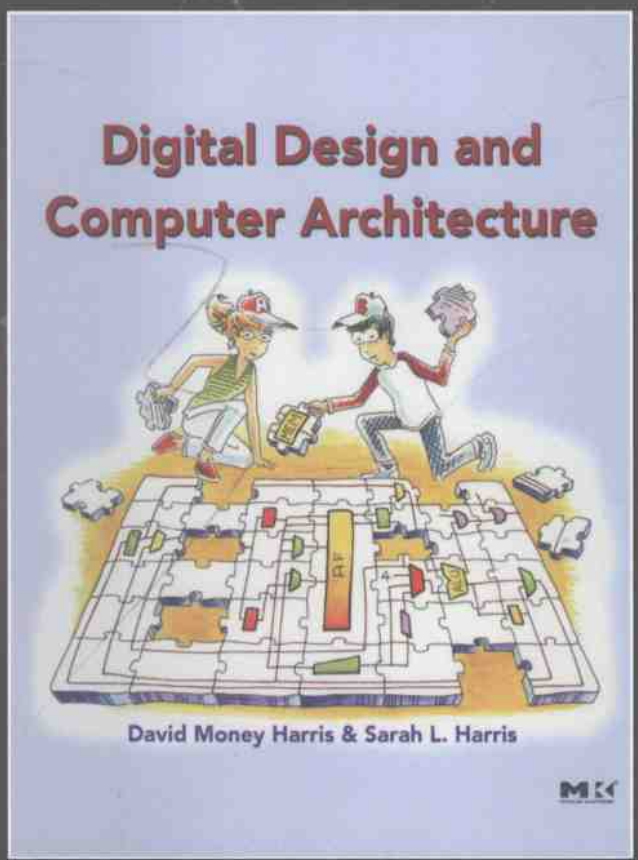
计 算 机 科 学 丛 书

数字设计和计算机体系结构

(美) David Money Harris 哈维玛德学院 Sarah L. Harris 著 陈虎 华南理工大学 等译



Digital Design and Computer Architecture

机械工业出版社
China Machine Press

数字设计和计算机体系结构

本书采用流行的教学法，从计算机组成和设计向下至更精细的层次，详细展示如何用Verilog和VHDL构建MIPS微处理器。本书为学生提供了一个很好的机会，使他们可以在现代FPGA上进行大型的数字设计，既能增长学生的专业知识，又能启发学生运用所学知识去解决实际问题。

——David A. Patterson，《计算机组成与设计》的作者之一

本书采用了一种独特的现代数字设计方法，先介绍数字逻辑门，接着讲述组合电路和时序电路的设计，并以这些基本的数字逻辑设计概念为基础，重点介绍如何设计实际的MIPS处理器。另外，在全书的实例中运用Verilog和VHDL展示基于CAD的电路设计方法和技术。通过本书，读者能够构建自己的微处理器，并能够自顶向下地理解微处理器的工作原理。

本书基于作者丰富的教学和实践经验，以幽默的写作风格展示了最新的数字设计方法。

本书特色

- 介绍数字逻辑设计的基础知识，并通过设计实际的MIPS微处理器来加强逻辑概念。
- 在例题部分，分别用两种最流行的硬件描述语言(Verilog和VHDL)给出相应数字系统设计的实现。
- 通过大量示例，帮助读者加深对关键概念和技术的理解和记忆。

作者简介

David Money Harris 哈维玛德学院工程学副教授。他曾经为英特尔、惠普、Sun等公司设计微处理器。

Sarah L. Harris 哈维玛德学院工程学副教授。她在斯坦福大学获得电子工程博士学位，擅长计算机体系结构设计和系统设计。

本书译自原版Digital Design and Computer Architecture，并由Elsevier授权出版

数字设计和计算机体系结构（英文影印版）
书号：978-7-111-22393-1
定价：65.00元

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

上架指导：计算机科学及应用

ISBN 978-7-111-25459-1

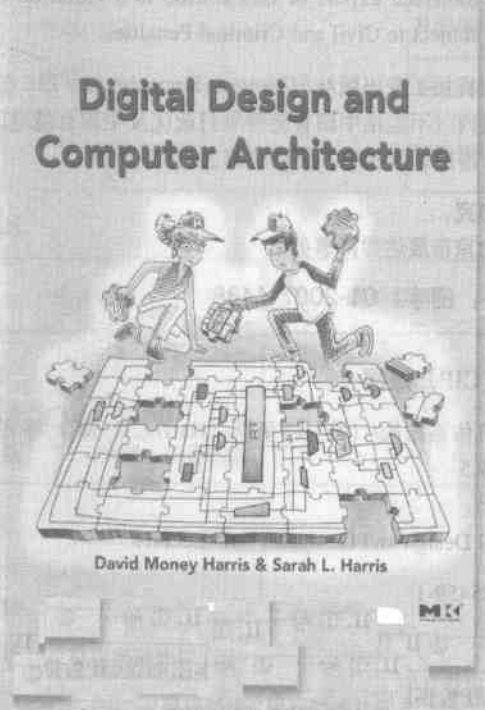
9 787111 254591

定价：58.00元

计 算 机 科 学 丛 书

数字设计和计算机体系结构

(美) David Money Harris Sarah L. Harris 著
哈维玛德学院
陈虎 等译
华南理工大学



Digital Design and
Computer Architecture

机械工业出版社
China Machine Press

本书以一种流行的方式介绍了从计算机组织和设计到更细节层次的内容，涵盖了数字逻辑设计的主要内容，展示了使用VHDL和Verilog这两种主要硬件描述语言设计MIPS处理器的技术细节，并通过MIPS微处理器的设计强化数字逻辑的概念。本书的典型特色是将数字逻辑和计算机体系结构融合，教学内容反映了当前数字电路设计的主流方法，并突出计算机体系结构的工程特点，书中的大量示例及习题设计也可以加强读者对基本概念和技术的理解和记忆。

本书不仅适合作为计算机、电子工程、电气与控制等专业的教材，同时也适合从事数字电路设计的工程师和技术人员参考。

David Money Harris and Sarah L. Harris: *Digital Design and Computer Architecture*.
ISBN-13:978-0-12-370497-9, ISBN-10:0-12-370497-9.
Copyright © 2007 by Elsevier Inc. All rights reserved.
Authorized Simplified Chinese translation edition published by the Proprietor.
ISBN:978-981-259-987-2
Copyright © 2009 by Elsevier (Singapore) Pte Ltd. All rights reserved.
Printed in China by China Machine 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.

本书简体中文版由机械工业出版社与Elsevier(Singapore)Pte Ltd.在中国大陆境内合作出版。本版仅限在中国境内（不包括中国香港特别行政区及中国台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2007-4488

图书在版编目（CIP）数据

数字设计和计算机体系结构/（美）哈里斯（Harris, D. M.）等著；陈虎等译. —北京：机械工业出版社，2009.5
（计算机科学丛书）
书名原文：Digital Design and Computer Architecture

ISBN 978-7-111-25459-1

I. 数… II. ① 哈… ② 陈… III. 数字电路-逻辑设计-教材 IV. TN79

中国版本图书馆CIP数据核字（2009）第059954号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：刘立卿
北京瑞德印刷有限公司印刷
2009年5月第1版第1次印刷
184mm×260mm · 25.75印张
标准书号：ISBN 978-7-111-25459-1
定价：58.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：（010）88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

华章教育

相关评论

本书以一种易于接受的方式介绍了从计算机组织和设计到更细节层次的教学内容，展现了如何使用VHDL和Verilog语言设计MIPS处理器的技术细节。为学生提供在现代FPGA上实现大型数字系统设计的机会，书中提供的方法既向学生传授了知识又具有启发性。

——David A. Patterson，加州大学Berkeley分校

本书为传统的教学内容提供了新的视角。很多教科书看上去像繁杂的灌木丛，作者在这本书中将“枯枝”去除，同时保留了最基本的内容，并把这些内容放到了现代的环境中。正因为如此，他们提供的教材可以激发学生为未来挑战设计解决方案的兴趣。

——Jim Frenzel，爱达荷大学

Harris的写作风格引人入胜，而且能提供很多知识。他们对材料的运用水平很高，通过大量的图来引导学生进入计算机工程领域。组合逻辑、微结构和存储器系统等内容处理得非常好。

——James Pinter-Lucke, Claremont McKenna学院

Harris所写的这本书非常清晰而且易于理解。练习的设计非常好，同时也提供了很多现实案例。这本书避免了许多其他教材中冗长而费解的解释。很明显，作者花费了很多时间和努力来提高本书的可读性。本人强烈推荐这本书。

——Peiyi Zhao, Chapman大学

Harris撰写了一部成功融合数字逻辑和计算机体系结构的教材。这是一本很受欢迎的教科书，它介绍了很多数字系统设计的内容，同时详细解释了MIPS体系结构的细节。本人强烈推荐这本书。

——James E. Stine, Jr., 俄克拉何马州立大学

这是一本令人印象深刻的书。Harris将晶体管、电路、逻辑门、有限状态机、存储器、算术部件等微处理器设计中的所有重要元素完美地结合在一起，并最终引出计算机体系结构。这本书为理解如何完美地设计复杂系统提供了很好的指导。

——Jaeha Kim, Rambus公司

这是一本写得非常好的书，不仅适用于第一次学习这些领域的年轻工程师，而且可以为有经验的工程师提供参考。本人强烈推荐这本书。

——A. Utku Diril, Nvidia公司

译者序

当译者从书店里第一次看到Harris所著的《Digital Design and Computer Architecture》的影印版时就被它深深吸引。这本书的与众不同之处在于以下方面：

1) 将数字逻辑和计算机体系结构融合。计算机硬件课程体系自身应该是完整的，数字逻辑是微处理器的设计基础，而微处理器又为数字逻辑提供了广阔的应用和设计空间。但是，当前的主流课程设置将这两者分割，学生不能通过微处理器设计来进一步加强数字系统的实际设计能力，而且计算机组成原理和体系结构课程也变成了一门“理论”课，学生几乎没有机会独立设计一个完整的处理器系统，使得这些课程的效果大打折扣。本书所采用的新结构为解决计算机专业硬件课程的问题提供了新的方法和途径。

2) 教学内容反映了当前数字电路设计的主流方法。集成电路是发展最快的领域，一个合格的数字电路设计工程师必须能掌握当前主流的数字电路设计方法和工具。遗憾的是，当前主流的数字逻辑课程还是以74系列的小规模集成电路为教学内容，对现代数字设计中最常用的硬件描述语言、FPGA等关键技术介绍不足，教学内容与工程实践脱节。本书通过大量的实例说明基于硬件描述语言和FPGA的数字电路设计方法，使得学生可以采用当前主流的CAD工具完成多个实验，大大增强了学生数字设计的实际应用能力。

3) 突出计算机体系结构的工程特点。与纯理论课程不同的是，计算机体系结构本质上是面向复杂数字电路的设计工程，在设计过程中最重要的就是根据芯片面积、速度、功耗、成本等诸多约束条件，从多种可能的设计方案中做出合理的设计抉择和折中。这种工程化的设计思想对于学生未来从事大型系统设计具有重要意义，而这一点恰恰又是当前教学中所欠缺的一环。本书通过对多种设计方案的介绍和对比，在“一题多解”的基础上说明了大型工程设计中经常碰到的设计抉择问题。对学生建立工程化的设计方法和思想很有帮助。

4) 非常易于理解。本书内容清新，文字流畅，没有冗长的概念解释，而是配以丰富的实例和清晰易懂的电路图和时序图。这对于帮助初学者理解数字电路的结构和常用表示方法无疑非常重要，可以帮助读者很快地理解数字电路和微处理器的基本原理和设计方法。

本书不仅适合作为计算机、电子、电气与控制等专业的教材，而且对于从事数字电路设计的工程师和技术人员也可以作为参考手册。

本书译者在华南理工大学软件学院和计算机学院使用本书的英文版对2007级本科学生进行双语教学，收到了很好的效果。

本书由华南理工大学陈虎副教授主持翻译定稿。此外，杨涛、王俊和何健华等人也参加了本书的部分翻译工作，对于他们的支持和帮助，在此表示衷心的感谢。

由于时间和水平有限，书中难免仍存在错误，敬请读者指正。

译者

2008年8月

前　言

目前已经有很多优秀的数字逻辑设计书籍，也有一些很好的计算机体系结构教材（例如Patterson和Hennessy撰写的经典教材），为什么还需要再出版一本包含了数字逻辑设计和体系结构的书呢？这本书的独特之处在于从计算机体系结构的视角来讲解数字逻辑设计，内容从基本的二进制开始，直到引导学生完成MIPS处理器的设计。

多年来，我们曾在Harvey Mudd学院使用了多个版本的《Computer Organization and Design》一书（由Patterson和Hennessy撰写）。我们特别欣赏该书涵盖了MIPS处理器的体系结构和微结构这种风格，MIPS处理器是获得了商业成功的体系结构，而且它也非常简单，可以用作学生的引导课程，也可由学生自主设计和实现。由于我们的课程没有预修课程，前半个学期需要介绍数字逻辑设计，而这部分内容《Computer Organization and Design》没有包含。其他大学也表示需要一本能包含数字电路设计和体系结构的教材。于是，我们开始着手准备这样一本包含了数字逻辑设计和体系结构的书。

我们相信设计处理器对于电子工程和计算机专业的学生来说是一个特殊而重要的经历。对于外行而言，处理器内部的工作几乎像魔术一样，然而事实证明，如果详细解释的话，处理器的工作原理非常易于理解。数字逻辑设计本身是一个强有力且令人激动的主题。汇编语言程序揭示了处理器内部所用的语言。而微结构将两者联系在一起。

这本书适合作为在一个学期内完成教学的数字逻辑设计和计算体系结构的入门课程，也可以用于两个学期的教学，以便有更多的时间来消化和理解书中所讲的知识并在实验室中进行实践。唯一需要的预修内容是熟悉一种高级程序设计语言（例如C、C++或Java）。本教材一般在大学本科二年级或者三年级使用，也可以提供给有编程经验和基础较好的一年级学生学习。

特点

这本书有以下特点。

并列讲述Verilog和VHDL语言

硬件描述语言（Hardware Description Languages, HDL）是现代数字逻辑设计实践的中心，而设计者分成了Verilog语言和VHDL语言两个阵营。在介绍组合逻辑和时序逻辑设计后，这本书紧接着就在第4章中介绍硬件描述语言。硬件描述语言将在第5章和第7章用于设计处理器的模块和整个处理器。然而，如果不讲授硬件描述语言的话，第4章可以跳过去，后续章节仍然可以继续使用。

这本书的特色在于使用并列的方式讲述Verilog语言和VHDL语言，使得读者可以快速地对比两种语言。第4章描述了适用于这两种硬件描述语言的原则，而且并列给出了这两种语言的语法和示例。这种并列方法使得在教学中教师可以选择其中一种硬件描述语言讲述，也可以让读者在专业实践中很快地从一种描述语言转到另外一种描述语言。

经典的MIPS体系结构和微结构

本书的第6章和第7章主要介绍了MIPS体系结构。这部分内容主要改编自Patterson和

Hennessy的论著。MIPS是一个理想的体系结构，因为每年有上百万实际产品投入使用，而且高效和易于学习。同时，世界各地上百所的大学已经围绕MIPS体系结构开发了教学内容、实验和工具。

现实视角

本书的第6～8章列举了Intel公司IA-32系列处理器的体系结构、微结构和存储器层次。这些章节以现实视角揭示了书中所讲的概念如何应用到大多数PC内部芯片的设计中。

高级微结构概览

第7章中介绍了现代高性能微结构的特征，包括分支预测、超标量、乱序执行操作、多线程和多核处理器。这些内容即使对于第一次上体系结构课程的学生也比较易于理解，展示了微结构原理如何扩展到现代处理器设计中。

章末的练习和面试题目

学习数字设计的最佳方式是实践。每章的结束部分都有很多练习来实际应用所讲述的内容。练习后面是一组工业领域的同事向申请工作的学生提出的面试题目。这些问题可以让学生了解到面试过程中可能遇见的典型问题类型。练习的答案可以通过本书的配套网址和教师支持网址获得。

在线补充资料

补充材料可以通过textbook.elsevier.com/9780123704979获得。这个出版公司对所有读者开放的网站包括了以下内容：

- 奇数编号练习的答案；
- Xilinx® 和Synplicity®公司专业版计算机辅助设计工具的链接；
- PCSPIM（一个基于Windows的MIPS模拟器）的链接；
- MIPS处理器的硬件描述语言代码；
- Xilinx的Project Navigator工具的提示；
- PPT格式的电子教案；
- 简单的课程和实验素材；
- 勘误表。

教师网站（链接到出版公司网站，仅提供给经textbooks.elsevier.com注册的使用者）包括：

- 偶数编号练习的答案；
- Xilinx® 和Synplicity®公司专业版计算机辅助设计工具的链接；取得资格认证的大学教师可以获得免费的Synplicity工具，用于教学和实验。更详细的内容请参见教师网站；
- JPG格式和PPT格式的书中插图。

关于在课程中使用Xilinx、Synplicity和PCSPIM工具以及构建实验的更详细内容请参见下文介绍。

如何使用课程中的软件工具

Xilinx ISE WebPACK

Xilinx ISE WebPACK是Xilinx公司ISE Foundation FPGA设计工具的免费版本。基于此软

件，学生可以使用原理图或者硬件描述语言（Verilog或VHDL）完成数字逻辑设计。在完成设计后，学生可以使用Xilinx WebPACK中包含的ModelSim MXE III Starter工具模拟电路。Xilinx WebPACK中还包含了用于综合Verilog或者VHDL程序的XST工具。

WebPACK和Foundation两个软件的差异在于WebPACK仅仅支持Xilinx公司部分常用FPGA器件。ModelSim MXE III Starter和ModelSim商业版的区别在于Starter版本降低了10 000多行的硬件描述语言代码的模拟速度。

Synplify Pro

Synplify Pro® 是一个面向FPGA和CPLD的高性能专业逻辑综合引擎。Synplify Pro包含了HDL Analyst（一个图形接口工具），用于根据硬件描述语言产生电路原理图。我们发现这个工具对于学习和调试过程有极大的帮助。

Synplicity公司非常慷慨地向有资格的大学捐献Synplicity Pro软件，并提供许多许可证用于大学的实验室。教师可以访问教师网站获知如何申请Synplicify Pro的许可证。关于Synplicity更详细的信息可以访问www.synplicity.com/university网站。

PCSPIM

PCSPIM（又称为简单的SPIM）是一个基于Windows的可运行MIPS汇编代码的MIPS模拟器。学生可以在文本文件中输入MIPS汇编代码，通过PCSPIM进行模拟。PCSPIM显示了指令、存储器和寄存器的值。用户手册和示范文件的链接可以通过本书配套网站（textbooks.elsevier.com/9780123704979）访问。

实验

配套网站提供了从数字逻辑设计到计算机体系结构一系列实验的链接。这些实验教授学生如何使用Xilinx WebPACK或Foundation工具来输入、模拟、综合和实现他们的设计。这些实验也包含了使用PCSPIM模拟器完成汇编语言编程的内容。

经过综合设计后，学生可以在Digilent公司的Spartan 3 Starter开发板或者XUP-Virtex 2 Pro（V2Pro）开发板上实现自己的设计。这些功能强大而且具有价格优势的开发板可以通过www.digilent.com获得。我们提供的实验描述了如何使用WebPACK在Digilent公司的Spartan 3开发板上实现一些设计。但是，Xilinx WebPACK不支持V2Pro开发板上的大容量FPGA。有资格的学校可以联系Xilinx大学合作计划以申请免费的完整Foundation工具。

为了运行这些实验，学生需要下载并安装Xilinx WebPACK、PCSPIM和Synplify Pro（可选）。教师也需要选择软件安装在实验室的机器上。这些实验包括了如何在Digilent公司的Spartan 3 Starter开发板上实现项目的指导。这些实现步骤可以跳过，但是我们认为它有很大的价值。这些实验可以使用XST综合工具，但我们推荐使用Synplify Pro工具，因为电路的原理图将反馈给学生重要的信息。

我们在Windows平台上测试了所有的实验，当然这些工具也可以在Linux上使用。

错误

正如所有有经验的程序员所知，比较复杂的程序毫无疑问都有潜在错误。这本书也不例外。我们花费了大量的精力查找和消除这本书的错误。然而，错误仍然不可避免。我们将在这本书的网站上维护和更新勘误表。

请将你发现的错误发送到ddcabugs@onehotlogic.com。第一个报告实质性错误而且在后续版本中采用了其修改的读者可以得到1美元的奖励！（请确认包含了自己的邮件地址。）

致谢

首先，我们要感谢David Patterson和John Hennessy，他们在《Computer Organization and Design》一书中对MIPS处理器微结构进行了开创性的介绍。我们多年以来讲授了该书的多个版本。我们感谢他们对这本书慷慨的支持，以及允许在他们的微结构上进行设计。

我们喜爱的卡通画作家Duane Bibby花了很长时间和努力来说明数字电路设计中有趣的奇遇。我们也很感激Morgan Kaufamann公司的Denise Penrose、Nate McFadden以及团队的其他同事，没有他们的热情这本书将无法面世。Graphic World出版服务公司的Joff Somers在出版过程中也提供了大力指导。

很多审阅人也大大地提高了这本书的质量。他们包括：Ithaca学院的John Barr，Charleston Southern 大学的Jack V. Briner，SK通信公司的Andrew C. Brown，Harvey Mudd 学院的Carl Baumgaertner，Nvidia公司的A. Utku Diril，Idaho大学的Jim Frenzel，Rambus公司的Jaeha Kim，ShotSpotter公司的Phillip King，Claremont McKenna学院的James Pinter-Lucke，Connecticut大学的Amir Roth和Z. Jerry Shi，Oklahoma州立大学的James E. Stine，Chapman大学的Luke Teyssier和Peiyi Zhao，以及一位匿名评阅人。Simon Moore是作者David在剑桥大学进行学术休假时的东道主，而本书的大部分内容正是在此期间完成的。

我们也非常感谢Harvey Mudd学院上这个课程的学生们，他们对此书的草稿提供了有帮助的反馈。需要特别记住的是Casey Schilling、Alice Clifton、Chris Acon和Stephen Brawner。

本书作者David特别感谢他的妻子Jennifer。在这个项目开始的时候，他们的儿子Abraham降生。感谢她的耐心和生活最忙碌时对另一个项目的支持。

目　录

出版者的话
相关评论
译者序
前言

第1章　二进制……1
1.1　课程计划……1
1.2　控制复杂性的艺术……1
1.2.1　抽象……1
1.2.2　约束……2
1.2.3　三条原则……3
1.3　数字抽象……4
1.4　数字系统……5
1.4.1　十进制数……5
1.4.2　二进制数……5
1.4.3　十六进制数……6
1.4.4　字节，半字节和全字……8
1.4.5　二进制加法……8
1.4.6　有符号的二进制数……9
1.5　逻辑门……11
1.5.1　非门……12
1.5.2　缓冲……12
1.5.3　与门……12
1.5.4　或门……12
1.5.5　其他二输入逻辑门……12
1.5.6　多输入门……13
1.6　数字抽象之下……14
1.6.1　电源电压……14
1.6.2　逻辑电平……14
1.6.3　噪声容限……15
1.6.4　直流电压传输特性……15
1.6.5　静态约束……16
1.7　CMOS晶体管*……17
1.7.1　半导体……17
1.7.2　二极管……18
1.7.3　电容……18
1.7.4　nMOS和pMOS晶体管……18
1.7.5　CMOS非门……20
1.7.6　其他CMOS逻辑门……20
1.7.7　传输门……22
1.7.8　类nMOS逻辑……22
1.8　功耗*……23
1.9　总结和展望……23
习题……24
第2章　组合逻辑设计……32
2.1　引言……32
2.2　布尔表达式……34
2.2.1　术语……34
2.2.2　与或式……34
2.2.3　或与式……35
2.3　布尔代数……36
2.3.1　公理……36
2.3.2　单变量定理……36
2.3.3　多变量定理……37
2.3.4　定理的统一证明方法……39
2.3.5　等式化简……39
2.4　从逻辑到门……40
2.5　多级组合逻辑……42
2.5.1　减少硬件……43
2.5.2　推气泡……44
2.6　X和Z……45
2.6.1　非法值X……45
2.6.2　浮空值Z……45
2.7　卡诺图……46
2.7.1　画圈的原理……47
2.7.2　卡诺图化简逻辑……48
2.7.3　无关项……50
2.7.4　小结……51
2.8　组合逻辑模块……51
2.8.1　多路选择器……51
2.8.2　译码器……54
2.9　时序……55
2.9.1　传输延迟和最小延迟……56

2.9.2 毛刺 ……59
2.10 总结 ……60
习题……61
第3章 时序逻辑设计……65
3.1 引言 ……65
3.2 锁存器和触发器 ……65
3.2.1 SR锁存器 ……66
3.2.2 D锁存器 ……67
3.2.3 D触发器 ……68
3.2.4 寄存器 ……68
3.2.5 带使能端的触发器 ……69
3.2.6 带复位功能的触发器 ……70
3.2.7 晶体管级的锁存器和触发器设计* …70
3.2.8 小结 ……71
3.3 同步逻辑设计 ……72
3.3.1 一些有问题的电路 ……72
3.3.2 同步时序电路 ……74
3.3.3 同步和异步电路 ……75
3.4 有限状态机 ……75
3.4.1 有限状态机设计实例 ……75
3.4.2 状态编码……79
3.4.3 Moore型状态机和Mealy型状态机 …82
3.4.4 状态机的分解 ……86
3.4.5 有限状态机小结 ……86
3.5 时序逻辑电路的时序 ……87
3.5.1 动态约束……88
3.5.2 系统时序……88
3.5.3 时钟偏移* ……92
3.5.4 亚稳态 ……94
3.5.5 同步器 ……95
3.5.6 分辨时间的推导* ……97
3.6 并行 ……99
3.7 总结 ……101
习题 ……102
第4章 硬件描述语言 ……109
4.1 引言 ……109
4.1.1 模块 ……109
4.1.2 硬件描述语言的起源 ……110
4.1.3 模拟和综合 ……111
4.2 组合逻辑……112
4.2.1 按位操作符 ……112
4.2.2 注释和空格 ……114
4.2.3 缩减运算符 ……114
4.2.4 条件赋值 ……115
4.2.5 内部变量 ……117
4.2.6 优先级 ……119
4.2.7 数字 ……120
4.2.8 z和x ……120
4.2.9 位混合 ……122
4.2.10 延迟 ……123
4.2.11 VHDL 库和类型* ……124
4.3 结构建模 ……126
4.4 时序逻辑 ……129
4.4.1 寄存器 ……129
4.4.2 可复位寄存器 ……131
4.4.3 带使能端的寄存器 ……132
4.4.4 多寄存器 ……133
4.4.5 锁存器 ……134
4.5 更多组合逻辑 ……135
4.5.1 选择语句 ……137
4.5.2 if语句 ……141
4.5.3 Verilog的casez语句* ……142
4.5.4 阻塞式和非阻塞式赋值 ……142
4.6 有限状态机……146
4.7 参数化模块* ……152
4.8 测试程序 ……155
4.9 总结 ……161
习题 ……162
第5章 常见数字模块……171
5.1 引言 ……171
5.2 算术电路 ……171
5.2.1 加法 ……171
5.2.2 减法 ……176
5.2.3 比较器 ……177
5.2.4 算术逻辑单元 ……178
5.2.5 移位器和循环移位器……179
5.2.6 乘法* ……180
5.2.7 除法* ……181
5.2.8 深入阅读 ……182
5.3 数制系统 ……182
5.3.1 定点数系统 ……183
5.3.2 浮点数系统* ……183
5.4 时序电路模块 ……186
5.4.1 计数器 ……186

5.4.2 移位寄存器 ……187
5.5 存储器阵列……190
5.5.1 概述 ……190
5.5.2 动态随机访问存储器……192
5.5.3 静态随机访问存储器……193
5.5.4 面积和延迟 ……193
5.5.5 寄存器文件 ……193
5.5.6 只读存储器 ……193
5.5.7 使用存储器阵列的逻辑……195
5.5.8 存储器的硬件描述语言……195
5.6 逻辑阵列 ……197
5.6.1 可编程逻辑阵列 ……197
5.6.2 现场可编程门阵列 ……198
5.6.3 阵列实现*……202
5.7 总结 ……203
习题 ……203
第6章 体系结构……210
6.1 引言 ……210
6.2 汇编语言 ……210
6.2.1 指令 ……211
6.2.2 操作数：寄存器、存储器和常数 …212
6.3 机器语言 ……216
6.3.1 R-类型指令 ……216
6.3.2 I-类型指令 ……217
6.3.3 J-类型指令 ……218
6.3.4 解释机器语言码 ……218
6.3.5 程序存储 ……219
6.4 编程 ……219
6.4.1 算术/逻辑指令 ……220
6.4.2 分支 ……222
6.4.3 条件语句 ……224
6.4.4 循环 ……225
6.4.5 数组 ……227
6.4.6 过程调用 ……230
6.5 寻址方式 ……236
6.6 编译、汇编和加载 ……238
6.6.1 内存图 ……238
6.6.2 转换成二进制代码和开始执行程序 ……239
6.7 其他主题 ……242
6.7.1 伪指令 ……242
6.7.2 异常 ……242
6.7.3 有符号和无符号的指令 ……243
6.7.4 浮点指令 ……244
6.8 真实世界透视：IA-32结构* ……245
6.8.1 IA-32的寄存器 ……246
6.8.2 IA-32的操作数 ……246
6.8.3 状态标志 ……247
6.8.4 IA-32指令集 ……247
6.8.5 IA-32指令编码 ……249
6.8.6 IA-32的其他特性 ……249
6.8.7 小结 ……250
6.9 总结 ……250
习题 ……251
第7章 微结构……258
7.1 引言 ……258
7.1.1 体系结构状态和指令集……258
7.1.2 设计过程 ……258
7.1.3 MIPS微结构 ……259
7.2 性能分析 ……260
7.3 单周期处理器……261
7.3.1 单周期数据路径 ……261
7.3.2 单周期控制 ……265
7.3.3 更多指令 ……267
7.3.4 性能分析 ……269
7.4 多周期处理器……270
7.4.1 多周期数据路径 ……270
7.4.2 多周期控制 ……275
7.4.3 更多指令 ……281
7.4.4 性能分析 ……282
7.5 流水线处理器……284
7.5.1 流水线数据路径 ……286
7.5.2 流水线控制 ……288
7.5.3 冲突 ……288
7.5.4 更多指令 ……297
7.5.5 性能分析 ……297
7.6 硬件描述语言表示*……299
7.6.1 单周期处理器 ……300
7.6.2 通用模块 ……306
7.6.3 测试程序 ……309
7.7 异常*……313
7.8 高级微结构*……315
7.8.1 深流水线 ……316
7.8.2 分支预测 ……317

7.8.3 超标量处理器 ……318
7.8.4 乱序处理器 ……319
7.8.5 寄存器重命名 ……321
7.8.6 单指令流多数据流 ……322
7.8.7 多线程 ……323
7.8.8 多处理器 ……323
7.9 现实世界透视：IA-32微结构* ……324
7.10 总结 ……328
习题 ……329
第8章 存储器系统 ……333
8.1 引言 ……333
8.2 存储器系统性能分析 ……335
8.3 高速缓存 ……336
8.3.1 高速缓存中存放的数据 ……337
8.3.2 高速缓存中的数据查找 ……337
8.3.3 数据的替换 ……343
8.3.4 高级高速缓存设计* ……344
8.3.5 MIPS处理器中高速缓存的发展* …347
8.4 虚拟存储器 ……347
8.4.1 地址转换 ……349
8.4.2 页表 ……350
8.4.3 地址转换后备缓冲 ……351
8.4.4 存储器保护 ……352
8.4.5 替换策略* ……352
8.4.6 多级页表* ……352
8.5 内存映射I/O* ……354
8.6 现实世界透视：IA-32存储器和I/O系统* ……357
8.6.1 IA-32高速缓存系统 ……357
8.6.2 IA-32虚拟存储器 ……359
8.6.3 IA-32的直接I/O编程机制 ……359
8.7 总结 ……359
习题 ……360
附录A 数字系统实现 ……365
附录B MIPS指令 ……391
延伸阅读材料 ……394

第1章 二 进 制

1.1 课程计划

在过去的三十年里，微处理器彻底变革了我们的世界。现在一台膝上笔记本电脑的计算能力都远远超过了过去一个房间大小的大型计算机。高级汽车上包含了大约50个微处理器。微处理器的进步使得移动电话和Internet成为可能，极大地促进了医学的进步，也改变了战争的式样。全球集成电路工业销售额从1985年的210亿美元发展到2005年的2270亿美元，其中微处理器占到了重要部分。我们相信微处理器对技术、经济和社会有重要意义，而且它也潜在地激发了人类的创造力。在读者学习完这本书后，将学会如何设计和构造一个属于自己的微处理器。这些基本技能将为读者设计其他数字系统奠定坚实的基础。

我们假设读者对电子学有基本的认识，有一定的编程经验和基础，同时对理解微处理器的内部运行原理有真正的兴趣。这本书将集中讨论基于0和1的数字系统的设计。我们从接收0和1作为输入，产生0和1作为输出的逻辑门开始本课程。接着，我们将研究如何利用这些逻辑门构成加法器、存储器等比较复杂的模块。随后，我们将学习使用以微处理器的语言——汇编语言进行程序设计。最后，我们将上述内容结合起来以构造一个能执行汇编程序的微处理器。

数字系统的一个重要特点是其构造模块相当简单：仅仅包括了0和1。它不需要繁杂的数学知识或高深的物理学知识。相反，设计者的最大挑战是如何将这些简单的部件组合起来构成复杂的系统。微处理器可能是读者构造的第一个复杂系统，其复杂性可能一下子难以全部接受。因此，如何控制复杂性是贯穿全书的一个重要主题。

1.2 控制复杂性的艺术

与非专业人员相比，计算机科学家或工程师的一个重要特征是掌握了控制复杂性的系统方法。现代数字系统由上百万，甚至数十亿的晶体管构成。没有人能通过为每个晶体管的电子运动建立并求解方程的方法来理解这样的系统。读者必须学会如何控制复杂性，从而理解如何构造微处理器系统，而不陷入繁琐的细节。

1.2.1 抽象

管理复杂性的关键技术在于抽象（abstraction）：隐蔽不重要的细节。一个系统可以从多个不同层面抽象。例如，美国的政治家将世界抽象为城市、县、州和国家。一个县包含了若干城市，而一个州则包含了若干县。当一个政治家竞选总统时，他对整个州的投票情况更有兴趣，而非单个县。因此，州在这个层次中的抽象更有益处。在另一方面，美国人口调查局需要统计每个城市的人口，必须考虑更低层次抽象的细节。

图1-1给出了一个电子计算机系统的抽象层次，其中在每个层次中都包含了典型的模块。最底层的抽象是物理层，即电子的运动。电子的特征由量子力学和麦克斯韦（Maxwell）方程描述。我们的系统由晶体管或先前的真空管等电子器件（device）构造。这些器件都有明确定

义的外部连接点，称为端子（terminal），并建立了每个端子上电压和电流之间的关系模型。通过器件级的抽象，我们可以忽略单个电子。更高一级抽象为模拟电路（analogy circuit）。在这一级中，器件组合在一起构造成放大器等组件。模拟电路的输入和输出都是连续的电压值。逻辑门等数字电路（digital circuit）则将电压控制在离散的范围内，以表示0和1。在逻辑设计中，我们将使用数字电路构造更复杂的组件，例如加法器或存储器。

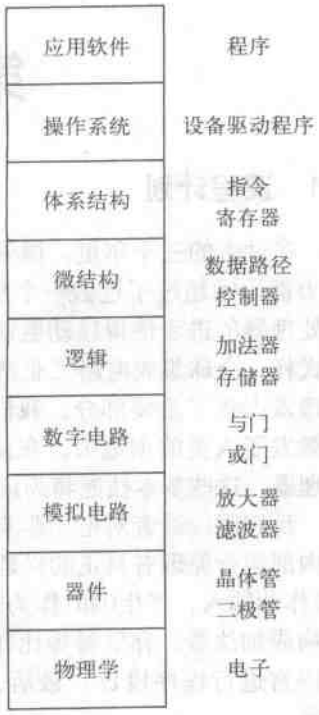


图1-1 电子计算机系统的抽象层次

微结构将逻辑和体系结构层次的抽象连接在一起。体系结构（architecture）层描述了程序员观点的计算机抽象。例如，目前广泛应用于个人计算机（Personal Computer，PC）的Intel公司的IA-32体系结构定义了一套指令系统和寄存器（用于存储临时变量的存储器），从而程序员可以使用这些指令和寄存器。微结构将逻辑组件组合在一起以实现体系结构中定义的指令。一个特定的体系结构可以有不同的微结构实现方式，以取得在价格、性能和功耗等方面的不同折中。例如，Intel公司的Core 2 Duo，80486和AMD公司的Athlon等都是IA-32体系结构的三种不同微结构实现。

进入软件层面后，操作系统负责处理底层的抽象，例如访问硬盘或管理存储器。最后，应用软件使用操作系统提供的这些功能以解决用户的问题。正是借助于抽象的威力，年迈的祖母可以通过计算机上网，而不用考虑电子的量子波动或计算机中的存储器组织问题。

这本书将主要讨论从数字电路到体系结构之间的抽象层次。当读者处于某个抽象层次时，最好能了解当前抽象层次之上和之下的层次。例如，计算机科学家不可能在不理解程序运行平台体系结构的情况下充分优化代码。在不了解晶体管具体用途的情况下，器件工程师也不能在设计晶体管时做出明智的设计选择。我们希望读者学习完本书后，能选择正确的层次以解决问题，同时评估自己的设计选择对其他抽象层次的影响。

1.2.2 约束

约束（discipline）是对设计选择的一种内在限制，通过这种限制可以更有效地在更高的抽象层次上工作。使用可互换部件是约束的一种常见应用，其典型例子是来复枪的制作。在19世纪早期，来复枪靠手工一支支地制作。来复枪的零件从很多不同的手工制作商那里买来，然后由一个技术熟练的做枪工人组装在一起。基于可互换部件的约束变革了这个产业：通过将零件限定为一个误差允许范围内的标准集合，就可以很快地组装和修复来复枪，而且不需要太熟练的技术。做枪工人不再需要考虑枪管和枪托形状等较低层次的抽象。

在本书中，对数字电路的约束非常重要。数字电路使用离散电压，而模拟电路使用连续电压。因此，数字电路是模拟电路的子集，而且在某种意义上其能力要弱于范围更广的模拟电路。然而数字电路的设计很简单。通过数字电路的约束规则，我们可以很容易地将元件组合成复杂的系统，而且这种数字系统在很多应用上都远远优于由模拟元件组成的系统。例如，数字化的电视、光盘（CD）以及移动电话正在取代以前的模拟设备。

1.2.3 三条原则

除了抽象和约束外，设计者还使用另外三条准则来处理系统的复杂性：层次化（hierarchy），模块化（modularity）和规整化（regularity）。这些原则对于软硬件的设计都是通用的。

- 层次化：将系统划分为若干模块，然后更进一步划分每个模块直到这些模块可以很容易理解。
- 模块化：所有模块有定义好的功能和接口，以便于它们之间可以很容易地相互连接而不会产生意想不到的副作用。
- 规整化：在模块之间寻求一致，通用的模块可以重新使用多次，以减少设计不同模块的数量。

我们回到制作来复枪的例子来解释这三条原则。在19世纪早期，来复枪是最复杂的常见物品之一。使用层次化原理，我们可以将它划分为图1-2所示的几个部件：枪机、枪托和枪管。

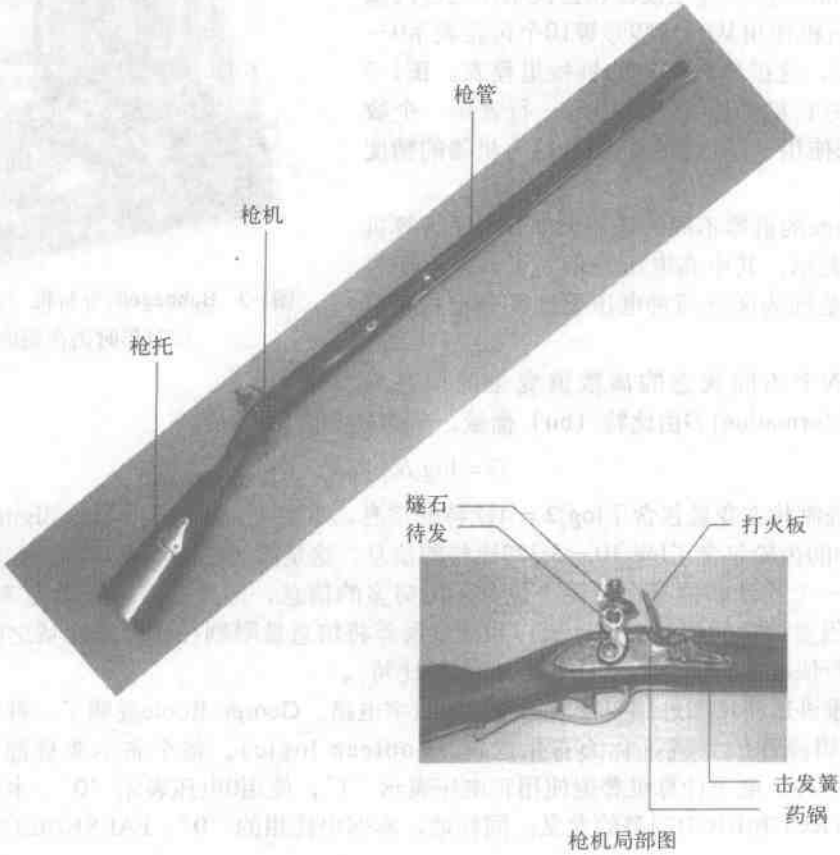


图1-2 燧石来复枪及其枪机的特写照片

枪管是一个长的金属管子，子弹就是通过这里射出；枪机是一种射击设备；而枪托是用木头制成的，它将各种部分连接起来并且为使用者提供牢固的握枪位置。然后，枪机包含扳机、击锤、燧石、扣簧和药锅。每种组件都可以用规则性来描述。

模块化使得每个模块都应有明确的功能和接口。枪托的功能是装配枪机和枪管，它的接口

包括长度和装配钉的位置。在模块化的来复枪设计中，如果长度和装配钉的位置均合适，那么来自于不同制造商的枪托可以用于特定的枪管。枪管的功能是使子弹更加精确地射出，模块化设计规定它不能对其他部件产生影响：对枪管的设计不能影响到枪托的功能。

规整化表明可互换部件是一个好的方法。利用规整化原理，一个损坏的枪管可以用相同的部分取代。可以在装配线上更有效地生产枪管，而不是辛苦地手工制作。

层次化、模块化和规整化等三条原则在本书中很重要，它们将贯穿本书的内容。

1.3 数字抽象

大部分物理变量是连续的。例如电线上的电压值、振动的频率、物体的位置等都是连续的值。相反地，数字系统使用离散值变量（discrete-valued variables）来表示信息——也就是说，变量是有限数目的不同离散值。

早期Charles Babbage的分析机（Analytical Engine）使用了具有10个离散值变量的数字系统。从1834年到1871年，Babbage一直在设计和尝试制作这种机械计算机。分析机使用从0号到9号等10个齿轮表示0～9这九个数字，这很像汽车里的机械里程表。图1-3展示了这种分析机的原型，其中每一行表示一个数字，Babbage使用了25行齿轮，因此这台机器的精度为25位数字。

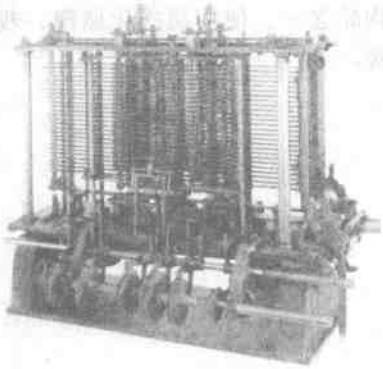


图1-3 Babbage的分析机（在他去世的1871年时仍在制造）

与Babbage的机器不同的是，大部分电子计算机使用二进制表示，其中高电压表示‘1’，低电压表示‘0’。这是因为区分两种电压要比区分十种要容易得多。

一个有N个不同状态的离散值变量的信息量(amount of information) D由比特（bit）衡量，N和D之间的关系是：

$$D = \log_2 N \text{ bits} \tag{1-1}$$

一个二进制状态变量包含了$\log_2 2 = 1$比特的信息，事实上，比特是binary digit的缩写。每一个Babbage的齿轮包含了$\log_2 10 = 3.322$比特的信息，这是因为它能够表示$2^{3.322} = 10$种不同状态的一种。一个连续的信号在理论上包含了无穷多的信息，因为它可以表示无穷多个数值。实际上对于很多连续的信号来说，噪声和测量误差将信息量限制在10～16比特之内。如果需要对信号进行快速测量，其信息量更低（例如8比特）。

本书着重讲述使用二进制变量1和0表示的数字电路。George Boole发明了一种针对二进制变量进行逻辑操作的系统，称为布尔逻辑（boolean logic）。每个布尔变量都是TRUE和FALSE中的一种。电子计算机普遍使用正电压表示‘1’，使用0电压表示‘0’。本书中使用的‘1’表示与TRUE和HIGH同等的含义。同样地，本书中使用的‘0’，FALSE和LOW也可以相互替换。

数字抽象（digital abstraction）的优势在于设计者可以只关注0和1，而忽略布尔变量的物理表示到底是特定电压，还是旋转的齿轮，或者是液体的高度。计算机编程人员可以不需要了解计算机硬件的细节就能工作。此外，对硬件细节的理解使得程序员可以针对特定计算机来优化软件。

一个单独的比特位并没有太多的信息。下一节将用一组比特位来表示数字，后面几节将

使用一组比特位来表示字母和程序。

1.4 数字系统

我们已经习惯于使用十进制数字。但在由0和1组成的数字系统中，二进制或者十六进制数字使用起来更方便。本节将介绍在后续章节中要用到的几种数字系统。

1.4.1 十进制数

小学就已经学习过用十进制（decimal）来计数和做算术。如同我们有10个手指一样，十进制也是由0，1，2，…，9这10个数字组成。多个十进制数字组合在一起可以形成更大的十进制数。十进制数字中，每一列的权都是前一列的10倍。从右到左，每一列的权分别为1、10、100、1000等等。十进制数的基（base）为10。基往往通过数值后方的下标表示，以避免与原数值混淆。例如，图1-4描述了十进制数9742_{10}是根据每一列的权和该列的数字相乘之后求和而得到的。

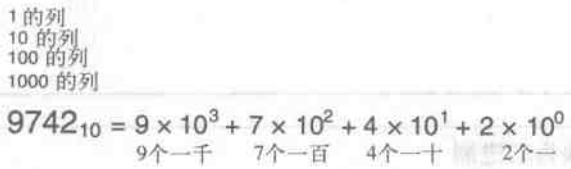


图1-4 一个十进制数的表示

一个N位的十进制数表示了10^N个数字中的某一个：0，1，2，3，…，10^N-1，称为数的表示范围（range）。例如，一个3位的十进制数表示了从0～999的1000个数字中的某一个。

1.4.2 二进制数

一位比特表示0和1两个值中的一个。将多个比特位合并在一起就形成了一个二进制数(binary numbers)。二进制数的每一列权都是前一列的2倍，因此二进制数的基数是2。在二进制数中，每一列的权(从右到左)分别为1，2，4，8，16，32，64，128，256，512，1024，2048，4096，8192，16384，32768，65536，依次类推。如果经常使用二进制数的话，记住这些2的n次方（$n \leqslant 16$）会节省很多时间。

一个N位的二进制数代表2^N个数字中的某一个：0，1，2，3，...，2^N-1。表1-1显示了1位、2位、3位和4位二进制数和与之相等的十进制数。

例1.1 二进制转换为十进制

将二进制数10110_2转换为十进制。

解：图1-5给出了转换方法。

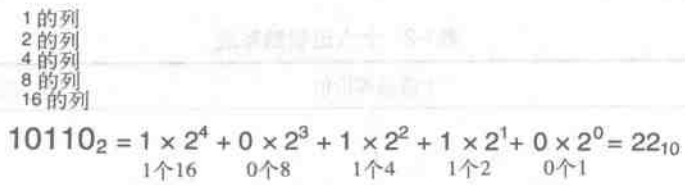


图1-5 二进制到十进制的转换

表1-1 二进制数和与之等价的十进制数

1位二进制数	2位二进制数	3位二进制数	4位二进制数	十进制等价值
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

例1.2 十进制转换为二进制

将十进制数84_{10}转换为二进制。

解：需要判断每一列的二进制数值是1还是0。从二进制数的最左或最右边都可以进行。

从左开始的话，首先从小于给定十进制数的2的最高次幂开始（本例中是64），$84 \geqslant 64$，因此权为64的这一列是1，还剩$84 - 64 = 20$，$20 < 32$，所以权为32的这一列是0，$20 \geqslant 16$，所以权为16的这一列是1，剩下$20 - 16 = 4$，$4 < 8$，所以权为8的这一列是0，$4 \geqslant 4$，因此权为4的这一列为1，剩下$4 - 4 = 0$，因此权为2和1的列的二进制数值均为0。将它们组合在一起，$84_{10} = 1010100_2$。

从右开始的话，用2重复除给定的十进制数，余数放在每一列中。$84/2 = 42$，因此权为1的这一列为0，$42/2 = 21$，因此权为2的这一列为0，$21/2 = 10$，余数是1，因此权为4的这一列为1，$10/2 = 5$，权为8的这一列为0，$5/2 = 2$，余数是1，权为16的这一列为1，$2/2 = 1$，权为32的这一列为0，$1/2 = 0$，余数是1，权为64的这一列为1。从而$84_{10} = 1010100_2$。

1.4.3 十六进制数

书写一个很大的二进制数将十分冗长且易出错。4位一组的二进制数可以表示$2^4 = 16$种数。因此，有时使用基数为16的表示会更方便，这称为十六进制(hexadecimal)。十六进制数使用数字0～9和字母A～F，如表1-2所示，十六进制数每一列的权分别是1，16，16^2（256），16^3（4096），依次类推。

表1-2 十六进制数系统

十六进制数	十进制等价值	二进制等价值
0	0	0000
1	1	0001
2	2	0010
3	3	0011

（续）

十六进制数	十进制等价值	二进制等价值
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

例1.3 十六进制转换为二进制和十进制

将十六进制数$2ED_{16}$转换为二进制和十进制

解：十六进制和二进制之间的转换很容易，其中每个十六进制数字相当于4位二进制数字。$2_{16} = 0010_2$，$E_{16} = 1110_2$，$D_{16} = 1101_2$，因此$2ED_{16} = 001011101101_2$。十六进制转换为十进制需要计算，图1-6给出了计算过程。

1 的列
16 的列
256 的列

$$2ED_{16} = \underset{\text{2个256}}{2 \times 16^2} + \underset{\text{14个16}}{E \times 16^1} + \underset{\text{13个1}}{D \times 16^0} = 749_{10}$$

图1-6 十六进制到十进制的转换

例1.4 二进制转换为十六进制

将二进制数1111010_2转换为十六进制。

解：转换非常容易，从右往左读取数据，最低4位是$1010_2 = A_{16}$，下面是$111_2 = 7_{16}$。因此$1111010_2 = 7A_{16}$。

例1.5 十进制转换为十六进制和二进制

将十进制数333_{10}转换为十六进制和二进制。

解：如同十进制转换为二进制那样，十进制转换为十六进制可以从左或从右进行。

从左开始时，从小于给定十进制数的16的最高次幂开始（本例中是256），333中仅包含了1个256，所以在权为256的这一列是1，还剩$333 - 256 = 77$，77中有4个16，所以在权为16的这一列是4，还剩$77 - 16 \times 4 = 13$，$13_{10} = D_{16}$，所以在权为1的这一列是D。因此，$333_{10} = 14D_{16}$。如例1.3所示，将十六进制转换为二进制是很容易的，$14D_{16} = 101001101_2$。

从右开始，用16重复除以给定的十进制数，余数放在每一列中。333/16 = 20，余数是$13_{10} = D_{16}$，所以权为1的这一列为D；20/16 = 1，余数为4，所以权为16的这一列为4，1/16 = 0，余数是1，所以权为256的这一列为1。最后，结果为$14D_{16}$。

1.4.4 字节，半字节和全字

8个一组的比特位称为字节（byte），它能表示$2^8 = 256$个数字。计算机内存中存储的数据习惯于用字节作单位，而不用位。

4个一组的比特位或者半个字节称为半字节(nibble)，它能表示$2^4 = 16$个数字。一个十六进制数占用1个半字节，两个十六进制数占用一个字节。半字节已经不是一个常用的单位，但这个术语很吸引人。

微处理器处理的一组数据称为字（word）。字的大小取决于微处理器的结构。在写作本书的2006年，很多计算机都采用32位处理器，意味着它们对32位的字进行操作。同时，处理64位字的计算机也即将被广泛应用。比较简单的微处理器，特别是应用在诸如烤面包机等小设备中的处理器，使用8位或16位字。

在一组位中，权为1的那一位称为最低位（least significant bit：lsb），处于另一端的位称为最高位（most significant bit：msb），如图1-7(a)所示的6位二进制数。同样地，对于一个字来说，也可用最低字节（least significant byte：LSB）和最高字节（most significant byte：MSB）来表示，如图1-7(b)所示。该图是一个4字节的数据，用8个十六进制数表示。

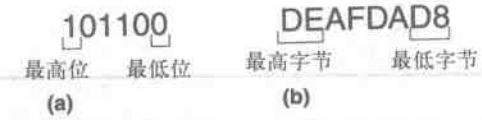


图1-7 最低位（字节）和最高位（字节）

可以利用一个很方便的巧合，$2^{10} = 102^4 \approx 10^3$，因此kilo（希腊文的千）表示$2^{10}$。例如$2^{10}$字节是1千字节（1KB）。相似地，mega（百万）表示$2^{20} \approx 10^6$，giga（十亿）表示$2^{30} \approx 10^9$。如果你知道$2^{10} \approx$ 1千，$2^{20} \approx$ 1兆，$2^{30} \approx$ 10亿，而且记住2的n次方（$n \leqslant 9$）的值，你将很容易地得出2的任意次方的值。

例1.6 估算2的n次方

不用计算器求2^{24}的近似值。

解：将指数分成10的倍数和余数。$2^{24} = 2^{20} \times 2^4$。$2^{20} \approx$ 1兆，$2^4 = 16$。因此$2^{24} \approx$ 16兆。精确地说，$2^{24} = 16\,777\,216$，但是16兆这个数据已经足够精确。

1024字节称为1千字节（kilobyte，KB）。1024比特被称为1千比特（kilobit，Kb或Kbit）。相似地，MB，Mb，GB和Gb分别叫做兆字节，兆比特，吉字节和吉比特。内存容量经常用字节做单位，信息传输速率一般用比特/秒（b/s）做单位。例如，拨号的调制解调器最大传输速率为56Kb/s。

1.4.5 二进制加法

二进制加法与十进制加法相似但更简单，如图1-8所示。在十进制加法中，如果两个数据之和大于单个数字所能表示的值，将在下一列的位置上标记1。图1-8比较了二进制加法与十进制加法。图1-8(a)的最右端的一列，7 + 9 = 16，因为16 > 9，故不能用单个数字表示，因此记录权为1的列结果（6），然后将权为10的列结果（1）进位到更高一列中。同样地，在二进制加法中，如果两个数相加之和大于1，那么我们将此按二进制进位到更高一列，如图1-8(b)所示，在图的最右端一列，$1 + 1 = 2_{10} = 10_2$，使用1个二进制位无法表示此结果，因此记录此和中权为1的列结果（0），并将权为2的列结果（1）进位到更高一列中。在加法的第二列中，$1 + 1 + 1 = 3_{10} =$

图1-8 显示进位的加法例子

11_2，记录此和的权为1的列结果（1），并将权为2的列结果（1）进位到更高一列。为了更明确地表示，进位到相邻列的位称之为进位（carry bit）。

例1.7 二进制加法

计算$0111_2 + 0101_2$。

解：图1-9给出了相加结果1100_2。可以通过计算它们的十进制来检验计算结果。$0111_2 = 7_{10}$，$0101_2 = 5_{10}$，结果为$12_{10} = 1100_2$。

数字系统常常对固定长度的数字进行操作。如果加法的结果太大，超出了数字的表示范围，将产生溢出（overflow）。例如，一个4位数的表示范围是[0, 15]，如果两个4位数相加的结果超过了15，那么就会产生溢出。结果的第5位被抛弃，从而产生一个不正确的结果。可以通过检查最高一列是否有进位来判断是否溢出。

例1.8 有溢出的加法

计算$1101_2 + 0101_2$，有没有产生溢出？

解：图1-10给出了计算结果是10010_2。此结果超出了4位二进制数的表示范围。如果结果一定要存储为4位的二进制数，那么它的最高位将被抛弃，剩下一个不正确的结果0010_2。如果计算结果使用5位或更多位来表示，结果10010_2将会是正确的。

```
  111
  0111
+ 0101
------
 1100
```

图1-9 二进制加法示例

```
  11 1
  1101
+ 0101
-------
 10010
```

图1-10 有溢出的二进制加法示例

1.4.6 有符号的二进制数

到目前为止，我们只考虑了表示正数的无符号（unsigned）二进制数。我们还需要一种能表示正数和负数的二进制数字体系。有多种方案可以表示有符号（signed）二进制数，其中最常用的两种为：带符号的原码和二进制补码。

带符号的原码

带符号的原码（sign/magnitude）是一种直观的数据表示方式，符合我们写负数的习惯：把负号标在数字前面。一个N位带符号的原码数中的最高位为符号位，剩下的$N - 1$位为数值（绝对值）。符号位为0表示正数，1表示负数。

例1.9 带符号的原码表示的数

用带符号的原码表示5和−5。

解：两个数字的值均为$5_{10} = 101_2$，所以，$5_{10} = 0101_2$，$-5_{10} = 1101_2$。

不幸的是，普通的二进制加法无法在带符号的原码下实现。例如，-5_{10}和5_{10}相加，用这种格式计算出来的结果是$1101_2 + 0101_2 = 10010_2$，这是没有道理的。

N位的带符号的原码的数据表示范围是$[-2^{N-1} + 1, 2^{N-1} - 1]$。这种格式在表示0时有两种方法：+0和−0。对于同一个数字有两种不同的表示方法有可能会造成麻烦。

二进制补码

二进制补码中最高位的权是-2^{N-1}而不是2^{N-1}，其他位的表示方法与无符号二进制数相同。

它克服了带符号的原码格式中0有两种表示方式的缺点。在二进制补码中，0只有一种表示方式，而且也可以使用普通的加法。

在二进制补码中，0表示成$00...000_2$。正数的最高位为0，$01...111_2 = 2^{N-1} - 1$。负数的最高位是1，$10...000_2 = -2^{N-1}$，-1表示成$11...111_2$。

注意：正数的最高位都是0，负数的最高位都是1，所以最高位可以当作符号位，然而剩余那些位的解释与带符号的原码有所不同。

二进制补码的符号位在求二进制补码（taking the two's complement）的过程中翻转。在此过程中首先对数据的每一位取反，然后在数据的最低位加1。这对于计算负数的二进制补码表示或根据二进制补码表示计算负数的值是很有用的。

例1.10 负数的二进制补码表示

把-2_{10}表示成一个4位的二进制补码数。

解：$+2_{10} = 0010_2$，为了得到-2_{10}的值，将所有位取反后加1。0010_2取反之后为1101_2，$1101_2 + 1 = 1110_2$，所以$-2_{10} = 1110_2$。

例1.11 二进制补码的负数的值

求二进制补码数据1001_2的十进制数值。

解：1001_2最高位是1，所以它一定是一个负数。为了求取它的值，将所有位取反然后加1。1001_2取反后的结果是0110_2，$0110_2 + 1 = 0111_2 = 7_{10}$，所以，$1001_2 = -7_{10}$。

用二进制补码表示的数据有一个明显的优点，加法操作对正数和负数都可得出正确的结果。当进行N位的数据加法时，第N位的进位（即第$N+1$位结果）被抛弃。

例1.12 两个二进制补码数据相加

使用补码计算（a）$-2_{10} + 1_{10}$和（b）$-7_{10} + 7_{10}$的结果

解：（a）$-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$。（b）$-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$，第五位被丢弃，剩下后四位结果$0000_2$。

减法是将第二个操作数改变符号后求取补码，然后跟第一个操作数相加来完成。

例1.13 两个二进制补码数据相减

使用补码计算（a）$5_{10} - 3_{10}$和（b）$3_{10} - 5_{10}$的结果。

解：（a）$3_{10} = 0011_2$，取二进制补码得$-3_{10} = 1101_2$，计算$5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$。注意：因为使用4位表示结果，其最高位的进位被丢弃。（b）第二个操作数5_{10}取补码得$-5_{10} = 1011$，计算$3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$。

计算0的二进制补码时，需要将所有的二进制位取反（产生$11...111_2$），然后加1，丢弃最高位，剩余$00...000$。因此，0的表示是唯一的。与带符号的原码系统不同，二进制补码表示方法中没有-0。0被认为是正数，因为它的符号位为0。

如同无符号数，N位二进制补码数能够表示2^N种数值，但是这些数分为正数和负数。例如，一个4位的无符号数可以表示16种数值：0到15，一个4位的二进制补码也可以表示16个数值：-8到7。一般而言，N位二进制补码的表示范围是$[-2^{N-1}, 2^{N-1} - 1]$。注意到负数比正数多一个，这是因为没有-0。最小的负数是$10...000_2 = -2^{N-1}$，这个数有时被叫做怪异数（weird number）。求取它的二进制补码时，首先各位取反变成$01...111_2$，然后加1，变成

$10...000_2$，与原数相同。因此，这个负数没有与之对应的正数。

两个N位正数或者负数相加，如果结果大于$2^{N-1}-1$或者小于-2^{N-1}，则会产生溢出。一个正数和一个负数相加肯定不会导致溢出。不像无符号整数加法中，最高位产生进位表示溢出。二进制补码加法中判定溢出的条件是：相加的两个数符号相同且结果的符号与被加数符号相反，表示发生溢出。

例1.14 有溢出的二进制数加法

用4位二进制数计算$4_{10}+5_{10}$。判断结果是否有溢出。

解：$4_{10}+5_{10}=0100_2+0101_2=1001_2=-7_{10}$。结果超过了4位二进制补码整数的表示范围，产生了不正确的负值结果。如果计算使用5位或更多位数计算，则结果为正确的值$01001_2=9_{10}$。

当二进制补码数扩展到更多位数时，需要将符号位复制到所有的扩展高位中。这个过程称为符号扩展（sign extension）。例如，数字3和−3的二进制补码表示分别为0011和1101。将这两个数扩展为7位时，可以将符号位复制到新的高三位中，分别得到0000011和1111101。

数制系统的比较

三种最常见的二进制数值系统分别为：无符号数、二进制补码和带符号的原码。表1-3比较了这三种数值系统中N位数的表示范围。由于二进制补码可以表示正数和负数，而且可以使用常见的加法，所以这个编码方式最为方便。减法采用将减数取反（也采用二进制补码）再和被减数相加的方法实现。除非特殊声明，我们都使用二进制补码表示有符号数。

表1-3 N位数的表示范围

数制系统	表示范围
无符号的原码	$[0, 2^N-1]$
带符号的原码	$[-2^{N-1}+1, 2^{N-1}-1]$
二进制补码	$[-2^{N-1}, 2^{N-1}-1]$

图1-11给出了每个数字系统中4位数的表示方法。无符号数在[0.15]范围内按照正常的二进制顺序排列。二进制补码表示范围为[−8, 7]。其中非负数[0, 7]的编码与无符号数相同，负数[−8, −1]中越大的二进制数越接近0。注意，怪异数1000表示−8，没有正数与之对应。带符号原码表示的范围为[−7, 7]。最高位为符号位。正数[1, 7]的编码与无符号数相同。负数表示与整数对称，而仅仅符号位为1。0可以表示为0000或1000。由于0有两种表示方法，所以N位带符号原码仅可以表示2^N-1个数。

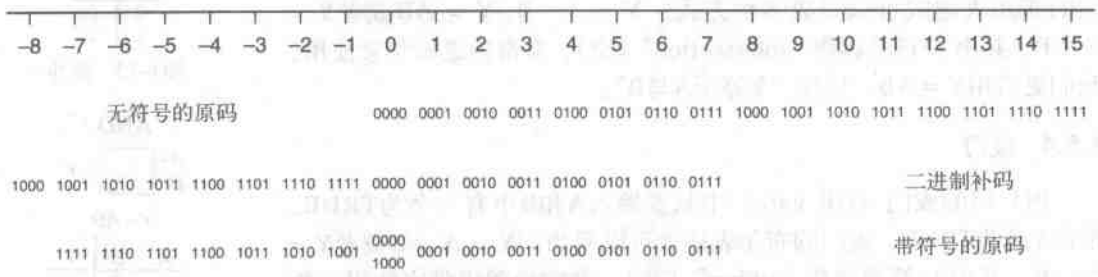


图1-11 4位二进制数编码

1.5 逻辑门

现在我们已经知道如何使用二进制变量表示信息，下面将研究对这些二进制变量进行操作的数字系统。逻辑门（logic gate）是最简单的数字电路，它们可以接收一个或多个二进制

输入并产生一个二进制输出。逻辑门用电路符号画出，并表示出输入和输出。输入往往画在左侧（或上部），输出往往画在右侧（或下部）。数字设计师往往使用字母表开始部分的字母表示门的输入，用Y表示门的输出。输入和输出之间的关系由真值表或布尔表达式描述。真值表（truth table）的左侧列出输入，右侧列出对应的输出，而且每种可能的输入组合对应一行。基于二进制变量的数学表达式。称为布尔表达式（Boolean equation）。

1.5.1 非门

非门（NOT gate）有一个输入A和一个输出Y，如图1-12所示。非门的输出是输入之反。如果A为FALSE，则Y为TRUE。如果A为TRUE，则Y为FALSE。这个关系由图中的真值表和布尔表达式所表示。布尔表达式中A上面的横线读作NOT，因此$Y = \bar{A}$读作“Y等于NOT A”。非门也称为反相器（inverter）。

还有一些对非门的表示，例如Y = A'，Y = ¬A，Y = !A，或Y =~ A。我们仅使用$Y = \bar{A}$，但读者在碰到其他类型的表示时也不要被迷惑。

1.5.2 缓冲

另一种单输入逻辑门称为缓冲（buffer），如图1-13所示。它仅仅将输入传递到输出。

从逻辑的角度看，缓冲和电线没有差异，好像没有用。然而，从模拟电路的角度看，缓冲可能有一些很好的特征使得它可以向电机传递大电流，或者将输出更快地传递到多个门的输入上。这个例子也说明了为什么我们要考虑整个系统的多个层次抽象；数字抽象掩盖了缓冲的真实作用。

三角符号表示一个缓冲。输出上的圆圈称为气泡（bubble），用来表示取反，正如图1-12中的非门符号一样。

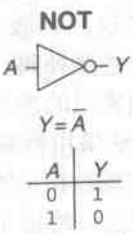


图1-12 非门

1.5.3 与门

两输入逻辑门更加有趣。图1-14中的与门（AND gate）在所有输入A和B都为TRUE时，输出Y才为TRUE。否则输出为FALSE。为了方便起见，输入按照00、01、10、11的二进制递增顺序排列。与门的布尔表达式可以写成多种方式：Y = A · B，Y = AB或者Y = A ∩ B。其中∩符号读作“intersection”（交），常常由逻辑学家使用。我们更常用Y = AB，读作“Y等于A与B”。

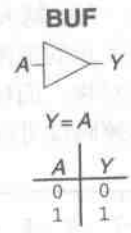


图1-13 缓冲

1.5.4 或门

图1-15的或门（OR gate）中只要输入A和B中有一个为TRUE，输出Y就为TRUE。或门的布尔表达式可以写为：Y = A + B或者Y = A ∪ B。其中∪符号读作“union”（并），常常由逻辑学家使用。数字电路工程师更常用Y = A + B，读作“Y等于A或B”。

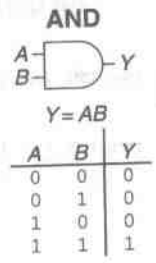
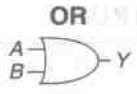


图1-14 与门

1.5.5 其他二输入逻辑门

图1-16给出了其他常见的二输入逻辑门。异或门（XOR, exclusive OR）的输入A和B中有且仅有一个输入为TRUE时，输出为TRUE。如果门后面有一个气泡，表示进行取反操作。NAND门执行与非操作。它的所有输入为TRUE时才为FALSE，其他情况都为TRUE。NOR门

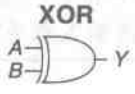
执行或非操作。它在输入A和B都不为TRUE时才输出TRUE。



$Y=A+B$

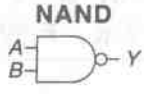
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

图1-15 或门



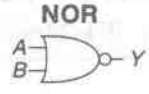
$Y=A\oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



$Y=\overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



$Y=\overline{A+B}$

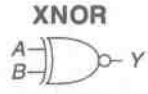
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

图1-16 其他两输入逻辑门

例1.15 XNOR门

图1-17给出两输入XNOR门的电路符号和布尔表达式。它执行异或非逻辑。请完成真值表。

解：图1-18给出了真值表。在所有输入都为TRUE或都为FALSE的情况下，XNOR输出TRUE。两输入XNOR门有时称为相等（equality）电路，因为在输入相等时输出为TRUE。



$Y=\overline{A\oplus B}$

图1-17 XNOR门

A	B	Y
0	0	
0	1	
1	0	
1	1	

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

图1-18 XNOR真值表

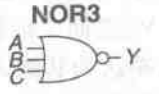
1.5.6 多输入门

有很多需要三个或三个以上输入的布尔函数。最常见的是AND、OR、XOR、NAND、NOR和XNOR。N输入与门在所有输入都为TRUE时才产生TRUE。N输入或门在有一个输入为TRUE时就产生TRUE。N输入XOR门有时也称为校验（parity）门，即有奇数个输入为TRUE时产生TRUE。正如二输入门一样，真值表中的输入组合按照二进制递增顺序排列。

例1.16 三输入NOR门

图1-19给出了三输入NOR门的电路符号和布尔表达式。请完成真值表。

解：图1-20给出了真值表。只有在所有输入都不为TRUE时，输出才为TRUE。



$Y=\overline{A+B+C}$

图1-19 三输入NOR门

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

图1-20 三输入NOR门的真值表

例1.17 四输入AND门

图1-21给出了四输入AND的电路符号和布尔表达式。请完成真值表。

解： 图1-22给出真值表。只有所有的输入都为TRUE时，输出才为TRUE。

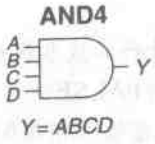


图1-21 四输入AND门

A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

图1-22 四输入AND门的真值表

1.6 数字抽象之下

数字系统采用离散取值的变量。然而这些变量需要由连续的物理量来表示，例如电线上的电压，齿轮的位置，或者桶中的水位高度。所以，设计者必须找到一种方法将连续变量和离散变量联系在一起。

例如，考虑采用电线上的电压来表示二进制信号A。当电压为0V时，A = 0；为5V时，表示A = 1。任何实际系统都必须能容忍一定的噪声，因此4.97V可能也可以解释为A = 1。但是对于4.3V呢？对于2.8V或2.500000V呢？

1.6.1 电源电压

假设系统中最低的电压为0V，称为地（ground，GND）。系统中最高的电压来自电源，常称为V_{DD}。在1970～1980年代的技术下，V_{DD}一般为5V。当芯片采用了更小的晶体管，V_{DD}降到了3.3V, 2.5V, 1.8V, 1.5V, 1.2V，甚至更低以减少功耗和避免晶体管过载。

1.6.2 逻辑电平

通过定义逻辑电平（logic level），可以将连续变量映射到离散的二进制变量，如图1-23所示。第一个门称为驱动源（driver），第二个门称为接收端（receiver）。驱动源的输出连接到接收端的输入上。驱动源产生LOW（0）输出，其电压处于0～V_{OL}之间；或者产生HIGH（1）输出，其电压处于V_{OH}～V_{DD}之间。如果，接收端的输入电压处于0～V_{IL}之间，则接收端认为其输入为LOW。如果接收端的输入电压处于V_{IH}～V_{DD}之间，则接收端认为其输入为HIGH。如果由于噪声或部件错误原因，接收端输入电压处于V_{IL}和V_{IH}之间的禁止区域（forbidden zone），则输入门的行为不可预测。V_{OH}和V_{OL}称为输出高和输出低逻辑电平，V_{IO}和V_{IL}称为输入高和输入低逻辑电平。

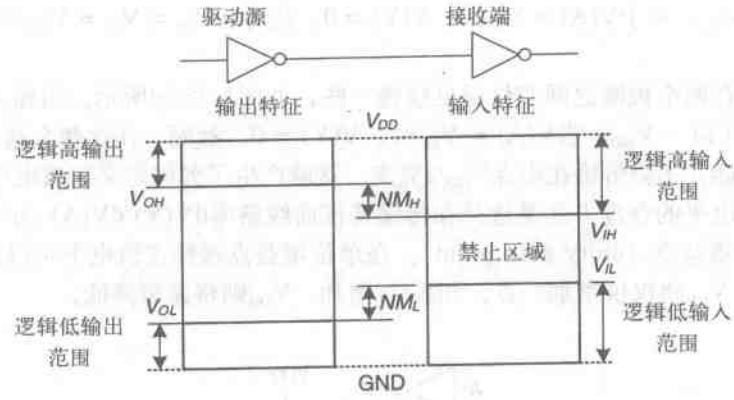


图1-23 逻辑电平和噪声容限

1.6.3 噪声容限

如果驱动源的输出能够被接收端的输入正确解释，我们必须选择$V_{OL} < V_{IL}$，$V_{OH} > V_{IH}$。因此，如果驱动源的输出被一些噪声干扰，接收端的输入依然能够检测到正确的逻辑电平。叠加在输出上但依然能正确解释为有效输入的最大噪声值，称为噪声容限（noise margin）。如图1-23可以看出，低电平和高电平的噪声容限分别为：

$$NM_L = V_{IL} - V_{OL} \tag{1-2}$$

$$NM_H = V_{OH} - V_{IH} \tag{1-3}$$

例1.18

考虑图1-24中的反相器。V_{O1}是反相器I1的输出电压，V_{I2}是反相器I2的输入电压。两个反向器遵循同样的逻辑电平特征：$V_{DD} = 5V$, $V_{IL} = 1.35V$, $V_{IH} = 3.15V$, $V_{OL} = 0.33V$, $V_{OH} = 3.84V$。反相器的低电平和高电平噪声容限分别为多少？这个电路可否承受V_{O1}和V_{I2}之间1V的噪声？

解：反相器的噪声容限为：$NM_L = V_{IL} - V_{OL} = (1.35V - 0.33V) = 1.02V$，$NM_H = V_{OH} - V_{IH} = (3.84V - 3.15V) = 0.69V$。电路在输出为LOW时，可以承受1V的噪声电压（$NM_L = 1.02V$）；但是在输出为HIGH时，不能承受此噪声电压（$NM_H = 0.69V$）。例如，在驱动源I1输出的HIGH值处于最坏情况，即$V_{O1} = V_{OH} = 3.84V$。如果噪声导致电压在到达接收端输入前降低了1V，$V_{I2} = (3.84V - 1V) = 2.84V$。这已经小于可接受的HIGH逻辑电平$V_{IH} = 3.15V$，因此接收端将无法检测到正确的HIGH输入。

图1-24 反相器电路

1.6.4 直流电压传输特性

为了理解数字抽象的局限性，我们必须深入考察门的模拟特征。门的直流电压传输特性（DC transfer characteristics）描述了当输入电压变化足够慢能保证输出能跟上输入的变化时，输出电压随输入电压变化的函数关系。这个函数之所以称为传输特性，是因为它描述了输入和输出电压之间的关系。

理想的反相器应在输入电压达到门限$V_{DD}/2$时产生一个跳变，如图1-25(a)所示。对于V(A)

$< V_{DD}/2$，$V(Y) = V_{DD}$。对于$V(A) > V_{DD}/2$，$V(Y) = 0$。此时，$V_{IH} = V_{IL} = V_{DD}/2$，$V_{OH} = V_{DD}$，且$V_{OL} = 0$。

真实的反相器在两个极端之间变化得更缓慢一些，如图1-25(b)所示。当输入电压V(A)等于0时，输出电压$V(Y) = V_{DD}$。当$V(A) = V_{DD}$时，$V(Y) = 0$。然而，在这两个端点之间的变化是平滑的，而且可能并不会恰恰在中点$V_{DD}/2$突变。这就产生了如何定义逻辑电平的问题。

一种选择逻辑电平的合理方法是选择在传输特征曲线斜率dV(Y)/dV(A) 为−1的位置。这两个位置称为单位增益点（unity gain point）。在单位增益点选择逻辑电平可以最大化噪声容限。如果V_{IL}减少，V_{OH}将仅仅增加一点。如果V_{IL}增加，V_{OH}则将显著降低。

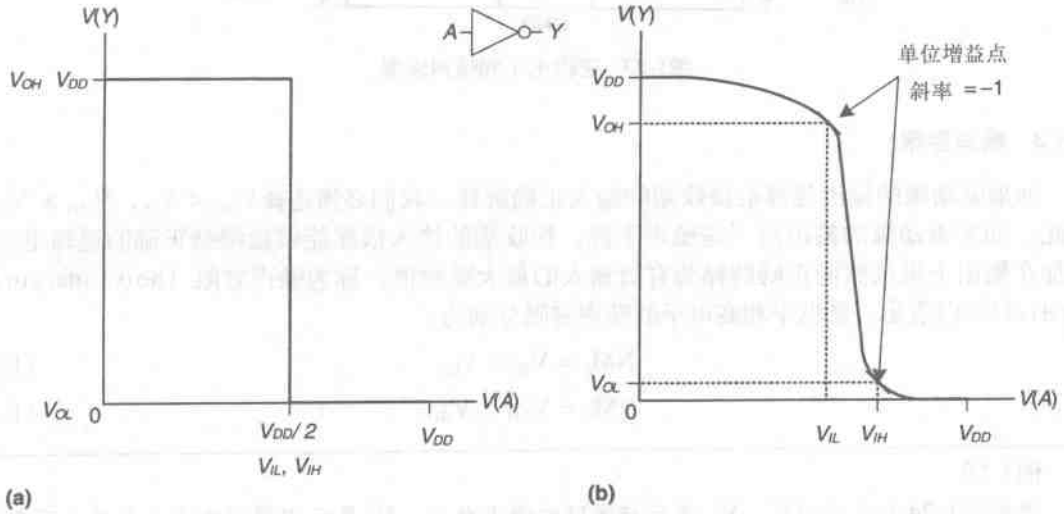


图1-25 直流传输特性和逻辑电平

1.6.5 静态约束

为了避免输入处于禁止区域，数字逻辑门的设计需要遵循静态约束（static discipline）。静态约束要求对于给定的有效逻辑输入，每个电路元件应该能产生有效的逻辑输出。

为了满足静态约束，数字电路设计师需要牺牲使用任意模拟器件的自由，但是换回了数字电路的简单性和可靠性。通过从模拟到数字之间抽象层次的提高，可以隐藏无需了解的细节来提高设计生产率。

V_{DD}和逻辑电平可以任意选择，但是所有相互通信的逻辑门必须保持兼容的逻辑电平。因此，逻辑门可以按照逻辑系列（logic family）来区分，其中同一逻辑系列的所有门都遵循相同的静态约束。同一逻辑系列中的逻辑门像积木一样组合在一起，使用相同的电源电压和逻辑电平。

20世纪70年代到90年代有四种主流的逻辑系列：TTL（Transistor-Transistor Logic，晶体管-晶体管逻辑），CMOS（Complementary Metal-Oxide-Semiconductor Logic，互补性金属－氧化物－半导体逻辑），LVTTL（Low Voltage TTL，低电压TTL），LVCMOS（Low Voltage CMOS，低电压CMOS）。表1-4比较了它们的逻辑电平。随着电源电压的不断降低，不断分化出新的逻辑系列。附录A.6更加详细地讨论了常见的逻辑系列。

表1-4 5V和3.3V逻辑系列的逻辑电平

逻辑系列	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5(4.75 – 5.25)	0.8	2.0	0.4	2.4
CMOS	5(4.5 – 6)	1.35	3.15	0.33	3.84
LVTTL	3.3(3 – 3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3(3 – 3.6)	0.9	1.8	0.36	2.7

例1.19 逻辑系列兼容性

表1-4中的哪些逻辑系列可以可靠地和其他逻辑系列通信？

解：表1-5列举了逻辑系列之间的兼容性。注意5V的TTL或CMOS逻辑系列可能产生的HIGH信号输出电压为5V。如果5V信号驱动3.3V的LVTTL或LVCMOS逻辑系列输入，可能会损坏接收端，除非接收端特殊设计为“5V兼容”。

表1-5 逻辑系列之间的兼容性

		接收端			
		TTL	CMOS	LVTTL	LVCMOS
驱动源	TTL	兼容	不兼容：$V_{OH} < V_{IH}$	可能兼容①	可能兼容①
	CMOS	兼容	兼容	可能兼容①	可能兼容①
	LVTTL	兼容	不兼容：$V_{OH} < V_{IH}$	兼容	兼容
	LVCMOS	兼容	不兼容：$V_{OH} < V_{IH}$	兼容	兼容

注：① 只要5V高电平不会损害接收端输入。

1.7 CMOS晶体管*

本节和后续带*的章节是可选的。它们对理解本书的核心内容并不重要。

Babbage的分析机由齿轮构成，早期的电子计算机由继电器和真空管构成。现代计算机则由廉价、微型和可靠的晶体管构成。晶体管（Transistor）是一个电子的可控开关：当由电压或者电流施加到控制端时，它将在ON和OFF之间转换。晶体管有两大类：双极晶体管（Bipolar Transistor）和金属－氧化物－半导体场效应晶体管（Metal-Oxide-Semiconductor field effect transistor, MOSFET或MOS）。

1958年，德州仪器（Texas Instrument）的Jack Kilby制造了第一个具有两个晶体管的集成电路。1959年，仙童半导体（Fairchild Semiconductor）申请了在一个硅芯片上连接多个晶体管的专利。在那个年代，每个晶体管的造价是10美元。

半导体制造技术经过三十多年的空前进步，人们已经可以在一个1平方厘米的芯片上集成10亿个MOS晶体管，而每个晶体管的造价已经低于10^{-6}美分。集成度和价格每8年左右改进一个数量级。MOS晶体管现在已经用于构造几乎所有的数字电路系统。本节中，我们将进入电路抽象层以下，来看看如何使用MOS晶体管构造逻辑门。

1.7.1 半导体

MOS晶体管由岩石和沙子中最主要的元素——硅构成。硅（Sillicon，Si）是第四族元素，在化合价外层的4个电子和四个相邻原子紧密连接，形成晶格（lattice）。图1-26(a)中简化显示了二维网格，但网格实际上三维晶格结构。图中的每条线表示一个共价键。在此结构下，

因为硅的电子都被束缚在共价键中，其导电性很弱。但是如果在其中加入少量杂质（掺杂元素，dopant atom），硅的导电性就会大大提高。如果加入第V族元素（例如砷，As），掺杂元素就会有额外的一个电子不受共价键的束缚。这个电子可以在晶格中自由移动，而留下一个带正电的掺杂元素（As^+），如图1-26(b) 所示。由于电子带有负电荷，所以我们称砷为n类掺杂元素。另一方面，如果掺入III族元素（例如硼，B），掺杂元素就将失去一个电子，如图1-26(c) 所示。失去的电子称为空穴（hole）。掺杂元素临近的硅原子可以移动一个电子过来以填充共价键，而产生一个带负电荷的掺杂元素（B^-），并在临近硅原子中产生空穴。类似地，空穴可以在晶格中迁移。空穴缺少一个负电荷，像一个具有正电的粒子。因此，我们称硼为p类掺杂元素。当掺杂浓度发生变化时，硅的导电性可以相差好几个数量级，硅就被称为半导体（semiconductor）。

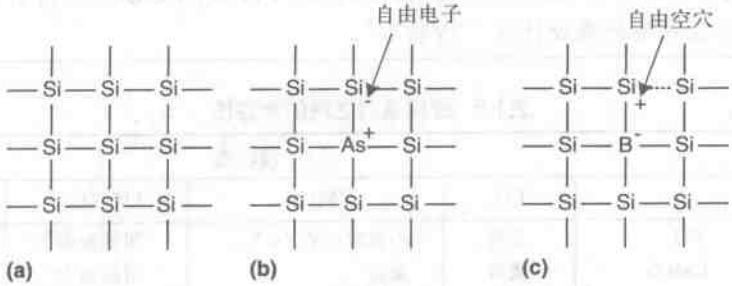


图1-26 硅晶格和掺杂元素

1.7.2 二极管

p类硅和n类硅之间的连接点称为二极管（diode）。n类区域为阳极（anode），p类区域为阴极（cathode），如图1-27所示。当阳极的电压高于阴极时，二极管处于正向偏压，电流从阳极流向阴极。当阳极的电压低于阴极时，二极管处于反向偏压。二极管符号表示了电流仅能沿一个方向流动。

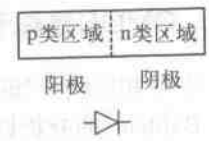


图1-27 基于p-n结的二极管结构和电路符号

1.7.3 电容

电容（capacitor）由夹着绝缘体的两片导体构成。当电压V加到电容一端的导体时，这个导体将积累负电荷Q，而另一端导体将积累正电荷−Q。电容的电容量C（capacitance）是电压和充电电荷之比：C = Q/V。电容量正比于导体的尺寸，反比于导体之间的距离，电路符号如图1-28所示。

电容之所以重要是因为导体的充电或放电需要时间和能量。大电容意味着电路比较慢，而且需要更多的能量。速度和能量问题将在本书中进一步讨论。

1.7.4 nMOS和pMOS晶体管

MOS晶体管由多层导体和绝缘体构成。MOS晶体管在直径15～30cm的硅片（wafer）上制造。制造过程从一个原始硅片开始，包括了掺杂元素的注入，氧化硅膜的生长和金属的淀积等多个步骤。在每个步骤之间，晶片上将形成特定图形使得需要的部分才暴露在外部。由于晶体管的尺寸仅有数微米（10^{-6}m），而且一次可以处理整个硅片，所以一次制作几十亿个晶体管的成本并不高。一旦处理结束，硅片将被切割成很多长方形的部分，每个部

图1-28 电容符号

分包含了成千上万，甚至十亿个晶体管，称为芯片（chip或dice）。这些芯片经过测试后，放置在塑料或陶瓷封装中，并通过金属引脚连接到电路板上。

MOS晶体管中最底层是硅晶片衬底（substrate），最顶上是导电的栅极（gate），中间是由SiO_2构成的绝缘层。过去栅极是由金属构造，因此被称为金属－氧化物－半导体。现在的制造工艺采用多晶硅制造栅极，以避免金属在后续的处理工艺中融化。二氧化硅常用于制造玻璃，在半导体工业中简称为oxide。MOS结构中，金属和半导体衬底之间的极薄二氧化硅绝缘层（dielectric）形成一个电容。

现有两类MOS晶体管：nMOS和pMOS（读作“n-moss”和“p-moss”）。图1-29给出了它们的截面图。nMOS晶体管在p型衬底上由两个与栅极相连的n类型掺杂区域，分别称为源极（source）和漏极（drain）。pMOS晶体管刚刚相反，在n型衬底上构造p型源极和漏极。

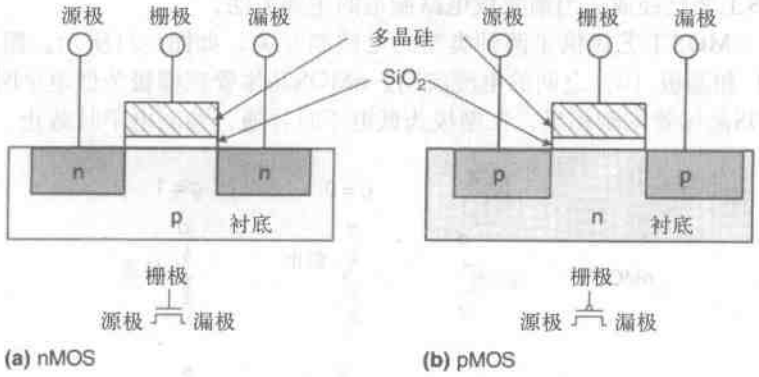


图1-29 nMOS和pMOS晶体管

MOS晶体管特性是一个由电压控制的开关：栅极电压将产生一个电场使得源极和漏极之间的连接处于导通或截止状态。“场效应晶体管”正是来源于这个操作原理。下面让我们继续研究nMOS晶体管的操作过程。

nMOS晶体管的衬底一般都连接到地（系统中最低的电压）。首先考虑当栅极电压为0V的情况，如图1-30(a) 中所示。由于源极或漏极的电压大于0，所以它们和衬底之间的二极管处于反向偏压状态，因此此时源极和漏极之间没有电流，晶体管处于截止状态。接着考虑当栅极电压为V_{DD}的情况，如图1-30(b) 所示。当正电压加在电容的上表面时，将建立一个电场并在上表面吸收正电荷，在下表面吸收负电荷。当电压足够大时，大量的负电荷积聚在栅极下层，使得此区域从p型反转为n型。这个反转区域称为沟道（channel）。此时就有了一个从n型源极经n型沟道到n型漏极之间的通路，电流就可以从源极流到漏极，晶体管就处于导通状态。导通晶体管的栅极电压称为门限电压（threshold voltage, V_T），一般在0.3~0.7V左右。

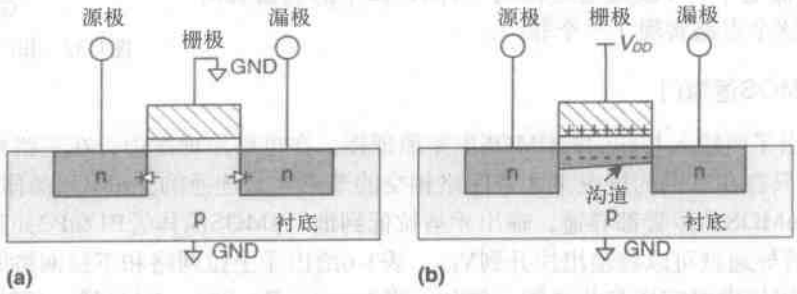


图1-30 nMOS晶体管操作过程

pMOS晶体管的工作方式刚刚相反，也可以从它的电路符号上看出。pMOS晶体管的衬底电压为V_{DD}，当栅极电压为V_{DD}时，处于截止状态，当栅极电压为0V时，沟道反转为p类型处于导通状态。

MOS晶体管并不是完美的开关。对于nMOS晶体管能很好地导通低电平，但导通高电平的能力比较弱：当栅极电压为V_{DD}时，漏极电压在0V到$V_{DD}-V_T$之间。同样的，pMOS晶体管导通高电平的能力很好，但是导通低电平能力较弱。但是，我们仍然可以仅仅利用晶体管较好的模式来构造逻辑门。

nMOS晶体管需要p型衬底，而pMOS晶体管需要n型衬底。为了在同一个晶片上同时构造这两种类型晶体管，制造过程采用p类型晶片，然后在需要pMOS晶体管的地方扩散n类型区域构成阱（well）。这种同时提供两种类型晶体管的工艺称为互补型MOS（complementary MOS，CMOS）。CMOS工艺已经成为当前集成电路制造的主要方法。

总而言之，CMOS工艺提供了两种类型的电控制开关，如图1-31所示。栅极(g)的电压控制了了源极（s）和漏极（d）之间的电流流动。nMOS晶体管在栅极为低电平时截止，为高电平时导通。pMOS晶体管刚刚相反，在栅极为低电平时导通，为高电平时截止。

图1-31 MOS晶体管的开关模式

1.7.5 CMOS非门

图1-32给出了用CMOS构成非门的电路原理图，其中三角形表示地，横线表示电源V_{DD}（这些标号在后续电路原理图中不再标出）。nMOS晶体管N1连接了地和输出Y，pMOS晶体管P1连接了电源和输出Y。两个晶体管的栅极都由输入A控制。

如果A = 0，则N1截止，P1导通，因此Y相当于连接到电源V_{DD}，而与地断开。由于P1可以很好地导通高电平，Y可以被拉升到逻辑1（高电平）。如果A = 1，则N1导通，P1截止，由于N1可以很好导通低电平，Y被拉至逻辑0。与图1-12中的真值表对比，可以看到这个电路实现了一个非门。

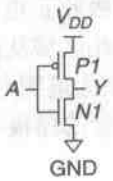


图1-32 非门电路原理图

1.7.6 其他CMOS逻辑门

图1-33给出了两输入与非门的CMOS电路原理图。在电路原理图中，在三路相交的节点上线是连通的，只有在有点的情况下才是四路相交的节点才是连通的。nMOS晶体管N1和N2串联：只有两个nMOS晶体管都导通，输出才被拉低到地。pMOS晶体管P1和P2并联：只要有一个pMOS晶体管导通就可以将输出拉升到V_{DD}。表1-6给出了上拉网络和下拉网络的操作与输出的状态，显示出该电路完成与非功能。例如，当A = 1，B = 0时，N1导通，但N2截止，阻塞

了从输出Y到地之间的通道；同时，P1截止，但P2导通，建立了从电源到输出Y的通道。因此，输出Y被上拉到高电平。

图1-34显示了构造任意反向逻辑门（例如非门，与非门，或非门等）的通用结构。nMOS晶体管可以很好地导通低电平，因此下拉网络采用nMOS晶体管连接输出和地，以将输出完整地下拉到低电平。pMOS晶体管可以很好地导通高电平，因此上拉网络采用pMOS连接输出和电源，以将输出完整地上拉到高电平。当晶体管并联时，只要有一个晶体管导通整个网络就导通。当晶体管串联时，只有所有的晶体管导通网络才能导通。输入上的斜线表示逻辑门可以有多个输入。

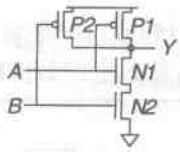


图1-33 两输入与非门电路原理图

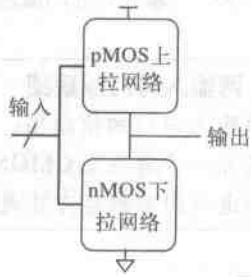


图1-34 反向逻辑门通用结构

表1-6 与非门操作

A	B	下拉网络	上拉网络	Y
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

如果上拉网络和下拉网络同时导通，则会在电源和地址之间产生短路（short circuit）。门的输出电压可能处于禁止区域，而晶体管将消耗大量能量，很可能会使其烧毁。另一方面，如果上拉网络和下拉网络同时截止，输出将即不连接到电源，也不连接到地，处于浮空状态（float）。浮空状态的电压是不确定的。通常不希望有浮空输出，但是在2.6节中可以看到，浮空也可以偶尔使用。

在具有正常功能的逻辑门中，上拉或下拉网络必然有一个导通，另一个截止，这样输出就可以被上拉至高电平或低电平，而不会产生短路或浮空。我们利用传导互补规则来保证这一点，即nMOS采用串联时，pMOS必须使用并联；nMOS使用并联时，pMOS必须使用串联。

例1.20 三输入与非门原理图

使用CMOS晶体管画一个三输入与非门的原理图。

解：与非门只有在所有的输入都为1时，输出才为0。因此，下拉网络必须是三个串联的nMOS晶体管。根据传导互补规则，pMOS晶体管必须采用并联。电路原理图如图1-35所示，读者可以自行验证其功能是否和真值表吻合。

例1.21 两输入或非门原理图

使用CMOS晶体管画一个两输入或非门。

解：或非门只要有一个输入为1，输出就为0。因此，下拉网络应该由两个并联的nMOS晶体管构成。根据传导互补规则，pMOS晶体管应该使用串联方式。电路原理图如图1-36所示。

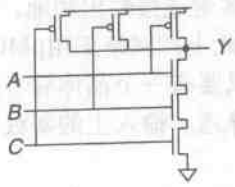


图1-35 三输入与非门原理图

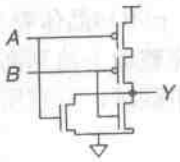


图1-36 两输入或非门原理图

例1.22 两输入与门原理图

画一个两输入与门的原理图。

解：不能用一个单独的CMOS门构成一个与门。但是与非门和非门却很容易构造。因此，使用CMOS构造与门的最佳方法是将与非门的输出连接到非门的输入上，如图1-37所示。

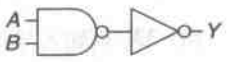


图1-37 两输入与门原理图

1.7.7 传输门

有时，设计者需要一个理想的开关能同时很好地导通0或1。注意到nMOS可以很好导通0，pMOS可以很好地导通1，两者的组合就可以很好导通两种电平。图1-38给出了传输门（transmission gate）的电路符号。由于开关是双向的，所以两边A和B不区分输入或输出。控制信号称为使能（Enable），EN和$\overline{EN}$。当EN = 1且$\overline{EN}$ = 0时，传输门导通（使能），任意逻辑值可以在A和B之间传递。

1.7.8 类nMOS逻辑

在一个N输入或非门中需要N个nMOS晶体管并联和N个pMOS晶体管串联。正如多个电阻串联阻值要大于并联，多个串联的晶体管速度也较慢。此外，由于pMOS晶体管的空穴在晶格中的移动速度要低于电子速度，所以pMOS晶体管的速度要慢于nMOS晶体管。因此，并联的多个nMOS晶体管速度要快于串联的多个pMOS晶体管，尤其当串联的晶体管数目较多时，速度差异更大。

类nMOS逻辑（pseudo-nMOS logic）将上拉网络中的pMOS晶体管替换为单个始终导通的pMOS晶体管，如图1-39所示。这个pMOS晶体管经常称为弱上拉(weak pull-up)，其物理尺寸被设计成满足当所有nMOS晶体管都不导通时，这个弱上拉pMOS晶体管可以维持输出高电平；只要有一个nMOS晶体管导通就能超过这个弱上拉pMOS晶体管，将输出Y下拉到地，而产生逻辑0。

可以利用类nMOS逻辑的特点构造多输入快速或非门。图1-40中给出了一个4输入或非门的例子。类nMOS门很适合构造存储器和逻辑阵列（第5章中介绍）。其缺点在于当输出为低电平时，弱pMOS晶体管和所有的nMOS晶体管都导通，在电源和地之间有短路。短路将持续消耗能量，因此类nMOS逻辑必须谨慎使用。

类nMOS门在20世纪70年代得名，当时的制造工艺仅能生产nMOS晶体管，还不能制造pMOS晶体管，因此使用一个弱nMOS晶体管来实现上拉。

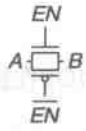


图1-38 传输门

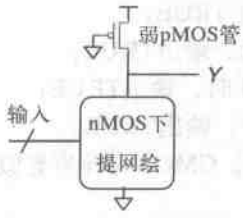


图1-39 类n MOS通用结构图

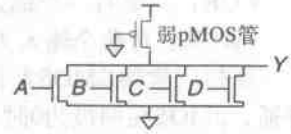


图1-40 或非门

1.8 功耗*

功耗（power consumption）是单位时间内所消耗的能量，在数字系统中非常重要。在手机、笔记本电脑等移动系统中，电池的使用时间取决于功耗。功耗对固定电源供电的系统也很重要，因为电力消耗需要花钱，而且如果功耗过高将导致系统过热。

数字系统包含动态功耗（dynamic power）和静态功耗（static power）。动态功耗是信号在0和1变化过程中电容充电所耗费的能量。静态功耗是信号不发生变化，系统处于空闲状态下的功耗。

逻辑门和连接它们的线都具有电容。将电容C充电到电压V_{DD}所需的能量为CV_{DD}^2。如果电容电压变换的频率为f（每秒变化f次），即在1秒中内要将电容充电$f/2$次，放电$f/2$次。由于放电过程不需要从电源中获取能量，所以，动态功耗为：

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^2 f \tag{1-4}$$

电子系统在空闲的时候也需要一些电流。当晶体管处于截止状态时，仍会有少量漏电流。有一些电路，例如1.7.8节中讨论的类nMOS电路，在电源和地之间始终有通路。这个静态电流I_{DD}称为电源和地之间的漏电流（leakage current）。静态功耗正比于漏电流。

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1-5}$$

例1.23 功耗

某手机的电池容量为6W-h（watt-hour），电源电压为1.2V。假设手机通话时的工作频率为300MHz，芯片中平均电容为10nF，天线需要3W功率。手机不通话时因为所有的信号处理过程停止，所以动态功耗降低到0。但是手机无论是否工作仍然具有40mA的漏电流。请确定不通话情况和连续通话情况下电池的使用时间。

解：静态功耗P_{static} = (0.040A)(1.2V) = 48mW。如果手机不通话，仅有静态功耗，因此其电池使用时间为(6W-h)/(0.048W) = 125h（约5天）。如果手机通话，其动态功耗为P_{dynamic} = $(0.5)(10^{-8}\text{F})(1.2\text{V})^2(3 \times 10^8\text{Hz})$ = 2.16W。加上静态功耗和天线功耗，总的通话功耗为2.16W + 3W + 0.048W = 5.2W。因此电池使用时间为6W. h/5.2W=1.15h。这个例子对手机的实际操作进行了简化，但是可以说明功耗的关键性问题。

1.9 总结和展望

本章介绍了理解和设计复杂系统的基本原则。虽然真实世界是模拟世界，但是数字电路设计师将这些模拟值约束起来，仅仅使用可能信号中的离散子集。特别地，二进制变量仅仅只有两个状态：0和1，也称为FALSE和TRUE，或者LOW和HIGH。逻辑门根据一个或多个二进制输入计算一位二进制输出。一些常见的逻辑门包括：

- NOT：输入为FLASE时，输出TRUE；
- AND：所有输入都为TRUE时，输出TRUE；
- OR：只要有一个输入为TRUE时，输出TRUE；
- XOR：奇数个输入为TRUE时，输出TRUE。

逻辑门常用CMOS晶体管构成。CMOS的行为类似于电子控制开关：nMOS在栅极为1时导通，pMOS在栅极为0时导通。

第2章到第5章中，我们将继续研究数字逻辑。第2章中着重研究输出仅仅依赖于当前输入的组合逻辑（combinational logic）。前面介绍的逻辑门都是组合逻辑的实例。在此章中将学习使用多个门来设计电路，以实现通过真值表或逻辑表达式描述的输入和输出之间的关系。第3章将着重研究时序逻辑（sequential logic），其输出依赖于当前输入和过去的输入。作为基本的时序器件，寄存器（register）可以记住它们以前的输入。基于寄存器和组合逻辑构成的有限状态机（finite state machine）提供了一种强有力的系统化方法来构造复杂系统。我们还将研究数字系统的时序，从而分析系统最快的运行速度。第4章介绍了硬件描述语言（HDL）。硬件描述语言和传统的程序设计语言相关，但是它们用于模拟和构造硬件系统而非软件。现代的大多数数字系统都使用硬件描述语言设计。Verilog和VHDL是两种流行的硬件描述语言。它们在本书中并列介绍。第5章中将研究其他组合逻辑和时序逻辑模块，例如加法器、乘法器和存储器等。

第6章将转移到计算机体系结构。此章介绍了信息产业常用的MIPS处理器。该处理器可以用于消费类电子产品、SGI工作站、电视、网络硬件和无线连接的通信系统中。MIPS体系结构由寄存器和汇编语言指令集定义。此章中将学习如何用汇编语言为MIPS处理器设计程序，这样就可以用处理器自身的语言和它们通信了。

第7章和第8章填补了数字逻辑和计算机体系结构之间的空隙。第7章将研究微结构，即如何将加法器、寄存器等数字模块组合在一起来构成微处理器。此章中，可以学习如何构造自己的MIPS处理器。而且，可以学习到三种不同的微结构，来说明在性能和成本之间的不同折中。处理器性能按照指数方式增长需要更复杂的存储器系统以提供处理器不断提出的数据需求。第8章深入介绍存储器系统体系结构，同时还描述了计算机和键盘、打印机等外部设备进行通信的方法。

习题

1.1 用一张图解释以下领域中出现的至少三个层次的抽象：
a) 生物学家研究细胞的操作；
b) 化学家研究物质的构成。

1.2 用一张图解释以下领域中使用的层次化、模块化和规整化技术：
a) 汽车设计工程师；
b) 管理业务的商人。

1.3 Ben正在盖房子。解释他如何应用层次化、模块化和规整化原则在建房过程中节省时间和金钱。

1.4 一个模拟信号的范围为0～5V。如果测量的精度为±50mV，此模拟信号最多可以传递多少位的信息？

1.5 教室中的旧钟的分钟指针已经折断。
a)如果你可以读取的时钟指针接近于15分钟，时钟传递了多少位的时间信息？
b) 如果你知道现在是否临近中午，则可以再多获得多少位关于时间的附加信息？

1.6 巴比伦人在4000年前提出了基为60的数制系统sexagesimal。一个六十进制的数字可以传递多少位的信息？你应该如何用六十进制方式写4000_{10}这个数字？

1.7 16位可以表示多少个不同的数？

1.8 最大的无符号32位二进制数是多少？

1.9 对于以下三种数制系统，最大的16位二进制数是多少？

a)无符号数

b) 二进制补码数

c) 符号/数值数

1.10 对于以下三种数制系统，最小的16位二进制数是多少？

a) 无符号数

b) 二进制补码数

c) 符号/数值数

1.11 将下列无符号二进制数转化为十进制。

a)1010_2

b) 110110_2

c) 11110000_2

d) 0001100010100111_2

1.12 重复习题1.12，但要转换为十六进制。

1.13 将下列十六进制数转换为十进制

a) $A5_{16}$

b) $3B_{16}$

c) $FFFF_{16}$

d) $D0000000_{16}$

1.14 重复习题1.12，但要转换为无符号二进制数。

1.15 将下列二进制补码数转换为十进制。

a) 1010_2

b) 110110_2

c) 01110000_2

d) 10011111_2

1.16 重复习题1.15，但是这些二进制数采用带符号的原理表示。

1.17 将下列十进制数转换为无符号二进制。

a) 42_{10}

b) 63_{10}

c) 229_{10}

d) 845_{10}

1.18 重复习题1.17，但是要转换为十六进制。

1.19 将下述十进制数转为8位二进制补码，并指出哪些十进制数超出了相应的表示范围。

a) 42_{10}

b) -63_{10}

c) 124_{10}

d) -128_{10}

e) 133_{10}

1.20 重复习题1.19，但是转换为8位带符号的原码表示。

1.21 将下列4位二进制补码数转换为8位二进制补码。
a) 0101_2
b) 1010_2
1.22 重复习题1.21，但二进制数为无符号二进制数。
1.23 基为8的数制系统称为octal。将习题1.17中的数转换为八进制数。
1.24 将下述八进制数转换为二进制、十六进制和十进制。
a) 42_8
b) 63_8
c) 255_8
d) 3047_8
1.25 有多少个5位二进制补码数大于0？有多少小于0呢？这个结果和带符号原码数制的结果有区别吗？
1.26 在一个32位字中有多少个字节？多少个半字节？
1.27 在一个64位字中有多少个字节？
1.28 某DSL调制解调器的数据传输率为768Kb/s？它一分钟内可以接收多少字节？
1.29 硬盘制造商使用MB字节表示10^6字节，使用GB表示10^9字节。在一个50GB字节的硬盘上真正能用于存储音乐的空间有多大？
1.30 不用计算器估计2^{31}有多大？
1.31 Pentium II计算机上的存储器按照位阵列的方式组织。其中有2^8个行和2^9个列。不用计算器估计器存储容量有多少位？
1.32 针对3位无符号数、二进制补码和带符号原码数画出类似于图1-11的图。
1.33 对下列无符号二进制数进行加法。指出在结果为4位情况下，和是否会溢出。
a) $1001_2 + 0100_2$
b) $1101_2 + 1011_2$
1.34 对下列无符号二进制数进行加法。指出在结果为8位情况下，和是否会溢出。
a) $10011001_2 + 01000100_2$
b) $11010010_2 + 10110110_2$
1.35 重复习题1.34，并假设二进制数采用二进制补码表示。
1.36 将下列十进制数转换为6位二进制补码表示，并完成加法操作。指出对于6位结果是否产生溢出。
a) $16_{10} + 9_{10}$
b) $27_{10} + 31_{10}$
c) $-4_{10} + 19_{10}$
d) $3_{10} + -32_{10}$
e) $-16_{10} + -9_{10}$
f) $-27_{10} + -31_{10}$
1.37 对下列无符号十六进制数进行加法操作。指出对于8位结果是否产生溢出。
a) $7_{16} + 9_{16}$
b) $13_{16} + 28_{16}$
c) $AB_{16} + 3E_{16}$
d) $8F_{16} + AD_{16}$
1.38 将下列十进制数转换为5位二进制补码，并进行相减。指出对于5位结果是否产生溢出。
a) $9_{10} - 7_{10}$

b) $12_{10}-15_{10}$

c) $-6_{10}-11_{10}$

d) $4_{10}-8_{10}$

1.39 在N位二进制偏置（biased）数制系统中，对于偏置量B，正数或负数的表示为其值加上B。例如，在偏置量为15的5位数制系统中，0表示为01111，1表示为10000，等等。偏置数制系统可以用于浮点算术中，详细内容在第5章中讨论。考虑一个偏置量为127_{10}的8位偏置数制系统，

a) 二进制数10000010_2对应的十进制数为多少？

b) 表示0的二进制数是多少？

c) 最小负数的值是多少？二进制表示是怎样？

d) 最大正数的值是多少？二进制表示是怎样？

1.40 针对偏置量为3的3位偏置数制系统，画出类似图1-11的数图。（参见习题1.39中对偏置数制系统的定义）。

1.41 在BCD（binary coded decimal）系统中，用4位二进制数表示从0到9的十进制数字。例如，37_{10}表示位00110111_{BCD}。

a) 写出289_{10}的BCD码表示。

b) 将100101010001_{BCD}转换为十进制数。

c) 将01101001_{BCD}转换为二进制数。

d) 解释为什么BCD码也是一种常用的数的表示方法。

1.42 一艘飞碟坠毁在Nebraska的庄稼地里。联邦调查局检查了飞碟残骸，并在一个工程手册中发现了按照Martin数制系统写的等式：325 + 42 = 411。如果等式是正确的，Martin数制系统的基数是多少？

1.43 Ben和Alyssa在争论一个问题。Ben说："所有大于0且能被6整除的正数的二进制表示中必然只有两个1"。Alyssa不同意。她说"不是这样，所有这些数的二进制表示中有奇数个1"。你同意Ben还是Alyssa，或者都不同意？解释你的理由。

1.44 Ben和Alyssa又争论另外一个问题。Ben说："我可以通过将一个数减1，然后将结果各位取反来得到这个数的二进制补码"。Alyssa说："不，我可以从一个数的最低位开始检查每一位，从发现的第一个1开始，将后续的所有位取反来获得这个数的二进制补码"。你同意Ben还是Alyssa，或都同意或都不同意？解释你的理由。

1.45 以你喜欢的语言（C，Java，Perl）写一个程序将二进制转换为十进制。用户应输入无符号二进制数，程序打印出十进制值。

1.46 重复习题1.45，但是将十进制数转换为十六进制。

1.47 重复习题1.45，但是将任意基b_1的数转换为另一基b_2的表示。支持的最大基为16，使用字母表示大于9的数。用户输入b_1和b_2，然后输入基为b_1的数。程序打印出基为b_2的等价数。

1.48 针对下述逻辑门，给出其电路符号、布尔表达式和真值表。

a) 三输入或门

b) 三输入异或门

c) 四输入异或非门

图1-41 三输入表决器

1.49 表决器电路在多于一半的输入为TRUE时输出TRUE。请给出图1-41所示的三输入表决器真值表。

1.50 如图1-42所示的三输入AND-OR(AO)门将在A和B都为TRUE，或者C为TRUE时输出TRUE。请给出其真值表。

1.51 如图1-43所示的三输入OR-AND-INVERT(OAI)门将在C为TRUE，且A或B为TRUE时输出FALSE。其余情况将产生TRUE。请给出其真值表。

图1-42 三输入AND-OR门

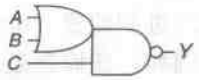


图1-43 三输入OR-AND-INVERT门

1.52 对于两个输入变量，请列出所有16种不同的真值表。对于每种真值表，请给一个简短的名字（例如OR，NAND等）。

1.53 对于N个输入变量，有多少种不同的真值表？

1.54 如果某器件的直流传输特性如图1-44所示，该器件可否作为反相器使用？如果可以，其输入和输出的高低电平（V_{IL}，V_{OL}，V_{IH}和V_{OH}），以及噪声容限（N_{ML}和N_{MH}）分别是多少？如果不能用作反相器，请说明理由。

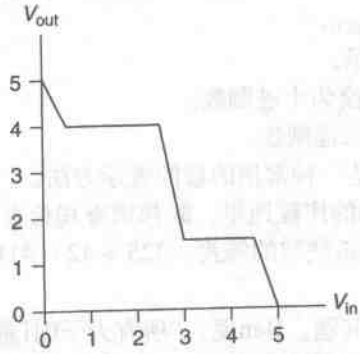


图1-44 直流传输特性

1.55 对图1-45所述部件重复习题1.54。

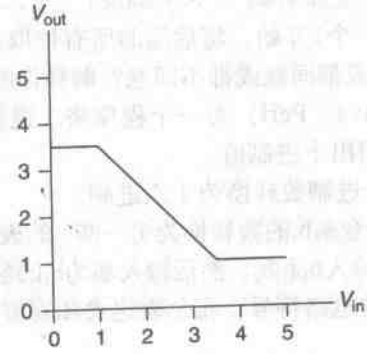


图1-45 直流传输特性

1.56 如果某器件的直流传输特性如图1-46所示，该器件可否作为缓冲器使用？如果可以，其输入和输出的高低电平（V_{IL}，V_{OL}，V_{IH}和V_{OH}），以及噪声容限（N_{ML}和N_{MH}）分别是多少？如果不能用作缓冲器，请说明理由。

1.57 Ben发明了一个缓冲器电路，其直流传输特性如图1-47所示，这个缓冲器能正常工作吗？请解释原因。Ben希望宣称这个电路和LVCMOS LVTTL逻辑兼容。Ben的这个缓冲

器可否正确接收这些逻辑系列的输入？其输出可否正确驱动这些逻辑系列？请解释原因。

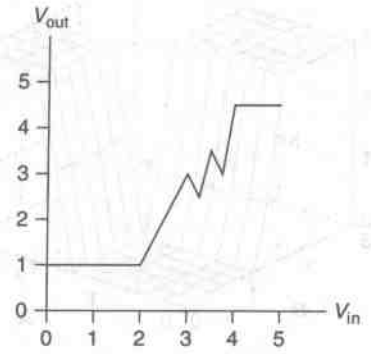


图1-46 直流传输特性

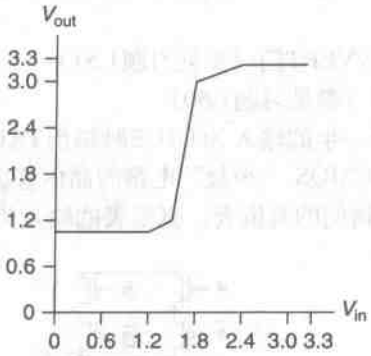


图1-47 Ben的缓冲器直流传输特性

1.58 Ben在黑暗的小巷中碰到了一个两输入门，其转换功能如图1-48所示。其中A，B为输入，Y为输出。

a) 他发现的逻辑门是哪种类型？

b) 其高低逻辑电平大约是多少？

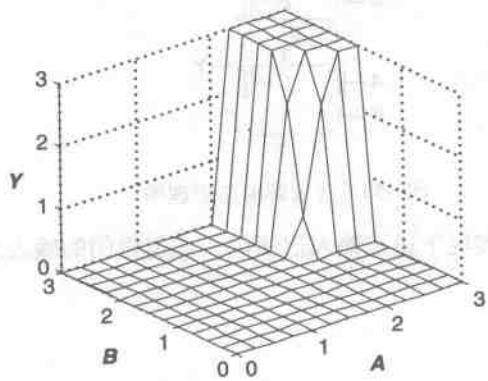


图1-48 两输入直流传输特性

1.59 对图1-49重复习题1.58。

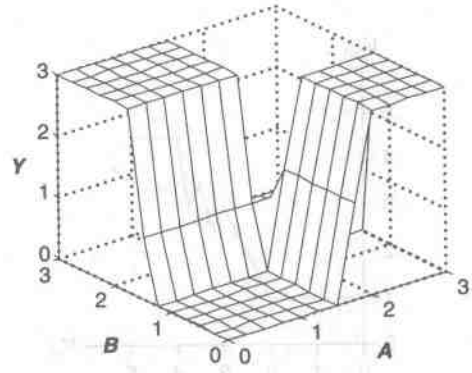


图1-49 两输入直流传输特性

1.60 用最少的晶体管画出下述CMOS逻辑门的晶体管级电路。

a) 四输入NAND门

b) 三输入OR-AND-INVERT门（参见习题1.51）

c) 三输入AND-OR门（参见习题1.50）

1.61 “少数”电路在少于一半的输入为TRUE时输出TRUE，否则将产生FALSE。请用最少的晶体管画出三输入CMOS “少数”电路的晶体管级电路。

1.62 请给出图1-50所示逻辑门的真值表。真值表的输入为A和B。请说明此逻辑功能的名称。

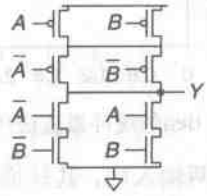


图1-50 待求解电路原理图

1.63 请给出图1-51所示逻辑门的真值表。真值表的输入为A，B和C。

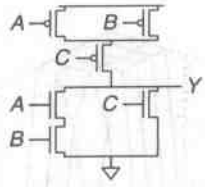


图1-51 待求解电路原理图

1.64 仅使用类nMOS逻辑门实现下述三输入逻辑门。该逻辑门的输入为A，B和C。使用最少的晶体管。

a) 三输入NOR门

b) 三输入NAND门

c) 三输入AND门

1.65 电阻晶体管逻辑（Resistor-Transistor Logic, RTL）使用nMOS晶体管将输出下拉到LOW，

在没有回路连接到地时使用弱电阻将输出上拉到HIGH。使用RTL构成的一个非门如图1-52所示。画出用RTL构成的三输入NOR门。使用最少数目的晶体管。

图1-52 RTL构成的非门

面试问题

1.1 请画出CMOS四输入NOR门的晶体管级电路。

1.2 国王收到了64个金币，但是其中有一个是假的。他命令你找出这个假币。如果你有一个天平，请问需要多少次才能找到哪个比较轻的假币？

1.3 一个教授，一个助教，一个数字设计专业的学生和一个新生需要在黑夜里经过一座摇摇晃晃的桥。这座桥很不稳固，每次只能有两个人通过。他们只有一把火炬，而且桥的跨度太大无法把火炬扔回来，因此必须有人要把火炬拿回来。新生过桥需要1分钟，数字设计专业的学生过桥需要2分钟，助教过桥需要5分钟，教授过桥需要10分钟。所有人都通过此桥的最短时间是多少？

第2章　组合逻辑设计

2.1　引言

在数字电子中，电路是一个处理离散变量的网络。一个电路可以看成一个黑盒，如图2-1所示，其中包括：

- 一个或者多个离散变量输入端（input terminal）；
- 一个或者多个离散变量输出端（output terminal）；
- 功能规范（functional specification）描述输入和输出的关系；
- 时序规范（timing specification）描述当输入改变时输出响应的延迟。

在黑盒的内部，电路由一些节点和组件组成。组件（component）本身又是一个带有输入、输出、功能规范和时序规范的电路。节点（node）是一段导线，它通过电压传递离散变量。节点可分为输入节点、输出节点或内部节点。输入节点接收外部世界的值，输出节点输出值到外部世界。既不是输入节点也不是输出节点的导线称为内部节点。如图2-2所示是一个带有3个元件和6个节点的电路，E1、E2和E3是三个组件，A、B、C是输入节点，Y和Z是输出节点，n1是E1和E2之间的内部节点。

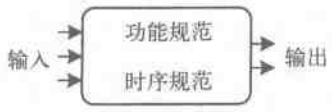


图2-1　将电路作为具有输入、输出和规范的黑盒子

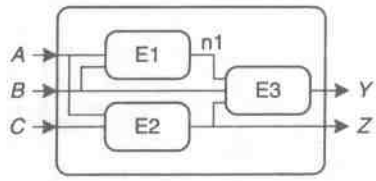
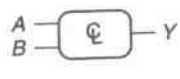


图2-2　组件和节点

数字电路可以分为组合电路（combinational）和时序电路(sequential)。组合电路的输出仅仅取决于输入的值，换句话说，它组合当前输入值来确定输出的值。举个例子，一个逻辑门是一个组合电路。时序电路的输出取决于当前输入值和之前的输入值；换句话说，它取决于输入的序列。组合电路是没有记忆的，但时序电路是有记忆的。这一章重点放在组合电路，第3章将考察时序电路。

组合电路的功能规范表示当前各种输入值情况下所得到的输出值。组合电路的时序规范包括从输入到输出的最大延迟和最小延迟。在这一章，我们首先将介绍函数表达式，在后面部分再回过来学习时序规范。

如图2-3所示，一个组合电路有2个输入和1个输出。图的左边是输入节点A和B，图的右边是输出节点Y。盒子里面的℄符号表示它只能使用组合逻辑实现。在这个例子中，逻辑功能F指定为或逻辑，$Y = (A, B) = F(A + B)$。简单地说，输出Y是2个输入A和B的函数，即，$Y = A$ OR B。



$Y = F(A, B) = A + B$

图2-3　组合逻辑电路

图2-4给出了图2-3中组合逻辑电路的两种可能实现（implementation）。在这本书中，我们将多次看到，一个功能通常存在许多种实现。可以根据配置和设计约束选择给定的模块来实

现组合电路。这些约束通常包括面积、速度、功耗和设计时间。

图2-5是一个有多个输出的组合电路。这个特殊的组合电路称为全加器（full adder），我们将在5.2.1节中再次遇到。图中的两个式子是根据输入A、B和C_{in}确定输出S和C_{out}的函数。

为了简化画图，我们经常用一根信号线表示总线（bus），即有多个信号构成的一组线。其中总线上有一个斜线，并在旁边标注一个数字，表示在总线上的信号数量。例如，图2-6(a)中的组合逻辑块有3个输入和2个输出。如果总线的位数不重要或者在上下文中很明显时，斜线旁可以不标识数字。图2-6(b)中，两个组合逻辑块有任意数量的输出，一个逻辑块的输出作为下一个逻辑块的输入。

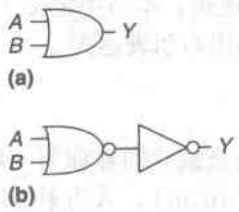


图2-4 或逻辑的两种实现

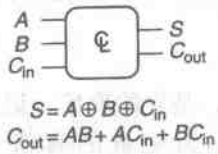


图2-5 多输出组合逻辑

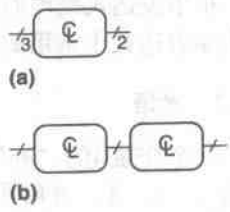


图2-6 斜线表示多个信号

组合电路的构成规则告诉我们如何通过较小的组合电路组件构造一个大的组合电路。如果一个电路由互相连接的电路组件构成，在满足以下条件时，它就是组合电路。

- 每一个电路组件本身都是组合电路；
- 每一个电路节点或者是一个电路的输入，或者仅仅连接到一个电路组件的一个输出端口；
- 电路不能包含回路：经过电路的每条路径最多只能经过每个电路节点一次。

例2.1 组合电路

根据组合电路的构成规则，图2-7所示电路中哪些是组合电路？

解：电路(a)是组合电路。它由2个组合电路组件构成（反向器I1和I2）。它有3个节点：n1、n2和n3。n1是整个电路和I1的输入，n2是内部节点，是I1的输出和I2的输入，n3是电路和I3的输出。(b)不是组合电路，因为它存在回路：异或门的输出反馈到一个输入端，从n4开始通过异或门到n5再返回到n4是一个回路。(c)是组合电路。(d)不是组合电路，因为节点n6同时连接I3和I4的输出端。(e)是组合电路，表示了两个组合电路连接而构成一个大的组合电路。(f)不符合组合电路的组成规则，因为它有一个回路通过2个组件。它是否为一个组合电路需要视组件的功能而定。

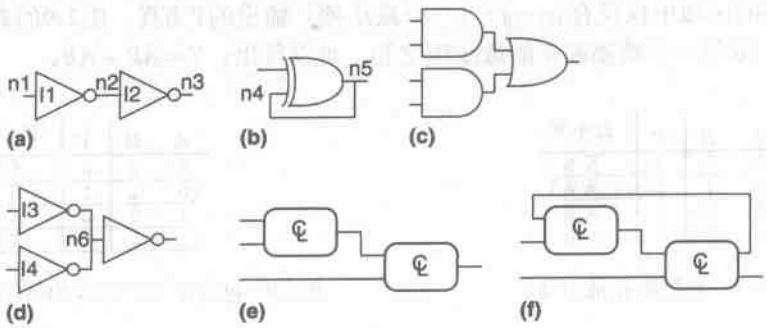


图2-7 示例电路

像微处理器这样的大规模电路非常复杂，我们可以用第一章介绍的原理来控制复杂性。应用抽象和模块化原则将电路视为一个明确定义了接口和功能的黑匣子。应用层次化原则由较小的电路构建复杂电路。组合电路的构成规则应用了约束原理。

一个组合电路的功能规范通常描述为真值表或者布尔表达式。在后续章节中，我们将介绍如何通过真值表得到布尔表达式，如何使用布尔代数和卡诺图来化简表达式，并说明如何通过逻辑门实现逻辑表达式和分析组合逻辑电路的速度。

2.2 布尔表达式

布尔表达式处理的变量是“真”或“假”，所以它很适合描述数字逻辑。本节中定义了一些布尔表达式中常用的术语，然后介绍如何为给定真值表的逻辑函数写出布尔表达式。

2.2.1 术语

一个变量A的“补”（complement），是它的取反，记为$\overline{A}$。一个变量或它的补称为一项。比如，A，$\overline{A}$，B和$\overline{B}$是项。我们称A为变量的真值形式（true form），$\overline{A}$为补形式（complementary form）。“真值形式”不表示A为真，仅仅是A的上面没有一条线。

一个或者多个项的“与”称为乘积项（product）或者蕴涵（implicant）。$\overline{A}B$，$A\overline{B}\overline{C}$和$B$都是3个变量函数的蕴涵。最小项（minterm）是一个包含全部输入变量的乘积项。$A\overline{B}\overline{C}$是输入为$A$、$B$、$C$的3变量函数的一个最小项，但是$\overline{A}B$不是最小项，因为它不包含$C$。同样地，一个或者多个项的“或”称为求和项（sum）。一个最大项（maxterm）是包括了全部输入项的和。$A+\overline{B}+C$是输入为A、B、C的3变量函数的一个最大项。

在解释布尔表达式时，其顺序很重要。布尔表达式$Y=A+BC$是表示$Y=(A \text{ OR } B) \text{ AND } C$，还是表示$Y=A \text{ OR } (B \text{ AND } C)$？在布尔表达式中，“补”的优先级最高，接着是“与”，最后是“或”。对于一个普通的等式，乘积在求和之前执行。所以，此等式读为$Y=A \text{ OR } (B \text{ AND } C)$。式（2-1）给出了解释顺序的另外一个例子。

$$\overline{A}B+BC\overline{D}=((\overline{A})B)+(BC(\overline{D})) \tag{2-1}$$

2.2.2 与或式

有N个输入的真值表包含2^N个行，每一行对应了输入变量的一组可能取值。真值表中的每一行都与一个为真的最小项相关联。图2-8是一个有2个输入A和B的真值表，给出了每一行对应的最小项。比如，第一行的最小项是$\overline{A}\overline{B}$，这是因为当$A=0$，$B=0$时，$\overline{A}\overline{B}$取值为真。

可以用输出Y为真的所有最小项之和的形式写出任意一个真值表的布尔表达式。比如，在图2-8中圈起来的区域中仅仅存在一行（一个最小项）输出的Y为真。图2-9的真值表中有多行Y的输出为真。取每一个被圈起来的最小项之和，可以得出：$Y=\overline{A}B+AB$。

A	B	Y	最小项
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}\,B$
1	0	0	$A\,\overline{B}$
1	1	0	$A\,B$

图2-8 真值表和最小项

A	B	Y	最小项
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}\,B$
1	0	0	$A\,\overline{B}$
1	1	1	$A\,B$

图2-9 包含了多个为真的最小项的真值表

因为这种形式是由若干积（“与”构成了最小项）的和（“或”）构成，称为一个函数的与

或范式（sum-of-products canonical form）。虽然有多种形式表示同一个函数，比如图2-9的真值表可以写为$Y = B\bar{A} + BA$，但可以将它们按照在真值表中出现的顺序进行排序，因此同一个真值表总是能写出唯一的布尔表达式。

例2.2 与或式

Ben正在野炊，如果天气下雨或者那儿有蚂蚁的话，Ben将不能享受到野炊的快乐。设计一个电路，当其输出为真表示Ben可以享受野炊。

解：首先定义输入和输出。输入为A和R，它们分别表示有蚂蚁和下雨，有蚂蚁时A为真，没有蚂蚁时A为假。同样，下雨时R为真，不下雨时R为假。输出为E，表示Ben享受野炊。如果Ben享受野炊，E为真，如果Ben没能享受野炊，E为假。图2-10表示Ben野炊经历的真值表。

使用与或式，我们写出的等式如下：$E = \bar{A}R$。我们可以用2个反向器和2个与门来实现等式，如图2-11（a）所示。读者可能发现这个真值表是1.5.5节中出现的或非函数：$E = A$ NOR $R = \overline{A+R}$。图2-11（b）表示或非操作。在2.3节中，我们将证明$\bar{A}R$和$\overline{A+R}$这两个等式是等价的。

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

图2-10 Ben的真值表

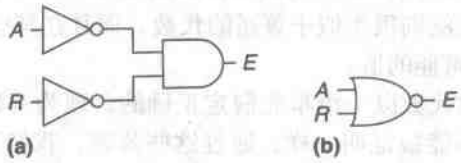


图2-11 Ben的电路

对具有任意多变量的真值表可以用与或式写出唯一的布尔表达式。图2-12是一个三输入真值表，逻辑函数的与或式如下：

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C \tag{2-2}$$

但是，与或式并不能一定产生最简等式。在2.3节中，我们将介绍如何用较少的项写出同样的函数。

2.2.3 或与式

布尔函数的第二种表达式是或与范式（canonical form）。真值表的每一行对应了为假的一个最大项。比如，二输入真值表的第一行的最大项为$(A + B)$，这是因为当$A = 0$，$B = 0$时，$(A + B)$为假。我们可以直接通过将真值表中每一个输出为假的最大项相与而得到电路的布尔表达式。

例2.3 或与式

为图2-13中的真值表写出或与表达式。

解：真值表中有2个行输出为假，所以函数可以写成或与式：$Y = (A + B)(\bar{A} + B)$。第一个最大项为$(A + B)$，在$A = 0$且$B = 0$时，任何值与0相与都等于0，保证了$Y = 0$。同样，第二个最大项为$(\bar{A} + B)$，在$A = 1$且$B = 0$时，保证了$Y = 0$。图2-13和图2-9是相同的真值表，表明可以用多种方法写出同一个函数。

同样地，通过圈出三个为0的行，图2-10表示Ben野炊的布尔表达式可以写成或与式，得到$E = (A + \bar{R})(\bar{A} + R)(\bar{A} + \bar{R})$。它比与或式的$E = \bar{A}R$要复杂，但这两个等式在逻辑上是等价的。

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

图2-12 一个三输入真值表

A	B	Y	最大项
0	0	0	A + B
0	1	1	A + $\overline{B}$
1	0	0	$\overline{A}$ + B
1	1	1	$\overline{A}$ + $\overline{B}$

图2-13 包含了多个假值最大项的真值表

当真值表中仅有少数行输出为真时，与或式产成的等式较短。当真值表中仅有少数行输出为假时，或与式比较简单。

2.3 布尔代数

前面的章节中学习了通过给定的真值表如何写出布尔表达式。但是，表达式不能导出一组最简逻辑门。就像用代数来化简数学等式一样，也可以用布尔代数来化简布尔表达式。布尔代数的法则很类似于普通的代数，而且在某些情形下更加简单，这是因为变量仅仅只有0和1这两种可能的值。

布尔代数以一组事先假定正确的公理为基础。公理是不用证明的，就某种意义上说，就像定义不能被证明一样。通过这些公理，我们可以证明布尔代数的所有定理。这些定理有较大的实用意义，它们可以指导我们如何去化简逻辑来生成较小而且成本更低的电路。

布尔代数的公理和定理都服从对偶原理。如果符号0和1互换，操作符·(AND)和+(OR)互换，表达式将依然正确。我们用上标(′)来表示对偶式。

2.3.1 公理

表2-1给出了布尔代数的公理。这5个公理和它们的对偶式定义了布尔变量，以及非、或、与的含义。公理A1表示变量要么是1，要么是0。公理对偶式A1表示变量要么是0，要么是1。A1和A1′告诉我们只能对布尔变量(二进制量)0和1进行操作。公理A2和A2′定义了非操作。公理A3到A5定义与操作；它们的对偶式A3′到A5′定义了或操作。

表2-1 布尔代数的公理

	公 理	对偶公理	名 称
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	二进制量
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \cdot 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4′	$1 \cdot 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

2.3.2 单变量定理

表2-2中的定理T1到T5描述了如何化简包含一个变量的等式。

同一性定理T1，表示对于任何布尔变量B，B AND 1 = B，它的对偶式表示B OR 0 = B。如图2-14所示的硬件中，T1的意思是在二输入与门中如果有一个输入总是为1，可以删除与门，用连接输入变量B的一条导线代替与门。同样，T1'的意思是在二输入或门中如果有一个输入

总是为0，可以用连接输入变量B的一条导线代替或门。一般说来，逻辑门要花费成本、功耗和延迟，用导线来代替门电路是有利的。

表2-2 单变量的布尔代数定理

	公 理		对偶公理	名 称
T1	$B \cdot 1 = B$	T1′	$B + 0 = B$	同一性定理
T2	$B \cdot 0 = 0$	T2′	$B + 1 = 1$	零元定理
T3	$B \cdot B = B$	T3′	$B + B = B$	重叠定理
T4			$\overline{\overline{B}} = B$	回旋定理
T5	$B \cdot \overline{B} = 0$	T5′	$B + \overline{B} = 1$	互补定理

零元定理T2，表示B和0相与总是等于0。所以，0被称为与操作的零元，因为它使任何其他输入的影响无效。对偶式表明，B和1相或总是等于1。所以，1是或操作的零元。在硬件电路中，如图2-15所示，如果一个与门的输入是0，我们可以用连接低电平（0）的一条导线代替与门。同样，如果一个或门的输入是1，我们可以用连接高电平（1）的一条导线代替或门。

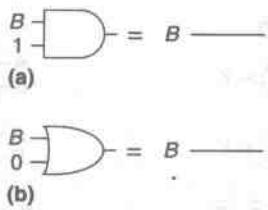


图2-14 同一性定理的硬件解释

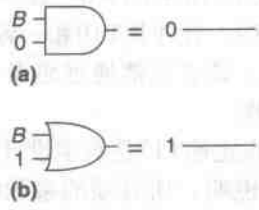


图2-15 零元定理的硬件解释

重叠定理T3，表示变量和它自身相与就等于它本身。同样，一个变量和它自身相或等于它本身。从拉丁语语源给出了定理的名字：同一的和强有力的。这个操作返回和输入相同的值。图2-16所示，重叠定理也允许用一根导线来代替门。

回旋定理T4，说明对一个变量进行2次求补可以得到原来的变量。在数字电子学中，两次错误将产生一个正确结果。串联的2个反向器在逻辑上等效于一条导线，如图2-17所示。T4的对偶式是它自身。

互补定理T5，表示一个变量和它自己的补相与的结果是0（因为它们中必然有一个值为0）。同时，对偶式表明一个变量与它自己的补相或的结果是1（因为它们中必然有一个值为1）。

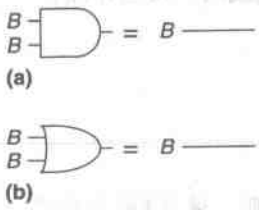


图2-16 重叠定理的硬件解释

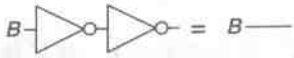


图2-17 回旋定理的硬件解释

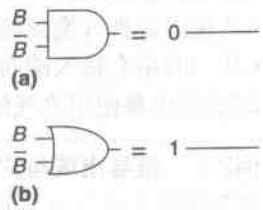


图2-18 互补定理的硬件解释

2.3.3 多变量定理

表2-3中的定理T6～T12描述了如何化简包含多个变量的布尔表达式。

表2-3 含多个变量的布尔定理

	公 理		对偶公理	名 称
T6	$B \cdot C = C \cdot B$	T6′	$B + C = C + B$	交换律
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	T7′	$(B + C) + D = B + (C + D)$	结合律
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	T8′	$(B + C) \cdot (B + D) = B + (C \cdot D)$	分配律
T9	$B \cdot (B + C) = B$	T9′	$B + (B \cdot C) = B$	吸收律
T10	$(B \cdot C) + (B \cdot \overline{C}) = B$	T10′	$(B + C) \cdot (B + \overline{C}) = B$	合并律
T11	$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D)$ $= B \cdot C + \overline{B} \cdot D$	T11′	$(B + C) \cdot (\overline{B} + D) \cdot (C + D)$ $= (B + C) \cdot (\overline{B} + D)$	一致律
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \ldots} = (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2})$	德·摩根定理

交换律T6和结合律T7与传统代数相同。交换律表明“与”或者“或”函数的输入顺序不影响输出的值。结合律表明特定输入的分组不影响输出结果的值。

分配律T8与传统代数相同。但是它的对偶式T8'不同。在T8中“与”的分配高于“或”，在T8'中“或”的分配高于“与”。在传统代数中，乘法分配高于加法但是加法分配不高于乘法，即$(B + C) \times (B + D) \neq B + (C \times D)$。

吸收律T9、合并律T10和一致律T11允许消除冗余变量。读者应能通过思考自己证明这些定理的正确性。

德·摩根定理T12是数字设计中非常有力的工具。定理说明，所有项的乘积的补等于每个项各自取补后相或。同样，所有项求或的补等于每个项各自取补后相与。

根据德·摩根定理，一个与非门等效于一个带反向器输入的或门。同样，一个或非门等效于一个带反向输入的与门。每个函数的这两种表达式称为对偶式。它们是逻辑等效的，可以相互替换。

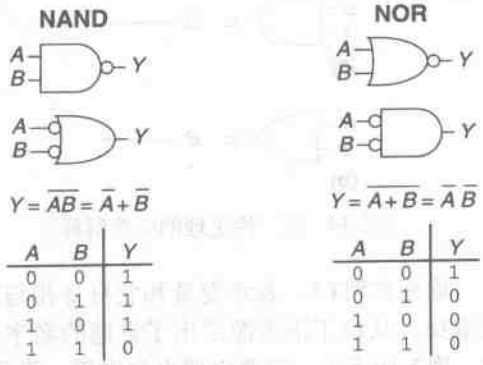


图2-19 德·摩根定理的等价电路

反向器也可称为气泡（bubble）。你可以直观地设想，推一个气泡通过一个门使它从门一边输入，在另一边输出，可以将与门替换成或门，反之亦然。例如，图2-19中的与非门是在输出端包含气泡的与门。推一个气泡到左边会使得每一个输入端带有气泡，并将与门变成或门。下面是推气泡的规则：

- 向后推输出端的气泡或者向前推输入端的气泡，需要将与门换成或门，反之亦然；
- 从输出端推气泡到输入端，把气泡放置在门的输入端；
- 从门的所有输入端向前推气泡，把气泡放置在门的输出端。

2.5.2节中将使用推气泡的方法分析电路。

例2.4 推导出或与式

图2-20给出布尔函数Y的真值表，以及它的反$\overline{Y}$。使用德·摩根定理，通过$\overline{Y}$的与或式，推导出Y的或与式。

解：图2-21打圈的部分表示了$\overline{Y}$中的最小项，$\overline{Y}$的与或式为：

$$\overline{Y} = \overline{A}\overline{B} + \overline{A}B \tag{2-3}$$

等式两边同时取反，并应用德·摩根定理2次，我们可以得到

$$\overline{\overline{Y}} = Y = \overline{\overline{A}B + A\overline{B}} = (\overline{\overline{A}B})(\overline{A\overline{B}}) = (A+B)(A+\overline{B}) \tag{2-4}$$

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

图2-20 包含Y和$\overline{Y}$的真值表

A	B	Y	$\overline{Y}$	最小项
0	0	0	1	$\overline{A}\,\overline{B}$
0	1	0	1	$\overline{A}B$
1	0	1	0	$A\overline{B}$
1	1	1	0	AB

图2-21 包含了$\overline{Y}$最小项的真值表

2.3.4 定理的统一证明方法

好奇的读者可能想知道如何证明定理的正确性。在布尔代数中，证明有限变量定理的方法很简单：只要证明针对变量的所有可能取值定理都保持正确。这种方法叫做完全归纳法，可以通过一个真值表完成证明。

例2.5 证明一致性定理

证明表2-3中的一致性定理T11。

解：针对B，C和D的8种组合检查等式的两边。图2-22中的真值表列出了它们的组合。因为在所有可能中均有$BC + \overline{B}D + CD = BC + \overline{B}D$，定理得证。

2.3.5 等式化简

布尔代数定理有助于化简布尔表达式。例如，考虑真值表2-9的与或式：$Y = \overline{A}\overline{B} + A\overline{B}$。应用定理T10，等式可以化简为$Y = \overline{B}$。这在真值表中是显而易见的。通常情况下，针对较复杂的式子需要多个步骤分步化简。

B	C	D	$BC+\overline{B}D+CD$	$BC+\overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

图2-22 证明一致性定理T11的真值表

化简与或式的基本原则是使用关系$PA + P\overline{A} = P$来合并项，其中P可以是任意蕴涵项。一个等式可以化简到什么地步呢？我们称使用了最少蕴涵项的等式为最小的与或式。对于蕴涵项数量一样的多个等式，带有变量最少的为最小与或式。

如果蕴涵式不能和其他任意的蕴涵式合并生成一个含有变量更少的形式。那么此蕴涵式称为主蕴涵式。最小等式中的蕴涵式必须都是主蕴涵式。否则，就可以合并来减少变量数量。

例2.6 化简等式

化简表达式2.2：$\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$

解：我们从原等式开始，如表2-4所示一步一步地运用布尔定理。

在这个起点上我们已经完全化简等式了吗？让我们进一步仔细分析。通过原等式，最小项$\overline{A}\overline{B}\overline{C}$和$A\overline{B}\overline{C}$仅仅在变量$A$上面不同，于是我们可以合并最小项为$\overline{B}\overline{C}$。继续观察原等式，我们注意到最后2个最小项$A\overline{B}\overline{C}$和$A\overline{B}C$同样只有一个字母不同（$C$和$\overline{C}$）。因此，应用相同的方法可以合并这两个最小项为$A\overline{B}$。我们称蕴涵项$\overline{B}\overline{C}$和$A\overline{B}$共同分享了最小项$A\overline{B}\overline{C}$。

这样，我们是仅仅化简这两个最小项中的一个，还是都进行化简？应用重叠律，可以复制我们想要的项：$B = B + B + B + B + \cdots$。运用这个原理，我们可以完全化简等式为2个主蕴涵式，$\overline{B}\overline{C} + A\overline{B}$，如表2-5所示。

表2-4 等式化简

步骤	等式	定理
	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	
1	$\bar{B}\bar{C}(\bar{A} + A) + A\bar{B}C$	T8: 分配律
2	$\bar{B}\bar{C}(1) + A\bar{B}C$	T5: 互补定理
3	$\bar{B}\bar{C} + A\bar{B}C$	T1: 同一性定理

表2-5 进一步等式化简

步骤	等式	定理
	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	
1	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	T3: 重叠定理
2	$\bar{B}\bar{C}(\bar{A} + A) + A\bar{B}(\bar{C} + C)$	T8: 分配律
3	$\bar{B}\bar{C}(1) + A\bar{B}(1)$	T5: 互补定理
4	$\bar{B}\bar{C} + A\bar{B}$	T1: 同一性定理

展开一个蕴涵项（比如，将AB变成$ABC + AB\bar{C}$）是化简表达式的常用技巧（虽然它有一点违反直觉）。通过展开一个蕴涵式，可以重复一个展开的最小项和其他的最小项合并。

读者可能注意到完全使用布尔代数定理来化简布尔表达式可能带来很多错误。2.7节介绍卡诺图化简方法会使处理简单一些。

为什么要花费精力化简逻辑表达式呢？化简减少了物理实现逻辑功能所需门的数量，从而使得实现电路更简单、成本更低、处理速度可能还更快。下一节将介绍如何用逻辑门实现布尔表达式。

2.4 从逻辑到门

电路原理图（schematic）描述了数字电路的内部组件及其相互连接。例如图2-23中的原理图表示了前面所讨论过逻辑函数式（2-2）的一种可能硬件实现。

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

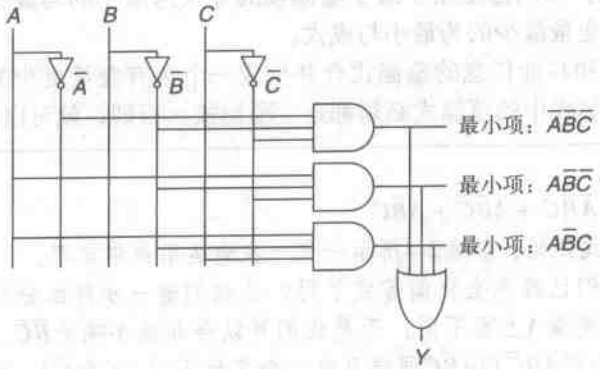


图2-23 $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$的电路原理图

原理图需要遵循统一的风格，使得它们更加易于阅读和检查错误，通常遵循以下的风格：

- 输入在原理图的左边或者顶部；
- 输出在原理图的右边或者底部；

- 无论何时，门必须从左输入、从右输出；
- 最好使用直线而不是使用有很多拐角的线（交错的线需要考虑如何走线，浪费精力）；
- 走线总是在T交叉点连接；
- 在两条线交叉的地方有一个点，表示它们之间有连接；
- 在两条线交叉的地方没有点，表示它们没有连接。

图2-24图示了最后三条准则。

图2-24 线的连接

任何布尔表达式的与或式可以用系统的方法画成与图2-23相似的原理图。按列画出输入，如果有需要的话在相邻列之间放置反向器提供输入信号的补。画一列与门实现每个最小项。为每一个输出画一个或门来连接和输出有关的最小项。因为反向器、与门和或门按照系统的风格排列，这种设计称为可编程逻辑阵列（Programmable Logic Array, PLA），将在5.6节讨论。

图2-25给出了例2.6中布尔代数化简后的实现。与图2-23相比，简化后的电路所需硬件明显减少。而且因其逻辑门的输入更少，简化电路速度可能会更快。

可以通过利用反向门替代单独反向器的方法进一步减少门的数量。注意到$B\bar{C}$是一个带反向输入的与门。图2-26给出了消除C上反向器的优化实现原理图。利用德·摩根定律，带反向输入的与门等效于一个或非门。基于不同的实现技术，使用更少的门或者某几种适合特定工艺的门会更加便宜。例如在CMOS实现中与非门和或非门优于与门和或门。

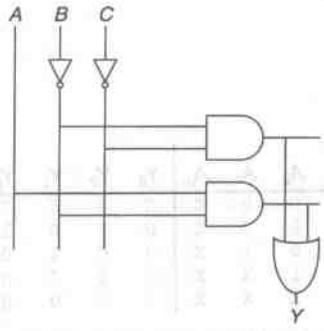


图2-25 $Y = \bar{B}\bar{C} + A\bar{B}$的原理图

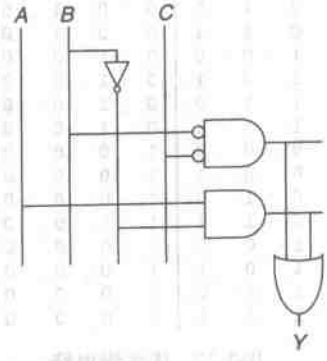


图2-26 使用更少门的原理图

许多电路有多个输出。针对每一个输出分别计算输入的布尔函数。我们可以分别写出每个输出的真值表。但更方便的方法是在一个真值表中写出所有输出，并画出带所有输出的原理图。

例2.7 多输出电路

院长、系主任、助教和寝室室长有时都会使用礼堂。不幸的是，他们偶尔会发生冲突。比如系主任准备和一些理事正在礼堂召开一个会议，同一时间，寝室室长正在举行狂欢会。Alyssa要设计一个会议室的预定系统。

这个系统有4个输入量A_3, …, A_0，和4个输出量Y_3, …, Y_0。这些信号也可以写成$A_{3:0}$和$Y_{3:0}$。当用户要求预定明天的礼堂时将其对应输入设置为真。系统的输出中最多有一个为真，从而

将礼堂的使用权给优先级别最高的用户。院长在系统中的优先级别最高，系主任、助教和室长的优先级依次递减。

为系统写出真值表和布尔表达式，画电路来实现这个功能。

解：这个功能称为4输入优先级电路。它的电路符号和真值表如图2-27所示。

可以写出每个输出的与或式，并使用布尔代数来化简等式。根据函数表达式（和真值表）可以很清楚地检查化简后的等式：当A_3有效时，Y_3为真，于是$Y_3 = A_3$。当A_2有效且A_3无效时，Y_2为真，所以，$Y_2 = \bar{A}_3 A_2$。如果A_1有效且没有更高优先级的信号有效时，Y_1为真：$Y_1 = \bar{A}_3 \bar{A}_2 A_1$。当$A_0$有效且没有其他信号有效时，$Y_0$为真，所以$Y_0 = \bar{A}_3 \bar{A}_2 \bar{A}_1 A_0$。图2-28给出了原理图。一个有经验的设计师经常通过观察来实现逻辑电路。对于给定的设计规范，直接把文字变成布尔表达式，再把表达式变成门电路。

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

图2-27 优先级电路

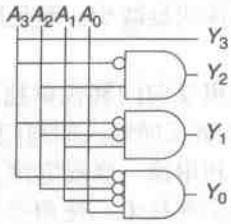


图2-28 优先级电路原理图

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

图2-29 带有无关项（X）的优先级电路真值表

注意到优先级电路中如果A_3为真，输出不用考虑其他输入量。我们用符号X表示输出不需要考虑的输入。图2-29中带无关项的4输入优先级电路真值表变得更小了。这个真值表通过无关项X可以很容易地读出布尔表达式的与或式。无关项也能出现在真值表中的输出项，这将在2.7.3节中讨论。

2.5 多级组合逻辑

与或式称为二级逻辑，因为它在第一级与门中连接所有的输入信号，然后再连接到第一级的或门。设计师们经常用多于两个级别的逻辑门建立电路。这些多级组合电路使用的硬件比两级组合电路更少。推气泡方法在分析和设计多级电路中尤其有帮助。

2.5.1 减少硬件

当使用两级逻辑时，一些逻辑函数需要大量的硬件。一个典型的例子是多输入的异或门函数。例如，采用我们所学的方法用两级逻辑建立一个3输入的异或门电路。

注意到对于一个N输入异或门，当奇数个输入为真时输出为真。图2-30表示一个3输入异或门真值表，其中带圈的行输出为真。通过真值表，可以读出布尔表达式的与或式形式，如式（2-5）所示。不幸的是，没有办法来化简这个等式为较小的蕴涵项。

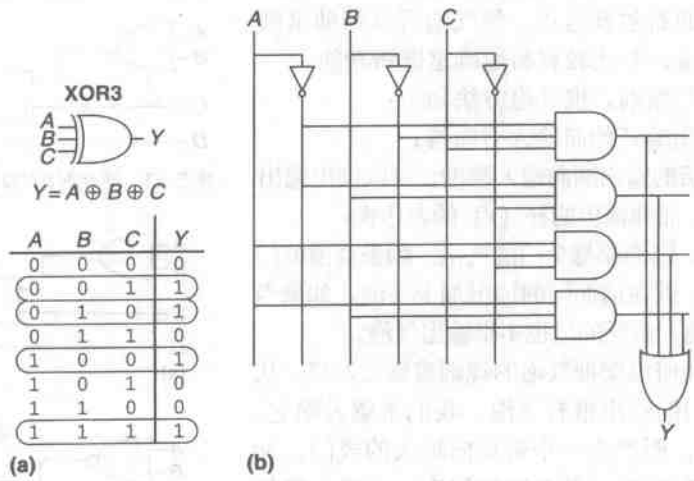


图2-30 3输入异或门：(a) 功能规范；(b) 两级逻辑实现

$$Y = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \tag{2-5}$$

另一方面，$A \oplus B \oplus C = (A \oplus B) \oplus C$（如果有疑问的话，可以通过归纳法证明）。因此，三输入异或门可以通过串连两个二输入异或门构造，如图2-31所示。

同样，一个八输入异或门的两级与或式逻辑实现需要128个8位输入的与门和一个128位输入的或门。一个更好的选择是使用二输入异或门树构成，如图2-32所示。

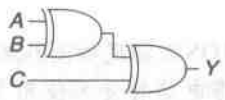


图2-31 使用两个二输入异或门构造一个三输入异或门

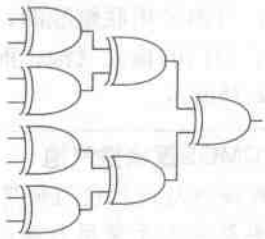


图2-32 用7个二输入异或门构造一个八输入异或门

选择最好的多级结构来实现特定逻辑函数不是一个简单的过程。而且，最好的标准有多重含义：门的数量最少、速度最快、设计时间最短、花费最少或功耗最低等。在第5章中将看到在某个工艺中最好的电路在另一种工艺中并不是最好的。比如，我们经常使用与门和或门，但是在CMOS电路中与非门和或非门会更加有效率。通过积累一些设计经验，读者对大多数的电路可以通过观察的方法来设计一个好的多层电路。通过学习本书后续部分的电路实例，

读者可以获得一些经验。在学习过程中，读者将探讨各种不同的设计策略并且权衡选择。计算机辅助设计（CAD）工具通常也可以有效地发现更多可能的多级设计，并且按照给定的限制条件寻找一个最好的设计。

2.5.2 推气泡

回忆1.7.6节中介绍的CMOS电路，其与非门和或非门的实现优于与门和或门。但是从带有与非门和或非门多级电路读出布尔表达式，可能会相当头痛。难以立刻通过观察方法得到图2-33中多级电路的布尔表达式。推气泡可以帮助重画这些电路，消除气泡，并比较容易地确定逻辑功能。

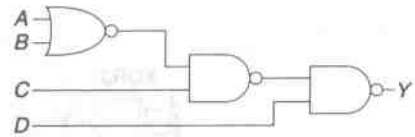


图2-33 使用NAND和NOR的多级逻辑

根据2.3.3节中的原则，推气泡方法如下：

1) 从电路的输出端开始向输入方向推；

2) 将气泡从最后的输出端向输入端推，可以读出输出（Y）的布尔表达式，而非输出的补（$\bar{Y}$）的表达式；

3) 继续向后推，以消除每个门的气泡。如果当前的门有一个输入的气泡，则在前面门的输出加上气泡。如果当前的门不带输入气泡，前面的门也不带输出气泡。

图2-34显示了如何根据推气泡的规则重画图2-33。从输出Y开始，与非门的输出带有气泡，我们希望去除它。将输出气泡向后推，而产生一个带反向输入的或门，如图2-34所示。继续向左看，最右边的门有一个输入的气泡，从而可以删除中间与非门的输出气泡，因此这个与非门就不需要变化，图2-34(b) 中，中间的门没有带输入气泡，于是可将最左边的门转变成不带输出气泡的，如图2-34(c) 所示。现在电路中所有除了输入以外的气泡都删除，于是可以通过观察产生基于输入原值或互补值并采用AND或OR方式构成表达式，$Y=\bar{A}\bar{B}C+\bar{D}$。

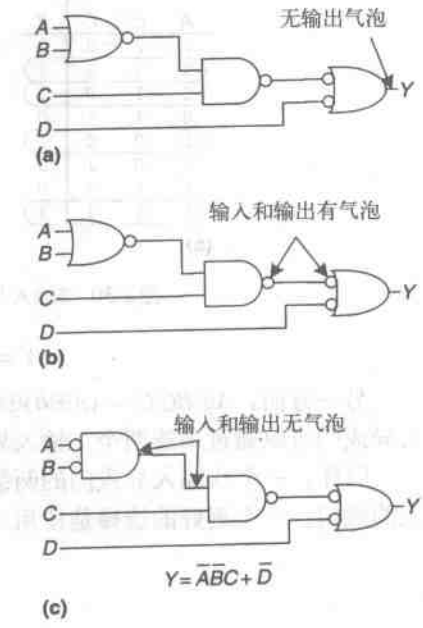


图2-34 推气泡式电路

为了强调最后一点，图2-35显示了和图2-34电路等效的逻辑电路。气泡是串联删除的，可以忽略中间门的输出气泡和最右边门的输入气泡，而产生一个逻辑等效电路图，如图2-35所示。

例2.8 为CMOS逻辑推气泡

很多设计师按照与门和或门来思考，但是假设你需要用CMOS来实现如图2-36的逻辑电路，而CMOS逻辑电路偏向于使用与非门和或非门。使用推气泡将电路转变为使用与非门、或非门和非门实现。

解：一个解决方案是用与非门和非门代替每一个与门，用或非门和非门代替每一个或门，如图2-37所示。这种方法需要8个门。注意到这里非门的气泡被画在前面而不是后面，下面将讨论如何删除这些非门。

一个较好的解决方案是：注意到气泡可以添加到门的输出和下一个门的输入而不改变逻辑功能，如图2-38(a) 所示。最后一个与门被改变成一个与非门和一个非门，如图2-38(b) 所示。这个解决方案仅仅需要5个门。

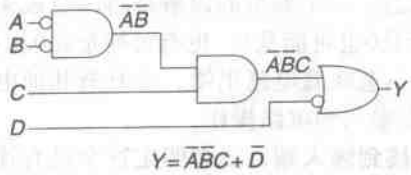


图2-35 采用推气泡法产生的逻辑等价电路

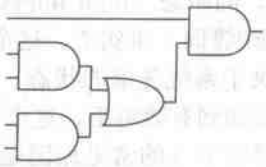


图2-36 使用AND和OR的电路

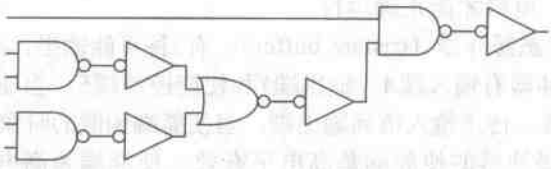


图2-37 使用NAND和NOR实现的较差电路

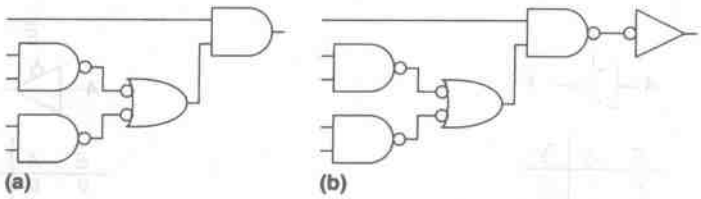


图2-38 使用NAND和NOR实现的较好电路

2.6 X和Z

布尔代数局限于为0和1。然而，真实的电路中会出现非法值和浮空现象，分别用X和Z来表示。

2.6.1 非法值X

符号X表示电路节点的值未知（unknown）或非法（illegal），通常会发生在此节点同时被0或者1驱动。图2-39显示了这种情况，节点*Y*同时被高电平和低电平驱动，称为竞争（contention），是必须避免的错误。在竞争节点上的真实电压可能处于0和V_{DD}之间，取决于驱动高电平和低电平两个门的相对强度。它经常，但也不总是，处于在禁止区域。竞争将在两个竞争的门之间产生大电流，使得电路发热并可能损坏器件。

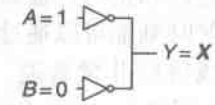


图2-39 有冲突的电路

X值有时也被电路模拟器用来表示一个没有初始化的值。比如，忘了明确说明一个输入的值，仿真器将假定它的值是X而发出警告。

回忆2.4节的内容，数字设计师经常使用符号X来表示在真值表中不用关心的值。一定不要混淆这两种意义。当X出现在真值表中时，它表示不重要的值。当X出现在电路中时，它的意思是电路节点有未知或者非法的值。

2.6.2 浮空值Z

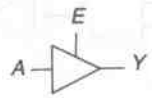
符号Z表示节点既没有被高电平驱动也没有被低电平驱动。这个节点被称为浮空

(floating)，高阻态（high impedance），或者高Z态。一个典型的误解是将浮空或未被驱动的节点和逻辑0等同。事实上，这个浮空的节点可能是0也可能是1，也有可能是在0和1之间的电压。这取决于系统先前的状态。一个浮空节点并不意味着电路出错，一旦有其他电路组件将这个节点驱动到有效电平，这个节点上的值就可以参与到电路操作。

产生浮空节点的常见原因是忘记将电压值连接到输入端，或者假定这个没有连接的输入为0。当浮空输入在0和1之间随机变化时，这种错误可能导致电路行为的不确定。实际上，人接触电路时，体内的静电就可能足够触发电路改变状态。我们就曾经发现只有在学生把一个手指压在芯片上时，电路才能正确运行。

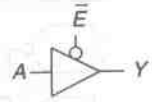
图2-40所示的三态缓冲器（tristate buffer），有3种可能输出：高电平（1），低电平（0）和浮空（Z）。三态缓冲器有输入端A、输出端Y和使能控制端E。当使能端为真时，三态缓冲器作为一个简单的缓冲器，传送输入值到输出端。当使能端为假的时候，输出被置为高阻态（Z）。

图2-40中三态缓冲器的使能端是高电平有效：使能端为高电平（1）时缓冲器使能。图2-41给出了低电平有效使能端的三态缓冲器：使能端为低电平（0）时三态缓冲器使能。为表示信号在低电平时有效，通常在输入上放置一个气泡，也经常在它的名字上面画一条横线（$\bar{E}$）。



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

图2-40 三态缓冲器



Ē	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

图2-41 低电平有效使能端的三态缓冲器

三态缓冲器经常在连接多个芯片的总线中使用。例如，微处理器、视频芯片和以太网芯片都可能需要和个人电脑的存储器通信，每个芯片可以通过三态缓冲器连接共享的存储总线，如图2-42所示。在某一个时刻只允许有一个芯片的使能信号有效，并向总线驱动数据。而其他芯片的输出必须为浮空，以防止它们和正与存储器通信的芯片产生竞争。任何芯片在任何时刻都可以通过共享总线来读取信息。这样的三态总线曾经非常普遍。但是，在现代计算机中需要点到点（point-to-point link）连接以获得更高的速度。此时芯片之间就是直接互连而不再通过共享的总线了。

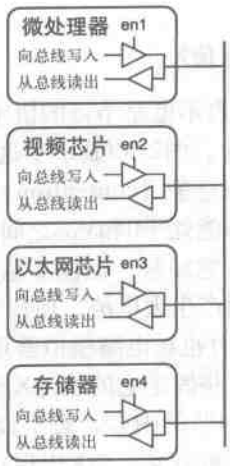


图2-42 通过三态总线互连多个芯片

2.7 卡诺图

通过使用布尔代数对几个布尔表达式进行化简，我们发现：如果不小心的话，有时会得到完全不同的表达式结果，而非最简的表达式。卡诺图（Karnaugh map，K-map）是一个图形化的化简布尔表达式方法，由贝尔实验室的电信工程师Maurice Karnaugh在1953年发明。卡诺图对处理最多4个变量的问题非常好。更主要的是，给出了布尔表达式化简的可视化方法。

回忆逻辑化简过程中需要组合不同的项。如果两个项包含同一个蕴涵项P，并包含与其他变量A的真和假形式，这两项就可以合并并消去A：$PA + P\bar{A} = P$。卡诺图将这些可以合并的项放在相邻的方格中，使得它们很容易被看到。

图2-43显示一个三输入函数的真值表和卡诺图。卡诺图的最上一行给出了输入值A、B的4种可能值。最左边列给出了输入值C的两种可能值。卡诺图中的每一个方格与真值表一行中的输出值Y相对应。比如，最高最左的方格与真值表中的第一行对应表示当$ABC = 000$时，输出$Y = 1$。就像真值表的每一行，在卡诺图中的每一个方格代表了一个最小项。图2-43（c）显示了和卡诺图中每一个方格相对应的最小项。

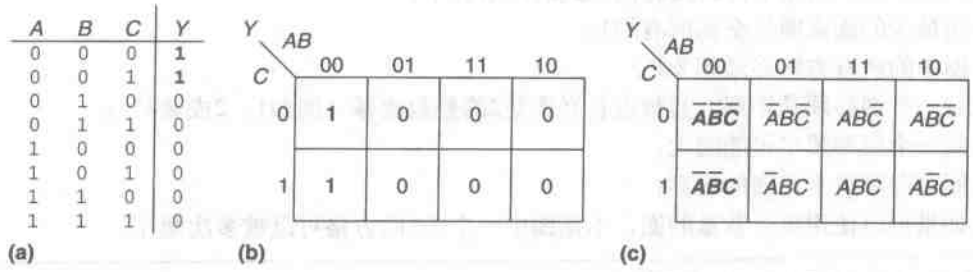


图2-43 三输入函数：(a) 真值表；(b) 卡诺图；(c) 卡诺图中的最小项

每一个方格（最小项）和相邻的方格中有一个变量的值不同。这意味着相邻的方格除了一个变量不同以外具有所有相同的变量，这个变量在一个方格里面为真，而在另一个方格中为假。比如，相邻的两个方格代表了最小项$\bar{A}B\bar{C}$和$\bar{A}BC$，它们仅仅是变量C不同。读者可能已经注意到在最高行中，A和B按照特殊的顺序组合：00、01、11、10。这种顺序称为格雷码。它与普通的二进制顺序不同（00、01、10、11），在相邻的项中只有一个变量不同。比如，在格雷码中01到11仅仅是A从0变化成1，在普通的二进制顺序中01到10则是A从1变化成0和B从0变化成1。所以，按照普通二进制顺序写出的组合项不具有“相邻方格中只有一个变量不同”的性质。

卡诺图也可以环绕。最右边的方格可以和最左边的方格按照仅仅只有一个变量A不同的方式进行有效的连接。换句话说，可以拿起图将其卷成一个圆柱体，连接圆柱体的末端成一个圆环，仍然保证了相邻的方格仅仅只有一个变量不同。

2.7.1 画圈的原理

图2-43的卡诺图中，仅仅在左边出现了2个为1的最小项$\bar{A}\bar{B}\bar{C}$和$\bar{A}\bar{B}C$。从卡诺图中读取最小项的方法完全与直接从真值表读与或式的方法相同。之前，我们用布尔代数将等式最小化成与或式。

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C = \bar{A}\bar{B}\,(\bar{C} + C) = \bar{A}\bar{B} \tag{2-6}$$

卡诺图中可用如图2-44所示图解法化简：将值为1的相邻方格圈起来。对于每一个圈可以写出相应的蕴涵项。2.2节指出，一个蕴涵项是由一个或者多个变量产生。在同一个圈中同时包含了某个变量的真和假时，该变量可以从蕴涵项中删除。在这种情况下，在圈中变量C的真形式（1）和假形式（0）都在圈中，因此它不包含在蕴涵项中。换句话说，当$A = B = 0$时Y为真，而不取决于C的值。于是蕴涵项为$\bar{A}\bar{B}$。卡诺图给

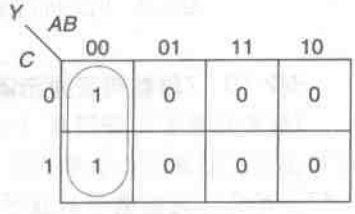


图2-44 卡诺图化简

出了利用布尔代数获得的相同结果。

2.7.2 卡诺图化简逻辑

卡诺图提供了一种简易直观的方式化简逻辑。用尽可能少数的圈来圈住卡诺图中所有为1的方格。每一个圈应该尽可能的大，然后读取每个圈的蕴涵项。

更加正式的说，布尔表达式写成最少数量的主蕴涵项相或时，布尔表达式得到最终化简。卡诺图中的每一个圈代表一个蕴涵项。最大的圈是主蕴涵项。

例如，图2-44所示的卡诺图中$A\bar{B}\bar{C}$和$A\bar{B}C$是蕴涵项，但不是主蕴涵项。卡诺图中只有$A\bar{B}$是主蕴涵项。从卡诺图中得到最简化等式的规则如下：

1) 用最少的圈来圈住全部所有的1；
2) 圈中的所有方格必须都为1；
3) 每一个圈必须是矩形，其每边长必须是2的整数次幂（例如1，2或者4）；
4) 每一个圈必须尽可能的大；
5) 圈可以环绕卡诺图的边界；
6) 如果可以使用更少数量的圈，卡诺图中一个为1的方格可以被多次圈住。

例2.9 用卡诺图化简3变量函数

假设有一个函数$Y=F(A, B, C)$，其卡诺图如图2-45所示。用卡诺图化简等式。

解：用尽可能少的圈来圈住卡诺图中为1的值，如图2-46所示。卡诺图中的每个圈代表了一个主蕴涵式，每一个圈的边长都是2的整数次幂（2×1和2×2）。可以通过写出包含在圈中仅仅出现真或假形式的变量来确定每一个圈的主蕴涵式。

例如，在2×1的圈中，B为真和为假的形式包含于圈中，于是主蕴涵式中不包含B。仅仅A的真（A）和C的补（$\bar{C}$）在这个圈中，可以得出这些变量的主蕴涵式为$A\bar{C}$。同样在2×2的圈中圈住了$B=0$的全部方格。于是主蕴涵式为$\bar{B}$。

注意，为了使主蕴涵项最大，右上角的方格（最小项）被覆盖了两次。这等效于布尔代数中共享最小项来减少蕴涵项大小的化简技术。同样需要注意的是卡诺图边上4个方格的圈。

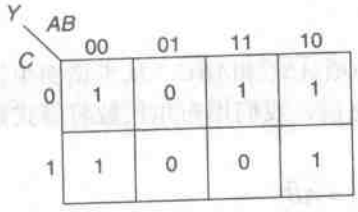


图2-45 例2.9的卡诺图

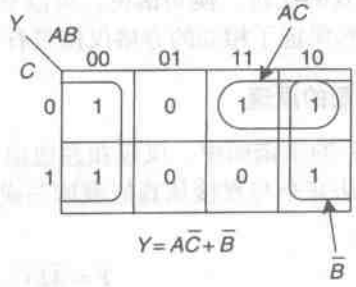


图2-46 例2.9的解

例2.10 7段数码管显示译码器

7段数码管显示译码器（seven-segment display decoder）通过4位数据的输入$D_{3:0}$，产生7位输出以控制发光二极管来显示数字0～9。这七位输出一般称为段a～段g，或者S_a～S_g，如图2-47所定义。这些数字的显示由图2-48中给出。写出针对输出S_a和S_b的4输入真值表，并用卡诺图化简布尔表达式。假设非法的输入值（10～15）不产生任何显示。

解：表2-6给出了真值表。例如，输入为0000将点亮处除S_g以外所有的数码管。

这7个输出分别是关于4个变量的独立函数。输出S_a和S_b的卡诺图如图2-49所示。注意行和列都按照格雷码顺序排列：00、01、11、10，因此相邻方格中仅仅只有一个变量不同。在方格中写输出量时，也必须注意记住这个顺序。

接着，圈住主蕴涵式。用最少数量的圈来圈住所有的1。一个圈可以圈住边缘（水平方向和垂直方向），一个1可以被圈住多次。图2-50显示主蕴涵式和化简了的布尔表达式。

注意包含最少变量的主蕴涵式不是唯一的。比如，在S_a的卡诺图中0000项可以沿着1000项圈起来，以产生最小项$\overline{D_2}\,\overline{D_1}\,\overline{D_0}$。这个圈也可以用0010代替，产生最小项$\overline{D_3}\,\overline{D_2}\,\overline{D_0}$，如图2-51中的虚线所示。

图2-52描述了一个产生非主蕴涵项的常见错误：用一个单独的圈来圈住左上角的1。这个最小项是$\overline{D_3}\,\overline{D_2}\,\overline{D_1}\,\overline{D_0}$，它给出的与或式不是最小的。这个最小项应该和相邻的较大圈组合，如在之前的图2-51所示。

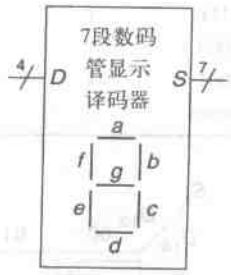


图2-47 7段数码管显示译码器电路符号

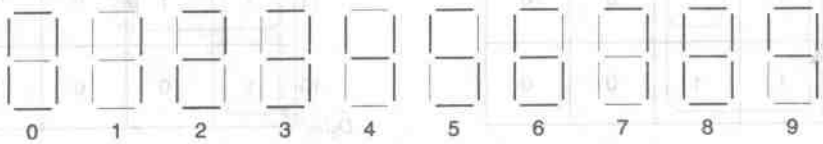


图2-48 7段数码管显示的数字

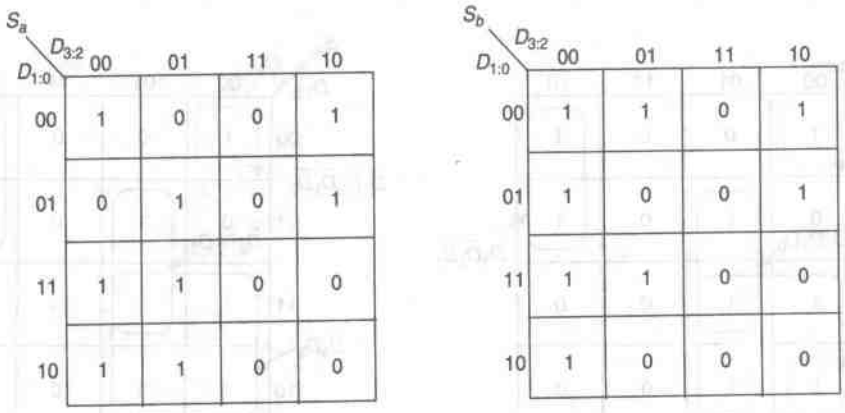


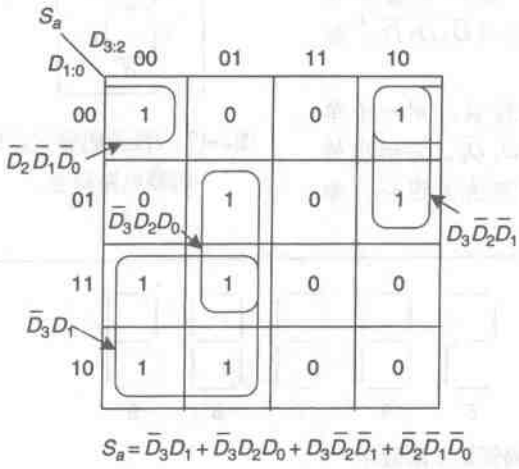
图2-49 S_a和S_b的卡诺图

表2-6 7段数码管显示译码器真值表

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1

（续）

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0



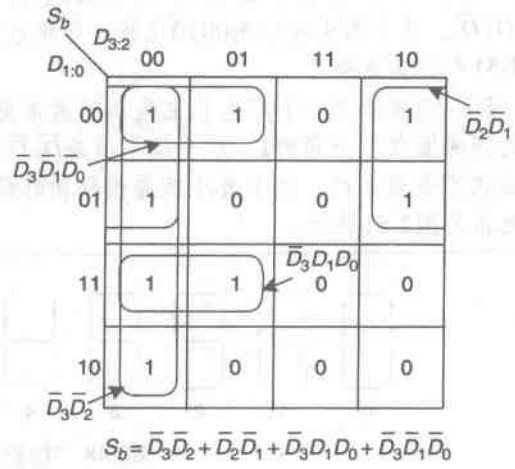


图2-50 例2.10的卡诺图解

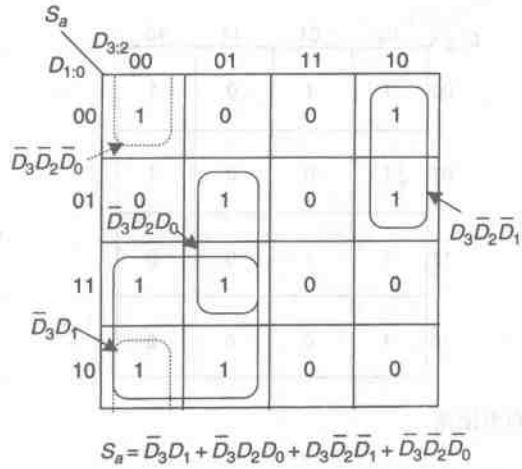


图2-51 产生S_a不同主蕴涵项集合的另一种卡诺图

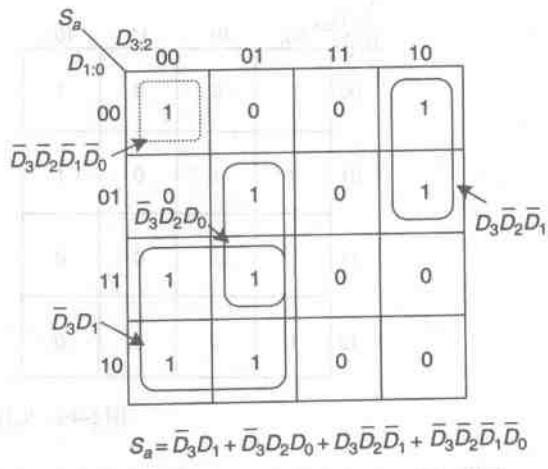


图2-52 产生S_a非主蕴涵项的另一种卡诺图

2.7.3 无关项

2.4节中介绍了真值表无关项。当一些变量对输出没有影响时可以减少表中行的数量。无关项表示成符号X，它的意思是输入可能是0或者是1。

当输出的值不重要或者相对应的输入组合从不出现时，无关项也会出现在真值表的输出。

由设计师决定这些输出是0还是1。

在卡诺图中，无关项可以进一步帮助化简逻辑。如果可以用较少和较大的圈覆盖1，这些无关项也可以被圈起来。但如果它们没有帮助，也可以不被圈起来。

例2.11 带有无关项的7段数码管显示译码

如果我们不考虑非法输入值10～15产生的输出，重复例2.10。

解：卡诺图如图2-53所示，X表示无关项。因为无关，所以可以为0或者1，如果它允许用较少和较大的圈来覆盖住1，我们就圈住这些无关项。被圈起来的无关项被认为是1，反之，没有被圈起来的无关项为0。注意到S_a中环绕四个角的2×2方格。利用无关项可以很好地来化简逻辑式。

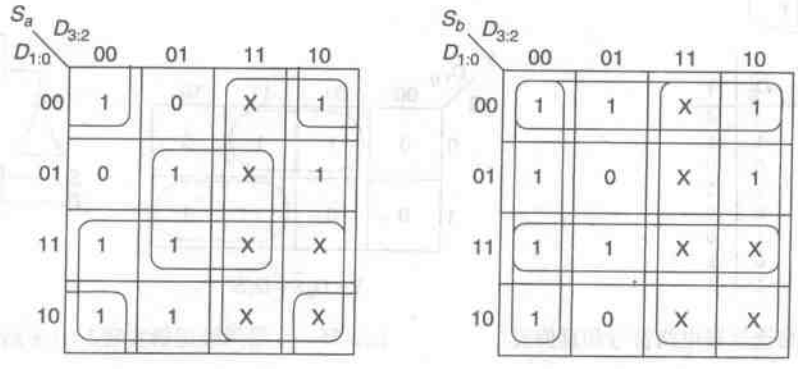


图2-53 带无关项的卡诺图解

2.7.4 小结

布尔代数和卡诺图是两种逻辑化简方法。最终的目标都是找出开销最低的特定逻辑函数实现方法。

在现代的数字设计实践中，逻辑综合（logic synthesizers）软件通过逻辑函数的描述来产生化简电路，这将在第4章中介绍。对于大的问题，逻辑综合比人工方法更高效。对于小问题，有经验的设计者可以通过观察的方法来找出好的解决方案。作者在工作中没有使用过卡诺图来解决实际问题。但是通过卡诺图获得的洞察力很有价值，并且卡诺图也经常出现在工作面试中。

2.8 组合逻辑模块

组合逻辑电路经常组成一个更大的模块以实现更复杂的系统。这是抽象原理的一个应用：隐藏不重要的门级细节，把重点放在模块的功能上。我们已经学习了三种组合逻辑模块:全加器（2.1节），优先级电路（2.4节）和7段显示译码器（2.7节）。在这一节，我们将介绍两种更常用的组合逻辑模块：多路选择器和译码器。第5章中将介绍其他的组合逻辑模块。

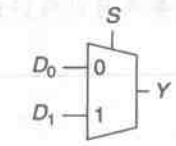
2.8.1 多路选择器

多路选择器（multiplexer）是一种最常用的组合逻辑电路。它从几个可能的输入中根据选择（select）信号的值来选择一个作为输出。多路选择器有时简称为mux。

2：1多路选择器

图2-54中给出了2：1多路选择器的原理图和真值表。它有2个输入D_0和D_1，一个选择输入S和一个输出Y。多路选择器根据选择信号的值在2个输入数据中选择一个数据输出：如果$S=0$，$Y=D_0$；如果$S=1$，$Y=D_1$。S也称为控制信号（control signal），因为它控制多路选择器如何操作。

一个2：1多路选择器可以由与或逻辑实现，如图2-55所示。多路选择器的布尔表达式可以通过卡诺图或者通过分析得到（如果$S=1$ AND $D_1=1$或者$S=0$ AND $D_0=1$，则$Y=1$）。



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

图2-54 2：1多路选择器电路符号和真值表

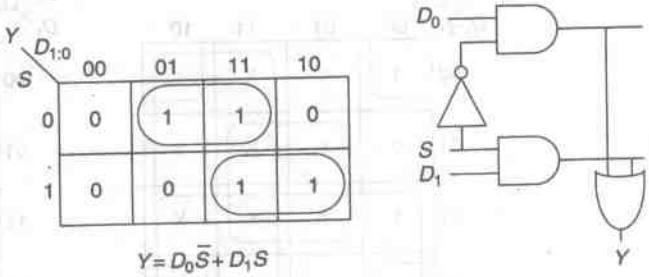


图2-55 使用两级逻辑实现2：1多路选择器

多路选择器也可以由三态缓冲门构成，如图2-56所示。应合理设置三态门使能信号使得在任何时刻仅有一个三态缓冲门有效。当$S=0$，三态门T0有效，允许D_0输出到Y；当$S=1$，三态门T1有效，允许D_1输出到Y。

更宽的多路选择器

一个4：1多路选择器有4个数据输入和1个输出，如图2-57所示。需要2位选择信号在4个输入数据中进行选择。4：1多路选择器可以使用与或逻辑构成，也可以使用三态门或多个2：1多路选择器构成。如图2-58所示。

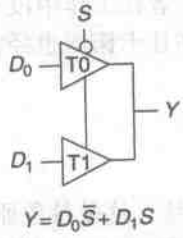


图2-56 使用三态缓冲实现2：1多路选择器

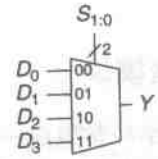


图2-57 4：1多路选择器

三态门的使能项可以用与门和非门组成，也可以用2.8.2节中介绍的译码器组成。

8：1和16：1等更宽的多路选择器可以如图2-58的扩展方法构造。总之，一个N：1多路选择器需要$\log_2 N$位选择线。好的实现选择还是取决于具体的实现技术。

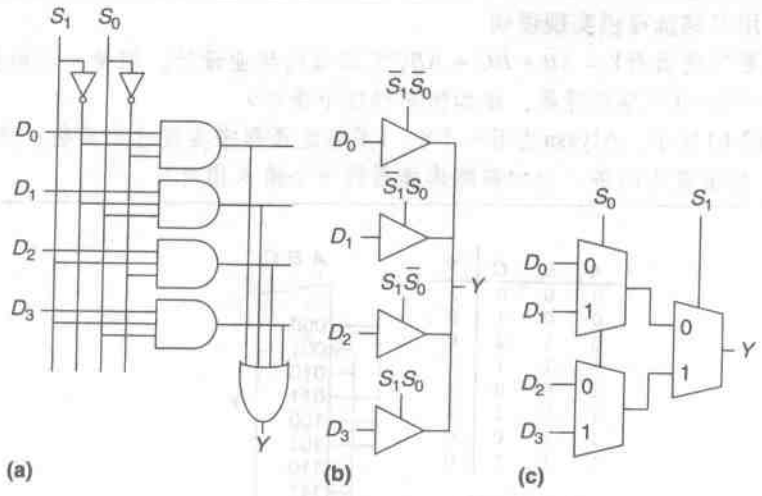


图2-58 4：1多路选择器的实现：(a) 两级逻辑；(b)三态；(c) 层次结构

用多路选择器实现逻辑

多路选择器可以用查找表（lookup table）的方式实现逻辑功能。图2-59中用一个4：1多路选择器实现二输入与门。输入A和B作为选择信号。多路选择器的数据输入根据真值表中相应的值（0或者1）相连。总之，一个2^N个输入的多路选择器可以通过将合适的输入连接到0或者1的方法实现任何的N输入逻辑函数。此外，通过改变数据的输入，多路选择器可以被重新编程来实现其他的函数。

稍微考虑一下，我们能够将多路选择器的大小减少一半：仅仅使用一个2^{N-1}输入的多路选择器来实现任何N输入逻辑函数。方法是将一个变量和0或1一样作为多路选择器的输入。

图2-60进一步说明这个原理。该图用2：1多路选择器分别实现了二输入与函数和异或函数。我们从一个普通的真值表开始，组合一对行并将此变量表示在输出中以消解最右边的变量。比如，与门的情况，当$A=0$时，不管B取何值，$Y=0$；当$A=1$时，如果$B=0$，则$Y=0$，如果$B=1$，则$Y=1$，于是可以得到$Y=B$。按照这个更小的新真值表，我们可以将多路选择器作为一个查找表来使用。

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y=AB$

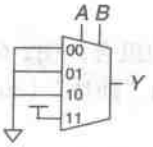


图2-59 用一个4：1多路选择器实现二输入的与门

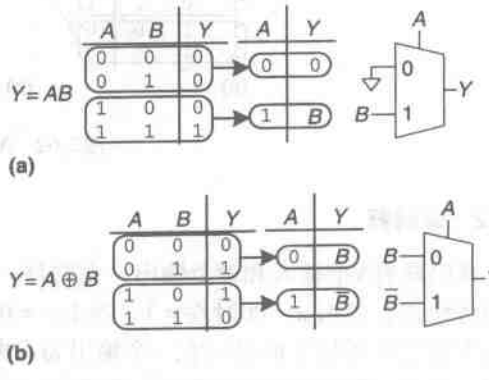


图2-60 使用可变输入的多路选择器逻辑

例2.12 用多路选择器实现逻辑

Alyssa需要实现函数$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$完成她的毕业设计。但是，当她查看实验工具箱，发现只剩下一个8：1多路选择器。她如何实现这个函数？

解：如图2-61所示，Alyssa使用一个8：1多路选择器来实现这个函数。多路选择器充当了一个查找表，真值表中的每一行和多路选择器的一个输入相对应。

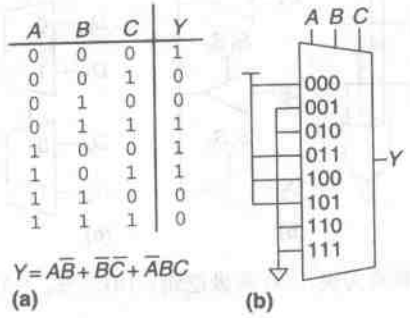


图2-61 Alyssa的电路：(a)真值表；(b) 多路选择器实现

例2.13 再次用多路选择器实现逻辑

在期末报告前，Alyssa打开电路的电源，结果将8：1多路选择器烧坏了（由于昨天整晚没有休息，她意外地用20V电压代替5V电压供电）。她请求她的朋友将余下的元器件给她。他们给了她一个4：1多路选择器和一个非门。她如何只用这些部件构造新电路？

解：通过让输出取决于C的值，Alyssa将真值表减少到4行。（她也尝试过重新排列真值表的列，使输出取决于A或B。）图2-62给出了新的设计。

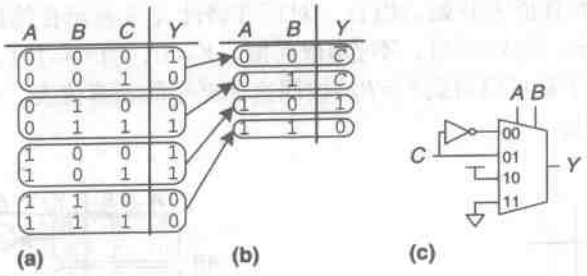


图2-62 Alyssa的新电路

2.8.2 译码器

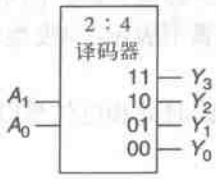
译码器有N个输入和2^N个输出。它的每一个输出都取决于输入的组合。图2-63给出了一个2：4译码器。当$A_{1:0} = 00$时$Y_0 = 1$，当$A_{1:0} = 01$时$Y_1 = 1$等。输出称为“独热”（one-hot）状态，因为在给定的条件下恰好只有一个输出为高电平。

例2.14 译码器的实现

用与门、或门和非门实现一个2：4译码器。

解：使用了4个与门来实现一个2：4译码器的电路如图2-64所示。每一个门依赖于所有输

入的真或补。总之，一个$N:2^N$的译码器可以由2^N个N输入的与门通过接收所有输入的真或者假的各种组合值构成。译码器的每一个输出代表了一个最小项。比如，Y_0代表最小项$\overline{A}_1\overline{A}_0$。这在和其他数字模块一同使用时会很方便。



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

图2-63 2：4译码器

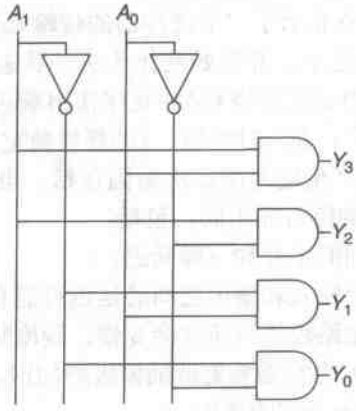


图2-64 2：4译码器实现

用译码器实现逻辑

译码器可以和或门组合在一起来实现逻辑函数。图2-65显示了用一个2：4译码器和一个或门来实现一个二输入XNOR函数。因为译码器的每一个输出都代表一个最小项，将以所有最小项的或形式来实现函数。在图2-65中，$Y = \overline{A}\overline{B} + AB = \overline{A \oplus B}$。

当使用译码器来构造逻辑电路时，很容易将函数表示成真值表的形式或者标准与或式。一个包含了M个输出为1的N输入函数，可以通过一个$N:2^N$译码器和M输入或门实现。在5.5.6节中，这个概念将应用到只读存储器中。

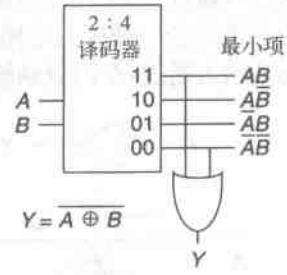


图2-65 使用译码器的逻辑函数

2.9 时序

在之前的章节，我们主要关心在使用最少数量门的理想状态下电路是否工作。但是，任何经验丰富的电路设计师都认为电路设计中最具有挑战性的问题是时序（timing）：如何使电路运行得最快。

一个输出响应输入的改变而改变需要一定时间。图2-66显示了缓冲器的一个输入改变和随后输出的改变所产生的延迟。这个图称为时序图（timing diagram），描绘了输入改变时缓冲器电路的瞬间响应（transient response）。从低电平到高电平的转变称为上升沿。同样，从高电平到低电平的转变称为下降沿（在图中没有显示）。在输入信号A的50%点到输出信号Y的50%点之间测量延迟。50%点是信号在转变过程中电压处于高电平和低电平之间中间点的位置。

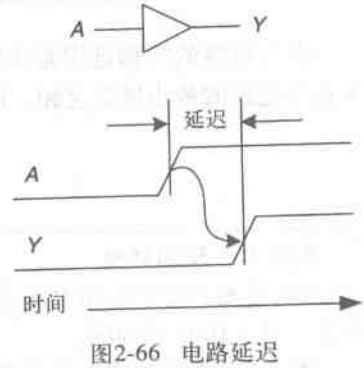


图2-66 电路延迟

2.9.1 传输延迟和最小延迟

组合逻辑电路的时序特征包括传输延迟（propagation delay）和最小延迟（contamination delay）的特征。传输延迟t_{pd}是输入改变直到对应的一个和多个输出达到它们最终的值所经历的最长时间。最小延迟t_{cd}是当一个输入发生变化到任何一个输出开始改变的最短时间。

图2-67显示了一个缓冲器的传输延迟和最小延迟。图中显示在特定时间内A的初值是高电平或者低电平，并开始变化为另一状态。我们只对值的改变过程感兴趣，而不用关心值是多少。Y在稍后时间将对A的变化作出响应，并产生变化。这些弧形表示在A发生转变t_{cd}时间后，Y开始改变；在t_{pd}时间后，Y的新值稳定下来。

电路产生延迟深层次原因包括：电路中电容充电所需要的时间和电信号以光速传播。由此t_{pd}和t_{cd}的值可能不同，包括：

- 不同的上升和下降延迟；
- 多个输入和输出之间的延迟可能有所不同；
- 当电路较热时速度会变慢，较冷时会变快。

计算t_{pd}和t_{cd}需要更低抽象级的知识，超出了本书的范围。但是芯片制造商通常提供数据手册以说明每个门的延迟。

根据已经列举出的各种因素，传输延迟和最小延迟也可以由一个信号从输入到输出的路径来确定。图2-68给出了一个四输入逻辑电路。关键路径（critical path）是从A或者B到输出Y。因为输入通过了3个门才传输到输出，所以它是最长的一条路径，也是最慢的路径。这个路径成为关键路径，是因为它限制了电路运行的速度。最短路径（short path）从输入D到输出Y。因为从输入通过1个门就到输出，此路径是最短的路径，也是通过电路的最快路径。

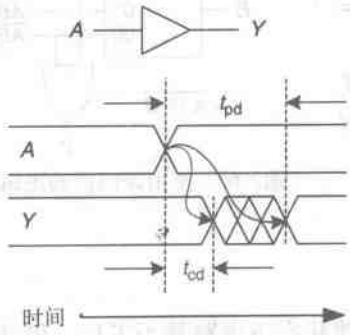


图2-67 传输延迟和最小延迟

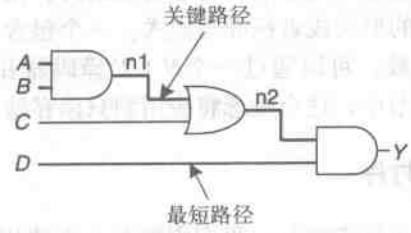


图2-68 最短路径和关键路径

组合电路的传输延迟是关键路径上每一个元件的传输延迟之和。最小延迟是在最短路径上每个元件的最小延迟之和。这些延迟如图2-69所示，也可由下列等式描述：

$$t_{pd} = 2\,t_{pd_AND} + t_{pd_OR} \tag{2-7}$$

$$t_{cd} = t_{cd_AND} \tag{2-8}$$

例2.15 延迟计算

Ben需要计算图2-70中的传输延迟和最小延迟。根据数据手册，每个门的传输延迟和最小延迟分别为100ps和60ps。

解：Ben首先确定电路中的关键路径和最短路径。关键路径是从输入A或者B通过3个门到输出Y，在图2-71中用粗线显示。所以，t_{pd}是单个门传输延迟的三倍，即300ps。

最短路径是从输入C、D或者E通过2个门到达输出Y，如图2-72所示。在最短路径上有2个门，所以t_{cd}为120ps。

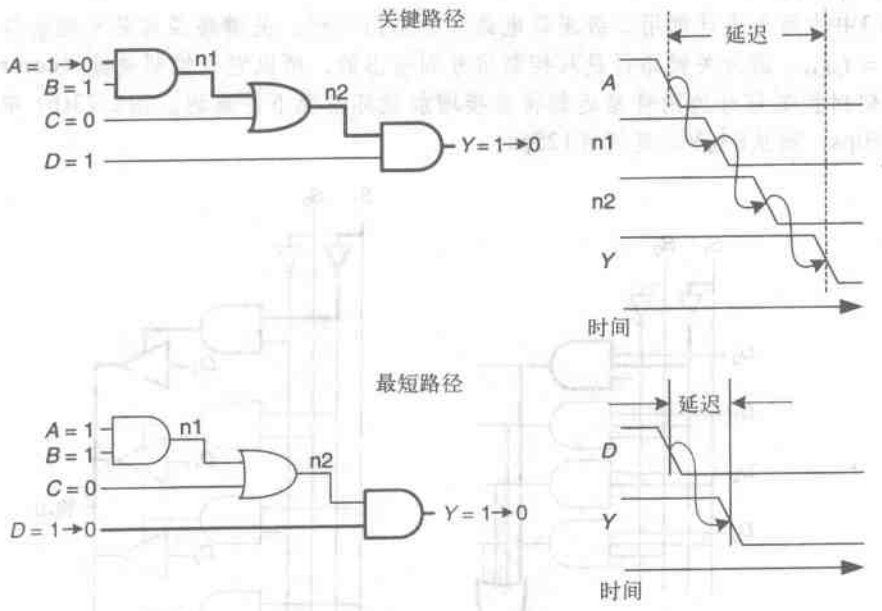


图2-69 关键路径和最短路径上的信号传输

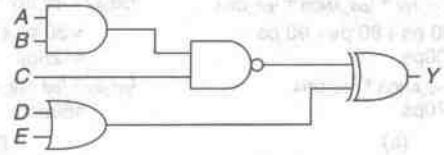


图2-70 Ben的电路

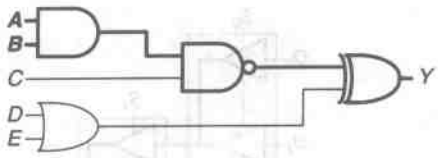


图2-71 电路的关键路径

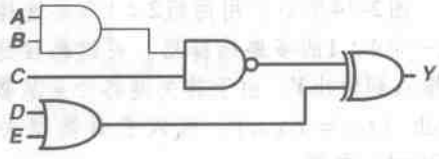


图2-72 电路的最短路径

例2.16 多路选择器的时序：比较控制关键路径和数据关键路径

比较2.8.1节中图2-58所示3种四输入多路选择器的最坏情况时序。表2-7列出了元件的传输延迟。每一种设计的关键路径是什么？选择一个你认为好的设计，并且给出时序分析。

表2-7 多路选择器电路组件的时序规范

门	t_{pd}(ps)	门	t_{pd}(ps)
NOT	30	4-input OR	90
2-input AND	60	tristate (A to Y)	50
3-input AND	80	tristate (enable to Y)	35

解：如图2-73和图2-74所示，3种设计方法的关键路径都用粗线显示。t_{pd_sy}表示从输入S到输出Y的传输延迟，t_{pd_dy}表示从输入D到输出Y的传输延迟，t_{pd}是这两个延迟的最坏情况，即$\max(t_{pd_sy}, t_{pd_dy})$。

图2-73中的两个设计都用二级逻辑电路和三态门实现。关键路径都是从控制信号S到输出信号Y：$t_{pd} = t_{pd_sy}$。因为关键路径是从控制信号到输出的，所以它是控制关键（control critical）电路。任何对控制信号的附件延迟都将直接增加最坏情况下的延迟。图2-73(b) 中从D到Y的延迟只有50ps，而从S到Y的延迟有125ps。

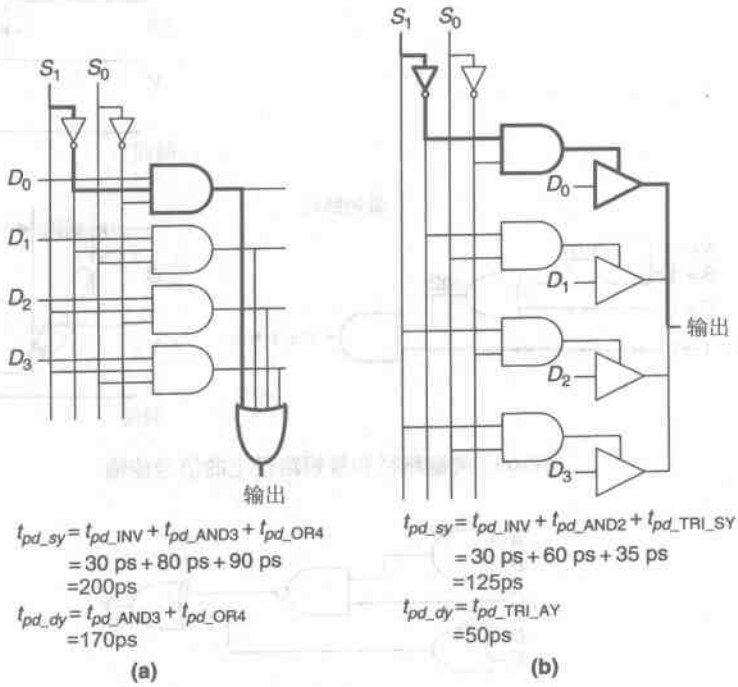


图2-73 4：1多路选择器传输延迟：(a) 两级逻辑；(b) 三态

图2-74显示了用两级2：1多路选择器分层实现一个4：1的多路选择器。关键路径是任意一个D输入到输出Y。由于其关键路径是从数据输入到输出（$t_{cd} = t_{cd_dy}$），所以它是数据关键（data critical）电路。

如果数据输入在控制输入之前到达，我们将倾向于具有最短控制－输出延迟的设计（如图2-74分层的设计）。同样，如果控制信号在数据信号之前到达，我们将选择具有最短数据－输出的延迟的设计（如图2-73(b) 中的三态设计）。

最好的设计选择不仅取决于通过电路的关键路径和输入到达的时间，而且取决于功耗、成本、可用性等诸多因素。

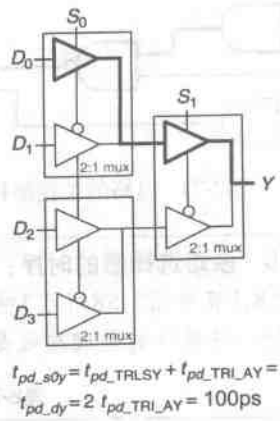


图2-74 使用2：1多路选择器层次实现的4：1多路选择器

2.9.2 毛刺

到目前为止，我们已经讨论了一个输入信号的改变导致一个输出信号的改变。但是，一个输入信号的改变可能会导致多个输出信号的改变。这被称为毛刺（glitch）或者冲突（hazard）。虽然毛刺通常不会导致什么问题，但是了解它们的存在和在时序图中识别它们也是很重要的。图2-75显示了一个会产生毛刺的电路的卡诺图。

布尔表达式的最小化是正确的，考察图2-76中当$A = 0$，$C = 1$，B从1变成0时会发生什么情况。最短的路径通过与门和或门2个门。关键路径通过了一个反相器和两个与门和或门。

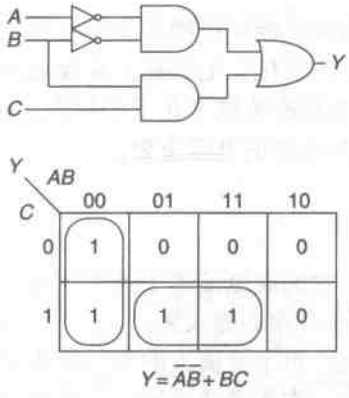


图2-75 产生毛刺的画圈方式

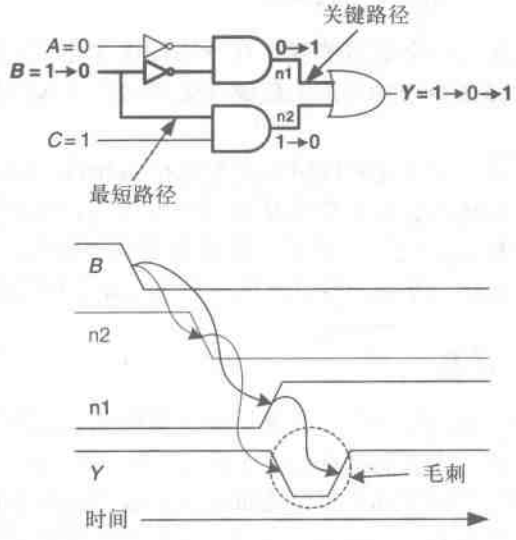


图2-76 产生毛刺的时序

当B从1变成0，$n2$（在最短路径上）在$n1$（在关键路径上）上升之前下降。直到$n1$上升前，2个输入到或门的都是0，输出Y下降到0。当$n1$最后上升后，Y的值回到1。时序图如图2-76所示，Y的值从1开始，结束时也为1，但是存在暂时为0的毛刺。

只要读取输出之前的等待时间和传输延迟一样长，出现毛刺是不会有问题的，这是因为输出最终将稳定在正确的值。

可以在已有实现中增加门电路来避免毛刺。这从卡诺图中最容易理解。图2-77显示了从$ABC = 001$变成$ABC = 011$时在B上输入的改变，使得一个主蕴涵项圈移到另外的一个。这个变化穿过了卡诺图中两个主蕴涵项的边界，从而可能会产生毛刺。

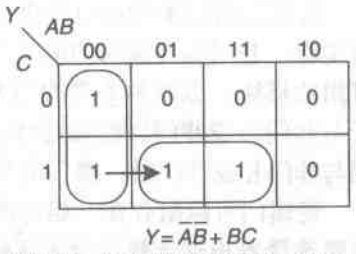


图2-77 输入的改变穿越了蕴涵项的边界

从时序图2-76中可以看出，在一个主蕴涵项的电路开启之前，如果另一个主蕴涵项的电路关闭，就会产生毛刺。为了去除毛刺，可以增加一个新的覆盖主蕴涵项边缘的圈，如图2-78所示。根据一致性定理，新增加的项$\bar{A}\bar{C}$是一致的或者多余的。

图2-79是一个防止毛刺出现的电路，其中增加了一个粗线显示的与门。现在当$A = 0$和$C = 1$时，即使B变化也不会造成输出毛刺，这是因为在整个变化过程中粗线显示的与门始终输出为1。

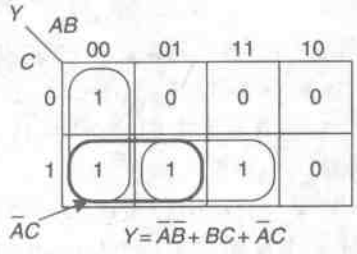


图2-78 无毛刺的卡诺图

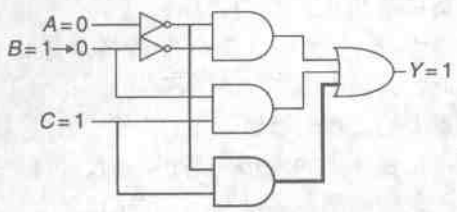


图2-79 无毛刺的电路

总之，一个信号的变化在卡诺图中穿越2个主蕴涵项的边缘时会导致毛刺。我们能够通过在卡诺图中增加多余的蕴涵项盖住这些边缘以避免毛刺。这当然以增加额外的硬件成本为代价。

然而，多个变量同时发生变化也会导致毛刺。这些毛刺不能通过增加硬件来避免。因为多数的系统都会有多个变量同时发生（或者几乎同时发生）变化，毛刺在大多数电路中都存在。我们虽然已经介绍了一种避免毛刺的方法，但讨论毛刺的关键不在于如何取去除它们，而是要意识到毛刺的存在。这一点在示波器和仿真器上看时序图时非常重要。

2.10 总结

数字电路是一个包含离散电压值输入和输出的模块。它的规范描述了模块实现的功能和时序。这一章我们将重点放在组合电路，其输出仅仅取决于当前的输入量。

组合电路的功能可以通过真值表或者布尔表达式确定。每个真值表的布尔表达式可以通过系统地使用“与或式”或者“或与式”获得。与或式中，布尔表达式写成一个或者多个蕴涵式的或（OR）。蕴涵式是各个变量的与（AND）。变量为输入量补真或补的形式。

布尔表达式可以使用布尔代数的规则化简。特别是，可以组合包含了某个变量真和补形式的两个蕴涵项化简成最简的与或式：$PA + \overline{PA} = P$。卡诺图是化简最多含4个变量函数的图形化方法。通过训练，设计师能够将布尔表达式化简成只含最少变量的形式。计算机辅助设计工具也常用于处理更加复杂的表达式，我们将在第4章中介绍更多的这类方法和工具。

连接逻辑门来构造组合电路，实现所希望的功能。任何表达式的与或式可以通过两级逻辑实现：非门实现输入的取反、与门实现变量的与，或门实现变量的或。根据不同的功能和可用的模块，以各种类型的门为基础的多级逻辑实现会更加高效。例如，CMOS电路更倾向于与非门和或非门，因为这些门可以直接用CMOS晶体管来实现而不需要额外的非门。当使用与非门和或非门时，推气泡有助于跟踪取反的过程。

逻辑门可以组合在一起构造更大规模的电路，例如多路选择器、解码器和优先级电路等。多路选择器根据选择信号来选择输出一个输入数据。译码器根据输入量将多个输出中的一个设置为高电平。优先级电路产生表示最高优先级输入的输出。这些电路都是组合电路模块的例子。第5章将介绍更多的组合逻辑模块，包括多种算术电路。这些模块将在第7章微结构中得到广泛的运用。

组合电路的时序包含了电路的传输延迟和最小延迟，它们说明了输入改变和随后的输出改变之间最长和最短的时间。计算电路的传输延迟需要首先确定电路中的关键路径，然后将路径上每一个器件的传输延迟相加。实现复杂的组合逻辑电路有很多方式；这些方式在速度和成本上各有侧重。

下一章将介绍时序电路，它的输出取决于先前的输入和当前的输入。换句话说，时序电路对过去的状态有记忆能力。

习题

2.1 写出图2-80中每个真值表的与或式布尔表达式。

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

图2-80 真值表

2.2 写出图2-80中每个真值表的或与式布尔表达式。

2.3 化简习题2.1中的布尔表达式。

2.4 画一个相对简单的组合电路实现习题2.3中的每一个函数。相对简单表示不要浪费门，但也不要浪费大量的时间来检验每一种可能的实现电路。

2.5 只使用非门、与门和或门来重做习题2.4。

2.6 只使用非门、与非门和或非门来重做习题2.4。

2.7 使用布尔代数方法来化简下列布尔表达式。用真值表或者卡诺图检验其正确性。

(a) $Y = AC + \overline{A}\overline{B}C$

(b) $Y = \overline{A}\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$

(c) $Y = \overline{A}\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

2.8 画一个相对简单的组合电路实现习题2.7中的每一个函数。

2.9 化简下列布尔表达式，画一个相对简单的组合电路来实现化简后的等式。

(a) $Y = BC + \overline{A}\overline{B}\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + \overline{(A + D + E)} + \overline{B}\overline{C}D + \overline{B}\overline{C}E + \overline{B}D\overline{E} + \overline{C}D\overline{E}$

2.10 给出一个行数在30亿和50亿之间的真值表，而此真值表可以用少于40个（至少1个）的二输入门来实现。

2.11 给出一个带有环路但仍然是组合电路的例子。

2.12 Alyssa说任何布尔函数的最小与或式为函数所有主蕴涵项的或。Ben说存在一些函数，它们的最小等式不含有所有的主蕴涵项。解释为什么Alyssa是正确的或者提供一个反例来证明Ben的观点。

2.13 使用完全归纳法证明下列定律。不需要证明它们的对偶式。

(a) 重叠律（T3）

(b) 分配律（T8）

(c) 合并律（T10）

2.14 用完全归纳法证明3个变量B2的德·摩根定律（T12）。

2.15 写出图2-81电路中的布尔表达式。不必化简表达式。

2.16 化简习题2.15中的布尔表达式，画出一个具有相同功能的改进电路。

2.17 使用德·摩根定理等效门和推气泡方法，重新画出图2-82中的电路，从而可以通过观察来写出布尔表达式。写出布尔表达式。

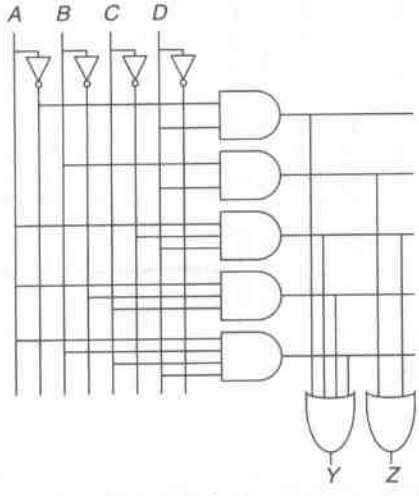


图2-81 电路原理图

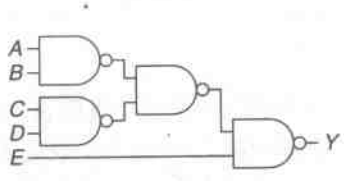


图2-82 电路原理图

2.18 对图2-83中的电路，重复习题2.17 中的练习。

2.19 写出在图2-84中函数的最小布尔表达式，记得要利用无关项。

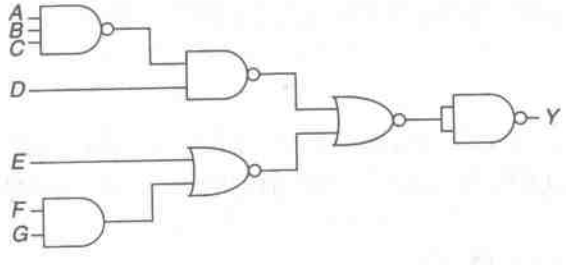


图2-83 电路原理图

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

图2-84 真值表

2.20 画出习题2.19函数的电路。

2.21 当有一个输入改变的时候，习题2.20中你的电路有一些潜在的毛刺码？如果没有，解释为什么没有。如果有，如何修改电路来消除这些毛刺。

2.22 Ben将在没有蚂蚁的晴天去野炊。如果他看到蜂鸟，即使野炊的地方有蚂蚁和瓢虫，他都会去野炊。根据太阳（S），蚂蚁（A），蜂鸟（H）和瓢虫（L）写出他去野炊（E）的布尔表达式。

2.23 完成7段译码器段S_c到S_g的设计。

(a) 假设输入大于9时输出均为0，写出输出$S_c \sim S_g$的布尔表达式。

(b) 假设输入大于9时输出是无关项，写出输出$S_c \sim S_g$的布尔表达式。

(c) 画出合理而简单的门级实现（b）。多重输出在合适的地方可以共享门电路。

2.24 一个电路有4位输入，2位输出。输入是A，代表从0～15中的一个数。如果这个输入的数是素数（0和1不是素数，但是2，3，5等都是素数），则输出P为真。如果输入的数可以被3整除，则输出D为真。给出并化简每个输出的布尔表达式，画出电路图。

2.25 一个优先编码器有2^N位输入。它将产生N位二进制输出表示输入为真的最高位，或者在没有任何输入为真时输出0。它还将产生一个输出NONE。在没有输入位为真时，该输出为真。设计一个八输入的优先编码器，输入为A，输出为Y和NONE。例如，输入为00100000时，输出Y将为101，NONE为0。写出并化简每个输出的布尔表达式，并且画出一个电路图。

2.26 设计一个新的优先级编码器（参见习题2.25）。它有8位输入$A_{7:0}$，产生2个3位输出$Y_{2:0}$和$Z_{2:0}$。Y指示出输入为真的最高位。Z指示出输入为真的第二高位。如果没有一个输入为真，则Y为0，如果不多于一个输入为真，则Z为0。化简每一个输出的布尔表达式，并且画出原理图。

2.27 一个M位温度计码（thermometer code）在最低K位上为1，在最高M-K位为0。二进制－温度计码转换器（binary-to-thermometer code convert）有N位输入和2^N-1位输出。它为输入的二进制数产生一个2^N-1位的温度计码。例如，如果输入是110，则输出是0111111。设计一个3：7二进制－温度计码转换器。写出并化简每一个输出的布尔表达式，并且画出原理图。

2.28 写出图2-85中函数的最小布尔表达式。

2.29 写出图2-86中函数的最小布尔表达式。

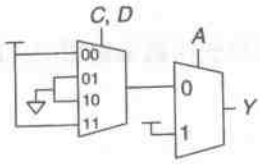


图2-85 多路选择器电路

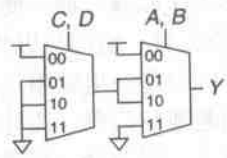


图2-86 多路选择器电路

2.30 请使用下列器件，实现图2-80(b) 中的函数。

(a) 一个8：1多路选择器。

(b) 一个4：1多路选择器和一个非门。

(c) 一个2：1多路选择器和2个其他的逻辑门。

2.31 请使用下列器件，实现图2-9(a) 中的函数。

(a) 一个8：1多路选择器。

(b) 一个4：1多路选择器和不用任何其他的逻辑门。

(c) 一个2：1多路选择器，一个或门和一个非门。

2.32 确定图2-83中电路的传输延迟和最小延迟。在表2-8中给出了所使用门的延迟。

表2-8 针对习题2.32至2.35的门延迟

门	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
2-input OR	55	45
2-input XOR	60	40

2.33 画一个快速3：8译码器的原理图。假定门的延迟在表2-8中给出（并且只能使用表2-8中门）。设计一个具有最短关键路径的译码器，并指出这些路径是什么？电路的传输延迟和最小延迟是多少？

2.34 重新为习题2.24设计一个尽可能快的电路。只能使用表2-8中的门。画出新的电路并且说明关键路径。电路的传输延迟和最小延迟是多少？

2.35 重新为习题2.25设计一个尽可能快的优先编码器。可以使用在表2-8中的任何门。画出新的电路并且说明关键路径。电路的传输延迟和最小延迟是多少？

2.36 设计一个数据输入到数据输出延迟尽可能最短的8：1多路选择器。可以使用表2-7中的任何门。基于表2-7中给出了门的延迟，确定电路的延迟。

面试问题

2.1 仅仅只使用与非门画出二输入或非门函数的原理图，你最少需要使用几个门？

2.2 设计一个电路，它将得出一个输入的月份是否有31天。月份用4位输入$A_{3:0}$指定，比如，输入0001表示1月，输入1100表示12月。当输入的月份有31天时，电路输出Y将为高电平。写出最简等式，使用最少数量的门画出电路图。（提示：记住利用无关项）

2.3 什么是3态缓冲器？如何使用它，为什么使用它？

2.4 如果一个或者一组门可以构造处任何布尔函数，那么这些门就是通用门。比如，{与门，或门，非门}是一组通用门。

(a) 与门是通用门吗，为什么？

(b) {或门，非门}是一组通用门吗，为什么？

(c) 与非门是通用门吗，为什么？

2.5 解释为什么电路的传输延迟可能小于（而不是等于）它的最小延迟。

第3章 时序逻辑设计

3.1 引言

上一章中，我们介绍了如何分析和设计组合逻辑。组合逻辑的输出仅仅取决于当前的输入值。如果给定一个真值表或者布尔表达式形式的规范，就可以得出一个优化的电路。

本章中，我们将分析和设计时序（sequential）逻辑。时序逻辑输出取决于当前的输入值和之前的输入值，所以说时序逻辑具有记忆功能。时序逻辑可能明确地记住某些先前的输入量，也可能从先前的输入量中提取其中一部分信息，称为系统的状态（state）。时序逻辑电路的状态由一组称为状态变量（state variables）的位构成，包含了用于解释电路未来行为的所有过去信息。

这一章中，我们将首先学习锁存器和触发器，它们是能够存储一位状态的简单时序逻辑电路。通常，时序逻辑电路的分析很复杂。为了简化设计，我们将只涉及同步时序逻辑电路。这种电路由组合逻辑和一组表示电路状态的触发器组成。本章中还将介绍有限状态机，它是设计时序电路的一种简单方法。最后，我们将分析时序电路的速度，讨论提高时钟速度的并行化方法。

3.2 锁存器和触发器

双稳态（bistable）元件是存储器件的基本模块。这个元件有两种稳定的状态，如图3-1(a)所示。一对反相器组成的环路构成了一个简单的双稳态元件。图3-1(b) 重画了相同的电路，以突出其对称性。这两个反相器交叉耦合（cross-coupled），即I1的输入是I2的输出取反。这个电路没有输入，但是它有两个输出，Q和$\bar{Q}$。这个电路的分析与组合电路不同，因为它是循环的：Q取决于$\bar{Q}$，$\bar{Q}$反过来又取决于Q。

考虑两种情况，Q = 0或者Q = 1。针对每一种情况的结果，我们可以得到：

• 情况 I：Q = 0

如图3-2（a），I2输入Q = 0，则$\bar{Q}$上的输出为1。I1输入为$\bar{Q}$ = 1，则在Q上的输出为0。这和原来假设的Q = 0是一致的，此时称为稳态（stable）。

• 情况 II：Q = 1

如图3-2（b），I2输入为1，则$\bar{Q}$上输出为0。I1输入为0，Q上输出为1，这是一种相反的稳态。

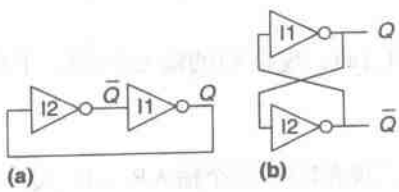


图3-1 交叉耦合的反相器对

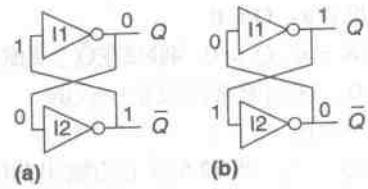


图3-2 交叉耦合反相器的双稳态操作

因为交叉耦合的反相器有两种稳定状态：Q = 0和Q = 1，所以电路被称为双稳态。隐含的

一点是电路可能存在第三种状态，其两个输出均处于0和1之间的中点附近。这种状态称为亚稳态（metastable），我们将在3.5.4节中再进行讨论。

具有N种稳态的元件可以表示$\log_2 N$位的信息，所以双稳态元件可以存储一位信息。交叉耦合反相器状态包含在二进制状态变量Q中。Q的值保持了用以解释电路未来行为的过去信息。在此例子中，如果Q = 0，它将永远保持0值；如果Q = 1，它将永远保持1值。如果Q已知，则$\bar{Q}$也已知，所以电路的另外一个节点$\bar{Q}$不包含其他任何信息。另一方面，$\bar{Q}$也可以作为一个有效的状态变量值。

当第一次加电到此时序电路时，它的初值往往未知和不可预料。电路每一次启动的初值都有可能不同。

虽然交叉耦合反相器可以存储一位的信息，但因为没有用于控制状态的输入，它并没有什么实用价值。其他的双稳态元件，比如锁存器和触发器，提供了可以控制状态变量值的输入，本节下面部分将介绍这些电路。

3.2.1 SR锁存器

SR锁存器是一个最简单的时序电路。如图3-3所示，它由一对耦合的或非门组成。SR锁存器有两个输入S和R，两个输出Q和$\bar{Q}$。SR锁存器与耦合的反相器相似，但是它的状态可以通过输入量S和R来控制，可以设置（set）或复位（reset）Q的输出。

我们通过真值表来理解这个陌生的电路。需要注意的是：或非门中只要有一个输入为1，则输出为0。考虑S和R的四种可能组合。

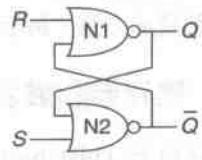


图3-3 SR锁存器原理图

• 情况Ⅰ：R = 1, S = 0

N1至少有一个R = 1的输入，则输出Q = 0，N2的输入Q = 0和S = 0，则输出$\bar{Q}$ = 1。

• 情况Ⅱ：R = 0, S = 1

N1输入0和$\bar{Q}$，因为我们还不知道$\bar{Q}$的值，所以我们不能确定输出的Q值。N2至少有一个S=1的输入，则输出$\bar{Q}$ = 0。再次研究N1，可以知道它的两个输入都为0，所以输出Q=1。

• 情况Ⅲ：R = 1, S = 1

N1和N2至少有一个的输入（R或者S）为1，于是分别产生一个为0的输出。所以Q和$\bar{Q}$同时为0。

• 情况Ⅳ：R = 0, S = 0

N1的输入为0和$\bar{Q}$。因为还不知道$\bar{Q}$的值，所以不能确定输出的值。N2的输入为0和Q。因为不知道Q的值，所以也不能确定输出的值。我们在这里被卡住了。这很像耦合的反相器。但是我们知道Q必须为0或者为1。于是，可以通过考察每一种子情况的方法来解决这个问题。

• 情况Ⅳa：Q = 0

因为S = 0, Q = 0，则N2在$\bar{Q}$上的输出为1，如图3-4（a），现在N1的输入$\bar{Q}$为1，于是它的输出Q = 0，这和原来假设是一致的。

• 情况Ⅳb：Q = 1

因为Q = 1，则N2在$\bar{Q}$上的输出为0，如图3-4（b），现在N1的两个输入R = 0，$\bar{Q}$= 1，于是它的输出Q = 1，这和原来假设是一致的。

综上所述，假设Q的初值已知，记为Q_{prev}。在我们进入情况Ⅳ之前，Q_{prev}为1或0表示了系

统的状态。当R = S = 0时，Q将保持初值Q_{prev}不变，$\bar{Q}$为其补$\overline{Q_{prev}}$。这个电路有记忆功能。

图3-5中的真值表总结了四种情况。输入S和R表示置位和复位。置位表示将一位设为1，复位表示将一位设为0。输出Q和$\bar{Q}$通常为互相取反的值。当R有效时，Q被清0，$\bar{Q}$= 1。当输入均为无效时，Q将保持初值Q_{prev}不变，R和S同时有效是没有意义的，因为锁存器不可能同时被置位或者复位，这样会产生两个输出为0的混乱电路响应。

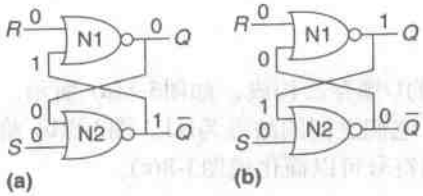


图3-4 SR锁存器的双稳态

情况	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

图3-5 SR锁存器的真值表

SR锁存器的符号表达式如图3-6。使用符号表示SR锁存器是抽象化和模块化的一个运用。有很多方法可以构造SR锁存器，包括使用不同的逻辑门或者晶体管。满足真值表3-5给定关系的电路组件都被称为SR锁存器，并采用图3-6中的符号表示。

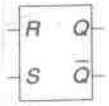


图3-6 SR锁存器的电路符号

和耦合的反相器一样，SR锁存器是一个在Q上存储一位状态的双稳态元件。但是Q的状态可以通过输入S和R控制。R有效时状态被复位为0，S有效时状态被置位为1。当S和R都无效时，状态保持初值不变。注意，输入的全部历史可以由状态变量Q解释。无论过去置位或复位是如何发生的，都需要通过最近一次置位或复位操作来确定SR锁存器的未来行为。

3.2.2 D锁存器

当SR锁存器中S和R同时取相同值时，其输出不确定，使用起来很不方便。而且，输入S和R混淆了时间和内容。输入有效时不仅需要确定是什么内容，而且需要确定将在何时改变。将内容和时间分开考虑会使电路设计将变得简单。如图3-7（a）所示的D锁存器解决了这些问题。它有两个输入：数据输入D用以控制下一个状态的值；时钟输入CLK用以控制状态发生改变的时间。

我们同样通过图3-7(b) 中的真值表分析D锁存器。为了方便，我们先考虑外部的节点$\bar{D}$，S和R。如果CLK = 0，则S = R = 0，D的值无任何意义。如果CLK=1，根据D的不同取值，一个与门输出为1，而另外一个与门输出为0。给定S和R，Q和$\bar{Q}$的值可以根据图3-5确定。注意到CLK = 0时，Q将保持要求原来的值Q_{prev}不变；当CLK = 1，Q = D。在所有的情况中，$\bar{Q}$的值始终是Q的取反，符合逻辑。D锁存器避免了S和R同时有效而造成的奇怪情况。

综上所述，时钟可以控制数据通过锁存器。当CLK = 1时，D锁存器是透明的(transparent)：数据D通过D锁存器流向Q，D锁存器就像是一个缓冲器。当CLK = 0时，D锁存器是阻塞的（opaque)：它阻止了新的数据D通过D锁存器流向Q，Q保持原来的值不变。所以D锁存器又是被称为透明锁存器或者电平敏感锁存器。D锁存器的电路符号如图3-7(c)所示。

当CLK = 1时，D锁存器不断更新它的状态。在本节的后面部分将看到，如何在特定的时刻更新状态。下一节将介绍D触发器。

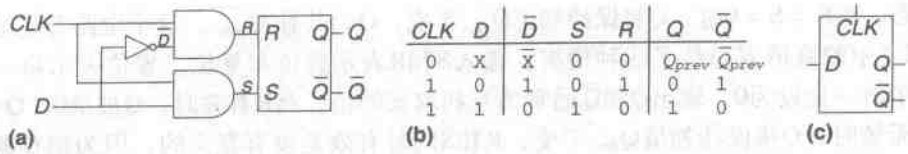


图3-7 D锁存器：(a) 原理图；(b) 真值表；(c) 电路符号

3.2.3 D触发器

一个D触发器可以由两个反相时钟控制的背靠背的D锁存器构成，如图3-8(a) 所示。锁存器L1称为主锁存器。第二个锁存器L2称为从锁存器。它们之间的结点为N1。图3-8(b) 给出了D触发器的电路符号。当不需要输出$\bar{Q}$时，D触发器的符号可以简化成图3-8(c)。

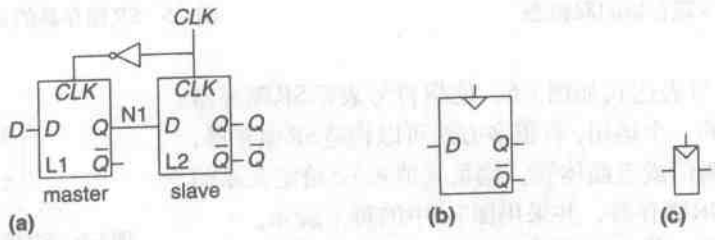


图3-8 D触发器：(a) 原理图；(b) 电路符号；(c) 简化电路符号

当CLK = 0主锁存器是通路，从锁存器是断路。所以D的值将无条件传送到N1。当CLK=1，主锁存器变成断路，从锁存器变成通路。N1的值将传送到Q，但是N1和D之间被切断。所以，时钟从0上升到1之前的D值在时钟变为1之后立即被复制到Q。在其他任何时刻，因为总有一个断开的锁存器来阻断D到Q的通路，Q将保持原来的值不变。

换句话说，D触发器在时钟上升沿时将D值复制到Q，在其他时间D触发器保持原有的状态。一定要记住这个定义，一个刚入门的数字设计师经常忘掉了触发器的功能。时钟的上升沿也经常简称为时钟沿（clock edge）。输入D确定了新的状态，时钟沿确定了状态发生改变的时间。

D触发器也常称为主从触发器（master-slave flip-flop），边沿触发器（edge-triggered flip-flop），或者正边沿触发器（positive edge-triggered flip-flop）。电路符号中的三角表示触发器使用时钟边沿触发。当不需要输出$\bar{Q}$时，经常被省略。

例3.1 计算触发器的晶体管数量

构成一个本节介绍的D触发器需要多少个晶体管？

解：构成一个与非门或者一个或非门需要用4个晶体管。一个非门需要用2个晶体管。一个与门可以由一个与非门和一个非门组成，所以，需要6个晶体管。一个SR锁存器需要用2个或非门，或8个晶体管。一个D锁存器由一个SR锁存器、2个与门和一个非门组成，即22个晶体管。D触发器由2个D锁存器和一个非门组成，即46个晶体管组成。3.2.7节中介绍了一种使用传输门的更高效CMOS实现方法。

3.2.4 寄存器

一个N位的寄存器由共享同一时钟CIK的一排N个触发器组成，所以寄存器的所有位同时

被更新。寄存器是组成时序电路的关键结构。图3-9中给出了其原理图和一个4位输入寄存器的电路符号，其输入是$D_{3:0}$，输出是$Q_{3:0}$，均为4位总线。

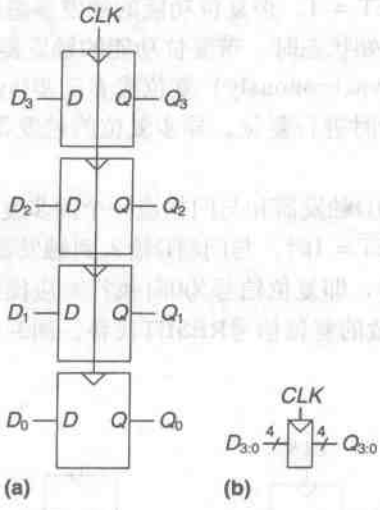


图3-9 4位寄存器：(a) 原理图；(b) 电路符号

3.2.5 带使能端的触发器

带使能端的触发器增加了另外一个输入EN（ENABLE），用于确定在时钟沿是否载入数据。当EN = 1时，带使能端的触发器和普通的D触发器一样。当EN = 0时，带使能端的触发器忽略时钟，保持原来的状态不变。当我们希望在某些特定时间（而不是在每一个时钟沿时）载入一个新值到触发器中时，带使能端的触发器非常有用。

图3-10给出了用一个D触发器和一个额外的门组成一个带使能端触发器的两种方法。在图3-10(a) 中，如果EN=1，一个输入多路器选择是否传递D的值。如果EN = 0，再次循环Q原来的状态。在图3-10(b) 中，时钟被门控（gated）。如果EN = 1，CLK像通常一样作为开关来控制触发器。如果EN = 0，CLK = 0，则触发器保持原来的值不变。注意，当CLK = 1时，EN不能改变，以免触发器出现一个时钟脉冲（在不正确的时间变化）。一般而言，在时钟上设置逻辑不是一个好的主意。时钟门控可能使时钟延迟而导致时序错误，我们将在3.5.3节中进一步介绍。只有当你确实知道要做什么时，才能这样做。带使能端的触发器电路符号如图3-10(c)所示。

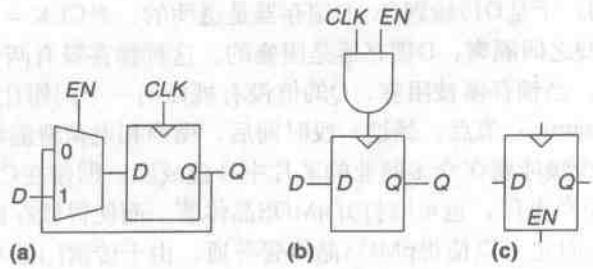


图3-10 带使能端的触发器：(a, b) 原理图；(c) 电路符号

3.2.6 带复位功能的触发器

带复位功能的触发器增加了一个输入RESET。当RESET=0时，带复位功能的触发器和普通的D触发器一样。当RESET = 1，带复位功能的触发器忽略D并且将输出Q复位为0。当系统加电需要将触发器设置为已知状态时，带复位功能的触发器十分有用。

触发器可能是异步（asynchronously）复位或者同步(synchronously)复位。同步复位功能的触发器仅仅在时钟上升沿时进行复位。异步复位的触发器只要RESET=1就可以对它进行复位操作，而与时钟无关。

图3-11(a) 显示了如何用D触发器和与门构造一个同步复位触发器。当 $\overline{\text{RESET}}$ = 0时，与门将0输入到触发器。当 $\overline{\text{RESET}}$ = 1时，与门将D输入到触发器。在这个例子中，$\overline{\text{RESET}}$ 是一个低电平有效(active low)信号，即复位信号为0时执行对应操作功能。增加一个反相器，电路中的 $\overline{\text{RESET}}$ 可以用高电平有效的复位信号RESET代替。图3-11(b) 和图3-11(c) 是高电平有效复位触发器的符号。

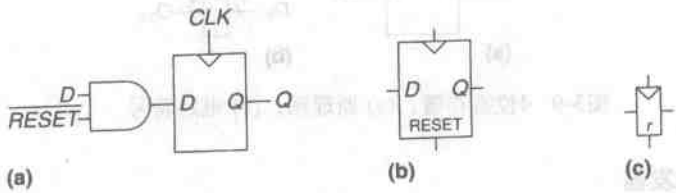


图3-11 同步复位触发器：(a) 原理图；(b, c)电路符号

异步复位的触发器需要调整触发器的内部结构，留在习题3.10中。它们也是设计师们经常要用到的基本元件。

带置位功能的触发器也偶尔被使用。置位时，触发器被置为1。它们以同步或异步的方式完成置位操作。带置位和复位功能的触发器可以带有使能输入端，也可以组成*N*位寄存器。

3.2.7 晶体管级的锁存器和触发器设计*

例3.1说明通过逻辑门构造锁存器和触发器需要大量的晶体管。但是锁存器的基本作用是穿透或者阻塞，类似一个开关。回忆1.7.7节内容，传输门是构成CMOS开关的高效方法，可以利用传输门的优点来减少晶体管的数量。

一个简捷的D锁存器可以用一个传输门构成，如图3-12（a）所示。当CLK = 1和 $\overline{\text{CLK}}$ = 0时，传输门是开放的，于是D传输到Q，D锁存器是透明的。当CLK = 0和 $\overline{\text{CLK}}$ = 1时，传输门是关闭的，于是D和Q之间隔离，D锁存器是阻塞的。这种锁存器有两个主要缺点：

- 输出节点浮空。当锁存器被阻塞，Q的值没有被任何一个门钳住为浮空（floating）节点或者动态（dynamic）节点。经过一段时间后，噪声和电荷泄漏将扰乱Q的值。
- 没有缓冲。缺少缓冲将在众多商业的芯片中导致故障。即使在CLK=0时，如果噪声尖峰将D值拉成一个负电压，也可以打开nMOS晶体管，而使得锁存器导通。同样，当D的噪声尖峰超过V_{DD}时也可以使得pMOS晶体管导通。由于传输门是对称的，所以这使得在Q上的噪声可能反向驱动而影响输入D。而一般的规则是传输门的输入或时序电路的状态节点都不应暴露在有噪声的外部世界中。

图3-12(b) 是在现代商业芯片中由12个晶体管构成的一个D锁存器。它也可以用时钟控制的传输门来构成，但是它增加了反相器I1和I2作为对输入和输出的缓冲。锁存器的状态被钳在节点N1上。反相器I3和三态缓冲T1，给N1提供反馈，使N1的值固定。当CLK = 0时，如果在N1上产生一个小的噪声，T1将驱动N1回到有效的逻辑值。

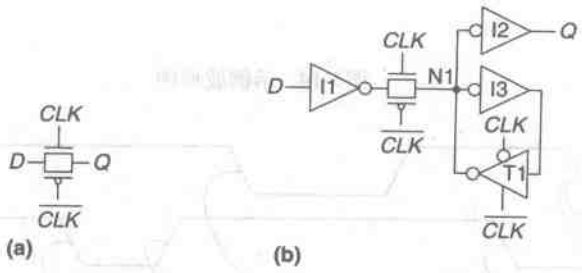


图3-12 D锁存器原理图

图3-13中D触发器可以通过两个由CLK和 $\overline{CLK}$ 控制的静态锁存器构成，同时去除了多余的内部反相器，所以这个触发器仅需要20个晶体管。

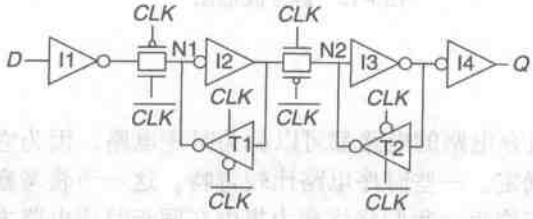


图3-13 D触发器原理图

3.2.8 小结

锁存器和触发器是构成时序电路的基本模块。记住：D锁存器是电平敏感的，D触发器是边沿触发的。当CLK=1，D锁存器可以穿透，允许输入D传输到输出Q。D触发器在时钟边沿时刻将D值复制到Q。在其他时候，锁存器和触发器保持原来的状态不变。寄存器由共享同一个CLK信号的多个D触发器构成。

例3.2 触发器和锁存器比较

如图3-14所示，Ben在一个D锁存器和一个D触发器上给定D和CLK输入。帮助Ben确定每一种设计下Q的输出值。

解：图3-15给出了输出波形。假设在相应输入值变化时，输出Q上有一个小的延迟。箭头表示导致输出改变的原因。Q的起始值未知，可能是0或者1，用一对水平线表示。首先考虑D锁存器。在第一个CLK的上升沿，D = 0，所以Q肯定变成0。当CLK = 1，每一次D的改变都会导致Q的改变。当CLK = 0，D改变，而Q不变。接着考虑D触发器。在每一个CLK时钟上升沿到来时，D被复制到Q。在其他时间，Q保持原来的状态不变。

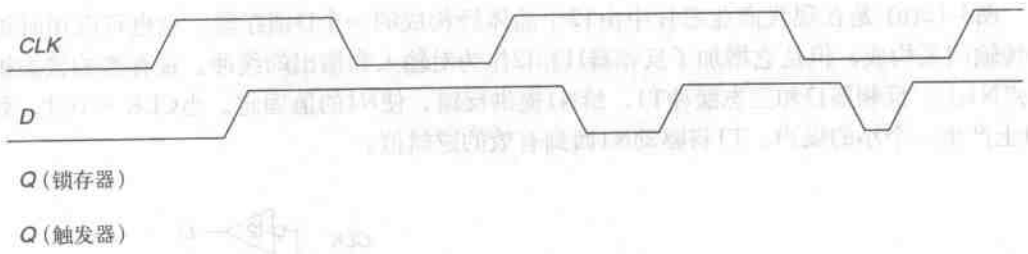


图3-14 示例波形图

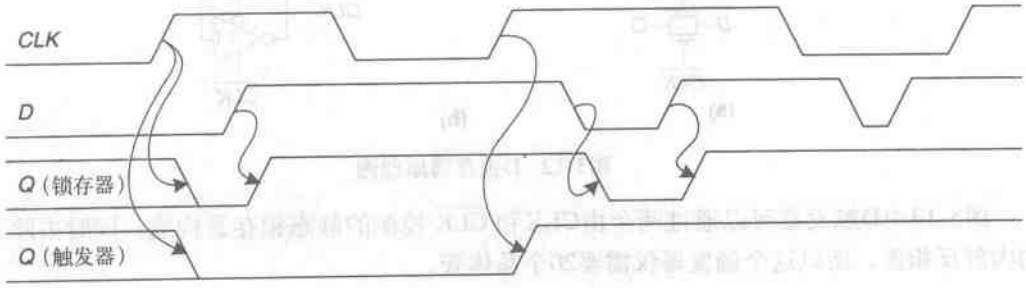


图3-15 解答波形图

3.3 同步逻辑设计

一般而言，所有不是组合电路的电路都可以称为时序电路，因为它们的输出不能简单地通过观察当前输入的值来确定。一些时序电路比较奇特。这一节将考察这些电路，然后介绍同步时序电路的概念和动态约束。我们将注意力集中在同步时序电路方面，将找到一种简易而系统的时序电路系统设计和分析方法。

3.3.1 一些有问题的电路

例3.3 非稳态电路

Alyssa遇到了三个设计很拙劣的反相器，它们以环状连接在一起，如图3-16所示。第三个反相器的输出反馈到第一个反相器的输入。每一个反相器都有1ns的传输延迟。确定电路的功能。

解：假设节点X的起始值是0。这使得Y = 1，Z = 0，所以X = 1，这和我们刚开始的假设不一致。这个电路没有稳定的状态，称为不稳定态（unstable）或者非稳态(astable)。电路的行为如图3-17所示。如果X在0时刻上升，Y将在1ns时刻下降，Z将在2ns时刻上升，X将在3ns时刻下降。接着，Y在4ns时刻上升，Z将在5ns时刻下降，X将在6ns时刻再次上升，这个模式将一直重复下去。每一个节点以6ns为周期在0和1之间摆动。这个电路称为环形振荡器（ring oscillator）。

环形振荡器的周期取决于反相器的传输延迟。这个延迟时间又取决于反相器的制造工艺、电源电压、甚至工作温度等诸多因素。所以环形振荡器的周期很难准确预测。简而言之，环形振荡器是无输入和一个周期性改变输出的时序电路。

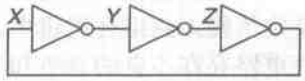


图3-16 三个反相器构成的环

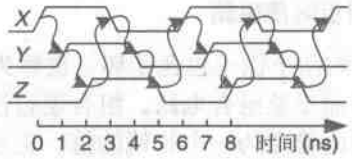


图3-17 环形振荡器波形图

例3.4 竞争条件

Ben设计了一个新的D锁存器，并宣布这个设计比图3-7中的D锁存器要好。因为在这个电路中门的数量更少。他写出了真值表以确定输出Q的值，其中包括D和CLK两个输入和锁存器的原始状态Q_{prev}。根据这个真值表，Ben得出了布尔等式。他通过输出Q反馈得到了Q_{prev}，设计如图3-18所示。他的锁存器工作是否正确，不考虑每一个门的延迟。

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$

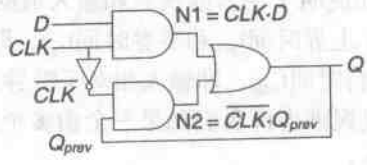


图3-18 一个看似改进的D锁存器

解：如图3-19所示，当某些门比其他的门速度慢时，电路有一个竞争情况将导致电路错误。假设CLK = D = 1。锁存器可以被穿透，将D传送到Q，Q = 1。现在CLK下降。锁存器将保持原来的状态，Q = 1。但是，假设从CLK到 $\overline{CLK}$ 通过反相器的延迟比与门和或门的延迟要长。所以在 $\overline{CLK}$ 上升之前，N1和Q将同时下降。在这种情况下，N2将不能上升，Q值将被钳为0。

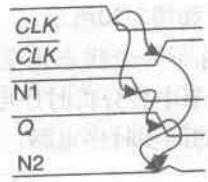


图3-19 描述竞争条件的锁存器波形图

这是一个输出直接反馈到输入的异步电路设计例子。异步电路中经常会出现竞争条件而难以掌握。之所以出现竞争条件，是因为其电路的行为取决于两条通过逻辑门的路径中哪条更快。这样的电路可能可以工作，但是对于从表面上看是相同的电路，如果用几个延迟稍微不同的门替换，可能就无法正常工作。或者，这样的电路只能在一定的温度和电压下正常工作，只有在这个特定的条件下其逻辑门的延迟才刚好正确。这种错误是极其难以被查出的。

3.3.2 同步时序电路

前面的两个例子包含了环，也称为环路（cyclic path），其输出直接反馈到输入。它们是时序电路而不是组合电路。组合逻辑没有环路和竞争。特定值输入到组合逻辑中，输出将在传输延迟内稳定为一个正确的值。但是，包含环路的时序电路存在不良的竞争和不稳定的动作。分析这样的电路十分耗时，很多聪明的人都会犯错误。

为了避免这些问题，设计师们在环路中插入寄存器以断开环路。这将电路转变成了组合逻辑电路和寄存器的组合。寄存器包含系统的状态，这些状态仅仅在时钟沿到达时发生改变，所以状态同步于（synchronized）时钟信号。如果时钟足够慢，在下一个时钟沿到达之前输入到寄存器的信号都可以稳定下来，所有的竞争都被消除。根据在反馈环路上总是使用寄存器的原则，可以得到同步时序电路的一个规范定义。

通过电路的输入、输出端、功能和时序说明可以定义一个电路。一个时序电路有一组有限的离散状态{S0，S1，…，S^{k-1}}。同步时序电路（synchronous sequential circuit）有一个时钟输入，它的上升沿表示时序电路状态发生转变的时间。我们经常使用当前状态（current state）和下一个状态（next state）两个术语来区别目前系统的状态和下一个时钟沿系统将进入的状态。功能说明详细说明了在当前状态和输入值的各种组合下，此电路的下一个状态和输出值。时序说明包括了上界时间t_{pcq}和下界时间t_{ccq}，即从时钟的上升沿直到输出发生改变的时间；建立时间t_{setup}和保持时间t_{hold}，即输入相对于时钟上升沿的稳定时间。

同步时序电路的组成规则告诉我们如果一个由多个元件连接构成的电路是同步时序电路，它必须需要满足以下条件：

- 每一个电路元件是寄存器或组合电路；
- 至少有一个电路元件是寄存器；
- 所有寄存器接收同一个时钟信号；
- 每一个环路至少包含一个寄存器。

非同步时序电路称为异步（asynchronous）电路。

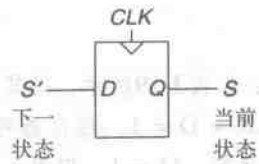


图3-20 触发器当前状态和下一状态

单个触发器是一个最简单的同步有0或1两种状态的时序电路。它包含了一个输入D，一个时钟CLK，一个输出Q。一个触发器的功能规范是：下一个状态是D，输出Q是当前的状态，如图3-20所示。

我们经常称当前状态为变量S，称下一个状态为变量S′。在这种情况下，S之后的撇号表示下一个状态，而不是取反的值。3.5节中将分析时序电路的时序关系。

有限状态机和流水线是两种常见的同步时序电路，将在后面的章节介绍。

例3.5 同步时序电路

图3-21中哪些电路是同步时序电路？

解：电路（a）是组合逻辑电路，不是时序逻辑电路，因为它没有一个寄存器。电路（b）是一个不带反馈回路的简单时序电路。电路（c）既不是组合电路也不是时序电路，因为它有一个锁存器，这个锁存器既不是寄存器也不是组合逻辑电路。电路（d）和电路（e）是同步时序逻辑电路，它们是有限状态机的两种形式，这将在3.4节中讨论。电路（f）既不是组合电路也不是时序电路，因为它有一个从组合逻辑电路的输出端电路反馈到同一逻辑电路输入端的回路，但是在回路上没有寄存器。电路（g）是同步时序逻辑电路的流水线形式，将在3.6节中讨论。电路（h）严格地说不是一个同步时序电路，因为两个寄存器的时钟信号不同，它们之间有两个反相器的延迟。

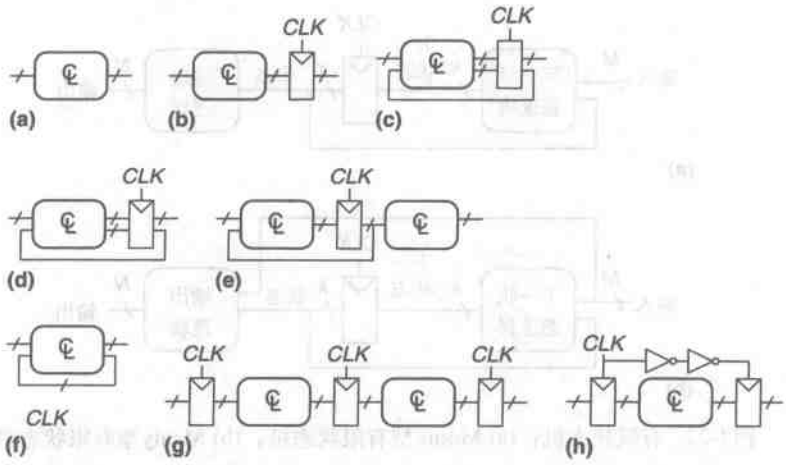


图3-21 例子电路

3.3.3 同步和异步电路

从理论上讲，异步电路设计要比同步电路设计更为通用，因为系统时序不受时钟控制的寄存器所约束。正如能使用任意电压的模拟电路比数字电路更为通用，能够使用各种反馈的异步电路似乎比同步电路具有更强的通用性。然而，同步电路比异步电路更加容易设计，就像数字电路比模拟电路更加容易。尽管异步电路进行了数十年的研究，但实际上几乎所有的系统本质上都是同步的。

当然，在两个不同时钟的系统之间进行通信时，或者在任意的时刻接收输入时，异步电路也很重要。类似地，模拟电路在连续电压的真实世界中通信很重要。此外，对异步电路的研究将产生一些有趣的知识，其中一些也有利于改进同步电路。

3.4 有限状态机

同步时序电路可以描绘成图3-22的形式，称为有限状态机（Finite State Machine, FSM）。这个名字源于具有k位寄存器的电路可以处于2^k种状态中的某一种唯一状态。一个有限状态机有M位输入，N位输出和k位状态，同时还具有一个时钟信号和一个可选的复位信号。有限状态机包含两个组合逻辑块：下一状态的逻辑（next state logic）和输出逻辑（output logic），以及一组用于存储状态的寄存器。在每一个时钟沿，有限状态机进入下一状态。这个下一状态是根据当前的状态和输入值计算出来的。根据有限状态机的特点和功能，通常被分为两类有限状态机：Moore型有限状态机的输出仅仅取决于当前的状态；Mealy型有限状态机中的输出取决于当前的状态和输入值。有限状态机提供了系统的方法来设计给定功能说明的同步时序逻辑电路。这一章的后续部分将由一个实例开始介绍这种方法。

3.4.1 有限状态机设计实例

为了解释有限状态机的设计，我们考虑在校园中繁忙的十字路口建立一个交通灯控制器。工程系学生在宿舍和实验室之间的Academic大街上漫步。他们正在阅读关于有限状态机的教科书，而没有看前面的路。足球运动员们正喧嚷地拥挤在运动场和食堂间的Bravado大道上。他们正在向前和向后投球，也没有看前面的路。在两条大道上的十字路口发生了一些严重的事故。为了防止事故再次发生，系主任要求Ben安装一个交通灯。

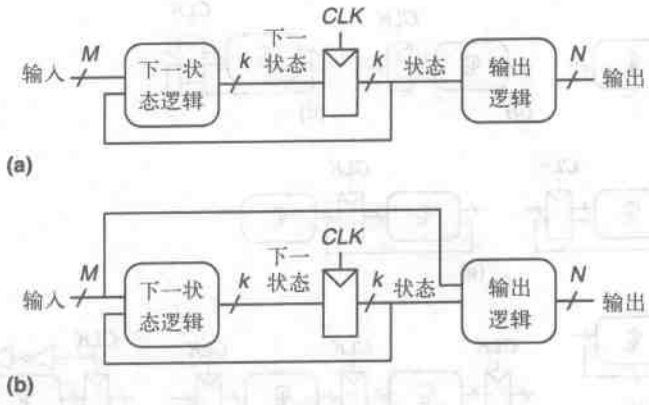


图3-22 有限状态机：(a) Moore型有限状态机；(b) Mealy型有限状态机

Ben决定用有限状态机来解决这个问题。他分别在Academic大道和Bravado大街上安装了2个交通传感器T_A和T_B。传感器输出1时，表示此大道上有学生；输出0时，表示大道上没有人。Ben又安装了2个交通灯L_A和L_B来控制交通。每一个灯接收数字信号输入，以确定显示绿色、黄色或红色。所以，有限状态机有T_A和T_B两个输入，L_A和L_B两个输出。十字路口的灯和传感器如图3-23所示。Ben采用了一个周期为5s的时钟。在每一个时钟上升沿，灯将根据传感器改变。同时，Ben还设计了一个复位按键以便技术员在打开交通信号灯时将控制器设置为一个已知的起始状态。状态机的黑盒视图如图3-24所示。

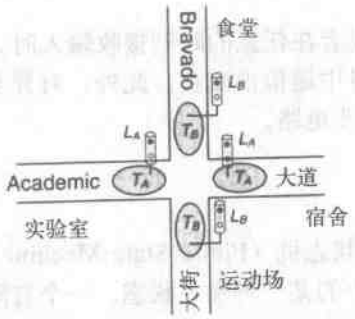


图3-23 校园地图

图3-24 有限状态机的黑盒视图

Ben的第二步是画出状态转换图（state transition diagram），如图3-25所示。此图说明了系统中每一种可能的状态和两种状态之间的转换。当系统复位时，Academic大道上的灯是绿色，Bravado大街上的灯是红色。每隔5秒钟，控制器检查交通模式并决定下一步该如何处理。此时，若在Academic大街有人，灯就不再改变；没有人时，此大道上的灯变成黄色并保持5s钟，然后再变成红色，同时Bravado大街上的灯变成绿色。同样，在Bravado大街上有人时，此大道上的灯保持绿色，然后变成黄色，最后变成红色。

在状态转换图中，圆圈代表状态，圆弧代表两种状

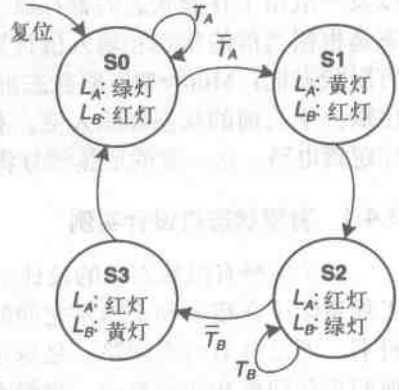


图3-25 状态转移图

态之间的转换。转换发生在时钟上升沿。时钟总是出现在同步时序逻辑电路中，而且仅仅控制何时发生状态转移，所以为简化起见，在状态转移图中不画出时钟。标有Reset指向的圆弧从外部进入状态S0，这说明不管当前是什么状态，系统复位时都应该进入状态S0。如果一个状态有多个离开它的圆弧，每个圆弧标有触发这个状态转换的输入条件。例如在S0状态，如果T_A为真，系统将保持当前状态；如果T_A为假，将转换成S1状态。如果状态只有一个离开它的圆弧，不管输入是什么，转换都会发生。例如在S1状态，系统将总是转换到S2状态。特定状态下的输出值也在状态中给出。例如S2状态下，L_A是红色，L_B是绿色。

Ben将状态转换图重写为状态转换表（state transition table），如表3-1所示。它说明了每一种状态和输入值所产生的下一个状态S′。表中使用了无关项（X），这表示下一个状态并不依赖于特定的输入。同时，在表中忽略了复位。实际上，可以使用带复位功能的触发器使得复位后总是进入S0状态，而不用考虑输入。

表3-1 状态转移表

当前状态S	输入		下一状态	当前状态S	输入		下一状态
	T_A	T_B	S′		T_A	T_B	S′
S0	0	X	S1	S2	X	0	S3
S0	1	X	S0	S2	X	1	S2
S1	X	X	S2	S3	X	X	S0

状态转换图中使用了抽象的状态标记{S0，S1，S2，S3}和输出标记{red，yellow，green}。为建立一个真实的电路，状态和输出必须按照二进制编码。Ben选择了简单的编码方式，如表3-2和3-3所示。每种状态和输出都编码成2位：$S_{1:0}$，$L_{A1:0}$和$L_{B1:0}$。

表3-2 状态编码

状态	$S_{1:0}$的编码
S0	00
S1	01
S2	10
S3	11

表3-3 输出编码

输出	$L_{1:0}$的编码
绿色	00
黄色	01
红色	10

Ben接着用二进制编码更新状态转换表，如表3-4所示。这个改进的状态转移表是一个可以确定下一个状态的逻辑真值表，定义了根据当前状态S和输入确定下一个状态S′的逻辑函数。改进的输出表是一个确定输出逻辑的真值表，定义了根据当前状态S确定输出L_A和L_B的逻辑函数。

表3-4 二进制编码的状态转移表

当前状态		输入		下一状态	
S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

通过这些表，可以直接读出下一个状态的与或式布尔表达式。

$$S_1' = \overline{S_1}S_0 + S_1\overline{S_0}\,\overline{T_B} + S_1\overline{S_0}T_B$$
$$S_0' = \overline{S_1 S_0 T_A} + \overline{S_1 S_0 T_B} \tag{3-1}$$

这些式子可以用卡诺图化简，但是使用观察方法化简更容易。例如，在S_1'的等式中的T_B和$\overline{T}_B$项明显是多余的。所以S_1'可以化简成一个异或操作。式3.2给出了下一个状态的逻辑表达式。

$$S_1' = S_1 \oplus S_0$$
$$S_0' = \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B \tag{3-2}$$

同样，Ben针对每一个状态指出的输出写出了输出逻辑表，如表3-5所示。也可以直接读出这些输出的化简布尔表达式。例如，可以发现仅仅在S_1的行为真时L_{A1}才为真。

表3-5 输出逻辑表

当前状态		输出			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S}_1 S_0$$
$$L_{B1} = \overline{S}_1$$
$$L_{B0} = S_1 S_0 \tag{3-3}$$

最后，Ben以图3-22(a) 的形式绘制了Moore型有限状态机的电路图。首先，画了一个2位的状态寄存器，如图3-26(a) 所示。在每一个时钟沿，这个状态寄存器复制下一状态S'到状态$S_{1:0}$。状态寄存器在启动时收到一个同步或异步复位信号，初始化有限状态机。然后，根据等式3-2画出下一状态逻辑的电路图，这部分逻辑根据当前的状态和输入值计算出下一个状态的值，如图3-26(b) 所示。最后，根据等式3-3画出输出逻辑的电路图，这部分逻辑根据当前的状态计算输出值，如图3-26(c) 所示。

图3-27给出了解释交通灯控制器一系列状态运行的时序图。图中显示了CLK、复位、输入T_A和T_B、下一个状态S'和当前状态S、输出L_A和L_B。箭头表明了因果关系，例如，当前状态的改变导致输出的改变，输入的改变导致下一个状态的改变。虚线表示在CLK的上升沿状态改变。

时钟的周期是5s，所以交通灯最多每5s改变一次。当这个有限状态机第一次启动，它的状态是未知的，如图3-27中的问号所示。所以系统应被复位到一个已知的状态。在这个时序图中，S被立即复位成S0，说明使用了带复位功能的异步触发器。在状态S0，灯L_A是绿色，灯L_B是红色。

此例中，复位后Academic大道上已经有人。所以，控制器保持在S0状态，并保持灯L_A是绿色。此时Bravado大街上尽管有人到达，也将开始等待。15s之后，在Academic大道上的人都通过了，T_A开始下降。在随后的时钟沿，控制器将进入S1状态，灯L_A变成黄色。在下一个

5s之后，控制器进入S2状态，灯L_A变成红色，L_B变成绿色。控制器在状态S2上等待，直到在Bravado大街上的人都通过了。它将进入状态S3，L_B变成黄色。5s之后，控制器将进入状态S0，L_B变成红色，灯L_A变成绿色。

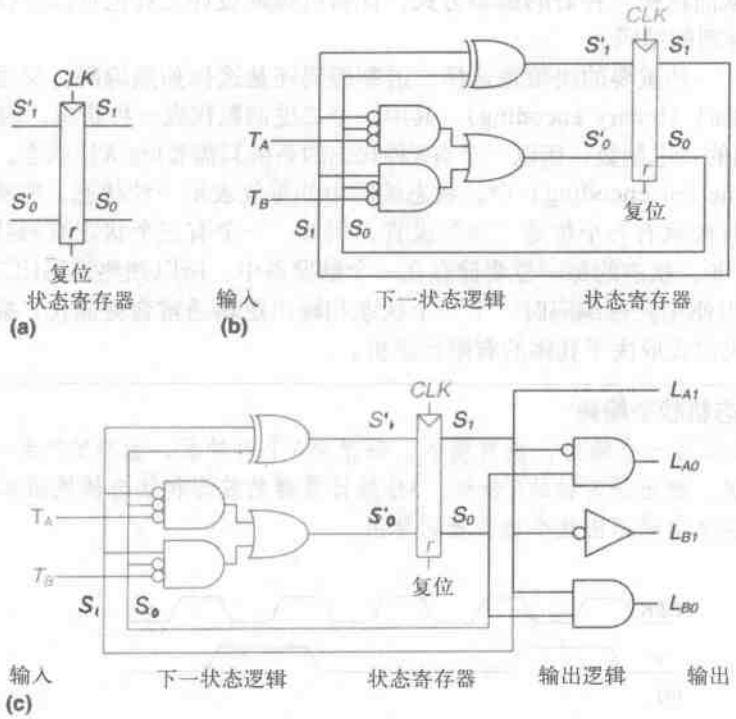


图3-26 交通控制器的状态机电路图

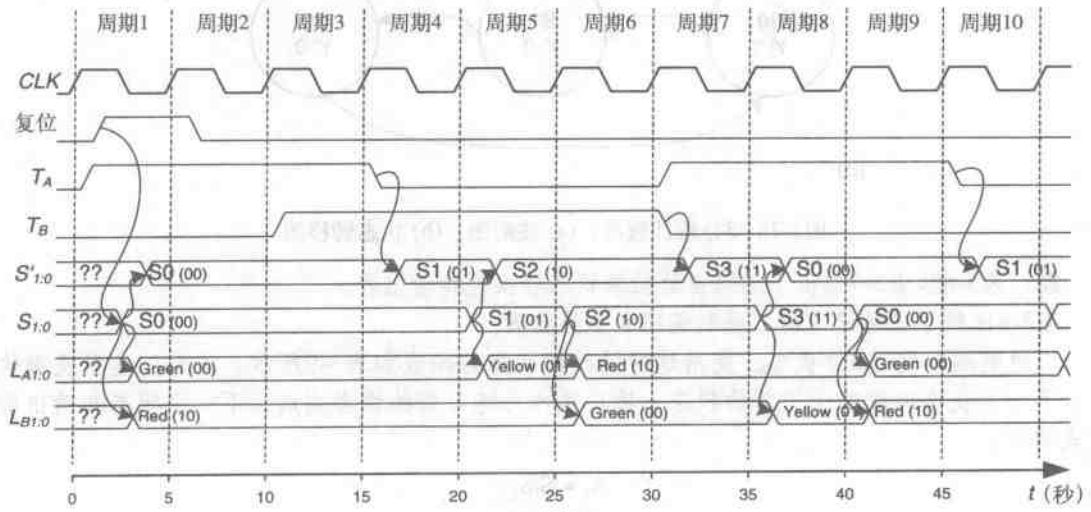


图3-27 交通控制器的时序图

3.4.2 状态编码

在先前的例子中，状态和输出的编码是任意选择的。不同的选择将产生不同的电路。一

个自然而然的问题是，如何确定一种编码，使之能够产生一个逻辑门数量最少，传输延迟最短的电路。不幸的是，没有一种简单的方法可以找出最好的编码。现有的方法是尝试所有的可能情况，但当状态的数量很大时这也是不可行的。往往可以通过观察的方法使相关的状态或输出共享某些位从而选择一种好的编码方式。计算机辅助设计工具也可以寻找可能的编码集合，并选择一种合理的编码。

在状态编码中，一种重要的决策是选择二进制编码还是选择独热编码。交通灯控制的例子中使用了二进制编码（binary encoding），其中一个二进制数代表一种状态。因为可以$\log_2 K$位可以表示K个不同的二进制数，所以一个有K种状态的系统只需要$\log_2 K$位状态。

在独热编码（one-hot encoding）中，状态编码中的每位表示一种状态。它被称为独热编码，是因为在任何时候只有一个位是“热”或真。例如，一个有三个状态有限状态机的独热编码为001，010和100。状态的每一位要储存在一个触发器中，所以独热编码比二进制编码需要更多的触发器。但使用独热编码时，下一个状态和输出逻辑通常会更简化，需要的门电路也更少。最佳的编码方式取决于具体的有限状态机。

例3.6 有限状态机状态编码

一个N分频计数器有一个输出，没有输入。每循环N个时钟后，输出Y产生一个周期的高电平信号。换句话说，输出是时钟的N分频。3分频计数器的波形和状态转换图如图3-28所示。使用二进制编码和独热编码画出这个计数器的草图。

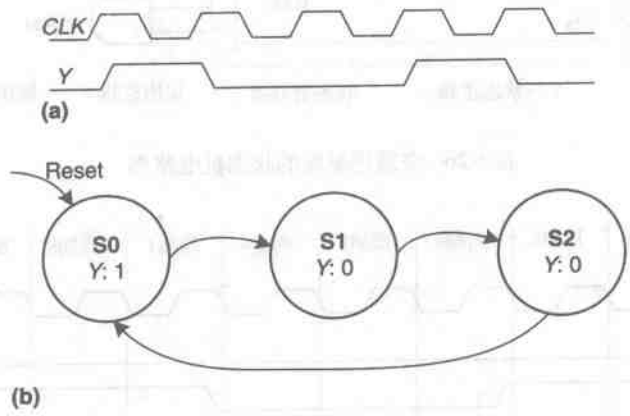


图3-28 3分频计数器：(a) 波形图；(b) 状态转移图

解：表3-6和表3-7给出了编码前的抽象状态转换表和输出表。

表3-8比较了三种状态的二进制编码和独热编码。

二进制编码使用2位状态。使用这种编码的状态转换表如表3-9所示。注意，这里没有输入，下一个状态只取决于当前的状态。输出表作为练习留给读者完成。下一个状态和输出的等式如下：

$$S_1' = \overline{S}_1 S_0$$

$$S_0' = \overline{S}_1 \overline{S}_0 \tag{3-4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3-5}$$

独热编码使用3位状态。这种编码的状态转换表如表3-10所示，输出表作为练习留给读者

完成。下一个状态和输出的等式如下：

$$S_2' = S_1$$
$$S_1' = S_0$$
$$S_0' = S_2 \tag{3-6}$$
$$Y = S_0 \tag{3-7}$$

图3-29给出了每一种设计的原理图。二进制编码设计的硬件可以通过使Y和S_0'共享相同的门电路进行优化。同样，独热编码设计需要可置位和可复位的触发器在复位时对状态机的S0进行初始化。最好的实现选择取决于门电路和触发器的相对成本，独热编码设计在这个特定例子中更加可取。

表3-6 3分频计数器状态转移表

当前状态	下一状态
S0	S1
S1	S2
S2	S0

表3-7 3分频计数器输出表

当前状态	输出
S0	1
S1	0
S2	0

表3-8 3分频计数器的二进制编码和独热编码

Stats	独热编码			二进制编码	
	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

表3-9 二进制编码的状态转移表

当前状态		下一状态	
S_1	S_0	S_1'	S_0'
0	0	0	1
0	1	1	0
1	0	0	0

表3-10 独热编码的状态转移表

当前状态			下一状态		
S_2	S_1	S_0	S_2'	S_1'	S_0'
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

一种相关的编码方式是独冷编码，K个状态通过K位表示，其中的一位恰好为FALSE。

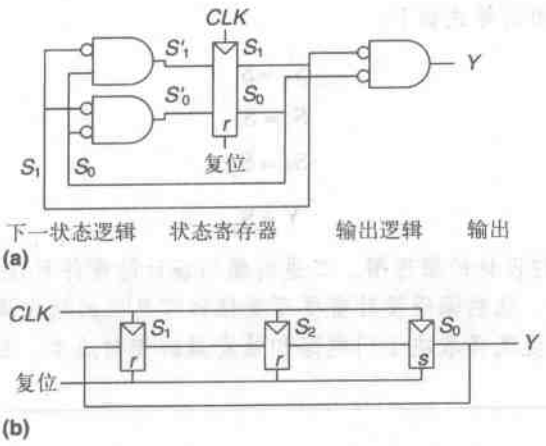


图3-29 3分频计数器电路：(a) 二进制编码；(b) 独热编码

3.4.3 Moore型状态机和Mealy型状态机

迄今为止，我们已经介绍了Moore型状态机的例子，其输出只取决于系统的状态。所以在Moore型状态机的状态转换图中，输出被标在圆圈内。Mealy型状态机和Moore型状态机很相似，但是输出取决于输入和当前的状态，所以在Mealy型状态机的状态转换表中，输出被标在圆弧上面而不是圆圈内。可以根据输入和当前状态通过组合逻辑模块计算出输出，如图3-22（b）所示。

例3.7 Moore型状态机和Mealy型状态机的对比

Alyssa有一个带限状态机大脑的机器宠物蜗牛。蜗牛沿着纸带从左向右爬行。这个纸带包含1和0的序列。在每一个时钟周期，蜗牛爬行到下一位。蜗牛从左到右爬行在纸带上，如果最后经过的4位是1101时，蜗牛会高兴得笑起来。设计一个有限状态机计算蜗牛何时会发笑。蜗牛触角下面的位是输入A。当蜗牛发笑时，输出的Y为TURE。比较Moore型状态机和Mealy型状态机设计。画出每种设计的原理图以及包含输出状态和输入的时序图。假设蜗牛爬行的序列是111011010。

解：Moore型状态机需要5个状态，如图3-30(a) 所示，确信你自己的状态转换图是正确的。特别是，当输入为1时为什么从S4到S2要画一个圆弧？

与Moore型状态机相比，Mealy型状态机只需要4个状态，如图3-30(b) 所示。每一个圆弧被标注成A/Y，其中A是引起转换的输入值，Y是输出值。

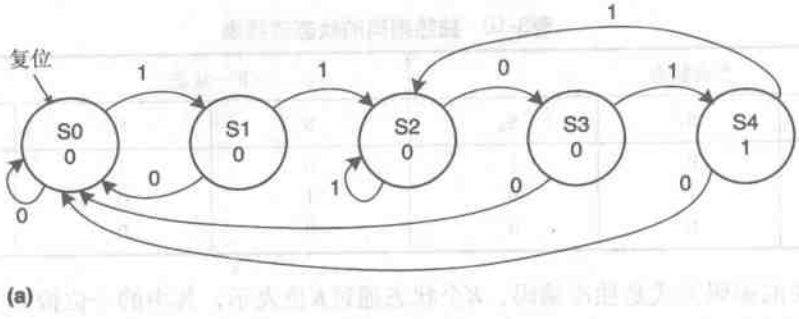


图3-30 有限状态机状态转移图：(a) Moore型有限状态机；(b)Mealy型有限状态机

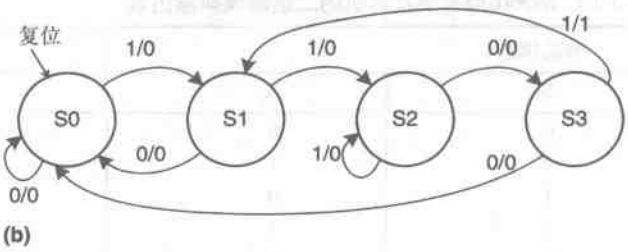


图3-30 （续）

表3-11和表3-12给出了Moore型状态机的状态转换图和输出表。Moore型状态机至少需要3位的状态。考虑使用以下二进制状态编码：S0=000，S1=001，S2=010，S3=011，S4=100。表3-13和表3-14重新写出了二进制状态编码的状态转换图和输出表。

表3-11 Moore型有限状态机状态转移表

当前状态S	输入A	下一状态S′
S0	0	S0
S0	1	S1
S1	0	S0
S1	1	S2
S2	0	S3
S2	1	S2
S3	0	S0
S3	1	S4
S4	0	S0
S4	1	S2

表3-12 Moore型有限状态机输出表

当前状态S	输出Y	当前状态S	输出Y
S0	0	S3	0
S1	0	S4	1
S2	0		

表3-13 Moore型有限状态机的二进制编码状态转移表

当前状态			输入	下一状态		
S_2	S_1	S_0	A	S_2'	S_1'	S_0'
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	0	1	0

表3-14 Moore型有限状态机的二进制编码输出表

当前状态			输出
S_2	S_1	S_0	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

对于这些表，可以通过观察找出下个状态和输出的表达式。注意，因为此状态机中不存在101，110和111等状态，所以可以进一步化简这些表达式。这些不存在的状态对应的下一状态和输出不用考虑（在表中没有显示），可以使用无关项来化简等式。

$$S'_2 = S_1S_0A$$
$$S'_1 = \overline{S}_1S_0A + S_1\overline{S}_0 + S_2A \tag{3-8}$$
$$S'_0 = \overline{S}_2\overline{S}_1\overline{S}_0A + S_1\overline{S}_0\overline{A}$$

$$Y = S_2 \tag{3-9}$$

表3-15给出了Mealy型状态机的状态转换和输出表。Mealy型状态机至少需要2位状态。考虑使用二进制编码：S0 = 00，S1 = 01，S2 = 10，S3 = 11。表3-16重新写出了二进制状态编码的状态转换图和输出表。

表3-15 Mealy型状态机的状态转移和输出表

当前状态S	输入A	下一状态S′	输出Y
S0	0	S0	0
S0	1	S1	0
S1	0	S0	0
S1	1	S2	0
S2	0	S3	0
S2	1	S2	0
S3	0	S0	0
S3	1	S1	1

表3-16 Mealy型状态机的二进制编码状态转移和输出表

当前状态		输入	下一状态		输出
S1	S0	A	S'_1	S'_0	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

对于这些表，可以通过观察得到下个状态和输出的表达式。

$$S'_1 = S_1\bar{S}_0 + \bar{S}_1 S_0 A \tag{3-10}$$

$$S'_0 = \bar{S}_1\bar{S}_0 A + S_1\bar{S}_0\bar{A} + S_1 S_0 A$$

$$Y = S_1 S_0 A \tag{3-11}$$

Moore型状态机和Mealy型状态机的电路原理图分别如图3-31(a) 和3-31(b) 所示。

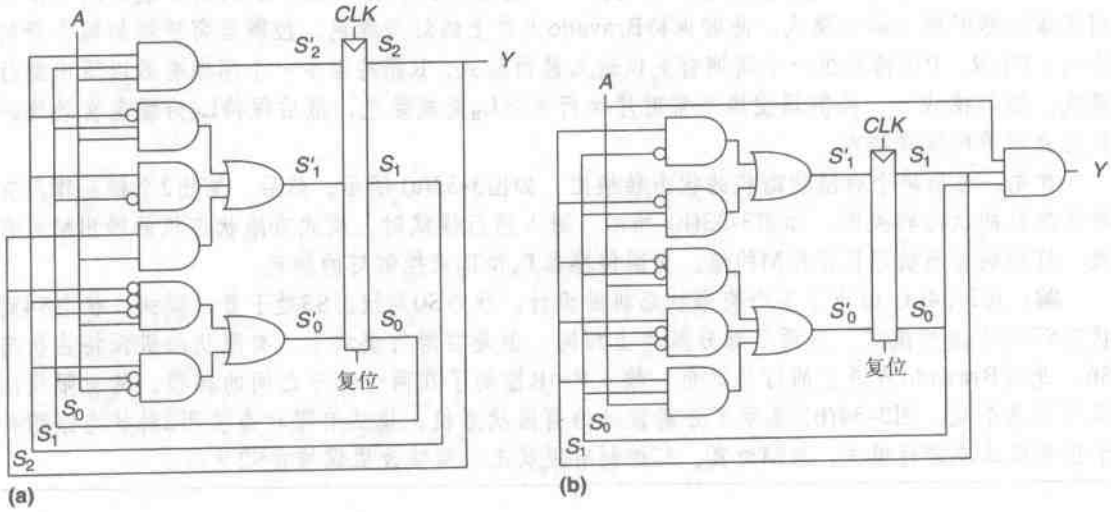


图3-31 有限状态机电路原理图：(a) Moore型状态机；(b) Mealy型状态机

Moore型状态机和Mealy型状态机的时序图分别如图3-32所示。两种状态机的状态序列有所不同。Mealy型状态机的输出上升要早一个周期。这是因为其输出直接响应输入，而不需要等待状态的变化。如果Mealy型状态机的输出通过触发器产生延迟，它的输出将和Moore型状态机一样。在选择有限状态机设计类型时，需要考虑何时输出响应。

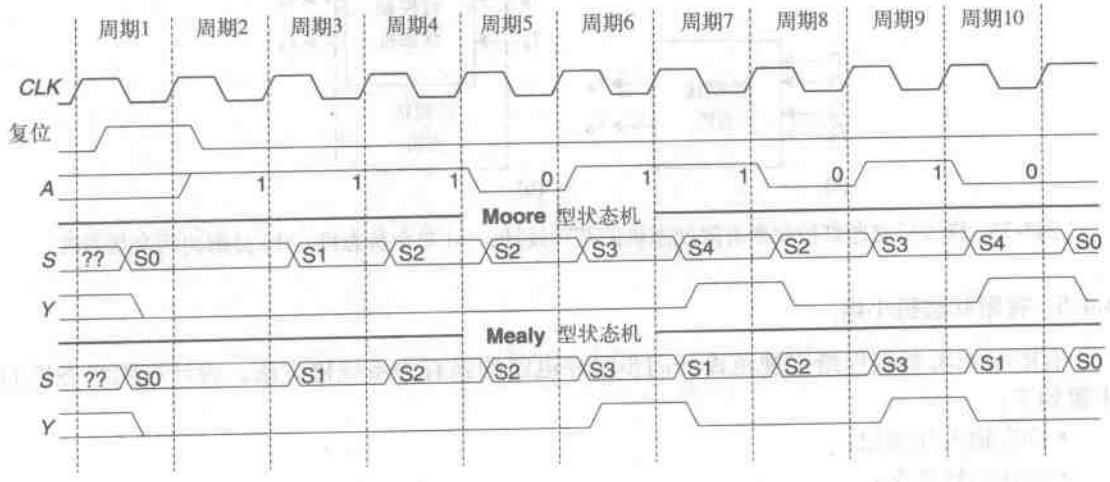


图3-32 Moore型状态机和Mealy型状态机的时序图

3.4.4 状态机的分解

简化复杂有限状态机设计的一个方法是将一个复杂的有限状态机分解成多个互相作用的简单状态机，其中一些状态机的输出是另外一些状态机的输入。这种层次结构和模块化的方法称为状态机的分解（factoring）。

例3.8 不分解的状态机和分解后的状态机

修改3.4.1节中的交通灯控制器，以增加一个游行模式。当观众和乐队以分散的队形漫步到足球比赛时进入游行模式，此时保持Bravado大道上的灯是绿色。控制器需要增加两个新的输入：P和R。P保持至少一个周期有效以进入游行模式，R保持至少一个周期有效以退出游行模式。游行模式下，控制器按照平常时序运行直到L_B变成绿色，然后保持L_B为绿色有效这种状态直到游行模式结束。

首先，画出单个有限状态机的状态转换图，如图3-33(a)所示。然后，画出2个相互作用有限状态机的状态转换图，如图3-33(b)所示。进入游行模式时，模式有限状态机的输出M为有效。灯控制有限状态机根据M的值、交通传感器T_A和T_B来控制灯的颜色。

解：图3-34(a)给出了单个有限状态机的设计。状态S0到状态S3处于普通模式。状态S4到状态S7处于游行模式。这两个部分基本上相同，但是在游行模式下，有限状态机保持在状态S6，此时Bravado大道上的灯为绿色。输入P和R控制了在两个部分之间的转移。其有限状态机设计很杂乱。图3-34(b)显示了分解设计的有限状态机。模式有限状态机用2种状态跟踪处于正常模式或游行模式。当M为真，灯控制有限状态机将修改成保持在S2状态。

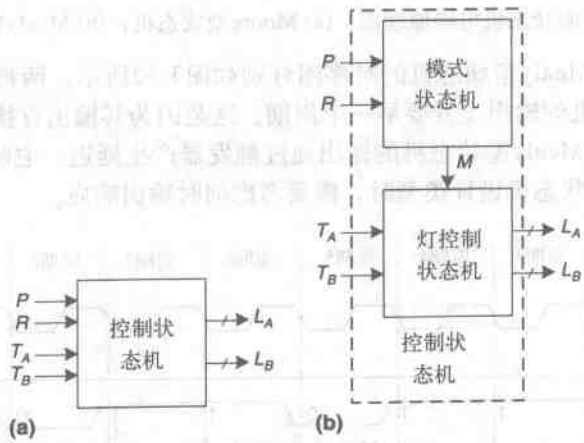


图3-33 修改后交通灯控制器有限状态机的两种设计：(a) 单个状态机；(b) 分解为两个状态机

3.4.5 有限状态机小结

有限状态机是根据给定规范设计同步时序电路的强有力系统化方法。设计有限状态机的步骤如下：

- 确定输入和输出；
- 画状态转换图；
- 对于Moore型状态机：
  - ➢ 写出状态转换表；

➢ 写出输出表；

- 对于Mealy型状态机：

➢ 写出组合的状态转换和输出表；

- 选择状态编码——这个选择将影响硬件设计；
- 为下一个状态和输出写出布尔表达式；
- 画出电路草图。

本书将多次使用有限状态机来设计复杂的数字系统。

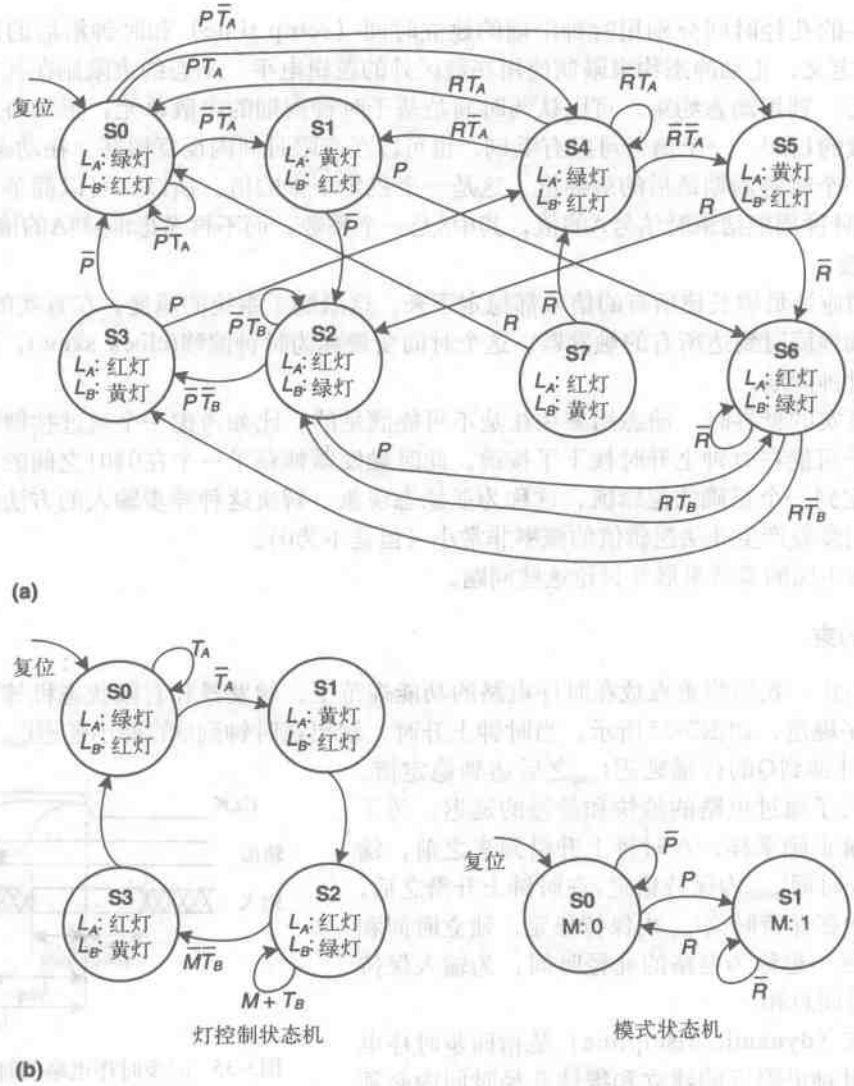


图3-34 状态转移图：(a) 未分解的；(b) 分解后的

3.5 时序逻辑电路的时序

我们知道，一个触发器在时钟上升沿到达时将D复制到输出Q。这个过程称为在时钟沿对D进行采样（sampling）。当在时钟上升沿时，如果D是0或1的稳定状态，这个动作定义很清

晰。但是，如果D在时钟上升时发生了变化，将会发生什么情况？

这个问题类似于一个正在捕捉图片的照相机。设想这样一个场景，一只青蛙正在从一个睡莲上跳入湖水里。如果在青蛙跳之前拍照，你将看到一只在睡莲上的青蛙。如果在青蛙跳过之后拍照，你将看到水面上的波纹。如果你刚好在青蛙跳的时候拍照，你将看到一只伸展的青蛙从睡莲跳入湖水的模糊影像。照相机的特征由孔径时间（aperture time）刻画，在此时间内一个物体必须保持不动，照相机才能获得清晰的图像。同样，一个时序元器件在时钟沿附近也有孔径时间。在孔径时间内输入必须稳定，触发器才能产生明确定义的输出。

时序元件的孔径时间分别用时钟沿前的建立时间（setup time）和时钟沿后的保持时间（hold time）定义。正如静态约束限制使用在禁区外的逻辑电平，动态约束限制在孔径时间内使用改变信号。利用动态约束，可以认为时间是基于时钟周期的离散单元，正如将信号的电平认为是离散的1和0。一个信号可以有毛刺，也可以在有限时间内反复振荡。在动态约束下，我们仅关心一个时钟周期最后的最终值，这是一个稳定下来的值。所以，可以简单地用A[n]表示在第n个时钟周期结束时信号A的值，其中n是一个整数，而不再考虑t时刻A的值A(t)，其中t是一个实数。

时钟周期应该足够长使所有的信号都稳定下来。这限制了系统的速度。在真实的系统中，时钟不能精确地同时到达所有的触发器。这个时间变量称为时钟偏移(clock skew)，从而需要进一步增加时钟周期。

在面对真实的世界时，动态约束往往是不可能满足的。比如考虑一个通过按键输入的电路。一只猴子可能在时钟上升时按下了按键。此时触发器捕获了一个在0和1之间的值，这个值不可能稳定到一个正确的逻辑值，这称为亚稳态现象。解决这种异步输入的方法是使用同步器，这个同步器产生非法逻辑值的概率非常小（但是不为0）。

我们将在下面的章节里展开讨论这些问题。

3.5.1 动态约束

到目前为止，我们将重点放在时序电路的功能规范上。触发器和有限状态机等同步时序电路也有时序规范，如图3-35所示。当时钟上升时，输出在时钟到Q的最小延迟t_{ccq}之后开始改变，并在时钟到Q的传播延迟t_{pcq}之后达到稳定值。它们分别代表了通过电路的最快和最慢的延迟。为了电路对输入量正确采样，在时钟上升沿到来之前，输入必须在建立时间t_{setup}内保持稳定，在时钟上升沿之后，输入必须至少在保持时间t_{hold}内保持稳定。建立时间和保持时间合在一起称为电路的孔径时间，为输入保持稳定状态的时间总和。

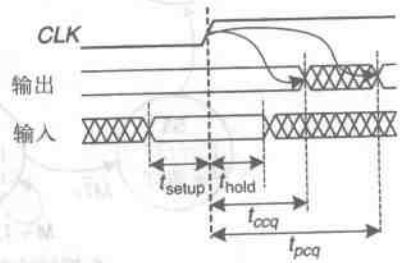


图3-35 同步时序电路的时间规范

动态约束（dynamic discipline）是指同步时序电路的输入在时钟沿附近的建立和维持孔径时间内必须保持稳定。为了满足这个要求，要保证触发器在对信号进行采样时，信号不能变化。因为在采样时仅关心最终的输入值，从而可以认为信号是在时间和逻辑电平上都是离散的量。

3.5.2 系统时序

时钟周期或者时钟时间T_c是周期性时钟信号中上升沿之间的时间间隔。它的倒数$f_c = 1/T_c$是时钟频率。其他情况一样的情况下，提高时钟的频率可以增加数字系统在单位时间内完成

的工作量。频率的单位是Hz，或者是每秒的周期数：1MHz = 10^6Hz，1GHz = 10^9Hz。

图3-36(a) 给出了同步时序电路中一条通用路径，我们希望计算其时钟周期。在时钟的上升沿，寄存器R1产生输出Q1。这些信号进入一个组合逻辑电路产生D2，并作为寄存器R2的输入。时序图如图3-36(b) 所示，每一个输出信号在输入信号发生改变的最小延迟后开始改变，在输入信号稳定之后的传输延迟时间内输出信号稳定到最终的值。灰色箭头表示通过R1和组合逻辑块的最小延迟，高亮的箭头代表通过R1和组合逻辑块的传输延迟。我们通过第二个寄存器R2的建立时间和保持时间来分析时序约束。

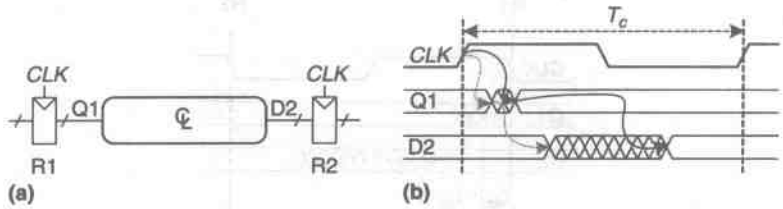


图3-36 寄存器中间的路径的时序图

建立时间约束

图3-37只显示了路径的最大延迟时序图。为了满足R2的建立时间，D2应在不迟于下一个时钟沿之前的建立时间前稳定。

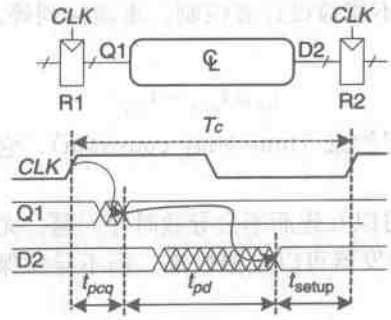


图3-37 对于建立时间约束的最大延迟

所以我们得出了最小延迟周期的等式：

$$T_c \geqslant t_{pcq} + t_{pd} + t_{setup} \tag{3-12}$$

在商业设计中，时钟周期经常由研发总监和市场部提出（以确保产品的竞争性）。而且，制造工艺确定了触发器时钟到Q的传输延迟t_{pcq}和建立时间t_{setup}。可以重写式（3-12）以确定通过组合逻辑的最大传输延迟，这时设计师经常只能控制的一个变量。

$$t_{pd} \leqslant T_c - (t_{pcq} + t_{setup}) \tag{3-13}$$

在圆括号内的项$t_{pd} + t_{setup}$称为时序开销（sequencing overhead）。理想状态下，整个周期时间T_C都应用于有用的组合逻辑计算，其传播延迟为t_{pd}。但是触发器的时序开销占用了周期时间。式（3-13）称为建立时间约束（setup time constrain）或最大延迟约束（max-delay constrain），因为它取决于建立时间，并限制了组合逻辑的最大延迟时间。

如果组合逻辑的传输延迟太大，D2有可能在R2对其采样时稳定不到它的最终值。所以，R2可能采样到一个不正确的结果或者一个处于禁止区域的非法电平。在这种情况下，电路将

出现故障。解决这个问题的方法有两个：增加时钟周期或重新设计组合逻辑以缩短传输延迟。

保持时间约束

图3-36(a) 中的寄存器R2也有保持时间约束。在时钟上升沿之后的保持时间（t_{hold}）内，它的输入D2必须保持不变。根据图3-38，D2在时钟上升沿之后$t_{ccq}+t_{cd}$内可能会变化。

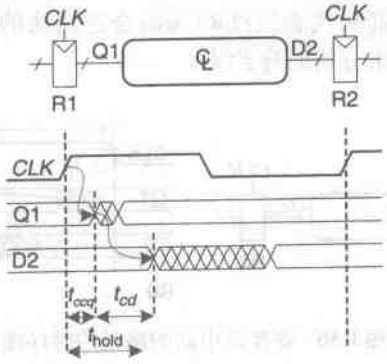


图3-38 对于保持时间约束的最小延迟

所以，我们可以得到

$$t_{ccq}+t_{cd}\geqslant t_{hold} \tag{3-14}$$

t_{ccq}和t_{cd}是触发器的属性，不能被设计者控制。重新排列等式，可以得到组合逻辑的最小延迟：

$$t_{cd}\geqslant t_{hold}-t_{ccq} \tag{3-15}$$

式（3-15）也称为最小延迟约束（min-delay constrain），它限制了组合逻辑的最小延迟。

我们假定任何逻辑元件都可以互连而不会导致时序问题。尤其是希望如图3-39所示的2个触发器可以直接级连，而不导致保持时间问题。

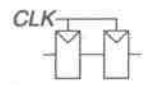


图3-39 直连的两个触发器

在此情况中，因为在触发器之间没有组合逻辑，所以$t_{cd}=0$。代入式（3-15）得出下列等式：

$$t_{hold}\leqslant t_{ccq} \tag{3-16}$$

换句话说，一个可靠触发器的保持时间要比它的最小延迟短。触发器经常被设计成$t_{hold}=0$，于是式（3-16）总是可以满足。除非特别注明，本书中总是忽略保持时间约束。

然而，保持时间约束又非常重要。如果它们不能满足，唯一的解决办法是需要重新设计电路以增加组合逻辑的最小延迟。与建立约束不同，它们不能通过调整时钟周期来改正。以目前的科技水平，重新设计和制造一个集成电路需要花费数月的时间和上千万美元。所以违反保持时间约束将产生非常严重的后果。

小结

时序电路中的建立时间和保持时间约束了触发器之间组合逻辑的最大延迟和最小延迟。现代的触发器经常将组合逻辑的最小延迟设计为0，即触发器可以背靠背地放置。因为高速电路中高时钟频率意味着短的时间周期，最大延迟约束限制了其关键路径上串连门的个数。

例3.9 时序分析

Ben设计了图3-40所示的电路。根据组件的数据手册，触发器的时钟到Q最小延迟和传输

延迟分别为30ps和80ps。它们的建立时间和保持时间分别为50ps 和60ps。每一个逻辑门的传输延迟和最小延迟分别为40ps和25ps。帮助Ben确定最大的时钟周期，是否能满足保持时间约束。这个过程被称为时序分析（timing analysis）。

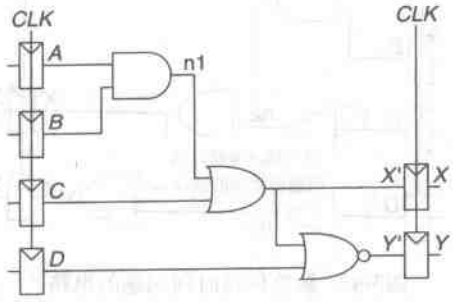


图3-40 用于时序分析的实例电路

解：如图3-41(a) 所示，当信号变化时的波形图。输入A到D被寄存，所以它们只在CLK上升后立刻改变。

关键路径发生在B = 0，C = 0，D = 0，且A从0上升为1，触发n1上升，X′上升，Y′下降，如图3-41(b) 所示。这条路径含有3个门的延迟。对于关键路径，我们假定对于每一个门都需要它全部的传输延迟。Y′必须在下一个时钟上升沿到来之前建立。所以最小的周期是

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{setup}$$
$$= 80 + 3 \times 40 + 50 = 250\text{ps} \qquad (3\text{-}17)$$

最大的时钟频率是$f_C = 1/T_C = 4\text{GHz}$

在最短路径上，当A = 0，C上升，导致X′上升，如图3-41(c) 所示。对于最短路径，我们假定每个逻辑门仅在最小延迟之后翻转。这条路径只包含一个门的延迟，所以它将在$t_{ccq} + t_{cd}$ = 30 + 25 = 55 ps之后发生。但是这个触发器需要60 ps的保持时间，意味着X′必须在时钟上升沿到来之后的60ps内保持稳定，X′触发器才能可靠地对它的值进行采样。在这种情况下，在第一个时钟上升沿的时候，X′ = 0，所以我们希望触发器捕获X = 0。因为X′不能保持稳定的状态足够长的时间，所以X的实际值不可预测。这个电路违反了保持时间约束，在任何时钟频率下其动作都可能不正确。

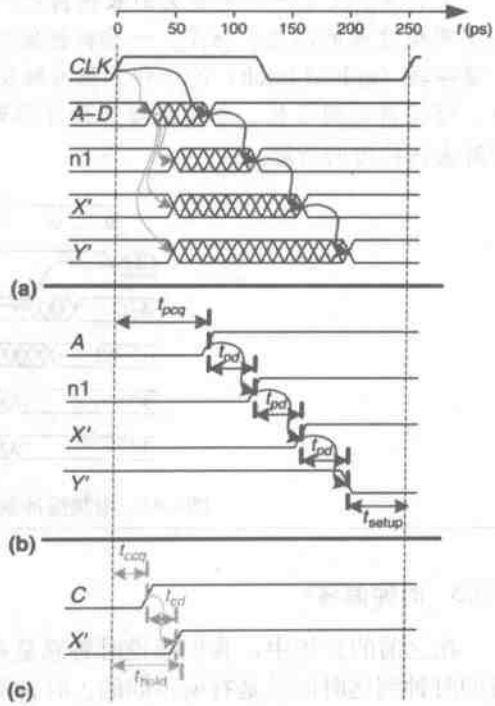


图3-41 时序图：(a) 一般情况；(b) 关键路径；(c)最短路径

例3.10 修改电路以维护保持时间约束

Alyssa打算通过增加缓冲器降低最短路径速度以修复Ben的电路，如图3-42所示。缓冲器

和其他的门有相同延迟。确定电路最大的时钟频率，是否会发生保持时间问题。

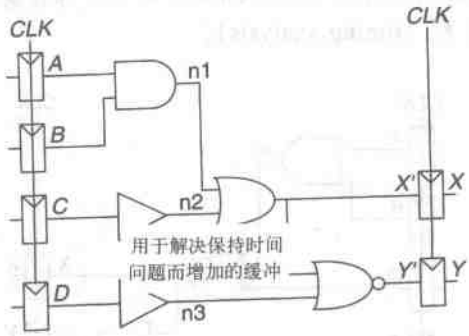


图3-42 解决保持时间问题的电路

解：图3-43显示了信号变化的波形图。从A到Y的关键路径不受影响，因为它没有通过任何缓冲器。所以最大的时钟频率依然是4GHz。但是，最短路径被缓冲器的最小延迟变慢了。X′在t_{ccq} + 2 t_{cd} = 30 + 2 × 25 = 80 ps之前都保持不变。这是在保持时间60 ps之后，所以电路运行正常。

这个例子用了一个不常见的长保持时间来说明保持时间问题。很多触发器被设计成$t_{hold} \leqslant t_{ccq}$来避免这类的问题。但是，一些高性能的微处理器（包括奔腾4）在触发器中使用了称为脉冲锁存器（pulsed latch）的组件。脉冲触发器的行为类似于触发器，但是它的时钟到Q延迟很短，而保持时间很长。总之，增加缓冲器通常（但并不总是）能在不降低关键路径速度的同时解决保持时间问题。

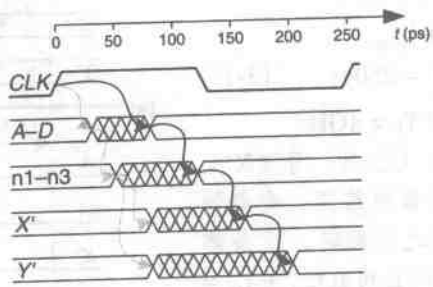


图3-43 增加缓冲解决保持时间问题后的时序图

3.5.3 时钟偏移*

在之前的分析中，我们假设时钟总是在同一时刻到达各个寄存器。在现实中，每个寄存器的时钟到达时间总是有所不同的。时钟沿到达时间的变化称为时钟偏移（clock skew）。例如，从时钟源到不同的寄存器之间的连线长度不同，导致了延迟的微小差异，如图3-44所示。噪声同样也可以导致不同的延迟。3.2.5节中介绍的时钟门控也可以进一步延迟时钟。如果一些时钟经过门控，而另外一些没有，则门控时钟和非门控时钟之间就一定会存在偏移。图3-44中的CLK2相对于CLK1要早，因为在两个寄存器之间的时钟线上有一条通路。如果时钟的布线不同，CLK1也可能会早一些。在进行时序分析时，需要考虑最坏的情况，以保证电路在所有的环境下都可以工作。

图3-45是在图3-36上增加了时钟偏移后的时序分析。粗时钟线表示时钟信号到达每个寄存器的最迟时间，虚线表示时钟可能提前t_{skew}时间到达。

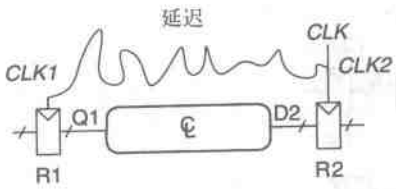


图3-44 由线延迟引起的时钟偏移

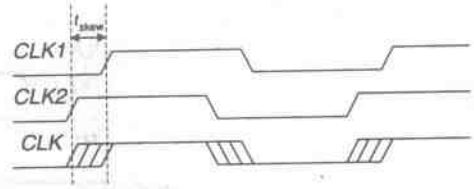


图3-45 带时钟偏移的时序图

首先考虑在图3-46中的建立时间约束。在最坏的情况下，R1收到最迟偏移时钟，R2收到最早偏移时钟，而留下的时间在两个寄存器之间进行数据传输。

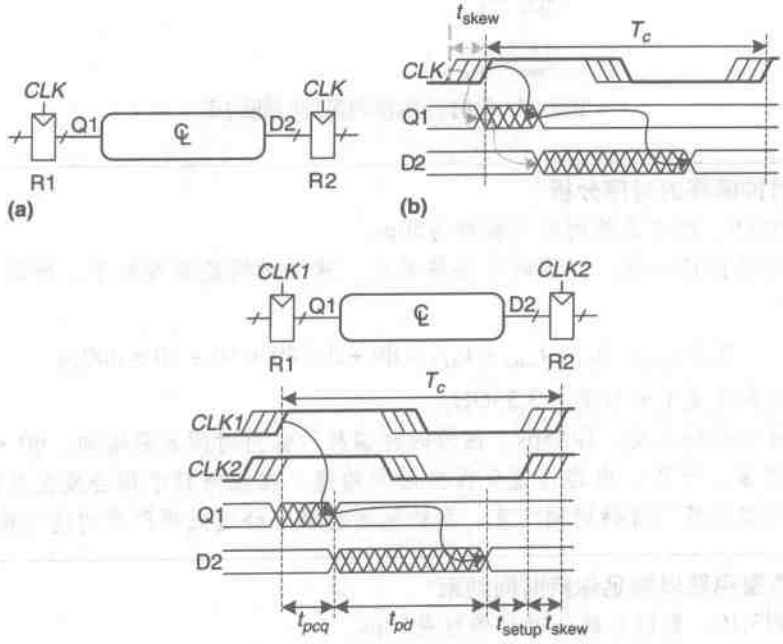


图3-46 带时钟偏移的建立时间约束

数据通过寄存器和组合逻辑传输，并必须在R2采样之前建立。所以可以得到：

$$T_c \geqslant t_{pcq} + t_{pd} + t_{setup} + t_{skew} \tag{3-18}$$

$$t_{pd} \leqslant T_c - (t_{pcq} + t_{setup} + t_{skew}) \tag{3-19}$$

接着考虑图3-47的保持时间约束。在最坏的情况下，R1接收最早偏移时钟CLK1，R2接收最迟偏移时钟CLK2。数据通过寄存器和组合逻辑传输，且不能在迟到时钟之后保持时间内到达，所以可以得到：

$$t_{ccq} + t_{cd} \geqslant t_{hold} + t_{skew} \tag{3-20}$$

$$t_{cd} \geqslant t_{hold} + t_{skew} - t_{ccq} \tag{3-21}$$

总之，时钟偏移将显著增加建立时间和保持时间。这将使时序总开销增加，从而减少了组合逻辑的有效工作时间。它也增加了通过组合逻辑所需的最小延迟。如果$t_{hold} = 0$，且t_{skew}

$>t_{ccq}$，一对背靠背的触发器将不满足式（3-21）。为了防止严重的保持时间错误，设计者绝对不能允许太多的时钟偏移。当时钟偏移存在时，触发器有时被故意地设计成特别慢（增大t_{ccq}），以防止保持时间问题。

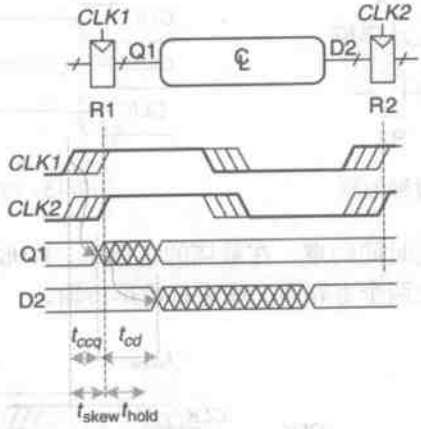


图3-47 带时钟偏移的保持时间约束

例3.11 时钟偏移的时序分析

重新考虑例3.9，假定系统的时钟偏移为50ps。

解：关键路径保持一样，但因时钟偏移关系，建立时间显著增加了。所以，最小的周期时间是

$$T_c \geqslant t_{pcq} + 3t_{pd} + t_{setup} + t_{skew} = 80 + 3 \times 40 + 50 + 50 = 300ps \tag{3-22}$$

最大的时钟频率是 $f_c = 1/T_c = 3.33GHz$

最短的路径也保持不变，即55ps。因为时钟偏移，保持时间显著增加，60 + 50 = 110ps，它比55ps要大很多。所以，电路将违反保持时间约束，在任何频率都会发生故障。即使没有时钟偏移时，电路也违反保持时间约束。系统的时钟偏移会造成更严重的违反情况。

例3.12 调整电路以满足保持时间约束

重新考虑例3.10，假设系统的时钟偏移是50ps。

解：关键路径不受影响，所以最大的时钟频率仍然为3.33GHz。

最短路径增加到80ps。这仍然比$t_{hold} + t_{skew}$的110ps小，所以电路仍然违反保持时间约束。

为了修复这个问题，需要插入更多的缓冲器。在关键路径上同样也需要增加缓冲器，从而降低时钟频率。作为选择，也可以选用其他保持时间更短的触发器。

3.5.4 亚稳态

如前所示，当输入来自外界时，到达时间可能在孔径期间内，而不能保证时序电路的输入总是稳定的。考虑一个按键连接到触发器的输入端，如图3-48所示。按键没有被按下时，D = 0；按下时，D = 1。一只猴子相对于时钟的上升沿随机地按按键。我们希望知道时钟上升沿后输出Q的值：

情况Ⅰ，当按键在CLK之前被按下，Q = 1。

情况Ⅱ，按键在CLK之后很久都没有被按下，Q = 0。

情况Ⅲ，当按键在CLK之前的t_{setup}和CLK之后的t_{hold}之间的某个时间被按下，输入破坏了动态规则，输出将无法确定。

亚稳定状态

实际上，当触发器对在孔径时间内发生变化的输入进行采样时，输出Q可能随时地取在禁区内的0和V_{DD}之间的一个电压。这被称为亚稳定状态。触发器最终将确定输出到0或者1的稳定状态。但是，到达稳定状态的分辨时间（resolution time）是无界的。

触发器的亚稳定状态类似于一个放在两个山谷间山峰顶点的球。处于山谷中的球为稳定状态，因为它们在不受干扰情况下可以一直保持它的状态。在山峰顶上的球为亚稳定状态，因为如果保持绝对的平衡，这个球将保持在山顶。但是没有绝对的平衡，球将最终滚落到一边或者另一边。发生这种改变所需时间取决于球的最初位置上与平衡点之间的距离。每一个双稳态的设备在两个稳定状态之间都存在一个亚稳定状态。

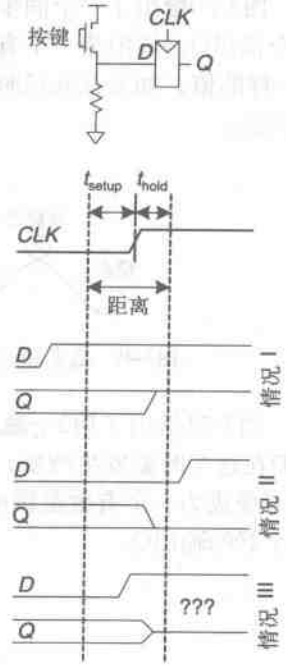


图3-48 输入在孔径时间之前，之后和之内改变

分辨时间

如果触发器在时钟周期内的任意时间发生改变，分辨时间t_{res}为达到一个稳定状态所需要的时间，这也是一个随机的变量。如果输入在孔径时间外改变，则$t_{res} = t_{pcq}$。但是如果输入在孔径时间内发生改变，t_{res}一定比较长。理论和实践上的分析（见3.5.6节）指出：分辨时间t_{res}超过任意时间t的概率按t的指数方式减少：

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3-23}$$

T_c为时钟周期，T_0和τ由触发器的属性决定。此式仅在t比t_{pcq}长的条件下有效。

直观地，$\frac{T_0}{T_c}$表示了输入在最坏时间（比如，孔径时间）内发生改变的概率。这个概率随周期T_c而减少。τ是一个时间常量，说明了触发器从亚稳态移开的速度，这与触发器中耦合门的延迟有关。

总之，如果触发器等双稳态设备的输入在孔径时间内发生改变，输出在稳定到0或者1之前是一个亚稳态的值。到达稳定状态的时间是无界的，因为对于任何的有限时间t，触发器仍处于亚稳态的概率都不会是0。但这个概率将随着t的指数方式而减少。所以，如果等待超过t_{pcq}的足够长时间，触发器到达一个有效逻辑电平的概率将很高。

3.5.5 同步器

对于数字系统而言，来自真实世界的异步输入是不可能避免的（比如，人的输入就是异步的）。如果处理不当，系统中这些异步输入将导致亚稳态电压，从而产生很难发现和改正的不稳定系统错误。数字系统设计师的目标是：对于给定的异步输入，确保遇到的亚稳态电压的概率足够小。“足够”取决于应用环境。对于数字移动电话，在10年间有一次失效是可以接受的。因为即使它锁定了，用户也可以关机，然后再打过去。对于医疗设备，在预期的宇宙

生命（10^{10}年）中才产生一次失效将是一个更好的指标。为了确保产生正确的逻辑电平，所有的异步输入必须经过同步器（synchronizer）。

图3-50给出了一个同步器。它接收异步输入信号D和时钟CLK。在有限的时间内，它产生一个输出Q；输出为一个有效逻辑电压的概率很高。如果在孔径时间内D是稳定的，Q将取和D一样的值。如果在孔径时间内D发生变化，Q可能取HIGH或者LOW。但是一定不会是亚稳定状态。

图3-49 稳定和亚稳定状态

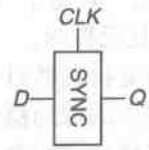


图3-50 同步器电路符号

图3-51给出了用2个触发器建立同步器的简单方法。F1在CLK的上升沿对D进行采样。如果D在这个时刻发生改变，则输出D2将出现暂时的亚稳态。如果时钟周期足够长，D2在周期结束变成为一个有效逻辑电平的概率很高。F2接着对D2进行采样，它现在是稳定的，将产生一个好的输出Q。

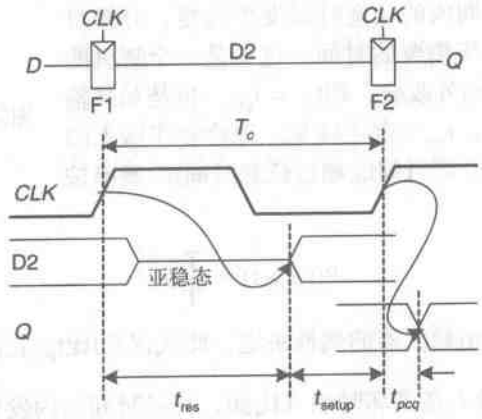


图3-51 一个简单的同步器

如果同步器的输出Q为亚稳态，那么这个同步器将失效。产生这种情况的原因是D2在F2上所必须的建立时间之前没有变成有效的电平：如果$t_{res} > T_c - t_{setup}$，由式（3-23），在随机时间内输出信号改变而导致同步器错误的失效概率是：

$$P(\text{失效}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}} \tag{3-24}$$

失效概率P(失效) 是信号D改变时，输出Q为亚稳态的概率。如果D每秒钟改变一次，每秒钟失效的概率就是P(失效)。如果D每秒钟改变N次，则失效的概率就需要乘以N倍：

$$P(\text{失效})/s = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}} \tag{3-25}$$

系统的可靠性通常由失效平均间隔时间（Mean time between failures, MTBF）衡量。根据

定义可以看出，MTBF是系统失效之间的平均时间，是系统失效概率的倒数：

$$\text{MTBF} = 1/P(\text{失效}) = T_c e^{\frac{T_c - t_{setup}}{\tau}} \Big/ N T_0 \tag{3-26}$$

式（3-26）指出MTBF按同步器延迟T_c的指数方式增加。对于多数系统，同步器等待一个时钟周期将提供给一个安全的MTBF。但在高速的系统中则不行，必须等待更多的周期。

例3.13 有限状态机输入的同步器

3.4.1节中的有限状态机交通灯控制器接收交通传感器的异步输入。假定同步器用于保证控制器得到稳定的输入信号。交通到达的次数是平均每秒钟0.2次。同步器中触发器的特性为$\tau = 200\text{ps}$，$T_0 = 150\text{ ps}$，$t_{setup} = 500\text{ps}$。这个同步器的时钟周期是多少才能使MTBF超过1年？

解：1年 $= \pi \times 10^7\text{s}$。解等式3.26

$$\pi \times 10^7 = T_c e^{\frac{T_c - 500\times10^{-12}}{200\times10^{-12}}} \Big/ (0.2)(150 \times 10^{-12}) \tag{3-27}$$

这个等式没有封闭解，但是，通过猜想和检验很容易解答。在数据表中，尝试将一些T_c的值带入计算MTBF，直到T_c的值可以保证MTBF为1年：$T_c = 3.036\text{ps}$。

3.5.6 分辨时间的推导*

式（3-23）可以使用电路理论，差分方程和概率论等基础知识得到。如果读者对推导不感兴趣或对相关数学知识不了解，可以跳过此节。

一个触发器在给定时间t后处于亚稳态，即如果触发器对正在变化的输入（将产生亚稳态条件）进行采样，而且输出在时钟沿后的这段时间内没有达到稳定的电平。这个过程可以用数学方式描述为：

$$P(t_{res} > t) = P(\text{采样了正在改变的输入}) \times P(\text{没有达到一个稳定的电平}) \tag{3-28}$$

我们认为每个概率项都是独立的。异步输入信号在0和1之间切换，t_{switch}如图3-52所示。在时钟沿附近的孔径时间内输入发生改变的概率是

$$P(\text{采样了正在改变的输入}) = (t_{switch} + t_{setup} + t_{hold})/\ T_c \tag{3-29}$$

如果触发器以P(采样了正在改变的输入) 的概率进入了亚稳态，从亚稳态到成为有效电平的时间取决于电路的内部工作原理。这个分辨时间确定了P(没有达到一个稳定的电平)，即触发器在时间t之后没有成为有效电平的概率。本节后续部分将分析一个简单的双稳态装置模型来估计这个概率。

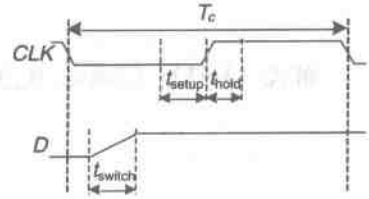


图3-52 输入计时

一个双稳态装置以正反馈方式存储。图3-53(a) 给出了用一对反相器实现的正反馈；这个电路的行为代表了典型的双稳态元件。一对反相器的行为就像缓冲器。我们以对称直流转换特征建模，如图3-53(b) 所示，其斜率为G。缓冲器只能提供有限的输出电流；可以将其建模为一个输出电阻R。所有真实的电路都必须有被充电的电容C。通过电阻对电容充电形成RC延迟，阻止了缓冲器在瞬间进行切换。所以，完整的电路模型如图3-53(c) 所示，其中$v_{out}(t)$为双稳态装置传输状态的电压。

电路的亚稳态点是$v_{out}(t) = v_{in}(t) = V_{DD}/2$；如果电路刚好在这一点开始工作，它将在没有噪声的情况下永远保持在这里。因为电压是连续变量，电路刚好在亚稳态这一点开始工作的

机会非常小。但是，电路在0时刻可能处于亚稳态附近的开始位置，$v_{out}(0) = V_{DD}/2 + \Delta V$，其中$\Delta V$是小偏移量。在这种情况下，如果$\Delta V>0$，正反馈最终将驱动$v_{out}(t)$到$V_{DD}$，如果$\Delta V<0$，将驱动$v_{out}(t)$到0。到达$V_{DD}$或0所需要的时间正是双稳态装置的分辨时间。

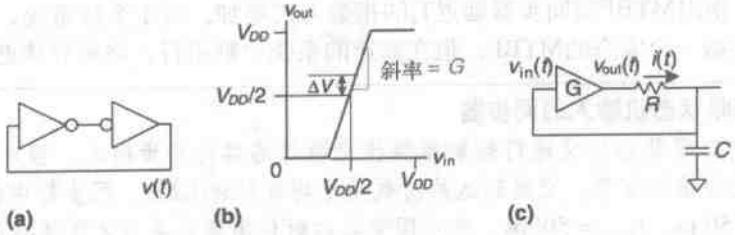


图3-53 双稳态装置的电路模型

直流转换特征是非线性的，但它在我们感兴趣的亚稳态点附近表现将线性。特别是，如果$v_{in}(t) = V_{DD}/2 + \Delta V/G$，当$\Delta V$很小时，则$v_{out}(t) = V_{DD}/2 + \Delta V$。通过电阻的电流是$i(t) = ((v_{out}(t) - v_{in}(t))/R$。电容充电的速率是$dv_{in}(t)/dt = i(t)/C$。将这些综合到一起，我们可以得出输出电压的产生方程。

$$dv_{out}(t)/dt = \frac{(G-1)}{RC}[v_{out}(t) - V_{DD}/2] \tag{3-30}$$

这是一个一阶线性微分方程。根据它们的初值$v_{out}(t) = V_{DD}/2 + \Delta V$求解。

$$v_{out}(t) = V_{DD}/2 + \Delta V\, e^{\frac{(G-1)t}{RC}} \tag{3-31}$$

图3-54描绘了在不同给定起始点上电压$v_{out}(t)$的轨迹。$v_{out}(t)$按照指数方式远离亚稳态点$V_{DD}/2$，直到它饱和为V_{DD}或者0。输出电压最终将成为1或者0。所花费的时间取决于从亚稳态点$V_{DD}/2$到最初电压的偏移量（ΔV）。

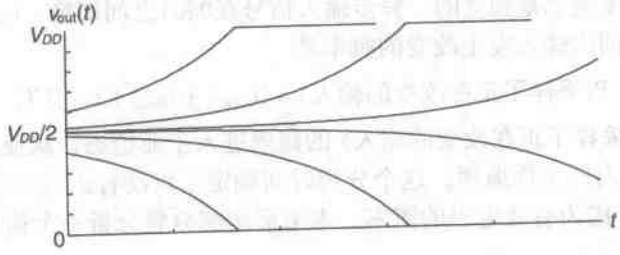


图3-54 解轨迹

解式（3-31），以求$v_{out}(t_{res}) = V_{DD}$或者0时的分辨时间$t_{res}$：

$$|\Delta V|\, e^{\frac{(G-1)t_{res}}{RC}} = V_{DD}/2 \tag{3-32}$$

$$t_{res} = \frac{RC}{G-1}\ln\frac{V_{DD}}{2|\Delta V|} \tag{3-33}$$

总之，如果双稳态装置有很大的电阻和电容将导致它的输出改变很慢，分辨时间会增加。如果双稳态装置有很大的增益G，分辨时间会缩短。当电路在紧挨着亚稳态的点开始（$\Delta V \to 0$），分辨时间也会成对数方式地增加。

定义τ为 $\frac{RC}{G-1}$。解式（3-33），求给定分辨时间t_{res}下的ΔV，以确定最初偏移量ΔV_{res}：

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{\frac{-t_{res}}{\tau}} \tag{3-34}$$

假设双稳态装置在输入变化时进行采样。其输入电压$v_{in}(0)$为在电压V_{DD}和0之间。在时间t_{res}之后输出没有成为合法值的概率取决于最初偏移量足够小的概率。尤其在v_{out}上的最初偏移量小于ΔV_{res}，所以在v_{in}上的最初偏移量必须小于$\Delta V_{res}/G$。双稳态装置对输入进行采样时得到足够小的最初偏移量的概率为

$$P(\text{没有达到一个稳定的电平}) = P\left(\left|v_{in}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{GV_{DD}} \tag{3-35}$$

综上所述，分辨时间超过时间t的概率，下式给出

$$P(t_{res} > t) = \frac{t_{switch} + t_{setup} + t_{hold}}{GT_c} e^{\frac{-t}{\tau}} \tag{3-36}$$

观察式（3-36），并按式（3-23）形式重写，$T_0 = (t_{switch} + t_{setup} + t_{hold})/G$，$\tau = RC/(G-1)$。总之，我们已经得出式（3-23），并证明$T_0$和τ取决于双稳态装置的物理属性。

3.6 并行

系统的速度可以用延迟和通过系统的任务（token）吞吐量来衡量。任务定义为经过处理后能产生一组输出的一组输入。可以采用可视化的方法理解，即在电路图中输入这些任务，并通过电路得到结果。延迟（Delay）是从开始到结束需要的时间。吞吐量（throughout）是系统单位时间内产生任务的数量。

例3.14 饼干的延迟和吞吐量

Ben决定举行一个牛奶和饼干晚宴来庆祝交通灯控制器成功安装。做好饼干并放入盘中需要花费5分钟。将饼干放入烤箱烤好需要花费15分钟。烤好一次饼干后，他开始做下一盘饼干。Ben做好一盘饼干的吞吐量和延迟是多少？

解：在这个例子中，一盘饼干是一个任务。每盘的延迟是1/3小时，吞吐量是3盘/小时。

读者可能会想到在同一时间内处理多个任务可以提高吞吐量。并行性（parallelism）有两种形式：空间和时间。空间并行（spatial parallelism）是提供多个相同的硬件，这样多个任务就可以在同一时间一起处理。时间并行（temporal parallelism）是一个任务被分成多个阶段，类似于流水装配线。多个任务可以分布到所有的阶段。虽然每一个任务必须通过全部的阶段，但在给定的时间内每段都有一个不同的任务，从而使得多个任务可以重叠起来。时间并行通常称为流水线（pipelining）。空间并行有时也称为并行，但是我们避免这种容易产生误解的命名约定。

例3.15 饼干并行

Ben有上百个朋友来参加他的晚宴，所以他要加快做饼干的速度。他考虑使用空间并行和时间并行。

空间并行：Ben请Alyssa提供帮助。Alyssa有她自己的饼干盘和烤箱。

时间并行：Ben拿来第二个饼干盘。他一旦把饼干盘放入烤箱，就开始做饼干并放入另外

的一个盘子里，而不是等着第一盘烤好。

使用空间并行和时间并行后的吞吐量和延迟是多少？两种方法同时使用后的吞吐量和延迟是多少？

解：延迟是完成一个任务从开始到结束所需要的时间。在所有情况下，延迟都是1/3小时。如果Ben开始时没有饼干，延迟是他完成第一盘饼干的所需要的时间。

吞吐量是每小时烤好的饼干盘数量。使用空间并行的方法，Ben和Alyssa每20分钟完成一盘饼干。所以吞吐量是以前的2倍，即6盘/小时。使用时间并行的方法，Ben每15分钟就把一个新盘放入烤箱，吞吐量是4盘/小时。如图3-55所示。

如果Ben和Alyssa同时使用这两种并行技术，吞吐量是8盘/小时。

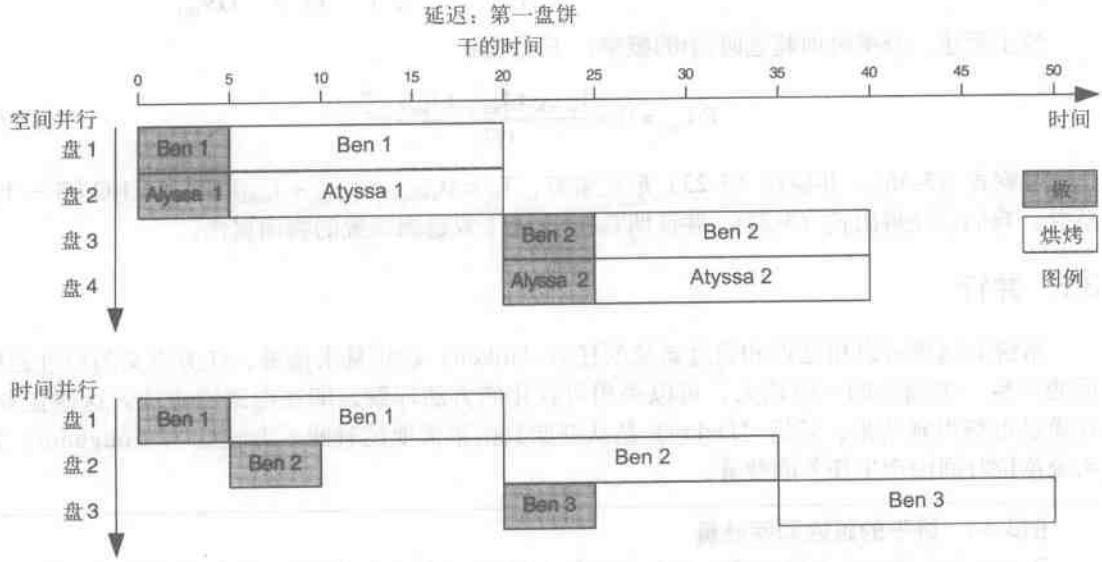


图3-55 烤饼干的时间并行和空间并行

考虑一个延迟为L的任务。没有并行的系统中，吞吐量为1/L。在空间并行系统中有N个相同的硬件，则吞吐量为N/L。在时间并行系统中，任务可以理想地分成等长的N个步骤或阶段。在这种情况下，吞吐量也是N/L，且只需要一套硬件。但是饼干的例子显示：将任务分解为N个等长的阶段是不切实际的。如果最长的延迟为L_1，则流水线的吞吐量为$1/L_1$。

流水线（时间并行）特别有吸引力，因为它没有增加硬件就可以加速电路运行。实现方法是将寄存器放置在组合逻辑块之间将逻辑分成较短的阶段，使之可以以较快的时钟频率运行。寄存器用于防止流水线中某一级的任务赶上和破坏下一个级的任务。

图3-56给出了一个没有流水线的电路例子。它在寄存器间包含4个逻辑块。关键路径通过第2，3，4块。假设寄存器时钟到Q的传输延迟为0.3ns，建立时间为0.2ns。则周期是T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5ns。电路的延迟为9.5ns，吞吐量为1/9.5ns = 105MHz。

图3-57给出了一个相同功能的电路，但在第3和第4块之间加入一个寄存器，从而将电路分割成两阶段流水线。第一个阶段的最小延迟周期是0.3 + 3 + 2 + 0.2 = 5.5ns。第二阶段的最小周期是0.3 + 4 + 0.2 = 4.5ns。时钟必须足够慢，使得所有的阶段都能正确工作。所以，T_c = 5.5ns。延迟为2个时钟周期，即11ns。吞吐量是1/5.5ns = 182 MHz。这个例子显示在真实的电路中，两级流水线通常可以得到几乎是双倍的吞吐量和稍微增加的延迟。相比之下，理想状

态流水线的吞吐量可以提高一倍，而延迟不变。产生差别的原因在于电路不可能分割成完全相等的两半，而且寄存器引入了额外的时序开销。

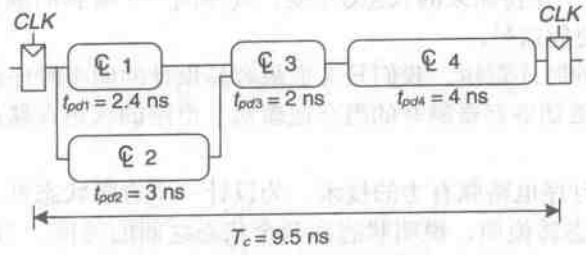


图3-56 无流水线电路

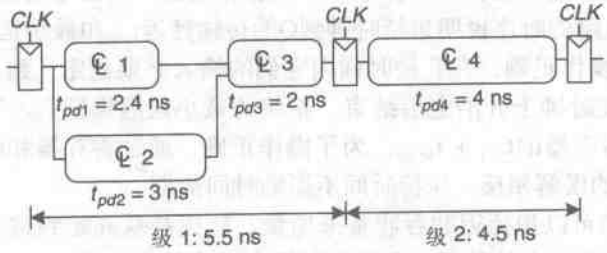


图3-57 两级流水线电路

图3-58给出来分割成3级流水线的相同电路。注意：需要2个以上的寄存器来存储在第一流水线结束后块1和块2的结果。周期被第三阶段限制为4.5ns。延迟为3个周期，即13.5ns。吞吐量为1/4.5ns = 222MHz。再增加一个流水阶段提高了吞吐量，也增加了一些延迟。

虽然这些技术都很强大，但它们并不能运用到所有的情况。并行的障碍是依赖关系(dependency)。如果当前的任务依赖于前一个任务的结果，而不是在当前任务中的前一步结果，此时只有前一个任务完成后，后一个任务才能开始。例如，Ben想在开始准备第二盘之前检查第一盘饼干的味道是否好。这就是一个阻止流水线或并行操作的依赖关系。并行性是设计高性能微处理器的重要技术。第7章中将进一步讨论流水线，并用例子说明如何处理依赖关系。

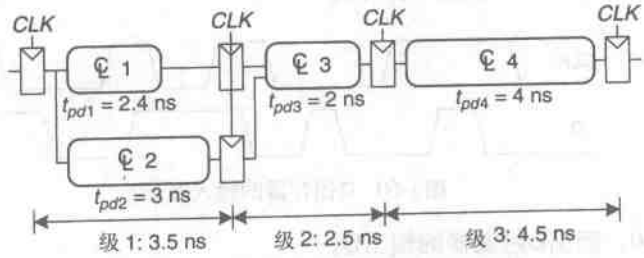


图3-58 三级流水线电路

3.7 总结

本章介绍了时序逻辑电路的分析和设计。和输出只取决于当前输入的组合逻辑电路相比，时序逻辑电路的输出取决于当前和先前的输入。换句话说，时序逻辑电路记忆了先前的输入信息。这种记忆称为逻辑的状态。

时序逻辑电路很难分析，并容易产生设计错误，所以我们只关心小部分成熟的模块。需要掌握的最重要元件是触发器，它的输入为时钟和D，产生一个输出Q。触发器在时钟上升沿将D值复制到Q，其他时候保持原来的状态Q不变。共享同一个时钟的触发器称为寄存器。触发器还可以接收复位和使能信号。

虽然有很多种形式的时序逻辑，我们只考虑最容易设计的同步时序逻辑电路。同步时序逻辑电路包含了由时钟驱动寄存器隔开的组合逻辑块。电路的状态存储在寄存器中，仅在时钟沿到达时进行更新。

有限状态机是设计时序电路强有力的技术。为设计一个有限状态机，首先要识别状态机的输入和输出，画出状态转换图，说明状态和两个状态之间的转换。为状态选择一个编码，然后将状态转移图重写为状态转换表和输出表，指出给定当前状态和输入的下一个状态和输出。通过这些表，设计组合逻辑来计算下一个状态和输出，画出电路图。

同步时序逻辑电路的时序说明包括时钟到Q的传输延迟t_{pcq}和最小延迟t_{ccq}，建立时间t_{setup}和保持时间t_{hold}。为了操作正确，在孔径时间内它们的输入必须稳定。建立时间在时钟上升沿之前开始，保持时间在时钟上升沿之后结束。系统的最小延迟周期T_c，等于通过组合逻辑块的传输延迟t_{pd}，加上寄存器的$t_{pcq} + t_{setup}$。为了操作正确，通过寄存器和组合逻辑的最小延迟必须大于t_{hold}。与常见的误解相反，保持时间不影响时间周期。

整个系统的性能可以用延迟和吞吐量来衡量。延迟是从开始到结束需要的时间。吞吐量时系统单位时间内处理任务的数量。并行可以提高系统的吞吐量。

习题

3.1 波形如图3-59，画出SR锁存器的输出Q。

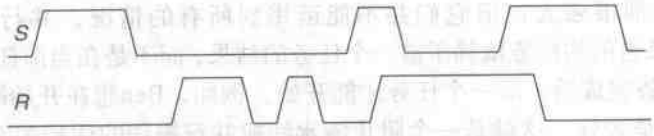


图3-59 SR锁存器的输入波形

3.2 波形如图3-60，画出D锁存器的输出Q。

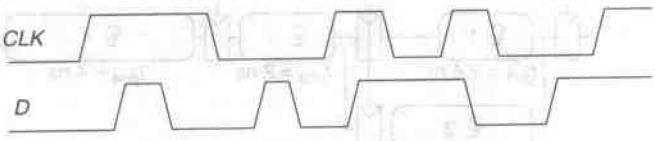


图3-60 D锁存器的输入波形

3.3 波形如图3-60，画出D触发器的输出Q。

3.4 图3-61中的电路是组合逻辑电路还是时序逻辑电路？说明输入和输出之间的关系。如何称呼这个电路？

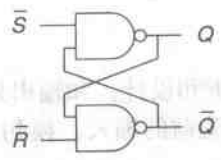


图3-61 待求解电路

3.5 图3-62中的电路是组合逻辑电路还是时序逻辑电路？说明输入和输出之间的关系。如何称呼这个电路？

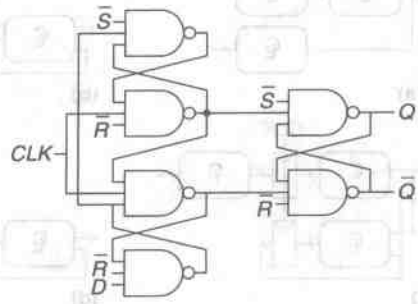


图3-62 待求解电路

3.6 T触发器（toggle flip-flop）有一个输入CLK，一个输出Q。在每一个CLK上升沿，Q的值就变成它的前一个值取反。使用D触发器和反相器画出T触发器的原理图。

3.7 JK触发器(JK flip-flop)接收一个时钟，两个输入J和K。时钟上升沿时，输出值Q被更新。如果J，K同时为0，Q将保持原来的值不变。如果只有J = 1，则Q = 1。如果只有K = 1，则Q = 0。如果J = K = 1，Q的值就切换成它的前一个值的取反。

(a) 使用D触发器和一些组合逻辑构造JK触发器。

(b) 使用JK触发器和一些组合逻辑构造D触发器。

(c) 使用JK触发器和一些组合逻辑构造T触发器。

3.8 图3-63中的电路称为Muller C元件。请简要说明输入和输出之间的关系。

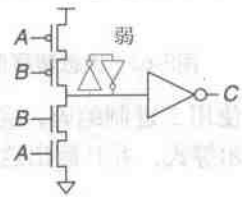


图3-63 Muller C元件

3.9 使用逻辑门设计一个带异步复位功能的D锁存器。

3.10 使用逻辑门设计一个带异步复位功能的D触发器。

3.11 使用逻辑门设计一个带同步置位功能的D触发器。

3.12 使用逻辑门设计一个带异步置位功能的D触发器。

3.13 假设由N个反相器以环连接而构成的一个环形振荡器。每一个反相器的最小延迟为t_{cd}，最大延迟为t_{pd}。如果N是奇数，确定这个振荡器的频率范围。

3.14 习题3.13中为什么N必须为奇数。

3.15 图3-64中的哪些电路是同步时序电路，为什么？

3.16 为一栋25层的建筑物设计一个电梯控制器。这个控制器有两个输入：UP和DOWN，以及指明当前电梯所在楼层的输出。没有13这个楼层。控制器的状态最少需要几位？

3.17 设计一个有限状态机来跟踪电子设计实验室里四个学生的心情。学生的心情有HAPPY（开心，电路正常工作）、SAD（忧愁，电路烧坏）、BUSY（忙碌，正在设计电路）、CLUELESS（愚笨，被电路所困扰）、ASLEEP（睡觉，趴在试验桌上睡着）。这个有限

状态机需要多少个状态？至少需要多少个位来代表这些状态？

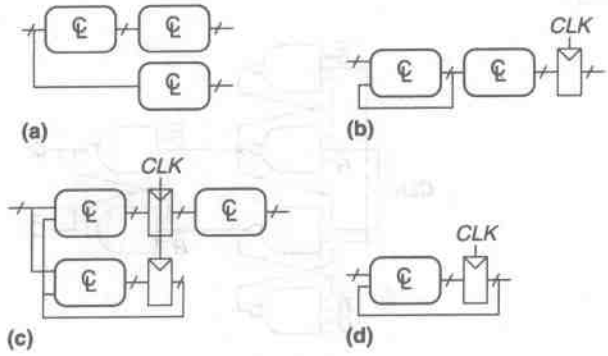


图3-64 一些电路

3.18 习题3.17中的有限状态机可以分解成多少个简单的状态机？每一个简单状态机需要多少个状态？分解后的设计总共需要多少最少位数？

3.19 说明图3-65状态机的功能。使用二进制编码，完成这个有限状态机的状态转换表和输出表。写出下一个状态和输出的布尔等式，并且画出这个有限状态机的原理图。

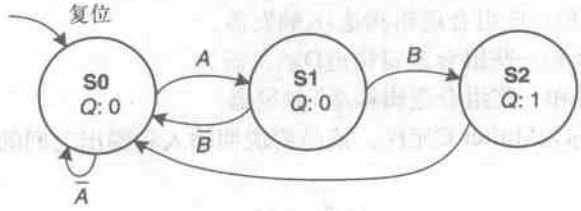


图3-65 状态转移图

3.20 说明图3-65状态机的功能。使用二进制编码，完成这个状态机的状态转换表和输出表。写出下一个状态和输出的布尔等式，并且画出这个状态机的原理图。

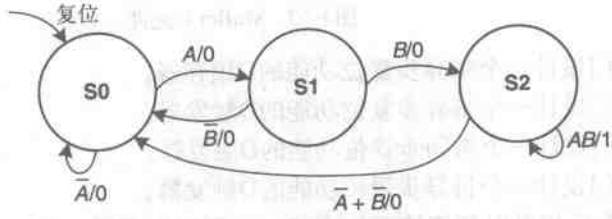


图3-66 状态转移图

3.21 在Academic大道和Bravado大街的十字路口上，交通事故还是时有发生。当灯B变绿色时，足球队冲进十字路口。在灯A变成红色之前，他们和蹒跚在十字路口缺乏睡眠的计算机专业硕士撞到了一起。扩展3.4.1节的交通灯控制器，在灯变成绿色之前，让2个灯保持红色5s。画出改进的Moore型状态机状态转换图，状态编码表，写出下一个状态和输出的布尔等式，并且画出这个有限状态机的原理图。

3.22 3.4.3节中Alyssa的蜗牛有一个女儿，它有一个Mealy型状态机的有限状态机大脑。当蜗牛女儿爬过1101或者1110时候就会微笑。为这只快乐的蜗牛用尽可能少的状态画出状

态转换图。选择状态的编码，使用你的编码画出组合的状态转换和输出表。写出下一个状态和输出等式，画出有限状态机原理图。

3.23 你将参与为部门休息室设计一个苏打汽水自动售货机。IEEE的学生会将给予不菲的奖学金，所以汽水的价格仅为25美分。机器接收5美分、一角硬币和二角五分硬币。当投入足够的硬币，苏打汽水自动售货机就会分配汽水和找零钱。为这个苏打汽水自动售货机设计一个有限状态机控制器。有限状态机的输入是Nickel（5美分），Dime（一角）和Quarter（两角五分），表示硬币已经投入机器。假设在一个周期投一个硬币。输出是Dispense，ReturnDime，ReturnTwoDime。当有限状态机到达25美分时候，它将给出Dispense和相应的Return输出，需要传递合适的找零。接着它将准备开始接收硬币以售卖下一瓶苏打汽水。

3.24 格雷码有一个很有用的特点，在连续的数字中只有一个信号位位置不同。表3-17列出了3位的格雷码表示0～7的数字。设计一个没有输入，有3位输出的8取模格雷码计数器有限状态机。（*N*取模计数器指从0到*N*-1计数，并不断重复。例如，手表的分钟和秒钟是以60为模的计数器，从0到59计数）。当重启的时候，输出为000。在每一个时钟沿，输出进入下一个格雷码。当到达100，它将从000开始重复。

表3-17 3位格雷码

数值	格雷码			数值	格雷码		
0	0	0	0	4	1	1	0
1	0	0	1	5	1	1	1
2	0	1	1	6	1	0	1
3	0	1	0	7	1	0	0

3.25 扩展习题3.24中的8取模格雷码计数器，增加UP输入变成UP/DOWN计数器。当UP = 1时，计数器进入下一个格雷码。当UP = 0时，计数器退回到上一个格雷码。

3.26 设计一个有限状态机，它有2个输入A和B，产生1个输出Z。在周期n内的输出是Z_n是输入A_n和前一个输入A_{n-1}的“与”或者“或”，其运算取决于另外一个输入B_n：

$Z_n = A_n A_{n-1}$ 如果$B_n = 0$

$Z_n = A_n + A_{n-1}$ 如果$B_n = 1$

(a) 根据图3-67给出的输入画出Z的波形图

(b) 它是Moore型状态机还是Mealy型状态机？

(c) 设计这个有限状态机。画出状态转换图和编码状态转换表，下一个状态和输出等式，原理图。

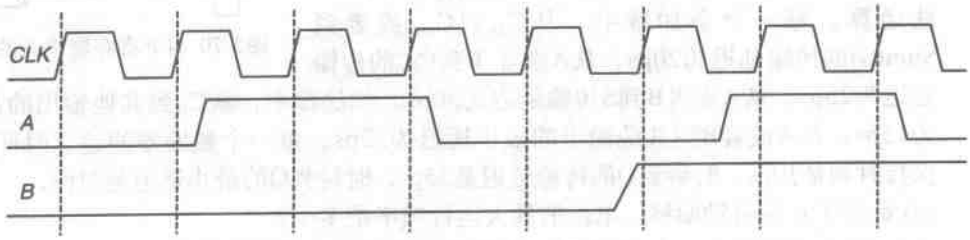


图3-67 有限状态机输入波形

3.27 设计一个有限状态机，它有1个输入A，2个输出X和Y。如果A有3个周期为1，则X为1

(没有必要连续)。如果A在至少2个连续的周期内为1，则Y为1。画出你的状态转换图和编码状态转换表，下一个状态和输出等式，原理图。

3.28 分析如图3-68所示的有限状态机。写出状态转换和输出表，画出状态转换图。简短地介绍有限状态机的功能。

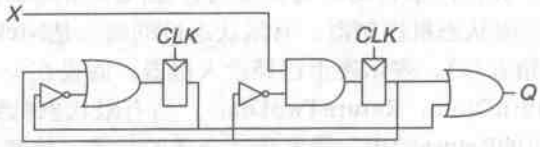


图3-68 有限状态机原理图

3.29 重复习题3.28，有限状态机如图3-69所示。注意寄存器的r和s输入分别表明复位和设置。

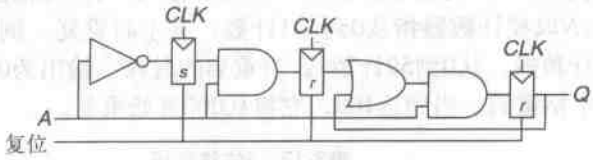


图3-69 有限状态机原理图

3.30 Ben设计了一个如图3-70的电路来计算4个寄存器输入的异或函数。每一个2输入异或门的传输延迟为100ps，最小延迟为55ps。每一个触发器的建立时间为60ps，保持时间为20ps，时钟到Q的最大延迟是70ps，时钟到Q的最小延迟是50ps。

(a) 如果不存在时钟脉冲相位差，电路的最大运行频率是多少?

(b) 如果电路必须工作在2GHz下，电路能够承受多大的时钟偏移?

(c) 在电路满足保持时间约束的条件下，电路能够承受多大的时钟偏移?

(d) Alyssa说她能够重新设计在输入/输出寄存器间的组合逻辑，使其他更快，且能够承受更大的时钟偏移。她的改进电路也使用2输入的异或门，但是它们的排列不同。她的电路是什么？如果不存在时钟偏移，它的最大频率是多少？在满足保持时间约束条件下，电路能够承受多大的时钟偏移?

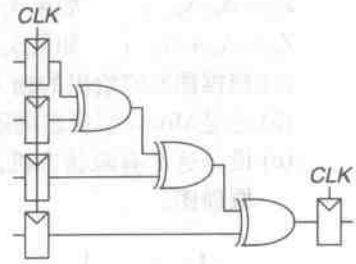


图3-70 4个寄存器输入的异或电路

3.31 为RePentium处理器设计一个2位加法器。这个加法器由2个全加器构成，第一个加法器的进位输出连接到第二个加法器的进位输入，如图3-71所示。加法器有输入和输出寄存器，必须在一个周期内完成加法运算。每一个全加器中，从C_{in}到C_{out}或者到Sum(S)的传输延迟为20ps，从A或者B到C_{out}的传输延迟为25ps，从A或者B到S传输延迟为30ps。加法器中，从C_{in}到其他输出的最小延迟为15ps，从A或者B到其他输出的最小延迟为22ps。每一个触发器的建立时间是30ps，保持时间是10ps，时钟到Q的传输延迟是35ps，时钟到Q的最小延迟是21ps。

(a) 如果不存在时钟偏移，电路的最大运行频率是多少?

(b) 如果电路必须工作在8GHz下，电路能够承受多大的时钟偏移?

(c) 在满足保持时间约束的条件下，电路能够承受多大的时钟偏移?

3.32 现场可编程门阵列（FPGA）使用可配置的逻辑块（CLBs）而不是逻辑门来实现组合逻

辑。Xilinx公司的Spartan 3 FPGA的每一个CLB的传输延迟和最小延迟分别是0.61ns和0.30ns。它包含的触发器的传输延迟和最小延迟分别是0.72ns和0.50ns。建立时间和保持时间分别是0.53ns和0ns。

(a) 如果你设计一个系统需要运行在40MHz，在2个触发器之间需要使用多少个连续的CLB？假设在CLBs之间没有时钟偏移，没有连线延迟。

(b) 假设在触发器之间的所有路径上至少通过一个CLB。FPGA有多少时钟偏移而不破坏保持时间约束。

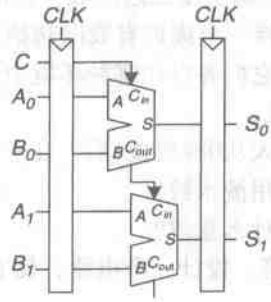


图3-71 2位加法器电路图

3.33 一对触发器建立的同步器，t_{setup} = 50ps，T_0 = 20 ps，τ = 30 ps。它对一个异步输入进行采样，每秒钟改变10^8次。这个同步器的最小时钟周期是多少，失效之间的间隔时间(MTBF)是100年。

3.34 设计一个接收异步输入的同步器，其MTBF为50年。系统运行主频为1GHz，采用T_0 = 110ps，τ = 100ps，t_{setup} = 70 ps的触发器进行采样。同步器每秒收到0.5次的异步输入(每2秒钟1次)。满足这个MTBF的失效概率是多少？在读出采样的输入信号之前你将等待多少周期以满足这个失效概率？

3.35 当你正在走廊下走路时遇见你的实验室伙伴正在朝另一个方向走。你们两个的第一步在同一条路上。然后你们两个同时踏上另一条相同的路。然后你们两个都等待一位时间，希望另一个人到另一边。你可以用亚稳态的观点为这种情景建模，并将相同的理论应用到同步器和触发器中去。假设你为你和你的实验室伙伴建立了一个数学模型。你们在亚稳态状态中相遇。在时间t秒后，你保持这种状态的概率是 $e^{\frac{-t}{\tau}}$，τ表示你的反应速度；今天你的大脑因为缺乏睡眠很模糊，τ = 20s。

(a) 需要多长时间，在有99%的可能性下你将从亚稳态中出来？(指穿过对方)

(b) 你不仅没有休息而且很饿。实际上，如果在3min内没到咖啡屋，你将饿死。你的实验室伙伴不得不把你推进太平间的概率是多少？

3.36 你使用T_0 = 20ps，τ = 30ps的触发器建立了一个同步器。你的老板需要将MTBF增加10倍，你需要将时钟频率升高多少？

3.37 Ben发明了一个改进的同步器如图3-72。他宣布在一个周期内消除了亚稳态。他解释在盒子M中的电路是一个逻辑亚稳态的检测器。如果输入电压在禁区V_{IL}和V_{IH}之间，产生一个为高电平的输出。通过检测，亚稳态检测器可以确定第一个触发器是否产生了一个亚稳态的输出D2。如果是，

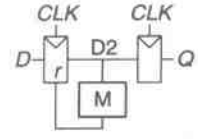


图3-72 改进的同步器

则它异步复位触发器使D2 = 0。第二个触发器对D2进行采样，在Q上总是可以产生一个有效的逻辑电平。Alyssa告诉Ben，电路中存在一个错误，因为消除亚稳态就像制造永动机一样不可能。谁是正确的？解释并说明Ben或Alyssa的错误。

面试问题

3.1 画一个状态机用于检测接收到的序列01010。

3.2 设计一个串行（每次一位）二进制补码有限状态机。它有两个输入Start和A，一个输出Q。输入A是一个从最低位开始的任意长度二进制数。在同一周期内，Q输出相同的位。在最小的有效位输入之前，Start保持一个周期有效以初始化有限状态机。

3.3 锁存器和触发器有什么不同？它们各自在哪种环境中更加可取？

3.4 设计一个5位计数器有限状态机。

3.5 设计一个边沿检测电路。在输入从0转变成1后，在一个周期内输出为高电平。

3.6 描述流水线的概念，为什么使用流水线？

3.7 描述触发器中负的保持时间是什么意思？

3.8 如图3-73，给出了信号A的波形，设计一个电路，使它产生信号B。

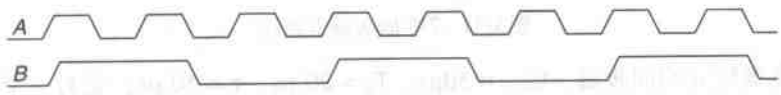


图3-73 信号波形

3.9 考虑在两个寄存器之间的组合逻辑块。解释时序约束。如果你在接收器方（第二个触发器）的时钟输入加入一个缓冲器，建立时间约束是变好了，还是变坏了？

第4章 硬件描述语言

4.1 引言

到现在为止，我们在原理图层面集中讨论了组合电路和时序电路的设计。需通过手工简化真值表和布尔表达式的方法来寻找一套有效的逻辑门电路来实现给定功能，同时还需要手工将有限状态机转换成逻辑门，这个过程非常麻烦，而且还容易出错。在20世纪90年代，设计者们发现如果他们工作在更高的抽象层，只说明逻辑的功能，同时引入计算机辅助设计（computer-aided design, CAD）工具去生成优化的门电路，可以获得更高的设计效率。这些对硬件的说明一般通过硬件描述语言（Hardware Description Language，HDL）给出。现在有两种主要的硬件描述语言，分别是Verilog语言和VHDL语言。

Verilog 和VHDL的基本原则相似，但是两者的语法却有所不同。在本章中，当讨论这两种语言时，我们会把Verilog放在前边，VHDL放在后边，分为两栏作比较。当读者第一次阅读本章时，只需要关注其中一种语言。掌握一种语言后，在需要的时候也可以很快掌握另外一种。

后续章节的硬件会以电路图和HDL等两种形式表示。如果读者选择跳过本章，不学习硬件描述语言，仍然可以通过学习电路图掌握计算机组成的基本原理。然而，现在大部分的商业系统都是使用HDL语言设计，而非原理图。如果读者希望在专业生涯中从事电子设计工作，那么学习一种硬件描述语言是必要的。

4.1.1 模块

一个包括输入和输出的硬件块称为模块（module）。与门、多路选择器和优先级电路都是硬件模块的例子。描述模块功能的主要形式有两种：行为模型（behavioral）和结构模型（structural）。行为模型描述一个模块做什么。结构模型应用层次化方法描述一个模块怎样由更简单的部件构造。HDL例子4.1中的Verilog和VHDL代码给出了例子2.6中布尔函数$y = \overline{abc} + a\overline{bc} + \bar{a}bc$的行为描述。在两种语言中，模块都被命名为sillyfunction，具有三个输入a、b、c和一个输出y。

HDL例子4.1 组合逻辑

Verilog

```
module sillyfunction(input a, b, c,
                     output y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;

endmodule
```

一个Verilog的模块以模块名以及输入和输出的列表开始。assign声明用于描述组合逻辑。~表示非，&表示与，|表示或。

Verilog的输入、输出信号是布尔变量（0和1）。它们也有浮空值和未定义值，这些将会在4.2.8小节讨论。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= ((not a) and (not b) and (not c)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c);
end;
```

VHDL代码有三个部分：库（library）调用子句，实体（entity）声明和结构（architecture）体。库调用子句是必需的，这将在4.2.11小节讨论。实体声明列出了模块的名字、输入和输出。结构体定义模块做什么。

VHDL信号（如输入或输出）必须声明类型。数字信号必须被声明为STD_LOGIC类型。STD_LOGIC信号可以为“0”，“1”值，浮空值或未定义值，这将在4.2.8小节讨论。STD_LOGIC在库IEEE.STD_LOGIC_1164中定义，所以必须调用这个库。

VHDL没有默认的操作顺序，所以布尔等式必须添加括号。

正如读者所预期的，模块是模块化的一个好的应用。它具有由输入、输出组成的接口，而且完成了特定功能。更进一步，其编码内容对于其他调用它的部分来说并不重要，只要它实现了自己的功能。

4.1.2 硬件描述语言的起源

在这两种硬件描述语言的选择上，大学一般无偏见地选择一种在课堂上讲授，而在工业界则基于喜好。作为一个由委员会发展起来的语言，VHDL比Verilog语句冗长且不灵活。美国军事承包商、欧洲太空总署和电信公司普遍采用VHDL。

两种语言都足以描述任何的硬件系统，也都有各自的特点。最好的语言就是在你的环境里已经使用的语言，或者是客户要求使用的语言。当前的大部分计算机辅助设计工具都允许两个语言混合使用，不同的模块可以用不同的语言描述。

Verilog

Verilog由Gateway Design Automation于1984年作为一个逻辑模拟的专用语言开发。Gateway公司于1989年被Cadence公司收购，在Open Verilog International组织的控制下，Verilog于1990年成为一个公开的标准，1995年成为IEEE标准（IEEE STD 1364），并在2001年更新。

VHDL

VHDL是VHSIC Hardware Description Language的首字母缩写，而VHSIC是美国国防部程序Very High Speed Integrated Circuits的首字母缩写。

VHDL最初是于1981年由美国国防部开发，目的在于描述硬件的结构和功能。它由Ada编程语言发展而来。IEEE于1987年将其标准化（IEEE STD 1076），之后又几次更新了这个标准。VHDL最初预想是用作文档化，但是很快就改为用作模拟和综合。

4.1.3 模拟和综合

两种硬件描述语言的主要目的是逻辑的模拟（simulation）和综合（synthesis）。在模拟阶段，在模块上加入输入，并检查输出以验证模块的操作是否正确。在综合阶段，将模块的文字描述转换成逻辑门。

模拟

人类周而复始地犯错误。这些错误在硬件设计里称作漏洞（bug）。显然，去除数字系统中这些漏洞十分重要，特别是当用户正在进行金钱交易或者性命攸关的操作时。在实验室测试一个系统是很花费时间的。而在实验室发现引起错误的原因可能非常困难，因为只有发送到芯片管脚的信号才能被观察到，没有方法直接观察在芯片里面发生了什么。在系统完成后才改正错误付出的代价极度昂贵。例如，改正一个尖端集成电路里的错误，会花费百万美金以上的金钱并耽误数月的时间。英特尔公司的奔腾处理器中，曾经出现过一个浮点除法（floating point division，FDIV）漏洞，迫使公司在交货之后又重新召回芯片，总共花费了4.75亿美元。可见在系统建立之前进行逻辑模拟是很必要的。

图4-1显示了之前sillyfunction模块的模拟波形，说明模块工作正常。正如布尔表达式表示的，当a，b，c为000，100和101时，y的值是true。

综合

逻辑综合将HDL代码转换成描述硬件（例如，逻辑门和连接它们的线）的网表（netlist）。逻辑综合可以进行优化以减少硬件的数量。网表可能是一个文本文件，也可以以原理图的形式绘制出来，使电路可视化。图4-2显示了sillyfunction模块综合的结果。注意：三个三输入与门是怎样被简化成两个二输入与门的，正如例2.6中使用布尔代数化简一样。

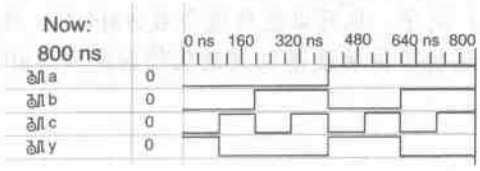


图4-1 模拟波形图

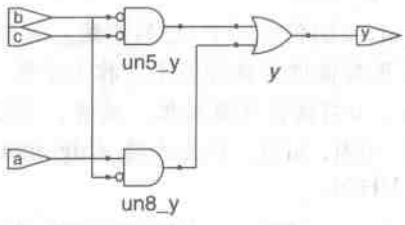


图4-2 综合后的电路

HDL中的电路描述与编程语言的代码很相像。然而，你必须记住这里的代码是代表硬件的。Verilog和VHDL有丰富的命令。并不是所有的命令都可以被综合成硬件。例如，一个在模拟时用于在屏幕上打印结果的命令，就不会被转换成硬件。因为我们的主要兴趣是构造硬件，所以会强调语言中可综合的子集。具体来说，我们会把HDL代码分成可综合（synthesizable）模块和测试程序（testbench）。其中可综合模块描述了硬件；测试程序包含了为模块提供输入的代码，以检测输出结果是不是正确，并打印出期待结果和实际结果的差别。测试程序代码只用以模拟，而不能用于综合。

对于初学者来说，一个最常见的错误就是认为HDL是计算机程序而不是描述电子硬件的快速方法。如果不能大致了解HDL要综合成的硬件，那也很可能不会得到满意的结果。可能会产生多余的硬件，也可能会写出模拟正确但却不能在硬件上实现的代码。因此，应该以组合逻辑块、寄存器、有限状态机的形式考虑一下系统。在你开始编写代码之前，首先在纸上描绘这些模块，并看看它们是怎样连接的。

以我们的经验，初学者学习HDL的最好方法就是通过例子。HDL有一些特定的方法描述各种逻辑；这些方法被称为风格（idiom）。本章将讲述如何针对每一类模块采用合适的风格书写HDL代码，然后如何把各个块聚集起来以成为一个可以工作的系统。当需要描述一个特别类型的硬件时，首先需要查看相似的例子，然后修改例子使之适合特定需要。我们不会严格地定义所有的HDL语法，因为那十分无聊，而且这样会使大家把HDL看作是一种编程语言，而不是对硬件的描述。IEEE Verilog、VHDL规范和大量详尽的书籍包括了所有的细节，可以帮助读者在特别的主题上获得更多的信息（参见本书后面的“延伸阅读材料”）。

4.2 组合逻辑

我们一直在学习基于组合逻辑和寄存器的同步时序电路设计。组合逻辑的输出只依赖当前的输入。这一节描述如何用HDL去编写组合逻辑的行为模型。

4.2.1 按位操作符

按位（bitwise）操作符对单独位信号和多位总线进行操作。例如，在HDL例子4.2里，inv模块描述4个连接到4位总线的反相器。

HDL例子4.2 反相器

Verilog

```
module inv(input  [3:0] a,
           output [3:0] y);

  assign y = ~a;
endmodule
```

a[3：0]代表一个4位的总线。这些位从最高端到最低端分别是a[3]、a[2]、 a[1]、a[0]。因为这里最低位的位号最小，称为小端（little-endian）顺序。也可以把总线命名为a[4:1]。这样的话，a[4]就会在最高位。或者，可以用a[0：3]，这样，位从最高位到最低位就分别为a[0]、a[1]、a[2]、a[3]，称为大端（big-endian）顺序。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL使用STD_LOGIC_VECTOR代表STD_LOGIC总线。STD_LOGIC_VECTOR(3 downto 0)代表一个4位总线。从最高位到最低位的位分别是3、2、1、0。因为最低位的位号最小，称为小端（little-endian）顺序。也可以声明总线为STD_LOGIC_VECTOR（4 downto 1）。这样，第4位就成为最高位。还可以写为STD_LOGIC_VECTOR(0 to 3)。这样，从最高位到最低位就是0、1、2、3，称为大端（big-endian）顺序。

反相器综合后电路如图4-3所示。

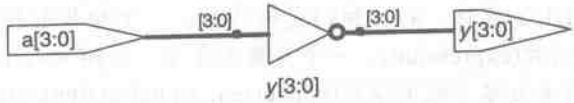


图4-3 inv综合后电路

总线大端或小端的选择是任意的。实际上，在上面的例子中，位顺序并不重要，因为这一组反相器并不关心位顺序。位顺序只与操作符有关，例如加法操作中要把前一列的进位输出到下一列中。只要保持一致，可以采用任意一种位顺序。我们将一直沿用小端顺序，对于N位的总线，在Verilog中是[N-1:0]，在VHDL中是（N-1 downto 0）。

在本章及以后的代码例子都采用了Synplify Pro 综合工具根据Verilog代码生成的原理图。图4-3显示inv模块综合为4个反相器，用反相器符号y[3：0]表示。反相器连接4位的输入和输出总线。相似的硬件也可从可综合的VHDL代码中产生。

HDL例子4.3中的gates模块表示了在4位总线上的按位运算实现其他基本逻辑功能。

HDL例子4.3 逻辑门

Verilog

```
module gates(input  [3:0] a, b,
             output [3:0] y1, y2,
                          y3, y4, y5);

  /* Five different two-input logic
     gates acting on 4 bit busses */
  assign y1 = a & b;    // AND
  assign y2 = a | b;    // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```

~、^和|都是Verilog运算符（operator），而a、b和y1是操作数（operand）。一个运算符和操作数的组合，如a & b，或~（a | b），称为表达式。一个完整的命令，如assign y4 =~(a&b)，称为语句。

assign out = in1 op in2，称为连续赋值语句（continuous assignment statement）。连续赋值语句以一个分号结束。连续赋值语句中，等号=右边的输入值一旦改变，等号左边的输出就会随之重新赋值。因此，连续赋值语句用于描述组合逻辑。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of gates is
begin
  -- Five different two-input logic gates
  --acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

not、xor和or都是VHDL运算符，a、b和y1是操作数。一个操作数和运算符的组合，如a and b，或a nor b，称为表达式(expression)。一个完整的命令，如y4 <= a nand b，称为语句。

out <= in1 op in2;称为并发信号赋值语句(concurrent signal assignment statement)。VHDL赋值语句以一个分号结束。并发信号赋值语句中，<=号右边的输入值一旦改变，左边的输出值就会随之重新赋值。因此，并发信号赋值语句用于描述组合逻辑。

图4-4给出了逻辑门模块综合后电路。

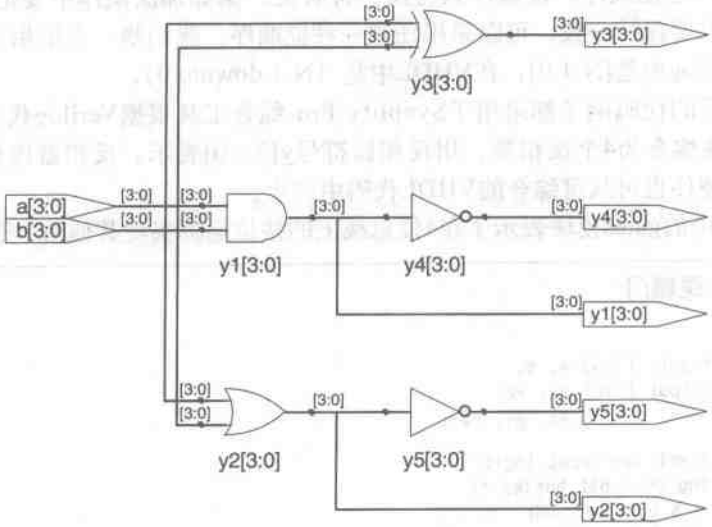


图4-4 gates模块综合后电路

4.2.2 注释和空格

上面gates例子展示了如何注释。Verilog和VHDL对空格并不敏感（包括：空格，tab键，换行）。然而，合理使用缩进排版和换行可以增加复杂设计的可读性。信号名和模块名命名时使用大写字母和下划线。模块名和信号名不能以数字开头。

Verilog

Verilog的注释与C和Java的一样。注释以/*开始，之后可以延续多行，最后以一个*/结束。以//开始的注释则一直延续到本行的末尾。

Verilog是大小写敏感的，y1和Y1在Verilog中被认为是不同的信号。

VHDL

VHDL注释以--开始，一直延续到本行的末尾。如果遇到连续几行的注释，就需要在每行的开始位置添加--。

VHDL是大小写不敏感的，y1和Y1在VHDL语言中是一致的。然而，其他读取文件的工具可能是大小写敏感的，轻易混用大小写的习惯就可能带来一些繁琐的错误。

4.2.3 缩减运算符

缩减运算符表示一个总线上操作的多输入门。HDL例子4.4描述了一个八输入的与门，其输入分别是a_7, a_6, …, a_0。

HDL例子4.4 八输入与门（图4-5显示了8输入与门综合后电路。）

Verilog

```
module and8(input [7:0] a,
            output      y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

和你想像的一致，|，^ ,~& 和~|这些缩减运算符和OR，XOR，NAND和NOR一样可用。回想一下，用多输入XOR实现奇偶校验，如果奇数个输入是TRUE，则返回一个TRUE。

VHDL

VHDL没有缩减运算符。但是它提供了一个generate命令（参阅4.7节）。作为替代，这个操作可以如下显式地表达。

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= a(7) and a(6) and a(5) and a(4) and
       a(3) and a(2) and a(1) and a(0);
end;
```

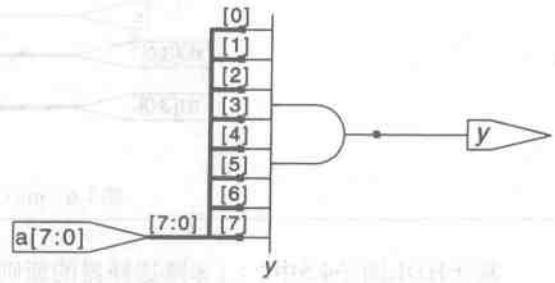


图4-5 and8综合后电路

4.2.4 条件赋值

条件赋值（conditional assignment）根据条件在所有可选的输入中选择一个输出。HDL例子4.5说明了一个使用条件赋值的2：1多路转换器。

HDL例子4.5 2：1多路选择器（图4-6给出2：1多路选择器综合后电路。）

Verilog

条件运算符（conditional operator）?: 基于第一表达式在第二和第三表达式之间选择。第一个表达式称为条件。如果条件判断值为1，那么操作就选择第二个表达式。如果条件是0，那么操作就选择第三个表达式。

?:对于描述多路选择器特别有用，因为它根据第一个输入，在之后的两个表达式中选择。以下的代码说明了用条件赋值实现4位输入和输出的2：1多路转换器的风格。

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

如果s为1，那么y = d1。如果s为0，那么y = d0。

?:称为三元运算符，因为它有三个输入。在C和Java编程语言中，它也有着相同的用途。

VHDL

条件信号赋值（conditional signal assignment）会基于不同的条件有不同的操作。它们在描述一个多路选择器时特别有用。例如，2：1多路选择器可以使用条件信号赋值从4位输入中选择一个。

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

如果s为0，条件信号赋值把d0赋给y。否则，把d1赋给y。

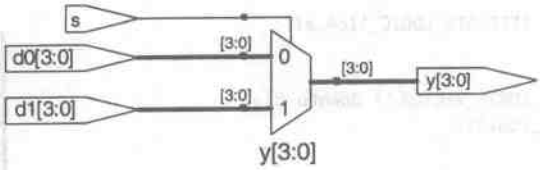


图4-6 mux2综合后电路

基于HDL例子4.5中2：1多路选择器的原则，HDL例子4.6给出了一个4：1多路选择器。

图4-7表示了一个用Synplify Pro产生的4：1多路选择器图。这个软件使用的多路选择器符号与本文目前使用的不同。多路选择器有多路数据（d）和独热使能（e）输入。当一个使能信号有效时，对应的数据就被传送到输出。例如，当s[1]=s[0]=0时，底部的与门un1_s_5产生信号1，使能多路选择器的底部输入，并使它选择d0[3：0]。

HDL例子4.6 4：1多路选择器

Verilog

一个4：1多路选择器可以用嵌套的条件运算符实现从4个输入中选择1个。

```
module mux4(input  [3:0] d0, d1, d2, d3,
            input  [1:0] s,
            output [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
           : (s[0] ? d1 : d0);
endmodule
```

如果s[1]为1，那么多路选择器选择第一个表达式（s[0]？d3：d2）。紧接着，表达式会基于s[0]选择d3或者d2（如果s[0]为1，则y = d3，如果s[0]为0，则y = d2）。如果s[1]为0，那么多路选择器会选择第二个表达式，这时会基于s[0]的值选择d1或者d0。

VHDL

一个4：1多路选择器使用条件信号赋值中的多重else子句实现从4个输入中选择1个。

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
```

```
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL也支持选择信号赋值语句（selected signal assignment statement）以提供从多个可能值中选择一个的简便方法。这与在一些编程语言中使用一个case语句代替多个if/else语句相似。4：1多路选择器可以重新用选择信号赋值语句写出：

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

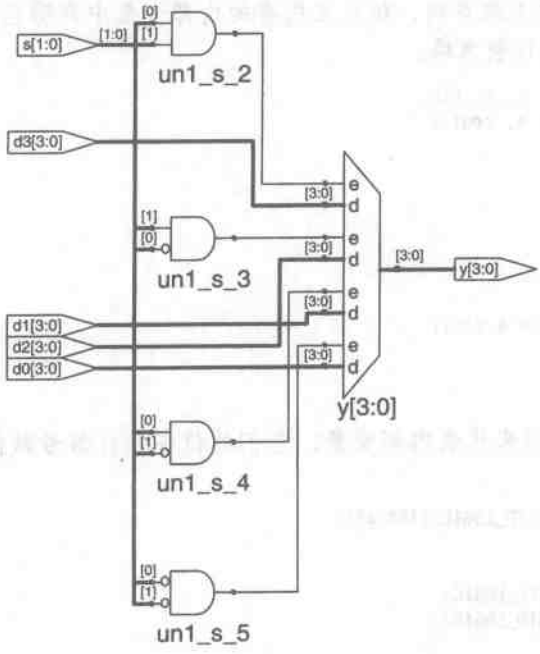


图4-7 mux4综合后电路

4.2.5 内部变量

通常来说，把一个复杂功能分为几个中间过程来完成会更方便。例如，将在5.2.1节描述的全加器是一个有3个输入和2个输出的电路。它由以下等式定义：

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in} \quad (4\text{-}1)$$

如果我们定义中间信号P和G，

$$P = A \oplus B$$
$$G = AB \quad (4\text{-}2)$$

则可以重写全加器，如下所示：

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in} \quad (4\text{-}3)$$

P和G称为内部变量(internal variable)，因为它们既不是输入也不是输出，只在模块内部使用，与编程语言中的局部变量相似。HDL例子4.7表示了在HDL中如何使用内部变量。

HDL赋值语句（Verilog的assign和VHDL的<=）是并行执行的。这与常见的编程语言（如C和Java）不同，其语句执行顺序按书写顺序决定。在常见的编程语言中，由于语句是顺序执行的，因此$S = P \oplus C_{in}$必须放在$P = A \oplus B$之后。在HDL中，顺序并不重要。就像硬件一样，赋值语句在右侧的输入信号改变时就会被计算，而不考虑赋值语句在模块里的出现顺序。

HDL例子4.7 全加器（图4-8给出了全加器综合后电路）

Verilog

在Verilog中，wire用于表示内部变量。它们的值用assign语句定义，如assign p=a^b; 虽然wire只是在多位总线时才必须声明，但是在所有的内部变量中声明它们是一个好的习惯。它们的声明在这个例子中可以被忽略。

```
module fulladder(input  a, b, cin,
                 output s, cout);

  wire p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g  |  (p & cin);
endmodule
```

VHDL

在VHDL中，signal用来代表内部变量。它们的值在并行信号赋值语句中定义，如p<=a xor b;

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

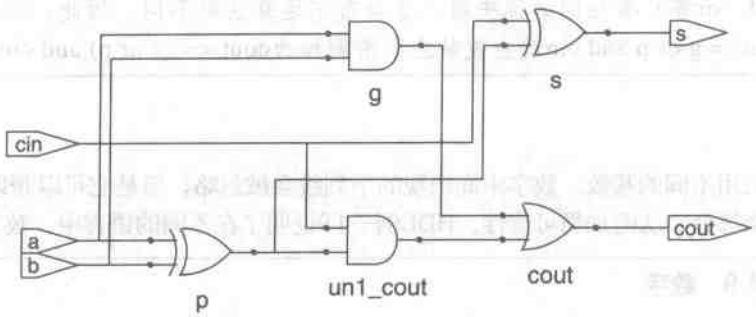


图4-8 fulladder综合后电路

4.2.6 优先级

注意到在HDL例子4.7中为cout的计算添加了括号，从而把计算定义为$C_{out} = G + (P \cdot C_{in})$，而不是$C_{out} = (G + P)C_{in}$。如果没有使用括号，就采用编程语言定义的默认计算顺序。HDL例子4.8说明了两个语言的运算符优先级，按从高到低的顺序给出。这个表包括了将在第5章定义的算术运算符、移位运算符和比较运算符。

HDL例子4.8 运算符优先级

Verilog运算符优先级与VHDL运算符优先级分别如表4-1和表4-2所示。

Verilog

表4-1 Verilog运算符优先级

	运算符	含 义		运算符	含 义
最高	~	取反		==, !=	相等比较
	*, /, %	乘，除，模		&, ~&	AND, NAND
	+, −	加，减		^, ~^	XOR, XNOR
	<<, >>	逻辑左/右移位		|, ~|	OR, NOR
	<<<, >>>	算术左/右移位	最低	?:	条件
	<, <=, >, >=	关系比较			

如你所想像的，在Verilog中的运算符优先级与其他的编程语言很相似。特别要说明的是，“与”的优先级比“或”的优先级高。我们可以利用这个优先级消除括号。

```
assign cout = g | p & cin ;
```

VHDL

表4-2 VHDL运算符优先级

	运算符	含 义		运算符	含 义
最高	not	取反		sra, sla	算术移位
	*, /, mod, rem	乘，除，模，余		=, /=, <,	比较运算
	+, −,	加，减		<=, >, >=	
	&			and, or, nand,	逻辑运算
	rol, ror,	循环移位	最低	nor, xor	
	srl, sll,	逻辑移位			

如你想像，VHDL中乘法比加法有更高的优先级。和Verilog不同的是，VHDL中所有的逻

辑运算符（and，or等）有相同的优先级，这与布尔运算法则不同。因此，括号的使用就很必要，否则，cout<= g or p and cin就会被从左到右解释为cout <= (g or p) and cin.

4.2.7 数字

数字可以采用不同的基数。数字中间出现的下划线会被忽略，但是它可以帮助我们把一些长的数字断成几个部分，从而加强可读性。HDL例子4.9说明了在不同的语言中，数字是如何书写的。

HDL例子4.9 数字

Verilog

Verilog的数字（见表4-3）可以说明基数和大小（用于代表它们的位数）。声明常量的格式是N′Bvalue，其中N表示位数，B是基数，value为给定的值。例如9′h25说明一个占用9位的数字，它的值是$25_{16} = 37_{10} = 000100101_2$。Verilog用'b表示二进制（基数为2），'o表示八进制（基数为8），'d表示十进制（基数为10），'h表示十六进制（基数为16）。如果没有提供基数的话，那么基数默认为10。

如果位数没有给出，那么数字就会被赋予当前表达式的位数。零会自动填补在数字的前面以达到满位。例如，如果w是6位总线，assign w = 'b11就会指定w的值为000011。这比明确地说明位数的大小更好。

VHDL

在VHDL中，STD_LOGIC数字（见表4-4）以二进制书写，并且加上单引号：'0' 和 '1' 代表逻辑的0和1。STD_LOGIC_VECTOR数字以二进制或者十六进制表示，并使用双引号标记。默认二进制为基数，也可以显式地用前缀X表示十六进制或者前缀B表示二进制。

表4-3 Verilog的数字

数字	位数	基数	值	存储
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010

表4-4 VHDL的数字

数字	位数	基数	值	存储
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	161	10101011

4.2.8 z和x

HDL用z表示浮空。z对于描述三态缓冲器尤其有用。当使能位为0时，它的输出为浮空。回想2.6节，总线可以由多个三态缓冲器驱动，但其中最多只有一个使能。HDL例子4.10展示

了一个三态缓冲器的风格。如果缓冲器被使能，则输入和输出一致。如果缓冲器被禁用，输出就为浮空值（z）。

HDL例子4.10 三态缓冲器（图4-9给出三态缓冲器综合后电路）

Verilog

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= "ZZZZ" when en = '0' else a;
end;
```

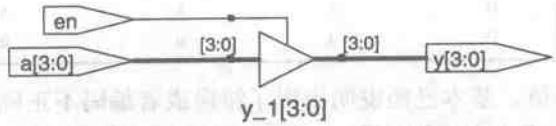


图4-9 tristate综合后电路

同样，HDL使用x表示一个无效的逻辑电平。如果一个总线被两个使能的三态缓冲器（或其他的门器件）同时驱动为0和1，则结果将是x，说明发生了冲突。如果所有的三态缓冲器以OFF同时驱动总线，那么总线将会用z表示浮空。

开始模拟时，触发器等状态节点的输出会被初始化一个未知的状态（在Verilog中是x，在VHDL中是u）。这对追踪因为在使用输出前忘记复位触发器而引起的错误十分有帮助。

如果一个门接收到一个浮空的输入，它不能确认正确的输出值，将在输出产生一个x值。类似地，如果它接收到一个无效或者未初始化的输入，它也会输出一个x值。HDL例子4.11表示了Verilog和VHDL逻辑门如何组合这些不同的信号值。

HDL例子4.11 未定义和浮空输入的真值表

Verilog

Verilog的信号值有0，1，z和x。在需要的时候，以z或者x开始的Verilog常量都会以z或者x（而不是0）填充，并填满位数。

表4-5列出了使用全部4个可能信号值的与门真值表。注意：有些时候即使一些输入值还是未知，门也可以决定输出值。例如0 & z会返回0，因为与门只要有一个输入为0，那么输出就会是0。另外，浮空或者不可用的输入会导致不可用的输出，在Verilog中以x表示。

VHDL

VHDL的STD_LOGIC信号有'0'，'1'，'z'，'x'和'u'。

表4-6列出了一个使用了所有5个可能信号值的与门真值表。注意：有些时候即使一些输入值还是未知，门也可以决定输出值。例如'0' and 'z'会返回0，因为与门只要有一个输入为'0'，那么输出就会是'0'。另外，浮空或者不可用的输入会导致不可用的输出，在VHDL中以'x'表示。未初始化的输入会产生未初始化的输出，在VHDL中以'u'表示。

表4-5 Verilog中带z和x的与门真值表

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

表4-6 VHDL中带z，x和u的与门真值表

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

在模拟时看到x或者z值，基本已经说明出现了错误或者编码不正确。在综合后的电路中，这表示门输入浮空、状态或内容未初始化。x或者u可能会被电路随机地解释为0或者1，导致不可预测的行为。

4.2.9 位混合

常常需要在总线的子集上操作，或者连接信号以构成总线。这些操作称为位混合(bit swizzling)。HDL例子4.12中，y用位混合操作被赋予9位的值$c_2c_1d_0d_0d_0c_0101$。

HDL例子4.12 位混合

Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

{}运算符用作连接总线。{3{d[0]}}表示3个d[0]连接在一起。

不要将3位二进制常量3'b101与以b命名的总线混淆起来。注意：常量中3位长度的说明很重要，否则在y的中间就会出现一个以零开头的未知数。

如果y长度大于9位，那么会在最高端填充0。

VHDL

```
y <= c(2 downto 1) & d(0) & d(0) & d(0) &
     c(0) & "101";
```

&运算符用于连接总线。y必须是一个9位的STD_LOGIC_VECTOR。不要将&和VHDL中使用的and运算符混淆起来。

4.2.10 延迟

HDL的语句可以与任意单位的延迟相关联。这对于为预测电路工作速度而进行的模拟（若指定了有意义的延迟），或者调试时需要知道原因和后果时（模拟结果中所有信号是同时改变的，因此推断一个错误输出的根源就非常棘手），就显得很有用。延迟在综合时会被忽略；综合器产生的门延迟是由t_{pd}和t_{cd}决定的，而不取决于在HDL代码中的数字。

在HDL例子4.1中 $y = \overline{abc} + a\overline{bc} + a\bar{b}c$的功能上，HDL例子4.13加上了延迟。假定反相器延迟为1ns，三输入与门延迟为2ns，三输入或门延迟为4ns。图4-10显示了输入后延迟7ns的y模拟波形。注意y在一开始模拟的时候是未知值。

HDL例子4.13 带延迟的逻辑门

Verilog

```
`timescale 1ns/1ps

module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Verilog文件可以用时间精度指令来说明每一个时间单位的值。语句的格式为`timescale unit/precision。在这个文件中，时间单位为1ns，模拟精度为1ps。如果在文件中没有给出时间精度指令，就会使用默认的单位和精度（一般两者都是1ns）。在Verilog中，#符号用于说明延迟单位的数量。它可以放置于assign语句中，就像非阻塞式（<=）和阻塞式（=）赋值语句一样，这两类语句会在4.5.4小节中讨论。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y <= n1 or n2 or n3 after 4 ns;
end;
```

在VHDL中，after子句用于说明延迟。在这个例子中，指定单位为ns。

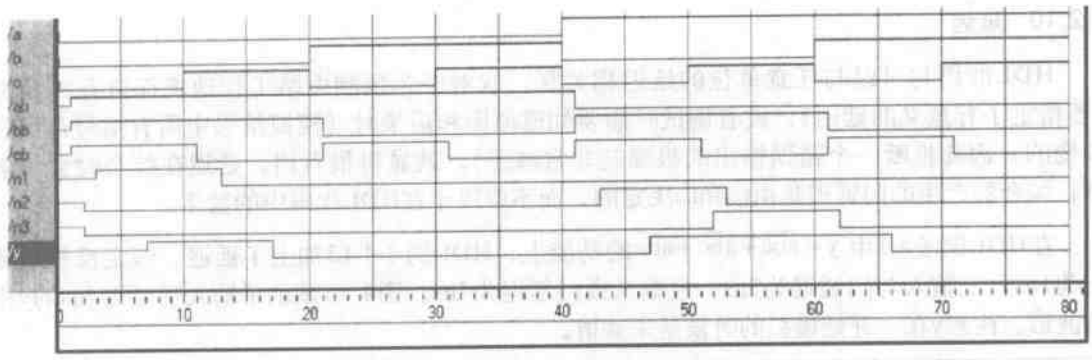


图4-10 带延迟的模拟波形示例（来自于ModelSim模拟器）

4.2.11 VHDL 库和类型*

（Verilog的使用者可以跳过本小节）与Verilog不同，VHDL要求一个严格的数据类型系统以保证用户不出现错误，尽管有时这也会显得有些冗繁。

尽管STD_LOGIC非常基本和重要，但是却没有直接建立在VHDL内。相反，它包含在IEEE.STD_LOGIC_1164库之中。因此，在之前例子中的所有文件都必须包含库语句。

而且IEEE.STD_LOGIC_1164缺少对于STD_LOGIC_VECTOR数据的基本操作，例如整数相加、比较、移位和转换等操作。大部分CAD供应商已经开始采用包含更多这些功能的库：IEEE.STD_LOGIC_UNSIGNED 和 IEEE.STD_LOGIC_SIGNED。请查看1.4小节关于无符号和有符号数的讨论和操作的例子。

VHDL还有一个BOOLEAN类型，包含两个值：true 和false。BOOLEAN值由比较操作返回（如等值比较s = '0'）。它也用在条件语句中，如when语句。尽管我们可能会认为BOOLEAN类型的true值应该等于STD_LOGIC的'1'，BOOLEAN类型的false值应该等于STD_LOGIC的'0'，但实际上这两个类型是不可以互换的。因此，以下的代码不合法：

```
y <= d1 when s else d0;
q <= (state = S2);
```

应该如下书写：

```
y <= d1 when (s = '1') else d0;
q <= '1' when (state = S2) else '0';
```

尽管我们没有声明任何信号为BOOLEAN类型，但是它们会在比较式中自动、隐式地产生，并在条件语句中使用。

```
y <= a(3) and a(2) and a(1) and a(0);
```

相似地，VHDL用INTEGER类型表示正整数和负整数。INTEGER类型的信号取值范围为$-2^{31}\sim2^{31}-1$。整型数可用作总线的标识。例如在语句中

0、1、2和3就是用于选择a信号位的索引整数。我们不能直接用STD_LOGIC或者STD_LOGIC_VECTOR信号去标识总线，而是必须把信号转换成INTEGER类型。这将由HDL例子4.14说明。这个例子是一个8：1多路转换器，用3位索引从向量中选择一位。CONV_INTEGER 函数在IEEE.STD_LOGIC_UNSIGNED库中定义，其作用是把STD_LOGIC_

VECTOR类型转换成INTEGER类型的无符号正整数。

VHDL严格定义out端口只能用于输出。例如，以下关于二和三输入与门就是非法的VHDL代码，因为v为输出而它又被用于计算w。

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in  STD_LOGIC;
       v, w:    out STD_LOGIC);
end;

architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

HDL例子4.14 带类型转换的8：1多路选择器

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(CONV_INTEGER(s));
end;
```

图4-11给出8：1多路选择器综合后电路。

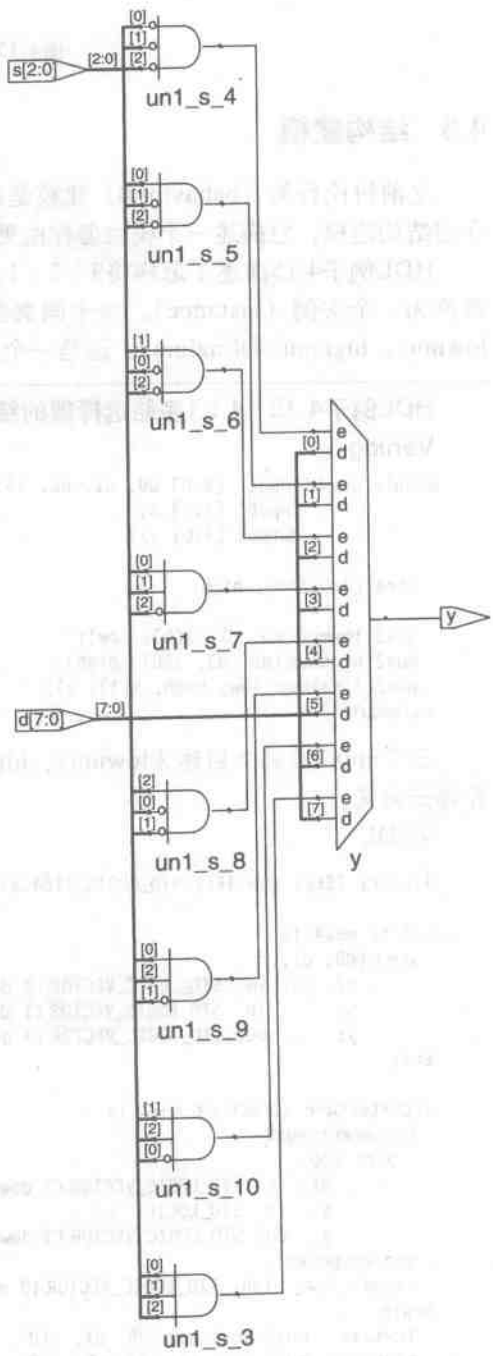


图4-11 mux8综合后电路

VHDL定义了一个特别的端口类型buffer去解决这个问题。连接到buffer端口的信号像输出一样工作，并可以在模块内使用。以下是改正的实体定义（and 23综合后电路如图4-12所示）。Verilog没有这个限制，所以也不需要buffer端口。

```
entity and23 is
  port(a, b, c: in     STD_LOGIC;
       v:       buffer STD_LOGIC;
       w:       out    STD_LOGIC);
end;
```

VHDL支持枚举（enumeration）类型，用抽象的方法代表没有明确指定二进制编码内容。例如，3.4.2节描述的以3作循环的有限状态机使用了3个状态。我们可以用枚举类型给出状态名，而不用二进制码给出。这个功能非常强大，因为它允许VHDL综合时选择最好的状态编码，而不是依靠用户指定的任意编码。

```
type statetype is (S0, S1, S2);
signal state, nextstate: statetype;
```

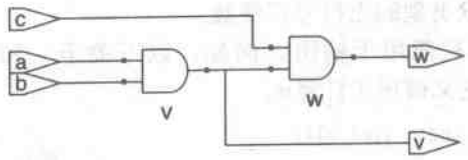


图4-12 and23综合后电路

4.3 结构建模

之前讨论行为（behavioral）建模是通过建立输入和输出之间的关系来描述模块。本节将介绍结构建模，它描述一个模块怎样由更简单的模块组成。

HDL例子4.15描述了怎样将3个2：1多路转换器组成一个4：1多路转换器。每个2：1转换器称为一个实例（instance）。多个同类型模块的实例由唯一的名字区分，在此例子中分别为lowmux，highmux和finalmux。这是一个规整的范例，2：1多路转换器重用了多次。

HDL例子4.15 4：1多路选择器的结构模型

Verilog

```
module mux4 (input  [3:0] d0, d1, d2, d3,
             input  [1:0] s,
             output [3:0] y);

  wire [3:0] low, high;

  mux2 lowmux (d0, d1, s[0], low);
  mux2 highmux (d2, d3, s[0], high);
  mux2 finalmux (low, high, s[1], y);
endmodule
```

三个mux2实例分别称为lowmux、highmux和finalmux。mux2模块必须在Verilog代码的另外部分定义。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port (d0, d1,
        d2, d3: in  STD_LOGIC_VECTOR (3 downto 0);
        s:      in  STD_LOGIC_VECTOR (1 downto 0);
        y:      out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port (d0,
          d1: in  STD_LOGIC_VECTOR (3 downto 0);
          s:  in  STD_LOGIC;
          y:  out STD_LOGIC_VECTOR (3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR (3 downto 0);
begin
  lowmux:   mux2 port map (d0, d1, s(0), low);
  highmux:  mux2 port map (d2, d3, s(0), high);
  finalmux: mux2 port map (low, high, s(1), y);
end;
```

在architecture部分必须先用component声明语句声明mux2的端口。这会让VHDL工具检查你想使用的组件和在代码其他部分声明的实体是否具有相同的端口，从而避免改变了实体但是没有改变实例而引起的错误。然而，component的声明使VHDL代码变得冗长。

注意到这个mux4的architecture被命名为struct，然而在4.2小节中，模块中包含行为描述的architecture却被命名为synth。VHDL允许在同一实体中有多个architecture（实现），这些architecture以名字区分。名字本身对CAD工具意义不大，只是struct和synth比较常用。可综合的VHDL代码一般在一个实体中只有一个architecture，所以我们不讨论在定义多个architecture的时候VHDL如何设置的语法。

4：1多路选择器综合后电路如图4-13所示。

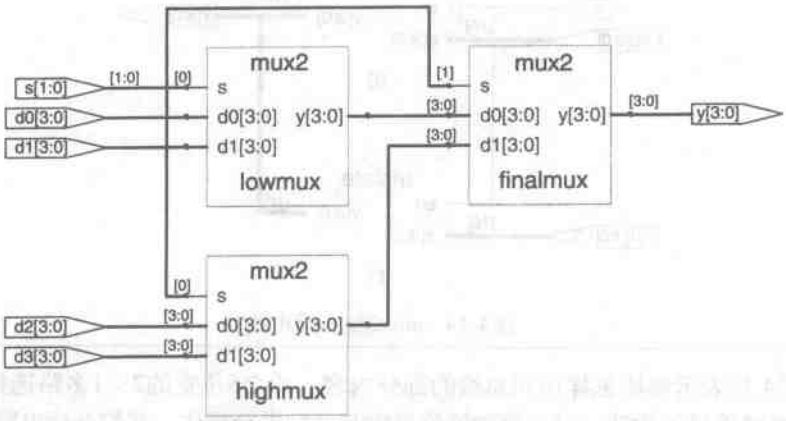


图4-13 mux4综合后电路

HDL例子4.16使用结构建模基于一对三态缓冲器建立一个2：1多路选择器。

HDL例子4.16 2：1多路选择器的结构建模

Verilog

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

在Verilog中，允许如~s这种表达式用于实例的端口列表中。随意和复杂的表达式虽然合法，但是并不提倡，因为这会降低代码的可读性。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
```

```
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

在VHDL中，不允许如not s这种表达式出现在实例的端口分配图中。因此，sbar信号应该分开定义。

2：1多路选择器综合后电路如图4-14所示。

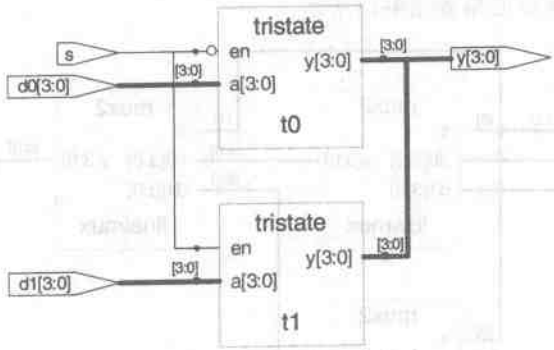


图4-14 mux2综合后电路

HDL例子4.17表示模块怎样访问总线的部分内容。一个8位宽的2：1多路选择器用两个已定义的4位2：1多路选择器构造，对高位和低位半字节分别进行操作。其综合后电路如图4-15所示。

一般来说，复杂的系统都是分层定义的。以实例化主要组件的方式结构化描述整个系统。而每一个组件也由更小的块构成，然后进一步分解，直到足够简单可以描述行为。避免（至少是尽量减少）在一个单独模块中混合使用结构和行为描述方式是一种好的程序设计风格。

HDL例子4.17 访问部分总线

Verilog

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in   STD_LOGIC_VECTOR(3
                                 downto 0);
         s:      in   STD_LOGIC;
         y:      out  STD_LOGIC_VECTOR(3 downto 0));
  end component;
```

```
begin
  lsbmux:  mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbhmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

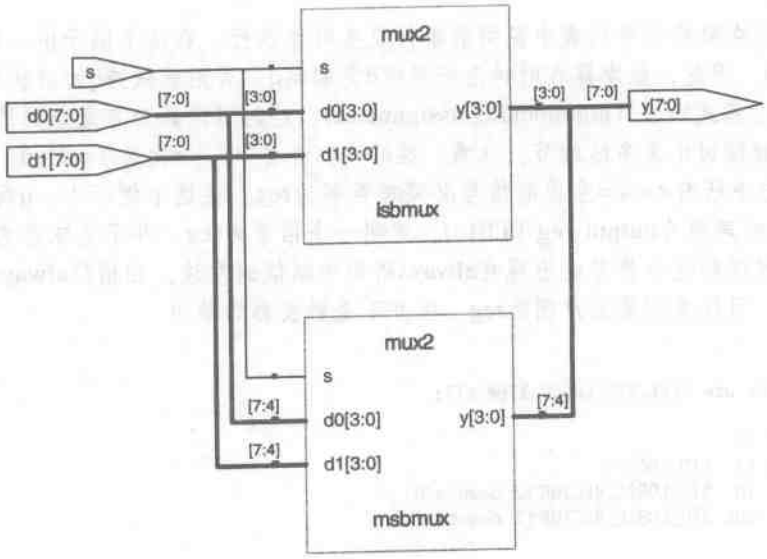


图4-15 mux2_8综合后电路

4.4 时序逻辑

HDL 综合器能辨别出特定的风格，然后把它们转换成特定的时序电路。其他风格的编码可能在模拟时正确，但是在综合成电路时会出现明显或不明显的错误。本节介绍寄存器和锁存器的正确描述方法。

4.4.1 寄存器

现代主要的商业系统都是由上升沿触发的D触发器寄存器组成。HDL例子4.18给出了这些触发器的书写风格。

Verilog的always语句和VHDL的process语句中，信号保持它们的旧值直到一个敏感信号列表中的事件发生，此时将引起它们的值改变。因此，具有合适敏感信号列表的代码可以用于描述有记忆能力的时序电路。例如，在触发器的敏感信号列表中只有clk。这说明q在下一个clk上升沿来临前都会保持旧值，即使d在中途发生过改变。

相反地，Verilog连续赋值语句（assign）和VHDL并发赋值语句（<=）在任何一个在右边的输入发生改变时，就会重新计算值。因此，这样的代码用于描述组合电路。

HDL例子4.18 寄存器

Verilog

```
module flop(input            clk,
            input      [3:0] d,
```

```
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;
endmodule
```

Verilog的always语句以这样的形式书写

```
always @ (sensitivity list)
  statement;
```

这个语句只在敏感信号列表中说明的事件发生时才执行。在这个例子中，语句是q<=d(读作“q gets d”)。因此，触发器在时钟上升沿把d复制到q，否则就保持q的旧状态。

<=称为非阻塞式赋值（nonblocking assignment)。这时可以认为它是普通的“=”。我们将在4.5.4小节中继续讨论更多的细节。注意：在always语句中，<=代替了assign。

always语句中所有<=或=左边的信号必须被声明为reg。在这个例子中，q既是输出也是一个reg，所以它被声明为output reg [3:0] q。声明一个信号为reg，并不意味着它就是一个寄存器的输出。它只说明这个信号会出现在always语句中赋值的左边。在稍后always语句描述组合逻辑的例子中，可以看到输出声明为reg，但并不是触发器的输出。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

VHDL的process语句以这样的形式书写：

```
process(sensitivity list) begin
  statement;
end process;
```

这个语句只当在敏感信号列表中的事件发生时才执行。在这个例子中，if语句在clk改变的时候执行，用clk'event表示。如果是一个上升沿改变（在事件发生后，clk='1'），那么q<=d(读作“q gets d”)。因此，触发器在时钟上升沿把d复制到q，否则就记录q的旧状态。

另一种触发器VHDL句法如下：

```
process(clk) begin
  if RISING_EDGE(clk) then
    q <= d;
  end if;
end process;
```

这里RISING_EDGE (clk)等同于clk'event和clk=1。图4-16给出了触发器综合后电路。

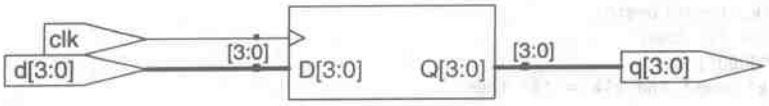


图4-16 触发器综合后电路

4.4.2 可复位寄存器

当开始模拟或者电路首次接通电源时，触发器或者寄存器的输出是未知的。在Verilog和VHDL中分别用x和u表示。一般来说，应该使用可复位寄存器，因为在上电时可以把系统置于已知的状态。复位可以是同步也可以是异步。异步复位马上就生效，而同步复位在下一个时钟上沿时才复位输出。HDL例子4.19说明了同步复位和异步复位触发器的风格。注意：在原理图中难以分辨同步和异步复位。Synplify Pro生成的原理图把异步复位置于触发器的底部，同步复位则置于左边，如图4-17所示。

HDL例子4.19 可复位寄存器

Verilog

```
module flopr (input             clk,
              input             reset,
              input      [3:0] d,
              output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr (input             clk,
              input             reset,
              input      [3:0] d,
              output reg [3:0] q);

  // synchronous reset
  always @ (posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

在always语句的敏感信号列表中，多个信号以逗号或者or分隔。注意到posedge reset在异步复位触发器的敏感信号列表中，但不在同步复位触发器中。因此，异步复位触发器会马上响应reset的上升沿，但是同步复位触发器只在时钟上沿时才响应reset。

因为两个模块的名字都是flopr，所以只能在设计中使用其中一个。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port (clk,
        reset: in  STD_LOGIC;
        d:     in  STD_LOGIC_VECTOR (3 downto 0);
        q:     out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture asynchronous of flopr is
begin
```

```
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk' event and clk = '1' then
      q <= d;
    end if;
  end process;
end;

architecture synchronous of flopr is
begin
  process (clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

在process语句的敏感信号列表中，多个信号以逗号分隔。注意到reset在异步复位触发器的敏感信号列表中，但不在同步复位触发器中。因此，异步复位触发器会马上响应reset的上升沿，但是同步复位触发器只在时钟上升沿时响应reset。

回想一下，触发器状态在VHDL模拟开始时被初始化为u。

如之前提及的，architecture的名字（此例中的asynchronous 或者synchronous）会被VHDL工具忽略，但是对人们阅读代码有帮助。因为两个结构都描述了实体flopr，所以只能在设计中使用其中一个。

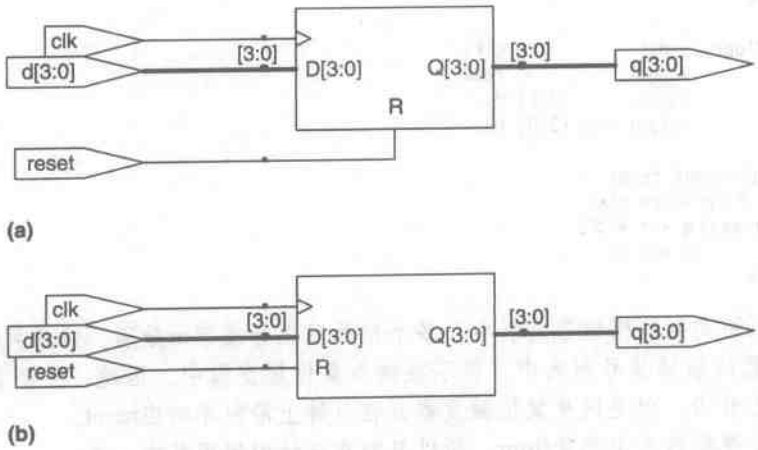


图4-17 flopr综合后电路：(a) 异步复位；(b) 同步复位

4.4.3 带使能端的寄存器

带使能端的寄存器只在使能有效时才会响应时钟。HDL例子4.20表示了一个异步复位使能寄存器，原理图如图4-18所示。当reset和en都是FALSE时，它就会保持旧值。

HDL例子4.20 可复位和带使能端的寄存器

Verilog

```
module flopenr (input        clk,
```

```
              input            reset,
              input            en,
              input      [3:0] d,
              output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port (clk,
        reset,
        en: in  STD_LOGIC;
        d:  in  STD_LOGIC_VECTOR (3 downto 0);
        q:  out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk'event and clk = '1' then
      if en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end;
```

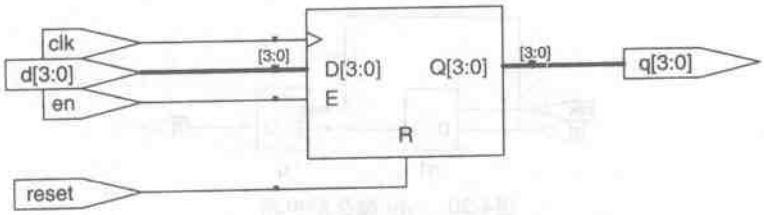


图4-18 flopenr综合后电路

4.4.4 多寄存器

一个单独的always/process 语句可以用于描述多个硬件。例如，3.5.5小节的同步器由两个背靠背连接的触发器组成，如图4-19所示。HDL例子4.21描述了同步器。在clk的上升沿，d被复制到n1。同时，n1被复制到q。

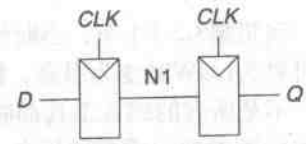


图4-19 同步器电路

HDL例子4.21 同步器（同步器综合后电路如图4-20所示）

Verilog

```
module sync (input      clk,
             input      d,
             output reg q);
```

```
  reg n1;

  always @ (posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

n1必须声明为reg类型。因为它是在always语句中<=左侧使用的内部变量。而且还注意到，begin/end结构是必要的，因为有多条声明语句出现在always语句里。这与C或Java里面的{ }相似。begin/end在flopr例子中并不是必须的，因为if/else是一个单独的语句。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

n1必须声明为signal，因为这是一个在模块内部使用的内部信号。

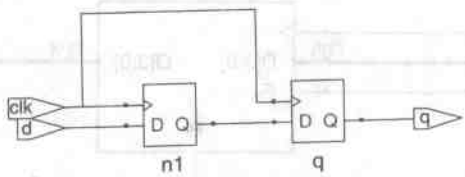


图4-20 sync综合后电路

4.4.5 锁存器

回想起3.2.2小节，当时钟为HIGH时，D锁存器通透，允许数据从输入流向输出。锁存器在时钟为LOW时变为阻塞，保持原有的状态。HDL例子4.22表示了D锁存器的风格。

不是所有的综合工具都能很好地支持D锁存器。除非你知道你的工具支持锁存器，或者你有很强烈的理由需要使用它，不然的话，最好使用边沿触发器代替。还要注意HDL代码不能隐含任何非预期的锁存器，如果不小心的话，这种器件很容易出现。很多综合工具在生成一个锁存器时会发出警告。如果你并不希望它存在，就需要在HDL中寻找这个漏洞。

HDL例子4.22 D锁存器（锁存器综合后电路如图4-21所示）

Verilog

```
module latch(input          clk,
             input      [3:0] d,
```

```
            output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;
endmodule
```

敏感信号列表包括clk和d，所以always语句当clk和d改变时就计算值。如果clk为HIGH，那么d值就会流向q。

q必须声明为reg，因为它出现在always语句中<=的左侧。这并不一定意味着q就是寄存器的输出。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

敏感信号列表包括clk和d，所以process语句当clk和d有改变时就计算值。如果clk为HIGH，那么d值就会流向q。

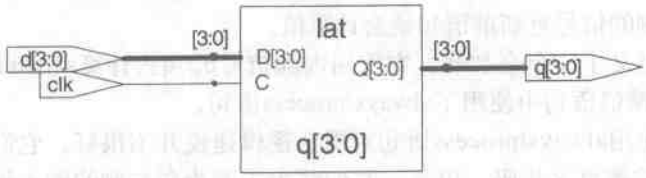


图4-21 latch综合后电路

4.5 更多组合逻辑

4.2节中使用赋值语句从行为上去描述组合逻辑。Verilog的always语句和VHDL的process语句可以用于描述时序电路，因为没有产生新状态时，将保持状态。然而，always/process语句也可以用于组合逻辑的行为级描述，其中敏感信号列表中包含了所有的输入以对其改变做出响应，同时正文对每一种可能的输入组合都定义了对应的输出。HDL例子4.23使用always/process语句描述了一组4个反向器（参见图4-3的综合电路）。

HDL例子4.23 使用always/process语句的反相器

Verilog

```
module inv(input      [3:0] a,
           output reg [3:0] y);

  always @ (*)
    y = ~a;
endmodule
```

当always语句中<=右侧的信号发生改变时，always @（*）就重新计算在always声明语句中的语句。因此，@（*）是描述组合逻辑的安全方式。在这个本例中，@（a）也将被满足。

"="在always语句中称为阻塞式赋值（blocking assignment），与之相对的是"<="为非阻塞式赋值（non-blocking assignmeng）。在Verilog中，在组合逻辑中适合使用阻塞式赋值，而在时序逻辑中需要使用非阻塞式赋值。这将在4.5.4小节中进一步讨论。

注意到y必须声明为reg，这是因为它在always语句中出现在<=或者=的左侧。然而，y是组合电路的一个输出而不是寄存器。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(a) begin
    y <= not a;
  end process;
end;
```

在VHDL语句中，begin和end process语句都是需要的，尽管process中只有一个赋值语句。

HDL在always/process语句中支持阻塞式和非阻塞式赋值。阻塞式赋值语句序列会以其在代码中出现的顺序计算值，这就如一些标准的编程语言一样。一组非阻塞式赋值语句则是并行地计算值。在左侧的信号更新前语句就会计算值。

HDL例子4.24定义了一个全加器。它使用内部信号p，q去计算s和cout。它产生的电路与图4-8一致，只是在赋值语句中使用了always/process语句。

这两个例子中使用always/process语句对组合逻辑建模并不很好，它们比4.2.1小节用赋值语句实现同等功能需要更多代码。但是，它们减少了因为在左侧的输入没有包括在敏感信号列表中造成的易犯错误。然而，case和if语句对于对更复杂的组合电路建模更加方便。case和if语句必须出现在always/process语句中，我们会在下一小节中继续讨论。

Verilog

在Verilog的always语句中，'='表示阻塞式赋值，'<='表示非阻塞式赋值（也被称作并行赋值）。

不要把连续赋值的assign语句与这里混淆。Assign语句应该在always语句外部使用，而且是并行计算值。

VHDL

在VHDL的process语句中，':='表示阻塞式赋值，'<='表示非阻塞式赋值（也被称作并行赋值）。这里是第一次介绍':='的章节。

非阻塞式赋值用于产生输出和signal。阻塞式赋值则用于产生在process语句中声明的变量（参阅HDL例子4.24）。

<=也可以在process外面出现，同时也是并行计算值。

HDL例子4.24 使用always/process语句的全加器

Verilog

```
module fulladder (input      a, b, cin,
                  output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p = a ^ b;                  // blocking
      g = a & b;                  // blocking

      s = p ^ cin;                // blocking
      cout = g | (p & cin);       // blocking
    end
endmodule
```

在这里，@（a，b，cin）与@（*）等价。不过@（*）更好，因为漏写敏感信号列表中的信号是一个常见的错误。

为了在4.5.4小节讨论的原因，在这里最好使用阻塞式赋值实现组合逻辑。此例中首先计算p和g，然后计算s，最后计算cout。

因为p和g出现在always中赋值语句的左侧，所以它们必须声明为reg。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port (a, b, cin:  in  STD_LOGIC;
        s, cout:    out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process (a, b, cin)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Process中的敏感信号列表必须包含a，b，c和cin，因为组合逻辑对所有的输入都要做出响应。基于4.5.4小节讨论的原因，这里最好使用阻塞赋值方式计算组合逻辑中的内部信号。这个例子中对p和g使用阻塞式赋值，因此会在计算依赖于它们值的s和cout之前首先计算它们。

因为p和g出现在process语句中阻塞式赋值(:=)的左侧，所以它们必须声明为variable而不是signal。变量声明出现在使用它的过程的begin之前。

4.5.1 选择语句

使用always/process语句实现组合逻辑的一个较好例子是利用case语句实现七段数码管译码器。case语句必须出现在always/process语句的内部。

正如你可能留意到的，例2.10中大模块组合逻辑的设计过程冗繁，且容易出错。HDL做出了巨大改进，它允许用户在更高的抽象层说明功能，然后自动把功能综合成门电路。HDL

例子4.25基于真值表，使用case语句描述七段数码管显示译码器（参阅描述七段数码管译码器的例子例2.10）。case语句基于输入值做出不同的行为。一个case语句中，当所有可能的输入组合都被定义时，就表示为组合逻辑。否则，它就表示时序逻辑，因为在未定义情况下输出会保持旧值。

Synplify pro把七段数码管译码器综合成一个针对16种不同输入而产生7位输出的只读存储器（ROM）。ROM将在5.5.6小节进一步讨论。

如果case语句中漏写了default或者other子句，那么译码器将在输入数据为10～15时，记录它之前的输出。这对于硬件来说是一个奇怪的行为。

普通的译码器一般也用case语句表示。HDL例子4.26描述了一个3:8译码器。

HDL例子4.25 七段数码管显示译码器

Verilog

```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);

  always @ (*)
    case (data)
      //                abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

case语句检查data的值。当data为0，语句就会执行冒号后的操作，设置segments为1111110。相似地，case语句检查其他data的值，最高是9(这里使用了默认的基10)。

default语句是一种非常方便的方法来定义没有明确列举情况下的输出，以确保产生组合逻辑。

在Verilog中，case语句必须出现在always语句中。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(data) begin
    case data is
      --                              abcdefg
      when X"0" => segments <= "1111110";
      when X"1" => segments <= "0110000";
      when X"2" => segments <= "1101101";
      when X"3" => segments <= "1111001";
      when X"4" => segments <= "0110011";
```

```
    when X"5" => segments <= "1011011";
    when X"6" => segments <= "1011111";
    when X"7" => segments <= "1110000";
    when X"8" => segments <= "1111111";
    when X"9" => segments <= "1111011";
    when others => segments <= "0000000";
  end case;
 end process;
end;
```

case语句检查data的值。当data为0，语句就会执行'=>'后的操作，设置segments为1111110. case语句检查其他data的值，最高是9（注意X用于表示十六进制数）。Others子句可以方便地定义其他没有明确列出情况下的输出，这样可以保证电路为组合逻辑。

和Verilog不同，VHDL支持选择信号赋值语句（参阅HDL例子4.6）。它与case语句很像，但可以出现在进程（process）的外部。因此，这里也不应该使用进程来描述组合逻辑了。

七段数码管显示译码器的电路如图4-22所示。

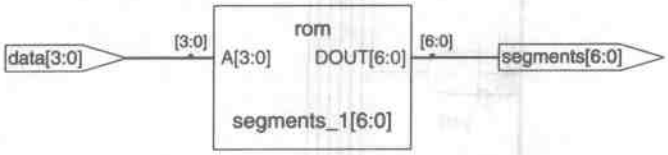


图4-22 sevenseg综合后电路

HDL例子4.26 3：8译码器

Verilog

```
module decoder3_8(input      [2:0] a,
                  output reg [7:0] y);
 always @ (*)
   case (a)
     3'b000: y = 8'b00000001;
     3'b001: y = 8'b00000010;
     3'b010: y = 8'b00000100;
     3'b011: y = 8'b00001000;
     3'b100: y = 8'b00010000;
     3'b101: y = 8'b00100000;
     3'b110: y = 8'b01000000;
     3'b111: y = 8'b10000000;
   endcase
endmodule
```

因为所有的情况都已经说明了，所以不需要default语句。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(a) begin
    case a is
      when "000" => y <= "00000001";
      when "001" => y <= "00000010";
      when "010" => y <= "00000100";
      when "011" => y <= "00001000";
```

```
      when "100" => y <= "00010000";
      when "101" => y <= "00100000";
      when "110" => y <= "01000000";
      when "111" => y <= "10000000";
    end case;
  end process;
end;
```

因为所有的情况都已经说明了，所以不需要other语句。3：8译码器的电路如图4-23所示。

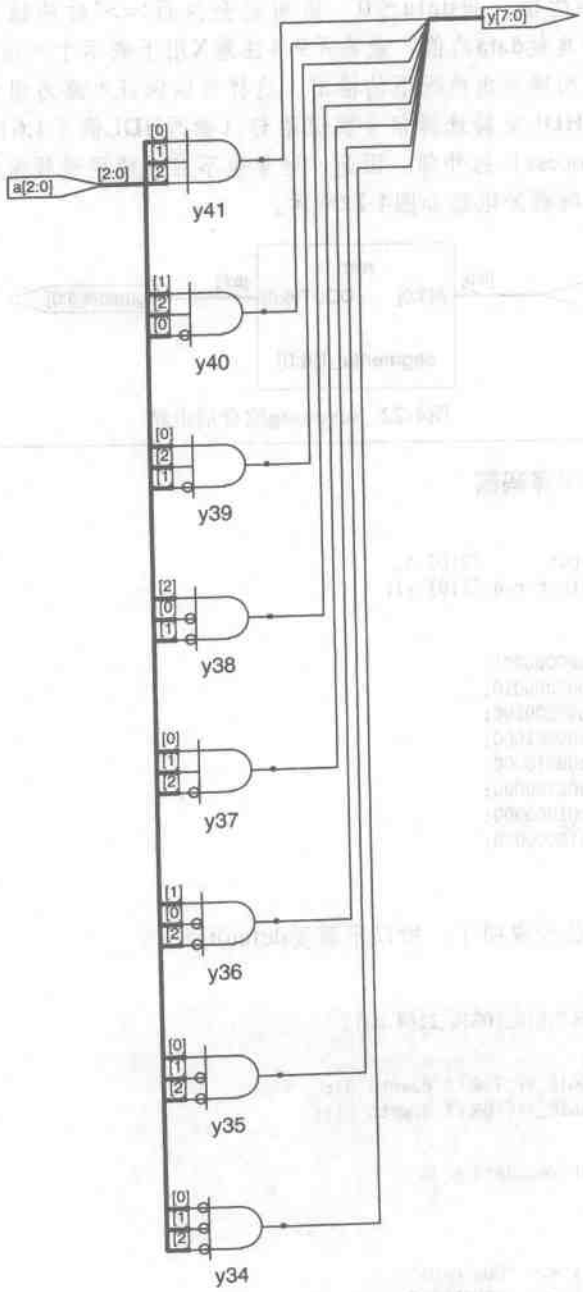


图4-23 decoder3_8综合后电路

4.5.2 if语句

always/process中可以包含if语句。if语句后面还可以出现else语句。如果所有的输入组合都被处理，那么这个语句就表示组合逻辑；否则，就产生时序逻辑（如4.4.5小节的锁存器）。

HDL例子4.27使用if语句描述2.4小节中介绍的优先级电路。一个n输入的优先电路在对应最高优先级输入为true的位输出true。

HDL例子4.27 优先级电路（图4-24给出了优先级综合后电路）

Verilog

```
module priority(input      [3:0] a,
                output reg [3:0] y);

  always @ (*)
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

在Verilog中，if语句必须出现在always语句中。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priority is
begin
  process(a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                  y <= "0000";
    end if;
  end process;
end;
```

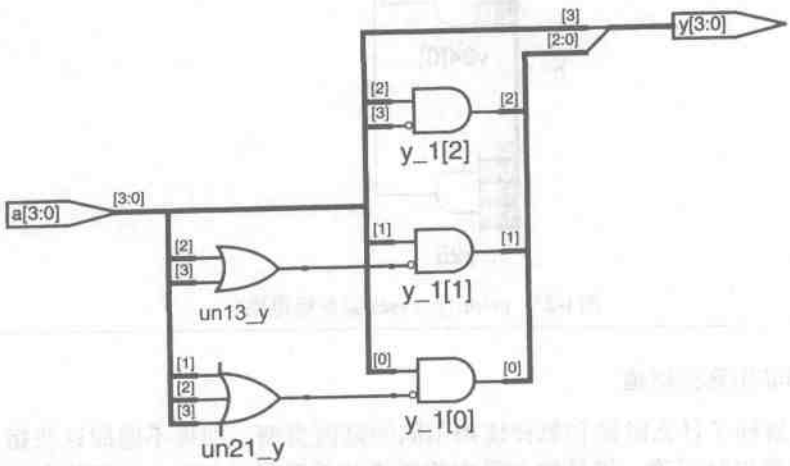


图4-24 priority综合后电路

与Verilog不同，VHDL支持条件信号赋值语句（参看HDL例子4.6）。它与if语句很像，但是出现在进程（process）外部。因此，这里也不应该使用进程描述组合逻辑了。

4.5.3 Verilog的casez语句*

（如果读者使用VHDL，可以跳过这一小节。）Verilog还提供了casez语句去描述带有无关值（在casez语句中以？表示）的真值表。HDL例子4.28表示了如何用casez语句描述一个优先电路。

Synplify Pro为这个模块综合生成的电路与图4-24的电路稍微有点不同，如图4-25所示。然而，它们是逻辑等价的。

HDL例子4.28 使用casez的优先级电路

```
module priority_casez(input      [3:0] a,
                      output reg [3:0] y);

  always @ (*)
    casez (a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
  endcase
endmodule
```

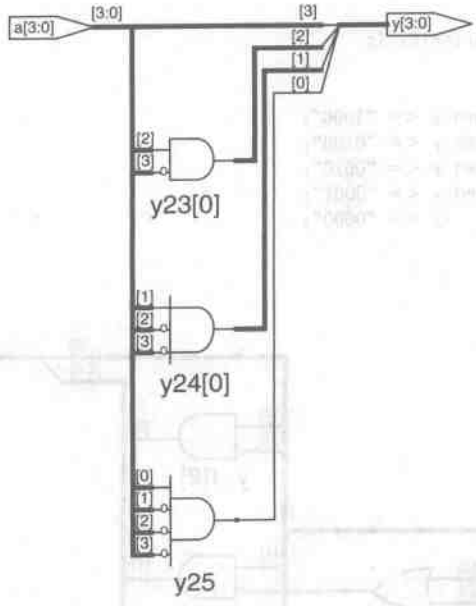


图4-25 priority_casez综合后电路

4.5.4 阻塞式和非阻塞式赋值

下面的内容解释了什么时候和怎样使用不同的赋值类型。如果不遵照这些指引，编写的代码就很可能在模拟时正确，但是综合得到的硬件却不正确。在这个小节的余下可选读部分

解释了这些指导背后的规则。

阻塞式和非阻塞式赋值指导

Verilog

1) 使用always@(posedge clk)和非阻塞式赋值描述同步时序逻辑。

```
always @ (posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2) 使用连续赋值去描述简单组合逻辑

```
assign y = s ? d1 : d0;
```

3) 使用always @ (*) 和阻塞式赋值描述复杂的组合逻辑将很有帮助。

```
always @ (*)
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4) 不要在多个always语句或者连续赋值语句中对同一个信号赋值。

VHDL

1) 使用process (clk) 和非阻塞式赋值描述同步时序逻辑。

```
process(clk) begin
  if clk'event and clk = '1' then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2) 使用process语句外的并行赋值描述简单组合逻辑。

```
y <= d0 when s = '0' else d1;
```

3) 使用process (in1, in2, ...) 描述复杂的组合逻辑将会有帮助。使用阻塞式赋值对内部变量操作。

```
process (a, b, cin)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4) 不要在多个process语句或者并行赋值语句中对同一个信号赋值。

组合逻辑*

HDL例子4.24中的全加器使用阻塞赋值方式可以正确描述。这个小节将探讨它如何操作，以及如果使用非阻塞式赋值会有何不同。

假设a，b，cin都初始化为0。因此，p，g，s和cout也是0。在某一时刻，a改变为1，触发了always/process语句。四个阻塞式赋值按顺序计算值（在VHDL代码里，s和cout并行赋值）。

注意到p和g是在阻塞式赋值中，所以它们在s和cout计算之前得到了新值。这很重要，因为我们希望s和cout使用p和g的新值进行计算。

$p \leftarrow 1 \oplus 0 = 1$
$g \leftarrow 1 \cdot 0 = 0$
$s \leftarrow 1 \oplus 0 = 1$
$cout \leftarrow 0 + 1 \cdot 0 = 0$

对应地，HDL例子4.29中说明了非阻塞式赋值的使用。

HDL例子4.29 使用非阻塞赋值的全加器

Verilog

```
// nonblocking assignments (not recommended)
module fulladder (input      a, b, cin,
                  output reg s, cout);

  reg p, g;

  always @ (*)
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

因为p和g出现在always语句的赋值的左侧，所以必须声明为reg。

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process(a, b, cin, p, g) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

因为p和g出现在process语句中非阻塞式赋值的左侧，所以它们必须被声明为signal而不是variable。信号声明出现在architecture里begin之前，而不是process中。

现在考虑到相同的情形，a从0上沿为1，这是b和cin都为0。四个非阻塞式赋值并行地计算值。

$p \leftarrow 1 \oplus 0 = 1$ $g \leftarrow 1 \cdot 0 = 0$ $s \leftarrow 0 \oplus 0 = 0$ $cout \leftarrow 0 + 0 \cdot 0 = 0$

注意到s并行地与p一起计算，因此使用了p的旧值，而不是新值。所以，s保持0而不是1。然而，p从0改变为1。这个改变触发了always/process语句，从而第二次计算值，过程如下：

p←1⊕0 = 1 g←1 · 0 = 0 s←1⊕0 = 1 cout←0 + 1 · 0 = 0

这一次，p变为1，所以s正确地改变为1。非阻塞式赋值最后改变为正确值，但是always/process语句必须计算两次。

使用非阻塞式赋值描述组合逻辑的另外一个缺点是，如果你忘记把内部变量包括在敏感信号列表中，HDL会产生错误的结果。

Verilog

如果HDL例子4.29中always语句的敏感信号列表写作always @ (a, b, cin)，而不是always@ (*)，那么这个语句就不会在p或者g改变时重新计算值。在之前的例子中，s会错误保持为0，而不是1。

VHDL

如果HDL例子4.29中process语句的敏感信号列表写作process (a, b, cin)，而不是process (a, b, cin, p, g)，那么这个语句不会在p或g改变时重新计算值。在之前的例子中，s会错误地保持为0，而不是1。

更糟的是，即使错误的敏感信号列表引起了模拟错误，一些综合工具也会综合得到正确的硬件。这就导致模拟结果和硬件实际操作不匹配。

*时序逻辑**

HDL例子4.21中可以使用非阻塞式赋值正确地描述同步器。在时钟的上沿，d被复制到n1。同时n1被复制到q，所以代码准确地描述了两个寄存器。例如，假设初始化d = 0，n1 = 1，q = 0。在时钟上沿沿，下面两个赋值同时发生，所以时钟脉冲上升沿后，n1 = 0，q = 1。

n1←d = 0 q← n1 = 1

HDL例子4.30尝试去用阻塞式语句去描述同一个模块。在上升沿clk，d被复制到n1，之后这个n1的新值被复制到q，导致n1和q中出现不正确的d。赋值在时钟上升沿后依次进行，q = n1 = 0。

n1←d = 0

q←n1 = 0

HDL例子4.30 使用阻塞赋值的错误的同步器

Verilog

```
// Bad implementation using blocking assignments

module syncbad (input      clk,
                input      d,
                output reg q);

  reg n1;

  always @ (posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation using blocking assignment
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if clk'event and clk = '1' then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

错误的同步器综合后电路如图4-26所示。

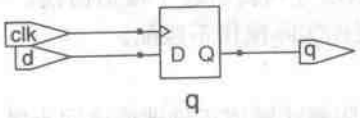


图4-26 syncbad综合后电路

因为n1对于外界来说是透明的，也不对q的行为有影响，所以同步器完全把它优化掉了，如图4-26所示。

在描述时序逻辑时，在always语句中必须使用非阻塞式赋值。如果读者足够聪明，也可以利用一些方法（例如调换赋值语句的顺序），也可以使阻塞赋值正确工作，但是阻塞赋值并没有优势，反而带来了未知行为的风险。有一些时序电路是无法使用阻塞式赋值描述的。

4.6 有限状态机

有限状态机（FSM）由状态寄存器和两个组合逻辑块组成，其中组合逻辑块用于计算当前状态和输入下的输出和下一状态，如图3-22所示。状态机的HDL描述对应划分成三个部分以分别描述状态寄存器、下一状态逻辑和输出逻辑。

HDL例子4.31 3分频计数器的有限状态机

Verilog

```
module divideby3FSM(input  clk,
                    input  reset,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  // state register
```

```
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

parameter语句用于定义模块内的常量。通过parameter对状态命名并不是必须的，但是这样做会使修改状态编码更容易，而且代码可读性更强。

注意case语句用于定义状态转换表的方法。因为下一状态逻辑必须是组合逻辑，所以dafault不能缺少，尽管这里2'b11状态不会出现。

当状态为s0时，输出y为1。如果a等于b，相等比较式a= =b得到值1，否则为0。不相等比较式a!=b则相反，在a不等于b时得到值1。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

这一例子中定义了一个新的枚举数据类型statetype，有3个可能值s0，s1，s2。state和nextstate都是statetype类型的信号。使用枚举代替状态编码时，VHDL综合器可以自由地搜索各种不同的状态编码，以选择一个最好编码。

当状态为s0的时候，输出y为1。不相等比较式使用/=表示。当把比较改为state/=0时，除了状态为s0外，其他状态的输出都为1。

3分频计数器的有限状态机电路如图4-27所示。

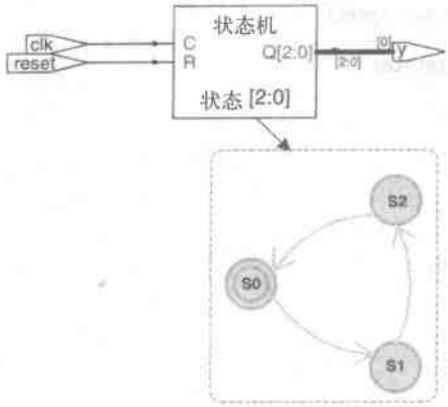


图4-27 divideby3fsm综合后电路

HDL例子4.31描述了3.4.2小节中3分频计数器的有限状态机。它提供异步复位以初始化有限状态机。状态寄存器使用了触发器的描述风格。下一状态和输出逻辑块则是组合逻辑。

Synplify Pro Synthesis工具仅产生状态机的模块图和状态转换图；而且没有表示出逻辑门，弧线和状态上输入输出。因此，要注意是否已经在HDL代码中正确地说明描述了有限状态机。图4-27的3分频计数器有限状态机状态转换图与图3-28(b) 相似。双圆圈表示s0是复位状态。3分频计数器有限状态机的门级实现如3.4.2小节所示。

如果因为某些原因，我们需要在状态s0，s1中输出为HIGH，那么输出逻辑应该按如下修改。

Verilog

```
// Output Logic
assign y = (state == S0  |  state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

以下两个例子描述了3.4.3小节中的模式识别器有限状态机。代码展示了如何使用case和if语句，根据输入和当前状态产生下一状态和输出的逻辑。这里给出了Moore型状态机和Mealy型状态机的模块。在Moore型状态机（HDL例子4.32）中，输出只与当前状态有关，而在Mealy状态机（HDL例子4.33）中，输出逻辑与当前状态和输入都有关。

HDL例子4.32 模式识别器的Moore型有限状态机

Verilog

```
module patternMoore(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [2:0] state, nextstate;

  parameter S0 = 3'b000;
  parameter S1 = 3'b001;
  parameter S2 = 3'b010;
  parameter S3 = 3'b011;
  parameter S4 = 3'b100;
```

```
  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S4;
          else   nextstate = S0;
      S4: if (a) nextstate = S2;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S4);
endmodule
```

注意：如何使用非阻塞式赋值（<=）在状态寄存器中描述时序逻辑。同时，如何使用阻塞式赋值（=）在下一状态逻辑中描述组合逻辑。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if a = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if a = '1' then
                   nextstate <= S2;
                 else nextstate <= S3;
                 end if;
```

```
      when S3 => if a = '1' then
                      nextstate <= S4;
                 else nextstate <= S0;
                 end if;
      when S4 => if a = '1' then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when state = S4 else '0';
end;
```

模式识别器的Moore型状态机的电路如图4-28所示。

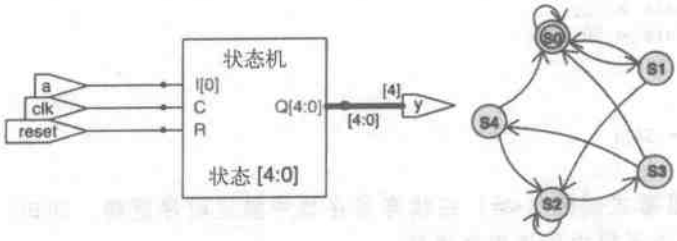


图4-28 patternmoore综合后电路

HDL例子4.33 模式识别器的Mealy型有限状态机

Verilog

```
module patternMealy(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S1;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase
```

```
  // output logic
  assign y = (a & state == S3);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is(S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
         end if;
      when S1 => if a = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
         end if;
      when S2 => if a = '1' then
                   nextstate <= S2;
                 else nextstate <= S3;
         end if;
      when S3 => if a = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
         end if;
       when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S3) else '0';
end;
```

模式识别器的Mealy型状态机的电路如图4-29所示。

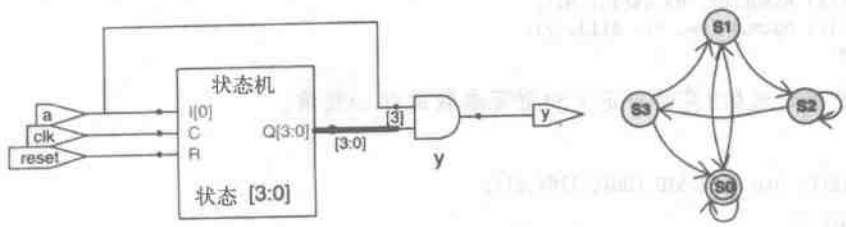


图4-29 patternmealy综合后电路

4.7 参数化模块*

目前为止，所有模块的输入和输出宽度都是固定的。例如，我们必须分别针对4位和8位宽2：1多路器转换器定义不同的模块。在使用参数化模块时，HDL允许位宽可变。

HDL例子4.34描述了一个宽（width）默认为8的参数化2：1多路转换器，之后使用它分别生成一个8位和一个12位的4：1多路转换器。

HDL例子4.35描述了一个更好的参数化模块——译码器。使用case语句描述大型n：2^n译码器很麻烦。但使用参数化代码来设置合适的输出位为1却很简单。特别需要注意的是，译码器使用阻塞式赋值首先设置所有的位为0，然后改变合适的位为1。

HDL例子4.34 参数化的n位多路选择器（12位4：1多路转换器综合后电路如图4-30所示）

Verilog

```
module mux2
  #(parameter width = 8)
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

Verilog允许在定义输入输出信号之前使用#(parameter...)语句定义参数。parameter语句包括参数width的默认值（8）。输入和输出信号的位数依赖于这个参数。

```
module mux4_8(input  [7:0] d0, d1, d2, d3,
              input  [1:0] s,
              output [7:0] y);

  wire [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[1], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

8位4：1多路器转换器使用了3个默认width的2：1多路转换器的实例。

相比之下，12位的4：1多路转换器mux4_12就需要在实例名前使用#（）重写默认宽度，如下所示。

```
module mux4_12(input  [11:0] d0, d1, d2, d3,
               input  [1:0]  s,
               output [11:0] y);

  wire [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[1], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

不要把表示延迟的#符号与定义和重写参数的#(...)混淆。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
```

```
  port (d0,
        d1: in  STD_LOGIC_VECTOR (width-1 downto 0);
        s:  in  STD_LOGIC;
        y:  out STD_LOGIC_VECTOR (width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

generic语句包括width的默认值（8）。这个值为整数类型。

```
entity mux4_8 is
  port (d0, d1, d2,
        d3: in  STD_LOGIC_VECTOR (7 downto 0);
        s:  in  STD_LOGIC_VECTOR (1 downto 0);
        y:  out STD_LOGIC_VECTOR (7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic (width: integer);
  port (d0,
        d1: in  STD_LOGIC_VECTOR (width-1 downto 0) ;
        s:  in  STD_LOGIC;
        y:  out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

8位4：1多路转换器mux4_8使用了3个默认width的2：1多路转换器的实例。

相比之下，12位4：1多路转换器mux4_12就需要用generic map覆盖默认的width，如下所示。

```
lowmux: mux2 generic map (12)
             port map (d0, d1, s(0), low);
himux:  mux2 generic map (12)
             port map (d2, d3, s(0), hi);
outmux: mux2 generic map (12)
             port map (low, hi, s(1), y);
```

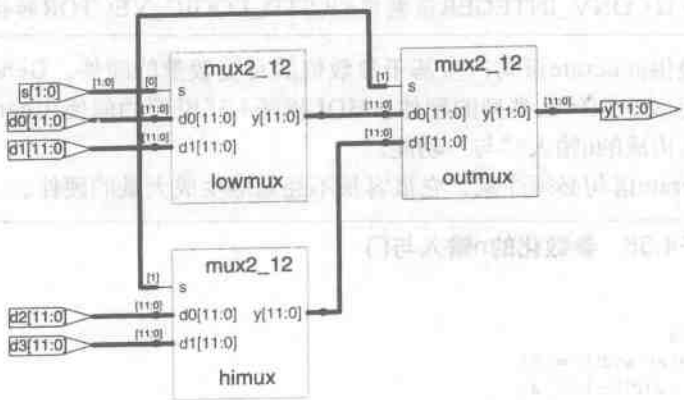


图4-30 mux4_12综合后电路

HDL例子4.35 参数化的n：2^n译码器

Verilog

```
module decoder #(parameter   N = 3)
                (input       [N-1:0]    a,
                 output reg  [2**N-1:0] y);

  always @ (*)
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

2**n代表2^n

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity decoder is
  generic (N: integer := 3);
  port (a: in  STD_LOGIC_VECTOR (N-1 downto 0);
        y: out STD_LOGIC_VECTOR (2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process (a)
    variable tmp: STD_LOGIC_VECTOR (2**N-1 downto 0);
  begin
    tmp := CONV_STD_LOGIC_VECTOR(0, 2**N);
    tmp (CONV_INTEGER(a)) := '1';
    y <= tmp;
  end process;
end;
```

2**n代表2^n

CONV_STD_LOGIC_VECTOR(0, 2**n)产生了一个包含长度为2^n的STD_LOGIC_VECTOR，其初始化值为全0。这需要STD_LOGIC_ARITH库。这个函数对于其他参数化功能很有用，例如在可复位触发器需要能产生参数化设置位数的常数时。VHDL的位索引必须是整数类型，所以CONV_INTEGER函数把a从STD_LOGIC_VECTOR转换为整型。

HDL还提供generate语句产生基于参数值的可变数量的硬件。Generate支持for循环和if语句来决定产生多少和什么类型的硬件。HDL例子4.36说明如何使用generate语句产生一个由2输入与门串连构成的n输入“与”功能。

使用generate语句必须注意，它很容易不经意地生成大量的硬件。

HDL例子4.36 参数化的n输入与门

Verilog

```
module andN
  #(parameter width = 8)
    (input [width-1:0] a,
     output            y);

  genvar i;
  wire [width-1:1] x;
```

```
  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
    end
  endgenerate
  assign y = x[width-1];
endmodule
```

For语句循环通过i=1，2，…，width−1产生许多连续的与门。在generate for循环里面的begin后面必须有‘:’和一个任意的标识（在这个例子中是forloop）。

当然，书写为assign y=&a会更容易一些。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic (width: integer := 8);
  port (a: in  STD_LOGIC_VECTOR (width-1 downto 0);
        y: out STD_LOGIC);
end;

architecture synth of andN is
  signal i: integer;
  signal x: STD_LOGIC_VECTOR (width-1 downto 1);
begin
  AllBits: for i in 1 to width-1 generate
    LowBit: if i = 1 generate
      A1: x(1) <= a(0) and a(1);
    end generate;
    OtherBits: if i /= 1 generate
      Ai: x(i) <= a(i) and x(i-1);
    end generate;
  end generate;
  y <= x(width-1);
end;
```

n输入与门综合后电路如图4-31所示。

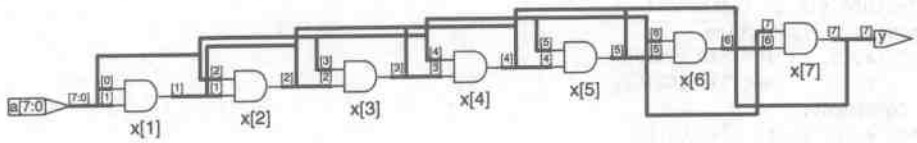


图4-31 andN综合后电路

4.8 测试程序

测试程序（testbench）是用于测试其他待测试模块（device under test, DUT）的硬件描述语言模块。此程序包含了向待测试模块提供输入的语句，以测试是否产生了理想的正确输出。输入和期待的输出模式称为测试向量（test vector）。

考虑测试4.1.1小节中计算$y = \overline{abc} + a\overline{bc} + \bar{a}bc$的sillyfunction模块。这是一个简单的模块，所以我们可以提供所有8个可能的测试向量表示全部的测试。

HDL例子4.37说明了一个简单的测试程序。它实例化DUT，之后提供输入。阻塞式赋值和延迟用于提供合适的输入顺序。使用者必须检查模拟结果以验证是否产生正确输出。测试程序也像其他的HDL模块那样被模拟，然而它们不能被综合。

HDL例子4.37 testbench

Verilog

```
module testbench1();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
    initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

initial语句在模拟开始时执行体内的语句。在本例中，它首先提供输入000，然后等待10个单位时间。随后提供001，之后等待10个单位时间，并按照此方式执行，以提供了所有8个可能的输入。initial语句只能用在测试程序的模拟，而不能用于可综合为实际硬件的模块中。这是因为首次启动时硬件无法执行一系列特殊的步骤。

就像always语句中的信号，initial语句中的信号必须声明为reg。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
```

```
  end process;
end;
```

process语句最先提供输入000，之后等待10ns。然后提供001，等待10ns，并按此方式提供所有8个可能的输入。

最后，process将无限等待；否则，process会再次开始，重复地提供测试向量。

检查输出是否正确的过程比较枯燥，而且容易出错。当设计在脑海里还是很清晰时，判断输出结果是否正确还比较简单。如果做出了很小的修改，而且数周后需要重新测试，判断输出是否正确就变得比较麻烦了。一个更好的方法是编写具有自测功能的测试程序，如HDL例子4.38所示。

HDL例子4.38 能自测试的测试程序

Verilog

```
module testbench2();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1;               #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0;        #10;
    if (y !== 0) $display("010 failed.");
    c = 1;               #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1;               #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0;        #10;
    if (y !== 0) $display("110 failed.");
    c = 1;               #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

当提供每个输入测试向量后，测试模块将y和期望值进行检查。在Verilog中，可以在不包括x和z值的信号中间使用= =或者!=的对比式。测试程序分别使用= = =和！= =运算符判断相等或不等，因为这些操作符能对包含x和z的操作数正确操作。在当发生错误的时候，使用系统任务$display向模拟器终端打印信息。$display只在模拟时才有意义，在综合时无意义。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
```

```
        y:        out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

assert语句检查条件，并当条件不被满足时打印report子句中的信息。assert只在模拟时有意义，在综合时无意义。

在需要很大量测试向量的模块中，为每个测试向量编写代码依然是冗繁的工作。一个比较好的方法是把测试向量置于独立的文件中。测试程序简单地从文件中读取测试向量，向DUT输入测试向量，检查DUT输出值是否与输出向量一致，之后重复这个过程直到测试向量文件的结尾。

HDL例子4.39说明了这种测试程序。测试程序使用没有敏感信号列表的always /process语句产生了一个时钟，因此它会连续不断地重复运行。模拟开始时，从测试文件读取测试向量，之后提供两个周期的reset脉冲。example.tv是包含了二进制格式输入和期待输出的文本文件：

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL例子4.39 带测试向量的测试程序

Verilog

```
module testbench3();
  reg        clk, reset;
  reg        a, b, c, yexpected;
  wire       y;
```

```
  reg [31:0] vectornum, errors;
  reg [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} =
           testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if(~reset) begin // skip during reset
      if(y !== yexpected) begin

      $display("Error: inputs = %b", {a, b, c});
      $display(" outputs = %b (%b expected)",
               y, yexpected);
      errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if(testvectors[vectornum] === 4'bx) begin
      $display("%d tests completed with %d errors",
               vectornum, errors);
      $finish;
    end
  end
endmodule
```

$readmemb读取二进制文件内容到testvectors数组，与$readmemh很相似，只不过它读取十六进制数字。

下一代码块在下一时钟上沿之后等待一个时间单位（以防止时钟和数据同时改变造成的混乱），然后根据当前测试向量的4位内容设置3位输入和1位期待输出。下一代码块在时钟下降沿检查DUT的输出，以保证输入有时间通过DUT传递并产生输出y。测试程序对期待输出yexpected与生成的输出y作对比。如果它们并不一致，就会打印出一个错误。%b和%d表示分别以二进制或者十进制输出值。例如，$display("%b %b", y, yexpected); 表示以二进制打印出y和yexpected两个值。%h则以十进制输出值。

这个进程重复直到tesevector数组中没有更多的可用测试向量。$finish结束模拟。注意：虽然Verilog模块最多支持10,001个测试向量，但是它在执行8个文件中的测试向量之后就结束模拟了。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected:  STD_LOGIC;
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(3 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;

    begin
      -- read file of test vectors
      i := 0;
      FILE_OPEN(tv, "example.tv", READ_MODE);
      while not endfile(tv) loop
        readline(tv, L);
        for j in 0 to 3 loop
          read(L, ch);
          if(ch = '_') then read(L, ch);
          end if;
          if(ch = '0') then
            testvectors(i)(j) <= '0';
          else testvectors(i)(j) <= '1';
          end if;
        end loop;
        i := i + 1;
      end loop;

      vectornum := 0; errors := 0;
      reset <= '1'; wait for 27 ns; reset <= '0';
      wait;
    end process;

    -- apply test vectors on rising edge of clk
    process(clk) begin
      if(clk'event and clk = '1') then
```

```
      a <= testvectors(vectornum)(0) after 1 ns;
      b <= testvectors(vectornum)(1) after 1 ns;
      c <= testvectors(vectornum)(2) after 1 ns;
      yexpected <= testvectors(vectornum)(3)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process(clk) begin
    if(clk'event and clk = '0' and reset = '0') then
      assert y = yexpected
        report "Error: y = " & STD_LOGIC'image(y);
      if(y /= yexpected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if(is_x(testvectors(vectornum))) then
        if(errors = 0) then
          report "Just kidding --" &
                 integer'image(vectornum) &
                 "tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 "tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

VHDL代码比较冗繁，而且使用读取文件指令已经超出了本节的范围，但是这里也给出了一个自测测试程序的概况。

在时钟上升沿时向待测试设备提供新输入，在时钟下降沿时检查其输出。如果测试时序逻辑，还应该向待测试设备提供时钟（和复位信号）。测试程序可以在发生错误时报告，也可以在模拟结束时打印全部测试向量和检测到的错误数。

对这样简单的电路使用HDL例子4.39的测试程序有点过分了。然而，经过简单地修改，它可以测试更复杂的电路，修改的内容主要包括：example.tv、实例化新的待测试设备、更改一些代码以设置输入和检查输出。

4.9 总结

对于现代数字设计，硬件描述语言（HDL）是十分重要的工具。当学会了Verilog或者VHDL，就可以比手工绘制图表更快地描述数字系统。而且因为修改时只需要改代码，而不是烦琐地重绘电路图，所以调试周期也会更快。然而，如果不熟悉代码所代表的硬件，使用硬件描述语言的调试周期可能会更长。

硬件描述语言用于模拟和综合。逻辑模拟是在系统转化为硬件前在计算机上进行测试的强大方法。模拟器可以检查系统中在物理硬件中不可能被测量的信号。逻辑综合把硬件描述语言代码转换成数字逻辑电路。

最需要记住的是：编写硬件描述语言代码是在描述一个真实存在的硬件，而不是编写一

个软件程序。很多初学者的常见错误是不考虑准备产生的硬件而编写硬件描述语言代码。如果不知道要表示的硬件是什么，那么也肯定不能得到想要的东西。相反地，应该从画系统的结构图开始，区分哪些部分是组合逻辑，哪些部分是时序逻辑或有限状态机。随后使用能描述目标硬件的正确风格为每一个部分编写硬件描述语言代码。

习题

以下的习题可以用读者习惯的硬件描述语言完成。如果可以使用模拟器，那么测试相应的设计。打印出波形，并解释它们如何证明设计是可行的。如果可以使用综合器，则综合相应的代码。打印出生成的电路图，解释为什么它符合预想。

4.1 画出用以下HDL描述的电路图。简化电路图以使用最少的门表示。

Verilog

```
module exercise1(input  a, b, c,
                 output y, z);

  assign y = a & b & c  |  a & b & ~c  |  a & ~b & c;
  assign z = a & b  |  ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and (not c)) or
       (a and (not b) and c);
  z <= (a and b) or ((not a) and (not b));
end;
```

4.2 画出用以下HDL描述的电路图。简化电路图以使用最少的门表示。

Verilog

```
module exercise2(input      [3:0] a,
                 output reg [1:0] y);

  always @(*)
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
```

```
begin
  process (a) begin
    if    a(0) = '1' then y <= "11";
    elsif a(1) = '1' then y <= "10";
    elsif a(2) = '1' then y <= "01";
    elsif a(3) = '1' then y <= "00";
    else                  y <= a(1 downto 0);
    end if;
  end process;
end;
```

4.3 编写一个HDL模块，计算四输入XOR函数。输入为$a_{3:0}$，输出为y。

4.4 为习题4.3编写一个自测测试程序。生成一个包含所有16个测试用例的测试向量文件。模拟电路的运行以证明它可以工作。在测试向量文件中引入一个错误，以检查测试程序能否报告相应的错误。

4.5 编写一个叫做minority的HDL模块。它接收a、b和c等3个输入。产生一个输出y，在至少两个输入为FALSE时输出TRUE。

4.6 编写一个HDL模块用于十六进制的7段数码管显示译码器。该译码器能显示0～9，也能显示A, B, C, D, E, F。

4.7 为习题4.6编写一个自测测试程序。生成一个测试向量文件，其中包括所有16个测试用例。模拟电路的运行以证明它可以工作。在测试向量文件中引入一个错误，以检查测试程序能否报告相应的错误。

4.8 编写一个8：1多路选择器模块mux8，输入为$S_{2:0}$，d0，d1, d2, d3, d4, d5, d6, d7，输出为y。

4.9 编写一个结构模块，使用多路选择器逻辑计算逻辑函数 $y = a\bar{b} + \overline{bc} + \bar{a}bc$。使用习题4.8中的8：1多路选择器。

4.10 使用4：1多路选择器和非门重新实现习题4.9的内容。

4.11 在4.5.4小节中指出如果赋值顺序合适的话，综合器也能正确地用阻塞式赋值描述。想出一个简单的时序电路，它无论采用何种顺序都不能用阻塞式赋值描述。

4.12 编写一个8输入优先电路的HDL模块。

4.13 编写一个2：4译码器的HDL模块。

4.14 使用习题4.13中的2：4译码器实例和一些3输入与门编写一个6：64译码器的HDL模块。

4.15 编写HDL模块实现习题2.7中的布尔等式。

4.16 编写HDL模块实现习题2.8中的电路。

4.17 编写HDL模块实现习题2.19的逻辑功能。注意如何怎样处理那些无关项。

4.18 编写HDL模块实现习题2.24的功能。

4.19 编写HDL模块实现习题2.25的优先编码器。

4.20 编写HDL模块实现习题2.27的二进制数到温度计码转换器。

4.21 编写一个模块实现问题2.2中判断一个月天数的功能。

4.22 给出以下HDL代码描述的有限状态机状态转换图。

Verilog

```
module fsm2 (input  clk, reset,
             input  a, b,
             output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
```

```
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always @ (*)
    case (state)
      S0: if (a ^ b)  nextstate = S1;
          else        nextstate = S0;
      S1: if (a & b)  nextstate = S2;
          else        nextstate = S0;
      S2: if (a | b)  nextstate = S3;
          else        nextstate = S0;
      S3: if (a | b)  nextstate = S3;
          else        nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port (clk, reset: in  STD_LOGIC;
        a, b:       in  STD_LOGIC;
        y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  process (state, a, b) begin
    case state is
      when S0 => if (a xor b) = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if (a and b) = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

4.23 给出以下HDL代码描述的有限状态机状态转换图。这种有限状态机用于一些微处理器的分支预测。

Verilog

```
module fsm1(input  clk, reset,
            input  taken, back,
            output predicttaken);

  reg [4:0] state, nextstate;

  parameter S0 = 5'b00001;
  parameter S1 = 5'b00010;
  parameter S2 = 5'b00100;
  parameter S3 = 5'b01000;
  parameter S4 = 5'b10000;

  always @ (posedge clk, posedge reset)
    if(reset) state <= S2;
    else      state <= nextstate;

  always @ (*)
    case(state)
      S0: if(taken) nextstate = S1;
          else      nextstate = S0;
      S1: if(taken) nextstate = S2;
          else      nextstate = S0;
      S2: if(taken) nextstate = S3;
          else      nextstate = S1;
      S3: if(taken) nextstate = S4;
          else      nextstate = S2;
      S4: if(taken) nextstate = S4;
          else      nextstate = S3;
      default:      nextstate = S2;
    endcase

  assign predicttaken = (state == S4)  |  |
                        (state == S3)  |  |
                        (state == S2 && back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset:   in  STD_LOGIC;
       taken, back:  in  STD_LOGIC;
       predicttaken: out STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S2;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  process(state, taken) begin
    case state is
      when S0 => if taken = '1' then
```

```
                    nextstate <= S1;
                else nextstate <= S0;
                end if;
    when S1 => if taken = '1' then
                    nextstate <= S2;
                else nextstate <= S0;
                end if;
    when S2 => if taken = '1' then
                    nextstate <= S3;
                else nextstate <= S1;
                end if;
    when S3 => if taken = '1' then
                    nextstate <= S4;
                else nextstate <= S2;
                end if;
    when S4 => if taken = '1' then
                    nextstate <= S4;
                else nextstate <= S3;
                end if;
    when others => nextstate <= S2;
  end case;
end process;

-- output logic
predicttaken <= '1' when
                ((state = S4) or (state = S3) or
                (state = S2 and back = '1'))
  else '0';
end;
```

4.24 为SR锁存器编写一个HDL模块。

4.25 为JK触发器编写一个HDL模块。触发器的输入为clk，J和K，输出为Q。时钟上升沿时，如果J = K = 0，Q保持旧值。当J = 1时，Q被设为1，如果K=1，那么Q被重设为0。如果J = K = 1，那么Q被设为相反值。

4.26 为图3-18的锁存器编写一个HDL模块。用一个赋值语句为每一个门赋值。每个门的延迟设置为1单位或者1ns。模拟锁存器，证明它能正确操作。之后增加反相器的延迟。设置多长的延迟，才能避免竞争产生的锁存器故障呢？

4.27 为3.4.1小节的交通灯控制器编写HDL模块。

4.28 为例3.8中巡游模式交通灯控制器的分解状态机编写3个HDL模块。模块名字分别为controller，mode，lights，它们的输入输出如图3-33（b）所示。

4.29 编写一个描述图3-40电路的HDL模块。

4.30 为习题3.19中图3-65给出的有限状态机的状态转换图编写一个HDL模块。

4.31 为习题3.20中的图3-66给出的有限状态机的状态转换图编写一个HDL模块。

4.32 为习题3.21的改进交通灯控制器编写HDL模块。

4.33 为习题3.22中蜗牛女儿的例子编写HDL模块。

4.34 为习题3.23中苏打汽水售卖机的例子编写HDL模块。

4.35 为习题3.24中Gary码计数器编写HDL模块。

4.36 为习题3.25中UP/DOWN Gary码计数器编写HDL模块。

4.37 为习题3.26的有限状态机编写HDL模块。

4.38 为习题3.27的有限状态机编写HDL模块。

4.39 为问题3.2中的连续2补码器编写一个HDL模块。

4.40 为习题3.28中的电路编写HDL模块。

4.41 为习题3.29中的电路编写HDL模块。

4.42 为习题3.30中的电路编写HDL模块。

4.43 为习题3.31中的电路编写HDL模块。可能需要用到4.2.5小节中的全加器。

Verilog习题

以下习题用Verilog完成。

4.44 Verilog中声明为reg的信号代表什么意思?

4.45 重写HDL例子4.30的synbad模块。使用非阻塞式赋值，但是把代码修改成用两个触发器产生正确的同步器。

4.46 考虑以下两个Verilog模块。它们的功能一样吗？描述各自表示的硬件。

```
module code1(input      clk, a, b, c,
             output reg y);
  reg x;

  always @ (posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2(input      a, b, c, clk,
             output reg y);
  reg x;

  always @ (posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

4.47 在每个赋值中用=代替<=，重新讨论习题4.46的问题。

4.48 以下的Verilog模块是作者在实验室看到的学生代码错误。说出每个模块的错误，并指出如何修改它。

(a)
```
module latch(input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (clk)
    if(clk)q <= d;
endmodule
```

(b)
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

  always @ (a)
    begin
      y1 = a & b;
      y2 = a | b;
      y3 = a ^ b;
      y4 = ~(a & b);
      y5 = ~(a | b);
    end
endmodule
```

(c)
```
module mux2(input      [3:0] d0, d1,
            input            s,
            output reg [3:0] y);

  always @ (posedge s)
    if(s)y <= d1;
    else  y <= d0;
endmodule
```

(d)
```
module twoflops(input      clk,
                input      d0, d1,
                output reg q0, q1);

   always @ (posedge clk)
      q1 = d1;
      q0 = d0;
endmodule
```

(e)
```
module FSM(input      clk,
           input      a,
           output reg out1, out2);

   reg state;

   // next state logic and register (sequential)
   always @ (posedge clk)
      if(state == 0) begin
         if(a)  state <= 1;
      end else begin
         if(~a) state <= 0;
      end

   always @ (*) // output logic (combinational)
      if(state == 0) out1 = 1;
      else           out2 = 1;
endmodule
```

(f)
```
module priority(input      [3:0] a,
                output reg [3:0] y);

   always @ (*)
      if      (a[3]) y = 4'b1000;
      else if(a[2]) y = 4'b0100;
      else if(a[1]) y = 4'b0010;
      else if(a[0]) y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

   reg [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @ (posedge clk, posedge reset)
      if(reset) state <= S0;
      else      state <= nextstate;

   // Next State Logic
   always @ (state)
      case(state)
        S0: nextstate = S1;
        S1: nextstate = S2;
         2: nextstate = S0;
      endcase

   // Output Logic
   assign out =(state == S2);
endmodule
```

(h)
```
module mux2tri(input  [3:0] d0, d1,
               input        s,
               output [3:0] y);

   tristate t0(d0, s, y);
   tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input            clk,
                input            reset,
                input            set,
                input      [3:0] d,
                output reg [3:0] q);

   always @(posedge clk, posedge reset)
     if(reset) q <= 0;
     else      q <= d;

   always @(set)
     if(set)   q <= 1;
endmodule
```

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

   reg tmp;

   always @(a, b, c)
   begin
     tmp <= a & b;
     y   <= tmp & c;
   end
endmodule
```

VHDL习题

以下习题用VHDL完成。

4.49 在VHDL中，为什么写为`q<='1' when state = s0 else '0';`而不写为`q<=(state = s0);`?

4.50 以下每个VHDL模块中都包含一个错误。为简单起见，只给出了结构描述。假设library调用子句和entity声明都是正确的。解释错误并修正它。

(a)
```
architecture synth of latch is
begin
  process(clk) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process(a) begin
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process(clk)
    if clk'event and clk = '1' then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process(a) begin
    if    a(3)= '1' then y <= "1000";
    elsif a(2)= '1' then y <= "0100";
    elsif a(1)= '1' then y <= "0010";
    elsif a(0)= '1' then y <= "0001";
    end if;
```

```
    end process;
  end;
```

(e)
```
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;
    process (state) begin
     case state is
       when S0 => nextstate <= S1;
       when S1 => nextstate <= S2;
       when S2 => nextstate <= S0;
     end case;
    end process;

  q <= '1' when state = S0 else '0';
end;
```

(f)
```
architecture struct of mux2 is
  component tristate
    port (a:  in  STD_LOGIC_VECTOR (3 downto 0);
          en: in  STD_LOGIC;
          y:  out STD_LOGIC_VECTOR (3 downto 0));
  end component;
begin
  t0: tristate port map (d0, s, y);
  t1: tristate port map (d1, s, y);
end;
```

(g)
```
architecture asynchronous of flopr is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;

  process (set) begin
    if set = '1' then
      q <= '1';
    end if;
  end process;
end;
```

(h)
```
architecture synth of mux3 is
begin
  y <= d2 when s(1) else
       d1 when s(0) else d0;
end;
```

面试问题

4.1 编写HDL代码以门控32位data总线，其中sel信号产生32位的result信号。如果sel为TRUE，result = data。否则，result等于0。

4.2 以例子说明Verilog中阻塞式赋值和非阻塞式赋值的不同。

4.3 以下的Verilog语句将会做什么操作？

```
result = | (data[15:0] & 16'hC820);
```

第5章　常见数字模块

5.1　引言

到目前为止，我们已经介绍了使用布尔表达式、电路图和硬件描述语言来设计组合电路和时序电路。本章将详细介绍数字系统中常见的组合电路和时序电路模块，主要包括算术运算电路、计数器、移位寄存器、存储器阵列和逻辑阵列。这些模块自身有重要作用，而且还说明了层次化，模块化，规整化的原则。复杂模块可以用层次化的方法由更简单的模块（如逻辑门电路、多路选择器、译码器等）组成。每个模块都有定义好的接口，当底层实现不重要时，可以被视为黑盒。每一个规整结构的模块都应易于扩展为不同规模。第7章中将使用这些模块构成一个微处理器。

5.2　算术电路

算术电路是计算机的主要模块。计算机和数字逻辑可以实现很多算术功能，例如：加法、减法、比较、移位、乘法和除法。本节将介绍实现这些操作的硬件。

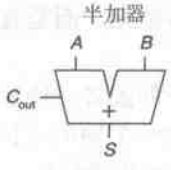
5.2.1　加法

加法是数字系统中最常见的操作之一。首先考察两个一位的二进制数如何相加。然后再扩展到N位二进制数。加法器同时说明了速度和硬件复杂度之间的不同折中。

半加法器

首先从构建一位半加法器（half adder）开始。如图5-1所示，半加法器有两个输入A和B，两个输出S和C_{out}。S是A和B之和。如果A和B都是1，S就是2，但2不能用一位二进制数表示。作为代替，用另一列输出C_{out}表示。半加法器可以用一个XOR门电路和一个AND门电路实现。

在多位加法器中，C_{out}会被相加或者进位到下一个高位。例如，在图5-2中的进位C_{out}是第一列的一位加法输出，同时也是第二列加法的输入C_{in}。然而，半加法器缺少一个输入C_{in}去接受之前列的输出C_{out}。下节中介绍的全加器会解决这个问题。



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

图5-1　1位半加器

```
  1
 0001
+0101
 0110
```

图5-2　进位

全加器

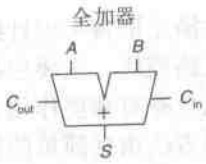
如图5-3所示，全加器（full adder）接收进位C_{in}。图5-3中还给出了S和C_{out}的输出表达式。

进位传播加法器

一个N位的加法器将两个N位输入和一位进位C_{in}相加，产生一个N位结果S和一个输出进位

C_{out}。因为进位将会传播到下一位中，这种加法器通常称为进位传播加法器(carry propagate adder, CPA)。CPA的符号如图5-4所示，除了A，B，S是总线而不是单独一位外，它和一个全加器画起来很像。三种常见的CPA实现分别是行波进位加法器、先行进位加法器和前缀加法器。

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

图5-3 1位全加器

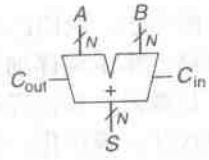


图5-4 进位传播加法器

行波进位加法器

构造N位进位传播加法器的最简单方法就是把N个全加器串联起来。如图5-5的32位加法器所示，行波进位加法器（Ripple-Carry Adder）中一级的C_{out}就是下一级的C_{in}。这是模块化和规整化的一个应用范例：全加器模块在一个更大的系统中被多次重用。行波进位加法器有一个缺点：当N比较大的时候，运算速度会慢下来。例如在图5-5中，S_{31}依赖于C_{30}，C_{30}依赖于C_{29}，C_{29}又依赖于C_{28}，如此类推。归根到底依赖于C_{in}。可以看出，进位以串行通过进位链。加法器的延迟t_{ripple}直接随位数的增长而增长，如等式5-1所示，其中t_{FA}是全加器的延迟。

$$t_{ripple} = Nt_{FA} \tag{5-1}$$

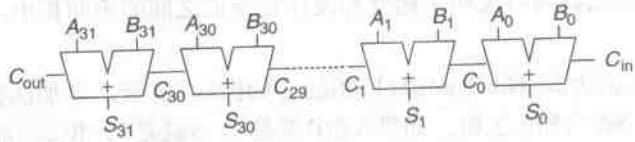


图5-5 32位行波进位加法器

先行进位加法器

大型行波进位加法器运算缓慢的根本原因是进位信号必须依次在加法器中的每一位传输。先行进位加法器（Carry-Lookahead Adder）是另外一种类型的进位传播加法器，它解决进位问题的方法是：把加法器分解成若干块，同时额外增加电路在每块一得到输入进位时就快速算出此块的输出进位。因此它不需要等待进位通过一块内的所有加法器单元，而是直接先行通过每个单元。这里以将一个32位加法器分解成8个4位的块为例。

先行进位加法器用G（generate）和P（propagate）两个信号描述一列或者一块如何确定进位输出。在不考虑进位输入的情况下，如果加法器的第i列必然能产生一个输出进位，则称此列为产生（generate）进位。一个加法器的第i列在A_i和B_i都为1时，必产生进位C_i。因此第i列的产生信号G_i可以这样计算，$G_i = A_iB_i$。如果在有进位输入时，第i列会产生一个进位输出，那么此列就称为传播（propagation）进位。如果A_i或者B_i为1，第i列会传播一个进位输入C_{i-1}，因此，$P_i = A_i + B_i$。利用这些定义，可以为加法器的特定列重写进位逻辑。如果加法器的第i列会产生一个进位G_i，或者传播进位输入P_iC_{i-1}，它就会产生进位输出C_i，表达式为

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1} \tag{5-2}$$

产生和传播的定义可以扩展到多位构成的块。如果一个块在不考虑进位输入的情况下也能产生进位输出，称其产生进位。如果一个块在有进位输入时候能产生进位，称其为传播进位。定义$G_{i:j}$和$P_{i:j}$为从第i位到第j位块的产生和传播信号。

一个块产生一个进位的条件是：最高位列产生一个进位，或者如果最高位列传播进位而且之前的列产生了进位，如此类推。例如，一个第3位到第0位的块产生逻辑如下所示

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)) \tag{5-3}$$

一个块传播进位的条件是：块中所有的列都能传播进位。例如，一个从第3位到第0位的传播逻辑如下

$$P_{3:0} = P_3P_2P_1P_0 \tag{5-4}$$

使用块的生成和传播信号，可以根据块的进位输入C_j快速计算出块的进位输出C_i。

$$C_i = G_{i:j} + P_{i:j}C_j \tag{5-5}$$

图5-6(a) 是一个由8个4位块组成的32位先行进位加法器。每一个单元包含一个4位的行波进位加法器和一些根据进位输入提前计算进位输出的逻辑，如图5-6(b) 所示。为简化起见，图中没有画出用于计算每一位A_i和B_i的产生信号G_i和输出信号和P_i所需要的AND门和OR门来计算。同样地，先行进位加法器也体现了模块化和规整化。

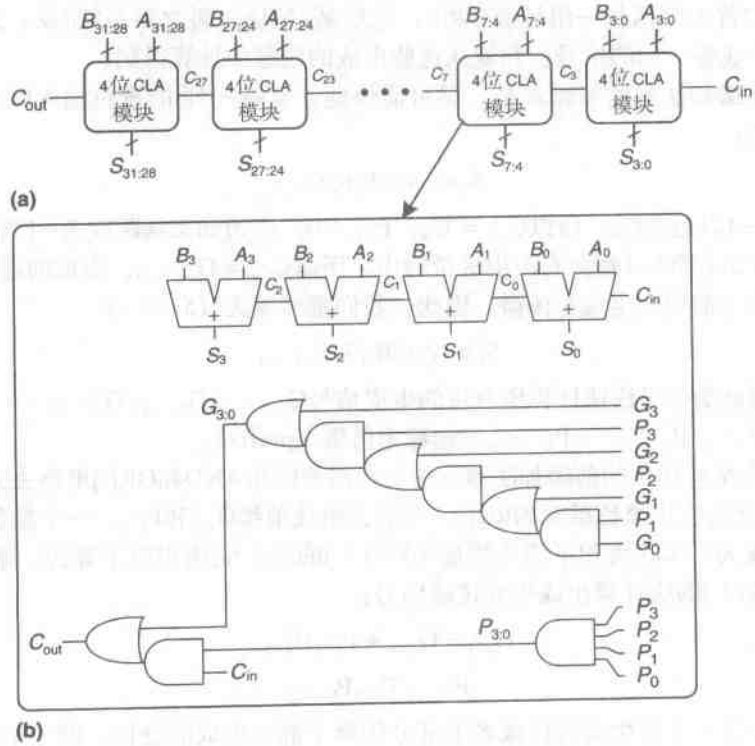


图5-6 (a) 32位先行进位加法器（CLA）；(b) 4位CLA模块

所有的CLA单元同时地计算一位并产生块的生成和传播信号。关键路径从首个CLA块中计算G_0和$G_{3:0}$开始。接着C_{in}直接通过每块中的AND/OR门电路向前传输，直到最后。在大型加法器中，这会比等待所有的进位行波式通过每一个加法器要快很多。最后，关键路径通过

最后一个块中包含的短行波进位加法器。因此，一个分解成k位块的N位加法器延迟为

$$t_{CLA} = t_{pg} + t_{pg_block} + \left(\frac{N}{k} - 1\right) t_{AND_OR} + kt_{FA} \tag{5-6}$$

式中t_{pg}为单独一个生成产生信号P和传播G的门电路（一个单独的AND或者OR门电路），t_{pg_block}为在k位块中生成产生信号$P_{i:j}$和输出信号$G_{i:j}$的延迟，t_{AND_OR}为在k位CLA块中C_{in}从AND/OR逻辑到C_{out}的延迟。当N>16时，先行进位加法器一般总会比行波进位加法器快很多。然而，加法器的延迟依然随N线性增长。

例5.1 行波进位加法器和先行进位加法器的延迟

对比32位行波进位加法器和4位块组成的32位先行进位加法器的延迟。假设每个两输入门电路的延迟为100ps，全加器的延迟是300ps.

解：通过式（5-1）计算，32位行波进位加法器的传输延迟是$32 \times 300ps = 9.6ns$。

CLA的$t_{PG} = 100ps$，$t_{pg_block} = 6 \times 100ps = 600ps$，$t_{AND_OR} = 2 \times 100ps = 200ps$。由式（5-6），4位块组成的32位先行进位加法器传输延迟为：$100ps + 600ps + (32/4-1) \times 200ps + (4 \times 300ps) = 3.3ns$，几乎比行波进位加法器快3倍。

*前缀加法器**

前缀加法器（Prefix Adder）扩展了先行进位加法器的生成和传播逻辑，可以进行更快的加法运算。它们首先以两位一组计算G和P，之后是4位块，再之后是8位块，之后是16位块，如此类推直到生成每一列的信号。和就从这些生成的信号中计算得到。

换言之，前缀加法器的策略就是，尽可能快地计算每一列i的进位输入C_{i-1}，之后使用下述等式计算总值：

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5-7}$$

定义列i = −1以包含C_{in}，所以$G_{-1} = C_{in}$，$P_{-1} = 0$。因为如果从跨度为−1到i−1位中生成一个进位的话，那么在列i−1将会有产生进位输出，所以$C_{i-1} = G_{i-1:-1}$。生成的进位要么在i−1列中生成，要么在之前列中生成并传播。因此，我们重写等式（5-7）为

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5-8}$$

因此，问题就集中于快速计算所有块的生成信号$G_{-1:-1}$，$G_{0:-1}$，$G_{2:-1}$，…，$G_{N-2:-1}$。这些信号和$P_{-1:-1}$，$P_{0:-1}$，$P_{2:-1}$，…，$P_{N-2:-1}$一起称为前缀（prefix）。

图5-7是一个N = 16位的前缀加法器。这个加法器以用AND和OR门电路去为每一列的A_i和B_i产生P_i和G_i。之后它用黑色单元的$\log_2 N = 4$层去组成前缀$G_{i:j}$和$P_{i:j}$。一个黑色单元的输入包括：上部分跨度为i：k的块和下部分跨度为k−1：j的块。它使用以下等式，组合这两部分信号为整个跨度为i：j的块计算生成生和传播信号：

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{5-9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{5-10}$$

换言之，如果上部分生成进位或者上部分传播下部分生成的进位，则一个跨度为i：j的块将会生成一个进位。如果上部分和下部分都能传播进位，则一个块也能传播进位。最后，前缀加法器使用式（5-8）计算总和。

总的来说，前缀加法器的延迟以加法器位数的对数增长，而不是线性增长。这明显提高了速度，特别在加法器位数超过32位时。但是它比简单的先行进位加法器需要消耗更多的硬

件资源。黑色单元构成的网络称为前缀树(prefix tree)。

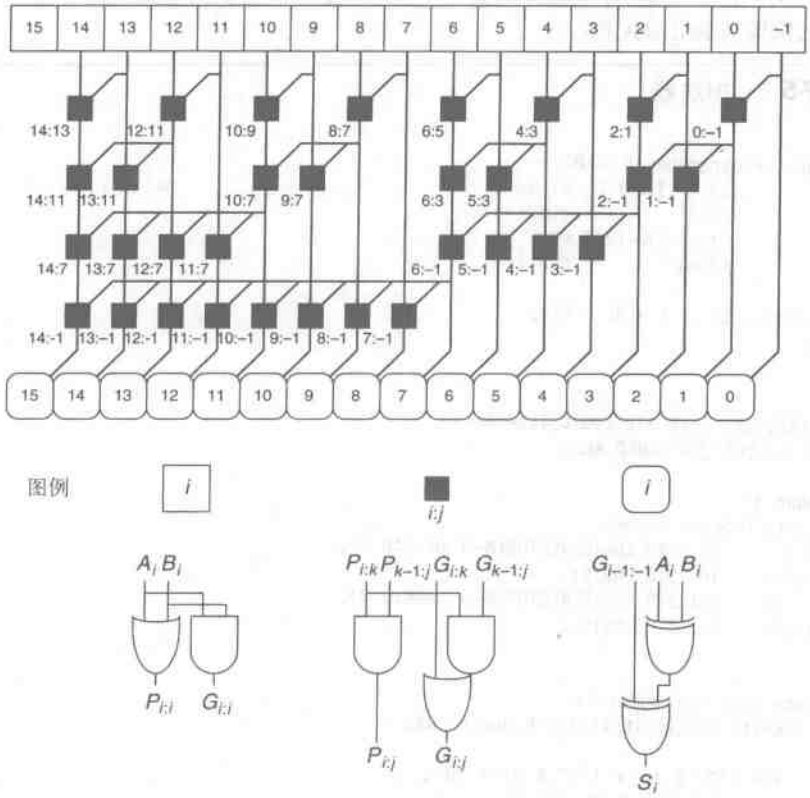


图5-7 16位前缀加法器

使用前缀树计算，使其执行延时按输入位数的对数增长。这种方法是很有用的技术。发挥一下智慧，它可以应用到其他类型电路中（参见练习5.7）。

N位前缀加法器的关键路径包括P_i和G_i的预计算。通过黑色前缀单元的$\log_2 N$步获得所有前缀。$G_{i-1:-1}$之后通过底部最后的XOR门电路计算S_i。N位前缀加法器的延迟可表示为

$$t_{PA} = t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR} \tag{5-11}$$

式中t_{pg_prefix}是黑前缀单元的延迟。

例5.2 前缀加法器的延迟

计算32位前缀加法器的延迟。假设每一个二输入门电路的延迟是100ps。

解：每一个黑前缀单元的输出延迟t_{pg_prefix}是200ps（即2个门电路的延迟）。因此，使用等式（5-11），32位的前缀加法器的输出延迟是$100\text{ps} + \log_2(32) \times 200\text{ps} + 100\text{ps} = 1.2\text{ns}$，比先行进位加法器快3倍，比例5.1中的行波进位加法器快8倍。在现实中，效益可能没有那么大，但是前缀加法器依然是所有选择中最快的。

小结

这个小节介绍了半加法器，全加法器和三种进位传播加法器：行波进位加法器、先行进位加法器和前缀加法器。更快的加法器需要更多的硬件，所以成本和功耗也都更高。设计中选择合适的加法器需要充分考虑这些折中。

硬件描述语言提供“+”操作来描述CPA。现代的综合工具会从众多可能的实现方法中选择最便宜（最小）的设计去满足速度的要求。这极大地简化了设计者的工作。HDL例子5.1描述了一个有进位输入输出的CPA。

HDL例子5.1 加法器

Verilog

```
module adder #(parameter N = 8)
              (input  [N-1:0] a, b,
               input          cin,
               output [N-1:0] s,
               output         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is
  generic (N: integer := 8);
  port (a, b:   in  STD_LOGIC_VECTOR(N-1 downto 0);
        cin:    in  STD_LOGIC;
        s:      out STD_LOGIC_VECTOR(N-1 downto 0);
        cout:   out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result   <= ("0" & a) + ("0" & b) + cin;
  s        <= result (N-1 downto 0);
  cout     <= result (N);
end;
```

图5-8给出了综合后的加法器

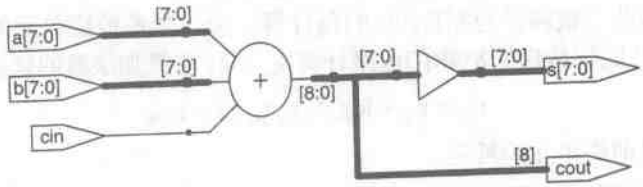


图5-8 综合后的加法器

5.2.2 减法

回想1.4.6节中加法器可以使用二进制补码表示完成正数和负数的加法。减法非常简单：改变减数的符号，然后做加法。改变二进制补码的符号就是翻转所有的位，然后加一。

计算Y = A−B，首先生成减数B的二进制补码。翻转B的所有位得到$\overline{B}$，之后加1得到−B = $\overline{B}$+1。把这个值和被减数A相加，得到Y = A +$\overline{B}$+ 1 = A−B。可以通过进位传播加法器得到和，其中设置C_{in} = 1，加数和被加数分别为A和$\overline{B}$。图5-9为一个减法器的符号和底层硬件实现。HDL例子5.2描述了一个减法器。

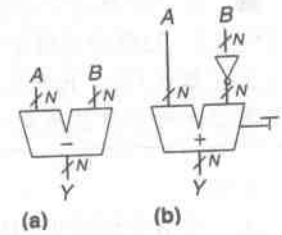


图5-9 减法器：(a) 电路符号；(b) 实现

HDL例子5.2 减法器

Verilog

```
module subtractor #(parameter N = 8)
                   (input  [N-1:0] a, b,
                    output [N-1:0] y);

  assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity subtractor is
  generic (N: integer := 8);
  port (a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
            y: out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

图5-10给出了综合后的减法器。

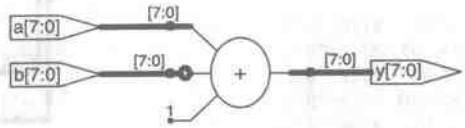


图5-10 综合后的减法器

5.2.3 比较器

比较器的作用是判断两个二进制数是否相等，或者一个比另一个大还是小。一个比较器输入为两个N位二进制数A和B。有两种常见类型的比较器。

相等比较器（equality comparator）产生一个单独的输出，以说明A是否等于B（A==B）。数量比较器（magnitude comparator）产生一个或者更多的输出以说明A和B的关系值。

相等比较器的硬件相对简单。图5-11给出了相等比较器的电路符号和4位相等比较器的实现。它首先通过XOR门电路检查A和B中每一对应的位是否相等。当每一位都相等时，它们就相等。

数量比较器首先计算A-B的值，再检查符号位（最高位）的结果，如图5-12所示。如果结果是负数（即符号位为1），A小于B；否则，A大于或等于B。

HDL例子5.3给出了不同的比较操作。

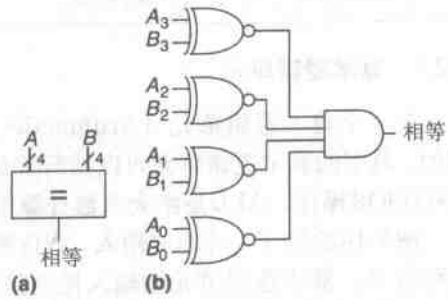


图5-11 4位相等比较器：(a) 电路符号；(b)实现

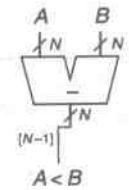


图5-12 N位数量比较器

HDL例子5.3 比较器

Verilog

```
module comparators # (parameter N = 8)
                    (input [N-1:0] a, b,
                    output         eq, neq,
                    output         lt, lte,
                    output         gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity comparators is
  generic (N: integer := 8);
  port (a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
        eq, neq, lt,
        lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparators is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

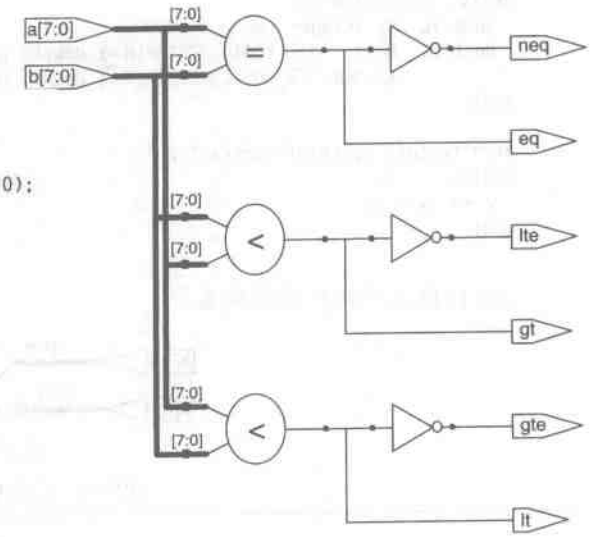


图5-13 综合后的比较器

图5-13给出了综合后的比较器。

5.2.4 算术逻辑单元

在一个算术逻辑单元（Arithmetic/Logic Unit, ALU）内组合了多种算术和逻辑的操作。例如，典型的算术逻辑单元可以执行加法、减法、数量比较、AND和OR操作。ALU是绝大多数计算机的核心。

图5-14给出了一个N位输入，N位输出的算术逻辑单元电路符号。算术逻辑单元的输入控制信号F说明执行哪个功能。表5-1列出了ALU可以执行的典型功能。SLT功能用作数量比较，将会在稍后的小节中讨论。

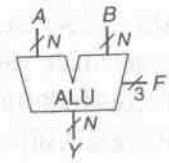


图5-14 算术逻辑单元电路符号

表5-1 ALU操作

$F_{2:0}$	功能	$F_{2:0}$	功能
000	A AND B	100	A AND $\overline{B}$
001	A OR B	101	A OR $\overline{B}$
010	A + B	110	A − B
011	未使用	111	SLT

图5-15给出一个算术逻辑单元的实现。其中包含了：一个N位的加法器和N个两输入AND和OR门电路；反相器，在F_2控制信号有效时翻转B；4：1多路选择器，根据$F_{1:0}$控制信号选择所需的功能。

更具体地说，此ALU中的算术和逻辑单元是对A和BB操作。BB是B或者是$\overline{B}$，其由F_2决定。如果$F_{1:0}=00$，多路选择器就会选择A AND BB作为输出。如果$F_{1:0}=01$，那么算术逻辑单元就会计算A OR BB。如果$F_{1:0}=10$，算术逻辑单元就执行加法或者减法。注意：F_2还是加法器的进位输入。而且二进制补码计算中，$\overline{B}+1=-B$。如果$F_2=0$，算术逻辑单元计算A+B。如果$F_2=1$，那么算术逻辑单元计算$A+\overline{B}+1=A-B$。

当$F_{2:0}=111$，算术逻辑单元就会执行小于置位(set if less than，SLT)操作。当A<B，Y=1；否则，Y=0。换言之，Y在A小于B时设置为1。

SLT以计算S=A－B的方法实现。如果S是负数(也就是说，符号位为1)，则A<B。零扩展单元的N位输出将1位输入和高位0拼接。S的符号位（第N－1位）是零扩展单元的输入。

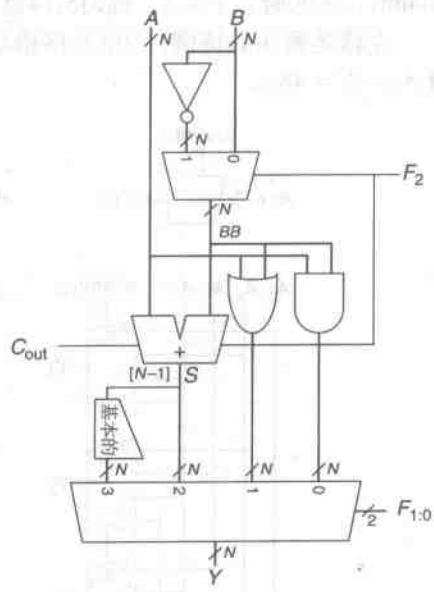


图5-15 N位算术逻辑单元

例5.3 小于置位操作SLT

配置一个32位算术逻辑单元用于SLT操作。假设$A=25_{10}$，$B=32_{10}$，写出控制信号和输出Y。

解：因为A<B，Y应为1。在SLT操作中，$F_{2:0}=111$。$F_2=1$设置加法器单元为减法运算，输出S为$25_{10}-32_{10}=-7_{10}=111\cdots1001_2$。$F_{1:0}=11$，最后的多路选择器会设置$Y=S_{31}=1$。

算术逻辑单元产生另外一些输出，称做标志位（flag）。它表示了算术逻辑单元的输出信息。例如，溢出标志说明加法器的结果有溢出。零标志说明算术逻辑单元的输出是0。

N位算术逻辑单元的硬件描述语言设计留在练习5.9中。这个基本算术逻辑单元有很多的变形以支持其他功能，例如XOR或者相等比较器。

5.2.5 移位器和循环移位器

移位器（shifter）和循环移位器（rotator）用于移动数字中位和完成2的整数次幂的乘法或除法操作。如名字所示，移位器根据特定的数左右移动二进制的数字。有一些常用的移位器：

- 逻辑移位器（logical shifter）——左移（LSL）或者右移（LSR）数，以0填充空位。

例如：11001 LSR 2 = 00110；11001 LSL 2 = 00100

- 算术移位器（arithmetic shifter）——和逻辑移位器一样，不过在算术右移（ASR）时会把原先数据的最高标志位填充在新数据的最高标志位上。这对于有符号数的乘法或者除法很有用（参看5.2.6小节和5.2.7小节）。算术左移（ASL）与逻辑左移是一样的。

例如：11001 ASR 2 = 11110; 11001 ASL 2=00100

- 循环移位器（Rotator）——循环转换数字，从一端移走的位重新填充到另一端的空位上。

例如：11001 ROR 2 = 01110; 11001 ROL 2 =00111

一个N位移位器可以用N个N：1多路选择器构成。根据$\log_2 N$位选择线的值，输入移位0到$N-1$位。图5-16为4位移位器的硬件和符号。操作符<<、>>和>>>分别表示左移，逻辑右移和

算术右移。根据2位位移量$shamt_{1:0}$，输出Y为输入A移动0到3位。对于所有的移位器，当$shamt_{1:0} = 00$时，Y = A。练习5.14包含了循环移位器的设计。

左移是乘法的特例。N位左移相当于对一个数乘以2^N倍。例如000011_2<<4=110000_2，相当于$3_{10} \times 2^4 = 48_{10}$。

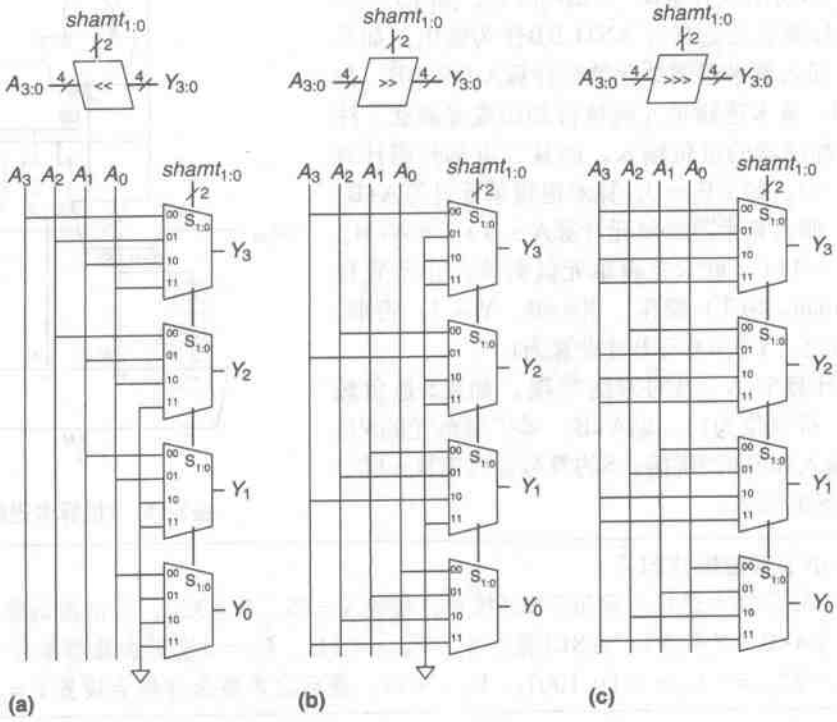


图5-16 4位移位器：(a) 左移；(b) 逻辑右移；(c) 算术右移

算术右移是除法的特例。N位的算术右移相当于对一个数除以2^N。例如11100_2 >>> 2 = 11111_2，相当于$-4_{10}/2^2 = -1_{10}$。

5.2.6 乘法*

无符号二进制数的乘法和十进制的乘法很相似，只不过它只有1和0而已。图5-17对比了二进制数和十进制数的乘法。在这两种情况下，部分积（partial product）为乘数的一位乘以被乘数的所有位。移位这些部分积，并将它们相加就可以得到最后结果。

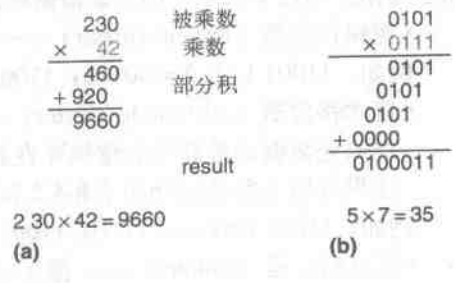


图5-17 乘法：(a) 十进制；(b) 二进制

概括地说，一个$N \times N$加法器对两个N位数相乘，产生一个$2N$位的结果。二进制乘法中，部分积要么是被乘数，要么全部为0。1位二进制乘法相当于AND操作，所以AND门电路用于产生部分积。

图5-18为一个4 × 4乘法器的电路符号、功能和实现。乘法器接收被乘数A和乘数B，然后产生积P。图5-18(b) 表示了如何形成部分积。每一个部分积是单独的乘数位（B_3, B_2, B_1, B_0）与被乘数的所有位（A_3, A_2, A_1, A_0）进行AND操作得出。对于N位操作数，会有N个部分积和$N-1$

级的N位加法器。例如，对于一个4×4的乘法器，第一行的部分积是B_0 AND （A_3, A_2, A_1, A_0）。这个部分积将会和已移位的第二个部分积B_1 AND （A_3, A_2, A_1, A_0）相加。后续行的AND门电路和加法器产生其他的部分积，并将它们相加。

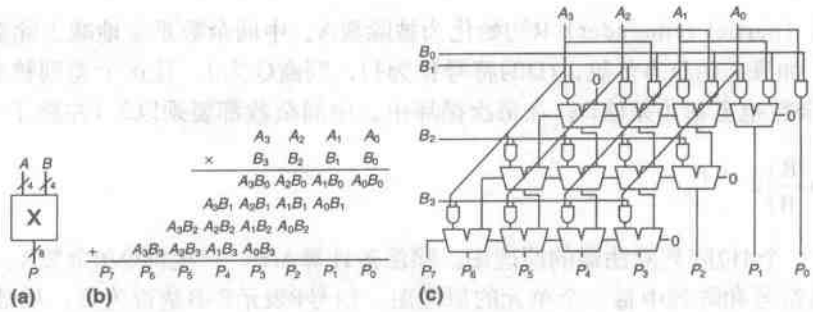


图5-18 4×4乘法器：(a) 符号；(b) 原理；(c) 实现

HDL例子5.4是一个乘法器的HDL语句。和加法器一样，不同的乘法器设计都有着不同的速度和成本。综合工具会根据给定的时间约束选择最合适的设计。

HDL例子5.4 乘法器

Verilog

```
module multiplier # (parameter N = 8)
                   (input  [N-1:0]   a, b,
                    output [2*N-1:0] y);

  assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier is
  generic (N: integer := 8);
  port (a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
        y: out STD_LOGIC_VECTOR(2*N-1 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

图5-19给出了综合后的乘法器。

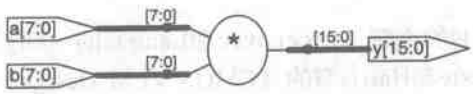


图5-19 综合后的乘法器

5.2.7 除法*

针对[2^N-1, 2^{N-1}]区间内规一化的无符号整数，二进制数除法按以下算法执行：

```
R = A
```

```
for i = N-1 to 0
  D = R - B
  if D < 0 then    Qi = 0, R' = R   // R < B
  else             Qi = 1, R' = D   // R ≥ B
  if i ≠ 0 then R = 2R'
```

中间余数（partial remainder）R初始化为被除数A。中间余数重复地减去除数B，以判断它是否合适。如果差值D为负数，(D的符号位为1)，则商Q_i为0，且这个差则被忽略。否则，Q_i为1，中间余数也更新为差值D。在每次循环中，中间余数都要乘以2（左移了一位）。结果符合 $\frac{A}{B}=\left(Q+\frac{R}{B}\right)2^{-(N-1)}$。

图5-20为一个4位阵列除法器的原理图。除法器计算A/B，产生商Q和余数R。图例给出了除法器的电路符号和阵列中每一个单元的原理图。信号P表示R-B是否为负，从最左行的单元输出C_{out}获得，为差值的符号位。

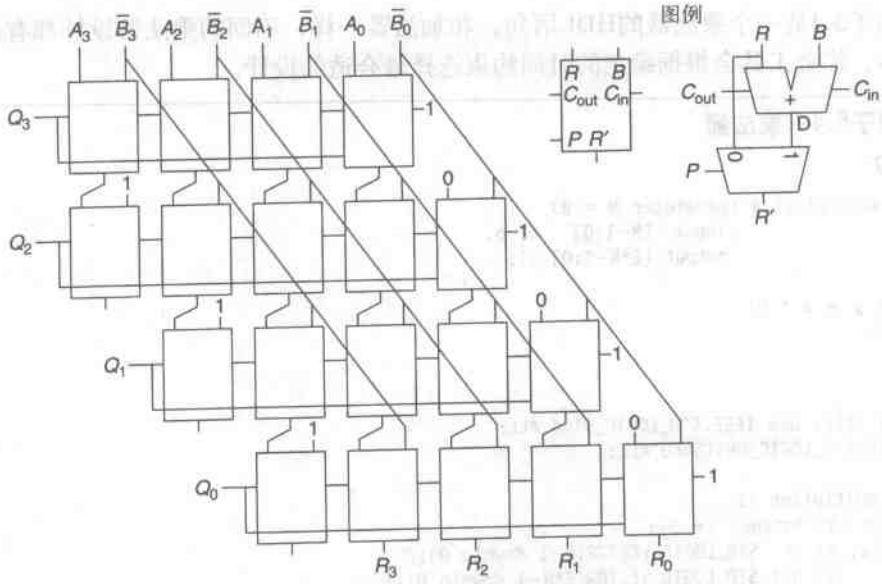


图5-20 阵列除法器

因为在确定符号和多路选择器决定选择R或者D前，进位必须逐次地通过一行中的所有N级，而且对于N行都需要完成这样的操作，所以N位除法器阵列延迟按N^2比例增长。除法是一个缓慢，并非常耗费硬件资源的操作，应尽量少地使用。

5.2.8 深入阅读

计算机算术可以是一本书的主题。Ercegovac和Lang写的《Digital Arithmetic》对这个领域进行了很精彩的介绍。Weste和Harris写的《CMOS VLSI Design》包括了高性能的算术运算电路设计。

5.3 数制系统

计算机可以对整数和小数操作。目前为止，我们只在1.4小节中考虑了有符号和无符号整数的表示。本节将介绍定点和浮点数系统，这样就可以表示有理数了。定点数与十进制

数类似，一些位表示整数部分，其余表示小数部分。浮点数和科学计数法相似，包括尾数和阶码。

5.3.1 定点数系统

定点（fixed-point）表示法有一个位于整数和小数位之间的隐含二进制小数点，类似于通常十进制数中位于整数和小数位之间的十进制小数点。例如，图5-21(a) 给出了一个有4位整数位和4位小数位的定点数。图5-21(b) 把隐含的二进制小数点标注出来，图5-21(c) 表示其十进制数值。

有符号定点数可以用二进制补码或者带符号的原码表示。图5-22给出了-2.375的定点数表示法，其中包括了4位整数位和4位小数位。为了阅读清楚，隐含的二进制小数点标注出来。在带符号的原码中，最高有效位用于表示符号。二进制补码形式是将数的绝对值取反，然后在最低有效位加1。在这个例子中，最低有效位的位置在2^{-4}列。

(a) 01101100
(b) 0110.1100
(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

图5-21 6.75用4个整数位和4个小数点位表示的定点表示

(a) 0010.0110
(b) 1010.0110
(c) 1101.1010

图5-22 −2.375的定点表示：(a) 绝对值；(b) 带符号的原码；(c) 二进制补码形式

和所有二进制数表示法一样，定点数只是位的集合。除非给出这些数的解释，否则无法知道是否存在隐含的二进制小数点。

例5.4 定点数的计算

使用定点数计算0.75 + −0.625

解：首先把第二个数字0.625转换为定点二进制表示。$0.625 \geqslant 2^{-1}$，所以在2^{-1}列有一个1，剩下$0.625-0.5 = 0.125$。因为$0.125<2^{-2}$，所以2^{-2}列为0。因为$0.125 \geqslant 2^{-1}$，于是2^{-3}列为1，剩下$0.125 - 0.125 = 0$。因此必须在2^{-4}列有一个0。把所有位放在一起，得到$0.625_{10} = 0000.1010_2$。

为使加法能正确进行，需要使用二进制补码代表有符号数。图5-23给出了−0.625转换为二进制补码表示的过程。

图5-24给出了二进制定点数加法，并与十进制加法之间进行了对比。注意：二进制定点加法中，图5-24(a) 中8位的结果忽略了溢出的1。

```
 0000.1010   二进制原码
 1111.0101   逐位取反
+        1   加1
 1111.0110   二进制补码
```

图5-23 定点数的二进制补码转换

```
  0000.1100          0.75
+ 1111.0110        + (−0.625)
 10000.0010          0.125
     (a)                (b)
```

图5-24 加法：(a) 二进制定点数加法；(b) 等价的十进制加法

5.3.2 浮点数系统*

浮点数与科学记数法相似。它解决了整数和小数位长度固定的限制，允许表示一个非常

大或者非常小的数。如科学计数法一样，浮点数包含了符号（sign），尾数（mantissa, M），基数（base, B）和阶码（exponent, E），如图5-25所示。例如，数字4.1×10^3是十进制数4100的科学计数法。它的尾数为4.1，基数为10，阶码为3。十进制小数点移动到最高有效位的后面。二进制浮点数的基数为2，并包含二进制尾数。在32位浮点数中用1位表示符号，8位表示阶码，23位表示尾数。

$\pm M \times B^E$

图5-25 浮点数

例5.5 32位浮点数

表示十进制数228的浮点数形式

解：首先转换十进制数为二进制数：$228_{10} = 11100100_2 = 1.11001 \times 2^7$。图5-26给出了其32位编码（后面将进一步修改以提高效率）。其中符号位为正（0），8阶码位表示值7，剩下的23位为尾数。

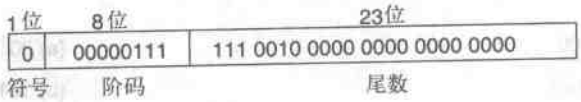


图5-26 32位浮点数表示——版本1

二进制浮点数中，尾数的第一位（二进制小数点的左端）经常为1，因为不需要储存。其称做隐含前导位（implicit leading one）。图5-27所示为经修改的浮点数$228_{10} = 11100100_2 \times 2^0 = 1.11001 \times 2^7$。因为效率关系，隐含前导位没有包含在23位的尾数中。只是小数部分的位被储存。这为有用的数据节省了一位。

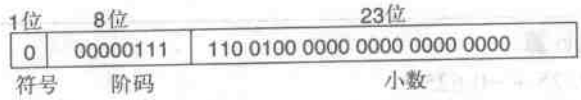


图5-27 32位浮点数表示——版本2

我们对阶码字段再做一次修改。阶码需要有正数形式和负数形式。要做到这点，浮点数使用了偏置（biased）阶码。它是原始的阶码加上一个恒定的偏置。32位浮点数使用的偏置是127。例如，对于阶码7，偏置阶码就是$7 + 127 = 134 = 10000110_2$。对于阶码−4，偏置阶码就是$-4 + 127 = 123 = 01111011_2$。图5-28给出了$1.11001_2 \times 2^7$的浮点表示，其中采用了隐含引导位和偏置阶码134（=7 + 127）。这种表示方法符合IEEE 754浮点数标准。

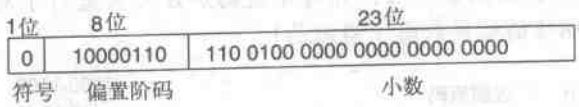


图5-28 IEEE 754 浮点表示法

特殊情况：0，±∞，和NaN

IEEE浮点数标准用特殊方式表示0，无穷大和非法结果等数。例如，在浮点数表示中隐含了前导位，所以表示数字0就存在问题。可以采用全0和全1填充的编码来解决了这些特殊的情况。表5-2为表示0、±∞和NaN的浮点数表示。和带符号的原码一样，浮点数也有正0和负0。NaN用于表示不存在的数，例如$\sqrt{-1}$或$\log_2(-5)$。

表5-2 IEEE 754对0、±∞和NaN的浮点数表示

数	符号	阶码	小数
0	X	00000000	000 000 000 000 000 000 000 00
∞	0	11111111	000 000 000 000 000 000 000 00
−∞	1	11111111	000 000 000 000 000 000 000 00
NaN	X	11111111	non-zero

单精度和双精度格式

目前为止，我们已经讨论过32位浮点数。这种格式称为单精度浮点（signle-precision, single or float）。IEEE 754标准还定义了64位的双精度浮点（double-precision, double）以提供更高的精度和更大的取值范围。表5-3所示为两种格式中不同字段的位数。

表5-3 单精度和双精度浮点数格式

格式	总位数	符号位	阶码位	小数位
单精度	32	1	8	23
双精度	64	1	11	52

排除前面提到的特殊情况，正常单精度数的取值范围是$\pm 1.175494 \times 10^{-38}$到$\pm 3.402824 \times 10^{38}$。它们有7位十进制有效数字（因为$2^{-24} = 10^{-7}$）。相似地，正常双精度取值范围为$\pm 2.22507385850720 \times 10^{-308}$到$\pm 1.79769313486232 \times 10^{308}$，精度为15位十进制有效数字。

舍入

算术结果中在有效精度外的数必须舍去，成为近似值。舍入的模式有：（1）向上舍；（2）向下舍；（3）向零舍；（4）向最近端舍。默认的舍入模式是向最近端舍。在向最近端舍进的模式中，如果两端的距离一样，则选择小数部分最低有效位为0的那个数。

当一个数的数值部分太大以致不能表示时，会产生上溢。同样地，当一个数太小时，会产生下溢。在向最近端舍进的模式中，上溢会被向上舍到±∞，下溢则向下舍成为0。

浮点数加法

浮点数的加法并不像二进制补码加法那么简单。同符号的浮点数加法步骤如下：

1) 分开阶码和小数位；

2) 加上前导1，形成尾数；

3) 比较阶码；

4) 如果需要，对较小的尾数移位；

5) 尾数相加；

6) 规整化尾数，并在需要时调整阶码；

7) 结果舍入；

8) 把阶码和小数组合成浮点数。

图5-29给出了7.875（1.11111×2^2）和0.1875（1.1×2^{-3}）的浮点数加法。结果为8.0625（1.0000001×2^3）。在第一步（分离阶码和小数部分），第二步（加上隐含前导位）后，通过较大阶码减去较小阶码的方式比较阶码字段。减法得出的结果就是第四步中较小的数右移以对齐二进制小数点的位数（使两者阶码相等）。对齐后的数相加。如果相加得到和的尾数大于等于2.0，需要结果右移一位以规整化，并在阶码中加1。在这个例子中，结果是准确的，所以不需要舍入。结果在去掉隐含前导位和加上符号位后，以浮点数格式存储。

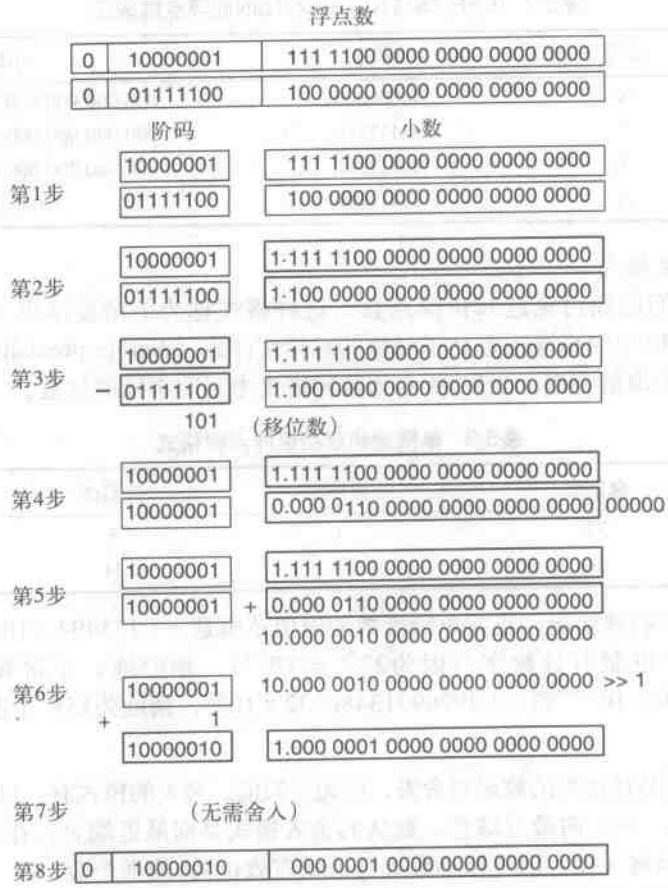


图5-29 浮点数加法

5.4 时序电路模块

本小节将介绍计数器和移位寄存器两种时序电路模块。

5.4.1 计数器

图5-30给出的N位二进制计数器（binary counter）是包含了时钟和复位输入，N位输出Q的时序算术电路。复位Reset将输出初始化为0。之后，计数器在每个时钟上升沿递增1，以按照二进制顺序输出所有2^N种可能的值。

图5-31给出了一个由加法器和可复位的寄存器构成的N位计数器。在每一个周期中，计数器对存储在寄存器中的值加一。HDL例子5.5描述了一个异步复位的二进制计数器。

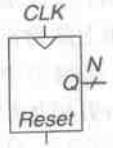


图5-30 计数器电路符号

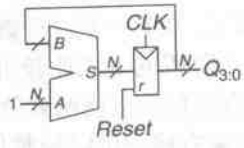


图5-31 N位计数器

HDL例子5.5 计数器

Verilog

```
module counter #(parameter N = 8)
               (input              clk,
                input              reset,
                output reg [N-1:0] q);

  always @ (posedge clk or posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity counter is
  generic (N: integer := 8);
  port (clk,
        reset: in STD_LOGIC;
        q: buffer STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= CONV_STD_LOGIC_VECTOR (0, N);
    elsif clk'event and clk = '1' then
      q <= q + 1;
    end if;
  end process;
end;
```

图5-32给出了综合后的计数器。

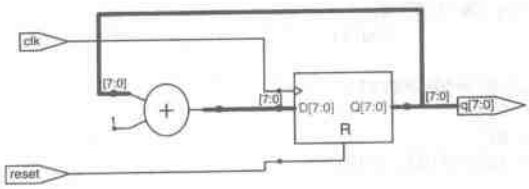


图5-32 综合后的计数器

UP/DOWN计数器等其他类型的计数器将在练习5.37到5.40中讨论。

5.4.2 移位寄存器

移位寄存器（shift register）的输入包括时钟、串行输入S_{in}，输出包括：串行输出S_{out}和N位并行输出$Q_{N-1:0}$，如图5-33所示。在每一个时钟上升沿，会从S_{in}移入一个新的位，所有后续内容都向前移动。移位寄存器的最后一位在S_{out}中。移位寄存器可以看作串行到并行的转换器。输入由S_{in}以串行方式提供（一次一位）。在N个周期后，前面的N位输入就在Q中，可并行访问。

移位寄存器可以用N个触发器串联而成，如图5-34所示。还可以采用复位信号来初始化移位寄存器中所有的触发器。

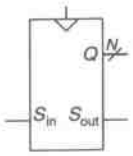


图5-33 移位寄存器电路符号

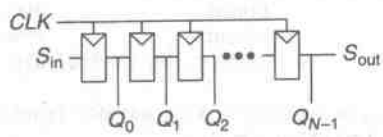


图5-34 移位寄存器电路原理图

一个相关的电路是并行到串行（parrallel-to-serial）转换器。它并行加载N位，然后一次移出一位。如图5-35所示，增加并行输入$D_{N-1:0}$和控制信号Load后，移位寄存器可以修改为既可完成串行到并行操作，也可完成并行到串行操作。Load信号有效时，触发器从输入D中并行加载数据。否则，移位寄存器就正常移位。HDL例子5.6描述了这样的移位寄存器。

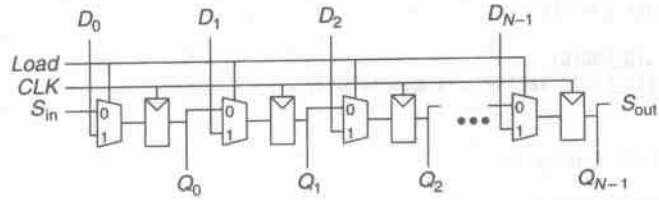


图5-35 并行读取的移位寄存器

HDL例子5.6 带并行加载的移位寄存器

Verilog

```
module shiftreg # (parameter N = 8)
                  (input                clk,
                   input                reset, load,
                   input                sin,
                   input      [N-1:0] d,
                   output reg [N-1:0] q,
                   output               sout);

  always @ (posedge clk or posedge reset)
    if (reset)       q <= 0;
    else if (load)   q <= d;
    else             q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shiftreg is
  generic (N: integer := 8);
  port(clk, reset,
       load: in   STD_LOGIC;
       sin:  in   STD_LOGIC;
       d:    in   STD_LOGIC_VECTOR(N-1 downto 0);
```

```
      q: buffer  STD_LOGIC_VECTOR(N-1 downto 0);
      sout: out  STD_LOGIC);
end;

architecture synth of shiftreg is
begin
 process (clk, reset) begin
  if reset = '1' then
    q <= CONV_STD_LOGIC_VECTOR (0, N);
  elsif clk'event and clk = '1' then
    if load = '1' then
      q <= d;
    else
      q <= q(N-2 downto 0) & sin;
    end if;
  end if;
 end process;
 sout <= q(N-1);
end;
```

图5-36给出了综合后的移位寄存器。

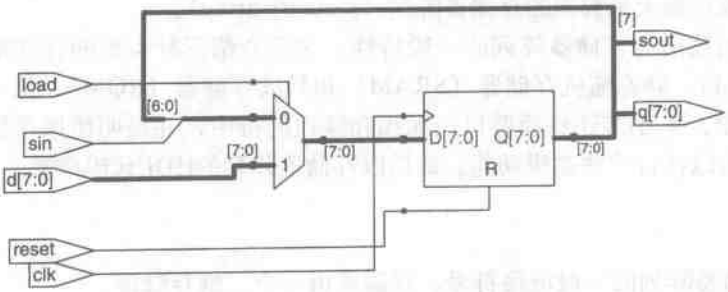


图5-36 综合后的移位寄存器

*扫描链**

通过扫描链（scan chain）技术，移位寄存器常用于测试时序电路。组合电路测试相对直观。向待测试系统输入测试向量（test vector），将结果与期待值比较。因为时序电路有状态，所以测试要困难一些。要使电路从一个已知的初始状态开始进入所需的状态，可能需要很多周期输入测试向量。例如，测试32位计数器的最高有效位从0到1的变化，需要复位累加器后，再提供2^{31}（大约二百万个）时钟脉冲！

为了解决这个问题，设计者希望可以直接观察和控制有限状态机的所有状态。这可以通过添加一个测试模式实现。在此模式下，所有触发器的内容可以读出或者载入所需要的值。大部分系统中的触发器数目太多，而不能为每个触发器分配一个管脚完成其读写。系统中所有的触发器可以连接在一个称为扫描链的移位寄存器中。在正常模式下，触发器从D输入读入数据，忽略扫描链。在测试模式下，触发器用S_{in}和S_{out}串行地移出它们的内容，或移入新内容。加载多路选择器常常集成于触发器中，构成一个可扫描触发器（scanable flip-flop）。图5-37为可扫描触发器的原理图和电路符号，并说明了这些触发器是如何级联起来，构成一个N位可扫描寄存器。

例如，在测试32位计数器时，可以在测试模式下移入011111...111，在正常模式中累加一个周期，然后移出结果（此结果应为1000...000）。这只需要32+1+32=65个周期。

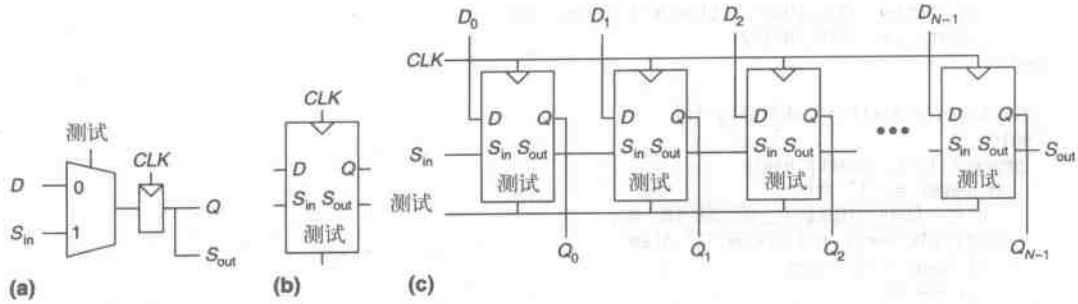


图5-37 可扫描触发器：(a)原理图 (b)电路符号 (c)N位可扫描寄存器

5.5 存储器阵列

前面章节介绍了用于数据操作的算术和时序电路。数字系统还需要存储器（memory）来存储使用过的数据和生成数据。用触发器组成的寄存器是一种存储少量数据的存储器。本节将会介绍可以有效存储大量数据的存储器阵列（memory array）。

本节将首先概述所有存储器阵列的一般特性，之后介绍三种类型的存储器阵列：动态随机存储器（DRAM），静态随机存储器（SRAM）和只读存储器（ROM）。每一种存储器以不同的方式存储数据。本小节还将简要讨论面积和延迟的折中，并说明使用存储器阵列不仅可以存储数据，还可以执行一些逻辑功能。最后以存储器阵列的HDL代码结束。

5.5.1 概述

图5-38是存储器阵列的一般电路符号。存储器由一个二维存储器单元阵列构成。存储器可以读取或者写入内容到阵列中的一行。这一行由地址（address）指定。读出或者写入的值称为数据（data）。一个有N位地址和M位数据的阵列就有2^N行和M列。每一行数据称为一个字（word）。因此，阵列包含了2^N个M位字。

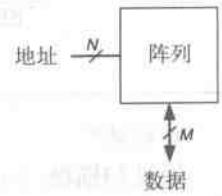


图5-38 一般存储器阵列电路符号

图5-39为一个有两位地址和三位数据的存储器阵列。两位地址指明了阵列中4行的哪一行（数据字）。每一个数据字有3位宽。图5-39(b)显示了存储器阵列中可能的内容。

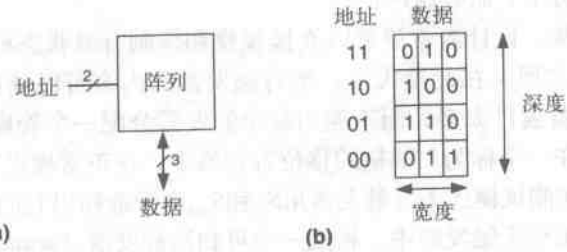


图5-39 4×3存储器阵列：(a) 电路符号；(b) 功能

阵列的深度（depth）是行数，宽度（width）是列数，也称为字大小。阵列的大小就是深度×宽度。图5-39为一个4字×3位的阵列，简称4×3阵列。1024字×32位阵列的符号如图5-40所示。此阵列的容量为32千位（Kb）。

位单元

存储器阵列以位单元（bit cell）构成的阵列组成，其中每个位单元存储1位数据。图5-41中每一个位单元与特定字线（wordline）和特定位线（wordline）相连。对于每一个地址位的组合，存储器将字线设置为高电平，并激活此行中的位单元。当字线为高电平时，就从位线传出或传入要存储的位。否则，位线就会与位单元断开。存储位的电路因存储器类型的不同而异。

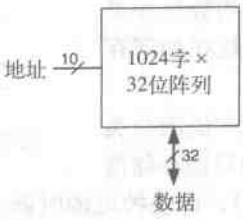


图5-40 32Kb阵列：深度 = 2^{10} = 1024字，宽度 = 32位

图5-41 位单元

读取位单元时，位线初始化为浮空（Z）。随后，字线转换为高电平，允许已存储的值驱动位线为0或者1。写入位单元时，位线被驱动到要写入的值。随后，字线转换为高电平，连接存储位到位线。强驱动的位线将改写位单的内容，向存储位写入要写入的值。

存储器组织

图5-42为4×3存储器阵列的内部组成。当然，实际的存储器会更大，但是大型阵列的操作与小型阵列类似。在这个例子中，阵列存储了图5-39(b) 中的数据。

读存储器时，一条字线设为高电平，相应行的位单元驱动位线为高电平或者低电平。写存储器时，首先驱动位线为高或者低电平，随后字线设置为高电平，允许位线的值存储到相应行的位单元中。例如，要读取地址10，位线首先浮空，译码器设置字线$_2$为高电平，相应行位单元存储的数据100被读出到数据位线。要写入001到地址11，位线首先被驱动到001值，然后字线$_3$被设置为高电平，随后新值001就被存储到位单元中。

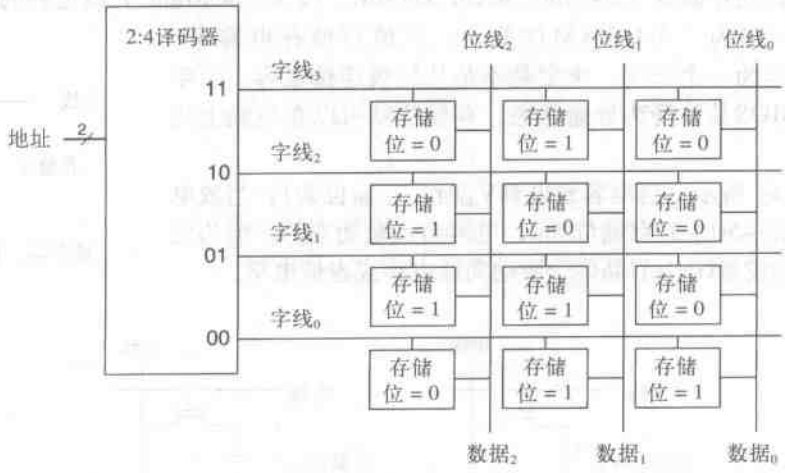


图5-42 4×3 存储器阵列

存储器端口

所有存储器都有一个或者多个端口（port）。每一个端口提供对一个存储器地址的读/写访

问。前面的例子都是单端口存储器。

多端口（multiported）存储器可以同时支持对多个地址的访问。图5-43为一个3端口存储器，其中有两个读端口和一个写端口。端口1根据地址A1读出数据到数据输出RD1，端口2根据地址A2读数据到RD2。当写使能WE3在时钟上升沿有效时，端口3将数据输入WD3的内容写到地址A3指定的行中。

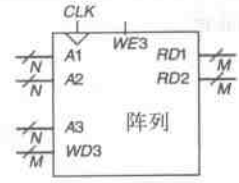


图5-43 三端口存储器

存储器类型

存储器阵列的规格包括容量（深度×宽度）、端口的数目和类型。所有的存储器阵列都以位单元阵列存储数据，但是在如何存储上却各有不同。

存储器可以根据如何在位单元上存储位来分类。最广泛的分类是随机访问存储器（Random access memory, RAM）和只读存储器（Read only memory, ROM）。RAM是易失的（volatile），即关掉电源时就会丢失数据。ROM是非易失的（non-volatile），即没有电源时也可以保存数据。

RAM和ROM因为一些历史的原因获得现在的名字，但是现在也不再有意义了。RAM之所以称为随机访问存储器，是因为访问任何数据字的延迟都相同。与之相对，顺序访问存储器（如磁带）获得临近数据会比获得相距较远数据（例如磁带另一端的数据）更快。ROM之所以称为只读存储器，因为在历史上它只能读，而不能被写入。这些名字容易让人混淆，因为ROM也是随机访问的。更糟糕的是，现在大部分ROM可以读也可以写。RAM和ROM的最重要区别是：RAM是易失的，ROM是非易失的。

RAM的两种主要类型包括：动态RAM（dynamic RAM, DRAM）和静态RAM（static RAM, SRAM）。动态RAM以电容充放电存储数据，静态RAM使用交叉耦合的反向器对存储。对于ROM而言，可以根据擦写方式的不同来区分出很多不同种类。这些不同类型的存储器会在后面小节中讨论。

5.5.2 动态随机访问存储器

动态随机访问存储器（Dynamic RAM, DRAM，读作“dea-ram”）以电容的充电和放电来存储位。图5-44为一个DRAM位单元。位值存储在电容中。nMOS晶体管作为一个开关，决定是不是从位线连接电容。当字线有效时，nMOS晶体管为导通状态，存储位就可以在位线上传入和传出。

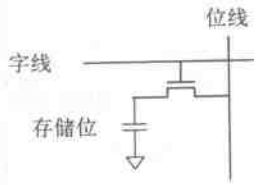


图5-44 DRAM位单元

如图5-45(a) 所示，当电容充电到V_{dd}时，存储位为1；当放电到GND时（图5-45(b)）存储位为0。电容节点是动态的，因为它不由钳制到V_{DD}或者GND的晶体管驱动为高电平或者低电平。

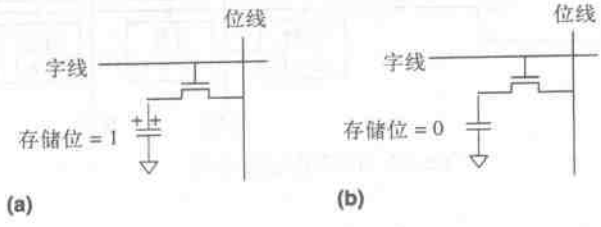


图5-45 DRAM存储值

读操作时，数据值从电容传递到位线。写操作时，数据值从位线传输到电容。读会破坏存储在电容中的位值，所以在每次读后需要恢复（重写）数据。即使DRAM没有被读，电容的电压也会慢慢泄漏，其内容也必须在若干毫秒内刷新（读，然后重写）。

5.5.3 静态随机访问存储器

静态随机访问存储器（static RAM, SRAM，读作“es-ram”）不需要刷新存储位。图5-46是一个SRAM位单元。数据位存储在3.2小节中所述的交叉耦合反向器中。每个单元有两个输出bitline和/bitline。当字线有效时，两个nMOS晶体管都打开，数据值就从位线上传出传入。和DRAM不同，如果噪声减弱了存储位的值，交叉耦合反向器会恢复存储值。

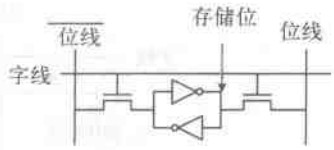


图5-46 SRAM位单元

5.5.4 面积和延迟

触发器、SRAM和DRAM都是易失性存储器，但是有各自不同的面积和延迟特性。表5-4比较了这三种易失性存储器。对于触发器而言，可以通过其输出直接访问存储的数据，但它至少需要20个晶体管构成。总的来说，晶体管数目越多的器件需要芯片面积更大，功耗和成本更高。DRAM延迟比SRAM更长，因为它的位线不是以晶体管驱动。DRAM必须等待充电，从电容将值移动到位线的速度较慢。DRAM的吞吐量也更低，因为它必须周期性地在读取之后刷新。

存储器延迟和吞吐量也与其规模有关；在其他条件的一致情况下，大容量存储器一般比小容量存储器更慢。对于一个特定设计，最好的存储器选择依赖于其速度、成本和功耗约束。

表5-4 存储器比较

存储器类型	每单元晶体管数	延迟
触发器	~20	快
SRAM	6	中等
DRAM	1	慢

5.5.5 寄存器文件

数字系统通常由一组寄存器去存储临时变量。这组寄存器称为寄存器文件（register file），通常由小型多端口SRAM阵列组成，其存储密度较触发器阵列更高。

图5-47是一个32寄存器×32位的3端口寄存器组，由与图5-43相似的3端口存储器组成。寄存器组有两个读端口（A1/RD1和A2/RD2）和一个写端口（A3/WD3）。地址线A1，A2和A3均为5位，可以访问所有的$2^5 = 32$个寄存器，可以同时读两个寄存器和写一个寄存器。

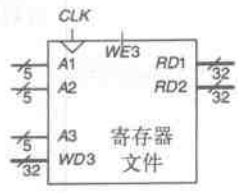


图5-47 有两个读端口和一个写端口的32×32寄存器文件

5.5.6 只读存储器

只读存储器（read only memory, ROM）以晶体管的存在与否来储存一个位。图5-48是一个简单的ROM位单元。读这个单元时，位线被缓慢地推至高电平。随后字线设置为有效。如果晶体管存在，它会使位线为低电平。如果它不存在，位线会保持高电平。注意到ROM的位

单元是组合电路，在无电源的情况下没有可以“忘记”的状态。

ROM的内容可以用点表示法描述。图5-49中，用点表示法描述了包含图5-39中数据的4字×3位ROM。在行（字线）和列（位线）交叉点中的点表示此数据位为1。例如，顶端字线在$Data_1$上有一个点，所以地址11中存储的数据字为010。

图5-48 包含0和1的ROM位单元

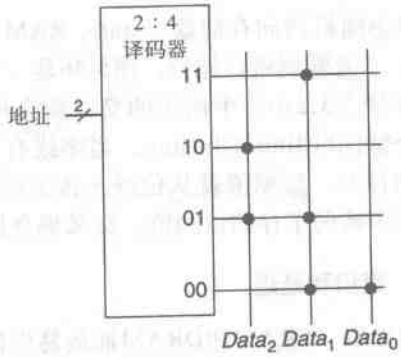


图5-49 4×3的ROM：点表示法

理论上，ROM可以由一组OR门电路后跟随一组AND门电路的2层逻辑组成。AND门电路产生所有可能的最小项，从而形成一个译码器。图5-50为用译码器和OR门电路组成图5-49的ROM。图5-49中每个有点的行就是图5-50中OR门电路的输入。对于只有一个点的数据位（如$Data_0$），就不需要OR门电路。这种ROM的表示方式很有趣，因为它表示了ROM如何执行任意两层逻辑的功能。实践中，ROM用晶体管而不是逻辑门器件组成，以节省面积和成本。5.6.3小节将深入探讨晶体管层的实现。

在制造时，图5-48中ROM位单元的内容可以用每一位单元中晶体管的有无来确定。可编程ROM（programable ROM, PROM），在每一个位单元都放置一个晶体管，然后提供方法决定晶体管是否接地。

图5-51为熔丝型可编程ROM（fuse-programable ROM）的位单元。使用者有选择地提供高压用于熔断熔丝，从而对ROM编程。如果熔丝存在，晶体管就接地，单元保持0。如果熔丝熔断，晶体管就与地断开，单元保持1。因为熔丝在熔断后就不能恢复了，它也称作一次可编程ROM。

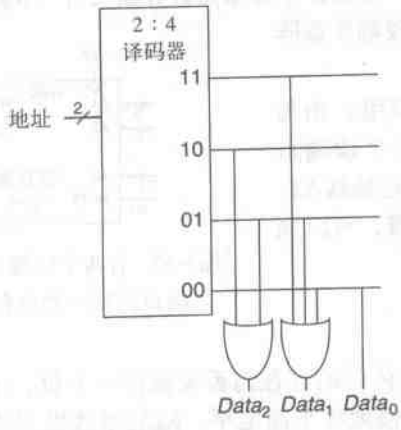


图5-50 使用门器件的实现4×3ROM

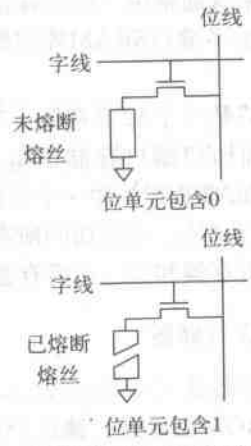


图5-51 熔丝型可编程ROM的位单元

可重复编程ROM提供一种可修改机制来确定晶体管是否连接地。可擦写PROM（Erasable PROMs, EPROMs，读作“e-proms”）把nMOS晶体管和熔丝替换成浮动栅晶体管。浮动栅并不与任何的线物理连接。当合适的高电平出现时，将产生从绝缘体到浮动栅的电子沟道，从而开启晶体管，把位线连接到字线（译码器的输出）。当EPROM暴露在强烈的紫外线中大约半小时，电子就会从浮动栅中移走，从而关闭晶体管。这两个过程分别被称为编程（programming）和擦除(erasing)。电子可擦写PROMs（Electrically Erasable PROMs, EEPROMs，读作“e-e-proms”或者“double-e-proms”）和闪存（flash）采用相似的工作原理，但是它们在芯片上由电路负责擦写，而不需要紫外线。EEPROM的位单元可单独擦写；闪存擦写更大的位块，所需要的擦写电路更少，价格更便宜。2006年，每GB闪存的价格仅为25美金，而且还以每年30%～40%的速度持续下降。闪存广泛应用于便携式电池供电系统（如数码相机和音乐播放器）中存储大量数据。

总体而言，现代的ROM不再只读，它们也可以写入。RAM和ROM的不同在于ROM的写入时间更长，但它是非易失性的。

5.5.7 使用存储器阵列的逻辑

尽管存储器阵列最初用于存储数据，但也可用于实现组合逻辑功能。例如，图5-49中ROM的输出$Data_2$实现了两位地址输入的XOR逻辑。同样，$Data_0$是两位地址输入的NAND逻辑。一个2^N字 × M位存储器可以实现任何N输入和M输出的组合逻辑功能。例如，图5-49的ROM实现两位输入的3种逻辑功能。

用于执行逻辑的存储阵列称为查找表（lookup tables, LUTs）。图5-52的一个4字 × 1位存储阵列可以执行Y = AB函数的查找表。用存储器实现逻辑时，用户可以根据给出的输入组合（地址）查找输出值。每个地址对应真值表的一行，每个数据位对应一个输出值。

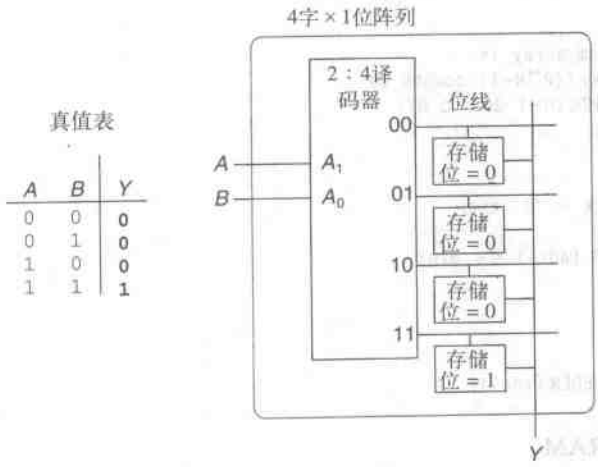


图5-52 用作查找表的4字 × 1位存储器阵列

5.5.8 存储器的硬件描述语言

HDL例子5.7描述一个2^N字 × M位的RAM。RAM有一个同步写使能。换言之，当写使能we有效时，在时钟的上升沿就会发生写入。读则可以立刻得到结果。刚刚加电时，RAM的内容不可预知。

HDL例子5.8描述了一个4字 × 3位ROM。ROM的内容在HDL的case语句中说明。像这样

小的ROM应该可能被综合成逻辑门电路而不是阵列。注意：HDL例子4.25的7段数码管译码器综合为图4-22的ROM。

HDL例子5.7 RAM

Verilog

```
module ram # (parameter N = 6, M = 32)
             (input           clk,
              input           we,
              input  [N-1:0]  adr,
              input  [M-1:0]  din,
              output [M-1:0]  dout);

  reg [M-1:0] mem [2**N-1:0];

  always @ (posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_array is
  generic (N: integer := 6; M: integer := 32);
  port (clk,
        we:   in  STD_LOGIC;
        adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
        din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
        dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array((2**N-1) downto 0)
       of STD_LOGIC_VECTOR(M-1 downto 0);
  signal mem: mem_array;
begin
  process (clk) begin
    if clk' event and clk = '1' then
      if we = '1' then
        mem (CONV_INTEGER (adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem (CONV_INTEGER(adr));
end;
```

图5-53给出综合后的RAM。

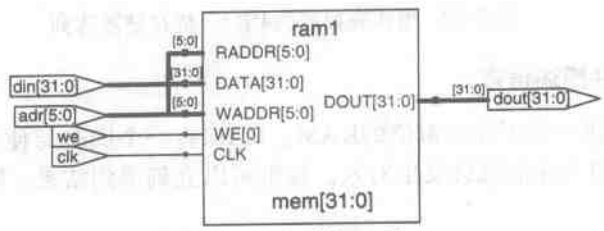


图5-53 综合后的RAM

HDL例子5.8 ROM

Verilog

```
module rom (input      [1:0] adr,
            output reg [2:0] dout);

  always @ (adr)
    case (adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port (adr:  in  STD_LOGIC_VECTOR(1 downto 0);
        dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process (adr) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

5.6 逻辑阵列

和存储器一样，门器件也可以组织成规整的阵列。如果门之间的连接可以编程，那么这些逻辑阵列（logic array）就可以配置执行任何功能，而不需要使用者以特定方式连线。规整的结构可以简化设计。逻辑阵列可以大量生产，所以其并不昂贵。软件工具允许用户将逻辑设计映射到阵列上。大部分的逻辑阵列是可重配置的，这允许设计者不需要替换硬件就可以修改设计。在开发过程中，可重配置的能力很有价值的，而且在使用现场中也很有用，使得简单下载新配置后就可以升级系统。

本节介绍两种类型的逻辑阵列：可编程逻辑阵列（Programmbale Logic Arrays, PLAs）和现场可编程门阵列（Field Programmable Gate Arrays, FPGAs）。可编程逻辑阵列是一个相对旧的技术，它只能实现组合逻辑。FPGA可以实现组合和时序逻辑。

5.6.1 可编程逻辑阵列

可编程逻辑阵列PLA以积之和的形式实现两级组合逻辑。如图5-54所示，PLA由一个AND阵列和跟随它的OR阵列组成。输入（以真值和取反的形式）驱动AND阵列。它产生的蕴涵项依次作OR运算而形成输出。一个$M \times N \times P$位的PLA有M位输入，N位蕴涵项和P位输出。

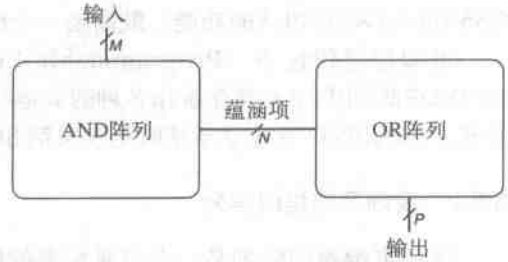


图5-54 $M \times N \times P$位PLA

图5-55所示是$3 \times 3 \times 2$位PLA实现函数$X = \overline{A}\overline{B}C + AB\overline{C}$和$Y = A\overline{B}$的点号表示法。其中，

AND阵列的每一行组成一个蕴涵项。AND阵列中每一行的点表示哪些输入组成蕴涵项。图5-55中的AND阵列形成了3个蕴涵项 $\overline{A}\overline{B}C$， $AB\overline{C}$， $A\overline{B}$。OR阵列的点说明输出函数中包含了哪些蕴涵项。

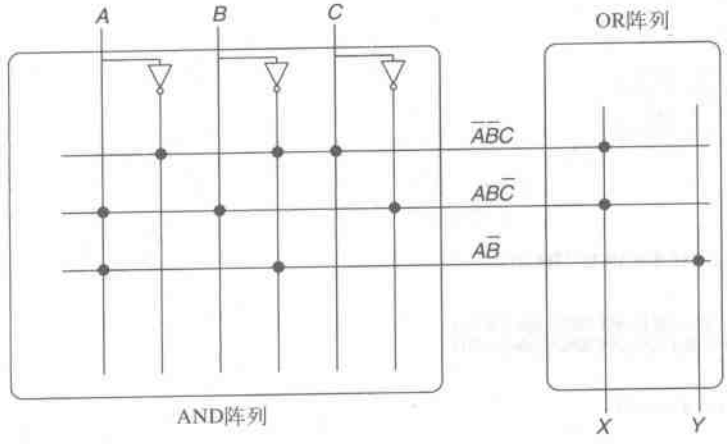


图5-55 3×3×2位PLA：点号表示法

图5-56所示为如何使用两层逻辑组成PLA。另一种的实现将在5.6.3节介绍。

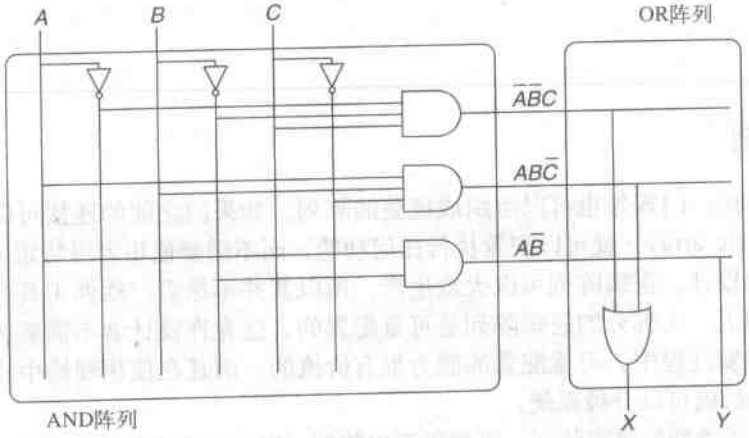


图5-56 使用用两层逻辑构成的3×3×2位PLA

ROM可以看做PLA的一种特殊情况。一个2^M字×N位的ROM就是一个$M\times 2^M\times N$位的PLA。译码器像AND阵列一样，产生所有2^M个最小项。ROM阵列像OR阵列一样，产生所有输出。如果函数不需要依赖所有的2^M个最小项，PLA就会比ROM要小。例如，要实现用图5-55和5.56中3×3×2位PLA的功能，就需要一个8字×2位ROM。

可编程逻辑设备（Programmable Logic Device，PLD）增强了PLA的功能，其中在AND/OR阵列中加入寄存器和各种的其他特殊功能。然而，很大一部分PLD和PLA将被FPGA替代，因为FPGA在建立系统时更加灵活和高效。

5.6.2 现场可编程门阵列

现场可编程门阵列是一个可重配置的门器件阵列。通过软件编程工具，使用者可以用硬

件描述语言或者原理图在FPGA上完成设计。FPGA比PLA更灵活，功能也更强大。FPGA可以实现组合逻辑和时序逻辑，还可以实现多级逻辑功能，而PLA只能实现两级逻辑。现代的FPGA还集成了其他有用的功能，如内部乘法器和大型RAM阵列。

FPGA由可配置逻辑单元（Configurable Logic Block, CLB）阵列构成。图5-57为Xilinx公司1998年发布的Spartan FPGA的结构图。其中，每个CLB可以配置为实现组合或者时序功能。CLB被与外部接口的输入输出单元（Input/Output Block, IOB）包围。IOB连接了CLB输入和输出和芯片封装的管脚。通过可编程的布线通道将CLB和其他CLB、IOB连接在一起。图中的其他单元为设备编程提供帮助。

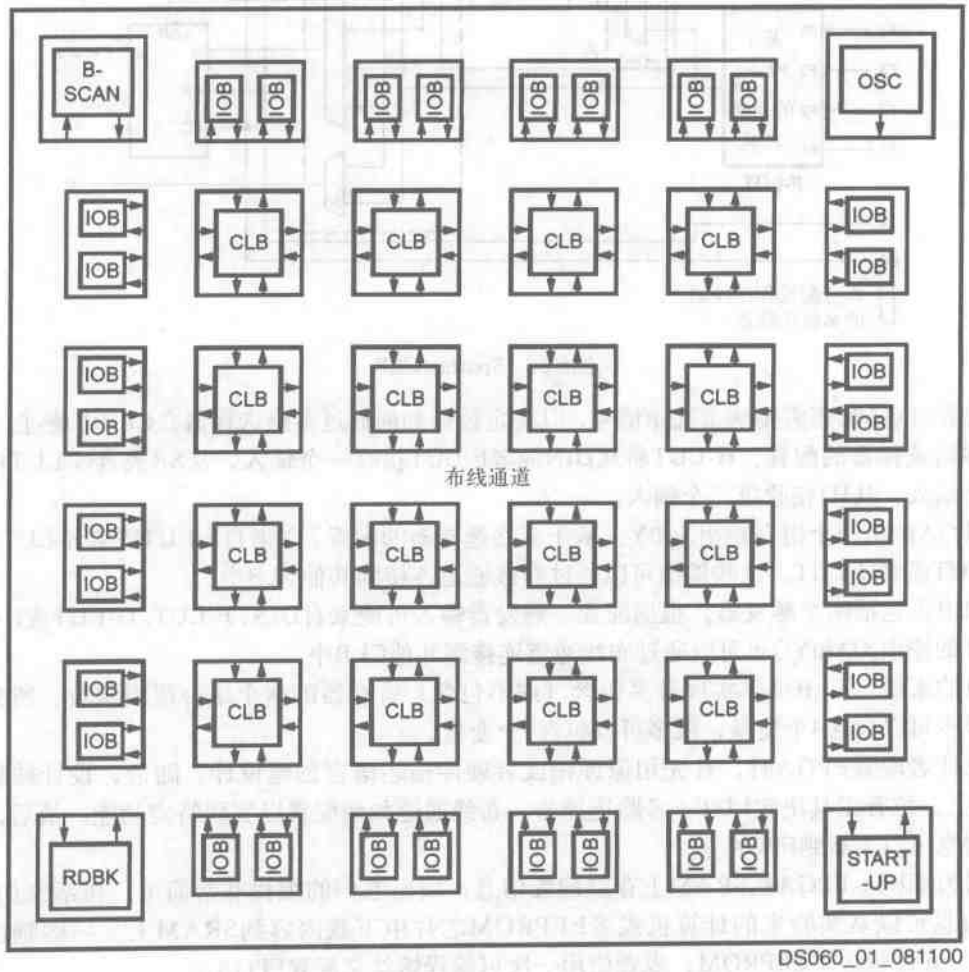


图5-57 Spartan结构图

图5-58为Spartan FPGA中的一个CLB。其他品牌的FPGA可能在组成上有些差异，但都遵循相同的设计原则。CLB由查找表（LUT）、可配置多路选择器、寄存器等构成。通过确定查找表的内容和多路选择器的选择信号来配置FPGA。

每一个Spartan CLB有3个LUT：4输入的F和G，3输入的H。为查找表加载合适的值后，F-LUT和G-LUT可以各自配置以实现所有可能的4变量逻辑函数，H-LUT可以实现所有可能的3变量逻辑函数。

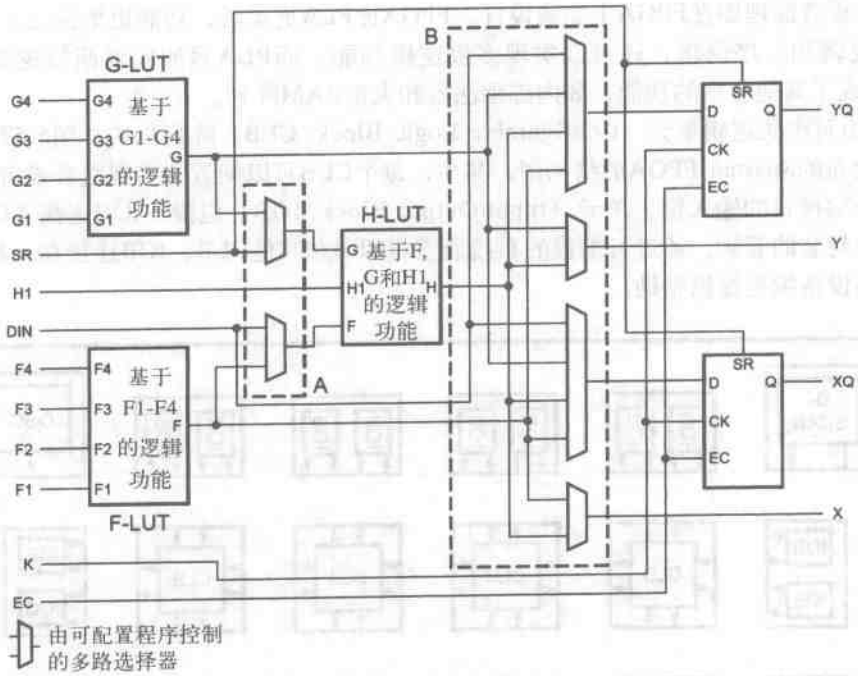


图5-58 Spartan CLB

配置FPGA时还需要确定选择信号，以决定数据如何通过多路选择器在CLB中流过。例如，依靠多路选择器的配置，H-LUT将从DIN或者F-LUT接收一个输入，从SR或者G-LUT中接受第二个输入，从H1接收第三个输入。

FPGA产生两个组合输出X和Y。基于多路选择器的配置，X来自F-LUT或者H-LUT。Y来自G-LUT或者H-LUT。这些输出可以通过布线通道连接到其他CLB中。

CLB还包括两个触发器。根据配置，触发器输入可能来自DIN, F-LUT, G-LUT或H-LUT。触发器的输出XQ和YQ也可以通过布线通道连接到其他CLB中。

总的来说，CLB可以实现最多包含（或不包含）寄存器的两个组合逻辑函数。所有逻辑函数至少可以包含4个变量，最多可以包含9个变量。

设计者配置FPGA时，首先用原理图或者硬件描述语言创建设计。随后，设计被综合到FPGA上。综合工具决定LUT、多路选择器、布线通道如何配置以实现特定功能。最后，这些配置信息可以下载到FPGA上。

因为Xilinx FPGA在SRAM上存储配置信息，所以它们的编程非常简单。在系统加电时，配置信息可以从实验室的计算机或者EEPROM芯片中下载内容到SRAM上。一些制造商的FPGA上直接包含了EEPROM，或者使用一次可编程熔丝来配置FPGA。

例5.6 使用CLB实现特定功能

解释如何配置CLB实现以下功能：

(a) $X = \overline{A}BC + AB\overline{C}$ 和 $Y = A\overline{B}$；

(b) Y = JKLMPQR；

(c) 二进制状态编码的模3计数器（参见图3-29（a））。

解：(a) 配置F-LUT计算X，G-LUT计算Y，如图5-59所示。输入F3, F2, F1分别代表A, B, C（这些连接由布线通道设置）。输入G2, G1为A和B。我们并不关心F4, G4, G3（也许连接到0）。

配置最后一级多路选择器从F-LUT上选择X，从G-LUT上选择Y。一般而言，一个CLB可以以这种方式计算任何两个4变量逻辑函数。

F4	(A) F3	(B) F2	(C) F1	(X) F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

G4	G3	(A) G2	(B) G1	(Y) G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0

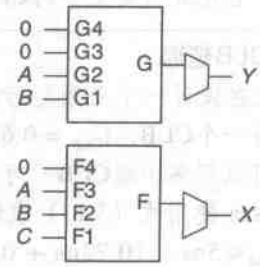


图5-59 用一个CLB配置两个4变量函数

(b) 配置F-LUT来计算F=JKLM，G-LUT计算G=PQR。之后配置H-LUT计算H=FG。配置最后一级多路选择器从H-LUT中选择Y，配置如图5-60所示。一般而言，一个CLB可以以这种方式计算一个9变量的函数。

(J) F4	(K) F3	(L) F2	(M) F1	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

G4	(P) G3	(Q) G2	(R) G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

H1	F	G	(Y) H
X	0	0	0
X	0	1	0
X	1	0	0
X	1	1	1

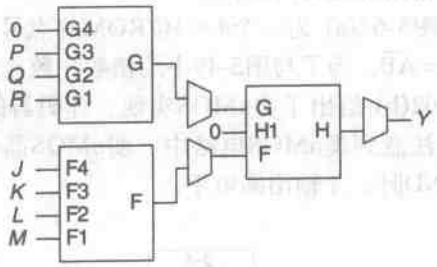


图5-60 用一个CLB配置多于四输入的函数

(c) FSM有两位状态（$S_{1:0}$）和一个输出(Y)。下一状态基于两位的当前状态。使用F-LUT和G-LUT从当前状态计算下一状态，如图5-61所示。使用2个触发器保持状态。触发器使用CLB的SR信号作为专门的复位输入。以虚线表示寄存器的输出反馈回输入的布线通道。一般来说，需要另一个CLB来计算输出Y。然而，在这个例子中，$Y=S'_0$。所以Y可以来自同一个

F4	F3	(S_1) F2	(S_0) F1	(S_0') F
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

G4	G3	(S_1) G2	(S_0) G1	(S_1') G
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

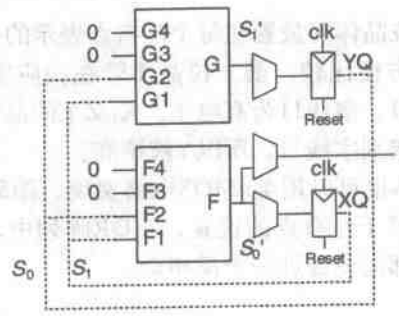


图5-61 两位状态有限状态机的CLB配置

用于计算S'_0的F-LUT。因此，整个FSM可以放在一个CLB中。总的来说，一个FSM中每两位状态需要至少一个CLB。如果用于计算输出或者下一状态的逻辑非常复杂而不能放在一个LUT中的话，它也许需要更多的CLB。

例5.7 CLB延迟

Alyssa正在设计一个必须运行在200MHz的有限状态机。她使用了Spartan3 FPGA，规格如下：对于每一个CLB，$t_{CLB} = 0.61ns$，对于所有的触发器，$t_{setup} = 0.53ns$，$t_{pcq} = 0.72ns$。她的设计中最多可以用多少级CLB？可以忽略内部连接延迟。

解：Alyssa使用式（3.13）求解处逻辑的最大传输延迟$t_{pd} \leqslant T_c - (t_{pcq} + t_{setup})$。

因此，$t_{pd} \leqslant 5ns - (0.72ns + 0.53ns)$，所以$t_{pd} \leqslant 3.75ns$。每一个CLB的延迟$t_{CLB}$为0.61ns，CLB的最大级数$N$需要满足$Nt_{CLB} \leqslant 3.75ns$，因此$N = 6$。

5.6.3 阵列实现*

为了减少尺寸和成本，ROM和PLA一般使用类nMOS或者动态电路（参见1.7.8小节）而不是常用的逻辑门器件。

图5-62(a)为一个4×3位ROM的点号表示法。它实现以下函数：$X = A \oplus B$，$Y = \bar{A} + B$，和$Z = \overline{AB}$。为了与图5-49中的函数一致，地址输入重命名为A和B，数据输出重命名为X, Y, Z。图5-62(b)给出了类nMOS实现。译码器的每一个输出连接到它这一行nMOS晶体管的门控端上。注意到类nMOS电路中，弱pMOS晶体管仅仅只在没有通过下拉(nMOS)网络的路径连接到GND时，才输出高电平。

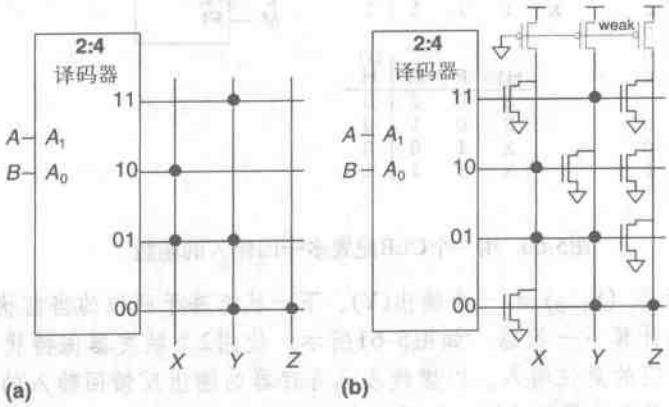


图5-62 ROM的实现：(a) 点号表示法；(b) 伪nMMOS电路

下拉晶体管放置在每个没有点表示的交汇点上。图5-62(a)中的点留在了图5-62(b)中，这样可以方便比较。弱上拉晶体管在相应字线中没有下拉晶体管时将输出拉到高电平。例如，当AB=11，字线11为高电平。X, Z上的晶体管就被打开，把输出拉为低电平。Y输出上没有晶体管连接到字线11，所以Y被拉高。

PLA也可以用类nMOS电路实现，图5-63为图5-55中PLA的实现。在AND阵列中，下拉晶体管放置于没有点的位置；在OR阵列中，则放置在有点的行。在反馈到输出位前，OR阵列的每位都需要经过一个反相器。

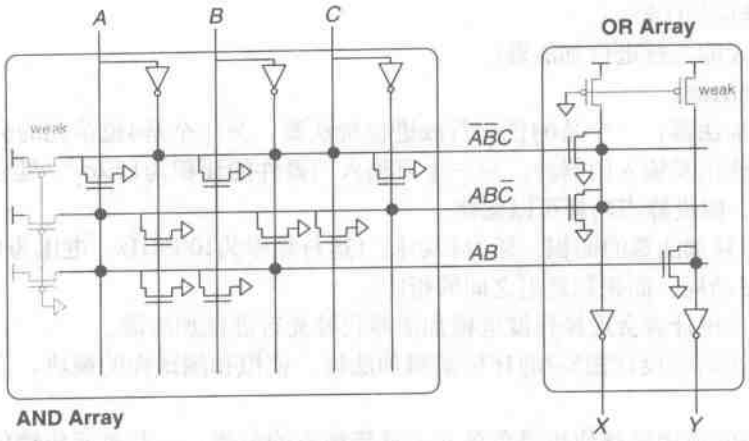


图5-63 用类nMOS电路实现的3×3×2位PLA

5.7 总结

本章介绍了数字系统中常用的电路模块，包括：加法器、减法器、比较器、移位器、乘法器和除法器等算术电路；累加器和移位寄存器等时序电路；存储器阵列和逻辑阵列。这一章还探讨了小数的定点和浮点表示法。第7章中将使用单元电路来构造微处理器。

加法器是大部分算术电路的基础。半加法器把两个1位输入A和B相加，产生一位及一位进位。全加器扩充了半加器以接收进位输入。N个全加器可以级联组成一个进位传输加法器（CPA），以实现两个N位数的加法。因为进位要逐次通过每一个全加器，这一类CPA称为行波进位加法器。更快的CPA可以使用先行进位或前缀技术。

减法器把减数变为负数，把它和被减数相加。数值比较器把一个数和另一个数相减，根据结果的符号判断它们的关系。乘法器使用与门电路形成中间结果，并把这些中间结果使用加法器相加。除法器重复地从中间余数中减去除数，并检查差值的符号，以决定商位。计数器使用加法器和寄存器来递增当前计数。

使用定点或者浮点形式表示小数。定点数与十进制数相似，浮点数与科学计数法相似。定点数使用一般的算术电路，而浮点数要求更多复杂的硬件提取和处理符号、阶码和尾数。

大型的存储器按字阵列方式组织。存储器有一个或者更多的端口完成字的读和（或）写。掉电时，易失性存储器（如SRAM和DRAM）就会丢失它们的状态。SRAM比DRAM更快，但是需要更多的晶体管。寄存器组是小型的多端口SRAM阵列。非易失性存储器（ROM）可以无限长地保持它们的状态。尽管名字为ROM，但是现代大部分的ROM也可以写入。

可以采用规则的阵列来构成逻辑。存储器阵列可以用作查找表，实现组合函数。PLA由可配置的AND和OR阵列连接组成。它们只能实现组合逻辑。FPGA由大量的小型查找表和寄存器组成，可以实现组合逻辑和时序逻辑。查找表的内容和它们的内部连接可以被配置，用于执行任意逻辑函数。现代FPGA能很简单地重编程，具有足够大的容量和便宜的价格来构造专用数字系统，广泛地用于中小型的商业和教育产品中。

习题

5.1 以下三种64位加法器的延迟是多少？假设每一个双输入的门器件延迟是150ps，全加器的延迟是450ps。

(a) 行波进位加法器；
(b) 4位单元的先行进位加法器；
(c) 前缀加法器。

5.2 设计两个加法器：一个是64位的行波进位加法器，另一个是4位单元的64位先行进位加法器。只使用双输入门器件。每一个双输入门器件的面积为15μm²，延迟为50ps，门电容为20fF。假设静态功耗可以忽略。
(a) 比较两种加法器的面积、延迟和功耗（运行频率为100MHz，电压为1.2V）；
(b) 讨论在功耗、面积和延迟之间的折中。

5.3 讨论为什么设计者会选择行波进位加法器代替先行进位加法器。

5.4 用硬件描述语言设计图5-7的16位前缀加法器。模拟和测试你的模块，以证明它能正确运行。

5.5 图5-7所示的前缀网格使用黑色单元去计算所有的前缀。一些单元传输信号并不是必须的。设计一个灰色单元从位i：k和k-1：j接收G和P信号，但是仅产生$G_{i:j}$，而不产生$P_{i:j}$。重画前缀网格，在需要的地方把黑色单元替换为灰色单元。

5.6 如图5-7所示的前缀网格不是在对数时间内计算所有前缀的唯一方法。Kogge-Stone网格是另外一个常见的前缀网格，它实现一样的功能，但使用不同的黑色单元连接。研究Kogge-Stone加法器，画和图5-7相似的原理图，表示在Kogge-Stone中的黑色单元连接。

5.7 回想一个N输入的优先级编码器有$\log_2 N$位输出，编码哪一个输入具有最高优先级。（参见习题2.25）。
(a) 设计一个N输入优先级编码器，延迟按N的对数增加。画出电路原理图，以电路模块延迟的形式给出电路的延迟。
(b) 以硬件描述语言编码完成设计。模拟和测试相应模块，证明它能正确地运行。

5.8 设计以下三种的32位数比较器。画出电路原理图。
(a) 不等
(b) 大于
(c) 小于等于

5.9 使用自己习惯用的硬件描述语言设计如图5-15所示的32位ALU。可以用行为模型或者结构模型设计顶层模块。

5.10 为习题5.9的32位ALU加上一位上溢输出。当加法器的结果上溢时，输出为TRUE。否则，就是FALSE。
(a) 为上溢输出写布尔表达式；
(b) 画出上溢电路；
(c) 以硬件描述语言设计修改后的ALU。

5.11 为习题5.9的32位ALU加上一个Zero输出。当Y等于0时，输出是TRUE。

5.12 编写测试程序用于测试习题5.9，5.10，5.11的32位ALU。之后使用它去测试ALU。包括需要的任何测试向量文件。确保测试了足够的分支用例，使人信服ALU的功能是正确的。

5.13 设计一个把32位输入左移两位的移位器。输入和输出都是32位。以文字解释设计，并画出电路图。用自己习惯使用的硬件描述语言实现设计。

5.14 设计一个4位向左和向右的循环移位器。画出电路图，用自己习惯使用的硬件描述语言实现设计。

5.15 使用24个2：1多路选择器设计一个8位的左移位器。移位器包括8位数据输入A，3位移位量$shamt_{2:0}$，8位输出Y。画出电路图。

5.16 解释如何用$N\log_2 N$个2：1多路转换器构造任意N位移位器或者循环移位器。

5.17 图5-64的漏斗移位器（funnel shifter）可以实现任何的N位移位或者循环移位操作。它将$2N$位输入右移k位。输出Y为结果的最低N位。输入的最高N位称为B，最低N位称为C。选择合适的B, C, k，漏斗移位器就可以实现任何类型的移位或者循环移位。在以下情况，使用A, shamt, N描述上这些值：

(a) A逻辑右移shamt位；

(b) A算术右移shamt位

(c) A左移shamt位；

(d) A右循环移位shamt位；

(e) A左循环移位shamt位。

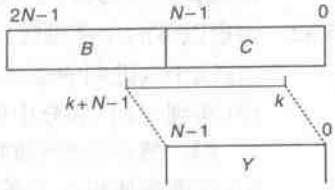


图5-64 漏斗移位器

5.18 找出图5-18中4×4乘法器的关键路径。以AND门器件延迟（t_{AND}）和全加器延迟（t_{FA}）形式给出。同样地，$N \times N$乘法器的延迟是多少。

5.19 设计一个处理二进制补码的乘法器。

5.20 符号扩展单元（sign extension unit）将M位二进制补码输入扩展到N位（$N>M$），其中将复制输入的最高位到输出的高位（参见1.4.6小节）。它接收一个M位的输入A，产生一个N位的输出Y。描绘一个4位输入、8位输入的符号扩展单元。写出设计的HDL代码。

5.21 零扩展单元（zero extension unit）将M位无符号数扩展到N位（$N>M$），其中把零放入输出的高位。描绘一个4位输入、8位输入的零扩展单元。写出设计的HDL代码。

5.22 以二进制计算$111001.000_2/001100.000_2$，使用初中的标准算术除法。

5.23 以下数字系统的数据表示区间分别是多少：

(a) 24位无符号定点数，12整数位和12小数位；

(b) 24位带符号原码定点数，12整数位和12小数位；

(c) 24位2进制补码定点数，12整数位和12小数位。

5.24 以16位定点带符号的原码（其中8位整数位和8位小数位）表示以下十进制数。以十六进制表示结果。

(a) -13.5625

(b) 42.3125

(c) -17.15625

5.25 以16位定点二进制补码形式（其中8位整数位和8位小数位）表示习题5.24的十进制数。以十六进制表示结果。

5.26 以IEEE754单精度浮点数格式表示习题5.24的十进制数。以十六进制表示结果。

5.27 把以下的二进制补码定点数转换成十进制

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

5.28 当两个浮点数相加，为什么阶码较小的数被移位？解释并给出一个例子证明你的解释。

5.29 把以下IEEE754单精度浮点数相加。

(a) C0D20004+72407020

(b) C0D20004+40DC0004

(c) (5FBE4000+3FF80000)+DFDE4000
(为什么结果与预想不同？请解释)

5.30 扩展5.3.2小节中实现浮点数加法的步骤，使它能计算负浮点数。

5.31 考虑IEEE 754单精度浮点数。
(a) IEEE 754单精度浮点数格式能表示多少数字？不需要算入±∞，NaN。
(b) 如果不包含±∞，NaN的表示，另外还可以表示多少个数字？
(c) 解释为什么±∞，NaN用特殊的形式表示。

5.32 考虑以下的十进制数：245和0.0625
(a) 以十六进制的形式写出两个数的单精度浮点表示。
(b) 实现（a）部分中两个32位数字的数值比较。换言之，将两个32位数字为二进制补码，然后比较它们。整数比较给出了正确的答案了码？
(c) 你准备使用一个新的单精度浮点表示法。每一样都和IEEE 754单精度浮点标准一样，除了使用二进制补码表示阶码，而不是二进制数。用你的新标准写出两个数。以十六进制给出答案。
(d) 使用你的新浮点数表示法，用整数比较器可以两个浮点数的大小吗？
(e) 为什么用整数方法比较浮点数进行会相对方便？

5.33 使用读者常用的HDL设计一个单精度浮点加法器。在编码前，画出设计的电路图。模拟和测试加法器，证明它正确运作。你可以只考虑正数，然后使用向零舍入（舍位），也可以忽略表5-2中给出的特殊情况。

5.34 开发一个32位浮点乘法器。乘法器有两个32位浮点输入并产生一个32位浮点输出。你可以只考虑整数，使用向零舍入（舍位）。可以忽略表5-2中给出的特殊情况。
(a) 写出实现32位浮点乘法的步骤；
(b) 画出32位浮点乘法器的电路图；
(c) 以HDL设计一个32位浮点乘法器。模拟和测试乘法器，证明它能正确运行。

5.35 开发一个32位前缀加法器。
(a) 画出电路图；
(b) 以HDL设计32位前缀加法器。模拟和测试加法器，证明它能正确运行；
(c) 部分(a)中32位前缀加法器的延迟是多少？假设每一个双输入门电路的延迟是100ps；
(d) 设计一个32位前缀加法器的流水线版本。画出电路原理图。流水线前缀加法器可以运行得多快？使这个设计尽可能快地运行；
(e) 以HDL设计流水线32位前缀加法器。

5.36 递增器将对N位数字加一。用半加器建立一个8位递增器。

5.37 设计一个32位同步Up/Down计数器。输入信号包括：复位和Up。当复位1为1，输出都是0。否则，当Up为1，电路开始上加，当Up为0，电路开始下加。

5.38 设计一个32位计数器，在每一个时钟沿加4。累加器输入信号包括：复位和时钟。复位时，所有输出为0。

5.39 修改习题5.38的计数器，使它在每个时钟沿可以以4累加，或者加载一个新的32位值D。这些都依赖控制信号PCSrc。当PCSrc=1，累加器就录入新值D。

5.40 N位的Johnson计数器由一个带复位信号的N位移位寄存器组成。移位寄存器的输出S_{out}取反，然后反馈到输入S_{in}。当计数器复位时，所有位被清零。

(a) 给出4位Johnson计数器复位后，产生的输出序列$Q_{3:0}$。

(b) N位Johnson计数器经过多少个周期就会重复出现相同的序列？解释你的结果。

(c) 使用一个5位Johnson计数器，10个AND门器件和反相器器设计一个十进制计数器。十进制计数器有时钟，复位和10个单状态输出$Y_{9:0}$。当计数器复位时，Y0有效。在每一个序列周期中，下一输出将有效入。10个周期之后，计数器必须重复。画出十进制计数器的电路图。

(d) 比起平常的计数器，Johnson计数器有什么优点。

5.41 为一个如图5-37所示的4位可扫描触发器编写HDL。模拟和测试你的HDL模块，证明其能正确运行。

5.42 英语里面有一些冗余，在传播使用时可以避免语义的混淆。二进制数据也有一些冗余的形式，从而可以纠正一些错误。例如，数字0可以编码成00000，数字1必须编码成111111。数字在有噪声的信道中传输，这些数字中最多可能翻转两位。接收器可以重构出原始数据，因为在接收的5位数据中，至少有3位为0，才为0；同样，应该至少有3位为1才是1。

(a) 设计一种编码方式，它使用5位信息编码发送00，01，10或者11。可以纠正1位错误。提示：00000和11111分别为00，11的编码不能成功。

(b) 设计一个电路，接收5位编码数据然后译码成00，01，10，11，即使一位传输数据被更改。

(c) 假设你想改变成另外一种5位编码。如何实现你的设计，可以使改变编码时而不用修改硬件。

5.43 快速EEPROM（简称为闪存）引起了近期消费电子产品的革命。研究和说明闪存如何工作。使用图说明浮空门器件。描述存储器中的位如何被编程。正确注明引用的资料。

5.44 一个地球外生命项目小组刚发现有外星人生活在Mono 湖的底部。需要设计一个电路来判断外星人来自哪个潜在的星球，基于NASA探测器获得它们的外形：绿色、褐色、黏稠的、丑的。小心研究宇宙生物学后，以得出以下结论：

- 如果外星人是绿色而且是黏稠或者丑陋的，褐色而且是黏稠的，它就可能来自火星；
- 如果这个生物是丑的、褐色而且黏稠，或者绿色但是又不黏稠又不丑陋，那么它可能来自金星；
- 如果它是褐色且既不是黏稠又不丑陋，或者是绿色又是黏稠的，它就可能来自木星。

注意到可能产生并不是唯一的结果，例如一个生命形式掺杂了绿色和褐色，黏稠的，但是不丑陋就可能来自火星或者木星。

(a) 编写一个$4\times4\times3$的PLA程序去识别外星人。你可能要用到点号表示法。

(b) 编写一个16×3的ROM去识别外星人。你可能要用到点号表示法。

(c) 用HDL实现你的设计。

5.45 使用单独的16×3 ROM实现以下的函数。使用点号表示法说明ROM的内容。

(a) $X = AB + B\overline{C}D + \overline{A}\overline{B}$

(b) $Y = AB + BD$

(c) $Z = A + B + C + D$

5.46 使用$4\times8\times3$PLA实现习题5.45的函数。你可能要用到点号表示法。

5.47 说明对以下每个组合电路编程时所需要的ROM容量。使用ROM实现这些功能是一个好选择吗？解释为什么。

(a) 带C_{in}和C_{out}的16位加法器/减法器；

(b) 8×8乘法器；

(c) 16位优先级编码器（参见习题2.25）。

5.48 考虑图5-65的ROM电路。对于每一行，是否可以对第II列中的ROM适当编程后实现和第I列电路相同的功能？

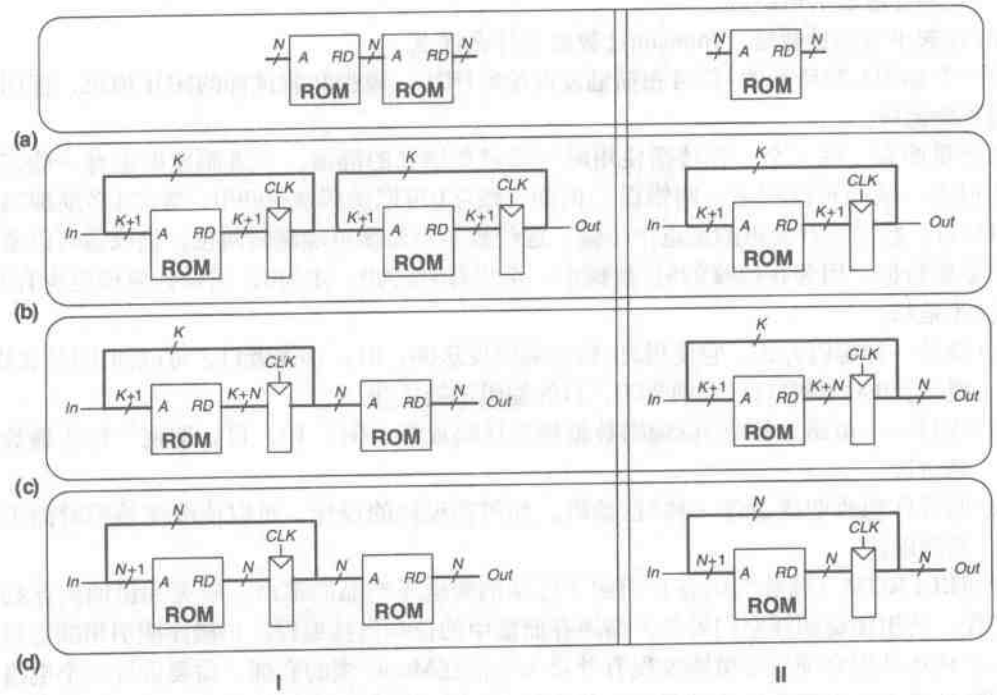


图5-65 ROM 电路

5.49 给出一个用Spartan FPGA CLB实现一个9输入函数的例子。给出不能仅用一个CLB实现的8输入函数例子。

5.50 对于以下函数，需要多少的Spartan FPGA CLB去实现？表示出如何配置一个或者多个CLB去实现函数。应该通过观察完成，而不需要实现逻辑综合。

(a) 习题2.7（c）中的组合函数；

(b) 习题2.9（c）中的组合函数；

(c) 习题2.15中的双输出函数；

(d) 习题2.24中的函数；

(e) 4输入优先级编码器（参见习题2.25）；

(f) 8输入优先级编码器（参见习题2.25）；

(g) 3:8译码器；

(h) 4位进位传播加法器（没有进位出或者进位入）；

(i) 习题3.19的FSM；

(j) 习题3.24中的Gray码计数器。

5.51 考虑图5-58中的Spartan CLB。它的规格如下：每一个CLB 中，$t_{pd} = t_{cd} = 2.7ns$；所有的触发器中，$t_{setup} = 3.9ns$，$t_{hold} = 0ns$，$t_{pcq} = 2.8ns$。

(a) 图3-26的FSM需要至少多少Spartan CLB去实现。

(b) 不计算时钟偏移，这个FSM可靠运行的最快时钟频率是多少？

(c) 时钟偏移是5ns时，这个FSM可靠运行的最快的时钟频率是多少？

5.52 你需要使用一个FPGA去实现一个M&M的分类器。用颜色传感器和电动机把红色的糖放在一个罐里，把绿色的糖放在另一个罐里。这个设计用Spartan 3系列的Spartan XC3S200 FPGA芯片实现FSM。它明显比原始的Spartan FPGA要快。通过数据资料，FPGA有以下的时间特性，如表5-5所示。假设设计足够小，以致线路延迟可以忽略。

表5-5 Spartan 3 XCS200时间参数

名称	值（ns）	名称	值（ns）
t_{pcq}	0.72	t_{pd}(per CLB)	0.61
t_{setup}	0.53	t_{skew}	0
t_{hold}	0		

希望FSM以100MHz运行，其关键路径上CLB最多有多少级？FSM可以运行的最快速度是多少？

面试问题

5.1 两个无符号*N*位数相乘的最大可能结果是多少？

5.2 BCD码（Binary coded decimal）使用4位编码每一个十进制数。例如42_{10}用01000010_{BCD}表示。为什么处理器会使用BCD码？

5.3 设计硬件把两个8位无符号BCD数相加（参见问题5.2）。给出你设计的电路图，并用硬件描述语言为BCD加法器编写一个模块。输入是A, B, C_{in}，输出是S和C_{out}。C_{in}和C_{out}是一位进位，A，B，S是8位BCD数。

第6章 体系结构

6.1 引言

前面章节介绍了数字电路设计的原理，并设计了一些模块。在本章中，我们将进入到新的抽象层次（参见图1.1）来定义计算机的体系结构（architecture）。体系结构是程序员所见到的计算机，它由指令集（汇编语言）和操作空间（寄存器和存储器）来定义。现在有不同类型的体系结构，例如IA-32、MIPS、SPARC和PowerPC等。

理解任何计算机体系结构的第一步是学习它的语言。计算机语言中的单词叫做指令（instruction）。计算机的词汇表叫做指令集（instruction set）。所有在同一计算机上运行的程序使用相同的指令集。即使是非常复杂的软件应用程序（如字处理软件或电子表格软件）也最终编译为一系列诸如加法、减法或跳转的简单指令。计算机指令包含需要完成的操作（operation）和需要使用的操作数（operand）两个部分，其中操作数来源于存储器、寄存器或者指令自身。

计算机硬件仅仅能理解二进制信息，所以指令也将被编码为二进制数，其格式称为机器语言（machine language）。正如使用字母来编码人类的语言，计算机使用二进制数编码机器语言。微处理器是一个可以读入并执行机器语言指令的数字电路系统。因为人类直接阅读二进制格式的机器语言会非常枯燥乏味，所以使用符号形式来表示指令，称为汇编语言（assemble language）。

不同体系结构的指令集更像是不同的方言，而非不同的语言。几乎所有的体系结构都定义了基本指令，例如加法、减法和跳转，操作数也都来源于存储器或寄存器。一旦学习了一个指令集，理解其他指令集就相当简单了。

计算机体系结构并未确定底层的硬件实现。对于一个计算机体系结构，往往会有不同的硬件实现。例如，Intel公司和AMD公司销售的不同处理器都属于相同的IA-32体系结构。它们可以运行相同的程序，但是它们使用了不同的底层硬件实现，可以提供在价格、性能和功耗等方面的各种折中。一些处理器专门对高性能服务器进行优化，而其他处理器可能为了延长笔记本计算机的电池寿命而针对功耗进行优化。寄存器、存储器、ALU和其他模块组织成微处理器的方式称为微结构（microarchitecture），我们将在第7章中讨论这个主题。经常会有针对同一体系结构的不同微结构设计。

在本书中，我们将介绍John Hennessy和他的同事于20世纪80年代在Stanford大学首先提出的MIPS体系结构。MIPS处理器得到了广泛的应用，例如Silicon Graphics, Nintendo和Cisco等公司都采用了这种处理器。我们将从基本指令、操作数地址和机器语言格式开始介绍，随后将介绍在一般程序构造中使用的指令序列，包括分支转移、循环、数组操作和过程调用。

通过本章，我们将看到由Patterson和Hennessy提出的MIPS体系结构设计的四个准则：1）简单设计有助于规整化；2）加快常见功能；3）越小的设计越快；4）好的设计需要好的折中方法。

6.2 汇编语言

汇编语言是计算机机器语言的人类可阅读表示。每条汇编语言指令都指明了需要完成的

操作和操作所处理的操作数。我们通过介绍简单的算术指令来体现如何用汇编语言来写这些操作，接着将定义MIPS指令系统中的操作数：寄存器、存储器和常数。

我们假设读者已经熟悉一种高级程序语言，如C、C++或Java。（实际上这些语言在本章的例子中都几乎差不多，在有所不同的时候，我们将使用C语言。）

6.2.1 指令

最常见的计算操作是加法操作。代码示例6.1给出了如何将两个变量b和c相加将结果写入a中。在左边的是高级语言（C、C++或Java），在右边使用MIPS汇编语言重写。注意C语言程序语句的最后是一个分号。

代码示例6.1 加法

高级语言代码	MIPS汇编语言代码
`a = b + c;`	`add a, b, c`

代码示例6.2 减法

高级语言代码	MIPS汇编语言代码
`a = b - c;`	`sub a, b, c`

汇编指令的第一部分（add）是助记符（mnemonic），指明了需要执行的操作。该操作基于源操作数（source operand）b和c，结果将写入目的操作数（destination operand）a。

代码示例6.2显示减法指令类似于加法。除了操作码sub以外，指令格式完全与加法指令相同。这种一致的指令格式很好地证明了第1条设计准则：

设计原则1：简单设计有助于规整化。

指令中包含固定数目的操作数（在本例中，有2个源操作数和一个目的操作数）将易于编码和硬件处理。更复杂的高级语言代码可以转化为多条MIPS指令，如代码示例6.3所示。

代码示例6.3 复杂代码

高级语言代码	MIPS汇编语言代码
`a = b + c - d;`	`sub t, c, d  # t = c - d` `add a, b, t  # a = b + t`

在高级语言例子中，单行注释以//开始直到一行结束，多行注释则由/*开始，由*/结束。在汇编语言中仅仅支持单行注释，由#开始直到一行结束。代码示例6.3中的汇编语言程序需要一个临时变量t来存储中间结果。使用多条汇编指令执行复杂的操作体现了第2条设计准则：

设计原则2：加快常见功能。

MIPS指令集通过仅仅包含简单、常用指令使得常见功能能够较快执行。指令的数目比较少，这使得用于指令操作和操作数译码的硬件比较简单、精练和快捷。更复杂但不常见的操作由多条简单指令序列执行。因此，MIPS属于精简指令集计算机（Reduced Instruction Set Computer, RISC）体系结构。具有复杂指令的体系结构，例如Intel的IA-32，称为复杂指令集计算机（Complex Instruction Set Computer, CISC）。例如，IA-32中定义的“字符串移动”指令将字符串从内存的一个位置复制到另外一个位置。这样的操作需要很多条（很可能上百条）RISC处理器的指令。然而，CISC体系结构中实现复杂指令的缺点在于增加了硬件而且降低了简单指令的执行速度。

RISC体系结构在简化硬件复杂性的同时，还使得指令编码中区分不同指令操作的位数比较少。例如，有64种简单操作的指令集需要$\log_2 64 = 6$位来编码各种操作。有256种复杂指令的指令集需要$\log_2 256 = 8$位来编码各种指令。在CISC处理器中，即使复杂指令使用的频率非常小，它们也将增加包括简单指令在内的所有指令的开销。

6.2.2 操作数：寄存器、存储器和常数

一条指令的操作需要基于操作数。在代码示例6.1中变量a、b、c都是操作数。但是计算机只能处理二进制，而不能处理变量名称。指令需要从一个物理位置中取出二进制数据。操作数可以存放在寄存器或存储器中，也可以作为常数存储在指令自身中。计算机使用不同的位置存放操作数以优化性能和存储容量。存放在指令中的常数或者在寄存器中的操作数访问非常快，但是它们只能包含少量数据。更多的数据需要从存储器访问得到。存储器虽然容量大但是访问速度比较慢。因为MIPS体系结构操作32位数据，所以被称为32位体系结构（MIPS体系结构的商业产品已经扩展到64位，但是本书仅仅考虑32位格式）。

寄存器

只有快速访问到操作数，指令才能执行比较快。但是存放在存储器中的操作数需要较长时间才能访问到。因此，大多数体系结构定义了一组容量比较小的寄存器用于存放常用的操作数。MIPS体系结构有32个寄存器，称为寄存器集（register set）或寄存器文件（register file）。越少的寄存器访问速度越快，这正验证了第3条设计准则：

设计原则3：越小的设计越快。

从书桌上少量相关的书籍中查找信息要远比从图书馆的书库中找要快很多。同样道理，从一个比较少数量的寄存器（例如32个）中读取数据要快于从1000个寄存器或大的存储器中读取。小的寄存器文件往往利用小SRAM阵列（参见5.5.3节）构造。这样的SRAM阵列中使用较小的译码器和位线连接相对少的存储单元，相对于大的存储器具有更短的关键路径。

代码示例6.4显示了具有寄存器操作数的add指令。MIPS寄存器名称由$符号开始。变量a、b、c存放在$s0、$s1和$s2寄存器中（$s1读做“register s1”或“dollar s1”）。指令将对$s1 (b)和$s2(c) 中存储的32位数据进行加法操作，并将32位结果写入到$s0 (a)。

MIPS有32个寄存器，其中使用18个寄存器存储变量：$s0~$s7，以及$t0~$t9。由$s开头的寄存器称为保存寄存器。根据MIPS的惯例，这些寄存器存储诸如a、b、c等变量。保存寄存器的一个特殊用途是用于过程调用（参见6.4.6节）。以$t开头的寄存器称为临时寄存器，用于存储临时变量。代码示例6.5中显示MIPS汇编代码使用临时寄存器$t0来存储c-d的中间结果。

代码示例6.4 寄存器操作数

高级语言代码

```
a = b + c;
```

MIPS汇编语言代码

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2      # a = b + c
```

代码示例6.5 临时寄存器

高级语言代码

```
a = b + c - d;
```

MIPS汇编语言代码

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d
sub $t0, $s2, $s3      # t = c - d
add $s0, $s1, $t0      # a = b + t
```

例6.1 高级语言代码翻译为汇编语言代码。

将下列高级语言代码翻译为汇编语言码。假设变量a～c存储在寄存器$s0～$s2，变量f～j存储在寄存器$s3～$s7中。

```
a = b - c;
f = (g + h) - (i + j);
```

解： 该程序用了4条汇编语言指令。

```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,
# $s6 = i, $s7 = j
  sub $s0, $s1, $s2   # a = b - c
  add $t0, $s4, $s5   # $t0 = g + h
  add $t1, $s6, $s7   # $t1 = i + j
  sub $s3, $t0, $t1   # f = (g + h) - (i + j)
```

寄存器集

MIPS体系结构定义了32个寄存器。每个寄存器都有一个名字和从0～31的编号。表6-1给出了寄存器的名字、编号和用途。$0始终为0，这是因为常数0经常在计算机程序中使用。我们已经讨论了$s和$t寄存器，其余的寄存器将在本章的后续内容中介绍。

表6-1 MIPS寄存器集

名 称	编 号	用 途	名 称	编 号	用 途
$0	0	常数0	$t8~$t9	24~25	临时变量
$at	1	汇编器临时变量	$k0~$k1	26~27	操作系统临时变量
$v0~$v1	2~3	过程返回值	$gp	28	全局指针
$a0~$a3	4~7	过程参数	$sp	29	堆栈指针
$t0~$t7	8~15	临时变量	$fp	30	框架指针
$s0~$s7	16~23	保存变量	$ra	31	过程返回地址

存储器

如果仅仅以寄存器作为操作数的存储空间，将限制程序中的变量不能超过32个。但是，数据也可以存储在存储器中。与寄存器文件相比，存储器可以存储更多的数据，但是访问数据需要的时间就更长。寄存器容量小、速度快，而存储器容量大、速度慢。所以，经常使用的变量保存在寄存器中。通过综合使用寄存器和存储器，程序可以以相对较快的速度访问大量的数据。如5.5节所述，存储器组织为数据字的阵列。MIPS体系结构采用32位存储器地址和32位数据字长。

MIPS采用字节寻址方式的存储器。也就是说，存储器的每一字节都有一个单独的地址。然而为了便于理解，本节首先讲解字寻址方式的存储器，然后再描述MIPS中字节寻址方式的存储器。

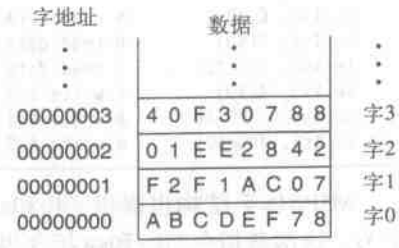


图6-1 字寻址方式的存储器

图6-1给出了字寻址（word-addressable）方式存储器的排列。每32位数据对应唯一的32位地址值，其中32位地址和32位数据用十六进制表示。例如，数据0xF2F1AC07存储在存储器第1号地址空间中。按照惯例，存储器图中的低地址在下，高地址在上。

MIPS使用读取字指令（load word, lw）从存储器中读出数据到寄存器，代码示例6.6表明

了如何将内存中的第1号字读入到$s3寄存器中。

lw指令中的内存有效地址（effective address）为基址（base address）和偏移量（offset）之和。基址为寄存器，写在括号内。偏移量是一个常量，写在括号前。在代码示例6.6中，基地址是总为0值的$0，偏移量是1，所以lw指令从内存中读出的地址为（$0+1）=1。lw指令执行之后，$s3中所存的值为0xF2F1AC07，此值就是图6-1内存中1号地址所存储的值。

代码示例6.6 读字寻址方式存储器

汇编代码

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

类似地，MIPS使用存储字指令（store word, sw）从寄存器向内存写入数据。代码示例6.7将寄存器$s7中的内容写入内存的第5个字中。上述示例简单地将$0作为基地址，然而任意寄存器均可作为基地址。

代码示例6.7 写字寻址方式寄存器

汇编代码

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw   $s7, 5($0)     # write $s7 to memory word 5
```

上述两个代码例子描述了字寻址方式存储器的计算机体系结构。然而MIPS存储器模型却是字节寻址的。每一字节的数据都有一个唯一的地址。一个32位的字包含4个8位字节，所以每一个字地址都是4的倍数，如图6-2所示。其中，32位字地址和数值都是用十六进制表示的。

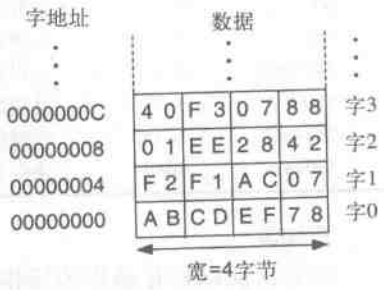


图6-2 字节寻址的存储器

代码示例6.8描述了如何在MIPS的字节寻址存储器中读写一个字。字地址是字号的4倍，MIPS汇编代码读取第0、2和3号字，写入第1，8和100号字。偏移量可用十进制和十六进制表示。

代码示例6.8 字节寻址的存储器访问

MIPS汇编代码

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

MIPS体系结构也提供了lb和sb指令来读出和写入单个字节。这两条指令与lw和sw指令很相似，将在6.4.5节介绍。

如图6-3所示，字节寻址存储器的组织方式有大端（big-endian）和小端（little-endian）两种形式。两种格式中，最高字节（Most Significant Byte，MSB）在左，最低字节（Least Significant Byte，LSB）在右。在大端形式的机器中，第0个字节是从最高字节开始；在小端形式的机器

图6-3 大端和小端存储器寻址

中，第0个字节是从最低字节开始。两种格式的字地址相同，而且并指向相同的4个字节，唯一不同的只是一个字中不同字节的地址。

例6.2 大端和小端存储器

假设$s0初始包含值为0x23456789。在一个大端系统中，后续程序运行结束后$s0的值为多少？如果在一个小端系统中呢？lb $s0, 1（$0）将字节地址为（1+$0）=1的数据存储到$s0的最低字节。其中lb指令如6.4.5节所讨论。

```
sw $s0, 0($0)
lb $s0, 1($0)
```

解：图6-4指出了大端和小端机器中存储器地址为0处如何存储数值0x23456789。在lb指令：lb $s0, 1 （$0） 执行后，大端机器系统中$s0中的数据为0x00000045，而小端机器系统中$s0中的数据为0x00000067。

IBM公司的PowerPC（在Macintosh计算机中采用）使用大端方式寻址。PC中使用的Intel公司IA-32体系结构使用小端方式寻址。一些MIPS处理器使用大端方式，而另一些使用小端方式。选择大端或小端方式完全是任意的，却因此引起了使用大端与小端方式的计算机之间共享数据的麻烦。在本文的例子中，我们在涉及字节顺序时都使用小端方式。

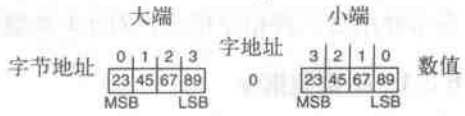


图6-4 大端和小端的数据存储

在MIPS体系结构中，lw和sw的字地址必须是字对齐（word aligned）的，即地址必须能被4 整除。所以指令lw $s0,7（$0） 是非法指令。在一些体系结构中（如IA-32）允许读写非字对齐数据，但在MIPS中，为了简化，要求严格的字对齐。当然，读出字节（lb）和写入字节（sb）指令的字节地址，不需要字对齐。

常数/立即数

读出字（lw）和写入字（sw）指令，说明了MIPS体系结构中常量（constant）的用法。因为它们的值可以被指令立即访问，而不需要通过访问寄存器或存储器得到，所以这些常量叫做立即数（immediate）。立即数加法（addi）指令是一个以立即数为操作数的常见MIPS指令。addi 将指令指定的立即数与某一寄存器中的值相加，如代码示例6.9所示。

代码示例6.9 立即数操作数

高级语言代码

```
a = a + 4;
b = a - 12;
```

MIPS汇编代码

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4          # a = a + 4
addi $s1, $s0, -12        # b = a - 12
```

指令指定的立即数采用16位补码表示，范围在[−32768，32767]之间。减法相当于加了一个负数，因此，为了简单起见，MIPS体系结构中没有subi指令。

回忆前面提到的add和sub指令用三个寄存器作为操作数。但是，lw、sw和addi指令用两个寄存器和一个常数。因为指令格式的差异，lw和sw指令违反了设计原则 1 ：简单设计有助于规整化。然而，这些问题让我们引出介绍最后一个设计原则：

设计原则4：好的设计需要好的折中。

单一的指令格式可能简洁但缺乏弹性。MIPS指令集支持三种指令格式。第一种格式有三

个寄存器操作数，如add和sub指令。第二种格式有两个寄存器操作数和一个16位的立即数操作数，如lw和addi指令。第三种格式有一个26位的立即数，但没有寄存器作为操作数，将会在后文讨论。下一节将讨论三种MIPS指令格式和其二进制代码的编码。

6.3 机器语言

汇编语言方便人们阅读。然而数字电路只能理解0和1，需要将汇编语言写的程序从助记符号翻译成仅使用0和1表示的机器语言（machine language）。

MIPS使用32位指令。这里再次强调，简单设计有助于规整化，最规整的选择是将指令编码成存储在存储器中的字。尽管一些指令可能不需要所有32位的编码，但变长度指令将大大增加复杂性。简单化也鼓励使用单指令格式，但是由于上文提到的原因，过于简单化将产生太多限制。MIPS做了折中，它定义了三种指令格式：R-类型、I-类型和J-类型。少量的指令格式允许一些适用于各种格式的规整设计，因此硬件可以保持简单，也可适用于不同的指令需要（例如，需要在指令中对大常数编码）。R-类型指令对三个寄存器操作。I-类型指令对两个寄存器和一个16位立即数操作。J-类型（跳转）指令对一个26位的立即数操作。我们将在本节介绍这三种指令格式，对于J-类型指令的讨论，将放在6.4.2节。

6.3.1 R-类型指令

R-类型是寄存器类型（register-type）的缩写。R-类型指令有三个寄存器操作数：2个为源操作数，另一个为目的操作数。图6-5给出了R-类型机器指令的格式。32位指令分为6个字段：op，rs，rt，rd，shamt和funct。如图所示，每个字段有包含5到6位。

R-类型

op	rs	rt	rd	shamt	funct
6位	5位	5位	5位	5位	6位

图6-5 R-类型机器指令格式

指令的操作编码为两段：op（也被称为opcode或操作码）和funct（也被称为function）。所有的R-类型指令的操作码都是0，特定操作由funct字段决定。例如add指令的opcode和funct字段分别为0（000000_2）和32（100000_2）。类似地，sub指令的opcode和funct字段为0和34。

指令的操作数编码包括三个字段：rs，rt和rd。前两个寄存器rs和rt是源寄存器；rd是目的寄存器。这些字段包含了图6-1给出的寄存器编号。例如，$s0为寄存器16。

第五个字段shamt仅仅用于移位操作。在这些指令存储在5位shamt字段中的二进制数值表示移位数。对于其他指令，shamt段为0。

图6-6给出了R-类型add和sub指令的机器码。注意，在汇编语言指令中，第一个寄存器是目的地址，而在机器语言指令中，目的地址为第三个寄存器所在的字段（rd）。举例来说，汇编指令add $s0, $s1, $s2 意味着rs=$s1（17），rt=$s2（18），rd=$s0（16）。

汇编代码	字段值						机器码						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
add $s0, $s1, $s2	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
sub $t0, $t3, $t5	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6位	5位	5位	5位	5位	6位	6位	5位	5位	5位	5位	6位	

图6-6 R-类型指令的机器码

附录B中表B.1和表B.2定义了所有MIPS指令的opcode值和R-类型指令funct字段的值。

例6.3 汇编语言翻译成机器语言

将下列汇编语言的语句翻译为机器语言

```
add $t0, $s4, $s5
```

解：根据表6-1，$t0，$s4和$s5是寄存器8，20和21。根据表B.1和B.2，add的opcode为0，funct码为32。各个字段和机器码如图6-7所示。写出十六进制机器码最简单的方法是首先将写为二进制，然后观察每四位为一组的数，每一组数表示一个十六进制数字。因此，机器语言指令为0x02954020。

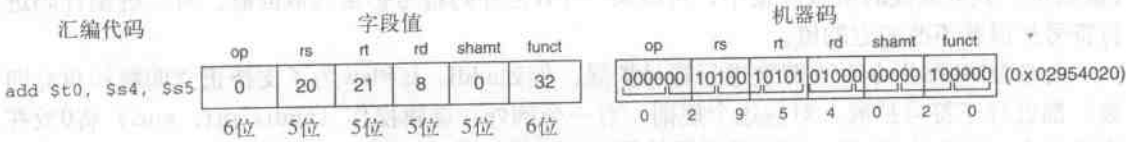


图6-7 例6.3中R-类型指令的机器码

6.3.2 I-类型指令

I-类型是立即数类型（immediate-type）的缩写。I-类型指令有两个寄存器操作数和一个立即数操作数。图6-8给出了I-类型机器指令的格式。一条32位指令有4个字段，分别为op，rs，rt和imm。前三个字段（op，rs和rt）与R-类型的指令一样。imm字段代表一个16位立即数。

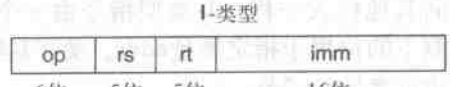


图6-8 I-类型指令格式

指令的操作由opcode字段决定。操作数则在另外的rs，rt和imm等三个字段中。rs和imm常用作源操作数。在一些指令中（如addi和lw），rt被用作目的操作数；但在其他的指令中（如sw），rt也作为源操作数。

图6-9给出了一些I-类型指令编码实例。记住，负立即数表示为16位有符号的二进制补码数。汇编语言中，rt作为目的操作数放在前面，但在机器语言指令中，它是第二个寄存器字段。

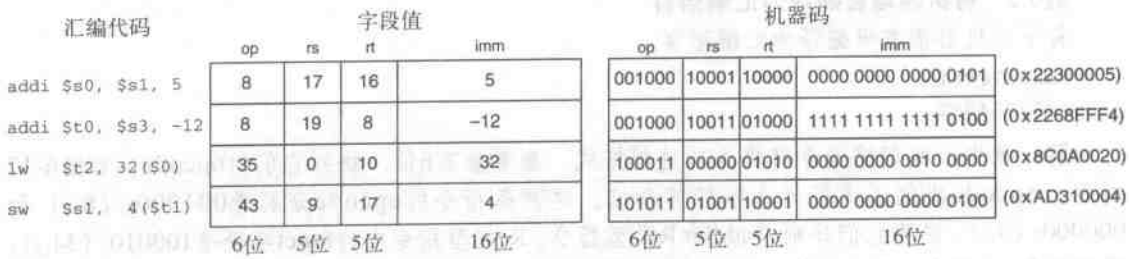


图6-9 I-类型指令的机器代码

例6.4 I-类型汇编指令翻译成机器码

将下列I-类型指令翻译为机器码。

```
lw $s3, -24 ($s4)
```

解：根据表6-1，$s3和$s4分别为寄存器19和20。表B.1给出了lw的opcode为35。rs指定了源地址$s4，rt指定了目的寄存器$s3，立即数imm编码表示16位偏移量−24。因此，指令的字段和机器码如图6-10所示。

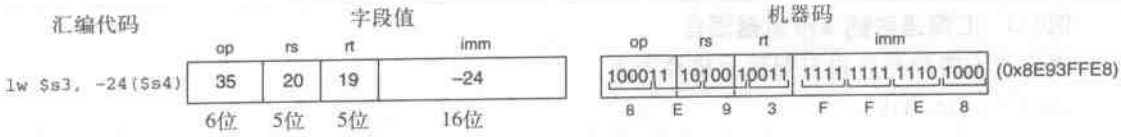


图6-10 I-类型指令的机器代码

I-类型指令有一个16位立即数字段，但这个立即数用于32位操作中。例如，lw将16位偏移量与32位源寄存器相加。那么，32位中的高16位应该是什么呢？对于正立即数，前16位为全0，但对于负立即数，高16位应该为全1。回忆1.4.6节，我们称它为符号扩展。通过复制符号位（最高位）到M位数的所有高位中，可以将一个N位补码符号扩展为M位数。对二进制补码进行符号扩展并不改变它的值。

大多数MIPS指令对立即数进行符号扩展。例如addi，ls和sw为了支持正立即数和负立即数，都进行了符号扩展。对应这个规则，有一个例外：逻辑操作（andi，ori，xori）将0放在高半字中，这称为0扩展，而不是符号扩展。逻辑操作将在后面6.4.1节中讨论。

6.3.3 J-类型指令

J-类型是跳转类型（jump-type）的缩写。这种格式仅用于跳转指令（参见6.4.2节）。如图6-11所示，这种指令格式有一个26位地址操作数addr。如同其他格式一样，J-类型指令由一个6位的opcode开始。剩下的位用于指定地址addr。关于J-类型指令进一步的讨论可参见6.4.2节。

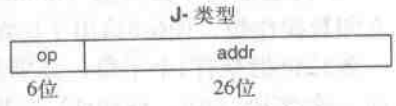


图6-11 J-类型指令格式

6.3.4 解释机器语言码

要解释机器语言，就必须对32位指令的每一个字段进行解码。不同的指令使用不同的格式，但是所有的指令都以一个6位的opcode段开始。所以，开始解释的最好地方是首先查看opcode。如果是0，则指令为R-类型，否则，为I-类型或J-类型。

例6.5 将机器语言翻译为汇编语言

将下列机器语言码翻译为汇编语言

```
0x2237FFF1
0x02F34022
```

解：首先，我们将指令转化为二进制格式，查看最高6位，找到它们的opcode，如图6-12所示。opcode决定了我们怎么解释其余位。这两条指令的opcode分别是001000_2（8_{10}）和000000_2（0_{10}），说明它们分别是addi和R类型指令。R-类型指令中的funct字段是100010_2（34_{10}），说明它是sub指令。图6-12给出了与两个机器指令等价的汇编代码。

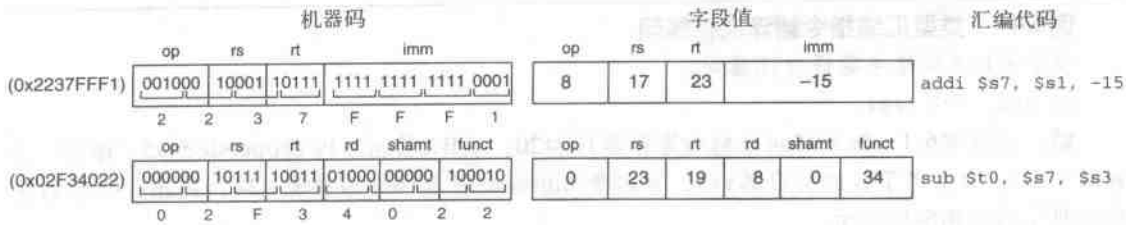


图6-12 机器代码与汇编代码的互译

6.3.5 程序存储

用机器语言编写的程序是一个表示指令的32位数序列。如同其他二进制数，这些指令存放在存储器中。这就是程序存储（stored program）的概念，也是计算机如此强大的一个关键原因。运行一个新的程序时，我们不需要花费大量的时间和精力对硬件进行重新装配或重新布线，只需要将一个新的程序写入存储器。程序存储提供了通用（general purpose）计算能力，而不是特定的硬件。在这种方式下，计算机仅仅只是改变存储的程序就可以运行计算器、文字处理程序、影音播放器等多种应用程序。

程序中的指令从存储器中找到或取出（fetch），然后由处理器执行。即使是大型的复杂程序也可以简化为仅仅包括简单的存储器读和指令运行的动作序列。

图6-13说明了机器指令是怎样存储在存储器中的。在MIPS程序中，指令一般从地址0x00400000开始存储。记住MIPS存储器地址是字节寻址的，所以32位（4字节）指令地址每次增加4个字节而不是1个字节。

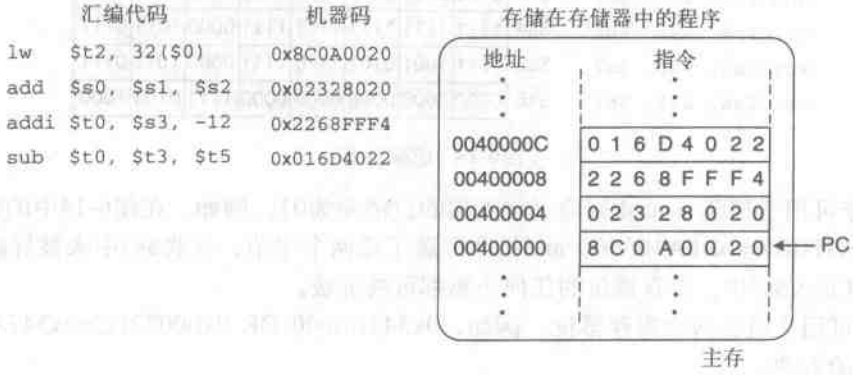


图6-13 存储器中的程序

运行（execute）程序时，处理器从存储器中顺序地取指令。然后，用数字电路硬件解码和执行这些取回的指令。当前指令地址存储在一个称为程序计数器（program counter, PC）的32位寄存器中。PC并不在先前表6-1所示的32个寄存器中。

为了运行图6-13所示的程序，操作系统将PC的值设为地址0x00400000。处理器将这个存储器地址的指令读出，并执行指令0x8C0A0020。然后，处理器将PC增加4，变为0x00400004，接着取出并执行该地址的指令，并循环执行上述步骤。

微处理器的体系结构状态（architectural state）保存了程序的状态。对于MIPS处理器，体系结构状态由寄存器文件和PC组成。如果操作系统在程序运行的某时刻保存了其体系结构状态，就可以中断该程序，做些别的事情。然后恢复原先的状态，被打断的程序又能够继续正确执行，而不知道它曾经被打断过。我们在第7章构建一个微处理器时，体系结构状态也十分重要。

6.4 编程

软件编程语言（如C或Java）称为高级编程语言，因为它们比汇编语言的抽象层次更高。许多高级语言使用算术和逻辑操作、if/else语句、for和while循环、数组索引和过程调用等常见的软件结构。本节将会探讨如何将这些高级语言结构翻译成MIPS汇编代码。

6.4.1 算术/逻辑指令

MIPS体系结构定义了大量的算术和逻辑指令。这些指令对实现高级语言结构是必须的，因此首先简要介绍它们。

逻辑指令

MIPS逻辑操作包括and、or、xor和nor。这些类型的指令对两个源寄存器和一个目的寄存器进行按位操作。图6-14给出了源操作数分别为0xFFF0000和0x46A1F0B7的操作示例。如图中所示，指令执行后结果存储在目的寄存器rd中。

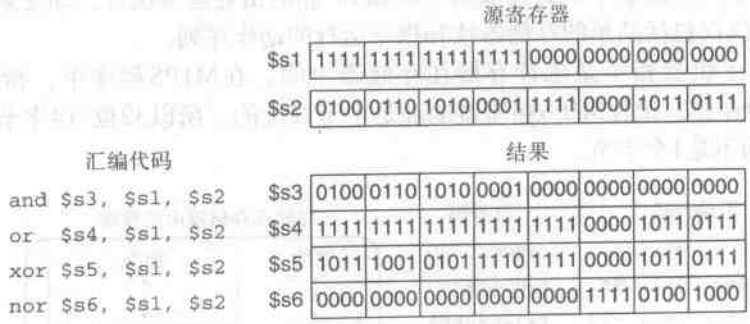


图6-14 逻辑操作

and指令可用于屏蔽（mask）位（将不用的位清除为0）。例如，在图6-14中的0xFFF0000 AND 0x46A1F0B7=0x46A10000。and指令屏蔽了低两个字节，并将$s3中未被屏蔽的高两个字节0x46A1写入$s3中。寄存器位的任何子集都可被屏蔽。

or指令可用于组合两个寄存器位。例如，0x347A0000 OR 0x00072FC=0x347A72FC，即将两个值组合起来。

MPIS不提供NOT指令，但A NOR $0=NOT A。因此，NOR指令可以代替NOT指令。

andi, ori, 和xori指令也可以对立即数进行逻辑操作。MIPS不提供nori指令，因为它的功能可以用其他指令轻松实现。习题6.11将研究这个问题。图6-15给出了andi，ori和xori指令的示例。图中给出了源寄存器和立即数的值，同时也给出了指令执行后目的寄存器rt的值。

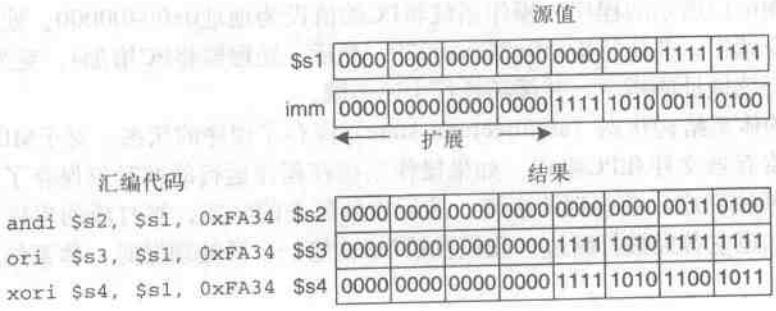


图6-15 带有立即数的逻辑操作

移位指令

移位指令可将寄存器中的值左移或右移最多31位。移位操作可将操作数乘或除以2的整数次幂。MIPS的移位操作包括逻辑左移指令（sll）、逻辑右移指令（srl）和算术右移指令（sra）。

在5.2.5节的讨论中，左移一般将低位补0。但右移可以是逻辑右移（高位补0）或算术右

移（高位补符号位）。图6-16给出了R-类型指令sll，srl和sra的机器码。rt（此例中的$s1）中保存了32位的待移位数，shamt给出了移位的位数（4）。移位的结果存放在rd中。

汇编代码	op	rs	rt	rd	shamt	funct
sll $t0, $s1, 4	0	0	17	8	4	0
srl $s2, $s1, 4	0	0	17	18	4	2
sra $s3, $s1, 4	0	0	17	19	4	3
	6位	5位	5位	5位	5位	6位

字段值

机器码	op	rs	rt	rd	shamt	funct	
	000000	00000	10001	01000	00100	000000	(0x00114100)
	000000	00000	10001	10010	00100	000010	(0x00119102)
	000000	00000	10001	10011	00100	000011	(0x00119903)
	6位	5位	5位	5位	5位	6位	

图6-16 移位指令的机器代码

图6-17给出了移位指令srl，slr和sra寄存器的值。如5.2.5节所讨论，将一个值左移N位相当于乘以2^N。同理，算术右移N位，相当于除以2^N。

源值

$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

结果

汇编代码									
sll $t0, $s1, 4	$t0	0011	0000	0000	0000	0010	1010	1000	0000
srl $s2, $s1, 4	$s2	0000	1111	0011	0000	0000	0000	0010	1010
sra $s3, $s1, 4	$s3	1111	1111	0011	0000	0000	0000	0010	1010

图6-17 移位操作

MIPS也提供移位位数可变的移位指令：可变逻辑左移指令（sllv），可变逻辑右移指令（srlv）和可变算术右移指令（srav）。图6-18给出了这些指令的机器码。rt（此例中的$s1）保存了待移位的值。rs（此例中的$s2）的低5位给出了位移的位数。和前面一样，移位的结果存放在rd中。shamt段为全0且被忽略。图6-19给出了每种可变移位指令寄存器的值。

汇编代码	op	rs	rt	rd	shamt	funct
sllv $s3, $s1, $s2	0	18	17	19	0	4
srlv $s4, $s1, $s2	0	18	17	20	0	6
srav $s5, $s1, $s2	0	18	17	21	0	7
	6位	5位	5位	5位	5位	6位

字段值

机器码	op	rs	rt	rd	shamt	funct	
	000000	10010	10001	10011	00000	000100	(0x02519804)
	000000	10010	10001	10100	00000	000110	(0x0251A006)
	000000	10010	10001	10101	00000	000111	(0x0251A807)
	6位	5位	5位	5位	5位	6位	

图6-18 变量移位指令的机器代码

源值

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

结果

汇编代码									
sllv $s3, $s1, $s2	$s3	0000	0100	0000	0010	1010	1000	0000	0000
srlv $s4, $s1, $s2	$s4	0000	0000	1111	0011	0000	0100	0000	0010
srav $s5, $s1, $s2	$s5	1111	1111	1111	0011	0000	0100	0000	0010

图6-19 变量移位操作

生成常数

addi指令可用于进行16位常数赋值，如代码示例6.10所示。

代码示例6.10 16位常数

高级语言代码

```
int a = 0x4f3c;
```

MIPS汇编代码

```
# $s0 = a
addi $s0, $0, 0x4f3c   # a = 0x4f3c
```

可以先用一条lui指令，接着用一条ori指令实现32位常数赋值，如代码示例6.11所示。lui指令将一个16位立即数读出寄存器的高16位。如前所述，ori将一个16位立即数并入寄存器的低16位。

代码示例6.11 32位常数

高级语言代码

```
int a = 0x6d5e4f3c;
```

MIPS汇编代码

```
# $s0 = a
lui $s0, 0x6d5e        # a = 0x6d5e0000
ori $s0, $s0, 0x4f3c   # a = 0x6d5e4f3c
```

乘法和除法指令*

乘法和除法指令有些不同于其他算术指令。两个32位数相乘，产生一个64位乘积。两个32位数相除，产生一个32位的商和一个32位的余数。

MIPS体系结构有两个用于存放乘法和除法结果的特殊用途寄存器hi和lo。mult $s0, $s1将$s0和$s1中的数相乘。结果的高32位存放在hi中，低32位存放在lo中。与此类似，div $s0, $s1计算$s0/$s1。商存放在lo中，余数存放在hi中。

6.4.2 分支

相对于计算器而言，计算机的优势在于它能做出判断，可以根据不同的输入处理不同的任务。例如，if/else语句，case语句，while循环和for循环，都是根据判断有条件地执行代码。

顺序执行指令时，程序计数器每执行一条指令后自加4。分支（branch）指令改变程序计数器的值，从而跳过某段代码或返回执行先前的代码。条件分支（condition branch）指令执行一次判断，并且当判断结果为真时执行分支语句。无条件分支（unconditional branch）指令称为跳转（jump）指令，它总执行分支语句。

条件分支

MIPS指令集有两条条件分支指令：beq和bne。当两个寄存器中的值相等时，beq执行分支语句；当两个值不相等时，bne执行分支程序。代码示例6.12说明了如何使用beq指令。注意：分支要写成beq $rs, $rt, imm, 即$rs为第一个源寄存器。这种顺序与大部分I-类型指令相反。

代码示例6.12 使用beq的条件分支

MIPS汇编代码

```
addi $s0, $0, 4          # $s0 = 0 + 4 = 4
addi $s1, $0, 1          # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
beq  $s0, $s1, target    # $s0 == $s1, so branch is taken
addi $s1, $s1, 1         # not executed
sub  $s1, $s1, $s0       # not executed

target:
add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

如代码示例6.12所示，当程序执行到分支指令beq时，$s0与$s1中的值相等，所以执行（taken）分支。下一条执行的指令是标号（label）后的目标指令（target），此例中为add指令。在分支指令后和标号前的两条指令将不执行。

汇编代码使用标号来说明程序中指令的位置。当汇编代码翻译成机器代码时，这些标号将翻译为指令地址（参见6.5节）。MIPS汇编语言标号后面跟着一个冒号“:”，并且不能使用指令助记符等保留字。大多数的程序员编程时为了突出标号，只缩进代码而不缩进标号。

代码示例6.13给出了使用条件分支指令bne的例子。在本例中，因为$s0与$s1相等，分支语句不发生跳转（not taken），并且继续执行bne指令后面的语句。

代码示例6.13 使用bne的条件分支

MIPS汇编代码

```
  addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne    $s0, $s1, target    # $s0 == $s1, so branch is not taken
  addi   $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub    $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
  add    $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

跳转指令

程序可以使用三种指令完成无条件分支或跳转（jump）。这三条指令分别是跳转指令（j），跳转链接指令（jal）和寄存器跳转指令（jr）。跳转指令（j）直接跳转到标号指定的指令。jal指令与j类似，但它保存了返回地址。这点我们将在6.4.6节中讨论。jr指令跳转到一个寄存器所保存的地址。代码示例6.14给出了使用跳转指令j的例子。

代码示例6.14 使用j无条件跳转

MIPS汇编代码

```
  addi   $s0, $0, 4     # $s0 = 4
  addi   $s1, $0, 1     # $s1 = 1
  j      target         # jump to target
  addi   $s1, $s1, 1    # not executed
  sub    $s1, $s1, $s0  # not executed

target:
  add    $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```

如代码示例6.14所示，在执行j target指令后，程序无条件继续执行标号target处的指令add。跳转指令和标号之间的所有指令都被跳过了。

代码示例6.15给出了使用寄存器跳转指令jr的例子。指令地址显示在每条指令的左边。jr $s0跳转到$s0所保存的地址，0x00002010。

代码示例6.15 使用jr无条件跳转

MIPS汇编代码

```
0x00002000   addi   $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004   jr     $s0              # jump to 0x00002010
0x00002008   addi   $s1, $0, 1       # not executed
0x0000200c   sra    $s1, $s1, 2      # not executed
0x00002010   lw     $s3, 44 ($s1)    # executed after jr instruction
```

6.4.3 条件语句

if语句、if/else语句和case语句是高级程序设计语言常用的条件语句。它们根据一条或多条指令，有条件地执行一块（block）代码。本节将介绍如何把这些高级语言结构翻译成MIPS汇编语言。

if语句

仅当满足条件时，if语句执行if块（if block）代码。代码示例6.16指出了如何将if语句翻译成MIPS汇编代码。

代码示例6.16 if语句

高级语言代码

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS汇编代码

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, L1     # if i != j, skip if block
  add $s0, $s1, $s2    # if block: f = g + h
L1:
  sub $s0, $s0, $s3    # f = f - i
```

if语句的汇编代码检测条件与高级语言代码相反。如代码示例6.16所示，高级语言检测i == j，但汇编代码检测i! = j。当i! = j时，bne指令执行分支（跳过if块）。否则，当i = = j时，不执行分支，按照希望的方式执行if块。

if/else语句

if/else语句根据条件执行两块代码中的一块。当满足if语句中的条件，执行if块，否则，执行else块。代码示例6.17给出了一个if/else语句的例子。

代码示例6.17 if/else语句

高级语言代码

```
if (i == j)
  f = g + h;

else
  f = f - i;
```

MIPS汇编代码

```
# $s0 = f, $s1 = g,   $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else     # if i != j, branch to else
  add $s0, $s1, $s2      # if block: f = g + h
  j   L2                 # skip past the else block
else:
  sub $s0, $s0, $s3      # else block: f = f - i
L2:
```

像if语句一样，if/else语句汇编代码检测的条件与高级语言中相反。例如，在代码示例6.17中，高级语言代码检测i = = j。汇编代码检测相反的条件（i! = j）。如果那个相反的条件为真，bne跳过if块，执行else块。否则，执行if块，并用一个跳转指令（j）跳过else块。

switch/case语句

switch/case语句根据条件执行多块代码中的一块。如果不能满足条件，则执行default块。一个case语句相当于多个嵌套的if/else语句。代码示例6.18给出了两个同样功能的代码片断。代码计算了自动柜员机（ATM）取出$20，$50和$100（定义为amount）的费用。MIPS汇编代码的实现与高级语言代码片断相同。

代码示例6.18 switch/case语句

高级语言代码

```
switch (amount) {
  case 20: fee = 2; break;
```

MIPS汇编代码

```
# $s0 = amount, $s1 = fee
case20:
```

```
    case 50: fee = 3; break;

    case 100: fee = 5; break;

    default: fee = 0;
  }

  // equivalent function using if/else statements
    if      (amount == 20)  fee = 2;
    else if (amount == 50)  fee = 3;
    else if (amount == 100) fee = 5;
    else                    fee = 0;
```

```
    addi $t0, $0,  20      # $t0 = 20
    bne  $s0, $t0, case50  # i == 20? if not,
                           #   skip to case50
    addi $s1, $0,  2       # if so, fee = 2
    j    done              # and break out of case
  case50:
    addi $t0, $0,  50      # $t0 = 50
    bne  $s0, $t0, case100 # i == 50? if not,
                           #   skip to case100
    addi $s1, $0,  3       # if so, fee = 3
    j    done              # and break out of case
  case100:
    addi $t0, $0,  100     # $t0 = 100
    bne  $s0, $t0, default # i == 100? if not,
                           #   skip to default
    addi $s1, $0,  5       # if so, fee = 5
    j    done              # and break out of case
  default:
    add  $s1, $0, $0       # charge = 0
  done:
```

6.4.4 循环

循环根据某一条件重复地执行一块代码语句。for循环和while循环是高级程序语言常用的循环结构。本节将介绍如何把它们翻译为MIPS汇编语言。

while循环

while循环重复地执行一块代码，直至某一条件不再满足。代码示例6.19中的while循环求出满足$2^x = 128$的x值。它循环7次，直到pow=128。

与if/else语句类似，while循环的汇编代码测试的条件与高级语言代码给定的相反。如果那个相反的条件为真，那么while循环就停止。

代码示例6.19中，while循环将pow值与128做比较，如果相等，就退出。否则，它将pow值乘以2（左移1位），递增x，然后跳转到while循环的开始处。

代码示例6.19 while循环

高级语言代码

```
int pow = 1;
int x   = 0;

while (pow != 128)
{
  pow = pow * 2;
  x = x + 1;
}
```

MIPS汇编代码

```
# $s0 = pow, $s1 = x
  addi  $s0, $0, 1      # pow = 1
  addi  $s1, $0, 0      # x = 0

  addi  $t0, $0, 128    # t0 = 128 for comparison
while:
  beq   $s0, $t0, done  # if pow == 128, exit while
  sll   $s0, $s0, 1     # pow = pow * 2
  addi  $s1, $s1, 1     # x = x + 1
  j     while
done:
```

for循环

与while循环类似，for循环重复地执行一段代码，直到某一条件不再满足。但是，for循环增加了对循环变量（loop variable）的支持，其典型作用是跟踪循环的执行次数。for循环的基本格式为：

```
for (initialization; condition; loop operation)
```

初始化（initialization）代码在for循环之前执行。每一次循环前检查条件（condition）是否满足，如果不满足条件，则退出循环。循环操作（loop operation）在每次循环后执行。

代码示例6.20将数字0加到9。循环变量i初始化为0.然后在每次循环后自增1。在每次循环前，当i不等于10时，for循环才执行。否则，循环结束。这种情况下，for循环执行10次。for循环可以用while循环来实现，不过for循环通常比较方便。

代码示例6.20 for循环

高级语言代码

```
int sum = 0;

for (i = 0; i ! = 10; i = i + 1) {
  sum = sum + i;
}

// equivalent to the following while loop
int sum = 0;
int i = 0;
while (i != 10) {
  sum = sum + i;
  i = i + 1;
}
```

MIPS汇编代码

```
# $s0 = i, $s1 = sum
  add   $s1, $0, $0     # sum = 0
  addi  $s0, $0, 0      # i   = 0
  addi  $t0, $0, 10     # $t0 = 10
for:
  beq   $s0, $t0, done  # if i == 10, branch to done
  add   $s1, $s1, $s0   # sum = sum + i
  addi  $s0, $s0, 1     # increment i
  j     for
done:
```

数量比较

目前为止，例子中使用了beq和bne指令执行相等或不相等的比较和分支转移。MIPS提供了小于设置指令（set less than，slt）实行数量比较。当rs<rt，slt将rd设为1，否则，rd为0。

例6.6 使用slt指令的循环

下述高级语言代码将从1～100中的2的整数次幂求和。将其转化为汇编语言程序。

```
// high-level code

int sum = 0;
for (i = 1; i < 101; i = i * 2)
  sum = sum + i;
```

解：汇编语言代码使用小于设置指令（slt）执行了for循环中的小于比较操作。

```
# MIPS assembly code

# $s0 = i, $s1 = sum
  addi $s1, $0, 0   # sum = 0
  addi $s0, $0, 1   # i = 1
  addi $t0, $0, 101 # $t0 = 101

loop:
  slt $t1, $s0, $t0     # if (i < 101) $t1 = 1, else $t1 = 0
  beq $t1, $0, done     # if $t1 == 0 (i >= 101), branch to done
```

```
    add $s1, $s1, $s0     # sum = sum + i
    sll $s0, $s0, 1       # i = i * 2
    j   loop
done:
```

习题6.12将考察了如何将slt用于大于、大于等于和小于等于等其他数量比较。

6.4.5 数组

数组用于访问大量类似数据。数组按照存储器中顺序数据的地址组织。每一个数组元素由下标（index）区分。数组中元素的个数被称为数组的长度（size）。本节展示如何访问存储器中数组的元素。

图6-20 基地址为0x10007000的五个表项数组的排列

数组下标

图6-20中的存储器中有一个包含5个整数的数组，下标的范围从0到4。这种情况下，数组存储在处理器主存中从基址（base address）0x10007000开始的连续区域。基址指定了第一个元素array[0]的地址。

代码示例6.21将数组的前两个元素乘以8，然后再存储到数组中。

代码示例6.21 数组访问

高级语言代码

```
int array [5];

array[0] = array[0] * 8;

array[1] = array[1] * 8;
```

MIPS汇编代码

```
# $s0 = base address of array
    lui   $s0, 0x1000      # $s0 = 0x10000000
    ori   $s0, $s0, 0x7000 # $s0 = 0x10007000

    lw    $t1, 0($s0)      # $t1 = array[0]
    sll   $t1, $t1, 3      # $t1 = $t1 << 3 = $t1 * 8
    sw    $t1, 0($s0)      # array[0] = $t1

    lw    $t1, 4($s0)      # $t1 = array[1]
    sll   $t1, $t1, 3      # $t1 = $t1 << 3 = $t1 * 8
    sw    $t1, 4($s0)      # array[1] = $t1
```

访问数组元素的第一步是将数组的基址读入寄存器。代码示例6.21将基址读入$s0。回忆可以用lui和ori指令将32位常数写入寄存器。

代码示例6.21同时也展示了为何lw指令需要基址和偏移量来形成有效地址。基址指向数组的起始位置，偏移量用于访问后面的元素。例如，array[1]存储在地址0x10007004（在array[0]的一个字或四个字节之后），所以它位于从基址之后偏移量为4的位置。

读者可能已经注意到：代码示例6.21中将两个数组元素相乘的代码除了下标外，其他都基本相同。对于访问两个数组元素来说，复制一段代码不成问题，不过在访问大数组的所有元素时，这十分不方便。代码示例6.22中用一个for循环将基址为0x23B8F000的数组中所有1000个元素乘以8。

代码示例6.22 使用for循环来访问数组

高级语言代码

```
int i;
int array[1000];
```

MIPS汇编代码

```
# $s0 = array base address, $s1 = i
# initialization code
    lui   $s0, 0x23B8      # $s0 = 0x23B80000
```

```
                                ori   $s0, $s0, 0xF000  # $s0 = 0x23B8F000
                                addi  $s1, $0           # i = 0
                                addi  $t2, $0, 1000     # $t2 = 1000
for (i=0; i < 1000; i = i + 1) {  loop:
                                slt   $t0, $s1, $t2     # i < 1000?
                                beq   $t0, $0, done     # if not then done
                                sll   $t0, $s1, 2       # $t0 = i * 4 (byte offset)
                                add   $t0, $t0, $s0     # address of array[i]
  array[i] = array[i] * 8;      lw    $t1, 0($t0)       # $t1 = array[i]
                                sll   $t1, $t1, 3       # $t1 = array[i] * 8
                                sw    $t1, 0($t0)       # array[i] = array[i] * 8
                                addi  $s1, $s1, 1       # i = i + 1
                                j     loop              # repeat
}                               done:
```

图6-21表示了存储器中有1000个元素的一个数组。数组下标为变量i而不是一个常数，所以不能使用lw中的立即数偏移量。相反，可以计算第i个元素的地址，存储在$t0中。需要注意的是：每一个元素为一个字长，存储器以字节编址。所以从基址开始，偏移量为i×4。在MIPS汇编语言中，可以用左移两位实现乘4的操作。这个例子可以很容易地扩展应用到任意长度的数组。

字节和字符

范围在[−128, 127]之间的数可以存储在一个单独的字节中，而不需要一个完整的字。因为英语键盘上按键的数目远远少于256，所以英语字符可以用字节表示。C语言使用char类型来表示字节或字符。

早期的计算机缺乏字节与英语字符之间的标准映射，所以计算机之间交换文本很困难。1963年，美国标准委员会发表了《用于信息交换的美国标准代码》（ASCII码）。它为每个文本字符确定了一个唯一的字节值。表6-2给出了可印刷字符的编码。ASCII码采用十六进制编码。大写字母与小写字母之间相差0x20（32）。

MIPS提供了读出字节和写入字节指令来操作字节或字符类型的数据，主要包括：无符号字节读出（lbu），字节读出（lb）和字节写入（sb）。图6-22描述了这三条指令。

图6-21 基地址为0x23B8F000的1000个数组在内存中的排列

图6-22 字节读出和存储指令

无符号字节读出指令（lbu）和字节读出指令（lb）分别对字节进行0扩展和符号扩展来填充32位的寄存器。字节写入指令（sb）存储32位寄存器中最低的一个字节到指定地址的存储器中。如图6-22，lbu指令将内存地址为2的数据读出到$s1的最低字节，并将$s1的剩余部分用

0填充。lb指令将内存地址为2的数据符号扩展后读出到$s2中。sb指令将$s3的最低字节存储到存储器地址为3处：用0x9B取代了0xF7。$s3的高字节部分忽略。

表6-2 ASCII编码表

#	字符	#	字符	#	字符	#	字符	#	字符	#	字符
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

例6.7 使用lb和sb来访问字符数组

下述高级语言代码将大小为10的数组中所有小写字母减去32，从而将其转化为大写字母。将此高级语言代码翻译成MIPS的汇编语言。注意数组元素之间的地址变化是1个字节而不是4个字节。假定$s0已经保存了chararray的基地址。

```
// high-level code

char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

解：

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i

        addi $s1, $0, 0      # i = 0
        addi $t0, $0, 10     # $t0 = 10
loop:   beq  $t0, $s1, done # if i == 10, exit loop
        add  $t1, $s1, $s0   # $t1 = address of chararray[i]
        lb   $t2, 0($t1)     # $t2 = array[i]
        addi $t2, $t2, -32   # convert to upper case: $t1 = $t1 - 32
        sb   $t2, 0($t1)     # store new value in array:
                             # chararray[i] = $t1
        addi $s1, $s1, 1     # i = i + 1
        j    loop            # repeat
done:
```

一个字符序列称为字符串（string）。字符串的长度可变，因此程序设计语言必须提供一种方法来确定字符串的长度或确定字符串的结尾。在C语言中，空字符（0x00）意味着字符串结束。例如，图6-23给出了字符串“Hello!”（0x48 65 6C 6C 6F 21 00）在内存中的存储方式。这个字符串有7个字节，地址从0x1522FFF0到0x1522FFF6，字符串的第一个字符（H = 0x48）存储于最低字节地址（0x1522FFF0）。

图6-23 字符串“Hello!”在存储器中的存储

6.4.6 过程调用

高级程序语言经常使用过程（procedure，也称为函数，functions）重用经常使用的代码，并使程序更加可读。过程的输入和输出分别称为参数（arguments）和返回值（return value）。过程将计算返回值并且不会产生其他非预期的不良影响。

当一个过程去调用其他过程时，调用过程（caller）和被调用过程（callee）必须要在参数和返回值上保持一致。MIPS系统的惯例是：调用过程在调用之前要将4个参数分别放在$a0-$a3中，被调用过程在完成之前将返回值放入$v0-$v1中。遵循这种惯例，即使由不同人所写的调用过程和被调用过程也均知道参数和返回值存在何处。

被调用过程不能影响调用过程的功能。简单地说，这意味着被调用过程必须知道当它完成之后要返回到哪里，而且它不能破坏调用过程用到的寄存器和内存。调用过程将返回地址（return address）存在$ra寄存器中，与此同时它使用jal指令跳转到被调用过程入口。被调用过程不能覆盖调用过程所需要的任何体系结构状态和内存。具体来说，被调用过程必须保证保存寄存器$s0-$s7，$ra以及用于存放临时变量的栈（stack）不被修改。

本节将介绍如何调用一个过程并从此被调用过程中返回，同时还将介绍如何访问输入参数和返回值，如何使用栈来存储临时变量。

程序调用和返回

MIPS使用jal指令调用一个过程，使用jr指令从过程中返回。代码实例6.23描述了main过程如何调用simple过程，其中main是调用过程，simple是被调用过程。调用simple过程时没有输入参数，也不产生返回值；它仅仅是返回到调用过程。在代码示例6.23中，指令地址在每一条MIPS指令左侧以十六进制方式给出。

代码示例6.23 简单的程序调用

高级语言代码

```
int main() {
  simple();
  ...
}
// void means the function returns no value
void simple() {
  return;
}
```

MIPS汇编代码

```
0x00400200 main:   jal simple # call procedure
0x00400204         ...

0x00401020 simple: jr $ra     # return
```

jal指令和jr指令是过程调用中两个很重要的指令。jal指令完成两种功能：1）存储下一条指令（jal之后的指令）的地址到返回地址寄存器$ra中；2）跳转到目标指令。

在代码示例6.23中，main过程通过执行jal指令调用simple过程。jal跳转到simple标号处，同时将0x00400204存储到$ra寄存器中。simple过程通过执行jr $ra（跳转到$ra寄存器中的指令

地址处）立即返回。主函数从地址0x00400204处继续执行。

输入参数和返回值

代码实例6.23中的simple过程没有什么用处。它既没有从调用过程main中获得输入，也没有返回值输出。根据MIPS的习惯，程序使用$a0-$a3保存输入参数，使用$v0-$v1保存返回值。在代码示例6.24中，将调用有4个参数的过程diffofsums，并返回一个返回值。

代码示例6.24 拥有参数和返回值的程序调用

高级语言代码

```
int main ()
{
  int y;
  ...
  y = diffofsums (2, 3, 4, 5);
  ...
}

int diffofsums (int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;
}
```

MIPS汇编代码

```
# $s0 = y
main:
  ...
  addi $a0, $0, 2   # argument 0 = 2
  addi $a1, $0, 3   # argument 1 = 3
  addi $a2, $0, 4   # argument 2 = 4
  addi $a3, $0, 5   # argument 3 = 5
  jal diffofsums    # call procedure
  add $s0, $v0, $0  # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1 # $t0 = f + g
  add $t1, $a2, $a3 # $t1 = h + i
  sub $s0, $t0, $t1 # result = (f + g) - (h + i)
  add $v0, $s0, $0  # put return value in $v0
  jr  $ra           # return to caller
```

根据MIPS的习惯，调用程序main将程序参数从左到右放入输入寄存器$a0～$a3中。被调用程序diffofsums将返回值存储到返回寄存器$v0中。

返回64位值（例如一个双精度浮点数）的过程将使用两个返回寄存器$v0和$v1。当调用多于4个参数的过程时，多出来的输入参数将放入栈中，这个问题我们下面讨论。

栈

栈（stack）是用于存储过程中局部变量的存储单元，当处理器需要更多空间时，栈会扩展（使用更多内存）；当处理器不再需要存在栈中的变量时，栈会缩小（使用较少的内存）。在解释程序栈如何存储临时变量前，我们首先解释栈是怎样工作的。

栈是一个后进先出的队列（last-in-first-out queue）。如同一堆盘子，最后入栈的元素（最上面的盘子）首先被推出。每一个过程需要分配栈空间来存储局部变量，并在过程返回前回收空间。栈顶（the top of stack）是最后分配的空间。然而一堆盘子的空间是向上增长的，MIPS栈在内存中是向下增长的。当一个程序需要更多的空间时，栈空间向内存中地址较低的方向扩展。

图6-24给出了栈的图示。栈顶指针（stack pointer）$sp是一个特定的MIPS寄存器，此寄存器指向栈的最顶端，其中指针（pointer）是对于内存地址的一个新名字。指针指向数据，给出了此数据的地址。例如，图6-24(a) 中栈指针$sp保存了地址的值0x7FFFFFFC，指向数值0x12345678。$sp指向栈的顶端（栈中可访问的最低内存地址）。因此，在图6-24(a) 中，栈不能访问比

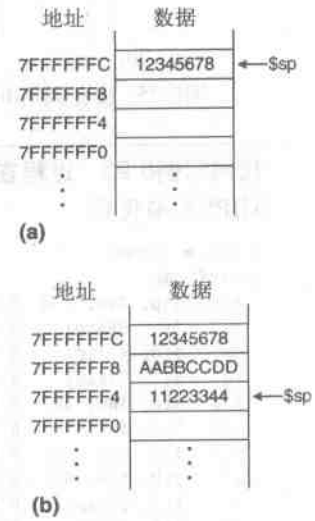


图6-24 栈

0x7FFFFFFC更低的内存地址。

栈指针（$sp）开始于一个较高的内存地址，通过地址的递减来扩展栈空间。图6-24（b）描述了栈如何扩展两个临时存储字。要实现扩展，$sp减少8而变成0x7FFFFFF4。两个新扩展的数据字（0xAABBCCDD和0x11223344）临时存储在栈中。

栈的一个重要应用是保存和恢复用于过程的寄存器。一个过程要计算返回值，但不应产生其他负面影响，具体来说，除了包含返回值的寄存器$v0，其他任何寄存器都不应该被修改。代码实例6.24中的diffofsums程序破坏了这个规则，因为它修改了$t0，$t1和$s0。如果main在调用diffofsums之前使用$t0，$t1或者$s0，这些寄存器的内容会被调用过程破坏。

为了解决这个问题，在修改寄存器之前，过程要将寄存器保存在栈中，然后在返回之前从栈中恢复这些寄存器。具体来说，将按照以下过程执行：

1) 创建栈空间来存储一个或多个寄存器的值；

2) 将寄存器的值存储在栈中；

3) 使用寄存器执行程序；

4) 从栈中恢复寄存器的原始值；

5) 回收栈空间。

代码示例6.25给出了diffofsums的改进版本，其存储和恢复了$t0，$t1和$s0。图6-25描述了调用diffofsums之前、之中和之后的栈的情况。diffofsums通过将栈指针减少12以得到3个字存储空间，然后在新分配的空间中存储$t0，$t1和$s0的当前值，接着将执行后续过程，并可以改变这3个寄存器的值。在过程执行的末尾，diffofsums从栈中恢复$t0，$t1和$s0的值，回收栈空间，并返回，当过程返回时，用$v0保存结果，但其他寄存器不受影响：$t0，$t1，$s0和$sp中的数值与过程调用之前的数值相同。

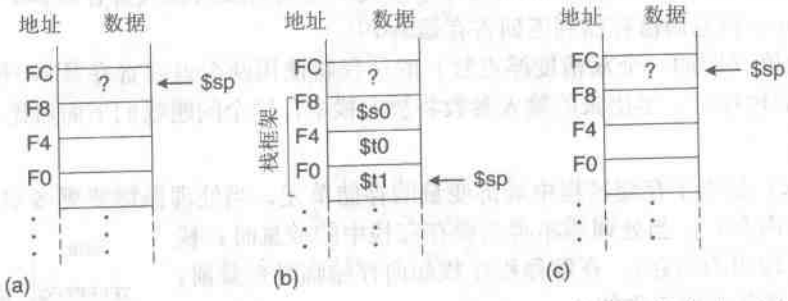


图6-25 栈在调用diffofsums之前、之中、之后的情况：(a) 之前；(b) 之中；(c) 之后

代码示例6.25 过程在栈中保存寄存器

MIPS汇编代码

```
# $s0 = result
diffofsums:
  addi  $sp, $sp, -12  # make space on stack to store three registers
  sw    $s0, 8($sp)    # save $s0 on stack
  sw    $t0, 4($sp)    # save $t0 on stack
  sw    $t1, 0($sp)    # save $t1 on stack
  add   $t0, $a0, $a1  # $t0 = f + g
  add   $t1, $a2, $a3  # $t1 = h + i
  sub   $s0, $t0, $t1  # result = (f + g) - (h + i)
  add   $v0, $s0, $0   # put return value in $v0
  lw    $t1, 0($sp)    # restore $t1 from stack
  lw    $t0, 4($sp)    # restore $t0 from stack
  lw    $s0, 8($sp)    # restore $s0 from stack
```

```
addi  $sp, $sp, 12   # deallocate stack space
jr    $ra            # return to caller
```

过程为自己分配的栈空间称为栈框架（stack frame）。diffofsums的栈框架深度为3个字。模块化的原则告诉我们，每个过程只应访问自己的栈框架而不应去访问其他过程的栈框架。

受保护的寄存器

代码示例6.25假定临时寄存器$t0和$t1必须被保存和恢复。如果调用函数不用这些寄存器，对它们的保存和恢复就是无用的操作。为了避免这种无用的操作，MIPS将寄存器划分为受保护（preserved）类型和不受保护（nonpreserved）类型。受保护寄存器包括$s0～$s7（它们的名字就是保存寄存器），不受保护寄存器包括$t0～$t9（它们的名字就是临时寄存器）。一个过程必须保存和恢复任何需要使用的受保护寄存器，但是可以随意改变不受保护的寄存器。

代码示例6.26描述了对diffofsums的进一步修改：只将$s0保存在栈中，$t0和$t1是不受保护的寄存器，所以它们不需要保存在栈中。

注意当一个过程调用另一个过程时，前者称为调用过程，后者称为被调用过程。被调用过程必须保存和恢复它要用到的受保护寄存器，被调用过程有可能改变任何不受保护寄存器，因此如果调用过程需要其不受保护寄存器中有效数据不被改变，那么它在过程调用之前需要保存不受保护寄存器，而且还需要在调用之后恢复这些寄存器。这种情况下，受保护寄存器也可以称为被调用者保存的（callee-save），不受保护寄存器称为调用者保存的（caller-save）。

代码示例6.26　将受保护的寄存器保存在栈中的程序段

MIPS汇编代码

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to store one register
  sw   $s0, 0($sp)    # save $s0 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr   $ra            # return to caller
```

表6-3总结了哪些寄存器是受保护的，$s0～$s7常用于保存过程中的局部变量，所以它们必须被保护。$ra也是要被保护的，这样才能使得过程知道返回到哪里。$t0～$t9用于在向局部变量赋值前保存临时结果，这些计算结果一般在过程调用之前完成，所以它们不用受保护，调用过程一般不需要保存它们。$a0～$a3经常在被调用的过程中被覆盖，因此，如果一个被调用的过程返回之后，调用过程的执行取决于它自身参数时，$a0～$a3需要由调用过程来保存。$v0～$v1不用被保护，因为被调用过程将返回结果放入这些寄存器中。

表6-3　受保护和不受保护寄存器

受保护寄存器	不受保护寄存器
保存寄存器：$0～$7	临时寄存器：$t0～$t9
返回地址：$ra	参数寄存器：$a0～$a3
堆栈：$sp	返回值寄存器：$v0～$v1
堆栈指针以上的空间	堆栈指针以下的空间

栈指针之上的栈空间要自动保护起来，被调用过程不向$sp之上的内存地址写入数据，这

样就使得任何过程不会去修改栈框架。栈指针自身是受保护的，这是因为被调用过程在返回之前需要回收自己的栈空间，栈空间的大小为过程结束时的地址减去过程开始时$sp保存的值。

递归过程调用

不用调用其他过程的过程称为叶子（leaf）过程，如diffofsums过程。需要调用其他过程的过程叫做非叶子（nonleaf）过程。如同前面提到的，非叶子过程一般会更复杂，因为在调用其他过程之前，它们需要把不受保护的寄存器保存到栈中，调用完之后再恢复这些寄存器。具体来说，调用过程保存它们所需要的任何不受保护的寄存器（$t0～$t9和$a0～$a3），被调用过程保存需要修改的任意受保护寄存器（$s0～$s7和$ra）。

递归（recursive）过程是调用自己的一个非叶子过程。阶乘函数可以使用一个递归过程来描述。回忆阶乘函数为factorial（n）=n×（n-1）×（n-2）×...×2×1。其可以递归地写成factorial（n）= n*factorial（n-1）。1的阶乘还是1。代码示例6.27描述了阶乘函数的递归写法。为了方便地标明过程地址，假定过程的开始地址为0x90。

factorial过程可能会修改$a0和$ra，所以要将这两个寄存器保存在栈中，然后检查n是否小于2，如果n小于2，就返回1并保存在$v0中，恢复栈指针，然后返回调用过程。这种情况下不需要重新读出$ra和$a0，因为它们没有被修改。如果n大于1，过程将递归调用factorial（n-1），然后从栈中恢复n（$a0）和返回地址（$ra），执行乘法，然后返回结果。乘法指令（mul $v0, $a0, $v0）将$a0和$v0相乘，其结果取代$v0中的值。在6.7.1节会有更进一步的讨论。

代码示例6.27 factorial的递归调用

高级语言代码

```
int factorial (int n) {


  if (n <= 1)
      return 1;



  else
    return (n * factorial (n-1));
}
```

汇编语言代码

```
0x90  factorial:  addi  $sp, $sp, -8    # make room on stack
0x94              sw    $a0, 4($sp)     # store $a0
0x98              sw    $ra, 0($sp)     # store $ra
0x9C              addi  $t0, $0, 2      # $t0 = 2
0xA0              slt   $t0, $a0, $t0   # a <= 1 ?
0xA4              beq   $t0, $0, else   # no: goto else
0xA8              addi  $v0, $0, 1      # yes: return 1
0xAC              addi  $sp, $sp, 8     # restore $sp
0xB0              jr    $ra             # return
0xB4        else: addi  $a0, $a0, -1    # n = n - 1
0xB8              jal   factorial       # recursive call
0xBC              lw    $ra, 0($sp)     # restore $ra
0xC0              lw    $a0, 4($sp)     # restore $a0
0xC4              addi  $sp, $sp, 8     # restore $sp
0xC8              mul   $v0, $a0, $v0   # n * factorial (n-1)
0xCC              jr    $ra             # return
```

图6-26描述了执行factorial（3）时栈的情况。假定$sp的初始值指向0xFC，如图6-26（a）所示。过程创建两个字的栈空间来保存$a0和$ra。在第一次过程调用时，factorial将$a0（$a0中保存着n=3）保存在0xF8中，将$ra保存在0xF4中，如图6-26（b）所示。然后过程将$a0中的内容改变为n=2并且递归调用factotial（2），$ra保存着0xBC。在第二次过程调用时，fuctorial将$a0（$a0中保存着n=2）保存在0xF0中，将$ra保存在0xEC中。这次我们知道$ra中存储了0xBC。然后过程将$a0中的内容改变为n=1并且递归调用factorial（1）。在第三次过程调用时，fuctorial将$a0（$a0中保存这n=1）保存在0xE8中，将$ra保存在0xE4中。这次$ra存储的还是0xBC。fuctorial的第三次递归返回1且保存在$v0中，而且在返回第二次调用之前回收栈空间。第二次调用将n恢复为2，将$ra恢复为0xBC（$ra中已经是这个值了），然后回收栈空间，返回$v0=2×1=2到第一次调用之处。第一次调用将n恢复为3，将$ra恢复为调用函数的返回地址，然后回收栈空间，返回$v0=3×2=6。图6-26(c) 描述了在递归调用过程返回时栈的情况。当factorial返回到调用过程时，栈指针指向它的初始位置（0xFC），指针之上栈空间的

内容没有变化，而且所有受保护的寄存器保存的是它们的初始值，$v0保存的是返回值6。

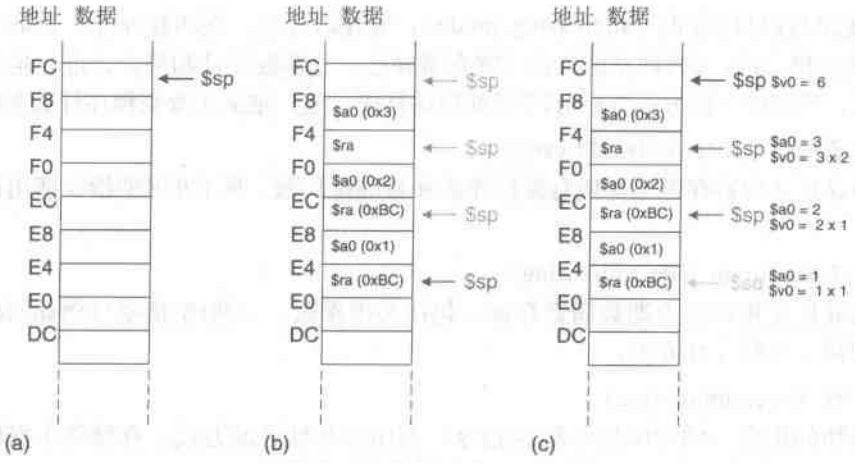


图6-26 在调用factorial过程（n=3）中栈的变化：(a) 调用前；(b) 最后一次递归调用后；(c) 返回后

*附加参数和局部变量**

过程的参数可能多于4个且含有局部变量，可以用栈存储这些临时数据。依照MIPS的惯例，如果一段过程有4个以上的参数，前4个参数像往常一样存储在参数寄存器中，另外的参数使用栈指针上面的空间保存在栈中。调用过程（caller）必须扩展栈空间来满足另外的参数，图6-27(a) 描述了调用多于4个参数的过程时栈的情况。

一个过程也可以定义局部变量或数组，局部变量在过程内部定义并且只能在过程内部有效。局部变量存储在$s0-～$s7中。如果有许多局部变量，它们也可以存储在这个过程的栈空间中。另外，局部数组也存储在栈中。

图6-27(b) 给出了被调用过程的栈结构。它保存了过程自己的参数（如果它调用其他过程）、返回地址、过程要修改的保存寄存器。此外，它还存储了局部数组和局部变量。如果被调用过程有4个以上的参数，可以从调用过程的栈框架中找到它们。访问额外的输入参数是一种特殊情况，在这种情况下过程可以访问不属于自己栈中的数据。

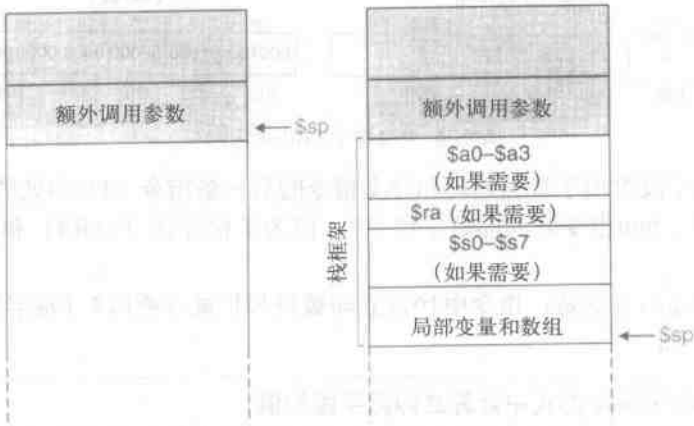


图6-27 栈的使用：左图为调用前，右图为调用后

6.5 寻址方式

MIPS使用5种寻址方式（addressing mode）：寄存器寻址、立即数寻址、基址寻址、相对PC寻址和伪直接寻址。前3种寻址方式（寄存器寻址、立即数寻址和基址寻址）定义了读写操作数的模式，后两种寻址方式（相对PC寻址和伪直接寻址）定义了改变程序计数器PC的模式。

寄存器寻址（register-only addressing）

寄存器寻址使用寄存器存储所有源操作数和目的操作数。所有的R型指令都用寄存器寻址方式。

立即数寻址（immediate addressing）

立即数寻址使用16位立即数同寄存器一起作为操作数。一些I型指令（例如addi指令和lui指令）都使用立即数寻址方式。

基址寻址（base addressing）

存储器访问指令（例如lw指令和sw指令）都使用基址寻址方式。存储器中操作数的有效地址由寄存器rs中的基址与16位立即数形成的偏移量相加得到。

相对PC寻址（PC-relative addressing）

条件分支指令在进行分支跳转时使用相对PC寻址方式来确定PC的新值。立即数字段中的偏移量与PC值相加得到新的PC值；因此，分支的目的地址与当前PC值相关。

代码示例6.28描述了代码实例6.27中factorial程序的一个片段。图6-28给出了beq指令的机器代码。如果分支执行，那么分支目标地址（branch target address，BTA）是下一条执行指令的地址。图6-28中beq指令的目标地址是0xB4，即标号else的指令地址。

代码示例6.28 计算分支指令的目标地址

MIPS汇编代码

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xB0          jr   $ra
0xB4    else: addi $a0, $a0, -1
0xB8          jal  factorial
```

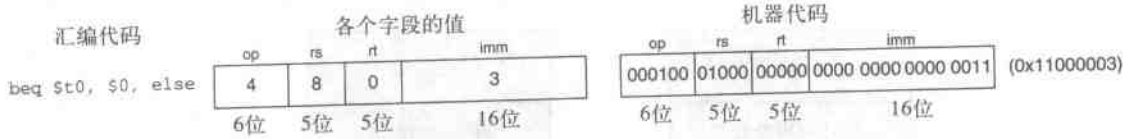


图6-28 beq指令的机器代码

16位的立即数字段给出了目标地址和分支指令的后一条指令（PC+4处的指令）之间的指令数目。在本例下，beq指令的立即数字段为3，因为目标地址（0xB4）和PC+4处（0xA8）之间有3条指令。

处理器计算目标的方法是：指令中16位立即数符号扩展并乘以4（将字转化为字节），然后将结果与PC+4相加。

例6.8 在相对PC寻址方式中计算立即数字段的值

计算下述程序中bne指令中立即数字段的值，并将此指令转化成机器码。

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

解：图6-29给出了bne指令的机器代码，它的分支目的地址0x40与PC+4（0x58）之间间隔6条指令，所以立即数字段的值为−6。

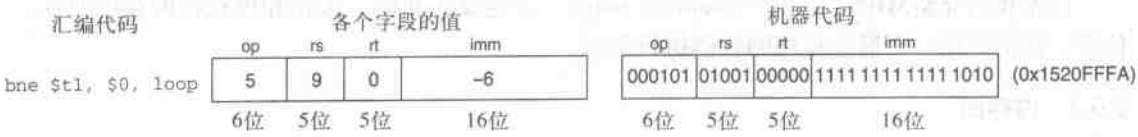


图6-29 bne指令的机器代码

伪直接寻址

直接寻址（direct addressing）的地址在指令中是直接给出的。在理想情况下，跳转指令j和jal应该用直接寻址方式，指明32位的跳转目标地址（Jump Target Address，JTA）来跳转到下一条指令处执行。

但是，J型指令编码没有足够的位数来表示32位的跳转目标地址，指令中有6位用于opcode，所以只剩26位来为跳转目标地址编码。幸运的是，最低两位$JTA_{1:0}$一直保持为0，这是因为指令是字对齐的。前面26位$JTA_{27:2}$由指令的addr字段指出。最高4位$JTA_{31:28}$由PC+4的最高4位得到。这种寻址方式被称为伪直接寻址（pseudo-direct addressing）。

代码示例6.29解释了使用伪直接寻址方式寻址的jal指令。jal指令的跳转目标地址为0x004000A0，图6-30给出了jal指令的机器代码，其中跳转目标地址的最高4位和最低2位被丢弃，剩下的内容存储在26位的地址字段（addr）中。

代码示例6.29 计算跳转目标地址

MIPS汇编编码

```
0x0040005C          jal     sum
...
0x004000A0  sum:    add     $v0, $a0, $a1
```

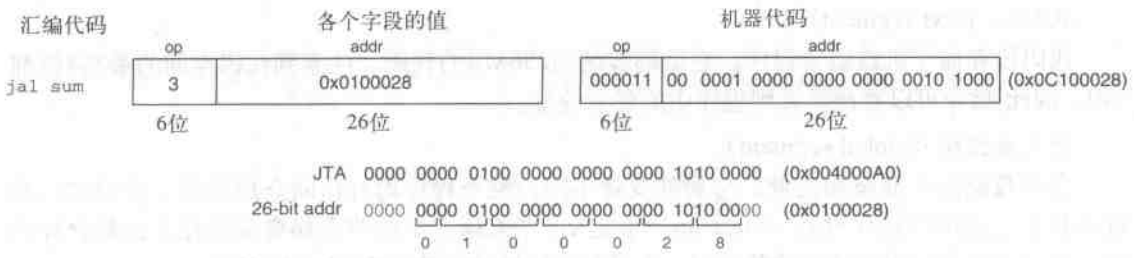


图6-30 jal机器代码

处理器计算J-类型指令跳转目标地址的方法是：在26位地址字段（addr）后面添加两个0，然后取PC+4的最高4位放在addr的前面。

由于JTA的最高4位是由PC+4得来的，所以跳转的范围受到了限制。分支跳转指令的受限范围见练习6.23到练习6.26。所有J-类型指令（包括j指令和jal指令）都使用伪直接寻址。

注意：寄存器跳转指令jr不是J-类型指令，而是一个R-类型指令，它将跳转到在rs寄存器中的地址。

6.6 编译、汇编和加载

到目前为止，我们讲解了怎样将一小段的高级语言转换成汇编语言和机器代码。本节将介绍如何编译和汇编一个完整的高级语言程序，以及如何将程序加载到存储器来执行。

首先我们介绍MIPS内存图（memory map），它定义了代码、数据和堆栈在内存中的存储位置，然后介绍一个简单程序的代码执行步骤。

6.6.1 内存图

MIPS地址宽度为32位，所以MIPS的地址空间为2^{32}字节（4GB）。字地址为4GB除以4，所以字地址范围为0～0xFFFFFFFC。图6-31展示了MIPS的内存图。MIPS体系结构将地址空间分为四个部分或者四段：代码段（text segment），全局数据段（global segment），动态数据段（dynamic data segment）和保留段（reserved segments）。下面分别介绍各个段。

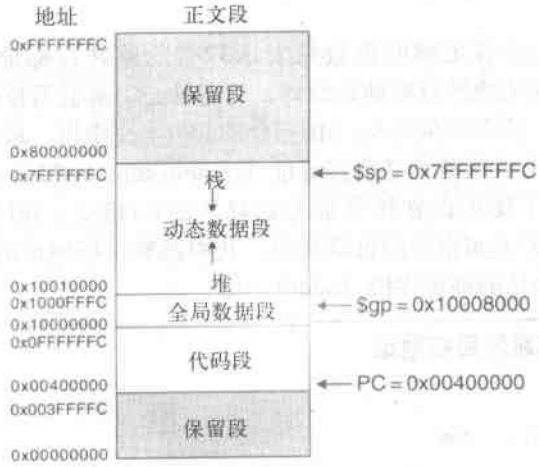


图6-31 MIPS内存图

代码段（text segment）

代码段存储了机器语言程序。它足够容纳约256MB的代码。注意到代码空间的最高4位都为0，因此j指令可以直接跳转到程序中的任意地址。

全局数据段（global segment）

全局数据段存储全局变量。与局部变量不同，整个程序均可访问全局变量。全局变量在程序执行之前的初始化过程（start-up）中定义。在C语言中这些全局变量是在主函数之外声明，且可以被程序中的任意函数访问。全局数据段足够容纳64KB的全局变量。

全局变量使用全局指针（$gp）访问，该指针初始化为0x10008000。与堆栈指针（$sp）不同的是，$gp在程序执行时保持不变。任意全局变量都可以基于$gp的16位正负偏移量来访问。偏移量在汇编时确定，因此可以使用带常数偏移量的基址寻址模式来访问全局变量。

动态数据段（dynamic data segment）

动态数据段保存在栈和堆（heap）中。段中的数据在程序初始化时还不确定，而是在程序执行过程中动态地分配和回收。动态数据段是一个程序占用内存最多的段，使用了大约2GB的地址空间。

如6.4.6节所讲，栈用于保存和恢复过程使用的寄存器，并且用于保存诸如数组之类的局部变量。栈从动态数据段（0x7FFFFFFC）上方向下增长，并以后进先出（LIFO）的顺序访问。

堆存储了程序运行中所分配的数据。C语言中使用malloc函数分配内存；在C++和Java语言中，使用new分配内存。如同宿舍地板上的一堆衣服，堆中的数据可以以任意顺序使用和丢弃。堆从动态数据段的下方向上增长。

如果栈和堆增长到对方的空间，程序中的数据就会被破坏（corrupted）。如果没有足够的空间来分配更多的动态数据，则内存分配器会通过返回内存溢出（out-of-memory）错误尽量确保不会发生数据被破坏的情况。

保留段（reserved segment）

保留段用于操作系统，不能直接被程序使用。部分保留段用于中断（见7.7节）和内存映射I/O（见8.5节）。

6.6.2 转换成二进制代码和开始执行程序

图6-32给出了将程序从高级语言转换成机器代码并开始执行的主要步骤。首先，高级语言程序被编译成汇编程序，汇编程序被汇编为机器代码放在目标文件（object file）中，链接程序将机器代码和来自库和其他文件的机器代码链接起来产生一个完整可执行文件。实际上，大部分编译器都执行了编译、汇编和链接这三步。最后，装载程序将可执行代码装载到内存中然后开始执行。本节按照以上步骤来执行一个简单的程序。

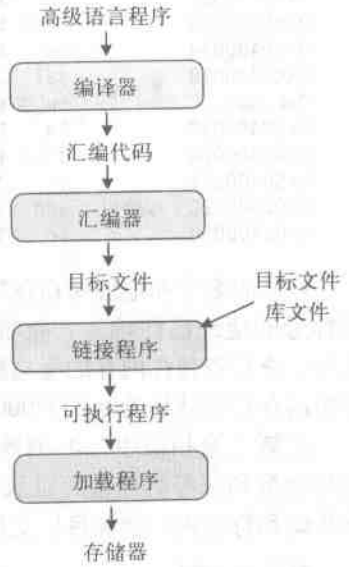


图6-32 编译和执行一个程序的步骤

步骤1：编译

编译器将高级语言程序转换成汇编语言，代码示例6.30描述了一个有3个全局变量和2个函数的简单高级语言程序，以及由典型编译器生成的汇编代码。.data和.text关键字是指明了数据段和代码段开始位置的汇编器指令（assembler directives）。全局变量f，g和y都被标记。它们的存储位置将由汇编器决定；到目前为止，它们仅作为代码中的符号而保留。

代码示例6.30 编译高级语言程序

高级语言程序

```
int f, g, y; // global variables


int main (void)
{
```

MIPS汇编程序

```
.data
f:
g:
y:

.text
main:
  addi  $sp, $sp, -4  # make stack frame
```

```
f = 2;
g = 3;
y = sum (f, g);
return y;
}

int sum (int a, int b) {
 return (a + b);
}
```

```
sw   $ra, 0($sp)   # store $ra on stack
addi $a0, $0, 2    # $a0 = 2
sw   $a0, f        # f = 2
addi $a1, $0, 3    # $a1 = 3
sw   $a1, g        # g = 3
jal  sum           # call sum procedure
sw   $v0, y        # y = sum (f, g)
lw   $ra, 0($sp)   # restore $ra from stack
addi $sp, $sp, 4   # restore stack pointer
jr   $ra           # return to operating sys

sum:
 add  $v0, $a0, $a1 # $v0 = a + b
 jr   $ra           # return to caller
```

步骤2：汇编

汇编器将汇编语言代码转成机器语言代码放入目标文件中。汇编器对汇编代码进行两遍扫描。在第一遍扫描中，汇编器分配指令地址，并且发现所有的符号（symbols）（如标号和全局变量名）。经过第一遍扫描后代码展示如下：

```
0x00400000  main:  addi $sp, $sp, -4
0x00400004         sw   $ra, 0($sp)
0x00400008         addi $a0, $0, 2
0x0040000C         sw   $a0, f
0x00400010         addi $a1, $0, 3
0x00400014         sw   $a1, g
0x00400018         jal  sum
0x0040001C         sw   $v0, y
0x00400020         lw   $ra, 0($sp)
0x00400024         addi $sp, $sp, 4
0x00400028         jr   $ra
0x0040002C  sum:   add  $v0, $a0, $a1
0x00400030         jr   $ra
```

符号的名字和地址保存在符号表（symbol table）中，如表6-4所示。在第一遍扫描后，如果标号的地址已经确定，就将填充对应符号的地址。此时，全局变量在内存的全局数据段中分配存储位置，其中内存起始地址为0x10000000。

表6-4 信号表

符号	地 址
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

在第二遍扫描中，汇编器将产生机器语言代码。全局变量和标号的地址可以从符号表中获得。机器语言代码和符号表存储在目标文件中。

步骤3：链接

许多较大的程序包括不止一个文件，如果程序员只改变其中的一个文件，那么重新编译和汇编其他文件就是一种浪费。实际上，程序总是调用库文件中的函数，而这些库文件几乎是不变的。如果高级语言代码的文件不变，那么与之相联系的目标文件就不需要更新。

链接器的工作是将所有的目标文件合并成一个机器语言文件，即可执行文件（executable）。链接器重新定位目标文件中的数据段和指令段使它们彼此不再相接。它使用符号表中的信息来调整重新定位后的全局变量和标号地址。

我们的例子只有一个目标文件，所以不需要重定位。图6-33展示了可执行文件，此文件有三个部分：可执行文件头、代码段和数据段。可执行文件头包含文件大小（代码大小）和

数据大小（全局定义的数据数目），两者均用字节表示。代码段包含指令序列和相应的存储地址。

可执行文件头	代码大小	数据大小
	0x34 (52字节)	0xC (12字节)
代码段	地址	指令
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E0008
数据段	地址	数据
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, –4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, –4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

图6-33 可执行文件

此图在机器代码旁给出了可读的汇编指令格式以便于理解，但是可执行文件只包括机器代码。数据段给出了每个全局变量的地址。全局变量根据全局指针$gp给出的基地址来寻址。例如，第一条存储指令sw $a0, 0x8000 ($gp)，将数值2存储在全局变量f中，f在内存中的地址为0x10000000。注意：偏移量0x8000是一个16位有符号数值，将此偏移量符号扩展后与基地址$gp相加，即 $gp + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000，此值即为变量f的内存地址。

步骤4：加载

操作系统通过从存储设备（一般是硬盘）读取可执行文件中的代码段将程序读出到内存的代码段中。操作系统将$gp设置为0x10008000（全局数据段的中间），将$sp设为0x7FFFFFFC（动态数据段的顶部），然后执行jal 0x00400000跳转到程序的开头。图6-34展示了程序开始执行时的内存图。

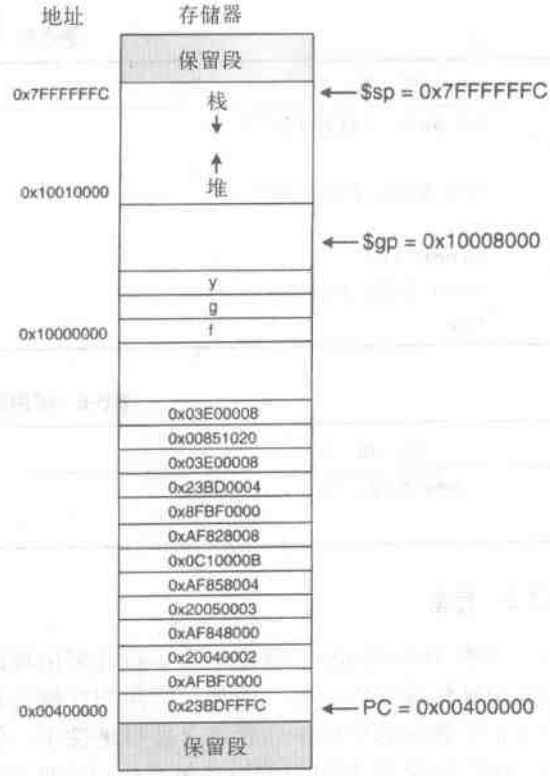


图6-34 可执行程序加载到内存

6.7 其他主题

本节包括了一些本章其他节没有涵盖的内容，主要包括伪指令、异常、有符号和无符号算术指令，以及浮点指令。

6.7.1 伪指令

如果一条汇编指令在MIPS指令集中找不到，很可能是因为可以用一条或者多条已有MIPS指令来实现同样操作。MIPS是一种精简指令集（RISC），要通过保持指令数目最少来降低指令大小和硬件的复杂性。

然而，MIPS定义了伪指令（pseduoinstruction）。伪指令并不是实际指令集的一部分，但常由程序员和编译器使用。在转化为机器代码时，伪指令被翻译成一条或多条MIPS指令。

表6-5给出了伪指令的例子以及MIPS指令怎样实现这些伪指令。例如，装载立即数（li）伪指令向寄存器加载一个32位的常数，它使用lui和ori的组合来实现。乘法伪指令（mul）提供一个三操作数的乘法，两个寄存器中的数值相乘，最低32位结果放入第三个寄存器。无操作伪指令（nop）不做任何操作，执行这条伪指令时，PC每次增加4，不会改变寄存器或内存的值。nop指令的机器代码为0x00000000。

一些伪指令要求一个临时寄存器来保存中间结果。例如，伪指令beq $t2, $imm_{15:0}$, loop将$t2与16位立即数$imm_{15:0}$比较，这条伪指令需要一个临时寄存器来存储16位的立即数，汇编器使汇编器寄存器$at来存储结果，表6-6描述了汇编器如何使用$at来将伪指令转换成真正的MIPS指令。练习6.31将实现循环左移（rol）和循环右移（ror）等伪指令。

表6-5 伪指令

伪 指 令	对应MIPS指令
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`mul $s0, $s1, $s2`	`mult $s1, $s2` `mflo $s0`
`clear $t0`	`add $t0, $0, $0`
`move $s2, $s1`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

表6-6 使用$at的伪指令

伪 指 令	对应MIPS指令
`beq $t2,` $imm_{15:0}$`, Loop`	`addi $at, $0,` $imm_{15:0}$ `beq $t2. $at, Loop`

6.7.2 异常

异常（exception）就像一个不按计划出现的过程调用，可以跳转到一个新的地址。异常可能由硬件或软件引起。例如，当用户在键盘按下键时，处理器可能接收到这个消息，然后停下正在处理的程序来判断哪个按键被按下，保存起来供将来使用，然后恢复当前程序的执行。由诸如键盘之类的I/O设备触发的硬件异常称为中断（interrupt）。比较而言，当程序遇到诸如没有定义指令之类的错误时，程序会转到操作系统（operating system，OS）代码，来决定是否结束终止违规的程序。软件异常有时称为自陷（trap）。其他导致异常的原因包括被0除、

尝试读不存在的内存、硬件故障、调试断点和算术溢出（见6.7.3节）等。

异常发生时，处理器将记录导致异常的原因和当前PC值，然后跳转到异常处理程序（exception handler）。异常处理程序（往往是操作系统的一个部分）主要是检测导致异常的原因，并做出适当的反应（例如在硬件中断时读键盘）。在处理完异常后，将返回异常发生前正在执行的程序。MIPS处理器中，异常处理程序总是起始于0x80000180处。当异常发生时，无论哪种情况引起的异常，处理器总是调转到这个地址。

MIPS体系结构使用称为cause寄存器的专用（special-purpose）寄存器来记录导致异常的原因。不同的导致异常原因具有不同的编码，如表6-7所示。异常处理程序读取cause寄存器来决定怎样处理异常。其他的一些体系结构将根据不同的异常原因跳转到不同的异常处理程序，而不是使用cause寄存器来区分异常原因。

表6-7 异常原因寄存器代码表

异常	寄存器中的值
硬件中断	0x00000000
系统调用	0x00000020
断点/被0除	0x00000024
未定义指令	0x00000028
算术溢出	0x00000030

MIPS体系结构使用称为异常程序计数器（Exception Program Counter，EPC）的专用寄存器来存储异常发生时的PC值。在处理完异常后，处理器返回到EPC中的地址。这与在jal指令执行时使用$ra来存储PC的旧值很相似。

EPC和cause寄存器不属于MIPS寄存器文件。mfc0（move from coprocessor 0）指令将EPC寄存器、cause寄存器或其他专用寄存器复制到一个通用寄存器（general purpose registers）中。协处理器0称为MIPS处理器控制器（MIPS processor control），用于处理中断和诊断处理器错误。例如，代码mfc0 $t0, cause将cause寄存器内容复制到$t0中。

syscall和break指令产生自陷来执行系统调用或者断点调试。异常处理程序使用EPC来找到发生异常的指令，并检查指令字段来决定是进行系统调用或者进入断点。

总之，异常使处理器跳转到异常处理程序。异常处理程序将寄存器内容保存在栈里面，然后使用mfc0指令来检查导致异常的原因，并做出相应的反应。异常处理程序执行完之后，从栈中恢复寄存器的内容，使用mfc0指令从EPC中复制返回地址到$k0，使用jr $k0指令返回。

6.7.3 有符号和无符号的指令

回忆前面提到的二进制数据。它可能是有符号的或者无符号的。MIPS体系结构使用两进制补码表示有符号数。MIPS也有特定的指令用于处理有符号和无符号类型，包括加和减、乘和除、小于设置和半字读出等。

加法和减法

对于有符号数和无符号数而言，其操作相同，但对于结果的解释却不同。

1.4.6节中提到，如果两个大的有符号数相加，结果可能是错误的。例如，两个较大的正数相加，结果却是负的：0x7FFFFFFF+0x7FFFFFFF=0xFFFFFFFE=－2。同样，两个较大的负数相加会产生正的结果：0x80000001+0x80000001=0x00000002。这种情况称为算术溢出（overflow）。

C语言忽略算术溢出，但其他语言（例如Fortran）则要求程序报告溢出。如6.7.2节提到的，MIPS处理器使用异常来处理溢出，异常程序可以决定如何处理溢出（例如，可以使用更高的精度重复计算来避免溢出），然后返回程序停止之处。

MIPS提供有符号和无符号的加减法指令，有符号的指令包括add，addi和sub。无符号指令包括addu，addiu和subu。溢出时，有符号指令出发异常，而无符号指令不发出异常。除此之外，

两类指令基本相同。由于C语言忽略异常，C程序使用MIPS指令中的无符号指令来实现。

乘法和除法

乘法和除法对于有符号和无符号数的操作是不同的。例如，0xFFFFFFFF作为一个无符号的数字时表示一个大数字，但是作为一个有符号数时表示－1。因此，如果是无符号数，0xFFFFFFFF×0xFFFFFFFF的结果为0xFFFFFFFE00000001；如果是有符号数，结果则为0x0000000000000001。

因此，需要对有符号数和无符号数采用不同的乘法和除法指令操作。mult和div对有符号数进行操作，而multu和divu对无符号数进行操作。

小于设置（set less than）

小于设置指令可以比较两个寄存器（slt）或者一个寄存器和一个操作数（slti）的大小。小于设置指令也分为有符号（slt和slti）和无符号（sltu和sltiu）两种形式。对于有符号数的比较，0x80000000小于任意数字，因为它是最小的负数；对于无符号数的比较，0x80000000大于0x7FFFFFFF但是小于0x80000001，因为所有的数字都是正数。

需要注意的是，sltiu将立即数先做符号扩展，然后把它当作无符号数进行操作。例如，sltiu $s0, s1, 0x8042 将$s1和0xFFFF8042进行比较，其中立即数为一个较大的正数。

读出

如6.4.5节所述，字节读出分为有符号（lb）和无符号（lbu）两类。lb对字节数据进行符号扩展，而lbu对字节数据进行0扩展来填充32位寄存器。同样地，MIPS提供有符号和无符号的半字读出指令（lh和lhu），这类指令将两个字节的数据读出到寄存器的低半字，然后进行符号扩展或者0扩展来填充寄存器的高半字。

6.7.4 浮点指令

MIPS体系结构定义一个可选的浮点协处理器，称为1号协处理器（coprocessor 1）。在早期MIPS的实现中，浮点协处理器是一个独立的芯片。如果用户需要快速的浮点计算能力，他们可以单独购买。在大多数现代MIPS实现中，浮点协处理器集成在主处理器旁边。

MIPS定义32个32位的浮点寄存器$f0～$f31。这些寄存器独立于目前使用到的通用寄存器，MIPS支持单精度和双精度的IEEE浮点运算单元，64位的双精度数据存储在两个32位寄存器构成的寄存器对中，所以只有16个双数下标的寄存器（$f0, $f2, $f4, …, $f30）用于指定双精度操作数，如表6-8所示。

表6-8 MIPS浮点寄存器集

名 称	寄存器号	用 途
$fv0-$fv1	0, 2	过程返回值
$ft0-$ft3	4, 6, 8, 10	临时变量
$fa0-$fa1	12, 14	过程参数
$ft4-$ft5	16, 18	临时变量
$fs0-$fs5	20, 22, 24, 26, 28, 30	保存变量

浮点指令的opcode字段均为17（10001_2）。它们需要funct字段和cop（coprocessor）字段来指明指令的具体类型。因此，MIPS为浮点指令定义F-类型（F-type）指令格式，如图6-35所示。浮点指令分为单精度和双精度两类，单精度类指令的cop字段为16（10000_2），双精度类指令的cop字段17

图6-35 F-类型机器指令格式

(10001_2)。如同R-类型指令一样，F-类型指令有两个源操作数fs和ft，一个目的操作数fd。

指令精度根据助记符中的.s和.d来判断。浮点运算指令包括加法（add.s, add.d）、减法（sub.sz, sub.d）、乘法（mul.s, mul.d）、除法（div.s, div.d）、取反（neg.s, neg.d）和取绝对值（abs.s, abs.d）。

浮点操作的分支转移需要两个步骤。首先，比较指令用于设置或清除浮点条件标志（floating-point condition flag, fpcond）；然后，条件分支检测fpcond标志的值。比较指令包括等于（c.seq.s , c.seq.d）、小于（c.lt.s, c.lt.d）、小于等于（c.le.s, c.le.d）。在fpcond为FALSE或TRUE时，对应的条件分支指令分别为bc1f和bc1t。不等、大于等于或大于的判断过程中，首先为seq、lt或le指令，后面跟随bc1f指令。

使用lwc1和swc1指令实现浮点寄存器从内存的读出和写入。这些指令每次搬移32位，因此需要两条指令完成对双精度操作数的存储器访问操作。

6.8 真实世界透视：IA-32结构*

目前几乎所有的个人计算机都在使用IA-32结构的微处理器。IA-32是一个32位体系结构，最初由Intel公司研发，AMD也在销售与IA-32兼容的微处理器。

IA-32体系结构漫长而曲折的历史可以追溯到1978年。当时，Intel推出了16位的8086微处理器，IBM选择了8086和它的姊妹产品8088作为IBM的第一代个人计算机。1985年，Intel公司推出了32位的微处理器80386。它对于8086是向后兼容的，可以运行使用为早期PC开发的软件。兼容80386的处理器体系结构称为IA-32或x86处理器。Pentium，Core和Athlon处理器都是著名的IA-32处理器，7.9节详细介绍了IA-32微处理器的发展。

这些年中，Intel和AMD把更多的新指令和功能都塞进了这个陈旧的体系结构中，其结果就是这种体系结构与MIPS相比很不优雅。如同Patterson和Hennessy所说的那样，“这种混杂的祖谱使得体系结构很难解释，也不可能去喜欢它。”然而，软件的兼容性比技术的优雅性更加重要，所以IA-32在这20年内都是PC事实标准。每年卖出的IA-32处理器超过一亿片，巨大的市场保证了每年50亿美元的研究开发经费来对处理器进行进一步的改进。

IA-32属于复杂指令集计算机（Complex Instruction Set Computer, CISC）体系结构。与诸如MIPS的精简指令集计算机（Reduced Instruction Set Computer, RISC）体系结构相比，每一条CISC指令可以做更多的工作。CISC体系结构上的程序一般需要较少的指令，指令编码更加紧凑。这样可以节省内存，尤其在RAM比较贵时，CISC更有优势。CISC体系结构的指令长度是可变的，一般都会小于32位。但其代价是复杂指令系统更难译码而且指令执行速度更慢。

本节介绍IA-32体系结构，但目标不是让读者成为IA-32的汇编程序员，而是说明IA-32与MIPS的一些相似点和不同点。我们认为了解IA-32的工作方式是一件很有趣的事情，但是本节的内容不妨碍对后面章节的理解。IA-32与MIPS的主要差异见表6-9。

表6-9 MIPS和IA-32的主要差异

特征	MIPS	IA-32
寄存器数目	32个通用寄存器	8个（在用途上有一些限制）
操作数数目	3个（2个源，1个目的）	2个（1个源，1个源/目的）
操作数位置	寄存器或立即数	寄存器、立即数或存储器
操作数大小	32位	8、16或32位
条件码	无	有
指令类型	简单	简单类型和复杂类型
指令编码	固定，4字节长	变长，1～15字节

6.8.1 IA-32的寄存器

8086微处理器提供8个16位的寄存器，其中一些寄存器会独立存取高8位和低8位字节。当32位的80386产生后，这些寄存器扩展为32位，称为EAX、ECX、EDX、EBX、ESP、EBP、ESI和EDI。为了保证向后兼容性，寄存器的低16位和一些低8位部分也是可用的，如图6-36所示。

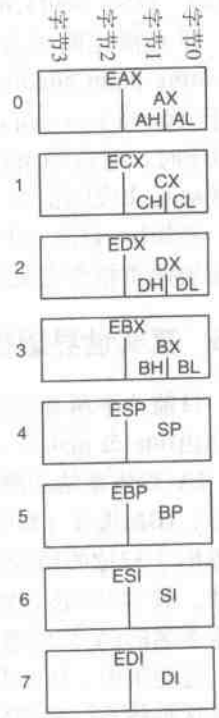


图6-36 IA-32寄存器

这8个寄存器大部分（但并不完全）是通用寄存器。特定的指令不能够使用特定的寄存器。其他指令总是将结果放入特定寄存器中。如同MIPS中的$sp，ESP寄存器在正常情况下用于保存栈指针。

IA-32的程序计数器称为EIP（extended instruction pointer）。如同MIPS中的PC，它从一条指令向下一条指令递增，或者通过分支、跳转或子程序调用指令来改变运行路径。

6.8.2 IA-32的操作数

MIPS指令一般对寄存器或立即数进行操作。需要显式的读出和写入指令完成内存和寄存器之间的数据移动。相对而言，IA-32指令系统可以对寄存器、立即数或者内存进行操作。这对于较少的寄存器集是一种补偿。

MIPS指令一般指定了3个操作数：2个源操作数和1个目的操作数。IA-32指令只指定了2个操作数，第一个操作数是源操作数，第二个操作数既是源操作数又是目的操作数，因此，IA-32指令会将结果覆盖到其中一个源操作数。表6-10列出了IA-32中操作数位置的组合。除了从内存到内存外，所有组合方式都有可能。

表6-10 操作数位置

源/目的	源	例 子	含 义
寄存器	寄存器	add EAX, EBX	EAX<-EAX+EBX
寄存器	立即数	add EAX, 42	EAX<-EAX+42
寄存器	存储器	add EAX, [20]	EAX<-EAX+Mem[20]
存储器	寄存器	add [20], EAX	Mem[20]<-Mem[20]+EAX
存储器	立即数	add [20], 42	Mem[20]<-Mem[20]+42

与MIPS类似，IA-32有字节寻址的32位内存空间，但是IA-32支持更多种内存寻址模式（addressing modes）。内存位置由基址寄存器（base register）、偏移（displacement）和变址寄存器（scaled index register）组合来确定，如表6-11所示。偏移可以是8位、16位或者32位数值。与变址寄存器（index register）相乘的数值可以是1、2、4或8。用于存储器读出和写入的基址+位移模式与MIPS中基址寻址模式相同。变址寻址提供了一个简便方法访问2、4、8字节元素的数组或结构，而不需要多条指令来产生地址。

表6-11 存储器寻址模式

例 子	含 义	解 释
add EAX, [20]	EAX<-EAX+Mem[20]	位移
add EAX, [ESP]	EAX<-EAX+Mem[ESP]	基址
add EAX, [EDX+40]	EAX<-EAX+Mem[EDX+40]	基址＋偏移
add EAX, [60+EDI*4]	EAX<-EAX+Mem[60+EDI*4]	基址＋变址
add EAX, [EDX+80+EDI*2]	EAX<-EAX+Mem[EDX+80+EDI*2]	基址＋偏移＋变址

MIPS总是对32位数进行操作。而IA-32指令可以对8位、16位、32位数据进行操作。详细说明见表6-12。

表6-12 对8位、16位和32位数据操作的指令

例 子	含 义	数据大小
add AH, BL	AH<-AH+BL	8位
add Ax, -1	AX<-AX+oxFFFF	16位
add EAX, EDX	EAX<-EAX+EDX	32位

6.8.3 状态标志

像很多CISC体系结构一样，IA-32使用状态标志（status flags），也称为条件码（condition codes），判断分支转移以及保存进位和算术溢出标志。IA-32使用称为EFLAGS的32位寄存器存储状态标志，EFLAGS寄存器中的一些位由表6-13给出，其他位被用于操作系统。

表6-13 部分EFLAGS标志位

名 称	含 义
CF（进位标志）	由最近一次算术操作引起的进位标志。在无符号算术运算中表示溢出。也用于多精度算术计算中在不同字之间传播进位
ZF（零标志）	最后一个操作的结果为0
SF（符号标志）	最后一个操作的结果为负数（最高位为1）
OF（溢出标志）	在二进制补码算术运算中产生溢出

IA-32处理器体系结构包括了EFLAGS寄存器、前面介绍的8个寄存器和EIP寄存器。

6.8.4 IA-32指令集

IA-32的指令集要比MIPS指令集大，表6-14描述了一些常见的指令，IA-32还有浮点运算指令和多个短数据合成一个长数据的运算指令。D表示目的操作数（寄存器或内存位置），S表示源操作数（寄存器或内存的位置，或者是一个立即数）。

表6-14 部分IA-32指令

指 令	含 义	功 能
ADD/SUB	加法/减法	D=D+S/D=D-S
ADDC	带进位加	D=D+S+CF
INC/DEC	递增/递减	D=D+1/D=D-1
CMP	比较	根据D-S设置标志位
NEG	取反	D=-1
AND/OR/XOR	逻辑AND/OR/XOR	D=D op S
NOT	逻辑NOT	$D=\overline{D}$
IMUL/MUL	有符号/无符号乘法	EDX:EAX=EAX X D
IDIV/DIV	有符号/无符号除法	EDX: EAX/D EAX = 商；EDX = 余数
SAR/SHR	算术/逻辑右移	D=D>>>S/D=D>>S
SAL/SHL	左移	D=D<<S

（续）

指 令	含 义	功 能
ROR/ROL	循环右移/左移	将D循环右/左移S位
RCR/RCL	带进位标志的循环右移/左移	将CF和D循环右/左移S位
BT	测试位	CF=D[S]（D的第S位）
BTR/BTS	测试位并清除/设置	CF=D[S]; D[S]=0/1
TEST	基于屏蔽位设置标志位	根据D和S设置标志位
MOV	数据移动	D=S
PUSH	压栈	ESP=ESP-4; Mem[ESP]=S
POP	出栈	D=MEM[ESP]; ESP=ESP+4
CLC, STC	清除/设置进位标志	CF=0/1
JMP	无条件跳转	相对跳转：EIP=EIP+S 绝对跳转：EIP=S
Jcc	条件跳转	if(flag) EIP=EIP+S
LOOP	循环	ECX=EXC-1 if ECX≠0 EIP=EIP+im
CALL	过程调用	ESP=ESP-4 MEM[ESP]=EIP; EIP=S
RET	过程返回	EIP=MEM[ESP]; ESP=ESP+4

注意：一些指令需要对特定寄存器进行操作，例如，32×32位的乘法操作总是从EAX寄存器中取出源操作数，然后总是把64位结果分别放入EDX和EAX中；LOOP总是将循环次数存储在ECX中；PUSH，POP，CALL和RET使用栈指针寄存器ESP。

条件跳转指令检查标志位，在满足合适情况时跳转。跳转指令有很多种，比如，JZ在零标志位（ZF）为1的时候跳转；JNZ在零标志位为0时跳转。跳转指令一般都是跟在某一条指令之后，例如用于设置标志位的比较指令（CMP）。表6-15列出了一些条件跳转指令，以及这些指令所依赖的先前比较操作设置的标志位。

表6-15 部分分支转移指令

指 令	含 义	比较d之后的功能
JZ/JE	若ZF=1时跳转	若D=S时跳转
JNZ/JNE	若ZF=0时跳转	若D≠S时跳转
JGE	若SF=OF时跳转	若D≥S时跳转
JG	若SF=OF和ZF=0时跳转	若D>S时跳转
JLE	若SF≠OF和ZF=1时跳转	若D≤S时跳转
JL	若SF≠OF时跳转	若D<S时跳转
JC/JB	若CF=1时跳转	
JNC	若CF=0时跳转	
JO	若OF=1时跳转	
JNO	若OF=0时跳转	
JS	若SF=1时跳转	
JNS	若SF=0时跳转	

6.8.5 IA-32指令编码

在数十年的时间内IA-32指令在不断修改，所以其编码非常杂乱。与MIPS指令固定32位长度不同，IA-32指令长度在1～15字节之间变化，如图6-37所示。操作码可能是1个字节、2个字节或者3字节。操作码之后是4个可选择的区域：ModR/M、SIB、偏移和立即数：ModR/M指定寻址模式；SIB指定特定寻址模式下的倍数（scale）、变址寄存器和基址寄存器；“偏移”指定一定寻址模式下的偏移是1字节、2字节还是4字节；“立即数”是作为指令中源操作数的1字节、2字节或4字节立即数。另外，指令还可以加上最长4个字节长度的可选前缀来修改它的行为。

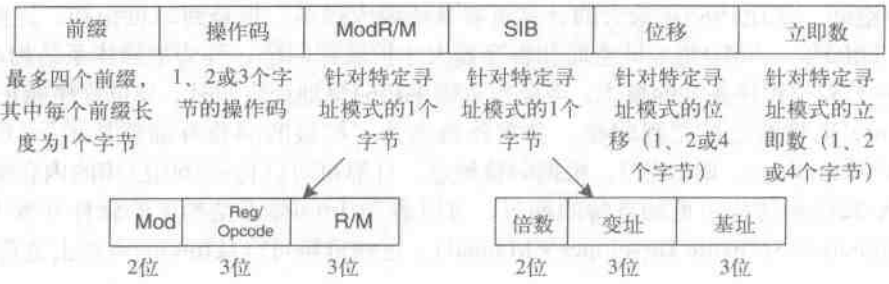


图6-37 IA-32指令编码

ModR/M使用2位的Mod字段和3位的R/M字段来确定其中一个操作数的寻址方式。操作数可以来自于8个寄存器之一，或者来自于24个内存寻址模式之一。根据编码的人为规定，在一定的寻址模式中，ESP和EBP寄存器不能用做基址或变址寄存器。Reg字段用来确定另外一个操作数使用的寄存器。对于不需要第二个操作数的指令，Reg字段用为操作码提供额外3位来共同形成opcode字段。

在使用变址寄存器的寻址方式中，SIB用于确定变址寄存器和倍数（1，2，4，8）。如果使用了基址寄存器和变址寄存器，SIB会指定基址寄存器。

MIPS使用指令的opcode和funct区域来确定指令，IA-32使用可变的数据位来确定不同的指令，它使用较少的位来判断较多的常见指令，减少了指令的平均长度，一些指令甚至有多个opcode域。例如，add AL, imm8 实现将一个8位的立即数累加到AL中，这条指令使用1字节的操作码（0x04），随后是1字节的立即数。寄存器A（AL，AX，EAX）称为累加器（accumulator）。另外，add D, imm8 实现将一个8位的立即数与一个目的操作数D（内存或者寄存器）相加，这条指令使用1个字节的操作码（0x80），随后是1字节以上的操作数D，最后是1字节的立即数。当目的操作数是累加器时，可以缩短许多指令的编码。

在最初的8086系统中，操作码用于判断指令对8位操作数还是16位操作数进行操作。在支持32位操作数的80386出现后，没有新的操作码字段用于判断32位的形式，而原来的操作码字段用于判断16位或32位操作。由操作系统在代码段描述符（code segment descriptor）中指定一位附加位，判断要选择哪种形式的处理器。当此位设为0时选择8086程序，操作码默认对16位操作数进行操作；当此位设为1时，程序默认按照32位操作。另外，程序员可以指定前缀来改变特定指令的形式。如果前缀0x66出现在操作码之前，则使用另一种操作数方式（在32位模式下操作数为16位，或者在16位模式下操作数为32位）。

6.8.6 IA-32的其他特性

80286使用分段（segmentation）方式将内存划分为若干段，其中每段的长度不大于64KB。

当操作系统允许分段时，相对于每段的起始位置计算地址。处理器检查是否超出段末尾的地址，如果是则产生一个错误，从而保护程序不会超出它们自己的段。分段技术引起了很多争议，也没有在现代Windows操作系统各个版本中使用。

IA-32包括了对由字节或字构成的整个字符串进行操作的字符串指令。操作包括移动、比较或对特定值的扫描，在现代处理器中，这些指令经常慢于实现同等操作的简单指令序列，所以这些指令应尽量避免使用。

如同前面提到的，前缀0x66用于选择16位或32位操作数。其他前缀包括：锁定总线（在多处理器系统中控制访问共享变量）；预测分支指令是否执行；在字符串移动中重复执行指令。

任何体系结构的危害都是将内存容量用尽。由于IA-32的地址是32位，IA-32可以访问4GB的内存空间。这比1985年最大的计算机容量还要大很多，但是到21世纪初，这些容量就很有限了。2003年，AMD将地址空间和寄存器大小扩展到64位，称为增强体系结构AMD64。AMD64存在与32位程序兼容的模式，当操作系统采用64位地址空间时，32位程序能正常运行。2004年，Intel也将地址扩充到64位，并重新命名为“扩展的64位存储器技术”（Extended Memory 64 Technology：EM64T）。根据64位地址，计算机可以访问160亿GB的内存空间。

对于IA-32体系结构中更加奇特的细节，可以参考 Intel体系结构上的软件开发人员手册（Intel Architecture Software Developer's Manual）。这些资料可以从Intel的网站上免费得到。

6.8.7 小结

本节讲述了MIPS RISC体系结构和IA-32 CISC体系结构的区别。IA-32倾向于更加短小的程序代码，这是因为其一条复杂指令的功能等同于多条简单的MIPS指令序列，而且指令按照占用内存空间较少的方式编码。然而IA-32体系结构是一个将多年技术聚集在一起的大杂烩，其中有些指令已经不再使用，但是为了保证兼容以前的程序，这些指令必须要保留。IA-32中的寄存器很少，并且指令很难译码，仅仅解释它的指令集就很困难。尽管有以上这些缺点，IA-32也不会轻易改变其主流PC计算机体系结构的地位，这是因为软件兼容性的价值太大以及巨大的市场保护了设计更快IA-32微处理器的努力。

6.9 总结

要指挥一台计算机，必须要说计算机的语言。计算机体系结构定义了如何指挥一个处理器。当前有许多不同的计算机体系结构已经得到广泛应用，但是一旦你了解了其中一种，其他的体系结构学起来就非常快了。在接触一种新体系结构时，需要提出以下关键问题：

- 数据的字长是多少?
- 寄存器如何组织?
- 内存是怎样组织的?
- 指令是怎样的?

MIPS是一个32位的体系结构，因为它对32位的数据进行操作。MIPS体系结构有32个寄存器，原则上讲，几乎所有的寄存器可以被用于任何目的。然而，特定的寄存器习惯上会用于特定目的，这样会使编程更容易，而且由不同程序员写的过程可以更容易地相互通信。例如，0号寄存器$0总是保存常量0，$ra保存jal指令之后的返回地址，$a0-$a3和$v0-$v1分别保存过程的参数和返回值。MIPS有一个32位地址的字节寻址内存。其内存映象由6.6.1节说明。指令长度为32位，而且按照字排列。本章讨论了最常见的MIPS指令。

定义一种体系结构的威力在于针对特定体系结构写出的程序可以运行在这种体系结构的

不同实现上。例如，在1993年的Intel Pentium处理器上所写的程序一般仍然可以在2006年的Intel Core 2 Dou处理器或者AMD Athlon处理器上运行（而且运行得更快）。

在本书的第一部分，我们理解了电路和抽象的逻辑层。本节我们跳到了体系结构层，下一节我们将要学习微体系结构，通过组合对多种电路模块来实现处理器体系结构。微体系结构是连接硬件和软件工程的纽带。而且，我们相信这是所有工程中最让人兴奋的主题之一：你将学会建立你自己的微处理器！

习题

6.1 给出三个依照如下原则设计MIPS体系结构的例子：（1）简单设计有助于规整化；（2）加快常见功能；（3）越小的设计越快；（4）好的设计需要好的折中方法。解释这些例子是如何体现这些特征。

6.2 MIPS体系结构中有一个包含32个32位寄存器的寄存器组。设计一个不包含寄存器组的计算机体系结构是否可能？如果可能，简单描述下此体系结构以及它的指令集。并且这种体系结构与MIPS相比，其优缺点各是什么？

6.3 考虑一个使用字节寻址的存储器，一个32位的字存储在此存储器第42个字的地方。

(a) 第42个字的字节地址是什么？

(b) 第42个字在内存中的地址范围是什么？

(c) 0xFF223344按照大端或小端形式将存储在第42个字中，画出如图6-4那样的示意图。清楚标出与每个字节数据对应的字节地址。

6.4 解释下述代码如何判断计算机是采用了大端还是小端：

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

6.5 下列字符串是用ASCII编码描述的，用十六进制写出结果。

(a) SOS

(b) Cool!

(c) 你自己的名字

6.6 说明习题6.5中的字符串怎样使用大端方式和小端方式存储在字节寻址的存储器中，其起始地址为0x1000100C。使用类似图6-4中的图表，清楚地表示每种方式下每个字节数据的内存地址。

6.7 将下述MIPS汇编代码转化成机器代码，并用十六进制描述。

```
add  $t0, $s0, $s1
lw   $t0, 0x20($t7)
addi $s0, $0, -10
```

6.8 将下面代码转化成机器代码，要求同习题6.7。

```
addi $s0, $0, 73
sw   $t1, -7($t2)
sub  $t1, $s7, $s2
```

6.9 考虑I型指令：

(a) 习题6.8中的哪些指令是I型指令？

(b) 给出（a）中的每条指令中16位立即数符号扩展后形成32位的数据。

6.10 将下列机器代码转化为MIPS的汇编代码，左边的数字是内存中的指令地址，右边的数字给出此地址的指令。然后将这些汇编指令反编译成高级语言程序，并解释这段代码实现了什么功能，其中$a0是输入，其初始值包括一个正数n，$v0是输出。

```
0x00400000   0x20080000
0x00400004   0x20090001
0x00400008   0x0089502a
0x0040000c   0x15400003
0x00400010   0x01094020
0x00400014   0x21290002
0x00400018   0x08100002
0x0040001c   0x01001020
```

6.11 nori指令不在MIPS指令集中，因为使用MIPS指令集中已有的指令可以完成相同的功能。写一个短的汇编代码片段来实现下述功能：$t0 = $t1 NOR 0xF234。使用的指令越少越好。

6.12 使用slt指令完成如下的高级语言代码段，假定整形变量g和h分别存储在寄存器$s0和$s1中。

```
(a) if (g > h)
        g = g + h;
    else
        g = g - h;

(b) if (g >= h)
        g = g + 1;
    else
        h = h - 1;

(c) if (g <= h)
        g = 0;
    else
        h = 0;
```

6.13 使用高级语言写一个程序段 int find42(int array[], int size)，其中size声明数组元素的个数，array声明数组的基地址。此程序返回数组中保存数值42的第一个指针，如果数组中没有42这个数，那么返回－1。

6.14 高级语言代码strcpy将字符串x复制到字符串y中。

```
// high-level code
void strcpy(char x[], char y[]) {
  int i = 0;

  while (x[i] != 0) {
    y[i] = x[i];
    i = i + 1;
  }
}
```

(a) 将strcpy函数用MIPS汇编代码完成，使用$s0存储i。

(b) 画图描述在调用strcpy之前、之中、之后栈的变化，假设在strcpy调用前$sp=0x7FFFFF00。

6.15 将练习习题6.13中的高级语言代码转化成MIPS汇编代码。

6.16 斐波纳契数列（Fibonacci series）中每个数字都是前两个数字之和。表6-16列出了序列中的前几个数fib(n)。

表6-16 斐波纳契数列

n	1	2	3	4	5	6	7	8	9	10	11	…
fib(n)	1	1	2	3	5	8	13	21	34	55	89	…

(a) 当n=0和n=-1时fib(n)是多少?

(b) 使用高级语言写一个fib的函数，对于任意非负值n，返回斐波纳契数列中的第n个值。提示：可能需要使用一个循环。并给你的代码添加注释。

(c) 将（b）中的高级语言代码转化为MIPS汇编代码。在每一行代码后面添加注释来描述代码完成什么功能。使用SPIM模拟器测试你的代码运行fib(9)是否正确。

6.17 考虑如下的MIPS汇编代码，其中proc1，proc2和proc3是非叶过程，proc4是叶过程。每个过程中的代码没有完全描述出来，但注释说明了在每个过程中使用哪些寄存器。

```
0x00401000   proc1:   ...          # proc1 uses $s0 and $s1
0x00401020            jal proc2
 ...
0x00401100   proc2:   ...          # proc2 uses $s2 - $s7
0x0040117C            jal proc3
 ...
0x00401400   proc3:   ...          # proc3 uses $s1 - $s3
0x00401704            jal proc4
 ...
0x00403008   proc4:   ...          # proc4 uses no preserved
                                   # registers
0x00403118            jr $ra
```

(a) 每个过程在栈框架中有多少个字?

(b) 画出proc4调用之后栈的情况，说明寄存器在栈中的具体位置，如果可能的话给出寄存器的值。

6.18 Ben试着编写函数f(a,b)=2a+3b，其中b为非负数。他使用过程调用和递归来实现，并用以下高级语言代码实现过程f和f2。。

```
// high-level code for procedures f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
}

int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x-1);
}
```

然后Ben将这两个函数翻译成如下所示的汇编代码。他写了一个测试过程test来调用f(5,3)。

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j f2: $a0 = x, $s0 = k

0x00400000   test:addi  $a0, $0, 5      # $a0 = 5 (a = 5)
0x00400004        addi  $a1, $0, 3      # $a1 = 3 (b = 3)
0x00400008        jal   f               # call f(5,3)
0x0040000c   loop:j     loop            # and loop forever

0x00400010   f:   addi  $sp, $sp, -16   # make room on the stack
                                        # for $s0, $a0, $a1, and $ra
0x00400014        sw    $a1, 12($sp)    # save $a1 (b)
0x00400018        sw    $a0, 8($sp)     # save $a0 (a)
0x0040001c        sw    $ra, 4($sp)     # save $ra
0x00400020        sw    $s0, 0($sp)     # save $s0
```

```
0x00400024        add   $s0, $a0, $0   # $s0 = $a0 (j = a)
0x00400028        add   $a0, $a1, $0   # place b as argument for f2
0x0040002c        jal   f2             # call f2(b)
0x00400030        lw    $a0, 8($sp)    # restore $a0 (a) after call
0x00400034        lw    $a1, 12($sp)   # restore $a1 (b) after call
0x00400038        add   $v0, $v0, $s0  # $v0 = f2(b) + j
0x0040003c        add   $v0, $v0, $a0  # $v0 = (f2(b) + j) + a
0x00400040        lw    $s0, 0($sp)    # restore $s0
0x00400044        lw    $ra, 4($sp)    # restore $ra
0x00400048        addi  $sp, $sp, 16   # restore $sp (stack pointer)
0x0040004c        jr    $ra            # return to point of call

0x00400050  f2:   addi  $sp, $sp, -12  # make room on the stack for
                                       # $s0, $a0, and $ra
0x00400054        sw    $a0, 8($sp)    # save $a0 (x)
0x00400058        sw    $ra, 4($sp)    # save return address
0x0040005c        sw    $s0, 0($sp)    # save $s0
0x00400060        addi  $s0, $0, 3     # k = 3
0x00400064        bne   $a0, $0, else  # x = 0?
0x00400068        addi  $v0, $0, 0     # yes: return value should be 0
0x0040006c        j     done           # and clean up
0x00400070  else: addi  $a0, $a0, -1  # no: $a0 = $a0 - 1 (x = x - 1)
0x00400074        jal   f2             # call f2(x - 1)
0x00400078        lw    $a0, 8($sp)    # restore $a0 (x)
0x0040007c        add   $v0, $v0, $s0  # $v0 = f2(x - 1) + k
0x00400080  done: lw    $s0, 0($sp)    # restore $s0
0x00400084        lw    $ra, 4($sp)    # restore $ra
0x00400088        addi  $sp, $sp, 12   # restore $sp
0x0040008c        jr    $ra            # return to point of call
```

画出与图6-26类似的栈空间图有助于回答如下问题。

(a) 如果代码由test程序开始，运行到loop时$v0的值是多少？他的程序能正确计算出2a+3b吗？

(b) 假设Ben删掉地址0x0040001C和0x00400044中用来保存和恢复$ra的指令。程序将出现以下哪一种情况？(1)进入无限循环但是没有崩溃；(2)崩溃（导致栈超出PC的动态数据段或者PC跳往程序外部）；(3)当程序返回循环时在$v0中产生一个错误值（如果是这样，值是什么？）；(4)尽管删除了一行，但程序仍能够运行。

(c) 当删掉下列地址存储的指令时，重复（b）中的问题：

(i) 0x00400018和0x00400030（保存和恢复$a0的指令）

(ii) 0x00400014和0x00400034（保存和恢复$a1的指令）

(iii) 0x00400020和0x00400040（保存和恢复$s0的指令）

(iv) 0x00400050和0x00400088（保存和恢复$sp的指令）

(v) 0x0040005C和0x00400080（保存和恢复$s0的指令）

(vi) 0x00400058和0x00400084（保存和恢复$ra的指令）

(vii) 0x00400054和0x00400078（保存和恢复$a0的指令）

6.19 将以下beq指令、j指令和jal指令的汇编指令代码转化为机器码。每条指令的指令地址在左侧给出。

(a)

```
0x00401000            beq $t0, $s1, Loop
0x00401004             . . .
0x00401008             . . .
0x0040100C     Loop:   . . .
```

(b)

```
    0x00401000            beq $t7, $s4, done
     . . .                . . .
    0x00402040     done:  . . .
```

(c)

```
    0x0040310C     back:  . . .
     . . .                . . .
    0x00405000            beq $t9, $s7, back
```

(d)

```
    0x00403000            jal proc
     . . .                . . .
    0x0041147C     proc:  . . .
```

(e)

```
    0x00403004     back:  . . .
     . . .                . . .
    0x0040400C     j      back
```

6.20 考虑下述MIPS汇编语言片段。左侧是每条指令的地址。

```
0x00400028        add  $a0, $a1, $0
0x0040002c        jal  f2
0x00400030 f1:    jr   $ra
0x00400034 f2:    sw   $s0, 0($s2)
0x00400038        bne  $a0, $0, else
0x0040003c        j    f1
0x00400040 else:  addi $a0, $a0, -1
0x00400044        j    f2
```

(a) 将以上的代码序列译成机器代码，并用十六进制写出。

(b) 在每一行代码之后列出其寻址模式。

6.21 考虑下述C语言代码片段。

```
// C code
  void set_array(int num) {
   int i;
   int array[10];

   for (i = 0; i < 10; i = i + 1) {
    array[i] = compare(num, i);
   }
  }
 int compare(int a, int b) {
   if (sub(a, b) >= 0)
    return 1;
   else
  return 0;
}
int sub (int a, int b) {
 return a - b;
}
```

(a) 使用MIPS汇编语言改写C代码片段。使用$s0保存变量i，确保正确使用栈指针。数组存储在栈中，此栈由set_array函数创建（见6.4.6节）。

(b) 假定set_array函数是第一个被调用的函数，画出调用set_array之前和执行每个函数调用时栈的情况，指出存储在栈中寄存器和变量的名字并且记录$sp指针的位置。

(c) 如果$ra没有保存在栈中，代码会完成什么功能？

6.22 考虑下述程序：

```
// high-level code
 int f(int n, int k) {
  int b;
```

```
  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

(a) 将这段高级语言编写的代码译成MIPS汇编代码，特别注意程序调用中保存和恢复寄存器以及使用MIPS保护寄存器的惯例，为你的代码做注释。可以使用MIPS中的mult、mfhi和mflo指令。程序开始于指令地址0x00400100，局部变量b保存在$s0中。

(b) 手工一步步地执行（a）中的程序以计算f(2, 4)，画出与图6-26(c)类似的栈空间图，写出栈中保存的寄存器名和变量值并且给出栈指针$sp的变化过程。在执行过程中，你将会发现跟踪$a0、$a1、$v0和$s0中的值很有用，假定调用f时，$s0 = 0xABCD，$ra = 0x400004。$v0的最终值是多少？

6.23 MIPS中beq和bne等分支转移指令的跳转地址范围有多少？答案以与分支转移指令相关的指令数表示。

6.24 下述问题考察跳转指令j的局限性。按照与跳转指令相关的指令数表示。

(a) 在最坏的情况下，j指令可以向前跳多远（跳向更高的地址）？（最坏的情况是指跳转指令不能向前跳转的情况），使用语言和例子分别解释。

(b) 在最好的情况下，j指令可以向前跳多远？（最好的情况是指跳转指令可以跳转最远）。并解释。

(c) 在最坏的情况下，j指令可以向后跳多远（跳往更低的地址）？并解释。

(d) 在最好的情况下，j指令可以向后跳多远？并解释。

6.25 解释为什么跳转指令j和jal中有一个大的地址空间字段addr。

6.26 使用汇编代码写一段程序，从第一条指令跳转到第64M条指令，其中1M指令 = 2^{20} = 1,048,576条指令。假定代码开始于0x00400000地址处，使用的指令条数越少越好。

6.27 使用高级语言写一段程序，将一个存储在小端格式下的10个32位整型数组转化成大端格式。写完高级语言代码之后，将之转化成MIPS汇编代码，并给你的代码加上注释，使用的指令条数越少越好。

6.28 考虑两个字符串：string1和string2。

(a) 使用高级语言写一个函数concat将两个字符串连接起来：void concat(char[] string1, char[] string2, char[] stringconcat)。程序不需要返回值，它将string1和string2连接起来取代stringconcat的值。假定stringconcat足够长来容纳连接起来的字符串。

(b) 将（a）中的代码转化为MIPS的汇编代码。

6.29 写一个MIPS汇编代码将存储在$s0和$s1中的两个单精度浮点相加。不要使用任何的MIPS浮点指令，不必考虑为特殊值（例如：0，NANs，INF）或者产生上溢或下溢。使用SPIM模拟器测试代码，证明代码功能的可靠性。

6.30 说明如下MIPS代码如何读出内存及执行。

```
# MIPS assembly code
main:
  lw $a0, x
  lw $a1, y
  jal diff
  jr $ra
diff:
  sub $v0, $a0, $a1
  jr $ra
```

(a) 首先说明每条汇编指令的指令地址。

(b) 画出符号表来描述符号的标志和它们的地址。
(c) 将所有的指令转化为机器代码。
(d) 数据段和地址段分别有多少字节?
(e) 画出内存草图来描述数据和指令存储在哪里?

6.31 用MIPS指令来实现下述伪指令，可以使用汇编器寄存器$at，但是不可以误用（覆盖）其他寄存器。

(a) beq $t1, $imm_{31:0}$ L
(b) ble $t3, $t5, L
(c) bgt $t3, $t5, L
(d) bge $t3, $t5, L
(e) addi $t0, $2, $imm_{31:0}$
(f) lw $t5, $imm_{31:0}$ ($s0)
(g) rol $t0, $t1, 5 (将$t1左移5位，然后将结果放入$t0中)
(h) ror $s4, $t6, 31 (将$t6右移31位，然后将结果放入$s4中)

面试问题

6.1 写一段MIPS的汇编代码，交换两个寄存器$t0和$t1中的内容，但不允许使用其他寄存器。
6.2 假设给定一个存有正数和负数的整型数组，写一段MIPS汇编代码找出两个数组元素和最大的元素。假定数组的基地址存在$a0中，数组长度存在$a1中，代码的最终结果起始地址放在$a2中，编写代码，使之运行越快越好。
6.3 数组里保存着一个C语言的字符串，设计算法来反转字符串并将新字串存储在原来的数组中。使用MIPS汇编代码完成算法。
6.4 设计算法来计算一个32位数字中‘1’的个数，使用MIPS汇编代码完成。
6.5 编写MIPS汇编代码完成一个寄存器中数据各个位的翻转，使用的指令越少越好，假设寄存器是$t3。
6.6 编写MIPS汇编代码测试$t2和$t3相加时是否有溢出，使用的指令越少越好。
6.7 设计算法测试给定的字串是否为回文（回文就是从前面读取和从后面读取是一样的，例如wow和racecar就是回文）。使用MIPS汇编代码完成算法。

第7章 微 结 构

7.1 引言

本章中，我们不仅将学习如何构造一个MIPS微处理器，而且还将仔细讨论三种不同的实现方案。这三种方案在性能、成本和复杂度上具有不同的折中。

在一开始的时候，设计一个微处理器就像魔术一样难以琢磨。但是实际上它是相对直观的，而且通过前面的学习，我们已经具备了所需的基本知识。具体来说，前文介绍了组合逻辑和时序逻辑设计方法以实现给定功能和满足特定时序要求。我们也熟悉了算术单元电路和存储器电路，而且已经掌握了MIPS处理器体系结构，从程序员的角度了解了MIPS处理器中的寄存器、指令和存储器等概念。

本章将主要介绍用于连接逻辑和体系结构的桥梁——微处理器微结构（microarchitecture）。微结构是将寄存器、ALU、有限状态机，存储器和其他逻辑模块等多种基本单元组合在一起，以实现微处理器体系结构。同一个微处理器结构（例如MIPS）可以有不同的微结构，从而在性能、成本和复杂性方面取得不同的折中。它们可以运行相同的程序，但是内部设计差异很大。本章中将设计三种不同的微结构以说明这些折中的差异。

本章的内容大部分基于David Patterson和John Hennessy所著《Computer Organization and Design》中的经典MIPS设计。他们慷慨地分享了精巧的设计，这些设计能够充分展示出一个真实的商业处理器体系结构，同时相对简单且易于理解。

7.1.1 体系结构状态和指令集

正如前面所讨论的，计算机体系结构包括指令集和体系结构状态（architecture state）。MIPS处理器中的体系结构状态包括程序计数器和32个寄存器。任何一个MIPS微结构都必须包含这些状态。基于当前状态，处理器基于特定的数据集执行一条特定指令后将产生新的体系结构状态。一些微结构包含附加的非体系结构状态（nonarchitecture state）以简化逻辑或改善性能；我们将在遇到它们的时候继续讨论。

为了使得微结构易于理解，我们仅考虑MIPS指令集的一个子集，主要包括：

- R类型算术/逻辑指令：add, sub, and, or, slt。
- 存储器访问指令：lw, sw。
- 分支指令：beq。

在构造了处理这些指令的微结构后，我们将扩展结构以处理addi和j指令。之所以选择这些特殊的指令，是因为通过它们就可以写出一些有趣的程序。一旦理解了如何实现这些指令，就可以进一步扩展硬件以实现其他指令。

7.1.2 设计过程

可以将微结构分为两个互相关联的部分：数据路径（datapath）和控制（control）。数据路径对数据以字为单位进行操作，包含了存储器、寄存器、ALU和多路开关等结构。MIPS是一个32位结构，因此我们使用32位的数据路径。控制单元从数据路径接收当前指令，并控制

数据路径如何执行这条指令。控制单元往往通过产生多路开关选择、寄存器使能、存储器写入等信号来控制数据路径的操作。

设计复杂系统时，一个好的方法是从包含存储组件的硬件结构开始。这些组件包括存储器和体系结构状态（程序计数器和寄存器）。接着，在这些存储组件之间增加组合逻辑以从当前状态计算新的状态。从部分存储器中读取指令，存储器访问指令则对另外一部分存储器进行读或者写操作。因此，为方便起见，往往将存储器分为两个部分，一部分包含指令，一部分包含数据。图7-1中给出了四个存储组件：程序计数器、寄存器文件、指令存储器和数据存储器。

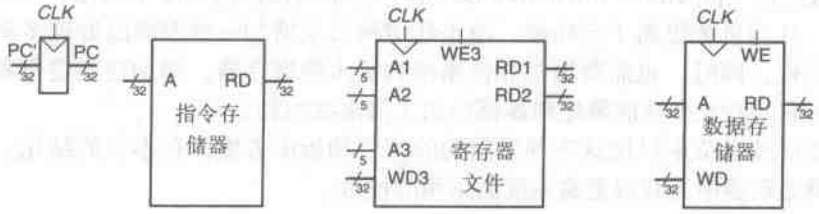


图7-1 MIPS处理器的存储组件

在图7-1中，粗线表示32位数据总线，中等粗细的线表示较窄的总线（例如寄存器文件的5位地址总线）。细线表示控制信号，例如寄存器文件的写使能。我们在本章中使用这些约定以避免数据总线宽度显得杂乱无章。而且存储组件都有复位信号以让它们进入到已知的初始状态。为了避免电路图杂乱，这些复位信号就不再画出。

程序计数器（program counter）是一个普通的32位寄存器。其输出PC指向当前地址，输入PC′表示下一条指令地址。

指令存储器（instruction memory）有一个读出端口，包括32位指令地址输入A，能从当前地址读出32位数据（指令）到数据输出端口RD。

32单元的32位寄存器文件（register file）有两个读出端口和一个写入端口。两个读出端口分别具有5位的地址输入A1和A2，用于指定$2^5 = 32$个寄存器中的一个作为操作数，可以将32位寄存器的值分别输出到RD1和RD2上。写入端口具有5位地址输入A3，32位数据输入WD，写入使能WE3和时钟信号CLK。如果写入使能为1，则寄存器文件将在时钟上升沿对特定寄存器写入数据。

数据存储器（data memory）有一个读写端口。如果写使能WE为1，则数据WD在时钟上升沿写入到地址A指定的单元中。如果写使能为0，则从地址A读出数据到RD。

指令存储器、寄存器文件和数据存储器在读出过程中都呈现组合逻辑特征。换句话说，如果地址发生改变，新的数据在若干传播延迟后就将在RD上出现，而不需要时钟参与。而写入过程仅仅在时钟上升沿发生。在此模式下，系统仅仅在时钟沿才改变状态。地址、数据和写使能必须在时钟沿前建立，而且必须在时钟沿后的保持时间内稳定。

由于存储组件仅在时钟上升沿发生变化，因此是同步时序电路。微处理器由时钟驱动的存储组件和组合逻辑构成，因此也是同步时序电路。而且处理器可以看做一个巨大的有限状态自动机，或者是若干简单而相互交互的有限状态自动机的组合。

7.1.3 MIPS微结构

本章中，我们将研究三种MIPS处理器微结构：单周期、多周期和流水线。它们的区别在于存储组件的连接方式和非体系结构存储组件的数量。

单周期微结构（single cycle microarchitecture）在一个周期中执行一条指令。该结构易于解释，而且控制单元简单。由于它在一个周期内完成操作，所以不需要其他的非体系结构存储组件。然而，时钟周期由最慢的指令决定。

多周期微结构（multicycle microarchitecture）利用多个较短的周期实现一条指令。简单指令的执行周期数较少。而且多周期微结构可以通过对加法器和存储器等复杂硬件部件的复用减少硬件成本。例如，同一个加法器可以在一条指令的不同时钟周期中用于不同目的。多周期微结构需要增加一些非体系结构寄存器以保存中间结果。多周期处理器在任意时刻仅执行一条指令，但是每条指令需要多个周期。

流水线微结构（pipelined microarchitecture）将单周期微结构流水线化，使得可以同时执行多条指令，从而显著提高了吞吐率。流水线结构必须增加一些逻辑以处理多条正在执行指令之间的相关性。同时，也需要增加非体系结构流水线寄存器。增加这些逻辑和寄存器是值得的，当前所有的商业高性能微处理器都使用了流水线结构。

我们将在后续章节中讨论这三种微结构的细节和折中方案。在本章的结尾，将简要介绍现代高性能微处理器中为获得更高速度而采用的技术。

7.2 性能分析

正如前面所提到的，一个微处理器体系结构可以有不同的微结构，以在成本和性能方面取得不同的折中。成本取决于硬件所需的数量和实现工艺。在同样的价格下，CMOS工艺每年都可以在一颗芯片上集成更多的晶体管，而处理器可以利用这些新增加的晶体管来提升性能。精确的成本计算需要实现工艺的专业知识，但是一般而言，越多的门和存储器就需要越多的成本。本节将主要为分析性能奠定基础。

有很多方法可以测量计算机系统的性能，而市场销售部门往往选择一些让处理器看起来更快的方法来测量，而不管这些测量是否和真实性能相关。举个例子，Intel公司和AMD公司同时销售支持IA-32体系结构的微处理器。在上个世纪90年代末和本世纪初，Intel公司的Pentium 3和Pentium 4处理器因其主频较高而大做广告。然而，其竞争对手AMD公司的Athlon处理器在同样主频时性能却优于Intel公司的产品。消费者该如何选择？

测量性能最直接的方法就是直接测试用户所需要程序的执行时间：执行时间较短的计算机系统性能更高。在用户还没有完成自己的程序或者没有人为用户测量性能时，另一个好的方法是测量一组类似用户应用的程序集合的运行时间总和。这样的一组测量程序称为测试基准程序（Benchmark），而这些程序的执行时间往往会公开以体现处理器的性能。

一个程序的执行时间（以秒为单位）如式（7-1）所示：

$$\text{程序执行时间} = \text{指令数} \times \left(\frac{\text{周期数}}{\text{指令}}\right) \times \left(\frac{\text{秒}}{\text{周期数}}\right) \tag{7-1}$$

程序中所包含的指令数取决于处理器体系结构。一些体系结构具有复杂指令，可以在一条指令中执行更多的功能，因此可以减少程序中的指令数目。然而，这些复杂指令往往硬件实现比较慢。指令数往往在很大程度上也取决于程序员的聪明才智。对于本章而言，我们假设执行的都是已有的MIPS程序，因此每个程序的指令数据都是固定的，与微结构无关。

每条指令所需要的周期数（cycles per instruction, CPI）是平均执行一条指令所需要的周期数。它是吞吐率（instructions per cycle, IPC）的倒数。不同微结构具有不同的CPI值。本章中，我们假设采用了理想的存储器系统而不会影响CPI。在第8章中，我们将发现处理器有时需要等待存储器，从而使CPI增加。

每个周期所需要的时间T_c为时钟周期。时钟周期由处理器中关键逻辑路径决定。不同微结构有不同的时钟周期。逻辑和电路设计也在很大程度上影响着时钟周期。例如，先行进位加法器的速度较行波进位加法器更快。同时制造工艺的进步使得晶体管速度每4～6年提高一倍，这样即使微结构和逻辑不发生变化，现在实现的微处理器速度也远比十年以前实现的要快。

微结构设计面临的挑战在于选择合适的设计以最大程度上减小程序执行时间，同时满足成本和功耗方面的限制。由于微结构对CPI和T_c都有很大影响，而且又受到逻辑和电路设计的影响，因此决定最优的设计需要精细的分析。

计算机系统中有很多部分都对整体性能有影响。例如硬盘、存储器、图形处理、网络连接等方面都将成为与处理器无关的性能约束因素。现实世界中最快的处理器也不能帮助提升拨号连接的网络速度。但是这些因素已经超出了本书讨论的范围。

7.3 单周期处理器

我们首先设计在单个周期中执行指令的MIPS微结构。设计过程将从以组合逻辑连接图7-1中的各存储组件开始，这些连接构成了数据通路以执行不同的指令。控制信号决定了给定时刻内需要由数据通路执行的特定指令。控制器包括根据当前指令产生合适控制信号的组合逻辑。本节的最后将分析单周期处理器的性能。

7.3.1 单周期数据路径

本节将一次增加图7-1中的一个组件，以逐步构成单周期数据路径。新的连接将以黑体突出显示。

程序计数器（PC）寄存器包含当前执行指令的地址。第一步是从指令存储器中读取指令。图7-2中显示了PC简单地连接到指令存储器的地址输入。指令存储器取出（fetch）的32位指令标为Instr。

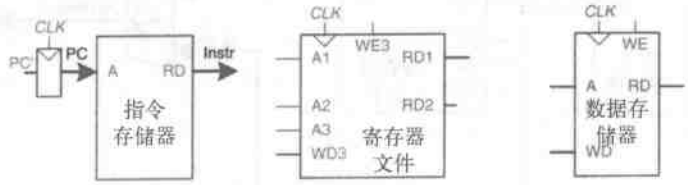


图7-2 从存储器中取指令

处理器的动作取决于取出的特定指令。首先，我们将针对lw指令构造数据路径。然后，将在此基础上考虑构造处理其他指令的数据路径。

对于lw指令，下一步是读取包含了基地址的源寄存器。这个寄存器由指令[25:21]位中的rs字段标识。指令中的相关位连接到寄存器文件中一个读出端口的地址A1上，如图7-3所示。寄存器文件读出的寄存器值由RD1给出。

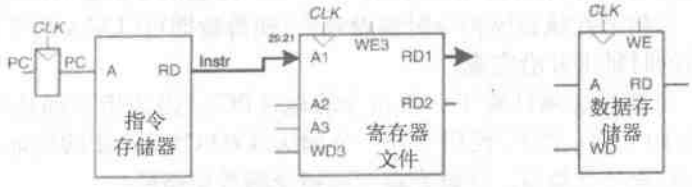


图7-3 从寄存器文件中读出源操作数

lw指令还需要一个偏移量，由指令[15:0]位中的立即数字段给出。由于16位立即数可能为正数或负数，因此必须将符号扩展为32位，如图7-4所示。32位符号扩展值称为SignImm。回忆1.4.6节中符号扩展只是简单地将较短输入的符号位（最高位）直接连接到较长输出的高位。此处，$SignImm_{15:0} = Instr_{15:0}$，$SignImm_{31:16} = Instr_{15}$。

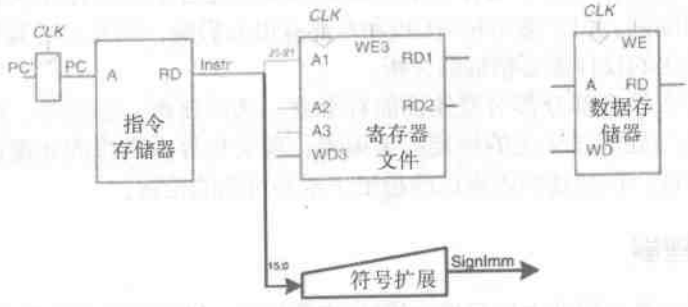


图7-4 立即数的符号扩展

处理器必须将基地址和偏移量相加得到读存储器的地址。图7-5引入一个ALU以执行加法。这个ALU接收两个操作数SrcA和SrcB，其中SrcA来自于寄存器文件，SrcB来自于符号扩展的立即数。如5.2.4节所述，ALU可以执行多种操作。其中3位控制信号ALUControl决定了操作类型。ALU产生32位的操作结果ALUResult和用以表示ALUResult是否为0的零标志Zero。对于lw指令，ALUControl信号应该设置为010，以完成基地址和偏移量的加法。对于读出指令，ALUResult将连接到数据存储器的地址，如图7-5所示。

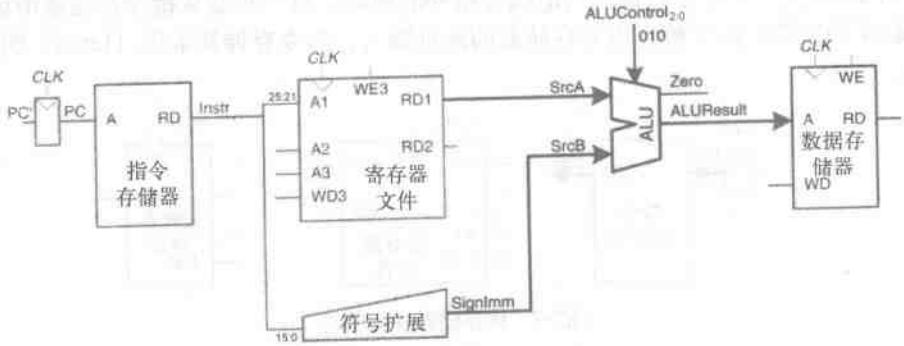


图7-5 存储器地址的计算

从数据存储器中读取的数据将出现在ReadData上，并将在时钟周期结束时写入到寄存器文件中的目的寄存器，如图7-6所示。寄存器文件中的端口3是写入端口。lw指令目的寄存器由指令中[20:16]位的rt字段决定，因此该部分连接到寄存器文件端口3的地址输入A3。ReadData则连接到寄存器文件端口3的数据输入WD3。控制信号RegWrite连接到端口3的写入使能输入端WE3，该信号在执行lw指令时被设置为1使得数据可以写入到寄存器文件。写入过程在时钟周期最后的时钟上升沿完成。

指令执行时，处理器必须计算下一条指令的地址PC'。因为指令都是32位，即4个字节，因此下一条指令位于PC+4。图7-7使用另外一个加法器对PC加4。新的地址将在下一个时钟周期的上升沿被写入到程序计数器。这就完成了lw指令的数据路径。

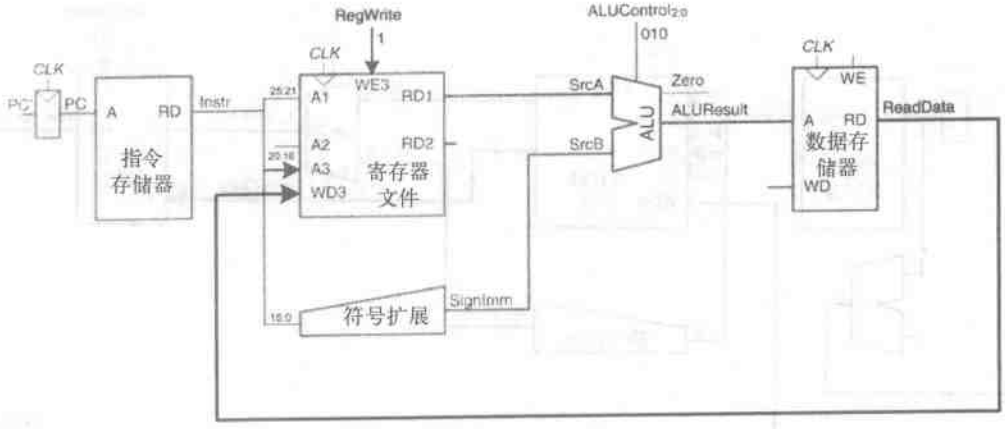


图7-6 向寄存器文件写入数据

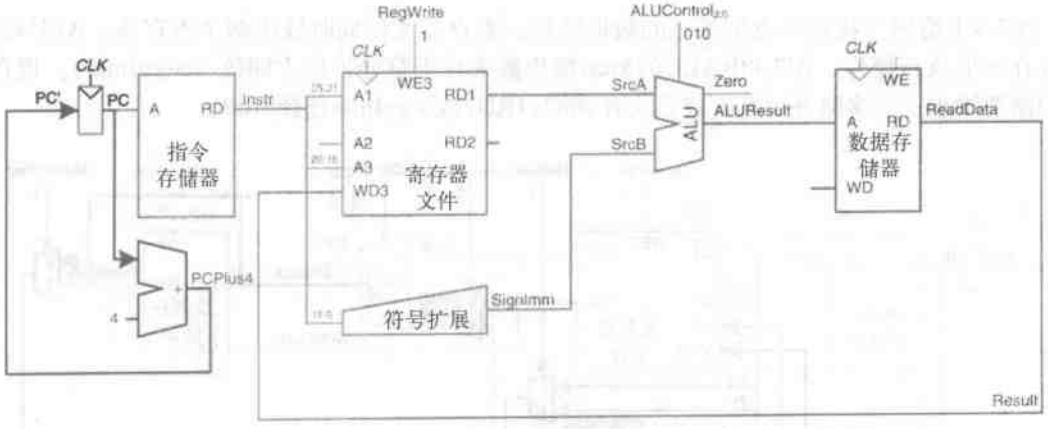


图7-7 决定PC的下一个地址

下一步将扩展数据路径以处理sw指令。与lw指令类似，sw指令从端口1读出基地址，从立即数得到符号扩展的偏移量。ALU将两者相加获得存储器访问地址。所有这些功能都已经由数据路径支持。

sw指令还需要从寄存器文件中读出另一个寄存器值并写入到存储器文件中。图7-8给出了实现这个功能的连接。写入数值的寄存器由指令中[20:16]位的rt字段确定。指令中的这些位将连接到寄存器文件的第2个读端口地址A2。寄存器值将从RD2端口上读出。这可以连接到数据存储器的写入数据端口。同时，数据存储器的写入使能端口WE由MemWrite控制。对于sw指令，MemWrite＝1，向存储器写入数据；ALUControl＝010，将基地址和偏移量相加；RegWrite＝0，不写入寄存器文件。需要注意的是，数据仍然将根据读出地址从数据存储器中读出，只是当RegWrite=0时ReadData被抛弃。

下一步考虑执行R-类型指令add, sub, and, or和slt的数据路径。这些指令将从寄存器文件中读取两个操作数，然后执行一些ALU操作，并将结果写回到寄存器文件中。它们的区别仅仅在于特定的ALU操作。因此，它们可以由同一个硬件结构实现，而仅仅使用不同的ALUControl信号。

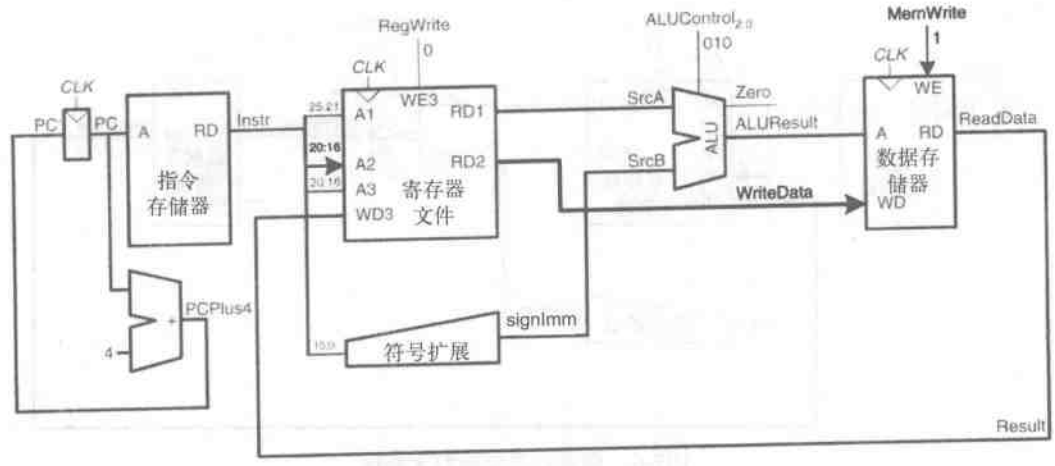


图7-8 sw指令中将数据写入到存储器

图7-9中给出了执行R-类型指令的数据路径。寄存器文件同时读出两个寄存器。ALU对两个寄存器值执行操作。图7-8中ALU的SrcB操作数来自于符号扩展立即数（SignImm）。现在，我们需要增加一个多路开关从寄存器文件读端口RD2或SignImm选择SrcB。

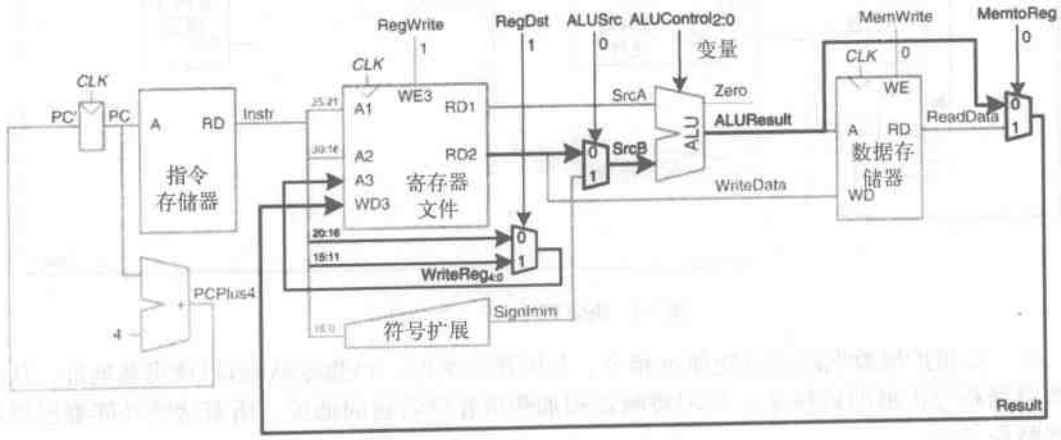


图7-9 对R-类型指令的数据路径

多路开关由一个新的信号ALUSrc控制。ALUSrc为0时，SrcB从寄存器文件中取出数据，以执行R-类型指令；为1时，从SignImm选择以执行lw和sw指令。这种通过增加多路开关选择几个可能输入的数据路径扩展方法非常有用。而且，我们将两次使用这种方法以完成R-类型指令的处理。

图7-8中寄存器文件从数据存储器中获取要写入的数据。然而，R-类型指令需要将ALUResult写入到寄存器文件。因此，我们增加另外一个多路开关Result以选择ReadData和ALUResult，其输出称为Result。这个多路开关由另外一个新的信号MemtoReg来控制。当MemtoReg为0时，选择ALUResult作为Result，以支持R-类型指令；为1时，选择ReadData，以支持lw指令。sw指令不需要写入寄存器文件，所以在执行指令时不关心MemtoReg信号。

最后，我们扩展数据路径以支持beq指令。beq指令将比较两个寄存器。如果两个寄存器相等，则在当前PC值上加上转移偏移量而进行跳转。注意到偏移量存放在指令Instr[15:0]的imm字段，可以为正数或者负数。偏移量表明了跳转目标指令和当前指令之间的指令数。因

此，立即数需要进行符号扩展而且要乘以4，以获得新的PC值：PC′ = PC+4+SigImm × 4。

图7-10给出了对数据路径的修改。用于转移的下一PC值PCBranch由SigImm左移2位并加上PCPlus4得到。左移2位是实现乘以4的简单方法，因为固定位数移位仅仅只是将线重排。两个寄存器通过ALU的减法操作SrcA−SrcB进行比较。如果ALUResult为0，即ALU的Zero标志有效，则两个寄存器相等。通过增加一个多路开关的方法从PCPlus4和PCBranch中进行选择。如果为分支指令，而且Zero标志有效，则PCBranch被选中。因此，当前指令为beq指令时设置Branch信号为1，其余情况为0。对于beq指令，ALUControl = 110，使得ALU完成减法操作；ALUSrc = 0，从寄存器文件中选择ScrB；因为转移指令不需要写入寄存器文件或存储器，所以RegWrite和MemWrite均为0。由于不需要写入寄存器，所以我们不需要关心RegDst和MemtoReg的值。

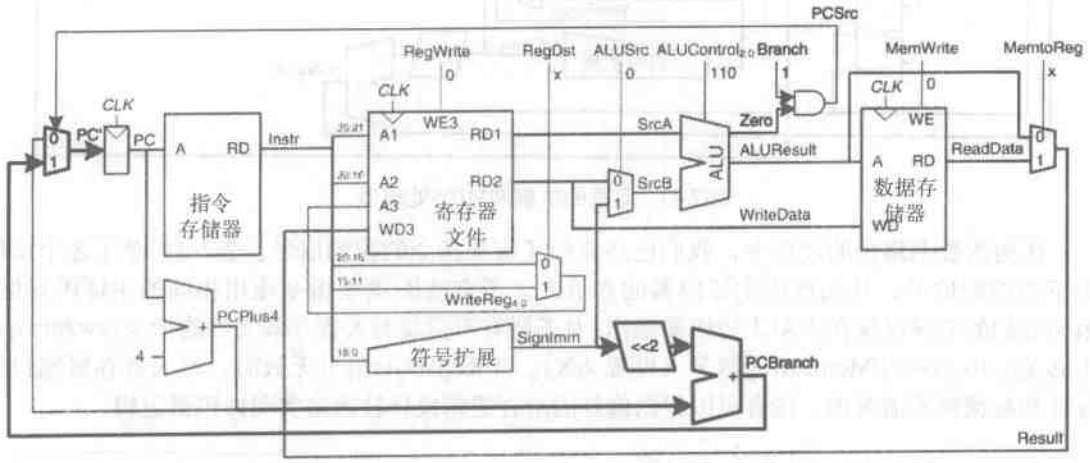


图7-10 beq指令的改进的数据路径

此时，我们已经完成了单周期MIPS处理器的数据路径设计。这里不仅仅是介绍设计本身，更重要的是其设计过程。在这个设计过程中，首先明确存储组件，然后系统地加入连接这些存储组件的组合逻辑。下一节中，我们将考虑如何计算控制信号，以指导数据路径的操作。

7.3.2 单周期控制

控制单元基于指令中的opcode字段（31到26位）和funct字段（5到0位）计算控制信号。图7-11给出了整个单周期MIPS处理器结构图，其中控制单元已经连接到数据路径。

大多数控制信号来源于opcode字段，但是R-类型指令也使用funct字段决定ALU的操作。这里我们将控制单元分成两个组合逻辑部分以简化设计，如图7-12所示。主译码器从opcode中计算大部分的输出信号，同时也产生一个两位的ALUOp信号。ALU译码器使用ALUOp信号并结合funct字段计算ALUControl。其中ALUOp信号的含义如表7-1所示。

表7-1 ALUOp编码

ALUOp	含义	ALUOp	含义
00	加法	10	依赖于funct字段
01	减法	11	无定义

表7-2为ALU译码器真值表。其中3位ALUControl信号的含义已在表5-1中给出。由于ALUOp不可能为11，因此真值表中不关心X1和1X，并以此替代01和10，从而简化逻辑。当ALUOp为00或01时，ALU应该为加法或减法。当ALUOp为10时，译码器将检查funct字段以确定ALUControl信号。注意到我们实现的R类型指令中，funct字段的最高两位总为10，因此可以忽略它们以简化设计。

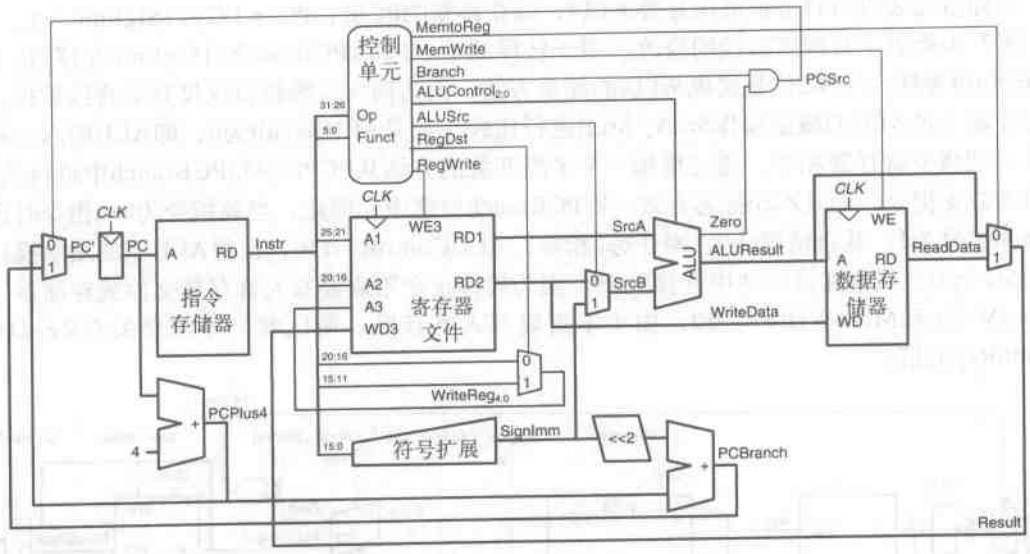


图7-11 完整的单周期MIPS处理器

在构造数据路径的过程中，我们已经介绍了每条指令的控制信号。表7-3汇总了各个操作码下的控制信号，从而得出主译码器的真值表。所有的R-类型指令使用相同的主译码器值，因为它们的差异仅仅在于ALU译码器输出。对于所有不需要写入寄存器文件的指令(sw和beq)，不必关心RegDst和MemtoReg信号（设置为X）。当RegWrite信号无效时，写入寄存器端口的地址和数据都无需考虑。读者可以根据偏好的组合逻辑设计技术来实现译码器逻辑。

表7-2 ALU译码器真值表

ALUOp	funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

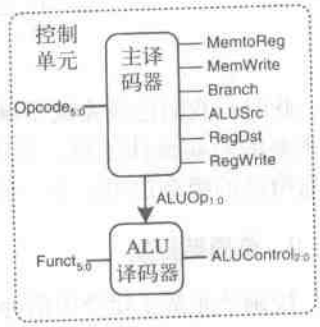


图7-12 控制单元内部结构

表7-3 主译码器的真值表

指令	操作码	RegWrite	RegDst	ALUsrc	Branch	MemWrite	MentoReg	ALUOp
R-类型	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

例7.1 单周期处理器操作

当执行or指令时，请确定控制信号的值和数据路径的实际通路。

解：图7-13给出了执行or指令时的控制信号和数据流。PC指向当前指令的存储器位置，指令存储器取出该指令。

经过寄存器文件和ALU的主要数据流由虚线画出。根据指令中的Instr[25:21]和Instr[20:16]读出寄存器文件中的操作数。SrcB应该来源于寄存器文件的第2个端口（而非SignImm），所以ALUSrc必须为0。or指令为R类型指令，因此ALUOp为10，表示由funct字段确定ALUControl信号为001。结果从ALU中读出，因此MemtoReg为0。结果写入到寄存器文件，因此RegWrite为1。指令不需要写入存储器，因此MemWrite为0。

目的寄存器的选择也由虚线画出。目的寄存器由rd字段（Instr[15:11]）确定，因此RegDst=1。

PC的更新由灰色虚线画出。指令不发生转移，因此Branch = 0，而PCSrc也为0。PC由PCPlus4获得下一个值。

注意数据也经过了没有加粗表示的路径，但是这些数据对这条指令并不重要。例如，立即数字段也经过了符号扩展，也由存储器读出了数据，但是这些值并不影响系统的下一状态。

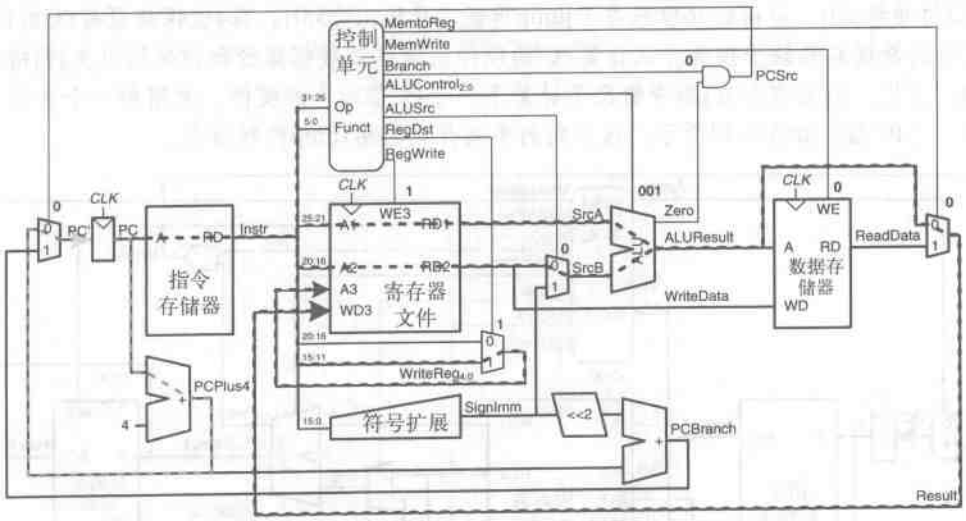


图7-13 执行or指令时的控制信号和数据流

7.3.3 更多指令

我们已经考虑了整个MIPS指令集中的一个很小子集。下面，我们将增加addi指令和j指令以说明处理新指令的原则，并将提供比较充足的指令集以完成一些有趣的程序。可以看到，对于某些指令而言仅仅需要增强主译码器就可以实现，而另外一些指令则需要在数据路径中增加新的部件。

例7.2 addi指令

加立即数指令addi将寄存器中的值和立即数相加，并将结果写入到另一个寄存器中。数据路径已经可以完成这个任务。修改控制器以支持addi指令。

解：我们所需要做的仅仅是在主译码器真值表上为addi指令加上一行控制信号的值，如表7-4所示。结果需要写入寄存器文件，因此RegWrite=1。目的寄存器由指令的rt字段决定，因此RegDst=0。SrcB来源于立即数，因此ALUSrc=1。指令不发生转移也不访问存储器，因此

Branch=MemWrite=0。结果来自于ALU，而非存储器，因此MemtoReg=0。最后，ALU应执行加法操作，因此ALUOp=00。

表7-4 支持addi指令的主译码器真值表

指令	操作码	RegWrite	RegDst	ALUsrc	Branch	MemWrite	MentoReg	ALUOp
R-类型	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

例7.3 j指令

无条件跳转指令j将向PC写入一个新的值。因为PC是字对齐的（总能被4整除），因此PC的最低2位始终为0。前面的26位来自于jump地址字段Instr[25:0]。高4位保持原有PC的值。

已有的数据路径缺少按此方式计算PC′的硬件。请修改数据路径和控制器以支持j指令。

解：首先，需要增加在j指令情况下计算下一个PC值PC′的硬件，并增加一个多路开关来选择下一个PC值，如图7-14所示。这个新的多路开关使用Jump控制信号。

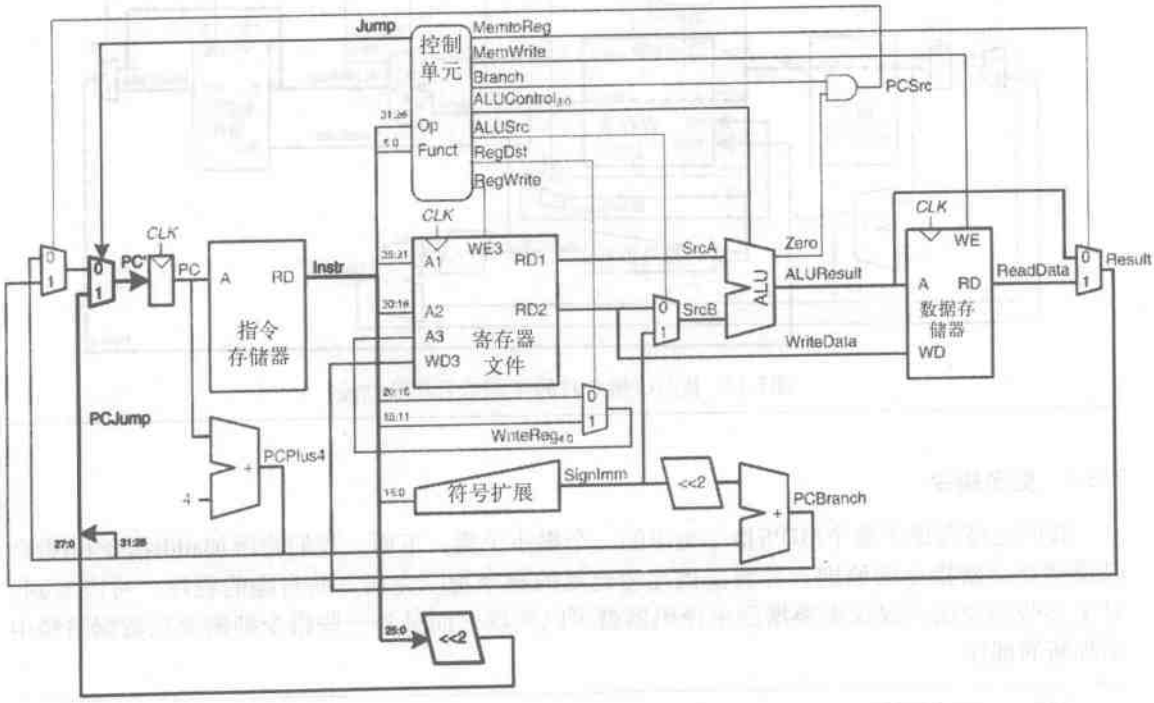


图7-14 支持j指令的单周期MIPS数据路径

现在需要在主译码器真值表中为j指令增加一行，并为Jump控制信号增加一列，如表7-5所示。Jump控制信号仅在j指令时为1，其他指令时均为0。j指令不需要写入寄存器和存储器，因此RegWrite=MemWrite=0。因此，我们不需要关心数据路径中的计算，从而设置RegDst=ALUSrc=Branch=MemtoReg=ALUOp=X。

表7-5 支持j指令的主译码器真值表

指令	操作码	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoRge	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

7.3.4 性能分析

在单周期处理器中，每条指令需要一个时钟周期，因此CPI为1。lw指令的关键路径如图7-15中的加粗虚线所示。它从PC在时钟上升沿加载新的地址开始。指令存储器读出下一条指令。寄存器文件读出SrcA。寄存器文件读出的同时，立即数字段进行符号扩展并在ALUSrc多路开关处进行选择以确定SrcB。ALU将SrcA和SrcB相加以得到有效地址。数据存储器从此地址读取数据。MemtoReg多路开关选择ReadData。最后，结果必须在时钟上升沿前建立于寄存器文件中，使结果能被正确写入。因此，时钟周期为

$$T_c = t_{pcq_PC} + t_{mem} + \max[t_{RFread}, t_{sext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \quad (7\text{-}2)$$

在绝大多数实现中，ALU、存储器和寄存器文件都比其他操作慢很多。因此，时钟周期可以简化为

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup} \quad (7\text{-}3)$$

这些时间的具体值依赖于特定的实现技术。

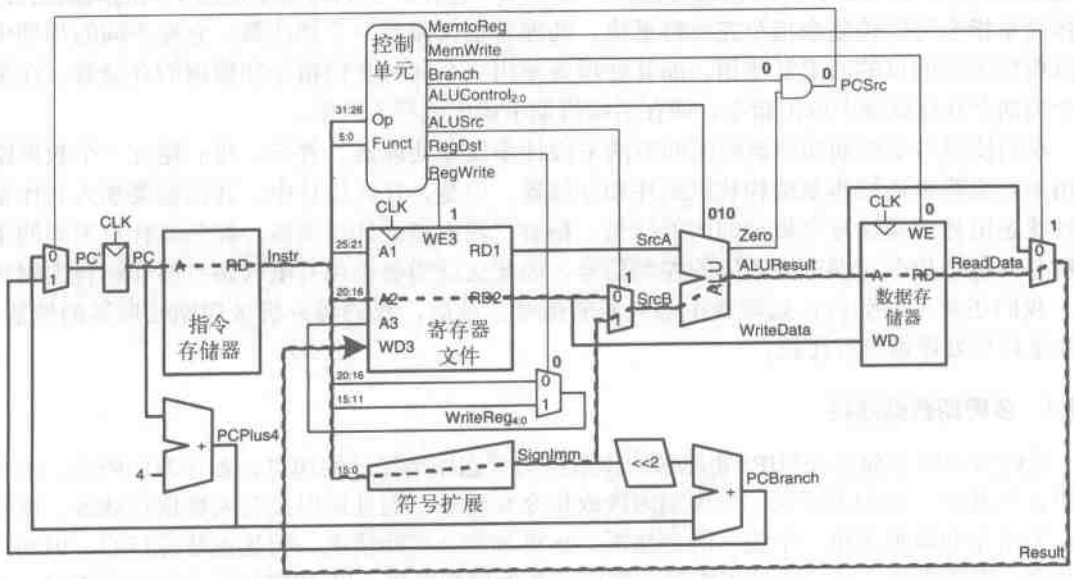


图7-15 lw指令的关键路径

其他指令的关键路径都比较短。例如R类型指令不需访问数据存储器。然而，我们采用同步时序逻辑设计，因此时钟周期必须是常数，而且应能满足最慢指令的要求。

例7.4 单周期处理器性能

Ben Bitdiddle在65nm的CMOS工艺上实现了单周期MIPS处理器。他所选逻辑组件的延迟如表7-6所示。请帮助他比较有1000亿条指令程序的执行时间。

解：根据式（7-3），单周期处理器的时钟周期为T_{C1} = 30 + 2(250) + 150 + 25 + 200 + 20 = 925ps。这里使用下标“1”来区别于后续的处理器设计。由式（7-1），整体执行时间为T_1 = (100 × 10^9指令) (1周期/指令) (925 × 10^{-12}秒/周期) = 92.5秒。

表7-6 电路组件的延迟

组 件	参数	延迟（ps）	组 件	参数	延迟（ps）
寄存器时钟到Q时间	t_{pcq}	30	存储器读	t_{mem}	250
寄存器建立时间	t_{setup}	20	寄存器文件读	t_{RFread}	150
多路开关	t_{mux}	25	寄存器文件建立时间	$t_{RFsetup}$	20
ALU	t_{ALU}	200			

7.4 多周期处理器

单周期处理器有三个主要缺点：首先，它需要足够长的周期以完成最慢的指令，虽然大部分指令的速度非常快。第二，它需要三个加法器（一个用于ALU，两个用于PC的逻辑），而加法器是相对比较占用芯片面积的电路，尤其在速度比较快的时候。第三，它采用了分离的指令存储器和数据存储器，而这在实际系统中是不现实的。大多数计算机仅有一个大容量存储器来存储指令和数据，而且可以支持读和写操作。

多周期处理采用将指令执行过程分解为多个周期以解决这个问题。在每个周期中，处理器完成存储器或寄存器文件的读或写操作，或者ALU操作。不同的指令使用不同的周期组合，这样简单指令可以较复杂指令完成得更快。处理器也仅需要一个加法器，它在不同的周期中可以根据不同的目的而重复使用。而且处理器使用一个同时存储指令和数据的存储器。在第一个周期中从存储器中取出指令，而在后续周期中读出或写入数据。

我们使用与单周期处理器相同的方法来设计多周期处理器。首先，我们建立一个数据路径用组合电路来连接体系结构状态组件和存储器。但是，在此设计中，我们需要加入非体系结构状态组件以保存每个周期的内部结果。接着，我们将设计控制器。控制器针对不同的单条指令的每个指令周期产生不同的控制信号，因此这就需要使用有限状态机而非组合逻辑实现。我们还将考察如何在处理器中加入新的指令。最后，我们将分析多周期处理器的性能，并和单周期处理器进行比较。

7.4.1 多周期数据路径

我们再次以存储器和MIPS处理器的体系结构状态作为设计的起点，如图7-16所示。在单周期处理器中，我们需要在一个周期内读取指令存储器，而且读出或写入数据存储器。现在我们为指令和数据使用一个统一的存储器。这更加符合实际情况，而且也是可行的，因为读指令在一个周期内完成，而读出或写入数据在其他周期完成。PC和寄存器文件保持不变。我们将逐步地添加组件以构造数据路径，以处理每条指令的各个周期操作。新建立的连接使用黑色表示，已加入的硬件组件将采用灰色表示。

图7-16 统一指令/数据存储器的状态组件

PC包含了需要执行指令的地址。第一个周期是从指令存储器中读取指令。图7-17显示了PC简单地连接到指令存储器的地址输入端。指令读出后将存储在一个新的非体系结构状态组件——指令寄存器（Instruction Register）中以供后续周期使用。指令寄存器有一个使能控制信号IRWrite，当此信号有效时指令寄存器将更新为一条新的指令。

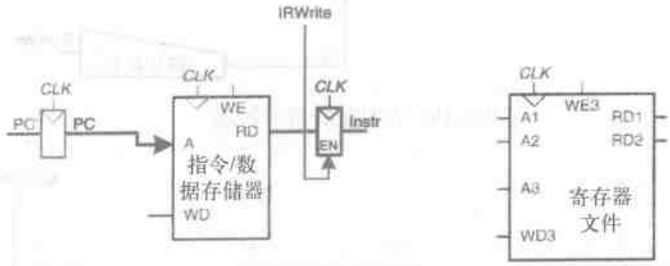


图7-17 从存储器中取指令

正如在单周期处理器中，我们首先为lw指令建立数据路径连接。对于lw指令，下一个周期将读取包含基地址的源寄存器。这个寄存器由指令中的rs字段（$Instr_{25:21}$）指定。指令中的这些位直接连接到寄存器文件的一个地址输入A1上，如图7-18所示。寄存器文件读取寄存器到RD1端口，输出的数据存储到另一个非体系结构寄存器A中。

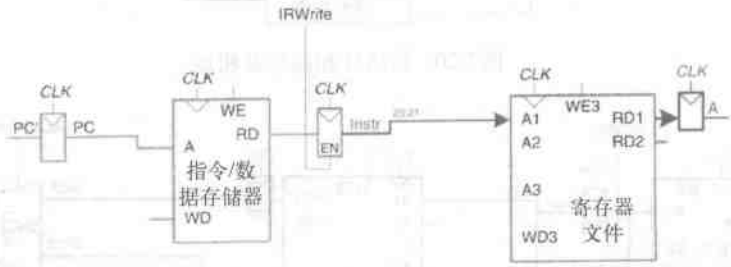


图7-18 从寄存器文件中读取源操作数

lw指令还需要一个偏移量。这个偏移量存储在指令的立即数字段（$Instr_{15:0}$），而且必须经过32位符号扩展。经过扩展的32位值称为SignImm。为保持一致性，我们应该将SignImm存储在非体系结构寄存器中。然而，SignImm是Instr产生的组合逻辑结果，而且在当前指令处理过程中都不会发生变化，因此就不需要一个特定的寄存器来存储这个不变的信号。

读出操作的地址为基地址和偏移量之和。我们使用ALU部件计算这个和，如图7-20所示。ALUControl应设置为010以完成加法操作。ALUResult存储在一个非体系结构寄存器ALUOut中。

下一个周期是根据地址从存储器中读取数据。我们在寄存器前面增加一个多路开关以从PC和ALUOut中选择存储器地址Adr，如图7-21所示。多路开关的选择信号为IorD，以区别指

令或数据地址。从存储器中读出的数据将存储器在另一个非体系结构寄存器Data中。注意，地址多路开关允许我们在lw指令中重用存储器。在第一个周期中，地址来自PC以读取指令；在后续周期中，地址来自ALUOut以读取数据。因此，IorD信号在不同的周期中应有不同的值。在7.4.2节中，我们设计FSM控制器以产生这些控制信号序列。

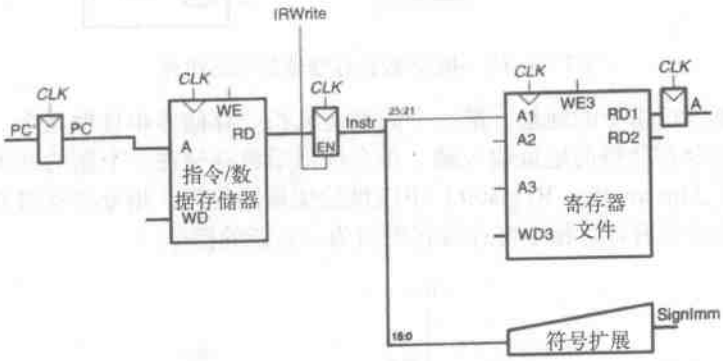


图7-19 立即数的符号扩展

图7-20 基地址和偏移量相加

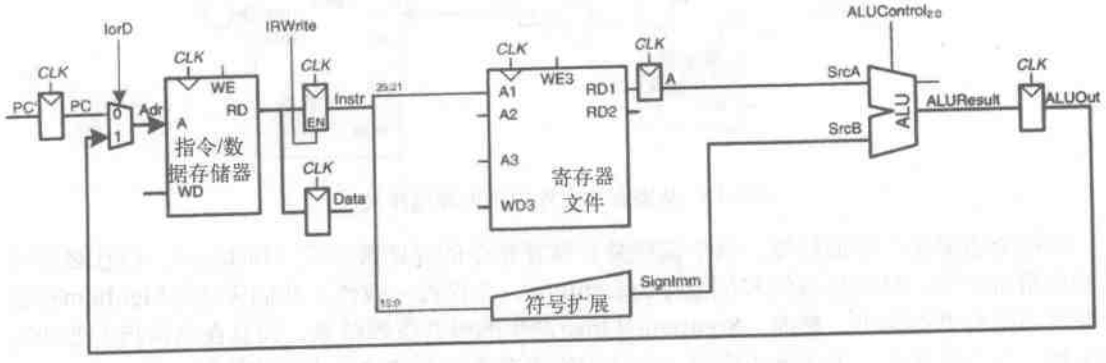


图7-21 从存储器中读取数据

最后，数据将写回到寄存器文件，如图7-22所示。目的寄存器由指令中的rt字段($Instr_{20:16}$)指定。

在完成上述操作后，处理器必须在原先的PC上加4以更新程序计数器。在单周期处理器中，

需要一个单独的加法器。在多周期处理器中，我们可以在ALU不忙的周期中使用它。为此，必须增加源多路开关以选择PC和常数4作为ALU的输入，如图7-23所示。两输入多路开关由ALUSrcA信号控制，以选择PC或寄存器A作为SrcA。四输入多路开关由ALUSrcB控制，以选择常数4或SignImm作为SrcB。剩下的两路输入将在后面用于扩展数据通路以处理其他指令（多路开关的输入数目可以为任意）。为了更新PC，ALU将SrcA（PC）和SrcB（4）相加，并将结果写入到程序计数器寄存器中。PCWrite控制信号将在特定周期中使能，使得PC寄存器可以写入结果。

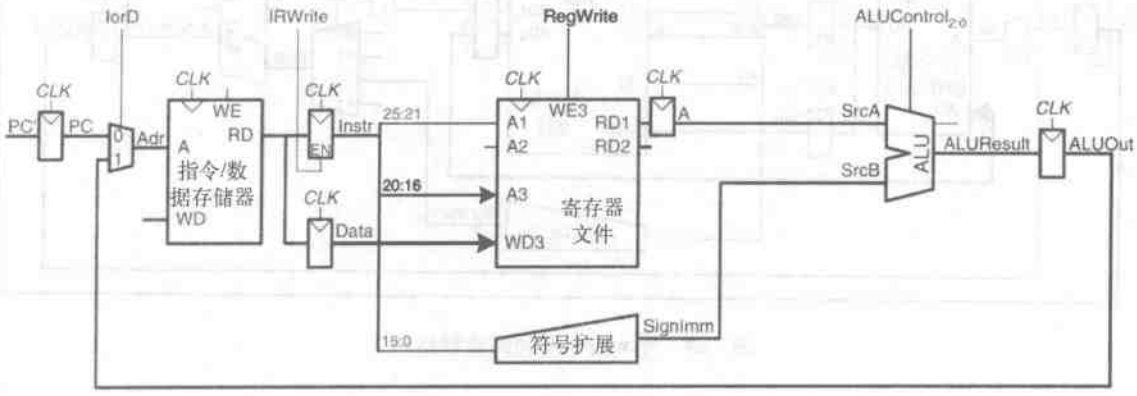


图7-22 写回数据到寄存器文件

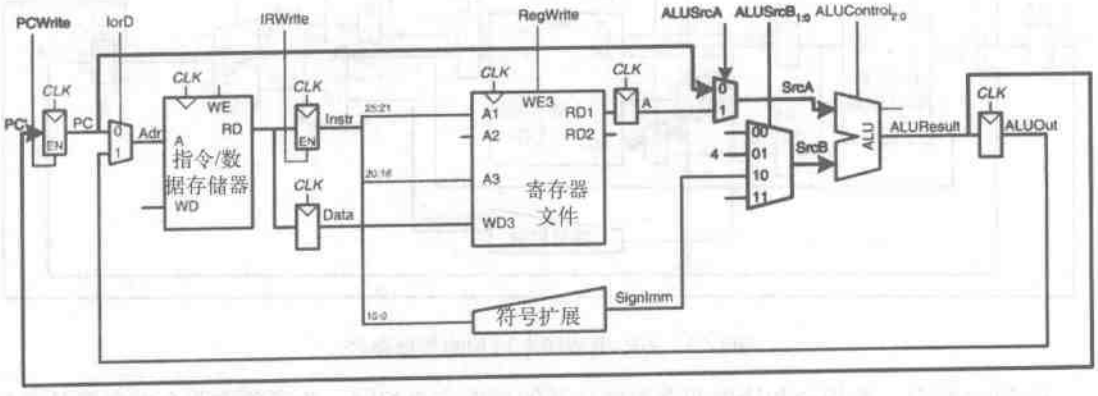


图7-23 PC递增4

此时，已经完成了lw指令的数据路径。下一步，我们将扩展数据路径以处理sw指令。与lw指令类似，sw指令从寄存器文件端口1读出基地址，并对立即数进行符号扩展。ALU将立即数和基地址相加以获得存储器地址。已有的数据路径已经可以这些功能。

sw指令新的特征是必须从寄存器文件中读出第二个寄存器的值并写入到存储器中，如图7-24所示。这个寄存器由指令中的rt字段（$Instr_{20:16}$）指定，因此这个字段将连接到寄存器文件的第二个端口上。当寄存器被读出后，其数值将存储在非体系结构寄存器B中。在下一个周期中，B将通过数据存储器的写入端口WD写入到存储器中。数据存储器需要新增加一个控制信号MemWrite以表示进行写入操作。

对于R-类型指令，取指令的操作完全相同，而且将从寄存器文件中读取两个源寄存器。SrcB多路开关的另外一个输入将用于选择寄存器B作为ALU的输入，如图7-25所示。ALU将

执行相应的操作并将结果存储在ALUOut中。在下一个周期中，ALUOut将写入到指令rd字段（$Instr_{15:11}$）指定的寄存器中。这需要增加两个多路开关：MemtoReg用于从ALUOut（针对R类型指令）或Data（针对lw指令）选择一个作为WD3的输入；RegDst用于选择指令中的rt字段或者rd字段作为寄存器文件写入的地址。

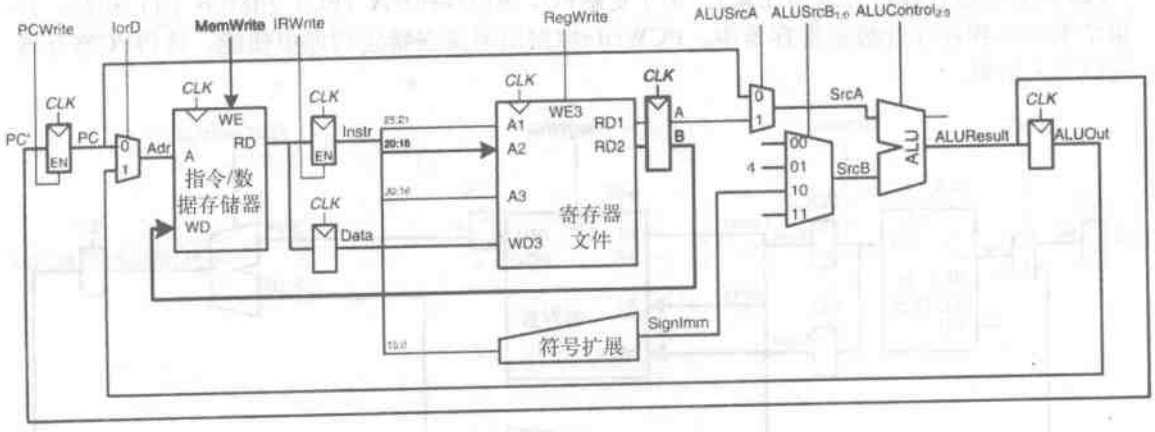


图7-24 为sw指令增加的数据路径

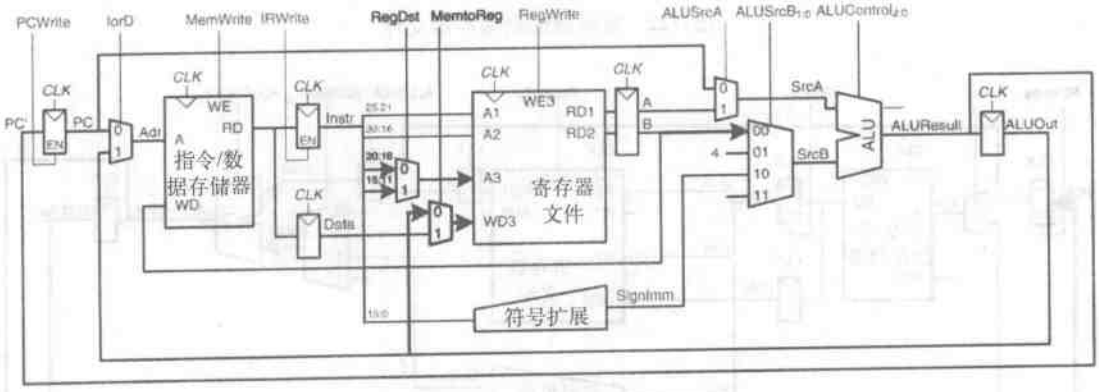


图7-25 为R-类型指令增加的数据路径

对于beq指令，取指令和读取两个源寄存器的操作完全相同。为了确定两个寄存器是否相等，ALU将两个寄存器读出的值相减，并检查Zero标志。同时如果发生转移的话，数据路径必须计算PC的新值：PC'=PC+4+SignImm×4。在单周期处理器中需要另外一个加法器来计算转移地址。在多周期处理器中，可以将ALU复用以减少硬件。在后续周期中，就像其他指令一样通过ALU计算PC+4并将结果写入到PC寄存器中。在此周期之后，ALU将使用更新后的PC完成PC+SignImm×4。SignImm左移2位以实现乘以4，如图7-26所示。SrcB多路开关选择这个值，并将其加入到PC中。结果表示了转移的目的地址并存储到ALUOut中。由PCSrc信号控制的一个多路开关将选择哪一个结果将送到PC'中。程序计数器当PCWrite有效或发生转移时将写入PC寄存器。新的控制信号Branch表示beq指令正在执行。当Zero标志有效时将发生转移。因此数据路径计算新的PC写入使能PCEn，当PCWrite有效或者当Branch和Zero同时有效。

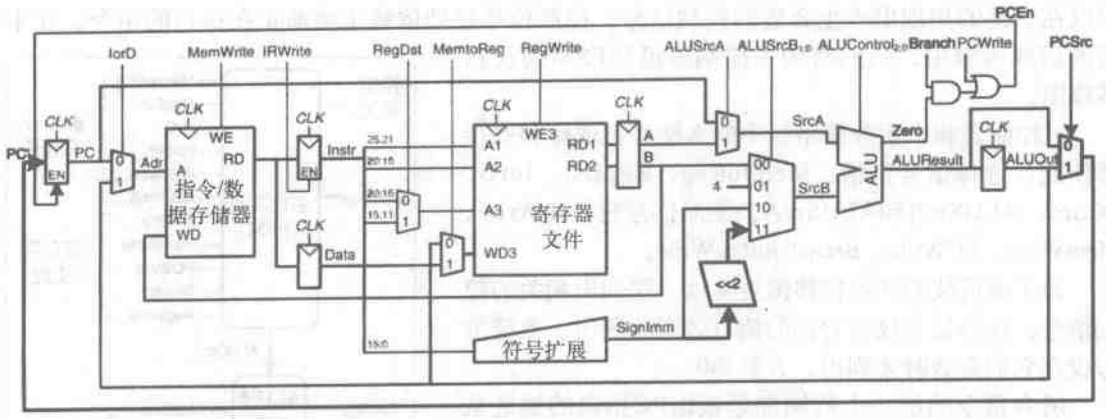


图7-26 为beq指令增加的数据路径

此时，已经实现了多周期MIPS处理器的数据路径。设计过程非常类似于单周期处理器，即根据每条指令链接状态组件。最大的区别是指令执行需要经过多个周期。需要插入非体系结构寄存器以保持每个周期的结果。在此方式下，ALU可以多次被复用，以节省加法器的额外开销。类似地，指令和数据可以存储在一个共享的存储器中。在下一节中，我们将开发FSM控制器以针对每条指令的每个周期产生控制信号序列。

7.4.2 多周期控制

正如单周期处理器，控制单元根据指令中的opcode字段（$Instr_{31:26}$）和funct字段（$Instr_{5:0}$）计算控制信号。图7-27给出了完整的多周期MIPS处理器的控制单元和数据路径链接。数据路径采用黑色显示。

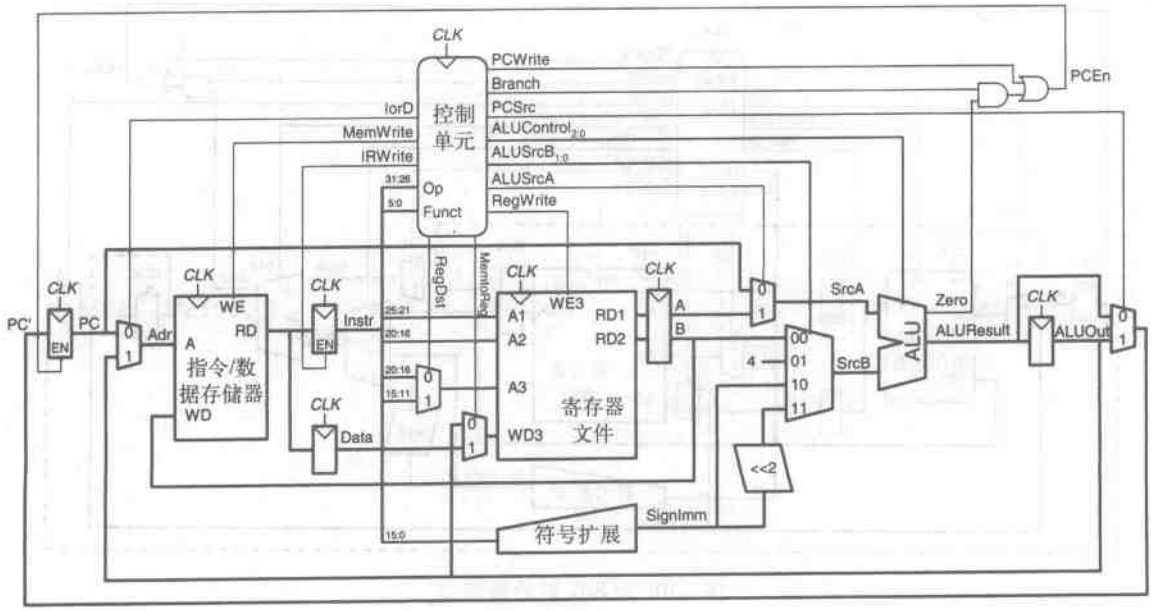


图7-27 完整的多周期MIPS处理器

在单周期处理器中，控制单元分为一个主控制器和一个ALU译码器，如图7-28所示。ALU译码器不发生变化，并满足表7-2所示的真值表。然而，现在需要将主控制器采用FSM实

现以在合适的周期中产生合适的控制信号。控制信号序列依赖于当前正在执行的指令。在本节的后续内容中，我们将为主控制器设计FSM的状态转移图。

主控制器将产生数据路径中的多路开关选择和寄存器使能。选择信号包括：MemtoReg，RegDst，IorD，PCSrc，ALUSrcB和ALUSrcA。使能信号包括IRWrite，MemWrite，PCWrite，Branch和RegWrite。

为了使后续的状态转移图更易读，仅列出相关的控制信号。选择信号仅在它们的值有效时才列出。使能信号仅在它们有效时才列出，否则为0。

所有指令的第一个周期都是根据PC指向的地址从存储器中读取指令。FSM在复位后将进入这个状态。为了读取存储器，IorD＝0，从PC获得地址。IRWrite将有效以将指令写入到指令寄存器IR中。同时PC应该递增4以指向下一条指令。由于ALU在此时没有被作为它用，因此处理器可以在取出指令的同时计算PC＋4。ALUSrcA=0，使得SrcA来源于PC；ALUSrcB＝01，使得SrcB来源于常数4；ALUOP=00，使得ALU译码器产生控制信号ALUControl=010完成加法。为了在此周期更新PC，PCSrc=0，并设置PCWrite为有效。控制信号如图7-29所示。在此周期的数据流如图7-30所示，其中取指令由虚线所示，PC递增由灰色虚线所示。

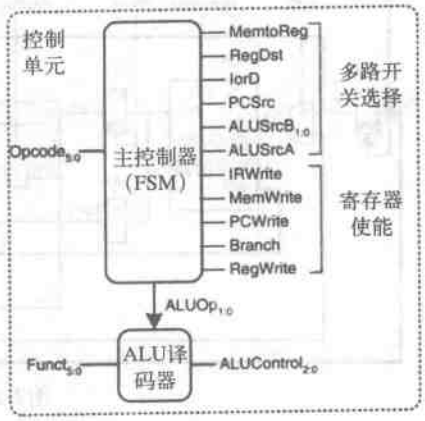


图7-28 控制单元内部结构

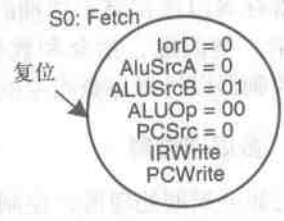


图7-29 取指令

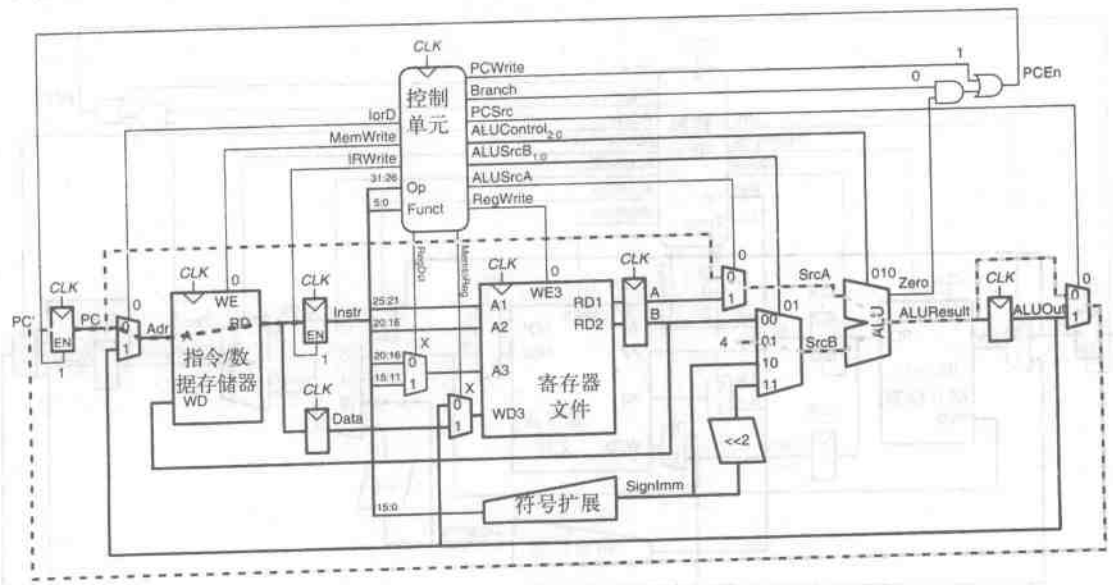


图7-30 取指阶段的数据流

下一个周期将读取寄存器文件，并对指令译码。寄存器文件始终由指令中的rs和rt字段读取。同时，立即数进行符号扩展。译码操作为检查指令中opcode字段以决定后续操作。译码阶段不需要控制信号，但是需要等待一个周期以完成读取和译码，如图7-31所示。数据流如

图7-32所示。

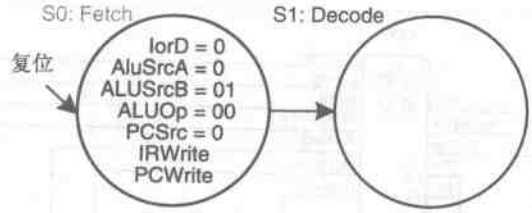


图7-31 译码

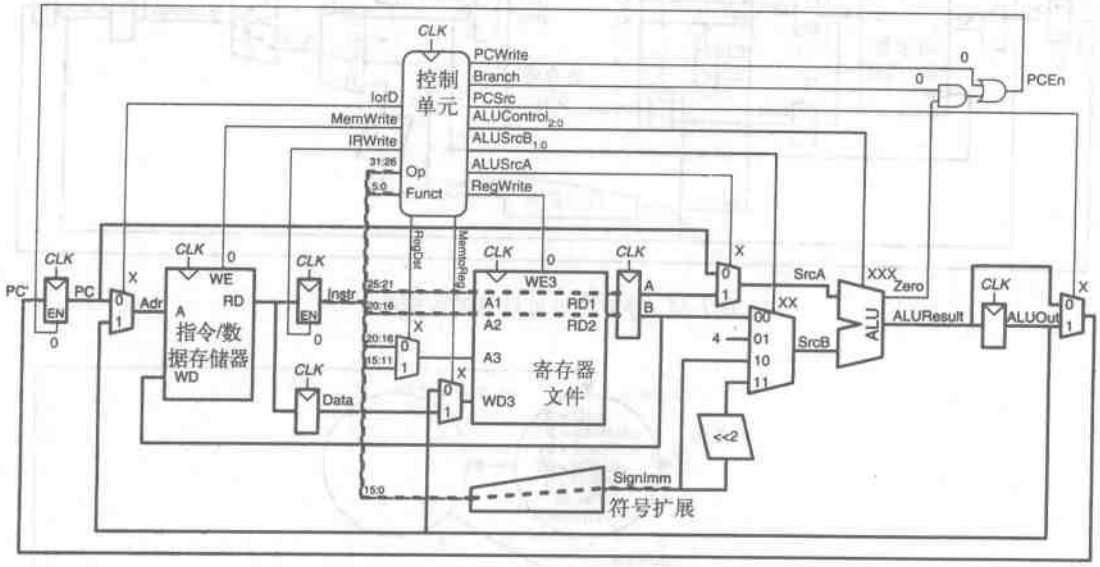


图7-32 译码阶段的数据流

现在FSM将根据opcode产生多种可能后续状态之一。如果指令时存储器读取或存储指令（lw或sw），多周期处理器将基地址和符号扩展的立即数相加以计算地址。这需要ALUSrcA＝1以选择寄存器A，ALUSrcB＝10以选择SignImm，ALUOp＝00使得ALU执行加法。有效地址将存储在ALUOut寄存器中为下一个周期做准备。此时FSM状态如图7-33所示，数据流如图7-34所示。

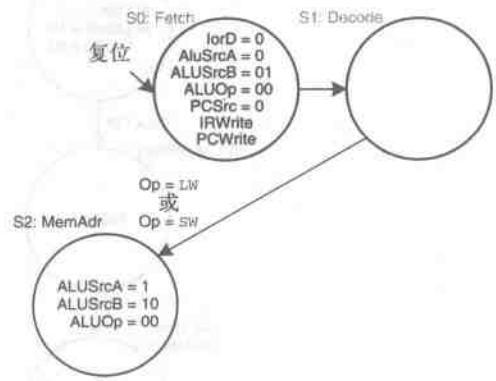


图7-33 存储器地址计算

如果指令为lw，多周期处理器必须紧接着从存储器中读取数据并写入到寄存器文件中。这两步如图7-35所示。为了读取存储器，IorD＝1以将ALUOut中刚刚计算得到的值作为存储器地址。存储器将读取这个地址，并将结果在S3周期存放到Data寄存器中。在下一个周期S4中，Data将写入到寄存器文件，MemtoReg＝1以选择Data，RegDst＝0以从指令中的rt字段获得目的寄存器地址。RegWrite将有效以执行写入操作，从而完成lw指令。最后，FSM将返回到初始状态S0，以取下一条指令。

对于上述步骤，请读者自行完成数据流图。

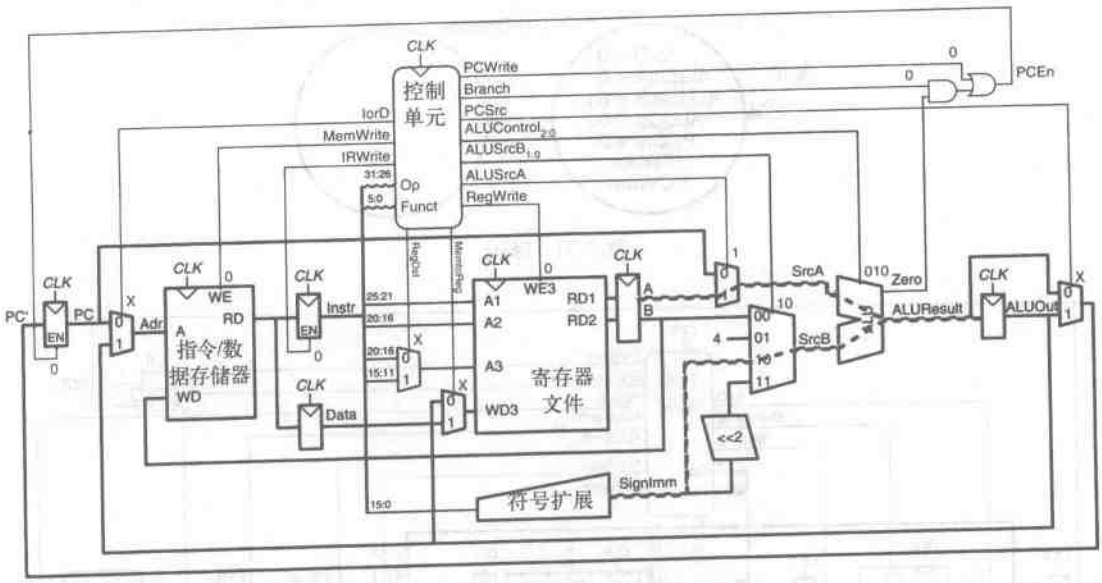


图7-34 存储器地址计算的数据流

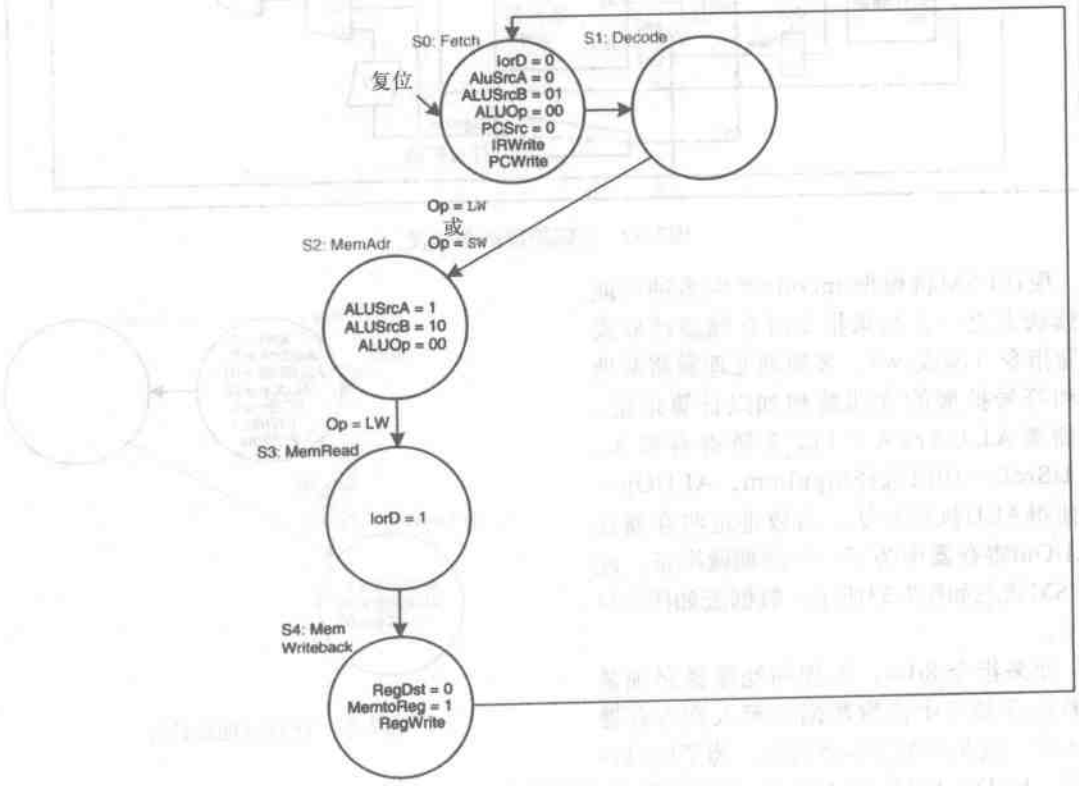


图7-35 存储器读

在状态S2中，如果指令为sw，从寄存器文件端口2中读取的数据简单地写入到存储器中。

IorD=1以选择S2周期中计算得到并保存在ALUOut中的地址。MemWrite将设置为有效以写入存储器。同样，FSM将返回到S0状态以取下一条指令。增加的步骤如图7-36所示。

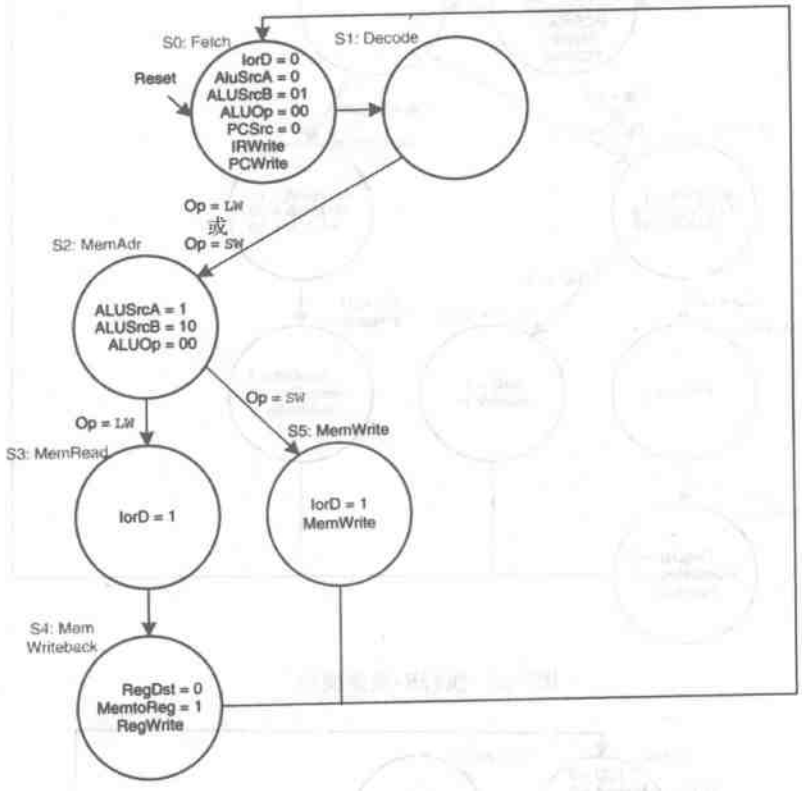


图7-36 存储器写

如果opcode为R类型指令，多周期处理器使用ALU计算结果并将结果写入到寄存器文件中。图7-37给出了这两步。在S6状态中，通过选择寄存器A和B（ALUSrcA＝1，ALUSrcB＝00）并根据指令中的funct字段执行ALU操作以完成指令。ALUResult存储器在ALUOut中。在状态S7，ALUOut将写入寄存器文件。RegDst＝1，这是因为目的寄存器由指令中的rd字段指定。MemtoReg=0，这是因为要写入数据WD3来自于ALUOut。RegWrite设置为有效以写入寄存器文件。

对于beq指令，处理器必须计算目的地址，并比较两个源寄存器以确定是否发生转移。这需要两次使用ALU，这看上去好像需要两个新的状态。然而，我们注意到在S1状态时完成寄存器读出操作，而没有使用ALU。处理器可以在此时利用ALU将递增后的PC和SignImm×4相加以计算目的地址，如图7-38所示。ALUSrcA=0，以选择递增后端PC；ALUSrcB=11以选择SignImm×4，而且ALUOp=00以完成加法。目的地址将存储器在ALUOut中。如果当前执行的不是beq指令，计算的地址将不再后续过程中使用，但是这个计算也没有坏处。在S8状态，处理器通过减法并判断结果是否为0来比较两个寄存器的值。如果结果为0，则处理器将转移到刚刚计算得到的地址。ALUSrcA=1，以选择寄存器A；ALUSrcB=00，以选择寄存器B；ALUOp=01，以完成减法；PCSrc=1，以从ALUOut中得到目的地址；当ALU结果为0时Branch=1，以更新PC[⊖]。

⊖ 这里我们可以看到为什么需要PCSrc多路开关来选择PC'，其中在S0状态选择ALUResult，在S8状态时选择ALUOut。

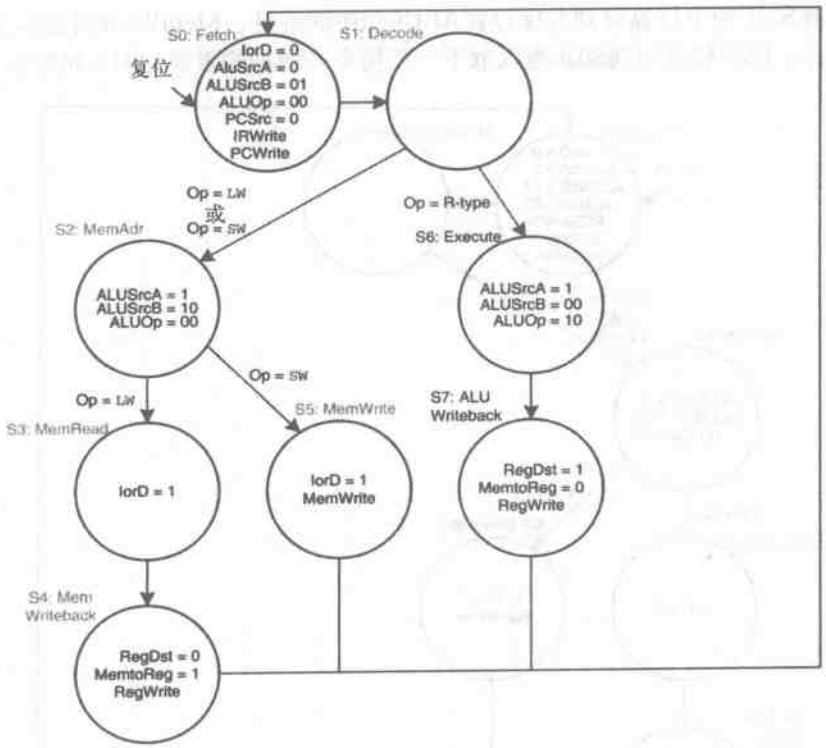


图7-37 执行R-类型操作

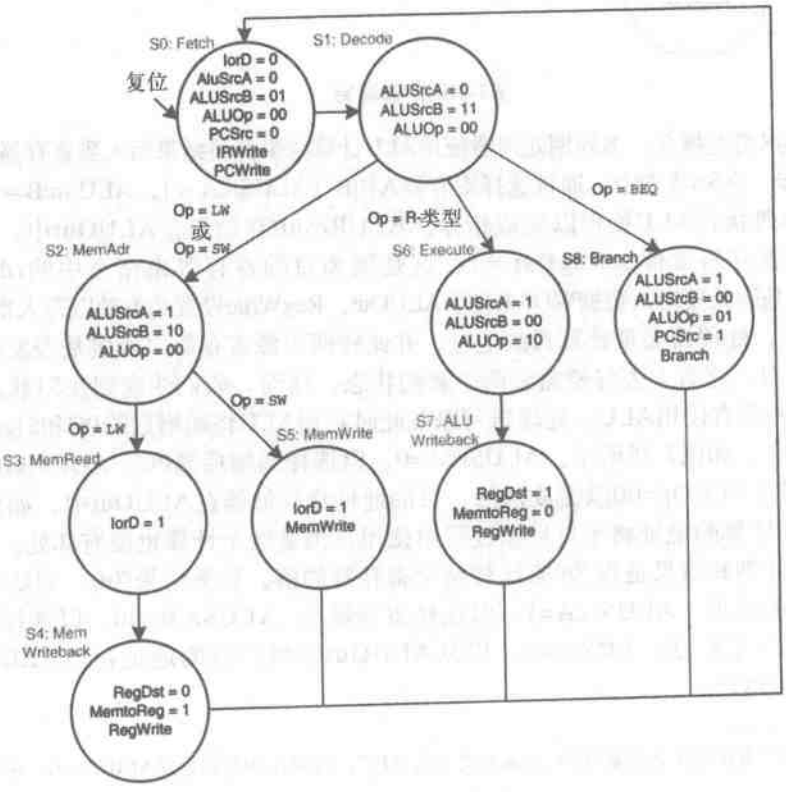


图7-38 分支指令

将上述步骤合并在一起，图7-39显示了多周期处理器的完整主控制器状态转移图。利用第3章所述的技术，将上述转移图转化为硬件是比较直观的，但是工作量比较大。比较好的是，可以根据第4章所述的技术对FSM进行综合和用HDL语言来编码。

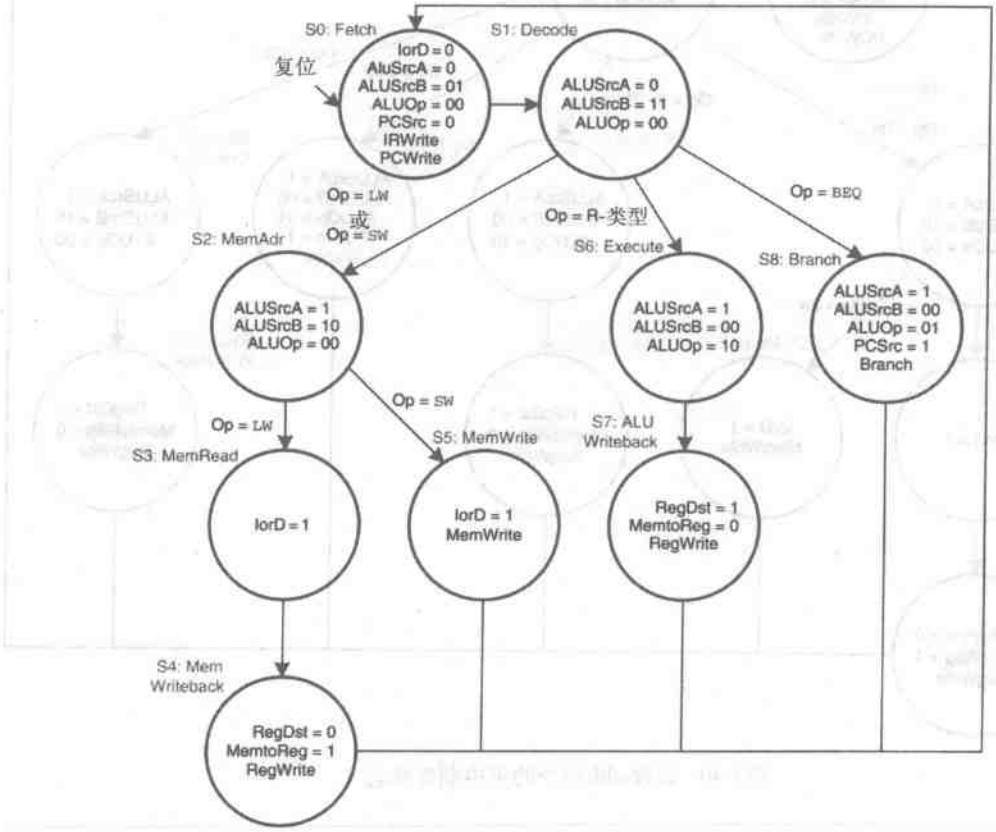


图7-39 完整的多周期控制FSM

7.4.3 更多指令

如7.3.3节中对单周期处理器所进行的工作，我们现在将扩展多周期处理器以支持addi和j指令。下面的两个例子显示了支持新指令的设计过程。

例7.5 addi指令

修改多周期处理器以支持addi指令。

解：数据路径已经提供了将寄存器和立即数相加的能力，因此我们所需要做的就是为addi指令在主控制器FSM中加入新的状态。这个状态类似于R类型指令。在状态S9中，寄存器A将加上SignImm（ALUSrcA=1，ALUSrcB=10，ALUOp=00），其结果ALUResult将存储在ALUOut。在S10状态，ALUOut将写入到指令rt字段指定的寄存器中（RegDst=0，MemtoReg=0，RegWrite为有效）。精明的读者可能会发现S2和S9是相同的，可以被合并到一个状态。

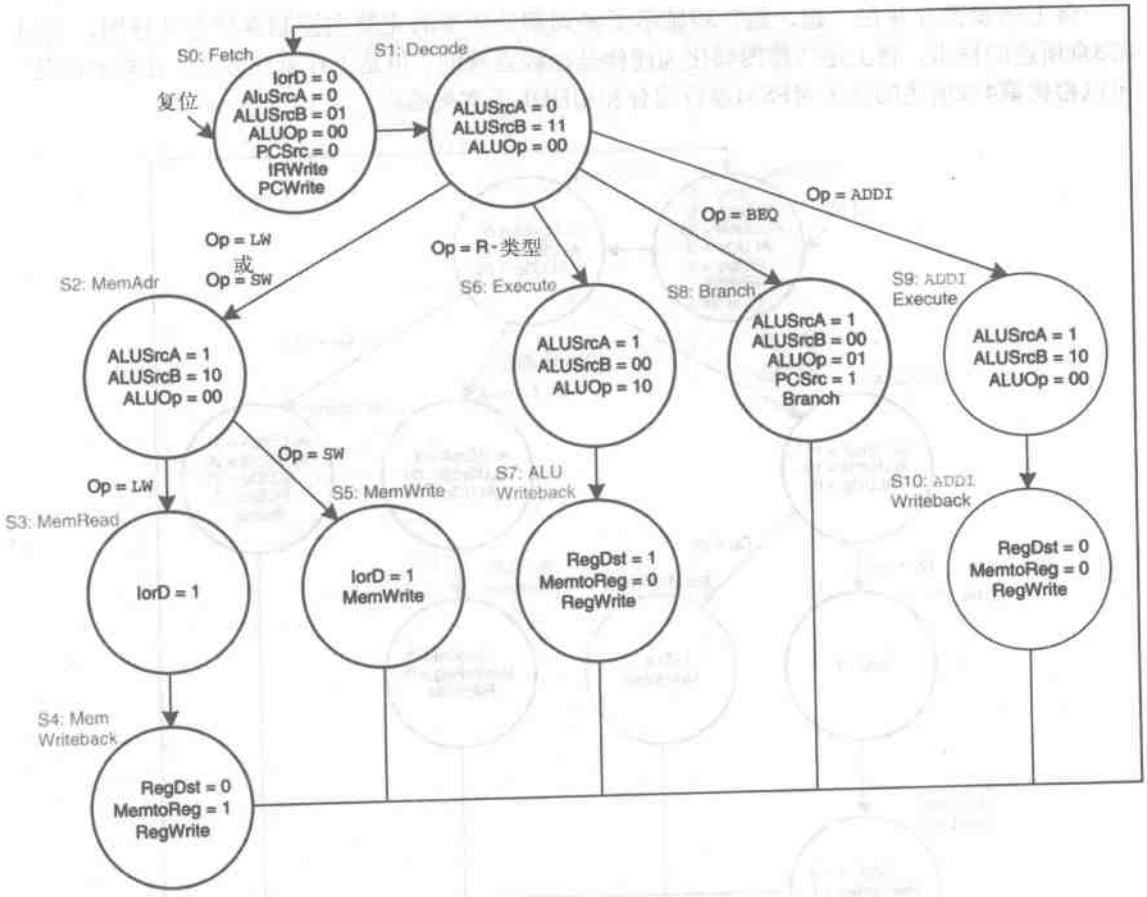


图7-40 处理addi指令的主控制器状态

例7.6 j指令

修改多周期处理器以支持j指令。

解：首先，我们必须修改数据路径为j指令并计算下一个PC值。而且我们在主控制器中要加入一个新的状态来处理这条指令。

图7-41显示了增强的数据路径。跳转目的地址由指令中的26位addr字段左移2位得到，接着需要保持递增后PC的最高四位。应扩展PCSrc多路开关以将此地址作为第三个输入。

图7-42给出了增强的主控制器。新的状态S11简单地选择PCJump值作为PC' (PCSrc = 10)，并写入到PC。转移PCSrc选择信号在S0和S8两个状态都已经扩展到2位。

7.4.4 性能分析

程序的执行时间依赖于所用的周期数和指令周期。单周期处理器在一个周期内执行完所有的指令，多周期处理器针对不同的指令周期数执行不同指令。然而，多周期处理器在一个周期中的工作比较少，因此指令周期将比较短。

多周期处理器对beq和j指令需要3个周期，对sw, addi和R类型指令需要4个周期，对lw指令需要5个周期。CPI取决于每类指令的相对频度。

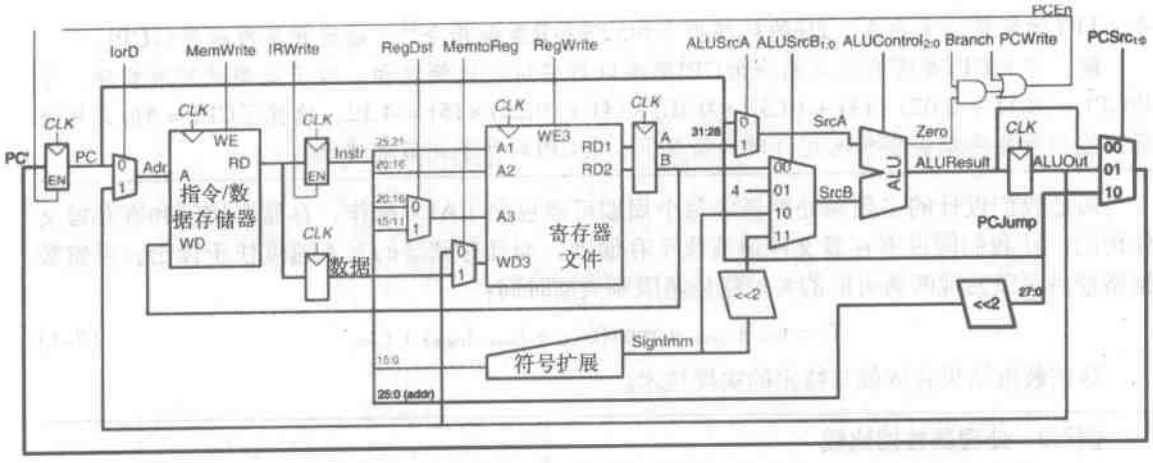


图7-41 支持j指令的多周期MIPS数据路径

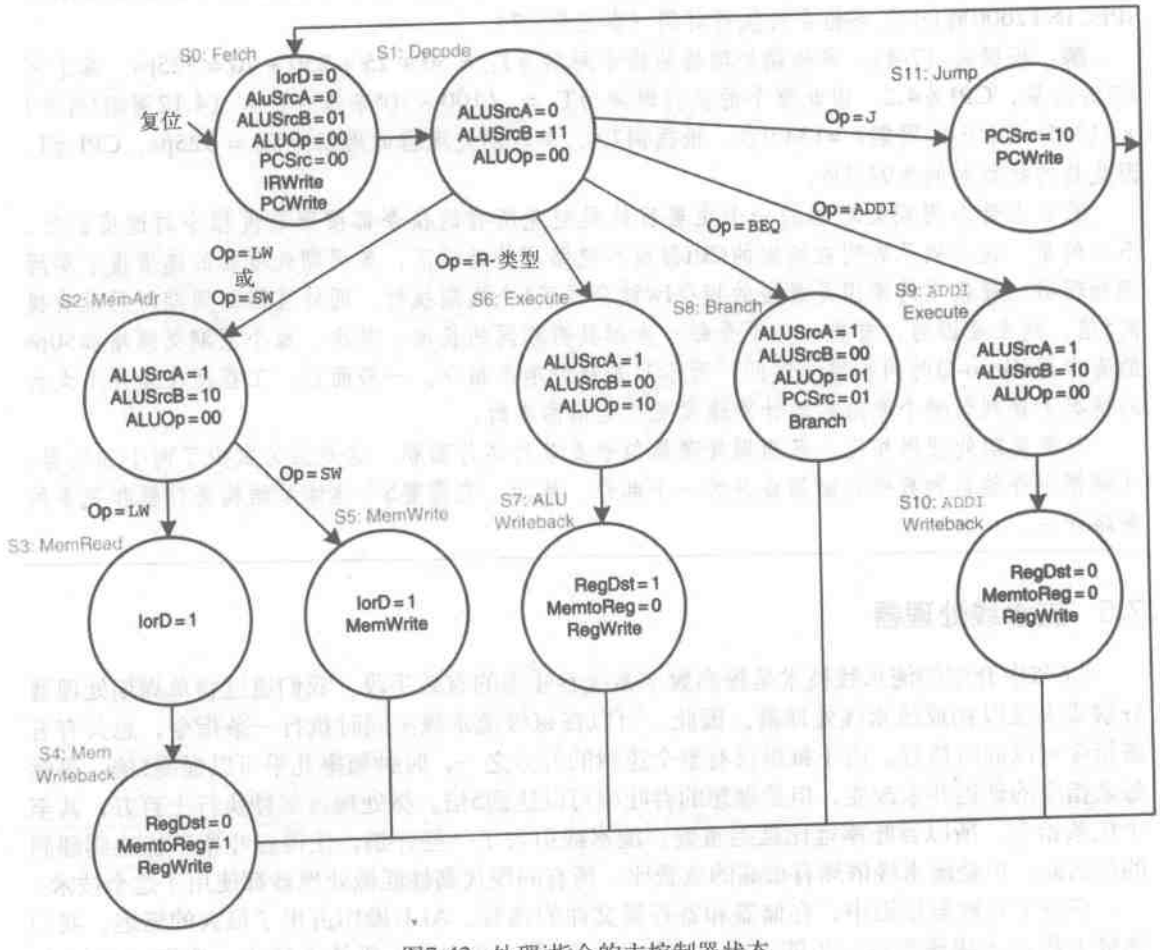


图7-42 处理j指令的主控制器状态

例7.7 多周期处理器的CPI

SPECINT2000测试基准测试程序包含了约25%的存储器读取指令，10%的存储器写入指

令，11%的条件分支指令，2%的转移指令和52%的R类型指令[⊖]。确定此基准程序的CPI。

解：平均CPI是所有各类指令的CPI值乘以程序所占比例之和。对于此测试基准程序，平均CPI = (0.11 + 0.02) × (3) + (0.52 + 0.10) × (4) + (0.25) × (5) = 4.12。这优于CPI = 5的最坏情况。在所有指令都需要相同运行时间情况下，其CPI均处于此最坏情况。

回忆我们设计的多周期处理器中每个周期可能包含了ALU操作，存储器访问和寄存器文件访问。让我们假设寄存器文件速度快于存储器，而且存储器的写入速度快于读出。考察数据路径后可以发现两条可能的关键路径将限制周期时间：

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \tag{7-4}$$

这些数值结果将依赖与特定的实现技术。

例7.8 处理器性能比较

Ben Bitdiddle正在考虑是否应该使用多周期处理器来代替单周期处理器。对这两个设计，他计划使用65nm CMOS工艺实现，其延迟由表7-6给出。请帮助他比较处理器执行来自SPECINT2000的1千亿条指令的执行时间（参见例7.7）。

解：根据式（7-4），多周期处理器的指令周期为T_{c2} = 30 + 25 + 250 + 20 = 325ps。基于例7.7的结果，CPI为4.2，因此整个的执行时间为T_2 = （100 × 10^9条指令）×（4.12周期/指令）×（325 × 10^{-12}秒/周期）=133.9秒。根据例7.4，单周期处理器的周期为T_{c1} = 925ps，CPI为1，因此总的执行时间为92.5秒。

原先设计多周期处理器的一个主要目的是避免所有的指令都按照最慢指令的速度执行。不幸的是，这个例子表明在给定的CPI假设和电路器件延迟下，多周期处理器的速度慢于单周期处理器。最根本的原因是最慢的指令lw被分成了5个周期执行，而处理器的周期时间没有提高5倍。这主要因为：首先，并不是每一步都具有相同的长度，其次，每个周期必须增加50ps的寄存器clk-to-Q时间和建立时间，而不只是针对单条指令。一般而言，工程师在差异不太大的情况下要判断哪个微结构的计算速度更快是很困难的。

与单周期处理器相比，多周期处理器似乎更节约芯片面积，这是因为减少了两个加法器，并将指令存储器和数据存储器合并为一个部件。然而，它需要5个非体系结构寄存器和更多的多路开关。

7.5 流水线处理器

3.6节中介绍的流水线技术是提高数字系统吞吐率的有效手段。我们通过将单周期处理器分解成五级以构成流水线处理器。因此，可以在每级流水线中同时执行一条指令，总共有五条指令可以同时执行。由于每级仅有整个逻辑的五分之一，时钟频率几乎可以提高5倍。虽然每条指令的延迟并未改变，但是理想的吞吐率可以达到5倍。微处理器每秒执行上百万，甚至十亿条指令，所以吞吐率远比延迟重要。流水线引入了一些开销，使得吞吐率不能达到理想的那么高，但是流水线依然有很高的效费比，所有的现代高性能微处理器都使用了这个技术。

在整个处理器延迟中，存储器和寄存器文件的读写、ALU操作占用了很大的延迟。我们选择五级流水线使得每一级能完成一个操作。这里，我们称五级流水线为：取指（Fetch），译码（Decode），执行（Execute），存储器访问（Memory）和写回（Writeback）。它们类似于

⊖ 数据来源于Patterson和Hennessy所著的《Computer Organization and Design》，第3版，Morgan Kaufmann, 2005.

多周期处理器中执行lw指令的五个步骤。在取指阶段，处理器从指令存储器中读取指令。在译码阶段，处理器从寄存器文件中读取源操作数并对指令译码以产生控制信号。在执行阶段，处理器使用ALU执行计算。在存储器访问阶段，处理器从数据存储器中读取或写入数据。最后在写回阶段，如果需要的话，处理器将结果写入到寄存器文件。

图7-43比较了单周期处理器和流水线处理器的时序图。其中时间为水平轴，指令为垂直轴。时序图中采用了表7-6的延迟假设，但是忽略了多路开关和寄存器的延迟。在单周期处理器中，如图7-43(a)，第一条指令从时间0开始从存储器中读出；下一步操作数从寄存器文件中读取；接着由ALU执行必要计算。最后，访问数据存储器，结果在950ps时写回到寄存器文件。第二条指令在第一条指令结束后开始执行。因此，在此时序图中，单周期处理器的指令延迟为250 + 150 + 200 + 250 + 100 = 950ps，而且吞吐率为每950ps执行1条指令（每秒执行1.05×10^9条指令）。

在图7-43(a) 的流水线微处理器中，流水线周期由最慢的一级决定。这里为取指或存储器访问级的存储访问操作，即250ps。在时间0，第一条指令从存储器中取出。在250ps时，第一条指令进入译码阶段，并开始取第二条指令。在500ps时，第一条指令执行，第二条指令进入译码阶段，并开始取第三条指令。如此类推，直至所有的指令完成。指令的延迟为$5 \times 250 = 1250$ps。吞吐率为每250ps执行一条指令（每秒执行4×10^9条指令）。由于流水线级的划分不能完美地平均分配所有的逻辑，因此流水线处理器的延迟将稍长于单周期处理器。类似地，对于五级流水线也不能较单周期处理器提高5倍的吞吐率。但是，其吞吐率的优势是非常明显的。

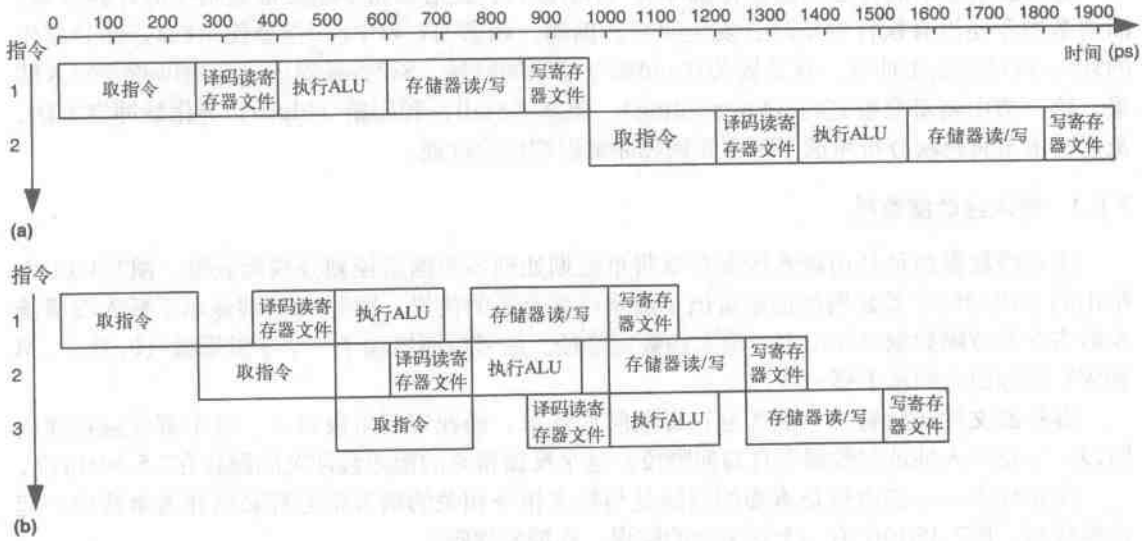


图7-43 时序图：(a) 单周期处理器；(b) 流水线处理器

图7-44中给出了流水线操作的一个抽象表示，其中每级都采用图形表示。每级流水线由主要的组件表示：指令存储器（IM）、寄存器文件（RF）读、ALU执行、数据存储器（DM）和寄存器文件写回，并以此表示流水线中的指令流。沿着每一行读，可以确定特定一条指令在每一级中的周期数。例如，sub指令在第3个周期中取指令，在第5个周期中执行。沿着每一列读，可以确定在特定时刻下不同流水线级的操作。例如，在第6个周期中，or指令从指令存储器中取出，同时$s1正在从寄存器文件中读取，ALU正在计算$t5和$t6的与操作，数据存储器空闲，而正在向寄存器文件的$s3写入和。每级流水线用阴影表示当前正在使用。例如数据存储器在第4个周期中由lw指令使用，在第8个周期由sw指令使用。指令存储器和ALU在每个

周期中都在使用。除了sw指令外，每条指令都要写入寄存器文件。我们假设流水线处理器中，寄存器文件在一个周期的前半部分写入，在后半部分读取。此时，数据可以在一个周期内完成写入和读取。

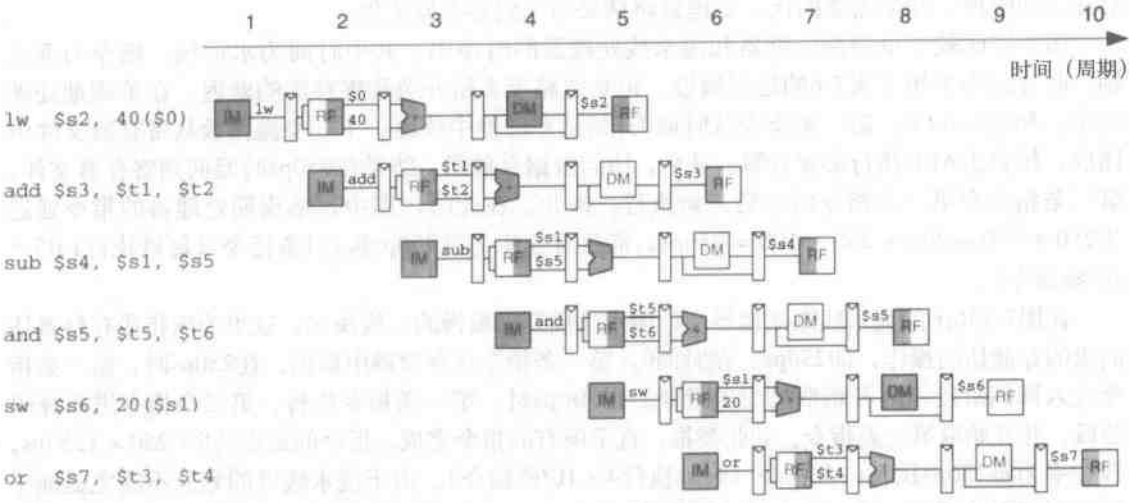


图7-44 流水线操作的抽象表示

流水线系统中的核心问题是化解冲突（Hazard）。在后条指令需要前条指令的计算结果，而前条指令还没有执行完毕时会发生冲突。例如，如果图7-44中add指令使用$s2，而非原先的$t2，将可能发生冲突。这是因为在add指令读取的时候，$s2寄存器还没有由lw指令写入结果。这一节中将研究重定向（Forwarding）、阻塞（Stall）和刷新（Flush）等化解冲突方法。最后，本节将再次分析考虑了时序开销和冲突影响后的性能。

7.5.1 流水线数据路径

流水线数据路径是由流水线寄存器将单周期处理器数据路径划分割为五级。图7-45(a)中给出的单周期处理器数据路径中留出了流水线寄存器的位置。图7-45(b)则显示了插入四级流水线寄存器分隔数据路径以形成流水线数据路径。信号后面增加了一个字母后缀（F, D, E, M和W）以标识它们属于哪一级。

寄存器文件比较特殊，因为它在译码阶段读取，而在写回阶段写入。这个部件画在译码阶段，但是写入地址和数据来自写回阶段。这个反馈带来的流水线冲突问题将在7.5.3中讨论。

流水线中一个细微但是重要的问题是与特定指令相关的所有信号都必须在流水线中一起向前传播。图7-45(b)中有一个这方面的错误，你能发现吗？

这个错误在于在写回阶段的寄存器写入逻辑。待写入的数据为ResultW，这是一个写回级的信号。但是写回的地址却来源于执行级的WriteRegE信号。在图7-44中流水线图中，在第5个周期时lw指令的结果就将被错误地写入到$s4寄存器中，而非正确的$s2寄存器。

图7-46给出了正确的数据路径。WriteReg信号现在经过存储器和写回两级向前传递，因此它将和指令的其他部分保持同步。WriteRegW和ResultW在写回阶段一起输出到寄存器文件中。

细心的读者可能发现PC'的逻辑也有问题，因为它由取值阶段或存储器阶段的信号更新（PCPlus4F和PCBranchM）。这个控制冲突将在7.5.3节中修正。

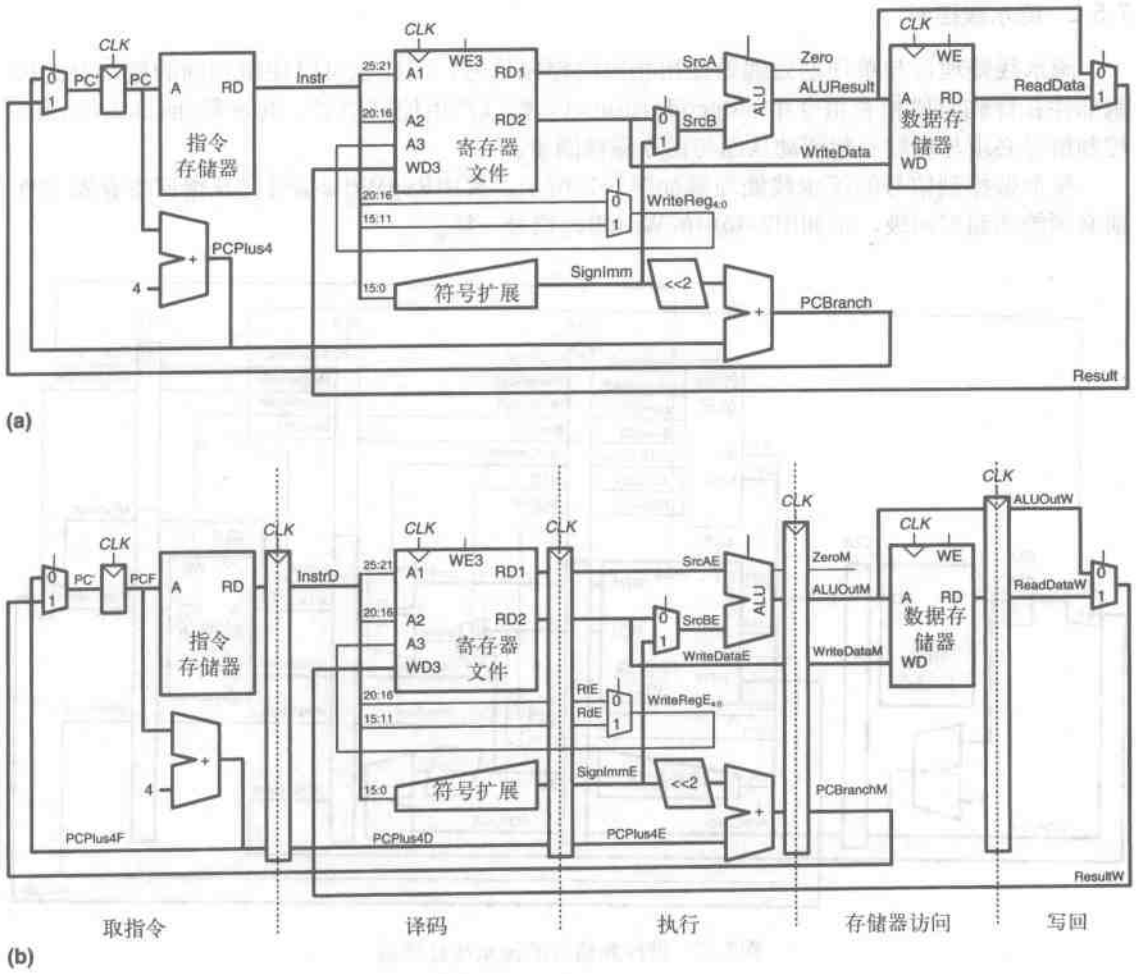


图7-45 单周期和流水线数据路径

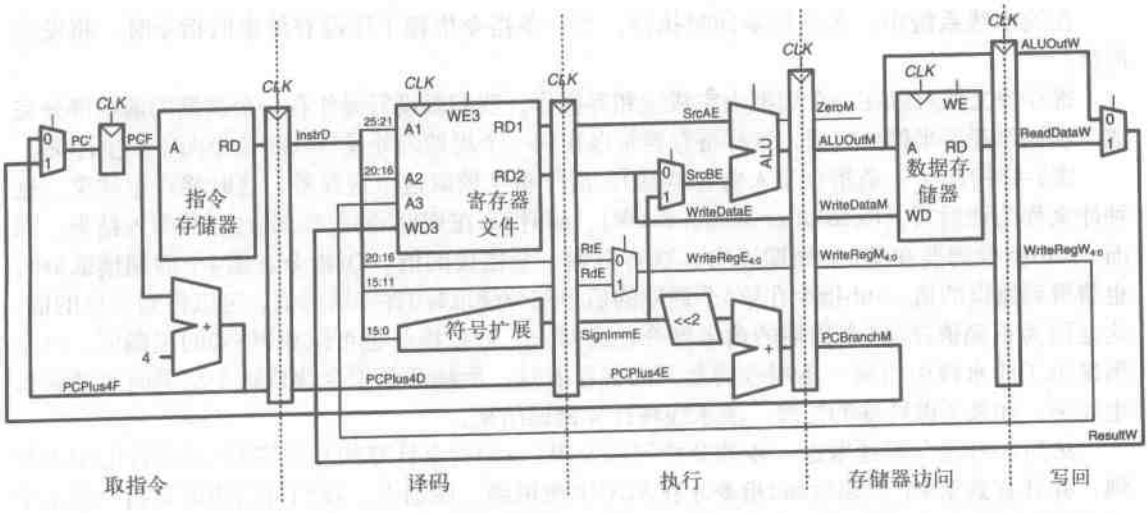


图7-46 正确的流水线数据路径

7.5.2 流水线控制

流水线处理器与单周期处理器使用相同的控制信号，因此也可以使用相同的控制器。控制部件在译码阶段检查指令中的opcode和funct字段以产生控制信号，其过程如7.3.2节介绍。控制信号必须与数据一起流动从而与指令保持同步。

整个带控制信号的流水线处理器如图7-47所示。其中RegWrite信号在反馈回寄存器文件前必须流动到写回级，正如图7-46中的WriteReg信号一样。

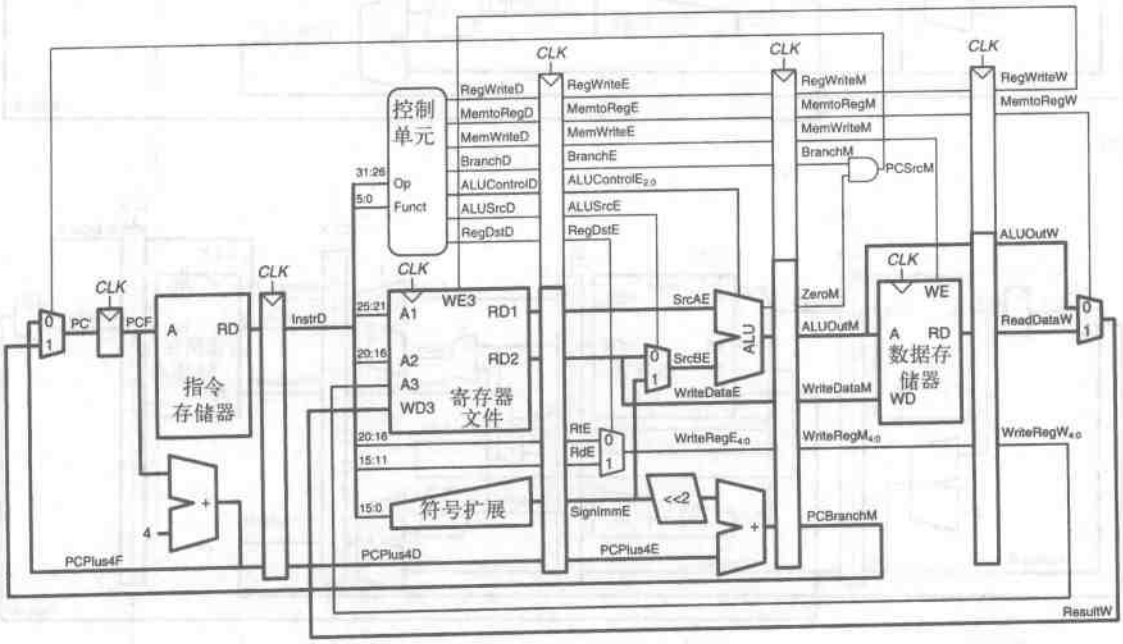


图7-47 带控制信号的流水线处理器

7.5.3 冲突

在流水线系统中，多条指令同时执行。当一条指令依赖于还没有结束的指令时，将发生冲突。

寄存器文件可以在一个周期内完成读和写操作。我们假设写操作在一个周期的前半部分完成，读操作在后半部分完成，这样寄存器可以在同一个周期内外成写入和读出而不产生冲突。

图7-48指出了一条指令写入寄存器$s0，后续指令读取这个寄存器，这时将产生冲突。这种冲突称为读后写（Read after Write, RAW）。add指令在周期5的前半部分向$s0写入结果。然而，and指令需要在第3个周期读出，这将得到一个错误的值。Or指令在第4个周期读取$s0，也将得到错误的值。sub指令在第5个周期的后半部分读取$s0寄存器的值，可以得到正确的值，这是因为正确值在第5个周期的前半部分已经写入。后续指令也可以获得$s0的正确值。这个图显示了流水线中当前一条指令需要写入寄存器时，后续两条指令读取这个结果时可能会发生冲突。如果不做特殊的处理，流水线将计算错误结果。

然而，对这个问题做进一步的分析可以发现：add指令计算的和在第3个周期就由ALU得到，并且直到第4个周期后and指令才在ALU中使用到。原则上，我们可以将前条指令的结果重定向到后条指令以化解RAW类型冲突，而不降低流水线的性能。在后续章节中讨论的情况

中，我们可能必须阻塞流水线以暂停后续指令执行，为前面指令获得计算结果赢得时间。无论何种情况下，必须对流水线进行处理以解决冲突问题，保证程序运行的正确性。

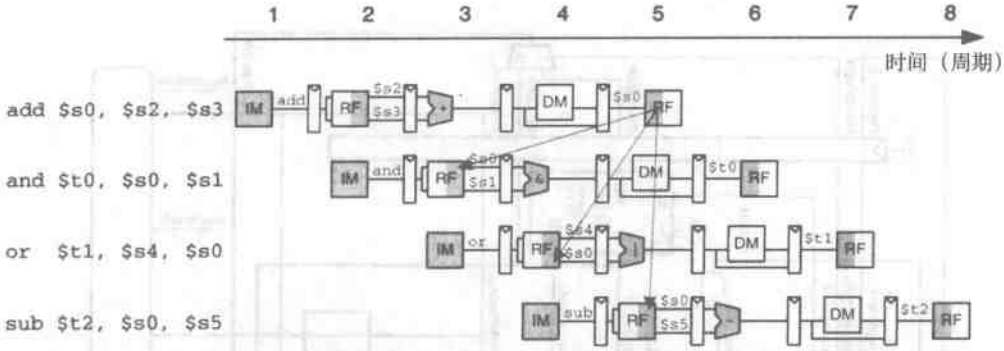


图7-48 说明冲突的抽象流水线示意图

冲突可以分为数据冲突和控制冲突。在当一条指令试图读取前条指令还未写回的寄存器将发生数据冲突。在取指令时还未确定下一条指令应取的地址将发生控制冲突。我们将在后续部分中为流水线处理器增加一个冲突单元以发现和正确化解冲突，从而保证处理器能正确执行程序。

使用重定向化解数据冲突

一些数据冲突可以通过将存储器访问阶段或写回阶段的结果重定向（forwarding）或旁路（bypassing）到执行阶段来化解。这需要在ALU的前端增加多路开关以选择从寄存器文件来的操作数，或从存储器级或写回级来的结果。图7-49显示了基本的设计原理。在周期4中，$s0寄存器从and指令的存储器级重定向到相关的and指令的执行级。在周期5，$s0从add指令的写回级重定向到相关的or指令的执行级。

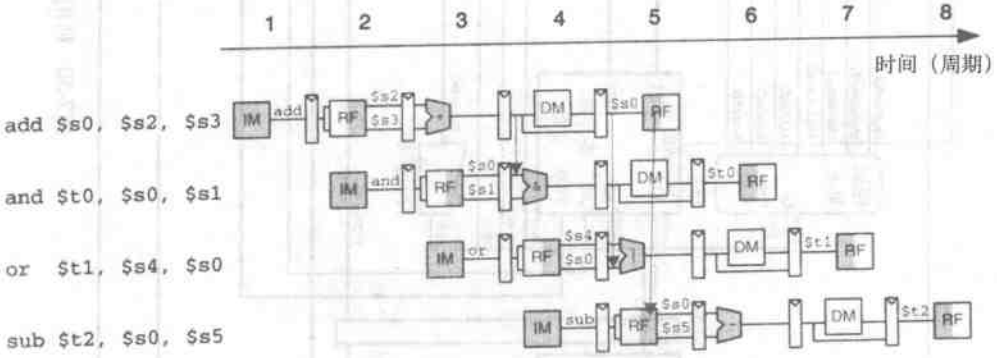


图7-49 说明数据重定向的抽象流水线示意图

重定向产生的条件是：执行阶段的其中一个源寄存器匹配存储器阶段或写回阶段的目的寄存器。图7-50修改了流水线处理器以支持重定向。其中增加了冲突检测单元和两个重定向多路开关。冲突检测部件接收执行阶段中指令的两个源寄存器和存储器与写回级指令的目的寄存器，并从存储器级和写回级接收RegWrite信号以明确目的寄存器是否的确要写入（例如，sw和beq指令不需要将结果写入寄存器，因此也不需要重定向）。注意到RegWrite信号是仅仅按照名字连接的。换句话说，为了保持电路图清楚不再将控制信号用长线从顶端连接到低端的冲突部件，而是使用一个带控制信号名称的短线连接。

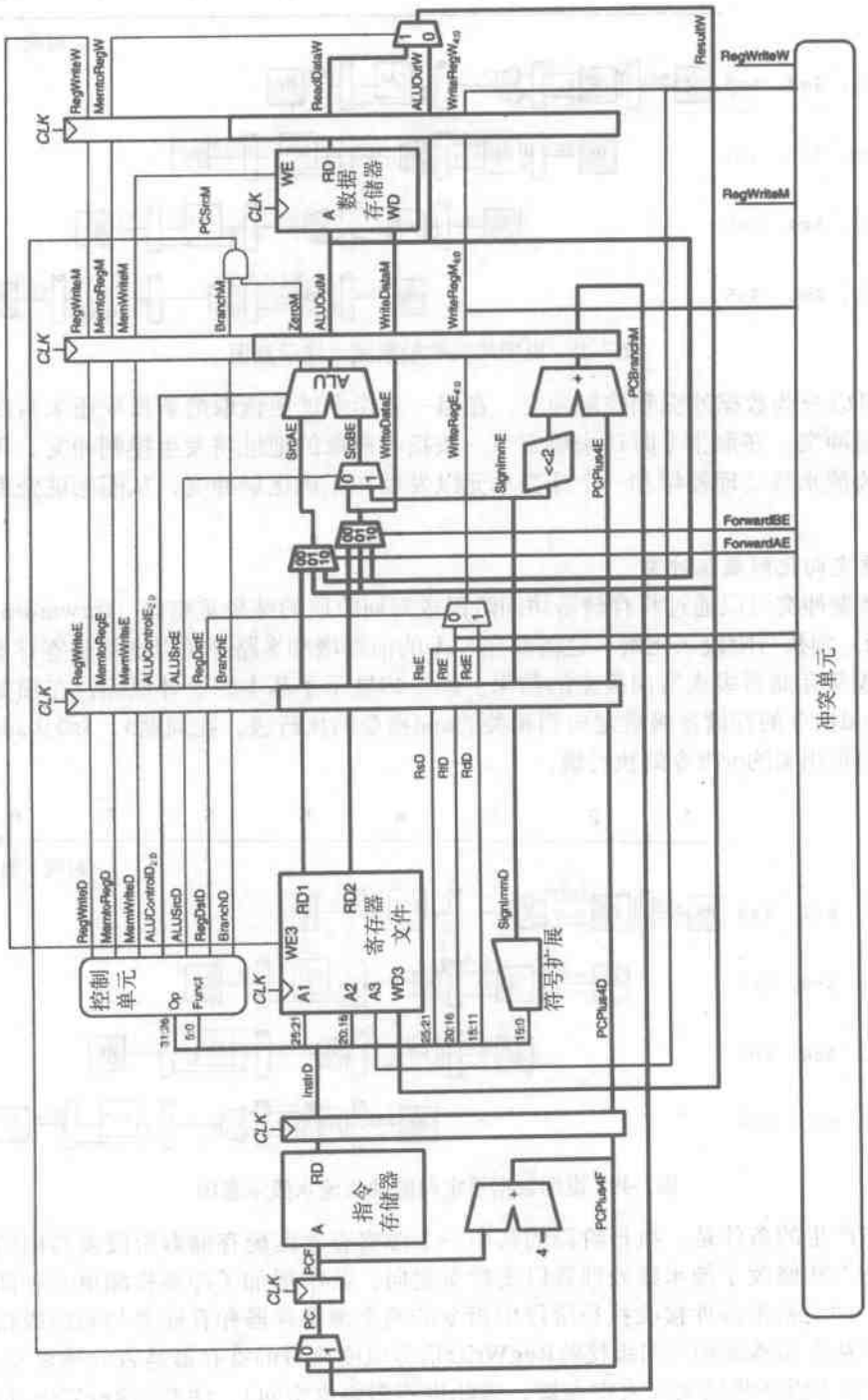


图7-50 利用重定向化解冲突的流水线处理器

冲突检测部件计算控制信号以确定重定向多路开关是选择来自寄存器文件的操作数，还是来自存储器级或写回级的结果。如果某一级的目的寄存器匹配源寄存器则重定向该级的结果。然而，$0寄存器始终为0，因此不必重定向。如果存储器级和写回级同时包含了匹配的目的寄存器，则存储器级具有更高的优先级，因为它包含了最近执行指令的结果。最后，针对SrcA的重定向逻辑功能如下所示。对于SrcB的重定向逻辑（ForwardBE）与之类似，只是以检查rt寄存器代替检查rs寄存器。

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
                          ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
                          ForwardAE = 01
else                      ForwardAE = 00
```

使用阻塞化解数据冲突

当指令的结果在执行阶段计算时，可以使用重定向方法化解RAW冲突。这是因为，结果可以重定向到后条指令的执行阶段。但是，lw指令直到存储器级后才能读取到数据，因此结果不能重定向到下一条指令的执行阶段。我们称lw指令有两个周期延迟，因为相关指令直到两个周期后才能使用到其结果。图7-51中指出了这个问题。lw指令在第4个周期的最后才从存储器中接收到数据。但是and指令在第4个周期的开始时就需要这个数据作为源操作数。使用重定向方法无法解决这种情况。

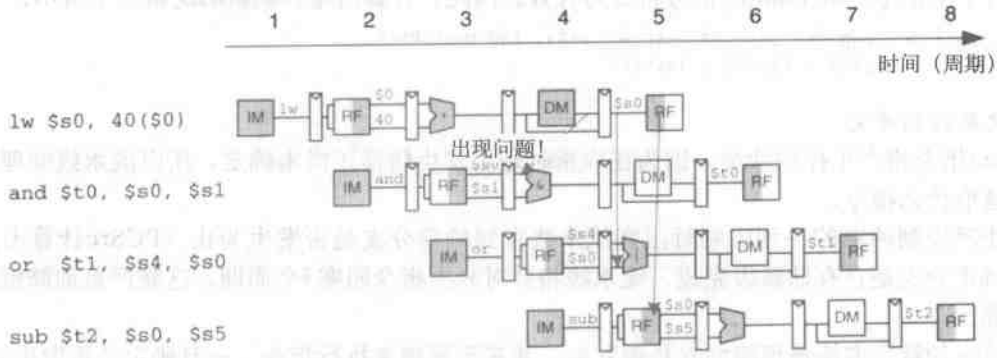


图7-51 说明lw重定向问题的抽象流水线示意图

另外一种解决方法是阻塞流水线：将操作挂起直至数据有效时。图7-52显示在译码级阻塞相关的and指令。该指令在第3个周期进入译码级，并一直阻塞到第4个周期位置。在此过程中，后续的or指令必须也保持在取指级，因为译码级已经满了。

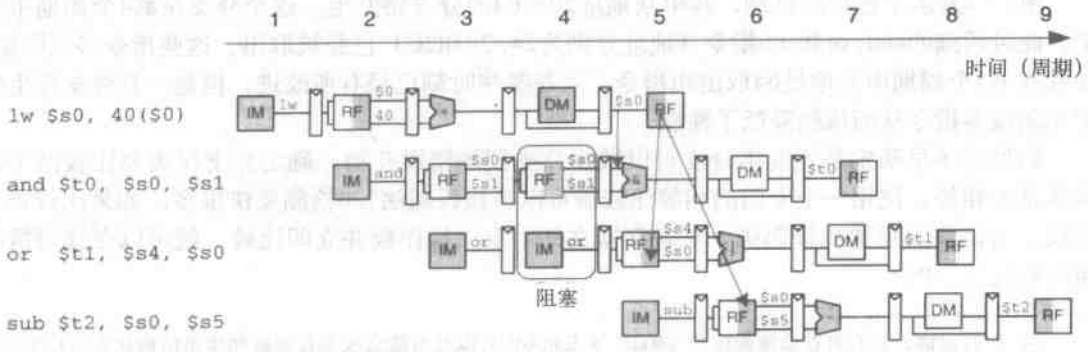


图7-52 通过阻塞方式解决冲突的抽象流水线示意图

在第5个周期时，lw指令的结果从写回级重定向到and指令的执行级。在第5个周期，or指令的$s0寄存器直接从寄存器文件中读取，而不需要重定向。

注意到执行级在第4个周期没有使用到。与之类似，存储器级和写回级也分别在第5个周期和第6个周期中也没有使用。这种沿着流水线未使用“级”的传播称为气泡（bubble），其特征类似与nop指令。气泡的产生是由译码阶段阻塞时对执行阶段产生无效的控制信号，使得气泡不发生操作，也不修改体系结构状态。

总而言之，可以通过禁止某一级的流水线寄存器实现阻塞流水线，此时寄存器的内容将不发生改变。当某一级流水线阻塞时，前面的各级也都应阻塞，这样后续的指令就不会丢失。在阻塞级后的流水线寄存器必须清除以防止错误信息向前传播。阻塞将降低性能，因此它们只有在必须的时候才能使用。

图7-53修改了流水线处理器为lw指令的数据相关增加阻塞功能。冲突部件将检查执行级中的指令。如果它是lw指令，而且其目的寄存器rtE匹配译码级中指令的任意一个源操作数（rsD或rtD），指令必须被阻塞在译码级直至源操作数准备好。

阻塞是通过对取指级和译码级的流水线寄存器增加使能（EN）和为执行级流水线寄存器增加同步复位/清除（CLR）的方式实现。当lw阻塞出现时，StallD和StallF信号有效以强制译码级和取指级流水线寄存器保持原有的值。FlushE也同时有效以清除执行级流水线寄存器中的内容，从而产生气泡[⊖]。

对于lw指令，MemtoReg信号将变为有效。因此，计算阻塞和刷新的逻辑如下所示：

```
lwstall = ((rsD  == rtE) OR (rtD == rtE)) AND MemtoRegE
StallF  = StallD = FlushE = lwstall
```

化解控制冲突

beq指令将产生控制冲突：因为在取指时是否发生转移还尚未确定，所以流水线处理器并不知道取什么指令。

处理控制冲突的一种机制时阻塞流水线直到确定分支是否发生为止（PCSrc计算出来）。因为确定分支是在存储器级完成，流水线将针对分支指令阻塞3个周期。这将严重地降低系统的性能。

另一种解决方法是预测分支是否发生，并基于预测来执行指令。一旦确定是否发生分支，而且如果预测错误的话，处理器将抛弃错误执行的指令。其中的一种特殊情况是，我们预测所有的分支都不会发生而只是简单地继续按照顺序执行程序。如果分支的确发生了，分支指令后的3条指令将通过清除这些指令的流水线寄存器方式而刷新（抛弃）。这些浪费的指令周期称为分支预测错误代价（branch misprediction penalty）。

图7-54显示了这样的机制，其中从地址20～64的分支将发生。这个分支在第4个周期中确定，此时后续的and, or和sub指令（地址分别为24, 28和2C）已经被取出。这些指令必须刷新，并且在第5个周期中从地址64取出slt指令。这在某些时刻已经有所改进，但是一旦分支发生需要刷新很多指令从而依然降低了性能。

如果能尽早确定是否发生分支可以减少分支预测错误开销。确定分支仅需要比较两个寄存器是否相等。使用一个专门的相等比较器可以将执行减法和0检测要快很多。如果比较器足够快，可以将其设置到译码级，从寄存器文件中取出操作数并立即比较，就可以在译码级结束时确定下一PC。

⊖ 严格地讲，只有寄存器使能信号（RsE，RtE和RdE）以及可能会改变存储器和体系结构状态的控制信号（RegWrite，MemWrite和Branch）需要清除；一旦这些信号被清除，气泡可能会包含随机数据，但是不会产生影响。

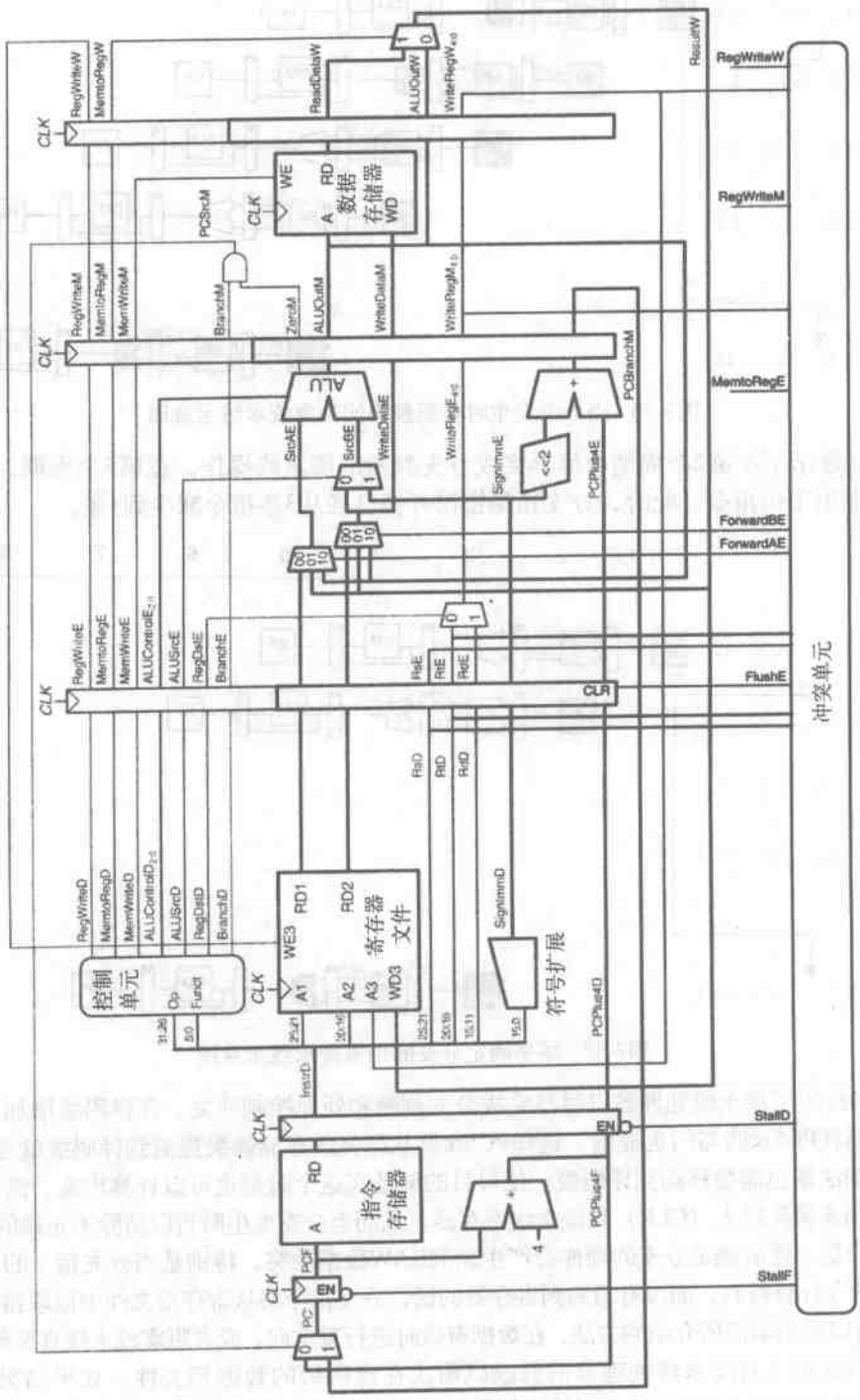


图7-53 通过阻塞方式解决lw指令数据冲突的流水线处理器

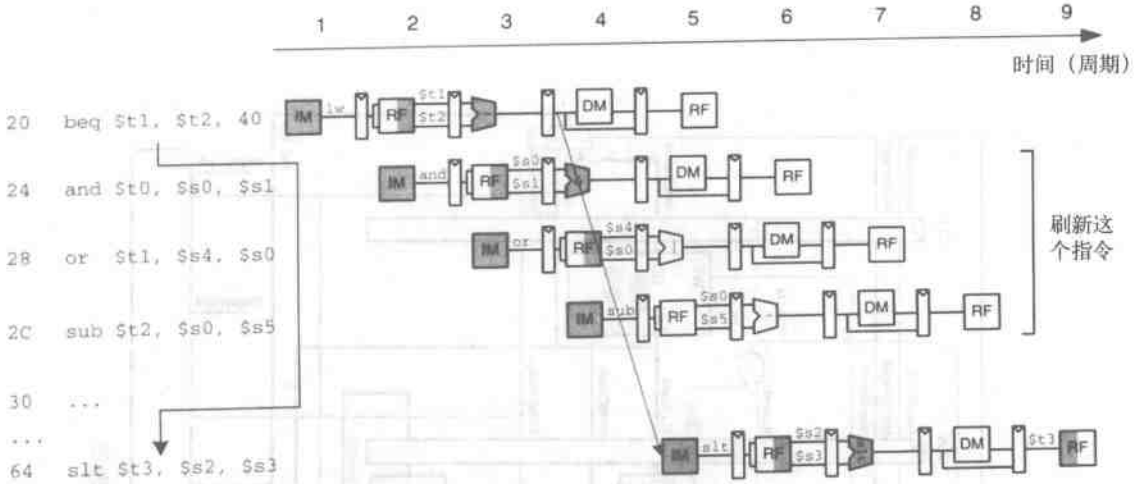


图7-54 当分支发生时刷新操作的抽象流水线示意图

图7-55显示了在第2个周期中尽早完成分支判断的流水线操作。在第3个周期，刷新了and指令而且取出了slt指令。此时，分支预测错误开销已经从3条指令减少到1条。

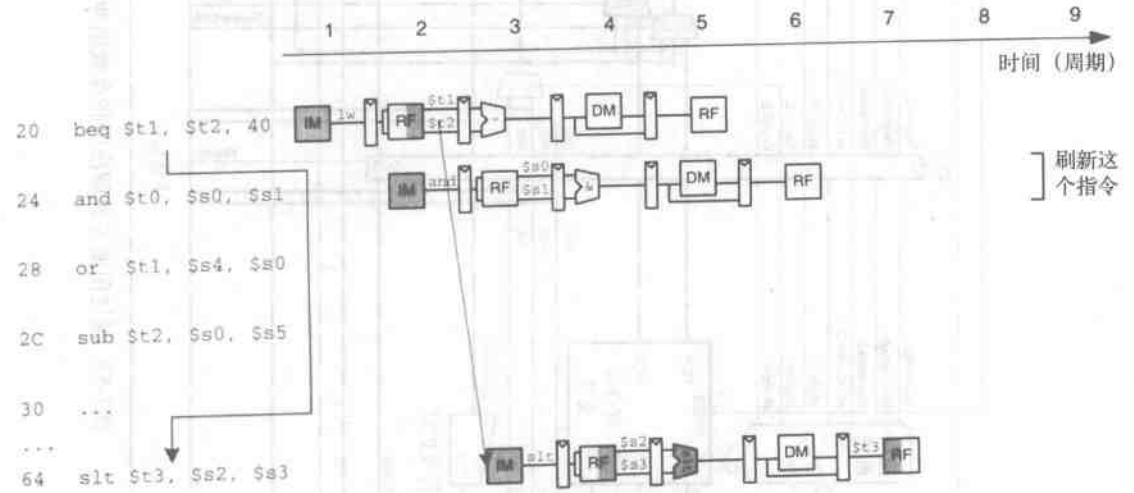


图7-55 尽早确定分支的抽象流水线示意图

图7-56修改了流水线处理器以尽早完成分支判断和处理控制冲突。在译码级增加了一个相等比较器，而且PCSrc的与门也提前，这样PCSrc就从原先的存储器级提前到译码级就可以确定了。PCBranch加法器也需要移动到译码级，使得目的地址在这个时刻也可以计算出来。需要增加连接PCSrcD的同步清除输入（CLR）到流水线寄存器，从而当分支发生时可以清除不正确的预取指令。

不幸的是，提前确定分支的硬件会产生新的RAW数据冲突。特别是当分支指令的一个源操作数由前条指令计算得到，而没有写回到寄存器的话，分支指令将从寄存器文件中读取错误的操作数值。我们可以采用前面所介绍的方法，在数据有效时进行重定向，或者阻塞流水线直至数据准备好。

图7-57显示了对流水线处理器的修改以解决在译码级的数据相关性。如果结果在写回级，它将在前半周期写入到寄存器，而读出操作是在后半周期，所以此时不存在冲突。如果ALU操作指令的结果在存储器级，可以将其重定向到相等比较器的两个多路开关。如果ALU指令的结果处于执行级或者lw指令的结果处于存储器级，流水线必须在译码级阻塞直至结果准备好。

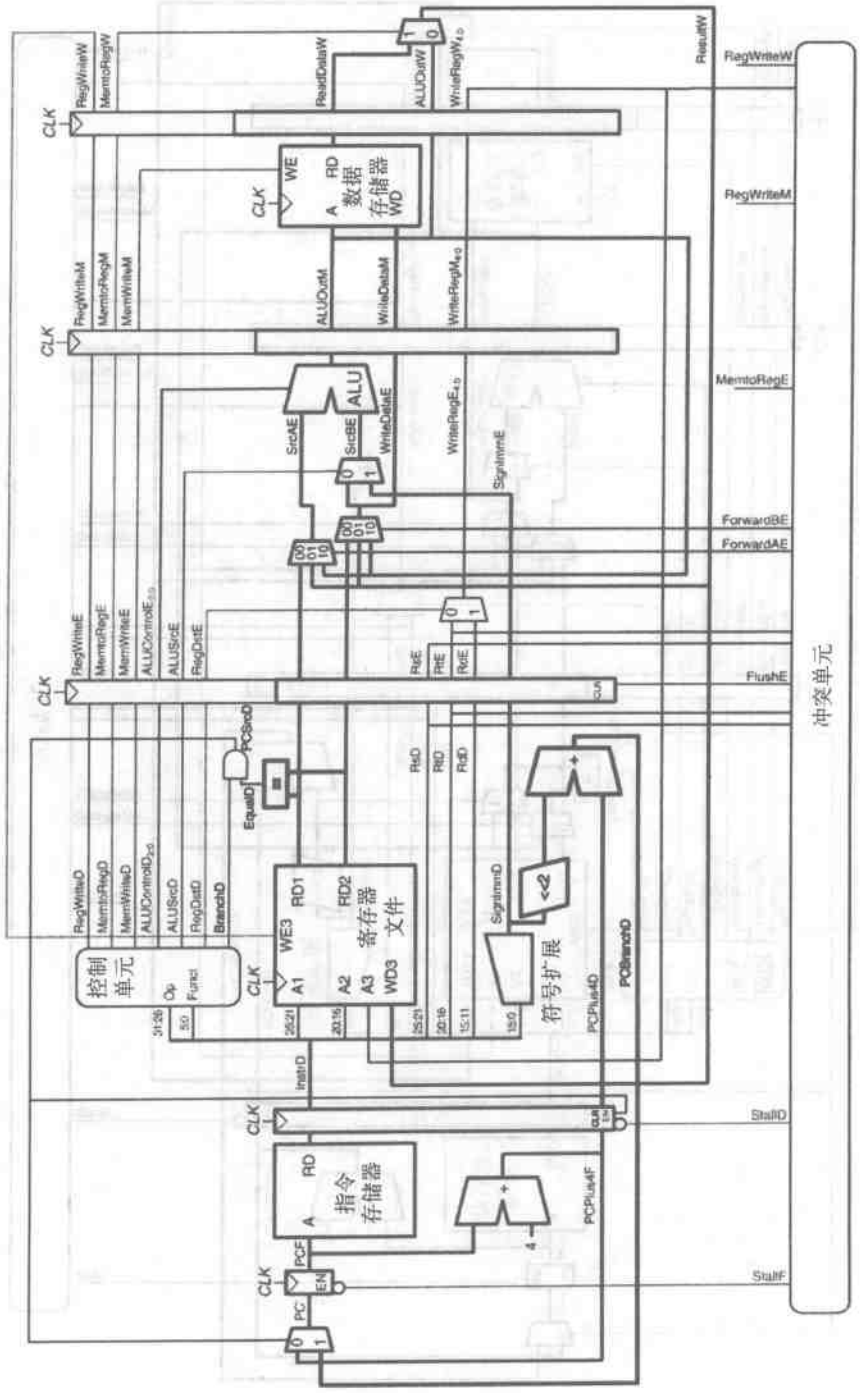


图7-56 处理分支控制冲突的流水线处理器

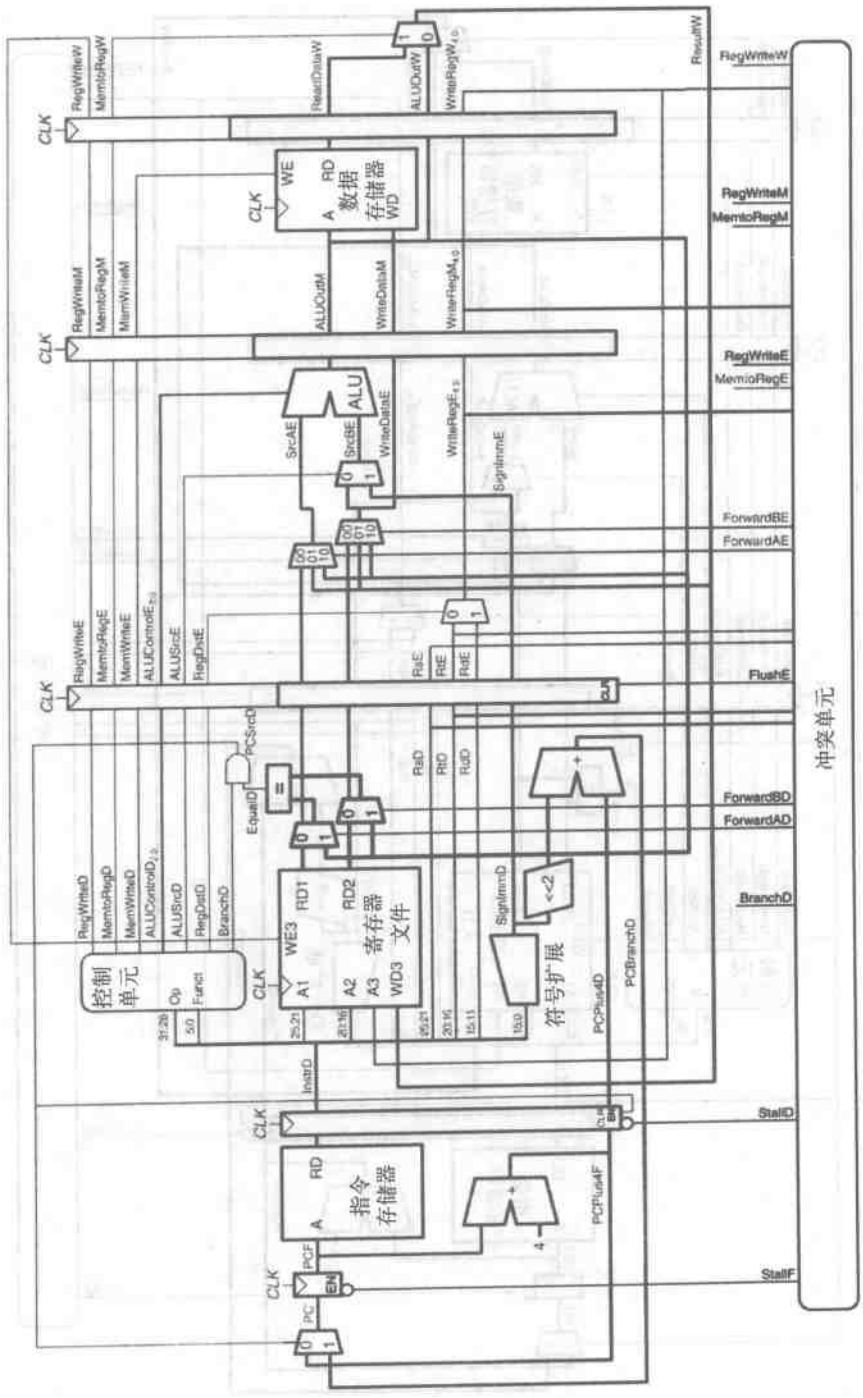


图7-57 处理分支指令的数据相关性的流水线处理器

译码级的重定向逻辑如下式给出。

```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

针对分支指令的阻塞检测逻辑如下所示。处理器必须在译码阶段完成转移判断。如果分支指令的任意一个源寄存器依赖于处于执行级的ALU指令，或者依赖与存储器级的lw指令，处理器将阻塞直至源操作数准备好。

```
branchstall =
  BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
                        OR
  BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

现在处理器阻塞可以会因为load或转移冲突：

```
StallF = StallD = FlushE = lwstall OR branchstall
```

冲突总结

当一条指令依赖于其他指令的结果，而此结果还未写入到寄存器文件时将产生RAW数据冲突。化解数据冲突的方法有两种：当结果已经计算出来时可以采用重定向方法，或者阻塞流水线直至结果可以使用。在要取下一条指令时还不能确定应该取何指令时将发生控制冲突。控制冲突可以通过下述方法化解：预测取应何指令，并在预测错误时刷新流水线。尽量将分支确定过程提前可以减少预测错误时刷新的指令数目。读者可以发现设计流水线处理器的主要挑战是理解指令间所有可能的相互关系并发现可能存在的所有冲突。图7-58显示了可以处理所有冲突的完整流水线处理器。

7.5.4 更多指令

在流水线处理器中增加新的指令非常类似于在单周期处理器中增加指令。然而，必须检测和化解新指令所带来的冲突。

在流水线处理器上增加addi指令和j指令需要增强控制器，如7.3.3节中所示，而且需要在数据路径中的转移多路开关后增加一个无条件跳转多路开关。类似于条件分支指令，无条件跳转也在译码阶段发生，在取指级的后续指令也应该刷新。这个刷新逻辑的设计留作习题7.29。

7.5.5 性能分析

理想的流水线处理器中CPI应该为1，这是因为在每个周期都将提交一条指令。然而阻塞或刷新将浪费一个周期，这使得CPI将稍高一些，并且和执行的特定程序密切相关。

例7.9 流水线处理器CPI

例7.7中考虑的SPECINT2000包含了大约25%的读出指令，10%的写入指令，11%的分支指令，2%的无条件跳转指令和52%的R类型指令。假设40%的读出指令后面跟随的指令需要立即使用其结果，从而产生阻塞。而四分之一的条件分支指令将产生判断错误，从而需要刷新。假设无条件转移指令需要总是刷新所有的后续指令。忽略其他冲突，请计算流水线处理器的平均CPI。

解： 所有指令的平均CPI是各种指令的CPI乘以其所占程序的比例。读出指令在无相关性时需要一个周期，在有相关性时需要两个周期，因此其CPI为 (0.6) (1) + (0.4)(2) = 1.4。分支指令在预测正确时需要一个周期，在预测错误时需要两个周期，因此其CPI为 (0.75) (1) + (0.25)(2) = 1.25。无条件转移指令的CPI总是为2。其他指令的CPI均为1。因此，对于此测试基准程序，平均的CPI为$(0.25)\times(1.4)+(0.1)\times(1)+(0.11)\times(1.25)+(0.52)\times(1)=1.15$。

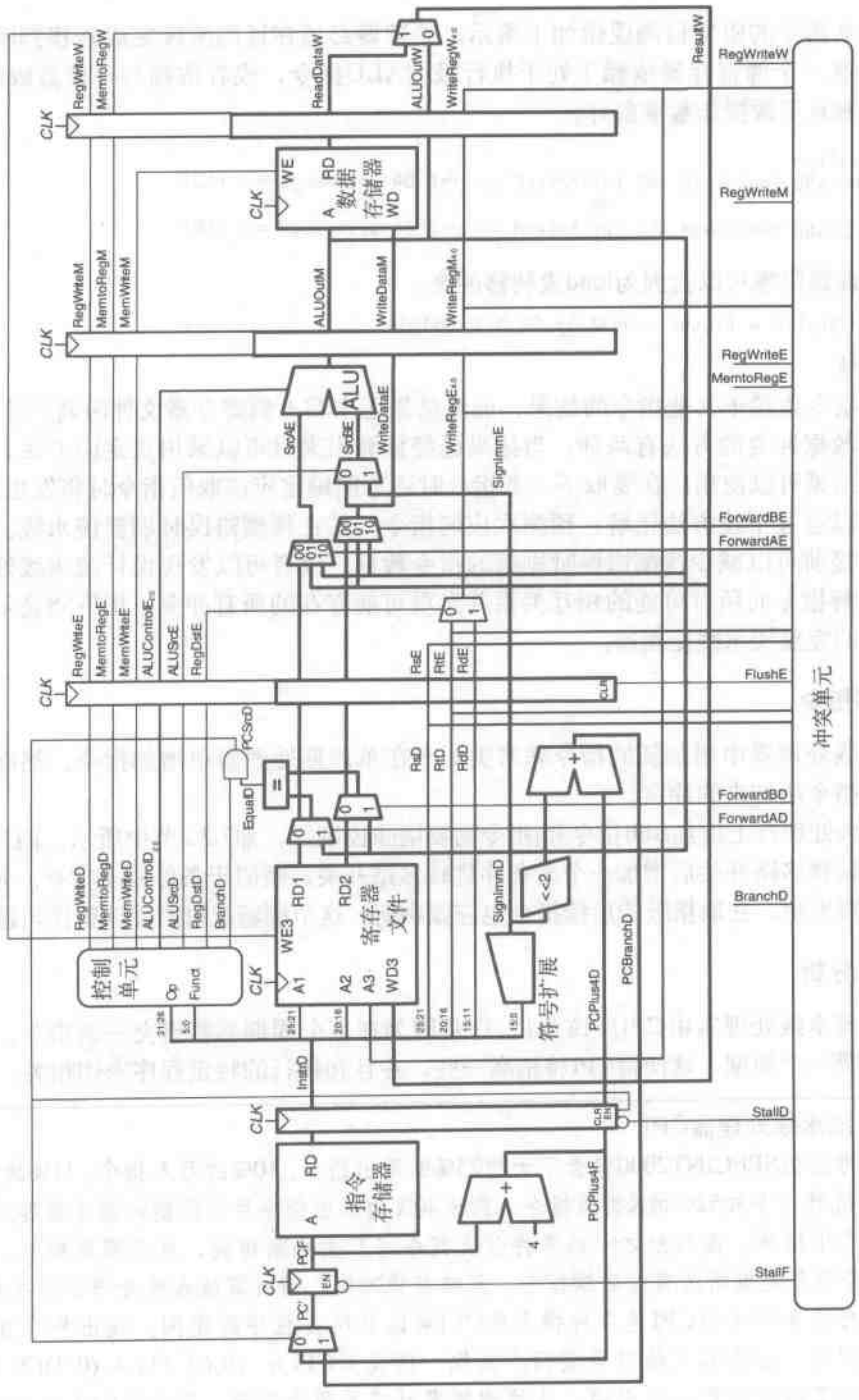


图7-58 处理所有冲突的流水线处理器

我们可以通过考虑图7-58中各级流水线的关键路径来确定周期时间。考虑到寄存器文件在写回级周期的前半部分写入，在译码级周期中的后半部分读出。因此，译码级和写回级的周期时间需要两倍于在半个周期内完成工作。

$$T_c = \max \begin{pmatrix} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{pmatrix} \begin{matrix} \text{取指} \\ \text{译码} \\ \text{执行} \\ \text{存储器} \\ \text{写回} \end{matrix} \tag{7-5}$$

例7.10 处理器性能比较

Ben需要将流水线处理器与例7.8中的单周期处理器和多周期处理器进行性能比较。大部分的逻辑组件延迟在表7-6中给出。其他组件包括：相等比较器延迟为40ps，与门为15ps，寄存器文件写为100ps，存储器写为220ps。请帮助Ben比较在各种处理器上执行SPECINT2000基准程序中10^{10}条指令的时间。

解：根据式（7-5），流水线处理器的周期时间为T_{c3} = max(30 + 250 + 20, 2 (150 + 25 + 40 + 15 + 25 + 20), 30+25+25+200+20, 30+220+20, 2(30+25+100))=550ps。由式（7-1），可知总的执行时间为$T_3 = (10^{10}\text{条指令}) \times (1.15\text{周期/指令}) \times (550 \times 10^{-12}\text{秒/周期}) = 63.3$秒。请与单周期处理器的92.5秒和多周期处理器的133.9秒比较。

流水线处理器明显将其他处理器更快。然而，它相对于单周期处理器的优势不是理想中五级流水线所带来的5倍加速比。流水线的冲突导致了比较小的CPI开销。更为显著的是寄存器的时序开销（包括clk-to-Q和建立时间）将对流水线的每一级都有影响，而非整个数据路径。时序开销限制了可能从流水线中得到的益处。

细心的读者可能已经发现译码级明显比其他级都慢，因为寄存器读和写、分支比较都需要在半个周期内完成。也许将分支比较放到译码级不是一个好的主意。如果在执行级完成分支判断，将可能导致预测错误时刷新两条指令，从而使得CPI有少量增加。但是将显著减少周期时间，从而提升总的加速比。

流水线处理器的硬件需求类似于单周期处理器。但是它需要增加大量的流水线寄存器和用于化解冲突的多路开关和控制逻辑。

7.6 硬件描述语言表示*

本节将介绍单周期MIPS处理器的硬件描述语言代码。该处理器可以支持前面所介绍的所有指令，包括addi和j指令。对于中等复杂程度的系统而言，这些代码可以作为一个良好的编码练习。对于多周期处理器和流水线处理器的硬件描述语言程序设计留作习题7.22和7.23。

在本节中，指令和数据存储器从主处理器中剥离开，而通过地址和数据总线与之连接。这样实现将更加符合实际情况，因为现代真实的处理器都有外部存储器。这也体现出处理器如何和外部世界进行通信。

处理器包括了数据路径和控制器。控制器由主译码器和ALU译码器构成。图7-59显示了带外部存储器接口的单周期MIPS处理器的框图。

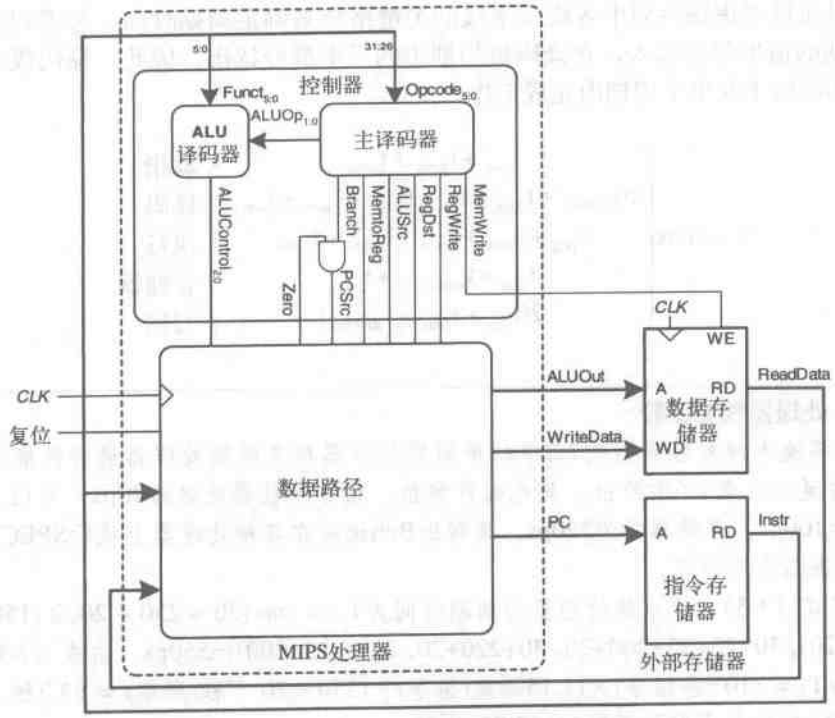


图7-59 具有外部存储器接口的MIPS单周期处理器

硬件描述语言代码分为几个部分。7.6.1节中的代码是单周期处理器的数据路径和控制器。7.6.2节中描述了微结构中的通用模块，例如寄存器、多路开关等。7.6.3节介绍了测试程序和外部存储器。硬件描述语言的电子版本可以从此书的Web网站上下载（请参见前言）。

7.6.1 单周期处理器

单周期MIPS处理器的主模块由下述硬件描述语言例子给出。

HDL例子7.1 单周期MIPS处理器

Verilog

```
module mips (input          clk, reset,
             output [31:0]  pc,
             input  [31:0]  instr,
             output         memwrite,
             output [31:0]  aluout, writedata,
             input  [31:0]  readdata);

  wire        memtoreg, branch,
              alusrc, regdst, regwrite, jump;
  wire [2:0] alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- single cycle MIPS processor
  port(clk, reset:        in  STD_LOGIC;
       pc:                out STD_LOGIC_VECTOR(31 downto 0);
       instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       memwrite:          out STD_LOGIC;
       aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(op, funct:         in  STD_LOGIC_VECTOR(5 downto 0);
         zero:              in  STD_LOGIC;
         memtoreg, memwrite: out STD_LOGIC;
         pcsrc, alusrc:     out STD_LOGIC;
         regdst, regwrite:  out STD_LOGIC;
         jump:              out STD_LOGIC;
         alucontrol:        out STD_LOGIC_VECTOR(2 downto 0));
 end component;
 component datapath
   port(clk, reset:        in  STD_LOGIC;
        memtoreg, pcsrc:   in  STD_LOGIC;
        alusrc, regdst:    in  STD_LOGIC;
        regwrite, jump:    in  STD_LOGIC;
        alucontrol:        in  STD_LOGIC_VECTOR(2 downto 0);
        zero:              out STD_LOGIC;
        pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
        instr:             in  STD_LOGIC_VECTOR(31 downto 0);
        aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
        readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
 end component;
 signal memtoreg, alusrc, regdst, regwrite, jump, pcsrc:
  STD_LOGIC;
 signal zero: STD_LOGIC;
 signal alucontrol: STD_LOGIC_VECTOR(2 downto 0);
begin
  cont: controller port map(instr(31 downto 26), instr
                            (5 downto 0), zero, memtoreg,
                            memwrite, pcsrc, alusrc, regdst,
                            regwrite, jump, alucontrol);
   dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc,
                         regdst, regwrite, jump, alucontrol,
                         zero, pc, instr, aluout, writedata,
                         readdata);
end;
```

HDL例子7.2 控制器

Verilog

```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc, alusrc,
                  output       regdst, regwrite,
                  output       jump,
                  output [2:0] alucontrol);

 wire [1:0] aluop;
 wire       branch;

 maindec md(op, memtoreg, memwrite, branch,
            alusrc, regdst, regwrite, jump,
```

```
            aluop);
  aludec ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
 port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
      zero:               in  STD_LOGIC;
      memtoreg, memwrite: out STD_LOGIC;
      pcsrc, alusrc:      out STD_LOGIC;
      regdst, regwrite:   out STD_LOGIC;
      jump:               out STD_LOGIC;
      alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
 component maindec
  port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
       memtoreg, memwrite: out STD_LOGIC;
       branch, alusrc:     out STD_LOGIC;
       regdst, regwrite:   out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out STD_LOGIC_VECTOR(1 downto 0));
 end component;
 component aludec
  port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
       aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
 end component;
 signal aluop:  STD_LOGIC_VECTOR(1 downto 0);
 signal branch: STD_LOGIC;
begin
 md: maindec port map(op, memtoreg, memwrite, branch,
                      alusrc, regdst, regwrite, jump, aluop);
 ad: aludec port map(funct, aluop, alucontrol);

 pcsrc <= branch and zero;
end;
```

HDL例子7.3 主译码器

Verilog

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always@(*)
   case(op)
     6'b000000: controls <= 9'b110000010; //Rtyp
     6'b100011: controls <= 9'b101001000; //LW
     6'b101011: controls <= 9'b001010000; //SW
     6'b000100: controls <= 9'b000100001; //BEQ
     6'b001000: controls <= 9'b101000000; //ADDI
     6'b000010: controls <= 9'b000000100; //J
```

```
    default:   controls <= 9'bxxxxxxxxx; //???
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity maindec is -- main control decoder
 port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
      memtoreg, memwrite: out STD_LOGIC;
      branch, alusrc:     out STD_LOGIC;
      regdst, regwrite:   out STD_LOGIC;
      jump:               out STD_LOGIC;
      aluop:              out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
 signal controls: STD_LOGIC_VECTOR(8 downto 0);
begin
 process(op) begin
  case op is
     when "000000" => controls <= "110000010"; -- Rtyp
     when "100011" => controls <= "101001000"; -- LW
     when "101011" => controls <= "001010000"; -- SW
     when "000100" => controls <= "000100001"; -- BEQ
     when "001000" => controls <= "101000000"; -- ADDI
     when "000010" => controls <= "000000100"; -- J
     when others   => controls <= "---------"; -- illegal op
  end case;
 end process;

 regwrite <= controls(8);
 regdst   <= controls(7);
 alusrc   <= controls(6);
 branch   <= controls(5);
 memwrite <= controls(4);
 memtoreg <= controls(3);
 jump     <= controls(2);
 aluop    <= controls(1 downto 0);
end;
```

HDL例子7.4 ALU译码器

Verilog

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [2:0] alucontrol);

always@(*)
 case(aluop)
  2'b00: alucontrol <= 3'b010; // add
  2'b01: alucontrol <= 3'b110; // sub
  default: case(funct)         // RTYPE
      6'b100000: alucontrol <= 3'b010; // ADD
      6'b100010: alucontrol <= 3'b110; // SUB
      6'b100100: alucontrol <= 3'b000; // AND
      6'b100101: alucontrol <= 3'b001; // OR
      6'b101010: alucontrol <= 3'b111; // SLT
      default:   alucontrol <= 3'bxxx; // ???
    endcase
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity aludec is -- ALU control decoder
  port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
       aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;
```

```
architecture behave of aludec is
begin
 process (aluop, funct) begin
  case aluop is
    when "00" => alucontrol <= "010"; -- add (for lb/sb/addi)
    when "01" => alucontrol <= "110"; -- sub (for beq)
    when others => case funct is          -- R-type instructions
                       when "100000" => alucontrol <=
                         "010"; -- add
                       when "100010" => alucontrol <=
                         "110"; -- sub
                       when "100100" => alucontrol <=
                         "000"; -- and
                       when "100101" => alucontrol <=
                         "001"; -- or
                       when "101010" => alucontrol <=
                         "111"; -- slt
                       when others   => alucontrol <=
                         "---"; -- ???
                   end case;
  end case;
 end process;
end;
```

HDL例子7.5 数据路径

Verilog

```
module datapath (input          clk, reset,
                 input          memtoreg, pcsrc,
                 input          alusrc, regdst,
                 input          regwrite, jump,
                 input  [2:0]   alucontrol,
                 output         zero,
                 output [31:0]  pc,
                 input  [31:0]  instr,
                 output [31:0]  aluout, writedata,
                 input  [31:0]  readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
  wire [31:0] srca, srcb;
  wire [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1 (pc, 32'b100, pcplus4);
  s12         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 result, srca, writedata);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata,
                     memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc,
                      srcb);
  alu         alu(srca, srcb, alucontrol,
                  aluout, zero);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use
IEEE.STD_LOGIC_ARITH.all;
```

```
entity datapath is -- MIPS datapath
  port(clk, reset:        in     STD_LOGIC;
       memtoreg, pcsrc:   in     STD_LOGIC;
       alusrc, regdst:    in     STD_LOGIC;
       regwrite, jump:    in     STD_LOGIC;
       alucontrol:        in     STD_LOGIC_VECTOR(2 downto 0);
       zero:              out    STD_LOGIC;
       pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
       instr:             in     STD_LOGIC_VECTOR(31 downto 0);
       aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
    port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
         alucontrol: in  STD_LOGIC_VECTOR(2 downto 0);
         result:     buffer STD_LOGIC_VECTOR(31 downto 0);
         zero:       out STD_LOGIC);
  end component;
  component regfile
    port(clk:           in  STD_LOGIC;
         we3:           in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
 end component;
 signal writereg: STD_LOGIC_VECTOR(4 downto 0);
 signal pcjump, pcnext, pcnextbr,
        pcplus4, pcbranch: STD_LOGIC_VECTOR(31 downto 0);
 signal signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
 signal srca, srcb, result: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch,
                                         pcsrc, pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump,
                                       pcnext);

  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                  instr(20 downto 16), writereg, result, srca,
                       writedata);
  wrmux: mux2 generic map(5) port map(instr(20 downto 16),
  instr(15 downto 11), regdst, writereg);
  resmux: mux2 generic map(32) port map(aluout, readdata,
                                        memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);

  -- ALU logic
  srcbmux: mux2 generic map (32) port map(writedata, signimm,
                                          alusrc, srcb);
  mainalu: alu port map(srca, srcb, alucontrol, aluout, zero);
end;
```

7.6.2 通用模块

这一节描述了可以用于任意MIPS微结构的通用模块，包括寄存器文件，加法器，左移单元，符号扩展单元，可复位触发器。ALU的硬件描述语言程序留在习题5.9。

HDL例子7.6 寄存器文件

Verilog

```
module regfile(input         clk,
               input         we3,
               input  [4:0]  ra1, ra2, wa3,
               input  [31:0] wd3,
               output [31:0] rd1, rd2);

  reg [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 0 hardwired to 0

  always @(posedge clk)
    if(we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:          in  STD_LOGIC;
       we3:          in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
       wd3:          in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array(31 downto 0) of STD_LOGIC_VECTOR(31
                        downto 0);
  signal mem: ramtype;
begin
  -- three-ported register file
  -- read two ports combinationally
  -- write third port on rising edge of clock
  process(clk) begin
   if clk'event and clk = '1' then
    if we3 = '1' then mem(CONV_INTEGER(wa3)) <= wd3;
    end if;
   end if;
  end process;
  process(ra1, ra2) begin
    if(conv_integer(ra1) = 0) then rd1 <= X"00000000";
       -- register 0 holds 0
    else rd1 <= mem(CONV_INTEGER(ra1));
    end if;
    if(conv_integer(ra2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(CONV_INTEGER(ra2));
    end if;
  end process;
end;
```

HDL例子7.7 加法器

Verilog

```
module adder (input  [31:0] a, b,
              output [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity adder is -- adder
  port (a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL例子7.8 左移2位单元（乘以4）

Verilog

```
module sl2 (input  [31:0] a,
            output [31:0] y);

  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sl2 is -- shift left by 2
  port (a: in  STD_LOGIC_VECTOR (31 downto 0);
        y: out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of sl2 is
begin
  y <= a(29 downto 0) & "00";
end;
```

HDL例子7.9 符号扩展单元

Verilog

```
module signext (input  [15:0] a,
                output [31:0] y);

  assign y = {{16{a[15]}}, a};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity signext is -- sign extender
port(a: in  STD_LOGIC_VECTOR (15 downto 0);
     y: out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of signext is
begin
```

```
  y <= X"0000" & a when a(15) = '0' else X"ffff" & a;
end;
```

HDL例子7.10 可复位触发器

Verilog

```
module flopr #(parameter WIDTH = 8)
              (input                  clk, reset,
               input      [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);

  always @(posedge clk, posedge reset)
    if(reset) q <= 0;
    else      q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use
IEEE.STD_LOGIC_ARITH.all;
entity flopr is -- flip-flop with synchronous reset
  generic(width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then q <= CONV_STD_LOGIC_VECTOR(0, width);
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

HDL例子7.11 二选一多路开关

Verilog

```
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

7.6.3 测试程序

MIPS测试程序将加载一段MIPS可执行代码到存储器中。图7-60中的可执行代码通过计算检测所有指令，只有当所有指令的功能实现完全正确才能得到正确的结果。对于该图中的例子，如果程序完全运行正确的话应向地址84的存储器写入数值7，否则就表明硬件设计中有问题。这种测试访问称为随机测试（ad hoc testing）。

机器码存储在一个十六进制文件memfile.dat中（见图7-61）。这个文件在模拟时由测试程序加载。这个文件包含了指令的机器码，其中每条指令对应一行。

```
# mipstest.asm
# David_Harris@hmc.edu 9 November 2005
#
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#          Assembly              Description              Address    Machine
main:      addi $2, $0, 5        # initialize $2 = 5      0          20020005
           addi $3, $0, 12       # initialize $3 = 12     4          2003000c
           addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
           or   $4, $7, $2       # $4 <= 3 or 5 = 7       c          00e22025
           and  $5, $3, $4       # $5 <= 12 and 7 = 4     10         00642824
           add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
           beq  $5, $7, end      # shouldn't be taken     18         10a7000a
           slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
           beq  $4, $0, around   # should be taken        20         10800001
           addi $5, $0, 0        # shouldn't happen       24         20050000
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
           add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
           sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
           sw   $7, 68($3)       # [80] = 7               34         ac670044
           lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
           j    end              # should be taken        3c         08000011
           addi $2, $0, 1        # shouldn't happen       40         20020001
end:       sw   $2, 84($0)       # write adr 84 = 7       44         ac020054
```

图7-60 MIPS测试程序的汇编指令和机器码

```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

图7-61 memfile.dat的内容

测试程序、顶层MIPS模块和外部存储器的硬件描述语言由下述例子给出。在本例子中存储器均包含了64个字。

HDL例子7.12 MIPS测试程序

Verilog

```
module testbench();

  reg      clk;
  reg      reset;

  wire [31:0] writedata, dataadr;
  wire        memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
```

```
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @ (negedge clk)
    begin
      if (memwrite) begin
        if (dataadr === 84 & writedata === 7) begin
          $display ("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display ("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:         in  STD_LOGIC;
         writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
         memwrite:           out STD_LOGIC);
  end component;
  signal writedata, dataadr:   STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, memwrite: STD_LOGIC;

begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, writedata, dataadr, memwrite);

  -- Generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 84
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and memwrite = '1') then
     if (conv_integer(dataadr) = 84 and conv_integer
         (writedata) = 7) then
        report "Simulation succeeded";
      elsif (dataadr /= 80) then
        report "Simulation failed";
```

```
      end if;
    end if;
  end process;
end;
```

HDL例子7.13 MIPS顶层模块

Verilog

```
module top(input         clk, reset,
           output [31:0] writedata, dataadr,
           output        memwrite);

  wire [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata,
            readdata);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity top is -- top-level design for testing
  port(clk, reset:          in     STD_LOGIC;
       writedata, dataadr:  buffer STD_LOGIC_VECTOR(31 downto
                                                    0);
       memwrite:            buffer STD_LOGIC);
end;

architecture test of top is
  component mips
    port(clk, reset:        in  STD_LOGIC;
         pc:                out STD_LOGIC_VECTOR(31 downto 0);
         instr:             in  STD_LOGIC_VECTOR(31 downto 0);
         memwrite:          out STD_LOGIC;
         aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
         readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
    port(a:  in  STD_LOGIC_VECTOR(5 downto 0)
         rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
    port(clk, we: in  STD_LOGIC;
         a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
         rd:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal pc, instr,
         readdata: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  mips1: mips port map(clk, reset, pc, instr, memwrite,
                       dataadr, writedata, readdata);
  imem1: imem port map(pc(7 downto 2), instr);
  dmem1: dmem port map(clk, memwrite, dataadr, writedata,
                       readdata);
end;
```

HDL例子7.14 MIPS数据存储器

```
module dmem(input         clk, we,
            input  [31:0] a, wd,
            output [31:0] rd);

  reg [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always @(posedge clk)
    if(we)
```

```
      RAM[a[31:2]] <= wd;
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity dmem is -- data memory
 port(clk, we: in  STD_LOGIC;
      a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
      rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of dmem is
begin
  process is
    type ramtype is array(63 downto 0) of STD_LOGIC_VECTOR
                          (31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if clk'event and clk = '1' then
          if(we = '1') then mem(CONV_INTEGER(a(7 downto
                                             2))) := wd;
          end if;
      end if;
      rd <= mem(CONV_INTEGER(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

HDL例子7.15 MIPS指令存储器

Verilog

```
module imem(input  [5:0]  a,
            output [31:0] rd);

  reg[31:0] RAM[63:0];

  initial
    begin
      $readmemh("memfile.dat",RAM);
    end

  assign rd = RAM[a]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;

entity imem is -- instruction memory
  port(a:  in  STD_LOGIC_VECTOR(5 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable index, result: integer;
    type ramtype is array(63 downto 0) of STD_LOGIC_VECTOR
                          (31 downto 0);
```

```
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop - set all contents low
      mem(conv_integer(i)) := CONV_STD_LOGIC_VECTOR(0, 32);
    end loop;
    index := 0;
    FILE_OPEN(mem_file, "C:/mips/memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
            result := result*16 + character'pos(ch) -
                      character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := result*16 + character'pos(ch) -
                    character'pos('a') + 10;
        else report "Format error on line" & integer'image
             (index) severity error;
        end if;
      end loop;
      mem(index) := CONV_STD_LOGIC_VECTOR(result, 32);
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(CONV_INTEGER(a));
      wait on a;
    end loop;
  end process;
end;
```

7.7 异常*

6.7.2节中介绍了异常，它将引起非正常的程序流变化。在本节中，我们将增强多周期处理器以支持两类异常：未定义指令和算术溢出。在其他微结构中支持异常也遵循类似原则。

如6.7.2节所述，当发生异常时，处理器将PC复制到EPC寄存器中并将异常代码存储在原因寄存器中以标识异常来源。异常代码中0x28表示未定义指令，0x30表示溢出（见表6-7）。处理器将跳转到位于0x80000180处的异常处理程序。作为响应异常的代码，异常处理程序是操作系统的一个部分。

同时，6.7.2节中指出异常寄存器是协处理器0的一个部分。该协处理器也是MIPS处理器中用于处理系统功能的部分。该协处理器中最多可定义32个特殊用途寄存器，包括EPC寄存器和原因寄存器。异常处理程序可以使用mfc0(从协处理器0移动数据)指令将这些特殊用途寄存器拷贝到寄存器文件中。原因寄存器是协处理器0的寄存器13；EPC寄存器是协处理器0的寄存器14。

为了处理异常，我们必须在数据路径中增加EPC寄存器和原因寄存器，并扩展PCSrc多路开关以接收异常处理程序地址，如图7-62所示。这两个新寄存器具有写使能（EPCWrite和CauseWrite）以在发生异常时存储PC和异常原因。由一个多路开关根据异常为原因寄存器选择合适的代码。ALU必须产生溢出信号，如5.2.4节所示[⊖]。

⊖ 严格地讲，ALU仅应针对add和sub产生溢出信号，而不对其他ALU指令产生。

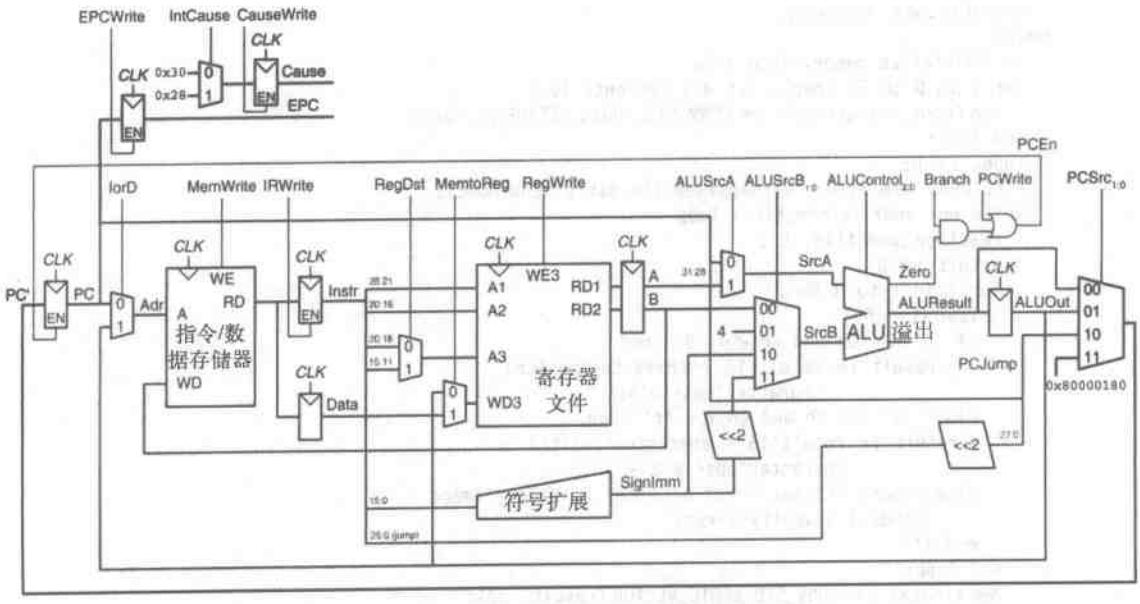


图7-62 支持溢出和未定义指令异常的数据路径

为支持mfc0指令，我们应增加一路以选择协处理器0的寄存器并将其写入到寄存器文件，如图7-63所示。mfc0指令通过$Instr_{15:11}$来指明协处理器0的寄存器号，图中仅支持了原因寄存器和EPC寄存器。我们为MemtoReg多路开关增加了另一路输入以从协处理器0中选择数值。

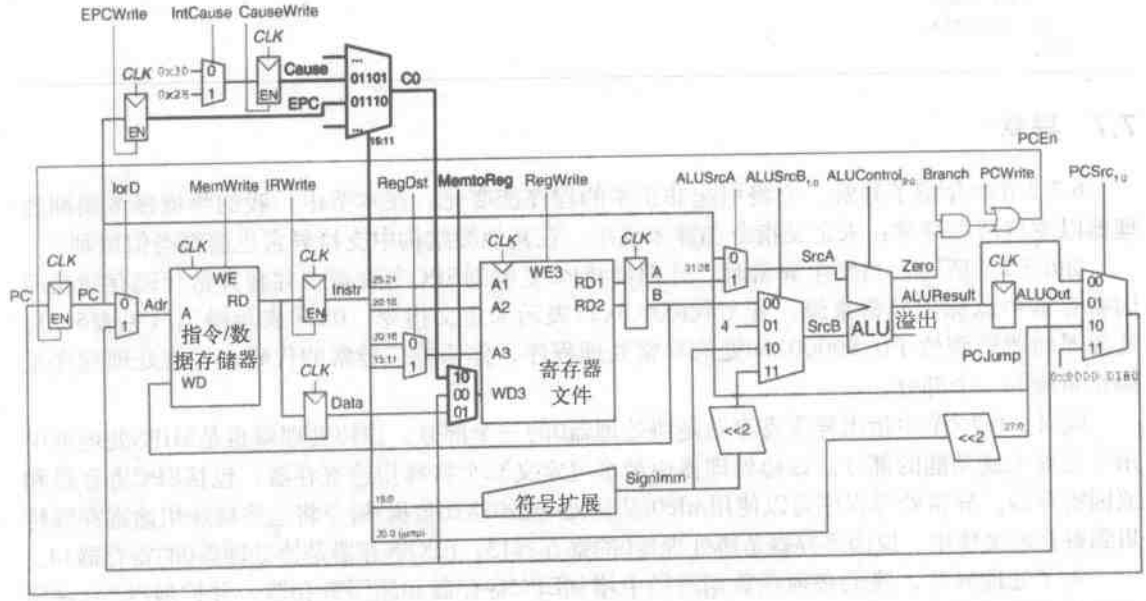


图7-63 支持mfc0指令的数据路径

修改后的控制器如图7-64所示。控制器从ALU接收溢出标志。它将产生三个新的控制信号：一个是EPC的写入使能，一个是原因寄存器的写入使能，最后一个是选择原因。同时，需要增加两个状态以支持两个异常，一个状态支持mfc0指令。

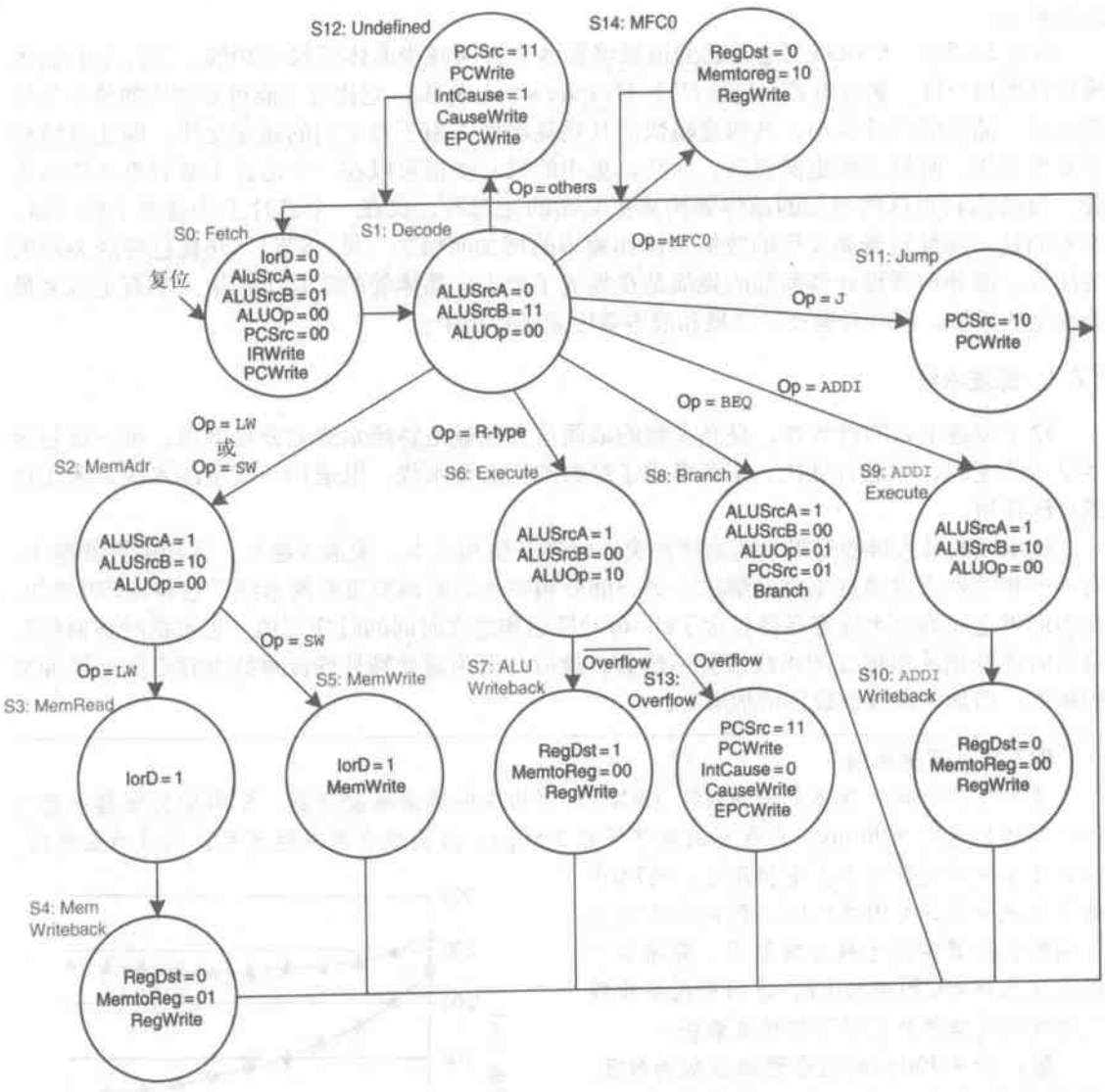


图7-64 支持异常和mfc0的控制器

如果控制器接收到一个未定义指令（不知道该如何处理的指令），它将转至S12，保存PC到EPC寄存器，向原因寄存器写入0x28。同样地，如果控制器检测到add或sub指令的算术溢出，将转至S13，保存PC到EPC寄存器，向原因寄存器写入0x30，并跳转到异常处理程序。需要注意的是：当异常发生时，将抛弃发生异常的指令，其结果也不写入寄存器文件。当mfc0指令译码时，处理器将进入S14，并选择合适的协处理器0寄存器写入到主寄存器文件中。

7.8 高级微结构*

高性能微处理器使用了多种技术来提高程序运行速度。程序运行时间正比于时钟周期和每条指令所需周期数（CPI）。因此，为了提高性能，我们应提高时钟频率，同时/或者降低CPI。本节将介绍一些已有的加速技术。因为实现细节比较复杂，我们这里仅主要介绍原理。如果读者想进一步了解细节，Hennessy和Patterson所著的《计算机体系结构》一书将提供合

适的参考。

每过2～3年，CMOS制造工艺的进展将在各个方向减少晶体管尺寸30%，芯片上的晶体管数目增加一倍。制造工艺由特征尺寸（feature size）标识，它代表了能可靠制造的最小晶体管尺寸。晶体管尺寸越小，其速度越快而且功耗越低。由于每个门的速度变快，即使微结构不发生变化，时钟主频也能提高。而且，更小的尺寸使得可以在一个芯片上放置更多的晶体管。微结构利用这些增加的晶体管构成更复杂的处理器，或在一个芯片上构造多个处理器。不幸的是，功耗随着晶体管的数据和操作频率的增加而增加（见1.8节）。功耗已经成为新的关注点。微处理器设计者面临的挑战是在集成了数十亿晶体管的芯片上构造人类有史以来最为复杂的系统，同时对速度、功耗和成本等因素进行折中。

7.8.1 深流水线

除了制造工艺的进步外，提高主频的最简单方法就是将流水线划分更多级。每一级包含尽量少的逻辑，以运行更快。本章考虑了经典的五级流水线，但是10～20级流水线目前正得到广泛使用。

流水线的最大级数受限于流水线冲突、时序开销和成本。流水线越长，其相关性就越多。有一些相关性可以通过重定向解决，另一部分相关性则必须要阻塞流水线，这将使CPI增加。在不同级之间的流水线寄存器包含了clk-to-Q延迟和建立时间的时序开销（也包括时钟偏移）。这些时序开销使得增加流水线级得不偿失。最后，因为需要额外的流水线寄存器和处理冲突的硬件，增加流水线级数将增加成本。

例7.11 深流水线

考虑将单周期处理器划分为N级（$N \geq 5$）而构成的流水线处理器。单周期处理器中整个组合逻辑的延迟为900ps。寄存器的时序开销为50ps。假设组合逻辑延迟可以划分为任意级，而且流水线冲突逻辑不会增加延迟。例7.9中的五级流水线的CPI为1.15。假设由于分支预测错误和其他流水线冲突原因，每增加一级流水线将对CPI增加0.1。请问采用多少级流水线时处理器执行程序的速度最快？

解：如果900ps的组合逻辑分级为N级，且每级用于流水线寄存器的时序开销为50ps，则周期时间为$T_c = 900/N + 50$。CPI为1.15+0.1(N-5)。每条指令时间为周期时间和CPI的乘积。图7-65画出了周期时间和指令时间与流水线级数之间的关系。指令时间在N=11时达到最小值231ps。这个最小值仅仅比六级流水线达到的250ps有少许提高。

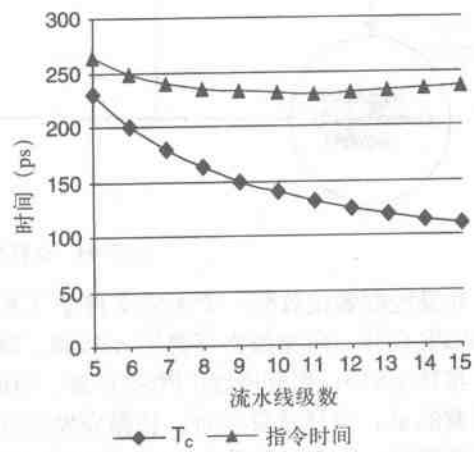


图7-65 周期时间和指令时间与流水线级数的关系

在20世纪90年代末期和21世纪初，微处理器在市场上的竞争力大多取决于时钟频率（$1/T_c$）。这使得处理器使用了非常深的流水线（在Pentium 4上有20～31级）以提高时钟频率，虽然这对整体性能提升是否有帮助仍是值得怀疑的。由于功耗正比于时钟频率，而且也随流水线寄存器的数目而增加。目前功耗变得更加重要，因此流水线深度正在减小。

7.8.2 分支预测

理想流水线处理器的CPI应为1。分支预测错误是增加CPI的主要原因。流水线越深，分支的化解在流水线中就越迟。因此，分支预测错误的开销就越大，这是因为在错误预测分支指令后提交的所有指令将被刷新。为解决这个问题，大多数流水线处理器使用分支预测器来猜测转移是否发生。在7.5.3节中，我们的流水线简单地预测所有的转移都不会发生。

有一些转移发生在程序运行到一个循环的结束部分（例如for或while语句），而且需要发生跳转以重复循环。循环往往需要执行多次，因此这些向后跳转的分支指令往往会发生。最简单的分支预测是检查分支的方向，而且预测后向分支总会发生。这称为静态分支预测（static branch prediction），因为它不依赖于程序的执行历史。

在不了解特定程序的情况下，前向分支预测往往非常困难。因此，大多数处理器采用动态分支预测器（dynamic branch predictor），使用程序运行的历史来预测分支是否发生。动态分支预测器保持了处理器最近执行的上百条（或者上千）条分支指令。这个表往往称为分支目标缓冲（branch target buffer），包含了分支目标和此分支是否发生的历史。

考虑例6.20中的循环代码以分析动态分支预测器的操作。这个循环重复10次，而且跳出循环的beq指令仅在最后一次发生。

```
add  $s1, $0, $0       # sum = 0
add  $s0, $0, $0       # i   = 0
addi $t0, $0, 10       # $t0 = 10

for:
  beq  $s0, $t0, done  # if i == 10, branch to done
  add  $s1, $s1, $s0   # sum = sum + i
  addi $s0, $s0, 1     # increment i
  j    for
done:
```

1位动态分支预测器（one-bit dynamic branch predictor）记住上次的分支转移是否发生，并预测下一次也采取同样的动作。当循环重复时，它将记住beq指令上次没有发生分支，并且预测下一次也不会发生分支。这在循环的最后一次分支前都是正确的预测，而最后一次执行分支指令是发生跳转的。不幸的是，如果循环再次执行，预测器记住了上一次分支发生了跳转。因此，它在循环重新第一次执行时将错误预测这个分支将发生。总之，一位动态分支预测器在循环的起始和结束时会错误预测。

2位动态分支预测器可以通过四个状态来解决这个问题：强跳转（strongly taken），弱跳转（weakly taken），弱不跳转（weakly not taken），强不跳转（strongly not taken），如图7-66所示。当循环重复时，它将进入“强不跳转”状态，并预测分支下一次不发生跳转。这个预测将一直正确直到最后循环的最后一次执行分支指令。此时，分支指令将发生并使得预测器转移到“弱不跳转”状态。当循环再次执行时，分支预测器预测将不发生跳转，并进入“强不跳转”状态。总之，两位动态分支预测器仅仅在循环结尾处才会预测错误。

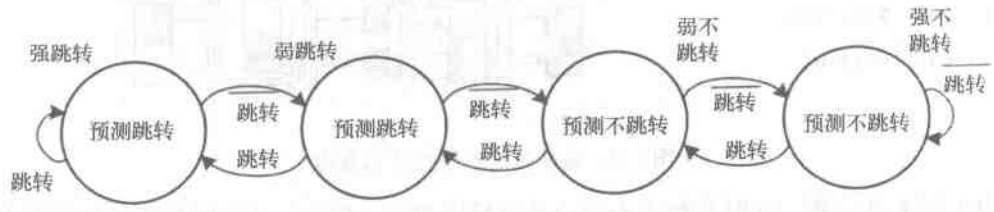


图7-66 2位分支预测器状态转移图

可以想像，分支预测器可以跟踪程序执行更多的轨迹以提高预测精度。对于典型程序，好的预测器可以达到超过90%的精度。

分支预测器运行在流水线的取指阶段，因此它可以确定下一个周期需要执行哪条指令。当预测分支将发生时，处理器从分支目标缓冲中取出分支目标指令。在分支目标缓冲中保持分支指令和无条件跳转指令的轨迹，处理器可以避免无条件跳转指令带来的流水线刷新。

7.8.3 超标量处理器

超标量处理器（superscalar processor）具有多个数据路径硬件以支持同时执行多条指令。图7-67显示了能同时取指和执行两条指令的两路超标量处理器框图。数据路径一次从指令存储器取出两条指令。寄存器文件具有6个端口，每个周期中4个端口用于读取源操作数，2个端口用于写回结果。它还包含了两个ALU和一个两端口数据存储器以同时执行两条指令。

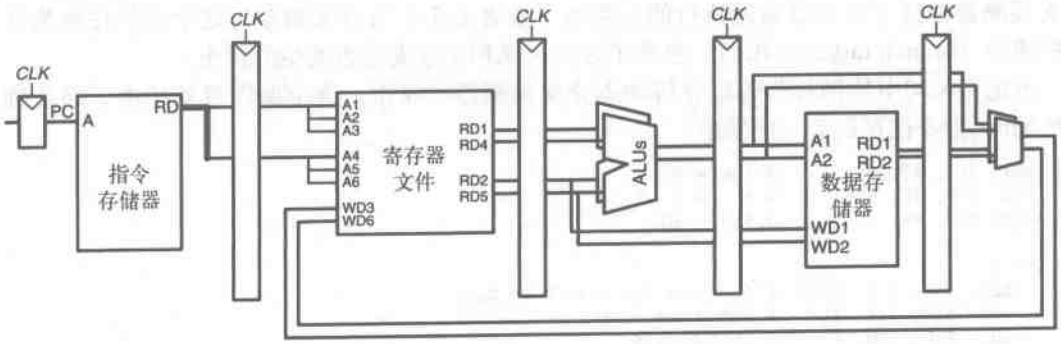


图7-67 超标量数据路径

图7-68显示了两路超标量处理器每个周期执行2条指令的流水线图。对于这个程序，处理器的CPI为0.5。设计者往往采用CPI的倒数作为每周期指令数（instructions per cycle, IPC）。此处理器对于这个程序的IPC为2。

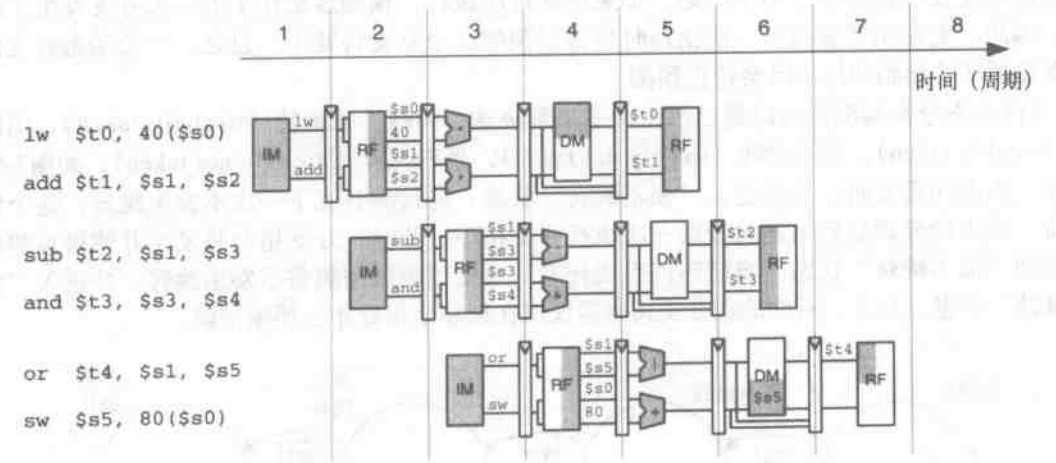


图7-68 超标量流水线操作示意图

由于相关性问题，同时执行多条指令是比较困难的。例如，图7-69显示了执行具有数据相关性的程序的流水线图。其中，add指令依赖于由lw指令产生的$t0，因此它不能与lw指令

同时提交。add指令阻塞了一个周期，使得lw指令可以在第5个周期重定向$t0的值给它。其他的相关性（包括sub和and之间针对$t0，or和sw之间针对$t3）采用重定向方式处理，即前一个周期产生的结果在后一个周期使用。如下所示的这个程序需要5个周期来提交6条指令，因此IPC为1.17。

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

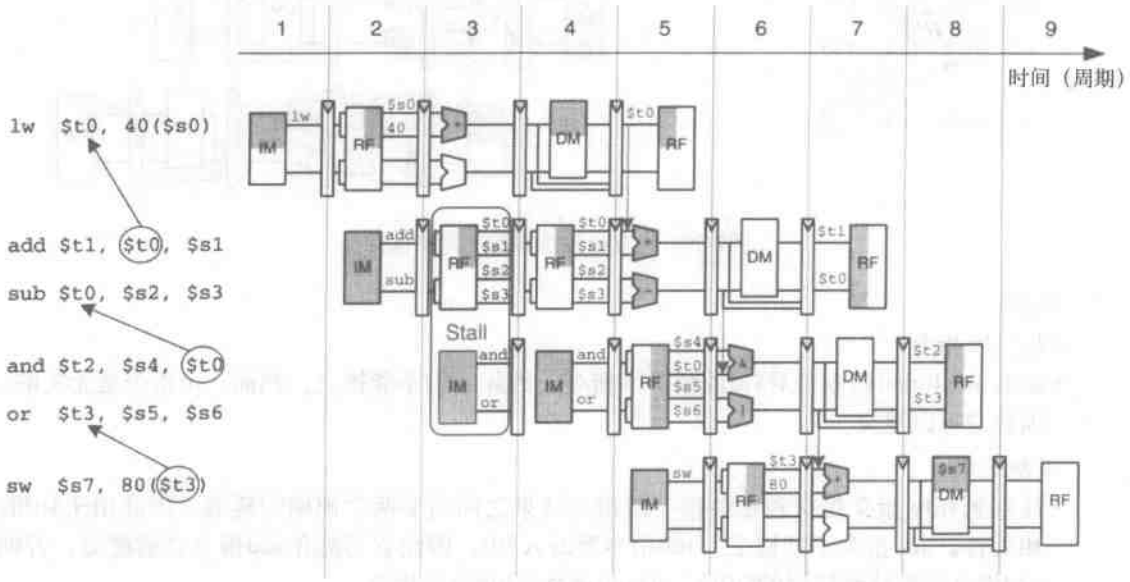


图7-69 具有数据相关性的程序

并行性有时间和空间两种形式。流水线是一种时间并行。多个执行单元是一种空间并行。超标量处理器开发了这两种并行性，使得其性能远远超越了单周期和多周期处理器。

现在商业处理器已经有三路、四路、甚至六路超标量结构。它们将分支等控制相关按照数据相关方式处理。不幸的是，真实的程序具有很多相关性，使得超标量处理器几乎无法完全利用所有的执行单元。而且，大量的执行单元和复杂的重定向网络需要大量的电路和功耗。

7.8.4 乱序处理器

为了解决相关性问题，乱序处理器（out-of-order processor）先行检查多条指令是否可以提交，并且尽可能早地开始执行不相关的指令。指令提交执行的顺序可以不同于程序员所写的顺序，同时保持相关性使得程序可以得到正确结果。

考虑在两路超标量处理器上执行图7-69的程序。处理器可以在保持相关性的前提下，每个周期中从程序中任何位置提交两条指令。图7-70中显示了数据相关性和处理器的操作。相关性可以分为读后写（RAW）和稍后将讨论的写后读（WAR）两类。提交指令的约束将在下面介绍。

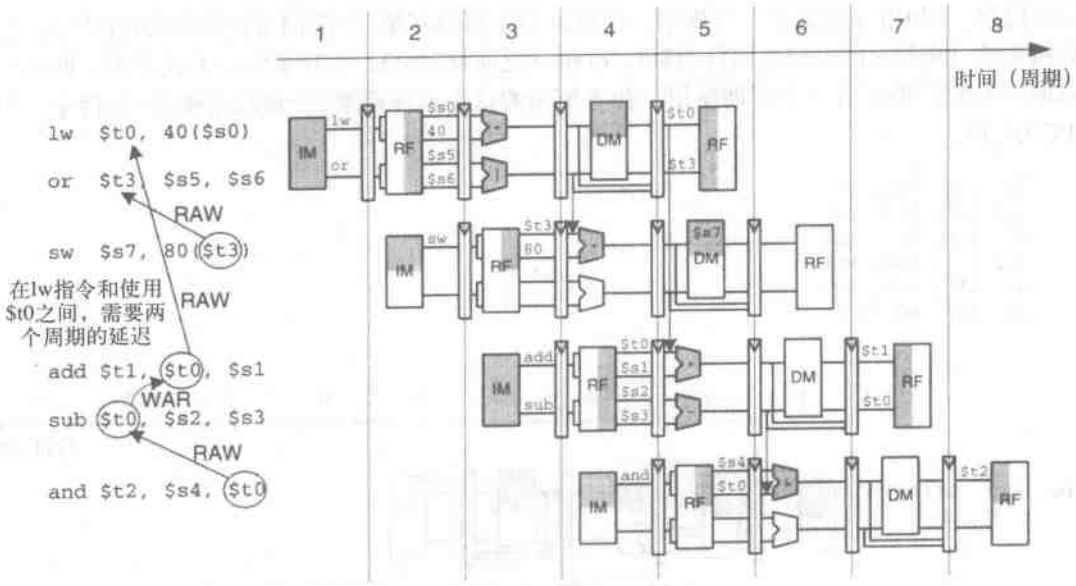


图7-70 具有相关性的程序乱序执行

周期1

- 提交lw指令
- add，sub和and指令针对$t0依赖于lw指令，因此它们不能提交。然而，or指令是无关的，因此它可以提交。

周期2

- 注意到在lw指令提交和相关指令使用其结果之间需要两个周期的延迟，因此由于$t0的相关性，add指令不能提交。sub指令要写入$t0，因此它不能在add指令之前提交，否则add指令将接收到错误的$t0值。而and指令于sub指令相关。
- 只有sw指令可以提交。

周期3

- 在第3个周期中，$t0的值可以使用，因此add指令提交，sub指令也可以同时提交，因为它在add指令使用$t0后才写入$t0。

周期4

- and指令提交，$t0从sub指令重定向到and指令。

乱序处理器在4个周期中提交了6条指令，因此IPC为1.5。

add指令和lw指令针对$t0的相关性为读后写（read after write，RAW）冲突。add指令只有在lw指令写入$t0后才能读出。这种类型的相关性我们在流水线处理器中已经了解如何处理。这种相关性从本质上限制了程序运行的速度，即使有无限多的执行单元可以利用。类似地，sw指令和or指令针对$t3，and指令和sub指令针对$t0也都是读后写相关的。

sub指令和add指令针对$t0的相关性为写后读（write after read，WAR）冲突，或称为反相关（antidependence）。sub指令必须不能在add指令读取$t0前写入，这样add指令才能根据程序的原始执行顺序获得正确的值。WAR冲突不会发生在简单的MIPS流水线处理器中，但是它们会出现在乱序处理器中，只要相关的指令（在本例中为sub指令）提前的过早。

WAR冲突并非是程序执行所不可避免的。它只不过是程序员为两条不相关的指令使用了

同一个寄存器而造成的人为冲突。如果sub指令将结果写入$t4（而非$t0），相关性就消失了，而sub指令就可以在add指令之前提交。MIPS处理器仅有32个寄存器，因此程序员不得不在所有其他寄存器都已使用的情况下重用寄存器，从而产生冲突。

第三类冲突为写后写（Write after Write, WAW）或输出相关（output dependence）没有在程序中显示出来。WAW冲突发生在一条指令试图向后续指令已经写入的寄存器进行写入。这个冲突导致错误值写入寄存器。例如，下述程序中add指令和sub指令都写入到$t0寄存器。考虑到程序的原有顺序，$t0寄存器的最后结果应来自于sub指令。如果乱序处理器先执行sub指令，WAW冲突将会发生。

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

WAW冲突也不是不可避免的；它们也是因为程序员为两条不相关的指令使用了同一个寄存器产生的。如果sub指令首先提交，则程序可以将采用抛弃add指令结果而不写入寄存器的方式化解WAW冲突。这称为去除（squashing）add指令[⊖]。

乱序处理器使用一个称为记分牌（scoreboard）的表来保持指令等待提交的次序，记录相关性的信息。这个表的大小决定了可以考虑提交的指令数目。在每个周期中，处理器检查这个表并提交尽可能多的指令，而仅仅受限于程序内在的相关性和可用的执行部件数目（例如ALU数目，存储器端口等）。

指令级并行性（instruction level parallelism, ILP）是针对特定程序和微结构上同时可以执行指令的数目。理论研究表明配合好的分支预测器和大量的执行部件，在超标量处理器上ILP可以达到很高。但是实际处理器中，即使采用了六路超标量乱序执行结构，其ILP也很难超过2或3。

7.8.5 寄存器重命名

乱序处理器使用寄存器重命名（register renaming）的方法来消除WAR冲突。寄存器重命名是在处理器中增加若干非体系结构的重命名寄存器（renaming register）。例如，MIPS处理器可以增加20个重命名寄存器$r0~$r19。因为这些寄存器不是体系结构的一个部分，所以程序员不能直接使用它们。但是处理器可以自由地使用它们来化解冲突。

例如在前节中，sub指令和add指令在重用$t0时发生了WAR冲突。乱序处理器可以将sub指令的$t0重命名为$r0。因此，由于$r0寄存器与add指令之间没有相关性，sub指令可以尽快执行。处理器中保持了一个表来表示哪些被重命名的寄存器，使得后续相关指令可以在寄存器的重命名上保持一致。在此例中，and指令也必须将$t0寄存器重命名为$r0寄存器，因为这个寄存器中包含了sub指令的结果。

图7-71显示了图7-70中相同的程序在具有寄存器重命名机制的乱序处理器上的执行过程。$t0寄存器重命名为$r0以化解RAW冲突。指令提交的约束如下所讨论。

周期1

- 提交lw指令
- add指令由于在$t0上依赖于lw指令，因此它不能提交。然而，sub指令由于其目的寄存器被重命名为$r0，所以变为不相关的指令而可以提交。

⊖ 读者可能会考虑到为什么还要提交add指令。原因在于乱序处理器必须确保产生的异常与程序在顺序处理器上运行所产生的异常完全相同。add指令可能会产生溢出异常，因此即使它的结果会被抛弃，也必须提交以检查异常是否发生。

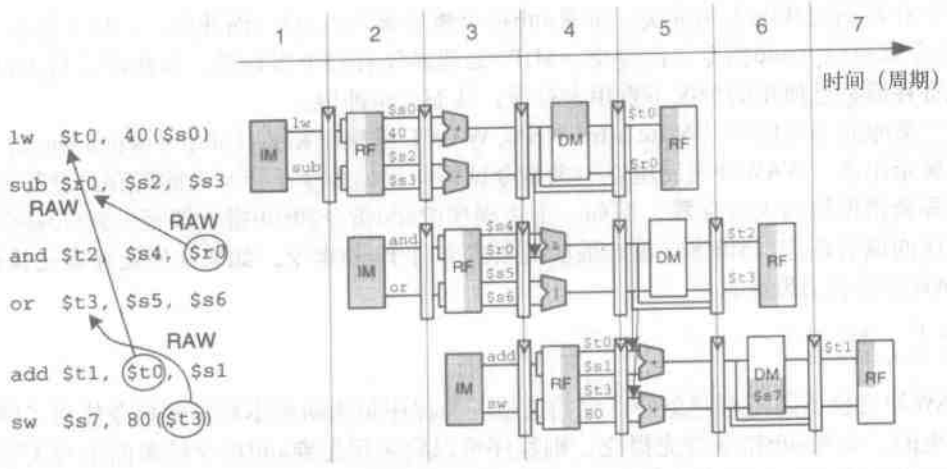


图7-71 使用寄存器重命名的乱序程序执行

周期2

- 注意到在lw指令提交和使用其结果的相关指令之间需要两个周期，因此由于$t0寄存器的相关性add指令不能提交。
- and指令依赖于sub指令，所以它可以提交，而$r0寄存器从sub指令重定向到and指令。
- or指令是不相关的，因此可以提交

周期3

- 此时，$t0可以使用，因此add指令提交。$t3也可以使用，因此sw指令提交。

带寄存器重命名的乱序处理器可以在3个周期内提交6条指令，因此其IPC为2。

7.8.6 单指令流多数据流

单指令流多数据流（single instruction multiple data, SIMD）是指同一条指令并行对多个数据片进行操作。SIMD的一个典型应用是在图像处理领域中同时执行多个比较短的算术操作，这也称为打包（packed）算术。

例如，一个32位微处理器可以将四个8位数据元素打包在一个32位字中。打包的加法和减法指令对在一个字中的四个数据元素并行操作。图7-72显示了对四对8位数据元素执行打包的8位加法而产生4个结果。一个32位字也可以分解为两个16位元素。执行打包算术需要修改ALU以消除较小数据单元之间的进位。例如，从a_0+b_0中得到的进位不应该影响a_1+b_1的结果。

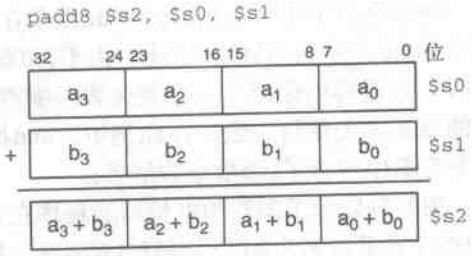


图7-72 打包算术：四个并行的8位加法

短的数据元素经常出现在图像处理中。例如，数字图像中的一个像素可能针对红、绿和蓝等每个原色分别采用8位数据存储。使用一个完整的32位字处理一个原色将浪费24位。当将四个相邻像素的原色值可以打包到一个32位字中，处理速度可以提高4倍。

SIMD指令对64位体系结构更有用处，此时可以将八个8位元素，或四个16位元素，或两个32位元素打包到一个64位字中。SIMD指令也可以用于浮点计算，例如，可以将四个32位单精度浮点值打包到一个128位字中。

7.8.7 多线程

由于实际程序的指令级并行性往往相当低，为超标量处理器或乱序处理器增加执行部件得到的回报在不断下降。另一个问题是存储器的速度远低于处理器（将在第8章讨论）。大量的读取和存储指令访问一个比较小而快的高速缓存（cache）。然而当指令或数据不在cache中时，处理器将阻塞上百个周期以从主存中读取信息。多线程技术可以在一个程序的指令级并行性比较低或因存储器访问而阻塞时，保持处理器中的多个执行单元处于有效工作状态。

为了解释多线程，我们需要引入一些新的概念。在计算机上运行的一个程序称为进程（process）。计算机可以同时运行多个进程，例如，你可以在PC上听音乐的同时上网和检查病毒。每个进程可以包含一个或多个可以同时运行的线程（thread）。例如，字处理器程序中有一个线程来处理用户的输入，另一个线程用于检查当前用户文档的拼写，第三个线程负责打印文档。此时，用户就不需要在打印结束前等待，而直接可以继续输入。

在传统的处理器中，多个线程的同时运行仅仅只是一个假相。线程在操作系统的控制下在一个处理器上依次执行。当一个线程运行结束时，操作系统将保存其体系结构状态，并加载另一个线程的体系结构状态并开始运行它。这个过程称为上下文切换（context switching）。只要处理器能足够快地切换完成所有线程，用户就会产生所有线程同时运行的假相。

多线程处理器包含了多个体系结构状态，因此可以有多个线程处于活跃状态。例如，如果我们将MIPS处理器扩展为具有4个PC和128个寄存器，就可以有四个线程同时运行。如果一个线程因为等待主存中的数据而阻塞，处理器就可以没有任何延迟地切换到另外一个线程。这是因为线程的PC和寄存器都是可以立即访问的。而且，当一个线程缺乏足够地并行性使得所有执行单元都保持工作状态，另一个线程就可以向空闲的执行单元提交指令。

多线程处理器不能提高指令级并行性，也不能提高单个线程的性能。但是，它可以提高处理器的整体吞吐率，因为执行单个线程时可能不能完全利用处理器的资源，而多个线程可以同时使用这些资源。而且多线程处理器仅仅复制了PC和寄存器文件，而没有增加执行部件或存储器，所以实现起来成本相对比较低。

7.8.8 多处理器

多处理器（multiprocessor）系统包含了多个处理器及这些处理器之间的通信机制。构造多处理器计算机系统的常用方法是对称多处理器技术（symmetric multiprocessing, SMP），其中两个或多个相同的处理器共享一个主存。

多个处理器可以分离在多个芯片上，也可以在同一个芯片上集成多个核（core）。现代处理器可以使用非常多的晶体管。使用这些晶体管来增加流水线深度或执行单元并不能显著提高性能，而且功耗很高。在2005年左右，计算机体系结构开始转向在同一个芯片上构造多个处理器，这些处理器称为核。

多处理器可以用于同时运行多个线程，或者使得特定线程运行速度更快。同时运行多个线程的方法很简单：只要将这些线程分配到多个处理器上。但是，PC的用户往往在特定时间内只需要运行少量的线程。提高特定线程运行速度更具有挑战性。程序员必须将一个线程分解成多片，以运行在多个处理器上。而且这在多个处理器需要互相通信时更加困难。如何有效利用大量的处理器核是计算机系统设计者和程序员面临的重要问题之一。

其他多处理技术包括异构多处理系统和集群。异构多处理器（asymmetric multiprocessor）使用专门处理器完成特定任务。例如，在手机中包含了一个具有特殊指令的数字信号处理器（Digital Signal Processor, DSP）以实时完成无线数据的编/解码，同时还有一个单独的通用处理器以与用户交互。管理电话号码，并支持游戏。在集群多处理（cluster multiprocessing）中，

每个处理器具有自己的局部存储器系统。集群可以将一组PC通过网络连接起来运行软件以联合解决一个比较大的问题。

7.9 现实世界透视：IA-32微结构*

6.8节中介绍的IA-32体系结构广泛应用于几乎所有的PC中。本节通过逐步提高的速度和越来越复杂的微结构来跟踪IA-32处理器的发展历程。我们在针对MIPS处理器微结构时所采用的原则也对IA-32结构有效。

Intel公司在1971年开发了第一个单芯片微处理器——4位的4004。该处理器可作为行式计算器的控制器灵活使用。它采用线宽为10μm的工艺，在12mm²的硅片上集成了2300个晶体管，运行主频为750kHz。显微镜下的芯片照片如图7-73所示。正如对4位微处理器所预期的一样，其中到处可以看到四列相似的结构。芯片的四周是引线，用于连接芯片和封装，并一直连接到电路板上。

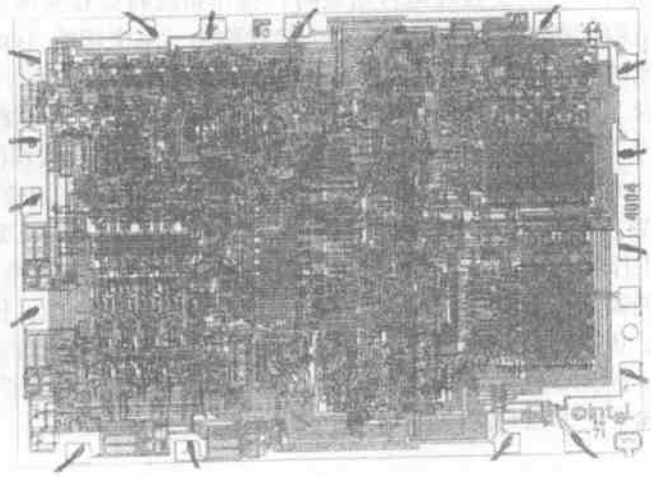


图7-73 4004微处理器芯片

4004的设计激励了8位的8008和后来的8080，并最终演化为16位的8086（1978年）和80286（1982年）。Intel在1985年推出了80386。它将8086体系结构扩展到32位，并确定了IA-32体系结构。表7-7总结了Intel主要的IA-32微处理器。从4004开始的35年内，晶体管的特征尺寸缩小了160倍，单芯片上的晶体管数目增长了5个数量级，而且工作频率增长了4个数量级。没有其他工程领域能在如此短的时间内取得这样惊人的进展。

表7-7 Intel的IA-32微处理器发展

微处理器	年份	特征尺寸	晶体管数目	主频	微结构
80386	1985	1.5～1.0	275k	16～25	多周期
80486	1989	1.0～0.6	1.2M	25～100	流水线
Pentium	1993	0.8～0.35	3.2～4.5M	60～300	超标量
Pentium Ⅱ	1997	0.35～0.25	7.5M	233～450	乱序
Pentium Ⅲ	1999	0.25～0.18	9.5M～28M	450～1400	乱序
Pentium 4	2001	0.18～0.09	42～178M	1400～3730	乱序
Pentium M	2003	0.13～0.09	77～140M	900～2130	乱序
Core Duo	2005	0.065	152M	1500～2160	双核

80386是一个多周期处理器。其主要部件标注在图7-74所示的芯片版图上。左侧的32位数据路径清晰可见。每一列处理一位数据。由微码PLA产生一些控制信号，在控制有限状态机的每个状态之间转移。右上角的存储器管理部件控制对外部存储器的访问。

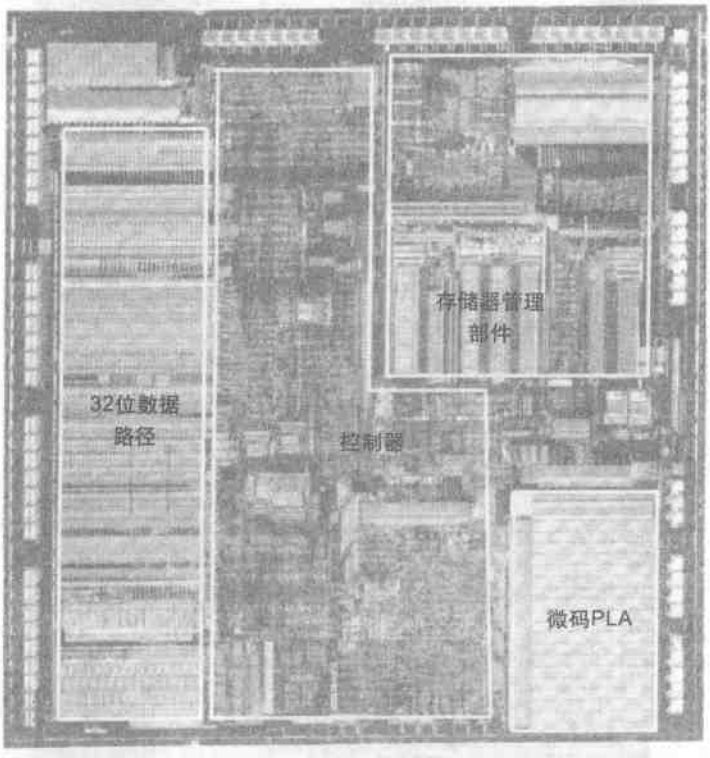


图7-74 80386微处理器芯片

图7-75中的80486利用流水线显著地提高了性能。与控制器和微码PLA在一起的数据路径依然清晰可见。80486增加了片上的浮点单元，而以前的Intel处理器将浮点指令发送到外部单独的协处理器上，或由软件仿真实现。80486的速度很快，使得外部存储器无法跟上，因此它集成了一个8KB的cache以存放最常用的指令和数据。第8章将更详细地介绍cache，并将再次研究Intel的IA-32处理器cache系统。

图7-76中的Pentium处理器是一个可以同时执行两条指令的超标量处理器。因为型号数字不能注册商标，同时AMD公司开始销售可替换的80486芯片成为了Intel公司的强劲对手，所以Intel公司不再采用80586转而使用Pentium来命名处理器。Pentium处理器使用单独的指令cache和数据cache。同时还包含了分支预测器来减少分支指令的性能损失。

Pentium Pro、Pentium II和Pentium III共享同样的乱序微结构P6。复杂的IA-32指令被分解为一条或多条类似于MIPS指令的微操作。接着，这些微操作在11流水线的快速乱序执行内核上执行。图7-77给出了Pentium III的照片。其32位数据路径称为整数执行部件（Integer Execution Unit, IEU）。浮点数据路径称为浮点单元（floating point unit, FPU）。处理器还有一个SIMD单元以执行对短整数和浮点数据的成组操作。芯片中用于乱序提交指令的面积比实际执行指令的面积还要大。指令和数据cache分别都增加到16KB。同时，Pentium III还在同一芯片上包含了一个慢速的256KB二级cache。

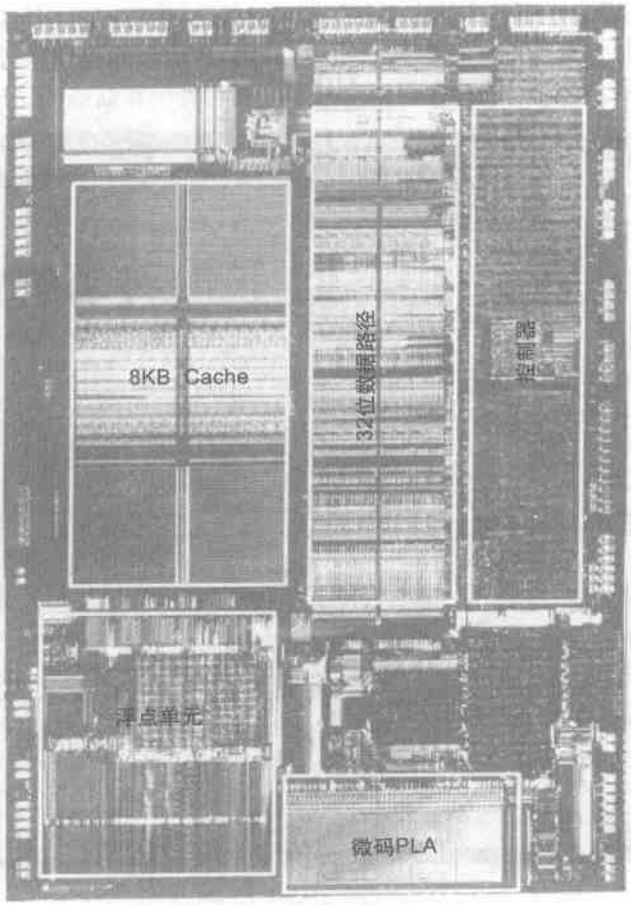


图7-75 80486微处理器芯片

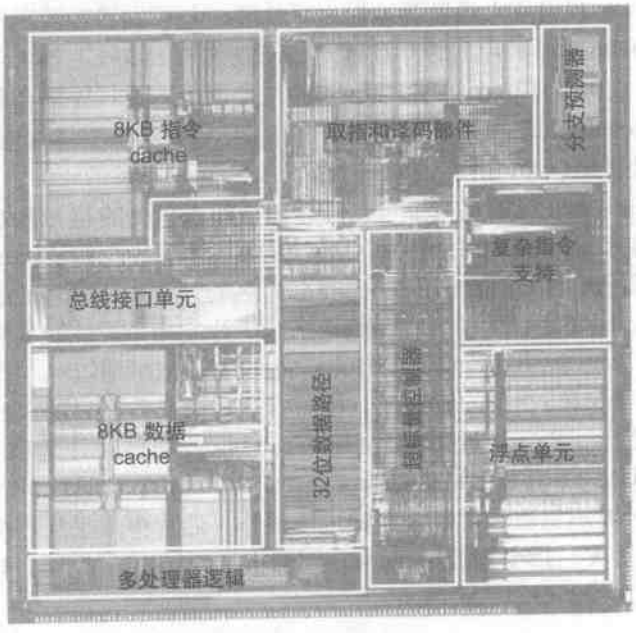


图7-76 Pentium微处理器芯片

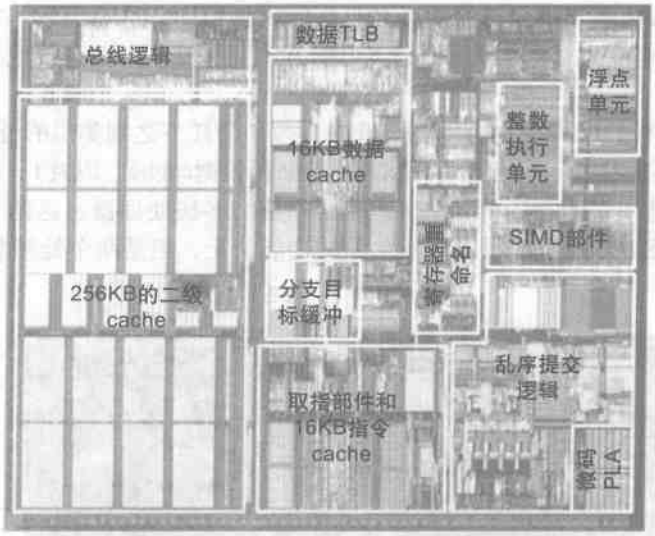


图7-77 Pentium III微处理器芯片

20世纪90年代后期，处理器市场主要在时钟速度方面竞争。Pentium 4是另一种乱序处理器，其通过非常深的流水线来达到很高的主频。它的起始版本有20级流水线，最后版本达到了31级流水线，从而主频超过了3GHz。图7-78所示的芯片包含了42到178百万个晶体管（取决于高速缓存容量），因此即使最大的执行单元也难以在照片上看到。由于指令编码复杂而且不规则，在如此高的频率下，要一个周期内完成3条IA-32指令的译码是不可能的。相反，处理器将执行预先译码位一些简单的微操作，然后将微处理器存储在称为追踪高速缓存（trace cache）的存储器中。Pentium 4的后续版本还支持了多线程以提高多个线程的吞吐量。

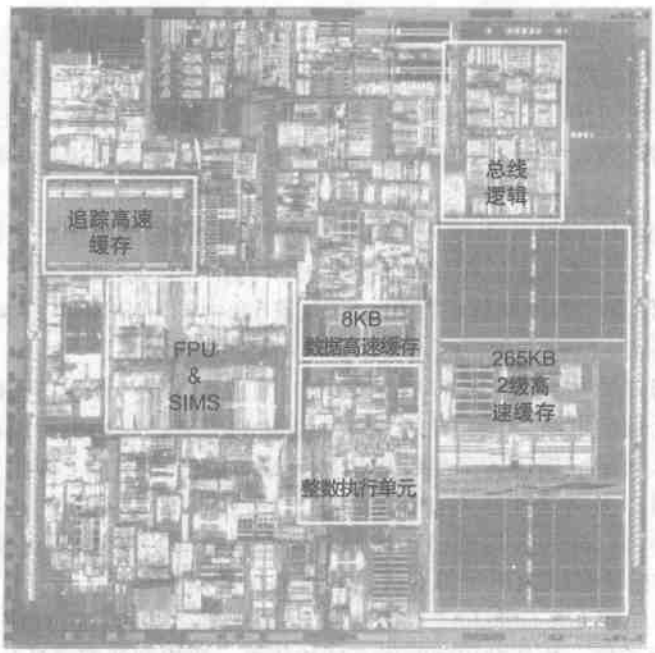


图7-78 Pentium 4微处理器芯片

Pentium 4处理器依赖于很深的流水线和很高的时钟频率，这将导致非常高的功耗。有些时候，功耗可以超过100W。这对膝上笔记本电脑是不可接受的，而且导致桌面计算机的冷却系统非常昂贵。

Intel公司发现原有的P6结构也可以在较低的主频和功耗下达到类似的性能。Pentium M使用了增强版的P6乱序微结构，并包含了32KB的指令和数据cache，以及1－2MB的二级cache。Core Dou处理器则是基于两个Pentium M处理器内核的多核处理器。这两个处理器共享一个2MB的二级cache。图7-79中已经很难看出单独的功能单元，但是两个处理器核和共享的cache则清晰可见。

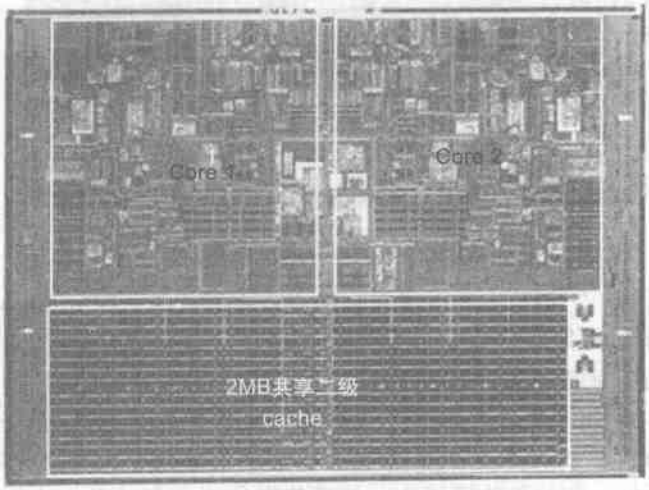


图7-79 Core Due微处理器芯片

7.10 总结

本章介绍了三种构成MIPS处理器的不同方法，每种方法在性能和成本上都各有侧重。我们发现这个问题非常不可思议：像处理器这样表面上非常复杂的设备实际上简单得可以用一张半页大小的原理图来表示？而且，其内部运行机制在外行看来像迷一样难以理解，但是实际上却非常直观。

本书中基本上对MIPS微结构中涉及的主题都画出了完整的电路图。将很多电路拼在一起构成完整的微结构，这个过程也说明了前面章节中介绍的主要原理：第2、3章中的组合电路和时序电路设计；第5章中的电路模块应用；以及第6章中的MIPS体系结构实现。MIPS微结构也可以由几页硬件描述语言表示，这需要用到第4章中的技术。

构造微结构也需要使用到管理复杂性的技术。微结构连接了数字逻辑和体系结构抽象，也构成了本书中关于数字电路设计和计算机体系结构的要点。我们也可以使用结构框图和硬件描述语言的抽象来简洁地描述各个组件的组合关系。微结构体现了规整化和模块化，重用了例如ALU、存储器、多路选择器和寄存器等通用部件。同时，微结构还在很多方面使用了层次化方法。微结构可以分为数据路径和控制单元。每个单元通过逻辑模块构造，通过前面五章描述的技术这些模块可以由逻辑门形成，并最终由晶体管构成。

本章中比较了单周期、多周期和流水线的MIPS处理器微结构。所有这三种微结构实现了MIPS指令系统的相同子集，并具有相同的体系结构状态。单周期处理器最为直观，其CPI为1。

多周期处理器使用较短的变长步骤来执行指令。同时可以重用ALU，而不需要多个加法

器。然而，它需要多个非体系结构寄存器来保存各个步骤之间的中间结果。由于所有指令执行时间不完全相同，所以多周期设计在理论上应该更快。在本例中，多周期实现要慢一些，这是因为它受到最慢步骤和各个步骤之间时序开销的限制。

流水线处理器将单周期处理器分解为5个相对较快的流水线级。它需要在各级之间加入寄存器以分割并行执行的5条指令。从名义上看，其CPI应该达到1。但是冲突导致的流水线阻塞和刷新会稍微增加CPI。冲突化解也需要花费额外硬件，并增加设计复杂性。在理想情况下，其时钟周期应该仅为单周期处理器的五分之一。实际上因为最慢流水线级和各级的时序开销，这很难达到。然而，流水线提供了潜在的性能优势。当前所有的现代微处理器都使用流水线。

虽然本章中的微结构仅仅实现了MIPS指令系统的一个子集，我们可以看到通过增强数据路径和控制器的功能可以支持更多的指令。

本章中的一个主要限制是我们假设了理想的存储器系统，它的速度足够快，而且容量足够容纳完整的程序和数据。实际上，大容量快速存储器的成本非常高。下一章将介绍如何将存放最常使用信息的小容量快速存储器与存放其余信息的大容量慢速存储器组合在一起以达到大容量快速存储器的效果。

习题

7.1 假设下述MIPS单周期处理器控制信号中的一个发生了固定0故障（stuck-at-0 fault），即无论赋任何值信号始终为0。哪些指令会产生错误？为什么？

(a) RegWrite

(b) $ALUOp_1$

(c) MemWrite

7.2 重复习题7.1，并假设信号发生固定1故障。

7.3 修改单周期MIPS处理器以实现下述指令之一。附录B中给出了指令的定义。在图7-11的复制上标注出对数据路径的修改。命名新的控制信号。在表7-8的复制上标注出对主译码器的修改。并描述其他需要修改的内容。

(a) sll

(b) lui

(c) slti

(d) blez

(e) jal

(f) lh

表7-8 用于标记修改的主译码器真值表

指令	操作码	R	R	A	B	M	M	A
R-类型	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

7.4 许多处理器由带后递增的载入（load with postincrement）指令。该指令在完成载入操作后，将变址寄存器的指针指向下一个内存字。lwinc $rt, imm($rs)等价于下述两条指令：

```
lw   $rt, imm($rs)
addi $rs, $rs, 4
```

针对lwinc指令重复习题7.3。有可能增加一条不用修改寄存器文件的指令吗？

7.5 在单周期MIPS处理器中增加一个单精度浮点单元以处理add.s、sub.s和mul.s指令。假设你已经有可以用的单精度浮点加法器和乘法器。解释对数据路径和控制器的必要修改。

7.6 你的朋友是优秀的电路设计工程师。她提出重新设计单周期MIPS处理器的一个部件，以其降低一半的延迟。使用表7-6中给出的延迟，她应该重新设计哪个部件以获得整个处理器的最大加速，改进后的周期时间应该为多少？

7.7 考虑表7-6给出的延迟，Ben通过设计一个前缀加法器将ALU的延迟减少了20ps。如果其他部件的延迟保持不变，请确定新的单周期MIPS处理器的周期时间，以及执行1000亿条指令的测试基准程序所需要的时间。

7.8 假设下述多周期MIPS处理器的控制信号中的一个发生固定0故障，即无论输入值为多少，此信号总为0。哪些指令会发生错误？为什么？

(a) MemtoReg

(b) $ALUOp_0$

(c) PCSrc

7.9 重复习题7.8，并假设这些信号发生固定1故障。

7.10 修改7.6.1节中给出的单周期MIPS处理器的HDL代码，以处理习题7.3中的一条新指令。扩充7.6.3节中的测试程序以测试新的指令。

7.11 修改多周期MIPS处理器以实现下述指令中的一条。附录B给出了指令的定义。在图7-27的拷贝上标注出对数据路径的修改。命名新的控制信号。在图7-39的复制上标注出对控制器FSM的修改。描述其他所必需的修改。

(a) srlv

(b) ori

(c) xori

(d) jr

(e) bne

(f) lbu

7.12 对多周期MIPS处理器重复习题7.4。给出对多周期数据路径和控制FSM的修改。有可能增加一条不用修改寄存器文件的指令吗？

7.13 对多周期MIPS处理器重复习题7.5。

7.14 假设习题7.13中的浮点加法器和乘法器都需要2个周期完成操作。换句话说，当输入在第一个周期开始时有效，则输出在第二个周期内有效。习题7.13的答案有何修改？

7.15 你的朋友是优秀的电路设计工程师。她提出要重新设计多周期MIPS处理器中的一个部件以提高性能。使用表7-6中的延迟，她应该重新设计哪个部件以获得整个处理器的最大加速？应该达到多快？（如果快得超过了所需要的值，就会浪费你朋友的努力）改进后的处理器周期时间为多少？

7.16 对多周期处理器重复习题7.7。

7.17 假设多周期MIPS处理器部件延迟由表7-6给出。Alyssa设计了一个新的寄存器文件部件，其功耗降低了40%，但速度却慢了一倍。她应该将这个慢速但低功耗的寄存器文件应用于多周期处理器的设计中吗？

7.18 Goliath公司宣称拥有三端口寄存器文件的专利。为了避免和Goliath公司对簿公堂，Ben设计了一个新的寄存器文件，其仅仅包含一个读/写端口（就像组合指令和数据存储器一

样)。重新设计MIPS处理器的多周期数据路径和控制器以使用这个新的寄存器文件部件。

7.19 习题7.18中重新设计的多周期MIPS处理器的CPI为多少？使用例7.7中的指令比例。

7.20 在多周期MIPS处理器上运行下述程序。需要多少个周期？这个程序的CPI是多少？

```
  addi $s0, $0, 5   # result = 5

while:
  beq  $s0, $0, done# if result > 0, execute the while block
  addi $s0, $s0, -1 # while block: result = result - 1
  j    while

done:
```

7.21 针对下述程序重复习题7.21

```
  add  $s0, $0, $0   # i = 0
  add  $s1, $0, $0   # sum = 0
  addi $t0, $0, 10   # $t0 = 10

loop:
  slt  $t1, $s0, $t0 # if (i < 10), $t1 = 1, else $t1 = 0
  beq  $t1, $0, done # if $t1 == 0 (i >= 10), branch to done
  add  $s1, $s1, $s0 # sum = sum + i
  addi $s0, $s0, 1   # increment i
  j loop
done:
```

7.22 为多周期MIPS处理器写硬件描述语言代码。处理器应和下述顶层模块兼容。存储器模块用于包含指令和数据。使用7.6.3节中测试程序测试你的处理器。

```
module top(input         clk, reset,
           output [31:0] writedata, adr,
           output        memwrite);

  wire [31:0] readdata;

  // instantiate processor and memories
  mips mips(clk, reset, adr, writedata, memwrite, readdata);
  mem mem(clk, memwrite, adr, writedata, readdata);

  endmodule

module mem(input         clk, we,
           input  [31:0] a, wd,
           output [31:0] rd);

  reg [31:0] RAM[63:0];

  initial
   begin
    $readmemh("memfile.dat",RAM);
   end

assign rd = RAM[a[31:2]]; // word aligned
  always @ (posedge clk)
    if (we)
      RAM[a[31:2]] <= wd;
endmodule
```

7.23 扩展你的多周期MIPS处理器HDL代码以处理习题7.11中的一条新指令。扩展测试程序以测试你的设计。

7.24 流水线MIPS处理器执行下述指令。在第5个周期内有哪个寄存器正在写入，哪个寄存器正在读出？

```
add $s0, $t0, $t1
sub $s1, $t2, $t3
and $s2, $s0, $s1
or  $s3, $t4, $t5
slt $s4, $s2, $s3
```

7.25 在流水线MIPS处理器上执行下述指令，使用类似图7-52的流水线图，画出所必须的数据重定向和阻塞。

```
add $t0, $s0, $s1
sub $t0, $t0, $s2
lw  $t1, 60($t0)
and $t2, $t1, $t0
```

7.26 针对下述指令，重复习题7.25

```
add $t0, $s0, $s1
lw  $t1, 60($s2)
sub $t2, $t0, $s3
and $t3, $t1, $t0
```

7.27 对于流水线MIPS处理器，需要多少个周期才能提交习题7.21中的所有指令？对于这个程序的处理器CPI是多少？

7.28 解释如何扩展流水线MIPS处理器以处理addi指令。

7.29 解释如何扩展流水线MIPS处理器以支持j指令。特别需要注意在跳转指令发生时需要刷新流水线。

7.30 例7.9和例7.10指出：在流水线MIPS处理器中，在执行级发生分支转移较在译码级发生要好。给出如何修改图7-58的流水线处理器以在执行阶段发生转移。阻塞和刷新信号如何修改？重新完成习题7.9和7.10以计算新的CPI，周期时间和执行程序的总时间。

7.31 你的朋友是优秀的电路设计工程师。她提出要重新设计流水线MIPS处理器中的一个部件以提高性能。使用表7-6中的延迟和例7.10，她应该重新设计哪个部件以获得整个处理器的最大加速？应该达到多快？（如果快得超过了所需要的值，就会浪费你朋友的努力）改进后的处理器周期时间为多少？

7.32 考虑表7-6中的延迟和例7.10。假设ALU的速度提高20%。流水线MIPS处理器的周期时间应该如何变化？如果ALU要慢20%呢？

7.33 为流水线MIPS处理器写HDL代码。处理器应和HDL例子7.13中的顶层模块兼容，应支持本章中描述的所有指令（包括习题7.28和习题7.29中的addi指令和j指令）。用HDL例子7.12中的测试程序测试你的设计。

7.34 为流水线MIPS处理器设计图7-58中的冲突部件。使用HDL实现你的设计。画出综合工具根据HDL代码可能产生的硬件原理图。

7.35 不可屏蔽中断（nonmaskable interrupt）由输入引脚触发，并输入到处理器中。当此引脚有效时，当前指令应停止执行，接着处理器将设置cause寄存器为0，并发生异常。如何修改图7-63和图7-64中的多周期处理器以处理不可屏蔽中断。

面试问题

7.1 解释流水线处理器的优点。

7.2 如果增加流水线级可以使得处理器运行速度更快，为什么处理器不会有100级流水线？

7.3 描述微处理器中出现的冲突，并解释如何化解它？每种方法有何利弊？

7.4 描述超标量处理器的概念以及其利弊。

第8章 存储器系统

8.1 引言

计算机的性能依赖于处理器微结构，同时也依赖于存储器系统。第7章假想了一个可以在单时钟周期内访问的理想存储器系统。然而，这种假想只有在非常小的存储器或非常低速的处理器时才会成立。早期的处理器相对较慢，存储器能跟上其速度。但是处理器速度的增长比存储器速度要快。当前，处理器速度是DRAM存储器速度的10～100倍。对于处理器和存储器之间不断增大的速度差异，需要借助于巧妙的存储器系统来和处理器的速度匹配。这一章将研究实际的存储器系统，并考虑速度、容量、成本之间的折中。

处理器通过存储器接口（memory interface）与存储器系统相连。图8-1为多周期MIPS存储器中使用的简单存储器接口。处理器通过地址总线发送一个地址到存储器系统。对于读操作，MemWrite信号为0，存储器通过读数据总线readdata返回数据。对于写操作，MemWrite为1，处理器通过写数据总线writedata发送数据到存储器。

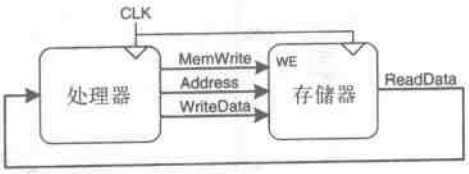


图8-1 内存接口

存储器系统设计的主要问题可以用图书馆里的书来比喻说明。图书馆的很多书都放在书架上。如果你正在写一篇以梦为主题的学期报告，你可能会去图书馆，取出弗洛伊德的《梦的解释》，然后带到工作室。在浏览之后，你会把它带回图书馆，然后取出荣格的《无意识心理学》。之后你可能因为一篇参考文献又回到图书馆查阅《梦的解释》，随之又来回查阅弗洛伊德的《自我与本我》。一个更聪明的办法是，把所有的书都保存在你的工作室以节省时间，而不是带着这些书来来回回。更进一步，当你取出一本弗洛伊德的书时，还应该在同一个书架取出他编著的其他书。

这个例子说明了6.2.1小节中所介绍的应使常见事件速度更快的原则。将那些最近使用或者最近最可能使用的书保存在工作室，可以减少了来回奔波的时间耗费。这里应用了时间局部性（temporal locality）和空间局部性（spatial locality）的原理。时间局部性意味着如果你最近使用的一本书，很可能很快就再用上它。空间局部性意味着当你使用一本书时，很可能会对同一主题的另外一些书也感兴趣。

图书馆自身也使用局部性原理使得常见事件速度更快。图书馆没有这么多的书架空间和预算来提供世界上的所有书。但是，它把一些不常用的书保存在地下室中。而且，它会与周边的图书馆建立馆际借阅约定，这样它就可以提供比其物理存储量更多的书籍。

总的来说，通过存储层次化可以对最常用的书做到大容量和快速访问。最常用书籍在你的工作室中；更多的书籍放在书架上；而其他更大一部分的可用书籍存储在地下室和其他图书馆。类似地，存储器系统使用存储器层次结构以快速访问最常用的数据，同时也有容量来存储大量的数据。

基于这种层次结构的存储子系统已经在5.5节中介绍了。计算机内存基本上由动态RAM（DRAM）和静态RAM（SRAM）组成。理想的计算机存储器系统应是快速、大容量和廉价的。

但实际上，某种特定内存只能具有这三个属性中的两个：必然会具有速度慢、容量小或者昂贵三个缺点之一。但是计算机系统可以将一个快速、小容量和廉价的存储器和一个低速、大容量和廉价的存储器组合起来以接近理想的存储器系统。快速的存储器存储最常用的数据和指令，所以平均来看，存储器系统看起来可以快速运行。大容量的存储器存储其余的数据和指令，所以总的容量很大。两个廉价存储器组合在一起比单独使用一个大容量快速存储器要便宜得多。这个原则可以扩展至整个存储器层次结构以降低成本，提高速度。

计算机主存一般由DRAM芯片组成。在2006年，一个典型PC的DRAM内存容量为256MB～1GB，其成本为100美元/1GB。在过去的30年中，DRAM的价格以每年大约30%的速度下降，存储容量以相同的速度增加。所以PC中存储器的总成本保持大致稳定。不幸的是，DRAM的速度只以每年7%的速度增长，然而处理器的性能则以每年30%～50%的速度增长，如图8-2所示。图中以1980年的存储器和处理器速度作为基线。在1980年，处理器和存储器的速度是一样的。但是性能却从此开始有了差别，存储器速度开始严重落后。

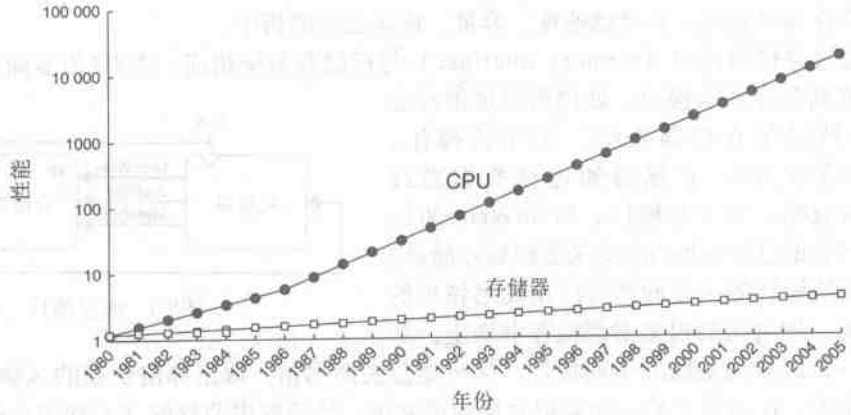


图8-2 逐渐分离的处理和存储器性能

注：得到Hennessy和Patterson所著《计算机体系结构：量化分析方法》第3版许可。

在20世纪70年代和80年代，DRAM的速度和处理器保持一致，但现在却慢得可怜。DRAM访问时间是处理器周期的一到两个数量级（前者需要几十纳秒，后者则不到1个纳秒）。

为了抵消这个趋势，计算机将最常用的指令和数据存储在更快、但较小的存储器中，这称为高速缓存（cache）。高速缓存通常置与处理器同一芯片上的SRAM中，其速度与处理器相近。这是因为SRAM比DRAM要快，而且片上存储器可以消除芯片间传输产生的延迟。2006年，片上SRAM的成本大约为10000美元/1GB，但是高速缓存容量较小（千字节到数兆字节），所以总的花费并不很高。高速缓存可以存储指令和数据，但是统称它们的内容为“数据”。

如果处理器需要的数据在高速缓存中可用，它就会被快速地返回。这称为缓存命中（hit）。否则，处理器就需要从主存（DRAM）中获得数据。这称为缓存缺失（miss）。如果大部分情况下缓存命中，那么处理器就基本上不需要等待低速的主存，平均访问时间就会比较短。

存储器层次结构的第三层是硬盘（hard disk），或称为硬盘驱动器（hard drive）。就像图书馆使用地下室储存没有放在书架上的书籍一样，计算机使用硬盘去存储不在主存的数据。在2006年，一个硬盘的价格为1美元/GB，访问时间为10ms。其价格以每年60%的速度下降，但是访问时间几乎没有提高。硬盘提供了一个比主存实际容量中更大的存储器空间，称为虚拟存储器（virtual memory）。正如地下室的书，需要花费很长的时间访问虚拟存储器中的数

据。主存，也称为物理存储器（physical memory），包含了虚拟存储器的一个子集。因此，主存可以看做硬盘中常用数据的高速缓存。

图8-3总结了后续章节讨论的计算机系统存储器层次结构。处理器首先在容量小，但是速度快的高速缓存内寻找数据。如果数据不在高速缓冲中，处理器会在主存中寻找。如果数据也不在主存，那么处理器就会从容量大，但是速度低的硬盘上的虚拟存储器中获取数据。图8-4说明了存储层次结构中容量和速度的权衡，列举了2006年技术水平下典型的成本和访问时间数据。其中访问时间越短，速度越快。

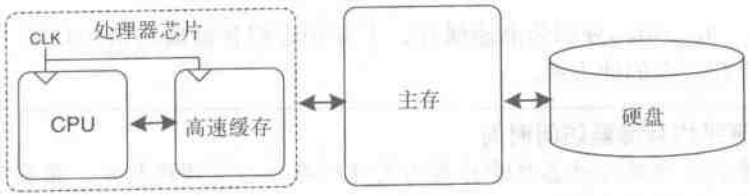


图8-3 典型的存储体系结构

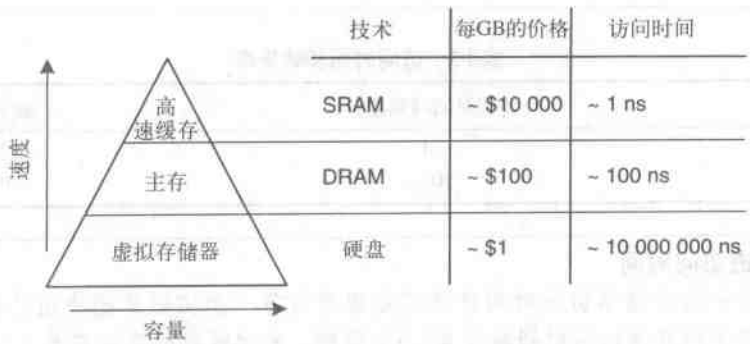


图8-4 2006年时存储层次结构中各组成部分的典型特征

8.2节分析了存储器系统的性能。8.3节讨论了几种高速缓存的组织方法。8.4节研究了虚拟存储器系统。本章最后的8.5节还将讨论存储器映射I/O，即处理器如何用与访问内存相似的方式来访问输入输出设备（如键盘和监视器）。

8.2 存储器系统性能分析

设计者（和计算机购买者）需要定量的方法衡量存储器系统的性能，以评估不同选择下成本和收益的平衡点。存储器系统性能的衡量标准为：缺失率（miss rate）或者命中率（hit rate），平均存储器访问时间（average memory access time）。缺失率和命中率计算如下：

$$\text{缺失率} = \frac{\text{存储器访问缺失的次数}}{\text{总的存储器访问次数}} = 1 - \text{命中率}$$

$$\text{命中率} = \frac{\text{存储器访问命中的次数}}{\text{总的存储器访问次数}} = 1 - \text{缺失率} \qquad (8\text{-}1)$$

例8.1 计算高速缓存性能

假设一个程序有2000条数据访问指令（读和写），其中1250条指令所需要的数据在高速缓

存中能找到，其余的750个数据由主存或者硬盘提供。高速缓存的缺失率和命中率是多少？

解：缺失率为750/2000 = 0.375 = 37.5%，命中率为1250/2000 = 0.625 = 1 − 0.375 = 62.5%

平均存储器访问时间（average memory access time，AMAT）为处理器的一条存储器访问指令执行过程中等待存储器的平均时间。图8-3的典型计算机系统中，处理器首先在高速缓存中查找数据。如果高速缓存找不到，处理器随之在主存中查找。如果主存中也缺失，处理器将访问硬盘上的虚拟存储器。因此，ATAM计算如下：

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM}) \tag{8-2}$$

其中，t_{cache}、t_{MM}和t_{VM}分别为高速缓存、主存和虚拟存储器的访问时间。MR_{cache}和MR_{MM}分别为高速缓存和主存的缺失率。

例8.2 计算平均存储器访问时间

假设某计算机系统的存储器层次结构由高速缓存和主存两级构成。根据表8-1给出的缺失率计算平均存储访问时间是多少？

解：平均存储访问时间为 1 + 0.1 (100) = 11周期。

表8-1 访问时间和缺失率

存储器层次	访问时间（周期）	缺失率
高速缓存	1	10%
主存	100	0%

例8.3 改进访问时间

11个周期的平均存储器访问时间意味着处理器对每一个实际需要使用的数据需要等待10个周期。为了将平均存储访问时间减低至1.5个周期，高速缓存缺失率应为多少？使用表8-1的数据。

解：如果缺失率为m，平均访问时间为1+100m。设置这个时间为1.5，解出需要的高速缓存缺失率m为0.5%。

值得注意的是，性能改进并不是像看起来那么好。例如，存储器系统速度提高10倍并不一定意味着计算机程序速度快10倍。如果程序指令中的50%为存储器访问指令，提高10倍速度的存储器系统只意味着能提高1.82倍的程序性能。这个通用原则称为Amdahl定律（Amdahl's law），即只在子系统的性能影响占全部性能中大部分的时候，提高子系统性能的努力才会有效果。

8.3 高速缓存

高速缓存中存放着常用的存储器数据，其存放数据字的量为容量（capacity）C。因为高速缓存的容量比主存要小，所以计算机系统设计者必须选择主存的子集存放在高速缓存中。

当处理器尝试访问数据，它首先检查高速缓存的数据。如果高速缓存命中，那么数据马上就可被使用。如果高速缓存缺失，处理器就会从主访问出数据，然后把它放在高速缓存以便以后使用。为了放置新的数据，高速缓存必须替换（replace）旧数据。本节将研究高速缓存设计中的以下问题：①在高速缓存中存放哪些数据？②如何在高速缓冲中寻找数据？③当高速缓存满时，如何替换旧数据以放置新数据？

阅读后续章节时，要记住解决这些问题的驱动力在于大部分应用中数据访问存在的固有的时间和空间局部性。高速缓存使用时间和空间局部性预测下一步需要的数据是什么。如果程序以随机方式访问数据，它将不会在高速缓存的策略中得益。

我们将在后续章节中，以容量C，组数S，块容量b，块数B和相联度N来刻画高速缓存。

尽管我们这里主要关注数据高速缓存的读出操作，但是对指令高速缓存也适用于同样原则。数据高速缓存的写入操作也与之相似，并将在8.3.4小节讨论。

8.3.1 高速缓存中存放的数据

理想化的高速缓存应能提早预知所有处理器需要的数据，并提前从主存访问得到，所以理想的高速缓存缺失率为零。因为不可能精确地预计将来所需的数据，所以高速缓存必须基于过去存储器访问的方式来猜测将来需要什么数据。特别地，高速缓存利用时间和空间的局部性以达到低缺失率。

时间局部性意味着：如果处理器最近访问过一块数据，那么它很可能再次访问这块数据。因此，当处理器读出和写入不在高速缓存中的数据时，数据需要从主存复制到高速缓存中。后续对此数据的请求将在高速缓存内命中。

空间局部性意味着：当处理器访问一块数据时，那么它很可能也访问此存储位置附近的数据。因此，当高速缓读出内存中一个字的时候，它也可以取出临近的一些字。这样的一组字称为高速缓存块（cache block）。一个高速缓存块中字的数量称为块大小b。容量为C的高速缓存包含了B=C/b个块。

实际程序的时间和空间局部性原理已经为实验所验证。如果在程序中使用了一个变量，那么同一变量就很可能被再次使用，从而产生了时间局部性。如果使用了一个数组的元素，那么同数组的其他元素也很可能被使用，从而产生了空间局部性。

8.3.2 高速缓存中的数据查找

一个高速缓存可以组织成S个组，其中每一组有一个或者多个数据块。主存中数据的地址和高速缓存中数据的位置之间的关系称为映射（mapping）。每一个内存地址可以准确地映射到高速缓存中的一个组。地址中的一些位用于确定哪个高速缓存组包含了数据。如果一个组包含了多个块，那么数据可能包含在组中的任何一块内。

高速缓存按照组中块的数目进行分类。在直接映射（direct mapped）高速缓存内，每一组包含一个块，所以高速缓存包含了S=B个组。因此，一个主存地址映射到高速缓存唯一的块。在N路组相联（N-way associative）高速缓存中，每一组包含N个块。地址依然映射到唯一的组，其中共有S=B/N个组。但是这个地址对应的数据可以映射到组中的任何块。全相联（full associative）高速缓存只有唯一一个组（S=1）。数据可以映射到组内B个块中的任何一块。因此，全相联高速缓存也是B路组相联高速缓存的别名。

为了说明高速缓存的组织方式，我们将考虑32位地址和32位字的MIPS存储器系统。内存按照字节编址，每个字有四个字节，所以内存包含了2^{30}个字，并按照字方式对齐。为了简化起见，我们将首先分析容量C为8个字，块大小b为1个字的高速缓存，之后推广到更大的块。

直接映射高速缓存

直接映射高速缓存的每个组内有一个块，所以其组数S等于块数B。要理解高速缓存块上内存地址的映射，可以想像内存就像高速缓存那样映射到多个b个字大小的块。内存中第0块的地址映射到高速缓存的第0组。内存中第1块的地址映射到高速缓存的第1组，这样一直到内

存中第B-1块的地址映射到高速缓存的第B-1组。此时高速缓存没有更多的块了，所以就开始循环，内存的第B块映射到高速缓存的第0组。

图8-5中用容量为8个字，块大小为1个字的直接映射高速缓存说明了这个映射。高速缓存中有8组，每一组有1块。因为地址是字对齐的，所以最低两位地址恒为00。紧接着的$\log_2 8=3$位说明了存储器地址映射到哪一个组。因此，地址0x00000004, 0x00000024, ..., 0xFFFFFFE4的数据全部映射到第1组。相似地，地址0x00000010, ..., 0xFFFFFFF0的数据全部映射到第4组，如此类推。每一个主存地址都可以映射到唯一的一个高速缓存组上。

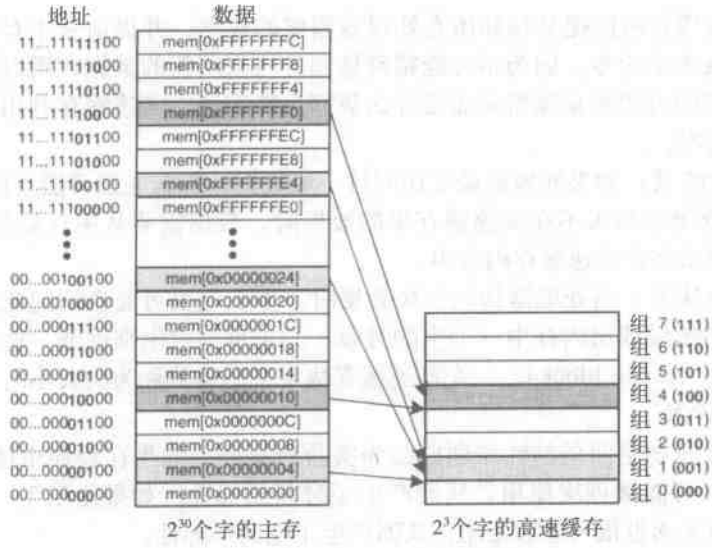


图8-5 将主存映射到直接映射高速缓存

例8.4 高速缓存字段

对于图8-5，地址0x00000014的字映射到哪一个高速缓存组？给出另一个映射到相同组上的地址。

解：因为地址是字对齐的，所以地址的最低两位为00。相邻三位为010，所以这个字映射到第5组。地址0x34，0x54，0x74，...,0xFFFFFFF4上的字都映射到这一组。

因为很多地址都映射到同一组上，高速缓存还必须保持实际包含在每一组内数据的地址。地址的最低有效位说明了哪一组包含数据。剩下的高位称为标志（tag），说明了包含在组内的数据是众多可能地址中的哪一个。

图8-6 当地址0xFFFFFFE4映射到图8-5高速缓存时相应的高速缓存字段

在我们先前的例子中，32位地址中的最低两位称为字节偏移（byte offset），说明了字节在字中的位置。紧接着的3位称为组位（set bits），说明了地址映射到哪一组（一般来说，组位的位数为$\log_2 S$）。剩下的27位标志说明了存储在特定高速缓存组中数据的存储器地址。图8-6给出了地址0xFFFFFFE4的高速缓存字段。它映射到第一组，且所有标志都为1。

例8.5 高速缓存字段

为具有1024（2^{10}）个组和块大小为1个字的直接映射高速缓存确定组位和标志位的数目。

其中地址长度为32位。

解：一个有2^{10}组的高速缓存的组位长度为$\log_2 2^{10} = 10$位。地址中的最低两位为字节偏移，剩下的32－10－2=20位作为标志。

有时（例如计算机刚启动时），高速缓存组没有包含任何数据。高速缓存的每一组都有一个有效位（valid bit），以说明此组是否包含有意义的数据。如果有效位为0，那么其内容就没有意义。

图8-7为图8-5中直接映射高速缓存的硬件结构。高速缓存由8个表项的SRAM组成。每一个表项（组）包含了一个32位的数据缓存行、27位的标志和1位有效位。高速缓存使用32位地址访问。最低两位的字节偏移位因为字对齐而省略，紧接着的3位（组位）指明了高速缓存中的表项号或组号。读指令从高速缓存中读出特定的表项，检查标志和有效位。如果标志与地址中的最高27位相同，而且有效位为1，则高速缓存命中，数据将返回到处理器。否则，高速缓存发生缺失，存储器系统必须从主存中取出数据。

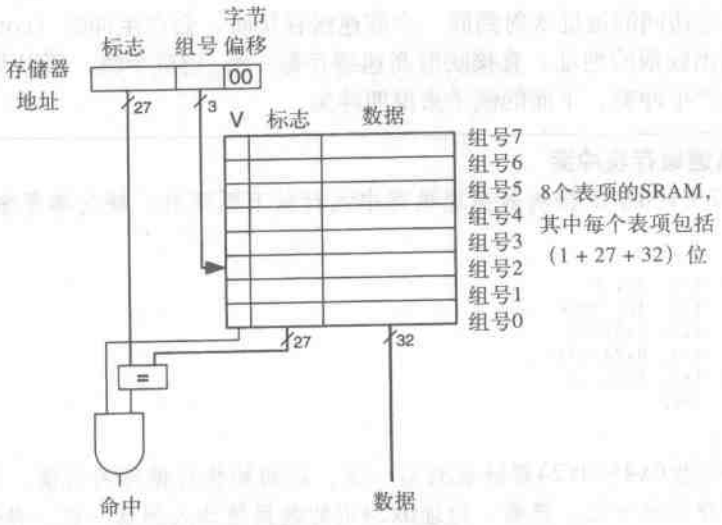


图8-7 8组的直接映射高速缓存

例8.6 直接映射高速缓存的时间局部性

实际应用中，循环是时间和空间局部性的常见来源。使用图8-7中8表项的高速缓存，给出在执行以下MIPS汇编代码循环后高速缓存中的内容。假设高速缓存开始状态为空。缺失率为多少？

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

解：这个程序包含一个重复5遍的循环。每一次循环涉及3次的内存访问（读取），最后总计产生15次内存访问。第一次循环执行的时候，高速缓存为空。必须分别从主存的0x4，0xC，0x8中获取数据，存放到高速缓冲的第1组、第3组和第2组。然后，以后四次的循环执行，数

据在高速缓存中都能被找到。图8-8所示为最后对内存地址0x4请求时的高速缓存内容。因为高27位的地址为0，所以标志全为0。缺失率为3/15=20%。

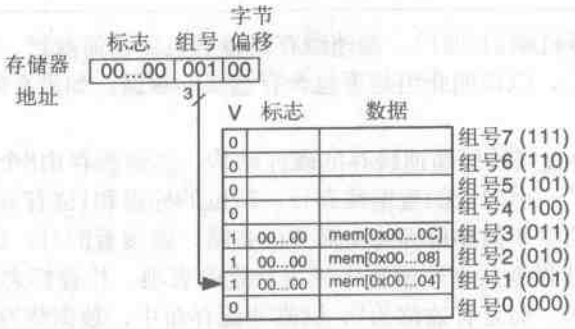


图8-8 直接映射高速缓存的内容

当两个最近访问的地址映射到同一个高速缓存块时，会产生冲突（conflict）。最近访问的地址从块中逐出较前的地址。直接映射高速缓存每一组只有1个块，所以两个映射到同一个组的地址常常会产生冲突。下面的例子来说明冲突。

例8.7 高速缓存块冲突

当在图8-7中的8字直接映射高速缓存中执行以下循环时，缺失率是多少？假设高速缓存初始为空。

```
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

解：内存地址0x4和0x24都映射到第一组。当初始执行循环的时候，地址0x4中的数据被读入到高速缓存的第一组。接着，地址0x24中的数据被读入到第一组，并逐出地址0x4中的数据。在下一循环执行时，这种访问模式会重复发生。高速缓存必须重新获取地址0x4中的数据，逐出地址0x24的数据。这两个地址产生冲突，缺失率为100%。

多路组相联高速缓存

N路组相联的高速缓存通过为每组提供N块的方式以减少冲突。每个内存地址依然映射到唯一的组中，但是它可以映射到一个组内N个块的任意一块。因此，直接映射高速缓存也可称为单路组相联高速缓存。其中，N称为高速缓存的相联度（degree of associative）。

图8-9给出了容量C为8个字，相联度N为2的两路组相联高速缓存的硬件。高速缓存现在只有4组，而不是直接映射高速缓存的8组。因此，只需要$\log_2 4=2$位组位来选择组，而不是直接映射高速缓存的3位。标志从27增加到28位。每一组包括了两路（相联度为2）。每一路由数据块、有效位和标志位组成。高速缓存从选定的组中读取所有两路中的块，检查标志和有效位以确定是否命中。如果其中一路命中，多路选择器从此路选择数据。

与同容量的直接映射高速缓存相比，组相联高速缓存的缺失率一般会较低。这是因为它们的冲突更少。然而，因为输出的多路选择器和额外的比较器，组相联高速缓存常常比较慢，成本也比较高。并且还会产生另外一个问题：当两路都满时，选择哪一路进行替换？这个问

题将在8.3.3小节中讨论。大部分的商业系统都使用组相联高速缓存。

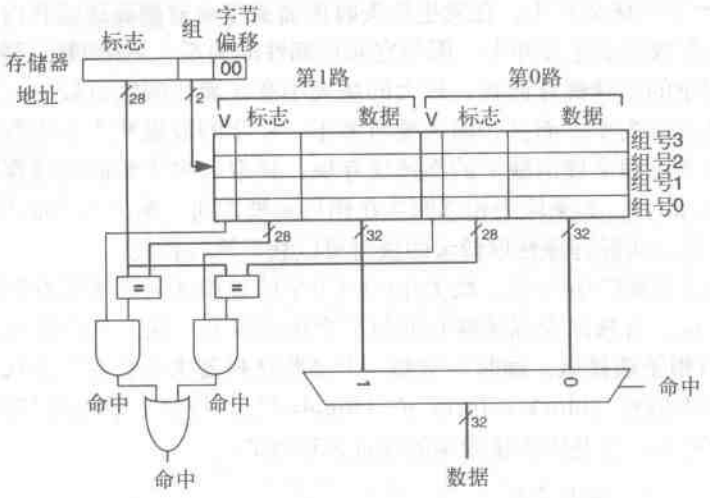


图8-9 两路组相联高速缓存

例8.8 组相联高速缓存的缺失率

重复例8.7的问题，使用图8-9中8字2路组相联高速缓存。

解： 两个对0x4和0x24地址的存储器访问都映射到第一组。然而，高速缓存中有两路，所以它能同时为两个地址提供数据空间。在第一次循环中，空的高速缓存对两个地址访问都产生缺失，然后读入两个字的数据存放到第1组的两路中，如图8-10所示。随后的4次循环中，高速缓存都命中。因此，缺失率为2/10=20%。对比例8.7中同容量大小的直接映射高速缓存缺失率为100%。

全相联高速缓存

全相联高速缓存只有一个组，其中包含了B路（B为块的数目）。存储器地址可以映射到这些路中的任何一个块。全相联高速缓存也可以称为B路单组组相联高速缓存。

图8-10 两路组相联高速缓存内容

图8-11为包含了8个块的全相联高速缓存SRAM阵列。对于一个数据请求，由于数据可能在任何一块中，所以必须对8个标志进行比较（图中没有表示出）。相似地，当一路命中时，需要使用8：1多路选择器选择合适的数据。在相同容量下，全相联高速缓存一般具有最小的冲突缺失，但是需要更多的硬件用于标志比较。因为需要大量的比较器，它们仅仅适合于较小的高速缓存。

图8-11 8块全相联高速缓存

块大小

前面的例子中块大小仅为一字，因此只能利用时间局部性。为了利用空间局部性，高速

缓存需要使用更大的块来保存若干连续的字。

块大小大于1个字的优势在于，在发生缺失时所需要字会取到高速缓存内，同时在块内临近的字也会从存储器取到高速缓冲中。因为空间局部性的关系，后续的访问就很可能会命中。然而，对于容量固定的高速缓存而言，较大的块大小意味着块的数目较少。这可能会导致更多的冲突，从而增加缺失率。而且，因为要将多于一个字的数据从主存中取出，所以在一次缺失时需要耗费更多时间来读取缺失的高速缓存块。读取缺失块到高速缓存所需的时间称为缺失代价（miss penalty）。如果块中相邻的字在稍后未被访问，那么用于取出它们的工作就会被浪费。然而，大部分实际程序在取较大的块时可以获得更好性能。

图8-12中给出了容量C为8个字，块大小b为4个字的直接映射高速缓存硬件。此时，高速缓存只有B=C/b=2块。直接映射高速缓存的每一个组里仅有一块，所以这个高速缓存中两个组，只需要$\log_2 2=1$用于选择组。同时，需要一个多路选择器来选择在一个块里的字。多路选择器由地址中的块内偏移（block offset）位（$\log_2 4=2$位）控制。最高的27位地址组成标志。整个块只需要一个标志，这是因为块内字的地址是连续的。

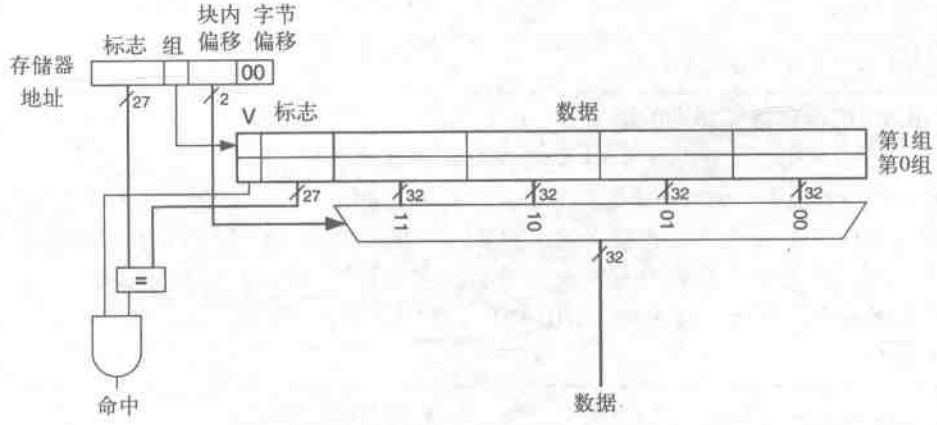


图8-12 组数为2，块大小为4个字的直接映射高速缓存

图8-13给出了地址0x8000009C映射于此直接映射高速缓存时的cache字段。对于字访问时，字节偏移总是0。后续$\log_2 b=2$位的块内偏移位指明了此字在块中的位置。后续一位指出组号。剩下的27位为标志位。因此，地址为0x8000009C的字映射到此高速缓存中第1组的第3个字。

图8-13 映射到图8-12的高速缓存时，对于地址0x8000009C的cache字段

例8.9 直接映射高速缓存中的空间局部性

其中采用容量为8个字、块大小为4个字的直接映射高速缓存重复例8.6。

解：图8-14给出了第一次存储器访问后高速缓存的内容。在第一次循环时，高速缓存在访问存储器地址0x4时产生缺失。这次访问将读取从地址0x0～0xC的数据到高速缓存的块中。所有的后续访问（如地址0xC所示）都将在高速缓存内命中。因此，缺失率为1/15=6.67%。

小结

高速缓存组织为二维阵列的结构。其中行称为组，列称为路。阵列中每个表项包括一个数据块、相应的有效位和标志位。高速缓存的关键参数为：

- 容量C；
- 块大小 b（以及块数量 B=C/b）；
- 一组内块的数量（N）。

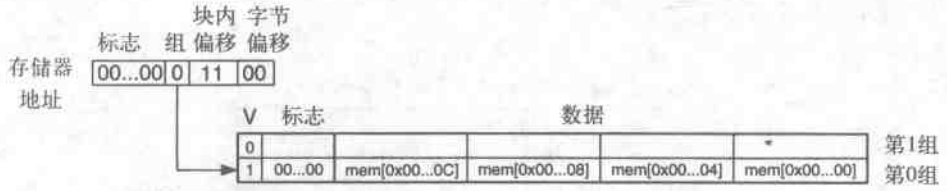


图8-14 块大小为4字的高速缓存内容

表8-2总结不同类型的高速缓存组织方式。存储器中的每个地址映射到唯一一组，但是它可以存放在此组的任何一路中。

表8-2 高速缓存的组织方式

组织方式	组数（N）	路数（S）
直接映射	1	B
组相联	1<N<B	B/N
全相联	B	1

高速缓存的容量、相联度，组大小和块大小一般都为2的整数次幂。这使得高速缓存字段（标志，组号和块内偏移）均为地址位的子集。

增加相联度N可以减少因为冲突引起的缺失。但是高相联度需要更多的标志比较器。增加块的大小b，会从空间局部性获益而减少缺失率。然而，对于固定容量的高速缓存这将减少组数，可能导致更多的冲突。同时，它也会增加了缺失代价。

8.3.3 数据的替换

在直接映射高速缓存中，每个地址映射到唯一的块和组上。如果一个组满的时候，而又必须读取新的数据，那么组里的块就会被替换为新数据。在组相联和全相联的高速缓存中，高速缓存必须在组满时选择哪一个块被替换。时间局部性原则建议最好选择最近最少使用的块，因为它看起来最近最不可能再次用到。因此，大部分相联高速缓存采用最近最少使用（least recently used, LRU）的替换原则。

在两路组相联高速缓存内，用1位使用位（use bit）U，说明组内的哪一路是最近最少使用的。每次使用其中一路，就修改U位来指示另一路为最近最少使用。对于多于两路的组相联高速缓存，跟踪最近最少使用的路将更为复杂。为了简化问题，组里的多路分成两部分（group），U指示那一部分为最近最少使用的。替换时，就从最近最少使用部分里面随机选择一块用于替换。这样的策略称为伪-LRU策略，也易于实现。

例8.10 最近最少使用替换

写出下述执行代码后，容量为8个字的两路组相联高速缓存的内容。假设采用最近最少使用替换策略，块大小为1个字，初始时高速缓存为空。

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

解：开始的两条指令从存储器地址0x4和0x24中读取数据到高速缓存的第1组中，如图8-15(a) 所示。U=0说明在第0路的数据是最近最少使用的。下一次存储器访问的地址为0x54，依然映射到组1，这将替换最近最少使用的第0路数据。如图8-15(b) 所示。随后使用位U设置为1，说明第1路的数据是最近最少使用的。

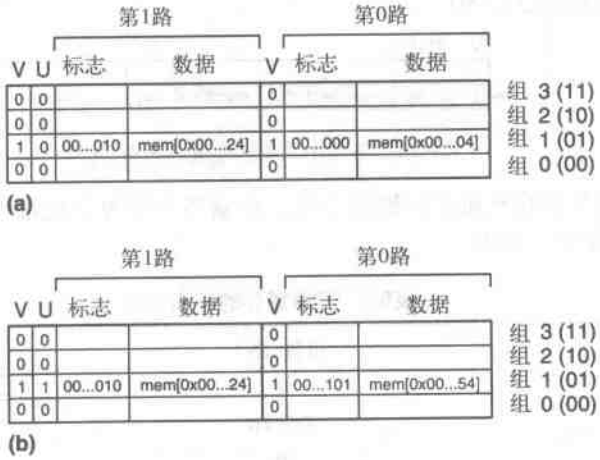


图8-15 LRU替换的两路相联高速缓存

8.3.4 高级高速缓存设计*

现代微处理器系统使用多层高速缓存以减少内存访问时间。本节将讨论两层高速缓存系统的性能，研究块大小、相联度和高速缓存容量对缺失率的影响。本节还介绍高速缓存如何使用直写或回写策略控制处理存储器写入。

多层次高速缓存

大容量高速缓存的效果更好，这是因为它们更有可能保存当前需要使用的数据，因此会有更低的缺失率。然而，大容量高速缓存的速度比小容量高速缓存要低。现代处理器系统常常使用两级高速缓存，如图8-16所示。第一级（L1）高速缓存足够小以保证访问时间为1到2个处理器周期。第二级（L2）高速缓存常常也由SRAM构成，但比L1高速缓存容量更大，因此速度也更慢。处理器首先在L1高速缓存中查找数据。如果在L1高速缓存中缺失，那么处理器将从L2高速缓存中查找。如果L2高速缓存也缺失，处理器将从主存访问数据。因为从主存的访问速度实在太慢，一些现代处理器系统在存储器层次结构中增加了更多级的高速缓存。

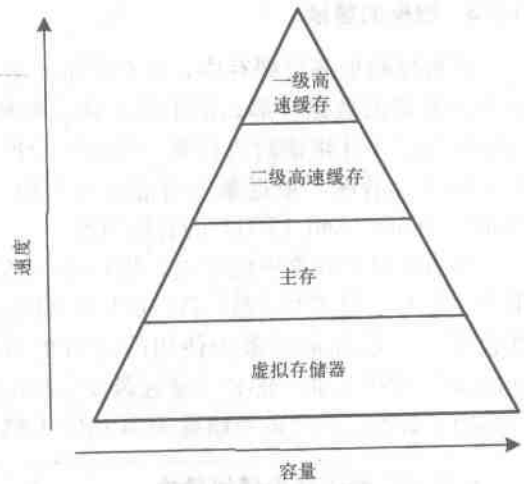


图8-16 带两级高速缓存的存储器体系结构

例8.11 带L2高速缓存的系统

使用图8-16中的系统，其中L1、L2高速缓存和主存的访问时间分别为1、10和100个周期。假设L1、L2高速缓存的缺失率分别为5%和20%。即5%的访问在L1中缺失，其中的20%在L2

中依然缺失。那么平均访问时间（AMAT）是多少？

解：每一次的内存访问都会检查L1高速缓存。当L1高速缓存缺失时（访问中的5%），处理器就检查L2高速缓存。当L2高速缓存缺失时（访问中的20%），处理器就从主存获取数据。使用等式8.2，可计算平均内存访问时间如下：1周期+0.05[10周期+0.2（100周期）]=2.5周期。

L2高速缓存的缺失率高是因为它只接收那些在L1高速缓存缺失的“硬”内存访问。如果所有的访问都直接由L2高速缓存中获得，那么L2的缺失率会是大约1%。

减少缺失率

可以通过改变容量、块大小和相联度的方式减少高速缓存的缺失率。减少缺失率的第一步是理解产生缺失的原因。缺失可以分为强制缺失、容量缺失和冲突缺失等三种原因。对于高速缓存块的第一次请求被称为强制缺失（compulsory miss），这是因为无论高速缓存任何设计，都必须先从内存读取。块因高速缓存太小以不能保存所有并发使用的数据时发生容量缺失（capacity miss）。因多个地址映射到同一组而被替换的块依然需要时发生冲突缺失（conflict miss）。

改变高速缓存的参数可以影响一种或更多的高速缓存缺失类型。例如，增加高速缓存容量可以减少冲突和容量缺失，但是不会影响强制性缺失。另一方面，增加块大小可以减少强制性缺失（因为空间局部性），但是可能增加冲突缺失（因为更多的地址可能会被映射到同一组中，这可能会冲突）。

存储器系统十分复杂，衡量它们性能的最佳方法是在不同的高速缓存配置参数下运行基准测试程序。图8-17描述了SPEC2000基准测试程序下高速缓存容量、相联度和缺失率的关系。这一基准测试程序中强制性缺失较少，以靠近x轴的黑色区域表示。正如所期望的，增加高速缓存容量可以减少容量缺失。特别对于小型高速缓存来说，增加相联性可以减少冲突缺失，如顶端的曲线所示。在四路或八路以上再增加相联性只能小幅度地减少缺失率。

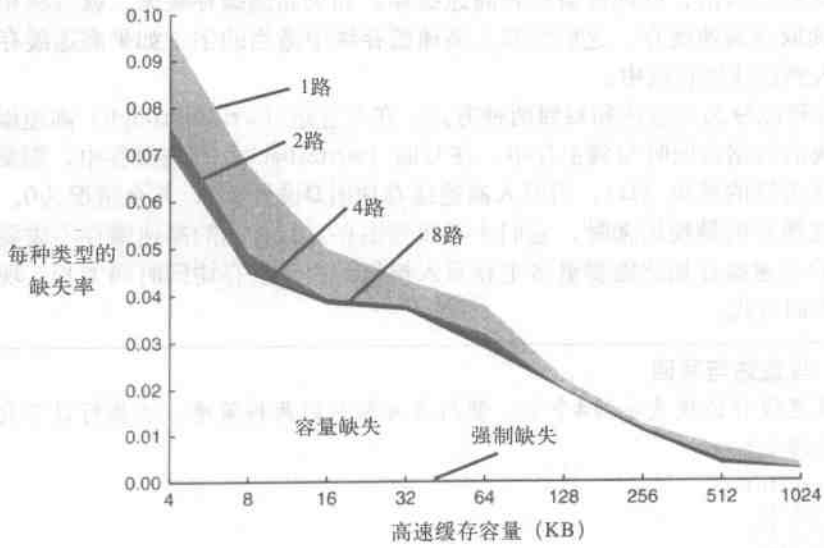


图8-17 基准测试程序SPEC2000下高速缓存容量、相联度和缺失率的关系

注：得到Hennessy和Patterson所著Computer Architecture: A Quantitative Approach，第3版，Morgan Kaufmann，2003许可。

正如前面提到的，可以用增加块大小的方法利用空间局部性，减少缺失率。但是在固定大小的高速缓存中，随着块大小的增加，组的数量将减少，从而增加冲突的可能性。图8-18描述了对不同容量下高速缓存块大小（以字节为单位）和缺失率之间的关系。对于小型高速缓存（如4KB高速缓存），在大于64字节时增加块大小会因为冲突而增加缺失率。对于大型高速缓存，增加块大小时缺失率并不改变。然而，较大的块大小可能还将增加执行时间，这是因为其缺失时从主存用于获取高速缓存块的缺失代价时间更多。

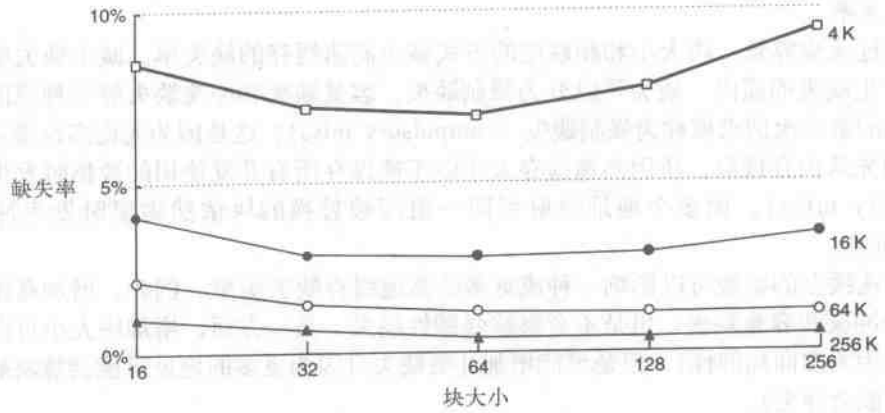


图8-18 基准测试程序SPEC2000下块大小、高速缓存大小和缺失率的关系

注：得到Hennessy和Patterson所著Computer Architecture: A Quantitative Approach，第3版，Morgan Kaufmann，2003许可。

写入策略

前面各节关注了存储器的读出操作。存储器的存储（写入）操作遵循与读出操作相似的过程。在存储器写入前，处理器会检查高速缓存。如果高速缓存缺失，就会将相应的高速缓存块从主存读取到高速缓存，之后将写入高速缓存块中适当的字。如果高速缓存命中，就简单地将字写入到高速缓存块中。

高速缓存可以分为写直达和写回两种方式。在写直达（write-through）高速缓存中，写入到高速缓存块的数据会同时写到主存中。在写回（write-back）高速缓存中，需要增加一个与各高速缓存块关联的脏位（D）。当写入高速缓存块时D设置为1，其余情况为0。只在脏高速缓存块从高速缓存内替换出来时，它们才写回到主存。写直达的高速缓存不需要脏位，但是与写回方式的高速缓存相比需要更多主存写入操作。由于主存访问时间太长，现代的高速缓存往往采用写回方式。

例8.12 写直达与写回

假设某高速缓存的块大小为4个字。使用直写和写回两种策略，在执行以下代码时主存访问次数分别为多少？

```
sw $t0, 0x0($0)
sw $t0, 0xC($0)
sw $t0, 0x8($0)
sw $t0, 0x4($0)
```

解：所有四个存储指令对同一个高速缓存块写入。在写直达高速缓存中，每一个存储指令对主存写入一个字，需要4次的主存写入。写回策略仅仅在脏高速缓存块被替换时才需要一次主存访问。

8.3.5 MIPS处理器中高速缓存的发展*

表8-3给出了1985年到2004年间MIPS处理器中高速缓存组织结构的发展情况。主要的趋势包括引入多级高速缓存结构，更大高速缓存容量，增加相联度。产生这些趋势的原因在于不断增长的CPU频率和主存速度差异，以及不断下降的晶体管成本。CPU和存储器之间不断增加的速度差异需要不断减少缺失率以克服主存瓶颈，不断下降的晶体管成本则为增加高速缓存容量提供了可能。

表8-3 MIPS处理器中高速缓存的发展*

年份	CPU	主频（MHz）	一级缓存	二级缓存
1985	R2000	16.7	无	无
1990	R3000	33	32KB 直接映射	无
1991	R4000	100	8KB 直接映射	1MB 直接映射
1995	R10000	250	32KB 两路组相联映射	4MB 两路组相联映射
2001	R14000	600	32KB 两路组相联映射	16MB 两路组相联映射
2004	R16000A	800	64KB 两路组相联映射	16MB 两路组相联映射

*来自于D. Sweetman所著See MIPS Run，Morgan Kaufmann，1999.

8.4 虚拟存储器

大部分现代计算机系统使用硬盘（也被称为硬盘驱动器）作为存储体系的最底层（如图8-4）。与理想化的大容量、快速、廉价存储器相比，硬盘容量大，价格便宜，但是速度却非常慢。硬盘比高成本效益的主存（DRAM）提供了更大容量。然而，如果大部分的存储器访问需要使用硬盘，性能将严重下降。在PC上一下运行太多程序时，就可能遇到这种情况。

图8-19所示为一个去了盖子的硬盘。顾名思义，硬盘包含了一片或者更多的坚硬盘片

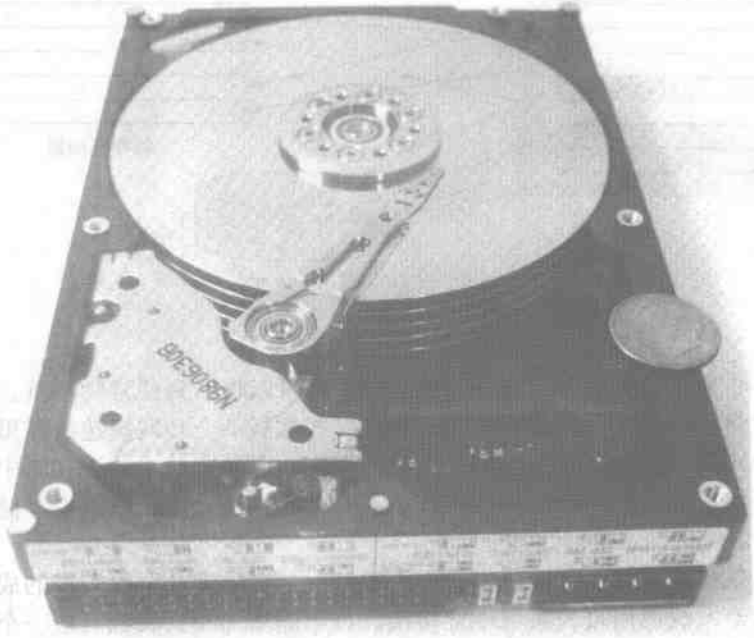


图8-19 硬盘

(platter)，每一个盘片的长三角臂末端都有一个读写头（read/write head）。首先移动读写头到盘片的正确位置，当盘片在它下面旋转时以磁方式读写。读写头需要毫秒级的时间完成盘片上的正确寻道，这对于人看来很快，但却比处理器慢百万倍。

在存储器层次结构中增加硬盘的目标是提供一个虚拟化的廉价大容量存储系统，而且在大部分存储器访问时，依然能提供较快速的存储器访问速度。例如，一个只提供128MB DRAM的计算机，可以用硬盘高效提供2GB的存储。较大的2GB存储空间被称为虚拟存储器(virtual memory)。较小的128MB主存称为物理存储器（physical memory)。本节中，我们将使用物理存储器这个术语来指主存。

程序会访问虚拟存储器内的任一数据，必须使用虚地址（virtual address）指明其在虚拟存储器内的位置。物理存储器内保存了虚拟存储器中大部分最近访问过的子集。从这点来看，物理存储器充当了虚拟存储器的高速缓存。因此，大部分访问将以DRAM的速度命中物理存储器，而程序却可以使用更大容量的虚拟存储器。

针对8.3节中讨论的高速缓存原理，虚拟存储器系统使用了不同的术语。表8-4总结了类似的术语。虚拟存储器被分为虚页（virtual page)，大小一般为4KB。物理存储器也类似地划分为物理页，大小也是4KB。一个虚页可能在物理存储器（DRAM）中，也可能在硬盘上。例如，图8-20中给出了一个大于物理存储器的虚拟存储器。长方形表示页。一些虚页当前在物理存储器中，另一些在硬盘上。根据虚地址确定物理地址的过程称为地址转换（address translation)。如果处理器试图访问不在物理存储器里的虚地址，就会产生页面失效（page fault)，由操作系统将页从硬盘读取到物理存储器中。

表8-4 高速缓存和虚拟存储器相似的术语

高速缓存	虚拟存储器
块	页
块大小	页大小
块偏移	页偏移
缺失	页面失效
标志	虚页号

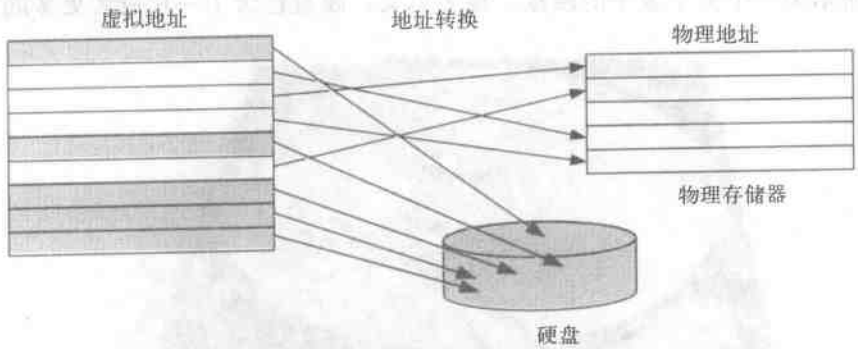


图8-20 虚页和物理页

为了防止因冲突而产生的页面失效，任意虚页都可以映射到任意物理页。换句话说，物理存储器就像针对虚拟存储器的全相联高速缓存。在常规的全相联高速缓存里，每一个高速缓存块都有一个比较器，来比较最高位地址位与标志，以确定请求是否在块中命中。在类似的虚拟存储器系统里，每一个物理页也应该需要一个比较器，来比较虚拟存储器地址的最高位和标志，以确定虚页是否映射到物理页上。

现实的虚拟存储器系统中有很多物理页，对每一个页提供一个比较器的成本很高。作为代替，虚拟存储器系统使用页表实现地址转换。每个虚页都在页表都对应一个表项，来说明它在物理存储器中的位置，或在硬盘中的位置。每次读出或者写入指令需要首先访问页表，

然后访问物理存储器。页表访问将程序使用的虚地址转换为物理地址。物理地址随后用于实际的读或者写数据。

页表常常太大而只能存放在物理存储器中。因此，每一次读出或者写入需要两次物理存储器访问：第一次是访问页表，第二次访问数据。为了加速地址转换，地址转换后备缓冲(translation lookaside buffer，TLB）缓存了最常用的页表表项。

本节的后续部分详细介绍了地址转换、页表和TLB。

8.4.1 地址转换

在包含虚拟存储器的系统中，程序使用虚地址访问大容量存储空间。计算机必须将虚地址转换以找到物理存储器中的地址，或产生一个页面缺失然后从硬盘获得数据。

前面提到虚拟存储器和物理存储器都被分成页。虚地址或者物理地址的高位分别说明了虚页或物理页的页号（page number)。低位说明指明了页内字的位置，也称为页偏移。

图8-21说明包含2GB虚拟存储器，128MB物理存储器，页大小为4KB的虚拟存储器页组织结构。MIPS处理器采用32位地址。对于2GB = 2^{31}字节虚拟存储器，仅使用虚拟存储器地址的最低31位，第32位总为0。相似地，对于128MB = 2^{27}字节的物理存储器，仅只用物理地址的最低27位，最高5位总为0。

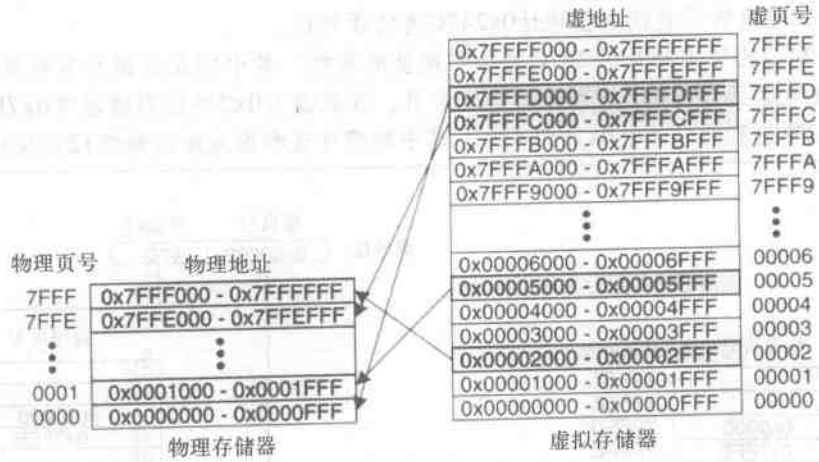


图8-21 物理页与虚页

因为页大小为4KB=2^{12}字节，所以有$2^{31}/2^{12} = 2^{19}$个虚页和有$2^{27}/2^{12} = 2^{15}$个物理页。因此，虚页和物理页的页号分别为19位和15位。物理存储器在任何时间只能最多保存1/16的虚页。其余的虚页保存在硬盘上。

图8-21表明虚页5映射到物理页1，虚页0x7FFFC映射到物理页0x7FFE等等。例如，虚地址0x53F8（虚页5内0x3F8的偏移）映射到物理地址0x13F8（物理页1内0x3F8的偏移)。虚地址和物理地址的最低12位是一样的(0x3F8)，它指明了虚页和物理页内的页偏移。从虚地址到物理地址的转换过程中，只需要转换页号。

图8-22 虚地址到物理地址的转换

图8-22说明了虚地址到物理地址间的转换。最低12位为页偏移，不需要转换。虚地址的最高19位为虚页页号（virtual page number, VPN)，可转换为15位物理页页号（physical page number, PPN)。后面两小节将进一步介绍页表以及如何使

用TLB实现地址转换。

例8.13 虚地址到物理地址的转换

用图8-21的虚拟存储器系统确定虚地址0x247C的物理地址。

解：12位页偏移（0x47C）不需要转换。虚拟存储器地址的其余19位给出了虚页页号，所以虚地址0x247C应在虚页0x2中。图8-21中，虚页0x2映射到物理页0x7FFF。因此，虚地址0x247C映射到物理地址0x7FFF47C。

8.4.2 页表

处理器使用页表（page table）完成虚地址和物理地址间的转换。前面提到页表对每一个虚页都包含了一个表项。表项中包括了物理地址页号和有效位。如果有效位是1，虚页映射到表项指定的物理页。否则，虚页在硬盘中。

因为页表比较大，需要存储在物理存储器中。假设页表存储为连续的数组，如图8-23所示。页表包含图8-21中的存储器系统映射。页表以虚页页号（VPN）作为下标。例如，第5个表项说明了虚页5映射到物理页1。第6个表项无效（V=0），所以虚页6在硬盘中。

例8.14 使用页表实现地址转换

使用图8-23给出的页表找出虚地址0x247C的物理地址。

解：图8-24给出了虚地址0x247C到物理地址的转换。其中12位页偏移不需要转换，虚地址的其余19位为虚页页号0x2，为页表建立索引。页表虚页0x2映射到物理页0x7FFF。所以，虚地址0x247C映射到物理地址0x7FFF47C。其中物理地址和虚地址的最低12位保持不变。

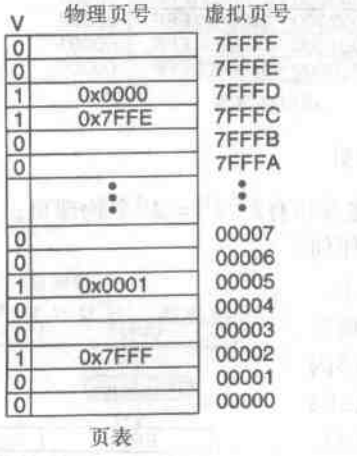


图8-23 图8-21的页表

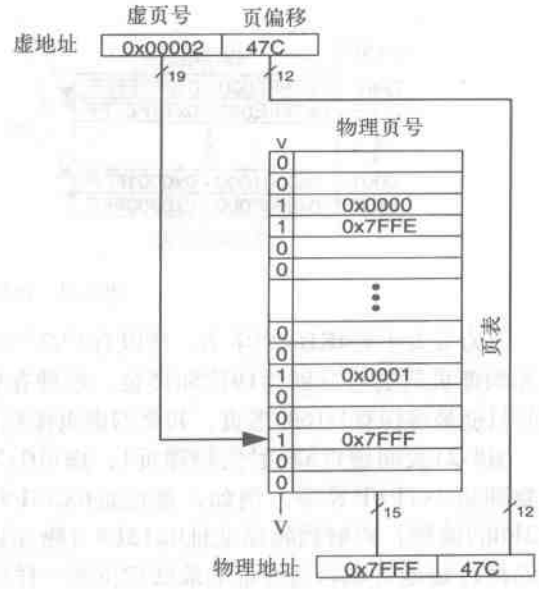


图8-24 使用页表进行地址转换

页表可以存放在物理存储器的任何位置，这由操作系统自由决定。处理器一般使用专用的页表寄存器（page table register）存放页表在物理存储器中的基址。

为了实现读出和写入操作，处理器必须首先进行虚地址到物理地址的转换，之后访问

物理地址中的数据。处理器从虚地址提取虚页页号，之后将其和页表寄存器相加，找到页表表项的物理地址。处理器随后从物理存储器读取页表表项，以获取物理页页号。如果表项有效，处理器将物理页页号和页偏移合并，生成物理地址。最后，它根据物理地址读出或者写入数据。因为页表存储在物理存储器中，所以每一次读出或者写入操作需要两次物理存储器访问。

8.4.3 地址转换后备缓冲

如果每一次的读出和写入都需要读取一次页表，将加倍读出和写入的延迟，对虚拟存储器的性能会产生严重的影响。幸运的是，页表访问有很好的时间局部性。数据访问的时间和空间局部性，以及比较大的页意味着很多连续的读出和写入操作都发生在同一页上。因此，如果处理器能记住它最后读出的页表表项，就很可能重用这个转换表项而不需要重读页表。一般来说，处理器可以将最近使用的一些页表表项保存在称为地址转换后备缓冲（TLB）的小型高速缓存内。处理器在访问物理存储器页表前首先查找TLB内的转换表项。在实际的程序中，绝大多数访问都在TLB中命中，避免了读取物理存储器中页表的时间消耗。

TLB以全相联高速缓存的方式组成，一般有16～512个表项。每一个TLB表项有一个虚页号和它相应的物理页号。TLB使用虚页页号访问。如果TLB命中，它返回相应的物理页号；否则，处理器必须从物理存储器里读页表。TLB设计得足够小使得它的访问时间可以小于一个周期。即使如此，TLB的命中率一般也大于99%。对于大多数读出和存取指令，TLB使内存访问数从2次减少为1次。

例8.15 使用TLB实现地址转换

考虑图8-21的虚拟存储器系统。使用一个2表项TLB完成虚地址0x247C和0x5FB0到物理地址的转换，或解释为什么必须访问页表。假设TLB目前包换有效的虚页0x2和0x7FFFD的转换内容。

解：图8-25为处理虚地址0x247C请求的2个表项的TLB。TLB接收传入虚拟地址的虚页号0x2，将其与每一个表项的虚页页号比较。表项0匹配而且有效，所以请求命中。将匹配表项的物理页页号0x7FFF与虚地址的页偏移拼接形成转换后的物理地址。如常一样，页偏移不需要转换。

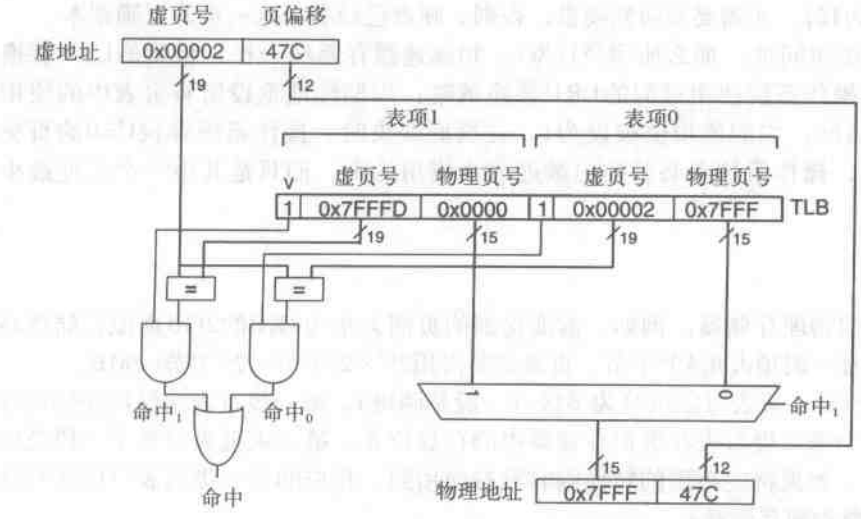


图8-25 使用2表项TLB的地址转换

对虚地址0x5FB0的请求在TLB中缺失。所以请求需要转向到页表进行转换。

8.4.4 存储器保护

到目前为止，本节都集中于如何使用虚拟存储器来提供一个快速、廉价和大容量的存储空间。使用虚拟存储器的一个同样重要原因是提供并发程序间的保护。

读者可能已经知道，现代计算机一般在同一时间运行好几个程序或者进程。所有程序在物理存储器内是并存的。在一个设计良好的计算机系统中，程序应当各自独立地保护起来，以避免某个程序使得其他程序崩溃，或破坏其他程序。更准确地说，在没有得到允许的情况下，某个程序不能访问其他程序的存储空间。这称为存储器保护（memory protection）。

虚拟存储器系统为每个程序提供独立的虚拟存储器地址空间（virtual address space）以提供存储器保护。每一个程序可以任意使用自己虚拟存储器地址空间中的存储器，但在任一时刻只有部分的虚拟存储器地址空间在物理存储器中。每一个程序可以使用它所有的虚地址空间而无需担心其他程序的物理位置。然而，一个程序只能访问已经映射到自身页表里的物理页。这样，程序就不能意外地或者恶意地访问其他程序的物理页，因为它们没有映射到程序的页表中。在某些情况下，多个程序可以访问公共的指令或者数据。操作系统会为每一个页表项加上控制位，以决定哪些程序可以写入共享的物理页。

8.4.5 替换策略*

虚拟存储器系统使用写回和近似的最近最少使用（LRU）替换策略。每一个对物理存储器的写都产生写硬盘的写直达策略是不实际的。如果采用写直达策略，写入指令将以毫秒级的硬盘速度操作，而不是纳秒级的处理器速度。在写回策略下，当物理页从物理存储器替换出来时，才写回到硬盘。把物理页写回到硬盘，然后从硬盘中读出另外的虚页称为交换（swap）。所以虚拟存储器系统里的硬盘有时称为交换空间（swap space）。处理器在发生页面缺失时换出一个最近最少使用物理页，然后把缺失的虚页替换到被换出的页。为了支持这一替换策略，每一个页表表项需要包含两个额外的状态位：脏位D和使用位U。

在物理页从硬盘读出后，若任何写入指令修改过它的话，脏位就设置为1。当物理页被换出时，只在它的脏位为1时，才需要写回到硬盘。否则，硬盘已经有了这一页的正确副本。

如果物理页最近被访问过，那么使用位U为1。和高速缓存系统一样，精确的LRU替换将会异常复杂。所以操作系统使用近似的LRU替换策略：周期性地重设所有页表中的使用位为0。当一个页被访问，它的使用位被设为1。在页面缺失时，操作系统寻找U=0的页交换物理存储器。因此，操作系统未必替换出最近最少使用的页，而只是其中一个最近最少使用页。

8.4.6 多级页表*

页表会占据大量的物理存储器。例如，前面提到的页面大小为4KB的2GB虚拟存储器将需要2^{19}个表项。如果每一表项占用4个字节，页表需要占用$2^{19} \times 2^2$字节$=2^{21}$字节=2MB。

为了节省物理存储器，页表可以被分为多级（一般是两级）。第一级页表常保持在物理存储器中。它指明了小的第二级页表在虚拟存储器中的存放位置。第二级页表包含了一段范围虚页的实际转换内容。如果特定范围的转换内容没有使用到，相应的第二级页表可以被替换到硬盘，而不需要浪费物理存储器。

在两级页表中，虚页页号分为两部分：页表号（page table number）和页表偏移（page table offset），如图8-26所示。页表号对驻留在物理存储器里的第一级页表进行寻址。第一级页表表项给出了第二级页表的基址或者在V为0时表示必须从硬盘获取。页表偏移对第二级页表进行寻址。虚拟存储器地址的余下12位为页偏移，页大小为2^{12}=4KB。

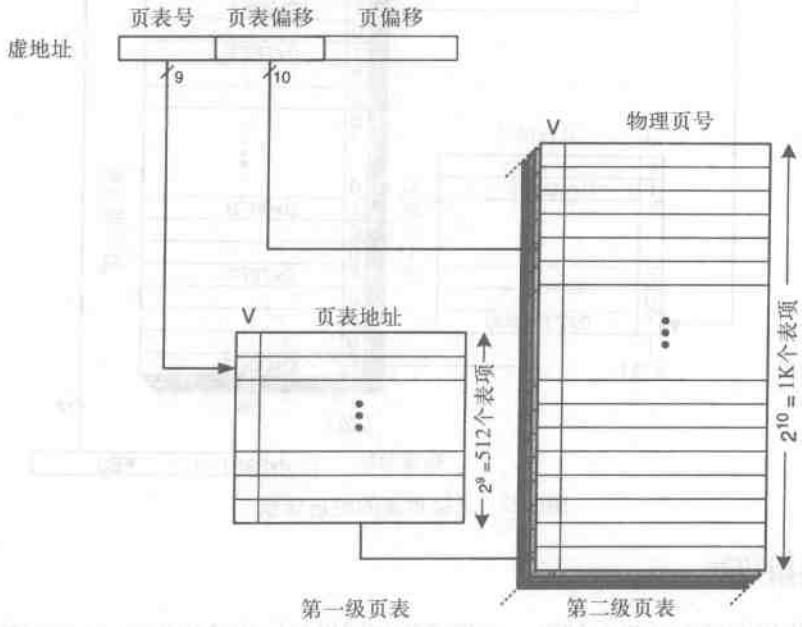


图8-26 多级页表

图8-26中，19位虚页号被分为9位的页表号和10位的页表偏移。因此，第一级页表有2^9 = 512个表项。这512个第二级页表均有2^{10} = 1K个表项。如果每个一级和第二级页表表项占用32位（4个字节），而且只有两个第二级页表同时在物理存储器中，这个层次化页表结构只使用了（512×4字节）+2×（1K×4字节）=10KB的物理存储器空间。两级页表只需要存储全部页表的2MB物理存储器的一小部分。两级页表的缺点是TLB缺失时转换过程将增加一次额外的存储器访问。

例8.16 使用多级页表完成地址转换

图8-27为图8-26所示的两级页表可能包含的内容。仅给出了一个第二级页表的内容。使用这个两级页表，描述访问虚地址0x003FEFB0时发生了什么情况。

解：如常一样，只有虚页号需要转换。虚地址中最高9位为页表号0x0，这是第一级页表的索引。第一级页表的0x0号表项说明第二级页表在内存中（V=1），其物理地址为0x2375000。

虚地址的后10位（0x3FE）为页表偏移，给出了第二级页表的索引。第二级页表的表项0位于底部，表项0x3FF位于顶部。第二级页表的第0x3FE号表项说明虚页在物理存储器（V=1）中，且物理页号为0x23F1。将物理页号和页偏移拼接起来，形成物理地址0x23F1FB0。

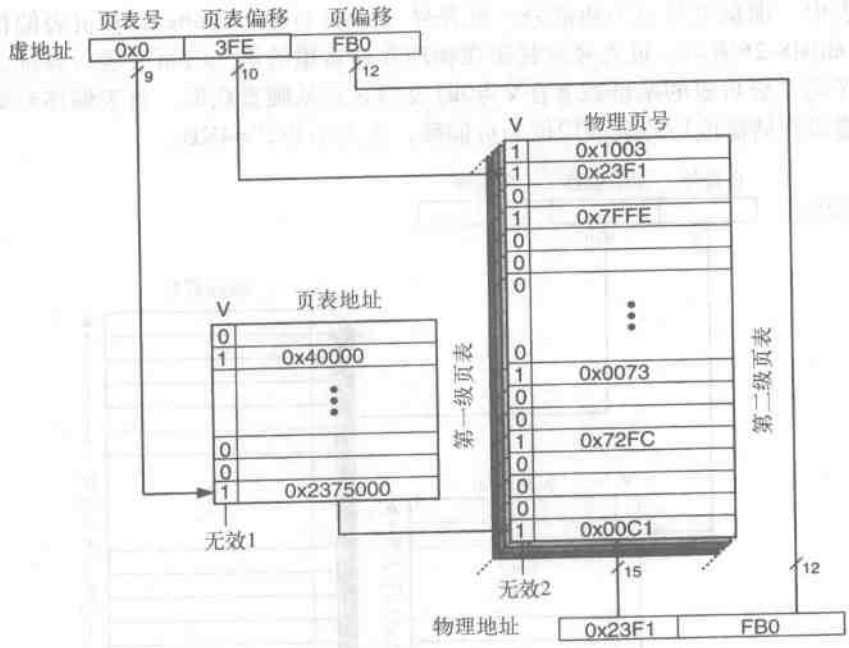


图8-27 两层页表的地址转换

8.5 内存映射I/O*

处理器也使用存储器接口与键盘，监视器和打印机等输入输出设备（Input/Output device, I/O device）通信。处理器就像访问内存一样使用地址和数据总线访问I/O设备。

一部分的地址空间用于I/O设备而非内存。例如，设定0xFFFF0000到0xFFFFFFFF范围内的地址用于I/O。回忆6.6.1节，这些地址为存储器映射的保留区域。每一个I/O设备可以指定到这一范围中的一个或者多个地址上。对于特定地址的写入操作就会给设备发送数据，读出操作就将从设备接收数据，这种I/O设备通信方式称为内存映射I/O（memory-mapped I/O）。

在内存映射I/O系统里，读出和写入可能访问内存，也可能访问I/O设备。图8-28给出了支持两个内存映射I/O设备所需要的硬件。其中，地址译码器决定处理器与哪个设备进行通信，

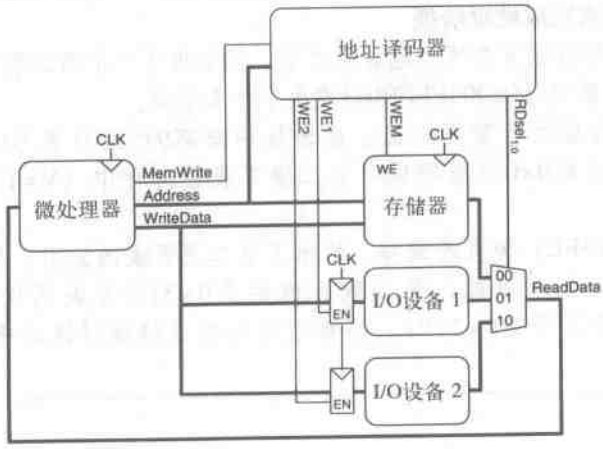


图8-28 地址映射I/O所需的硬件

根据Address和MemWrite信号来产生对其他硬件的控制信号。ReadData多路选择器在内存和各种I/O设备间选择读出数据。写使能寄存器保存了写入到I/O设备的值。

例8.17 与I/O设备通信

假设图8-28中的I/O设备1被指定到了内存地址0xFFFFFFF4。写出把值7写入到I/O设备1和从I/O设备1读出输出值的MIPS汇编语言。

解：以下MIPS汇编代码把值7写入到I/O设备1[⊖]。

```
addi $t0, $0, 7
sw   $t0, 0xFFF4($0)
```

因为地址为0xFFFFFFF4，地址译码器将设置WE1，同时设置MemWrite为TRUE。WriteData总线的值（7）被写入到与I/O设备1输入管脚相连的寄存器中。

为了从I/O设备读出，处理器应执行以下的MIPS汇编代码。

```
lw $t1, 0xFFF4($0)
```

因为地址译码器发现地址位0xFFFFFFF4且Mem Write为FALSE，所以它把$RDsel_{1:0}$设置为01。I/O设备1的输出通过多路选择器传递到ReadData总线，然后被读入到处理器的$t1中。

与I/O设备通信的软件称为设备驱动程序（device driver）。读者可能已经下载或者安装过打印机或者其他I/O设备的设备驱动程序。编写一个设备驱动程序需要详细了解I/O设备硬件知识。其他程序调用设备驱动提供的函数来访问设备，而不需知道底层设备的硬件。

为了说明内存映射I/O的硬件和软件，本节的后续部分将介绍商业用语言合成芯片与MIPS处理器的通信。

语音合成硬件

Radio Shack SP0256语音合成芯片生成像机器人说话的语音。每一个单词由一个或者多个基本音构成。例如，单词“hello”使用5个基本音，在SP0256语音合成芯片信号中用以下符号表示：HH1 EH LL AX OW。语音合成芯片使用6位代码表示英语中出现的64种不同基本音。例如，这五个在单词“hello”中基本音的十六进制表示分别为0x1B，0x07，0x2D，0x0F，0x20。处理器发出基本音序列到语音合成芯片，并由其来驱动扬声器发出声音。

图8-29为SP0256声音芯片的引脚排列。这些I/O引脚是与MIPS处理器相连的接口。引脚$A_{6:1}$接收6位经处理器编码的基本音。在Digital Out管脚上产生基本音。Digital Out信号首先被放大，之后被送到扬声器。另外两个引脚SBY和$\overline{ALD}$分别为状态引脚和控制引脚。当SBY输出为1时，表示语音合成芯片准备好接收新的基本音。在地址读取输入信号$\overline{ALD}$的下降沿，语音合成芯片读取由$A_{6:1}$指定的基本音。电源和地（V_{DD}和V_{SS}）、时钟（OSC1）等其他引脚必须按图示连接，但是它们不由处理器驱动。

图8-30为声音合成器连接到MIPS处理器的接口电路。处理器使用三个内存映射I/O地址与语音合成芯片通信。我们任意选择$A_{6:1}$端口映射到地址0xFFFFFF00，$\overline{ALD}$映射到0xFFFFFF04，SBY映射到0xFFFFFF08。尽管WriteData总线为32位，但只有最低6位用于$A_{6:1}$，最低1位用于$\overline{ALD}$，其他位则被忽略。相似地，SBY在ReadData总线的最低位被读取，其他位为0。

声音合成芯片设备驱动程序

设备驱动程序通过内存映射I/O接口传输一系列合适的基本音来控制语音合成芯片。它遵

⊖ 注意16位立即数0xFFF4符号扩展位32位值0xFFFFFFF4。

循SP0256芯片所希望的协议，如下所示：

- 设置 $\overline{\text{ALD}}$ 为1；
- 等待芯片设置SBY信号为1，表明它已经结束发出前面的基本音同时为下一次做好了准备；
- 写入6位基本音到$A_{6:1}$；
- 设置 $\overline{\text{ALD}}$ 为0，初始化语音。

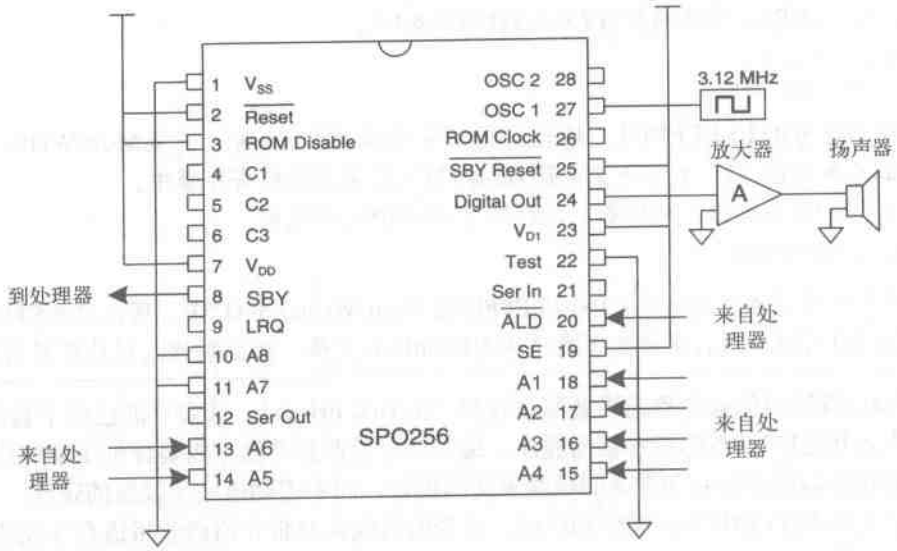


图8-29 SP0256语音合成芯片的引脚排列

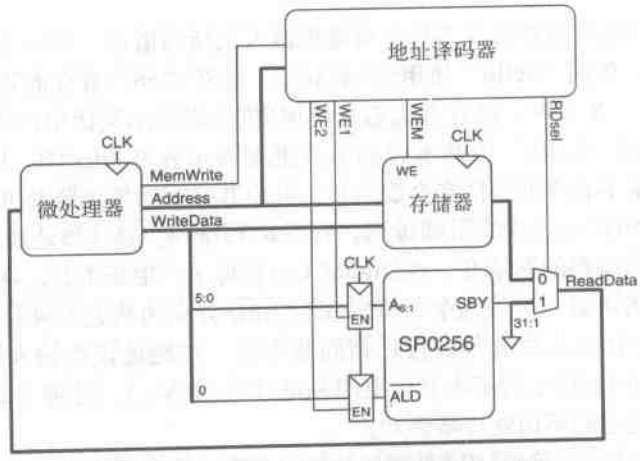


图8-30 SP0256语音合成器的硬件驱动

这一过程可以对任何数目的基本音重复进行。代码示例8.1中的MIPS汇编代码写入5个基本音到语音合成芯片。基本音编码以32位形式编码，存储在从内存地址0x10000000开始的包含5个表项的数组中。

代码示例8.1 语音芯片设备驱动

```
init:
     addi $t1, $0, 1      # $t1 = 1 (value to write to ALD)
     addi $t2, $0, 20     # $t2 = array size * 4
     lui  $t3, 0x1000     # $t3 = array base address
```

```
        addi $t4, $0, 0      # $t4 = 0 (array index)

start:
        sw   $t1, 0xFF04($0) # ALD =1
loop:
        lw   $t5, 0xFF08($0) # $t5 = SBY
        beq  $0,  $t5, loop  # loop until SBY == 1

        add  $t5, $t3, $t4   # $t5 = address of allophone
        lw   $t5, 0($t5)     # $t5 = allophone
        sw   $t5, 0xFF00($0) # A6:1 = allophone
        sw   $0,  0xFF04($0) # ALD = 0 to initiate speech
        addi $t4, $t4, 4     # increment array index
        beq  $t4, $t2, done  # last allophone in array?
        j    start           # repeat

done:
```

代码示例8.1中的汇编代码使用了轮询（poll）方式重复检查SBY信号，以确定语音合成芯片何时准备好接收新的基本音。代码虽然可以正确运行，但是浪费可以执行有效工作的宝贵处理器周期。处理器可以使用将中断（interrupt）连接到SBY以代替轮询方法。当SBY电平上升时，处理器停止当前正在运行的程序，并跳转到中断处理程序代码。在这个语音合成器的例子里，中断控制程序将发出下一个基本音，然后让处理器恢复中断之前进行的程序运行。处理器采用6.7.2节描述的处理其他异常方法来处理中断。

8.6 现实世界透视：IA-32存储器和I/O系统*

处理器速度越来越快，需要更复杂的存储器层次结构以保持提供稳定的数据流和指令流。本节介绍了IA-32处理器的存储器系统，以说明这个发展过程。7.9节中的处理器照片突出显示了片上高速缓存。IA-32还使用了一个与大部分常见内存映射I/O不同的直接I/O编程方式。

8.6.1 IA-32高速缓存系统

1985年开始生产的80386以16MHz的频率工作。它没有高速缓存，直接访问主存以获取所有指令和数据。由于存储器速度的差异，处理器可能立即得到响应，也可能会暂停一到两个周期等待内存响应。这些等待的周期称为等待状态（wait state），将会增加处理器的CPI。从那时候开始，微处理器的频率每年都提高至少25%，然而内存的延迟并没有降低。从处理器向主存发出地址到主存返回数据的延迟目前已经超过100个时钟周期。因此，低缺失率的高速缓存是提高性能所必须的。表8-5总结了英特尔IA-32处理器高速缓存系统的发展。

表8-5 Intel IA-32微处理器存储器系统的发展

微处理器	年代	主频（MHz）	第一级数据高速缓存	第一级指令高速缓存	第二级高速缓存
80386	1985	16～25	无	无	无
80486	1989	25～100	8KB统一高速缓存		不在片上
Pentium	1993	60～300	8KB	8KB	不在片上
Pentium Pro	1995	150～200	8KB	8KB	256KB～1MB（多芯片模块）
Pentium II	1997	233～450	16KB	16KB	256～512KB（盒装）
Pentium III	1999	450～1400	16KB	16KB	256～512KB（盒装）
Pentium 4	2001	1400～3730	8～16KB	12KB微操作追踪高速缓存	片上256KB～2MB
Pentium M	2003	900～2130	32KB	32KB	片上1～2MB
Core Duo	2005	1500～2160	32KB/核心	32KB/核心	片上2MB共享

80486引入了一个统一的写直达高速缓存来保存指令和数据。大部分高性能的计算机系统使用速度大大高于主存的商业SRAM芯片在主板上提供了更大的二级高速缓存。

Pentium处理器引入了分离的指令和数据高速缓存，以防止同时请求指令和数据时所产生的竞争。高速缓存使用写回策略以减少与主存的通信。同样地，在主板上提供了一个更大的第二级高速缓存（一般为256～512KB）。

P6系列处理器（Pentium Pro，Pentium Ⅱ，和Pentium Ⅲ）的设计可以支持更高的时钟频率。主板上的第二级高速缓存没有继续保留，而被移到与处理器更接近的位置，以改进其延迟和吞吐量。Pentium Pro的封装采用了多芯片模块（Multichip Module, MCM），其中包含了处理器芯片和第二级高速缓存芯片，如图8-31所示。和奔腾相似，它采用分离的8KB第一级指令和数据高速缓存。然而，这些高速缓存是非阻塞的（nonblocking）：即使特定访问在高速缓存中缺失而必须从主存获取数据时，乱序执行的处理器仍然可以继续后续的高速缓存访问。第二级高速缓存容量为256KB、512KB或者1MB大小，并可以处理器相同的速度运行。不幸的是，对于大批量的制造而言，多芯片模块封装过于昂贵。因此Pentium Ⅱ以包含处理器和第二级高速缓存的低成本盒装发售。第一级高速缓存容量增加了一倍，以弥补第二级高速缓存以处理器一半速度运行的不足。Pentium Ⅲ在处理器的同一颗芯片上直接集成了一个全速第二层高速缓存。同一芯片上的高速缓存能以更佳的延迟和吞吐量运行，它比片外的同等容量高速缓存更有效率。

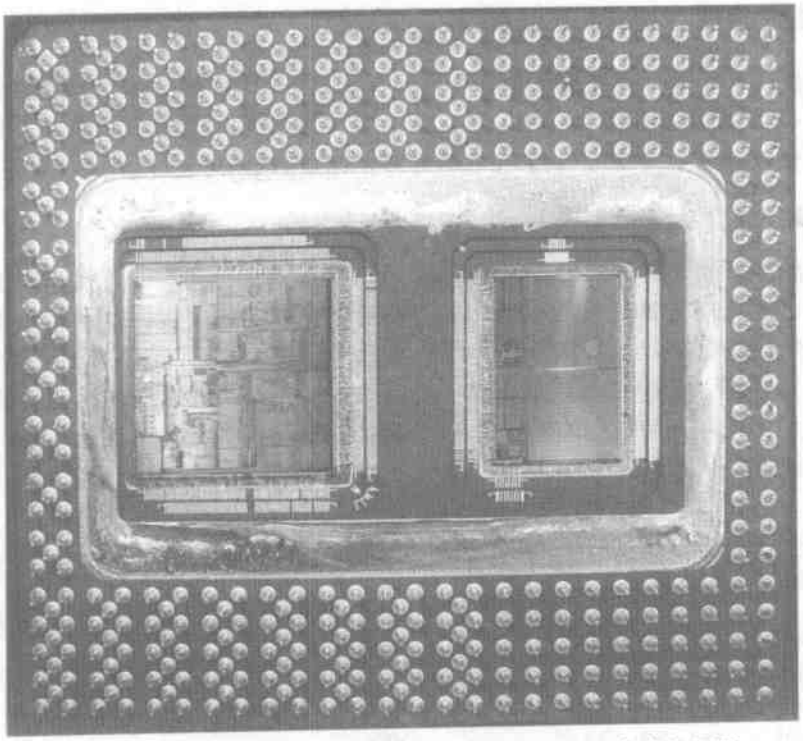


图8-31 PGA封装中的Pentium Pro 多芯片模块，左边为处理器，右边为256KB高速缓存（得到Intel公司许可）

Pentium 4处理器提供一个非阻塞的第一级数据高速缓存。在指令被译码为微操作后，它变为追踪高速缓存（trace cache）来存储指令，从而避免指令每次从高速缓取出后而进行重复译码的延迟。

Pentium M处理器的设计来源于Pentium Ⅲ处理器。它大幅度增加第一级高速缓存容量到32KB，配置一个1到2MB的第二级高速缓存。Core Duo包含了两个改进的Pentium M处理器和一个共享的2MB片上高速缓存。共享的高速缓存可用于处理器间的通信：一个处理器写入高速缓存，另一个处理器读出。

8.6.2 IA-32虚拟存储器

IA-32处理器在实模式或者保护模式下运行。实模式（real mode）向后兼容原始的8086。它只使用20位地址，内存容量限制为1MB，也不支持虚拟存储器。

80286中引入了保护模式（protected mode），80386将其被扩展到32位地址。它支持4KB页的虚拟存储器。它还提供内存保护，一个程序不能访问属于其他程序的页。因此，一个有漏洞或者恶意的程序就不能破坏或者影响其他程序了。所有现代的操作系统都使用保护模式。

32位的地址允许最多4GB的存储器空间。从Pentium Pro开始，处理器就已经使用物理地址扩展（physical address extension）技术扩充存储器容量到64GB。每一个进程使用32位地址，虚拟存储器系统将这些地址映射到更大的36位虚拟存储器空间。它对每个进程使用不同的页表，所以每一个进程可以它自己的4GB地址空间。

8.6.3 IA-32 的直接I/O编程机制

大部分体系结构使用8.5节描述的内存映射I/O机制，其中程序以读和写内存位置的方式访问I/O设备。IA-32的直接I/O编程机制中采用特别的IN和OUT指令实现读和写I/O设备。IA-32定义2^{16}个I/O端口。IN指令从DX指定的端口读取1，2或者4字节到AL、AX或EAX寄存器。OUT指令与之相似，但是写入端口。

把外围设备连接到直接I/O编程系统跟把它连接到内存映射系统很相似。当访问一个I/O端口时，处理器发送端口号（而非内存地址）到地址总线的最低16位。设备从数据总线读或者写数据。最大的不同在于处理器还产生一个M/$\overline{IO}$信号。当M/$\overline{IO}$ = 1时，表示处理器正在访问内存。当它为0时，进程正在访问一个I/O设备。地址译码器也将检查M/$\overline{IO}$信号以产生合适的主存和I/O设备的使能信号。I/O设备同样可以向处理器发送中断来表示它们准备好通信。

8.7 总结

存储器系统的组织是决定计算机性能的主要因素。DRAM、SRAM和硬盘等不同的存储技术在容量，速度和消耗等三方面提供了不同的折中。本章介绍了基于高速缓存和虚拟存储器的组织结构，它们使用存储器层次结构方法提供了接近理想的大容量、快速、廉价的存储器系统。主存一般用DRAM构成，其速度明显比处理器慢。高速缓存把常用数据保存在快速SRAM中以减少访问时间。虚拟存储器用硬盘存储暂时不需要在主存的数据以增加内存容量。高速缓存和虚拟存储器增加了计算机系统的复杂度和硬件，但是益处往往超过成本。所有的现代个人计算机都使用高速缓存和虚拟存储器。大部分处理器还使用内存接口与I/O设备通信。这被称为内存映射I/O。程序使用读出和写入操作访问I/O设备。

结语

这一章把我们带到了数字系统世界旅程的终点。我们希望本书不仅让读者学习到工程技术知识，也能令你感受到美妙和令人神往的数字电路设计艺术。读者学习了如何使用原理图和硬件描述语言设计组合和时序逻辑，熟悉了多路选择器、ALU、存储器等较大的数字电路模块。

计算机是最吸引人的数字系统应用之一。读者已经学习了如何用汇编语言对MIPS处理器编程和如何使用数字电路模块构造微处理器和存储器系统。读者可以发现了抽象、规范、层次化、模块化和规整化等原则贯穿了全书。通过这些技术原则，我们可以完成微处理器内部运行这个拼图难题。从移动电话到数字电视再到火星探测器和医学影像系统，我们的世界日益数字化。

试想像，在一个半世纪之前Charles Babbage在与魔鬼的交易中也有相似的经历？他只不过渴望以机械精度来计算数学用表。今天的数字系统是昨天的科幻小说。Dick Tracy（20世纪30年代美国连环漫画人物）曾在电话里听说过iTunes吗？Jules Verne（19世纪法国科幻作家）会发射全球定位卫星星座到太空吗？Hippocrates（古希腊物理学家和医学家）能用过高分辨率的脑部数字照片治疗疾病吗？但是同时，George Orwell噩梦中无所不在的政府监视正一天天地走向现实。而且罪犯声称可以用先进的膝上电脑开发核武器，其计算能力比冷战时期用于模拟炸弹实验的房间大小的超级电脑还强。微处理器的发展和进步仍在加速。未来10年的变化将会超过以往。读者现在已经有工具去设计和建造那些可以改造我们未来的新系统。更高的能力带来更多的责任。我们希望读者不仅仅为了娱乐或金钱来利用它，而应为了人类的利益。

习题

8.1 用简短的语言描述4个日常活动来说明时间局部性和空间局部性。每一种局部性说出两个例子，并加以解释。

8.2 用一个段话描述两个短的可以利用时间局部性和（或）空间局部性的计算机应用。说明原理。

8.3 给出一个地址序列，针对此地址序列，容量为16个字，块大小为4个字直接映射的高速缓存性能将优于具有同样容量和块大小、采用LRU替换策略的全相联高速缓存。

8.4 重做习题8.3的例子，这时全相联高速缓存优于直接映射高速缓存。

8.5 在下述高速缓存参数中，增加其中一项而保持其他参数不变时，描述所产生的性能变化。

(a) 块大小

(b) 相联性

(c) 高速缓存大小

8.6 两路组相联高速缓存的性能一定比同样容量和块大小的直接映射高速缓存好吗？请解释。

8.7 以下是关于高速缓存缺失率的说法。标志每句话是对还是错。简单解释你的原因，当说法是错的时候，给出一个反例。

(a) 一个两路组相联高速缓存比有同样容量和块大小的直接映射高速缓存有更低的缺失率。

(b) 一个16KB大小的直接映射高速缓存比有同样块大小的8KB直接映射高速缓存有更低的缺失率。

(c) 块大小为32字节的指令高速缓存一般比一个8字节块大小，有同样的相联度和总容量的指令高速缓存有更低的缺失率。

8.8 高速缓存有以下的参数：块大小b（以字为单位），组数S，路数N，地址位数A。

(a) 以给出的参数表示，高速缓存容量C是多少？

(b) 以给出的参数表示，需要多少位来存放标志？

(c) 全相联高速缓存的容量是C，块大小是B，这是S和N是多少？

(d) 直接映射高速缓存的容量为C，块大小为b，S为多少？

8.9 16字高速缓存的参数如习题8.8给出。考虑以下重复的lw地址序列（以16进制给出）：

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

假设对相联高速缓存采用最近最少使用(LRU)替换策略，如果这个地址序列输入到以下高速缓存，忽略开始的影响（也就是强制缺失），计算有效缺失率。

(a) 直接映射高速缓存，S=16，b=1字。

(b) 全相联高速缓存，N=1，b=1字。

(c) 两路组相联高速缓存，S=8，b=1字。

(d) 直接映射高速缓存，S=8，b=2字。

8.10 假设以以下数据访问模式运行程序。这个模式仅只运行一次。

0x0，0x8，0x10，0x18，0x20，0x28

(a) 如果使用直接映射高速缓存，容量为1KB，块大小为8个字节（2字），高速缓存内有多少组？

(b) 针对（a）中的高速缓存，在此直接映射高速缓存中，针对给出的内存访问模式，缺失率是多少？

(c) 针对给出的内存访问模式，以下哪一个方法最能减少缺失率？（高速缓存容量保持不变）。圈出你的答案。

(i) 增加相联度为2

(ii) 增加块大小到16字节

(iii) (i) 和 (ii) 都可以

(iv) (i) 和 (ii) 都不可以

8.11 你正在为MIPS处理器设计一个指令高速缓存。它的总容量为$4C = 2^{c+2}$字节，采用$N=2^n$路组相联（$N \geq 8$），块大小为$b = 2^b$个字节（$b \geq 8$）。以已知参数的形式给出以下问题的答案。

(a) 地址的哪些位用于选择块里的字？

(b) 地址的哪些位用于选择高速缓存里的组？

(c) 每一个标志有多少位？

(d) 整个高速缓存中有多少标志位？

8.12 考虑以下参数的高速缓存：N（相联度）=2，b（块大小）=2个字，W（字大小）=32位，C（高速缓存大小）=32K字，A（地址大小）=32位。你只需要考虑字地址。

(a) 给出地址中的标志、组、块偏移和字节偏移位，每个字段需要多少位？

(b) 高速缓存中所有标志占多少位？

(c) 假设每个高速缓存块还有一位有效位(V)和一位脏位（D）。每一个高速缓存组（包括数据，标志和状态位）需要有多少位？

(d) 使用图8-32中的模块和少量的2输入逻辑门电路设计高速缓存。高速缓存的设计必须包括标志存储、数据存储、地址比较、数据输出选择和任何你认为需要的部件。注意多路选择器和比较器块可以为任何大小（分别为n或者p位宽），但是SRAM块必须为16K×4位。请给出包含简明标志的电路模块图。

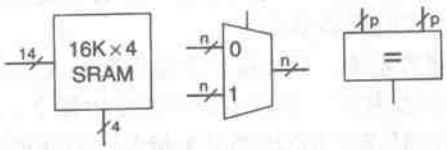


图8-32 电路模块

8.13 你参加了一个热门的新互联网创业，用内嵌传呼机和网络浏览器开发腕表。它使用的嵌入式处理器中采用了图8-33所述的多级高速缓存方案。处理器包括一个小型的片上高速缓存和一个大型的片外第二级高速缓存（对，这个手表重3磅，但是你可以用它上网）。

假设处理器使用32位物理存储器地址但是只以字边界访问数据。表8-6给出了高速缓存参数。DRAM的访问时间为t_m，大小为512MB。

图8-33 计算机系统

表8-6 存储器特性

特征	片上高速缓存	片外高速缓存
组织方式	四路组相联	直接映射
命中率	A	B
访问时间	t_a	t_b
块大小	16字节	16字节
块数目	512	256K

(a) 对于存储器中给定的字，在片上高速缓存和第二级高速缓存中总共有多少个可能的位置能找到它？

(b) 片上高速缓存和第二层高速缓存的每个标志各需要多少位？

(c) 给出内存平均访问时间的表达式。两级高速缓存按顺序连续访问。

(d) 对于某一特定问题，测量发现片上高速缓存命中率为85%，第二层高速缓存命中率为90%。然而，当屏蔽片上高速缓存时，第二层高速缓存的命中率提高到98.5%。请解释这个现象。

8.14 本章描述了最近最少使用（LRU）的多路相联高速缓存替换策略。还有一些不太常见的策略，如先入先出（FIFO）策略和随机策略。FIFO策略替换出存在最长时间的块，而不考虑它是否最近被访问过。随机策略则随机选择一个块作替换。

(a) 讨论这些替换策略的优缺点。

(b) 描述一个FIFO会比LRU性能更好的访问模式。

8.15 你正在设计的计算机存储体系结构为分离的指令和数据高速缓存，并使用图7-41中的MIPS多周期处理器，主频为1GHz。

(a) 假设指令高速缓存已经完美（总是命中），但是数据高速缓存有5%的缺失率。在高速缓存缺失时，处理器暂停60ns访问主存，之后恢复正常操作。考虑高速缓存缺失的情况，平均内存访问时间为多少？

(b) 考虑到非理想的存储器系统，平均每一条读出和写入指令需要多少时钟周期？

(c) 考虑例7-7中的基准测试程序，其中有25%的读出指令，10%的存储指令，11%的分支判断指令，2%的跳转指令和52%的R类指令。对非理想化的存储器系统考虑，这个基准测试程序的平均CPI是多少？

(d) 现在假设指令高速缓存也是非理想化的，缺失率为7%，那么（c）部分基准测试程序的平均CPI是多少？把指令和数据高速缓存缺失都考虑在内。

8.16 如果计算机使用64位虚地址，那么可以访问多少虚拟存储器。注意2^{40}字节=1terabyte，2^{50}字节=1petabyte，2^{60}=1exabyte。

8.17 一个超级计算机的设计者花费1百万美元在DRAM上，同时花费同样多的钱在硬盘上以作为虚拟存储器。根据图8-4的价格，这台计算机可以拥有多大的物理存储器和虚拟存储器。需要多少位的物理和虚地址来访问这个存储器系统？

8.18 考虑一个可以寻址全部2^{32}字节的虚拟存储器系统。你有无限的硬盘空间，但是只有有限的8MB物理存储器。假设虚页和物理页都是4KB大小。

(a) 物理地址为多少位？

(b) 系统中最大的虚页号是多少？

(c) 系统中有多少物理页？

(d) 虚页号和物理页号占多少位？

(e) 假设你设计了一个把虚拟存储器映射到物理存储器上的直接映射方案。该映射使用虚页号的若干最低有效位来确定物理页号。每一个物理页上可以映射多少虚页？为什么这里的直接映射不是一个好的方案？

(f) 明显地，需要一个比（e）部分更有灵活性和动态性的虚拟存储器地址到物理地址转换方案。假设你使用一个页表存储映射（从虚页号到物理页号的转换）。页表将需要包含多少个页表表项？

(g) 除了物理页号以外，每个页表表项还要包括一些状态信息，例如有效位（V）和脏位（D）。每一个页表表项需要占用多少字节？（按照整数字节向上取整）

(h) 给出页表的布局图。页表的大小是多少字节？

8.19 你决定使用地址转换后备缓冲（TLB）为习题8.18的虚拟存储器系统加速。假设内存系统的参数如表8-7所示。TLB和高速缓存的缺失率表示所请求的内容找不到的概率。主存缺失率表示页面缺失的概率。

表8-7 存储器特性

存储器部件	访问时间（Cycles）	缺失率
TLB	1	0.05%
高速缓存	1	2%
主存	100	0.0003%
硬盘	1 000 000	0%

(a) 在加上TLB前虚拟存储器系统的平均内存访问时间是多少？假设页表常驻在物理存储器，而不会保存在数据高速缓存内。

(b) 如果TLB有64个表项， TLB大小为多少位？每个表项中包括以下字段：数据（物理页号），标志（虚页号）和有效位。给出各个字段所占用的位数。

(c) 画出TLB的草图，清楚标志所有字段和尺寸。

(d) 需要多大容量的SRAM来构(c)部分描述的TLB？以深度×宽度的形式给出答案。

8.20 假设7-4节描述的MIPS多周期处理器使用虚拟存储器系统。

(a) 在多周期处理器原理图内画出TLB的位置。

(b) 描述加入TLB后如何影响处理器的性能。

8.21 你正在设计的虚拟存储器系统使用一个以专用硬件（SRAM和相关逻辑）构成的单层页表。它支持25位虚地址，22位物理地址和2^{16}字节（64KB）的页。每一页表表项包含一个物理页号，可用位（V）和脏位（D）。

(a) 页表的总大小为多少位？

(b) 操作系统组建议将页大小从64KB减少到16KB。但是你们小组的硬件工程师坚决反对，认为这将增加硬件开销。说出他们的理由。

(c) 页表将与片上高速缓存一起集成在处理器芯片上。片上高速缓存只对物理地址（不对虚拟地址）操作。对于给定的内存访问，可以同时访问片上高速缓存合适的组和页表吗？简要解释同时访问高速缓存组和页表表项的必要条件。

(d) 对于给定的内存访问，可以同时执行片上高速缓存标志比较和访问页表吗？简要解释原因。

8.22 描述虚拟存储器系统可能影响应用写入的场景。必须讨论页大小和物理存储器大小如何影响程序的性能。

8.23 假设你的个人电脑使用32位虚地址。

(a) 每一个程序可以用最多多少的虚拟存储器空间？

(b) PC硬盘的大小如何影响性能？

(c) PC的物理存储器大小如何影响性能？

8.24 使用MIPS内存映射I/O与用户交互。每一次用户按下按钮，所选择的模式就在五个发光二极管（LED）上显示。假设输入按钮映射到地址0xFFFFFF10，LED映射到地址0xFFFFFF4。当按钮按下时，输出为1，否则为0。

(a) 写出实现这个功能的MIPS代码。

(b) 为这个内存映射I/O系统画出和图8-30相似的原理图。

(c) 写出HDL代码以实现内存映射I/O系统的地址译码器。

8.25 第3章中设计的有限状态机（FSM），也可以用软件实现。

(a) 使用MIPS汇编语言实现图3-25的交通灯有限状态机。输入（T_A和T_B）映射到地址0xFFFFF000的第1位和第0位。两个3位输出（L_A和L_B）分别映射到地址0xFFFFF004的第0～2位和第3～5位。假设每一个灯的输出L_A和L_B采用独热编码：红100，黄010，绿001。

(b) 为这个内存映射I/O系统画出和图8-30相似的原理图。

(c) 写出HDL代码以实现内存映射I/O系统的地址译码器。

面试问题

8.1 解释直接映射，组相联和全相联高速缓存的不同。对于每一种高速缓存类型，给出一个程序，其性能要好于其他两种高速缓存。

8.2 解释虚拟存储器系统是如何工作的。

8.3 解释使用虚拟存储器系统的优点和缺点。

8.4 解释存储器系统的虚页大小如何影响高速缓存的性能。

8.5 用于内存映射I/O的地址可以被高速缓存吗？解释为什么。

附录A 数字系统实现

A.1 导言

本附录介绍了数字电路设计中的实践方面内容。本附录中的材料对理解此书的其他内容并非必不可少。但是，本章内容有助于向读者揭示构造实际数字系统的过程。而且，我们认为理解数字系统的最佳方法就是在实验室里自主实现和调试实际系统。

数字系统往往由一片或多片芯片组成。一种实现策略是将单独的逻辑门或较复杂的芯片（例如算术逻辑单元，存储器等）连接在一起。另一种实现策略是使用可编程逻辑芯片，其中包含的通用电路阵列可以实现特定的逻辑功能。第三种实现方法是设计包含了系统所需功能的专用集成电路。这三种策略在成本、速度、功耗和设计时间上各有不同，这将在后续章节中介绍。此附录还将介绍电路的物理封装，连接芯片的传输线，以及数字系统的经济性。

A.2 74系列芯片

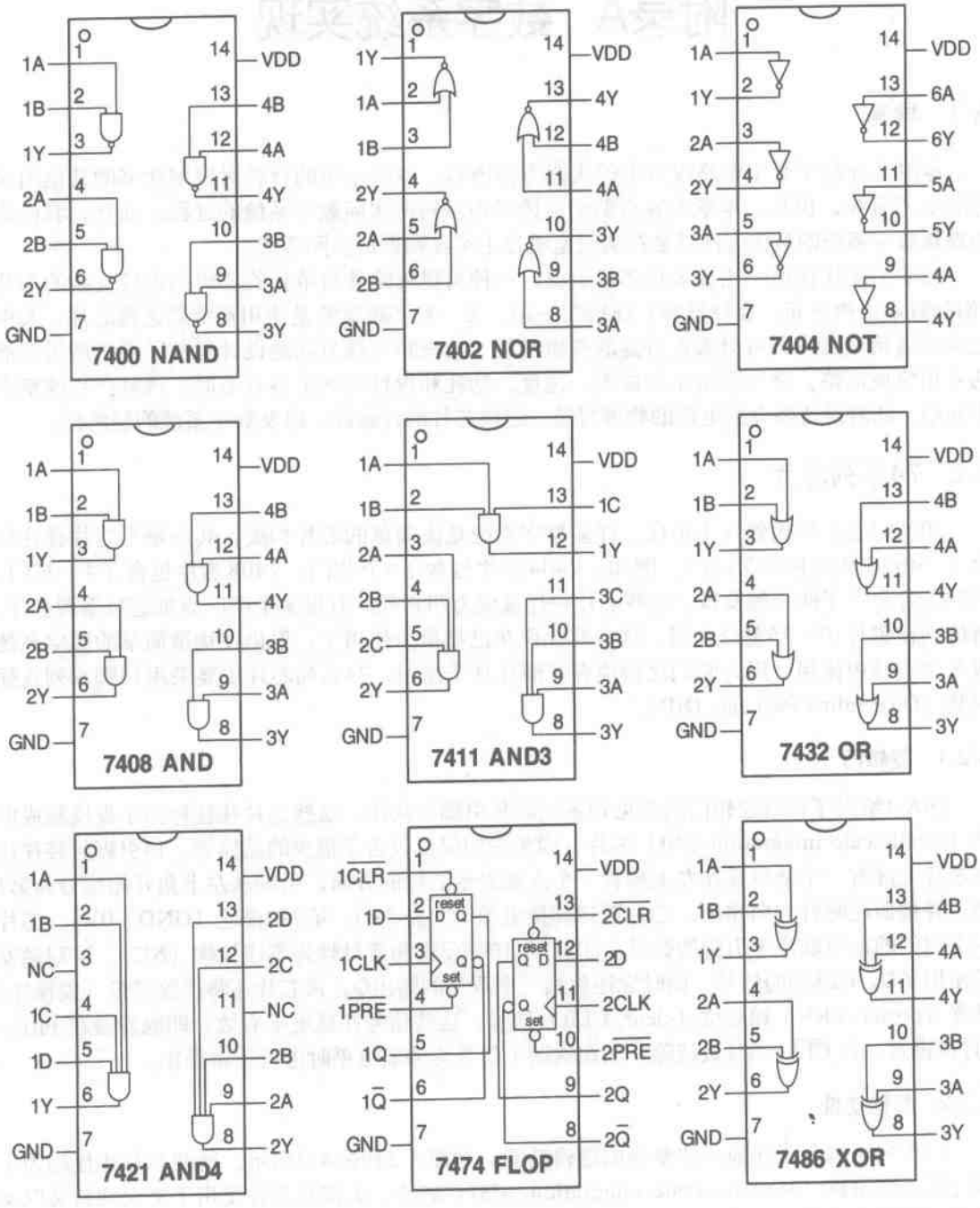
20世纪七十年代到八十年代，许多数字系统是由简单的芯片构成，其中每个芯片往往包含了一组相同功能的逻辑门。例如，7404芯片包含了6个非门，7408芯片包含了4个与门，7474芯片包含了两个触发器。这些芯片一起被成为74系列。有很多生产厂商制造74系列芯片，售价约在每片10～25美分之间。这些芯片现在已经很少使用了，但仍在构造简单的数字系统或教学实验中使用，因为它们比较便宜，而且易于使用。74系列芯片主要采用14脚双列直插封装（Dual Inline Package, DIP）。

A.2.1 逻辑门

图A-1给出了包含逻辑门的常见74系列芯片引脚布局图。这些芯片往往称为小规模集成电路（small-scale integration, SSI）芯片，因为它们仅仅包含了很少的晶体管。14引脚封装往往在芯片上部有一个缺口或在左上脚有一个点来表示芯片的方向。引脚从左上角开始编号为第1脚，并按照逆时针方向递增。芯片第14脚接电源（V_{DD}=5V），第7脚接地（GND＝0V）。芯片中逻辑门的数目取决于引脚的数目。注意7421的第3脚和第11脚为不连接脚（NC）。7474触发器使用了D，CLK和Q信号。同时它还包括一个取反的输出$\overline{Q}$。该芯片还将接收异步（或预先）设置（preset, PRE）和复位（clear, CLR）信号。这些信号在低电平有效，即触发器在$\overline{PRE}$＝0时被设置，在$\overline{CLR}$＝0时被清除，而在这两个信号均为高电平时进行正常操作。

A.2.2 其他功能

74系列还包含了其他一些复杂的逻辑功能，如图A-2和图A-3所示。这些芯片往往称为中等规模集成电路（Medium-scale integration, MSI）芯片。大部分芯片使用了更大的封装以实现更多的输入和输出。电源和地依然连接在芯片的右上角和左下角。这里只提供了芯片的基本功能，完整的描述请参见生产商提供的手册。



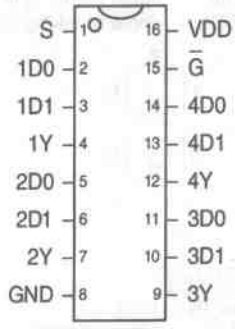
图A-1 常用74系列逻辑门

74153 4:1 多路选择器

两路4：1多路选择器

$D_{3:0}$:	data
$S_{1:0}$:	select
Y:	output
Gb:	enable

```
always @(1Gb, S, 1D)
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always @(2Gb, S, 2D)
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```

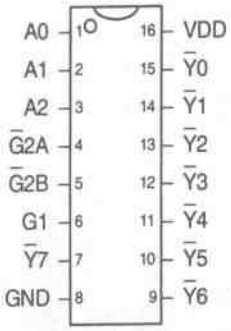


74157 2:1 多路选择器

四路2：1多路选择器

$D_{1:0}$:	data
S:	select
Y:	output
Gb:	enable

```
always @(*)
  if (~Gb) 1Y = 0;
  else     1Y = S ? 1D[1] : 1D[0];
  if (~Gb) 2Y = 0;
  else     2Y = S ? 2D[1] : 2D[0];
  if (~Gb) 3Y = 0;
  else     3Y = S ? 3D[1] : 3D[0];
  if (~Gb) 4Y = 0;
  else     4Y = S ? 4D[1] : 4D[0];
```

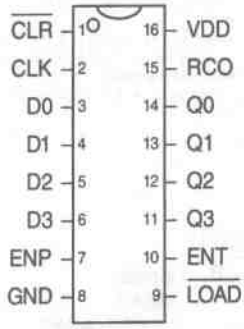


74138 3:8 译码器

3：8译码器

$A_{2:0}$:	address
$Yb_{7:0}$:	output
G1:	active high enable
G2:	active low enables

G1	$\overline{G2A}$	$\overline{G2B}$	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



74161/163 计数器

4位计数器

CLK:	clock
$Q_{3:0}$:	counter output
$D_{3:0}$:	parallel input
CLRb:	async reset (161)
	sync reset (163)
LOADb:	load Q from D
ENP, ENT:	enables
RCO:	ripple carry out

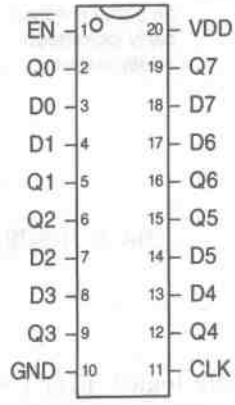
```
always @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```

74244 三态缓冲

8位三态缓冲

$A_{3:0}$:	input
$Y_{3:0}$:	output
ENb:	enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENB ? 4'bzzzz : 2A;
```



74377 寄存器

8位带使能的寄存器

CLK:	clock
$D_{7:0}$:	data
$Q_{7:0}$:	output
ENb:	enable

```
always @(posedge clk)
  if (~ENb) Q <= D;
```

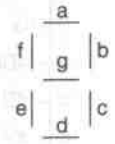
图A-2 中等规模集成电路

注：Verilog的名称不能以数字开始，但在图A-2中所示的示例代码中的名称之所以用数字是为了与制造商的数据表匹配。

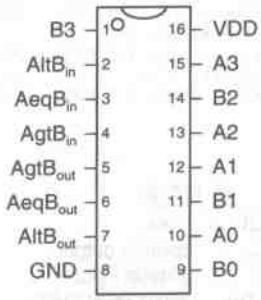
7447 7-段译码器

7段数码管显示译码器

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

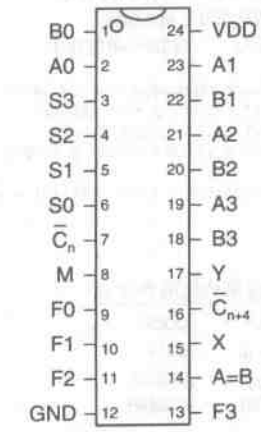


7485 Comparator

4位比较器

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always @(*)
  if (A > B | (A == B & AgtBin)) begin
    AgtBout = 1; AeqBout = 0; AltBout = 0;
  end
  else if (A < B | (A == B & AltBin) begin
    AgtBout = 0; AeqBout = 0; AltBout = 1;
  end else begin
    AgtBout = 0; AeqBout = 1; AltBout = 0;
  end
```



74181 ALU

4位ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always @(*)
  case (F)
    0000: Y = M ? ~A       :  A                        + ~Cbn;
    0001: Y = M ? ~(A | B) :  A       +  B             + ~Cbn;
    0010: Y = M ? (~A) & B :  A       + ~B             + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111                  + ~Cbn;
    0100: Y = M ? ~(A & B) :  A       + (A & ~B)       + ~Cbn;
    0101: Y = M ? ~B       : (A | B)  + (A & ~B)       + ~Cbn;
    0110: Y = M ? A ^ B    :  A       -  B             - Cbn;
    0111: Y = M ? A & ~B   : (A & ~B)                  - Cbn;
    1000: Y = M ? ~A + B   :  A       + (A & B)        + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A       +  B             + ~Cbn;
    1010: Y = M ? B        : (A | ~B) + (A & B)        + ~Cbn;
    1011: Y = M ? A & B    : (A & B)                   + ~Cbn;
    1100: Y = M ? 1        :  A       +  A             + ~Cbn;
    1101: Y = M ? A | ~B   : (A | B)  +  A             + ~Cbn;
    1110: Y = M ? A | B    : (A | ~B) +  A             + ~Cbn;
    1111: Y = M ? A        :  A                        - Cbn;
  endcase
```

图A-3 中等规模集成电路

A.3 可编程逻辑

可编程逻辑（programmable logic）包含了可配置成特定逻辑功能的电路阵列。我们已经介绍了三种可编程逻辑器件：可编程只读存储器（programmable read only memory, EROM），可编程逻辑阵列（programmable logic array, PLA）和现场可编程门阵列（field programmable gate array, FPGA）。本节将介绍它们的芯片实现技术。配置这些器件可以采用熔断片上熔丝的方法来控制电路组件的连接或断开。由于一旦熔丝熔断就不可以恢复，所以这种方法为一次性可编程（one-time programmable, OTP）。此外，也可以将配置信息存储在存储器中，这样

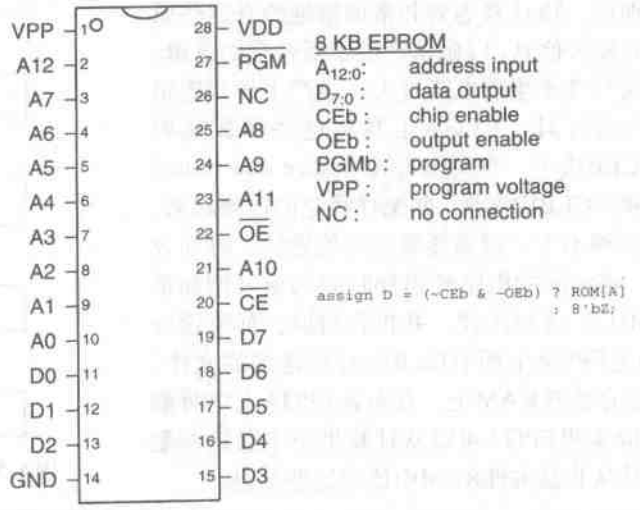
就可以多次编程。可编程逻辑在实验室中非常方便，因为同样的芯片可以在开发过程中多次使用。

A.3.1 PROM

正如5.57节讨论，PROM可以用于查找表。一个2^N个M位宽字的PROM可以编程为任意N输入和M输出的组合逻辑功能。当设计改变时，仅仅需要修改PROM的内容，而不需要重新连接芯片之间连线。查找表对小函数比较有效，但是当输入增加时其成本将非常高。

例如，典型的可擦写PROM芯片2764容量为8KB（64Kb），如图A-4所示。EPROM有13位地址线以确定访问8K字中的哪一个，8位数据线来读取此字的字节数据。在读操作时，芯片使能和读出使能都必须有效。最大传播延迟为200ps。在正常操作下，$\overline{PGM}$ = 1且VPP不使用。EPROM通常由特定编程器编程，其中将设置$\overline{PGM}$ = 0，VPP为13V，并通过特殊的输入序列来配置存储器。

现代PROM在概念上与之类似，但是容量更大，而且引脚更多。Flash存储器较PROM更为便宜。在2006年时每GB容量售价约为30美元，而且其价格平均每年下降30%～40%。



图A-4 2764KB EPROM

A.3.2 PLA

正如5.6.1节讨论，PLA包含了与门和或门以实现任意积之和形式的组合逻辑功能。与门和或门都使用PROM作为可编程方法。一个PLA芯片对每个输入有两列，对于每个输出有一列，对于每个最小项有一行。这种组织方法在实现很多函数时较PROM更为有效，但是对于具有较多IO和最小项数目的函数，阵列将增长速度也很快。

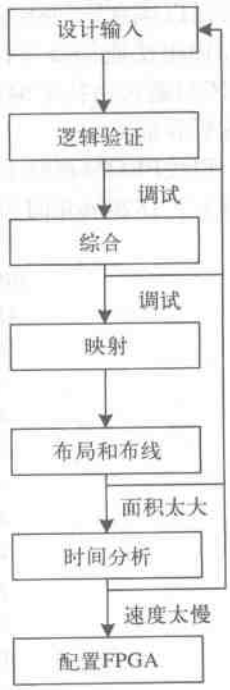
许多不同的制造商扩展了基本的PLA概念，生产出包含了寄存器的可编程逻辑器件(programmable logic device, PLD)。22V10是一个最流行的经典PLD器件。该芯片有12个输入和10个输出引脚。输出可以直接从PLA引出，也可以从芯片上时钟控制寄存器上引出。输出也可以反馈输入到PLA上。22V10可以直接实现最多12个输入，10个输出和10位状态的有限状态机。22V10在采购100片时价格约为2美元/片。由于FPGA在容量和成本上的进步，PLA已经逐渐过时。

A.3.3 FPGA

正如5.6.2节讨论，FPGA中包含了可配置逻辑块（configurable logic block, CLB）阵列，并通过可编程线连接起来。这些CLB中包含了小的查找表和触发器。FPGA规模可以平滑地扩展到非常大的容量，来包含数以千计的查找表。Xilinx公司和Altera公司是FPGA领域的两家领军企业。

查找表和可编程线提供了足够的灵活性来实现任意逻辑功能。然而，它们在芯片面积和速度方面较硬连线逻辑实现要差一个数量级。因此，FPGA中常包含特殊模块，例如存储器，多路选择器，甚至整个微处理器。

图A-5给出了FPGA上数字系统的设计流程。虽然一些FPGA工具也支持原理图方式，但设计往往通过硬件描述语言（HDL）给定。接着设计就需要经过模拟。对电路模块给定特定输入并比较是否与期待的输出相同，以验证（verify）逻辑是否正确。在这个阶段通常需要进行一些调试。接着，逻辑综合（synthesis）将硬件描述语言变为布尔函数。好的综合工具还可以产生函数的原理图，设计者需要非常谨慎地检查这些原理图和综合阶段报出的警告信息，以确认产生了所希望的逻辑。一些马虎的编码可能会导致产生的电路过大，或产生异步逻辑电路。当综合结果比较好时，FPGA工具将这些函数映射（map）到特定芯片的CLB块中。布局和布线（place and route）工具决定哪个函数在哪个CLB中实现，并怎样将它们连接起来。如果设计过大，芯片容纳不下，则需要重新开始设计。时间分析（timing analysis）将实际的电路延迟和时间约束（例如希望时钟主频达到100MHz）进行比较，并报告错误。如果设计正确，将产生一个指定FPGA上所有CLB内容和连线的文件。许多FPGA将配置信息存放在RAM上，在每次FPGA上电时都需要重复加载。在实验室里FPGA可以从计算机上下载这些配置信息，或者在上电时从非易失性ROM中读取这些信息。



图A-5 FPGA设计流程

例A.1 FPGA时序分析

Alyssa使用FPGA实现一个M&M排列器，其中根据颜色传感器的信息驱动马达将红色的糖果发到一个罐子里，将绿色的糖果放到另一个罐子。她的设计使用了有限状态自动机，并基于Spartan 3 系列的Spartan XC3S200 FPGA芯片实现。根据数据手册，FPGA的时间特征如表A-1所示。假设设计比较小，线延迟可以忽略不计。

Alyssa希望她的有限状态自动机可以运行在100MHz。那么在关键路径上最多可以有多少个CLB？其有限状态自动机最快的运行速度可以到多少？

解：在100MHz时，时钟周期T_C为10ns。Alyssa可以用式3.13来计算每个周期中最大组合路径传播延迟t_{pd}：

$$t_{pd} \leq 10\text{ns} - (0.72\text{ns} + 0.53\text{ns}) = 8.75\text{ns} \quad \text{(A-1)}$$

Alyssa最多可以用14个连续的CLB（8.75/0.61）来实现下一状态逻辑。

在Spartan 3 FPGA上实现的最快有限状态自动机是用一个CLB来产生下一状态逻辑。因此最小的时钟周期为：

$$T_c \geqslant 0.61ns + 0.72ns + 0.53ns = 1.86ns \quad (A\text{-}2)$$

因此，最大工作主频为538MHz。

表A-1 Spartan 3 XC3S200 时间参数

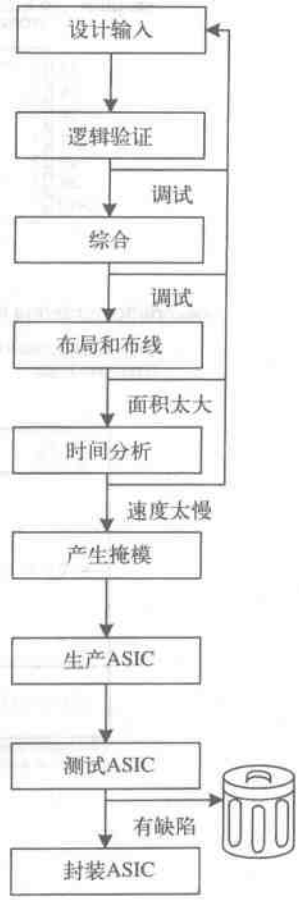
名字	值（ns）	名字	值（ns）
t_{pcq}	0.72	t_{pd}(per CLB)	0.61
t_{setup}	0.53	t_{skew}	0
t_{hold}	0		

Xilinx公司的XC3S100E FPGA具有1728个查找表和触发器，2006年时500 000片批量的售价为2美元/片。FPGA的价格平均每年下降30%左右，因此其使用将更加普及。

A.4 专用集成电路

专用集成电路（application-specific integrated circuits，ASICs）是为专门用途设计的集成电路。图形加速器、网络接口芯片、无线电话芯片都是ASIC的常见例子。ASIC设计师将布局晶体管以产生逻辑门，并将这些门连接起来。由于ASIC是针对特定功能硬连线的，所以它一般比FPGA实现同样功能时速度要快好几倍，芯片面积（成本）也要小一个数量级。然而，确定芯片上晶体管和连线如何布局的掩模（mask）需要花费上万美元生产。工厂的生产流程也需要6～12个月来完成制造、封装和测试ASIC芯片。如果在ASIC生产出来后发现错误，设计者必须改正问题，重新产生掩模，并等待下一批芯片生产出来。因此，ASIC仅仅适合于大量生产的芯片，而且其功能也需要预先准确定义。

图A-6给出的ASIC设计流程类似于图A-5中的FPGA设计流程。逻辑验证尤其重要，这是因为一旦在掩模生产出来后才发现错误，其修改成本非常高。综合产生的网表（netlist）包括了逻辑门和逻辑门之间的连接；对网表中的逻辑门进行布局，对门之间的连线进行布线。当设计满足要求时，将生产掩模并用于制造ASIC。一个灰尘的污点就可能毁坏一个ASIC芯片，因此芯片在生产后必须测试。生产出来后能正常工作芯片的比例称为良品率（yield）。良品率一般在50%～90%左右，这依赖于芯片大小和制造工艺的成熟程度。最后，能正常工作的芯片将放置在A.7节讨论的封装中。



图A-6 ASIC设计流程图

A.5 数据手册

集成电路的生产商都提供数据手册（data sheet）以描述其芯片的功能和性能。阅读和理解芯片手册是非常必要的。导致数字电路系统错误的一个重要原因就是没有正确理解芯片的操作。

数据手册往往可以从制造商的网站上获得。有一些数据手册的入口被隐藏以来了。制造商提供的芯片手册往往包含了多个相关的数据手册。芯片手册的起始部分一般都是介绍信息。

通过仔细的查找，这些信息一般都可以在Internet上找到。

本节将深入剖析德州仪器公司（Texas Instruments, TI）的非门芯片74HC04。这个数据手册相对简单但是说明了很多关键性问题。TI公司还生产74系列的很多其他芯片。过去有很多公司生产这类芯片，但是由于销售减少市场已经开始萎缩。

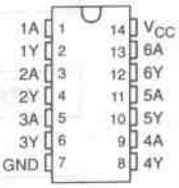
图A-7给出了数据手册的第一页。标题是SN54HC04, SN74HC04 HEX INVERTERS。

SN54HC04, SN74HC04
HEX INVERTERS

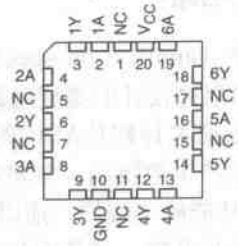
SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- Wide Operating Voltage Range of 2 V to 6 V
- Outputs Can Drive Up To 10 LSTTL Loads
- Low Power Consumption, 20-μA Max I_{CC}
- Typical tpd = 8 ns
- ±4-mA Output Drive at 5 V
- Low Input Current of 1 μA Max

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)



SN54HC04 . . . FK PACKAGE
(TOPVIEW)



NC – No internal connection

description/ordering information

The 'HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this data sheet.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Copyright (c) 2003, Texas Instruments Incorporated
On products compliant to MIL-PRF-38535, all parameters are tested unless otherwise noted. On all other products, production processing does not necessarily include testing of all parameters.

图A-7 7404数据手册的第一页

HEX INVERTERS表示芯片包含了6个非门。SN为TI的制造商标号。其他的制造商也有类似编号，例如Motorola标号为MC，National Semiconductor标号为DM。这些标号可以忽略，因为所有制造商提供的74系列芯片相互兼容。HC是逻辑族（high speed CMOS）。逻辑族决定了芯片的速度和功耗，而不是功能。例如7404，74HC04和74LS04芯片都包含了6个非门，但性能和成本却不相同。其他的逻辑族将在附录A.6中介绍。74系列芯片的操作温度为在商业区域（0～70℃）或工业区域（－40～85℃），54系列芯片则为军用区域（－55～125℃），其售价更高但也具有同样的功能。

7404有多种不同的封装，在购买之前需要认真考虑选择。封装由型号数字后面的一个字母表示。N表示塑料双列直插（plastic dual linline package, PDIP），它可以直接插在面包板中或可以固定在印刷电路板的通孔中。其他的封装将在附录A.7节中讨论。

功能表表示每个门将其输入取反。如果A为高（H），则Y为低（L），相反时也成立。这个表在本例子中显得很不重要，但是在复杂的芯片中却非常有意义。

图A-8中给出了数据手册的第二页。逻辑电路图表示了芯片包含了非门。如果工作条件超过了绝对最大（absolute maximum）部分描述的条件，芯片将被毁坏。其中，电源电压（V_{CC}，此手册中也称为V_{DD}）不能超过7V，连续输出电流不超过25mA。温阻（thermal resistance）或温阻抗θ_{JA}用于计算芯片在特定功耗情况下温度的上升情况。如果芯片周围的环境温度为T_A且芯片的当前功耗为P_{chip}，则芯片在此封装下自身节（conjunction）上的温度为：

$$T_J = T_A + P_{chip}\,\theta_{JA} \qquad \text{(A-3)}$$

例如，如果7404芯片使用塑料DIP封装，处于50℃的热盒子中，功耗为20mW，则节温度升为50℃＋0.02W×80℃/W=51.6℃。在74系列芯片中内部功耗问题并不重要，但是在对现代功耗超过10W以上的芯片却非常重要。

推荐操作条件（recommended operating conditions）定义了芯片使用的正确环境。在此工作条件下，芯片将满足规范要求。这些条件较绝对最大条件更为严格。例如，工作电压应处于2V～6V之间。对HC逻辑族，输入逻辑电平依赖于V_{DD}。当V_{DD}=5V时，应使用4.5V输入以允许系统中噪声产生的10%的电源电压降。

图A-9 给出了数据手册的第3页。电气特征（electrical characteristics）描述了在推荐操作条件下输入恒定时器件的操作。例如，如果V_{CC} = 5V（也可降为4.5V），在I_{OH} = 4.4V或I_{OL} = 0.1V时，输出电流I_{OH}/I_{OL}均不超过20μA。如果输出电流增加，芯片上的晶体管将尽力产生电流，使得输出电压将不处于理想工作状态。HC逻辑族使用的CMOS晶体管将产生非常小的电流。每个输入的电流将小于1000nA，而且在室温情况下典型值仅为0.1nA 。在芯片处于空闲状态下，电源静态（quiescent）电流（I_{DD}）不超过20μA。每个输入的电容小于10pF。

交流特征（switching characteristics）定义了器件在推荐操作条件下当输入发生变化时的工作情况。传输延迟（propagation delay）t_{pd}为了输入为0.5V_{CC}到输出变为0.5V_{CC}之间的时间间隔。如果V_{CC}一般为5V，芯片驱动电容不超过50pF，传播延迟将不超过24ns（而典型情况要更快一些）。前面提到每个输入的电容为10pF，因此在全速工作情况下输出不应该连接超过5个同样的此类芯片，而且芯片连线上的分布电容也将进一步减少可用负载。转换时间（transition time）也称为上升/下降时间，为输出在0.1V_{CC}到0.9V_{CC}之间的时间间隔。

1.8节中曾经介绍过芯片功耗分为静态(static)和动态（dynamic）两个部分。HC电路的静态功耗很低。在85℃时，最大静态电流为20μA。在工作电压5V时，其功耗为0.1mW。动态功耗取决于驱动电容和工作频率。7404中每个非门的内部功耗电容为20pF。如果7404的所

有6个非门都工作在10MHz，而且驱动负载电容均为20pF，则由式1.4可以计算动态功耗为 $\frac{1}{2}(6)(20\text{pF}+25\text{pF})(5^2)(10\text{MHz})=33.75\text{mW}$，而最大功耗为33.85mW。

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC} ..... –0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1) ..... ±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1) ..... ±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC}) ..... ±25 mA
Continuous current through V_{CC} or GND ..... ±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package ..... 86° C/W
N package ..... 80° C/W
NS package ..... 76° C/W
PW package ..... 131° C/W
Storage temperature range, T_{stg} ..... –65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

图A-8 7404数据手册第2页

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A=25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	V_I=V_{IH} or V_{IL}	I_{OH}=–20 μA	2V	1.9	1.998		1.9		1.9		V
			4.5V	4.4	4.499		4.4		4.4		
			6V	5.9	5.999		5.9		5.9		
		I_{OH}=–4 mA	4.5V	3.98	4.3		3.7		3.84		
		I_{OH}=–5.2 mA	6V	5.48	5.8		5.2		5.34		
V_{OL}	V_I=V_{IH} or V_{IL}	I_{OL}=20 μA	2V		0.002	0.1		0.1		0.1	V
			4.5V		0.001	0.1		0.1		0.1	
			6V		0.001	0.1		0.1		0.1	
		I_{OL}=4 mA	4.5V		0.17	0.26		0.4		0.33	
		I_{OL}=5.2 mA	6V		0.15	0.26		0.4		0.33	
I_I	V_I=V_{CC} or 0		6V			±0.1		±100		±1000	nA
I_{CC}	V_I=V_{CC} or 0,	I_O=0	6V			2		40		20	μA
C_i			2V to 6V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, C_L = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A=25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2V		45	95		145		120	ns
			4.5V		9	19		29		24	
			6V		8	16		25		20	
t_t		Y	2V		38	75		110		95	ns
			4.5V		8	15		22		19	
			6V		6	13		19		16	

operating characteristics, T_A = 25° C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

图A-9 7404数据手册第3页

A.6 逻辑族

74系列芯片可以使用不同的技术制造。这些技术称为逻辑族（logic family），可以提供不同的速度、功耗和逻辑电平。其他芯片往往设计为兼容这些逻辑族。最初的芯片（如7404）由双极型晶体管制造，称为晶体管–晶体管逻辑（Transistor-Transistor Logic, TTL）。在74系列后面增加一个或两个字母来表示更新的技术。表A-2汇总了常见的5V逻辑族。

表A-2 典型5V逻辑族的规范

	双极型/TTL						CMOS		CMOS/TTL兼容	
特征值	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd}(ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH}(V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL}(V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH}(V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL}(V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH}(mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL}(mA)	16	20	8	20	8	20	4	8	4	8
I_{IL}(mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH}(mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD}(mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{pd}(pF)	n/a						20	12	20	14
花费*(US $)	废除	0.57	0.29	0.53	0.33	0.20	0.15	0.15	0.15	0.15

*片批号的单价1000，来自2006年Texas Instruments 7408。

双极型电路和工艺技术产生肖特基（Schottky，S）和低功耗肖特基（LS）逻辑族。它们较TTL逻辑族更快。肖特基类型需要更多功耗，而低功耗肖特基的功耗则较低。高级肖特基（AS）和高级低功耗肖特积（LAS）较S和LS类型在速度和功耗方面均有改进。快速F系列速度更快，而功耗较AS低。所有这些逻辑族在输出低时的电流要超过输出高时，因此为非对称逻辑电平。它们都支持TTL逻辑电平：$V_{IH}=2V$，$V_{IL}=0.8V$，$V_{OH}>2.4V$，$V_{OL}<0.5V$。

在1980～1990年代，CMOS工艺已经成熟起来，并因其低功耗和输入电流特性而成为最为主流的工艺。高速CMOS（High speed CMOS，HC）和高级高速CMOS（Advanced High speed CMOS，AHC）逻辑族的静态功耗几乎为0。而且在高电平和低电平时输出同样大小的电流。它们符合CMOS逻辑电平：$V_{IH}=3.15V$，$V_{IL}=1.35V$，$V_{OH}>3.8V$，$V_{OL}<0.44V$。这些逻辑电平和TTL电路不兼容，主要是TTL的高电平输出2.4V不是CMOS合法的高电平输入。这就出现了高速TTL兼容CMOS（High-speed TTL-compatible CMOS，HCT）和高级高速TTL兼容CMOS（Advanced High-speed TTL-compatible CMOS，AHCT）逻辑族。这两个逻辑族接收TTL输入逻辑电平且产生有效的CMOS输出逻辑电平。它们较纯CMOS组件要慢一点。所有CMOS器件都对静电放电（electrostatic discharge，ESD）敏感。在接触CMOS芯片前，应首先接触大块的金属以向地释放自身的静电，防止击穿这些CMOS芯片。

74系列逻辑价格比较低廉。新的逻辑族价格往往比老的还要便宜。LS逻辑族应用最为广泛和可靠，在没有特殊性能要求的情况下是实验室和一般爱好者项目的首选。

5V标准的应用在1990年代中期就逐渐减少了。这是因为晶体管变得非常小从而难以支撑这个电压。而且更低的电压可以降低功耗。目前，3.3V，2.5V，1.8V，1.2V，甚至更低的电压得到了广泛的使用。过多的电压标准使得不同电源供电芯片之间的通信更加困难。表A-3列举了一些低电压逻辑族。并非所有74系列芯片都支持这些逻辑族。

所有的低电压逻辑族都使用现代集成电路的基础——CMOS晶体管。这些逻辑族工作电压V_{DD}很宽广，但是在电压较低的时候工作速度也将下降。低电压CMOS（Low-Voltage CMOS，LVC）和高级低电压CMOS（Advanced Low-Voltage CMOS，ALVC）逻辑往往工作在3.3V，2.5V或1.8V。LVC可以支持的最高输入电压为5.5V，因此它可以接收来自5V CMOS或

TTL电路的输入。高级超低电压CMOS（Advanced Ultra-Low-Voltage CMOS, AUC）常工作在2.5V, 1.8V和1.2V，而且速度特别快。ALVC和AUC逻辑族支持的最高输入电压为3.6V，因此可以接收3.3V电路的输入。

FPGA的内部逻辑（称为core）和外部IO引脚往往需要独立的电压输入。当FPGA不断进步时，内部逻辑电压从5V逐步降到了3.3V, 2.5V, 1.8V和1.2V，以减少功耗和避免对小晶体管的损坏。FPGA的IO引脚电压可以配置为不同逻辑族，从而可以和系统中其他部分兼容。

表A-3 典型的低电压逻辑族规范

	LVC			ALVC			AUC		
V_{dd}(V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd}(ns)	4.1	6.9	9.8	2.8	3	?[1]	1.8	2.3	3.4
V_{IH}(V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL}(V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH}(V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL}(V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O(mA)	24	8	4	24	12	12	9	8	3
I_I(mA)		0.02			0.005			0.005	
I_{DD}(mA)		0.01			0.01			0.01	
C_{pd}(pF)	10	9.8	7	27.5	23	?*	17	14	14
花费(US $)		0.17			0.20			不适用	

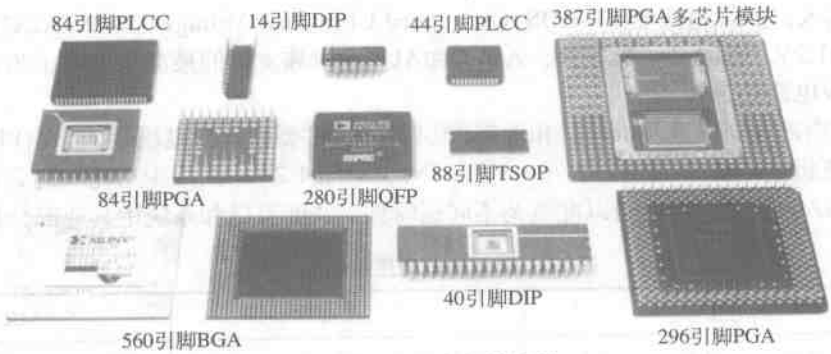
*延迟和容量不适用于写的情况。

A.7 封装和装配

集成电路往往被放置在塑料或陶瓷的封装（package）内。封装的主要作用包括：将芯片外围很小的金属IO引脚引出到封装上较大的引脚上便于芯片间连接；防止芯片受到物理损坏；将芯片产生的热扩散到封装更大的面积上以便冷却。这些封装好的芯片放置到面包板或印刷电路板上并一起组成系统。

封装

图A-10给出了不同集成电路的封装。封装可以分为通孔（through-hole）和表面贴装（surface mount, SMT）两类。顾名思义，通孔类封装的引脚可以插到印刷电路板上的通孔或芯片插孔中。双列直插（Dual inline packages, DIP）有两列引脚，每个引脚的间距为0.1英寸。引脚格状阵列（Pin grid arrays, PGA）将引脚放置在封装的下方，使得在一个小封装中可以容纳更多的引脚。SMT封装是直接将芯片焊在印刷电路板上而不用通孔。SMT封装的引脚称为导线（lead）。TSOP（thin small outline packages）封装有两列排列紧密的导线（间距仅为0.02英寸）。PLCC（plastic leaded chip carriers）封装在四周有J形的导线（间距为0.05英寸）。它们可以直接固定在印刷电路板的特殊插座上。QFP（quad flat packs）封装采用在四周有大量紧密排列的引脚。BGA（ball grid array）封装则完全消除了引脚，取而代之的是芯片下部数以百计的小焊球。在焊装时，需要将它与印刷电路板上的焊盘精确对准，然后加热焊料，焊料融化后就将芯片和底下的电路板连接在一起。

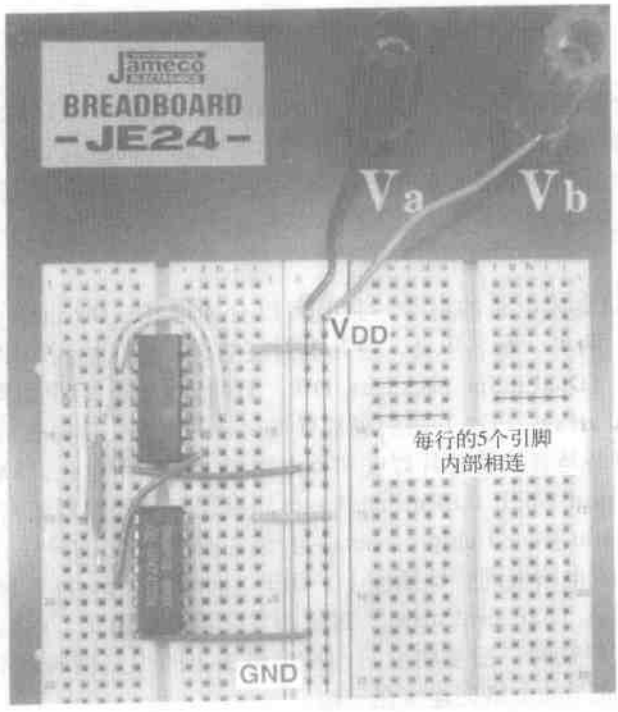


图A-10 集成电路的封装

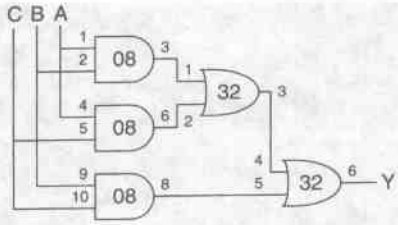
面包板

DIP封装很容易构成原型系统，因为它们可以放置在面包板上。面包板是一个包含了很多行插孔的塑料板，如图A-11所示。每行有五个连通的插孔。封装的每个引脚放置在单独行的一个插孔中。引线可以插在同一行的插孔中从而和芯片的引脚相连。面包板往往在板的边缘提供单独连接插孔的列以放置电源和地线。

图A-11中显示了包含74LS08与门和74LS32或门的面包板。其电路原理图如图A-12所示。电路原理图中每个门都标识了芯片（08或32）和其输入、输出引脚（参见图A-1）。注意此连接在面包板上的接法。输入连接到08芯片的第1、2、5脚，输出连接到32芯片的第6脚。电源和地分别通过垂直的电源和地线将香蕉形插孔上的Va和Vb连接到每个芯片的14和7脚。原理图一般按照这种方式标识，并在检查连接时打上勾。这对减少构造面包板电路时的错误将是很好的一种方法。



图A-11 面包板的主要电路



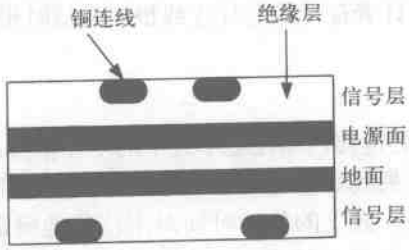
图A-12 包括了芯片和引脚标识的主要门电路原理图

不幸的是，往往很容易将线插错孔或有线飞出孔，因此需要非常小心（或在实验室中需要一些调试）来构造面包板电路。面包板仅仅适用于构造原型系统，而不能大批量生产。

印刷电路板

与面包板不同的是，芯片的封装可以直接焊在印刷电路板（printed circuit board, PCB）上。印刷电路板由很多层导电铜和绝缘的环氧树脂构成。铜被刻蚀成导线，称为连线(trace)。通孔（via）是在电路板上钻的孔，并覆盖了金属从而连接了不同层。PCB板往往通过计算机辅助设计（computer-aided design, CAD）工具设计。你可以在实验室里自己刻蚀或钻孔来生产简单的电路板，也可以将电路板设计送到专业厂家进行比较廉价的大规模生产。工厂的生产周期一般为几天（或几周，针对于廉价的大规模生产），需要数百美元的初始费用和少量的每块板生产费用。对于大批量中等复杂的板，每块板的生产费用仅为几美元。

印刷电路板的连线一般由电阻较低的铜制成。这些连线被封装在绝缘材料中。常见的绝缘材料是绿色的阻燃塑料FR4。印刷电路板通常在信号层之间有铜制的电源层和地层，称为面(plane)。图A-13显示了一个印刷电路板的剖面图。信号层在板的顶部和底部，电源和地面嵌入在板的中间。电源和地面的电阻很低，可以为板上的组件分配稳定的电源。它们也使得连线的电容和电导一致和可预测。

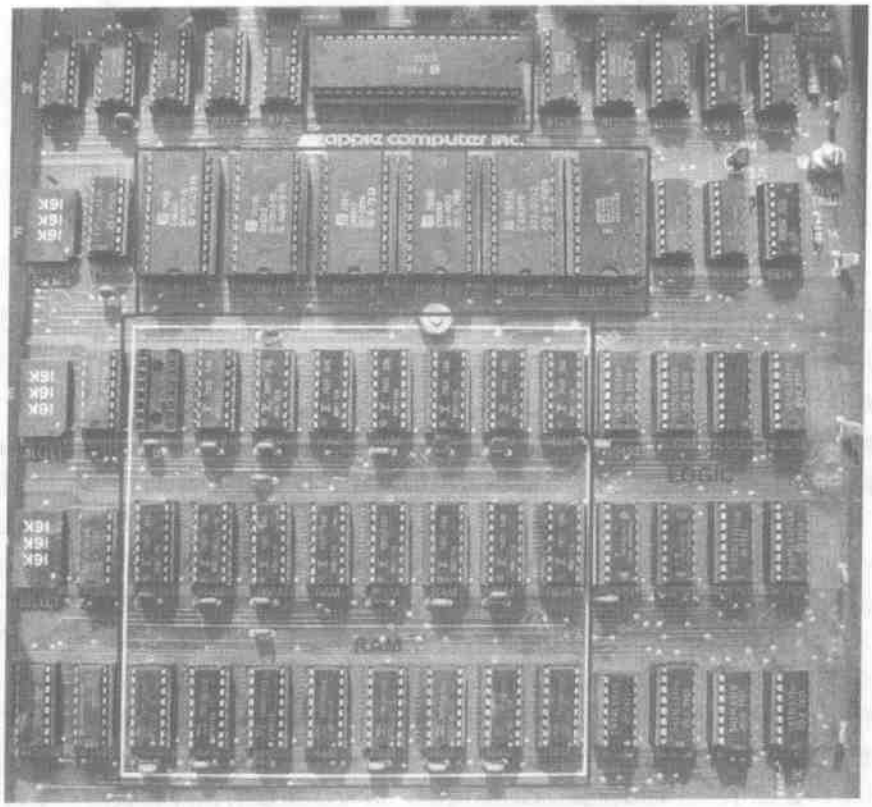


图A-13 印刷电路板剖面图

图A-14给出了1970年代最成功的Apple Ⅱ＋型计算机的印刷电路板。照片上方为6502微处理器，下方为6片16Kb的ROM芯片，构成了存储器操作系统的12KB的ROM系统。三行（每行8片）的16Kb的DRAM芯片提供了48KB的RAM。右边是用以存储器地址译码和其他功能的一些74系列芯片。芯片之间的连线将各个芯片连接起来。一些连线的点状终点是覆盖了金属的通孔。

小结

大部分现代芯片有大量的输入和输出引脚，往往采用SMT封装，其中QFP和BGA最为常用。这些封装需要使用印刷电路板，而不能使用面包板。使用BGA封装时更为困难，因为它需要特殊的焊装设备。而且在实验室的调试中，这些焊球因为隐藏在封装下面而不能通过电压表或示波器测量。



图A-14 Apple Ⅱ+型电路板

总之，系统设计者需要在早期考虑封装问题，以决定在原型期间是否使用面包板，并且是否使用BGA封装。专业的设计者在对其芯片连线很有自信时很少使用面包板进行实验。

A.8 传输线

我们一直假设线是等电势的连接，即在整个线中的电压全部相等。信号在线上实际上是按光速以电磁波的方式传播。如果线足够短或者信号变化很慢，等电势假设是正确的。但是当线长度较长或者信号变化很快，线上的传输时间对于精确地确定电路延迟非常重要。我们必须为传输线（transmission line）建立模型，其电压和电流按光速以波的形式传播速度。当波传输到线的终点时，可能会沿着线产生反射。如果不加以抑制，这种反射将产生噪声和不规则特征。因此，数字电路设计者必须考虑到传输线特征以精确计算长线中的延迟和噪声影响。

电磁波在给定介质中按照光速传播，虽然速度很快但也需要时间。光的传播速度取决于介质的介电常数（permittivity）ε和磁导率（permeablility）μ：$v = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$。真空中的光速为$v = c = 3 \times 10^8$m/s。因为FR4绝缘层的磁导率4倍于真空，PCB板中的信号传输速度大约为光速的一半。因此，PCB板的信号传输速度大约为1.5×10^8m/s（合15cm/ns）。长度为l的传输线，其信号延迟为

$$t_d = l/v \tag{A-4}$$

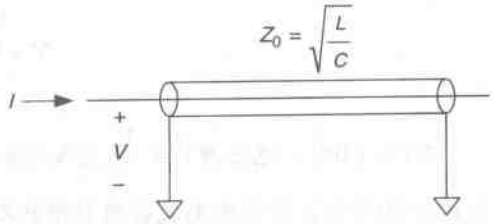
传输线特征阻抗（characteristic impedance）Z_0为传输线中电压和电流之比：$Z_0 = V/I$。但

是注意，波阻抗不是电阻（数字系统传送线的电阻可以忽略不计）。它依赖于传输线的电感和电容（参见A.8.7的推导），典型值为50Ω～75Ω。

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A-5}$$

图A-15给出了传输线符号。这个符号就像有线电视的同轴电缆一样，内部为信号传输线，外部为接地的屏蔽导体层。

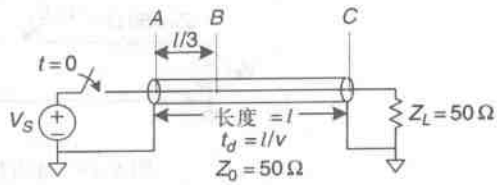
理解传输线特征的关键在于将信号传输想像为电压波沿着传输线以光速传播。当波达到终点时，根据终端负载可能会产生反射或被吸收。反射波将沿传输线向后传播，并和线上已有电压叠加。终端可以分为匹配、开路、短路和失配等情况。下面各节将分析波是如何在线上传输，以及传输到终端时的情况。



图A-15 传输线符号

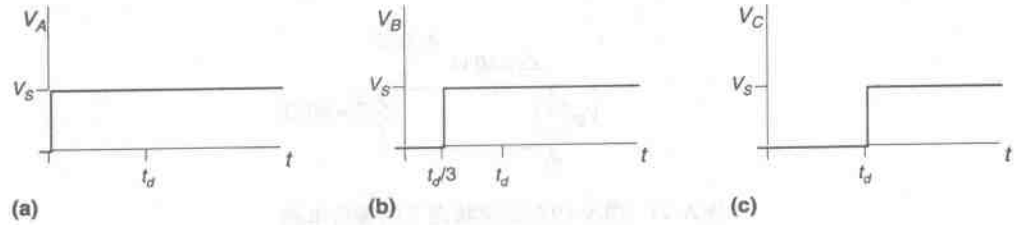
A.8.1 匹配终端

图A-16中长度为l的传输线的终端匹配，即终端阻抗Z_L等于传输线波阻抗Z_0（50Ω）。传输线一端是开关电源，在t=0时闭合，另一端是匹配的50Ω终端负载。本节将分析处于线起始点A，三分之一长度处B和终端C等三个位置的电压和电流。



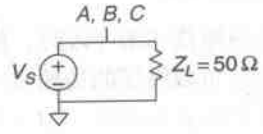
图A-16 终端匹配的传输线

图A-17给出了A，B，C三点电压随时间的变化。初始时开关未闭合，传输线中的电压和电流都为0。当t=0时，开关闭合，电源的电压升为V=V_S。由于波阻抗为Z_0，所以电流为$I = V_S/Z_0$。电压立即到达线的起始点A，如图A-17(a)所示。波沿着传输线以光速传播，在t_d/3时到达B点。此点的电压从0跳变到V_S，如图A-17(b)所示。在t_d时，跳变波到达线终点C，电压也随之上升。流经负载Z_L的电流所产生的电压为$Z_L I = Z_L(V_S/Z_0) = V_S$，这是因为终端阻抗$Z_L$等于特征阻抗$Z_0$。电压$V_S$随着波在传输线上稳定传播。因此，波被负载阻抗所吸收，传输线达到稳定状态。



图A-17 图A-16中A，B，C点的电压波形

在稳定状态下，传输线行为类似于一个理想等电势线。线上所有点的电压都相同。图A-18显示了电路图A-16中稳定状态的等价模型。线中所有位置的电压都是V_S。



图A-18 图A-16在稳定状态下的等价电路

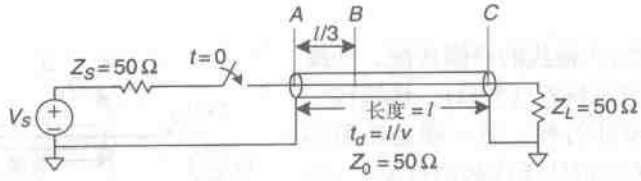
例A.2 具有匹配源和负载终端的传输线

图A-19中的传输线具有匹配的源阻抗Z_S和负载阻抗Z_L。请画出A，B，C点关于时间的电压变化图。何时系统达到稳定状态，稳定状态的的等价电路是什么？

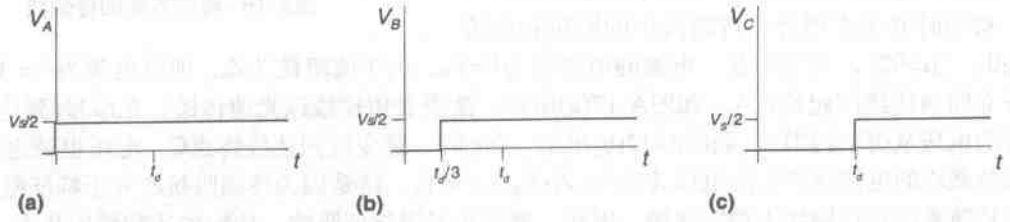
解：由于电压源有源阻抗Z_S与传输线串联，在Z_S上将产生电压降，其余的电压将沿传输线传播。首先，传输线的特征为阻抗Z_0。这是因为波还没有按光速到达传输线终端前，其负载并不会对传输线行为产生影响。因此，根据分压方程，沿着传输线传输的跳变电压为

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{A-6}$$

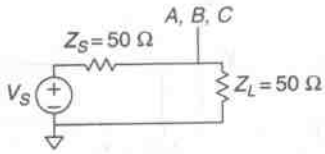
在t等于0时，电压波$V = \frac{V_S}{2}$从A点沿传输线传播。在$t_d/3$时刻达到B点，在t_d时刻达到C点，如图A-20所示。所有的电流被负载阻抗Z_L吸收，因此电路在$t = t_d$时稳定。稳定状态下，整个线的电压均为$V_S/2$，其稳定状态等价电路如图A-21所示。



图A-19 具有匹配源和负载阻抗的传输线



图A-20 图A-19中A，B，C点的电压波形



图A-21 图A-19在稳定状态下的等价电路

A.8.2 开路终端

当负载阻抗不等于Z_0时，终端将不能吸收所有的电流，部分波将被反射。图A-22给出了一个具有开路负载终端的传输线。由于终端为开路，必无电流流过，所以C点的电流必然恒为0。

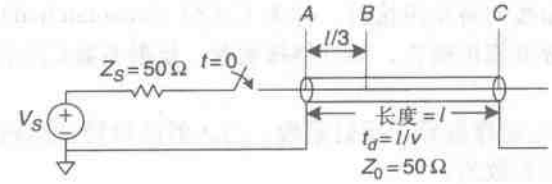
线上电压初始为0。在$t = 0$时刻，开关闭合，电压波开始沿传输线传播，其电压为$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2}$。初始波与例A.2完全相同，与终端无关，这是因为在t_d时刻之前终端的负载

不会影响到传输线特征。该电压波在$t_d/3$时刻达到B点，在t_d时刻到达C点。

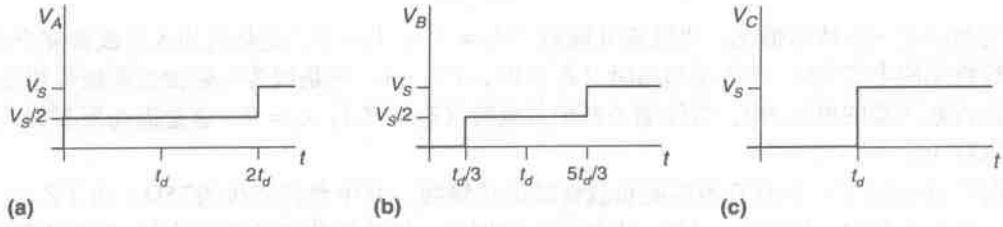
当电压波达到C点时，由于终端开路，所以无法向前传播，导致所有波被反射回源。所以反射电压依然为 $V=\frac{V_S}{2}$ 。

线上任意点电压等于初始波和反射波之和。在$t = t_d$时刻，C点的电压为 $V=\frac{V_S}{2}+\frac{V_S}{2}=V_S$。

反射波在$5t_d/3$时刻达到B点，在$2t_d$时刻到达A点。当到达A点时，由于源终端阻抗与传输线特征阻抗匹配，所以反射波被完全吸收。因此系统在$2t_d$时刻达到稳定，且传输线成为等电势线，其电压为V_S，电流为0。



图A-22 开路负载终端的传输线

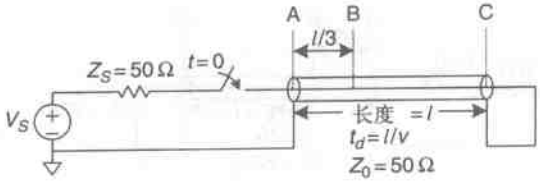


图A-23 图A-22中A，B，C点的电压波形

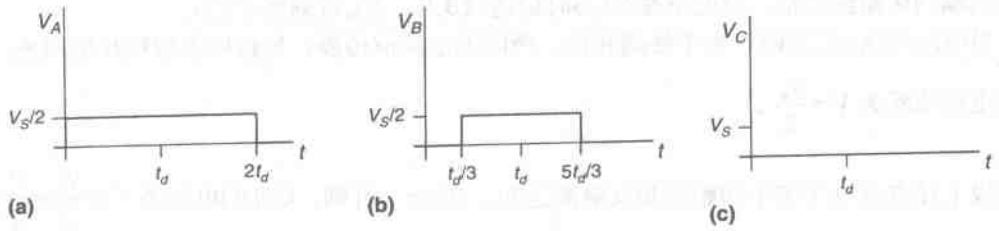
A.8.3 短路终端

图A-24给出另一个终端为短路连接到地的传输线。因此C点的电压必然为0。

与前面的例子相同，初始时线上的电压为0。当开关闭合后，电压波（$V=\frac{V_S}{2}$）沿线传播。当到达终端时，将按照相反极性反射。发射波的电压为 $V=-\frac{V_S}{2}$ ，并叠加到初始波上，以保证C点电压为0。反射波在$t = 2t_d$时刻达到源，并被源阻抗吸收。此时，系统达到稳定状态，传输线的电压等价为等电势线，且电压为0。



图A-24 短路终端的传输线



图A-25 图A-24中A，B，C点的电压波形

A.8.4 不匹配终端

当终端阻抗不等于传输线的特征阻抗时，称为不匹配（mismatched）。一般而言，当跳变波达到不匹配终端时，部分波将被吸收，部分将被发射。反射系数k_r表示了入射波中被发射的比例，即$V_r = k_r V_i$。

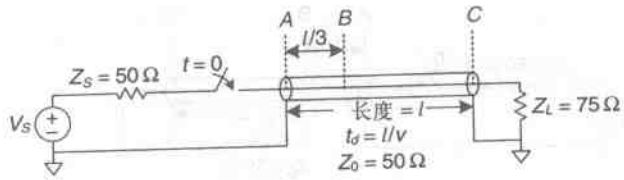
A.8.8节中利用电流恒定定律推导了反射系数。当入射波沿特征阻抗为Z_0的传输线达到负载阻抗为Z_T的终端时，反射系数为：

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0} \tag{A-7}$$

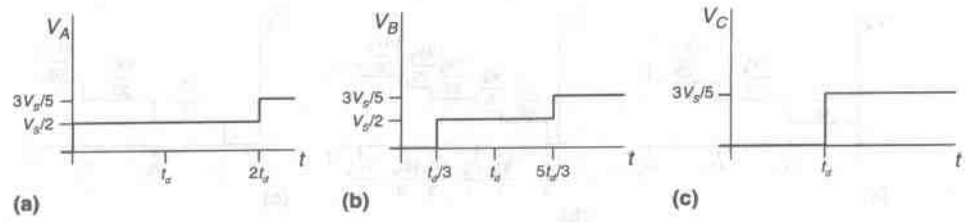
需要注意一些特殊情况。当终端开路时（$Z_T = \infty$），$k_r = 1$，这是因为入射波被完全反射，使得线终端的电流为0。当终端短路时（$Z_T = 0$），$k_r = -1$，这是因为入射波按照极性相反被反射，使得线终端的电压为0。当终端为匹配负载时（$Z_T = Z_0$），$k_r = 0$，这是因为所有的入射波都被吸收了。

图A-26给出了一个具有不匹配负载终端的传输线，其中负载阻抗为75Ω。由于$Z_T = Z_L = 75\Omega$，且$Z_0 = 50\Omega$，因此$k_r = 1/5$。正如前面的例子，传输线的初始电压为0。当开关闭合后，电压波（$V = V_S/2$）沿传输线传播，在$t = t_d$时刻达到终点。当入射波达到线终端时，五分之一的波被反射，剩下五分之四的波被终端负载吸收。因此发射电压为 $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$。C点的电压为入射和反射电压之和，$V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$，在$t = 2t_d$时刻，反射电压达到A点，并被50Ω的源阻抗$Z_S$吸收。图A-27画出了线上的电压和电流波形。注意，进入稳定状态后（$t > 2t_d$），传输线等价为等电势线，如图A-28所示。在稳定状态下，系统类似一个分压器，满足：

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

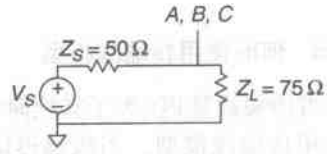


图A-26 不匹配终端的传输线



图A-27 图A-26中A，B，C点的电压波形

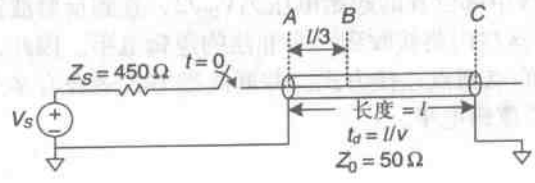
反射可以发生在传输线的任意一端。图A-29中传输线的源阻抗Z_S为450Ω，负载端为开路。负载端发射系数k_{rL}和源端反射系数k_{rS}分别为1和4/5。此时，波将在传输线两端反射直至到达稳定状态。



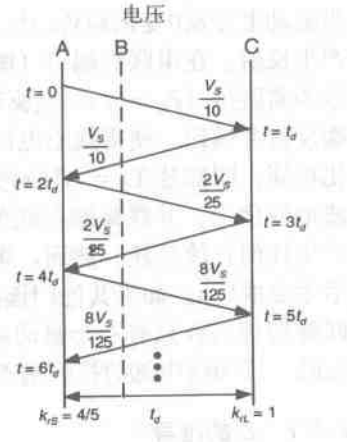
图A-28 图A-26在稳定状态下的等价电路

图A-30中的反射图显示了在传输线两端的反射过程。水平轴代表了沿传输线的距离，垂直轴代表向下递增的时间。反射图的两端代表了传输线的源端A和负载端C。入射和反射信号波以A和C之间的箭头线表示。在t = 0时刻，源阻抗和传输线类似分压器，产生一个电压为$\frac{V_S}{10}$的波从A点向C点传播。在$t = t_d$时刻，信号达到C点，且被完全反射（$k_{rL} = 1$）。在$t = 2t_d$时刻，电压为$\frac{V_S}{10}$的反射波达到A点，并以反射系数$k_{rS} = 4/5$反射，并产生电压为$\frac{2V_S}{25}$的反射波向C点传播。后续过程依此类推。

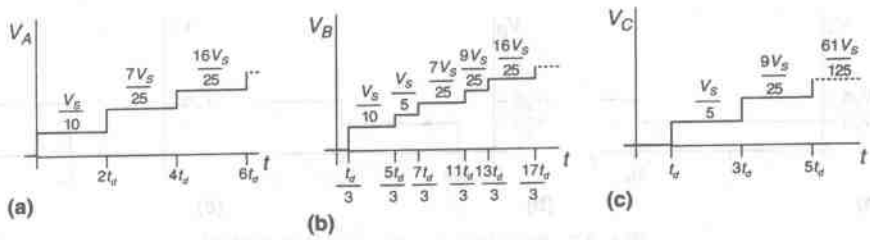
传输线上每点的电压是所有入射波和反射波之和。因此，在$t = 1.1t_d$时刻，C点电压为$\frac{V_S}{10}+\frac{V_S}{10}=\frac{V_S}{5}$。在$t$=3.1$t_d$时刻，C点电压为$\frac{V_S}{10}+\frac{V_S}{10}+\frac{2V_S}{25}+\frac{2V_S}{25}=\frac{9V_S}{25}$，后续过程依此类推。图A-31给出了电压和时间的关系。当t趋近于无穷大时，电压趋近于稳定状态，即$V_A = V_B = V_C = V_S$。



图A-29 不匹配源阻抗和终端的传输线（图中C点似乎有错！应该为开路，而不是接地）



图A-30 图A-29的反射图



图A-31 图A-29的电压和电流波形

A.8.5 何时使用传输线模型

当传输线延迟t_d大于转换时间（信号上升或下降时间）的某个比例（典型为20%）时就需要使用传输线模型。当线延迟比较小时，信号传播延迟的影响非常微弱，反射在信号变化过程中很快消散。如果传输延迟比较大，必须按照延迟线模型考虑以精确预测波延迟和信号的波形。特别需要指出的是，反射可能会破坏数字特征的波形，从而导致错误逻辑操作。

如前所述，信号在印刷电路板上传输延迟为15cm/ns。对于TTL电平，较换时间10ns。当线长度大于30cm时（10ns × 15cm/ns × 20%），必须作为传输线看待。印刷电路板的线长度一般都小于30cm，因此大多数可以作为理想等电势线。但是，对于现代很多芯片的较换时间率为2ns甚至更小，此时长度大于6cm（约2.5英寸）的线都需要作为传输线看待。

由于面包板缺少地线层，所以每个信号的电磁场不规则而且难以建模。而且信号之间的电磁场相互作用，这使得信号之间的反射和串扰更加强烈。因此当工作频率超过几兆赫兹时，面包板就不太可靠。

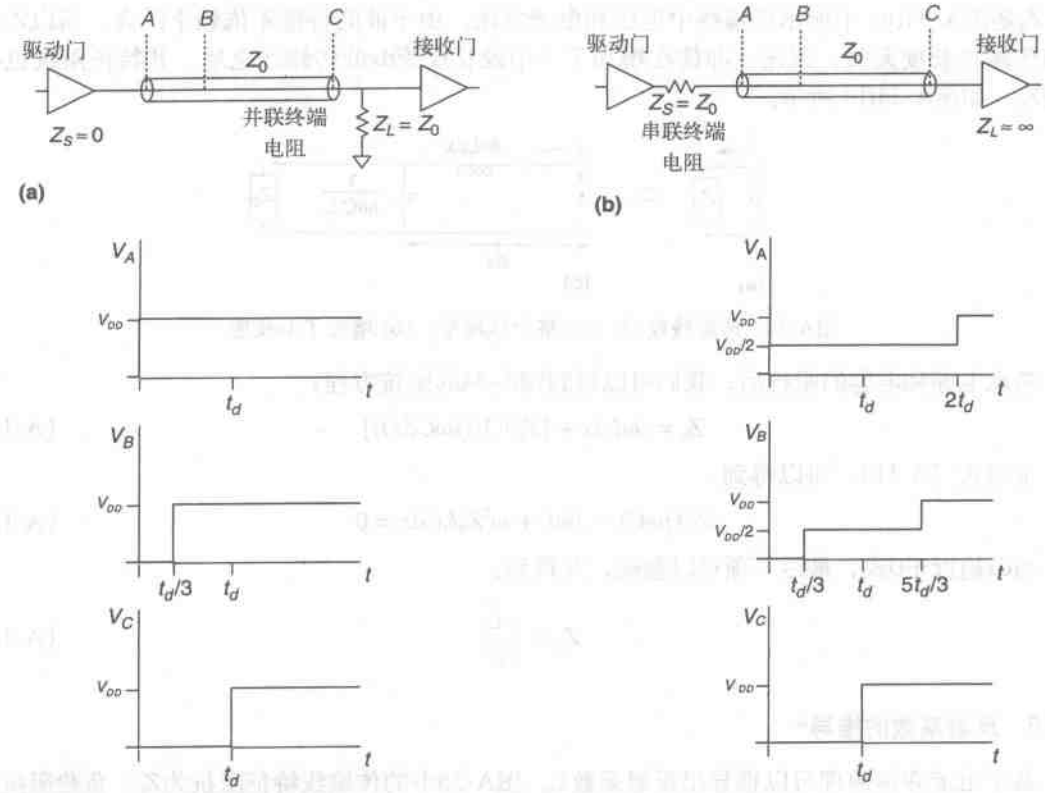
相反，印刷电路板在整个传输线上有比较稳定的特征阻抗和传输速度。只要源阻抗或负载阻抗与传输线的特征阻抗匹配，印刷电路板上就不会受反射影响。

A.8.6 正确的传输线终端

图A-32给出了两种常用的传输线终端连接方法。在并联终端（图A-32(a)），驱动门的阻抗很低（$Z_S \approx 0$）。负载电阻（Z_L）与传输线特征阻抗Z_0并联在负载端（在接收门和地之间）。当驱动电压从0变化到V_{DD}时，它将沿线传播电压为V_{DD}的波，并被匹配负载终端所吸收，而不产生反射。在串联终端中（图A-32(b)），源电阻（Z_S）与驱动端串联使得源阻抗为Z_0。此时负载为高阻抗（$Z_L = \infty$）。当驱动端电压变化时，将产生电压为$V_{DD}/2$的波沿线传播，并在开路终端反射并返回，使得线上电压升为V_{DD}。反射波在源端被吸收。这两个电路在接收端的电压变化相同，即都是在$t = t_d$时刻变为V_{DD}，这也正是我们所希望的。它们的区别在于功耗和沿线的波形变化上。并联终端在线为高电平时持续消耗电能。因为负载连接在开路上，串联终端不产生任何直流功耗。然而，串联终端的传输线中部位置的起始电压为$V_{DD}/2$，直到反射波返回后才变成V_{DD}。如果其他门连接在线的中部，这些门将接收到一个非法的逻辑电平。因此，串联终端最适合只有一个驱动端和一个接收端的点到点连接方式。并联终端方式适合有多个接收端，因为线中部的接收端不会接收到非法的逻辑电平。

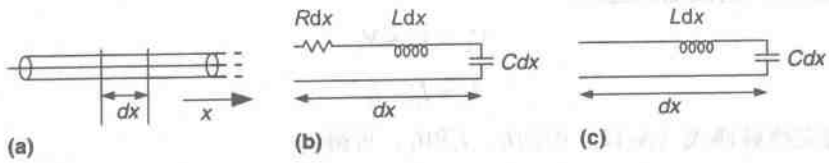
A.8.7 Z_0的推导*

Z_0是波沿着传输线传播过程中电压与电流之比。本节将推导Z_0，这需要有关于RLC（Resistor-inductor-capacitor, 电阻-电导-电容）电路分析的前导知识。



图A-32 终端连接方式：(a) 并联；(b) 串联

考虑当跳变电压输入到半无限传输线的一端（此时无反射）。图A-33给出了半无限传输线和对dx长度的一段传输线建立的模型。R、L、C分别是单位长度的电阻、电导和电容。图A-33(b)给出的是有电阻R的传输线模型。因为在传输过程中有能量消耗在电阻上，这种模型为有损（lossy）传输线模型。然而，这种损耗往往可以忽略不计，从而忽略电阻元件而简化分析。这种模型称为理想传输线，如图A-33(c) 所示。



图A-33 传输线模型：(a) 半无限长传输线；(b) 有损传输线；(c) 理想传输线

经过传输线的电流和电压是空间和时间的函数，如式（A-8）和式（A-9）式所示：

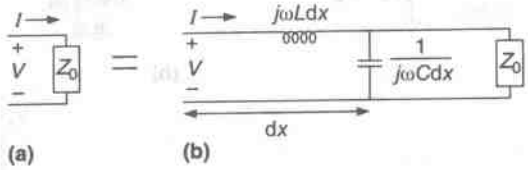
$$\frac{\partial}{\partial x}V(x,t)=L\frac{\partial}{\partial t}I(x,t) \tag{A-8}$$

$$\frac{\partial}{\partial x}I(x,t)=C\frac{\partial}{\partial t}V(x,t) \tag{A-9}$$

对式（A-8）求对空间的微分，对式（A-9）求对时间的微分，并带入就可以得到波方程式（A-10）：

$$\frac{\partial^2}{\partial x^2}V(x,t)=LC\frac{\partial^2}{\partial t^2}V(x,t) \tag{A-10}$$

Z_0是图A-34(a) 中所示传输线中电压和电流之比。由于波的特征不依赖于距离，所以Z_0必然与传输线长度无关。因此，即使在增加了一小段长度为dx的传输线之后，其特征阻抗也必然是Z_0，如图A-34(b) 所示。



图A-34 传输线模型：(a) 整个线模型；(b) 增加了dx长度

考虑电导和电容的阻抗后，我们可以得到图A-34的阻抗方程：

$$Z_0 = j\omega L\mathrm{d}x + [Z_0 \| (1/(j\omega C\mathrm{d}x))] \tag{A-11}$$

重写式（A-11），可以得到：

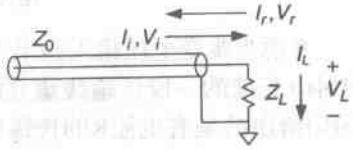
$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 LC\mathrm{d}x = 0 \tag{A-12}$$

当dx趋近于0式，最后一项可以忽略，并得到：

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A-13}$$

A.8.8 反射系数的推导*

基于电流守恒原理可以推导出反射系数k_r。图A-35中的传输线特征阻抗为Z_0，负载阻抗为Z_L。假设入射波电压为V_i，电流为I_i。当波达到终点时，部分电流I_L，经过负载阻抗，产生电压降V_L。其余电流被反射回传输线，其电压为V_r，电流为I_r。Z_0为传输线中波的电压和电流之比，因此有$\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$。



图A-35 传输线的入射、反射和负载的电压和电流

传输线终端的电压是入射波和反射波电压之和，正向电流是入射波电流和反射波电流之差。

$$V_L = V_i + V_r \tag{A-14}$$

$$I_L = I_i - I_r \tag{A-15}$$

使用欧姆定律替换式（A-15）中的I_L、I_r和I_L，可得：

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A-16}$$

可从此式得到反射系数k_r：

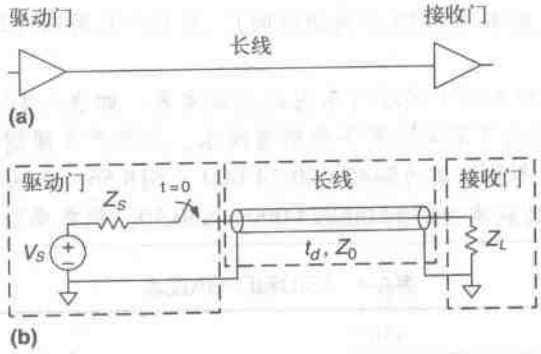
$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{A-17}$$

A.8.9 总结

由于光速为有限，传输线模型刻画了信号沿线传输需要时间这一事实。理想的传输线具有一致的单位长度电感L，电容C，而且电阻为0。传输线的关键特征是其特征阻抗Z_0和传输延

迟t_d。这两个参数可以从单位长度电感、电容和线长度推导出来。当信号的上升/下降时间超过$5t_d$时，传输线对信号的延迟和噪声有明显影响。这意味着，当系统的上升/下降时间为2ns时，如果印刷电路板线长度超过6cm时就需要采用传输线模型进行分析，以得到其精确特征。

当数字系统中包含了如图A-36所示由逻辑门驱动，经过长线到第二级门的情况时，就可以使用传输线模型。第一个门作为包含了源阻抗（Z_S）的电压源在时刻0输出电压由0变化到1。驱动门不能提供无限大电流，这可以由Z_S建模。逻辑门的Z_S一般比较小，设计者可以串联一个电阻来增加Z_S，以匹配线阻抗。第二个门输入作为Z_L。CMOS电路的输入电流往往很小，因此Z_L可接近无限大。设计者可以选择在第二个门的输入和地之间并联一个电阻，使得Z_L匹配传输线的阻抗。



图A-36 数字系统传输线模型

当第一级门电压变化时，电压波被驱动到传输线上。源阻抗和传输线构成分压器，这使得入射波电压为：

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A-18}$$

在t_d时刻，波达到线的终端。部分波被负载吸收，部分被反射。反射系数k_r为反射比例：$k_r = V_r/V_i$，其中V_r为反射波电压，V_i为入射波电压。

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A-19}$$

反射波叠加到传输线上的已有电压，并在$2t_d$时刻到达源。此时，又有部分被吸收，部分被反射。反射过程持续进行，线上的电压最终将趋近于稳定，此时传输线成为等电势线。

A.9 经济问题

虽然数字电路设计非常有趣，很多人乐意无偿做，但是大多数设计者和公司希望从中赢利。因此，经济问题是设计决策中一个重要的考虑因素。

数字电路的花费可以分为不可回收费用（Nonrecurring Engineering Costs, NRE）和可回收费用（recurring costs）。NRE费用决定了系统设计费用，包括设计成员的报酬、计算机和软件费用以及生产第一版产品的费用。在2006年，美国一个设计者的所有费用（包括薪水、健康保险、退休保险、设计工具和计算机）大约为每年20万美元，因此设计成本比重非常高。可回收费用是每增加一个产品所需要的花费，这包含了元器件、制造、市场、技术支持和分发。

销售价格不仅包括了系统的成本，而且还包括了办公室场地租金、税、非直接设计人员

工资（例如janitor和CEO）等。在去除这些费用后，公司才能获得利润。

例A.3 BEN试图挣钱

Ben Btiddidle设计了一个巧妙的雨滴计数电路。他决定销售这些设备来赚钱，但是他需要决定使用何种实现方法。有两种方法可以选择：FPGA和ASIC。设计和测试FPGA的开发工具价格是1500美元，每个FPGA成本是17美元。制作ASIC掩膜的成本是60万美元，每片ASIC的生产成本是4美元。

无论选择何种芯片实现方式，Ben需要在印刷电路板上配装芯片。每个印刷电路板需要1.5美元。Ben希望每个月能销售1000套设备。Ben召集了一组本科学生设计这个芯片作为他们的毕业设计，因此不需要设计成本。

如果销售价格两倍于成本（100%的利润空间），而且产品周期为2年，那么哪种实现方式比较好？

解：Ben画了一个两种实现方式在两年内的总成本表，如表A-4所示。两年后，Ben将销售24000个设备，表A-4给出了不同选择下的所有成本。如果产品周期仅为2年，FPGA实现明显是更好的选择。每个设备的成本为$445 000/24 000 = $18.56，在100%利润空间下销售单价是37.13美元。ASIC实现的成本为$732 000/24 000 = $30.50，销售单价为61美元。

表A-4 ASIC和FPGA成本

成本	ASIC	FPGA
NRE	$600 000	$1500
单芯片成本	$4	$17
PCB	$1.50	$1.50
总计	$600 000 + (24 000 × $5.50) = $732 000	$1500 + (24 000 × $18.50) = $445 500
单价	$30.50	$18.56

例A.4 BEN更加贪婪

在看到产品的市场广告后，BEN觉得他可以比原先预期每个月销售更多的产品。如果他选择ASIC实现，每个月需要销售多少个产品才能使得ASIC实现比FPGA实现更加有利？

解：Ben通过下述方程求解两年内最小的销售数量N：

$$\$600\,000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

此方程结果为N = 46 039，即每个月1919个设备。他需要月销售量增加一倍才能从ASIC实现中获利。

例A.5 BEN没有这么贪婪

Ben意识到他的胃口太大了，不能假定每个月能买1000个设备。但是他认为产品周期可能会超过2年。如果设备的每月销量为1000个，产品周期需要多长才能使得ASIC实现更为合算？

解：如果Ben能总共销售46 039个设备，那么ASIC实现就是最佳选择。因此，Ben在每个月销售1000个设备时需要销售47个月（向上取整），大约4年时间。那个时候，他的产品将会被淘汰。

芯片往往都是从分销商处购买，而不是直接从制造商购买（除非你购买上万个芯片）。Digikey（www.digikey.com）是一个销售多种电子元件的优秀分销商。Jameco（www.jameco.com）和All electronics（www.allelectronics.com）也都有适合于爱好者且价格有竞争力的丰富产品目录。

附录B MIPS指令

本附录总结了本书使用的MIPS指令。表B1～表B3定义了每条指令的opcode和funct字段，并且对指令的功能做了简单的说明。使用以下符号：

- [reg]：寄存器内容
- Imm：I型指令中的16位立即数字段
- Addr：J型指令中的26位地址字段
- SignImm：立即数符号扩展
  = {{16{imm[15]}}, imm}
- ZeroImm：立即数0扩展
  = {16'b0, imm}
- Address：[rs] + SignImm
- [Address]：Address处的存储器内容
- BTA：分支目的地址
  = PC + 4 + (SignImm << 2)
- JTA：跳转目的地址
  = { (PC + 4)[31: 28], addr, 2'b0}

表B-1 按opcode字段排列的指令

opcode	名 称	描 述	操 作
000000(0)	R型	所有R型指令	见表B-2
000001(1)	bltz/bgez	小于0跳转/大于等于0跳转	如果([rs]<0) PC=BTA/
(rt = 0/1)			如果([rs]≥0)PC=BTA
000010(2)	j	跳转	PC=JTA
000011(3)	jal	跳转并连接	$ra=PC+4, PC=JTA
000100(4)	beq	如果相等则转移	如果([rs]==[rt]) PC=BTA
000101(5)	bne	如果不等则转移	如果([rs]!=[rt]) PC=BTA
000110(6)	blez	如果小于等于0则转移	如果([rs]≤0) PC=BTA
000111(7)	bgtz	如果大于0则转移	如果([rs]>0) PC=BTA
001000(8)	addi	立即数加法	[rt]=[rs]+SignImm
001001(9)	addiu	无符号立即数加法	[rt]=[rs]+SignImm
001010(10)	slti	根据立即数设置寄存器rt的值	[rs]<SignImm?[rt]=1 : [rt]=0
001011(11)	sltiu	根据无符号数设置寄存器rt的值	[rs]<SignImm?[rt]=1 : [rt]=0
001100(12)	andi	和立即数进行与运算	[rt]=[rs] & ZeroImm
001101(13)	ori	和立即数进行或运算	[rt]=[rs] | ZeroImm
001110(14)	xori	和立即数进行异或运算	[rt]=[rs] ^ ZeroImm
001111(15)	lui	载入立即数的高16位	[rt]={Imm, 16'b0}
010000(16)	mfc0	从协处理器0移出/移入协处理器0	[rt]=[rd] / [rd]=[rt]
(rs=0/4)	mtc0		(rd在协处理器0里)
010001(17)	F型	fop=16 / 17: F型指令	见表B-3
010001(17)	bclf/bclt	fop=8: 如果fpcond为假/真则转移	if (fpcond == 0) PC = BTA

（续）

opcode	名　称	描　述	操　作
(rt = 0/1)			if (fpcond == 1) PC = BTA
100000(32)	lb	字节载入	[rt] = SignExt([Address]$_{7:0}$)
100001(33)	lh	半字载入	[rt] = SignExt([Address]$_{15:0}$)
100011(35)	lw	字载入	[rt] = [address]
100100(36)	lbu	无符号字节载入	[rt] = ZeroExt([Address]$_{7:0}$)
100101(37)	lhu	无符号半字载入	[rt] = ZeroExt([Address]$_{15:0}$)
101000(40)	sb	字节存储	[Address]$_{7:0}$ = [rt]$_{7:0}$
101001(41)	sh	半字存储	[Address]$_{15:0}$= [rt]$_{15:0}$
101011(43)	sw	字存储	[Address] = [rt]
110001(49)	lwcl	将字载入到FP协处理器1中	[ft] = [Address]
111001(56)	swcl	将FP协处理器的内容存储到内存	[Address] = [ft]

表B-2　按funct字段排列的R型指令

funct	指令名字	指令描述	操　作
000000(0)	sll	逻辑左移	[rd] = [rt] << shamt
000010(2)	srl	逻辑右移	[rd] = [rt] >> shamt
000011(3)	sra	算术右移	[rd] = [rt] >>>shamt
000100(4)	sllv	带变量的逻辑左移	[rd] = [rt] << [rs]$_{4:0}$ 汇编编码：sllv rd, rt, rs
000110(6)	srlv	带变量的逻辑右移	[rd] = [rt] >> [rs]$_{4:0}$ 汇编编码：srlv rd, rt, rs
000111(7)	srav	带变量的算术右移	[rd] = [rt] >>> [rs]$_{4:0}$ 汇编编码：srav rd, rt, rs
001000(8)	jr	寄存器跳转	PC = [rs]
001001(9)	jalr	寄存器跳转并连接	$ra = PC + 4, PC = [rs]
001100(12)	syscall	系统调用	系统调用例外处理指令
001101(13)	break	中断	中断处理指令
010000(16)	mfhi	从hi寄存器中移出	[rd] = [hi]
010001(17)	mthi	移入hi寄存器	[hi] = [rs]
010010(18)	mflo	从lo寄存器中移出	[rd] = [lo]
010011(19)	mtlo	移入lo寄存器	[lo] = [rs]
011000(24)	mult	乘法操作	{[hi]. [lo]} = [rs] × [rt]
011001(25)	multu	无符号乘法	{[hi]. [lo]} = [rs] × [rt]
011010(26)	div	除法操作	[lo] = [rs]/[rt] [hi] = [rs]%[rt]
011011(27)	divu	无符号除法	[lo] = [rs]/[rt] [hi] = [rs]%[rt]
100000(32)	add	加法操作	[rd] = [rs] + [rt]
100001(33)	addu	无符号加法	[rd] = [rs] + [rt]
100010(34)	sub	减法操作	[rd] = [rs] − [rt]
100011(35)	subu	无符号减法	[rd] = [rs] − [rt]
100100(36)	and	与操作	[rd] = [rs] & [rt]
100101(37)	or	或操作	[rd] = [rs] | [rt]
100110(38)	xor	异或操作	[rd] = [rs] ^ [rt]
100111(39)	nor	或非操作	[rd] = ~([rs] | [rt])
101010(42)	slt	小于设置	[rs]<[rt] ? [rd]=1 : [rd]=0
101011(43)	sltu	小于设置（无符号比较）	[rs]<[rt] ? [rd]=1 : [rd]=0

表B-3 F型指令（fop = 16/17）

funct	指令名字	指令描述	操作
000000(0)	add.s/add.d	浮点加法	[fd] = [fs] + [ft]
000001(1)	sub.s/sub.d	浮点减法	[fd] = [fs] - [ft]
000010(2)	mul.s/mul.d	浮点乘法	[fd] = [fs] * [ft]
000011(3)	div.s/div.d	浮点除法	[fd] = [fs] / [ft]
000101(5)	abs.s/abs.d	浮点取绝对值	[fd] = ([fs] < 0) ? [-fs] : [fs]
000111(7)	neg.s/neg.d	浮点取反	[fd] = [-fs]
111010(58)	c.seq.s/c.seq.d	浮点比较是否相等	fpcond = ([fs] == [ft])
111100(60)	c.lt.s/c.lt.d	浮点比较是否小于	fpcond = ([fs] < [ft])
111110(62)	c.le.s/c.le.d	浮点比较是否小于等于	fpcond = ([fs]≤[ft])

延伸阅读材料

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

The fascinating biography of Robert Noyce, an inventor of the microchip and founder of Fairchild and Intel. For anyone thinking of working in Silicon Valley, this book gives insights into the culture of the region, a culture influenced more heavily by Noyce than any other individual.

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.

An insider's tale of the development of several generations of Intel's Pentium chips, told by one of the leaders of the project. For those considering a career in the field, this book offers views into the managment of huge design projects and a behind-the-scenes look at one of the most significant commercial microprocessor lines.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

The most complete text on computer arithmetic systems. An excellent resource for building high-quality arithmetic units for computers.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach, 4th ed.*, Morgan Kaufmann, 2006.

The authoritative text on advanced computer architecture. If you are intrigued about the inner workings of cutting-edge microprocessors, this is the book for you.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

A classic story of the design of a computer system. Three decades later, the story is still a page-turner and the insights on project managment and technology still ring true.

Pedroni V., *Circuit Design with VHDL*, MIT Press, 2004.

A reference showing how to design circuits with VHDL.

Thomas D., and Moorby P., *The Verilog Hardware Description Language, 5th ed.*, Kluwer Academic Publishers, 2002.

Ciletti M., *Advanced Digital Design with the Verilog HDL*, Prentice Hall, 2003.

Both excellent references covering Verilog in more detail.

Verilog IEEE Standard (IEEE STD 1364).

The IEEE standard for the Verilog Hardware Description Language; last updated in 2001. Available at *ieeexplore.ieee.org*.

VHDL IEEE Standard (IEEE STD 1076).

The IEEE standard for VHDL; last updated in 2004. Available from IEEE. Available at *ieeexplore.ieee.org*.

Wakerly J., *Digital Design: Principles and Practices, 4th ed.*, Prentice Hall, 2006.

A comprehensive and readable text on digital design, and an excellent reference book.

Weste N., and Harris D., *CMOS VLSI Design, 3rd ed.*, Addison-Wesley, 2005.

Very Large Scale Integration (VLSI) Design is the art and science of building chips containing oodles of transistors. This book, coauthored by one of our favorite writers, spans the field from the beginning through the most advanced techniques used in commercial products.