计 算 机 科 学 丛 书

Education

计算机体系结构

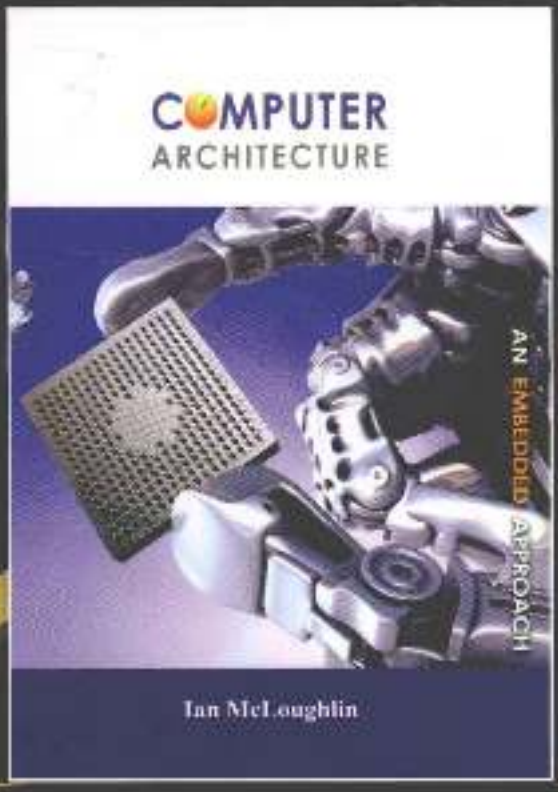
嵌入式方法

Ian McLoughlin
新加坡南洋理工大学 著

王沁 齐悦
北京科技大学 译

Computer Architecture

An Embedded Approach



机械工业出版社
China Machine Press

计算机体系结构 嵌入式方法

Computer Architecture An Embedded Approach

计算机科学正在发生深刻的变革，随着嵌入式系统的广泛应用，每年全世界售出的嵌入式计算机大约是桌面计算机的40倍，大学毕业生将更多地从事嵌入式系统硬件的设计，而不是设计一个传统的桌面计算机。计算学科发展和产业界的技术进步，要求高等教育也进行与时俱进的改革。本书采用现代视角来看待当今的计算机体系结构，侧重日益重要的嵌入式系统，以满足教育界和工业界对于硬件设计人才培养的需求。

本书在提供计算机体系结构背景知识的基础上，将讲述的重点放在我们每天生活和工作都要依赖的嵌入式系统上。书中首先介绍了算术和处理中的基础知识，然后分几章讨论了CPU结构、功能单元、性能优化、外部接口、实际嵌入式处理和计算的未来。本书还提出了嵌入式工业中关键的几个特定主题，以及在CPU设计项目中包含设计、仿真、测试和规划一个简单嵌入式计算机的一切必要步骤。

本书所采用的嵌入式系统相关的方法，使得书中的知识内容更加贴近工业界的需要，激励学生更投入地学习，并构建起该门课程与传统课程体系内其他相关课程（如电子学、计算机工程或计算机科学课程）的相互联系。本书不是在传统计算机体系结构教材的基础上扩展了嵌入式系统的一章，而是以新颖的视角来看待今天的计算机体系结构——它以历史上巨大而古老的机器为基础，现在正朝着嵌入式系统高度集成化的方向不断发展。

作者简介

Ian McLoughlin 现任新加坡南洋理工大学计算机工程学院副教授，拥有英国伯明翰大学博士学位。过去的20年，他在三大洲的工业部门、政府部门和学术界都工作过。作为经验丰富的工程师，他设计和工作的系统涉及航空、电信、日常消费品等诸多领域。他经常主持面向电子行业的技术培训课程，并为嵌入式系统（尤其是与音频和无线技术相关）的开发项目提供咨询和顾问服务。由于其在农村电信解决方案方面的杰出工作，他与新西兰大吉电子有限公司（Tait Electronics Ltd）的团队成员在2005年荣获首届IEEE工行创新奖。他是IET会员、IEEE高级会员、英国注册工程师和全欧工程师。

Mc Graw Hill Education
http://www.mheducation.com

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-37904-1

9 787111 379041

定价：59.00元

计 算 书

计算机体系结构

嵌入式方法

Ian McLoughlin
新加坡南洋理工大学 著

王沁 齐悦
北京科技大学 译

Computer Architecture

An Embedded Approach



机械工业出版社
China Machine Press

本书在传统计算机体系结构教科书的基础之上扩展了嵌入式系统的内容，并以新颖而完整的视角看待今天的计算机体系结构。前两章讲了计算机的发展和算术处理基础知识，随后分别介绍了CPU结构、功能单元、性能优化、外部接口、实际嵌入式处理和计算的未来。本书不仅通过大量的图表和例子来增强可读性，同时文中还穿插了许多注释框来拓宽读者的视野，其中包括一些额外的例子、有趣的信息摘要和附加的解释。书中除了包含嵌入式工程师所需的所有与典型计算机体系结构理论课程有关的主要内容外，还包括大量对目标读者有用的信息——甚至给读者提供建立和测试自定义软核处理器的机会，每一个主要的章节末尾都配有思考题，在教学指南中有参考答案。更多的案例和建议的阅读材料参见网站www. mheducation. asia/olc/mcloughlin。

本书适合于选择了计算机体系结构相关课程的本科生，尤其是大三学生，也适合于那些在开始更深层次的课题之前需要了解计算机体系结构最新知识的研究生，还适合于行业工程师。

Ian McLoughlin: Computer Architecture: An Embedded Approach ISBN 978-007-131118-2.

Copyright © 2011 by McGraw-Hill Education (Asia).

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education (Asia) and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2012 by McGraw-Hill Education (Asia), a division of the Singapore Branch of The McGraw-Hill Companies, Inc. and China Machine Press.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔（亚洲）教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内（不包括香港特别行政区、澳门特别行政区和台湾）销售。

版权© 2012 由麦格劳-希尔（亚洲）教育出版公司与机械工业出版社所有。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

封底无防伪标均为盗版

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：01-2011-3362

图书在版编目（CIP）数据

计算机体系结构：嵌入式方法／麦克洛克林（McLoughlin, I.）著；王沁，齐悦译. —北京：机械工业出版社，2012.5

（计算机科学丛书）

书名原文：Computer Architecture: An Embedded Approach

ISBN 978-7-111-37904-1

Ⅰ. 计…　Ⅱ. ①麦…　②王…　③齐…　Ⅲ. 计算机体系结构　Ⅳ. TP303

中国版本图书馆CIP数据核字（2012）第058549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：朱秀英

北京诚信伟业印刷有限公司印刷

2012年6月第1版第1次印刷

185mm×260mm · 21.75印张

标准书号：ISBN 978-7-111-37904-1

定价：59.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

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公司较早意识到“出版要为教育服务”。自 1998 年开始，我们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。华章公司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
华章科技图书出版中心

前 言

Computer Architecture: An Embedded Approach

在任何时候都有大批关于计算机体系结构的书籍出版。许多著名作者都试图在该领域有所作为，然而，计算机是一个高速发展的领域，因此，几乎没有书籍可以不加改动便能跟上时代的步伐。首先，向嵌入式计算系统的迅速转变就让许多作者及其所写的东西落伍了。有些教科书坚持将计算机看做是20世纪50～60年代像房子大小的机器，而更多的是将计算机视为20世纪80～90年代的台式机和服务器。只有少数人认为绝大多数现代计算机已经嵌入到我们日常所见的物品中，且几乎还没有人意识到未来是嵌入式的，即终有一天台式计算机将会和50年前的穿孔卡片计算机一样，被认为是过时的。

本书面向嵌入式计算的未来。关于嵌入式处理器的论题将与其他教科书中的更为传统的论题放在一起讨论，而且会尽可能强调来自嵌入式世界的实例。

这本书的目标读者主要由三部分人群组成。首先是选择了计算机体系结构相关课程的本科生，尤其是大三学生。其次是那些在开始一个更深层次的课题之前需要计算机体系结构最新知识的研究生们。第三类便是行业工程师。随着可重构逻辑电路，特别是FPGA（现场可编程门阵列）不断变大、变快和更加廉价，人们对于软核计算机愈发感兴趣，它是由工程师为特定任务而设计的CPU。这也许是有史以来第一次，设计工具让普通工程师有机会设计和构建自己定制的计算机。这本书将会为工程师们提供一个理解计算机体系结构的传统和现代技术及其权衡取舍的坚固的知识平台，这是一门计算机设计的艺术。

本书以全新的视角写作而成，不会依靠任何一本现有书籍。这种做法能让本书避免许多计算机发展史上人们走过的死胡同和无关区域，并使它有一个更加精确定义的目标。本书不仅仅在计算机体系结构教科书的基础之上扩展了几章嵌入式系统内容，而且以新颖而又完整的视角来看待今天的计算机体系结构——它以历史上巨大而古老的机器为基础，现在正朝着嵌入式系统高度集成化的方向不断发展。

该书旨在通俗易懂。在书中穿插了许多图表，以便读者理解那些晦涩难懂的概念，同时也有许多注释框贯穿全文，其中包括一些额外的例子、有趣的信息摘要和附加的解释，用于扩充正文。除了包含嵌入式工程师所需的所有与典型计算机体系结构理论课程有关的主要内容外（这其中不包括磁带存储、温切斯特驱动器和超级计算机设计），本书还包括大量对目标读者有用的信息——甚至给读者提供建立和测试自定义软核处理器的机会。

本书通篇将使用国际单位（SI），也包括新的计算机存储度量单位KiB和MiB（在附录A里有相关解释）。每一个主要的章节末尾都配有思考题，在教师手册中有参考答案。更多的例子和推荐的进一步阅读材料参见本书相关网站www.mheducation.asia/olc/mcloughlin。

Ian McLoughlin

在此特别感谢我那极富耐心的妻子 Kwai Yoke，还有我那双可爱的儿女 Wesley 以及 Vanessa，是他们给了我充足的时间撰写此书。而 Tom Scott、Benjamin Premkumar、Stefan Lendnal 和 Adrian Busch 则在我需要的时候给了我许多鼓励（本书有一个长达 5 年的构思时期）。Doug McConnell 是一个能给他人带来鼓舞的人，如同已故的 Angus Tait 先生一般——本书的绝大部分都是我在新西兰基督城的新西兰大吉电子有限公司（Tait Electronics Ltd）担任其科研小组的首席工程师期间撰写的。该公司是大洋洲最大的集电子设备研究与开发为一体的高科技专业通信公司，30 年前由时年 55 岁的 Angus Tait 创立，要知道在这个年龄段绝大多数人都在考虑退休事宜了，然而 Angus Tait 不是这样，他坚持每天工作，为公司指明前进的方向，直到他于 2007 年 8 月离世。

我要感谢那些在新加坡南洋理工大学（NTU）学习计算机体系结构、高级计算机体系结构以及计算机外围设备的学生们，正是他们问了我一些颇有难度的问题，才延伸了我的知识面并促使我在教学方面取得进步。Lee Keok Kee 副教授督促我收集与该书有关的资料，同时我还要感谢我在 NTU 的其他朋友和同事，以及在大吉电子有限公司、Simoco、伯明翰大学、英国皇家政府通信中心（HMGCC）和 GEC Hirst 研究中心的前同事们。除此之外，我还得对 Gerald Bok 和 McGraw-Hill 出版公司的同事们表示衷心的感谢，特别是 Doreen Ng 和编辑团队，正是由于他们的专业技巧、职业精神和认真努力，才使这份手稿蜕变成一本美妙的书籍。

在此，我最想感激的是我的母亲，是她始终鼓励着我，不仅仅是在写这本书时，而且在我整个生命历程中，不离不弃，风雨相随。正是她对我的殷切期望，使我最终走上科学研发的道路，而她也始终以最饱满的热情来关注我对于写作的尝试。谢谢你，妈妈！

Ian McLoughlin

目 录

Computer Architecture: An Embedded Approach

出版者的话

前言

致谢

第 1 章 引言 …… 1

1.1 本书组织结构 …… 1

1.2 进化过程 …… 1

1.3 计算机发展阶段划分 …… 4

1.3.1 第一代计算机 …… 4

1.3.2 第二代计算机 …… 4

1.3.3 第三代计算机 …… 5

1.3.4 第四代计算机 …… 6

1.3.5 第五代计算机 …… 7

1.4 云、普适、网格和超并行计算机 …… 7

1.5 未来 …… 8

1.6 小结 …… 9

第 2 章 基础知识 …… 10

2.1 计算机组成 …… 10

2.1.1 Flynn 分类法 …… 10

2.1.2 连接方式 …… 11

2.1.3 计算机结构层次视图 …… 11

2.2 计算机基本原理 …… 12

2.3 数字格式 …… 15

2.3.1 无符号二进制 …… 15

2.3.2 原码 …… 15

2.3.3 反码 …… 15

2.3.4 补码 …… 15

2.3.5 移码 (excess-n) …… 16

2.3.6 BCD 码 …… 16

2.3.7 定点数表示法 …… 17

2.3.8 符号扩展 …… 17

2.4 算术运算 …… 18

2.4.1 加法 …… 18

2.4.2 并行进位传递加法器 …… 18

2.4.3 超前进位 …… 20

2.4.4 减法 …… 20

2.5 乘法 …… 21

2.5.1 加法迭代法 …… 22

2.5.2 部分积方法 …… 22

2.5.3 移位加方法 …… 24

2.5.4 Booth 和 Robertson 方法 …… 25

2.6 除法 …… 26

2.7 定点数格式的运算 …… 28

2.7.1 定点数的运算 …… 28

2.7.2 定点数的乘除 …… 29

2.8 浮点数 …… 30

2.8.1 广义浮点数 …… 30

2.8.2 IEEE754 浮点标准 …… 30

2.8.3 IEEE754 标准模式 …… 31

2.8.3.1 规格化模式 …… 31

2.8.3.2 非规格化模式 …… 32

2.8.3.3 其他模式数 …… 33

2.8.4 IEEE754 数的范围 …… 33

2.9 浮点数处理 …… 35

2.9.1 IEEE754 数的加减运算 …… 36

2.9.2 IEEE754 数的乘除法 …… 38

2.9.3 IEEE754 中间格式 …… 38

2.9.4 舍入 …… 39

2.10 小结 …… 39

思考题 …… 40

第 3 章 CPU 基础 …… 42

3.1 什么是计算机 …… 42

3.2 让计算机为你服务 …… 42

3.2.1 程序存储 …… 42

3.2.2 存储架构 …… 43

3.2.3 程序传输 …… 44

3.2.4 控制单元 …… 44

3.2.5 微指令 …… 48

3.2.6 RISC 和 CISC 的对比 …… 49

3.2.7 处理器实例 …… 51

3.3 指令处理 …… 51

3.3.1 指令集 …… 52

3.3.2 取指和译码 …… 54

3.3.2.1 指令译码 …… 55

3.3.2.2 取操作数 …… 55

3.3.2.3 分支 …… 56

3.3.2.4 立即数 …… 57
3.3.3 压缩指令集 …… 57
3.3.4 寻址模式 …… 59
3.3.5 堆栈机和逆波兰表示法 …… 61
3.4 数据处理 …… 62
3.4.1 数据的格式和表达 …… 63
3.4.2 数据流 …… 65
3.4.3 数据存储 …… 66
3.4.4 内部数据 …… 66
3.4.5 数据处理 …… 67
3.4.5.1 在小位宽 CPU 上处理大位宽数字 …… 67
3.4.5.2 定点 CPU 上的浮点数 …… 68
3.4.5.3 复数 …… 69
3.5 自顶向下方法 …… 69
3.5.1 计算机的能力 …… 69
3.5.1.1 功能 …… 70
3.5.1.2 时钟频率 …… 70
3.5.1.3 位宽 …… 70
3.5.1.4 内存 …… 70
3.5.2 性能衡量和统计 …… 70
3.5.3 性能评估 …… 72
3.6 小结 …… 73
思考题 …… 74
第 4 章 处理器内部组成 …… 76
4.1 内部总线结构 …… 76
4.1.1 程序员的角度 …… 76
4.1.2 分解互联排列 …… 77
4.1.3 ADSP21xx 总线排列 …… 78
4.1.4 数据与程序同时访存 …… 78
4.1.5 双总线体系结构 …… 80
4.1.6 单总线体系结构 …… 81
4.2 算术逻辑单元 …… 82
4.2.1 ALU 功能 …… 82
4.2.2 ALU 设计 …… 83
4.3 内存管理单元 …… 85
4.3.1 对虚拟存储的需求 …… 85
4.3.2 MMU 操作 …… 85
4.3.3 退回算法 …… 87
4.3.4 内部存储碎片和片段 …… 87
4.3.5 外部碎片 …… 88
4.3.6 改进的 MMU …… 89
4.3.7 内存保护 …… 89
4.4 cache …… 90
4.4.1 直接相联 cache …… 92
4.4.2 组相联 cache …… 93
4.4.3 全相联 cache …… 94
4.4.4 局部性原则 …… 94
4.4.5 cache 替换算法 …… 95
4.4.6 cache 性能 …… 98
4.4.7 cache 一致性 …… 99
4.5 协处理器 …… 100
4.6 浮点运算单元 …… 101
4.6.1 浮点仿真 …… 102
4.7 SIMD 流指令扩展（SSE）和多媒体扩展 …… 103
4.7.1 多媒体扩展（MMX） …… 103
4.7.2 MMX 实现 …… 103
4.7.3 MMX 的使用 …… 104
4.7.4 SIMD 流指令扩展（SSE） …… 105
4.7.5 使用 SSE 和 MMX …… 105
4.8 嵌入式系统中的协处理 …… 105
4.9 小结 …… 106
思考题 …… 107
第 5 章 提高 CPU 性能 …… 110
5.1 CPU 加速技术简介 …… 110
5.2 流水线 …… 111
5.2.1 多功能流水线 …… 112
5.2.2 动态流水线 …… 113
5.2.3 改变流水线模式 …… 113
5.2.4 数据相关冒险 …… 114
5.2.5 条件冒险 …… 116
5.2.6 条件分支 …… 117
5.2.7 编译时流水线补偿 …… 118
5.2.8 相对地址分支 …… 119
5.2.9 流水线的指令集补偿 …… 120
5.2.10 运行时流水线补偿 …… 122
5.3 复杂指令集（CISC）和精简指令集（RISC） …… 123
5.4 超标量体系结构 …… 124
5.4.1 简单超标量 …… 124
5.4.2 多发送超标量 …… 125
5.4.3 超标量的性能 …… 126
5.5 每周期的指令数 …… 127

5.5.1 不同体系结构的 IPC …… 127
5.5.2 IPC 度量 …… 128
5.6 硬件加速器 …… 129
5.6.1 零开销循环 …… 129
5.6.2 地址处理硬件 …… 131
5.6.3 影子寄存器 …… 134
5.7 分支预测 …… 134
5.7.1 分支预测的必要性 …… 134
5.7.2 单 T 位预测器 …… 136
5.7.3 双位预测器 …… 137
5.7.4 计数器和移位器预测器 …… 138
5.7.5 局部分支预测器 …… 139
5.7.6 全局分支预测器 …… 141
5.7.7 G 选择预测器 …… 142
5.7.8 G 共享预测器 …… 143
5.7.9 混合预测器 …… 144
5.7.10 分支目标缓冲 …… 145
5.7.11 基本代码段 …… 147
5.7.12 分支预测总结 …… 148
5.8 并行机器 …… 148
5.8.1 SISD 向 MIMD 的演变 …… 150
5.8.2 为提高性能而采用并行 …… 152
5.8.3 其他并行处理 …… 153
5.9 Tomasulo 算法 …… 155
5.9.1 Tomasulo 算法的原理 …… 155
5.9.2 Tomasulo 系统的例子 …… 155
5.9.3 嵌入式系统中的 Tomasulo 算法 …… 159
5.10 小结 …… 160
思考题 …… 160
第 6 章 外部总线 …… 163
6.1 总线接口 …… 163
6.1.1 总线控制信号 …… 164
6.1.2 直接存储器存取（DMA） …… 164
6.2 并行总线规范 …… 165
6.3 标准接口 …… 166
6.3.1 系统控制接口 …… 166
6.3.2 系统数据总线 …… 167
6.3.2.1 ISA 总线及其衍生总线 …… 167
6.3.2.2 PC/104 …… 168
6.3.2.3 PCI …… 170
6.3.2.4 LVDS …… 170
6.3.3 输入/输出总线 …… 171
6.3.4 外设器件总线 …… 172
6.3.5 与网络设备的接口 …… 172
6.4 实时性问题 …… 172
6.4.1 外部激励 …… 173
6.4.2 中断 …… 173
6.4.3 实时性定义 …… 173
6.4.4 时间范围参数 …… 174
6.4.5 硬件体系结构对实时操作系统的支持 …… 175
6.5 中断和中断处理 …… 176
6.5.1 中断的重要性 …… 176
6.5.2 中断过程 …… 176
6.5.2.1 中断事件通知处理器 …… 176
6.5.2.2 CPU 完成正在进行的工作 …… 177
6.5.2.3 转入中断服务例程 …… 177
6.5.2.4 中断重定向 …… 179
6.5.3 高级中断处理 …… 180
6.5.4 共享中断 …… 180
6.5.5 可重入代码 …… 181
6.5.6 软件中断 …… 181
6.6 无线 …… 181
6.6.1 无线技术 …… 181
6.6.2 无线接口 …… 183
6.6.3 无线相关问题 …… 183
6.7 小结 …… 183
思考题 …… 184
第 7 章 实用嵌入式 CPU …… 187
7.1 概述 …… 187
7.2 微处理器不只是核 …… 187
7.3 功能需求 …… 189
7.4 时钟 …… 192
7.5 时钟与功耗 …… 194
7.5.1 传输延迟 …… 195
7.5.2 电流相关问题 …… 195
7.5.3 时钟问题解决方法 …… 196
7.5.4 低电压设计 …… 196
7.6 存储 …… 197
7.6.1 早期的计算机存储 …… 198
7.6.2 只读存储器 …… 198
7.6.3 随机存取存储器 …… 203
7.6.3.1 静态 RAM …… 204

7.6.3.2 动态 RAM …… 205
7.6.3.3 DRAM 的寻址机制 …… 206
7.7 分页与重叠 …… 209
7.8 嵌入式系统中的存储 …… 210
7.8.1 非易失存储器 …… 211
7.8.2 易失存储器 …… 212
7.8.3 其他存储器 …… 213
7.9 测试和验证 …… 214
7.9.1 集成电路设计和制造问题 …… 215
7.9.2 内置自测（BIST） …… 216
7.9.3 联合测试行动小组（JTAG） …… 218
7.10 错误检测和纠正 …… 220
7.11 看门狗定时器和复位监测 …… 223
7.12 逆向工程 …… 225
7.12.1 逆向工程过程 …… 226
7.12.1.1 功能分析 …… 228
7.12.1.2 物理结构分析 …… 228
7.12.1.3 材料清单 …… 228
7.12.1.4 系统架构 …… 229
7.12.2 详细的物理布局 …… 229
7.12.2.1 电气连接原理图 …… 230
7.12.2.2 存储程序 …… 230
7.12.2.3 软件 …… 230
7.13 防止逆向工程 …… 232
7.13.1 存储程序的被动模糊 …… 233
7.13.2 可编程逻辑家族 …… 234
7.13.3 主动 RE 防范 …… 234
7.13.4 主动 RE 防范分类 …… 235
7.14 小结 …… 236
思考题 …… 236
第 8 章 CPU 设计 …… 238
8.1 软核处理器 …… 238
8.1.1 微处理器不仅仅有处理器核 …… 238
8.1.2 软核处理器的优点 …… 239
8.1.2.1 性能 …… 239
8.1.2.2 可用性 …… 239
8.1.2.3 效率 …… 240
8.1.2.4 人为因素 …… 240
8.2 软硬件协同设计 …… 241
8.3 现成的软核 …… 243
8.4 制作自己的软核 …… 245
8.5 CPU 设计规格说明 …… 245
8.5.1 CPU 体系结构 …… 246
8.5.2 总线 …… 246
8.5.3 程序及数据存储 …… 247
8.5.4 逻辑运算 …… 248
8.5.5 指令处理 …… 248
8.5.6 系统控制 …… 249
8.6 指令集 …… 249
8.6.1 CPU 控制 …… 251
8.6.1.1 闲置状态 …… 252
8.6.1.2 取指状态 …… 252
8.6.1.3 执行状态 …… 252
8.7 CPU 实现 …… 253
8.7.1 测试的重要性 …… 253
8.7.2 定义操作和状态：defs.v …… 253
8.7.3 第一个小模块：counter.v …… 254
8.7.4 CPU 控制：state.v …… 256
8.7.5 程序和变量存储：ram.v …… 257
8.7.6 堆栈：stack.v …… 259
8.7.7 算术、逻辑和乘法单元：alu.v … 261
8.7.8 综合测试：tinycpu.v …… 263
8.8 CPU 测试及运行 …… 267
8.9 编程并使用 CPU …… 267
8.9.1 编写 TinyCPU 程序 …… 268
8.9.2 TinyCPU 编程工具 …… 271
8.10 小结 …… 271
思考题 …… 272
第 9 章 未来 …… 274
9.1 单比特体系结构 …… 274
9.1.1 位串行加法 …… 274
9.1.2 位串行减法 …… 275
9.1.3 位串行逻辑电路和处理 …… 275
9.2 超长指令字体系结构 …… 276
9.2.1 VLIW 的基本原理 …… 276
9.2.2 VLIW 的难题 …… 277
9.3 并行与大规模并行计算机 …… 278
9.3.1 大型计算机集群 …… 278
9.3.2 小型计算机集群 …… 279
9.3.2.1 从嵌入式系统中解放计算资源 …… 279
9.3.2.2 并行处理单元 …… 279
9.3.3 并行和集群处理注意事项 …… 283
9.3.4 互连策略 …… 283

9.4 异步处理器 …… 285
9.4.1 数据流控制 …… 286
9.4.2 避免流水线冒险 …… 286
9.5 替代数字格式系统 …… 287
9.5.1 多值逻辑 …… 287
9.5.2 带符号数字的表示 …… 288
9.6 光计算 …… 289
9.6.1 光电全加器 …… 289
9.6.2 光电底板 …… 290
9.7 科幻小说还是未来的现实 …… 291
9.7.1 分布式计算 …… 291
9.7.2 湿件（大脑） …… 291
9.8 小结 …… 292
附录A 内存大小的标准表示方法 …… 293
附录B 开放系统互连模型 …… 295
B.1 引言 …… 295
B.2 OSI层次 …… 295
B.3 小结 …… 296
附录C 探索cache大小和结构安排的权衡设计方法 …… 297
C.1 引言 …… 297
C.2 准备工作 …… 297
C.3 安装Cacti和Dinero …… 297
C.4 工具的使用 …… 298
C.5 不同cache设计方案的实验 …… 299
C.6 cache设计中的进一步信息 …… 299
思考题 …… 300
附录D 嵌入式计算机上的无线技术 … 301
D.1 引言 …… 301
D.2 802.11a，b和g …… 301
D.3 802.11n …… 302
D.4 802.20 …… 302
D.5 802.16 …… 302
D.6 蓝牙 …… 303
D.7 GSM …… 303
D.8 GPRS …… 304
D.9 ZigBee …… 305
D.10 无线USB …… 305
D.11 近距离通信 …… 306
D.12 WiBro …… 307
D.13 无线设备总结 …… 307
D.14 应用举例 …… 308
D.15 小结 …… 308
附录E 编译和仿真TinyCPU的工具 …… 309
E.1 准备和软件获取 …… 309
E.2 如何编译和仿真Verilog …… 309
E.3 如何查看仿真输出 …… 313
E.4 高级测试平台 …… 318
E.5 小结 …… 318
附录F TinyCPU编译和汇编代码的工具 …… 319
F.1 引言 …… 319
F.2 汇编过程 …… 319
F.3 汇编器 …… 320
F.4 汇编程序实例 …… 322
F.5 编译器 …… 323
F.6 小结 …… 323
索引 …… 325

引　言

1.1　本书组织结构

计算机进化经历了一个漫长的历程：从1834年Charles Babbage的分析引擎（图1-1给出了他的差分机，它与后来的数字计算处理机具有类似设计）到今天的超级计算机，是一段为提高处理能力、复杂性和微型化而不懈努力的佳话。

令人吃惊的是，Babbage当年的很多技术（以及20世纪40年代初早期电子计算机）非常具有前瞻性，在当今系统中依然随处可见。遗憾的是，与过去的关联并不总是正面的，今天Intel台式处理器由于8086等约束在性能提升方面受到了限制。今天，我们有机会在事后通过计算发展的历史进行反思，从而可以发现许多革命的进化分支，这些分支最初看来很有前途，但很快失去了生命力。它们当中有些可能会在后来的一些专门化机器中重新出现，但更多的仅仅成为了一段历史。

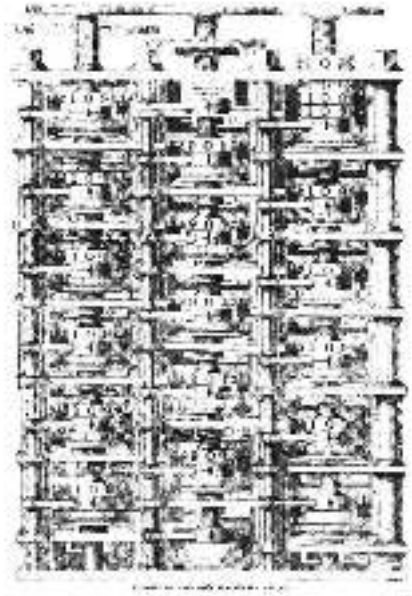


图1-1　Babbage分析差分机的一部分，来源于Harper的月刊杂志第29卷，第173期，p.34，1864年；伦敦科学博物馆可以见到其原始文档和一台可工作的重建机器

当今的计算机很有可能会建立在今天所用的技术之上。当前技术的快照（正如所有计算机教科书所必须做的）要如实反映这个事实，而不能脱离历史地介绍技术。

本书将按照技术进化过程展开。前面几章重点描述计算机基础，掌握这些基础能使学生们在纸面上构造已可工作但效率低的计算机。之后的章节将描述当今体系结构在提高性能方面的先进技术。这些先进技术与计算机基础分开讲述，原因在于它们当中有些可能是进化过程中的死胡同，仅在当今推动Moore定律快速向前。

在计算机体系结构方面最终会出现革命性的变化——它将打破进化趋势，使得许多曾经有助于提高性能的技术被淘汰。本书中不可能确定这些技术是什么，但我们可以做一些非正式的猜测。在本书的最后一章，我们将讨论在未来几十年可能带来革命性变化的先进技术。

1.2　进化过程

动物进化的概念是有争议的：至今还没有科学的证明，很多人也只是选择相信它而已。一些人喜欢"等等看"，期待着科学最终给出结论，而另一些人则选择相信存在一个万能的不可见的创世者。不管动物从何而来，作为人造设备的计算机遵循了一个不断改进的进化过程，这是一个不争的事实。由于缺少突破性进展，计算的历史多年来所取得的是很多小的增量式改进。

当然，像计算机这样复杂的设备需要由有才华的工程师来设计。我们能够记得一些工程师

的名字，尤其是那些做出重大改进的人（其中有些人还活着，向我们讲述这些工作）。另外，那些开创性机器的设计和历史成就着巨大代价，应当被很好地记录下来。

因此，人们往往希望计算机的发展历史是清晰定义的，对半个世纪以来出现的那些开创性计算机的认识没有困惑和争议。但遗憾的是实际并不是这样：存在着非常不同的意见，关于准确的时间、贡献、"第一"等几乎没有一致意见。任意比较两本关于计算体系结构或计算机历史的书就会发现这一点。出于本书目的，我们将从巨人 Colossus 开始展现计算机历史。

Colossus（如图 1-2 所示）由工程师 Tommy Flower 于 1943 年建造，由 Alan Turing 及其在 Bletchley Park 的同事完成程序设计，目前普遍认为这是世界上第一台可编程电子计算机。这台计算机作为针对德国 Enigma 编码的密码破译工程（最终成功）的一部分构建于第二次世界大战中的英国。遗憾的是，Colossus 属于英国官方保密法之列而被隐藏了整整 50 年。战后，首相 Winston Churchill（以典型的保密文件的形式）命令将这台机器肢解成不大于人手的碎片，同时命令销毁所有与 Colossus 相关的文章。规划和图纸由发计者焚烧，解密操作员在被监禁或叛国罪的威胁下宣誓保守秘密。

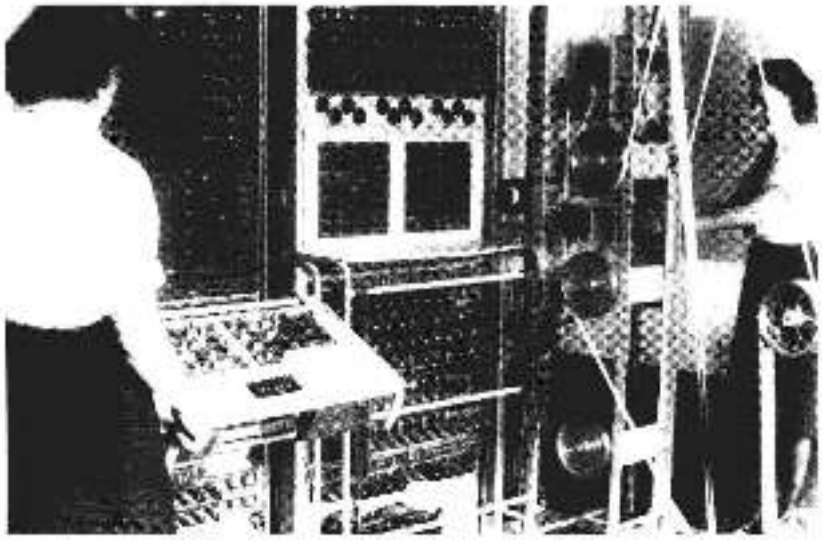


图 1-2 在第二次世界大战中使用的十台 Colossus 计算机之一（来源于 Bletchley Park Trust，在 www.bletchleypark.org.uk 上可以找到）

这台机器被隐藏得很成功。尽管偶尔有一些关于这台机器的传闻，但公众最终是在 2000 年所剩无几的几份相关文件被解密时才得知 Colossus 的存在，因此，在很多计算机历史的相关描述中都没有提到 Colossus，整整一代计算机设计师从来没有听说过它。

然而，另有一些著名的与 Colossus 同期的机器在之后几年开始使用。其中最著名的 ENIAC（Electronic Numerical Integrator And Computer）在美国建造。当 Colossus 还在被完全隐藏时，1944 年开始投入使用的 ENIAC 显然成为世界上第一台数字计算设备，许多不知道 Colossus 的教科书作者将 ENIAC 作为第一台现代计算机。事实上，Colossus 不仅投入使用时间超前，而且与现代计算机相似，是一台二进制计算机，而 ENIAC 是一台十进制计算机。但 Colossus 和 ENIAC 一样都很难编程，需要通过调整开关和变换连线位置实现重新编程。

令人惊叹的是，出现在一个世纪之前的 Charles Babbage 的分析引擎是数字的而不是模拟的，且完全可编程，比起那些被称为第一的电子计算机在某些方面更为先进。Babbage 还设计了一台打印机外设，可以输出数字计算结果。Babbage 的机器还具有完整的程序设计语言，能够处理循

环和条件分支，这使得 Babbage 的朋友 Lovelace 伯爵夫人 Ada Byron（著名诗人拜伦的女儿）在这台计算机上工作并写出了世界上第一个计算机程序。这可能是历史上诗歌创作和编程第一次也是最后一次走到一起。

在差分机和 Colossus 之间，计算机领域并不完全是荒漠：德国人 Konrad Zuse 在 1940 ~ 1941 年间构建了一台电力计算机，归类为电力而不是电子计算机的原因是它基于继电器建造而成。另一个电子计算机的早期尝试是 1941 年在美国爱荷华州立大学（Iwoa State College）构建了 Atanasoff-Berry 机器。虽然这台机器不可编程且不可靠，但它还是展示出一些现代计算机的概念，并且毫无疑问地推动了当时计算机领域的发展。

晶体管计算机的起源同样是一个令人困惑的领域。美国贝尔实验室于 1948 年发明了晶体管。鉴于它功耗低、尺寸小的特征，很适合构建计算机（虽然早期晶体管的可靠性低于电子管[㊀]）。有一些文字记载误将美国 MIT 建于 1956 年的 TX-0 作为第一台晶体管计算机，实际上曼彻斯特大学于 1953 年投入运行的晶体管计算机才应该享有“第一”的荣誉。

最后，关于第一台存储程序计算机（与此对应的是通过插线或开关编程）依然存在记录不清的情况。应当是曼彻斯特大学构建的小型试验机器（SSEM，或爱称“Baby”），它于 1948 年首次成功地运行了所存储的程序。

另一个早期的存储程序计算机是 Maurice Wilkes 的 EDSAC（Electronic Delay Storage Automatic Calculator），于 1949 年 5 月在剑桥大学开始运行。同样著名的是美国陆军的 EDVAC（Electronic Discrete Variable Automatic Computer），它也是二进制存储程序计算机，始建于 1944 年，但直到 1951 ~ 1952 年才开始运行。 4

显然，从上述三个存在异议的方面来看，计算机的历史并不像想象的那样一目了然。曼彻斯特大学是一个非常重要但是很低调的角色，被很多计算机历史学家忽视了。曼彻斯特大学于 1951 年[㊁]还设计出世界上第一台商用计算机 Ferranti Mark 1，但最终计算机商业中心却跑到了别的地方。

表 1-1 列出了计算机领域的世界第一，同时给出了它们实际开始运行的年代。

表 1-1 计算机发展史上重要的计算机

年份	研发地点	名称	首次产品
1834	Cambridge	差分机	可编程计算机
1943	Bletchley	Colossus	电子计算机
1948	Manchester	SSEM（Baby）	存储程序计算机
1951	MIT	Whirlwind 1	实时 I/O 计算机
1953	Manchester	晶体管计算机	晶体管计算机
1971	California	Intel 4004	大众市场 CPU & IC
1979	Cambridge	Sinclair ZX-79	大众市场家用计算机
1981	New York	IBM PC	个人计算机
1987	Cambridge	Acorn A400 系列	高街 RISC PC
1990	New York	IBM RS6000	超标量 RISC 处理器
1998	California	Sun picoJAVA	基于一种语言的计算机

㊀ 在局部真空中包含微小灯丝电极的玻璃热离子阀门是大部分早期计算机的基本逻辑开关。在北美称这种阀门为“真空管”或简单称为“管”。有趣的是，虽然它们现在已不再参与计算，但在很高端的音频放大设备中却是抢手货。

㊁ Ferranti Mark 1 之后是 LEO 计算机（从 EDSAC 派生而来），从 1951 年春季开始在其上运行财务程序，为遍布各处的里昂茶馆服务。

这表展示了计算机技术发展过程。尽管人们不清楚20世纪60年代是怎么回事，但可以看出这是一个进化过程，而不是一个革命性过程。

1.3 计算机发展阶段划分

有些时候计算机像人一样按照"代"进行描述。这是一种按照时间建立的划分，通常取决于建造方法、计算逻辑器件，以及计算机用途等要素。

[5] 任何看过20世纪80年代计算机杂志上的广告的人都会记得制造商如何教资不同代的计算机，并且反复为计算机新产品做广告，好像它们就是第五代计算机一样。值得庆幸的是，这种做法逐渐被弃，计算机世界步入了一个平坦高原。下面，我们来了解一下这五代计算机。

1.3.1 第一代计算机

- 基于真空管，通常会占据整个房间。
- MTBF（平均故障间隔时间）很短，两个故障之间间隔只有几分钟。
- 基于十进制的算术运算。
- 通过开关、电缆或硬连线实现编程。
- 在机器代码之上没有程序设计语言。
- 有程序存储，引入了码·诺依曼体系结构。

最典型的例子是ENIAC，耗电100kW才能达到每秒500个加法运算。这台巨大机器使用了1800个真空管，重30吨，占地面积为1300平方米。用户界面（这代计算机的典型界面）如图1-3所示。ENIAC由美国陆军设计，用于求解弹道方程以计算火炮射程表。

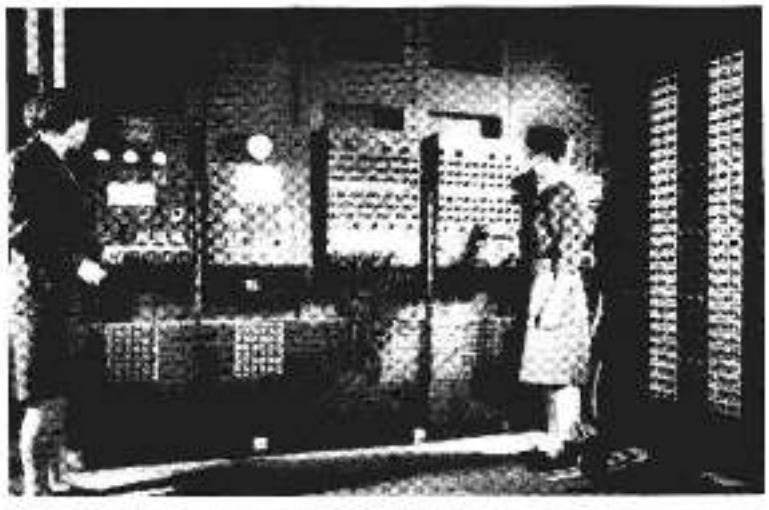


图1-3 两名女士正在操作ENIAC的主控制盘（照片由美国陆军提供）

Colossus计算机同样巨大，早期用于解码：破解强大而秘密的Enigma密码，为第二次世界大战中盟军胜利做出了贡献。然而，令人伤心的是，解码出的第一条德军信息是"我们正准备轰炸Coventry"。为了不让敌人知道信息已经被破解，政府决定不通知居民，致使很多居民在这次

[6] 轰炸中死亡或受伤。

1.3.2 第二代计算机

- 基于晶体管，但还是很重、很大。

- 较好的可靠性。
- 采用二进制逻辑。
- 用穿孔卡片或纸带进行程序输入。
- 支持早期的高级程序设计语言。
- 基于总线的系统架构。

当时的 CDC6600 被誉为智能外围设备，另一个更加为人所知的是有 4K 字的 RAM，运行速率为 0.2MHz 的 PDP-1。这台伟大机器使得现今已经倒闭的数字设备公司（DEC）在当时胜出。PDP-1 标价 10 万美元左右，且有令人印象深刻的一些外设，包括光笔、EYEBALL 数字照相机、两声道音频输出、电话接口、磁鼓存储设备、打印机、键盘接口和控制台显示设备。PDP-1 以及外设差不多占据了整个房间，如图 1-4 所示。



图 1-4 PDP-1（由 Lawrence Livermore 国家实验室档案，来源于 www.computer history.info）

1.3.3 第三代计算机

- 采用集成电路。
- 很好的可靠性。
- 可以进行仿真（模程序）。
- 多道程序设计、多任务、分时。
- 普遍使用高级程序设计语言，在用户界面设计方面进行了一些尝试。
- 使用虚拟存储和操作系统。

最为人所知且功能强大的 IBM System/360 包括 512KB 的 8 位存储器，运行主频为 4MHz。这是一个基于寄存器的计算机，具有流水线中央处理单元（CPU）体系结构和存储器读写机制，该设计方案为当今程序员所熟知。IBM 在基本机器架构基础上针对不同的用户要求进行了扩展设计，只通过微码仿真方法实现其他指令集合，从而保证为第二代计算机用户（那些已经在他们的机器上投入巨额资金的用户）提供向后兼容性。经过修改和小型化的五台计算机在 NASA 空间站中完成数值计算。

虽然没有占据一个房间，基本 S/360 机器仍然是一台很大的设备，如图 1-5 所示。

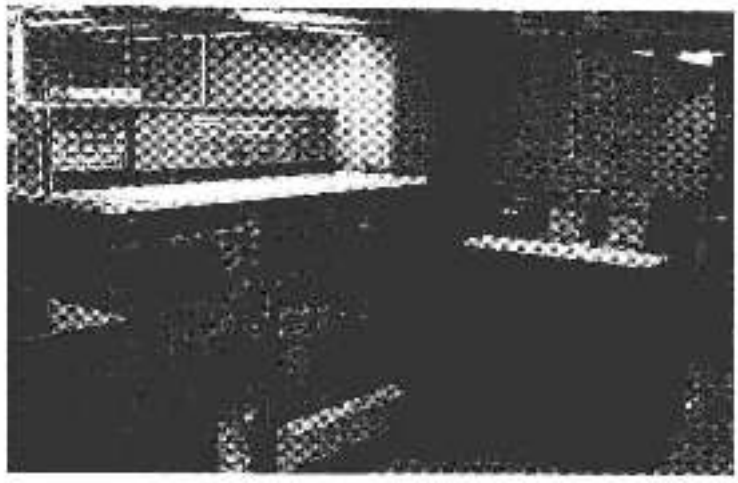


图 1-5 IBM System/360（由 Ben Franske 拍摄，来源于 Wikipedia IBM System/360 页面）

1.3.4 第四代计算机

- 采用超大规模（VLSI）集成电路。
- 高可靠性和高性能。
- 可以将整个 CPU 集成到一块芯片上。
- DOS，CP/M 及更先进的操作系统。
- 这就是今天的计算机。

例子非常之多，包括所有台式计算机和笔记本计算机。图 1-6 给出了 Acorn 公司的 Phoebe 系
[8] 统，它集成了创新性的 RISC 体系结构和先进的视窗操作系统。遗憾的是，该公司没有生存足够长时间来市场化该机器——原因可能在于该机器是明黄颜色。相反，Apple 公司在市场营销上表现得更聪明，初始投放市场的 333MHz 的 iMac 有 5 种可选颜色，尽管最近变成了全白、全黑和铝合金色。图 1-7 给出了一些 iMac 的新产品。

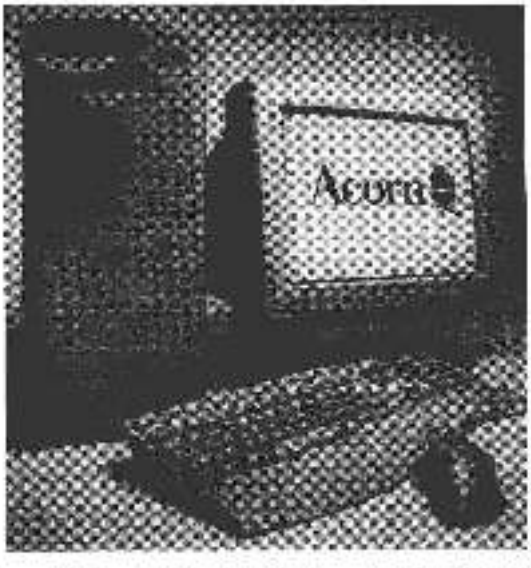


图 1-6 Acorn Phoebe（照片来源于 Acorn 计算机公司的公开材料，1998，http://acorn.chrisandjo.co.uk/AcornPlcs/pboeh2.html）

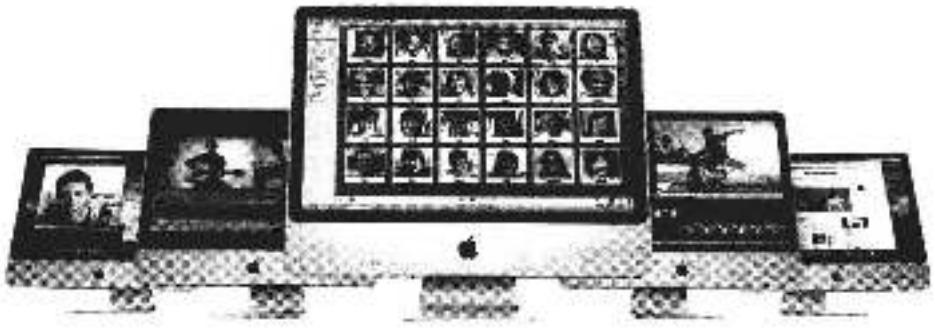


图 1-7 Apple iMac 系列产品：运行在可靠的基于 UNIX 操作系统上的时尚且用户友好的机器（图片来源于 Apple 公司）

1.3.5 第五代计算机

- 自然的人机交互。
- 非常高级的程序设计语言——甚至可以用英语编程。
- 对用户表现出智能。

在撰写本书的时候还没有合适的例子。一旦有了这样的例子，很可能也没有什么值得拍照的，它们可能是成千上万的嵌入式计算机，分布在我们周围，不以米色或黄色的盒子形式出现。

也许不真是什么第五代，但拥有良好配置和工艺的 Apple iMac 计算机（见图 1-7）也许能够预示未来，即拥有时尚外观且以用户为中心的计算机。或者，也许 Apple 的包括 8 个独立 ARM 处理器核的小型化产品 iPhone（如图 1-9 所示）及其令人印象深刻的 iPad 预示了第五代计算机？

1.4 云、普适、网格和超并行计算机

思考一下计算机发展历史。最开始计算机是占据整个房间的东西，无论是机械部分还是电子部分都需要专门的人来维护。技术的不断进步使得较小的晶体管取代了基于真空管的硬件。像房间大小的计算机开始逐步缩小。后来发明了集成电路，最初集成电路可以承载几百个晶体管，之后是几千个，后来更多。Rockwell 6502 处理器产生于 1975 年，在一个 40 管脚的双列直插式封装（DIP）芯片上集成了大约 4000 个晶体管。到 2008 年，Intel 可以在单个芯片上集成 20 亿个晶体管。

计算机从房间大小缩小到几台冰箱大小，然后缩小到一台冰箱大小，进而缩小到桌面上的盒子，即个人计算机（PC）时代。PC 又变得更小。20 世纪 80 年代早期出现了可以随身携带的计算机，然后是便携式计算机、膝上计算机、笔记本计算机和掌上计算机。今天，你可以买到一个用于医疗诊断的药片，其中不仅包含完整的嵌入式计算机，还包括传感器和 CMOS 照相机。

那么计算机发展史是一个单向小型化的故事吗？回答是否定的，因为计算机在某种意义上又变得越来越大。像为特网所表现的网络化趋势使得计算机更容易相互链接，同时还将支持计算机之间的资源共享。过去单机完成的工作今天可以由多个并行计算单元或计算机集群并行完成，甚至可以由地理位置各异的计算机共同完成（9.3 节将详细讨论大规模并行计算问题）。

因此，假设我们需要完成的任务可以在单个小盒子上执行，也可以由分布在我们周围的几台计算机（包括嵌入式的）共同完成，那么问题就变成我们应如何定义“计算机”——是这个盒子自身，还是某个可以执行该任务的“东西”？

50 年前很容易定义什么是“计算机”，因为计算机是在计算机机房中。今天，在我们桌上的单个盒子中就有可能包含两个或更多 CPU，而每个 CPU 又可能包含几个计算核，而我还是称这

个盒子是一台“计算机”。当我进行一个Web搜索时，搜索请求会被送到某搜索门户，在那里包
[10] 括10 000多个计算单元（每一个类似于一台PC）的“服务器场”（server farm）将处理该搜索请求。当一个服务器场协同完成处理时，称之为一台超级计算机，即还是一台计算机。

因此，计算机又变大了，由许多较小的单个计算单元组成。一个许多计算机组合起来协同工作的突出例子是巴塞罗那的超级计算机，即MareNostrum，安装在巴塞罗那Torre Girona教堂，如图1-8所示。



图1.8 由位于Torre Girona教堂内的巴塞罗那超级计算机中心开发的MareNostrum设施（图片由巴塞罗那超级计算机中心提供，www.bsc.es）

1.5 未来

小型化过程还将继续，就来越多的产品、器件和系统包含嵌入式计算机。而且没有任何征兆显示这种趋势会消亡。计算机速度会继续增加，最终，存在一个很美妙的历史轨迹，请看表1-2中的数字，它显示出计算机如何从最早期开始在计算速度上不断进步——当然也要记住，计算机的定义已经改变了若干次。

表1-2 从始至终，计算机在计算速度上那令人瞩目的进步
（该数据由美国田纳西大学的Jack Dongarra教授负责提供）

年份	每秒浮点运算次数（FLOPS）	年份	每秒浮点运算次数（FLOPS）
1941	1	1987	1 000 000 000（1 GigaFLOPS，1 GFLOPS）
1945	100	1992	10 000 000 000
1949	1 000（1 KiloFLOPS，1 KFLOPS）	1993	100 000 000 000
1951	10 000	1997	1 000 000 000 000（1 TeraFLOPS，1 TFLOPS）
1961	100 000	2000	10 000 000 000 000
1964	1 000 000（1 MegaFLOPS，1 MFLOPS）	2007	478 000 000 000 000（478 TFLOPS）
1968	10 000 000	2009	1 100 000 000 000 000（1.1 PetaFLOPS）
1975	100 000 000		

未来考察一下该进步的幅度。我们可以看到一种难以置信的、持续的性能改进。根据这些历史数据，我们也许可以放心地把小型化和性能改进过程完全交给像ARM、Intel和AMD这样的大
[11] 公司。

能吗？尽管存在小型化趋势，我们同时也看到（超级）计算机在不断变大且更加抢眼。并行计算已经成为构造世界上最快的计算系统时采用的主流技术。大型机时代也许又会回来。所不同的是大型机可以坐落在其他国家，用户可以通过因特网和无线网络访问到这台大型机。大型机也许会安装在寒冷的国家，其散发出的热量可以用来温暖附近的房屋。

由于将计算机和实际使用计算能力的地方分离开的技术已经存在且已很成熟，但对无线通信连接而言可能存在例外，因此这一改进过程中的关键因素变成了服务和软件。

然而，这并不意味着现在要放弃对计算机及其体系结构的改进（那意味着你可以不继续往下读了）。但这确实意味着重点可能发生变化，即从大的、强的转向小的、移动的，从大规模 [12]
的数值计算转向嵌入式和专用计算。

回到本书的预备目的，在计算机系统上工作的工程师一直被问到这样的问题："在我的系统中应该使用什么处理器？""在我的系统中如何才能使处理器正确工作？"本书将提供回答这两类问题所需的背景知识。另外，这些知识还可以用来回答新的问题，例如："我是否应当专门为我的系统开发一个新的处理器，如果需要，如何做到？"或"我是否应使用简单 CPU 且连接到后端服务器，或在内部完成全部处理？"



图 1-9 Apple 的 iPhone，包括 8 个分离的 ARM 处理器，带有完整的触摸屏（图片来源于 Apple 公司） [13]

尽管存在许多像 MareNostrum 这样的超级计算机，但当今计算基本上可归入嵌入式工程领域，即嵌入式器件和消费电子器件中的普适计算技术。考察一下图 1-9 所示的 iPhone，它包含 9 个分离的微处理器，其中 8 个是基于 ARM 的。因此，对于未来是怎样的这一问题的答案是，我们可以预测两种趋势：一种是更少但更大的大型计算机集群；另一种是更多但更小的个人化计算器件。

这也许能够使你意识到要学习一些普适存在的 ARM 计算机技术。

1.6 小结

你可能不会去缔造世界上最快的超级计算机（也许你会，谁知道呢？），但你很有可能会去设计嵌入式系统或进行嵌入式程序设计。

本章介绍了计算技术发展历史：不断向前进步，在技术和思想上有许多大的飞跃，但也存在成百万的小步伐改进。Isaac Newton 在给他的竞争对手 Robert Hooke 的一封信中写道："如果我能够看得更远，是因为我站在了巨人的肩膀上……"

对于大多数计算机设计者来说这是完全正确的。站在巨人肩膀上的最好办法是利用现有的计算机设计下一代计算机。

基于这样的历史观察和在这个领域继续前行的信心，现在应当做的是从过去几十年的计算机系统设计者那里学习技术。下面的章节将进行这个过程。首先不考虑计算机性能提升问题，而是学习无论是台式机还是嵌入式都需要的基本的基础技术。然后，我们花一些时间进一步了解嵌入式系统，并学习构建我们自己的嵌入式 CPU。最后，我们进一步考察一下计算机世界中未来被寄希望的技术。 [14]

第2章

Computer Architecture: An Embedded Approach

基础知识

本章介绍一些背景知识，用来解读现代中央处理单元（CPU）的设计。本章我们将关注计算机组成与分类的形式化方法；定义大量的描述计算机系统的术语；讨论计算机算法以及数据表示；了解几个组成结构的模块，这些模块在后面分析计算机系统时会再次遇到。

2.1 计算机组成

计算机是由什么组成的？这些组成元素之间是如何连接的？为了回答这些问题，我们首先应认识到计算机的结构内部存在各种各样的可能性。比如今天的台式计算机，许多原来连接在CPU周围的外设模块现在都封装在一个集成电路芯片之内，这并不是先驱者们所认定的计算机。然而，一直以来计算机中的大部分模块都仍然存在，即使它们不是立即就能被识别出来。在嵌入式系统中这个趋势更为明显：将几乎所有必需的功能都集成在一个芯片上的片上系统（SoC）现在已占据主导地位。

其次，并非所有的计算机都以同样的方式构成，或有同样的需求。它们的尺寸范围从一个房间大小的超级计算机到类似于手表的个人数字助理（PDA）或更小。

尽管可能性各种各样，但大多数系统都包含功能相似的模块。这些模块在布局上是放在CPU芯片的内部还是外部，取决于设计和成本方面的考虑，它们之间的互连（包括内部的和外部的）通常是并行总线，总线的宽度和速度也取决于设计或成本的考虑。

现在的设计中，每个功能模块可能都有多个副本或者模块间有多个互连总线。

正是有这些差异，需要按照某种方式定义不同的体系结构。最早是在1966年，Michael Flynn
15 首先提出了一种描述计算机系统的分类方法。

2.1.1 Flynn分类法

现在广泛使用的Flynn分类法，是基于指令流和数据流的数量对计算机进行分类的方法。

一系列修改那些流经数据处理单元的数据（数据流）的命令，可以被认为是一个指令流。图2-1显示了四种不同情况，包括：

- **单指令流单数据流**（SISD）——传统的计算机包含单个CPU，它从存储在内存中的程序
16 那里获得指令，并作用于单一的数据流（本例中就是一个指令处理一条数据）。
- **单指令流多数据流**（SIMD）——单个的指令流作用于多于一个的数据流上。例如有数字4、5和3、2，一个单指令执行两个独立的加法运算：4+5和3+2，就被称为单指令流多数据流。SIMD的一个例子就是一个数组或向量处理系统，它可以对不同的数据并行执行相同的操作。
- **多指令流单数据流**（MISD）——用多个指令作用于单个数据流的情况实际上很少见。这种冗余多用于容错系统。
- **多指令流多数据流**（MIMD）——这种系统类似于多个SISD系统。实际上，MIMD系统的一个常见的例子是多处理器计算机，如Sun的企业级服务器。

虽然Flynn设计此分类法的初衷是描述处理器级的布局，但它也同样可以适用于处理器内部的单元模块。例如，Intel Pentium处理器上的多媒体扩展（MMX）和以后的单指令多数据流扩展（SSE），都是一个SIMD的例子。它允许发出一个单一的指令，去操作多个数据项（如8对数据

同时执行不同的加法)。我们将在后面的4.7节一起介绍MMX与SSE。

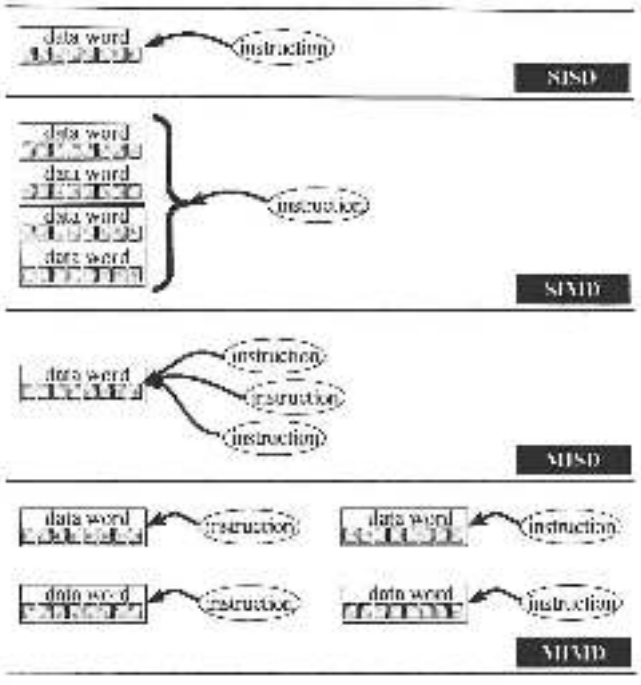


图2-1 Flynn分类法中SISD、SIMD、MISD和MIMD处理示例。这4个分类显示了在某个时钟点上执行的指令和数据之间的关系(因为这个分类是按照指令流和数据流的共同特征进行划分的)

2.1.2 连接方式

另一种对处理器体系结构的常见描述是基于处理程序指令和数据是共同处理还是单独处理。

- 冯·诺依曼体系结构的系统共享数据和指令的存储以及传输资源。许多现代计算机都属于这一类,它们在共享内存中存储的程序和数据,并使用单一的总线将程序和数据从内存传输到CPU。共享总线带宽往往意味着系统性能受限,但其优点是设计简单、成本低廉。
- 哈佛体系结构的系统对数据和指令有单独的存储和传输。由于指令和数据可同时传输,这种系统能够提供高性能。
- 其他的体系结构包括带有多个专用总线的系统(如ADSP2181内部总线),地址总线共享数据/指令和数据总线是独立的或类似的结构。第4章还将进一步介绍并详细阐述内部总线结构。

一些CPU虽然通过一个单一的总线共享内存接口,却宣称是哈佛体系结构处理器。如DEC或Intel的StrongARM处理器。在这里,StrongARM处理器内部是哈佛体系结构,因为它包含了独立的内部数据和指令高速缓存模块,但它的外部是冯·诺依曼连接方式。 [17]

2.1.3 计算机结构层次视图

有时可以把计算机系统看做是相互联系的多个层次。图2-2描述了层与层之间的连接操作,也可以看做是层次化的操作。

从下往上,任何CPU都可以被看做是执行逻辑运算的门集合。这些逻辑运算是通过微程序或状态机的控制来完成所需功能的,微操作序列是由指令集中的一条或多条指令指定。指令可以直接来自用户程序,或预定义的基本输入/输出系统(BIOS),或操作系统。

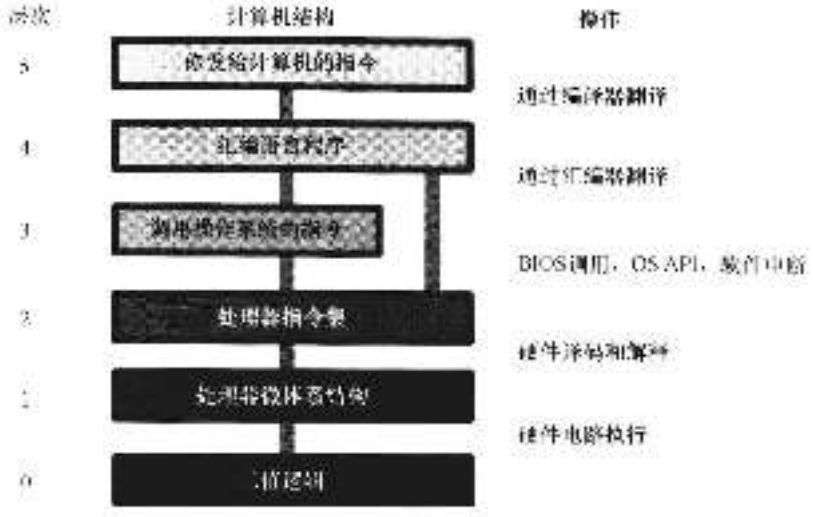


图 2-2 计算机组织与结构层次视图

有趣的是，这个层次模型很像用于计算机软硬件的开放系统互连（OSI）参考（OSI 模型在附录 B 中有介绍）。

2.2 计算机基本原理

本书所描述的计算机系统，如 2.1.1 节中讨论的 SISD 计算机，通常包括一些由总线连接起
[18] 来的功能单元。本章将简要介绍这些单元，在后续的章节中再详细论述。

- 中央处理单元（CPU）——通过对指令的解释以及内置的行为来控制计算机的部件。它可以处理输入/输出功能，完成算术和逻辑运算（也可以说包含一个 ALU）。最近，CPU 已用来指代一个物理的集成电路芯片，在有些产品中，它包含了作为一台独立计算机所必需的所有部件。
- 算术逻辑单元（ALU）——CPU 中的这个部件完成加、减、与、或等简单的算术逻辑运算。它是一个异步单元，从并行连接的寄存器或总线输入两个数据，输出直接连接到寄存器或者是通过三态缓冲器连接到总线。此外，它有一个控制输入用来选择执行哪个运算，以及一个状态寄存器接口。在现代处理器的芯片内部，它仅被用来处理定点二进制（偶尔是 BCD 码）数值。
- 浮点运算单元（FPU）——或者在片上，或者作为外部协处理器，负责浮点的算术运算。大多数现代 FPU 支持的浮点格式是 IEEE754 标准。它相对较复杂（可能包含数十或数百个指令周期来完成一个计算），通过专用浮点寄存器连接到主 CPU。
- 内存管理单元（MMU）——该单元在处理器寻址空间和真正的物理存储之间提供了一个抽象层。这种抽象被称为虚拟内存。MMU 把处理器需要访问的虚拟地址转换为一个真正的内存中的物理地址。处理器通常看到的是大片连续的内存地址空间，这是因为 MMU 屏蔽了物理存储的组织，真正的物理存储空间可能是由不同尺寸（更大或更小）、不连续的部分 RAM 和部分的硬盘构成。

另外，我们还需要讨论一些概念，在后面详细介绍这些概念之前先在这里进行定义：

- 寄存器——直接连接到 CPU 内部总线的片上[⊖]存储单元，访问速度非常快（通常是一个

⊖ 尽管这些都是独立的概念，但现在由于空间的便利性和速度的考虑都集成到了一个芯片上。

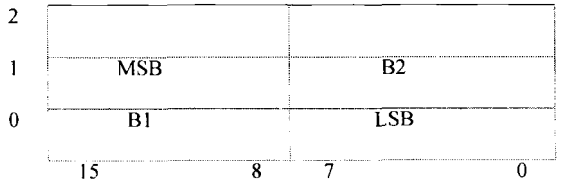
指令周期）。它容易和某些 CPU 的片上内存或 picoJava Ⅱ处理器中的堆栈混淆。

- 三态缓冲器——用于驱动总线的使能设备，通常位于寄存器和总线之间，由寄存器控制何时驱动总线。三态中的两种状态是驱动总线电压为逻辑高或逻辑低；第三种状态为高阻态，这意味着该设备不驱动总线。 [19]
- 复杂指令集计算机（CISC）——将能想到的所有有用的操作都放到 CPU 硬件中，不必担心有多大、多耗电或使 CPU 变慢，那么最终得到的就是一个 CISC 机器。早期的 VAX 机器，据说包含超过 2000 个时钟周期的指令。
- 精简指令集计算机（RISC）——CPU 的性能受其内部最慢组件的限制以及芯片面积的限制。基于 80% 的指令只使用了 20% 的执行时间而剩下的 20% 的指令却占用了 80% 的芯片面积这样一个前提，CPU 被精简到只包含这 80% 最有用的指令。有时，一个 RISC 的定义是指“支持一个小于 100 条的指令集”。一个值得注意的新兴趋势是用一个 RISC CPU 核去模拟 CISC 机。
- 指令周期——指一条指令被取指、译码、执行到返回结果的时间，可能是一个或多个主时钟周期（由外部晶振产生的）。对 RISC 处理器，指令通常都是在单个时钟周期内执行完；对 CISC 处理器，有些指令会需要很长时间。
- 大小尾端——大尾端（big endian）指高字节放在前面，常被用在像 68000 或 SPARC 这些处理器中。小尾端（little endian）指低字节放在前面，用于 Intel x86 系列中。某些处理器（如 ARM7）允许进行大、小尾端的切换。 [20]

不幸的是，在现代计算机中，内存的各种宽度使尾端问题变得复杂。如果所有的内存都是字节宽度那就简单了，但现在增加了难度。给定一个未知的系统，可能会比较容易先判断是否是小尾端，如果不是，再归类为大尾端。在框 2.1、框 2.2、框 2.3 和框 2.4 中详细讨论了这个问题。 [21]

框 2.1 尾端实例 1

问：将一个 32 位宽的字存储在一个 16 位体系结构的存储系统中，下图所示为存储字最低有效字节（LSB）、第二字节（B1）、第三字节（B2）和最高有效字节（MSB），这是小尾端还是大尾端？

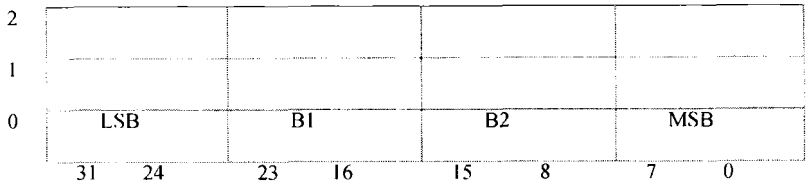


图中，内存的行号（16 位字）在左边，位的位置标在下面。

答：首先检查是不是小尾端，我们确定最低字节的内存地址然后向上计数。本例中，最低字节的地址行号为 0，最低位也是从第 0 位开始。内存中的下一个字节开始于第 8 位，仍然是在第 0 行。其次是 1 行 0 位，最后是第 1 行的第 8 位。从最低字节的地址的内容开始向上数，可以得到｛LSB、B1、B2、MSB｝。由于这个顺序遵循从最低字节到最高字节的格式，因此它一定是小尾端。

框 2.2 尾端实例 2

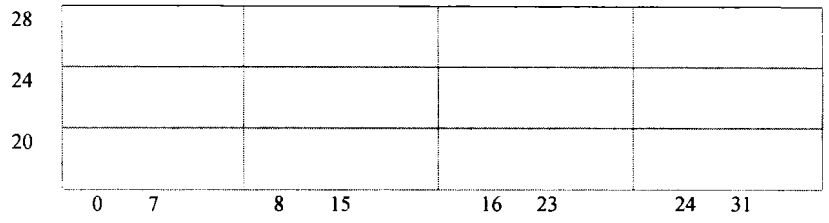
问：一个 32 位的字存储如下。这表示的是小尾端还是大尾端？



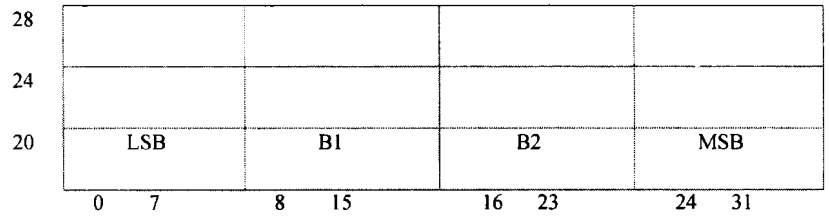
答：首先，确定最低字节的内存地址。这显然是从地址第0行第0位开始，其次是第0行第8位，依次下去。从最低字节到最高字节依次写出内容，我们得到序列｛MSB，B2，B1，LSB｝。这个顺序不遵循由最低字节到最高字节的格式，它不是小尾端。因此，它一定是大尾端。

框2.3 尾端实例3

问：给定内存映射如下图所示，在图中填入以小尾端格式表示的一个32位数字的MSB、B1、B2和LSB字节。



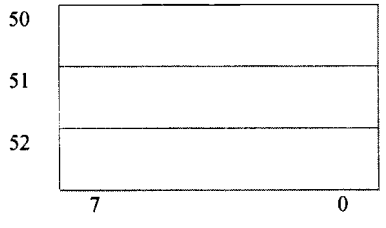
答：小尾端通常都比较容易：LSB是在最低字节地址，然后顺着内存向上数到MSB。首先，需要确定内存中最低字节地址的位置。本例中，位号标注在表格的下方——它们从左边开始向右递增，因此最低字节的地址是指第20行第0位，下一字节是第20行第8位，依次向上。最终的结果应该是：



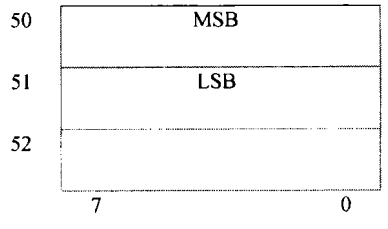
注意：再关注一下这个地址。它的位置是不连续的，行号不是一个一个增加的（像其他的例子那样），这些地址是4字节递增的。这表明内存是按字节编址，而不是按字编址。这是典型的ARM处理器，内存按字节独立编址，尽管拥有32位宽的存储器。

框2.4 尾端实例4

问：给定内存映射如下图所示，写出用大尾端格式表示的16位数字的MSB和LSB字节。



答：同样，需要确定在图示的内存中哪个是最低字节的地址，然后将MSB字节放进去，因为是大尾端模式。本例中，内存是从上向下编址的——某些处理器制造商所用的共同的格式，上面是低地址，依次向下计数。由于内存是字节宽度，因此很容易写出答案如下：



2.3 数字格式

现代各种计算机在算术和逻辑数据处理上其方法都是相同的：利用相同的数字格式，可以按照同时处理的数据位数（数据宽度）对计算机进行分类。它们甚至往往采用类似的技术来处理数字。早期的计算机并非如此，很多非标准的数据宽度和格式丛生，其中大部分已成为历史。

可以说，大约有7种（或更少的）二进制数字格式今天仍然在使用。框2.5讨论了什么是数字格式，但为了让我们在后面的章节中更好理解硬件的处理，需要在这里回顾一下本书中会遇到的各主要格式。

框2.5 什么是数字格式

我们都知道十进制格式，无论是整数123还是小数1.23，都是以10为基数的例子。

事实上，有无限多种方式来表示一个数字（有无限多种基数），但其中只有少数是常见的。除了十进制格式以外，十六进制格式（基数为16）经常会用在软件中，而二进制格式（基数为2）在常用硬件中，并在本书所有的例子中采用。

2.3.1 无符号二进制

在无符号二进制格式中，一个数据字中每一位的权值是和其位置相关的2的相应次幂。例如，8位二进制字00110101b相当于十进制的53。末尾的b表明它是一个二进制数字；从右向左读，数值计算过程为：

$$1(2^0) + 0(2^1) + 1(2^2) + 0(2^3) + 1(2^4) + 1(2^5) + 0(2^6) + 0(2^7)$$ 23

在一般情况下，一个n位二进制数x的值v可表示如下，其中$x[i]$是从右向左读的第i位（从第0位开始）：

$$v = \sum_{i=0}^{n} x[i] \cdot 2^i$$

人们对于无符号二进制格式只需稍加练习就很容易阅读，并能有效地通过计算机处理。

2.3.2 原码

这种格式保留最高有效位（MSB）来表示正负（称为“符号位”），然后利用其剩余的低有效位的无符号二进制数值来表示绝对值的大小。按照惯例，MSB为0表示正数，为1表示负数。

例如，4位带符号数1001表示-1，8位数值10001111b相当于十进制的-15。

2.3.3 反码

这种格式在很大程度上已经被补码表示法取代，但偶尔还是可以看到。这种格式同样是用MSB表示符号，其余位表示绝对值。但是，如果该数字为负（即符号位为1），则绝对值位的值是反的。

例如，8位的反码1110111等于十进制的-8。

2.3.4 补码

补码（two's complement）无疑是现代计算机中最常见的数字表示格式。由于它的高效性，补码已经取得了相当的优势：无符号数算术运算的硬件电路同样可以用于补码运算。补码表示同样用MSB表示符号，正数的表示类似于无符号数的二进制形式。然而，负数的补码的数值位是在反码的基础上加1（框2.6提供了用此方法求“一个数字的补码”的二进制例子）。

框 2.6 负数的补码

负数的补码很容易从绝对值的反码加 1 得到，例如，求 -44 的补码：

先写出 +44 的 7 位二进制数：　　010 1100

然后，各位取反（得到反码）：　　101 0011

最低位加 1：　　101 0100

最后，加上符号位（1 表示负数）：　1101 0100

如果你不习惯写二进制数，可以试着把它分成 4 个一组来写。这样就很容易按列划分，有助于转换成十六进制（因为二进制的每 4 位对应十六进制的 1 位）。

例如，由二进制数字 1011 表示的 4 位补码相当于 -8 +2 +1 = -5，8 位的补码 10001010 相当于 -128 +8 +2 = -118。

无疑补码比上面提到的一些其他格式的表示法更难读懂，但是相对于降低硬件复杂度来说，这是一个很小的代价。框 2.7 给出了补码的一些例子，包括正数和负数的补码。

框 2.7 数的转换示例

问 1：写出十进制值 23 的 8 位二进制补码。

答 1：我们先写出 8 位补码每一位的权值，从左开始连同符号位一起。

-128	64	32	16	8	4	2	1

只有负数时才设符号位，本例中是正数，所以填 0；接下来，看权值为 64 的位，如果我们的数字大于 64 则会在这位写 1，但 23 不大于 64，所以这位写 0；同样，看权值为 32 的位，现在写下了：

0	0	0	16	8	4	2	1

到权值为 16 的位，数字 23 比 16 大，因此我们在 23 中减掉 16 得到余数 23 - 16 = 7，并在权值 16 这栏写下 1。

接下来我们再比较余数和权值 8，余数比 8 小，所以在权值 8 这栏写 0；再比较权值 4，余数大于 4，因此计算新的余数为 7 - 4 = 3 并在权值 4 这栏写下 1；按此规律，接着权值 2 和 1 这栏都写下 1。得到最终结果是：

0	0	0	1	0	1	1	1

问 2：写出十进制数 -100 的 8 位二进制补码。

答 2：同样看上面的这行数字，我们知道负数需要在权值为 -128 的栏中填写 1。计算减法 -100 -(-128) 或 -100 +128 得到余数 28。后续的计算正常进行——在权值 64 这栏写 0、32 这栏写 0、16 这栏写 1，得到新余数 28 - 16 = 12。继续在权值 8 这栏写 1，余数为 4，在权值 4 这栏写 1，其余的位写 0。得到：

1	0	0	1	1	1	0	0

注意：对于补码表示，能够一目了然地观察到该数是否为负数（最高位是 1），以及是否为奇数（最低位是 1）。

24 ~ 25

2.3.5 移码（excess-*n*）

我们后面讨论浮点数的时候会看到这种表示法。该格式采用无符号值 $v+n$ 来表示数 v。例如 8 位的移 127 表示法，可以表示 -127 ~ +128 的数字（以二进制位的方式存储，看起来就像无符号数 0 ~ 255）。

这种格式对于学生而言可能会有一点儿疑惑，比如 8 位移 127 格式的二进制码 00000000 等于 -127（这是无符号二进制的值，本例中需要用 0 减去 127）。看另一个例子 11000010，此二进制码的值应该是 128 +64 +2 = 194，但由于是移 127 表示，因此需要从 194 中减去 127 得到十进制的 67。

2.3.6 BCD 码

BCD 码（Binary-Coded Decimal）曾广泛用于早期的计算机中。由于每个十进制数字（0 ~ 9）

都对应到 BCD 中由一组 4 位二进制编码表示，使得人们使用时很容易阅读。如十进制 73 表示成 BCD 码就是 0111 0011。由于 4 位二进制可以表示 0 ~ 15 的值，因此有些二进制位的组合在 BCD 码中没有用到。BCD 码已最终被取代，因为其既不能高效存储又不易于硬件实现。

2.3.7 定点数表示法

实际上定点数表示法可应用于任何一种二进制表示法（也适用于十进制，参见框 2.8），但在计算机体系结构领域通常用于无符号数或补码表示。定点数表示的数字在概念上可以严格地定义为，采用一种按比例缩放的加权模式代替通常的（即最低位是权值 2^0，下一位是 2^1，第三位是 2^2，依次类推）位权模式。在一些数字信号处理（DSP）电路中，定点数表示法也被称为 Q 格式。定点数表示法通常用（$m.n$）的格式描述，其中 m 是假定的基之前的位数（十进制的基称为小数点，但在处理其他进制的数时，我们不能称它为一个“十进制”小数点，所以我们称它为基（radix)），n 是基（小数点）后面的位数。 26

框 2.8 二进制是一种定点数格式吗

二进制并没有什么特殊的，它只是基数为 2 的一种表示数的方式，而不是我们熟悉的基数为 10 的方式（十进制）。

我们用十进制可以像写整数（如 19）那样写小数（如 9.54），同样我们可以用任何其他基数表示方式像写整数格式那样写定点数格式。到目前为止我们只看到了整数的二进制格式，然而二进制定点数格式也非常重要，它被广泛用于如数字信号处理领域。

两个 8 位的二进制数按照无符号格式和（6.2）格式分别表示如下：

无符号	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
（6.2）格式	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

更多二进制定点数格式的例子可参考框 2.9。

无符号数或补码的定点数表示的好处是在硬件实现上对其值的处理和对非定点数表示的处理方式一样，小数点只是在程序上的假设而已。

框 2.9 定点数格式实例

问：把十进制数 12.625 改写成一个（7.9）定点数格式的二进制补码。

答：首先确定（7.9）格式的每一位位权：

-64	32	16	8	4	2	1	1/2	1/4	1/8						

这里由于空间原因位权低于 1/8 的已经被删除了。接下来，因为是正数，所以权值为 -64 的这一位写 0。然后，按照标准补码表示（或无符号数）的方式从左向右依次扫描每一位，使用上面给出的权值。

得到 12.625 = 8 + 4 + 0.5 + 0.125，因此结果为：

0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0

2.3.8 符号扩展

符号扩展是指将一个指定宽度的有符号补码数的位宽扩大。例如，把一个 8 位的数字转换为 16 位。虽然由程序员显式指定该操作只是偶尔发生，但在加法或乘法运算中却是很常见的一个操作。 27

可以通过把一个 4 位补码转换为 8 位的例子来解释什么是符号扩展。首先，写下 4 位二进制补码数 1010。

如果考虑是有符号数，我们知道 4 位数的位权分别是 [-8，4，2，1]，而 8 位数的位权分

别是 [-128, 64, 32, 16, 8, 4, 2, 1]。对于4位数来讲，1010的值显然是

$$-8+2=-6$$

如果变成8位数时只是简单地在4位数前面补0，即00001010，那么对照这8位数的权值来看，这个值就是

$$8+2=10$$

结果显然是错的。如果注意到负数需要设置符号位，进而简单地把符号位变为1，即10001010，那么这个值又变为

$$-128+8+2=-118$$

结果还是错的。实际上，为了能够正确地从4位扩展到8位，不仅需要将最高的符号位设置正确，而且每一个增加的位（原始数据MSB位左边的每一位）都必须设置成和原始数据的MSB位相同。因此符号位被扩展得到**11111**010，其值为

$$-128+64+32+16+8+2=-6$$

得到了正确的结果。符号扩展的另一个例子在框2.10中给出。

框2.10 符号扩展实例

问：写出-4的4位补码表示，把MSB位向左复制4次，然后读出该8位补码的结果。

答：1100(-8+4+0+0)

MSB位是1，向左复制4次得到**11111**100。

读出这个8位有符号数：(-128+64+32+16+8+4)=-4。

进一步思考：对一个正数（如3）重复上述练习。该方法是否同样适用于正数？

对于正数补码来讲符号扩展显然没有难度，但此规则仍然适用（它没有任何影响，却使硬
28 件设计变得简单，因为该规则是适用于所有的数而不是部分的数）。

2.4 算术运算

本节讨论能够执行两个二进制数加法和减法运算的硬件。几乎所有的处理器，此功能都是由算术逻辑单元（ALU）完成的，它同时也处理与（AND）、或（OR）、非（NOT）等基本的逻辑运算。ALU作为CPU的功能单元将在后面的4.2节讲述。

2.4.1 加法

二进制运算是按位完成的，相邻的低位计算可能会产生进位。在硬件上，一个全加器完成两个加数位和一个进位输入的加法，产生一个带进位输出的结果。

图2-3是一个全加器的符号示意图，其中每个箭头代表一个逻辑位。半加器类似，但没有进位输入。

2.4.2 并行进位传递加法器

要构建一个8位并行加法器，通常使用8个全加器，每一个输入位对应一个，虽然至少有一个最低位可以使用稍微简单的半加器，并行加法器如图2-4所示。

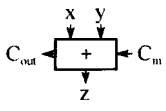


图2-3 一个全加器，两个输入位和一个进位输入相加，得到一位输出和进位输出

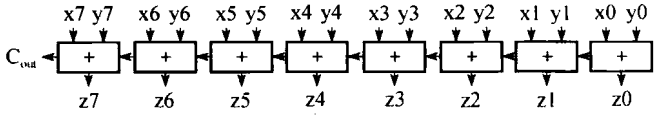


图2-4 进位传递加法器或行波进位加法器，由一串全加器和一个半加器构成

在图2-4中，x[7:0]和y[7:0]表示两个字节的输入，z[7:0]表示一个字节的输出，C_{out}是最终的进位。对于无符号数的加法，当C_{out}为1时表示计算的结果超出了8位的表示。例如，我们知道8位能表示的最大无符号数是$2^8-1=255$，如果两个较大的数如200和100相加，结果（300）就无法用8位来表示。这种情况下，进位就会被置1，并且结果的值（z）为余数300－256＝44。

因此，最前面的C_{out}同时用来作无符号数溢出标志：如果计算完成后它被设置为1，表明当前的加法器位数不够表示其结果。进一步的考虑见框2.11。 29

框2.11 读者练习

当进行有符号数的补码加法时，最高位的C_{out}信号将会怎样？

1. 试着手算一个4位的加法器，4位补码表示数的范围为－8～＋7。
2. 试着计算一些加法，如2＋8＝?，2＋(－8)＝?，7＋7＝？以及（－8)＋(－8)＝?
3. 你对C_{out}信号有何结论：对有符号数加法，其意义和无符号数加法时一样吗？

这种加法机制在几乎所有的二进制加法器中都较常见。虽然并行加法器似乎是一个相对高效的结构，甚至和人们手工（或者使用算盘）计算二进制加法的方式很相似，但它面临的问题就是进位传递的速度非常受限，这阻碍了它在大多数微处理器ALU中的使用。

鉴于加法器的两个输入数据是同时给出的，那么加法器的速度可以通过计算其输出所用的时间长短来衡量。加法链中的每个全加器或半加器相对是较快的：（现在的硬件系统中）给出输入数据和进位输入后几个纳秒即可得到进位输出和结果。问题是，最低位的半加器（adder 0）必须在下一位的计算（adder 1）开始之前就完成计算，因为adder 1需要adder 0提供进位才可以完成其计算，而该进位只有当adder 0计算结束时才有效。然后adder 1再把它计算得到的进位提供给adder 2，依次类推。进一步沿着加法链向前传递，在输入进入加法器相当长一段时间后，adder 6才能将其产生的进位提供给adder 7。

一个完整的行波进位加法器的传输延迟计算实例见框2.12。这一点很重要，因为如果加法器是在一个同步系统中，那么传输延迟将成为系统的一部分而限制系统的最大时钟频率。

框2.12 实例

问：一个4位并行进位传递加法器的全加器和半加器如下：

从前一个数据输入（x或y）或进位输入到结果z的时间：15ns

从前一个数据输入（x或y）或进位输入到进位输出的时间：12ns

如果输入的加数x[3:0]和y[3:0]在时刻0给出并稳定住，经过多长时间加法器可以产生稳定而正确的4位输出？

答：从加法链尾部的最低位开始，adder 0在时刻0接收到稳定的输入，其结果z在15ns后准备好，进位在12ns后得到。adder 1需要低位的进位才能开始计算，所以在12ns时刻才能开始，需要24ns后才能产生给adder 2的正确进位，这样给adder 3的进位输入需要在36ns时刻得到。然后adder 3开始加法计算，它的输出z将在51ns(36＋15)时刻完成，而进位输出在48ns(36＋12)时刻完成。因此尽管这些一位加法器本身很快，但当连成链后，需要51ns才能计算出结果。

注意：前面提到的全加器或半加器“开始计算”可能有一点误解。实际上它们都是组合逻辑模块，输入状态的任何一个变化都会在某个延迟时间（本例中最多15ns）后改变输出。因为是组合逻辑，电路一直保持对输入数据进行处理，并且输出也一直是激活的。然而，根据规则，我们知道只有在给出正确的输入15ns和12ns后（分别对应结果z和进位输出），输出数据才能保证正确。

2.4.3 超前进位

为了加速前面提到的并行加法器，一种方法是对加法链上的每个加法器都尽量早点给出进位输入。

可以进行进位预测，这是一块能够直接计算输出进位值的组合逻辑。实际上，它可以同时为加法链上的每一个加法器提供进位值，其传输延迟约一个单独的全加器并不多。图2-5是一个3位加法器的进位预测逻辑。注意一下描述这些超前进位单元的逻辑方程非常形成是很有意义的（见框2.13）。

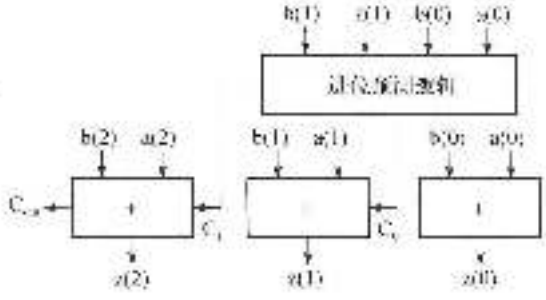


图2-5 超前进位加法器由几个全加器和进位预测逻辑构成

框2.13 读者练习

1. 写出一个全加器的等效方程。

2. 扩展刚上面这张图的3位加法器。

3. 重新写出 C_1 和 C_2 的方程，不根据输入产生（而不含任何一个进位输入）。实现产生 C_2 所需要的逻辑运算量较少。因此完成此计算所需的门传输延迟也较小。

4. 扩展到得到 C_3 的方程，需要多少引脚来实现？是不是比 C_2 的多？对于更多的加法器，你能不能看出这种方法的规律（从寄存器结构逻辑复杂性角度考虑）？

2.4.4 减法

和加法类似，减法也是按位计算的。当执行减法运算时，我们是否需要通过相邻位来考虑结果？答案是肯定的，但这里都是从高位的借位，而不是从低位的进位。这种借位存在和加法类似的问题。

考虑到硬件的可计算性，如果不是加法和减法可以互相转换（在多种数字格式中），就会需要一个专门的减法器。例如，一个十进制的计算 99 − 23 = 76，可以重新写成 99 + (−23)，得到相同结果。

虽然结果是相同的，但它是通过计算加法而不是减法来得到结果。而且第二个操作数的符号改变了。许多类似 ALU 的工作方式类似：只包含一个加法电路和一个操作数的符号转换机制。在2.3.4节我们看到，改变一个数在补码表示的符号非常简单：先各位取反再末位（LSB）加一。末位加一就当于把那个加法器的进位输入置1。

不用说，这是很容易实现的硬件电路，如图2-6所示的减法逻辑。在这个电路中，输入操作数y经过异或门用来完成按位取反（可以就像一个取反开关，如果一个输入端为高电平，那么另一个输入端的每一位将被取反，否则维持不变）。如果上路是执行减法，add/subtract 线保持高电平，减数被取负，且 C_{in} 也设置为高有效，从而实现末位加1的效果。

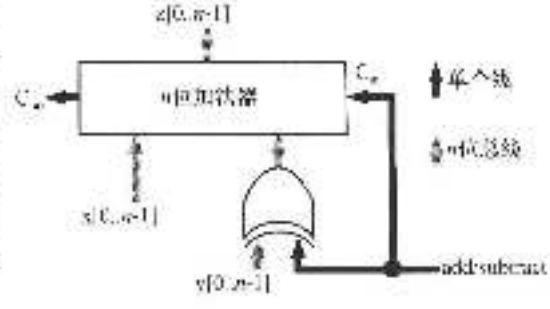


图2-6 减法逻辑主要包括一个带异或门的加法器

执行减法时还有一块逻辑需要说一下，就是溢出：进行加法运算时，可以使用最高位的进位 C_{out} 作为溢出标志。但这在减法运算中却不行，我们通过下面关于4位补码的例子来具体阐明：

0010 + 1110 = ?　　2 − (−2) = ?

$0010 + 1110 = 0000 + C_{out}$

显然结果应该是零没有溢出，但是最高位进位 C_{out} 也被置位。再看一个本应有溢出的例子：

$0111 + 0110 = ?$ $\quad 7 + 6 = ?$

$0111 + 0110 = 1101$ $\quad$ 结果 $= -3?$

结果是 -3 还是不对，应该是 13。由此可见，只用 C_{out} 来判断的电路显然不够，还应考虑到处理过程中值的变化。那就是符号位应该在加之前进行检测，在此基础上再对结果进行检测。这个计算量并不大，一个简单的查找表就够了：

正数 + 正数 = 正数

正数 + 负数 = 符号未知

负数 + 正数 = 符号未知

负数 + 负数 = 负数

对于两个异号数（一个正数和一个负数）的计算，结果的符号位是未知的，但却绝对不会产生溢出的问题（可以这样考虑：相加后负数会使正数的值变小，唯一不会变小的情况就是正加数是零，那结果就是输入的负加数，其输入数据本身不包含进位标志）。

对于两个正数相加，结果的符号位应该是 0，如果不是则发生溢出。对于两个负数相加，结果的符号位应该是 1，如果不是则发生溢出。因此看到只用 C_{out} 的值判断是否溢出是不够的。在多数处理器中，会提供一个单独的溢出标志，通过上面这种符号位的判断逻辑来进行设置。思考实例见框 2.14。

框 2.14 读者练习

请读者试着在本文基础上对减法进行扩展讨论。使用 4 位补码的有符号数格式，执行一些加法和一些减法。验证所有的减法 $a - b$ 都能用二进制加法 $a + (-b)$ 来实现；验证 C_{out} 信号并不能标识溢出状态。

执行加法 $-5 + -5$ 和 $-5 + -1$，观察符号位和结果的进位位。你能否总结一下，C_{out} 信号是无用的还是能够用来增加结果位的范围呢？

33

2.5 乘法

在早期的微处理器时代，在 CPU 内部用逻辑实现乘法太复杂，都是采用外部单元实现。即使当它终于挤进同一块硅片中时，也是非常拥挤：在早期的 ARM 处理器上，乘法硬件占用的硅片面积超过整个 ARM CPU 核。

后来，厂家调整了乘法的目标应用。对快速实时嵌入式处理器（可能是一个处理 GSM 手机语音编码的 ARM7），需要尽可能快地执行乘法，从而有了高速乘法器。相比于非实时处理器上低速、多周期的乘法器来说（如在 20 世纪 90 年代初的 ARM610，其目的是增强台式机的性能，或作为世界上第一个 PDA——Apple Newton 的核心），这显然会占用相当大的硅片面积。

有很多种方法可以实现 $m \times n$ 的乘法运算，各有不同的效率和不同的复杂度。典型的方法有以下几种：

1. 加法迭代（将 m 累加 n 次）；
2. 部分积移位加；
3. 将 n 拆分成一系列数的加法，再对 m 左移；
4. Booth 和 Robertson 方法。

每一种都会在下面的小节中讨论。当然还有更深奥的方法，因为这是一个活跃的研究领域。有趣的是还有一些方法是通过估值而不是计算来实现乘法，或者是以损失精度为代价。比如，将

操作数转换到对数域再通过加来实现，或者使用替代数字格式或余数格式。

替代数字格式会在9.5节简要介绍，但当使用这样的硬件执行二进制计算时，有太多种替代以致无法把它们全部描述出来。

2.5.1 加法迭代法

实现乘法最简单的方法也是实现复杂度和芯片面积最小的方法，但这是以慢为代价的。整数相乘 $m \times n$ 的伪代码看起来就像：

```
set register  A ← m
set register  B ← 0
loop while (A ← A − 1) ≥ 0
           B ← B + n
```

由于包含一个 n 次的循环，执行时间就会和 n 相关。如果 n 比较小，那么结果B就会早点计算出来。

如果我们考虑一个32位整数，它可以表示超过20亿的值，那就会需要很多次的循环迭代，
34 因而意味着一个相当长的执行时间。

2.5.2 部分积方法

除了规模 n 的迭代方法外（像上面介绍的加法迭代方法），部分积是一个迭代规模为 n 的位数的方法。

对数字 n 的每一位依次进行检查，从最低位到最高位，如果这一位是1，将 m 左移到检测为1的这一位对齐，将得到的部分积进行累加。在乘法术语上，这两个数被称为*乘数*和*被乘数*，尽管我们知道对十进制数无论谁做乘数结果都是一样的，即 $(m \times n) = (n \times m)$。

下面是一个部分积的例子：

```
    1001     被乘数 9
    1011     乘数 11
  ------
    1001     （由于乘数的第 0 位 =1，将 9 左移到与第 0 位对齐）
   1001      （由于乘数的第 1 位 =1，将 9 左移到与第 1 位对齐）
  0000       （由于乘数的第 2 位 =0，将 0 左移到与第 2 位对齐）
 1001        （由于乘数的第 3 位 =1，将 9 左移到与第 3 位对齐）
 --------
01100011     乘积结果 =99（部分积的累加和）
```

如果要进行带符号整数的补码运算，情况会稍微复杂一些，首先乘数的最高位是符号位，其次会用到符号扩展（见2.3.4节）。

对于有符号的情况，所有的部分积都需要符号扩展，将其扩展到乘积的位宽（由于符号位占1位，所以默认是两个输入数据的长度相加再减1，例如，一个6位的有符号数加上7位的有符号数将需要12位来表示结果）。

由于每个部分积对应乘数的一位，且根据乘数的位权进行移位，那么最高有效位（MSB）对应的部分积是一个特殊的情况：位的权值为负，因此，这个部分积必须从累加和中减去而不是加上。图2-7显示了这个流程图，其中假设灰色的补码累加模块能够完成符号扩展。

为了更好地理解这一过程，最好是用这种方法手工做几个简单的二进制乘法的例子，读者可以仿照框2.15中的例子完成。

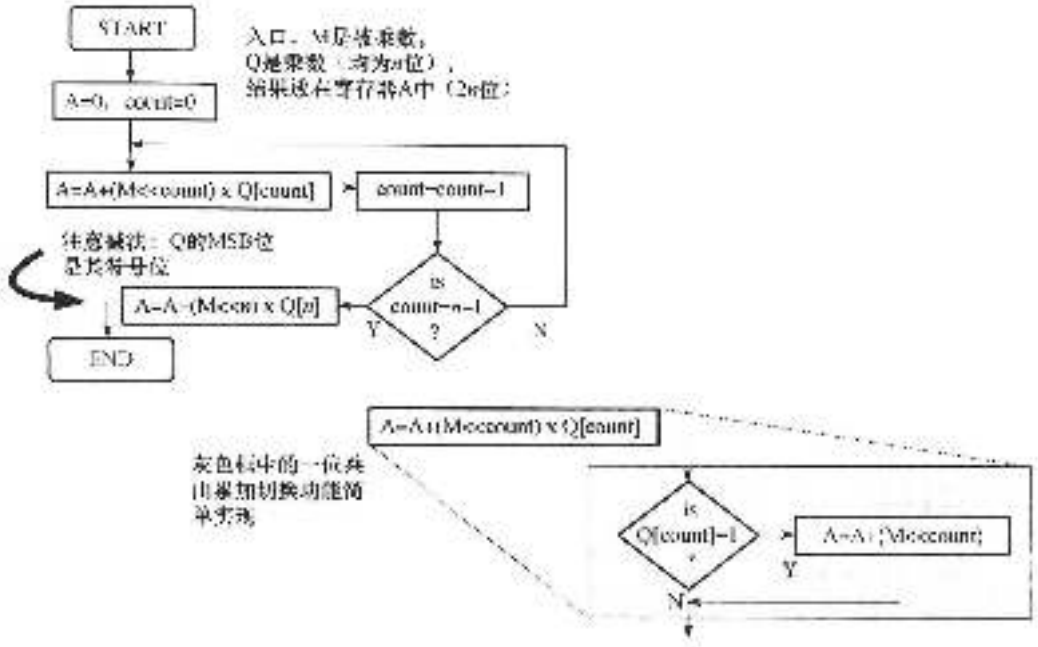


图2-7 部分积乘法流程图

注2.15 补码乘法实例

例如：-5×4（有符号数）：

```
      1011    被乘数 -5
      0100    乘数 4
  00000000    （由于乘数第0位=0，将0左移0位，且符号扩展）
 +0000000     （由于乘数第1位=1，将0左移1位，且符号扩展）
 +111011      （由于乘数第2位=0，将-5左移1位，且符号扩展）
 +00000       （由于乘数第3位=0，将0左移1位，且符号扩展）
 -11101100    结果 = -128+64+32+8+4 = -20
```

类似地，我们再看$4\times(-5)$（有符号数）：

```
      0100    被乘数 4
      1011    乘数 -5
  00000100    （由于乘数第0位=1，将4左移0位，且符号扩展）
 +0000100     （由于乘数第1位=1，将4左移1位，且符号扩展）
 +000000      （由于乘数第2位=0，将0左移2位，且符号扩展）
 -00100       （由于乘数第3位=1，将0左移3位，且符号扩展）
 -11101100    结果 = -128+64+32+8+4 = -20
```

但是此例中最后一行运算是减法，我们可以对要减的数连同符号位一起各位取反再加1（00100000→取反→11011111→+1→11100000），然后再把它和其他的部分积相加即可，如下：

```
  00000100
 +0000100
 +000000
 +11100
 -11101100    结果 = -20
```

我们考虑结果都是一样的。至此我们讨论了需要符号扩展的情况，以及当乘数为负时所出现的最后一行部分积要加方式的情况。

实际上，反方向运行部分积的累加可能会更有效（即向下循环而不是向上循环）。最好的情况是不需要以不同方式处理乘数符号位的部分积（因为这并不是累加，它只是加法运算之前累加器中的值，从而使得其符号标志在数据加载时就被处理掉）。

图2-8中的框图描述了用于无符号数的另一种部分积乘法器方法（此方法较容易扩展为补码格式），图中给出了一旦设置（加载操作数）完成各操作的执行顺序。

在复位阶段，将累加器的输出端Q复位为零，把乘数和被乘数加载到正确位置；步骤1，测
35 试乘数的最低位；如果是1（步骤2），则将被加数加到累加器（步骤3）；不管前面两个步骤是否满足条件，步骤4都将整个累加器右移一位。系统循环n次（该控制逻辑没有画出）结束，将结果保存在一个长寄存器中。

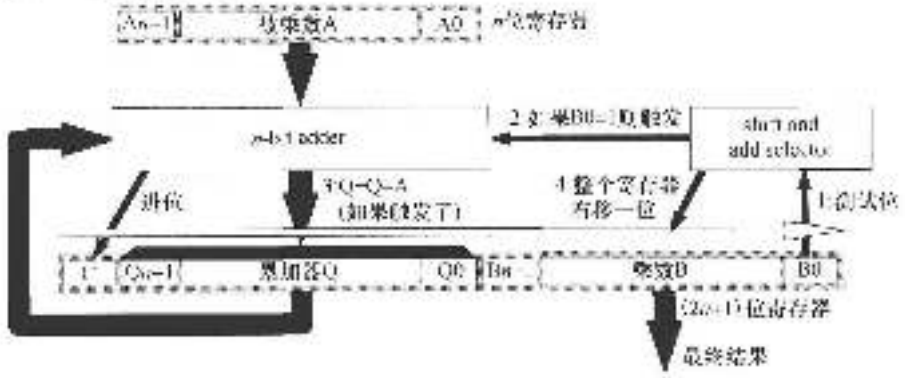


图2-8 用累加器实现的有符号数部分积乘法器控制框图

比较此方法和图2-7中流程图的不同，涉及寄存器个数、总线宽度、连接、开关、加法器大
36 小以及控制逻辑各方面。

有趣的是，这种乘法运算方法，也包括右移方法（表示一个数除以2），据说已经被俄罗斯农民使用了几百年，他们用此方法轻松地计算相当复杂的十进制乘法。该算法首先是将待乘的两个数A和B分别写在两列的标题栏上，我们以31乘以17作为例子：

$$B = 17, \quad A = 31$$

从上向下写，每一行都将B的值除以2，丢弃小数部分，直到值为1；与此类似再在A栏，
37 但改成每一行的值为上面的两倍：

B	A
17	31
8	62
4	124
2	248
1	496

接下来，把所有B列为奇数的行的A列的值加起来即可。本例中，B列只有17和1是奇数，因此最后乘积结果是31 + 496 = 527，显然是正确的。

注意本节给出这种方法绝不是唯一的硬件能够实现其部分积方法，也绝不是唯一可用的乘法方法（即便是在俄罗斯农民中）。

2.5.3 移位加方法

移位加方法基于这样一个事实：对二进制数来讲，左移一位相当于乘以2，左移两位相当于乘以4，依次类推。

应用这个属性执行乘法操作时将不可避免的遇到问题，当在此基础上进行迭代加法时，操作的次数将取决于乘数的具体值，而不是乘数本身的位宽。鉴于此，该方法在商业处理器的通用乘法器中很少见到。但当乘数是固定值并且近似于2的幂时，该方法显得非常有效。因此，该方法多用于数字滤波器（包含一系列乘法运算的器件），其乘数的值是预先确定的。

此方法也易于实现基于FPGA[⊖]的定点滤波器设计，因为在这类设计中，从一个加法器传到下一个就是简单地把两个逻辑单元用线连起来，右移可以通过把一个单元的输出位0，1，2，…连接到下一个单元的输入位1，2，3，…而轻松地完成。

2.5.4 Booth 和 Robertson 方法

Booth方法类似于部分积方法，从右向左扫描乘数的每一位，然后根据乘数位的值，对被乘数经过移位后的值进行加或减。不同的是在Booth方法中，乘数的位是两位两位进行检测，而不是一位一位。对该方法的一种扩展是4位并行检测，而Robertson方法是整个字节并行检测。

这些方法的优点是速度极快，但其逻辑会随着并行检测的位宽增加而变得复杂。

Booth方法的关键是定义一个规则，就是如何按照乘数的某两位的值对被乘数进行加或减。定义乘数中相邻的两位为X_i和X_{i-1}，从$i=0$开始扫描乘数，表2-1列出了相邻两位的所有组合及相应的规则。

表2-1 Booth方法中的规则定义

X_i	X_{i-1}	规则
0	0	无操作
0	1	加移位后的被乘数
1	0	减移位后的被乘数
1	1	无操作

38

当从累加器中加或减一个被乘数时，和部分积方法类似，先要将其左移到第i位的位置。框2.16和框2.17给出了这一过程的详细实例。

框2.16 读者练习

考虑9×10（无符号乘）：

```
    1001    被乘数为9
    1010    乘数为10
  ------
    0000    (i=0时，无操作，因为位对为末位0和一个隐藏的0)
   -1001    (i=1时，减去被乘数，因为位对=10)
  +1001     (i=2时，加被乘数左移2位的值，因为位对=01)
  -1001     (i=3时，减去被乘数左移3位的值，因为位对=10)
 +1001      (i=4时，加被乘数左移2位的值，因为位对=01)
  ------    (i=5及以后，无操作，因为所有位对=00)
```

因此，下面的累加就可得到结果：

```
 10010000
 -1001000
  +100100
   -10010
 --------
```

或者把减法转换成加法（方法见2.4.4节）：

```
  10010000
 +10111000
   +100100
 +11101110
 --------
 =01011010
```

⊖ FPGA：现场可编程门阵列（Field Programmable Gate Array），一个灵活的、可编程的逻辑设备。

得到乘积：

39 1011010 = 64 + 16 + 8 + 2 = 90（正确）

框 2.17 Booth 方法实例

考虑 −9 × 11（有符号乘）：

```
 11110111     被乘数为 -9
 00001011     乘数为 11
-11110111     (i=0 时，减去被乘数，因为位对 =10)
 0000000      (i=1 时，无操作，因为位对 =11)
+110111       (i=2 时，加上被乘数左移 2 位的值，因为位对 =01)
-10111        (i=3 时，减去被乘数左移 3 位的值，因为位对 =10)
+0111         (i=4 时，加上被乘数左移 2 位的值，因为位对 =01)
 000          (i=5 及以后，无操作，因为所有位对 =00)
```

因此，下面的累加就可得到结果：

```
-11110111
+11011100
-10111000
+01110000
```

或者把减法转换成加法（方法见 2.4.4 节）：

```
 00001001
+11011100
+01001000
+01110000
=10011101     +进位
```

得到乘积：

10011101 = −128 + 16 + 8 + 4 + 1 = −99（正确）

需要注意的是，当 $i=0$ 时，检测的两位分别是乘数位的最低位和一个假想位 0。因此当乘数的最低位是“1”时，被乘数一定要被减掉（即是按照“10”对待的）。在第二个例子（框 2.17）中就是这样。

还有两点值得提一下。首先，处理有符号补码格式操作数时，部分积必须进行符号扩展，这和完全的部分积乘法器是一样的。其次，从右向左扫描时，由于最右侧有一个假想位 0 存在，这意味着遇到的第一个非等值对（两位）一定是“10”，对应着一个减法。这条规律可能会有助于进行硬件实现。

尽管有人已经从事二进制运算很多年，本书的作者还是要提醒：在二进制加法中太容易犯
40 许多非常细小的错误了。如果你在考试中遇到这个题目，请仔细检查你的运算，第一次就答对并不像看上去的那么容易。

前面提到，用查找表方法可以把 Booth 方法扩展为同时检测 4 位，Robertson 方法又向前迈出了一步，建立了一个 8 位的查找表。这些方法其实常见于各种现代处理器，尽管它们会占用相当大的硅面积。

2.6 除法

许多年来，作为商品的 CPU 甚至是 DSP 都没有用硬件实现除法，因为其复杂性和占用的硅面积都太大。Analog Devices 公司的 DSP 及其他几种芯片中包括 DIV 除法指令，但通常这只是一个辅助硬件，通过采用非常基本的减法进行迭代而实现。

减法迭代

除法的过程就是判断被除数 Q 中包含多少个除数 M（结果是商 Q/M），因此可以简单地数一下 Q 减 M 可以减多少次，直到余数小于 M 为止。例如，计算 13/4，讨论下面的循环：

迭代次数 $i=1$，余数 $r=13-4=9$；

迭代次数 $i=2$，余数 $r=9-4=5$；

迭代次数 $i=3$，余数 $r=5-4=1$；

余数 1 小于除数 4，因此商为 3 余数是 1。

对二进制的计算过程是一样的，框 2.18 中给出的实例也许是最好的长除法例子。

框 2.18 长除法实例

考虑 23 ÷ 5（无符号除）。

首先以长除法的格式写下其二进制值：

```
101 | 010111
```

除数　　被除数

然后，从最高位（左）开始向最低位（右）扫描被除数的每一位，看在被除数中是否能“找到”除数。每一次，如果找不到则在被除数上面相应的位置写“0”，再看下一位。经过 3 次迭代，得到：

```
    000 （商）
101 | 010111
```

但是现在，在被除数的当前位置能找到 101，因此把 101 写在被除数的下面，并在被除数上面的相应位置写“1”。然后从被除数中减掉除数（在找到那一位的位置），形成一个新的被除数：

```
    0001
101 | 010111
  -    101
    ------
    000011
```

接下来，继续从左向右扫描，但是这次从新的被除数中寻找除数。本例中这次没找到；扫描所有位后，得到：

```
    000100
101 | 010111
  -    101
    ------
    000011
```

结果如上：商是 000100，余数是 000011。我们做的是 23 除以 5，期望的商是 4（正确）余数是 3（也正确）。

所以现在的问题是，如何进行有符号整数的除法？答案：大概最有效的方法是记下两个操作数的符号，把它们都转换为无符号数，执行完除法后再把正确的符号位加回去。除法的符号规则和乘法一样，只有在两个操作数异号的时候结果为负。

主流微处理器的除法过程如图 2-9 所示。仔细检查后会提出一些问题，如“为什么 A 和 Q 每次迭代都左移?”“为什么在循环中执行一个加法 Q = Q + M?”要回答这些问题，可以考虑这些操作是如何通过 CPU 里的寄存器执行的。这可以作为一个纸上练习，遵循算法的操作完成一个除法例子，可能是两个 6 位的数字。这样的练习有助于阐明系统是如何工作的。

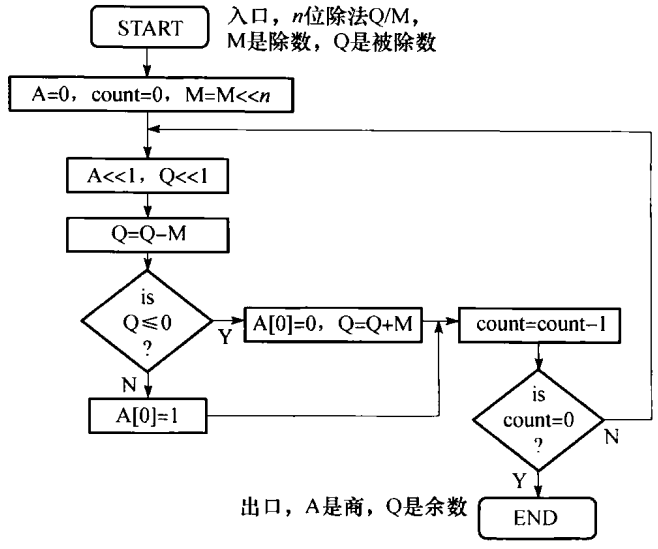


图 2-9 除法算法流程图

41～42 注意在算法结束时，寄存器 A 中保存了结果的商，余数保存在寄存器 Q 中。算法将迭代 n 次，n 是输入字的位宽。一如既往，完全有可能推导出其他不同的流程图，例如，有些方法甚至会从相反的方向扫描每一位并迭代。

2.7 定点数格式的运算

2.3.7 节介绍了使用 Q 格式表示的定点数表示法。尽管需要定点数表示的原因有很多，但其中一个主要原因是在数字信号处理中，有很长的数字滤波器，需要成百甚至上千的乘累加运算来确定结果。

试想一下，如果滤波器的一些权值（滤波器中固定的值，用来乘以输入数据）非常小，那么被这些小的值乘了很多次之后，结果会更小，在所使用的数字格式下会被舍入到零。相反，如果一些权值很大，被乘过多次之后的结果就会非常大，导致溢出。可以理解，这种滤波器的设计是非常敏感的平衡行为。

幸运的是，有一个合理、高效的解决方案：确保操作数是定点数格式，小于但尽可能接近 1.0。其理由是，任何一个数字乘以小于或等于 1.0 的结果不会大于它本身。因此，我们确保这样的两个数字相乘之后，绝不会导致溢出。同样，任何数乘以一个值略小于 1.0 的数之后，其结果不会太小，因此结果不太可能快速地被舍入到零。

这种做法是可行的，因为在滤波器中只是进行乘和加，这些是线性过程：$(a \times b + c)$ 和 $(10a \times b + 10c)$ /10 的结果是一样的。

再次提醒，实际使用的定点数格式和用于执行计算的硬件是不相关的。它只是一个抽象的概念，软件工程师必须牢记。这将在本章后面的讲述中通过不同的例子来说明。 43

2.7.1 定点数的运算

加法一定可以在两个定点数格式的数字之间完成，但只有当两个操作数的格式统一时才能得到正确的结果，结果的格式和操作数的格式一样：

$$(m.n) + (m.n) = (m.n)$$

$$(m.n) - (m.n) = (m.n)$$

框 2.19 中的两个例子说明了这种定点数格式数字的运算。

框 2.19 定点数表示实例

问 1：用（2.2）格式的定点数表示法表示 1.75 和 1.25，把两个数相加并给出结果。

答：首先计算（2.2）格式的位权：小数点右边有两位，小数点左边有两位。向左的数字是整数权值，是 2 的幂，从 1 开始。往右边的数字是定点数权值，是 2 的幂的倒数，从 1/2 开始：

2^1	2^0	2^{-1}	2^{-2}

我们可以把 1.75 分解为 1 +0.5 +0.25，把 1.25 分解为 1 +0.25，写出（2.2）二进制格式为 0111 和 0101。

这两个操作数的二进制加法结果为 1100，它正确吗？

1100 在（2.2）格式时等于 2 +1 =3，当然 1.75 +1.25 =3，所以结果是正确的。

下面，我们讨论当某些东西错了时会发生什么。

问 2：用（2.2）格式定点数表示法表示 1.75，用（1.3）格式定点数表示法表示 0.625，把两个数相加并给出结果。

答：在问题 1 中 1.75 已经被表示为 0111。（1.3）格式定点数表示法的权值是 1，0.5，0.25，0.125，因此把 0.625 分解为 0.5 +0.125，得到二进制数 0101。

接下来执行加法 0111 +0101，结果为 1100。

但是我们并不知道结果用什么定点数格式。我们猜测是否是（2.2）格式或（1.3）格式，对每种都转换为十进制值看一下。

用（2.2）格式时结果是 2 +1 =3，用（1.3）格式时结果是 1 +0.5 =1.5，而真正的结果是 1.75 + 0.625 =2.375。显然，这不匹配任何一种猜测的答案。

我们应该做的就是改变其中的一个，让它们有相同的格式，然后再执行加法。

注意：你看到这两个例子中的二进制模式是相同的吗？这只是我们对各例子中不同位模式的解释。按这种方式使用不同的解释，可能会导致相同的位模式有多重意义——但用于执行计算的硬件并不需要改变。 44

2.7.2 定点数的乘除

在乘法的情况下有更多的灵活性，操作数可以有不同的定点数格式，而结果的定点数格式又是从操作数的格式推导出来的：

$$(m.n) \times (p.q) = (m+p) \times (n+q)$$

显然乘法结果的位数是两个乘数的位数之和，这一结论从 2.5 节中我们已经了解的乘法器硬件结构上就能推断出来。

除法会较为复杂。事实上，执行除法的最好方式是：先去掉两个操作数的小数点，一步一步地将小数点逐位向右移，直到它们在最大操作数的最低有效位后面，适当扩大了较小的操作数。然后就可以按照标准的二进制方式相除了。

框 2.20 中的实例阐明了定点数除法是如何完成的。

框 2.20 定点数除法实例

考虑 11.000 ÷01.00（无符号数）。

它和 3 ÷1 一样没有太大的实际意义。完成此运算的第一步是将小数点向右移一个位置：

110.00 ÷010.0

这样不够，因为数字中仍然包含小数点，所以重复一次：

1100.0 ÷0100.

这还不够，所以再来一次，作为 1100.0 移除小数点的副作用，0100 同时被扩大，如下：

11000. ÷01000.

然后按照标准二进制除法去除：

```
01000 | 11000
```

继续二进制长除法：

```
      00011
01000 | 11000
    - 1000
      01000
    - 01000
      00000
```

45 结果是11，就像3÷1的值为3一样，结果正确。

观察上述实例，显然实际的除法并不比标准二进制算法复杂，但是考虑小数点的位置可能会有问题。事实上，它需要程序员小心编码。

2.8 浮点数

浮点数和二进制定点数格式很相似，但它们更灵活，因为小数点位置是可变的（作为数字本身的一部分存储）。正是这种灵活性使得浮点数只用较少的位数编码就能够表示相当大范围的值。

2.8.1 广义浮点数

一个浮点数包含尾数 S（或称定点数部分）和指数 E（或称幂），也可能有一个符号位（σ），这样，基数为 B 的数可以表示为：

$$n = \sigma \times S \times B^E$$

或者如果更正确地考虑符号，在二进制中，1代表负数，0代表正数，因此：

$$n = (-1)^{\sigma} \times S \times B^E$$

一个基数为10的例子 2.3×10^6，我们知道它就是2 300 000的一种科学记数法表示。其实这正说明了浮点数的一个主要优点：浮点数通常写出来会更短（在二进制中就是更少的位数），相比较于这个数实际的十进制（或者二进制）值。

在二进制中就是 $B = 2$ 而不是10，典型的例子如 01001111×2^6 这样的数，如果尾数（01001111）是无符号数，则：

$$01001111 \times 2^6 = 79_{10} \times 64_{10} = 5056_{10} = 1001111000000$$

其中下标10表示这是一个十进制值。

当然，最终的值和把尾数进行指数次移位后的值是一样的（由于同样的原理，我们对上面基数为10的例子中加了5个0）。

通常情况下，构成一个浮点数（σ，S 和 E）的所有位都存储在相同的位置，具有方便的位宽如16、32、64或128位。因此在处理的时候，需要有位级的操作将它们从一个整体存储分离为3个不同部分。

2.8.2 IEEE754 浮点标准

虽然有多种可能的浮点格式，而且在计算史上也出现过各种实现例子，但产生于20世纪70年代的IEEE754已经成为到目前为止最流行的标准，被所有主要的CPU制造商采用。行业内普
46 遍认为，IEEE754是一个深思熟虑的、高效的浮点格式，从而得到高度重视。

这里不打算描述 IEEE754 整个标准，但我们将介绍一些较为常见的功能。我们会讨论单精度和双精度格式，分别存储于 32 位和 64 位存储空间。在 C 语言程序设计中，这些通常对应为 float 和 double 数据类型：

名称	位宽	符号位 σ 宽	指数 E 位宽	尾数 S 位宽
单精度	32	1	8	23
双精度	64	1	11	52

此外，在中间计算阶段会在表示中加入其他位（在硬件浮点单元中），以确保整体精度保持不变，这将在后面第 2.9.3 节介绍。

除了所有使用 32 位或 64 位的有符号尾数和指数表示外，IEEE754 格式还巧妙地表示了另外 4 种模式，采用的是常规数字中不会出现的特殊位模式。如下表所示，第一行是默认模式，也称为“规格化”：

名称	σ	E	S
规格化数	1 或 0	不是全零也不是全 1	任意数
零	1 或 0	全零	全零
无穷大	1 或 0	全 1	全零
不是一个数（NaN）	1 或 0	全 1	非零
非规格化数	1 或 0	全零	非零

当写一个符合 IEEE754 标准的数时，我们通常从左向右按顺序写每一位（σ，E，S），如下：

σ	E	S

其中，整个框表示一个 32 位或 64 位的二进制数，包含一个 IEEE754 标准的值。本书中给出的所有例子都只使用 32 位单精度以节省纸张。

2.8.3 IEEE754 标准模式

在随后的讨论中，我们将用 S 来表示无符号定点数（0.23）或（0.52）格式的尾数位模式，用 E 来表示无符号二进制补码格式的指数位模式，用 σ 表示符号。请注意，他们在 IEEE754 标准的 5 个模式中含义不同。

这样一个 IEEE754 标准的数字书写格式为：

0	10110010	11100000000000000000000000

其中 $\sigma=0$，因此是正号，

$$E = 128 + 32 + 16 + 2 = 178$$

$$S = 0.5 + 0.25 + 0.125 = 0.875$$

47

我们将始终保持这个 E 和 S 的命名约定。因此，在后面的文字中当提到“尾数”和“指数”时是用来表示所写位模式的含义，而 S 和 E 是表示所写的二进制值。

例如，S 的值为 10110010b = 0.875d，可能意味着尾数是 0.875 或是 1.875 或是其他不相关（不是一个数，NaN）的值。后面将会看到，实际所写的位模式的含义会随着 IEEE754 的 5 个模式的变化而改变。

2.8.3.1 规格化模式

大部分非零数字都采用这种数字格式。依据这种模式的数字格式可以真正地被称为“浮点数”。在此模式下，数字由下面的位模式（σ，E，S）表示：

$$n = (-1)^{\sigma} \times (1 + S) \times 2^{E-127}$$

这里，首先我们看到指数是用一个移127的移码表示法（在2.3.5节中已介绍）；其次，尾数需要加上个“1”，也就是说，尾数等于$S+1$，其中S如我们所知，是以（0.23）格式书写的。

这可能会非常混乱，所以我们回到IEEE754标准数的例子，见框2.21，然后在框2.22中给出第二个例子。

框2.21 IEEE754规格化模式实例1

下面给出了一个用IEEE754标准表示的二进制值，请判断其十进制值。

0	10110010	11100000000000000000000000

首先，注意到$\sigma=0$，因此其值是正号。再看这个数是按照规格化模式写的，因此：

$$E=128+32+16+2=178$$

$$S=0.5+0.25+0.125=0.875$$

使用规格化模式数的公式，我们可以计算其值，过程如下：

$$\begin{aligned} n &= (-1)^0 \times (1+0.875) \times 2^{178-127} \\ &= 1.875 \times 2^{51} \\ &= 4.222 \times 10^{15} \end{aligned}$$

如我们所看到的，例子的结果是一个很大的数，说明浮点格式能够表示一些相当大的值。

框2.22 IEEE754规格化模式实例2

下面给出了一个用IEEE754标准表示的二进制值，请判断其十进制值。

1	00001100	01010000000000000000000000

此例中$\sigma=1$，因此是负数，从剩余的位模式得到：

$$E=8+4=12$$

$$S=1/4+1/16=0.3125$$

使用规格化模式数的公式，我们可以计算其值，过程如下：

$$\begin{aligned} n &= (-1)^1 \times (1+0.3125) \times 2^{12-127} \\ &= -1.3125 \times 2^{-115} \\ &= -3.1597 \times 10^{-35} \end{aligned}$$

这次结果是一个非常小的数，这说明浮点数可以表示一个足够大的数的范围，而且通过所表示的数的范围（2.8.4节将进一步阐述），保证了精度。

我们例子中很多数都是尾部有一长串零。我们通过考察将其中一个尾部的最低有效位从“0”变为“1”会导致其结果有什么不同，来获得关于IEEE754基本精度的一个想法。框2.23
48 提供了一个指导，告诉我们如何去测试这个效果。

框2.23 读者练习

请注意，在上述两个实例中，23位长的尾数都是以几个“1”开始，却以一长串“0”在后面结束。这样做是为了减少计算尾数值的难度，因为在（0.23）定点数格式中，左手边的权值比较容易处理，都是值如0.5、0.25、0.125等。而实际上，当我们向右移动时，位的权值迅速变得很难写下来。

本例的练习就是重复框2.21和框2.22中的一个实例，但是将尾数的最低有效位设为1。如果尾数最高有效位（即第23位）的权值是$2^{-1}(0.5)$，下一位（即第22位）的权值是$2^{-2}(0.25)$，那么第0位的权值是多少？

当把这个加到结果中时，和我们书面写下来的结果有什么不同呢？

现在真正的问题是，这是否意味着IEEE754中浮点数的精度呢？

2.8.3.2 非规格化模式

49 有些数的绝对值非常小，以至于IEEE754无法表示它们。广义的浮点数会把这样的值舍入为零，但IEEE754有一个特殊的非规格化模式，它允许所表示数的绝对值逐渐变小直到零——

优雅地降低精度直到达到零。

非规格化模式实际上并不是浮点数，因为指数（数的一部分，用来指明小数点位置）全部设置为零，所以不再“浮”。然而，这种模式下，允许扩展范围是 IEEE754 标准数的重要优势。

该模式下，位模式（σ，E，S）表示的数字如下：

$$n = (-1)^{\sigma} \times S \times 2^{-126}$$

首先，可以看到指数是固定值，上面已经提到过。其次，我们不再需要给尾数加“1”。当我们在 2.8.4 节研究数的范围时，会发现这样做的原因很明显。

由于指数固定，位模式中指数总是全零，而尾数不是零。框 2.24 中的实例将有助于澄清任何混淆。

由于非规格化数向下扩展了 IEEE754 的范围，因此，其绝对值总是很小。

框 2.24 IEEE754 非规格化模式实例

下面给出了一个用 IEEE754 标准表示的二进制值，请判断其十进制值。

0	00000000	11010000000000000000000

首先，我们注意到 $\sigma=0$，此位模式表示的数是正数。

$E=0$，因此我们查 2.8.2 节中的模式表，看我们处理的是一个零还是一个非规格化数。我们实际上需要测试尾数来决定（零必须尾数也为零，否则是一个非规格化数）。

查看尾数，我们看到是非零值，因此按照非规格化模式数如下：

$$S = 0.5 + 0.25 + 0.0625 = 0.8125$$

使用非规格化模式数的公式，我们可以计算其值，过程如下：

$$n = (-1)^{0} \times 0.8125 \times 2^{-126}$$
$$= 9.5509 \times 10^{-39}$$

2.8.3.3 其他模式数

零、无穷大和 NaN 由其特殊的位模式确定。这些都可以是正数或负数，在硬件上需要特殊的处理（见框 2.25）。 50

框 2.25 IEEE754 无穷大以及其他“数”

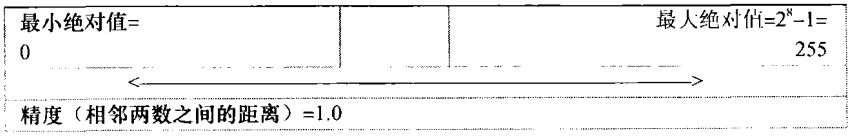
无穷大是最常产生的数，通过除以零或一个规格化模式的溢出产生。无穷大可以是正也可以是负，用来指明溢出发生在哪一边。

NaN，表示不是一个数，由一个未定义的数学运算产生，如无穷大乘以零，或者零除以零。

零本身可能表示一个运算确实结果为零，例如（2 -2），或者由一个下溢导致，当结果太小甚至是非规格化模式也无法表示时，这种情况下，正负零的意义是说明这个不能表示的数是略高于零还是略低于零。

2.8.4 IEEE754 数的范围

理解 IEEE754 的一个很好的方式就是构建一个数轴，它表示了这种格式下数的可能范围。下面这个数轴描述了无符号 8 位数字的表示范围，其具体包括：



有三个参数来描述此格式。第一个是绝对值最小的数（0000 0000），第二个是绝对值最大的数（1111 1111），最后是精度。精度定义为在此格式下相邻两个数之间的距离。在此处，数是按照整数向上数的：1，2，3，4，5，…，255，所以步长就是 1。

现在，我们用同样的方式为 IEEE754 格式定义一个数轴。为了简化问题，我们只考虑正数

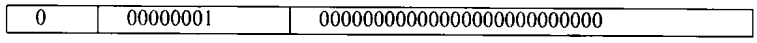
并且只考虑单精度的情况，但我们对规格化和非规格化两种模式都会顾及。

规格化模式要求 E 不能全零或全1，但 S 可以是任何值，其表示的实际值如下：

$$n = (-1)^{\sigma} \times (1 + S) \times 2^{E-127}$$

如果我们寻找绝对值最小的规格化模式数，需要找到最小 S 和最小 E 的可能值。显然最小的 S 是0，但最小的 E 不能是0（因为0意味着非规格化模式或是零），因此只能是00000001：

51



把这两个值放到公式中，假设是正数，那么：

$$\text{min norm} = (1 + 0) \times 2^{1-127} = 1 \times 2^{-126} = 1.175 \times 10^{-38}$$

下面寻找绝对值最大的数，我们知道 S 可以是任何值，但 E 不能是11111111（因为那意味着无穷大或NaN模式），因此最大的 E 为11111110，最大的 S 是全1。

先看 E 的值等于254。然而 S 的值稍微难算一些：

111 1111 1111 1111 1111 1111

但是记住这是（0.23）格式，且该值略小于1.0，我们看到如果在最低位加一个二进制1，那么这个字中所有的二进制1都通过进位变成0，因为进位传递经过了整个加法链，得到值如下：

```
  111 1111 1111 1111 1111 1111
+ 000 0000 0000 0000 0000 0001
-----------------------------
= 1000 0000 0000 0000 0000 0000
```

基于这个事实，且此处有23位，最高位的位权是 2^{-1}，第二高位的位权是 2^{-2}，依次类推，第23位最高位（实际也是最低位）的位权一定是 2^{-23}。

所以，S 的值一定是 $(1.0 - 2^{-23})$，因为把 2^{-23} 加到 S 上将使其等于1.0：



把它们放到公式中，得到：

$$\text{max norm} = (1 + 1 - 2^{-23}) \times 2^{254-127} = (2 - 2^{-23}) \times 2^{127} = 3.403 \times 10^{38}$$

数字的精度如何呢？如果我们看一下得到的这个数字，就会发现精度并不是常数。最小位始终是指数所表示范围的 2^{-23} 倍。

最终，规格化数的表示范围如下：



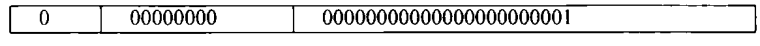
由于符号位只改变符号，并不影响绝对值，因此负数的表示范围一定是一样的。

非规格化模式可以用类似的方法处理，尽管其指数定义为全零，其表示的值如下：

52

$$n = (-1)^{\sigma} \times S \times 2^{-126}$$

注意尾数不允许是零，所以最小的非规格化数就是把尾数的最低位置1：



因此根据介绍规格化模式时对最大值的讨论可知 2^{-23} 的值，公式变成：

$$\text{min denorm} = 2^{-23} \times 2^{-126} = 2^{-149} = 1.401 \times 10^{-45}$$

至于非规格化数的最大值，很简单就是 S 为最大值时所表示的数。查看2.8.2节中的模式表可知其为全1：



再次使用规格化最大值情况下的结论，最大值有 $(1-2^{-23})$ 的意义，其值为：

$$\max \text{ denorm} = (1-2^{-23}) \times 2^{-126} = 2^{-149} = 1.175 \times 10^{-38}$$

现在推断一下数的精度：本例中，由于指数是固定的，精度仅由尾数最低位值乘以指数决定：

$$2^{-23} \times 2^{-126}$$

最小值1.401×10^{-45}		最大值1.175×10^{-38}
<——————————————————>		
精度（相邻两数之间的距离）$=2^{-23} \times 2^{-126}$		

把之前所有的精度范围放在一起，可以看到 IEEE754 单精度数的巨大表示范围。请注意这只是实数轴中正数的那一半，负数那边同样。

零	非规格化数		规格化数	
0	1.401×10^{-45}	1.175×10^{-38}	1.175×10^{-38}	3.403×10^{38}
0	<——————>		<——————>	
0	精度$2^{-23} \times 2^{-126}$		精度$2^{-23} \times 2^{E-127}$	

当我们想把一个十进制数转换成 IEEE754 的浮点数时，精度表很有帮助，它告诉我们该使用哪一种模式，是零、非规格化数、规格化数还是无穷大。为了阐明这个问题，下面在框 2.26 中给出一个十进制转换为浮点数的实例。

在 2.9.1 节和 2.9.2 节还会有更多这种转换的例子。 53

框 2.26 实例：十进制数到浮点数的转换

问：写出在 IEEE754 单精度格式下的十进制数 11 的值

答：在 2.8.4 节中查找我们的精度表，可以看到此值正好位于规格化数的范围里，因此查找规格化数形式：

$$n = (-1)^{\sigma} \times (1+S) \times 2^{E-127}$$

知道了这个，首先需要将十进制数 $N=11$ 写成 $A \times 2^{B}$ 的格式，其中 A 的值等于 $(1+S)$。已知 $1 \geqslant A \leqslant 2$，$0 \geqslant S \leqslant 1$，可能最简单的方法就是反复对 N 除以 2，直到得到一个介于 1 和 2 之间的 A 值：得到一个序列，11 除以 2 后是 5.5，之后是 2.75，最后是 1.375。

故 $A=1.375$，因此 $N=1.375 \times 2^{B}$，无需太多的运算，B 由原始数除以 2 的次数得到，在此例中是 3。因此数为：$n=(-1)^{0} \times (1.375) \times 2^{3}$。

考察规格化数公式，需要：

$$\sigma = 0$$

$$E = 130 (\text{因为 } E - 127 = 3)$$

$$S = 0.375 (\text{因为 } 1 + S = 1.375)$$

寻求 E 的二进制位模式，为 128 +2 即 10000010，由于 0.375 很容易表示为 0.25 +0.125，故整个数为：

0	10000010	01100000000000000000000

2.9 浮点数处理

到目前为止我们只考察了浮点数的表示，特别是 IEEE754 标准的表示。只有能够通过对这种数据的处理来完成任务，这种表示方法才是有用的，因此这里对此进一步进行讨论。

在许多计算机系统中，浮点数处理是通过使用专门硬件来完成的，称为浮点协处理器或浮点运算单元（FPU）。事实上，尽管这个部件往往存在于商业 CPU 的芯片内部，但其通常还是作为协处理器访问的，而不是主处理器的一部分。

对于那些没有浮点运算硬件支持的计算机来说，广泛使用软件仿真，除了执行时间较长

（参见4.6.1节）外，用户可能并不知道浮点计算是在哪里进行的，不管是硬件还是软件。大多数浮点支持（硬件或软件）都是基于IEEE754标准上的，虽然偶尔有软件选项，允许以牺牲
54 IEEE754标准的准确性为代价来提高计算速度。

IEEE754数的处理过程包括以下步骤：

1. 接收操作数；
2. 检查数字格式模式，如果是固定值，则立即从查找表中产生结果；
3. 如果需要，变换指数和尾数；
4. 执行运算操作；
5. 结果被转换回合法的IEEE754数字格式。保持尾数最左侧的最高位为1，因为这样可以获得最大精度。

2.9.1 IEEE754数的加减运算

在广义浮点数中，执行加减运算之前指数的值必须是相同的。这类似于在定点数格式加法$(n.m)+(r.s)$中，进行加运算之前应确保$n=r$且$m=s$，正如我们在2.7.1节看到的那样。

例如，考察十进制数$0.824\times10^2+0.992\times10^4$。为了使运算简单，必须让两个指数相同，然后就可以简单地进行尾数相加了。但是我们是把它们都转换为10^2、10^4，还是中间的某个值如10^3？

为回答这个问题，让我们首先看一下如何将指数的阶向下转换。我们知道10^3的值等于10×10^2，10^4的值等于100×10^2，由于我们在谈论十进制，所以每次降低指数的阶时都是给尾数乘以基数值10。在我们的计算中执行后得到和如下：

$$0.824\times10^2+99.2\times10^2$$

相反向上转换为：10^2的值等于0.01×10^4，因此和为：

$$0.008\,24\times10^4+0.992\times10^4$$

在纸上，以十进制表示，这两个指数的值是相同的，但在二进制中，在硬件中，并不是如此。因此问题仍然是：我们采取哪个操作？是转换较小的指数以匹配较大的，还是把较大的指数转换为较小的，还是转换为中间的某个？

答案是，首先，我们并不愿意把两个数都转换，因为那样会带来额外的工作；其次，考虑二进制位，我们知道把一个指数变小就需要相应地把尾数变大，显然尾数变得太大时会有溢出的风险。因此我们选择从不增大尾数，这意味着我们不得不增加较小的那个指数，并把其相应的尾数变小：

$$0.008\,24\times10^4+0.992\times10^4$$

这就是所谓的指数等值或归一化操作。稍后，我们将看到，有一些方法能够帮助防止尾数在
55 此变换过程中向下舍入为零。

一旦指数相等，我们就可以执行尾数相加：

$$0.008\,24\times10^4+0.992\times10^4=(0.008\,24+0.992)\times10^4$$

IEEE754的加减运算类似于十进制情况，只是因为基数是2，将其中一个指数增加到和另一个指数相同的操作会导致这个数的尾数以2为因子递减。在二进制中除以2可以通过右移一位完成。

还有一个因素我们必须考虑到，就是结果的格式。注意到规格化模式中尾数不能大于1，因此，如果尾数的计算结果太大，我们必须右移尾数并相应地增加指数。类似地，如果尾数变小，则必须左移尾数并相应地减小其指数。这些因素我们将通过框2.27中的实例阐明。

现在我们进一步看这个过程。知道了如何在执行运算之前把指数归一化，可以结合我们掌

握的 IEEE754 格式知识，直接在 IEEE754 格式上执行这些运算。

参考框 2.27 中的实例，我们现在可以写下数的 IEEE754 位模式，并在框 2.28 中进行转换。

减法同加法相似，所有的步骤都相同，除了尾数是对应相减外。当然，我们还要考虑结果尾数的溢出问题，因为可能会进行两个负数相减，那么结果会大于任何一个原来的操作数。

框 2.27 浮点运算实例

问：将十进制值 20 和 120 转换为 IEEE754 格式，将它们相加并将结果转换为十进制。

答：在 2.8.4 节中查找我们的精度表，发现两个值都位于 IEEE754 的规格化数范围内，但最初我们只考虑通用的 $A \times 2^B$ 格式，这里并不关注确切的 IEEE754 位模式。只是简单记住 $A=(1+S)$ 和 $B=(E-127)$。

先从 20 开始，反复除以 2 直到结果在 1 和 2 之间：10，5，2.5，1.25，因此 $A=1.25$。共除了 4 次，故 $B=4$。

120 同样，除以 2 后依次为：60，30，15，7.5，3.75，1.875，故 $A=1.875$。除了 6 次，所以 $B=6$。

将结果放到下表中，在这一阶段我们并不需要得出 E 和 S 的位模式，我们更关注它们的表示：

σ	B	A	二进制值	十进制值
0	4	1.25	1.25×2^4	20
0	6	1.875	1.875×2^6	120

下一步是指数归一化。如正文中描述的，让它们都等于较大的指数值，把较小的数对应的尾数适当减小。

因此，1.25×2^4 变成 0.625×2^5，再变成 0.3125×2^6，重新形成下表中的操作数：

σ	B	A	二进制值	十进制值
0	6	0.3125	0.3125×2^6	20
0	6	1.875	1.875×2^6	120

由于两个指数相等，现在可以进行尾数相加，进而得到结果如下：

σ	B	A	二进制值	十进制值
0	6	2.1875	2.1875×2^6	?

但是，这不是 IEEE754 的合法表示，因为尾数值太大了。还记得公式中的 $(1+S)$ 吗？是的，$A=(1+S) \leqslant 2$ 是我们的约束。如果两个操作数都是 IEEE754 兼容的，那么我们应该能够保证不需要移位超过一位以上就能正确表示这个数，因此我们将 A 的值右移一个二进制位，并增加 B 的值：

σ	B	A	二进制值	十进制值
0	7	1.09375	1.09375×2^7	?

用计算器检查可知，1.09375×2^7 的确是正确答案，其十进制为 140。

框 2.28 IEEE754 运算实例

首先，看规格化模式公式：

$$n = (-1)^{\sigma} \times (1+S) \times 2^{E-127}$$

以十进制的 20 为例，在前面的实例中，它被表示为 1.25×2^4。带入公式中得到 $(1+S)=1.25$，则 $S=0.25$；$(E-127)=4$，则 $E=131$。表示如下：

0	10000011	01000000000000000000000

十进制的 120 是 1.875×2^6，因此 $S=0.875$ 且 $E=133$。

0	10000101	11100000000000000000000

加法的结果是 1.09375×2^7，因此 $S=0.09375$ 且 $E=134$。

由于0.09375不是一个明显的2的幂的定点数，我们可以使用普通的方法来确定位模式。在此，我们不断地乘以值2，当结果等于或大于1时减去1，当余下数为零时结束：

0：0.09375

1：0.0187

2：0.375

3：0.75

4：1.5－1＝0.5

5：1－1＝0

我们在第4和5次迭代时减掉了1，据此将左起第4和5位设置为1。事实上，我们也可以用这种方法计算最前面的两个数，但它们太简单了：

0	10000110	00011000000000000000000

2.9.2 IEEE754数的乘除法

对于乘法和除法运算，我们不需要先规格化操作数，但我们需要对这两个数进行两个计算：一个是对尾数的；一个是对指数的。下面是基为B的数的运算关系：

$$(A \times B^C) \times (D \times B^E) = (A \times D) \times B^{(C+E)}$$

$$(A \times B^C)/(D \times B^E) = (A/D) \times B^{(C-E)}$$

以一个十进制数为例说明此点：

$$(0.824 \times 10^2) \times (0.992 \times 10^4) = (0.824 \times 0.992) \times 10^{(2+4)} = 0.817\,408 \times 10^6$$

这里还是有一个因素要考虑，在IEEE754格式下，结果必须转换为正确的表示格式，且应检查是否为特殊结果值（零、无穷大、NaN）。

2.9.3 IEEE754中间格式

56
~
57

虽然在一个特定的IEEE754标准计算中其输入输出是IEEE754标准格式的操作数，但还是会出现输出结果数字不正确的情况，除非在计算中有非常高的精度。一个9位减法的小例子将说明这点：

1.0000 0000 $\times 2^1$ A

$-1.1111\ 1111$ $\times 2^0$ B

在我们进行减法之前，仍然有必要把这两个数规格化为相同的指数，我们通过增加较小的指数来完成，正如我们在2.9.1节所做的那样：

1.0000 0000 $\times 2^1$ A

$-0.1111\ 1111$ $\times 2^1$ B

现在我们可以进行计算，结果如下：

58 0.0000 0001 $\times 2^1$ C

然后把尾数尽量左移：

1.0000 0000 $\times 2^{-7}$

让我们看一下实际使用的几个数字。操作数A的值为2.0，操作数B的值为$(2.0-2^{-8})$即十进制的1.996 093 75。因此结果应该是：

$$2.0 - 1.996\,093\,75 = 0.003\,906\,25$$

然而，我们计算的结果是1×2^{-7}或0.007 812 5，一定是哪里出了问题。

现在我们重新计算，但这次在中间阶段加上一个所谓的*看守位*（guard bit）。通过在最低位这边增加一位，有效地扩展了尾数的长度。我们从规格化数字这点重新开始，注意额外的这

一位：

1.0000 0000 **0** $\times 2^{1}$ A

−1.1111 1111 **0** $\times 2^{0}$ B

接下来规格化指数，将 B 的尾数右移一位时，其最低位移到看守位：

1.0000 0000 **0** $\times 2^{1}$ A

−0.1111 1111 **1** $\times 2^{1}$ B

减法后的结果如下：

0.0000 0000 **1** $\times 2^{1}$ C

再把尾数尽量左移：

1.0000 0000 **0** $\times 2^{-8}$

注意 C 这一行，这次其为 1 的最高位是看守位，而之前是在前一位。规格化值为 1×2^{-8} 或 0.000 390 65，这次结果正确。

尽管这个例子示意了一般的 8 位浮点数，但对 IEEE754 数来说规则是相同的。

上面的例子显示了减法中由于精度丢失错误导致的结果错误。当然同样的错误也可能会发生在加法中，因为 A − B 和 A +（ − B）是一样的。但在乘法和除法中也会发生吗？这个留给读者作为练习，请试着找出一个简单例子来说明这个问题。

在 IEEE754 术语中，用到不止一个看守位，这个方法称为扩展中间格式，由下面的位宽进行标准化：

名称	位宽	符号位位宽	指数位宽	尾数位宽
扩展单精度	43	1	11	31
扩展双精度	79	1	15	63

59

显然用计算机去处理 43 位或 79 位的数字比较尴尬，因为计算机是基于 8 位一组的二进制数字尺寸的，但这通常并不是问题，因为扩展中间格式是设计用于硬件浮点运算单元的。输入操作数和输出操作数仍然是 32 位或 64 位的。

2.9.4 舍入

有时，为了把一个扩展中间值表示成想要的输出格式，需要对其进行舍入。其他情况如从双精度格式变为单精度格式时也可能需要舍入。有时舍入是定点数和浮点数计算所必需的。

有不止一种方法可以完成数字的舍入，在操作系统控制下，许多计算机系统都支持其中的一种或多种：

- 舍入到最接近的（最常用）——舍入到最接近的表示值，如果两个值都一样近，默认舍入为 LSB = 0 的那个值。例如，1.1 舍入到 1，1.9 舍入到 2，1.5 舍入到 2。
- 向上舍入——舍入到更大的一个数，例如 −1.2 舍入到 −1，2.2 舍入到 3。
- 向下舍入——舍入到更小的一个数，例如 −1.2 舍入到 −2，2.2 舍入到 2。
- 向零舍入——相当于始终截断要舍入的部分，例如 −1.2 舍入到 −1，2.2 舍入到 2。

对于非常高精度的计算，可以对每个计算执行两次，一次向上舍入，一次向下舍入。两次结果的平均值可能是答案（起码在线性系统中是）。尽管用这种方法得不到一个高精度的答案，但得到的两个结果的差异可以为计算中数值的精度给出好的建议。

2.10 小结

本章标题是“基础知识”，我们的计算机之旅从这里真正开始——无论它是一个房间大小的

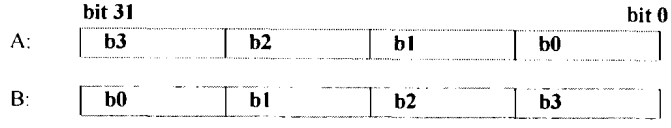
大型机、一个台式机或者一个嵌入式系统。本章也是基础，因为几乎所有的计算机，无论其大小，都是基于相类似的规则。它们使用相同的数字格式，执行同样类型的计算如加、减、乘、除。我们看到的主要区别就是存在一些更快的方法来完成这些运算，但以增加复杂性、面积、功耗为代价。

60 我们以对计算机的定义及其组成的考虑开始本章。我们介绍了 Flynn 提出的计算机类型（或 CPU）的分类，学习了连接关系及所包含的功能层次。然后我们更新了数字格式和基本运算的知识，之后掌握了一些更详细的关于如何计算的知识。

本章已经涵盖了基础知识，下一章将侧重于如何获得连接和计算，即如何在一个 CPU 中将
61 这些功能单元放在一起，编写和存储程序，控制所需的内部操作。

思考题

2.1 一个程序员撰写一个 C 语言程序，将 4 个字节（b0、b1、b2、b3）存储到连续的内存空间，在有 32 位宽内存的小尾端计算机中运行。如果他在程序运行后检查内存，他能够看到像下表中 A 或 B 这样的数吗？



2.2 完成下表（8 位二进制数），不能进行转换的请标出。

值	无符号数	补码	原码	移 127 码
123				
-15				
193				
-127				

2.3 采用补码（2.30）格式，我们如何表示值 0.783 203 125？能够用（a）32 位，（b）16 位或（c）8 位位宽精确表示吗？

2.4 一个 BCD 码数字包含 4 位。使用 4 位行波进位加法器，用额外的一位加法器和逻辑门进行改进，搭建出一个新加法器，使其能够完成两个 1 位 BCD 数字加法并产生 BCD 格式的和。扩展本设计，使之能够完成两个 4 位 BCD 数的加法。

2.5 使用部分积（长乘法）方法，手工完成两个 4 位二进制数 X = 1011 和 Y = 1101 的乘法，假设它们都是无符号数。

2.6 使用 Booth 算法重做上一道题。

2.7 如果加、移位和比较操作，每个需要一个 CPU 周期完成，那么执行 2.5 题中的计算需要多少个 CPU 周期？将其和 2.6 题中的 Booth 算法相比较。在大字宽运算时 Booth 算法更有效吗？

2.8 在 RISC CPU 中有一条指令称为“MUL”，它能将两个寄存器中内容相乘并把结果存储到第三个寄存
62 器。寄存器是 32 位宽，存储的结果是 64 位逻辑结果的前 32 位（请注意 32 位 × 32 位应该得到 64 位）。但是程序员想得到全部 64 位结果，他怎么才能得到呢？（提示：你需要做不止一次的乘法，以及一些与和加运算来得到结果。）验证你的方法，确定需要多少指令。

2.9 如果我们进行两个（2.6）格式无符号数 X = 11010000 和 Y = 01110000 的乘法，那么我们将得到一个（4.12）格式的结果。将结果左移 2 位，变成（2.14）格式（即有效地去除前两位），然后再截取到（2.6）格式（丢弃低 8 位）。这样会导致溢出吗？截断会丢失某些位吗？

2.10 按照 IEEE754 单精度浮点数标准：

（a）给出在下面的情况中，以（σ，E，S）格式存储的数 N 的值：

ⅰ. $E=255$，$S\neq0$

ⅱ. $E=255$，$S=0$

ⅲ. $0<E<255$

ⅳ. $E=0$，$S\neq0$

ⅴ. $E=0$，$S=0$

（b）以 IEEE754 单精度规格化格式表示下列值：

ⅰ. -1.25

ⅱ. 1/32

2.11 采用标准指数/尾数的浮点数格式能够以多于一种的方式表示零吗？IEEE754 能够以多于一种的方式表示零吗？如果是，给出并解释不同的表示方式。

2.12 根据图 2-9 所示的除法流程图，给出下面运算的商和余数值：无符号 5 位二进制除法 Q/M，其中Q = 10101b 且 M = 00011b。

2.13 根据图 2-7 给出的乘法流程图，部分积乘法执行两个无符号 5 位二进制数 00110 和 00101 的相乘。请给出用到的寄存器数量、大小以及每一次迭代时寄存器的值。

2.14 根据图 2-8 给出的乘法框图重做上面的问题，比较并对比这两种方法在效率、步骤数、寄存器的数量等方面的情况。 63

2.15 考察在 C 编程语言中下面的运算：

$$0.25 + (\text{float})(9 \times 43)$$

假设整数采用 16 位二进制数表示，浮点数采用 32 位 IEEE754 单精度格式表示，请按照执行计算所需要的步骤产生符合 IEEE754 格式的结果。

2.16 下面描述的处理器属于 Michael Flynn 分类的哪一种：有一条指令可以同时将一组 5 个内部寄存器的每个字节右移一位？

2.17 判断自修改代码（是一种软件，通过重写它的部分代码可以修改它自己的指令）更适合冯·诺依曼体系结构系统还是哈佛体系结构系统。

2.18 使用一个 16 位处理器且只有一个结果寄存器，按照过程将（2.14）格式无符号数 X = 01.11000000000000 和（1.15）格式无符号数 Y = 0.110000000000000 相加。为避免结果溢出应该采用什么格式？本例中的计算有任何精度损失吗？

2.19 识别下列数字是 IEEE754 模式的哪一种：

1	10100010	10100000000000000000000000
0	00000000	10100000000000000000000000
0	11111111	00000000000000000000000000

2.20 按照基数为 7 进行下列计算和表示，结果的尾数和指数会怎样？

$$(3 \times 7^8)/(6 \times 7^4)$$

提示：你不需要使用计算器来得到结果。

2.21 使用部分积（长乘法）方法，手工计算两个 6 位二进制数 X = 100100 和 Y = 101010 的相乘，假设二者都为有符号数。

2.22 交换前面题目中的乘数和被乘数，重新完成乘法（即两个 6 位有符号二进制数 X = 101010 和 Y = 100100）。比较执行部分积累加所需的加法数量。有没有可能通过交换乘数和被乘数来简化这个过程？如果有，为什么？ 64

2.23 使用 Booth 算法重做上面两个乘法，当交换乘数和被乘数后，部分积累加的数量有所不同吗？

2.24 参考 2.9 节，执行一个单精度 IEEE754 浮点规格化数的乘法时，确定所需的整数加法、移位和乘法操作的数量，将其和运行一个（2.30）×（2.30）格式乘法时所需的基本操作数量进行比较。本题忽略扩展中间模式、溢出和饱和效应，并假设两个浮点数是不同的指数值。

2.25 使用减法迭代来执行一个 8 位除法，需要多少计算操作？ 65

第3章

Computer Architecture: An Embedded Approach

CPU 基础

在这一章里，我们开始着眼于一个组合的整体，我们称之为计算机——尤其关注它的大脑，即中央处理单元（CPU）。本章将侧重介绍计算机体系结构的传统观点，既不会考虑现在最先进的扩展和加速，这些将在第5章中介绍，也不会对计算机中的单个功能单元进行深入的讲解，这会在第4章中具体讲述。

相反，本章将关注计算机是由什么构成的、各部分是怎么组织和被控制的以及它是如何编程的。

3.1 什么是计算机

当人们说起一台计算机，他们的印象是一个米色的箱子，有显示器、键盘和鼠标。虽然他们印象中的这个箱子确实包含着一台计算机，但我们知道其实箱子里需要了解的东西还很多。

系统中的“计算机”部分包括CPU、存储子系统和连接它们的总线——实际上正是这几块构成了以存储程序来执行功能的数字计算机。它并不需要显卡、无线接口卡、硬盘和音响系统来计算或执行所存储的程序。

存储程序的数字计算机可以说是一个非常灵活但又相当基本的数据计算和传输的机器，通过编程来实现需要的功能。

如今，科技发达地区的人们常被十几台或上百台计算机包围。它们可能在微波炉中、烤炉中、手机中、MP3播放器中，甚至是电子门锁中。据估计，一辆豪华汽车含有超过100个处理器，甚至一个入门机型都可能包括不止40个单独的设备。最新一个令人惊讶的例子，一条双人电热毯已经发展到包含4个专用微处理器——在电热毯的两边各有一套独立控制，每套中一个工作、一个备用。随着这种规模的应用，容易想象得到“未来就是嵌入式的”。本章的内容适用于各种计算机，无论是房间大小的还是蚂蚁大小的。

3.2 让计算机为你服务

正如我们所看到的，在最基本的层面，计算机只是一个能够传输数据和执行逻辑操作的单元。所有更高级别的计算功能都是由这些基本数据传输和逻辑运算构成的一个序列或组合。计算机内各种各样的单元模块用于执行不同的任务，而这些是非常标准的组成模块，被大多数计算机所采用。例如，一个算术逻辑单元（ALU）执行算术运算，而一个总线用来从一点到另一点传输数据。显然，需要一些方法来指导计算机执行——决定在何时何处使用这些基本模块传输数据，以及执行哪一个逻辑操作。计算机（包括其内部单元和总线）必须通过编程才能执行人们希望它完成的工作。

作为第一步，需要将要完成的工作拆分为一系列可行的操作。这样的一个序列称为程序，并且每个操作是通过指令加操作数来发布命令。计算机中能够支持的操作列表称为指令集。

3.2.1 程序存储

程序中的所有指令需要以一种可被计算机访问的方式储存。最初的电子计算机是通过将导线插入不同的插孔来编程的。后来，使用手动转换，随后又使用自动的打孔式读卡器，穿孔之后

磁带被发明出来，但是不管是哪种存储格式，每一次上电之后的新编程都需要手工输入。

现代计算机将程序存储在磁盘、ROM、EEPROM、闪存或类似的媒体上。程序总是在执行前从存储设备读取到RAM中，这是出于性能的原因：RAM比大多数的大容量存储设备速度更快。

在内存中存储的条目需要有一个能够访问的位置。这个存储的位置也需要被标识，这样才能进行访问。早期的计算机设计者称存储的位置为地址，因为这让CPU可以选择和访问任何存放在独立地址中的指定的信息或程序代码。实现这件事最有效的方式就是由CPU通知存储设备它要访问的地址，然后等待该地址的内容，在一段时间后从设备接口读取这个内容值。

大家知道，CPU的编程用最低级的机器代码指令完成，这些指令或者是定长的（在大多数RISC机器中，如ARM、PIC或MIPS），或者是指令字长度可变的（在一些CISC机器中，如Motorola 68000）。程序是按特定序列排列的指令簇，用来指示计算机执行所要求的任务。

执行这些指令序列完成正确的工作，可能需要访问一些需要处理的数据。历史上曾经主张将程序存储空间和数据存储空间分离，特别是因为这两种类型的信息有不同的特点：程序通常 67
是顺序且只读的，而数据可能需要读/写双向访问，而且对数据的访问可能是按顺序的也可能是随机方式。

3.2.2 存储架构

一个计算机中的所有存储位置都可以被定义成“存储器”，因为一旦写入它们就记得写入的值。然而，通常情况下，我们提到这个术语时指的是固态的RAM和ROM，而不是寄存器、光盘等。无论是哪一种命名约定，对存储器的定义都由不同的权衡和技术选择来决定，包括以下几个特点：

- 成本。
- 密度（每立方厘米的字节数）。
- 电源效率（每次写或读消耗多少纳焦耳，或存储时间的秒数）。
- 访问速度（包括寻道时间和平均访问时间）。
- 访问大小（字节、字、页等）。
- 易失性（即数据在设备断电后丢失）。
- 可靠性（它有移动部分吗？有年限吗？）。
- CPU管理代价。

这些因素导致存储系统有一个层次结构，如图3-1所示的金字塔，无论一个大的台式机/服务器或一个典型的嵌入式系统都是如此架构。所示的两个主题——内存管理单元（MMU）和高速缓存将在随后的第4章讨论。而对于目前本节的讨论，请注意寄存器——距CPU功能单元非常近的临时存储位置——是最快的，但也是最昂贵的资源（因此一般数量很少，范围从简单微 68
控制器的1、2或3个到128个，在一些大型的UNIX服务器中会更多）。

金字塔从顶向下，每字节的成本逐渐降低（从而提供的数量逐步增加），但代价是访问速度也随之降低。无论是嵌入式系统、台式机或超级计算机，几乎都是包含以下几个层次的一个架构：

- **寄存器**：存储程序的临时变量、计数器、状态信息、返回地址、堆栈指针等。
- **RAM**：存放堆栈、变量、待处理的数据，往往是程序代码的临时存储位置。
- **非易失性存储器**：如闪存、EPROM或硬盘——存储要执行的程序——在初始上电启动之后显得特别重要，那时易失性RAM的存储空间都是空的。

其他的存储层次是为了方便或有速度方面的原因，正因为存储架构中有这么多层次，有几个地方都能够存储所需的信息条目，因此，需要一个方便的手段能够在需要的时候在各存储位置之间传输信息。

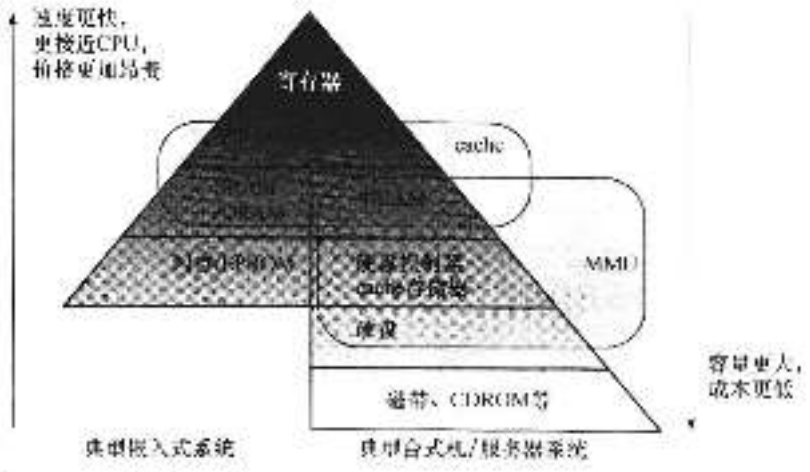


图3-1 用金字塔形的框图在速度、尺寸和开销等方面说明嵌入式系统（左边）和传统台式机（右边）的存储架构

3.2.3 程序传输

为了把一个程序从外部存储器读取到RAM，需要用到I/O接口，如IDE（Integrated Drive Electronics——一个非常流行的硬盘接口）、SCSI（Small Computer Systems Interface——可以连接光驱，扫描仪和其他许多设备）、其他的并行总线或串行总线（如USB）。这些接口将在后面的6.3.2节和6.3.4节介绍。

RAM和CPU之间的连接是通过一个总线实现的，这也是CPU和I/O设备之间的连接方式。在传送一个程序时，可同时传一个字节或一个字。RAM可能在CPU所在的集成电路（IC）的内部，也可能在其外部。

当程序的一条指令从RAM读入CPU后，它需要解码然后被执行。由于CPU内部的不同单元执行不同的任务，因此待处理的数据需要被定向到一个能够执行相应功能的单元，如果希望在CPU内部模块间传输信息，则需要在取指/译码单元和其他不同的处理单元间有一个内部总线，也许是一个能够从每一个处理单元收集结果并把其送到某处的总线。

通常情况下，待处理的数据已经在内部寄存器中（尤其是在许多现代的CPU中，称为load-store，其体系结构的约束要求待处理的数据必须来自寄存器）。这个数据通过总线从寄存器传输到处理单元，结果将再由总线送回给寄存器。往往将内部的所有寄存器放在一起构成寄存器组以方便使用。此外，在常规体系结构的机器中，每一个处理单元都会通过总线连接到这个寄存器组。

在第4章中，我们将以不同的方式来研究计算机总线，就像研究现代CPU中许多功能模块一样，并考虑不同的总线安排对性能的影响。这里，我们假设这样的内部总线确实存在，已满足我们的要求。

69

给定一个（可能相当复杂）CPU内部总线互连网络，以及连接在这个网络上的多个内部功能单元和寄存器，问题就出来了：如何仲裁和控制总线上和总线间的数据传输？

3.2.4 控制单元

多个总线、寄存器、各种功能单元、内存、I/O端口等都需要进行控制，这就是控制单元的工作职责。大多数操作都需要一个定义在CPU内部的处理流程。如：

- 取指。

- 指令译码。
- 执行指令。
- 指令执行结果的写回（如果需要）。

此外，还需要一个方法确保这些步骤发生并且按照正确的顺序执行。这就预先假定了需要在设备内部，从某些控制单元到需要被控制的片上单元之间，有一组控制线和信号。

在早期的处理器中，控制单元是一个简单的有限状态机（FSM），在预先定义的几个状态之间无休止地转换。控制线从该控制单元以各种线和连接构成的蜘蛛网连接到需要控制的每一个端点。当我们在第8章设计自己的处理器时将会详细地看到此方法。

控制不仅在读取和分发指令时需要，单个指令的执行也同样需要。考虑执行一个数据传输的简单情况，通过一个32位单总线从寄存器A传到寄存器B（LDR B, A），如图3-2所示。

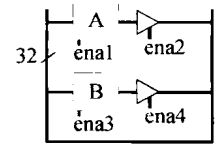


图3-2 一个非常简单的计算机控制单元框图，显示了两个寄存器，每一个都配有三态缓冲器，一个32位总线连接到所有端口

图中的两个三角形是三态缓冲器——类似于一个开关设备，当控制信号有效时数据信号能够通过缓冲器，但当控制信号无效时，数据信号不能通过。例如，三态缓冲器可用于总线上以决定哪个寄存器被允许驱动总线。任何时刻都只有一个寄存器能够驱动总线，因此其他所有到总线的三态连接必须都是关闭的。

铭记这一点，数据传输所需要的动作总结如下：

1. 关闭驱动总线的任何三态缓冲器（本例中将ena1到ena4设为无效）；
2. 允许ena2以打开32位三态门，将寄存器A中的内容驱动到共享总线上；
3. 允许ena3从而将总线数据反馈给寄存器B；
4. 禁止ena3从而将总线数据锁存在寄存器B中；
5. 禁止ena2从而释放总线以进行其他操作。

处理的详细过程可能会因设备而异（尤其是使能信号通常是在不同的时钟边沿触发的），但是这样的处理部件是需要的——按顺序给出——更重要的是在不同阶段之间需要足够的时间：

- 1到2：等待“关”信号沿着控制线传播，直到三态缓冲器并进行控制；
- 2到3：等待总线电压稳定（即寄存器A的内容由总线电压正确反映出来）；
- 3到4：给寄存器足够的时间去捕获总线上的值；
- 4到5：等待控制信号传递到寄存器，在总线为其他用途被释放之前寄存器停止“监视总线”。

有时等待时间非常重要。在现代处理器中，它是由系统的时钟周期计数的，每一个处理阶段被分配一个或多个时钟周期。

图3-3描述了在动作之间一个时钟的周期级时序，显示了在处理的每个阶段事件的顺序。显然需要一个同步的控制系统以执行这个动作序列，即使只是最简单的CPU指令序列。

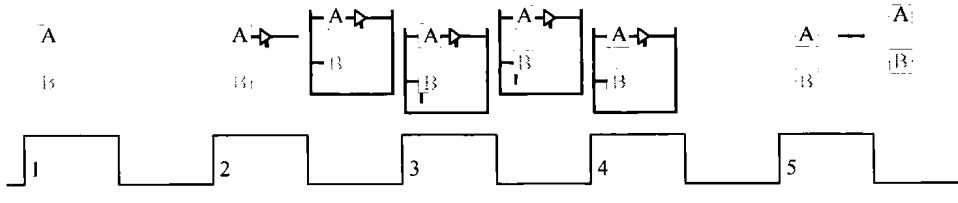


图3-3 一个简单控制单元（如图3-2所示）从寄存器A传输数据到寄存器B时的周期级时序，深色的线表示在那个时刻特定的总线或信号是激活的

不是所有的指令都需要按步骤经过相同的状态。有些指令，如没有返回结果的，能够早点终止。这些指令或者通过一个状态机完成，任凭其执行完所有状态（对那些没用的状态定义虚拟的动作），或者通过提早终止或自定义状态转换来完成。在这个给定的例子中，这样一条指令在
[71] 结束前不得完成全部的5个状态。

有些指令可能需要专门的处理，进一步扩展状态机。CPU设计者为了应付这种问题，通常会增加状态机的复杂度以解决这样的例外，所有这些都是为了提高运行效率。

多年来，越来越多怪异的或完美的指令相继被提出。没有一个天才能指出它们会在哪里结束——真是越来越复杂的状态机！有些情况下，CPU的控制单元是设计中最复杂的部分，最多会占用芯片一半的面积。而另一些情况下，状态机变得非常复杂以至于它自己就是另一个CPU实现——导致一个简单的处理器要应对一个更大、更复杂处理器的控制需求。在IC设计领域（和其他许多领域一样），众所周知复杂性会导致错误，正因为如此一些替代的方法正处于研究之中。

到目前为止，我们只考虑在一个处理器中处理不同指令的情况。现在，我们考虑这个实际的任务，就是随着不断增长的内部总线连接，更大的寄存器组、更多的功能单元和更高的时钟复杂度与灵活性。控制信号要分布在更大的甚至还在增长的IC面积上。期待有一个更大程度的处理器内部布线逻辑（即走线从设备的一边穿越到另一边）。这种难度已经超越了指令控制的复杂度。事实证明在一个IC硅片上，能够达到整个芯片的互连是一种稀缺资源：这通常都留给快速数据总线。把越来越多的这种资源用于专业控制目的的需求从另一方面促进了替代控制策略的研究。

因此产生了三种通用的方法学，称为分布式控制、自定时控制和简化（增加规律性）。分布式控制的一个主要例子是将在3.2.5节讨论的微指令的使用中。简化的例子可以在3.2.6节讨论RISC处理器时见到。让我们简要分析一下每个控制方法。

[72] 图3-4所示是一个非常简单的CPU内部的一部分，每个寄存器组有4个寄存器，两个ALU，都通过两组共享数据总线连接。总线上的每个点有一个三态缓冲器。每个总线、三态门、寄存器和ALU端口都是多位宽的。

显然，即使是这么简单的系统，从控制单元发出的单控制线也有很多。这些线用来控制每一个三态缓冲器和ALU的模式（ALU可以执行几种可选的功能）。图中有些线没有显示，如寄存器选择逻辑。在第4章及后续章节中，将讨论不同的总线安排，但像这样的控制信号线将不再画出，因为这些线使图变得过于复杂。

一个可行的简化办法是使用一根控制总线或几组控制总线，而不是图3-4所示那样每个寄存器需要两根控制总线。事实上，若要控制每根数据总线在某个时刻只能获取一个值，只需给每根数据总线一根2位的选择总线去驱动（即对于系统总共4位的控制总线）即可，这被称为寄存器选择（register-select）总线。这种方法在一个4个寄存器的系统中可能还看不到什么好处，但在32个寄存器的系统中会将寄存器选择控制线的数量从64个减少到6个。一个小例子如图3-5所示。

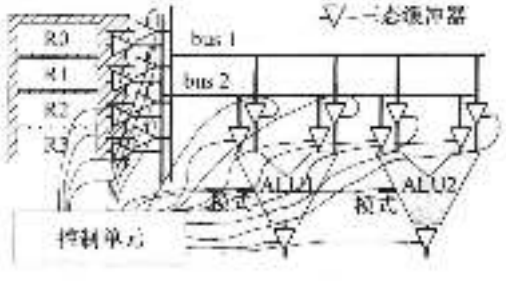


图3-4 一个简单CPU所需的集中控制线布线框图

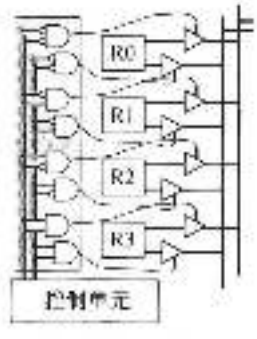


图3-5 一个小型控制单元连接到寄存器组的输入选择逻辑，该寄存器组包含4个寄存器

在图3-5中，从控制单元发出的到寄存器组的线的数量是4根，在寄存器组那里译码并选择合适的寄存器。这不一定是最小的逻辑，但最小化了CPU到组连接的数量。

总而言之，控制无处不在，如内部总线的种数，取指、译码和指令执行步骤的发起，与外界的交互（如I/O接口）以及将CPU内的每件事都排好序。控制甚至会扩展到对外部存储的处理，下一章在讨论内存管理单元时将介绍一个相关的重要例子。

自定时控制（self-timed control）是一个替代的策略，它将控制分布到整个CPU，这是基于大部分指令在处理器中都是遵循一个常用的“控制路径”——取指、译码、执行和存储。在执行阶段，以下过程也非常普遍——驱动某些寄存器书值到总线上，驱动总线上书值到一个或多个功能单元，一段时间后再把结果收集起来（还是通过一组或多组总线）并锁存回寄存器。 |73

此例中使用自定时控制并不暗示这是一个异步系统，因为每个模块都是可步的，尽管是以更快的时钟工作（注意我们将在第9章介绍在某些深奥的异步系统中使用自定时系统，但在这里我们只处理同步逻辑）。

一个集中式控制单元能够指出这个顺序，“正在取指”，然后“正在译码”，然后“正在执行”，最后是“正在存储”。这需要控制从负责每一个任务的IC区域回到中央处理单元的这些连接。然而，自定时控制策略要求控制单元只能是从指令读取开始执行过程，“正在译码”信号是由取指单元触发的，而不是从中央位置来的。类似地，“正在执行”信号是由译码单元产生并传递给执行单元的。按照这种方式，需要有从每个单元到下一个单元的控制互连，而不是所有的都回中央位置集中。实际上，控制信号是跟随数据通路的，在一个流水线机器（将在第5章介绍）上某些东西将变得更加有效。

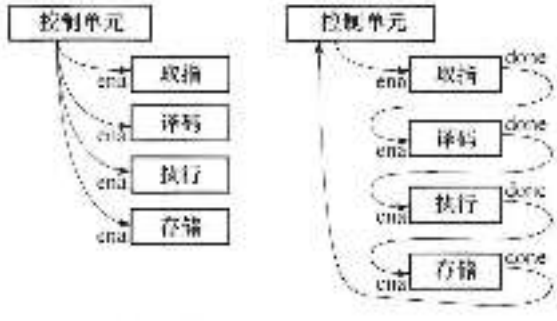


图3-6 集中式控制（左）和自定时控制（右）两种替代策略的控制流程图

集中式控制和自定时控制这两个替代方法的流程图如图3-6所示。此图中，数据总线没有画出，它可能来自外部存储器并依次流过取指、译码、执行和存储（FDES）过程。左边显示了一个包含4根控制总线的控制单元，每一根控制总线连接到4个独立单元的使能输入。在内部状态机指定的相关时间内，控制单元将启动FDES单元的操作。

根据正在处理的指令，控制单元状态机可能需要对FDES单元分别进行不同的控制（可能是较长的执行阶段或跳过存储）。这个知识必须被编码到控制单元中，即就是必须记住每一个与其相连单元的操作组合。

状态机首先必须包含详细的时序信息和每个单元的要求信息。它必须清楚经过这些单元的可能同时处理的多个指令。

图的右边是一个自定时系统：控制单元仍然会控制处理过程，但在这种情况下，每个后续单元都由前一个单元在需要的时候进行启动。因为这些单元本身要启动下一步，所以假设数据总线（没有显示）能够在正确的时间得到正确的信息。 |74

根据被处理的指令，这些单元可能会决定跳过它们自己，把请求直接传给下一个单元。因此每个单元必须把这个知识和时序编码到自己的功能中。

也许更麻烦的是需要把不同的信息传送给不同的单元，例如，执行单元需要知道要运行什么功能——是与、或、减法还是其他，它不需要知道执行的结果应存到哪里——而这个信息对存储单元来说是需要的，而反它不需要知道执行的是什么运算。在自定时情况下，或者将所需信息的全部字段送到一个个单元，每个单元只读取与自己相关的，或者先这些信息集中存储。实现兼

路的选择取决于对复杂度和性能的要求。

3.2.5 微指令

随着CPU变得越来越复杂，它们最终成为一个基本功能需求的组合体，带有越来越多的用户定制的专用指令，有些也已成为事实。一个例子就是20世纪80年代的Intel 8086、80386、80486处理器上的BCD码处理指令，就是为了反向兼容那些已经几十年的旧版商业软件。商业的需求驱动了CPU不断提高其处理速度，这种改进一方面是通过提高时钟频率来实现，另一方面是通过用一条指令执行更多的功能实现的。

复杂指令的提出更多是因为外存和内存之间的速度差异。内存需要花更多的钱，但其速度是外存的1000倍。一个大的瓶颈就是让指令从外存取到处理器中。因此有一个很好的想法就是创造单个复杂指令以替代原来的100条小指令序列。

实际上，我们可能会想到助记符。外部程序是以助记符（指令）写的，它被缓慢地送入CPU。每一个程序都会带来更长序列的内部操作。每个助记符可能会产生一个内部微程序序列，这些内部序列才是用微指令写的真正的程序。微指令（microcode）是该处理器的基本指令集，但往往与外部指令不太类似。所有的外部指令在进入CPU时，可能都要被翻译为微指令程序或微程序。

微程序设计作为一门技术，实际上是20世纪50年代初由Maurice Wilkes在剑桥大学发明的，虽然IBM System/360家族的计算机成员可能是第一台使用该技术实现的商业机器。

图3-7中概述了一些微指令的概念，其中外部程序存储在慢速存储器中，并通过CPU执行。当前程序计数器（PC）指向指令DEC A，可能是递减寄存器A中的值。这条指令被CPU取出
75 来，译码为一个微指令序列，包括将A的值加载到寄存器X中，然后把1加载到寄存器Y中，再用X减去Y，最后把结果保存回A中。

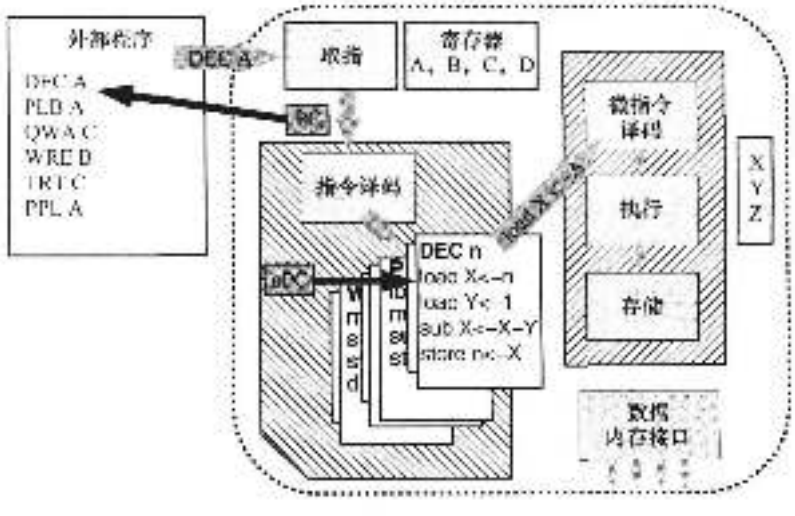


图3-7 指令执行概述：指令从外部慢速存储器取出，由CPU译码，以非常简单的微指令序列执行

由DEC指令产生的一个小型的4条指令微程序是存储在CPU内部的，位于模块化片上只读存储器（ROM）中，并需要一个内部的微程序计数器。它们在外部程序的"外部世界"是不可见的，这些外部程序甚至不知道寄存器X、Y和Z存在于CPU中。

进一步扩展该方法，会得到一个使用纳指令（nanocode）的处理器：外部程序转换成微指令

的微程序。每个微程序再进一步翻译成由纳指令构成的纳程序！尽管这项技术实际上只是简单的扩展，但比使用微指令方式还会进一步减少返回值。这个好处主要是依赖于外部存储器已成为瓶颈这个事实。在外部随机存储器（RAM）又慢又贵的内部ROM却非常快的时代，这样做无疑是正确的。但是随着RAM技术的改进，包括静态RAM（SRAM）、动态RAM（DRAM）以及随后的同步动态RAM（SDRAM）都削弱了ROM的速度优势，在20世纪90年代这些技术之间的差别已经不大了。

由于速度优势已不明显，因此微指令的普及性开始消退。

一个例外是这种指令翻译的好处还是有用的，微指令方法所固有的这个特点允许一种类型的CPU可以执行另一种机器的指令集。

在20世纪90年代末期，处理器继续改进，内部为RISC的机器却可以执行CISC指令集（见下一节）。没有哪个处理器能像x86系列处理器这样更清楚地表现出这一优势。这些CPU设计要追溯到1971年以来的遗留问题，因此不仅要保证代码的向上兼容性，使得能够执行古老的、并不优化的CISC指令集，还要比竞争对手的处理器运行更快。包含到这些处理器中的老式CISC指令将被翻译成更快的RISC风格的优化汇编序列。因此RISC指令取代了现代的微指令。 76

微指令翻译的另一个好处是一个处理器可以模仿其他设备。这样的处理器可执行一个ARM程序，就像一个本地的ARM处理器一样，它还可以转而执行德州仪器（TI）公司的DSP代码，就像是一个TI的DSP——是能够以各种CPU的身份运行各种程序的终极方法。

尽管有这么有利的市场，但微指令背后的驱动因素还是消失了，它在20世纪80年代已经不再流行。市场更倾向于能做得更多，更快的处理器；摩尔定律如火如荼。

3.2.6 RISC和CISC的对比

RISC（精简指令集计算机）和CISC（复杂指令集计算机）背后的思想已经在2.2节简单提到过。CISC体系结构包括许多复杂且功能强大的指令，而RISC体系结构则集中在只包含常用指令却处理快速的小型子集上。即使是复杂操作也被分解为多条RISC指令，它们和直接用一条CISC指令一样快甚至更快。

图3-8解释了这一概念。图中有两个程序：一个运行在RISC机器上，拥有快速的每周期一条指令的能力，在12个时钟周期内可以完成一个12条指令（A到L）的程序。它下面是一个CISC计算机，它的时钟周期更长些（因为硬件会更复杂，因此变慢），以差不多同样的时钟周期数完成同样的处理。但在这时只用5条复杂指令取代了RISC机器的12条指令。由于时钟周期更长，它完成任务要比RISC机器更慢。这是通常的情况，当然也存在其他条件，尤其是对更小的程序，CISC处理器能够计算得更快。

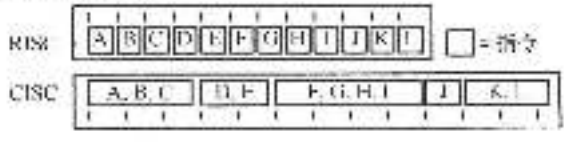


图3-8 本图阐述CISC指令和RISC指令的不同大小、速度和功能。RISC指令（上面的）都很短小，每条占用一个CPU周期，图中用竖线标出一相反，CISC指令（下面的）需要多个周期执行，通常每条指令比RISC完成更多的功能 77

然而这样一个无法并没有并述清楚这两种方法的前后关系，因此我们需要作一点事后解释。从历史的角度来看，早期计算机的使用者都是其设计者自己。设计者知道在他们的程序中需要什么基本操作，并把这些直接匹配到硬件中。随着硬件变得更能干，添加新指令到计算机中成为可能，这个新指令能够完成原来更耗时的一个指令串所能实现的功能。

随着时间的推移，计算机程序员专注于软件开发，而计算机架构师则只专注于硬件方面。程序员会找架构师要求自定义指令以使他们的程序运行更快。架构师通常都遵循建议，但有时他们也自作主张地添加他们认为有用的指令，却让程序员摸不着头脑。

到20世纪80年代中期，各种设计团体，尤其是美国的伯克利大学和斯坦福大学，开始质疑当时的设计理念。他们提出这个观点，可能基于在IBM悄悄进行的一些开创性工作，他们正在研究简单和常规设计在不太复杂的机器上可以运行更快的时钟。这些机器证明了简单的指令可以被处理得更快。尽管有时多条RISC指令才能完成一条CISC指令的功能，但一个RISC程序总体上还是明显运行得更快。

“精简指令集计算机”这个名字赞扬了原始设计的简单性，虽然没有实际的理由要减少指令集的大小，而仅仅是要降低指令复杂性。这些团体推广RISC技术，随之产生了RISC Ⅰ、RISC Ⅱ以及MIPS处理器。这些又演变为能够提供强大工作站性能的商业机器，它们不再向上兼容x86代码，即称为SPARC和MIPS的机器。

同时，在英国剑桥大学的橡子计算机有限公司（Acorn Computers Ltd），一个很小的设计小组基于早期伯克利的研究已经设计了他们自己的处理器，该公司非常成功地制造了基于6502的BBC微型计算机（为英国做出了贡献，使其具有世界上最高的计算机拥有率）。这台Acorn RISC Machine，即ARM1，是基于运行BASIC的2MHz BBC微型计算机设计的。Acorn为这款处理器编写了他们自己的芯片设计工具，在该处理器之后很快产生了ARM2，成为世界上第一台商业RISC处理器芯片。这增强了新兴的Acorn公司的计算机种类范围。到2002年，ARM，现在重新命名为Advanced RISC Machine，成为世界上最畅销的32位处理器，宣称占有76.8%的市场份额。到2005年中期，已售出超过25亿美元的ARM处理器产品，到2009年年初其销售数量已经增长到超过地球上人手一个的程度了。ARM处理器的普及程度还在继续增加。框3.1简要介绍了惊人的ARM处理器的开发背景。

78 当Intel坐享台式个人计算机的热潮时，ARM处理器则在享受嵌入式处理器更大的热潮。现在CPU几乎存在于每一个电子产品中，而其中大部分都是基于ARM的。同时Acorn公司现已不存在，已经在1999年注销。

框3.1 ARM是如何设计的

在20世纪80年代中期，具有开创性的英国计算机公司Acorn和英国广播公司（BBC）签订了合同要设计并推销BBC的微型计算机，当时该公司正在寻找一种方法来超越其获得巨大成功的8位BBC微型计算机。精简、高效的Rockwell 6502处理器推动了这一设计。BBC的举措大大促进了计算机在英国的使用，据报道，英国的人均计算机数量远远超过了世界上其他地方。例如，Clive Sinclair爵士的ZX系列计算机已经卖了400万台，而那时IBM PC的销售只达到100万台。Acorn公司也总共售出了超过100万台的BBC计算机。

在“计算机革命”的早期，Acorn公司很快意识到，来自如Intel和Motorola等公司的16位处理器已经不够强大，无法满足他们预计的未来需求——包括在20世纪80年代晚期发布世界上第一个多任务图形桌面操作系统（后来一些观察家得出结论，这被微软公司抄袭并成为Windows 95、XP及后续操作系统的基础）。

由于没有足够好的处理器，Acorn决定以典型的创业方式，创造自己的处理器。他们在两年内设计了ARM1以及它的支持IC（如MEMC和VIDC），尽管他们之前从未进行过任何芯片开发。

Acorn希望设计一个常规体系结构的机器——类似于6502，但功能要异常强大。他们选择采用RISC方法，还通过分析操作系统代码重新审视了软件需求，确定了最常用的指令，并为ARM处理器做了指令优化。他们用同样的方法得到一个指令集（见3.3节）及其编码。后来由于迫切需要又增加了乘法指令和乘累加指令。

全球知名的ARM处理器恰好符合英国政府基金资助的BBC项目的初衷，实现了多种优秀的功能：ARM软件中断、监控模式、快速中断、无微指令、静态流水线以及load-store体系结构，所有这些都从Acorn所采用的软硬件体系结构中派生出来。

3.2.7 处理器实例

多年来，自从IBM研究小组发表了他们的初步结果，RISC方法已经影响了处理器设计的几乎所有领域，尤其是ARM RISC处理器家族，现在主宰着嵌入式系统的世界。因此，本书中，几乎所有的汇编语言代码例子都是以ARM的汇编格式给出。例如：

```
ADD R0, R1, R2
```

79

是把寄存器R1和R2的内容相加，结果保存在寄存器R0中。

今天，虽然很容易找到“纯”RISC处理器的例子，如ARM和MIPS，但甚至铁杆的CISC设备（如Motorola 68000或Freescale Coldfire和一些Intel x86系列）现在也是由CISC到RISC硬件翻译和RISC内核来实现的。“纯”CISC处理器现在看来并不受追捧，出于这个原因，当提到CISC处理器时我们是定义了一个伪ARM汇编格式，而不是使用任何专门CISC设备的格式：

```
ADD  A, B, C
```

把寄存器B和C的值相加，结果放到寄存器A中。通常这里的例子都能被识别出是RISC的还是CISC的，它们是有区别的，因为RISC例子使用ARM型的寄存器R0到R15，而CISC例子使用字母寄存器A、B、C等。某些特殊用途的寄存器在后面的章节中也会提到，SP是堆栈指针，LR是连接寄存器。

本书中伪ARM指令使用的唯一一个例外是在讨论有关AD（Analog Devices）公司的ADSP21xx处理器和单独的德州仪器（TI）的TMS320例子时。尤其是ADSP使用的汇编语言其格式类似于C编程语言，因此相当易读。这些例外将会在介绍其代码段时突出显示。

注意某些处理器，尤其是68000，实际上是最后指定目的寄存器，而不是像ARM那样最先指定。然而，本书中目的寄存器都是以ARM风格指定的，且任何注释都跟在分号（“;”）后面：

```
SUB  R3,  R2,  R1  ;R3 = R2 - R1
```

有时目的寄存器和第一个源寄存器是同一个：

```
ADD  C,  D  ;C = C + D
```

或者可能只有一个源寄存器：

```
NOT  E,  F  ;E = not F
```

或者可能没有源寄存器：

```
B  R3  ;跳转至包含在R3中的地址
```

通常指令本身是能够表明其含义的（如ADD、AND、SUB等）。后面一节将介绍更多的例子，并详细介绍ARM指令格式，包括所有指令的列表。

在ARM中，还要小心目的寄存器在所有的指令中都是第一个指定的，只是在内存中写指令是不同的：

```
STR  R1,  [R3]
```

将R1的值存到以R3中内容为内存地址的位置。

80

3.3 指令处理

正如3.2节所述，计算机都是通过称为程序的指令序列执行的。通常称这些程序为软件。使用高级语言（HLL）可以编写出各种软件，每条高级语言命令可能由数十条CPU指令序列组成。低级语言的一条命令通常都只调用很少或者只有一个CPU操作。

如果我们将一个CPU操作定义为由CPU执行的数据移动或者逻辑处理，那么一条指令是从

程序到CPU的一个命令（会导致一个或多个CPU操作）。一条高级语言命令是由一条到多条指令组成的，一个存储在计算机上的程序是这样的指令序列集合。

在某些计算机上，一个单一的指令可用于调用多个CPU操作。这可能是由性能方面的原因决定的，尤其体现在访问速度上，从外部存储器上读取程序的速度远比CPU执行这些操作要慢很多。事实上，以前正是这种思想导致了微指令的出现（已经在3.2.5节中探讨过）。

机器代码通常指的是与CPU内已知动作相对应的二进制数字标识符。这可能意味着，例如，当你检查程序内存时，十六进制字节数0x4E及其后的0xA8可能代表了两条8位处理器指令，也可能代表一条16位处理器指令0x4EA8。对于现代的处理器，程序员几乎不需要应付处理器所能理解的底层二进制数字标识符，而是通过一套称为*汇编语言*或者*汇编码*的缩写助记符来进行处理。高级语言程序被编译成可执行程序时生成和存储的正是这些代码。

指令集是一系列可能的汇编语言助记符，是某具体CPU所支持的所有指令的列表。

3.3.1 指令集

指令集描述了CPU所能执行的操作集合，每个操作都由指令集中的一条指令来编码。某些指令需要一个或多个操作数（例如，在指令ADD A，B，C里，A、B和C被称为源操作数和目标操作数，这些操作数可能是立即数、寄存器、内存位置或者其他可用的寻址方式——详见3.3.4节）。通常操作数有类型和取值范围的限制，例如，移位指令所能移动的最大位数受移位硬件限制。

指令集包含所有的指令，从而能够完全描述处理器硬件的处理能力。指令集可按照命令所
81 涉及的处理器单元的不同而进行分组，例如下面为ADSP2181处理器所定义的指令集：

指令分组	组内操作举例
ALU	加、减、逻辑与、逻辑或等
MAC	乘法、乘累加等
SHIFT	算术/逻辑的左移/右移、指数运算等
MOVE	寄存器/寄存器、内存/寄存器、寄存器/内存、输入/输出等
PROGRAM FLOW	分支/跳转、调用、返回、循环等
MISC	空闲模式、无操作指令、堆栈控制、配置等

许多处理器还会添加一个FPU组或MMX组到指令集定义中，但是ADSP2181是只支持定点而没有多媒体扩展的处理器。

ARM处理器的指令集，确切地说是ARM7，以表格形式在图3-9中列出供大家参考（请注意，该表列出了ARM格式的指令，但不包括很多ARM处理器支持的16位Thumb模式）。该指令集表中使用的记号包括：

- S 第20位，标志着指令应该根据完成情况更新条件标志位（参见框3.2）；
- S 第6、22位，标志着转移指令是否该恢复状态寄存器；
- U 乘法是表示有符号/无符号，数据传输是表示向上/向下修改索引；
82 - I 立即寻址的指示位；
- A 结果累加/不累加；
- B 无符号字节/字；
- W 写回；
- L 读/存；
- P 向前和向后、递增和递减运算符；
- R 指明16个寄存器中的一个；
- CR 指示一个协处理器寄存器（可识别的8个协处理器中的一个）。

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	
条件标志位	0	1	1	×	×	×	×	×	× × × ×	× × × ×	× × × ×	×	×	×	1	× × × ×	未定义
条件标志位	0	0	I	操作码				S	Rn	Rd	第二操作数						数据处理
条件标志位	1	0	1	L	目的数偏移地址												分支
条件标志位	0	0	0	0	0	0	A	S	Rn	Rd	Rs	1	0	0	1	Rm	乘
条件标志位	0	0	0	0	1	U	A	S	RdHi	RdLow	Rn	1	0	0	1	Rm	长乘
条件标志位	0	1	I	P	U	B	WL		Rn	Rd	address offset						LDR/STR
条件标志位	0	0	0	P	U	1	WL		Rn	Rd	offset	1	S	H	1	offset	半字传输
条件标志位	0	0	0	P	U	0	WL		Rn	Rd	0 0 0 0	1	S	H	1	Rm	半字传输
条件标志位	1	0	0	P	U	S	WL		Rn	寄存器表							块传输
条件标志位	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn	BX
条件标志位	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	单数据交换
条件标志位	1	1	0	P	U	N	WL		Rn	CRd	CP no.	offset					LDC
条件标志位	1	1	1	0	CP操作码				CRn	CRd	CP no.	CP			0	CRm	CDP
条件标志位	1	1	1	0	CP操作码			L	CRn	Rd	CP no.	CP			2	CRm	MCR

图 3-9 ARM 指令集列表。列的上面标出了指令字的位。表中列出了某版本 ARM 指令集中所有的 14 类指令

这些修饰符多数是 ARM 处理器专用的，这里不再进一步讨论，但我们会详细关注位“S”以及寻址能力（见 3.3.4 节）。有兴趣的读者可以参考 ARM 公司的网站[⊖]，那里有更多的解释和文档资料。不同的 ARM 处理器之间的指令集略有不同，上面给出的版本是较常用的 ARM7TDMI 版本[⊜]。

近日，ARM 公司已经完成了品牌的重塑，在此过程中将他们的处理器命名为目前知名的 Cortex。原来的 ARM7、ARM9 和 ARM11 处理器被冠以“经典”。想必这一举动对整合巨大的 ARM 市场做出了努力，需要一个基本的体系结构（如 ARM）能够覆盖广阔的范围和多样的需求，其范围从微小、慢速的传感器系统到更大、更快的掌上计算机。在撰写本书的时候，新的处理器被归类成三个范围，更好地细分了 ARM 设备的传统优势领域：

Cortex-A 系列处理器是面向应用的。它们有内置的硬件支持，适合于运行丰富的现代操作系统如 Linux，及图形丰富的用户界面，如 Apple 的 iOS 和 Google 的 Android。它们能够处理从高效的 Cortex-A5 到 A8、A9 直到最高性能的 Cortex-A15 设备。这些处理器都支持 ARM、Thumb 和 Thumb-2 指令集（Thumb-2 据说在性能和紧凑性方面对 Thumb 做了改进）。

Cortex-R 系列处理器针对的是有高性能要求的实时系统。它们包括智能手机、媒体播放器和照相机。ARM 公司也在推进 Cortex-R 系列在汽车及医疗系统中的应用，在这种应用中可靠性和硬实时反应能力往往很重要。这些不需要复杂、丰富的操作系统，而只要小的、硬性且快速的实时系统。在写作本书的时候，只发布了 Cortex-R4，并已找到了其在全球许多实时系统中的使用方式。

Cortex-M 系列处理器属于低端产品，用于比较在意成本且要求低功耗的系统中，可以说它适用于传统的微控制器型应用，不需要先进的操作系统支持（甚至可能不需要任何操作系统）。该 83
系列适用于没有丰富图形界面要求、时钟速度不会超过几十兆赫兹的应用。撰写本书时，Cortex-M0 是入门级芯片，此外还有 Cortex-M3 和 Cortex-M4 可提供更好的性能。

尽管大多数 ARM7 的变种可支持 16 位 Thumb 模式（见 3.3.3 节），但所有的 ARM 芯片都支持上面所述的标准 32 位定长指令。我们看到在 ADSP21xx 中有各种指令组，如数据处理、乘法或分支等。如果有 15 个指令组，则需要 4 位编码来表示指令组，其余的位用来表示每组中具体的指令。

注意所有的指令都有固定的条件位，无论哪个指令正在使用，条件位都位于指令字的同一位置，这种规则有助于处理器的指令译码。重要的是要注意，条件位的意义是使每一个指令都可

⊖ http://www.arm.com。

⊜ 此信息是从 ARM 公司公开的文件 DDI 0029E 中提取的。

以有条件地操作。这是不寻常的。在常见的现代处理器中只有ARM是这样的：大多数其他处理器只支持条件分支指令。在ARM上，指令字中的S位控制该指令在完成时是否可以改变条件代码（见框3.2）。这两个特点相互配合使用，是非常灵活和高效的。

还要注意，对每一条指令，目的寄存器（如果需要）都在指令字中同样的位置，这一规则也简化了译码处理。

框3.2 ARM中的条件和S位说明

考虑ARM处理器的效率，与其语法最相似的标准RISC处理器（其所有指令都不允许条件操作）作比较。

使用内部下助记符来做于ARM（但不保证分级）。首先，我们以其在标准RISC处理器上的程序，将寄存器R0和寄存器R1中的值相加，然后根据计算结果判断，如果计算结果小于0则将0放入寄存器R2，否则将1放入寄存器R2。

```
        ADDS R0, R0, R1
        BLT pos1   如果小于0则执行分支
        MOV R2, #1
        B pos2
pos1    MOV R2, #0
pos2    ...
```

该程序使用5条指令且无论寄存器R0和R1中为何值，都会需要一个分支。

下面的代码段在ARM处理器中得到了相同的行为，但使用了条件转移代替分支。在这种情况下，R0和R1相加，在ADD助记符之后的S表明加法的结果会更新占据条件标志。接下来，如果前一个条件编码设置指令的结果小于0，则把1写入R2；如果结果大于等于0，则把0写入R2。

```
ADDS R0, R0, R1
MOVLT R2, #1
MOVGE R2, #0
...
```

ARM版本的代码显然更好——只需要3条指令，且不需要分支结构。正是这种机制使得ARM的程序更高效，而RISC处理器却是传统机器效率的代码密度，在更高级的语言中，这种代码结构则很常见：

```
IF condition THEN
    action 1
ELSE
    action 2
```

3.3.2 取指和译码

在一个现代计算机系统中，正在运行的程序通常驻留在RAM中（它们可能是从硬盘或闪存复制到这里的）。一个内存控制器，通常是内存管理单元（将在4.3节讨论）的一部分，控制外部RAM并代表CPU处理内存访问。

在CPU内部，取指和译码单元（IFDU或简写为IFU）在每个指令周期读取下一条待执行的指令。下一条指令主地址指针来确定，地址指针保存在程序计数器（PC）中，现在几乎每一款处理器都是如此。程序计数器通常在指令被读取后自动增加，但当有跳转或分支发生时会被新值覆盖。这些如图3-10所示。

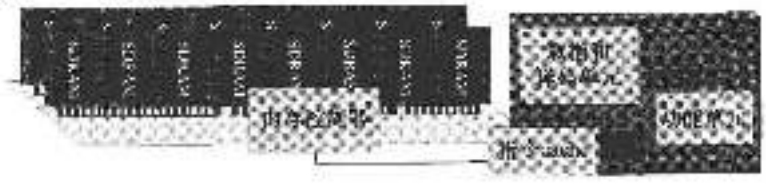


图3-10 此图显示了一个典型CPU系统中内存控制器的连接关系

一旦取指单元读取了一条指令，它便开始进行解码，随后经过一系列如图 3-11 的流程图所示的步骤。

84 - 85



图 3-11 典型处理器的指令处理流程图

3.3.2.1 指令译码

在 ARM 中，因为所有的指令都是有条件的，所以 IPU 首先看指令编码中的条件位，与当前处理器状态寄存器中的条件标志按位比较。如果这条指令需要的条件和当前条件标志不匹配，则丢弃该指令，读取下一条指令。

在 ARM 中，指令集的简化意味着每一个读取的指令字的条件位可以与状态寄存器的第 28 位到第 31 位进行简单与运算（这几位是当前条件标志的编码）。在 3.3 解释了 ARM 中可用的相当广泛的条件码集合。

框 3.3 ARM 处理器中的条件码

如图 3-9 所示，ARM 为每条指令保留 4 位（第 31、30、29 和 28 位）用于条件编码。这意味着每一条 ARM 指令都可以是有条件执行的（虽然并不编译器编写者只是在指令可能不得条件）。

通常，这些条件码是被附加到指令上的，正是 ADDGT 是一条 ADD 指令，只有当处理器的条件标志表明用于设置该条件标志的上一条指令的结果大于 0 时才会执行。

ARM 中条件的全集如下表所示（严格来讲最后两个是无条件的条件！）。

4 位条件编码	条件助记符	含义	标志条件
0000	EQ	等于	Z = 1
0001	NE	不等于	Z = 0
0010	CS	进位设置	C = 1
0011	CC	进位清除	C = 0
0100	MI	减	N = 1
0101	PL	加	N = 0
0110	VS	溢出设置	V = 1
0111	VC	溢出清除	V = 0
1000	HI	高于	C = 1, Z = 0
1001	LS	低于等于	C = 0, Z = 1
1010	GE	大于等于	N = V
1011	LT	小于	N = = V
1100	GT	大于	N = V, Z = 0
1101	LE	小于等于	(N = = V) or Z = 1
1110	AL	总是	—
1111	NV	从不	—

我们再看一下 ARM 指令集，所有指令字中的目的寄存器（对于有目的地址的指令）都位于同样的位置。译码时，IPU 简单地采用 4 位（用于寻址 16 个寄存器）作为寄存器组的目的地址。

3.3.2.2 取操作数

显然，操作数的值并不总是在指令字本身的编码中。ARM 和多数其他的 RISC 处理器都通过使用 load-store 体系结构进行简化，内存中的操作数不能直接提供给操作——它们必须先转移到

寄存器中。只有立即数是例外，立即数是作为数据传送指令的一部分编码的，如 MOV 指令（见框 3.4 中的例子）。

所以 ARM 为一个操作准备操作数通常是通过从指令字中解码一个立即数，或者从一个或多个源寄存器或目标寄存器中进行选择。一个例外是加载（LDR）和保存（STR）指令，它们实现内存和寄存器之间 32 位值的移动。

在许多其他的处理器中（通常是 CISC 体系结构而不是 RISC），可能执行一条指令就能够完成一个内存地址中数据的某些运算并将结果写回到另一个内存地址。显然在这样一个处理器中，来回移动操作数将需要一次到两次内存访问，由于 RISC 处理器的目标是尽量在一个时钟周期内完成每条指令，因此这种操作是不可能的。

86
~
87

3.3.2.3 分支

在 ARM 指令集的分支指令组中，正如所料，其所有的条件指令与其他处理器的分支指令是一样的。在分支指令中，第 24 ~ 27 位是表示指令属于分支指令组的唯一标识。L 位用来区分是跳转指令还是调用指令（ARM 术语中称之为 branch-and-link，其中 link 意味着返回地址放在连接寄存器 LR 中，在分支情况下放在 R14 中）。除了指令类型需要 4 位编码以外，还需要 4 位编码条件码，所以只剩下 24 位，这 24 位称为偏移量（offset），它们标明跳转到何处——目的地指令地址将会放在程序计数器中。

由于 ARM 是 32 位处理器，所以其指令字是 32 位宽。但是内存只有一个字节宽度，这样一条指令占用 4 个连续的内存单元。ARM 的设计者要求指令不能随便在某个地址开始，而只能在 4 字节边界处开始，即地址为 0、4、8、12 等。所以偏移量是指 4 字节的块。

现在计算机体系结构中通常有两种方式指定跳转地址，*绝对地址*和*相对地址*。绝对地址是指完整的内存地址，而相对地址是指从当前位置向前或向后偏移一定数量。随着计算机内存空间越来越大，指定绝对地址已经无法实现——局部性原理（见 4.4.4 节）指出分支的距离通常会比较小，与指定整个绝对跳转地址相比只需要较少的位数（实际上在 ARM 中使用了 28 位）即可指定。

再回到 ARM，跳转地址被称为偏移量，意味着它必须是一个和当前程序计数器中位置相关的跳转。对于 24 位偏移量，可表示的跳转范围是 2^{24} 个字，为内存的 64MiB。当然偏移量必须是有符号数，这样可以向前（如在循环中）和向后跳转，因此跳转范围应该是 ±32MiB。

有限的跳转范围是否成为一种限制？答案通常不是。尽管代码疯狂地膨胀，甚至在撰写本书时就如此，但单个的程序通常不会超过 64MiB 长度。因此，这可能是 ARM 的设计者们为照顾到指令的绝大多数跳转要求而设置的限制。

然而一个 32 位内存总线允许最多编址 4GiB 的内存空间，这远远大于地址跳转的能力，因此如果需要一个 70MiB 的跳转，该如何实现呢？

在这种情况下，ARM 提供一个*分支交换*（branch and exchange）指令。用这条指令，目的地址首先被加载到一个寄存器（32 位宽）中，然后这条指令发出命令跳转到该寄存器所保存的这个地址。当然，新的问题又来了，寄存器自己如何加载一个 32 位数呢？3.3.4 节将讨论寻址模式，其中一种是立即数——数值作为指令的一部分被编码。框 3.4 中也讨论了在 MOV 指令中如何加载立即数。

框 3.4 理解 ARM 中的 MOV 指令

和所有 ARM 指令一样，MOV 也是 32 位长，它的格式如下：

4 位条件码	0	0	I	操作码	S	Rn	Rd	4 位循环次数	8 位立即数

或者

4位条件码	0	0	0	操作码	S	Rn	Rd	立即数/寄存器移位和 Rm

4位条件码在所有ARM指令中都有，操作码（opcode）定义数据处理类的确切指令，Rn是第一个操作数寄存器，Rd是第二个操作数寄存器，通过设置第25位为1来选择，Rm是第三个操作数寄存器。

我们将主要针对第一种命令格式，这里提供了一个8位的立即数和一个4位的循环次数（实际的循环次数是给定值的两倍）。其中操作码（opcode）被指定为MOV指令，立即数经过指定次数的循环移位后加载到目的寄存器。下面是几个例子：

```
MOV R5, #0xFF     ;Rd=5, Rn=0, rotation=0, value=0xFF
MOV R2, #0x2180   ;Rd=2, Rn=0, rotation=2, value=0x43 (加载值:0x43≪4)
```

注意：对这些MOV指令，Rn由于没有用到因而总被置0。

问题：处理器如何将一个寄存器设为值0xF0FFFFFF?

答案：程序员可能会写成：

```
MOV R0, #0xF0FFFFFF
```

但汇编器可能会警告（“立即数的值太大”或类似的信息），因为指定的32位值无法放进8位寄存器，无论进行怎样的移位。某些汇编器或有经验的程序员知道将其转换为一个“move NOT”指令：

```
MVN R0, #0x0F000000  ;Rd=0, Rn=0, rotation=12, value=0x0F
```

正如你所看到的，尽管指令字段内只能容纳相对较小的立即数，但结合指令的灵活性和移位，实际上可以编码相当大范围的常数。

3.3.2.4 立即数

问题是，用32位的指令字，不可能传送一个32位的常数以及专用条件位、目的寄存器、S位等信息。因此立即数（编码在指令中的值）必须小于32位。

在ARM中，立即数通过MOV指令（属于数据处理指令组）加载到一个寄存器。立即数可以放到ARM指令集（图3-9）中标为“第二个操作数”的位置，但不是所有的操作数位都用于存放立即数，实际上，只有8位用于立即数，剩下的4位用来指定循环移位次数。

88 ~ 89

因此，虽然处理器有32位的寄存器，但只能加载8位数字。但由于有循环移位机制（用4位表示循环，可指定向左或向右的15个位置），可以获得很大范围的数字。框3.4详细展示了ARM处理器中MOV指令的位字段，我们可以看到其如何影响指令的灵活性。

许多处理器的工作方式不同。它们通常允许一次加载至少16位的立即数，这16位是作为指令字的一部分编码的。CISC处理器常常是可变长度的指令，或者使用两个连续的指令。一个变长指令在加载8位常数时可能是16位长的，当加载16位或24位常数时可能是32位长的。变长指令要求取指单元非常复杂，因此得到同样结果的更简单方法是使用两条连续的指令。第一条指令可能表示“取下一条指令中的数到寄存器R2”，因此IFU只需直接读取下一个值放到寄存器，而不用再进行译码。这显然意味着某些指令需要两个指令周期完成，从而带来时间代价，尤其是在流水线处理器中（见5.2节）。

以ARM处理器为例，尽管有对立即数的限制，但实际上很多常数都可以用一个8位值加上移位来编码，所以这并没有成为非常严重的性能瓶颈。ADSP2181也用类似的方式处理立即数负载，并已经设计为高速单周期操作。

3.3.3 压缩指令集

尤其是在变长指令的处理器中，*霍夫曼编码*（Huffman encoding）用于提高处理器效率。事实上，稍后我们将看到，类似的想法即使在一个定长的处理器中也可以使用，但这时效率不是主要原因。

霍夫曼编码的基本原理是减少最常用指令的长度而增加最不常用指令的长度，从而使平均

指令长度减少。显然，这需要有指令发生概率的知识，然后让表示指令的编码长度与其概率成反比。框3.5给出了一个应用霍夫曼编码的指令集设计实例。

应该指出，在现实世界中，相比于平均情况，一个特定的应用可能会出现非常不同的指令概率统计。

框3.5 霍夫曼编码简介

例如处理器有5条指令，为此做了一个1000条指令的软件程序的分析，得到各指令发生的次数情况如下：

CALL 60, ADD 300, SUB 80, AND 60, MOV 500

在该指令集中如果对各种指令的表示采取等长编码，则需要3位（因为那样最多允许7种可能）。忽略操作数，表示整个程序需要1000×3位=3000位。

处理器设计者希望采用霍夫曼编码来减少程序的大小。首先，计算每条指令的概率（通过用每条指令的发生次数除以总次数）：

CALL 0.06, ADD 0.3, SUB 0.08, AND 0.06, MOV 0.5

接下来，按照概率对它们排序，将概率最低的两个合并得到重排的序列，分别如下：

MOV 0.5	MOV 0.5
ADD 0.3	ADD 0.3
SUB 0.08	**C/A** 0.12
CALL 0.06	SUB 0.08
AND 0.06	

重复这个过程，直到只留下两个选项：

MOV 0.5	MOV 0.5	MOV 0.5	MOV 0.5
ADD 0.3	ADD 0.3	ADD 0.3	**C/A/S/A** 0.5
SUB 0.08	**C/A** 0.12	**C/A/S** 0.2	
CALL 0.06	SUB 0.08		
AND 0.06			

接下来，从右向左遍历这棵树，对每一列最下面的两项进行编号，上面的指定为二进制“1”，下面的指定为二进制“0”，这些数字必须在遍历过程中写下来，其他的列项只需简单地跟左边一样即可，不需要多写，直到到达左手边的原始指令则停止。

例如最右边的列，“1”表示MOV，“0”表示CALL/AND/SUB/ADD中的任何一个；向左，现在“01”表示ADD，而“00”是CALL/AND/SUB的前缀；下一列，“001”表示CALL或AND，“000”表示SUB。写下所有这些可以得到：MOV是“1”，ADD是“01”，SUB是“000”，CALL是“0011”以及AND是“0010”。如果我们关注每种操作使用的位数的话，看到最常用的指令（MOV）只用一位表示，而最不常用的指令（AND）需要4位，因此这种编码方法看起来在用较少位数表示最常用指令方面是有效的。利用原始的每种操作的发生次数和霍夫曼指令位数，我们可以计算新的程序大小：

$$(500 \times 1) + (300 \times 2) + (80 \times 3) + (60 \times 4) + (60 \times 4) = 1820$$

这显然大大小于固定3位表示法所用的3000位。

多数ARM处理器包含另一种16位指令集，称为Thumb。该设计是为了提高代码密度。但是请注意，即使给定内存的大小可以支持两倍的Thumb指令，相比于32位ARM指令，平均来讲，实现和译码后所映射的底层ARM指令相同的功能也需要更多的Thumb指令（这主要是因为供选择的不同Thumb指令较少）。

ARM工程师设计Thumb指令集的过程是值得注意的，因为他们使用了类似于霍夫曼编码的

想法。ARM工程师考察了一个应用程序代码实例的数据库，并计算出每个指令的使用次数，只有最常见的指令才会出现在Thumb模式中。在固定的16位指令字里面，用于表示指令的二进制编码是基于其他操作数所需位数的长度编码。

90～91

Thumb指令集的一些特点如下：

- 只有一个条件指令（偏移量跳转）；
- 没有“S”标志位，多数Thumb指令自动更新条件标志；
- 目的寄存器通常和源寄存器是一个（在ARM模式中，目的和源通常总是分开指定的）；
- 所有的指令都是16位的（但寄存器和内部总线宽度还是32位）；
- 立即数的寻址方式和偏移地址非常受限；
- 多数指令只能访问（16位寄存器中的）低8位寄存器。

Thumb指令集的复杂程度远远超过ARM指令集，尽管其译码过程（从Thumb指令被从内存中取出到ARM指令准备好在处理器中执行）是自动完成，并且速度很快。下面是指令的一些例子：

16位二进制指令位模式			指令名称	举例
1101	条件（4位）	偏移量（8位）	条件分支	BLT loop
11100	偏移量（11位）		分支	B main
01001	目的寄存器（4位）	偏移量（8位）	读内存中的数据到寄存器	LDR R3，[PC，#10]
101100001	立即数（7位）		堆栈加法	ADD SP，SP，#23

92

从这里给出的有限的例子可以看到，少数的几个最高位表示指令，实际上整个指令集的这个长度范围是从3位到9位。在所示的ADD指令中，在其上操作的寄存器是固定的：只能和堆栈相加——几乎所有指令均可在任何寄存器上操作这种ARM指令集的灵活性和规律性丢失了——这是考虑到了软件中最常用的操作。

应该指出一点，Thumb指令集是16位宽，当到外部存储器的接口是16位时它才真正处于其最佳状态，在这种情况下，每个ARM指令需要两个内存周期来读取（导致处理器只能以其最高速度的一半运行），而Thumb代码可以全速执行。

3.3.4 寻址模式

寻址模式描述了确定指令中操作数的不同方法。有许多种操作指令，它们可能不含操作数，可能含一个操作数，或者两个、三个操作数。特殊情况也有超过三个操作数的指令。多数现代处理器中常见的非零操作数的例子如下：

类型	例子	操作数
单操作数	B address	地址可能是直接给出的，可能是一个距离当前位置的偏移量，或者可能是一个存储在寄存器/内存位置中的地址
两操作数	NOT destination，Source	目的或源操作数可能是寄存器、内存地址或由寄存器指定的内存地址。源操作数还可能是一个立即数
三操作数	ADD destination，source，source	目的或源操作数可能是寄存器、内存地址或由寄存器指定的内存地址。源操作数还可能是一个立即数

当然，并非所有可能的操作数类型都适用于所有的指令，尽管如此在某些处理器中有些操作数还不能使用（例如RISC处理器的load-store，通常限制算术运算指令的操作数是寄存器，而在CISC处理器中它们可以存储在内存中或其他地方）。最后一点需要注意的是在上面例子中的

后两个，假设其第一个操作数是目的数——对于ARM汇编语言即如此，但对其他某些处理器则相反（见3.2.7节）。这可能是对不同处理器编写汇编代码时引起混淆的真正原因（也是计算机体系结构讲师/作者的职业风险所在）。

93 术语寻址模式（addressing mode）指加载或存储时地址的表示方式，使用一种或几种不同的技术。下表列出了常用的寻址模式，以ARM风格的汇编语言为例（虽然应该指出PUSH不存在于ARM指令集中，只在Thumb中才有）。

名称	举例	解释含义
立即数寻址	MOV R0, **#0x1000**	传送十六进制值0x1000到寄存器R0
绝对寻址	LDR R0, **#0x20**	加载内存地址为0x20处的内容到R0
寄存器直接寻址	NOT R0, **R1**	将R1中的内容取反并存储到R0中
寄存器间接寻址	LDR R0, [**R1**]	如果R1中有值0x123，那么取出内存中0x123位置的内容并放入R0
堆栈寻址	PUSH R0	在这种情况下，R0中的内容被入栈（假设只有一个堆栈）

如下扩展和组合的基本思路也很常见：

名称	举例	在R1=1 & R2=2时
寄存器相对间接寻址	LDR R0, [**R1, #5**]	第二个操作数的内存地址为1+5=6
基址加变址寄存器间接寻址	STR R0, [**R1, R2**]	第二个操作数的内存地址为1+2=3
相对基址加变址寄存器间接寻址	LDR R0, [**R1, R2, #3**]	第二个操作数的内存地址为1+2+3=6
寄存器移位间接寻址	STR R0, [**R1, R2, LSL #2**]	第二个操作数的内存地址为1+(2<<2)=9

各种处理器，包括ARM和ADSP2181，还提供一种自动方式以便在寄存器被用做偏移地址后对它们进行更新。例如，用立即数偏移的寄存器间接访问，在偏移量加完之后会更新这个寄存器。如下面例子所示，其中R1=22：

```
LDR R0, [R1], #5    将内存地址22处的内容加载到R0中，然后设置R1=22+5=27
```

94 `LDR R0, [R1, #5]!` 设置R1=22+5=27，然后把内存地址27处的内容加载到R0中

注意，我们并不想在这里教大家ARM指令集的细节，只是以此作为底层寻址技术的辅助教学。⊖

分析这些局限性使得CPU的设计者必须在处理器中提供某些层次的功能——除了关注指令集之外，没什么更能揭示这些限制，在这方面CISC处理器更有意思。下面给出了一些例子，假设有一个CISC处理器，主存地址mA、mB和mC用于绝对操作数存储，还有一个RISC处理器，其中寄存器R0、R1和R2用于寄存器直接寻址：

- **CISC处理器：** `ADD mA, mB, mC ; mA = mB + mC`

  这里，一旦CPU读取了指令并进行译码，它必须进一步读取存储了操作数值mB和mC的两个内存地址的数据，这可能需要两个内存总线周期。然后这些值被内部总线传送到ALU（由于是顺序进行的，因此只需要一组总线）。一旦ALU计算出结果，则结果通过总线传送到内存接口，以写回到主存中mA的位置。

  该指令的开销除了ALU的运算时间之外，还有三个外部存储周期。外部存储周期通常远远慢于内部ALU操作，所以这显然是一个瓶颈。这只需要在处理器中有一个内部总线即可解决。

  指令字中必须包含三个绝对地址。对32位内存，等于96位，构成一个很长的指令

⊖ 建议那些想要了解ARM指令集的读者参考《ARM System Architecture》这本书，作者是Steve Furber（ARM处理器的发明者之一）。

字。这可以通过偏移量或相对地址来缩减位数，但对32位来说可能仍然太长了。

- **RISC处理器：** `ADD R0, R1, R2 ; R0 = R1 + R2`

  同样的操作现在用寄存器来完成。所有操作数的值都已经在CPU内部，这意味着可以快速地访问它们。一旦指令被读取并译码，寄存器R1将驱动一个内部操作数总线，同时寄存器R2驱动另一个内部操作数总线。因此在一个很快的内部总线周期，两个操作数被送到ALU。一旦ALU计算完结果，一个内部结果总线将获得该值。R0将会监听这个总线，在适当的时候，从该总线上锁存结果值。

  指令的开销除了ALU的运算时间外，还有两个快速内部总线周期。在我们讲述的例子中，CPU必须包含三个内部总线：两个同时传送两个操作数，一个收集结果。其他类似的安排也是有可能的。 95

  指令字需要包含三个寄存器值，然而，对一个有32个寄存器的寄存器组来说，只需要5位即可指定每个寄存器，因此总共需要15位。这很容易将操作编码在32位宽的指令中。

- **CISC处理器：** `ADD mA, mB ; mA = mA + mB`

  类似于第一个例子，CPU必须读取存储了操作数值的两个外部存储位置，需要两个内存总线周期。也需要将结果传送回内存，因此执行时间没有变。

  但是，这次指令字只需要包含两个绝对地址而不是三个。这在实际系统中是可行的，尤其是如果第一个操作数用绝对地址而第二个用偏移量的话。

- **CISC处理器：** `ADD mB ; ACC = mB + ACC`

  20世纪80年代及更早的CISC处理器通常是利用累加器。这些是通用寄存器（寄存器组的前身）被用来作为所有算术运算和数据模式操作的操作数，并保存这些操作的结果。另一个操作数几乎总是内存中的一个绝对值。本例中，指令需要在加法之前从内存加载一个值，因此涉及一个外部存储总线周期。

  指令字只需要包含一个内存绝对地址，这可以通过加载一个包含该地址的第二个指令字来实现（因此在指令执行前需要先读取两条指令）。

- **堆栈处理器：** `ADD`

  这是一个特殊情况（将会在下一节尤其是第8章进一步讨论），CPU弹出栈顶两个数，相加后再将结果入栈。这需要访问栈，如果栈是一个内部存储块，访问会很快，但栈更多的是位于片外存储上。使用堆栈方法的主要好处是指令不再需要编码任何内存地址。从理论上说，这可以得到一个非常小的指令宽度。

3.3.5 堆栈机和逆波兰表示法

人们通常采用*中缀*表示法表示一个写在纸上的运算（如$a+b\div c$），其中公认的运算符优先级[⊖]（可以通过使用括号改变）决定了各种运算执行的顺序。波兰表示法（注意不是逆波兰表示法）是 96
由波兰数学家Jan Lukasiewicz在20世纪20年代发明的，将运算符放在操作数的前面，因此是一个*前缀*表示法。通过这种方式指定操作数，运算符优先级则不重要了，因此不再需要括号。

相反，逆波兰表示法（RPN）是一个*后缀*表示法，等式的顺序即完全定义了优先级。这是在20世纪50～60年代，为辅助基于堆栈体系结构的工作而创造的。它后来被两代HP电子计算器用户所引进并迷恋（或讨厌）。

⊖ 许多读者可能还记得，在小学学算术时，为了辅助记忆优先级所学的BODMAS缩写。BODMAS代表：括号（Braclcet）、级数（Order，即幂和平方根）、除法（Division）、乘法（Multiplication）、加法（Addition）和减法（Subtraction）。见http：//www. malton. n-yorks. sch. uk/MathsWeb/reference/bodmas. html。

RPN 的一个例子如 $bc \div a +$，其中操作数 b 和 c 先给出，且随后是相除的命令并保存结果。然后操作数 a 被加载，并跟随一个加法命令，把前面的结果加到 a 上，并将新的结果存储在某个地方。以下是更多的例子并且如图 3-12 所示。

中缀	后缀	中缀	后缀	中缀	后缀
$a \times b$	$ab \times$	$a+b+c$	$ab+c+$	$(a+b)+c$	$ab+c+$

考虑这几个操作，显然使用堆栈是执行 RPN 运算的一个有效方式。这种情况下，一个堆栈是一个存储设备，有单一的入口/出口。数字可以被推入到堆栈的“顶”部，再从“顶”部弹出去。这是一种后进先出（LIFO）结构。

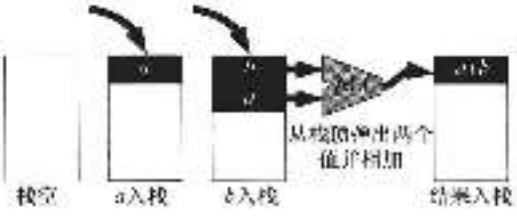


图 3-12 堆栈处理过程示意：两个操作数按顺序入栈，操作数出栈，ALU 执行运算，计算两个数的和并将结果入栈

图 3-12 显示了一个用堆栈运行 $ab+$ 的例子，从左向右读。需要注意的事情有，每个步骤只有一个入栈发生（在基于堆栈的处理器中每一步可能需要一个周期），尽管弹出的数字个数由该操作所需的操作数个数决定。例如，ADD 需要两个操作数，因此通过两次 POP 操作将操作数加载到 ALU。每个运算操作的结果被推回栈顶。

97

考虑用这样一个堆栈机去执行一个复杂程序任务会很有趣。对简单操作看起来似乎有效，但有时，堆栈经过一系列操作后的最终状态可能是在栈顶没有得到正确的结果。这可能是由多任务或中断服务例程引起的。必须有一个堆栈重新排序的方式，如将数据项弹出栈并存储到主存，再以不同的顺序将它们推入栈。这可能是一个非常耗时的过程，会严重影响堆栈机的整体性能。这一过程也在框 3.6 中进行了讨论，重新排序以尽量缩减堆栈的使用。

框 3.6 重编码 RPN 指令以缩减堆栈空间

考虑中缀表达式 $a+(b \times c)$，由于加法顺序对最终结果并不重要，因此该表达式也可以写成 $(b \times c)+a$。

对每个表达式，写下其后缀表达式，并写出执行需要的堆栈操作顺序。考虑每个表达式的堆栈使用情况。

显然正确，表达式的一种写法使用的最大堆栈深度是 3 层位置，而另一种写法使用的堆栈深度只有 2 层位置。

看来，后缀表达式的顺序可以显著影响所需的堆栈资源（因此也是硬件资源），尽管这并不会改变整个计算所需的步骤数。

并非所有的中缀表达式都对顺序不敏感，加法和乘法可以，但除法和减法是绝对不行的。

3.4 数据处理

到目前为止，本章一直在讨论 CPU 基础——什么是计算机和计算机的组成。我们已经提到了指令和程序等。作为其中的一部分，3.3 节讲述的是指令处理，包括一些变量，以及寻址方式等重要子课题。

后面的 3.5 节将从自顶向下的角度去看计算机。然而，在高层次总览和低层次细节这两个极端之间，有一个哲学问题，那就是计算机的目的是什么。我们可以借用“黑匣子”（black box）[⊖]来举例，从黑匣子的角度，我们把计算机看做是一个这样的东西：它修改一些输入数据，产生一些输出数据。

98

输入数据和输出数据都有很多种形式：指令、知识、传感器采集的数据、多媒体等。对于某些系统，输入数据可能是由某个单一触发事件组成，输出数据可能是由一些驱动器开关执行信

⊖ 对于那些从来没见过这个词的人来说，“黑匣子”通常用于描述一个功能模块，它只根据该模块的输入和输出给出定义，它不关心模块内部是怎样的，只要给定正确的输入就能够产生正确的输出。

号组成。在控制系统中就是这样的，它通常是因为实时处理数据（实时问题将在6.4节深入讨论）的需求而执行的。某些系统是数据密集型的——输入或者输出由稠密的数据流组成，比如数字音频或者视频，这些系统也需要实时处理。然而，大部分的计算机系统都是通用机器，它既能够执行控制也能够执行数据处理任务，很少涉及实时问题。

这里我们讨论的主题是数据：计算机处理数据，无论它是一个一位触发器，有非常严格的时序要求，还是1TB的多媒体数据，只需要几分钟的时间来完成数据处理。本节将讲述计算机的这些重要内容：什么是数据？如何产生、存储和处理数据？

3.4.1 数据的格式和表达

在2.3节我们已经讨论了一般的数字格式，包括和计算机最相关的（无符号二进制数、补码等）。无论使用的是哪种格式，数的宽度——由一个数占用的位数——能够由计算机体系结构设计人员进行调整，或者增加所能存储数据的最大绝对值，或者增加其精度。通常情况下，因为计算机是基于字节的，所以数的位宽是8的倍数。

大多数CPU有一个默认大小的数据格式，这是由内部总线的宽度决定的，例如，在老的6502处理器上是字节宽，在ARM上是32位宽。尽管ARM也可以处理字节和16位的半字，但它以32位的方式访问主存，因此处理32位的值并不比处理字节慢。在ARM中，寄存器、内存单元、大多数操作数等都是32位宽。

程序员通常采用像C这样的高级语言来处理内存或寄存器中的数据。尽管某些编程语言严格按照该语言所提供的数据类型来定义数据格式，但在C语言中并非如此，只是定义一个字节时总是指8位宽。

通常情况下，虽然实际上是由所使用的特定C编译器决定，但与整型数据类型相匹配的是处理器的默认位宽，即16位宽或以上。因此，在16位机器上的整型数通常会是一个16位数，而在64位机器上往往是64位宽。

程序员应注意：如果你想编写可移植代码，请确定没有对整型、短整型数等的确切大小做出任何假设。表3-1给出了gcc编译器针对不同的目标处理器时几种数据类型的宽度[⊖]。一些原始 99
的C语言数据类型其性质的不断变化，导致许多开发人员采用特定长度的数据类型，这在框3.7中给出了进一步说明。

表3-1 从8位CPU到64位CPU，比较其上的C编程语言数据类型的数据宽度。注意不同处理器之间哪些数据类型的大小发生变化，而其他保持不变。对于一个特定的实现，数据宽度通常通过在配置头文件types.h中定义最大和最小的具有代表性的数来实现。还要注意，其字节顺序在大小尾端处理器上也会不同（见2.2节）

C语言数据类型	8位CPU	16位CPU	32位CPU	64位CPU
char	8	8	8	8
byte	8	8	8	8
short	16	16	16	16
int	16	16	32	64
long int	32	32	32	64
long long int	64	64	64	64
float	32	32	32	32
double	64	64	64	64
long double	由具体编译器决定——可能是128、96、80或64位宽			

⊖ 注意某些编译器的实现会有所不同，可能不符合ISO或ANSI C语言规范。

当然，有经验的程序员会知道，C编程语言中的任何整数数据类型（即表中前6行）可以被指定为有符号或无符号。默认（如果没有指定）的数据类型是有符号补码。

long int和long long int也可分别被定义为long和long long。除了最大的机器外，在所有机器中，这两种类型都需要多个内存位置进行存储。

char类型通常使用7位的有效值，遵守ASCII标准（美国标准信息交换代码），如表3-2所示。任何一个最高位被置位的字符（即一个第8位非零的字符）将被看做是一个扩展的ASCII字符（未在表中列出的ASCII字符）。有趣的是，低于十进制数32（space）的字符和包括十进制数127（delete）的字符是非打印字符，具有最初为电传终端所定义的特殊值。例如，ASCII字符8即“\b”是响铃字符，输出时将导致一个“哔”声。简单在网上搜索一下，可以很容易地找到其他特殊的ASCII字符的含义。

当使用计算机的人是以英语为母语时，ASCII码是非常出色的，但对于使用其他语言的人则
100 不是特别有用。因此，人们多年来一直付出巨大的努力希望可以为其他语言定义不同的字符编码。也许最终极的挑战是汉语，它拥有近13 000个象形字（独立符号）：很明显用一个8位的数据类型是不可能把汉语言文字编码的。在过去的二十年已经出现过许多解决方案，其中大部分使用两个或两个以上的连续字节来保存单个字符，目前成为事实标准的编码称为Unicode，这种方式采用完全不同的风格，但它可以使用最多四个连续字节对绝大多数的字符编码，其中包括汉语、日语、韩语等。

框3.7 嵌入式系统的数据类型

虽然用C或C++语言编写的程序通常使用如表3-1所示的标准数据类型，但当移植代码时还是可能会造成混乱。如果程序员对一个特定数据类型的大小做了一个隐含的假设，那么在另外的处理器上编译时，这种假设可能不再正确。

在gcc编译器被广泛采用之前，这种情况实际上更糟——许多编译器有自己的编译模式，比如“大内存模式”和“小内存模式”，这可能会导致表示变量的位宽不同（虽然gcc已可以通过命令开关来改变这一点，但不经常使用）。在嵌入式系统中，其目标机器可能不同于主汇编机，因此交叉编译格外重要，以确保在主机上测试过的所有代码也能够在目标机器上执行。

为了实现这样的目标，并保持对不同数据类型的限制，也许最简单的方式是在声明变量时直接指定每个类型的大小。在C99编程语言（发布于1999年的C正式版）中，这些已在<stdint.h>头文件中为我们定义好了。

大小	无符号	有符号	大小	无符号	有符号
8	int8_t	uint8_t	32	int32_t	uint32_t
16	int16_t	uint16_t	64	int64_t	uint64_t

64位的定义（以及其他不常见的位宽，如24位）可能存在于某个特定的处理器实现。当然，如果它存在，就会占用给定的位宽，但除此之外这些都是可选的，所以对于一些机器，编译器将只支持主流的8位、16位和32位的定义。比起那些写台式机软件的编程员，嵌入式系统的编码人员往往更经常遇到这些较安全的类型声明。笔者鼓励嵌入式系统的开发人员如有可能尽量使用指定宽度的数据类型。

表3-2 美国标准信息交换码，7位ASCII表，表中列出了字符（或非打印字符的名称/标识符）以及十进制和十六进制代码表示

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
\0	0	0x00	(spc)	32	0x20	@	64	0x40	`	96	0x60
(soh)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(stx)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62

（续）

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(etx)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(eot)	4	0x04	$	36	0x24	D	68	0x44	d	100	0x64
(enq)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ack)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
\a	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
\b	8	0x08	(	40	0x28	H	72	0x48	h	104	0x68
\t	9	0x09	)	41	0x29	I	73	0x49	i	105	0x69
\n	10	0x0a	*	42	0x2a	J	74	0x4a	j	106	0x6a
(vt)	11	0x0b	+	43	0x2b	K	75	0x4b	k	107	0x6b
\f	12	0x0c	'	44	0x2c	L	76	0x4c	l	108	0x6c
\r	13	0x0d	-	45	0x2d	M	77	0x4d	m	109	0x6d
(so)	14	0x0e	.	46	0x2e	N	78	0x4e	n	110	0x6e
(si)	15	0x0f	/	47	0x2f	O	79	0x4f	o	111	0x6f
(dle)	16	0x10	0	48	0x30	P	80	0x50	p	112	0x70
(dc1)	17	0x11	1	49	0x31	Q	81	0x51	q	113	0x71
(dc2)	18	0x12	2	50	0x32	R	82	0x52	r	114	0x72
(dc3)	19	0x13	3	51	0x33	S	83	0x53	s	115	0x73
(dc4)	20	0x14	4	52	0x34	T	84	0x54	t	116	0x74
(nak)	21	0x15	5	53	0x35	U	85	0x55	u	117	0x75
(syn)	22	0x16	6	54	0x36	V	86	0x56	v	118	0x76
(etb)	23	0x17	7	55	0x37	W	87	0x57	w	119	0x77
(can)	24	0x18	8	56	0x38	X	88	0x58	x	120	0x78
(em)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(sub)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(esc)	27	0x1b	;	59	0x3b	[	91	0x5b	{	123	0x7b
(fs)	28	0x1c	<	60	0x3c	\	92	0x5c	|	124	0x7c
(gs)	29	0x1d	=	61	0x3d	]	93	0x5d	}	125	0x7d
(rs)	30	0x1e	>	62	0x3e	^	94	0x5e	~	126	0x7e
(us)	31	0x1f	?	63	0x3f	_	95	0x5f	(del)	127	0x7f

虽然这种编码系统的细节已经超出了本书的范围，但其影响还是要提一下：早期的计算机是字节宽的，自然能够处理字节大小的 ASCII 字符。现在，它需要一个 32 位的机器，在一个单一的操作中处理 4 字节的 Unicode 字符。同样，早期的接口也是基于字节的，如 PC 的并行端口和串行端口（见第 6 章）。内存访问也常常是基于字节的。有观点说，对于简单计数和文本处理来说，字节是一个便于处理的宽度。然而，这种说法在许多情况下不再适用。对非英文字母的系统而言，一个字节宽的处理系统只是一个历史而已。

有关数据的大小，最后一点需要注意的是 float 和 double 类型的一致性。这种一致性得益于 IEEE754 标准的普及，以及事实上大多数的硬件浮点单元都符合标准（这一点将在 4.6 节解释）。

3.4.2 数据流

仍然采用黑匣子的观点，一台计算机需要输入、处理，并产生输出。显然，对这些数据在实时性、数量、质量等方面的要求非常重要。

101
~
103

今天的计算机，尤其是许多消费类嵌入式电子系统，大量是以用户为中心。这意味着，输入或者输出需要与人交互。某些数据也相当庞大（如视频、音频等）。总线宽度需要按照数据的流量来确定，并且系统也应考虑人的需求，其中总线我们会在第6.1节更充分地讨论。例如，人类的感觉器官往往对突然的中断比连续的错误（噪声）更为敏感。通常情况下，当听者从CD播放机听音乐时，漏音比听有嘈杂背景的声音更加让人恼火。视频也一样：跳帧比看稍微有噪声的画面更恼人。

大多数重要的实时问题将在第6.4节探讨。然而，在这一点上，我们需要强调的是，计算机体系结构的设计者应牢记，他们设计的系统将会用在什么地方。嵌入式计算机体系结构设计者可能具有的优势是，他们的系统更灵活、更通用，因而能更好地以满足用户。不幸的是，他们也有许多不利之处，诸如对系统规模、成本和功耗的限制更严格，因此需要更精细的权衡取舍设计。

从技术上讲，计算机中数据流途径的通路称为总线。这个数据可能来自外部设备或某种形式的数据存储，在CPU或协处理器上以某种方式得到处理，然后同样输出到另一个外部设备或数据存储。

3.4.3 数据存储

存储层次结构图3-1突出显示了在存储架构方面嵌入式系统和典型的台式机/服务器系统之间的差异：除了一些类似iPod的设备以外，在嵌入式系统中的数据存储通常是基于闪存的。而在台式系统中，它往往是存储在硬盘（用于短期存储）上，或磁带/CDROM或DVD（用于备份存储）上面。

计算机“内”的数据是存储在RAM、高速缓存存储器、寄存器等设备内部的。从程序员的角度来看，它是存储在寄存器或者主存（因为高速缓存通常对程序员是不可见的）上。数据从外部设备或者硬盘通过总线（见6.1节）进入内存，这些数据传输或者按字节或字单次传送，或者按照突发模式，或者采用直接内存访问方式（有关DMA的内容见6.1.2节）。大量数据占据在内存页中，由内存管理单元（见4.3节）处理，少量可能存在于固定变量区或系统堆栈中。由于嵌入式系统通常使用并行总线连接闪存设备，因此这种系统中的数据已经可以被主处理器直接访问，从而将这些数据看做已经在计算机“内部”。

数据从内存调入CPU进行处理，同样可能按单次模式传输或整块传输。对于load-store机器（见3.2.3节），待处理的数据必须首先被加载到各个寄存器，因为所有的处理操作只从寄存器获取输入数据，且只输出到寄存器。有些专用机器（如向量处理器）可以直接处理数据块，有
104 些机器有专门的协处理单元，可以直接访问内存，而不需要CPU来处理内存数据的加载和存储。

3.4.4 内部数据

编译C代码时，编译器决定如何处理程序变量。有些变量，通常是最经常访问的变量，当它们正在被访问的时候是放在寄存器中的。然而，大多数处理器没有足够多的寄存器，所以只有极少的变量会有这种待遇。

在整个程序执行过程中，会为全局变量指定一个专门的内存地址，而其他性质的变量则存储在内存堆栈中。这意味着，当一个程序包含如“i++”这样的声明时，其中i是一个局部变量，编译器认为其不能保留在寄存器中时，编译器会为该变量在堆栈中提供一个位置。在load-store机器上该语句执行的伪代码指令如下：

1. 将特定堆栈偏移量中的对应于变量i的数据项加载到寄存器中；
2. 将该寄存器中存储的值加1；

3. 保存寄存器的内容到刚才从中读取数据的堆栈偏移位置。

如果后续有一个对变量 i 的判决（例如，if i > 100 then...），编译器就知道 i 已经位于某个寄存器之中，因此在随后的比较和判决中它会重新使用该寄存器。有些变量，正如我们所提到的，在整个计算过程中都可以保留在寄存器中。这一切都取决于有多少寄存器可用、总共有多少变量以及对变量的访问频率如何。

实际上，程序员几乎无法控制哪些变量应存储在寄存器中，哪些要保存在一个堆栈中，尽管 C 编程语言中有 register 关键字可以要求编译器如果可能的话保存变量在寄存器中。例如，如果我们想保存 i 在寄存器中（如果可能），我们会如下声明 i：

```
register int i=0;
```

泄漏代码（spill code）得名于少数机器代码指令，执行这些指令时编译器会在程序中增加在存储器和寄存器之间存取变量的操作。由于内存访问远远慢于寄存器访问，因此泄漏的代码不仅会轻微增加程序的大小，而且对执行速度也有不利影响。长期以来，尽量减少泄漏代码一直是编译器研究人员和计算机体系结构设计者的目标。

3.4.5 数据处理

两个 8 位的整数相加，在一个 8 位处理器上通常是一件简单的事，在 32 位处理器上执行两个 8 位整数相加也相对比较简单[⊖]，因为两者的算术运算可以用一条单一的指令执行。 105

这种单一的指令通常很容易由硬件完成：从寄存器将两个操作数送给 ALU，然后将结果传送回另一个寄存器。

当用一个更小的处理器处理更大的数，并且是更复杂的处理时，情况会变得更有趣。让我们依次考虑三种可能性：操作数大于处理器的宽度、在一个定点 CPU 上处理浮点数以及复数的情况。

3.4.5.1 在小位宽 CPU 上处理大位宽数字

由于 C 编程语言可以定义 32 位甚至 64 位的数据类型，因此，任何服务于 8 位、16 位甚至 32 位 CPU 的 C 编译器必须能够支持比处理器的设计宽度更大的算术和逻辑运算。

首先请注意，许多处理器，具有指定的数据总线宽度，实际上却支持更高精度的算术运算。例如，大多数 ARM 处理器能够执行两个 32 位数的乘法。我们知道，这种操作的结果的最大规模可能是 64 位。ARM 中原始的乘法器可能只允许将结果的低 32 位存储到目标寄存器。然而，在新的 ARM 处理器上，一个“长乘法”指令允许将完整的 64 位结果存储到两个 32 位的目标寄存器中。由此可见，存储结果的操作将需要两倍长的时间来完成（但这比使用其他方法来确定高 32 位而言还是节省了很多时间）。

我们来看一下在一个没有“长”乘法指令的 ARM 处理器上，我们如何能够执行 64 位乘法（但请注意，这未必是完成这件事的最快方式）：

1. 加载操作数 1 的低 16 位到 R1。
2. 加载操作数 1 的高 16 位到 R2。
3. 加载操作数 2 的低 16 位到 R3。
4. 加载操作数 2 的高 16 位到 R4。
5. R0 = R1 × R3。
6. R0 = R0 + (R2 × R3) << 16。
7. R0 = R0 + (R1 × R4) << 16。

⊖ 注意将 8 位值放到 32 位寄存器中时需要先进行符号扩展（见 2.3.8 节），否则负数的补码会产生错误。

R0 = R0 + (R2 × R4) << 32。

在图3-13中进行了解释说明，其中数据载入被看做设置阶段，乘法和加法运算作为操作阶段。在操作阶段，需要进行四个乘法、一个移位和三个加法从而得到计算结果。

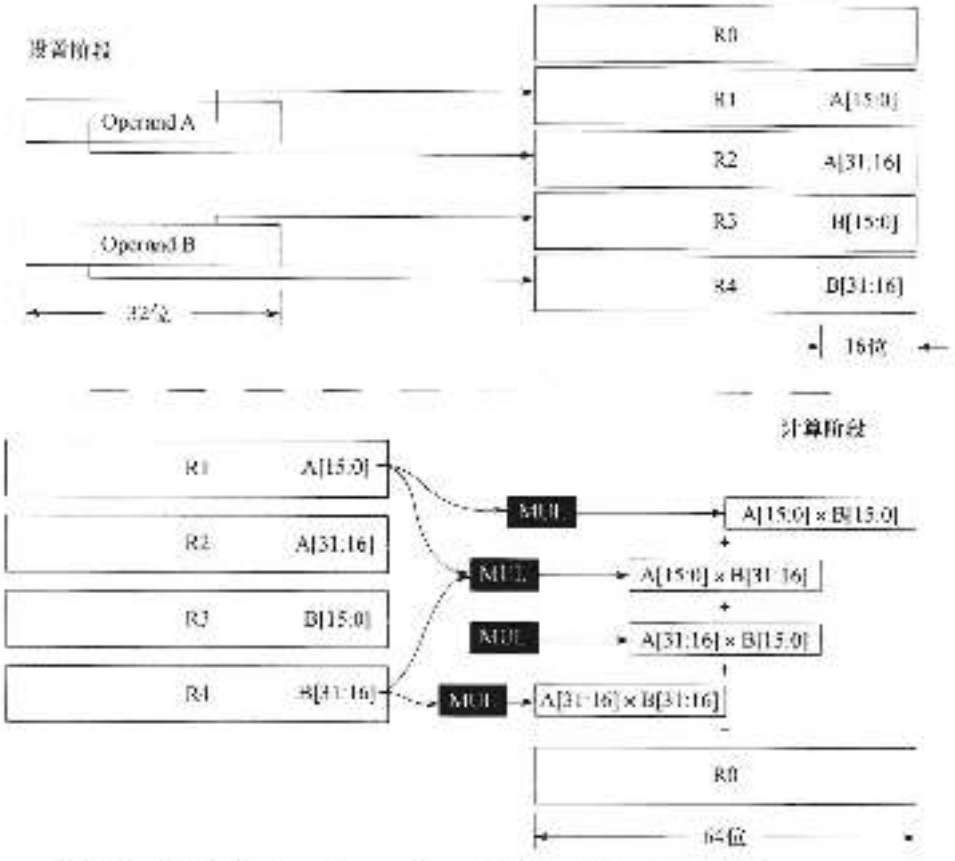


图3-13 本幻图阐明了执行一个32位×32位→64位乘法，其多步骤程序执行时所需的设置阶段和计算阶段，所使用的乘法器硬件只能够返回32位的结果（即16位×16位=32位的硬件）

这里明确的信息是，缺乏一个单独的“长”乘法指令意味着将采用一些额外的操作来代替，并可能用到额外的寄存器。当然，也有比这里所示的相快或开销较低的方法，在某些情况下也可
[1116] 以实现。然而，对于通用乘法没有什么比使用一个专有指令更好。

较长数据的逻辑运算很简单：拆分操作数，对拆分后的每一部分分别进行逻辑运算，然后再把结果重新拼在一起即可。这是因为在一元逻辑中每一位的逻辑运算结果并不影响其相邻位的运算结果。

算术运算则比逻辑运算需要多考虑一些事情（通比乘法或除法简单）。算术运算的问题是溢出：两个16位数字相加的结果可能是17位。因此，当对拆分的数字相加时必须考虑到额外的位
[1117] （即进位）。通常情况下，会先计算低字节部分，然后将进位加到高字节部分的计算结果中。

3.4.5.2 定点CPU上的浮点数

我们已经在2.8节讨论了浮点数，并在2.9节中讨论了浮点数处理。大部分情况下，当我们在计算机体系结构中讨论“浮点”时，我们指的是符合IEEE754标准的浮点数。事实上，大多数的硬件浮点单元也是按照IEEE754标准实现的。

没有浮点运算能力的处理器，要么依赖于编译器的支持，把每个C程序中的浮点运算转化为更慢的定点运算子程序，要么编译成浮点机器代码指令，然后由处理器通过捕获（trap）执

行。这种捕获效果上是一个中断触发，由收到一个处理器不能处理的指令而触发。然后由中断服务程序负责执行特定的浮点运算，直到返回正常执行位置，中断服务程序也是采用定点代码完成。这就是所谓的浮点仿真（FPE），将在4.6.1节进一步研究。第一种方法只适用于程序员知道在编译的时候FPU是否存在，所以可能不适合通用的软件，如在个人计算机上。

当一个系统中不包含硬件FPU时，用FPE替代（或编译器替代）可能不会实现完整的IEEE754标准，因为全部实现将使得速度相当缓慢。因此，代码最终将可能没有程序员预期的那样准确（也慢了很多）。

让我们回过头来看2.9.1节和2.9.2节，我们已经讨论了浮点数的加法/减法和乘法：加法/减法的过程中需要一个规格化的过程，而乘法过程中只需要直接计算，尽管包含几个子计算。如2.9.2节中的简单乘法：

$$(A \times B^C) \times (D \times B^E) = (A \times D) \times B^{(C+E)}$$

对于配有FPU的机器，$(A \times B^C)$ 和 $(D \times B^E)$ 将是单独的32位（单精度）或64位（双精度）值。它们将被分别载入两个FPU寄存器，一个单独的指令发出乘法执行命令，从目的FPU寄存器中将得到运算结果。相比之下，对于没有FPU的机器，将需要几个定点运算如下：

1. 拆分尾数 A 和指数 C，分别存储在R1和R2中。
2. 拆分尾数 D 和指数 E，分别存储在R3和R4中。
3. 计算新的尾数：R1 × R3。
4. 计算新的指数：R2 + R4。
5. 规格化指数。
6. 以IEEE754格式重组并存储。

显然，独立的FPU指令优于用几个定点数的运算操作来代替。 108

3.4.5.3 复数

复数，其形式为 $(a + \mathrm{j}.\, b)$，其中 $\mathrm{j} = \sqrt{-1}$，经常用于科学系统和无线通信系统。几乎所有CPU都不支持复数，而且也很少有编程语言对复数有所考虑[⊖]。

一个硬件系统中的复数计算实际上只是处理实数，就像浮点数计算是使用定点运算来完成一样，会需要几个步骤。考虑两个复数的乘法和加法运算：

$$(a + \mathrm{j}.\, b) \times (c + \mathrm{j}.\, d) = (a.\, c - d.\, b) + \mathrm{j}(a.\, d + b.\, c)$$

$$(a + \mathrm{j}.\, b) + (c + \mathrm{j}.\, d) = (a + c) + \mathrm{j}(b + d)$$

复数乘法需要4个实数乘法和两个加法。复数的加法稍微简单一些，只需要两个实数相加。这要求程序员（或编译器）能够将操作拆分成几个简单的指令步骤。

对复数运算有硬件支持的处理器，应具备单独的指令才能够执行这些操作。而底层的硬件体系结构实际上需要执行所有的拆分、子操作及分别进行乘法，但是这些在CPU内部处理会非常迅速，无需分别载入数据、存储和移动数据。

3.5 自顶向下方法

3.5.1 计算机的能力

纵观今天各种不同的处理器，它们都有各自不同的功能、时钟频率、位宽以及指令集等。我

⊖ 注意有一个例外是FORTRAN语言（FORmula TRANslation），它是IBM在20世纪50年代中期推出的通用编译语言。FORTAN语言已经更新了几次（最新一次是在2003年），从50多年以前开始，该语言本身就支持复数数据类型。在现代编程语言中，Java作为一门科学语言已经在复数扩展上有一些推广，但不幸的是，现在Java明显比FORTRAN的执行速度慢很多。

们的问题是究竟我们需要怎样的一台计算机？以下将对计算机的能力进行探讨。

3.5.1.1 功能

假如所有的计算功能都可以通过一串逻辑操作完成，那么为什么不是所有的功能都采用这种方式（即可能的一长串逻辑操作）来实现呢？主要原因是其与效率有关——该功能需要多长时间完成和需要什么样的硬件来完成？将计算机变得简单从而可以使用更快的时钟是需要一些折中的，这导致了RISC的出现，它相对简单且时钟更快——其代价是必须用多个步骤来执行某
109 些功能，而CISC则是将多个功能通过一条指令来实现。

而在确定如何实现一个特定功能之前考虑一下该功能在软件中的使用频率是有实际意义的。简单来说，如果一个功能在日常使用时经常被用到，则设置一个专有硬件来快速地处理它也许是很有用的。如此，在所有现代的处理器里都含有ALU，而且几乎所有处理器也都含有乘法单元。

不仅是CPU的功能，CPU指令集的灵活性也是CPU的一个重要特征。例如，尽管省时指令（time-saving instruction）使用简单的硬件就可以实现，但并不是所有的设计都支持。同样的例子还有比如在ARM指令集中普遍存在的条件指令（见3.3.1节）和在一些数字信号处理器中的零开销循环指令（zero-overhead loop instruction，稍后在5.6.1节介绍）。

CPU的内部结构如总线数量、寄存器数量以及它们的组织，也是CPU性能的一个重要考虑因素。总体而言，更多的总线意味着有更多的数据可以同时传输，由此可以获得更好的性能。类似地，更多的寄存器可以存储更多的软件变量，从而不需要频繁访问较慢的内存，由此又可以改善CPU的性能。

3.5.1.2 时钟频率

更高的时钟频率并不总是意味着更快的操作。例如，设计一个快速ALU相对简单，而设计一个快速的乘法单元则难得多。当比较两款处理器时，单考虑时钟频率是不能断定哪款处理器更快的。还需要考虑如功能、总线带宽、内存速度等因素，实际上，应该问：在一个时钟周期内可以完成什么？这个问题将在下一节讨论。

3.5.1.3 位宽

直到最近，用于手表、计算器等中的绝大部分CPU都是4位处理器，而各式用于手机和网络应用的32位处理器（一般都是基于ARM的）则开始将重心移向更高位宽的处理器。

虽然看上去更高位宽的处理器会获得更快的执行速度，但这仅是对于所要处理的数据能够使用这么多位的情况而言的。高端的服务器使用了64位甚至128位的体系结构，但是如果用于处理文本（如7位或8位ASCII或16位Unicode），那么更多的位宽将会被浪费。

3.5.1.4 内存

与处理器相连的内存也是决定处理速度的关键因素，不仅仅是内存的访问速度，位宽（也指带宽，位/秒）和技术也同样重要。其他因素还包括突发访问模式、分页或分包、单沿或双沿时钟。

110 片上存储并不都是单周期访问的，而是比片外存储稍快一些。给定某一软件任务，则需要考虑需要提供多少片上存储和多少片外存储。cache（4.4节）就是用于最大化高速存储的使用，而内存单元的硬件复杂度通常也影响着内存使用的优化。就内存而言，软件的写法和编译方法通常也影响着硬件资源的利用效率。

3.5.2 性能衡量和统计

为了确定计算机的执行速度，最简单而通用的方法是测试其在每秒能处理多少条指令。

MIPS（每秒百万条指令）是用于衡量指令和操作处理的速度。这是一个有用的低层次衡量，

但它并不真正反映计算机能力：有些操作本身很简单，因此多个操作可以用来实现一个有用的任务。换句话说，一个简单的拥有高 MIPS 的计算机（如 RISC 处理器）在处理实际的任务时也许会比一个拥有低 MIPS 但其每条指令可以完成多个工作的计算机（如 CISC 处理器）要慢。bogomips 是在 Linux PC 启动时计算出来的，它是一个著名的测量软件 MIPS 的尝试，但它并不是太精确。

MIPS 由时钟频率 f（单位 Hz）和 CPI（每条指令周期数）决定：

$$\text{MIPS} = f/\text{CPI}$$

更一般地，对于某一个包含 P 条指令的程序，其完成时间为：

$$T_{\text{complete}} = (P \times \text{CPI})/f$$

所以 CPI 越低、f 越高或 P 越少（即指令越少的程序执行时间可能会越短）则完成时间 T_{complete} 越短。在现代 CPU 的时钟频率都已经提高时，计算机体系结构里的 P 和 CPI 之间的折中又让话题重新回到 RISC 和 CSIC 之间的竞争上。

降低 CPI 是当代计算系统设计的一个方面。20 世纪 80 年代，CPI 大于 2，在一些 CISC 处理器中可以达到几百。而 RISC 方法则降低了 CPI，它的目标是使 CPI 接近于 1。ARM 系列处理器典型的 CPI 大约为 1.1，而其他一些处理器会比这个值更低一些。

之后，超标量体系结构的出现使得 CPI 低于 1，这是通过允许多条指令同时执行而获得的。CPI 和其倒数（IPC）将在 5.5.1 节中进行探讨。

有些时候浮点性能也是一个重要的因素，其单位为 MFLOPS（每秒百万次浮点运算）。最近，GFLOPS 更经常被使用，它指千 MFLOPS，甚至还有 petaFLOPS（PFLOPS）。这些值比 MIPS 更能真实地反映实际性能，这是因为我们需要统计更有用的计算操作数而不是低层次的指令数。 111

标准检测程序（benchmark）是很重要的，多家公司都已经提供这种产品（框 3.8 探讨了标准检测程序的背景及其必要性）。BDTi 是标准检测程序的一个例子，它用于比较几个数字信号处理器（DSP）的速度。其测量倾向于整体的计算性能，而计算性能是 DSP 市场的支柱。

而 SPECint 和 SPECfp 标准检测程序直接计算整数和浮点性能，可以通过向标准性能评定组织（Standard Performance Evaluation Corporation，SPEC）付费获得其源代码格式，并可以在某一个体系结构上编译从而获得其性能评估。每个测量都是通过计算一系列算法并将结果混合而获得的。一般而言，年份代表了 SPEC 的版本，因此 SPECint92 就是 SPEC 整数标准的 1992 年版本。

SPEC 测试本身包含了两个测试：Dhrystone 和 Whestone，它们都起源于 20 世纪 70 年代，分别用于衡量整数性能和浮点性能。还有其他一些影响性能的因素分别用于衡量不同任务的性能（如图形渲染、实时性能、字节处理等）。

框 3.8 基准性能

在 20 世纪 80 年代中期，全世界计算机产业发现制造商之间一个意想不到层次的竞争。这并不是简单的 AMD 与 Intel 之间的赛跑，而是上千厂家所销售的各种各样的计算机——不同的体系结构、不同的内存、几十种 CPU 类型、定制的操作系统、8 位、16 位甚至一些非常规选择。

在英国，如 Sinclair、Acorn、Oric、Amstrad、Research Machines、Apricot、Dragon、ICL、Ferranti、Tandy、Triumf-Adler 等公司在市场上开始与 IBM、Apple、Compaq、DEC、Atari、Commdore 等公司对抗。各种与性能相关的争论在广告和宣传小册上随处可见。然而，由于没有标准和基准，这些争论通常都毫无意义。

为此，英国标准协会（British Standards Institute，BSI）为计算机出台了一个性能标准，它测试一些有用的任务如整数计算、浮点计算、跳转性能和图形性能，以及磁盘读写等。然而，由于当时可选的编程语言为 BASIC，所以这个测试软件是用 BASIC 写的。从今天的观点来看，图形和磁盘读写测试已经过时："图形"测试是测试将文本输出到屏幕上或虚拟显示单元（VDU）的时间。这对许多只对字处理感兴趣的

用户来说很重要。而磁盘读写是指软盘（比磁带快得多，当时在一般家用机器上都有配备），硬盘（当时称为温切斯特驱动器）十分昂贵，当时一些主流的计算机都没有配备。而那时程序更多的是存储在磁带上。

今天，计算机杂志和网站用于新硬件和软件的测试都带有电池测试，它与BSI相差很远，但初衷都是一样的。因此，“玩QuakeⅢ时的刷新率”和“对100万行随机数进行排序”等测试都相继出现。也有其他一些标准（一般都不是免费获得的），但极少采用：毕竟，大部分用户对玩Quake比对多快能计算到π小数点后100位更感兴趣。

然而，众所周知，为了测试某一项性能，计算机设计人员会通过牺牲其他方面的性能来让其设计在这一项性能上获得较高的分值。而这种测试并不能真正反映任何任务的整体完成时间，而只是某最简单任务单独执行的时间。所以像中断任务，操作系统调用，不同的内存速度、磁盘速度，多任务执行以及cache都会影响测试结果。

在计算机里，cache（4.4节有详细讨论）是提供给系统的一块高速存储，而系统上有一块更慢的主存。任何程序在cache上执行都比在主存上执行明显要快得多。这有什么关系呢，在过去有的处理器制造商通过增加cache大小以装下整个性能测试算法（如整个SPECint或Dhrystone程序），如此获得了比其他竞争者的机器更快的执行速度。

在这个例子中，如果主存的速度比cache慢10倍，性能的测试结果并不会因此而改变，因为整个测试程序都是在cache上执行而不是在主存上。显然，这样的一个性能测试并不能反映真实情况。事实上，这样的机器会比那些拥有更小cache和更大主存的机器获得更快的执行速度——在执行实际任务时会快些。

在给定一些如我们之前所提到的重要可变性能因素的情况下，显然当前的性能测试将面临一些困难，系统设计人员会由此变得小心翼翼。在实际中，这可能意味着需要理解设备运行的细节，构建大量的安全机制，或在交付设备之前现场测试最终代码。虽然在实际的工业生产里极少会出现软件设计完成而硬件并没有完成的情况，但如果这种情况出现，那么现场测试还是很值得推荐的。

3.5.3 性能评估

6.4.4节将对实时系统和多任务系统的完成时间与执行性能进行讨论，而这里我们将讨论如何进行性能评估。为了强调精确性能评估的重要性，这里举一个工业上的例子：

112
~
113

几年以前，一个嵌入式设计小组需要一块处理能力为12MIPS的硬件运行一个算法。他们选中了一个以40MHz频率工作、提供40MIPS处理能力的32位处理器。为了降低设计风险，设计人员在把该处理器确定为最终选择前，拿到了一块开发板，在其上装载了Dhrystone测试程序并试图测试其真实性能。

在设计的过程中，他们发现片上存储不能满足他们的软件的需要并由此增加了额外的DRAM。由于CPU本身封装很小并且管脚数很少，因此将外部存储总线限制为16位宽。外部存储的访问由此从32位宽变为16位宽。

在完成硬件设计并搭建好系统后他们装上了程序，却发现在执行时间上并不满足需要，他们走错哪一步了呢？

首先，Dhrystone测试程序能完全存放在快速的片上存储里，并由此可以全速执行。而他们的程序太大不能完全装在片上存储里，因此需要存放在DRAM上。片外DRAM不仅访问速度比片上存储慢，而且还需要间歇性“超时”来更新自己。在“超时”期间，来自CPU的所有访问都要停顿。

其次，16位的接口使得对每条32位的指令需要两次访存来取指，而每个32位的数据也同样需要两次访存。这就意味着，每当程序是从DRAM上执行时，CPU有一半时间是空闲的。在每个奇数周期取出指令的前半部分，在每个偶数周期取出指令的后半部分，然后才能处理它。

16位接口导致执行速度从40MIPS降为20MIPS，而DRAM的慢速访问和更新时间将20MIPS性能进一

步降低至9MIPS左右。

解决这个问题的方案也并不令人满意：更换为速度具有更快但费用为原来20倍的片外存储（SRAM），或是将CPU升级为具有更快的处理速度或拥有更宽的外存访问接口，或两者皆具。但设计人员并没有采用任何一种方法，而是在其旁边增加了一块CPU来处理部分任务。

这个例子强调了性能需求与硬件相匹配的必要性。总体而言，目前有两种方法来实现性能测试。第一种是通过对体系结构清晰地理解，第二种是通过细致地评估体系结构。对于这两种方法，体系结构不仅仅指处理器，还包括其他重要的外设。

清晰地理解软件需求意味着需要确定系统上运行的软件，分析软件的具体要求（特别是瓶颈），然后根据分析的结果找到匹配的硬件。在最基本的层次上，这意味着当大部分计算为浮点计算时需要避免使用仅支持整数的CPU。

这种方法通常为DSP系统设计所采用，其评估内容包括内存传输，使得对变量能够同时进行访问的不同的内存区域布局（4.1.4节），输入和输出瓶颈，以及对于这类处理器最重要的算术运算等。除了整体程序存储容量需求之外，慢速机构、用户接口以及控制代码在这种计算中都会被忽略。 114

在此有必要指出，大部分的软件开发完之后所需的初始程序内存空间比估计的要大。良好的编码能够降低数据内存的使用，减少处理时对内存的需求，但无法节省太多程序占用内存的空间。与台式计算机设计人员不同，嵌入式设计人员不能提供RAM扩展：这往往是在设计时就已经定好的。由此，给内存足够的冗余是明智之举。

上面提到的第二种使所需性能与硬件匹配的方法是细致地评估。这种方法不需要了解体系结构的细节，但是需要了解每个层次测试的细节。理想状况下，最终运行的软件需要在候选硬件上执行以便对它需要消耗多少CPU时间进行评估。对其他的任务也需要进行测试，以确定这些任务在硬件上执行是否有处理时间冗余。软件分析工具（如GNU gprof）将指出运行代码的瓶颈，找出软件的哪些部分需要耗费大量CPU时间。

多次运行每一个测试很有必要（但如果对时序要求严格时不能取结果的平均值，则应采用最坏的结果），测试时可以大幅度增加软件大小以使其超出cache或片上存储的承载能力，如果可以还应允许最终系统上的任何中断和辅助任务。

如果目标软件在其他机器上已经运行过（大部分情况都是如此），可以将其在不同机器上的执行进行比较——但必须同时考虑在前面两章里所讨论的所有重要的体系结构因素。在这种情况下，在两台机器上编译和比较标准检测程序会有所帮助（假设所选的标准检测程序与目标软件相关）。

当前全世界范围有很多设计人员错误地评估处理器性能和（或）内存需求的例子（其中包括1999年作者给亚洲某制造商所做的设计：设计一个便携式MP3播放器，由于没有预料到它的内存总线带宽很低，因此它每次只能重放7秒钟的MP3音频。还好它后来采用了更快速的处理器）。

请注意，要时刻警惕性能测试评估的陷阱。尤其重要的是，要记得阅读手册中性能声明下的附属条件。

3.6 小结

本章的讨论覆盖了微处理器基础内容，从CPU的功能，到CPU对程序的控制，最后到程序的传输和存储。 115

控制单元需要保证处理器按规则执行，管理操作和异常，并能在计算机程序的引导下接受一系列指令。控制单元可以是集中式的，或根据状态机的时序为分布式，或微指令引擎，或采用

自定时逻辑。

程序中每条指令都是所允许指令集中的一部分（取决于你的观点），它描述了处理器能执行的操作，或指定处理器的行为，这些行为包括通过内部总线将数据传输到不同的功能单元。通过上一章和本章对CPU设计基础的铺垫，我们将在第4章对目前主流CPU的内部布局和功能单元
116 进行深入探讨，并尝试把编程人员的经历融合于其中。

思考题

3.1 如果汇编指令LSL表示“逻辑左移”，LSR表示“逻辑右移”，ASL表示“算术左移”，ASR表示“算术右移”，那么对以下的有符号16位数字操作后的结果是什么？

a. `0x00CA ASR 1`
b. `0x0101 LSR 12`
c. `0xFF0F LSL 2`
d. `0xFF0F LSR 2`
e. `0xFF0F ASR 3`
f. `0xFF0F ASL 3`

3.2 对一个RISC处理器的典型代码（只包含8种指令）进行分析后得到以下这些指令的出现次数统计：

指令	出现次数	指令	出现次数
ADD	30	NOT	15
AND	22	ORR	10
LDR	68	STR	60
MOV	100	SUB	6

a. 如果每条指令（不包括操作数）为6位长，那么这个程序将占据多少位内存空间？

b. 使用以上信息对这些指令设计霍夫曼编码。

计算对指令集采用霍夫曼编码后的程序需要占据多少内存空间。

3.3 给出执行以下逆波兰表示法（RPN）操作时堆栈的PUSH和POP队列，并将每条RPN转换成中缀表达式：

a. $ab+$

b. $ab+c\times$

c. $ab\times cd\sin+-$

117 考虑执行以上操作所需的最大堆栈容量。

3.4 ROT（rotate）指令与移位指令相似，只是它要回绕——当向右移位时，从字的LSB端出来的每一位成为新的MSB；当向左移位时，从该字的MSB出来的每一位成为新的LSB。

ROT的参数为正时进行左移，为负时进行右移。

假如一台计算机只有ROT指令而没有移位指令，如何进行算术运算和逻辑位移操作？

3.5 将以下中缀表达式转换为RPN：

a. （A and B）or C

b. （A and B）or（C and D）

c. （（A or B）and C）+D

d. C+{pow（A，B）×D}

e. 能否使用3种方法对下式进行转换：{C+pow（A，B）}×D

3.6 分别计算上题e中三种方法所需的堆栈深度。

3.7 将下列RPN转换为中缀表达式：

a. AB+C+D×

b. ABCDE+××-

c. DC not and BA ++

3.8 使用条件指令 ADDS 重写下列 ARM 汇编代码，移除所有跳转指令。

```
          ADDS R0, R1, R3
          BGE step2
          ADD R2, R1, R6
          BLT step3
step2     ADD R2, R3, R6
step3     NOP
```

3.9 使用 ARM 汇编语言时，确定进行以下立即数加载时所需的最少指令数（提示：使用 MOV 指令）：

a. 加载 0x12340001 至寄存器 R0

b. 加载 0x00000700 至寄存器 R1

c. 加载 0xFFFF0FF0 至寄存器 R2

118

3.10 写出在一个 RISC 处理器上将两个内存地址 m1 和 m2 上的内容进行相加后再将结果存到地址 m3 上的操作序列。

3.11 科学家发现了新的内存单元硅片，半导体工程师将这种硅片设计成新的存储芯片。指出计算机体系结构设计人员需要考虑的 6 个因素，以决定是否将这种新技术应用在嵌入式视频播放器的主存储器上。

3.12 指出下列指令是来自 RISC 还是 CISC 处理器。

a. MPX：将内存两个地址上的内容相乘，然后将结果加到累加器上。

b. BCDD：将两个寄存器里的内容进行 BCD 码的相除，结果使用科学记数法表示，并以 ASCII 的形式存放在内存块中，为在显示器上显示做好准备。

c. SUB：对两个操作数进行相减，结果作为第三个操作数。操作数和结果都为寄存器的内容。

d. LDIV Rc，Ra，Rb：执行 100 个时钟周期的长除法：Ra/Rb，将结果放在寄存器 Rc 中。

3.13 写一个微指令程序来实现上题中任意的两个指令。假设是在一个 RISC 体系结构上执行。

3.14 什么是加载存储（load-store）体系结构？为什么计算机设计者会采用这种方法？

3.15 在一个简单的流水线上，指令取指阶段之后跟着什么处理？

3.16 假设有一个 32 位处理器，其指令以十六进制机器码的形式存储。在内存位置 0x9876 上存储字 0x1234，其指令如下：

```
0x0F00 1234 088D 9876
```

观察该指令机器码，确定这个处理器是否支持绝对寻址。解释你的答案。

3.17 假设另外有一个 8 位处理器，拥有 8 个寄存器。问这个处理器的指令是否能够支持两个寄存器操作数和一个结果寄存器操作数？

119

3.18 假设以下指令为 ARM 汇编语言（对 ARM 处理器不是必需的），指出以下指令的寻址方式：

a. `MOV R8, #0x128`

b. `AND`

c. `STR R12, [R1]`

d. `AND R4, R5, R4`

e. `LDR R6, [R3, R0, LSL #2]`

f. `LDR R2, [R1, R0, #8]`

g. `STR R6, [R3, R0]`

3.19 哪种处理器处理 32 位浮点数据会更快：900MHz 的 32 位浮点 CPU 或 2GHz 的 16 位整数 CPU？

3.20 对不同处理器写 C 程序时，byte 是否总是表示为 8 位？short 和 int 类型呢？它们的大小为多少？是否总是相同？

120

第4章

Computer Architecture: An Embedded Approach

处理器内部组成

在第2章中，我们已经讲述了由计算机提供的大部分数值计算，也对计算机的功能单元给出了定义并对计算机一些内部连接结构给出了分类。在第3章中，前面提及的这些机构又组合成一个有着不同功能的完整结构，用来执行由程序员编写的指令序列。因此，我们知道计算机，特别是CPU，可以在逻辑上分为多个用来执行不同任务的功能单元。

这一章，我们进一步给出CPU内部组成的高级描述，并且集中讨论当今处理器中常见的最大、最显著、最重要的内部单元。我们将更细致地查看这些单元执行什么样的任务以及它们怎样具体执行这些任务。这些讨论主要覆盖算术逻辑运算单元（ALU）、浮点运算单元（FPU）、内存管理单元（MMU）以及高速缓冲存储系统单元。在开始着手讨论上述问题之前，我们将首先考虑这些单元是如何通过总线连接起来的。

是时候去检视真正的体系结构，特别是CPU内各个单元的内部与联总线结构了。

4.1 内部总线结构

4.1.1 程序员的角度

从程序员的角度出发，一个处理器的内部总线结构可以从两个相互联系的方面来看。第一，寄存器使用的灵活性。显然在可用的寄存器集合中，这些寄存器可以在一条特定指令中直接作为操作数。以ARM为例，寄存器操作数是允许的，任何寄存器组中的寄存器可以进行如下操作：

```
ADD R0, R1, R2  ;R0 = R1 + R2
```

可以使用任何寄存器，甚至可以使用相同的寄存器：

```
ADD R0, R0, R0  ;R0 = R0 + R0
```

很多处理器没有这样的灵活性或者灵活性有差。第二，在一条指令周期中能够做多少工作
[121] 也是程序员所关注的，这个问题一般都蕴含在指令集自身当中。再以ARM为例，对于其任何算术或者逻辑指令，必然有最多两个输入操作数寄存器和一个结果操作数寄存器：

```
ADD R0, R1, R2  ;R0 = R1 + R2
```

关于寄存器和ALU之间的数据传输方法：如果传入和传出都在一个周期内完成，则意味着输入和输出都有着各自的总线（因为在任何时刻只能有一操作数在一条总线上传输）。一条总线传输R1的内容，另一条则传输R2的内容。还有一条用来将结果从ALU写回到寄存器R0。

从以上内容中会发现，每个寄存器都与一条属于自己的总线相连，因此就至少需要二条内部总线。

关于寄存器和ALU的排列我们可以从刚刚如图4-1所示的指令集的简单分析中归纳出来。这实际上是对ARM处理器内部总线结构的简单原理阐述。其中箭头指示着可控的三态缓冲器，三态缓冲器作为门来控制寄存器和总线之间的读写操作。此处没有给出控制逻辑（在3.2.4节中已经讲过）。

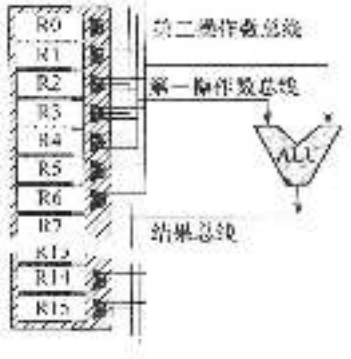


图4-1 一个ALU与一个寄存器组成的一个三路总线结构

4.1.2 分解互联指令

ARM之所以如此著名，是因为它的规整性和简洁性。其他的一些处理器对底层程序员来说不是特别友好，而ARM由于给出16个完全相同的寄存器以完全相同的方式连接而成的寄存器组，导致此它往往可以赋予寄存器集合以特殊的用途。对这些寄存器集合最常见的指令就是分派几个寄存器成为专门的存储和处理地址的地址寄存器，而其他的为数据寄存器。设计一个只用来连接这些地址寄存器的内部地址总线，是一种容易想到的分解方法。在ARM中，每个寄存器都可以存放地址（因为可以直接寻址，详见3.3.4节），每个寄存器也必须和内部地址总线相连。 122

在一些处理器中，比如ADSP21xx，并没有寄存器组，取而代之的是每一个处理单元都有附带的输入输出专用寄存器。这意味着当在这些处理器上运行一条特殊指令时，底层程序员必须记住（或者查询程序手册）哪个寄存器是允许使用的。有时候为了执行某个功能，不得不浪费一条指令来将一个寄存器中的值传到另外一个寄存器。尽管聪明的指令集设计可以尽量减少这些无效率的操作。目前，通用处理器已经很少使用这类体系结构，但是依旧可以在数字信号处理器中找到类似的结构，比如ADSP21xx[二]系列处理器。

为什么设计者要自找麻烦设计如此复杂的指令集呢？这个问题的答案需要我们对处理器所要完成的一些功能做个快速扫视。在这个例子中，我们可以观察到，ARM是使用如图4-2所示的硬件来完成以下功能的：

```
SUB R0, R1, R2  ;R0 = R1 - R2
ADD R4, R5, R6  ;R4 = R5 + R6
```

图4-2显示了数据从R1和R2同时输出到两条操作数总线上（用深色线标出），然后进入乘累加单元（MAC），把结果经过结果总线送回R0。

在这张图中值得注意的是，R3以前者寄存器以及ALU都处于闲置状态。当CPU设计者看到有资源闲置时，他们会尝试着尽可能将这些闲置资源利用起来。比如，可否有一种方法能够将ALU和MAC同时利用起来。这个问题的答案是如图4-3所示的划分设计。

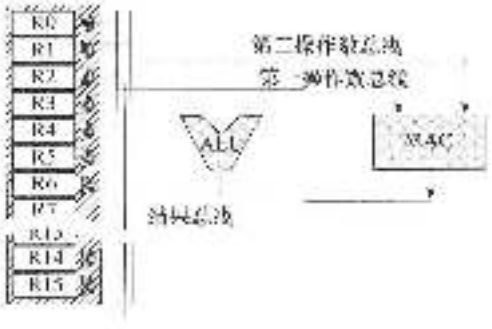


图4-2 一个ALU、一个MAC以及一组寄存器以三路总线相互连接的非列原理示意图。本身看上去该结构能同时传送两个操作数到一个功能单元的能力

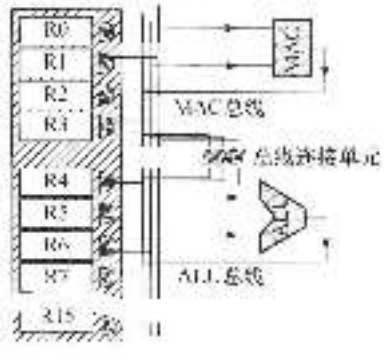


图4-3 一个ALU、一个MAC以及一组寄存器以三路总线相互连接的非列原理示意图。这与图4-2中的资源使用方式类似，不过这里由于使用了总线划分，从而允许两个功能单元同时向（从）总线传送数据 123

㊀ 实际上，寄存器R14和R15分别是指定为产生间址寄存器（地址生成器），单从指令角度分析很难看出这个特征。另外，寄存器功能还会因它们的影子（shadowing）实体而有所变化。

㊁ "xx"是指在ADSP21系列产品中不同的产品系列号，每一系列产品都与同样的特点，如ADSP2181、ADSP2191等。

如图中排列所示，MAC 和 ALU 都有它们各自的输入和结果总线，通过扩展，它们也有各自优先使用的寄存器集合。这样，只要程序员记住了在调用 MAC 时使用 R0～R3，以及在调用 ALU 时使用 R4～R7 就可以了，这样示例指令：

```
ADD R0, R1, R2 ; R0 = R1 + R2
ADD R4, R5, R6 ; R4 = R5 + R6
```

就可以在一个周期内完成。

这个过程也许是受 ADSP21xx 的设计影响，使得设计者尽可能地去追逐处理器的性能提升。

4.1.3 ADSP21xx 总线排列

在 ADSP21xx 的硬件中，每个处理单元都是有限制的，它们只能从一小部分寄存器中读入数据，也只能将结果写回到一小部分寄存器中。这意味着，在这个处理器中有很多内部总线和操作可以以并行的方式快速执行。

[124] 图 4-4 简单地描述了 ADSP21xx 众多内部总线的一部分。在该图中，PMA 代表了程序存储器的地址，DMA 是数据存储器的地址。这两个地址总线都与程序存储器和数据存储器相连，这也说明该处理器采取了哈佛体系结构（见 2.1.2 节）。然而，从其对地址空间的划分来讲，它比基本的哈佛体系结构又更深入了一步。PMD 和 DMD 分别代表了程序和数据内存总线。注意总线的大小，ADSP 不仅在内部总线互联上比较复杂，而且其总线宽度也各不相同。

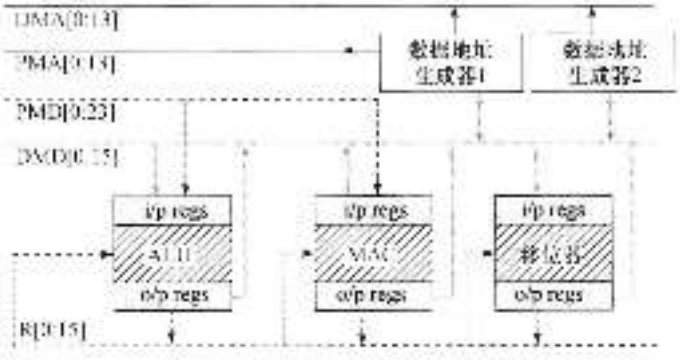


图 4-4 ADSP 内部总线排列示意图

该图显示了 ALU 和 MAC（不包括移位器）可以从 PMD 总线上接受 24 位的输入，还可以从 16 位的 DMD 上接受输入，同时也可以对其进行输出。

4.1.4 数据与程序同时访存

在诸如数字信号处理（DSP）领域中，有一个重要问题就是外部数据可以以多快的速度进入处理器，然后得到处理并最终输出。数字信号处理器一般都是对一些流数据进行操作，比如高保真的音频信号或者无线宽带信号。

数字信号处理操作往往是某些形式的数字滤波。可以考虑一段时间内的信号采样，x[0]，x[1]，x[2] 等，分别表示时刻 0（当前时刻）的采样、前一个时刻的采样、前两个时刻的采样，依次类推。y[0]，y[1]，y[2] 是根据这些时刻采样而输出的结果值。如果这些信号为音频数据，那么 x 和 y 应该是音频信号采样，可以是 16 位，如果我们以 48kHz 的频率进行采样，则上面所说的时间间隔为 1/48 000 = 21μs。

我们不去深究数字信号处理的原理，在这里仅简单介绍两个通用的滤波公式：有限冲激响应（FIR）滤波和无限冲激响应（IIR）滤波。FIR 的输出是将前 n 个采样与一些预设好的值相乘

后全部累加到一起所得到的。可以写成如下数学形式：

$$y[0] = \sum_{i=0}^{n-1} a[i] \times x[i]$$

所以，当前的输出 y[0] 是由前 n 个输入数据分别乘以滤波系数 a[] 然后全部累加在一起所得到书。输入数据的个数决定了滤波器的阶数。一个 10 阶的滤波器应该预先定义 n = 10 并且提前设定好 10 个滤波系数 a[]。一个自适应的 FIR 滤波器可以让滤波系数随着时间的变化而变化。

相反，IIR 滤波器的输出结果依赖于之前所有的输入和输出，其数学形式可以写成：

$$y[0] = \sum_{i=0}^{n-1} a[i] \times x[i] + \sum_{i=1}^{n-1} b[i] \times y[i]$$

这个公式包含了一个更加高级的滤波系数 b[]。IIR 滤波器同样是自适应的，并且一般可以完成和 FIR 滤波器一样的工作，但是在阶数较低的情况下（n 值较小）。这种强大的滤波计算需要付出一定的代价，可以看到如果设计得不够仔细，IIR 滤波器也有可能变得不稳定。 125

设计一个高性能数字信号处理器的关键在于尽快地完成这些公式。也就是在尽可能少的时钟周期内计算 y[0] 值。回过头来看看 FIR 公式，我们可以看出这种类型的计算基本上都是在重复以下的底层操作：

```
ACC:= ACC + (a[i] x x[i])
```

将两个值相乘并加上某个已存在的值称为乘累加（multiply-accumulate），它需要使用一个累加器，通常简写为 ACC。现在我们需要将这个功能与数字信号处理器的硬件联系起来。这里有很多巧妙的设计可以讨论，但最重要的一点是要提好存储器访问。

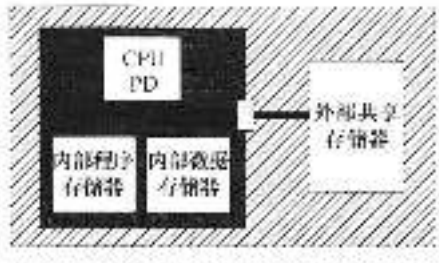


图 4-5 以 DSP 为例的哈佛体系结构框图，其中拥有内部程序存储器和内部数据存储器，也支持外部共享存储器

从图 4-5 中我们可以看到一个数字信号处理器包含了一个 CPU、两个存储模块和一个外部共享存储器。这个设备看上去也是采用了哈佛体系结构（程序与数据分开存储、独立总线），但同时又与一个外部共享存储器相连。这种类型的存储器安排是很常见的，一般内部存储都是 SRAM，而外部存储器都选用更廉价的 SDRAM（参见 7.6 节对存储技术及其特点的详细介绍）。

片上存储器一般都采用较短的总线并且速度非常快，有时可以在一个时钟周期内取到指令。有些情况下，也采用两周期的存储模块，而之所以该存储器要使用两个周期，是因为其请求数据占用一个周期，将数据有效化占用一个周期。

我们现在暂时忽略存储器的速度，再看一下乘累加这个例子。我们需要给乘法器输入两个值：一个预先设定好的滤波系数 a[] 和输入数据 x[]。在一个共享总线下，不可能同时获取或者传送这两个数据。然而，以图中的内部分离式总线设计，如果这两个数据分别来自两个独立的片上存储器，则它们可以同时获取并可以在一个周期内完成乘法操作。总体来看，这将是一个多周期操作：一个周期载入指令并进行译码，接下来的周期载入操作数，之后的一个或多个周期对这些操作数完成相应的操作。然而，考虑到快速单周期片上存储器的存在，取操作数有可能成为内部指令周期的一部分。 126

通常，任何在片外总线上的传送速度要比片上的慢，这也是使用 cache 存储的一个主要原因（详见 4.4 节）。在拥有 SDRAM 外存的地方通常都会有片上 cache 来缓解 SDRAM 带来的问题，这个问题就是不论 SDRAM 有多快，总会在对存储器进行数据访问和数据准备时有 2～3 个周期的延迟。

4.1.5 双总线体系结构

退一步说，大量硬件的节省是压缩总线数量的结果——在集成电路中，总线是由一束并行的硬连线组成，不同的缓存、寄存器或者互联结构配置，有着不同的总线宽个，对于珍贵的硅面积来讲，这些实际占用芯片面积的总线无疑会影响其总成本。因此，通过采用双总线结构来降低芯片面积（以及成本）是完全有可能的。采用单总线结构效果会更好（见4.1.6节）。

在这里我们采用实例并没有与计算机体系结构的发展传统相对比。之所以这样做是因为一个三总线的结构比一个单总线的结构更容易让人接受，也更易于解释。设计单总线结构是需要技巧的，这种应用于硅片上的技巧曾在20世纪80年代前后现过，但是这却无法复杂化一个简单的观点，那就是总线就是一条位于源操作数和目的结果值之间的路径。本章中所有的例子都是虚构的，它们呈现出一些类似于ARM体系结构的东西，但在总线安排上却又不一样。原始的简化总线的设计，例如值得尊敬的6502处理器，只有很少的寄存器组和一个乘法器。这里隐含着一个问题，那就是硅片面积过小将限制着某一个很好的体系结构设计，有时候甚至是只有较高时效性体系结构的设计。在仅仅有三个通用寄存器的限制下，6502的设计者从来没想过去设计为外一条并行的总线，而是在有限的硬件资源下增加了一些寄存器。

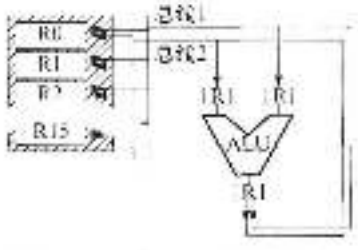


图4-6 连接ALU和寄存器组的双路总线

图4-6显示了一个寄存器组与ALU相连的双总线排列，在ALU周围有二个寄存器或锁存器（不考虑较大的寄存器组，这跟6502非常相似）。

为了实现这一设计，并且让接下来的例子更有意义，我们有
[127] 必要回顾一下关于ALU的一些知识，这就是传输延迟时间。当我们给ALU的两个输入端送入稳定的电信号时，需要等待一定的时间来获取最终有效的计算结果。一些控制逻辑（图4-6中未给出）用来控制ALU应该执行哪些具体的算术或逻辑操作。但是具体的等待时间是由具体的操作所决定的，最大时间（也就是最坏情况）决定了在这块ALU电路上的时钟能运行的频率。在当今系统中，这种延迟一般为一个或两个纳秒。

这种延迟不能忽略，但这里的问题是没有有效长最小延迟，这意味着只要输入信号被取消或发生改变，输出结果就会混乱。这个问题所带来的结果就是输入操作数必须保存在一个地方来驱动ALU完成计算结果输出和存储，只有这些做完，输入才可以改变。

因此，ALU的输入端必须配置寄存器。对于一个用双总线来驱动ALU的结构来说，要想同时输入数据和得到结果，必须至少配备一个寄存器。如果有两个寄存器，那么就会有更多的可用锁存器硬件的方案供选择，不过影响更多的是下面的操作：

```
ADD R0, R1, R2  ;R0 = R1 + R2
```

下面每个序号标注的步骤按时间顺序执行：

1. 建立系统，清空总线并且将ALU的功能调制成加法模式；

2. 让寄存器R1驱动总线1（打开寄存器的输出缓冲器），让寄存器R2驱动总线2；

3. 将总线1上的数据锁存进ALU的第一个操作数寄存器，同时让总线2上的数据锁存进ALU的第二个操作数寄存器；

4. 关闭R1的寄存器输出缓冲器（总线1空闲），关闭R2的寄存器输出缓冲器（总线2空闲）；

5. 等待最大的传输延迟时间；

6. 将ALU的结果锁存进ALU的输出缓冲器中；

7. 允许 ALU 的输出缓冲器驱动一个总线；

8. 将刚刚计算出的结果锁存进寄存器 R0 中；

9. 关闭 ALU 的输出缓冲器（两个总线空闲，系统准备好下次操作）。

可以看到，非常简单的加法指令实际上包含了很多必须在硬件上执行的步骤。除了 ALU 的传输延迟外，这个过程只有 8 个步骤。在三路总线的设计中（见 4.1.1 节），这样的一个加法也许只需要 3 个时间周期或者说 3 步。

一个简单加法指令的复杂执行步骤说明了在 CPU 中控制单元的重要性（见 3.2.4 节）。你能想象得到对一个多周期的 CISC 指令进行控制的复杂性吗？ 128

4.1.6 单总线体系结构

单总线体系结构可以从上一节的内容中推测出来。我们仍然使用一个 ARM 风格的处理器作为例子，这个结构也许和图 4-7 看上去很相似。

注意体系结构简洁性的设计，其实就是将这个例子中的多步骤操作的复杂性给彻底掩盖掉。我们考虑上一节中的那个计算功能，其在单总线体系结构下的表现为：

1. 建立系统，清空总线并且将 ALU 的功能调制成加法模式；

2. 让寄存器 R1 驱动总线（打开寄存器的输出缓冲器）；

3. 将总线上的数据锁存进 ALU 的第一个操作数寄存器；

4. 关闭 R1 的寄存器输出缓冲器，让寄存器 R2 驱动总线；

5. 将总线上的数据锁存进 ALU 的第二个操作数寄存器；

6. 关闭 R1 的寄存器输出缓冲器，等待 ALU 的最大传输延迟；

7. 将 ALU 的结果锁存进 ALU 的输出缓冲器中；

8. 允许 ALU 的输出缓冲器驱动总线；

9. 将计算结果锁存进寄存器 R0 中；

10. 关闭 ALU 的输出缓冲器（总线空闲，系统准备好下次操作）。

与上一节中的双总线体系结构相比，多出来的两步和效率的降低值得注意。历史上对单总线体系结构的改进就是增加了一个非常简单低成本的结果反馈总线，如图 4-8 所示。

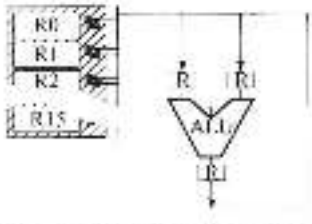


图 4-7 连接 ALU 和寄存器组的单路总线

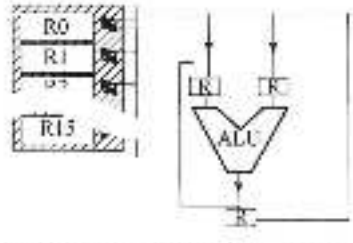


图 4-8 以如图 4-7 所示连接 ALU 和寄存器组的单总线结构为基础，增加了一个从 ALU 的输出到 ALU 输入锁存的反馈连接 129

虽然在这里有几个备选方案也可以执行以上操作，但是所有都是将 ALU 的运算结果反馈到 ALU 输入的一支。这也许对于累加操作或者随后的算术/逻辑操作有用。这样，图中 ALU 左边的寄存器就成了著名的累加器，它几乎是所有操作的基础，在整个系统中是使用频率最高的，也是程序员的朋友。老牌底层程序员都爱并且要累加器，其中很多人为它被 RISC 和 CISC 所现杀而感到伤心。新西兰的著名工程管理专家 Adrian Busch 总结道："如果 CPU 没有一个累加器，那就算不得一个真正的 CPU。"

4.2 算术逻辑单元

4.2.1 ALU功能

显然，一个算术逻辑单元（ALU）作为计算机的一部分用来执行算术和逻辑操作。但是这些操作具体是指什么？下面两个处理器的ALU操作也许会对我们有所提示：

- ADSP2181——加、减、自增、自减、与、或、异或、传递/清空、取反、非、绝对值、置位、位测试、位翻转。有一些限制是某些寄存器只能用做输入并且只有两个寄存器可以用做输出。
- ARM7——加、减、自增、自减、与、或、异或、传递/清空、非。所有的寄存器都可以用做输入寄存器，也可以用做输出寄存器。

一般情况下，ALU执行按位逻辑操作、测试、加或减。同时，通过对以上这些多个ALU操作进行组合，可以派生出许多其他的功能操作。

一个基本的ALU，能够执行加法和减法，可以分割为单个位的操作链，与2.4.2节中的进位传递加法器类似，如图4-9所示。图中没有给出控制和功能选择逻辑，8个独立的单个位ALU按位完成2字节输入和1字节输出的加法。这个操作可以写为：

R = ALU.op(A, B)

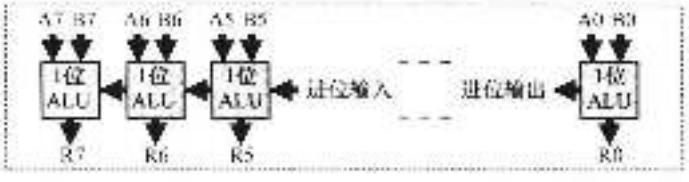


图4-9 并行按位功能链示意图，其中并行的单个位单元并行地组成了1字节位宽的ALU

[30]

一共4位的ALU操作例子在下表中给出：

0001	AND	1110	=	1000	按位与
0011	AND	1010	=	0010	按位与
1100	OR	0001	=	1101	按位或
0001	OR	1001	=	1001	按位或
0001	ADD	0001	=	0010	加法
0100	ADD	1000	=	1100	加法
0111	ADD	0001	=	1000	加法
	NOT	1001	=	0110	取反
0101	SUB	0001	=	0100	减法
0110	EOR	1100	=	1010	异或

从第2章的内容我们知道，加法和减法都不是并行的按位操作，也就是说一个加法的第n位结果不仅仅依赖于每一个输入数据的第n位，而且还和前面的第$n-1$位、第$n-2$位、…、第0位有关系。事实上，两个值之间的算术操作一般都不是由单个位并行的方式完成的，但是逻辑操作是可以的。

知道了ALU可以完成哪些类别的功能以及学习完这些例子，现在让我们去完成一个ALU底层设计来探究它究竟是如何具体操作的。

4.2.2 ALU设计

图4-10是ALU模块通用符号示意图，可以看到该ALU有n位宽输入操作数A和B，还有n位的输出操作数。

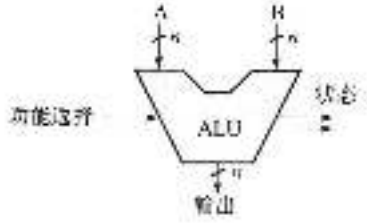


图4-10 常使用模块符号来表示一个ALU，该ALU拥有n位的输入操作数A和B，功能选择逻辑，以及最终n位的结果输出和状态标志输出

一个多位的控制接口作为功能选择用来选通ALU的具体功能。状态信息包含了结果是正、负、等于零，以及一个进位或者一个溢出。在一些处理器中，这些值都简写成N、Z、O[2]和C。

之前		操作	之后	
R1	R2		R0	标志
5	5	SUB R0, R1, R2	0	Z
5	10	SUB R0, R1, R2	-5	N
假设寄存器是8位的，这样一个8位的寄存器就可以存储0～255的无符号数和-128～127的有符号数				
255	1	ADD R0, R1, R2	0	C
127	1	ADD R0, R1, R2	128（无符号数），-128（有符号数）	O, N
-1	1	ADD R0, R1, R2	0	Z, C

[131]

注意：对于二进制的8位数，其加法，比如01111111+00000001总是等于10000000。问题是如何去解释这些。输入的数据是十进制的127和1，如果将它们解释为补码（有符号二进制数），那么输出是-128；如果解释为无符号二进制数，那么输出是+128。没有更进一步的信息，只有程序员才会知道这些算术运算究竟是哪种类型的运算。

溢出标志O是用来辅助两个补码计算的。对于ALU来说，有符号数和无符号数是没有区别的。然而，当一个涉及补码的计算有可能溢出时，ALU会通过写入溢出标志位来提示程序员该运算溢出。如果程序员是在处理无符号数，那么可以忽略这个标志位。可是当运算涉及两个补码时，就需要注意这个溢出标志了，它表示计算结果太大而无法在规定范围内表示。

对于在这里我们所设计的ALU，将考虑一个简单的进位标志，而其他的状态标志暂时不考虑。我们仅涉及与、或和加这3种功能。该ALU是个多位并行ALU，我们只设计一位加法器（因为所有位都相同）。

最终的设计结果，其逻辑图如图4-11所示。表4.1以该设计为基础演示了如何计算该ALU的最大传输延迟。

---

② 此处采用“V”来表示溢出标志，而不是“O”，这可能是担心与0（零）混淆。

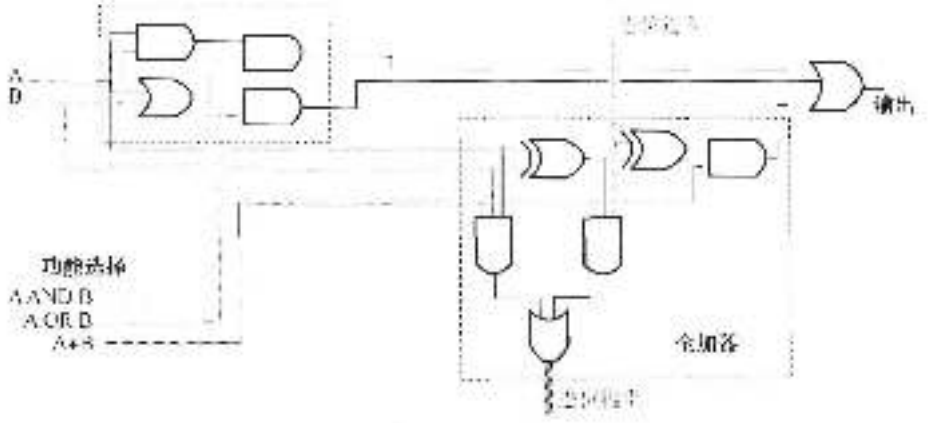


图4-11 具有单个位ALU的内部逻辑连接示意图

练4.1 探讨ALU的传输延迟

让我们来计一下方框中每个逻辑门会产生4ns传输延迟，这意味着每个新的数据输入要产生一个有效的稳定输出数据所需的时间。

在图4-11中的ALU到未来开发最不理想的情况下的最长路径（忽略功能选择信号），输入A和B都要经过两个区域的门，在左上部的区域中只有两级门，而经过右下方的全加器时，到达输出的其输入要经过4个门，即为C_{out}要经过3个门。

此状况说，最低有效位输入到其之前最经过两个门，而结果Z输出之前最经过3个门。这个结论可以表格如下：

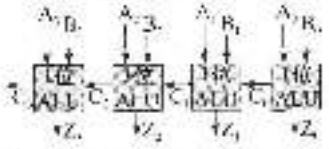
A/B到Z：4×4ns=16ns

A/B到C_{out}：3×4ns=12ns

C_{in}到Z：3×4ns=12ns

C_{in}到C_{out}：2×4ns=8ns

我们可根据这些数字来计算一个4位ALU在执行加法时在最坏情况下的传输延迟：



这是A+B=Z，因为这是一个加法，我们需要计算进位传输。我们假定可以忽略最初输入传输所经过的门，直接输入到ALU的右侧，即由进位输出来传送。最后ALU最高有效位，因为从任何输入到Z输出的延迟都不过是这位所经过的门延迟，因此最坏情况可以总结如下：

Bit 0：A/B到C_{out}12ns

Bit 1：C_{in}到C_{out}8ns

Bit 2：C_{in}到C_{out}8ns

Bit 3：C_{in}到Z 12ns

总共：40ns

如果你也不最快时间如下执行，则传输周期不能为40ns，以确保对于每个输入都正确的选择结果。当然时序正确输出比我的速度要比上面快，但是没有一个简单的方法以事先决定结果输出的次数。因为我们必须考虑最长传输延迟路径时间，所以当时钟频率为1/40ns=25MHz。

实际中典型处理器最快的时钟频率。我们可以采用更快的逻辑门，允许加法器在两个时钟周期内完成，或者使用一些其他技巧加快加法器。一个技巧是采用进位预测表超前方法（8.2.4.3节），这种方法很大，但是当加法器的操作数位数很大时要比用大量的门好。

4.3 内存管理单元

对CPU而言，一个内存管理单元（MMU）允许物理存储器以另外一种逻辑组合方式呈现。它是存在于CPU和主存储器之间的硬件，该硬件连接在存储器读写总线和逻辑存储组织之上，即所谓的虚拟存储。它是1962年在曼彻斯特大学发明的，有时候也称之为分页存储。

4.3.1 对虚拟存储的需求

虚拟存储给CPU提供了一个很大的存储空间，而且是用户程序可见的。现实中，物理存储往往很小，而CPU所需要的虚拟存储内容必须载入物理存储中。许多现在的操作系统，例如Linux就是建立在虚拟存储之上的。

虚拟存储允许在计算机上运行的程序规模超过可用的RAM。当然，这个功能的实现要求巧妙的编程，并且要外接一个更大的存储器，比如硬盘。然而，一个MMU可以确保程序在执行时存储器看上去是连续的，而且是够大。确实来说，MMU所负责的是计算机中存放程序的虚拟存储空间以及所对应的物理RAM。

提出虚拟存储的初衷是为了解决快速而又昂贵的RAM和缓慢而又廉价的硬盘之间的不平衡。使用虚拟存储可以让一台计算机的成本更加低廉，这台拥有较小RAM的计算机可以表现得和拥有更多存储空间的高成本计算机一样好，唯一的不同就是有时候存储器的访问会比较慢。

当MMU工作时，与纯RAM相比，平均存储器访问速度将降低，这是因为硬盘的速度实在是太慢了，该代价可以看成是对这种设计的折中。

注意，二级存储不一定非得是硬盘，它可以是任何存储介质，只要可以提供更大的廉价容量以及速度比在RAM慢就行，包括较慢的闪存等。

4.3.2 MMU操作

在现代的MMU系统中，未使用的页面通常存储在低速而又廉价的硬盘中。

图4.12是一个MMU连接的简单例子。在这里，就CPU而言，该系统有一个32位的地址空间（因此它可以寻址2^{32}个内存单元，或者说4GiB的内存），然而该例子中存储器的位宽只有20位（即2^{20}个内存单元，也就是1MiB）。而通过MMU，CPU可以看见4GiB的空间，而不是1MiB。

132 - 134

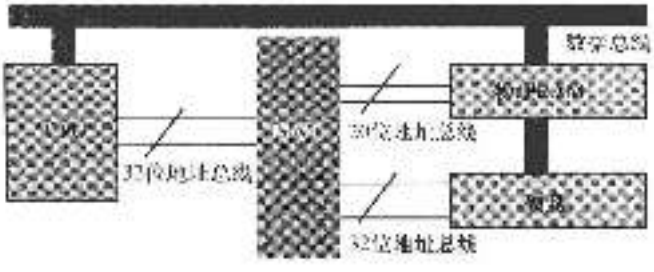


图4.12 一个MMU与连接CPU和物理RAM及硬盘，数据总线将这些部件直接相连，MMU用来转换地址总线信号以控制存储器上的数据读写

存储器划分为页（page）。如果我们假设每个页面为256KiB大小（这是个典型值），那么主存可以存放4个页面，但是CPU可以访问多达16 384个页面。

MMU将新的页面载入RAM，并且将暂不使用的页面写回硬盘（硬盘足够大）。如果CPU要请求的页面没有加载到RAM上，这个时候MMU首先需要将一个未使用的页面从RAM返回到硬盘上，然后再将CPU请求的页面加载到RAM上。

至少需要跟踪每个页面的状态，知道它们哪些正在被使用，哪些处于空闲状态。这种设计思想与 cache 类似（见 4.4 节）。两个查找表用来记录当前 RAM 和硬盘的存储信息，这两个表分别叫做物理 RAM 占用表和磁盘存储占用表。

在 MMU 的管理下，如果 CPU 的请求页面已经在 RAM 中，那么我们称之为命中；如果没有在 RAM 中，我们称之为页面缺失或未命中。这个操作如图 4-3 所示，栏 4.2 给出了一个实例。

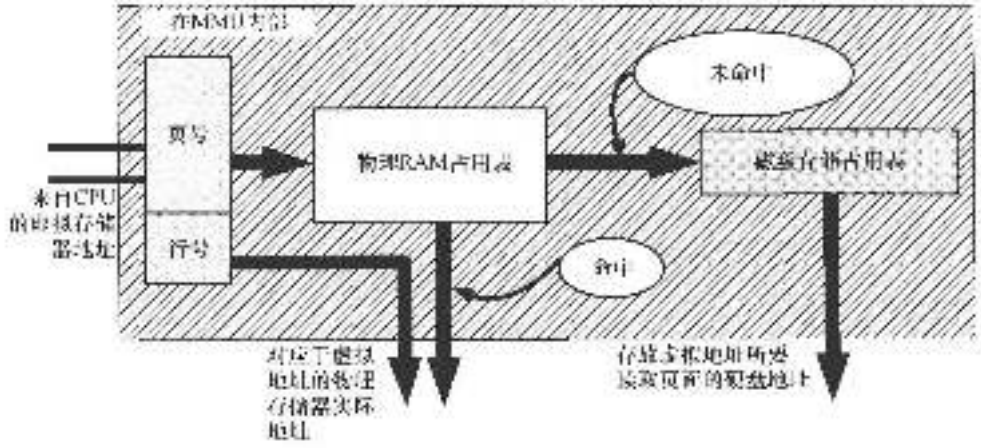
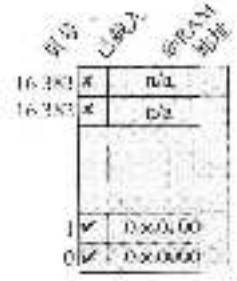


图 4-13 MMU 的简单示意图，给出了内部连接的关系，以及缺失和命中所带来的结果

栏 4.2 MMU 实例

对一个简单的 CPU 中，使用 RAM 占用表如下图所示。为简单，表中的每一行都是看作计算机内为一个独立页地址。有点的表格表示每行第一个页面被载入了 RAM，以及如何载入。载入了 RAM 中的哪个位置。



注意表中的从 0 到 RAM 中的 0 号地址，而从 1 到 0x0100 单元。为了，我们知道页可以放置在 RAM 的任何地方，为简单起见我们假定页面的大小是 0x0100（256）个单元。这是非常小的地址空间，允许从 0～255 之间的数表示行号。

我们也可以看到总共有 16 384 个页面，需要 14 个才能够完全表示。这样我们的虚拟 CPU 上可以发送的存储器地址是 14 + 8 = 22 位。8 位的地址用来表示行号，还余为 14 位用来表示页号。因此 22 位来表示，我们将能够有 2^{22} = 4MB 为容量空间（假设每个位置可以存放一个字节），也就是 16 384 × 256 = 4 194 304。

注意：从这个例子中我们可以知道在计算机中从 CPU 的逻辑地址是如何分为行号和页号的规则。

当一个 CPU 从存储器的 X 位置读入时需要经过以下步骤：

1. CPU 将地址 X 写入地址总线中，然后发出一个读信号。
2. MMU 发出一个信号让 CPU 等待一段时间，在这段时间内 MMU 将取回地址 X 的内容。
3. MMU 将地址 X 分成页号与该页中的行号。
4. MMU 查询物理 RAM 占用表。在这一步中有命中和未命中两种情况：

- 如果请求的页面已经命中，则 MMU 输出要读取的物理 RAM 地址。物理 RAM 地址加上

页面内的行号成为最终的物理 RAM 地址。

- 如果请求的页面缺失，则将页号传递给磁盘存储占用表。在该表中查找到需要读取的页，然后将整个页面加载进 RAM。因为此时该页面才进入 RAM，因此不以像页面命中时的方式将地址 X 的内容送回给 CPU。
- 注意到物理 RAM 在大小上是不受限制的，则必然有一个将页面退回给硬盘的过程，这个过程根据追踪页面的使用情况来决定返回哪些页面。

5. MMU 输出地址 X 的内容到数据总线上，然后发出信号通知 CPU 数据已经准备好。

很明显，在页面缺失的时候 CPU 必须等待一段较长的时间来从存储器中得到一个值。硬盘可能比 RAM 要慢好几百倍，不管厂商如何努力，查找的过程相对来说还是比较慢。我们将这个等待时间称为*停滞时间*（stall time）。

有时候一个程序员并不想等待这么一段停滞时间。在这种情况下，一些对速度要求极为严格的变量或者程序一般都存放在一个特殊的页面中，该页面被固定在物理 RAM 上。事实上，页面属性允许高级 MMU 用几种不同的方式来处理页面。现代操作系统将中断服务例程和底层调度代码以这种形式固定在物理 RAM 上。

将存储页面放在硬盘上并在使用时将其载入 RAM 的方法，看上去比较符合逻辑，使用户体验到的存储空间比实际更大。然而，实施这种方法也存在很多困难，当要载入一个新页面时哪个页面应该退回？或者每个页面究竟为多大最好？这些都是下面两节中所要考虑的问题。

4.3.3 退回算法

如果一个新的页面要从硬盘加载进物理 RAM，除非 RAM 刚好有空间，否则必须有一个已经载入内容的页退回给硬盘，来给新页面腾出空间，并同时更新 RAM 的占用表。

有多种算法用来解决此类问题，以决定应该将哪个页面退回给硬盘。

- **LRU**：最近最少使用算法，也就是最近一段时间内最少使用的页面应该退回。
- **FIFO**：先入先出算法，最早进入的页面应该退回。

这两个算法各有优缺点。微软的 Windows 操作系统用户在配置较低的机器上使用该操作系统时对磁盘抖动（disc thrashing）问题较为熟悉——此问题表现为磁盘看上去是一直在运行。这个问题产生的原因就是采用了一个很坏的退回算法。考虑一个很大的循环程序，它的代码分别存储在好几个页面上，在这种情况下，仅仅是从循环底部返回到循环的顶部就可能会引起这类问题，而这一现象是存放循环顶部的页面已经被退回所引起的。

135 ~ 137

最坏的情况是一个很大的程序，其变量分布在多个页面上。如果一个小程序要将单个数据分别赋给这些变量，则包含这些变量的页面也许会仅仅为一个变量的写入而全部被调入 RAM。在这种情况下，编译器和操作系统很难通过聚集存储位置来优化程序。

这个退回问题和 cache 一样，面临着相同的问题，将在 4.4 节中讨论。

4.3.4 内部存储碎片和片段

如果一个完整的页面用来存放一点点的内容，对于存储器来说就会非常低效。或者如果一个程序比一个页面稍大一点儿，以至于仅仅几行的程序不得不存放在另一个空的页面中，这样这个程序就占用了两个页面，但其实这个程序更近似于一个页面的大小。

这两种情况下，计算机快速而又珍贵的 RAM 必须得保存这些没有用到的空间。而且，为了载入这些零头数据，冗长而又慢速的页面退回和载入新页面的过程必须每次都得执行。我们称这种情况为内部存储碎片。

解决此类问题的一种方法就是减小页面的规模。然而，这样做的后果是 MMU 的查找表变得

很大，而且会使查找过程本身成为 MMU 操作的一个瓶颈。

最新的解决方法是引入一个存储器片段——一种可变长度的页面，而且这种页面可随着程序执行的需要增大或减小。一段 C 语言程序可能会使用一个存储片段来存放本地函数变量，用另一个片段存放全局变量，再用一个片段存放程序堆栈。尽管 C 程序员不需要关心底层细节，但潜在的程序操作可能会通过片段编号和片段（行）中的地址来读取变量，这种设计叫做二维存储。

存储片段的一个优点就是片段之间不会彼此扰乱。片段中存储的程序会执行，而存储的数据则不会。与此类似，如果一个程序要把变量写到另外一个应用程序的片段上，也是不允许的。

4.3.5 外部碎片

片段存储方式会更加复杂，其原因是这种方式需要对每个片段的大小和内容进行追踪，也就使得各种占用表更加复杂。然而，由于避免了内部存储碎片的麻烦，这种方式比起原始的页管理方法更加高效。

138

遗憾的是，当将这种方式用于解决外部存储碎片时会有点儿麻烦，如图 4-14 所示。从左到右看此图，一个原始的程序载入进来①，占用了 4 个存储片段。在②中，操作系统想要从片段 5 中读取一些数据，所以要退回一个片段（本例中选择了片段 3）。然后在③中，片段 5 被加载进来。在④中，片段 1 被退回到硬盘，而在⑤中操作系统想要读取片段 3 并将其重新加载进来。

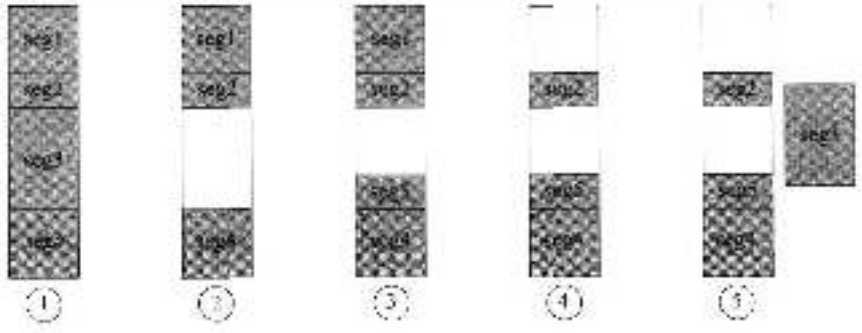


图 4-14 图解外部碎片：在内存片段加载和移除过程中，经过以上 5 个步骤之后，在外部存储中有足够的总存储空间，但没有足够的连续存储空间用于加载 seg3

此时，对于片段 3 来讲很明显有足够的空闲 RAM 空间，但是没有足够的连续空闲空间。有两个备选解决方案。第一个就是分割片段 3 为两部分，然后将其加载到合适的地方。第二个就是先整理存储空间，然后再载入片段 3。第一个解决方法也许在这个例子里是可行的，但是会在复杂性和时间上变得很差，因为这个片段会被越分越小。由于这个原因，通常采用第二个方法。这种整理的过程称为压缩，并且在载入片段 3 之前执行，如图 4-15 所示。

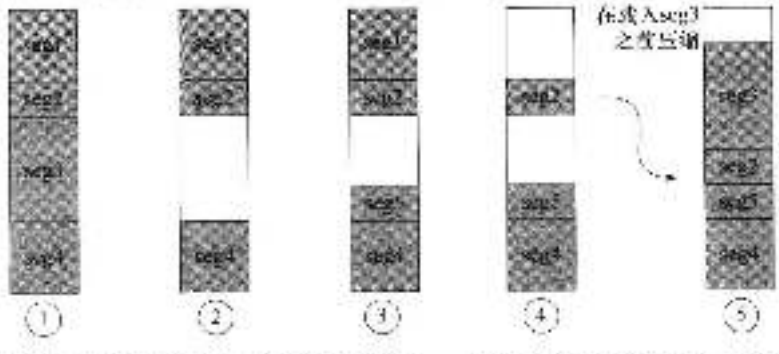


图 4-15 完成图 4-14 中的存储操作，并且在重新载入 seg3 之前压缩空门片段，对存储内容进行重组

很明显压缩会花费一些时间，所以只能在必需时使用该方法。

片段管理算法多年来一直是一个研究热点。一般做法都是追踪存储器的使用和未使用的状况，然后以某种方式高效地压缩空白片段。有一些简单的算法，默认的会在出现空隙时执行压缩。

4.3.6 改进的 MMU

4.3.2 节中所讲述的 MMU 硬件可以很好地处理定长页面，但是对于可变长度的片段会怎样呢？要知道所有被请求的位置必须在物理 RAM 占用表中查找，因此其速度对于全局的存储器读写来说显得至关重要。简单地将地址总线一分为二，将后面的一些比特位看做行号，前面的一些比特位看做页号，这种做法无法满足片段存储的需求，因为在这里每个页面的大小都不一致。这也意味着占用表成为一个复杂的地址内容查找表（LUT）。 [139]

这些复杂的 LUT 的查找时间与其规模成正比，也就是说表的规模越大，查找起来越慢。问题是，为了减小外部存储碎片，系统需要对相对较小的片段/页面进行处理。考虑 UltraSPARC Ⅱ 这个例子，它支持 2200GB 的 RAM，但是其页面规模为 8KB 大，也就是最坏情况下有 200 000 个页面。存储这些页面信息的 LUT 会变得非常慢，意味着所有的存储读写，在或者不在 RAM 上，都将会被查找过程极大延误。

解决这个问题的方法是对经常使用的页面，引入一种较小而又快速的查找表，而把不经常使用的页面存放在慢速的查找表中（或者 RAM 中）。这可以有效地缓存查找表的内容，也叫做转换后援缓冲器（Translation Look-aside Buffer，TLB），它也有其他名字，比如转换缓冲器（Translation Buffer，TB），目录后援表（Directory Look-aside Table，DLT）或者地址转换高速缓存（Address Translation Cache，ATC），如图 4-16 所示。

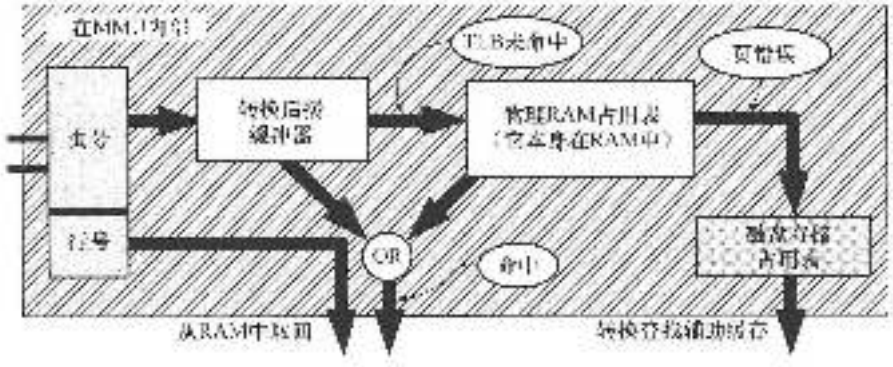


图 4-16 MMU 使用转换查找辅助缓存 TLB 进行操作，可以与 4.13 没有使用 TLB 的情况进行对比

在本书的写作过程中，Ultra SPARC Ⅱ 和 MIPS R3000 处理器采用了这种技术，但是像 ARM7、x86 系列以及数字信号处理器并没有采用。它通常用于一些要求速度非常快的工作站级处理器中。

4.3.7 内存保护

除了能够对进出 RAM 和硬盘上的数据进行交换控制外，对系统设计者而言，MMU 还有一些其他的好处。事实上，RAM 越来越廉价，很多软件已经不把节省 RAM 空间看做一个设计目标了，对于嵌入式处理器来讲也是同样的，尽管没有外部存储器（比如硬盘），MMU 依旧存在。问题是，为什么当最初的目标已经渐渐消失的时候，系统设计者还依旧坚持使用 MMU 呢？ [140]

最主要的原因就是存储保护。因为 MMU 是位于主存储器和处理器之间的，无需处理器的干

预就可以很快地对地址进行扫描和更改。遇到任何问题时（比如说你请求了一个不存在的地址）MMU 可以向处理器发出预警信号。在 ARM 中，这可以通过称为数据异常（data abort）的中断信号来实现。或者当取一条不存在的指令时，叫做取指异常（prefetch abort）。设计人员应该编写底层异常处理程序作为操作系统的一部分以应对这些问题的发生。

从软件的角度来看待此问题，系统程序员可以让 MMU 去限制对存储器各个位置的访问，或者标记存储器区域哪些已分配哪些未分配。编译后的代码一般都有一些程序区域和数据区域，对程序区域通常都不会再进行写入操作，但是数据区域可以。当应用了 MMU 后，如果一段程序区域的指令正在执行，那么存储器上的其他区域可以读写，而不会向正在执行的存储区域进行写入。

在大部分的现代操作系统中，用户代码分不清哪些存储器位置是可写的哪些是不可写的，
141 它只能朝被允许的地方写。这也就防止了用户代码写入错误的地方，进而导致系统崩溃。

非操作系统的代码是不允许对操作系统控制的寄存器进行写入的，也不允许对其他程序的数据区域进行覆盖写入。这对操作系统的安全性和可靠性至关重要。一个很重要的例子就是保护 0 地址。一些经常犯的编程错误（见框 4.3），都是由于从 0 地址读或者写入了 0 地址而造成的。在 Linux 中，如果一个编译好的 C 程序试图尝试此类操作，将会被自动退出并给予一个“segmentation fault”的错误提示。

框 4.3 追踪 C 程序中的软件错误

一般 C 编译器都会将新定义的变量初始化为 0。这有助于追踪在 0 地址发生的错误：

```
int *p;
int x;
x=*p; //since p is set to NULL (0), a read from here will
trigger a data abort
```

如果没有足够的存储空间，用 malloc() 库函数来定义一块存储空间时就会失败，并且返回 NULL。

```
void *ptr=malloc(16384);
//we forgot to check the return address to see if malloc failed
*ptr=20;
//since ptr holds NULL (0), this will trigger a data abort
```

类似地，还有一个函数在运行时分配内存空间。

```
boot_now()
{
     void (*theKernel)(int zero, int arch);
     ...
     ...
     printf("Launching kernel\n");
     theKernel(0, 9);
}
```

在这段代码里，函数 theKernel() 在第一行中被定义并且指定了一个存储器地址，而该地址已经被操作系统内核所使用。然而，程序员忘记了这个问题。在默认情况下，该函数将被分配到 0 地址，从而在执行函数 theKernel() 时会跳到 0 地址上，最后导致一个预取异常（prefetch abort）。

注意传递给函数的值 0 和 9 在分支发生前是存放在寄存器 R0 和 R1 内的。如果在嵌入式 Linux 系统中，
142 该函数依旧执行，那么系统在运行时就会使用当前 R0 和 R1 内的数据，不会抛出异常错误。

4.4 cache

高速缓存存储器（cache）有着非常快的存取速度，但其成本也非常高，通常其物理位置更靠近 CPU。如果成本不是问题，计算机设计者在其系统中就只采用快速存储，但这样一来对大众消费者来说，这种设计就会变得非常不经济，当然超级计算机除外。

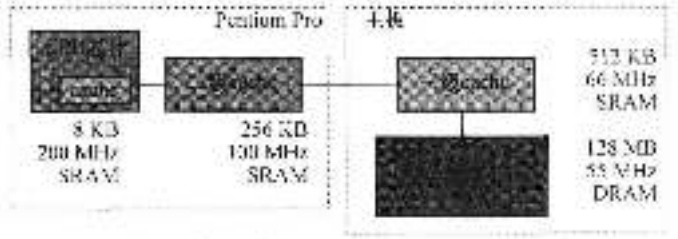
cache 在存储架构中的层次在 3.2.2 节中有相关介绍。我们知道，在存储架构中越接近顶部的存储器速度也就越快，但成本也越高；反之亦然。

cache 是试图提高存储器的平均存取速度，而 MMU 则用来允许计算机拥有更大的存储空间，但这样一来，势必会降低平均存取速度。与 MMU 不同，cache 是不受任何操作系统干涉的。然而，它对于应用程序员来说是透明的，这一点就像 MMU 一样。

在实际系统中，不只有一个 cache，事实上有很多不同等级的 cache 以不同的速度在运行。最高等级的 cache（最靠近 CPU）通常以片上存储的形式实现，这些 cache 通常都很小（例如 ARM 和 80486 仅有 8KB），随着 cache 的等级降低其容量逐渐增加。框 4.4 以 Pentium Pro 处理器系统作为实例很好地阐述了 cache 的概念。

框 4.4 Intel Pentium Pro 的 cache 实例

Intel 的 Pentium Pro 的 cache 设计方法在当年极具创新性，在 CPU 内芯片内用了 256KB 的 cache，不过是与 CPU 相互开来。过渡的是，这个设计（见 7.3）后来发现不是很稳定，并且最终成为一个失败的产品系列。



在该图中，可以看到 CPU 内置了一个相对较小的一级 8KB cache，而二级 cache 与 CPU 在同一个芯片内，但是速度为一级 cache 的一半，容量可是其 32 倍。三级 cache 是快速 SRAM，位于主板上，比二级 cache 要慢但容量为二级 cache 的双倍。最后，相比前面 3 种 cache，主存容量足够大，但也很慢，它是一种低成本高密度的存储体 DRAM（dynamic RAM），但速度要慢于 SRAM。

注意：当今的 cache 系统与此仍然十分类似，但是在规模上大了许多，每一个 RAM 的大小都可能多了几个零，甚至出现了更高级别。主存已经从 SDRAM（Synchronous DRAM）转变为 RDRAM（Rambus），或者 DDR（Double Data Rate，双速率）RAM 等（见 7.6 节）。

将 cache 划分开来，可以分别用于数据和指令存储，这在哈佛体系结构（见 2.1.2 节）的处理器中是必要的，但对于冯·诺依曼体系结构来说也是很有益处的。例如，具有创新性的 DEC StrongARM 处理器是基于 ARM 的，因此其内部是冯·诺依曼体系结构，不过该处理器却采用了一个哈佛体系结构的 cache。这样可以让两个 cache 部分对不同的存储行为进行优化：比如说程序存储存取偏向于顺序操作，而数据存储操作更偏向于跳跃式读写，因此需要不同的 cache 策略和结构来适应这两种不同类型的操作。

与虚拟存储类似，当 CPU 请求的数据不在 cache 中时会产生 cache 缺失，并且不得不从速度较慢的存储器中调入数据。就像虚拟存储那样，cache 中的某些数据必须首先返回，然后进行一些压缩。

请求的位置可以在 cache 中找到的比率称为命中率，因此这也是考察 cache 性能最重要的指标。通过良好的 cache 组织和一个高效的 cache 算法可以提高命中率。

有很多不同形式的 cache 组织方式，这些方式对 cache 的成本和性能表现有极大的影响。其中有 3 个最最常见的：直接相联 cache、组相联 cache 和全相联 cache，接下来将对这 3 种方式进行介绍。

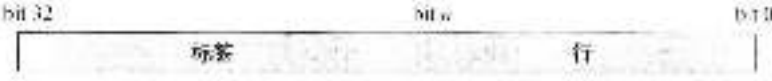
注意在现代 CPU 中，cache 实际上是整块地读取内存，可能是一次读取 32 字节或者 64 字节，而不是只读取一个存储单元。为了简单起见，本节给出的例子中若每个 cache 块只包含一个内

存地址。在更加现实的存储块情况下，cache 中的标记位地址是块地址的起始，cache 控制器知道连续 n 个存储器位置必须从一个缓存行（cache line）读入。这种方式的好处是在像 SDRAM 或者 RDRAM 这样的现代存储器中，对连续位置的读写比对多个单个地址的读写要高效很多。

4.4.1 直接相联 cache

这个设计方案中，每个 cache 位置可以保存一行来自主存储器的数据。每个主存储器的地址对应一个固定的 cache 位置。由于 cache 的容量远小于存储器，因此每个 cache 位置要对应很多存储器位置。

因此，当要求直接相联 cache 返回一个特定存储器地址上的内容时，只需要查看 cache 位置上的标签的正确性。cache 位置取自存储器地址的低 n 位（假设 cache 和存储器的位宽相等），比如32 位的例子，如下图所示：



144

将 cache 位置分为标签和行的做法类似于在 MMU 中的页与行的划分法（见 4.3 节）。直接相联 cache 的位置数等于行数。存储器上的每个页与有相同的行数，所以如果从一个页中读取一个数据，它将按照自己所属的行来载入具体的 cache 中。

每个 cache 位置实际上包含了一些域：一个已用/清空（dirty/clean）标志位，用来标识 cache 位置上的数据是否更新或者有没有存进主存里；一个有效位用来标识该 cache 位置是否被占用；一个标签位用来标识哪一个可能的存储器页已经实际载入 cache 中；在这些标志位之后，cache 存放了从 RAM 中读取的数据。

所以直接相联 cache 算法应该是：

- **CPU 从存储器中读取**——将读取地址分为 TAG 和 LINE 两部分。检查 cache 的 LINE 位置，并查看标签位是否与 CPU 要求的一致。如果一致，则从 cache 中读取数据。如果标签位不匹配，则查看已用标志位。如果这个已用标志位设定为有效，则首先回退当前 cache 位置内的数据到主存上，然后读取相应地址的数据到当前位置 cache 上。清空已用标志位，设置有效标识，并且更新 TAG 位。
- **CPU 向存储器写入**——有以下几种情形：
  - 直写（write through）：将数据写入 cache 行中并且也写入主存中。
  - 写回（write back）：不写回主存（这种情况仅发生在下一个时间另外一个存储器位置要使用该 cache 位置的时候），只存在 cache 上。
  - 延迟写（write deferred）：允许写到 cache 上，然后经过一定时间后写回主存（假设有这样一个时间，并且 CPU 没有在等待）。

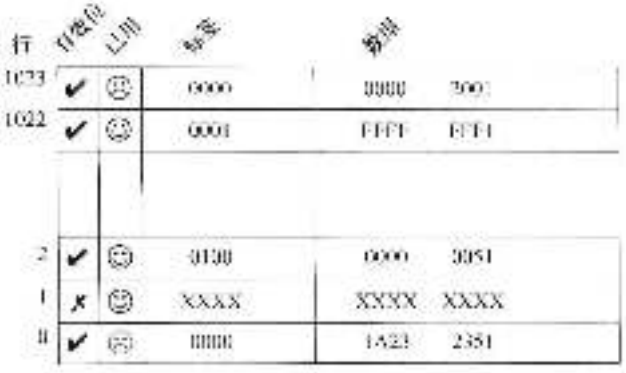
cache 数据一旦写回到主存，就要清空已用标签，以标识现在的主存数据和 cache 数据是一致的，这叫做 cache-memory 一致性。

在直写方式中，如果将要向其中写数据的存储位置不在 cache 中，则可以直接向存储器写入数据，从而跳过 cache。这种方式叫做无写分配的直写（write through with no write allocate，WTNA）。旧数据总是存储在 cache 中，不管这个数据是否写到了存储器中。这种方式叫做有写分配的直写（write through with write allocate，WTWA）。

直接相联 cache 的主要优点就是查找速度快。对于每一个主存储器的位置来说，只有一个 cache 位置与之对应。然而，这个优势也是个问题，每个 cache 行要对应很多真实的物理存储位置。框 4.5 给出了一个直接相联 cache 存取的例子。

框4.5 直接相联 cache 实例

下图给出了在一个简单微型计算机系统中，直接相联 cache 当前正在使用的情况。



行	有效位	已用	标签	数据	
1023	✔	☺	0000	0000	2001
1022	✔	☺	0001	FFFF	FFFF
2	✔	☺	0100	0000	0051
1	✗	☺	XXXX	XXXX	XXXX
0	✔	☺	0000	1A23	2351

这个 cache 在 1024 行（对应于 10 位的地址总线），每行有两个标志位，一个标签位和实际缓存的数据。笑脸符号表示未写（clean），悲伤符号表示已写（dirty）。

在系统的开始阶段，所有的 cache 条目都被清空而且是无效的，像第 1 行。这可能意味着第 1 行在系统重置后就没有使用过。

第 0 行是有效的，所以它肯定缓存了实际数据。它该行是已用状态，说明该数据最近有定改变过，并且新数据还没有写入 RAM 中。其标签位为 0，这第 0 行必然存放的是 CPU 的 0 地址数据，而且最后的数据为 32 位的 0x1A23 2351。

由于 cache 中总共有 1024 行，因此第 0 行还可以存放 0x400（1024）、0x800、0xC00 这些地址的数据，但实由于其标签位为 0，所以必然代表的是 0 地址。

第 2 行也有效但是清空的，意味着其存储的数据与 RAM 中所存放的一致。它所对应的页（标签）位置为 0x100，因为行数表示了低 10 位的地址总线，因而所缓存的数据在 RAM 中的地址为（0x100 << 10）+ 2 = 0x40002，且该数据为 0x51。

最后，第 1023 行有效但已用，意味着该位置的数据已经改变。其标签位为 0，所以读取的数据在 RAM 中的地址为（0x0 << 10）+ 1023 = 0x003FF。

4.4.2 组相联 cache

直接相联 cache 的问题在于物理地址为 0，1024，2048，3072…的内存单元都链接在同一个 cache 行上。如果我们的软件要使用 0，1024，2048 这些地址存放数据，那么在同一时刻只有一个数据能够进入 cache。 145

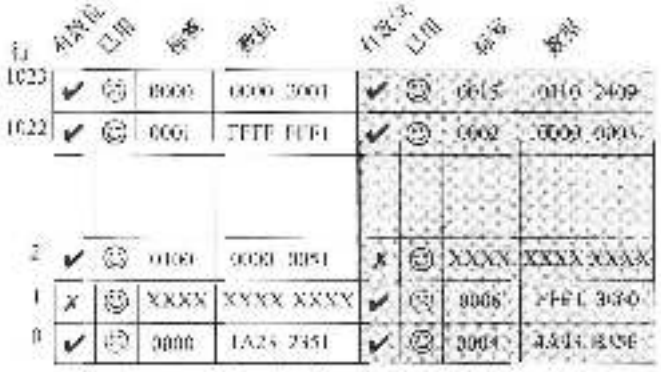
为了解决类似的问题，一种 n 路的组相联 cache（set-associative cache）允许 n 个 cache 行和存储器行相关联。其本质就是 n 个直接相联 cache 以并行的方式操作。

在一个 2 路的组相联 cache 中，可以有两个 cache 位置与某个存储器地址相联（详见框 4.6）。从这种方式的 cache 中读取数据，其过程仍非常快，等同于连接了两个查找表。这项技术已经被广泛采用，例如最早的 DEC StrongARM 处理器，包含了一个 32 路的组相联 cache。

就像所有的 cache 一样，一个新的位置在载入数据之前必须要退回之前的数据。问题是，n 路的 cache 如何退回？这与 MMU 的例子中的问题相似，在 4.4.4 节中也将介绍相关的退回算法。

框4.6 组相联 cache 实例

下图表示了在一个微型计算机系统中 2 路组相联 cache 正在使用的情况。



这个 cache 有两行，4.5 中的直接 cache 极为相似的组织。但是每一行有两个条目（因为是三路组相联）。有 1024 行（对应于虚拟系统的 10 位）。笑脸符号表示有效，哭脸符号表示无效。

在系统的开始阶段，所有的 cache 条目都是无效的，就像第 1 行的两个条目和第 2 行的所有一样。这可能意味着这些条目在系统重置后就没有被写过。

直接相联 cache 和组相联 cache 有效的不同之处可以从第 0 行看出。对第 0 号的查找请求就像 4.4.1 节中的一样完全一样。但是在这个例子中，相同的请求可以同时从两个路中的第 0 页读取数据。也就是说有两个条目，其标志位显示为已经正确，这意味着这个数据已经被写入到 RAM 中。当前的 32 位数据为 0x4A93 B35F，对存储地址（0x004 << 10）+0 = 0x1000 的数据有效的缓存

4.4.3 全相联 cache

146
3
147

如果我们的软件经常使用 0，1024 和 2048 这 3 个地址，而不用 1、1025 和 2049，那么直接相联 cache 与组相联 cache 的第 0 行将会一直进行输入输出交换，满负荷运转。相反，cache 的第 1 行将会空置。

全相联 cache（full-associative cache）解决了该问题，因为这种方法可以允许任何的存储器位置映射到任意的 cache 位置上。在这种情况下，标签位就包含了全部的存储器地址信息。

问题是，当 cache 从存储器中读取数据时，都必须检查 cache 中的每个标签。换句话说，对 cache 中的每一行都要进行检查。在直接相联 cache 中，只需要检查一个标签。而在 n 路组相联 cache 中也只需要检查 n 个标签。

所以，尽管全相联 cache 方法可以获得很好的命中/缺失比率，但是这种方法会由于其上有很多检查操作而导致开销较大。这个问题类似于 MMU 中的物理 RAM 占用表问题。

4.4.4 局部性原则

被加载或回载的实际存储模式依赖于高速缓存的使用情况。但在一般情况下，在运行一些典型程序的计算机中，有两个良定义的特征：数据存储器和程序存储器。这就引出了计算机体系结构中一个众所周知的术语：局部性原理（principle of locality）。实际上局部性原则有两个：第一个是空间局部性（spatial locality），它涉及地址的紧密性；第二个是时间局部性（temporal locality），它涉及时间的紧密性。

通过一种可视化方法就可以发现这个原则，查看计算机内存并给过去几千个时钟周期里使用过的数据做上色。如果在操作过程中冻结计算机，你可能发现一些已着色的内存地址紧密以及大量当前未使用的内存。几秒钟后再次冻结，你会发现另一些活跃着的不同的内存集群。一个好的 cache 操作应该尝试把尽可能多的着色集群放到快速缓存中，从而加快程序的平均执行时间。

如果把可视化方法应用到程序存储器，就会发现一些连续着色内存块像缎带一样通过内存流淌。

空间局部性原则是指在任何一个时刻，活跃的项目可能在内存地址上是彼此相邻的。对程序存储器来说这是由于程序指令的连续性，而对数据存储器来说这是由于编译器会把定义的变量集群到相同的内存段中。

时间局部性原则是指一个近期被访问过的项目相比于其他位置的项目更可能被再次访问。对程序存储器来说可以通过循环结构来解释，而对数据存储器则可以通过整个程序中重复使用一些变量来解释。

图4-17举例说明了两个局部性原则，图中演示了3个内存页，给出了它们在程序运行过程中的几个瞬间快照。内存使用的密度通过页中不同深浅的矩形块显示出来。内存地址通过在矩形页中的位置表示出来。可以看出随着时间的推移，时间局部性原则导致不同内存集群之间的逐渐移动。空间局部性意味着内存的访问往往集中在一起。请注意，跨多个页存储的变量（或者堆栈项目）在任何时刻都可能变得有效。这是因为不同类型的项目可以停驻在不同的页面（特别是，数据项和程序项不可能共享一个内存页）。 148

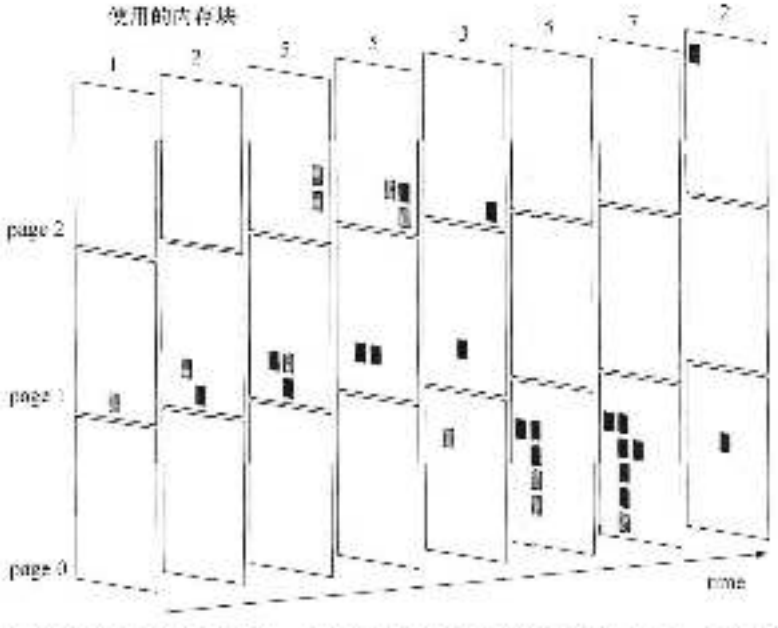


图4-17 空间局部性和时间局部性原理的一个实例，显示了内存使用（用在几个内存页上涂黑的内存块来表示）是如何随时间变化的。时间局部性：前一个时刻有效的内存集群往往在下一个时刻还要使用，但与更晚些时间里用到的集群却不同。空间局部性：有效内存集群驻留在每一页的特定区域中，而不是均匀分散的。页中任何时刻使用的块的数目标记在图的上方

局部性的含义是，在一般情况下大致预测将来可能访问的内存位置是有可能的。一个好的高速缓存的功能就是利用这个信息来缓存这些位置，从而增加平均访问速度。

4.4.5 cache 替换算法

替换算法用于在运行的 cache 中记录位置。当请求一个新位置而高速缓存中适当的部分已经满了时，替换算法便会发挥作用，这意味着 cache 中一些位置要被新的位置替换。如果 cache 中适当的位置被标记为“dirty”（换句话说，它已经被重写但还没有被保存回 RAM），那么必须先将数据保存到 RAM。相比之下，“clean”的缓存条目可以马上更换，因为只需让它们存放与 RAM 中的缓存位置相同的值即可。当然，谁成为适当的位置是缓存组织的功能：全相联 cache 将不限制位置，但直接相联 cache 或纠和联 cache 会限制一个内存地址可以被缓存在哪一 149

行（或哪几行）。

然而，仍然存在一个问题，如果刚被送回 RAM 的某一行在很短的时间后又需要被访问，就不得不再把它装载回去。而这可能需要更多的数据退回操作，是一个耗时的过程。

一个好的 cache 能最大限度地减少加载和卸载的次数，或者以另一种方式，最大限度地提高命中率。要做到这一点，一种方式是确保正确的数据（定义为最不常用数据）被收回，而这正是 cache 替换算法的工作。有几个值得一提的常用算法：

- LRU（最近最少使用） 其规模复杂性与 cache 大小相关，因为它需要维持每个条目的使用顺序的清单。下一个要收回的条目就是清单底部的那一项。LRU 在大多数情况下可以合理地执行。
- FIFO（先进先出） 替换 cache 中最先进来的位置。这在硬件上很容易实现，因为每个加载行标识符只进入一个 FIFO，当需要收回某一项时，FIFO 输出的标识符就是下一个被替换项。在某些内存位置重复使用很长时间而另外一些内存位置仅仅使用很短时间的情况下，FIFO 不如 LRU 有效。
- LFU（最不常使用） 替换最不常用的位置。这很难实现，因为每个 cache 条目需要有计数器，而且需要有比较所有计数器的电路。然而，在大多数情况下，LFU 性能非常好。
- *随机算法* 用硬件实现很容易，只需挑一个（伪）随机位置。令人惊讶的是，这种技术实际上执行得相当好。
- *循环算法* 采取轮流收回缓存行的方法。这种算法在 n 路组相联 cache 中很常见，n 路的每一路轮流被收回。它的主要优点在于便于实施，但对较小的 cache 性能很差。

记住，cache 一定要快，因为这些算法需要记录哪些行已被访问以及当需要替换时替换哪一行，它们需要以一种不会限制 cache 性能的方式实现，如果把缓存的速度减慢得和主存储器一样，那么再完美的算法也是没有用的。这些算法需要用快速的硬件而不是软件实现。因此，实现的复杂性是一个问题。

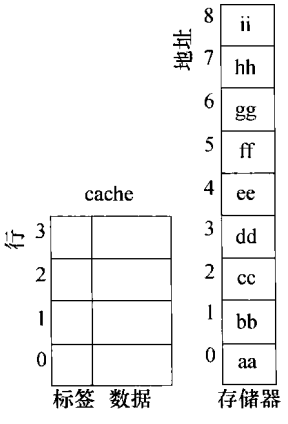
150 框 4.7 和框 4.8 列举了一些 cache 替换算法如何操作读写序列的例子。

框 4.7 cache 替换算法示例 1

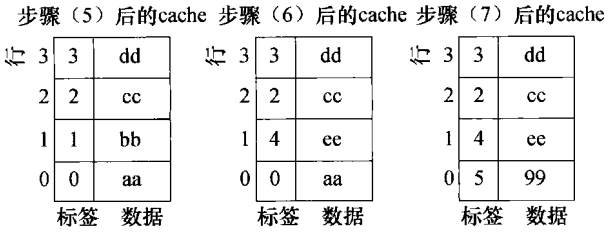
问：计算机系统 cache 和主存储器状态如下面右图所示。复位时，cache 完全是空的，但主存储器位置填满了 aa、bb、cc 直到 ii，每个缓存行可以缓存一个内存地址。

如果采用写回系统的 LRU 替换算法，而且 cache 是全相联的（从底往上填），追踪需要采取的行为，画出经过下面的操作序列后 cache 的最终状态：

（1）从地址 0 读；
（2）从地址 1 读；
（3）从地址 0 读；
（4）从地址 2 读；
（5）从地址 3 读；
（6）从地址 4 读；
（7）把 99 写入地址 5。.



答：我们将研究每一步操作，并画出在步骤（5）、（6）、（7）后 cache 填满后的状态图，如下图所示：



首先，因为 cache 是空的，所以步骤（1）缺失。因此从内存中检索到值 aa 并放置在 cache 第 0 行上，标签为 0（因为全相联 cache 标签是完整的内存地址）。步骤（2）同样缺失，这将导致 bb 被放置在第 1 行。步骤（3）命中——地址 0 已经在第 0 行中了，所以不需要采取进一步行动。步骤（4）缺失，导致 cc 被写入第 2 行。步骤（5）同样缺失，这将导致在 cache 第 3 行填入 dd。

此时 cache 已满，所以增加任何新的条目都会要求收回某一条目。由于我们使用 LRU（最近最少使用），因此需要记录最后一次访问每个条目的时间。步骤（6）缺失，所以内存中位置 4 的值需要加载到 cache 中。往回看，最近最少使用的是步骤（2）中的第 1 行而不是步骤（1）中的第 0 行，因为加载了第 1 行后我们在步骤（3）又访问了第 0 行。所以，步骤（6）把内存地址 4 中的数据 ee 存到第 1 行。

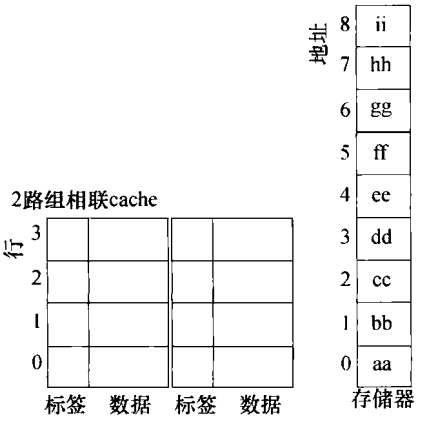
最后，步骤（7）涉及从 CPU 写入内存。由于我们有一个写回系统，这个值必须同时放到 cache 和主存储器中。再次应用 LRU 算法，我们可以看到第 0 行是这次最近最不常用的位置，因此，它要被新的数据取代（它没有被退回，因为从它被载入起我们还没有对它进行写入操作）。

框 4.8 cache 替换算法示例 2

问：计算机系统的 cache 和主存储器如下面右图所示。

复位时，cache 是空的，但主存储器位置已填满了 aa、bb、cc 直到 ii。每个 cache 行可以容纳两个内存地址（换句话说，它是一个 2 路组相联 cache）。如果采用写回系统的 FIFO 算法，追踪需要采取的行动，画出经过下面操作的序列后 cache 的最终状态：

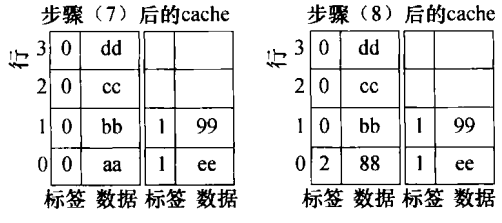
（1）从地址 0 读；
（2）从地址 1 读；
（3）从地址 0 读；
（4）从地址 2 读；
（5）从地址 3 读；
（6）从地址 4 读；
（7）把 99 写到地址 5；
（8）把 88 写到地址 8。



答：首先，确定标签范围是很重要的。由于 cache 有四行，内存地址 {0~3} 停驻在标签区域 0，{4~7} 在标签区域 1，{8~11} 在标签区域 2，以此类推。内存地址 0、4 和 8 映射到第 0 行，地址 1、5、9 映射到第 1 行，以此类推。

现在研究每一步操作。步骤（1）引起缺失并导致 aa 被加载到 cache 第 0 行。为了增加可读性，我们将先填左半边。步骤（2）同样缺失，填入第 1 行。步骤（3）命中，导致 cache 第 0 行左边的值被读出。步骤（4）和步骤（5）也是缺失，将 cc 和 dd 分别填入第 2 行和第 3 行。此时，cache 左边的每一行都填满了。所以步骤（6）从地址 4 读缺失将导致 ee 被放到 cache 中。地址 4 映射到 cache 第 0 行并且由于第 0 行左半边已经满了，所以它被写到了右半边。注意地址 4 在标签区域 1 中。

步骤（7）是对地址 5 写入，地址 5 映射到 cache 第 1 行，标签为 1。我们还没有访问过地址 5，所以是缺失，并将导致数据 99 被放到 cache 第 1 行余下的部分，即右半边。此时的 cache 状态图如下图左边所示。



最后的步骤（8）是把 88 写到地址 8。地址 8 映射到 cache 第 0 行并在标签区域 2 中。它必定被放到 cache 中，因为有一个写回正在使用。然而，cache 第 0 行已满，所以有一项需要收回。采用 FIFO 方案，先进先出。对于第 0 行的情况，先进来的是左半边的位置，所以它被 88 替换，如上图右边所示。

4.4.6 cache 性能

命中所花费的时间相当于检查是否命中（访问 cache 查找表）加上从 cache 中检索数据以及返回到请求数据的 CPU 所需的时间。这是以假设更新替换算法的运行时间部分并不会增加总花费时间为前提的。由于 cache 顾名思义就是要快，所以检查是否命中花费的时间应尽可能得少。

缺失所花费的时间稍微有些复杂。首先需要时间来测试是否命中（访问 cache 查找表），然后运行替换算法，然后查询选定的 cache 行的 dirty 标志位。如果该标志位有效，则必须将把不想要的数据退回主 RAM 的时间也添加进来，再加上把需要的数据从主 RAM 加载到 cache 上所需的时间，从 cache 中检索数据到 CPU 所需的时间也要列入考虑因素。

如果 cache 位置 M_1 命中的访问时间是 T_1，但是在 cache 缺失的时候我们需要把字 M_2 从主存储器传送到 cache M_1，传输时间为 T_2，那么命中率 H = cache 命中数/总请求数，平均访问时间 T_S 可以表示为：

$$T_S = H \times T_1 + (1 - H)(T_1 + T_2) = T_1 + (1 - H)T_2$$

151
~
153

由于 T_1 比 T_2 小得多（当然命中比缺失快很多），因此要想使平均访问时间接近 T_1，则需要高命中率（换句话说，尽量达到 $H \approx 1$）。

如果 C_1 是大小为 S_1 的 cache 每位的成本，C_2 是大小为 S_2 的主存储器每位的成本，则每位的平均成本可以表示为：

$$C_S = (C_1S_1 + C_2S_2)/(S_1 + S_2) = C_1S_1/(S_1 + S_2) + C_2S_2/(S_1 + S_2)$$

考虑到 C_1 远大于 C_2，则 cache 不得不非常小，否则就会过于昂贵。cache 设计就是成本、速度和大小之间的权衡（考虑大小是因为低级别的 cache 通常不得不安装到 CPU 所在的同一个硅芯片上，分享了宝贵的空间）。

定义访问效率为：$T_1/T_S = 1/\{1 + (1 - H)(T_2/T_1)\}$，可以认为是命中率为 1 时的理论最大速度除以前面推导的实际平均访问速度。T_1/T_S 访问效率的一些典型值期望的命中率在框 4.9 中给出。

框 4.9 访问效率示例

T_1/T_S 访问效率的一些典型值对应的命中率如下所示：

H	5	10	20
0.6	0.33	0.20	0.11
0.8	0.50	0.33	0.20
0.9	0.67	0.50	0.33

这些都是一些真正 CPU 中的典型数字：在内存为 16MHz 的 75MHz ARM7 中，其 T_2/T_1 的值逼近 5，可以达到 0.75 的命中率（有一个很好的 cache 贯穿于整个良性的或可预见的程序执行）。其他拥有更快 cache 的系统会有更高的命中率。在多级 cache 的情况下，可以反复分析达到 T_3 和 T_4 等的比率。当然，如果将正在执行的程序全部放入 cache 中，命中率将达到 1.0。

请注意，拥有一个大缓存的系统并不少见。这是一些数字信号处理器采用的有效方法：大容量、非常快的单周期内部 RAM 使得 CPU 可以全速运行其操作而不用等待访问内存。一个流行的例子就是 ADI 公司的 ADSP2181 芯片有 80KB 的高速片上存储器。在这种情况下，用户都愿意花大价钱拥有与 CPU 紧挨在一起的大 RAM 块，以满足性能需要（所有操作——包括内存访问——在一个周期内完成）。

有各种改善 cache 性能的技术，例如预测性预读和自适应替换算法。好的全相联 cache 可以
154
提供高达 0.9 的命中率，尽管这可能是在一个专门的系统中并且只对小规模的程序才能实现。

4.4.7 cache 一致性

cache 一致性是保证 cache 中一个内存位置的所有副本都具有相同的值。在迄今为止给出的例子中我们通过简单地指定 clean/dirty 以及有效/无效标志来解决这个问题。cache 一致性在共享存储多处理器系统中是很重要的。然而，维保 cache 一致性特别困难。

想象一下一个共享变量被两个 CPU（A 和 B）使用的情况。如果它被两个 CPU 上读取，它将最终在两个 CPU 上都进行缓存。现在，如果两个 CPU 之一，比如 A，改变了变量（通过写入），存储在 CPU A 的 cache 中的变量将被更新。在内写系统中，变量的新值会立即写回到内存，所以内存中的值也是最新的。然而，CPU B 的 cache 中仍然是变量的旧值。如果 CPU B 读取这个变量，将 cache 命中并会使用 cache 中的旧值，而不是 RAM 中最新的正确值。CPU B 现在读取不正确变量的问题称为一致性（coherency）问题：CPU B 的 cache 的变量与该变量存储在其他地方的值不一致。

一个并行计算机系统的例子如图 4-18 所示，它可以被扩展成更多的处理器。由于 CPU 和共享总线带宽，它将往往成为性能瓶颈。为此将独立 cache 做得很大以尽可能减少访问共享 RAM（因此也减少了总线的使用）。然而，这只能加剧一致性问题。

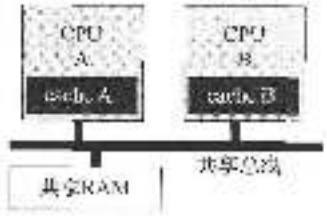


图 4-18 此图给出了两个 CPU，每个 CPU 有一个独立 cache，通过共享总线和被写共享主 RAM 相连

现代计算机系统中使用了很多技术来解决这个问题。有一个常用的解决方法我们先它叫做侦听（snooping）。侦听就是 cache "听"共享总线上其他 cache 访问的过程。这可以提供两个有用的信息：第一个是何时另一个 cache 读取的位置已在本地 cache 中；第二个是何时另一个 cache 写回到内存的位置也在本来 cache 中。

使用收集来的信息，智能 cache 控制器可以采取某种形式的行动来防止一致性问题的发生。例如，当其他的 cache 写回到共享 RAM 的某个位置时，使相应的本地 cache 条目无效。事实上，有许多处理这个问题的方法，而 MESI 协议是其中最受欢迎的方法之一。

MESI 协议以它的几个状态 modified（修改）、exclusive（独占）、shared（共享）和 invalid（无效）的首字母命名，它基于如图 4-19 所示的几个状态机。在图中，读缺失后，（S）或者（E） 155

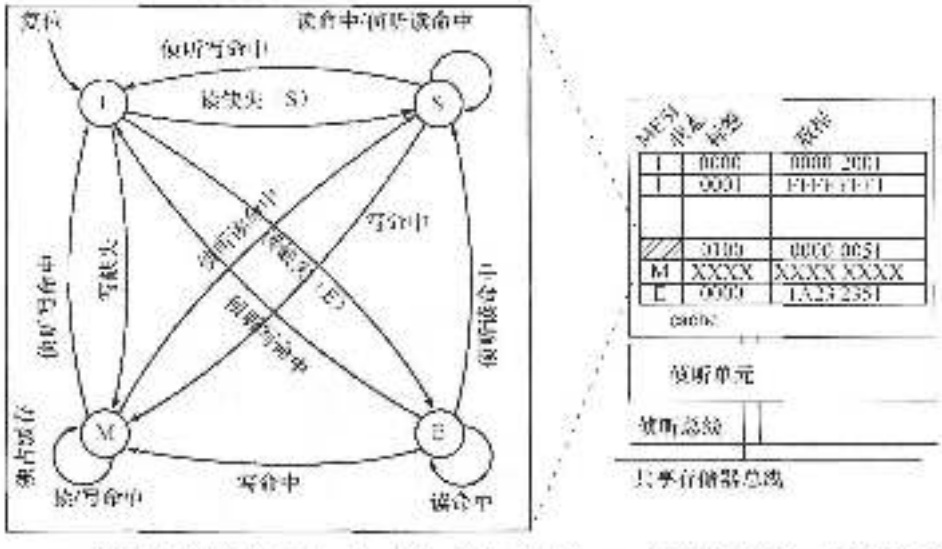


图 4-19 MESI 协议状态转换图（上）以及缓存一部分来显示 MESI 状态协议位于哪个特定的缓存行

表明当值从主存储器取出时，另一个 cache 侦听单元表示它也保持着一个副本（因而用 S 表示共享），或者没有其他侦听单元表示正在使用它（因而用 E 表示独占）。因此可以看出，侦听单元不仅仅负责侦听其他 cache 的访问，而且当它们自己缓存着其他 cache 请求的值时也要通知其他处理器的 cache。

每个缓存行可以有与之相关的 4 个状态之一（而不是有效/无效和已用/清空两个状态）：

- I 是无效状态，表明那一行的数据不正确或者没有缓存任何东西。
- S 是共享状态，意味着其他 CPU 可能也缓存着这个值。cache 可以通过侦听共享存储总线来做出决定。cache 中的值与主存储器中的值相同。
- M 是修改状态，发生在值已被更新时。意味着其他保持着这个值的 cache 实际上保持的是旧值。
- E 是独占状态，表明目前没有其他 cache 保持这个值，它与主存储器中的值相同。

如果这个方案应用于共享存储的多处理器系统，那么每个 CPU 有它自己的 cache 并且每个 cache 使用 MESI 协议进行控制。cache 中的每一行仍需要通常的行号和标签，但是有效/清空标
156 志由两个标志位取代，即表示无效、修改、独占或共享 4 个状态。

复位时，所有缓存行被设置为无效，这意味着缓存行中任何数据都是不正确的。

读者可以参考框 4.10 中给出的 MESI 协议在双处理器共享存储系统中运行的例子。

框 4.10 MESI 协议示例

为了说明 MESI 协议在双处理器共享存储系统中的运作，我们将通过典型事件序列来跟踪系统状态。CPU 被命名为 A 和 B，它们的 cache 从复位开始（所以所有条目从状态 I 开始）。

CPU A 从共享内存中位置 X 读取。由于 cache 全部无效，因此这将是一个读缺失并导致值从主存储器中检索。cache B 将侦听总线，发现有数据传送并从内部看到自己没有缓存位置 X。因此它将保持安静。观察状态图并应用于 cache A，I 状态没有侦听信息的读缺失将会导致进入 E 状态。

想象一下，CPU B 随后也读取位置 X，而 cache B 中没有内容，因此读缺失。cache B 从共享 RAM 中读取值，而 cache A 侦听总线。cache A 从内部看到自己也缓存了位置 X，于是 cache A 在侦听总线上告知 cache B 它保持着位置 X。cache B 将继续读取值，但由于它是共享读取，状态图告诉我们沿着从状态 I 到状态 S 的读缺失（S）线走。同样，cache A 内部侦听读命中，所以缓存行保持位置 X 的状态从 E 移动到 S。这时，两个 cache 都保持有位置 X 并且两个都处于共享状态。

接下来，想象 CPU A 向位置 X 写。根据直写策略（任何写操作都直接到主存储器），cache A 意识到写命中，行状态从 S 移动至 E。cache B 侦听单元正在监视总线并确定一个侦听写命中。由于它也在状态 S，这会使它转换到状态 I，也就意味着无效。这是正确的，因为它缓存的值已经不是最新的值——最新的值在其他 cache 中并且现在返回到了主存储器。

4.5 协处理器

某些计算任务用硬件可以更好地执行，这种硬件不属于标准 CPU 的范畴。一个常见的例子就是浮点数的处理，使用专用的浮点运算单元通常比用 CPU 更快（早期的个人计算机硬件中不提供浮点运算：一些读者可能还记得 Intel 80386PC 母板上的插座——用于接插 Intel 80387 浮点协处理器和其他替代品）。事实上，从最早的计算机开始，就有专用硬件用来执行某些特定的功
157 能，它与 CPU 是分开的，而 CPU 负责执行通用计算。

除浮点数处理之外，也许最突出的例子是 Intel 的 MMX 扩展到奔腾处理器的范畴，后来继续扩展并命名为 SIMD 流指令扩展（Streaming SIMD Extension，SSE）。然而，还有其他的——许多现代嵌入式处理器包含专用协处理器单元来执行特定的功能，例如加密、音频或视频处理，甚至专用的输入输出处理。

我们将在后面的4.7节研究MMX和SSE，但就目前而言，我们将考虑最突出的例子——浮点运算单元。这是每一个现代台式计算机都会包含的部分，内置在它们的CPU中，但在为嵌入式系统设计的处理器中会较少看到它们。

4.6 浮点运算单元

我们已在第2章讲过，浮点数通过使用尾数和指数来为特定的基本系统传输数字信息。正如我们前面介绍的，IEEE 754标准浮点数是目前最常用的表示方法，在计算机行业内广泛应用。

由于这种标准化，实现这个标准的设备并不像使用标准的计算机系统的其他部分那样变化频繁。例如，Intel 80486[⊖]和奔腾处理器包含一个片上FPU，它与80387一样，从20世纪80年代中期其原始版本出现开始，基本上保持不变。它是80386的独立的协处理器芯片。那时，买台式PC可以选择带或不带板上FPU，大多数没有FPU的计算机可以通过购买芯片并把它插到主板的空槽上进行升级，正如前面介绍过的那样。

不支持浮点运算是有原因的（现在仍然是），这要归因于FPU的性质：硅片面积大并且耗电多。特别是对嵌入式和电池供电系统，使用没有浮点运算的处理器往往是首选，所有算法用定点运算实现，或者使用更高级的语言以及借用软件浮点模拟器。

使用FPU时，CPU把操作数加载到主CPU和FPU共享的特殊寄存器里（它或者是一个单独的芯片，或者在同一硅片上）。通过发送一条特别的指令激活FPU。然后FPU从共享寄存器读并开始处理所需的指令。一段时间后，FPU返回结果到特殊寄存器并通过中断通知CPU进程已经完成。许多现代处理器在执行流水线中都包含FPU，因此不需要额外的中断（流水线将在5.2节介绍）。

FPU通常不能直接访问内存中或者共享总线上的数据。它只能作为一个从处理器，去操作那些由主CPU加载到通用共享寄存器中的数据。这些寄存器足够长，可以容纳多个IEEE 754双精 158
度数，虽然内部使用扩展的中间格式（见2.9.3节）。

在更近的586及以上级别的处理器中这些寄存器与MMX单元或其派生的SSE家族（见4.7节）共享，这意味着主CPU加载数据到寄存器，然后激活MMX或者FPU。所以在许多586级别的处理器中，MMX和浮点运算单元不能一起使用，程序员在任何特定的时间不得不选择两个模式中的一种。

FFU或MMX的局限性导致了AMD 3DNow!的发展。这一包含21条新指令的扩展能有效地让AMD处理器在同一代码块中交错使用浮点指令和MMX指令。而这促使Intel公司开发出了SSE，我们将在4.7节把它作为协处理器的另一个例子做进一步的讨论。ARM FPU作为一种替代方法，框4.11介绍了它的发展。

框4.11 一个替代方法：ARM处理器上的FPU

注意ARM工程师采用的另一种浮点运算单元设计方法，Steve Furber在《ARM System Architecture》一书中这样描述：

工程师们首先调查大量的常用软件以找到什么类型的浮点运算最常使用。采用RISC设计方法，他们在硅片上设计了FPA 10这一ARM浮点协处理器来实现这些最常见的指令。

FPA 10有4个阶段的流水线，使它在每个周期处理操作数并同时执行4个计算。不太常见的指令或者纯粹用定点软件执行，或者用部分定点软件与浮点FPA 10指令相结合的方法执行。

⊖ 某些486系列处理器没有浮点运算能力，尤其是那些面向低功耗应用的处理器。

4.6.1 浮点仿真

正如我们看到的那样，FPU 是一个可以操作浮点数的设备。通常情况下，它提供标准的算术、逻辑、比较以及乘法功能。通常也支持除法及其他更专门的操作（如四舍五入）。大多数 FPU 符合 IEEE 754 标准，它定义了它们的操作、精度等。

每当高级语言程序员在他们的程序中使用浮点数据类型时，就会访问 FPU。例如，在 C 程序语言中，这些类型几乎全是非常我们已经在 3.4.1 节定义过的。

- float（单精度）——一个 32 位单精度浮点数包括符号位、8 位指数以及 23 位尾数。

159

- double（双精度）——一个 64 位双精度浮点数包括符号位、11 位指数以及 52 位尾数。

在 C 语言中有一个进一步的浮点数据类型，拥有比双精度类型更高的精度，那就是 long double。然而，long double 精度似乎不太标准（正如 3.4.1 节简要提到的那样），它的范围分布从与 double 相同，到 IEEE 754 扩展中间格式（见 2.9.3 节），最多能达到真正的四精度数。

然而，尽管“浮点”通常意味着要遵守 IEEE 754 标准，但这也不是必需的。如 3.4.5.2 节所述，这只有当底层提供的硬件与 IEEE 754 兼容时才支持。在一些功耗和尺寸更值钱的嵌入式系统中，设计师作出了务实的选择，用略低于 IEEE 754 的精度来提供浮点运算。从程序员的角度来看，float 和 double 数据类型仍然存在，但使用它们进行计算的准确性可能有所不同。

在没有支持浮点运算的硬件的地方，或者说在没有 FPU 的地方，指定浮点操作的指令将被 CPU 识别出来，引起中断（或捕获——见 3.4.5 节）并由专门的代码来处理。代替 FPU 的代码称为浮点仿真器（FPE）。

很多时候，FPE 代码在精度方面是 IEEE 754 的替代品。使用多个定点指令计算 IEEE 754 操作会花费非常多的时间，这是一个速度和精确度之间的权衡。通常设计者更热衷于速度。

这种权衡的另一个方面在图 4-20 中进行了说明，其中显示了有硬件 FPU 的处理器和一个定点处理器，在两者上执行相同的代码。在所有其他因素都一样（即在第一种情况下两者之间的唯一区别是 FPU 协处理器存在与否）的理想情况中，有 FPU 的处理器可以把浮点操作送给 FPU，它进行操作时耗能显著，而主 CPU 执行其他无关的功能。一旦完成浮点运算，结果被传回 CPU，操作继续进行。

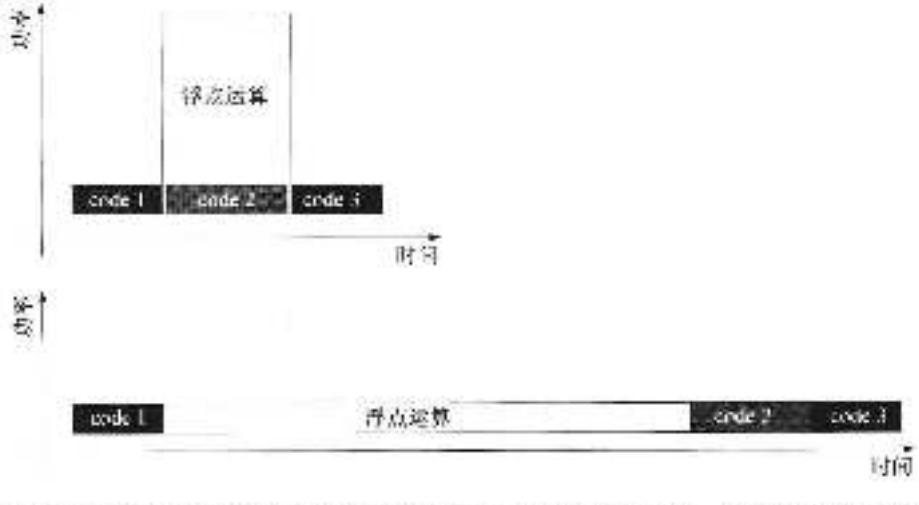


图 4-20 两种设计的权衡示意图：（i）在专门硬件 FPU 上执行浮点运算，固定点代码继续在主 CPU 上执行（上方的图）；（ii）使用 FPE 代码执行浮点运算，它花时更长，但耗能更低（下方的图）

在定点处理器的情况下，FPE 代码仿真浮点运算运行在主 CPU 上。由于这种情况下没有协处理器，浮点代码就不可能与其他代码并行执行。显然，程序将执行得更慢，即使 FPE 代码可以像 FPU 那样快。不过，通常 FPE 是分几次执行的，可能 10 次或者更多，比在 FPU 上执行得慢。

在能量消耗方面——在需要考虑电池寿命的便携式电子产品中的一个重要度量——能量（功率×时间）在图中阴影部分显示。虽然 FPU 比定点 CPU 耗能明显要多，但它可以在更短的时间内完成[⊖]，因此它很可能比浮点仿真的能量效率更高。当然，正如我们前面所介绍的，在这 [160] 种情况下系统工程师可以决定采用低精度浮点程序以加速计算。对程序员来说更可取的是避免使用浮点运算，这往往是一个嵌入式系统开发者的目标。程序员可以考虑使用长整数或者小数（Q 格式）表示法编程（见 2.3.7 节）。

4.7 SIMD 流指令扩展（SSE）和多媒体扩展

多媒体扩展（multimedia extensions，MMX）是 Intel 给奔腾处理器的硬件多媒体协处理器的名称。MMX 单元实际上是一个 SIMD（单指令多数据）机，正如 2.1.1 节定义过的那样。在使用时，一组数据被加载到 MMX 寄存器组，然后一个 MMX 指令可以并行地在每个寄存器上操作数据。这种处理的一个例子是两个地方的 8 个整数同时右移，或者 4 个寄存器的值与另外 4 个寄存器值相加，并且前 4 个寄存器保存计算结果。这一主题有许多变化，但重点在于多个独立的操作由单一的指令引发，并同时发生。

Intel 发布了 MMX 后，竞争对手 Cyrix 和 AMD 很快为他们的设备提供了类似的加速器，而其他厂家如 ARM 和 Sun 也为他们的 RISC CPU 建造了定制设计的同等设备。这些硬件设备是片上提 [161] 供的，而不是作为外部协处理器，这源于对多媒体数据往往涉及大量数据的相对简单算术操作的重复应用这一考虑。

4.7.1 多媒体扩展（MMX）

这种类型处理的一个例子是 MMX 技术用来适应显示屏上一个区域的色彩调整。如果屏幕上显示数据的每个像素是一个字节或者字，那么调整颜色也许仅仅是给每个字添加一个固定值，或者可能是一个逻辑掩码操作。无论确切的操作是什么，大量的像素必须重复同样的操作，可能是 1280×1024 像素或者更多。如果在标准 CPU 上执行，将有 1280×1024 = 1 300 000 个重复累加。

加入了 MMX 单元，CPU 就可以把数据块加载到 MMX 单元，然后同时对数据块里的所有数据项执行算术运算。同时，CPU 本身是空闲的，可以执行其他的操作。很容易看到如果 MMX 单元有 16 个数据项，那么处理所有像素所需的时间可以减少到 1/16 左右。

4.7.2 MMX 实现

对 MMX 扩展的论证是有说服力的，尤其鉴于在 MMX 发展的几年里个人计算机对多媒体处理需求的增长。然而，相关的问题也被提了出来，那就是如何找到最好的实现这种处理的方式以及支持什么类型的处理。

在 Intel 奔腾处理器中，问题的实现主要在于 Intel 需要任何新的奔腾处理器向后兼容，与早

⊖ 这是假设当 FPU 不再计算时保持关闭，因而不产生功耗。遗憾的是，实际中这个假设往往不是真的。

期的8088以及更早的DOS上使用的16位软件，甚至是与一些令人惊讶的微软Windows的现代版本兼容。因此，通过改变x86 CPU的指令集来扩大它的功能的可能性非常小，否则新的软件不能在旧机器上运行，这会令消费者不高兴（这种兼容性改变需要逐步完成，获得消费者的理解需要时间）。此外，从一个奔腾版本到下一个，寄存器数目不能突然增加，因为这会使旧软件中使用的上下文保存和恢复过程无效。

不过，Intel的工程师发现了两个聪明的方法来实现他们的目标。第一个方法是给奔腾一个额外的指令，可将处理器设在MMX模式（他们发布了简单的代码，让程序员首先检查MMX的能力，然后在有MMX的机器上运行一个代码版本，而在没有MMX的机器上运行另一个）。在MMX模式，额外的57条新指令对MMX处理可用。旧软件不能使用这种模式，因此不会遇到额外的指令。Intel工程师的第二个创新是重复使用浮点运算单元寄存器来保存MMX数据。在正常模式下，这些寄存器由FPU使用，但在MMX模式下，它们被用于MMX处理。

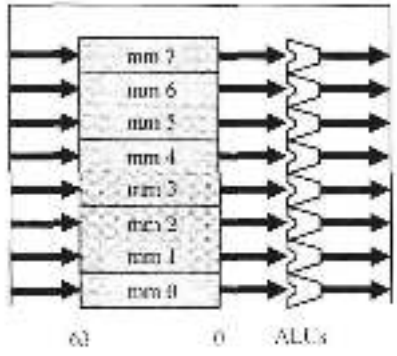


图4-21 具有MMX功能的Intel CPU的MMX寄存器、进行功能单元（看起来像小ALU）以及总线连接示意图

162

遗憾的是，程序员并不采用MMX全集。也有一些针对选择MMX模式完全去除浮点运算功能的批评（在第4.6节曾提到）。最终，源于SSE的灵感，导致了AMD 3DNow!的产生。然而，在我们讨论SSE（见4.7.4节）之前，让我们来看看这些系统一般情况下是如何工作的，先从MMX开始。

如图4-21所示为MMX单元的逻辑结构，包括它的8个寄存器（不过需要指出这个图是高度形式化的——真实的MMX比画在这里的要复杂得多）。注意总线从8个ALU块的输出反馈回寄存器。这是MMX单元内部结构的简单表示，但足以说明每个寄存器路径的并行性。每一行都是一个单独的总线。

在MMX模式，有8个64位宽的寄存器（为什么是64位？还记得表示双精度浮点数需要64位吗？双精度浮点数在FPU模式下通常保存在这些寄存器中）。指令是并行执行的，并且除了加载和存储指令外，指令都是在寄存器间进行的。

每个寄存器都是64位大小，它可以容纳8字节或4个16位字或2个32位双字或1个64位四字。这些都是在程序员的控制下进行的，为创建MMX代码带来了极大的灵活性。

它支持算术、逻辑、比较和转换操作。这些操作可以应用到传输寄存器内的任何已知位宽的数据。当然，加载正确大小的数据以及选择合适的操作应用于该数据是程序员的责任。

4.7.3 MMX的使用

为了在配置合适的奔腾处理器上使用MMX的功能，首先要检查CPU是否可以进入MMX模式（有一个简单的向后兼容机制可以做到这一点）。如果可以，MMX模式处理就可以继续；否则，必须借助CPU的能力提供执行相同功能的代码。显然，这会慢很多，但在每一个可移植的程序中，向后兼容性都是需要的。

163

然而，对于特殊的程序使用这种技术所获得的速度提升是很显著的，现实生活中图像处理对MMX能力的测试表明：在Linux环境下，MMX优化代码比测试软件中没有MMX代码要快至少14倍。

4.7.4　SIMD 流指令扩展（SSE）

MMX 实际上是 Intel 对 x86 指令集上单指令流多数据流（SIMD）扩展的专有命名，最初于 1997 年推出。AMD 推出了他们的硬件扩展，并命名为 3DNow!，在 Intel 只支持整数的硬件上增加了浮点运算的功能。互不示弱，Intel 各种 SIMD 流指令扩展（Streaming SIMD Extension，SSE）与 AMD 的 3DNow! 之间的斗争愈演愈烈。

SSE 为 SIMD 的数据处理提供了 70 条新指令以及 8 个新的 128 位寄存器[⊖]。这些寄存器可以存放一般的整数值，但现在当然也允许使用浮点数：

- 4 个 32 位整数
- 8 个 16 位短整数
- 16 字节或字符
- 2 个 64 位双精度浮点数
- 4 个 32 位单精度浮点数

SSE 实际上已经有了相当大的变化，从最初的版本到 SSE2、SSE3、SSE4 以及最近的 SSE5。每次更迭都带来需要程序员学习的新功能、新指令。有趣的是，从 SSE4 往后，Intel 就不再继续支持使用旧的 MMX 寄存器了。

SSE4 指令集引入了一些快速的字符串处理操作，也有许多的浮点操作，例如并行乘法、内积、四舍五入等。现在 Intel 和 AMD 的版本之间也有一定程度的兼容性（可能比之前 x86 处理器时代兼容更多），但是功能的不断革新，加上一些积极的营销策略，使得对这两个 x86 式处理器功能的直接比较变得十分困难。

4.7.5　使用 SSE 和 MMX

有这么多的版本，在不同的 CPU 之间有不同的兼容性，更不用说制造商之间了，软件工具往往落后于硬件功能的发展。许多编译器默认不支持这些协处理器，或充其量在可能的硬件功能范围内提供少量的支持（宁愿限定只支持最常用的选项）。虽然近些年情况有显著的改善，尤 164
其 Intel 自己提供了可以大概支持这些扩展的编译器，但编程工具并不倾向于充分利用这个专用硬件。

此外，需要给各种不同的处理器写几个专用的代码版本，这意味着 SIMD 扩展指令的使用往往只限于专业软件实例，而不是商业操作系统和应用的一般发行。然而，它们的存在和使用，尤其是在台式机和服务器中，拥有绝对最高的处理性能。

4.8　嵌入式系统中的协处理

今天很少有嵌入式系统使用 x86 式处理器，尽管也有低功耗的变形，例如 Atom 处理器。到目前为止，占比例最大的是基于 ARM 的处理器或者使用类似的低功耗 RISC CPU。即使是 x86 处理器，也很少有完整的 SSE 功能（因为这些协处理器以其高耗能而著称）。然而，它们确实在台式机和服务器系统应用上有优势，原因在于与台式机系统可以运行任何软件相比，许多嵌入式系统只运行控制或专用软件。台式机系统需要软件向后兼容（如在根本没有扩展的情况下，对 SSE 的代码、MMX 的代码、SSE4 的代码以及裸 x86 代码的要求），而在嵌入式系统中，程序员事先确切地知道哪些硬件可用并能适当地开发自己的软件。

⊖ 在 64 位模式时会翻番到 16 个 128 位寄存器。

反过来也是如此——了解要运行什么软件也给修改或创建自定义硬件提供了机会。作为这个过程的一个例证，在4.6节我们介绍过FPA 10，它是主要的ARM浮点协处理器，它的设计是基于对最常见软件需求的分析。

除了已经提到的FPU和MMX/SSE外，在嵌入式系统中还可以看到许多其他的协处理器。考虑下面的ARM特定的协处理器：

- **Jazelle**——这个名字似乎是将Java语言中的“J”加到Gazelle（羚羊）上，让人联想到迅速而敏捷地执行Java代码的情景，而这正是目的所在。设计Jazelle的ARM工程师们创建了一个硬件单元，能够不需要中断而直接处理许多Java指令（字节码），这带来了速度和效率方面的改进。Java的分支（BXJ）指令进入Jazelle处理，使得CPU能自然地执行大多数常见的字节码（并捕获其余部分以在优化的软件程序中执行）。
- **NEON，先进的SIMD**——与Intel的SSE类似，这是一个拥有非常完备指令集的64位或128位SIMD扩展，它能够并行执行成批的整数和浮点数操作。这也许正是SSE本来应该的样子，如果它从零开始，作为一个现代处理器从底向上进行设计（而不是把MMX添加到30岁的半向后兼容的硅片上）。
- **VFP**——作为ARM处理器的向量协处理器，它增强了浮点运算的功能（VFP的全称为vector floating point，即向量浮点）。它用于矩阵和向量计算，即对数据数组的重复序列操作。

165

还记得早在3.2.6节我们讨论过的RISC和CISC处理器的不同原理吗？CISC处理器在进化的过程中被描述为臃肿而笨重，它把越来越多的功能包含到单个CPU指令中。相比之下，RISC精简而快捷。

RISC指令往往很简单，但很快速。其论据在于即使做任何有用的事需要更多的指令，这些指令也可以执行的更快，从而整体性能与CISC方法相比有所提升。然而，协处理器的使用使得RISC处理器（小、精简、快速）把特定的计算任务交给单独的处理单元。因此，CISC处理器提供的一些特定于应用的指令也可以交给RISC协处理器单元来处理。

回想起Intel为早期MMX使用的双模式，一个进一步的改进涉及可重构协处理器。它允许调整协处理器使用的硅资源以适应任何特定时间的计算需要。显而易见，可重构是有代价的——它将耗费时间和能耗。然而，一些复杂运算对快速处理的需求很容易使这些代价物有所值。

对嵌入式系统设计者来说，可能最好的例子是现场可编程门阵列（FPGA）。FPGA中的“软核”处理器是用像Verilog这样的高级硬件描述语言编写的。事实上，我们将在后面第8章开发一个这样的处理器。现在我们需要考虑的FPGA的首要特征之一是它们的可重构性。已有的许多免费的和商业的软核实现了协处理器接口，一些研究人员已经尝试把可重构处理单元附加到它们之上。人们很可能会继续探索这些方法对于嵌入式系统的重要性，并且必然会越来越多地采用它们。

4.9 小结

本章研究了现在的通用微处理器的一般内部元素，包括通过内部总线与ALU、FPU或其他协处理器和加速单元等不同的功能单元之间传输数据的方式。

系统内有内存管理单元和cache，可以把它们看做在处理器核和外部存储系统之间的地址与数据总线上的过渡区。通过预测未来的内存召回模式以及存储过去的一些与所预测的未来访问相匹配的内存访问，cache可以加速平均内存访问时间。同时，内存管理单元扮演两个重要的角色：第一个是允许使用虚拟内存，这扩展了处理器能使用的地址范围和存储空间；第二个是允许定义和使用内存段和页——一个很重要的好处就是运行中进程间的内存保护（阻止某些进程覆

166

盖其他进程或内核的私有存储，从而防止或者至少是减少崩溃的可能性）。但是使用虚拟内存需要付出代价：它往往会降低平均内存访问时间。

本章内容是现代 CPU 中通常会实现的标准功能单元和通用处理器的功能。在第 5 章中，我们将把注意力转向性能改善——一些常见的加速技术。我们将会看到，在 CPU 制造商一味追求或越来越快或功耗越来越低的过程中（但这两个特点很少同时具备），出现了一些有趣的方法，并且必将得到应用。 167

思考题

4.1 4.2.2 节谈到了 ALU 设计，如果每个逻辑门的输入和输出之间有 10ns 的传输延迟，那么 ALU 的工作频率最大能达到多少？

4.2 参照问题 4.1 考虑 2 位的 ALU：

a. 4 个这样的 ALU 怎样结合成一个 8 位的 ALU（针对无符号数）？

b. 要处理有符号补码，你将如何修改设计？

4.3 下面的伪代码段在一个 RISC 处理器上执行：

```
loop i = 0,1
read X from memory address 0
read Y from memory address i
Z = X + Y
write Z to memory address i+1
```

该处理器需要一个周期来完成所有内部操作（包括 cache 访问）。从 cache 把数据保存到 RAM 需要 4 个周期，从 RAM 加载数据到 cache 需要 4 个周期（加上 1 个周期继续从 cache 到 CPU）。

假设该系统采用直接相联 cache，初始化为空。如果 cache 使用下面的策略，此段代码需要多少个周期？

a. 写回。

b. 无写分配的直写（WTNWA）。

c. 有写分配的直写（WTWA）。

4.4 你有一台小冯·诺依曼计算机，它的数据 cache 可以在 2 路组关联和直接映射之间切换。它有 512 个缓存行，每行可以容纳一个数据字，所有的数据传输以字为单位。在处理器上运行如下算法：

```
define data area A from address 0 to 1023
define data area B from address 1024 to 2047
set R0 = 512, R1 = address 0, R2 = address 1024

{
lp [R1]= R0+R0     ;save to address stored in R1
   [R2]=[R1-1]+[R1]
   R1 = R1+1
   R2 = R2+1
   R0 = R0-1
   if R0>0 then goto lp
}
```

168

a. 如果系统采用写回协议，哪一种 cache 结构最好？

b. 写出 3 种 cache 替换算法并评价它们的硬件复杂性。

c. 给出的算法在清空 cache 复位后运行并且迭代两次。如果系统采用使用直写（和写分配）的直接相联 cache，CPU 到 cache 之间的传输需要 10ns，cache 到 RAM 之间的传输需要 50ns，回答下面的问题：

i. 命中率是多少？

ii. 两次读代码的平均访问时间是多少？

4.5 写出前一个问题的算法以提高命中率。（提示：可数数据区定义，它不是循环代码。）

4.6 一个先进的MP3/照片播放器使用显示内存。从而能使CPU访问1GB的物理存储空间，与系统只有1MB的RAM。操作系统设置MMU的页面宏为4KiB大小。字节宽度的RAM有90ns的访问时间。可使用JDL接口最高以2.2MiB/s的速度传输数据。RISC CPU使用32位指令。

a. 同时能有多少页驻在RAM中？

b. MMU到RAM之间的地址总线应当有多少条线？

c. 从RAM中读每条指令需要多长时间？

4.7 使用问题4.6中的信息，计算从光盘到RAM（或从RAM到光盘）加载一个页需要多少时间。使用该答案来确定CPU从被我同行内存页中读取一个指令的两个可能的时间

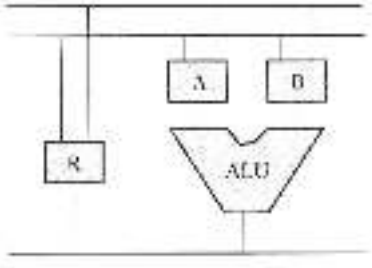
4.8 前面问题一的MMU到RAM之间的地址总线不够宽，不足以容纳更多的内存器。写出3种克服此总线大小限制以及在物理接口上连接更多内存的方法（硬件或软件）。

4.9 一个双处理器的机器有一个共享的存储块和一条单面总线。每个处理器模块中的写回cache实现MESI协议。所有缓存行从无效（I）状态开始。

通过以下事件跟踪cache状态（X、Y和Z不相等）：

1. CPU1读取RAM地址X
2. CPU1写入地址X
3. CPU2读取地址Y
4. CPU1读取地址Y
5. CPU1写入地址Y
6. CPU2读取地址X
7. CPU2读取地址Z
8. CPU1写入地址Z

4.10 考虑如下框图，在一个三总线CPU中连接了一个ALU单元和3个寄存器。假设除了一个与每个总线相连的接口以外，这个框图是完整的，并且存储器中数据传输至寄存器1数据的转移要很多。



a. 在两个时钟头指示每个总线连接中允许的数据流向。

b. X + Y这个操作的效率如何？

c. X = X这个操作的效率如何？

d. (X = Y) + Z这个操作的效率如何？

e. 给出另外一个可以提高效率的连接图。

4.11 给出每个ALU单元能实现的两个主要算术操作和4个基本的逻辑操作（移位操作除外）。

4.12 给出在最简单的CPU中能实现的3种不同类型的或者说移动方向不同的操作，除了循环（rotate）指令以外。（你能解释为什么只要求3种类型而不是4种吗？）

4.13 假设在4.1中的传输延迟的例子，假定8位ADD操作和8位AND条件的传输延迟。在每种情况中，假设功能选择信号都是正确的（不变（这样它们就不会影响计时）。如果这个ALU单元是单周期的，那么这个设备的最大时钟速度是多少？

4.14 如果说cache可以提高处理速度，那么你能想出制造商为什么不将其集成了大型片上缓存块的集成电路

的原因吗？ 170

4.15 一个包含直接相联 cache 的计算机系统，它命中的访问时间为 10ns，未命中的访问时间为 120ns。当命中率为 0.3 时，计算其平均访问时间。

4.16 假设问题 4.15 中那台计算机的设计者想要提高性能，他们只能改变系统中的 3 个部件（考虑到每个改变都要花钱，他们只想改变其中的某一个部件，选择其中最好的一个）。确定下面哪项可以最好地改善这个系统的平均访问时间：

a. 装一个更快的主存储器，其访问时间为 100ns。

b. 装一个更快的 cache，其访问时间为 8ns。

c. 在一个更大的 cache 中采用更好的排列方法和更聪明的替换算法将命中率提高到 0.4。

4.17 假设一个小型的 16 位嵌入式系统主要用来处理整数代码，但是有时需要快速地处理一块浮点数据的代码，这可以用专门的在 FPE 中执行的 FPU 来处理，也可以将代码转换成大型整数来处理。讨论对于解决方案中是否应该包含 FPU，应该考虑的主要因素都有哪些。

4.18 第 3 章中已经介绍了相对寻址的概念。简单讨论这与空间和时间局部性原理（见 4.4.4 节）的关联。

4.19 在有关 cache 的部分，“有写分配的直写”（WTWA）是什么意思？它与“无写分配的直写”（WTNWA）有什么不同？在一个输出大量临时的图像数据给内存映射显示的系统中哪个更合适？

4.20 在一个配有经过全面开发和调试的软件套件的嵌入式系统中，一个有经验的程序员在调试一段时不时出问题的代码段时在 RAM 的 0x000 位置设置了一个观察点[⊖]。但是你的代码、数据和变量都在存储器中的另外某个位置，你当然没有固定哪个变量或者代码在内存 0x000 的位置。你能理解为什么别人应该关注这个内存位置，即便这个位置可能永远也不会用到吗？ 171

⊖ 观察点（watchpoint）是内存中的一个位置，调试软件将不断地监测，当这个位置的内容变化时停止程序执行。

第5章
Computer Architecture: An Embedded Approach

提高 CPU 性能

很少有读者完全按顺序来学习像本书这样的书，这是可以理解的。我本人总是鼓励学生能从不同的教材中选择合适自己的，这些教材因作者而异，描述不同的问题会用不同的方法，对于不同读者来说总会有更合适的（这就是为什么图书馆会存在了）。而有的读者喜欢按顺序学习不同章节，对于这样的读者，恭喜你们顺序地读到了这里。我希望读者能够在脑海中呈现出计算机设计进化过程的图像。所有需要的模块都按功能集合成一个可以工作的 CPU 并将被评估，限制模块性能的部分将被调整或加速。小幅度的加速很普遍，贯穿整个设计；相反，真正的革命性改变却很少。在大多数情况下，设计的改进源于性能的需要，而从根本上说，是源于市场的需求。比如在嵌入式系统中，功耗影响着电池的寿命，是一个显著的因素，因此通常总能找到很多理由采用解决功耗问题的革新技术而不是采用性能提高技术。

每个人都希望获得更高速的计算机。有人曾说，在信息的高速路上是没有速度限制的。在大多数情况下，用户的直觉是高速意味着节约更多的时间（然而本书作者十分质疑这种观点：现在实验室中的学生在使用高速的计算机，却比上一代使用相对较慢计算机的学生浪费了更多的时间）。对于嵌入式系统，特别是那些需要实时处理的嵌入式系统，更快的计算速度无疑能够带来更高的性能。然而对于台式机却有这样的质疑，大部分计算机速度的提高，内存、外存容量的增长都被软件开发人员给消耗掉了，他们开发的软件需要大量的存储，特别是对于操作系统。尽管如此，性能目标依旧是计算机工业中最主要的驱动因素，并且已经催生出了许多伟大的解决
172 方案。在本章中，我们将对一些主流的性能提高方法进行探讨。

5.1 CPU 加速技术简介

对于第四代计算机来说，时钟是影响性能的主要方法，人们总是想方设法把它弄得更快。但这会导致一系列问题，如发热和昂贵的处理器价格，因为目前处理器生产工艺已经接近当前技术的极限，再往上提高将变得越发困难。

部分处理器设计人员已经把目光转向其他方法，比如 RISC 处理器的出现和逐渐壮大。一些公司主要致力于增加处理器字长，从 4 位到 8 位、16 位、一直到 32 位。近期的一些设计已经达到了 64 位和 128 位，甚至出现了 1024 位的系统（更多内容详见第 9 章）。

现在的技术不仅单纯提高时钟速度，而是更侧重于在每个时钟周期做更多的事情，这导致了并行和流水线的出现（或是两种技术同时使用）。

Sun 公司则在他们的 Java 处理器上采用了一种不同的技术，这种技术重新运用了 CISC 处理器的思想，但这会是从软件的角度（这种方法极好地将堆栈和 RISC 处理器整合到一起）。目前，PicoJava 及类似的处理器都是为 Java 语言量身定制的，而不像其他大多数处理器那样通过对语言进行翻译来解释并在处理器上执行。这种以软件为本的设计方法在商业上取得了一定成功，或许可以预示这种设计方法的时代将要到来。

本章主要涉及目前处理器设计领域所采用主流设计思想和方法，这些思想和方法更多是出于商业利益而不是学术上的考虑，总的目的就是让消费者尽可能快速地获得更快和更便宜的产品。我们从最重要、最常见的流水线技术开始。

5.2 流水线

虽然流水线的出现有时候更应该归功于现代工业制造技术而不是计算机的发展，但它能够提高处理器的指令吞吐率（throughput）而不是缩短每条指令的执行时间（事实上缩短每条指令的执行时间能够在未来更多的性能提高）。这种技术是将每条指令的处理划分为多个阶段，从而能够同时处理多条指令，进而获得整体吞吐率提高。

吞吐率是指每秒执行操作的数目。在3.5.2节中是指每条指令所需的时钟周期数。吞吐率度量方法比每条指令需要多长时间完成更重要。为了证明这一点，我们来考虑一条典型处理器指令处理流程，如图5-1所示。 [173]

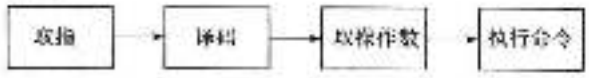


图5-1 一个简单CPU的四级指令处理流程图

在这个例子中，每条指令分4个阶段完成处理，其中需要强调的是，每个阶段的持续时间均为1个时钟周期。每条指令需要一次通过所有4个阶段来完成处理，因此需要4个时钟周期。

对于一个非流水线的机器，取到并处理一条指令需要在前一个指令处理完成之后才能开始。我们通过以下预约表（reservation table）来展示：

取指	指令1				指令2				指令3
译码		指令1				指令2			
取操作数			指令1				指令2		
执行指令				指令1				指令2	
时钟周期	1	2	3	4	5	6	7	8	9

表的左边是处理指令所需的不同功能单元，底部是时钟周期。表中给出了在每个时钟周期处理器所进行的工作。这个表展示了连续9个时钟周期处理器所进行的工作。

在第一个时钟周期处理器取到指令1，然后依次进行译码、取操作数、最后执行指令所表示的功能。完成以上步骤，指令2才开始它的处理流程。

然而从另一个角度考虑这个预约表：假如我们把列当做存在的资源，把行当做时序（time slot），可以清楚地发现每个资源在大部分的时间上什么也没做。如果能够将多条指令重叠执行，让每个资源在更多的时间里有事情做，执行效率将会提高很多。接下来试试我们的想法：

取指	指令1	指令2	指令3	指令4	指令5	指令6	指令7	指令8	指令9
译码		指令1	指令2	指令3	指令4	指令5	指令6	指令7	指令8
取操作数			指令1	指令2	指令3	指令4	指令5	指令6	指令7
执行指令				指令1	指令2	指令3	指令4	指令5	指令6
时钟周期	1	2	3	4	5	6	7	8	9

这种方法最显著的效果是原本在第9个时钟周期才开始指令3，现在通过重叠执行，包含了9条指令，这样指令速度为原来的3倍。这种方法并没有提高时钟频率或者改变处理顺序，只是让指令重叠执行。 [174]

这种重叠执行叫做流水线，已经被几乎所有现代处理器所采用来提高执行速度。这种技术的控制单元变得越来越复杂，但与在速度上获得的提高相比不值一提。请参考框5.1中的分析。

框5.1 流水线加速比

流水线速度有两个指标：加速比和吞吐率。为了说明这两个指标，我们考虑一个含有n条指令的程序，每条指令需要s个时钟周期完成处理。

在非流水线处理器上，这个程序的执行时间为$n \times s$个时钟周期。

现在我们将这个处理器划分为s个流水阶段，完成每个阶段需要1个时钟周期。现在，需要多长时间来执行这个程序呢？

第1条指令需要s个时钟周期来完成。在随后的每个时钟周期将会完成1条指令，所以总的执行时间为$s+(n-1)$个时钟周期。

加速比是指一个流水线处理器的执行时间和等效的非流水线处理器的执行时间之比：

$$S_p = \frac{ns}{s+n-1}$$

可以证明，当$n \to \infty$时，$S_p \to s$。这意味着程序越长，它的流水线加速比（大致）等于流水线级数越多，我们希望流水线有尽可能多的级数（这里的级数一般等于一条指令的周期数）。但实际上，流水线的开销和额外设备的引入使得不可能无限增加流水线级数。

衡量另一个度量指标——吞吐率，它是指流水线在单位时间内所完成的指令数。吞吐率是指令执行的速率，因此流水线处理器的吞吐率为：

$$T_p = \frac{n}{s+n-1}$$

这个公式有点类似于加速比公式，实际上$T_p = S_p/n$。关注吞吐率是有意义的，在上一章我们曾经使用过每条指令的周期数（CPI）这个指标。

下面将讨论更多流水线所带来的难题，但首先我们先来了解一下不同类型的流水线。

5.2.1 多功能流水线

流水线并不一定是单一功能的，它可以处理不同类型的指令，即多功能流水线。事实上这很
175 平常，只是会增加控制的复杂度。考虑如图5-2所示的一个例子。

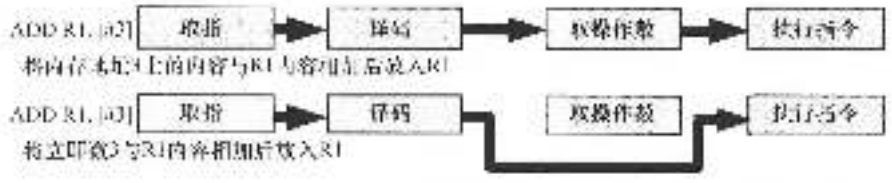


图5-2 在一个简单四级流水线CPU上处理两条不同的指令的流程图。第1条指令使用了流水线中的每一级，而第2条流水线因为不需要从内存里取操作数，因此跳过了第3级。这说明了多功能流水线的概念，即根据指令的不同需要按不同的方式处理指令

在图5-2上部所示的流水线中，第1条指令需要从内存中取出内容，所以它需要“取操作数”这个单元。在下部所示的相同流水线显示刚启动了一条不同指令。这条指令不需要取操作数（因为立即数3作为指令的一部分已经存在于处理器中），所以在这个条件下不需要“取操作数”这个单元。然而，这并不意味着流水线会跳过这一级因而使得第2条指令能够更快地完成。考虑以下预约表，表中两条指令顺序执行。

取指	ADD R1, [#5]	ADD R1, #3	指令3	指令4	指令5	指令6
译码		ADD R1, [#5]	ADD R1, #3	指令3	指令4	指令5
取操作数			ADD R1, [#5]	NOP	指令3	指令4
执行指令				ADD R1, [#3]	ADD R1, #3	指令3
时钟周期	1	2	3	4	5	6

在第4个时钟周期，第2条指令被标为“NOP”（无操作）。但处理器却不能立即从译码跳到执行阶段，因为在第4个时钟周期，处在执行阶段的硬件仍在处理前一条指令（ADD R1，[#3]）。

这还阐明了一个有趣的观点：流水线需要满足不同类型指令的需求，而且受限于最慢的指令。对于非流水线处理器，简单指令可以快速地处理，复杂指令会慢一些。但对于流水线处理器，所有的指令都需要相同的时间来处理，除非是采用了更先进的技术。

设计人员需要小心对待流水线，至关重要的一点是要让流水线的所有单元在大多数时间内保持工作，然而我们却在预约表中加入了“NOP”。“NOP”意味着这个时钟周期无操作或资源浪费。因此为了最小化这种浪费时隙，需要调查所有指令的需求和指令的出现频率。 176

5.2.2 动态流水线

在多功能流水线概念的基础上，动态流水线并不是简单地绕过没用的功能，而是根据正在处理的指令和处理器状态，允许选择不同的流水线执行路径。

图5-3展示了一个虚构的例子，它拥有4个未命名的流水线单元（T_1到T_4），处理着3条不同的指令，每条指令经过了流水线不同的路径。然而这种流水线复杂的选择控制部件和用来为跳跃单元的指令（例如指令3绕过了流水线的T_2单元）进行降速的延迟部件在图中并没有展现出来。

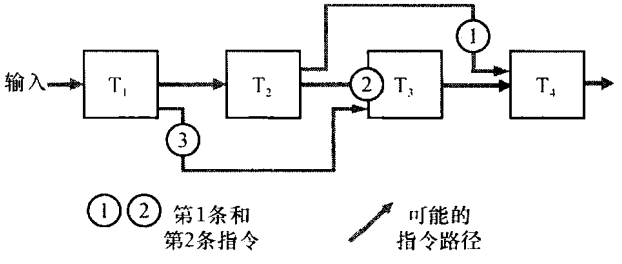


图5-3 动态流水线示意图，它允许不同指令根据其执行的需要通过流水线不同的路径

延迟部件也必须是动态的，它们只在需要保证指令按顺序到达流水线各单元的时候被激活。例如，指令3为了跟在指令2后，需要延迟一个时钟周期以避免在T_3单元与指令2发生冲突。反之，指令1需要绕过流水线的T_3单元，但是并不需要保持跟在任何指令后边，因此不需要延迟。

感兴趣的读者可能会注意到，很多处理器会足够智能地为自己决定哪些指令需要通过流水线上的各单元并按顺序执行，哪些指令可以乱序执行从而避免过度的延迟。

5.2.3 改变流水线模式

到目前为止，所有的讨论都假设每条经过流水线的指令是不相关的，并且每一条指令都能在之前的指令结束后进入流水线。 177

显然，这种假设并不总是正确。我们将考虑3种情况，它们将在本节和接下来的两节中对流水线的操作产生影响。

首先，现在很多处理器都会发生模式改变，这是通过接收一条模式改变指令来触发的，将会改变随后到来指令的处理方式。例如：

1. 在ARM处理器中，指令集替换（16位Thumb指令集替换原有的32位ARM指令集）。

2. 在大多数处理中（包括ARM处理器），大小尾端模式转换。先前的指令存储为小尾端模式，在改变大小尾端模式后，后来的指令存储为大尾端模式。

3. 对于一些定点DSP处理器如TMS320系列，改变算数模式来开启或关闭符号扩展，从而影响后续指令的执行。

虽然这些模式指令会被使用，但是并不频繁。例如对于前两种指令，只会在程序开始时可能执行；而第三种指令只在每个算术运算块执行一次。

对于这种极少执行的指令，大多数处理器会在接收到这种指令后清空流水线，这意味着所

有已经进入流水线的指令都将被丢弃而流水线会重新开始执行。这种方法虽然看起来十分极端，但是对于电路逻辑却十分简单。同时，这种方法虽然对流水线效率影响很大，但是由于在大多数程序里极少出现，所以对处理器的性能几乎没有影响。

下面的预约表展示了接收到这种模式改变指令（ChM）时流水线的情况。从表中可以看到，指令3、4、5都已经进入流水线，在接收到模式改变指令后，这些指令被丢弃，处理器在第6个时钟周期改变执行模式，然后这些指令被重新取指。

取指	指令1	ChM	指令3	指令4	指令5	X	指令3	指令4	指令5
译码		指令1	ChM	指令3	指令4	X		指令3	指令4
取操作数			指令1	ChM	指令3	X			指令3
执行				指令1	ChM	X			
时钟周期	1	2	3	4	5	6	7	8	9

这个预约表可以是以下指令序列的执行结果：

```
指令1：ADD  R0, R0, R1
指令2：MODE big_endian
指令3：SUB  R4, R1, R0
指令4：NOP
指令5：NOP
```

178 指令6：NOP

其中，指令3、4、5以大尾端模式译码（虽然在编译器的助记符中并没有体现出来，但是通过查看存储这一块程序的内存是可以看到的）。

一旦流水线的模式改变，则需要清空流水线，后续的指令将被重新取指。

在比较新的处理器中，这种操作由处理器自动执行；但对于比较旧的处理器，这种操作不会自动执行，而需要由编译器来完成（甚至是由编程人员手动修改汇编代码来完成）。在上面的例子中，只需对程序进行简单修改就可以完成模式改变时流水线清空的工作，这是通过改变指令的顺序来完成的：

```
指令1：ADD  R0, R0, R1
指令2：MODE big_endian
指令3：NOP
指令4：NOP
指令5：NOP
指令6：SUB  R4, R1, R0
```

换句话说，只需在模式改变指令后插入一系列的NOP指令即可，这是因为NOP指令不管使用大尾端模式读取还是小尾端模式读取，对于译码的结果都是一样的。例如，指令0x0000和指令0xFFFF反过来读也是0x0000和0xFFFF，使得当发生模式改变时，指令译码不管从哪个方向读指令，译码的结果都是一样的。

5.2.4 数据相关冒险

与改变处理器模式会引发处理器运行的一系列问题一样，处理器内部寄存器和内存存储位置的变化也会对程序的运行造成影响。考虑以下指令序列，它们就会引发一些问题：

```
ADD  R0, R2, R1     ; R0 = R2 + R1
AND  R1, R0, #2     ; R1 = R0 AND 2
```

从以上的指令序列可以看到，第二条指令依赖第一条指令的结果，R0需要先被写回才能够被第二条指令正确读取。但在流水线中，这会引发问题。我们看如图5-4所示最简单的五级流水线。

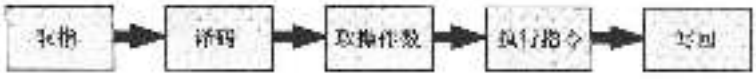


图5-4 五级流水线示意图

179

这条流水线的主要特点是，流水线末尾额外增加了一级写回（store result），而不管运算是否发生。增加这一级首先是为了简单数据相关这个问题，其次大部分处理器中确实也包含着这一级。

下面的这个预约表是上述两条指令在这个流水线中执行的情况，需要注意的是，在每个时隙中都有寄存器R0内容的标示：

取指	ADD R0	AND R1				
译码		ADD R0	AND R1			
取操作数			ADD R0	AND R1		
执行指令				ADD R0	AND R1	
写回					ADD R0	AND R1
时间周期	1	2	3	4	5	6
R0	X	X	X	X	R2 + R1	R2 + R1

这里需要注意的是AND这条指令，需要R0作为它的一个操作数（R1 = R0 AND 2），而这个"取操作数"是流水线的第三级（如表中黑体字所示）。在这个例子中，第二条指令的取操作数发生在第4个时钟周期，但是第一条指令在第5个时钟周期才将结果写回寄存器R0。如此，第二条指令将会从R0取回一个不正确的值。

这种现象称为RAW（read after write，先读后写）冒险（hazard），按照程序意图，寄存器R0应该是在被写之后才被读，但在上述情况中则是在被写回前被第二条指令读取了。

如果仔细观察上述例子，还存在另一个冒险。在这个例子中，寄存器R1存在WAR（write after read，先写后读）反相关性。第一条指令需要读R1，第二条指令则需要写R1，这个冒险就是要保证第二条指令在写R1前第一条指令必须已经对R1读取完毕。在上述流水线中，这种冒险不会发生，但是对于现在一些比较先进的带有乱序执行（out-of-order）的动态流水线中，这种冒险是存在的。

还有一种冒险是WAW（write after write，写后写）冒险，例子在练5.2中给出。

练5.2 WAW冒险

这种冒险的基本条件比较简单，即有两条或更多条指令对同一寄存器进行写操作，而第二条指令则需要从这个地址读取新值发生。在某些乱序执行处理器中，这种冒险可能会导致一个问题：

```
ADD  R0, R2, R1     ; R0 = R2 + R1
AND  R1, R0, #2     ; R1 = R0 AND 2
SUB  R0, R3, #1     ; R0 = R3 - 1
```

在这个例子中，R0存在WAW冒险。只要条件允许，我们就可以使第二条指令的取操作数发生在第一条指令写回完成后和第二条指令写回完成之前。

需要注意的是，在这一段指令之后，R0存放的是最后的结果，所以第一条指令对R0的写只是一个暂存。这种数据可以通过对数据存放的寄存器或者通过数据旁路（在5.2.10节中讲到）来实现。与RAW不同的是，WAW冒险可以在流水线中发生，因为RAW与写回速度比读取速度慢。现在这种冒险主要与cache或存储器相关，因为不在我们考虑之列。

5.2.5 条件冒险

给定一些在一定条件下才能够执行的指令，问题是在什么时候检查这些条件以确定指令是否应该执行。以下给出一个程序段例子：

```
ADDS  R0, R2, R1     ; R0 = R2 + R1，并设置条件标志位
ANDEQ  R1, R0, #2    ; 如果零标志位被置1，R1 = R0 AND 2
```

之前提到过的ARM处理器中，如果在指令的末尾带“S”则表示指令的执行结果需要更新条件标志位（如零标志位（zero flag）、负数标志位（negative flag）、进位标志位（carry flag）以及溢出标志位（overflow flag），这些标志都存放在ARM处理器的CPSR寄存器中，框5.3对几种常见的条件标志位进行了讨论）。第2条指令的执行是带有条件的，“EQ”指出了这条指令当且仅当先前指令的执行结果为0（在这个例子中是当且仅当R0为0）时才执行。

接下来，我们根据以上例子的指令填写预约表：

取指	ADDS R0	ANDEQ R1	指令3	指令4		
译码		ADDS R0	ANDEQ R1	指令3		
取操作数			ADDS R0	ANDEQ R1		
执行指令				ADDS R0		
写回						
时钟周期	1	2	3	4	5	6
NZCV	0000	0000	0000	0000		

框5.3 条件标志位

虽然不同的处理器对寄存器的命名有所区别，但是在目前市场上大部分处理器中，通常还是可以看到如下的条件标志位：

N（负数标志位）：最后一条条件设置指令的结果为负数；

Z（零标志位）：最后一条条件设置指令的结果为0；

C（进位标志位）：最后一条条件设置指令的结果产生了进位；

V（溢出标志位）：最后一条条件设置指令的结果产生溢出。

在下表中我们将给出一些影响这些标志位发生变化的指令例子。根据指令末尾是否带有“S”来决定指令的结果是否影响条件标志位。

指令	操作	N	Z	C	V
MOV R0, #0	R0 = 0	0	0	0	0
MOV R1, #2	R1 = 2	0	0	0	0
SUBS R2, R1, R1	R2 = R1 - R1，结果为0	0	1	0	0
SUBS R3, R0, R1	R3 = R0 - R1，结果为负，0xFFFFFFFD	1	0	0	0
SUB R2, R1, R1	R2 = R1 - R1，结果为0，但不带“S”	1	0	0	0
ADDS R4, R1, R1	R4 = R1 + R1，结果为正，0x4	0	0	0	0
ADDS R5, R4, R3	R5 = R4 + R3，0x4 + 0xFFFFFFFD	0	0	1	0
MOV R8, #0x7FFFFFFF	最大的正32位有符号数	0	0	0	0
ADDS R9, R8, R1	R9 = R8 + R1，结果为0x80000001	0	0	0	1

值得注意的是，0一般被当成正数而不是负数来处理。而进位标志位和溢出标志位的使用对于将操作数当做有符号数或无符号数也是有区别的，如果处理的是有符号数，则溢出标志位十分重要，而如果处理的是无符号数，则需要考虑进位标志位。更多讨论可以参考2.4节的内容。

在第4个时钟周期末尾，第1条指令已经执行并且条件标志位被更新。注意到第2条指令此时

已经进入流水线，虽然此时并不知道这条指令是否应该执行，但是还是允许它进入流水线并且塞流水线直到第1条指令执行完毕。在一些处理器中，会使用预测执行（speculative execution）来加载和处理第2条指令。一旦条件标志位确定，就会决定是否中断第2条指令或让它继续执行。

180 / 182

我们继续填写预约表，同时假设第1条指令结果不为0，则第2条指令不需要执行（或者即使执行但是结果也将被丢弃）。

取指	ADDS R0	ANDEQ R1	指令3	指令4	指令5	指令6
译码		ADDS R0	ANDEQ R1	指令3	指令4	指令5
主操作数			ADDS R0	ANDEQ R1	指令3	指令4
执行指令				ADDS R0	X	指令3
写回					ADDS R0	X
时钟周期	1	2	3	4	5	6
XREV	0000	0000	0000	0000	0000	0000

由于零标志位在第5个时钟周期没有被置1，因此移除第2条指令并以NOP指令代替。这导致了预约表中一条对角线资源的浪费。相较而言，如果流水线在等待至第5个时钟周期第1条指令完成后再取下一条指令，则将会浪费3条对角线的资源。

至此，读者应该会有疑问：流水线还需要什么样的额外部件才能够支持预测执行？对于预测执行，在指5.5中会有简要的说明，深入讨论将等到5.7节进行。

5.2.6 条件分支

ARM处理器有一个指令集，里边的大部分指令都是带有条件操作的。然而，对于大多数处理器来说，它们的条件操作只有条件分支，用来改变程序的指令流。以下是一个条件分支的例子：

```
loop:   MOV R1, #5          ;R1=5
        AND R4, R3, R1      ;R4=R3 AND R1
        SUBS R2, R0, R1     ;R2=R0-R1
        BGT loop            ;如果结果为正则分支
        NOT R3, R4
```

其中重要的一行为BGT（如果条件标志位大于0则分支）和它之前设置这个条件标志位的行。很明显，在SUBS指令结束并且更新条件标志位以前，无法知道分支是否应该发生。

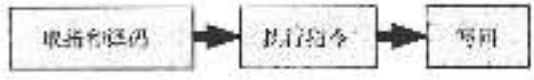


图5-5 一条简单的三级流水线流程图。该流水线将取指和译码放在同一阶段并且没有专用于取操作数的阶段

183

假设上述程序在一个只有三级的流水线上执行，如图5-5所示。

我们使用这条流水线在以下预约表中"执行"操作序列（到分支处为止）：

取指和译码	MOV	AND	SUBS	BGT					
执行指令		MOV	AND	SUBS	BGT				
写回			MOV	AND	SUBS	BGT			
时钟周期	1	2	3	4	5	6	7	8	9

在第5个时钟周期，SUBS的结果确定了，条件标志位被更新，因而执行分支指令。如此，下一条指令在第6个时钟周期才能被正确取指，但造成了流水线一条对角线资源的浪费：

取指和译码	MOV	AND	SUBS	BGT	X	NOT			
执行指令		MOV	AND	SUBS	BGT	X	NOT		
写回			MOV	AND	SUBS	BGT	X	NOT	
时钟周期	1	2	3	4	5	6	7	8	9

为了减少这种浪费，如在5.2.5节中所提到的，一些处理器会采用预测执行。这意味着将直接对NOT这条指令进行取指。如果分支发生，则将NOT指令从流水线中清除；如果未发生，执行将继续执行。以下是分支预测且预测错误的预约表：

取指和译码	MOV	AND	SUBS	BGT	NOT	MOV			
执行指令		MOV	AND	SUBS	BGT	NOT	MOV		
写回			MOV	AND	SUBS	BGT	X	MOV	
时钟周期	1	2	3	4	5	6	7	8	9

分支预测当然不会总是预测正确：当预测正确时，流水线满效率工作；但是当预测错误时，效率会降低，但这并不会比不带分支预测的时候差。在现代处理器中，都带有一些奇怪但是十分
184 先进的技术来提高分支预测硬件的正确性（详见框5.4）。

框5.4 分支预测

对于支持分支预测的处理器，它们总是去预测该执行分支或者不该执行分支。正确的预测将不会降低处理器的效率，而错误的预测则会浪费处理器一些时钟周期。

对于一些处理器，它们的预测总是不变的，比如不分支。这样编译器将代码优化为不分支比分支多，从而提高处理器的性能。

而更智能的处理器则会记录之前条件分支执行的结果。如果之前执行结果分支比不分支多，则预测后续的条件分支执行为分支；反之为不分支。这就是所谓的**全局预测器**（global predictor）。更先进的硬件则会对不同的分支分别记录，或者是使用类似于cache的32位或64位记录器对低5位或低6位地址相同的分支统一记录，这就是**局部预测器**（local predictor）。

而最复杂的预测器包含了一个全局预测器和多个局部预测器，它具有极高的预测率。这是以性能为目标的研究领域，到目前为止，最优的预测率还是由编译器和硬件来协同获得。

我们将在5.7节对此进行更深入的讨论，在框5.5中先给出一个简单预测器硬件的例子。

框5.5 预测执行

近几年来，各种预测执行被开发出来，尤其是基于IBM的分割流水（split-pipeline），它对条件分支的两个分支分别在两条独立的流水路径上同时执行。一旦分支条件确定，则会丢弃错误的路径。显然，由于将不同分支独立开来，这种机制并不会损失处理器的效率，但是会增加硬件的开销。

框5.4中所提到的分支预测中所描述的“可能性跳转”模型的能力稍弱，它记录过去条件分支指令的执行、结果，并将在5.7节中进行更深入的讨论。

尽管一些非常先进的硬件预测机制被提出，大多数分支预测系统都是使用固定的分支路径即“总是执行分支”或“总是不执行分支”。编译器需要注意这种特点，它们可以通过改变代码的顺序来最大化发挥处理器预测的正确性。目前已经有很多这方面的研究在继续进行。

5.2.7 编译时流水线补偿

在我们讨论分支补偿之前，还存在一个问题，即由流水线阻塞引起的效率下降。这个问题与流水线的构造和长度有关，但先来看看这两者之间的关系。

在现代处理器中，三级流水的流水线相当少，更多的是七级、八级，甚至更多级的流水线，并且流水线往往十分复杂。在前面的三级流水线例子中的对角线资源浪费，成为七级流水线的

一大难题，严重影响了处理器的性能和效率。也许这也就是为什么近些年来人们仍旧会花费大量时间和精力来研究流水线的原因。

用于提高流水线性能的编译时策略涉及范围很广，从最微小的细节到高度复杂的整体。为了展示其中一种普通但却十分有效的方法，考虑5.2.6节提到的那个例子的代码：

```
loop:   MOV R1, #5          ;R1=5
        AND R4, R3, R1      ;R4=R3 AND R1
        SUBS R2, R0, R1     ;R2=R0-R1
        BGT loop            ;如果结果为正则分支
        NOT R3, R4
```

185

这段代码的问题在于取指后续指令之前，不能确定条件分支指令是否应该执行分支，所以必须等待分支结果出来后再取指或者是预测取指。

但对于这种情况，我们可以调整一下代码的顺序，使得条件设置指令（SUBS）和条件指令（BGT）之间稍微有些空隙，如下所示：

```
loop:   MOV R1, #5          ;R1=5
        SUBS R2, R0, R1     ;R2=R0-R1
        AND R4, R3, R1      ;R4=R3 AND R1
        BGT loop            ;如果结果为正则分支
        NOT R3, R4
```

在这个例子中，打乱代码的顺序并不影响程序的最后结果（因为SUBS的结果对于AND的执行并不造成任何影响）。我们来看看预约表：

取指和译码	MOV	SUBS	AND	BGT	NOT				
执行指令		MOV	SUBS	AND	BGT	NOT			
写回			MOV	SUBS	AND	BGT	NOT		
时钟周期	1	2	3	4	5	6	7	8	9

186

无论是否执行分支，SUBS指令都会在第3个时钟周期的末尾更新条件标志位，而条件分支语句只需要在第5个时钟周期前得到条件标志即可。由此，在条件标志位改变和条件分支之间有充足的时间，不需要等待条件标志位发生改变，使得流水线的执行能保持高效和连贯。

这种改变代码来适应流水线的方法对于其他冒险如数据改变冒险、模式改变冒险也有效。当编译器无法打乱代码顺序时（例如两个连续的分支或存在大量的相关性），编译器会插入一些NOP指令，或假设流水线会足够智能地自动阻塞一些时钟周期来保证流水线执行的正确性。虽然一些早期的流水线要依赖于编译器或编程人员加入NOP指令来保证流水线执行的正确性，但对于现代处理器来说这种假设是可行的。

5.2.8 相对地址分支

观察之前讨论过的预约表，可以看到不同的流水线级是由不同的功能单元组成的。预约表可以指出这些功能单元什么时候在工作。

执行级包含了ALU（与FPU的最大区别在于ALU里边都是单周期数值计算器，而FPU通常都需要多周期执行），给人的第一感觉是在分支指令中并没有使用ALU。然而，当分支指令需要计算目标地址时，就需要用ALU来做这些运算，这就是相对地址分支的一种情况。相对地址分支指通过向前或向后移动一些地址完成分支（详见框5.6和参考第3章）。这种分支与程序计数器（PC）有关。它需要将一个指定的偏移量加上PC值，然后将结果设置为PC的新值。

框5.6 相对地址分支

在ARM处理器中，指令都是32位的（地址线和数据线也是32位的，而早期ARM处理器的地址线是26位的）。假设32位的地址线可以指定任意一条指令如分支指令的地址，并且为了指定所有的地址，地址线的32位全部需要使用。

由此，对于分支指令来说不可能32位全部用来存放地址信息，因为每条指令都需要一些位来指出其他的信息（如指出分支指令类型、分支条件等）。因此，ARM处理器不使用全地址编码，而使用相对地址。

因此，存放在分支指令中的值是一个偏移量，需要与当前程序计数器（PC）的内容相加才能得到分支的目标（branch target）地址。

事实上，ARM处理器把分支指令的偏移量当成24位有符号数。还记得地址都是按字节计算的吧，但对于指令都是4字节的。如果所有的指令都是按4字节地址（如0、4、8、12、1004等）对齐的，则任何分支目标地址的最低两位都将为0，所以这两位不需要存放在指令中。

换句话说，这24位数字指定的是目标地址与当前PC值前后相距多少条指令，而不是相距多少字节。这是一个+/-32MiB的范围：已经大大超出了ARM早期处理器设计时台式计算机512KiB内存容量的范围，但对比今天的计算机则小很多。

事实上，这种分支指令需要像加法指令一样执行加法操作：

```
ADD PC, PC, #24
```

将向前移动24字节地址。与之类似：

```
ADD PC, PC, #-18
```

将向后移动18字节地址。再来看看之前的预约表，很明显，当一个相对地址分支指令出现时，不管条件执行与否，处理器都需要等待分支指令完成执行级的操作从而得到下一条指令的地址后，才能对下一条指令进行取指。例如：

```
ADD R2, R0, R1          ; R2 = R0 + R1
B   +24                 ; 向前跳24字节地址
187 NOT R3, R4          ; R3 = NOT R4
...
(分支指令后24字节)
SUB R1, R0, R1          ; R2 = R0 - R1
```

使用三级流水线来执行以上代码的预约表如下：

取指和译码	ADD	B	X	SUB					
执行指令		ADD	B	X	SUB				
写回			ADD	B	X	SUB			
时钟周期	1	2	3	4	5	6	7	8	9

这个冒险有涉及流水线的效率问题。即使一个分支不是条件性执行，而是相对地址分支，流水线也必须阻塞。对此有两种解决方法，一种方法是为相对地址分支额外增加一个单独的ALU；
188 另一种方法将在下一节讨论。

5.2.9 流水线的指令集补偿

由于编译器可以通过改变代码顺序（如5.2.7节中所讨论的）来分开条件设置指令和条件分支指令，因此也可以向指令集加入这种机制。如在一些早期的MIPS处理器和比较老的TI的DSP处理中的延迟分支（delayed branch）指令。

延迟分支操作是将分支延迟一定的周期后再执行，所延迟的周期足够解决由相对地址分支或由于条件设置指令和条件分支距离太近而引起的任何问题。在作者个人看来，延迟分支操作需要由汇编程序员来实现，通过对所提到的两种处理器编写代码，作者已经意识到有时需要忽视延迟分支机制所带来的性能提升，在延迟分支指令后加入两条 NOP 指令来确保安全。正如我们所看到的，由于观察不到延迟从而产生的诡异代码将对优秀程序员产生极大的挑战。以下是延迟分支的代码例子：

```
loop:   MOV R1, #5          ;R1=5
        SUBS R2, R0, R1     ;R2=R0-R1
        BGTD loop           ;条件分支，需要延迟执行
        AND R4, R3, R1      ;R4=R3 AND R1
        NOT R3, R4          ;R3=NOT R4
        NOP
```

与之前的一些例子一样，里边存在一个条件分支，同时也是一个相对地址分支。汇编器会把“BGTD loop”翻译成“BGTD - 2”，因为循环标志是在分支指令的前两条指令上，在运行时，如果分支执行，则处理器将执行 PC = PC - 2。

由于分支需要延迟执行，所以需要知道被延迟多少条指令，这些信息在指令集的细节中可以找到。假设分支被延迟两条指令后执行，这意味着分支不会发生在包含 BGTD 这条指令的这一行，而是发生在这条指令后的第 2 行，即 NOT 和 NOP 之间。让我们观察如表 5-1 所示的预约表。

表 5-1　预约表截取了延迟分支例子的 12 个时钟周期

取指和译码	**MOV**	**SUBS**	**BGTD**	**AND**	**NOT**	MOV	SUBS	BGTD	AND	NOT	NOP	
执行指令		**MOV**	**SUBS**	**BGTD**	**AND**	**NOT**	MOV	SUBS	BGTD	AND	NOT	NOP
写回			**MOV**	**SUBS**	**BGTD**	**AND**	**NOT**	MOV	SUBS	BGTD	AND	NOT
时钟周期	1	2	3	4	5	6	7	8	9	10	11	12

189

在这 12 个时钟周期里，循环执行了两次。在第一次循环中（加粗表示），分支执行，而在第二次循环中，分支没有执行。在第 3 个时钟周期首次遇到分支指令时，处理器将它取入流水线中，并由于是条件分支，需要等待前一条条件设置指令（SUBS）结束。虽然执行了分支，但是分支之后的两条指令（AND 和 NOT）也被取入流水线中，而直到第 6 个时钟周期分支才执行，此时 PC 被设置为“loop”循环标志所在的指令 MOV 的地址。

第二次循环执行了与第一次相同的指令序列，但由于此时不执行分支，因此在 NOT 后的指令由 NOP 取代了 MOV。

由于这个分支是相对地址分支，所以第一次循环的 BGTD 在第 4 个周期进入执行阶段（使用 ALU 计算分支的目标地址），使得能够及时向 PC 提供分支的目标地址，从而在分支后的指令在第 6 个周期能被正确取指。

预约表中没有足够的空间以全效率地指定不同分支类型（条件分支或非条件分支，相对地址分支或绝对地址分支）。

对于汇编程序员而言必须记住的是分支指令后的 AND 和 NOT 两条指令，不管分支是否执行它们都会被执行。一般情况下为了避免混淆，会用 NOP 指令来替代：

```
BD 目标地址
NOP
NOP
```

这种方式会帮助低层次的编程人员记住分支是延迟执行的，但是这种代码在效率上会带来损失，而编译器会自动注意到延迟分支这种特性。

5.2.10 运行时流水线补偿

我们来看看在5.2.4节中讨论的写后写（WAW）、先读后写（RAW）以及先写后读（WAR）冒险，虽然大部分流水线处理器能够在运行时自动处理这些冒险而不需要编译器干预，但这些冒险其实是可以被编译器处理掉的，而那些使用运行时方法来处理这些冒险的处理器是非常复杂的。

如果$O(i)$是受指令i影响的输出地址集合（包括寄存器、内存地址以及条件标志），$I(j)$是影响指令j的输入地址集合，那么指令i和指令j之间的冒险存在这样的关系：

对于RAW冒险有：$O(i) \cap I(j) \neq \varnothing$

对于WAR冒险有：$I(i) \cap O(j) \neq \varnothing$

[190] 对于WAW冒险有：$O(i) \cap O(j) \neq \varnothing$

一般而言，这些冒险都可以通过转发（取指－取指、写回－写回、写回－取指）来解决。我们来看一个RAW冒险的例子：

```
ADD R2, R0, R3      ;R2=R0+R3
AND R1, R2, #2      ;R1=R2 AND 2
```

在R2上存在冒险，在第2条指令读之前它必须被第1条指令写入（在一条足够长的流水线上是存在的）。然而，我们可以想象在硬件中有一条额外的路径，将第1条指令的输出直接送到第2条指令的输入，如图5-6所示，只需在将结果写回给R2的同时，在执行单元（EX）增加一条路径，将输出送回给其中一个输入。这种方式高效地传递了写回级的数据，在数学上与下面的转换等效：

```
R2 = R0 + R3; R1 = R2 & 2 → R1 = (R0 + R3) & 2; R2 = R0 + R3
```



图5-6 一条四级流水线示意图。它将指令执行的结果直接转发至执行单元的输入以作为下一条指令的输入，而不需要首先将结果存回目标寄存器R2

转发还可以提高执行的速度，例如通过提高使用片上寄存器的比例，减少读写片外存储的次数。来看以下给出的例子：

```
LDR R0, [#0x1000]       ;从内存地址0x1000读数据到R0
ADD R2, R0, R3          ;R2=R0+R3
LDR R1, [#0x1000]       ;从内存地址0x1000读数据到R1
ADD R3, R2, R1          ;R3=R2+R1
```

可以改写成：

```
LDR R1, [#0x1000]       ;从内存地址0x1000读数据到R1
ADD R2, R1, R3          ;R2=R1+R3
[191] ADD R3, R2, R1          ;R3=R2+R1
```

这个取指－取指转发（fetch-fetch forwarding）例子提高了25%的执行速度，而不需要在编译时或运行时进行任何优化。写回－写回转发（store-store forwarding）也会在读内存时产生类似的效果。

需要注意的是，一般情况下多次读写还包含了与内存映射外设（memory mapped peripheral）

的通信，如UART[⊖]，它会出现对一个相同的地址（如串行字节输出寄存器）进行多次写的情况，如果是RAM，这种写操作就会造成浪费。在C语言中，这种地址指针需要用关键字“volatile”声明以避免编译器将其优化掉（原因可以参考7.8.3节）。一个智能（或拥有良好内存组织）的处理器会检测到这种特殊的地址不在正常内存地址范围内，从而不进行这种优化。

以下人工编制的代码段给出了一个最简的数据转发例子：

```
指令1    LDR R0, [m1]        ;从内存地址m1取数据放入R0
指令2    ADD R0, R0, [m2]    ;R0=R0+内存地址m2的内容
指令3    MUL R0, R0, [m3]    ;R0=R0×内存地址m3的内容
指令4    STR R0, [m4]        ;将R0的内容存到内存地址m4上
```

图5-7的上半部表示了这段代码，代码中的操作涉及了8个数据的传输；而在图的下半部对这段代码进行了优化，只涉及5个数据的传输。在两种情况下，传输所归属的指令都已经标出。 192
对上两个图示的操作，所得到的结果是一致的，并与原始代码也相同，但是执行的速度和资源的利用差别很大。在运行时，转发的原则是最小化时间消耗以及数据传输资源的使用，以此来提高执行的速度。

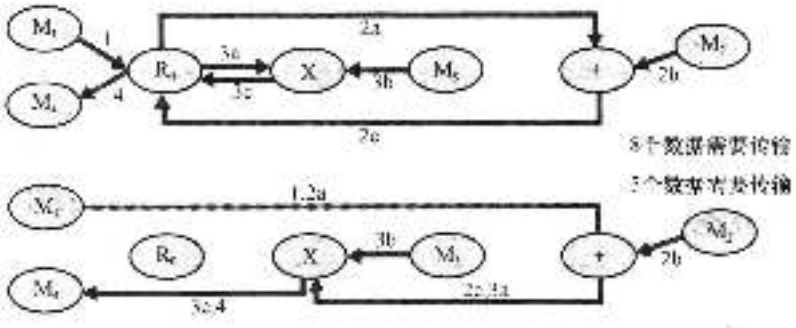


图5-7 一个执行简单算术运算的例子。上部没有采用数据转发，而下部采用数据转发减少了内存读取的操作

运行时外推的缺点是：需要额外的硬件支持，导致功耗、面积增加，由此提高了处理器的价格。然而，单纯从性能上考虑，或者需要考虑向后兼容性时，除去编译时加速的方法外，运行时方法是合适的。

5.3 复杂指令集（CISC）和精简指令集（RISC）

3.2.6节介绍了RISC和CISC体系结构的对比，并将RISC处理器作为处理方式发展的顶峰。RISC处理器起源于一个简单的控制单元，然后逐渐转向微指令，最终将微指令（精简指令）运用到整个CPU上，从而形成了RISC体系结构。

由此，RISC处理器广义上指的是那些比普通处理器指令少且指令简单的处理器。一般公认100条指令是RISC处理器的上限。然而，从RISC处理器问世至今经过了这么多年的发展，RISC处理器更多的特征被提出来，需要注意的是，提升硬件实现和速度的条条框框，这些特征大多是来源于设计公司的市场部门：

- **单周期执行** 所有的指令都在一个时钟周期内完成。这不仅仅可以降低处理器设计的难度，提高指令集的规整度，还减少了中断的响应时间（将在6.5节论述）。实际中，

⊖ UART：通用异步收发器（Universal Asynchronous Receiver/Transmitter），通常称为串口（serial port）。

许多RISC处理器都对这一点有所放宽，例如在ARM体系结构上，存/取指令（STM/LDM）需要多个周期来完成

- **指令不需要解释**——这是指指令执行不需要片上解释器，因为每一条指令都与处理器上一个特定的物理硬件直接关联。
- **指令集规整**——观察一个普通CISC处理器的指令集可以发现，指令间极少存在共性。不同指令的相同位域往往意味着完全不同的内容。有些指令可以访问一个寄存器，而有些则不行。这些都会加大汇编人员编程的难度，同时也增加了片上译码单元的面积。相反，RISC处理器则有一套十分规整的指令集，指令译码简单得多。

[193]

- **规整的寄存器和总线**——有助于指令集规整的一个方法是保持一个（最好大一些）独立的寄存器组，这些寄存器需要有相同的数据范围和操作方法。在CISC处理器中，为了方便地使用最少的指令在不同的功能单元之间传送数据，需要能观测到内部总线结构，而这对于RISC处理器则会变得非常简单：只要一个寄存器可以“看到”一个数据，那么其他所有的寄存器都能“看到”这个数据。
- **存/取体系结构**——由于内存比寄存器慢得多，因此通常很难做到在一个很快的时钟周期内从一个内存单元取数，对这个数做运算，然后再将结果存回内存中。事实上，避免内存访问成为瓶颈的最好方法是，保证在发生一个外部存/取操作时，没有其他导致指令执行变慢的动作发生。因此，就应该分别有一条指令进行读操作以及一条指令进行写操作，而所有的数据操作都只发生在寄存器上或是只有立即数。

正如前文所述，没有统一定义什么是RISC什么是CISC，并且许多现代处理器的设计都同时采用了二者的设计思想。

5.4 超标量体系结构

随着以性能为主导的流水线技术的发展，除了RISC所带来的简洁性外，流水线的复杂度在不断提高。现在的研究大部分涉及多功能动态流水线，并且包含了更多用户自定义的特殊指令，伴随而来的是控制复杂度的提高。

伴随流水线复杂度提高的还有冒险的增加，由此冒险的检测和运行时冒险的解决变得越来越重要。这些都显著地提高了流水线管理对硬件资源的要求。

5.4.1 简单超标量

解决不断增加的流水线复杂度有这样一种方案：流水线仍旧按照简单线性布局，但在执行级含有多个功能单元。如此，指令仍旧以串行顺序执行，但在执行级会经过不同的路径。

通常情况下，执行级是流水线上最耗时的部分，而最耗时的部分则称为流水线的瓶颈。由此，在一个超标量流水线系统上，取指单元会以比执行级上任何单个执行单元存吐率都高的速率将指令

[194] 发送到流水线中，执行单元的多个副本将会并发处理指令。图5-8展示了这样一个五级流水线。

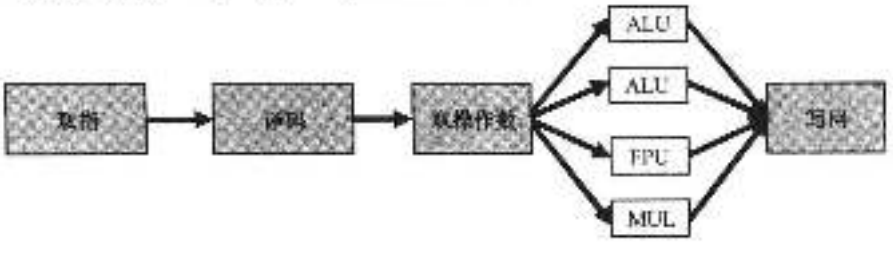


图5-8 一个五级超标量流水线的示意图，其执行级含有多个功能单元

这种方法最先在DSP上采用，其含有多个乘累加单元（MAC），但这种方法只有在通用CPU

上采用时才被正式地称为超标量。

如图5-8所示，这条超标量流水线加入了浮点运算单元（FPU）。FPU的缺点是，将它放在一条线性流水线上（有固定的指令时钟频率）会严重影响处理器的速度。但对于超标量处理器，发送至FPU的指令可以与其他如ALU的指令、乘法单元的指令等并行执行。目前新研制的超标量处理器通常包含8个ALU和16个MAC，或者是多个ALU和4个FPU。

如表5-2所示的预约表是超标量流水线的一个例子。该流水线包含1个取指和译码单元，每个时钟周期发出一条指令。指令被发送至4个功能单元（2个ALU、1个FPU和1个MUL（乘法单元））。流水线末端由1个写回单元结束。观察表5-2，可以注意到取指单元发送指令的速度比流水线执行级上任何单元处理指令的速度都快，并且相比指令输入顺序，写回单元可以乱序执行。并不是所有的处理器都支持乱序执行，它需要处理器支持复杂的运行时冒险规避硬件（runtime hazard-avoidance hardware）。我们将在5.9节对一种支持乱序执行的处理器进行讨论，即Tomasulo方法，而框5.7则简要介绍了另一种方法——记分牌。

表5-2 预约表截取了一个超标量流水线工作的12个时钟周期。注意到MUL2在第7个时钟周期不被执行，它将执行取指和译码单元阻塞至第10个时钟周期，此时它才被执行

取指和译码	ADD	SUB	AND1	FADD	NOT	MUL1	MUL2	—	—	—	NOR	AND2	NOT
ALU		ADD		AND1		NOT						NOR	
ALU			SUB										AND2
FPU					FADD								
MUL							MUL1				MUL2		
写回				ADD	SUB	AND1		FADD	NOT		MUL1		
时钟周期	0	1	2	3	4	5	6	7	8	9	10	11	12

框5.7 记分牌

记分牌（scoreboard）用来记录所有被处理指令的相关性，并且允许此刻没有相关性的指令能够进入流水线执行，这需要打乱原有代码的顺序。下面我们对此进行详细讨论。

发送指令时，处理器根据指令确定其所用的源和目标操作数寄存器，然后阻塞直至遇到两种情况：（i）其他对相同寄存器进行写操作的指令执行完毕；（ii）所需的功能单元可用。这两个条件正好分别消除了WAW冒险和结构冒险。

一旦一条指令被发送至一个功能单元，操作数就会从指令源寄存器取出，但取出过程被阻塞至当前任一条写源寄存器的指令执行完毕，如此可以避免RAW冒险。

收集完所有的操作数后指令开始执行（此时记分牌会一直记录该条指令直至该指令处理完毕）。

最后，指令执行完毕并准备好将结果写回目标寄存器。然而此时，如果先前已发送的指令还没有将操作数从当前将要写回的寄存器取出，将会阻塞写回。换句话说，如果先前的指令仍旧被阻塞以等待执行，并且需要从Rx寄存器取操作数，而当前的指令需要对Rx寄存器进行写回，那么当前的指令将会被延迟直到先前的指令不再被阻塞并已经从Rx寄存器中取出操作数后才能写回。这种机制避免了WAR冒险。

虽然表5-2的程序例子很短，但可以看到流水线的指令输出率比输入率低，这是因为流水线需要暂停以保证有空闲处理单元去执行指令。因此在处理现实程序代码时，这种处理器需要有比平均指令吞吐率高的峰值指令吞吐率。在标准测试中，厂商可能会人为生成使处理器运行在峰值吞吐率而不是实际平均吞吐率的指令序列（我们已经在3.5.2节有过简单的讨论）。

195
~
196

然而，对超标量的讨论并未结束，我们将在5.4.2节中继续讨论在每个时钟周期同时处理多条指令的超标量流水线。

5.4.2 多发送超标量

在上一节，我们已经将多个功能单元加入标量流水线中，虽然比一般标量流水线好一些，但

是这并没有真正成为一个超标量流水线。

多单超标量流水线的好处就是它能够在每个时钟周期同时发送多条指令。也就是说，相比每个时钟周期向多个功能单元发送一条指令，超标量流水线能够在每个时钟周期向多个功能单元发送多条指令。

对比图 5-8 并没有什么显著变化，然而在每个时钟周期发送多条指令却会产生不同的预约表。虽然与之前的预约表很相似，但是在每个时钟周期有两个取指和译码以及两个写回，并且与我们以后将遇到的预约表也完全不同。

下边的执行表展示了在每个周期进行两次取指。这个取指单元向 3 个不同的执行单元发送指令，这些执行单元轮流使用两个写回单元。有趣的是，在第 3 个时钟周期和第 5 个时钟周期，流水线上分别有两个明显的空隙。在这两个时钟周期上，第二个取指和译码单元不能够取新指令。这是因为，在这两种情况下，先前取到的指令因为在等待被占用的功能单元（两种情况下都是执行单元 1）而没有发送。这是这种超标量处理器在现实中常见的现象。

取指和译码	I_1	I_2	I_3	I_4	I_5	I_6	I_7
取指和译码	I_8	I_9	—	I_{10}	—	I_{11}	I_{12}
执行单元1		I_1	I_4	I_7	I_{10}	I_{11}	I_{12}
执行单元2		I_2				I_6	
执行单元3			I_3				I_8
写回			I_1	I_3	I_4	I_6	I_7
写回			I_2		I_7		I_8
时钟周期	1	2	3	4	5	6	7

至此我们已经为不同的流水线操作制作了很多预约表，然而还有其他制作预约表的方法，在图 5-9 中就给出了其中一种。图中按顺序从上到下显示发送的指令，横坐标轴表示时间。在这个例子中，流水线上并没有出现阻塞，所以所有的指令都按顺序执行并且按顺序完成。然而现实 197 中不需要都这样执行。



图 5-9 预约表的另外一种格式，它展示了指令按顺序从上往下执行，时间上是从左往右。垂直方向的直线与我们之前看到的预约表一样表示了在某一时刻流水线的操作

5.4.3 超标量的性能

超标量体系结构的一个特征是它发送指令的速度与处理指令的速度相当。理论上，超标量处理器并不需要支持流水线，但是在现实中，几乎所有的超标量处理器都是流水线的。

任何事物都取决于用什么性能指标去衡量它（在 3.5.2 节有讨论）。我们已经提到过，超标

量处理器需要能够高速发送指令，虽然现实中平均指令发送率比峰值要低，但这主要是取决于执行单元的占用率。从这种局限性出发，可以通过改变编译器设置将使用不同类型功能单元的指令进行交错来提高指令发送率。指令发送率主要取决于所处理任务的性质。

最好的情况可以在图5-9中看到，超标量处理器可以并行处理指令。这也是一种并行计算机的形式（在5.8节我们将会有更细致的探讨）。

5.5 每周期的指令数

每周期的指令数（IPC）至少在理论上是一个非常重要的衡量处理器执行程序快慢的指标。但IPC并不能衡量每周期做多少工作，因为这还取决于每条指令能做多少工作以及编程人员和编译器的能力。因此，在不同的机器上比较代码的执行速度也是没有意义的。

然而IPC是一个有用的指标，它是通过典型代码得到的平均数据，可以指出某一种体系结构 [198]
原始的处理能力。事实上，平均值与CPI峰值的比率可以作为一个直观的衡量单位，一个非常接近峰值的平均值表明某体系结构对所执行的代码是非常优化的。

5.5.1 不同体系结构的IPC

不同类型的处理器的IPC值不一样，并且因为结构不同而导致的工作方式也不一样：

- **CISC处理器** IPC值远低于1，这是因为它们的指令相对来说都很耗时，而且在发展史上也极少去对这种体系结构的指令进行简化。
- **RISC处理器** 相比之下，IPC值可以达到1，虽然在不同情况下可能达不到这个值。达不到1的原因包含：偶尔出现的冗长指令（如乘法、除法等）以及取/存指令需要等待慢速的外部存储器——RISC处理器几乎都是取数-存数机器（参见3.2.3节）。后一种因素在DSP处理器使用外部存储器时特别普遍。流水线可以帮助RISC处理器的IPC更接近1。
- **超标量处理器** 我们已经知道，超标量处理器能够并行发送多条指令。对于一个拥有n个指令发送单元的机器（即每个时钟周期最多能够发送n条指令），那么它的IPC能够达到n。然而在5.4.2节中我们已经看到，如果一段指令都需要相同的功能单元（硬件相关）或者存在数据相关问题，都将导致流水线停顿。显然，流水线的停顿越多，超标量处理器的IPC值就会越低。
- **VLIW[⊖]/EPIC[⊜]处理器** 它们都将在9.2节中讨论，其IPC都远远大于1.0。它们适用于专用领域如多媒体处理和信号处理。
- **并行机器** 在一台计算机上包含两个或多个处理器核，虽然每个处理器核的IPC不高，但在并行执行时会有很高的吞吐率。由此，整台机器的IPC就等于每个处理器核的IPC乘以处理器核的数量。我们将在下面做更深入的讨论。

提高IPC是当前许多处理器设计人员的主要关注点，并且也是计算机体系结构设计师用于提高机器性能的主要方法。

在写本书的时候，并行机器非常有吸引力，这源于主要的处理器制造商如Intel的推动，他们开发出了双核、四核甚至更高层次的并行。我们将在5.8.1节中简单讨论双核处理器，在5.8节中也将对并行计算方法进行更全面的讨论。简单而言，目前制造商们在通过提高处理器体系结构复杂度来提高处理器时钟频率的方法上遇到了一个转折点：不论使用什么方法，都不能使 [199]

⊖ VLIW（Very Long Instruction Word）：超长指令字。

⊜ EPIC（Explicitly Parallel Instruction Computing）：显式并行指令计算。

得性能的提高和体系结构复杂度的提高等价。换句话说，使用今天的工具来提高处理器的性能已经变得越发困难。

如果对以上我们所描述的那些方法进行更深入的讨论，将会看到现代的计算机体系结构设计师不断将日益增长的性能需求转嫁到编译器和软件上。让我们归纳一下：CISC处理器在硬件上实现了很多操作。相反，RISC处理器通过使用简单的指令简化了（也加速了）硬件。RISC意味着更多的软件指令，但会执行得更快。所以RISC程序一般会比CISC程序长，同时其编译器也比CISC的复杂一些。超标量系统在流水线上实现了一些有限的并行，而为了避免流水线停顿，其处理数据相关性的问题也变得十分重要。因此，为了获得好的性能，超标量的编译器必须要考虑相关性问题，并且要熟知超标量流水线的能力。对于将会在第9章讨论的VLIW和EPIC，比目前任何我们所讨论的体系结构都要复杂，并且完全要依赖于编译器的调度。

对于并行机器也是同样。虽然处理器本身可能十分简单，但是它们之间的互联关系变得十分复杂。而且现在存在这样的问题：当前的软件工程师能否将软件都写成并行的。除去编程人员的因素，还存在这样的问题：当前最流行的编程语言并没有针对并行处理器进行任何优化。这样看来，在并行机器被完全开发出来之前，有两件事情要先完成：（ⅰ）新一代的能够写并行代码的软件工程师；（ⅱ）新一代支持并行计算机的编程语言及相关工具。

关于并行处理还有一点要注意的是，虽然“转向并行”成为当前处理器制造商持续满足不断增长的性能需求的有效手段，但获得所需的加速还需要编程人员的支持。对于一个独立的程序，这种加速是难以获得的。然而，特别是对于服务器或台式机，它们运行的是多任务操作系统如Linux，通常会有多个线程（任务）同时执行。并行机器会将不同的线程划分到不同的处理器上执行，对于单个线程，虽然从CPU时间上讲并不会运行得更快，但由于不用与其他任务分享时间隙和优先权，所以它会完成得更快。在嵌入式系统中，在一段时间内通常只有少数几个任务在执行或只有一个任务在执行，因此使用并行技术很难获得性能提升。在这种系统中，只有当关键任务本身是能够并行化[⊖]的，才能够得到加速。这又重新回到那个话题，需要好的并行工具和
200 语言给好的并行编程人员使用。

5.5.2 IPC度量

正如我们在5.4.3节和其他地方（包括3.5.2节）所提到的，处理器运行所获得的性能是不能和所预测的性能相提并论的。如果工程师想执行一个已知的算法，可以将该算法分别运行在不同的体系结构上，看看哪个执行得更快。然而，对于非特定代码的性能预测依赖于很多因素，但可以这么估计：将IPC除以指令周期频率，再乘以所需执行的指令数。

随着程序大小和普遍性的提高，这种定义变得更准确。需要记住的是，嵌入式系统的计算任务通常小而且固定；相反，运行在台式机或服务器上的代码通常在设计时很难预测。

问题随之产生，IPC数据是否精确。对于任意体系结构，都有特定的规定来提高IPC准确性，下面是其中的一些：

- 使用峰值IPC而不是平均IPC来得到最佳情况下的数据。
- 使用特定测试代码而不是具有全局代表性的代码来获得平均IPC。
- 用于获得IPC的执行指令全部来自内部存储器。
- 需要使用外部存储器时，仅考虑在时钟频率低时获得的IPC（由于处理器时钟频率很慢，因此访问外存的速度不足以影响IPC的值）。
- 在选择代码时，不使用或极少使用慢速指令来评估IPC。

⊖ 并行化（parallelised）：使程序可并行执行。

- 移除已知的慢速代码段。

通过本书的讨论，读者应该能意识到不同的体系结构都有自己的优缺点。然而为特定的计算任务选择合适的处理器不仅是一门科学，更是一门艺术：这需要一定的直觉。在作者看来，需要忽略与性能相关的市场和销售因素。性能极少成为关键性准则，其重要性往往比不过易编程性、可扩展性、可获得的支持和开发工具、产品寿命，以及其他技术因素。

5.6 硬件加速器

在现代CPU中，大部分的硅片面积都是专门用来加速基本的处理操作的。加速方法包括：使用高速缓存，为体系结构添加额外的总线、流水线，以及混合专门的数值处理单元。 201

起初，处理器只包含了一个基本ALU来对数值进行处理（可以看到，所有的操作都可以通过ALU来完成，只要执行速度不是太重要）。然后，加入了乘累加单元以加快乘法操作，其通过反复相加来实现乘法。

浮点计算模块曾经作为额外的可选配置，需要插入一个单独的芯片，但现在却被当做台式机的标准配置。与浮点模块相同的是，台式处理器现在按常规都包含SIMD硬件（参见2.1.1节），并且已经开始为支持无线网络而整合了不同的硬件加速器。

其他的硬件加速器还包含了图形控制、加密、通信以及数据压缩。目前看来，硬件加速器的种类还在不断扩大，并且还可以为特定应用提供加速——尤其是在专门的嵌入式片上系统处理器中。

另一方面，硬件加速还对体系结构做了改进，从而提高了处理的速度，而这种提高是与数据无关的。在之前的讨论中已经提到了一些，如流水线（5.2节）、高速缓存（4.4节）、多总线体系结构（4.1节）以及自定义指令（3.3节）。本节将对更多此类的硬件加速机制进行讨论。

5.6.1 零开销循环

许多算法都包含了循环，如for()、while()以及do()。总体来说，循环控制都需要有开销。考虑以下给定循环次数的循环例子：

```
i = 20;
while (i-- > 0)
{
        <循环体>
}
```

这串代码需要以下操作步骤：

1. 设置i=20。
2. 比较i是否为0。
3. 如果等于0，则分支至循环体后的指令。
4. $i=i-1$。
5. 执行循环体。
6. 跳回循环开始处（第2步）。

根据循环类型，需要在循环体之前或之后检测循环条件，以判断是否要继续执行循环，但如果循环体本身很简单，检测循环条件的开销就相对变得很大。来看看以下一段来自DSP的代码，它实现了一个数字滤波器：

```
for(i = 20;i>0;i--)
    y = y + x[i]* coeff[i];
```

202

循环体的计算虽然看上去比较复杂，但是在现在的DSP处理器中只需一条简单指令就可以实现。不过，如果采用刚才提到的6步循环操作，那么这段代码将由1条设置指令以及随后的20次第2～6步重复执行组成，加起来总共有101条指令。

由于很多DSP代码的循环都是类似于这种循环体很小的循环，DSP设计人员已经意识到使用大量额外的指令来支持这种循环效率非常低。因此开发了零开销循环（Zero Overhead Loop，ZOL）。

以下是一个德州仪器（TI）TMS320C50处理器的汇编代码例子：

```
        set  BRCR to #20
        RPTB loop-1
        ...<循环体>
loop    ...<紧跟循环体>
```

在这个例子中，使用一条指令设置BRCR循环计数器，然后使用另一条指令进入循环。DSP处理器会检查PC指针，当它到达（loop－1）时，自动将PC指针复位回循环起始位置。在这个例子中总共进行20次这样的操作，总共执行了22条指令，而不是像不支持ZOL时需要执行101条指令。

Analog Devices在他们的ADSP2181处理器上也有类似方法：

```
        set  CNTR to #20
        DO loop UNTIL LE
        ...<循环体>
loop    ...<紧跟循环体>
```

可以看到这两种方法的基本原理是一样的，但后一种方法可以支持不同的循环终止条件（LE指小于或等于，除此以外还有其他15种条件），5.6.2节将会在此基础上对ADSP2181的寻址能力进行探讨。

203. 支持ZOL的硬件相对比较简单，图5-10是其模块图。

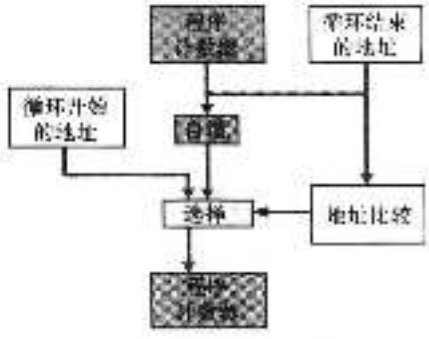


图5-10 在一个处理器上实现零开销循环所需的硬件及其模块间通信示意图

图中硬件的功能需求如其名字一样，有用来存放循环起始地址的，有用来存放循环结束地址的，有一条PC指针达到循环结束的路径（地址比较器），以及一条通过重置PC指针为循环起始地址分支回到循环起始处的路径。除此之外，还需要循环计数器保持和自减的方法以及判断循环停止条件的机制（例如，循环计数器为0）。

一种可能发生的比较复杂的情况是，循环指令不是一个简单指令，而是一个调用其他函数的指令，而其所调用的函数又包含了循环。由此，这就需要嵌套循环。在ADSP中，ZOL寄存器是包含在堆栈里的，因此只要循环结束地址不同就可以以最小开销自动嵌套循环。相反，TMS缺少这样的硬件支持，因此循环嵌套就需要手动保存/重置循环寄存器。

另一种复杂的情况是，虽然以上两种ZOL例子都是用汇编语言写的，但现在大部分代码都是用C语言写的。因此C编译器需要能够意识到可以使用ZOL硬件。简单的C结构如之前的while和for循环，以及如下的循环都可以很容易地用ZOL实现：

```
k = 20;
do{
    <循环体>
} while (k-- >0)
```

注意到以上例子的循环计数器都是向下计数的。在TMS中，循环计数器是不能向上计数的。

所以如下代码：

```
for(i = 0;i <20;i++)
{
      <循环体>
}
```

在汇编代码中需要转换成向下计数（如计数器从20递减到0），通常假设所用到的编译器都足够智能，可以完成这种转换。

同时，软件编程人员负有让C代码结构能够使用ZOL硬件的责任。对于这样的硬件，最好避免代码中的循环计数器不是以1为计数递增或递减，也要避免在循环体的算法里将循环索引用于计算。

给定一些支持零开销的循环，那条老的嵌入式代码原则——将所有能够合并的独立循环都合并——并非总是正确的。事实上，如果在循环时强迫将变量存放到外存上（可能因为缺少临时寄存器），那么这也许是有害的。

在ADSP2181中支持硬件无限循环，但可以将循环体中的某部分作为终止循环条件。这对于 204
C程序是十分有利的，因为C语言包含所有可能的循环结构。

此类循环硬件加速器称为PC陷阱（PC trap）。目前还有更多复杂的硬件用来完成类似任务，将在下一节讨论。

5.6.2 地址处理硬件

ARM处理器只有一个通用寄存器组，而许多处理器却将数据和地址分别存放在不同的寄存器上。事实上，这些处理器中之所以这么做是因为它们的数据总线和地址总线宽度不一样。

Motorola 68000系列处理器的寄存器都是32位的，却把寄存器分为8个数据寄存器D0～D7和7个地址寄存器A0～A6。虽然任意的值都可以保存在这些寄存器中，但是只有将地址存放在地址寄存器中时才会被当成地址使用。类似地，正在处理的指令不能将结果直接存放到地址寄存器中。程序员也可以利用地址寄存器所附带的硬件在访问前或访问后增加或减少地址值，以及将地址值用于索引。然而，如果需要更复杂的地址运算，就需要将地址从地址寄存器转移至数据寄存器中进行算术运算，然后再将结果放回地址寄存器中。

ADSP21xx系列DSP处理器通过数据地址生成器（Data Address Generator，DAG）扩展了这种方法。在ADSP2181中包含两组这种寄存器，每组分别包含4个I寄存器、4个L寄存器和4个M寄存器：

I0	L0	M0
I1	L1	M1
I2	L2	M2
I3	L3	M3

DAG1

I4	L4	M4
I5	L5	M5
I6	L6	M6
I7	L7	M7

DAG2

每一个索引（I）寄存器存放用于访问存储器的真实地址，L寄存器存放了与这些地址一致的存储区域长度，M寄存器存放更改的值。

在汇编代码中，读存储器通过下述语法实现：

```
AX0 = DM(I3, M1);
```

这意味着从数据存储地址为I3寄存器所指的地方读取一个数据，读出的数存放到AX0寄存器中，然后通过加上M1寄存器的内容完成对I3寄存器的修改。如果I3的新值超过了长度寄存

器L3 + 起始I3，那么将I3中的值模起始I3值然后保存（起始I3值是指缓冲区的首地址）。如果
205 L3寄存器的值为0，则意味着I3的内容没有发生改变。这种安排将会通过一些例子给出更详细的说明（见框5.8），但需要注意到，指令中的任何地方都没有提及L3。这是因为，I寄存器和L寄存器操作都是同时进行的，而M寄存器则是独立的：在每个DAG中，任何M寄存器都可以用于修改I寄存器，但这个DAG中的M寄存器不能对其他DAG中的I寄存器进行修改。框5.8给出了ADSP21xx ZOL硬件操作的3个例子。

框5.8 ZOL示例

例1：考虑一个访存部分使用ADSP汇编器汇编、其余使用ARM汇编器汇编的混合例子。对本例及其他例子所指的地址都会有明确的说明。通常情况下，由于有一些特定的约束（在这不予讨论，但在ADSP21xx的手册中有说明），这些地址都是通过链接器（linker）来分配的。

```
        MOV I0, #0x1000      ; 设置I0 = 0x1000
        MOV L0, #0x2         ; 设置L0 = 2
        MOV M0, #0           ; 设置M0 = 0
        MOV M1, #1           ; 设置M1 = 1
loop:   AX0 = DM(I0, M0)     ; AX0取数
        ADD AX0, AX0, #8     ; AX0 = AX0+8
        DM(I0, M1) = AX0     ; AX0存数
        B loop
```

接下来，我们将对循环执行过程中I0的值的变化构建一个表：

	指令	I0变化		指令	I0变化
1	MOV M1, #1	0x1000	6	AX0 = DM(I0, M0)	0x1001
2	AX0 = DM(I0, M0)	0x1000	7	ADD AX0, AX0, #8	0x1001
3	ADD AX0, AX0, #8	0x1000	8	DM(I0, M1) = AX0	0x1000
4	DM(I0, M1) = AX0	0x1001	9	B loop	0x1000
5	B loop	0x1001			

注意到在第2行中I0被M0更改了，但是由于M0为0，所以I0的值不会改变。在第4行，I0被M1更改，由于M1 = 1，所以I0加1。同理，第6行以及第8行的更改也类似。虽然I0的值应为0x1002，但由于L0 = 2，即缓冲区的长度为2，所以地址又跳回了0x1000。

例2：L1设置为0，I1为0x1000，M0为0x10。

通过AX0 = DM(I0,M0)连续地从I0读值将会看到地址寄存器I0中出现以下值：0x1000、0x1010、0x1020、0x1030、0x1040、0x1050等。这是因为L1为0，所以地址不会出现循环。

例3：在这个例子中，L4被设置为50，I0为0，M4为2，M5为10。这构建了一个拥有50个存储空间的循环缓冲区，起始地址为0。执行以下循环：

```
loop:   AX0 = DM(I4, M5)
        AY0 = DM(I4, M4)
        B loop
```

随着循环的执行，I4的值将会如此变化：0、10、12、22、24、34、36、46、**48**、**8**、10、20、22等。注意使用黑体标注的数字。先看48，I4索引应加上10变为58，但由于L4为50，所以这超过了缓冲区的长度，因而I4被循环回起始处，由此48之后为8。

毋庸置疑，ADSP拥有强大的地址处理能力，但考虑3.3.4节所提到的基于ARM处理器的寻址模式。事实上，ADSP除了其先进的寻址硬件外，并没任何超越那些寻址模式的能力。

由此DAG和与其相关的硬件对于维护环形缓冲区和执行同步地址变化（例如预定义向前或向后）是十分有用的。然而，在所获得的这些效率之外，这种硬件并没有从本质上提高处理器

的性能。这种效率的获得带来的开销是，需要为硬件——如图5-11所示的ADSP2181中的一个DAG单元提供芯片面积。

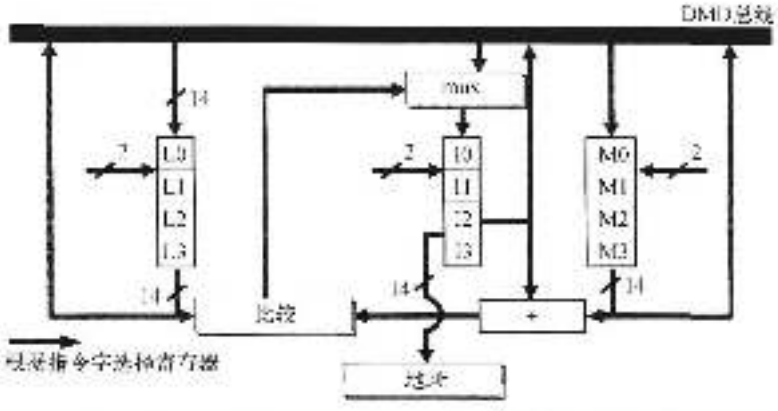


图5-11 数字信号处理器ADSP2181内的第二条数据地址生成器（DAG）的简化框图。它显示了内部长度寄存器L0～L3、索引寄存器I0～I3及修改寄存器M0～M3是如何与地址专用加法器和内部DMD（数据－存储－数据）总线相连的

从图中可以看到，由于在每个指令周期最多访问DAG中的一个寄存器，因此每一个L、I及M寄存器都通过共享总线访问。DMD（Data-Memory-Data，数据－存储－数据）总线用于传输数据操作数并和数据存储器连接（4.1.3节中更详细地说明了ADSP这种非常规的内部总线结构）。另外，DAG1（没有画出来）可以按位反序输出地址，这在快速傅里叶变换和其他数字信号处理中能带来显著的性能提高。

由于在ADSP21xx中片上数据和代码是分开存储的并分别占独立的总线支持，同时有两个DAG，因此ADSP21xx可以通过后改（post-modification）和循环地址（wraparound）直接访问内存中的两个DAG地址操作数。一经访问，这两个操作数可以在1个指令周期内被处理并存储。相反，对于ARM处理器，在功能上也能完成这样的操作，但是并不能在1个指令周期内完成。我们也曾提到过，ARM不允许在一条指令内访问两个不同地址的操作数（因为取数－存数（load-store）结构最多只有一个地址操作数——在框5.9中有详细讨论）。

206
～
207

框5.9 ARM中的地址编址

作为一款RISC设计，ARM一次地址计算除了移步内专用处理硬件，但只能简单变址寄存器的字段，ARM处理这步指令的速度也会提升。

正如3.2.3节所讨论的，ARM通过一条指令load指令和一条指令store指令实现load-store结构（事实上在多处理器系统中，还有一条swap指令）。装载和存储的地址可以为前面偏移或后面偏移（偏移为减少），也可以为预索引或后索引。

ARM利用ALU和移位器进行地址计算，因为这些硬件在流水线执行实际存储指令的阶段是可用的（详见5.2.8节）。与ADSP的DAG相比，ARM的ALU在移位器比DAG的ALU和移位器更通用更高。

以下是这种通用性的例子：

```
LDR   R0,   [R1, R2, LSL#2]
```

这条指令装载内存地址（R1＋（R2×4））的内容放入寄存器R0。LSL是逻辑左移指令，这种地址计算对于ADSP21xx的DAG是无法实现的。

最后，需要注意在ADSP21xx中没有选择（alternate）或影子（shadow）DAG寄存器（将在5.6.3节中介绍）。这意味着DAG的使用是依赖于程序上下文以及中断支持的，需要通过直接写

208 汇编代码才能完全利用这种地址处理加速器。

5.6.3 影子寄存器

CPU寄存器是处理器上下文的一部分，程序运行时可见。其他上下文则包括状态标志和可见存储（viewable memory）。

当一个正在运行的线程被一个外部中断信号中断时（将在6.5.1节中讨论），中断服务例程（ISR）就会运行以响应这个中断信号。一旦ISR运行完毕，控制又转回到原先的程序。“控制”一词在这里是指程序计数器的指向。程序正常情况下应该是按照汇编代码一条接一条顺序往下运行的，而中断会将程序切换到ISR上运行，在ISR执行完毕后，再转回到原先的程序上继续执行，就好像什么事情都没发生过一样。

来考虑这样一个过程，当中断使能时，就有可能在任意两条指令之间触发一个ISR。因此对于ISR来说，它必须能够在返回时将所有的东西都恢复，使得程序的上下文跟它被调用前一模一样。

在几年前，程序员必须在ISR开始的时候对当前程序进行所谓的上下文保存，并在ISR退出前恢复上下文。保存上下文的过程是将每个寄存器按顺序进行压栈，而恢复就是将其按相反顺序进行出栈。这个过程会花费20～30条指令的开销，并且必须在ISR执行正式的中断操作前完成。

为了避免这种开销，影子寄存器被开发出来。影子寄存器是第二个寄存器组，在各方面与主寄存器组相一致。然而，它可以在ISR需要的时候被利用（包括修改），而不需要修改主程序可见的原始寄存器。以TMS320C50为例，一旦有中断发生，处理器会跳到对应的ISR并自动转换至影子寄存器。当ISR执行完毕时，一条专用的跳回指令使程序跳回中断前的PC，并转换回原始寄存器。

有了这种影子寄存器，就不需要手动地在ISR开始时保存上下文、结束时恢复上下文。在代码的任何地方都可以进行中断而不需要任何额外开销。然而，如果只有一个影子寄存器组，中断就不能嵌套，这意味着在一个中断发生时，不能响应另外一个中断。

5.7 分支预测

在5.2.6节，我们研究了由于分支导致的流水线性能下降的现象。我们看到分支本身引发了很多问题，而条件分支冒险和相对地址分支会加剧这些问题。在前面的框5.4中我们简单介绍了分支预测并认为预测执行（见框5.5）是降低分支开销（branch penalty）的一种办法。

在本节中，我们首先对由于分支引发的性能下降问题做一个总结，然后讨论用于减少此性
209 能下降并与预测执行能力相关联的分支预测方法。回想我们之前介绍的方法中所特有的问题，这些问题使得分支引发的性能降低成为计算机体系结构设计师心中永远的痛。

在理想世界中，我们可以训练程序员避免使用分支指令，但当这些指令出现时，本节所介绍的专用硬件仍将需要，并且也是CPU性能研究的热点。

5.7.1 分支预测的必要性

首先，我们概括一下分支所带来的问题。考虑以下代码在一个四级流水线（取指、译码、执行、写回）上执行：

```
i1        ADD R1, R2, R3
i2        B loop1
i3        ADD R0, R2, R3
i4        AND R4, R2, B3
loop1:  STR R1, locationA
```

不建立预约表，我们执行开始的几个周期：

- *i*1 被取指。
- *i*2 被取指，同时 *i*1 被译码。
- *i*3 被取指，同时 *i*2 被译码并且 *i*1 被执行。在这个周期的末尾，CPU“知道” *i*2 是一条分支指令。

此时，*i*3 已经被取入流水线。然而，由于 *i*2 是一条分支指令，因此正确的操作应该是将 loop1 所标记的指令作为下一个周期执行的指令。由此 *i*3 应该被清除出流水线并取入正确的指令。清除的工作将会在流水线中形成一个“气泡”，从而降低了流水线的效率。

在5.2.8节中我们也讨论过相对地址分支的问题：分支的目标地址（分支指令后应该执行的指令的地址）通常是一个以当前程序计数器（PC）为基准的偏移量，保存在分支指令中。所以CPU需要使用ALU将偏移量与PC相加计算出下一条指令的地址。

在上面给出的例子中，如果分支的地址（在这个例子中是 loop1 所指指令的位置）需要通过计算得出，那么在分支指令被译码后还需要另一个周期。通常情况下，使用这种技术的处理器会立即将流水线清空并执行分支。操作序列如下所示：

- *i*1 被取指。
- *i*2 被取指，同时 *i*1 被译码
- *i*3 被取指，*i*2 被译码，并且 *i*1 被执行。 210
- *i*4 被取指，*i*3 被译码，*i*2 被执行（这意味着分支的目标地址将通过 ALU 计算出来），并且 *i*1 的结果被写回。
- *i*2 的结果即分支的目标地址被写回——但写回的地址是 PC 而不是其他寄存器，同时流水线被复位（丢弃 *i*3 和 *i*4）。
- 分支地址处的指令被取指。[⊖]

至此，我们还没有提到条件分支冒险的情况，这种情况下流水线需要等待分支指令之前的条件设置指令计算完成来决定是否执行分支。

然而，我们已经对用于缓解分支问题的预测进行了讨论。总体而言，预测执行是在等待分支条件得出结果前或分支地址计算完成前执行某一个分支路径。在预测执行的路径执行完前，处理器确定该预测是正确（此时所预测执行的指令可以完成）或不正确（此时所预测执行的指令及其结果将被抛弃）。

在某些处理器上，预测路径是不确定的，例如总是预测会执行分支，或总是预测不执行分支。当然，在不考虑其他因素的情况下，这种预测的正确性总是很难超过50%。在编译器的帮助下，让编译器组织的代码在有任何分支的地方更多地走所预测的路径。

实际上，预测是在赌博：猜测某一条路径被执行。正确的猜测开销小，因为在此情况下，处理器通常不会有流水线停顿。而错误的猜测将会引发流水线停顿，因为错误的执行需要从流水线上清除。

一种更精细的预测是分支预测，它通过如下的信息进行更智能的猜测：

⊖ 需要注意的是，许多处理器将会在更早一个周期对该指令取指，这是通过将从ALU计算所得到的分支地址直接输出到地址总线上（数据转发的一种形式）实现的，并同时将结果写入PC。

- 过去的行为
- 代码域/地址
- 编译器在代码中给出的暗示（例如执行/不执行指示位——TDTB[⊖]）

211 动态分支预测通常依赖一些过去的行为来预测未来的分支，这在框5.4中有总结。当CPU看到一个分支时，它通过预测器快速地决定执行哪条路径。然后，当实际的分支结果出来后，它会更新预测器，这是为了让未来的预测更准确一些。

我们将对7种不同的预测方法分别进行探讨，研究它们的操作和性能：

- 单T位预测器
- 双位预测器
- 计数器和移位器预测器
- 局部分支预测器
- 全局分支预测器
- G选择预测器
- G共享预测器

在这些小节之后，将会讨论混合使用以上技术（5.7.9节），然后讨论使用分支目标缓冲（5.7.10节）。

5.7.2 单T位预测器

在一个比较简单的单T位预测器机制中，当T标志被设置为1时表示需要执行分支，而当被设为0时表示不执行分支。在执行完每条分支指令后（即所有的条件及其他相关因素都被解决后）会更新T标志。全局的T位预测器硬件开销很小，因为整个CPU只有一个T位执行预测。

无论何时遇到分支指令，流水线都会根据T位的状态进行预测。换句话说，如果上一个分支指令执行了分支（T=1），则下一个分支指令也被预测为需要执行分支。相反，如果上一个分支指令没有执行（T=0），则下一个分支指令被预测为不执行分支。这种预测器制并不是智能的，却有令人意想不到的好效果，特别是在编译器支持的情况下。从根本上说，当需要执行的代码存在大量的简单循环时，这是一种好方法。

例如，以下是一段ARM风格的汇编代码，初始时，R1=1，R2=4：

```
i1    loop:    SUBS R2, R2, R1     ;R2 = R2 - R1
i2             BGT loop            ;当结果大于0时，执行分支
```

现在在一个具有全局T位预测器的CPU上执行这段代码，来验证这种预测器在应对这种简单循环时表现得有多优秀：

记录	*i1*	*i2*	*i1*	*i2*	*i1*	*i2*	*i1*	*i2*
R1	1	1	1	1	1	1	1	1
R2	3	3	2	2	1	1	0	0
T位	—	1	1	1	1	1	1	0
分支	—	执行	—	执行	—	执行	—	不执行
212 正确	—	—	—	是	—	是	—	不

⊖ 执行/不执行指示位（take/don't take bit，TDTB）通过智能编译器插入在代码中，用于告诉预测单元在当前位置上最有可能的分支结果。注意到编译器比分支单元更清楚分支的情况——编译器可以“看到”未来，知道循环、函数、程序的整体内容，也知道在每条路径上下一条指令是什么。

从这个跟踪表（trace table）[1]最左列开始，当 i1 执行完（减法）后，R2 的内容由 4 变为 3。在下一个周期，当 i2（分支指令）执行完后，因为 SUBS 指令的结果大于 0，所以执行分支。在第一次循环中，由于预测器初始条件不定，所以不能进行预测。

正如所跟踪过程所示，循环在重复执行两次后退出（在最后一次循环后没有跳回循环的起始处）。在第二次循环中，预测器学习到上一个分支被执行，所以正确地预测出下一个分支需要执行。同样，在第三次循环中也预测正确，而在最后一次预测中错了。

总体而言，该循环中第一次碰到分支时有可能是不正确的，这取决于之前执行代码的 T 位预测器状态。最后一次分支也没有预测正确，但在循环体中，不管重复多少次循环，预测都正确。对于任意大小的简单循环，这种结论都是正确的：不管 i1 和 i2 之间存在多少代码，只要不包含分支指令，预测的结果都会如此。

请憾的是，循环极少会如此的简单，通常情况下在循环体的代码中都会包含其他分支指令。让我们通过另一段简单例子来说明这个问题：

```
i1   loop:   SUBS R2, R2, R1    ;R2 = R2 - R1
i2           BLT error          ;如果结果小于0，则执行分支
i3           BGT loop           ;如果结果大于0，则执行分支
```

我们再次通过一个带有一个全局 T 位预测器的 CPU 执行这段代码，以检测这种预测器遇到这种代码时的预测情况。这一回，我们假设初始条件为 R2 = 3，这是为了减少跟踪表中列的数量。需要注意，表中用于预测的 T 位值是分支指令前一列的值，因为每一列显示的是每条指令执行后的状态：

记录	i1	i2	i3	i1	i2	i3	i1	i2	i3
R1	1	1	1	1	1	1	1	1	1
R2	2	2	2	1	1	1	0	0	0
T 位	—	0	1	1	0	1	1	0	0
分支	—	不执行	执行	—	不执行	执行		不执行	不执行
正确			不		不	不		不	是

在这种情况下，预测器的性能很差；预测器的每一个预测都失败了。遗憾的是，这种结果对于简单 T 位预测器来说太平常了。正如我们在下一节将要看到的那样，对这种预测器加以改进：对每一次预测增加一些复杂度，为每一个分支采用单独的预测器。首先，介绍一下双位预测器。

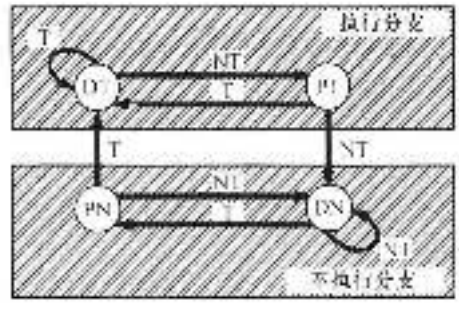


图 5-12 双位预测器状态转移图，其中有 DT（确定执行）、PT（可能执行）、PN（可能不执行）以及 DN（确定不执行）四种状态。前两种状态表示转移可能会执行，后两种状态表示转移可能不执行。在分支条件确定后，预测器的状态根据实际执行（T）或不执行（NT）分支来更新状态

213

5.7.3 双位预测器

虽然双位预测器从概念上讲与 T 位预测器有点类似，但却是用前两次分支的结果来对下一个分支进行预测，而不是像 T 位预测器那样只用前一次分支的结果。这种方法使用了如图 5-12 所示的状态机。这种方法也称

[1] 这种跟踪表并不能完全取代码的表，因为它既不能给出在某一个固定时间流水线发生的情况，也不能描出每条指令需要多少个周期去执行。它只能给出每条指令在顺序执行后系统的状态。

为双模（bimodal）预测器。

由于有4种状态（使用两位来描述状态），这种预测器将会比单T位预测器精确。这么说太笼统，但在一个嵌套循环的例子中（此时单T位预测器的性能非常差），双位预测器获得了更好的性能。

为了说明这种情况，我们使用与前一节相同的代码：

```
i1    loop:    SUBS R2, R2, R1     ;R2 = R2 - R1
i2             BLT error           ;如果结果小于0，则执行分支
i3             BGT loop            ;如果结果大于0，则执行分支
```

这一回我们将使用一个带有全局双位预测器的CPU来执行这段代码。我们再次假设初始条件为R2=3，R1=1，并且预测器初始状态为“DT”：

记录	*i1*	*i2*	*i3*	*i1*	*i2*	*i3*	*i1*	*i2*	*i3*
R1	1	1	1	1	1	1	1	1	1
R2	2	2	2	1	1	1	0	0	0
预测器	DT	**PT**	**DT**	DT	PT	DT	DT	PT	DN
分支	—	不执行	**执行**	—	不执行	执行	—	不执行	不执行
正确	—	不	**是**	—	不	是	—	不	不

214

这个跟踪表与5.7.2节的很相似，但需要精确地阅读这个表。记住每一列显示的是这一列所指示的指令执行完毕后处理器的状态，并且在表中没有时序上的信息，只给出了简单的操作序列。例如，先找到*i2*执行的位置，在这列中我们看到左边的R1和R2没有发生变化，但由于分支没有被执行，它将预测器的状态从“DT”改变为“PT”（以黑体字显示）。当*i3*执行完毕时，由于分支被执行（以黑体字显示），预测器的状态又转回“DT”。在*i3*开始执行时，预测器的状态还是“PT”，因此预测器预测分支需要执行，而事实上这是一个正确的预测。因此在底部以黑体字显示的“是”表示预测正确。因此，在确定预测正确性时，需要将预测的结果与前一列的预测进行比较。

虽然这种预测器对于所有的分支不会都预测正确，但它对循环体内不在循环末端的其中一个分支的预测都是正确的。这样的结果介于单T位预测器和理想预测器之间。

接下来更严密地解释这种预测器的原理：单T位预测器存在一些问题，而双位预测器在一定程度上可以解决这些问题。但如果双位预测器也有问题，可以增加更多的位来解决问题吗？答案是肯定的，因为总体而言使用更多资源会获得更好的性能。然而，从需求上来说，需要使用尽量小的硬件资源来提高尽量高的性能。

由此，我们需要意识到，通过分支指令*i2*的结果来预测*i3*的结果是非常困难的。用过去*i2*的结果来预测*i2*，用过去*i3*的结果来预测*i3*，才会得到更好的结果。换句话说，需要为不同的指令分配单独的预测器。事实上，这种机制将会在5.7.5节介绍双模预测器时碰到。然而，接下来我们先对使用更多位的预测器进行探讨。

5.7.4 计数器和移位器预测器

简单的饱和计数器（saturating counter）在执行分支时加1，在不执行分支时减1。这种计数器可以保持在饱和状态而不会循环计数，因此经过一长串连续的执行分支后，可以使得这种计数器达到最大值并一直保持这个最大值。

对于基于这种计数器的分支预测器，就可以只通过最高有效位（MSB）的值来判断。因为

当MSB为1时，计数器的值为其最大值的一半或一半以上；MSB为0时，计数器的值小于其最大值的一半。

虽然计数器是一种非常简单的硬件，但是它需要很长一段时间去“学习”在什么时候执行“通常执行分支”（normally-taken）循环与“通常不执行分支”（normally-not-taken）循环间的转换。另外，它在嵌套循环中带有分支时性能不好。 215

与计数器在硬件尺寸上类似的是移位器。一个n位移位器存放着过去n个分支的结果。无论什么时候处理器执行了一条分支指令，其结果都会进入移位器，同时此前的结果都会向右或向右移动一位，而最先进入移位器的结果就会被丢弃。例如，以1表示执行分支，0表示不执行分支，5位移位器中存放着最近5个分支的结果（NT，NT，NT，T，T，NT，T，NT），其值为00011010。如果下一个分支执行，移位器的内容将都向左移动1位，丢弃最左边的0，而1会进入移位器最右端，更新为00110101。虽然可以通过移位器的内容进行分支预测，但是极少单独使用以上我们学到的技术进行预测，而是将这些技术混合使用。至此，我们已经初步介绍了4种分支预测器。

5.7.5 局部分支预测器

简单或嵌套级语句会发现，有些分支总是会执行的，而有些则总是不执行。全局单T位预测器和全局双位预测器对CPU内所有分支的处理看起来都是一样的。更明智的方法是局部处理每个分支，而不是采用全局的方法。这与4.4.4节所提到的局部性原则有关：例如，假设系代码中的分支行为与用户代码中的分支行为是不一样的，所以应该为它们采用不同的预测方法。甚至在用户代码中，不同区域的程序所包含的分支也是不同的。

正如之前所提到的，可以为每一个单独的分支分配一个单T位预测器（或者双位预测器）。然而，在一些代码中，可能会有几千条甚至几百万条分支指令，假如为每条分支指令都分配一个预测器，硬件开销就实在太大了。

所以在全局预测器和为每个分支分配一个预测器之间有一个折中，这也引出了预测器块（a bank of predictors）的概念。在一定程度上，这与cache（见4.4节）的硬件组织相类似。同时也会引发类似的问题：查找时间（look-up time）。使用这种机制时，每当遇到一条分支指令，预测器就会为这条分支指令进行查找，然后确定预测结果。如果需要进行搜索的预测器越来越多，查找的时间就会变长，甚至会超过1个流水线周期。因此，计算机体系结构设计师的重点是使用更少的预测器获得更智能的预测。

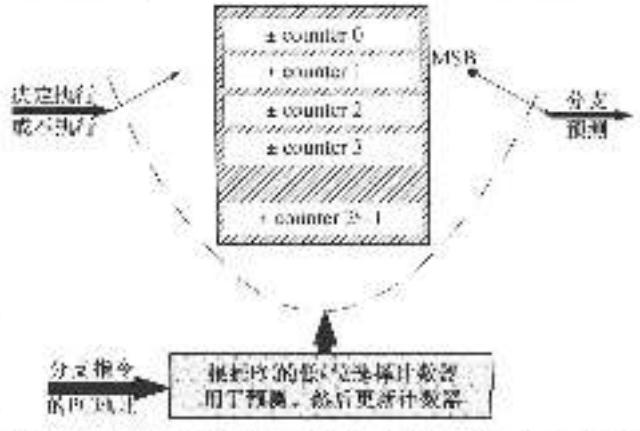


图5-13 局部分支预测器示意图。它带有一系列分支历史记录和计数器。它们使用给定分支指令所在地址上的低位地址进行索引（由此不同的计数器映射到了不同组的分支指令）。正如在5.7.4节里解释的，计数器的最高有效位用于进行分支预测

图5-13是分支历史记录和计数对的示意图。它没有为所有的分支指令都分配一个计数器预测器，而是只有2^n个计数器预测器，正在预测的分支指令分配在不同的地址上。 216

这种预测器使用地址线[⊖]的低 k 位来选择使用哪个计数器为该分支指令做预测（该计数器也理所当然被该指令的分支结果更新），例如地址0上的分支指令使用0号计数器，地址1上的分支指令使用1号计数器，以此类推。如果只有8个计数器，0号计数器就会由地址0、8、16、32、64等上的分支指令共同使用。

注意预测器块中的饱和计数器可以使用单T位预测器或双位预测器来替代。最重要的是引入了局部性原则：至少有一部分预测都是基于地址的。接下来我们使用之前在全局单T位预测器和双位预测器使用过的代码来说明这种局部性预测：

```
i1   loop:   SUBS R2, R2, R1    ;R2 = R2 − R1
i2           BLT error          ;如果结果小于0，则执行分支
217  i3      BGT loop           ;如果结果大于0，则执行分支
```

这回我们使用带有如图5-13所示的局部预测器的CPU来运行这段代码。我们仍旧假设初始条件R2=3，而预测器的计数器为4位并且在执行前被初始化为0111。指令 $i1$ 的位置为地址0。

记录	*i1*	*i2*	*i3*	*i1*	*i2*	*i3*	*i1*	*i2*	*i3*
R1	1	1	1	1	1	1	1	1	1
R2	2	2	2	1	1	1	0	0	0
c0	**0111**	0111	0111	**0111**	0111	0111	**0111**	0111	0111
c1	0111	**0110**	0110	0110	**0101**	0101	0101	**0100**	0100
c2	0111	0111	**1000**	1000	1000	**1001**	1001	1001	**1010**
分支	—	不执行	执行	—	不执行	执行	—	不执行	不执行
正确	—	是	不	—	是	是	—	是	不

这个跟踪表这回给出了预测器3个计数器（c0、c1和c2）的情况，由于代码从地址0开始，所以这3个计数器分别映射了指令 $i1$、$i2$ 和 $i3$ 的地址。在这个例子中，预测器的0号计数器c0并不会发生改变，因为在这个地址上没有分支指令去更新这个计数器。而其他两个计数器则被其映射地址上的分支指令的执行结果所更新。表中每次预测所用的计数器都用黑体字标出。

每一次分支预测都是通过检查与相关预测计数器的MSB来进行，该预测器来自当前指令的前一列（一如既往，每一列所包含的状态都是前边指令执行的结果，并且预测都是发生在指令执行之前）。

这种预测器的性能与之前所遇到的预测器有很大不同。第1条分支指令在每次循环中都被正确预测。第2条分支指令在第1次循环和最后一次循环中被错误预测，而在循环内部，无论重复几次循环或中间包含多少条非分支指令，都被正确预测。这相对于5.7.3节中的那个例子有不少改进。

遗憾的是，故事到此并没有结束，因为当预测器变得强大后，就会受如框5.10所示的混叠效果（aliasing effect）影响。

框5.10 局部预测的混叠

在一个拥有4组的局部3位饱和计数器预测器的处理器上执行以下汇编代码：

```
0x0000    loop0    DADD R1, R2, R3
0x1001             BGT loop1
0x1002             B loop2
...       ...      ...
0x1020    loop1    DSUB R3, R3, R5
0x1021             B loop0
```

⊖ 对于一些处理器如ARM，地址采用字节计数，但其指令超过一个字节。在这种情况下，由于指令地址都是0、4、8、16等，因此所有ARM指令地址线的A0和A1都永远为0。所以这些地址线将被局部预测器忽略，而使用从A2开始的地址线。

假设 R2 = 0，R3 = 2，R5 = 1，c0、c1、c2 和 c3 都被初始化为 011，且 loop2 所指的代码块序列：

地址	结果	分支	预测值	正确	地址	结果	分支	预测值	正确
0x0030	R2←2				0x1020	R3←0			
0x1021		执行	c2←100	不	0x1021		执行	c1←111	是
0x1029	R3←1				0x0006	R5←0			
0x1021		执行	c2←110	是	0x1021		不执行	c1←110	不
0x0009	R1←1				0x1030		执行	c2←100	是
0x1001		执行	c1←110	是					

在这个表中，所执行指令的地址显示在最左列，其次是执行该指令的结果（即寄存器发生的改变），第三列指出分支指令的分支是否被执行，第四列为本次预测所使用计数器的值，第五列指出预测是否正确。每一列给出一个结论。

总体而言，这种预测表现良好。然而，要考虑能提高预测的最后一点是，这里只使用了两个预测器。计数器 c0 被更新为分支指令很少被执行（即地址 0x0030 和地址 0x1020 上的分支指令）。因此，只能预测这两个分支指令的执行结果。为了更好地在不同分支中使用这些计数器，我们需要引入其他一些机制，其中两种我们将在 5.7.7 节和 5.7.8 节中描述。

5.7.6 全局分支预测器

基本全局分支预测器是尝试使用一种特殊的方法在基本局部分支预测器基础上进行改进，它将上下文引入分支预测器中。我们已经知道了分支预测的局部性，但是局部分支预测中的混叠问题使得在相同的预测器中不同类型的软件都有分支无效件。

在这个全局分支预测器中，使用了全局移位寄存器而不是低位地址位（这两种部件在 5.7.4 节中都有简单介绍）对计数器预测器阵列进行索引。其整体结构如图 5-14 所示，看起来与局部预测器很相似，除了我们将要进行讨论的计数器选择机制之外。

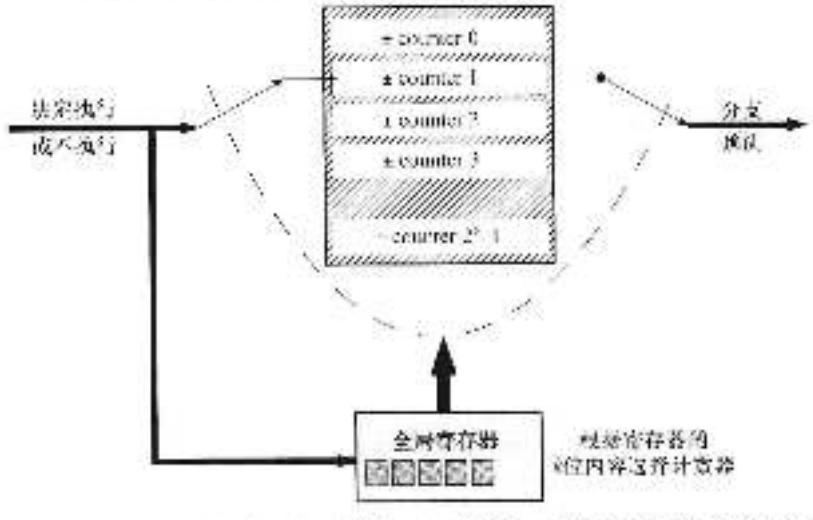


图 5-14 全局预测器示意图。它带有一组计数器，它们通过存储了前 n 个分支指令结果的移位寄存器的内容进行索引。与以往一样，当分支执行时，与之相关的计数器增加；当分支不执行时，计数器减少。计数器会饱和而不是循环，因此它们始终具有效值可用于预测分支。全局移位寄存器在每条分支结果确定后进行更新

由于被选择用来对特定分支进行预测的计数器的选择是基于过去 n 条分支指令的结果，因此

这种预测在一定程度上依赖的是这个分支指令是如何到达的，而不是这个分支指令位在内存的什么位置。换句话说，它更像一个基于跟踪的选择器。

在一些条件下，这种预测选择机制是很明智的：例如，一个简单的常规库函数会被一段代码中不同的地方多次调用，它被调用时的行为（相当于分支行为）取决于要求它做什么，即如何被调用（以及在哪里被调用）。通过观察许多常规软件的执行轨迹可以看到，许多复杂的分支序列都被重复执行。通过使用这种预测器，可以让分支序列来选择预测器，使得不同的计数器与不同的分支映射得更紧密。

我们通过另一个简单的例子来检查一下这种全局预测器的操作：

```
i1      loop1     ADD R1, R1, R2
i2                BEZ lpend
i3                SUB R8, R8, R1
i4                B loop1
220     i5        lpend     NOP
```

我们假设初始时 R1 = 3，R2 = −1，R8 = 10，并且这是一个 4 位全局寄存器（GR，初始为 0000），即有 16 个计数器预测器，每个为 3 位，被初始化为 011。

地址	结果	分支	GR	预测器	正确	地址	结果	分支	GR	预测器	正确
*i*1	R1←2		0000			*i*3	R8←7		0010		
*i*2		不执行	0000	c0←010	是	*i*4		执行	0101	c5←100	不
*i*3	R8←8		0000			*i*1	R1←0		0101		
*i*4		执行	0001	c1←100	不	*i*2		执行	1011	c11←100	不
*i*1	R1←1		0001			*i*5			1011		
*i*2		不执行	0010	c2←010	是						

以上表的结构与前面几节遇到的很相似，GR 的值在每一行都显示——只有一个 GR，并且在每个分支指令之后都被更新。虽然这个代码循环了 3 次，但最显著的特点是每一个分支都被分配到了不同的计数器预测器上。即使是在同一个分支中的每次执行使用的也是不同的计数器。

总体而言，这种预测器制避免了混叠问题，而分支指令在不同的计数器预测器中“混合”在一起，但遗憾的是，其分支历史被丢弃——而我们是可以用历史记录来很好地对 *i*2 特别是 *i*4 分支进行很好的预测的。

已经证实，在比这个小代码大很多的例子中，这种预测器的性能十分好：它执行基于循环的测试代码获得了前所未闻的超过 90% 的精确度。然而，其基本缺点还是存在的：局部性信息丢失。因此我们开始转向将基于行为跟踪选择的全局寄存器与基于地址的局部选择进行混合。

5.7.7 G 选择预测器

如图 5-15 所示，G 选择预测器（gselect predictor）通过考虑将要被预测分支的地址对预测器进行更新。事实上，用于为确定分支而选择特定计数器预测器（或 T 位或双位预测器）的 k 位索引是由一个 n 位全局寄存器与 PC 的最低 m 位联合组成的。

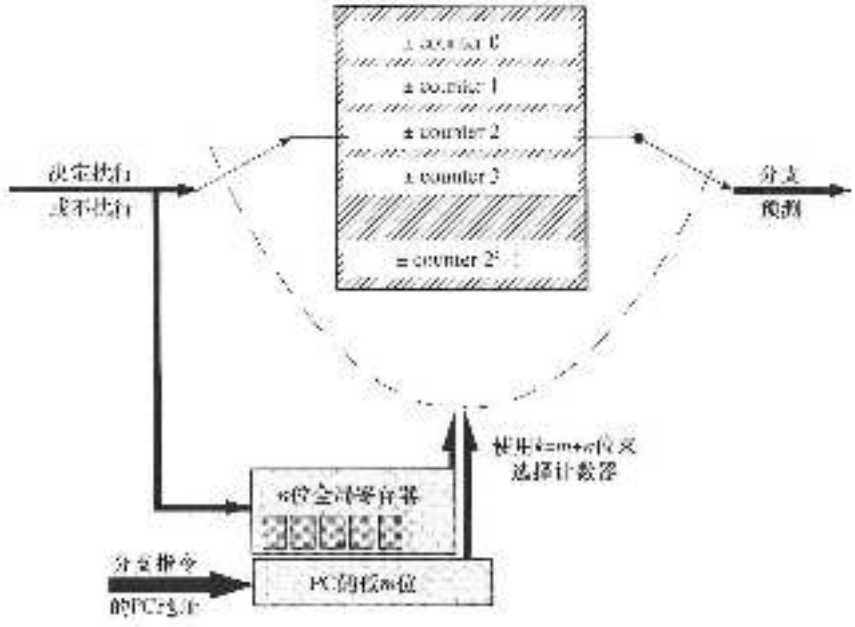


图5-15 G选择预测器示意图。它有一组计数器，通过存储了前n条分支指令结果的移位寄存器的内容与地址线低m位共同进行索引。同样，每个计数器在分支执行时增加，在分支不执行时减少。计数器会饱和而不是循环，因此计数器的最高有效位可以用于分支预测。全局移位寄存器在每条分支指令确定是否分支时被更新

例如，如果$k=10$，则由一个4位全局寄存器G及地址线A的6位组成，其10位索引如下：

G_3	G_2	G_1	G_0	A_1	A_0	A_3	A_2	A_1	A_0

[221]

据报导G选择预测器非常适合那些小的独立预测器块，这可能是指它非常适合于那些资源受限的嵌入式系统，而当块变大，也许为$k>8$时类似于下一节将会讨论的G共享机制，G选择预测器的性能可能会更好。[1]

5.7.8 G共享预测器

G共享预测器是5.7.7节中G选择预测器的指标。将如图5-16所示的G共享预测器与如图5-15所示的G选择预测器进行比较，唯一的区别是G共享预测器将k位全局寄存器与PC的低k位进行了异或，用于对预测器阵列进行索引。 [222]

G共享预测器与G选择预测器和全局分支预测器一样，只要进行适当的设置和调节，都可以获得超过90%的预测准确度。然而，G共享预测器和G选择预测器都是在块相对较小的情况下才能获得好性能。小尺寸块（即更少的预测计数器）意味着查找过程可以变得很快。除了特别小的块以外，G共享预测器比G选择预测器性能更佳，因为它能更好地将不相关的分支指令分布到不同的预测器上。换句话说，G共享预测器可以看到分支在计数器上的分布，而G选择预测器只能看到少数计数器在许多分支指令上的划分。

---

[1] 回忆一下前面的讨论，高性能取决于很多因素，而不是某一段特定的代码。当我们从整体上预测性能时，只能在真正的代码及条件能够测试以某种机制的代表性。

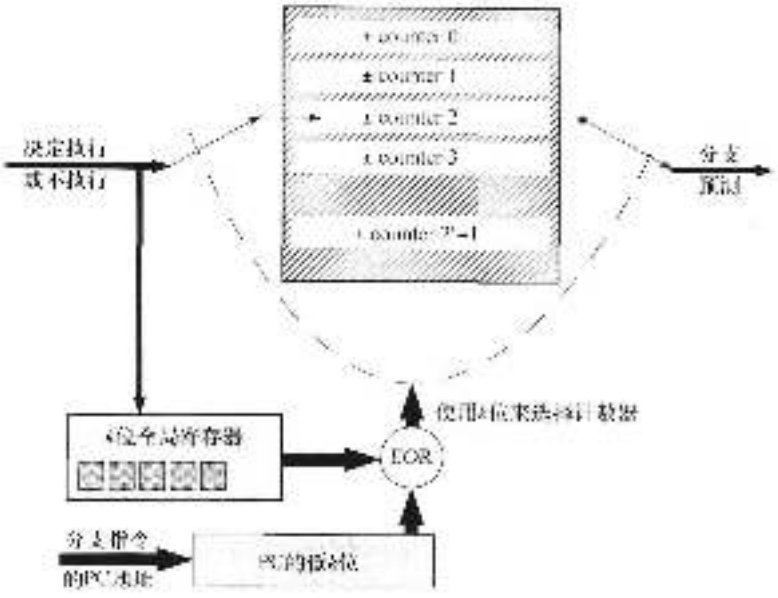


图5-16 G共享预测器示意图。它也拥有一组计数器，通过存储了前k条指令执行结果的移位寄存器的内容与地址线低k位共同进行索引（根据两个k位数字异或的结果选择计数器）。同样，每个计数器在分支执行时增加，在分支不执行时减少。计数器会饱和而不是循环，因此计数器的最高有效位可以用于分支预测。全局移位寄存器在每条分支指令确定是否分支后被更新

5.7.9 混合预测器

我们应该可以强烈地感觉到，不同程序中的分支特性应该会千差万别。直到现在，我们已经提出了许多预测机制并且也讨论了它们各自的优缺点。

223

现在我们的重点是要找出一种性能最好的分支预测机制。但是，通过学术界对这些机制独立进行的测试发现，它们都是分别对某一类代码会获得良好的性能。因此，我们应该将这些预测机制进行组合。

这正是所谓的混合预测器。它是将多种预测机制集合在一起，预测时通过一个逻辑选择最合适的预测机制。图5-17展示了在A和B两种预测机制中进行选择的方案（与5.7.5节的双模预测器看起来有些类似）。在这个方案中，A/B选择器用于记录A、B两个预测器预测的准确性。哪一种预测器预测的结果最准确，就选择其作为整个系统的预测器。

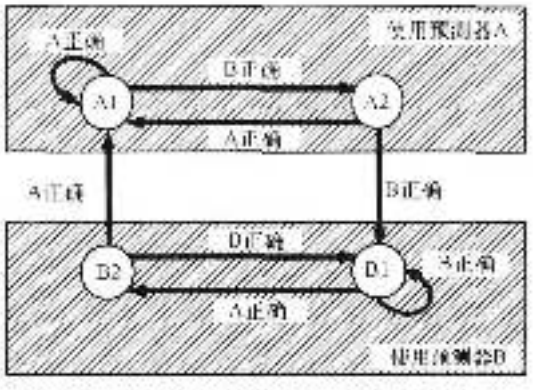


图5-17 两种不同的预测器。它们各自的特点结合某种特定代码类型的需求，可以将这两种预测器结合起来。结合的方法之一是采用一个与双位预测器十分相似的双位状态机，来选择最佳的预测方式。在这个状态机中，如果两个预测器在任何状态都预测正确，我们就认为不发生状态转移

我们可以预期，不同的程序，或程序中不同的区域，会趋向于不同的预测器，而这在实际测试中也得到了验证。

一个最著名的混合预测器的例子是建立在

Alpha 21264处理器上的。图5-18是其示意图。在该图中，所示的A/B预测器用于选择全局预测器或两层局部预测器。

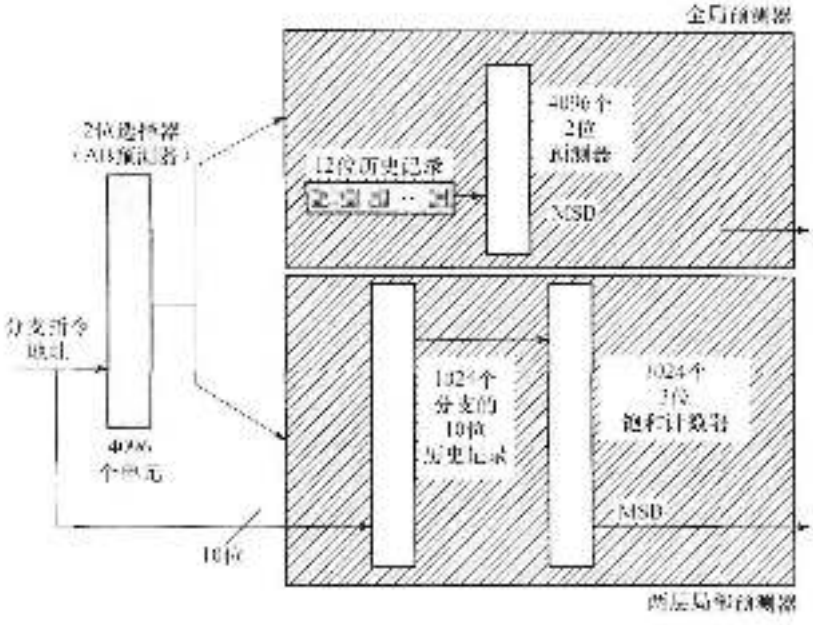


图5-18 Alpha 21264处理器混合预测器示意图。它使用一个与图5-17所示的A/B预测器结构相似的状态机（最左边的模块）来选择是使用全局预测器还是两层局部预测器，从而获得优秀的预测性能

全局预测器使用12位分支历史记录器在4096个2位预测器中进行选择。这个预测器精确地预测了分支行为。换句话说，它真实地反映了最终何时达一个特定分支语句的（参见5.7.6节）。

局部预测器使用地址线的低10位在1024个10位移位寄存器预测器中进行选择。这个移位寄存器预测器是一个局部版的全局预测器。它记录了发生在当前的低10位地址上的所有分支历史。 224
注意不要误以为地址线和移位寄存器都必须为10位，它们可以有不同的位宽。

移位寄存器的值将用于在1024个3位饱和计数器（分别预测不同的分支）中进行选择。预测的结果是这些计数器的最高有效位（MSB）。

Alpha 21264中的预测器使用了多层结构（针对局部预测）以及在两个不同的预测器中进行动态选择，它像是结合了我们至今所讨论过的所有预测机制。

然而，我们还要检验它的性能。在CPU中给定一个有限的空间来支持分支预测，我们应该考虑这个空间适合使用哪种预测机制或者是用来改进流水线的其他方面。

这个问题在1999年给出了回答，在这一年中数字设备公司（DEC）的Alpha 21264处理器的分支预测单元被设计出来。对它的测试表明，混合方法的性能超越了相等规模的全局预测器和相等规模的局部预测器。事实上，这个处理器的分支预测器在实际代码中获得了令人惊讶的98%正确性——这是一个即使在最先进的处理器上也很难被超越的成绩。 225

5.7.10 分支目标缓冲

正如我们在前几节所看到的，分支预测器可以精确地知道是否执行某个特定的分支。回到我们为什么需要预测分支的原因上，这是为了提高所预测的执行代码是正确代码的几率，降低清空流水线的可能性。

我们需要预测执行代码的主要原因之一是，当一个分支执行时，其目标地址是按相对偏移存储的，需要通过ALU将偏移量与当前PC值相加来得出。这个过程需要使用ALU，而在处理器上并没有为地址计算设计独立的ALU。地址计算需要在分支指令进行到"执行"阶段时共享流水线上的ALU，这我们已经在5.2.6节中做过讨论。

然而，即使我们正确地预测了某一个分支是否会执行，我们还是需要进行这个地址计算。换句话说，我们可以很快地进行预测，但是我们需要等待计算完成（或至少两者同时进行——这样就是等待两者之间的最慢者）。

所以计算机体系结构设计师会想出这样一个巧妙的想法：为什么不将目标地址存储在预测器上？与其简单地预测执行分支或不执行分支，为什么不预测整个目标地址？毕竟，只有唯一一个地址分支指令会分支到，而如果我们能够记录分支行为，那么我们也能够轻松地记录下分支的地址。

而这就是分支目标缓冲（BTB）所要做的。

使用BTB意味着如果我们预测正确并且当前分支已经至少被执行过一次，则不需要等待
226 ALU计算出分支目标地址。图5-19给出了BTB的流程图。当运行分支预测时，我们首先查看BTB，如果BTB命中（当前分支指令在BTB中有一个记录，即之前已经遇到过该条指令），则将BTB中的目标地址装载进PC，然后从该地址获取。

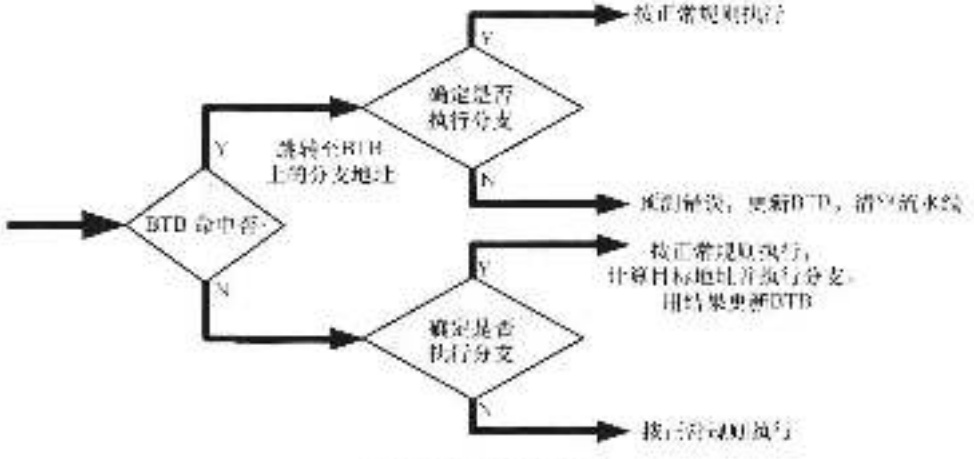


图5-19 BTB流程图

一旦确定是否执行分支（对于非条件分支为立即获得，而对于条件分支则要等到条件设置指令执行完毕），我们就知道是否要继续这个预测的执行，或是要清空流水线，然后更新BTB并取出正确的指令。

而如果BTB不命中，则预测分支不执行。当分支结果确定时，如果应当执行，则将目标地址更新至BTB，如果已经预测执行还应清空流水线，然后分支至正确的地址继续执行。

实际上，如图5-20所示的BTB中的内容看起来与内存的cache（4.4节）很像，以分支指令所在的地址作为标志。一个记录存储了分支预测（可以使用我们之前讨论过的任何预测器制）和目标地址。像cache一样，BTB也可以为全关联、组关联，或者更为奇特的关联方法。

然而，BTB的讨论并没有到此为止，还有一个重要的改进：想一想在CPU分支到目标地址时发生了什么——它将目标地址处的指令装载进流水线。在流水线中对它进行译码和执行时，之前的分支指令已经执行完毕，所以此时知道该条指令应该被继续执行还是被清出流水线。

227 但是我们可以提早完成这个过程，通过将这条指令而不是这条指令的地址存在BTB中。这样在BTB命中时，就可以直接把所保存的指令送入流水线而不再需要经过取指阶段。



图5-20 BTB结构图。它的组织与cache相似，而实际上它所完成的功能也与cache相似，是为了减少访问它所存储的指令的时间

5.7.11 基本代码段

对于5.7.10节介绍的BTB技术还有一些改进值得注意，这些改进是针对一段代码而不是独立的指令而进行的。实际上，对单独指令的移动也会涉及一整段代码的修改。在计算机体系结构领域和软件体系结构领域中常使用以下3种类型的代码段：

- **基本代码段**（basic block）：是指按顺序执行没有分支进入或分支出去的指令序列（即只有一个入口和一个出口）。
- **超级代码段**（super block）：是指基本代码段的一个执行轨迹，但其只有一个入口而可能有多个出口。
- **特级代码段**（hyper block）：是指与超级代码段相似的多个基本代码段集合，其只包含一个入口而可能有多个出口。与超级代码段不一样的是特级代码段有多条执行轨迹（即多个控制路径）。

在本节中，我们将讨论范围限制为最为简单的基本代码段，为基于代码段的BTB方法所采用。想象一下BTB，或者内存的cache，可以存储一段指令并将它们送入流水线。对于支持指令重排（re-order）或乱序执行的流水线，这种方法可以获得最大的灵活性及最佳性能提升。

基本代码段可以由分支指令及其目标地址之间的一串指令组成，而程序的执行轨迹可以由一系列基本代码段的连接图来标识。图5-21展示了一个由多个基本代码段组成的路径图。

首先我们预测了分支是否会执行，其次我们也预测了分支的目标地址，最后我们预测了目标地址指令。现在我们可以预测基本代码段。

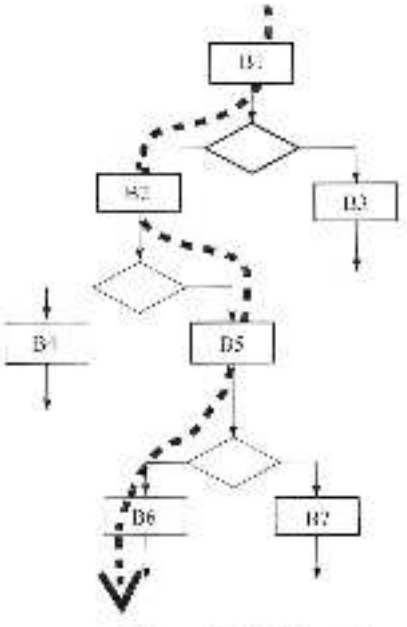


图5-21 执行一个程序时将一组基本模块互联起来的路径图

通过标定出基本代码段的指令，我们可以将其缓存起来并快速发送。例如，参考图5-21，一个BTB块可以直接将包含了B1、B2、B5和B6的指令发送至流水线而不需要分支——假设我们已经正确地预测了这些块的路径。

当然，我们仍需要检查我们所预测的分支行为是否正确，且在预测错的时候仍需要清空流

水线。在实际代码中，可能包含了多个基本代码段（BB），每个代码段可能包含数十条指令（平均每个BB的大小约为7条指令，但是对于不同的计算、不同的处理器及编译器是不同的）。

跟踪cache随着时间的推移不断更新，并且每当CPU命中根BB（B1），分支预测算法就要对未来路径进行预测。如果此预测与跟踪cache的第二个入口（B2）相符，即为命中，CPU开始
228 执行来自跟踪预测的基本代码段内容（这些指令也被存储在cache中）。

这种系统在奔腾4处理器上得到了应用，但不同的是，在奔腾4上的系统不缓存BB内的指令，而是存储器译码后的指令内容，即我们不仅可以绕过流水线中的“取指”阶段，而且可以绕过“译码”阶段。

5.7.12 分支预测总结

在前面几节里，我们花了很大精力对如何保持流水线不间断地执行指令进行了讨论。所有花费在分支预测、加速等方面的努力都是为了尽可能快地发送指令。流水线以及指令级并行都是保证指令能够快速且高效执行的方法。

我们需要注意的是，没有一种独立的技术是最佳的，只有合理地选择多种技术并让它们相互融洽地配合才能获得最理想的性能。

还有一点需要注意，硬件的加速少不了优秀的编译器支持[⊖]，将一部分努力花在搭建一个好
229 的编译器上比把所有努力都花在硬件提速上可以获利更多。

5.8 并行机器

2.1.1节介绍了Flynn的处理器分类方法，其根据处理指令和数据的方法将处理器分为了4组，分别称为：

- SISD——单指令流单数据流
- SIMD——单指令流多数据流
- MISD——多指令流单数据流
- MIMD——多指令流多数据流

总的来说，我们讨论至此一直都只考虑SISD机器——在嵌入式系统及台式机中常见的微处理器。我们也讨论了一些在MMX和SSE单元中（见4.5节）以及在一些ARM专用协处理器（见4.8节）上遇到的SIMD要素。我们将跳过MISD，其大部分都应用在容错系统（fault-tolerant system）上，例如对数据进行多次计算然后比较每次计算的结果——这些将会在7.10节中进行更深入的讨论。所以接下来我们将讨论SIMD和MIMD。

在写此书的时候，处理器产业已经由SISD扩展至SIMD和MIMD。MIMD因此变得越来越流行。我们在4.5节中已经讨论了一些常用的协处理器，它在主CPU上扩展一些功能单元用于执行各种特殊功能。这里，我们将更进一步讨论在MIMD结构下多个相同的处理器一起并行工作。

准确地说，在计算机里有不同层次的并行，而“并行机器”是一种比较广泛的定义。让我们先来简单了解不同层次的并行：

- **位级并行**：与计算机的字长相关。一个8位计算机并行处理8位数据，而32位机器可以同时处理4倍于它的数据。
- **指令级并行**：是一系列技术的集合，它们允许同时执行多条指令。正如我们已经看到的许多例子那样，不同的指令可以同时重叠执行，只要它们之间没有数据相关。流水线就是这样一个例子，而超标量机器、协处理器以及Tomasulo算法（见5.9节）也是指令级并行的例子。
- **向量并行**：与SIMD机器相关，它们处理的是整个向量中的数据而不是单一字长的数据。

⊖ 作者个人推荐使用GCC（GNU Complier Collection）。

SSE 和 MMX 都是这种类型并行的例子。

- **任务并行**：是指整个任务或程序子程序和函数可以同时被不同的硬件执行。我们将在本 230
  节对其进行讨论。
- **机器并行**：是指大型公司如 Google、Amazon 所采用的服务器场（server farm）。它们包含了上百甚至上千台单独的计算机，每台计算机都为某一个特定任务而并行运行。我们将在 9.3 节对这种系统进行讨论。

图 5-22 展示了这些不同层次的并行，概括地表示了从通过指令按位操作向更高层次并行的发展。

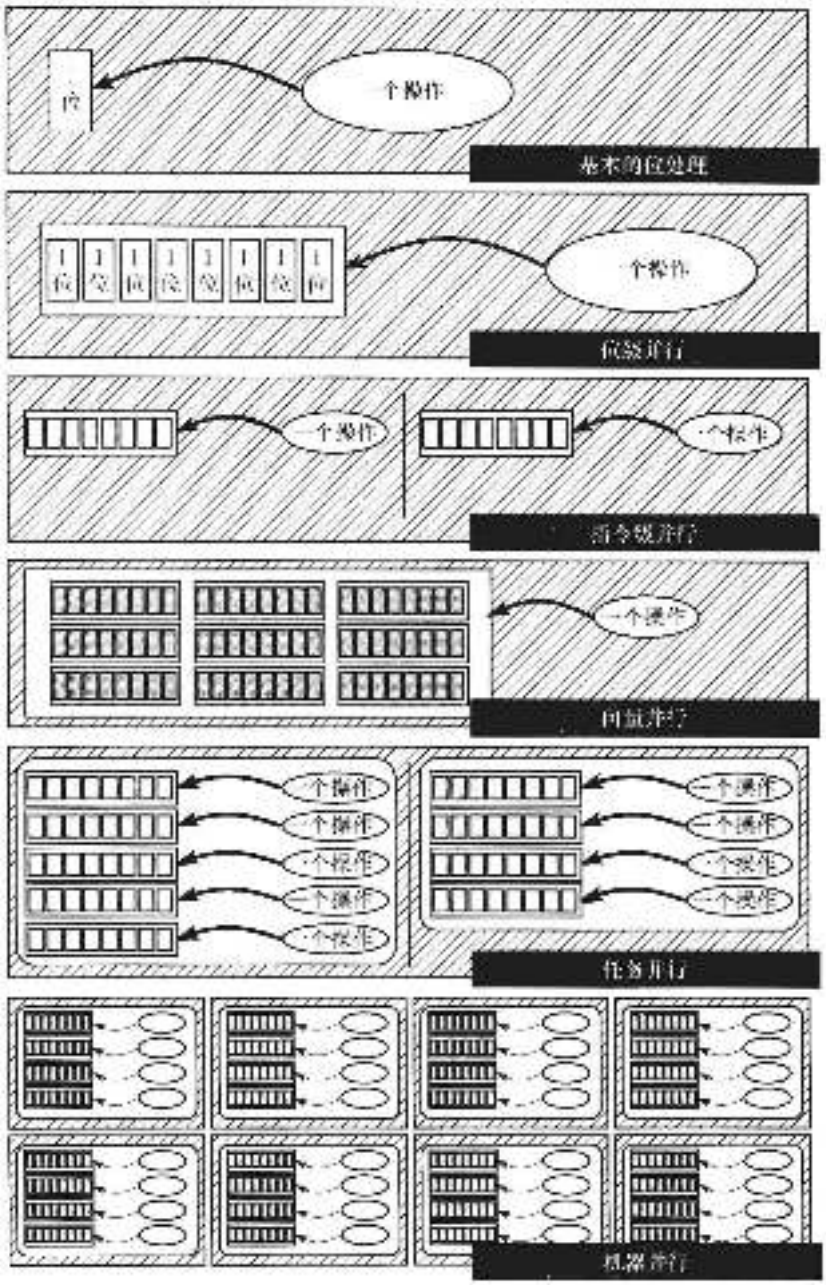


图 5-22 不同层次并行示意图。展示了从最底层的按位操作不断向高层次并行的发展，这种发展建立在不断重复最底层的并行操作之上

在讨论并行处理的同时，指出什么需要“联合”（coupling）并行执行是很必要的。广义的联合（loosely coupled）并行处理指不同的并行执行线程没有相关性，可以独立地执行。这种任务可以很容易地被独立的并行处理器核分别执行。例如，有两个来自不相关用户的Google搜索请求，在Google服务器堆里分别由两台机器响应。另一方面，紧密的联合（tightly coupled）任务之间的相关性则很强。它们可能需要共享数据，经常进行通信，并且出现一个任务与另一个任务相关的状况。如果将两个任务都放在同一台机器上执行，这样两台机器之间的通信就不会成为性能的瓶颈。自然，机器的体系结构与这些任务是否匹配就变得不太重要了。

对于计算机体系结构，并行中常用的大都是那些我们之前所讨论的顶尖结构。我们已经涉及了大部分计算机种类，并将在9.3节讨论大规模并行机器，但现在，我们将讨论的是任务并行这一层次。这种并行比超标量和向量计算机的层次要高，比机器并行要低。任务并行在台式机领域里变得越来越重要，并且逐渐影响着嵌入式领域。

在并行中有两个重要的演变，我们将依次进行讨论。第一个是由带有附加功能单元的SISD向MIMD的演变。第二个是为了提高性能而采用并行。我们将在接下来的几个小节里对这两个演变进行讨论。

5.8.1 SISD向MIMD的演变

写一个SISD的程序是比较简单的——从程序员的角度，在任何时刻通常只需考虑一件事情，并且程序在分支之后都是按顺序执行。在早期的程序存储计算机上，设计人员所愿意看到的是：装载今天的程序然后执行它，明天再装载不同的程序然后执行。改变任务只需更换不同的打孔卡即可。

231
~
232

然而，就在这10年间各类计算机一直在寻求能被人们广泛接受时，软件已经开始由以计算为主导（如会计计算、模拟计算、方程求解等）经由控制领域（如传感器监控、自动化控制等）转向解决复杂的多任务，通常包括智能感知（multi-sensory）、沉浸式人机界面（immersive human-computer interfacing）等。

以往的计算机只需要在一段时间内完成一个任务，而现在的计算机（包括台式机和嵌入式系统）几乎都需要同时处理多个并发任务。这些不同的任务都包含了不同的时间和操作需求。6.4节将对实时任务进行讨论，这里只需认为软件通常需要在不同的时间执行不同的代码段。程序的每一段代码都可以被当做是一个独立的任务，因此程序可以划分为不同的任务，完成不同的功能，它们可以在同一计算机上的不同时间内完成。

通常，这些任务都带有独立性，因此通常不同任务间会产生需求冲突。一般而言，当面对两个（或多个）冲突需求时，系统体系结构设计师通常会将系统划分为多个部分，让硬件和软件的不同部分分别满足多个需求。划分通常是针对软件的：让软件的两个部分分别处理两个不同的方面，但共享CPU资源。然而，硬件也可以被划分，如让两个处理器分别处理一个任务。

可以用一个简单的例子来说明需求冲突，即台式机运行一个使用鼠标控制的视窗桌面显示，同时运行一个MP3后台播放系统。在这个例子中，MP3后台播放系统需要进行数学计算来处理音频流，而音频的各个采样需要及时地输出，任何一个采样被延时都会产生“喀嚓”声甚至是噪声。系统的设计人员已经意识到这一点，提高了MP3后台播放的优先级提高使其能够在任何时候都可以运行而不需要等待别的任务运行完毕。遗憾的是，用户由此通过鼠标去控制播放器时会发现鼠标指针移动不流畅。解决的办法就是将鼠标指针的优先级设为比MP3播放器高，或更好的方法是，采用动态优先级系统。

然而还可以有第三种选择：使用MIMD系统，它允许一台机器包含两个（或多个）指令流和数据流，并同时处理它们。由此，MIMD就不需要分时共享单一的处理器资源，但由于两个

（或多个）处理器在同一个芯片上，因此需要共享内存和外设，可以高效地处理多个任务。

图5-23显示了几种可供选择的硬件，给出了一个基本的SISD处理器、一个共享内存的MIMD处理器和一个SIMD处理器。这个SISD处理器拥有一个ALU、一个乘法器、一个I/O模块、一个内存单元、一个控制单元和一个取指/译码单元（IDU）、4个寄存器与内部总线相连，给定两个软件任务在这个SISD处理器上运行，需要各自的时间片。在图5-23b中，加入几个额外的功能单元运行转变成了SIMD处理器，它可以对多个数据同时进行计算——本此允许将两个 233
任务混合在软件里同时运行。由于这种处理器并不是SISD处理器的完全升级，因此内部总线布局就不会的瓶颈。在图5-23c中展示的是一个共享内存的MIMD处理器，每个独立的CPU都有独立的总线系统，是真正意义上的并行。它在一个芯片上包含了两个完整的处理器核，但是共享外部存储也成了它的瓶颈。

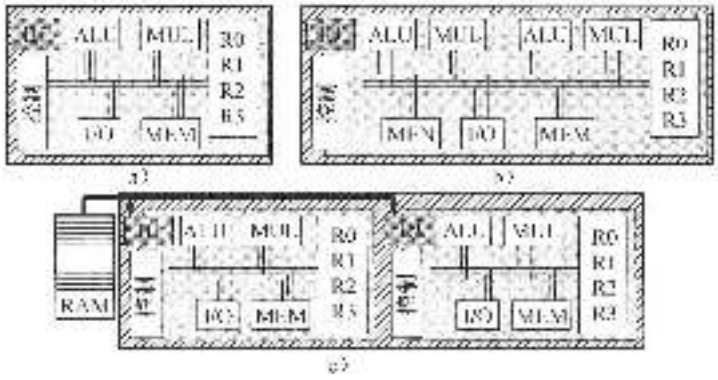


图5-23 几种处理器结示意图

a）一个基本的SISD处理器，包含四个功能单元（ALU、乘法器、I/O模块和内存模块）、一个控制单元和一个取指/译码单元（IDU），它们都与一组寄存器相连；b）一个SIMD处理器，它增加了额外的额外功能单元；c）一个完全共享存储的MIMD处理器，它在一个芯片上包含了两个完整的处理器核

与将软件划分为独立的线程时一样，处理器设计人员也遇到了时钟频率、数据位宽等限制，下一个性能的提高方向转向了提高并行度——这正是由SISD到SIMD最后到MIMD的历程。

在嵌入式计算领域，ARM946双核平台（Dual Core Platform，DCP）是一个重要的双核解决方案。它将两个ARM9核集成在一个芯片上，共享存储接口和片上通信接口。图5-24给出了这个处理器的结构图。

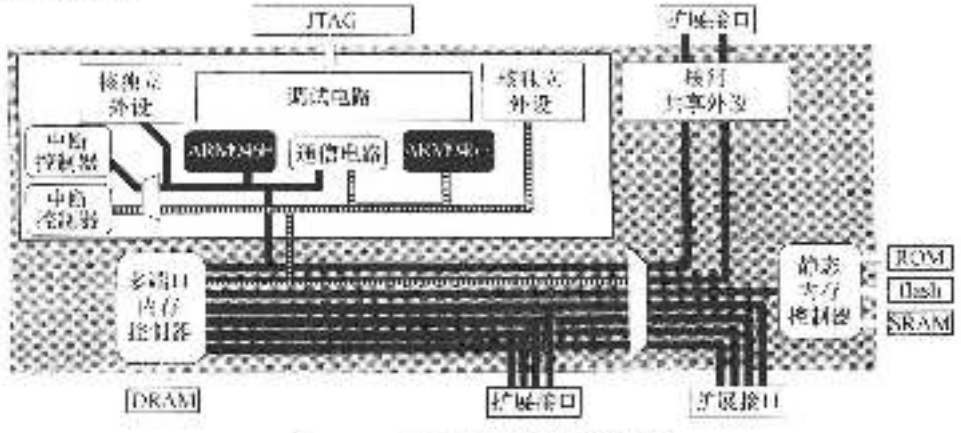


图5-24 ARM946双核平台的模块图

这个处理器称为广义联合（loosely-coupled）、前集成（pre-integrated）双核结构，它在硬件上支持同步调试和程序跟踪。大部分的软件和固件都支持这个系统，并且支持ARM9的操作系统也能在这个处理器上运行。这些软件的支持包含了可以让不同的软件线程（任务）运行在不同
234 的处理器核上，通过硬件通信接口（图中标为“通信电路”）进行仲裁。

虽然这里通过ARM946介绍并行计算，但它的本质仍是一个双核处理器而不是一个并行机器。两个处理器核相比多个独立单元更容易进行同步，并且在此例中核独立外设都被设置了两份。在写此书时，四核的Cortex-A9处理器也即将发布，这也将成为嵌入式并行处理里程上划时代的一步。

由于这些处理器都是针对嵌入式应用而开发的，一种可能的系统划分为，用户界面代码运行在一个处理器核上，每当有用户调用时就会被触发运行，而媒体处理（有实时性）运行在另外一个处理器核上。或者对于一个基于无线局域网的音频设备，一个核运行MP3解码，另一个核运行Ethernet处理。

不管面对什么样的应用，这种双核处理器都将逐渐成为主流。它们已经开始占据处理器市场。而在未来，更多的核将被集成在一起来满足不断增长的性能需求。

5.8.2 为提高性能而采用并行

我们已经提到了计算机设计人员为提高性能而面临的压力。虽然摩尔定律已经很好地得到了验证，但是消费者还是希望在他们的性能已经很高的计算机上获得更多的性能提升。

235 或许软件设计人员也已经学会了期待计算机的能力（还有内存大小）应该逐年增长。而计算机体系结构设计师也总是在抱怨程序员——这种抱怨根源于计算机发展史上软件设计人员与计算机设计人员的划分。大部分的计算机设计人员（也包括作者）都认为他们可以将软件设计得比程序员实际设计的更好。

不管这样的说法是否能够站得住脚，不断增长的软件规模（通常被计算机体系结构设计师认为是臃肿）和不断下降的运行速度，消耗掉了大部分通过体系结构发展、时钟频率提高、智能的流水线技术等所获得的性能提高。2009年典型的台式计算机的速度[⊖]就比作者10年前所用的计算机至少快50倍——但是网页的装载速度仍旧很慢，保存和读取文件仍旧很慢，加载操作系统仍旧耗费10秒左右的时间。但除了CPU本身的因素外，其他的外部因素也在影响着速度，如Internet等连接设备以及硬盘等的有限速度。就软件范围来讲，没有什么是当前计算机可以完成而老的计算机是无法完成的，但是一个操作系统却已经由几十个MB膨胀到超过1GB。

这并不是要把责任分摊到软件设计人员上，而是因为软件规模和复杂度都已经在逐年增长：在一台10年前的计算机上运行当前的大部分软件是难以想象的，并且大部分是不可能的。

当前软件随着计算机速度和处理能力的提高在不断增长，从这样一个角度，我们可以认为，软件本身也是硬件发展的一个动力。

不管是什么样的因素和动力，计算机制造商都感觉到了巨大的计算机性能持续提高的压力。这也带来了许多收获，如时钟频率的提高、IPC的增长等（见5.5.1节）。遗憾的是，计算机设计人员已经很难再从这些方面来提高计算机的性能了，他们付出了越来越多的努力，获得的性能提升却越来越少。计算机设计人员因此转而借助并行来提高性能。设计一个相对简单的处理器并在一个集成电路（IC）上放置16个会比在一个快16倍的IC上设计一个处理器并利用所有的资源要容易得多。同时并行使用两个已有的处理器也比开发一个速度是现有处理器两倍的处理器容易得多。

理论上，更多的处理器或执行单元并行运行能提高计算的速度，但这也只有在计算能够并行执行时才会成立。给定m个并行任务，每个任务需要T_m秒执行，一个单核处理器执行这些任

⊖ 在这里，速度是以一段简单代码的执行速率来衡量的，即在3.5.2节中介绍的Linux bogomips速率。

务需要 $m \times T_e$ 秒。

[236]

当所拥有的处理单元 n 比等执行的任务多，即 $n > m$ 时，这些任务就可以在 T_e 秒内执行完毕，则所获得的加速比为 $(m \times T_e)/T_e = m$，也称为理论加速比。当然，这个等式并没有考虑消息传递的消耗以及操作系统执行并行处理所需要的支持，同时也假设了这些任务之间没有数据相关。

总体而言，对于一个串行任务所占比例为 f 的程序，完全串行执行需要 T_e 秒。串行部分的执行时间为 $f \times T_e$ 秒，而并行部分使用串行执行的时间为 $(1-f) \times T_e$ 秒。假设没有其他开销，则使用 m 个执行单元并行执行消耗的总时间将为 $(1-f) \times T_e/m + f \times T_e$，而加速比等于原来的执行时间除以并行的执行时间为：

$$\text{加速比} = m/[1 + (m-1) \times f]$$

当 f 为0（意味着没有串行部分）时，结果就与之前的理想加速比相同。这个等式即为 Amdahl 定律，表明了加速比与处理器个数之间的关系，并指出通过并行计算可以获得的潜在性能提高。

5.8.3 其他并行处理

对称多处理（Symmetrical Multi-Processing，SMP）是指在两个或多个相同的处理单元连接到一个共享存储器上。这种技术有很多种变化，包括共享 cache，分布 cache（使用 MESI cache 一致性协议，参见4.4.7节）等。另一种选择是非对称多处理（Asymmetrical Multi-Processing），这个称谓并不常用，但协处理器就属于这一类。在写本书时 SMP 比较常见的例子就是有两个核的 Intel Core 体系结构。图5-25给出了 Core 2 duo 的双核处理器结构，它拥有两个同构的处理单元，中间为共享存储器（L2 cache）。

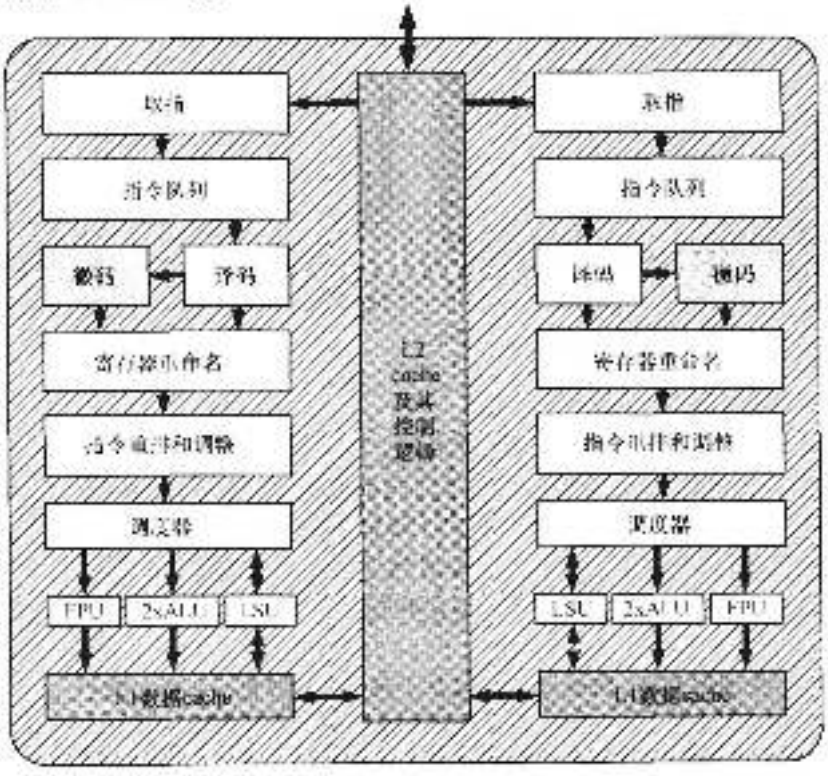


图5-25 Intel 双核体系结构内部模块图。该体系结构为一个对称的双核结构，具有两套对等的处理单元（包含完整的超标量流水线，指令处理硬件等），共享一个与外部总线相连的2级 cache

多核处理器（multi-core machine）是指在一个集成电路（IC）上包含了两个或更多的处理单元（运算是完整的处理器）。一些双核或四核IC也被称为多核，但实际上它们是在一个IC上包含了两块分开的硅片（因而是一个多片模块（multi-chip module，MCM））。随着IC上处理器核越来越多，一些处理器被称为众核处理器（many-core machine）。对于设计人员来说，使用FPGA的数核来探讨多核和众核处理器是比较简单的（参见第8章与此有关的一个例子）。

同构体系结构是指那些处理器里所有的核都是完全相同的。这种处理器在许多方面都比较容易设计和编程。然而，有些时候异构体系结构会更有优势——异构处理器包含了两个或多个不同类型的核，允许将专门用于不同处理的核包含在一起。当前许多智能手机就包含了一个TI
237 的异构OMAP处理器，它由一个ARM核和一个更快的DSP组成。

然而，最著名的异构多核处理器是来自IBM、Sony和Toshiba的Cell处理器。这个处理器装备在了多个超级计算机和（据说是）数百万台Sony的Playstation III上，这是一个将多个较为平凡的处理器能力集合在一个卓越的多核处理器上的典范。

Cell（更准确的称谓是Cell Broadband Engine Architecture，Cell宽带引擎体系结构）的结构如图5-26和图5-27所示。它包含了8个相同的、相对简单的SIMD体系结构处理器，称为SPE（Synergistic Processing Element，协同处理单元），它们由一个IBM Power体系结构的PPE（power processing element，power处理单元）进行调度管理。PPE与现成的IBM PowerPC RISC处理器十分相似。这8个SPE作为基本数据处理单元由PPE控制，而PPE运行着一个操作系统。

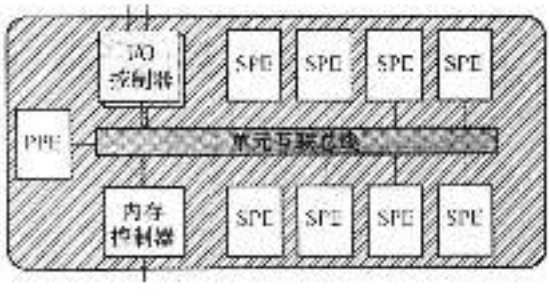


图5-26 Cell宽带引擎体系结构模块图。它有8个SPE、1个单元互联总线（EIB）以及必需的存储器和I/O接口，再加上IBM Power体系结构的power处理单元（PPE）



图5-27 PPE和SPE内部结构，显示了本地存储/cache在设计中的重要性

238 虽然可以作为计算机体系结构一个重要的代表，但由于面积、功耗、热耗散等原因，Cell处理器本身并不太适合大多数嵌入式应用，不过它终将是要影响嵌入式领域的。除去物理和电气方面的因素，针对异构Cell处理器的软件开发工具也在阻碍着它的发展。据报道，许多运行在SPE上的代码需要手工编制，SPE和PPE处理的划分和各个SPE之间处理的划分都需要人工操作。只有等到这些工作可以自动完成或有相应的开发工具支持，Cell处理器才可能会成为受欢迎的产品。

集群计算机，特别是Linux Beowulf，都包含了完整的计算机，每台计算机是独立的而不是共享内存的（通常硬盘也是独立的）。集群将会在9.3节与类似的网格计算和云计算一起讨论。在写本书的时候，世界上多个最快的超级计算机[⊖]（都是集群）都采用了IBM的Cell处理器。

⊖ 世界上最快计算机的最新列表公布在www.top500.org上，每6个月更新一次。

5.9 Tomasulo 算法

在结束本章以前，让我们把目光转向40多年前创新性的IBM System/360。本书通篇都在强调计算机技术的不断发展，历史上也有很多革命性的思想不断出现，而Tomasulo算法就是其中之一，它被应用到了嵌入式系统里（我们将在5.9.3节进行讨论）。

Robert Tomasulo曾面临过一个性能瓶颈，他为IBM System/360所设计的浮点协处理单元在运行程序时会因为指令相关而发生停滞。因此他设计了一种智能的方法，通过有限的乱序执行来打通流水线停滞。这种方法就是著名的Tomasulo算法。

5.9.1 Tomasulo 算法的原理

在我们正式讨论Tomasulo算法如何工作之前，让我们先来看看为什么需要Tomasulo这种类型的算法。这个问题要回溯到在5.2.4节曾讨论过的数据相关性，它表明任何指令使用之前指令的输出作为其输入都需要等待之前的指令完全执行完毕才能开始执行。简单地说，一条指令必须等待它的所有操作数都可用后才能开始执行。

我们已经讨论过使用一种编译时补偿方法（5.2.7节）对指令进行重新排序来解决数据相关问题，重排后的指令消除了与其周围指令的数据相关。另一种解决数据相关的方法就是乱序执行，它不需要让CPU等待存在数据相关的指令执行完毕，而是提前执行那些没有数据相关的指令，这种CPU可以通过执行后边的指令从而保持CPU能够一直正常执行。由此，CPU需要能够取到当前指令往后的指令，而这需要一个强大的分支预测执行部件来支持，否则处理器就不能够取到跨越条件分支指令之后的指令，而且重排的指令也往往被限制在分支指令间的局部代码上。

239 ~ 240

Tomasulo是这样解决这些问题的，允许指令序列中的指令可以带有未知的操作数发送出去，这些未知的操作数称为虚拟操作数（virtual operand），而不需要等到完全求出这些操作数。这些指令将被送往各自目标功能单元的等候站（Reservation Station，RS），直到它们的操作数完全确定后再进入功能单元进行处理。这意味着指令序列就不会被数据冒险所阻塞，当然还是存在一些冒险会阻塞指令发送。

可以通过考察医疗系统的改进来说明这个方法。以前病人去医院看病都需要先在一个大房间里候诊，这个过程有可能持续好几个小时。轮到自己看病后，医生通常又会让病人先去做一些额外的检查如血液检查等。而做完这些检查后病人又要继续在候诊室等待检查结果出来后才能让具体的专科医生给自己看病。如今，这个过程变成所有的病人都到达医院后（指令队列）由护士先把他们按照病种分配到不同的专科诊室（等候站）。然后这些病人在这里提前进行血液或尿液等检查，在检查结果出来并且医生空闲后进入诊室（功能单元）看病。

5.9.2 Tomasulo 系统的例子

我们将对一个采用Tomasulo方法的处理器进行讨论。如图5-28所示是一个采用了Tomasulo控制的动态调度系统，它拥有常规数据总线。四个功能单元都有各自的等候站（RS）。这些RS与不同的总线和寄存器组（一组寄存器用于保存整数，另一个用于保存浮点数）相连，且指令队列（IQ）向它们发送指令。

首先，让我们来看看这个系统是如何工作的。初始时，IQ包含着一系列指令，正常情况下按照顺序向各个功能单元发送这些指令。每条指令中包含一个操作码以及一个或多个操作数。IQ将这些指令按顺序发送到正确RS的空闲槽里。例如，有一条ADD.D指令（双精度加法指令）将由IQ发送至浮点运算单元的RS里。当正确的目标RS满时，IQ就需要停顿一些周期不发送任何指令。

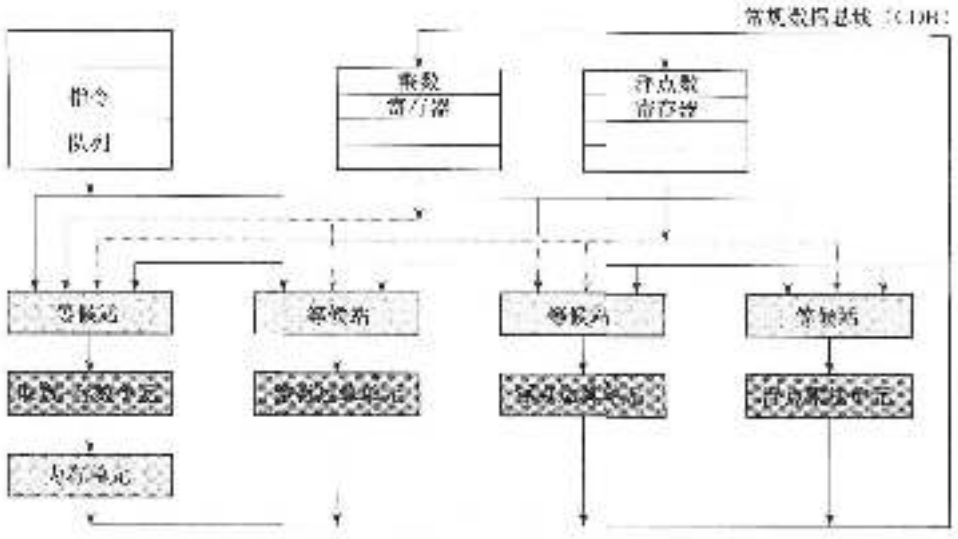


图5-28 在一个运用处理器上实现Tomasulo算法的概要图。左上角的指令队列向四个功能单元专属的保留站通过专用总线传送指令。操作数通过常规数据总线（CDB）从两组寄存器被传送到功能单元。功能单元的输出也通过CDB传送到寄存器组上

显然，每个RS的大小都是设计这种系统的一个重要参数。理想状况是，保持RS有几个空闲的槽，如此IQ就可以随时向这些RS发送指令。

每当IQ发送一条指令（例如一个操作数加一个操作码），它都要检查操作数之间的相关性，
241 即检查这条所发送的指令其操作数是否来自之前尚未执行的指令，或句话说，检查有没有尚未解决的数据相关。如果存在相关性，那么指令就会被发送至RS，同时赋予一个“虚拟”的操作数来替代还未得到的操作数。如果不存在相关性，那么发送指令时就赋予它一个真正的（已知的）操作数。

所有的RS都是相互独立的。只要指令的操作数全部已知并且功能单元空闲，RS在每个周期都可以向功能单元发送指令。

一般来说，每个功能单元处理指令的时间长短各不相同。因此RS空闲率是不同的。如果一个RS内有超过一条指令的所有操作数都是已知的，那么最先进入RS的指令将会被先发送。常规数据总线（Common Data Bus，CDB）用于将结果写回寄存器（这是一台读数-存数机器），但在每个时钟周期CDB只能写回一个结果。所以如果在一个时钟周期里有两条指令同时完成，那么别旧是最先发送的指令首先被放至CDB上。

每个RS都会不停地“监听”CDB，任何存放了符合所须操作数的指令的RS都要查看所写
242 回的寄存器是否为那些指令所等待的操作数。一旦最指令所等待的操作数，RS就会从CDB上取出这个操作数给那些指令。这意味着CDB不仅要携带结果值本身及目标寄存器，还要携带能够通知RS本结果最终为那些指令所等待的操作数的信息（因为一个等待写回R3寄存器的结果书指令有可能会“看到”CDB多次将结果写回R3——而仅有在源代码中就在该指令之前目标寄存器为R3的指令的结果才是该指令所要书操作数）。

IQ为每条指令的每一个操作数都提供一个非的标签，该标签伴随着操作数通过RS、功能单元，最后与该指令的结果一起出现在CDB上。我们已经知道，那些依赖以前指令结果的指令在发送时会带有虚拟操作数。但是这些虚拟操作数包含两个信息——寄存器名和标签值，存在依赖的指令“监听”CDB实际上是在“监听”那些写回正被寄存器且带有正确标签的结果。

让我们通过一个例子说明这个过程。假定存在一个如图5-28所示的Tomasulo机器，其时间

参数如下：

取数－存数单元：	5个周期完成
浮点加法器：	2个周期完成
浮点乘法器：	2个周期完成
整数单元：	1个周期完成
等待站长度：	1条指令

每个周期发送指令数：1

寄存器数：32个通用寄存器（gpr）+32个浮点寄存器（fp）

在这台机器上执行以下嵌入式指令：

```
i1    LOAD.D fp2,(gpr7, 20)
i2    LOAD.D fp3, (gpr8, 23)
i3    MUL.D fp4, fp3, fp2
i4    ADD.D fp5, fp4, fp3          ;fp5=fp4+fp3
i5    SAVE.D fp4, (gpr9, 23)       ;将fp4保存在地址(gpr9+23)
i6    ADD gpr5, gpr2, gpr2
i7    SUB gpr6, gpr1, gpr3
```

该程序执行的预约表如表5-3所示，它展示了当虚拟操作数确定时指令从指令队列通过等候站进入功能单元的操作。最后结果通过CDB写回。

表5-3 这个预约表给出了一台Tomasulo机器执行一段存放于IQ中的程序，该机器包含几个等候站（RS），分别为一个取数－存数单元（LSU）、一个ALU、一个浮点ALU（FALU）和一个浮点乘法单元（FMUL）发送指令。完成执行的指令的结果通过常规数据总线（CDB）被写回寄存器组。在RS里等待虚拟操作数确定的指令和在功能单元内多周期执行的指令使用灰色表示

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
IQ	i1	i2	i3	i4	i5	i5	i5	i6	i7												
RS:lsu		i1	i2	i2	i2	i2	i2	i5	i5	i5	i5	i5	i5	i5	i5	i5					
LSU			i1	i1	i1	i1	i1	i2	i2	i2	i2	i2					i5	i5	i5	i5	i5
RS:alu									i6	i7											
ALU										i6	i7										
RS:falu					i4	i4	i4	i4	i4	i4	i4	i4	i4	i4	i4	i4					
FALU																	i4	i4			
RS:fmul				i3	i3	i3	i3	i3	i3	i3	i3	i3	i3								
FMUL														i3	i3						
CDB								i1			i6	i7	i2			i3			i4		

注意到指令序列i1至i7通过乱序执行的顺序为：i1、i6、i7、i2、i3、i4和i5，但最后并没有都出现在CDB上。有趣的是，如果我们通过手动对这些指令进行重排，使得在一个简单的流水线处理器上执行的时间最小化，那么有可能获得相同的执行顺序。指令i6和i7与其他指令不存在数据相关，可以提前执行以便把存在数据相关的指令分开。 243

最后一点需要指出的是，在这个执行中最主要的延迟来自存取单元（load-store unit，LSU）。当然，给出的参数中指出每次存取需要5个周期（这在现代处理器中并不夸张，虽然使用片上cache可以提高速度），在给定这种参数的条件下，LSU就成为瓶颈了。

一种解决这种瓶颈的方法可以是额外增加一个LSU（既可以拥有自己的RS，也可以与当前的LSU共用RS）。当然，不管有多少个LSU，对取数－存数操作进行重排才是在Tomasulo机器上解决这种瓶颈的主要办法。然而，读者也需要意识到，在访存时也会存在数据相关，而Tomasulo算法并不能解决这种问题。我们通过考虑这样一小段代码来了解这个问题；虽然以下所提到

的3个地址看起来并不一样，但它们在实际中也可能不存在。

```
i1    read from (gpr7, 20)
i2    read from (gpr8, 23)
i5    write to (gpr9, 23)
```

指令i1从地址（gpr7 + 20）读数据。如果将gpr7的值设为1003，那么这个地址就是1023。类似地，如果gpr8的值为1000，那么i2也将从地址1023读数据，这就引起了一个读后读（read-after-read）冒险；虽然这并不是一个令人困扰的问题，但是如果能提前发现还是可以优化掉的。

而更多的关注可能会集中在此：如果gpr8与gpr9相等，那么i2和i5就形成一个WAR冒险（在5.2.4节有描述）。在当前的代码段中只有一个LSU，不能将i5重排放置在i2之前，因此在这种情况下就不会造成什么问题。然而，这也要归功于i5与其他指令存在着寄存器依赖。

244

让我们通过改变代码来说明这个问题。在此，i5变成了i5，SAVE.D fp1，（gpr9，23），这样，这条指令就与其余的指令没有寄存器依赖。我们将加入第二个LSU及其RS，如图5-29所示，并运行以下代码：

```
i1    LOAD.D fp2, (gpr7, 20)
i2    LOAD.D fp3, (gpr8, 23)
i3    MUL.D fp4, fp3, fp2
i4    ADD.D fp5, fp4, fp1          ;fp5=fp4+fp3
i5    SAVE.D fp1, (gpr9, 23)       ;将fp1保存在地址(gpr9+23)
i6    ADD gpr5, gpr2, gpr2
i7    SUB gpr6, gpr1, gpr3
```

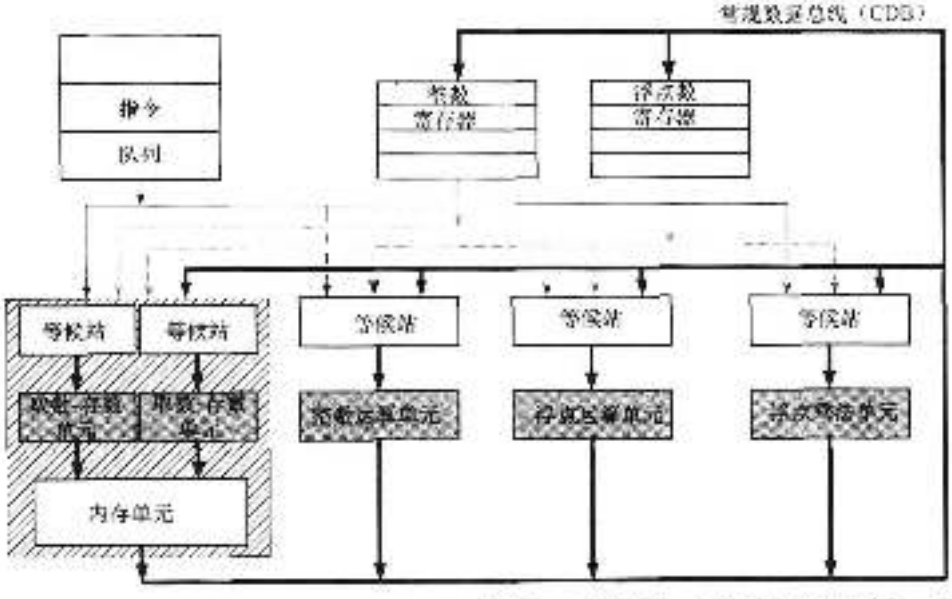


图5-29 对图5-28的基本Tomasulo机器的一个改进，将单取数－存数单元改为双取数－存数单元，同时对应的等候站也改为两个

表5-4给出了更改后的机器运行这段新代码的预约表。值得注意的是，所加入的第二个LSU显著提高了程序执行的速度。现在整个程序完成需要15个周期而不是原来的21个周期了，并且更加紧凑。

表5-4 这是一个与表5.3类似的Tomasulo机器的预约表，但它拥有两个LSU及相应的等候站。所执行的程序被略微修改。在CDB、RS或是功能单元里等待的“空间”的指令，被标上“*”号，如“i6*”在第9个周期在ALU的输出进行等待，因为指令i2的结果当前正占用着CDB

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IQ	i1	i2	i3	i4	s5	i6	i7									
RS:fmul		i1				s5	s5*									
LSU1			i1	i1	i1	i1	i1	s5	s5	s5	s5	s5	s5			
RS:fmul			i2													
LSU2				i2	i2	i2	i2	i2								
RS:alu							i6	i7	i7*							
ALU								i6	i6*	i7						
RS:falu					i4	i4*	i4*	i4*	i4*	i4*	i4*	i4*				
FALU													i4	i4		
RS:fmul				i3	i3	i3	i3	i3	i3							
FMUL										i3	i3					
CDB								i1	i2	i6	i7	i3	i2	s5	i4	

加速是好事情，但是让我们先来更细致地考察访存。请注意s5在第8个时钟周期进入第一个LSU并对内存地址（gpr9 + 23）进行写操作。同时，i2仍从该址（gpr8 + 23）读数据。显然，当gpr8 = gpr9时，读和写操作的目的地址是相同的，即i2将从这个地址读数据，而此时s5要往这个地址写数据，那么指令i2所要读取的数据就会被污染从而造成错误。 245

此类这种问题是由于没有机制来控制访存的冒险。除了跟踪访存或通到未确定的内存源与按顺序执行之外，没有更简单的方法来解决这种问题。

5.9.3 嵌入式系统中的Tomasulo算法

正如之前所说的，Tomasulo算法是为大型机如IBM System/360，特别是91模型而设计的。我们在第1章的图1-5中给出了这个巨人机器的照片。那么为什么在一本计算机体系结构的书中介绍这种方法时又要强调嵌入式系统？

首先，乱序执行是不能简单实现的，而对于那些用于嵌入式系统的CPU设计，乱序执行并不是设计人员太愿意考虑的。然而Tomasulo算法在提高性能时仅需添加硬件（等候站里的额外寄存器）。它也不需要依赖其他一些先进的技术如分支预测、动标量流水线等，只需要相对简单的处理器设计就能实现乱序执行。

其次，Tomasulo算法可以将指令执行分布在整个系统中。在指令发送单元中并没有实际的瓶颈，它也并不真正受制于时钟速度（实际上，Tomasulo算法很容易将处理器扩展为多功能单元，只需对结构进行很小的改变），系统分布的本质很适合FPGA。Tomasulo算法中最主要的瓶颈来自CDB，它必须延伸至每一个等候站和每一个寄存器组中的每一个寄存器。然而，这种类型的全局总线在FPGA中早已经布好，对其进行扩展比设计“较短的”并行总线还要方便。 246

最后，我们在5.9.2节的例子中也提到了额外的功能单元是如何提高性能的（在这个例子中是第二个LSU），虽然我们也提到这种额外的LSU还导致了特定的内存冒险相关问题。在嵌入式系统中，更倾向于在编译时为定变量和矩阵的地址，并且不需要为其指定相关的基址寄存器，如此就解决了额外LSU带来的问题。更重要的是，在嵌入式领域一般都可以预知在系统上将会运行什么软件，如此就可以预先为这种软件选择合适的功能单元（实际中很多例子都是如此）。

5.10 小结

前面几章大多关注的是计算的基础、计算机（CPU）内的功能单元以及这些设备的操作，而本章开始讨论这些设备的性能——主要原因是性能已经成为当前计算机发展的主要动力。

我们看到了几种类型的加速手段，从传统的提高时钟频率，到现在的已经很成熟的流水线、CISC与RISC、超标量及其他硬件加速器（如零开销循环和专寻地址硬件）。

本章的重点是流水线，即由冒险和分支引起的性能下降问题，以及如何使用延迟分支和（或）分支预测来解决这种问题。

到此我们已经对CPU的内部结构都有了了解（除了将在第9章讨论的一些更隐秘的方法
247 外），接下来，我们将注意力转向如何与CPU进行通信，即向系统输入或从系统输出信息。

思考题

5.1 在一些流水线处理器上，条件分支有可能会引起流水线阻塞或浪费周期。以下的代码段会阻塞一个三级流水线，为什么？

```
MOV R0,R3          ;R0 = R3
ORR R4,R3,R5       ;R4 = R3 OR R5
AND R7,R6,R5       ;R7 = R6 AND R5
ADDS R0, R1, R2    ;R0 = R1 +R2, 并设置条件标志位
BGT loop           ;当结果大于0时分支
```

注意：指令后带有“S”的指令意味着指令的执行结果将会对条件产生影响，而不带“S”的指令其结果不会对条件代码产生影响。同时假设每条指令都在一个周期内完成。

5.2 对问题5.1的代码进行重排以避免阻塞的发生。

5.3 如果ARM支持延迟分支，那么可以用BGTD来代替上面代码里的BGT。使用BGTD重写问题5.1里的代码。（提示：只需改动一条指令。）

5.4 在一个8位RISC处理器上，执行下列ARM指令，初始条件为R0 = 0x0，R1 = 0x1，R2 = 0xff，确定每条指令完成后的4个标志位的状态。

指令	N	Z	C	V
MOVS R3, #0X7f				
ADDS R4, R3, R1				
ANDS R5, R2, R0				
MOVS R5, R4, R4				
SUBS R5, R4, R1				
ORR R5, R4, R2				

5.5 指出以下包含延迟条件分支的ARM汇编指令中的4种冒险：

```
i1    ADD R1, R2, R3
i2    NOTS R1, R2
i3    BEQD loop
i4    SUBS R4, R3, R2
i5    AND R5,R4,R1
248   i6    NOT R1,R2
```

5.6 分支指令往往因为数据相关、指令排序以及硬件能力而引起流水线阻塞。延迟分支可以避免这种阻

塞。请说出另外两种用于提高分支性能的方法。

5.7 说出两种减少或移除数据冒险带来的影响的方法。

5.8 画出一个能够实现任何数与2或10相乘的硬件框图。使用数据转发实现反馈路径。要求如下：

- 最多两个单位移位器。
- 最多两个全加器。

忽略所有的控制逻辑和存储寄存器。

5.9 将上题实现的乘法器流水线化，使用一个单位加法器和一个单位移位器，同样忽略所有的控制逻辑和存储寄存器。

5.10 画出以上三级流水乘法器的预约表。

5.11 指出在CPU及其协处理器之间传送数据的主要机制，并指出这与同构双核处理器有什么不同。

5.12 列出5条RISC处理器区别于CISC处理器的典型特性。

5.13 在一个纯RISC处理器上每周期执行指令数（IPC）的范围是多少？它与一个理想的能够同时发送3条指令的超标量处理器有什么区别？

5.14 一个数字信号处理器（DSP）实现了简单的零开销循环硬件，这个硬件带有一个循环计数器、一个起始地址寄存器和一个终点地址寄存器。该硬件会检测程序计数器（PC）在什么时候与终点地址寄存器相等，如果循环计数器非零则会重设PC为起始地址寄存器的值。指出以下几种C代码中哪种适合以上硬件执行。

a.
```
for (loop = 0; loop <99; loop++){
        < 有大量计算 >
}
```

249

b.
```
loop = 99;
do{
        < 有大量计算 >
} while (loop-- >0)
```

c.
```
while(x + y != 23)     {
        < 有大量计算 >
}
```

5.15 假设一个程序中包含1224个任务（其中200个需要串行执行，而1024个能够并行执行），在一个理想的16路同构并行机器上执行该程序，每个任务需要2ms的CPU时间来执行。请计算并行执行的加速比。

5.16 参考框5.1的流水线加速比和效率的计算。如果某CPU流水线设计具有68%的效率和3.4的加速比，请确定该流水线的级数。

5.17 为了对一个数字音频进行延时，处理器需要持续读取音频采样，经过延时后再在某个时刻输出它们。对于一个采样率为8kHz的16位音频，延时1008ms需要多少个采样？在一个采用循环缓冲的ADSP2181上实现这个延时，使用以下指令编写伪代码。假设开始时缓冲中为空。

```
<reg>=IO(audioport)     将数据读入寄存器
IO(audioport)=<reg>     读出寄存器里的数据
<reg>=DM(I0,M0)         从内存读出数据，地址由指针I0指出，完成后指针增加M0
DM(I0,M1)=<reg>         将寄存器里的数据存入内存，地址由指针I0指出，完成后指针增加M1
I0=buffer_start         将指针I0设置为内存中缓冲的起始位置
L0=buffer_end           设置循环缓冲的上限（当I0=L0时，重置I0）
M0=x                    设置地址改变器M0的值为x
M1=x                    设置地址改变器M1的值为x
B loop                  分支至标签为loop的程序段上
```

<reg>可以为AX0、AX1、AY0和AY1里的任意一个。

250

5.18 指出执行以下ARM条件指令前哪些条件标志位需要设置：

指令	含义	N	Z	C	V
BEQ loop	如果等于0则分支				
ADDLT R4, R9, R1	如果小于0则执行加法				
ANDGE R1, R8, R0	如果大于等于0则执行与				
BNE temp	如果不等于0则分支				

5.19 简单阐述在什么情况下需要使用影子寄存器。如果处理器不支持影子寄存器，那么程序员需要采取什么方法？

5.20 在一个拥有12位全局分支预测器的处理器上跟踪以下指令的执行，预测器初始时为“DT”：

```
i1          MOV R8, #6          ;将立即数6加载入寄存器R8
i2          MOV R5, #2          ;将立即数2加载入寄存器R5
i3    lp1:  SUBS R8, R8, R5     ;R8 = R8 - R5
i4          BLE exit            ;如果结果小于等于0则分支
i5          BGT lp1             ;如果结果大于0则分支
```

251

外部总线

前面五章描述了微处理器发展过程中那些相对稳定、没有什么革命性变化的方面，包括：设计具有高速处理能力器件的驱动力、RISC 的概念，以及体系结构和指令集对于高性能程序设计的支持等。

从本章开始我们结束基本 CPU 知识的学习，来了解一下核心逻辑与外部世界的相互作用。外部世界是指接口、总线以及一些嵌入式系统中与此相关的特有概念——实时处理和实时互操作等。

6.1 总线接口

在当今市场上完全可以买到一个片上计算机，它是一块集成了 CPU 及计算机外部逻辑的集成电路芯片。

这块芯片直接提供了一个计算机系统所要求的所有外部总线。而在没有集成之前，在 CPU 和外设之间只有内部总线——现在这部分处于芯片内部而不是芯片外部。这种器件称为片上系统或 SoC 处理器。

以如图 6-1 所示的 20 世纪 90 年代以来标准个人计算机体系结构为例，它包括一个 CPU 以及围绕着它的各个器件。以往这个体系结构在一个母板上实现，包括约 20 个芯片，而近几年，它可以在一个片上系统器件上实现。系统同样采用标准接口，只是全部在一块集成电路中实现，见图 6-1 中的阴影区域。

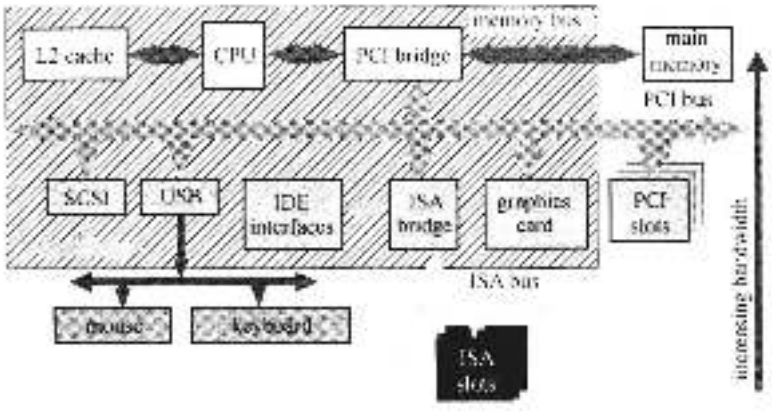


图 6-1 20 世纪 90 年代后期标准个人计算机体系结构框图

在基于 ARM 的系统中一般采用两种标准总线——AHB（ARM Host Bus）和 AMBA（Advanced Microcontroller Bus Architecture）。在许多厂家出品的基于 ARM 的集成电路中可以看到这两种总线。同时，在诸如 ARM 集成平台这类母板上也可以看到以离散器件实现这两种总线的方案。ARM 总线已成为一种业界标准，用于一些非 ARM 处理器接口，例如基于 SPARC 的 ERC32 处理器。这些外部或内部总线标准使外设制造商（即独立集成电路厂商或内部逻辑部件厂商）能够生产出在系统中协同工作的标准单元。 [252]

虽然读者很可能已经了解并行总线的概念，在此我们还是要简单回顾一下。并行总线最重要的作用是它能够支持多个器件共享同一物理资源——数据总线——以传送输入或输出信息。通常 CPU 作为主器件，负责以控制信号实现对并行总线的控制。当两个 CPU 共享总线时就必须有仲裁，这个角色或者由其中的一个 CPU 担当，或者采用一个独立的外部总线仲裁器。

主器件通过总线控制信号告诉其他器件何时从总线读、何时向总线写。总线兼容的器件必须确保不向总线写数据时一定不去驱动总线，即其输出总线处于高阻状态。

6.1.1 总线控制信号

典型的总线控制信号如下所示，其中小写字母“n”表示该信号低电平有效。

- nOE 和 nRD——输出使能/读使能，表示主控制器允许某个器件向总线写数据。存储地址或片选信号决定了被选中的器件。
- nWE 和 nWR——写使能，表示主控制器本身将数据输出到总线上，而另外的一个或多个器件读取该数据。读取数据的具体器件通过 nOE/nRD 指定。
- RD/nWR——读/不写。当该控制线为高时，若任何有效地址或片选信号出现则完成一个读操作；而当该控制线为低时，若任何有效地址或片选信号出现则完成一个写操作。
- nCS 和 nCE——片选/片使能信号，每个器件有一个该信号，当信号有效时表示相应器件与总线“通话”。以前，由分立的地址译码芯片产生片选/片使能信号，而多数现代嵌入式处理器本身就会产生该信号。

253

在使用 DIP 封装技术的年代，设计者必须最小化集成电路管脚的数量，因此导致了一些奇特的复用、混合并行总线设计，其中包含了一些非常规的总线控制信号。而上面给出的信号是当代嵌入式处理器和外设中最常见的形式。

其他信号还包括 nWAIT 线等，慢速外设使用该信号通知正在存取它们的 CPU 在该慢速设备准备好之前不要利用总线做其他事情。与此相配合的信号还包括总线准备好、总线请求、总线许可等，而后两条控制线用来实现直接存储器存取（DMA）。

6.1.2 直接存储器存取 （DMA）

直接存储器存取允许共享总线的两个器件彼此通信而无需打扰控制 CPU。如果没有 DMA，CPU 要用一条或几条指令先从源外设上读取一个数据，然后再将数据写到目的外设上。对取数 - 存数体系结构机器而言，以上操作还会导致数据传输过程中占用内部寄存器。

DMA 只需要少量的 CPU 介入以启动传输过程，之后的传输操作基本上独立于 CPU 进行。源外设通过外部总线将数据送到目的外设上。除了初始启动过程，在每个字的传输过程中不再需要 CPU 介入，也不需要占用 CPU 内部寄存器。在数据传输的同时，CPU 可以完成任何其他需要的操作。

对于有很多设备共享总线的系统而言，会有一些 DMA 通道，每一个通道被赋予不同的端点和优先级，如果有两个以上 DMA 通道同时要求操作，那么允许优先级高的通道先使用总线。框 6.1给出了基于 ARM 的处理器中 DMA 系统的工作过程。

框 6.1 一种商用处理器中的 DMA

让我们来看一个实例——基于 ARM9 的 S3C2410，它是三星公司生产的一款片上系统处理器。它有 4 个 DMA 通道，一个控制器位于内部总线和外部总线之间，实现内外总线数据传输的各种组合。

每个通道有 5 种可能的触发源，以一个包括三个状态的有限状态机控制。假设我们已经选定了重复性操作，正确设定了源和目的地址，则具体操作如下所示：

状态 1：DMA 控制器等待 DMA 请求，此时，DMA ACK 和 INT REQ 均为不活动状态（0）。如果得到

DMA 请求，进入到状态 2。

状态2：DMA ACK 置 1，读取数据并写入目的地（可由读操作的等待数来置），然后进入到状态 3。

状态3：从源地址读数据，然后写到目的地址上。重复 n 轮，计数器递减，直至为 0，说明，有效地处理器仅在等待操作完成，然后返回状态 1。

虽然 DMA 在许多设计中改进了处理器效率，但对于性能关键的系统仍存在进一步改进的空间。事实上，在一些 CPU 中 DMA 控制器本身就是一个简单的足够智能的 CPU。以基于 ARM 的 Intel IXP425 网络处理器为例，它包括一些集成外设，例如 USB、高速串口和两个以太网 MAC（介质存取控制器：一种以太网接口器件）。主处理器工作频率为 533MHz，其他二个从处理器工作频率为 100MHz，专门处理系统总线上的输入/输出。这些 RISC 处理器将主 ARM CPU 从冗长、烦琐的总线操作和存储器存取操作中解放出来，而每个从处理器专门用来运行 MAC 协议，使得 IXP425 非常适合完成网络操作。 254

6.2 并行总线规范

基于 ARM9 的三星 S3C2410 片上系统器件的总线工作时序如图 6-2 所示。之所以以此为例是因为三星清楚地定义了时序与参数，以及时序与为数不多的控制寄存器之间的对应关系。对大多数 CPU 而言，情况要复杂得多——需要人工完成周期数计算，包含组合和分离参数，以及背后的非常规行为等。

HCLK 时钟是主要时钟之一，用于驱动存储器接口和片上其他器件。其工作频率一般为 100MHz，且在片外无效——在此仅作为参考时钟。25 位外部地址总线和 nGCS 片选信号定义了到外部器件的接口，例如 ROM 或类似器件（包括使用该接口的大部分外部总线桥接器件）。当与任何外部器件互操作时这些信号有效。图 6-2 底部的阴影部分表示读写信号及其相应行为。

时序图中一些总线为高阻状态，即线的位置非高非低处于中间，表示该线没有被驱动。 255

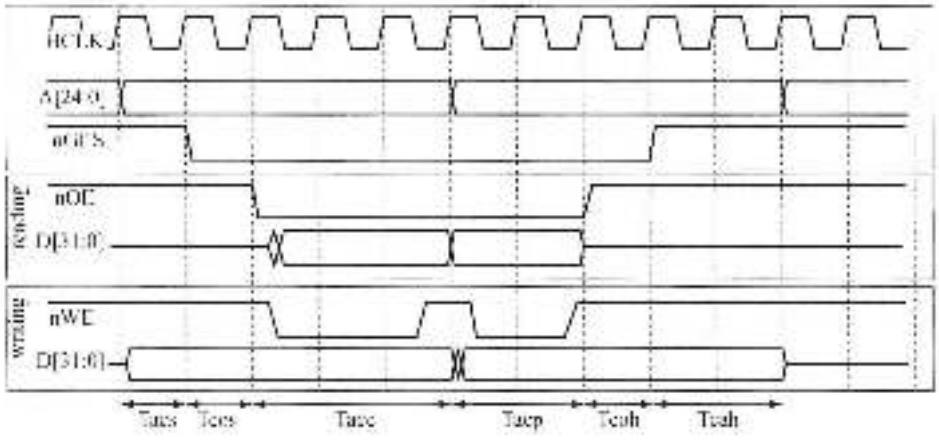


图 6-2 三星基于 ARM9 的片上系统 S3C2410 的 SRAM 总线工作时序图。上半部分是时钟和总的控制信号；中部是读操作相关信号（这时写 nWE 始终保持高电平），底部是写操作相关信号（这期间 nOE 始终保持无效，即高电平）。注意读写操作不会同时发生，在任何时候最多只有一个操作发生

该时序适用于读和写操作，通过 S3C2410 寄存器可以设置通过接口所连接的外部器件的访问方式。由 nGCS 片选的每个外部器件共享数据、地址、读/写线等，但其时序为独立设置。因此，快速和慢速器件可以共存于同一物理总线上，只是片选信号各自独立。

下表给出了图 6-2 中所示时序信号的含义以及与该时序图对应的设置。

信号参量	含义	设置
Tacs	先于 nGCS 有效的地址设置时间（0、1、2 或 4 周期）	1 周期
Tcos	先于 nOE 的片选设置时间（0、1、2 或 4 周期）	1 周期
Tacc	存取周期（1、2、3、4、6、8、10 或 14 周期）	3 周期
Tacp	页模式下存取周期（2、3、4 或 6 周期）	2 周期
Tcoh	在 nOE 无效后片选保持时间（0、1、2 或 4 周期）	1 周期
Tcah	在 nGCS 无效后地址保持时间（0、1、2 或 4 周期）	2 周期

页模式是指一些针对某一器件的重复性操作以快速突发方式完成，不需要存取其他器件。框 6.2 讨论了一些采用上述总线连接外部器件的例子。注意，一些器件直接利用 nWAIT 信号准确地告诉 CPU Tacc 的长度（即外部器件需要 CPU 等待的时间），而有些器件则跟上述器件的连
256 接方式大不相同，例如 SRAM 类器件。

框 6.2 外设连接总线设置

下面是一些外部器件连接示例。假设总线时钟为 100MHz（即每周期 10ns），我们来看一下如何利用上述信号实现连接。

问：一个慢存储器件需要 120ns 查找一个内部地址。

答：这意味着读写周期必须大于 120ns。相关设置参数是存取周期 Tacc 应设置为 14 周期，即大于 120ns 的最小周期数。

问：一个外设的片选信号必须在读信号之前至少 25ns 有效。

答：在这种情况下，nGCS 必须先于 nOE 设置为低电平，相关参数为 Tacs 应设置为 4 周期，即大于 25ns 的最小设置。

问：一个外设在被读取数据之后，保持 12ns 对总线驱动状态。

答：在这种情况下，我们需要确保在读该外设之后的 12ns 时间内总线不被其他设备占用。相关参数为 Tcah 或 Tcoh 或两者（多数情况下是 Tcoh）。为安全起见我们将二者均设置为 1 周期，总共为 20ns。该时间称为保持期（hold-off period）。

通常，任何外设手册上都有时序图，由此可以推导出需要的信息。但是，当存在不确定之处时，可以先选择最长或最慢的数值作为起点，然后在保证系统可靠工作的前提下逐步减小它们。另外，为保险起见，应选择稍慢于最快设置的数据——因为可以在实验室环境下工作的最快设置未必能够在寒冷或炎热的实际环境下工作，未必能够在使用若干年后仍正常工作。

6.3 标准接口

现代计算机，无论是嵌入式系统、台式机还是服务器，都倾向于使用有限的几种标准接口。本书只是简单介绍一下比较常见的几种接口及其特性。

这些接口根据数据传输、系统控制、存储器连接等功能进行分类。值得注意的是很多接口被创造性地应用，超出了原设计者的设计考虑。

6.3.1 系统控制接口

257 系统控制接口用于控制和设置各种低速器件。它们的典型特征是管脚空间有效、相对低速、结构简单。下面是一些系统控制接口的例子。

- SPI(Serial Peripheral Interconnect)，串行多点编址，20MHz。
- IIC(Inter-IC Communications)，串行多点编址，1MHz。
- CAN(Controller (or Car) Area Network)，串行多点编址，若干 MHz。

还存在一些变种，例如 Atmel 公司的 TWI（Two Wire Interface），Dallas 半导体公司的 1-wire

接口等。

6.3.2 系统数据总线

多年来，业内一直致力于引入标准总线和并行总线体系结构。下表给出了在个人计算机体系结构中常见的并行总线。值得一提的是嵌入式系统往往倾向于使用与此不同的总线体系结构。其中有两个典型例子，一是 AMBA，源于 ARM 公司和 GEC Plessey 半导体公司（后成为 Marconi 公司的一部分，最后被 Mitel 半导体公司收购），参见 6.2 节 S3C2410 例子中的相关描述；二是 APB（ARM Peripheral Bus）。在大量的片上系统和嵌入式处理器设计中均可以看到 AMBA 和 APB。这些总线的命运远强于 IBM 于 20 世纪 80 年代后期引入的 MCA（MicroChannel Architecture），尽管 MCA 被认为是一个良定义的总线系统，但最终被 EISA 淘汰。

总线名称	宽度（位）	速度（MHz）	数据速率（MiB/s）
8 位 ISA(Industry Standard Architecture)	8	8	4
16 位 ISA	16	8	8
EISA(扩展 ISA)	32	8.33	33.3
32 位 PCI(Peripheral Component Interconnect)	32	33	132
64 位 PCI	64	33	264
1 x AGP(Advanced Graphic Port)	64	66	266
8 x AGP	64	533	2100
VL-BUS	33	50	132
SCSI-I & II	8	5	40
Fast SCSI-II	8	10	80
Wide SCSI-II	16	10	60
Ultra SCSI-III	16	20	320
PCIexpress——每通道（最多32通道）（采用 LVDS①）	1	2500	>500
RAMBUS（184 管脚 DRAM 接口）	32	1066	4200
IDE②/ATA③	16④	66④	133⑤
SATA（串行 ATA），采用 LVDS	1	1500	150
SATA-600	1	未知	600

①LVDS：Low-Voltage Differential Signaling。

②IDE(Integrated Drive Electronics)：对应于第一个 ATA 实现。

③ATA(Advanced Technology Attachment)：现在重命名为并行 ATA 或 PATA，以区别于 SATA。

④假设 ATA-7 操作。

⑤在 45 厘米最大线长上运行时为 133MHz。

258

虽然有大量的总线系统（这里列出的只是比较常见的），但它们存在着很多的共性，大多数采用了相同的基础通信方式和仲裁策略。有为数不多的几种电压和时序标准供选择。

有时一些电器特性相同的总线在基于它建立的实际通信协议中被赋予了不同的名字和使用方式。OSI 层次参考模型（见附录 B）将底层电器参数、硬件和时序参数等定义为物理层的一部分，而将通信协议定义在数据链路层。例如物理层接口 LVDS 已被越来越多地用做嵌入式计算机系统中的高速串行总线。

下面，进一步探讨在 SATA 和其他方案中作为物理层的 LVDS 之前，我们先来了解一下两种常见的传统总线。

6.3.2.1 ISA 总线及其衍生总线

工业标准体系结构（ISA）总线由 IBM 在 20 世纪 80 年代早期推出，用于个人计算机（特别

是IBM个人计算机）总线系统。很快，该8位总线于1988年在ISA的扩展版本EISA中扩展为16位及32位总线。每个新的扩展版本都与前面的版本兼容。

正如前面所提到的，IBM在此之后试图转到微通道体系结构（MCA），但是由于IBM没有完全授权该总线成为一个完整的总线标准，很自然地其他计算机厂家依然继续使用EISA总线。之后，IBM放弃了MCA，但ISA后来的衍生总线，即外部设备单元互联（PCI）总线和VESA本地总线，确实融合了IBM MCA总线的一些特性。

作为总线，从使用年限的角度看ISA和EISA是很成功的，但它们始终面临着严重的可用性问题（见框6.3）。这些问题伴随着无休止的提速压力，最终导致了用于台式机的PCI总线的定义和采用。

ISA不仅派生出PCI和VESA总线，还有ATA标准，ATA标准又衍生出IDE标准、增强型IDE（EIDE）、PATA和SATA等。事实上，ISA还衍生出了PC卡标准接口。[⊖]尽管ISA作为标准已经经历了30年，但它仍然作为传统总线在当今计算机系统中使用。

框6.3 ISA面临的问题

ISA作为它那个时代的产品，其设计是很合理的：用于Intel 8086的8位总线，时钟频率为4.77MHz，工作在5V电压下。然而，从先进的CPU角度看它存在着一些与生俱来的硬件局限性和可用性问题。

硬件局限性

Intel 8086和8088采用40管脚双列直插式封装（DIP），分别包括16位和8位外部数据总线。由于缺少管脚，外部总线被复用，即一些物理管脚完成两个功能。即使如此，最多也只能有20根地址线，即只能支持存取1MB(2^{20})地址空间。更强的约束来源于8086内部的16位地址寄存器，它意味着只能存取64KB(2^{16})地址空间。Intel也提供了两种类型的外部总线存取：存储器存取（使用20位地址总线）和I/O存取（使用20位总线中的16位总线）。有趣的是现代许多系统中仍然保留着这种存储器和地址空间划分——这与追求简洁的ARM之类的处理器不同，它们只将存储器空间映射到外部地址空间。

虽然8088的管脚通过缓存和解复用后连接到ISA总线上，但总线仍然受到20位地址约束，且采用分立的I/O存取（为此提供了单独的控制管脚集合）。这样做的优点是ISA总线可以很好地支持4通道DMA（见6.1.2节）。

可用性问题

这一点并不单单针对嵌入式计算机系统，但有助于理解ISA被PCI替代的原因。许多个人计算机用户在安装ISA卡（或EISA卡）时碰到了问题。用户不仅需要将总线卡物理地插入系统，还想要安装软件提供多数普通用户不知道的信息，例如总线卡所连接的I/O端口、DMA通道IRQ（中断请求）线等。当然，相比更早的需要跳线安装卡的设备而言，这种提供配置信息的做法已经是进步了。

一些安装软件可以扫描ISA总线以寻找已安装的卡。有时可以正常工作，有时却会使系统崩溃，其原因是用户输入了错误的信息。一些个人计算机允许在BIOS控制下或在自动引导过程中交换ISA插槽，这意味着总线卡今天能正常工作，而明天未必能。

为此，制造商开始定义称为“即插即用”（plug and play）的标准，简写成PnP。理论上讲，这意味着插入卡后即可工作。但是，这个标准很快就被称为“即插即祷”（plug and pray），由此导致了这种总线在战略上让位。很幸运，由PCI替代ISA/EISA。

259
~
260

6.3.2.2 PC/104

在嵌入式系统中出现最多的ISA总线版本是由PC/104组织[⊜]推出的PC/104总线标准。

⊖ PC卡以前叫做PCMCIA（Personal Computer Memory Card International Association），不过它还有一个名字“People Can't Memorize Computer Industry Acronyms”（见http://www.sucs.swan.ac.uk/cmckenna/humour/computer/acronyms.html）。

⊜ http://www.pc104.org。

PC/104 标准对印制电路板小型化提出了强制性要求，即 96mm × 60mm，这对于许多嵌入式系统而言是很理想的。该电路板的一个边上有一个 8 位 ISA 总线连接器。这个 2.5mm 宽的连接器上分两排排列了 64 个管脚。在顶部该连接器表现为一个插槽，而在底部该连接器表现为管脚。这种设计使得电路板可以叠放在一起。通常，一个 40 管脚的连接器 J2/P2 与另一个连接器 J1/P1 叠加在一起可提供一个 16 位扩展 ISA 数据总线。

表 6-1 给出了 PC/104 管脚的定义。列 A 和列 B 是原始 ISA 信号定义，包括与存储器相连的 8 位数据线（SD0 ~ SD7）和 20 位地址线（SA0 ~ SA19），I/O 读写线（SMEMW*，SMEMR*，IOW*），若干 IRQ 管脚，以及 DMA 信号线（以“D”开头）。连接器提供的电压有 +5V、-5V、+12V、-12V 和 GND。而通常只使用 +5V，除非有驱动 EISA232 等的需求。

表 6-1 PC/104 连接器管脚定义，包含双列连接器 J1/P1 和 J2/P2，其中低电平有效信号用“*”标记。表中的两个“key”是 0.1 英寸连接器上孔的位置

管脚号	J1/P1 列 A	J1/P1 列 B	J2/P2 列 C1	J2/P2 列 D1
0	—	—	GND	GND
1	IOCHCHK*	GND	SBHE*	MEMCS16*
2	SD7	RESETDRV	LA23	IOCS16*
3	SD6	+5V	LA22	IRQ10
4	SD5	IRQ9	LA21	IRQ11
5	SD4	-5V	LA20	IRQ12
6	SD3	DRQ2	LA19	IRQ15
7	SD2	-12V	LA18	IRQ14
8	SD1	ENDXFR*	LA17	DACK0*
9	SD0	+12V	MEMR*	DRQ0
10	IOCHRDY	key	MEMW*	DACK5*
11	AEN	SMEMW*	SD8	DRQ5
12	SA19	SMEMR*	SD9	DACK6*
13	SA18	IOW*	SD10	DRQ6
14	SA17	IOR*	SD11	DACK7*
15	SA16	DACK3*	SD12	DRQ7
16	SA15	DRQ3	SD13	+5V
17	SA14	DACK1*	SD14	MASTER*
18	SA13	DRQ1	SD15	GND
19	SA12	REFRESH*	key	GND
20	SA11	SYSCLK		
21	SA10	IRQ7		
22	SA9	IRQ6		
23	SA8	IRQ5		
24	SA7	IRQ4		
25	SA6	IRQ3		
26	SA5	DACK2*		
27	SA4	TC		
28	SA3	BALE		
29	SA2	+5V		
30	SA1	OSC		
31	SA0	GND		
32	GND	GND		

包括列 C1 和列 D1 的第二个连接器提供了更大的地址空间，并把数据总线扩展到 16 位（同时提供了更多 DMA 功能）。这是一个并行总线，以 SYSCLK 实现信号之间的操作同步。

6.3.2.3 PCI

外部设备单元互联（PCI）总线于 20 世纪 90 年代早期发布，全面替代了 ISA/EISA。虽然 USB 近年来也成为那些通过内部插卡形式连接的外部设备的又一种可选接口，但 PCI 恐怕仍然是当代应用最为广泛的 PC 机内部总线。另外，以更加快速的串行连接为基础的 PCI 增强型（PCIe）在最近的系统中正在逐步取代 PCI。

与 ISA 类似，PCI 也是同步总线，工作时钟为 33MHz（或 66MHz）。另外与 EISA 类似，PCI 一般采用 32 位数据总线，在较长的连接器中也可以采用 64 位版本。连接器的电压可以不同，有 3.3V 和 5V 两个版本。不同版本的连接器上有不同的凹槽，以防止插入错误的连接器（一些“通用”卡上有两种凹槽，因而可以插入到两种版本的系统中）。与 ISA 相同，PCI 也提供了 +12V和 -12V 管脚，但通常不会使用。

PCI 总线复用地址和数据管脚 AD0 ~ AD31（64 位版本中到 AD63），保证了快速数据传输和大的存取空间。PCI 定义了总线仲裁系统，使得任何与总线相连的设备可以请求总线，而仲裁器对于这些请求给予许可应答。总线上置控制信号的设备为主设备，也称为动作发起方（initiator），而从设备称为动作接收方（target）。这实际意味着驱动总线的电压信号可以来自总线上的任何设备。这一点对 PCI 总线上信号的完整性有很大影响。因而，对于所有总线上的设备而言 PCI 实现了非常严格的信号条件规范。

可能是因为始终没有忘记与 ISA 和 EISA 相关的可用性问题，PCI 器件必须实现通过总线可存取的寄存器以标识器件的种类、生产商、项目编号等。更重要的是，这些寄存器还定义了器件
261 I/O 地址、中断相关细节以及存储器范围。

6.3.2.4 LVDS

低电压差分信号（LVDS）是一种非常高速的差分串行方案，它利用同步的小范围电压变化来表示数据位。它的宣传词为“以毫瓦获得吉比特”，原因是 LVDS 发出信号速度超过每秒
262 2Gbit。

注意 LVDS 不是像 ISA、PCI 这样的总线协议。它只是一个物理层通信方案（参见附录 B 可了解这种系统的分层结构）。然而，LVDS 已被现存的许多总线标准采纳。下面将要讨论的扩展 PCI 就是其中一个例子。

在 LVDS 中每个信号通过两条线发送。它们是差分信号，两条线上电压的不同组合表示逻辑 0 和逻辑 1。差分发送方案可以排除常见形式噪声的干扰，这种噪声的特点是在两条线上同时出现（例如电力线噪声以及来自附近其他器件的噪声）。事实上，LVDS 能够用于噪声水平超过信号电压的情况。

这种抗噪性使得 LVDS 连接可以工作在较低的电压变化范围。因此需要相对较小的功耗，并能够达到较快速度和产生较低的电磁干扰。图 6-3 为 LVDS 信号传输的示意图，说明了系统差分信号的性质及其抗噪声原理。

LVDS 的电压变化范围通常为 0.25 ~ 0.3V。由于信号变换（和传输）速度取决于信号从一个状态变到另一个状态所需的时间，而 LVDS 的电压变化范围非常小，因此信号变换速度就很快。传输系统的功耗与电压的平方成正比，因此像 LVDS 这样的低电压通信方案的功耗比 3.3V 或 5V 逻辑系统明显要低。类似地，低电压变换范围还使得 LVDS 产生的电磁干扰很低。

263 采用差分线传输信号还意味着当一根线上的电压增高时另一根线上的电压降低。如果我们将此与驱动电流相关联，则在任何时刻发送器件必须有驱动电流进入一条线、离开另一条线。当一个系统设计正确时，电流入出可以达到平衡，这与大多数电压变换方案效果不同，那些方案在

信号变化的瞬间电流会出现尖峰。供电电流尖峰转换成供电电压波动，因此会影响系统中的其他电路。

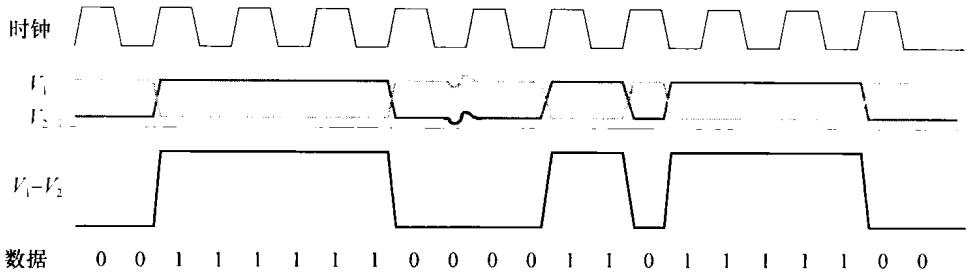


图6-3 低电压差分信号（LVDS）示意图，显示了两个被传输的差分信号。在接收端计算这两个信号的差值（$V_1 - V_2$）并以此确定在每个时钟周期传输的数据（见底部）。虽然在接收端和发射端都需要精确地同步时间信息，但实际上只有两个信号 V_1 和 V_2（灰色阴影部分）被传输。在两个被传输信号中可以看到存在少量噪声，但在接收端取差分后被消除了

LVDS 接收器通常需要从差分数据线中提取时钟信号。时钟恢复处理比较复杂。但是，若要与数据同时但分立地传输时钟信号，通常还需要一对差分信号线。总线 LVDS（BLVDS）是 LVDS 的变型，允许多个器件共享一对物理差分信号线。

前面提到的 PCI 增强型（PCIe）在台式机系统中逐步替代了 PCI。PCIe 通常需要指明有多少通道可用。例如，PCIe 1x 有 1 条通道可用，PCIe 4x 有 4 条通道可用，PCIe 32x 有 32 条通道可用，中间的操作过程相同。每个通道是一对 LVDS 发送器和接收器（即 4 个电连接器，两个在同一个方向上）。每个通道的工作频率为 2.5GHz。

PCIe 1x 连接器非常小，由 36 个管脚组成，数据传输率至少为 500Mbit/s（不考虑协议产生的开销）。PCIe 16x 连接器的尺寸与并行 PCI 连接器相似，但要快得多。

6.3.3 输入/输出总线

下表列出了典型的输入/输出（I/O）总线，其中一些在个人计算机体系结构中比较常见（USB 不在其中，之后讨论）。

总线名称	类型	速度	备注
EIA232，即 RS232	串行	115200bit/s	-12V 和 0V
EIA422，即 RS422	平衡串行 32 器件多接收端	最高 10Mbit/s	最低速时可传输 1km
EIA485，即 RS485	与 422 相同但多输出端	最高 10Mbit/s	最低速时可传输 1km
DDC，显示数据通道	串行，数据线，时钟线，地线	基于 I^2C 总线	
PS/2，键盘和鼠标	串行，6 管脚的 minDIN	电特性同 AT 接口	
IEEE1284，打印机接口	并行，25 管脚 D	最高 150kbyte/s	最长 8m

264

EIA 标准由电子工业联盟（Electronic Industries Alliance）批准，它采用“RS”为前缀表示推荐标准（即尚未批准的建议标准）。例如，EIA232 在没有成为正式标准前称为 RS232。然而，由于它以 RS 前缀的形式应用于几乎所有的家庭和台式计算机中，因此这个名字沿用至今。可见，相对于消费者市场的采纳速度，标准处理速度存在滞后的问题，这对于各标准组织而言也许是一个教训。

6.3.4 外设器件总线

下面给出一些常见的外设总线。近年来该领域一直朝着简单即插即用串行总线方向发展。很多年长的计算机工程师肯定还记得20世纪80年代将打印机连接到计算机上时的痛苦，当时串行外设连接经常出错，只有并行总线（即6.3.3节中所述的IEEE1284）才被认为是安全的选择。

- USB1.2，通用串行总线，原来是为键盘和鼠标这类设备设计的一种串行格式，但之后被广泛用于各种外设上。USB1.2的最长距离约7m，基本数据传输速率为12Mbit/s。作为一个串行总线，其带宽主机连接到所有器件共享，且每个器件要付出很大的控制代价。采用USB的主要原因也许是它可以给所有外设供电，使各个外设不需要分立电源及电源线。
- USB2.0，是因为引入了Firewire（见下面）而出现的。它很大程度上改进了USB1.2的速度——提高到480Mbit/s。在USB1.2和USB2.0之间，Firewire在视频市场上获得了很大的份额，成为将视频信息传输到计算机的业界认可方法。
- Firewire，由苹果公司开发，也是一种串行格式，被批准为IEEE1394标准，传输速率为400Mbit/s。IEEE1394b将速率提高一倍，达到800Mbit/s。但最大线长为4.5m左右。与USB类似，Firewire能够给外设供电，但厂商并没有在电压或电流上达成一致标准。
- PCMCIA，个人计算机存储卡国际协会（6.3.2节中曾简单提及）于20世纪90年代初基于ATA或IDE接口开发了插卡接口。它是并行接口，有一些变种，具有很高的速度。它后来演变成紧致闪存（compact flash，CF）接口。
- 多媒体卡（MMC）是一种串行接口，主要用于机顶盒和便携式音频播放器中的闪存卡接口。后面演化成安全数字（SD和xD）存储卡格式，它保持着串行接口性质，但允许多位并行传输。索尼的记忆棒是又一种产品，在规格说明和小尺寸封装方面与MMC类似。

[205]

6.3.5 与网络设备的接口

近年来，网络无处不在。人们长时间离开网络就会感觉若有所失，片上系统设计师没有忽略掉这一趋势，他们将处理网络连接的硬件模块集成到现代嵌入式处理器中。

典型地，介质存取控制器（MAC）硬件模块被集成到芯片中，而物理层驱动器（PHY）还没有被集成进去，主要原因是以太网物理层接口的模拟驱动和不同的电压要求。但可以买到将MAC和PHY结合在一起的器件。因此，预计MAC和PHY有可能完全被集成到一个片上系统中。当前的集成方案与图6-4所示的类似。

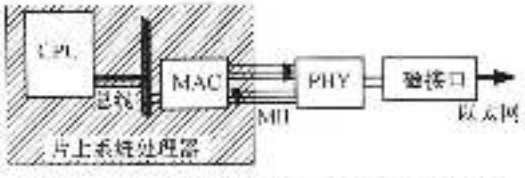


图6-4 网络（以太网）数据通过MAC层送到CPU的示意图

假设网络基础设施是以太网，图6-4中所示的连接口与PHY相连形成了一个常见的系统方案。MAC和PHY之间的接口是传输介质独立的接口（Media-Independent Interface，MII），表示连信不简单局限于有线以太网。它也可用诸符合MII标准的光接口，只是可能需要不同的PHY器件。无线是另外一种越来越常见的通信方式，它也基于相同的标准处理过程（将于6.6节讨论）。

6.4 实时性问题

还记得现代计算机的祖先们吗？它们的计算机占满整个房间，完成一张张的数据运算，通过离散开关或穿孔卡片进行编程，需要几分钟甚至几小时才能得出结果。它们与嵌入式系统相差甚远，那些小小的器件可以植入到人体中调节血液里的化学物质，也可以植入到家用汽车中控制

刹车系统。后者是硬实时系统的典型例子。所谓硬实时是指必须在一定的时间内做出响应，否则后果非常严重。

以前的系统没有实时性要求。它的设计者可能会考虑提高计算速度以便可以早些回家，但不会去设计一个计算机在毫秒级对外部激励做出响应。这意味着传统计算机体系结构和编程语言开发没有考虑到实时响应的需求。 266

今天，由于嵌入式处理器远多于台式机（而台式机又远多于大型机），计算机世界越来越多地在实时运行。大量实时地与真实世界相互作用的嵌入式器件就是实时系统，它们或者是硬实时或者是软实时（若错过时限不会造成灾难性后果，则称为软实时）。

6.4.1 外部激励

外部激励可以有多种形式，但通常与一些传感器相关。例如核反应堆的过温度传感器、汽车中由气囊控制的加速度传感器、电机管理系统中的真空开关，以及老式鼠标中的光电门等。这些传感器在任何时候都有可能被触发。

其他外部激励还有以太网上有数据到来，或数据通过并行端口从 PC 机输出到激光打印机上。这两种激励来源于计算机本身，但由于它们到达目标处理器的时间不可预测，因此它们对于目标处理器而言还是实时激励。

6.4.2 中断

到达实时处理器的激励都会被转换成标准形式来触发 CPU。这些中断信号通常是低电平有效，与 CPU 的中断管脚相连（在片上系统中一个片内信号也可以被转换成低电平有效输入与 CPU 相连）。

多数处理器能够同时支持多个中断信号。这些信号按优先级排列，当两个或多个中断同时被触发时，首先响应优先级最高的中断。

在 6.5 节中将更全面地讨论中断，这里只是需要认识到 CPU 只用很短的时间来发现中断信号，之后要经过一段较长的时间 CPU 才能够处理这个中断，而针对该中断实际完成服务则需要更长的时间。中断服务通过中断服务例程（ISR）完成——在 5.6.3 节中讨论影子寄存器时曾有简单介绍。当设计一个实时系统时，必须确定中断时序并将它们与任务的时间范围相关联（见 6.4.4 节）。

6.4.3 实时性定义

前面提到了软时限和硬时限，它们均是实时性约束，区别在于错过时限导致的后果不同。错过硬时限对系统而言是灾难性的，错过软时限是不幸的但非致命失败。

这些术语也与整个系统有关：*硬实时系统*包括硬时限要求。如果所有时限都是软时限，则这个系统是*软实时系统*。在选择操作系统时，也要考虑硬实时的程度。例如 uCos 能够满足硬时限 267
要求，而嵌入式 Linux 通常只能进行软实时响应。SymbianOS 为相对硬实时，而微软的 Windows CE 则比较偏软实时——这就是完成关键任务的实时系统中不使用 Windows CE 的原因。

一个*任务*就是一段实现一个或多个功能的程序代码，可能与实时输入或输出相关联。在多任务*实时操作系统*（RTOS）中，会有几个任务并发运行，每一个任务带有一个优先级。多数系统围绕着中断或时钟设计，这样每次发生中断时，都会触发一个相应的任务来处理它。其他一些任务会由时钟到期来触发。任务本身可以是中断服务例程，但多数情况下任务是单独的代码（目的是保持 ISR 尽量短），因此，当 ISR 运行时利用 RTOS 专门的函数启用相应的任务。这些专门函数包括 semaphores、queues 和 mailboxs，但它们超出了本书讨论的范围，读者可以在大多数

介绍实时系统的标准教科书中找到相关内容。

许多任务大部分时间处于休眠状态，等待被 ISR 或其他任务唤醒，此时低优先级的后台任务运行完成一些系统相关的功能或日志，其中也包括了调整优先级的工作。

6.4.4 时间范围参数

任务的时间范围（temporal scope）参数是一个用以描述实时需求的五元组。对于有时限约束的多任务实时系统而言这是非常有用的形式化描述。

下面是定义时间范围的五个参数，除特别说明外，所有的时间起始于触发该任务的事件到来的时刻。

任务开始前最小延时	通常为0，有时也会特别要求
任务开始前最大延时	原则上讲应该尽快地响应中断，但有时会给出一个硬时限
任务处理的最长时间	从任务开始到结束的总时间
任务占用的 CPU 时间	它也许不同于上面的参数，因为任务有可能被中断，因此推迟一段时间，但这期间不占用 CPU
任务最大完成时间	从事件触发到任务完成的时间

多数时间范围参数可以通过分析系统需求来确定，但占用 CPU 时间只能通过记录任务执行的指令数或通过 OS 提供的测量处理器周期数的方法得到。关于 CPU 时间有一点需要注意，即条
268 件循环根据所处理的数据可长可短。而所说的 CPU 时间是指所有循环均按照可能的最长时间来计算。这也就是设计紧凑任务代码的原因。

一个任务图如图 6-5 所示。图中列有三个有效任务及它们占用 CPU 的时间。竖线表示调度器工作的时间点。如果需要，它可以切换任务。调度器通常被设计为一个系统任务，决定什么时刻选择什么用户任务占用 CPU。根据 RTOS 类型的不同，可以通过两种方式激活调度器——以固定的时间间隔或在任务调度点通过软件本身的调用功能实现。任务调度点一般出现在完成操作系统级任务库函数的时候，例如简单的有 printf()，通常在先进先出（FIFO）、队列、mailbox、信号量操作时刻出现。有时将若干方法组合起来激活调度器。

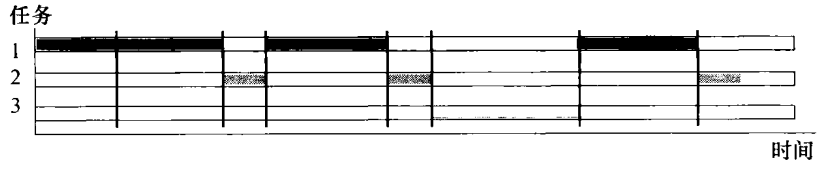


图 6-5 在单 CPU 上执行的三个任务的调度图

在图 6-5 的任务图中，调度器第一次被激活时（第一个竖线）任务 1 正在执行。调度器没有进行任务切换，因此任务 1 继续执行，其原因可能是任务 1 在三个任务中优先级最高。习惯上也因此将任务 1 放在图的最上边。

任务 2 每次出现的长度相同，说明它每次运行时均完成相同的工作。

从上面这个例子也能够稍稍理解调度器的工作方法。首先，任务按优先级排列。最高优先级的是调度器，最低优先级的是系统空闲时才执行的空闲任务（idle task）。在嵌入式系统中，空闲任务可以处理低优先级 I/O，例如打印调试信息或刷新 LED（利用低优先级任务打印调试信息是很常见的，但是当调试一个整体上崩溃的程序时，这样的安排就起不到帮助调试的作用了，因为无法从崩溃的任务那里得到调试信息，而且如果这个任务一直在运行，那么空闲任务就没有机会运行）。

调度器中有一个表用来记录所有任务的状态：运行、等待、休眠。在一个时刻只会有一个运

行态任务，可以有多个处于等待状态的任务（即等待运行机会）。休眠任务处于停止状态，可能是临时等待信号量或某个数据以进入队列或 mailbox。框 6.4 列出了一些设置调度优先级的方法。 269

框 6.4 调度优先级

给定实时系统中的一些任务，设计者面临的一个困难是如何给任务设置优先级以保证它们被正确调度。这一点非常重要——一些选择可能会导致系统不能满足要求的时限（即不可调度），而一些小小的改变就有可能使得系统正常工作。下面是一些常见的形式化优先级设置方法。它们均需要系统中所有任务的时间范围参数。

时限单调调度：最紧时限的任务拥有最高优先级。

频度单调调度：越经常被触发的任务其优先级越高。

最早时限最先调度：这是一个动态调度方法，根据即将出现的时限赋予相应任务高优先级。

其他还有**最重要最优先**、**自组织**、**轮询**和其他一些混合方案（多数都自称比其他的好）。

6.4.5 硬件体系结构对实时操作系统的支持

这是一本计算机体系结构而不是实时系统的教科书，因此它更多考虑的是在处理器上运行实时操作系统对硬件的需求，可不讨论实时性对操作系统的需求。让我们来回顾一下当实时事件发生时经历的过程。

1	事件引发对处理器的中断信号
2	处理器发现中断
3	处理器需要一小段时间完成正在进行的工作，转到中断向量，再由此获得与该中断向量关联的 ISR 入口地址
4	处理器从当前执行的程序转到一个中断服务例程
5	ISR 应答中断，释放一个任务处理该事件
6	任何较高优先级的等待态任务优先执行
7	最后，切换到处理该事件的任务上下文
8	该任务处理这个事件

这 8 个步骤（详见 6.5.2 节）中的每一个都会占用一些时间，因此会影响系统的实时响应时间。

硬件中断支持（详见 6.5 节）能够极大地改进响应时间。而服务于任务的 OS 功能，特别是从前一个运行代码切换到 ISR，以及任务之间的切换，是十分耗时的，也可以通过硬件加速。 270

首先，影子寄存器（见 5.6.3 节）可以加速从一段代码到另一段代码的切换。ARM 实现了几个影子寄存器集合，其中之一是*监控程序*（supervisor），专门用于支持 OS 代码，例如调度器，因此，运行调度器不需要进行耗时的上下文存储和恢复过程。

一些 CPU 采用了更进一步的设计，实现了若干组寄存器，给每一组寄存器分配单独的任务，因此，任务之间切换就很容易了。不需要上下文存储和恢复，简单地从一个寄存器组切换到另一个寄存器组即可。

也可以用硬件 FIFO 和堆栈来高效实现 mailbox 和队列以进行任务之间的通信（否则就要用软件实现数据在不同存储块之间的搬移）。硬件实现方案因为大小固定而缺少灵活性，但速度非常快。

虽然硬件实现的调度器没有被计算机体系结构设计师采纳，但理论上讲这是可行的。也许对于调度过程最高性能的硬件支持是双核（或多核）处理器，它可以支持超线程之类的技术。这样，两个任务就可以同时运行了。Intel 的最新处理器 Centrino Core 中采用了一种 MIMD 处理结构（见 2.1.1 节）。其他制造商肯定也会紧随其后（MIMD 和双核的有关内容参见

5.8.1 节)。

6.5 中断和中断处理

本节将讨论中断及其开销，并考虑如何提高服务速度。我们在 5.6.3 节已经描述了用于中断服务例程（ISR）的影子寄存器，所以本节不再论述它们对提高性能的作用。

6.5.1 中断的重要性

中断及其处理是计算机体系结构和嵌入式软件工程最重要的课题之一。随着计算机与真实世界之间相互作用程度的不断增加以及嵌入式计算机系统的作用越来越关键，中断承担起越来越重要的工作，其中就包括确保处理器能够尽快地响应实时事件。

前面讨论过实时事件，这里还有必要对三个与中断相关的时序特征做一个回顾。

1. 中断探测时间——从 CPU 发现事件发生时刻到能够做出响应动作时刻之间的时间。

2. 中断响应时间——从 CPU 开始服务一个事件时刻到按最坏情况计算所有动作均完成时刻之间的时间。

271

3. 最小中断区间——从一个中断发生时刻到该中断可以再发生的最早时刻之间的时间。如果中断没有规律性，则考虑所允许的最短时间间隔。

6.5.2 中断过程

搞清楚一个中断线被置有效后会发生什么是非常重要的，因为这些事件对于系统体系结构有巨大影响。下面进一步细化 6.4.5 节给出的表格。

1	外部事件引发对处理器的中断信号
2	处理器发现中断
3	处理器首先完成正在进行的工作，然后转到中断向量，再由此获得与该中断向量关联的 ISR 入口地址
4	处理器从当前执行的程序转到对应的中断服务例程
5	ISR 应答中断，释放所有等待该事件的任务
6	任何较高优先级的等待态任务优先执行
7	最后，切换到处理该事件的任务上下文
8	该任务处理这个事件

下面我们具体看看前 5 步，它们受体系结构的影响非常大。

6.5.2.1 中断事件通知处理器

按照常规，给 CPU 的中断信号是低电平有效，可以是沿触发也可以是电平触发。当状态发生变化时沿触发中断信号通知 CPU。处理器以尽可能快的速度响应沿触发信号——在这个过程中中断线有可能已经自复位。像按键这类事件有可能产生这种中断（无论键按下的时间有多长，处理器都以同样的方式响应）。

电平触发中断在物理上与此类似——但处理器在事先定义的时间采样中断线状态，例如每周期一次。一旦中断信号给出，需要在处理器采样前提前一个时间段设定，这个时间段的长度可以设置。例如，应该提前三个采样间隔而不是一个采样间隔，以防止因噪声尖锋带来误判。

中断信号一旦发出，无论物理上的中断线是否停用，内部触发电路都应当保持设置。最终，处理器中的某种编码将会用于服务这个中断。问题是：假若在前面的中断没有被服务之前其他事件又触发了中断线，将会带来怎样的结果呢？同样，其结果依赖于采用什么处理器，但一般情况下忽略第二次中断事件。这是因为内部关于“中断发生”的标识一直保持，直到在 ISR 中清

除中断指令时才能复位。

272

然而，过去也有一些处理器能够将中断信号排队（特别是响应中断速度较慢的处理器）。将中断信号排队听起来是一个好想法，但使得实时处理变得很复杂，因此当今通常不把它作为一种潜在硬件方案。最好的解决方法是无论什么中断事件发生都以尽可能快的速度处理。

6.5.2.2 CPU 完成正在进行的工作

现代处理器不能够在执行一条指令的过程中间被中断——它们必须首先完成正在执行的指令。过去的 CISC 处理器用许多周期完成一条指令，这导致中断响应时间被延后。例如，据说 DEC 的 VAX 计算机需要 1ms 时间完成一条指令，这对于正在等待服务的中断而言是一个相当长的时间（以音频处理为例，这意味着单个中断可以支持的最高采样率为 1kHz，远低于今天 MP3 播放器的 48kHz 和 44.1kHz）。

人们试图在使用微码的处理器中允许在指令执行过程中中断，但这样做十分复杂因此没有被广泛采纳。实时系统设计者由于 RISC 处理器（3.2.6 节）的出现得到解脱，其单周期指令的设计非常明智。这意味着完成一条指令的最长时间理论上为一个指令周期，在 RISC 处理器上一个指令周期通常很短。这还意味着在这样短的时间内就可以完成中断发现以及转到中断向量的过程。

实际上，一些设计者并没有完全坚持 RISC 的概念。例如 ARM，提供多周期寄存器取和寄存器存指令，这些指令对于快速数据搬移或上下文存储和恢复很有用，但完成它们最多需要花费 16 周期。因此，中断获得中断向量在最坏情况下的等待时间为 16 周期。

还有一件值得注意的事情，就是流水线的作用。在以流水线方式执行指令的体系结构中，虽然每个周期有一条指令进入到流水线，但需要花费 n 周期才能够完成一条指令，n 为流水线的长度。若没有复杂的专用硬件支持，则一个影子寄存器系统在跳转到 ISR 之前需要等待当前指令全部流过流水线并保存好结果。流水线对于提高指令吞吐量是非常有效的，但对于中断而言却降低了中断响应速度。

6.5.2.3 转入中断服务例程

处理中断的传统方法是一旦中断发生，就将一个预先设置好的值放到程序计数器中，由此引起 CPU 跳转到一个特殊的地址，这个特殊地址根据 CPU 的类型而有所不同。这个地址对应的存储单元在主存中称为中断向量。

在 ARM 中，中断向量开始于主存地址 0。地址 0 对应着复位向量（rest vector），即在上电或复位时 CPU 的入口地址。CPU 的每一个事件和中断按顺序排列。向量表中的内容是转移到事件处理程序的分支指令。复位向量中存储着跳转到启动程序的分支指令。在 IRQ1 中是跳转到处理 IRQ1 中断的 ISR 的分支指令（当从 C 语言转换为汇编程序时，经常使用双下划线表示）。

273

下面是 ARM 程序中典型的中断向量表：

```
B   __start
B   _undefined_instruction
B   _software_interrupt
B   _prefetch_abort
B   _data_abort
B   _not_used
B   _irq
B   _fiq
```

图 6-6 刻画了在执行程序的过程中如何利用中断向量表处理中断事件。

可以看到，首先从复位向量开始，执行跳转到起始程序的分支指令（B_ _start）。程序正常执行（如左边的实线箭头所示）直到在第...条 SUB 指令执行过程中发生中断，该指令完成后跳转到该中断对应的中断向量，在这个例子中是 IRQ。虽然图中没有给出，但我们可以假设在此过程中执行了向影子寄存器组的切换。IRQ 的中断向量中包含一条到相应中断服务例程的分支指令，在这里是 ISR1。这个程序服务于该中断，一旦完成就返回到原来被中断的程序中的下一条指令。同样，虽然图中没有给出，我们依然可以假设在从中断返回时从影子寄存器组切换回到原来的寄存器组。在一些处理器中切换过程自动完成，但在一些处理器中则需要使用不同的返回指令（例如，TMS320C50 的 RET 指令完成从子程序返回，以及 RETI 指令完成从中断返回，RETI 在返回时自动弹出影子寄存器组），很显然这种情况下机器使用了影子寄存器组，因为 ISR 和主程序代码中使用了相同的寄存器名，而没有显式地保存和恢复上下文的操作。

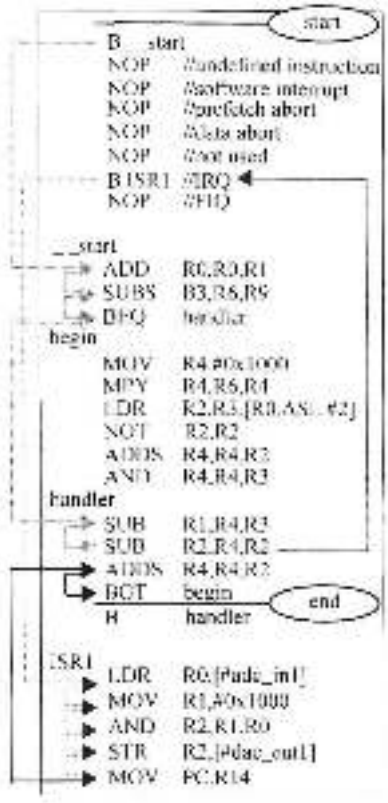


图 6-6 通过分支到中断向量表调用中断服务例程。图中从上电开始的执行包括位于 _start 的初始化分支。在 handler 子程序的第二个 SUB 执行过程中中断发生，因此，在 ADDS 前转入处理该中断的服务程序。实线箭头表示正常操作。虚线箭头表示中断处理过程中的控制流。

还有一件事值得一提：向量表中没有使用的中断都对应着 NOP 指令，这意味着如果这类中断发生，则执行 NOP 指令，然后执行下一条 NOP 指令，如此下去直到有什么事情发生。例如，如果 data_abort 事件发生（由于某种存储器错误），则跳转到中断向量，执行 NOP，再执行下一个 NOP，直到最后到达跳转到 ISR 的分支语句。此时，虽然没有 IRQ 发生，但 ISR1 将被执行。因此，对于所有的中断，无论它们有没有被使用，最好都有对应的中断服务例程，可以只是输出错误信息，这样当该中断发生时至少有错误提示。

274

或 6.5 给出了 ARM 中断时序的一个例子。

例 6.5 ARM 中断时序计算

ARM 支持两个外部中断源，标准中断（IRQ）和快速中断（FIQ），FIQ 有较高优先级。对于 FIQ 有 8 个影子寄存器，而对 IRQ 只有一个影子寄存器（假设我们需要 4 个寄存器）。由于指令是 32 位，外部总线是 16 位，因此，从存储器读取到指令要或从寄存器写到存储器需要 2 个周期。

IRQ 中断向量位于中断向量表的中部，而 FIQ 则是最后一个（这意味着如果中断程序直接放到这个位置，则不需要 FIQ 从中断向量表到中断服务例程跳转的分支指令）。

ARM7 为最长指令是从存储器的连续空间批量取 16 个到寄存器，需要 20 个周期。我们用 3 个周期来做首次取，假设分支指令需要 3 个周期。考虑一个包含对于 FIQ 和 IRQ 的操作，即 SDRAM 刷新操作，需要消耗 25 个周期完成刷新操作，假定时钟为 66MHz。

我们可以确定系统最多允许多少次连续 IRQ 和 FIQ，以周期为单位。下面是当 IRQ 有效且连续发生的事情。

1. 响应中断需要的时间：3 周期

2. 最坏情况下完成当前指令需要的时间：20 个周

3. 若 SDRAM 需要刷新，则需要等待：25 周期

此时，CPU 可以响应中断。

4. 从当前位置跳转到中断向量表以读取其中某行：2 周期

5. 执行向量表中的指令，分支到 ISR：2 周期

此时进入到 ISR。

6. 上下文保存 3 个寄存器（我们需要 4 个，另一个是影子寄存器）2×3=6 周期

7. 执行响应中断的第一条指令：2 周期

总指令周期数：60 周期

总时间（66MHz 处理器每周期大约为 15ns）：0.9μs

就中断响应而言 1μs 是比较快的。的确，中断响应速度快是 ARM 体系结构的优点之一。

下面我们考虑 FIQ 的情况。在这个例子中，主要有两点不同。一点是有更多的寄存器可以通过影子寄存器方式保存上下文；另一点是 FIQ 代码可以直接驻存在中断向量表中，不需要跳转。因此 FIQ 与 IRQ 的不同在于：

8. 不需要分支到 ISR：2 周期

9. FIQ 有 6 个影子寄存器，因此不需要保存上下文：6 周期

总指令周期数：52 周期

总时间（66MHz 处理器每周期大约为 15ns）：0.78μs

在时钟频率不变的前提下我们还可以做进一步改进吗？是的，我们可以在程序中避免使用 20 周期的最长时间指令，可以变换存储器技术。避免使用批量取数－存数指令和去掉 SDRAM 刷新周期，能够帮助我们达到 0.2μs 的总时间。注意基于 ARM7 的处理器通常不采用 SDRAM，而基于 ARM9 或以上版本的处理器倾向于使用。

6.5.2.4 中断重定向

275
~
276

关于中断向量表还有一点需要解释。假设存储器的低地址部分映射到 ROM，因为其中包含引导加载程序（bootloader），而高地址部分的存储器映射包括 RAM。如果没有修改中断向量表的机制，就意味着存储在 RAM 中的代码不能利用中断向量。这对 RAM 中想使用中断的代码是很不利的。

因此，硬件中通常会有一个机制将中断向量重新映射到存储器的另一个地址空间（框 6.6 给出了一个 ARM 处理器上的例子）。在启动复位时，执行引导加载程序，装载某个程序并开始执行。这个程序导致中断向量表被重新映射到 RAM，映射到一个独占的地址空间，因此可以为所有的中断修改其中断向量。

框 6.6 引导过程中存储器重新映射

通常需要执行两个分支才能够进入 ISR，一些处理器采用了稍微变化的方法绕过这个问题。例如基于 ARM 的 Intel IXP425 XScale，在上电初始化阶段，将闪存或 ROM 映射到存储器的地址 0 及以上的空间，以便执行引导（boot）程序。通过设置 CPU 内部的一个寄存器还可以将引导程序区映射到存储器的最上段空间，而将 SDRAM 映射到地址 0 及以上空间。

因此，引导加载程序只需要确保包含中断向量的程序被装载到内存，且最低段地址空间对应着 RAM，则引导加载程序可以发出重新映射命令。

可是事情并不这样简单，因为引导加载程序本身从 ROM 地址空间执行，当重新映射发生时，引导加载程序代码就会消失。换句话说，如果程序计数器（PC）位于地址 0x00000104 时执行重新映射指令，则当 PC 递增指向下一条指令地址 0x00000108 时（因为指令长度为 32 位，因此 PC 每次递增 4 字节），那条指令已经不在那里，而是被重新映射到较高地址空间上了。

有一个简单的技巧可以解决该问题。看看你能否在继续往下读之前知道答案。

如果在重新映射之后相同地址上恰好出现原来的代码，我们的问题就解决了。实际上这就意味着将引导加载程序代码拷贝到 RAM 的较高地址空间，然后再进行重新映射，XScale 的一些引导加载程序版本正是

关闭了这种方法，如 U-Boot。

另一种解决方法是将引导加载程序分成两个部分或两个阶段。第一阶段中将第二阶段执行的代码转入到 RAM 地址空间，然后第一阶段跳转到第二阶段并完成重新映射。

当使用实时操作系统 RTOS 时，可能会有第二层向量化：所有的中断触发 OS 中同一个 ISR，而 OS 中注册了外部函数与相关事件之间的对应关系。当这样注册的某个事件发生时，中断照常进行，但 OS 中的 ISR 通过分支跳转到注册的中断处理程序。这种机制可以在不支持硬件中断共享的处理器或片上系统中实现中断共享。在这种情况下，OS 负责决定那一个共享中断发生并跳转到相应的处理程序。硬件中断共享方法将在 6.5.4 节中介绍。

277

6.5.3 高级中断处理

在标准中断处理过程的基础上，我们再来研究一下提高该过程效率的机制，即预先将中断转移地址取到寄存器中。

设想常规情况：当中断发生时，处理器将跳转到中断向量表中的指定位置。表项中包含一条（有时是两条）指令指示 CPU 转移到相应 ISR 的入口地址。这样，该过程需要两个顺序分支，如 5.2 节所述分支指令在采用流水线机制的机器中效率很低。因此，这种需要两次分支的方案不是个好方案。

可见，CPU 最好知道对应着每个事件中断向量表中存储的分支地址。为此，这个分支地址应当存储在处理器内部，即一些寄存器中，当中断发生时将该地址拷贝到程序计数器 PC 中。若向量地址寄存器可写，就可以对应不同事件填写向量地址。这使得 ISR 首地址可以存放在向量地址寄存器中。当中断发生时，处理器直接跳转到 ISR 首地址而不需要通过中断向量表——这种方法既可以用于共享中断，也可以用于独享中断。

这种方法的代价是需要一组可写寄存器（它比只读寄存器需要更多的硅面积）且中断控制器稍微复杂了一些。

6.5.4 共享中断

今天的很多计算机系统实现了共享中断。其最初原因是只有有限的集成电路管脚用于硬件中断，且在 CPU 内部用于控制中断的寄存器有限。因而导致许多中断共享很少的 CPU 物理上的中断。例如，ARM 有两个分立中断：一个中断请求（IRQ）和一个快速中断（FIQ），而一个基于 ARM 的典型片上系统嵌入式处理器往往会有 32 个中断源，它们共享 IRQ 和 FIQ 线。

当一个被共享的中断发生时，ISR 程序的开始部分就要读一个寄存器以区分这个共享中断是由哪个中断源触发的，之后调用相应的代码来响应中断。中断也可以由 RTOS 或软件中断触发。有些时候，一个很大的 ISR 可以完成服务很多共享中断的工作。

共享中断需要一个中断控制器。中断控制器可以是一个专门用于处理中断的集成电路芯片，也可以是在片上系统嵌入式处理器中集成一个先进中断控制器（Advanced Interrupt Controller，AIC）模块，后一种方案在当今的设计中更常见。图 6-7 是一个例子。

278

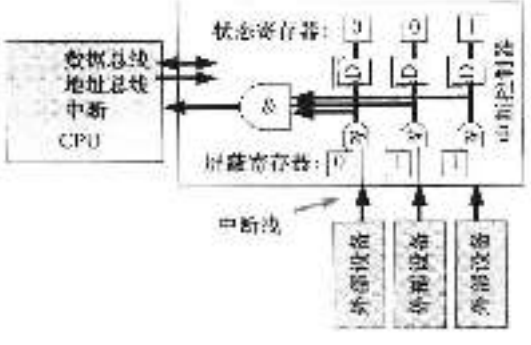


图 6-7 用于片上系统中中断控制模块的中断共享硬件框图

从这个例子可见 CPU 本身只有一条由三个外设共享的中断线。中断控制器内部有一个 CPU 可写寄存器，用于屏蔽共享这条中断线的某些中断，

而那些没有被屏蔽的中断则会引发 CPU 中断。

当 CPU 中断被触发时，CPU 通过读状态寄存器来确定引发该中断的中断源。通常，读状态寄存器动作同时完成清空寄存器的操作以便接受下一个中断请求（图中的逻辑电路没有包括这一部分）。

6.5.5 可重入代码

尽管中断信号为了保证触发中断响应保持了足够长的有效时间，且又有清除中断机制，但这并不意味着同一个中断可以立即重新触发。虽然具体情况随处理器的不同而不同，但大多数器件在服务某一中断的过程中不允许相同的中断再次被激活（即不允许再入（re-entrant）该中断），直到 ISR 执行完毕。或者在第一个中将断处理过程中将忽略第二个中断，或者在 ISR 完成时立即引发重新中断。

一些更先进的处理器允许高优先级事件中断较低优先级的 ISR，这通常要求硬件为每个 ISR 提供影子寄存器或提供很精心设计的上下文保存和恢复功能。

6.5.6 软件中断

软件中断（SWI）是低级别软件中断高级别代码的一种方法。在 RTOS 中，它通常用于操作系统干预任务级代码处理。在 ARM 处理器中，发出软件中断的代码为： 279

```
SWI 0x123456
```

它将触发影子寄存器集合的切换。这时处理器将进入到系统态（而常规程序工作在用户态）。ARM 在系统态下有权修改在用户态下不能修改的设置，系统态结束时跳转到中断向量表的第三项，即地址 8（参见 6.5.2 节中的 ARM 中断向量表）。

软件中断是一种典型的处理器捕获，对程序调试很有用。在软件某一行设置断点的一种方法就是用软件中断代替这条指令。一旦执行到这条指令时处理器就会被中断，跳转到软件中断向量，然后执行软件中断服务例程。

在软件中断服务例程中，调试软件可以存取内存和用户态下的寄存器。然后调试程序等待来自用户的命令。

6.6 无线

在一本计算机体系结构教科书中出现以“无线”为题目的章节是很少见的。这样做的原因是本书是从嵌入式的角度审视计算机体系结构，而越来越多的嵌入式系统是关于或为了实现无线通信而设计的。

因此，让我们简单考察一下无线技术与计算机特别是嵌入式系统的相关性。我们首先定位无线技术，然后讨论接口技术及其相关问题。附录 D 提供了更详细的信息并概述了嵌入式系统可能的解决方案。

6.6.1 无线技术

虽然无线工程师根据射频频带（RF）、信道带宽、发射功率、调制方式等将无线技术分为许多类别，但是，嵌入式系统工程师出于其目的将会考虑不同的问题：

- **与 CPU 的连接**——尤其是串行连接还是并行连接。6.6.2 节将详细讨论。
- **数据格式**——数据是以位、字节/字符、字还是包为单位传输。这不仅涉及连接方式，还涉及数据交换的标准，例如 USB 或 IP（网际协议）包。

- **数据速率**——通常以位每秒为度量单位（注意厂家给出的标称值往往未包括额外开销，例如分包、包头、校验码等，因此实际可用数据速率可能会远低于标称值）。当然，数据速率与应用需求相匹配十分重要，但值得注意的是对于实时应用而言数据速率并不一定与处理时间相关联。一个每秒可送出几兆位数据的系统对单个事件的反应时间可能慢于一个每秒只能送出几千位的系统。
- **外形因素**——包括物理尺寸、数量、天线大小等。低频器件往往需要较大的天线。
- **范围**——与发射功率相关。无线电管理法规对比会有约束。（根据频带和用途发射功率通常约束在0.25W，最多不超过1W。）
- **功耗因素**——与发射功率、范围和数据速率相关。
- **错误处理**——是需要保证无错误通信，还是不需要考虑错误处理？6.6.3节将详细讨论该问题。
- **CPU开销**——是一个需要考虑的重要因素。

当设计者需要在嵌入式系统中实现无线通信功能时，要考虑上述问题并寻求综合平衡点。

目前存在许多无线标准且适合嵌入式系统。附录D描述了其主要候选方案。本节重点讨论设计者在分析和评估候选方案时需要考虑的问题。

首先，图6-8给出了一个无线模块到应用处理器模块的连接框图。该应用处理器通常是应用系统中唯一与无线模块相连的CPU。

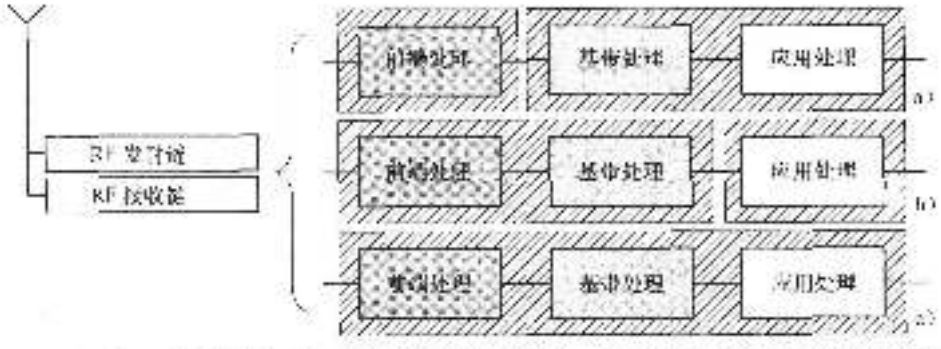


图6-8 用于嵌入式计算机的一个无线通信处理方案框图，其中包括两个计算器件负责无线通信处理和一个嵌入式应用处理。这些器件或者分别处理各个需求，或者a）基带处理与应用处理在一起完成，或者a）一个附加设备提供无线通信功能给应用处理器，或者c）在一个设备上处理无线通信和应用

很清楚，无线通信系统通常需要信号处理，且当今大多数嵌入式系统设计师采用的无线解决方案通常采用数字方法实现数字信号处理。前端处理（可以用模拟方法实现，但逐渐倾向于采用数字方法）是对接收到的或要发送的无线信号在最前端进行调节。要处理的数据速率往往在MHz或GHz量级，是位速率的若干倍。相反，基带处理是较慢的协议层计算，例如包处理、包错误校验、跟踪重试和重发等。

当系统设计者没有选择遵循标准而是定义自己的方案时，基带处理功能会放在应用处理器内完成（如图6-8a所示）。甚至有可能采用一个计算器件完成所有的处理（如图6-8c所示）。这在技术上是可行的，但会给应用处理器带来很大的协议处理开销，或者设计者无法免费获得这部分的源代码形式。因此，基带处理需要单独完成，或者在一个单独的器件中，或者集成到前端处理中（如图6-8b所示）。

将无线处理从应用处理器中分离出来的最重要原因可能是不想从头开始。重新设计一个可靠的无线通信系统的确是很困难的。当存在已证明可工作的产品时，最合理的选择就是使用它。

6.6.2 无线接口

我们在6.1节中已经将CPU总线分为串行和并行两种，对于无线连接我们也可以应用这种划分方法。虽然数据在空气中传播可以是串行、并行或某种混合方式，但是无线设备与CPU之间一定存在一个接口，或是串行接口或是并行接口。

简单、慢速的无线设备采用串行接口：如果在无线链路的一端送出一个串行数据，就能够在无线链路的另一端接收到这个数据。如果该链路存在误码控制，则可以假设收到的数据是（相对而言）没有错误的。否则，应用程序中就要加入误码校验功能。无线USB标准属于串行接口范畴。

基于网际协议的方案，例如IEEE802.11（Wi-Fi）和IEEE802.16（WiMAX），是基于块的。协议的处理单位是整个数据包。因此，为了提高效率，这类标准的无线模块与CPU的接口通常采用并行总线并通过直接存储器存取方式（见6.1.2节）进行数据传输。事实上这很像标准以太网设备与CPU的接口（见6.3.5节）。

6.6.3 无线相关问题

加入无线模块使得系统又增加了一种连接方式。当然，无线通信明显会影响系统功耗需求。 282
但是，更重要的还有以下问题必须加以考虑。

第一个就是前面提到过的一个问题：CPU开销。显然，当协议处理功能由应用处理器完成时，势必会占用很大一部分处理器时间（最坏情况是每一个接收到的包都有错）。然而，即使是采用分离器件完成无线信号处理和协议处理，而应用处理器只是负责输入输出，还是需要许多CPU周期去管理无线通信。

当需要考虑误码处理时，需要辨识和处理不同类型的误码。当然，以太网也面临同样问题。但以太网要么没有误码，要么整个包都接收不到，而无线连接的误码状况往往处在两个极端之间。

进一步的问题是安全问题——采用有线网络，很容易知道被连接者（只需要顺着线找即可）。但是无线连接是不可见的。设计者必须意识到接收到数据并作出回复的并不见得是正确的接收者。随着嵌入式系统中计算机技术的快速发展，更多的人依赖这样的系统生活和理财，但一些观察者已发现在这个领域里安全方面的进步速度远低于技术创新。

最后，无线信号本身渗透到发射天线周围的空间，这些信号时常会返回来进入到产生它的系统，成为系统中的总线和连接线上电噪声的主要来源。这就是*电磁干扰*问题，即EMI。最近的研究发现它是影响系统稳定性的重要因素。

对计算机系统设计者而言有两个主要影响。第一个问题，任何一个电子系统都是一个潜在的EMI源。不同的总线设计会引起不同级别的EMI。例如，ISA总线因其电压变化幅度较大且不平衡的特质，会比LDVS总线产生更大的电磁干扰。不同的存储器技术也会产生差异很大的EMI。计算机系统产生的EMI会影响到它周边的系统（一些读者可能还记得当像ZX Spectrum那样的家用计算机上电时所产生的EMI会使得它旁边的FM收音机停止工作），也会影响到系统中的其他部件。第二个问题，嵌入式系统设计者应当考虑将系统设计成为像ZX Spectrum那样在有干扰的环境下也能够工作。设计这样系统的方法不属于计算机体系结构的范围，因此在本书中不作详细讨论。在关于电路设计和PCB布局布线的文章和书籍中会涵盖相关内容。 283

6.7 小结

对于搭建一个计算机而言，有一个计算能力很强的CPU是很好的开端，但要想成功还取决于给CPU提供数据以及向外部输出结果的方法。无用的输入数据将会导致无用的输出数据，这

是一条计算的公理。而这条公理不仅适用于数据的质量，也适用于数据的数量和实时性。

本章中我们讨论了计算机接口，尤其是使用内部和外部总线传递信息。所有计算机，无论是占据一个房间的大型机还是嵌入在一个药片中的微型医疗诊断计算机，都要求通过总线完成通信。虽然存在大量标准总线，但还在不断涌现出更多总线（而且没有什么可以阻挡工程师设计他们自己的总线）。

本章我们讨论了与总线相关的两个话题，一是对于当今以人为中心的嵌入式系统非常重要的实时性问题；二是用于嵌入式计算器件的无线技术。

至此，我们已经完成了关于计算机体系结构的讨论。在后续章节中，我们将重点转向实际应
284 用这些技术。

思考题

6.1 一个嵌入式40MHz主频CISC CPU，最慢的指令（除法）需要100个时钟周期完成。最快的指令（分支）只需要2个时钟周期。有两个中断管脚分别对应于高优先级中断（HIQ）和低优先级中断（LIQ）。一旦中断管脚有效，需要4个时钟周期发现中断并开始进入一个到中断向量表的分支。假设其他中断不使能，且中断必须等待当前指令执行完才能够被服务。

a. 计算最坏情况下HIQ中断的响应时间，计时从中断管脚有效开始，直到转到包含在中断向量表中的ISR。

b. HIQ的ISR要求10ms完成执行（从HIQ管脚有效开始测量的最坏情况）。最坏情况下LIQ的响应时间是多少？

6.2 问题6.1中的CPU包含16个通用寄存器。描述可以在CPU设计中用来从减少上下文保存和恢复时间的角度改进ISR性能的硬件技术。

6.3 从影响中断响应时间的角度评价以下四种技术：

a. 虚拟存储器

b. 基于堆栈的处理器

c. RISC而不是CISC设计

d. 较长的CPU流水线

6.4 确定下列系统的实时性要求，确定每个实时输入输出是硬实时还是软实时：

a. 便携式MP3播放器

b. 装在家用汽车上的防抱死刹车系统

c. 火灾报警控制和显示控制台

d. 台式个人计算机

6.5 画出一个连接到100MHz处理器上的Flash存储器件的总线工作图。Flash存储器说明书给出以下信息：

- 40ns存取时间
- 20ns的保持时间
- 20ns的地址选择时间

285

6.6 一个实时嵌入式系统监控一个压力容器中的温度。如果温度超过某个值，系统必须以1Hz闪烁报警灯并且打开一个减压阀。系统每隔100ms从一个串行线上读一次温度，然后用10ms左右将从串行线上读得的数据解码成温度值。在最坏情况下，温度可以在150ms内达到引起爆炸的温度。

如果三个输入输出信号（串行温度输入，脉冲报警灯输出，减压阀控制）分别由不同的任务处理，确定每一个任务的时间范围并给出它们的硬实时程度。

6.7 考虑表6-1给出的PC104接口及其管脚定义。在一个实现了整个连接集合的嵌入式系统中，数据总线宽度是多少？当使用扩展连接器J2/P2时，系统有一条扩展地址总线。计算所允许的最大寻址空间，以MiB为单位。

6.8 在LVDS（低电压差分信号）方案中，从表示逻辑0到表示逻辑1的电压摆幅远小于其他信号格式。例如，EIA232（RS232）中逻辑0和逻辑1之间的压差为12V，而许多LVDS驱动器只输出0.25V的压差。这是否意味着在电气噪声较高的系统中EIA232是更可靠的选择？对你的回答给出说明。

6.9 将6.3.5节中关于以太网驱动器的部分与附录B中介绍的OSI层次模型关联起来描述（尽管实际上通常使用以太网的TCP/IP网络系统采用了与OSI模型稍有不同的层次架构）。

6.10 一个简单的抢占式多任务嵌入式计算机执行三个任务，T1，T2，T3，它们按顺序优先（优先级最高者先执行）。任务T1需要1ms的CPU时间，每10ms触发一次，且必须在后一次触发前完成。任务T2需要3ms的CPU时间，每9ms触发一次，且必须在被触发后8ms内完成。任务T3要求1ms的CPU时间，每6ms触发一次且必须在被触发后4ms内完成。

假设所有的任务在时间$t=0$时被触发，画出一个该系统调度图（类似于图6-5），在横轴上以ms标注时间，从时间$t=0$到$t=40$ms。从所示的时间间隔来看，判断是否所有任务能够满足它们的时限。 286

6.11 重复问题6.10。所不同的是任务采用频度单调调度。就各个任务在第一个$t=40$ms的区间内满足时限的要求而言这种改动带来什么不同吗？

6.12 一种消费电子设备需要一种小尺寸、低功耗、中等速度的CPU控制器。讨论是并行连接的数据存储器系统还是串行连接的数据存储器系统更适合。

6.13 如果变化一下问题6.12所述的系统要求，使得性能和速度比尺寸和功耗更重要，是否会影响你对总线的选择？

6.14 如图6-9所示为Atmel AT29LV512 Flash存储器（闪存）设备时序图。其中的时间参数来源于Atmel器件手册。

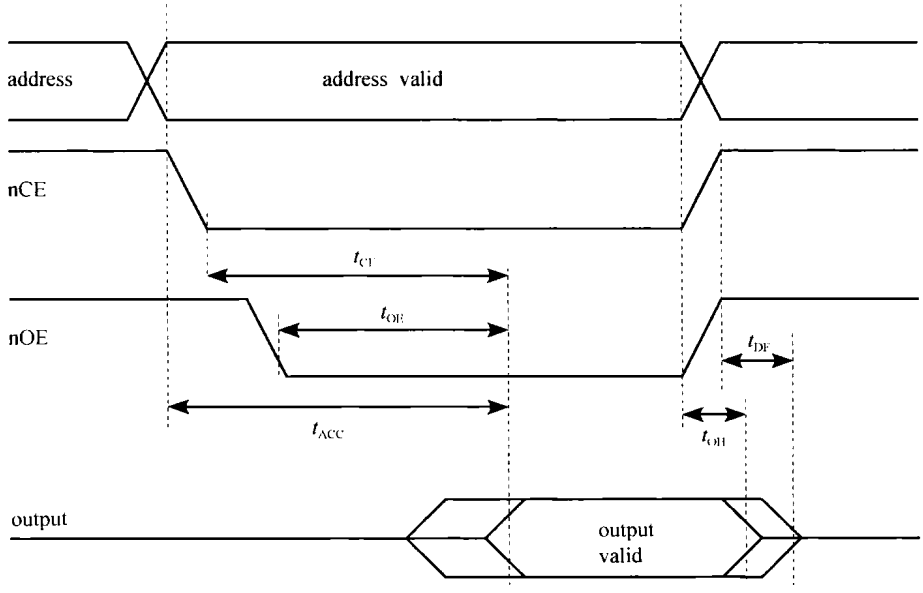


图6-9 Atmel AT29LV512闪存设备的读周期（波形根据Atmel AT29LV512器件手册绘制） 287

参数	含义	Minimum	Maximum
t_{ACC}	访问时间（地址有效到输出延迟）	—	120ns
t_{CE}	nCE到输出延迟	—	120ns
t_{OE}	nOE到输出延迟	0ns	50ns
t_{DF}	nCE或nOE①撤销到输出Hi-Z	0ns	30ns
t_{OH}	输出保持地址，nCE或者nOE②	0ns	—

①从其中任何一个被设置为无效时开始。
②当其中任何一个被设置为无效或变化时开始。

假设若某个值没有给定，则该值不重要。该时序图是从某个外部设备（如 CPU）读闪存的角度给出的。它给出了 CPU 为正确地读闪存所必须遵循的时序。

本题的问题是，确定如何设置 S3C2410 并行接口时序寄存器以保证它能够正确存取并行连接其上的 Atmel AT29LV512 器件。这将需要仔细阅读 6.2 节（以及框 6.2）。注意 HCLK 信号（即总线时钟）的工作频率为 100MHz，而 Atmel 芯片使能信号 nCE 连接在 S3C2410 的 nGCS 信号线上。

下表给出了需要确定的设置（注意，此例中我们忽略了页模式存取周期）。

信号	含义	No. of cycles
Tacs	地址建立到 nGCS 有效（0，1，2 或 4 个周期）	
Tcos	芯片选择设置时间到 nOE（0，1，2 或 4 个周期）	
Tacc	访问周期（1，2，3，4，6，8，10 或 14 个周期）	
Tcoh	nOE 无效后片选的保持时间（0，1，2 或 4 个周期）	
Tcah	nGCS 无效后地址的保持时间（0，1，2 或 4 个周期）	

6.15 确定问题 6.14 中读单个字的操作在最坏情况下的时间，同时对一个更为现代的闪存重复该计算，该存储器的存取时间为 55ns，t_{CE} = 55ns。

288 **6.16** Atmel AT25DF041A 是一个 4Mbit 的串行闪存，使用 SPI 接口，最高工作频率为 70MHz。从选定的 AT25DF 器件上读一个字节需要作为控制器的 CPU 首先输出一个读命令（即字节 0x0B），紧接着是一个 24bit 地址，然后跟着一个空字节。这些域均按照时钟串行输出，串行输出管脚的最高频率为 70MHz。假设 CPU 不停止 SPI 时钟，该器件将在随后的 8 个周期里顺序输出存储在该地址上的字节给 CPU。

确定一个读字节的操作总共需要多少时钟周期，并确定从该器件上读一个字节的最小时间长度。简单计算一下问题 6.14 中 AT29LV512 器件上的读操作要快多少倍？

注意：值得提醒的是这样的比较不是很公平。首先，当读一连串存储单元时两个器件都会更有效，SPI 器件尤其如此。其次，SPI 器件还有更为快速的读命令，而我们没有用到，该命令为 0x03 而不是 0x0B，0x03 命令不需要在地址最后一位输出之后插入空字节，不过该模式只适用于最高工作频率为 33MHz 的情况。

6.17 将下列应用（a～e）与适当的总线技术相匹配，主要考虑带宽、延时、功耗、内外计算机通信、连线数、抗噪声干扰、距离等。

a. 一个禁止用户开关滑动窗户的器件连接到一个嵌入式计算机上，且在窗户被打开时有一个 LED 报警。

b. 嵌入式计算机内置有图形输出器件，它完成从 CPU 以 1.8Gbit/s 的速率输出视频数据。

c. 工业自动化计算机需要将位于 500m 外的传感器穿过充满噪声的电气工厂连接到计算机（此处由于干扰无线设备不能工作）。传感器以每秒几十 Kbit 的速率返回温度数据。

d. 一个 FPGA 协处理器内置在 x86 处理器系统中以尽可能快的速度传输大量数据。

e. 一台小尺寸嵌入式工业 PC 需要一个连接 20 个模数转换器（ADC）的外设卡，总的数据速率为 6MiB/s。

针对这五个应用，有五个总线技术供选择，一个应用对应一个技术。

- AGP 4x

289

- USB 1.1
- PC/104（16 位 ISA）
- 16x PCIe（16 通道 PCI 增强版）
- EIA422

6.18 哪五个时间参数能够描述实时系统中一个任务的时序特征？

6.19 给出当在嵌入式 CPU 中断发生时一般的操作序列。

6.20 描述像 ARM 这样的只有单个通用中断信号（IRQ）的处理器若要实现中断共享所必需的硬件。注意

290 这可能增加中断服务例程的额外开销。

实用嵌入式 CPU

7.1 概述

几十年来计算机体系结构一直作为一门学术研究学科，教育了几代工程专业学生，计算机体系结构随着硬件发展而更新，但它作为课程教授给学生时往往落后十几年。在大型计算机时代课程滞后十几年并没有什么，而在个人计算机时代如此滞后的时间就带来了一些问题。

我还记得在学校时学习的是8086、6502和Z80，而当时我已使用第一代ARM台式机（在那个年代它是非常快的）。奇怪的是，这种课程滞后很多年来依然如故，这表现为准备到嵌入式系统或消费电子界工作的学生，所接受的教育却是更适合大型计算机领域的。

本书的目的有所不同——对大型机相关技术只是简要概述，而对嵌入式系统工程师感兴趣的技术进行深入介绍。把重点放在实用性上，鼓励读者将学到的知识应用于实际。

截至目前，本书所讲内容还属于基础和理论方面的。而从本章开始，我们进入到实践部分，开始探索嵌入式系统的现实世界。我们分析为了使现实中的嵌入式计算机工作需要做什么，因此要跨越嵌入式计算机体系结构的理论和现实之间的鸿沟。

7.2 微处理器不只是核

一种在本书写作时期（大约5年前）最为流行的微处理器是我们前面提到过的三星生产的基于ARM的S3C2410。让我们来看一下这个小小的器件，了解一下它的特性： 291

- 1.8V/2.0V的ARM9处理器核，最高工作频率为200MHz
- 16KiB指令缓存和16KiB数据缓存
- 内部MMU（内存管理单元）
- 用于外部SDRAM（同步动态随机存取存储器）的存储控制器
- 彩色LCD（液晶显示）控制器
- 带有外部请求管脚的4路DMA（直接存储器存取）机制
- 3路UART（通用异步收发器），支持IrDA1.0，16字节发送缓存和16字节接收缓存
- 2路SPI（串行外部设备接口）
- 1路多主IIC（内置集成电路）总线驱动器和控制器
- SD（安全数位卡）和MMC（多媒体卡）接口
- 双端口USB（通用串行总线）主机加上单端口USB器件（1.1版本）
- 4通道PWM（脉冲宽度调制）时钟
- 内部时钟
- 看门狗定时器
- 117位通用I/O（输入/输出）端口
- 24路外部中断源
- 功耗控制，包括常态、慢速态、空闲态和电源关闭态
- 8路10位ADC（模数转换）和触摸屏接口
- 实时钟表，带有日历功能
- 片上时钟产生器

S3C2410是一款功能很丰富的器件，适用于嵌入式系统，因此至今为止一直被工业界开发者使用。正如我们在6.1节所见，这种芯片有时称为片上系统（SoC）[⊖]处理器，其特征是包含许多外设单元。处于系统中心的核是ARM处理器，其他ARM9系统也是如此。

虽然三星没有给出有关S3C2410中各个单元的尺寸以及它们在芯片中的布局等细节，我们还是可以估计出其中缓存所占的芯片面积最大。占据芯片面积仅次于缓存的是中央处理单元（CPU）核。其他较大的单元是MMU、SDRAM存储控制器和ADC。

[292]

在早期集成电路中，CPU是一块单独的芯片，集成了许多以前离散分布的单元。随着时间的推移，越来越多的功能被集成到这样的器件中。对于嵌入式系统而言，半导体厂商已经意识到设计者喜欢尽可能少地使用单独的器件，因此片上系统提供了许多功能。在任何一个嵌入式系统设计中，并不会用到所有的功能，但至少会需要其中一部分。使用这种高集成度的片上系统意味着具有以下设计特点：

1. 降低芯片门数就会降低面积，因此通常也会降低产品成本。

2. 当挑选一款SoC时，设计者首先要列出需要的功能，然后挑选与此尽量匹配的器件。没有包含在该芯片中的功能可以通过外接方法实现。

3. 一些硬件功能可以通过软件有效实现，这时设计的问题是“如何使用片上外设”，而不是“如何用硬件实现这个功能”。

4. 有时，有限的片上功能可能会制约产品的功能，而改变外部实现的功能要比改变片内实现来得简单。

5. 设计者现在不得不通读超过1000页的CPU数据手册，而关键细节很有可能藏在第991页的脚注中。

6. 一些功能不能共存。例如，功能表中说既提供了IIC也提供了UART支持，但忘记说明在一个时刻只能支持其中一种功能，原因可能是没有足够的管脚，或没有足够的内部中行使性支持。

主流处理器倾向于将较多的建面积用于高速缓存而不是CPU常规功能，因为高速缓存被看做是改进处理器性能的最好方法。以64位VIA Isaiah体系结构（又叫做VIA Nano）为例，这是一种近来出现的x86兼容处理器，如图7-1所示，其中最大部分建面积用于高速缓存。其中还包括一

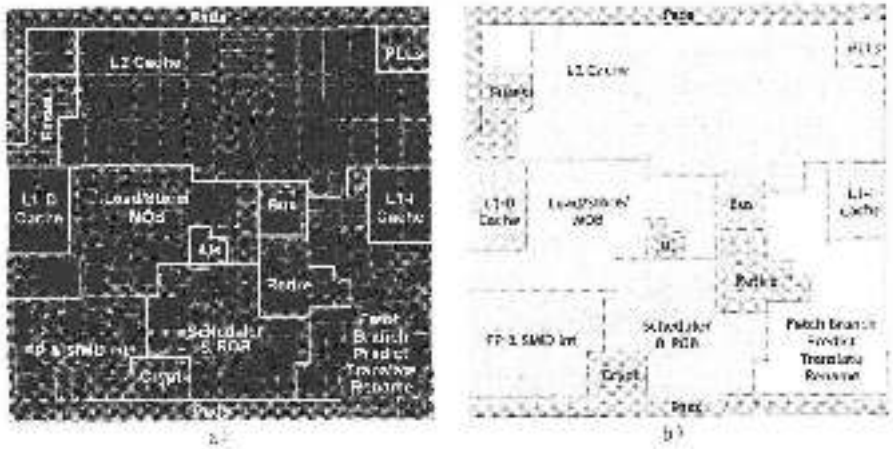


图7-1 VIA Isaiah体系结构，一个低功耗x86型CPU，特别适合笔记本计算机之类的移动计算应用，图中给出了各器件在硅片内部的安排（本照片和结构图由VIA公司提供）。a）芯片照片，显示了功能区域块；b）芯片面积对应的功能区域方框图

⊖ 较小的SoC系统有时也称为单片微处理器或单片微控制器。

些独立模块，分别用于时钟生成（锁相环，PLL）、高速浮点（FP）、SIMD体系结构（该器件支持在4.7.4节讨论过的SSE-3扩展指令，因此它与浮点运算单元FPU并行排列），其他有趣的模块还包括用于加密处理的模块，用于乱序执行的重排序缓冲器（ROB）、在据说长度为十几级的流水线末端的扩展分支预测与硬件回退（retirement hardware）。还包括2个64位整数单元（IL）和3个缺数－存数单元以及存储重排序缓冲器（MOB）。顶部和底部Pad用于连接内部电路和集成电路封装焊点。这个采用65nm工艺实现的器件有64KiB一级缓存和1MiB二级缓存，使用了大约9400万个晶体管。对比一下台式计算机/服务器中的CPU，AMD的四核Phenom有4.5亿个晶体管，它还包括2MiB三级缓存，如图7-2所示。 293

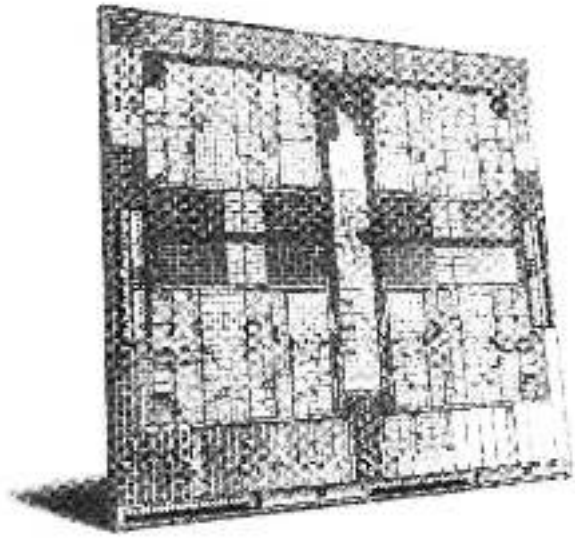


图7-2 AMD Phenom™四核处理器芯片。其中对称的水平和垂直线将芯片分成4个核，器件顶部和底部的非对称条是双速率随机存取存储器接口和2MiB共享三级缓存。中间垂直方向的长方形是用于连接4个核的总线桥接系统，而左右两侧的是物理接口（本照片由AMD公司提供）

7.3 功能需求

在许多系统中，有些功能属于有则更好，而有些是必需的。确定一个功能在SoC中属于哪一类取决于它所面向的应用领域。例如，一个系统可能要求串口，而另一个系统则要求SPI。

正因为如此，SoC厂家在定义什么是必需的功能这一点上不能达成一致意见，而由此产生的产品变化对于我们搜寻适合嵌入到自己的设计中的器件是一件好事。事实上，这种情形也是消费者驱动的：一个能够卖出几百万产品的公司有能力说服半导体厂家设计他们所需的芯片，而小的独立设计者却很难要求半导体厂家为他们增加特定外设。

然而，有一两种外设被认为是必需的，且在几乎所有的主流SoC处理器中都可以见到。

1. 复位电路（见7.11.1节）是必需的，用来确保每一个器件启动后寄存器和状态处于预定值。

2. 时钟电路是必需的，用以将全局时钟分配到一个同步设计的各个部分。通常一个锁相环（Phase-Locked Loop，PLL）或延迟锁定环（Delay-Locked Loop，DLL）用于调整外部晶振产生的 294
振荡以及调节频率。

3. I/O驱动器连接外部管脚，提供足够的电流在连接外部器件的线上产生电压翻转，同时协助防止片外微器件对IC内部电路产生静态充电、短路和电压尖峰。许多器件带有通用I/O（GPIO），它在方向、驱动特性、缺值等方面是可编程的，详见图7.1中的讨论。

4. 总线连接器也是与外部连接的接口，主要用于连接外部存储器、外设等。它们通常以一

组I/O驱动器协同工作的方式实现。

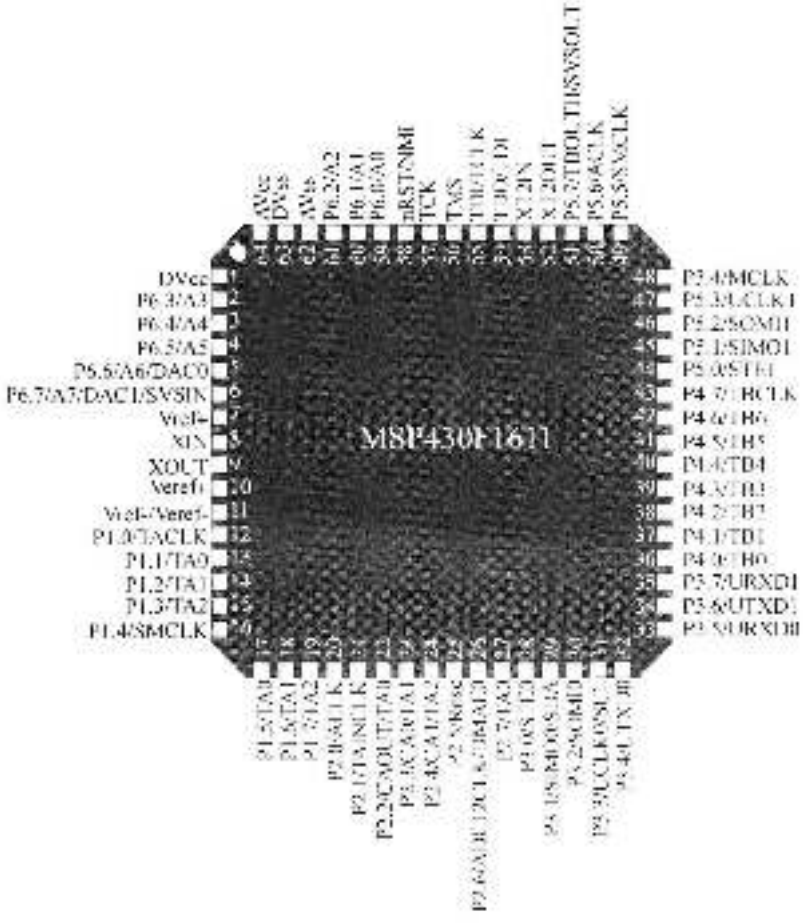
5. 存储器本身既可以处于片上也可以处于片外，还常组合使用存储变量和堆栈的易失存储器和存储程序代码的非易失存储器。

6. 功耗管理电路用来将电源分派到各个器件，关闭芯片上不工作的部分等。

[295] 7. 调试电路，例如IEEE1149 JTAG，在多数情况下被认为是非常必需的（见7.9.3节）。

框7.1 MSP430可配置管脚

我们十分多为嵌入式系统设计的处理器，TI的MSP430系列具有很高的I/O管脚配置能力。下面以该系列中MSP430F1611为例来考察管脚的可配置性。



这64个脚封装件上除了电源、地、参考电压输入、外部晶体连接和两个JTAG管脚外，64个管脚中的51个是双向配置的，每一个使用多个功能。例如，管脚5可为GPIO 6位端口6（P6.6），同时为12位ADC的输入信道6或12位DAC的输出信道0，取决于最终使用者在软件中的设置。

[296] 在框7.2中我们将加以配置管脚。

框7.2 MSP430管脚控制

在7.1节讨了TI MSP430F1611芯片的管脚布局，此处可见单个管脚可以具有多种配置。事实上，这些管脚配置是在软件中完成的。下面我们来看一下该机制是如何工作的。

MSP430有若干管脚控制寄存器，管脚8位一组，例如P1.0～P1.7构成端口1，P2.0～P2.7构成端口2，以此类推。每个端口的8个I/O管脚可以分别配置方向，即作输入或输出。在许多情况下，它们还可以用软件读、写。让我们来看一下对应端口2的寄存器。

寄存器P2DIR是一个8位方向寄存器，为存储中的一位控制着对应的管脚为输入或输出管脚。若0写

入某一位导致其对应管脚为输入管脚，而将1写入某一位导致其对应管脚为输出管脚。例如将0x83写入寄存器使得P2.7、P2.1和P2.0为输出管脚而其他为输入管脚。

寄存器P2IN是一个8位寄存器，每一位代表对应管脚上的输入值。因此，如果读该寄存器的返回值为0x09，则意味着P2.3和P2.0上的电压为高，其余管脚电压为低。注意，如果我们已将P2.0配置为输出管脚、P2.3配置为输入管脚，则我们可知P2.0上输出逻辑高，且其他器件输入逻辑高电压给P2.3。

寄存器P2OUT是另一个8位寄存器，用以确定每一个配置为输出的管脚的输出电压。配置为输入的管脚忽略写入该寄存器的值。

还剩下一项需要确定的配置，即是将这些管脚作为GPIO还是将它们用做其他指定功能。为此，寄存器P2SEL被用来选通GPIO端口寄存器和外设模块。当逻辑低写入P2SEL时意味着管脚连接到GPIO寄存器，而逻辑高意味着选择外设功能到管脚。例如将0x81写入P2SEL将实现下列选通：

管脚	20	21	22	23	24	25	26	27
功能	ACLK	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6	TA0

有两件事需要注意。第一，外设确切的功能由外设模块决定，其具体配置方法应在该设备手册中说明。一些管脚有三种定义，其一通常是GPIO端口，另外两种是外设模块（管脚选通逻辑不能完成对这两种外设模块之一的选择，需要借助于外设模块本身的配置）。

第二，若管脚被配置成与外设连接，则管脚方向必须通过写P2DIR正确设置。一些处理器可以自动完成这种相关设置，但MSP430必须通过程序来实现。例如，如果通过写P2SEL将某一管脚定义为串口输出，则P2DIR的相应位就应当为逻辑1，否则，将不会有输出产生。

多数器件还包含一个或多个内部UART（通用异步收发器）或USART（通用同步/异步收发器），一个内部实时时钟模块（RTC），几个时间计数器，以及内部高速缓存等。

比较面向不同市场设计的CPU是非常有趣的事，如表7-1所示，所列三个广泛用于嵌入式系统的器件分别属于三类处理器。单芯片微处理器，TI的MSP430F1612，功耗非常低（在最低用电模式下芯片可以基于两个柠檬产生的电力工作），且内嵌了多种低层次外设。其设计目标是使选择它的设计者可以获得单芯片解决方案，因此它没有外部存储器接口。与此相反，三星S3C2410是一种基于ARM9的片上系统，具有丰富的功能，足以支持个人数字助手（PDA）、智能手机等应用。它不仅有SDRAM接口，在并行总线（参见6.2节）上可连接扩展静态随机存取存储器（SRAM）、只读存储器（ROM）和Flash，还提供了多种外设接口——尤其是通信和内部连接外设。最后一个是VIA Nano，我们在7.2节中介绍过。某种意义上，尽管它为提高功耗效率做了重新设计，且比典型的个人计算机处理器尺寸小，但它还是一种标准个人计算机处理器。对于要求x86结构处理器的嵌入式系统，它可作为一个候选。该器件主要关注于计算能力，强调低功耗约束下的高性能。VIA Nano有很多另外两款处理器没有的外设，当然，那两款处理器也可以通过附加芯片来提供这些外设。

表7-1 从三类微处理器中选择示例以比较其内置特性：单芯片微控制器、片上系统微处理器和个人计算机CPU。其中，TI的MSP430系列在写这本书的时候已有171个变种，每个都有特殊的性质和能力——系列中器件最高频率可达25MHz，包含16KiB RAM和256KiB Flash，以及十分多样化的外设配置。相反，三星和VIA都只有很少的变种器件

	单芯片微处理器 TI MSP430F1612	SoC CPU 三星S3C2410	个人计算机CPU VIA Nano
时钟速度	8MHz	266MHz	1.8GHz
功耗	<1mW	330mW	5~25W
封装	64管脚LQEN/P	272管脚FGBA	479管脚BGA
内部缓存	无	16KiB I+16KiB D	128KiB L1+1MiB L2

（续）

	单芯片微处理器 TI MSP430F1612	SoC CPU 三星 S3C2410	个人计算机 CPU VIA Nano
内部 RAM	5KiB	无	无
内部 Flash	55KiB	无	无
内部带宽	16 位	32 位	64 位
外部数据总线	无	32 位	64 位
外部地址总线	无	27 位	未知
存储支持	无	ROM 到 SDRAM	DDR-2 RAM
ALU	1	1	2
FPU	不	不	是
SIMD	不	不	SSE-3
乘法器	16 位	32 位	最高 128 位
ADC	12 位	8×10 位	无
DAC	2×12 位	无	无
RTC	不	是	不
PWM	不	4	不
GPIO	48 管脚	117 管脚	无
USART	2	3	不
I^2C	是	是	不
SPI	2	2	不
USB	不	2 主 1 设备	不
看门狗定时器	是	是	不
掉电检测	是	不	不
时钟	2	1	是
JTAG	是	是	未知

297
~
299

下面我们将比较详细地讨论一些 CPU 必须具备的功能，包括时钟、功耗管理和存储器。之后在 7.11 节，我们将了解一下器件复位机制，特别是看门狗定时器、复位监测器和掉电检测器。

7.4 时钟

在 3.2.4 节中我们讨论 CPU 控制时考虑了系统时钟在控制微操作中的重要作用。事实上对于时钟重要性我们强调得还不够。除了非常少见的异步处理器（我们将在 9.4 节中介绍）外，所有处理器、大多数外设、总线和存储设备都是依赖时钟同步信号正确地完成操作。

CPU 功能模块仅包括组合逻辑，例如算术逻辑单元，在它周围时钟尤其重要。如果时钟沿控制 ALU 的输入，则同一个时钟沿不能够用来获取 ALU 的输出，因为 ALU 需要一定的时间完成其功能。必须使用后一个时钟沿，或采用两相位时钟（即两个非对称时钟，它们不相互重叠，且两个沿之间间隔大于时钟系统中的最大组合逻辑延时）。

实际当中通常采用一个时钟，但在时钟波形的不同沿上完成不同功能。图 7-3 给出了一个例子，其中 ALU 工作在不同的时钟沿上。从第一个下降沿开始，首先（ⅰ）驱动来自 R0 的单总线，在第一个上升沿完成（ⅱ）将输入值锁存到第一个 ALU 寄存器且释放总线驱动。之后，（ⅲ）和（ⅳ）重复该过程将 R1 写入 ALU 的第二个寄存器。（ⅴ）接收到稳定输入后 ALU 需要

一定时间完成计算。（vi）之后将结果写入 R0。

图 7-3 在图的底部给出主时钟信号，频率为 $F_{clk}=1/T_{clk}$。时钟上升沿或下降沿有效时刻是时钟信号通过阈值电压的时候（如图中虚线所示）。注意时钟信号沿不是完全垂直的，即有时钟上升时间和下降时间。事实上，在每个周期时钟通过阈值的时刻会有一点差异，其原因来自电噪声、电路电容、电感、温度等影响。该偏差称为抖动（jitter）。

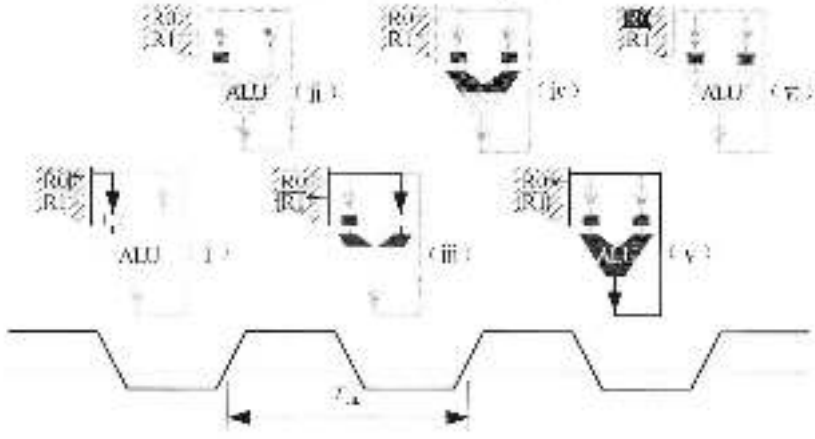


图 7-3　不同的门和锁存器驱动 ALU 与 CPU 时钟单沿同步，类似于图 3-3 的周期级时序图，所完成的操作为 R0 = R0 + R1，分为 6 个顺序阶段完成

抖动也会因为阈值电压的变化而产生（通常阈值电压保持不变，但时钟电压会随时间漫慢变化）。反过来，抖动会导致每个周期的 T_{clk} 值不相等。显然，必须选择能够满足信号有足够时间通过 ALU 的时钟频率，而抖动偶尔会引起时钟周期变短，从而导致 ALU 结果不能按时准备好，以致发生错误行为。

因而，时钟集成十分重要，巨大多数系统时钟低于可以达到的最快时钟周期。这意味着如果有非常稳定的时钟和供电电压，系统实际可操作在快于它们标定的时钟频率上（这也是多年来个人计算机中普遍使用 CPU 超频率工作的原因）。

时钟生成

近来，多数 CPU 和 SoC 中的处理器在外部连接的时钟晶振的基础上产生内部时钟频率，最多仅需要两个很小的外部耦合电容。

为了产生内部时钟，这些当代流行的器件中必须包含锁相环电路以根据原外部时钟产生内部时钟。通常还包括内部分频器和乘法器硬件。例如，三星 S3C2410 采用 12MHz 外部时钟产生 266MHz 内部时钟（事实上，时钟分频寄存器允许在一个特定外部晶振的基础上产生多个操作频率）。

延迟锁定环技术与此类似，只是灵活性较小且精度较低，但制造简单、成本低廉。注意当频率适当时，也可以将外部晶振信号直接输入到 CPU 中使用。

当前一些系统要求实时时钟，一般是通过单独的 32.768kHz 外部晶振（以及独立的 PLL）提供的。32.768kHz 晶振器件很便宜，而且很小。该晶振常用于钟表，因为该信号经分频 2^{15} 次后可以产生 1 秒的定时脉冲，用来驱动钟表和日历电路（记作 1pps 或每秒一个脉冲）。

虽然可以使用很多精准晶振，例如用于射频（RF）电路的恒温晶振（OCXO），但大多数微处理器使用标准石英晶振或陶瓷谐振器。它们的准确率大约 100ppm（每百万中的份数），等价

于0.0001%，相当于最坏情况下每年小于1小时的误差。稍微贵些的晶振很容易就可以达到
301 10ppm精度，而OCXO可以达到的精度在1ppm分之几的范围。

7.5 时钟与功耗

当阅读CPU数据手册时，经常可以发现时钟和功耗控制在同一章介绍的情况，很多情况下，系统控制寄存器也在其中。这样做的原因在于时钟是导致CPU内部功率消耗最直接的因素。

让我们花点时间来回顾一下现代CMOS（Complementary Metal Oxide Semiconductor，互补金属氧化物半导体）中的功耗原理。我们暂时不深入研究半导体理论，先考虑一个简单的门，比如说如图7-4所示的与非门（NAND）结构。CMOS这一名称中的“互补”含义就是因为该结构的输出可以通过晶体管连接到Vss（负极电压）上，也可以连接到Vdd（电源电压）上。

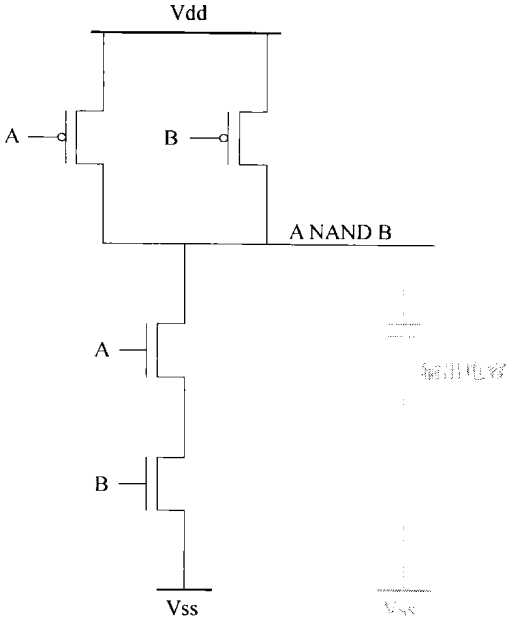


图7-4 利用CMOS门来设计一个与非电路，可以看到MOS晶体管直接与源电压和漏电压相连。灰色的输出电容器用来反映NAND输出的电容负载

在理想情况下，CMOS系统连接到Vss或者Vdd上的输出是没有电阻的，但是我们知道在现实世界中0电阻是不存在的，导线上存在电阻，或者存在漏源电阻等。这些电阻所导致的结果就是限制了从Vss或者Vdd中流出或流入的电流，因而也要花费些时间来充满输出电容。当门从一个状态跳到另外一个状态，电流就会被激发，也就是要充满输出电容或者释放输出电容。随着充放电流的改变，电容电压也随着升高或者降低。这可以从图7-5中很清晰地看到，其中CMOS门用一个理想开关来替代。
302 最重要的是图底部的逻辑值输出：在一个数字电路中，从一个事件发生（比如说开关位置改变）到输出逻辑值稳定的时间，叫做传输延迟，这个概念我们曾在2.4.2节讨论进位延迟加法器时讨论过。

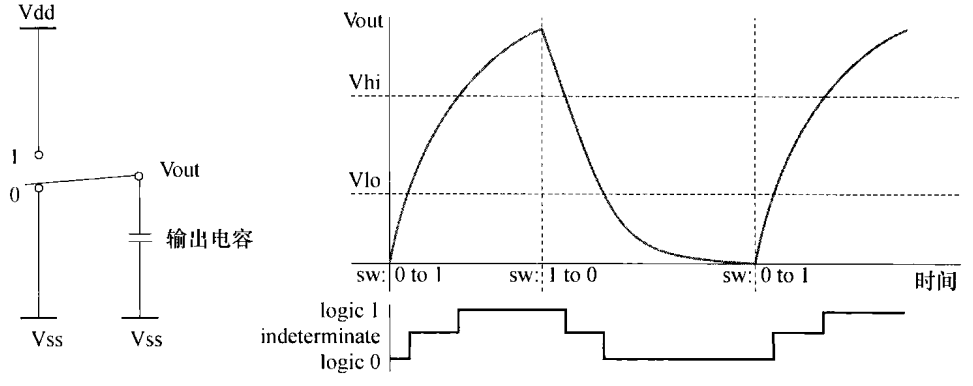


图7-5 左边所示为电容里的切换电压，它需要一定的时间来充电和放电，如图中曲线所示，电容电压会随着开关位置的改变而发生变化。注意逻辑电压的Vlo和Vhi阈值，图中的底部折线显示了电容电压的逻辑值随时间的变化

事实情况远比我们所说的复杂，在所有的硅门中均有寄生电容存在，在各种导线及门连接中也有寄生电阻存在，甚至存在寄生电感。这样，就使得我们前面所讨论的负载电容问题更为严重。

理解了系统中电容的基本问题后，我们要注意到两个重要的概念：

- 传输延迟，是指通过小电阻的导线和导电硅片对电容进行充放电时所花费的时间。
- 门切换所产生的电流，是由于在充放电时必须有电流经过电容器。

7.5.1 传输延迟

为了降低传输延迟，硅片设计者可以使用多种方法：可以降低电容（通过设计更小的门，因为电容与硅片上门结构的面积成正比）；可以降低阈值电压，这样可以让电压更快地达到阈值，从而提供更多的电流让电容充放电更加迅速。硅片上门的尺寸正在逐年递减，也许很快就会接近其物理极限，但更小的器件尺寸意味着更高的电阻，进而限制了电流，所以要更换材料以设 303
计更低电阻的半导体。降低阈值电压已经被目前的IC厂商所采用，从5V到3.3V、1.8V、1.2V甚至更小。但是，降低阈值会使得硅片更容易受到电子噪声的干扰。

从根本上说，IC设计者会通过各种方法来改进他们的电路系统，比如通过各种可行的办法来仔细平衡取舍从而降低传输延迟。从20世纪50年代到2007年，通过采取这些设计方法，芯片时钟频率逐年增高。然而，提升频率所带来的困难已经迫使大家的重心从高频率转向高并行化，也就是说，如果你的频率不够快，那么可以尝试更多的并行来达到高频率的效果（5.8.2节）。

许多用于提升处理器性能的技术已经使其芯片内部的电流升高（将在7.5.2节讨论），并且这一现象在芯片特征尺寸逐渐减小的面积有限的硅片上更为明显。由于电流是通过寄生电阻传输，这样会在传输过程中产生热能。所产生的电阻热能耗散和电流的平方乘以电阻的结果成正比（由于电流和电压成正比，因此这也是为什么厂商热衷于降低供电电压的原因，这样可以降低电流，因而可以降低功耗）。不幸的是，电阻与面积成反比，由于特征尺寸降低导致面积减小，所以电阻也会随之升高。这是一个在面积设计上进行折中的重要参考。

总的来说，电阻功耗在增加，另外随着时钟频率提高，门的开关更加频繁，因而带来的功耗也更多，最终会导致更大的功耗损失。这也意味着用于门开关而引起热量耗散的时间变短，致使硅片的温度自然升高。对于实现CPU的硅来讲，125℃甚至更高的温度是很常见的。

在额定电压下，更小的特征尺寸会产生出更多的热量，意味着减小IC封装的尺寸会让散热更加困难。这样，风扇、散热器、导热管等对于CPU来讲就成为必需的设备。

下面，我们不考虑风扇和散热器，我们将关注一些在计算机中减小能量消耗的方法，特别是对于电池能量供给有限的嵌入式系统。

7.5.2 电流相关问题

在CMOS门电路中电阻会消耗一些能量（即便是门处于空闲状态，也会有微弱的电流经过电阻并消耗能量），但是这与门开关过程中所消耗的能量相比相形见绌。

驱动单个MOS晶体管开关的瞬时电流通常是由供电电路通过在印制电路板（PCB）上的一个电源层或者电源线来提供。切换至地面（0电压）的电流通常为PCB上的GND层所吸收。不幸的是，电源线、电源层和GND层均有很小的电阻。当一个由门开关所引起的很短但比较大的电流脉冲经过这些电阻时，就会产生一个补偿压降。 304

实际中，有几十万个门，而且是在同一时刻进行开关转换，所有瞬时电流会累加在一起。一个好的示波器，只需要把示波器接到设备的电源输出接口和接入引脚上，就可以很容易地直接

检测到发生在一个系统时钟过程中的这个压降。好的电路设计实践是：在芯片的电源和地线引脚旁边设计一个旁路电容。这种设计的作用是耦合电源的高频噪声。这种设计还可以用做电源储蓄库，在需要和系统时钟同步时释放一个很短的电流脉冲。

门开关所产生的电流可以非常大，对于一个x86结构的设备来说，甚至达到几百安培，但是仅会持续几个纳秒。另外一个问题就是由此所引发的电磁干扰（EMI，6.6.3节中有简单介绍），任何时候只要有电子移动，便会有由此引发的相应的移动电场，而且事实上，包含电流脉冲的电路也会像一个天线一样同步地辐射噪声，或者接收噪声。

7.5.3 时钟问题解决方法

除非降低开关频率、系统电压或者改变门的设计，否则电流问题是不会从根本上改变的，尽管像旁路电容和储蓄电容这种技术可以减轻这种问题。

然而，我们可以考虑一些方法来解决时钟所引起的EMI问题。第一种方法就是引入多个时钟，每个时钟之间有轻微的相位差。如果有四个不同相位的时钟，并且将一个电路分为相应的四个部分，通过这四个不同时钟进行控制，这样电路的峰值电流会降到原来的四分之一。

维持一个稳定的电流可以极大减小电磁干扰，因为电磁辐射跟电压变化有很大关系，也就是说我们如果采用直流电源，那么所有的EMI问题都能解决。

另外一种方法就是扩频时钟。本质上讲，这种方法是通过利用一些离散的步骤来周期地或者随机地改变时钟频率，这样能量辐射可以分散到几个不同的时钟频段上。还可以故意引入一些振动的波形信号，来防止时钟上升沿或下降沿过于齐整。

由电源或者信号线所产生的EMI还可以通过并行运行的一个等量但电流方向相反的信号线抵消。这被称为均衡电路，并且通常用于LVDS中以降低EMI。

7.5.4 低电压设计

如果一个CPU的能量消耗主要和其时钟频率有关，那么一个降低功耗的很好的办法就是让时钟频率慢下来。在嵌入式系统中，这种方法是通过对时钟缩放寄存器写入控制信息来实现的，
305 这种寄存器可以在很多微控制器和SoC处理器中见到。在某些的时间段内，处理器可能会满负荷运行，而在其他时间段可能会处于空闲状态。峰值CPU时钟速度应该与处理器的峰值工作负载相匹配，不需要在所有的时刻都以峰值速度运行。

在实时系统中，一个简单的控制时钟缩放的方法就是在众多的运行任务中指定一个特殊的、运行在后台的任务，该任务拥有最低的优先级。在后台运行该任务会在一个确定的时间段内检测到它自己占用了多少CPU时间，如果占用时间超过某个时间段阈值，说明系统在大部分时间内处于空闲状态，因而可以降低时钟频率。而当该后台任务占用的CPU时间近乎为0时，则说明系统在满负荷运行，这样时钟频率应该升高。

大部分的主流CPU生产厂商，甚至x86级处理器设计者目前也是采用这种办法，这样可以延长笔记本计算机的电池寿命。

另外一种减少功耗开销的办法更加简单，关掉没有在使用的部分。令人惊奇的是，这个想法并没有立即被IC设计者所接受，但是目前大部分嵌入式处理器设计中包含功耗控制寄存器，这样可以用来关掉空闲的电路模块，从而降低功耗。当真正应用这个技术时，大部分程序员可以简单地在程序初始化阶段启用某些模块或者禁止某些模块。不过，更为常见的做法是在运行时动态地控制这些模块。

306 图7-6解释了以上两种方法，其中一个SoC处理器的电流消耗被描绘为一个程序执行过程，

该程序使用了一些片上外设。在程序执行的开始阶段，静态功耗控制通过关掉不使用的外设来减少电流消耗；而动态功耗控制首先关掉所有的外设，当外设被调用时才开启，且只在使用阶段处于开启状态。在这几种情况中，图下面的面积代表了总共消耗的能量，如果这个系统是在电池供电下工作，那么可以反映出电池电量在三种不同情形下的消耗。

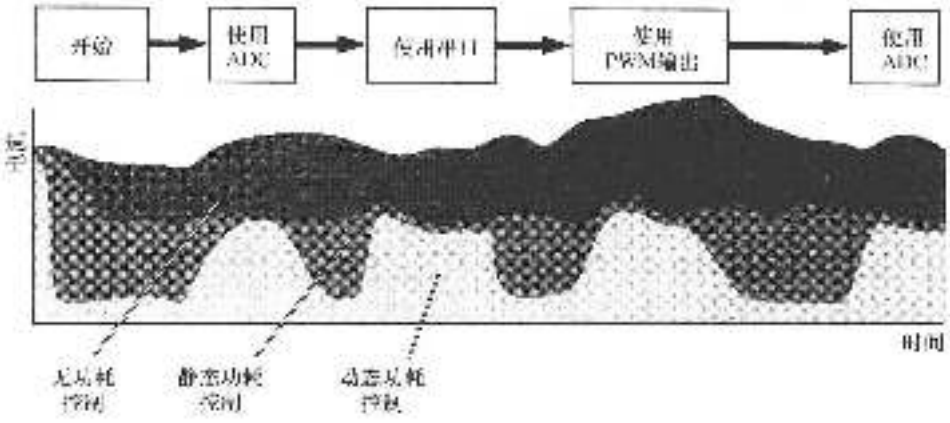


图7-6 CPU内的功耗控制示意图：一个简单的程序依次操作几个外设（分别为ADC、串口、PWM，然后是ADC），并且外设所消耗的电流量也被记录下来。其中显示了三种情况：无功耗控制，静态功耗控制（在开始阶段，所有其他未在使用的外设都被关闭）和动态功耗控制（除了在开始阶段就从关闭外，所有设备只在被使用时才开启）。三种不同颜色的图形分别表示了三种控制模式下的能量消耗

在嵌入式系统中，还有很多其他的有用方法来控制功耗，比如以下这些：

- 没有必要让LED一直亮着，因为人眼在其关闭后依旧可以看见残存的固体光，只需每隔50ms打开1ms即可（只会消耗功率的1/50）。
- 使用时钟缩放和智能动态功耗控制的组合机制来获取最低功耗开销。
- 当等待一个软件事件时，可以尝试寻找一个休眠方法，这样可以让处理器进入一个非常低功耗的模式，而不是进入一个单等待循环状态，在这种状态下处理器会进行反复的轮询。
- 即使轮询有必要时，也可以考虑在可能的时候插入一个短暂的睡眠（可以通过时钟中断来退出），此时CPU处于空闲状态。
- 定点计算往往比浮点计算更加低功耗。
- 片上存储往往比片外存储更加低功耗，因此，可能的话尽量使用片上存储来存放经常存取的变量。
- 数据移动也会产生功耗，因此，最好最大化在数据结构上的操作，通过将数据引用给操作函数，而不是把整个数据复制过去，以此减少数据移动所产生的功耗。
- 将使用高功耗器件的操作集中在一起。比如，在早期的iPod中，磁盘驱动消耗了大部分的电池电量，所以苹果公司设计了一个更大的缓存系统，使得系统可以从硬盘中一次读入一首或者更多歌曲，然后在播放这些歌曲的时候关闭硬盘。然后，几分钟过后，硬盘重新开启为读取下面几首歌做准备。这样，只需要对硬盘间歇供电。

7.6 存储

我们已经在之前的章节里对存储进行过多次讨论，并对其中一些存储器件进行了介绍，如SDRAM、DDR（双倍速率动态随机存取存储器）等。现在再让我们一起对这些存储以及它们与

计算机体系结构和嵌入式系统相关的特性进行探讨。在我们对ROM和RAM进行探讨前，首先来
307 回顾一下计算机存储发展历史。

7.6.1 早期的计算机存储

值得一提的是，在早期的计算机中有多个存储，特别是极少将程序存储和变量存储混淆在一起并且不会认为它们是同等的。一直到冯·诺依曼机器出现之后，程序和数据才开始共享存储空间。

一般来说，最早期的可编程计算机（正如第1章里提到的）都是通过导线进行硬编程（hard-coded）或通过开关进行编程，并使用真空管或延迟线（delay line）实现位级存储。对这种机器进行重新编程很麻烦，它需要每天不停地重置导线（或开关）来对系统进行编程，这是一件耗时并且容易发生错误的事情。随后穿孔卡片（或磁带）被用来进行程序存储，它在纺织工业里已经使用了200多年。

过去的数据存储都会使用到延迟线，有时还伴随使用其他一些有趣的技术（如阴极射线管延迟线、水银延迟线、声波延迟线等）。它们可以在短时间内保持位信息，从而让计算机能够对其他数据进行操作，效果上相当于简单数字计算器中的存储器功能。

之后，磁芯存储器被发明出来，它被用于对数据和程序进行存储。磁盘也被用于数据和程序存储，后来演化为软盘和硬盘两种形式。

和其他领域一样，电路集成到硅片上带来了存储技术的最大发展。在20世纪60年代中期产生了针对变量的可重写存储器，和针对程序的只读存储器。然而相比磁芯存储器，硅片存储每位的开销要高得多，其结果是在20世纪80年代硅存储虽然占据了大部分计算机市场，但是对于大数据量的存储，还是使用硬盘驱动器。除了最小型的嵌入式系统，非硬盘式计算机直到最近才被认为是可行的。

然而，今天几乎所有的嵌入式系统都包含了闪存（Flash），并且一些品牌的笔记本电脑也开始使用固态存储：理论上说这些存储器功耗低，不会受物理撞击影响，比硬盘式计算机要可靠。

目前对于程序存储和数据存储不加任何区别，本章后面讨论的存储器都可以用来对程序和数据进行存储。然而，不同的存储器与不同类型数据的访问特征相匹配，这点我们将在稍后进行讨论。

7.6.2 只读存储器

只读存储器（Read-Only Memory，ROM）并不是指一种技术，而是一种存取方式：存储在ROM中的数据只能被计算机读取而不能被改写。这意味着这种数据是稳定不变的，这种特征很适合程序代码，也适合需要保持不变的数据（如MP3播放器中的数字滤波系数和开机
308 画面）。

从最底层上说，半导体ROM是一个在硅片上实现的查找表。给定一个地址输入，它会选择该地址对应的门，然后这个门就会将它的状态输出到对应每一位的数据线上。如图7-7所示，它给出了一个4字节ROM示意图，但目前实际使用的ROM布局会比这个图复杂一些。

一些ROM（尽管还使用这个名字）是可写的。但ROM的命名说明了它们最主要的功能还是读取，写操作或者无法实现或者不方便实现。接下来我们对一些ROM技术的变种进行讨论。

一个基本的掩模型ROM芯片包含了地址总线、片选输入、读信号输入以及电源和接地管脚。它能够将当前所选择地址的内容输出到数据总线上。

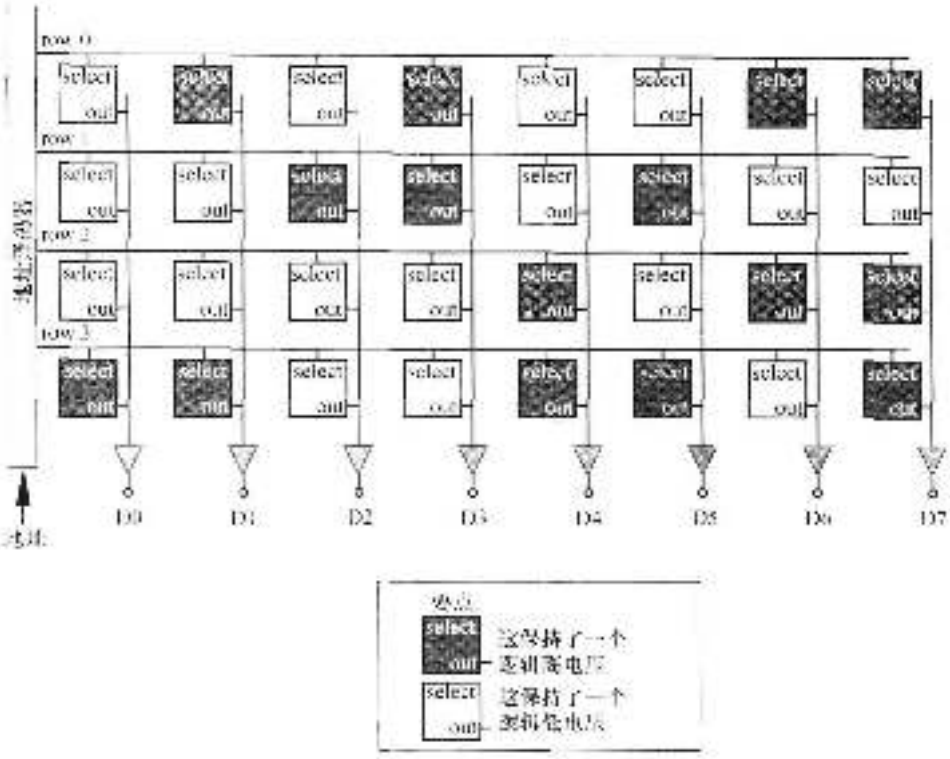


图 7-7 一个 ROM 的基本组织。它是一个由选择单元矩阵组成的阵列，按行寻址，可以输出 8 位数据。如果连接绿色单元的值为逻辑 1，黄色单元所包含的值为逻辑 0，那么选择第 1 行将输出二进制值 00110100b 或十六进制值 0x34。正常操作下，每次只有 1 行被选择

EPROM— 可擦可编写 ROM（erasable programmable ROM（PROM））。在其芯片上方有一个 [309]
称“窗”，通过它可以看到其内部。经过该窗口照射 10 分钟紫外线光，其内部存储数据即可被擦除。然后经过向所选择的数据管脚施加高电压就可以对其进行编写。这个步骤可以通过 EPROM 编程器进行，它通常带有一个插座。由此可以方便地对 EPROM 进行编程。如果一个设备不带有证窗，那么它就是不可擦除的 EPROM（即 PROM）。市面上还有一些基于熔丝的 ROM，它们通过输入高电压对芯片上的熔丝进行熔断来打开或关闭连接。

作为 EPROM 的升级，E²PROM 或 EEPROM 是电可擦除 PROM（electrically erasable PROM），即闪存（flash memory）。这种设备可以通过 12V 电压对其内容进行擦除和重写。而许多现代设备都可以在其内部将 3.3V 或 5V 的供电电压升为 12V。根据测试得知，这种设备都有使用寿命。与字节的数据保留时间和擦除次数相关，通常范围为超过 10 年于 1000 到 10 000 次之间。工程师在选择使用这种设备时都要注意，在其寿命周期内，对其读的次数多而改写次数少，可以延长其寿命。图 7-8 展示了这种设备中的一种管脚分布，它带有一个地址总线和一个数据总线。nWE 管脚（active-low write enable，低电平有效写允许）是一个输出管脚，指出该设备当前可以写入。 [310]
一个真正的 EPROM 与其很相似，有相同的引脚定义，只是 nWE 管脚除外（有发各标为“NC”，即指“没有连接”（no connection））。

目前有两种闪存技术：NAND 闪存和 NOR 闪存，表 7.3 里给出了详细介绍。

---

① 寻址可以根据设计改变，即需要花费的时间，因此，工程师如果希望程序能够保持几天或几个星期，需要个例（十至十五年以上）选用非挥发。

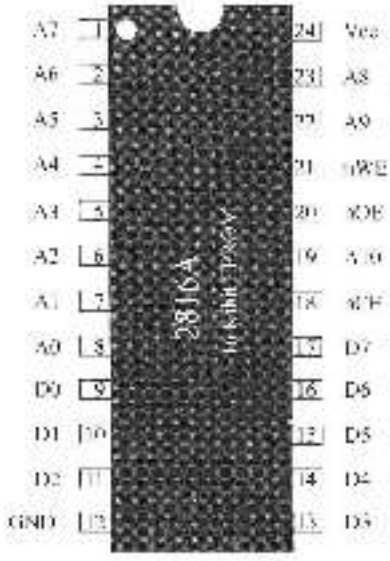


图7-8 一款主流（虽然很老）的电可擦除可编程只读存储器（EEPROM）。它有11个地址管脚，可寻址16Kbit存储空间（即2048字节，字长为8位）。片选（nCE）、写使能（nWE）、读/输出使能（nOE）、GND和Vcc等在图上标出。此款2816A可以进行超过10 000次写操作，并可以保持10年

框7.3 NAND和NOR闪存

当前实用闪存产品有两种技术：NAND闪存和NOR闪存。它们都是以其使用的门电路来命名。NAND闪存是一种廉价、容量很大的闪存技术，其命名来源于其内部结构类似于“与非”门。NAND闪存广泛应用于嵌入式系统中用作数据存储，目前电子盘和MP3播放器等设备都广泛采用了NAND闪存。

NOR闪存与NAND闪存在性能上有很大不同，它的读速度很快，对于程序代码来说就像ROM（另一种存储器）一样，甚至一些设备将NOR闪存集成在CPU内部，使得CPU可以直接从其中读取指令而不需要额外的存储器。

特征	NOR	NAND
容量	大	更大
接口	与SRAM类似	非常复杂
存储单元	独立存取	按页存取
擦除周期	约100 000次	约1 000 000次
擦除速度	很慢	很快
写速度	慢	快
读速度	快	快
坏块大小	无	有
价格	较贵	较便宜

这两种闪存的其他特性如下表所示：

对于嵌入式应用，代码存储器，我们最常接触到的是NOR闪存（本书重点，特点是支持片内执行）。因此，我们将在本书的后续章节中着重介绍NOR闪存。

串行闪存。如图7-9所示，是采用串行接口而非并行接口的闪存，它仅有25MHz的串行总线，命令字、地址字节以及控制信号都要通过串行总线进行传输，这是造成串行闪存速度比并行闪存慢的主要原因。首先要指定读/写地址（这种指定通常需要花费一定时间），然后是任意数量的字节读或写（其速度会快很多），这种寻址机制使得它很适合那些顺序读取的信息存储，但对于随机读取或对单个字节进行写操作，其效率很低。

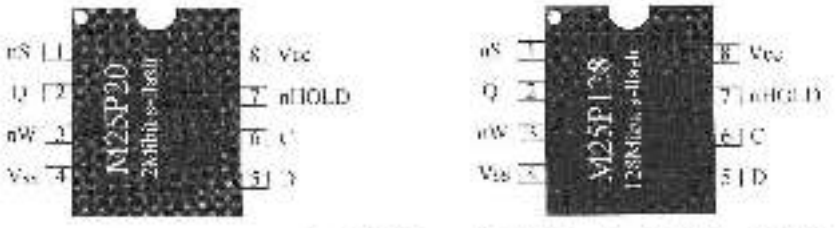


图7-9 串行闪存在同一个接口类型下使用串行接口、多项控制、地址和数据。存储器阵列闪存时并不需要额外的专用引脚，尽管右侧的数据大小是存储器阵列左侧的64倍。可以看到的是这个数据闪存是很小的，只有6mm×5mm大小

大多数闪存设备，不管是串行存取还是并行存取，在其内部都被布局为多个块或页。这些闪 311
存在最初使用时所有的字节都被初始化为0xff。换句话说，这些闪存的每一位在最初都被初始化为“1”。可以寻址和读取闪存中的每个位置，对每个字节的访问都会返回0xff。

对存储中的任何位置都可以进行编程。当需要对某一位编程为“0”时需要将该位的“1”清为“0”，而需要编程为“1”时则保持不变。

例如，初始时某位置为字节0xff，如果需要将其编程为0xf3，则该字节将变为0xf3。如果再将同一个位置编程为0xa7，则该位置将变为0xf3和0xa7相与的结果，为0xa3（因为1010 0111 AND 1111 0011 = 1010 0011）。显然，如果不断地写某个字节，它最终将变为0x00。所以开发人员在使用闪存时可以看到未被擦除的位置被设置为0xff。

每当闪存被擦除，它上边的每个字节都将被置为0xff。事实上，这种擦除是按块来进行的，所以一旦执行了擦除命令，所选择的一整块都将变为0xff。同时也可以将某一块锁住，禁止对其进行擦除操作。

读闪存与读ROM的方式是一样的，都要遵循6.2节中所述的标准总线传输。本质上，这意味着如果CPU要读取所连接的外设闪存，则需要进行（ⅰ）在地址总线上设置需要访问的地址；（ⅱ）置片选信号nCE；（ⅲ）置输出使能信号nOE；（ⅳ）允许设备在某个时刻访问到指定的位置，确定其内容并输出到数据管脚上；（ⅴ）从数据总线上读数据，这个步骤必须在（ⅵ）对所有的信号进行复位前完成。

写操作与读操作类似，只是需要在数据总线上设置要写入的数据以及置写使能信号（nWE）而非nOE。如果这个过程是在一个SRAM芯片（将在接下来的章节中讨论）上进行，它将被写到指定的位置。然而，对于闪存这个过程稍微有点复杂。它需要写入一系列指令以对其进行控制 312
（在指定存储位置被写入前）。下表分别给出了来自Atmel和Intel的两款闪存设备的写入控制指令序列：

	Atmel AT29xxx		Intel 28F008SA	
	数据	地址	数据	地址
编程	0xaaaa	0x5555	0x10	<*addr*>
	0x5555	0x2aaa	<*data*>	<*addr*>
	0xa0a0	0x5555		
	<*data*>	<*addr*>		
擦除扇区	0x00aa	0x5555	0x20	<*addr*>
	0x0055	0x2aaa	0xd0	<*addr*>
	0x0080	0x5555		
	0x00aa	0x5555		
	0x0055	0x2aaa		
	0x0050	<*addr*>		
擦除设备	0x00aa	0x5555	不支持	
	0x0055	0x2aaa		
	0x0080	0x5555		
	0x00aa	0x5555		
	0x0055	0x2aaa		
	0x0010	0x5555		

因此，对Atmel闪存的地址0x1001写入字0x1234需要4个写周期：

- 写 0xaaaa 到地址 0x5555。
- 写 0x5555 到地址 0x2aaa。
- 写 0xa0a0 到地址 0x5555。
- 最后，该闪存向地址 0x1001 写入数据 0x1234。

对于 Intel 闪存，指令序列相对短一些：

- 写 0x0010 到地址 0x1001。
- 写 0x1234 到地址 0x1001。

之所以需要如此烦琐的写步骤是为了避免对闪存的意外重写（这种情况在 CPU 程序错误运行时可能会发生——产生一个随机地将数据写入不同地址的程序并不困难）。闪存上一般还有另一层保护机制，它会检测供电电压，如果电压较低或出现明显波动，将会禁止写入。CPU 还可以对闪存上不同的状态寄存器进行读取（这也需要通过写一系列的指令将设备转入“读取状态寄存器模式”或其他类似模式，之后的一条或两条读指令将返回状态寄存器的内容）。另一条指令可以读出设备的制造商和设备识别信息，因此一个好的程序可以根据所连接的不同设备确定相应的编程算法。

需要注意的是，虽然之前所示的是目前大部分厂商所遵循的两种主要指令控制序列类型
313 （即大部分设备的控制方式与这两种相似），但是不同的制造厂商对自己的闪存是有不同的指令控制序列的。

闪存从本质上是一种基于块的技术——虽然根据需要可以对独立的字进行读或编程，但擦除操作是按块进行的（这一点对于所有基于闪存的技术是一样的，例如紧凑式闪存（compact flash，CF）存储卡、安全数字（secure digital，SD）存储卡、记忆棒（memory stick）等，虽然对于这点大部分用户都是不知道的）。对闪存中一个 64KiB 块改变一个字节通常需要经过以下步骤：

- 将闪存的整个块读入 RAM。
- 在 RAM 上找到需要改变的字节并用新的值进行替换。
- 发送指令序列对闪存上的块进行擦除。
- （等待以上操作完成。）
- 发送指令序列开始进行写操作，将整块写回到闪存上。

块一般都非常大——之前提到 64KiB 的块并不是太常见的，所以闪存对于存储那些经常要改变的小变量并不是太好的选择。

从一个编程人员的角度，指定不同类型的信息存储在不同的块上是很有用的。在嵌入式系统中，对于引导内存有特别要求（这点将在 7.8 节进行讨论）。一种简单的方法就是将经常需要重写的内容（如配置信息）放在同一个块内，而将不经常重写的内容放在另一个块内。

随着闪存使用时间的增加，它的速度会逐渐变慢，擦除或编程几个字节都变得十分耗时。显然，我们更希望闪存不会降低所连接的计算机的速度，因此闪存的设计人员提出了一些聪明的方法来解决这种问题。图 7-10 是其中一种方法的模块图，它将一个块大小的 RAM 加入到闪存中。编程人员想要对闪存中的一个块进行写操作时，可以先将数据快速地写入到这个基于 SRAM 的 RAM 块上，然后发出指令让设备自行将整个 RAM 的内容拷贝到目标闪存块上。类似地，当只有一个字节需要改变的时候，闪存块首先被自行拷贝到这个 RAM 上，然后编程人员将所需改变的字节写入到 RAM 上，再发出指令擦除目标闪存块并从 RAM 上拷贝。

如图 7-10 所示的闪存结构在目前主流的并行闪存设备中被广泛采用。在串行闪存中，nOE、nWE 及其他控制信号都是由串行接口控制器发出，而不是直接从并行接口获得。

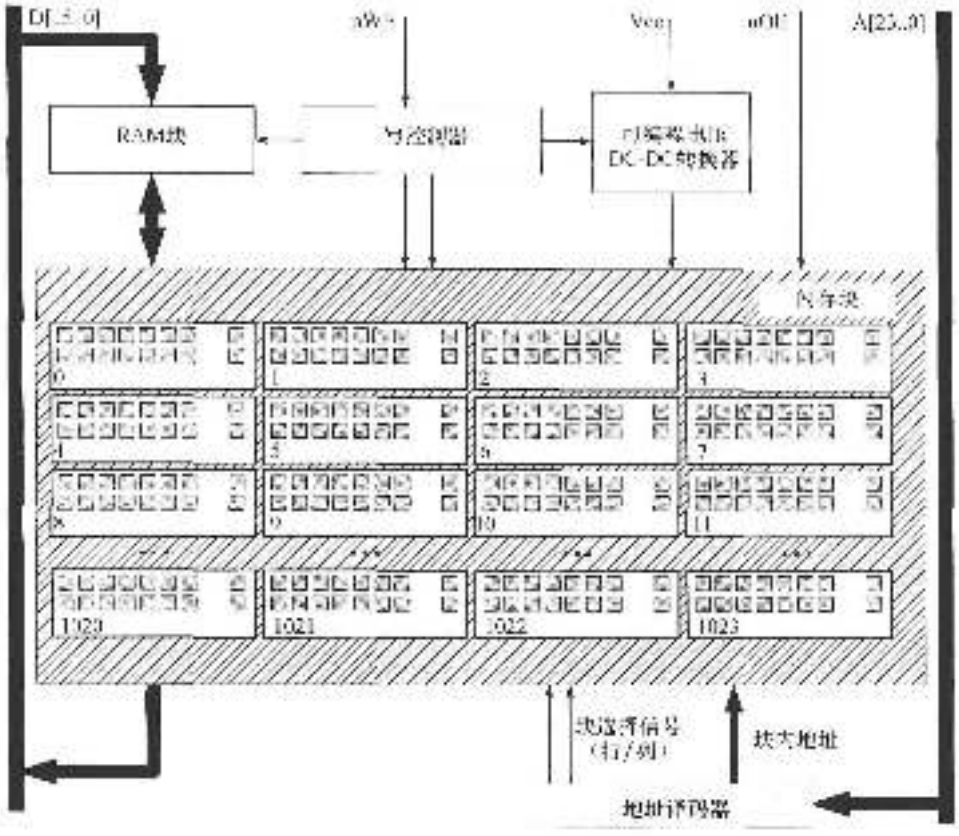


图7-10 带有RAM的闪存内部结构模块图。该RAM为块大小，用于存储编程数据。这个闪存阵列包含多个相等的块。这种规整结构使得制造商可以通过增加更多行的块来提高闪存容量（实际中一般都多于4列）。箭头所指的方向连接着数据总线。

7.6.3 随机存取存储器

"随机存取存储器"（Random Access Memory，RAM）这个词与ROM一样，描述的是一种存取方式而不是一种技术：它是指存储器中的任何位置都可以根据需要任意存取（包括读出和写入）。这种存取对于今天的计算机来说是很普遍的，与它相对的是串行存取，如在磁盘和基于延迟的存储器上，获取数据的顺序与数据被写入的顺序相同。在早期的计算机中串行数据存取产生的约束问题不是太普遍。 314

当然，串行存取和随机存取之间还有一个区别——RAM是可以寻址的，因此它需要一个地址指出所要存取数据的位置。对于更为普遍的并行总线存储，这个地址由并行地址总线给出。有时候地址总线与数据总线复用。而串行存储设备则使用串行机制来传输地址（如7.6.2节所讨论的串行闪存设备）。 315

总体而言，目前有两种类型的RAM技术：静态RAM（static RAM，SRAM）和动态RAM（dynamic RAM，DRAM）。而DRAM又可以细分为几种子类，我们将会在指出进行简要介绍。右表是SRAM和DRAM的主要区别。

SRAM	DRAM
每位由六个晶体管构成	每位由一个晶体管构成
较低周期	较高周期
不需要规新	需要周期性刷新
大容量存储需要	大容量存储便宜
工作时需要较高电压	工作时需要较低电压

7.6.3.1 静态RAM

SRAM虽然被称为"静态"的，但是它的IC是一个易失性存储器——当失去供电时，所存储的数据就会丢失。之所以被称为"静态"是因为它的每个存储单元只要保持供电就可以一直保持数据状态，而不需要刷新过程。而我们稍后将提到的动态RAM，就需要进行周期性刷新。

SRAM速度相对较快，但由于其每个逻辑单元的电路复杂度是DRAM的数倍，所以它价格更贵、密度更低并且在读写过程中消耗能量也更多。但现代SRAM在非读写状态时的功耗比DRAM低，这是因为DRAM不像SRAM，它在没有访问时候也需要进行周期性的刷新操作。

SRAM在使用和连接上与ROM很相似。参考图7-11中所示的两款SRAM管脚分布，与图7-8的EEPROM在数据连接方面有相似之处，只是管脚位置略有不同。图7-11中的两款SRAM的容量分别为16Kibit和1Mibit。16Kibit的SRAM有11个数据总线管脚（因为$2^{11}=2048\times8\text{bit}=16\,384\text{bit}$），而1Mibit的SRAM则有17个（A[16..0]，因为$2^{17}=131\,072\times8\text{bit}=1024\text{Kibit}=1\text{Mibit}$）。

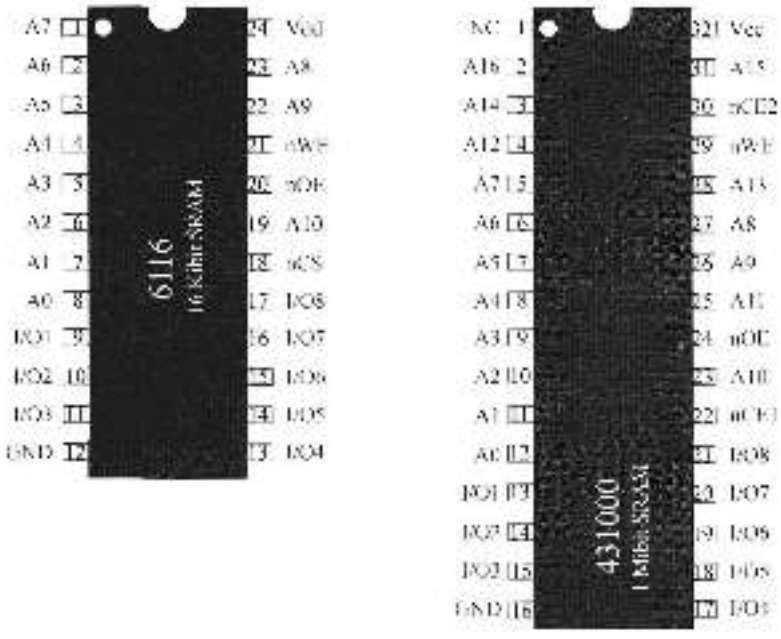


图7-11 两款早期SRAM（16Kibit的6116和1Mibit的431000）管脚图。注意到它们都拥有相同的8位输入/输出端口（通常与数据总线连接），单电管脚，片选管脚（nCS）和读写管脚。而右边芯片的容量是左边的64倍，因此上面多的6个地址管脚（A11～A16）

SRAM与ROM一样拥有规整的内部单元结构。图7-12给出了一个简化的SRAM矩阵图，对这些单元可以并行地进行独立寻址（与图中的8位并行总线相连接），并且在地址被选择的情况下可以对相应单元进行读出或写入。双向缓冲将外部数据总线和内部数据线连接起来，受方向控制，以避免跟其他连接在相同外部数据总线上的单元发生总线竞争。

SRAM通常在单片计算机上被用做cache存储器以上存储。在简单的小型嵌入式微控制器中，SRAM通常被用做外部存储，其存储容量一般为数十KiB（因为至少在低密度的DRAM和SRAM的价格差别不大，且数控制器一般较为简单，从而不支持DRAM）。

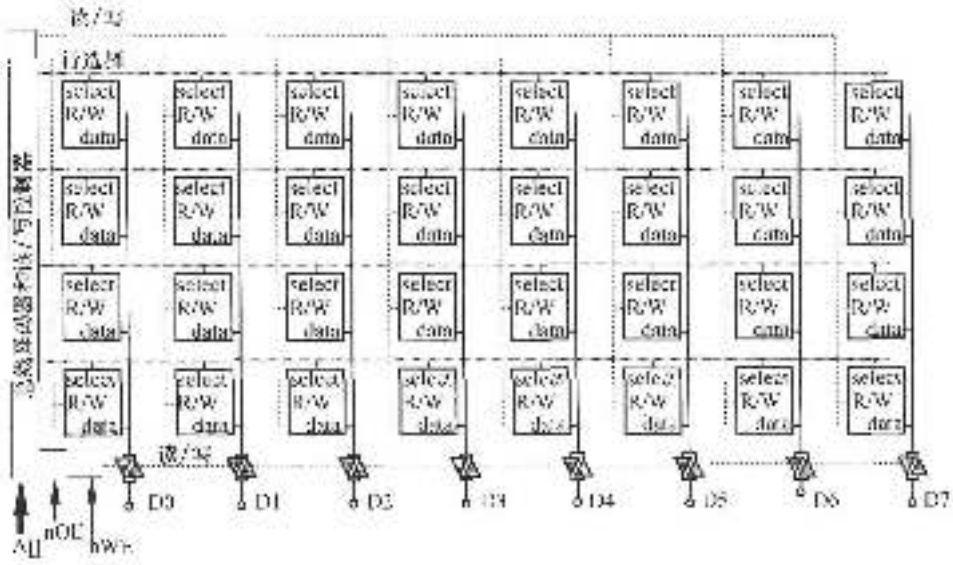


图 7-12 简化的 SRAM 内部结构模块图。它由一个存储单元阵列组成，通过总线译码器和读/写控制器的控制可以对其进行读出和写入

7.6.3.2 动态 RAM

之前曾提到过，动态 RAM 之所以被称为“动态”是因为它总是在不断地变化：每个单元所存储的值由与每位对应的单一晶体管相连接的电容决定。由于门是会“漏电”的，因此这些电容会不停地放电。刷新的过程是依次读取每个单元，然后对其电容进行适当的充电。任何未被刷新的单元都将在几毫秒内丢失其电荷。

写过程是将电荷通过晶体管充入电容（逻辑高）或对电容进行放电（逻辑低）。有趣的是，通过读过程确定单元内的值后，还会刷新该单元。因此对整个 DRAM 进行周期性的读取也可以对该 DRAM 进行刷新。

现代 DRAM 都是高度集成的，并且大部分与 DRAM 相连接的微处理器（或其他支持 DRAM 的芯片）都会自动进行刷新过程，这只需对几个寄存器进行正确的设置。然而，刷新需要消耗一定的时间，这个时间也许就是 CPU 等待存储数据的时间，但这对 CPU 性能影响不大。 317

DRAM 从 20 世纪 60 年代中期发展到现在经过了一段非常长的历史，在这期间经历了许多发展。表 7-2 列出了一些重要里程碑，包括发布时间、时钟速度以及工作电压。

表 7-2 SDRAM 技术发展史上一些重要的里程碑

名称	年份	时钟频率	电压
基本 DRAM	1966	—	5V
快页模式(Fast Page Mode，FPM)	1990	30MHz	5V
扩展数据输出（Extended Data Out，EDO）	1994	40MHz	5V
同步 DRAM(Synchronous DRAM，SDRAM)	1994[*]	40MHz	3.3V
rambus DRAM(Rambus DRAM，RDRAM)	1998	400MHz	2.5V
双倍数据速率（Double-Data-Rate，DDR）SDRAM	2000	266MHz	2.5V
DDR2 SDRAM	2003	533MHz	1.8V
DDR3 SDRAM	2007	800MHz	1.5V

[*]IBM 较早发布了 DRAM 的时间比这个时间早。

注意：DDR 和 DDR RAM 各种不同的速率被略去，因为它们工作的核心频率是相同的。 318

DRAM与SRAM的最大区别在于它的本质是动态的，需要持续刷新。由于DRAM位存储单元比SRAM的小很多，所以DRAM比SRAM要便宜且密度更大。但是DRAM比SRAM速度慢，并且在没有读写的时候的功耗比SRAM要大（但在有存取时SRAM的功耗更大）。

DRAM的寻址机制也是与SRAM的另一个主要区别。参考图7-13中给出的两个大小分别为16Kibit和1Mibit的早期DRAM芯片的管脚分布图，首先，可以看到它们都带有几个不常见的信号nWRITE、nRAS、nCAS、Din和Dout，这些我们将在稍后介绍。其次，将DRAM芯管脚分布与图7-11所示的SRAM管脚分布相比较。我们发现虽然这两张图给出的都是16Kibit和1Mibit大小的两个存储芯片，但在SRAM的例子中，右边芯片（1Mibit）的地址管脚数比左边芯片（16Kibit）的多6个，而在DRAM的例子中，右边芯片的地址管脚数只比左边芯片多了3个。由于64倍容量的差别在地址空间上应为2^6扩展，所以应该需要6个额外地址管脚。由此看来DRAM芯片并没有我们目前所看到的那样简单，我们将在稍后进行探讨。

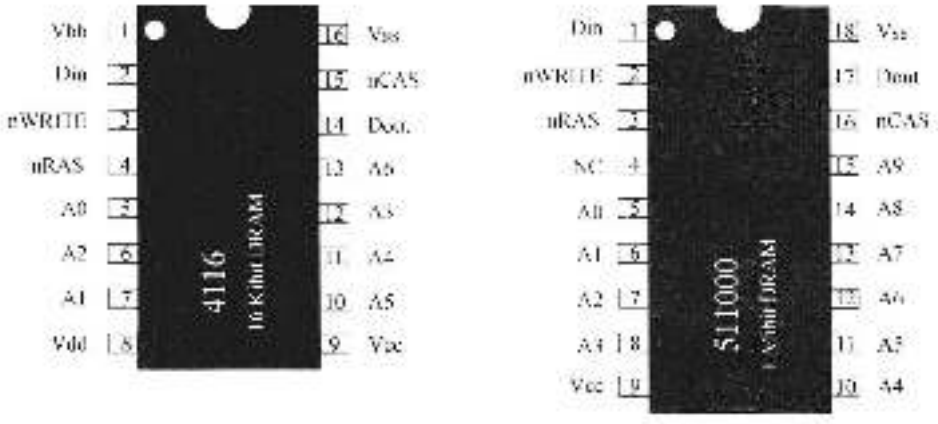


图7-13 两个早期DRAM芯片（16Kibit的4116和1Mibit的511000）的管脚图。这两个芯片都只输出单个位数据（因此当与8位数据总线连接时需要8片并行连接）。注意它们的控制信号都是一样的，虽然右边芯片的容量为左边芯片容量的64倍，但只多出3个地址管脚（A7 - A9）。Vbb、Vss、Vdd和Vss分别是不同的电源管脚

7.6.3.3 DRAM的寻址机制

319 首先，我们需要提醒的是，DRAM都是按数据行（简称为“页”）和列一起寻址的。这点与我们之前讨论的那些只按行寻址的存储结构不一样。事实上，管脚分布与如图7-14所示的管脚图相似的DRAM是1位DRAM——为了构成8位的数据总线，需要8片这样的DRAM并行执行，每片对应一位数据。这些并行芯片的Dout管脚分别依次连接到数据总线的D0、D1、D2、D3等上。

从如图7-14所示的DRAM内部结构中我们可以更清楚地了解这种按行和列一起寻址的机制。图中所有的内部存储单元分布成一个矩形矩阵。每个存储单元内包含一个晶体管和一个用于存储电荷的电容。当行地址开关（nRAS）被激活时，将会把地址总线上的内容装载至行地址锁存器。多路输出选通器根据行地址信号匹配确定哪行（页）将要输出所存储的电荷。列地址开关（nCAS）也被激活，将地址总线上的内容装载至列地址锁存器，然后根据列地址信号确定所选择的行中哪一位将要被选择输出。

与每一条位线（列）相连的输出放大器负责检测所选择单元电容中的电荷，并将其充满。由此，在选择某一页之后，如果电荷比位线上的某个阈值大，则输出放大器输出一个电压，对这
320 条位线上相关的单元电容进行充电。如果所检测到电压低于该值，则输出放大器不会输出该电压。

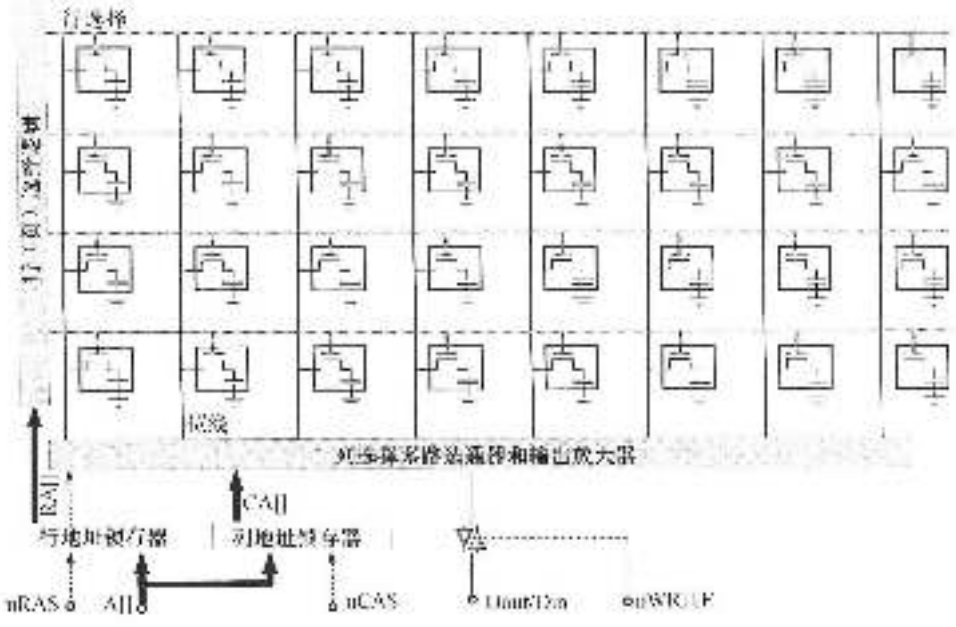


图7-14 DRAM的内部行/列选择机制示意图。其中行地址锁存器和列地址锁存器分别存储行地址和列地址，行地址和列地址都由相同的地址总线传输，分别由行地址开关（nRAS）和列地址开关（nCAS）选择。这个矩阵到输出与数据总线上的一位相连

实际上，输出放大器在nRAS选择某一行之后被触发，而这个充电过程也是自动进行的。即DRAM的“刷新”过程不需要寻址列地址——所需的就是轮流按行进行刷新（正如之前所提到的，大部分支持DRAM或SDRAM的CPU都将自动完成这个过程）。DRAM一般每64ms需要进行一次刷新，因此每一行都需要能够在此时间内被选择一次。

当然，许多DRAM芯不是1位DRAM，而是字节型或字型的。这种情况是将基本的DRAM在一个芯片上复制成多个副本。图7-15是一个与8位总线连接的DRAM。它密度很低，只是一个256位的存储器。由于每一位对应的单元都为4行8列，所以行地址为2位，列地址为3位。

一个16Kbit的存储（如图7-13所示的4116）有128行128列（$128 \times 128 = 16384$），因此需要7（$2^7 = 128$）条地址线来对这些单元进行寻址。一个与总线相连的CPU从这种芯片上读取一位主步骤如下（初始时所有信号都为无效的——nRAS、nCAS、nWRITE都是高电平）：

1. 将行地址输出到地址总线上。

2. 使能nRAS（将其从逻辑高变为逻辑低，由此使得行地址锁存器能够捕捉地址总线上的行地址）。

3. 将列地址输出到地址总线上。

4. 使能nCAS以锁存列地址。

5. 存储器将在一定时间后将所选择的存储单元的内容输出到与其相连的数据总线上，这些内容就可以被CPU读取了。

6. 撤销nCAS并停止驱动地址总线。

7. 撤销nRAS。

通过观察我们知道对DRAM进行读写要遵循一些严格的时序。有一点是明确的，由于每存取一位需要两个地址对其进行寻址，因此DRAM的存取速度没有像SRAM那种只需一个地址就能进行访问的存储器快。但从开销和密度上考虑，这是可以容忍的：如图7-13所示，将16Kbit的DRAM扩展为1Mbit只需额外增加3条地址线，而对于SRAM（图7-11）则需增加6条。对于

更大密度的存储，DRAM 个管脚数量上的优势则更为明显。

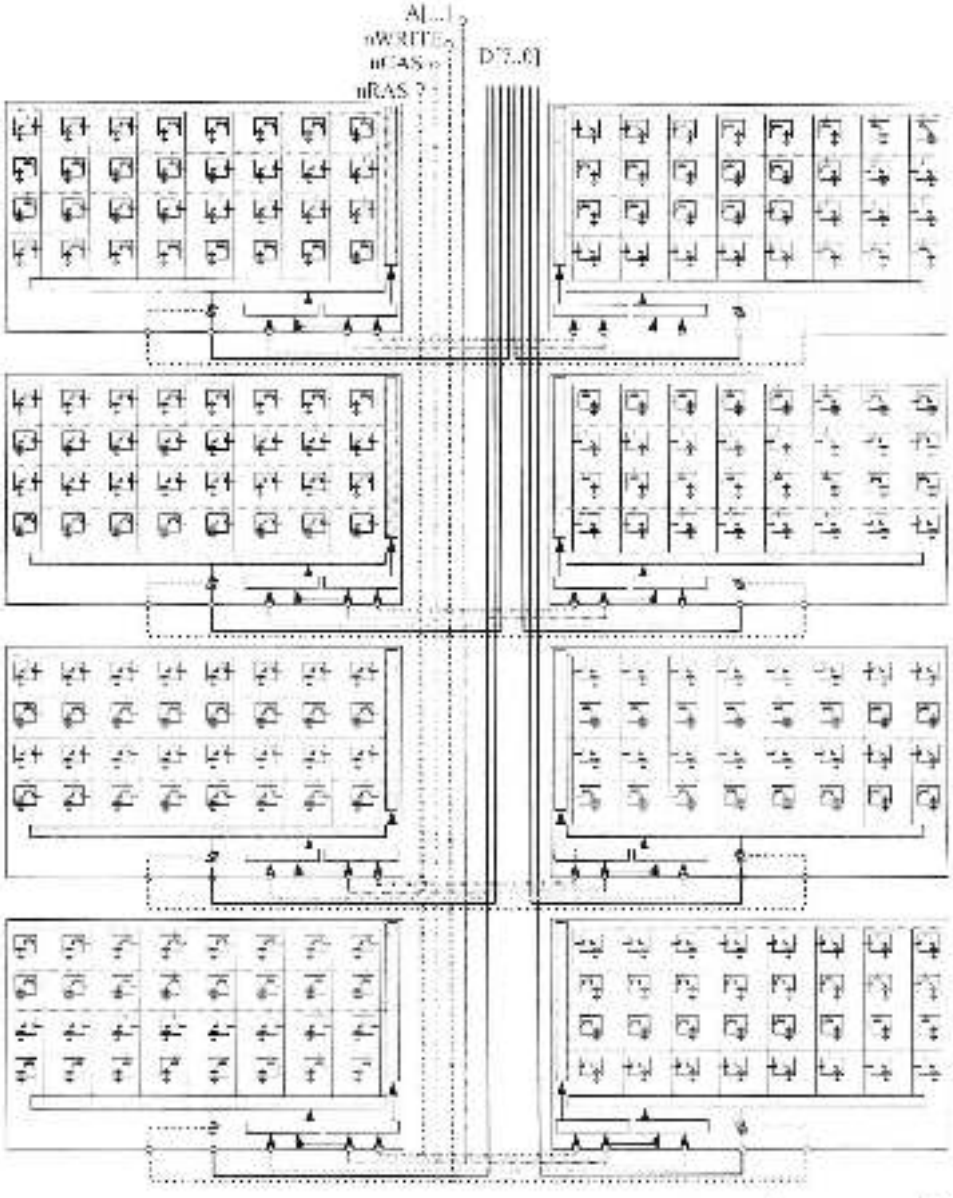


图 7-15 将图 7-14 的 1 位 DRAM 复制 8 个副本构成一个与 8 位总线相连接的 DRAM。对所有内部块而言控制信号和寻址信号都是一样的（实际上所有的块都共享同一个行地址锁存器和同一个列地址锁存器）

为了避免增加管脚数量，设计人员在行/列寻址机制上找到了更聪明的办法。例如，对同一行连续读就可以避免使能 nRAS 信号（因为所读取的行相同，其地址也相同），读一写或写一读组合也能对其进行简化。

321
322

这些技术所带来的优势巨大，我们在表 7-2 中列举了一些。表中列出的第一项快新是快页模式（fast page mode），它允许对某一页的多个位置进行读从而不需要重复使能 nRAS。

DRAM 也适合用作视频卡，称为视频 RAM（Video RAM，VRAM）。它的特征是拥有两个数据输出口且十从存储阵列中读取数据。一个端口（与主 CPU 相连）用于处理器对存储器进行读写

访问。另一个端口与视频数模转换器（Digital-to-Analogue Converter，DAC）相连接，是只读的，可以对阵列中的数据进行按位读取并显示在屏幕上。

回到通用DRAM，*扩展数据输出*（Extended Data Out，EDO）类型的存储器采用了一个内部锁存器保存该页的数据，如此可以使得CPU在开始读取下一页时仍能读取当前页的数据。这实际上是流水线的一种形式，可以通过阻塞多个读操作对其进行优化，如此这些读操作可以一起执行（在EDO DRAM的*突发模式*下最多支持4个读操作）。特别是对于多片存储，如果能够良好地对存储块进行交织，则可以交错读取存储块，从而进一步提高存取速度。

至此，之前所提到的各DRAM变型虽然都由CPU控制，但它们与CPU都是异步的，而CPU自身是同步的。而事实上很明显，如果需要进一步提高这些存储设备的性能就需要利用总线时钟，由此同步DRAM（SDRAM）就被发明出来了。同步使得存储设备能够提前预取数据以便为下一个时钟周期做好准备，从而能够更好地通过交织内部存储访问或使用其他技术来对读写操作进行流水化处理。

SDRAM性能的最大改善来自提高它们的时钟频率和允许在存储时钟的上升沿和下降沿都进行数据传输（即将每个周期传输一个字变为每周期传输两个字——一个在下降沿传输，另一个在上升沿传输）。这就是所谓的*双倍数据速率*SDRAM（DDR SDRAM）。

7.7 分页与重叠

我们在7.6节中见识了真正的存储器设备，现在再来回顾一下4.3节中介绍的内存管理单元（MMU）。MMU的配置通常被认为是一个相当复杂的话题（作者有真实经历，在十分紧迫的工程时间约束下使用低级汇编语言实际配置MMU是十分困难的）。

在大部分拥有MMU的系统中，存储页和外部大容量存储设备进行进出交换，这些大容量存储设备通常都是硬盘。存储器管理系统保存着这些页面信息，在任意时刻知道它们是存放在内部存储器上还是外部硬盘上，并且可以根据CPU的指令来进行载入或者保存操作。

MMU功能是长期创新和逐渐改进的结果，但是现在往回退几代，我们来考察一下在没有 323
MMU的情况下如何设计。这并不是一个简单的思想实验，在现代嵌入式处理器中通常都会受到片上存储的限制，设计者经常会耗尽处理器上的RAM空间。

让我们考虑一种真实情况，软件工程师正在为一部手机中的嵌入式处理器开发控制程序。临近开发末期，他们所编写的代码规模以及所要占用的存储空间分别如下：

- 运行时存储：18KiB的RAM
- 程序存储：15KiB的ROM

可能发生的情况是处理器只有16KiB的内部RAM，明显不能满足程序执行和存储要求。如果在系统中片上存储或者并行外部ROM可用，那么程序就可以直接从这些存储器上执行（但是任何读写代码一般都存放在RAM中，大部分情况下所谓的ROM可能实际上是闪存）。然而，让我们假设这种情况下可提供的闪存是1MiB，并连接在一个25MHz的SPI（串行外设接口）串口上。

不幸的是，从这些存储器上执行代码实在是太慢了。

事实上，设计者很关注系统的时间特性。从上电开始，这个处理器在程序开始执行之前，用了将近5ms来将程序代码从闪存中读入RAM（$15 \times 1024 \times 8\text{bit}/25 \times 10^6\text{s}$）。

假设不考虑提高代码效率和采用更多RAM的解决方案，设计者必须在系统中采用重叠（overlay）技术来适应当前的存储器配置。这个技术的使用原则是所有软件并不都在同一时刻运行，事实上，软件的某些部分是互斥的。例如，无线设备软件允许设备运行在传统模式，设备在上电时可以选择是普通模式或者传统模式，但是不能同时选择两个模式。记住这些，就不难解释为什么这两种模式的软件程序不应该一起存放在RAM中，而是在具体的模式被选择时再载入所

对应的程序。

设计者因此将所要执行的代码分为两个可执行部分或者重叠部分。一部分是传统模式的，一部分是普通模式的。这种做法看上去不是很高效，因为两个部分共享很少的功能，而且共享的功能可能要提供两次，也就是对两个重叠部分分别提供。另外还需要一个额外的代码初始化选择器来实现两个重叠部分之间的切换。那么这种做法是一种解决方案吗？

考察下面的情形，代码所需的存储规模如下所示：

- 在普通模式下运行所需要的运行时存储容量：12KiB
- 在传统模式下运行所需要的运行时存储容量：10KiB
- 重叠选择器的代码规模：ROM 中的 1KiB
- 普通模式下的代码规模：ROM 中的 10KiB
- 传统模式下的代码规模：ROM 中的 9KiB

324

所占用的闪存总和为：1 + 10 + 9 = 20KiB（上面的方案需要 15KiB）。然而，这对于 1MiB 的闪存来说并不是个问题。

但是开启速度如何呢？一些工程师很关心这种方法是否会让软件的开启速度变慢。从实验结果中可以看到这种方法实际上使得开启速度变快了。

在普通模式下只需要 3.6ms 就可以将 11KiB 的数据传送完毕，在此加入了选择代码，它们的执行时间可以忽略不计，因此总的时间为 $11 \times 1024 \times 8\text{bit}/25 \times 10^6\text{s}$。

在传统模式下，开启速度只需要 3.3ms（$10 \times 1024 \times 8\text{bit}/25 \times 10^6\text{s}$）。

总之，这两种模式下的开启时间及在 RAM 上的运行时间在使用重叠技术后都有所优化，尽管需要更多的软件空间和 ROM 空间。

没有 MMU 对存储器进行管理，对于代码体积大于 RAM 本身的这种情况，其处理方法很直观：编写一个和选择器一样简单的重叠模块载入程序，在两个执行文件间进行切换，或者设计一个更复杂的重叠文件，其自身就包含一部分选择和载入代码，来自行选择和载入下一个重叠模块。

然而，对于现代嵌入式处理器来讲还有另外一个选择，使用一个先进的操作系统，这个系统可以模仿 MMU 的部分功能。一个很著名的例子就是 uCLinux，这是一种针对不含 MMU 的处理器的 Linux，该操作系统中包含很多标准的已编译 Linux 代码，包括闪存文件系统、现场执行（execute-in-place，XIP）驱动器等。

最后一点：这种重叠方法也适用于 FPGA（现场可编程门阵列）技术。这种设备可以通过重新编程完全改变它们的固件功能，顾名思义，它们可以在任何地方（甚至现场）对设备进行编程。我们可以考虑时下比较热门的一个概念——软件定义的无线电（Software Defined Radio，SDR），其含义是指采用通用硬件来实现多种数字无线电处理。这种设计思想允许设备载入多种译码结构中的一个来匹配当前频率下的通信模式。一个前端选择器监控当前的无线信号频率，并且决定哪种调制方法适用于当前频率，之后把相应的固件载入到 FPGA 中来解调和译码当前信号。由此看来，在未来几年的移动电话中，重叠技术将会有用武之地。

7.8 嵌入式系统中的存储

大部分计算机体系结构教材都会介绍大型计算机的存储系统，甚至包括共享存储的并行处理机，但是他们忽略了对嵌入式系统的讨论。

325

嵌入式系统与台式计算机在存储器使用上有很多不一样的地方。尽管嵌入式系统应用于不同方面，外形大小各异，但现代嵌入式系统中的大规模存储通常使用闪存，而在台式计算机则使用硬盘。

在这里，我们来观察一个典型的嵌入式系统，它基于ARM9并且运行嵌入式Linux操作系统。我们所提供的这种结构在中等规模的嵌入式系统中非常典型，我们也可以设计一个更小规模的系统，其最大RAM约为100KiBit，这种系统能够运行一个单片机实时操作系统（在一个可执行模块中包含一个操作系统、应用程序和引导程序）；我们还可以构造一个类似PC的较大系统，其中采用小型x86处理器，相当于精简的低功耗PC。

如图7-16所示是一个中等规模的系统（这是现实中存在的，包含一个三星S3C2410），非易失的程序代码存放在闪存中，易失的运行代码和数据存放在SDRAM中。闪存为16位宽，SDRAM则为32位（这里使用了两个16位宽的SDRAM）。

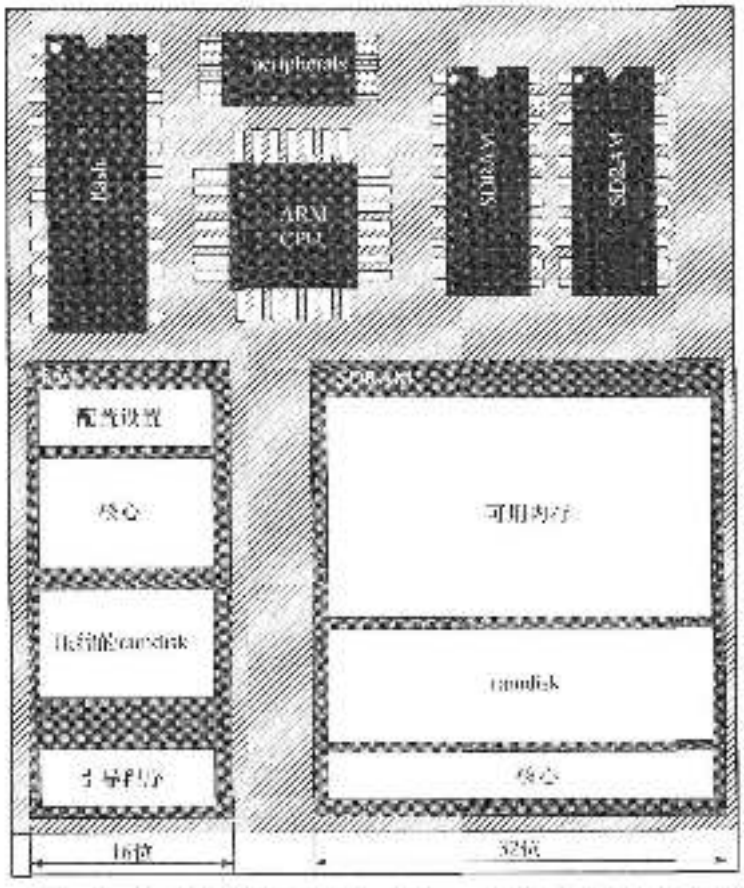


图7-16 基于ARM的嵌入式系统的存储配置示意图，并给出了系统正常操作过程中闪存和SDRAM的内容

在图7-16的靠下面部分可以看到每个类型的存储器在执行过程中所包含的内容，而我们仅考虑在三个阶段的操作过程中的存储器内容。

7.8.1 非易失存储器

在处理器断电期间，只有闪存中的内容能够得到保存，SDRAM基本上是空白的。当ARM处理器上电或者复位后，CPU就开始载入指令，并从0x0000 0000地址处执行一段程序。在这种情况下，如大部分嵌入式系统一样，闪存被分配到上面这个起始地址。因此，闪存中的第一条指令将在复位后立即执行。

CPU将直接从闪存上提取指令和数据。引导加载程序需要完成一些重要任务，例如重新设置处理器状态，关掉看门狗定时器（7.11节），并且复位SDRAM。这也是为什么大部分的嵌入式开发者

需要学习一些SDRAM知识的原因，我们需要在引导加载程序执行过程中配置这些存储器。

有很多免费的引导加载程序可以选择，比如流行的U-boot。很少看见设计者自己编写为实现特定功能而定制的引导加载程序。引导加载程序可以完成的功能如下：

- 完成上电自检（POST）。
- 复位存储器，特别是SDRAM。
- 设置CPU寄存器，比如用于时钟分频、功耗控制、MMU、cache等。
- 发出一条信息给串口、LCD显示屏等。
- 随机等待用户干预（比如说“按任意键进入启动菜单，或者等待5秒后继续”等）。
- 将内核或者内存盘（ramdisk）从闪存载入SDRAM。
- 跳转到起始地址上执行代码（比如内核）。
- 检测内存。
- 擦写闪存上的块。
- 将新的内核或者RAM磁盘载入SDRAM。
- 将SDRAM中一个内核或者RAM磁盘写入闪存。

326 ~ 327

在我们所考虑的系统中，有三项需要载入闪存。第一个，在闪存的“底部”，也就是0x0000 0000地址处，存放引导加载程序。第二项是经过压缩的RAM磁盘，最后一项是操作系统内核。

对嵌入式Linux操作系统也进行划分，其中RAM磁盘（代替台式系统中的硬盘）包含了应用软件和数据，而内核包含了操作系统最基本的执行模块。RAM磁盘实际上是一个文件系统，包含了各种文件，有些是可执行的，所有这些文件以gzip形式进行压缩，一般会在1MB或2MB这个数量级上。

内核是操作系统的核心部分，包含了所有的系统级功能、内置的驱动、底层的访存程序等。内核程序是不可改变的，即使由于新的应用软件出现而有可能更新RAM磁盘。在嵌入式Linux中，引导加载程序最先执行内核以开始系统运行，然而，内核和RAM磁盘必须放置在正确的存储区域。让我们看一下引导的具体步骤：

1. 系统上电。
2. 引导加载程序启动，设置系统并且给LCD或者串口发送一个指令。
3. 引导加载程序等待用户的输入或者等待没有收到任何输入后的超时指令。
4. 引导加载程序从闪存中复制一份内核并放入SDRAM中的某一区域（这等同于对内核的编译时设置）。
5. 然后以类似的方式复制RAM磁盘。
6. 引导加载程序执行一条跳转指令，使此时的指令地址变为内核的起始地址，从而将控制权交给内核。

现在，内核可以运行了。

7.8.2 易失存储器

内核开始执行，会在屏幕上打印一行信息并且解压它自己（内核基本上都是压缩过的，只有在开始的一小部分代码未压缩，负责解压缩并且执行后面的程序）。

328

然后，按照引导参数，内核将在SDRAM的特殊位置寻找一个RAM磁盘。找到后，将RAM磁盘解压到SDRAM的另一个位置上，并且把它“安装”成一个磁盘镜像。然后按照Linux常规启动规则开始执行这个镜像中的应用代码，即一个初始化程序。

之前经过压缩的内核程序和解压缩程序会从SDRAM中被删除以释放空间。压缩过的RAM磁盘也会从SDRAM中被删去，因为它现在已经在存储器中了。

SDRAM的剩余空间将会存放代码执行过程中所产生的临时变量和数据。如图7-17所示。

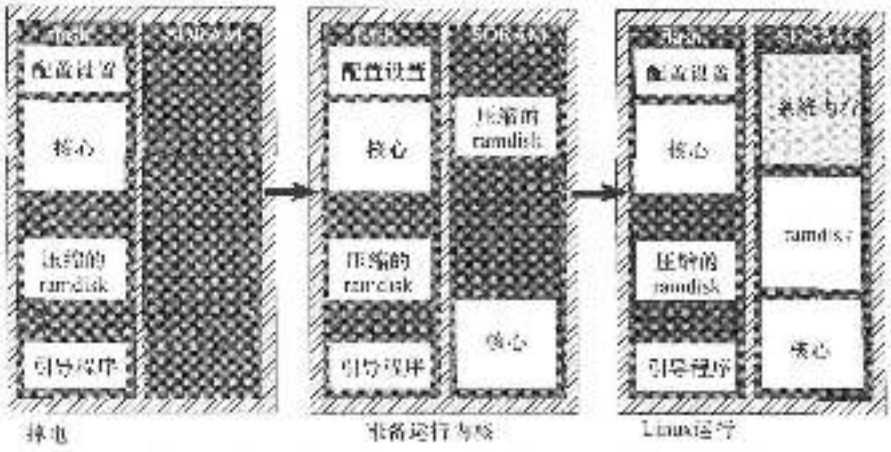


图7-17 嵌入式ARM平台上存储器内容，包括闪存和SDRAM，分别是在掉电阶段、引导阶段和嵌入式Linux运行阶段

一个很小的嵌入式处理器中的存储器结构，比如TI的MSP430x1，由于缺少外部的数据或地址总线，具有较低的可配置性。然而，这个处理器也有很多外设和大小不一的存储器可供选择（参7.4提供了更多信息）。

7.8.3 其他存储器

很多设备都有一个并行接口，使其能够加入到CPU存储映射中。其中包括一些外置设备，比如存储器、以太网芯片和硬盘接口，也有一些内部设备的接口，比如SoC上集成的外设。

然而，还有另外一类常见的实体也可以加入到存储映射中，这就是系统和外设控制寄存器。参7.4介绍了MSP430的存储映射（在存储映射的底部，从特殊功能寄存器开始，接着是外设控制寄存器）。事实上，如果返回头去看，查看图7.2中给出的MSP430控制管脚描述，可以发现一些我们在这里所讲的寄存器。

图7.4 MSP430中的存储映射

MSP430是一个大型的拥有大量内部功能的小型低功耗微控制器，这些功能大部分通过映射到有管空间上的片上集成外设实现。

如下图表示是MSP430x1xxx系列处理器的存储映射图，底部为0地址，图中的不同部分有不同的宽度，有些是8位宽可存储是16位宽。因为该部件中存在大量不同宽度被挂的外设，因此TI能够使写被挂宽度划分存储映射空间。例如，设置寄存器和被挂寄存器是一个8位的外设，位于地址0x10-0x0FF之间。



在图的底部是特殊功能寄存器，用于控制整个系统、处理器以及其他部分（如功耗和时钟）。处理器的中断向量表在映射的顶部，这意味着一旦设备重启，将从这部分读取指令开始执行。出于这个原因，包含引导程序的非易失存储器一般都会放置在存储映射的顶部。RAM位于较低位置。

有趣的是，TI提供了很多种类的MSP设备供选择，每个设备都有不同的可选特征、外设以及不同规模的闪存/ROM和内部RAM。它们的存储映射相同，RAM的上边界和ROM的下边界除外，它们的位置视各自的具体规模而定。

329
~
330

所有这些寄存器，在MSP430数据手册中都有详细说明，它们都可以在存储映射中找到，这也意味着它们将占据具体的处理器存储器地址。对于框7.2中提到的特定寄存器来说，这些寄存器可以在右面这些地址中找到。

名称	地址
P2DIR	0x02A
P2IN	0x028
P2OUT	0x029
P2SEL	0x02E

这样，通过对这些地址进行读写就可以控制或者查询这些寄存器。

对于我们查看的这几个寄存器，返回头去看一下存储映射图，可以发现它们在8位外设模块部分，这正是我们所期待的，因为端口（以及相应的控制寄存器）宽度为8位。

如果用C语言进行编程，那么读写这些寄存器最安全的代码如下：

```
unsigned char read_result;
void *addr;
read_result = *((volatile unsigned char *) addr); //读
*((volatile unsigned char *) addr) = 0xFF;              // 写
```

为什么要使用volatile关键字？

许多编译器在进行编译时如果发现在一个程序中有对同一个地址进行连续两次写入操作时，会删除掉第一个写入以提高程序效率。例如，如果一个程序想要在X位置上存储一个值，然后在几个时钟周期后再在相同位置上写入，且两个写入操作之间没有读操作，显然第一个写入操作是浪费时间——不管第一次写入了什么内容都会被后面的写操作所覆盖。

对RAM进行写入时是这样的。然而，有些情况下，我们需要合法地对同一地址进行重复写入操作，比如在一个闪存上改写算法或者写入地址对应的是一个有存储映射的寄存器。

这里的一个例子就是串口单元的输出数据寄存器。一个程序员想要设置好串口然后连续输出2字节的数据，这就需要一个字节接一个字节地写入有存储映射的串口输出寄存器。

这时volatile关键字会告诉编译器这个写入是“易失的”，也就是说需要刷新。这样编译器会让这个连续写入完成，不删掉前一个写入。

编译器并不是只能发现连续写操作，也可以发现连续读操作，并且尽可能地优化成一个读操作。

连续读出现在代码中也很正常，事实上，编译器会插入读操作作为泄漏代码的一部分（3.4.4节）。然而，对于程序设计者而言这部分代码并非设计初衷。

当然，正如上文所述，连续读和连续写一样必要。例如，在串口输入寄存器中读出一组连续
331 的数据，或者轮询串口状态寄存器以检测发送缓存是否为空。以上这些情况，正如连续写入一样，volatile关键字会告诉编译器这种连续读是故意设计的，不可优化掉。

与上面类似，也可以使用volatile定义一个易失性变量类型：

```
volatile unsigned char * pointer;
```

7.9 测试和验证

测试和验证会覆盖所有的章节，旨在解决计算中存在的实际问题，尤其是在嵌入式计算中。引入测试和验证的主要原因是计算性能的提高使得处理器变得越来越复杂和庞大。但这也会让处理器

的设计和制造变得更困难。另外，测试和验证会导致需要在设备中加入测试支持和错误控制机制。

7.9.1 集成电路设计和制造问题

在20世纪70年代，工程师一个人不可能了解和检测一个完整的现代处理器。虽然良好的团队合作和卓越的设计工具大量替代了手工检测，但是在集成电路（IC）设计上依然容易出现错误。事实上，我们在处理器第一版发布时不能保证完全找不到硬件设计错误。大部分错误只是小麻烦，它们可以简单地通过软件设置来修复，比如当中口工作时，在模式发生变化后总是需要加入一条NOP指令，而其他的错误可能是更严重的。

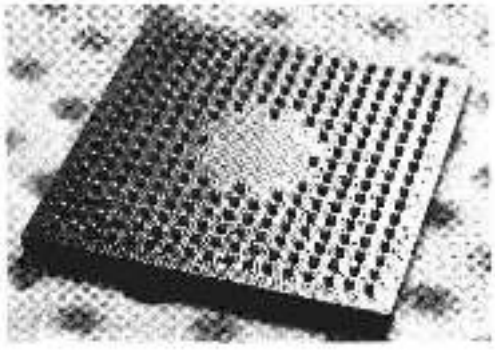


图7-18 ARM处理器底面的球栅阵列（器件尺寸为14mm×14mm）

Intel 80486处理器的FDIV错误是一个比较著名的设计错误，幸亏发现它的时候卖出和安装进计算机的处理器只有几千个而不是上百万个。这虽然只是一个百万分之一的错误，可以保证在几个月中不被检查出来，但是会给公司在经济上和公共关系上带来沉重的代价。

制造过程中的错误更为常见。粗略对比一下现在顶级PC中的PCB板和在20世纪80年代所设计的同类产品，不仅可以看到在硅片中集成各种分立部件是一个循序渐进、不断整合的过程，还能发现现在的电路板设计带有大量的管脚和球栅阵列。如图7-18所示的就是三星S3C2410 ARM处理器底面的BGA（Ball Grid Array，球栅阵列）照片。可以看到当加热焊锡倒焊接相对应的PCB板表面焊盘时，这些小的焊接锡点就会熔化。

BGA通过很紧凑有效的方法连接IC和PCB，但这使得调试和修复变得困难了。基于之前几代的IC封装技术，可以对在顶部可见的外部成簇连接点进行探测。而BGA技术将连接点隐藏它的底层，几乎只能通过X光才能检查已安置在PCB上的每一个连接点。图7-19展示的就是通过X光所看到的一个封装效果，在这里IC的内部结构和IC下面PCB的细节都是可见的。

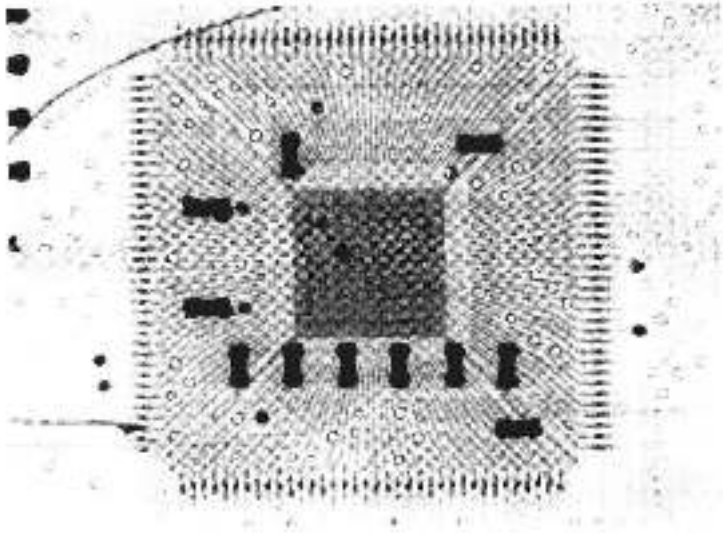


图7-19 IC（方形扁平封装）的X光照片，可以看到IC封装、引线框和内部硅片

诸多的设计功能集成在单个硅片设备上，例如在现代个人计算机上的SuperIO芯片。因此需要集成原有设备使用的外部接口，例如鼠标类似装置、光盘，软盘驱动器之类的存储媒介，这也是影响器件大小的主要因素。使用多管脚的设计已经出现大概30年了。而USB、Firewire和SATA（串行ATA）等现代接口技术使用较少的管脚也非偶然。这样导致大多数I/O连接器支持传统的接口而非支持这种现代接口。因此看起来旧的ISA、EISA、IDE、SCSI和软盘等接口会最终消失，从而产生更小的SuperIO[⊖]芯片。

现在同样需要测试和验证。具有大量I/O连接的芯片测试主要存在两个问题。

首先，生产IC时一般需要经过测试。有些制造商倾向于只使用批处理测试，而有些对于错误零容忍，会测试每个器件。这些测试需要覆盖两个主要的制造步骤：生产硅芯片和在其上附加管脚/锡球。

其次，还有一小概率事件——每个焊接点是否工作。电路板制造缺陷率大约与焊接的管脚数成正比。因此拥有大量I/O管脚的装备就会更容易出问题，需要一种验证方法判断这些管脚焊接点是否正确。

为清晰起见，我们将这些技术分为两类，并在下面分小节讨论计算机处理器测试中所用的解决方法。

1. 设备制造测试——用，在设备离开半导体制造厂前确保集成电路能正常工作。更多详情参考以下章节：

- 7.9.2节，BIST（Built In Self-Test，内置自测）
- 7.9.3节，JTAG（Joint Test Action Group，联合测试行动小组）

2. 运行时测试和管理——这是为了保证最终制造的系统能正常工作（涉及之前的两种技术：BIST和JTAG）将在以下章节中探讨：

- 7.10节，EDAC（Error Detection and Correction，错误检测与纠正）
- 7.11节，看门狗定时器与复位监测

7.9.2 内置自测（BIST）

BIST是特定于器件的片上硬件资源，主要用于协助测试器件内部功能。举个例子，当标贴片离开硅刻生产线或者单个IC已经被安装待出售时，测试机器会使用BIST。有时候，允许客户访问内部的BIST单元来验证自己的设计。

334

BIST单元可以在某种程度上调查待测试的IC部分，向该部分输入已知值和状态，然后检查这部分输出是否正确。这些可以概略地表示在图7-20中。在测试模式下数据通过多路选通器从BIST单元出来或者进入BIST单元。

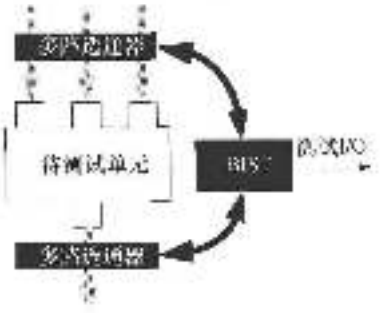


图7-20 内置自测单元可以测试某些器件的输入/输出，使得在给定输入下验证输出的正确性

BIST还可以用于CPU测试各种外设单元的内部程序。这种情况下往往需要其他方法验证外设单元的功能正确性。例如需要一个时钟。BIST单元可以完成这个功能，如图7-21所示，其中在测试模式下多路选通器将模拟输出信号反馈到外设输入端口。

⊖ SuperIO是一种大的IC的名字，这种IC通过PC传统外设接口提供功能和系统功能上辅助CPU工作，例如，内存映射，USB接口，并口以及串口等。

反馈外部信号意味着制造商可以生成一个测试序列，通过模拟输出驱动器（例如EIA232串口包括一个负电压信号电平）输出，然后通过模拟输入，从而验证串口硬件、输出驱动或缓冲以及输入检测器。 335

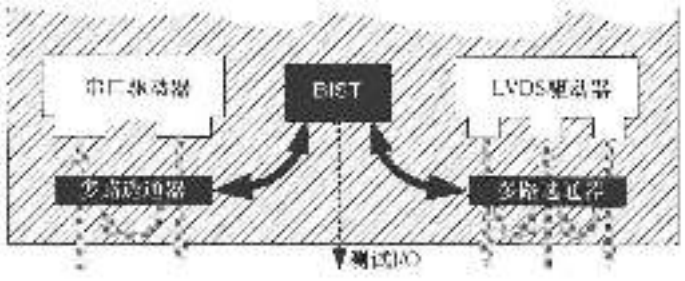


图7-21 内置自测单元可以用于测试或设置兴趣于某单元外部管脚及其内部逻辑之间的输入/输出信号

片上测试方法能简单、方便地测试IC上所有逻辑和许多模拟单元，但是这会增加硅片面积和复杂度。这来源于三个部分的开销：

1. BIST单元本身。

2. 每个被测单元和I/O端口需要多路选通器或类似的开关电路。

3. 从BIST单元到每个选通器的开关和数据连线。

BIST单元并不是很复杂，容易扩展到更大规模的设计中。对于大多数逻辑实体来说，增加输入和输出选通器并不会明显增加逻辑大小。然而，从BIST到每个测试点的数据和开关是一个问题。测试数据通路时可能需要操作在同一个时钟频率上，并需要一组并行线路连接输入和输出总线。这些线路（或者硅片IC上的金属/多晶硅通道）必须从芯片的各个部分连接到集中式的BIST单元上。这种布线使得IC设计非常困难且显著增加了代价。将BIST电路化分成许多更小的单元有助于解决该问题，但是，IC设计复杂性增加并且BIST复杂性也同时增加的问题依然存在。

一个解决因规模产生的问题的方法是使用一系列的"扫描通路"，其中多路选通器之间的连接是串行的，而多路选通器本身只是简单的并行/串行的寄存器，如图7-22所示。

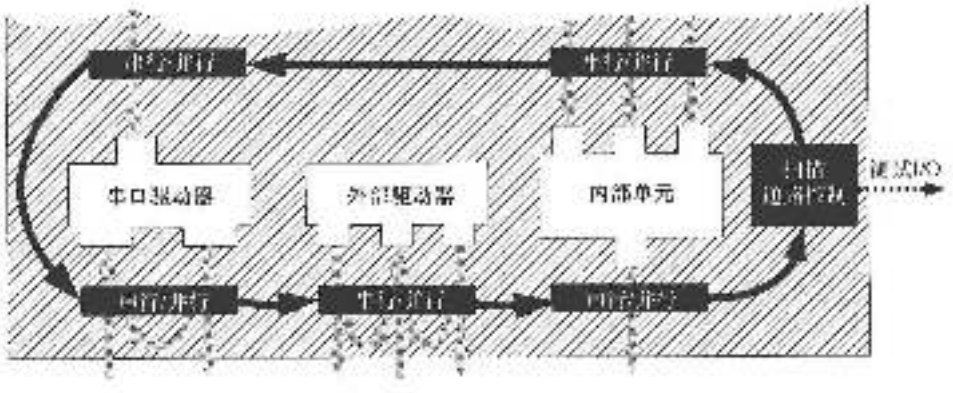


图7-22 一个待测试的扫描链，利用串-并转换逻辑隔离被测部分

可以看到扫描通路控制单元和所有测试点之间通过一条骨互联，称为扫描链。其长度取决于该链上所有串行/并行寄存器的位数量。扫描链本质上是一条高速串行总线，包括时钟，数据和控制线。最重要的是，很容易环绕IC布线，而BIST单元（或扫描通路控制器）可以放置在芯 336

并不是面向总线的。

7.9.3 联合测试行动小组 (JTAG)

联合测试行动小组（Joint Test Action Group，JTAG）是IEEE的一个工作组，目前已经发展为IEEE标准1149。现已在各种逻辑器件中广泛用于扫描链控制单元。IEEE1149最初是用于边界扫描测试，边界扫描测试是扫描链路的一种，用以测试器件的外部输入和输出，而不是用于内部单元的I/O。

事实上，近些年来遵循JTAG的测试单元已经集成了很多实质性的上校功能，并且目前包含了除边界扫描链路之外的各种内部存取功能，在很多场合，JTAG是硬件调试器存取目标处理器的方法。一共很先进的处理器既有测试用JTAG单元又有内置仿真器（in-circuit emulator，ICE）用JTAG单元，后者通常用于调试。

JTAG定义了一个标准测试接口，包括以下器件上的外部信号：

1. TCK（测试时钟）
2. TMS（测试模式选择）
3. TDI（测试数据输入）
4. TDO（测试数据输出）
5. TRST（可选的测试复位）

实现ICE功能的JTAG单元，通常会有4个或者8个的输入/输出信号来组成一个高速总线以快速传送调试数据。

我们来看最原始的JTAG。图7-23给出了JTAG在类ARM处理器中的硬件实现。

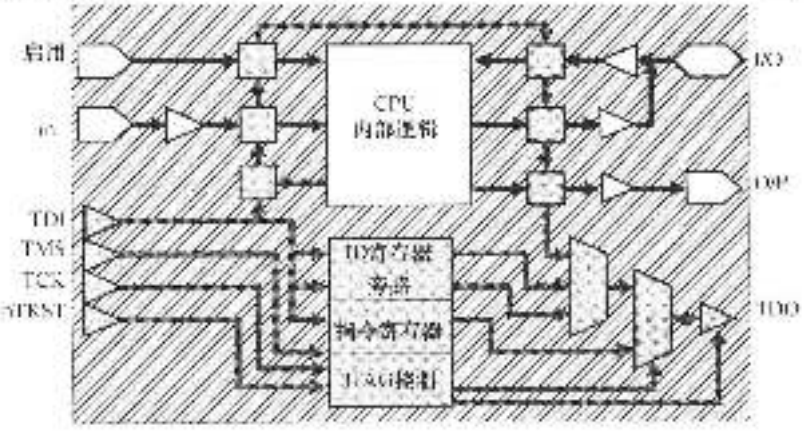


图7-23 与JTAG相关的主要寄存器被附加类ARM处理器中串行数据寄存器立述举例

JTAG电路 并不是完全按照图7-23的比例实现——位于图中CPU内部逻辑的下面，且作为边界扫描与这个模块的所有输入/输出相连。使用5个JTAG管脚，所有连接到CPU内部逻辑上的输入、输出和双向管脚都可以被存储到根据需要作适当调整。

JTAG在许多方面都是很有用的，比如跟踪连续问题焊接问题（见框7.5）。另外一个常见应用（这并不是JTAG设计者的初始考虑）就是向嵌入式系统的入存中下载引导程序，见框7.6。

框7.5 采用JTAG寻找焊接错误

想象一下你拥有一块新的计算机主板。一切看上去都很好：没有短路或错误，板载存储器芯片测试正常，但是主板就是不工作。哪里出错了？

使用JTAG就连CPU，测试人员可以对板载外设进行检查。特别，他们可以测试PCB测试的信号

输出是否正确。他可能设置地址总线为0xAAAA（二进制为1010101010101010），这样可以测试这些管脚有没有互相短路，然后设置为0x5555（二进制为0101010101010101），这样每个管脚上的信号都翻转，从而可以发现不能正常驱动高或低电平的管脚。检查管脚的这两个状态是必要的，因为PCB上有些信号不驱动时为高，有些则会为低。

而后，测试人员可能在PCB中设置已知值到各测试点上，然后通过JTAG读回CPU上所有输入管脚的状态。然后更换为其他值，比如这些值的反转，并重复这个步骤。

这样，CPU上所有的输入、输出以及双向信号都可以被检测到。如果CPU的某个管脚或者锡球没有焊接上，这样就会看到CPU没有驱动这个信号，或者CPU输入错误。

但这种方法有它的局限性。首先，测试只能告诉我们焊接点是否工作，但并不能说明它们的工作质量（这可以帮我们预知未来可能出现的潜在错误）。其次，有些管脚无法测试，例如电源管脚、模拟电路I/O管脚以及典型的锁相环输入管脚。最后，测试过程太慢了。

框7.6 使用JTAG引导CPU

大多数基于ARM体系结构的处理器没有内部闪存，一般复位后从0号地址开始执行。这个地址和与外部闪存相连的片选信号0（nCS0表示低电平有效）有关联。

因此外部闪存包含一个引导加载程序，它是CPU在复位或者启动时首先执行的程序。该程序会引导到主应用程序或者操作系统，比如用于智能手机的移动Linux或用于基本手机的SymbianOS等。

在20世纪90年代之前，引导代码在EPROM（可擦除可编程只读存储器）中。只需简单地插入一个可编程EPROM设备，开启电源，系统就可以工作。由于ROM被认为太昂贵、尺寸较大，因此EPROM已经逐渐被可重复编程的闪存所取代。

每个新设备离开生产线时闪存里面没有数据，因此需要将引导代码写入到闪存中。

这可通过基于JTAG的程序完成。这需要一台外部PC连接到CPU JTAG控制器。PC机控制连接到CPU管脚上的闪存，驱动闪存将引导程序写入。对于闪存来说，它不知道是外部PC在对其进行控制，而简单地认为是CPU的一种常规控制。

外部PC通过JTAG到CPU进行工作，控制CPU端口，使用命令让外部闪存擦除它本身，然后按字节将引导程序从地址0写入闪存。

337～338

JTAG的控制以简单状态机实现。数据在TCK的下降沿从TDI管脚输入。TMS管脚用来选择和改变模式。有几个模式可供选择，通常都包含BYPASS功能来跳过扫描链，这样可以保证从TDI输入的数据直接从TDO输出。IDCODE输出ID寄存器的内容来标识设备生产商。EXTEST和INTEST通过扫描链输出数据，分别支持检测外部和内部的链接。

生产商可以在一个器件内实现几种扫描链。一个典型例子就是与CPU位于同一芯片中的闪存有一个单独的扫描链，独立于主CPU的扫描链，但它们共用同一个JTAG接口。

典型的扫描链有数百位长。例如三星的S3C2410 ARM9处理器是272管脚的BGA，有扫描链427位，其中每位对应的管脚为：

- 输入管脚
- 输出管脚
- 双向管脚
- 控制管脚
- 保留或者隐藏管脚

339

通常，输出和双向管脚都相应有一个控制位，以确定输出缓存是关闭还是开启。这些控制位可以是高电平有效或者低电平有效，它们和其他控制JTAG所需的信息都存放在边界扫描数据（BSD或BSDL）文件中，包括扫描链长度、命令寄存器长度、实际命令字和扫描链到管脚或功能的映射关系。

最后，应该注意的是由于JTAG标准以串行连接实现，因此可以使单个JTAG接口服务于扫

描链上的不同设备。一个外部测试控制器可以通过JTAG接口寻址并测试每一个设备。

JTAG是一种非常高效的硬件资源，因此在CPU、FPGA、显卡、网络控制器和配置器件中应用非常广泛。任何还记得JTAG出现之前调试数字硬件之困难的人都会同意作者的观点，即JTAG技术虽然简单，但使得计算机设计者的原型设计能力发生了革命性的变化。

7.10 错误检测和纠正

除了错误的编程以外，数字系统中的错误也可以通过其他途径产生。在糟糕的系统设计中，我们可以看到以下问题：模拟噪声干扰数字线路，电源线上产生电压急降（也称为“掉电”，在7.11.1节中将会介绍），时钟抖动（见7.4节）会造成在错误的时间采样数字信号，来自其他设备的电磁干扰也会对信号造成影响。

一个很少被提及的错误产生来源是宇宙辐射——所谓的SEU（Single Event Upset，单粒子翻转）是指在电子设备中由宇宙射线触发的随机位翻转。由于地球的大气层会削弱宇宙辐射和太阳辐射，因此SEU会随着海拔的升高而变得明显。家用电子产品在全球定位卫星所处的高度（约20 000km）上将会完全不可用。然而，在一个较低的地球轨道高度（500km）它们可能每天都会遭受一些SEU。在高山上，这样的事件每个月会发生一到两次。在地面，可能一年才会发生几次。这个错误源在日常生活中听起来并不值得我们担忧，但是还是会有明显受此因素影响的一些设计，比如，将会被用在空中交通控制系统、核反应堆控制室或宇航员生命维持系统中的计算机。

幸运的是存在完善的技术来解决以上错误，并且这样的技术在太空科学中是一个活跃的研究领域。常用的技术范围很广，从类似的NASA决策到并行运行5台单独的计算机，并通过“多数票决”制来决策，更简单的例子是存储总线上的奇偶校验。

随着时代的进步，像DEC、Sun和IBM这样的著名公司设计的UNIX工作站都设计成为能与
340 带奇偶校验的存储器互相传输信息。这样的存储器中，每个字节以9位方式存储，或者32位数据以36位形式存储，每个字节中多出来的一位作为奇偶校验位：

7	6	5	4	3	2	1	0	P
1	0	1	1	0	1	0	1	**1**
1	0	1	0	0	1	0	1	**0**

如果这个字节中“1”的个数是奇数，P位的值就是1，否则为0。如果单个位产生了错误，与字节内容相比较奇偶校验位就会是错误的，因此它有可能检测出是否有单个位的错误（比如由SEU引起的）。即使奇偶校验位恰好就是受SEU影响的那一位，这个奇偶校验方法同样可以应用。

然而，单个位的奇偶校验并不能检查出字节中的两位错误。更糟糕的是，尽管我们知道发生了一个错误，但是我们并不知道具体是哪个位产生了错误，所以无法纠正这个错误。

更多的错误检测方法中用到了汉明码和Reed-Solomon编码方法。一种近来非常流行并且比较先进的技术是功能强大的Turbo码，经常用在卫星通信中。这些方法的细节超出了本书讨论范围，在这里，我们只需知道所有这些方法增加了必须处理的数据量，但提高了修复受损数据的能力。实际上，在设计中需要以下多方面的权衡：

- 编码的复杂度——需要多少MIPS编码一个数据流
- 解码的复杂度——需要多少MIPS解码一个数据流
- 编码开销——需要多少额外的数据位加到数据之中
- 纠错能力——能纠正多少位错误数据
- 检错能力——能检测出多少位错误数据

我们可以考虑每一种设计要素，每个设计都有它独有的特点。而且，要考虑设计所基于的数据单元——可能是单个字节（有重复的编码），也可能是几个KB，甚至更大（Turbo码）。我们必须就不同情况做实际考虑：一些设计会在处理几个数据位之后就输出经过纠正的数据，而一些设计会在处理完一个庞大的数据块后再进行解码。

一些例子如下：

- 三重冗余——有时称为重复编码。在这种编码设计中，数据的每一位重复了三次，所以编码开销是300%。因此一个错误可以在每3位中被纠正。编码和解码极其容易。三重模块冗余（TMR）是其中一个例子，将3个或更多个模块的输出由"多数票块"制决定输出，如图7-24所示。被决策的信号不是必须为数据位，有可能是字节，字或者更大的数据块。我们可以对单独的每个位或者整个项目的输出数据进行"投票"。其中一个例子是这样的：NASA的航天飞机拥有5台IBM用于航天的计算机，其中4台运行相同的代码并且支持多数票块。第五台运行和其他机器相同任务的软件，但是被单独开发和编写（这样就不会受到其他计算机相同的软件错误的影响）。 341

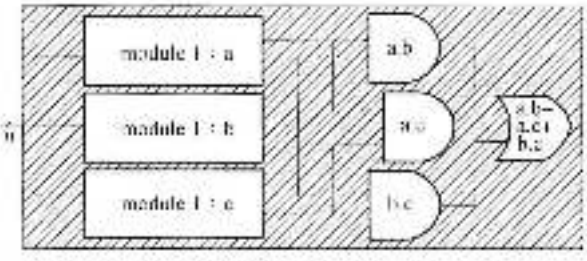


图7-24 一个TMR例子——一个处理模块被重复了三次。一个简单的输出电路演示了"多数票决"制。例如，如果三个模块的输出分别是位级别的0，0，1，那么最后的输出就是0，我们会认为输出为1的模块是错误的。以此类推，如果三个模块的输出分别是1，0，1，那么最后的输出是1，我们会认为中间输出0的模块是错误的。注意：信号不一定必须是位，有可能是更大的数据块

- 汉明码——一个非常流行的代码形式，通常形式为（7，4）编码，即为每4位的数据添加3位的校验位数据。它可以纠正每个数据块里的单位错误，并且还可以检测出每个数据块中2位的数据错误。编码和解码都是相对比较简单的——要求对1、0组成的矩阵所代表的数字进行简单的模2算术[㊀]。从框7.7和框7.8中可知（7，4）代码的编码开销是75%。注意：汉明码存在许多不同的形式，从而有不同的代码开销以及检错和纠错特点。

框7.7 汉明（7，4）编码举例

对于一个将要传输的4位数据字，它的各个数据位分别是b_0、b_1、b_2、b_3。我们利用模2算术来定义从p_0到p_2的4个奇偶校验位：

$$p_0 = b_1 + b_2 + b_3$$
$$p_1 = b_0 + b_2 + b_3$$
$$p_2 = b_0 + b_1 + b_2$$
$$p_3 = b_0 + b_1 + b_3$$

实际上我们要传输的7位的数据字是由4位的初始数据和3个奇偶校验位组成的，如下所示：

b_0	b_1	b_2	b_3	p_0	p_1	p_2

㊀ 模2算术用0和1来标记数值。它表示任何大于1的数除以2的余数。这样，偶数模2后用0来表示，奇数模2后用1来表示。例如，3 = 1(mod 2)，26 = 0(mod 2)。以此类推，任何数的改a算法就是它们除以a后所得的余数。

当接收到这个7位的数据字时，我们很容易重新计算3个奇偶校验位并且检测它们是否正确。它意味着数据或者被正确地接收，或者存在着大于1位的数据错误。如果检测到一个错误，我们可以确切地知道（假设只是单位的数据错误）哪个数据位是错误的。例如，如果检测到 p_1 和 p_2 是错误的，但是 p_0 是正确的，那么数据位 b_0 或 b_3 有可能产生了错误。然而，b_3 是用来计算 p_0 的，而 p_0 又是正确的，因而我们可以确定只有 b_0 位发生了错误。

汉明码（或者其他编码形式）更常用矩阵举例见框7.8。

框7.8 利用矩阵的汉明（7，4）编码实例

实际上，汉明编码、验认和纠错是利用线性代数（矩阵）来实现的。汉明将 $\boldsymbol{G}$ 定义为生成矩阵，$\boldsymbol{H}$ 定义为奇偶校验矩阵：

$$\boldsymbol{G}=\begin{bmatrix}1&1&0&1\\1&0&1&1\\1&0&0&0\\0&1&1&1\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\qquad \boldsymbol{H}=\begin{bmatrix}1&0&1&0&1&0&1\\0&1&1&0&0&1&1\\0&0&0&1&1&1&1\end{bmatrix}$$

我们用一个4位的数据向量来证明。首先，向量 $\boldsymbol{d}$（1101）与生成矩阵 $\boldsymbol{G}$ 做乘法，从而生成7位的要传输的数据字：

$$\boldsymbol{x}=\boldsymbol{Gd}=\begin{bmatrix}1&1&0&1\\1&0&1&1\\1&0&0&0\\0&1&1&1\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1\\1\\0\\1\end{bmatrix}=\begin{bmatrix}3\\2\\1\\2\\1\\0\\1\end{bmatrix}\ \text{modulo 2}\ =>\ \begin{bmatrix}1\\0\\1\\0\\1\\0\\1\end{bmatrix}$$

从而用欲传输的数据1010101代表原始的数据1101。假设存在一个单位数据错误，我们接收到一个不同的数据：$\boldsymbol{y}=1000101$。下面来看一看我们如何用矩阵 $\boldsymbol{H}$ 来检查接收到的数据：

$$\boldsymbol{Hy}=\begin{bmatrix}1&0&1&0&1&0&1\\0&1&1&0&0&1&1\\0&0&0&1&1&1&1\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\\1\\0\\1\end{bmatrix}=\begin{bmatrix}3\\1\\2\end{bmatrix}\ \text{modulo 2}\ =>\ \begin{bmatrix}1\\1\\0\end{bmatrix}$$

通过查看奇偶校验矩阵 $\boldsymbol{H}$，我们看到［110］位于第3列，那么，我们接收到的 $\boldsymbol{y}$ 的第三个数据位就是错误的。通过比较 $\boldsymbol{x}$ 和 $\boldsymbol{y}$，我们发现确实如此。将第三个数据位翻转过来，由0变为1就纠正了 $\boldsymbol{y}$ 并且重新构造了原始的数据信息。

- **Reed-Solomon（RS）**——一种基于数据块的代码形式，具有相对较低的编码复杂度和较高的解码复杂度。RS实际上是一种基于数据块大小的代码形式——它的检错和纠错能力是由所处理的数据块的大小设定的。一种常用的代码形式是RS(255，223)，用于处理具有255字节的数据块。经过编码后的数据块包含223字节的数据和32字节校验码数据，并且在每个223字节的数据块中，可以纠正16字节的错误数据。注意：这些字节可能有多种多样的错误，所以有时可能会纠正多于16个的单位错误。对于RS(255，223)，

编码长度的开销是223个字节需增加32个字节，即14%。

一些CPU（例如欧洲太空总署中所用的SPARC处理器，被称为ERC32——也可以叫做Leon软核）自身就嵌入了EDAC（错误检测和纠正）的能力，其他的一些CPU则依赖于外部的EDAC单元，如图7-25所示。

在图7-25中，数据总线上的数据和连接到CPU的数据并不受EDAC保护，我们需要一个外部EDAC设备给每个从CPU写出到存储器的数据字添加纠错代码，并且检测每个从存储器读入CPU的数据字。一旦检测到一个不可修复的错误，则启动一个中断来通知CPU。而那些可修复的错误无需CPU干预便可自动修复。

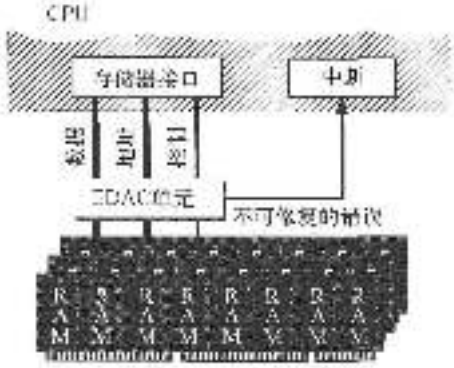


图7-25 EDAC单元位于CPU存储器接口和外部存储器之间

342 ~ 344

注意：并不是所有纠错代码都可以迅速地在CPU和存储器之间进行操作。例如，Reed-Solomon编码的数据字替要相对比较长的一段时间来解码，每次检测到错误并且纠正的过程很可能导致CPU的暂时停顿。相比之下，汉明码是比较快捷的，经常在系统中被用做错误检测和纠正（见图7.7和图7.8中汉明编码的例子）。

总之，要求一些计算机系统高度可靠，因而错误检测和纠正单元就是必须具备的，或者是内部嵌入的单元，或者是在容易受噪声影响的外部数据线上。类似地，那些更容易受SEU错误影响的高密度存储也需要在EDAC单元的保护下才能正常工作。

7.11 看门狗定时器和复位监测

除了直接的奇偶校验，EDAC在基于地面的计算机中很少使用，但看门狗定时器和掉电检测器（见7.11.1节）却极其常见，它们通常在专用CPU集成电路中实现。

一个看门狗定时器对于处理器来说就像是起搏器对于人的心脏一样：它需要不断地重新确定处理器在正确地执行程序。如果一段时间没有进行重新确定，看门狗定时器就可以确定这个处理器已经“挂”了，并且会触发复位键——就像是起搏器给已经停止跳动的心脏传输一次小小的电击一样。

从编程者的角度来看，处理器必须反复在超时时间段内写入或读出看门狗定时器（WDT）。它会不断地写入或读出，但在一段特定的时间内如果读写失败，就会引起复位。

内置的WDT会允许编程人员通过写入内部的配置寄存器来设定超时时间，这个配置寄存器通常有存储映射。看门狗的构造是一个倒数计时器，通过输入分频系统时钟具体实现，分频比率在很多情况下也是可以设置的。系统复位时，看门狗计时器的配置寄存器中的值会被载入到一个硬件计数器中。系统没有复位的时候，这个计数器会随时钟递减，比较器会监督它何时减为零，此时将触发复位信号。一旦CPU读出或写入数据到WDT寄存器中，硬件计数器会将看门狗计时器的配置寄存器的数值重新载入。

一个外部的看门狗定时器可以由一个电容、电阻和一个比较器组成。这样的看门狗定时器和外部复位电路（见7.11.1节）的工作方式类似，只是CPU要定期地向电容“写入”一个逻辑“高”以保持它的充电状态，从而呈现复位状态。

典型的看门狗计时时间是几百毫秒或可能是几秒——如果时间太短将会意味着过多的CPU周期浪费在周期性存取WDT的代码上。解决这个问题的最好方法就是周期性地执行一些底层代码，比如每100ms执行一次的操作系统（OS）定时器程序。如果定时器停止，我们就可以断定 345

系统已经崩溃了，结果看门狗会重启处理器。看门狗定时器因此要确保系统时刻处在运行之中，否则，它会重启CPU并且重新执行OS程序。

复位监测器和掉电检测器

计算机行业的许多资深人士也许还记得早期IBM PC上的“大红色开关”和机器上很显眼的复位按钮。这些明显的按钮可能是一种运行在机器上的操作系统（即MS-DOS和Microsoft Windows）的可靠性的体现。幸运的是我们不再用MS-DOS了，但是仍然在用Windows——尽管它一般不会被用于那些可靠性要求很高的“任务至关重要”的应用中。

相比之下，嵌入式系统就不会有大的复位按钮，它经常用“软”开关而不是“硬”开关（即那些处于软件控制之下的“开关”，而不是在硬件上直接控制电源系统的开关）。嵌入式系统经常被认为是更加可靠的，尤其在远程操作方面。例如，在火星探测器上的“大红色开关”是不现实的。

在提高系统可靠性的要求方面，嵌入式系统趋向于广泛地利用看门狗定时器。它还有电源和复位监测电路。

复位电路，经常由一个外部的复位输入驱动，对于确保一个设备在已知状态下开始操作是非常重要的。不管是在CPU、SoC、FPGA中，还是在分立的硬件系统中，缺少复位清零信号是许多系统错误的起因。

一个外部的复位控制设备，或者监测IC如图7-26所示，一旦系统上电就会“启动”复位信号。一段时间后，复位信号撤销，就会允许设备从一个已知的位置开始运行。一些SoC处理器在内部会包括所有的复位逻辑和时序电路。其他设备可能会允许设计者简单地将复位管脚接到和GND相连的电容上，和一个与Vcc相连的电阻上，但是在许多情况下这样做是非常危险的，所以一定要非常注意。[⊖]

346

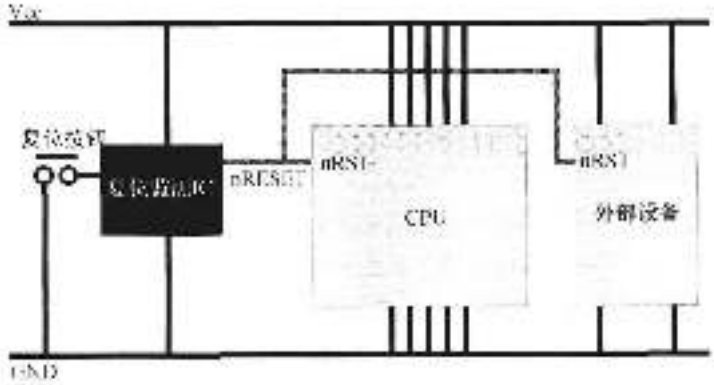


图7-26 位于Vcc和地（GND）之间的复位监测IC——用于为CPU和任何需要它的外部设备产生低电平有效的nRESET信号。按照约定，复位信号是低电平有效的，以确保第一次上电时设备处于复位状态。如果设计中要求有复位按钮，那么这也可以作为复位监测器的一个输入。

⊖ 这种做法危险的原因在于复位信号被破坏的方式。系统一上电，就过电容的电压初始为0，意味着复位管脚为低电平。由于电容通过与Vcc相连的电阻慢慢充电，电压会一直上升并达到复位输入管脚的阈值。这时，它会作为逻辑“高”驱动复位输入信号，从而使设备从复位中退出。然而，不幸的是，在任何系统中总会存在电噪声，引起电压小范围波动，使电容电压和复位管脚阈值，导致设备快速地进入和退出复位，被乱复位动作。这促使大多数制造商不得不规定一个他们的设备保持在复位状态的最小时间。

掉电是指在电压线[⊖]上的一次电压下降。由于规定 CPU 只可以在很小的电压变动范围内运行，因此当掉电发生时会引起故障。如果完全掉电，外部复位芯片会启动复位状态（即一旦电源恢复，它们会在制造商规定的时间内将 CPU 保持在复位状态）。然而，当供电电压不在规定的电压范围内时，只有拥有掉电检测器的复位芯片才会做同样的事情。

另外，一些带掉电检测的复位芯片可以给处理器提供即时的掉电中断，使处理器在完全掉电的几毫秒之前及时处理好该做的事情，然后“干净”地断电。复位监测（reset supervision）和掉电检测的处理如图 7-27 所示，提供给处理器的 Vcc 电压值随时间的变化在图中绘出。设备的工作电压是 3.3V ±5%，因而复位监测系统被设置为检测任何超出这个范围的 Vcc 电压偏移。当检测到电压偏移时，系统会触发复位状态。复位状态会在每次发生时保持 10ms（实际上，它经常被设定为超出处理器制造商所规定的时间，规定时间一般远小于 10ms）。这时会像如图 7-26 所示的标准复位监测 IC 一样连接和使用掉电检测设备。 347

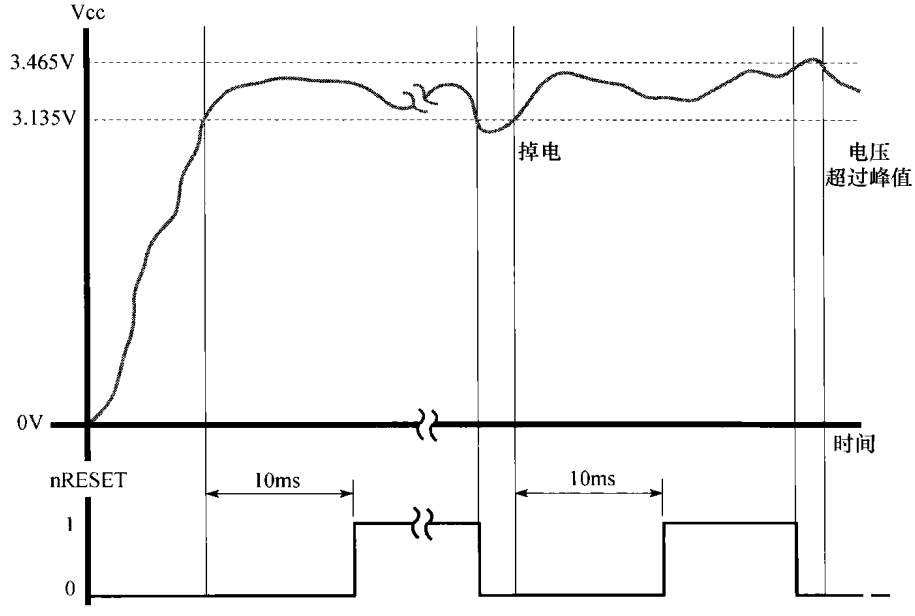


图 7-27　一个复位监测芯片的例子。当提供的电源（上面的轴线）上升到额定的 3.3V 时会引起复位（如处于下面轴线的 nRESET 信号所示）经过一段时间的正常操作，当电压值下降时掉电发生了。监测芯片必然会将处理器干净地复位直到电压又上升到额定电压。稍后，电压值超过峰值的情况开始出现，在这种情况下也会做类似的处理

7.12　逆向工程

随着嵌入式技术的发展，消费者看到了令人满意和惊叹的新产品，但对开发人员来说，这是一个长期、艰巨而昂贵的设计过程。当然，任何新的嵌入式系统的先锋发明者可以期望有时间上的竞争优势，这个时间让他们能够改善其原始设计，企业往往仰赖这头几个月在不拥挤的市场上的销售额来收回大量前期设计和制造成本。通常情况下，竞争对手的产品也包含类似的先驱产品投入，因为同样需要类似的开发代价。

⊖ “掉电”（brownout）就像是“断电”（blackout），但不如它严重。也许我们可以依据这种颜色比喻将电源的峰值比作“whiteout”。

348 然而，当竞争对手对先驱设计进行廉价而快速的逆向工程[⊖]设计时，经济情况将会发生大幅度变化。他们的开发成本在很大程度上被逆向工程成本取代，如果我们假设这些都大大减少，那么竞争者将很容易削减先驱设备的价格。影响是双重的：首先，先锋公司的市场额定地位被削弱；其次，其市场份额由于竞争对手产品定价较低而减少。逆向工程（Reverse Engineering，RE）过程比完整的原型开发项目更短，更便宜的假设在仿冒产品的商业例子上得到证明。前期开发成本和RE成本的差别越大，先锋公司的风险就越大，而有意仿冒其产品的恶意竞争对手的收益也越大。而在一个真正具有革命性的产品很容易对其实施逆向工程的情况下，这种差别最大。

当然，应该指出的是，了解事物如何工作的逆向工程是一个历史悠久的工程方法。它更是一个有效的研究领域，是许多工程师喜欢从事的工作。然而，通过逆向工程设计仿冒品在嵌入式行业很受关注，从而导致了一些与计算机体系结构相关的挑战和对策，我们将在后面讨论。

但是首先简要观察RE过程本身是有用的，因为这能够抛砖引玉。

7.12.1 逆向工程过程

在本节中，我们将从对未受保护的嵌入式系统进行逆向工程的津贝公司的角度来讨论。其目的是研究每一步的困难，专用设备和工作，以测定此过程中的成本结构以及它如何与处于"攻击"之下的体系结构和关联。

RE过程涉及系统的自上而下和自下而上两种分析。图7-28描述了嵌入式系统的层次结构，可以看到系统本身举在地包含不同的子系统（sub-assembly），每个子配件包含一个或更多印制电路板（PCB）的单个模块或模块组。自上而下意味着从整体系统功能开始，一直向下，把设计划分成过程块，逐步说明设计功能，直至底层。自下而上通常包括先确定关键设备，然后从它们推断进一步的信息。一个例子就是在一块PCB上找到一个已知的CPU，从而推断出系统内的许多"情报"集中在该模块内。

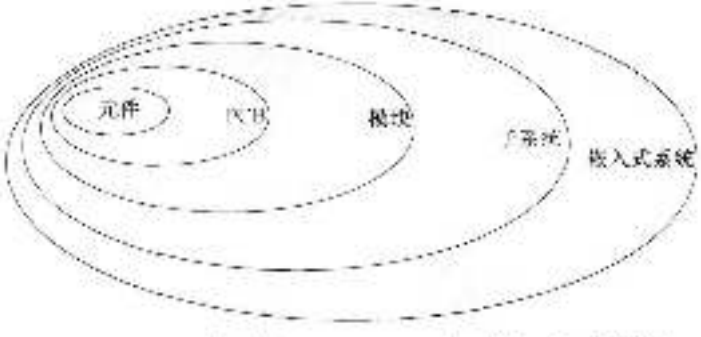


图7-28 对嵌入式系统进行逆向工程时的信息层次示意图。从外向内，把系统作为一个整体分析，包括一个或多个子系统（包括布线），模块（和它们的固定装置），模块包括一个或多个PCB（包括子板、插卡等），直至安装在PCB上或位于系统内其他地方的单个元件

嵌入式系统的自上而下RE通常包括几个分析步骤。尽管在实践中个别的RE攻击可能不一定涉及每一步，或者按照一个特定的顺序，但RE各阶段的逻辑列表一般如下：

A：系统功能分析

B：物理结构分析

- B.1：机电布局

⊖ "逆向工程"通常被定义为一个过程，涉及对设备功能、结构和技术的分析和理解，并用一种使其结构和技术与重用的方式来表现出来。

- B.2：外壳设计 349
- B.3：印制电路板布局
- B.4：布线和连接器
- B.5：制造指令

C：材料清单

- C.1：有源电子元件
- C.2：无源电子元件
- C.3：互连线和连接器
- C.4：机械项目

D：系统架构

- D.1：功能块和它们的接口
- D.2：连通性

E：详细的物理布局

- E.1：单个元件的位置
- E.2：元件之间的电气连接
- E.3：阻抗约束和位置敏感定位

F：电气连接原理图

G：对象/可执行代码

- G.1：代码处理器剥离
- G.2：可重构逻辑固件代码剥离

H：软件分析

为了突出这个过程，关于未受保护/未硬化的嵌入式系统，我们将用如图7-29所示的非常通用的系统级框图来讨论每个RE阶段。它包含一个大规模集成电路（IC），连接了易失性存储器（在这种情况下是SRAM），非易失性存储器（闪存）、现场可编程门阵列（FPGA），某种形式的用户接口，连接器以及一些与外界接口的设备，一般为模数转换器（ADC）和数模转换器（DAC）。特定的系统可能会有所不同，但作为普遍的一类，嵌入式系统通常包括一个从闪存启动的CPU，在SRAM外执行（这两者越来越倾向于要集成到IC内部），连接到离散部件或可编程逻辑（FPGA，可编程逻辑器件等）或专用集成电路（ASIC），某种形式的用户界面以及与外部模拟世界的接口。更大型的系统往往会使用DRAM、SDRAM甚至硬盘存储。还有一些集成系统倾向于在FPGA或ASIC中集成CPU软核。 350

现在我们以图7-29中的系统为例讨论RE过程的每个阶段。我们假定系统没有被保护或以任何方式故意硬化。

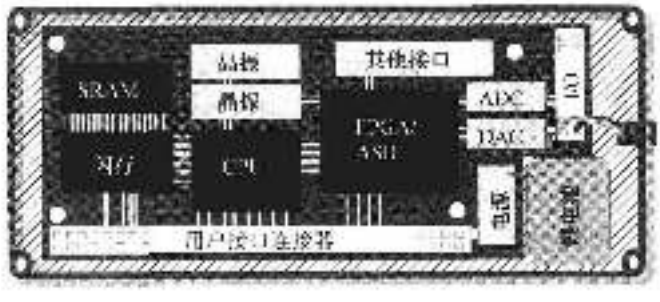


图7-29 待分析的嵌入式系统示例原图。包含两个有源IC（按敏定义为CPU和FPGA/ASIC）、两个存储元件（易失性SRAM和非易失性闪存），加上几个连接器、电源电路、晶振和接口设备

7.12.1.1 功能分析

一个RE团队通常会收到一些单元进行逆向工程。这一过程开始于咨询用户文档、维修手册、产品简介等。最低限度，要给出经过仔细检查的功能列表，随后的分析将会显示有足够的硬件和软件来支持每个所确定的功能。

这是相对简单的工作，并可以通过搜索因特网上的新闻组、博客、黑客网站等中的信息来增强。了解制造商和任何原始设备制造商（OEM），其电子邮件域的个人记录可以被跟踪和相关连。

7.12.1.2 物理结构分析

[351] 拆分可能像卸下几个螺丝来打开一个盒子一样简单，或像通过层层微机械工作一样难。在许多情况下，一个复杂的拆分过程文档是决定相应的制造装配过程的关键。可能设计者已经投入了大量的时间和精力在可制造性问题上了，因此有可能要了解这些问题的隐含价值。这些信息在服务手册中可能有记载。

拆分部件的顺序和位置应该记录下来，借助照片或录像记录会很容易。理想情况下，团队中的一名成员应专门负责过程的记录工作。执行拆分的任何观察和见解也需要在这个阶段加以注意。虽然复制外壳、内部结构、布线图等的详细机械图纸可从对零件的静态分析中获得，但装备图仍需要通过拆卸和重组来得到。物理结构分析不太可能是RE的昂贵部分。然而，不寻常的机械安排和结构背后的原因起初可能不是显而易见的，需要集思广益。

7.12.1.3 材料清单

材料清单（BOM）列出了设计中使用的所有元件，可以简单到计算螺丝、电阻等的数目。但是，可能会发生某些元件难以识别的情况，特别是半定制IC和高度微型化封装的设备（没有为识别标志留出足够的表面积）。如果用简化代码显示，它们可能会遵循JEDEC、JIS或Pro-Electron的标准化格式。可能需要对分立部件进行隔离测试，它们的特性可能与已知设备精确匹配。然而，5%或以上的公差很常见，在进行精确测定前，对许多系统的部件进行隔离和测试是有必要的。

在真正的学术研究领域，尤其是在过时的部件需要重新创建的情况下，某些部件可以被原样复制。一般情况下，物理测量中的实体模型可以与材料分析结合起来以完整地描述许多部件，包括结构项目、附件和无源元件。

PCB丝印标记对于识别微小的无标记部件常常能提供有用的线索（如Z12可能是一个齐纳二极管，而L101可能是一个电感）。标记不寻常或缺失的IC是比较麻烦的，尤其是制造商并不确定的时候。有时，片上系统的处理核厂家相比其他更容易被确定。如果可以识别制造过程，就可以将其与该过程其他已知产品的发布者关联起来。

否则，后续的系统分析（如数据总线、地址总线的位置和大小，控制信号和电源连接）可以帮助确定那些不能立即识别的部件。

大多数嵌入式系统会纳入现成的部件，甚至提供有用的丝印注释，有助于使RE过程变得廉
[352] 价。最常见的困难似乎与定制的硅器件相关，无论它来源于OEM或大规模集成（LSI）器件的内部开发。然而，虽然OEM硅器件常常无正式文档，但有时可以在线追踪到中文、韩文或日文文档。此外，内部LSI器件可以通过母公司发售，在这种情况下，一个功能列表将被刊登在某处，但在看到完整的数据表前可能需要有一个保密协议（NDA）。

显然，最好在一个粗略的检查过程中识别出主要IC，但即使不能立即识别，这个过程也不会就此完结。可以执行详细而昂贵的分析以确定确切的输入和输出，并由此推断内部功能。这可能包括研究电压等级（例如CPU核电压）、时钟频率、总线连接、去耦安排等。更具破坏性地，可以打开设备外壳，一层一层地分析硅层。IC逆向工程将在7.12.2.2节中进一步讨论。

7.12.1.4 系统架构

系统架构分析揭示了连接的粗略框图以及负责各个功能项的子系统：这涉及了解系统内模块、电路板和设备之间的分区设计。需要在这个阶段决定的另一个重要方面是确认电源或接地层以及系统的配电区。系统内的总线连接也需要确认。调试端口或 IEEE1149 JTAG（见 7.9.3 节）接口的存在对协助 RE 过程作用显著，因此它的任何迹象都是重要的。线索可能包括 5 个测试点的集合以及靠近 CPU 的上拉电阻。

在大多数系统中，各部件的电路连续性测试和视觉检测以及它们的安排可以相继进行。例如，在嵌入式系统中，相同的 CPU 数据总线可能同时与闪存和 SRAM 相连。通过连续性测试并结合数据手册很容易发现这种安排。这种测试适合球栅阵列这样的现代封装，但依然存在困难。电源引脚的位置往往是事先可预测的，很容易进行测试。对于大多数嵌入式系统，这种类型的分析是简单而廉价的，但是正如我们将在 7.13.1 节看到的那样，它可以被设计师故意弄得很复杂。

7.12.2 详细的物理布局

在没有丝印布局注释的地方，元件位置和方向的照片可以揭示所需的两个外层位置信息。接下来，所有的元件将被移开，看出钻孔位置。作为快速检查，可以比较顶层和底层孔的位置：如果它们是相同的，那么就没有盲孔，并且不太可能（虽然并不是不可能）有任何的埋孔。

下一阶段是 PCB 分层（通过逐层剥离），从一个恒定的参考位置对每一层照相。这可以被用来建立一个正确照相的层堆栈。从这一点来说，它是相对简单地复制 PCB，但是还需要铜和每个 353
PCB 层的组成和厚度信息。在实践中，可以通过研究每层 PCB 都有铜的那部分的截面发现这一点（为此，许多 PCB 有一个附连测试板（test coupon）区域，因为制造过程的变化会特别影响铜的厚度，这反过来又会影响系统性能，从而可能需要进行测试）。

通常情况下，多层区域从 PCB 附连测试板上开孔，将对端（end-on）置于一个冰球形状的模具内，模具用环氧树脂填充。设置时，镜片研磨机可用来准备一个对端截面，用于在显微镜下的测量检查，由此可以简单地读出铜厚度和层厚度。

对于大电路板，可能要从 PCB 上的几个区域开孔检查，因为制造过程中的铜蚀刻槽可能存在变化（如靠近蚀刻槽上层角落的 PCB 的边缘跟靠近底层中心区域的蚀刻不同，无论哪种情况，局部的铜覆盖密度同样也会影响蚀刻）。

越来越多的嵌入式系统需要对高速或无线电频率相关信号进行跟踪阻抗控制。在这种情况下，PCB 的确切特征很重要，包括介电常数、预浸编织厚度和树脂类型。总体而言，阻抗可以通过时域反射法或使用网络分析仪来决定。预浸料的类型和特点可以通过显微镜发现，树脂类型可以通过查看整体数据来决定。

表 7-3 给出了再构造电器等效 PCB 所需要的信息的示例，正确照相的层堆栈除外。

表 7-3 四层 PCB 的板层特性

名称	组成	厚度	名称	组成	厚度
L1 信号	1/4oz 铜箔	0.0176mm	L3 信号	1/4oz 铜	0.0177mm
预浸材料	7628 ×2	0.3551mm	预浸材料	7628 ×2	0.3543mm
L2 信号	1/4oz 铜	0.0177mm	L4 信号	1/4zo 铜箔	0.0176mm
层压材料	FR4	0.91mm	总计		1.69mm

X 射线也可能是提取布局信息的一个可行方法，甚至可以提供不明 IC 内部的有用信息。作

354 为一个例子，前文的图7-19显示了安装在PCB上的FPGA器件的低倍率X射线，可以清楚地看到连线、去耦电容（在PCB的背面）和安装在板子顶层的FPGA内部引线框。实心圆是测试点，而空心圆是连接不同PCB层上的通道的通孔。穿过左上角、像头发一样的线是焊接在IC的一个引脚上的细导线。

虽然在物理布局分析阶段可能需要一些专门设备（例如测量显微镜和反射仪），但除非涉及阻抗控制，否则复制PCB布局和层堆栈既不困难也不昂贵。

7.12.2.1 电气连接原理图

电气连接最常见的表示是网表。它指定了各个节点之间的电气连接，通常也指定连接到这些节点上的设备。网表本身并没有考虑到实际物理定位。它只关注节点间的连接关系，尽管在实际系统中物理定位本身也很重要（也许有保留区以减少干扰，或者是为了在高压环境下保证安全）。这些节点通常是连接元件的衬垫和孔，连接通常是导线或者PCB走线。

网表可以通过连接性检查生成，可检查X射线照片或者分层PCB的照片。这非常耗时且容易出错，但用以下方法核实至少比较简单：（i）在原始板上测试预期的连续性；（ii）参阅器件数据手册上的预期连接；（iii）寻找挂起顶点和意想不到的短路，比如两引脚元件只连接了一个引脚，或一个两引脚元件的两个引脚连接在了一起。

一旦发现网表，并且设备在BOM中被确认，下一步将是重新建立一个原理图来描绘系统。由网表生成原理图是一个既定的研究领域，已有商业工具可用。然而，现实中大部分RE尝试通过已知信息重绘一个完整的原理图。由原理图再生成的网表可以作为参考，与导出的系统网表进行比较以纠错。

还要注意BOM和已知的原理图允许使用仿真工具来协助BOM和网表准确性验证。

7.12.2.2 存储程序

在使用多个可编程器件（如CPU和FPGA）的地方，最简单的电气安排会让每一个器件有单独的闪存装置（分别为CPU和FPGA提供并行和串行连接）。然而，由于成本原因，通常系统内所有非易失性程序存储会聚集到一个单一的器件上。在现代嵌入式系统中，这个装置通常是闪
355 存——如果可能的话串行连接，否则并行连接。

非易失性存储器中的存储项目可能包括独立的引导代码、CPU操作代码、系统配置、FPGA配置数据或者其他特定于系统的项目。在本小节中，我们考虑决定存储程序的存储器位置的方法，着眼于它们的个体提取方法（在随后的章节中，我们将讨论固件/软件程序本身的逆向工程）。

掩模编程门阵列、非易失性PLD和ASIC不需要外部的非易失性存储器件，它们内部有存储结构。在某些情况下，可以隔离一个可编程器件，并读出其内部的配置代码。但在不可能读出或设备安全措施有效的情况下，就需要大量的黑盒子分析或内部检查。后者可通过溶解塑料外壳和/或精心打磨硅层，用电子显微镜或反射的激光读取每个存储位的状态来实现。

毫无疑问，有安全设置的程序存储设备对逆向工程师来说比多数只包含单一非易失性存储块的设计更麻烦和昂贵。这里以普遍设计为例，即CPU负责为FPGA编程，两者依次从闪存获得自己的代码。

7.12.2.3 软件

从内存转储获得的软件很容易原样复制。变化只涉及一些简单的调整，如重写字符串的内容来改变制造商的名字、序列号和版本代码。对可执行代码块也可以小心地剪切和粘贴。

与嵌入式硬件RE相反，各种规模的软件RE是一个经过充分研究的领域。从好的方面说，软件RE是实现面向对象代码重用的有效手段，而从坏的方面来看，它可以用于规避复制保护，从而导致软件遭盗版和偷窃。没有迹象显示这些结论只局限于软件。这也是笔者的经验，嵌入式系统的复制和设计偷窃比其他领域更为普遍。这可能是由于这种态度的差异或针对设计偷窃的法律保护需要变更。

软件在嵌入式系统中扮演着越来越重要的角色，虽然制造商考虑软件RE和软件安全是明智的，但是总体而言，它只是软件RE和安全保护的一个子集。

然而，嵌入式系统软件逆向工程的一个重要子集还有待讨论。在典型的嵌入式系统中，它包括嵌入式操作系统、引导加载程序和软件的非易失性存储安排。考虑一个典型的嵌入式系统，如先前图7-29讨论的那样。在硬件上运行的通用实时操作系统包含存储在闪存上的引导程序、操作系统和应用程序代码。然而，随着嵌入式系统中逐渐使用嵌入式Linux，出现了越来越多的差 356
异。这种嵌入式Linux系统通常包含如下项目：

- 引导代码
- 操作系统
- 文件系统
- 系统配置设置
- FPGA配置数据

通过移开设备并转储其内容（静态分析），或者通过操作时用逻辑分析仪分线总线信号（动态分析），可以很容易地提取非易失性存储器的内容。逻辑分析仪方法可以给出上下文相关的有用线索——例如，检测到内存读信号紧跟着上电可判定是引导代码。然而，这个方法显然只揭示了分析期间访问的内存地址的内容——实际上是目前的执行/访问轨迹，而以这种方式完全决定存储的代码在大部分真实系统中是不可能的。它要求用输入信号的每一种可能的组合和时序，以每一种可能的操作模式来操作系统，以保证100%的代码覆盖率。然而，这两种技术的结合是一个功能强大的分析工具。

地址和数据总线通常错杂在密集的PCB上以辅助布线（对它的解释见框7.9）。使用这两种方法时需注意这种布线方式会使分析复杂化。

框7.9 总线引脚交换

对像四运算放大器这样每个封装包含不止一个放大器的IC，它通常无所谓哪一个放大器使用电路的某个特定部分。因此，在布局过程中，即使原理图把单个放大器与电路不同的部分连接在一起，设计者也可以自由交换它们以改善布线。这是一个行之有效的技术。

事实上，存储设备也同样如此。例如，虽然我们自然会把CPU上的D0、D1、D2和D3跟存储设备上的D0、D1、D2和D3相连，但我们仍可以自由交换位线。如果需要我们也可以自由交换地址引脚（只要CPU总是用相同的位宽访问内存——否则只能在字节内交换，而不是字节间）。例如，考虑CPU和存储设备之间的字节连接：

CPU数据引脚	存储器数据引脚	示例位	CPU数据引脚	存储器数据引脚	示例位
D0	D6	1	D4	D4	1
D1	D0	1	D5	D3	0
D2	D1	0	D6	D7	0
D3	D5	0	D7	D2	1

这似乎没有意义。让我们考虑CPU写一个字节B到位置A，并且当从位置A读回时接收相同的字节B，它将正常运作。字节B在内存中存储的确切方式是不重要的。当写入到SRAM中，地址总线也同样如此。

CPU地址引脚	存储器地址引脚	示例位	CPU地址引脚	存储器地址引脚	示例位
A0	A3	1	A6	A9	0
A1	A2	0	A7	A8	0
A2	A1	1	A8	A7	1
A3	A6	0	A9	A10	0
A4	A5	1	A10	A0	0
A5	A4	0			

这对SRAM没有问题，但使用闪存时存在一些问题。还记得7.6.2节中介绍的编程算法吗？闪存期望接收到特定的字节模式，这意味着在特定引脚上的特定位。如果系统设计者加扰（scramble）数据总线，那么编程人员不得不解扰（descramble）闪存控制字和相应的地址。例如，使用上述加扰方案，如果闪存预期在地址0x0AA上收到字节0x55，那么编程人员需要写字节0x93到地址0x115（如上面的表格所示）。

这里的总线加扰是解决棘手的PCB布线问题很常见的一种手段。然而，使用SDRAM要非常小心，一些地址引脚专门用于列地址，一些专门用于行地址（见7.6.3.3节）。此外，一些SDRAM引脚有其他特殊的含义：特别是对于SDRAM，它实际上是通过SDRAM控制器内的一个写状态机来编程，这与闪存编程算法类似，所不同的是SDRAM不由编程人员控制，因此不能用软件进行解扰。

静态闪存分析首先需要确定不同存储区域的范围、界限和特性。在有可擦除闪存的分隔符的地方（即块边界上以0xFFFF或0xFF结束的长字符串），这个过程是微不足道的。否则，引导代码可能从向量表开始并最可能保留在闪存的最低地址或者一个特定的引导块中。一个FPGA编程图大约是FPGA数据表中指定的大小，或者用标准算法压缩（zip、gzip或压缩，将从可搜索到的签名字节开始）。文件系统将通过它的结构识别（在Linux台式机上，一旦计算中某些项目被转储用于分析，file命令可快速地确定这些项目的性质）。Linux内核以及其他操作系统内核包含了不同的签名码，甚至可能包含可读的字符串（在Linux台式机上strings命令将会找到并显示它们）。

静态和动态分析相结合功能强大并可以提供重要的存储器内容信息。例如，系统配置数据可以存储在闪存的任何地方，单凭内容很难确定。然而，简单地操作设备和改变单一配置设置将会导致内存内容的变化。这可以通过比较前后的内容来确定，或者用逻辑分析仪跟踪写到闪存地址单元中的内容。

357 ~ 358

在极端情况下，闪存可被原样复制并在复制品中进行复制。总体而言，对揭示存储程序的非易失性存储器进行逆向工程过程并不难，除非设计者专门采取措施来保护嵌入式系统软件。

7.13 防止逆向工程

由于逆向工程的不可阻止性，这个问题就转化为一个经济问题，即我们如何在最小化自身额外成本的情况下，最大化竞争对手进行逆向工程的成本。为了分析这个问题，我们将会借助7.12节中关于嵌入式上下文环境中的逆向工程的描述，并对它们进行分类。首先，会基于实现复杂度、成本，以及逆向工程实施者的经济影响，对抑制方法进行评级。我们首先将汇集所有嵌入式系统设计者感兴趣的方法，然后聚焦到其中与计算机体系结构相关的部分。

逆向工程抑制方法从技术上可以分为两个大类：专注于设计时期的*被动方法*，在逆向工程攻击发生时予以抵抗的*主动方法*。前者倾向于实施结构性变化，这在实现上比后者要廉价。我们将详细讨论两种方法。

逆向工程的成本影响因子取决于逆向工程保护措施，主要有三个影响因子：

- 由于逆向工程系统而花费的更多的时间导致劳动成本增加。
- 由于逆向工程对更高层次专业技能的要求而导致的劳动成本增加。

359

- 由于针对逆向工程过程需要购买特殊的设备而导致的成本增加。

在某些情况下，如果需要额外的组件，则还会增加BOM成本。

按照7.12.1节中对逆向工程过程的描述，第一个层次的保护可以应用于功能评估：逆向工程阶段A。在这种情况下，限制服务手册和文档的发布，可以降低逆向工程团队的信息获取度。制造商应该控制、监督，甚至在理想情况下限制员工不经意地提供相关信息，尤其是上传到网上的信息。这无疑将会增加逆向工程所花费的时间和努力。

阶段B，通过使用防篡改的配件（如torx）和定制的螺丝形状来使得必须购买这些特殊的设

备才能进行物理结构的分析，以此稍微增加物理结构的分析难度。单向螺丝和胶接外壳起着相似的作用。全灌封的PCB提供了另一层次的保护。在最小成本的情况下，令人不愿使用这些方法的主要诱因是产品的可服务性，这通常是产品必须具备的。

连接线没有使用颜色编码可能会使得制造和服务过程变得复杂，但这会给逆向工程团队造成更大的困难，从而阻碍他们的工作。

不同寻常、定制和匿名的部件会在阶段C中使逆向工程的系统材料清单（BOM）变得复杂。然而，无源器件（在阶段C.2）会很容易被移除并对其做单独的测试。丝印的缺失会给制造和服务造成一定的困难，但是也会相应地在阶段C.3、E.1、E.2和F减少提供给逆向工程团队的信息。但是，到目前为止最为有效的防止BOM被逆向工程师的方法就是使用定制芯片（或者使用逆向工程团队无法购买到的芯片）。在阶段C.1，逆向工程师面对的是大量无标记的最小化无源器件组成的集成电路，没有丝印，也没有更进一步的信息，这的确会有效地给他们的逆向工程增加困难和花费。确认并复制或者只是确认芯片就会显著增加他们的开销和前期成本，这样高的成本只有在大规模生产时才是经济可行的。

从最佳安全性考虑，JTAG（7.9.3节）和其他的调试端口都应该从半定制的芯片上移除，而且不要从标准部件连接到连接器或者测试板上，更不应该标记上TDI、TDO、TMS、TCK。对于有引脚暴露在外的封装器件，这些还是很容易被访问到。这种情况下，BGA（球栅阵列）器件就是首选。但即使是BGA器件，未布线JTAG引脚往往仍可以通过从PCB的另一面深度钻探而被访问到，这意味着背靠背式的BGA布局是最安全的（例如，在PCB的一侧放置BGA CPU，而在其正下方的板对面安装BGA闪存设备）。这种做法的劣势在于双面布局会增加制造的成本。双面的BGA仅仅是更贵的一步，并不一定能防止逆向工程，因为还是有可能（虽然极其困难） 360
移除BGA器件，重新形成焊球，然后改装成一个焊接到PCB上的载体的。通过载体的中间信号可以用于分析。

背靠背式的BGA封装通常需要盲孔或者埋孔，这会使得PCB制造成本增加（根据经验是10%），布局过程也变得复杂，从而明显影响硬件调试和所需要的改动。然而，紧凑的PCB也成为产品的一个特点。相应地，PCB的层数也经常需要增加以适应背靠背式的布局，因此也增加了逆向工程中划分层次和逐层分析的成本。对于多层的PCB设计，通过使用X射线在阶段E.2和E.3来分析布局的细节是很困难的，并且通过对所有可用空间填补电源平面的方法，可以使得这种分析更加复杂化。电源平面的填充会在X射线照片上形成交叉阴影线，掩盖内层的单个器件的信息。

在E.2阶段，当器件以非寻常的方式来操作时，如跳跃的地址和数据总线，电气连接就会很难确定。连接没有使用的引脚在不增加制造成本的同时，却给逆向工程增加了困难。

7.13.1 存储程序的被动模糊

对于嵌入式系统中存储的代码，可以找到很多结构化的方法来使之产生模糊，这使得RE的G.1和G.2变得更加复杂。我们不会更多地讨论这个问题，因为这是一个热门的研究领域。然而，这里有另外一些体系结构方面的问题我们可以进行探讨。

首先，如前所述，代码段之间的部分（闪存中非可擦除部分）可以填充随机数字或者冗余代码，致使对于分离内存区域的监测没有意义。与初始引导代码不同，闪存的其他部分同样可以被加密，如果不需要从闪存执行代码的话。这将使对于闪存内容的分析变得困难。然而，对于未被加密的引导代码，则很容易追踪和分解，并从中发现系统未加密的入口点，致使这种加密策略的安全性受到质疑。

在闪存范围内分散代码、数据和配置内容将会带来一些编程上的困难，但是却是应对存储

程序分析的主要保护手段。如果一个FPGA图像被存储在闪存中，那么一种简单的模糊方法是对其每个数据字节与闪存其他某区域中的数据字节做异或运算，并存储成一种定制的压缩FPGA图像（不是gzip、zip或者相似的有可识别签名的方式）。

这里所讨论的几种防止RE的方法在表7-4中汇总给出。其中，增加的RE成本和所付出的
361 设计成本以及对制造的影响以一种5点评分法来标注。

表7-4 增加硬件逆向工程成本的被动方法评定指标，5=增加得最多，0=增加得最少

	设计成本	逆向工程成本	制造影响		设计成本	逆向工程成本	制造影响
防篡改螺钉	2	0	1	盲孔或埋孔	2	2	4
胶接外壳	1	1	1	总线信号跳跃	1	1	0
灌封	1	1	2	ASIC信号路由	5	3	2
无丝印	1	1	1	FPGA信号路由	2	2	2
擦除元件标识符	1	1	2	无调试端口	1	1	2
使用BGA封装	1	3	3	随机填充未使用的内存	2	2	0
只有内层布线	2	2	3				

7.13.2 可编程逻辑家族

基于SRAM的FPGA通常需要一种配置比特流。这种比特流由外部器件提供，例如串行闪存配置器，或者像示例系统那样由微处理器提供。因为物理上这种比特流可以通过很小的代价获得，所以经常会通过读出和复制的方式拷贝这种固件。

基于EEPROM的可编程逻辑器件（PLD）（代替了EPROM版本）和新型的基于闪存的产品更为安全，因为配置程序贮存在内部，无需在复位时传递到设备中。注意一些带有闪存设备实际上包含两个硅片，一个存储芯片和一个逻辑芯片，因此使安全性降低，因为一旦外封装被拆除就可以读取到配置比特流。通常来讲，那些在复位之后需要马上配置的设备通常包含分布在硅片
362 周围的非易失内存单元，在复位后需要几毫秒进行配置，其间配置比特流可以被获取。在任何一种情况下，包括从Altera到Xilinx在内的大多数设备，都会提供安全设置以阻止从正在配置的设备中读取比特流。在此强烈建议使用这种特性。

在通常的单元结构设备中，包括掩模编程门阵列（MPGA），内存配置单元的位置是已知的，根据设备制造商的设备分类可以确定。通过使用7.12.2.2节介绍的方法，配置数据和原始程序就可以被获取，虽然这也需要复杂的技术支持。

通过逐层分析硅片（类似于PCB划分，但需要对硅层进行细致的研磨），一个全定制ASIC也可以被逆向工程化。但是可以通过一些策略使之复杂化，例如加入网状覆盖层。反熔丝（anti-fuse）FPGA通常被看做是一种最安全的标准可编程逻辑设备，因为其熔丝位置在较深的硅布线层之下，而不是暴露于表面。

RE包括ASIC和反熔丝FPGA不是不可能，但需要非常级别的专家，使用昂贵的专门设备，耗费大量的时间。

7.13.3 主动RE防范

在7.13.1节中提供的被动RE防范方法也有主动版本。可以通过使用处理器多余的输入和输出管脚路由那些时间上不关键但功能关键的信号，从而达到模糊电气连接的目的。

跳跃地址和数据总线对于RE来说已经很困难了，而动态跳跃总线将会使RE更加困难，但同时也增加了为防范RE所付出的代价，因为主动设备必须增加跳跃/解跳跃总线的功能。

ASIC有可能是防范RE企图的最终方法，但看似平常的FPGA方法也会相当高效。在这两种情况中，在逻辑电路中实现的IP核心（将会在第8章讨论）不容易辨识和分离出来，且可以通过各种方法存取外部存储程序——可以线性地、非线性地存取，或者使用置换或加密方法。一个完全定制且没有任何公开文档的CPU内核可以通过保护指令集体系结构细节进一步增加安全性。进而，在每个版本的产品实现中对指令集可以稍加变化从而防止对核心程序反复RE。这是一个不昂贵的软件/固件保护措施。

7.13.4 主动RE防范分类

RE防范（RE mitigation）的基本形式可以从两个方向上细分：一是主动模糊方法，即隐藏；二是达到主动模糊目的的时间或空间方法。任何现实系统都可以通过结合这些方法达到最大化的效果。

信息隐藏（information hiding）利用现有资源通过各种方法对攻击者隐藏信息。通常采用的 363
方法是组合代码与数据，将软件隐藏在像启动引导映像这样的数组中，或以不明显的方式读数据来实现信息共享。还包括在边界电压上运行设备，依靠非常规握手和数据处理策略等。

模糊主要作为一种被动方法（例如在程序中交换标签名和功能名，或者搞乱PCB丝印注释），它也可以用于主动防范，例如改变总线连接和设备管脚使用方式（如多路选通中断输入管脚和信号输出管脚）。这是又一种利用已有资源进行专门的设计，误导RE团队使RE过程复杂化的方法。

还可以增加一些资源刻意误导或扰乱RE团队。其中可能包括大量的伪随机数据传输、乱序代码读取等。也可以在传输信号上叠加上随机调制的电压信号，或采用有某种意义的信号驱动冗余信号线。动态地看，这或许包括一些基于篡改监测的模式变化或相应的极端反应。

空间方法作用于布局和联通层，例如，根据存储器地址加扰总线顺序，以不明显的方式开关信号路径路由设备。

时间方法通过修改事件序列或时间来实现模糊。例如，引导加载程序只执行所取入指令的一个子集。又如，一个能够从存储器中预取代码页的存储管理器件以非线性方式存取，尤其是与执行顺序不一致。

这些分类组合如表7-5所示，其中标出了相关强度级别。

表7-5 主动防范方法的相关强度，5——最高，0——最低

	静态方法	动态方法		静态方法	动态方法
信息隐藏	0	2	刻意扰乱	4	5
模糊	1	3			

从开销上说，与静态方法相比动态方法用于开发、调试和测试的开销会更多，同时还会增加制造和服务成本。信息隐藏和模糊方法的开发成本类似，主要是增加了NRE。而刻意扰乱方法 364
与前两种方法相比无疑会带来更多的开发成本，且会提高制造成本。

很显然用户定制芯片通过实现主动模糊和保护措施可以提供最大程度的保护。在出于安全目的构建全定制ASIC时，如果开发者比较在乎成本，可以考虑开发一些通用全定制安全ASIC，以适用于一系列产品之中。对于RE而言，若设计采用了主动保护方法，特别是动态时间方法，则意味着需要有技术更加高超的工程师队伍，他们需要使用一些专门设备。例如，若要分析以最低限度工作的时序信号，则需要使用带有非常低电容有源探测器的高速数字示波器，甚至使用超导量子干涉设备（SQUID）。对于非常规握手机制的分析则需要多信道向量分析仪。

7.14 小结

本章讨论了计算领域的一些实际问题，比如存储技术、片上外设、时钟策略和复位信号部署等。嵌入式系统通常会受到内存短缺的困扰，这可以通过使用内存分页和重叠技术缓解。我们还以广泛使用的嵌入式 Linux 操作系统为例分析了典型嵌入式系统的存储结构。

看门狗定时器是一种确保实时嵌入式系统可靠性的有效手段。针对可靠性问题，我们还讨论了错误检测和纠正。

因为 CPU 速度越来越快且越来越复杂，由此增加了制造和开发难度，为此需要测试和验证。该问题我们从三个方面进行了讨论：IC 制造、系统制造和运行过程。

最后讨论了逆向工程相关问题。这是一个与嵌入式系统休戚相关的问题，尤其是那些消费
365 类设备。我们讨论了恶意逆向工程的执行过程，并介绍了几种防范逆向工程的方法。

思考题

7.1 确定在嵌入式系统中可以使用片上系统（SoC）处理器的 4 个因素。

7.2 根据下列所需功能，列出在微控制器上实现可编程 I/O 管脚的最少控制寄存器设置：

（1）可以配置通用输入/输出（GPIO），或者一个内置的外部设备的专用输出（如 UART）。

（2）当在 GPIO 模式时，可以配置为输入或是输出。

（3）每个引脚都可以单独读取和写入。

7.3 你是否会期望一个单芯片微控制器或四核高速服务器处理器在内存方面投入更大比例的硅片面积。通过在这两种类型机器上的应用来证明你的回答。

7.4 列举几种在过去二三十年中，半导体设计者为减少 CPU 传输延迟而采用的方法。

7.5 对于计算机系统的时钟策略（或时钟本身）进行怎样的变化可减少系统产生的电磁干扰（EMI）。

7.6 在靠近 CPU 的电源引脚处放置什么样的外部器件可以减少 EMI 的产生？为什么？

7.7 在下面列出的应用中，指明最合适的存储技术。

a. MP3 播放器需要以 350Kbit/s 的速率从 8GB 存储器中访问音频数据。即使电源关闭，数据（你的歌曲）也应该保留在存储器中。

b. 一个简单的很小的嵌入式系统的程序存储器，只执行一个任务。制造商会制造几百万个这种器件，不需要提供可再次编程功能。

c. 在个人数字助理中，256MB 系统存储内置于 ARM9 嵌入式系统中，运行先进的嵌入式操作系统，如嵌入式 Linux。

d. 在上述系统中已包含一个 16MB 的非易失程序存储器，用于存放操作系统的例程，且程序直接从那
366 里执行。

e. 在一个小的嵌入式系统中，4 KB 运行时存储连接到一个中等大小的微控制器。

可选存储技术（一个应用选择一种存储技术）如下：

- 串行闪存
- 并行闪存
- SDRAM
- SRAM
- RAM

7.8 说出 7 个在嵌入式系统中引导加载程序（例如 u-Boot）的常用功能。

7.9 一个 BGA 封装的典型嵌入式系统 CPU 安装在原型嵌入式系统的 PCB 上。设计师怀疑有焊接故障致使系统不能正确运行。列出两种识别潜在问题的方法。

7.10 一个字节 0xF3 通过两个半字节的方式在有噪声的无线信道中传输，每半字节都使用汉明(7，4)码进

行编码。请参阅框 7.7 的方法，以十六进制的形式写出这两个 7 位编码。

7.11 重复问题 7.10 的汉明编码问题，这次使用框 7.8 的方法传输字节 0xB7。

7.12 请确定 3 个主要的原因，为什么必须在嵌入式系统中加入逆向工程保护，但会稍微减少制造商的利润。

7.13 在嵌入式系统中，为了确定以下几个方面，逆向工程团队如何利用 JTAG 到 CPU 的连接。

a. 识别 CPU

b. 电路连接和系统原理图

c. 安装在系统中的非易失性存储器（闪存）的内容

7.14 为什么这么多的 SoC 微处理器使用 32.768kHz 晶振？

7.15 什么是时钟抖动？它如何影响处理器的最快时钟速度？ 367

7.16 如果将字节 0xA7 编程到并行闪存的一个位置，其后又将字节 0x9A 编程到同样的位置（在两次操作之间没有擦除），之后闪存的这个位置包含的是什么值？

7.17 EPROM 存储设备有一个小玻璃窗口，它可以用来接收紫外线以擦除存储器阵列。闪存（以及 EEPROM）存储器内容电可擦除。说明闪存技术与 EPROM 相比的两个优势。

7.18 想象一下，你在领导一个小型设计团队设计新的嵌入式产品：硬件已经准备好了，软件工程师基本完成了系统代码。系统中有大量的串行闪存，但是只有少量的 SRAM 可用。距离推出产品的时间只有几个星期了，软件开发团队发现无法在 SRAM 中运行代码，也没有办法减小代码规模。在不改变硬件的情况下，提出一种存储解决方案来解决该问题。

7.19 一个 JTAG 扫描链可能有几百位长。这条串行链可以向 CPU 的扫描路径中输入以改变其行为，也可以从 CPU 中输出以读取其状态。举例说明一些位的含义（即它们能改变什么行为，它们能确定什么状态）？

7.20 如何利用三重模块冗余确定正确的计算输出？假设 3 个相同的块在一个故障系统各自的输出为 0xB9、0x33 和 0x2B，基于此阐述你的答案。如果这些输出接到一个按多数位选取的选举器上，系统的最后输出字节应该是什么？ 368

第8章

Computer Architecture: An Embedded Approach

CPU 设计

在前面几章里，我们已经介绍了很多用于微处理器的从简单到高级的概念，这些概念贯穿于整个计算机体系结构学科。

本章我们将继续介绍这些知识并在实际应用中进行巩固。我们以一个真实的处理器为例进行介绍，这个处理器可以让一般的嵌入式工程师进行设计、修改、使用和复用。我们通过软核（soft core）在嵌入式系统中的使用来完成相关实例的阐述。

8.1 软核处理器

所谓软核（或者软处理器）是指用逻辑描述语言编写的 CPU 设计，且该设计可以在可编程逻辑器件上进行综合。一般采用 Verilog 或者 VHDL[⊖]高级语言，并最终在 FPGA 上进行综合。

这不同于大部分处理器制造商的角度，他们尽可能地针对半导体制造工艺进行更低层次的设计，这样做的目的是使半导体芯片获得更好的性能表现。有时候，对于一个特定的处理器会有定制及软核两种设计，比如我们所熟悉的 ARM，在这些案例中，软核设计会表现出较弱的性能（主频较慢，功耗较高），但是软核设计却在使用中拥有更高的灵活性。

目前有很多软核处理器，并且大部分可以免费获得[⊜]。这些软核处理器在 FPGA 上实现后，其效率、速度及成本几乎无法与定制的微处理器相比。

369 其他的可能性就是使用商用软核——目前主流 FPGA 厂商都拥有自己的软核——或者设计用户自己的核。我们首先剖析若干软核，然后介绍获得核的三种主要途径，最后设计一个完整的定制核并运用本书前面描述的很多技术。

8.1.1 微处理器不仅仅有处理器核

一个在 FPGA 上实现的软核处理器，是一个能像 CPU 那样运行的逻辑块。简单地说，这块逻辑能够在时钟的控制下工作，可以加载数据并且按照指定的程序进行处理。程序可以存放在 FPGA 内部，也可存放在外部存储器上，比如在大多数的嵌入式系统中均采用 RAM 或者闪存。

以上的这种存储安排很好，但是，微处理器不仅仅只是处理器核。参考 7.2 节中讨论的三星 S3C2410 基于 ARM 微处理器的特点，有一个关于其内部特征及其外部设备的很长的列表，主要包括：

- 16KiB 的指令存储器和 16KiB 的高速数据缓存，并有 MMU。
- 用于外部 SDRAM 的存储控制器。
- 彩色 LCD 控制器。
- 大量串行端口，比如 UART、SPI、IrDA、USB、IIC 等。
- SD 存储卡及 MMC 存储卡接口。
- 一个 8 路 10 位的模数转换器和触摸屏接口。

⊖ VHDL 代表 VHSIC 硬件描述语言，其中 VHSIC 是指甚高速集成电路。

⊜ 这里指的是针对免费处理器和其他“IP”核（IP 是指知识产权）的项目集合，见 www. opencores. org。

- 拥有日历功能的实时时钟。

显然，该处理器核自身（只是附带地作为一个条目，并没有被列入三星自己的文档中）仅仅是由一小部分命为S3C2410的集成电路组成，可以在市场上买到并应用在嵌入式系统中。

进一步来说，如果一个工程师得到了一个用高级硬件描述语言（HDL）编写的ARM处理器核，并将其加载到一个FPGA中，他仍旧没有获得一个完全的微处理器。此外，FPGA不可能达到S3C2410的200MHz频率（尽管现在的FPGA工艺据称支持1GHz的时钟频率）。

更多的工作是需要在FPGA上实现全部其他外部设备及接口，而这也将给FGPA带来更大的负担，最终结果是整个处理器更慢、功耗更高并且在价格上将远远高于一个现成的ARM处理器。

那么阐述了这么多缺点，为什么还有很多人考虑使用软核？

8.1.2 软核处理器的优点

在世界范围内，大约有数百万的系统在使用软核处理器，尽管这个数量要远远小于达数百亿的ARM设备。但是，肯定有更好的理由去选择软核处理器。让我们从性能、可用性和效率这三个方面来考虑这些理由。

8.1.2.1 性能

由于之前我们提到过软核处理器一般都比专用器件要慢，因此性能这个指标明显应该是标 370
准微处理器的优势。这些是毋庸置疑的，不过需要注意的是有一些性能指标比时钟频率更重要：

- 并行系统允许多个处理器或者处理器核并行地实现或者运行。我们可以很容易地将几个或多个软核集成进单个FPGA中，并组成一个并行系统。就像目前大部分人遇到的问题一样，如何高效地使用这些多核系统是一个不能被忽视的问题。
- 复杂指令集计算机（CISC）被公认为是程序员创建定制指令的通用方法。相反，精简指令集计算机（RISC）消除了具有更高复杂性或者较少通用性的指令，而将注意力放在了让最具通用性的指令运行更快上（这样，复杂的CISC指令可以通过多个简单的RISC指令来完成）。然而，在一个嵌入式系统中，往往代码规模较小并且固定，这样我们可以选择一个不同的指令集来完成相应的功能。比如，在一个系统中有很多除法运算但没有逻辑操作，那么最适合的RISC处理器应当有一个除法器并有很少的逻辑指令。在这里代码事先已知并且固定，为了能够快速执行这些代码，需要定制设计指令集。
- 尽管没有为适应特定的代码段而修改指令集，但对于一个在FPGA里给定的处理器核，我们通常是增加一个专门的功能单元或者协处理器。在上述的例子中，我们会选择对一个标准核增加一个除法器。而现成的处理器却无法在内部增加除法器，尽管它有外部协处理器接口。
- 软核通常用VHDL或者Verilog编写。它们通常不包含复杂的总线，也不包含存储（有时连cache也没有）。设计者使用这些软核时必须编写总线和存储来配合软核的使用。这是个缺点，但针对具体应用编写定制化总线，也有可能使缺点变为优点。相比之下，一个现成的处理器有可能搭配了一个总线方案，而该总线与其实际应用并不是完全相配。

8.1.2.2 可用性

对于软核，其可用性有两层含义。第一层就是如何能容易地获得并使用软核，第二层就是能够确保所获得的软核按照需求正确运行。我们将就这两层含义分别进行阐述：

- 使CPU的设计标准化对于产品设计者而言就是痛苦之源（包括作者自己在职业生涯早期时），他们为了让其产品发布而一直努力地工作，但是很快会收到CPU厂商的通知，说他们设计的CPU宣告失败。这些对软件和硬件来说都需要一个非常基础性的重新设计。而这种重新设计的问题不仅仅存在于大公司中，对于中小型嵌入式系统公司来讲也普遍 371

存在。那么，考虑一下拥有一个自主CPU设计的吸引力，它可以让你永远与喜欢消减成本的半导体厂商保持在一条起跑线上。你可以改动你自己的CPU，复用它的软件代码，复用硬件，随心所欲地扩展及升级。尽管软核处理器最终是在FPGA上进行综合，并且具体的FPGA可能会走向生命的终点，不过你可以很容易地将自己的CPU移植到另外一块FPGA上，同样的代码、同样的处理器将运行在新的FPGA上，也许会比之前更快一点。

- 在欧洲及北美以外的设计师也体会到了相同的问题。新款CPU在这些国家的市场里上市通常会很缓慢，并且很难让用户买到。对于一个需要购买上万个CPU的公司来讲这通常不会成为问题，但是对于中小型嵌入式公司来讲，这是个很严重的问题。例如，在新加坡，作者每次购买这些CPU，都不得少于100件，这很不利于原型产品的开发。很感谢地说，FPGA厂商对于这些中小企业来说还是比较贴心的。
- 可用性对于电子系统而言意味着系统可以正确地工作，并且可以在你需要它时仍旧在工作。好的设计是确保后者，即可靠性，但有时候为了确保一个CPU可以正常工作并且稳定地工作，我们必须对该CPU进行复制，也就是进行冗余设计。换句话说，就是两个CPU肯定比一个CPU好，三个CPU也肯定比两个的更好，等等。一个软核处理器可以尽可能多次数地进行复制和并行化，只不过是消耗更多的FPGA资源和功耗而已。相比较而言，复制一个专用处理器意味着翻倍集成电路芯片，使成本成倍增加。

8.1.2.3 效率

效率可以从功耗、成本、空间等几个方面来衡量。现在存在一些观点，全部都是基于以下几个基本原因：

- 在8.1.1节中列出的S3C2410的特征列表令人印象深刻，并且对于任何一个设计者来说要想在定制的软核处理器中复制出这些处理器特性都是很难的事情。然而，所有这些特征都是必需的吗？当设计一个全功能SoC解决方案时这个答案是肯定的。不过，在特定情况下，只需要有一小部分特征就足够了，所以在这里答案也许可以是“不”。软核仅包含这一小部分特征就可以了，因此接口及外设是绝对需要的。它们不会浪费硅资源（或者FPGA资源）在一些用不到的功能上，从这一点来讲，有时软件比标准CPU更高效。
- 黏合（glue）逻辑是将微处理器与其他设备绑定在一起而得名的。例如将反向器和与门黏合在一起。有时候，一个较大规模的黏合逻辑可以通过使用小规模的FPGA来完成。假设黏合逻辑是一种普遍存在的设计方式，那么它将会被应用到各个地方，用软核黏合
372 上其他逻辑的FPGA来取代传统的微处理器。有时候，对比专用CPU，这样的设计结果可以减小PCB板上的空间，降低生产成本等。

8.1.2.4 人为因素

人为因素往往都被工程师们所忽略，然而，像任何技术因素一样，人为因素也同等重要。只有目睹过工程师们在小组设计会议中由于灵感枯竭而导致的沮丧和不理智，才会对人为因素的重要性有所认识。在软核设计中的人为因素主要有以下几方面的考虑：

- 开发自己的计算机才是真正的乐趣。动力充足的设计工程师有着极高的效率和刻苦的工作态度。做一件事情的动机通常来自兴趣，设计一个定制的软核往往是工程师们最感兴趣的一件事情，而这一点大部分管理人员却意识不到。
- 当工程师可以完全自主设计一个软核时，他往往也承担了上面所说的灵感枯竭的风险，而这对于工程师来讲恰恰是另外一个很大的动力，这样的风险可以帮助他们追逐设计上的完美。你可以很轻松地设计并且拥有你自己的软核。
- 当开始着手一个新的嵌入式设计项目时，往往都有一定的时间去考虑应该采用哪个嵌入

式处理器才能适合该项目。这个适合度将决定设计所需要的各种设备，以及这些设备的最佳选择。不过，让工程师制定一个学习计划来学习一个新的微处理器这件事情却有点欠考虑。有时候，不得不让工程师去熟悉处理器的新特性及工作方式，这样，对一个个新设备的学习过程往往会导致项目数月的拖延，或者会因为初学者不可避免的错误从而延长设计周期。使用一个团队都熟悉的处理器往往更好，尽管这个处理器可能不是适合度最高的。此时使用软核会解决此类问题，因为一个团队如只熟悉一种软核，那么该软核将会被连续使用在很多设计中。通过 FPGA 实现时，对外设、功能单元及协处理器的轻微改动可以确保软核在新项目中是较优书选择，并且可以避免让开发团队陷在新书处理器培训中。

8.2 软硬件协同设计

软硬件协同设计是对一个包含硬件和软件的系统进行设计时的特定术语。这个概念与嵌入式系统尤其相关，因为嵌入式系统一般都包含定制化的硬件和定制化的软件。

当为台式 PC 编写软件程序时，程序员们通常假设硬件无误差并且功能上是正确的。当设计一个新的 PC 时，设计者们会使用标准测试软件来测试该 PC，而这些测试软件已经被证明正确无误并且在可工作的硬件上能够正确运行（比如在上一代的 PC 上）。

在嵌入式系统中，潜在的问题是硬件和软件一般都是一起开发，这样一来，硬件无法证明软件 373
是正确无误的，反过来也是一样。因而，调试及开发系统的过程会陷入这样一个矛盾的问题中。[⊖]

如图 8-1 所示，假设一个系统包含一个 FPGA 和一个 CPU。一个嵌入式系统设计者，已知该系统的需求，需要决定如何实现这些需求。有些需求可以用软件来实现，有些用硬件来实现，而有许多是这两者的结合。一般来讲，软件实现更加灵活，容易调试，更改和增加新特性，而硬件实现能达到更高的性能和尽可能低的功耗。

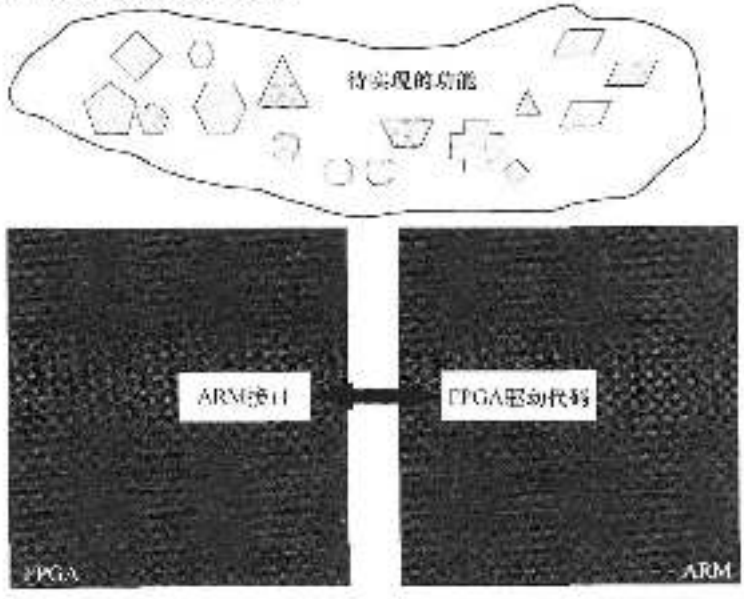


图 8-1 对于一个包含 CPU 和 FPGA 的嵌入式系统来讲，其设计过程需要明确放两个处理器单元将要实现的设计任务，前提是假设存在一个可以工作的 CPU-FPGA 接口

⊖ 在硬件设计者中有一个传统：当事个设备工作不正常时使用激光笔。而在程序员中也有一个传统：当代码出现问题时就指责硬件。无论对人员的培训中，这种开发环境都有趣，后个相关问题。

有一些任务很自然地适合用FPGA来处理，比如位级操作、串口处理或者并行处理。有些任
[374] 务适合用高级编程语言在CPU上以软件实现，比如控制软件、高级协议、文本处理等。考虑FPGA的规模和MIPS（每秒百万条指令）存储器限制，有助于设计者对处理任务的分配。还有很多问题值得考虑，而这些问题甚至需要作出折中处理。比如："谁来编码？""如何维护这些代码？"和"系统在随后需要升级吗？"

在FGPA和CPU连接的过程中有一个特别的问题需要注意。这个连接将会有共同的带宽和延迟限制：它只支持一定数目的数据流并将自然地引入消息传递过程中的小延迟（这个延迟对于实时系统很重要）。当然，通常一个设备（通常是CPU）作为主设备，另外的设备作为从设备。消息和数据传输始终来自主设备，所以延迟可能会根据消息的不同来源而产生变化。带宽也会因此不一样。还有可能的是，这两个设备不会时钟同步，所以任何在这两个设备之间传递的数据必须缓冲。几两边都需要进行缓冲，这也增加了数据传输延迟。

如果FPGA里包含一个软核，上面这种情况会更糟糕。这意味着要做一个更加艰难的决定，任务是运行在CPU上，在包含逻辑功能/状态机的FPGA上，还是在FPGA的软核上。

尽管困难重重，一个分配好的设计将最终确定下来，如图8-2所示。包含接口规格在内的各种详细设计将被一一列出并分发到软件组和硬件组，然后他们着手按照要求开始各自的实现工作。

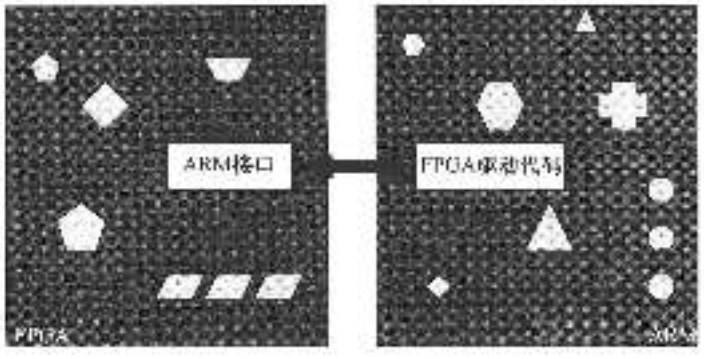


图8-2 在嵌入式系统设计的软硬件划分步骤中，将任务分配给一个CPU或FPGA的示例图

一段时间后，整合工作开始，将软件设计和硬件设计连接在一起，结果往往会发现系统无法正常工作。这个时候，两支团队在让他们各自的工作能够相互配合之前，先要承担属于自己的
[375] 责任。

不幸的是，即便是系统最终能够工作，也很少有最理想的办法来解决这种整合时出现的问题。究其原因，是由于在设计的最初划分过程中有着过多的人为主观因素，并将这些因素延续到后面的实现工作中。

近些年来，软硬件协同设计的出现正是为了应对设计同时包含软件和硬件的系统时所存在的困难。协同设计方法学一般都通过计算机辅助设计工具来实现，其目的是简化设计过程（缩短设计周期，降低设计成本，减少设计错误），优化软硬件的划分，使整合过程更容易。

软硬件协同设计涉及以下步骤，我们假设以一个混合的FPGA/CPU系统为目标：

1. 建模。在这里，一种系统的具体设计必须用机器可读的格式表现出来。这种表达必须是一种形式化的设计语言或者一种简单的编程语言，比如C或者MATLAB，来评估系统在给定输入数据后的输出。这个表现会在随后用来验证系统的功能正确性。

2. 分配。正如前面所提到的，也许人工才能根据系统描述信息做出最好的分配，而不能单纯依靠机器。有时候，可以很容易地把系统划分为几个不同模块，但是往往存在一些重要且有

可能需要对原始模型进行经验的改动。

3. 协同综合。使用CAD工具可以构建出一个由以下三项组成的模型：FPGA代码、C语言代码以及这两者之间的接口。FPGA代码用FPGA设计工具进行综合，C代码用编译器编译并装载进处理器模拟器中，而这两者之间接口往往是以前面这些生成文件为基础的。

4. 协同仿真。这意味着将在设计工具中一起运行第3步所述的三项。理想情况下，这些都是实时地运行，但是往往比真实的硬件速度要慢上千倍，然而对于软件实现来讲要求是位级精确。

5. 验证。这意味着将第4步中的结果与原始模型的结果进行对比测试。

在这个过程中将会有一些重复：当发现错误时（其实也可以认为是出现了更进一步优化的机会），对分配和设计的经验改动是有可能的。图8-3清晰地显示了这些重复的步骤，在系统设计的每个步骤中都需要进行验证，从而可以看出系统模型的重要性。

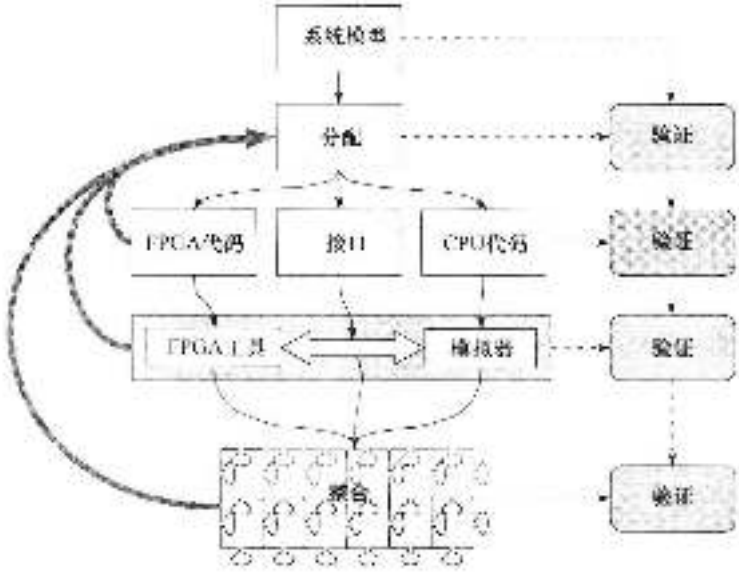


图8-3 软硬件协同设计过程示意图，给出了开发流程，同时在每个步骤中都有验证过程，如果检测出错误则会返回之前的步骤并重复执行

376

需要的事实是需要一起仿真：硬件（通常是FPGA）、软件以及它们之间的接口可以用设计工具来开发，并且在仿真中进行测试，这样问题可以尽早发现和改进掉。当系统最终可以和模型保持一致时，就可以将它移植到硬件上进行集成和测试。正如大家所希望的，这时软件和硬件可以完美结合，程序员和硬件开发人员可以一起庆祝了。

8.3 现成的软核

在本章前面，我们已经看到目前有很多可免费获得的处理器核能够综合到FPGA中。在下面一节中，我们将讨论如何使用我们自己的定制核，我们也可能使用来自几个厂家的商用核，而不仅限于主流FPGA厂商，为此，让我们花一些时间来考虑在写作本书时市面上的软核：

- Altera Nios II是一个为Altera FPGA优化过的32位RISC处理器，基于原始的Nios核。很多人认为，这款软核是针对广为应用的ARM处理器而推出的。该软核执行单一指令集，并且可以在绝大多数见类的FPGA上应用（700个逻辑单元以上）。最大、最快的配置是一条六级流水线，独立的数据cache和指令cache，专用的乘法器，分支预测单元，甚至还有可选的除法器及MMU。性能非常强大。从一个嵌入式系统体系结构设计师的角度来

看，其最有价值的地方是该核支持256条定制指令来访问以定制逻辑编写的专用模块，并且可以支持拥有流水线的专用硬件加速器。该软核支持一系列的操作系统，包括嵌入
377 式的Linux。

- Xilinx MicroBlaze也是一个32位的RISC处理器，是针对Xilinx的设备而开发。它有一个可选的三级或者五级流水线，并且有很多根据总线、功能单元以及MMU等来进行配置的选项。该软核为哈佛体系结构（指令存储和数据存储相互独立），可以对cache大小进行配置。拥有硬件除法器、快速乘法器和与IEEE754标准兼容的FPU。如Nios一样，该软核也支持一些操作系统，包括嵌入式的Linux。
- Actel是较晚加入软核家族的，起初也没有针对上面两家大型厂商推出产品，但后来和ARM签订了一项非常轰动的协议，ARM公司授权该公司销售基于ARM7的软核。该核的应用范围非常广泛，并且代码可以运行在ARM7上。然而，相比Altera和Xilinx，Actel是一个非常小的供应商，并且它瞄准的是不同领域的FPGA市场。在现成的微处理器市场上，ARM公司的成功是显而易见的，只有时间可以告诉我们这样的成功能否在FPGA软核市场上延续。
- 作为最后的竞争者，Lattice也发布了自己的32位软核RISC处理器——LatticeMico32，该软核处理器在一款Lattice的FPGA中使用了不到2000个查找表（LUT），尽管在可配置性上它没有Xilinx和Altera完整，其性能也不如这两家公司的产品，但是LatticeMico32速度更快、占用的资源也更少。该软核支持各种外设（如UART）和总线接口并且可配置。而且，它是完全开放的，这意味着可以在任何地方使用和更改该软核。使用和重新设计后出售该软核不需要任何许可。

除了以上这些处理器，还有些公司也专注于IP核市场，出售可以用于任何FPGA的核。甚至ARM公司自身也发布了一款很小的软核RAM Cortex。很显然，这是一个很活跃的领域，对于嵌入式系统市场愈来愈重要。

最后一点值得注意：这些核并不是孤立存在的。为了可以使其完全发挥作用，我们可以看见这些核还需要在一块FPGA上进行综合，需要外部总线、外设（比如存储器）、时钟信号及其他设施。此外，它们还需程序。

为软核处理器开发软件程序是一个必不可缺的部分，这样才可以确保设计出来的软核能正确工作。因此，有很重要的事情需要确定，比如是否有现成的工具链（用来开发软件），是否有该软核可用的现成操作系统，是否有调试工具可用。

一个标准的嵌入式工具链（toolchain），比如GNU工具链，包括以下几个要素：C(C++)编译器、汇编器和连接器。同时，也应该有一个库管理工具、目标文件工具、剥离器（用来从目标文件中移除调试注释以减少文件大小）和分析工具等。作者非常推崇的调试器是GDB，
378 它可以单步执行，设置断点、观察点和运行监控代码。GNU工具链甚至还包含了收集运行代码信息的软件，比如确定消耗在各个功能上的CPU时间、程序轨迹、所执行的循环次数等。

一个操作系统，特别是实时操作系统（RTOS），通常都会出现在开发项目的需求中。不幸的是，我们很难编写或移植一个操作系统到新的处理器上，这也许就是很多人在选择处理器时赞成那些支持成熟操作系统（比如嵌入式Linux）的处理器的原因。尽管如此，我们也有很多理由选择软核，例如，只需要很小规模的代码的时候，比如手工编写的汇编程序。

事实上，在下面几节中，我们将构建一个定制的软核，并在随后为该软核开发一个汇编器（我们也要介绍一个基于类C的编译器）。

8.4 制作自己的软核

在本节及接下来的内容中，我们将通过设计一个简单的软核，把所学的知识整合在一个点上。我们将使用一个用 Verilog 编写的软核，该软核可以在 FPGA 上应用。我们所使用的 CPU 叫做 TinyCPU，是由日本广岛大学工程学院信息工程系的 Koji Nakano[⊖]教授发明的。TinyCPU 的 Verilog 源代码仅有 420 行。

尽管设计该软核的目的是为了教学以及演示最基本的计算机体系结构特征，但是 TinyCPU 是一个拥有完整功能的 CPU。由于该软核用 Verilog 编写，因而它可以运用在绝大部分的 FPGA 上，比如来自 Altera、Xilinx 和 Actel 的 FPGA，并且可以执行实际的程序任务。Koji Nakano 教授和他的团队同时也为该软核发布了一个简单汇编器和一个支持 C 语言的编译器。

对于正在寻找在其 FPGA 上可应用的软核处理器的读者来说，TinyCPU 也许会是一个非常不错的选择。但是，对于这些读者来说更重要的是要首先理解 TinyCPU 并且对其做一些实验：为什么不扩展 TinyCPU 或者运用自己所学的知识重新设计或者选择一个定制的处理器核，而是将 TinyCPU 直接应用于项目中呢？TinyCPU 也许不是最有效率的，或者对于特定应用来说也许不是最适合的，但是运用本章所讲述的 CPU 设计知识并结合前面几章中的基础知识，读者已经具备了一些基本技巧以创建定制化软核处理器方案，或者从现有方案中选择合适的软核。

值得注意的是，对一些设计来说选择一个较为通用的处理器核效果会更好，即便该核明显属于次优选择。这样做可能有几方面的好处：代码/库的复用，共享的开发工具，共享的开发技 379
巧和知识。将一个处理器的开发经验应用于另外一个全新的处理器，会因较长的学习时间而延长开发周期，这也是为什么大家坚持选择一个成熟的、标准的处理器而不是去设计一个定制化处理器核的原因。

对功耗/空间效率或者性能这两个参数标准有严格要求的人来说，一个全定制的处理器核也许是最佳选择。

本章会一步步地讲述对 TinyCPU 的完整 CPU 设计。如果读者对硬件描述语言（HDL）不熟悉的话也没有关系，所有的设计特征都会以节的形式分块阐述。事实上，在学习这些 CPU 设计知识时，也是一个轻松构建 Verilog 基本知识体系的过程。[⊜]

8.5 CPU 设计规格说明

本章中所设计的 CPU 显然是一个教学工具。然而，它也必须是一个拥有全部功能的系统。让我们先定义一些 TinyCPU 的关键特征：

- 一个全功能的 CPU，可以在 FPGA 上综合。
- 在保证操作正确性的前提下应尽量简单。
- 所需的 Verilog 源代码数量应该最小。[⊜]
- 可以用汇编语言进行编程（最好用 C 编程语言）。
- 有一个简单但是功能完整的指令集。
- 有一个至少 16 位的体系结构。

⊖ 承蒙 Nakano 教授好意，允许我们在这里使用 TingCPU 的源代码和设计。关于 TingCPU 的更多信息，请见他的 HDL 维基网页 http://www.cs.hiroshima-u.ac.jp/~nakano/wiki/。

⊜ 也可以参见在《Design Wave Magazine》（2007～2009）中刊登的 K.Nakano 和 Y.Ito 的连载文章“VerilogHDL & FPGA design learned from basics”。

⊜ 尽管本书作者本身是 VHDL 的长期用户和拥护者，但似乎许多教师现在认识到 Verilog 更容易掌握，而且对于初学者是一种更“友好”的语言。

- 支持输入/输出。
- 支持一般的条件语句操作（比如NE，GZ，EQ等）。
- 采用简洁的非线性系结构（见3.3.5节）。

根据上面所罗列的特征，现在我们可以从逻辑上定义并描述CPU结构和操作。当然，在任何工程问题中都有一些不同的解决方案，在这里我们以TinyCPU的设计为例来讲述。

本章的其他部分将按顺序介绍TinyCPU设计。我们首先考虑CPU的体系结构，在提出Verilog设计之前讨论指令的处理和控制。读到这里，读者往往更倾向于跳到8.7节来查看和验证我
380 们的设计，然后再回过头来阅读8.5.1节到8.6.1节关于分析设计选择的内容。

8.5.1 CPU体系结构

回顾第3章，在3.2节中我们可以看到一个计算机或者CPU只不过是一种设备，这种设备可以转换信息（数据），并根据这些数据执行逻辑操作，这些操作均是根据一系列的指令来进行的。

如果我们设计一个CPU，那么至少需要四个元素：第一，一些信息传输的方法；第二，一些存储数据和程序的方法；第三，一些执行逻辑操作的方法；第四，让这一系列指令能够具体化为操作和传输的方法。让我们先了解一下这些元素，然后从8.6.1节开始学习具体的Verilog代码结构。

8.5.2 总线

这是我们的CPU设计实例中所要求的第一条，数据传输的方法，也叫做总线。在第4章中我们曾对总线的结构做了全面的探讨，这些探讨同时也包括了总线结构对指令集设计和数字的潜在影响。在这里，我们以最简单的总线安排为起始点，我们称之为单总线体系结构（详见4.1.6节）。

TinyCPU因此名一个单数据总线。在这里，总线的宽度不最得特别重要，但显然一切和数据处理相关的部分都将连接到这条公共总线上。TinyCPU的总线结构如图8-4所示，而且这条总线会在我们的设计过程中随着额外的功能单元和连接的增加而扩充。



图8-4 TinyCPU为部单总线结构框图，显示了它的数据总线dbus，一个输入端口，输出缓冲寄存器。在这里省略了控制逻辑

在TinyCPU中，主要的数据总线被命名为dbus，并且为16位宽以匹配设计要求。总线的宽度影响着为实现设计所要求的CPU资源，并且如果要求提供一个立即数载入操作，则对指令集
381 会有一个额外的影响，但是除了这些以外，目前这个阶段这个问题相对来说不是很重要。

我们也提到了这个设备的输入/输出。很显然，输入和输出数据宁需要在dbus上传输。这个结构如图8-4所示。另外一些总线仲裁的解释应该事先给出。

因为输入in来自外部驱动进入总线，所以总线上的电压信号可以随时进入CPU。这明显会扰乱CPU的正常操作，所以有必要在输入和dbus之间设置不同类型的关口来避免此类问题。这个问题可以用缓冲单元来解决。设计一条具体的CPU指令，这样程序员可以从输入线上读取逻辑值。这条指令将打开连接输入线和dbus之间的缓冲单元。这样输入信号将进入总线，在这里一些其他逻辑将这些信号存入某个地方，具体逻辑未在图中给出。

类似地，一个具体的CPU指令将数据总线上的内容输出到输出线上。这条指令仅在非常短暂的时间内有效，它激活输出缓冲（obuf0），让其从dbus上读取逻辑值，待数据读取完毕立即关闭输出缓冲。

显然，尽管我们目前定义了输入、输出和数据传输的主要设计，但依然还有大量的CPU设计没有讲述。

8.5.3 程序及数据存储

对于用来控制最终CPU行为的一系列指令来讲，一些形式的程序存储器是必需的。正因为这样，和几乎所有的计算机一样，我们假设这里程序采用的是二进制机器代码指令。同样，与目前大多数处理器特别是RISC处理器一样，为简便起见，我们将采用固定指令长度设计。由于该特性，所有的指令可以直接写在Verilog源代码中。

在一个真实的FPGA实现中，设计者更倾向于使用外部存储器（如SRAM、SDRAM、闪存之类）。在那样的情况下，TinyCPU的程序将存储于外部存储器中，并且通过一条与FPGA相连的总线进行传输。如果程序代码的规模超过了FPGA中较小的专用程序存储空间，那么这种方法是有必要的。在写作本书的时候，FPGA可支持的最大程序存储空间大约为1MB。然而在我们这个实例和许多嵌在小型嵌入式系统中，存储需求很少有超过32KB的。一个专用的模块完全可以满足我们对程序存储和数据存储的需求，这个专用模块我们称之为RAM0。

像外部专用存储设备一样，RAM0也是可寻址存储器。因此除dbus外它还需要附加一个地址总线（abus），并且要加上读写控制信号。对于程序存储器为什么是可写的，没有一个具体的理由，但确实需要为变量读写存储器。在这里，我们将使用冯·诺依曼方法（见2.1.2节），即程序代码和数据存储于相同的存储器。

还有另外一个数据存储元素没有提及，那就是堆栈。我们在3.3.5节中简要描述了堆栈机器，它用来在程序执行中临时存放变量。事实上，这个例子就表明了一个堆栈和一个ALU之间的联系。 232

将数据加载进一个堆栈需要将堆栈的输入与dbus相连。一个堆栈需要两个输出来供应逻辑操作（因为这些操作通常都有两个参数），而这两个输出分别是堆栈顶的两个数据。

我们定义一个堆栈为stack0，它挂在dbus上，同时它的输出为其顶部的两个数据。如图8-5所示，ram0、地址总线abus以及刚刚添加的stack0，现在这些都已经加入到了CPU设计中。

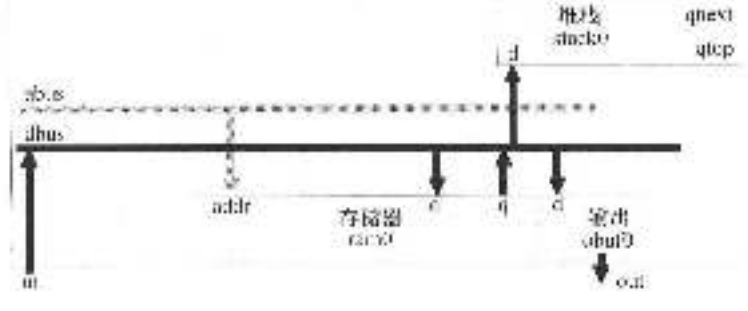


图8-5 包含存储器、堆栈和输出端口的TinyCPU框图

注意图中，我们使用数字逻辑中的通用设计，字母 d 代表数据输入到一个模块，而字幕 q 代表数据输出。值得注意的是堆栈的输出 qnext 和 qtop 也将最终连接到 dbus 上。然而，当 ALU 连接到系统时，这一细节就会立即添加进来。

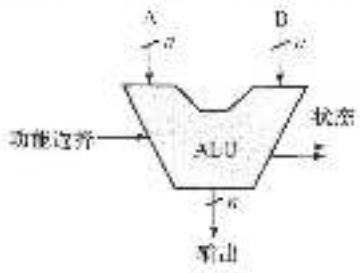


图 8-6 TinyCPU 内连接的通用算术逻辑单元（ALU）模型

8.5.4 逻辑运算

4.2 节已经从功能和设计两方面介绍了 ALU，并展示了它所提供的逻辑和算术运算。显然，这个 CPU 中需要一个 ALU 进行逻辑运算，如图 8-6 所示。

在堆栈体系结构系统中，ALU 的 A 和 B 输入总是从栈顶两
383 项取数（当前设计命名为 qtop 和 qnext），输出也总是反馈到堆栈。在 TinyCPU 中堆栈输入通过数据总线传送，因此，ALU（alu0）输出与主总线相连。如图 8-7 所示，栈顶也直接和数据总线相连，以便在需要时把数据直接反馈到数据总线。

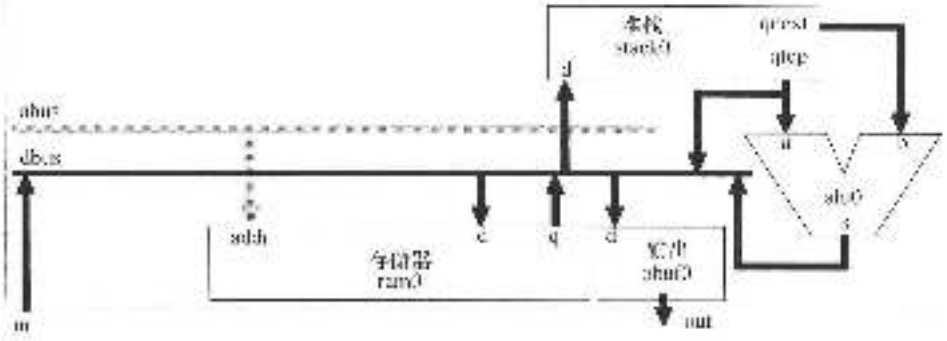


图 8-7 包含一个连接到内部总线和堆栈的 ALU（alu0）的 TinyCPU 框图，ALU 输入端口标记为 a 和 b，输出端口标记为 s

我们还需要收集 ALU 的状态输出，以决定随后的条件指令的需要，并且需要指定 ALU 对从堆栈来的数据执行什么功能的操作。这些连接没有包含在系统框图中，但值得记住的是，连同用来控制每个锁存器和缓冲器（用于伸载与总线的连接）的信号一起，我们也需要这些信号。

我们后面将看到 TinyCPU 实际上比 4.2.2 节讨论的基本类型的 ALU 智能一点，这涉及条件指令在 TinyCPU 中的处理方式。更智能并不影响图 8-7 中的数据通路，而是影响控制信号。

8.5.5 指令处理

我们在前面提到指令位于 ram0 中，这些指令都是二进制机器码。指令在内存中根据它们的地址辨别，并且如同大多数 CPU 一样，程序计数器（pc0）保存执行的下一条指令的地址。当从 RAM 中访问下一条指令时，pc0 将会驱动地址总线。当程序跳转到一个新的地址时，这个地址的值需要重载到程序计数器，这样除了驱动地址总线，pc0 还可以从地址总线上加载。

由于 ram0 的数据通过数据总线输出，CPU 中负责保存、解码和执行每条指令的单元理所应当要能从相同的数据总线加载指令。这个单元就是指令寄存器（ir0）。有时指令中会包含立
384 即数或者分支目标地址值，它们将分别通过数据总线和地址总线被送到 stack0 和 pc0 中。因此，指令寄存器在适当的时候需要有向数据总线或者地址总线输出的能力。这种结构如图 8-8 所示。

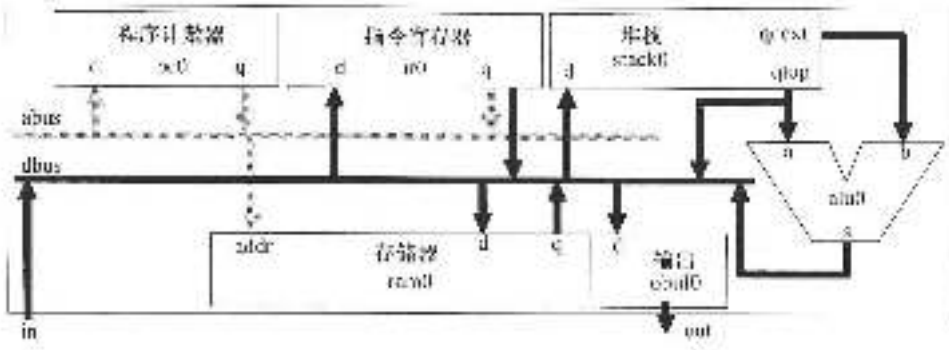


图 8-8 加入指令寄存器（ir0）和程序计数器（pc0）后的 TinyCPU 全图

8.5.6 系统控制

TinyCPU 基本上已经完整了，至少就数据通路和地址通路来说我们都考虑到了。然而，对 CPU 的运行来说有几件事仍然是必需的，即需要控制总线来开关锁存器和缓冲器（为清晰起见，我们此处省略）以及一个控制器通过控制总线来读出 CPU 中的操作顺序。

图 8-9 中包含了额外的一个模块——状态机控制器（state0）。图中并没有显示它与 CPU 中其他单元的连接。它既实不与数据总线或者地址总线相连，然而，它却几乎广泛地与 TinyCPU 中的每个单元和总线驱动器都相连。 [585]

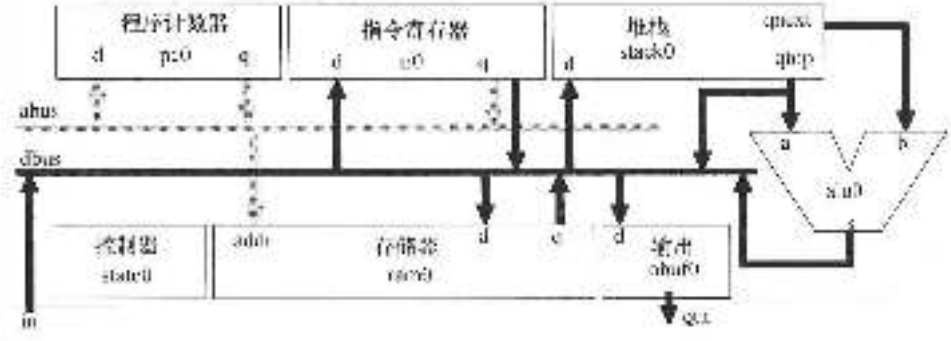


图 8-9 TinyCPU 内部结构及连接关系完整视图，包含除控制信号外的所有部分

我们将在 8.6.1 节研究这个 CPU 中的操作顺序以及控制方面的内容，不过，我们有必要首先讨论它的指令集，因为它将严重影响到控制系统，并在一定程度上影响体系结构细节。

8.6 指令集

显然 TinyCPU 将功能主要是算术和逻辑，因此这些指令在指令集中占有较大比重。8.5 节要求 CPU 有一个功能齐全的可用指令集，所以条件和控制流操作也要被包含进来。

表 8-1 对 TinyCPU 支持的指令进行编码。可以看出指令集是很有规律的，前 4 个最高有效位决定了指令类型，任何直接或者绝对地址依被编码在低 12 位。下面我们依次看一下每一条指令。

表 8-1 TinyCPU 目前支持的 10 个可能的 16 位指令类型

助记符	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	十六进制
HALT	0	0	0	0	X（无关）												0XXX

（续）

助记符	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	十六进制
PUSHI I	0	0	0	1	I（有符号整数）												1000 + I
PUSH A	0	0	1	0	A（无符号整数）												2000 + A
POP A	0	0	1	1	A												3000 + A
JMP A	0	1	0	0	A												4000 + A
JZ A	0	1	0	1	A												5000 + A
JNZ A	0	1	1	0	A												6000 + A
IN	1	1	0	1	X												D000
OUT	1	1	1	0	X												E000
OP f	1	1	1	1	X							f					F000 + f

- HALT 指令被用来终止处理器操作。这在程序终止时是合法事件，而且它是一种安全功能，可以在一个意外的程序分支使 CPU 跳转到其程序代码之外时发挥作用。在这种情况下，跳转到未初始化的内存可能导致一条“指令”被读入，其值为数字值 0——这样使处理器终止而不是错误地继续操作。
- PUSHI I 指令把立即数压入堆栈。I 的数值可以是以 12 位表示的最大值，而机器（包括堆栈）是 16 位宽的。这样，对压入堆栈的立即数首先进行符号扩展，否则就不可能把负值压入堆栈。
- PUSH A 指令检索 ram0 中存储位置 A 的内容并把它压入堆栈。

386

- POP A 指令执行相反的操作，从栈顶弹出数据并把它保存到 RAM 的地址 A 中。
- JMP A，JZ A 和 JNZ A 指令分别表示总是跳转到地址 A 执行下一条指令，或者当栈顶弹出的数据项是 0（JZ）或非 0（JNZ）时跳转到地址 A 执行下一条指令。跳转的过程涉及被编码成机器码程序和保存在程序计数器中的绝对地址 A。请注意除了 NZ 和 Z 外，跳转不支持其他条件，因而，必须以其他的方式（后面将讲到）支持大范围的传统条件（如 GT、LE 等）。
- IN 指令读取输入端口并把值压入堆栈。
- OUT 指令弹出栈顶值并把它锁存到输出缓冲中。
- *OP* f 本身并不是一条指令，它是一类指令。这些指令使 ALU 执行 f 中的编码所要求的功能。由于 ALU 与栈顶的两项相连，因此该功能可以使用这些堆栈值。

OP f 指令类目前编码了 19 个独立的操作（尽管保留了 5 位用来区分 f 位域的操作，但最多可以额外添加 13 个操作）。大部分操作通过它们的助记符就可以明白它们的功能：其中 16 个涉及双操作数（来自栈顶 qtop 和次栈顶 qnext），因此将输出结果写入堆栈前应先把堆栈弹出。

有三个操作是一元的，即仅仅从栈顶 qtop 取数进行操作，然后再返回到堆栈。在这种情况下，弹出是没有必要的，因为在一元指令中用到的栈顶的单个值将会直接被结果覆盖。

表 8-2 给出了 TinyCPU 目前提供的数据操作。

表 8-2 TinyCPU 的 *OP* 指令类格式：算术、逻辑、单一乘法指令及较不经常使用的比较指令

助记符	4	3	2	1	0	十六进制	栈顶内容	弹出?
ADD	0	0	0	0	0	F000	next + top	Y
SUB	0	0	0	0	1	F001	next-top	Y
MUL	0	0	0	1	0	F002	next * top	Y

（续）

助记符	4	3	2	1	0	十六进制	栈顶内容	弹出?
SHL	0	0	0	1	1	F003	next >> top	Y
SHR	0	0	1	0	0	F004	next << top	Y
BAND	0	0	1	0	1	F005	next&top	Y
BOR	0	0	1	1	0	F006	next | top	Y
BXOR	0	0	1	1	1	F007	next^top	Y
AND	0	1	0	0	0	F008	next&&top	Y
OR	0	1	0	0	1	F009	next || top	Y
EQ	0	1	0	1	0	F00A	next == top	Y
NE	0	1	0	1	1	F00B	next ! = top	Y
GE	0	1	1	0	0	F00C	next >= top	Y
LE	0	1	1	0	1	F00D	next <= top	Y
GT	0	1	1	1	0	F00E	next > top	Y
LT	0	1	1	1	1	F00F	next < top	Y
NEG	1	0	0	0	0	F010	- top	N
BNOT	1	0	0	0	1	F011	~ top	N
NOT	1	0	0	1	0	F012	! top	N

几个逻辑比较（AND，OR，EQ，NE，GE，LE，GT，LT，NOT）会在比较结果为真时把0值压入堆栈，比较结果为假时把非0值压入堆栈。这样，如果A大于B就跳转到子程序可以用下面的指令序列实现：

```
PUSHI valueA
PUSHI valueB
EQ
JZ subroutine
```

观察上述指令集可以发现一些结构性的机会和限制，正如它在任何其他CPU中做的那样。

首先，考虑扩展TinyCPU的可能性。我们已经注意到几种可能的f位组合还没有被使用——最多可以额外添加13个操作。以同样的方式，机器码指令集中4个最高有效位可以编码16种可能的变化，但只有10个使用到了，因此最多可以额外添加6个操作。

在机器码字中的输入和输出指令只需要4位：低12位（目前在这些指令中还未用到）可能指定进一步的信息，例如允许输出立即数，允许输出和输入指定内存地址中的数据。另外，可以支持多个输入/输出端口，或者可以把指令变成条件式。 387

因此在指令集内有很多机会进行进一步扩展，可以提供单总线堆栈体系结构支持的任何新指令。这样，给定体系结构支持什么指令本质上就成为该指令需要什么操作数和功能单元的问题了。这属于控制系统领域。

8.6.1 CPU控制

3.2.4节介绍了CPU的控制单元，它编织了互连的控制信号及时序单元的网络。即使在像TinyCPU这种简单的处理器中这也是一个潜在的复杂问题。在这种情况下，为了保持简洁性，一个非常简单的状态机控制器将用于CPU操作的同步。这个基本状态机在模块state0中实现，如图8-10所示。

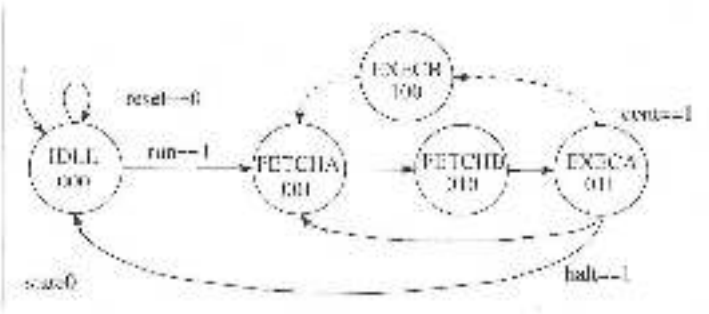


图8-10 TinyCPU中使用的状态：state0中的状态转换及3位状态编码

8.6.1.1 闲置状态

刚启动时CPU处于IDLE状态，意味着CPU没有进行任何操作。这样做有几个原因，尤其是连接外部设备（如内存）时需要等待这些存储设备变为可用，操作才能得以进行。

f068 run信号使CPU开始正常执行操作，从内存中取第一条指令。一旦CPU进入正常操作模式，它只能一次执行一条指令。只有从RAM发出的HALT指令（或无法识别的指令）才会使CPU重新进入IDLE状态。这通常表示一个程序的终结。

8.6.1.2 取指状态

FETCHA是正常操作中遇到的第一个状态。FETCHA使程序计数器（pc0，初始化设置为0）中的地址输出到地址总线dbus。然后ram0模块查找这个内存地址中的内容，一旦发现ram0将会输出该内容。

FETCHA将自动在一个时钟周期后进入FETCHB状态来结束取指进程。在FETCHB状态，在ram0中找到的存储内容将被允许来驱动数据总线dbus，ir0会对其进行锁存。这时，寄存器ir0将包含pc0指向的ram0地址中的指令机器码。

事实上，程序计数器pc0在从状态FETCHA转换到FETCHB时会自动增加。

从状态FETCHB经过一个时钟周期后，就进入了下一个执行状态：EXECA。

8.6.1.3 执行状态

通过首先执行指令解码功能，EXECA开始了指令的执行。注意表8-1中给出的机器码指令位域，在状态EXECA，只需通过检查指令的前4位来识别指令类。

根据不同的指令类，驱动各种总线的值也有如下不同：

- PUSHI指令——数据总线被ir0中的低12位驱动（将立即数压入堆栈），这个立即数的
f389 符号位将被扩展成数据总线上的前4位。
- PUSH，POP，JMP，JZ，JNZ——它们每一个指定一个绝对地址，这样ir0中的低12位用来驱动地址总线。而数据总线或者被用来查找RAM，或者被加载到程序计数器。
- IN让输入线来驱动数据总线，然后再把它压入堆栈。
- OUT让堆栈来驱动数据总线，然后弹出堆栈。接下来，它会告诉输出缓冲锁存数据总线上的逻辑值。
- OP指令类要求栈顶两项进入ALU，ALU被告知执行什么功能，ALU的输出驱动数据总线，数据总线的值被压入堆栈。在指令使用双操作数的情况下，当堆栈驱动ALU时必须弹出堆栈（这很容易实现，因为所有一元指令都有标识：ir0_4_=1，如表8-2所示）

对于其些指令，状态EXECA之后需要跟着一个延续状态EXECB，但在其他时候，CPU会准备执行下一条指令，从而转换回FETCHA。凡在ir0中存储的指令为0，就表明HALT指令已经被执行过了，所以在这种情况下CPU必须转换到IDLE状态。

对于这里所介绍的CPU设计，延续状态（EXECB）只对PUSH指令是必要的。原因是需要有内存查找作为指令的一部分（就像指令加载内存查找时需要两个取指状态一样）。记住PUSH A是把内存地址A中的值压入堆栈。在进入状态EXECA时，指令已经被加载到ir0中。在状态EXECA，内存地址A的值从指令寄存器输出到地址总线。然后ram0模块查找这个内存地址中的内容，但不能立刻在数据总线上返回这个值，因为从存储器阵列中检索需要花费一些时间。因此，当RAM在数据总线上返回这个值时第二个执行状态EXECB发生了，同时指示堆栈把当前数据总线上的内容压入堆栈。

EXECB状态后紧跟的始终是下一条指令的取指状态：FETCHA。

8.7 CPU实现

由于TinyCPU将用Verilog实现，因此实现的第一步就是有一个可用的Verilog编译器。从Altera和Xilinx上免费下载的FPGA设计工具（在编写本书时分别命名为Quartus和ISE）是适合的。然而，两者的安装和使用入门都是庞大而烦琐的工作。在一般情况下，作者更倾向于只用ModelSim对VHDL和Verilog进行仿真和测试。然而，由于这个工具可能无法向许多读者免费提供[⊖]，因此附录E介绍了一个轻量级的开源替代品。 390

在本书中，TinyCPU将只在仿真环境中实现和测试。然而，一旦仿真工作完成，将它移植到硬件平台上的FPGA中是很容易的。同样，读者可以到Nakano教授的网站，在那里可以找到CPU映射到一个Xilinx Spartan-3E入门套件的实例，包括输入键的驱动程序以及LED和LCD两个输出代码。

8.7.1 测试的重要性

HDL代码通常以模块化的方式开发。这些模块有明确定义的输入/输出端口，并执行特定的功能，它们相互独立，只通过这些输入和输出相互作用。

开发代码时，通常一次编写一个模块并按照它的接口定义进行测试。只有经过测试且可以正常工作的模块才能被并入到更大的设计中。在强层次化的设计结构中，每个模块可能进一步包括多个模块。

测试模块的方法是创建一个测试平台（testbench）。熟悉C编程的人可以把模块想象成一个main函数，给模块提供各种输入参数，然后研究模块运行时不同的输出参数。一个好的测试平台对预期行为编码，然后把实际的输出与预期行为作比较，找出它们之间的差异。

这样，如果合并在较大设计中的一个模块日后做了轻微的修改，仍然可以将它“插”回到它自己的测试平台，在隔离状态下对它进行行为验证。

我们要确保整个设计过程都提供测试平台。然而，现在的建议是在阅读接下来的章节时认为设计过程只涉及主要Verilog源代码。随后，在8.8节中，我们将研究使用测试平台来验证代码的过程。

8.7.2 定义操作和状态：defs.v

首先，在开始任何逻辑定义之前，有必要定义一种在逻辑块之间使用的通用位定义语言。在8.6节中，我们介绍了指令集，以及每个机器码指令类中每个位的定义。为了增加TinyCPU代码的可读性，我们可以用名称以及与汇编助记符一致的位定义来定义一些常量。同样，8.6.1节介绍了处理器中使用的各种运行状态，这些状态也可以用名称定义。

⊖ 在编写本书时，有一个免费的6个月的学生模拟版可用。

所有的这些常量定义使我们能够创建一个包含在所有其他 Verilog 源文件中的头文件，就像我们在 C 编程语言中使用 .h 文件一样。这些定义存储在一个名为“defs.v”的文件中，

程序清单 8.1 再现了这个文件。

程序清单 8.1 defs.v

```
1  `define IDLE   3'b000
2  `define FETCHA 3'b001
3  `define FETCHB 3'b010
4  `define EXECA  3'b011
5  `define EXECB  3'b100
6
7  `define  ADD  5'b00000
8  `define  SUB  5'b00001
9  `define  MUL  5'b00010
10 `define  SHL  5'b00011
11 `define  SHR  5'b00100
12 `define  BAND 5'b00101
13 `define  BOR  5'b00110
14 `define  BXOR 5'b00111
15 `define  AND  5'b01000
16 `define  OR   5'b01001
17 `define  EQ   5'b01010
18 `define  NE   5'b01011
19 `define  GE   5'b01100
20 `define  LE   5'b01101
21 `define  GT   5'b01110
22 `define  LT   5'b01111
23 `define  NEG  5'b10000
24 `define  BNOT 5'b10001
25 `define  NOT  5'b10010
26
27 `define HALT  4'b0000
28 `define PUSHI 4'b0001
29 `define PUSH  4'b0010
30 `define POP   4'b0011
31 `define JMP   4'b0100
32 `define JZ    4'b0101
33 `define JNZ   4'b0110
34 `define IN    4'b1101
35 `define OUT   4'b1110
36 `define OP    4'b1111
```

8.7.3 第一个小模块：counter.v

由于 TinyCPU 在它的设计内部包含许多寄存器和缓冲区，因此使用一个标准组件来实现这些项目是有意义的。这样做的第一步工作就是弄清楚它们需要做什么。首先，各寄存器需要存储二进制字，在总线上输出，从其他总线上加载一个新值以及重置为 0。就程序计数器 pc0 来说，寄存器还需要在适当的时候递增地址（为获取下一条指令做准备）。

391
~
392

一些寄存器可能不会用到的功能：如果当模块实例化时有些硬连线处于非活动状态，Verilog 编译器会简单地忽略掉该功能的逻辑。

考虑到这些需求，一个满足要求的简单计数器的输入/输出规格说明如下所示：

标示	方向	含义	标示	方向	含义
clk	in	系统时钟	load	in	触发计数器存储当前输入数据总线上的逻辑值
reset	in	低电平复位，进入闲置状态	inc	in	增加存储逻辑值的信号
d	in	输入数据总线	q	out	输出总线，映射存储的逻辑值

请注意 TinyCPU 跟其他几乎所有的 CPU 一样是同步设计，从而系统时钟在它的设计中是一个主要部分，它定义了系统操作的时序，因此需要被送到几乎每个 Verilog 模块。除了时序因素外，计数器的 Verilog 源代码是很简单的，如程序清单 8.2 所示。参数 N 允许为不同的总线带宽定义 *counter.v* 模块。在这种情况下，我们有一个 16 位的体系结构，所以 N 被默认设定为 16。

如上所述，这个计数器将被实例化并在 CPU 设计的许多地方使用，包括以下单元：

- 程序计数器（pc0）
- 指令寄存器（ir0）
- 输出缓冲区（obuf0）

程序清单 8.3 提供了一个简单的测试平台来测试这个计数器。

程序清单 8.2　counter.v

```
1  module counter(clk,reset,load,inc,d,q);
2    parameter N = 16;
3
4    input clk,reset,load,inc;
5    input [N-1:0] d;
6    output [N-1:0] q;
7    reg [N-1:0] q;
8
9    always @(posedge clk or negedge reset)
10     if(!reset) q <= 0;
11     else if(load) q <= d;
12     else if(inc) q <= q + 1;
13
14 endmodule
```

程序清单 8.3　counter_tb.v

```
1  `timescale 1ns / 1ps
2  module counter_tb;
3  reg clk, reset, load, inc;
4  reg [15:0] d;
5  wire [15:0] q;
6  counter counter0(.clk(clk), .reset(reset), .load(load),
       .inc(inc), .d(d), .q(q));
7
8  initial begin
9    clk=0;
10   forever
11     #50 clk = ~clk;
12 end
13
```

```
14 initial begin
15   reset=0; load=0; inc=0; d=16'h0000;
16   #100 reset=1;
17   #100 inc=1;
18   #300 inc=0; load=1; d=16'h1234;
19   #100 inc=1; load=0; d=16'h0000;
20   #500 reset=0;
21 end
22 endmodule
```

计数器模拟、绘制波形以及比较 d 和 g 的值应该表现出加载、递增和复位功能以及输出 q。下一步，我们考虑的是 TinyCPU 中的操作序列以及如何控制它。

8.7.4 CPU 控制：state.v

在 8.5.6 节和 8.6.1 节中描述的状态机 state0 控制 TinyCPU 的操作。它有 5 个状态（因此需要 3 位进行编码），状态间的转换基于系统时钟 clk 再加上下面接口定义表中所示的几个控制信号：

标示	方向	含义	标示	方向	含义
clk	in	系统时钟	cont	in	表明需要再次执行状态的连续信号
reset	in	低电平复位，进入空闲状态	halt	in	终止 CPU 操作，但是回落到空闲状态
run	in	如果 CPU 处于空闲状态，触发 CPU 运行	cs[2:0]	out	3 位状态输出

这个模块只有单一的输出：当前处理器状态从 3 根 cs 线上反映出来。TinyCPU 中的操作序列完全由当前状态定义，所有项目必须在适当的时候协同工作，从而使系统正确地实现功能。在程序清单 8.4 给出的 state.v 的 Verilog 源代码中，注意状态间的转换使用的是 case 语句。

程序清单 8.4 state.v

```
1  `include "defs.v"
2
3  module state(clk,reset,run,cont,halt,cs);
4
5    input clk, reset, run, cont, halt;
6    output [2:0] cs;
7    reg [2:0]cs;
8
9    always @(posedge clk or negedge reset)
10     if(!reset) cs <= `IDLE;
11     else
12       case(cs)
13         `IDLE: if(run) cs <= `FETCHA;
14         `FETCHA: cs <= `FETCHB;
15         `FETCHB: cs <= `EXECA;
16         `EXECA: if(halt) cs <= `IDLE;
17                 else if(cont) cs <= `EXECB;
18                 else cs <= `FETCHA;
19         `EXECB: cs <= `FETCHA;
20         default: cs <= 3'bxxx;
21       endcase
22
23 endmodule
```

状态机的测试平台通过预期的输入控制信号 reset、run、cont 和 halt 的变化序列产生 3 位状态位的各种输出。程序清单 8.5 给出了一个最小的测试平台。

程序清单 8.5　state_tb.v

```
1  `timescale 1ns / 1ps
2  module state_tb;
3  reg clk, reset, run, halt, cont;
4  wire [2:0] cs;
5  state state0(.clk(clk), .reset(reset), .run(run), .cont(cont),
      .halt(halt), .cs(cs));
6
7  initial begin
8    clk=0;
9    forever
10     #50 clk = ~clk;
11 end
12
13 initial begin
14   reset=0; run=0; halt=0; cont=0;
15   #100 reset=1; run=1
16   #100 run=0;
17   #200 cont=1;
18   #100 cont=0;
19   #600 halt=1;
20   #100 halt=0;
21 end
22 endmodule
```

395

8.7.5　程序和变量存储：ram.v

正如 8.5.3 节所讨论的，RAM 模块在 TinyCPU 中有两个作用：首先，是作为程序代码的存储区域，它在 FETCH 状态时要被检索；其次，是作为变量的存储区域。TinyCPU 不包含寄存器组。相反，它依赖于堆栈结构。如果代码中用到的变量超过了堆栈的容纳量，或者命令中用到的大数据块或变量不一定方便用堆栈处理，那么这些变量将被存储到别的地方。因此，任何内存地址都能用来保存数据或者存储变量。

这意味着 ram0 应当是一个可读写的可寻址的存储器。基于这个目的，大多数现代 FPGA 结构中包含专用的 RAM。但是，默认的而且是最简单的方法是使用 FPGA 逻辑块中的触发器用于位编码存储。然而，这比较“昂贵”，因为虽然每个逻辑块或逻辑单元（注意对于这些 FPGA 中的最小可编程单位，不同的厂家有不同的名称）可能只包含一个单一的触发器，但它也将包含一个组合逻辑的配置块，各种缓冲区，可能还包含一个查找表，如果触发器用来存储内存中的某个位，那么其他资源大部分将被浪费。

RAM 模块与 CPU 的其他模块一样也是同步的，它需要一个与外界相连的很简单的接口：

标示	方向	含义	标示	方向	含义
clk	in	系统时钟	d	in	输入数据总线
load	in	触发计数器存储当前输入数据总线逻辑值的信号	q	out	存储在 addr 地址上的输出数据字
addr	in	地址总线			

396

程序清单8.6中给出的源代码以非常简单的方式实现了RAM模块。数据和地址位宽是可配置的，这里我们设定为16位的数据总线和12位的地址总线。整个4096字的内存区域被初始化为0，然后一些值被加载到前几个地址位置。

例如，注意加载12'h001那一行。这是第二个地址的位置，我们指定的值16'h3010将被初始化存储到这个位置上。从8.6节对指令集的讨论我们知道，机器码0x3010对应的是带参数0x10的POP指令。这个参数就是堆栈弹出的值要被送入的内存地址。在这种情况下，看一下地址12'h010的说明，我们注意到一个数值0已经初始化存储在这个位置（我们现在将覆盖它），并且从注释上看，我们这个位置用来存储变量n。

程序清单8.6 ram.v

```
1  module ram(clk, load, addr, d, q);
2   parameter DWIDTH=16,AWIDTH=12,WORDS=4096;
3
4   input clk,load;
5   input [AWIDTH-1:0] addr;
6   input [DWIDTH-1:0] d;
7   output [DWIDTH-1:0] q;
8   reg [DWIDTH-1:0] q;
9   reg [DWIDTH-1:0] mem [WORDS-1:0];
10
11  always @(posedge clk)
12    begin
13      if(load) mem[addr] <= d;
14      q <= mem[addr];
15    end
16
17  integer i;
18  initial begin
19     for(i=0;i<WORDS;i=i+1)
20        mem[i]=0;
21 mem[12'h000] = 16'hD000;        //      IN
22 mem[12'h001] = 16'h3010;        //      POP n
23 mem[12'h002] = 16'h2010;        //      L1:     PUSH n
24 mem[12'h003] = 16'hE000;        //      OUT
25 mem[12'h004] = 16'h2010;        //      PUSH n
26 mem[12'h005] = 16'h500F;        //      JZ L2
27 mem[12'h006] = 16'h2010;        //      PUSH n
28 mem[12'h007] = 16'h1001;        //      PUSHI 1
29 mem[12'h008] = 16'hF001;        //      SUB
30 mem[12'h009] = 16'h3010;        //      POP n
31 mem[12'h00A] = 16'hD000;        //      IN
32 mem[12'h00B] = 16'h1005;        //      PUSHI 5
33 mem[12'h00C] = 16'hF001;        //      SUB
34 mem[12'h00D] = 16'hE000;        //      OUT
35 mem[12'h00E] = 16'h4002;        //      JMP L1
36 mem[12'h00F] = 16'h0000;        //      L2:     HALT
37 mem[12'h010] = 16'h0000;        //      n: 0
38 end
39
40 endmodule
```

397

一个验证 RAM 操作的简单测试平台只需要读取一些预定义的位置，然后测试对数据变量的读写操作。程序清单 8.7 中所示的测试平台，首先读取 RAM 中预定义的一些指令，然后写入到变量位置 n。然后它从位置 n 回读。在读操作的过程中，应该对输出线 q 上的数据值加以验证，以确保它分别正确地与要求的机器码指令字及变量 n 的内容相匹配。

程序清单 8.7　ram_tb.v

```
1  `timescale 1ns / 1ps
2  module ram_tb;
3  reg clk, load;
4  reg [7:0] addr;
5  reg [15:0] d;
6  wire [15:0] q;
7
8  ram ram0(.clk(clk), .load(load), .addr(addr), .d(d), .q(q));
9
10 initial begin
11    clk=0;
12    forever
13       #50 clk = ~clk;
14 end
15
16 initial begin
17    reset=0; load=0; d=0;
18    #100 reset=1; addr=12'h000;
19    #100 addr=12'h001;
20    #100 addr=12'h006;
21    #100 addr=12'h010; load=1; d=8'h55;
22    #100 addr=12'h00D;  load=0; d=0;
23    #100 addr=12'h010;
24 end
25 endmodule
```

8.7.6　堆栈：stack.v

在 8.5.3 节讨论的堆栈模块，它的功能是负责存储数据，这些数据位于当前操作序列中——即从一个操作中输出的数据或者将要被一个操作所运用的数据或者是两者的结合。

TinyCPU 的堆栈支持标准的 POP 和 PUSH 操作。此外，它能够在栈顶载入一个值，覆盖栈顶数据，这种操作在一些特定情况中会用到。顶部和第二个堆栈项在输出 qtop 和 qnext 中总是可见的：

信号	方向	定义
clk	in	系统时钟
reset	in	清空堆栈
load	in	将当前数据总线 d 上的数据置于栈顶的位置，其他保持不变
push	in	将堆栈内所有项往下推一个单元：位于顶部的项移到第二个位置，原来的第二项移到第三个位置，以此类推。栈底项出栈。如果也设置了 load 指令，则当前位于数据总线 d 上的数据将只会进入栈顶
pop	in	将处于堆栈中第二个位置的项移到栈顶，其他依次上移
d	in	输入数据总线
qtop	out	处于栈顶的输出数据字
qnext	out	处于栈中第二个位置的输出数据字

以上所示的与外部连接的接口在功能上比 RAM 模块的多一些。但存储容量却较之小了许多。实际上，在程序清单 8.8 中所示的栈的深度仅为 8（由于 N = 8），不过如果有要求，我们可以将这个参数设置得更大一些。

程序清单 8.8 stack.v

```
1  module stack(clk, reset, load, push, pop, d, qtop, qnext);
2    parameter N = 8;
3
4    input clk, reset, load, push, pop;
5    input [15:0] d;
6    output [15:0] qtop, qnext;
7    reg [15:0] q [0:N-1];
8
9    assign qtop = q[0];
10   assign qnext = q[1];
11
12   always @(posedge clk or negedge reset)
13     if(!reset) q[0] <= 0;
14     else if(load) q[0] <= d;
15     else if(pop) q[0] <= q[1];
16
17   integer i;
18   always @(posedge clk or negedge reset)
19     for(i=1;i< N-1;i=i+1)
20       if(!reset) q[i] <= 0;
21       else if(push) q[i] <= q[i-1];
22       else if(pop) q[i] <= q[i+1];
23
24   always @(posedge clk or negedge reset)
25     if(!reset) q[N-1] <= 0;
26     else if(push) q[N-1] <= q[N-2];
27
28 endmodule
```

398 ~ 399

通过将数据压入栈内然后再将它弹出栈外的方法，我们可以测试栈的操作。一个更全面的测试就是检查不同顺序的 push 和 pop 操作。然而，由于这个例子中堆栈编码相对简单，我们将只是直接地按先 push 后 pop 的次序再加一个 load 操作来进行测试，如程序清单 8.9 所示。

程序清单 8.9 stack_tb.v

```
1  `timescale 1ns / 1ps
2  module  stack_tb;
3  reg clk, reset, load, push, pop;
4  reg [15:0] d;
5  wire [15:0] qtop;
6  wire [15:0] qnext;
7
8  stack stack0(.clk(clk), .reset(reset), .load(load), .push(push),
       .pop(pop), .d(d), .qtop(qtop), .qnext(qnext));
9
10 initial begin
```

```
11    clk=0;
12    forever
13      #50 clk = ~clk;
14  end
15
16  initial begin
17    reset=0; load=0; push=0; pop=0; d=0;
18    #100 reset=1; push=1; d=16'h1111;
19    #100 push=1; d=16'h2222;
20    #100 push=1; d=16'h3333;
21    #100 push=1; d=16'h4444;
22    #100 push=1; d=16'h5555;
23    #100 push=1; d=16'h6666;
24    #100 push=1; d=16'h7777;
25    #100 push=1; d=16'h8888;
26    #100 push=1; d=16'hEEEE;
27    #100 push=0; pop=1;
28    #100 pop=1;
29    #100 pop=1;
30    #100 pop=1;
31    #100 pop=1;
32    #100 pop=1;
33    #100 pop=1;
34    #100 pop=1;
35    #100 pop=0; load=1;  d=16'h1234;
36    #100 load=0; pop=1;
37  end
38  endmodule
```

400

8.7.7 算术、逻辑和乘法单元：alu.v

ALU，或者更适当地称为“算术、逻辑和乘法单元”，负责这几种运算的操作。在8.5.4节中，我们讨论了单总线堆栈结构的TinyCPU对ALU的要求，传统的ALU的模块符号也已经给出。然而，这一符号的形状和它的描述跟ALU的Verilog代码相差很大，我们接下来马上就会看到这一点。首先，我们将定义这个代码模块的输入及输出：

信号	方向	定义	信号	方向	定义
a	in	第一个输入操作码	f	in	多选线来定义不同的操作
b	in	第二个输入操作码	s	out	结果输出接口

表中显示了ALU中并没有时钟信号——这是异步操作。实际上，ALU操作中最慢的操作在最坏情况下的传输延迟将决定这个设备所支持的最大时钟频率。[⊖]

401

上文提到过，ALU的代码和它的传统符号有着很大的不同，但却和8.6节中提到的OP操作类似。将它与程序清单8.10的代码作比较，很容易看出这一点。

⊖ 注意，当运用外部RAM时，到外部RAM的load/store的操作会构成影响时钟频率的限制因素。

程序清单8.10 alu.v

```
1  `include "defs.v"
2
3  module alu(a, b, f, s);
4
5   input [15:0] a, b;
6   input [4:0]  f;
7   output [15:0] s;
8   reg  [15:0]   s;
9   wire [15:0] x, y;
10
11  assign x = a + 16'h8000;
12  assign y = b + 16'h8000;
13
14  always @(a or b or x or y or f)
15    case(f)
16      `ADD : s = b + a;
17      `SUB : s = b - a;
18      `MUL : s = b * a;
19      `SHL : s = b << a;
20      `SHR : s = b >> a;
21      `BAND: s = b & a;
22      `BOR : s = b | a;
23      `BXOR: s = b ^ a;
24      `AND : s = b && a;
25      `OR  : s = b || a;
26      `EQ  : s = b == a;
27      `NE  : s = b != a;
28      `GE  : s = y >= x;
29      `LE  : s = y <= x;
30      `GT  : s = y > x;
31      `LT  : s = y < x;
32      `NEG : s = -a;
33      `BNOT : s = ~a;
34      `NOT : s = !a;
35      default : s = 16'hxxxx;
36    endcase
37
38 endmodule
```

对ALU操作的测试并不困难：搭建一个简单的测试平台，一些信号被载入该模块的输
402 入a、b，选择合适的功能f，检测输出的正确性。注意ALU是异步操作，由于受传输延迟所限，不需要时钟。然而，测试平台是需要时钟的（否则所有的输入将会在0时刻快速地载入模块）。当CPU被用在实际项目中时，首先全面测试不同输入和功能的组合是一个比较好的方法。不过，在这里，我们只是简单地进行了一些功能的测试，如程序清单8.11所示。

程序清单 8.11　alu_tb.v

```
1  `timescale 1ns / 1ps
2  module alu_tb;
3  reg clk;
4  reg [15:0] a;
5  reg [15:0] b;
6  reg [4:0] f;
7  wire [15:0] s;
8
9  alu alu0(.a(a), .b(b), .f(f), .s(s));
10
11 initial begin
12   clk=0;
13   forever
14     #50 clk = ~clk;
15 end
16
17 initial begin
18
19         a=16'h0000; b=16'h1234; f=5'b00000;   //ADD
20   #100  a=16'h000A; b=16'h0100; f=5'b00010;   //MUL
21   #100  a=16'h1010; b=16'hFFFF; f=5'b01000;   //AND
22   #100  a=16'h0008; b=16'h1234; f=5'b00100;   //SHR
23   #100  a=16'h0003; b=16'h0100; f=5'b00011;   //SHL
24   #100  a=16'h0010; b=16'h0001; f=5'b00001;   //SUB
25   #100  a=16'h0000; b=16'h1234; f=5'b10010;   //NOT
26   #100  a=16'h0005; b=16'h0004; f=5'b01100;   //GE
27   #100  a=16'h0003; b=16'h0004; f=5'b01100;   //GE
28   #100 $finish;
29 end
30 endmodule
```

8.7.8　综合测试：tinycpu.v

TinyCPU 所执行的代码存储在其内部，通过两个接口和外面连接：输入接口和输出接口。然而，Verilog 模块需要由几个信号来启动运行。一是系统时钟信号；二是全局低电平有效的复位信号；三是使 CPU 开始操作的触发信号（称为“run”）。

由于 TinyCPU 是一个为教学目的而设计的研究型处理器，它的几个内部信号暴露在上层接口中。在由 Nakano 教授设计并在 FPGA 开发平台上实现的原始系统中，这些信号可以通过 7 段 LED 阵列来显示。

右表定义了所要求的信号和为了可视性被“带到”表面的信号。

信号	方向	定义
clk	in	系统时钟
reset	in	对于整个 CPU 的全局低电平有效复位信号
run	in	用于触发 CPU 运行的控制信号（如图 8-10 所示）
in	in	输入接口（可以用 IN 指令读取）
out	out	16 位的输出缓冲区
为了检测内部操作所设置的可视信号		
cs	out	表示当前 CPU 状态
pcout	out	12 位程序计数器
irout	out	指令寄存器内容
qtop	out	栈顶的值
abus	out	12 位内部地址总线
dbus	out	内部数据总线

403

最终的TinyCPU的源代码在程序清单8.12中给出。对于一个功能完整的16位CPU来说，这段代码不是特别长。整段源代码，其中包括所有的功能模块，少于500行。毕竟，它的名字中的前缀是“Tiny”！

程序清单8.12 tinycpu.v

```
1  `include "defs.v"
2
3  module tinycpu(clk, reset, run, in, cs, pcout, irout, qtop,
       abus, dbus, out);
4
5    input clk,reset,run;
6    input [15:0] in;
7    output [2:0] cs;
8    output [15:0] irout, qtop, dbus, out;
9    output [11:0] pcout, abus;
10   wire [15:0] qnext, ramout, aluout;
11   reg [11:0] abus;
12   reg halt, cont, pcinc, push, pop, abus2pc, dbus2ir, dbus2qtop,
         dbus2ram, dbus2obuf, pc2abus, ir2abus, ir2dbus,
       qtop2dbus, alu2dbus, ram2dbus, in2dbus;
13
14   counter #(12) pc0(.clk(clk), .reset(reset), .load(abus2pc),
       .inc(pcinc), .d(abus), .q(pcout));
15   counter #(16) ir0(.clk(clk), .reset(reset), .load(dbus2ir),
       .inc(0), .d(dbus), .q(irout));
16   state state0(.clk(clk), .reset(reset), .run(run), .cont(cont),
       .halt(halt), .cs(cs));
17   stack stack0(.clk(clk), .reset(reset), .load(dbus2qtop),
       .push(push), .pop(pop), .d(dbus), .qtop(qtop),
       .qnext(qnext));
18   alu alu0(.a(qtop), .b(qnext), .f(irout[4:0]), .s(aluout));
19   ram #(16,12,4096) ram0(.clk(clk), .load(dbus2ram),
       .addr(abus[11:0]), .d(dbus), .q(ramout));
20   counter #(16) obuf0(.clk(clk), .reset(reset),
       .load(dbus2obuf), .inc(0), .d(dbus), .q(out));
21
22   always @(pc2abus or ir2abus or pcout or irout)
23     if(pc2abus) abus <= pcout;
24     else if(ir2abus) abus <= irout[11:0];
25     else abus <= 12'hxxx;
26
27   assign dbus = ir2dbus ? {{4{irout[11]}},irout[11:0]} :
       16'hzzzz;
28   assign dbus = qtop2dbus ? qtop : 16'hzzzz;
29   assign dbus = alu2dbus ? aluout : 16'hzzzz;
30   assign dbus = ram2dbus ? ramout : 16'hzzzz;
31   assign dbus = in2dbus ? in : 16'hzzzz;
32
33   always @(cs or irout or qtop)
34     begin
```

```
35        halt = 0; pcinc = 0; push = 0; pop = 0; cont = 0; abus2pc
              = 0; dbus2ir = 0; dbus2qtop = 0; dbus2ram = 0;
              dbus2obuf = 0; pc2abus = 0; ir2abus = 0; ir2dbus = 0;
              qtop2dbus = 0; alu2dbus = 0; ram2dbus = 0; in2dbus = 0;
36        if(cs == `FETCHA)
37          begin
38            pcinc = 1; pc2abus = 1;
39          end
40        else if(cs == `FETCHB)
41          begin
42            ram2dbus = 1; dbus2ir = 1;
43          end
44        else if(cs == `EXECA)
45          case(irout[15:12])
46          `PUSHI:
47             begin
48               ir2dbus = 1; dbus2qtop = 1; push = 1;
49             end
50          `PUSH:
51             begin
52               ir2abus = 1; cont = 1;
53             end
54          `POP:
55             begin
56               ir2abus = 1; qtop2dbus = 1; dbus2ram = 1; pop = 1;
57             end
58          `JMP:
59             begin
60               ir2abus = 1; abus2pc = 1;
61             end
62          `JZ:
63             begin
64               if(qtop == 0)
65                 begin
66                   ir2abus = 1; abus2pc = 1;
67                 end
68               pop = 1;
69             end
70          `JNZ:
71             begin
72               if(qtop != 0)
73                 begin
74                   ir2abus = 1; abus2pc = 1;
75                 end
76               pop = 1;
77             end
78          `IN:
79             begin
80               in2dbus = 1; dbus2qtop = 1; push = 1;
81             end
82          `OUT:
```

```
83              begin
84                qtop2dbus = 1; dbus2obuf = 1; pop = 1;
85              end
86          `OP:
87              begin
88                alu2dbus = 1; dbus2qtop = 1;
89                if(irout[4] == 0) pop = 1;
90              end
91          default:
92              halt = 1;
93          endcase
94        else if(cs == `EXECB)
95          if(irout[15:12]==`PUSH)
96            begin
97            ram2dbus = 1; dbus2qtop = 1; push = 1;
98          end
99      end
100
101 endmodule
```

TinyCPU的代码逻辑性很强，因此很容易解释。在此，我们特别提出几点：

- 在为CPU引入定义文件和定义上层输入和输出之后，各个信号和总线才被定义。
- 上层实例化了之前讨论的所有模块（它们的测试平台除外），即counter.v、state.v、stack.v、alu.v和ram.v。计数器在这个设计中实际上用了三次，分别是作为程序计数器、指令计数器和输出缓冲区。
- 通过控制信号的指定将不同的总线和端口进行连接（例如，ir2dbus这个控制信号指定数据总线由指令寄存器中带符号扩展的低12位来驱动，这种驱动可能发生在PUSHI指令中）。
- 代码的主体是根据当前状态来执行的。在EXECA状态，操作就是根据当前指令寄存器中的指令来确定的（或者更准确地说，是根据指令寄存器的［15：12］位——这些位确定指令的实际功能）。

TinyCPU，如在这里和之前章节所定义的一样，是一个能够在FPGA上通过顶层设计进行组合的模块，并给它提供了一个时钟信号、一个复位信号和输入/输出的端口。一旦CPU开始运行（当run信号为“1”的时候），它会执行内部程序直到HALT指令被读取或者系统复位。

404 ~ 407

正如所写的代码那样，包括用于存储运行时变量和程序指令的易失性存储器在内的所有存储都已在其内部定义。也可以将CPU与外部的存储器进行连接。由于程序代码为内部定义，新的程序必须手动输入到Verilog源文件ram.v中，然后重新编译整个CPU。

当使用FPGA的时候，新程序的综合需要使用如Altera的Quartus II或者Xilinx ISE逻辑设计工具。在运行前，需要通过程序数据线将所输出的编程文件下载到FPGA中。整个周期可能很费时，有时需要超过一个小时的时间来完成一个复杂的设计，虽然基础的TinyCPU设计只需要几分钟。

相比之下，外部程序存储的使用，使得TinyCPU程序代码中的每一处改变，只需要在存储器中重新编程即可，而不需要重建整个Verilog设计。

在8.9节，我们将会探讨TinyCPU的编程和使用。在此之前我们将先检查整个CPU的测试和运行。

8.8 CPU测试及运行

到目前为止，我们已经从体系结构元素（8.5节）、程序运行（8.6节）以及讲述如何用Verilog实现（8.7节）等几个方面阐述了TinyCPU的设计。在8.7节里我们曾说过使用测试平台对设计进行测试的重要性，因此，我们对构成TinyCPU的6个Verilog模块（tinycpu.v、state.v、stack.v、ram.v、counter.v以及alu.v）都设计了一个小而基础的测试平台。

我们至今还没讨论过如何使用测试平台对系统进行测试。我们的测试将在模拟层次上进行，而不会在FPGA硬件层次上进行。在这里需要说明的是，在实际的工业开发环境中，这些代码都要被部署到FPGA中，因此不仅需要进行模拟层次上的测试，也需要进行FPGA硬件层次上的测试。Altera和Xilinx都提供了基于JTAG的工具（见7.9.3节），使得对来自运行硬件的测试向量的收集和插入成为方便的事情。在编写本书时，这些工具分别称为SignalTap/SignalTap II和ChipScope/ChipScope Pro。SignalTap II可以通过下载免费网页版Quartus II获得（只需开启TalkBack——它通过Quartus软件向Altera提供有关软件使用的信息）。在之前没有这些工具的时候，最常用的方法是从FPGA的I/O管脚上引出导线至数字存储示波器（DSO）或者逻辑分析仪上，然后将这些输出映射到所需测试的VHDL/Verilog代码的信号上。这种方法的劣势在于，测试不同的信号都需要重复编译。

正如上文所说的，我们讨论的重点是在模拟层次上。在FPGA上运行之前对于Verilog至少有两个层次的模拟。第一个是*功能模拟*，它评估代码的基本操作和逻辑的正确性。第二个是*时序模拟*，它考虑测试目标FPGA[⊖]上各条导线及组件的传输延迟。第一种模拟得到周期级精确的结 408
果。换句话说，如果有一个时钟用于系统的时钟同步，那么这个设备的操作就会按周期进行测试，每个周期的评估都与下一个周期相互独立。如果设计中带有组合逻辑，那么一旦输入发生改变，输出立即发生变化。第二种模拟产生的结果与目标FPGA实现的结果是最相似的。组合逻辑的输出需要消耗一定的传输时间通过整个逻辑。如此，在某个时钟周期内发生的事件也许在下一个周期时钟开始前不能完成。这种类型的分析可以帮助设计人员对设计的最大时钟速度进行估计。可以通过使用越来越快的时钟进行时序模拟直到系统运行失败，也可以采用*关键路径分析*以确定最慢的路径，然后设置该设计能够支持的最快时钟。

由于时序模拟与整个设备有关，因此我们在此只对功能模拟进行讨论。功能模拟要比时序模拟快且更容易进行，并可以使用各种不同的工具。

我们在此需要的工具为一个用于Verilog源代码的编译器、一个功能模拟工具，以及一种显示结果的方法（最好的方法是波形以图形显示）。免费下载的网页版Quartus II和ISE都支持以上两种模拟（虽然支持某些特定的设备，通常不支持最新的设备）。如果能够获得ModelSim，那将是一个优秀的功能模拟工具。如果能够获得特定于设备的时序库的话，那么它也能够用于时序模拟。附录E对Verilog的编译、模拟和波形视图等开源工具进行了介绍。

8.9 编程并使用CPU

Koji Nakano教授是TinyCPU的发明人，他为该处理器开发了一个汇编器和一个C语言编译器。虽然编译器和汇编器不在计算机体系结构研究的范围内，但是它们对于实际设备的使用是必不可少的，因此我们将对它们进行简要讨论。但首先，我们先讨论为TinyCPU进行手工汇编和编写代码的过程。

⊖ 目标FPGA：其设备名称、库以及速度都已被选中来实现硬件设计。

8.9.1 编写 TinyCPU 程序

对于任何一位要为新设备写程序的编程人员，首先要做的一件事就是了解这个设备——特别是设备的约束和其内部结构。在第3章（主要是3.3.4节）我们已经看到内部结构是如何影响
409 它的指令集的，并最终决定了该设备执行某一操作的效率。

对于一个要写高效的低级代码的编程人员来说，对体系结构及其约束的了解很重要。在TinyCPU的例子中，我们已经讨论过它的体系结构（8.5.1节等）和它的指令集（8.6节）。不管体系结构的灵活性有多大，最终还是指令集限制了编程人员能够拿这个设备干什么。

除此以外，TinyCPU的编程人员还需要记住只有一个输入端口（in）和一个输出寄存器（out），一个能够进行常用逻辑操作和乘法运算的ALU。所有的程序代码和内存存储都是16位形式的，并且内存的大小为4096字。任何常量需要使用PUSHI（用于任意12位有符号整数）进行加载或存放在内存里，并可以通过它们的标号名称索引。

最重要的是，TinyCPU是一个堆栈结构机器。现有的8个堆栈位置限制了不涉及内存的计算不能超过8层深。这意味着所有的操作数和结果都是16位的（注意这也包含乘法单元，由此MUL只能将8位有符号数作为操作数）。显然，这也意味着计算也要模型化为堆栈结构（参见逆波兰表示法，3.3.5节），这点需要事先考虑。

作为一个例子，我们首先编写一个程序，将数据从输入端口读入，然后减去一个常量，再将结果放入输出寄存器。我们假设常数已加载在内存上并以“const”命名。

从输入端口读入数据并不复杂：浏览前面表8-1所示的TinyCPU指令集，可以看到IN指令用于从输入端口读入数据并存放在堆栈里：

```
IN
```

为了对其做减法，我们还需要将一个常量装入堆栈。如果是一个立即数常量，我们可以使用PUSHI，但此时常量存放在内存中，所以我们需要从内存取出该值然后将它压入堆栈，应该使用PUSH：

```
PUSH const
```

接下来，我们进行减法操作：

```
SUB
```

它将从堆栈里弹出两个输入操作数，执行减法，然后把结果压回堆栈。最后，我们可以将结果送入输出寄存器：

```
OUT
```

注意，在这个简单程序中没有用到操作数！这就是堆栈结构机器的主要特征之一——如果
410 没有寄存器，我们也不需要对其进行指定。

除了把这些代码放在一块之外，我们还需要一个存放常量的地方。整个程序代码如程序清单8.13所示，其中我们将常量的值设为3。

程序清单8.13 subtract.asm

```
1         IN
2         PUSH cnst
3         SUB
4         OUT
5         HALT
6   cnst: 3
```

接下来，我们为8.6节中所给的指令集（特别是表8-1和表8-2）中的每条指令确定机器码（十六进制）。例如，在表8-1中找到IN指令，我们可以看到它使用十六进制数D000表示。第二条指令PUSH const使用十六进制数2000 + A表示，A是常量存放的地址。在这个例子中，我们需要将标号“const”转换为一个地址——可以通过简单计算获得这个地址。该标号在程序清单8.13的第6行，但是由于计算机地址是从0开始的，所以这个常量的实际地址为5。因此，这条指令的十六进制编码为2005。

对剩余的指令重复这个过程，我们最终得到如程序清单8.14所示的机器码。

程序清单8.14 subtract.hex

```
1  D000 \\        IN
2  2005 \\        PUSH cnst
3  F001 \\        SUB
4  E000 \\        OUT
5  0000 \\        HALT
6  0003 \\ cnst: 3
```

在此，只有那些亲自做过这种转换工作的人才知道这个过程是多么枯燥乏味，即使是一个很简单的程序（而且还很容易出错）。这就是为什么Nakano教授给这个处理器构建了一个汇编器（以及一个简单的编译器）的原因，这也解释了为什么现在没有人直接写机器码。我们将在8.9.2节介绍TinyCPU的编程工具，但现在更重要的是在我们开始使用便捷方式前了解这个过程。

接下来我们将要把这个程序使用正确的格式插入ram.v中。我们可以通过查看8.7.5节中的语法格式，然后删除里边原有的程序并将我们的减法代码插入其中来实现。如程序清单8.15所示。 411

程序清单8.15 ram_subtract.v

```
1  module ram(clk, load, addr, d, q);
2   parameter DWIDTH=16,AWIDTH=12,WORDS=4096;
3
4   input clk,load;
5   input [AWIDTH-1:0] addr;
6   input [DWIDTH-1:0] d;
7   output [DWIDTH-1:0] q;
8   reg [DWIDTH-1:0] q;
9   reg [DWIDTH-1:0] mem [WORDS-1:0];
10
11  always @(posedge clk)
12    begin
13      if(load) mem[addr] <= d;
14      q <= mem[addr];
15    end
16
17  integer i;
18  initial begin
19     for(i=0;i<WORDS;i=i+1)
20        mem[i]=0;
21 mem[12'h000] = 16'hD000; //  IN
22 mem[12'h001] = 16'h2005; //  PUSH cnst
23 mem[12'h002] = 16'hF001; //  SUB
```

```
24 mem[12'h003] = 16'hE000; //  OUT
25 mem[12'h004] = 16'h0000; //  HALT
26 mem[12'h005] = 16'h0003; //  cnst: 3
27 end
28
29 endmodule
```

让我们来模拟和测试这段代码。首先，我们需要确保我们的测试平台构建正确。在这个例子中，我们将数值7作为常量提供给输入端口，然后执行减法。原来的TinyCPU测试平台被修改为如程序清单8.16所示。

程序清单8.16 tinycpu_tb_subtract.v

```
1  `timescale 1ns / 1ps
2  module tinycpu_tb;
3
4   reg clk, reset, run;
5   reg [15:0] in;
6   wire [2:0] cs;
7   wire [15:0] irout, qtop, dbus, out;
8   wire [11:0] pcout, abus;
9
10  tinycpu tinycpu0(.clk(clk), .reset(reset), .run(run), .in(in),
        .cs(cs), .pcout(pcout), .irout(irout), .qtop(qtop),
        .abus(abus), .dbus(dbus), .out(out));
11
12 initial begin
13     clk=0;
14     forever #50 clk = ~clk;
15 end
16
17
18 initial begin
19     reset=0; run=0; in=3;
20     #100 reset=1; run=1;
21     #100 run=0; in=7;
22     #12000 $finish;
23 end
24
412 25 endmodule
```

如果我们现在要模拟这段代码，需要使用如附录E中给出的方法（Icarus Verilog和GTKwave），我们将会得到执行时的波形，如图8-11所示。

图中显示输入端口得到常数值7。根据运行时给出的提示信号，指令0至指令5轮流被装载（看abus的值）。qtop显示了堆栈顶部的值：开始为0，接着是来自输入端口的7，然后是来自内存的3，最后是减法的结果4。这个结果最后被送入输出寄存器，如图中光标所标示的地方。

显然，7－3＝4是正确的。然而，读者有可能感觉到应该还有许多更简单的方法来实现这个计算。

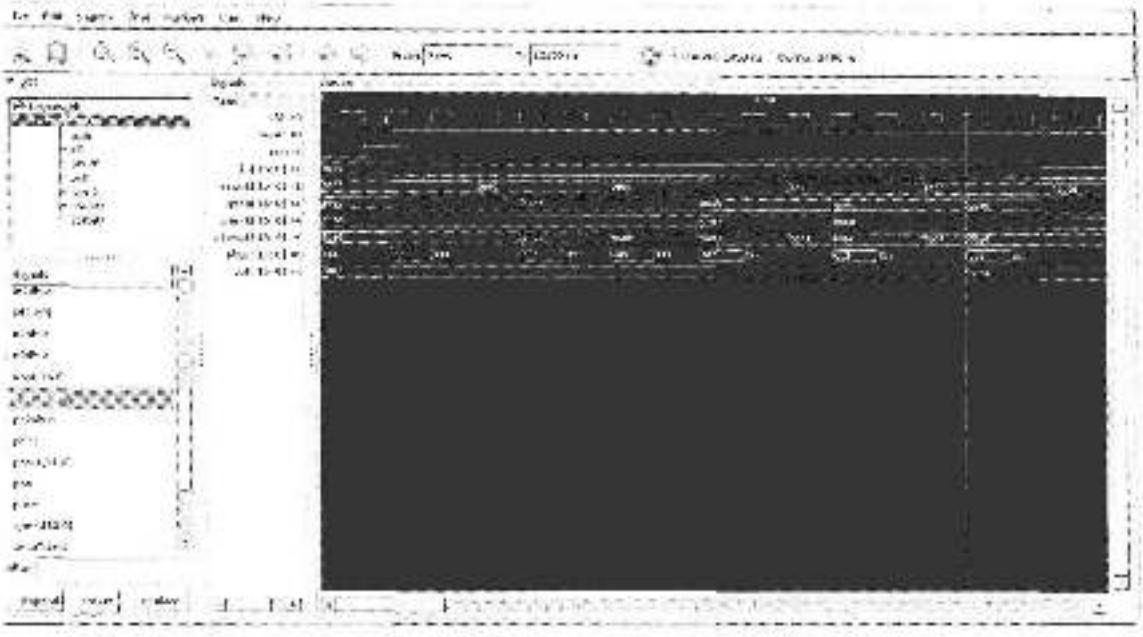


图8-11 减法例子的CTKwave波形观察仪的截屏，它显示了模拟中一些重要的信号

8.9.2 TinyCPU编程工具

在8.9.1节我们已经讨论了如何手工编写机器码，然后将它们载入TinyCPU并进行模拟，读者会觉得这个汇编过程十分机械，需要（i）确定代码中标号的实际地址；（ii）将汇编语言转换为十六进制机器码；（iii）合并所有操作数（如将常量的地址插入到PUSH指令中合适的位域）。 413

写一个汇编器并不困难。实际上，Nakano教授所做的只是写了一个汇编器和一个简单编译器。读者可以参考他的几篇文章[1]以及他的网站来获得更多信息。

附录E给出了汇编器的整个代码。它是一个Perl程序，采用二次（two-pass）汇编。第一次扫描输入的代码，确定指令的地址以及代码中标号的地址。第二次则将汇编语言转换成十六进制机器码，并将第一次扫描获得的标号地址值插入正确的地方。

这个工具给出的汇编器输出还需要另外一个辅助应用程序对之进行格式化以产生一个能够直接插入ramv里的输出。 414

8.10 小结

基于先前章节，本章首先构建了针对可编程系统（如FPGA）进行微处理器设计的基础。分别对其进行了讨论，然后介绍了一些现成的软核，并提到了其他一些不错的负责开源软核。

主要采用在本书中介绍的一些概念，我们探讨了一些使用Verilog构建一个完整的可运行CPU所需的技术。特别地，我们重现了TinyCPU的设计，它是由日本广岛大学的Koji Nakano教授开发的。TinyCPU正如它的名字一样是一个很小但完整的CPU。它这种基于堆栈的处理器可以很容易地在FPGA内部使用，并运行一些简单程序。

本章中我们首先讨论了使用软核微处理器的思想，包括使用下载的核或使用来自商家的IP

[1] Koji Nakano, Yasuaki Ito, "Processor, Assembler and Compiler Design Education using FPGA", Proc. International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, Australia, Dec. 2008.

核。另外，我们也参考了来自四个主要 FPGA 厂商的内部核。最后我们着手设计一个全定制的 CPU。我们首先给出了 CPU 设计规格说明，然后一步一步开发这个体系结构。接着，在编写整个系统前我们构建了一个指令集，而整个系统只使用了 6 个小的 Verilog 模块。伴随着这些过程，我们还为这个 CPU 的每一个部分开发了测试平台并进行测试。

最后，我们构建了以 TinyCPU 为例的程序，并模拟了它的操作。TinyCPU 是一个完美的作品：它是一个简单的自制 CPU，并足够可视和开源，使得我们可以观察其内部的工作情况。然而，它只能作为一个简单的开始：读者应该对其原始代码进行扩展、修改、重写、改进和实验。
415 让它成为引领我们进入开发新一代专用定制处理器的一课。

思考题

8.1 现成的 CPU 通常可以比在最快的 FPGA 上实现的类似体系结构 CPU 拥有更快的时钟。这是否意味着现成方案会更快？解释你的答案。

8.2 什么是软硬件协同设计？为什么这种方法在嵌入式系统开发中比在台式 PC 开发中更重要？

8.3 指出软硬件协同开发的步骤。评估哪个步骤需要对系统了解最多以及哪个步骤对于系统正确性最重要。

8.4 在你的设计中选择现成软核或商用软核的主要优势是什么？主要劣势是什么？

8.5 对于半导体厂商来说生产少量但寿命长的产品是异常困难的。怎样使用软核去帮助这些生产商？从现有的市场化 CPU 转向自定义软核方案最主要的前期成本是什么？

8.6 TinyCPU 是否总是在状态 FETCHA 之后跟着状态 FETCHB？为什么 TinyCPU 需要两种取指状态？（关于 FETCHA 和 FETCHB 参见 8.6.1.2 节。）

8.7 为什么 TinyCPU 堆栈有 qtop 输出和 qnext 输出？能否只有 qtop 输出？

8.8 根据 4.1.1 节所给的定义，TinyCPU 能否实现为单总线、双总线或三总线结构？如果能够实现，那么它的指令集将会是怎样的？

8.9 TinyCPU 的 PUSHI 和 PUSH 指令有什么区别？为什么两条指令都需要实现？归纳实现这两条指令在硬件上的区别。

扩展和升级 TinyCPU

416 接下来的问题假设 TinyCPU 使用 Verilog 实现，并对其及其指令集、工作方式提出各种升级、扩展和调整。接下来的问题都建立在自定义软核计算及这些系统的折中问题上。

对于每个改变，都要写一个新的 TinyCPU 程序（或测试程序），并且对每个新的代码都要就其正确性以及对现有代码的有效性进行测试。对于那些拥有完整 FPGA 开发系统且能够在其上实现 TinyCPU 并进行模拟的读者，还应该考察这些改变在实现 TinyCPU 的开销上有什么不同（即它的设备内存和面积资源占用率）。

8.10 扩展 TinyCPU 支持，使其支持循环右和循环左指令（ROR 和 ROL），它们在实现上与现有的 SHL 和 SHR 指令相似。

8.11 TinyCPU 能处理 16 位数据。将内部数据通路、端口、RAM 和堆栈升级至 32 位，需要新的方法（即新指令）将立即数装入堆栈地址的高 16 位上。然而，通常还有更多的方法能够实现 32 位扩展，所以请读者想出自己的办法来实现 32 位数据处理。

8.12 TinyCPU 只有一个输入端口和一个输出端口，实现新的指令将输入/输出端口增加到 4 个。

8.13 向 TinyCPU 中增加中断支持（以及一个中断向量表）。

8.14 影子寄存器（参见 5.6.3 节）可以提高中断服务例程的速度。在问题 8.13 的基础上对 TinyCPU 实现影子堆栈。这种扩展是否有必要？

8.15 在 TinyCPU 上实现与 5.6.1 节类似的基本重复指令。

8.16 思考如何将问题 8.15 的重复指令扩展为一个完全的零开销循环。

8.17 通过在 stack.v 上实现一个 SIMD 输入触发机制将堆栈进行扩展，使堆栈顶部增加到 4 项。增加一条新指令来控制这个触发机制。在另外一种方式中，让一条 SIMD 指令实现自动出栈四次，并在每次出栈后自增，然后按原来的顺序将它们压回堆栈。这两种方式哪种更快？哪种需要最多的 FPGA 资源？ 417

8.18 为 TinyCPU 实现一个协处理器。首先构建一个新的 16 位输出端口让 TinyCPU 能够向其写数据，和一个输入端口使 TinyCPU 能够从其上读入数据。这些端口与协处理器相连，每当触发时，协处理器从 TinyCPU 的输出端口读 16 位数据并将数据按位逆序（即位 0 与位 15 交换，位 1 和位 14 交换等），然后输出结果输出到 TinyCPU 的输入端口，使 TinyCPU 能够读取。

8.19 对于能够在 FPGA 上实现 TinyCPU 的读者，设计一个系统包含两个能够并行的处理器。然后在两个处理器上实现一个端口和寄存器，使得两个处理器能够互相通信（这与问题 8.18 的协处理器接口设计类似）。

8.20 向 TinyCPU 引入流水线并说明它至少在某些类型的指令上能够提高吞吐率（注意：最好能够在 FPGA 设计工具上实现，因为它们能够对系统的最长路径进行评估，即能够自动计算设计所支持的最大时钟速度）。 418

第9章

Computer Architecture: An Embedded Approach

未　来

正如题目所表达的，本章所关注的是计算机及其体系结构未来发展的方向。正如我们在之前多次提到的那样，今后的计算重点可能是嵌入式，但新出现的议题也包括环境智能（我们周围都是计算机的理念）、普适计算和云计算（类似于分布式）、量子计算机、生物计算机等。许多支持者认为对于并行计算来讲，这是一个迟到的复兴。

本章对未来计算与主流计算的不同进行了讨论，包括未来计算日益增加的重要性及潜在的广泛影响力。有些所谓的未来技术是那些曾经尝试过并且被遗忘的，但现在我们需要重新审视它们。还有，像量子计算机，听起来更像是应该在科幻小说中出现，而不是在计算机体系结构的教科书中。

无论未来怎样，目前正在研究计算机体系结构的你，完全有可能作为一分子构建并促成这些梦想的实现。

9.1　单比特体系结构

在4.2.2节，我们设计了一个ALU，由几个独立的1位ALU组成。这个方法很常见（ARM内核曾使用过），称为位片（bit-slicing）。实际上，连接到ALU的总线是并行的，因而对每一位都进行独立的并行处理。

除此之外，ALU也能以串行的方式接收这些位，再逐次处理它们并串行地输出结果。事实上，串行CPU就是所有处理都通过使用位串行体系结构来实现。

这就意味着片上时钟速度越高，需要的片上总线连接就越少。然而，CPU并不总是这么精简的，因为串行控制器要安排CPU周围的所有串行操作数，这就使得时序电路很复杂。一个较大的优点就是同一个CPU在不改变ALU的情况下能处理不同的字长（只是时序
419 不同）。

对于一些串行操作，串行位一旦被送入ALU就可以开始进行处理。对于其他的一些运算符，处理开始之前必须将所有位送入ALU中。

9.1.1　位串行加法

作为例子，考虑两个位串行数字的加法。这两个数字被送到一个最低有效位优先的加法器中，按位相加，产生的进位反馈给随后的加法操作。

在如图9-1所示的例子中，前四位已经相加，结果已经产生。这个加法器的逻辑并不复杂，实际上和图9-2所示的框图有些类似。

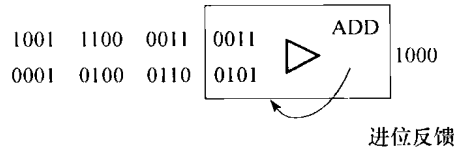


图9-1　两个串行二进制数字流按位相加的例子。每一个按位相加操作的进位来自之前位运算的进位输出

在图9-2中，A和B代表两个比特流。它们与反馈得到的前位相加产生一个和以及一个进位。最低有效位（LSB）控制信号用来禁止任何前位反馈，使得没有进位进入最低有效位的位置（按照预期）。锁存器延迟加法器的输出以使和位时钟同步，并且延迟进位以使为下一个位加法操作做准备。

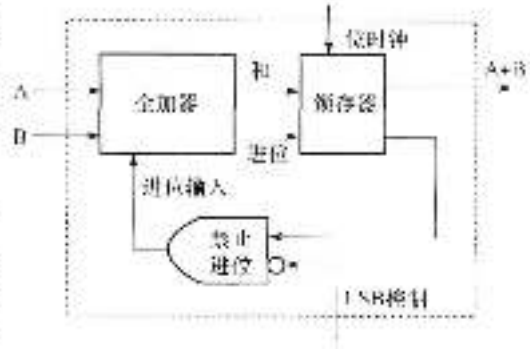


图9-2 位串行加法单元的全加器电路

使用这种机制在输入数据之间不需要任何间隔，只要最低有效位控制信号能够在正确的进位从一个加法结果传至下一个。它的好处在于只要LSB控制信号的时序划分了输入字，复用同一硬件就能够使任何字长的数字正确相加，如图9-3所示。

累加器的实现也同样简单，作为练习留给读者。

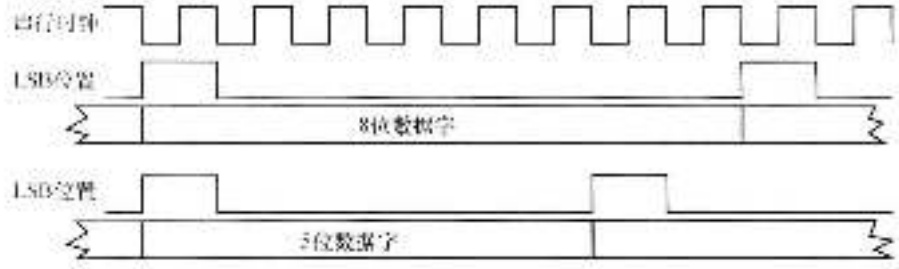


图9-3 与位时钟、LSB位置标识和数据字相关的时序波形

9.1.2 位串行减法

考虑9.1.1节的加法器，注意到进位从开始到结束被自然地传递。因为减法包含借位而不是进位，所以也是一个类似的处理过程，但是使用一些小技巧可能会使处理过程更加简单。

回顾第2章中2的补码变换相对容易（尽管对于有符号数不那么容易）：把所有的1换成0，0换成1，然后在最低有效位加一个1。我们可以利用A－B等价于A＋(－B)，因此可以简单地执行一个可负操作数的加法。

在输入信号处放一个非门可以实现位串行输入的所有位反转，在最低有效位加一个1就是确保第一个进位被置位而不是清零（即最低有效位控制信号使进位被置位而不是清空）。

位串行减法的逻辑电路如图9.4所示。与上一节中的加法电路比较，应该很容易对系统进行转换使得加法或是减法的执行取决于外部控制信号。

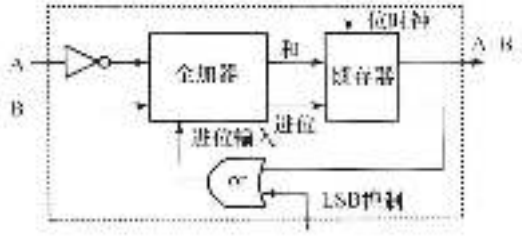


图9-4 为了完成减法B－A而经过修改的全加器

9.1.3 位串行逻辑电路和处理

考虑到由一个全加器和一个锁存器为主要构成的硬件可以使任意长度的字进行加减运算，很明显，位串行逻辑电路的硬件利用率是很高的。位串行逻辑电路能以快速时钟工作（尽管锁存器之间有非常小的逻辑进位延迟），但是缺点是不论时钟有多快，每个长度为n位的字运算都需要$n+1$个时钟周期来完成。

串行逻辑电路在FPGA逻辑电路设计方面适得其所。由于硬件的高效性，可以并行地让几个

处理流同时进行。这也符合 FPGA 的硬件体系结构，9.1.1 节和 9.1.2 节中描述的逻辑电路（或是它们的可选版本）可以在设备的单个单元中实现。这个单元可能被称为逻辑元件（LE）、逻辑单元（LC）或是来自不同制造商的类似命名，但是重点是每个单元都包含一个查找表，该查找表有执行一个全加器以及一个触发器（可以充当锁存器）的功能或是分别执行全加器和触发器的功能。单元之间的连接比单元内部的连接慢，其速度与单元之间的距离成反比。

随着数字宽度的增加，在 FPGA 中因单元互连而引起的延迟会变得严重。这就意味着大约 256 位或是更多的数字相加时，在任何时钟速率下进位传输问题都会变得非常困难。而且，目前的中档设备并不包含足够的路由互连来执行这样的功能（附带提一下，这个功能不是模棱两可的需求，它在加密算法中是一个常见功能）。在这种情况下，在单个单元中实现位串行操作逻辑变得极具吸引力。

9.2 超长指令字体系结构

超长指令字（Very-Long Instruction Word，VLIW）体系结构通过引入指令级并行（Instruction-Level Parallelism，ILP）来使 CPU 变得更快，即把高效指令排序的任务从处理器转移到编译器。这个排序包括指令分组，以便并行地执行指令。

正如我们之前提到的，VLIW 有时也被称为显式并行指令计算（Explicitly Parallel Instruction Computing，EPIC）。

9.2.1 VLIW 的基本原理

并行执行可以提高性能。这就好比并行总线在单一时钟周期内传输若干位数据，比串行总线传输速率快，在一个周期传输若干位数据，将导致需要更大的指令集带宽和额外硬件资源的权衡。处理器制造商已经尝试逐年增加时钟速度，得到了几十年不间断的处理性能提升。然而，
422 VLIW 允许时钟速度保持不变，但是每个时钟周期执行的操作数量增加了，这样也可以显著提高整体性能。我们将在 9.2.2 节讨论这个方法的一些不足，但是我们先来思考一下它的基本原理和优点。

尽管以并行方式执行运算不是一个新的想法，但是直到 20 世纪 80 年代后期 VLIW 作为一个概念才出现在 TI 的基于多媒体的 DSP 处理器中，使得许多公司都生产了高速的现代处理器，比如 TI、Philips（最著名的是 Tri-Media 系列）和 Mitsubishi（始于 V30 处理器）。尽管起步阶段相对缓慢，但是随着时间的推移，VLIW 很有可能成为主流的处理器技术，因为它打破了处理器性能与时钟速度的紧密关系。VLIW 对于多媒体而言是个好的解决方案，尤其是流式的音频和视频，而这类信息在处理器系统中的占有量在不断增加。Intel 已经采纳了 VLIW 体系结构，尽管典型的营销方式称它为 EPIC。它还广泛应用于 IA-64 体系结构的机器上。

VLIW 是 RISC 计算机理论在并行方向上的一个扩展。它有 RISC 固有的特点，单个子指令是简单而规则的，通常在一个周期内执行。不同的 RISC 类型的子指令被编译器交叠起来放入一个长指令字中，使用下列部件并行执行：

- 独立的 CPU 功能单元（例如 ADSP2181）。
- 功能单元的多个副本（若干数字信号处理器和之后的奔腾处理器）。
- 流水线功能单元。

因为 VLIW 和超标量体系结构（见 5.4 节）都涉及多个功能单元以及处理器内部的并行，这就出现了一个问题，VLIW 和超标量体系结构之间有什么区别？它们在很多方面存在差异，但最重要的是超标量体系结构取指单元的取指速度高于处理单元的处理速度，指令需要等待被处理。这是因为超标量体系结构中处理器在运行时负责安排每个指令单元做什么，指令被送入哪个并

行执行单元。相比之下，VLIW 依靠编译器来完成时序安排。编译器指示每个执行单元每一时刻做什么，从哪里获取数据，数据写回到哪里。并行指令以一个规律的速度被取出、执行，处理器指令的处理硬件没有超标量体系结构方式下复杂，因此可能运行得更快。框 9.1 给出了一个 VLIW 硬件的例子。

框 9.1 VLIW 硬件的例子

思考这个例子的代码，代码来自 "VLIW Architecture For Media Processing" K. Konstantinides, *IEEE Signal Processing Magazine*, March 1998 (© 1998IEEE)：

标准处理器		VLIW处理器	
周期	操作	周期	操作
1	add t3 = t1, t2	1	add t3 = t1,t2
2	store [addr0] = t3		add t5 = p2,p10
3	fmul f6 = f7,f14		add t1 = p2,p7
4	...等待...		fmul f6 = f7,f14
5	...等待...	2	add t4 = t1,t5
6	fmul f7 = f7,f15		fmul f7 = f7,f15
7	...等待...		store [addr0] = t3
8	...等待...	3	store [addr1] = t4
9	add t1 = p2,p7		
10	add t5 = p2,p10		
11	store [addr1] = t4		

这个例子使用伪码表示了普通处理器（左半部分）如何执行一些加法、浮点数乘法以及存储操作。执行这些操作需要 11 个指令周期，包括等待乘法操作完成所需的时间。右半部分的 VLIW 处理器带有多个可并行执行的功能部件，将几条指令装入单个字中，执行相同操作只需 3 个指令周期——没有任何理由认为这些指令周期比非 VLIW 处理器慢。

再考虑一分钟，对于右半部分的 VLIW 处理器需要哪些并行硬件支持，这个问题在 9.2.1 节中作为 VLIW 基本理论的一部分被探讨过。

框 9.1 所提供的例子中，VLIW 编译器输出如下所示：

	ALU1	ALU2	ALU3	FPU1	取数 - 存数
Instruction 1	ADD	ADD	ADD	FMUL	NOP
Instruction 2	ADD	NOP	NOP	FMUL	STORE
Instruction 3	NOP	NOP	NOP	NOP	STORE

423

以这个例子为例，一个超标量机器将要取出 8 条顺序指令（忽略若干 NOP）。它们能够并行到何种程度取决于硬件的灵活性和系统的当前状态。

VLIW 指令通常是 1024 位长，直接控制多个硬件单元，例如 16 个 ALU、4 个 FPU 和 4 个分支单元。

9.2.2 VLIW 的难题

在之前内容的讨论中已经显示出 VLIW 的硬件比等价的超标量机器要更规范和简化，但是增加了编译器的复杂性。

为了使 VLIW 机器能够有效编译，编译器不得不考虑分离数据流——在用户程序中重新安排指令顺序来提高指令的吞吐量，注意先前的指令结果会影响到之后的指令。换句话说，编译器要 424
尤其注意避免流水线冒险（如在 5.2 节中提到的情况）。

下面是其他一些与 VLIW 代码有关的潜在问题：

- 糟糕的代码密度——有时它使得程序没办法实现完全并行，在这种情况下 VLIW 代码使

用很多 NOP“填充”指令字。

- 需要复杂的编译器——这完全是把困难从硬件领域转移到了软件领域。
- 需要高带宽内存——平均来说，一个 VLIW 处理器比其他处理器（例如超标量）需要更多的指令带宽，填充额外的 NOP 加剧了这个现象。指令存储器通常有 64 位的、128 位的或是 256 位的。这意味着需要更多的内存芯片，需要更多的印制电路板（PCB）空间来布置总线，需要更多的引脚给处理器 IC。
- VLIW 使用汇编语言编码很难——使用高级语言（HLL）是使用 VLIW 处理器的一个重要先决条件。

为什么 VLIW 不能广泛用于个人计算机体系结构系统（其中向后兼容性是必需的），编译器的复杂度问题是一个原因。如果采用 VLIW，编译器将需要更加智能的版本——目标代码会发生改变，现有的低层次工具需要被替换。相比之下，超标量技术与遗留代码是完全兼容的。它们需要更加复杂的指令处理硬件，但是编译器依然是简单的。

像 Mitsubishi 和 Philips 这些公司已经设计出没有遗留代码问题的全新结构，在采用 VLIW 方面已经取得了一些初步的成功。

9.3 并行与大规模并行计算机

随着 CPU 或核心变得越来越小，以及以太网等方式使互联越来越方便，将计算机集聚成群以便相互合作变得越来越容易。抛开为这类系统开发高效的软件不提，单单从硬件的角度来看，只需简单地将现成的 PC 放在一起就可以组成一个计算机集群。

在 5.8 节中我们曾指出，在计算机系统中存在各种层次的并行，并且不同种类的并行也符合我们在该章提出的松耦合与紧耦合系统的区别。在本章我们将重点关注最高层次的并行即机器级并行。

对于由多个松散耦合任务组成的计算问题（例如一组用于执行困难而复杂功能的代码，并且它们之间只能用较低的带宽进行通信），并行执行每个任务可以加快整体的完成时间。

425

还有另一种情况，当系统中的任务彼此之间需要频繁通信或者使用较大的带宽时，由于 CPU 之间的通信存在瓶颈，因此并行执行并不能加快整体的执行速度。然而，当系统中存在足够多可以并行执行的任务时，并行处理可以将整个系统的处理进程在一段时间内向前推进。

在大型并行处理系统中，任务在传统意义上都是在独立的 CPU 上执行。这里所指的独立 CPU 可以是一组单独的 CPU 或者 PC、机架服务器或刀片服务器。我们还可以将该观点扩展到集群与集群之间，但是这已经超过了计算机体系结构的领域，最好放在有关并行与分布式计算的教科书中讨论。

本章中，我们首先会从普适计算机的观点讨论这一问题，然后将该观点特化到嵌入式系统中。

9.3.1 大型计算机集群

我们接触最多的关于计算资源共享的概念是*普适计算*（我们大多数人身边的计算机设备都能够以这种方法协同工作）。在协同提供服务时也被叫做*环境智能*（ambient intelligence），最近也被称作*随处*（everywhere）计算。

云计算（cloud computing）的概念更加贴近实际，它是指多个分布式计算机彼此通过某种方式互联（通常是通过因特网）来协同完成一项处理任务。云计算的云端一直在发生着变化，其中的计算机会不断地加入和离开云，但整体上的计算服务不会停止（至少应该如此）。有时需要使用*虚拟化*（virtualisation）技术来使云端彼此连接的不同体系结构的计算机彼此看起来有相同

的结构。因此实际上是这些虚拟化的主机形成了集群。这种云的比喻来自以图形描述因特网时往往采用云的形状。这种类型的设计被更形式化地称为网格计算（grid computing）。在这种形式下，集群中相互连接的机器是分布式的，甚至可以像云计算一样，是人们办公桌上的PC机。网格的比喻来源于能源网的思想，在能源网中生产者与消费者彼此互联来达到让资源共享和平衡负载的作用。

许多拥有巨大云端或网格的公司，会销售这些系统上的计算时间，通常以CPU时间或类似的概念来定价。采用这种方案是因为需要保证这些服务可以获取利润，以维持集群计算服务的建立和运行。一些提供此类服务的服务器区（server farm）占地量巨大，其大小甚至可能超过一个足球场；而另外一些则占地较小且更具美感，比如在1.4节中提到的位于巴塞罗那的超级计算机MareNostrum。

9.3.2 小型计算机集群

将并行计算与嵌入式系统结合在一起的思想甚至是在十年前也是不可想象的。然而，现在这两个领域的交叠越来越多。 426

本节将主要从两个方面讨论有关嵌入式系统中大规模并行化的问题。其一是嵌入式（通常指便携式）系统将它们的计算任务放到固定的集中式计算机上进行；另一点与其正好相反，是集中式计算机由于并行处理部件的嵌入化而变得更加便携。

9.3.2.1 从嵌入式系统中解放计算资源

首先，普适计算指的是计算无处不在并且彼此相连。假设我们周围充满了始终工作的计算资源，我们可以在需要的时候使用它们。现在的观点认为，比起在嵌入式系统中设计高功耗的高性能处理模块，不如为其添加无线传输模块，即让计算任务在功能强大的固定的计算机上执行，而将结果无线传递给嵌入式设备。拿一个便携式的媒体播放器来说，它包含了显示和音频硬件，而真正播放的资源则在其他地方存储和处理，之后只是将结果简单地传递给播放器。

这也是远程处理一个非常吸引人的特点。它将极大程度地解决便携式系统的能源局限性（电池技术的发展速度远远落后于处理器性能增长的速度）。然而，目前能够可靠、低能耗传输的无线技术还很匮乏。另外，值得我们注意的是，单单就那些有潜力提供高质量无线链路的技术而言，它们本身就非常复杂而且计算量大（在很多情况下，甚至超过了便携式设备本身需要的计算量）。

毫无疑问，现在有很多令人期待的关于解放嵌入式系统计算资源的例子。然而，由于受到成本、可靠性以及可用性的制约，该方法目前只能应用于小型项目，移动典型行业中的现有连接模式除外。

嵌入式系统大规模并行化的另一个方向，是在嵌入式系统中设置并行处理单元，这已经在实际中有所应用。我们在5.8.1节中讨论了双核ARM946，并且我们可以在嵌入式FPGA设计中找到很多关于多核的设计。然而，真正嵌入式集群的例子还是非常有限的，不过在不久的将来肯定会越来越多。接下来我们讨论最早的一种嵌入式集群，并行处理单元（PPU）。

9.3.2.2 并行处理单元

并行处理单元（Parallel Processing Unit，PPU）用来为微型卫星（指重量在100kg以下的卫星）提供高度可靠的计算服务。这种卫星用来在距地球500km的轨道上获取画面，对之进行处理并传送回地球。

从7.10节的介绍中我们知道宇宙中存在着干扰电信号的宇宙辐射，因此大多数卫星设计者们会选择耐辐射或者能容忍辐射的CPU。不幸的是，由于制造、测试、加工工艺的限制，使得这类CPU的制作成本高、制造困难、耗能大且速度慢。大部分微型卫星搭载8086系列的处理器，这使得它们的处理速度很少能超过10MIPS。但即便如此，卫星设计者们仍然十分保守（虽然有 427

很好的理由，但他们也该冒险尝试花费上百万美元来研究一种板上计算机），而且他们基本上将处理器降频到出厂设计的一半使用。

由于如此有限的处理能力，毫无疑问卫星不在自身内部进行数据处理：大多数只是简单地捕获信息并传输给地面计算机处理。这种做法明显与将更多的计算任务放在移动端处理的趋势相违背。本文并不是为了讨论这些做法的好坏，只是想说明逐步采用商用现成（commercial off-the-shelf，CTOS）CPU 改进卫星板上计算机能力的趋势。

PPU 就是使用这种方法。它基本上是基于 Intel 公司的 StrongARM（SA1110 系列 CPU，遗憾的是该项目已被 Intel 终止）设计，该芯片拥有 200MHz 的频率和两块可容忍辐射的现场可编程门阵列（FPGA）。由于 COTS 处理器在宇宙空间的生命周期很短，因此在 PPU 中实际上有 20 个独立的 CPU，并且系统被设计成可以适应随着使用时间增加而逐渐出现的 CPU 故障问题。基于现在已公开的辐射容忍信息，PPU 在其 3 年的生命周期中被设计成有足够多的 CPU 资源来支持它完成相应的任务。

PPU 的结构如图 9-5 所示。从中我们可以发现两块 Actel 公司的 AX1000 FPGA 分别连接 10 个处理节点（PN），每个 PN 通过一条并行总线与 FPGA 相连，该总线的作用会在稍后介绍。在 FPGA 中一条分时全局底板（time-slotted global backplane，TGB）总线会像令牌系统一样向各个节点发送消息和数据。每个 PN 都有专有的 TGB 节点，外部链路、内部可配置处理模块（PM）、状态寄存器（SR）也是如此。外部链路主要是用于连接固态记录器、大容量闪存阵列以及由 C515C 控制器控制的 CAN 总线。其中 CAN 总线用于与 PPU 的信息交互。两块 FPGA 芯片通过低电压差分信号（LVDS，参见 6.3.2 节）在 TGB 的连通处相互连接，此外，LVDS 还在固态记录器的连接中使用（并且顺带连接摄像头模块以及数据高速下载射频模块），该固态记录器用于快速数据交互。

PN 的控制代码存储在闪存中，在每一个 FPGA 中都有三份相同的副本以三重冗余的方式存储（详见 7.10 节）。

整个设计贯彻了“以冗余为代价获取可靠性”的思想，并在一开始就将可靠性考虑在内。基于可靠性的设计主要有：

- PN 的多个副本——由于有大量的 PN，因此系统能在其中一些 PN 出现故障的情况下继续正常工作。
- 独立总线——如果所有 PN 共享一条公共总线，那么它们的地址或数据总线管脚很可能由于宇宙射线的干扰而产生错误，出现电压过高或过低的状况。这样可能会导致连接在一条共享总线上的所有设备都无法正常通信。因此，每个 PN 与 FPGA 之间都有独立的并行总线。一个 PN“死掉”并不会影响其他 PN 的正常工作。
- 分布式存储——类似地，共享存储的故障会影响所有连接的处理器，因此该系统除了共享的固态记录器以外不使用任何共享存储。
- 三重冗余控制代码——每个 FPGA 包含 3 块闪存，这使得 FPGA 可以在控制代码的每个字上进行按位多数票决。
- FPGA 之间的两条链路——如果一条 LVDS 链路出现故障，另一条仍然可以正常工作。
- 两条固态记录器链路——类似地，如果一条 LVDS 链路出现故障，另一条仍然可以正常工作。
- 两条 CAN 总线——再一次，这种机制在一条总线发生故障时为其提供后备。
- TGB 总线节点——非常简单的错误容忍单元，用来检测所连接的设备是否依然能够工作，除此之外，它们不能阻止 TGB 上的通信。
- TGB 数据包——包的源地址、目的地址和数据域都有奇偶校验保护。

- TGB 总线回路——TGB 回路覆盖了 32 个节点，每个 FPGA 各占一半。在一共独立节点发生故障的情况下，保证其他节点不受影响。当 FPGA 之间的链路发生故障时，每一边的 TGB 总线会检测到该故障、"修复"这个破坏并且能够使各自剩余的部分不受影响地继续工作。
- 双 FPGA——保证在其中之一出现故障的情况下，PPU 仍然能够工作。由于能容忍辐射的 FPGA 的可靠性远远高于 SA1110，因此我们不需要像 PN 那样为其准备 20 个备份，取而代之，只需两个 FPGA 即可。

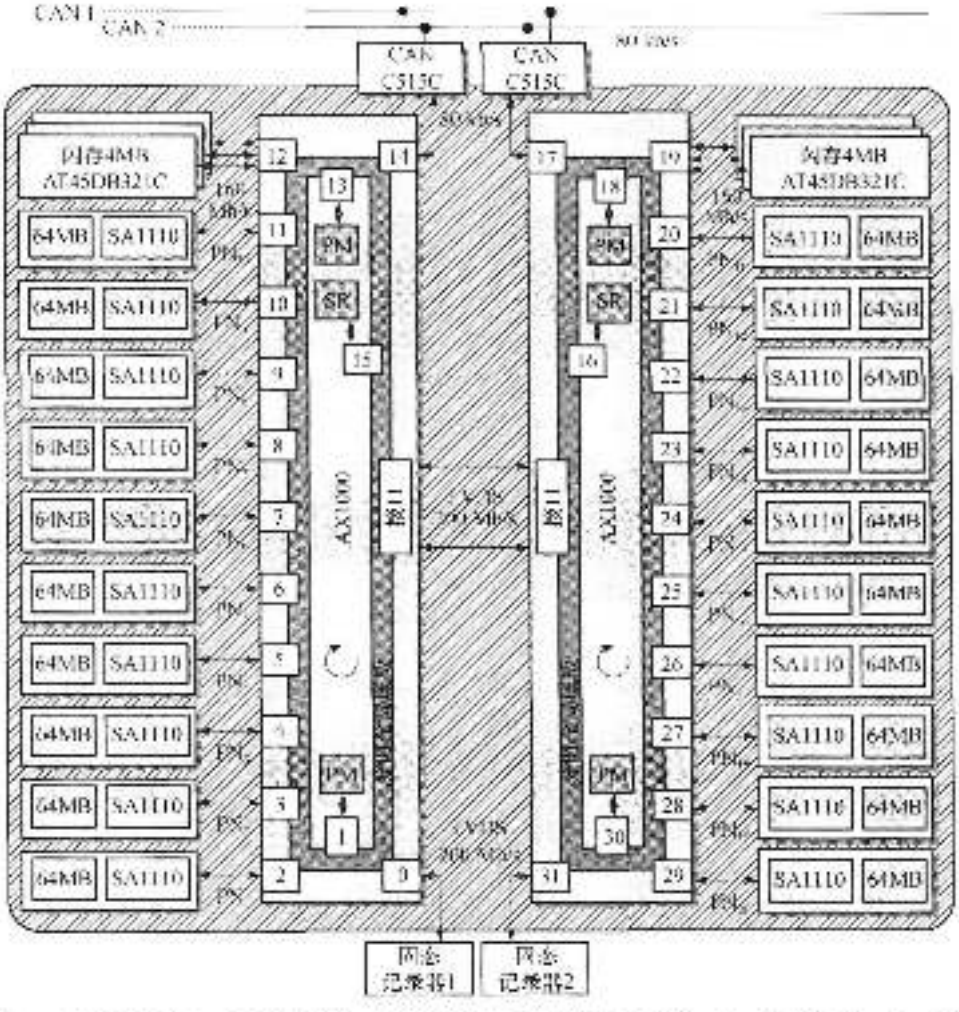


图9-5 并行处理单元（PPU）概图，包括了 20 个处理节点（PN），每一个都包含一个 SA1110 CPU 和 64MB 的本地存储单元，并且通过存在总线与 Actel 的 AX1000 FPGA 相连。两块 FPGA 中的每一块控制 10 个 PN 并且与一个固态记录器和一条 CAN 总线相连。两块 FPGA 通过两条双向低电压差分信号（LVDS）链路相互连接。一条分时全局底板总线用于在 PN、外部链路、内部可配置处理模块（PM）和状态寄存器（SR）之间传送数据

虽然 PPU 具有故障容忍性，但它还只是传统类型的并行处理器。每一个 PN 能够独立工作并且能够与其周围的节点通信（通过 TGB）。计算机系统中存在一种机制可以将物理节点数（0，1，2 直到 31）重定位为不同类型的逻辑连接，其中的各种情况我们会在 9.3.4 节中讨论。

图 9-6 就是一个重定位的例子。任何一个需要向 PN 提交一个计算任务的节点，都可以将该任务提交给该 PN 节点本身，或其他节点，使得系统本身具有很高的灵活性

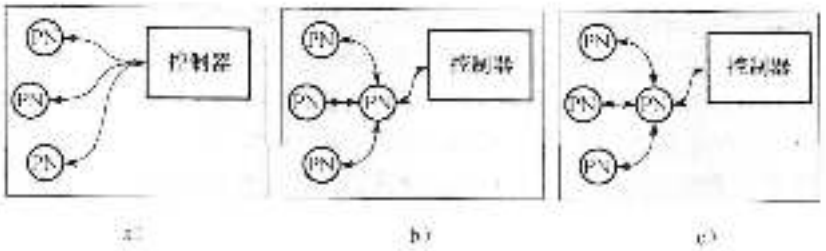


图9-6 对于PPU中PN节点的重定位，以及PN节点之间路径的建立可以使用多种不同的策略。在本图中，a）表示3个PN独立地工作，可以看做一个由外部控制器控制的3路多数票决器；在b）中，另外一个PN将多数票决处理从控制器中分离出来，通过这个PN来轮流访问其他3个PN使它们相互合作；在c）中，4个PN相同，并且其中一个负责与控制器的通信

在PPU生命周期的开始，所有部件都能够正常工作，此时它完全符合嵌入式计算机的规范（几乎是10年前设计的规范）：4000MIPS、6W的耗电量、1800cm³封装大小（与小型笔记本电脑的大小相当）。一个传统的微型卫星上使用的板上计算机的速度要比其慢200倍、体积大2到3倍并且功率相当。除此之外，它的耗电量是PPU的10倍——虽然耗电量不是卫星设计的首要因素。

428
-
430

虽然在PPU的设计中还有很多令人感兴趣的因素，比如为了实现数据的高速传输而设计超常规的17位并行数据总线，但我们在本节中只关注并行处理问题。如图9-7所示为在多个处理器之间共享一个图像处理任务时的加速比。加速比（定义见5.8.2节）指示了一个系统并行处理的并行度。一个完美的加速比（如图9-7中的斜线所示）意味着一个任务可以在一个n处理器计算机上加速n倍。示例中在PPU上运行的并行算法并没有达到完美的加速比，不过已经可以看出随着并行度的增加而带来的优越性。

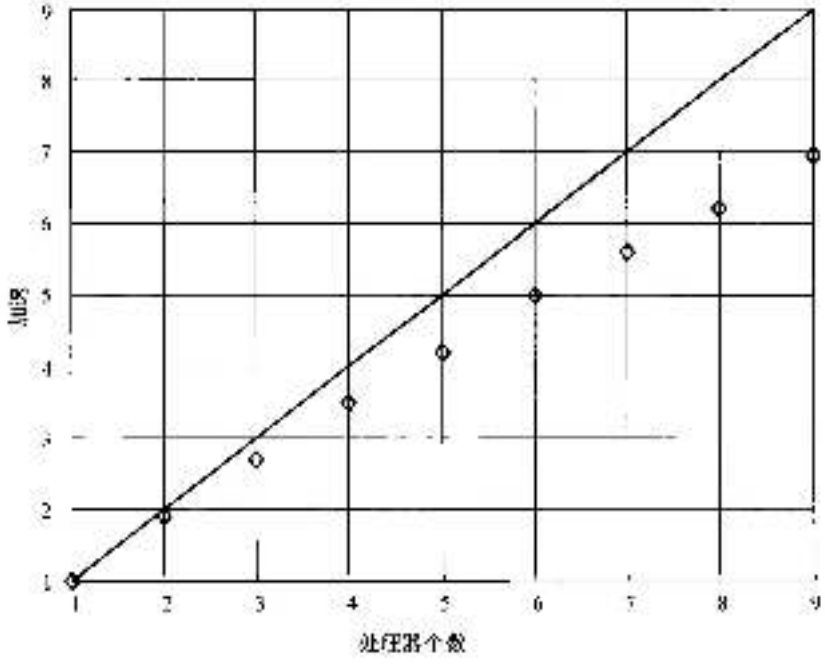


图9-7 通过使用最多9个处理节点完成共享图像处理工作，并行处理单元所能达到的加速比程度是一条完美的对角线。显而易见的是，PPU并不能达到如此完美的加速比曲线，但明显会从并行处理获得好处。能够得到这样的结果是因为采用了PPU技术，该技术是由Timo Bretschneider博士和他的学生们为了处理大被监视的图像分类而共同发明的

9.3.3 并行和集群处理注意事项

设计并行处理系统需要思考以下几个问题：

- 需要多少处理器？
- 处理器之间是怎么连接起来的？
- 每个处理器的能力有多大？
- 系统是同构的还是异构的，即系统中所有的 CPU 是相同的还是混合的？

如果有 n 个处理器在运行，那么我们可以计算得出，完成任务的时间不等于单个处理器所消耗时间的 n 倍，甚至在一个同构的系统中也是如此。实际的开销时间或多或少要小于单个处理器开销时间的 n 倍。这个区别主要取决于原始的单线程操作和并行操作的不同。我们一般会很容易地把一些计算问题分成几个子任务，这样在这些子任务之间传输的数据量是很小的。基于这个假设，可以为每个子任务分配不同的处理单元进行处理。在其他的系统中，这样的处理过程并不是很简单的。 431

因为子任务的复杂程度并不都相同，所以采用异构的系统——包含各种不同性能的处理器——会带来更多的优势。处理器之间的内部连接方式需要满足在处理计算问题过程中的一些必要条件，也就是说，采用异构连接是有可能的。但是，控制这样一个系统会更复杂——尤其是完成动态任务分割和动态选择不同的已给定类型的处理器。

9.3.4 互连策略

让我们设计一个更一般的系统，这个系统具有多个相同的（同构）处理器，我们把这些处理器称为节点。如果这些节点采用一种规则连接起来的话，该规则主要应该考虑节点间的互连方式和连接的数量这两个方面。

互连方式定义了在链路中传输的数据宽度和消息传输的延迟时间。典型的互连方式是以太网、ATM（异步传输模式）、光学互连和无限宽带。这些类型取决于它们的带宽和成本。 432

此外，有两个分布式并行处理系统的示例，在它们之间有很多不同之处——主要是共享内存和消息传递。消息传递采用结构化方法在节点之间进行通信，例如消息传递接口（MPI），并且这样的技术能很好地适应以低带宽数据连接的松耦合任务。当不同的处理器在同样的数据源上进行操作或是对通信的带宽要求比较高时，共享内存是很有用的。具有共享内存的系统包括 4.4.7 节中曾讨论过的 MESI cache 一致性协议。

每个节点所拥有的互连数量限制了它所能连接的其他节点的数量。其中一个极端是每个节点完全和其他节点连接起来。在处理器之间的连接数据传输速度相对比较慢的前提下，因为每一次传输都是在单跳的情况下进行的，所以，一个全连接系统会使得数据传输的持续时间缩短。另外一个极端是环形结构，即每个节点和另外两个节点相连接。这些互连策略如图 9-8 所示。

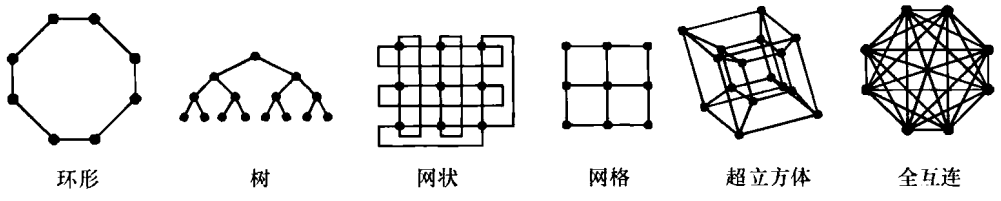


图 9-8 6 种不同的并行互连方式，图中以圆点表示计算节点，以直线表示节点之间的链路

环形——每个单元都有两个连接，它可以在不改变每个单元连接结构的情况下进行扩展。但是，在目标节点之间的连接会存在潜在的数据多跳。

树——每一个单元（顶层和底层除外）都有三个连接。它很容易进行扩展，并简化软件数据路径，但是节点之间的连接会存在数据多跳。

网状——每个单元都有四个连接。它很容易扩展，但是数据路径很复杂并且存在多跳。网格（grid）也是一种相似的连接方式，但是在边缘环绕连接方式上有所不同。

433 **全互连**——每个单元和其他的每个单元都有连接，在这种连接方式下不易于扩展。但是任意节点之间的连接最大只有一跳。

超立方体——尽管每个单元以四面体的方式都有四个连接，但是在数据路径上的跳数可以降低到最小。在大多数情况下，这是一个体系结构的选择，有时仅仅是因为它在公司新闻稿看上去显得高科技才会被选择。

的确，存在混合策略是理所当然的。例如，一个网状-环形结构，环上的每个节点是一组机器，并连接成网状结构。也许一个更普遍的例子是超立方体的网格结构，这样的结构常出现在一个以网格结构连接的并行处理中心中，在网格中的每个顶点都包含一个以超立方体为体系结构的处理器。

粒度大小用来描述并行的级别。在大多数细粒度的机器中，实际的机器指令是以一种并行的方式进行处理（例如向量机或 VLIW 处理器），相应地，粗粒度机器能够并行地运行大型软件。这部分内容在 5.8 节中有详细介绍。

随着诸如 MPI 等抽象概念的出现，粗粒度并行算法能够在不同的程序实例中执行。这些算法能否运行在一个 CPU 上或同时运行在多个 CPU 上并不重要。类似地，这些 CPU 是集成在一个板子上，或是在配有一个数据中心的几块板子上，或是在以云结构或网格结构组成的计算机系统中的不同的地理位置上也是不重要的。

粗粒度机器都是松耦合的，相反细粒度机器更倾向于紧耦合。单元之间数据传输的大小决定了它们之间的数据连接的速度和数据传输的跳数，并且这方面要考虑到带宽和延迟时间等因素（例如，如果处理器之间的数据必须传送两跳的距离，那么每一跳将是带宽的两倍）。数据传输的要求对存储器结构也有一定的影响，例如每个处理单元应该使用局部存储还是之前提到的共享存储。局部存储器可以进行分布式存储或简单地使用缓存的多个副本。有关大规模并行处理机的例子见框 9.2。

框 9.2 并行处理机的例子

Roadrunner，在编写本书时最快的超级计算机，位于美国新墨西哥州的 Los Alamos 实验室。实际上它是一系列 IBM 计算机的集群，包括 6912 个 1.8GHz 的 AMD Opteron 处理器和 12 960 个 3.2GHz 的 IBM PowerXCell 8i Cell 处理器（5.8.3 节）。在 Opteron 和 Cell 内总共有超过 103TB 的 RAM 空间，所有机器之间通过无限宽带（Infiniband）技术相连接。核的总数（记住，一个 Cell 包含 9 个核，一个 Opteron 包含 2 个核）超过130 000 个。驱动这台机器的操作系统理所当然是 Linux。整体来说，这个系统峰值速度可以达到 1.71 petaFLOPS（10^{15}FLOPS——见 3.5.2 节）。但是同时要消耗 2.35MW 的电能，大约等于一个英国小镇的耗能。

Eka，印度最快的超级计算机（世界排名第 13 位）。供职于杰出的 Tata 集团，可以以 0.786MW 的电能达到 172 TeraFLOPS（10^{12}FLOPS）的运算速度。14 240 个核散布在 1800 个 3.0GHz 的 Intel Xeon 53xx 处理器节点上。据说这台 Linux 驱动的机器是基于一个商业计划的尝试，可以使他们的工作运行于这台功能强劲的野兽身上。

DeepComp 7000 是中国最杰出的超级计算机，位于中国科学院的计算机网络信息中心。该机器使用了通过无限宽带技术相连的 3.0GHz 的 Intel Xeon E54xx 处理器，总共包含了 12 216 个核，运行速度排名世界第 19 位，峰值速度达到 145 TeraFLOPS。近些年来，中国的超级计算机在排行榜上进步明显，已经逐步越过了英国和美国等传统的超级计算机强国。

超然洒脱的是 Google 的服务器群。如果把它们看成一个集群，那它们极有可能超越所有的超级计算机。然而 Google 和他的竞争者们都是暗地里安装他们的机器。很少公开声明，Google 几乎从不显露它自己的东西。因此，Tata Eka 仍旧是极少数的对外公开的私人商用机器之一。

9.4 异步处理器

现阶段，所有常用的现代 CPU 在执行操作时都采用同步方式，这意味着它们使用一个或多个全局时钟（和域），如处理器时钟、存储时钟、系统时钟、指令时钟以及总线时钟等。

在某一特定时钟域内（芯片上的一个物理区域，所包含的部件受同一时钟驱动），触发器和基于基本触发器的单元在执行操作时是同步的。时钟的速度需要考虑一个由最慢部件所决定的 434
特定域的上限。一般来说，这意味着有些部件本可以运行得更快，但实际运行速度却要被最慢的部件所抑制。

举例来说，一个 ALU 从两个保持寄存器中获取它的输入值，一个时钟周期后，将运算结果锁存到一个输出寄存器中。如果执行一个加法运算，那么操作基本上会刚好完成，也许只是早 0.01 个时钟周期。然而，如果执行的操作比较简单，如与运算这样不需要进位传递的运算，那么操作通常会提前很多，也许会早 0.9 个时钟周期完成。所以，基于 ALU 所执行的不同操作，运算时间也许会占据一个时钟周期或者会提前完成等待结果。无论如何，控制该过程的时钟会被设定为与最慢的操作相匹配。 435

对于 ALU 操作的分析表明：在大量的时间里，这些单元通常处于空闲状态，这说明整体的使用效率很低。有很多技术可以用来克服效率上的制约，包括允许并行操作（几个事件同时发生而不是依次发生）和使用流水线技术。流水线技术将单个元素划分为更小、更快的元素并且在操作过程中重叠地使用它们。得益于每一个元素的加速，整体时钟的速度可以得到提高。

一种非常不常用的技术是使用异步处理器。异步处理器允许每一步操作在不浪费任何时钟周期的情况下全速执行。实际上，也许根本没有必要设立一个时钟，因为每个元素都是在以最大速度执行操作，当操作完成后将通知硬件控制部分。

使用同步方法的优点如下：

- 设计相对简单而且相对来说有更多的关于同步方法的设计经验。
- 几乎所有的 CPU 设计工具都假定了一种时钟设计。
- 消除竞争条件。
- 可预见的延迟。

使用同步方法的缺点如下：

- 在高速条件下，时钟偏移将成为一个问题。
- 几乎所有的锁存器和门电路会随着时钟进行切换（不管是否有数据要处理）。由于 CMOS 的能耗基本上都是由切换引起的，因此会导致更多的电能消耗。
- 实际性能低于理论最大值（最慢元素的操作速度会导致部分时钟周期的浪费）。
- 大量的硅资源将被用于时钟生成和分布。

使用异步方法是有理可循的，尽管每一个异步元素仍然与它的邻居相连，而且这也需要某种形式的同步，但是不需要一个全局时钟。这种同步方式可能相对简单，但是复制到一个由许多元素组成的集成电路中会导致额外的逻辑。

理论上来说，与相似的同步处理器相比，异步处理器会有更低的功耗和更高的运行速度。然而，设计者需要高度注意竞争条件发生的可能性。为了避免这些情况，异步处理器比同步处理器在规模上会稍大一些。

AMULET 是异步处理器的一个很好的例子（实际上是在编写本书时世界上唯一的商业异步体系结构）。它由英国的曼彻斯特大学设计，基于非常流行的 ARM 处理器。在 AMULET 的设计过程中解决了很多特定的问题。我们将在下面几节中分析一下其中某些问题以及设计者采用的解决方法。 436

9.4.1 数据流控制

如果处理器没有相应的基准时钟，那么如何对数据流从一个单元到下一单元的流动过程进行控制呢？AMULET 采用了一种请求-确认握手机制，在单向并行总线中按照下列事件顺序执行：

1）发送端将数据发送到总线上去。

2）发送端触发请求事件。

3）一旦准备就绪，接收端从总线上读取数据。

4）接收端之后触发确认事件。

5）发送端将数据从总线上移除。

请求和确认信号是在沿着标准总线运行的两条单独线路中传送。图 9-9 描述了实际应用中边缘触发信号的情况。

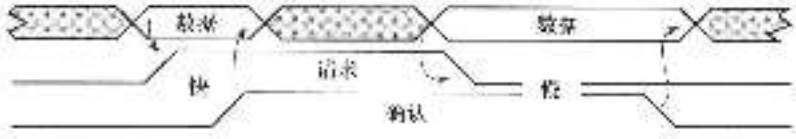


图 9-9 用于异步总线运行的请求-确认总线协议

使用请求-确认信号协议每一个组成元素可以使用自己的时钟，AMULET 的每一个流水线元素根据实际执行指令的情况按照自己的速度各自运转。换句话说，执行简单操作的单元会很快完成操作，执行较为复杂的单元可能会需要更长的时间。在最糟糕的情况下，一旦采用同步时钟（如上文所言，最慢的元素决定整体运行速度），整个流水线将以相同的速度执行。尽管如此，在实际应用过程中，流水线的效率还是要高于整体采用同步时钟的情况。

9.4.2 避免流水线冒险

如果指令在不同时间甚至是未知的时间里完成，那么我们如何能够在流水线中避免先写后读风险呢？

由于处理器采用类似于 ARM 芯片的载入-存储体系结构，因此几乎所有的 CPU 操作数都是寄存器到寄存器的。因此，我们需要一种机制来避免先写后读问题的发生，即一条读取指令要从某寄存器中读取数据，然而该数据结果仍未被前一条写指令写入。

解决的方法是采用一种基于 FIFO 的锁寄存器机制。当一条指令将要写入一个特定的寄存器时，它按照 FIFO 方式设定一个锁，然后当写完成后清除该锁。当读指令发生时，首先检测被读寄存器是否存在锁。如未存在锁，则暂停该指令直至寄存器解锁。

437

图 9-10 给出了寄存器锁的例子。图中给出了前 8 个 FIFO 的寄存器锁。锁的位置与流水线位置以及流向相一致。在正在运行的程序中，指令 1 的结果被送到 r1，指令 2 的结果被送到 r3，指令 3 的结果被送到 r6，结果到达后清除相应的寄存器锁。在当前时间内，如果有任何指令读取 r1，r3 或者 r6，则相应地读取指令将被暂停直至相应的寄存器锁解锁。

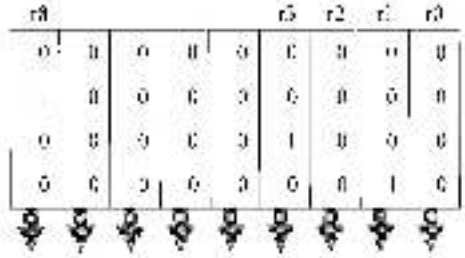


图 9-10 关于避免流水线行为的 AMULET 处理器的寄存器锁硬件。目标寄存器通过 FIFO 依次排列（与流水线有关），可以指示寄存器接下来将要进行的写操作的顺序

尽管寄存器锁机制解决了可能存在的先写后读冒险，然而它会导致流水线频繁停摆。因此，最近的 AMULET 处理器在异步应用中采用寄存器转发机制。

9.5 替代数字格式系统

纵观计算机世界和 CPU 发展的今天，我们看到大量的二元逻辑设备，高电压用逻辑 1 表示，低电压用逻辑 0 表示。二进制字由几个位组成——通常为 4，8，16，24，32 或 64。无论是二进制补码还是无符号数，每个位的加权在二进制字中都代表 2 的幂。进一步扩展的浮点数分别代表尾数和指数，但这些也都是二进制字，其加权基于 2 的幂。

模拟计算机是一个令人耳目一新的选择，但在 20 世纪 80 年代，数字计算机变得更快、更便宜、更小、更准确，模拟计算机逐步被淘汰。为了满足计算需求，数字二进制系统处于主导地位。

然而，处于研究中的某些替代系统在某些应用中可能迟早成为主流。这些都将在下面几节中探讨。

9.5.1 多值逻辑

越来越快的时钟速度是未来几年计算业界发展的规律，但是受到了很多限制。诚然，这些限制正在慢慢变小，但它们仍然制约着前进的脚步。正如我们在第 6 章中提到的，如果时钟速率受到限制，可以通过多线并行或在一个时钟的上下双沿都传送数据（双数据速率或 DDR）来传输更多的数据。 438

还有一种替代方式，即允许电压改变以使得每根数据线可以携带更多的信息。这可以通过使用多值逻辑来实现。例如，编码一个两位的字到 4 路电压中：

```
00    0.0V
01    1.7V
10    3.3V
11    5.0V
```

这和电子学中常用的 CMOS 电压水平是截然不同的，需要更复杂的驱动器和检测电路，但两倍的数据量可以在一根线上表示。相对于利用两个（二进制）电压水平的系统，它降低了数据的抗干扰度。CPU 使用这样的逻辑需要考虑模拟以及数字化设计问题。随着抗噪声能力的减弱，它有些不耐噪声和干扰尖峰，但可以更快地传递更多的数据。

据笔者所知，目前没有任何商业 CPU 利用这种方法，但是这项技术在内存存储中得到了合理的应用。在持续的压力下，Intel 与其他闪存供应商共同发布“较大的”设备——这意味着可以在给定的体积下使用能存储更多位的设备。制造商通常会降低硅的特征尺寸，允许小型晶体管装进更密集的集成电路。然而，Intel 在几年前提出了更激进的设计，使用多值逻辑方法让 2 位数据存储在一个晶体管单元中。市场上如 StrataFlash（®Intel）这样的设备，在手机、MP3 播放器和个人数字助理（PDA）中得到广泛应用。

注意到多值增加的数量是收益减少的——从 1 位移动至 2 位，意味着对半分阈值电压（但可以表示数据的数据量增加一倍）。从 2 位移动到 3 位是指再次减半阈值电压（但只增加了 50% 的表示数据的数据量）。这表明了收益递减，再加上噪声的敏感度增加，因此在实践中该技术往往限制在每单元/晶体管/电线 2 位。

这里要提到的最后一点是宇宙射线辐射的影响，我们在 7.10 节曾简要地对单粒子翻转（SEU）的发生进行了讨论。由于宇宙射线撞击硅栅从而导致存储电荷的变化，这表现为一个电压波动。多值逻辑设备对电压噪声免疫力下降，这意味着这些设备最好避免用在高海拔的地方，

如飞机上、电子登山装备中和墨西哥城或拉萨等地的消费类电子产品中。

9.5.2 带符号数字的表示

带符号位（Signed Digit，SD）是有冗余的二进制表示的扩展（即有多种方式表达）。冗余来
439 自所引进每一个数字都有可能是带符号的，并且表示数字有一定的自由度。

使用符号就意味着SD数字字的每个位的位置有“1”，“0”或“-1”三种状态、实际位的权重和一个标准的二进制数相同。当然，这有一个明显的缺点：在每个位的位置必须处理负数（而不是仅仅在补码的最高有效位）。不过，这个缺点可以忽略不计，因为二进制加法器执行加减运算时是几乎相同的方式，无需额外的硬件。

下表给出了十进制值3的许多替代表示方法的SD例子：

SD向量	值	权重
(0 0 0 0 1 1)	2+1=3	2
(0 0 0 1 0 -1)	4-1=3	2
(0 0 1 -1 0 -1)	8-4-1=3	3
(0 1 -1 -1 0 -1)	16-8-4-1=3	4
(1 -1 -1 -1 0 -1)	32-16-8-4-1=3	5
(0 0 1 -1 -1 1)	8-4-2+1=3	4
(0 1 -1 -1 -1 1)	16-8-4-2+1=3	5
(1 -1 -1 -1 -1 1)	32-16-8-4-2+1=3	6

我们稍后将会看到，如果选择一种有更多零位的其他表示方法，则会在加法操作特别是乘法操作中使用更少的操作。我们定义一个带符号位数字的权值为非零数字的总数。权值越低越好，因为这会使部分积运算速度更快。

以2为基数的二进制数转换成SD表示方法，可以使用以下算法：

令 a_{-1}，a_{-2}，…，a_b 表示一个二进制数，所需的SD表示结果为 c_{-1}，c_{-2}，…，c_b。在SD表示方法下的每一位都是按照下面的方法确定：

$$c_{-1} = a_{-i-1} - a_{-1}, \quad \text{其中 } i = b, b-1, \cdots, 1$$

其中 $a_{-b-1}=0$。

为了在FPGA或类似的系统中更好地利用这种表示方法的冗余特性，应保证尽可能少用非零数字表示。这可以通过一个最小的带符号数向量实现。这是一个有最小权值的（或可能是很多）SD表示方法。

前面表中的所有例子都代表相同的数字3，表中的第一项和第二项的最小权重是2并且是最小的带符号数向量。注意第二行（0 0 0 1 0 -1）是最小的带符号数向量。此外，两个非零数字之间有一个零。事实上可以证明每一个数字的SD表示存在的条件是没有非零的数字相邻。有时
440 可能有一个或更多的表示，这些数字被称为规范的。

因此，规范的带符号数字（Canonical Signed Digit，CSD）数字是最小的带符号数向量，并且保证至少有一个零在任意两个非零数字中间。

除了有许多零参与计算从而使硬件运算量减少以外，选择CSD数字还有另外一个很好的理由。在2.4.2节中介绍的并行加法器执行加法的最高速度受向上的进位传递限制。当然，也有超前进位或预测技术，但当操作码字变大时需要大量的逻辑运算。但是，如果我们能保证一个非零位的下一个最高有效位的数字始终是零，就不可能有进位传播。

这样，使用CSD数字执行加法运算的速度非常快，没有进位传播的问题。

现在让我们看看产生这样一个数字的一种方法（此方法在由 Kai Hwang 所著，于 1979 年出版的《Computer Arithmetic：Principles，Architecture and Design》一书中进行了讨论）。

首先，用（$n+1$）个二进制位表示一个向量 $B=b_nb_{n-1}\cdots b_1b_0$，其中 $b_n=0$，每个元素 $b_i\in\{0,1\}$，$0\leqslant i\leqslant n-1$。由此看来，我们要找到长度为（$n+1$）的规范的带符号数字（CSD）向量 $D=d_nd_{n-1}\cdots d_1d_0$，$d_n=0$，$d_i\in\{1,0,-1\}$。尽管表达的格式不同，但 B 和 D 表示相同的值。

请记住，就确定一个数（事实上任意一个带符号数字向量，包括 SD、CSD 等）的值而言，二进制的一般规则适用于每个位和加权值的关系：

$$\alpha=\sum_{i=0}^{n}b_i\times 2^i=\sum_{i=0}^{n}d_i\times 2^i$$

下面提出一种基于 Hwang 方法的启发式算法。该算法获得一个二进制数的 CSD 表示，简单但逻辑性很强。

第 1 步	首先从 B 的最低有效位开始，设置索引 $i=0$ 和初始进位 $c_0=0$。
第 2 步	从 B 取两个相邻位 b_i 和 b_{i+1} 以及进位 c_i，并用这些产生下一个进位 c_{i+1}。进位的产生方式和全加是一样的：因此，$c_{i+1}=1$ 当且仅当在 $\{b_{i+1},b_i,c_i\}$ 中存在两个或三个 1。
第 3 步	在 CSD 数字中计算当前位 $d_i=b_i+c_i-2c_{i+1}$。
第 4 步	递增量 i，转到第 2 步。当 $i=n$ 时结束。

请注意，计算之前，原始二进制数的最高有效位固定为 0（这样位的位置有效地延长了 1 位）。因此，CSD 表示可能有一个额外的数字。框 9.3 中给出了 CSD 的另一个例子。 441

框 9.3 CSD 例子

让我们分析下面的 8 位二进制数：

(0 1 0 1 0 1 1 1)，十进制值是 87。

采用 Hwang 启发式算法，CSD 的表示为：

(0 1 0 −1 0 −1 0 0 −1)，值为 128 − 32 − 8 − 1 = 87。

在他的例子中，因为数字是规范的，所以没有相邻的非零数字并且 CSD 的权重是 4。

9.6 光计算

目前，很多高级研究学者把目光集中在研究可以提高 CPU 性能的新技术上。本节主要介绍基于光处理的两个有趣的想法。

任意一台数字计算机都必须具有开关。过去的 20 多年，针对光学开关技术已经投入了大量的科研努力，但是，微型全光学开关尚处于实验室研究阶段。集成光学是光学技术的一个分支，主要是利用类似于电子集成电路技术（在同一个基板上混合各种电子元器件）的构造工艺，在硅和其他基板上开发光学电路的技术。当前许多商用设备都采用了这类技术，包括多路复用器和滤波器等。

尽管全光学计算机是主要的研究目标，但是近几年，基于混合的光电系统已经开发了很多应用。采用光学信号的主要动因是它们的速度：信号是以光速进行传输的。几个信号可以在没有相互干扰的前提下（即光线交叉）同时存在于一个物理信道中，此外光学干扰要比电子干扰更容易控制。

9.6.1 光电全加器

回忆一下 2.4.2 节中带有进位传输延迟的全加器，我们所遇到的问题是直到进位从最低位到最高位都运算完时，才能输出计算结果。因此向上的传输延迟成为全加器运算速度的主要限制

因素。

光电全加器的工作原理是以光速进行进位的运算。进位的运算电路如图9-11所示，需要重点关注的是图中的x比特流和y比特流是电子信号，而输入的进位和输出的进位都是光学比特流。

$$C_{OUT} = x.y + x.C_{IN} + y.C_{IN}$$

[442] 它是一个以并行加法器呈现出来的结构，C_{IN}作为计算C_{OUT}过程的一部分，是由数据的次低位运算出来的结果。每一位有两个开关元素，输入位一旦出现，开关就会启动。换句话说，所有的开关会随着所有位运算的发生而同时发生。光学进位会以光速传输通过整个结构，后续的电路（图中未显示）用来计算每一位的输出结果（取决于输入位和C_{IN}）。这部分相较于之前的电路部分不是很重要，因为一旦所有的进位都以光速的速度运算完之后，实际的位加法会正常地运算完。

把这项技术所产生的传输延迟和一个标准的n位全加器相比较，后者的一个单独加法单元的传输延迟（几个AND门和OR门的传输延迟）是前者的n倍。这项技术是当前计算机体系结构领域中光学辅助技术研究的主题之一。

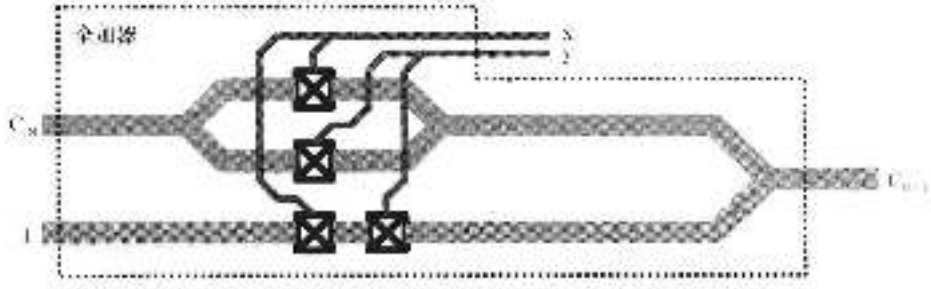


图9-11 光电全加器结合电子开关和光学通路构成一个快速加法器，其速度不会受到逻辑门的传输速度的限制

9.6.2 光电底板

随着总线不断变宽（数据总线和地址总线为64位或更宽），计算机中大量的信号连接各个模块和区块。它们的时钟也会随之加快，因此会产生更多的电磁干扰（EMI），从而导致更多的信号受到电磁干扰的影响。这也使得总线设计工作变得复杂，例如，在嵌入式单片机的设计中，12或16层的PCB板很常见。特别是，在PCB板上设计大量总线交叉的区域很困难，更困难的是（从EMI的视角来分析），存在很多并行并且长度很长的总线。

以上提到的困难的一种解决方案是采用光电技术。采用这种光学技术的最大好处是光线具有的那种可以在互不干扰的前提下相互交织或以很近的距离进行传输的特性。光学底板已经证明了光学的这种相互连接的优势。它们为每个信号输出使用了对应的光电二极管，以及为每个信号输入使用了对应的光电二极管。图9-12显示了信号输出到接收阵列路径的全像图。

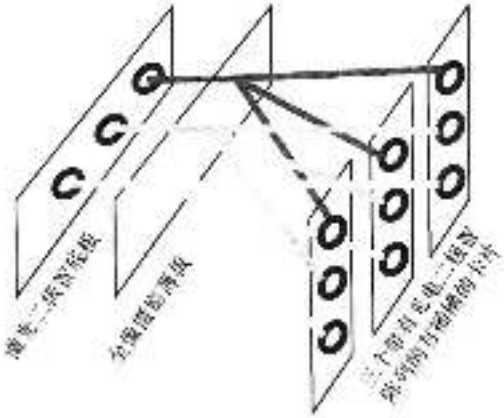


图9-12 光电底板使用了全像摄影板将光线信号从激光二极管传输器（或LED灯）传输到多个接收阵列，在空闲区域分裂成多条光线，最终将分裂后的光线信号定位在物理上分隔开的有插槽的卡片上

光学底板没有最大时钟速度（它们仅仅受限于激光二极管的调制和光电二极管的带宽），时钟的速度是非常快的——至少是以 GHz 来计量的。这些底板可以进行热插入（即在系统运行过 443
程中，将没有插槽的卡片做成有插槽的，但是底板信号并不改变）。相比之下，快速的电子总线则需要根据不同的负载定位信号传输的目的地，因此快速传输的总线并不支持热插入。

由于没有电子触点造成腐蚀、降解或机械磨损，因此这样的系统能够达到高可靠性。

但是，插槽卡片准确无误的校准是必需的，以使光线信号只能准确地射中光电二极管位置。这要假定光线是在空闲区域中进行传输的，尽管同样的技术有可能会用于其他的光学透明介质中，如硅酸盐。

9.7 科幻小说还是未来的现实

或许本不该设立本节。因为本节涉及的内容在现阶段无法得到验证，而且它的相关知识并不被包含在任何的教学大纲之中。不过，不妨让我们把这部分内容当成研习本书至此的一种奖励，也可以作为对计算机科学边缘研究领域那些令人叫绝的想法的惊鸿一瞥。

9.7.1 分布式计算

正如前言中提及的那样，计算领域正在变得越来越嵌入式。有趣的是，这种变化与无线连接的趋势相一致。随着科技的进步，人们慢慢得到了一个逻辑聚集点，这个聚集点关注这样的一个未来，即未来通过无线网络相连的微型化处理单元在数量上将是人口数量的数千倍。尽管这些处理器将被专用于不同的功能（如微波处理器、电话处理器、中央供暖\空调处理器等），然而 444
在给定的时间内，它们极有可能不需要执行其原有的功能（即处于空闲状态）。

简单地允许空闲处理器进行通信和合作带来的集合计算能力比现阶段台式机的计算能力要强上好几个数量级。通过提出新形式的人机接口并提高软件水平，我们可以期待在未来能够与自己专用的计算机特征进行交互操作。这种特征会被设置在一系列不断变化的处理器中，但是会给用户提供一个一致的接口。这将会是一个分布式的程序，可以进行远程访问，可以作为我们的私人助理。

这听起来像科技小说吗？其实基础技术现阶段已经存在了。

9.7.2 湿件（大脑）

假设更进一步进入科幻小说的领域，如果基因技术和生物计算持续进步，我们将在下一个十年内开始看到在生物机器上进行的计算。

如果我们考虑到现存的最复杂、能力最强的计算机是在我们的头脑中，我们就会发现前面的预测并非牵强附会。另外，随着医疗分析技术的进步，越来越多的关于人类和哺乳动物大脑运作方式的细节被揭露出来。这里吸引人的地方在于计算能力：尽管有着数十年的发展，然而人类还没有发明出具有人类自身大脑能力的计算机，除了在快速的固定计算任务这一领域。而在几乎所有其他方面的比较中，人（甚至哺乳动物）的大脑都取得了胜利。既然大自然已经设计了如此令人称奇的生物计算机器，那么也许我们可以模仿这种设计，或者直接驾驭它。

我们可以在这里确定几个潜在的未来。最简单的一个是生物或者化学的构筑模块将被用于执行计算。一些像人造生物计算机之类的东西将基于人脑的结构进行设计，它的处理过程将在人造生物神经细胞上运行。至少一种生物晶体管（控制开关）已经经过论证，因此我们知道逻辑功能是可能实现的。然而，与简单地将使用硅的方法复制到生物结构中相比，新型结构也许会更加适合生物计算。举例来说，模糊联想集（fuzzy associative set）不等同于离散二进制计算。

第二个潜在的未来是人脑的功能将随着人工智能的加入而加强。使用电子传感器和刺激器

直接与人脑和神经系统交互，这种技术已经存在几十年了，如用于盲人的视觉系统和用于听力受损的耳蜗植入技术。很容易想象，我们可以通过设置一种协处理器来实现计算机单元与人脑
445 的连接。这样至少可以对经常出现的有关人机交互的课题有所帮助。

就个人而言，作者很喜欢自己大脑的工作方式，但是用自然发展的眼光来看，正在发展的人工神经网络可以帮助残疾人，包括学习有障碍的人，进而扩展到帮助那些健全的人。这种增强方式可能会有很多形式，但是会包括一种用于真实游戏体验的更高带宽的计算机接口，一种记忆唤醒装置或者感受超越五感（视觉、听觉、嗅觉、触觉、味觉）的一种感觉。一旦一种可以与人脑相兼容的接口技术得到解决，那么进步的可能性将是无限的，不过真正的进步主要在于发展人造全生物计算机（这至少意味着你不需要到哪儿都带着电池）。

9.8 小结

在本章中，我们探究了未来计算机的方向。一开始我们讨论了一些比较靠谱的预测，例如单比特体系结构、VLIW、并行和异步系统（靠谱在这里指的是一个既定的技术，已经与我们关系密切，尽管近期主要限于专门的处理应用）。并行处理已经提上 Intel 未来的议程：双核、四核和八核处理器已经实现，不难想象这一趋势将继续延续下去。大规模并行计算也是一个靠谱的预测，因为我们大多数人已经感受到这样的计算给世界带来的好处。

本章概述了数字格式上可能的变化，这些替代表示方法不仅对专门的计算领域有重要影响，而且可能会影响未来的主流计算。

除此之外，我们认为，尽管光电混合成为可行的技术已有二十多年，但它还没有对计算机世界产生任何重大影响。

最后是科幻小说。实事求是地说，科幻小说是一条引领科学和工程研究发展的途径。无论是声波螺丝刀和时光机器，向量阵列和星舰企业（Starship Enterprise）的运输机，还是《星球大战》中的机器人和激光剑，大多数工程师都会受到那些技术幻想的启发。让我们尝试和维护那些很“酷”的技术因素，如果说有自我意识的计算机可能有点非我们力所能及，但我们还是可
446 以一起创造伟大的、跨越式的进步，那将不只是进化，而是革命性的创新。

内存大小的标准表示方法

在学校里大多数人学习国际单位制（简称 SI），其中单位前缀是 10 的幂，例如毫米表示 10^{-3} 米，厘米表示 10^{-2} 米，公里表示 10^{3} 米。

下表是一些常用的前缀：

前缀名称	前缀字母	倍率
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}

然而，当遇到以 2 为底的幂方（如 2，4，8，16，32，64 等）结构来表示计算机内存大小的情况时，国际单位制不仅不方便，而且还容易混淆。

由于 2^{10} 的实际值为 1024，非常接近 1000，因此 2^{10} 已经被当做“千”（kilo）来使用。所以，通常使用的 1kbyte 实际上是 1024byte，而不是国际单位制中的“千”。尽管这个差异在日常使用中不会造成不便，但确实有很多场合需要更加精确，而且在这些场合这种非国际单位制的使用会造成混淆。

因此，国际电工委员会（IEC）引入了一种新的、非歧义的计算机数据存储术语集。该术语集与国际单位制相似，但是也有区别。在计算机可用的数据大小范围内，有以下前缀：

前缀名称	前缀字母	倍率
exbi	Ei	2^{60}
pebi	Pi	2^{50}
tebi	Ti	2^{40}
gibi	Gi	2^{30}
mebi	Mi	2^{20}
kibi	Ki	2^{10}

因此，如果一台计算机硬盘的容量有 1TiB（tebibyte），那么实际上它包含 1 099 511 627 776byte，比 1 TB（terabyte）的硬盘容量（即 1 000 000 000 000byte）多出近 10%。

IEC 单位制得到了 IEEE 和其他一些机构的认可，在本书中也得以采用。

示例

128Kibyte
128KiB
128Kibibyte
相当于：$128 \times 2^{10} = 131\ 072$byte

20Mibyte
20MiB
20Mebibyte
相当于：$20 \times 2^{20} = 20\ 971\ 520$byte

500Pibyte
500PiB
500Pebibyte
448 相当于：$500 \times 2^{50} = 562.96 \times 10^{15}$byte

开放系统互连模型

B.1 引言

开放系统互连（OSI）模型或参考系统，首先由国际电信联盟（ITU）的开放系统互连（OSI）组提议，然后作为建议 X.200 与国际标准化组织（ISO）联合推出，成为一种计算机互连分层方法。

现在我们知道很多名词的缩写，我们使用缩写“OSI”代表该模型。OSI 模型由许多的层次构成，这些层次被用于划分计算机互连中的逻辑连接性和功能。每个层次可以由层号或层的名字来标识，我们将在下文中提到。

OSI 模型普遍适用于网络连接协议，但是在本书中，我们仅使用这个模型的“低层”，也就是那些更接近于硬件的层次。特别用它来讨论通信和总线系统，从低层硬件和电压细节中分离出诸多协议。

对于不太了解 OSI 模型的读者来说，这种分层方法看起来似乎有点不必要。然而，可以肯定的是，当事情变得复杂的时候，特别是在嵌入式系统中，这种分层方法确实简化了系统设计和理解，因此我们在这里做简要的介绍。

B.2 OSI 层次

基本的 OSI 模型包含 7 个水平堆叠的层次。从底向上，它依次包含了从比特级的信号到使用该信号的各应用的所有内容（例如跨越了从 1000BASE-T 以太电缆上的电压变化到一个互联网银行系统上的所有中间步骤）。[⊖]

OSI 层次	层次名称	数据单元
7	应用层	数据
6	表示层	数据
5	会话层	数据
4	传输层	段
3	网络层	包
2	数据链路层	帧
1	物理层	比特

OSI 模型的思想是每层次只与该层次在堆栈中紧邻的上下两个层次通信，并且通信方式也被严格定义。因此，每个层次的开发者仅仅需要关心与相邻层次的通信即可。这样的细分使得我们在通信中有更多的规律可循，并且在理论上有更高的可靠性。

第 1、2 和 3 层与传输介质有关，而第 4～7 层被称为主机层。低层次更倾向于由硬件实现，而高层次则倾向于由软件实现（有人可能会说代码量也会自底向上递增）。接下来，我们将对介质层逐一进行分析。

⊖ TCP/IP 分层模型和 OSI 模型应用了相同的原理，仅在分层方法和层次命名上有所不同。

第 1 层：物理层

物理层包括了一个单元与某个通信介质的电气连接，例如，数据总线中的电线、时序和电压。物理层负责确保通信单元能够与传输介质“对话”和“监听”（这种传输包括有线传输、无线传输、光纤传输等）。

这一层负责建立与介质的连接，且参与到调度中使得不同单元能共享传输介质（在适当的时候），同时将传输出去的信号从逻辑比特转化成介质所期望的格式，并将收到的信号转化为逻辑比特。

基本上，物理层将来自数据链路层的通信逻辑请求转化为具体的硬件传输或信号接收。在计算机网络中，处理物理层信息的设备往往被称做“PHY”。

第 2 层：数据链路层

数据链路层（DLL）在物理层的物理通信的基础上，规定了一种点对点或（多）点对多点的结构。通常，它要求对物理层中发生的错误做纠错处理，从而为网络层提供一种无错的帧接口。

实际上，数据链路层有两个子层：介质访问控制（MAC）子层和逻辑链路控制（LLC）子
450 层。如果这两层都存在，那么 MAC 面向物理层接口而 LLC 面向网络层接口。MAC 层将需要传输的数据以帧的格式封装起来，校验收到的数据，并在传输介质被多个单元共享的情况下提供仲裁、流量控制等功能。而 LLC 层为更高层提供纠错和流量控制等功能。

一些物理（PHY）设备同样也包含一个 MAC 层，由此这种设备被称为“MACPHY”设备。

第 3 层：网络层

网络层允许传输和接收任意大小的数据包。网络层的通信是端对端的，因为网络层可以向（从）一个特定的接收者发送（接收）数据包。而实际实现这些请求则是较低层的功能。

第 1 ~3 层，以及其他层

在网络层之上的层被称为主机层。它们负责在特定的主机之间传输大量的信息，建立和维护主机之间的通信会话，允许不同的信息共享链接并且提供到同一个特定应用的接口。尽管这些对于基于因特网的应用非常重要，但是它们往往不属于嵌入式计算机体系结构的领域，因此我们更关注底下三个层次。

举一个 OSI 系统中的例子，考虑一个即将被传送的数据包。它与地址信息一起被传送到数据链路层，从而指明该数据包要去哪里。数据链路层将收到的信息分割成帧，也许会在将信息以比特的形式传给物理层之前对它进行加密，然后交给物理层传输。紧接着，物理层会以一个给定的时序，通过在线路上驱动高低电压来调制线路从而以并行或串行的方式传送数据。

B.3 小结

本书主要在 6.3 节介绍了几个第 1 层的例子，包括 LVDS、EIA232 等。我们同时也讨论了一到两个第 2 层的例子，例如以太网。然而，我们讨论的很多总线系统，如 USB、SCSI 等，实际上同时包含了模型的低两个或三个层次。

最重要的一点是，尽管诸如 EIA232 和 USB 系统定义了物理层的连接，但是 OSI 模型分层抽象使得它们都能透明地给高层传输任何它们想要的信息。例如，EIA232 和 USB 都能使一台 PC 连接到因特网并传送 TCP/IP 数据包（这反过来可以传送超文本传输协议即 HTTP 网页）。USB 也可以传送文件给拇指驱动器（thumb drive），或与外部音频硬件之间传送音频数据。

正是这种灵活性——抽象成为层的思维方式——成为了许多现代系统的特征，尤其是当网
451 络互连越来越普遍的时候。

探索 cache 大小和结构安排的权衡设计方法

C.1 引言

本附录将介绍两个用来评估和调查 cache 配置的软件工具 Dinero 和 Cacti 的使用。[⊖]Cacti 是一个描述 cache 访问时间、时钟周期时间、区域、长宽比和功耗的综合模型。计算机体系结构设计师利用 Cacti 模型可以更好地理解 cache 不同大小和结构安排所带来的性能权衡。Dinero 是一个跟踪驱动 cache 的模拟器，它用输入来跟踪 cache 的设计参数，以确定 cache 的性能（主要测量 cache 的命中率）。一次跟踪就是由一个程序所访问的内存序列（包括指令和数据存储器），或者是由程序的解释执行得到的内存序列，或者是由编译器向程序调入调试代码得到的内存序列。

惠普研究院的 Premkishore Shivakumar、Norm Jouppi 以及 Mark Hill 和 Jan Edler 分别是 Cacti 和 Dinero 的作者，制作和发布了这两个工具。需要注意的是，Dinero 和 Cacti 都是有版权的软件而不是开源代码。尽管如此，两个软件的作者允许这些代码用于非商业和学术研究。

C.2 准备工作

同本书中的其他例子一样，假设读者已经访问了一个运行 Linux 的标准计算机。虽然本书作者倾向于 Kubuntu 或者 Mandrake，但其实目前任何版本的操作系统都已经足够了。也可以在（虽然可能并不容易）Microsoft Windows 下的 MacOS-X 和 Cygwin 上运行这些工具。所有的“行为”都可以在命令行上实现。 452

1. 从下面的网址下载 Cacti 3.2：

http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html

所需的文件是 cacti3.2.tar.gz，可以在有关 Cacti 3.2 的章节中名为 gzip'ed 压缩文件的超链接下找到。也可以使用互联网搜索引擎来寻找这个文件的其他信息库。

2. 从下面的分布源下载 Dinero Ⅳ：

http://www.cs.wisc.edu/markhill/Dinero Ⅳ

所需的文件是 d4-7.tar.gz，它位于标记为 Wisconsin 的链接下。这个文件也可以通过互联网搜索引擎找到。

还有一个可用的在线版 Cacti，位于：

http://www.ece.ubc.ca/~stevew/cacti/

C.3 安装 Cacti 和 Dinero

1. 将源文件（cacti3.2.tar.gz 和 d4-7.tar.gz）拷贝到一个工作目录下。

2. 建立 Cacti。

创建一个名为 cacti 的新文件夹：

⊖ 我们介绍的 Dinero Ⅳ和 Cacti 3.2 会随着时间而更新，因此具体的选项和指令可能会有所改变，但是对性能的研究依然是有效的。

```
mkdir cacti
cd cacti
tar xvzf  ../cacti3.2.tar.gz
make
```

这会产生 Cacti 可执行文件。

```
cd ../
```

3. 建立 Dinero：

```
tar xvzf d4-7.tar.gz
```

该操作会生产一个名为 d4-7 的子文件夹。

```
cd d4-7
./configure
make
```

这会产生 Dinero Ⅳ可执行文件。

```
cd ../
```

C.4 工具的使用

下面进行一个实验，来说明使用 Cacti 和 Dinero 设计不同的指令 cache 和数据 cache，就像设计嵌入式处理器 cache 一样。指定每一个 cache 的限制区域，使得在运行测试程序时 cache 达到效
453 率最大化。

首先，我们来介绍一下要使用到的设计流程。

1. Cacti 可以用来“创建”一个 cache：

```
./cacti/cacti C B A X Y
```

其中 C 代表以字节为单位的 cache 的大小（即它的容量），B 是块大小，A 代表关联性。在这种情况下，我们将设置 X = Y = 1。

2. 可以给定一个样本 cache 的 B 和 C 参数，在其上运行 Cacti。请注意，在运行过程中 Cacti 会产生很多信息。搜寻这些信息，可以看到一个名为“Total area One subbank”的输出域，这就是样本 cache 将要占用的区域。

3. 通常，需要设计几个采用不同输入参数的 cache，在每一种情况下，记下每个设计的 cache 区域。

4. 使用下面的命令（都在同一行上）运行 Dinero Ⅳ：

```
./d4-7/dineroIV cache-config -informat p
< d4-7/testing/mm.32
```

其中 *cache-config* 即缓存配置是下面两行之一：

对于 I-cache: `-l1-isize` *capacity* `-l1-ibsize` *block-size* `-l1-iassoc` *associativity*

对于 D-cache: `-l1-dsize` *capacity* `-l1-dbsize` *block-size* `-l1-dassoc` *associativity*

-l1 是指 1 级，-size 表示指令 cache 的大小，-dassoc 是指数据 cache 的关联性。输入的 mm.32 是在 cache 上运行的测试文件，它包含在 Dinero Ⅳ的包中。

如果在样本 cache 上运行 Dinero Ⅳ（从第二项开始），通过测试程序同样会产生很多的信息。

Dinero Ⅳ 可以同时模拟指令 cache 和数据 cache。因此，为了防止混淆，我们需要分开来看（因为我们将以缺失率来作为衡量性能的标准，所以每一个 cache 需要独立工作——除非它们必须共享有限的区域份额。有关这方面的重要信息见名为“Total Demand misses”的域）。

5. 需要记录下不同 cache 设计的总的需求缺失率，以达到通过改进和尝试不同的 cache 来探索权衡设计的目标。

C.5 不同 cache 设计方案的实验

为了说明如何使用这些工具，将规定每个 cache 的区域大小不超过 $0.9cm^2$。对于那些想要将硅片面积分配在集成电路上的“真正的”CPU 设计者来说，这是一种求实的态度。对于为 FPGA 454
软核处理器设定 cache 内存的 FPGA 设计者来说，这同样是一种求实的态度——我们相信本书的读者会经常看到这样一种态度。在这种情形下，每种不同的使用情况会有不同的平方厘米大小，但是权衡设计的理念不会改变。

假设采用哈佛体系结构（见 2.1.2 节），我们将建立两种 cache：I-cache 和 D-cache。我们调整设计的参数来获得 cache 的最佳性能（用总缺失数来衡量，即 $I_{misses}+D_{misses}$）。

每个 cache 的参数会有所不同：cache 的大小，关联性和块大小。这些参数定义了一个多维的设计探索空间。一个穷尽测试应该试遍所有组合方式（但是，那就是筋疲力尽而不是穷尽了）。因此，可行的方法是运行一些设计来确定不同参数造成的探索空间的不同结果，随后“缩小”出最佳的设计方案。

在这种情形下，可以通过限定可使用的参数值来简化测试。首先，只用 2 的幂值（例如 1，2，4，…，8192 等）。其次，给予特定的大小，根据一些经验限定的关联性最大值为 32，块大小应该在 8～64 字节范围内。使用这个工具时，其他的一些值可以放心地舍弃。

在指定的 cache 上运行追踪文件 mm.32 时，总的 cache 面积和总缺失数将作为数据 cache 以及指令 cache 的设计指标。在一个嵌入式系统中，可以指定一个“真正的”追踪文件，这个文件可以从运行在系统中的代码得到。

因此，在给定的面积限制下，可以在实际运行的软、硬件上寻求到最佳的 cache 设计。

C.6 cache 设计中的进一步信息

以下是一些有用的建议，可以对 cache 的设计过程有所帮助。

建议 1：首先，确定那些对 cache 产生最大面积限制的参数组合。它们中的一个有可能是最佳性能的解决方案。

建议 2：由于我们只将缺失数作为性能的主要标准，两个 cache 性能的测量又是相互独立的，因此，要分别在 I-cache 和 D-cache 上运行 Dinero（即运行只有一个指定 I-cache 的 Dinero，再运行只有一个指定 D-cache 的 Dinero；将结果结合在一起就相当于运行了同时指定 I-cache 和 D-cache 的 Dinero）。如果考虑其他的性能测量方法，则可以结合 cache 的研究过程。

建议 3：Cacti 和 Dinero Ⅳ都有 readme 文件，可以提供信息。而且，Dinero 有内置帮助：

```
./dineroIV -help
```
455

在本附录中，需要认识到的是，我们忽略了几个在真实世界中的性能指标。最重要的是 cache 的访问时间（由 Cacti 报告的每次访问时间），它会随着设计参数的改变而改变并影响程序的执行速度，和 cache 缺失率一样，对 cache 而言都很重要。

此外，尽管 Cacti 和 Dinero Ⅳ中参数的缺省值是合理的设定，但在现实世界中，还需要根据硅的特征尺寸和硅的制作工业标准来设定，或者分别根据 FPGA 设计参数来设定。

在现实世界里，还可能要调整一些参数，比如带宽和时钟速度。

本附录以一种谨慎的态度，准确地描绘了 cache 的权衡设计方法。说明了如何使用 Cacti 和
456 Dinero 在真正的约束下探索设计空间和确定最佳的 cache 设计。

思考题

C.1 根据设计空间的测试方案的顺序，完成表 C-1 的内容，从而确定最好的缓存设计。

C.2 解释 cache 的设计参数与 cache 的区域面积在整个设计空间中是如何相关联的。

C.3 提供一个理由，证实你所观察到的。

C.4 可以到达表 C-1 中最佳解决方案的 95% 性能的最小 cache 区域是多少？

表 C-1 缓存权衡设计参数记录表

缓存 I 大小（字节）	缓存 I 块的大小（字节）	缓存 I 块的关联值	缓存 D 大小（字节）	缓存 D 块的大小（字节）	缓存 D 块的关联值	缓存 I 面积（平方厘米）	缓存 D 面积（平方厘米）	总面积（平方厘米）	缓存 I 不命中次数	缓存 D 不命中次数	总不命中次数

457
~
458

嵌入式计算机上的无线技术

D.1 引言

和嵌入式计算机一样，无线技术是一个非常先进并且处于快速发展阶段的领域。尽管它离计算机体系结构这一主题有点远，但由于越来越多的嵌入式系统需要通过无线互连，因而它在嵌入式系统中是一个非常重要的话题。因此未来无线互连将像嵌入式计算机一样得到广泛应用。

大型计算机的设计目的是用于数值运算，而嵌入式系统是为了其自身应用特点而设计的。假如其设计需求是连接，则必须从体系结构的角度来满足需求，正如大型计算机要满足数据传输和处理需求一样。

本附录目的是为了呈现目前最主要的无线连接系统。其中将对每一种系统进行简要描述，并对嵌入式系统中的相关设备进行讨论。对诸如协议栈需求等也将进行重点探讨。

然而，无线设备是在不断变化的——数据传输速率和通信范围每年都在增加而能耗却在降低。新设备的出现导致了旧设备被淘汰。甚至当你读这个附录的时候，世界上某个角落就有可能诞生了一种新的无线设备。

本附录将会覆盖一些主要的无线设备（和一些不常用的设备），并以设备、范围和带宽等的总结表结束。附录的最后将会给出一个小的应用系统实例，我们会在一个标准的片上系统处理器（基于 ARM9 内核的三星 S3C2410）上增加一个无线连接来进行演示。 459

D.2 802.11a，b 和 g

IEEE 于 1999 年底通过了著名的无线网络标准——802.11b。不久后又批准了 802.11a，它使用一种新的编码方案——正交频分复用技术（OFDM），提高了数据传输速率和无线信道可用性。802.11a 比 802.11b 更快，在 5GHz 频率范围上支持 54Mbps 最大速率，而 802.11b 仅在 2.45GHz 频率范围上支持 11Mbps 的最大速率。

802.11g 的峰值数据传输速率为 54Mbps，听起来似乎很快，但是其差不多一半带宽消耗在传输开销上。一个 Wi-Fi 设备工作时通常需要 30～100mW 的功耗，而最大通信距离为 50～100m。

嵌入式系统 802.11a/b/g 解决方案

当前一些支持单芯片 802.11a/b/g 的 Wi-Fi 解决方案包括：

- Atheros 有很多用于固定与移动应用的设备（如 AR54xx 系列——FBGA[⊖]：13mm × 13mm）。
- Broadcom BCM 4xxx 和 5xxx 系列设备（如 BCM4328——FBGA：10mm × 10mm）。
- CSR 的 UniFi 系列（如 WLCSP[⊜]中的 UF6026：3.7mm × 4.2mm）。
- 德州仪器的 WiLink 4.0/5.0/6.0 版本（如 WL1253——BGA：6mm × 6mm）。

⊖ FBGA：细间距球栅阵列（fine pitch ball grid array）。

⊜ WLCSP：晶圆级芯片规模封装技术（wafer-level chip scale packaging）。

D.3 802.11n

802.11n 草案的产品已经出现在市场上并且嵌入到许多消费类设备中。据称 IEEE 802.11n 的最大数据速率为 600Mbps 并且保证最小速率为 100Mbps（在减去协议管理特征如前导码、帧间距、应答和其他控制开销的情况下）。它使用的是多输入多输出（MIMO）技术。

目前 802.11n 解决方案据称可达到 300Mbps 左右的传输速率，通信距离为 50m。在这个速度下，访问点需要通过千兆级以太网设备连接到基础设施以保证连接质量。

嵌入式系统 802.11n 解决方案

802.11a/b/g 解决方案中列出的生产商以及以下生产商都生产出了支持 802.11n 草案的无线局域网设备：

460

- Marvell TopDog。
- Metalink（MtW8171/MtW8151）。
- Qualcomm/Airgo（WFB4030/WFB4031）。

其中一些是使用多芯片的解决方案，但是现在通过单芯片基本上都可以解决。

D.4 802.20

802.20 也称为移动宽带无线访问（Mobile Broadband Wireless Access，MBWA），授权工作在 3.5GHz 频段以下，峰速数据传输速率超过了 1Mbps。它支持多种移动速度最高达 250km/h 的车载系统，在城域网（MAN）中通信距离可以达到 8km。

D.5 802.16

802.16 也称为全球微波访问互操作性（Worldwide Interoperability for Microwave Access，WiMAX），属于无线宽带技术，支持点对多点无线（PMP）访问。802.16 作为一个确定的无线标准发布于 2002 年，它基于视距（line-of-sight，LOS）技术，旨在向那些不容易铺设光缆和铜线等基础设施的地方提供 T1/T3 级别的服务。

802.16 目标是商业用户，它工作的许可带宽为 10～66GHz，信道带宽在 20MHz、25MHz 到 28MHz 范围，并且需要基站和用户之间的 LOS。数据传输速率可以达到 134Mbps，但是通信距离限制在基站的 2～5km 范围内。这个技术可以对工作参数、频率带宽、数据速率、范围等进行不同的配置。

802.16 解决方案

目前一些提供支持 802.16a/d 标准的芯片组的生产商如下：

- Atmel（如 AT86RF535B QFN-56：8mm×8mm）。
- Fujitsu Microelectronics America（如在一个 BGA-436 封装中的 MB87M3550）。
- Intel（如 Intel PRO/Wireless 5116：PBGA-360）。
- Sequans Communication（如 SQN1010——PBGA-420：23mm×23mm）。
- TeleCIS（如 TCW1620）。

目前一些支持 802.16e 标准的芯片组如下：

- Altair Semiconductor（如 ALT2150）。
- Intel（如 Intel WiMAX Connection 2250：PBGA-360）。
- NXP（如 UXF234XX——HVQFN48：7mm×7mm）。

- Runcom Technologies Ltd（如 RNA2000/RNF2000）。
- Sequans Communication（如 SQN1130——VBGA-256：11mm × 11mm）。
- Wavesat（如 UMobile WiMAX 系列）。 461

D.6 蓝牙

蓝牙[⊖]，最初是由爱立信公司开发的，但现已成为世界标准，是一种短距离通信技术，旨在取代电缆连接的便携式或固定设备，同时保持高度的安全性。蓝牙设备可以在短距离内互相连接和通信，组成自组织网络（又名微微网，piconet）。

每个设备可以同时与其他 7 个设备在单一微微网内相互通信。此外，每个设备也可以同时属于几个微微网。微微网可以动态建立并会在蓝牙设备进入和离开无线区域时自动启动。蓝牙 2.0 设备与增强型数据速率（EDR）的组合于 2004 年 11 月开始采用，速率可达 3Mbps。

蓝牙技术主要应用于工业、科学和医疗（ISM）领域，工作在 2.4 ~ 2.485GHz 频段，主要有以下几种分类：

- 三类无线工作范围为 1m（最大发射功率为 1mW）。
- 二类无线工作范围为 10m，经常用于移动设备（最大发射功率为 2.5mW）。
- 一类无线工作范围为 100m，大多用于工业领域（最大发射功率为 100mW）。

两个设备之间的蓝牙连接有三种安全模式：模式 1 实际上是非安全模式，模式 2 提供服务级强制安全，模式 3 加强了链路层的安全。

每个蓝牙设备在进行所有的蓝牙通信时几乎都会涉及两个参数：第一个参数是一个唯一的 48 位地址——蓝牙设备地址（BD_ADDR），在生产时就已分配好，记录在蓝牙设备的硬件上，它不能被修改；第二个参数是一个自由运行的 28 位时钟，每 312.5μs 一个滴答，这个时间恰好与 1600 次/秒的跳频停留时间一致。

在省电模式下，蓝牙设备的工作电流约为 30μA，唤醒和响应需要消耗几秒钟的时间。通过使用 TCP/IP 协议，蓝牙设备实际上可以跟与互联网连接的任何其他设备进行通信。

蓝牙解决方案

一些嵌入式蓝牙芯片组如下：

- Broadcom（BCM20XX）。
- CSR（BlueCore 系列）。 462
- Infinneon（PMB8753——WFSGA-65：5mm × 5mm 和 PBA31308）。
- NXP（BGB210S——TFBGA-44：3.0mm × 5.0mm）。
- STMicroelectronic（STLC2500C——WFBGA-48：4.5mm × 4.5mm）。
- Texas Instruments（BRF6300 BlueLink 5.0）。

在编写本书时蓝牙 3.0 标准已经提出，它的数据传输速率超过 400Mbps。

D.7 GSM

1982 年，欧洲邮政和远程通信会议（CEPT）成立了一个研究小组，负责为整个欧洲移动电话系统制定一个标准。这个小组名为 GSM（GroupeSpeciale Mobile）。GSM 小组在 1989 年演变为欧洲电信标准协会（ETSI）。

⊖ 蓝牙是以传奇的 10 世纪挪威国王命名的，他把遥远的斯堪的纳维亚部落统一成一个团结的王国。大概是为了彰显爱立信向诺基亚进军的野心。

“GSM”缩写的含义已从“GroupeSpeciale Mobile”改成了“Global System for Mobile Communications”（这是为了展示这个欧洲标准要成为全世界标准的雄心）。GSM 是当前使用范围最广泛的手机系统：一种开放的数字蜂窝技术，用于传输移动语音和数据服务。它被归类为第二代（2G）蜂窝移动通信系统。

虽然 GSM 极适合语音通信，但是它仅支持 9.6kbps 的本地数据传输速率。它的基本数据传输服务 SMS（短消息服务）可以同时发送 140 个字节，或在通常方式下将数据挤在一起，允许发送 160 个 ASCII 字符（140×8 位/7 位）。

GSM 带有中等水平的安全设计。它使用共享密钥加密（shared-secret cryptography）的方法验证用户。用户与基站之间的通信可以被加密。GSM 只对使用 GSM 网络的用户进行身份验证（反之则不然）。因此，这种安全机制虽然提供保密性和身份验证，但仅限于有限的授权，并且没有不可否认性（non-repudiation）。GSM 针对安全性所使用的几种加密算法虽然理论上强度合适，但是还是可以通过一些方法破解。

GSM 解决方案

下面列出了一些目前使用的 GSM 芯片组：

- Broadcom（如 BCM2124——FBGA-296：10mm×10mm）。
- Infineon 的 E-GOLD 系列。
- NXP 的 AeroFONE（PNX490——PBGA：10mm×10mm 和 PNX4905——PBGA：12mm×12mm）。
- 德州仪器（如 LoCosto ULCGSM TCS2305 和 LoCosto ULCGSM TCS2315）。

463 以上所有的芯片组都支持通用分组无线服务（General Packet Radio Service，GPRS）。

D.8 GPRS

GSM 扩展到 2.5G 主要是由于通用分组无线服务（GPRS）技术。GPRS 把分组交换机制加入到了 GSM 当中。采用 GPRS 连接，使得手机一直处于联网状态并可以立即传输数据，让用户可以像拨号上网一样随意进行访问，而且支持在任何地方都可以保持连接并拥有更高的数据传输率（一般来说是 32～48kbps）。与基本的 GSM 不同，GPRS 在进行数据传输的同时仍然可以接听电话。GPRS 已经覆盖当前蜂窝网络，其优点是使用了 IP（Internet Protocol）协议传输的特性。

由于 IP 协议是按分组传输，网络不需要连续的数据传输，因此 IP 传输可以支持信道共享。一个用户可以在其他人读信息的时候收发数据，另一个用户在这个时候不需要占用信道。所以，GPRS 要比不论是否传输数据都会占用信道的电路交换网络（2G）高效。

GPRS 手机的级别可以决定其数据发送速度，从技术角度来说，这种级别是指它们能够用于上传（从手机发送数据）和下载（从网络接收数据）的时隙数量。每个信道被分成 8 个时隙，每个时隙上最大的数据传输率为 13.4kbps。其中一个时隙用于控制传输，而语音传输一般来说要占用两个时隙。

如果将 8 个时隙都分配给了同一个用户，从理论上讲，将获得最高 171.2kbps 的数据传输率。用户可以获得的最大传输速率是通过采用 4+1 这种方案（4 个时隙用于下载，1 个时隙用于上传）获得的，为 53.6kbps，但是实际中只有 40～50kbps。GPRS 设备也可以根据其处理 GSM 语音通话和 GPRS 连接的能力进行分类：A 类型的手机可以同时连接到 GPRS 和 GSM 服务。B 类型的手机可以同时绑定 GPRS 和 GSM 服务，但每次只能使用一种服务。B 类型的手机支持在 GPRS 连接时拨打、接听电话或者发送、接收 SMS 信息。在通话或者收发 SMS 信息时让 GPRS 服务挂起，然后当通话或者 SMS 会话结束时自动恢复。C 类型的手机可以绑定 GPRS 或者 GSM 语音服

务中的一种，使用这种类型手机的用户需要在两种服务之间切换。

D.9 ZigBee

ZigBee，即 IEEE 802.15.4 无线个人区域网（WPAN）标准，于 2004 年被批准，它特别针对嵌入式应用。ZigBee 层在 802.15.4 协议顶层，带有 mesh 网络、安全和应用控制。ZigBee 应用所关注的重点为低功耗、高密度网络节点、低成本以及实现便捷性。 464

ZigBee 设备可分为三种类型：网络协调器（Network Coordinator）、全功能设备（FFD）和精简功能设备（RFD）。其中只有 FFD 定义了完整的 ZigBee 功能，并能成为网络协调器。RFD 资源有限，由于它是一个低成本终端的解决方案，所以它不允许使用一些高级的功能（如路由）。每一个 ZigBee 网络有一个指定的 FFD 作为其网络协调器。

协调器作为管理员并组织网络。ZigBee 的寻址空间为 64 位的 IEEE 地址设备，并支持多达 65 535 个独立的网络。它支持多种网络拓扑结构，其中包括星形、点对点和网状。ZigBee 具有主从配置功能，非常适合于许多仅仅通过小数据包交互且不常使用的设备。这方面就意味着，ZigBee 非常适合于建立自动化系统、照明控制、安全传感器等。

ZigBee 的另一个重要特征是低延迟：当一个 ZigBee 设备断电（除了一个 32kHz 的时钟外所有电路都关闭）时，它可以在 15ms 内醒来，然后传输一个包。这个延迟同时也提供了一些能耗上的优势（它可以快速地将设备打开进行传输，然后再将设备转入睡眠模式，这样就能实现非常低的平均功耗）。

ZigBee 定义了多个信道：0 号（868MHz），1 号到 10 号（915MHz）和 11 号到 26 号（2.4GHz）。这些频段的最大数据传输速率分别为 250kbps（2405～2480MHz 之间，在全球范围使用），40kbps（902～928MHz 之间，在美洲范围使用），20kbps（863.3MHz，在欧洲范围使用）。当然，这些速率都是理论上的原始数据，并不是实际可达的。由于协议的开销，实际的数据传输速率将低于这些数据。

ZigBee 的数据包长度是 127 个字节，其中包括包头和 16 位的校验位，数据的有效载荷可达 104 个字节长度。无线信号的最大输出功率一般为 1mW，范围达 75m。ZigBee 对软件的可配置选项包括加密和认证，密钥处理和帧保护。当使用 CPU 控制 ZigBee 时，其协议栈需要 32KiB 的空间存储，但可以精简为 4KiB（这被认为是非常小）。

ZigBee 解决方案

一些适合在嵌入式计算机系统中使用的 ZigBee 芯片组由以下生产商制造：

- Atmel（例如 PQFN-32 封装的 AT86RF230：5mm×5mm）。
- Freescale（例如 LGA-64 封装的 MC132XX：9mm×9mm）。
- Microchip（例如 QFN-40 封装的 MRF24J40：6mm×6mm）。
- 德州仪器（例如 QLP-48 封装的 CC2420：7mm×7mm）。 465

D.10 无线 USB

无线 USB（WUSB）旨在扩大有线 USB 标准的战果：一般认为有线 USB 是用户友好和可靠的，而无线 USB 倡导者希望取得与有线 USB 类似的理念。

WUSB 是专为房间大小的空间而设计的，采用点对点 127 通道体系结构（链路的一端是一个“集线器”，为多个终端服务）。它在 3m 以内的距离内都能获得高达 480Mbps 的数据传输速率，超过 10m 的范围能获得 110Mbps 的传输速率。它使用 3GHz 左右的频段（这意味着这项技术在不少国家都不能被授权使用）。

在大多数方面 WUSB 与 USB 类似：容易使用，127 个可编址设备，相同的集线器以及发言式拓扑结构，相同的 480Mbps 最大数据传输速率，相同的计算机接口等。

无线 USB 解决方案

以下是一些 WUSB 芯片组：

- Alereon（例如 AL5100 无线收发器加上 AL5300 的 ARM 供电的基带处理器和 MAC）。
- Atmel（例如 128 引脚 TQFP 中的 AT76C503A：14mm × 14mm，其中包含一个用于基带处理的 ARM7 处理器）。
- 三星（例如在 FBGA 8mm × 8mm 封装下的 S3CR650B。有趣的是，前缀“S3”与三星 S3C2410 相同，后者我们讨论过多次，事实上，S3CR650B 也包含一个 ARM 处理器内核，专门用于 WUSB 的基带处理）。
- Wisair 的单芯片解决方案（TFBGA 封装的 WSR610：13mm × 13mm，猜猜它包含什么处理器？毫无疑问，当然也是 ARM）。

D.11 近距离通信

近距离通信（Near Field Communication，NFC）是一种最新的无线网络技术，提供专用的短程连接。NFC 是由索尼和 NXP 共同开发，并为通信距离超过 4cm 的电子设备之间提供直观、简单和安全的通信。它在 2003 年被批准为 ISO 标准。

NFC 的工作频段为 13.56MHz，数据传输速率可达到 424kbps，并与其他一些非接触式通信方式兼容，如 ISO 14443A 和 ISO 14443B（与索尼的 FeliCa 技术一同使用）。与 NFC 那样，它们也工作在 13.56MHz 频段。

NFC 接口可以工作在几种不同的模式下，不同模式决定了设备是否会产生自身的射频场，或设备能否从其他设备产生的射频场中获得能量。如果一个设备能产生自己的场，则我们称之
466 为*有源*设备，否则称之为*无源*设备。

NFC 技术主要针对移动电话应用，但也可以扩展至短距离通信（如 RFID——无线射频识别任务）。

NFC 解决方案

以下是目前的一些 NFC 芯片组：

- NXP（如 HVQFN40 封装下的 PN511 和 HVQFN40 或 TSSOP38 封装下的 PN531）。
- Sony 的 FeliCa 系列。

RedTacton 是另一种低功率技术，它使用人类皮肤做导体。因此，这是一个为人类区域网络（human area networking，HAN）服务的协议。它于 1996 年由麻省理工学院医学实验室的 Thomas Zimmerman 和 Neil Gershenfeld 共同提出。在他们工作的基础上，日本电报电话公司（NNT）进行了进一步的研究和发展，创造了 ElectAura-Net，后来成为 RedTacton。

RedTacton 可以安全地将人体体表作为一个速率高达 10Mbps 的数据传输通路。RedTacton 有如下三大主要功能：

- 通信通路可以由一个物理接触创建，触发比如说一个安装在人身上的电子传感器和嵌入式计算机之间的数据流。另一个例子是，两个配备了 RedTacton 设备的人之间可以通过简单的握手来交换数据。
- 除了人体，RedTacton 还可以利用许多材料来作为传输介质。只要材料是导电的和绝缘的，如水和其他许多液体、金属面料、部分塑料等。

- 与无线技术不同的是，即使大量人同时在会议室、礼堂、商店等地方通信，传输速度也不会恶化。这将造就一个固有的可靠的通信系统。这意味着，可以实现一个口袋中的嵌入式计算机对另一个口袋中的设备“讲话”，或查询安装在身体上的传感器、安装在鞋子上的传感器、助听器等设备。

D.12 WiBro

韩国的 WiBro（Wireless Broadband，无线宽带）是一种基于移动 WiMAX 技术（IEEE 802.16e 的 TDD OFDMA 标准）的无线宽带服务，它提高了数据传输速度，也带来了更高的复杂度。这是一个高速的服务，能够在用户行驶速度达 120km/h 的情况下提供语音、数据和视频服务。

WiBro 技术规范是 IEEE 802.16-2004、P802.16e 和 P802.16-2004 标准的一个子集。2002 年，韩国政府为 2.3GHz 的范围分配 100MHz 的频谱，允许 WiBro 为基站周围半径 1～5km 范围内（在 10MHz 的通道中）提供 20～30Mbps 的数据吞吐量。 467

三星公司提供了基于 PCMCIA 的 WiBro 接入卡，其他几个制造商的 WiBro 开发人员也正在开发类似的芯片组。

D.13 无线设备总结

表 D-1 总结了前面几节的内容。但我们在“引言”中也曾提到，这是一个迅速发展的领域。表中的信息都是作者写作本书时的最新版本，但随着技术的不断进步将迅速过时，尤其是在表的下半部分。

表 D-1 几种相对持久地用于当代嵌入式计算机系统中的无线标准及其重要参数

技术	频率	信道带宽	最大数据传输速率	一般数据传输速率	功率需求	一般通信范围
802.11a	5.0GHz	20MHz	54Mbps	25Mbps	50～1000mW	25m
802.11b	2.4GHz	25MHz	11Mbps	5.5Mbps	10～1000mW	30m
802.11g	2.4GHz	25MHz	54Mbps	25Mbps	10～1000mW	50m
802.11n	2.4/5GHz	20/40MHz	600Mbps	300Mbps	50～1000mW	50m
802.20	<3.5GHz	1.25/2.5MHz	1Mbps	—	—	3～8km
802.16	10～66GHz	20，25，28MHz	33～134Mbps	—	—	2～5km
802.16a/d	2～11GHz	1.5～22MHz	75Mbps	40Mbps	250～2500mW	10km
802.16e	2～11GHz	1.5～22MHz	75Mbps	15Mbps	250～2500mW	3km
WiBro	2.6GHz	10MHz	20～30Mbps	1～3Mbps	250～2500mW	1～5km
蓝牙	2.4GHz	1MHz	1～3Mbps	0.7～2.1Mbps	1～100mW	1～100m
GSM	0.8/0.9，1.8/1.9GHz	200kHz	9.6～19.2kbps	—	20～3000mW	100m～35km
GPRS	0.8/0.9，1.8/1.9GHz	200kHz	171kbps	40～50kbps	20～3000mW	100m～35km
ZigBee	868～868.9MHz	300/600kHz	20kpbs	—	1～1000mW	10～75m
ZigBee	902～928MHz	300/600kHz	40kpbs	—	1～1000mW	10～75m
ZigBee	2400～2483.5MHz	2MHz	250kbps	—	1～1000mW	10～75m
WUSB	3.1～10.6GHz	ultra wideband	480Mbps	—	100～300mW	10m
NFC	13.56MHz	—	424kbps	—	—	0～20cm
PAN	0.1～1MHz	400kHz	417kbps	2400bps	1.5mW	0cm
RedTacton	—	—	10Mbps	—	>100mW	0cm

D.14 应用举例

为了举例说明如何选择标准，想象我们有一个嵌入式 ARM 系统，现在我们需要将它升级为支持无线技术。

这个系统需求包括以下几点：

- 一个 200MHz 的三星 S3C2410 微处理器，支持最大 4Mbps 速率的 32 位并行总线接口和串行端口。（对于与 S3C2410 连接的众多外设，请参见 7.2 节）。
- 32MiB 的 SDRAM 和 16MiB 的并行连接的闪存。
- 功率消耗不超过 1.5W。
- 办公室周围 10m 以上的数据传输速率为 4Mib/s。
- 我们不希望购买频段。相反，ISM 频段是首选。
- 3.3V 的电源供给。

成本不是一个问题（不像大多数“现实世界”的研发），这种情况对我们来说很幸运。在 3.3V 电源方面，可以使用线性稳压器以降低电压，或使用开关稳压器（效率可接近 80%）。

所使用的频率需要在未被许可的 ISM 频段范围内。由于这涉及公共频率，频谱效率如延迟、安全和启动时间相对就不重要。可能所有的 200MIPS 微处理器都可用于支持无线通信（因为在 MIPS 上没有任何更高的要求），但一个低成本的独立单芯片解决方案肯定会是首选，因为这涉及较少的开发工作（不需要写软件协议或进行测试）。

468 ~ 469

在 4Mbps 的数据传输速率需求的基础上，我们可以排除许多表 D-1 中给出的无线技术。剩下 802.11a/b/g/n、802.16a/d/e、ElectAura-Net（一种专用网络）、WUSB 和 WiBro。WUSB 也被排除，因为它使用一个非公开的频率范围。

在 10m 通信距离需求的基础上，可以进一步从列表中排除 ElectAura-Net。而由于功率预算是 1.5W，我们应该选择 802.11a/b/g 技术，因为它们在基于距离的要求上是能耗最低的。

802.16a/d/e 和 WiBro 可能会超过给定的功率预算。802.11n 标准还没有最后确定，因此它也应该从列表中排除。选择 802.11a/b/g 的另一个好处是，它可以与 TCP/IP 兼容。

要找到一个合适的单芯片解决方案，我们可以参阅前面关于 802.11a/b/g 解决方案的介绍。然后可以参考列出设备的数据表以及通过互联网搜索其他最新方案。在查询所有的方案后，Broadcom 的 BCM4328 看起来最有优势：它是单芯片并且支持 IEEE 802.11a/b/g 标准，自带一个 CPU 以处理通信协议。BCM4328 要求 3.3V 的电源。这意味着，我们不需要升压或降压电源稳压器。BCM4328 还支持安全数字（secure digital）接口和 USB2.0——而 S3C2410 也支持安全数字接口和 USB1.1。因此当最大速率不超过 12Mbps 时，我们能够使用 USB 接口，否则使用支持超过 100Mbps 速率的安全数字接口。

D.15 小结

本附录简单介绍了如何为一个嵌入式计算机系统增加无线通信连接。同时依次介绍了大部分常用的无线协议，并通过一张表格对各种无线协议的参数进行了总结。

最后，给出了一个简单的应用例子，说明如何为三星 S3C2410 嵌入式系统选择无线通信设备。

470

编译和仿真 TinyCPU 的工具

目前针对 FPGA 的开发有许多先进的工具。主要的 FPGA 供应商都会提供自己的软件，通常发布免费的 Web 版本以供下载，而专业的芯片开发公司则会提供运行在 UNIX 和 Linux 工作站上的工具，用于工业生产中开发最尖端的项目。Mentor Graphics 公司的 ModelSim 便是这些最常见的开发工具之一。

作者建议在大型或是关键的项目设计中使用 ModelSim。但是，为了进行快速评估和轻量级的测试，我们将提出一个简单的开源方案：Icarus Verilog[㊀]，结合使用 GTKwave[㊁]波形查看工具。同时也有备选方案，特别是对于波形的查看。

E.1 准备和软件获取

该软件的最佳运行环境是安装有 Linux 操作系统的计算机，特别是 Kubuntu 或者 Ubuntu 版本的 Linux。由于一些读者可能还没有将 PC 的操作系统从 Windows 更换成 Linux，因此可以先安装 Wubi[㊂]，Wubi 可以在计算机的 C 盘里创建一个大型文件，并且在 Windows 的“启动”菜单中添加一个选项，使得他们在下一次启动计算机时可以选择运行 Kubuntu。卸载过程十分快捷。可以轻松地删除这个大型文件，并且卸载掉整个软件。Mac 操作系统的用户可以获取并运行两个版本的软件，有能力的用户甚至可以利用源代码编写出这个软件。

这里，我们假定读者已经有了一个可以工作的 Linux 发行版，或是类似的版本。Kubuntu/Ubuntu 用户现在就可以开始安装该软件的两个部分。在 Shell 命令行窗口中，键入以下内容：

```
sudo apt-get install verilog gtkwave
```


当出现提示符时，输入你的密码。如果给出警告说你不在 sudoers 的列表中，请切换成该计算机的管理员账户或是 root 账户。如果 APT 找不到该软件或是下载失败，则到网页上搜索“gdk-wave Ubuntu 软件包”或者“verilog Ubuntu 软件包”，下载 .deb 文件到计算机桌面上，然后双击鼠标并选择“install package”选项。

以上步骤都顺利完成之后，你就有了一个可以工作的 Verilog 编译器和仿真器（Icarus），以及示波器（GTKwave）。

E.2 如何编译和仿真 Verilog

我们用 8.7.6 节的 TinyCPU 栈堆作为一个例子，在程序清单 E.1 里重复列出。

程序清单 E.1 stack.v

```
1  module stack(clk, reset, load, push, pop, d, qtop, qnext);
2    parameter N = 8;
3
4    input clk, reset, load, push, pop;
```

㊀ http://www.icarus.com/eda/verilog/。

㊁ http://gtkwave.sourceforge.net/。

㊂ 从 http://wubi-installer.org 网站下载并运行 wubi 安装包，然后跟随提示选择 kubuntu 或 ubuntu 进行安装。

```
5    input [15:0] d;
6    output [15:0] qtop, qnext;
7    reg [15:0] q [0:N-1];
8
9    assign qtop = q[0];
10   assign qnext = q[1];
11
12   always @(posedge clk or negedge reset)
13     if(!reset) q[0] <= 0;
14     else if(load) q[0] <= d;
15     else if(pop) q[0] <= q[1];
16
17   integer i;
18   always @(posedge clk or negedge reset)
19     for(i=1;i< N-1;i=i+1)
20       if(!reset) q[i] <= 0;
21       else if(push) q[i] <= q[i-1];
22       else if(pop) q[i] <= q[i+1];
23
24   always @(posedge clk or negedge reset)
25     if(!reset) q[N-1] <= 0;
26     else if(push) q[N-1] <= q[N-2];
27
472  28 endmodule
```

假设这是我们在当前目录下保存的一个名为“stack.v”的文本文件，我们使用如下所示的 Shell 命令，利用 Icarus Verilog 去编译 Verilog 源代码：

```
iverilog  -o stack  stack.v
```

这条命令指示 Icarus Verilog 编译器（iverilog）对 Verilog 源程序 stack.v 进行编译，并且在当前目录下生成一个名为 stack 的可执行输出文件。

但是仅靠这个是不够的，我们必须指定程序的输入或输出，而这也是测试平台的核心。因此，我们还需要编写出一个测试平台来“使用”Verilog 模块。所幸我们已经在 8.7.6 节里编写初始代码的时候创建了一个测试平台。

当我们使用 ModelSim 进行仿真的时候，可以使用这个测试平台。但是，对于 Icarus Verilog 和其他的一些工具，我们需要仔细区分开源代码中的哪些信号是仿真中我们需要检测的，还要确定我们将要把这些信息存在何处。而后者很容易用 Verilog 中的 $dumpfile 仿真命令完成：

```
$dumpfile("stack_tb.vcd")
```

然后，前者的规范可以利用包含了一些信号的语句来制定，而这些信号是由下面 Verilog 的 $dumpvars 仿真命令产生的：

```
$dumpvars(0, stack_tb);
```

最后要指出的是，在我们所有的测试平台中，已经创建了一个恒动的时钟。在没有任何其他信息的情况下，我们的仿真将会一直持续下去（直到测试平台中的指定活动完成）。这时，我们可以使用另一条 Verilog 仿真命令来结束整个仿真任务：

```
$finish;
```

473 程序清单 E.2 列出了经过修改过的原始测试平台：

程序清单 E.2　stack_tb.v

```
1  `timescale 1ns / 1ps
2  module stack_tb;
3  reg clk, reset, load, push, pop;
4  reg [15:0] d;
5  wire [15:0] qtop;
6  wire [15:0] qnext;
7
8  stack stack0(.clk(clk), .reset(reset), .load(load), .push(push),
         .pop(pop), .d(d), .qtop(qtop), .qnext(qnext));
9
10 initial begin
11   clk=0;
12   forever
13     #50 clk = ~clk;
14 end
15
16 initial begin
17
18   $dumpfile("stack_tb.vcd");
19   $dumpvars(0,stack_tb);
20
21   reset=0; load=0; push=0; pop=0; d=0;
22   #100 reset=1; push=1; d=16'h1111;
23   #100 push=1; d=16'h2222;
24   #100 push=1; d=16'h3333;
25   #100 push=1; d=16'h4444;
26   #100 push=1; d=16'h5555;
27   #100 push=1; d=16'h6666;
28   #100 push=1; d=16'h7777;
29   #100 push=1; d=16'h8888;
30   #100 push=1; d=16'hEEEE;
31   #100 push=0; pop=1;
32   #100 pop=1;
33   #100 pop=1;
34   #100 pop=1;
35   #100 pop=1;
36   #100 pop=1;
37   #100 pop=1;
38   #100 pop=1;
39   #100 pop=0; load=1;  d=16'h1234;
40   #100 load=0; pop=1;
41   #100 $finish;
42 end
43 endmodule
```

我们将需要测试的栈模块连同测试平台一起进行编译：

```
iverilog  -o stack_tb  stack.v stack_tb.v
```

正如我们前面所看到的，这条命令之后会产生一个可执行的输出文件，这个时候名为 stack_tb。接下来，我们用 Icarus Verilog 的 vvp 命令来执行 stack_tb 的仿真：

```
vvp -stack_tb
```

474 -H 仿真完成，很快就会生成一个新的文件，文件名称就是之前在 Sdumpfile 命令中给定的；本例中为 stack_tb.vcd。这个是会改变转存（VCD）文件，这是 Verilog 标准当中的一部分，可以由一定数目的观察者打开。在我们的例子中，我们使用 GTKwave 来打开并显示它：

```
gtkwave stack_tb.vcd &
```

此时将出现如图 E-1 所示的空白的波形窗口。然后就要放大屏幕左侧“SST”窗口里的信号，点击触后出现在“stack0”标识，如图 E-2 所示。

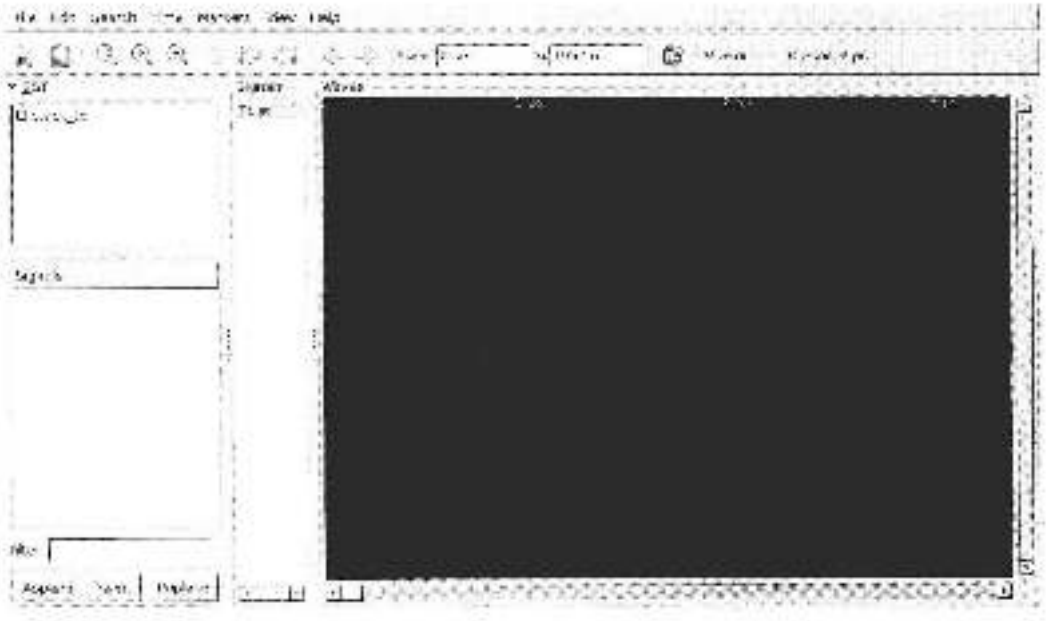


图 E-1 确认要观察的信号之前的 GTKwave 显示截图

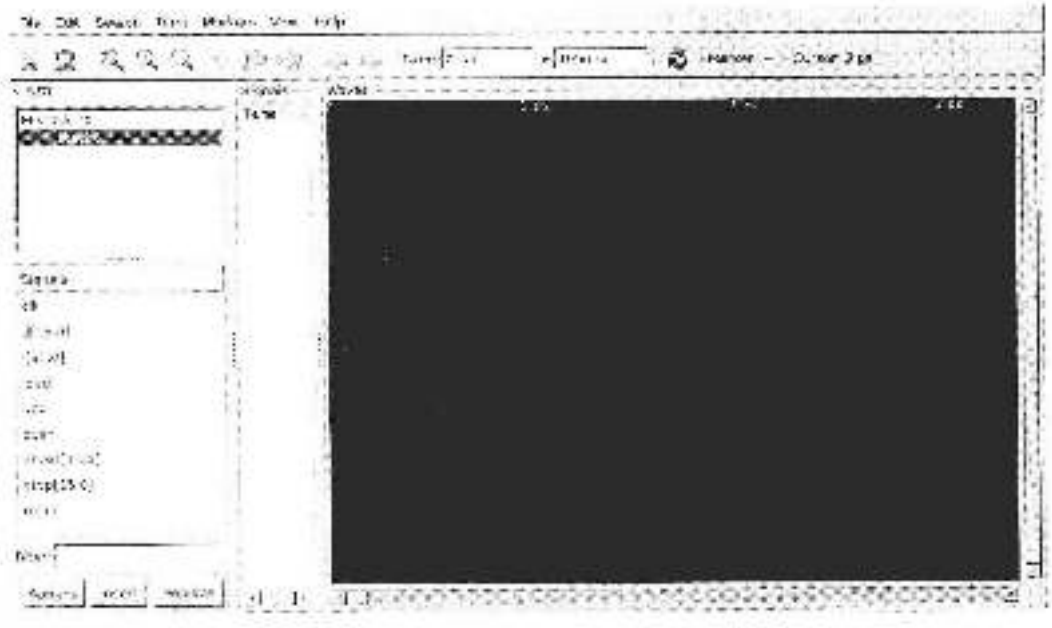


图 E-2 GTKwave 显示器列出可用信号的截图

通过选择所需的信号，并点击“Append”按钮，这些都可以添加到主显示区。通常也需要在主菜单中选择 Time->Zoom Full（或点击放大镜图标，就是类似方形镜框的符号）来把显示输出放大到最大倍数。在图 E-3 中可以看到屏幕上的 7 个主要信号，都是用前面所提到的方法放大过的。



图 E-3 GTKwave 显示器显示测试平台产生的时钟以及 VCD 文件中的 6 个控制和数据信号，仿真的精度设置为 2μs

E.3 如何查看仿真输出

为了确保测试平台上的仿真结果符合预期，最基本也是容易出错的方法就是用眼睛观察波形。在有大量仿真结果的情况下这种方法效率很低（人眼当然不适合用来专门查看波形查看工具中的大量波形）。经验表明用这种方法往往容易忽略错误。

用波形查看工具获得少量波形要好得多，但是需要使用单独的工具来做模块的通过/失败测试。而且，这样做还需要我们学习一些 Verilog 的仿真命令。在 E.2 节中我们已经见到了三个这样的命令，这里我们再给出几个更有用的命令：

命令	含义
$monitor	当信号改变时打印出来
$input	从文件中读取命令
$display	与 printf 命令等效
$stop	暂停仿真
$finish	终止进程
$time	仿真时间
$readmemb	将文件中二次进程的数列导入为存中
$readmemb	将文件中二次进程的数列导入为存中
$dumpfile	确定输出将要写入的 VCD 文件
$dumpvars	确定要指定波形的变量

475
-
476

Verilog 中的 $display 命令就像 C 语言中的 printf() 命令，输出仿真期间需要的信息（但是为 FPGA 编译代码时将忽略）。⊖

为了说明输入和输出文本数据在验证一个 Verilog 仿真模块的操作中的作用，我们回过头再看看堆栈的例子。在这种情况下，我们可以重构测试平台，使它从一个文件中读取数据而不是手
477 工向 Verilog 中输入数据项。

程序清单 E.3 显示了这个测试平台。在这种情况下，在测试平台 data 的开始处就定义了一个寄存器，这个测试平台用来存储输入的测试矢量文件。readmemh 命令此时用来将数据从一个输入文件导入到这个寄存器（这个寄存器实质上是一个存储结构）。数据文件的格式应该符合这个存储器的排列。在这种情况下，我们的数据文件，每行按从左到右的方式排列，包含了这些输入：reset，load，push，pop 和 d。因此在输入的矢量文件中每行有 5 项，如程序清单 E.4 所示。

程序清单 E.3 stack_file_tb.v

```
1  `timescale 1ns / 1ps
2  module stack_tb;
3  reg clk, reset, load, push, pop;
4  reg [15:0] d;
5  wire [15:0] qtop;
6  wire [15:0] qnext;
7  stack stack0(.clk(clk), .reset(reset), .load(load), .push(push),
       .pop(pop), .d(d), .qtop(qtop), .qnext(qnext));
8
9  reg [500:0] data [0:100];  // each line of input data has 5
       words.  We have 20 lines, 20x5=100
10
11 initial $readmemh("infile.txt", data);
12 integer i;
13
14 initial begin
15   clk=0;
16   forever
17     #50 clk = ~clk;
18 end
19
20 initial begin
21   $dumpfile("stack_tb.vcd");
22   $dumpvars(0,stack_tb);
23 end
24
25 initial begin
26    reset=0; load=0; push=0; pop=0; d=0;
27    $display("\t\t\ttime\treset\tload\tpush\tpop\td\tqtop\tqnext");
28    for(i=0;i<5*20;i=i+5) begin
29    #100
30    reset=data[i];
31    load=data[i+1];
32    push=data[i+2];
```

⊖ 注意 Verilog 2001 还有输入/输出功能，但并不是都能被 Icarus Verilog 支持。

```
33      pop=data[i+3];
34      d=data[i+4];
35      $display("\t%d\t%b\t%b\t%b\t%b\t%04h\t%04h\t%04h",$time,
           reset, load, push, pop, d, qtop, qnext);
36      end
37  #100 $finish;
38  end
39  endmodule
```

程序清单 E.4　infile.txt

```
1   0 0 0 0 0
2   1 0 1 0 1111
3   1 0 1 0 2222
4   1 0 1 0 3333
5   1 0 1 0 4444
6   1 0 1 0 5555
7   1 0 1 0 6666
8   1 0 1 0 7777
9   1 0 1 0 8888
10  1 0 1 0 EEEE
11  1 0 0 1 XXXX
12  1 0 0 1 XXXX
13  1 0 0 1 XXXX
14  1 0 0 1 XXXX
15  1 0 0 1 XXXX
16  1 0 0 1 XXXX
17  1 0 0 1 XXXX
18  1 1 0 0 1234
19  1 0 0 1 XXXX
```

478

回头再看程序清单 E.3 中的测试平台，寄存器包含了能存储 500 个元素的空间，以 5 × 100 的阵列排列，因此这可以容纳 100 个文本矢量行。然而，在读进测试矢量的主循环中，我们每次只读进了 20 行（每行包含 5 个元素）。

所以下一步我们这样编译并且仿真：

```
iverilog  -o stack_file_tb stack.v stack_file_tb.v

vvp stack_file_tb
```

这次，因为我们已经使用了 $display 命令，所以在显示器上会打印出与如下所示相似的信息：

```
VCD info: dumpfile stack_tb.vcd opened for output.
time     reset    load     push     pop      d        qtop     qnext
 100       0        0        0        0        0        0        0
 200       1        0        1        0     1111        0        0
 300       1        0        1        0     2222        0        0
 400       1        0        1        0     3333        0        0
 500       1        0        1        0     4444        0        0
 600       1        0        1        0     5555        0        0
 700       1        0        1        0     6666        0        0
 800       1        0        1        0     7777        0        0
```

```
 900      1      0      1      0     8888       0      0
1000      1      0      1      0     eeee       0      0
1100      1      0      0      1     xxxx       0      0
1200      1      0      0      1     xxxx       0      0
1300      1      0      0      1     xxxx       0      0
1400      1      0      0      1     xxxx       0      0
1500      1      0      0      1     xxxx       0      0
1600      1      0      0      1     xxxx       0      0
1700      1      0      0      1     xxxx       0      0
1800      1      1      0      0     1234       0      0
1900      1      0      0      1     xxxx    1234      0
2000      x      x      x      x     xxxx       0      0
```

479

打印出来的这些列都是测试平台测出的结果，提供了时间、文本矢量的输入信息、stack.v 模块的输出信息：qtop 和 qnext。必要的时候，可以将这些信息捕捉到一个文件中。事实上，启动仿真器，用 UNIX 语法将输出重新导入到一个文件比较好实现：

```
vvp stack_file_tb  > dump.txt
```

现在把注意力转到已经输出的测试矢量，我们首先可以发现输入参数不出意外地和文件 infile.txt中的参数一样。再看输出栏，qtop 栏显示了堆栈顶端的当前值。然而，好像有一个问题——我们可以发现 qtop 在时间值为 1800 的时候其值为 1234，但是为何 1111、2222 这些值没有被推入堆栈中呢?

最有可能存在问题的是文件 infile.txt 或者测试平台本身。我们回过头去检查原始测试平台以及仿真的输出（见图 E-3）。仔细检查波形图，很明显存在同样的问题：qtop 有同样的值 1234，且这个值接近仿真的结尾处。

眼尖的读者或许已经意识到了问题所在。回到 8.7.6 节，我们在那里首次定义了堆栈。观察这个定义了输入和输出以及栈的行为的表格，发现 PUSH 信号本身不会引起数据入栈，只会使数据向堆栈的下一级推进。要想装入数据，必须设置 PUSH 和 LOAD 命令。检查我们的测试矢量，我们可以发现这个问题——我们只有 PUSH 设置而没有 LOAD 设置。现在我们在测试矢量中修改这个问题，如程序清单 E.5 所示：

程序清单 E.5 infile.txt

```
1  0 0 0 0 0
2  1 0 1 0 1111
3  1 1 1 0 2222
4  1 1 1 0 3333
5  1 1 1 0 4444
6  1 1 1 0 5555
7  1 1 1 0 6666
8  1 1 1 0 7777
9  1 1 1 0 8888
10 1 1 1 0 EEEE
11 1 0 0 1 XXXX
12 1 0 0 1 XXXX
13 1 0 0 1 XXXX
14 1 0 0 1 XXXX
15 1 0 0 1 XXXX
16 1 0 0 1 XXXX
17 1 0 0 1 XXXX
```

```
18  1  1  0  0  1234
19  1  0  0  1  xxxx
```

480

重新进行仿真（没有必要重新编译——Verilog 本身可以处理），然后检查新打印出来的输出文本矢量：

```
VCD info: dumpfile stack_tb.vcd opened for output.
```

time	reset	load	push	pop	d	qtop	qnext
100	0	0	0	0	0	0	0
200	1	0	1	0	1111	0	0
300	1	1	1	0	2222	0	0
400	1	1	1	0	3333	2222	0
500	1	1	1	0	4444	3333	2222
600	1	1	1	0	5555	4444	3333
700	1	1	1	0	6666	5555	4444
800	1	1	1	0	7777	6666	5555
900	1	1	1	0	8888	7777	6666
1000	1	1	1	0	eeee	8888	7777
1100	1	0	0	1	xxxx	eeee	8888
1200	1	0	0	1	xxxx	8888	7777
1300	1	0	0	1	xxxx	7777	666
1400	1	0	0	1	xxxx	6666	5555
1500	1	0	0	1	xxxx	5555	4444
1600	1	0	0	1	xxxx	4444	3333
1700	1	0	0	1	xxxx	3333	2222
1800	1	1	0	0	1234	2222	2222
1900	1	0	0	1	xxxx	1234	2222
2000	x	x	x	x	xxxx	2222	2222

你可能发现这比起之前得到的仿真数据更合理。作为第二轮检查，我们再来查看在 GTKwave 中新得到的文本矢量输出波形，如图 E-4 所示。



图 E-4 与 E-3 相对应的 GTKwave 显示截图，这一次使用的是改正后的测试矢量输入文件

很明显，新的 GTKwave 显示比图 E-3 中拥挤得多。然而，注意到第一次中丢失的数据了吗？大多数读者可能都没有发现，这就是一个典型的例子，说明波形显示不总是检测代码模块是否
481 正常工作的最好的方法，无论这个模块是用 Verilog、VHDL、C 还是 Java 写的。

E.4 高级测试平台

更高级的测试平台并不使用波形显示或者文本输出：它们使用另一种工具来进行分析。作者最爱用的工具之一就是用 MATLAB 或者 Octave[⊖]建立一个工作模型，然后按如下方式来使用：

- 在模型上进行扩展测试，以此来保证它能正常工作。这就有了作为比较的基础——任何运行出来与这个模型不相符的代码都是错误的。
- 写一个程序来为这个模型提供输入数据。
- 将这些输入数据导出到一个文件中进行格式化，使它能够被 Verilog 仿真访问，作为输入测试向量。
- 运行 Verilog 仿真，将输出放入一个文件。
- 把这个输出的文件读入 MATLAB 或者 Octave。
- 把同样的测试向量放到 MATLAB 或者 Octave 的模型中运行，获取输出结果。
482 • 查看模型中的输出值的向量与 Verilog 仿真的结果是否一致。有几种方法可以实现：将两种结果绘制在一个图中，减掉两者的输出阵列并找到非零元素，计算两者的均方差等。

另一种产生测试平台的方法——在 Verilog 代码中——就是使用一个 for 循环自动产生作为待测模块输入的测试数据。以下是用这种方法在 Verilog 测试平台中使用 for 循环的代码片段：

```
reg x, y, z;
integer k;
...
...
initial begin
   {x,y,z} = 0;

   for (n=0; n<=16; n=n+1)
      #100 {x,y,z} = n;
   #200 $finish;
end
...
```

E.5 小结

本附录为编译和仿真 Verilog 源代码提供了一个完全开源的解决方案，尤其是用 Icarus Verilog 对 TinyCPU（参见第 8 章）进行的编译和仿真。讨论了用测试向量输入和输出进行编译、仿真以及调试的方法。还介绍了 GTKwave 这个针对 VCD 文件的开源的波形查看工具。

使用这里介绍的方法，对整个 TinyCPU 进行仿真并且运行良好，尽管就像之前所说的，对于
483 任何工业项目，作者更加推荐使用 ModelSim 并结合许多设备制造商的工具。

⊖ Octave 是一个开源（免费）的数学工具，与 MATLAB 几乎完全相同。可以从 http://www.gnu.org/software/octave/处下载。

TinyCPU 编译和汇编代码的工具

F.1 引言

我们在 8.9 节已经介绍过如何对 TinyCPU 进行代码编写和编程。我们给出了一个非常简短的例子，实现简单的整数减法。在这个例子中，我们手工将其汇编成机器码并插入到 TinyCPU 的 Verilog 代码中（即 ram.v 中）。这么做的目的是表明人工编译是多么的烦琐和冗长。

在 8.9.2 节，我们讨论过 Nakano 教授给出的 TinyCPU 的汇编器和编译器[⊖]，但是没有给出任何细节。

在本附录中，我们将展示整个汇编器，解释其工作原理，并通过 8.9 节的例子证明其可用性。我们也会简单地讨论 C 编译器。

F.2 汇编过程

汇编器是由一段程序实现的，程序中包括汇编语言助记符、标签、常量和其他信息。引用 8.9.1 节的简单的 TinyCPU 程序，如程序清单 F.1 所示，用来说明其语法和格式：

程序清单 F.1 subtract.asm

```
1         IN
2         PUSH cnst
3         SUB
4         OUT
5         HALT
6   cnst: 3
```


汇编器的任务是生成机器码的输出。机器码是由十六进制指令所组成的向量，代表 CPU 中的源程序。程序清单 F.2 中所展示的是相同例子的机器码输出：

程序清单 F.2 subtract.hex

```
1  D000
2  2005
3  F001
4  E000
5  0000
6  0003
```

如果我们将这个程序清单和 TinyCPU 的指令集（见表 8-1 和表 8-2）作比较的话，就能很容易看出十六进制代码的含义：D000 代表 IN，0000 代表 HALT 等。在汇编语言输入和机器码输出之间是行与行对应的。除了扩展的宏[⊜]之外，这对所有的汇编器都适用。

⊖ 汇编器和编译器均可从网络上获得，网址：http://www.cs.hiroshima-u.ac.jp/~nakano/wiki/。另外在本附录中，我们也会给出汇编器全部的源代码。

⊜ 宏（macro）是一段代码。由于这些代码会被重复很多次，所以可以只预定义一次，并给定标识符，然后根据需要使用标识符替换这些代码。宏在很多情况下也可以被参数化。

值得一提的是第 2 行和第 6 行。在第 2 行中，汇编语言助记符为 PUSH，其对应的机器码应为 2000 + A，这里被汇编为 2005。这反映出压栈的这一项被存储在内存地址为 5 的位置，而这个位置是在第 6 行定义的。第 6 行本身相当于常量 3。如果 CPU 把这一行作为指令来读取就会出错，因为它并不代表一条指令。然而 0000 代表 HALT 指令，在这种情况下 CPU 执行时不会超过第 5 行。

F.3 汇编器

汇编器本身是用 Perl 语言写的。Perl 是一种计算机语言，它特别适合文本处理操作，而汇编过程实际上就是文本处理操作。在现代 Linux 计算机中，Perl 通常是被默认安装进去的，其他的操作系统也可以免费获得 Perl。[⊖]

程序清单 F.3 中给出了汇编器程序 tinyasm.perl，它包含了正常工作的汇编器的全部源代码。
485 正如所看到的，在程序一开始就给出了每条指令对应的十六进制的值。

程序清单 F.3 tinyasm.perl

```
1  #!/usr/bin/perl -W
2
3  %MCODE = (HALT=>0x0000,
4            PUSHI=>0x1000,
5            PUSH=>0x2000,
6            POP=>0x3000,
7            JMP=>0x4000,
8            JZ=>0x5000,
9            JNZ=>0x6000,
10           IN=>0xD000,
11           OUT=>0xE000,
12           ADD=>0xF000,
13           SUB=>0xF001,
14           MUL=>0xF002,
15           SHL=>0xF003,
16           SHR=>0xF004,
17           BAND=>0xF005,
18           BOR=>0xF006,
19           BXOR=>0xF007,
20           AND=>0xF008,
21           OR=>0xF009,
22           EQ=>0xF00A,
23           NE=>0xF00B,
24           GE=>0xF00c,
25           LE=>0xF00D,
26           GT=>0xF00E,
27           LT=>0xF00F,
28           NEG=>0xF010,
29           BNOT=>0xF011,
30           NOT=>0xF012);
31
32 $addr=0;
33 while(<>){
```

⊖ http://www.perl.org。

```
34      push(@source,$_);
35      if(/(\w+):/){
36          $label{$1}=$addr;
37          s/\w+://;
38      }
39      if(/-?\d+|[A-Z]+/){
40          $addr++;
41      }
42  }
43
44  print "***_LABEL_LIST_***\n";
45  foreach $l (sort(keys(%label))){
46      printf "%-8s%03X\n",$l,$label{$l};
47  }
48
49  $addr=0;
50  print "\n***_MACHINE_PROGRAM_***\n";
51  foreach (@source){
52      $line = $_;
53      s/\w+://;
54      if(/PUSHI\s+(-?\d+)/){
55          printf
                "%03X:%04X\t$line",$addr++,$MCODE{PUSHI}+($1&0xfff);
56      } elsif(/(PUSH|POP|JMP|JZ|JNZ)\s+(\w+)/){
57          printf "%03X:%04X\t$line",$addr++,$MCODE{$1}+$label{$2};
58      }elsif(/(-?\d+)/){
59          printf "%03X:%04X\t$line",$addr++,$1&0xffff;
60      } elsif(/([A-Z]+)/){
61          printf "%03X:%04X\t$line",$addr++,$MCODE{$1};
62      } else {
63          print "\t\t$line";
64      }
65  }
```

在汇编器中有两个主要的循环。第一重循环在整个程序执行过程中都在运行，作用是寻找任何以冒号结尾的文本项（第 35 行），很可能指的是一个标签。标签的名称存储在一个叫做“% label”的列表中，扫描结束之后立即输出这些标签（第 44 ~ 47 行）。

程序中的第二重循环在汇编器代码执行过程中运行，作用是读取助记符和其操作数（如果要读取的指令有操作数的话）。助记符直接被转换成其映射的十六进制代码；对于那些有操作数的助记符，其操作数的值也会加到十六进制数中。

为了将输出部分从汇编器中分离出来，我们又写了一个程序，并将其恰当地格式化，使得输出部分能够被直接插入到 ram.v 的 Verilog 源代码之中。程序清单 F.4 中给出了这个程序，名为 mac2mem.perl。

486 ~ 487

程序清单 F.4 mac2mem.perl

```
1   #!/usr/bin/perl
2
3   while(<>){
4       if(/([0-9A-F]+):([0-9A-F]+)\s*(.*)/){
```

```
5       print "mem[12'h$1] = 16'h$2;\t\/\/\t$3\n";
6       } elsif(/\s+(\w+:)/){
7       print "\t\t\t\t\/\/$1\n";
8       }
9   }
```

在下一节中我们会介绍怎样使用这些程序。

F.4 汇编程序实例

让我们再次使用程序清单 F.1 中的减法代码的例子。假设这个例子存放在名为“subtract.asm”的文本文件中。假设计算机系统中已经安装了 Perl，我们可以通过以下命令执行 tinyasm.perl 来汇编这段源代码：

```
perl  tinyasm.perl  subtract.asm
```

这段程序的输出结果已经复制到程序清单 F.5 中。程序清单 F.5 给出了标签列表（本例中只有一个）和助记符与十六进制代码之间一对一的匹配结果。

程序清单 F.5 subtract.asm 执行 tinyasm.perl 的输出

```
1   *** LABEL LIST ***
2   cnst    005
3
4   *** MACHINE PROGRAM ***
5   000:D000 IN
6   001:2005 PUSH cnst
7   002:F001 SUB
8   003:E000 OUT
9   004:0000 HALT
10  005:0003 cnst: 3
```

通常情况下，我们会将输出放到一个文件当中：

```
perl  tinyasm.perl  subtract.asm > subtract.out
```

下一步就是使用 mac2mem.perl 程序将这个输出的格式标准化，从而将其插入到 ram.v 中。我们使用下面的命令来处理保存的这个文件：

```
perl  mac2mem.perl  subtract.out > subtract.out.v
```

输出结果按照 Verilog 源代码的语法格式进行了标准化（将这个格式与程序清单 8.6 中 ram.v 的格式化程序进行对比），如程序清单 F.6 所示。这个可以直接插入到 Verilog 源代码中，然后如 488 8.9.1 节的最后部分所描述的那样，将整个设计程序进行编译和仿真。

程序清单 F.6 减法代码执行 mac2mem.perl 之后的输出

```
1   mem[12'h000] = 16'hD000; // IN
2   mem[12'h001] = 16'h2005; // PUSH cnst
3   mem[12'h002] = 16'hF001; // SUB
4   mem[12'h003] = 16'hE000; // OUT
5   mem[12'h004] = 16'h0000; // HALT
6   mem[12'h005] = 16'h0003; // cnst: 3
```

F.5　编译器

TinyCPU 的发明者，Nakano 教授已经设计了一个编译器。[⊖]事实上，根据功能层次的不同，可以分为两个编译器。

主 TinyCPU 编译器叫做 tinyc，是用标准的编译器生成工具 flex 和 bison 写成的，有兴趣的读者可以参考 Nakano 教授的 wiki 页面以获取编译器执行的细节。我们在这里不转载编译器的代码，但是可以从上面提到的 wiki 页面上进行下载（例如 tinyc.l 和 tinyc.y，在它们各自由 flex 和 bison 处理过后，生成一个 C 语言源。这个 C 语言源可以被编译成可执行的文件，叫做 tinyc，而这个 tinyc 实际上就是编译器）。实际上 tinyc 可以支持一个 C 语言子集的编译。C 语言的子集遵循 C 语言的语法结构，但是可以使用几个简化和内置的操作。程序清单 F.7 中给出了一个 C 语言源代码的例子，它能够执行与之前例子相同的减法运算。

程序清单 F.7　subtract.c

```
1  out(in-cnst);
2  halt;
3  int cnst=3;
```

我们注意到尽管这个子集与 C 语言很相似，但是显而易见还是有几点不同：内置函数 out() 设置了输出缓冲区，内置变量 in 读取输入端口的数据，并且还使用了 halt 指令。另外，不像 C 语言在程序开始处声明变量，这个子集是在程序的结尾处声明变量——如果我们将变量的声明放在程序的开头，那么这个变量将会存储在内存中的第一个位置，一旦 TinyCPU 开始运行，它就会从读取这个常量开始执行，并将其作为指令试着去执行。

除了这些不同，这个子集和 C 程序很相似，而且与写汇编器相比较，这个编译器的代码写起来要更容易。尤其是 tinyc 编译器的强大之处在于其能够明确地阐述基于栈的方程式（这一点 489
在例子中并没有体现）。实际上，在将复杂的方程式转换成 TinyCPU 需要的逆波兰表达式的时候，这个编译器完全有能力实现。

程序清单 F.8 中给出了在 TinyCPU 下编译 subtract.c 源代码后的输出。

程序清单 F.8　subtract.out

```
1          IN
2          PUSH cnst
3          SUB
4          OUT
5          HALT
6   cnst:  3
```

我们没有办法将这个输出与程序清单 F.1 中所给出的汇编代码区分开来，因为这个输出实际上就是原来的汇编代码！然后我们通过 Perl 汇编器 tinyasm.perl 用通常的方法创建机器码，并且用 mac2mem.perl 来将其按照 Verilog 的语法格式进行标准化。

F.6　小结

本附录首先手工汇编了一个简短的 TinyCPU 程序。由于这个过程非常烦琐并且需要不断重复，于是我们引出了汇编器。汇编器几乎能将汇编助记符一对一地转换成十六进制机器码，另外

⊖ 可以参考 Nakano 教授的 wiki 页面 http://www.cs.hiroshima-u.ac.jp/~nakano/wiki/以获取更多细节。

还能识别出标签的位置。

尽管汇编器相对于手工汇编过程来说已经是一个很大的进步，但是高级语言的编译器会提供进一步的改进。于是我们引入了 TinyCPU 编译器。为了使创建 TinyCPU 程序的过程变得容易些，特别是那些涉及数学计算的过程，TinyCPU 的编译器使用 C 语言的语法格式，并对其进行了一些补充和删减。

这些工具，尤其是编译器，可能不是最终的解决方案。对于汇编器和编译器来说，都还有进行改进的余地，我们鼓励有兴趣的读者对其进行改写和扩展。最重要的是，本书的作者想要再次传达其在第 8 章结尾处的信息：鼓励读者运用已有的知识去增强 TinyCPU（以及它的汇编器、编
490 译器）的功能，并且可以进一步设计他们自己的解决方案。

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

6502，10，78，79，99，127，291
8086，*See Intel* 8086

A

Absolute addressing（绝对地址），188
Acorn（Acorn 公司），5，8，78，79，112
Actel
  ARM core，378
  AX1000 FPGA，428
Adder 加法器
  ripple carry（行波进位），30，130
Address（地址）
  bus architecture（总线结构），206
  handling hardware（处理硬件），205-206
ADSP2181，17，81，82，90，94，130，204-205
Advanced graphics port（高级图形接口），258
Advanced interrupt
  controller（高级中断控制器），279
Advanced microcontroller bus
  architecture（高级微控制总线体系），252，258
Advanced technology
    attachment（高级技术附件），258
  parallel（PATA）（并行），258，259
  serial（串行），334
  serial（SATA）（串行），258，259
AGP，*See Advanced graphics port*
AHB，*See ARM host bus*
Altera（美国芯片公司），362
  Nios Ⅱ，377
  Quartus-Ⅱ，408
ALU，*See Arithmetic logic unit*
AMBA，*See Advanced*
  microcontroller bus
  architecture
AMD（超微半导体公司），11，112，161，164
  3DNow，159，163，164
  Phenom，293，295
Amdahl's law（Amdahl 定律），237
AMULET，436，437，438
Analog Devices（美国模拟器件公司），41
  ADSP21xx，67，80，81，82，
  84，123，124，203，205，207，
  208，423
Analytical difference machine（差分机），
  1，2，4
ANSI C（标准 C），99
Apple（苹果公司），112，265，307
  iMac，9，10
  iPhone，10，13
  iPod，104
  Newton，34
Application specific integrated
  circuit（专用集成电路），351
Architecture（体系结构）
  bit serial（位串行），419
  dual bus（双总线），127-129
  electro-optical（光电的），442
  load store（读数取数），69，79，86，194
  multiple bus（多总线），121-130，202
  regular（常规），69
  related to IPC（与 IPC 相连），199-200
  single bus（单总线），129-130
architecture
  von Neumann（冯·诺依曼体系结构），17，143
Arithmetic（算术）
  binary addition（二进制加法），29-30
  binary subtraction（二进制减法），30-33
  logical（逻辑），18，19，23，107，130
  reverse Polish notation（逆波兰表示法），96-98
  stack（堆栈），96，97
Arithmetic logic unit（算术逻辑单元），19，29，
  67，72，130-132，244
ARM，10，11，13，22，34，67，
    77-80，83，86，90，99，106，

110, 112, 122, 127, 143, 165, 207, 252, 254, 610, 34
address handling（地址处理）, 208
ARM (*Cont.*)
ARM 7, 21, 34, 82, 84, 130, 140, 275
ARM 9, 83, 298
ARM7TDMI, 83
ARM946, 234, 235
branching（分支）, 86, 187-188
condition codes（条件码）, 87
conditional instructions（条件指令）, 110, 183-185
Cortex, 83, 235, 378
desktop computer（台式机、桌面计算机）, 291
dual core（双核）, 234, 235
FIQ（快速中断请求）, 275, 276, 278
floating point unit（浮点单元）, 159
FPA10, 159, 165
immediate constants（中间常数）, 89, 90
indirect addressing（间接寻址）, 94
instruction format（指令格式）, 81-98, 121
interrupt response（中断响应）, 273
interrupt timing（中断时序）, 275-276
IRQ（中断请求）, 275-276, 278
Jazelle, 165
JTAG scanchain（JTAG 扫描链）, 339
Linux, 326
MOV instruction（MOV 指令）, 89-90
NEON, 165
on FPGA（现场可编程门阵列）, 378
registers（寄存器）, 205
S-flag（S 标记）, 85, 181, 182, 190
shadow registers（影子寄存器）, 271
Thumb（16 位）, 178
Thumb mode（16 位模式）, 84, 90
vector table（向量表）, 274
VFP, 166
ARM host bus（ARM 主线）, 252, 258
ASCII, 110
ASIC, *See Application specific integrated circuit*
Asymmetrical multi-processing（异步多核处理）, 237
Asynchronous computer（异步计算机）, 74
Asynchronous processors（异步处理器）, 434-438
ATA, *See Advanced technology attachment*
Atanasoff-Berry machine（Atanasoff-Berry 机器）, 4
Atmel, 258
Atom（原子）, 165

B

Babbage, Charles, 1, 4
Baby, *See SSEM*
Barcelona supercomputer（巴塞罗那超级计算机）, 11, 426
BASIC, 112
Basic blocks（基础模块）, 228-229
Basic input/output stream（基础输入/输出流）, 18
BBC, *See British Broadcasting Corporation* 78
BCD, *See Number, binary coded decimal*
Bell Labs（贝尔实验室）, 4
Berkeley University（伯克利大学）, 78
Bill of materials（材料清单）, 352-353
BIOS, *See Basic input/output stream*
BIST, *See Built in self test*
Bit-serial（比特-串）
addition（加法）, 420-421
architecture（结构）, 419-420
arithmetic（算术的）, 420-422
logic and processing（逻辑和处理）, 422
subtraction（减法）, 421
word size（字长）, 422
Blanket, electric, 66
Bletchley Park, 3
Bloat（膨胀）, 141, 236
Bluetooth（蓝牙）, 462-463
BODMAS, 96
Bogomips, 111, 236
Bootloader（引导加载程序）, 277, 326
Branch（分支）
conditional（条件性）, 183-185, 211
delayed（延迟）, 189
global prediction（全局预测）, 218-221
global predictor（全局预测器）, 185

gselect predictor（G 选择预测器），221-222
gshare predictor（G 共享预测器），222-223
hybrid predictor（混合预测器），223-225
instruction（指令），86，88
local prediction（本地预测），216-218
prediction（预测），209-212
prediction algorithms（预测算法），212-225
prediction counter（预测计数），215-216
probabilistic（概率），186
speculation（推测），185，211
target buffer（目标缓冲），226-228
to relative address（相对地址），187-188，210
British Broadcasting（英国广播公司）
Corporation，79
British Standards Institute（英国标准研究所），112
Brownout（掉电），340，346-348
Built in self test（电压降低），334-337
Bus arbitration（总线仲裁），73，261
Busch，Adrian，130

C

Cache area trade-offs（Cache 方案折中），454-455
coherency（连贯性），155-157
design tools（设计工具），452
direct（直接的），144-145
efficiency（效率），154
full associative（全连接），147-148
layering（层次），144
MESI protocol（MESI 协议），155-156
performance of（运行），153-154
replacement algorithms（替换算法），149-153
set associative（指令集联合），145-147
tag（标签），145
worked example（应用示例），146，147，151，152
Cache memory（缓存），68，104，111，121，143-144
Cacti，452-454
Cambridge University（剑桥大学），4，75
CAN，*See Controller area network*
Canonical signed digit（规范的带符号数），441
Carry（进位）
look-ahead（超前），30
propagation（传递），30
propagation example（传递示例），31
CDC6000，7
Cell processor（单元处理器），238，239，240
Churchill，William（人名），3
CISC，*See Complex instruction set computer*
Clock（时钟）
asynchronous（异步），434
cycle（周期），110
delay locked loop（延迟锁相环），301
domain（域），434
double edged（双边），110
generation of（生成），301
oscillator（振荡器），301
phase locked loop（锁相环），301
solutions（解决方法），305
speed（速度），110
synchronous logic（同步逻辑），300
system（系统），294-295
Cloud computing（云计算），426
Cluster computers（机群），240，425
Co-processor（协同处理器），157-158，165-166
Co-simulation（协同仿真），376
Co-synthesis（协同综合），376
Colossus（巨人），3，4，6
Commercial-off-the-shelf（现有的商用），428
Common mode noise（公用模式噪点），263
Communications hardware（通信硬件），203
Compiler（编译器）
error trapping（异常陷入），142
handling of stored data（存储数据控制），105
loop handing（循环控制），204
optimizations（优化），105
support for branch prediction（分支预测的支持），185
support for VLIW（VLIW 的支持），425
Complex instruction set computer（复杂指令集计算机），20，76-79，80，93，95，166，193，199，371
Computer design（计算机设计），373-377
Computer generation（计算机发展阶段），5-10
fifth（第五代），9-10
first（第一代），6
fourth（第四代），8-9，173
second（第二代），7
third（第三代），7-8

Computer system bus（计算机系统总线），259
Condition（条件）
  codes（代码），87
  flags（标记），182
  simple flags（简单标记），380
Control（控制）
  self-timed（时间自控），72，73-74
  distributed（分布式），72
  of a pipeline（流水线），175
  of asynchronous machine（异步机器），437
  simplified（简单化），72
Control program for microcomputers（微型计算机的控制程序），8
Control unit（控制单元），70-75
Controller area network（控制领域网络 CAN），258，428，429
COTS，*See Commercial-off-the-shelf*
CPI，*See Cycles per instruction*
CP/M，*See Control program for microcomputers*
Cryptography（密码学），202
CSD，*See Canonical signed digit*
Cycles per instruction（周期每指令），111，198-201
Cyrix，161

D

DAG，*See Data address generator*
Dallas Semiconductor（达拉斯半导体），258
Data（数据）
  compression（压缩），202
  format and representation（格式和表达），99-103
  handling（处理），98-109
  stream（流），16-17，341
Data address generator（数据地址生成器），205，206
Data dependency（数据相关），179-180，196，200
Data link layer（数据连接层），450-451
DDC，*See Display data channel*
Debug（调试），295
  using serial port（使用串口），335-336
DEC，7，112，340
Alpha 21264，225
StrongARM，*See StrongARM*
Design ownership（设计归属），373
Design partitioning（设计划分），353，375，376
Dhrystone，benchmark，（Dhrystone 标准检测程序）112-113
Differential signalling（差分信号），262-264
Digital filter（数字滤波器），38，202
Digital signal processor（数字信号处理），110，112，123，126，140
Dinero（Dinero 软件工具），452，453-455
Direct memory access（直接存储器访问），104，254-255，261
Disk operating system（磁盘操作系统），8，162，346
Display data channel（显示数据通道），264
Distributed computing（分布式计算），444-445
Division（除法），41-43
DMA，*See Direct memory access*
DOS，*See Disk operating system*
DRAM（动态 RAM），76，114，144，258，316，317-323
DSP，*See Digital signal processor*
Dual core processor（双核处理器），234，271

E

EDAC，*See Error detection and correction*
EDSAC，4
EDVAC，4
EEPROM，67，310
EIA232 interface（EIA232 接口），264，265，335，451
EIA422 interface（EIA422 接口），264
EIA485 interface（EIA485 接口），264
EISA，*See Extended industry standard architecture*
Electromagnetic interference（电磁干扰），283，305
Electronically erasable
  programmable read only
  memory（电可擦除可编程 ROM），*See EEPROM*
Embedded future（嵌入式未来），66
Embedded designs（嵌入式设计），115
EMI，*See Electromagnetic interference*
Endian（段）
  big（大端），20-21
  little（小端），20-21
  switching（转换），179
  worked example（应用示例），20-23

ENIAC, 3, 4, 6
Enigma code (恩尼格玛编码), 3, 6
EPIC, *See Explicitly parallel instruction computing*
EPROM, 69, 309-310
Erasable programmable read only memory (可擦写可编程 ROM), *See EPROM*
ERC32, 252, 343
Error detection and correction (错误检测与纠正), 340-345
Ethernet (以太网), 254
  interface (接口), 266
  memory mapped driver (内存映射驱动器), 329
  processing (处理), 235
  service layers (服务层), 449
European Space Agency (欧洲宇航局), 343
Execution (执行)
  out of order (无序), 180, 196, 228, 240, 246, 293
Explicitly parallel instruction
  computing (显示并行指令计算), 199, 422
Extended industry standard
  architecture (扩展的工业标准结构), 258, 259

F

FDIV bug (FDIV 错误), 332
Ferranti (Ferranti 公司), 112
  Mark 1 (Ferranti Mark 1 计算机), 5
Field programmable gate array (现场可编程门阵列), 166, 237, 247, 325, 340, 346, 351, 355, 356, 357, 362, 363, 369, 370, 371, 372, 374, 375, 376, 379, 380, 382, 396, 408, 427, 428, 429, 430, 471
Finite impulse response filter (有限脉冲响应滤波器), 125
Finite state machine (有限状态机), 70
FIR, *See Finite impulse response filter*
Firewire (一种串行标准), 265
Flash memory (闪存), 67, 265, 310, 311, 312, 314, 326, 339
Floating point (浮点), 46-54, 106
  data types (数据类型), 159
  emulation (仿真), 108, 159-161
  hardware (硬件), 202
  power consumption (功耗), 160
  processing (处理), 54-60, 108
  unit (单元), 19, 54, 82, 108, 121, 157, 158-161, 162, 163, 195
Flowers, Tommy (人名), 3
Flynn
  classification (分类), 16-17
  Michael (人名), 15, 16-17, 230
  MIMD (多指令流多数据流), 16, 17, 230, 231-235, 271
  MISD (多指令流单数据流), 16, 17, 230
  SIMD (单指令流多数据流), 16, 17, 18, 161, 164, 165, 230, 233, 293
  SISD (单指令流单数据流), 16-17, 230, 231-235
FORTRAN, 109
Forwarding (转发)
  fetch-fetch (取指-取指), 191, 192
  store-store (写回-写回), 191, 192
FPGA, *See Field programmable gate array*
FPU, *See Floating point unit*
Fragmentation (碎片)
  external (外部的), 138-139
  internal (内部的), 138
Freescale (飞思卡尔), 80
FSM, *See Finite state machine*
Full adder (全加器), 29
Furber
  Steve (人名), 159

G

GEC Plessey (GEC Plessey 公司), 258
GFLOPS (每秒十亿次浮点运算), 111
Glue logic (黏合逻辑), 372-373
Google, 10, 83, 230, 435
gprof (一种软件工具), 115
GPRS (通用分组无线服务), 463, 464
Graphics processing (图形处理), 202
Grid computing (网格计算), 426
GSM, 34, 463
GTKwave (工具), 414, 471, 472, 476, 481
Guard bit (保护位), 59

H

Half adder (半加器), 29

Hamming code（汉明码），341，342
Hardware acceleration（硬件加速），201-209
Hardware software co-design（软硬件协同设计），373-377
Harvard architecture（哈佛结构），17，125，126，143
Hazard
  avoidance in asynchronous machine（异步的避免）（机器），437-438
  data（数据），179-180，196
  pipeline remedies for（流水线补偿），190
  read after write（写后读），190，196
  structural（结构的），196
  write after read（读后写），180，190，196
  write after write（写后写），180，181，190，196
Heterogeneous architecture（异构体系结构），237
High level language（高级语言），81，90，369，425
Homogeneous architecture（同构体系结构），237
Huffman coding（霍夫曼编码），90，91，92
Hyperblocks（特级代码段），228
Hypercube（超立方体），434

I

I/O pins（输入/输出引脚）
  configuration（配置），297-298
  multiplexing（多路复用），296
IA-64 architecture（IA-64 体系结构），423
IBM，5，78，79，112，238，258，259，340，435
  Cell processor（单元处理器），*See Cell processor*
  PC，79，346
  power architecture（Power 架构），238，239
  RS6000，5
  System/360，8，75，240，246
Icarus Verilog（软件工具），471，473，474
ICE，*See In-circuit emulator*
IDE，*See Integrated drive electronics*
IEEE 802.11n，460-461
IEEE 802.16，461
IEEE1149 JTAG，295，296，337，353，408
IEEE1284 interface，264，265
IEEE754，19，46-47
  arithmetic（算术的），55-56，57，58
  denormalised mode（非规格化模式），49-50，52-53
  division（除法），56
  double precision（双精度），159
  extended intermediate format（扩展中间格式），56，57-60，159
  in industry（在工厂），158
  infinity（无穷大），50，51
  modes（模式），47-51
  multiplication（乘法），56，108
  NaN（不是一个数），50，51
  normalised mode（规格化模式），48-49，51-52
  number range（数字范围），51-54
  on fixed point CPU（定点 CPU），108
  processing（处理），54-60
  rounding（舍入），60
  single precision（单精度），53-54
  standard（标准），159
  worked example（应用实例），48-49，50，54，57-58
  zero（零），50，51
IEEE802.11 a，b and g，460，470
IIC，*See Inter-IC communications*
IIR，*See Infinite impulse response filter*
Immediate constants（中间常数），88-90
In-circuit emulator（内置仿真器），337
Indirect addressing（间接寻址），94
Industry standard architecture（工业标准结构），258，259
Infinite impulse response filter（无线脉冲响应滤波器），125
Information hiding（信息隐藏），363，364
Instruction（指令）
  application specific（专用），166
  condition setting bit（条件设置位），82，85
  custom（定制），202
  decode（编码），84-90
  fetch（获取），84-90
  format（格式），80
  handling（控制），81-98
  level parallelism（级并行），229，230，422
  microcode（微指令），75-77
  set（集），81-84，95

set regularity（集规整），193
stream（流），16
translation（翻译），76
Instructions per cycle（每周期指令数），198-201，236
Integrated drive electronics（IDE 接口），69，258
Integration（整合），375
Intel，10，11，78，79，112，159，161，164，165，313，423，439
4004，5
8086，1，75，260，291，428
8088，162，260
80386，157
80387，157
80486，143，158，332
Core，237，271
IXP425，254，277
Pentium，161-163，423
Pentium Pro，143
SA1110，428
StrongARM，*See StrongARM*
XScale，143，277
Inter-IC communications（IC 间通信），258，292
Interrupt（中断）
advanced handlers（高级处理），278
and real time（实时性），267
event（事件），272
flag（标志），272
handlers and memory management（控制和内存管理），141
handling（控制），271-280
importance of（重要性），271-272
queue（队列），273
redirection（重定位），276-278
service routine（服务例程），209，268，277
sharing（共享），278-279
software（软件），279-280
Interrupt vector（中断向量），273
IPC，*See Instructions per cycle*
ISA，*See Industry standard architecture*

J

Java（高级语言），109，165，173
JTAG，*See IEEE1149 JTAG*
JTAG for booting a CPU（JTAG 引导 CPU），339

K

Kernel（内核），142，167，327，328，357

L

Lattice
Mico32（Lattice Mico32 处理器），378
Linux，83，142，164，200，329，435，471
Beowulf，240
determination of MIPS（MIPS 的确定），111
embedded（嵌入的），142，326，356，377，378
uCLinux，325
Load store architecture（存取体系结构），194
Loosely coupled tasks（松耦合任务），425
Low voltage differential signalling（低电压差分信号），259，262-264
LVDS，*See Low voltage differential signalling*

M

MAC，*See Multiply accumulate unit*
Machine parallelism（机器并行），231
Manchester University（曼彻斯特大学），4，5，132
Marconi（马可尼式无线电报），258
MareNostrum，11
Massachusetts Institute of Technology（麻省理工学院），4，5
MCA（微通道体系结构），258，259
MCM，*See Multi-chip module*
Media independent interface（多媒体独立接口），266
Memory（存储器）
access（访问），125
access in C（基于 C 语言访问），331
background（背景），307
burst mode（突发模式），110
cycle（周期），126
DRAM（动态 RAM），316，317-319
DRAM addressing（DRAM 寻址），319-323
DRAM refresh（DRAM 刷新），317，319
DRAM structure（DRAM 结构），320
EDO DRAM，323
EEPROM（电可擦除可编程只读存储器），362
EPROM（可擦除可编程只读存储器），309，

339, 362
flash（闪存）, 311, 351
flash blocks（闪存模块）, 314
flash memory control（闪存控制）, 313
for an FPGA core（对于 FPGA 核心）, 382
fragmentation（碎片）, 138
in embedded systems（嵌入式系统）, 325-332
map of ARM 9（ARM9 的映射）, 326
map of MSP430（MSP430 的映射）, 330
mapped registers（映射寄存器）, 329
NAND flash（NAND 闪存）, 311
NOR flash（NOR 闪存）, 311
on-chip（片上）, 114, 126
overlays（覆盖）, 323-325
pages（分页）, 323-325
parity checking（奇偶校验）, 340, 341
pin swapping of（引脚转换）, 358-359
PROM（可编程只读存储器）, 309
protection（保护）, 140-142
RAM, 314-323
remapping（重映射）, 277
ROM, 308-314
SDRAM, 323, 359
serial flash（串行闪存）, 311
SRAM, 316, 351, 359, 362
stack（堆栈）, 138
VRAM, 323
Memory management unit（存储管理模块）, 19, 68, 104, 121, 292, 323-324
address translation cache（地址转换高速缓冲）, 140
advanced designs（高级设计）, 139-140
operation（操作）, 133, 135-137
rationale（原理）, 133
translation look-aside buffer（转换后援缓冲器）, 140
worked example（应用示例）, 137
Mesh（网格）, 433-434
MESI
in shared memory system（共享内存系统）, 433
protocol（协议）, 155
worked example（应用示例）, 157
MFLOPS（每秒百万次浮点运算）, 111
Micro channel architecture（微通道体系结构）, *See MCA*
Microcode（微指令）, 75-77
Microprogramming（微编程）, 75
MIMO, 460
MIPS, 67, 78, 111, 114, 140, 189, 341
Mitel, 258
MMC, *See Multimedia card*
MMU, *See Memory management unit*
MMX, *See Multimedia extensions*
ModelSim（工具软件）, 409, 471
Moore's law（摩尔定律）, 1, 77, 235
Motorola, 79
68000, 20, 67, 80, 205
Coldfire, 80
MP3, 66, 115, 233
MS-DOS, 346
Multi-chip module（多芯片模块）, 237
Multi-core（多核）, 237
Multimedia card（多媒体卡）, 265, 292
Multimedia extensions（多媒体扩展）, 17, 82, 158, 159, 161-165, 230
Multiple-valued logic（乘法逻辑）, 438-439
Multiplication（乘法）, 34-41
Booth's method（Booth 算法）, 34, 38-41
on a small machine（小型机）, 106
partial products（部分积）, 35-38, 440
repeated addition（加法迭代）, 34
Robertson's method（Robertson 算法）, 34, 38
shift and add（移位加）, 38
Multiplication, by repeated addition（乘法，多次加法）, 34
Multiply accumulate unit（乘累加单元）, 123, 195, 202

N

NASA
computers（计算机）, 340
space shuttle（太空火箭）, 8, 341
Near field communication（近距离通信）, 466-467
Network layer（网络层）, 451
Newton, Isaac, 14
Number（数字）
alternative formats（替代格式）, 438-442
binary coded decimal（BCD 编码）, 26

complex（复数），109
conversion examples（转换例子），25-26
excess-n（移码），24，26
（m. n）format（m. n 格式），26
format（格式），23-28
fractional（定点的），27
fractional arithmetic（定点运算），44
fractional examples（定点例子），27，44
fractional multiply（定点乘法），45
fractional notation（定点表示法），26-27
negative two's complement（负数的补码），24，25
one's complement（反码），24
Q-format（Q-格式），26，43，161
sign extension（符号扩展），27-28
signed digit representation（符号位的表达），439-442
two's complement（补码），24
unsigned binary（无符号二进制），23-24

O

Obfuscation（模糊化），364
of software（软件的），361
OFDM，*See Orthogonal frequency division multiplexing*
Open systems interconnection（开放系统互连），259，449-451
Orthogonal frequency division multiplexing（正交频分复用），460
OSI，*See Open systems interconnection*

P

Parallel（并行）
at different levels（不同层次），230
considerations（考量），431
coupling（耦合），425
for performance（对于性能），235-237
grain size（粒度大小），434
interconnecting links（互连链接），432-434
machines（机器），199，230，425
processing（处理），200
speedup（加速），237
worlds biggest machines（世界最大机器），435
Parallel adder（并行加法器），29-31
Parallel architectures（并行体系结构），433
Parallel port（并口），103
Parallel processing unit（并行处理单元），427-431
Parallel topology（并行拓扑），433
PC-card，265
PC/104，259，261
PCB characteristics（PCB 特点），354
PCI，*See Peripheral component interconnect*
PCI express（PCI 增强型），258，264
PCMCIA，*See Personal computer Memory card international association*
PDA，*See Personal digital assistant*
PDP-1，7
Performance（性能）
assessing（评估），113-115
measures（测量），111-113
Peripheral memory mapped（内存映射外设），192
Peripheral component interconnect（外设单元互连），258，261
Personal computer memory card international association（个人计算机存储卡国际协会），259，265
Personal digital assistant（个人数字助理），15，34，439
Pervasive computing（普适计算），426
PFLOPS（每秒 peta 次浮点运算），111
Phase locked loop（锁相环），293
Physical layer（物理层），450
PIC（PIC 机器），67
Pin swapping during layout（布局时引脚转换），358-359
Pipeline（流水线），175
compiler support for（编译器支持），185
dynamic（动态），177，194
efficiency（效率），188
FPU（浮点处理单元），158
mode change（模式转换），177-179
multi-function（多功能），175-177，194
multiple issue superscalar（多发射超标量），197
speedup（加速），175
split（分离），186
stall（停止），187
superscalar（超标量），195，197
superscalar performance（超标量性能），198

throughput（吞吐量），173
PLL，*See Phase locked loop*
Plug and play（即插即用），260
Power（电源）
due to current switching（电流交换），304
ideas for reduction of（减少），307
in semiconductors（半导体），302
low power design（低功耗设计），305-307
on self test（自测），326
PPU，*See Parallel processing unit*
Principles
of locality（定位原则），148-149
Programmable logic device（可编程逻辑设备），362
Propagation delay（传递延迟），303
Propagation delay，example（传递延迟，示例），134
PS/2，264
Pyramidal，view of memory（金字塔形，内存结构），68

Q

Quad core processor（四核处理器），237，295
Quake（震颤），112

R

Radiation damage（辐射损害），340
RAM，67，69，76，314-323
RAMBUS，258
Ramdisk（Ram 磁盘），327，328-329
Random access memory（随机访问存储器），*See RAM*
RDRAM，144
Re-entrant code（重进代码），279
Read only memory（只读存储器），*See ROM*
Real-time（实时），113
definitions（定义），267-268
hard or soft system（硬件或软件系统），266-267
issues（问题），266-271
operating system（操作系统），268，270-271，326，356，379
scheduling（计划），270
stimuli（激励），267
task（任务），268
Reconfigurability（可重构性），166
Reduced instruction set computer（精简指令集计算机），20，67，72，76，79，80，85，93，95，111，159，161，173，193，199，273，371
Reed-Solomon（李德所罗门处理机），342
Register（注册机）
shadow（阴影），209，280
Relative addressing（相对寻址），188
Remote processing（远程处理），427
Reservation station（保留基站），241
Reservation table（保留表），174，181，184
Reset（重置）
circuitry（单回路），294
controller（控制器），346
supervisory IC（监测 IC），346
Retirement algorithm（退回算法），137-138
Reverse engineering（反向工程）
analytical steps（分析步骤），349
mitigation（防范），363
of computer devices（对于计算机设备），349
of software（对于软件），356
structure anaysis（结构分析），351-352
the process of（运行），349-353
Reverse Polish notation（逆波兰表示法），96-98
RISC，*See Reduced instruction set computer*
Rockwell 6502，10，79
ROM，67，76，308-314
RPN，*See Reverse Polish notation*
RS232，*See EIA232 interface*
RS422，*See EIA422 interface*
RS485，*See EIA485 interface*

S

S3C2410，*See Samsung S3C2410*
Samsung（三星公司）
S3C2410，255，256，288，291，298，326，339，370，459
S3CR650B，466
Scan path（扫描通路），336
Scheduling（安排）
deadline monotonic（时限单调），270
earliest deadline first（最早时限优先），270
most important first（最重要优先），270
rate monotonic（频度单调），270
Scoreboard（记分牌），196

Scoreboarding（记分牌），196
Scrambling of bus signals（总线信号加扰），359
SCSI，*See Small computer systems interface*
SD，*See Secure digital*
SDRAM，76，126，127，144，275，292，326
Secure digital（安全位），265，292
Segmentation（片段），138
Serial peripheral interface（串行外围接口），258，324
Serial port（串口），103，331，370
Sign extension（符号扩展），27-28
Signed digit（符号位），439-442
Simulation（仿真）
  of FPGA code（FPGA 代码），471
  of Verilog designs（Verilog 设计），475
Sinclair（Sinclair 公司），112
Sir Clive，79
  ZX Spectrum，79，283
  ZX-79，5
Single chip computer（单芯片计算机），293，316
Single event upset（单粒子翻转），340
Single T-bit branch predictor（单 T-比特分支预测），212-214
Slave processor（从处理器），159
Small computer systems
  interface（小型计算机系统接口），69
SMP，*See Symmetrical multi-processing*
Snooping（侦听），155，157
SoC，*See System-on-chip*
Soft core（软核），166，369-373
Software（软件）
  in embedded systems（嵌入式系统），328
  real time（实时），267
  support for zero overhead loops（支持零开销循环），204
Sony playstation（SONY 游戏平台），238
SPECint and SPECfp（SPEC 整型和 SPEC 浮点型），112
Speculation（预测），182-186
SPI，*See Serial peripheral interface*
Spill code（泄露代码），105
SRAM，115，126，316-317
SSE，*See Streaming SIMD extensions*
SSEM，4
Stack computer（堆栈计算机），96，380
Stanford University（斯坦福大学），78
Streaming SIMD extensions（SIMD 流扩展），17，158，164-165，230，293
StrongARM，17，143，428
SUN（SUN 公司），5，161
  Java processor（JAVA 处理器），173
  picoJAVA，5
  picoJAVA Ⅱ，19
  SPARC，20，78，252，343
Superblocks（超级代码段），228
Superscalar processors（超标量处理器），199
Supervisor mode（系统态），280
Symmetrical multi-processing（对称多任务处理），237
System International（系统国际化），447
System modelling（系统建模），376
System-on-chip（片上系统），15，252，266，292，352

T

Task parallelism（并行任务），230
Temporal scope（时间尺度），268-269
Test（测试），332
  benches（平台），482-483
  by development stage（开发阶段），334
Testing（测试），391
Texas Instruments（德州仪器），77
  DSP processor（DSP 处理器），189
  MSP430，296-298，330
  TMS320，80，178
  TMS32C50，275
  TMS320C50，203，209
TinyCPU
  ALU（算术逻辑单元），383-384，401-403
  alu. v，401-403
  architecture（体系结构），381
  assembler（汇编器），484-488
  comparison operations（比较操作），387
  compiler（编译器），489-490
TinyCPU（*Cont.*）
  control system（控制系统），385-386，388
  counter. v（verilog 源程序），391-394
  data bus（数据总线），381-382

defs. v（verilog 源程序：定义的头文件），391
design specification（设计说明书），380-386
execution state（运行状态），389-390
implementation（实现），390-408
instruction handling（指令控制），384-385
instruction set（指令集），386-390
instruction types（指令类型），386-387
inventor（发明者），379
memory space（存储空间），382-383，396
overview（综述），403-408
programming（编程），409-414
programming tools（编程工具），413-414
RAM，382-383
ram. v（verilog 源程序），396-399
stack（堆栈），382，399-401
stack. v（verilog 源程序），399-401
state machine（矢量状态机），388
state. v（verilog 源程序），394-396
testing（测试），408-409
tinycpu. v（verilog 源程序），403-408
writing code for（编码），409，484
Tomasulo algorithm（Tomasulo 算法），196，240-247
Trace table（跟踪表），213，215
Transistor computer（晶体管计算机），4，5
Trap（陷阱）
of program counter（程序计数），205
Triple redundancy（三重冗余），341-342，430
Tristate buffer（三态缓存），19，70，71，72，73，122
Turbo code（Turbo 码），341
Turing，Alan（人名），3
TWI，*See Two wire interface*
Two wire interface（双线接口），258
Two-bit branch predictor（2-比特分支预测），214-215
TX-0 computer（TX-0 计算机），4

U

U-boot，277，326
UART/USART（通用异步收发器/通用同步异步收发器），192，292，298
UltraSPARC Ⅱ，140
Unicode（统一编码），103
Universal serial bus（通用串行总线），69，254，265，451
UNIX，69，340，480
USB，*See Universal serial bus*

V

VAX（VAX 机器），20，273
Vector parallelism（向量并行），230
Vector processor（向量处理器），104，166
Verification（验证），376，391，482
Verilog（硬件描述语言），369，379，471
Very large scale integration（超大规模集成），8
Very long instruction word（超长指令字），199，422
VHDL（硬件描述语言），369，371，390，408
VIA
Isaiah architecture（Isaiah 体系结构），293-294，298
Nano，293-294，298
Virtual memory（虚拟存储），19，132-133
VLIW，*See Very long instruction word*
VLSI，*See Very large scale integration*
Volatile（C 语言中关键字），192，331，332
Voltage droop（电压降），340，347
von Neumann（冯·诺依曼），17，143，308

W

Watchdog timer（看门狗定时器），345-347
Wetware（大脑），445-446
Whetstone，benchmark（Whetstone 标准检测程序），113
Whirlwind（Whirlwind 计算机）1，5
WiBro（无线宽带），467-468
Wilkes，Maurice（人名），4，75
Wireless（无线）
features（特点），280
for embedded systems（对于嵌入式系统），459-470
interfacing（接口），282
issues（问题），282-283
technology（技术），280-282
USB，466

X

X-ray of circuit（电路的 X 光图），333，354

x86, 21, 80, 140, 162, 164, 165, 305
Xilinx, 363, 379, 390, 408
  ISE, 408
  MicroBlaze, 378

Z

Zero overhead loop（零开销循环）, 110, 202-205
  worked example（工作示例）, 207-208
Zero padding（补零）, 28
ZigBee, 464-465
Zilog
  Z80, 291
  ZOL, *See Zero overhead loop*
  Zuse, Konrad（人名）, 4