理器总线相连的额外走线。

### 5.7 使用JTAG端口

与其他很多现代器件一样，今天的FPGA配备有JTAG端口。支持联合测试行动组和IEEE1149.1的技术规范。最初JTAG是设计用于实现测试电路板和集成电路的边界扫描技术的。

JTAG和边界扫描的细节超出了本书的范围。在这里，我们懂得FPGA有一些引脚是作为JTAG端口来使用的就足够了。其中一个引脚用来输入JTAG数据，另一个输出这些数据。每个FPGA都为I/O引脚保留了一个相应的JTAG寄存器（一个触发器），这些寄存器连在一起构成菊花链（图5-9）。

1847年，英格兰George Boole公布了他的数理逻辑思想。

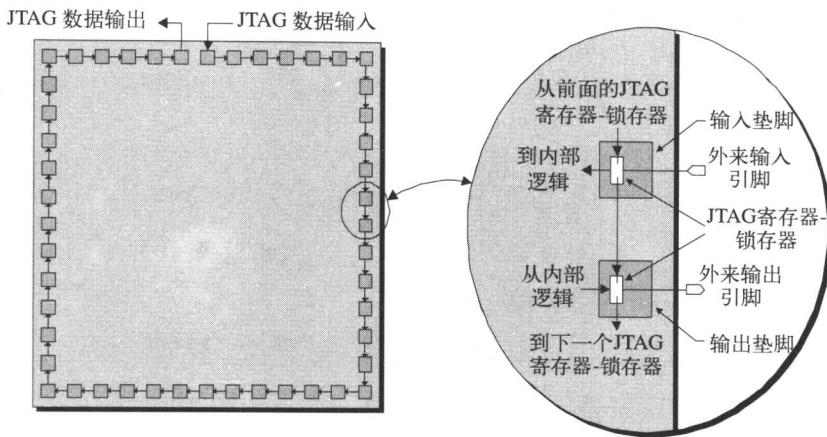


图5-9 JTAG边界扫描寄存器

边界扫描背后的思想是，通过JTAG端口，使串行时钟数据进入与输入引脚有关的JTAG寄存器，使器件（这里指的是FPGA）操作这些数据。这个过程的结果保存在与输出引脚相关的JTAG寄存器中，而且最终，此结果在串行时钟作用下从JTAG端口读出。

然而，JTAG器件也包括一些额外的JTAG相关逻辑。而且，对于FPGA来说，JTAG的使用远远超出了边界扫描。比如说，利用JTAG端口的数据输入引脚把专门的命令载入一个专用的JTAG命令寄存器中（没有显示在图5-9中）。一个这样的命令可以指示FPGA把内部的SRAM配置移位寄存器连接到JTAG链上。这种情况下，JTAG端口可以用于配置FPGA。这样，今天的FPGA能够支持5种不同的编程模式。因此，要求使用3个模式引脚，如表5-2所示（未来可能还会增加新的模式）。

112

表5-2 现在的五个配置模式

模式引脚	模 式
0 0 0	FPGA作为主设备串行下载
0 0 1	FPGA作为从设备串行下载
0 1 0	FPGA作为主设备并行下载
0 1 1	FPGA作为从设备并行下载
1 × ×	仅使用JTAG端口

注意，JTAG端口是一般的FPGA都具备的，所以器件能够通过传统的配置端口使用标准配置模式中的一种来初始化，并且可以接下来按照需要使用JTAG端口来重新配置。或者，器件也可以只使用JTAG端口来配置。

## 5.8 使用嵌入式处理器

等一等，事情还没有完！在第4章，我们讨论了一些FPGA运行嵌入式处理器

核，而且这些核都具有它们自己的专用JTAG边界扫描链。考虑只有一个嵌入式处理器核的FPGA的情况（图5-10）。

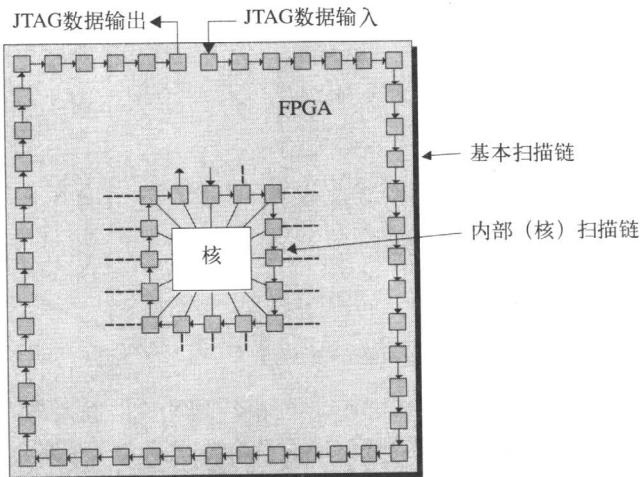


图5-10 嵌入式处理器边界扫描链

FPGA本身应该只有一个外部JTAG端口。如果需要，JTAG命令可以通过这个端口载入，指示器件把处理器局部的JTAG链与器件的主JTAG链相连接（这取决于制造商。也有可能在默认情况下，这两个链是连接着的，使用一个相反的命令使内部的链脱离）。

这里，JTAG端口可以用来初始化内部微处理器核（和有关的外围设备），使配置过程的主体部分接下来能够移交给核。某些情况下，核可能用来查询FPGA所处的环境，接着选择配置相应的数据载入FPGA。

# 第6章 谁在参与游戏

## 6.1 引言

第1章曾经提到，本书不局限于特定的FPGA制造商或专门的FPGA器件，因为新的特性和器件类型不断出现。在可能的范围内，本书也避免提及单独的EDA提供商或他们的工具。因为这个领域是如此的不稳定，以致于那些工具名称和特性设置第二天就可能会发生变化。

尽管如此，本章提供了一个关于FPGA及相关领域关键FPGA或EDA厂商的指引。

## 6.2 FPGA和FPAA提供商

本书的主要内容都是关于数字FPGA的。然而，说起来也挺有趣，目前也可以获得现场可编程模拟阵列（field-programmable analog arrays, FPAA）。而且，与提供FPGA器件正好相反，有的公司专门提供FPGA IP核作为ASIC标准单元或ASIC结构化设计的一部分。

公司	网址	产品
Actel Corp.	<a href="http://www.actel.com">www.actel.com</a>	FPGA
Altera Corp.	<a href="http://www.altera.com">www.altera.com</a>	FPGA
Anadigm Inc.	<a href="http://www.anadigm.com">www.anadigm.com</a>	FPAA
Atmel Corp.	<a href="http://www.atmel.com">www.atmel.com</a>	FPGA
Lattice Semiconductor Corp.	<a href="http://www.latticesemi.com">www.latticesemi.com</a>	FPGA
Leopard Logic Inc.	<a href="http://www.leopardlogic.com">www.leopardlogic.com</a>	嵌入式FPGA核
QuickLogic Corp.	<a href="http://www.quicklogic.com">www.quicklogic.com</a>	FPGA
Xilinx Inc.	<a href="http://www.xilinx.com">www.xilinx.com</a>	FPGA

## 6.3 FPNA 提供商

这是个有点微妙的类别，不仅仅是因为现场可编程节点阵列（field programmable nodal arrays, FPNA）这个名字是由笔者在写下这些词之前几秒钟刚刚发明的（他只是团队中的一员）。这里的思想是，每个这种器件都是由节点阵列构成一个百万级的粗粒结构，每个节点都是可以涵盖ALU类型操作到FFT算术功能任务的复杂处理单元，以各种方式来实现通用微处理器核。

在传统意义上这些器件已经不是FPGA了。然而，在今天的环境下要定义什么是FPGA而什么不是FPGA是有些愚蠢的。公平地讲，带有嵌入式RAM、嵌入式微处理器和千兆比特收发器的FPGA已经不是传统意义上的FPGA了。对于FPNA，这些器件也是数字的和现场可编程的，所以在这里应当提它们一下。

在写下这些文字的时候，有30~50家公司在认真地实验不同风格的FPNA，其中最令人感兴趣的几个典型公司列举如下（在第23章将再次看到他们）：

公 司	网 址	产 品
Exilent Ltd.	<a href="http://www.elixent.com">www.elixent.com</a>	基于ALU的节点
IPflex Inc	<a href="http://www.ipflex.com">www.ipflex.com</a>	基于操作的节点
Motorola	<a href="http://www.motorola.com">www.motorola.com</a>	基于处理器的节点
PACT XPP Technologies AG	<a href="http://www.pactxpp.com">www.pactxpp.com</a>	基于ALU的节点
picoChip Designs Ltd.	<a href="http://www.picochip.com">www.picochip.com</a>	基于处理器的节点
QuickSilver Technology Inc.	<a href="http://www.qstech.com">www.qstech.com</a>	算法元素节点

## 6.4 全线EDA提供商

每个FPGA、FPAA和FPNA提供商提供与他的器件有关的设计工具的选择。对于FPGA来说，这些工具都要包括布局—布线引擎。FPGA提供商也可能会从其他EDA公司获得OEM版本的工具（通常是“瘦”版本）。（在这里的上下文中，OEM意味着FPGA提供商从第三方获得软件许可证，接着把它封装到自己的环境中作为其中一部分提供给用户。）

首先，我们了解一下大亨们——能够提供完整解决方案的EDA全线产品提供商（准确地说，这些解决方案中可能会包括个别软件的OEM版本，后面小节将提到这些特定EDA提供商）。

公 司	网 址	产 品
Altium Ltd.	<a href="http://www.altium.com">www.altium.com</a>	FPGA上的系统硬件-软件设计环境
Cadence Design Systems	<a href="http://www.cadence.com">www.cadence.com</a>	FPGA 设计基础和仿真 (OEM综合)
Mentor Graphics Corp.	<a href="http://www.mentor.com">www.mentor.com</a>	FPGA 设计基础、仿真和综合
Synopsys Inc.	<a href="http://www.synopsys.com">www.synopsys.com</a>	FPGA 设计基础、仿真和综合

这个世界的事总是要更复杂一些。比如，对于像“三巨头”那样的公司来说，相对较小的公司看上去像个陌生的群体。然而，在FPGA的领域里，Altium提供了一个完整的硬件和软件协同设计环境用于在FPGA上系统（system-on-FPGA）的开发。它包括了设计输入、仿真、综合、编辑/编译、调试工具和一个多FPGA开发板。

## 6.5 专业FPGA和独立EDA提供商

与购买现成的解决方案相反，有的设计团队更愿意利用大量EDA公司的工具

来创建自己的定制环境。通常情况下，这些工具比来自全线提供商的产品更为便宜，但是它们可能也没有那么强大和复杂的功能。有时，规模稍小的提供商也会开发出很好的产品而且大力推广，这些产品更易接受，更关注客户的意见。（正如老人们过去所说的：“你在花自己的钱做出自己的选择”。）

117

公 司	网 址	产 品
0-In Design Automation	www.0-In.com	基于推断的验证
AccelChip Inc.	www.accelchip.com	基于FPGA的DSP设计
Aldec Inc.	www.aldec.com	混合语言仿真
Celoxica Ltd.	www.celoxica.com	基于FPGA的系统设计和综合
Elanix Inc.	www.elanix.com	DSP设计和算法验证
Fintronic USA Inc.	www.fintronic.com	仿真
First Silicon Solutions Inc.	www.fs2.com	FPGA逻辑和嵌入式处理器的片上器件化和调试
Green Hills Software Inc.	www.ghs.com	RTOS和嵌入式软件专家
Hier Design Inc.	www.hierdesign.com	基于FPGA的SVP
Novas Software Inc.	www.novas.com	验证结果分析
Simucad Inc.	www.simucad.com	仿真
Synplicity Inc.	www.synplicity.com	基于FPGA的综合
The MathWorks Inc.	www.mathworks.com	系统设计和算法验证
TransEDA PLC	www.transeda.com	验证IP
Verisity Design Inc.	www.verisity.com	验证语言和环境
Wind River Systems Inc.	www.windriver.com	RTOS和嵌入式软件专家

## 6.6 使用专门工具的FPGA设计顾问

有一些小的设计机构专门从事FPGA设计。其中一些拥有内部的先进设计工具，这里也值得我们看一看。

118

公 司	网 址	产 品
Dillon Engineering Inc.	www.dilloneng.com	ParaCore 架构
Launchbird Inc.	www.launchbird.com	合流系统设计语言和编译

## 6.7 开源、免费和低成本的设计工具

最后，设想一下你想建立一个小的FPGA设计团队，或者把你当作一个FPGA设计顾问，然而这时候你只有很少的一点资金（相信我，我经历过）。对于这种情况，可以在为设计工具花费不超过一点格罗特（groat）的基础上，使用一系列开源、免费和低成本技术来建立一个新的FPGA设计机构。

118

公 司	网 址	产 品
Altera Corp.	www.altera.com	综合和布线
Gentoo	www.gentoo.com	Linux 开发平台
Icarus	http://icarus.com/eda/verilog	Verilog仿真器
Xilinx Inc.	www.xilinx.com	综合和布线
—	www.cs.man.ac.uk/apt/tools/gtkwave/	GTKWave 波形界面
—	www.opencores.org	开源硬件核和EDA工具
—	www.opencollector.org	开源数据库 硬核和EDA工具
—	www.python.org	Python编程语言（用于定制工具和DSP编程）
—	www.veripool.com/dinotrace	Dinotrace波形界面
—	www.veripool.com/verilator.html	Verilator (Verilog转换为C)

如果使用Linux作为开发平台，两个主要的FPGA提供商——Xilinx和Altera现在把他们的工具都转向了Linux平台。Xilinx和Altera也分别提供他们的ISE和Quartus-II设计环境的免费版本（这些环境的完整版本会包括在大多数项目启动时的预算里）。

# 第7章 FPGA与ASIC设计风格

## 7.1 引言

我的母亲对“我是电子工程师”这一事实感到无比骄傲。我能理解——而且牢记——这其中包含着对这个星球上的每一件电子产品（来自任何时代）的绝对和不可动摇的信任。实际上，事实并没有那样令人陶醉，因为我们中间很少有人在任何事情上都是专家。

类似地，一些设计者把他们青春的大部分时光花在看上去没有尽头的ASIC开发上，而另一些人则在他们的小房间里研习属于FPGA大师的领域里那些不可思议的秘密。

当精通其中一门实现技术的设计者突然转入另一个领域时，问题就出现了。例如，常常见到的情形是，懂得ASIC任何知识并受到大家赞誉的设计者突然收到一个要求使用FPGA来实现设计目标的任务。

这是个棘手的问题，因为它需要多方面考虑。在这里我们希望能够提供一个关于ASIC和FPGA之间最显著不同的纵览。

121

## 7.2 编码风格

在语言驱动的设计流程（第9章将进一步讨论）出现后，ASIC设计者经常要编写非常轻便的代码（用VHDL或Verilog），并且使用最少的实例化（专门命名的）单元来实现。

相比之下，FPGA设计者更倾向于例化专门的低级单元。例如，FPGA使用者可能并不乐意使用综合工具来产生像乘法器之类的东西，他们可能会手工处理自己的版本并且在他们的代码中例化这些器件。此外，纯粹的FPGA使用者往往会比ASIC设计者在他们的综合引擎中更多地使用工艺细节属性。

## 7.3 流水线和逻辑层次

### 7.3.1 什么是流水线

人们经常听说流水线这个词，但是很少有人深刻理解这个词。当然，工程师

知道它的意思，然而这本书是面向广大读者的，所以花一点时间来解释这个词是值得的。假设我们在制造小汽车，而且我们手里拿着所有的部件。我们进一步设想其主要步骤。

- (1) 车轮安装到底盘上。
- (2) 发动机安装到底盘上。
- (3) 座位安装到底盘上。
- (4) 车身安装到底盘上。
- (5) 对所有部位喷漆。

122

是的……我知道，我知道。你们这些工程师的牢骚和叹息声我都听到了。我意识到没有安装方向盘或是车灯……，但这只是个图方便的例子。

现在设想这些任务中的每一个步骤都要求专人来完成。一个办法是大家围坐在一旁玩牌，第1个人站起来把车轮安装到底盘上，然后回去玩游戏。接下来，第2个人起身并给底盘安装发动机，然后回去接着玩。现在第3个人走过去安装座椅。当第3个人回来以后，第4个人把车身装上去，等等。当第1辆车完成以后，他们又重新开始这个过程。

很明显，这是非常低效的。如果，我们假设每一步要花一个小时，整个过程就需要5个小时。此外，对于其中每一个小时，只有一个人处于工作中，而其他4个人正在自娱自乐。如果在任一时刻都有5辆车处于装配线上，将会高效得多。这种情况下，当第1个人给第1辆车的底盘安装完车轮后，第2个人开始给这辆车安装发动机，此时第1个人开始给第2辆车的底盘安装车轮。当装配线完全开始运行时，每个人都将持续工作，每个小时都将生产出一辆新车。

### 7.3.2 电子系统中的流水线

实际上，在电子系统中我们常常处于上述的情景。例如，设想我们有个可以123用一系列组合逻辑块实现的设计（或设计中的一个功能），如图7-1所示。

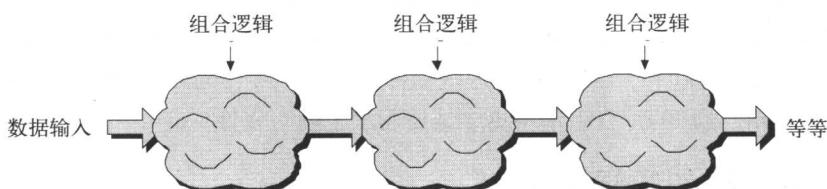


图7-1 仅由组合逻辑实现的功能

假设我们有5个这样的逻辑块（当然，在图7-1中只显示了其中3个），每个块要花费Y纳秒来完成它的任务。这种情况下，传播一个字节的数据通过这个功能要

用 $5 \times Y$ 纳秒，从数据到达第1个块的输入开始，当它离开最后一个模块的输出后结束。

一般来说，我们不希望直到与第1个自己数据有关的输出结果已经存储后，新的一个数据才出现。这意味着我们面临着与低效率汽车装配时同样的情形，这样将花费更多的时间来处理每一字节数据，而且大部分工人（逻辑块）闲置着浪费着大块时间。最好的办法就是对寄存器块之间的组合逻辑“岛”使用流水线设计技术，见图7-2所示。

所有寄存器簇都是由一个通用的时钟信号驱动的。在每个时钟上升沿，寄存器都把前一阶段的结果输出给逻辑块。接着这些值横穿逻辑块到达逻辑块的输出，在那里它们准备在下一个时钟周期被载入后面的寄存器中。

124

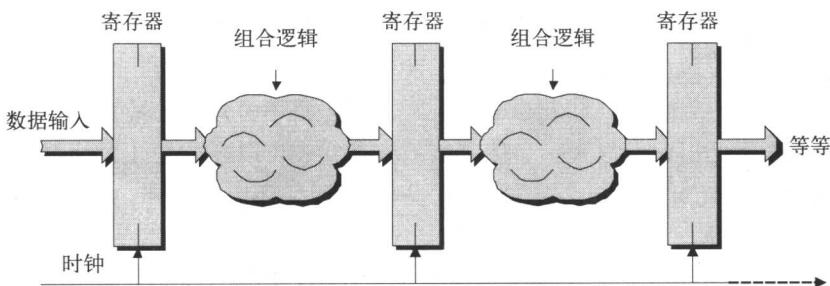


图7-2 流水线设计

这种情况下，当“泵发动起来”并且流水线满负荷运转时，每经过 $Y$ 纳秒就有新的一字节数据处理完毕。

在电子系统的上下文中，等待时间是指某模块的数据通过一个功能，器件或系统所花费的时间（时钟周期）。

理解等待时间的一个办法是回到自动装配线的概念中去。在这种情况下，系统的生产能力可能是每一分钟都从生产线的中开出一辆车。然而，系统的等待时间是整8个小时，因为需要几百个步骤才能完成一辆车（这里的每一个步骤对应着一个流水线设计中的逻辑/寄存器级）。

### 7.3.3 逻辑层次

对于这些活跃的单元，设计工程师必须使之保持均衡。把组合逻辑划分成更小的模块并且增加寄存器级数的数量，然而这也会消耗芯片上的更多资源（和硅片面积），增加了设计的等待时间。

这也是我们要在这里提出“逻辑层次”概念的原因，与在逻辑块输入和输出之间的逻辑门数有关。例如，图7-3所示的是由3个层次的逻辑门组成的，因为在

125

最坏情况下，信号经过3个门才能到达输出。

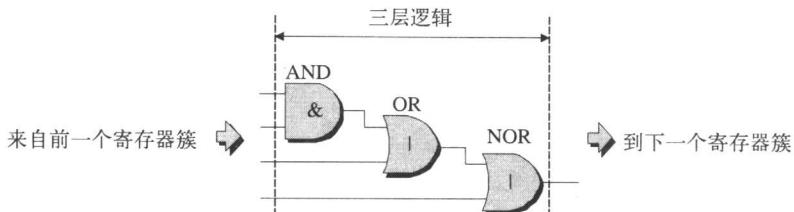


图7-3 逻辑的层

在ASIC的情况下，图7-3所示的一组逻辑门会彼此排列得很紧密，这样它们的线延迟会很小。这就意味着，取决于设计，ASIC工程师有时能够把这些设置得宽松些（比如说，给15层或是更多的逻辑层次布置走线也可以）。

相比之下，如果用FPGA实现这一类的设计，每个门都用一个独立的查找表(LUT)来实现，它将“飞得像一块砖头”（运行得不可思议的慢），因为相对而言FPGA的走线延迟要大得多。当然，实际上查找表可以表达好几层的逻辑（图7-3所示的逻辑功能可以用单个4输入查找表来实现），所以这个问题并不像一开始看上去的那样严重。

所以，归根到底，为了提高（或维持）性能，FPGA设计者比ASIC设计者们更愿意使用流水线技术。更为方便的是，每个FPGA逻辑单元一般都由查找表和寄存器组成，这使输出经过寄存器非常方便。

## 7.4 异步设计实践

### 7.4.1 异步结构

取决于手中的任务，ASIC工程师可能在他们的设计中包括了异步结构，这种  
126 结构需要依赖信号的相对传播延迟才能保证其功能正确。但在FPGA世界里不使用这种技术，因为每次运行布局—布线引擎都会显著地改变其走线（和相对延迟）。

### 7.4.2 组合回路

作为具有一定关联的主题，当产生一个信号需要它自身通过一个或多个逻辑门反馈时，会发生组合回路。当逻辑值取决于布线延迟时，这是主要的临界竞争条件产生来源。尽管根据惯例应该对这种做法皱眉头，但是ASIC设计者还是可以使用这种结构的，因为他们可以把走线（和相应的传播延迟）精确地确定下来。而在FPGA领域这是不可能的，所以这些反馈回路都应该包括一个寄存器单元。

当然，每个锁存器都是基于内部反馈的（而且每个触发器基本上是由两个锁存器构成），然而，这种反馈是由器件制造者紧紧控制的。

### 7.4.3 延迟链

而且，ASIC工程师可能会使用一系列的缓冲或反相器来产生一个延迟链。这些延迟链具有各种各样的用途，比如在设计的异步部分设定竞争条件。除了在FPGA世界里预测这种链产生的延迟很困难，这种类型的结构增加了设计对操作条件的敏感性，降低了可靠性，如果移植到另一种结构或实现技术时可能会比较麻烦。

## 7.5 时钟考虑

### 7.5.1 时钟域

ASIC设计会涉及大量的时钟（有人听说过一个设计包括300多个不同的时钟域）。但是对于FPGA，在任何器件中都只具有数目有限的全局时钟源。所以强烈建议设计者在专用时钟资源范围内预算他们的时钟系统（或者相反，使用通用输入作为用户定义时钟）。

一些FPGA允许它们的时钟树被分割成时钟片断。如果目标工艺确实支持这种特性，在映射为外部或内部时钟时，它将能够被识别和计数。

### 7.5.2 时钟平衡

在ASIC设计的情况下，必须使用专门工艺来平衡通过器件的时钟延迟。相比之下，FPGA具有的全局的、低抖动的时钟布线资源。时钟平衡对于设计者来说是不必要的，因为FPGA制造商已经考虑了这个问题。127

### 7.5.3 门控时钟与使能时钟

ASIC设计经常使用门控时钟技术来减少功耗，如图7-4a所示。但是，这会产生设计的异步性质，并对逻辑门输入密集切换引起的冒险（假信号）敏感。

相比之下，FPGA设计者更倾向于使用使能时钟。最初采用外部的多路复用器来实现，如图7-4b所示；现在，几乎所有的FPGA结构、寄存器自身都有专门的时钟使能引脚，如图7-4c所示。

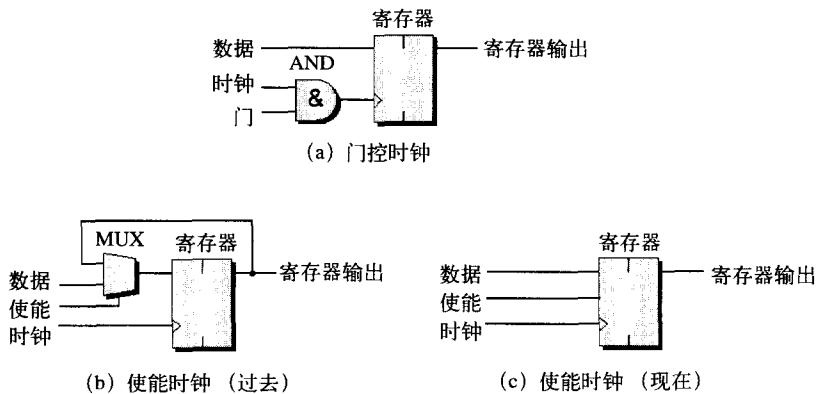


图7-4 门控时钟与使能时钟

### 7.5.4 PLL和时钟调节电路

典型的FPGA包括PLL或DLL功能——可以用于任何一个专用全局时钟（见第4章中的讨论）。如果这些资源用于片上时钟的生成，那么设计中应当也包括一些对它们禁止（disabling）或旁路（bypass）的机制，来辅助芯片的测试或调试。

1859年，德国 Hittorf 和 Pucker 发明了阴极射线管（CRT）。

### 7.5.5 跨时钟域数据传输的可靠性

现实中，这个要求对于ASIC或FPGA设计来说都很重要，在两个独立的时钟域之间进行数据交换时要非常小心，避免数据丢失或错误。差的同步会引起亚稳定性问题和细微的时序问题。为了达到可靠的跨时钟域传输，推荐采用握手、双触发器，或异步FIFO技术。

## 7.6 寄存器和锁存器考虑

### 7.6.1 锁存器

ASIC工程师经常在他们的设计中利用锁存器。作为一个通用的准规则，如果你在设计FPGA，而且你想使用一个锁存器，不要这么做！

### 7.6.2 具有“置位”和“复位”输入的触发器

许多ASIC设计库提供了范围广泛的触发器，有的提供了“置位”和“复位”输入（通常可同时获得同步和异步版本）。

相比之下，FPGA触发器通常可以配置一个置位或复位输入。这种情况下，实现置位或复位输入都要求使用一个查找表（LUT）。所以FPGA设计工程师经常围绕这个作尝试，得到各种不同的实现方案。

### 7.6.3 全局复位和初始化条件

FPGA中的每一个寄存器在编程后都有一个默认的初始条件（也就是说，包含一个逻辑1或逻辑0）。而且，典型的FPGA有一个全局复位信号来恢复所有的寄存器（但是不包括嵌入式RAM）为他们的初始值。一般来说，ASIC设计者不能实现与这种能力同等的任何事情。[129]

## 7.7 资源共享（时分复用）

资源共享是一种优化技术，可以使用一个功能模块（如一个加法器或校验器）来实现几个操作。例如，乘法器可以首先用来处理A和B两个值，然后这个乘法器可以处理C和D另外两个值（第12章提供了资源共享的很好的例子）。

资源共享的另一个名称是时分复用。FPGA上的资源比ASIC上的资源更有限，因此，FPGA设计师更愿意在FPGA而不是ASIC的资源共享上花费更多精力。

在通信领域中，TDM指一种通过把数据流分成许多小块（每块都有非常短的持续时间）并使它们多路复用，来传输多数据流使它们组合为单一信号的方法。

相比而言，在资源共享的语言环境下，TDM是指通过在不同的时间复用器件（比如说乘法器）的输入，使不同的数据路径使用资源的一种共享资源的方法。

### 7.7.1 使用它或者放弃它

实际上，对于FPGA硬件有一个特殊的基本考虑，事情要比前面注释中的建议更微妙一些。这意味着FPGA只能是特定的尺寸，所以当你不能获得更小的尺寸时，你可能会去利用手头能获得的任何东西。

例如，你的设计要求使用两个嵌入式硬处理器核。除了这些处理器，你可能还想利用资源共享，那么，10个乘法器和2兆比特的RAM就已经够你用了。然而，如果包括两个处理器的FPGA提供了50个乘法器和10兆比特的RAM，你无法退回不用，所以你就会去充分利用这多余的能力。

### 7.7.2 其他内容

在FPGA的情况下，LUT/CLB从特殊元件像乘法器和MAC中得到数据通常比与其他LUT/CLB相连而得到数据代价更为高昂（在连通性方面），由于资源共享增

[130] 加了互连线的数量，你必须对这种情况保持警惕。

除了大的部件像乘法器和MAC之外，你也可以共用如加法器之类的东西。很有趣的是，在进位链技术中（比如Altera和Xilinx所使用过的），作为一种一维近似，建立一个加法器的代价很大程度上等价于建立一个数据总线的共享逻辑的代价。例如，实现两个具有完全独立输入和输出的加法器将要消耗两个加法器，而不需要资源共享多路复用器。但是如果你采用了资源共享，你将有一个加法器和两个多路复用器（每组输入有一个）。在FPGA条件下，这将增加复杂性而不是减少（在ASIC条件下，多路复用器的成本要比加法器的成本小得多，所以你要进行不同的考虑）。

在真实世界里，对于不同的工艺和情形，“使用它还是放弃它”与连通性代价之间的相互作用是不同的。也就是说，Altera的元件不同于Xilinx的元件等。

## 7.8 状态机编码

对于ASIC设计来说好的东西可能并不适合FPGA的实现，用于状态机的编码方案就是一个很好的例子。

如我们所知，FPGA的每个LUT都有一个触发器。这就意味着有数量可观的触发器闲置着。因此，许多情况下，“独热”编码方案对于基于FPGA的状态机是最好的选择，特别是当不同状态下的行为具有内在独立性的情况下。

“独热”编码是指在状态机中每一个状态都在触发器中有自己的状态变量，而且在每一时刻只有一个状态变量可以为高（“热”）。

## 7.9 测试方法学

ASIC设计者通常要花大量的时间使用工具来执行扫描链的插入和自动测试模式的产生（ATPC）。可能在他们的设计中也包括了执行内建自测试（BIST）。其中的大部分都是为了测试出器件的制造缺陷。相比之下，FPGA设计者一般不用考虑这种类型的器件测试，因为FPGA已经被制造商预先验证过了。

类似地，ASIC工程师通常会付出很多努力，他们要在设计中插入边界扫描（JTAG）并且验证它们。相比之下，FPGA在它们的结构中已经具有了边界扫描能力。

# 第8章 基于原理图的设计流程

## 8.1 往昔的时光

为了重现发展历程，我们就从20世纪60年代早期开始，回顾一下数字集成电路在那个年代的设计方式。这部分内容对于缺乏技术背景的读者来说是有趣的，对于仅熟悉当前设计工具和流程但是并不知道它们是如何演变的新手来说同样如此。此外，这些讨论是建立在一个潜在的框架上的，它将会有助于理解后面章节中更先进的设计流程。

在那些日子里，电子线路是手工处理的。电路图，也被称为原理图，是用钢笔、纸和模板（也许有人也会用桌布，当他恰好在餐厅里想到一个绝妙点子的时候）。这些图展现了用来实现设计的逻辑门和功能的符号，以及它们之间的连接。

通常，每个设计团队至少有一位成员擅长执行逻辑最小化和优化，也就是最终将一组逻辑门用另一组执行同样任务更快或消耗更少硅面积的逻辑门来代替。

检查设计是否会按计划实现它的逻辑功能——功能验证，通常是由一群工程师围坐在桌旁讨论出来，检查原理图并且不时地说，“嗯，这里看上去是正确的。”类似地，时序验证——检查设计满足要求的输入到输出和内部路径延迟，并且没有违例时间（比如建立时间和保持时间参数）引起内部寄存器出错——是用钢笔和纸完成的（如果你确实幸运，你也可能会使用了机械或电子计算器）。

最后，用手画出一组制图表示用来形成逻辑门的结构（或者，更准确地说，形成逻辑门的晶体管）和它们之间的互连线。这些由成群的多边形比如正方形和矩形构成的制图，最后用来制作光刻掩模，然后就可以制造真正的硅芯片。

在集成电路中，连接逻辑门的线可能是指线、轨或内部连接，以上所有这些词汇都可以表示内部互连线。

在某些情况下，敷金属也可以用来指这些走线，因为它们大部分是由集成电路的金属（敷金属）层形成的。

1865年，英格兰 James Clerk Maxwell 预言了电磁波的存在，指出它像光一样传播。

## 8.2 EDA初期

### 8.2.1 前端工具，如逻辑仿真

很显然，上面所说的手工设计方式要花费大量时间并且容易出错。大量的公司和大学把精力投入到各种各样的研究中。比如，在功能验证方面，到了20世纪60年代末至70年代初，已经有初步的逻辑仿真程序出现。

为了理解工作机制，设想我们有一个简单的门级设计，原理图已经手工绘在纸上（见图8-1）。

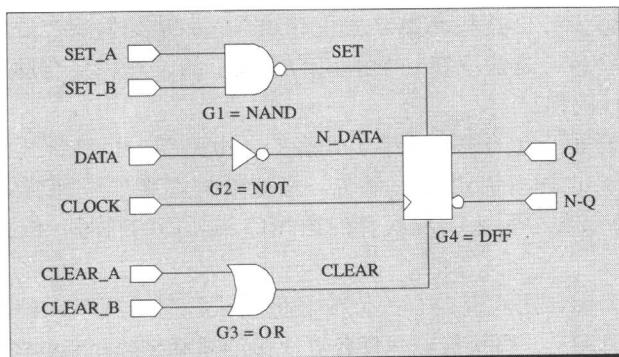


图8-1 简单的原理图 (画在纸上)

134

对我们来说，“门级”意味着设计是由基本逻辑门和功能的集合以及它们之间的连线表示的。为了使用逻辑仿真，工程师首先需要建立一个“门级网表”来表示电路图。在那个遥远的时代，工程师们一般要使用一个大型机，用一组打孔的卡片来记录网表。当计算机（带有存储设备如硬盘）变得更普遍时，网表开始以文本的形式存储（见图8-2）。

```
BEGIN CIRCUIT=TEST
INPUT SET_A, SET_B, DATA, CLOCK, CLEAR_A, CLEAR_B;
OUTPUT Q, N_Q;
WIRE SET, N_DATA, CLEAR;

GATE G1=NAND (IN1=SET_A, IN2=SET_B, OUT1=SET);
GATE G2=NOT (IN1=DATA, OUT1=N_DATA);
GATE G3=OR (IN1=CLEAR_A, IN2=CLEAR_B, OUT1=CLEAR);
GATE G4=DFF (IN1=SET, IN2=N_DATA, IN3=CLOCK,
              IN4=CLEAR, OUT1=Q, OUT2=N_Q);

END CIRCUIT=TEST;
```

图8-2 简单的门级网表

每个逻辑门都有相应的延迟，这些延迟（为了不复杂化我们在这里忽略了它们）可作为某些核的参数用于时序仿真（相关内容见第19章）。

注意，图8-2所示的格式是仅为这个例子而虚构的。受那个时代的局限，因为任何想要创造工具如逻辑仿真的人需要发明私人所有的网表语言。

对于原语门如AND、NAND、OR、NORD等，所有早期逻辑仿真工具都有内部的表示。这些表示被称为仿真原语。一些仿真工具还对更复杂个功能如D触发器有内部表示。这种情况下，图8-2中的G4=DFF可以直接映射成内部表示。

或者，人们可以产生一个DFF子电路，它的功能可以通过AND、NAND等原语的网表来表示。这时，图8-2中的G4=DFF功能可以看作仿真器实例化了这个子电路。

接下来，用户可以产生一组测试向量，即激励以逻辑1和逻辑0的形式作为电路的输入。这些测试向量天然地具有文本的形式，它们通常呈现为表格形式，如图8-3所示（在“；”后的任何字符都被视作注释）。

C C		
L L		
S S	C E E	
E E	D L A A	
T T	A O R R	
— T C —		
TIME	A B	A K A B
0	1 1	1 0 0 0 ; Set up initial values
500	1 1	1 1 0 0 ; Rising edge on clock (load 0)
1000	1 1	1 0 0 0 ; Falling edge on clock
1500	1 1	0 0 0 0 ; Set data to 0 (N_data = 1)
2000	1 1	0 1 0 0 ; Rising edge on clock (load 1)
2500	1 1	0 1 0 1 ; Clear_B goes active (load 0)
:		
etc.		

图8-3 一组简单的测试向量（文本形式）

1866, 爱尔兰/美国第1条永久性的跨大西洋电报电缆铺设完毕。

左边显示的是激励值的输入时间，输入信号的名称在上方空白处垂直地显示出来。

如图8-1和图8-2所示，在DATA输入和D型触发器之间有一个反相门(NOT)。这样，当DATA输入在0时刻呈现为1时，值1将被反相为0，接着在时钟上升沿到来的500时刻，值0载入寄存器。类似地，在1500时刻DATA输入为0，这个值先反相为1，然后在下一个时钟上升沿到来时的2000时刻载入寄存器。

在今天的术语中，图8-3所示的测试向量文件被视为一个初步的testbench。专

门规定的时间值再次作为某些仿真核单元的集成参数。

工程师接下来调用逻辑仿真器，读入门级网表并在计算机内存中构造一个电路的虚拟表示。仿真器将读入第1个测试向量（激励文件的第1行），将这些值送入适当的虚拟输入端口，它们的影响接着传递到整个电路。这个过程将被后面Testbench里的测试向量所重复（如图8-4）。

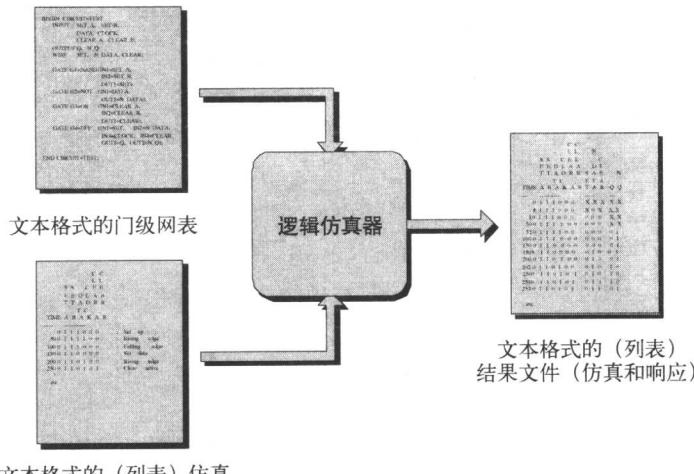


图8-4 运行逻辑仿真器

仿真器还需要一个或多个控制文件（或在线命令），来告诉它哪个内部节点（线网）和输出引脚需要监控、仿真进行多长时间等。最终结果与原始激励一起以表格的形式存入输出文件中。

假设我们开始运行一个早期的仿真器，电路如图8-1和图8-2所示，激励如图8-3所示。我们也假设非门（NOT）具有5个仿真单位时间的延迟，这意味着在这个门输入端的变化要花5个单位时间来传播并显示。类似地，我们也假设与非门和或门相应地有10个时间单位的延迟，而D型触发器具有20个时间单位的延迟。

这种情况下，如果命令仿真器监视所有的内部节点和输出引脚，输出文件将包括方针结果，看上去类似于如图8-5所示的形式。

为了讨论的方便，信号值的任何变化都在这个表中以黑体显示出来，但在现实世界中是不存在这种情况的。

在这个例子中，在0时刻初始值送入输入引脚。这时，所有的内部节点和输出引脚显示x值，表明它们处于未知状态。经过5个单位时间，初始值逻辑1通过DATA输入端，经过非门反相后在内部N\_DATA节点显示为逻辑0。类似地，送入输入端SET\_A和SET\_B的初始值经过异或门后到达内部的SET节点，而输入端CLEAR\_A和CLEAR\_B的初始值经过或门后到达内部节点CLEAR。

```

      C C
      L L     N
      S S   C E E   _ C
      E E D L A A   D L
      T T A O R R   S A E   N
      T C   _   E T A
      _ _ A B A K A B   T A R   Q Q
      -----
      0 1 1 1 0 0 0   X X X   X X ; Set up initial values
      5 1 1 1 0 0 0   X 0 X   X X
      10 1 1 1 0 0 0   0 0 0   X X

      500 1 1 1 1 0 0   0 0 0   X X ; Rising edge on clock
      520 1 1 1 1 0 0   0 0 0   0 1

      1000 1 1 1 0 0 0   0 0 0   0 1 ; Falling edge on clock

      1500 1 1 0 0 0 0   0 0 0   0 1 ; Set data to 0
      1505 1 1 0 0 0 0   0 1 0   0 1

      2000 1 1 0 1 0 0   0 1 0   0 1 ; Rising edge on clock
      2020 1 1 0 1 0 0   0 1 0   1 0

      2500 1 1 0 1 0 1   0 1 0   1 0 ; Clear_B goes active
      2510 1 1 0 1 0 1   0 1 1   1 0
      2530 1 1 0 1 0 1   0 1 1   0 1
      :
      etc.
  
```

图8-5 输出结果（文本文件）

在500时刻，CLOCK输入端的上升沿（从0变为1）引起D型触发器从N\_DATA节点载入逻辑值。20个单位时间以后结果显示在输出引脚Q和N\_Q上。仿真就这样进行下去。

输出文件的空白行，比如10时刻和500时刻之间的空白行，用来分隔出相关行为的组。例如，在0时刻设置初始值会引起5时刻和10时刻的信号变化。接着在500时刻CLOCK的输入转换引起在520时刻的信号变化。这两组的行为彼此是完全独立的，所以它们之间用空白行分隔开。

直到不久以前，工程师们还在处理着包括数千个门和内部节点的电路，在运行仿真时会包括几千步。（哦，我花了几个小时来审视这些文件，想要发现是否电路会像期望的那样工作；如果它不是期望的那样，就拼命地寻找问题！）

139

## 8.2.2 后端工具如版图设计

与用来辅助工程师定义IC集成电路（以及电路板）功能的逻辑仿真工具正相反，一些公司专注于开发用于IC版图设计的工具。在这里的上下文中，版图设计(layout)是指决定逻辑门如何在芯片的表面摆放（实际上是晶体管形式的逻辑门）以及在它们之间布线。

在20世纪70年代，像Calma、ComputerVision和Applicon这样的公司开发了专门的程序来帮助设计部门的人获得手绘设计的数字表示。这时，设计置于一个大

比例的图表中，然后用像鼠标一样的东西来把外形（多边形）的边界数字化，来定义晶体管和内部互连。这些数字文件接下来用于产生掩模，这些掩模就可以用于产生实际的硅片了。

后来，这些早期的计算机辅助制图工具发展成叫做多边形编辑器（polygon editors）的交互式程序，可以让使用者直接把这些多边形画在计算机屏幕上。这些工具最后发展到可以接受与驱动逻辑仿真器同样的网表，并自动地执行版图设计（布局和布线）任务。

### 8.2.3 CAE + CAD = EDA

像逻辑仿真这样用于设计流程前端（逻辑设计和功能验证）部分的工具最初都被划分为计算机辅助工程（CAE）软件。相比之下，像版图设计（布局布线）这样用于设计流程后端（物理）部分的工具起初被归类为计算机辅助设计（CAD）软件。  
[140]

在许多其他的工程学科里，CAD一词也用来指计算机辅助设计工具，比如机械和建筑设计。

因为以前的设计大部分都是基于CAE和CAD的，所以设计工程师（或简单地称为工程师）通常是指工作在设计流程前端的人；也就是完成如构思和描述（制图）一个集成电路（它做什么以及如何让它这样做）功能等任务的人。相比之下，版图设计者或设计者多指处于设计流程后端的人，也就是完成版图设计（决定逻辑门位置并用互连线把它们连接到一起）任务的人。

到了20世纪80年代，所有用于电子器件和系统设计的CAE和CAD工具被统一到电子设计自动化或EDA的旗帜下，这样对于每个人都很适用（除了那些不这样认为的人，但是没有人听到过他们的叹息和抱怨）。

## 8.3 简单的原理图驱动ASIC设计流程

到了20世纪70年代末和80年代初期，像Daisy、Mentor和Valid这样的公司开始提供图形化的原理图设计输入软件，允许工程师交互地创作电路（原理）图。使用一个鼠标，工程师能够在屏幕中从专门的符号库中选择出代表实体如IO引脚、逻辑门和功能的符号并布置它们。接着工程师可以在屏幕上使用鼠标来画线（互连线）把这些符号连接在一起。

当电路已经输入完成，原理图设计输入包可以根据命令产生相应的门级网表。这个网表首先可以用于驱动验证设计功能的逻辑仿真器。同样的网表还可以用来驱动布局布线软件（见图8-6）。  
[141]

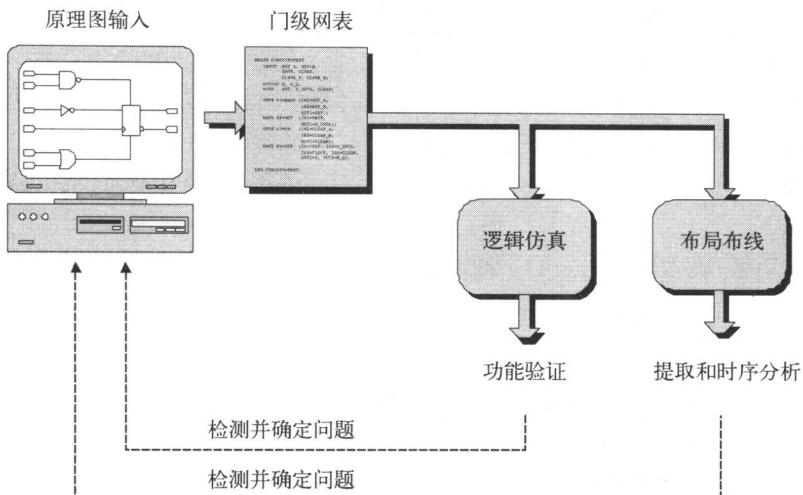


图8-6 简单的（早期）原理图驱动ASIC流程

最初由逻辑仿真使用的任何时序信息是估计的，特别是对于互连线来说，直到所有的布局布线完成以后才可能进行准确的时序分析。这样，在布局布线完成以后，将使用一个提取程序来计算与形成电路的结构（互连线段、过孔和晶体管等）有关的寄生电阻和电容值。接着时序分析程序将使用这些值，为器件产生时序分析报告。在有的流程里，为了执行更精确的仿真模拟，这些时序信息也反馈给逻辑仿真器。

一件重要事情是，在创造最初的原理图时，设计者从专门的设计库中得到逻辑门和功能的符号，这个专门设计库与ASIC目标工艺有关<sup>①</sup>。类似地，仿真器要按照指令使用与ASIC目标工艺相对应的具有适当逻辑功能性<sup>②</sup>和时序的仿真模型库。最终结果是布局布线软件直接把门级网表映射为逻辑门和功能在硅片上物理实现（这里与FPGA设计流程有些许不同，稍后将要讨论这方面的内容）。

142

## 8.4 简单（早期）的原理图驱动FPGA设计流程

当1984年第1批FPGA出现在人们视野中时，它们的设计很自然地基于已有的原理图驱动的ASIC设计流程。确实，因为流程的前面部分非常相似，用原理图设

<sup>①</sup> 总会有不同办法来解决这件事情。例如，有的流程基于使用包含所有ASIC单元库通用逻辑功能子集的通用符号库概念。由原理图设计输入应用产生的网表，接下来可以通过运行一个解释器转换为它们在ASIC目标工艺库中的等价单元名称。

<sup>②</sup> 关于逻辑功能性，人们可能会期望有简单的逻辑实体就像2输入与门，对于多个库来说功能仍是相等的。这在输入为“好的值”（逻辑1和逻辑0）时是确定的，但是当输入为高阻Z值或未知X值时，事情会有变化。甚至在输入为好的0或1值时，更复杂的功能如D型锁存器和触发器在“不正常情况”下，比如置位和清除输入同时被驱动时，表现会很不一样。

计输入包再次把电路描述为基本逻辑门和功能的集合，并产生相应的门级网表。像以前一样，这个网表接着用来驱动逻辑仿真器来完成功能验证。

差别开始于流程中的实现部分，因为FPGA的结构包括一个可配置逻辑块（CLB）的阵列，每个可配置逻辑块都是由大量的查找表LUT和寄存器组成的。这就要求在设计流程中引入额外的叫做映射（mapping）和包装（packing）的步骤（见图8-7）。

1875年 美国爱迪生发明了油印机。

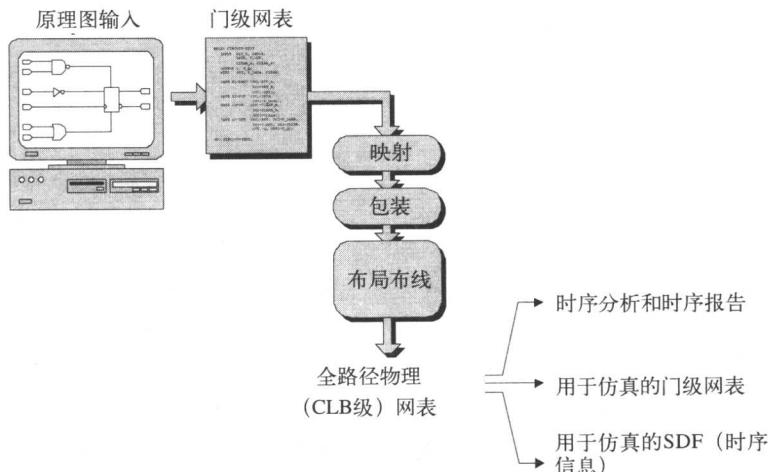


图8-7 简单（早期）的原理图驱动FPGA设计流程

#### 8.4.1 映射

在这里的上下文中，映射指将门级网表中有关的实体如门级功能块与FPGA中的LUT级功能块关联在一起的过程。当然，这不是一对一映射，因为每个查找表

144 LUT可以用来表示大量的逻辑门（见图8-8）。

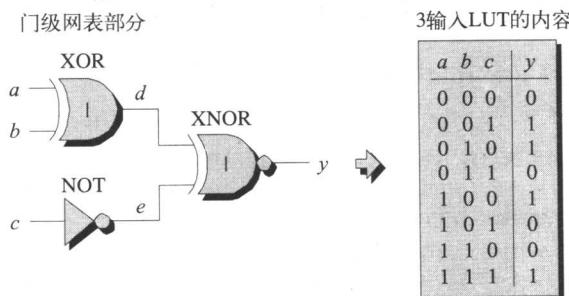


图8-8 逻辑门与LUT的映射

映射是一个不寻常的问题，因为把形成网表的逻辑门划分成小一些的组并映射到LUT里的办法会有很多。一个简单的例子，在图8-8中所示的非门功能可以从这个查找表LUT中删去，改为合并到上游驱动c线网的查找表中。

### 8.4.2 包装

紧跟着映射阶段，下一个阶段是包装，这时查找表LUT和寄存器被包装到可配置逻辑块CLB中。包装也不是一个简单问题，因为会有无数种排列和组合的可能。例如，一个非常简单的设计，它只包括一对少数的逻辑门，最后被映射到4个分别称为A、B、C和D的3输入LUT中。现在设想我们在处理一个FPGA，它的每个可配置逻辑块CLB包括两个3输入查找表LUT。这种情况下就需要2个CLB（称为1和2）来容纳这4个LUT。这样，我们的查找表LUT有 $4!$ （4的阶乘 $= 4 \times 3 \times 2 \times 1 = 24$ ）种不同的方式来包装到两个CLB中（见图8-9）。

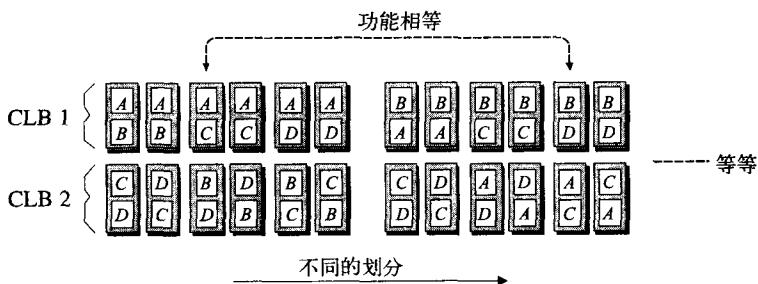


图8-9 包装LUT到CLB中

这里只显示24种不同方式中的12种（剩下的部分作为一个练习留给读者）。实际上，在现实中只有12种划分是有意义的，因为其中的每一个都有与它功能等价的“镜像”，比如AC-BD和BD-AC对。这是因为当我们开始布局布线时，这两个CLB的实际位置是可以互换的。

145

在FPGA出现以前，在CPLD中布局布线的等价功能是由叫做fitter的应用完成的。

当FPGA第一次出现在人们视野中时，人们使用同样的fitter应用程序，但是后来他们改为使用“布局布线”，因为这样更准确地反映了所发生的情况。

与使用基本逻辑门和寄存器的符号库正相反，大约在20世纪90年代早期发生了很有趣的变化，开始使用与稍微更复杂一些的逻辑功能（比如说大约70个）对应的符号库。原理图的输出是一个逻辑功能块的网表。这些逻辑功能块实际上已经映射到LUT并包装到CLB中。

这样就具有了更好数目的连接寄存器级的逻辑层次的优势，但是它受到比如优化和交换的限制。

### 8.4.3 布局和布线

紧随包装之后，我们开始布局和布线。对于前一点，设想我们的两个CLB需要连接到一起，然而——仅仅由于我们这部分讨论的需要——它们只能彼此水平或垂直地连接在一起，这种情况下有4种可能（见图8-10）。

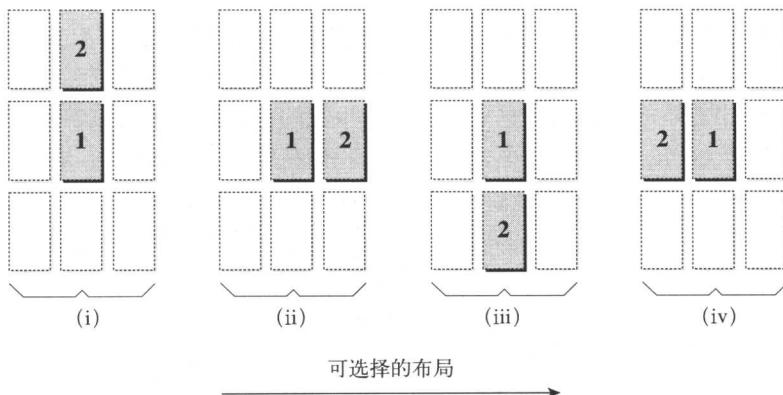


图8-10 CLB的布局

例如，在布局（i）的情况下，如果CLB1包括LUT A-C而CLB2包括LUT B-D，那么对调这两个CLB的位置与交换它们的内容是等价的。

如果我们只有如图8-10所示的2个CLB，将很容易决定它们的优化布局（也就是上面所示的4种选择中的一个）和这个2-CLB组相对于整个硅片的相对位置。

真实情况中的布局问题要复杂得多，因为一个真正的设计会包括极大量数量的CLB（在早期是成百上千，在2004年为几十万）。除了CLB1和CLB2要连接到一起，它们几乎确定地要与其他CLB相连。例如，CLB1可能也需要连到CLB3、5和8，而CLB2可能需要连接到CLB4、6、7和8。而且每个新的CLB可能还需要彼此连接或连接到别的CLB上。于是，尽管把CLB1和CLB2放在一起对它们来说是合适的，但是这样可能对与它们有关的其他CLB是有害的。实际上，最好的解决办法是把CLB1和CLB2分开一定的距离。

尽管布局是困难的，但在各种CLB之间决定信号的最佳路径更是一个拜占庭式（困难）的问题。这些任务的复杂性令人难以置信，所以我们把它留给那些写布局和布线算法的伙计们，然后迅速地转移到别的方面。

### 8.4.4 时序分析和布局布线后仿真

布局布线之后，我们有了一个全路径物理（CLB级）网表，如图8-7所示。这样，将用静态时序分析（STA）计算所有的输入-输出和内部路径延迟，并检查任

何与内部寄存器有关的时序违例（建立时间、保持时间等）。

有趣的是，当设计工程师想要用准确的时序信息重新仿真设计的时候，他们必须使用适合产生一种新的叫做（或许并不奇怪）标准延时格式（SDF）的工业标准文件格式形式的门级网表的FPGA工具，产生的这种网表带有相关时序信息。当原始网表转换为它的CLB级等价形式时，产生这种新门级网表的主要原因是，在新的表达中才有的时序显然不可能在原始的门级网表体现出来。

## 8.5 平坦的原理图与分层次的原理图

### 8.5.1 沉闷的扁平原理图

起初的原理图设计输入包允许将输入设计成很大的、扁平的并分割成很多“页”电路图。为了更加形象地说明这个问题，假设把一个包括1000个逻辑门的电路图画在一张纸上。那么，最终得到的是一张很大的图纸，电路的基本输入都在左边，电路的基本输出置于右边，中间是电路主体。

很显然，携带这张电路图并把它展示给你的朋友是一件苦差。于是，你就会想把它裁减成很多页，并把它们都放到一个文件夹里。这时，你需要确定你的划分是合乎逻辑的，这些页包括了与设计详细功能相关联的所有逻辑门。同样，你要使用连接器（一种虚拟的输入输出接口）把不同的页之间的信号联接起来。

这就是早期原理图设计输入包的工作方式。这样可以设计出一个扁平的、由页间的连接器符号连接起的一系列“页”构成的电路原理图，通过给这些符号命名来告诉系统应当把哪些页联在一起。例如，考虑一个画在纸上的简单电路（见图8-11）。

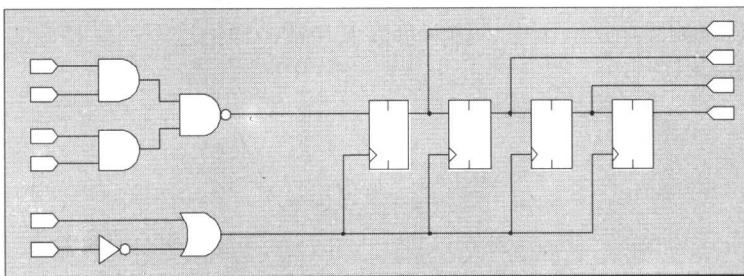


图8-11 画在一张纸上的简单电路图

假设左边的门代表一些控制逻辑，而右边的四个寄存器实现了一个4比特移位寄存器。这显然是一个很简单的设计，实际的电路会具有多得多的逻辑门。我们只是想在这里加深一下基本概念，比如，在你把这个电路输入到计算机系统时，可能会把它分为两页（见图8-12）。

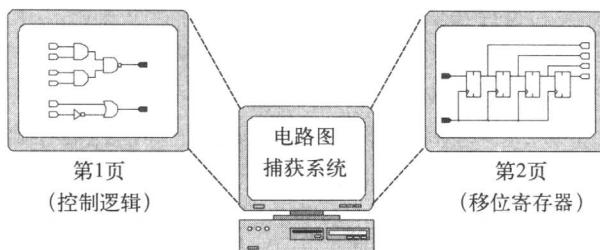


图8-12 简单的两页电路图

### 8.5.2 分等级（基于模块）的原理图

上面讨论的扁平原理图有大量的问题，特别是当处理的实际电路要求50或更多页数时。

- 很难高层次地、自顶向下地直观去审视这个设计。
- 设计中的一部分很难保存并在未来的项目中重用。
- 当设计中电路的某些部分多次重复使用时（这是很常见的），这部分将不得不重画或复制到很多页中去。如果你接下来要对它做一些改变时将非常痛苦，因为你必须对所有的副本做同样的改变。

解决办法就是把电路图设计输入包增强为支持层次的概念。例如，对于移位寄存器电路，可以用一个顶层页作为开始，其中包括两个叫做控制和移位的模块，每一个模块都需要一定数目的输入输出引脚。接着，把这两个模块彼此连接到一起，对基本输入输出也同样如此。

接下来，将命令系统输入控制模块，这将打开一个电路原理图页。如果幸运的话，系统将自动地把与父模块引脚相应的输入和输出连接器符号（以及相应的名称）预先放置在这一页上。接着可以像往常一样把这个模块相应的原理图画出来（见图8-13）。

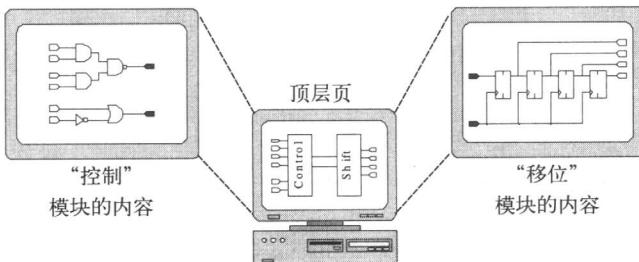


图8-13 简单的层次原理图

实际上，每个模块能够进一步包括模块级原理图，或门级原理图，或（非常

多见) 二者的混合。这些基于模块的层次原理图解决了扁平原理图产生的这些问题。

150

- 很容易直观的从高层次、自顶向下地观察并完成整个设计。
- 在未来的项目中保存并重用设计的一部分变得更容易了。
- 在设计的某些部分要重复使用好多次时, 仅仅需要创建这部分——作为一个分立的模块——一次并接着多次实例化(调用)这个模块。如果你意识到必须修改了, 那也很简单, 因为你只需要修改初始模块的内容。

## 8.6 今天的原理图驱动设计流程

通常所有的电路原理图、映射、包装和布局布线应用软件都是由FPGA公司自己开发和拥有的。然而, 普遍认为一个公司能够擅长开发EDA工具, 或者擅长制造硅芯片, 但是很难二者兼备。

问题的另一方面是, 原先ASIC世界的设计工具极端昂贵(甚至像现在认为最常用的原理图输入这样的工具)。相比之下, FPGA制造商集中于出售芯片, 所以他们一直以非常低的成本提供工具(实际上, 如果你是一个需求量足够大的顾客, 他们将会把整个设计工具免费给你)。虽然这对最终用户有明显的吸引力, 然而后来的趋势是, FPGA制造商不再热心地花大量金钱来提升没有给他们带来多少回报的设计工具。

因此, 随着时间的流逝, 外部的EDA开发商开始帮助解决这个难题, 开始是原理图设计输入, 接着是映射和包装(经由逻辑综合, 如第9章和第19章讨论的)。FPGA制造商通常仍然提供内部开发的、相对简单版本(与那些艺术级的相比)的设计工具。比如, 原理图设计输入作为他们的基本工具组合的一部分, 并且他们仍然维持布局布线工具作为他们的掌上明珠。

151

使用原理图设计工具在门级抽象层次上作开发, 对于今天的工程师来说, 已经是遥远的记忆了。有的情况下, 为了支持包含最近的前一代元件的原理图库的器件, FPGA制造商还提供一些这种流程类型的支持。但是, 仍然在一些老工程师和那些需要对以前遗留的设计做小修改的人们那里, 原理图设计输入工具仍有用武之地。除此以外, 图形输入机制正在从那些在现代设计流程中找到位置的输入包中退出来, 就如第9章所讨论的那样。

152

# 第9章 基于HDL的设计流程

## 9.1 基于原理图流程的问题

到了20世纪80年代末，由于设计规模和复杂性的增加，基于原理图的设计流程逐渐开始退出历史舞台。在门级的抽象层次上观察、输入、调试、理解和维护一个设计变得非常困难，而且在处理规模在5000门以上的电路时这种方法几乎失效。

在门级抽象层次上设计并输入一个大型设计除了容易出错以外，还非常耗费时间。这样，一些EDA提供商开始在使用HDL（硬件描述语言）的基础上开发新的设计工具和流程。

## 9.2 基于HDL设计流程的出现

HDL背后的思想，或许并不令人惊讶，就是你能够用它来描述硬件。广义地说，“硬件”一词用来指电子系统中的任何物理部分，包括集成电路、印制电路板、机箱和电缆，甚至还有把系统固定在一起的螺钉和螺帽。然而在HDL的概念范围内，“硬件”仅仅是指集成电路的电子部分（元件和连线）和印制电路板。（HDL也可以用来提供有限的电缆和连接器件的表示，用来将电路板连在一起。）

在电子设计的早期，几乎所有开发EDA工具的人都需要开发自己的HDL。153这些语言中有一些是模拟HDL，因为有人想要表达模拟领域的电路，而其他的则专注于表达数字功能。出于本书的需要，我们只对设计ASIC和FPGA形式的数字IC中的HDL感兴趣。

### 9.2.1 不同的抽象层次

稍后本章将介绍一些最常见的HDL。现在，让我们先看看HDL怎样在设计流程中使用。要注意的第一件事就是数字电路的功能可以在不同的抽象层次上表达，而且不同的HDL支持这些抽象层次的程度是不同的（图9-1）。

数字HDL的最低的抽象层次是开关级，也就是能够将电路描述为一个由晶体管开关组成的网表。稍高一些的抽象层次是门级，是指将电路描述为基本逻辑门和功能的组成的网表。这样，前面章节中所讨论的由原理图设计输入包产生的早期门级网表格式实际上就是初步的HDL。154

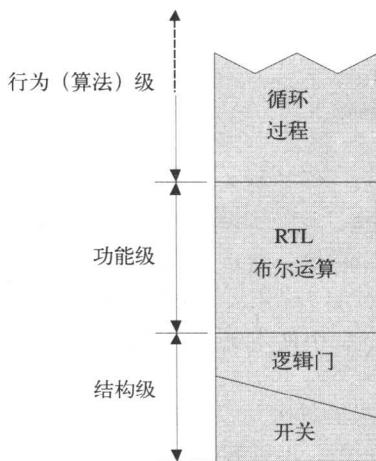


图9-1 不同的抽象层次

开关级和门级网表都可以被归为结构化表达的一类。然而应当注意，结构可能具有不同的内涵，因为它还可以表示层次的模块级网表，每个模块可能包括使用如图9-1所示的任何抽象层次来说明的内容。

HDL的下一层级具有支持功能性表达的能力，覆盖了大量的结构。其中低端的部分是能够使用布尔等式描述一个功能。例如，假设我们已经声明了一组叫做Y、SELECT、DATA\_A和DATA\_B的信号，我们可以使用下面的布尔等式表示简单的2 : 1多路复用器：

```
Y = ( SELECT & DATA_A ) | ( ! SELECT & DATA_B );
```

注意，这里是为这个例子使用的普通语法，不局限于任何特殊的HDL（像我们在第3章中讨论的那样，符号“&”表示逻辑与，符号“|”表示逻辑或，而符号“!”表示逻辑非）。

George Boole几乎完全通过自学在数学的很多领域都做出了重大贡献。他于1847年和1854年完成的两项著作名垂史册，在这两部著作中，他用数学的形式表示逻辑表达式，也即现在所说的布尔代数。

1938年，Claude Shannon发表了一篇基于他在MIT攻读硕士学位期间研究课题的论文，说明了布尔代数的概念能如何用于表达电子电路中的开关功能。

抽象的功能层次还包括寄存器传输级（RTL）表达。RTL涵义丰富，然而对其基本概念最简单的理解就是，设计是用组合逻辑链接起来的寄存器集合形成的。这些寄存器通常是由公共时钟信号控制，所以假设已经声明了2个叫做CLOCK和CONTROL的信号，还有一组叫做REGA、REGB、REGC、REGD的寄存器，那么可能一个RTL型语句某种程度上就像这样：

```

when CLOCK rises
  if CONTROL == "1"
    then REGA = REGB & REGC ;
    else REGA = REGB | REGD ;
  end if ;
end when ;

```

这种情况下，`when`、`rises`、`if`、`then`、`else`以及类似的信号都是关键字，语义由HDL的所有者来定义。同样地，这是普通的语法，并不局限于哪一种HDL。

传统的HDL最高的抽象层次是行为级，即能够使用抽象结构象循环和过程来描述电路行为。这里也包括在等式中使用比如加法器和乘法器的算法单元。比如：

```
Y = ( DATA_A + DATA_B ) * DATA_C ;
```

注意，这里还有一个系统级的抽象层次（图9-1中没有显示出来），其属性构造是为了用于系统级设计应用，稍后再讲解这一层次。

许多早期的数字HDL仅支持开关或门级网表的结构化表达。其他的比如ABEL、CUPL和PASLSM主要是用来满足PLD器件的设计输入需要。这些语言（第3章介绍过的）支持不同的逻辑抽象层次，比如布尔表达式、真值表和基于文本的状态机（FSM）描述。

下一代的HDL，主要目标是为了逻辑仿真、支持更复杂级别的抽象如RTL和一些行为级结构。就是这些HDL构成了下面所讨论的最初的基于HDL设计流程。

### 9.2.2 早期基于HDL的ASIC设计流程

以HDL为基础的ASIC设计流程的关键特征是，它们使用了逻辑综合技术，这种技术是1980年代中期开始出现于市场上的。这些工具能够接受一个带有时序约束的RTL表达设计。这种情况下，时序约束表达为包括很多行语句的附加文件，就像这样“从输入X到输出Y的最大延迟应当不超过N纳秒”（实际格式会稍简洁而且更加枯燥一些）。

应用逻辑综合可以自动地把RTL表达转换为寄存器和布尔等式的混合体，执行一系列最小化和优化（包括面积优化和时序优化），接着产生一个将（或至少、应该）满足原始时序约束的门级网表（见图9-2）。

这种新型的流程具有很多优势。首先，设计工程师的生产率显著提高，因为与处理大张的门级电路原理图正相反，在RTL抽象层次上描述、理解、讨论和调试设计必需的功能要容易得多。而且逻辑仿真器运行RTL描述的设计要比相应的门级设计快得多。

有个小问题是，逻辑仿真器能够对包括行为结构的高抽象层次所指定的设计起作用，但是早期的综合工具只能接受最高到RTL层次的功能表达。这样，设计工程师不得不使用他们选择的HDL的可综合子集。

156

157

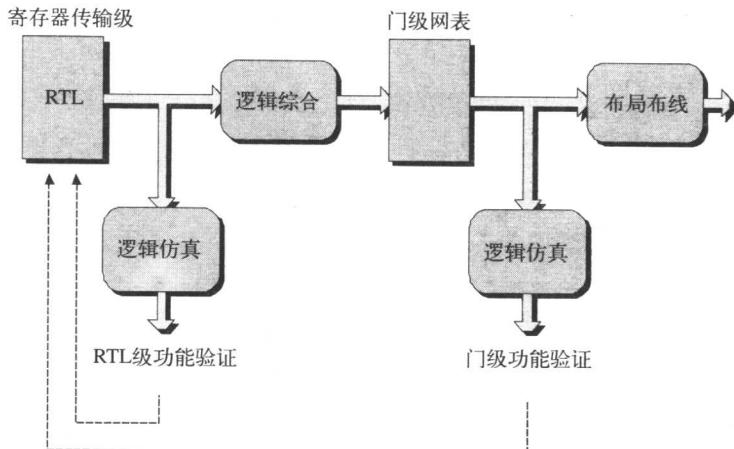


图9-2 简单的基于HDL的ASIC设计流程

一旦综合工具产生了门级网表，流程就开始变得和前面所讨论过的基于原理图的ASIC设计流程很相似。门级网表可以通过仿真来确保功能的正确性，也可以在线路和其他电路元件估计值的基础上用于时序分析。网表可以驱动布局和布线软件，提取出的线路电阻和电容值可用来进行更精确的时序分析。

### 9.2.3 早期基于HDL的FPGA设计流程

经过一段时间以后，基于HDL的流程才在ASIC世界繁荣起来。其间，设计师开始抓住FPGA的概念。这样，到了1990年代的早期，在FPGA世界中完全可以获得基于HDL流程特征的逻辑综合技术（图9-3）。

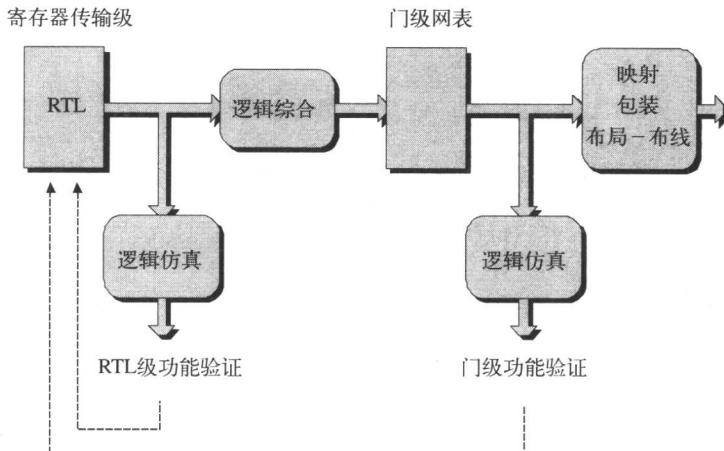


图9-3 简单的基于HDL的FPGA设计流程

158

如以前一样，一旦综合工具产生了门级网表，流程变得与前面章节讨论的基于原理图的FPGA设计流程很相似。门级网表可以用来仿真以确保功能正确性，也可以在线路和其他电路元件估计值的基础上执行时序分析。网表可以用来驱动FPGA的映射、包装和布局－布线软件，随之使用真实世界（物理）的值产生更精确的时序报告。

### 9.2.4 知道结构的FPGA流程

最初困扰基于HDL的FPGA流程的主要问题是，它们的逻辑综合技术是来自ASIC世界。这样，这些工具按照逻辑门和寄存器原语来“思考”。于是，这意味着输出门级网表，并把它留给FPGA制造商来执行映射、包装和布局布线功能。

159

到了1994年的某个时候，综合工具拥有了关于不同FPGA结构方面的知识。这意味着它们内部能够执行映射——和某种程度的包装——功能并且输出LUT/CLB级别的网表。这个网表随后传给FPGA制造商的布局－布线软件。这种方法的主要优势在于综合工具在时序估计和面积优化方面具有更好的想法，这使它可以产生更高质量的结果（QoR）。实际上，通过知道结构综合工具产生的FPGA设计要比传统的门级提供的结果快15%~20%。

### 9.2.5 逻辑综合与基于物理的综合

起初设计逻辑综合工具是用于1980年代中期的几微米级ASIC技术。在这些器件中，与逻辑门有关的延迟明显要比与线路有关的延迟的多。除了在门数相对较少之外（以今天的标准看），这些设计还具有相对小的时钟频率和相应地宽松设计约束。所有这些因素组合到一起，意味着早期的逻辑综合工具能够采用相对简单的算法来估计这些线路延迟，而这些估计值也将与器件的真实（布局布线后）值非常接近。

很多年来，ASIC设计的规模（门数）和复杂性不断增加。与此同时，硅片结构的维数收缩为两个重要结果。

- 通常延迟效应变得更为复杂。
- 与线路有关的延迟变得比与门有关的延迟更为明显。

160

到了1990年代中期，ASIC设计变得规模十分巨大，而且它们的延迟也显著地复杂了，大大超过以前逻辑综合工具曾经设计过的规模。结果就是，使用逻辑综合工具估计的延迟与最终的布局布线后延迟几乎没有任何关系。于是，这意味着达到时序收敛（把设计“拧”得与原设计目标相同）变得越来越困难而且很费时。

由于这个原因，大约到了1996年ASIC流程开始使用基于物理的综合。我们会在第19章给出这种考虑物理情况下的更多综合细节。目前，我们只需要注意到，在完成规划的过程中，物理综合引擎决定了逻辑门和功能的初始布局。基于这些

布局，工具能够产生更精确的时序估计。

终于，这种基于物理的综合输出了一个布局（然而还没有布线）后的门级网表。ASIC物理实现（布局和布线）工具是用这个初始布局信息作为起点，来执行局部的（细粒的）根据布线细节而作的布局优化。最终结果就是，基于物理的综合应用所使用的延迟估计非常接近于布局布线后延迟。于是，这意味着达到时序收敛时的负担减轻了很多。

“那么FPGA是怎么样做的，”你快要叫出来了。好，接着往下看。这些器件在整个1990年代时规模和复杂性也不断增加。到了2000年时，FPGA在时序收敛方面遇到了严重的问题。此时，EDA开发商开始提供FPGA为中心基于物理的综合工具，能够输出一个映射、包装和布局后的LUT/CLB级网表。这种情况下，FPGA的物理实现（布局和布线）工具是用这些初始布局信息作为起点，执行局部的（细粒的）根据布线细节而作的布局优化。

### 9.3 图形设计输入的生活

先说一段题外话，当第1个基于HDL的流程出现时，许多人设想图形设计输入和可视化的工具，比如原理图设计输入系统，将要准备永远离开舞台了。确实，有的时候，许多设计师为自己使用文本编辑器如VI（来自图形接口）或EMACS作为他们唯一的设计输入手段而感到自豪。

在一个专家的手中，VI编辑器曾是（现在也是）最有力的工具，但是它对于新手来说太困难了。

161

然而，人们常说一图胜千言，而且图形输入技术在多种级别中都是最流行的。例如，使用模块级原理图编辑器进行输入使设计作为高层次模块连接在一起的集合是非常普遍的。接着系统会自动创建HDL结构的框架，包括所有的模块名和输入以及输出的声明。或者，使用者还可以用HDL创建一个框架，系统用它来自动创建模块级原理图。

从使用者的视角来看，进入其中任何一个原理图模块都会自动打开HDL编辑器。它可以是一个像VI那样的基于纯文本－命令的编辑器，也可以是更复杂的具有用不同颜色显示关键字、自动完成声明等功能的HDL专用编辑器。

此外，当进入一个原理图模块时，现代设计系统经常要求你做选择，比如输入和作为其他人看这个模块的内容、低层次模块级原理图、纯HDL代码、图形化状态图（通常表示一个FSM）、图形化流程图，等等。对于这些图形表示，如状态图和流程图，随后可以自动产生与它们等价的RTL表示（见图9-4）。

此外，通常它还会具有一个表格式的文件，包括与器件外部输入和输出有关的信息。这种情况下，顶层模块图和表格文件将（有希望地）直接链接到相同的

数据，并将只提供这些数据的不同视角。在任何视角做出改变都将更新核心数据，  
 [162] 并在所有视角中直接反映出来。

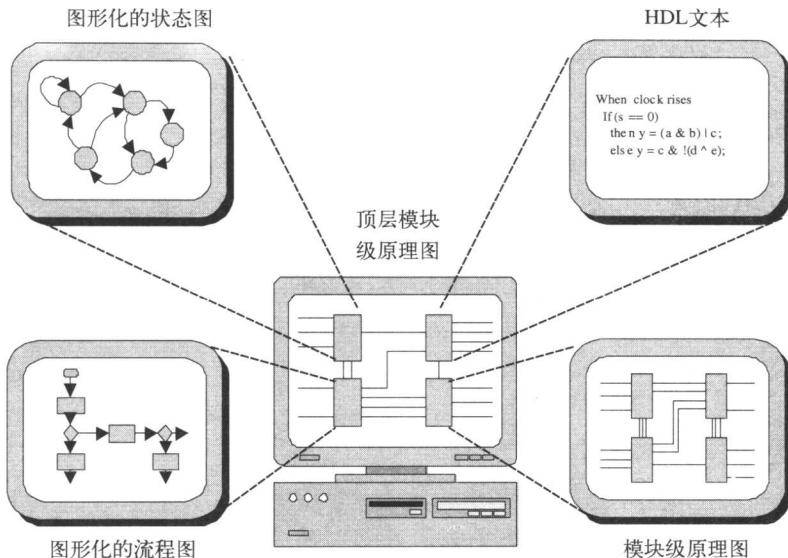


图9-4 多级别混合设计输入环境

## 9.4 绝对过剩的HDL

如果只有一种HDL，生活将会简单得多，然而生活没有这么悠闲。前面曾经提到，在数字IC电子设计的早期（大约在20世纪70年代），任何开发基于HDL设计工具的人都需要创建他们自己的语言来配合它。不出所料，结果是一片混乱。人们所需要的只是一个可以由多种EDA工具和开发商使用的HDL工业标准，但是这样的宝贝到哪里去找呢？

### 9.4.1 Verilog HDL

到了20世纪80年代中期的某个时候，Phil Mooby（创建过著名的HILO逻辑仿真器的小组的初始成员之一）设计了一种新的叫做Verilog的硬件描述语言。1985年，他所在的公司Gateway Design Automation，把这种语言和相应的Verilog-XL逻辑仿真器一起投入市场。  
 [163]

有个随同Verilog和Verilog-XL一起的概念很酷，这被叫做Verilog编程语言接口（PLI）。这种技术的更普通的名字是应用编程接口（API）。API是一个软件函数库，允许外部软件向一个应用传送数据并且从这个应用中访问数据。这样，Verilog

PLI就是一个允许用户扩展Verilog语言和仿真器的API。

看一个简单的例子，假设一个工程师在设计一个电路，利用了外部现有模块来完成一个如FFT算法的功能。这个现有功能的Verilog表达可能要花很长的时间来仿真，如果工程师真正想要验证的是电路的新部分，那么这将是很痛苦的。这时，工程师可以用C语言来创建这种功能的一个模型，用于验证时要比原来的Verilog模型快1000倍。这个模型将合并到PLI结构中，使它链接到方针环境中。随后剩余部分电路的Verilog描述将通过PLI调用访问这个模型，PLI调用提供双向链接可以在主要电路（用Verilog描述）和FFT（使用C语言设计）之间来回传送数据。

然而Verilog和Verilog-XL的一个更加有用的特性是，能够将时序信息详细列于一个叫做标准延迟格式（SDF）的外部文本文件。这就允许像布局布线后时序分析包这样的工具产生SDF文件，供仿真器使用以产生更精确的结果。

作为一种语言，最初的Verilog在结构（开关和逻辑门）级抽象（尤其是延迟建模能力）上相当强大，它在功能（布尔表达式和RTL）级抽象非常强大，而且它支持一些行为（算法）级结构（见图9-5）。

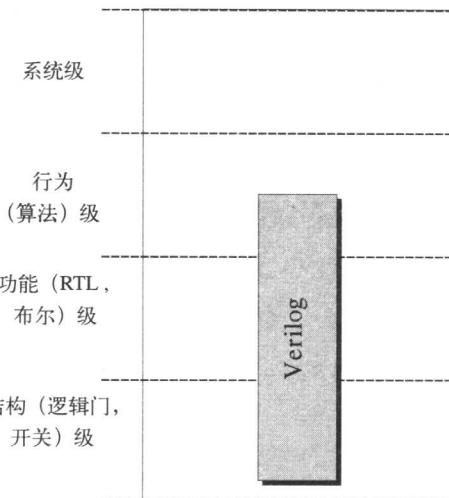


图9-5 抽象级别 (Verilog)

164

到了1989年，Gateway Design Automation公司连同Verilog（HDL）和Verilog-XL（仿真器）一起并入Cadence Design System公司。在那时，最有可能发生的情景是Verilog继续作为一种私有的HDL。然而，几乎令整个产业界惊讶的是，Cadence在1990年把Verilog HDL、Verilog-XL和Verilog SDF全部公开。

这是一个非常大胆的举动，因为任何人都可以开发一种Verilog仿真器，这样可能会成为Cadence潜在的竞争者。Cadence主动馈赠的原因是VHDL语言（这个内容稍后将会介绍）开始赢得可观的追随者。把Verilog投入公共领域后会使更大范围内的开发基于HDL设计工具，比如逻辑综合应用的公司感到用Verilog作为他们选择的语言更为合适。

仿真、综合和其他工具能够使用同一种设计表达使每个人都倍感方便。然而还有重要的一点，Verilog起初是为仿真而专门设计的。这意味着在创建一个用于仿真和综合的Verilog描述时，人们应该严格使用语言的可综合子集（宽松定义并使各种逻辑综合包能够理解和支持的语言结构集合）。

165

Verilog的严格定义压缩在一个文件中，也就是语言参考手册（LRM），包括语言的语法和语义的细节。在这里的上下文中，语法指的是语言的文法，比如词语的顺序和彼此涉及的符号，而语义指的是词语和符号的根本含义和它们所表示事物之间的联系。

在理想世界里，LRM的定义很严格以至于不会产生任何歧义。然而在真实世界中，关于Verilog LRM有很多含糊的地方。诚然，像“如果一个寄存器的控制信号在时钟信号触发的同时达到稳定，仿真器将首先计算哪个值”这样的语句会伴随着附加条件。然而这样将导致不同的Verilog仿真器可能产生不同的结果，对最终用户来说始终是不太妥当。

Verilog很快变得非常流行，问题是不同的公司开始在不同的方向上扩展这个语言。针对这种情况，1991年建立了一个叫做国际开放Verilog（OVI）的非盈利组织，代表当时所有主要的EDA厂商来管理Verilog HDL和Verilog PLI。

Verilog继续以指数增长的速度流行着，最终OVI要求IEEE设立一个工作组来把Verilog制定成IEEE标准。这样，众所周知的IEEE1364工作组于1993年建立。1995年5月，第1个官方的IEEE Verilog发布，正式名称为IEEE1364-1995，非正式名称为Verilog95。

2001年对这个标准进行了较小的修改。因此，这个新版本被称为Verilog2001或Verilog2K1。在写作本书的时候，IEEE1364工作组正在为即将来临的Verilog2005而工作着，而整个设计界都在屏住呼吸期待着（在本章后文中将看到“Superlog和System-Verilog”）。

#### 9.4.2 VHDL和VITAL

在1980年，美国国防部（DoD）发起了甚高速集成电路（VHSIC）计划，基本目标是提升数字集成电路技术的开发状态。

这个计划在寻求一种其他的方式来开发电路。当时的情况是因为零件功能没有以严格的方式形成文件，集成电路（和电路板）在军事装备的长生命周期内重用非常困难。此外，一个系统的不同部件经常是使用不同的和不兼容的仿真语言和设计工具进行设计和验证的。

为了解决这些问题，1981年发起了开发一种叫做VHSIC HDL（或简称为VHDL）的HDL的计划。这个计划的独特性之一是，产业界在早期就参与进来。在1983年，包括Intermetrics、IBM和Texas Instruments组成的团队获得了发展VHDL的合同，1985年发布了第1个官方版本。

有趣的是，为了使产业界接受VHDL，美国国防部随后在1986年将定义VHDL语言的所有权利赠给IEEE。对一些已知的问题作了修正后，1987年VHDL作为IEEE 1076正式标准发布。先在1993年然后又在1999年，语言作了更多的扩展并公

诸于众。

作为一种语言，VHDL在功能级（布尔表达式和RTL）和行为级（算法）抽象方面十分强大，而且它也支持一些系统级设计结构。然而，当VHDL到了结构级（开关和逻辑门）抽象时，它的功能就不那么强了，特别是在建立延迟模型能力方面。

起初，VHDL没有可以与Verilog的PLI相对应的部分。今天，不同的仿真器按它们自己的方式来做这部分事，比如Modelsim的外部语言接口（FLI）。我们希望它们最终统一到一个通用标准下面。

VHDL用于时序标注后仿真精度的不足很快突显起来。由于这个原因，1992年设计自动化讨论会（DAC）开始发起VITAL。面向ASIC库的VHDL（VITAL）是为了提升VHDL在ASIC和FPGA设计环境中时序建模能力而做出的一个努力。最终结果既包括一个ASIC/FPGA功能原语的库，也包括一个将延迟信息反标注到这些库模型的相关方法，而这个延迟机制的基础与Verilog所使用的基本列表格式相同（见图9-6）。

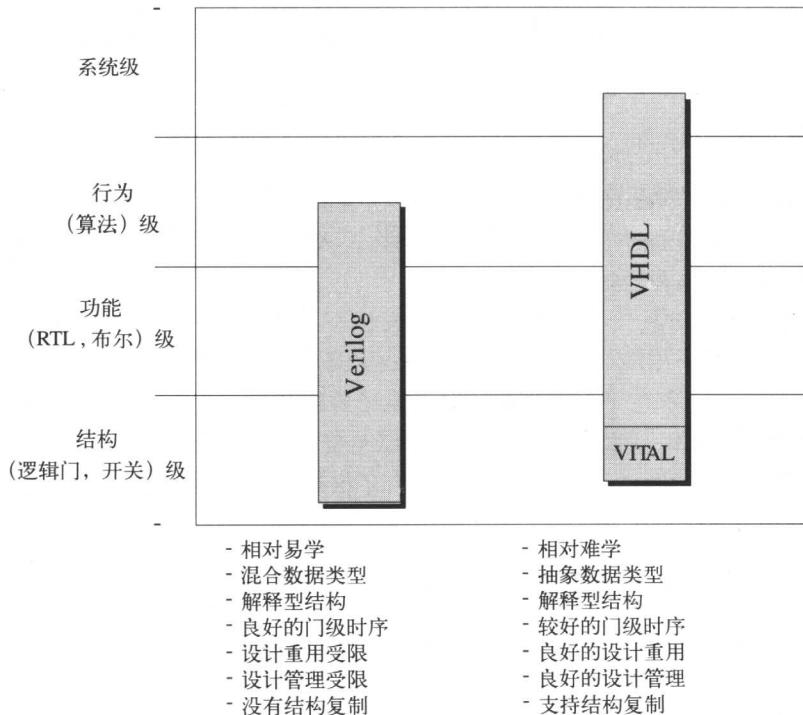


图9-6 抽象级别（Verilog 与 VHDL）

### 9.4.3 混合语言设计

从前，整个设计的输入通常是使用同一种HDL（Verilog或VHDL）。然而，随着设计规模和复杂度的增加，设计的不同部分由不同的团队来创建变得很普遍。这些团队可能来自不同的公司甚至居住在不同的国家，而且不同的队伍使用不同的设计语言是毫不奇怪的。

另一个考虑是，为了更多的使用已有设计模块和第三方IP，后者是指一个设计团队从外部供应者那里获得预定义的功能。根据墨菲法则，如果你正在使用一种语言，那么你只能获得其他语言的IP。

**墨菲法则：**如果事情可能出错，那么它就一定会出错。这个法则要归功于一个叫Edward Murphy的上尉，他是1949年在爱德华兹空军基地工作的一名工程师。

在1990年代早期有一个阶段叫做HDL战争，一种语言（Verilog或HDL）的支持者尖锐地预言另一种语言将会消失……然而很多年过去了两种语言都拥有了众多的追随者。最终结果是，EDA提供商开始支持包括逻辑仿真、逻辑综合应用和其他工具在内的混合设计环境，能够使用Verilog和VHDL模块混合来组成设计。

### 9.4.4 UDL/I

前面曾经提到，Verilog起初是设计用于仿真的。类似地，VHDL是作为考虑了仿真的文件和设计规范语言而创建的。那么，人们可以使用这些语言来描述可用于仿真但不能被综合的结构。

为了解决这个问题，日本电子工业发展协会（JEIDA）开发了他们自己的  
169 HDL，也就是1990年出现的集成电路的统一设计语言（UDL/I）。

UDL/I的主要优点在于，它是在同时考虑了方针和综合基础上开发的。UDL/I环境包括一个仿真和一个综合工具，而且可以免费获得（包括源码）。然而，到了UDL/I出现的时候，Verilog和VHDL已经具有了很雄厚的基础，而UDL/I在日本以外也从未获得过真正的关注。

### 9.4.5 Superlog 和 SystemVerilog

1997年，当一个叫做Co-Design Automation的公司建立以后，事情开始变得复杂起来。经过废寝忘食的工作后，Co-Design的工程师们开发出叫做Superlog的“Verilog类固醇”。

Superlog令人非常吃惊，它把Verilog的简练与C语言的强大能力结合了起来。

它也包括了时间逻辑、复杂设计验证能力、动态API和对于模型检查的形式验证策略很关键的断言概念（VHDL已经有了简单的断言结构，而最初的Verilog在这方面没有任何可以自夸的东西）。

Superlog的两个主要问题是，它本质上是另一种私有语言，而且它比Verilog95复杂得多，以至于其他的EDA提供商把他们的工具提升到支持它，这简直是壮举。

当每个人都对未来的方向感到疑惑时，OVI组织与名为VHDL国际的组织开始联合建立一个新的叫做Accellera的团体。这个新组织的任务是专注于检验新的标准和格式，发展这些标准和格式，并培养基于这些标准和格式的新技术。

在2002年的夏天，Accellera发布了叫做SystemVerilog3.0（先忽略关于1.0和2.0的事情）的混合语言规范。这种语言的最大优点在于它是已有Verilog基础上的增强，而不是跳跃性地完全用Superlog来取代。实际上，SystemVerilog具有很多由Co-Design赠送的Superlog的语言结构。包括了断言和每个人都需要的扩展后的综合能力，而且作为一种Accellera标准，它很快被广泛接受。170

1880年，法国 Pierre 和 Jacques Currie发现了压电现象。

目前（写作本书的时候）的状态是，在2002年秋季Co-Design被Synopsys获得，Synopsys维持了从Superlog赠送给SystemVerilog语言结构的政策，然而没有人继续把Superlog作为一种独立的语言了。经过一段时间的磨合之后，所有主流EDA厂商正式认可了SystemVerilog，并给他们的工具增加了功能，根据专门应用领域和要求来接受这个语言的不同子集。SystemVerilog3.1于2003年夏天面世，接着大约在2004年初又发布了3.1a版本（作了少许增强并解决了一些烦人的问题）。IEEE在2005年发布Verilog的新版本。为了避免在Verilog2005和SystemVerilog之间可能的分裂，Accellera已经在2004年夏天把SystemVerilog的版权赠送给IEEE。

#### 9.4.6 SystemC

现在我们还有SystemC，有的工程师热爱它，还有一些却恰恰相反。SystemC（将在第11章中讨论它的更多细节）能够在RTL的抽象级别上<sup>①</sup>描述设计。随后这些描述用于仿真时的速度要比Verilog或VHDL快5~10倍，而且综合工具可以把这些SystemC的RTL转换成门级网表。171

一个支持SystemC的一个理由是，它提供了更为自然的软硬件协同设计和协同验证的环境。而反对它的理由是大部分设计工程师都对Verilog或VHDL非常熟悉，但是他们不熟悉面向对象的SystemC。另一个考虑是，目前综合中的大部分是在Verilog或VHDL转译成门级网表方面投入了数百名工程师多年开发的结果。相比

<sup>①</sup> SystemC支持比RTL更高级别的抽象，然而那些界别超出了本章的讨论范围，在第11章中将讨论进一步的细节。

之下，基于SystemC的综合工具要少得多，而且那些已经问世的工具在某种程度上还没有他们的对手那么精密复杂。

在现实中，SystemC更多地用于系统级设计而不是RTL级设计。说了这么多，SystemC看上去已经在亚洲和欧洲赢得了很多动力，而且在SystemC与Verilog和VHDL之间的辩论无疑还要持续一段时间。

## 9.5 值得深思的事

### 9.5.1 担心，非常担心

大部分软件工程师在面对另一个程序员的代码时都会惊骇地举起双手，而且他们无一例外的咒骂缺少注释、没有连贯性……而且他们面对这些很不情愿。

他们不知道他们是多么地幸运，因为一个设计的RTL源代码经常为庄严设置了新的标准。有点悲哀的是，大部分用RTL描述的设计对于别的设计者几乎都是莫名其妙的。在理想情况下，一个设计的RTL描述应当读起来像一本书，以“目录”（设计结构的解释）开始，按照逻辑流程分成“章节”（设计中的逻辑划分），而且拥有大量的“注释”（解释设计结构和操作的内容）。

172

1883年，美国 William Hammer 和 Thomas Alva Edison 发现了“爱迪生效应”。

还有很重要的一点是编码风格会影响性能（通常对FPGA的影响要比ASIC大得多）。其中一个原因是，尽管它们可能逻辑上等价，但不同的RTL描述能够产生不同的结果。同样，工具也是另一个因素，因为不同的工具也能够产生不同的结果。

各种各样的FPGA厂商和EDA厂商有为他们的客户提供大量相关信息的责任，这些信息包括各自的详细代码风格和关于他们的芯片和工具的考虑。然而，以下几点是相当普遍的而且适用于大部分场合。

### 9.5.2 串行与并行多路复用器

在创建RTL代码时，理解你的综合工具如何运行是很重要的。例如，每一次你使用if-then-else语句时，结果将是一个2：1多路复用器。这在if-then-else语句嵌套的情况下会变得很有趣，因为它将会综合成一个具有优先级的结构。例如，假设我们已经声明了信号Y、A、B、C、D和SEL（用于选择），并且我们使用它们创建了一个if-then-else的嵌套结构（见图9-7）。

173

如以前一样，这里使用的是一般语法，并不真正反映任何主流的语言。这种情况下，最里面的if-then-else将是最快的路径，而最外边的if-then-else将会成为关

键信号（在时序方面）。即便如此，在很多FPGA中，这种结构的所有路径都将比使用case语句产生的路径更短。说到这里我们要提一下，以上功能使用case语句实现的结果是一个4:1多路复用器，与输入相联的所有时序路径将是（相对）均等的（图9-8）。

```
if      SEL == "00" then Y = A;
elseif SEL == "01" then Y = B;
elseif SEL == "10" then Y = C;
else                      Y = D;
end if;
```

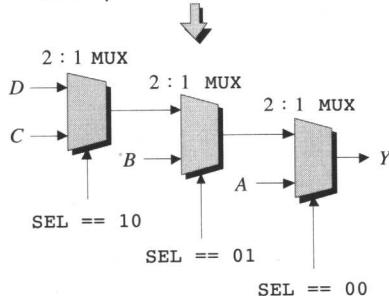


图9-7 综合一个if-then-else的嵌套结构

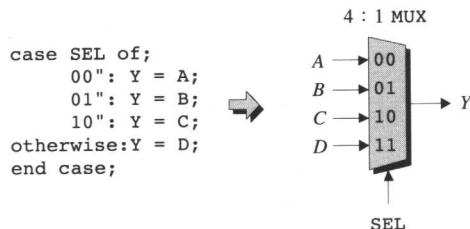


图9-8 综合一个case语句

### 9.5.3 小心锁存器

一般来讲，除非真的需要，否则在FPGA设计中避免使用锁存器是个好主意。另外要注意的是，如果使用一个if-then-else语句，然而疏忽了结束时的else部分，那么大多数综合工具都将推断出一个锁存器。

### 9.5.4 聪明地使用常量

在更为复杂的操作中，加法器是使用最多的。在某些情况下，ASIC设计者有时采用由半加器和全加器组合而成的版本。在门阵列器件中，这将非常有效，然而使用FPGA实现时一般结果都会很糟。

如果利用加法器时使用常量，一点想法就可以指导行动走得很远。比如，“ $A+2$ ”能够更有效地实现为“ $A+1$ 和进位”，而“ $A-2$ ”可以实现为“ $A-1$ 和

进位”。

类似地，使用多路复用器时，“ $A*2$ ”可以更有效地实现为“ $A \text{ SHL } 1$ ”（可理解为“ $A$ 向左移一位”），而“ $A*3$ ”将更好地实现为“ $(A \text{ SHL } 1) + A$ ”。

事实上，在FPGA中也可以很有效地利用一些代数学。比如，用“ $(A \text{ SHL } 3) + A$ ”代替“ $A*9$ ”将至少节约百分之四十的面积。

1886年 Charles Lutwidge Dodgson (Lewis Carroll) 牧师出版了用于关于逻辑表示的图表技术的书籍*The Game of Logic*。

### 9.5.5 资源共用考虑

资源共用是在HDL代码中使用同一个功能模块（比如一个加法器或比较器）来实现好几个操作的一种优化技术。

如果你没有使用资源共用，那么建立每个RTL操作时都将使用它自己的逻辑。这样的结果具有更好的性能，但是使用了更多逻辑门（相当于更多的硅面积）。如果你决定使用资源共用，结果将是减少了门数，然而通常会损失一点性能。例如，考虑图9-9展示的语句。

注意，图9-9所示的频率值只是为了这个比较而设置的，因为这些值将随具体的FPGA结构而改变，而且联机时它们在新程序节点到来时将会变化。

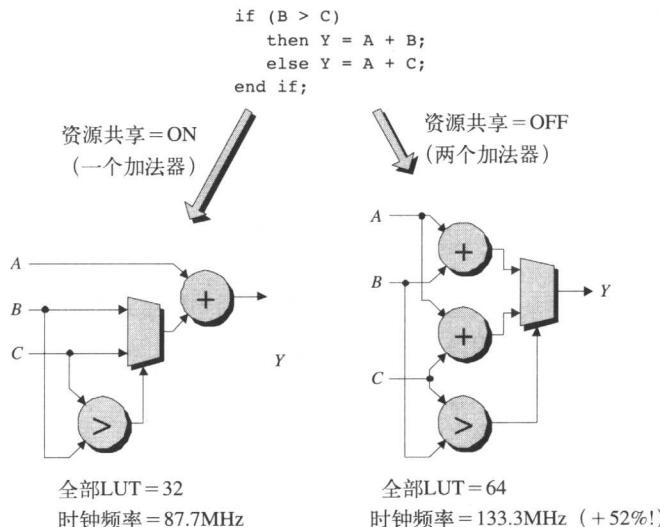


图9-9 资源共享

以下操作符可以利用同一操作符的其他实例或同一行的有关操作符来实现共用：

\*

+ -

> < >= <=

例如，+操作符能够与其他+操作符或-操作符的实例共用，而\*操作符只能与其他的\*操作符共用。

175

这里为没有太多技术背景的读者解释一下，带有“>”标志的圆形代表比较器（比较两个数决定其中哪个大的电路）；带有“+”的圆代表加法器；楔形模块是2:1多路复用器，可以根据比较器的输出而在两个输入之间进行选择。

如果没有其他问题，检查综合工具默认情况下是否具有资源共用使能或禁止是个好主意。而且最重要的一点是在ASIC中的资源共用更易于忍受布线拥挤，而在FPGA中这样可能会引起布线问题。

### 9.5.6 还有一些不可忽视的内容

在大部分FPGA中三态总线是非常慢的，应当避免使用它，除非你百分之百地确信自己知道应当如何处理。如果要列出所有可能性，只有在设计的最顶层模块才使用三态总线。如果确实希望使用内部三态总线，那么对于不支持这种门的FPGA器件族，今天大部分的综合工具都提供自动地三态到多路复用的转换（这基本上是把RTL中定义的三态缓冲器转换成相应的基于LUT/CLB的逻辑）。

176

同样，双向缓冲会引起时序回路问题。所以如果你使用它们，先要确定已经把任何错误路径都清晰地做好了标记。

177

# 第 10 章 FPGA设计中的硅虚拟原型

## 10.1 什么是硅虚拟原型

在讨论FPGA设计中的硅虚拟原型（silicon virtual prototyping, SVP）概念之前，有必要先回顾一下以下问题：这一概念最初是怎样由ASIC设计领域发展而来的，一些替代的SVP方案是怎样出现在ASIC领域的，以及与这些SVP方案相关的一些问题。

由于高端ASIC器件通常集成了几千万个逻辑门，因此这些百万门级设计的容量和复杂性都使得设计流程出现了不稳定因素。

困难在于，在传统的设计流程中，设计中存在的许多问题并不明显，只有做了精确的时序分析，并从布局布线结果中提取出实际的物理参数（电容、电阻以及电感）后，才能发现这些问题。这就需要工程师走完设计流程中所有的步骤（包含综合与布局布线）后才可以发现设计中存在的主要问题，而这些问题本来可以更容易发现、更早解决掉。

这是非常令人头痛的，要得到最终的结果往往需要进行许多次很耗时的设计迭代，因此可能对设计造成延迟，使得设计错过上市时间（许多情况下，市场中只有第一，没有第二！）。

一个解决方法是创建一个硅虚拟原型（SVP），这是一种可以快速生成的设计描述，不过（有希望）其中包含了足够的信息，设计者凭借这些信息可以在进入设计流程中那些耗时的阶段前发现和解决大部分潜在的问题。理论上，使用SVP的设计，迭代一次只需要几个小时，而传统的设计流程中则需要几天甚至几个星期。

## 10.2 基于ASIC的SVP方法

正如前一章讨论的那样，逻辑综合的作用是接受设计的RTL描述和相应的时序约束，然后自动将这些RTL描述转换为寄存器和布尔表达式的混合形式，执行最小化和优化（包括面积和时序优化）后就产生一个门级网表，这个网表有可能满足最初的时序约束。

传统的逻辑综合方案是在逻辑门尺寸与延迟组成的平面内操作的，即这样的综合工具经常会针对门的尺寸及其相关的延迟进行替换评估。鉴于这些工具的工作方式，它们执行着大量的计算集中而且非常耗时的评估任务。甚至更糟糕的是，

综合工具所完成的许多优化工作在设计提交给物理实现工具（布局布线）时显得毫无意义。

### 10.2.1 门级SVP（由快速综合产生）

SVP有一个很关键的特点就是可以快速而容易地产生。当前ASIC设计中的SVP，大多数都是用设计的门级网表表示的，随后又使用粗略的布局算法做了布局。但是，传统的综合工具为了满足原型的速度要求而消耗了大量的时间和计算资源。所以，有些ASIC设计的SVP流程就利用了一种快速的综合引擎，如图10-1所示。

180

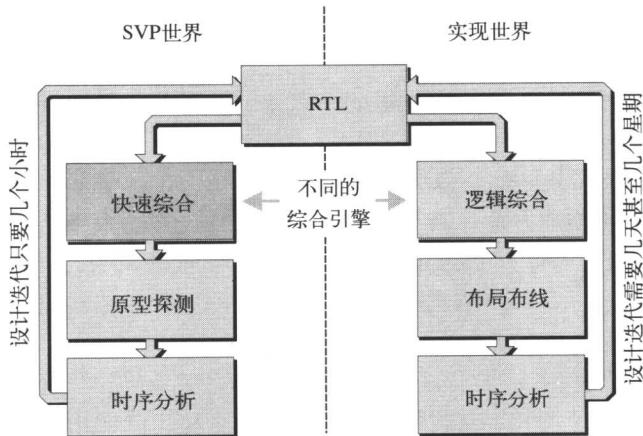


图10-1 使用快速综合的SVP

这种快速综合引擎通常使用与主流综合引擎完全不同的算法，例如，直接RTL映射。因此，构成SVP的门级网表并不像人们希望的那样能够精确地描述设计的最终实现。

这就意味着，一旦使用SVP进行了RTL探测和时序分析，工程师就必须用一种完全不同的综合引擎来执行整个设计的逻辑综合（或者物理综合），以便产生出要交给物理实现（布局布线）工具的真正的网表文件。

这样，这种基于SVP的方法就存在一个很大的问题，即原型建模工具与方法学及其实现工具与方法学是完全独立的，而且迥然不同。这就会产生一个后果：由于前后缺乏相关性，设计是否收敛就无法预测，有可能导致设计从后端向前端的迭代，相当耗时，这就有违原先使用SVP的初衷！

### 10.2.2 门级SVP（由基于增益的综合产生）

传统的逻辑综合技术是以逻辑门尺寸和延迟的相对关系为基础的，与之相比，

181

一种称为“基于增益的综合”<sup>①</sup>就显得完全不同了（我可从来没有暗示说我不喜欢这种技术）。

这种形式的综合来自于Ivan Sutherland、Bob Sproull和David Harris于1999年撰写的《逻辑效应》一书中提到的想法：设计快速的CMOS电路<sup>②</sup>。使用这种综合技术的情况下，综合引擎会利用逻辑效应的概念建立一个时序固定的平面，此后物理实现（布局布线）工具就在这个平面内工作。

这就是说，在综合结束后，所有的时序优化就完成了，所有的电路延迟都已经定了不能再更改。布局工具在执行任务时，用的是尺寸驱动的算法，所有单元都被动态地调整尺寸，并以看到的实际负载为基础满足它们的时序目标。布局之后，负载驱动的布线工具就会调整线路的宽度和间距，以便保持原始的时序目标，并保证信号的完整性。

基于增益的综合方法中，有意思的一点是，为了执行综合所需要的计算机内存和计算性能与传统综合工具相比只是一小部分。这就是说，基于增益的综合引擎在综合容量上比传统的综合方法提高了很多。

另一个有意思的地方是，基于增益的综合工具会自动用完延迟路径中所有的时序余量（slack）。即每一个逻辑门都会使用最小可能的尺寸，只要能够满足时序目标就可以了。所以，最终的实现结果占用最小的实际芯片面积，这就极大地减少了芯片面积的浪费，也减少了功耗和噪声问题。

182

但是，“你会喊到，“所有这一切与SVP有什么关系吗？”好，基于增益的综合技术与生俱来的速度和容量优势使得同样的综合技术既能用于原型设计，又能用于设计实现，如图10-2所示。

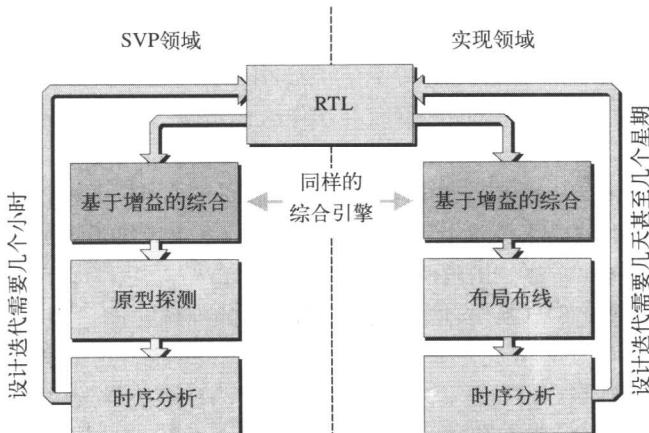


图10-2 基于SVP的快速综合技术

<sup>①</sup> 写作本书时，Magma Design Automation公司 ([www.magma-da.com](http://www.magma-da.com)) 是基于增益的综合技术的主要支持者之一。

<sup>②</sup> Ivan Sutherland因其在逻辑设计中的先驱性工作而享有很高的国际声誉。

原型设计和实现环境使用同一种算法、同一种工具和同样的方法学，因此两者之间具有很高的相关性、可预测的设计收敛性，另外还可以极大地减少后端向前端的设计迭代次数。

### 10.2.3 团簇SVP

前面讨论过，今天大多数SVP都是基于成熟的门级网表描述的。尽管这些描述是使用快速综合技术产生的，但其中仍然包含多达数百万的逻辑门，这就使得SVP布局和分析引擎的容量有些紧张。

一个解决方法是，使用团簇的概念作为SVP布局和线路延迟估计的基础。这种情况下，由快速综合或者基于增益的综合产生的单元（逻辑门和寄存器）自动聚集成小组，称为团簇（cluster）。每一个团簇一般包含几十到几百个单元，由于足够小，所以可保持很高的布局质量；与单元的数量相比，团簇的数量要少得多，因此可以极大地提高运行性能。183

为了使各个团簇的面积尽可能地接近，各团簇中包含的单元数量可能有所不同。为减小计算复杂度，降低对容量的要求，优化和分析都是针对分簇后的数据进行的。此外，当两个团簇被多条线路连接时（这是很普遍的情况），这些线路往往被看成是一条“加重”线，这是为了估计连线资源的利用率，因为它对团簇的布局有一定影响。

### 10.2.4 基于RTL的SVP

电子工程中一个被广泛接受的规则是，在设计、实现或者部署过程的任何阶段检测、隔离、解决问题所需要的成本要比在这些过程之前解决同样的问题所需成本高10多倍。对于数字IC而言，在设计流程中主要有3个节点会影响到面积和时序分析等方面，如图10-3所示。184

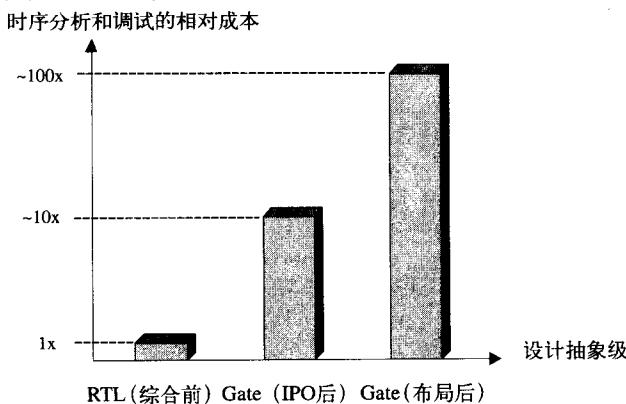


图10-3 影响面积、时序分析等方面的主要节点

时序收敛是指，对一个设计或者架构进行检测，并对任何有问题的时序路径进行修正。不管是在哪个抽象级别上，时序收敛都是一个反复的过程，即为了实现时序的收敛，分析——检测——修正这一过程通常需要执行很多次。

对于图10-3所示的抽象级别，目前最精确的是布局后的时序分析，但是这种分析的成本和时间也是最多的。布局后发现需要进行设计迭代是非常令人头痛的，所以设计团队总是努力避免在这一级别上做变更。

在传统的设计流程中，与精确时序分析有关的第一个节点是在门级，随后就是综合与在位优化（IPO）。问题是，要达到这个在位优化后的节点需要使用物理综合产生一个布过局的门级网表。所以，这种方法需要大量的计算，耗费很长的时间，一些大型的模块往往需要几天才能完成整个物理综合与时序分析过程。这不但延长了设计过程和时序收敛过程，而且还涉及昂贵的EDA工具，这些工具一般用来实现芯片而不是用来分析时序的。

IPO的意思是，布局算法在首次运算通过后，做某些“微调”（最优化）是有可能的，例如根据对线路长度的最新估计来改变单元的尺寸。

一个替代方案是使用上面谈到的门级SVP，但是前面已经说过了，这些描述也存在自己的问题，包括需要用到某些类型的计算密集和耗时的综合与布局过程。

另一种方法是使用基于RTL的SVP<sup>①</sup>，这能让工程师快速找到并解决引起时序问题的路径。为了说明其工作原理，首先要了解一种相关的应用工具，这种工具可以接受ASIC单元库中的逻辑与物理（LEF和DEF）定义文件，并由此产生一个相应的设计组件数据库，以供基于RTL的SVP使用，如图10-4所示。

185 LEF是“logical exchange format”的缩写，即逻辑交换格式，这个文件详细地说明了单元库中各个单元的逻辑功能。

与此类似，DEF是“design exchange format”的缩写，即设计交换格式，这个文件详细地说明了单元库中各个单元的物理特性，例如单元的电阻、电容以及物理尺寸等。



图10-4 产生一个设计组件

重要的是要注意到，这种设计组件并不是一个具有某种特征的逻辑门的库，

<sup>①</sup> 写本书时，基于RTL的SVP的主要支持者之一是InTime软件公司 ([www.intimesw.com](http://www.intimesw.com))。

而是一个某种特征的逻辑功能数据库（例如计数器、异或结构，等等）。设计组件生成器以这些逻辑功能的行为作为输入，包括时序和面积评估。

基于RTL的SVP生成器和分析引擎随后接受设计的RTL代码、设计模块的时序约束（工业标准的SDF格式）以及目标单元库的设计组件为输入信息。当SVP生成器读取RTL代码后，就将这些代码转换为一个称为工作函数（work function）的网表实体。每一个工作函数都是对设计组件中等效功能的一个抽象描述。

一旦RTL被转换为工作函数网表，SVP生成器就会执行一些通常在门级才执行的逻辑操作，包括通用子表达式消除（common subexpression elimination）、常数传播（constant propagation）、循环展开（loop unraveling）、消除冗余函数计算等。

SVP生成器和分析引擎利用上述过程产生的最小的无冗余的工作函数网络，并对它们执行一次“虚拟布局”。然后利用这个布局产生精确的面积评估，进而得到精确的时序评估。与设计组件结合起来，SVP生成器和分析引擎就可以知道各种综合引擎是如何为不同的因素分配比重以及如何修改实现策略的（例如交换计数器实现），并以此来满足指定的时序约束。在分析过程中，所有这些因素都要考虑进来。

186

基于RTL的SVP的支持者宣称，与用物理综合方法产生一个IPO后、布局布线前的门级网表相比，产生一个基于RTL的SVP的速度则提高了40多倍。在2003年前后，有一个450万门的设计，产生一个基于RTL的SVP需要两个半小时，而产生和分析一个IPO后的门级网表则需要99个小时。

当然，还有一个较大的问题，基于RTL的SVP是否精确？这种形式的SVP的支持者认为，其时序分析结果一般与IPO后的延迟有不到20%的差异（最坏情况下可能达到30%）。听起来尽管有些吓人，但是最新的综合工具已经有能力将这种差异控制在20%~30%之内（带来问题是那些偏差在80%、150%、200%甚至更高的路径）。因此，基于RTL的SVP已经足够精确，可以让设计工程师生成RTL代码，随后下游的综合与布局工具进行完全的实现。

我知道，我知道。我们又跑题了，尽管你也承认这都是很有趣的材料！不过我们现在就回过头来考虑FPGA。

## 10.3 基于FPGA的SVP

毫不奇怪，数百万的FPGA设计也正在面临着与ASIC同样的问题，包括花很长时间进行布局布线以及进行时序分析。

在这一过程中，特别令人痛苦的一点是，尽管原始设计的RTL描述几乎总是层次化的<sup>①</sup>，但是FPGA布局布线工具通常会将设计完全打平，从而失去层次化信

187

<sup>①</sup> 所谓的“层次化”，是指设计的顶层一般是由多个功能模块组成的，每一个模块称为子模块。

息。这就意味着，即使只对RTL代码中某个模块作最微小的改变，并且仅对该模块重新综合，你也必须对整个设计重新运行布局布线操作。如此一来，等整个设计最终实现了时序收敛，你都老得头发花白了。

为了解决这些问题，一些EDA厂商已经开始提供一些工具，能够将平面规划与布局布线前的时序分析结合起来以提供对FPGA SVP的支持。再加上对单独的设计模块进行布局布线的能力，这就极大地加速了整个实现的过程。<sup>①</sup>

这种形式的SVP是从目标FPGA器件的一个图形化的自顶向下的视图开始的，其中列出了所有的内部逻辑资源，例如查找表（LUT）、寄存器、片（slice）、可配置逻辑块（CLB）、嵌入式RAM和乘法器，等等。

在接下来的逻辑综合步骤中（使用自己选择的综合器），SVP产生器载入层次化的LUT/CLB级网表，以及与之相对应的时序和物理约束，随后自动产生一个初始的平面规划。这个自动生成的平面规划图由一些正方形或者矩形块组成，每一个块都对应于设计中的一个顶层模块。此外，如果这些顶层模块本身还包含一些子模块，那么在平面规划中这些子模块将显示为嵌入式模块（一直持续下去，直到最低层次）。

**188** SVP产生器对每一个模块使用的资源（查找表、RAM、乘法器等）进行各自的初始化布局。这些资源也显示在器件的自顶向下视图中，还有一些连接这些模块的布线资源的图形化描述。

### 10.3.1 交互式操作

设计的初始布局可以在运行布局布线之前提供一些模块级的精确时序估计。如果发现任何区域存在潜在的问题，你可以交互地修改平面规划图以便解决它们。

最简单的修改方式是改变平面规划图中矩形的形状，拖动它们的边，使这些矩形变得更高、更细、更短或者更宽。另外，你也可以构造一些更复杂的形状，例如L形、U形和T形等（除了正方形和矩形外，你还可以构造出很多不同的形状）。

接下来，你还可以移动这些模块。当你选定一个模块并将它拖出了器件的表面，系统就会弹出一个图形指示框，确定是否有必要的资源来实现这个位置的模块（你只能在具有足够资源的区域内移动模块）。而且，当你改变某个模块的形状或者四处移动它时，系统会动态地显示模块内部相关资源的利用率（查找表、寄存器、RAM和乘法器等），即相对于该模块当前包含的各种资源的总数而言的利用率。

你也可以将现有的模块拆分成两个或者更多的子模块，然后分别对它们进行操作。另外，你还可以将两个或者更多模块合并为一个单独的模块。还有些情况（大多为控制逻辑），你可能希望将一个或者多个子模块从它们的父模块中拉出来，移动到设计的顶层中，这时候你也可以对它们进行变形、合并、移动等操作。这

<sup>①</sup> 写作本书时，这里描述的FPGA SVP的主要支持者之一是Hier Design公司 ([www.hierdesign.com](http://www.hierdesign.com))。

189

反映了一种不同的哲学，即怎样使用ASIC平面规划工具。例如，对于ASIC设计而言，如果你有两个中间有很多互连线的模块，你通常会在布局时将它们并排放在一起。相比而言，在FPGA设计中，将这两个模块合并起来（这样就可以让布局布线工具利用局部和全局布线资源更好地完成优化工作）可能更有效一些。

而且，你不仅局限于对原始的RTL描述的模块进行操作。实际上，你可以操作单个的FPGA资源，例如查找表（LUT）、寄存器、片（slice）、可配置逻辑块（CLB），等等。这些操作包括：在当前层次模块内进行拖动和重新配置，从一个层次模块中拖动到另一个，创建新的模块，从一个或者多个现有的模块中向新的模块中拖动查找表组，等等。

在这里，一切才开始真正变得智能起来，如果你返回去对原始的RTL做一些修改并重新综合了相关的模块，那么当你重新输入新得到的LUT/CLB级网表时，SVP产生器会自动找到对应的模块，并将正确的逻辑装载入适当的平面规划模块中。（它们是怎么做到这一点的？我对此毫不知情！）

### 10.3.2 增量式布局布线

一旦你作好布局布线的准备，你就可以选择一个或者几个平面规划模块，并且撇开FPGA厂商的布局布线软件。每一个模块被看成是一个独立的实体，所以只要你部署好一个模块后，它将保持不变，除非你决定改变它。这种方法有很多优点。首先，对单个模块进行布局布线所需的时间要比传统的对整个数百万门的设计进行布局布线的时间少得多。

另外，即使单独对某个模块做布局布线的时间增加了，但总的时间还是要比将整个设计作为一个整体进行布局布线的时间要少得多。这是因为，布局布线的复杂度（与运行时间有关）随着要处理的模块规模的增加而非线性地增长。而且，只要你对所有模块做了布局布线，此后你就可以对单个模块进行修改，仅仅对相关的模块重新运行一下布局布线即可，而不会影响芯片的其余部分。

另一个优点是，这种SVP方法有助于创建和保护IP。也就是说，一个模块做完布局布线后，你就可以将其锁定，把它作为一个新的结构化的LUT/CLB级网表输出，当然还有相应的物理和时序约束。这个模块以后就可以在其他设计中被重用（它的部署是相对的，即这个模块可以按照上面提到的方法在芯片内部进行拖动和重新部署）。

### 10.3.3 基于RTL的FPGA SVP

在理想情况下，能够用基于RTL的FPGA SVP工作是非常好的。各个FPGA和EDA厂商提供各种复杂度的RTL级平面规划工具。但是，在笔者写这本书时，还没有出现能够与基于RTL的ASIC SVP技术相媲美的FPGA SVP技术（不过，在不久的将来，毫无疑问我们会看到这种技术）。

190

191

# 第11章 基于C/C++等语言的设计流程

## 11.1 传统的HDL设计流程存在的问题

在第9章中我们介绍了传统的基于HDL的设计流程，在这种流程中，设计是从原始概念开始的，其高层定义由系统架构师和系统设计师决定。宏观结构的定义正是在这个阶段制定的，例如将设计划分为硬件和软件部分（见第13章）。

注意，本章重点关注普通数字设计中的C/C++设计流程。有些需要考虑的事项，如量化（将设计的浮点描述转换为对应的定点描述的过程）将在第12章的以DSP为中心的设计中讨论。

将制定好的设计规范交给硬件设计工程师，他们首先会定义一些微观结构，例如细化控制结构、总线结构和主要的数据路径等。这些微观结构的定义通常在脑力激荡会上完成，有可能包含并行操作和串行操作的对比、流水线操作和非流水线操作的对比、资源共享（例如，两次操作共用一个乘法器或者各自用专门的资源）等。

最后，用VHDL或者Verilog在RTL级将设计意图描述出来。接着通过仿真对设计进行验证，然后将RTL代码综合成结构化网表，以便被目标工艺的布局布线软件使用，如图11-1所示。

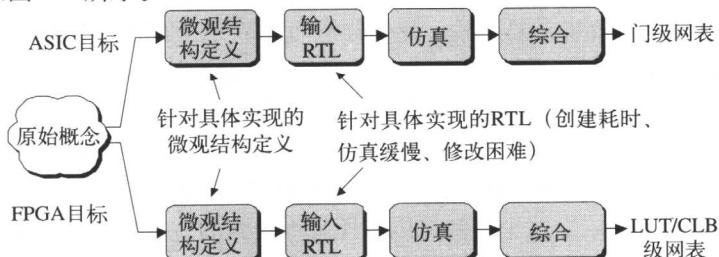


图11-1 传统（简单）的基于HDL的设计流程

在目标器件为FPGA时，根据FPGA厂商的不同，LUT/CLB级网表可能使用EDIF、VHDL或者Verilog来描述。

对于物理级综合流程，EDIF还是“最好的选择”。这种情况下，布局信息可能集成到了EDIF文件中，也可能需要外界的“约束”文件指定。

在撰写本书时，基于VHDL或Verilog的设计流程在所有ASIC和FPGA设计中大约占95%的比例；但是这些流程还存在很多问题需要解决：

- 编写RTL代码很耗时间：尽管Verilog和VHDL都是专门为描述硬件而产生的，但是用这些语言描述一个设计的功能仍然需要相当长的时间。
- 验证RTL代码也很耗时间：用仿真来验证大型的RTL设计，其计算费用非常昂贵，而且需要大量的时间。
- 评估可替代实现方案很困难：修改和重新验证RTL，以便对各种替代微观结构进行一系列的假设评估是困难而且耗时的。这就是说，设计团队能够执行的评估次数是有限的，从而导致最后的实现并不是最优的。
- 规范变更很困难：如果在设计过程中对规范做了任何变更，要将这些变更反映到RTL中并进行必要的重新验证，这将是非常漫长的。在某些应用领域内，这是一个很重要的问题，例如无线通信应用，因为无线电广播标准和协议处于不断的发展中，经常在改变。
- RTL是针对实现的：用FPGA完成一个设计所用的RTL编码类型通常与ASIC设计中的不同（也可参考第7、8、18章）。这就是说，将一个用RTL描述的复杂的设计从一种实现技术上转移到另一种实现技术上可能非常困难。当我们要将一个现有的ASIC设计移植到FPGA中，或者创建一个作为未来ASIC实现原型的FPGA设计时，这就有很大的关系了。

换句话说，设计的所有实现思想都被硬编码成RTL，因此总是针对某种专门的实现。这种实现的专门性跨越ASIC和FPGA两个领域，这表明用于FPGA设计的RTL并不适于最优的ASIC设计，反之亦然，理解这一点很重要。即使用的是单一的目标器件架构，根据目标应用领域的不同，一组算法处理数据的方法可能也需要许多不同的微观结构。

实际上，我们或许应该注意到，ASIC和FPGA设计中所用的RTL代码可能是相同的。这样做的原因是为了避免在转换代码过程中引入功能性的缺陷，但是这样一来也付出了一定的代价。也就是说，如果原始代码是为FPGA设计的，而随后被用到了ASIC设计中，那么最后得到的ASIC一般要比用专门针对ASIC设计的RTL得出的结果占用更大的硅片面积，功耗也更大。同样，如果原始代码是为ASIC设计的，而随后被用到了FPGA中，那么最终的FPGA设计要比用专门针对FPGA设计的RTL得出的结果具有更低的性能。

- RTL用于软硬件协同设计时不够理想：片上系统（SoC）器件一般被理解为含有微处理器内核。不管这些片上系统设计是用ASIC实现还是用FPGA实现，今天的SoC正在展示出越来越多的软件成分。再加上硬件方面的设计重用，许多情况下，同时验证硬件和软件以完全确认某些情况是非常必要的，

193

194

195

例如系统诊断、实时操作系统（RTOS）、设备驱动和嵌入式应用软件等。一般而言，将VHDL或者Verilog描述的硬件与C/C++或者汇编语言写的软件联合起来进行验证（仿真）是一件很痛苦的事情。

实时系统是指这样的系统，其计算或者动作的正确性不仅与系统怎样运行有关，而且还与何时运行有关。

要解决上面列举的问题，有一个方法是在一个比VHDL或者Verilog更高抽象级上开始输入设计。这样的抽象级中首先就是使用某些形式的C/C++语言，但是这并不简单，因为有许多选择，例如SystemC、增强的C/C++和纯C/C++。

## 11.2 C对C++与并行执行对顺序执行

在开始讨论以前，应该做几个标志点，以确保我们都可以步调一致地前进。首先，有多种多样的编程语言可供使用，但是除了一些专门的领域外，使用最普遍的还是传统的C语言及其面向对象的扩展C++语言。在这里，我们将它们统称为C/C++。

下一个重点是，默认情况下，C/C++这类语言中的语句是顺序执行的。例如，我们已经声明了3个整型变量，分别是a、b和c，以及以下语句：

```
a = 6;      /* Statement in C/C++ program */
b = 2;      /* Statement in C/C++ program */
c = 9;      /* Statement in C/C++ program */
```

196

很显然，这些语句的执行是依次发生的。但是，这也产生了某种暗示。例如，如果接下来是如下几个语句：

```
a = b;      /* Statement in C/C++ program */
b = a;      /* Statement in C/C++ program */
```

那么，a（初始值为6）将被赋值为当前存储在b中的值（即2）。接下来，b（初始值为2）将被赋值为当前存储在a中的值（现在为2），这样一来，a和b中最后存储的是同样的值。

编程语言的顺序结构是软件工程师的思考方式。但是硬件工程师的看法却很不相同。我们假设某个硬件电路包含两个寄存器，分别是a和b，由同一个时钟信号驱动。这两个寄存器中分别存储着6和2。如果在HDL中有如下代码行：

```
a = b;      /* Statement in VHDL/Verilog Code */
b = a;      /* Statement in VHDL/Verilog Code */
```

通常，上面的语法并不代表实际的VHDL或者Verilog语法，它只是本例使用的一个通用语法。一般而言，硬件工程师希望这两个语句并行（同时）执行。也就是说，变量a（初始值为6）将被存入变量b中的值（即2），同时变量b（原值为2）将存入a中的值（即6）。结果就实现了a和b内容的互换。

197

当然，上面所说只是一些简单的情况。不过，HDL语句默认情况下都是并行

执行的，除非通过技术手段如阻塞型赋值来强迫使用顺序执行。因此，默认情况下，基于RTL的仿真器将以这种并行方式执行上面提到的语句。同样，基于RTL的逻辑综合工具也会综合出同时处理这两个动作的硬件电路。相比而言，除非是显式地规定为并行方式（通过本章后面介绍的技术方法），否则C/C++语句总是顺序执行的。

## 11.3 基于SystemC的设计流程

### 11.3.1 什么是SystemC以及它从哪里来

在我们开始讨论基于SystemC的设计流程之前，先简要概括一下什么是SystemC，这可能是个好主意，因为在这一点上还有一些模糊（笔者认为这是很重要的）。

SystemC由Open SystemC Initiative (OSCI) 管理。这是一个独立的非盈利性组织，由一些公司、大学和一些致力于将SystemC发展成为系统级设计的开源标准的个人组成。

SystemC的代码，以及一个集成的仿真器和设计环境可以从该组织的官方网站上得到，网址为[www.systemc.org](http://www.systemc.org)。

### 11.3.2 SystemC 1.0

SystemC背后的基本概念之一是，它是一个开源的环境，每一个人都可以做出自己的贡献。举一个例子，Linux最初是非常简陋的。随着很多人对Linux做出了自己的贡献，Linux最终成为一个真正的操作系统（OS），并具备了向微软挑战的潜力。

在这种氛围中，一个无文档的SystemC 1.0于2000年前后诞生了，其结构还相当松散。SystemC 1.0是一个C++类库，主要是方便某些概念的描述，例如并发性、时序特性以及输入输出引脚（I/O）等。工程师通过这个类库就可以使用SystemC语言在RTL抽象级上描述设计了。

这个早期雏形的优点之一是，它方便了软硬件协同设计环境。另一个优点是在RTL抽象级的SystemC设计描述在仿真速度上要比相应的VHDL或者Verilog描述快5倍~10倍<sup>①</sup>。缺点是用SystemC 1.0进行RTL级的设计比用VHDL或Verilog更难也更耗时间。此外，能将SystemC 1.0描述的设计综合为各种复杂度的等效网表级

<sup>①</sup> 这与设计有关。事实上，有些SystemC描述的RTL级仿真与HDL描述的仿真在运行速度上相差不多。

设计的设计工具还很缺乏。

### 11.3.3 SystemC 2.0

后来，在2002年，SystemC 2.0出现了。这一版本对1.0版进行了增强，具备了一些高层次建模结构，例如FIFO（一种先进先出存储器）。2.0版也引进了许多种行为、算法以及系统级建模能力，例如事务与通道的概念（这些概念可以用来描述抽象级模块之间的数据通信）。

为了对这些概念有一些初步的认识，我们先来看看使用SystemC 1.0时是怎样工作的。举一个简单的例子，假设有两个函数 $f(x)$ 和 $g(x)$ ，它们之间需要进行数据通信，如图11-2所示。

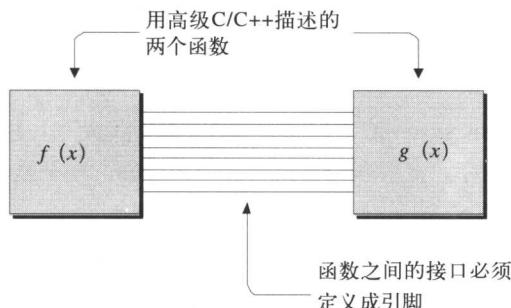


图11-2 SystemC 1.0中的接口

在这种情况下，各个模块之间的接口必须在引脚级进行定义。但是如果用这种方法，在设计的早期阶段就会出现问题，因为那时你就已经定义了一些实现的细节，例如总线宽度。如果你想试验一下各种不同的结构，这些细节改变起来就很困难。在SystemC 2.0中，这些事情就变得更加容易，可以先声明一些模块间的抽象接口，如图11-3所示。

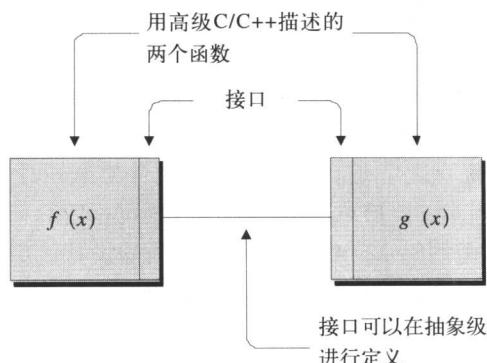


图11-3 SystemC 2.0中的接口

现在，模块间的接口可以在较为抽象的层次上连接，这是因为在设计的早期，我们真正关心的不是数据怎样从a点到达b点，而是为何到达那里。

这些抽象的接口方便了在设计的早期进行架构评估。一旦架构确定下来后，就可以开始用一些高级结构如FIFO来细化接口的定义，例如赋予一些属性，如宽度和深度，以及一些特征如块写、非块读、在空和满时怎样动作，等等。再往后，这一逻辑接口就可以用一个完整的规范化（引脚级）接口代替，这个新的接口将各个功能模块在物理级上连接在一起。[200]

### 11.3.4 抽象级

就实际情况而言，事情在这里开始变得有些模糊了，因为对不同的人有不同的定义。但是，由于是第1次讨论，我们先将SystemC的不同抽象级别画在图中，如图11-4所示。

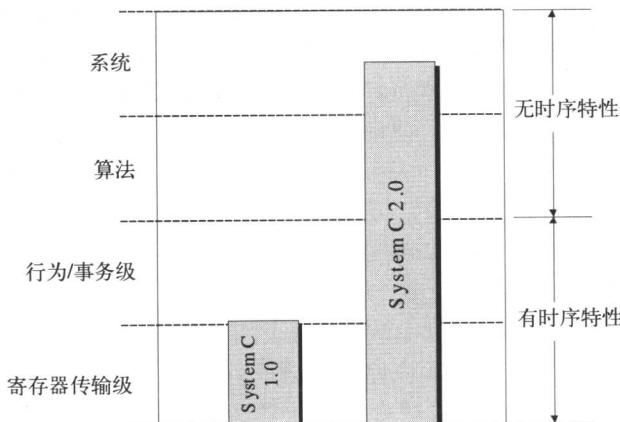


图11-4 SystemC的抽象级

这就使事情变得混乱，因为SystemC对于所有人而言意味着一切。有些人认为它是RTL级的VHDL或Verilog的替代品，也有些人认为它是一种既能用于制定系统级规范、进行算法与结构分析、行为设计，又能用于在验证中编写testbench的单一语言。

当谈到行为综合的时候，就遇到了这种混乱的一个方面。这既包含算法的因素，又包含事务级的因素（在后者中，定义事务时必须小心）。

### 11.3.5 基于SystemC设计流程的可选方案

这是一个奇妙的世界，条条大路通罗马。例如，今天的设计中有许多是为实现一个很复杂的算法。这种情况下，普遍的做法是先建立一个C或者C++的设计描