计 算 机 科 学 丛 书

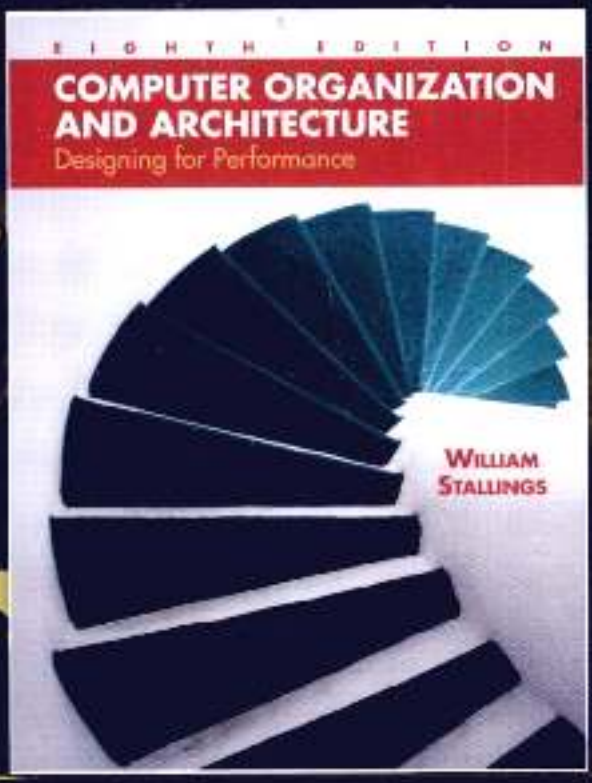
原书第8版

PEARSON

计算机组成与体系结构
性能设计

（美）William Stallings 著 彭蔓蔓 吴强 任小西 等译

Computer Organization and Architecture
Designing for Performance Eighth Edition



机械工业出版社
China Machine Press

计算机组成与体系结构 性能设计（原书第8版）

Computer Organization and Architecture Designing for Performance Eighth Edition

本书是介绍当代计算机体系主流技术和最新技术的优秀教材，以Intel x86和ARM两个处理器系列为例，深入讨论了计算机组成与体系结构的基本原理和概念，并将它们运用到当代计算机系统设计的问题中。

自第7版出版以来，计算机组成与体系结构领域又有了不少革新和进展。第8版坚持全面覆盖整个领域，并在此基础上尽量跟上新技术的步伐。

新增内容

- 交互式模拟工具：提供了20个基于Web的交互式模拟工具，为理解现代处理器的复杂机制提供了有力的支持。
- 嵌入式处理器：以ARM体系结构为例，介绍嵌入式处理器以及它们提供的独特的设计问题。
- 多核处理器：阐述计算机体系结构最流行的新进展——单个芯片上多处理器的使用。
- 高速缓存：对高速缓存内容进行了全面的修订、更新和扩充，涵盖了更宽泛的技术领域。
- 性能评估：扩充了对性能评估的讨论，增加了对基准程序和阿姆达尔定律的分析。
- 汇编语言：增加了一个关于汇编语言和汇编器的新附录。

作者简介



William Stallings 拥有美国麻省理工学院计算机科学博士学位，现任教于澳大利亚新南威尔士大学国防学院（堪培拉）信息技术与电子工程系。他是世界知名计算机学者和畅销教材作者，出版了40多本书籍，内容涉及计算机安全、计算机网络和计算机体系结构等方面，堪称计算机界的全才。他两十次荣获美国“教材和学术专著作者协会”颁发的“年度最佳计算机科学教材”奖。



ISBN 978-7-111-38429-5
定价：56.00元



ISBN 978-7-111-24749-2
定价：49.00元



ISBN 978-7-111-29877-0
定价：56.00元

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

PEARSON

www.pearsonhighered.com

ISBN 978-7-111-32878-0

9 787111 328780

定价：79.00元

原书第8版

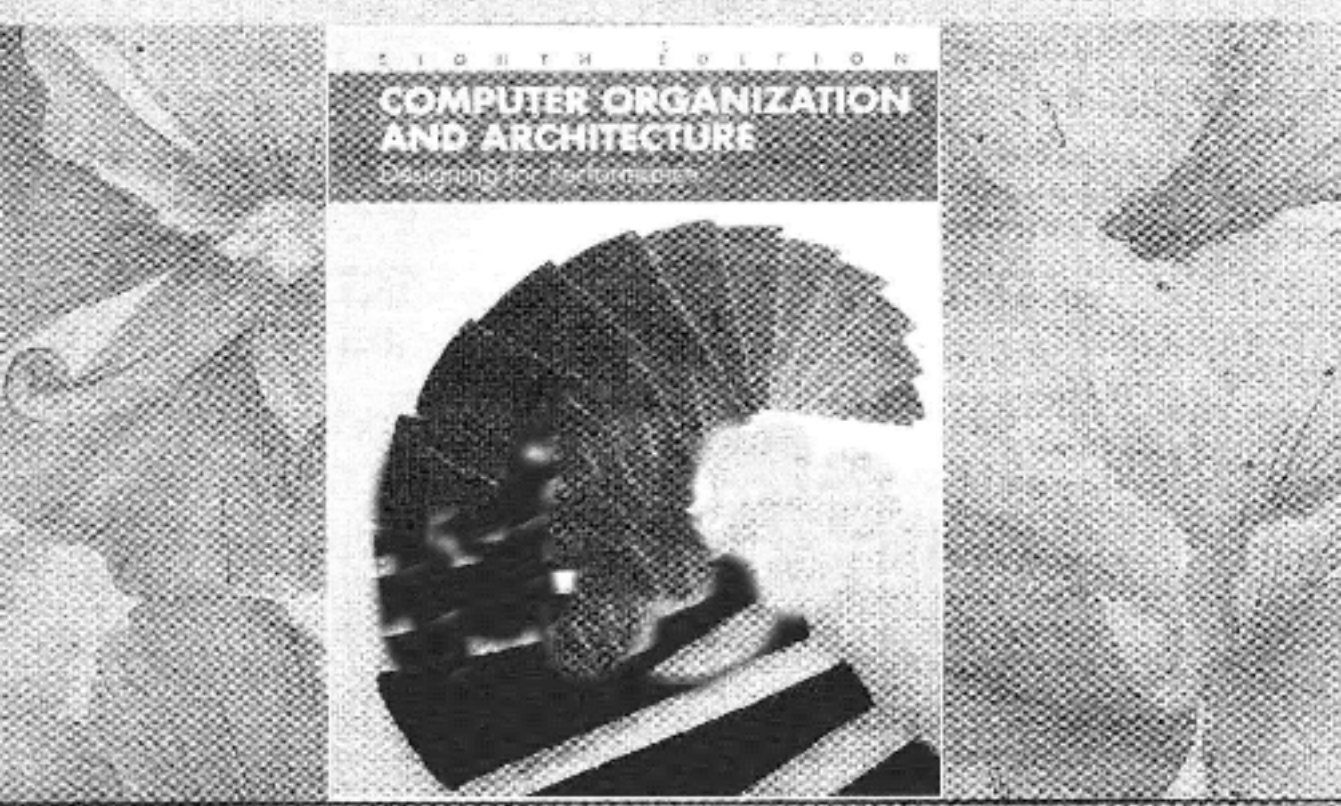
计算机组成与体系结构

性能设计

(美) William Stallings 著　彭蓓蓉 吴 强 任小西 等译

Computer Organization and Architecture

Designing for Performance Eighth Edition



机械工业出版社
China Machine Press

本书以 Intel x86 和 ARM 两个处理器系列为例，结合当代计算机系统性能设计问题，介绍了计算机体系结构的主流技术和最新技术。本书共 18 章，分 5 个部分，第一部分（第 1～2 章）概述计算机组成与体系结构，并讨论计算机的演变和性能；第二部分（第 3～8 章）讨论计算机的主要部件及其互连；第三部分（第 9～14 章）讨论处理器的内部结构和组织；第四部分（第 15～16 章）讨论处理器中控制器的内部结构和微程序设计的使用；第五部分（第 17～18 章）讨论并行组织，包括对称多处理器、集群系统和多核体系结构。

本书可作为高等院校计算机及相关专业的计算机体系结构课程教材或教学参考书，同时也可以作为从事计算机研究与开发的技术人员的参考书。

Simplified Chinese edition copyright © 2011 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Computer Organization and Architecture: Designing for Performance, Eighth Edition* (ISBN 978-0-13-607373-4) by William Stallings, Copyright © 2010, 2006.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，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：01-2009-3516

图书在版编目（CIP）数据

计算机组成与体系结构：性能设计（原书第 8 版）/（美）斯托林斯（Stallings, W.）著；彭蕾蕾等译. —北京：机械工业出版社，2011.5

（计算机科学丛书）

书名原文：Computer Organization and Architecture: Designing for Performance, Eighth Edition

ISBN 978-7-111-32878-0

Ⅰ. 计… Ⅱ. ①斯… ②彭… Ⅲ. 计算机体系结构 Ⅳ. TP303

中国版本图书馆 CIP 数据核字（2010）第 257421 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：迟振春

北京诚信伟业印刷有限公司印刷

2011 年 6 月第 1 版第 1 次印刷

185mm × 260mm　· 31.75 印张

标准书号：ISBN 978-7-111-32878-0

定价：79.00 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，不仅奠定了研究的范畴，还展示了学术的源流，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

译者序

Computer Organization and Architecture: Designing for Performance, 8E

William Stallings 是美国最著名的计算机专业作家之一，他在计算机体系结构，计算机网络和信息安全等方面成就卓著，十次荣获美国教材与大学作者协会颁发的“年度最佳计算机科学与工程教材”奖。本书译自他编著的《Computer Organization and Architecture: Designing for Performance》，该书畅销欧美，一直是美国深受欢迎的大学教材。该教材先后推出了 8 版，这一版对原有内容进行了彻底的修订和重组，使新版对计算机体系结构各专题的阐述更先进、更全面、更清晰。

作者编著本书的目的是使读者知晓当代计算机组成和体系结构的设计原理和实现考虑，并非单纯地讲述概念或理论。为此，本书选用了许多不同机器的例子来阐明和强化所提供的概念。大部分例子来自两种计算机系列：Intel x86 系列和 ARM（先进的 RISC 机器）系列。这两种系统共同概括了当前计算机设计趋势的大部分。Intel x86 结构基本上是一个带有某些 RISC 特征的复杂指令集计算机（CISC），而 ARM 本质上是一个精简指令集计算机（RISC）。两个系统都利用了超标量设计原则，并且都支持多个处理器和多核配置。

全书共分为 5 个部分，第一部分（第 1～2 章）概述计算机组成与体系结构，并讨论计算机的发展演变、性能设计与测评；第二部分（第 3～8 章）介绍计算机的主要部件及其互连、cache 存储器、内部存储器、外部存储器、输入/输出（I/O）以及计算机体系结构与操作系统之间的关系；第三部分（第 9～14 章）介绍中央处理器，包括计算机的运算、指令集结构、处理器的结构和功能、精简指令集计算机（RISC）、指令级并行和超标量方法；第四部分（第 15～16 章）讨论处理器中控制器的内部结构和微程序设计技术；第五部分（第 17～18 章）研究并行组织，包括对称多处理器、集群系统和多核体系结构。本书还具有许多有利于教学的特点，例如，20 个独立的交互式模拟工具的使用、大量插图和表格使讨论更清楚等；每一章均包含关键词、思考题、习题以及推荐的读物和 Web 站点；书末附有一个术语表和参考文献；本书包含的大量扩展性知识可以从配套网站 http://www.pearsonhighered.com/stallings 下载。本书可作为高等院校计算机及其相关专业的计算机体系结构课程教材或教学参考书，同时也可以作为从事计算机研究与开发的技术人员的参考书。

本书的第 0～8 章、前言和术语表由彭蔓蔓翻译，第 9～16 章、第 18 章和附录由吴强翻译，第 17 章由任小西翻译。此外，喻品、李冬妮、李朝、张吉良、邓胜球等研究生也参与了部分工作。在翻译过程中，我们参阅了张昆藏教授等翻译的《计算机组织与体系结构：性能设计》（第 6 版，清华大学出版社 2004 年出版），在此深表感谢！

尽管我们从事计算机组成与体系结构的教学和科研工作多年，在翻译过程中本着认真负责、力求精准的精神，但错误难免，希望广大读者批评指正。

译　者

2011 年 3 月于长沙

目标

这是一本关于计算机结构和功能方面的书，力求清晰完整地给出当今计算机系统的性质和特征。

这项任务极具挑战性，主要有如下几个原因：首先，有非常多的产品类型都冠以“计算机”的名字，从只值几美元的单片机到价值几千万美元的超级计算机。这种多样性不仅体现在价格上，也体现在规模、性能和应用上。其次，计算机技术的快速变化步伐从未停止过，这一点以来就是它的特征。这些变化涉及计算机技术的所有方面，从用于构造计算机部件底层的集成电路技术，到组合这些部件的并行组织概念。

尽管计算机领域存在着种类的多样性和变化的迅速性，但某些基本概念始终不变。当然，这些概念的应用取决于当前的技术状况和设计者的性能/价格目标。本书的目的在于深入讨论计算机组成与体系结构的基本原理和概念，并将它们运用到当代计算机系统设计的问题上。

副标题（性能设计）指出了本书的主题和采用的方法。计算机系统的高性能设计历来都是最重要的，但这一要求从来没有像现在这样强烈和难以满足。计算机系统的所有基本性能特征，包括处理器速度、存储器速度、存储器容量和互连的数据速率，都在迅速提高，而且是以不同的速率在提高。这就使得设计一个实现性能最大化并考虑所有因素影响的平衡系统变得非常困难。于是，计算机设计越来越成为一种博弈，它要以改变一个领域的结构和功能来补偿另一领域的性能失配。我们将会看到，这种博弈贯穿本书的许多设计。

像任何系统一样，计算机系统由一组相互关联的部件组成。通过结构（部件互连的方式）和功能（单个部件的操作）最能表征一个系统。而且，计算机的组织是层次化的，通过将每个主要部件分解成主要子部件，并描述各主要子部件的结构和功能，可以进一步描述各主要部件。为了清晰和易于理解，本书自顶向下地描述这种层次化组织。

- **计算机系统**：主要部件是处理器、存储器和I/O。
- **处理器**：主要部件是控制器（或控制单元）、寄存器、ALU和指令执行单元。
- **控制器**：为所有的处理器部件的操作和协调提供控制信号。传统上，一直使用的是微程序实现方式，其主要部件包括控制存储器、微指令序列逻辑和寄存器。最近，微程序设计已经不占主导地位，但仍是一种重要的实现技术。

本书力求采用在清晰的上下文中组织新素材的方式讲解有关内容，以最大限度地避免读者的迷惑，这样应该比自底向上的讲解方式更好。

考察系统的两个着眼点是体系结构（机器语言程序员可见的系统属性）和组成（实现体系结构的操作单元和它们的互连），它们将贯穿于全书的讨论之中。

使用的范例

本书希望读者熟悉当代操作系统的设计原理和实现方法。因此，纯概念或纯理论的阐述是不适合的。为了说明概念并将它们与现实世界中必须要做的设计选择联系起来，本书选用以下两个处理器系列作为贯穿全书的范例：

- **Intel x86体系结构**：Intel x86体系结构非常广泛地应用于非嵌入式计算机系统。Intel x86基本上是一种复杂指令集计算机（CISC），但具有某些RISC特征。目前的x86系列成员

都采用超标量体系结构和多核设计原则。x86 体系结构的特征演变为学习大多数计算机体系结构的设计原理提供了一种独特的案例。

- ARM：ARM 嵌入式体系结构已被证明是应用最广泛的嵌入式处理器，它应用于手机、音乐播放器、远程传感设备以及很多其他设备。ARM 基本上是一种精简指令集计算机（RISC）。目前的 ARM 系列成员都采用超标量体系结构和多核设计原则。

很多例子（但非全部）都出自这两个计算机系列：Intel x86 和 ARM 嵌入式处理器系列。许多其他系统，包括当代的和历史上的，也提供了计算机体系结构设计特征的重要案例。

本书结构

本书由以下五个部分组成：

- 概论
- 计算机系统
- 中央处理器
- 控制器
- 并行组织，包括多核

本书具有许多利于教学的特点，例如，交互式模拟工具的使用、大量插图和表格使讨论更清楚等。每一章均包含关键词、思考题和习题以及推荐的读物和 Web 站点。书末还附有术语表（列出了常用的缩写词）和参考文献。

读者对象

本书面向高校师生和专业技术人员。它可以作为计算机科学、计算机工程、电气工程等专业本科生一学期或两学期的教材。它覆盖了 CS 220 Computer Architecture 课程要求的所有主题，而计算机体系结构是 IEEE/ACM Computer Curricula 2001 的核心课程之一。

对关注此领域的专业技术人员，本书可作为有参考价值的基础读物，并适合自学。

教学资源

为辅助教学，本书提供了下列资料：

- **解题手册**：解答各章后面的思考题和习题。
- **课题手册**：提供各类课题的详细安排目录。
- **幻灯片**：提供了所有章节的、适于教学的幻灯片。
- **PDF 文件**：复制了书中的所有表格和插图。
- **试题库**：包括判断对错题、多项选择题、填空题与答案。

所有这些支持材料都在教师资源中心（IRC）。要获取这些资源，可以从教师资源中心 http://www.pearsonhighered.com/stallings 下载，或者登录华章网站（www.hzbook.com）下载。

网上资源

本书网站 williamstallings.com/COA/COA8e.html 提供了与其他相关站点的链接、一组有用的文档和勘误等。

本版在 Web 站点上更新了一组习题的有效解决方案，学生可以通过做习题并核对答案来加深对内容的理解。

课题和其他学生练习

许多教师都很清楚，课题训练是计算机组成与体系结构课程的重要部分，它可以使学生获得

实践链接并强化所学到的概念。本书提供的这方面的支持包括：

- **交互式模拟作业**：稍后描述。
- **研究性课题**：一系列研究性作业，指导学生研究 Internet 上的一个特定课题并写出报告。
- **模拟性课题**：教师资源中心提供了 SimpleScalar 和 SMPCache 两个模拟软件包。前者可用于探索计算机组成与体系结构的设计问题；后者为研究对称多处理器（SMP）的 cache 设计问题提供了强有力的教学工具。
- **汇编语言课题**：使用一种简化的汇编语言 CodeBlue，并提供了一些基于流行的 Core Wars 游戏概念的作业。
- **阅读/报告类作业**：每章有一篇或几篇文献论文列表，可以要求学生阅读并写出简短报告。
- **写作作业**：提供一系列写作题目来帮助读者学习。
- **试题库**：包括判断对错题、多项选择题、填空题与答案。

这些课题和其他练习为使用本书教学提供了方便，教师可根据需要酌情选择，选择方法参见附录 A。

交互式模拟工具

第 8 版提供了一些交互式模拟工具，这些模拟工具为理解现代计算机系统的复杂设计特征提供了有力的支持。总共有 20 个交互式模拟工具用于说明计算机组成与体系结构设计中的关键功能与算法，在本书的相应位置用图标来指明。因为这些工具可以让用户建立最初的环境，所以可以利用它们布置学生课外作业。

第 8 版新增内容

自第 7 版出版以来，计算机组成与体系结构领域又有了不少革新和进展。第 8 版努力全面涵盖整个领域，并在此基础上尽量跟上新技术的步伐。为了开始这个修订过程，讲授本课程的几位教授和本领域的几位专家对第 7 版进行了广泛深入的讨论，从而使第 8 版的描述更清晰、更紧凑，图表也进行了改进，同时增加了很多课堂测试的习题。

除了为使本书便于教学和对用户更友好所做的这些工作之外，本书还有许多实质性的改变。第 8 版虽然保留了与第 7 版大致相同的章节，但重新改写并添加了许多内容，主要包括：

- **交互式模拟工具**：第 8 版提供了 20 个基于 Web 的交互式模拟工具，涉及高速缓存（cache）、主存、I/O、分支预测、指令流水线和向量处理等主题。
- **嵌入式处理器**：第 8 版包括一些嵌入式处理器以及它们提供的独特的设计问题。ARM 体系结构被用作一个很好的学习案例。
- **多核处理器**：第 8 版包括现在计算机体系结构最流行的新进展——单个芯片上多处理器的使用，第 18 章将会讨论这一主题。
- **高速缓存**：对第 4 章高速缓存的内容进行了全面的修订、更新和扩充，涵盖了更宽泛的技术领域，并通过使用大量的图片以及交互式模拟工具来改进教学方法。
- **性能评估**：第 2 章显著地扩充了对性能评估的讨论，增加了对基准程序的讨论和对阿姆达尔（Amdahl）定律的分析。
- **汇编语言**：增加了一个关于汇编语言和汇编程序的新附录。
- **可编程逻辑设备**：在第 20 章数字逻辑（网站上）中，扩充了对可编程逻辑设备（PLD）的讨论，介绍了现场可编程门阵列（FPGA）。
- **DDR SDRAM**：DDR 已经成为桌面计算机和服务器中的主流主存技术，尤其是 DDR2 和 DDR3。DDR 技术在第 5 章中介绍，更详细的讨论参见附录 K（网站上）。

- 线性磁带开放协议（LTO）：LTO 已经成为最流行的“超级磁带”形式，并广泛应用于小型和大型的计算机系统中，尤其是作为文件备份。LTO 在第 6 章中介绍，更详细的讨论参见附录 J（网站上）。

对于每个新版本，既要保持合理的页数又要增加新的内容，是一件困难的工作。这是通过删除旧的内容和精简描述实现的。第 8 版将不特别重要的章节和附录放在网上，作为独立的 PDF 文件。这既满足了新内容的扩充，又不增加本书的厚度。

致谢

第 8 版得到许多人的帮助，他们慷慨地为本书贡献了自己宝贵的时间和知识。下列学者审阅了全部或大部分书稿，并提出许多建设性意见：内布拉斯加大学奥马哈分校的 Azad Azadmanesh、夏威夷大学的 Henry Casanova、格林奈尔学院的 Marge Coahran、新墨西哥大学的 Andree Jacobson、加利福尼亚大学波维斯分校的 Kurtis Kredo、奥斯汀皮耶州立大学的 Jiang Li、纽约州立大学奥斯威戈分校的 Rachid Manseur、乔治·梅森大学的 John Masiyowski、温斯顿-萨勒姆州立大学的 Fuad Muztaba、东密歇根大学的 Bill Sverdlik 和科罗拉多大学温泉校区的 Xiaobo Zhou。

同时向以下学者致谢，他们各自审阅了一章内容，并提出了详细的技术意见：Tim Mensch、Balbir Singh、Michael Spratte（惠普公司）、Francois-Xavier Peretmore、John Levine、Jeff Kenton、Glen Herrmannsfeldt、Robert Thorpe、Grzegorz Mazur（波兰华沙工业大学计算机科学学院）、Ian Anseline、Terje Mathisen、Edward Brekelbaum（Varilog 研究公司）、Paul DeMone 和 Mikael Tillenius。还要感谢 AltM 公司的 Jon Marsh 审阅了关于 ARM 的材料。

阿帕拉契州立大学的 Cindy Norris 教授、新布朗斯维克大学的 Bin Mu 教授和阿拉斯加大学的 Kenrick Mock 教授提供了一些习题。

马萨诸塞大学的 Aswin Sreedhar 开发了交互式模拟作业和试题库。

西班牙埃斯特雷马杜拉大学的 Miguel Angel Vega Rodriguez 教授、Juan Manuel Sanchez Perez 教授（博士）和 Juan Antonio Gomez Pulido 教授（博士）为教师手册准备了 SMPCache 问题并提供了 SMPCache 用户指南。

威斯康星大学的 Todd Bezenek 和里海大学的 James Stine 为教师手册准备了 SimpleScalar 问题，Todd 还提供了 SimpleScalar 用户指南。

还要感谢为本书制作幻灯片的利物浦希望大学的 Adrian Pullin。

最后，感谢为本书出版付出努力的人们，他们都做了优秀的工作，包括编辑 Tracy Dunkelberger 和她的助手 Melinda Haggerty 以及产品经理 Rose Kernan。另外，Wayde 出版公司的 Jake Warde 负责本书的审核工作，Patricia M. Daly 负责本书的文字加工。

出版者的话
译者序
前言

第0章 读者指南 …… 1
0.1 本书概要 …… 1
0.2 导读 …… 1
0.3 为何要学习计算机组成和体系结构 …… 1
0.4 因特网与 Web 资源 …… 2
0.4.1 本书的 Web 站点 …… 2
0.4.2 其他 Web 站点 …… 3
0.4.3 USENET 新闻组 …… 3

第一部分 概 论

第1章 导论 …… 6
1.1 计算机组成与体系结构 …… 6
1.2 结构和功能 …… 7
1.2.1 功能 …… 7
1.2.2 结构 …… 8
1.3 关键词和思考题 …… 9
第2章 计算机的演变和性能 …… 10
2.1 计算机简史 …… 10
2.1.1 第一代：真空管 …… 10
2.1.2 第二代：晶体管 …… 15
2.1.3 第三代：集成电路 …… 16
2.1.4 后续几代 …… 20
2.2 性能设计 …… 22
2.2.1 微处理器的速度 …… 23
2.2.2 性能平衡 …… 23
2.2.3 芯片组成和体系结构的改进 …… 25
2.3 Intel x86 体系结构的进展 …… 26
2.4 嵌入式系统和 ARM …… 27
2.4.1 嵌入式系统 …… 27
2.4.2 ARM 的进展 …… 29
2.5 性能评价 …… 30
2.5.1 时钟速度和每秒指令数 …… 30
2.5.2 基准程序 …… 32
2.5.3 阿姆达尔定律 …… 34
2.6 推荐的读物和 Web 站点 …… 35
2.7 关键词、思考题和习题 …… 36

第二部分 计算机系统

第3章 计算机功能和互连的顶层视图 …… 42
3.1 计算机的部件 …… 42
3.2 计算机的功能 …… 44
3.2.1 指令的读取和执行 …… 44
3.2.2 中断 …… 46
3.2.3 I/O 功能 …… 51
3.3 互连结构 …… 51
3.4 总线互连 …… 52
3.4.1 总线结构 …… 52
3.4.2 多总线层次结构 …… 54
3.4.3 总线的设计要素 …… 55
3.5 PCI …… 58
3.5.1 总线结构 …… 58
3.5.2 PCI 命令 …… 61
3.5.3 数据传送 …… 62
3.5.4 仲裁 …… 63
3.6 推荐的读物和 Web 站点 …… 64
3.7 关键词、思考题和习题 …… 64
附录3A 时序图 …… 67
第4章 cache 存储器 …… 69
4.1 计算机存储系统概述 …… 69
4.1.1 存储系统的特性 …… 69
4.1.2 存储器层次结构 …… 71
4.2 cache 存储器原理 …… 73
4.3 cache 的设计要素 …… 75
4.3.1 cache 地址 …… 75
4.3.2 cache 容量 …… 76
4.3.3 映射功能 …… 77
4.3.4 替换算法 …… 85
4.3.5 写策略 …… 85
4.3.6 行大小 …… 86

4.3.7 cache 刷新 …… 86
4.4 Pentium 4 的 cache 组织 …… 88
4.5 ARM 的 cache 组织 …… 90
4.6 推荐的读物 …… 91
4.7 关键词、思考题和习题 …… 91
附录 4A 典型存储器的性能特点 …… 95
第 5 章 内部存储器 …… 100
5.1 半导体主存储器 …… 100
5.1.1 组织 …… 100
5.1.2 DRAM 和 SRAM …… 100
5.1.3 ROM 类型 …… 102
5.1.4 芯片逻辑 …… 103
5.1.5 芯片封装 …… 104
5.1.6 模块组织 …… 105
5.1.7 多体交叉存储器 …… 106
5.2 纠错 …… 107
5.3 高级 DRAM 组织 …… 110
5.3.1 同步 DRAM …… 111
5.3.2 Rambus DRAM …… 112
5.3.3 DDR DRAM …… 113
5.3.4 cache DRAM …… 114
5.4 推荐的读物和 Web 站点 …… 114
5.5 关键词、思考题和习题 …… 115
第 6 章 外部存储器 …… 118
6.1 磁盘 …… 118
6.1.1 磁读写机制 …… 118
6.1.2 数据组织和格式化 …… 119
6.1.3 物理特性 …… 121
6.1.4 磁盘性能参数 …… 122
6.2 RAID …… 124
6.2.1 RAID 0 级 …… 125
6.2.2 RAID 1 级 …… 128
6.2.3 RAID 2 级 …… 128
6.2.4 RAID 3 级 …… 128
6.2.5 RAID 4 级 …… 129
6.2.6 RAID 5 级 …… 130
6.2.7 RAID 6 级 …… 130
6.3 光存储器 …… 131
6.3.1 光盘 …… 131
6.3.2 数字多功能光盘 …… 133
6.3.3 高清晰光盘 …… 134
6.4 磁带 …… 135
6.5 推荐的读物和 Web 站点 …… 136
6.6 关键词、思考题和习题 …… 137
第 7 章 输入/输出 …… 140
7.1 外部设备 …… 140
7.1.1 键盘/监视器 …… 141
7.1.2 磁盘驱动器 …… 142
7.2 I/O 模块 …… 142
7.2.1 模块功能 …… 142
7.2.2 I/O 模块结构 …… 143
7.3 编程式 I/O …… 143
7.3.1 编程式 I/O 概述 …… 144
7.3.2 I/O 命令 …… 144
7.3.3 I/O 指令 …… 144
7.4 中断驱动式 I/O …… 146
7.4.1 中断处理 …… 146
7.4.2 设计问题 …… 148
7.4.3 Intel 82C59A 中断控制器 …… 149
7.4.4 Intel 82C55A 可编程外围接口 …… 150
7.5 直接存储器存取 …… 151
7.5.1 编程式 I/O 和中断驱动式 I/O 的缺点 …… 151
7.5.2 DMA 功能 …… 151
7.5.3 Intel 8237A DMA 控制器 …… 153
7.6 I/O 通道和处理器 …… 155
7.6.1 I/O 功能的演变 …… 155
7.6.2 I/O 通道的特点 …… 155
7.7 外部接口：FireWire 和 InfiniBand …… 156
7.7.1 接口类型 …… 156
7.7.2 点对点和多点配置 …… 156
7.7.3 FireWire 串行总线 …… 157
7.7.4 InfiniBand …… 159
7.8 推荐的读物和 Web 站点 …… 162
7.9 关键词、思考题和习题 …… 162
第 8 章 操作系统支持 …… 166
8.1 操作系统概述 …… 166
8.1.1 操作系统的目标与功能 …… 166
8.1.2 操作系统的类型 …… 168
8.2 调度 …… 173
8.2.1 长期调度 …… 173
8.2.2 中期调度 …… 173
8.2.3 短期调度 …… 173
8.3 存储器管理 …… 176

8.3.1 交换 …… 177
8.3.2 分区 …… 177
8.3.3 分页 …… 179
8.3.4 虚拟存储器 …… 180
8.3.5 快表 …… 182
8.3.6 分段 …… 183
8.4 Pentium 存储器管理 …… 184
8.4.1 地址空间 …… 184
8.4.2 分段 …… 184
8.4.3 分页 …… 186
8.5 ARM 存储器管理 …… 187
8.5.1 存储器系统组织 …… 187
8.5.2 虚拟存储器地址转换 …… 187
8.5.3 存储器管理格式 …… 189
8.5.4 存取控制 …… 190
8.6 推荐的读物和 Web 站点 …… 191
8.7 关键词、思考题和习题 …… 191

第三部分 中央处理器

第9章 计算机算术 …… 196
9.1 算术逻辑单元 …… 196
9.2 整数表示 …… 196
9.2.1 符号-幅值表示法 …… 197
9.2.2 2 的补码表示法 …… 197
9.2.3 不同位长间的转换 …… 199
9.2.4 定点表示法 …… 200
9.3 整数算术 …… 200
9.3.1 取负 …… 200
9.3.2 加法和减法 …… 201
9.3.3 乘法 …… 203
9.3.4 除法 …… 207
9.4 浮点表示 …… 208
9.4.1 原理 …… 208
9.4.2 二进制浮点表示的 IEEE 标准 …… 211
9.5 浮点算术 …… 212
9.5.1 浮点加法和减法 …… 213
9.5.2 浮点乘法和除法 …… 214
9.5.3 精度考虑 …… 215
9.5.4 二进制浮点算术的 IEEE 标准 …… 216
9.6 推荐的读物和 Web 站点 …… 218
9.7 关键词、思考题和习题 …… 219

第10章 指令集：特征和功能 …… 222
10.1 机器指令特征 …… 222
10.1.1 机器指令要素 …… 222
10.1.2 指令表示 …… 223
10.1.3 指令类型 …… 224
10.1.4 地址数目 …… 225
10.1.5 指令集设计 …… 226
10.2 操作数类型 …… 226
10.2.1 数值 …… 227
10.2.2 字符 …… 227
10.2.3 逻辑数据 …… 228
10.3 Intel x86 和 ARM 数据类型 …… 228
10.3.1 x86 数据类型 …… 228
10.3.2 ARM 数据类型 …… 229
10.4 操作类型 …… 230
10.4.1 数据传送 …… 232
10.4.2 算术运算 …… 233
10.4.3 逻辑运算 …… 233
10.4.4 转换 …… 234
10.4.5 输入/输出 …… 235
10.4.6 系统控制 …… 235
10.4.7 控制转移 …… 235
10.5 Intel x86 和 ARM 操作类型 …… 238
10.5.1 x86 操作类型 …… 238
10.5.2 ARM 操作类型 …… 244
10.6 推荐的读物 …… 246
10.7 关键词、思考题和习题 …… 246
附录 10A 栈 …… 250
附录 10B 小端、大端和双端 …… 253
第11章 指令集：寻址方式和指令格式 …… 256
11.1 寻址方式 …… 256
11.1.1 立即寻址 …… 257
11.1.2 直接寻址 …… 257
11.1.3 间接寻址 …… 257
11.1.4 寄存器寻址 …… 258
11.1.5 寄存器间接寻址 …… 258
11.1.6 偏移寻址 …… 258
11.1.7 栈寻址 …… 260
11.2 x86 和 ARM 寻址方式 …… 260
11.2.1 x86 寻址方式 …… 260
11.2.2 ARM 寻址方式 …… 262
11.3 指令格式 …… 264

11.3.1 指令长度 …… 264
11.3.2 位的分配 …… 265
11.3.3 变长指令 …… 267
11.4 x86 和 ARM 指令格式 …… 269
11.4.1 x86 指令格式 …… 269
11.4.2 ARM 指令格式 …… 271
11.5 汇编语言 …… 272
11.6 推荐的读物 …… 274
11.7 关键词、思考题和习题 …… 274
第 12 章 CPU 结构和功能 …… 277
12.1 CPU 组成 …… 277
12.2 寄存器组成 …… 278
12.2.1 用户可见寄存器 …… 278
12.2.2 控制和状态寄存器 …… 280
12.2.3 微处理器寄存器组成的例子 …… 281
12.3 指令周期 …… 282
12.3.1 间接周期 …… 282
12.3.2 数据流 …… 283
12.4 指令流水线技术 …… 283
12.4.1 流水线策略 …… 284
12.4.2 流水线性能 …… 286
12.4.3 流水线冒险 …… 288
12.4.4 处理分支指令 …… 289
12.4.5 Intel 80486 的流水线 …… 292
12.5 x86 系列处理器 …… 293
12.5.1 寄存器组成 …… 294
12.5.2 中断处理 …… 298
12.6 ARM 处理器 …… 299
12.6.1 处理器组成 …… 300
12.6.2 处理器模式 …… 301
12.6.3 寄存器组成 …… 301
12.6.4 中断处理 …… 303
12.7 推荐的读物 …… 304
12.8 关键词、思考题和习题 …… 304
第 13 章 精简指令集计算机 …… 308
13.1 指令执行特征 …… 309
13.1.1 操作 …… 309
13.1.2 操作数 …… 310
13.1.3 过程调用 …… 311
13.1.4 推论 …… 311
13.2 大寄存器组方案的使用 …… 311
13.2.1 寄存器窗口 …… 312
13.2.2 全局变量 …… 313
13.2.3 大寄存器组与高速缓存的对比 …… 313
13.3 基于编译器的寄存器优化 …… 314
13.4 精简指令集体系结构 …… 315
13.4.1 采用 CISC 的理由 …… 315
13.4.2 精简指令集体系结构特征 …… 317
13.4.3 CISC 与 RISC 对比对比 …… 318
13.5 RISC 流水线技术 …… 319
13.5.1 使用规整指令的流水线技术 …… 319
13.5.2 流水线的优化 …… 320
13.6 MIPS R4000 …… 322
13.6.1 指令集 …… 322
13.6.2 指令流水线 …… 324
13.7 SPARC …… 327
13.7.1 SPARC 寄存器组 …… 327
13.7.2 指令集 …… 328
13.7.3 指令格式 …… 329
13.8 RISC 与 CISC 的争论 …… 330
13.9 推荐的读物 …… 331
13.10 关键词、思考题和习题 …… 331
第 14 章 指令级并行性和超标量处理器 …… 335
14.1 概述 …… 335
14.1.1 超标量与超级流水线的对比 …… 336
14.1.2 限制 …… 337
14.2 设计考虑 …… 338
14.2.1 指令级并行性和机器并行性 …… 338
14.2.2 指令发射策略 …… 339
14.2.3 寄存器重命名 …… 341
14.2.4 机器并行性 …… 342
14.2.5 分支预测 …… 342
14.2.6 超标量执行 …… 343
14.2.7 超标量实现 …… 343
14.3 Pentium 4 …… 343
14.3.1 前端 …… 347
14.3.2 乱序执行逻辑 …… 348
14.3.3 整数和浮点执行单元 …… 349
14.4 ARM CORTEX-A8 …… 349

14.4.1 指令取指单元 …… 349
14.4.2 指令译码单元 …… 351
14.4.3 整数执行单元 …… 353
14.4.4 SIMD 和浮点流水线 …… 354
14.5 推荐的读物 …… 355
14.6 关键词、思考题和习题 …… 356

第四部分 控 制 器

第 15 章 控制器操作 …… 362
15.1 微操作 …… 362
15.1.1 取指周期 …… 363
15.1.2 间接周期 …… 364
15.1.3 中断周期 …… 365
15.1.4 执行周期 …… 365
15.1.5 指令周期 …… 366
15.2 处理器控制 …… 367
15.2.1 功能需求 …… 367
15.2.2 控制信号 …… 367
15.2.3 控制信号举例 …… 368
15.2.4 处理器内部的组织 …… 369
15.2.5 Intel 8085 …… 370
15.3 硬布线实现 …… 373
15.3.1 控制器输入 …… 373
15.3.2 控制器逻辑 …… 374
15.4 推荐的读物 …… 374
15.5 关键词、思考题和习题 …… 375
第 16 章 微程序控制 …… 376
16.1 基本概念 …… 376
16.1.1 微指令 …… 376
16.1.2 微程序控制器 …… 378
16.1.3 Wilkes 控制 …… 379
16.1.4 优缺点 …… 382
16.2 微指令定序 …… 382
16.2.1 设计考虑 …… 382
16.2.2 定序技术 …… 382
16.2.3 地址生成 …… 384
16.2.4 LSI-11 微指令定序 …… 384
16.3 微指令执行 …… 385
16.3.1 微指令的分类法 …… 385
16.3.2 微指令编码 …… 387
16.3.3 LSI-11 微指令执行 …… 388
16.3.4 IBM 3033 微指令执行 …… 390
16.4 TI 8800 …… 391
16.4.1 微指令格式 …… 391
16.4.2 微定序器 …… 393
16.4.3 寄存器式 ALU …… 395
16.5 推荐的读物 …… 397
16.6 关键词、思考题和习题 …… 397

第五部分 并 行 组 织

第 17 章 并行处理 …… 400
17.1 多处理器组织 …… 401
17.1.1 并行处理器系统的类型 …… 401
17.1.2 并行组织 …… 402
17.2 对称多处理器 …… 402
17.2.1 组织 …… 403
17.2.2 多处理器操作系统设计考虑 …… 405
17.2.3 大型机 SMP …… 405
17.3 cache 一致性和 MESI 协议 …… 407
17.3.1 软件解决方案 …… 408
17.3.2 硬件解决方案 …… 408
17.3.3 MESI 协议 …… 409
17.4 多线程和片上多处理器 …… 411
17.4.1 隐式和显式多线程 …… 412
17.4.2 显式多线程的方式 …… 413
17.4.3 示例系统 …… 415
17.5 集群 …… 416
17.5.1 集群配置 …… 417
17.5.2 操作系统设计问题 …… 418
17.5.3 集群计算机体系结构 …… 419
17.5.4 刀片服务器 …… 420
17.5.5 集群与 SMP 的对比 …… 421
17.6 非均匀存储器访问 …… 421
17.6.1 动机 …… 421
17.6.2 组织 …… 422
17.6.3 NUMA 的优缺点 …… 423
17.7 向量计算 …… 424
17.7.1 向量计算的方法 …… 424
17.7.2 IBM 3090 向量机制 …… 427
17.8 推荐的读物和 Web 站点 …… 432
17.9 关键词、思考题和习题 …… 433
第 18 章 多核计算机 …… 437
18.1 硬件性能问题 …… 437
18.1.1 增加并行 …… 437
18.1.2 功耗 …… 439

18.2 软件性能问题 …… 440
18.2.1 多核软件 …… 440
18.2.2 应用实例：Valve 游戏软件 …… 441
18.3 多核组织结构 …… 442
18.4 Intel x86 多核结构 …… 443
18.4.1 Intel Core Duo …… 443
18.4.2 Intel Core i7 …… 445
18.5 ARM11 MPCore …… 445
18.5.1 中断处理 …… 446
18.5.2 cache 一致性 …… 448
18.6 推荐的读物和 Web 站点 …… 448
18.7 关键词、思考题和习题 …… 449
附录 A 计算机组成与体系结构的教学课题 …… 451
附录 B 汇编语言及相关主题 …… 455
术语表 …… 473
参考文献 …… 480

读者指南

本书及其配套 Web 站点介绍了大量材料。本章将为读者做简要的概述。

0.1 本书概要

本书由五部分组成：

第一部分：概述计算机组成与体系结构，并讨论计算机设计的发展演变。

第二部分：考察计算机的主要部件及其互连，这既包括部件之间的彼此互连，也包括它们与外部世界的互连。这部分还详细讨论内部存储器、外部存储器和输入/输出（I/O）。最后，考察计算机体系结构与运行于其上的操作系统之间的关系。

第三部分：考察处理器的内部结构和组织。首先，深入讨论计算机的运算，然后考察指令集结构，其余章节研究处理器的结构和功能，包括精简指令集计算机（RISC）和超标量方法的讨论。

第四部分：讨论处理器中控制单元的内部结构和微程序设计的使用。

第五部分：研究并行组织，包括对称多处理器、集群系统和多核体系结构。

本书网站中的许多在线章节和附录包含了与本书相关的一些附加主题。

每一部分的开头给出了这一部分所包含的各章的简要介绍。

本书的目的是使读者知晓当代计算机组成和体系结构的设计原理和实现考虑。因此，纯概念或纯理论的讲述是不妥当的。本书选用了许多不同机器的例子来阐明和强化所提供的概念。很多（但不意味全部）例子来自两种计算机系列：Intel x86 系列和 ARM（先进的 RISC 机器）系列。这两种系统涵盖了当前计算机设计趋势的大部分。Intel x86 结构基本上是一个带有某些 RISC 特征的复杂指令集计算机（CISC），而 ARM 本质上是一个精简指令集计算机（RISC）。两个系统都利用了超标量设计原则，并且都支持多个处理器和多核配置。

0.2 导读

本书采用自顶向下的方法来介绍。正如我们在 1.2 节中详细讨论的那样，计算机系统可以看成是一种层次结构。在顶层，我们关注计算机的主要功能部件：处理器、I/O、存储器、外围设备。第二部分考察这些功能部件，并且详细分析除处理器以外的各个功能部件。这种方法使我们能够了解驱动处理器设计的外部功能需要，从而导出第三部分。在第三部分中，我们将详细地考察处理器。由于有了第一部分介绍的知识，因此，在第二部分中，我们可以看清必须做的设计决定，以便处理器能够支持计算机系统的全部功能。然后，在第四部分，我们讨论控制单元，它是处理器的心脏。而且控制单元的设计最好能利用第三部分介绍的处理器执行的功能来解释。最后，第五部分介绍多处理器系统，包括集群系统、多处理器计算机和多核计算机。

0.3 为何要学习计算机组成和体系结构

《IEEE/ACM 计算机课程 2001》是由 IEEE（电气与电子工程师学会）的计算机学会和 ACM（美国计算机协会）共同组织的 Joint Task Force on Computing Curricula 制定的新世纪第一个计算机课程大纲，它将计算机体系结构列为所有计算机科学和计算机工程专业的学生必修的核心课程之一。该报告中指出：

计算机是计算的核心。若没有计算机，今天的大多数计算机学科将只是理论数学的分支。对

于当今任何计算领域的专业人员而言，不应当把计算机只看成是盲目地执行程序的黑匣子，而应当要求所有计算机专业的学生对计算机系统的功能部件、特征、性能以及它们之间的相互作用有某种程度的理解和正确评价。当然这里也有实践的关联性。学生需要理解计算机体系结构，以便更好地编制程序使其在实际机器上高效运行。在选择使用的系统时，他们应该能够理解各种部件间的权衡，例如 CPU 的时钟速率与内存的大小权衡。

《计算机工程 2004 课程指导》是 Task Force 小组的一个更近版本，它强调计算机组成与体系结构课程的重要性如下：

计算机体系结构是计算机工程的一个关键部件，在业的计算机工程师应该对这个主题有实践性的理解，它关系到中央处理单元的设计与组织，以及集成 CPU 到计算机系统本身的所有方面。体系结构向上可以拓展到计算机软件，因为处理器的架构必须与操作系统和系统软件结合，没有计算机体系结构的知识是很难设计出一个很好的操作系统的。此外，计算机设计者为了实现优化的系统结构又必须掌握软件方面的知识。

计算机系统结构课程必须完成多个目标，它必须提供一个计算机系统的整个概貌，以及教会学生操作典型的计算机系统。它必须阐需基本的原理，同时承认现有的商业系统的复杂性。理想上，它应该加强计算机工程中其他领域的一些普遍概念，例如，寄存器间接寻址应加强 C 语言中的指针概念。最后，学生必须理解各种外部设备如何相互结合以及它们如何与 CPU 相连接。

文献［CLEM00］给出了作为学习计算机体系结构理由的如下例子：

（1）假设一个大学毕业生进入一个企业工作，被要求为一个大型组织选择一台性价比最高的计算机。他将面临各种各样的选择，例如更大的高速缓存或更高的时钟速率，理解设备之间的相互关联是做出决定所必要的。

（2）许多处理器不是用于 PC 机或服务器，而是用于嵌入式系统，如嵌入在智能汽车电子控制器这类实时系统或更大的系统中，设计者可以用 C 语言给处理器编程。调试系统时可能会使用逻辑分析仪，它能显示引擎传感器发出的中断请求与机器级代码间的相互关系。

（3）计算机体系结构使用的概念也能应用到其他课程中。特别是，计算机对程序设计语言和操作系统提供结构性支持的方式，强化了这些领域的概念。

正如仔细阅读本书目录时所看到的，计算机组成和体系结构包括了范围广泛的设计问题和概念。对这些概念有一个好的、全面的理解，不仅对今天其他领域的学习而且对毕业后的工作都是有益的。

0.4 因特网与 Web 资源

Internet 和 Web 网上有许多有用的资源，它们提供对本书的支持，并且帮助读者跟踪此领域的最新进展。

0.4.1 本书的 Web 站点

本书的 Web 站点是 WilliamStallings.com/COA/COA8e.html。关于此站点的详细描述请见本书的开始部分。

此 Web 站点有本书的勘误表，它列出了已发现的错误，并将随时更新。如果读者发现了书中的错误，请给作者发 E-mail。WilliamStallings.com 还有作者其他著作的勘误表。

作者还维护着计算机科学学生资源站点，网址为 WilliamStallings.com/StudentSuport.html。该站点的目的是为计算机科学的学生和专业人士提供文档、信息和链接。其链接和文档被组织成如下 6 类：

- Math（数学）：包括基本的数学复习资料，排队论入门、数值系统的初级读物以及与许多数学站点的链接。

- How-to（如何做）：为解决家庭作业问题、写技术报告和准备技术报告所提供的建议和指导。
- Research resources（研究资源）：到重要论文、技术报告和文献目录的链接。
- Miscellaneous（混杂资源）：其他各类有用的资料和链接。
- Computer science careers：为计算机科学从业人员准备的有用文献和链接。
- Humor and other diversions（幽默与娱乐）：您可以放松您的心情一小会儿。

0.4.2 其他 Web 站点

还有许多其他 Web 站点提供了与本书主题相关的信息，在后续的各章中，可以在参考文献和 Web 站点部分找到列出的特殊 Web 站点。因为 Web 站点地址经常发生变动，故本书未提供这些站点的 URL，但在本书中列出的所有 Web 站点都可以找到这些站点的相近链接。本书未提到的其他链接将随时添加到 Web 站点中。

下列站点提供有关计算机组成与体系结构的重要信息：

- WWW Computer Architecture Home Page（WWW 计算机体系结构主页）：提供有关计算机体系结构领域研究人员的全面检索信息，包括结构小组的名址和课题、技术组织、文献、成员和商业信息。
- CPU Info Center（CPU 信息中心）：专门提供处理器的信息，包括技术论文、产品信息和最近发布的公告。
- Processor Emporium（处理器商业中心）：提供有趣和有益的信息集合。
- ACM Special Interest Group on Computer Architecture（美国计算机协会的计算机体系结构专业组）：提供 SIGARCH 的活动和出版消息。
- IEEE Technical Committee on Computer Architecture（美国电气电子工程师学会的计算机体系结构技术委员会）：提供 TCAA 的业务通信。

0.4.3 USENET 新闻组

几个 USENET 新闻组致力于讨论计算机组成和结构的某些方面。与所有实际新闻组一样，这里也常充斥着令人不感兴趣的文章，但尝试一下，看看是否有感兴趣的内容，仍是值得的。最相关的几个新闻组如下所示。

- comp.arch：讨论计算机体系结构的综合新闻组，经常有相当好的文章。
- comp.arch.arithmetic：讨论计算机算法和标准的新闻组。
- comp.arch.storage：讨论的范围从产品到技术，再到实际应用问题。
- comp.parallel：讨论并行计算机及其应用。

第一部分 概 论

第一部分为本书其余部分提供了背景知识以及各章节之间的联系，并给出了计算机组成与体系结构的基本概念。

第1章 导论

第1章介绍了计算机作为层次系统的概念。一台计算机可以看成是由一些部件组成的，其功能由各个部件的集合功能来决定。每个部件又是通过其内部结构和功能来描述的。第1章按层次化观点介绍了计算机的主要层次，而本书的其他各章都是根据这些层次自上至下组织的。

第2章 计算机的演变和性能

第2章有两个目的。首先讨论计算机技术发展的简史，这是引入计算机组成和体系结构基本概念的轻松而有趣的方式。计算机系统设计聚焦于性能，本章接着介绍影响性能的技术发展趋势，并概述为实现平衡、有效的系统性能而采用的各种技术和策略。

第1章

Computer Organization and Architecture: Designing for Performance, 8E

导　论

本书是关于计算机结构和功能的书，其目标是尽可能清晰、完整地介绍当代计算机系统的性质和特征。这项任务颇具挑战性，主要原因有两方面。

首先，有各种各样的产品，从几美元的单片机到价值几千万美元的超级计算机，都可以称为计算机。多样性不仅表现在计算机的成本上，而且还表现在计算机的体积、性能和应用上。其次，计算机的发展步伐相当快，绝无停顿。这些发展表现在计算机技术的各个方面，从用于构成计算机底层的集成电路技术，到越来越广泛采用的将这些部件组合起来的并行组织技术。

尽管计算机领域富有多样性并且仍在改变，但始终存在一些基本的概念。当然，这些概念的应用依赖于技术的发展状况，以及设计者所要达到的性能/价格目标。本书的目的在于深入讨论计算机组成与体系结构的基本原理，并将它们应用到当代计算机设计的问题上。本章介绍将要采用的描述方法。

1.1　计算机组成与体系结构

在描述计算机系统时，常常要区分计算机体系结构和计算机组成这两个基本概念。虽然很难给出这两个术语的精确定义，但对它们所涉及的领域存在着共识（见文献［VRAN80］、［SIEW82］和［BELL78a］），一种有趣的可供选择的观点可参见文献［REDD76］。

计算机体系结构是那些对程序员可见的系统属性，换句话说，这些属性直接影响到程序的逻辑执行。**计算机组成**是实现结构规范的操作单元及其相互连接。例如，体系结构的属性包括指令集、用来表示各种数据类型（例如，数据、字符）的比特数，输入输出机制以及内存寻址技术。组成的属性包括那些对程序员可见的硬件细节，如控制信号，计算机和外设的接口以及存储器使用的技术。

例如，计算机是否有乘法指令是体系结构设计的问题。而这条指令是由特定的乘法单元实现，还是通过重复使用系统的加法单元来实现，则是组成的问题。组成基于乘法单元使用的预期频度、两种方案的相对速度以及特定乘法单元的成本和物理尺寸等因素。

无论是过去还是现在，了解体系结构与组成之间的差别都是很重要的。很多计算机制造商会提供系列机产品，它们有着相同的体系结构，但组成是不相同的，因此，同一系列中不同型号的计算机的价格和性能也不相同。进一步来说，一种特殊的体系结构可以存在多年，并且覆盖多种不同的计算机型号，但它的组成则随着技术的进步而不断更新。这种现象的一个突出例子是IBM System/370体系结构。这种架构于1970年推出，包括多种型号。低需求的客户可以购买较便宜、速度较慢的型号，如果今后要求提高了，可以升级到更贵的、速度更快的型号，而不必丢弃已经开发的软件。近几年，IBM通过改进技术推出了许多新型号来替代旧的型号，为用户提供高速、低价或两者兼备的产品。这些新型号保留了同样的体系结构，因而保障了用户的软件资源。值得注意的是，System/370体系结构经过几次增强，不但生存至今，而且仍是IBM的旗舰产品。

在被称之为微计算机的一类计算机系统中，体系结构和组成的关系非常紧密。技术的更新不仅影响了计算机的组成，而且还导致了更强大和更复杂的体系结构。通常，越小的机器，新旧两代之间的兼容性要求越少，因此组成和体系结构设计决策的关系就更加紧密。关于它的一个有趣的例子就是精简指令集计算机（RISC），本书将在第13章进行深入的讨论。

本书介绍计算机组成和计算机体系结构两个方面的内容，或许更强调组成方面的内容。但

是，计算机组成的设计必须遵照特定的体系结构规范，因此，对组成的深入论述也要求对体系结构有同样细致的考察。

1.2 结构和功能

计算机是一个复杂的系统，当代计算机包含数百万个电子元件，那么，怎样才能清楚地描述它们呢？关键就在于认识包括计算机在内的大多数复杂系统的分层性质［SIMO96］。层次系统是一系列互相关联的子系统，每个子系统又在结构上分层，直到分成我们所能达到的一些基本子系统的最低级。

复杂系统的层次特性是设计和描述它们的基础。设计者每次只需要处理系统的某个特定层次即可。在每一层中，系统由一组部件及其相互关联所组成。每一层的行为仅仅依赖于系统下一层更为简单的抽象特征。在每一层上，设计者关心的是结构和功能。

- **结构**：部件相互关联的方法。
- **功能**：作为结构组成部分的单个独立部件的操作。

根据描述，有两种选择：由底层开始，向上建立完善的描述；或者从顶层开始，将系统分解成各个子部分。许多领域的事实证明，自顶向下的方法是最清晰且最有效的方法［WEIN75］。

本书采用的方法也遵循这一观点，将自顶向下地描述计算机系统。从系统的主要部件开始，描述它的结构和功能，然后逐级深入推进到层次结构的底层。本节的其余部分将为这种逐级推进的描述提供简短的概述。

1.2.1 功能

从本质上来说，计算机的结构和功能都很简单。图1-1描述了计算机所能执行的基本功能，概括起来包含4项：数据处理、数据存储、数据传送、控制。

当然，计算机必须能处理数据。数据可以有多种形式，处理的要求也很广泛。但是我们将看到数据处理的基本方法或类型只有几种。

计算机存储数据也很重要。即使计算机最简单地处理数据（例如，数据输入并处理、结果直接输出），它至少也必须在某个特定的时刻临时存储它正在运算的数据值。因此，计算机至少要有短期数据存储的功能。计算机长期存储数据的功能也同样重要。存储在计算机中的数据文件可以用于以后的检索或更新。

计算机必须能在自身和外界之间传送数据。计算机的操作包含作为数据源或者目的地的设备。当数据从直接与计算机相连的设备中发送或接收时，这个过程被称为输入输出（I/O），而这个设备被称为外围设备（peripheral）。当数据传至更远处，或从远方设备接收时，这个过程称为数据通信。

最后，必须对这3种功能进行控制。这种控制功能最终是由给计算机提供指令的人来施加的。在计算机内部，控制器根据这些指令管理计算机的资源，并协调各个功能部件的性能。

在这种通常讨论的层次上，能够执行的操作非常少。图1-2描述了4种可能的操作类型。计算机可以作为数据传送功能的设备（如图1-2a所示），简单地将数据从一个外围设备或通信线路传到另一个。计算机同样可以作为数据存储功能的设备（如图1-2b所示），将数据从外部环境传送到计算机的存储器（读），反之亦然（写）。最后的两个图描述了涉及数据处理的操作，被处理的数据或者在存储器中（如图1-2c所示），或者在存储器与外部环境之间的路径中（如图1-2d所示）。

前面的讨论似乎过于概括，但即使在计算机结构的最高层次，区分许多不同的功能仍是可能的。这里引用文献［SIEW82］中的一段话："为适应功能而改变计算机的结构的情况很少发生。计算机的通用性是根本，所有的功能专门化均发生在编程阶段，而不是设计阶段。"

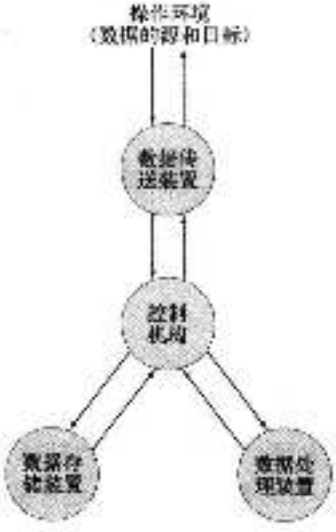


图 1-1 计算机的功能

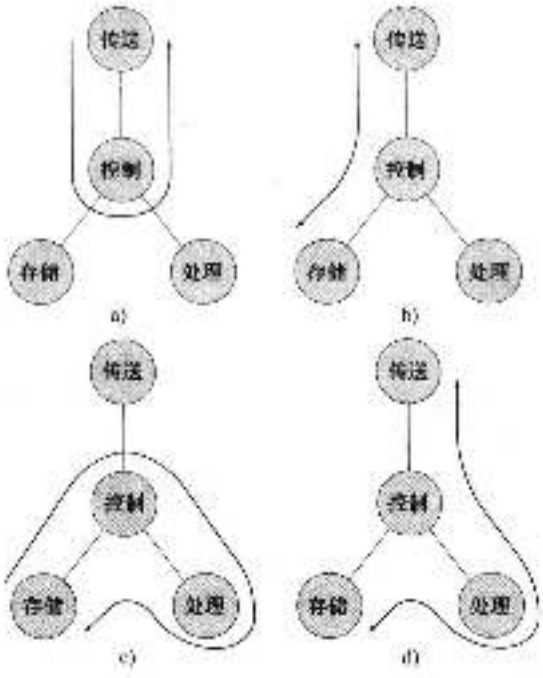


图 1-2 可能的计算机操作

1.2.2 结构

图 1-3 是对计算机尽可能简单的描述。计算机是以某种方式与其外部环境交互的实体，通常，它与外部环境的所有连接可以划分为外围设备和通信线路，我们将在后面对这两种连接进行讨论。

但本书更关心的是计算机本身的内部结构，如图 1-4 所示。有 4 种主要的结构组件：

- **中央处理单元（CPU）**：它控制计算机的操作并且执行数据处理功能，通常简单地被称为处理器。
- **主存储器**：存储数据。
- **I/O**：在计算机及其外部环境之间传输数据。
- **系统互连**：为 CPU、主存储器和 I/O 之间提供一些通信机制。



图 1-3 计算机

系统互连常见的例子是利用**系统总线**，它由一系列导线组成，所有其他组件连接在导线上。

上述各种组件可能有一个或者多个，但传统上，处理器仅仅具有一个。近年来，在单机系统中采用多个处理器的情况越来越多。随着内容的深入，本书会涉及和讨论有关多处理器系统的设计问题，第五部分将专门讨论这样的系统。

第二部分将详细地考察以上每个组件。但对我们来说，最有趣的、在某种程度上也是最复杂的组件是 CPU，其主要结构组件如下所示：

- **控制单元**：控制 CPU 以至于整个计算机的操作。
- **算术逻辑单元（ALU）**：执行计算机的数据处理功能。
- **寄存器**：提供 CPU 的内部存储。
- **CPU 内部互连**：提供控制器、ALU 和寄存器之间的某种通信机制。

第三部分将仔细讨论以上各个部件，我们将看到使用并行和流水线技术所带来的复杂性。最后，实现控制器的方法有多种，一种常用的方法是微程序的实现技术。从根本上讲，微程序控

制器通过执行一系列的微指令来操作，而微指令定义了控制器的功能。使用这种方法，控制器的结构可以描述为图 1-4 所示，本书在第四部分将会讨论这种结构。

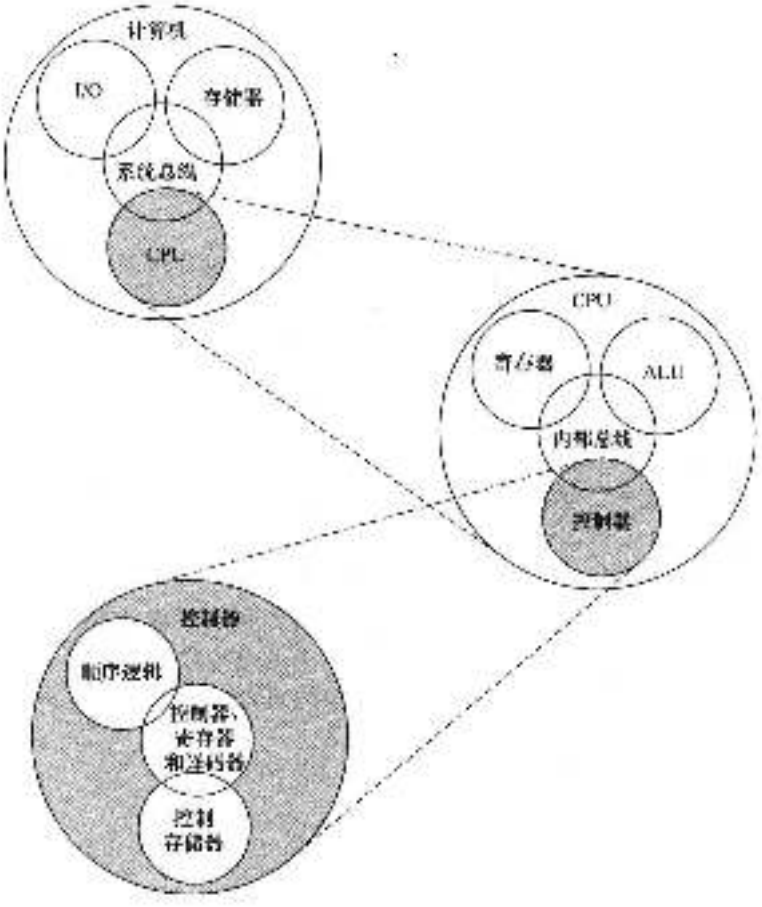


图 1-4 计算机：顶层结构

1.3 关键词和思考题

关键词

arithmetic and logic unit（ALU）：算术逻辑单元
central processing unit（CPU）：中央处理单元，中央处理器
computer architecture：计算机体系结构，计算机系统结构
computer organization：计算机组成，计算机组织
control unit：控制器，控制单元
input-output（I/O）：输入/输出
main memory：主存
processor：处理器
registers：寄存器
system bus：系统总线

思考题

1.1 计算机组成与计算机体系结构在概念上有何区别？
1.2 计算机结构与计算机功能在概念上有何区别？
1.3 计算机的四个主要功能是什么？
1.4 列出并概要定义计算机的主要结构部件。
1.5 列出并概要定义处理器的主要结构部件。

第2章

Computer Organization and Architecture: Designing for Performance, 8E

计算机的演变和性能

本章要点

- 计算机的发展历史主要由提高处理器速度、减小部件尺寸、增大存储容量、加快I/O能力和速度来表征。
- 导致处理器速度大幅度提高的一个关键因素是，缩减微处理器部件的尺寸，即减少部件间的距离，从而提高速度。然而近年来在速度上的真正增益却是来自于处理器的组成，这包括流水线处理技术和并行执行技术的大量使用，以及推测执行技术的使用（推测执行技术可使将来可能需要的指令提前探测性的执行）。所有这些技术的出发点是，最大限度地使处理器保持运行状态。
- 计算机系统设计的一个关键问题是在各元器件之间的性能平衡，以便在一个领域内所获得的性能增益不被另一领域的滞后所妨碍。特别是，处理器速度的提高远超出存储器存储速度的提高，因此，包括高速缓存技术、从存储器到处理器的更宽的数据路径和更智能的存储芯片等多种技术，用来补偿这种平衡的失调。

本章首先介绍计算机的发展简史，这一历史不但其本身令人感兴趣，而且也是为了概述计算机的结构和功能。然后，讨论性能问题。在贯穿本书的介绍过程中，将一直坚持"计算机资源的平衡利用"的思想。最后，将简要介绍贯穿本书的两个关键例子：Intel x86 和 ARM 系列处理器。

2.1 计算机简史

2.1.1 第一代：真空管

1. ENIAC

由宾夕法尼亚大学设计并建造的 ENIAC（Electronic Numerical Integrator And Computer，电子数字积分器和计算器）是世界上第一台通用电子数字计算机。这个项目是为了满足美国在第二次世界大战时的需要，当时军队的弹道研究实验室（BRL），一个负责开发新式武器的射程和弹道表的机构，在提供数据表的精确性和及时性上遇到了困难。没有这些发射表，新式武器和火炮对炮手来说是没有用处的。BRL 雇用了 200 多人，使用桌面计算器来求解所需要的火炮公式。为单件武器提供数据表将耗费一个人几小时，甚至几天时间。

宾夕法尼亚大学电子工程系的约翰·莫克利教授和他的一个研究生约翰·埃克特，计划采用真空管建造一台通用计算机，用于满足 BRL 的应用需求。1943 年，这个计划被军方采纳，ENIAC 项目开始启动。最终建造出来的机器体积庞大，重约 30 吨，占地 1500 平方英尺，采用了 1.8 万多个真空管。它工作时，消耗了 140kW 的功率。但它比任何电子机械计算器要快许多，每秒能执行 5000 次加法。

ENIAC 是一台十进制（而非二进制）机器，也就是说，其数字是以十进制表示，而且算法也是以十进制完成的。其存储器包含 20 个累加器，每个都能保存一个 10 位十进制数。每一位数由 10 个真空管环来表示，在任何时候，仅有一个真空管处于 ON 状态，代表 10 个数字中的一个。ENIAC 的主要缺点是必须手动编程，一切都要通过设置开关和插拔电缆头来实现。

ENIAC 于 1946 年建成，因此未能在第二次世界大战中出力。ENIAC 的第一项任务是，完成一系列复杂的运算，以帮助判断氢弹的可行性。将 ENIAC 用于有别于最初建造它的目的，表明

了它的通用性。ENIAC 在 BRL 的管理下继续工作，直到 1995 年被拆除。

2. 冯·诺伊曼机

正如前面所提到的，为 ENIAC 输入和修改程序被其繁琐。如果程序能够以某种形式与数据一同存在于存储器中，编程的过程就可以简化。这样，计算机就可以通过在存储器中读取程序来获取指令，而且通过设置一部分存储器的值就可以编写和修改程序。

这个思想被称为“存储程序概念”，它归功于 ENIAC 的设计者们，特别是 ENIAC 项目顾问、数学家约翰·冯·诺伊曼。冯·诺伊曼在 1945 年的一份新型计算机 EDVAC（Electronic Discrete Variable Computer，电子离散变量自动计算机）的计划中首次公布了这一思想。几乎在同时，阿兰·图灵也提出了同样的构想。

1946 年，冯·诺伊曼和他的同事在普林斯顿高级研究院开始设计一种新的存储程序计算机。这种新型机器称为 IAS 计算机。虽然直到 1952 年仍未完成，但它却成为了后来通用计算机的原型。

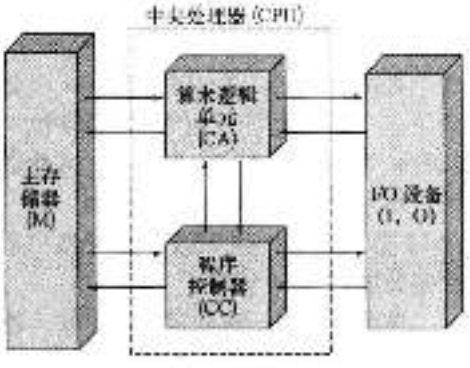


图 2-1 IAS 计算机结构

图 2-1 给出了 IAS 计算机的普遍结构（与图 1-4 的中间部分进行比较），它包括：

- 主存储器，用于存储数据和指令[⊖]。
- 能够处理二进制数的算术逻辑运算单元（ALU）。
- 控制器，负责解释内存中指令并执行之。
- 由控制器操纵的输入/输出设备（I/O）。

冯·诺伊曼早期的计划中概括了这种结构，在此值得引用下面的论述（见文献［VONN45］）：

2.2 第一：因为这台设备主要是一台计算机，所以它必须能够执行最频繁的基本算术运算，即加、减、乘、除运算。因此，它应该包含特殊的器件来执行这些操作。

必须看到，虽然这些原理可能是合理的，但是需要仔细研究实现它的特殊方式。无论如何，这一设备的中央算术（central arithmetical）部分必须存在。它组成了第 1 个特定的部分——CA。

2.3 第二：设备中控制操作顺序的逻辑控制部分，能够由中央控制器最有效的实现。如果此设备有灵活性，也就是说，几乎适用于所有应用，那么必须区分对待给定的指令和所给出的特殊问题，以保证这些指令（无论它是什么）都能被通用控制器执行。前者必须以某种形式存储，后者通过定义设备的操作部分来表示。中央控制（central control）仅指后者的功能。中央控制和实现它的器件组成了第 2 个特定部分——CC。

2.4 第三：任何执行长而复杂的操作序列（特别是计算序列）的设备都必须有一个相当大的存储器……

（b）管理一个复杂问题的指令集可能包含很多内容，特别是根据情况而编码（这存在于大多数情况中），这些素材必须被记忆。

无论如何，所有的存储器（memory）组成了设备的第 3 个特定的部分——M。

2.6 这 3 个特定的部分，CA、CC（一起被称为 C）和 M，对应于人类神经系统中的联想神经元。后面还需要讨论感觉和运动神经元的对应物，即设备的输入和输出器件。

设备必须具有接触某些特定媒体并进行输入和输出（感觉和运动）的能力。这种媒体被称为设备的外部记录媒体：R。

2.7 第四：设备必须有从 R 到特定的 C 和 M 传送信息的器件。这类器件形成了它的输入（input），因此第 4 部分是 I。我们将看见，最好使得所有的传送都是从 R（通过 I）到 M，而绝非直接来自 C。

⊖ 在本书中，除非特别声明，术语“指令”是指机器指令，与高级语言（例如 Ada 或 C++）中的语句不同。它由处理器直接解释和执行。而高级语言中的语句在被执行之前，必须首先被编译成一系列机器指令。

2.8 第五：设备必须有从它的特定部分 C 和 M 传送信息到 R 的器件。这些器件形成它的输出（output），因此第 5 部分是 O。我们还将看见，最好使得所有的传送都从 M（通过 O）至 R，而绝非直接来自 C。

除了少数特例外，当今所有计算机都具有与上述相同的结构和功能，因此它们都被称为"冯·诺伊曼机"。所以值得在这里简单描述一下 IAS 计算机的操作（参见［BURK46］）。依据文献［HAYE98］，冯·诺伊曼术语和概念变得更贴近当代的用法。伴随这些讨论的示例也基于文献［HAYE98］。

IAS 的存储器包含 1000 个存储单元，它们被称为字（word），每个字有 40 位（bit）[⊖]。数据和指令都存储在此。数据被表示为二进制形式，而且指令是二进制编码。图 2-2 给出了这种格式。每个数被表示为 1 个符号位和 39 个数值位。一个字也可以包含两条 20 位的指令，每条指令包含一个 8 位的操作码，用来指定所执行的操作类型，和一个 12 位的地址，用于指定存储器中某个字的地址（0～999）。

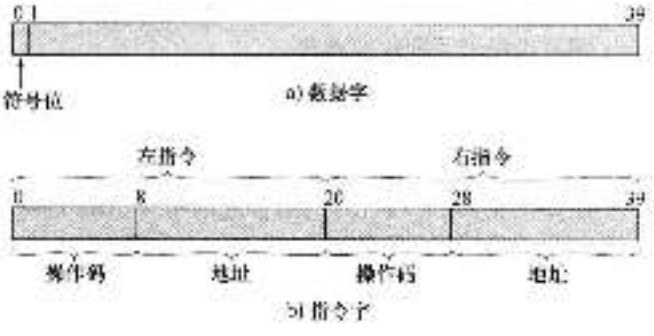


图 2-2 IAS 存储格式

控制器通过一次从存储器中取一条指令并执行它的方式来操作 IAS。为了解释这一过程，需要一张更详细的结构图，如图 2-3 所示。此图表明无论是控制器还是 ALU 都包含了存储区域，它们被称为寄存器（register），具体定义如下：

- **存储器缓冲寄存器**（MBR）：包含将要写到存储器或送到 I/O 单元或者接受来自存储器或 I/O 单元的一个字。
- **存储器地址寄存器**（MAR）：指定将要从 MBR 写进存储器或从存储器读入 MBR 的存储器字单元的地址。
- **指令寄存器**（IR）：包含正在执行的 8 位操作码指令。
- **指令缓冲寄存器**（IBR）：用来暂存来自存储器一个字的右边指令。
- **程序计数器**（PC）：存放将要从存储器中获取的下一对指令的地址。
- **累加器**（AC）和**乘商寄存器**（MQ）：用来暂存 ALU 运算的操作数和结果。例如，两个 40 位的数相乘，结果是一个 80 位的数，则高 40 位放在 AC 中，低 40 位放在 MQ 中。

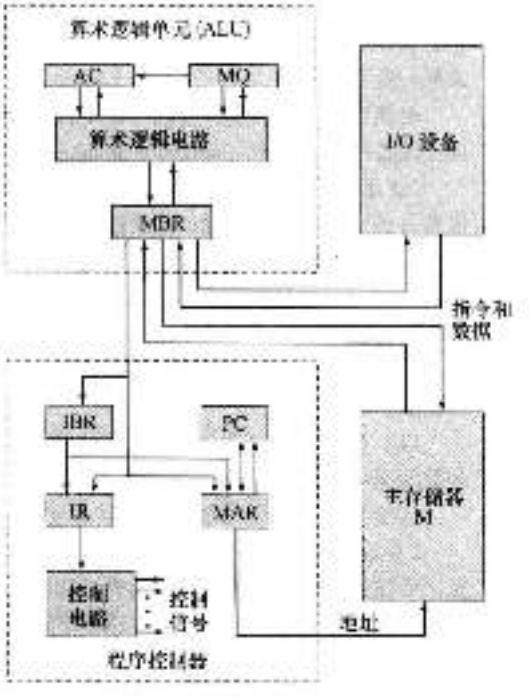


图 2-3 IAS 计算机结构扩展

IAS 通过反复执行指令周期（instruction cycle）来运行，如图 2-4 所示。每个指令周期由两

⊖ 术语"字"没有统一定义。通常，一个字被定义为一组有序字节或位，它是某一给定计算机中可以存储、传送或处理的信息基本单位。特别是，如果一个处理器具有固定长指令集，那么其指令长等于其字长。

个子周期组成。在取指周期（fetch cycle）中，下一条指令的操作码装入 IR，地址部分装入 MAR。指令可以从 IBR 中获得，也可以从存储器中获得，即先从存储器装在一个字到 MBR，然后将该字解开放入 IBR、IR 和 MAR。

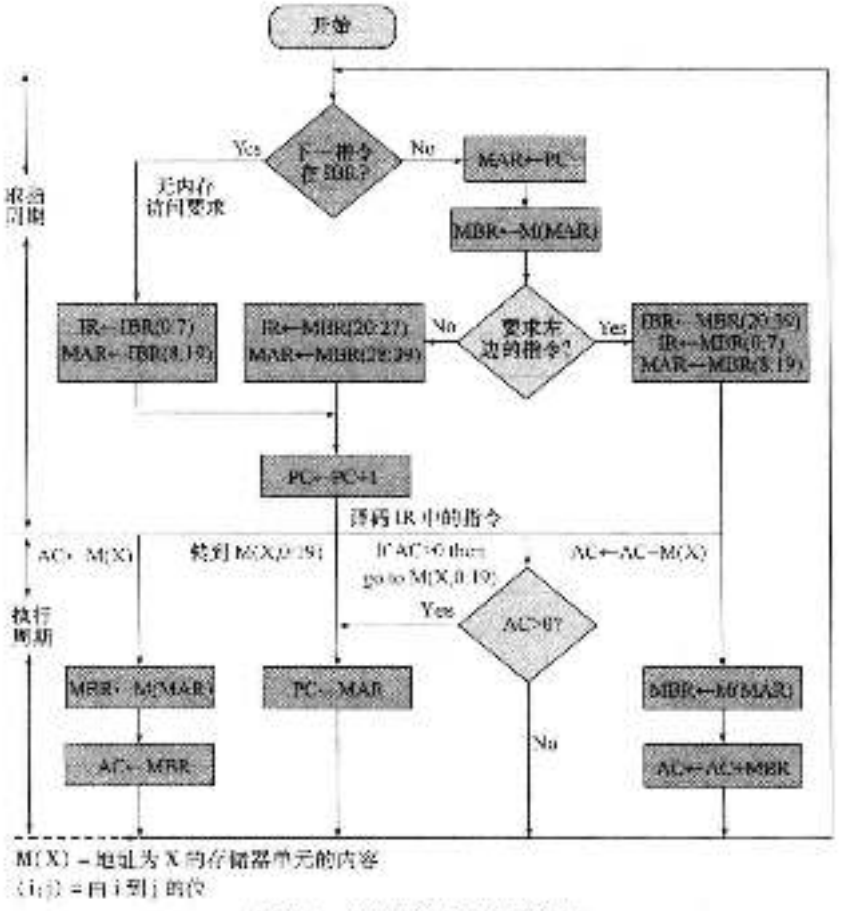


图 2-4　IAS 操作控制分流程图

为什么不直接获取？因为这些操作都是由电子电路控制并且导致数据路径的使用。为了简化电路，只用一个寄存器来指定存储器中读写的地址，而且也只用一个寄存器来存放数据源或目标。

一旦操作码在 IR 中，则进入执行周期（execute cycle）。控制电路翻译操作码，并且通过发送相应的控制信号来执行指令，这些信号控制数据的传送和 ALU 操作的执行。

IAS 计算机共有 21 条指令，在表 2-1 中列出，可分为以下几类：

- **数据传送**：在存储器和 ALU 的寄存器之间或在两个 ALU 寄存器之间传送数据。
- **无条件转移**：通常，控制器按顺序执行存储器中的指令，但这一顺序能通过跳转指令加以改变，以便执行重复的操作。
- **条件转移**：可以依据条件来决定是否跳转，从而选择从何处跳转。
- **算术运算**：由 ALU 完成的操作。
- **地址修改**：允许在 ALU 中计算地址，并将它插入存储器的指令中，为程序寻址带来很大的灵活性。

表 2-1 以符号化的、易读的形式列出了指令。事实上，每条指令必须遵循图 2-2b 的格式。操作码部分（前 8 位）指定了将要执行的 21 条指令。地址部分（剩余的 12 位）指定了执行指令所涉及的 1000 个存储单元的某一个。

表 2-1 IAS 指令集

指令类型	操作码	代表符号	说 明
数据传输	00001010	LOAD MQ	传送寄存器 MQ 的内容到累加器 AC
	00001001	LOAD MQ,M(X)	传送内存单元 X 的内容到 MQ
	00100001	STOR M(X)	传送累加器的内容到内存单元 X
	00000001	LOAD M(X)	传送 M(X)到累加器
	00000010	LOAD -M(X)	传送-M(X)到累加器
	00000011	LOAD | M(X) |	传送 M(X)的绝对值到累加器
	00000100	LOAD - | M(X) |	传送-| M(X) | 到累加器
无条件转移	00001101	JUMP M(X,0:19)	从 M(X)的左半字取下一条指令
	00001110	JUMP M(X,20:39)	从 M(X)的右半字取下一条指令
条件转移	00001111	JUMP + M(X,0:19)	如果累加器的值非负，则从 M(X)的左半字取下一条指令
	00010000	JUMP + M(X,20:39)	如果累加器的值非负，则从 M(X)的右半字取下一条指令
算术运算	00000101	ADD M(X)	M(X)加 AC，结果放入 AC
	00000111	ADD | M(X) |	| M(X) | 加 AC，结果放入 AC
	00000110	SUB M(X)	从 AC 中减 M(X)，结果放入 AC
	00000100	SUB | M(X) |	从 AC 中减 | M(X) |，结果放入 AC
	00001011	MUL M(X)	M(X)乘 MQ，结果的高位部分存入 AC，低位部分存入 MQ
	00001100	DIV M(X)	AC 除以 M(X)，商放入 MQ，余数放入 AC
	00010100	LSH	累加器内数乘以 2，即累加器左移一位
	00010101	RSH	累加器内数除以 2，即累加器右移一位
地址修改	00010010	STOR M(X,8:19)	用 AC 最右边的 12 位来替代 M(X)的左地址区段
	00010011	STOR M(X,28:39)	用 AC 最右边的 12 位来替代 M(X)的右地址区段

图 2-4 给出控制器执行指令的几个例子。注意，每个操作都要求用几步来完成，其中有些是相当精巧的。乘法运算需要 39 个子操作，除符号位外，每个位对应一个子操作。

3. 商用计算机

20 世纪 50 年代，有两家公司（Sperry 和 IBM）见证了计算机工业的诞生，当时这两家公司控制着计算机市场。

1947 年，埃克特和莫克利创办了埃克特-莫克利计算机公司，制造商用计算机。他们的第一个成功机型是 UNIVAC Ⅰ（Universal Automatic Computer，通用自动化计算机），它由美国统计局委托制造，用于 1950 年的计算。埃克特-莫克利计算机公司后来成为 Sperry-Rand 公司 UNIVAC 部门的一部分，继续生产了一系列后续的机型。

UNIVAC Ⅰ是第一台成功的商用计算机。顾名思义，它适合于科学计算和商业两方面的应用。作为它能够完成的任务示范，介绍该系统的第一篇论文列出了矩阵代数运算、统计问题、人寿保险公司的养老金清单和数理逻辑问题。

UNIVAC Ⅱ比 UNIVAC Ⅰ有更大的内存容量和更高的性能，它诞生于 20 世纪 50 年代后期，并且印证了计算机工业一直保持的几个特征。首先，先进的技术使公司能不断地推出更大的、功能更强的计算机；其次，每个公司尽量使它们的新机型向上兼容[⊖]旧的机型，这意味着在旧机型上

⊖ 也称为向下兼容，从较早系统的观点来看，同样的概念被称为向上兼容或向前兼容。

编写的程序可以在新机型上运行。采用这个战略是为了留住客户，换句话说，当一位顾客决定购买一台新机器时，为了避免程序投资的损失，他很有可能购买同一家公司的产品。

UNIVAC 部门也开发 1100 系列计算机，这成为它的主要业务。这一系列计算机表明了那时存在的一个区别。第一个型号 UNIVAC 1103 以及多年以后的后续产品，主要致力于科学应用，包括长且复杂的计算；而其他公司则致力于商业应用，包含对大量文本数据的处理。现在，这一区别已经消失，但它曾经存在了许多年。

IBM 那时还只是穿孔卡处理设备的主要制造商。IBM 于 1953 年生产了自己的首台程序存储计算机 IBM701，它主要面向科学应用［BASH81］。1955 年 IBM 生产了另一种产品 IBM 702，其硬件上许多特点使它适用于商业应用，它们是 700/7000 这一大系列的开端产品，这一系列使 IBM 成为占据主导地位的计算机制造商。

2.1.2　第二代：晶体管

电子计算机的第一个主要改变是使用晶体管代替电子管。晶体管比电子管的体积更小、更便宜、发热更少，而且能以与电子管相同的方式建造计算机。电子管由导线、金属片、玻璃外壳和真空管构成；而晶体管是固态器件，由硅片制成。

1947 年贝尔实验室发明了晶体管，从而在 20 世纪 50 年代引发了一场电子革命。直到 20 世纪 50 年代末，完全晶体管计算机才能实际应用于商业。IBM 这次仍不是第一个提供这种技术的公司。NCR 和 RCA 是生产小型晶体管机器的先驱者，两者相比，RCA 更成功一些。IBM 不久之后才开发了 7000 系列。

晶体管的使用是第二代计算机的标志。基于所采用的基本硬件技术来划分各个计算机时代，已成为人们的一个共识（如表 2-2 所示）。新一代计算机以比旧一代机器具有更快的处理速度、更大的存储容量和更小的尺寸。

表 2-2　计算机的发展阶段

发展阶段	大致时间	技术	典型速度（每秒钟的操作次数）
1	1946 – 1957 年	真空管（电子管）	4 万次
2	1958 – 1964 年	晶体管	20 万次
3	1965 – 1971 年	小规模和中规模集成电路	100 万次
4	1972 – 1977 年	大规模集成电路	1000 万次
5	1978 – 1991 年	超大规模集成电路	1 亿次
6	1991 年以后	甚大规模集成电路	10 亿次

这个时期，还发生了其他变化。第二代计算机采用了更复杂的算术逻辑单元和控制器，使用了高级编程语言，并为计算机提供了系统软件。

值得注意的第二个主要改变是，这个时期出现了数据设备公司（DEC）。DEC 建立于 1957 年，同年，DEC 开发了它们的第一台计算机——PDP-1。这种系列的计算机以及由其开发的小型计算机，使这家公司在第二代计算机中变得异常著名。

IBM 7094

从 1952 年出现 700 系列到 1964 年 7000 系列的最后一个产品，IBM 的产品经历了计算机产品的典型演变历程。每个产品的后续产品都有更高的性能、更大的容量和/或者更低的价格。

表 2-3 说明了这一趋势。主存容量是 2^n 个 36 位字的倍数，从 2K（$1K = 2^{10}$）字增加到 32K

字[⊖]。访问内存中一个字的时间，称为内存周期，从30μs下降到1.4μs。操作码的数量也从可怜的24种增加到185种。

最后一栏表示中央控制单元（CPU）的相对执行速度。速度的提高是因为电子元件的改进（例如，晶体管比电子管速度更快）和更复杂的电路。例如，IBM 7049包含指令缓存寄存器（IBR），它用来暂存下一条指令。控制器在每个指令的取指周期从存储器取相邻的两个字，除非遇到不常发生的分支指令，这意味着控制器只需在一半的指令周期才需要访问内存取指令。这一预取指令方式显著减少了平均指令周期时间。

表2-3的其余几列将会随着文章的深入的进一步清晰。

图2-5表示了IBM 7049的大型配置环境（带有许多外设），这是第二代计算机的代表（见[BELL71]）。IBM 7049与IAS计算机的区别值得特别注意，最重要的是它使用了数据通道（data channel）。数据通道是独立的I/O模块，具有自己的处理器和指令集。在带有这些设备的计算机系统中，CPU不执行具体的I/O指令。这些指令被存放在主存中，由数据通道本身的专用处理器执行。CPU通过给数据通道发送一个控制信号来初始化I/O传送，指示它执行内存中的一串指令。数据通道独立地执行它的任务，并在操作完成时通知CPU。这种方式减轻了CPU很多的处理负担。

表2-3 IBM 700/7000系列成员的例子

型号	首次推出时间	CPU技术	存储器技术	周期时间（μs）	存储器容量（K）	操作码数目	变址寄存器数目	浮点	I/O单元（通道）	取指令重叠	速度（相对于701）
701	1953	电子管	静电管	30	2～4	24	0	否	否	否	1
704	1955	真空管	磁芯	12	4～32	80	3	是	否	否	2.5
709	1958	真空管	磁心	12	32	140	3	是	是	否	4
7090	1960	晶体管	磁芯	2.18	32	169	3	是	是	否	25
7094 I	1962	晶体管	磁芯	2	32	185	7	（双精度）是	是	是	30
7094 II	1964	晶体管	磁芯	1.4	32	185	7	（双精度）是	是	是	50

另一个新特点是多路（复用）器（multiplexor），它是数据通道、CPU、内存的中心连接点。多路器调度CPU和数据通道对内存的访问，允许这些设备独立运行。

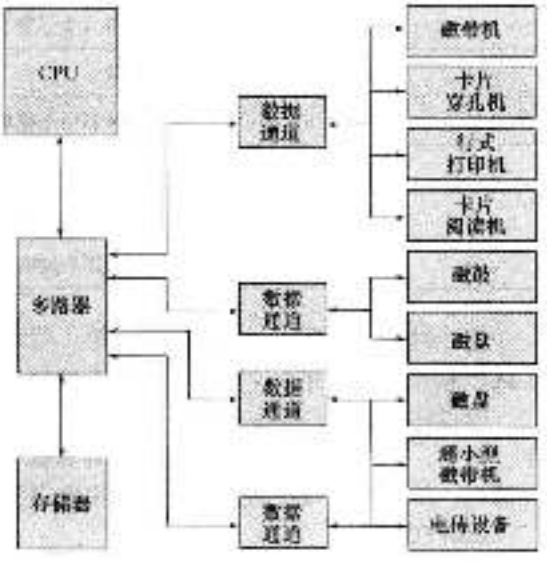


图2-5 IBM 7049配置

2.1.3 第三代：集成电路

单个独立封装的晶体管称为分立元件（discrete component），从20世纪50年代到60年代早期，电子设备包含大量的分立元件——晶体管、电阻、电容等。分立元件独立制造，封装在自己的容器中，然后一起焊接到或连接到纤维板（类似电路板）上。最后再安装到计算机、示波器或其他电子设备中。在电子设备需要晶体管的地方，一个包含针尖大小硅片的小晶体管就会被焊接到电路板上。从晶体管到电路板的整个制造过程都是昂贵且麻烦的。

⊖ 数字前缀，例如千（K）和千兆（G）的使用讨论包含在计算机科学学生资源站点的支持材料中，网址为WilliamStallings.com/StudentSupport.html。

这种情况给计算机工业带来了问题。早期的第二代计算机包含约 10 000 个晶体管，这一数字后来增长到数十万，使得生产更新、更强大的计算机变得越来越困难。

1958 年出现了电子学革命性的成就，开创了微电子时代：集成电路的发明。正是集成电路定义了第三代计算机。这一节将简单介绍集成电路技术，然后，考察可能是第三代计算机中最重要的两个成员：IBM 的 System/360 和 DEC 的 PDP-8，两者都在这个时代的初期出现。

1. 微电子技术

顾名思义，微电子技术是“微小的电子技术”。从数字电子技术和计算机工业一开始，就存在持续不断地减小数字电子电路尺寸的趋势。在考察这一趋势的内涵和好处之前，我们需要先介绍数字电子技术的一些性质，而更详细的讨论在第 20 章。

如我们所知，电子计算机的基本器件必须执行存储、传送、处理、控制等功能。只有两种基本类型的元件是必需的：逻辑门和存储器位元（如图 2-6 所示）。逻辑门是实现简单布尔或逻辑功能的元件，例如，“如果 A 和 B 是真，那么 C 是真（与门）”。由于它们控制数据流的方式与闸门相似，因此这种设备被称为逻辑门。

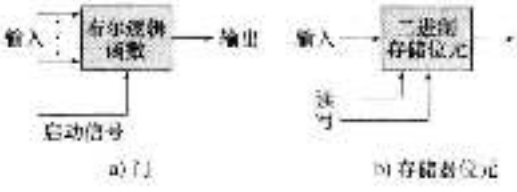


图 2-6 基本的计算机元件

存储器位元是一个能够存储一位数据的元件，也就是说，该元件在任何时刻都可以处于两个稳定状态之一。将大量的基本元件连接起来，就能够建造一台计算机。我们可以将此与如下 4 个基本功能联系起来：

- **数据存储**：由存储器位元提供。
- **数据处理**：由逻辑门电路提供。
- **数据移动**：部件间的通路用于将数据从内存传送到内存，或从内存通过门电路再传送到内存。
- **控制**：部件间的通路传送控制信号。例如，一个门有一或两个数据输入和激活启动门的控制信号输入。当控制信号是 ON 时，逻辑门对数据输入执行其功能并产生数据输出。类似地，存储器位元在写控制信号为 ON 时，存入其输入线的位值；而在读控制信号为 ON 时，将位元的位值放置在其输出线上。

因此，计算机包含门、存储器位元和它们之间的互连。而这些门和存储器位元是由简单的数字电子元件构造的。

集成电路利用了一个事实，即晶体管、电阻、导线都可以用硅之类的半导体制成。将整个电路安装在很小的硅片上而不是用分立元件搭成的等价电路，只不过是“固态技术”的一种扩展，而且在一块硅晶片上能同时制造很多个晶体管。同样重要的是，这些晶体管能够通过金属化过程相互连接，以形成电路。

图 2-7 描述了集成电路的关键性概念。一块薄硅晶片（wafer）划分由多个小区域排列而成的阵列，每个区域有几平方毫米，它们上面都有相同的电路。这块晶片被划分成许多块芯片（chip），每块芯片都包含许多逻辑门和/或存储器位元以及许多输入、输出连接点。然后封装这块芯片，使之得以保护，并加上引脚，用以连接芯片外部的其他设备。许多这样的集成电路块可以连接在印刷电路板上，产生更大、更复杂的电路。

起初，只有几个门和存储器位元可以可靠地制造并封装在一起。这些早期的集成电路被称为小规模集成电路（SSI）。随着时间的推移，将越来越多的元件放在同一块芯片上成为可能。图 2-8 显示了密度的增长，它是所有曾经记录的最显著技术趋势之一[⊖]。此图反映了著名的摩尔定

⊖ 注意纵轴使用了对数（log）刻度。关于对数刻度的基本回顾包含在计算机科学学生资源站点的数学复习资料中，网址为 WilliamStallings.com/StudentSupport.html。

律，该定律是 Intel 合伙创办人之一高登·摩尔（Gordon Moore）于 1965 年提出的（参见文献[MOOR65]）。摩尔观察到单芯片上所能包含的晶体管数量每年翻一番，并正确断言这种态势在不远的将来还会继续下去。令许多人（包括摩尔在内）惊奇的是，这种态势年复一年地持续了下来。直到 1970 年，这种态势就变成每 18 个月翻一番，但从此之后，这个新增长速率又持续下来。

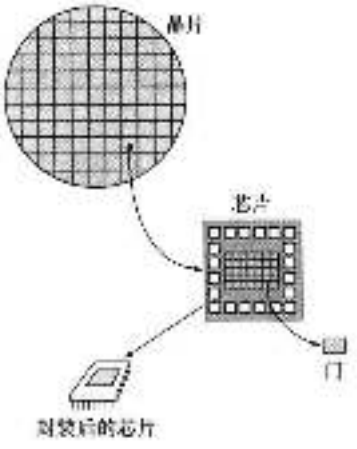


图 2-7 晶片、芯片和逻辑门之间的相互关系

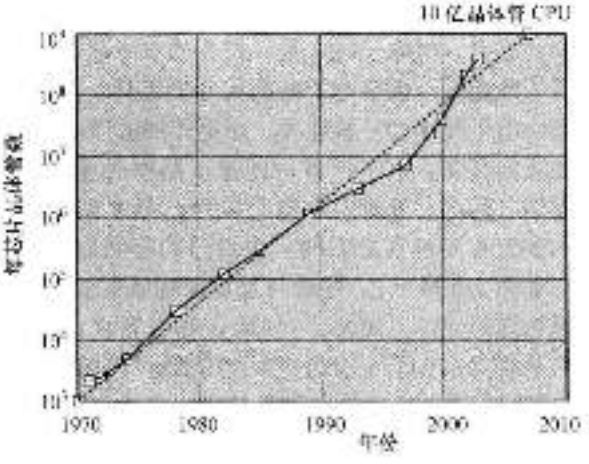


图 2-8 CPU 晶体管数量的增长（参见[BOHR03]）

摩尔定律的影响是深远的：

（1）在芯片集成度快速增长的期间，单个芯片的成本几乎没有变化。这意味着计算机逻辑电路和存储器电路的成本显著下降。

（2）因为在集成度更高的芯片中逻辑和存储器单元的位置更靠近，电子线路长度更短，所以提高了工作速度。

（3）计算机变得更小，更容易放置在各种环境中。

（4）减小了电能消耗及对冷却的要求。

（5）集成电路内部的连接比焊接更可靠。由于芯片上电路的增加，芯片间的连接变得更少。

2. IMB System/360

到 1964 年，IBM 的 7000 系列机已经牢牢占据了市场。那年，IBM 发布了 System/360，这是计算机产品的一个新家族。虽然发布消息本身并不惊人，但它包括了一些令现有 IBM 客户感到不太舒服的信息：360 产品线与原来的 IBM 机器不兼容。因此，现有的客户转移到 360 将会遇到困难。这是 IBM 所迈出的大胆一步，IBM 认为有必要突破 7000 体系结构的限制，制造能随着新的集成电路技术一起发展的系统[PADE81，GIFF87]。这一战略在经济上和技术上都会得到回报。360 在那时获得了成功，巩固了 IBM 作为计算机销售的绝对优势地位，IBM 占有 70% 以上的市场份额。经过相应的修改和扩展，360 体系结构至今仍是 IBM 大型机[⊖]的体系结构，本书将一直采用此结构作为范例。

System/360 系列是工业上第一个计算机系列，该系列机覆盖的性能和价格范围很广。表 2-4 列出了 1965 年各种型号机器的一些关键特征（系列机中的不同成员有不同型号的指令）。各种型号之间的兼容性体现在，为系列机中某个型号写的程序可以在系列机中任一个型号上运行，只是运行时间不同。

⊖ 术语“大型机”（mainframe）是指除超级计算机之外的更多、更强的计算机。大型机的典型特征是，支持大型数据库、具有精细的 I/O 设备和用于中央数据处理设备中。

表 2-4　System/360 系列机的关键特征

特点	Model 30	Model 40	Model 50	Model 60	Model 70
最大存储器容量（字节）	64K	256K	256K	512K	512K
存储器的数据传输率（MB/s）	0.5	0.8	2.0	8.0	16.0
处理器的时钟周期（μs）	1.0	0.625	0.5	0.25	0.2
相对速度	1	3.5	10	21	50
数据通道的最大数目	3	3	4	6	6
单通道最大数据传输率（KB/s）	250	400	800	1250	1250

兼容计算机的系列化概念不但新颖而且取得了成功。用户起初可以购买较便宜的 Model 30 来满足不高的要求和预算。之后，如果用户的需要提高了，可以将它升级到内存更多、速度更快的机器上，而不必对已经开发的软件再付出投资。IBM 系列机具有下列特征：

- **相同的或相似的指令集**：多数情况下，系列机中的所有成员都具有完全相同的机器指令集。这样，能够在一台机器上执行的程序同样也能在另一台机器上执行。某些情况下，系列机中低端产品的指令集是高端产品的一个子集。这意味着程序可以向上移植而不能向下移植。
- **相似或相同的操作系统**：系列机中的所有成员都有相同的基本操作系统，在某些情况下，高端成员会增添一些新特征。
- **更高的速度**：系列机中从低端成员到高端成员，其指令执行速度逐渐增加。
- **更多的 I/O 端口数**：系列机中从低端成员到高端成员，其 I/O 端口数越来越多。
- **更大的内存容量**：系列机中从低端成员到高端成员，其主存容量越来越大。
- **更高的成本**：在某一时间段，系列机中从低端成员到高端成员，其成本越来越高。

系列机的概念如何实现？区别在于 3 个因素：基本速度、容量和并行程度［STEV64］。例如，通过在 ALU 中使用更复杂的电路，允许子操作并行地执行，使得某一给定指令的执行速度更快。另一个提高速度的方式是增加主存与 CPU 之间的数据宽度，Model 30 每次只能从主存中读取一个字节（8 位），而 Model 75 每次能够读取 8 个字节。

System/360 不仅指明了 IBM 未来发展的方向，而且对整个计算机行业有着深远的影响，其许多特点已成为其他大型机的标准。

3. DEC 公司的 PDP-8

在 IBM 交付其第一个 System/360 产品的同一年，还出现了另一重要的产品——数字装备公司（DEC）的 PDP-8。在多数计算机仍需要空调环境的时候，PDP-8（在业界称为小型机，在迷你短裙出现之后）足够小，以至于可以放在实验室的工作台上，或被安装在其他系统中。这种小型机不能完成大型机的所有工作，但价格只要 1.6 万美元，这个价格足够便宜，使得每个实验室技术人员都能配备它。与之相比，几个月前出现的 System/360 系列大型机价格高达几十万美元。

PDP-8 的低价格、小尺寸使得其他制造商可以购买 PDP-8，并将它集成到自己的计算机系统中再行出售。这些其他制造商被称为原始设备制造商（OEM），并且 OEM 市场成为计算机市场的主要部分。

PDP-8 立即引起轰动，并使 DEC 大发其财。这种机器和 PDP 系列机的后续成员（如表 2-5 所示）达到了原来只有 IBM 计算机才能达到的状态，DEC 在此后的 12 年中销售了大约 5 万台计算机。正如 DEC 在自己的公司史中写到的，PDP-8 "创造了小型机的概念，并领导了数十亿美元的产业"。PDP-8 同样使 DEC 成为最大的小型机制造商。到 PDP-8 "退隐"时，DEC 已成为仅次于 IBM 的第二大计算机制造商。

表 2-5 PDP-8 的演变［VOEL88］

型号	最初面世时间（年/月）	处理器+4K 个 12 位字的存储器的价格（单位：1000 美元）	存储器的数据传输率（字/微秒）	体积（立方英尺）	革新和改进
PDP-8	1965/4	16.3	1.26	8.0	自动布线-包装生产
PDP-8/5	1966/9	8.79	0.08	3.2	实现口行指令
PDP-8/I	1968/4	11.6	1.34	8.0	中等规模集成电路
PDP-8/L	1968/11	7.0	1.26	2.0	更小的机箱
PDP-8/E	1971/2	4.99	1.52	2.2	Omnibus 总线
PDP-8/M	1972/6	3.69	1.52	1.8	与 8/E 相比，机箱容量减小一半，插槽更少
PDP-8/A	1975/1	2.6	1.34	1.2	半导体存储器，浮点处理器

与 IBM 在 700/7000 和 360 系列机中所采用的中央交换机结构（如图 2-5 所示）相比，PDP-8 的后续型号采用了一种现在几乎已被所有微型机所采用的结构，即总线结构，如图 2-9 所示。PDP-8 的总线被称为 Omnibus，包含 96 个独立的信号通路，用来传送控制信号、地址信号和数据信号。由于所有的系统元件都分享同一套信号通路，因此它们的使用必须由 CPU 来控制。这种结构具有高度的灵活性，允许将模块插入总线以形成各种配置。

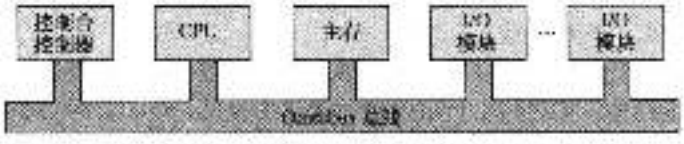


图 2-9 PDP-8 的总线结构

2.1.4 后续几代

如何定义计算机第三代以后的各代，意见不太一致。表 2-2 建议根据集成电路技术的发展来划分以后几代。随着大规模集成电路（LSI）的采用，一块集成电路芯片上可以放置 1000 个元件，超大规模集成（VLSI）达到每个芯片 1 万个元件，而现在的巨大规模（ULSI）芯片集成了超过 100 万个元件。

随着技术的高速发展，产品的迅速更新，软件和通信变得和硬件同等重要。各代之间的界限变得越来越模糊且没有太大意义。可以说新发展的商业应用导致了 20 世纪 70 年代的主要变化，这些变化的结果仍在发挥作用。本节介绍其中两个最重要的结果。

1. 半导体存储器

集成电路技术在计算机中的最初应用是采用集成电路芯片来制作处理器（控制单元和算术逻辑单元），但人们同时还发现这一技术也可以构造存储器。

20 世纪 50 年代和 60 年代，所有计算机存储器都是由微小的铁磁体环做成，每个直径约 1/16 英寸。这些小环被吊在计算机内用细线做成的网格上。一个环（或称为磁芯）的一种磁化方向代表 1；另一个磁化方向则代表一个 0。磁芯存储器速度相当快，读存储器速度中的一位只需百万分之一秒。但是磁芯的价格昂贵，体积大，而且读出是破坏性的；对磁芯的一次读取会擦除其存储的数据。因此必须安装读出后立即恢复数据的电路。

1970 年，Fairchild（仙童公司）生产了第一个容量较大的半导体存储器。一块相当于单个磁芯大小的芯片，包含了 256 位的内存。这种内存芯片是非破坏性的，而且读写速度比磁芯快很多。读取一位只需要 70ns，但其每位的价格比磁芯的要贵。

1974 年，一件关键性的事情发生了：半导体存储器的每位价格低于磁芯存储器的每位价格。

这以后，存储器的价格持续快速下跌，但物理存储密度不断增加。这导致了新的机器比几年前的机器更小、更快、内存容量更大而且价格更便宜。存储器技术的发展，与将要讨论的处理器技术的发展一起，在不到10年的时间里改变了计算机的生命力。虽然庞大、昂贵的计算机仍然存在，但计算机已经以办公设备和个人电脑的方式走向了"最终用户"。

自从1970年起，半导体存储器经历了13代：单个芯片1K、4K、16K、64K、256K、1M、4M、16M、64M、256M、1G、4G和现在的16Gb（$1K=2^{10}$，$1M=2^{20}$，$1G=2^{30}$）。每一代比其前一代存储密度提高4倍，而每位价格和存取时间都在下降。

2. 微处理器

同存储器芯片一样，处理器芯片的单元密度也在不断地增加。随着时间的推移，越来越多的单元被放置在每一块芯片上，因此，构建一个计算机的处理器所需要的芯片越来越少。

1971年出现了突破，Intel开发了Intel 4004。4004是第一个将CPU的所有元件都放在同一块芯片内的产品　于是，微处理器诞生了。

4004能完成两个4位数相加，通过重复相加来完成乘法。以今天的标准，4004虽然相对简单，但是它却成为微处理器的能力和功能不断发展的开端。

这一演变从处理器一次所能处理的位数就可能容易地看出。虽然没有明确的衡量方法，但最好的衡量方法也许就是数据总线的宽度：处理器能够一次同时输入或输出的数据位数。另一种衡量方法是看累加器或通用寄存器组的数据位数。虽然有时两种方法得出的结构雷同，但并非总是如此。例如，有许多微处理器的寄存器位数是16位，但其数据总线却只有8位。

微处理器演变中另一个主要进步是1972年出现的Intel 8008，这是第一个8位的微处理器，它几乎比4004复杂一倍。

这两种产品都没有下面的这一事件影响深远：1974年出现了Intel 8080，这是第一个通用微处理器。4004和8008是为特殊用途而设计的，而8080是为通用微机设计的中央处理器。它与8008一样，都是8位微处理器，但8080更快、有丰富的指令集且有更强的寻址能力。

大约在同时，16位微处理器被开发出来，但直到20世纪70年代末才出现强大的通用16位微处理器，8086便是其中之一。这一发展趋势中的另一阶段是在1981年，贝尔实验室和HP公司开发出了32位单片微处理器。Intel于1985年推出了其32位微处理器Intel 80386（如表2-6所示）。

表2-6 Intel微处理器的演变

a）20世纪70年代的处理器

	4004	8008	8080	8086	8088
发布时间	1971年	1972年	1974年	1978年	1979年
时钟速度	108kHz	108kHz	2MHz	5MHz、8MHz、10MHz	5MHz、8MHz
总线宽度	4位	8位	8位	16位	8位
晶体管数量	2300	3500	6000	29 000	29 000
特征尺寸（微米）	10		6	3	6
可寻址存储器	640字节	16KB	64KB	1MB	1MB

b）20世纪80年代的处理器

	80286	386TM DX	386TM SX	486TM DX CPU
发布时间	1982年	1985年	1988年	1989年
时钟速度	6～12.5MHz	16～33MHz	16～33MHz	25～50MHz
总线宽度	16位	32位	16位	32位
晶体管数量	134 000	275 000	275 000	1 200 000
特征尺寸（微米）	1.5	1	1	0.8～1

（续）

b）20 世纪 80 年代的处理器

	80286	386TM DX	386TM SX	486TM DX CPU
可寻址存储器	16MB	4GB	16MB	4GB
虚拟存储器	1GB	64TB	64TB	64TB
高速缓存	—	—	—	8kB

c）20 世纪 90 年代的处理器

	486TM SX	Pentium	Pentium Pro	Pentium II
发布时间	1991 年	1993 年	1995 年	1997 年
时钟速度	15～33MHz	60～166MHz	150～200MHz	200～300MHz
总线宽度	32 位	32 位	64 位	64 位
晶体管数量	1.185 百万	3.1 百万	5.5 百万	7.5 百万
特征尺寸（微米）	1	0.8	0.6	0.35
可寻址存储器	4GB	4GB	64GB	64GB
虚拟存储器	64TB	64TB	64TB	64TB
高速缓存	8kB	8kB	512kB L1 和 1MB L2	512kB L2

d）最近的处理器

	Pentium III	Pentium 4	Core 2Duo	Core 2Quad
发布时间	1999 年	2000 年	2006 年	2008 年
时钟速度	450～600MHz	1.3～1.8GHz	1.06～1.2GHz	3GHz
总线宽度	64 位	64 位	64 位	64 位
晶体管数量	9.5 百万	42 百万	167 百万	820 百万
特征尺寸（微米）	0.25	0.18	0.065	0.045
可寻址存储器	64GB	64GB	64GB	64GB
虚拟存储器	64TB	64TB	64TB	64TB
高速缓存	512kB L2	256kB L2	2MB L2	6MB L2

2.2 性能设计

计算机系统的价格都在逐年下降，而它们的性能和容量却在显著提高。在仓储商场，用不到 1000 美元就能买到与 10 年前的 IBM 大型机具有相同性能的个人电脑。因此，我们拥有几乎是“免费”的计算机功能，而这一持续不断的技术革命使开发极复杂和极高性能的应用成为可能。例如，今天基于微处理器系统的功能强大的桌面应用包括：图像处理、语音识别、视频会议、多媒体创作、文件的语音和视频注解、模拟建模。

工作站系统目前支持高度复杂的工程和科学应用以及模拟系统，并且具有支持图像和视频应用的能力。此外，工商业正依赖于功能强大的服务器来完成交易和进行数据库处理，并用它来支持大型客户/服务器网络，以替代昔日庞大的大型机的计算中心。

从计算机组成和结构的角度来看，发人深省的是：一方面，组成今天计算机奇迹的基本模块与 50 年前的 IAS 计算机基本相同；另一方面，从现有材料中挤出最后一丁点性能来技术都变得日益复杂。

这一观察结果是贯穿本书的指导原则。当考察计算机各个组成部件的时候，我们追求两个目标：第一，本书解释每个所考察领域的基本功能；第二，本书探寻为达到最大性能所需求的技术。本节的剩余部分将突出性能设计所涉及的关键因素。

2.2.1　微处理器的速度

Intel x86 处理器或 IBM 大型机如此震撼人心的强大功能来自处理器芯片制造商对速度的执着追求。这些机器的演变一直遵循前面介绍的摩尔定律。只要这个定律保持有效，芯片制造商就能每 3 年发布新一代的芯片，其晶体管数为上一代芯片的 4 倍。对于内存芯片，仍旧采用基本的主存储器技术，动态随机存储器（DRAM）的容量每 3 年提高 4 倍。对于微处理器，通过增加新的电路，减小电路间的距离来提高速度，使得性能每 3 年提高 4 ~ 5 倍，从 1987 年开始推出的 Intel x86 系列也是如此。

但是，除非以计算机指令的形式不断向它提供持续的工作流，否则微处理器将达不到它的潜在速度。任何阻碍工作流的事件都会降低处理器的功能。因此，当芯片制造商忙于研究怎样不断提高芯片集成度的同时，处理器的设计者必须提供更加复杂的技术来填饱这个怪物。当代处理器所包含的技术有：

- **转移预测**：处理器提前考察取自内存的指令代码，并预测哪条分支指令或哪组指令可能下一步将会被执行。如果处理器大部分时间的猜测是正确的，则它能预取正确的指令，并将它们放入缓存，这样处理器就会始终处于繁忙之中。这种预测策略的更复杂例子是不只预测下面一个分支，还要提前预测多条分支。如此，转移预测增加了可供处理器执行的工作量。
- **数据流分析**：处理器通过分析哪一条指令依赖其他指令的结果或数据，来优化指令调度。事实上，准备好的指令就可以被调度执行，不必按照原来程序的顺序，这减少了不必要的延时。
- **推测执行**：使用转移预测和数据流分析，一些处理器让指令在程序实际执行之前就“推测执行”，并将结果存储在暂时的空间。通过执行可能需要的指令，可以使处理器的执行机制尽可能地保持繁忙。

以上技术以及其他复杂的技术是实现处理器强大功能所必需的。它们使得充分利用处理器速度成为可能。

2.2.2　性能平衡

当处理器的性能以惊人的速度向前发展的时候，计算机的其他关键部件并没有跟上。这引发了寻求性能平衡的需要：调整组成和结构，以补偿各种部件之间的能力不匹配。

处理器和主存储器的接口问题是这些不匹配问题中最重要的。考虑图 2-10 中所描述的历史。随着处理器速度的快速增长，主存储器和处理器之间的数据传输率却严重滞后。处理器和主存储器之间的接口是整个计算机中最关键的通路，因为它负责在存储器芯片和处理器之间运送持续的程序指令和数据流。如果存储器或通路跟不上处理器持续不断的需求，处理器就会经常处于等待状态，宝贵的处理器时间就被浪费。

有许多系统结构的方法来解决这个问题，而所有这些方法又反映在当代计算机设计中。考虑如下一些例子：

- 通过使 DRAM 的接口“更宽”而不是“更深”，以及增大总线的数据宽度，来增加每次所能取出的位数。
- 通过在 DRAM 芯片中加入高速缓存[①]或其他缓冲机制来改变 DRAM 接口，使其更加有效。

---

① 高速缓存是插在大而慢的存储器与存取此存储器的处理器或机构之间插入的一个相对小而快的存储器。此高速缓存保存最近被访问过的数据，并被设计用来加快对同一数据的后续访问。高速缓存的讨论在第 4 章。

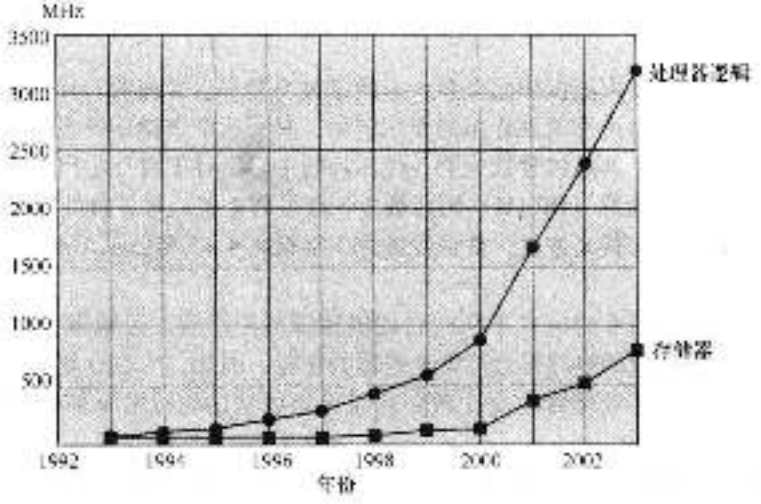


图 2-10 处理器逻辑和存储器的性能差距 [BOHR03]

- 通过在主存和处理器之间引入更复杂、更有效的高速缓存结构，来减少存储器访问频度。这包括在处理器芯片中加入一级或者多级高速缓存，以及在靠近处理器芯片的地方加入片外高速缓存。
- 通过使用高速总线和分层总线来缓冲和结构化数据流，从而增加处理器与存储器之间相互连接的带宽。

另一个设计焦点是 I/O 设备的处理。由于计算机变得更快、更强大，人们开发出更加复杂的应用来支持使用要求频繁 I/O 操作的外设。图 2-11 给出个人计算机和工作站使用的一些典型例子。这些设备要求很高的数据吞吐量。虽然目前的处理器能够处理这些设备输出的数据，但在处理器和外设之间传送数据仍存在着问题。这里的策略包括缓冲和暂存机制，以及使用高速互连和更为精巧的总线结构。此外，使用多处理器配置有助于满足 I/O 的需要。

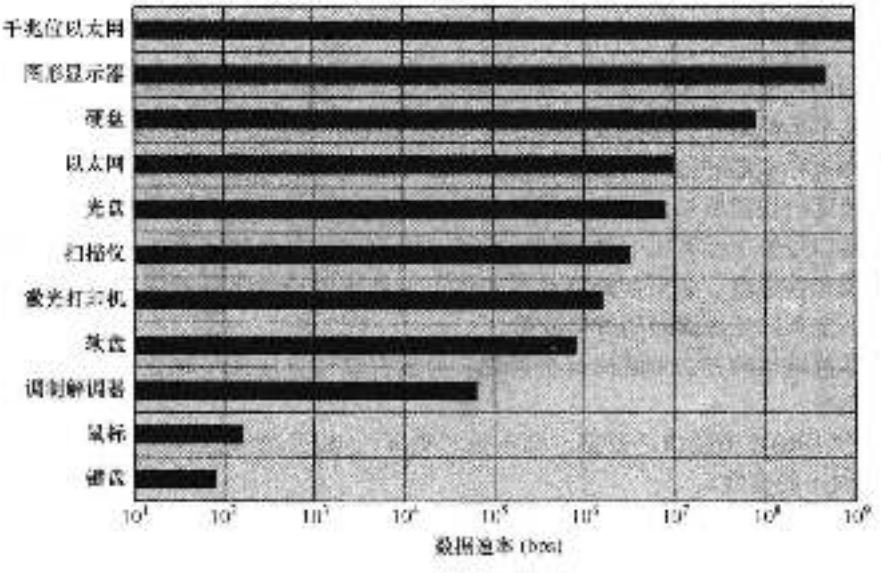


图 2-11 典型 I/O 设备的数据速率

所有问题的关键是平衡。设计者始终努力平衡处理器部件、主存储器、I/O 设备及互连结构的吞吐量和处理要求。设计必须不断更新，以应付两个始终变化的因素：

- 从一种类型的元器件到另一种类型的元器件，对于各种不同的技术领域（处理器、总线、存储器、外设），它们性能提高的速率相差很大。
- 新的应用和新的外围设备根据典型指令的描述和数据访问模式不断改变对系统特性的要求。

因此，计算机设计是一种不断演变的艺术。本书试图呈现这种艺术形式所依赖的基础，以及这种艺术的当前发展状况。

2.2.3　芯片组成和体系结构的改进

当设计人员努力解决处理器性能与主存储器及其他部件的平衡问题时，提高处理器速度的需求仍在增长。有三种办法可实现处理器的提速：

- 提高处理器硬件速度：这个提速基本上要归功于处理器芯片上逻辑门的尺寸减小，以便更多的门能更紧密地组装在一起；也要归功于时钟频率的提升。随着门电路更紧密地集成在一起，信号的传播时间显著地降低，从而允许处理器提速。时钟频率的提升意味着每个操作被更迅速地执行。
- 提高插入在处理器和主存之间的 cache 容量和速度。尤其是，将处理器芯片自身的一部分用做 cache，cache 的存取时间会显著降低。
- 改变处理器的组成和体系结构以提高指令执行的有效速度。典型地，这包含使用各种形式的并行性。

传统上，性能增益的主导因素是时钟速度的提升和逻辑密度的提高。图 2-12 说明了 Intel 处理器芯片的这种趋势。然而，随着时钟速度和逻辑密度的提高，几个障碍变得更加显著[INTE04b]：

- 功耗：随着芯片上逻辑密度和时钟速度的提高，芯片消耗的功率密度（瓦/cm^2）也随之提高。高密、高速芯片的散热因难成为一个重要的设计问题（[GIBB04]，[BORK03]）。
- RC 延迟：电子在芯片上各晶体管间流动的速度受限于连接它们的金属线的电阻和电容。特别是，延迟随 RC 之积的增长而增长。由于芯片上元件尺寸变小，互连线更细，从而电阻增加了；同时，线排列更紧密，电容也增大了。
- 存储器滞后：正如前面所讨论过的，存储器速度落后于处理器速度。

于是，这里更强调以组成和体系结构的办法来改善性能。图 2-12 说明了这些年来在提高并行性以及所带来的处理器计算效率的提高等方面发生的变化。本书的后几章将讨论这些技术。

除了通过简单地提高时钟速度所实现的性能提升之外，从 20 世纪 80 年代后期开始并一直持续了约 15 年，为进一步提升性能还采用两种主要策略。第一，增加 cache 容量。现在，处理器与主存之间一般都有两级或三级 cache。由于芯片密度的提高，更多的 cache 存储器已集成到处理器芯片上，从而允许更快的 cache 访问。例如，最初 Pentium 芯片将大约 10% 的芯片面积用于 cache，而最近的 Pentium 4 芯片将大约一半的芯片面积用于 cache。

第二，处理器内指令执行逻辑变得越来越复杂，以允许处理器内指令并行执行。两个值得重视的设计办法是流水化和超标量化。指令流水线的工作情况非常类似于制造厂的装配线，它允许不同指令的不同执行段在流水线上同时工作。本质上讲，超标量办法是允许在单个处理器内有多条指令流水线，以便彼此无关的指令能并行地执行。

这两种策略正在到达收益递减点。当代处理器的内部组织已是非常复杂，并能够从指令流中压榨出大量的并行性。看起来，在这个方向上进一步显著地提升性能是相当有限的[GIBB04]。随着处理器芯片设置三级 cache，每级都有相当大的容量，看来增加 cache 容量所带来的好处也达到了瓶颈。

然而，简单地依靠提高时钟频率来提高性能又走入了已指出的功率消耗问题。时钟频率越

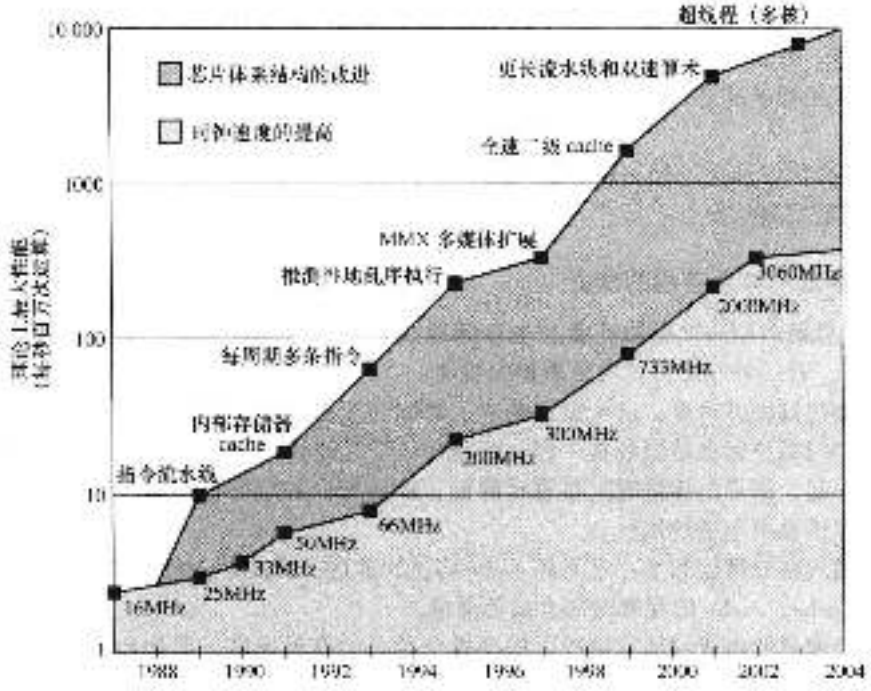


图 2-12 Intel 微处理器性能 [GIBB04]

快，消耗的功率就越大，并且将达到某些基本的物理限制。

考虑到所有这些困难，设计者已转向一种根本性的新办法来改善性能：在同一芯片上安排多个处理器并带有大的共享 cache。同一芯片上多个处理器的使用，也称为多核（multiple cores 或 multicore），提供了无需提高时钟频率就能提高性能的潜力。已有研究指出，在处理器内，性能的提高大致正比于复杂度提高的平方根 [BORK03]。但是，如果软件能够有效地支持多个处理器的使用，则处理器数目的加倍几乎使性能加倍。因此，此策略是使用芯片上两个较简单的处理器，而不是一个更复杂的处理器。

此外，具有两个处理器，一个更大的 cache 也是恰当的。因芯片上存储逻辑的功耗远小于处理逻辑的功耗，故这种安排很重要。未来几年，我们可以期望最新的处理器芯片将具有多个处理器。

2.3 Intel x86 体系结构的进展

贯穿本书，我们依据计算机设计和实现的许多具体实例来说明各种概念，并阐述各类方法之间的权衡。本书主要依据两个计算机系列的例子：Intel x86 和 ARM 体系结构。当代的 Intel x86 代表了在复杂指令集计算机（CISC）中几十年设计成果的结晶。它采用了过去只有大型机和超级计算机中才会采用的复杂设计原则，是 CISC 设计的优秀范例。处理器设计的另一种方法是精简指令集计算机（RISC）。ARM 体系结构被广泛应用于各种类型的嵌入式系统，它是市场上基于 RISC 技术的功能最强大、设计最好的系列之一。

下面将简单描述这两个系统。

依据市场份额，Intel 公司在过去的几十年中始终是微处理器（非嵌入式系统）的领先制造商，这一地位似乎难以动摇。其旗舰的微处理器产品的演变历史，是整个计算机技术演变的一个很好的写照。

表 2-6 给出了这一演变的历史。有趣的是，当微处理器变得更快更复杂时，Intel 的步调总是

那么合拍。过去，Intel 习惯于每 4 年开发出一种新的处理器，但它期望将每代开发时间缩短到 1 ~ 2年，以继续保持领先优势，并且已经这样做了，从而加速推出了几代最新的 x86 产品。

列出 Intel 产品系列的一些主要进展是有益的：

- 8080：世界上第一台通用微处理器。它是 8 位机，存储器的数据通路为 8 位。8080 曾用于第一台个人计算机 Altair。
- 8086：比 8080 更强大的 16 位微处理器。除了更宽的数据通路和更大的寄存器外，8086 还支持指令高速缓存（或称为队列），在指令被实际执行之前，它能预取几条指令。这种处理器的一个变形是 8088，8088 曾用于 IBM 公司的第一台个人计算机，并确保了 Intel 的成功。8086 标志着 x86 体系结构的首次出现。
- 80286：它是 8086 的扩展产品，可以寻址 16MB 的存储器，而不是 1MB 存储空间。
- 80386：Intel 的第一个 32 位机器，是一个有重大改进的产品。其 32 位的体系结构，使 80386 的复杂程度和功能可以与几年前推出的小型机和大型机相媲美。80386 是 Intel 公司第一个支持多任务的处理器，即它能够同时运行多个程序。
- 80486：80486 采用了更为复杂、功能更强的高速缓存技术和指令流水线技术。它的内置式的数学协处理器，减轻了主处理器的复杂算术运算的负担。
- Pentium：从 Pentium 开始，Intel 推出了超标量技术，它允许多条指令并行地执行。
- Pentium Pro：Pentium Pro 继续推进由 Pentium 开始的超标量结构，极富进取性地使用了包括寄存器重命名、分支预测、数据流分析、推断执行等技术。
- Pentium Ⅱ：融入了专门用于有效处理视频、音频和图形数据的 Intel MMX 技术。
- Pentium Ⅲ：融入了一些附加的浮点数指令，以便支持三维图形软件。
- Pentium 4：包括了另一些浮点指令，并对其他多媒体应用进行了增强[⊖]。
- Core：这是第一款具有双核的 Intel x86 处理器，涉及在单芯片上双处理器的实现。
- Core 2：Core 2 将体系结构扩展到 64 位。Core 2 Quad 在单芯片上提供了 4 个处理器。

Intel x86 体系结构自 1978 年推出至今已有 30 多年，它一直统治着除嵌入式系统之外的处理器市场。虽然 x86 机器的组成和技术在几十年间发生着戏剧性的变化，但其指令集结构一直保持着向后兼容其早期的版本，因此，任何写于 x86 体系结构早期版本的程序都可以在其更新的版本上运行。所有对指令集结构的改变都是添加指令集，而不是减少指令集。30 多年来，指令集的变化速度大约是每月增加 1 条其他指令［ANTI108］，因此，目前 x86 指令集有 500 多条指令。

Intel x86 很好地说明了过去 30 多年来计算机硬件的发展。1978 年推出的 8086，其时钟频率为 5MHz，包含 29 000 个晶体管。而 2008 年推出的 4 核 Core 2，其主频为 3GHz，是 8086 的 600 倍；它包含 8.2 亿个晶体管，大约是 8086 的 28 000 倍。然而，Core 2 只是比 8086 的封装稍微大一点，且价格可比。

2.4　嵌入式系统和 ARM

ARM 体系结构是指其处理器结构遵循 RISC 设计原则，并用于嵌入式系统之中。

第 13 章将详细说明 RISC 的设计原则。本节将简要给出嵌入式系统的概念，然后考察 ARM 的演变。

2.4.1　嵌入式系统

术语“嵌入式系统”是指电子学和软件在产品中的使用，它与通用计算机系统不同，例如膝

⊖ 在 Pentium 4 中，Intel 公司将罗马数字转换为阿拉伯数字作为模型数字。

上计算机或桌面系统。下面是一种很好的定义[⊖]：

> **嵌入式系统**：是计算机硬件、软件和可能附加的机械或其他部分的一种组合，用于执行特定的功能。在许多情况下，嵌入式系统是大型产品和系统的组成部分，例如轿车的刹车系统。

嵌入式系统超出了通用计算机，具有更广泛的应用，如表2-7所示。这些系统有各种变化的需求和限制，如下所述［GRIM05］：

- 从小到大的系统，意味着完全不同的成本限制，因此，对优化和再利用有不同的要求。
- 不严格的到非常严格的需求和不同质量要求的组合，例如，关于安全性、可靠性、实时性、灵活性和合法性的需求。
- 从短到长的生命时间。
- 依据不同的环境条件，例如，辐射、振动和潮湿环境。
- 不同的应用特征导致了静态负载对动态负载、慢速对快速、计算密集型任务对交互密集型任务以及它们的组合。
- 不同的计算模型，从离散事件系统到包含连续时间动态的系统（通常是指混合系统）。

表2-7 嵌入式系统及其市场的例子［NOER05］

市场	嵌入式设备
汽车	点火系统 发动机控制 刹车系统
消费电子	数字和模拟电视 置顶盒（数字化视频光盘、录像机、分线盒） 个人数字助理（PDA） 厨房用具（冰箱、炸锅、微波炉） 汽车 玩具/游戏机 电话/手机/呼机 照相机 全球定位系统
工业控制	机器人和用于制造业的控制系统 传感器
医学	泵输系 透析机 矫形术设备 心脏监视器
办公自动化	传真机 影印机 打印机 监视器 扫描仪

通常，嵌入式系统与其环境密切相关。这能够通过环境的相互影响而引起实时限制。所有限制，例如运动速度的要求、测量精度的要求和时间期间的要求，表明了软件操作的定时。如果必须同时处理多个行为，则包含更复杂的限制。

基于［KOOP96］，图2-13给出了一个嵌入式系统组成的基本术语。除了处理器和存储器，还有很多不同于典型桌面或膝上计算机的元件：

⊖ 贾克尔·巴尔，嵌入式系统术语表，参见 Netrino 技术图书馆，网址为 http://www.netrino.com/Publication/Glossary.html。

- 可能有各种用于测量、操作和其他与外部环境交互的界面。
- 人机界面可以像闪光一样简单，也可以像实时机器人视觉一样复杂。
- 诊断端口可以用来诊断正在控制的系统，而不仅仅是诊断计算机。
- 特殊用途的现场可编程门阵列（FPGA）、专用集成电路（ASIC）、甚至非数字式硬件，可以用来增强性能或安全性。
- 软件常常有一个固定的功能并且对该应用有特效。

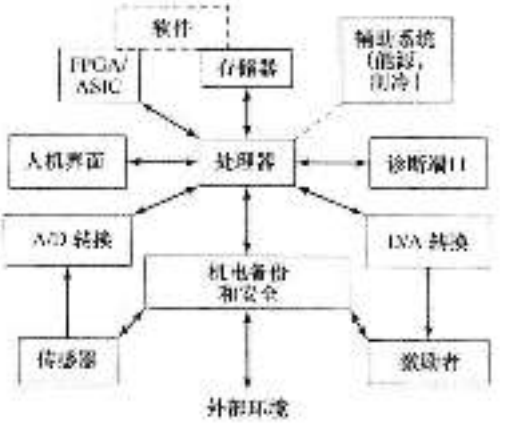


图 2-13　一个嵌入式系统的可能组成

2.4.2　ARM 的进展

ARM 是一种由英国剑桥 ARM 公司设计的基于 RISC 的微处理器和微控制器序列。该公司并不生产处理器，而是设计微处理器和多核的体系结构，然后向制造商发放许可。ARM 芯片是高速的处理器，这是因为它们的小特征尺寸和低能耗需求。它们被广泛应用于 PDA 以及其他手提设备中，包括手机和游戏机以及各种消费产品。ARM 芯片是苹果公司流行的 iPod 和 iPhone 设备的处理器。ARM 可能是最广泛使用的嵌入式处理器体系结构，并且确实是世界上各种应用中使用最广泛的处理器体系结构。

ARM 技术的起源可以追溯到英国的 Acorn 计算机公司。在 20 世纪 80 年代早期，Acorn 获得了英国广播公司（BBC）的合同，为其公司的计算机文化项目开发一款新的微计算机体系结构。这个合同的成功促使 Acorn 继续开发出其第一款商用 RISC 商业处理器：ARM（Acorn RISC Machine）。第一个版本 ARM1，在 1985 年变成可用，并用于内部的研究和开发，以及用于 BBC 机器上作为协处理器。同样在 1985 年，Acorn 推出了 ARM2，它在相同的物理空间内比 ARM1 有更强的功能和更快的速度。进一步的改进是 1989 年推出的 ARM3。

整个这个时期，Acorn 利用公司的 VLSI 技术做处理器芯片的实际制造。VLSI 允许授权其自己的芯片市场，同时使得其他一些公司在其产品中使用 ARM 也获得了一些成功，尤其是在嵌入式处理器中。

ARM 设计顺应了嵌入式应用中对高性能、低功耗、小体积和低成本的处理器的不断增长的商业需求。但进一步发展超出了 Acorn 的能力范围，于是，由 Acorn、VLSI 以及苹果计算机作为股东，成立了一家新公司，叫 ARM Ltd。Acorn 的 RISC 机器变成了先进的 RISC 计算机[⊖]。新公司最先推出了 ARM6，它是 ARM3 的一种改进版本。接着，公司推出了许多新的系列，以增强功能和性能。表 2-8 列出了各种 ARM 结构系列的一些特征，表中的数据只是一个近似的指导值，实际值因不同的实现会有所不同。

根据 ARM 网站 arm.com，ARM 处理器用于满足三种系统类别的需要：

- **嵌入式实时系统**：储藏室、汽车和动力火车、工业以及网络应用的系统。
- **应用平台**：在无线消费娱乐和数字图像应用领域中，运行开放式操作系统的设备。开放式操作系统包括 Linux、Palm OS、Symbian OS 和 Windows CE。
- **安全应用**：智能卡、SIM 卡和支付终端。

⊖ 该公司在 20 世纪 90 年代后期停止使用先进的 RISC 机器的名称，现在简称为 ARM 体系结构。

表 2-8 ARM 的进展

序列	显著特征	cache	典型的 MIPS@ MHz
ARM1	32 位 RISC	无	
ARM2	乘法和交换指令；集成存储器管理单元，图形和 I/O 处理器	无	7MIPS@12MHz
ARM3	第一次使用处理器 cache	4KB 统一	12MIPS@25MHz
ARM6	第一次支持 32 位地址；浮点单元	4KB 统一	28MIPS@33MHz
ARM7	集成 SoC	8KB 统一	60MIPS@60MHz
ARM8	5 段流水线；静态分支预测	8KB 统一	84MIPS@72MHz
ARM9		16KB/16KB	300MIPS@300MHz
ARM9E	增强 DSP 指令	16KB/16KB	220MIPS@200MHz
ARM10E	6 段流水线	32KB/32KB	
ARM11	9 段流水线	可变的	740MIPS@665MHz
Cortex	13 段超级流水线	可变的	2000MIPS@1GHz
XScale	应用型处理器；7 段流水线	32KB/32KB L1 512KB L2	1000MIPS@1.25Hz

注：DSP – 数字信号处理器
SoC – 系统芯片

2.5 性能评价

在评价处理器硬件和设置新系统的需求时，性能是必须考虑的关键因素之一，这包括成本、尺寸、安全性、可靠性以及某些情况下的能源消耗。

在不同的处理器之间，甚至在同一系列的处理器之间进行意味深长的性能比较是困难的。当执行一个给定的应用时，未加工的速度指标远不及处理器如何完成任务来得重要。不幸的是，应用的执行并不仅仅取决于处理器的速度，还依赖于其指令集、实现语言的选择、编译器的效率以及实现该应用的编程技巧。

本节首先介绍一些处理器速度的传统测量方法，然后考察评定处理器和计算机系统性能的最常用的方法，接着考虑如何从多个测试中获得平均结果，最后讨论通过考虑 Amdahl 定律所产生的现象。

2.5.1 时钟速度和每秒指令数

1. 系统时钟

处理器执行的操作，例如取指令、译码该指令、执行算术运算等，都是由系统时钟掌控的。典型的做法是，所有操作都随着时钟的脉冲开始。因此，在最基本的级别，处理器的速度由时钟产生的脉冲频率来指示，用每秒周期数或赫兹（Hz）来测量。

一般情况下，时钟信号由水晶振子产生。水晶振子在有动力供应时能产生一个连续的信号波。该波被转化为一个数字电压脉冲流，连续地供应给处理器电路（如图 2-14 所示）。例如，一个 1GHz 的处理器每秒接受 10 亿个脉冲。脉冲的速率被定义为时钟频率，或时钟速度。每增加一个脉冲或时钟被称为一个**时钟周期**，或**时钟滴答声**。两个脉冲之间的时间定义为**周期时间**。

时钟频率不是任意的，它必须适应处理器的物理层。处理器中的操作需要信号将其从处理器的一个元件传送到另一个元件。当信号被放在处理器内部的一根线上时，它将占用一些有限的时间量来使电压水平平静下来，以便一个正确的值（1 或 0）可用。此外，这取决于处理器电路的物理层，有些信号可能比其他信号变化得更快，因此，操作必须同步，以便适当的电信号

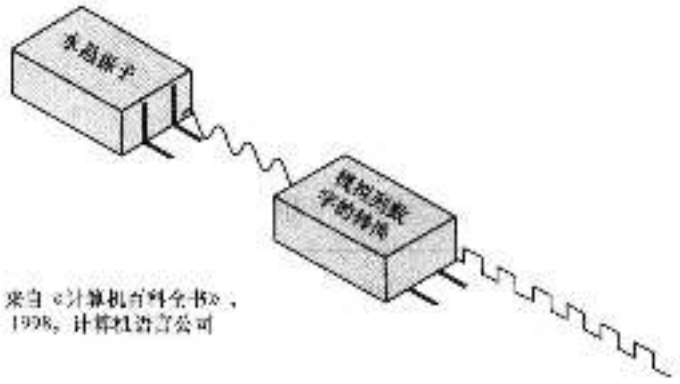


图2-14 系统时钟

（电压）值可为每个操作周到。

指令的执行包含很多离散的步骤。例如，从存储器中取出该指令、译码指令的各个部分、取和存数据以及执行算术和逻辑运算，因此，大多数处理器的大部分指令需要多个时钟周期来完成。有些指令可能只需要几个周期，而另一些指令需要几十个周期。此外，当使用流水线时，多条指令被同时执行，因此，不同处理器的时钟速度的直接比较是不能说明性能的整体情况的。

2. 指令执行速度

处理器由时钟驱动，时钟具有固定的频率 f，或等价为固定的时钟周期 τ，这里 $\tau = 1/f$。定义一个程序的指令条数 I_c，为运行完该程序所执行的机器指令条数。注意这是指令执行的条数，而不是该程序目标代码中的指令条数。程序的一个重要参数是每条指令的平均周期数（average cycles per instruction，CPI）。如果所有指令需要相同的时钟周期数，则该程序的 CPI 就是一个固定值。然而，对于任意指定的处理器，不同的指令类型，例如取数、存数、分支等等，会需要不同的时钟周期数。如果用 CPI_i 来表示指令类型 i 所需要的周期数，用 I_i 表示在某一给定程序中所执行的 i 类指令的条数，则我们可以计算整个 CPI 如下：

$$\mathrm{CPI} = \frac{\sum_{i=1}^{n} (\mathrm{CPI}_i \times I_i)}{I_c} \tag{2.1}$$

处理器执行一个给定的程序所需要的时间可以表示为：

$$T = I_c \times \mathrm{CPI} \times \tau$$

我们可以通过分析一条指令的执行过程来定义该公式，指令执行的一部分工作是由处理器完成的，而另一部分时间花费在处理器与存储器之间的字传送上。在后一种情况中，传送时间取决于存储器周期时间，它可能比处理器周期时间长。将上等式改写成：

$$T = I_c \times [p + (m \times k)] \times \tau$$

这里，p 是译码和执行指令所需要的处理器周期数，m 是所需的存储器访问次数，k 是存储器周期时间和处理器周期时间之比。上面等式中的 5 个性能因子（I_c、p、m、k、τ）受 4 个系统属性影响：指令集设计（亦称指令集结构）、编译技术（编译器如何高效地将高级语言程序转换为有效的机器语言程序）、处理器实现以及 cache 与主存的层次结构。基于 [HWAN93]，表 2-9 是一个矩阵，其横轴表示 5 个性能因子，纵轴表示 4 个系统属性。单元中的 X 说明系统属性影响着性能因子。

表 2-9 性能因子和系统属性

	I_c	p	m	k	τ
指令集结构	X	X			
编译技术	X	X	X		
处理器实现		X			X
cache 与主存的层次结构				X	X

处理器性能的一个通用度量是指令执行的速率，表示成每秒百万条指令（MIPS），也称为MIPS速度。我们可以用时钟频率和CPI表示MIPS速率如下：

$$\text{MIPS 速度} = \frac{I}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} \tag{2.2}$$

例如，考虑在一个400MHz的处理器上运行一个包含200万条指令的程序，该程序由四种主要的指令类型组成。基于程序踪迹实验的结果，得出的指令混合和每一种指令类型的CPI如下：

指令类型	CPI	指令混合比
算术和逻辑	1	60%
cache命中的取数/存数	2	18%
分支	4	12%
cache失效的存储器访问	8	10%

当由单一处理器执行该程序时，其平均CPI = 0.6 + (2 × 0.18) + (4 × 0.12) + (8 × 0.1) = 2.24，相应的MIPS速度 = $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$。

另一个通用的性能度量仅仅用于浮点指令，这在很多科学计算和游戏应用中常见。浮点性能表示为每秒百万条浮点操作（MFLOPS），定义如下：

$$\text{MFLOPS 速度} = \frac{\text{程序中执行浮点操作的次数}}{\text{执行时间} \times 10^6}$$

2.5.2 基准程序

使用像MIPS和MFLOPS一样的度量来评价处理器的性能已经被证明是不充分的。因为不同的指令集，所以指令执行速率不是很好地比较不同体系结构性能的方法。例如，考虑如下高级语言语句：

```
A = B + C          /* 假设所有变量在主存中 */
```

传统的指令集结构是指复杂指令集计算机（CISC），该语句能够被编译成一条处理器指令：

```
add    mem(B), mem(C), mem(A)
```

而在典型的RISC机上，该语句可能被编译成：

```
load   mem(B), reg(1);
load   mem(C), reg(2);
add    reg(1), reg(2), reg(3);
store  reg(3), mem(A)
```

因为RISC结构的特性（将在第13章中讨论），所以两种机器可能花费并不多相同的时间运行最初的高级语言语句。如果这个例子是两种机器的代表，则若CISC机的速度为1MIPS，那么RISC机的速度将是4MIPS。但二者处理同样的高级语言程序所需的时间相同。

进一步来说，处理器执行某一给定程序的性能并不能决定它将如何执行其他类型的应用程序。于是，从20世纪80年代后期和90年代早期开始，业界和学术界喜欢使用一系列基准程序来测量系统的性能。一组相同的程序可以运行在不同的机器上，并对执行时间进行比较。

文献[WEIC90]列出了作为基准程序所需要具备的一些特征：

(1) 它由高级语言编写，可以方便地应用于不同的机器。

(2) 它是各种特殊程序设计方式的代表，例如，系统程序设计、数字程序设计或商业程序设计。

(3) 易于度量。

(4) 它有广泛的发行。

1. SPEC 基准程序

业界、学术界和研究院对公认的计算机性能衡量方法的共同需求导致了标准基准程序集的发展。基准程序集就是一个程序集合，使用高级语言定义，它试图对在特殊应用或系统程序设计领域中的计算机提供一种有代表性的测试。最著名的测试程序集由系统性能评估公司（SPEC，一种工业社团）定义和维护。SPEC 性能测试广泛应用于比较和研究的目的。

最著名的 SPEC 基准程序集是 SPEC CPU2006，这是一种测量强调处理器应用的工业标准集，也就是说，SPEC CPU2006 适合于测试计算密集型应用而非 I/O 密集型应用的性能。SPEC CPU2006 程序集基于 SPEC 工业成员使用的各种平台上现有的应用，它包含 17 种使用 C、C++ 和 Fortran 编写的浮点程序以及 12 种使用 C 和C++ 编写的整数程序，其代码总量超过了 300 万行。SPEC CPU2006 是 SPEC 强调处理器应用的第五代产品，替代了 SPEC CPU2000、SPEC CPU95、SPEC CPU92 和 SPEC CPU89 四代产品 [HENN07]。

其他 SPEC 测试集如下：

- SPECjvm98：试图评估 Java 虚拟机（JVM）客户端的软硬件组合的性能。
- SPECjbb2000（Java 商业基准程序）：用于评估基于 Java 的电子商务应用的服务器端的基准程序。
- SPECweb99：评估万维网（WWW）服务器端的性能。
- SPECmail2001：设计用于测量作为邮件服务器的系统的性能。

2. 平均结果

为了可靠地比较各种计算的性能，最好是在每个计算机上运行多个不同的测试程序，然后取平均结果。例如，若有 m 个不同的测试程序，则简单的算术平均值可如下计算：

$$R_a = \frac{1}{m}\sum_{i=1}^{m} R_i \tag{2.3}$$

其中，R_i 是第 i 个测试程序的高级语言指令的执行速度。

另一个办法是取其调和平均值：

$$R_H = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}} \tag{2.4}$$

基本上来说，用户关心的是系统的执行时间，而非它的执行速度。如果我们获得了各种测试程序的指令执行速度的算术平均，那么就能得到一个与执行时间倒数之和成正比的结果，但这不与执行时间之和成反比。换句话来说，指令执行速度的算术平均并不能准确地刻画执行时间。另一方面，指令执行速度的调和平均值是平均执行时间的倒数。

SPEC 基准程序不关心指令执行的速度，它感兴趣的是另外两个基本的度量：速度度量和频率度量。速度度量测量一台计算机完成单个任务的能力。SPEC 通过参照机为每个基准程序定义一个基本的运行时间。在系统上的测试结果表示为参考运行时间与系统运行时间的比值，比值的计算如下：

$$r_i = \frac{Tref_i}{Tsut_i} \tag{2.5}$$

其中，$Tref_i$ 是基准程序 i 在参照系统上的运行时间，而 $Tsut_i$ 是基准程序 i 在被测系统上的运行时间。

作为计算和报告的例子，考虑 Sun 公司的 Blade 6250，该机器包含两块芯片，每片上有 4 个核或处理器。SPEC CPU2006 整数基准程序之一是 464.h264ref，它是最新上市的视频压缩标准 H.264/AVC（先进的视频编码）的参考实现。Sun 公司的 Blade 6250 执行这个程序需要 934 秒，而参考实现需要 22 135 秒，因此，计算的比值为：22 136/934 = 23.7。

由于被测系统的时间是分母，因此比值越大，速度越高。被测系统全面的性能测量通过计算所有 12 个整数基准程序的平均比值得到。SPEC 详细说明了几何平均数的使用，定义如下：

$$r_G = \left(\prod_{i=1}^{n} r_i\right)^{1/n} \tag{2.6}$$

其中，r_i 为第 i 个基准程序的比值。对于 Sun 公司的 Blade 6250 系统来说，SPEC 整数速度比值报告如下：

基准程序	比值	基准程序	比值
400.perlbench	17.5	458.sjeng	17.0
401.bzip2	14.0	462.libquantum	31.3
403.gcc	13.7	464.h264ref	23.7
429.mcf	17.6	471.omnetpp	9.23
445.gobmk	14.7	473.astar	10.9
456.hmmer	18.6	483.xalancbmk	14.7

对比值的乘积开十二次方可以计算得到速度度量：

$(17.5 \times 14 \times 13.7 \times 17.6 \times 14.7 \times 18.6 \times 17 \times 31.3 \times 23.7 \times 9.23 \times 10.9 \times 14.7)^{1/12} = 18.5$

频率度量测量一台机器执行多个任务的吞吐量或频率。对于频率度量，基准程序的多个拷贝被同时运行。通常，拷贝的数量与机器的处理器个数相同。虽然计算更加复杂，但频率也用于报告结果。频率的计算公式如下：

$$r_i = \frac{N \times Tref_i}{Tsut_i} \tag{2.7}$$

其中，$Tref_i$ 是基准程序 i 的参照运行时间，N 是同时运行的程序的拷贝数目，$Tsut_i$ 是基准程序 i 的所有拷贝在被测系统的所有 N 个处理器上从开始执行到完成任务所需要的时间。同时，计算几何平均值来决定总体性能测量。

SPEC 选择使用几何平均值，因为它最适合数据归一化，例如比值。[FLEM86] 揭示了几何平均值与所获得的计算机的性能有持续的关系，无论该计算机是否作为归一化的基础。

2.5.3 阿姆达尔定律

当考虑系统性能时，计算机系统设计者希望通过改进技术或者改变设计来提高性能，例如，并行处理器的使用，存储器高速缓存的使用，以及由于技术的改进来加快存储器的访问和 I/O 的传送速率。在所有的这些情况中，需要特别注意的是，只是加速技术或设计的一个方面并不能提高性能的相应改善。用阿姆达尔（Amdahl）定律可以很好地说明其局限性。

阿姆达尔定律最早由 Gene Amdahl 在文献 [AMDA67] 中提出，用于研究一个程序在使用多个处理器时与使用单个处理器时可能出现的加速比。考虑一个运行在单处理器上的程序，执行时间中的 $(1-f)$ 部分表示其代码是固有的、只能被串行执行的部分，f 部分表示其代码可以无限制地并行、无调度负载。若假设 T 为该程序在单个处理器上的总执行时间，则使用具有 N 个处理器的并行系统后，探索该程序整个并行部分的加速比如下：

$$\text{Speedup} = \frac{\text{在单个处理器上执行程序的时间}}{\text{在 } N \text{ 个并行处理器上执行程序的时间}}$$

$$= \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

可以推导出两个重要的结论：

(1) 当f非常小时，使用并行处理器只有一点点影响。

(2) 随着N接近于无限大，加速比被$1/(1-f)$所限制，因此使用更多的处理器只能导致速度下降。

这些结论太悲观。一个新言在文献[GUST88]中首次提出。例如，一个服务器可以支持多线程和多任务来处理多个客户端，并且并行地执行线程和任务，突破处理器数目的限制；许多数据库应用包含大量数据的计算，这可以分成多个并行的任务。然而，阿姆达尔定律揭示了计算机工业在开发具有不断增长的核数的多核机器时所面临的问题：运行在多核机器上的软件必须适应高速并行执行环境，以利用并行处理的能力。

阿姆达尔定律在评价计算机系统的任何设计和技术改进方面是通用的。考虑任何影响加速比的系统改进特征，加速比可以如下表示：

$$\text{Speedup} = \frac{\text{改进后的性能}}{\text{改进前的性能}} = \frac{\text{改进前的执行时间}}{\text{改进后的执行时间}} \tag{2.8}$$

假如改进前系统可改进部分的执行时间为f，改进后，可改进部分的加速比为SU_f，则系统的总加速比可以表示成：

$$\text{Speedup} = \frac{1}{(1-f) + \dfrac{f}{SU_f}}$$

例如，假如一个任务密集地使用浮点操作，浮点操作占整个操作时间的40%。现有一个新的设计，其浮点操作部分被加速了K倍，则总加速比为：

$$\text{Speedup} = \frac{1}{0.6 + \dfrac{0.4}{K}}$$

即与K无关，最大加速比为1.67。

2.6 推荐的读物和Web站点

IBM 7000系列的介绍可以在[BELL71]中找到。[SIEW82]对IBM 360的介绍较全面，[BELL78a]对PDP-8和其他DEC机器有较好的说明。这三本书同时包含了20世纪80年代早期各种其他计算机的许多详尽的例子。[BLAA97]是一本较新的书，它包含一组学习以前机器的优秀实例。[BETK97]很全面地介绍了微处理器的历史。

[OLUK96]、[HAMM97]和[SAKA02]讨论了单芯片上多处理器的动机。

[BREY09]对Intel微处理器较提供了一个好的综述，Intel自己的资料[INTE08]也很好。

[SEAL00]是目前可得到的、对ARM结构介绍最全面的资料。[FURB00]是另一个优秀的信息源。[SMIT08]对ARM和X86在无线移动设备中作为嵌入式处理器进行了有趣的比较。

对摩尔定律及其效果的有趣讨论见[HUTC96]、[SCHA97]和[BOHR98]。

[HENN06]对CPU2006中的每个基准程序进行了详细的描述。[SMIT88]讨论了算术平均值、其和平均值和几何平均值的关系。

BELL71 Bell, C., and Newell, A. *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

BELL78A Bell, C.; Mudge, J.; and McNamara, J. *Computer Engineering: A DEC View of Hardware Systems Design.* Bedford, MA: Digital Press, 1978.

BETK97 Betker, M.; Fernando, J.; and Whalen, S. "The History of the Microprocessor." *Bell Labs Technical Journal*, Autumn 1997.

① 在ARM社团中被称为"ARM ARM"。

BLAA97 Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

BOHR98 Bohr, M. "Silicon Trends and Limits for Advanced Microprocessors." *Communications of the ACM*, March 1998.

BREY09 Brey, B. *The Intel Microprocessors: 8086/8066, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4 and Core2 with 64-bit Extensions*. Upper Saddle River, NJ: Prentice Hall, 2009.

FURB00 Furber, S. *ARM System-On-Chip Architecture*. Reading, MA: Addison-Wesley, 2000.

HAMM97 Hammond, L.; Nayfay, B.; and Olukotun, K. "A Single-Chip Multiprocessor." *Computer*, September 1997.

HENN06 Henning, J. "SPEC CPU2006 Benchmark Descriptions." *Computer Architecture News*, September 2006.

HUTC96 Hutcheson, G., and Hutcheson, J. "Technology and Economics in the Semiconductor Industry." *Scientific American*, January 1996.

INTE08 Intel Corp. Intel® 64 and *IA-32 Intel Architectures Software Developer's Manual (3 volumes)*. Denver, CO, 2008. intel.com/products/processor/manuals.

OLUK96 Olukotun, K., et al. "The Case for a Single-Chip Multiprocessor." *Proceedings, Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

SAKA02 Sakai, S. "CMP on SoC: Architect's View." *Proceedings, 15th International Symposium on System Synthesis*, 2002.

SCHA97 Schaller, R. "Moore's Law: Past, Present, and Future." *IEEE Spectrum*, June 1997.

SEAL00 Seal, D., ed. *ARM Architecture Reference Manual*. Reading, MA: Addison-Wesley, 2000.

SIEW82 Siewiorek, D.; Bell, C.; and Newell, A. *Computer Structures: Principles and Examples*. New York: McGraw-Hill, 1982.

SMIT88 Smith, J. "Characterizing Computer Performance with a Single Number." *Communications of the ACM*, October 1988.

SMIT08 Smith, B. "ARM and Intel Battle over the Mobile Chip's Future." *Computer*, May 2008.



推荐的 Web 站点

- Intel Developer Page：Intel 为开发人员设置的 Web 页，提供了访问 Pentium 信息的起始位置，还包括了 Intel 技术期刊。
- ARM：ARM 有限公司（ARM 体系结构的开发者）的主页，包括技术资料。
- Standard Performance Evaluation Corporation：SPEC 是计算机工业广泛认可的组织，它开发的标准基准程序被用来评价和比较各种不同的计算机系统的性能。
- Top500 Supercomputer Site：对当代超级计算机产品的体系结构和组成提供重要的描述，并进行比较。
- Charles Babbage Institute：提供与几个涉及计算机历史的 Web 站点的链接。

2.7 关键词、思考题和习题

关键词

accumulator（AC）：累加器
Amdahl's law：阿姆达尔定律
arithmetic and logical unit（ALU）：算术逻辑单元
benchmark：基准程序
chip：芯片
data channel：数据通道
embedded system：嵌入式系统
execute cycle：执行周期

fetch cycle：取指周期
input-output（I/O）：输入/输出
instruction buffer register（IBR）：指令缓冲寄存器
instruction cycle：指令周期
instruction register（IR）：指令寄存器
instruction set：指令集
integrated circuit（IC）：集成电路
main memory：主存储器，主存，内存
memory address register（MAR）：存储器地址寄存器
memory buffer register（MBR）：存储器缓冲寄存器
microprocessor：微处理器
multicore：多核
multiplexor：多路选择器
opcode：操作码
original equipment manufacturer（OEM）：原始设备制造商
program control unit：程序控制器，程序控制单元
program counter（PC）：程序计数器
SPEC：系统性能评估公司
stored program computer：存储程序式计算机
upward compatible：向上兼容
von Neumann machine：冯·诺伊曼机
wafer：晶片
word：字

思考题

2.1 什么是存储程序式计算机？
2.2 生成通用计算机的4个主要部件是什么？
2.3 对集成电路级别而言，计算机系统的3个基本组成部分是什么？
2.4 解释摩尔定律。
2.5 列出并说明计算机系列的主要特征。
2.6 区分微处理器的关键特征是什么？

习题

2.1 假设A = A(1)，A(2)，…，A(1000)和B = B(1)，B(2)，…，B(1000)是两个向量（一维数组），每个向量包含1000个数。将它们加起来形成数组C，当I = 1，2，…，1000，有C(I) = A(I) + B(I)。试用IAS指令集编写一个程序来解决这个问题。忽略ISA只有1000个存储器字单元的事实。
2.2 (a) 在IAS机上，取存储器地址2的内容的机器代码指令应是怎样的？
(b) 为了完成这条指令，在指令周期内CPU需要访问多少次存储器？
2.3 在IAS机上，需要通过将什么放入MAR、MBR、地址总线、数据总线和控制总线，CPU才能完成由存储器读取一个值或向存储器写入一个值？请用英语描述此过程。
2.4 给出IAS机的存储器内容如下：

地址	内容
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

试写出从地址08A开始的该程序的汇编语言代码，并说明这段程序做什么。
2.5 指出图2-3中每条数据路径（例如，AC和ALU之间）的位宽度。
2.6 在IBM 360的Model 65和Model 75中，地址在两个分开的主存储器中交错排放（例如，所有的奇数序号字存放在一个存储器中，而所有的偶数字存放在另一个存储器中），采用这一技术的目的是什么？
2.7 参照表2-4，可以看出IBM 360 Model 75的相对性能是360 Model 30的50倍，而指令周期时间只快了5倍。你如何解释这种差异？
2.8 在比利·鲍勃的计算机商店时，你听到一个顾客问比利·鲍勃他在该商店能买到的最快的计算机是什么。比利·鲍勃回答说："你正在看的是我们的Macintosh机器，最快的Mac机以1.2GHz的时钟速度运行。如果你实在想要最快的机器，你应该购买我们的2.4GHz的Intel Pentium 4计算机。"比利·鲍勃的回答对吗？你应该怎么来帮助这位顾客？
2.9 ENIAC是一个十进制机器，它用10个电子管组成一个环来表示一个寄存器。在任何时刻，只有一个电子管处于ON状态，表示10个数字中的一个。假定，ENIAC有能力使多个电子管同时处于ON和OFF状态，为什么这种表示法是一种"浪费"，我们用10个电子管所能表示整数范围是什么？

2.10 某基准程序在一个 40MHz 的处理器上运行，其目标代码有 100 000 条指令，由如下各类指令及对应时钟周期计数混合组成：

指令类型	指令条数	执行每条指令的周期数
整数运算	45 000	1
数据传送	32 000	2
浮点数运算	15 000	2
控制传送	8000	2

试计算该程序的有效 CPI、MIPS 速率和执行时间。

2.11 考虑两类不同的机器，具有两种不同的指令集，两者的时钟频率都为 200MHz。在此两种计算机上运行一组给定的基准程序的结果如下：

指令类型	指令条数（百万）	执行每条指令的周期数
机器 A		
算术和逻辑运算	8	1
取数和存数	4	3
分支	2	4
其他	4	3
机器 B		
算术和逻辑运算	10	1
取数和存数	8	2
分支	2	4
其他	4	3

(a) 试计算每台机器的有效 CPI、MIPS 速率以及执行时间。

(b) 评论结果。

2.12 CISC 和 RISC 设计的早期例子分别是 VAX 11/780 和 IBM RS/6000。使用一个典型的基准程序，产生如下的机器特征结果：

处理器	时钟频率	性能	CPU 时间
VAX 11/780	5 MHz	1 MIPS	12x 秒
IBM RS/6000	25 MHz	18 MIPS	x 秒

最后一列显示 VAX 需要的 CPU 时间是 IBM 机器的 12 倍。

(a) 运行在这两台机器上的基准程序的机器代码的指令条数的关系是什么？

(b) 这两台机器的 CPI 各是多少？

2.13 在三台计算机上运行四个基准程序的结果如下：

	计算机 A	计算机 B	计算机 C
程序 1	1	10	20
程序 2	1000	100	20
程序 3	500	1000	50
程序 4	100	800	100

下表显示以秒为单位在各台机器上运行 1 亿条指令的执行时间，试计算每台计算机运行每种程序的 MIPS 值。假设这 4 个程序的权重相同，试计算其算术平均值和调和平均值并列该三台计算机。

2.14 下表中的数据来自文献 [HEAT84]，显示在三台机器上运行五种不同基准程序的执行时间，以秒为单位。

基准程序	处理器		
	R	M	Z
E	417	244	134
F	83	70	70
H	66	153	135
I	39 449	35 527	66 000
K	772	368	369

(a) 首先，正规化到机器 R，计算每台计算机运行每种基准程序的速度度量。也就是说，让 R 的速度值都为 1.0，将机器 R 作为参照系统，使用公式(2.5)计算其他机器的速度。使用公式(2.3)计算每个系统的算术平均值。这就是文献［HEAT84］采用的方法。

(b) 采用机器 M 作为参照系统重做问题 (a)，这个计算在文献［HEAT84］中未做。

(c) 基于前面两种计算之一，指出哪一台机器是最慢的。

(d) 使用几何平均值的计算公式(2.6)，重做问题 (a) 和 (b) 的计算，并基于这两种计算指出哪一台机器是最慢的。

2.15 为了分清前面问题的结果，考虑一个更简单的例子。

基准程序	处理器		
	X	Y	Z
1	20	10	40
2	40	80	20

(a) 首先将 X 作为参照机器，然后将 Y 作为参照机器，分别为每个系统计算算术平均值。解释直观上该三台机器有大致相等的性能以及算术平均值给出了误导性的结果。

(b) 首先将 X 作为参照机器，然后将 Y 作为参照机器，分别为每个系统计算几何平均值。说明这些结果比算术平均更现实。

2.16 考虑2.5节中计算平均 CPI 和 MIPS 速度的例子，该例子得到的结果是 CPI = 2.24 和 MIPS 速度 = 178。现假设该程序可以以 8 个并行的任务或线程的方式执行，而每个任务的指令条数大致相等。该程序在一个 8 核的系统上执行，每个核（处理器）的性能与最初使用的单核的性能相同。各部分之间的协调和同步使每个任务增加额外的 25 000 条指令的执行。假设每个任务的指令混合与例子中相同，但由于对存储器的竞争，使 cache 失效时存储器访问的 CPI 增大到 12 个周期。

(a) 计算平均 CPI。

(b) 计算相应的 MIPS 速度。

(c) 计算加速比因子。

(d) 将实际加速比因子与由阿姆达尔定律决定的理论加速比因子进行比较。

2.17 一个处理器访问主存的平均访问时间为 T_2。一个容量比较小的 cache 存储器插在处理器与主存之间。cache 的访问速度比主存快很多，即 $T_1 < T_2$。任何时候，cache 保存主存的一部分拷贝，以便在不久的将来 CPU 最可能访问的字在 cache 中。假设处理器访问的下一个字在 cache 中的可能性为 H，即命中率为 H。

(a) 对于任何单个存储器访问，在 cache 中访问该字与在主存中访问该字的理论加速比是多少？

(b) 假设 T 为平均访问时间，将 T 表示为 T_1、T_2 和 H 的函数，总加速比与 H 的函数是什么？

(c) 实际上，系统被设计为处理器必须首先访问 cache，并决定该字是否在 cache 中，如果该字不在 cache 中，然后才访问主存。因此，当 cache 访问失效（与“命中”相反）时，存储器的访问时间是 $T_1 + T_2$。将 T 表示为 T_1、T_2 和 H 的函数。现在计算其加速比，并与 (b) 中的结果进行比较。

第二部分　计算机系统

一个计算机系统包括处理器、存储器、I/O以及连接主要部件之间的互连。除了处理器以外（处理器十分复杂，本书将在第三部分研究它），本部分将详细考察所有的这些部件。

第3章　计算机功能和互连的顶层视图

从最顶层看，计算机由处理器、存储器和I/O部件组成。系统实功能行为由这些主要部件之间的数据和控制信号的交换所构成。为了支持这种交换，这些部件必须互连。本章首先介绍计算机部件及其输入/输出要求，然后考察影响互连设计的主要问题，尤其是支持中断的需求。本章大部分篇幅用于讨论最普遍的互连方法——总线结构的使用。

第4章　cache存储器

计算机的存储器在类型、技术、组织、性能和成本等方面的范围很广泛。典型的计算机系统都具备了一个层次化的存储子系统，有些存储器是内部的（处理器通过I/O模块来存取），有些存储器是外部的（处理器通过I/O模块来存取）。第4章首先概述这一层次体系，然后详细讨论高速缓存的设计，包括分立的代码和数据cache以及二级cache。

第5章　内部存储器

主存系统设计是一个在容量、存储速度以及成本三个设计要求之间相互竞争、永无休止的过程。随着存储器技术的发展，三个特征的每一个都在改变，就设计上有组织的必须考虑每一种新的实现。第5章关注内部存储器的设计问题，首先考察半导体主存的特性和组成，然后讨论目前先进的DRAM存储器组织。

第6章　外部存储器

为了得到比主存更大的存储容量和更持久的信息存储，都要有外部存储器。最广泛使用的外部存储器类型是磁盘。第6章首先介绍磁盘技术及其设计方面的考虑，然后考察改善磁盘存储器性能的RAID组织，最后考察光盘和磁带机。

第7章　输入/输出

与处理器、主存互联的是I/O模块，每个I/O模块都控制一个或多个外部设备。第7章介绍I/O组织的各个方面。怎如何满足性能需求方面，与计算机设计的其他领域相比，它是一个更复杂、更难理解的领域。第7章考察如何使用程序I/O、中断I/O和直接存储器存取（DMA）这三种技术来实现I/O模块与系统其余部分的交互，并介绍I/O模块与外部设备之间的接口。

第8章　操作系统支持

详细考察操作系统超出了本书的范围。然而，理解操作系统的基本功能以及操作系统如何开发硬件资源以提供所需性能是很重要的。第8章描述操作系统的基本原理，并讨论关于支持操作系统的计算机硬件的专门设计特征。首先介绍操作系统简史，以说明操作系统的主要类型及其使用。然后，通过关键和近期调度功能的考察，来说明多道程序设计。最后考察存储器管理，包括对分段、分页和虚拟存储器的讨论。

第3章

Computer Organization and Architecture: Designing for Performance, 8E

计算机功能和互连的顶层视图

本章要点

- 指令周期的组成如下：首先读指令，然后读本个或多个操作数，再后存本个或多个操作数，最后是中断检查（若中断允许）。
- 计算机系统内主要部件（处理器、主存、I/O模块）为了交换数据和控制信号，需要彼此互连。最流行的互连方式是使用多条线组成的共享系统总线。在当今系统中，通常采用层次式总线来改善性能。
- 总线的设计要素包括：仲裁（以集中式或分布式控制来裁决是否允许把信号发送到总线上）、时序（总线上的信号是与中央时钟同步，还是基于最近传送事件的异步传送）和宽度（地址线条数和数据线条数）。

从顶层来看，计算机包括CPU（中央处理器）、存储器和I/O部件，每种类型有一个或多个模块。这些部件以某种方式互相连接，实现计算机的基本功能，即执行程序。因此，在顶层，我们可以通过两种方法来描述计算机系统：（1）描述每个部件的外部操作，即它与其他部件之间交换的数据和控制信号。（2）描述互连结构和管理互连结构所要求的控制。

从顶层考察结构和功能的重要性在于它有助于理解计算机的特性。另一个重要性在于它可以用来理解性能评估这一日趋复杂的问题。只要把握顶层的结构和功能，就可以洞察系统的瓶颈、选择其他替代通路、了解因部件失效而导致的系统故障的程度，并容易提升系统的性能。在许多情况下，只有通过修改设计而不仅仅是提高单个部件的速度和可靠性，才能满足对更大的系统功能和故障安全能力的要求。

本章重点讨论用于计算机部件互连的基本结构。作为背景，本章首先简要考察基本部件及其接口要求，然后提供功能概述，最后论述如何使用总线来连接系统部件。

3.1 计算机的部件

正如第2章所述，几乎所有的当代计算机设计都是以普林斯顿高级研究院的冯·诺伊曼提出的概念为基础。这种设计称为冯·诺伊曼结构，它基于以下3个主要概念：

- 数据和指令存储在单一的“读/写存储器”中。
- 存储器的内容通过位置寻址，而不关心存储在其中的数据类型。
- 以顺序的形式从一条指令到下一条指令的（除非有明确的修改）执行。

形成这些概念的原因在第2章中已经讨论过，但在这里仍值得总结一下。可以将一小组基本的逻辑部件以各种方式组合起来，用以存储二进制数据和完成对数据的算术和逻辑操作。如果要执行一种特定的计算，需要构造一个专门用于特殊计算的逻辑单元的配置。将各种元件连接成所需配置的过程，可以看成是某种形式的编程。得到的“程序”以硬件的形式存在，并被称为硬布线程序。

现在考虑另一种方案。假设我们构造一个具有算术和逻辑功能的通用结构。这组硬件将根据提供给它的控制信号，对数据执行各种功能。在原先专用化硬件的情况中，系统接收数据并生成输出（如图3-1a所示）。而对于通用的硬件，系统接收数据和控制信号并生成输出。因此，对于每个新程序，程序员只需提供一个新的控制信号集，而不用重新连接硬件。

如何提供控制信号？答案简单又微妙。整个程序实际上由许多步骤组成，对某些数据执行某

种算术或逻辑类作，每一步都需要一组新的控制信号。让我们为每一组控制信号提供一个唯一的代码，并为通用硬件增加能够接收代码和产生控制信号的部分（如图 3-1b 所示）。

现在编程容易多了。不再需要为每个新的程序重新连接硬件，所需要做的只是要提供新的代码序列。事实上，每个代码就是一条指令，而硬件的一部分翻译每条指令并且产生相应的控制信号。为了区分这一新的编程方法，这一代码或指令序列被称为软件。

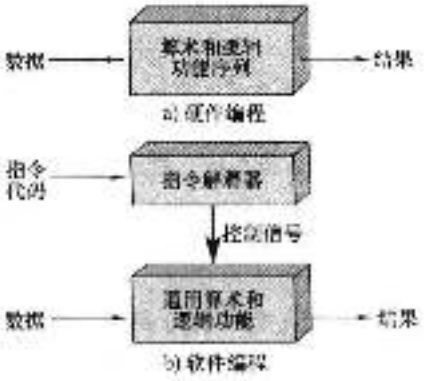


图 3-1　硬件和软件方法

图 3-1b 指出了系统的两个主要部件：指令解释器和通用算术逻辑功能模块。这两部分组成了 CPU。为了制造能够工作的计算机还需要其他几个部件。数据和指令必须能够输入系统，为此，需要某种输入模块。这个模块包含几个基本部件，它们能够以某种形式接收数据和指令，并将其转换成系统能够使用的信号的内部形式。需要有某种报告结果的方法，它可以用输出模块的形式实现。两者放到一起，被称为 I/O 部件。

还需要另外一个部件。输入设备顺序地输入指令和数据，但一个程序并不会始终顺序地执行，它可能跳转到其他地方（例如 IAS 的跳转指令）。类似地，对数据的操作可能要求不只是以一种预先确定的序列每次访问一个单元。因此，必须有一个可以临时存放指令和数据的地方，这个模块被称为存储器或主存，这种称呼是为了将它同外部存储器或外部设备区分开。冯·诺伊曼指出，同一存储器既可以存放指令又可以存放数据。

图 3-2 表示了这些顶层部件并暗示了它们之间的相互作用。CPU 负责与存储器间交换数据，为了这个目的，CPU 一般使用两个内部的寄存器：一个是存储器地址存储器（MAR），为下一次读或写指定存储器的地址；另一个是存储器缓冲寄存器（MBR），容纳写到内存或从内存接收的数据。类似地，I/O 地址寄存器（I/O AR）指定了一个特定的 I/O 设备；I/O 缓冲寄存器（I/O BR）用于 I/O 模块与 CPU 之间的数据交换。

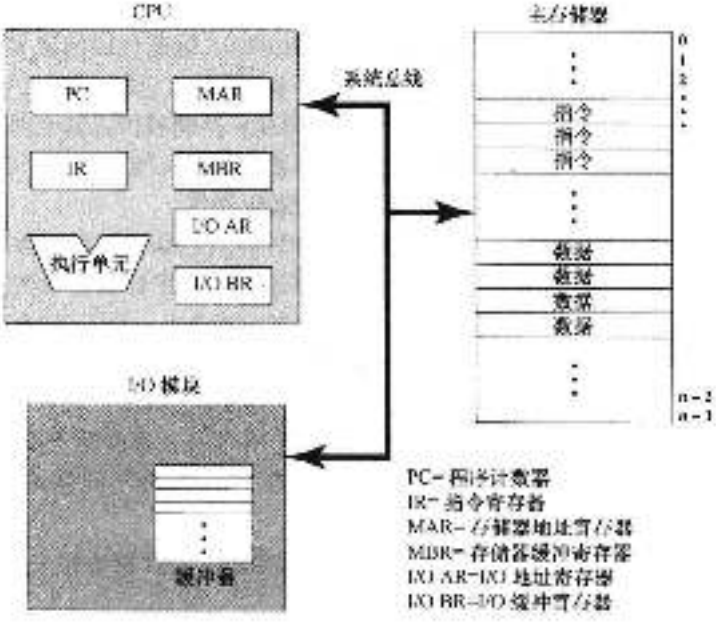


图 3-2　计算机部件：顶层视图

存储器模块包含一组单元，由连续的编号来定义其地址。每个单元都含有一个二进制数，它既可以解释为指令也可以解释为数据。I/O 模块将数据从外设传送到 CPU 或存储器，反之亦然。I/O 模块包含内存缓冲器，用来暂时存放 I/O 数据，直到它们被发送出去。

以上简单介绍了这些部件，下面将概述这些部件如何共同工作来执行程序的。

3.2 计算机的功能

计算机完成的基本功能是执行程序，该程序由存储在存储器中的一串指令组成。处理器通过执行程序中指定的指令来完成实际的工作。本节提供了执行程序的关键元素的概况。在其最简单的形式中，指令的处理由两个步骤组成：处理器从存储器中每次读取（fetch）一条指令，然后执行每条指令。程序的执行便是重复地取指令和执行指令的过程。当然，根据指令的特点，指令的执行可能包含许多步骤（例如，图 2-4 的下半部）。

一条指令所要求的处理过程被称为**指令周期**。根据以上描述的两个简单步骤，图 3-3 描绘了指令周期的处理步骤。这两个步骤分别称为**取指周期**和**执行周期**。只有当关机或某种不可恢复的错误发生或计算机遇到的是一条停机指令时，程序的执行才停止。

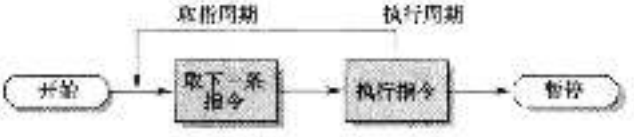


图 3-3 基本指令周期

3.2.1 指令的读取和执行

在每个指令周期的开始，处理器都从存储器中取指令。在典型的处理器中，用一个称为程序计数器（PC）的寄存器来保存下一条将要读取指令的地址。除非特别说明，否则处理器在每次取指令之后总是将 PC 的值加上一个增量，以便将来顺序地读取下一条指令（也就是位于下一个更高存储器地址中的指令）。举例来说，考虑一台计算机，其每一条指令占住存储器的一个 16 位的字单元。现假设程序计数器的值为 300，则处理器下一次将读取 300 单元中的指令；再下一个指令周期，它将取 301 单元中的指令，接着是 302、303 等。正如刚才所解释的，这一顺序允许有所改变。

读取的指令装入处理器中的指令寄存器（IR）。指令以二进制代码的形式存在，它规定了处理器将要执行的动作。处理器解释这条指令并执行所要求的操作。总的来说，这些操作归为 4 类：

- **处理器–存储器**：数据可从处理器传送到存储器或从存储器传送到处理器。
- **处理器–I/O**：通过处理器和 I/O 模块之间的传输，数据可传送到或来自外部设备。
- **数据处理**：处理器可以对数据执行一些算术或逻辑操作。
- **控制**：指令可以用来改变执行顺序。例如，处理器可能从 149 单元取得一条指令，而这条指令指出下一条指令取自 182 单元。处理器通过将程序计数器设置为 182 来记录这一事实。这样，在下一个取指令周期，指令将取自 182 单元，而不是 150 单元。

当然，一条指令的执行可能包含这些操作的组合。

考虑一个简单的例子，使用一台包含图 3-4 所列特点的假想机器，其处理器包含唯一的一个数据寄存器，被称为累加器（AC）；其指令和数据都是 16 位长。这样便于用 16 位的字来组织存储器；其指令格式提供 4 位的操作码，表示最多可以有 $2^4 = 16$ 种不同的操作码，最多有 $2^{12} = 4096$(4K)个字的存储器可以直接寻址。

图 3-5 举例说明部分程序的执行，显示了存储器和处理器寄存器的相关部分[⊖]。该程序段将

⊖ 这里使用的基十六进制表示法，在此表示法中，每个数字代表 4 位。当字长是 4 的整数倍时，它为表示存储器和寄存器内容的较方便的表示法。见第 19 章中的数字系统的基本复习（十进制、二进制和十六进制）。

存储器中地址940的内容与地址941的内容相加，结果放在941中。这里需要3条指令，它们可用3个取指令周期和3个执行周期来表示：

（1）程序计数器（PC）的内容是300，即第1条指令的地址。这条指令（其值为十六进制数1940）被装入指令寄存器IR，并且PC加1。注意，这一过程包含对存储器地址寄存器（MAR）和存储器缓冲寄存器（MBR）的使用。为简便起见，忽略了这些中间寄存器。

（2）IR中的前4位（第1个十六进制数字）指出要装入累加器（AC）。而其余12位（3个十六进制数字）指定从那个地址（940）取数据装载。即地址940的数据0003装入AC。

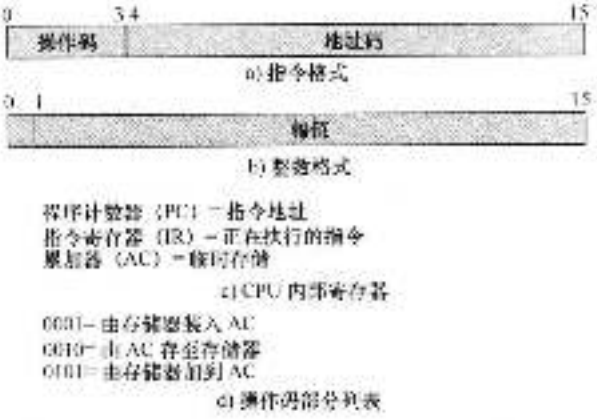


图3-4 假想机特性

（3）从单元301中取下一条指令（5941），并且PC加1。

（4）AC中存放的内容和941单元的内容相加，结果放入AC。

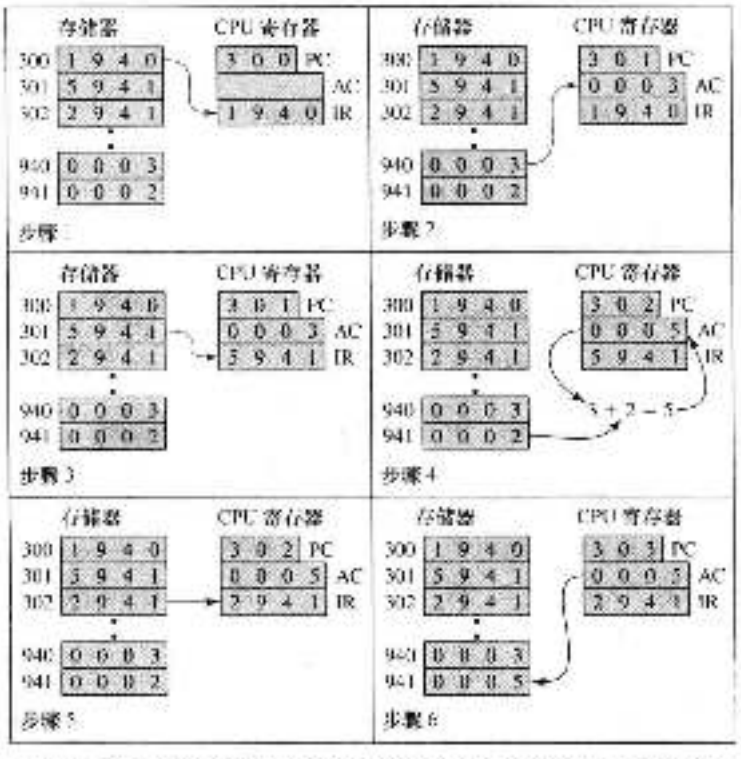


图3-5 程序执行的例子（存储器和寄存器的内容以十六进制表示）

（5）从单元302中取下一条指令（2941），并且PC加1。

（6）将AC的内容存入941单元。

在这个例子中，将940单元的内容加到941单元用了3个指令周期，每个指令周期都包含了一个取指周期和一个执行周期。如果用更复杂的指令集，则需要更少的周期。例如，一些较老的处理器包含具有多个存储器地址的指令，因此，这类处理器中的特殊指令在执行周期可以多次访问存储器。而且，指令可能不用访问存储器，而是指定一个I/O操作。

例如，PDP-11 处理器包含一条指令，其符号表示为“ADD B,A”，它将存储器单元 A 和 B 相加，然后将和存入单元 A 中。这一个指令周期由下列几步组成：

- 取 ADD 指令。
- 将存储单元 A 的内容读入处理器。
- 将存储单元 B 的内容读入处理器。为了使 A 的内容不丢失，处理器必须至少有两个寄存器，而不是一个单一的累加器来存放存储器的值。
- 将两个值相加。
- 将结果从处理器写入内存单元 A 中。

因此，某个特定指令的执行周期可能包含对存储器的多次访问。而且，除访问存储器以外，一条指令可能指定一次 I/O 操作。考虑到这个附加的因素，图 3-6 对图 3-3 的基本指令周期提供了更详细的考虑，它以状态图的形式显示。对于任意给定的指令周期，有些状态可能为空，而另一些可能出现多次。这些状态的描述如下所示：

- **指令地址计算（iac）**：决定下一条将要执行的指令的地址。通常是将一个固定的值与前一条指令的地址相加。例如，如果每条指令有 16 位长，并且存储器是由 16 位字构成的，则将原地址加 1；如果存储器是由可独立寻址的 8 位字节构成的，则将原地址加 2。
- **读取指令（if）**：将指令从存储器单元读到处理器中。
- **指令操作译码（iod）**：分析指令，以决定将执行何种操作以及将使用的操作数。
- **操作数地址计算（oac）**：如果该操作包含对存储器或通过 I/O 的操作数访问，那么决定操作数的地址。
- **取操作数（of）**：从存储器或从 I/O 中读取操作数。
- **数据操作（do）**：完成指令需要的操作。
- **存储操作数（os）**：将结果写入存储器或输出到 I/O。

图 3-6 上半部分的状态图包含处理器与存储器或 I/O 模块的数据交换。图 3-6 下半部分的状态图仅涉及处理器内部的操作。oac 状态出现了两次，这是因为指令包含了读、写或两者兼而有之。但在两种情况中该状态所完成的操作基本相同，因此只需要一个状态标识。

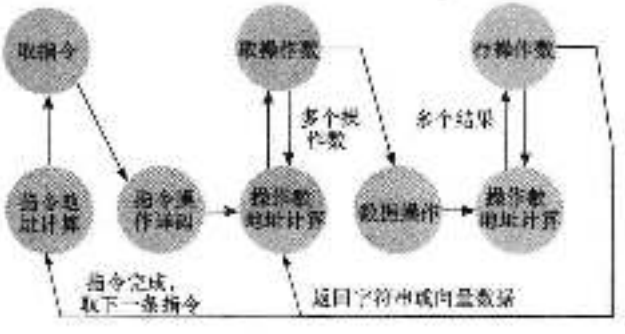


图 3-6 指令周期状态图

还请注意，图 3-6 允许多个操作数和多个结果，因为有些指令和有些机器要求这样。例如，PDP-11 的“ADD A,B”指令导致以下的状态序列：ica、if、iod、oar、of、oac、of、do、osc 和 os。

最后，在某些机器中，单条指令可以指定对向量（一维数组）数据或字符串（一维数组）执行操作。如图 3-6 所示，它将重复取操作数和（或）存储操作数。

3.2.2 中断

几乎所有的计算机都提供一种机制，其他模块（I/O 或存储器）通过此机制可以中断处理器的正常处理。表 3-1 列出了最常见的中断类型。这些中断的特点将在本书的后半部分，特别是在第 7 章和第 12 章中予以讨论。但是为了更清楚地理解指令周期的特性和中断对互连结构的影响，有必要在此介绍这个概念。读者在本阶段不用关心中断的产生和处理等细节，只需专注由中断引起的模块之间的通信。

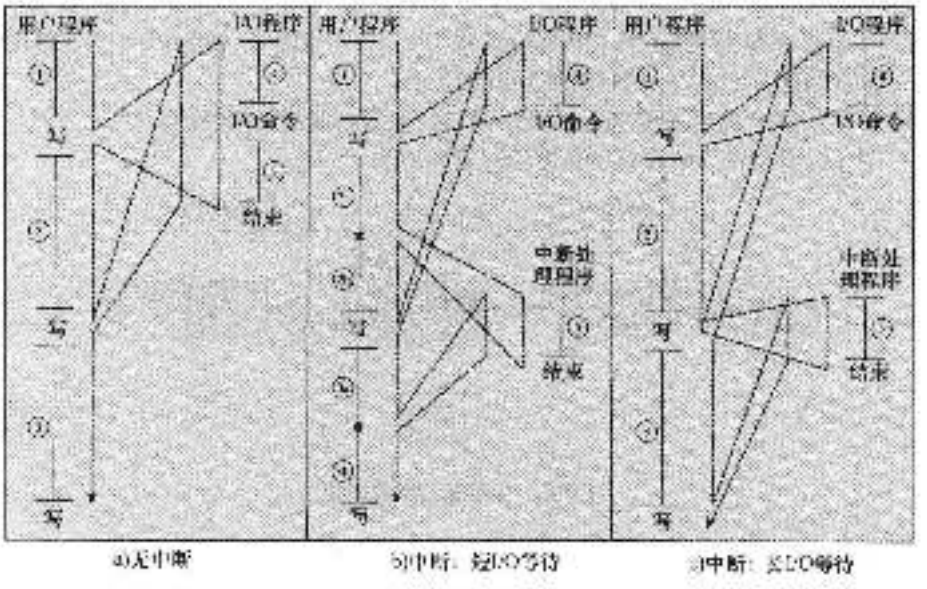
表 3-1 中断的分类

中断类型	产生的原因
程序	由指令执行结果伴随而出现的某些条件所产生，例如，算术溢出、除以零、企图执行一些非法的机器指令或访问的地址超出了用户存储器空间
定时器	由处理器中的定时器产生，它允许操作系统以规整的时间间隔执行特定的功能
I/O	由 I/O 控制器产生，以通知操作的正常完成或各种出错情况
硬件故障	由电源故障或存储器奇偶校验出错这类的故障产生

提供中断主要是为了提高处理的效率。例如，大部分外设比处理器慢很多。假如处理器使用如图 3-3 所示的指令周期的方法将数据传送给打印机，在每次写操作之后，处理器都会暂停并处于空闲状态，直到打印机跟上进度。暂停的时间可能有几百个甚至上千个指令周期，这还不包括存储器。显然，这是处理器使用上的巨大浪费。

图 3-7a 说明了事件的这种状态。用户程序执行一系列 WRITE 调用，WRITE 调用与处理过程交错进行。代码段 1、2、3 是指不包含 I/O 操作的指令序列。WRITE 调用是对一段 I/O 程序的调用，它是执行实际 I/O 操作的系统实用程序。这段 I/O 程序包含以下三个部分：

- 用于为实际 I/O 操作准备的指令序列，在图中标记为 4。它可能包括将待输出数据复制到专用的缓冲区，以及为设备命令准备参数。



a)无中断　　b)中断：短I/O等待　　c)中断：长I/O等待

图 3-7 有中断和无中断的程序控制流

- 实际的 I/O 命令。如果没有中断的使用，一旦此命令发出，程序必须等待 I/O 设备完成需要的功能（或周期性地测试设备）。程序可以通过简单地重复执行一个测试操作来决定该 I/O 操作是否完成。
- 完成该操作的指令序列，在图中标记为 5。这可能包含设置标志位来表示操作是成功还是失败。

因为 I/O 操作可能需要花较长的时间才能完成，所以 I/O 程序挂起，等待操作完成；于是，用户程序在 WRITE 调用这一点暂停很长一段时间。

1. 中断和指令周期

有了中断，处理器就可以在 I/O 操作进行的时候执行其他指令。考虑如图 3-7b 所示的控制

流。与前面相同，用户程序到这一点，以 WRITE 调用的形式进行系统调用。此时，被调用的I/O程序仅仅由准备代码和实际的 I/O 命令组成。当这几条指令执行后，控制权返回给用户程序。同时外部设备忙于从计算机存储器中接收数据并打印。这次 I/O 操作与用户程序中的指令执行同时进行。

当外部设备准备好接收服务，即当它准备从处理器中接受更多的数据时，外部设备的 I/O 模块发送**中断请求信号**给处理器。处理器通过挂起当前程序的操作，跳转服务于某个特定 I/O 设备的程序来响应，这个程序被称为**中断处理程序**，并且在设备服务完后恢复原来的执行。中断发生的断点在图 3-7a 中用星号（*）表示。

从用户程序的角度来看，中断只是打断正常的执行序列。当中断处理完成之后，恢复执行原来的序列（如图 3-8 所示）。因此，用户程序不必为适应中断而提供任何特殊代码。处理器和操作系统负责用户程序挂起和在中断点处恢复等操作。

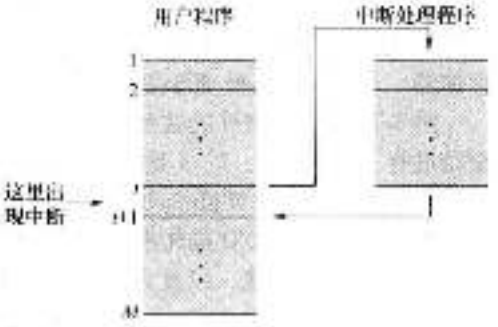


图 3-8 发生中断时的控制转换

为适应中断，将**中断周期**加入指令周期中，如图 3-9 所示。在中断周期中，处理器检查是否发生了中断，这将由中断请求信号的出现来指示。如果没有中断请求，则处理器继续进入取指周期，读取当前程序的下一条指令。如果出现中断请求，则处理器执行以下操作：

- 挂起当前正在执行的程序，并保存其状态。这意味着保存下一条即将执行的指令的地址（程序计数器 PC 的当前内容）以及任何与处理器当前活动相关的数据。
- 将程序计数器设置为**中断处理程序**的起始地址。

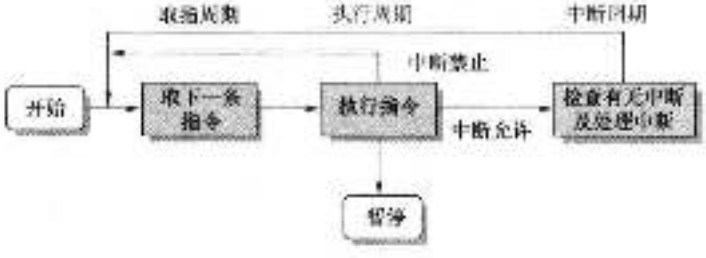


图 3-9 有中断的指令周期

处理器现在进入取指周期，并读取服务于该中断的中断处理程序的第 1 条指令。中断处理程序一般是操作系统的一部分。通常中断处理程序判定中断的性质并且完成所需要的任何操作。在我们已经使用过的例子中，中断处理程序判断是哪个 I/O 模块产生了中断，然后跳转到向 I/O 模块写数据的程序。中断处理程序结束后，处理器能够在断点处恢复用户程序的执行。

显然，这一处理增加了系统的开销。必须执行额外的指令（在中断处理程序中）来判断中断的性质并决定相应的操作。然而，由于简单地等待 I/O 操作要浪费大量的时间，因此中断的使用能使处理器更有效地运行。

为了评价获得的效率增益，考虑图 3-10，它是基于图 3-7a 和图 3-7b 控制流的时序简图。图 3-7b 和图 3-10 假设 I/O 操作所需的时间较短：比完成用户程序中与操作之间的指令执行时间要短。更典型的情况，对于打印机之类的慢速设备，I/O 操作的时间要比执行一系列用户指令的时间长很多，图 3-7c 给出了这种事件的状况。在这种情况下，用户程序在第 1 次 WRITE 调用产生的 I/O 操作完成之前，就面临第 2 次 WRITE 调用，结果是用户程序在这一点挂起。当前一个 I/O

操作完成之后，才可能处理新的 WRITE 调用，启动新的 I/O 操作。图 3-11 显示了这种情况下使用中断和未使用中断的时序。无论从哪种情况都可以看出，效率得到了提高，因为部分 I/O 操作的时间与用户指令执行的时间相重叠。

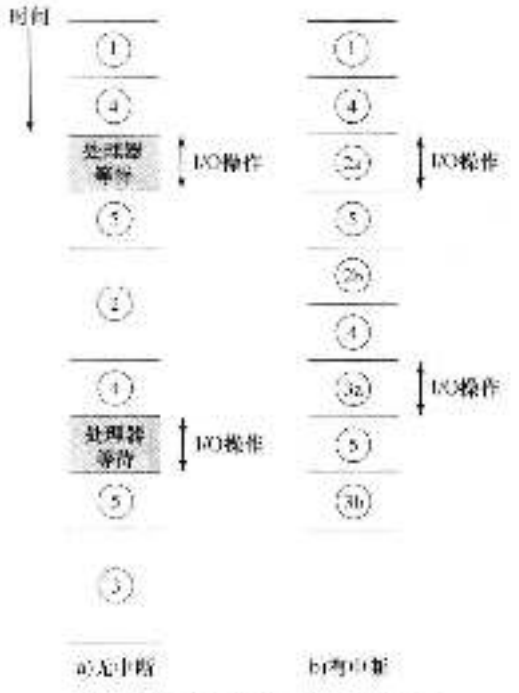


图 3-10 程序时序：短 I/O 等待

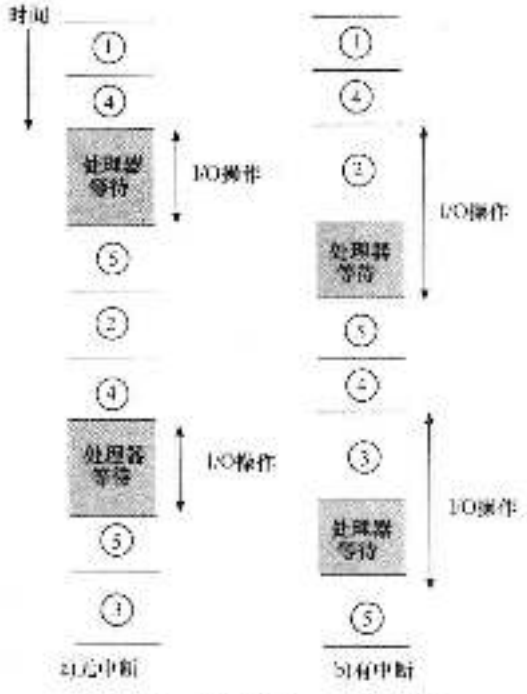


图 3-11 程序时序：长 I/O 等待

图 3-12 显示了修改后包含中断周期处理的指令周期状态图。

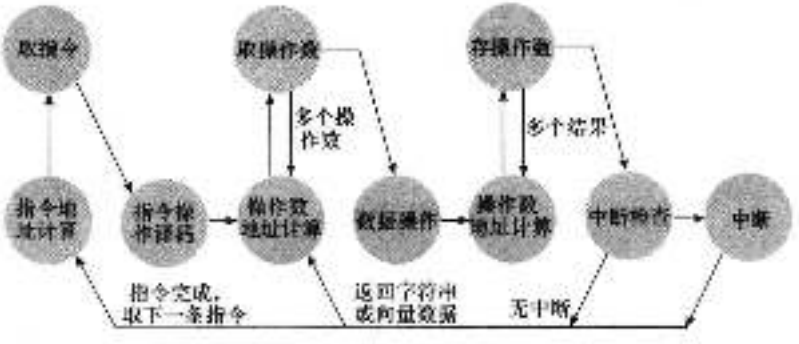


图 3-12 有中断的指令周期状态图

2. 多重中断

迄今为止，只讨论了发生单个中断的情形。但是实际的情况是可能产生多个中断。例如，一个程序可以从通信线路接收数据和打印结果。打印机每完成一次打印操作便会产生一次中断。每次到达一个单元的数据，通信线路控制器将产生一次中断。根据通信规程，一个单元可以是一个字符，也可以是一个数据块。任何情况下，在打印机中断处理时都可能再发生通信中断。

处理多重中断有两种方法。第 1 种是在中断处理过程中禁止其他中断。禁止中断仅仅意味着处理器可以并且将忽略中断请求信号。如果中断在此时发生，一般会保持在“未决状态”，在处理器允许中断后就会检测到这种未决状态。于是，当用户程序执行时如果有一个中断发生，则该

中断会立即被禁止。在中断处理程序完成后，不用等到用户程序恢复就可以再次允许中断，这时候处理器检查是否发生了其他中断。这种方法既简单又有效，因为中断严格按顺序处理（如图 3-13a所示）。

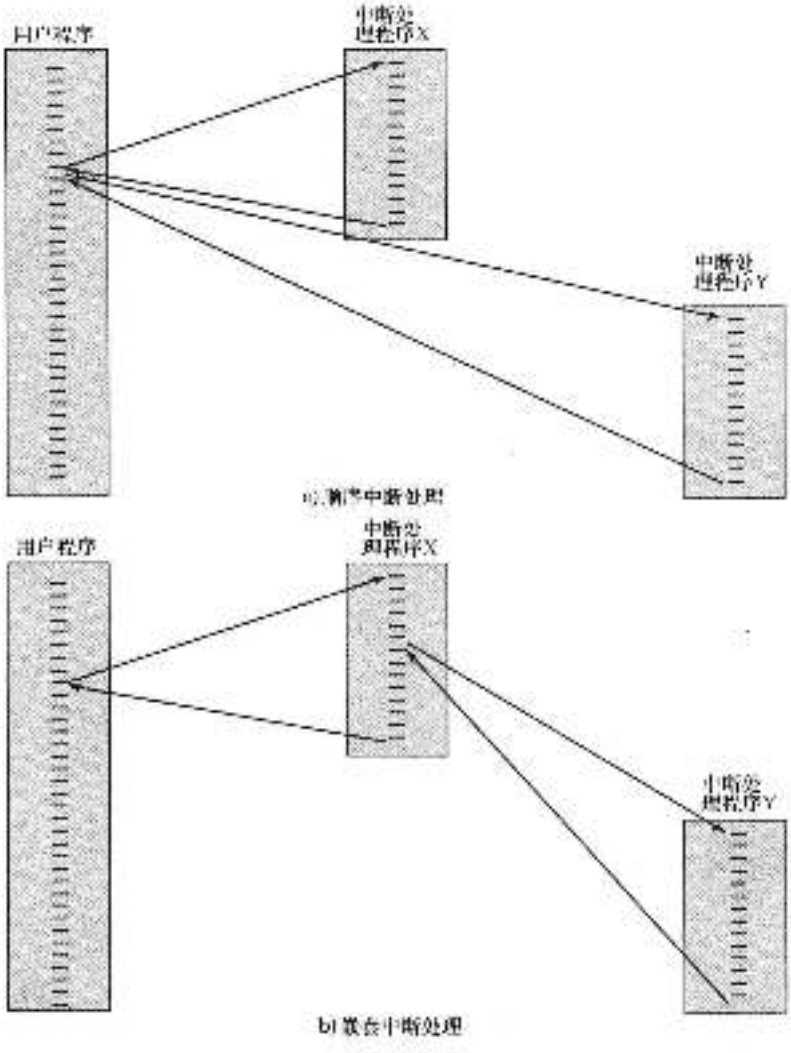


图 3-13 带多重中断的控制转换

上面这种方法的缺点是没有考虑到相对的优先级或时间紧迫的需要。例如，当输入数据从通信线路到达时，需要被迅速接收，以便给更多的输入腾出空间。如果第 2 批数据到达之前第 1 批数据还没有处理，就可能丢失数据。

第 2 种方法是定义中断的优先级，且允许优先级高的中断引起低级中断处理程序本身被中断（如图 3-13b 所示）。作为第 2 种方法的例子，考虑一个有 3 个 I/O 设备的系统：打印机、硬盘和通信线路，它们的优先级逐个递增，分别是 2、4、5。基于文献 [TANE97] 中的例子，图 3-14 举出了一个可能的序列。用户程序开始于 $t=0$ 时刻。当 $t=10$ 时，发生了打印机中断，用户信息放入系统栈，并继续从打印机的中断服务程序（ISR）开始执行。当程序仍在执行时，在 $t=15$ 时刻，通信中断发生。由于通信线的优先级比打印机高，这个中断得到响应。打印机 ISR 被中断，它的状态压入栈，继续从通信 ISR 执行。当这个通信 ISR 正在执行时，发生了磁盘中断（$t=20$）。由于它的优先级相对较低，只好挂起，而通信 ISR 运行到结束。

当通信 ISR 完成时（$t=25$），原来的处理器的状态恢复，即执行原来的打印机 ISR。但是在这一例程中的一条指令都没有来得及执行以前，处理器响应优先级更高的磁盘中断，将控制权传送给磁盘 ISR。仅当这一例程结束后（$t=35$），打印机的 ISR 才恢复。在它完成后（$t=40$），控制权才最终交还给用户程序。

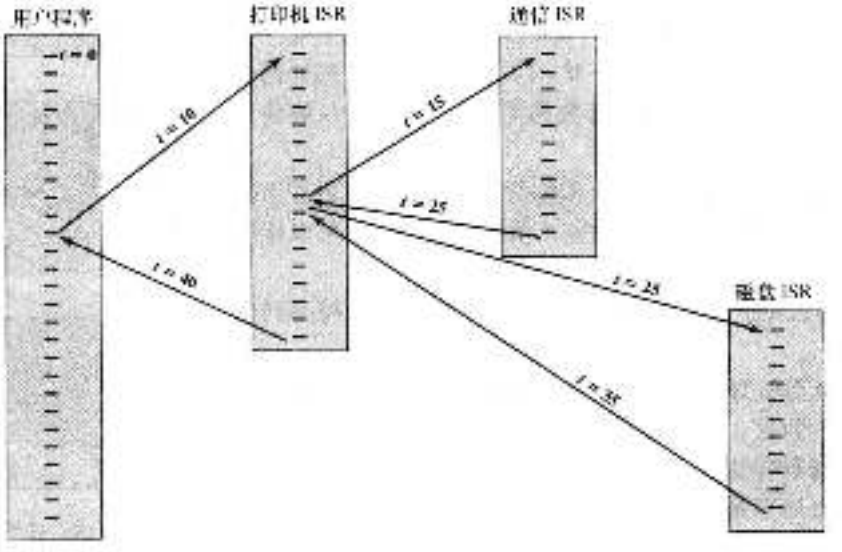


图 3-14 多重中断的时序示例

3.2.3 I/O 功能

至此，我们已经讨论了由处理器控制的计算机操作，而且主要考察了处理器和存储器的交互操作。前面的讨论只提到了 I/O 部件的作用，这一作用将在第 7 章中详细讨论，但这里有必要给出一个简单的介绍。

I/O 模块（例如磁盘控制器）能直接与处理器交换数据。如同处理器通过指定某个单元地址就可以启动一次对存储器的读/写一样，处理器同样也能把数据写到 I/O 模块或从 I/O 模块中读出。在后一种情况中，处理器识别由特定 I/O 模块控制的特定设备。因此，会发生与图 3-5 中形式相似的指令序列，只不过它们是 I/O 指令，而不是存储器访问指令。

在某些情况下，需要允许 I/O 直接与存储器交换数据。在这种情况中，处理器授予 I/O 模块读或写存储器的权利，以使 I/O 和存储器的传输不需要 CPU 的介入。在这种传输中，I/O 模块向存储器发出读或写的命令，解脱了处理器负责数据交换的责任。这种操作称为直接存储器访问（DMA），将在第 7 章中详细论述。

3.3 互连结构

计算机包含一组部件或 3 种基本类型的模块（处理器、存储器和 I/O），模块之间相互通信。实际上，计算机是一个基本模块的网络，因此必须有连接这些模块的通路。

连接各种模块的通路的集合称为互连结构（interconnection structure）。这种结构的设计将取决于模块之间所必须进行的交换。

图 3-15 通过指出每种模块类型的主要输入、输出形式给出了所需的信息交换的种类[1]：

- **存储器**：通常，存储器模块由 N 个等长的字组成，每个字分配了一个唯一的数值地址（0、

[1] 图中，箭头代表多根信号线，它们并行携带信息的多位。每个箭头代表单个信号线。

1、…、$N-1$），数据字可以从存储器中读出或写进存储器。操作的性质由读和写控制信号指示。操作的单元由地址指定。

- I/O 模块：从（计算机系统）内部的观点看，I/O 在功能上与存储器相似。它们都有读和写两类操作。此外，一个 I/O 模块可以控制多个外设，我们可以定义每个与外部设备的接口为端口（port），给它分配一个唯一的地址（例如，0、1、…、$M-1$）。此外，还有向外部设备输入和输出数据的外部数据路径。最后，I/O 模块可以给处理器发送中断信号。
- 处理器：处理器读入指令和数据，并在处理之后写出数据。它还用控制信号控制整个系统的操作。也可以接收中断信号。

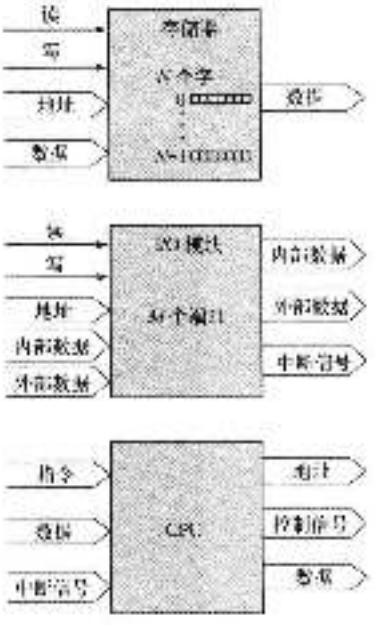


图 3-15 计算机模块

前面所列定义了要交换的数据。互连结构必须支持下列类型的传送：

- 存储器到处理器：处理器从存储器中读一条指令或一个单元的数据。
- 处理器到存储器：处理器向存储器写一个单元的数据。
- I/O 到处理器：处理器通过 I/O 模块从 I/O 设备中读数据。
- 处理器到 I/O：处理器向 I/O 设备发送数据。
- I/O 与存储器之间：对于这两种情况，I/O 模块允许与存储器直接交换数据，使用直接存储器存储（DMA），而不通过处理器。

多年来，人们尝试过各种各样的互连结构，迄今为止最普遍的是总线和各种多总线结构。本章后续部分将专门讨论总线结构。

3.4 总线互连

总线（bus）是连接两个或多个设备的通信通路。总线的关键特征是共享传输介质。多个设备连接到总线上，并且任何一个设备发出的信号可以被其他所有连接到总线上的设备所接收。如果两个设备同时发送，它们的信号将会重叠，这样会引起混淆。因此，每次只能有一个设备能够成功地利用总线发送数据。

通常，总线由多条通信路径或线路组成。每条线能够传送代表二进制 1 和 0 的信号。一段时间里，一条线能传送一串二进制数字。总线的几条线放在一起，就能够用来同时（并行地）传送二进制数字。例如，一个 8 位的数据能通过总线中的 8 条线传送。

计算机系统含有多种总线，它们在计算机系统的各个层次提供部件之间的通路。连接计算机的主要部件（处理器、存储器、I/O）的总线称为系统总线。最常见的计算机互连结构是基于一个或多个系统总线的使用。

3.4.1 总线结构

系统总线通常包含 50 到上百条分立的导线。每条导线被赋予一个特定的含义或功能。虽然总线的设计有多种，但任何总线的线路都可以分成如下 3 个功能组（如图 3-16 所示）：数据线、地址线和控制线。此外，还有为连接的模块提供电源的电源线。

数据线提供系统模块间传送数据的路径。这些线组合在一起称为数据总线。典型的数据总线包含 32、64、128 或更多的分离导线，这些线的数目称为数据总线的宽度。因为每条线每次能

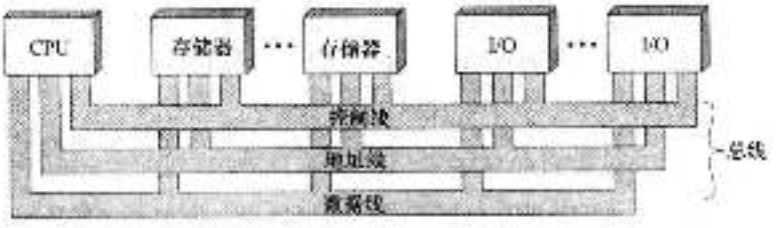


图 3-16　总线互连策略

传送 1 位，所以线的数目决定了每次能同时传送多少位。数据总线宽度是决定系统总体性能的关键因素。例如，如果数据总线为 32 位宽，而每条指令 64 位长，那么处理器在每个指令周期必须访问存储器模块两次。

地址线用于指定数据总线上数据的来源或去向。例如，如果处理器希望从存储器中读取一个字（8 位、16 位或 32 位），它将所需要的字的地址放在地址线上。显然，地址总线的宽度决定了系统能够使用的最大的存储器容量。而且，地址线通常也用于 I/O 端口的寻址。通常，地址线的高位用于选择总线上指定的模块，低位用于选择模块内具体的存储器单元或 I/O 端口。例如，在一个 8 位地址总线上，小于等于 01111111 的地址可以用来访问有 128 个字的存储器模块（模块 0），而大于等于 10000000 的地址可用来访问接在 I/O 模块上的设备（模块 1）。

控制线用来控制对数据线和地址线的存取和使用。由于数据线和地址线被所有模块共享，因此必须用一种方法来控制它们的使用。控制信号在系统模块之间发送命令和时序信号。时序信号指定了数据和地址信号的有效性，命令信号指定了要执行的操作。典型的控制信号如下所列：

- **存储器写**（Memory Write）：引起总线上的数据写入被寻址的单元。
- **存储器读**（Memory Read）：使所寻址单元的数据放到总线上。
- **I/O 写**（I/O Write）：引起总线上的数据输出到被寻址的 I/O 端口。
- **I/O 读**（I/O Read）：使被寻址的 I/O 端口的数据放到总线上。
- **传输响应**（Transfer ACK）：表示数据已经从总线上接收，或已经将数据放到总线上。
- **总线请求**（Bus Request）：表示模块需要获得对总线的控制。
- **总线允许**（Bus Grant）：表示发出请求的模块已经被允许控制总线。
- **中断请求**（Interrupt Request）：表示某个中断正在悬而未决。
- **中断响应**（Interrupt ACK）：未决的中断请求被响应。
- **时钟**（Clock）：用于同步操作。
- **复位**（Reset）：初始化所有模块。

总线的操作如下，如果一个模块希望向另一个模块发送数据，它必须做两件事情：（1）获得总线的使用权；（2）通过总线传送数据。如果一个模块希望向另一个模块请求数据，它也必须做两件事情：（1）获得总线的使用权；（2）通过适当的地址线和控制线向另一模块发送请求，然后它必须等待另一模块发送数据。

从物理上讲，系统总线实际上是多条平行的电导线。这些导线是在卡或板（印刷电路板）上刻出来的金属线。总线延伸至所有系统部件，每一个系统部件都连接到总线的全部或部分线。图 3-17 描绘了其典型的物理布置。在这个例子中，总线由两组垂直的导体组成。沿着导体每隔一段就有一组水平伸出插

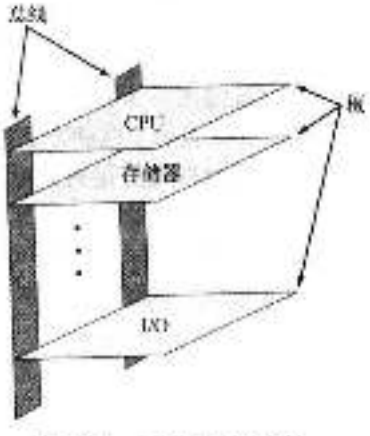


图 3-17　总线结构的典型物理实现形式

槽形式的连接点，这种插槽用以支持印刷线路板。每个主要系统部件都占用一个或多个插插板，并从这些插中插入总线。整个装置被安在底架上。这种方式仍可以用于连接一个计算机系统的部分总线。然而，当代系统倾向于使用像处理器那样的芯片，芯片上有更多的元器件，从而能使所有的主要部件都在同一块板上。因此，一个片上总线可连接处理器和 cache，而一个板上总线可连接处理器到主存和其他部件。

这种布置是最方便的。小配置的计算机系统以后可以无过增加更多的板来扩展（更多的存储器、更多的 I/O）。如果板上的某个部件出现了故障，将这块板拔下并替换即可。

3.4.2 多总线层次结构

如果太多的设备连到总线上，性能将会下降。这主要有两个原因：

（1）总的来说，总线上连接的设备越多，总线长度就越长，传输延迟就越大，而这个延迟决定了设备协调总线使用所花费的时间。当总线控制频繁地由一个设备传送到另一个设备时，传输延迟显著地影响性能。

（2）当聚集的传输需求接近总线容量时，总线便会成为瓶颈。通过提高总线的数据传输率或使用更宽的总线（例如，将数据总线由 32 位增加到 64 位），这个问题在某种程度上可以得到缓解。然而，由扯接设备（例如，图形和视频控制器、网络接口）产生的数据速度快速增长，所以这是单一总线最终注定要失败的竞赛。

因此，多数计算机系统都使用多总线，通常布置为层次结构。图 3-18a 显示了典型的传统总线结构。局部总线连接处理器和高速缓存，它可用于支持一个或多个局部设备。高速缓存控制器不仅将高速缓存连接到该局部总线，而且将它连到接入了所有主存储器模块的系统总线。正如第 4 章中将要讨论的一样，高速缓存的使用避免了处理器对主存储器的频繁访问。因此，主存储器可以从局部总线移到系统总线。同样，I/O 和主存储器通过系统总线来传送信息，不影响处理器的活动。

将 I/O 控制器直接连接到系统总线上是可行的。更高效的方法是，为此使用一个或多个扩展总线。系统总线与挂在扩展总线上的 I/O 控制器之间的数据传输可以通过扩展总线接口来进行缓冲。这种方案允许系统支持更广泛的各种 I/O 设备，同时将存储器到处理器的传输与 I/O 传输隔开。

图 3-18a 显示了某些可以连接到扩展总线上的 I/O 设备的典型实例。网络连接包含局域网（例如 10Mb/s 的以太网）和广域网（例如分组交换网）。SCSI（小型计算机系统接口）本身就是一种总线，它支持本地磁盘驱动器和其他外设。串行口可用来支持串行打印机或扫描仪。

这种传统的总线结构比较有效，但在性能越来越高的 I/O 设备面前，它开始显得力不从心。为了满足这些不断增长的需要，业界采用的普遍方法是构造与系统其余部分紧密集成的高速总线，而它仅要求一个在处理器总线和高速总线之间的桥。有时称这种方案为中间层结构。

图 3-18b 表示了这种方法的典型实现，它由有连接处理器和高速缓存控制器的局部总线，而高速缓存控制器又连接到支持主存储器的系统总线上。高速缓存控制器集成到连接高速总线的桥或缓冲设备中。这个总线支持如 100Mb/s 快速以太网这样的高速 LAN、视频和图形工作站控制器以及包括 SCSI 和 FireWire 局部外设总线的接口控制器，后者是专门用来支持大容量 I/O 设备的高速总线。低速设备仍然由扩展总线支持，以接口来缓冲扩展总线和高速总线之间的传输量。

这种配置的好处是，高速总线使高需求的设备与处理器有更紧密的集成，同时又独立于处理器。因此，不同的处理器和高速总线速度及信号线定义都可以兼容。处理器结构的变化不影响高速总线，反之亦然。

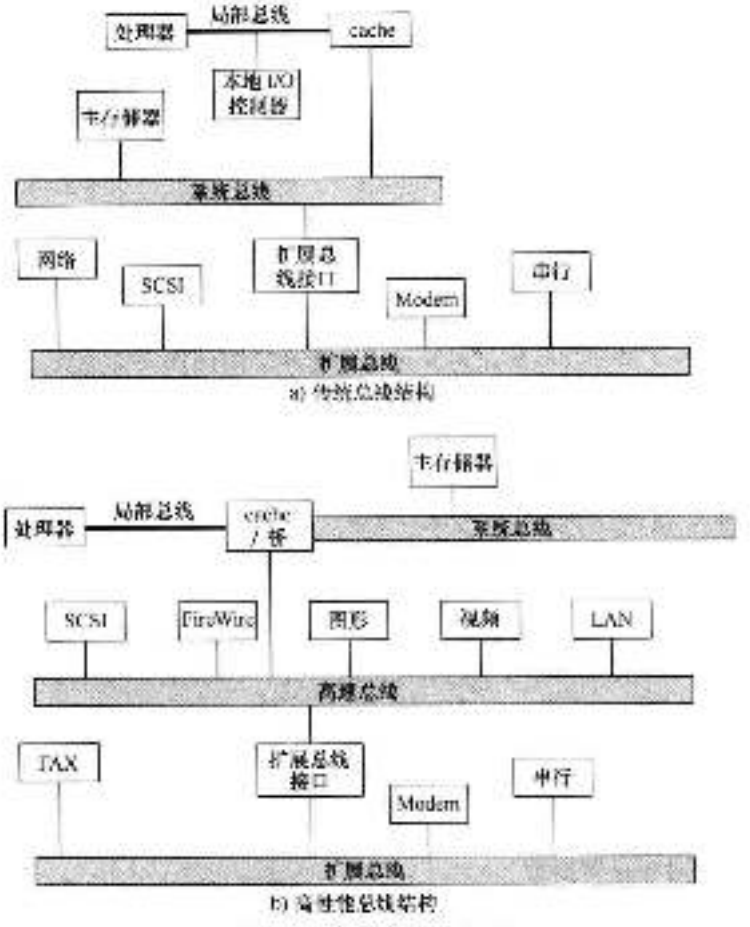


图 3-18　总线配置的实例

3.4.3　总线的设计要素

虽然存在着各种不同的总线实现方案，但也有几个用来区分总线的基本参数或设计要素。表 3-2 列出了其中的关键要素。

表 3-2　总线的设计要素

类型	仲裁方法	时序	总线宽度	数据传输类型
专用/复用	集中式/分布式	同步/异步	地址/数据	读/写/读-修改-写/写后读/块

1. 总线类型

总线可以分为两种基本类型：专用的和复用的。专用总线始终只负责一项功能，或始终分配给计算机部件的一个物理子集。

功能专用的一个例子是使用分立专用的地址线和数据线。这种情况在许多总线中很常见，但这不是必须的。例如，使用一个地址有效（address valid）控制线，地址和数据信息就可以通过同一组线来传输。在数据传送的开始，地址放到总线上，地址有效线被激活。在这一点，每个模块在规定的一段时间内复制地址，并判断自己是否是被寻址的模块。然后地址从总线上撤销，相同的总线连线随后用于读或写数据的传输。这种将相同的线用于多种目的的方法称为分时复用。

分时复用的优点是使用的布线数较少，从而节省了空间和成本；其缺点是在每个模块中需要更复杂的控制电路，而且还可能导致性能降低，因为共享总线的特定事件不能同时发生。

物理专用总线指的是使用多条总线，每一总线仅与模块的一个子集相连接。一个典型的例子是使用 I/O 总线连接所有的 I/O 模块，然后这一总线再通过 I/O 适配器模块连到主总线上。物理专用的潜在优点是总线冲突减少，所以具有高吞吐量；缺点是增加了系统的规模和成本。

2. 总线的仲裁方法

在除最简单系统以外的所有系统中，可能有不止一个模块需要控制总线。例如，一个 I/O 模块可能需要直接读或写存储器，而不是通过 CPU 发送数据。由于在总线上每次只有一个器件能够成功发送，因此需要某种仲裁方法。各种仲裁方法可大致划分为“集中式”或“分布式”两类。在集中式的方法中，被称为总线控制器（或仲裁器）的硬件设备负责分配总线时间。这个设备可以是独立的模块，也可以是 CPU 的一部分。在分布式的方法中，没有中央控制器，而是在每个模块中包含访问控制逻辑，这些模块共同作用，分享总线。这两种仲裁方法的目的都是为了指定一个设备（CPU 或 I/O 模块）作为主控器。然后这个主控制器启动与其他设备的数据传输（例如读或写），后者在这一次数据交换中作为从属设备。

3. 时序

时序是指总线上协调事件的方式，总线使用同步时序或异步时序。

对于同步时序，总线上事件的发生由时钟决定。总线包含时钟信号（clock）线，它传送相同长度的由 0、1 交替的规则信号组成的时钟序列。一次 1-0 传送称为时钟周期或总线周期，它定义了一个时间槽。总线上其他所有设备都能读取时钟线，而且所有事件都在时钟周期的开始时发生。图 3-19 显示了一个典型的、但简单的同步读和写操作的时序图（见附录 3A 对时序图的描述）。其他总线信号可以在时钟上升沿处发生改变（稍有反应延迟）。大多数事件占用一个时钟周期。在这个简单的例子中，处理器在第 1 个时钟周期将存储器地址放到地址总线上，并且可以声明各种状态线。一旦地址线上的信号稳定，处理器就会发出一个地址允许信号。对于读类作，处理器在第 2 个时钟周期开始时发出一个读命令。存储器模块识别地址，在延迟 1 个周期后，将数据放到数据线上。处理器从数据线上读取数据，并且撤销读信号。对于写操作，处理器在第 2 个周期开始时将数据放到数据线上，并且在数据线稳定后发出一个写命令。存储器模块在第 3 个时钟周期从数据线上复制信息。

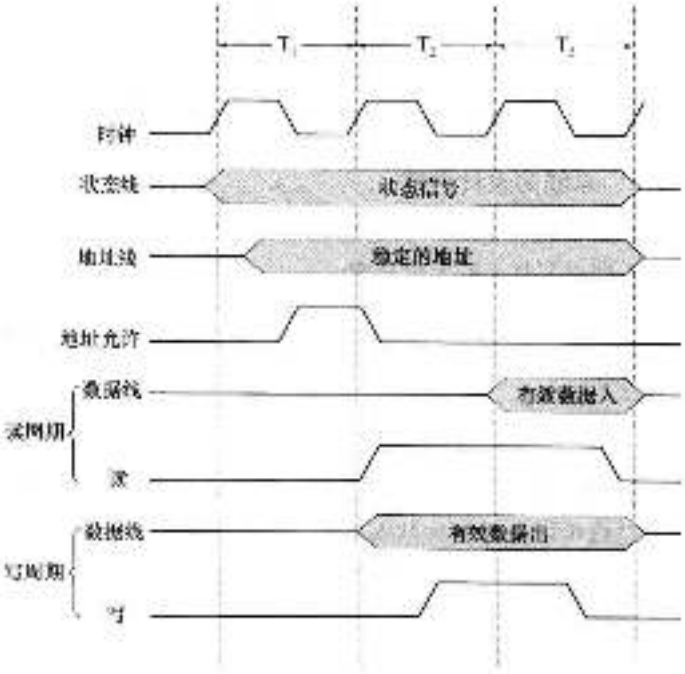


图 3-19 同步总线操作时序

对异步时序来说，总线上一个事件的发生取决于前一个事件的发生。在图 3-20a 读周期的简单例子中，处理器发送地址信号和状态信号到总线上。待这些信号稳定后，它发出读命令，指示有效地址和控制信号的存在。相应的存储器模块译码地址并将数据放到数据线上。一旦数据线上的信号稳定，则存储器模块使确认信号有效，以便通知处理器数据可用。

一只处理器从数据线上读取数据后，它立即撤除读信号。这会引起存储器模块撤除数据和确认信号。最后，一旦确认信号撤除，则处理器撤除地址信息。

图3-20b给出一个简单的异步写操作例子。在此情况中，总线的主控者将数据放到数据线上，与此同时启动状态线和地址线。存储器模块通过从数据线上复制数据来响应写命令，并使确认线上的信号有效。然后，主控者撤除写命令信号，而存储器模块撤除确认信号。

同步时序的实现和测试都很简单，但同步时序没有异步时序灵活，因为同步总线上的所有设备都遵循固定的时钟频率，系统不能发挥高性能设备的优势。对于异步时序来说，不论设备是快还是慢，使用的技术是新还是旧，都可以共享总线。

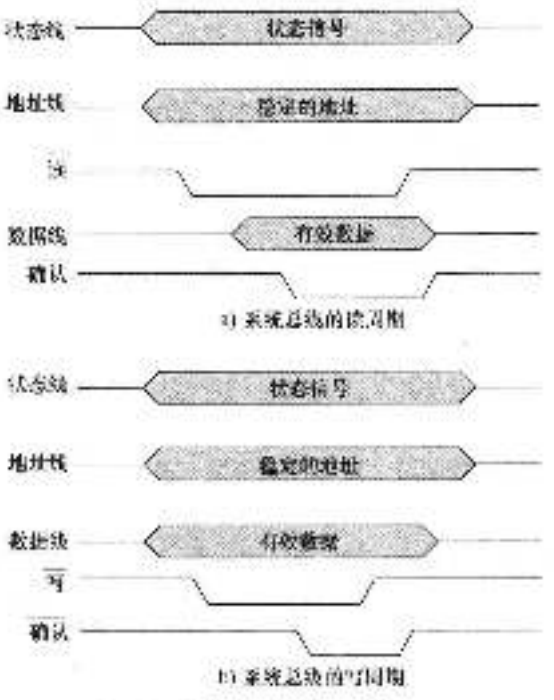


图3-20 异步总线操作时序

4. 总线宽度

前面已经介绍了总线宽度的概念。数据总线的宽度对系统性能有重要影响：数据总线越宽，一次能传送的位数就越多。地址总线的宽度对系统容量有重要影响：地址总线越宽，可以访问的单元就越多。

5. 数据传输类型

总线支持各种数据传输类型，如图3-21所示。所有的总线都支持写（主控制器到从属设备）和读（从属设备到主控制器）的传输。在复用型地址/数据总线中，总线先用于指定地址，然后用于传输数据。对于读操作，数据由从属设备取得并放到总线上时，典型的情况是有一个等待。无论是读还是写，如果有必要通过仲裁为其余的操作获得总线的控制权（也就是说，先占有总线来请求读写，然后再一次占有总线执行读写），则同样存在延迟。

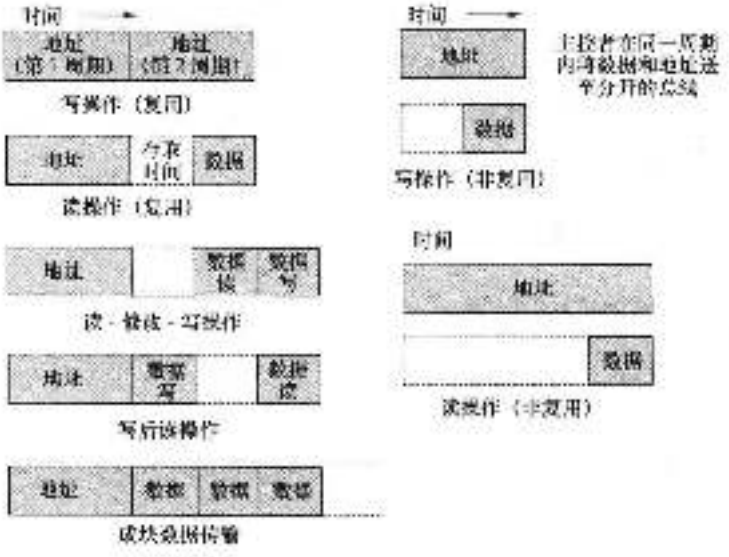


图3-21 总线数据传输类型

在专用的地址和数据总线中，地址放到地址线上并一直保持到数据出现在数据线上。对于写操作，地址一旦稳定，主控制器就把数据放到数据线上，这时从属设备已经有机会识别其地址。对于读操作，从属设备一旦识别出地址并准备好数据，就将数据放到数据总线上。

某些总线允许几种组合操作。"读-修改-写"操作是在读之后紧接着向同一地址写数据。地址仅在操作的开始广播一次。为了防止其他潜在的主控制器访问此数据单元，整个操作是不可分的。这一原则是为了在多道程序系统中保护共享的存储器（参见第 8 章）。

"写后读"也是一种不可分割的操作，它是指写之后紧接着对同一地址的读取。这个读操作可用于校验。

有些总线系统还支持数据块的传输。在这种情况中，一个地址周期后面跟着 n 个数据周期。第 1 个数据传输使用指定的地址，其余的数据项传输使用后续地址。

3.5 PCI

外设部件互连（PCI）总线是一种高带宽、独立于处理器的总线，它能够作为中间层或外围设备总线。与其他普通的总线规范相比，PCI 为高速的 I/O 子系统（例如，图形显示适配器，网络接口控制器、磁盘控制器等）提供了更好的性能。当前的标准允许在 66MHz 的频率下使用多达 64 根数据线。从理论上讲，其速率可达到 528MB/s 或者 4.224Gb/s。然而，PCI 的优势不仅仅是它的高速度。PCI 是专门为满足现代系统的 I/O 要求而设计的较经济的总线。实现它只需很少的芯片，而且它支持将其他总线连到 PCI 总线上。

Intel 在 1990 年开始为其 Pentium 系统开发 PCI。很快，Intel 将所有的专利向外界公开，并促进了工业协会，PCI SIG（PCI 特别兴趣组）的创建，它的任务是进一步开发并维护 PCI 规范的兼容性。结果是 PCI 被广泛地采纳，越来越多地应用到个人计算机、工作站以及服务器系统中。由于这个规范是公开的，而且它得到了许多微处理器和外围设备生产商的支持，因此不同生产商的 PCI 产品是相互兼容的。

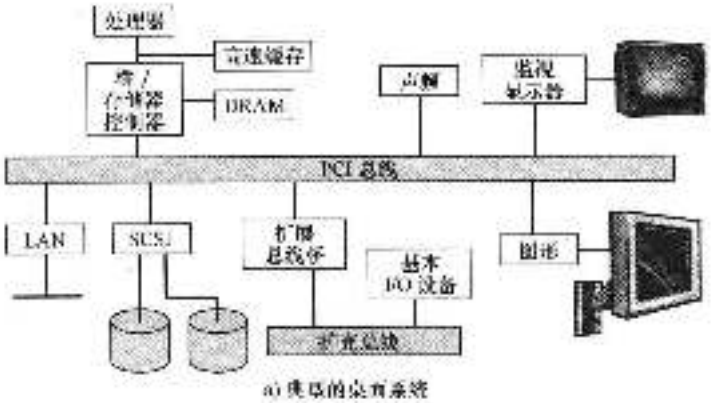
PCI 支持各种基于处理器的配置，包括单处理器和多处理器系统。相应地，它提供一组通用的功能，并使用同步时序以及集中式仲裁机制。

图 3-22a 表示了在单处理器系统中使用 PCI 的一个典型例子。DRAM 控制器与到 PCI 总线的桥接器相结合，提供了与处理器更紧密的耦合，同时提供高速传输数据的能力。这个桥接器扮演着"数据缓冲"的角色。这样，PCI 总线的速度可以与处理器的 I/O 处理器速度不同。在多处理器系统中（如图 3-22b 所示），一个或者多个 PCI 总线可通过桥接器连接到处理器所在的系统总线上。系统总线只支持处理器/高速缓存单元、主存储器以及 PCI 桥接器。而且，桥接器的使用保证了 PCI 独立于处理器，同时又能提供快速传送或接收数据的能力。

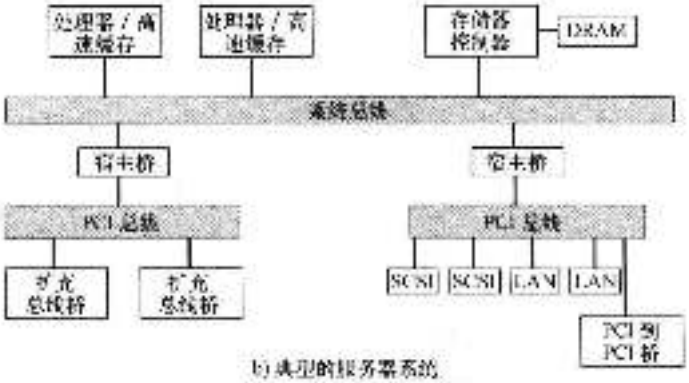
3.5.1 总线结构

PCI 可以配置成 32 位或 64 位总线。表 3-3 定义了 49 线 PCI 所需的信号线，按照功能分为以下几组：

- **系统引脚**：包括时钟和复位引脚。
- **地址和数据引脚**：包括 32 根分时复用的地址线和数据线。在这一组中，其他信号线用来解释并使传送的地址和数据的信号线有效。
- **接口控制引脚**：控制数据交换的时序，并提供发送端和接收端的协调。
- **仲裁引脚**：不同于其他 PCI 信号线，它们是不共享的线，每个 PCI 主控制器有自己的一对仲裁线，它们直接连到 PCI 总线仲裁器上。
- **错误报告引脚**：用于报告奇偶校验位以及其他的错误。



a) 典型的桌面系统



b) 典型的服务器系统

图 3-22　PCI 配置实例

表 3-3　必备的 PCI 信号线

符号名称	中文名称	类型	说　明
			系统引脚
CLK	时钟信号	in	所有交换的时钟基准，其上升沿被所有的输入所采样，支持的最高时钟速率为 33MHz
RST#	复位信号	in	强迫所有的 PCI 专用的寄存器、定序器和信号转为初始状态
			地址和数据引脚
AD[31::0]	地址/数据线	t/s	用于地址和数据的复用引脚
C/BE[3::0]#	命令/字节允许信号	t/s	复用的总线命令和字节允许信号，在传送数据阶段，这 4 条线指示 4 个字节通路中的哪个字节有效
PAR	检验信号	t/s	比 AD 和 C/BE 延后一个时钟周期，提供“偶校验”。主控制器（Master）在写数据阶段驱动 PAR，而接收响应器在读数据阶段驱动 PAR
			接口控制引脚
FRAME#	成为数信号	s/t/s	当前的主控制器用其指示本次传输开始并在整个传输中保持有效。FRAME#在传输开始时有效（呈低电平），并当发送响应器开始最后的数据交换阶段时撤销
IRDY#	发送端就绪	s/t/s	由当前的总线主控制器（交换的发送端）驱动，在读操作周期，表示主控制器已准备好接收数据；在写操作周期，表示有效数据已放到地址/数据线上

（续）

符号名称	中文名称	类型	说 明
		接口控制引脚	
TRDY#	目标就绪	s/t/s	由目标（被选中的设备）驱动，在读操作中，表示有效数据已放到地址/数据线上；在写操作期间，表示目标已准备好接收数据
STOP#	停止信号	s/t/s	表示当前的目标希望发送端暂停当前的交换
IDSEL	初始化设备选择	in	可初始化被读/写设备，在读和写交换时作为片选信号
DEVSEL#	设备选择	in	由目标驱动，表明其有效地址时有效，用以向当前的发送端指示设备是否被选中
		仲裁引脚	
REQ#	请求信号	t/s	向仲裁器申请使用总线。它是一条设备专用的点对点的信号线
GNT#	允许信号	t/s	仲裁器准许请求设备使用总线，它是一条设备专用的点对点的信号线
		错误报告引脚	
PERR#	奇偶校验错	s/t/s	表示在写数据阶段目标检测到一个数据的奇偶校验错，或在读数据阶段由发送端检测到奇偶校验错
SERR#	系统错误	o/d	可由任何设备发出，用以报告地址奇偶检验错和奇偶校验以外的其他严重错误

此外，PCI 规范还定义了 51 个可选的信号线（如表 3-4 所示），它们分为以下几个功能组：

- **中断引脚**：它们提供必须请求服务的 PCI 设备。同仲裁引脚一样，它们也是不共享的。每个 PCI 设备有自己的中断线或连接到中断控制器的线。
- **高速缓存支持引脚**：需要用这些引脚来支持在处理器或其他设备中能被高速缓存的 PCI 上的存储器。这些引脚支持高速缓存监听协议（参见第 8 章有关这个协议的讨论）。
- **64 位总线扩展引脚**：包括 32 根分时复用的地址线和数据线。它们与必有的地址/数据线一起，形成 64 位的地址/数据总线。这一组中的其余线用于解释传送该地址和数据的信号并使之有效。最后，还有使两个 PCI 设备具备 64 位能力的两根线。
- **JTAG/边界扫描引脚**：这些信号线支持 IEEE 标准 1149.1 中定义的测试程序。

表 3-4 可选的 PCI 信号线

符号名称	中文名称	类型	说 明
		中断引脚	
INTA#	中断请求 A	o/d	用于请求中断
INTB#	中断请求 B	o/d	用于请求中断，仅对多功能设备有意义
INTC#	中断请求 C	o/d	用于请求中断，仅对多功能设备有意义
INTD#	中断请求 D	o/d	用于请求中断，仅对多功能设备有意义
		cache 支持引脚	
SDO#	监听后退	in/out	表示命中某个已经修改的行
SDONE	监听完成	in/out	表示当前监听状态，当前的监听完成时有效
		64 位总线扩展引脚	
AD[63::32]	扩展的地址/数据线	t/s	将总线扩展为 64 位的地址和数据复用线

（续）

引脚名称	中文名称	类型	说　明
		64 位总线扩展引脚	
C/BE[7::4]#	扩展的命令/字节允许信号	t/s	总线命令和字节允许信号的复用线。在地址阶段，这些线提供了额外的总线命令。在数据阶段，这些线用于指示 4 个扩展的字节通道中哪几个有效
REQ64#	64 位请求信号	s/t/s	用于请求 64 位的传输
ACK64#	64 位响应信号	s/t/s	表示目标将执行 64 位传输
PAR64	64 位校验信号	t/s	在 AD 和 C/BE 后一个时钟周期，提供扩展的 64 位偶校验
		JTAG/边界扫描引脚	
TCK	测试时钟	in	在边界扫描阶段，用于为状态信息和测试数据输入、输出设备提供时钟
TDI	测试输入	in	用于串行地将数据和指令移入设备
TDO	测试输出	out	用于串行地将数据和指令移出设备
TMS	测试模式选择	in	用于控制测试访问端口控制器的状态
TRST#	测试复位	in	用于初始化测试访问端口控制器

注：in 单向输入信号。
out 单向输出信号。
t/s 双向、三态、I/O 信号。
s/t/s 每次只能由一个拥有者驱动的持续三态信号。
o/d 集电极开路，允许多个设备通过“线或”共享连接。
低电平有效的信号。

3.5.2 PCI 命令

在发送端（或称为主控方）与目标端之间进行信息交换时，总线活动开始。当总线的主控方获得总线控制权时，它决定即将发生的传送类型。在地址传送周期，C/BE（命令/字节）信号线用来指示传送类型。这些命令包括：中断响应、特殊周期、I/O 读、I/O 写、存储器读、存储器读行、存储器读多行、存储器写、存储器写和无效、配置读、配置写、双地址周期。

中断响应是一条读命令，其目的是为了使设备作为 PCI 总线上的中断控制器。此时，地址传送周期中地址线并不使用，而字节允许信号指示可返回的中断标识号。

特殊周期命令用来由发送端向一个或多个目标广播消息。

I/O 读和 I/O 写命令用来在发送端和 I/O 控制器之间传送数据。每个 I/O 设备有其自己的地址空间，其地址线用于指定特定的设备，并向指定设备发送或从设备接收的数据。I/O 地址的概念将在第 7 章深入讨论。

存储器读和存储器写命令用于激发数据的突发式传输，传输占用一个或多个时钟周期。这些命令的解释依赖于 PCI 总线上的内存控制器是否支持 PCI 协议在存储器和高速缓存之间进行传输。如果支持，则与存储器之间的数据传输一般以高速缓存的行或块[⊖]来进行。3 条存储器读命令的用途如表 3-5 所示。存储器写命令用于向存储器传送数据，它占用一个或多个数据周期。存储器写和无效命令也用一个或多个周期来给存储器传送数据。而且，它保证至少一个高速缓存行是写操作。这条命令支持向存储器回写一行的高速缓存功能。

两个配置命令使主控制器能够读和更新与 PCI 相连的设备的配置参数。每个 PCI 设备可能包含最多 256 个内部寄存器，它们用来在系统初始化时配置设备。

⊖ 高速缓存的基本原理将在第 4 章中介绍，基于总线的高速缓存协议将在第 17 章中描述。

表 3-5 PCI 读命令的解释

读命令类型	可高速缓存类型	不可高速缓存类型
存储器读	突发传送半个或不到一个高速缓存行的数据	突发 2 个数据传输周期或更少
存储器行读	突发传送半个到 3 个高速缓存行的数据	突发 3 ~ 12 个数据传输周期
存储器多读	突发传送 3 个以上的高速缓存行的数据	突发超过 12 个数据传输周期

双地址周期命令由发送端来指示它使用 64 位寻址。

3.5.3 数据传送

PCI 总线上的每个数据传送都是由一个地址周期和一个或多个数据周期组成的单一事项。在本讨论中，我们说明一个典型的读操作，写操作的过程与之相似。

图 3-23 显示了读事项的时序。所有事件均在时钟的上升沿同步，即在每个时钟周期的中央发生。总线设备在总线周期开始时的上升沿对总线路采样。下面是图 3-23 中所标出的重要事件：

(a) 一旦总线的主控制器获得总线控制权，它通过使 FRAME 有效（降为低电平）来启动事务，FRAME 线将一直保持有效，直到发送端就绪完成最后一次数据传送。FRAME 有效之后，发送端将起始地址放到地址总线上，并且将读命令放在 C/BE 信号线上。

(b) 当 CLK 2 开始时，目标设备将会识别 AD 线上的地址信号。

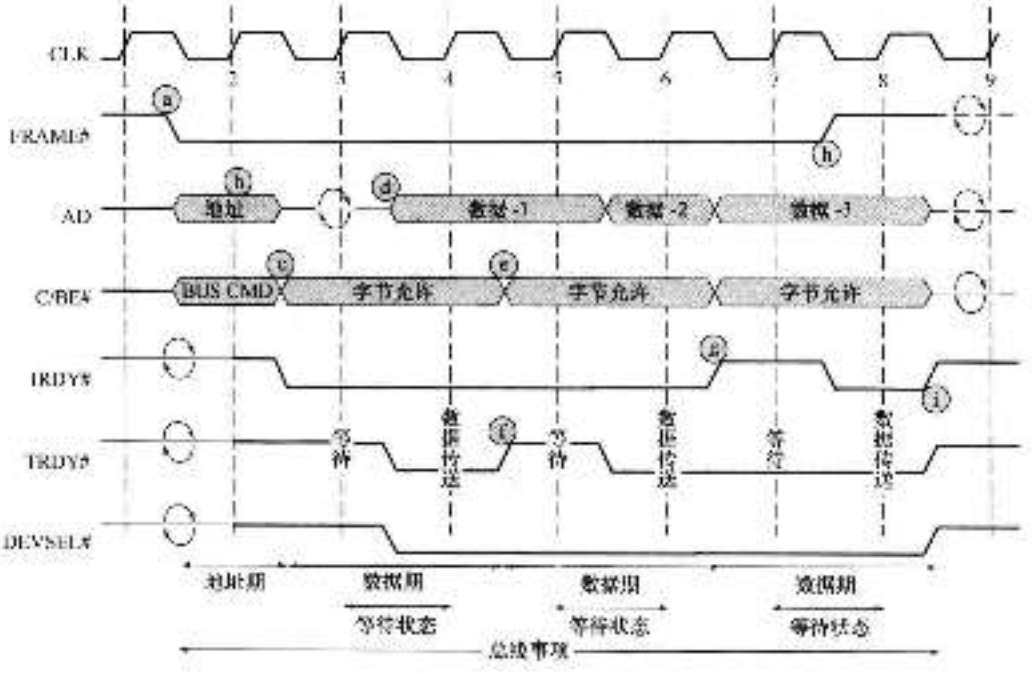


图 3-23 PCI 读操作

(c) 发送端停止驱动 AD 总线。所有可能由多个设备驱动的信号线要求一个切换周期（用两个环形的箭头来表示），这样，地址信号会失效而使总线被目标设备所用。发送端修改 C/BE 线上的信息，以指示 AD 线传送数据中哪些字节有效（从 1 ~ 4 字节）。发送端同时使 IRDY 降为低电平（有效），表示它已准备接收第 1 个数据项。

(d) 被选中目标的 DEVSEL 信号线有效（降为低电平），表示收到了有效的地址，并将响应。目标设备将所要求的数据放到 AD 总线上并使 TRDY 信号有效（降为低电平），表示已将有

效数据放在总线上了。

(e) 发送端在 CLK 4 的开始时读数据，并改变字节允许信号，为下一次读操作做准备。

(f) 在这个例子中，目标设备需要一些时间为第 2 块数据的传输做准备。因此，它将 TRDY 置为无效（高电平），以通知发送端下一周期没有新的数据。相应地，发送端在 CLK 5 开始时不会读数据，并且在这个周期中不改变字节允许信号。第 2 块数据在 CLK 6 开始时被读取。

(g) 在 CLK 6 周期中，目标将第 3 块数据项放到总线上。但是在这个例子中，发送端还没有准备好读数据（例如，它暂时处在缓冲区满的状态）。于是，它使 IRDY 无效（变为高电平），迫使目标设备的第 3 块数据项在总线上多保持一个时钟周期。

(h) 发送端知道传送的第 3 块数据是最后一个，因此它使 FRAME 变为高电平（无效），以通知目标这是最后一次数据传送，同时发送端再次使 IRDY 降为低电平（有效），表示它已准备好完成这次传送。

(i) 完成传输后，发送端取消 IRDY，使总线变回空闲状态。同时目标使 TRDY 和 DEVSEL 处于无效状态。

3.5.4 仲裁

PCI 使用集中式的同步仲裁方法，主控方有独立的请求（REQ）信号和允许（GNT）信号。这些信号线连接到中央仲裁器上（如图 3-24 所示），它使用一种简单的“请求-允许”的握手联络方式来访问总线。

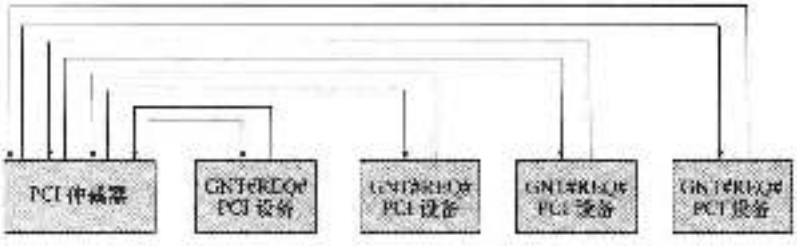


图 3-24 PCI 总线仲裁器

PCI 规范没有规定具体的仲裁算法。仲裁器能够使用“先到先服务”的方法、“轮转”方法或某种优先方法。PCI 主控方必须为它期望完成的每个事项进行仲裁，单一事项包含一个地址周期，后面跟着一个或多个数据周期。

图 3-25 给出设备 A 和 B 为使用总线进行仲裁的例子，其仲裁的时序如下所述：

(a) 在 CLK 1 开始前，A 已经产生了低电平的 REQ 信号（有效），仲裁器在 CLK 1 的开始时采样到这一信号。

(b) 在 CLK 1 中，B 的 REQ 信号变为低电平（有效），也请求使用总线。

(c) 同时，仲裁器输出的 GNT-A 变为低电平，允许 A 访问总线。

(d) 在 CLK 2 开始时，总线主控方 A 采样到 GNT-A 信号，知道它已经被允许访问总线。同时它发现 IRDY 和 TRDY 此时仍为无效的高电平，表明此时总线是空闲状态。因此，A 使 FRAME 信号有效（降为低电平），并将地址信息放到地址总线上，将命令放到 C/BE 总线上（没有画出）。同时 A 继续使 REQ-A 保持有效，因为它随后要执行第二个事项。

(e) 总线仲裁器在 CLK 3 的开始时对所有 REQ 线采样，并决定将总线准许给 B 进行下一事项，然后，它声明 GNT-B 有效并撤销 GNT-A。但 B 只有等到总线恢复为空闲状态时，才能够使用总线。

(f) A 撤销 FRAME，表示正在进行最后一次数据传输，同时把数据放到数据总线上，并通

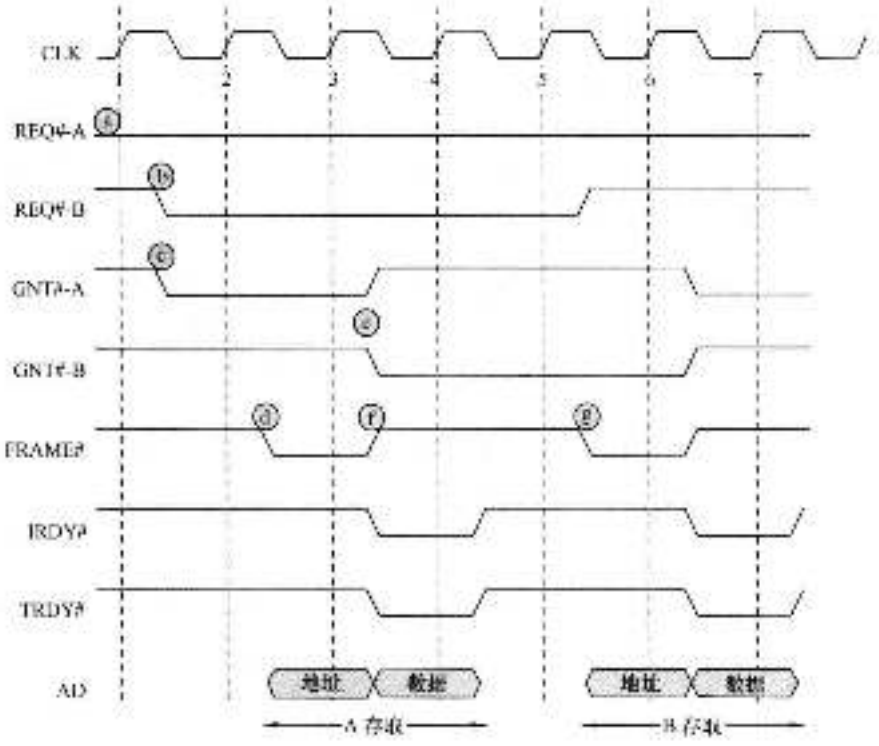


图 3-25 两个主控方之间的 PCI 总线仲裁

过 IRDY 降为低电平，通知目标设备，使目标设备在下一个时钟周期开始时读数据。

（g）在 CLK 5 时钟周期开始时，B 发现 IRDY 和 FRAME 已经呈无效状态，因此将 FRAME 置为低电平取得总线的控制。同时它取消 REQ，因为它只想完成一个事项。

随后，主控方 A 被授予总线访问权，以进行下一事项。

注意，仲裁可以在当前总线主控方进行数据传送的时候同时发生，因此，总线仲裁不浪费总线周期，这称为隐式仲裁。

3.6 推荐的读物和 Web 站点

最清晰地以全书篇幅介绍 PCI 的一本书是 [SHAN99]。[ABO04] 同样包含了许多关于 PCI 的实质信息。

ABO04 Abbot, D. *PCI Bus Demystified*. New York: Elsevier, 2004.
SHAN99 Shanley, T., and Anderson, D. *PCI Systems Architecture*. Richardson, TX: Mindshare Press, 1999.

推荐的 Web 站点

PCI Special Interest Group：提供 PCI 规范和产品的有关信息。
PCI Pointers：指向 PCI 厂商和其他信息源的链接。

3.7 关键词、思考题和习题

关键词

address bus：地址总线
asynchronous timing：异步时序
bus：总线
bus arbitration：总线仲裁

bus width：总线宽度
centralized arbitration：集中式仲裁
data bus：数据总线
disable interrupt：禁止中断
distributed arbitration：分布式仲裁
instruction cycle：指令周期
instruction execute：指令执行
instruction fetch：取指令
interrupt：中断
interrupt handler：中断处理
interrupt service routine：中断服务程序
memory address register（MAR）：存储器地址寄存器
memory buffer register（MBR）：存储器缓冲寄存器
peripheral component interconnect（PCI）：外设部件互连
synchronous timing：同步时序
system bus：系统总线

思考题

3.1 计算机指令指定的功能通常分为哪几类？
3.2 列出并简要定义指令执行的可能状态。
3.3 列出并简要说明多重中断的两种处理方法。
3.4 计算机互连结构（如总线）必须支持何种类型的传送？
3.5 与单总线相比，使用多总线有什么好处？
3.6 列出并简要定义 PCI 信号线的功能组。

习题

3.1 图 3-4 所示的假想机器同样有两个 I/O 指令：
0011 = 从 I/O 装入 AC
0111 = 将 AC 内容存入 I/O
在上述约定下，使用 12 位的地址来标识一个特定的 I/O 设备。给出下列程序的执行过程（用图 3-5 的格式）：
(1) 从设备 5 装入 AC。
(2) 与内存单元 940 的内容相加。
(3) 将 AC 存入设备 6。
假设从设备 5 提取的下一个值为 3，而内存单元 940 的内容为 2。
3.2 本书中使用 6 步来描述图 3-5 的程序执行。请解释这些步骤以说明 MAR 和 MBR 的作用。
3.3 考虑一个假想的 32 位微处理器采用 32 位的指令格式，这种指令有两个部分：第 1 个字节包含操作码，其余部分是立即操作数或操作数的地址。
(a) 最大可能直接寻址的存储器容量是多少（以字节为单位）？
(b) 讨论下面的微处理器总线对系统的影响：
(1) 32 位局部地址总线和 16 位局部数据总线。
(2) 16 位局部地址总线和 16 位局部数据总线。
(c) 程序计数器和指令寄存器需要多少位？
3.4 考虑一个假想的微处理器，它产生 16 位地址（例如，假设程序计数器和地址寄存器都是 16 位），并具有 16 位数据总线。
(a) 如果处理器连接到“16 位存储器”，那么它能直接访问的最大存储器地址空间是多少？
(b) 如果处理器连到“8 位存储器”，那么它能直接访问的最大存储器地址空间是多少？
(c) 结构上的什么特点允许处理器访问独立的“I/O 空间”？
(d) 如果输入和输出指令能够指定一个 8 位的 I/O 端口号，那么处理器能支持多少个 8 位的 I/O 端口？它能支持多少个 16 位的 I/O 端口？请解释。
3.5 考虑一个 32 位微处理器，它有 16 位外部数据总线，由 8MHz 的输入时钟驱动。假设该处理器的总线周期占最小持续时间等于 4 个输入时钟周期，这个处理器利用总线能够维持的最大数据传输率是多少（字节/秒）？如果将其外部数据总线扩展为 32 位，或使微供给处理器的外部时钟频率加倍，能否提高它的性能？请按速所做其他假设的理由，并加以解释。提示：确定每个总线周期所能传送的字节数。
3.6 考虑一个计算机系统，它包括控制简单的键盘/电传打印机的 I/O 模块。下列寄存器包含在 CPU 中，

并且直接连接到系统总线上。
INPR：输入寄存器（8 位）
OUTR：输出寄存器（8 位）
FGI：输入标志（1 位）
FGO：输出标志（1 位）
IEN：中断允许（1 位）
从电传的按键输入到电传的打印机输出，均由 I/O 模块控制。电传能够将字母符号编码成 8 位的字，并将 8 位的字解码为字母符号。
（a）描述该处理器如何能通过使用本题中给出的前 4 个寄存器来完成与电传的输入、输出。
（b）描述这一功能如何能通过使用 IEN 更有效地完成。

3.7 考虑两个微处理器，除了一个使用 8 位宽的外部数据总线而另一个使用 16 位宽的外部数据总线之外，两者没有其他不同。它们的总线周期也是等长的。
（a）假设所有指令和数据都是两字节长，则它们的最大数据传输率差几倍？
（b）假设指令是一字节长，数据是 2 字节长，重复上述问题。

3.8 图 3-26 表示一种分布式的仲裁方法，它可用于 Multibus I 的旧总线方式中。各单元按优先级次序在物理上以菊花链的方式连接。图中最左边的单元持续接收总线优先权输入（BPRN）信号，表示没有更高优先级的单元需要使用总线。如果这个单元不需要使用总线，则它声明自己的总线优先权输出（BPRO）线。在时钟周期的开始，任何单元都可以使其 BPRO 信号线为低来请求总线的控制。这使菊花链中的下一单元的 BPRN 线为低，而这个单元也相应地降低自己的 BPRO 线。因此，信号就沿着菊花链传播。在链式反应的末尾，应只有一个单元，它的 BPRN 有效，而 BPRO 无效，于是这个单元获得了优先权。如果在总线周期的开始，总线处于空闲状态（BUSY 线无效），则获得优先权的单元可以通过声明 BUSY 线来夺取总线的控制权。

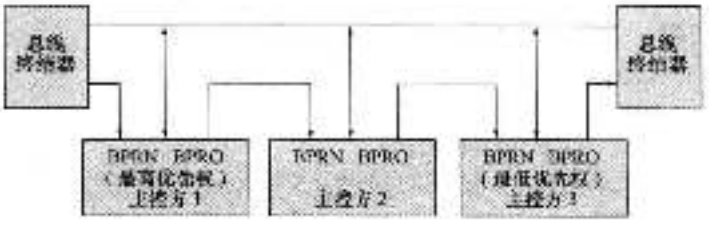


图 3-26 Multibus I 分布式仲裁

BPR 信号从具有最高优先级的单元传播到具有最低优先级的单元需要一定的时间。这段时间必须比时钟周期小吗？为解释原因。

3.9 VAX SBI 总线使用分布式同步仲裁方案。每个 SBI 设备（例如处理器、存储器、I/O 模块）有唯一的优先级，而且被分配一根独立的传输请求（TR）线。SBI 有 16 根这样的线（TR0、TR1、…、TR15）。其中 TR0 的优先级最高。当一个设备想要使用总线时，它通过在当前时间片中声明 TR 线来预约未来的时间片。在当前时间片结束时，每个具有预约请求的设备检查 TR 线，其中具有最高优先级的设备使用下一个时间片。
最多可有 17 个设备连接到总线上，优先级为 16 的设备没有 TR 线，请说明原因。

3.10 在 VAX SBI 机上，具有最低优先级的设备通常有最低的平均等待时间。因此，CPU 在 SBI 中通常被赋予最低的优先级。为什么优先级为 16 的设备有最低的平均等待时间？这在什么条件下不成立？

3.11 对于同步读操作（如图 3-19 所示），存储器模块必须在 Read 信号下降沿前有充分的时间将数据放到总线上，以允许信号稳定下来。假设定数处理器总线时钟频率是 10MHz，而 Read 信号在 T_2 后一半的中间开始下降。
（a）确定存储器读指令周期的长度。
（b）存储器至少应何时将数据放到总线上？假定数据线稳定需要 20ns。

3.12 考虑一个具有图 3-19 的存储器读时序的微处理器。进行一些分析后，设计者确认存储器提供的读数据大约落后了 180ns。
（a）若总线时钟频率为 8MHz，要进行恰当的系统操作，需要插入多少等待状态（时钟周期）？
（b）为了实施等待状态，使用了 Ready（就绪）状态线。一旦处理器发出一个 Read 命令，它必须等待直到 Ready 线有效后才能试图读数据。要指定处理器插入所要求的等待状态数，Ready 线必须

在什么时间间隔内保持为低（无效）？

3.13 某微处理器具有图 3-19 所示的存储器写时序。它的厂家指出，Write 信号的宽度能用 7-50 来确定，T 是以 ns 为单位的时钟周期。

(a) 若总线时钟速率是 5MHz，则我们预期的 Write 信号的宽度是多少？

(b) 该微处理器的数据资料指出，在 Write 信号下降沿之后，数据要继续保持 20ns 的时间有效。提供给存储器的有效数据总的持续时间是多少？

(c) 如果存储器需要提交的数据至少在 190ns 时间内有效，则应插入几个等待状态？

3.14 某微处理器具有对存储器单元内容（单元的值）增 1 的直接指令。此指令有 5 个步骤：取操作码（4 个总线时钟周期），取操作数地址（3 个周期），取操作数（3 个周期），对操作数加 1（3 个周期）和存操作数（3 个周期）。

(a) 如果每次存储器读或写操作都要插入两个总线等待状态，则此指令的周期将增加多少（以百分数计）？

(b) 若对操作数加 1 操作不是 3 而是 13 个周期，试重做问题（a）。

3.15 Intel 8088 微处理器具有类似于图 3-19 的读总线时序，但要求 4 个处理器时钟周期。总线上有效数据持续时间已延长到包括第 4 个时钟周期。假设处理器的时钟速率是 8MHz。

(a) 最大的数据传送速率是多少？

(b) 若每字节的传送都需要插入一个等待状态，试重做问题（a）。

3.16 Intel 8086 是一个许多方面类似于 8088 的 16 位处理器，它使用 16 位总线，能一次传送 2 字节，以低字节有偶地址为前提。然而，8086 除了使用偶对齐的字之外，也允许使用奇对齐的字。若存取一个奇对齐的字，则传送一个字需要两个存储器周期，每个存储器周期由 4 个时钟周期组成。考虑一条涉及两个 16 位操作数的 8086 指令，取这些操作数需用多少时间？给出可能答案的范围。假定时钟速率是 4MHz 和无等待状态。

3.17 考虑一个 32 位的微处理器，其总线周期与一个 16 位微处理器的总线周期相同。假定，平均而言，20% 的操作数和指令是 32 位长，40% 的为 16 位长，其余 40% 的为 8 位长。请计算，此 32 位微处理器取指令或操作数时所实现的性能改进。

3.18 如习题 3.14 的微处理器，它正在执行一条增量存储器直接指令。若在取操作数步骤开始的同时，键盘激活了一条中断请求线，假定总线时钟速率是 10MHz 则该处理器多长时间之后才进入中断处理周期？

3.19 画出 PCI 写操作的时序图，并解释之（与图 3-23 相似）。

附录 3A 时序图

在本章中，时序图用于表示事件序列和事件之间的依赖关系。对于不熟悉时序图的读者，本附录提供了简单的解释。

连接到总线上的设备之间的通信，通过一组能够传递信号的线来实现。这些线可以传送两种不同的信号级别（电压级别，电平），分别代表二进制的 0 和 1。时序图能够将线上的信号电平作为时间的函数来表示（如图 3-27a 所示）。习惯上，二进制 1 用比二进制 0 高的电平表示。通常，二进制 0 是默认值，也就是说，如果不传输数据和其他信号，则线上的电平代表二进制 0。一个从 0 到 1 的信号转变一般称为信号上升沿（leading edge），而从 1 到 0 的转变称为下降沿（trailing edge）。这种转变不是瞬间发生的，但其转变时间与信号的持续时间相比通常非常小。为清晰起见，通常将该转变画成一条带角的线，但该线夸大了转变实际花费的时间。偶尔，你会看到使用垂直线画的图，这并不建议认为信息的转变是瞬间发生的。时序图中，在我们感兴趣的事件之间可能要经过一段变化的或至少与问题无关的时间，用时间线上的一条间隙来表示。

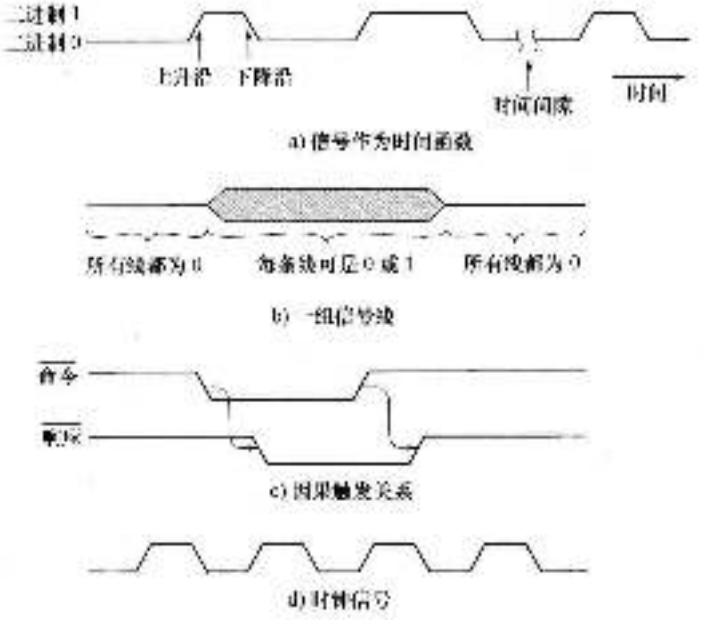
信号有时成组表示（如图 3-27b 所示）。例如，如果一次传送一个字节的数据，则需要 8 条线。通常，我们只想知道信号是否出现，而这一组线中确切的值对我们并不重要。

一条线上的信号跳变可能触发与之相连的设备改变其他线上的信号。例如，如果存储器模块检测到一个读控制信号（0 或 1 跳变），它将把数据信号放到数据线上。这样的因果关系产生了事件的序列。时序图上的箭头用于表示它们的依赖关系（如图 3-27c 所示）。

在图 3-27c 中，信号名称上的横线条表示该信号低电平有效。例如，Command在 0 状态是有效，这意味

若$\overline{\text{Command}}=0$表示逻辑1或真。

时钟线通常是系统总线的一部分。一个时钟电路接到时钟线上并提供重复的，规则的跳变序列（如图3-27d所示）。其他事件可能与时钟信号保持同步。



a) 信号作为时间函数

b) 一组信号线

c) 因果触发关系

d) 时钟信号

图3-27 时序图

第4章

Computer Organization and Architecture: Designing for Performance, 8E

cache 存储器

本章要点

- 计算机的存储器被组织成层次结构。最顶层（靠着处理器的一层）是处理器为数有限的寄存器。接下来是一级或多级的高速缓存，当使用多级 cache 时，它们分别标记为 L1、L2 等。再往下的是主存，它通常由动态随机存取存储器（DRAM）构成，所有这些都被认为是系统内部的存储器。存储层次继续划分外部存储器，下一层通常是固定硬盘，再往下的是可装卸的存储设备，如光盘、磁带机等。
- 沿着存储器层次结构自顶向下，存储器成本也逐层下降，其容量在变大，而存取时间在变长。只使用最快的存储器当然好，但是它也是最昂贵的，因此我们通过使用更多较慢的存储器，以便达到存取时间与成本之间的均衡。其中的技巧是，在存储器中恰当地组织程序和数据，使需要存取的数据通常在较快的存储器中。
- 通常，处理器将要访问的主存位置很有可能是刚被访问过的或其临近的位置，所以 cache（高速缓存）会自动地保存一些从主存最近使用过的 DRAM 字的副本。如果 cache 设计得合理，那么大多数时候处理器所需要的存储器数据便已经在 cache 中。

计算机的存储器虽然从概念上来看比较简单，但是从计算机系统的类型、技术、组织、性能和价格几方面的特点来看，存储器的范围或许是最广的。目前没有一种最佳的能满足计算机系统对存储器需求的技术。所以，计算机系统通常配备分层结构的存储子系统，一些在系统内部（由处理器直接存取），一些在系统外部（处理器通过 I/O 模块存取）。

本章和下一章将重点论述系统内部存储器部件，而第 6 章将专门论述外部存储器。本章首先介绍计算机存储器的关键特性，剩余部分讨论所有当代计算机系统所必备的部件——cache 存储器。

4.1 计算机存储系统概述

4.1.1 存储系统的特性

如果我们按照关键特性对存储系统进行分类，那么计算机存储器的复杂问题就会变得更易于管理，表 4-1 列出了存储系统最重要的一些特性。

表 4-1 计算机存储系统的关键特性

存储位置	内部（如寄存器、主存、cache），外部（如光盘、磁盘、磁带）
容量	字数，字节数
传送单元	字，块
存取方法	顺序存取，直接存取，随机存取，关联存取
性能	存取时间，周期时间，传输率
物理类型	半导体，磁介质，光介质，磁-光介质
物理特性	易失性/非易失性，可擦除/不可擦除
组织	存储模块

存储位置是指存储器处于计算机的内部或外部。内部存储器通常指主存，但还有其他形式。处理器需要有自己的局部存储器，它们以寄存器的形式存在（如图 2-3 所示）。进一步，我们就

会明白处理器的控制器部分也需要有自己的内部存储器。我们将在后面章节对这两类存储器进行分别讨论。cache 是内存存储器的另一种形式。外部存储器由外围存储设备（如磁盘、磁带等）组成，处理器可以通过 I/O 控制器访问它们。

存储器的一个明显特性是**存储容量**（capacity）。对于内部存储器，存储容量通常用字节（1 字节 = 8 位）或字来表示。普通的字长为 8 位、16 位或 32 位。外部存储器的存储容量通常也用字节来表示。

一个与之相关的概念是**传输单元**（unit of transfer）。对于内部存储器，传输单元等同于输入和输出到存储器模块的数据线数，它等于字长，但通常更大，如 64 位、128 位或 256 位。为了说明这一点，我们引入三个与内部存储器相关的概念：

- **字**：存储器组织的“自然”单元。字长通常与一个整数的数据位数和指令长度相等，但也有很多例外。例如，CRAY C90（一种较老的模型 CRAY 超级计算机）有 64 位的字长，但它用 46 位表示整数。而 Intel x86 体系结构有各种指令长度，用多个字节表示，但其机器的字长为 32 位。
- **可寻址单元**：在某些系统中，可寻址单元是字，但许多系统允许在字节级上寻址。在任何情况下，地址位长度 A 和可寻址的单元数 N 之间的关系为：$2^A = N$。
- **传输单元**：对于主存储器，这是指每次读出或写入存储器的位数。传输单元不必等于一个字或一个可寻址单元。对于外部存储器，数据的传送经常是以比一个字大得多的单元来传送，这就是所谓的块。

不同种类的存储器之间的另一个区别是数据单元的**存取方法**（method of accessing）不同，存取方法包括如下四类：

- **顺序存取**：存储器组织成许多称为记录的数据单元，它们以特定的线性序列方式存取。存储的地址信息用于分隔记录和帮助索引。采用共享读-写结构，经过一个个的中间记录，从当前的存储位置移动到所要求的位置，因此，存取不同记录的时间相差很大。第 6 章中讨论的磁带机采用的是顺序存取方式。
- **直接存取**：同顺序存取一样，直接存取也采用了共享读-写结构。但是，单个块或记录有基于物理存储位置的唯一地址。通过采用直接存取到达所需的块处，然后在块中顺序搜索，计数或等待，最终到达所要求的位置。同样，存取不同记录的时间相差很大。第 6 章中讨论的磁盘机系统采用的是直接存取方式。
- **随机存取**：存储器中每一个可寻址的存储位置有唯一的物理编排的寻址机制。存取给定存储位置的时间是固定的，不依赖于前面存取的序列。因此，任何存储位置可以随机选取，直接寻址和存取。主存和某些高速缓存系统采用随机存取方式。
- **关联存取**：这是一个随机存取类的存储器。它允许对一个字中的某些指定位进行检查比较，看是否与特定的样式相匹配，而且能同时在所有字中进行。因此，字是通过它的内容而不是它的地址进行检索。与普通的随机存取存取器相同，每个存储位置有自己的寻址机制，并且检索时间是固定的，不依赖于存储位置或前面的存取方式。高速缓存可以采用关联存取。

从用户的观点来看，存储器两种最重要的特性是容量和性能（performance），通常需要考虑 3 种性能参数：

- **存取时间（延迟）**：对于随机存取存储器，这是执行一次读或写操作的时间，即从地址传送给存储器的时刻到数据已经被存储或使用为止所花的时间。而对于非随机存取存储器，存取时间是把读-写结构定位到所需要的存储位置所花费的时间。
- **存储周期时间**：这个概念主要用于随机存取存储器，它是存取时间加上下一次存取开始之前所需要的附加时间。这里附加时间用于瞬变的信号消失或数据破坏性读后的再生。

需要注意，存储周期时间是与系统总线有关，而不是与处理器相关。

- **传输率**：这是数据传入或传出存储单元的速率。对于随机存取存储器，它等于“1/周期时间”。而对于非随机存取存储器，有下列关系：

$$T_n = T_A + \frac{n}{R} \tag{4.1}$$

其中：T_n = 读或写 N 位的平均时间

T_A = 平均存取时间

n = 位数

R = 传输率，单位为 b/s（位/秒）

存储器有许多种物理类型，目前最常用的有半导体存储器，用于磁盘和磁带的磁表面存储器以及光学和磁-光存储器。

数据存储的几个物理特性很重要。在易失性存储器中，当电源开关断开时，信息自动衰减或丢失。而在非易失性存储器中，信息一旦记录，就会保留到下一次有意改变它时为止，不需要电源来维持信息。磁表面存储器是非易失性的。半导体存储器可以是易失性的，也可以是非易失性的。不可擦除存储器不能修改，除非破坏存储单元。这种类型的半导体存储器被称为只读存储器（ROM）。当然，不可擦除存储器也必定是非易失性的。

对于随机存取存储器，存储单元的组织是一个关键的设计问题。组织的意思指通过物理排列位来形成字。如第 5 章介绍的那样，并不总是使用简单的排列。

4.1.2　存储器层次结构

计算机中存储器的设计限制可以归纳为 3 个问题：容量有多大？速度有多快？价格有多贵？

容量大小似乎并没有限制，不管容量多大，总要开发应用程序去使用它。速度多快的问题从某种意义上来说更容易回答。为了获得最佳性能，存储器的速度必须能够跟上处理器的速度。也就是说，当处理器在执行指令时，我们并不期望它因为等待指令或操作数而暂停执行。最后一个问题也必须考虑。对于实用系统，存储器的价格相对于其他组件来说必须是合理的。

正如我们所预料的，在存储器的 3 个关键特性即容量、存取时间和价格之间需要进行权衡。用来实现存储系统的技术有多种，在这一系列的技术中都存在如下关系：

- 存取时间越短，平均每位的花费就越大。
- 存储容量越大，平均每位的花费就越小。
- 存储容量越大，存取时间就越长。

设计者面临进退两难的局面是明显的。设计者想要使用存储器技术提供大容量的存储器，因为这既满足容量的要求，也使每位的价格低。然而，为了满足性能的要求，设计者又不得不使用昂贵的、相对来说容量较小而存取速度较快的存储器。

解决这个难题的办法不是只依赖单一的存储部件或技术，而是采用**存储器层次结构**（memory hierarchy）。图 4-1 给出了一种典型的层次结构，随着层次的下降，我们会发现：

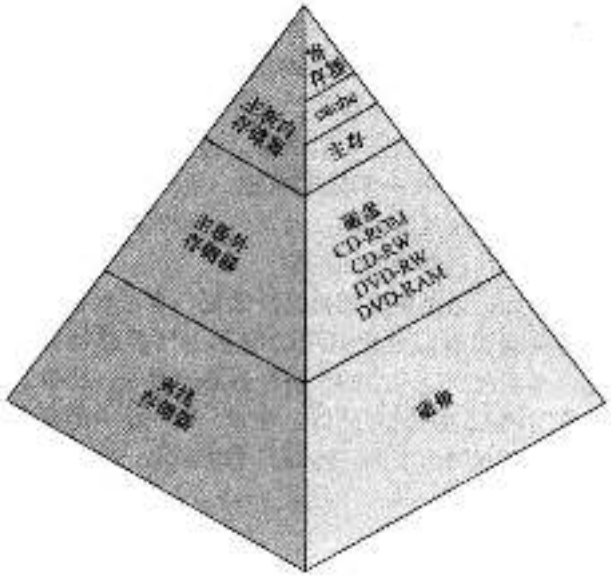


图 4-1　存储器层次结构

(a) 每位价格下降；
(b) 容量增大；
(c) 存取时间变长；
(d) 处理器访问存储器的频率降低。

因此，容量较小、价格较贵、速度较快的存储器可作为容量较大、价格较便宜、速度较慢的存储器的补充。这种组织方法要取得成功的关键是最后一项（d），即降低访问频率。我们将在本章后面部分讨论 cache 和第 8 章讨论虚拟存储器时详细介绍这个概念，这里只做简单解释。

例 4.1 假设处理器支持二级存储器结构。第一级存储器包含 1000 字，存取时间为 0.01μs；第二级存储器包含 100 000 字，存取时间为 0.1μs。假定要访问的字在第一级存储器中，则处理器能直接对它进行存取。如果字在第二级存储器中，那么该字将首先被传送到第一级，然后处理器再对它进行存取。为了使问题简化，我们忽略处理器用来判断要访问的字在哪一级所需要的时间。图 4-2 给出包含此问题的曲线的基本形状，该图显示了二级存储器结构下平均存取时间和命中率 H 之间的函数关系，其中 H 被定义为在较快存储器（如 cache）中完成的存取占所有存储器存取的百分比，T_1 是访问第一级存储器所需要的时间，而 T_2 为访问第二级存储器所需要的时间[⊖]。可以看出，在第一级存储器中的访问百分比越高，总的平均访问时间就越接近访问第一级存储器所需要的时间，而不是第二级的时间。

在此例中，假设 95% 的存储器访问都可以在 cache 中找到，那么平均访问一个字的时间可以表示为：

$$(0.95)(0.01\mu s) + (0.05)(0.01\mu s + 0.1\mu s) = 0.0095 + 0.0055 = 0.015\mu s$$

正如我们所期望的，平均访问时间更接近于 0.01μs，而不是 0.1μs。

从原理上讲，使用二级存储器可以减少平均存取时间，但此时要求条件（a）到条件（d）都满足。通过采用各种技术，出现了一系列满足条件（a）到条件（c）的存储系统，幸运的是，条件（d）通常也能满足。

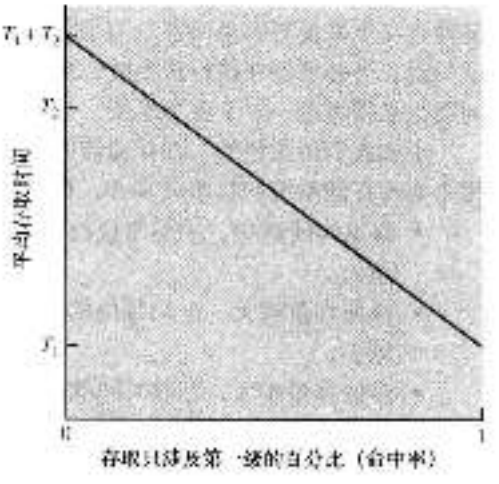


图 4-2 只包含第一级的存取性能（命中率）

条件（d）有效的基础是访问的局部性原理[DENN68]。在程序执行的过程中，处理器倾向于成簇（块）地访问存储器中的指令和数据。程序通常包含许多迭代循环和子程序。一旦进入一个循环或子程序，则会重复访问一小组指令。同样，对于表和数组的操作包含存取一簇簇的数据。在一段较长的时间中，使用的簇是变动的，但在一小段时间内，处理器主要访问存储器中的固定簇。

因此，通过层次结构组织数据，有可能使访问较低层存储器的百分比低于访问其上层存储器的百分比。考虑前面已给出的二级存储器的例子，让第二级存储器包含所有程序指令和数据。当前簇可以临时调入第一级存储器。有时，第一级中的某个簇将不得不被交换回第二级存储器，以便为所需要的新簇腾出空间。然而，平均来说，处理器大部分是对第一级中的指令和数据进行访问。

⊖ 如果被访问的字在较快速的存储器中找到，则被定义为命中（hit）；如果被访问的字在较快速的存储器中找不到，则发生缺失（miss）。

这一原理可以应用于多级存储系统。如图4-1所示的层次结构所建议的一样，速度最快、容量最小而价格最高的存储器是处理器内的寄存器。通常，一个处理器包含几十个这样的寄存器，但也有一些机器包含数百个寄存器。下面两层是主存储器，它是计算机中主要的内存系统。主存中的每个存储位置都有唯一的地址。主存通常用更快速度、更小容量的cache对其进行扩充。cache对于程序员乃至处理器来说是透明的，它在主存和寄存器之间分段传送数据以提高性能。

上面所介绍的三种形式的存储器通常都是易失性存储器，普遍采用半导体技术。这3层存储器的使用导致了各种类型的半导体存储器的开发，它们的速度和价格各不相同。数据在各种外部的、大容量的存储设备上能存储得更持久，最常用的是硬盘和可移动媒体，如可移动磁盘、磁带和光存储器。外部非易失性存储器也称为**辅助存储器**或**辅存**（secondary memory或auxiliary memory），它们常用于存储程序和数据文件，以文件或记录的形式而不是以一个一个的字节或字的形式为程序员所使用。磁盘也可以用于主存储器的扩展，称为虚拟存储器，这将在第8章中讨论。

层次结构中还包含其他形式的存储器。例如，IBM大型机通常包含了一种称作为扩展存储器的内存储器，它采用一种比主存储器要慢且更便宜的半导体技术。严格来讲，这种存储器不列入层次结构中，而只是一个分支：数据可以在主存和扩展存储器之间进行传送，但不能在扩展存储器和外存储器之间进行传送。其他形式的辅存包括光盘和“磁-光盘”设备。最后，可以在层次结构中以软件形式有效地加入附加层。主存储器中的一部分可用做缓冲区，暂时保存写入磁盘中的数据，这种技术有时称为磁盘高速缓存[⊖]，它用两种方法改进性能：

- 磁盘写入是以簇的形式进行的。一次传输的数据量较大，而不是很多次小的数据传送。这改善了磁盘的性能，减少了对处理器的占用。
- 某些指定输出的数据在转存到磁盘之前可被程序访问。这样，可以快速地从软件高速缓存中检索，而不是从相对较慢的磁盘中检索。

附录4A论述了多级存储器结构对性能的影响。

4.2 cache存储器原理

cache（高速缓存）存储器的目的是使存储器的速度逼近可用的最快存储器的速度，同时以较便宜的半导体存储器的价格提供一个大的存储器容量。图4-3a说明了这一概念，图中有一个相对大而慢的主存，加之一个小而快的cache。cache中存放了主存储器的部分副本。当CPU试图访问主存中的某个字时，首先检查这个字是否在cache中，如果是，则把这个字传送给CPU；如果不是，则将主存中包含这个字固定大小的块读入cache中，然后再传送该字给CPU。因为访问的局部性，当把某一块数据存入cache，以满足某次存储器的访问时，CPU将来还很有可能访问同一存储位置或该数据块中的其他字。

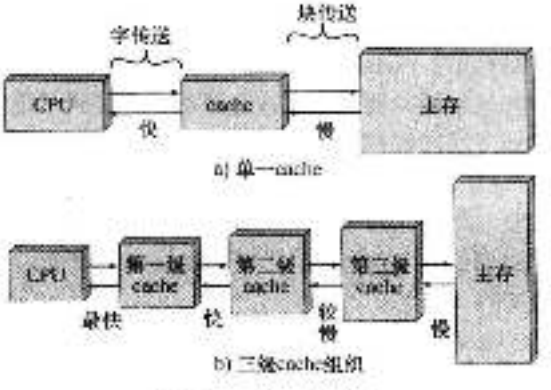


图4-3 cache和主存

图4-3b描述了多级cache的使用，其中，第二级cache比第一级cache要慢，但通常存储容量较大；而第三级cache比第二级cache慢，但通常存储容量要大。

⊖ 磁盘高速缓存并不是一种硬件，在本书中不作详细说明。详细解释可参见[STAL09]。

图4-4描述了cache/主存系统的结构。主存储器由多达2^n个可寻址的字组成，每一个字有唯一的n位地址。为了实现映射，我们将主存储器看成是由许多定长的块组成，每块有K个字。即有$M=2^n/K$个块。而cache包含m个块，称作行[⊖]，每行包括K个字和几位标记。每行还包括控制位（图中没有给出），如用作判断装入cache中的行是否被修改的控制位。行的长度，不包含标记和控制位，称为行大小（line size）。行大小可以小到32位，其每个“字”就是单个字节，此时，行大小是4个字节。行的数量远远小于主存储器块的数目（$m \ll M$）。任何时候，只有主存储器块的子集驻留在cache行中。如果要读取主存储器块中的某个字，则包含该字的块将被传送到cache的一个行中。由于块数多于行数，所以单个行不可能永久地被某块专用。因此，每行都有一个标记（tag），用来识别当前存储的是哪一块。这个标记通常是主存储器地址的一部分，这将在本节后面加以讨论。

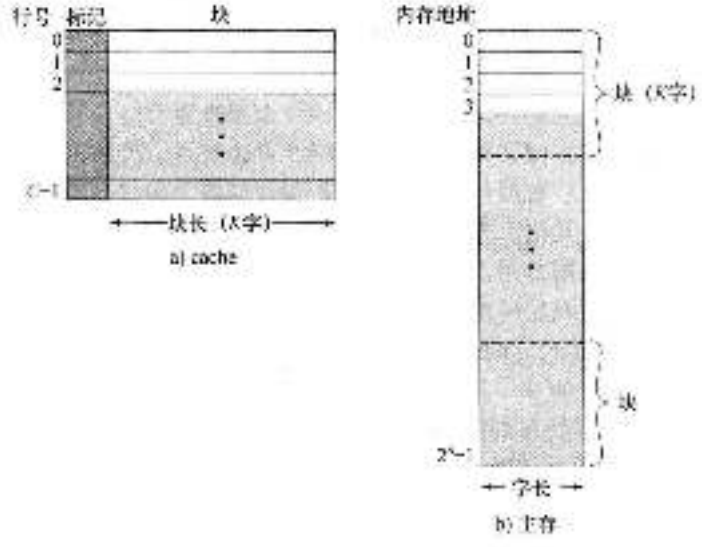


图4-4 cache/主存结构

图4-5说明了一个读操作。处理器产生一个要读取字的地址RA，如果这个字在cache中，则把它直接传送给处理器。否则，将包含这个字的块装入cache中，然后再传送给CPU。图4-5表示，最后两步操作是并行进行的，这在图4-6所示的当代cache经典组织中有所反映。在这种组织结构中，cache经数据线、控制线和地址线差接到处理器，数据线和地址线也分别与数据缓冲器和地址缓冲器相连，这些缓冲器都接到系统总线上，从而与主存连接。当cache命中时，数据和地址缓冲器都不启用，通信只在处理器和cache之间进行，此时系统总线上没有信号传输。当cache未命中时，所需求的地址被加载到系统总线上，数据通过数据缓冲器提交给cache和CPU。在其他的组织结构中，通过所有数据线、地址线和控制线，cache直接介于处理器和主存之间，在这种情况下，当cache未命中时，要读取的字先被装入cache，然后再由cache传送给CPU。

有关使用cache的性能参数讨论可参见附录4A。

⊖ 为了定义cache的基本单位，使用了术语“行”（line），而不是术语“块”（block），出于两个原因：(1) 为了避免与主存储器的块混淆，一个主存块包含的数据字数与一个cache行的相同；(2) 因为一个cache行不仅包含了K个数据字（正如主存块一样），而且它还包含标记和控制位。

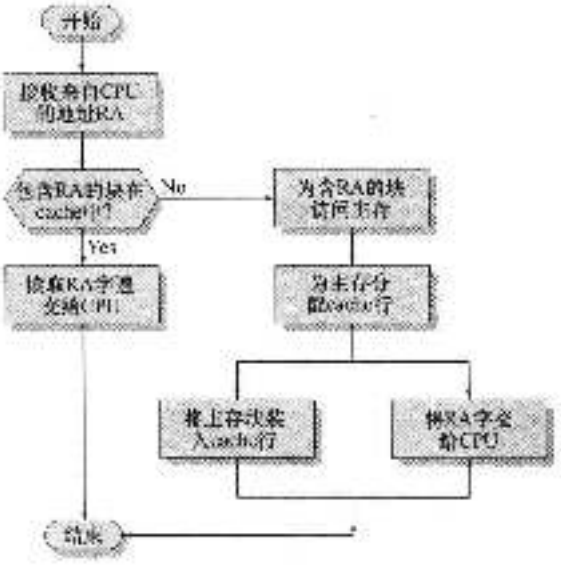


图 4-5　cache 读操作

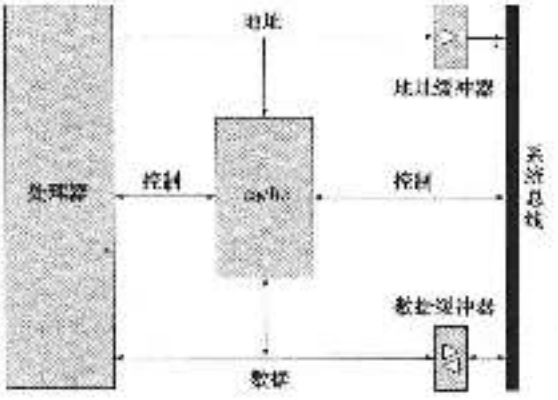


图 4-6　典型的 cache 组织

4.3　cache 的设计要素

本节给出了 cache 设计参数的概述并报告了某些典型结论。我们偶尔会谈及 cache 在高性能计算（high-performance computing，HPC）中的使用。HPC 涉及超级计算机（supercomputer）和超级计算机软件，主要应用于科学计算，因为这类应用中包含了大量的数据、向量和矩阵的计算，以及并行算法的使用。为 HPC 设计的 cache 与为其他硬件平台和应用所设计的 cache 是非常不同的。的确，许多研究人员已经发现 HPC 应用在使用了 cache 的计算机体系结构上性能会变差［BAIL93］。而另一些研究人员也已指出，如果应用软件适合使用 cache，那么 cache 层次将对改善性能很有帮助［WANG99，PRES01］[⊖]。

表 4-2　cache 的设计要素

cache 地址	写策略
逻辑地址	写直达法
物理地址	写回法
cache 容量	写一次
映射功能	行大小
直接	cache 数目
全相联	一级或两级
组相联	统一或分离
替换算法	
最近最少使用（LRU）	
先进先出（FIFO）	
最不经常使用（LFU）	
随机	

尽管已有大量的 cache 实现方案，但只有几个基本要素用于区别和分类 cache 的体系结构，表 4-2 列出了关键的要素。

4.3.1　cache 地址

几乎所有的非嵌入式处理器以及很多嵌入式处理器都支持虚拟内存。虚拟内存的概念将在第 8 章中讨论。本质上讲，虚拟内存是一种内存扩充技术，这种技术不会使主存物理地址空间大小发生改变，但允许程序在逻辑上访问更多的地址。当使用虚拟内存时，机器指令的地址域包含虚拟地址。为了从主存中进行读写操作，硬件存储器管理单元（MMU）将每个虚拟地址翻译成主存中的物理地址。

⊖ 有关 HPC 与一般讨论，请参见［DOWD98］。

当使用虚拟地址时，系统设计人员可能选择将 cache 置于处理器和 MMU 之间，或者置于 MMU 和主存之间，如图 4-7 所示。逻辑 cache，也称为虚拟 cache，使用虚拟地址存储数据。处理器可以直接访问逻辑 cache，而不需要通过 MMU。而物理 cache 使用主存的物理地址来存储数据。

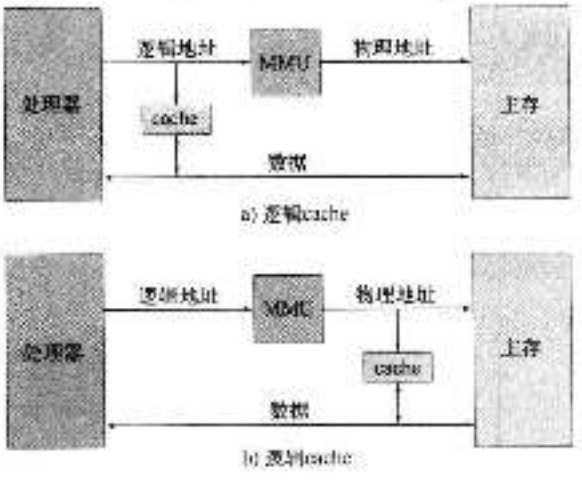


图 4-7 物理和逻辑 cache

逻辑 cache 的一个明显优势是其访问速度比物理 cache 快，因为该 cache 能够在 MMU 执行地址翻译之前作出反应。而其不足之处在于大多数虚拟存储系统为每一个应用程序提供相同的虚拟内存空间，也就是说，每个应用程序都可以从地址为 0 的虚拟内存开始。因此，相同的虚拟地址在两个不同的应用程序中涉及不同的物理地址。所以，cache 存储器必须用每一个应用程序的上下文开关对其进行完全刷新，或者为 cache 的每一行增加额外的几位来标记与该地址相关的虚拟地址。

逻辑 cache 和物理 cache 的比较是一个很复杂的问题，在本书中不进行过多的讨论。想了解更多，可参考［CEKL97］和［JACO08］。

4.3.2 cache 容量

我们前面已经对 cache 容量进行过讨论，表 4-2 中也有提到。我们希望 cache 的容量足够小，以至于整个存储系统的平均每位的价格接近于单个主存储器的价格，同时我们也希望 cache 足够大，从而使得整个存储系统的平均存取时间接近于单个 cache 的存取时间。还有几个减小 cache 容量的动机。cache 越大，寻址所需要的电路门就会越多，结果是大的 cache 比小的稍慢，即使是采用相同的集成电路技术制造并放在芯片和电路板的同一位置。cache 容量也受芯片和电路板面积的限制，因为 cache 的性能对工作负载的性能十分敏感，所以不可能有“最优”的 cache 容量。表 4-3 列出了过去和当前的某些处理器的 cache 容量。

表 4-3 一些处理器的 cache 容量

处理器	类型	推出年份	L1 cache[①]	L2 cache	L3 cache
IBM 360/85	大型机	1968	16～32kB	—	—
PDP-11/70	小型机	1975	1kB	—	—
VAX 11/780	小型机	1978	16kB	—	—
IBM 3033	大型机	1978	64kB	—	—
IBM 3090	大型机	1985	128～256kB	—	—
Intel 80486	PC	1989	8kB	—	—
Pentium	PC	1993	8kB/8kB	256～512KB	—
PowerPC 601	PC	1993	32kB	—	—
PowerPC 620	PC	1996	32kB/32kB	—	—
PowerPC G4	PC/服务器	1999	32kB/32kB	256KB～1MB	2MB
IBM S/390 G4	大型机	1997	32kB	256KB	2MB
IBM S/390 G6	大型机	1999	256kB	8MB	—
Pentium 4	PC/服务器	2000	8kB/8kB	256KB	—
IBM SP	高端服务器/超级计算机	2000	64kB/32kB	8MB	—
CRAY MTA[②]	超级计算机	2000	8kB	2MB	—
Itanium	PC/服务器	2001	16kB/16kB	96KB	4MB

（续）

处理器	类型	推出年份	L1 cache[1]	L2 cache	L3 cache
SGI Origin 2001	高端服务器	2001	32kB/32kB	4MB	
Itanium 2	PC/服务器	2002	32kB	256KB	6MB
IBM POWER5	高端服务器	2003	64kB	1.9MB	36MB
CRAY XD-1	超级计算机	2004	64kB/64kB	1MB	
IBM POWER6	PC/服务器	2007	64kB/64kB	4MB	32MB
IBM z10	大型机	2008	64kB/128kB	3MB	24～48MB

① 表示由符号分开的两个值分别表示指令 cache 和数据 cache 的容量。
② 两个 cache 都是指令 cache，无数据 cache。

4.3.3 映射功能

由于 cache 的行比主存储器的块要少，因此需要一种算法来实现主存块到 cache 行的映射。而且，还需要一种方法来确定当前哪一块占用了 cache 行。映射方法的选择决定了 cache 的组织结构，主要采用三种映射方法：直接映射、全相联映射和组相联映射。下面我们依次讨论这三种方法，分析每种方法的通用结构及其后面的具体例子。

例 4.2 对于所有这三种映射方法，该例子中都包含下列条件：

- cache 能存储 64KB。
- 数据在主存和 cache 之间以每块 4 字节大小传输。这意味着 cache 被组织成 $16K = 2^{14}$ 行，每行 4 字节。
- 主存容量为 16MB，每个字节直接由 24 位的地址（$2^{24} = 16M$）寻址。因此，为了实现映射，我们把主存看成是由 4M 个块组成，每块 4 字节。

1. 直接映射

直接映射是最简单的映射技术，将主存中的每个块映射到一个固定可用的 cache 行中。直接映射可表示为：

$$i = j \bmod m$$

其中：i = cache 行号

j = 主存储器的块号

m = cache 的行数

图 4.8a 给出了主存中前 m 块的映射情况。如图所示，主存中的每一块映射到 cache 中的唯一行，然后接下来的 m 块依次映射到 cache 中相应位置。也就是说，主存中的 B_m 块映射到 cache 中对应的 L_0 行，B_{m+1} 块映射到 L_1 行，等等。

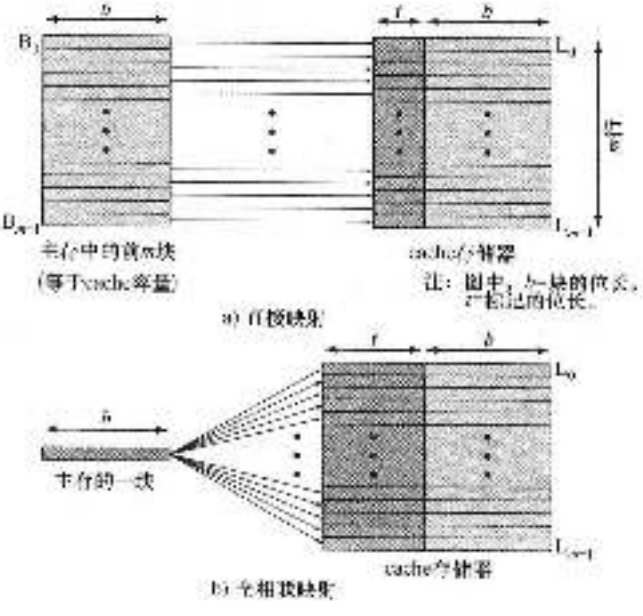


图 4.8 主存到 cache 的映射：直接映射和全相联映射

映射功能通过主存地址很容易实现。图 4.9 描述了基本的映射机制。为了访问 cache，每一个主存地址可以看成是由三个域组成。最低的 w 位标识某个块中唯一的一个字或字节；在当代大多数机器中，地址是字节级的。剩余的 s 位指定了主存 2^s 个块中的一个。cache 逻辑将这 s 位转换为 $s-r$ 位（最高位部分）的标记域和一个 r 位的行域。后者

标识了$m=2^r$个cache行中的一个。小结如下：

- 地址长度 = $(s+w)$位
- 可寻址的单元数 = 2^{s+w}个字或字节
- 块大小 = 行大小 = 2^w个字或字节
- 主存的块数 = $2^{s+w}/2^w = 2^s$
- cache的行数 = $m=2^r$
- cache的容量 = 2^{r+w}个字或字节
- 标记长度 = $(s-r)$位

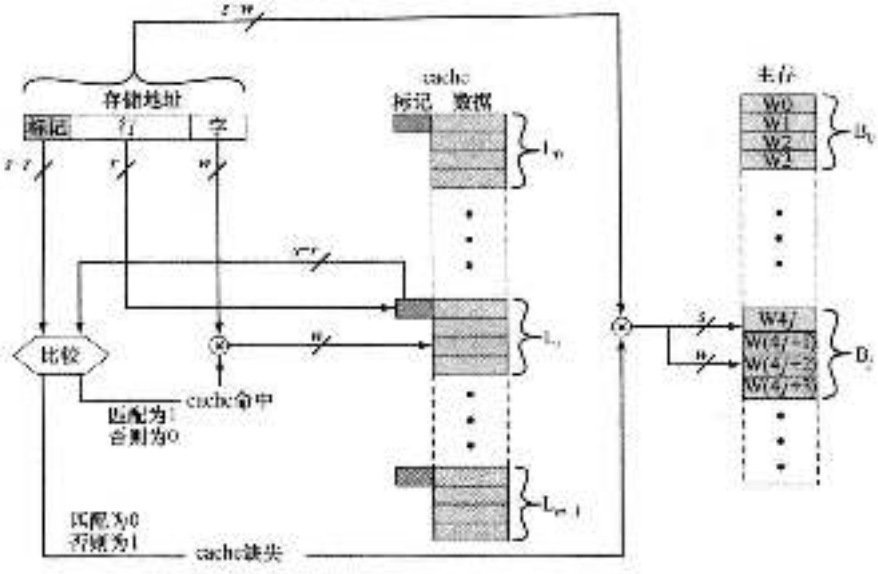


图4-9 直接映射cache的组织结构

这种映射的结构是把主存中的块分配给如下所示的cache行中。

cache行	被分配给主存块
0	$0, m, 2m, \cdots, 2^s-m$
1	$1, m+1, 2m+1, \cdots, 2^s-m+1$
⋮	⋮
$m-1$	$m-1, 2m-1, 3m-1, \cdots, 2^s-1$

因此，采用部分地址作为行号提供了主存中的每一块到cache的唯一映射。当一块读入到分配给它的行时，必须给数据做标记，从而将它与其他能装入这一行的块区别开来。最高的$s-r$位用来做标记。

例4.2a

图4-10表示了一个使用直接映射的实例系统[⊖]。在这个例子中，$m=16\text{K}=2^{14}$，而且$i=j \bmod 2^{14}$，映射变为：

⊖ 在本图和后续图中，存储器的值使用十六进制表示，参见第19章中关于数字系统的复习（十进制、二进制和十六进制）。

cache行	主存块的起始地址
0	000000，010000，…，FF0000
1	000004，010004，…，FF0004
⋮	⋮
$2^{14}-1$	00FFFC，01FFFC，…，FFFFFC

注意，映射到相同行号的两块不会有相同的标记数。因此，开始地址为000000，010000，…，FF0000的块对应的标记数分别为00，01，…，FF。

回头来再看图4-5，读操作的流程是这样的：cache系统用24位地址表示，14位的行号用来做特定行的索引。如果8位标记数与当前存储在该行的标记数相匹配，则用2位字号来选取行中的4个字节。否则，22位的标记加行号域被用来从主存中取出一块。取主存块所用的实际地址是22位的标记加行号再接两位0，因此，在块的边界起始处读取4个字节。

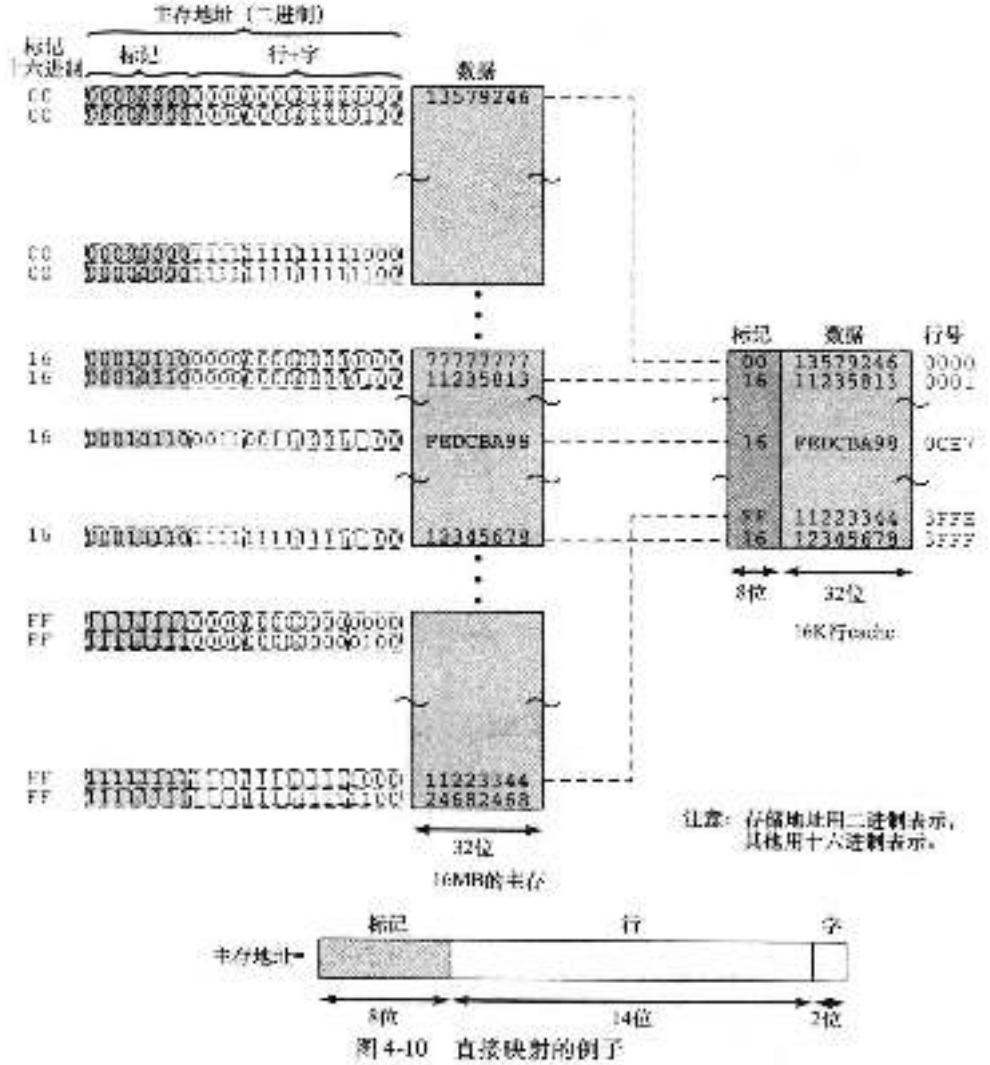


图4-10 直接映射的例子

直接映射技术简单，实现起来花费也少。其主要缺点是：对于任意给定的块，它所对应的cache位置是固定的。因此，如果一个程序恰巧重复访问两个需要映射到同一行中且来自不同块的字，则这两个块将不断地被交换到cache中，cache的命中率将会降低（一种所谓的抖动现象）。

一种降低缺失开销的办法是保存被丢弃的数据以备再次需要用到它。因为被丢弃的数据已经被取进 cache 中过，因此再次使用的开销比较小。使用 Victim cache 可以实现这种资源重复利用机制。最初提出 Victim cache 这一概念是为了减少直接映射 cache 中冲突缺失的次数，并且不影响其快速存取的时间。Victim cache 是一种全相联 cache，其存储容量一般为 4～16 个 cache 行，置于使用直接映射的第一级 cache 和下一级存储器之间。这一概念将在附录 D 中进行探究。


选择生的 Victim 高速缓存模拟器

2. 全相联映射

全相联映射克服了直接映射的缺点，它允许每一个主存块装入 cache 中的任意行，如图4-8b所示。在这种情况下，cache 控制逻辑将存储地址简单地表示为一个标记域加一个字域。标记域用来唯一标识一个主存块。为了确定某块是否在 cache 中，cache 控制逻辑必须同时对每一行中的标记进行检查，看其是否匹配。图 4-11 说明了这一逻辑。注意，地址中无对应行号的字段，所以 cache 中的行号不由地址格式决定。总结如下：

- 地址长度 = $(s+w)$ 位
- 可寻址的单元数 = 2^{s+w} 个字或字节
- 块大小 = 行大小 = 2^w 个字或字节
- 主存的块数 = $2^{s+w}/2^w = 2^s$
- cache 的行数 = 不由地址格式决定
- 标记长度 = s 位

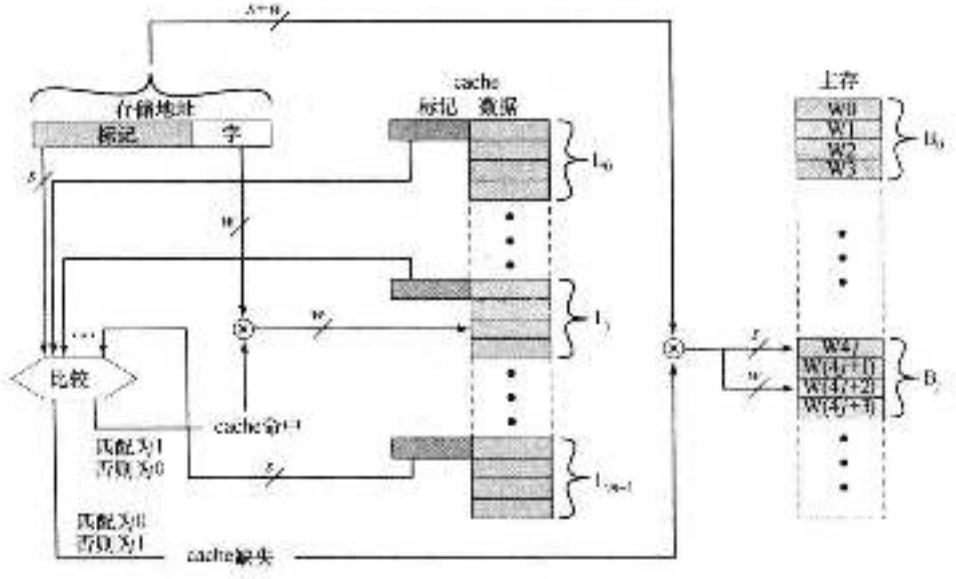


图 4-11 全相联映射的 cache 组织

例 4.2b 图 4-12 给出了使用全相联映射的例子。主存地址由 22 位标记和 2 位的字节号组成。22 位标记必须与 32 位数据块一起存储在 cache 行中。注意，地址的最左（最高）22 位形成标记。因此，24 位十六进制地址 16339C 有 22 位标记 058CE7。这由二进制表示法很容易看出：

存储地址	0001	0110	0011	0011	1001	1100	（二进制）
	1	6	3	3	9	C	（十六进制）
标记（最左 22 位）	00	0101	1000	1100	1110	0111	（二进制）
	0	5	8	C	E	7	（十六进制）

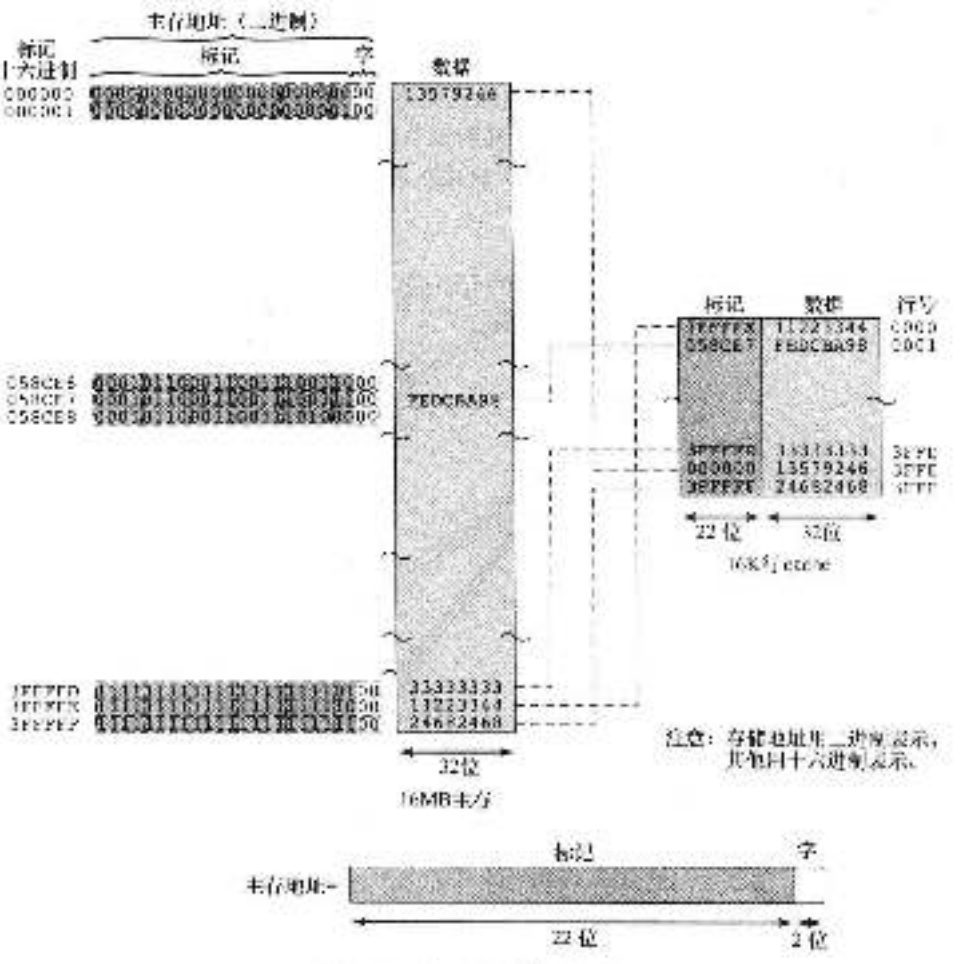


图 4-12 全相联映射的例子

对于全相联映射，当新的块读入 cache 中时，替换旧的一块很灵活。替换算法本节后面将要讨论，它用来使命中率最大。全相联映射的主要缺点是需要复杂的电路来并行检查所有的 cache 行标记。



3. 组相联映射

组相联映射是一种折中方法，它既体现了直接映射和全相联映射的优点，又避免了两者的缺点。

在组相联映射中，cache 分为 v 个组，每组包含 k 个行，它们的关系为：

$$m = v \times k$$

$$i = j \bmod v$$

其中：i = cache 组号

j = 主存块号

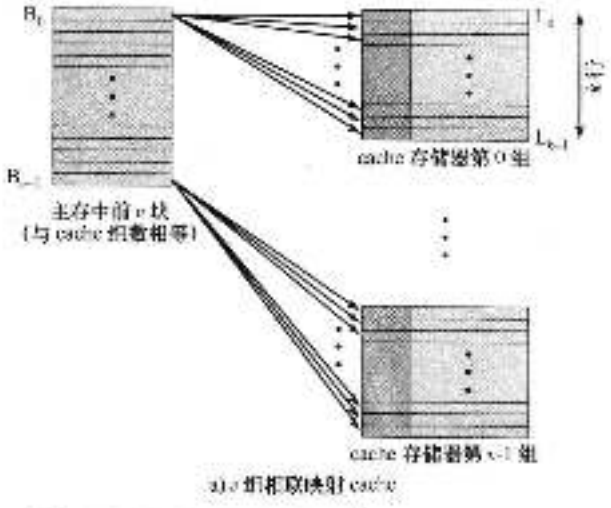
m = cache 的行数

v = 组数

k = 每组中的行数

这被称为 k 路组相联映射。采用组相联映射，块 B_j 能够映射到组 j 的任意行中。图 4-13a 给出了主存中前 v 块与 cache 行的映射关系。在全相联映射中，每一个字映射到多个 cache 行中。

而对于组相联映射，每一个字映射到特定一组的所有 cache 行中，于是，主存中的 B_0 块映射到第 0 组，如此等等。因此，组相联映射 cache 在物理上是使用了 v 个全相联映射的 cache。同时，它也可看作为 k 个直接映射的 cache 的同时使用，如图 4-13b 所示。每一个直接映射的 cache 称为路，包括 v 个 cache 行。主存中首 v 个块分别映射到每路的 v 行中，接下来的 v 个块也是以同样的方式映射，后面也如此。直接映射一般应用于轻度关联（k 值较小）的情况，而全相联映射应用于高度关联的情况 [JACO08]。



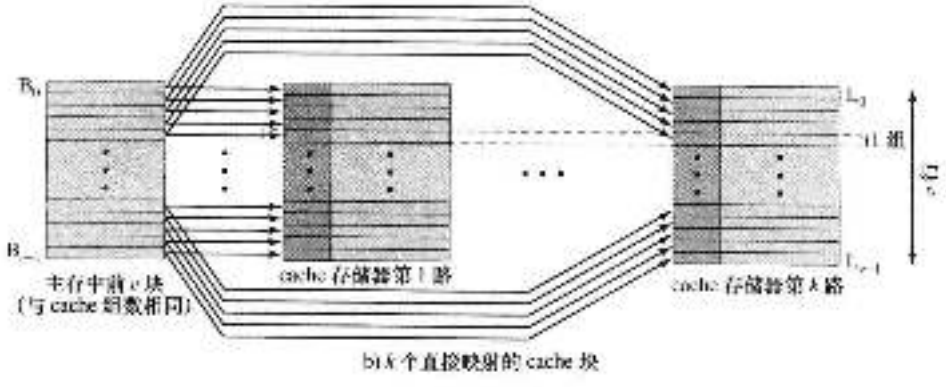


图 4-13 从主存到 cache 的映射：k 路组相联

在组相联映射中，cache 控制逻辑将存储地址表示为三个字段：标记、组和字。长度为 d 位的组字段指定了 $v = 2^d$ 个组中的唯一一个组。标记字段和组字段共长 s 位，用以标明主存中 2^s 个块中具体某一块。图 4-14 描述了 cache 控制逻辑。在全相联映射中，主存地址中的标记字段很长，而且还必须与 cache 中每一行匹配。而在 k 路组相联映射中，主存地址中的标记字段要短很多，而且只需与某一组中的 k 行匹配。总结如下：

- 地址长度 = $(s+w)$ 位
- 可寻址的单元数 = 2^{s+w} 个字或字节
- 块大小 = 行大小 = 2^w 个字或字节
- 主存的块数 = $2^{s+w}/2^w = 2^s$
- cache 中每组的行数 = k
- 组数 = $v = 2^d$

- cache中的行数 $= m = kv = k \times 2^d$
- cache存储容量 $= k \times 2^s$ 字或字节
- 标记长度 $= (s-d)$ 位

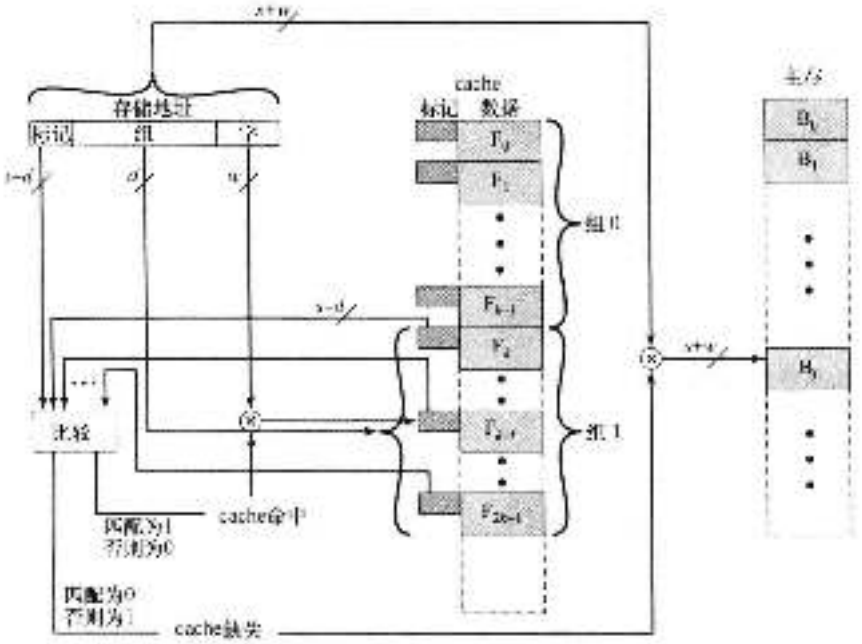


图4-14 k路组相联映射的cache组织

例4.2c 图4-15给出了一个用组相联映射的例子，其中每一组有两行，也就是二路组相联。13位长的组号标识了cache中唯一的两行组，也给出了用2^{13}取模后的主存块号。这确定了块到行的映射。因此，主存中的块000000，008000，…，FF8000映射到cache中的第0组，其中每一块都能装入该组两行中的任意一行。注意，两个映射到同一cache组的块不可能具有相同的标记数。对于读操作，用13位组号检查确定组地址，组中的两行与被存取地址的标记数进行匹配检查。

在$v=m$、$k=1$的极端情况下，组相联技术简化为直接映射。而对于$v=1$、$k=m$的情况，它又会等同于全相联映射。每组两行（$v=m/2$，$k=2$）是最常用的组相联结构。与直接映射相比，它明显地提高了命中率。四路组相联（$v=m/4$，$k=4$）用相对较少的附加成本使命中率有一些提高［MAYB84，HILL89］。继续增强每组的行数对cache命中率的提高几乎没什么效果。

表4-16给出了组相联cache性能的一个模拟研究结果，图中显示了不同cache容量对cache性能的影响［GENU04］。我们注意到，当cache的容量达到64kB之前，直接映射和二路组相联映射的性能区别是非常显著的。同时也注意到，二路组相联和四路组相联在cache容量均为4kB时的性能差别要远小于cache容量从4kB变到8kB时的差别。由于cache的复杂性与相联性成正比例，因此，在这种情况下，将cache容量增大到8kB甚至16kB都是没有道理的。最后需要注意的是，当cache容量超过32kB时，cache容量的增加对提高性能的作用并不明显。



图4-16是基于对GCC编译器执行的模拟。不同的应用可能会产生不同结果。例如，［CANT01］报告了使用很多CPU2000 SPEC基准程序时的cache性能结果。［CANT01］中比较命中率与cache容量的结果与图4-16中的基本相同，但是指定的值有些不同。

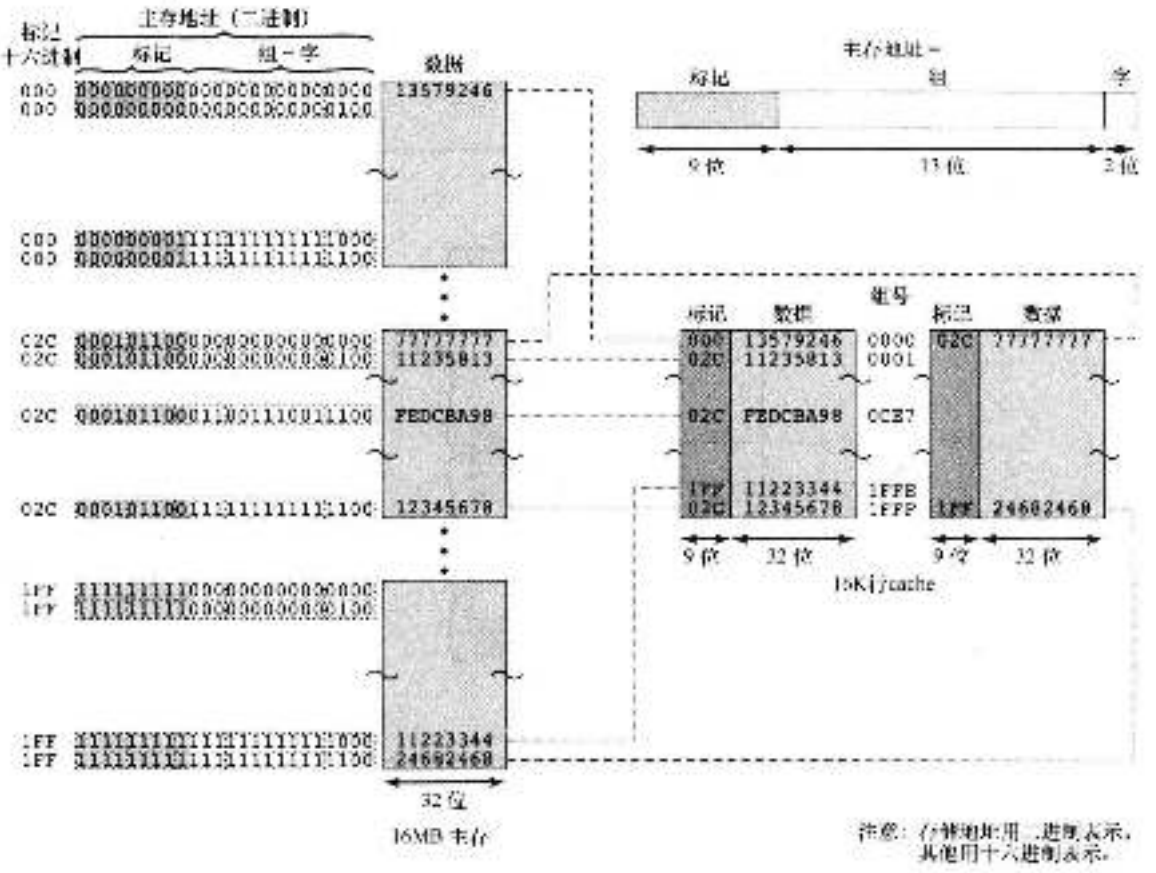


图 4-15 二路组相联映射实例

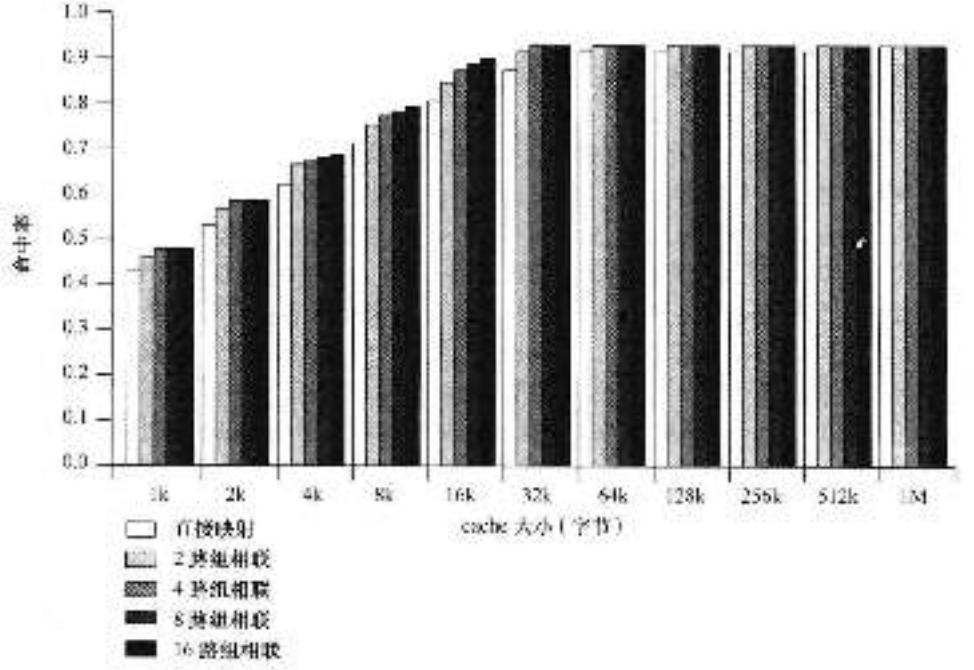


图 4-16 不同相联度在不同 cache 大小下的命中率

4.3.4 替换算法

一旦 cache 行被占用，当新的数据块装入 cache 中时，原存在的块必须被替换掉。对于直接映射，任意特殊块都只有唯一的一行可以使用，没有选择的可能。对于全相联映射技术和组相联映射技术，则需要一种替换算法。为了获得高速度，这种算法必须由硬件来实现。人们尝试过许多算法，下面介绍最常用的 4 种算法。可能最有效的算法是最近最少使用的算法（LRU）：替换掉那些在 cache 中最长时间未被访问过的块。对于两路组相联，这种方法很容易实现，每行包含一个 USE 位。当某行被引用时，其 USE 位被置为 1，而这一组中另一行的 USE 位被置为 0。当把一块读入到这一组中时，就会替换掉 USE 位为 0 的行。由于我们假定越是最近使用的存储单元越有可能将被访问，因此，LRU 会给出最佳的命中率。对于全相联 cache，LRU 也相对容易实现。高速缓存机制会为 cache 中的每行保留一个单独的索引表。当某一行被访问时，它就会移动到表头，而在表尾的行将被替换掉。因为其实现简单，LRU 是目前使用最广泛的替换算法。

另一种可能的算法是先进先出（FIFO）：替换掉那些在 cache 中停留时间最长的块。FIFO 采用时间片轮转法或环形缓冲技术很容易实现。还有另一个可能算法是最不经常使用（LFU）：替换掉 cache 中被访问次数最少的块。LFU 可以用与每行相关的计数器来实现。第四种算法是一种不基于使用情况的技术（不是 LRU，LFU，FIFO，或其他变体），它是在候选行中任意选取，然后进行替换。模拟实验结果表明，随机替换算法在性能上只稍逊于基于使用情况的算法[SMIT82]。

4.3.5 写策略

当驻留在 cache 中的某块要被替换时，必须考虑两点。如果 cache 中的原块没有被修改过，那么它可以被直接替换掉，而不需要事先写回主存。如果在 cache 某行中至少在一个字上进行过写操作，那么在替换掉该块之前必须将该行写回主存对应块，以进行主存更新。各种可行的写策略都对性能和价格进行了权衡，但还存在两个争论的问题。首先，有一个以上的设备已经访问了主存储器。例如，I/O 模块可能直接读/写存储器。如果一个字只在 cache 中修改过，那么相应的存储器字就是无效的。进一步，如果某 I/O 设备修改了主存储器，则 cache 中的字是无效的。当多个处理器连接到同一总线上，并且每个处理器都有自己局部的 cache 时，则出现了更复杂的问题。因此，如果在一个 cache 中修改了一个字，那么可以设想在其他 cache 中该字是无效的。

最简单的技术称为写直达（write through）。采用这种技术，所有写操作都同时对主存和 cache 进行，以保证主存中的数据总是有效的。任何其他处理器-高速缓存模块监视对主存的访问，都是维护它自己 cache 的一致性。这一技术的主要缺点是产生了大量的存储通信量，可能引起瓶颈问题。另一种技术称为写回法（write back），它减少了主存的写入。使用写回技术时，只更新 cache 中的数据。当更新操作发生时，需要设置与该行相关的脏位（dirty bit）或使用位（use bit）。然后，当一个块被替换掉时，当且仅当脏位被置位时才将它写回主存。写回的缺点是，部分主存数据是无效的，因此 I/O 模块的存取只允许通过 cache 进行，这就使得电路设计更加复杂而且存在潜在的瓶颈问题。经验表明，写操作占存储器操作的 15% [SMIT82]。然而，对于 HPC 应用，这个值可接近 33%（向量-向量乘法），甚至可高达 50%（矩阵转置）。

例 4.3 考虑一个行大小为 32 字节的 cache 和一个传送一个 4 字节字用时 30ns 的主存。cache 的任意行被替换之前至少已被写过一次，如果要使写回法比写直达法更高效，在被替换之前平均每行被写的次数是多少？

采用写回法时，每一个脏行只在交换时写回主存一次，需要 $8 \times 30 = 240$ns。而采用写直达法时，每一次更新 cache 中的某行都要求有一个字写到主存，耗时 30ns。因此，如果行换出之前写入平均超过 8 次的话，则写回法更有效。

在不止一个设备（通常是处理器）有 cache 且共享主存的总线结构中出现了一个新的问题。如果某个 cache 中的数据被修改，则它不但会使主存中的相应字无效，而且也会使其他 cache 中的对应字无效（如果其他 cache 中恰巧也有这个字）。即使采用“写直达”策略，其他 cache 也可能包含无效的数据。防止这个问题的系统被说成是维护 cache 的一致性。保证 cache 一致性的方法有：

- **写直达的总线监测**：每个 cache 控制器监视地址线，以检测总线的其他主控者对主存的写操作。如果有另一个总线主控者向共享存储单元写入数据，而这个单元内容同时驻留在 cache 中，则该 cache 控制器使 cache 中的这一项无效。这一策略要求所有 cache 控制器都使用写直达策略。
- **硬件透明**：使用附加的硬件来保证所有通过 cache 对主存的修改反映到所有 cache 中。因此，如果某个处理器修改了自己 cache 中的一个字，则同时会修改主存对应单元，任何其他 cache 中相同的字也同时会被修改。
- **非 cache 存储器**：只有一部分主存为多个处理器共享，这称为非 cache。在这样的系统中，所有对共享存储器的访问都导致 cache 缺失，因为共享存储器中的数据不会复制到 cache 中。非 cache 存储器能采用片选逻辑或高地址位来标识。

cache 一致性是一个活跃的研究领域，我们将在第五部分对其进行深入探讨。

4.3.6 行大小

另一个设计要素是行大小。当一个数据块被检索并放入 cache 中时，所需的字和一些相邻的字都会被取出。当数据块大很小变得较大时，命中率刚开始会因为局部性原理而增加。局部性原理是指：被访问字附近的数据很可能会在不久的将来被访问到。随着块大小的增加，更多有用数据被装入 cache。但是，当块变得相当大，并且使用新取信息的概率变得小于重用已被替换出的信息概率时，命中率开始下降。块的两个特殊作用如下：

- 较大的块减少了装入 cache 中的块数。因为每个新块都会覆盖掉原来 cache 块中的内容，少量的块导致了装入的数据很快被改写。
- 当块变大时，每个附加字就会离所需字更远，因此被使用的可能性也就更小。

块大小与命中率之间关系复杂，它取决于特定程序的局部性特征，目前还没有找到确定的最优值。块大小为 8～64B 时，比较接近最优值 [SMIT87，PRZY88，PRZY90，HAND98]。对于 HPC 系统来说，最常用的行大小是 64B 和 128B。

4.3.7 cache 数目

最初引入 cache 时，系统通常只有一个 cache。近年来，使用多个 cache 已经变得相当普遍。我们所考虑设计问题的两个方面是：cache 的级数以及采用统一或分立的 cache。

1. 多级 cache

由于集成度的提高，将 cache 与处理器置于同一芯片（片内 cache）成为可能。与通过外部总线连接的 cache 相比，片内 cache 减少了处理器在外部总线上的活动，从而减少了执行时间，全面提高了系统性能。当所需的指令或数据在片内 cache 中时，消除了对总线的访问。因为与总线长度相比，处理器内部的数据路径较短，访问片内 cache 甚至比零等待状态的总线周期还要快。而且，在这段时间内，总线是空闲的，可用于其他数据的传送。

片内 cache 导致了另一个问题：是否仍需要使用一个片外的或外部的 cache。通常，答案是肯定的，多数当代的设计既包含片内 cache，又包含外部 cache。这种组织方式中最简单的是两级 cache，其中，片内 cache 为第一级（L1），外部 cache 为第二级（L2）。包含 L2 cache 的理由如下：如果没有 L2 cache 并且处理器要求访问的地址不在 L1 cache 中时，则处理器必须通过总线访

问DRAM或ROM存储器。因为通常总线速度较慢且存储器存取时间较长，这就导致了较低的性能。另一方面，如果使用了L2 SRAM（静态RAM）cache，则经常缺失的信息能够很快被取来。如果SRAM的速度快到能与总线速度相匹配，则数据能够用零等待状态来存取，这是总线传输最快的一种类型。

当代多级cache设计的两个特点值得注意：第一，对于片外L2 cache，许多设计都不是用系统总线作为L2 cache和处理器之间的传送路径，而是使用单独的数据路径，以便减轻系统总线的负担。第二，随着处理器部件持续缩小，现在已有许多处理器将L2 cache结合到处理器芯片上，改善了性能。

若想使用L2 cache，则取决于L1和L2中的命中率。一些研究表明，使用两级cache通常确实可以提高性能（参见[AZIM92]、[NOVI93]、[HAND98]）。然而，多级cache若要用也使得关于cache设计的所有问题都变得复杂，包括cache容量、替换算法和写策略等，详见[HAND98]和[PEIR99]。

图4-17给出了在不同cache大小情况下两级cache性能的模拟研究结果[GENU04]。图中假定两级cache都有相同的cache行大小，并给出了不同情况下的总命中率。也就是说，如果所需的数据在L1 cache中或L2 cache中出现，则算一次命中。图中显示了不同L1 cache大小下L2 cache大小对总命中率的影响。直到L2 cache大小至少为L1 cache大小的两倍时，才对提高总命中率有明显的作用。注意：当L1 cache为8KB时，曲线最陡的点出现在L2 cache为16KB时。同样，当L1 cache为16KB时，曲线最陡的点出现在L2 cache为32KB时。在最陡的点出现之前，L2 cache对总cache性能几乎没多少影响。需要L2 cache比L1 cache大时，才能使提高性能成为可能。如果L2 cache与L1 cache有相同的行大小和容量，则其内容将或多或少与L1 cache中相同。

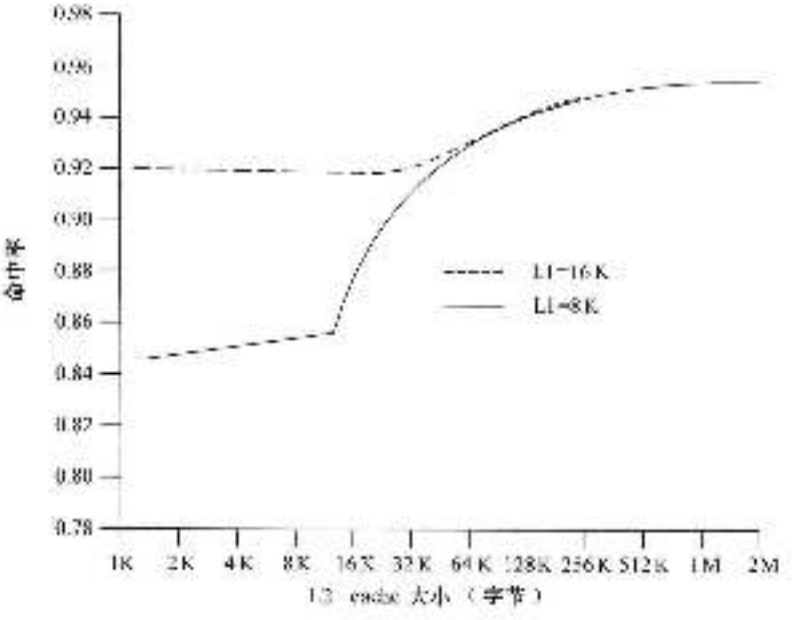


图4-17 L1为8KB和16KB时的总命中率（L1和L2）

随着适用于cache的芯片面积的可用性的提高，大多数当代处理器已将L2 cache移到处理器芯片上，并添加了一个L3 cache。最初，L3 cache是经过外部总线来存取的，而最近，大多数处理器已集成到L3 cache上。无论哪种情况，加入L3 cache使性能明显地得到提升（参见[GHAI98]）。

2. 统一与分立 cache

当片内 cache 首次出现时，许多设计都采用单个 cache，既存放数据，又存放指令。近年来，通常把 cache 分为两部分：一个专门用于指令，另一个专门用于数据。这两种 cache 被置于同一级，通常作为两个 L1 cache。当处理器试图从主存中取指令时，它首先查阅指令 L1 cache；而当处理器试图从主存中取数据时，它会先查阅数据 L1 cache。

统一 cache 有两个潜在的优点：

- 对于给定的 cache 容量，统一 cache 较独立 cache 有更高的命中率，因为它在获取指令和数据的负载之间自动进行平衡。也就是说，如果执行方式中取指令比取数据要多得多，则 cache 倾向于被指令填满。如果执行方式中要读取的数据相对较多，则会出现相反的情况。
- 只需设计和实现一个 cache。

尽管统一 cache 有这些优点，但分立 cache 是一种发展趋势，特别是对于超标量机器，例如 Pentium 和 PowerPC，它们强调并行指令执行和带预测的指令预取。分立 cache 设计的主要优点是消除了 cache 在指令的取指/译码单元和执行单元之间的竞争，这在任何基于指令流水线的设计中都是重要的。通常处理器会提前获取指令，并把将要执行的指令装入缓冲区或流水线。假设，现在有统一指令/数据 cache，当执行单元执行存储器访问来存和取数据时，这一请求被提交给统一 cache。如果同时指令预取器为取指令向 cache 发出读请求，则后一请求会暂时阻塞，以便 cache 能够先为执行单元提供服务，使它能完成当前的指令执行。这种对 cache 的竞争会降低性能，因为它干扰了指令流水线的有效使用。分立 cache 结构解决了这一问题。

4.4 Pentium 4 的 cache 组织

可以从 Intel 微处理器的发展中清晰地看到 cache 组织的演变（如表 4-4 所示）。80386 不包含片内 cache；80486 包含一个 8KB 的片内 cache，它采用四路组相联结构，每行 16B。所有的 Pentium 处理器都包含两个片内 L1 cache，一个用于数据，另一个用于指令。对于 Pentium 4，其 L1 数据 cache 的容量是 16KB，每行 64B，采用四路组相联结构。Pentium 4 的指令 cache 将在后面进行介绍。Pentium II 还包括一个 L2 cache，它为 L1 数据 cache 和指令 cache 提供信息。该 L2 cache 采用 8 路组相联结构，容量为 512KB，每行 128B。Pentium III 添加了一个 L3 cache，而到 Pentium 4 的高端版本，L3 cache 已经移到了处理器芯片内。

表 4-4 Intel 的 cache 进展

问题	解决方案	首次采用该特征的处理器
外部存储器比系统总线慢	使用更快的存储器技术增加外部 cache	386
增加的处理器速度导致外部总线成为 cache 访问的瓶颈	将外部 cache 移到片内，以处理器相同的速度进行操作	486
由于片内空间的限制，片内 cache 太小	使用比主存更快的技术，增加外部 L2 cache	486
当指令预取器和执行单元同时需要访问 cache 时出现了竞争。在此种情况下，指令预取器在执行单元访问数据时只能暂停	形成分离的数据 cache 和指令 cache	Pentium
增加的处理器速度导致外部总线成为 L2 cache 访问的瓶颈	形成分离的后端总线，它比主（前端总线）外部总线运行速度更快。BSB 总线服务于 L2 cache	Pentium Pro
	将 L2 cache 移到处理器芯片内	Pentium II
某些应用需要处理庞大的数据库，并且必须对大量数据进行快速访问。片上 cache 太小	增加外部 L3 cache	Pentium III
	将 L3 cache 移到片内	Pentium 4

图4-18给出了一个Pentium 4组织的简化图，着重强调了3个cache的布局。处理器核心由4个主要部件组成：

- **取指/译码单元**（fetch/decode unit）：按顺序从L2 cache中读取程序的指令，将它们译成一系列的微类作，并存入L1指令cache中。
- **乱序执行逻辑**（out-of-order execution logic）：依据数据相关性和资源可用性调度微操作的执行，于是，微操作可按不同于所取指令流的顺序被调度执行。只要时间许可，此单元调度将来可能需要的微操作的推测执行。
- **执行单元**（execution unit）：这些单元用来执行微操作，它们从L1数据cache中获取所需的数据，并且将结果暂存在寄存器中。
- **存储器子系统**（memory subsystem）：这部分包含L2和L3 cache以及系统总线。当L1和L2 cache未命中时，使用系统总线访问主存。系统总线还可用于访问I/O资源。

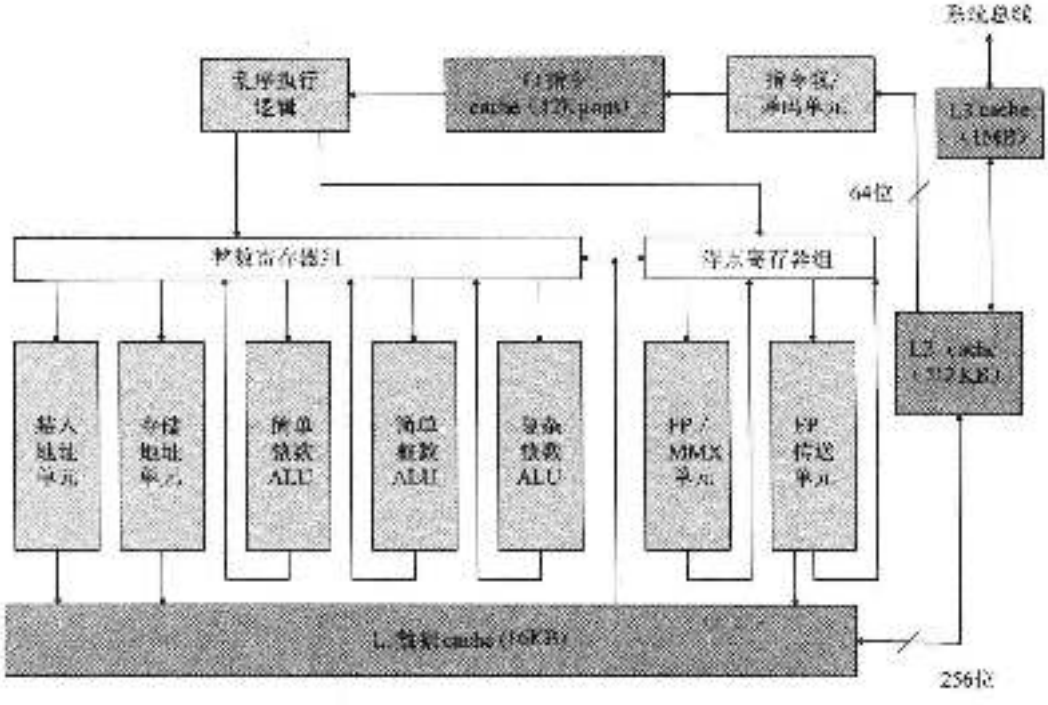


图4-18 Pentium 4框架

不同于所有先前的Pentium处理器和大多数其他处理器所采用的组织结构，Pentium 4的指令cache位于指令译码逻辑和执行核心之间。使用这种设计的理由如下：Pentium处理器将机器指令译码或转换成被称为微操作的简单RISC类指令，使用简单的定长微操作允许采用超标量流水线和调度技术，从而提高了性能。然而，Pentium机器指令译码不方便，它们长度可变并且有许多不同的选项。研究结果表明，若独立于调度和流水线逻辑来译码的话，性能会增强。在第14章我们再进行更全面的讨论。

数据cache采用写回策略：仅当修改过的数据被替换出cache时，才写回主存。Pentium 4处理器也能动态配置以支持写直达高速缓存。

L1数据cache由控制寄存器中的两位控制。它们标记为CD（cache disable，cache禁用）和NW（非写直达）位（如表4-5所示）。Pentium 4还有两条控制数据cache的指令：INVD用于清除内部cache存储器，并向外部cache（如果有）发送清除信号；WBINVD先执行写回操作并使内部cache无效，然后再执行写回操作并使外部cache无效。

L2 和 L3 cache 都是采用 8 路组相联结构，每行 128B。

表 4-5 Pentium 4 cache 操作模式

控制位		操作模式		
CD	NW	cache 写入	写命中	使无效
0	0	允许	允许	允许
1	0	禁止	允许	允许
1	1	禁止	禁止	禁止

注：CD = 0，NW = 1 是一对无效组合。

4.5 ARM 的 cache 组织

ARM cache 组织随着整个 ARM 系列体系结构的发展而发展，这也折射出所有微处理器设计者不断追求高性能的动力。

表 4-6 显示了这一发展历程。ARM7 模型使用一个统一的 L1 cache，而接下来的模型普遍使用分立的指令/数据 cache。所有的 ARM 设计都是使用组相联 cache，但 cache 的相联度和行大小有所不同。对于 ARM7 到 ARM10 系列处理器，包括 Intel StrongARM 和 Intel Xscale 处理器，带有 MMU（Memory Management Unit）单元的 ARM 缓存内核都使用一个逻辑 cache。而 ARM11 系列使用物理 cache。逻辑 cache 和物理 cache 的区别在本章前面部分已经进行了讨论（如图 4-7 所示）。

表 4-6 ARM cache 特性

核心	cache 类型	cache 容量（KB）	cache 行大小（字）	组相联度	存储单元	写缓冲大小（字）
ARM720T	统一	8	4	4 路	逻辑	8
ARM920T	分立	16/16 数据/指令	8	64 路	逻辑	16
ARM926EJ-S	分立	4-128/4-128 数据/指令	8	4 路	逻辑	16
ARM1022E	分立	16/16 数据/指令	8	64 路	逻辑	16
ARM1026EJ-S	分立	4-128/4-128 数据/指令	8	4 路	逻辑	8
Intel StrongARM	分立	16/16 数据/指令	4	32 路	逻辑	32
Intel Xscale	分立	32/32 数据/指令	8	32 路	逻辑	32
ARM1136-JF-S	分立	4-64/4-64 数据/指令	8	4 路	物理	32

ARM 体系结构的一个有趣特点是使用了一个小型先入先出（FIFO）缓冲区以提高内存写性能。该写缓冲区内置于 cache 和主存之间，它由一组地址和一组数据字组成。与 cache 相比，写缓冲区要小很多，可能仅能容纳 4 个独立的地址。通常，虽然写缓冲区在页级会选择性关闭，但它可供所有主存使用。图 4-19 显示了写缓冲区、cache 和主存之间的关系（来源：SLOS04）。

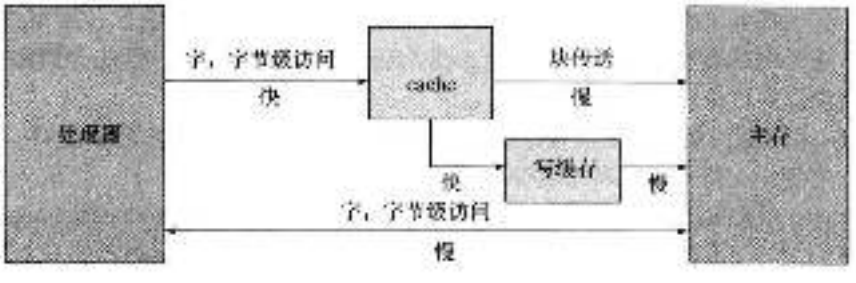


图 4-19 ARM cache 写缓冲组织结构

写缓冲区上的操作如下：当处理器向缓冲区执行一个写操作时，数据以处理器时钟速度被放入写缓冲区，而处理器继续执行其他操作。写操作在cache中的数据被写回主存时发生。因此，要写的数据是由cache传送到写缓冲区中的。然后写缓冲区异行执行外部写操作。然而，如果写缓冲区暴满了（不管是因为缓冲区中的字数已达上限，还是没有新地址槽），处理器就会停止执行，直到缓冲区中有足够的空间为止。随着非写入操作的执行，写缓冲待续地向主存写入数据直到缓冲区完全为空。

写入写缓冲区的数据在传送到主存之前不能读回cache中，主要原因是写缓冲相当小。虽然如此，除非在一个执行程序中写操作所占比例非常高，写缓冲区才能提高性能。

4.6 推荐的读物

[JACO08]关于cache设计的论述非常出色，也跟上了最新技术的发展。另一个关于cache设计的详细论述在[HAND98]中。到现在仍值得一读的经典论文是[SMIT82]，它总结了cache设计的各种要素，同时提供了非常丰富的分析结果。另一篇非常有趣的经典之作是[WILK65]，它可能是第一篇引入cache概念的论文。[GOOD83]也提供了cache行为的有用分析。另一篇有价值的分析是论文[BELL74]。[AGAR89]给出了许多cache设计问题中关于多道程序和多线程部分的详细论述。[HIGB90]提供了一组简单的公式，这些公式能用来评估在不同参数设置下的cache性能。

AGAR89 Agarwal, A. *Analysis of cache Performance for Operating Systems and Multiprogramming*. Boston: Kluwer Academic Publishers, 1989.

BELL74 Bell, J.; Casasent, D.; and Bell, C. "An Investigation into Alternative cache Organizations." *IEEE Transactions on Computers*, April 1974. http://research.microsoft.com/users/GBell/gbvita.htm.

GOOD83 Goodman, J. "Using cache Memory to Reduce Processor-Memory Band- width." *Proceedings, 10th Annual International Symposium on Computer Architecture*, 1983. Reprinted in [HILL00].

HAND98 Handy, J. *The cache Memory Book*. San Diego: Academic Press, 1993.

HIGB90 Higbie, L. "Quick and Easy cache Performance Analysis." *Computer Architecture News*, June 1990.

JACO08 Jacob, B.; Ng, S.; and Wang, D. *Memory Systems: cache, DRAM, Disk*. Boston: Morgan Kaufmann, 2008.

SMIT82 Smith, A. "cache Memories." *ACM Computing Surveys*, September 1982.

WILK65 Wilkes, M. "Slave Memories and Dynamic Storage Allocation." *IEEE Transactions on Electronic Computers*, April 1965. Reprinted in [HILL00].

4.7 关键词、思考题和习题

关键词

access time：存取时间，访问时间
associative mapping：全相联映射
cache hit：cache命中
cache memory：cache存储器，高速缓冲存储器
cache miss：cache缺失，cache未命中
data cache：数据cache
direct access：直接存取
direct mapping：直接映射
high-performance computing (HPC)：高性能计算
hit ratio：命中率
instruction cache：指令cache
L1 cache：一级cache
L2 cache：二级cache
L3 cache：三级cache
Locality：局部性
logical cache：逻辑cache
memory hierarchy：存储器层次结构
multilevel cache：多级cache
physical cache：物理cache
random access：随机存取
replacement algorithm：替换算法
sequential access：顺序存取
set associative mapping：组相联映射
spatial locality：空间局部性
split cache：分立cache
tag：标记
temporal locality：时间局部性
unified cache：统一cache

virtual cache：虚拟 cache
write back：写回
write once：写一次
write through：写直达

思考题

4.1 顺序存取、直接存取和随机存取三者有何不同？
4.2 存取时间、存储器成本和容量之间的一般关系是什么？
4.3 局部性原理如何与多级存储器的使用相联系？
4.4 直接映射、全相联映射、组相联映射间的区别是什么？
4.5 对于一个直接映射 cache，主存地址可看成由 3 段组成。请列出并定义它们。
4.6 对于一个全相联映射 cache，主存地址可以看成由 2 段组成。请列出并定义它们。
4.7 对于一个组相联映射 cache，主存地址可看成由 3 段组成。请列出并定义它们。
4.8 空间局部性和时间局部性的区别是什么？
4.9 通常，利用时间局部性和空间局部性的策略是什么？

习题

4.1 一个组相联 cache 由 64 个行组成，每组 4 行。主存储器包含 4K 个块，每块 128 字。请表示主存地址的格式。
4.2 一个二路组相联 cache 具有 8KB 容量，每行 16 字节。64MB 的主存是字节可寻址的。请给出主存地址格式。
4.3 对于十六进制主存地址：111111、666666、BBBBBB，请用十六进制格式表示如下信息：
(a) 使用图 4-10 的格式的直接映射 cache 的标记、行和字的值。
(b) 使用图 4-12 的格式的全相联映射 cache 的标记和字的值。
(c) 使用图 4-15 的格式的二路组相联 cache 的标记、组和字的值。
4.4 请给出下列值：
(a) 对于图 4-10 给出的直接映射 cache 的例子：地址长度，可寻址单元数，块大小，主存的块数，cache 的行数，标记的位数。
(b) 对于图 4-12 给出的全相联映射 cache 的例子：地址长度，可寻址单元数，块大小，主存的块数，cache 的行数，标记的位数。
(c) 对于图 4-15 给出的二路组相联映射 cache 的例子：地址长度，可寻址单元数，块大小，主存的块数每组行数，组数，cache 的行数，标记的位数。
4.5 考虑一个 32 位的微处理器，它采用 16KB 片内四路组相联 cache，假设 cache 每行包含 4 个 32 位字。画出此 cache 的框图，并在图中表示其结构和如何使用不同的地址域来确定 cache 是否命中。存储单元地址 ABCDE8F8 映射到 cache 的什么地方？
4.6 给出下列外部 cache 存储器的规范：四路组相联，每行包含 2 个 16 位字，总共能容纳主存储器的 4K 个 32 位字；使用可发出 24 位地址的 16 位处理器。利用上述相关信息设计 cache 的结构，并说明它如何转换处理器地址。
4.7 Intel 80486 有一个统一的片内 cache，其容量为 8KB，采用四路组相联结构，每块包含 4 个 32 位字。cache 组织成 128 组，每行有一个“行有效位”与另外 3 个位 B0、B1、B2（LRU 位）。在 cache 未命中时，80486 在一个总线存储器读周期内从主存读取一个 16 字节的行。画出简化的 cache 框图，并说明地址的不同字段是如何转换的。
4.8 考虑一台机器，其主存可以按字节寻址，容量是 2^{16} 字节，块大小为 8 字节。假设该机器使用一个包含 32 行的直接映射 cache。
(a) 16 位存储器地址如何划分成标记、行号和字节号？
(b) 如下地址的内容将存入 cache 的哪些行？

0001	0001	0001	1011
1100	0011	0011	0100
1101	0000	0001	1101
1010	1010	1010	1010

(c) 假设地址0001 1010 0001 1010的字节内容存入cache，那么与它同存一行的其他字节的地址各是什么？

(d) 存储器总共有多少字节能保存于cache中？

(e) 为何标记必保存在cache中？

4.9 Intel 80486采用的替换算法被称为“伪最近最少使用算法”。与128个4行组（标记为L0，L1，L2和L3）相关的是3个位：B0，B1和B2。替换算法工作原理如下：当某一行必须被替换时，cache首先判断最近使用的是来自L0和L1，还是L2和L3，然后判断哪一对块是最近最少使用的，将它标记为替换。图4-20说明了此逻辑。

(a) 试说明如何设置B0，B1和B2位，并使用语言描述如何在图4-20表示的替换算法中使用它们。

(b) 试说明80486采用的算法接近于真正的LRU算法。提示：考虑最近使用的顺序是L0，L2，L3，L1的情况。

(c) 证明真正的LRU算法要求每组6位。

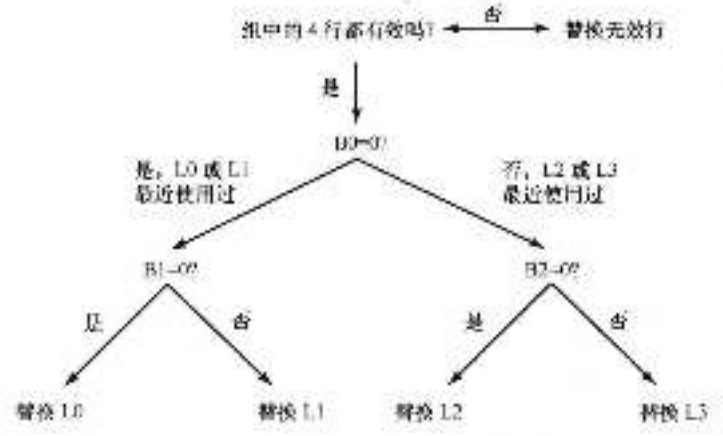


图4-20 Intel 80486片上cache的替换算法

4.10 一个组相联cache，每块大小为4个16位字，组大小为2，cache总共能容纳4096个字。可缓存的主存容量为64K×32位。设计一种cache的结构，并说明如何转换处理器的地址。

4.11 考虑一个由32位地址字节级寻址的主存和行大小为64B的cache所组成的存储子系统。

(a) 假定cache采用直接映射技术，并且地址中的标记字段为20位。请给出地址格式并确定下列参数：可寻址单元数，主存的块数，cache的行数。

(b) 假定cache采用全相联映射技术。请给出地址格式并确定下列参数：可寻址单元数、主存的块数、cache的行数、标记的长度。

(c) 假定cache采用4路组相联映射技术，并且地址中的标记字段为9位。请给出地址格式并确定下列参数：可寻址单元数，主存的块数，组中的行数，cache的组数，cache的组数，cache的行数，标记的长度。

4.12 考虑一个具有下述特征的计算机：主存1MB，字长1B，块大小16B；cache容量64KB。

(a) 若为直接映射cache，请给出主存地址F0010、01234和CABBE相应的标记、cache行地址和字偏移。

(b) 若为直接映射cache，试给出映射到同一cache行而有不同标记的两个主存地址。

(c) 若为全相联映射cache，请给出主存地址F0010和CABBE相应的标记和字偏移。

(d) 若为二路组相联cache，请给出主存地址F0010和CABBE相应的标记、组号和字偏移。

4.13 描述在四路组相联cache中实现LRU替换算法的简单技术。

4.14 再次考虑例4.3。若主存使用块传送方式，第一个字的存取时间是30ns，后续相邻地址的每个字的存取时间是5ns，答案将发生怎样的变化？

4.15 考虑如下代码：

```
For (i = 0; i < 20; i++)
  For (j = 0; j < 10; j++)
```

```
a[i][j] = a[i][j] * f
```

(a) 给出代码中空间局部性的一个例子。

(b) 给出代码中时间局部性的一个例子。

4.16 将附录4A中的式(4.1)和式(4.2)推广到N级的存储器分层结构。

4.17 计算机系统包含容量为32K×16位的主存，具有4KB的cache，每组4行，每行64个字。假设cache初始时是空的，处理器顺序地从存储单元0，1，2，…，4351中取数，然后它再重复这一顺序9次，并且cache比主存快10倍，同时假设块替换使用LRU算法。请估算一下使用cache后系统性能的改进。

4.18 考虑一个4行且每行16字节的cache，主存按每块16字节划分，即块0有地址0到15的16个字节，等等。现在考虑一个程序，它以如下地址顺序访问主存：

一次：63～70

循环10次：15～32，80～95

(a) 假设cache采用直接映射技术，主存块0，4，…指派到行0；块1，5，…指派到行1；依次类推。请计算命中率。

(b) 假设cache采用二路组相联映射技术，共有两组，每组两行。偶序号块被指派到组0，奇序号块被指派到组1。请计算使用LRU替换策略的二路组相联cache的命中率。

4.19 考虑有下列参数的存储器系统：

$$T_c = 100\text{ns} \qquad C_c = 10^{-4}\text{美元/位}$$

$$T_m = 1200\text{ns} \qquad C_m = 10^{-5}\text{美元/位}$$

(a) 1MB主存的价格是多少？

(b) 使用cache技术的1MB主存的价格是多少？

(c) 如果该存储系统的有效存取时间比cache存取时间大10%，则其命中率是多少？

4.20 (a) 考虑一个存取时间为1ns、命中率$H=0.95$的L1 cache。假设我们能修改此cache的设计（cache容量、cache组织），从而使命中率H提升到0.97，但主存存取时间增大到1.5ns。若要改善此设计性能，则这个改变必须满足什么条件？

(b) 解释这个结果为什么有直观意义。

4.21 考虑一个单级cache，其存取时间是2.5ns，行大小是64字节，命中率$H=0.95$。主存使用块传送，第一个字（4字节）存取时间是50ns，其后的每个字存取时间是5ns。

(a) 出现一次cache缺失的存取时间是多少？假设此时cache等待，直到此行从主存取来，然后才作为命中而重复执行。

(b) 假定行大小增大到128字节时，命中率H提升到0.97。这是否降低了平均存储器存取时间？

4.22 计算机有cache、主存和用于虚拟存储器的磁盘。若所访问的字在cache中，则存取它只需要20ns。若该字在主存中而不在cache中，则需要60ns将它装入cache，然后再从cache中存取。若该字不在主存中，则需要12ms将它由磁盘取来装入主存，再用60ns将它复制到cache，最后从cache存取。cache命中率是0.9，主存命中率是0.6，那么此系统访问一个字的平均存取时间是多少（以ns为单位）？

4.23 考虑一个行大小为64字节的cache。假定cache中平均30%的行是脏数据。一个字由8个字节组成：

(a) 假定缺失率为3%（0.97的命中率），对于写直达和写回两个写策略，通过每指令字节数来计算主存的通信量。由主存读入cache是一次一行；然而，对写回策略，一个单字能由cache写到主存。

(b) 若缺失率为5%，重复问题(a)。

(c) 若缺失率为7%，重复问题(a)。

(d) 由这些结果你能得出什么结论？

4.24 在Motorola 68020微处理器上，一次cache存取占用两个时钟周期。而在无等待状态插入的情况下，由主存经总线到处理器的数据存取占用3个时钟周期；且数据并行地交给处理器和cache。

(a) 给定命中率为0.9和时钟速率为16.67MHz，请计算存储器周期的实际长度。

(b) 假定每个存储器周期插入两个等待状态（1个等待状态为1个时钟周期），重复上面的计算。由这些结果你能得出什么结论？

4.25　假定一个处理器具有 300ns 的存储器周期时间和 1MIPS 的指令处理速率。平均而言，每条指令要求一个总线存储器周期用于取指令，另要求一个周期用于指令所涉及的操作数。

（a）试计算处理器的总线利用率。

（b）假设此处理器配备了一个指令 cache，其相应的命中率为 0.5。试确定这对总线利用率的影响。

4.26　一个单级 cache 系统的读操作性能可用下式来表征：

$$T_s = T_c + (1 - H)\ T_m$$

其中，T_s 是平均存取时间，T_c 是 cache 存取时间，T_m 是存储器存取时间（主存到处理器寄存器），H 是命中率。为简单起见，假定字是并行装入到 cache 和处理器寄存器的。上式是与等式 4.2 相同的格式。

（a）定义：T_w = 在 cache 和主存之间传送一行的时间，W = 在所有访问中，写访问占的比率。试使用写直达策略，改写上式使之既考虑到读也考虑到写。

（b）定义 W_c 为 cache 中的一行已被修改的概率，试为写回策略提供一个 T_s 等式。

4.27　对一个有两级 cache 的系统，定义：

T_{c1} = 第一级 cache 存取时间；

T_{c2} = 第二级 cache 存取时间；

T_m = 存储器存取时间；

H_1 = 第一级 cache 命中率；

H_2 = 组合的第一/二级 cache 命中率。

请对读操作给出 T_s 等式。

4.28　假定 cache 读缺失有如下性能特征：用 1 个时钟周期发送地址到主存和用 4 个时钟周期由主存读取一个 32 位字传送给处理器和 cache。

（a）若 cache 行大小是 1 个字，缺失开销是多少（即为读所要求的附加时间）？

（b）若 cache 行大小是 4 个字，并执行多字的非突发式块传送，cache 的缺失开销是多少？

（c）若 cache 行大小是 4 个字，并执行突发式块传送，传送每个字用 1 个时钟周期，读缺失开销是多少？

4.29　上题的 cache 设计中，若行大小由 1 个字增加到 4 个字，则导致读缺失率由 3.2% 下降到 1.1%。对于两种行大小，计算突发式和非突发式两种传送的平均读缺失代价。

附录 4A　两级存储器的性能特点

本章介绍了作为处理器与主存之间缓冲器的高速缓存，创建了两级内部存储器。这种两级存储结构探讨了被称为局部性的特性，与单级存储器相比，它提高了性能。

主存储器高速缓存机制是计算机体系结构的一部分，它由硬件实现，而且通常对操作系统是透明的。还有另外两种二级存储器的方法也利用了局部性特征，并且它们是（至少是部分）由某些操作系统实现的：虚拟存储器和磁盘高速缓存（如表 4-7 所示）。虚拟存储器将在第 8 章中讨论，磁盘高速缓存则超出了本书的讨论范围，但参考文献［STAL05］对它进行了考察。在本附录中，我们考虑这 3 种方法所共有的两级存储器的性能特点。

表 4-7　两级存储器的特性

	主存储器 cache	虚拟存储器（分页）	磁盘 cache
典型的存取时间比	5:1（主存比 cache）	10^6:1（主存储器比磁盘）	10^6:1（主存储器比磁盘）
存储管理系统	由专门硬件实现	硬件和系统软件的结合	系统软件
典型的块或页大小	4～128 字节（cache 块）	64～4096 字节（虚拟存储器页）	64～4096 字节（磁盘块或页）
处理器对第二级存储器的存取	直接存取	间接存取	间接存取

4A.1　局部性

两级存储器性能优越性的基础是访问的局部性原理（参见［DENN68］）。该原理表明，存储器要访问的

数据多集中在某个局部（成簇）。虽然在较长的时间里使用的簇（cluster）会改变，但在短时间内处理器会集中访问几个固定的簇。

显然，局部性原理是有意义的，考虑如下理由：

（1）除了分支和调用指令（它们只占所有程序指令的一小部分）以外，程序的执行是顺序的。因此，在大多数情况下，下一次要取的指令紧跟着上一次取得的指令。

（2）很长的无中断的子程序调用序列后跟着相应数量的返回的情况是很少的。程序通常限制在一个较窄的子程序调用深度的窗口内，因此，在短时间内，指令的访问趋向于局限在几个子程序中。

（3）所有迭代结构由相对较少的指令组成，使它们重复运行多次。在迭代期间，计算限制于一小段程序中。

（4）在许多程序中，很多计算涉及数据结构的处理，例如数组或顺序的记录。许多情况下，对这些数据结构中相继访问的数据项是连续存放的。

这些理由在许多研究中得到了证实。例如，让我们考虑第（1）点。许多研究分析了高级语言程序的行为。下面的研究统计了在执行中各种高级语言出现的频度，表4-8包含了关键的结果。Knuth［KNUT71］最早研究了程序设计语言的行为，他考察了一组用做学生练习的FORTRAN程序。Tanenbaum［TANE78］公布了从300多个过程中收集的统计结果，这些过程来自操作系统程序，而且以支持结构化程序设计（SAL）的语言写成。Patterson和Sequein［PATT82a］分析了一组来自编译器和用于非版、计算机辅助设计（CAD）、排序和文件比较程序的统计结果。他们研究的是Pascal和C语言。Huck［HUCK83］分析了4个用于代表通用科学计算的程序，它们包括快速傅里叶变换和微分公式的综合系统。从这些不同的语言和应用得到的结果有着较好的一致性。它们表明，在一个程序的执行期间，分支和调用只占执行指令的一小部分。因此，这些研究结果证实了上面的第（1）点。

表4-8 高级语言操作的相对动态频度

参考文献	［HUCK83］	［KNUT71］	［PATT82a］		［TANE78］
语言	Pascal	FORTRAN	Pascal	C	SAL
工作负载	科学计算	学生练习	系统程序	系统程序	系统程序
Assign（赋值）	74	67	45	38	42
Loop（循环）	4	3	5	3	4
Call（调用）	1	3	15	12	12
IF（条件判断）	20	11	29	43	36
GOTO（转移）	2	9	—	3	—
Other（其他）	—	7	6	1	6

对于第（2）点，参考文献［PATT85a］的研究报告提供了证实。图4-21对其进行了说明，它显示了调用返回的行为。每次调用由向右下倾斜的线段表示，而每次返回由向右上倾斜的线段表示。图中定义了一

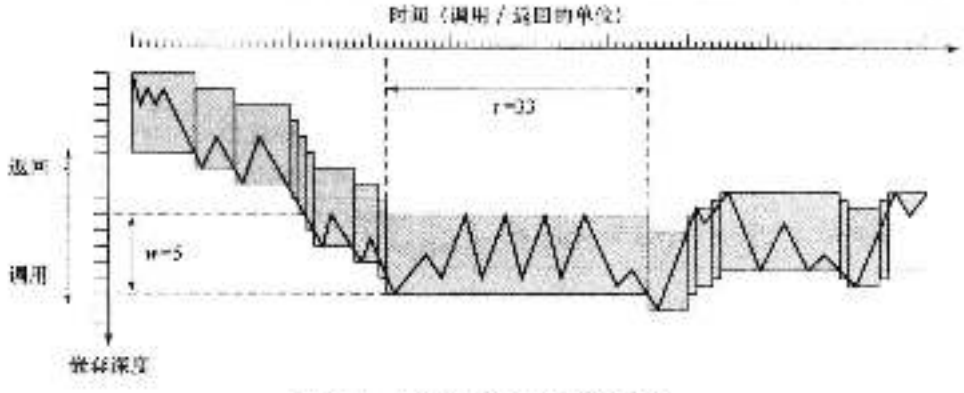


图4-21 程序调用/返回行为的实例

个宽度为5的窗口。只有在某个方向上连续移动为6的调用和返回序列才会引起窗口的移动。正如我们所见到的，程序的执行能够在一个固定的窗口内等留很长时间。对C和Pascal程序的研究表明，深度为8的窗口的移动只占调用和返回次数的不到1% [TAMI83]。

文献中常将局部性区分为空间局部性和时间局部性。空间局部性（spatial locality）是指程序执行涉及大量成簇的存储位置的倾向性。空间局部性既反映了处理器顺序访问指令的倾向性，也反映了程序顺序访问数据单元的倾向性。例如处理一个数据表时。时间局部性（temporal locality）是指处理器访问刚被使用的存储位置的倾向性。例如，执行一个循环时，处理器重复地执行同一组指令。

传统上，通过将最近刚使用的指令和数据保存在cache中并利用cache层次结构来开拓时间局部性。而通过使用较大cache块并将预取机制（预取期望用到的项）纳入cache控制逻辑来开拓空间局部性。近来，出现了相当多的关于精炼这些技术以便更大地提高性能的研究，但是基本策略还是一样的。

4A.2 两级存储器的操作

可以按两级存储器的形式来利用局部性特征。与低一级存储器（M2）相比，高一级存储器（M1）更小、更快、更贵（每一位）。M1用作临时存储器，用来存放M2的部分内容。当访问存储器时，先试图访问M1中相应的块。如果成功，则实现了一次快速访问；否则，块得从M2复制到M1，然后再通过M1访问。根据局部性原理，一旦某个块放入M1，将会对这个块的单元访问多次，这使得总体服务速度变快。

为了表示访问一个项的平均时间，不但必须考虑两级存储器的速度，而且要考虑其次访问能在M1中找到的概率。有：

$$\begin{aligned} T_s &= H \times T_1 + (1-H) \times (T_1 + T_2) \\ &= T_1 + (1-H) \times T_2 \end{aligned} \tag{4.2}$$

其中：T_s =（系统）平均存取时间

T_1 = M1（例如，高速缓存、磁带高速缓存）的存取时间

T_2 = M2（例如，主存储器、磁带）的存取时间

H = 命中率（在M1中找到所需信息的概率）

图4-2显示了平均存取时间随命中率的变化函数。可见，对于较高的命中率，平均访问时间更接近于M1，而不是M2。

4A.3 性能

下面考虑与两级存储器机制相关的一些参数。首先考虑价格，有如下计算公式：

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{4.3}$$

其中：C_s = 两级存储器的每位平均价格

C_1 = 高一级存储器M1的每位平均价格

C_2 = 低一级存储器M2的每位平均价格

S_1 = M1的容量

S_2 = M2的容量

我们希望 $C_s \approx C_2$，由于 $C_1 \gg C_2$，这要求 $S_1 \ll S_2$。图4-22显示了这一关系。

下面考虑访问时间。由于两级存储器导致地改进了性能，我们需要让 T_s 约等于 T_1（$T_s \approx T_1$）。因为 T_1 比 T_2 小得多（$T_1 \ll T_2$），所以需要命中率接近于1。

因此，我们希望M1小，以降低价格；又希望它大，因为这样才能提高命中率，从而改进性能。是否存在一个在合理的范围内满足这两种要求的容量呢？回答这个问题需解决以下几个子问题。

- 为使 T_s 近似等于 T_1 需要什么样的命中率？
- M1的容量为多少才能获得需要的命中率？
- 是否这一容量满足了价格要求？

为达到这个目的，考虑数量 T_1/T_s，称为存取有效性。它表示了平均访问时间（T_s）与M1访问时间（T_1）的近似程度。从公式（4.2）可得：

$$\frac{T_1}{T_s} = \frac{1}{1 - (1-H)\dfrac{T_2}{T_1}} \tag{4.4}$$

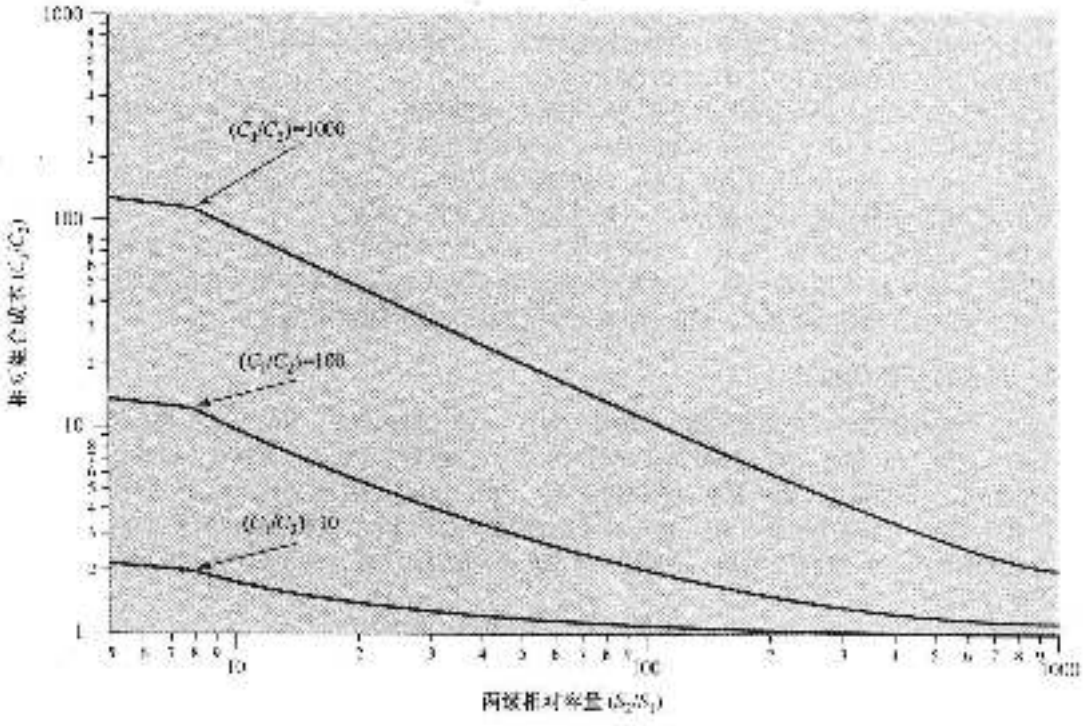


图 4-22 两级存储器平均存储器成本和相对存储容量之间的关系

图 4-23 画出了 T_1/T_2 与命中率 H 的关系函数，数据 T_2/T_1 作为参数。通常，片上 cache 的存取时间比主存的存取时间快 25 ~ 50 倍（即 T_2/T_1 为 5 ~ 10），片外 cache 的存取时间比主存的存取时间快 5 ~ 15 倍（即 T_2/T_1 为 5 ~ 15），而主存储器的存取时间比磁盘快大约 1000 倍（$T_2/T_1 = 1000$）。因此，命中率在近于 0.9 左右的范围即能满足性能要求。

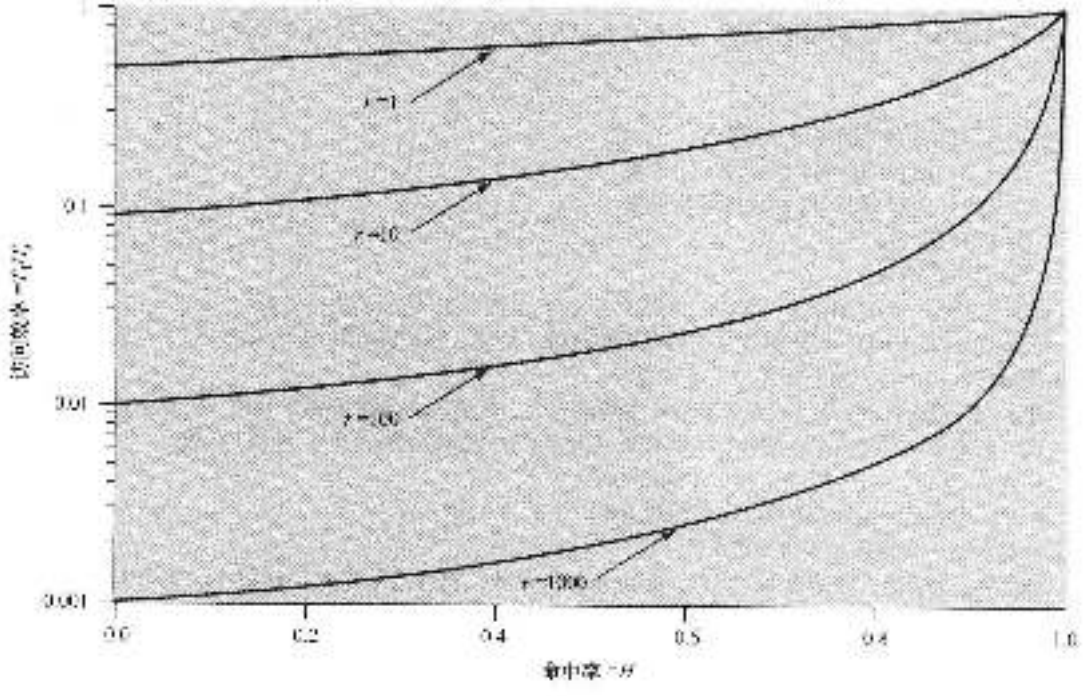


图 4-23 访问效率与命中率的函数关系（$r = T_2/T_1$）

现在我们能够更确切地表述相对的存储器容量的问题。是否命中率为0.8以上就可以使 $S_1 << S_2$？这将依赖于许多因素，它们包括所执行软件的特性和两级存储器的设计细节。具有决定性的因素是局部性的程度。图4-24为局部性对命中率的效果。显然，如果M1与M2的容量相同，那么命中率为1.0，因为所有M2的内容都存储在M1中。现在假设没有局部性，也就是说，访问完全是随机的。在这种情况下，命中率是相对存储器容量的严格线性函数。例如，如果M1是M2容量的一半，那么任何时刻M2的一半内容在M1中，命中率为0.5。实际上，访问总是存在一定程度的局部性。中等局部性和强局部性的影响都在图中描述。注意，图4-24不是从任何特殊的数据或模型中推导得出的，而是该图表示了随局部性程度的不同性能也会不同。

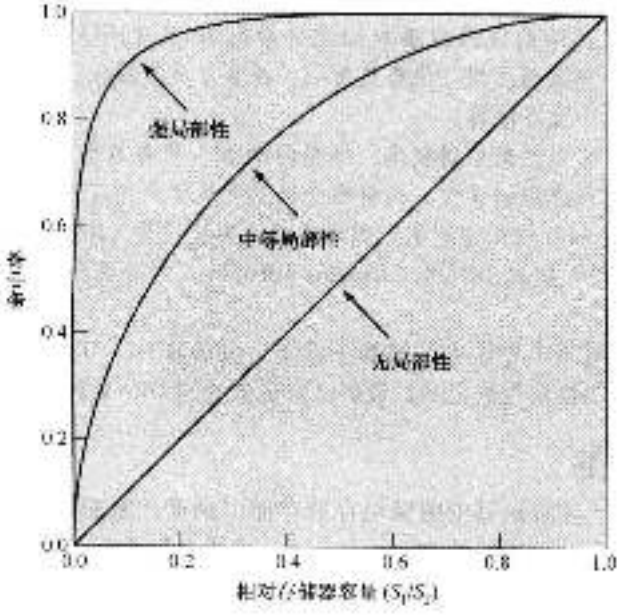


图4-24 命中率和相对存储容量的函数关系

同此，如果存在较强的局部性，即使上层存储器的容量较小，也能达到高的命中率。例如，很多研究表明，即使很小的高速缓存也能产生高于0.75的命中率，不管主存的容量是多少（参见[AGAR89]、[PRZY88]、[STRE83]和[SMIT82]）。通常最速缓存的容量在1KB到128KB的范围就足够了，而现在的主存容量通常是以GB衡量。在考虑虚拟存储器和磁盘高速缓存时，我们引用的其他研究结果证实了同一现象，即由于局部性，相对小的M1产生了高的命中率。

现在来回答前面所列的最后一个问题：是否这两种存储器的相对容量就足价格要求？答案是肯定的，只要为了达到高性能只需要一个相对较小的高一级存储器，那么该两级存储器每位的平均价格将接近于较便宜的低一级存储器。

请注意，若涉及L2 cache，甚至涉及L2和L3 cache，分析将会复杂得多。详见[PTER99]和[HAND98]的讨论。

第5章

Computer Organization and Architecture: Designing for Performance, 8E

内部存储器

本章要点

- 半导体随机存取存储器的两种基本形式是动态 RAM（DRAM）和静态 RAM（SRAM）。SRAM 比 DRAM 存取速度快、价格更昂贵，并且集成度较低。一般用于 cache 存储器。而 DRAM 则一般用于主存储器。
- 存储器系统中通常都使用纠错技术，这包括加上一些与原数据位成函数关系的冗余位来构成纠错码。如果出现错误位，纠错码会检测并通常能纠正该错误位。
- 为了补偿 DRAM 相对较低的速度，现已推出几种先进的 DRAM 组织。使用最普遍的两种是同步 DRAM 和总线式 DRAM（Rambus DRAM）。两者都使用系统时钟以支持握手协议。

本章首先概要性地讨论半导体主存储器子系统，包括 ROM、DRAM 和 SRAM 存储器；然后分析提高存储器可靠性的错误控制技术；最后介绍更先进的 DRAM 体系结构。

5.1 半导体主存储器

在早期的计算机中，主存储器中的随机存取存储器最通用的形式是使用一组环形的铁磁体圈，称为磁芯。因此，主存储器通常称为核心，这一术语沿用至今。在磁芯存储器消失以前，微电子技术已经出现了很久，优势已很明显。目前，几乎所有的主存储器都采用半导体芯片。本节将讨论这一技术的关键方面。

5.1.1 组织

半导体存储器的基本元件是存储位元。虽然有各种电子技术可采用，但所有的半导体存储位元都具有某些相似的性质：

- 呈现两种稳态（或半稳态），分别代表二进制的 1 和 0；
- 能够写入信息（至少一次）来设置状态；
- 能够读出状态信息。

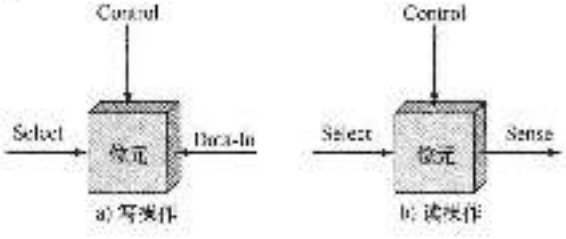


图 5-1 存储器位元操作

图 5-1 是一个存储位元的操作示意图。最普遍地，每个位元有 3 个能传输电信号的功能端。顾名思义，Select（选择）端口用于为读或写操作选择一个存储位元。Control（控制）端口指明是读还是写操作。对于写操作，为一端口 Data-In（数据输入）提供设置位元状态为 1 或 0 的电信号；对于读操作，Sense（感应）端口用于输出位元的状态。存储位元的内部结构、功能、时序的细节依赖于所采用的特定的集成电路技术，这超出了本书所要介绍的范围，只做简要的总结。我们的目的是给出单个存储位元，它能被选择用于读或写操作。

5.1.2 DRAM 和 SRAM

本章将讨论的所有存储器类型都是随机存取的，即通过编排的寻址逻辑，存储器的单个字

直接被存取。

表5-1列出了半导体存储器的主要类型。最常用的是随机存取存储器（random-access memory，RAM）。当然，这是术语误用，因为表中列出的所有类型的存储器都是随机存取的。RAM的明显特征是，可以方便快捷地从存储器读取数据和向存储器写入新数据，且读写操作都是通过使用电信号来完成的。

表5-1 半导体存储器类型

存储器类型	种类	可擦除性	写机制	易失性
随机存取存储器（RAM）	读/写存储器	电可擦除、字节级	电	易失
只读存储器（ROM）	只读存储器	不可能	掩膜	非易失
可编程 ROM（PROM）			电	
可擦除 PROM（EPROM）	主要进行读操作的存储器	紫外线可擦除，芯片级		
电可擦除 PROM（EEPROM）		电可擦除，字节级		
快闪存储器		电可擦除，块级		

RAM另一个明显特征是易失性。RAM必须持续供电，一旦断电，数据就会丢失。因此，RAM仅能用于暂时存储。计算机中使用的两种传统的RAM形式是DRAM和SRAM。

1. 动态RAM

RAM技术分为动态和静态两类。动态RAM（dynamic RAM，DRAM）利用电容充电来存储数据，位元中的电容有、无电荷分别代表二进制的1或0。因为电容器有漏电的自然趋势，因此动态RAM需要周期地充电刷新来维持数据的存储。动态一词就是指这种存储电荷丢失的趋势，即使电源一直在供电。

图5-2a是存储1位信息的单个位元的典型DRAM结构。当要读出或写入该位元的位值时，激励地址线。晶体管像开关一样工作，如果有电压施加到地址线上，晶体管导通；如果无电压施加到地址线上，则晶体管开路（无电流通过）。

对于写操作，一个电压信号施加到位线上：高电压代表1，低电压代表0。然后一个信号施加到地址线，允许电荷传输到电容器。

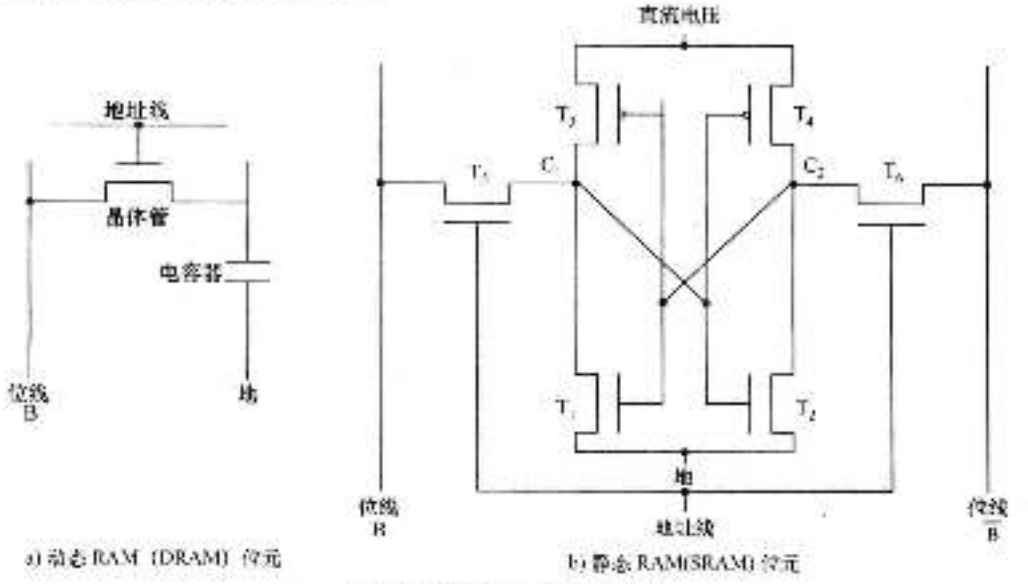


图5-2 典型存储器位元结构

对于读操作，当地址线被选中时，晶体管导通，存储在电容上的电荷被送出到位线和读出放大器。读出放大器将此电容电压与一参考值进行比较，并确定位元保存的是逻辑 1 还是逻辑 0。位元的读出放掉了电容上的电荷，必须重新存储才算完成本次操作。

虽然 DRAM 位元能用来存储单一位值（0 或 1），但它本质上是一个模拟设备。因为电容能存储一定范围内的任何电荷值，因此必须使用一个阈值来确定该电荷值代表的是 1 还是 0。

2. 静态 RAM

相对而言，静态 RAM（static RAM，SRAM）是一个数字设备，它使用与处理器相同的逻辑元件。静态 RAM 采用传统的触发器、逻辑门配置来存储二进制值（请见第 20 章中有关触发器的描述）。只要电源不断，SRAM 将一直保持它所存储的数据。

图 5-2b 是单个位元的典型 SRAM 结构。4 个晶体管（T_1、T_2、T_3 和 T_4）交叉连接组成一个有稳定逻辑状态的排列。在逻辑状态 1 下，C_1 点是高电平而 C_2 点是低电平，此时，晶体管 T_1 和 T_4 截止，而 T_2 和 T_3 导通[1]。在逻辑状态 0 下，C_1 点是低电平而 C_2 点是高电平，此时，晶体管 T_1 和 T_4 导通，而 T_2 和 T_3 截止。只要直流电源一直供电，这两个状态就都是稳定的。不同于 DRAM，这里不需要刷新来维持数据。

如同 DRAM 中一样，SRAM 地址线用来控制开关的通断。这里，地址线控制两个晶体管（T_5 和 T_6），当信号施加到地址线上时，两个晶体管导通，允许读/写操作。对于写操作，位值施加到 B 线，位值的反值加到 $\overline{B}$ 线，这强迫 4 个晶体管（T_1、T_2、T_3 和 T_4）进入一个相应的稳态。对于读操作，位值由 B 线读出。

3. SRAM 与 DRAM 对比

静态与动态 RAM 都是易失的，即二者都要求电源持续供电才能保存位值。与静态存储器位元相比，动态存储器位元要小，而且电路更简单。因此，与 SRAM 相比，DRAM 的密度要高（较小的位元 = 每单位面积上更多的位元），且价格更便宜。另一方面，DRAM 要求有支持刷新的电路。但是，对于较大容量的存储器，DRAM 位元较低的可变成本足以补偿刷新电路的固定成本。因此，DRAM 更趋向于满足大容量存储器的需求。最后还需指出，通常 SRAM 要比 DRAM 快。由于这些相对特征，SRAM 一般用于 cache 存储器（片上的或片外的），而 DRAM 则用于主存储器。

5.1.3 ROM 类型

顾名思义，只读存储器（read-only memory，ROM）含有不能改变的永久性数据。ROM 是非易失性存储器，即存储器中的数据并不要求供电来维持。ROM 可读，但不能写入新数据。ROM 的一个重要应用是微程序设计，这将在第四部分讨论。其他可能的应用包括：常用功能的子程序库、系统程序、函数表。

对于中等规模的要求，ROM 的优点是数据或程序可永久地保存在主存中，绝不需要从辅存中调入。

制造 ROM 同制造其他集成电路芯片是一样的，在制造过程中把数据固化到芯片上。这存在两个问题：

- 固化数据需要较大的固定成本，不论是制造一片还是复制上千片特殊的 ROM。
- 无出错处理机会，如果一位出错，则整批的 ROM 芯片只能报废。

当只需要少量的存储特定内容的 ROM 芯片时，可选择较廉价的可编程 ROM（programmable ROM，PROM）。和 ROM 一样，PROM 是非易失性的，但它能写入一次，且只能一次。对于 PROM，

[1] T_3 和 T_4 旁边的圆圈表示信号反向。

写过程是用电信号执行，由供应商或用户在芯片生产后写入一次。需要特殊设备来完成写或"编程"过程。PROM 提供了灵活性和方便性，而 ROM 在大批量生产领域仍具有吸引力。

只读存储器的另一种变体是主要进行读操作（read-mostly）的存储器，常用于读操作远多于写操作且要求非易失数据的应用场合。常见的主要进行读操作的存储器有 3 种：EPROM、EEPROM 和快闪存储器。

典型的光可擦除/可编程只读存储器（erasable programmable read-only memory，EPROM）与 PROM 一样可读可写。然而，在写入操作前，必须通过让封装芯片暴露在紫外线辐射下使所有的存储位元都被擦除，以还原成初始状态。擦除需要通过让设计在芯片上的窗口在强紫外线下长时间照射来完成。这种擦除过程可重复进行，每次擦除需要约 20 分钟。因此，EPROM 可以修改多次，并且和 ROM、PROM 一样能够长久保存数据。对于等容量的存储器来说，EPROM 比 PROM 更贵，但它具有可多次改写的优点。

更具吸引力的主要进行读操作的存储器形式是**电可擦除/可编程只读存储器**（electrically erasable programmable read-only memory，EEPROM）。这种存储器在任何时候都可写入，而无需擦除原先内容，且只更新寻址到的一个或多个字节。写操作比读操作时间要长得多，每字节需要几百微秒的时间。EEPROM 把非易失性和数据修改灵活的优点结合起来，修改数据时只需要使用常规的控制、地址和数据总线。EEPROM 比 EPROM 贵，且密度低，支持小容量芯片。

另一种半导体存储器是**快闪存储器**（flash memory），这是由于其重编程速度快而得名。快闪存储器在 20 世纪 80 年代中期首次推出，其价格和功能介于 EPROM 和 EEPROM 之间。与 EEPROM 相似，快闪存储器使用电擦除技术，整个快闪存储器可以在一秒或几秒钟内被擦除，速度比 EPROM 快得多。另外，它能擦除存储器中的某些块，而不是整块芯片。快闪存储器用于微芯片中，一次或"一瞬间"可以只擦除一部分存储器位元，因此而得名。然而快闪存储器不要供字节级的擦除。像 EPROM 一样，快闪存储器每位只使用一个晶体管，因此，与 EEPROM 相比较，能获得与 EPROM 一样的高存储密度。

5.1.4 芯片逻辑

如同其他集成电路产品，半导体存储器也是封装的芯片（如图 2-7 所示）。每块芯片包含一组存储位元阵列。

在整个存储器层次结构中，需要在速度、容量和价格之间进行权衡。当我们考虑芯片的存储位元组织和功能逻辑时，也要做这些权衡。对于半导体存储器，一个关键的设计问题是每次可以读/写数据的位数。一种极端的情况是阵列中位元的物理排列与存储器中字的逻辑排列（从处理器的角度看）相同。阵列组织成 W 个字，每个字 B 位。例如，16Mb 的芯片能够组织成 1M 的 16 位字。另一种极端的情况是所谓的每芯片一位的结构，此时数据每次只能读/写 1 位。可以用一个 DRAM 来说明存储器芯片的结构；ROM 的结构与之类似，且更简单。

图 5-3 表示了 16Mb DRAM 的一种典型结构。这种情况下，一次读或写 4 位。逻辑上，存储器组织成 4 个 2048 × 2048 的方阵。可以采用各种物理排列。在任一种情况下，阵列元素由行（row）控制线和列（column）控制线连接，每根行控制线连接到它所在行中每个位元的 Select 端口，而每根列控制线连接到相应列中每个位元的 Data-In/Sense 端口。

地址线提供了被选择字的地址，总共需要 $\log_2 W$ 条线。在这个例子中，需要 11 根地址线来选中 2048 行中的一行，这 11 根地址线连接到行译码器的输入线。行译码器有 11 根输入线和 2048 根输出线，其逻辑依据 11 根输入线的位模式激活 2048 根输出线中的一根（$2^{11} = 2048$）。

另外的 11 根地址线可选中 2048 列中的一列，每列出 4 位组成。4 根数据线用于与数据缓冲器交换 4 位数据。输入（写）时，每根位线的位驱动器根据相应数据线的值被激活为 1 或 0；输出（读）时，每根位线的值经过读出放大器，传送到数据线上。行线选择那一行的位元参与读

或写操作。

因为此 DRAM 每次只有 4 位位元参与读/写，因此，必须将多片 DRAM 连接到 DRAM 控制器上才能读/写一个字到总线上。

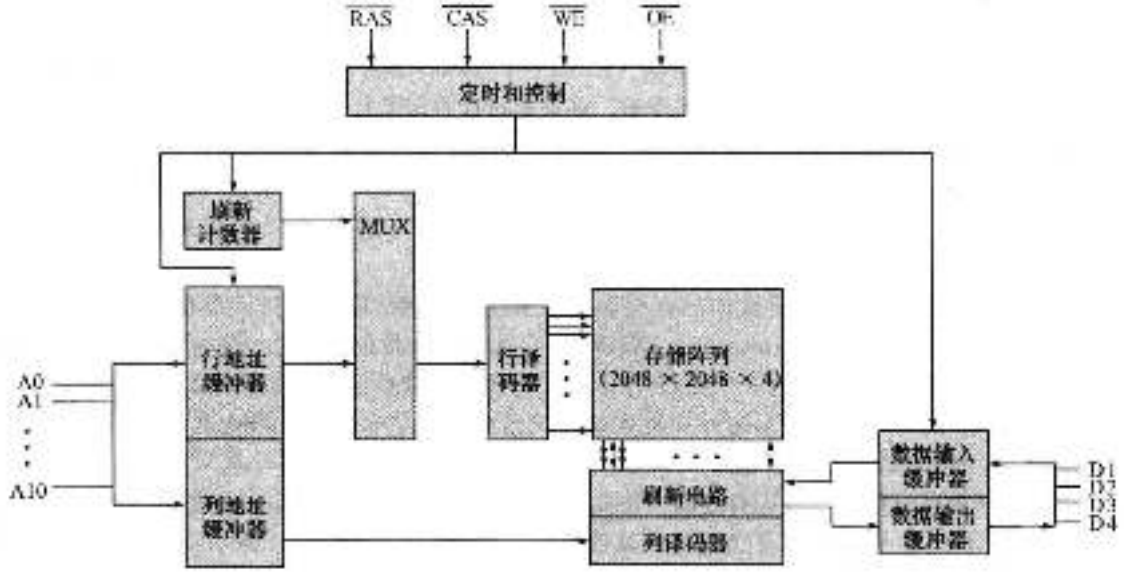


图 5-3 典型的 16Mb DRAM（4M×4）

注意，11 根地址线（A0 – A10）只是选中 2048×2048 阵列所需地址位数的一半，这样做是为了节省引脚数。22 根需要的地址线通过外部的选择逻辑连接到芯片，并且被复用为 11 根地址线。首先，11 位地址信号经过芯片去定义阵列的行地址，然后，另 11 位地址信号去定义列地址。伴随着这些信号有行地址选通（$\overline{RAS}$）信号和列地址选通（$\overline{CAS}$）信号，为芯片提供时序控制信号。

写允许（WE）和输出允许（$\overline{OE}$）引脚确定完成的是写或是读操作。图 5-3 中未示出的另外两个引脚是地（V_{ss}）和电源（V_{cc}）。

此外，地址线的复用和方阵型行列结构，导致每出现新一代存储器芯片，其容量就以 4 倍的方式增长。每增加一个专用的地址引脚，便使行地址和列地址的指示范围加倍，因此存储器芯片的容量以 4 的倍数增长。

图 5-3 中还包含了刷新电路，所有的 DRAM 都需要刷新操作。实际上，简单的刷新技术是使 DRAM 芯片丧失读写能力而刷新所有数据位元。刷新计数器遍历遍过所有的行值。对每一行，刷新计数器的值被当作行地址输出到行译码器，并且激活 RAS 线，数据被读出后又写回原地址，从而使得相应行的所有位元被刷新。

5.1.5 芯片封装

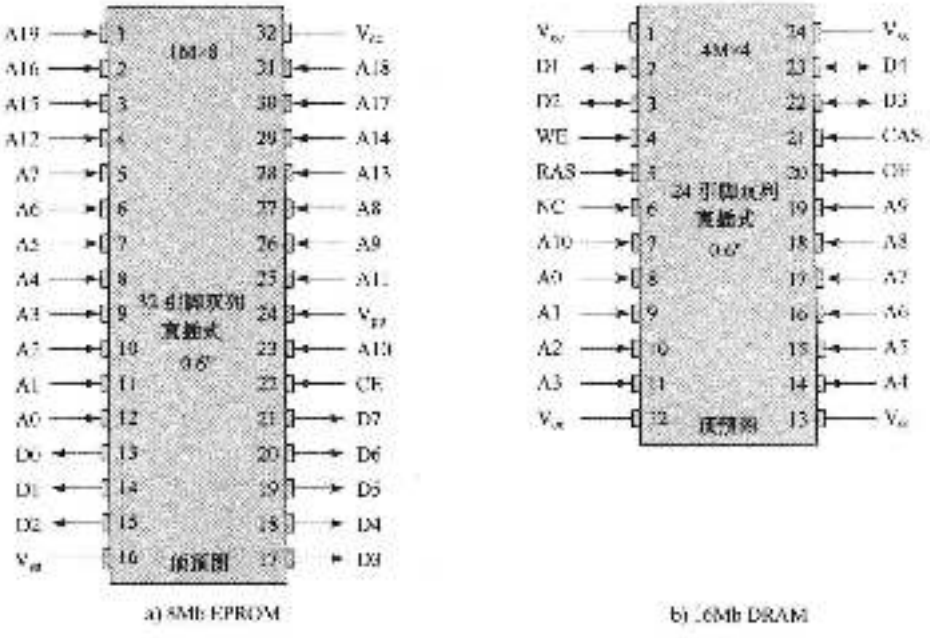
如同第 2 章所述，集成电路封装成组件，并包含有与外界相连接的引脚。

图 5-4a 是一个 EPROM 组件的例子，它是一个 1M×8 的 8Mb 芯片组织。此时，这种组织被看成由单个芯片提供整个字（one-word-per-chip）的形式。芯片组件包括 32 个引脚，属于标准的芯片组件尺寸，其引脚包含了下列信号线：

- 被访问字的地址。对于 1M 字，总共需要 20（2^{20} = 1M）根引脚（A0 – A19）。
- 读出的数据，包含 8 根线（D0 – D7）。

- 电源（V_{cc}）。
- 地线（V_{ss}）。
- 芯片允许（CE）引脚。因为可能有多个存储器芯片，每片都连接到相同的地址总线，CE引脚用于指示地址线上的地址对本芯片是否有效。CE引脚由连接到地址总线的高序位（如高于A19的地址位）的逻辑激活。该信号的使用将在后面说明。
- 程序电压（V_{pp}），在编程（写操作）时提供。

图5-4a给出了一个典型的DRAM引脚结构，此16Mb芯片的结构为4M×4。它与ROM芯片有所不同。由于RAM能修改，因此其数据引脚兼备输入、输出功能。写允许（WE）和输出允许（OE）引脚指明是写操作或是读操作。因为DRAM是由行和列存取，并且地址是多路复用的，所以只需要11根地址引脚来指定4M的行/列组合（$2^{11}\times 2^{11}=2^{22}$）。行地址选通（RAS）和列地址选通（CAS）引脚的功能前面已讨论过，这里不再赘述。最后，还有一个无连接（NC）引脚，以使引脚数凑成偶数。



a) 8Mb EPROM　　b) 16Mb DRAM

图5-4　典型存储器组件的引脚和信号

5.1.6　模块组织

如果一个RAM芯片仅仅包含每个字数据的一位，那么，所需的芯片数很显然至少等于每字的位数。例如，图5-5表示了一个包含256K个8位字的存储模块的组织形式。对于256K字，需要18根地址线，它们从外部提供给存储模块（例如，总线上的地址线连到存储模块）。地址输入到8个256K×1位的芯片，每片提供1位的输入/输出数据。

只要存储器容量等于每个芯片的位数就可以采用这种结构。当要求更大容量的存储器时，则需要芯片的阵列。图5-6表示1M字的存储器一种每字8位的结构。在此种情况时，有4列芯片，每一列都包含如图5-5所示的256K字。对于1M字，需要20根地址线，其中低18位连接到所有的32个模块。高2位输入到组选择逻辑模块，由它向4列模块的某一列发送芯片允许信号。

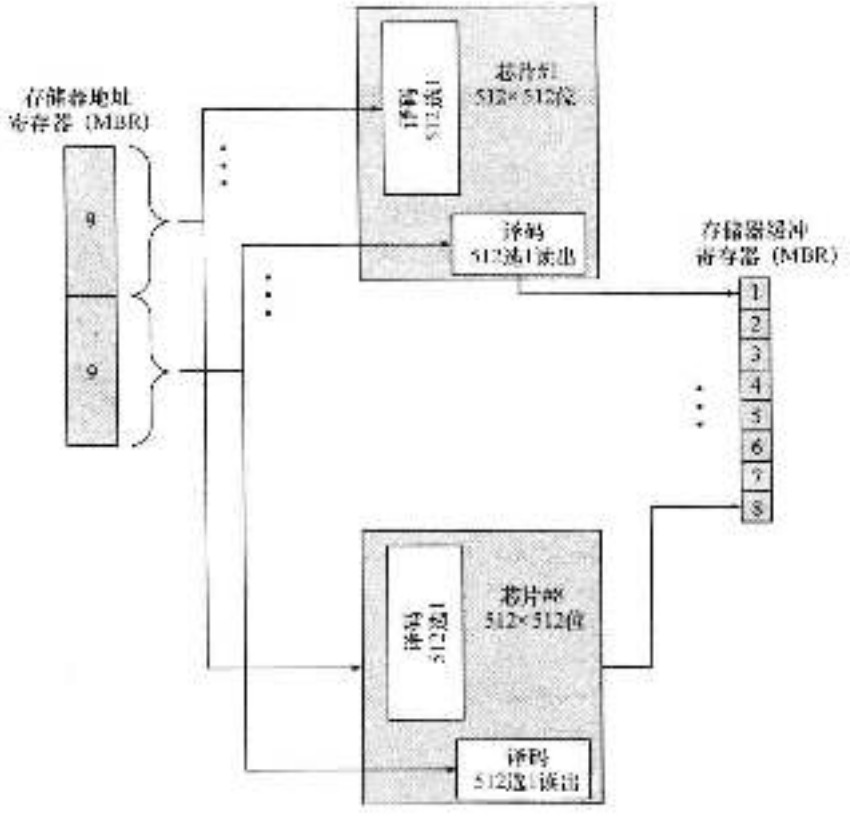


图 5.5 256KB 存储器组织

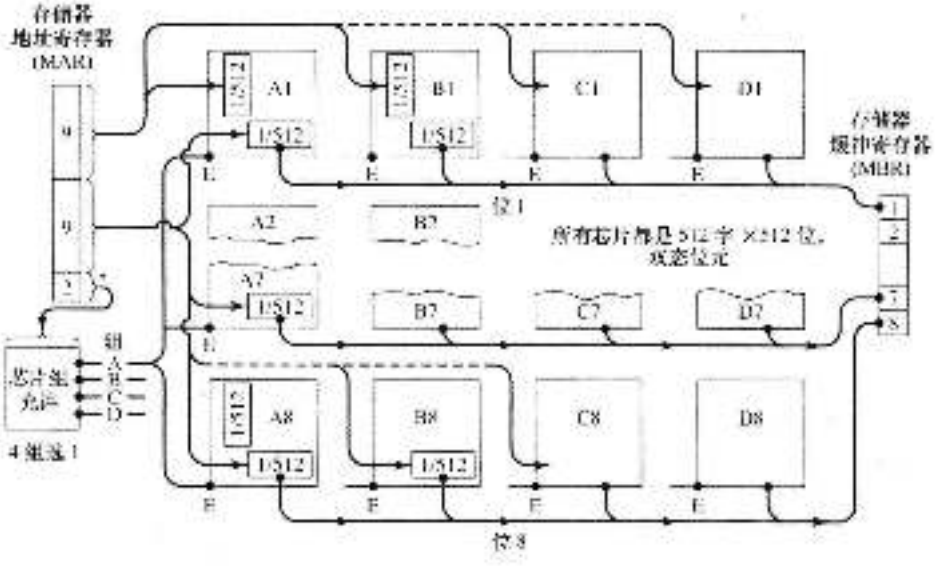


图 5-6 1MB 存储器组织

5.1.7 多体交叉存储器

主存储器由多块 DRAM 芯片组成，大量的芯片能组合形成存储体（memory bank），可以将多个存储体组织成多体交叉存储器。每一个存



存体可以独立提供存储器的读或写服务，因此，一个包含 K 个存储体的系统能够同时满足 K 个存取需求，使存储器的读/写速度增加到原来的 K 倍。如果存储器的一系列连续字存放在不同的存储体中，则传输一个内存块的速度会明显加快。附录 E 详细探讨了多体交叉存储器。

5.2 纠错

半导体存储系统会出现差错，差错可以分为硬故障和软差错两类。**硬故障**（hard failure）是永久性的物理故障，以至于受影响的存储单元不能可靠地存储数据，成为固定的“1”或“0”故障，或者在0和1之间不稳定地跳变。硬故障可由恶劣的环境、制造缺陷和旧损引起。**软差错**（soft error）是随机非破坏性事件，它改变了某个或某些存储单元的内容，但没有损坏存储器。软差错可以由电源问题或 α 粒子引起，这些粒子起因于放射性衰减，它们非常普遍，因为几乎所有材料中都有少量的放射性物质。硬故障和软差错显然都不受欢迎，因此，大多数的现代主存储器系统都包含了查错和纠错的逻辑。

图5-7给出了一般情况下的处理过程。当数据送入存储器时，对数据进行某种计算（用函数 f 表示）以产生一个校验码，校验码和数据同时存储。因此，如果存储的数据字长是 M 位，而校验码长是 K 位，则实际存储的字长是 $M+K$ 位。

当原先存储的字被读出时，这个校验码用于查错，甚至可能纠错。从 M 位数据中产生一组新的 K 位代码，并与取出的校验码位进行比较，比较后产生以下3种结果之一：

- 没有检测到差错，取出的数据位传送出去。
- 检测到差错，并且可以纠正。数据位和纠错位一起送入纠正器，然后产生一组正确的 M 位数据发送出去。
- 检测到差错，但无法纠正，报告这种情况。

用这种方式操作的代码称为纠错码（error correcting code）。纠错码以在字中能检测并纠正的出错位数来表征。

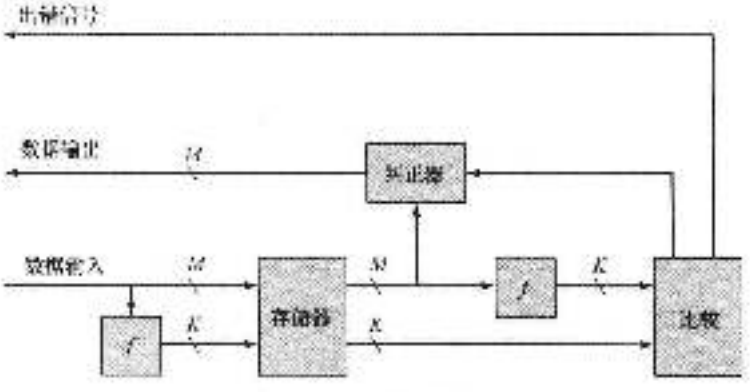


图5-7 纠错码功能

最简单的纠错码是汉明码（Hamming code），它由贝尔实验室的理查德·汉明发明。图5-8采用维恩图来说明汉明码在4位字（$M=4$）上的使用。由3个相交的圆，分割成7部分，将数据的4位分配给内部的4部分（如图5-8a所示），其余的部分填入奇偶校验位（parity bit）。选择适当的奇偶校验位，使得每个圆圈中1的总数是偶数（如图5-8b所示）。由于圆A包括了3个“1”，因此，这个圆中奇偶位被设置成“1”。现在，如果数据中有一位出错（如图5-8c所示），则很容易发现。通过检查奇偶校验位，发现圆A和圆C中不符合上述规则，所以确定出错位在圆A和圆C中，而不在圆B中（如图5-8d所示）。7部分只有1部分是在圆A和圆C中而不在圆B中，因此，改变此位就纠正了错误。

为了说明所涉及的概念，我们将设计一种检测和纠正 8 位字中单个位出错的代码。

首先，确定码长，参照图 5-7，比较逻辑接受两个 K 位值作为输入，通过两个输入的“异或”运算来进行逐位比较，结果被称为故障字（syndrome word）。根据两个输入的位是否匹配，确定故障字的每位是 0 或 1。

因此，故障字有 K 位宽，其值范围为 0 到 2^K-1。0 值表示没有检测到差错；而剩余 2^K-1 个值则指明当只有一位出错时，出错位是第几位。现在，由于差错可能发生在 M 个数据位或 K 个校验位中的任意一个，因此必须有：

$$2^K-1 \geq M+K$$

这个公式给出了纠正 M 位的数据字中单个位出错所需的位数。例如，对于一个 8 位（$M=8$）的数据字，我们有

- $K=3: 2^3-1<8+3$
- $K=4: 2^4-1>8+4$

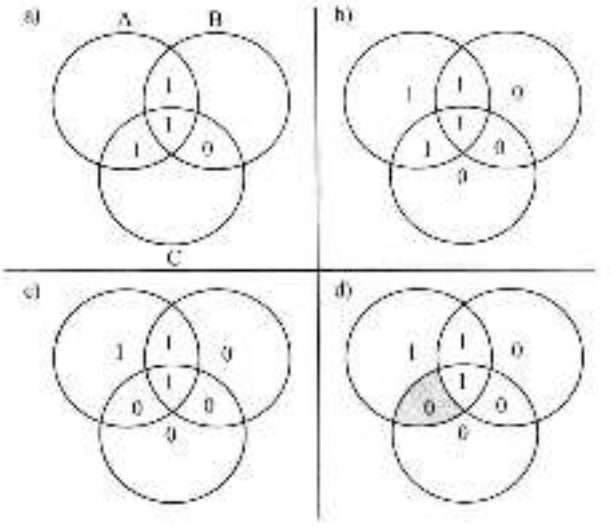


图 5-8 汉明纠错码

因此，8 个数据位要求 4 个校验位。表 5-2 的前三列给出了各种数据字长所需的校验位数。

表 5-2 带纠错码的字长增加情况

	单纠错		单纠错/双检错	
数据位	校验位	增加的百分率（%）	校验位	增加的百分率（%）
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

为方便起见，我们希望 8 位数据字产生的 4 位故障字具有如下特征：

- 如果故障字全部是 0，则表示没有检测到差错。
- 如果故障字有且仅有 1 位为 1，则错误发生在 4 个校验位中的一位，不需要纠正。
- 如果故障字有多位为 1，则故障位的数据值指明出错的数据位的位置。将这个数据位取反即可纠正。

为获得这些特征，数据位和校验位排列成 12 位的字，如图 5-9 所示。各位的位置编号从 1～12，位置号为 2 的幂次方的位置被定为校验位。校验位可用异或操作（符号为⊕）计算如下：

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$

$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$$

$$C4 = D2 \oplus D3 \oplus D4 \oplus D8$$

$$C8 = D5 \oplus D6 \oplus D7 \oplus D8$$

位的位置	12	11	10	9	8	7	6	5	4	3	2	1
位置编号	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
数据位	D8	D7	D6	D5		D4	D3	D2		D1		
校验位					C8				C4		C2	C1

图5-9 数据位和校验位的安排

每个校验位对那些在相应二进制序列位置编号为1的每个数据位位置进行操作。因此，位置3、5、7、9、11（D1、D2、D4、D5、D7）都在其位置编号的最低位包含一个1，用来计算C1；而位置3、6、7、10、11都在其次低位包含一个1，用来计算C2；如此类推。用另一种方式来看，如果位的位置n由几个C_i位检查，则$\sum_i = n$。例如，位置7由处于4、2、1位置的位检查，且7=4+2+1。

让我们用一个例子来验证这个方案。假设一个8位的输入字为00111001，数据位D1在最右边，计算如下：

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

现在，假设数据位3遇到一个错误，它由0变成为1。重新计算校验位，有：

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

当新的校验位与老的校验位进行比较时，形成故障字：

	C8	C4	C2	C1
	0	1	1	1
⊕	0	0	0	1
	0	1	1	0

结果是0110，表示出错位是位置6，即第3个数据位。

图5-10说明了上述计算过程。8数据位和4校验位被给当地安排成12位字，其中只有4个数据位的值是1（图中用阴影表示）。将00111001数据字按上述规则进行异或运算，产生的汉明码是0111，它形成4个检测数据。存储的整个12位字是001101001111。现在，假设数据的第3位，也就是整个12位字的第6位出错，它由0变为了1，则取出的12位字是001101101111，其中的汉明码是0111。对其中的数据位按上述规则进行异或产生的新汉明码是0001。将新汉明码与取出的汉明码按位异或产生故障字0110，该非零结果检测出1位错并指出第6位出错。

刚才描述的代码被称为单纠错（single-error-correcting，SEC）码。而半导体存储器更常用的是单纠错双检错（single-error-correcting，double-error-detecting，SEC-DED）码。如表5-2所示，SEC-DED码与SEC码相比，多一个附加位。

图5-11说明了这种代码如何对4位数据字进行工作。序列显示，如果有两个错误发生（如图5-11e所示），则检查过程误入歧途（如图5-11d所示），并产生第3个差错（如图5-11e所示），使问题变得糟糕。为了避免这种情况，增加了一个第8位，使图中"1"的总数为偶数。于是，这个附加的奇偶校验位能捕捉到差错（如图5-11f所示）。

位的位置	12	11	10	9	8	7	6	5	4	3	2	1
位置编号	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
数据位	D8	D7	D6	D5		D4	D3	D2		D1		
校验位					C8				C4		C2	C1
存入字	0	0	1	1	0	1	0	0	1	1	1	1
取出字	0	0	1	1	0	1	1	0	1	1	1	1
位置编号	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
校验位					0				0		0	1

图 5-10 校验位计算

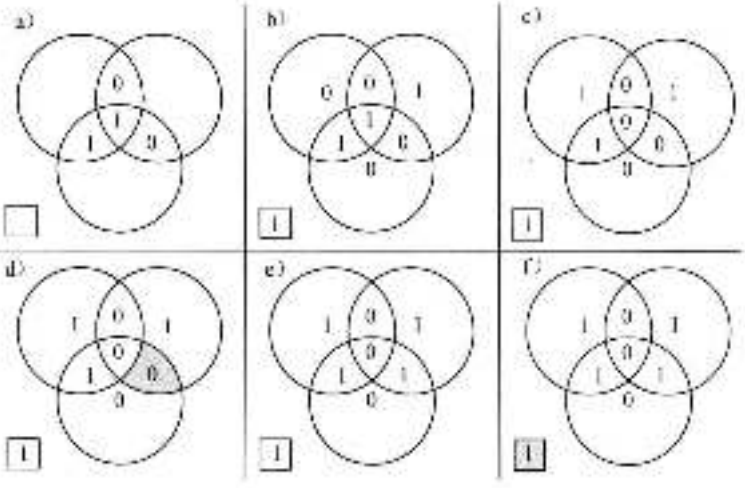


图 5-11 汉明 SEC-DED 码

纠错码以增加复杂性为代价来提高存储器的可靠性。对于每片一位的组织，通常采用 SEC-DED 码。例如，在 IBM 30xx 的实现中，主存中每 64 位数据采用了 8 位的 SEC-DED 码。因此，主存的实际容量比用户见到的容量要大 12%。VAX 计算机的存储器中每 32 位采用 7 位 SEC-DED 码，从而有 22% 的额外开销。一些当代的 DRAM 为每 128 位的数据使用了 9 个校验位，从而有 7% 的额外开销 [SHAR97]。

5.3 高级 DRAM 组织

正如第 2 章所述，在使用高性能处理器时，最严重的系统瓶颈之一是处理器与内部主存储器的接口，该接口是整个计算机系统最重要的路径。从几十年前到现在，主存储器的基本构件仍然是 DRAM 芯片。从 20 世纪 70 年代早期起，DRAM 结构一直没有发生任何显著的变化。传统的 DRAM 芯片受限于其内部结构及其与处理器的存储总线的连接。

我们看到，解决 DRAM 主存储器性能问题的一种方法是，在 DRAM 主存储器和处理器之间插入一级或多级高速 SRAM cache。但是 SRAM 比 DRAM 贵得多，扩展 cache 容量超过一定限度时，将得不偿失。

在过去的几年中，人们开发了许多对基本 DRAM 结构的增强功能，市场上也出现了一些产品。现在占据市场的几种方案是 SDRAM、DDR-DRAM 和 RDRAM，表 5-3 提供了这几种方案的性能比较。另一个值得重视的方案是 CDRAM。本节将一一分析这些方案。

表 5-3 几种 DRAM 类型的性能比较

	时钟频率（MHz）	传输率（GB/s）	存取时间（ns）	引脚数
SDRAM	166	1.3	18	168
DDR	200	3.2	12.5	184
RDRAM	600	4.8	12	162

5.3.1 同步 DRAM

DRAM 最广泛使用的一种形式是同步 DRAM（synchronous DRAM，SDRAM）（参见[VOGL94]）。SDRAM 与传统的 DRAM（异步的）不同，它与处理器的数据交换同步于外部的时钟信号，并且以处理器/存储器总线的最高速度运行，而不需要插入等待状态。

在典型的 DRAM 中，处理器将地址和控制信号提供给存储器，表示存储器中特定单元的一组数据应该被读出或写入 DRAM。经过一段延时（即存取时间），DRAM 写入或者读出数据。在存取时间延迟中，DRAM 执行各种内部功能，如激活行地址线或列地址线的高电容，读取数据，以及通过输出缓冲将数据输出。在这段时延中，处理器只能处于等待状态，这降低了系统的性能。

有了同步存取机制，DRAM 就能在系统时钟的控制下输入输出数据。处理器或其他主控器发出指令和地址信息，它们被 DRAM 锁存。然后，DRAM 在几个时钟周期后响应。与此同时，在 SDRAM 处理请求时，主控器能安全地做其他事情。

图 5-12 显示了 IBM 64 兆位的 SDRAM 的内部逻辑[IBM01]，这是一个典型的 SDRAM 组织结构。表 5-4 给出了芯片引脚分配。SDRAM 采用突发方式来消除地址建立时间和第一次存取之

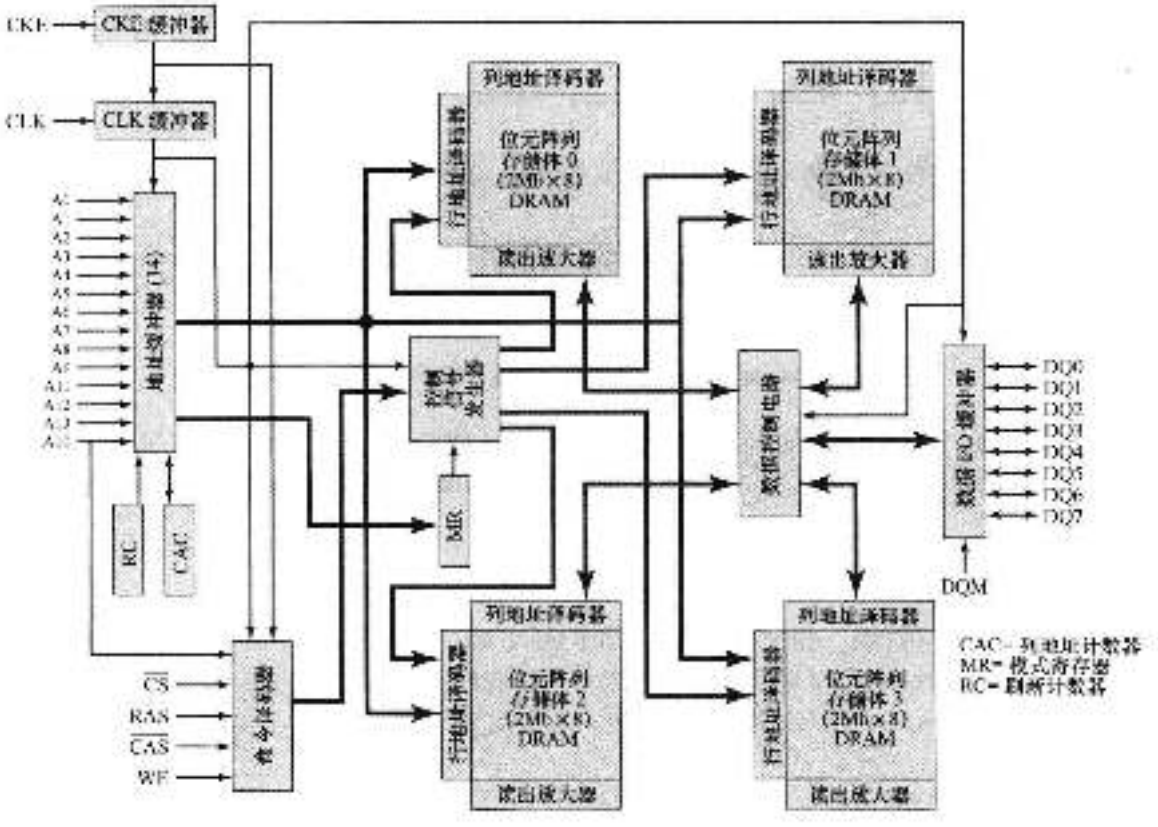


图 5-12 同步动态 RAM（SDRAM）

后行线和列线的预充电时间。在爆发方式中，访问第1位以后，一系列的数据位能够快速地随着时钟输出。当要访问的所有位是顺序的，并且与访问的第1位处于阵列中的同一行时，这种方式非常有用。此外，SDRAM的多存储体内部结构改进了片内的并行性。

表5-4 SDRAM芯片引脚分配

A0到A13	地址输入	A0到A13	地址输入
CLK	时钟输入	$\overline{CAS}$	列地址选通
CKE	时钟允许	$\overline{WE}$	写允许
$\overline{CS}$	芯片选择	DQ0到DQ7	数据输入/输出
RAS	行地址选通	DQM	数据屏蔽

模式寄存器和相关的控制逻辑是SDRAM不同于传统的DRAM的另一个关键特点。它提供了定制的SDRAM以满足特定系统需求的机制。模式寄存器指定了爆发存取长度，该长度是同步地向总线发送数据的单元个数。该寄存器也允许程序员调整从接受读请求到开始数据传输的延迟时间。

SDRAM在连续传输大数据块时性能最佳，例如字处理、电子表格和多媒体等应用。

图5-13给出SDRAM操作的例子。此情况下爆发存取长度是4，延迟是2。在时钟上升沿，通过使$\overline{CS}$和$\overline{CAS}$为低电平的同时保持RAS和WE为高电平来启动爆发读命令。地址输入确定爆发存取的起始行地址，模式寄存器设置爆发的类型（顺序的或交错的）和爆发存取长度（1，2，4，8，全页）。从命令开始到第1个位元的数据出现在输出线上的延时等于模式寄存器中设置的CAS延迟值。

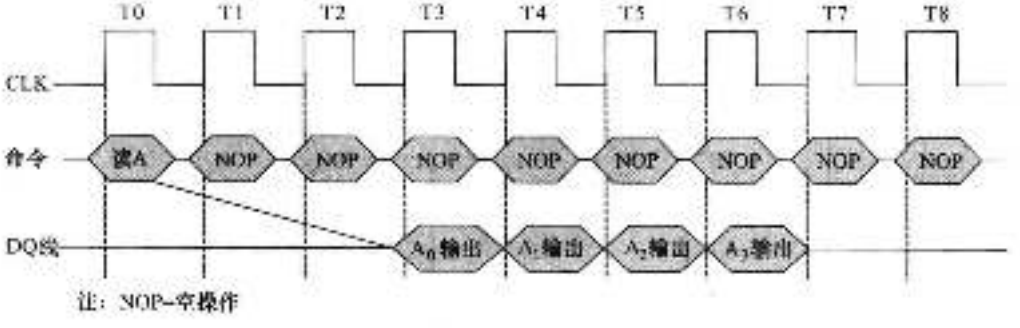


图5-13 SDRAM读操作时序（爆发存取长度=4，CAS延时=2）

现在又出现了一种SDRAM的增强型版本，称为双倍数据速率的SDRAM（double data rate SDRAM，DDR-SDRAM），它突破了每周期一次的限制。DDR SDRAM能每周期向处理器传送两次数据。

5.3.2 Rambus DRAM

由Rambus［FARM92，CRIS97］开发的RDRAM为Intel的Pentium和Itanium处理器所采用，它已成为SDRAM的最大竞争对手。RDRAM芯片是垂直封装的，所有的引脚都在一侧。芯片通过28根不超过12cm长的线与处理器交换数据。总线最多能寻址320块RDRAM芯片，传输率可达1.6GB/s。

这种特殊的RDRAM总线使用异步的面向块的协议来传送地址信息和控制信息。经过480ns初始访问时间后，就产生1.6GB/s的数据速率。使这一速率能够实现的是总线本身，它非常精确地定义了驱动、时序和信号。不像传统的DRAM采用显式的RAS、CAS、R/W和CE信号来控

制，RDRAM 通过高速总线获得存储器请求，这一请求包括了操作时所需的地址、操作类型和字节数。

图 5-14 说明了 RDRAM 的配置情况，该配置包含一个控制器和几个经由公共总线连接在一起的 RDRAM 模块。配置的一端是控制器，总线的远端是这些总线的一个并行终接器。总线包括 18 根数据线（16 根是实际数据，2 根是奇偶校验），周期率是时钟速率的两倍，即时钟信号的前、后沿各送出 1 位，这导致了每个数据线上的信号速率是 800Mb/s。另有一组分离的 8 根线（RC）用于地址和控制信号。还有一个时钟信号，它由控制器的远端出发，传播到控制器之后再返回到远端，形成一个环路。RDRAM 模块与主控器时钟同步传送数据到控制器；而控制器以此时钟的反方向同步传送数据到一个 RDRAM 模块。总线的其余线包括参考电压、地和电源。

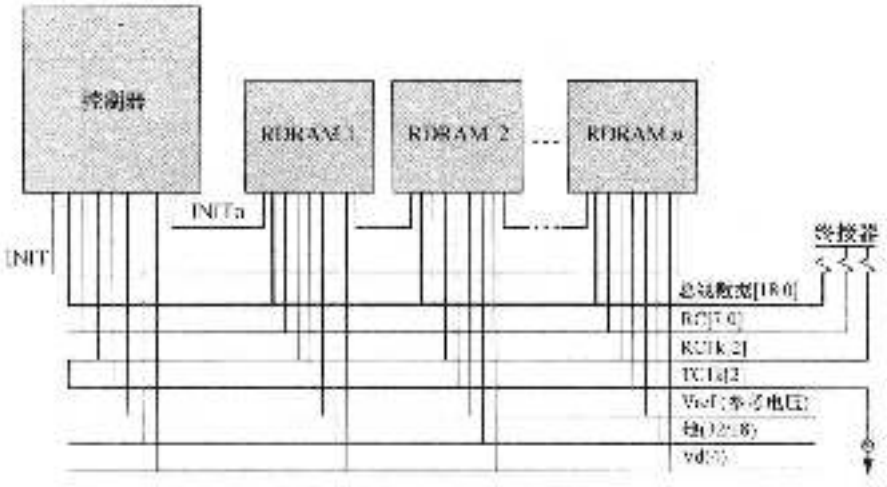


图 5-14 RDRAM 结构

5.3.3 DDR DRAM

SDRAM 受限于它每个总线时钟周期仅能发送一次数据到处理器。一种 SDRAM 的新版本，称为双速率 SDRAM，能每时钟周期发送两次数据，一次在时钟脉冲的上升沿，一次在下降沿。

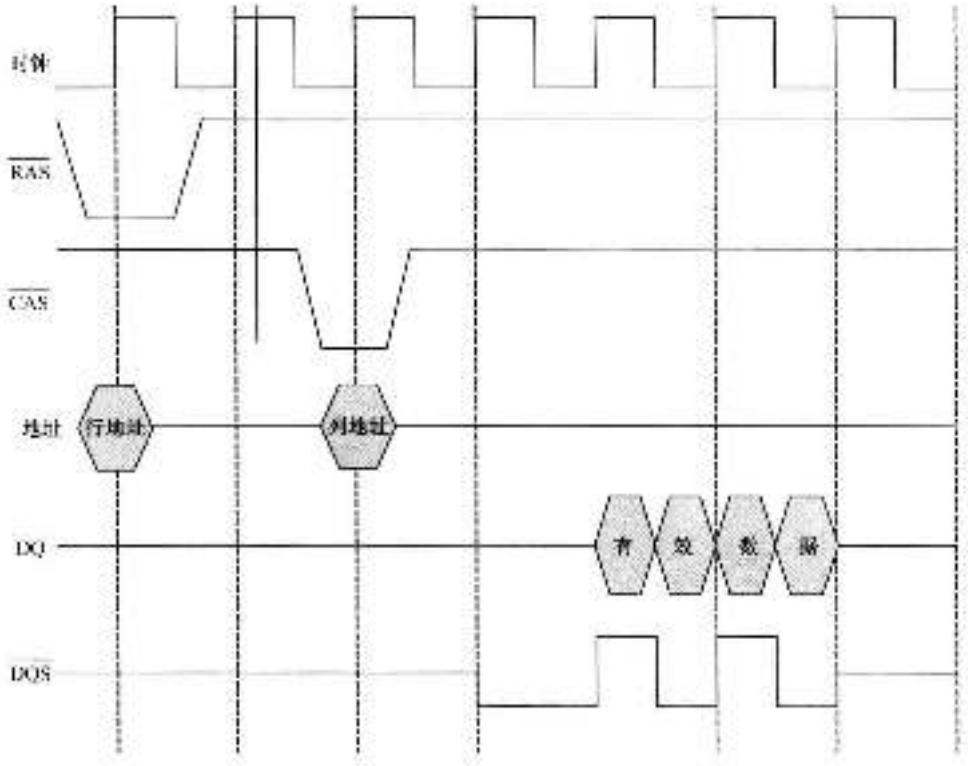
DDR DRAM 是由 JEDEC（电子设备工程联合委员会）固态技术协会开发的，JEDEC 固态技术协会是电子工业联盟的半导体工程标准协会。许多公司都在制造 DDR 芯片，它们被广泛地用于桌面计算机和服务器。

图 5-15 显示了 DDR 的一个基本读时序。数据传输与时钟的上升沿和下降沿同步，它还和双向数据选通信号（DQS）同步。双向数据选通信号在读操作期间由存储控制器提供，而在写操作期间由 DRAM 提供。在典型的实现中，DQS 在读取数据时被忽略。关于 DQS 在写入数据时的使用说明超出了我们所要讨论的范围，想详细了解可参见 [JAC008]。

DDR 技术已经经历了两代改进。DDR2 通过提高 RAM 芯片的操作频率和将每片芯片的预取缓冲器由 2 位增加到 4 位，从而提高了数据传输率。预取缓冲器是一个置于 RAM 芯片上的存储器 cache。该缓冲器使 RAM 芯片能够尽可能快地将前置位放入数据中。DDR3 于 2007 年推出，它将预取缓冲器容量增大到 8 位。

理论上，DDR 模块能以 200 ~ 600MHz 的时钟速率传送数据，DDR2 可以在 400 ~ 1066MHz 的时钟速率内传输数据，而 DDR3 的传输速率在 800 ~ 1600MHz 之间。当然，实际能达到的速率都要低一些。

附录 K 提供了关于 DDR 技术的更加详细的讨论。



RAS = 行地址选择
CAS = 列地址选择
DQ = 数据（输入或输出）
DQS = DQ 选择

图 5-15 DDR SDRAM 读操作时序

5.3.4 cache DRAM

由三菱公司开发的 cache DRAM（CDRAM）[HIDA90，ZHAN01]，在常规的 DRAM 芯片内集成了一个小的 SRAM cache（16Kb）。

CDRAM 上的 SRAM 能以两种方式使用。首先，它能作为真正的 cache 使用，每行由 64 位组成。CDRAM 的 cache 模式对普通的存储器随机存取非常有效。

其次，CDRAM 上的 SRAM 也可用做支持串行存取数据块的缓冲器。例如，为刷新位映射屏幕，CDRAM 能从 DRAM 预取数据到 SRAM 缓冲器中，随后对芯片的存取只对 SRAM 进行。

5.4 推荐的读物和 Web 站点

[PRIN91] 提供了对包括 SRAM、DRAM 和快闪存储器在内的半导体存储器技术的综合论述。[SHAR97] 介绍了同样的内容，并强调了测试和可靠性问题。[SHAR03] 和 [PRIN02] 专注于高级的 DRAM 和 SRAM 体系结构。深入考察 DRAM，见 [JACO08] 和 [KETT01]。[CUPP01] 提供了各种 DRAM 方案的性能比较，很有意义。[BEZ03] 全面介绍了快闪存储器技术。

[MCEL85] 中有纠错码的很好阐述。若需要更深入的研究，[ADAM91] 和 [BLAH83] 两书值得阅读。[ASH90] 对纠错码进行了理论和数学的分析，值得一读。[SHAR97] 包含有对当代主存储器中所用代码的全面综述。

ADAM91 Adamek, J. *Foundations of Coding*. New York: Wiley, 1991.

ASH90 Ash, R. *Information Theory*. New York: Dover, 1990.

BEZ03 Bez, R.; et al. Introduction to Flash Memory. *Proceedings of the IEEE*, April 2003.

BLAH83 Blahut, R. *Theory and Practice of Error Control Codes*. Reading, MA: Addison-Wesley, 1983.

CUPP01 Cuppu, V., et al. "High Performance DRAMS in Workstation Environments." *IEEE Transactions on Computers*, November 2001.

JACO08 Jacob, B.; Ng, S.; and Wang, D. *Memory Systems: cache, DRAM, Disk*. Boston: Morgan Kaufmann, 2008.

KEET01 Keeth, B., and Baker, R. *DRAM Circuit Design: A Tutorial*. Piscataway, NJ: IEEE Press, 2001.

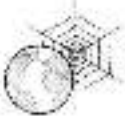
MCEL85 McEliece, R. "The Reliability of Computer Memories." *Scientific American*, January 1985.

PRIN97 Prince, B. *Semiconductor Memories*. New York: Wiley, 1997.

PRIN02 Prince, B. *Emerging Memories: Technologies and Trends*. Norwell, MA: Kluwer, 2002.

SHAR97 Sharma, A. *Semiconductor Memories: Technology, Testing, and Reliability*. New York: IEEE Press, 1997.

SHAR03 Sharma, A. *Advanced Semiconductor Memories: Architectures, Designs, and Applications*. New York: IEEE Press, 2003.



推荐的 Web 站点

- The RAM Guide：对 RAM 技术进行了全面论述，并提供了几个有用的链接。
- RDRAM：提供 RDRAM 信息的另一个有用的来点。

5.5 关键词、思考题和习题

关键词

cache DRAM（CDRAM）：带高速缓存的 DRAM
dynamic RAM（DRAM）：动态 RAM
electrically erasable programmable ROM（EEPROM）：电可擦除/可编程 ROM
erasable programmable ROM（EPROM）：可擦除/可编程 ROM
error-correcting code（ECC）：纠错码
error correction：错误纠正
flash memory：快闪存储器
Hamming code：汉明码
hard failure：硬故障
nonvolatile memory：非易失性存储器
programmable ROM（PROM）：可编程 ROM
RamBus DRAM（RDRAM）：总线式 DRAM
read-mostly memory：主要进行读操作的存储器，主读存储器
read-only memory（ROM）：只读存储器
semiconductor memory：半导体存储器
single-error-correcting（SEC）code：单纠错码
single-error-correcting, double-error-detecting（SEC-DED）code：单纠错双检错码
soft error：软差错
static RAM（SRAM）：静态 RAM
synchronous DRAM（SDRAM）：同步 DRAM
syndrome：故障，综合症
volatile memory：易失性存储器

思考题

5.1 半导体存储器的主要特点是什么？
5.2 术语"随机存取存储器"在使用上有哪两种含义？
5.3 DRAM 和 SRAM 在应用上有何不同？
5.4 在速度、容量、成本等特性方面，DRAM 和 SRAM 有何区别？
5.5 说明为什么一种类型的 RAM 被认为是模拟设备，而另一种类型的 RAM 被认为是数字设备。
5.6 试给出 ROM 的某些应用。
5.7 EPROM、EEPROM 和快闪存储器三者之间有何不同？
5.8 解释图 5-4b 中每个引脚的功能。
5.9 什么是奇偶校验位？
5.10 如何解释汉明码的术语字？

5.11 SDRAM 如何不同于普通的 DRAM?

习题

5.1 传统的 RAM 被组织成每芯片只有一位，而 ROM 通常被组织成每芯片多位，请说明原因。

5.2 考虑动态 RAM 每芯片必须刷新 64 次，每次刷新操作需要 150ns，而一个存储周期需要 250ns。试问：必须将存储器总操作时间的百分之几用于刷新?

5.3 图 5-16 给出了一个 DRAM 经由总线进行读操作的简化时序。存取时间被认为是由 t_1 到 t_2，然后是再充电时间，从 t_2 延续到 t_3，在此期间 DRAM 芯片必须再充电，然后处理器才能再次访问它们。

(a) 假定存取时间是 60ns，再充电时间是 40ns。试问：存储周期时间是多少? 假定 1 位输出，这个 DRAM 所支持的最大数据传输率是多少?

(b) 使用这些芯片构成一个 32 位宽的存储系统，其产生的数据传输率是多少?

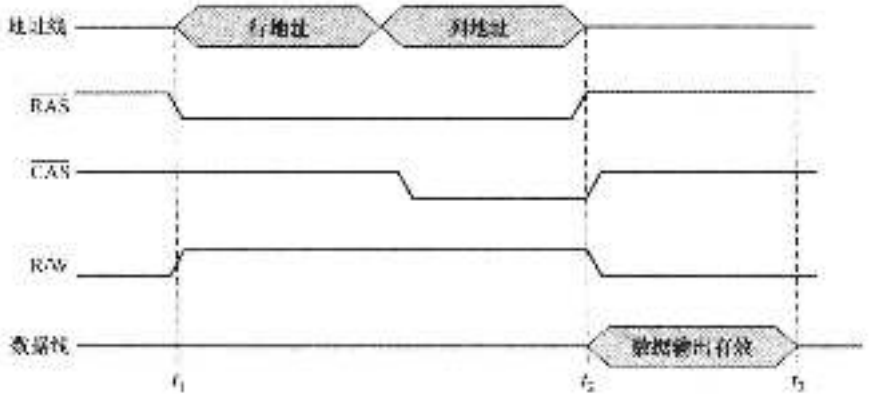


图 5-16 简化的 DRAM 读时序

5.4 图 5-6 指出如何利用 4 个 256KB 的芯片组成构成一个能存储 1MB 的芯片模块。假定该芯片模块已包装成为一个独立的 1MB 芯片，其字长是 1 个字节。请画出使用 8 个 1MB 的芯片来构成一个 8MB 存储器的连接图。请确定在你的图中两有地址线，并说明它们的用途?

5.5 在一个经系统总线连接 DRAM 存储器的典型 Intel 8086 系统上，由地址允许信号的下降沿启动$\overline{RAS}$有效，从而开始 DRAM 芯片的读操作（见图 3-19）。然而，由于线路传播延迟和其他延迟，使地址允许信号变低后再经历 50ns 才使$\overline{RAS}$有效。假定$\overline{RAS}$开始有效出现在 T_1 状态的后一半的中间（比图 3-19 中所示的快一点），在 T_3 的终端处理器读入数据。然而，为了使数据适时地交给处理器，存储器必须先于 60ns 提供数据。这个时间间隔考虑到沿数据路径（由存储器到处理器）的传输延迟和处理器数据保持时间的要求。假定时钟速率是 10MHz。

(a) 如果没有插入等待状态，则 DRAM 应多快（存取时间）?

(b) 如果 DRAM 的存取时间是 150ns，则何次存储器的读操作应插入多少个等待状态?

5.6 一个特定的微计算机的存储器由 64K × 1 的 DRAM 芯片构成。依据数据资料可知，DRAM 芯片的位元阵列组织成 256 行，每行每 4ms 必须至少刷新一次。假定系统严格按照此要求周期性地刷新该存储器。

(a) 连续刷新请求之间的时间周期是多少?

(b) 所需的刷新地址计数器是多少位?

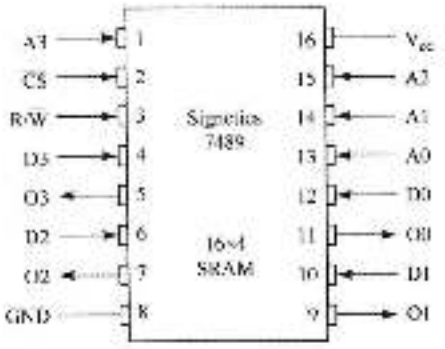
5.7 图 5-17 显示了一种早期的 SRAM，它是一个 16 × 4 的 Signetics 7489 芯片。

(a) 列出图 5-17c 中给出的每个$\overline{CS}$输入脉冲下的芯片操作模式。

(b) 列出在脉冲 a 后，字位置 0 到 6 的存储器内容。

(c) 对输入脉冲 b ~ m，数据输出线的状态是什么?

5.8 使用容量为 64 × 1 位的 SRAM 芯片设计一个总容量为 8192 字的 16 位存储器。要求给出芯片在存储器板上的阵列配置，确定为存储器分配最低地址空间所要求的输入和输出信号，并且该设计应既能满足以字节存取，又能满足以 16 位字存取。

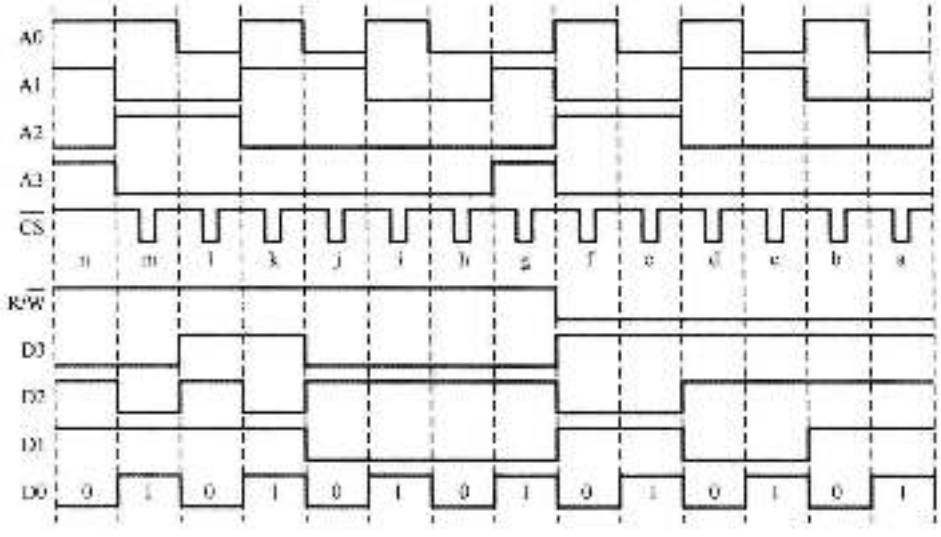


a) 引脚安排

操作模式	输入			输出
	$\overline{CS}$	R/$\overline{W}$	Dn	On
写	L	L	L	L
	L	L	H	H
读	L	H	X	数据
禁止写	H	L	L	H
	H	L	H	L
存储-禁止输出	H	H	X	H

H = 高电平
L = 低电平
X = 不关心

b) 真值表



c) 脉冲时序

图 5-17 Signetics 7489 SRAM

5.9 测量电子元件故障率的常用单位是非特（Failure unIT，FIT），表示每十亿（10^9）设备小时的故障率。另一个著名但不太常用的测量单位是平均故障间隔时间（mean time between failures，MTBF），它是一特定元件正常（无故障）运行的平均时间。考虑在一个 16 位微处理器中使用 256K × 1 的 DRAM 芯片所构成的一个 1MB 存储器，假定每个 DRAM 芯片是 2000FIT，请计算此存储器的 MTBF。

5.10 对于图 5-10 所示的汉明码，试说明当一个校验位而不是一个数据位出错时，会发生什么？

5.11 假设存放在存储器中的一个 8 位数据字是 11000010，请使用汉明算法确定需要哪些校验位与此数据字一起存放在存储器中，并说明解题步骤。

5.12 一个 8 位字 00111001，与它一起存储的校验位应该是 0111。假定从存储器读出时，计算出的校验位是 1101，那么由存储器读出的数据字是什么？

5.13 若使用汉明纠错码来确定 1024 位数据字中的单个错误，则需要多少校验位？

5.14 试设计一个 16 位数据字的 SEC 码，假定数据字为 0101000000111001，说明 SEC 码如何正确识别数据位 5 的错误。

第6章

Computer Organization and Architecture: Designing for Performance, 8E

外部存储器

本章要点

- 磁盘至今仍是最重要的外部存储器。从个人计算机到大型机，几乎每台计算机，都广泛使用活动式磁盘和固定式磁盘（硬盘）。
- 为了实现更高的性能和更好的可用性，服务器以及更大系统使用 RAID 磁盘技术。RAID 使用了多个磁盘作为数据存储设备的并行阵列的一系列技术，并具有内在冗余性来弥补磁盘故障。
- 在各种类型的计算机系统中，光存储技术变得日益重要。CD-ROM 技术已广泛使用多年，而像可写式 CD 和 DVD 这类更加新近的技术也正在兴起。

本章考察外部存储器的设备和系统。首先从最重要的磁盘设备开始，磁盘几乎是所有计算机系统外部存储器的基础；然后介绍使用磁盘阵列以获得更好的性能，特别考察被称为 RAID（redundant array of independent disks）的磁盘冗余阵列技术；接着介绍对许多计算机系统来说日益重要的部件——外部光存储器；最后讨论磁带存储器。

6.1 磁盘

磁盘是一种由非磁性材料制成称为衬底的圆盘，其上涂有一层磁性材料。传统上，衬底一直使用铝或铝合金材料，而最近，已推出玻璃衬底。玻璃衬底具有很多优点，主要包含如下几点：

- 改善了磁层表面的均匀性，从而增强了磁盘的可靠性。
- 显著地减少了整个表面的缺陷，从而有助于读-写错误的减少。
- 能支持更低的磁头飞行高度（后面将介绍）。
- 更好的刚度，从而降低了磁盘动力需求。
- 更好的耐冲击和耐损伤能力。

6.1.1 磁读写机制

数据的记录和其后的读出都是通过一个叫做磁头（head）的导电线圈进行的。多数系统使用两个磁头：一个读磁头，一个写磁头。在读或写操作期间，磁头静止不动，而盘片在非常靠近磁头的下方高速旋转。

写机制使用了电流通过线圈时产生磁场这个效应。脉冲电流送入写磁头，形成的磁化模式被记录在其下的磁盘表面上，正、负电流分别产生不同的磁化模式。写磁头本身是一个由易磁化的材料所组成的矩形环，其一侧开有缝隙，而相对的一侧绕有数圈导线（如图 6-1 所示）。线圈中的电流在缝隙间感应出一个磁场，此磁场在记录介质上磁化出一个小域；改变电流的方向，磁域的磁化方向也随之改变。

传统的读机制利用了磁盘相对线圈运动时在线圈中产生电流这个效应。当磁盘表面在磁头下通过时，产生一个与数据记录电流极性相同的电流。这种方式的读磁头结构本质上与写磁头结构相同，因此，同一磁头既可用于读也可用于写。这种单磁头结构主要用于软盘系统和老式硬盘系统。

当代硬盘系统采用一种不同的读机制，它要求使用一个单独的读磁头，通常紧靠写磁头安装。读磁头由一个部分被屏蔽的磁阻（magnetoresistive，MR）传感器组成。MR 材料的电阻大小

取决于在它下面运动的介质的磁化方向。让电流通过 MR 传感器，电阻的变化作为电压信号被检测出来。MR 设计允许更高频率的操作，这等同于更高的存储密度和更快的操作速度。

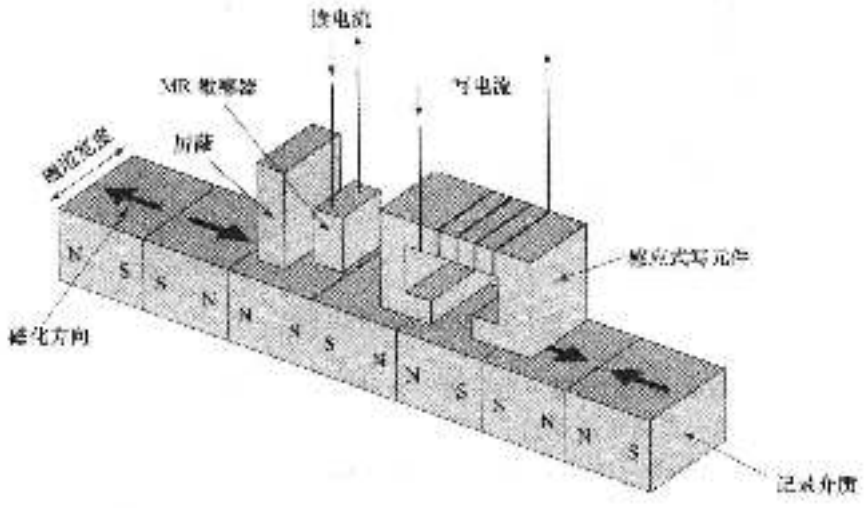


图 6-1 感应式写/磁阻式读的磁头

6.1.2 数据组织和格式化

磁头是一种相对较小的装置，它能从处于其下方的、正在旋转的盘片上读取数据或向盘片写入数据。由此，盘上的数据组织呈现为一组同心圆环，圆环被称为磁道（track）。每个磁道与磁头同宽。每个盘面上有数千个磁道。

图 6-2 描述了磁盘的数据分布。相邻磁道之间有间隙（gap），它可以防止或至少可以减少由于磁头未对准或磁域干扰所引起的错误。

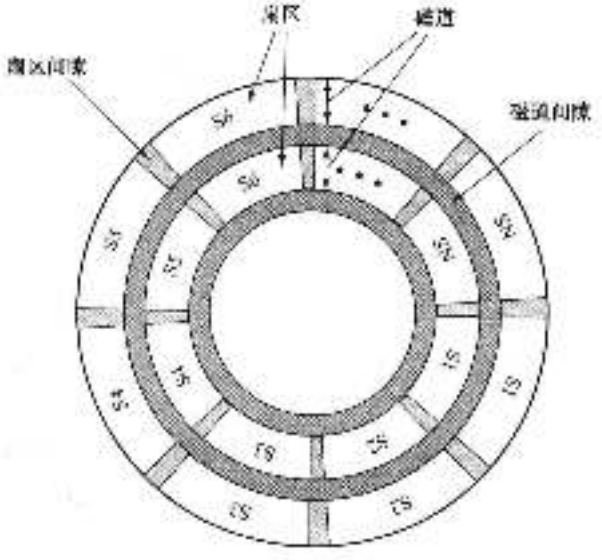


图 6-2 磁盘数据分布

数据以扇区（sector）为单位传入或传出磁盘（如图 6-2 所示）。每个磁道通常有数百个扇区，其长度可固定也可变化。当前，大多数系统使用固定长度的扇区，512 字节几乎是通用的扇区大小。为避免对系统提出不合理的定位精度要求，相邻扇区也留有间隙。

靠近旋转盘中心的位经过一固定点（如读-写磁头）的传输要比盘外沿的位慢。因此，必须寻找一种方式来补偿速率的变动，使磁头能以同样的速度读取所有的位。这可以通过增大记录在盘片区域上的信息位之间的间隔来实现。于是，以固定速度旋转的磁盘能够以相同的速率来扫描所有的信息，该速度称为**恒定角速度**（constant angular velocity，CAV）。图 6-3a 显示了使用 CAV 的磁盘布局，盘面被划分成一些同心圆磁道和多个饼形扇区。使用 CAV 的好处是，能以磁道号和扇区号来直接寻址各个数据块。为了将磁头从当前位置移动到指定位置，只需将磁头径向移动到指定磁道，然后等待指定扇区转到磁头下，整个寻时很少。CAV 的缺点是，外圈的长磁道上存储数据需与内圈的短磁

道上所存储数据一样多。

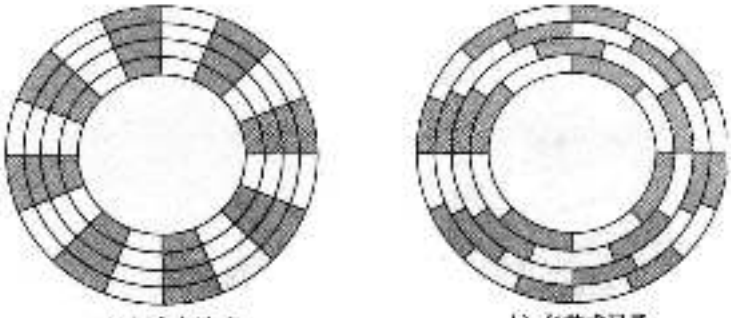


图 6-3 盘面布局方法比较

因为线性密度（density），即每英寸的位数，由外圈磁道到内圈磁道是逐渐增大的，所以在古板了当的 CAV 系统中，存储容量受到了内圈所能实现的最大记录密度的限制。为了提高密度，当今的硬盘系统使用了一种称之为多重区域记录（multiple zone recording）的技术，它将盘面划分成多个区域（典型的是 16 区）。在一个区域中，各磁道的位数是恒定的，远离中心的区域要比靠近中心的区域容纳更多的位（更多的扇区），这就允许以稍微复杂一些的电路为代价全面提升磁盘存储容量。当磁头由一个区域移动到另一个区域时（沿磁道），位长度的改变引起读写时序做相应变动。图 6-3b 给出了多重区域记录的磁盘布局，在此，每个区域只是一个简单的磁道宽。

需要某种方式来确定一个磁道内的扇区位置。显然，磁道必须有一些起始点和辨识每个扇区的起点及终点的方法。这些需求由记录在磁盘上的控制数据来处理。因此，磁盘格式化时，会附有一些仅被磁盘驱动器使用而不被用户存取的额外数据。

磁盘格式化的一个例子如图 6-4 所示。在此例中，每个磁道包含 30 个固定长度的扇区，每个扇区有 600 字节，其中包含 512 字节的数据和磁盘控制器使用的控制信息。ID 域是唯一的一个标识或地址，用于定位具体的扇区。同步字节是一个特殊的位模式，用来定义区域的起始点。磁道号标识磁盘表面上的一个磁道。磁头号标识一个磁头，因为磁盘有多个面（不久将解释）。ID 和数据域各包含一个检错码。

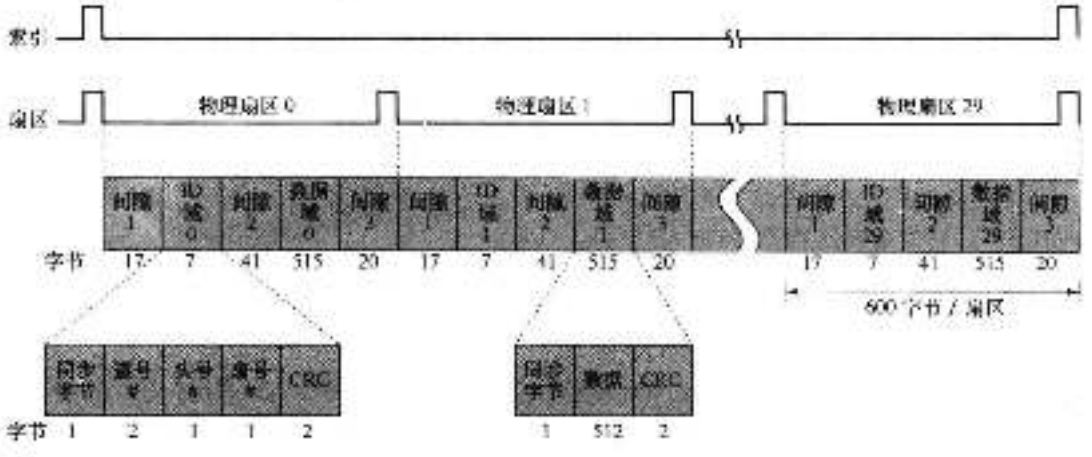


图 6-4 温彻斯特磁盘磁道格式（Seagate ST506）

6.1.3 物理特性

表6-1列出了区分各类磁盘的主要特性。首先，磁头在磁盘的径向上既可以是固定的也可以是移动的。在**固定头磁盘**（fixed-head disk）中，每个磁道有一个读-写头，所有磁头安装在跨越所有磁道的固定支架上，这种系统目前已很少。而在**可移动头磁盘**（movable-head disk）中，只有一个读-写头，与前者相似，磁头固定在支架上，但支架能伸缩，以使磁头能定位到任何磁道上。

表6-1 磁盘系统的物理特性

磁头互动	磁盘可更换性	面	盘片	磁头机制
固定磁头（每磁道一个）	不可更换磁盘	单面	单盘片	接触（软盘）
可移动磁头（每面一个）	可更换磁盘	双面	多盘片	固定间隙
				空气动压气膜（温氏磁盘）

磁盘本身安装在磁盘驱动器内，而驱动器由支架、带动盘片旋转的主轴和二进制数据输入/输出所需要的电路组成。**不可更换磁盘**（nonremovable disk）永久安装在磁盘驱动器内。个人计算机上的硬盘就是这种类型；而**可更换磁盘**（removable disk）可从磁盘驱动器内取出，并且用另一张盘替换。可更换磁盘的优点是，在容量有限的磁盘系统中，可以得到无限量的数据；而且，磁盘可以从一台计算机系统移动到另一台计算机系统。软盘和ZIP盒式磁盘是可更换磁盘的例子。

大多数磁盘是两面都有可磁化的涂层，这被称为**双面**（double-sided）磁盘；而一些低价位的磁盘系统使用**单面**（single-sided）磁盘。

某些磁盘驱动器内垂直安装**多个盘片**（multiple platter），盘间相隔约1英寸，同时安装多个支架（如图6-5所示）。多盘片磁盘使用可移动磁头，每面有一个读-写磁头，所有这些磁头被机械地固定，以便与盘片中心等距离并一起移动。于是，任何时刻，所有磁头都位于与盘片中心等距离的各面磁道上，所有盘片上相同的相对位置的一组磁道被称为一个**柱面**（cylinder）。例如，图6-6中所有画有阴影的各磁道属于同一柱面。

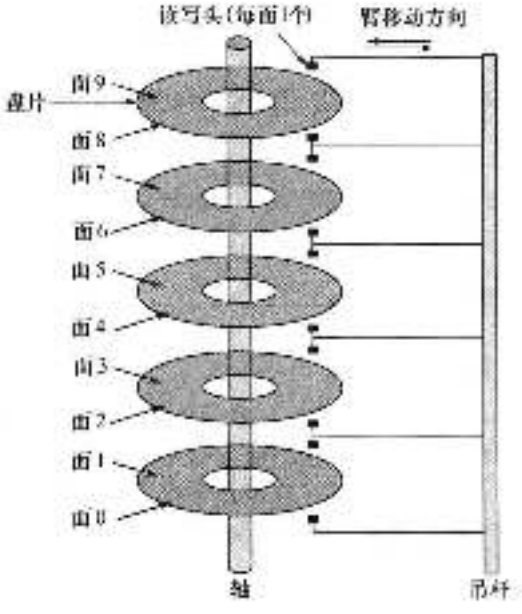


图6-5 磁盘驱动器部件

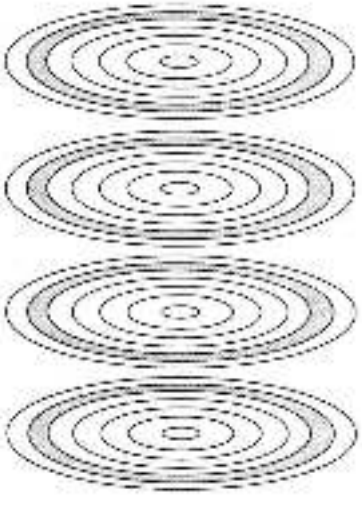


图6-6 磁道和柱面

依据磁头机制可以把磁盘分成三大类。传统上，读-写头放置在盘片上方的固定距离，允许有一个气隙。另一个极端的情况是，读-写头在读或写操作时实际地物理接触磁表面，这种机制用于软盘（floppy disk），其容量小、使用灵活和价格便宜。

要理解第三类磁盘类型，需要讨论数据密度和气隙大小的关系。磁头只有能产生或感应有足够强度的电磁场才能恰当地进行写和读操作。磁头越窄，离盘面越近才能起作用。较窄的磁头意味着较窄的磁道，因而有较大的数据密度，这是我们所需要的。然而，由于磁盘介质不纯和有缺陷等原因，磁头离盘面越近，其出错率就越高。为了解决这一问题，人们开发了温彻斯特（Winchester，简称温氏）磁盘。温氏磁盘磁头封装在几乎无污染的密封装置中，这种磁头与常规刚性磁头相比，其读写操作更加贴近磁盘表面，因此数据密度更大。当磁盘不动时，磁头实际上以气垫的形态轻停在磁盘表面。而磁盘旋转时，由盘片旋转产生的气压使气垫升高而将磁头与磁盘分离。结果是这种非接触系统可以比常规刚性磁头使用更窄的磁头，以更贴近的距离来读写磁盘表面数据[⊖]。

表 6-2 给出了典型的当代高性能磁盘的磁盘参数。

表 6-2 典型的硬盘驱动器参数

特性	Seagate Barracuda ES.2	Seagate Barracuda 7200.10	Seagate Barracuda 7200.9	Seagate	Hitachi Microdrive
应用	大容量服务器	高性能桌面系统	入门级桌面系统	膝上型电脑	手持设备
容量	1TB	750GB	160GB	120GB	8GB
最小寻道时间	0.8ms	0.3ms	1.0ms	—	1.0ms
平均寻道时间	8.5ms	3.6ms	9.5ms	12.5ms	12ms
旋转速度	7200rpm	7200rpm	7200rpm	5400rpm	3600rpm
平均旋转延时	4.16ms	4.16ms	4.17ms	5.6ms	8.33ms
最大传输率	3GB/s	300MB/s	300MB/s	150MB/s	10MB/s
每扇字节数	512	512	512	512	512
每盘面磁道数	8	8	2	8	2

6.1.4 磁盘性能参数

磁盘 I/O 操作的实际细节取决于计算机系统、操作系统、I/O 通道特性和磁盘控制器硬件。图 6-7 给出了一个常规的磁盘 I/O 传送时序图。

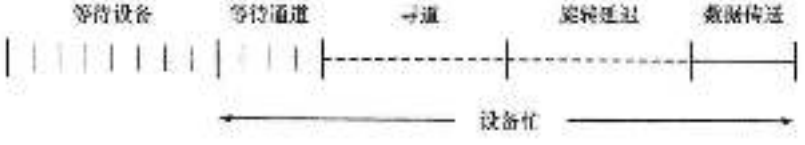


图 6-7 磁盘 I/O 传送的时序

当磁盘驱动器运行时，磁盘以恒定的速度旋转。为了读或写操作，磁头必须精确定位到想要的磁道和该磁道上想要的扇区的起始处。磁道的选择包括在可移动磁头系统中移动磁头或在固定磁头系统中选择某个磁头。在可移动磁头系统中，磁头定位到该磁道所花的时间称为寻道时间（seek time）。无论那一种磁头系统，一旦磁道被选定，磁盘控制器将处于等待状态，直到相应

⊖ 由于历史的原因，Winchester 这一术语在其对外发布之前，最早是作为 IBM 公司 3340 磁盘模型的代码名称。3340 磁盘是一种可更换式磁盘，其磁头都被封装在驱动器内。目前，这一术语用于任何采用气体动压磁头设计的密封磁盘驱动器。温彻斯特磁盘普遍用于大量个人计算机和工作站，只不过这里称之为硬盘。

的扇区旋转到磁头可以进行读/写的位置。等待相应扇区的起始处到达磁头的这段时间称为旋转延迟（rotational delay 或 rotational latency）。寻道时间和旋转延迟的总和称为存取时间（access time），即定位到读或写位置的时间。一旦磁头定位后，扇区旋转到磁头下方时，就可完成读或写操作；这是整个操作的数据传送部分，传送所需的时间称为传送时间（transfer time）。

除存取时间和传送时间以外，还有几个与磁盘I/O操作有关的队列延迟。当进程发出一个I/O请求后，它首先要在一个队列中等待所需设备变为可用，直到此设备被分配给该进程。如果该设备与其他磁盘驱动器共享一个或一组I/O通道，则还可能有一个附加的等待，等待相关通道变为有效。此时，寻道工作完成，开始磁盘存取。

在某些高端的服务器系统中，使用了一种称为旋转定位监测（RPS）的技术。其工作过程如下：当寻道命令已发出时，释放它所占据的I/O通道去处理其他I/O操作；当寻道操作完成时，设备确定所需数据何时将转到磁头下；当所需扇区接近磁头时，设备设法重新建立与主机通信的路径。如果控制器或通道正在忙于处理其他I/O，则重连接失败，并且设备必须旋转一整周后，才能再次试图重连接，这称为一次RPS失效。这个额外的延迟单元必须加入到图6-7的时序中。

1. 寻道时间

寻道时间是指移动磁盘臂到所要求的磁道处所花费的时间。这是一个难于精确定量的时间，它由两个主要部分组成：初始启动时间和一旦访问臂加速到指定速度后还必须跨越若干磁道所用的时间。遗憾的是，跨越时间不是磁道数的线性函数，它还包括一个校正时间（即从磁头定位到目标磁道至磁道标识被证实的一段时间）。

许多改进来自使用更小、更轻的磁盘元件。几年前，典型的磁盘直径是14英寸（36厘米），而现在的普遍尺寸是3.5英寸（8.9厘米），减少了磁盘臂必须跨越的距离。当代硬盘的典型平均寻道时间小于10ms。

2. 旋转延迟

与软盘不同，硬盘的旋转速率从每分钟3600转（如数码照相机这类手持设备中）到每分钟20 000转。对于20 000转/分的转速而言，则是每3ms转一周，因此平均旋转延迟是1.5ms。

3. 传送时间

硬盘的数据传送时间与磁盘旋转速度之间的关系如下式所示：

$$T = \frac{b}{rN}$$

这里：

T = 传送时间

b = 传送的字节数

N = 每磁道的字节数

r = 旋转速率，单位是转/秒

于是，总的平均访问时间可以表示为：

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

其中，T_s 是平均寻道时间。注意，对于多带记录式磁盘，因为每磁道的字节数是变化的，所以计算会变复杂[3]。

4. 定时比较

根据上面定义的参数，让我们来考察两个不同的I/O操作，它们说明过分依靠平均值的危险。

---

[3] 将上述两个公式与公式（4.1）进行比较。

考虑一个广告称平均寻道时间为4ms的磁盘，其转速为15 000rpm，每磁道500扇区，每扇区512B。假设我们希望读取一个由2500个扇区组成的总长为1.28MB的文件，让我们来估计总的传送时间。

首先，假定该文件尽可能紧凑地存储于磁盘上，即该文件占据相邻5个磁道的全部扇区（5道×500扇区/道=2500扇区）。这也就是所谓的顺序组织（sequential organization）。现在，可求出读取第1个磁道所用的时间：

平均寻道	4ms
平均旋转延迟	2ms
读500个扇区	4ms
合计	10ms

假设其余磁道基本上不再需要寻道时间，即I/O操作能够跟得上磁盘数据流速度，则读取后续磁道最多只需要考虑旋转延迟。于是，读每一个后续磁道的用时为2+4=6ms。读整个文件用时：

$$\text{总时间} = 10 + 4 \times 6 = 34\text{ms} = 0.034\text{s}$$

现在让我们来重新计算随机存取而不是顺序存取方式读取同一数据所需要的时间，即假设该文件的各扇区随机散布在磁盘上。对每一扇区，有：

平均寻道	4 ms
旋转延迟	2 ms
读1个扇区	0.008ms
合计	6.008ms

于是，读整个文件的总时间=2500×6.008=15 020ms=15.02s。

很明显，扇区的读取次序对I/O性能有巨大影响。当文件由多个扇区组成时，我们有某种控制扇区分布的方法。虽然如此，即使是多道程序下的文件访问，也会存在I/O请求竞争同一磁盘的问题。因此，考虑一种改善磁盘I/O性能的办法，使之能超出纯随机磁盘存取的性能，是很有价值的。这导致对磁盘调度算法的考虑，这属于操作系统的范畴，超出了本书的范围（详细讨论参见[STAL05]）。



6.2 RAID

如前所述，辅存性能的改进速度比处理器和主存的性能改进速度要慢得多。这种不匹配也许会是磁盘存储系统成为改进整个计算机系统性能的焦点所在。

和计算机的其他性能一样，磁盘存储器设计者认识到：如果一种元件只能推进到这个程度，那么通过并行使用多个元件会获得意外的性能。在磁盘存储器的研究上，这种思想导致了独立操作和并行处理的磁盘阵列的开发。由于是多盘，因此，只要请求的数据驻留在分离的盘上，则分离的I/O请求可并行处理。更进一步，如果将要存取的数据块分布在多个盘上，则单个I/O请求也能够并行处理。

随着多盘的使用，出现了很多种使用多盘组织数据和增加数据冗余来提高其可靠性的方法。但这却很难开发出可用于多操作平台和操作系统的数据库方案。幸运的是对多盘数据库的设计，工业上通过了一个称为RAID（独立磁盘冗余阵列）的标准方案。RAID方案分为7级[⊖]（0～6级），但这些级别并不是简单地表示层次关系，而是表示具有下列3个共同特性的不同设计结构。

⊖ 一些研究人员和一些公司还定义了其他的级，但本节描述的六个级是世界公认的。

（1）RAID 是一组物理磁盘驱动器，在操作系统下被视为一个单一的逻辑驱动器。

（2）数据以条带化（striping）的方式分布在一组物理磁盘上，这在后面会讨论。

（3）冗余磁盘容量用于存储奇偶校验信息，保证磁盘万一损坏时能恢复数据。

第（2）和（3）条特性的详细内容在不同的 RAID 级中是不同的，RAID 0 和 RAID 1 不支持第（3）特性。

术语 RAID 最早出现在一篇由加州大学伯克利分校的研究小组撰写的论文［PATT88］[⊖]中，此论文概括了各种 RAID 的配置和应用，并介绍了现在仍在使用的 RAID 级定义。RAID 策略是使用多个磁盘驱动器，它以这样一种方法来分布数据，以便能同时从多个磁盘中存取数据，因而改善了 I/O 性能，并且更方便增加磁盘容量。

RAID 方案的独特贡献是有效解决了对冗余的需求。尽管允许多个磁头和机械臂移动机构同时操作，以达到更高的 I/O 和传输速度，但多个设备的使用增加了出错概率。为了对这种可靠性的降低进行补偿，RAID 使用存储的奇偶校验信息来恢复因磁盘损坏而丢失的数据。

下面我们讨论 RAID 中的每一级，表 6-3 对这 7 级进行了粗略的介绍。表中显示了 I/O 性能在数据传输能力（移动数据的能力）和 I/O 请求速率（满足 I/O 请求的能力）两项。因为相对这两个度量标准，RAID 各级表现出本质的不同。RAID 各级的支撑点是用阴影突出的部分。图 6-8 表示一个支持 4 个无冗余磁盘数据容量的 7 级 RAID 方案。此图突出了用户数据和冗余数据的分布，并指出不同级相应的存储容量需求。这张图将贯穿下面讨论的整个过程。

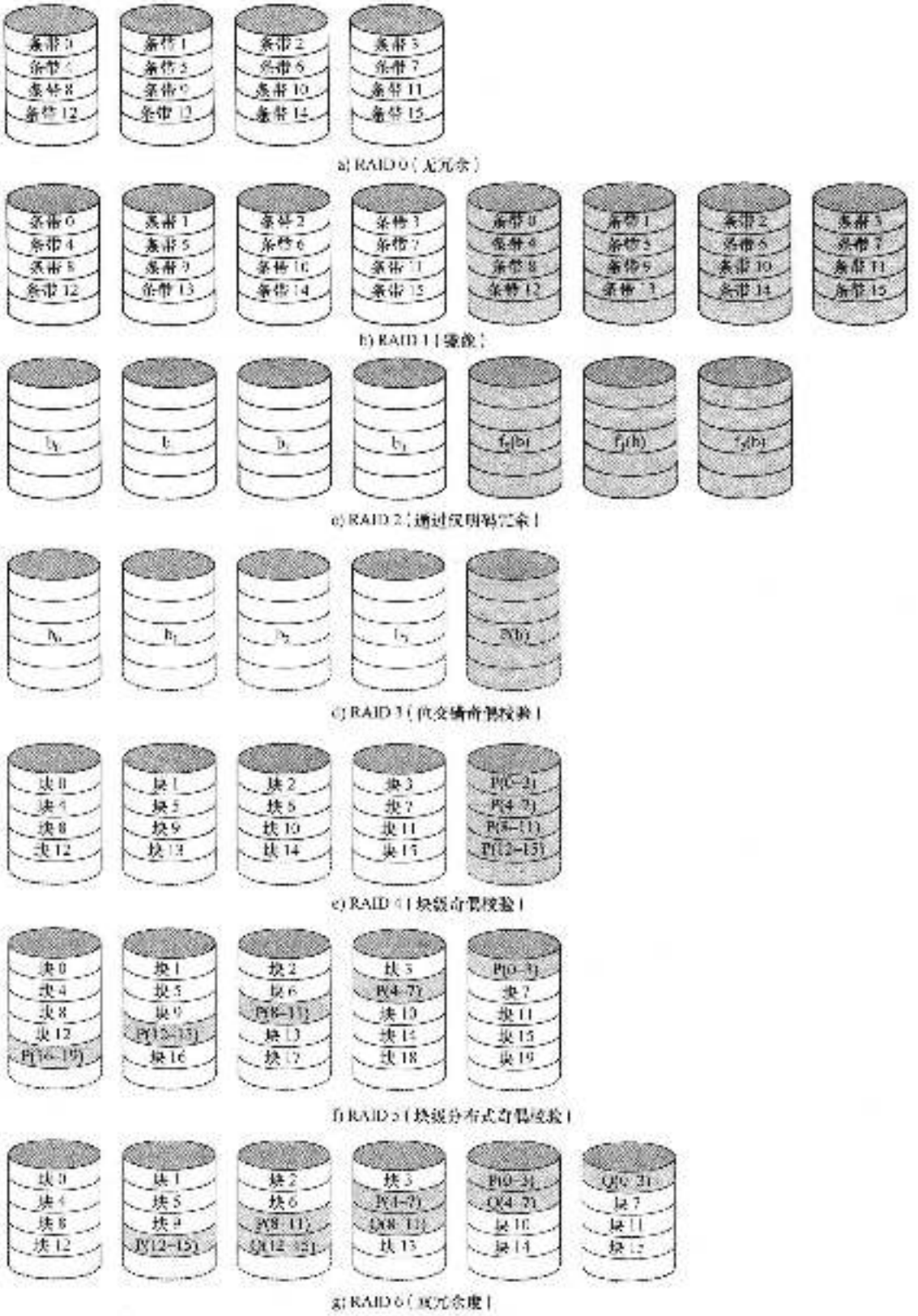
表 6-3　RAID 级别

种类	级别	描述	磁盘要求	数据可用性	大 I/O 数据传输能力	小 I/O 请求速率
条带化	0	非冗余	N	比单盘低	很高	读和写都很高
镜像	1	镜像	$2N$	比 RAID 2、3、4、5 高；比 RAID 6 低	读比单盘高；写与单盘类似	读高达单盘的两倍；写与单盘类似
并行存取	2	汉明码冗余	$N+m$	比单盘高很多，与 RAID 3、4、5 差不多	列表各级中最高	接近于单盘的两倍
	3	位交错奇偶校验	$N+1$	比单盘高很多；与 RAID 2、4、5 差不多	列表各级中最高	接近于单盘的两倍
独立存取	4	块交错奇偶校验	$N+1$	比单盘高很多；与 RAID 2、3、5 差不多	读与 RAID 0 类似；写低于单盘	读与 RAID 0 类似；写显著低于单盘
	5	块交错分布式奇偶校验	$N+1$	比单盘高很多；与 RAID 2、3、4 差不多	读与 RAID 0 类似；写低于单盘	读与 RAID 0 类似；写显著低于单盘
	6	块交错分布式奇偶校验	$N+2$	列表各级中最高	读与 RAID 0 类似；写比 RAID 5 低	读与 RAID 0 类似；写显著低于 RAID 5

6.2.1　RAID 0 级

RAID 0 级不是 RAID 家族中的真正成员，因为它不采用冗余来改善性能。但是，它有一些应用，例如应用于超级计算机上时，其性能和容量仅是基本的考虑，低成本比改善可靠性更重要。

⊖ 在这篇论文中，首字母缩写词“RAID”代表廉价的磁盘冗余阵列。术语“廉价的”是指将 RAID 阵列中小且相对便宜的磁盘与可供选择的、单个的大磁盘（SLED）所进行的比较。SLED 采用 RAID 和非 RAID 配置相似的磁盘技术，但已是过时设备。因此，业界采用术语“independent”（独立的）来阐述 RAID 阵列带来了显著的性能和可靠性收益。



a) RAID 0（无冗余）

b) RAID 1（镜像）

c) RAID 2（通过汉明码冗余）

d) RAID 3（位交错奇偶校验）

e) RAID 4（块级奇偶校验）

f) RAID 5（块级分布式奇偶校验）

g) RAID 6（双冗余度）

图 6.8 RAID 级别

对于RAID 0，用户和系统数据分布在阵列中的所有磁盘上，与单个大容量磁盘相比，它的显著优点是：如果两个I/O请求正在等待两个不同的数据块，则被请求的块可能在不同的磁盘上，这是一个好机会。因此，两个请求能够并行发出，减少了I/O的排队时间。

RAID 0以及其他所有的RAID级，与在磁盘阵列中简单地分布数据相比，它能以条带的形式在可用的磁盘上分布数据，因而更加完善。仔细研究图6-9可加深理解。所有用户数据和系统数据被看成是存储在逻辑磁盘上，磁盘以条带的形式划分，这些条带可以是一些物理的块、扇区或其他单位。数据条带以轮转方式映射到RAID阵列中连续的物理磁盘。一组逻辑上连续的条带被定义为条带集，它准确地与阵列中每个磁盘中一个条带相映射。在一个有n个磁盘的阵列中，第1组的n个逻辑条带存储在每个磁盘的第1个条带上，构成第一个条带集；第2组的n个逻辑条带存储在每个磁盘的第2个条带上；以此类推。这种布局的优点是，如果单个I/O请求由多个逻辑相邻的条带组成，则多达对n个条带的请求可以并行处理，这样大大地减少了I/O传输时间。

图6-9表示使用阵列管理软件在逻辑磁盘和物理磁盘间进行映射，此软件可在磁盘子系统或主机上运行。

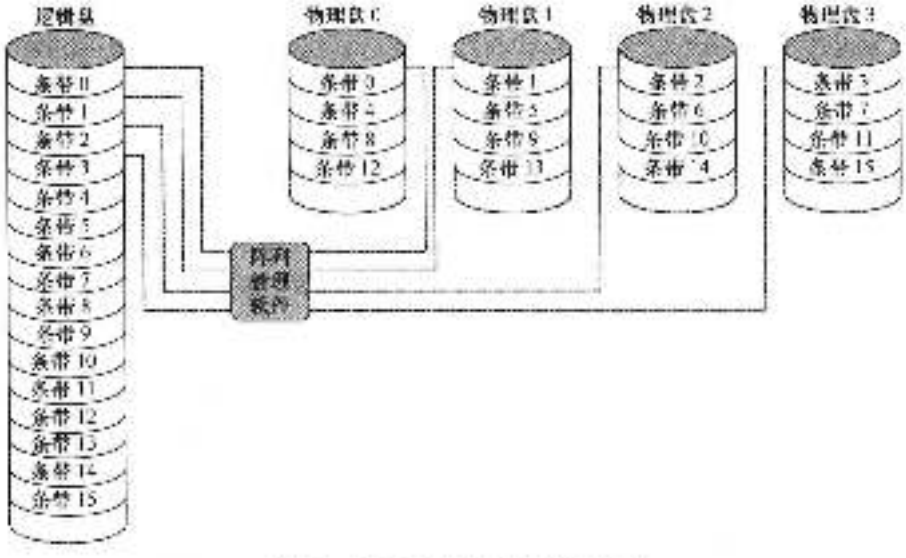


图6-9 RAID 0级阵列的数据映射

1. RAID 0用于高速数据传输

任何RAID级的性能关键取决于主机的请求方式和数据分布。由于RAID 0中无冗余影响，所以这些问题能在RAID 0中很清楚地得以分析。首先，我们考虑使用RAID 0达到高速数据传输的情况。为了在应用中达到高速数据传输，必须满足两个要求：第一，高速传输能力必须存在于主存和各独立磁盘之间的整个路径上，它包括内部控制器总线、主系统I/O总线、I/O适配器和主机存储器总线。

第二，应用必须使驱动磁盘阵列的I/O请求有效。与一个条带的大小相比，如果请求的是大量的逻辑相邻的数据，则满足这个要求。此时，单一I/O请求包含了多个磁盘的并行数据传输，与单个磁盘传输相比较，显然增大了有效的传输率。

2. RAID 0用于高速的I/O请求

在面向事务处理的环境中，用户普遍关心的是响应时间，而不是传输率。对于一个针对少许数据的单个I/O请求，其I/O时间由磁头的活动（寻道时间）和磁盘的活动（旋转延迟）决定。

在这一环境中，每秒可能有上百个I/O请求。通过平衡多磁盘中的I/O负载，磁盘阵列能提供较高的I/O执行速度。只有当多个I/O请求发出时，才能实现有效的负载平衡。这意味着存在

多个独立的应用或能进行多个 I/O 异步请求的面向事务的单个应用。性能也将受到条带大小的影响。如果条带容量相对大，则单个 I/O 请求只涉及一个磁盘存取，因此多个等待 I/O 的请求能并行处理，这样就减少了每个请求的排队时间。

6.2.2 RAID 1 级

RAID 1 与 RAID 2～6 级的区别在于实现冗余的方法。在 RAID 2 到 RAID 6 中，采用了奇偶校验计算的某种形式来引入冗余。而在 RAID 1 中，只是采用简单地备份所有数据的方法来实现冗余。同 RAID 0 一样，RAID 1 采用了数据条带集，如图 6.8b 所示；但在此情况中，它的每个逻辑条带映射到两个不同的物理磁盘组中，因此，阵列中的每个磁盘都有一个包含相同数据的镜像盘。

RAID 1 组织的优点如下：

（1）一个读请求可以由包含请求数据的两个磁盘中的某一个提供服务，只要它的寻道时间加旋转延迟较小。

（2）一个写请求需要更新两个对应的条带，但这可以并行完成。因此，写性能由两者中较慢的一个写来决定（即包含较大的寻道时间和旋转延迟的那一个写）。然而，RAID 1 无"写损失"。RAID 2 到 RAID 6 使用奇偶校验位，因此当修改单个的条带时，阵列管理软件必须先计算，然后在修改实际条带时也修改奇偶校验位。

（3）恢复一个损坏的磁盘很简单。当一个磁盘损坏时，数据仍能从第 2 个磁盘中读取。

RAID 1 的主要缺点是价格昂贵，它需要支持两倍于逻辑磁盘的磁盘空间。因此，RAID1 的配置只限于用在存储系统软件、数据和其他关键文件的驱动器中。在这种情况下，RAID 1 对所有的数据提供实时备份，即使一个磁盘损坏，所有的关键数据仍能立即可用。

在面向事务的环境中，如果有大批的读请求，则 RADI 1 能实现高速的 I/O 速率。此时，RAID 1 的性能可以达到 RAID 0 性能的两倍。然而，如果 I/O 请求有相当大的部分是写请求，则它不比 RAID 0 的性能好多少。对读请求的百分比高和数据传送密集的应用，RAID 1 也可以提供对 RAID 0 性能的改进。如果应用把每个读请求分割，使得两个磁盘都参与，则性能就会改善。

6.2.3 RAID 2 级

RAID 2 和 RAID 3 都使用了并行存取技术。在并行存取阵列中，所有的磁盘成员都参与每个 I/O 请求的执行。一般情况下，各个驱动器的轴是同步旋转的，因此，每个磁盘上的每个磁头在任何时刻都位于同一位置。

和其他 RAID 方案一样，RAID 2 采用数据条带。在 RAID 2 和 RAID 3 中，条带非常小，经常小到一个字节或一个字。在 RAID 2 中，通过各个数据盘上的相应位计算纠错码，编码的位存储在多个奇偶校验盘的对应位。通常，采用汉明码，它能纠正一位错误，检测两位错误。

尽管 RAID 2 比 RAID 1 需要的磁盘少，但价格仍相当昂贵，冗余磁盘的数目与数据磁盘数目的对数成正比。对于单个读操作，所有磁盘同时读取，请求的数据和相关的纠错码被传送到阵列控制器。如果有一位错误出现，则控制器能够马上识别并纠正错误，因此读取时间不会减慢。对于单个写操作，所有数据盘和奇偶校验盘必须被访问以进行写操作。

RAID 2 只是一种在多磁盘易出错环境中的有效选择。对于单个磁盘和磁盘驱动器已给出高可靠性的情况，RAID 2 没有什么意义。

6.2.4 RAID 3 级

RAID 3 的组织方式与 RAID 2 相似，所不同的是：不管磁盘阵列大小，RAID 3 只需要一个冗余盘。RAID 3 采用并行存取，数据分布在较小的条带上。它不采用纠错码，而采用对所有数据

盘上同一位置的一组独立位进行简单计算的奇偶校验位。

1. 冗余

当某一驱动器损坏时，访问奇偶校验盘，并由其余设备重构数据。一旦损坏的驱动器被替换，则可以在新盘上重新保存丢失的数据，并且恢复操作。

数据的重构很简单。现在考虑一个5磁盘的阵列，X0到X3保存数据，X4是奇偶校验盘，奇偶校验的第i位的计算公式如下：

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

其中⊕是异或运算。

假设，磁盘X1损坏，在上述等式两边同时加上$X4(i) \oplus X1(i)$，则得到以下等式：

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

因此，X1磁盘上的每个数据条带的内容都可以从阵列中剩余磁盘的相应条带中重新生成。这条原则适用于RAID第3～6级。

当磁盘损坏时，所有的数据仍有效的情况，我们称为简化模式。在这个模式下的读操作，利用异或运算可以立即重新生成丢失的数据。当数据写入简化模式的RAID 3阵列中时，要保持重新生成数据的奇偶校验的一致性。返回到全模式操作需要更换损坏的磁盘，并在新磁盘上重新生成损坏磁盘上的全部内容。

2. 性能

因为数据被分成非常小的条带，所以RAID 3能够获得非常高的数据传输率。任何I/O请求将包含所有数据盘的并行数据传送。对于大量传送，性能改善特别明显。另一方面，一次只能执行一个I/O请求，因此，在面向事务的环境中，性能将受损。

6.2.5 RAID 4级

RAID 4到RAID 6都采用一种独立的存取技术。在独立存取阵列中，每个磁盘成员的操作是独立的，因此各个I/O请求能够并行处理。基于此原因，独立存取阵列更适合于需要高速I/O请求的应用，而相对较少用于需要高数据传输率的场合。

与其他RAID方案一样，它采用数据条带。在RAID 4到RAID 6中，数据条带相对大些。在RAID 4中，通过每个数据盘上的相应条带来逐位计算奇偶校验条带，奇偶校验位存储在奇偶校验盘的对应条带上。

当执行较小规模的I/O写请求时，RAID 4蒙受了写损失。对于每一次写操作，阵列管理软件不仅要修改用户数据，而且要修改相应的校验位。现在来考虑一个5磁盘的阵列，其中X0到X3包含数据，而X4是奇偶校验盘。假设写操作只在X1盘的一个条带上执行。初始时，对于每个第i位，有下列关系：

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \tag{6.1}$$

修改后，可能改变的位以撇号（'）表示：

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

由上面的一组公式可推导出下列规则。第一行表明X1发生改变将影响奇偶校验盘X4。在第二行中，加入了$\oplus X1(i) \oplus X1(i)$。因为任意位与自身相异或结果为0，而0对等式不产生影响，但它能方便地通过重新排序来形成第三行。最后，根据等式6.1用$X4(i)$来替换前面4项。

为了计算新的奇偶校验位，阵列管理软件必须读取旧的数据条带和奇偶校验条带，然后用新的数据和新计算出的奇偶校验位更新上述两个条带。因此，每个条带的写操作包括两次读操

作和两次写操作。

当涉及所有磁盘的数据条带的较大 I/O 写操作时，只要用新的数据位来进行简单的计算即可得奇偶校验位。因此，奇偶校验盘和数据磁盘并行更新，不再需要另外的读或写操作。

在任何情况下，每一次的写操作必须涉及奇偶校验盘，因此它成为一个瓶颈。

6.2.6 RAID 5 级

RAID 5 和 RAID 4 的组织方式相似，不同的是，RAID 5 在所有磁盘上都分布了奇偶校验条带。常用轮转分配方案，如图 6-8f 所示。对于一个 n 磁盘阵列，最初的 n 条带的奇偶校验条带位于不同的磁盘，然后以此样式重复。

在所有磁盘上分布奇偶校验条带避免了 RAID 4 中潜在的 I/O 瓶颈问题。

6.2.7 RAID 6 级

伯克利研究小组在后来的论文中提出了 RAID 6 [KATZ89]。RAID 6 方案进行两种不同的奇偶校验计算，并将校验码以分开的块存于不同的磁盘中。因此，用户数据需要 N 个磁盘的 RAID 6 阵列，由 $N+2$ 个磁盘组成。

图 6-8g 说明了这种策略。P 和 Q 是两个不同的数据校验算法，其中一个是用于 RAID 4 和 RAID 5 中的异或计算，另一个是一种独立的数据校验算法。这样，即使是包含用户数据的两个盘出现故障了，数据照样能重新生成。

RAID 6 的优点是提供了极高的数据可用性，只有在平均修复时间（mean time to repair，MTTR）间隔内 3 个磁盘都出了故障，才会使数据丢失。另一方面，RAID 6 存在实质性的写损失，因为每次写都要影响两个奇偶块。性能基准 [EISC07] 显示，与 RAID 5 的实现相比，RAID 6 控制器可以承受超过 30% 的写性能下降。而 RAID 5 和 RAID 6 的读性能相当。

表 6-4 是 7 个级别的比较小结。

表 6-4 RAID 比较

级	优 点	缺 点	应 用
0	通过将 I/O 负载分散到多个通道和驱动器，极大地改善了 I/O 性能 无奇偶计算开销 很简单的设计 易实现	只要有某一个驱动器失效就导致阵列全部数据丢失	视频制作和编辑 图像编辑 预压缩应用 任何要求高带宽的应用
1	数据 100% 的冗余，意味着磁盘失效时无需重构，只需对替代盘拷贝即可 某些环境下，RAID 1 能承受多个驱动器同时失效 最简单的 RAID 存储子系统设计	在所有 RAID 类型中，磁盘数开销最大（100%）——低效	统计、工资单、财务和任何要求很高可用性的应用
2	可能有极高的数据传输率 数据传输率要求得越高，数据盘对 ECC 盘的比值越好 与 RAID3、4 和 5 级相比，控制器设计相对简单	短字长时，ECC 盘对数据盘的比值非常高——低效 入门级成本很高——要求证实很高数据传输率的需求是恰当的	无商品实现的存在/无商业化应用
3	很高的读数据传输率 很高的写数据传输率 磁盘失效时对吞吐率无显著影响 ECC（奇偶）盘对数据盘的低比率意味着高效率	最好情况（如果主轴同步旋转）下的事务率等同于单盘的事务率 控制器设计相当复杂	视频制作和存储 图像编辑 视频编辑 预压缩应用 任何要求高吞吐率的应用

（续）

级	优 点	缺 点	应 用
4	很高的读数据事务率 ECC（奇偶）盘对数据盘的低比率意味着高效率	十分复杂的控制器设计 最差的写事务率和写聚集传输率 磁盘失效事件中，数据重构困难并低效	无商品实现的存在/无商业化应用
5	最高的读数据事务率 ECC（奇偶）盘对数据盘的低比率意味着高效率 好的聚集传输速率	最复杂的控制器设计 磁盘失效事件中，数据重构困难（与RAID 1级相比）	数据和应用服务器 数据库服务器 Web，E-mail 和新闻组服务器 Intranet 服务器 用途最多的 RAID 级
6	提供极高的数据故障容忍能力并能承受多个驱动器同时失效	较复杂的控制器设计 计算奇偶校验地址的控制器开销非常高	对丢失数据严重的应用是理想的解决方案

6.3 光存储器

在1983年，推出的最成功的消费产品之一是光盘（CD）数字音频系统。光盘是一种不可擦除盘，它能在单面上存储超过60分钟的音频信息。CD的巨大商业成功促进了低成本光盘存储技术的发展，这一技术带来了计算机数据存储技术的一次革命。现已推出多种光盘系统（如表6-5所示），下面进行简要介绍。

表6-5 光盘产品

名 称	描 述
CD（compact disk，光盘）	存储数字音频信息的不可擦除盘。标准系统使用12厘米的盘，能够记录可连续播放60分钟以上的信息
CD-ROM（compact disk read-only memory，光盘只读存储器）	用于存储计算机数据的不可擦除盘。标准系统使用12厘米向盘，能够存储650MB以上的信息
CD-R（CD recordable，可刻录光盘）	类似于CD-ROM，用户只能向盘写入一次
CD-RW（CD rewritable，可重写光盘）	类似于CD-ROM，用户能多次擦除和重写盘
DVD（digital versatile disk，数字多功能光盘）	一种制作数字化的压缩的视频信息以及其他大容量数字数据的技术。使用直径为8或12厘米的盘，双面容量高达17GB。基本的DVD是只读的（DVD-ROM）
DVD-R（DVD recordable，可刻录DVD）	类似于DVD-ROM，用户只能向盘写入一次，只有一面盘能使用
DVD-RW（DVD rewritable，可重写DVD）	类似于DVD-ROM，用户能多次擦除和重写盘，只有一面盘能使用
Blu-Ray DVD（High definition video disk，高清晰视频光盘）	使用405nm（蓝-紫色）的激光，提供比DVD大约多的数据存储密度，单面单层能存储25GB的信息

6.3.1 光盘

1. CD-ROM

音频CD和CD-ROM二者采用类似的技术，主要区别是CD-ROM播放器更耐用，并且有纠错设备来保证数据正确地从光盘传输到计算机。这两类盘均采用相同方法制造。盘本体由树脂（如聚碳酸酯）制成。数字形式记录的信息（音乐或计算机数据）以一系列微凹坑的样式刻录在表面上。首先，用精密聚焦的高强度激光束制造成一个母盘，然后以母盘作为模板压印出聚碳酸

能的复制品。首先在凹坑表面上镀上一层高反射材料（铝或金），并在这薄层上涂一层丙烯酸树脂以防灰尘或划伤。最后，在丙烯酸树脂层上面用丝网印刷术印制标签。

通过安装在光盘播放器或驱动装置内的低强度激光束从 CD 或 CD-ROM 处读取信息。当当次转动盘片使之经过激光束时，激光束能穿过透明的聚碳酸酯层；当遇到一个凹坑时，激光的反射光强度发生变化（如图 6-10 所示）。具体来说，激光束照在凹坑上，由于凹坑表面有些不平，因此光被散射，反射回的光强度变弱。凹坑之间的区域称为台（land），台的表面光滑平坦，反射回的光强度更高。光传感器检测凹坑与台之间的反射光强弱变化，并将其转换成数字信号。传感器以规整的间隔检测表面。一个凹坑的开始或结束表示一个 1；间隔之间无标高变动出现时，记录为一个 0。

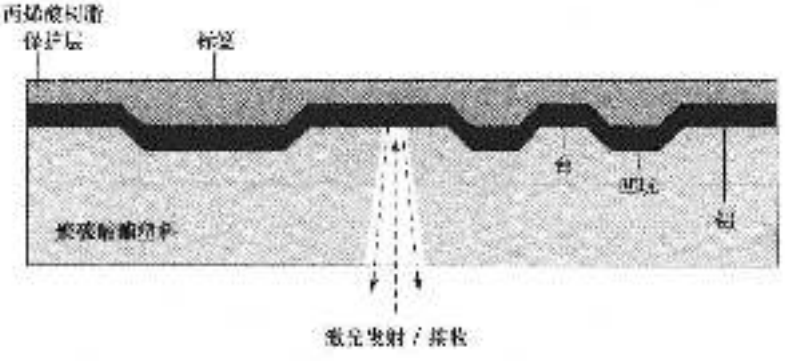


图 6-10 CD 操作

同磁盘，其信息被记录在同心圆形各磁道上。对简单的恒定角速度（CAV）系统而言，每个磁道的位数是固定的。为增加存储密度可使用多带记录方式，盘表面被划分成几个带，远离中心的带比最近中心的带能容纳更多的位。虽然这一技术增大了容量，但它仍然不是最优的。

为了实现更大的存储容量，CD 和 CD-ROM 不使用同心圆的追来组织信息，而是磁盘有一条螺旋式轨道，从靠近中心处开始，逐圈向外旋转直到盘的外沿。靠近外沿的扇区与靠近中心的扇区具有相同的长度。于是，信息以相同大小的段均匀分布在整个盘上，并且以相同的速度进行扫描，而盘片以变速旋转。因此，凹坑被激光以恒定线速度（constant linear velocity，CLV）读出。盘片在存取外沿时要比存取内沿时的旋转速度慢。于是，光道的容量和旋转延迟都使定位光道外沿的时间变长。CD-ROM 的数据容量大约是 680MB。

CD-ROM 上的数据被组织成一系列的块，典型的块格式如图 6-11 所示，它包含如下的一些域：

- Sync：此同步域标志一个块的开始，由 12 个字节组成，第 1 个字节为全 0，第 2～11 个字节为全 1，第 12 个字节为全 0。

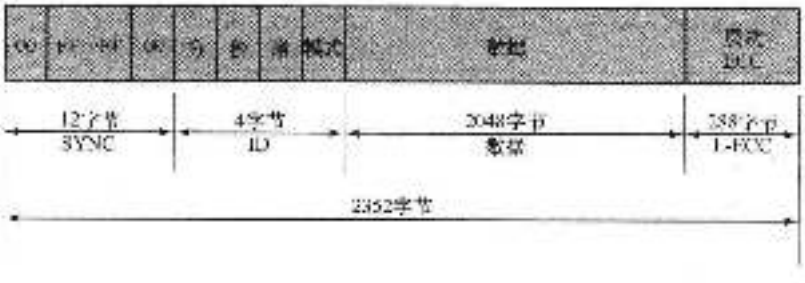


图 6-11 CD-ROM 块格式

- Header：此头部域包含块地址和模式字节，也称为块的标识（ID）域。模式0表示一个空的数据域，模式1表示使用纠错码和2048字节的数据，模式2表示不带纠错码的2336个字节的用户数据。
- Data：用户数据域。
- Auxiliary：此辅助域在模式2下是附加用户数据，在模式1下是288字节的纠错码。

采用CLV，随机存取变得更加困难。定位到一个指定地址，涉及移动头到某个区域、调整转速和读地址，然后再微调以找到并读取指定扇区。

CD-ROM适合于将大量数据发布给众多用户。由于初始的写过程开销较大，因此它不适合于单个应用。与传统的磁盘相比，CD-ROM有两个优点：

- 与磁盘不同，存储有信息的光盘批量复制价格便宜；而磁盘上的数据库复制每次都要通过两个磁盘驱动器间的复制来完成。
- 光盘是可换的，允许光盘作为归档存储；而大多数硬盘是不可更换的，硬盘存储新信息前，必须将原来有用信息转存到其他存储介质上。

CD-ROM的缺点是：

- 它是只读的，不能修改。
- 其存取时间比磁盘驱动器的长得多，大约半秒钟。

1. CD-R

为适应只需一组数据的一个或少数几个备份的应用，开发了一写多读CD，称为可刻录CD（CD-R）。CD-R盘用适当强度的激光可写入一次。因此，其盘控制器比CD-ROM的要贵些，用户一次性写入后可多次读出。

CD-R的介质类似于但不完全等同于CD或CD-ROM的介质。对于CD和CD-ROM，信息通过介质表面的凹坑来记录，这些凹坑改变了激光的反射率。而对于CD-R，其介质还包括了一个染色层。该染色层被用来改变反射率，并且由高强度激光激活。生成的盘既能在CD-R驱动器上也能在CD-ROM驱动器上读出。

CD-R光盘适宜用于文档和文件的归档存储，它提供了大量用户数据的永久性记录。

2. CD-RW

这种可重写光盘（CD-RW）和磁盘一样可重复地写和改写。虽然人们尝试了几种办法，但真正可行的纯光学办法是相变（phase change）盘。相变盘使用了一种在两种不同相位状态下有两种显著不同反射率的材料。一种是无定形状态，分子展示出一种随机取向，从而反射率低；另一种是结晶状态，表面光滑，反射率高。激光束能改变材料的相位状态，用于记录信息。相变盘的主要缺点是材料老化最终会失掉相位可变的特性。当前的材料可用于50万次到100万次的擦除。

与CD-ROM和CD-R相比，CD-RW的明显优势在于能够重写，因此可作为真正的辅助存储器。正因如此，它能与磁盘竞争。这种光盘的关键优点是：光盘的工程容错比大容量磁盘小得多，因此它具有较高的可靠性和较长的使用寿命。

6.3.2 数字多功能光盘

由于大容量的数字多功能光盘（DVD）的出现，电子业界最终找到了模拟VHS视频带的一种可以接受的替代物，DVD已经替代用于视频卡盒记录器（VCR）中的视频带。最重要的是它还将取代个人计算机和服务器中的CD-ROM。DVD使影视节目进入数字时代，它演播的影片有极好的画面质量，并且也能像音频CD盘（DVD机器也可以播放）那样随意访问。DVD的容量非常大，当前是CD-ROM容量的7倍。由于DVD的大容量和高质量，PC游戏已经变得更逼真、教育软件也会包括更多画面。紧接着这些开发而来的是，当这些内容进入Web站点之后，Internet和企业内联网上的信息流量将会达到一个新高峰。

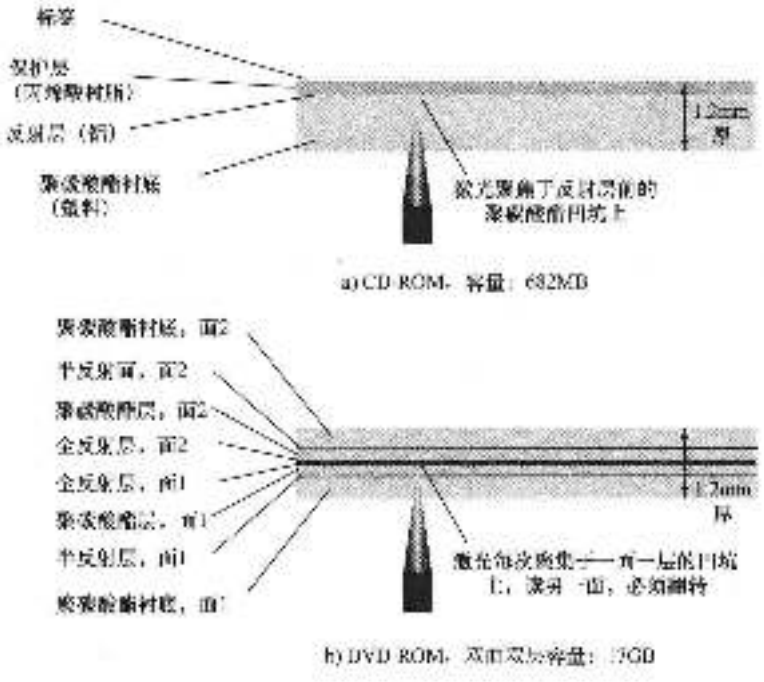


图 6-12 CD-ROM 和 DVD-ROM

DVD 的更大容量是由于它与 CD 有 3 个不同点（参见图 6-12）：

（1）DVD 上的位组装更紧密。CD 上的螺旋式光道的两个间间隙是 1.6μm，光道上凹坑之间的最小距离是 0.834μm。而 DVD 的激光波长更短，并实现圈间间隙是 0.74μm 和凹坑间最小间距是 0.4μm。这两方面的改进，导致 DVD 的容量大约是 CD-ROM 的 7 倍，大约 4.7GB。

（2）DVD 采用双层结构（在第一层的顶部设置凹坑和台的第二层）。双层 DVD 在反射层上有一个半反射层。DVD 驱动器中的激光通过调整焦距能分别读取每一层。这一技术使盘的容量几乎翻一番，达到大约 8.5GB。由于第二层的反射率较低，限制了此层的容量，故容量的完全翻倍未能实现。

（3）DVD 能用两面记录数据，而 CD 只能用一面，这使 DVD 盘的容量高达 17GB。

与 CD 一样，DVD 也有只读的和可写的版本（参见表 6-5）。

6.3.3 高清晰光盘

高清晰光盘是为了存储高清晰度的视频，并提供比 DVD 容量大得多的存储空间而设计的。通过使用更短波长的激光，在蓝-紫光范围，可实现更高的位密度。与 DVD 相比，高清晰光盘由于使用的激光波长较短，因此其盘面上组成数字 0 和 1 的数据凹坑较小。

最初竞争市场的两种光盘格式和技术是：HD DVD 和蓝光 DVD。但蓝光光盘方案最终占据了市场的统治地位。HD DVD 方案能够在光盘单面的单层上存储 15GB 的数据。蓝光放置光盘上的数据层更靠近激光（如图 6-13e 所示），这就使得焦距变短、失真变小，因此凹坑和光道都变小。蓝光光盘能单面存储 25GB 的数据。目前有 3 个可用版本：只读（BD-ROM）、写入一次（BD-R）和可重复写（BD-RE）。

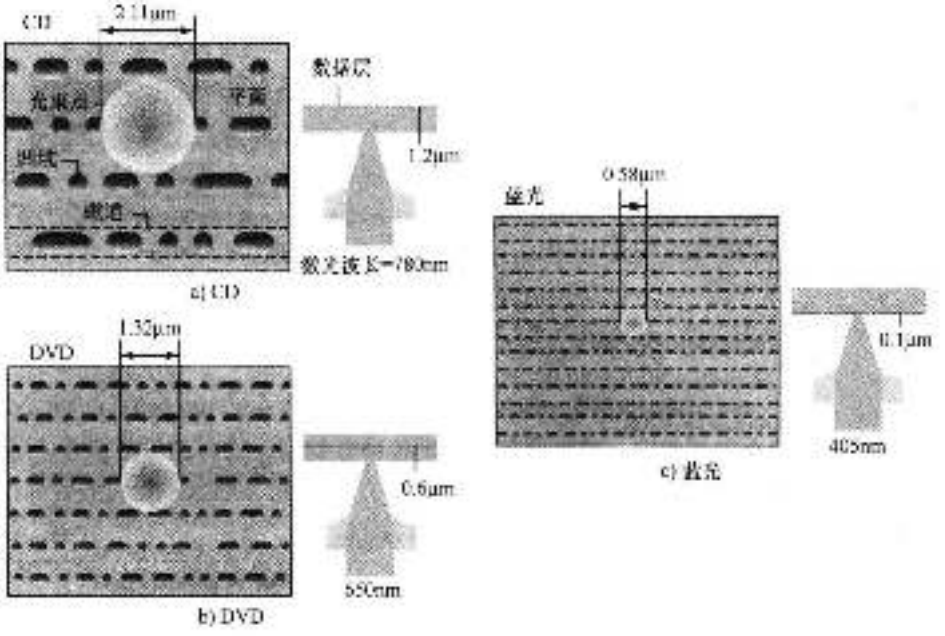


图 6-13　光存储器特性

6.4　磁带

磁带系统使用与磁盘系统相同的读取和记录技术。介质是柔韧的聚酯（与某些衣物的材料相似）薄膜带，外涂磁性材料；它可以是用专用胶黏合的纯金属材料，也可以是汽相扩散渗镀的金属薄膜。磁带和磁带机与家用的磁带录音机相似。磁带宽度从 0.38 厘米（0.15 英寸）到 1.27厘米（0.5 英寸）之间变化。磁带组装成一个有开口的卷筒，使用时要将它穿到第二个转轴上。实际上，今天所有的磁带都是盒式的。

磁带上的数据被构造成几个并行的磁道。早期的磁带系统一般使用 9 个磁道，这使得每次可以存取一个字节，而第 9 磁道上有附加的奇偶校验位。后来的磁带系统使用 18 或 36 个磁道，对应于一个数据字或双字。此种格式的数据记录被称为**并行记录**（parallel recording）。而大多数现代的系统使用**串行记录**（serial recording），数据作为一系列的位沿磁道顺序排放，如同磁盘那样。与磁盘相同，磁带数据以连续的块来进行读和写操作，这些块被称为**物理记录**（physical records）。磁带上的块由**记录间隙**（interrecord gap）来分隔。与磁盘一样，磁带格式化有助于定位物理记录。

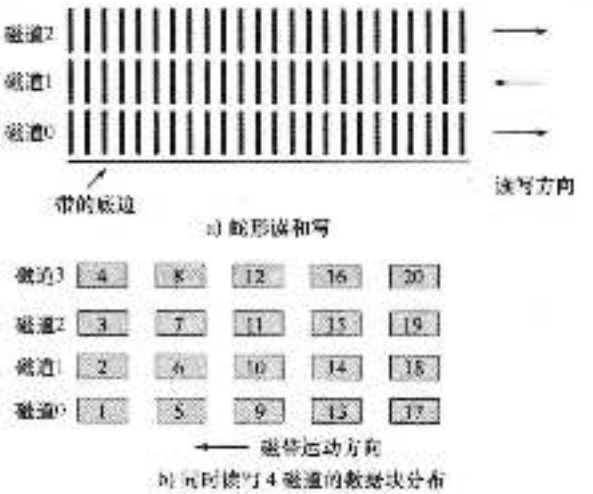


图 6-14　典型的磁带特征

用于串行磁带的典型记录技术称为**蛇形记录**（serpentine recording）。在此技术中，当记录数据时，首先沿整个磁带长记录下第一组数据位；到达带的尾端时，磁头再重新定位到新磁道上，又一次沿整个带长记录，只不过这次是在相反方向上。这个过程继续进行下去，直到磁带各磁道写满（如图 6-14a 所示）。为了提高速度，读-写头能同时对几个相邻磁道（通常是 2

到8（磁道）进行读写操作。数据仍是沿各磁道串行记录，但各块依序存储在相邻的磁道上，如图6-14b所示。

磁带驱动器是一种顺序存取（sequential-access）设备。如果磁头当前定位于记录1，则为了读记录N，它必须依次读物理记录1到$N-1$，每次一个。如果磁头当前定位已超越所需记录，则它必须倒带一定距离，再开始向前读。与磁盘不同，磁带只是在读/写操作期间才运动。

与磁带相比，磁盘驱动器是一种直接存取（direct-access）设备。磁盘驱动器不需要顺序读取磁盘上所有扇区来得到要求的某个扇区。它只需要在一个磁道上等待相关扇区并能连续存取任一磁道。

磁带是最早的辅助存储器，作为存储器层次体系结构中最低成本、最慢速度的成员，至今仍广泛使用。

现在市场上占统治地位的磁带技术是一种称为线性磁带开放协议（LTO）的盒驱动系统。LTO是20世纪90年代后期开发的，它是市场上可供各种专利系统选择的开放资源。表6-6给出了各代LTO产品的参数。详细内容参见附录J。

表6-6 LTO磁带驱动器

	LTO-1	LTO-2	LTO-3	LTO-4	LTO-5	LTO-6
发布时间	2000	2003	2005	2007	TBA	TBA
压缩容量	200 GB	400 GB	800 GB	1600 GB	3.2 TB	6.4 TB
压缩传输率（MB/s）	40	80	160	240	360	540
线密度（bit/mm）	4880	7398	9638	13 300		
磁带磁道	384	512	704	896		
磁带长度（m）	609	609	680	820		
磁带宽度（cm）	1.27	1.27	1.27	1.27		
写元素	8	8	16	16		

6.5 推荐的读物和Web站点

[JACO08]提供了磁盘的立体图。[MEE96a]给出了对磁盘和磁带系统底层的记录技术的很好综述。[MEE96b]则关注磁盘和磁带系统的数据存储技术。[COME00]是一篇短文章，但在论述磁盘存储技术当前发展趋势方面具有指导意义。[RADD08]和[ANDE03]提供了最近的磁盘存储技术讨论。

[CHEN94]由RAID概念的提出者撰写，它是对RAID技术的绝妙总结。[FRIE96]是一篇好的综述文章。[CHEN96]给出了几种RAID体系结构的性能比较。

[MARC90]很好地概述了光存储器领域。[MANS97]给出了基础记录和读取技术的综述。

[ROSO03]介绍了所有的外部存储系统和每种系统的详细技术。[KHUR01]是另一篇好的综述文章。

[HAEI07]提供了LTO的细节处理。

ANDE03 Anderson, D. "You Don't Know Jack About Disks." *ACM Queue*, June 2003.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; and Patterson, D. "RAID: High-Performance, Reliable Secondary Storage." *ACM Computing Surveys*, June 1994.

CHEN96 Chen, S., and Towsley, D. "A Performance Evaluation of RAID Architectures." *IEEE Transactions on Computers*, October 1996.

COME00 Comerford, R. "Magnetic Storage: The Medium that Wouldn't Die." *IEEE Spectrum*, December 2000.

FRIE96 Friedman, M. "RAID Keeps Going and Going and ..." *IEEE Spectrum*, April 1996.

HAUE08　Haeusser, B., et al. *IBM System Storage Tape Library Guide for Open Systems.* IBM Redbook SG24-5946-05, October 2007. ibm.com/redbooks.

JACO08　Jacob, B.; Ng, S.; and Wang, D. *Memory Systems: cache, DRAM, Disk.* Boston: Morgan Kaufmann, 2008.

KHUR01　Khurshudov, A. *The Essential Guide to Computer Data Storage.* Upper Saddle River, NJ: Prentice Hall, 2001.

MANS97　Mansuripur, M., and Sincerbox, G. "Principles and Technologies of Optical Data Storage." *Proceedings of the IEEE*, November 1997.

MARC90　Marchant, A. *Optical Recording.* Reading, MA: Addison-Wesley, 1990.

MEE96a　Mee, C., and Daniel, E. eds. *Magnetic Recording Technology.* New York: McGraw-Hill, 1996.

MEE96b　Mee, C., and Daniel, E. eds. *Magnetic Storage Handbook.* New York: McGraw-Hill, 1996.

RADD08　Radding, A. "Small Disks, Big Specs." *Storage Magazine*, September 2008.

ROSC03　Rosch, W. *Winn L. Rosch Hardware Bible.* Indianapolis, IN: Que Publishing, 2003.



推荐的 Web 站点

- Optical Storage Technology Association：有关光存储技术和厂商的好的信息源，并有相关链接的详细列表。
- LTO Web site：关于 LTO 技术和供应商的信息。

6.6　关键词、思考题和习题

关键词

access time：存取时间
Blu-ray：蓝光光盘
CD：光盘
CD-ROM：只读 CD
CD-R：可刻录 CD
CD-RW：可重写 CD
constant angular velocity（CAV）：恒定角速度
constant linear velocity（CLV）：恒定线速度
DVD：数字多功能光盘
DVD-ROM：只读 DVD
DVD-R：可刻录 DVD
DVD-RW：可重写 DVD
fixed-head disk：固定头磁盘
floppy disk：软磁盘
gap：间隙
head：头
land：台
magnetic disk：磁盘
magnetic tape：磁带
magnetoresistive：磁阻
movable-head disk：可移头磁盘
multiple zoned recording：多重区域记录
nonremovable disk：不可更换式盘
optical memory：光存储器
pit：凹坑
platter：盘片
RAID：磁盘冗余阵列
removable disk：可更换式磁盘
rotational delay：旋转延迟
sector：扇区
seek time：寻道时间
serpentine recording：蛇形记录
striped data：条带数据
substrate：衬底
track：磁道，道
transfer time：传送时间

思考题

6.1　磁盘使用玻璃衬底有什么好处？
6.2　数据如何写到磁盘中？
6.3　数据如何从磁盘读出？

6.4 说明简单的 CAV 系统与多带记录系统的区别。
6.5 定义磁道、柱面和扇区三个术语。
6.6 扇区的通常大小是多少？
6.7 定义寻道时间、旋转延迟、存取时间和传送时间四个术语。
6.8 所有 RAID 级别的公共特征是什么？
6.9 简要定义 RAID 的 7 个级别。
6.10 解释术语“条带化数据”。
6.11 RAID 系统中如何实现冗余？
6.12 在 RAID 环境中，并行存取和独立存取有何不同？
6.13 CAV 和 CLV 有何不同？
6.14 什么原因造成 DVD 比 CD 有更大的盘容量？
6.15 解释何为蛇形记录。

习题

6.1 考虑一个有 N 个磁道的磁盘，磁道编号由 0 到 $(N-1)$，并假定所要求的扇区随机分布在磁盘上。请计算寻道平均越过的磁道数。
(a) 计算磁头当前位于磁道 t 之上时长度为 j 的寻找概率。提示：此时需要确定无需越道的各种组合的总数。
(b) 计算长度为 K 的寻道概率。提示：这涉及对超过 K 个磁道的所有可能组合求和。
(c) 计算寻道超过的平均磁道数。对期望值使用如下公式：

$$E[x] = \sum_{i=1}^{N-1} i \times \Pr[x = i]$$

提示：可使用如下等式：

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}; \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

(d) 说明：对于一个大的 N 值，寻道平均越过的磁道数约为 $N/3$。

6.2 为一个磁带系统定义如下参数：
t_s = 寻道时间；磁头定位在磁道上的平均时间
r = 磁带的旋转速度，转/秒
n = 每个扇区的位数
N = 一个磁道的容量，单位：位
t_a = 存取一个扇区的时间
请推导 t_a 关于其他参数的函数关系式。

6.3 考虑一个有 8 个面的磁盘驱动器，每面有 512 个磁道，每道上有 64 个扇区，扇区大小为 1KB。平均寻道时间是 8ms，道间移动时间是 1.5ms，磁盘转速为 3600rpm。可以读取同一柱面上的连续磁道而磁头不需要移动。
(a) 磁盘容量是多少？
(b) 平均存取时间是多少？假设某文件被存储在连续柱面的连续扇区和连续磁道上，起始位置为柱面 i 上第 0 道的第 0 号扇区。
(c) 估计传送 5MB 大小的文件所需要的时间。
(d) 实发传送率是多少？

6.4 考虑一个单片磁盘，它有如下参数：旋转速率是 7200rpm，一面上的磁道数是 30 000，每道扇区数是 600，寻道时间是每横越 100 个磁道用时 1ms。假定开始时磁头位于磁道 0，磁盘收到一个存取随机磁道上随机扇区的请求。
(a) 平均寻道时间是多少？
(b) 平均旋转延迟是多少？
(c) 一个扇区的传送时间是多少？
(d) 满足此请求的总的平均时间是多少？

6.5 物理记录和逻辑记录有区别。逻辑记录（logical record）是相关数据元素的集合，它作为概念性的单

位，与信息如何存储和在何处存储无关。物理记录（physical record）是存储空间的一个连续区域，它由存储设备的特性和操作系统定义。

假定在一个磁盘系统中，每个物理记录包含30个120字节长的逻辑记录。若此磁盘系统有8面，每面有110磁道，每道有96扇区，每扇区固定长512B；请计算存储300 000逻辑记录将需要多大的磁盘空间（以扇区、磁道、面数来表示）。忽略任何文件头部记录和磁道索引，并假设记录不能跨两个扇区。

6.6 考虑一个转速为3600rpm的磁盘。磁头在相邻磁道间的移动时间为2ms，每个磁道上32个扇区，它们以线性次序存储，编号从0到31。磁头以升序访问这些扇区。假设开始时读/写磁头恰好在第8道第1号扇区的位置。而主存储器缓冲器足够容纳整个磁道。数据传送在磁盘区域之间进行，通过先从源磁道读入主存储器缓冲器中，然后再由主存储器缓冲器写入目标磁道来完成。

（a）将第8磁道第1号扇区的数据传送到第9磁道第1号扇区需要多长时间？

（b）若将第8磁道上所有扇区的数据传送到第9磁道的相应扇区上，那耗时又是多少？

6.7 当条带的大小小于I/O请求时，很显然，以条带划分磁盘能提高数据的传送速度。由于多个I/O请求能并行处理，RAID 0相对于单个大磁盘提供了改进的性能，这一点也是很明显的。然而，后一种情况磁盘需要用条带划分吗？即与不用条带划分的磁盘阵列相比，它是否改进了I/O请求速度的性能？

6.8 考虑一个有4个磁盘，每个磁盘200GB的RAID阵列。对于RAID 0、1、2、3、4、5和6级中的各级，有用数据的存储容量是多大？

6.9 对于某一光盘，声音通过16比特采样来转换成数字信号，并且被处理成8位的字节流进行存储。存储这种数据的一种简单方案称为直接记录，使用台表示1，凹坑表示0。而在另一种方案中，每个字节扩展为14位二进制数。结果表明在16134（2^{14}）个14位数据中正好有256（2^8）个数据，其二进制在每两个1之间至少有两个0，这些数据用来将8位扩展为14位。光学系统检测到从台到凹坑或从凹坑到台的跳变记为1，同时通过计量强烈变换之间的距离来记录0。这种方案要求不存在连续的1，因此采用8－14位编码。

这种方案的优点如下：对于给定的激光束直径，无论每一位用什么表示，都存在一个最小凹坑尺寸。采用这一方案，该最小凹坑存储3位，因为每个1后面都有两个0。而采用直接记录时，同样的凹坑仅仅只能存储1位。考虑每一个凹坑存储的位数和8－14位扩展，哪种方案能存储更多的位，并说明原因？

6.10 为计算机系统设计一种备份策略。一种方法是使用外部可插入磁盘，其成本是每500GB的磁盘驱动器花费150美元。另一种方法是花2500美元购买一个磁带驱动器，并用50美元购买一个400GB的磁带（这是2008年的市场价格）。通常的备份方案是使用两组现场备份媒体，此时备份可选择性地写入它们中，因此万一系统备份失败，前面的备份还是完整的。同时还存在场外的另一组媒体，场外的组定时地和现场的某一组进行数据交换。

（a）假设你有1TB（1000GB）的数据要进行备份，一个磁盘备份系统的费用是多少？

（b）一个1TB的磁带备份系统的费用是多少？

（c）为了使磁带备份方案更省钱，每一个备份需要多大？

（d）哪种备份方法更适用于磁带？

第7章

Computer Organization and Architecture: Designing for Performance, 8E

输入/输出

本章要点

- 计算机系统的I/O体系结构是系统与外部世界交接口。这种体系结构提供了一种控制计算机与外部世界交互的系统化方式，并向操作系统提供有效地管理I/O活动的必要信息。
- 有3种基本的I/O技术：**编程式I/O**（Programmed I/O）技术，即在请求I/O操作的程序的直接和连续的控制下所发生的I/O操作；**中断驱动式I/O**（Interrupt-driver I/O）技术，即程序发出I/O命令后继续执行，直到被I/O硬件中断，通知它I/O操作完成；还有**直接存储器存取**（DMA）技术，即一个专门的I/O处理器接管I/O操作的控制，在I/O设备与存储器之间直接传送大量数据。
- 外部I/O接口的两个重要实例是：FireWire和Infiniband。



除了处理器和一组存储器模块外，计算机系统的第三个关键部件是一组输入/输出（I/O）模块。每个模块连接到系统总线或中央交换器，并且控制一个或多个外围设备。一个I/O模块不是简单地将设备连接到系统总线的一组机械连接器，而是包含了执行设备与系统总线之间通信功能的逻辑。

读者可能会奇怪为什么不把外设直接连接到系统总线上，原因如下：

- 各种外设的操作方法是不同的，将控制一定范围的外设的必要逻辑合并到某个处理器内是不现实的。
- 外设的数据传送速度一般比存储器或处理器的慢得多，因此，使用高速的系统总线直接与外设通信是不切实际的。
- 另一方面，某些外设的数据传送速率比存储器或处理器要快，同样，若不适当管理，则速度失配将导致无效。
- 外设使用的数据格式和字长度通常与处理器不同。

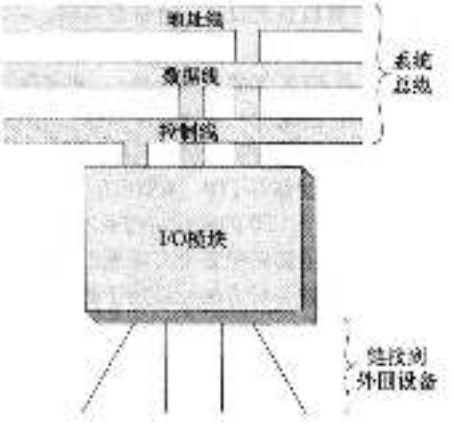


图7-1 I/O模块通用模型

因此，I/O模块是必需的，它有两大主要功能（如图7-1所示）：

- 通过系统总线或中央交换器与处理器和存储器连接。
- 通过专用数据线与一个或多个外设连接。

本章首先简要讨论外部设备，接着介绍I/O模块的结构和功能。然后叙述与处理器和存储器协同工作以完成I/O功能的各种方式：内部I/O接口。最后讨论I/O模块与外界之间的外部I/O接口。

7.1 外部设备

I/O操作是通过各种外部设备来完成的，这些外部设备提供了外部环境和计算机之间交换数据的方式。外部设备通过I/O模块的链接而与计算机相连（如图7-1所示），这种链接可以实现I/O模块与外部设备之间的控制信息、状态信息和数据信息的交换。连接到I/O模块的外部设备

通常被称为外围设备（peripheral device）或简称为外设（peripheral）。

从广义上可以将外部设备分为3类：

- 人可读设备：适用于与计算机用户通信。
- 机器可读设备：适用于与设备通信。
- 通信设备：适用于与远程设备通信。

人可读设备如视频显示终端（VDT）和打印机。机器可读设备如磁盘和磁带系统，以及传感器和动臂机构，如机器人中动臂机构的应用。注意，在这一章中，我们将磁盘和磁带系统视为I/O设备，而在第6章中，我们把它们作为存储设备来看待。从功能上来看，它们是存储器分层结构的一部分，其用途已在第6章中讨论过。而从结构上来看，它们由I/O模块控制，详细内容将在本章讨论。

通信设备允许计算机与远程设备交换数据，该远程设备可以是人可读的设备，如终端，也可以是机器可读的设备，甚至可以是另一台计算机。

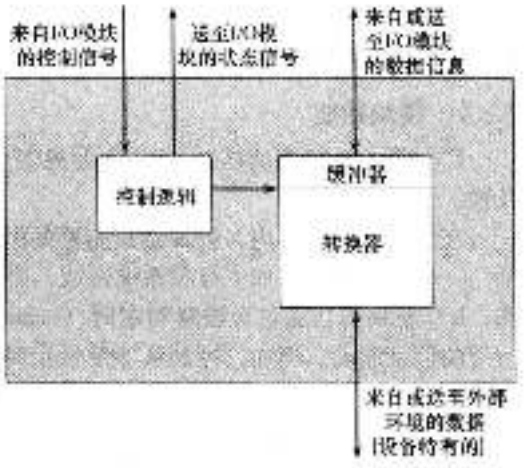


图7-2 外部设备框图

图7-2使用非常通用的术语表示了外部设备的性质。I/O模块的接口以控制、状态和数据信号的形式出现。控制信号（control signal）决定设备将要执行的功能。例如，发送数据到I/O模块（INPUT或READ信号），接收来自I/O模块的数据（OUTPUT或WRITE信号），报告状态或对特定设备进行控制（如定位磁头）。数据（data）信息是以一组位的形式发送到I/O模块或从I/O模块接收。状态信号（status signal）表示设备的状态，如READY/NOT-READY表示进行数据传送的设备是否就绪。

与设备相关的控制逻辑（control logic）控制设备的操作，以响应来自I/O模块的命令。输出时，转换器（transducer）把数据从电信号转换成其他的能量形式；输入时，转换器把其他信号形式转换成电信号。通常，缓冲器与转换器有关，它缓存I/O模块和外部环境之间传送的数据，缓冲器的大小一般为8位或16位。

I/O模块与外部设备之间的接口将在7.7节中讨论。外部设备和外部环境的接口超出了本书范围，这里只给出几个简单的例子。

7.1.1 键盘/监视器

计算机与用户交互最常用的方式是键盘/监视器装置。用户通过键盘提供输入，此输入传送到计算机内或在监视器上显示。另外，监视器也显示由计算机提供的数据。

信息交换的基本单位是字符。与每个字符相关的是代码，长度一般为7位或8位。最常用的文本代码是IRA码（International Reference Alphabet，国际参考字母表）[⊖]。在IRA码中，每个字符用一个唯一的7位二进制代码表示，一共可以表示128个不同的字符。字符分为可打印字符和控制字符两种类型。可打印字符包括字母、数字和能打印在纸上或显示在屏幕上的一些特殊字符。一些控制字符能控制字符的打印或显示，如回车符（CR）；另外一些控制字符与通信过程相关。具体内容参看附录F。

⊖ IRA是国际电信联盟（ITU-T）T.50建议书中定义的，曾被称为国际5号字母表（IA5）。IRA的美国版能称为美国标准信息交换码（ASCII）。

对于键盘输入，当用户按下某个键时，键盘首先产生电信号，再由键盘中的转换器解释并转换成与IRA码对应的位模式，然后将该位模式传送到计算机的I/O模块。同样，在计算机中，文本也能以IRA代码来存储。输出时，IRA码字符从I/O模块传送到外部设备。设备中的转换器解释这种代码并发送所需要的电信号到输出设备，再由输出设备来显示字符或执行相应的控制功能。

7.1.2 磁盘驱动器

磁盘驱动器包含两部分的电子部件：一是用于与I/O模块交换数据、控制和状态信号，二是用于控制磁盘的读/写机制。在磁头固定的磁盘中，转换器能够在运动的磁盘表面的磁化模式和设备缓冲器中的位之间进行转换（参见图7-2）。而在可移动磁头的磁盘中，磁臂也必须能迅速在磁盘表面来回移动。

7.2 I/O模块

7.2.1 模块功能

I/O模块的主要功能或需求分为控制和定时、处理器通信、设备通信、数据缓冲、检错几种。

在任何一段时间内，处理器都能根据程序对I/O的要求，非预期地与一个或几个外设进行通信。一些内部资源，如主存和系统总线，必须被包括数据I/O在内的几个功能操作所共享。因此，I/O模块的功能包含**控制和定时**（control and timing）的需求，以协调内部资源和外部设备之间的信息流动。例如，控制从外设到处理器的数据传送包括以下几个步骤：

（1）处理器查询I/O模块，以检验所连接设备的状态。

（2）I/O模块返回设备状态。

（3）如果设备运转正常，并准备就绪，则处理器通过向I/O模块发出一条命令，请求数据传送。

（4）I/O模块获得来自外设的一个数据单元（如8位或16位）。

（5）数据从I/O模块传送到处理器。

如果系统采用总线，则每次处理器和I/O模块之间的交互作用都涉及一次或几次总线仲裁。

前面所述的简化方案也表明，I/O模块必须与处理器以及外设进行通信，**处理器通信**（processor communication）包括：

- **命令译码**：I/O模块接受来自处理器的命令，这些命令一般作为信号发送到控制总线。例如，一个用于磁盘驱动器的I/O模块，可能接受READ SECTOR（读扇区）、WRITE SECTOR（写扇区）、SEEK（寻道）磁道号和SCAN（扫描）记录标识等命令。后两条命令中的每条都包含一个发送到数据总线上的参数。
- **数据**：数据是在处理器和I/O模块间经由数据总线来交换的。
- **状态报告**：由于外设速度很慢，所以知道I/O模块的状态很重要。例如，如果要求一个I/O模块发送数据到处理器（读操作），而该I/O模块仍在处理先前的I/O命令而对此请求未能就绪，则可以用状态信号来报告这个事实。常用的状态信号有忙（BUSY）和就绪（READY），还有报告各种出错情况的信号。
- **地址识别**：正如存储器中每个字对应一个地址一样，每个I/O设备也有地址。因此，I/O模块必须能识别它所控制的每个外设的唯一地址。

另一方面，I/O模块必须能进行**设备通信**（device communication），通信内容包括命令、状态信息和数据（如图7-2所示）。

I/O模块的一个基本功能是**数据缓冲**（data buffering）。由图2-11可以看出实现这一功能的

条件。由于主存和处理器传入、传出数据的速度很快，而许多外设速度要低几个数量级并且范围很宽，所以来自主存的数据通常以高速发送到I/O模块，数据保存在I/O模块的缓冲器中，然后以外设的数据传送速度发送到该外设。而反向传送时，由于数据被缓冲，内存不会被束缚在低速的传送操作中。因此，I/O模块必须既能以设备速度又能以存储器速度传送。类似地，如果外设以高于存储器存取速度的速度操作，则I/O模块也要完成所需的缓冲操作。

最后，I/O模块通常还要负责检错（error detection），并将差错信息报告给处理器。一类差错是设备报告的机械和电路故障（如塞纸、磁道坏）；另一类差错是在信息从设备传送到I/O模块时，数据位发生了变化。对于传输中的差错，经常用一些校验码进行检测。一个简单的例子是在每个数据字符上使用一个奇偶校验位。例如，IRA字符码占据了一个字节中的7位，而第8位被设置为奇偶校验位，以便该字节中"1"的个数为偶数（称为偶校验）或为奇数（称为奇校验）。当I/O模块接收一个字节时，它检查奇偶校验位来确定是否有差错发生。

7.2.2 I/O模块结构

I/O模块在复杂性和控制外设的数目上变化很大，这里只给出大概的描述（一种专门设备，Intel 82C55A，将在7.4节讨论），图7-3给出了I/O模块常用的框图。I/O模块通过一组信号线（如系统总线）连接到计算机的其他部分。传送到I/O模块或从I/O模块传出的数据缓存在一个或几个数据寄存器中，同时也有一个或几个状态寄存器提供当前的状态信息。状态寄存器也能用做控制寄存器，接收来自处理器的具体的控制信息。模块内的逻辑通过一组控制线与处理器交互，处理器使用这些控制线给I/O模块发送命令。控制线中的一部分也可以被I/O模块使用（例如，用于仲裁和传递状态信号）。模块还必须能够识别和产生与其控制的设备相关的地址，每个I/O模块有一个唯一的地址，或者，如果它控制一个以上的外部设备，那么就对应唯一的一组地址。最后，I/O模块还包含与其控制的每个设备进行连接的特定逻辑。

I/O模块提供的多种功能使处理器能以简便的方式管理多种设备。它能隐藏外设的定时、格式、机电结构等细节，使处理器能借助于简单的读和写命令、可能的打开和关闭文件命令对外设进行操作。在最简单的形式中，I/O模块仍可以将许多控制设备的任务（如反绕磁带）留给处理器处理。

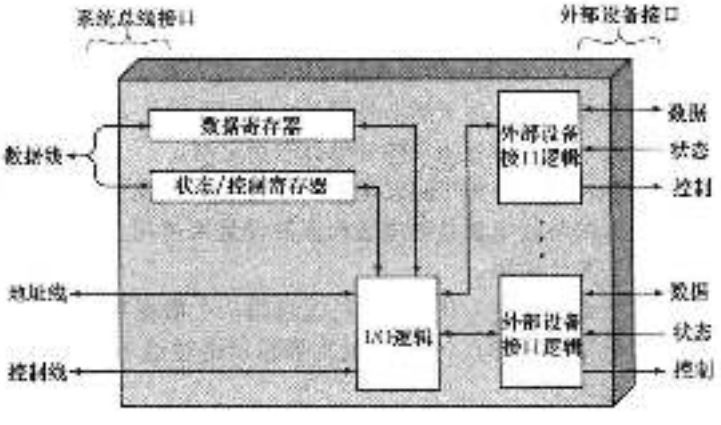


图7-3 I/O模块框图

担负大量详细的处理任务并为处理器提供高级接口的I/O模块，称为I/O通道（I/O channel）或I/O处理器。一种相当基础并需要详细控制的I/O模块通常称为I/O控制器或设备控制器。I/O控制器常用于微型计算机，而I/O通道常用于大型计算机。

下文中，当不混淆结果时，将使用普通术语：I/O模块。如果需要，则采用更特定的术语。

7.3 编程式I/O

I/O操作可采用三种技术。对于编程式I/O（programmed I/O），数据在处理器和I/O模块之间交换，处理器通过执行程序来直接控制I/O操作，包括检测设备状态、发送读或写命令以及传送数据。当处理器发送一条命令到I/O模块时，它必须等待，直到I/O操作完成。如果处理器速

度快于 I/O 模块，那么处理器的时间就白白浪费了。对于中断驱动式 I/O（interrupt-driven I/O），处理器发送一条 I/O 命令后，继续执行其他指令；并且当 I/O 模块完成其工作时，才去中断处理器工作。对于编程式 I/O 和中断式 I/O，都是由处理器来负责在输出时从主存提取数据，在输入时把数据存入主存。另外一种技术被称为直接存储器存取（direct memory access，DMA），在这种模式下，I/O 模块与主存直接交换数据，而不需要处理器的干涉。

表 7-1 列出了这三种技术之间的关系。本节将探讨编程式 I/O，而中断式 I/O 和 DMA 将在接下来的两节中分别加以阐述。

表 7-1 I/O 技术

传送方式	无中断	使用中断
I/O 与存储器之间的传送通过处理器实现	编程式 I/O	中断驱动式 I/O
I/O 与存储器直接传送		直接存储器存取（DMA）

7.3.1 编程式 I/O 概述

当处理器在执行程序的过程中遇到了一条与 I/O 操作有关的指令时，它通过发送命令到适当的 I/O 模块来执行这条指令。对于编程式 I/O，I/O 模块将执行所要求的动作，然后在 I/O 状态寄存器中设置适当的一些位（如图 7-3 所示）。I/O 模块不采用进一步的动作来通知处理器，特别是它不去中断处理器。因此，处理器就需要周期性地检查 I/O 模块的状态，直到发现该操作完成。

为了解释编程式 I/O 技术，我们首先从处理器发送给 I/O 模块的 I/O 命令讨论，然后再讨论处理器执行的 I/O 指令。

7.3.2 I/O 命令

为了执行与 I/O 相关的指令，处理器发送一个指定具体 I/O 模块和外设的地址，并发送一条 I/O 命令。当 I/O 模块被处理器寻址时，它可能会接收如下四种类型的 I/O 命令：

- **控制命令**：用于激活外设并告诉它要做什么。例如，可以指示磁带机快退或快进一个记录。这些命令为具体外设类型而定制。
- **测试命令**：用于测试与 I/O 模块及其外设相关的各种状态条件。处理器想要知道，感兴趣的外设电源是否接通和该外设是否可用。它还想知道，最近的 I/O 操作是否完成，是否发生差错。
- **读命令**：使 I/O 模块从外设获得一个数据项，并把它存入内部缓冲区（图 7-3 描述一个数据寄存器）。然后，处理器可以通过请求 I/O 模块把数据传送到数据总线以获得该数据项。
- **写命令**：使 I/O 模块从数据总线获得一个数据项（字节或字），然后把它传送到外设。

图 7-4a 给出了一个使用编程式 I/O 从外设读取数据（例如磁带上的一个记录）到内存的例子。每次读一个字（如 16 位）的数据。对于每个读入的字，处理器必须停留在状态监测周期，直到确定这个字在 I/O 模块的数据寄存器中有效为止。这个流程图突出了这种技术的主要缺点，它是一个使处理器一直处于不必要的忙碌状态的耗时过程。

7.3.3 I/O 指令

对于编程式 I/O，在处理器从内存获取的 I/O 指令与为执行此指令处理器发送到 I/O 模块的 I/O 命令之间存在着紧密的对应关系。也就是，该指令很容易被映射成 I/O 命令，并且二者之间通常是简单的一一对应关系。指令的形式取决于外设寻址的方式。

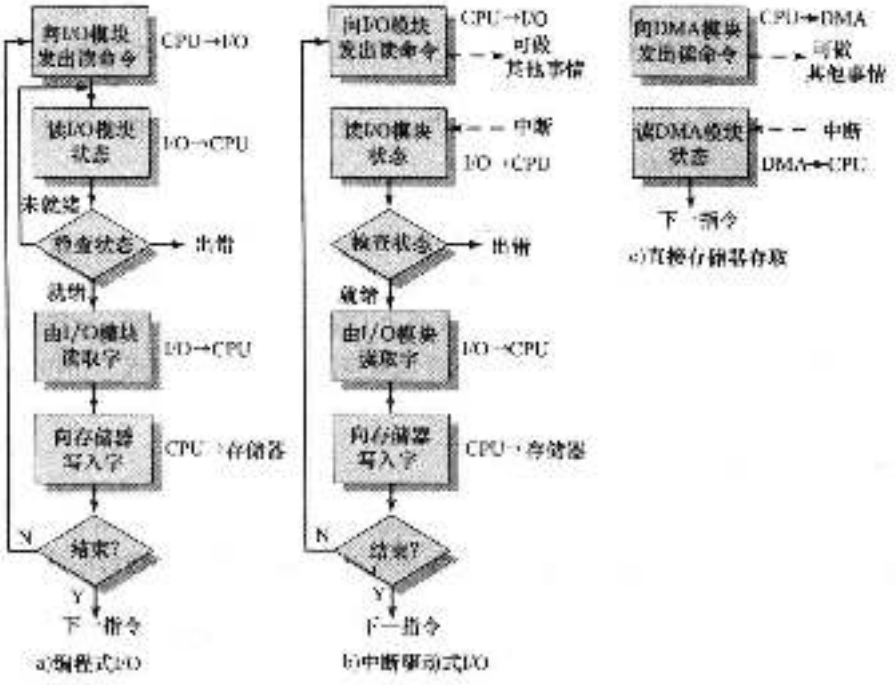
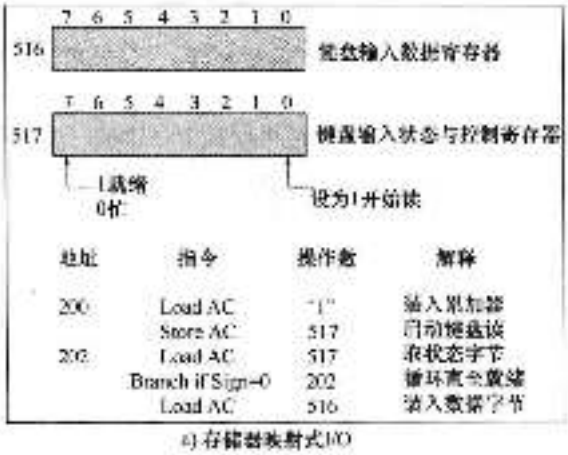


图7-4 数据块输入的三种技术

通常，有多个I/O设备通过I/O模块连接到系统，每个设备有一个唯一的标识符或地址。当处理器发送I/O命令时，该命令就包括所需设备的地址。因此，每个I/O模块必须对地址线译码，确定命令是否是发送给自己的。

当处理器、主存和I/O共享一条公共总线时，有两种可能的编址方式：**存储器映射式**（memory-mapped）和**分离式**（isolated）。对于**存储器映射式I/O**，存储单元和I/O设备有单一的地址空间。处理器将I/O模块的状态和数据寄存器看成存储单元一样对待，使用相同的机器指令来访问存储器和I/O设备。例如，使用10根地址线，可以组合成总数为$2^{10}=1024$个存储单元和I/O地址。

存储器映射式I/O在总线上只需要单一的读线和单一的写线。另一种方式是，让总线既有存储器的读线和写线，同时也有输入和输出命令线。此时，命令线指定该地址是说明存储单元还是说明I/O设备的。整个地址范围对两者都适用。再来看上述带有10根地址线的例子，系统现在不仅支持1024个存储单元，也支持1024个I/O地址。因为I/O的地址空间与存储器的地址空间是分离的，因此这被称为**分离式I/O**。



a)存储器映射式I/O

地址	指令	操作数	解释
200	Load AC	"1"	输入初始化
	Store AC	517	启动键盘读
202	Load AC	517	取状态字节
	Branch if Sign=0	202	循环直至就绪
	Load AC	516	输入数据字节

b)分离式I/O

图7-5 存储器映射式I/O和分离式I/O

图7-5对比了这两种编程式I/O技术。图7-5a表示键盘终端之类的简单输入设备怎样采

用存储器映射 I/O 方式来编程。假设一个 10 位的地址，有 512 个存储单元（单元 0 - 511），最多有 512 个 I/O 地址（单元 512 ~ 1023），两个地址专用于从特定终端连到键盘输入，地址 516 代表数据寄存器，地址 517 代表状态寄存器，状态寄存器也可用做控制寄存器来接收处理器的命令。图中的程序表示从键盘读入一个字节的数据传送到处理器的累加寄存器中的过程。注意，处理器一直循环检测到数据字节有效为止。

对于分离式 I/O（如图 7-5b 所示），I/O 端口只有被特定的 I/O 命令访问，这种命令能激活总线上的 I/O 命令线。

对于大部分类型的处理器，有相对多的一组不同指令用于访问存储器。而如果采用分离式 I/O，则只有少数几种 I/O 指令。因此，存储器映射式 I/O 的优点是能使用大的指令系统，这样可进行更有效的编程；其缺点是占用了宝贵的内存地址空间。存储器映射式 I/O 和分离式 I/O 都很常用。

7.4 中断驱动式 I/O

编程式 I/O 存在的问题是，处理器必须为 I/O 模块准备接收或传送数据等待很长一段时间。在等待的过程中，处理器必须不断地询问 I/O 模块的状态，因此使整个系统性能严重下降。

另外一种方法是，处理器发送一个 I/O 命令到模块，然后去处理其他有用的工作。当 I/O 模块准备和处理器交换数据时，它中断处理器以请求服务。然后，处理器执行数据传送，最后恢复它原先的处理工作。

让我们来考虑这是如何工作的。首先从 I/O 模块的角度来看，对于输入，I/O 模块接收来自处理器的 READ 命令，然后从相关的外设中读入数据。一旦数据进入 I/O 模块的数据寄存器后，该模块通过控制总线给处理器发送中断信号，然后等待，直到处理器请求该数据时为止。当处理器有数据请求时，I/O 模块把数据送到数据总线上，并准备另一个 I/O 操作。

从处理器的角度来看，输入的行为如下。首先，处理器发送一个 READ 命令，然后它离开去处理其他的事情（例如，处理器可以同时处理几个不同的程序）。在每个指令周期结束时，处理器检查中断（如图 3-9 所示）。当来自 I/O 模块的中断出现时，处理器保存当前程序的现场（例如，程序计数器和处理器寄存器），并处理该中断。此时，处理器从 I/O 模块读取数据字并保存到主存中。然后恢复刚才主在运行的程序（或其他程序）的现场，并继续运行原来的程序。

图 7-4b 表示了使用中断式 I/O 读取一个数据块的情形。与图 7-4a 相比，中断式 I/O 比编程式 I/O 效率高，因为它消除了空闲的等待。然而，中断式 I/O 仍然要耗费处理器很多的时间，因为从存储器到 I/O 模块或从 I/O 模块到存储器的每个数据字都必须经过处理器传送。

7.4.1 中断处理

让我们更详细地考虑中断式 I/O 时处理器的作用。中断的出现会触发一系列处理器软硬件中的事件，图 7-6 给出了一个典型的序列。当 I/O 设备完成一次 I/O 操作时，下列硬件事件序列会发生：

（1）设备给处理器发送一个中断信号。

（2）处理器在响应该中断之前完成当前指令的执行，如图 3-9 所示。

（3）处理器检测中断，确定中断源，并发送一个确认信号给发送中断的设备，允许设备取消中断信号。

（4）处理器需要准备传送控制给中断例程。首先，它需要保存将来在中断点恢复当前程序所需要的信息。所需要的最小信息是：（a）处理器状态，它包含在一个被称为程序状态字（PSW）的寄存器中；（b）下一条将被执行的指令的位置，它包含在程序计数器中。这些信息都

可以压入系统控制栈[⊖]中保存。

（5）处理器将响应该中断的中断处理程序的入口地址装入程序计数器。中断处理程序的个数取决于计算机的体系结构和操作系统的设计，可能是单一一个程序，也可能是每个中断类型对应一个程序，或者每个设备和每个中断类型对应一个程序。如果有一个以上的中断处理程序，处理器必须确定调用哪个程序。而这一信息可能已经包含在原先的中断请求信号中，或者处理器发送一个请求到提出中断的设备，以得到包含所需信息的响应。

一旦程序计数器装入后，处理器进入下一个指令周期，并始取指阶段，因为取指令由程序计数器中的内容决定，结果是控制权转移到了中断处理程序。中断处理程序的执行包含下列几步操作：

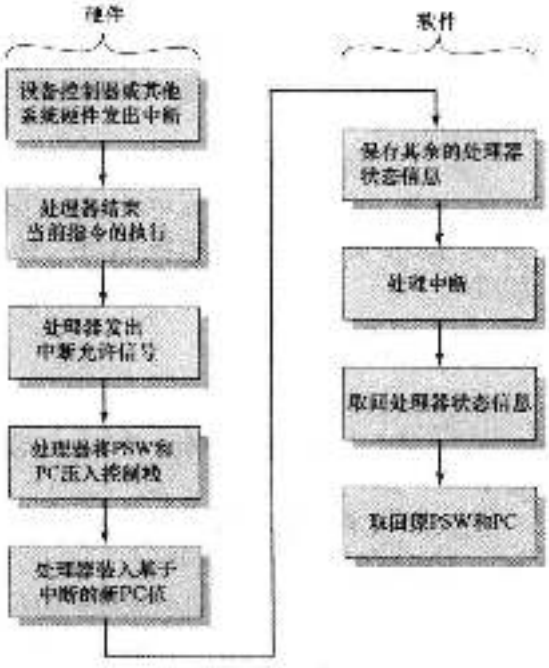


图7-6 简单中断处理

（6）此时，被中断程序的程序计数器和PSW已存入系统堆栈。然而，还有一些执行程序状态的其他信息要考虑，特别是，处理器中寄存器的内容也需要保存，因为这些寄存器可能被中断处理程序使用。因此，所有这些值，以及任何其他状态信息，都需要保存。通常，中断处理程序将从保存所有寄存器的内容入栈开始。图7-7a给出了一个简单的例子。此例中，用户程序在位置N的指令后被中断，所有寄存器内容和下一条指令地址（$N+1$）全部进栈。堆栈指针指向新的栈顶，程序计数器被更新为指向中断服务程序的入口地址。

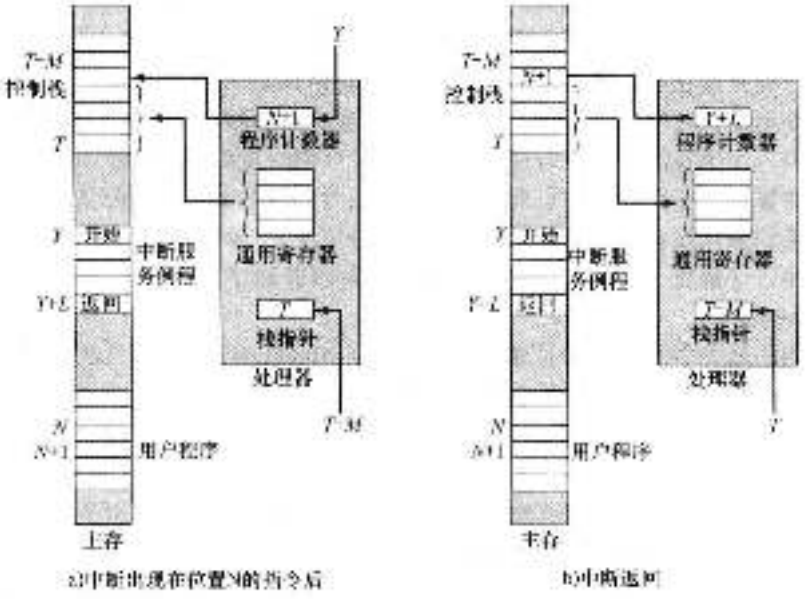


图7-7 中断时存储器和寄存器的变化

⊖ 参见附录10A中关于堆栈操作的讨论。

(7) 中断处理程序接下来处理中断，包括检查I/O操作或引起中断的其他事件的状态信息。可能还包括发送附加命令或确认信息给I/O设备。

(8) 当中断处理完成时，被保存的寄存器值出栈并恢复到原寄存器中（如图7-7b所示）。

(9) 最后的工作是出栈恢复PWS和程序计数器的值。于是，将要执行的下一条指令来自于被中断的原程序。

注意，保存所有被中断原程序的有关状态信息对后面的恢复工作很重要，这是因为中断不是原程序所调用的例程，中断在用户程序执行的任何时刻和任何位置都可能发生，它的出现无法预定。甚至，如在下一章我们将看到的一样，这两个程序可能完全不同，甚至可能属于两个不同的用户。

7.4.2 设计问题

实现中断I/O操作会出现两个设计问题。第一，因为几乎总是有多个I/O模块，所以处理器如何确定是哪个设备发生了中断？第二，如果有多个中断出现，处理器如何确定优先处理哪个中断？

首先，我们来考虑设备识别，设备识别通常有多条中断线、软件轮询、菊花链（硬件轮询，向量）、总线仲裁（向量）4种技术。

解决问题最直接的方法是在处理器和I/O模块之间提供多条中断线（multiple interrupt line）。然而，将较多的总线或处理器引脚用做中断线是不实际的，因此，即使有多条中断线可用，每条线也必须连接多个模块。在每根线上还要采用其他3种技术中的一种。

另一种技术是**软件轮询**（software poll），当处理器检测到一个中断时，进入中断服务程序，这个程序的任务是轮询每一个I/O模块来确定是哪个模块产生的中断。轮询可以采用单独的命令线形式（如TEST I/O）。这时，处理器启动TEST I/O命令线，并将该I/O模块的地址送到地址线上。如果是该I/O模块发出的中断，则它肯定会响应以得到相应的响应。另一种方法是每个I/O模块包含一个可寻址的状态寄存器，处理器通过从每个I/O模块的状态寄存器中读取信息来识别中断模块。一旦识别正确的模块，处理器就转向该设备的设备服务例程。

软件轮询的缺点是费时。相比之下，更为有效的方法是使用**菊花链**（daisy chain）电路，实际上，它提供一种硬件轮询。图3-26给出了一个菊花链配置的例子。对于中断，所有的I/O模块共享一条中断请求线，中断应答线采用菊花链穿过这些中断模块。当处理器检测到有中断请求产生，就发出一个应答信号，此信号穿过一系列I/O模块直到请求中断的模块。而请求中断的模块通常通过放置一个字在数据线上来响应此应答信号。这个字被称为向量，它或者是I/O模块的地址，或者是识别此设备的唯一标识符。在任一种情况，处理器用这个向量作为指针指向相应的设备服务程序，这避免了首先执行一个常规的中断服务程序的需要。这种技术称为**向量式中断**（vectored interrupt）。

另一种使用向量式中断的技术是**总线仲裁**（bus arbitration）。使用总线仲裁技术，I/O模块在发出中断请求前必须首先获得总线控制权，因此，一次只有一个模块能占用总线。当处理器检测到有中断请求时，它发中断应答信号响应中断。然后，请求中断的模块将其向量放在数据线上。

以上技术用于识别I/O中断请求模块。当一个以上的设备请求中断服务时，这些技术还提供一种分配优先级的方法。对于多条中断线的方式，处理器仅仅挑选具有最高优先级的中断线。对于软件轮询方式，模块的轮询次序就决定了模块的优先级。类似地，菊花链上的模块次序也决定了模块的优先级。最后，总线仲裁可以采用优先级方案，这已在3.4节中讨论过。

现在我们来看两个中断结构的例子。

7.4.3 Intel 82C59A 中断控制器

Intel 80386 提供了单一的中断请求（INTR）线和单一的中断应答（INTA）线。为了使80386灵活地处理各种设备和优先级结构，它通常配有外部的中断控制器——82C59A。外设连接到82C59A，82C59A再连接到80386。

图7-8显示了采用82C59A连接多个I/O模块到80386的情形。一个82C59A最多能处理8个I/O模块，如果需要控制8个以上的模块，则使用级连方式，最多能处理64个模块。

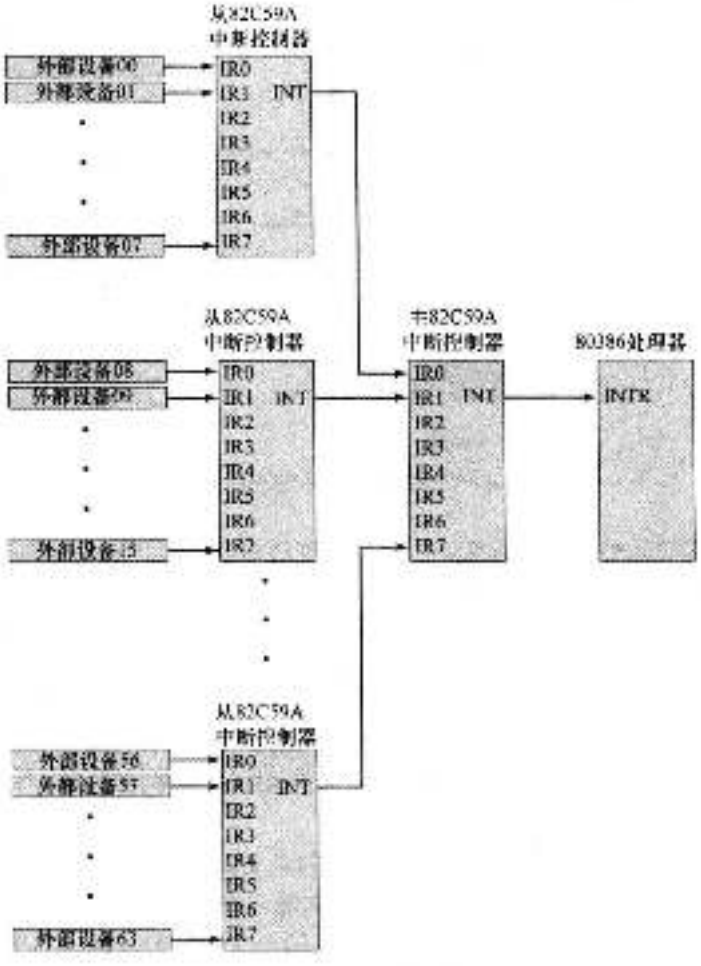


图7-8 82C59A中断控制器的使用

82C59A的唯一职责是管理中断。它从连接的模块中接收中断请求，并确定那个中断的优先级最高，然后通过INTR线发出请求信号给处理器。处理器通过INTA线应答。这就提示82C59A将对应的向量信息放到数据总线上。然后，处理器可以开始处理中断，并直接与I/O模块进行通信，读或写数据。

82C59A是可编程的，80386通过设置82C59A中的控制字来决定其优先级方式。下面是可用的中断模式：

- **全嵌套**：中断请求按优先级从0（IR0）到7（IR7）排序。
- **轮转**：在有些应用中，多个中断设备具有相同的优先级，这时，刚获得服务的设备在本组中具有最低的优先级。

- 特殊屏蔽：它允许处理器有选择地禁止来自其他设备的中断。

7.4.4 Intel 82C55A 可编程外部接口

作为一个用于编程式 I/O 和中断驱动式 I/O 的 I/O 模块的例子，我们考虑 Intel 82C55A 可编程外部接口。82C55A 是一种单芯片的通用 I/O 模块，与 Intel 80386 处理器一起使用。图 7-9 表示了一个通用的框图和其 40 个引脚封装的引脚分配。

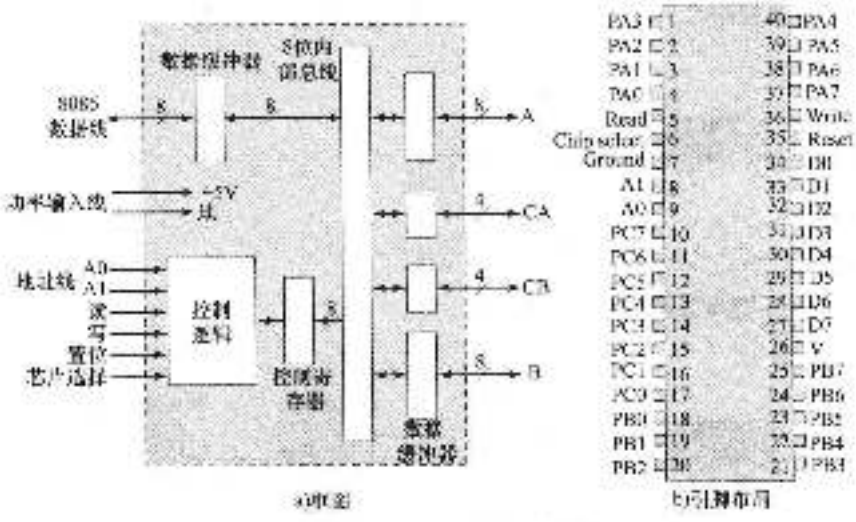


图 7-9 Intel 82C55A 可编程外部接口

该框图的右边是 82C55A 的外部接口。24 条 I/O 线由 80386 利用控制寄存器编程，80386 能通过设置控制寄存器的值来指定各种操作方式和配置。24 根线划分成 3 个 8 位组（A，B，C），每个组可以作为一个 8 位的 I/O 端口。另外，组 C 又细分成两个 4 位的组（C_A 和 C_B），它们可以与 A 和 B 的 I/O 端口合并使用。以这种方式配置时，组 C 的线可以传送控制信号和状态信号。

该框图的左边是与 80386 总线的内部接口，它包括一个 8 位的双向的数据总线（D0～D7），用于与 I/O 端口进行数据传输，和传送控制信息到控制寄存器。两根地址线指定 3 个 I/O 端口中的一个或者是控制寄存器。当芯片选择（CS）信号和 READ 或 WRITE 信号有效时，可以进行数据传输。RESET 信号用于初始化 82C55A 芯片。

由处理器装载的控制寄存器用来控制操作模式和定义信号。以模式 0 操作时，3 组 8 位的外部线作为 3 个 8 位的 I/O 端口，每个端口能指定作为输入或输出。否则，组 A 和组 B 作为 I/O 端口，而组 C 的线作为 A 和 B 的控制线。控制信号有两个主要目的：握手和中断请求。握手是一种简单的时序机制。一条控制线由发送器作为“数据就绪”（DATA READY）线使用，指示数据已传到 I/O 数据线上。另一条线由接收器用于“应答”（ACKNOWLEDGE），指示数据已经读取，数据线可以清除。另一条线可以指定为“中断请求”（INTERRUPT REQUEST）线，并连接到系统总线。

因为 82C55A 可以通过控制寄存器进行编程，所以它能够用于控制各种简单的外设。图 7-10 说明了它用于控制键盘/显示器终端。该键盘提供 8 位的输入，这些位中的两位，SHIFT 和 CONTROL，对处理器执行的键盘处理程序有具体的含义。然而，这个含义对 82C55A 是透明的，82C55A 只是简单地接收 8 位数据，并将它们送到系统的数据线上。两根握手控制引脚线提供给键盘使用。

显示器也连到 8 位数据端口，且其中两位有特殊含义，它们对 82C55A 也是透明的。除两个握手信号线外，另外两根线提供附加的控制功能。

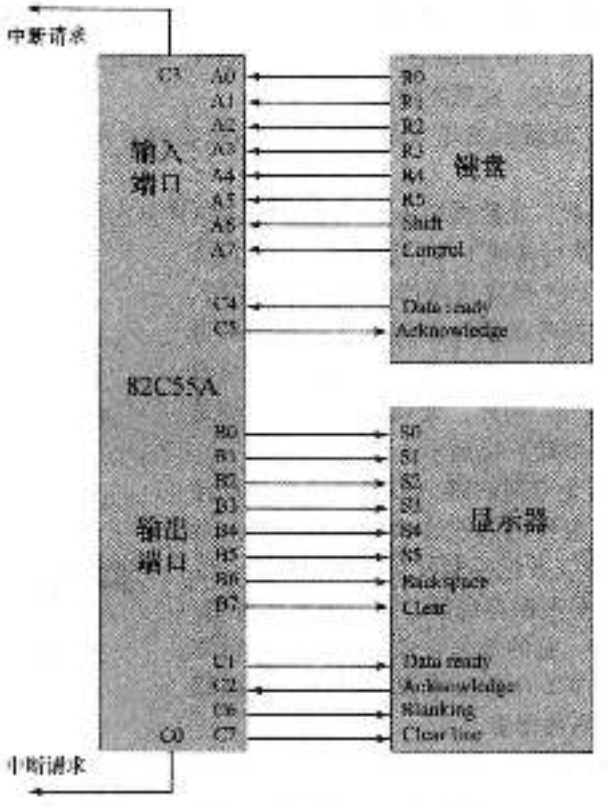


图 7-10 键盘/显示器与 82C55A 的接口

7.5 直接存储器存取

7.5.1 编程式 I/O 和中断驱动式 I/O 的缺点

尽管中断驱动式 I/O 比简单的编程式 I/O 有效，但它仍需要处理器及时干预才能在存储器与 I/O 模块之间进行数据传送，并且任何数据传送都必须穿过处理器的通路。因此，这两种 I/O 方式均存在以下两点不足：

（1）I/O 传送速度受处理器测试和服务设备速度的限制。

（2）处理器负责管理 I/O 传送，对于每一次的 I/O 传送，处理器必须执行很多指令（如图 7-5 所示）。

在这两点不足之间存在着一些平衡。考虑一个数据块的传送，当采用简单的编程式 I/O 时，处理器专门用来处理 I/O 任务，并以相当快的速度传送数据，其代价是处理器不做其他事情。而采用中断式 I/O 时，处理器在某种程度上减少了干预活动，但 I/O 传输率却降低了。不管怎样，这两种方式给处理器的利用率和 I/O 的传输率都带来了不利的影响。

当需要传送大量的数据时，必须采用一种更加有效的技术：直接存储器存取（DMA）。

7.5.2 DMA 功能

DMA 在系统总线上增加了一个模块，该 DMA 模块（如图 7-11 所示）能够模仿处理器，并且确实从处理器那里接管了系统控制的工作。它需要通过控制系统总线来管理从存储器输出或输入存储器的数据。为此，DMA 模块必须只在处理器不需要总线时占用系统总线，或者必须强制处理器暂时挂起。后一种技术更加通用，并称为周期窃取（cycle-stealing），即 DMA 模块有效地窃取一个总线周期。

当处理器希望读或写数据时，它发送一个命令给 DMA 模块，向 DMA 模块发送如下信息：

- 通过使用处理器与 DMA 模块之间的读或写控制线，说明需要的是读还是写操作。
- 相应的 I/O 设备地址，经数据线传输。
- 读或写操作的存储器起始单元地址，经数据线传输，并被 DMA 模块存入其地址寄存器中。
- 读或写操作的字数，经数据线传输，并被 DMA 模块存入其数据计数寄存器中。

然后，处理器继续执行其他工作，它已将该 I/O 操作委派给 DMA 模块。而 DMA 模块负责传送全部的数据块，每次一个字，直接将数据传送到存储器或从存储器中读出，不经过处理器。当该数据传送完成时，DMA 模块给处理器发送一个中断信号。因此，处理器只在数据传送的开始和结束时参与（如图 7-4c 所示）。

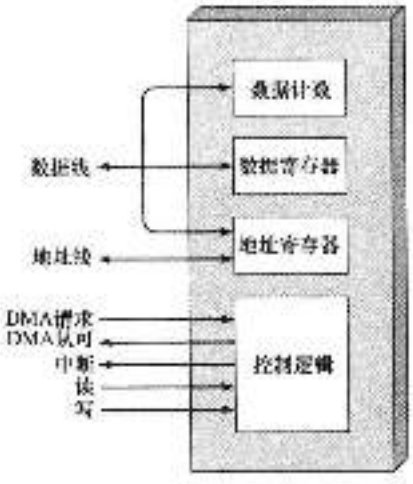


图 7-11 典型的 DMA 框图

图 7-12 显示了在指令周期的哪个位置处理器可以挂起。每次，仅仅在它需要使用总线之前挂起处理器。然后 DMA 模块传送一个字，并把控制权交还给处理器。注意，这不是中断，处理器不保存现场，也不做其他事情，而是等待一个总线周期。总的效果是使处理器的执行速度下降，但是对于多字节 I/O 传送来说，DMA 比中断驱动式 I/O 和编程式 I/O 有效得多。

DMA 的配置机制有多种方式，几种可能的配置如图 7-13 所示。在第一个例子中，所有的模块共享系统总线。DMA 模块作为处理器的代理，采用编程式 I/O，在存储器与 I/O 模块之间通过 DMA 模块交换数据。虽然这种配置价格便宜，但其效率很低。如同处理器控制的编程式 I/O，每传送一个字要消耗两个总线周期。

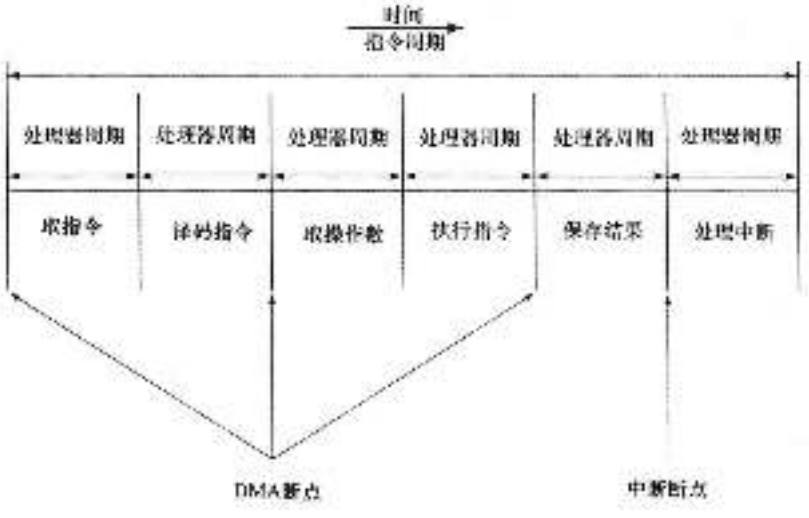


图 7-12 指令周期中的 DMA 和中断断点

通过集成 DMA 和 I/O 的功能，可以减少所需要的总线周期数。如图 7-13b 所示，DMA 模块与一个或多个 I/O 模块之间有一条路径（不包括系统总线）。DMA 逻辑实际上可能是 I/O 模块的一部分，也可能是控制一个或几个 I/O 模块的独立模块。这个概念可以进一步扩充为通过一条 I/O 总线将 I/O 模块连到 DMA 模块，如图 7-13c 所示。这样可以减少 DMA 模块连接的 I/O 接口数，

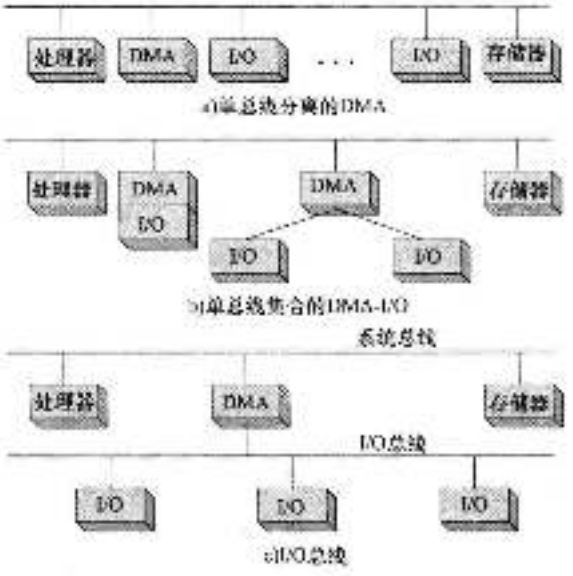


图 7-13 可能的 DMA 配置

并使系统容易扩展。在后两种情况中（图 7.3b 和图 7-3c），DMA 模块和处理器、存储器共享的系统总线只在与存储器交换数据时才由 DMA 模块使用，DMA 与 I/O 模块之间的数据交换不在系统总线上进行。

7.5.3 Intel 8237A DMA 控制器

Intel 8237A DMA 控制器是 80x86 系列处理器和 DRAM 存储器之间的接口，用以提供 DMA 能力。图 7-14 指出了 DMA 模块所处的位置。当 DMA 模块需要使用系统总线（数据、地址或控制总线）来传送数据时，它发出 HOLD（保持请求）信号给处理器。处理器发出 HLDA（保持确认）信号来响应，告诉 DMA 模块能使用系统总线了。例如，若该 DMA 模块要将一块数据由存储器传送到磁盘，则要做如下工作：

（1）外围设备（如磁盘控制器）通过置 DREQ（DMA 请求）信号为高电平来请求 DMA 服务。

（2）DMA 模块置 HRQ（保持请求）为高电平，通过 CPU 的 HOLD 引脚通知 CPU 它需要使

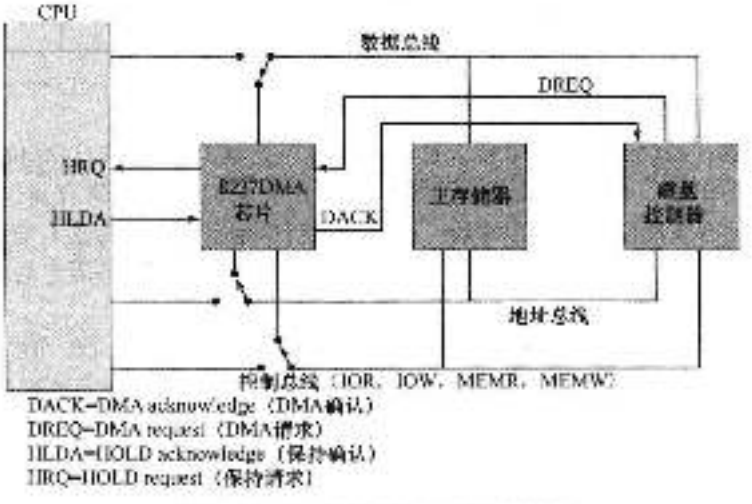


图 7-14 8237 DMA 的系统总线使用

用总线。

(3) CPU 完成当前的总线周期（不必是当前的指令周期）后，置 HLDA（保持确认）有效响应 DMA 请求，告诉 8237DMA 现在可以使用总线执行其任务。DMA 在执行其任务期间，HOLD 信号必须保持有效。

(4) DMA 模块启动 DACK（DMA 认可信号），告诉外设它将开始传送数据。

(5) DMA 模块开始将数据从存储器传送到外围设备。它将数据块的首字节地址放到地址总线上，并启动 MEMR 信号，从而读取存储器的字节数据放到数据总线上。然后，启动 IOW 操作，将数据写入外设。之后，DMA 模块的字节计数器减 1 而地址指针加 1，DMA 重复上述传送过程直到计数器减为 0，任务结束。

(6) DMA 模块结束它的任务后，HRQ 失效，通知 CPU 可以重新获得对总线的控制。

当 DMA 模块占用总线传送数据时，处理器是空闲的。类似地，当 CPU 占用总线时，DMA 模块是空闲的。8237 DMA 被称为飞越式（fly-by）DMA 控制器。这表示数据从一个主存单元传输到另一个主存单元时不需要经由 DMA 芯片，也不需要将数据存储到 DMA 芯片中。因此，DMA 只能在 I/O 端口与存储器之间传送数据，而不能在两个 I/O 端口之间或两个存储器地址之间传送数据。然而，如下所述，DMA 芯片能通过寄存器实现存储器到存储器的数据传送。

8237 包含 4 个可独立编程的 DMA 通道，每个通道可在任何时候启用，这些通道的编号为 0、1、2 和 3。

8327 有四组 5 个控制/命令寄存器，每个通道一组，用于编程和控制 DMA 操作（如表 7-2 所示）。

表 7-2 Intel 8237 A 寄存器

位	命令	状态	模式	单屏蔽	全屏蔽
D0	存储器到存储器 E/D	通道 0 已达到 TC	通道选择	选择通道屏蔽位	清除/设置通道 0 屏蔽位
D1	通道 0 地址保持 E/D	通道 1 已达到 TC			清除/设置通道 1 屏蔽位
D2	控制器 E/D	通道 2 已达到 TC	校验/写/读传送	清除/设置屏蔽位	清除/设置通道 2 屏蔽位
D3	正常/压缩时序	通道 3 已达到 TC		不使用	清除/设置通道 3 屏蔽位
D4	固定/轮转优先权	通道 0 请求	自动初始 E/D		
D5	滞后/扩展写选择	通道 0 请求	地址增/减选择		不使用
D6	DREQ 有效电平为高/低	通道 0 请求			
D7	DACK 有效电平为高/低	通道 0 请求	需求/单一/块/级联模式选择		

注：E/D = enable/disable（允许/禁止）
TC = terminal count（终止计数）

- **命令寄存器**：处理器将命令字装入这个寄存器以控制 DMA 操作，D0 位允许存储器到存储器的数据传送。此时，通道 0 用于将字节传送到 8237 的暂存寄存器，而通道 1 用于将此字节从暂存寄存器传送到存储器。当存储器到存储器的数据传送有效时，D1 能用于通道 0 的地址保持（禁止地址增或减），从而使写入存储器的值是固定的。D2 位用于启用或禁止 DMA。
- **状态寄存器**：处理器通过读取这个寄存器来确定 DMA 的状态。D0 ~ D3 位被用来指示通道 0 ~ 3 是否已达到终止计数（TC）。D4 ~ D7 位被处理器用来指示通道 0 ~ 3 是否有 DMA 请求未解决。
- **模式寄存器**：处理器设置此寄存器来决定 DMA 的操作模式。D0 和 D1 位用于选择通道，其他位用于指定被选通道的操作模式。D2 和 D3 确定数据是从 I/O 设备到存储器（写操作）还是从存储器到 I/O 设备（读操作），或者是一个校验操作。若 D4 位有效，则 DMA

传送结束时存储器地址寄存器和计数器重新装入初始值。D6 和 D7 位指定 8237 的使用方式，其中单一模式传送字节，块模式和需求模式用于传送数据块，需求模式还允许传送过程提前结束。级联模式允许多个 8237 级联，使通道数多于 4。

- **单屏蔽寄存器**：处理器设置此寄存器。D0 和 D1 位选择通道，D2 位用于清除或设置该通道的屏蔽位。通过这个寄存器指定某一通道的 DREQ 输入被屏蔽或允许。命令寄存器能使整个 DMA 芯片无效，而单屏蔽寄存器允许程序员对某一通道设置有效或无效。
- **全屏蔽寄存器**：它类似于单屏蔽寄存器，与之不同的是，它可以通过一次写操作屏蔽或启用四个通道。

此外，8237 A 还有 8 个数据寄存器：每个通道有一个存储器地址寄存器和一个计数寄存器。处理器设置这些寄存器来表示传送数据的主存地址和存储块大小。

7.6 I/O 通道和处理器

7.6.1 I/O 功能的演变

随着计算机系统的发展，单个部件的复杂性不断提高，这一变化明显地体现在 I/O 功能上，我们已经看到了部分演变，其演变步骤可以归纳如下：

（1）CPU 直接控制外设，这主要用于简单的微处理器控制设备。

（2）增加控制器或 I/O 模块，处理器使用编程式 I/O 而不是中断，使处理器从外设的特殊细节中解脱出来。

（3）采用与 2 相同的配置，但使用了中断，处理器不需要浪费时间等待 I/O 操作完成，提高了处理器的工作效率。

（4）I/O 模块通过 DMA 直接存取存储器，传输数据不需要处理器的参与，除了在传输的开始和结束时参与以外。

（5）I/O 模块成为有自主控制权的处理器，有处理 I/O 的专用指令集。CPU 指示 I/O 处理器执行存储器中的 I/O 程序，I/O 处理器不需要 CPU 的干涉就能获取和执行 I/O 指令。这允许 CPU 指派一系列 I/O 活动，并只在整个活动执行完成后才中断 CPU。

（6）I/O 模块带有其局部存储器，成为一台自治的计算机。这种结构可以控制大量的 I/O 设备而最小化了 CPU 的干涉，它常用于与交互式终端进行通信。I/O 处理器负责大部分任务，包括控制终端。

从上面的演变过程可以看出，越来越多的 I/O 功能在执行时不需 CPU 参与，CPU 日益从 I/O 相关工作中解放出来，改善了系统性能。最后两步（（5）和（6））的主要改变体现在引入了 I/O模块能够执行程序的概念。对于第（5）步，该 I/O 模块常称为 I/O 通道（I/O channel）；而在第（6）步中常常使用术语 I/O 处理器（I/O processor）。然而，这两个术语偶尔才用于两种情况。在下面的介绍中，我们将统一使用 I/O 通道这一术语。

7.6.2 I/O 通道的特点

I/O 通道是 DMA 概念的扩充。I/O 通道可以执行 I/O 指令来控制 I/O 操作，此时，CPU 不执行 I/O 指令，这些指令存储在主存中，由 I/O 通道本身的一个专用处理器执行。因此，CPU 通过请求 I/O 通道执行存储器中的程序来启动一次 I/O 数据传送，程序将指定一个或几个设备，一块或几块存储器区域、优先级以及出错时的处理行为，而 I/O 通道执行这些指令来控制数据传送。

I/O 通道通常有两种类型，如图 7-15 所示。第一类是选择通道，它控制多个高速设备，并且每次只与其中的一个设备进行数据传送，即 I/O 通道选择一个设备，并有效地进行数据传送。每个设备或一小组设备由控制器或 I/O 模块管理，因此，I/O 通道代替 CPU 控制这些 I/O 控制器。第二类是多路通道，它能够同时处理多个设备的 I/O 操作。对于低速设备，字节多路选择器可以很快地

接收和传送数据到多个设备。例如，来自三个设备的字符流分别是 $A_1A_2A_3A_4\cdots$，$B_1B_2B_3B_4\cdots$ 和 $C_1C_2C_3C_4\cdots$，由于各字符流速度的不同，综合的字符流可以是 $A_1B_1C_1A_2C_2A_3B_2C_3A_4$ 等等。对于高速设备，一种称为块多路选择器可以交叉存取来自多个设备的数据块。

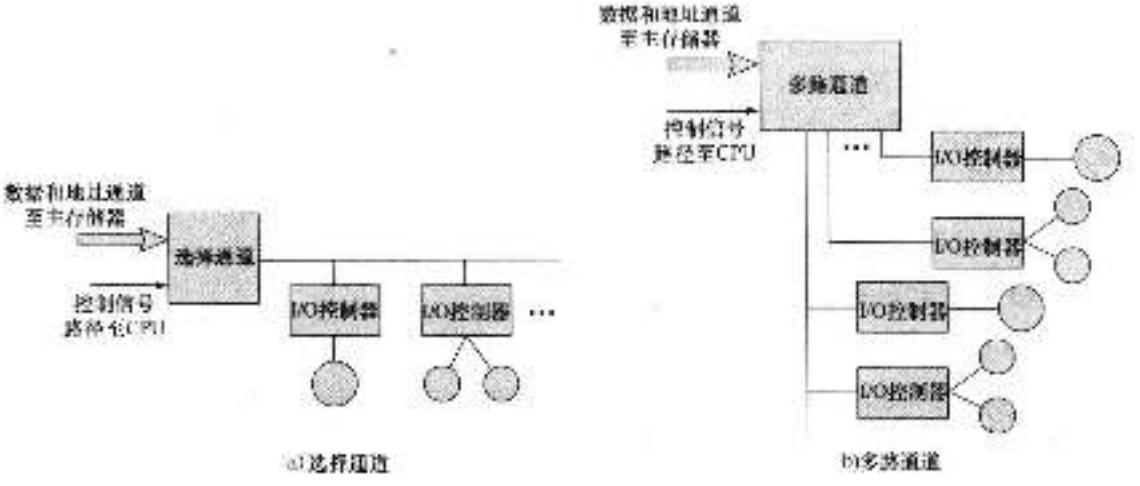


图 7-15 I/O 通道体系结构

7.7 外部接口：FireWire 和 InfiniBand

7.7.1 接口类型

I/O 模块与外设的接口是根据外设的性质和操作进行设计的，接口的一个主要特性是串行传输或者并行传输（如图 7-16 所示）。在并行接口中，有多根线连接 I/O 模块和外设，同时传送多位，正如一个字的所有位在数据总线上同时传输一样。在串行接口中，只有一根线用于传送数据，每次只传输一位。通常，并行接口用于高速外设，如磁带和磁盘，而串行接口通常用于打印机和终端。随着新一代高速串行接口的出现，并行接口变得不大普遍了。

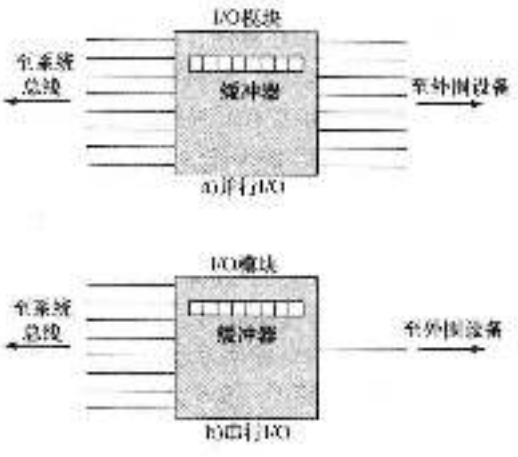


图 7-16 并行 I/O 和串行 I/O

在任一种情况下，I/O 模块必须与外设进行对话。通常情况下，写操作的对话过程如下：

（1）I/O 模块发送控制信号，请求发送数据。

（2）外设响应该请求。

（3）I/O 模块传送数据（一次传送一个字或一个数据块，取决于外设）。

（4）外设确认接收到数据。

读操作的对话过程与此类似。

I/O 模块操作的关键是内部缓冲器，它能暂存在外设与系统之间传输的数据，以平衡系统总线与其外部导线的速度不匹配。

7.7.2 点对点和多点配置

在计算机系统中，I/O 模块和外设的连接可以采用点对点或多点方式。点对点方式为 I/O 模块和外设之间提供了一条专用线。小型系统（PC 机或工作站）中，一般点对点连接包括键盘、打印机和外部调制解调器。典型的例子是 EIA-232 规范（参见［STAL07］）。

多点外部接口越来越重要，它用于支持外部海量存储设备（如磁盘和磁带机）和多媒体设备（CD-ROM、视频和音频设备）。这些多点接口在外部总线上有效，它们展示了与第3章中讨论的总线同类型的逻辑。本节将介绍两个关键的例子：FireWire 和 InfiniBand。

7.7.3 FireWire 串行总线

随着处理器速度达到 GHz 范畴和存储设备具有多个千兆位，个人计算机、工作站及服务器对 I/O 的要求越来越高，而在大型机及超级计算机系统中开发的高速 I/O 通道技术对于这些较小的计算机系统来讲还是太昂贵、太笨重了，因此必须致力于开发高速的替代小型计算机系统接口（SCSI）及其他小系统 I/O 接口的产品。结果导致了 IEEE 标准 1394，一种高性能的串行总线，通常称为 FireWire。

FireWire 比老式的 I/O 接口具有更多的优点，它速度快、价格便宜而且容易实现。事实上，FireWire 不仅在计算机系统，而且在消费者电子产品，如数字照相机、DVD 播放器/录像机和电视机方面都很受欢迎，FireWire 用于传送日益丰富的数字视频图像。

FireWire 接口的特点之一是采用串行传送（每次一位）而不是并行传送。相对而言，并行接口（如 SCSI），需要更多的线，也意味着更宽、更昂贵的电缆和更宽、更昂贵的带有多外引脚的连接器。多线电缆需要屏蔽保护，以避免线与线之间的电气干扰。而且并行接口需要保证线之间的同步，如果电缆长度增加则问题会更突出。

此外，计算机体积变小的同时计算能力及 I/O 要求却不断增加，手提式和袖珍式计算机给连接器很小的空间，却要求高速的数据传输率来处理图像和视频信息。

FireWire 的目的是提供单一的 I/O 接口，它带有一个简单的连接器，能通过单个端口来处理多个设备，因此可以作为鼠标、激光打印机、外部磁盘驱动器、声音设备和局域网的连接器都能被这个连接器所取代。

1. FireWire 配置

FireWire 采用菊花链配置，一个端口最多可以连接 63 个设备。而且，高达 1022 条 FireWire 总线能用桥互联，以便系统支持所需要的大量外设。

FireWire 提供热插拔，连接或断开外设时不需要关闭计算机系统，也不需要重新配置系统。而且，FireWire 支持自动配置，即不需手工设置设备的 ID 或相关的设备位置配置。图 7-17 给出了一种简单的 FireWire 配置，FireWire 无需终结器，系统会自动进行设备地址分配。注意，FireWire 也可以采用树结构配置方式，而不一定总是精确的菊花链方式。

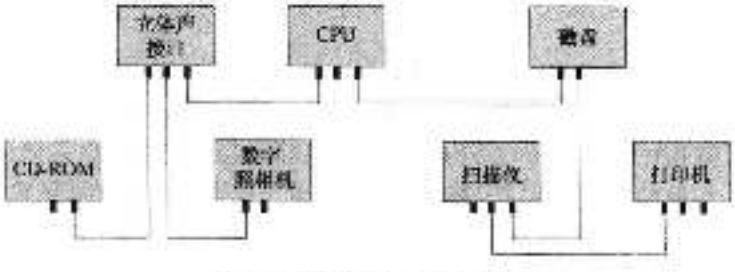


图 7-17 简单的 FireWire 配置

FireWire 标准的重要特点是，它用一组三层协议来标准化主机与外设间的串行交互，图 7-18 是模示意图，栈的三层协议是：

- 物理层：定义 FireWire 允许传输的媒体及每种媒体的电气和信号特性。
- 链路层：描述数据包的传输。
- 业务层：定义请求-响应协议，对应用层隐藏了 FireWire 较低层的细节。

2. 物理层

FireWire 物理层指定了一些可选传输媒体及连接器，它们有不同的物理特性和数据传输特性。物理层定义的数据传送速度为 25～3200Mb/s，它负责将二进制数据转换成多种物理媒体的电信号，并提供仲裁服务以保证每次只有一个设备发送数据。

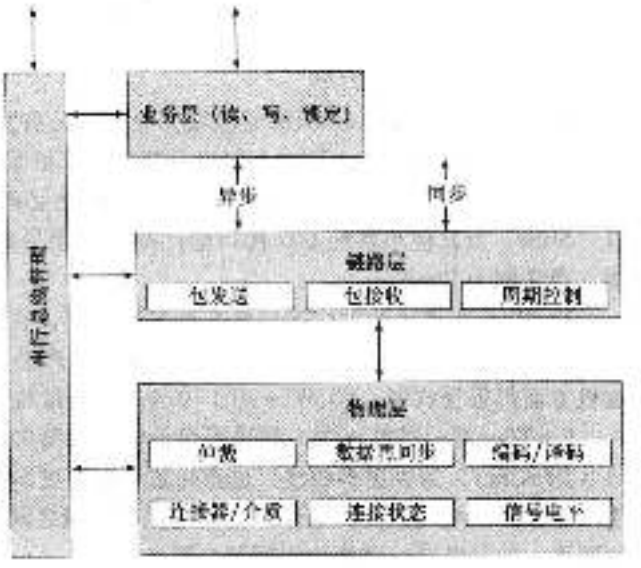


图 7-18 FireWire 协议栈

FireWire 提供了两种仲裁形式。最简单的一种形式是较早提出的基于 FireWire 总线上节点按树结构排列的方式。在特殊情况下它可以成为一种线性菊花链结构。物理层的逻辑允许所有连接设备对自己进行配置，因此一个节点成为树根，其他节点以父/子关系组织形成树的拓扑结构。一旦这种配置建立起来，根节点作为中央仲裁器，按先来先服务方式处理总线请求。如果同时有几个请求，则具有最高自然优先级的节点先存取。离树根越近的节点具有越高的优先级，离树根距离相等的节点，其 ID 数越小优先级越高。

上述仲裁方法还补充了公平仲裁和紧急仲裁两种附加功能。对于公平仲裁，总线上的时间被划分为相等的时间间隔。在一个时间间隔开始时，每个节点设置允许仲裁标志，在这个间隔内每个节点都可以竞争总线访问，一旦某个节点获得了总线访问，它就清除其允许仲裁标志，并在此时间间隔内不能再竞争总线。这种方案保证了仲裁的公平，避免了一个或多个忙的高优先级的设备独占总线。

除了公平方案外，还可以给一些设备配置紧急优先级。这种节点在一个间隔期间内可以多次获得总线控制权。实质上，在每个高优先级节点中使用一个计数器，使高优先级节点可控制总线的 75% 的时间。即一个包用于非紧急传送，而三个包可用于紧急传送。

3. 链路层

链路层定义了数据包的传输，它支持两种传送类型：

- 异步：将可变数量的数据和几个字节的传输层信息打包后传送到一个显式地址，并返回一个确认信号。
- 同步：将可变数量的数据以一系列固定大小的包形式、按规则的间隔传输，这种传输方式使用简单的寻址方式，且无需返回确认信息。

异步传输用于数据传输率可变的传输，公平仲裁和紧急仲裁都可以采用这种方式，其默认方法是公平仲裁。如果设备需要总线中数据的重要部分，或有严格的时间要求，则采用紧急仲裁方法。例如，当关键的数据缓冲区已满了一半以上时，高速实时数据采集节点就可以使用紧急仲裁。

图 7-19a 表示一个典型的异步传输。传递单个包的过程称为子动作，它包括五个时钟周期：

- 仲裁序列：这是发送给具有总线控制权的设备的信号交换。
- 包传输：每个包的包头包含了源 ID 和目的 ID，包头还包含包的类型信息、CRC（循环冗余码校验）检验和以及包的特殊类型参数。包还可能包含一个由用户数据和另一个 CRC 检验组成的数据块。
- 确认间隙：这是目标端接受和译码数据包并生成确认信号所需要的时延。

- **确认**：接收端回送一个确认包，表示数据包已经收到。
- **子动作间隙**：这是一个强制空闲周期，以保证总线上的其他节点在确认包传送前不进行总线仲裁。

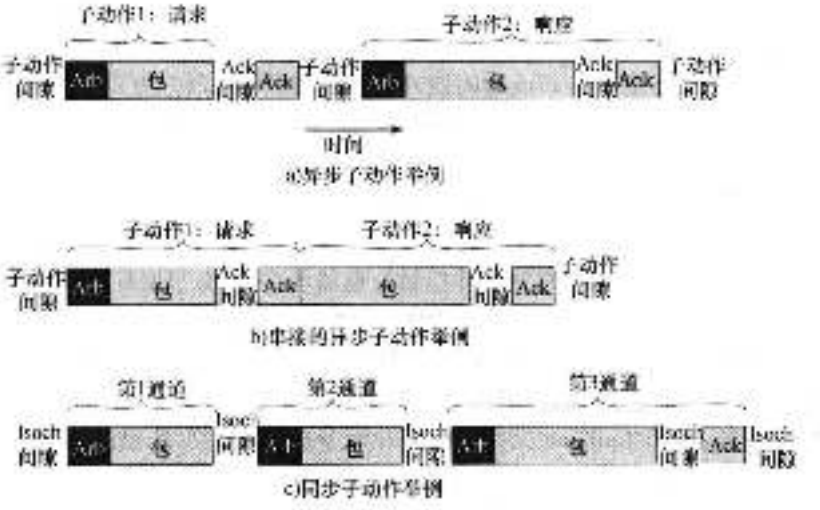


图 7-19 FireWire 子动作

接收端发送确认包时，它具有总线控制权，因此，如果在两个节点之间进行一次请求/响应传输，则接收节点不需要进行总线仲裁就可以立即传输确认包（如图 7-19b 所示）。

对于那些定期发出和接收数据的设备，如音频设备和视频设备，可以采用同步传输方式，这样可以保证数据在指定的时延内以一定的数据传输率进行传输。

为了协调同步信息和异步信息的混合传输，要指定一个节点为周期主节点。周期主节点定期地发送一个周期开始包，告诉其他所有节点同步周期开始。这个周期只进行同步传输（如图 7-19c 所示），通过总线仲裁，获得总线控制权的同步节点可以立即发送一个数据包，且无需确认。该数据包传输完毕后，其他的同步节点立即进行总线仲裁。这样，在一个包传输到下一个包仲裁之间会有一小段时间间隙，表示总线延迟，这个延迟称为同步延时，它小于子动作间隙。

同步传输结束后，总线保持一段足够长的空闲时间作为子动作间隙，这相当于告诉异步源节点：现在可以竞争总线了。然后，异步数据源使用总线一直到下一个同步周期开始。

同步数据包用 8 位通道号作为标识，由交换同步数据的两个节点间的对话先行指定。同步数据包的包头包含表示数据长度的域和头部 CRC 检验码，它比异步数据包的包头要短。

7.7.4 InfiniBand

InfiniBand 是定位高端服务器市场的一种最新 I/O 规范[⊖]，该规范的第一个版本发布于 2001 年早期并吸引了众多厂商。此标准描述了一种体系结构和处理器与智能 I/O 设备之间的数据流传输的规范。InfiniBand 已经成为了网络存储器和海量配置存储器的通用接口。实际上，它允许服务器、远程存储器以及其他网络设备连接到由交换器和链路组成的中央网络。这种基于交换器的体系结构最多可以连接 64 000 个服务器、存储系统和网络设备。

1. InfiniBand 体系结构

尽管 PCI 是一种可靠的传输方式，而且其速度在不断提升，目前已达 4Gb/s，但与 InfiniBand

⊖ InfiniBand 是两个竞争项目合并的结果，这两个项目是未来的 I/O（由 Cisco、迈普、康柏和 IBM 支持）和下一代 I/O（由 Intel 开发和其他很多公司支持）。

相比，它仍是一种受限的体系结构。拥有 InfiniBand，远程存储器和网络系统与服务器之间的连接是通过连接到一个由交换器和链路组成的中央网带实现的，它不需要服务端提供底层的硬件 I/O 接口。可以灵活地卸载 I/O 设备或者添加新的独立节点，允许更大的服务器密度且系统容易扩展。

与 PCI（它与 CPU 主板只相距厘米级的距离）不同，InfiniBand 的通道设计为：如果使用铜线，则允许 I/O 设备在距离服务器 17m 远的地方；如果是多模光纤，可以允许 300m 的距离；如果是单模光纤，则允许 10km 的距离。而且其数据传输率可以高达 30Gb/s。

图 7-20 说明了 InfiniBand 体系结构，其主要部件如下所示：

- **主机通道适配器**（host channel adapter，HCA）：典型的服务器不再采用多个 PCI 槽，而是只需要一个 HCA 接口，由 HCA 将服务器连接到 InfiniBand 交换器。HCA 连接到服务器的存储控制器，此控制器访问总线，并控制处理器与存储器、HCA 与存储器之间的数据传输。HCA 采用直接存储器存取方式（DMA）读写存储器。
- **目标通道适配器**（target channel adapter，TCA）：TCA 用于将远程存储系统，路由器和其他外围设备连接到 InfiniBand 交换器。
- **InfiniBand 交换器**：InfiniBand 交换器给多种设备提供点到点的物理连接，并将一条链路上的信息切换到另一条链路上。服务器与设备利用它们的适配器，经交换器相互通信。InfiniBand 交换器可以智能地管理各个链接而不需要中断服务器的操作。
- **链路**：介于 InfiniBand 交换器与通道之间，或两个交换器之间的链接线路。
- **子网**（subnet）：一个子网由一个或多个交换器，以及连接到这些交换器的设备组成。图 7-20 表示的子网只包含了一个交换器，但是更复杂的子网往往连接着大量的设备。它可以为管理员提供子网内部的广播和多播服务。
- **路由器**（router）：连接 InfiniBand 各子网或者将 InfiniBand 交换器连到局域网、广域网或存储区域网等。

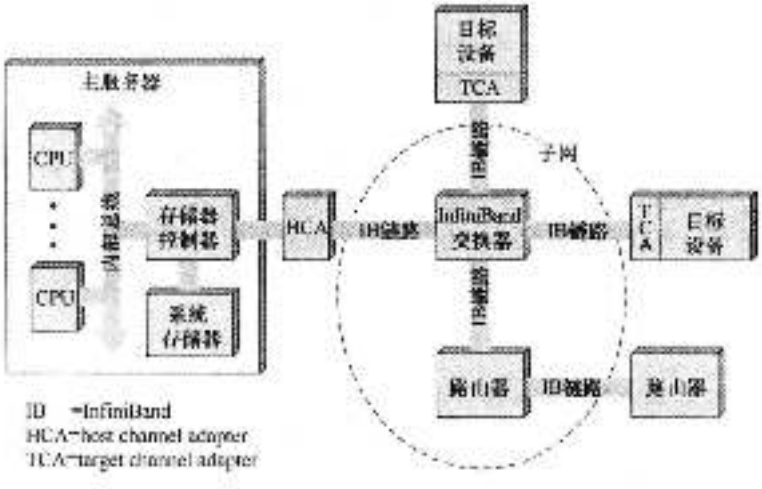


图 7-20 InfiniBand 交换器网带

通道适配器是一种智能设备，它能管理所有 I/O 操作而无需中断服务器的处理进程。例如，有一个控制协议，通过协议转换通道适配器能找到网带中所有的 TCA 和 HCA，并给每个 TCA 和 HCA 指派一个逻辑地址，而完成这些工作都无需处理器介入。

InfiniBand 交换器可以即时打开处理器与设备之间的通道，各设备并不共享通道的容量。这与 PCI 基于总线的设计不同，PCI 中各设备共享总线，访问处理器时要先进行总线仲裁。而且，

InfiniBand 通过将设备的 TCA 连接到 InfiniBand 交换器，就可以将设备添加到配置系统中。

2. InfiniBand 操作

交换器和与之连接的接口（HCA 或 TCA）之间的每条物理链路能支持多达 16 条逻辑通道，称为虚拟通路（virtual lanes）。一条通路用于网络管理，其余用于数据传输。数据以数据包流的形式发送，每个包包括部分待传数据以及地址信息和控制信息，使用一组通信协议管理数据传输。一条虚拟通路可以临时地作为专用数据传输路径，经 InfiniBand 网络将数据由一个端节点传送到另一个端节点。InfiniBand 交换器将来自输入通道的传输映射到输出通道，路由节点间的数据流动。

图 7-21 给出了支持 InfiniBand 数据交换的逻辑结构。事实上，有些设备发送数据的速度比接收设备接收数据的速度要快，因此在链路的两端都有一缓冲队列，用于临时存放输入和输出的数据。缓冲队列可位于通道适配器内，也可以在连接的设备存储器内。每条虚拟通路都有一对分离的缓冲队列，主机以如下方式使用这些队列。主机放置一个事件，称为工作队列项（Work Queue Entry，WQE），到该队列对的发送队列或接受队列，两个最重要的 WQE 是 SEND（发送）和 RECEIVE（接收）。对于 SEND 操作，其 WQE 为硬件指定设备存储空间的一块数据发送给目标设备。当有设备发送数据时，RECEIVE WQE 指定接收设备将接收的数据放在何处。通道适配器以适当的优先级顺序来处理每个递交的 WQE，并产生一个完成队列登记（Completion Queue Entry，CQE）来表示完成情况。

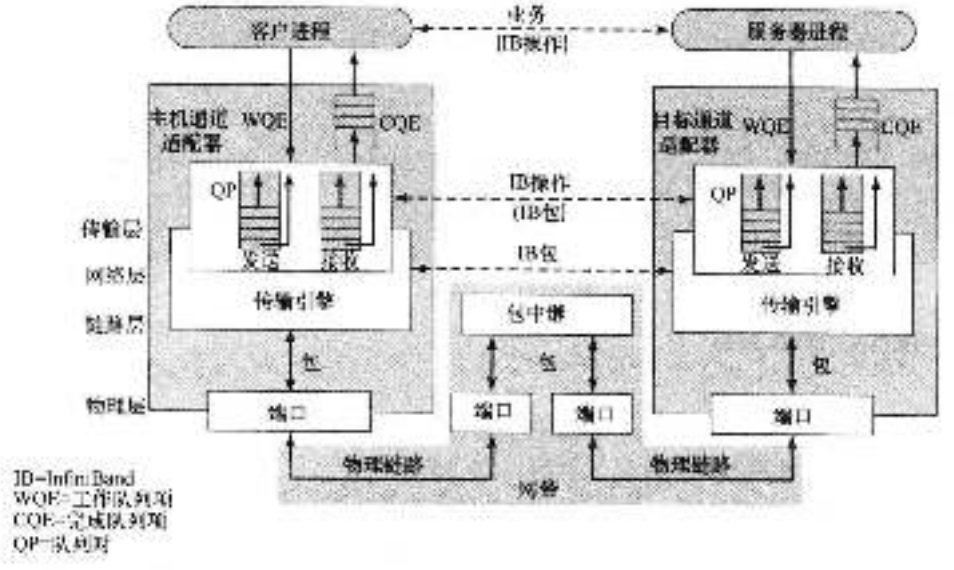


图 7-21 InfiniBand 通信协议栈

图 7-21 还表示了所用的协议层结构，一共有四个层次：

- **物理层**：该层定义了三种链路速度（1X，4X 和 12X），传输率分别是 2.5、10 和 30Gb/s（如表 7-3 所示）。它还定义了一些物理媒介，包括铜线和光纤。
- **链路层**：该层定义了数据传输的基本包结构，数据包中包含为子网中每个设备指定一个唯一的链路地址的寻址方式。该层还包含了一些逻辑，用来建立虚拟通路或将数据从源设备交换到目标设备。数据包结构还包含检错码，以提高数据传输的可靠性。
- **网络层**：该层提供不同 InfiniBand 子网之间的数据包的路由。
- **传输层**：该层为经过一个或多个子网的端到端的数据包传输提供可靠性机制。

表 7-3 InfiniBand 链路和数据吞吐率

链路	信号速度（单向）	可用容量（信号速率的 80%）	有效数据吞吐率（发送 + 接收）
1 宽	2.5Gb/s	2Gb/s(250MB/s)	(250 + 250)MB/s
4 宽	10Gb/s	8Gb/s(1GB/s)	(1 + 1)GB/s
12 宽	30Gb/s	24Gb/s(3GB/s)	(3 + 3)GB/s

7.8 推荐的读物和 Web 站点

关于 Intel I/O 模块和体系结构的讨论，包括 82C59A、82C55A、8237A，可参见［MAZI03］和［BREY09］。［ANDE98］中对 FireWire 有详细的介绍，［WICK97］和［THOM00］对 FireWire 也有简单的介绍。［SHAN03］和［FUTR01］对 InfiniBand 进行了详细的阐述。［KAGA01］也对它进行了简明的综述。

ANDE98 Anderson, D. *FireWire System Architecture*. Reading, MA: Addison-Wesley, 1998.

BREY09 Brey, B. *The Intel Microprocessor: 8086/8066, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4 and Core2 with 64-bit Extensions*. Upper Saddle River, NJ: Prentice Hall, 2009.

FUTR01 Futral, W. *InfiniBand Architecture: Development and Deployment*. Hillsboro, OR: Intel Press, 2001.

KAGA01 Kagan, M. "InfiniBand: Thinking Outside the Box Design" *Communications System Design*, September 2001. (www.csdmag.com)

MAZI03 Mazidi, M., and Mazidi, J. *The 80X86 IBM PC and Compatible Computer: Assembly Language, Design and Interfacing*. Upper Saddle River, NJ: Prentice Hall, 2003.

SHAN03 Shanley, T. *InfiniBand Network Architecture*. Reading, MA: Addison-Wesley, 2003.

THOM00 Thompson, D. "IEEE 1394: Changing the Way We Do Multimedia Communications." *IEEE Multimedia*, April-June 2000.

WICK97 Wickelgren, I. "The Facts about FireWire." *IEEE Spectrum*, April 1997.



推荐的 Web 站点

- T10 Home Page：T10 是（美）国家信息技术标准委员会的技术委员会，它负责低层接口，主要从事小型计算机系统接口（SCSI）方面的工作。
- 1394 Trade Association：提供 FireWire 的技术信息和厂商信息。
- InfiniBand Trade Association：提供 InfiniBand 的技术信息和厂商信息。
- National Facility for I/O Characterization and Optimization：是 I/O 设计和性能领域的教学和研究的专门机构，提供有用的工具和指导。

7.9 关键词、思考题和习题

关键词

cycle stealing：周期窃取
direct memory access（DMA）：存储器直接存取
FireWire：高速串行连接总线标准
InfiniBand：高带宽 I/O 标准
interrupt：中断
interrupt-driven I/O：中断驱动式 I/O
I/O channel：I/O 通道
I/O command：I/O 命令
I/O module：I/O 模块
I/O processor：I/O 处理器
isolated I/O：分离式 I/O
memory-mapped I/O：存储器映射式 I/O
multiplexor channel：多路转换通道
parallel I/O：并行 I/O

peripheral device：外围设备
programmed I/O：编程式 I/O
selector channel：选择通道
serial I/O：串行 I/O

思考题

7.1 列出外设或外围设备的三种主要分类。
7.2 什么是间接寻址？
7.3 I/O 模块的主要功能是什么？
7.4 列出并简单定义实现 I/O 的三种技术。
7.5 存储器映射式 I/O 与分离式 I/O 有什么区别？
7.6 当设备出现中断时，处理器如何知道是哪个设备发出的中断？
7.7 DMA 模块取得总线控制权并占用了总线时，处理器做什么？

习题

7.1 在典型的微处理器中，使用不同的地址去访问指定设备控制器中的 I/O 数据寄存器、控制和状态寄存器，这些寄存器称为端口。在 Intel 8088 中，使用两种 I/O 指令格式，一种是 8 位的操作码指定 I/O 操作，随后是 8 位的端口地址；另一种是 I/O 操作码指定端口地址在 16 位的 DX 寄存器中。对于上面两种寻址方式，8088 各能寻址多少个端口？

7.2 Zilog Z8000 微处理器系列采用类似的指令格式，一种是直接寻址，即指令中包含 16 位的端口地址；另一种是间接寻址，端口地址存放在 16 位的通用寄存器中。这两种方式各自的寻址范围是多少？

7.3 Z8000 还包含 I/O 数据块传输功能，与 DMA 不同，它是在处理器的直接控制下进行。块传送指令指定一端口地址寄存器（Rp），一个计数器（Rc）和一个目标寄存器（Rd）。Rd 存放主存地址，从输入端口读取的首个字节将存放在这一地址中。Rc 是 16 位通用寄存器。试问它一次能传送多大的数据块？

7.4 假设有一个微处理器，它有如 Z8000 这样的 I/O 数据块传输指令。这种指令第一次执行后，每 5 个时钟周期再执行一次。如果不用块传输指令，取指令和执行指令一共要用 20 个时钟周期。试计算，如果传送 128 字节的块，用块传输指令，速度可以提高多少？

7.5 一个基于 8 位微处理器的系统有两个 I/O 设备，系统两个控制器有各自独立的控制和状态寄存器，两个设备都是每次处理一个字节的数据。第一个设备有 2 根状态线和 3 根控制线，另一个设备有 3 根状态线和 4 根控制线。
(a) 要读取每个设备的状态信息和控制信息，I/O 控制模块需要多少个 8 位的寄存器？
(b) 假设第一个设备是只输出设备，那么寄存器数量又是多少？
(c) 控制两个设备，需要多少地址单元？

7.6 图 7-5 所示的编程式 I/O 需要处理器迫入一个等待时期来循环检测 I/O 设备的状态。为了提高效率，可以让处理器周期性地检查设备状态，即设备不就绪时，处理器就跳转到其他任务，一定时间间隔后处理器再来检查设备状态。
(a) 考虑采用上述办法向打印机一次一个字符地输出数据，打印机以 10 字符/秒的速度进行打印。若每 200ms 检查一次打印机的状态，将会出现什么情况？
(b) 考虑一个具有单一字符缓冲器的键盘，平均以 10 字符/秒的速度从键盘输入字符，两次连续按键的时间间隔可以短至 60ms，I/O 程序应该以多大的频率扫描键盘状态？

7.7 某微处理器每 20ms 扫描一次输出设备的状态，由定时器每 20ms 提醒处理器来完成。设备接口包括两个端口：一个表示设备状态，一个用于数据输出。若处理器时钟频率是 8MHz，那么它扫描并服务此设备要花费多长时间？为简单起见，所有相关指令的周期都取 12 个时钟周期。

7.8 在 7.3 节中已列出了存储器映射 I/O 相对于分离式 I/O 的一个优点和一个缺点。试再列出两个优点和两个缺点。

7.9 一特殊系统由操作员键入命令来控制，每 8 小时键入的平均命令数是 60。
(a) 假设处理器每 100ms 扫描一次键盘，那么 8 小时一共扫描了多少次？
(b) 若采用中断驱动式 I/O，处理器访问键盘的次数是问题 (a) 的百分之几？

7.10 考虑某一设备使用中断驱动式 I/O，此设备以平均 8KB/s 的速度连续传送数据。
(a) 假设中断处理大约用 100μs（即转移到中断服务例程（ISR），执行中断程序，然后返回到主程序

其用掉的时间)。如果每字节中断一次，则处理器百分之几的时间用于这个 I/O 设备？

(b) 假设这个设备有两个 16 字节的缓冲器，当一个缓冲器满时才中断处理器一次。当然，中断处理时间比较长，因为 ISR 还要传送 16 字节到缓冲器。在执行此 ISR 时，处理器每传送一字节大约要用 8μs。这种情况下，处理器百分之几的时间用于此设备？

(c) 假设处理器具有 Z8000 那样的数据块传输指令，允许相应的 ISR 每传送块中一字节仅用 2μs。这时，处理器百分之几的时间用于此设备？

7.11 在所有的含有 DMA 模块的系统中，DMA 访问主存储器的优先级比处理器访问主存储器的优先级高，为什么？

7.12 DMA 模块采用周期窃取方式把字符传输到存储器，设备的传输率是 9600b/s，处理器以 1 000 000 条指令/秒的速度获取指令（1MIPS）。这样，由于 DMA 模块窃取了总线周期，处理器速度将减慢多少？

7.13 考虑一个系统，其总线周期为 500ns。无论是从处理器到 DMA 模块，还是从 DMA 模块到处理器，总线控制的传递都用 250ns。一个 I/O 设备使用 DMA 方式，其数据传输率是 50KB/s。数据每次传送一个字节。

(a) 若使用突发式 DMA，即数据块传送之前 DMA 模块获得总线控制权，并一直维持对总线的控制，直到整个数据块传输完毕。当传送 128 字节的数据块时，设备占用总线多长的时间？

(b) 若使用周期窃取式 DMA，重复上问。

7.14 从 8237A 时序图可看出，一旦数据块传输开始，每个 DMA 周期占用 3 个总线周期。在 DMA 周期中，8237A 在存储器与 I/O 设备之间传输一个字节。

(a) 若 8237A 的时钟频率是 5MHz，那么传送一个字节需用多长时间？

(b) 可达到的最大数据传输率是多少？

(c) 假设存储器不够快，每个 DMA 周期必须插入 2 个等待状态，则实际数据传输率是多少？

7.15 假定在习题 7.14 的系统中，存储器周期为 750ns。我们能将总线的时钟频率降低到多少而不影响数据传输率？

7.16 一个 DMA 控制器服务于 4 条仅接收远程通信的链路（每个 DMA 通道一条链路），每条链路的速率是 64kb/s。

(a) 应以突发模式还是周期窃取模式来运行此控制器？

(b) 为服务各 DMA 通道，应采用哪种优先权策略？

7.17 一个 32 位的计算机有两个选择通道和一个多路转换通道，每个选择通道支持两个盘组和两个磁带设备；每个多路转换通道与两台行式打印机，两台卡片输入机和 10 个 VDT 终端连接。假设传输速率如下：

磁盘驱动器	800KB/s
磁带驱动器	200KB/s
行式打印机	6.6KB/s
卡片输入机	1.2KB/s
VDT	1KB/s

估算这个系统最大的总 I/O 传输速率是多少？

7.18 一台计算机有一个处理器和一个 I/O 设备 D，D 通过单字宽的共享总线连到主存储器 M，处理器的最大速度是 10^6 条指令/秒，平均一条指令需 5 个机器周期，其中有 3 个机器周期需要占用存储器总线。存储器的读或写操作占用一个机器周期。假设处理器连续执行“后台”程序，它的执行速度是指令执行速度的 95%，并假定处理器周期等于总线周期。现在，假设 I/O 设备负责 D 与 M 间的大批数据块传输。

(a) 如果采用编程式 I/O，I/O 传输一个字需处理器执行两条指令，试估算可能经过 D 的最大 I/O 数据传输速率，单位用字/秒。

(b) 如果采用 DMA，条件同上，其传输率又是多少？

7.19 一个数据源产生 7 位 IRA 的字符，每个字符增加一个奇偶校验位。推导在以 Rb/s 速度传输的线上最大的有效数据传输速度（IRA 码数据位的速度）的表达式。

(a) 异步传输，有 1.5 个单元的停止位。

(b) 位同步传输，由 48 个控制位和 128 个数据位组成一帧。

(c) 与上题相同，但有1024个数据位。

(d) 字符同步，每帧有9个控制字符和16个信息字符。

(e) 与上题相同，但有128个信息字符。

7.20 下面的问题基于IECKE00...中的输入/输出机制（如图7-22所示）：

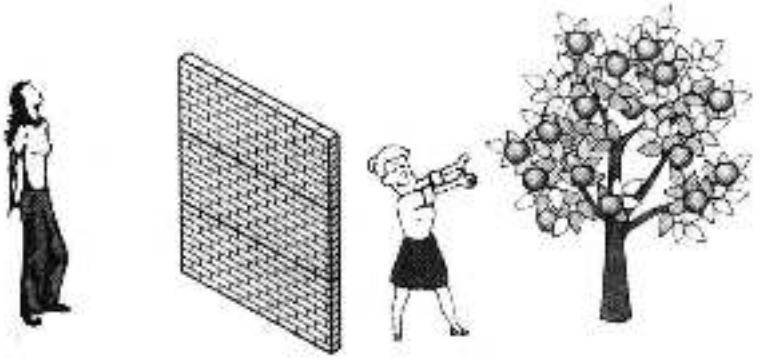


图7-22 苹果问题

两名妇女分别住在高篱笆的两边，一名叫Apple-server，她的这边有一棵长满苹果的苹果树，她愿意随时把苹果给另一名想要苹果的妇女。另一边的妇女名叫Apple-eater，喜欢吃苹果，但她没有苹果树。事实上，Apple-eater吃苹果的速率是固定的（一天一个苹果能保持健康），如果她的速率比这个速率快，则会生病；如果比这个速率慢，则会营养不良。这两名妇女都不能说话。问题是Apple-eater怎样适时地从Apple-server那里得到苹果。

(a) 假设在篱笆的顶上有一只警钟，它有多个振臂的设置。怎样使用警钟来解决这个问题，画出时序图来说明方案。

(b) 现在假设没有警钟，但Apple-eater有一面旗，当她需要苹果时能挥动它。请提出来一个新的解决方案。如果Apple-server也有一面旗的话，有助于解决这个问题吗？如果有，综合到此答案中，讨论这种方案的缺点。

(c) 现在拿走旗，并假设有一根长绳，给出使用长绳比问题(b)更好的解决方案。

7.21 假设有一个16位和两个8位的微处理器连接到系统总线，给定下列条件：

(1) 所有的微处理器具有所需的硬件特性，用来支持各种类型的数据传送：编程式I/O、中断驱动式I/O和DMA。

(2) 所有的微处理器有16位的地址总线。

(3) 有两块存储板与总线相连，每块容量是64KB。设计者希望尽可能地使用共享存储器。

(4) 系统总线最多支持4根中断线和1根DMA线。

所需的其他假设条件都成立，要求：

(a) 根据线数和类型给出系统总线规范。

(b) 写出可能用到的总线传输协议（即读-写、中断、DMA序列）。

(c) 描述上述设备是怎样连到系统总线上的。

第8章

Computer Organization and Architecture: Designing for Performance, 8E

操作系统支持

本章要点

- 操作系统（OS）是控制程序在处理器上执行和管理该处理器资源的软件。操作系统具有许多功能，包括进程调度和存储管理，但这些功能的实现离不开处理器硬件的支持。事实上，所有处理器都或多或少地具备这种能力，如虚拟存储器管理硬件和进程管理硬件。这些硬件包括专用寄存器、缓冲器以及完成基础资源管理任务的电路。
- 操作系统最重要功能之一是进程调度任务的调度，操作系统决定在给定时间内运行哪个进程。一般情况下，硬件不断中断运行进程，使操作系统做出新的调度裁决，从而使处理器时间被几个进程公平分配。
- 操作系统的另一个重要功能是存储管理。大多数当代操作系统都包含虚拟存储器的功能，虚拟存储器有两个优点：(1) 进程在主存中运行时不需要将程序的全部指令和数据一次性地装入主存；(2) 程序可用的总存储空间可以大大超过系统实际的主存容量。虽然存储管理是用软件完成的，但操作系统依赖于处理器中的硬件支持，包括分页管理硬件和分段管理硬件。

尽管此书的重点是计算机硬件，但软件中的一大领域——计算机操作系统必须要说明。操作系统是一个管理计算机资源，为程序员提供服务以及调度其他程序执行的系统软件。理解操作系统对于理解 CPU 控制计算机系统的机制有很重要的意义。特别是，它能很好地解释中断的作用和存储器层次结构的管理。

本章首先概述操作系统及其简史，然后详细介绍与计算机组成和体系结构密切相关的操作系统的两大功能——调度和存储器管理。

8.1 操作系统概述

8.1.1 操作系统的目标与功能

操作系统是一种控制应用程序运行和在计算机用户与计算机硬件之间提供接口的程序，它有两个目标：

- **方便**：操作系统使计算机使用起来更方便。
- **有效**：操作系统允许计算机系统的资源以有效的方式使用。

下面依次探讨操作系统的这两个方面。

1. 操作系统提供用户与计算机之间的接口

为用户提供各种应用服务的软硬件可以划分为不同的层次，如图 8-1 所示。这些应用程序的用户（最终用户）通常不关心计算机的体系结构，因此最终用户将计算机系统看成是一个应用系统。它可以用程序设计语言描述，并且由应用程序员开发。开发一组机器指令构成的应用程序来完全负责计算机硬件的管理，这是一个极为复杂的任务，为了简化这个任务，提供了一组系统程序，其中一些系统程序称为实用程序。

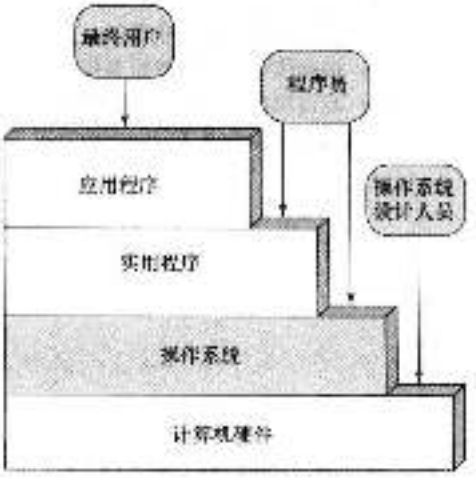


图 8-1 计算机系统的布局和视图

这些程序通常用于程序创建、文件管理和I/O设备控制。程序员借助系统程序开发应用程序，并在运行应用程序时调用系统程序以执行某些功能。最主要的系统程序是操作系统，操作系统对程序员屏蔽硬件细节，为程序员使用计算机系统提供方便的接口。它作为一个中间软件，使程序员和应用程序都十分容易访问和使用系统提供的资源和服务。

通常，操作系统主要提供以下服务：

- **程序创建**：操作系统提供了各种工具和服务，如编辑器和调试器，帮助程序员创建程序。这些工具通常以实用程序的形式出现，实际上并不是操作系统的一部分，但可以通过操作系统来使用。
- **程序执行**：执行一个程序需要完成许多任务。必须将指令和数据调入主存，必须初始化I/O设备和文件，以及必须准备好其他资源等。操作系统为用户处理所有这些事情。
- **存取I/O设备**：每个I/O设备都有其特定的指令集或控制信号。操作系统负责处理这些细节，以便程序员只需考虑简单的设备读和写操作。
- **文件的存取控制**：实现文件的存取控制需要深入了解I/O设备（磁盘驱动器、磁带机）的属性和存储介质上文件的存储格式。操作系统负责管理这些细节，而且对于多用户系统，操作系统还要提供控制文件存取的保护机制。
- **系统存取**：在共享或公共系统中，操作系统控制对整体或特定系统资源的存取，必须提供对资源和数据的保护，防止未授权的访问，解决共享资源的访问冲突。
- **错误检查和响应**：在计算机系统的运行时可能出现各种错误，既包括如存储器错、设备故障或失效之类的内部和外部硬件错误，也包括如算术溢出、企图存取禁止的内存地址以及未授权的应用请求之类的各种软件错误。出错时，操作系统必须响应并消除错误条件，尽量减少对应用程序运行的影响。响应的方法有终止出错程序、重试和简单报告错误等。
- **统计**：好的操作系统应该能统计各种资源的使用情况，并能监督各性能参数，如响应时间。对于任何系统，这些信息为将来增加系统功能和改善系统性能提供了有益的参考。对于多用户系统，这些信息还可用作计费账单。

2. 操作系统作为资源管理器

计算机是一组具有传送、存储和处理数据并控制这些功能的资源，而操作系统负责管理这些资源。

我们能否说是操作系统在控制数据的传送、存储和处理呢？从某种程度上来说，答案是肯定的。通过管理计算机的资源，操作系统控制计算机的基本功能，而且这种控制是以独特的方式进行的。通常，控制机构对受控对象来讲是外部的，或至少是与受控对象分离的独立部分（例如，住宅供热系统由恒温器控制，它完全不同于热产生和热传送装置）。而操作系统则不同，操作系统作为控制机构在两个方面很独特：

- 操作系统功能的实现与普通的计算机软件相同，也就是说，它是由处理器执行的程序。
- 操作系统经常放弃控制权，并必须依赖处理器的启用而重新获得控制权。

事实上，操作系统只不过是一个计算机程序，和其他计算机程序一样，它为处理器提供指令。两者主要的区别在于程序的目的不同。操作系统指导处理器使用其他系统资源并为其他程序的执行定时。但是为了完成上述任务，处理器必须停止执行操作系统而去执行其他程序。因此，操作系统放弃控制，让处理器做一些“有用的”工作，然后恢复控制一段时间，使处理器有足够的时间为下一步工作做准备。随着本章的深入，这一机制将逐渐清晰。

图8-2给出了操作系统管理的主要资源。操作系统中的一部分在主存中，包括操作系统内核，或者称为核，它包含操作系统中使用最频繁的功能，以及目前正在使用的功能。还有一部分操作系统正在使用。主存的其余部分用于存放用户程序和数据。我们将看到，主存中的这些资源

分配由操作系统和处理器中的存储管理部件联合控制。操作系统决定什么时候I/O设备被某个可执行程序使用，并控制文件的存取和使用。处理器自身也是一种资源，操作系统必须决定分配多少处理器时间给用户程序。在多处理器系统中，操作系统还要对所有处理器进行裁决。

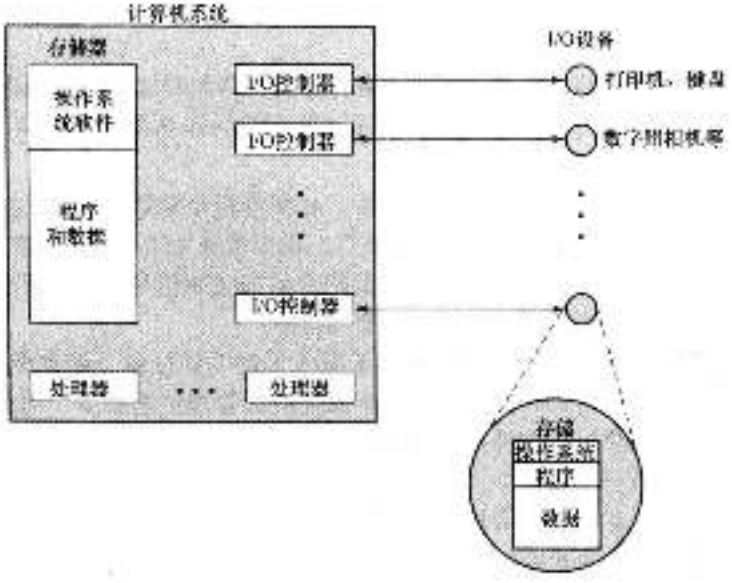


图 8-2 操作系统作为资源管理器

8.1.2 操作系统的类型

根据不同的特性可以将操作系统分类，分类方式有两种。一种是把操作系统分成批处理和交互式处理。在交互式系统中，用户或程序员通常用键盘或显示器终端直接与计算机交互，请求执行一个作业或处理一个事务。而且用户可以根据应用程序的性质，在作业执行期间与计算机通信。批处理系统与交互式系统相反，许多用户程序打包后由计算机操作员成批提交处理。程序处理结束后，打印处理结果。单纯的批处理系统目前用得很少，然而，简单地了解批处理系统对了解现代操作系统是有帮助的。

另一种分类是看系统是否采用了**多道程序设计**。多道程序设计系统通过让处理器一次处理多个程序，使处理器尽可能地忙碌。将多个程序同时装入主存，处理器迅速地在它们中间进行切换。另外一种是**单道程序设计**系统，处理器一次只运行一个程序。

1. 早期的系统

最早的计算机出现在20世纪40年代末到50年代中期，那时没有操作系统，程序员直接与计算机硬件交互。控制台控制机器运行，它由指示灯、触发器、输入设备和打印机组成。输入设备（如卡片机）装入机器代码程序，如果程序出错中止，则指示灯指示错误状态。程序员便不得不着手检查寄存器和主存，以确定出错原因。如果程序最终能够正常完成，则结果由打印机打印输出。

早期的系统存在两个主要问题：

- **调度**：通常，用户可以签约一段时间，如数个半小时，等等。如果用户签约了1小时，而实际只用了45分钟完成任务，这就浪费了处理器的时间。另外，如果用户遇到问题，则在分配的时间内不能完成，只能停止任务的执行，等解决问题后再去执行该任务。
- **安装时间**：一个单一程序称为作业，一个作业的完成分为多个步骤：把编译程序和高级语言程序（源程序）装入内存，保存已编译的程序（目标程序），然后连接目标程序和

库函数。每一步都要安装或卸下磁带或卡片组。如果出错，用户只得从头开始重新安装一遍。于是，大量的时间浪费在程序安装上。

这种操作模式称为串行处理，即用户必须按顺序使用计算机。后来，人们开发出了各种系统软件工具，试图提高串行处理的效率，包括公共函数库、连接库、装载程序、调试程序和I/O驱动程序，它们作为共享软件可被所有用户使用。

2. 简单的批处理系统

早期的处理器非常昂贵，因此最大化处理器的利用率是很重要的，因调度和安装而浪费时间是不允许的。

为了提高利用率，人们开发了简单的批处理系统。这种系统也称为监控程序，用户不再直接操作机器，而是将作业提交在卡片上或磁带上，由计算机操作人员按顺序把作业成批地放在一起，然后放到输入设备上，最后由监控程序执行这些任务。

为了理解这种工作机制，我们从监控程序和处理器两个方面来讨论。从监控程序的角度看，它控制事件的发生顺序，为此监控程序的大部分必须驻留在主存中，并随时可执行（如图8.3所示），这部分监控程序称为**常驻监控程序**。其他的监控程序在任务开始执行时作为子函数给用户提供基本的功能和服务。监控程序每次从输入设备（典型的有卡片阅读机和磁带机）读入一个作业。读入时，当前作业调入用户程序区，并对它进行控制，作业完成后，控制权交回给监控程序以读入下一个作业，然后打印出每个作业结果给用户。

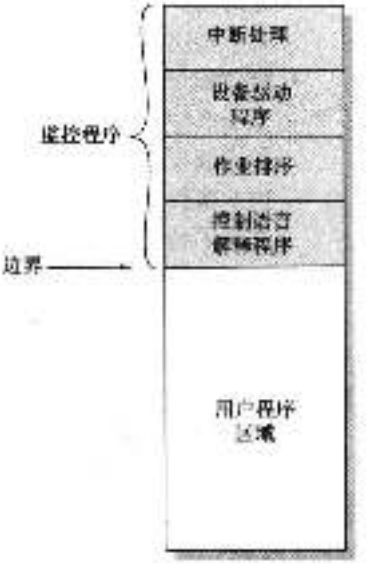


图8.3 常驻监控程序的内存分布

现在，从处理器的角度来看这个执行序列。在某个特定时间，处理器执行主存中包括监控程序在内的部分指令。这些指令读入下一个作业到主存的另一部分单元中。作业读入后，处理器遇到监控程序的分支指令，指示处理器去继续执行用户程序首地址处的程序。然后，处理器执行用户程序中的指令，直到结束或出现错误，这时处理器从监控程序读取下一条指令。因此，惯用语“对作业进行控制”表示处理器正在读取和执行用户程序中的指令，而“返回到监控程序控制”表示处理器正在读取和执行监控程序中的指令。

很显然，监控程序处理了调度问题。将一批作业排好序，并尽快地执行作业，从而不会浪费处理器的时间。

作业安装时间呢？监控程序也能处理好作业安装问题。对于每个作业，指令包含在作业控制语言（job control language，JCL）中，这是提供给监控程序的一种特殊的程序设计语言，例如，用户提交一个用FORTRAN语言书写的程序以及相关的一些数据，每一条FORTRAN指令和每一个数据都在一个单独的卡片或者磁带记录区中。除了FORTRAN程序和数据卡片外，卡片组中还包括作业控制指令，用起始符“$”表示。整个程序有如下格式：

```
$ JOB
$ FTN
.
.          } FORTRAN 指令
.
$ LOAD
$ RUN
.
.          } 数据
.
$ END
```

执行这个作业时，监控程序读到 $FTN 行，并将相应的编译程序从海量存储器（通常是磁带机）中取来装载。编译程序将用户程序翻译成目标代码，并存于内存或海量存储器上，如果存于内存，则此操作过程称为“编译、装载和运行”；如果是存于磁带，则需要 $LOAD 指令。监控程序读取该指令，在编译操作完成之后重新取得程序控制权，它调用装载程序，将目标代码装入内存替代编译程序，并将控制权传送给目标代码。这样，大段的主存可被几个不同的子系统共享。虽然一次只能有一个子系统驻留于主存并被执行。

可以看出，监控程序和批处理操作系统就是简单的计算机程序，它借助于处理器从主存各单元获取指令以达到获取和放弃控制权交替进行的目的；同时它也需要其他硬件的支持：

- **存储器保护**：当用户程序正在执行时，禁止修改含有监控程序的存储器单元。否则，处理器硬件将检测到错误，并把控制权交付给监控程序。监控程序停止执行这个作业，然后打印出错信息，并装载下一个作业。
- **定时器**：定时器用来避免单个作业独占系统。它在每个作业开始执行时设置，如果时间到，则出现一个中断，并且将控制权返还监控程序。
- **特权指令**：特权指令只能被监控程序执行。如果处理器在执行用户程序时遇到特权指令，则会产生一个错误中断。特权指令包括了 I/O 指令，因此监控程序保留对所有 I/O 设备的控制权，这避免了用户程序偶然地从下一个作业中读取作业控制指令。如果用户程序想要执行 I/O 指令，则必须请求监控程序来执行该指令。如果处理器在执行用户程序时遇到了特权指令，则处理器硬件认为出现了错误，并把控制权转交给监控程序。
- **中断**：早期的计算机不具备中断功能。中断使操作系统更方便地挂起和获得控制权。

处理器交替执行用户程序和监控程序，这就存在两个开销：一是监控程序占用一些主存；另一个是监控程序占用了一些处理器时间。这两部分都属于系统开销，尽管如此，简单的批处理系统仍提高了计算机的利用率。

3. 多道程序批处理系统

尽管简单的批处理操作系统提供了自动作业序列，但处理器仍经常空闲，问题就是 I/O 设备的速度比处理器要慢。图 8-4 详述了一个典型的计算，它考虑的程序处理一个文件记录并且平均每个记录需执行 100 条机器指令，此例中，计算机花费 96% 以上的时间等待 I/O 设备传送数据！图 8-5a 描述了这个过程。处理器运行一段时间直到它到达一个 I/O 指令便停止，然后它必须一直等待直到 I/O 指令结束。

从文件中读一个记录	15μs
执行 100 条指令	1μs
向文件写一个记录	15μs
总计	31μs

CPU 利用率 $= \frac{1}{31} = 0.032 = 3.2\%$

图 8-4 系统利用率举例

这种低效率是可以避免的，我们知道，必须有足够的存储空间来装载操作系统（常驻监控程序）和一个用户程序。假设存储空间足够装载操作系统和两个用户程序，那么当一个作业需等待 I/O 时，处理器可以转去处理另一个作业，而不必等待 I/O（如图 8-5b 所示）。此外，可以扩充内存来装载 3 个、4 个或更多的程序，并在它们中切换处理（如图 8-5c 所示）。这种技术称为多道程序设计或多任务化[⊖]，它是现代操作系统的中心议题。

例 8.1 为了理解多道程序设计的优点，我们来看一个例子。假设某计算机有 250MB 的可用存储空间（不被操作系统使用）、1 个磁盘、1 台终端和 1 台打印机，同时提交了 3 个程序 JOB1、JOB2 和 JOB3，表 8-1 列出了它们的资源需求情况。假设 JOB2 和 JOB3 需要最少的处理需求，JOB3 需要连续使用磁盘和打印机。如果是单道程序设计环境，这些作业将顺序执行。因此，JOB1

⊖ 术语“多任务化”有时专指于指同一程序中可被处理器并发处理的多个任务，而术语“多道程序设计”是指来自多个程序的多个任务。但是经常的是将这两个术语等价使用，正如大多数标准字典中的一样（例如，IEEE Std 100-1992，The New IEEE Standard Dictionary of Electrical and Electronics Terms）。

用5分钟完成；JOB2等待5分钟后执行，然后用了15分钟完成；JOB3在20分钟后开始，从提交到作业完成花了30分钟。平均的资源利用率、吞吐量和响应时间在表8-2所示的单道程序设计栏中表示。每个设备的利用率在图8-6a中表示。很明显，当平均周期时间是30分钟时，所有资源总的利用率很低。

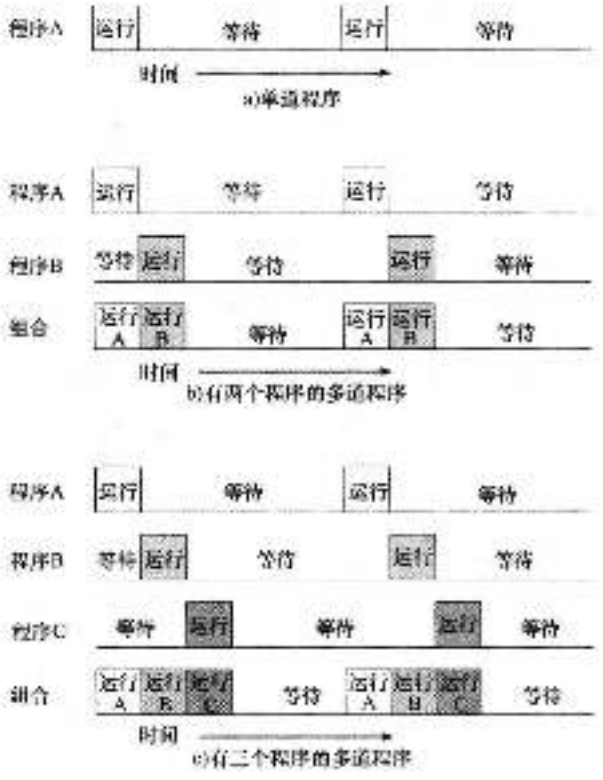


图8-5 多道程序举例

表8-1 程序执行属性范例

	作业1	作业2	作业3
作业类型	计算密集	I/O密集	I/O密集
持续时间	5分钟	15分钟	10分钟
存储需求	50M	100M	80M
需要磁盘?	否	否	是
需要终端?	否	是	否
需要打印机?	否	否	是

表8-2 多道程序设计对资源利用率的影响

	单道程序设计	多道程序设计
处理器利用率	20%	40%
存储器利用率	33%	67%
磁盘利用率	33%	67%
打印机利用率	33%	67%
占用时间	30分钟	15分钟
吞吐率	6份作业/小时	12份作业/小时
平均响应时间	18分钟	10分钟

现在假设3个作业在多道程序设计操作系统下并发运行。因为在这些作业中几乎无资源竞争，当3个作业共存于计算机中时，运行时间可以实现最小化（假设JOB2和JOB3分配了足够的处理器时间用于输入和输出操作）。JOB1仍需要5分钟完成，但这时，JOB2已完成了它1/3的作业，JOB3完成了它一半的作业，所有这3个作业在15分钟内完成。从表8-2所示的多道程序设计一栏及图8-6b所示的直方图中可以看到性能有明显的改善。

和简单的批处理系统一样，多道程序设计的批处理系统也必须依赖于计算机的硬件特性，多道程序设计最显著的附加特点是其硬件支持I/O中断和DMA。有了中断驱动I/O或DMA功能，处理器能为一个作业发送一个I/O命令，并继续执行另一个作业，I/O操作由设备控制器执

行。当I/O操作完成时，处理器被中断，转而执行中断处理程序，然后操作系统再去控制另一个作业。

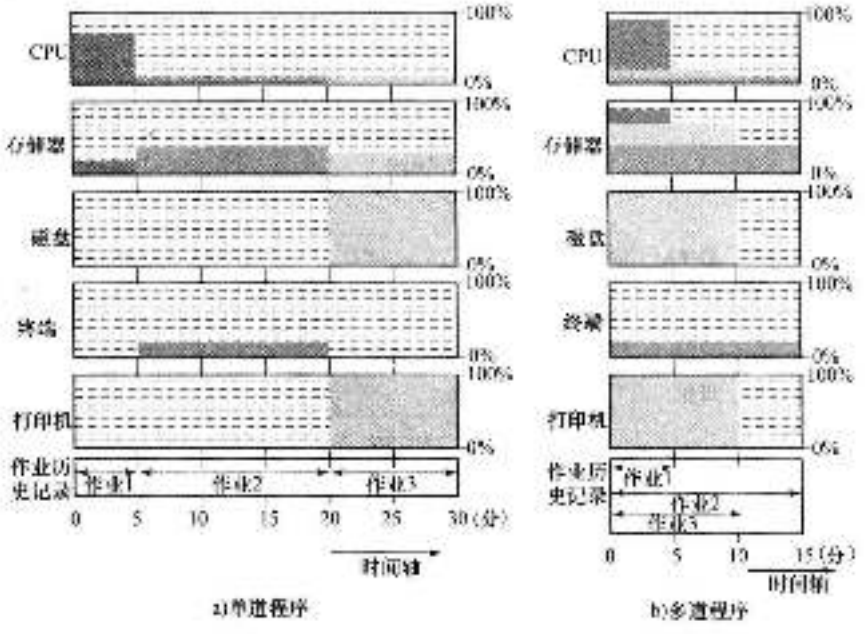


图8-6 利用率直方图

与单一程序或单道程序设计系统相比，多道程序设计的操作系统要复杂得多。作业运行前必须存入内存，需要存储管理机制。此外，如果有几个作业准备就绪，则处理器必须决定先运行哪一个，这就需要调度算法。这些概念在本章后面将进行讨论。

4. 分时系统

随着多道程序设计的使用，批处理方法显得非常有效。然而，许多作业需要用户与计算机直接交互，例如一些事务处理的作业，必须采用交互模式。

现在，通常采用专用的微型计算机作为交互式计算设备。在20世纪60年代，由于许多计算机体积大且价格昂贵，采用专用微型计算机来实现交互式是不可选的，因此，分时系统应运而生。

正像多道程序设计允许处理器一次处理多个批量作业一样，多道程序也能同时处理多个交互式作业，即分时技术，它允许多个用户共享处理器时间。分时系统中，多个用户通过终端同时访问系统，操作系统按很短的时间片或计算量来交错执行每个用户程序。于是，若一次有n个用户请求服务，则每个用户平均只看到计算机有效速度的1/n，不计操作系统开销。然而，在人机交互时间相对较慢的情况下，恰当设计系统，其响应时间能与专用系统差不多。

批处理的多道程序设计和分时系统都采用多道程序设计，它们的主要区别在表8-3中列出。

表8-3 批处理多道程序设计和分时系统的比较

	批处理多道程序设计	分时系统
主要目标	处理器利用率最大	响应时间最小
对操作系统的输入	作业提供的作业控制语言命令	在终端上输入的命令

8.2 调度

多道程序的关键是调度。事实上，调度有4种典型的类型（如表8-4所示），在此我们将说明这些类型。首先介绍一个概念：进程。这个术语在20世纪60年代首次由Multics操作系统的设计者提出，在某种程度上讲，它是比“作业”更通用的术语。对于术语进程，已经有许多定义，包括：一个运行的程序、程序的“活灵魂”、处理器分配的实体。

随着我们讨论的深入，进程的概念将变得越来越清楚。

表8-4 调度类型

长期调度	决定添加到待执行的进程池中的进程数
中期调度	决定添加到主存的进程数（可全部或部分在主存中）
短期调度	决定处理器将执行哪个进程
I/O调度	决定哪个进程未决的I/O请求将被I/O设备处理

8.2.1 长期调度

长期调度确定哪些程序提交给系统处理，因此，它控制多道程序设计的深度（内存中的进程数）。一旦提交，作业或用户程序就成为进程，并加入到短期调度的队列中。在某些系统中，新建的进程处于换出状态，此时，它们会加入到中期调度的队列中。

在批处理系统或通用操作系统的批处理部分中，新提交的作业保存在磁盘上，并在批处理队列中。长期调度程序从该队列创建进程，这有两个决定性的因素：第一，调度程序必须确定操作系统能运行一个或多个额外的进程。第二，调度程序必须决定接受哪一个作业或作业组，并把它转换成进程。采用的标准可以包含优先级、预期执行时间和I/O需求。

对于分时系统中的交互程序，当用户试图访问系统时会产生一个进程请求。分时用户不是简单地排队和等待系统接收它，而是操作系统通过使用某种算法和预决定测量，不断地接收所有授权的用户，直到系统饱和。此时，连接请求会遇到“系统已满和以后再试”的信息。

8.2.2 中期调度

中期调度是8.3节将介绍的交换功能的一部分。一般情况下，换入决策是基于需要管理多道程序的深度。在没有虚拟存储的系统中，存储管理也是一个问题，因此，换入决策要考虑到换出进程的存储器要求。

8.2.3 短期调度

长期调度程序执行相对较少，并且它只粗粒度地判定是否执行一个新的进程，以及调度哪一个进程进入系统。而短期调度程序，也称为派遣程序，执行频繁，并且它细粒度地判定接下来运行哪个作业。

1. 进程状态

为了理解短期调度程序的机制，我们需要考虑一个概念：进程状态。在进程的生命周期中，它的状态会变化很多次，任何时刻所处的状况称为一个进程状态。采用术语“状态”是因为在任何一种状态它都有特定的状态信息。通常，进程有5种定义的状态（如图8-7所示）：

- **新建**：由高级调度程序提交一个程序，但此程序未就绪。操作系统将为此程序创建一个进程，并将它移入就绪状态。
- **就绪**：进程已准备就绪，正在等待处理器的执行。
- **运行**：进程正被处理器执行。

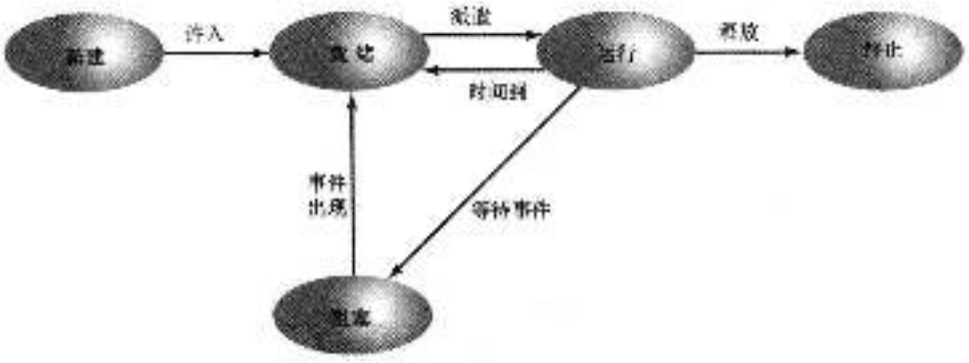


图 8-7　五状态进程模型

- 等待：进程为等待一些系统资源（如 I/O），由运行状态挂起。
- 终止：进程结束，将由操作系统撤销。

对于系统中的每个进程，操作系统必须保存进程状态信息和进程执行所需的其他信息。为此，每个进程在操作系统中用一个进程控制块（如图 8-8 所示）来表示，通常，进程控制块（PCB）包含如下信息：

- 标识符：每个当前进程都有一个唯一的标识符。
- 状态：进程的当前状态（新建、就绪等）。
- 优先级：进程的相对优先级别。
- 程序计数器：要执行的程序中的下一条指令的地址。
- 存储器指针：进程在存储器中的起始位置和结束位置。
- 现场数据：进程正在执行时，处理器寄存器中的数据，它们将在本书的第三部分予以讨论。目前，可以说这些数据表示进程的“现场”。当进程离开运行状态时，要保存现场数据和程序计数器的值；当恢复执行这个进程时，处理器需要读回这些信息。
- I/O 状态信息：包括未完成的 I/O 请求、分配给这个进程的 I/O 设备（如磁带机）和分配给进程的文件列表等。
- 统计信息：可以包括使用处理器的时间和时钟时间、时间限制、账户编号等。



图 8-8　进程控制块

当调度程序接收一个新的作业或用户的执行请求时，它创建一个空的进程控制块，并使相关进程处于新的状态。当系统填完 PCB 后，该进程进入就绪状态。

2. 调度技术

为了理解操作系统是如何管理存储器中各种作业的调度问题，让我们首先考虑图 8-9 中的简单例子。该图表示在某一给定时刻主存的分区情况。当然，操作系统的内核常驻内存，另外，有一些活动的进程，包括 A 和 B，都需要分配存储空间。

先从进程 A 正在运行的这一时刻开始考虑，这时处理器正在执行 A 程序的指令，后面几个时刻，处理器停止执行 A 的指令，开始执行操作系统的指令。这是由下列三种原因之一引起的：

（1）进程 A 发送一个服务请求（例如一个 I/O 请求）给操作系统，进程 A 处于挂起状态，直到操作系统满足这个请求为止。

（2）进程 A 产生一个中断，这个中断是由硬件产生并发送给处理器的信号。当处理器检测

到这个信号时，它停止执行A，转去执行操作系统中的中断处理程序。与A相关的各种事件都可以产生中断，一个简单的例子是出错，例如试图执行特权指令。另一个例子是超时，为防止某个进程独占处理器，每个进程只允许每次占用处理器很短的时间。

(3) 与进程A无关，但需注意一些事件会引起中断，例如一个I/O操作完成。

无论何种原因，结果均如下。首先处理器保存进程A控制块中当前现场信息和程序计数器信息，然后开始运行操作系统。操作系统可以完成一些工作，例如初始化I/O操作。接着操作系统的短期调度程序决定下次将执行的进程，此例中是B进程。操作系统指示处理器恢复B的现场数据，执行B进程。

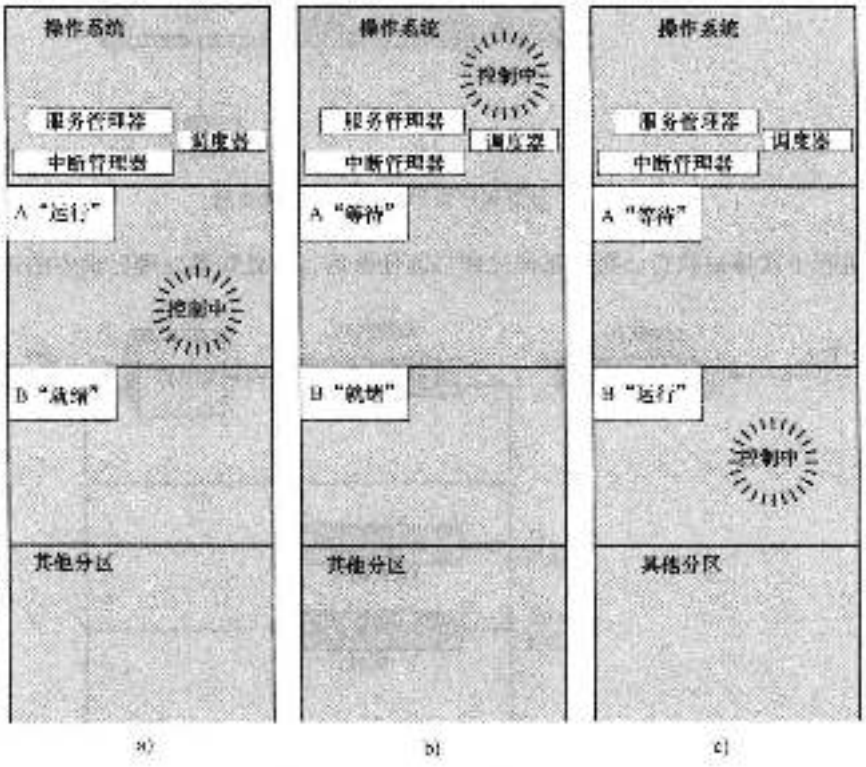


图8-9 调度举例

这个简单的例子描述了短期调度程序的基本功能。图8-10表示了涉及多道程序设计和进程调度的操作系统中的主要元素。当出现一个中断时，操作系统接收处理器的控制权，执行中断处理程序；当出现一个服务调用时，执行服务调用处理程序。一旦中断或服务调用处理完毕，短期调度立即调度某个进程进入运行状态。

为了实现进程调度机制，操作系统需要维护多个队列，每个队列都是一个简单的等待某些资源的进程列表。长期队列是等待系统资源的作业列表，若条件许可，高级调度程序将为等待的作业分配内存，并创建一个进程。短期队列包含所有就绪状态的进程，它们中某一个可能下次被处理器调用。短期调度程序将从其中挑选一个进程，通常采用轮转算法，给每个进程轮流分配一些时间，或者采用优先级算法。最后，每个I/O设备有一个I/O队列，多个进程可以请求使用相同的I/O设备，等待使用每个设备的所有进程排列在那个设备队列中。

图8-11给出了在操作系统控制下的进程流程。每个进程请求（批作业、用户定义的交互式作业）都被放置在长期队列中，当某一进程的请求满足时，此进程转为就绪状态并进入短期队列。处理器交替执行操作系统的指令和用户程序指令。当操作系统获得控制权时，它决定短期队

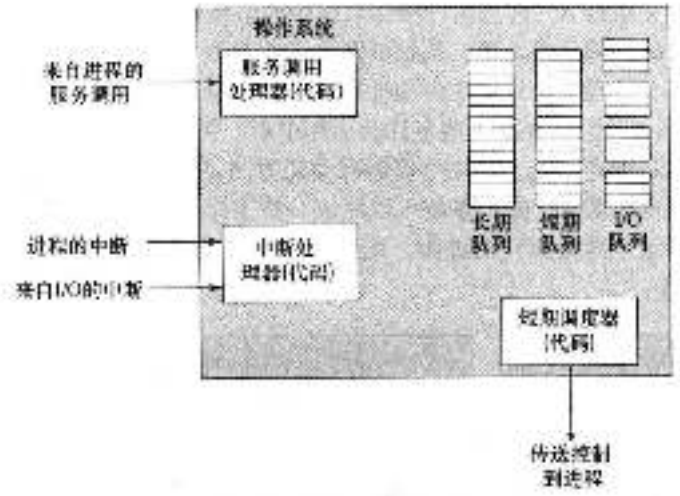


图 8-10 多道程序设计的操作系统的要素

列中的那个进程下次将被执行。操作系统完成当前任务后，由处理器处理已选中的进程。

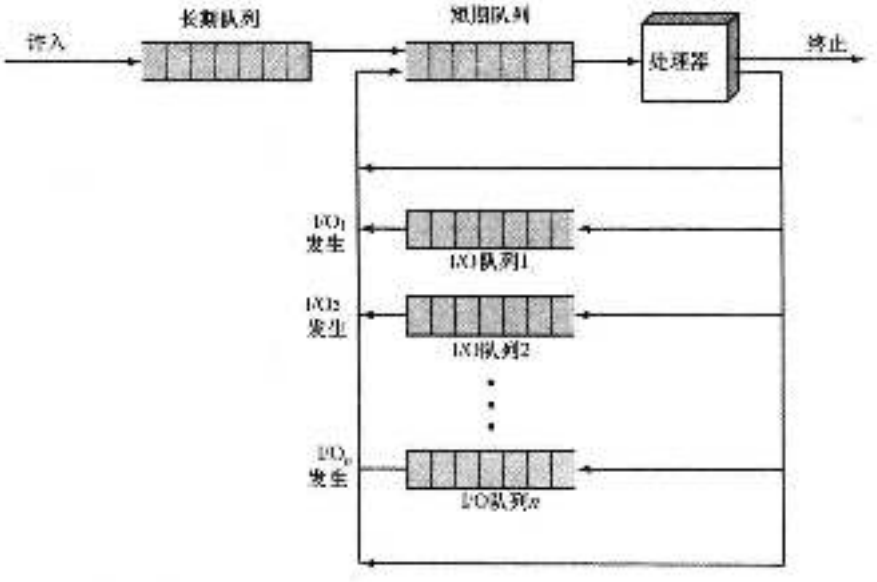


图 8-11 处理器调度的队列图

前面已经提到，正在执行的进程由于一些原因可能被挂起。如果挂起的原因是进程请求 I/O 服务，则它会进入相应的 I/O 队列。如果挂起的原因是超时或操作系统必须处理紧急事务，则它处于就绪状态，进入短期队列。

最后，需要说明的是操作系统也管理 I/O 队列。当一次 I/O 操作完成时，操作系统从 I/O 队列中移出 I/O 请求得到满足的进程，并将其放入短期队列中，然后它选择另一个等待进程（如果有的话），指定 I/O 设备来满足进程的请求。

8.3 存储器管理

在单道程序设计系统中，主存划分成两大部分：一部分分配给操作系统（常驻监控程序），另一部分分配给当前正在执行的程序。在多道程序设计系统中，存储器的“用户”存储区需进

一步细分供多个进程使用。细分存储器的任务由操作系统动态地执行，并称为存储器管理。

在多道程序设计系统中，有效的管理存储器是很重要的。如果在存储器中只有很少的进程，进程可能会花费很多时间来等待 I/O 而使处理器常处于空闲状态。因此，存储器需要合理分配，尽可能让更多的进程进入存储器。

8.3.1 交换

再看图 8-11，我们已讨论过 3 种队列：请求新进程的长期队列、进程就绪等待处理器的短期队列以及进程未就绪的 I/O 队列。重提一下，这种处理方法的原因是 I/O 操作比处理器计算慢得多，因此，在单道程序设计系统中，处理器大多数时间处于空闲状态。

图 8-11 中的列队不能完全解决这个问题。事实是：在这种情况下，存储器保存了多个进程，当一个进程等待时，处理器能转去处理另一个进程。但是处理器速度比 I/O 快得多，以至于存储器中所有进程总是在等待 I/O 操作。因此，即使在多道程序设计中，处理器仍可能有许多时间处理空闲状态。

解决的方法是什么？扩充主存以能够容纳更多的进程。但这种方法有两个缺点：首先是主存很贵，即使是现在。其次程序对存储器容量的需求增长很快，如同存储器价格下降一样快。因此，更大的存储器扩充导致更大的进程，而不是更多的进程。

另一种方法是交换，如图 8-12 所示。有一个进程请求的长期队列，通常存储在磁盘上，当主存有空间时，进程被调入，每次调入一个；当进程完成时，移出主存。现在将出现一种情况，存储器中无进程处于就绪状态（例如，所有进程都在等待 I/O 操作）。这时，处理器不是保持在空闲状态，而是把这些进程中的一个调回磁盘，排入中间队列，它是用来排列临时从内存调出的进程的队列。然后，操作系统从中间队列中调入另一个进程，或处理长期队列中的一个新进程请求，并执行新到达的进程。

然而，交换是一种 I/O 操作，它有可能使问题变坏而不是变好，但是，磁盘 I/O 通常是系统中最快的 I/O（与磁带和打印设备相比），所以交换通常能够提高性能。另一种更复杂方案是虚拟存储器，它比简单的交换更能提高性能，我们将在后面对其做简要的介绍。接下来我们介绍分区和分页。

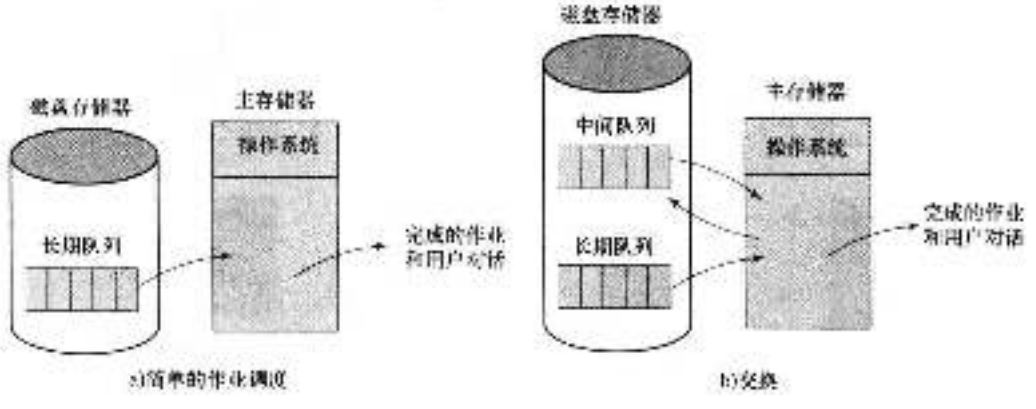


图 8-12 交换技术的运用

8.3.2 分区

最简单的分区机制是固定长度的分区，如图 8-13 所示。注意，虽然分区是固定长度，但每个分区的长度可以不相等。当一个进程调入主存时，分配给它一个能容纳它的最小的分区。

尽管使用了不等的固定长度分区，但也浪费了主存。在多数情况下，进程对分区大小的需求

不可能和提供的分区大小完全一样，例如，一个需要3MB存储空间的进程将被放置在图8-13b中的4MB分区中，这就浪费了能分配给其他进程的1MB的空间。

一种更有效的方法是使用变长分区，当一个进程调入主存时，分配的分区大小可以与进程所需的大小一样。

例8.2 图8-14给出一个使用64MB主存的例子。初始时，主存除了操作系统占据的一些空间外，其余为空（如图8-14a所示）。首先把3个进程调入主存，在操作系统区后连续分配3个空闲区给这3个进程（如图8-14b、c和d所示），结果在存储器尾部留下一个“空块”，因为它很小，不能装入第4个进程。某一时刻，没有进程进入就绪状态。然后，操作系统交换出进程2（如图8-14e所示），这使得有足够的空间来装载新进程，进程4（如图8-14f所示）。由于进程4比进程2小，因此又产生了一个“空块”。随后，某一时刻，再次出现主存中的进程没有一个进入就绪状态的情况，而此时进程2进入就绪挂起状态。因为主存中没有足够的空间给进程2，于是操作系统要先换出进程1（如图8-14g所示），再换入进程2（如图8-14h所示）。

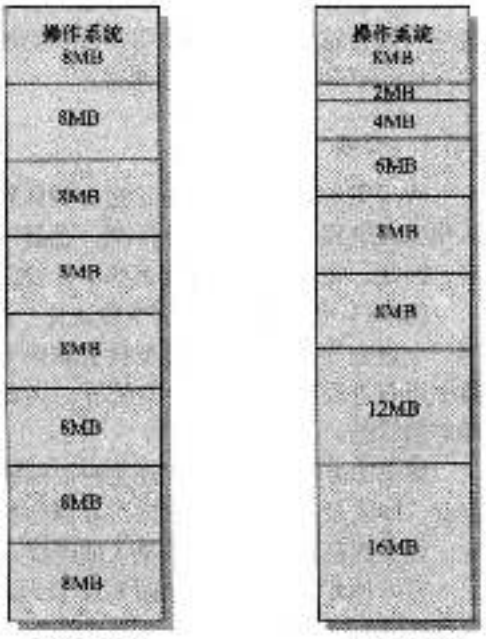


图8-13 64MB存储器固定分区举例

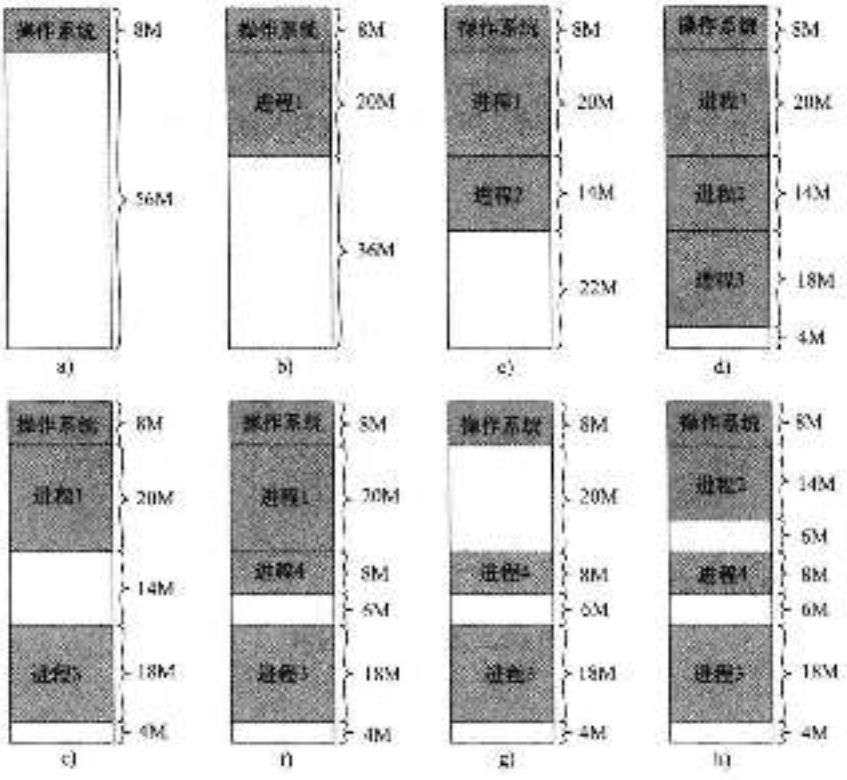


图8-14 动态分区的效果

这个例子表明：这种方法开始时比较好，但到最后它将导致在存储器中出现许多小的空块。时间越长，存储器中的碎片就会越来越多，而使存储器的利用率下降。解决这个问题的一种技术是紧缩。操作系统一次又一次地移动存储器中的进程，把所有空闲块放置在一起，组成一个块。但是这个过程很费时，会浪费处理器的时间。

在考虑克服分区缺点的方法前，必须弄清一个问题。如果读者对图 8-14 稍加注意，就可以看到，一个进程每次换入时不可能分配到与上次相同的位置，而且如果采用紧缩技术，则进程在主存中可以移动。主存中的进程由指令和数据组成，指令有两种类型的存储器单元的地址：

- 数据的地址。
- 指令的地址用于转移指令。

但是这些地址是不固定的，进程每交换一次，地址都将发生变化。为了解决这个问题，我们把地址分为逻辑地址和物理地址。**逻辑地址**表示相对于程序起始单元的地址，程序中的指令只包含逻辑地址。**物理地址**是主存中的实际单元地址。当处理器执行一个进程时，通过把当前进程的起始单元地址（称为**基址**）加到每个逻辑地址上，自动地把逻辑地址转换成物理地址。这种逻辑地址与物理地址的转换机制也是为了满足操作系统的需求而对处理器硬件特性设计的另一个例子，这种硬件特性的精确度取决于使用的存储器管理策略。我们将在本章的后面介绍一些相关例子。

8.3.3 分页

不等的固定长度分区和可变长度分区，其存储器的使用效率都很低。假设把存储器分成相当小的、相等的固定长度的存储块，将每个进程也划分成小的固定长的程序块，那么程序的每个程序块（称为页）能分配到存储器中可用的每个存储块（称为**帧**或**页帧**）中，于是，存储器分配进程后浪费的空间最多只是最后一页的一小部分。

图 8-15 表示了一个使用页和页帧的例子。在某一时刻，存储器中的一些帧被占用，而另一些帧处于空闲状态。空闲帧的列表由操作系统维护。存储在磁盘上的进程 A 由 4 个页组成。当操作系统装入这个进程时，它发现有 4 个空闲帧，并把进程 A 的 4 个页装入这 4 个空闲帧中。

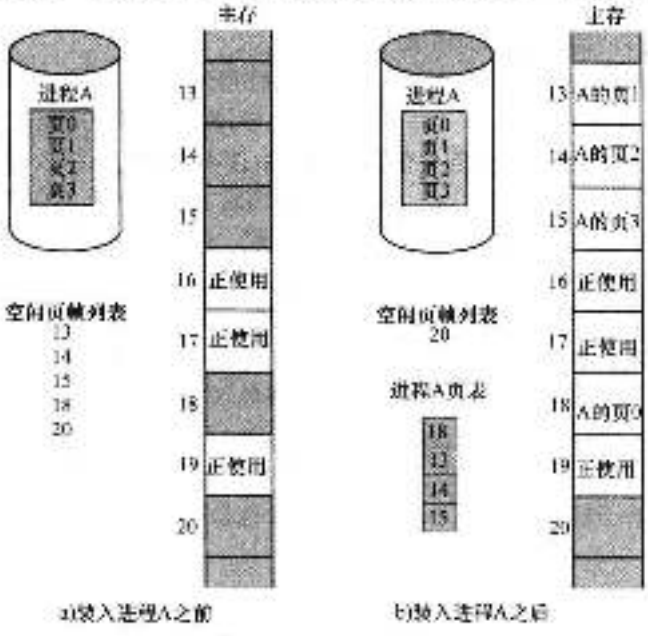


图 8-15 空闲页帧的分配方案

现在假设在此例中，无足够的连续空闲帧存储这个进程。这会妨碍操作系统装入进程A吗？答案是否定的，我们可以再一次引入逻辑地址这一概念，只是简单的基地址已经不够用了。操作系统为每个进程保存一个页表，页表记录了进程每页的帧地址。在程序中，每个逻辑地址由一个页号和页中相对地址组成。回忆一下简单分区的情况，逻辑地址是相对于程序起始地址的存储单元地址，用一个字表示，处理器负责将逻辑地址转换为物理地址。使用分页技术，逻辑地址到物理地址的转换仍由处理器硬件来完成，但是处理器必须知道怎样访问当前进程的页表。提供了逻辑地址（页号、相对地址）后，处理器使用页表来产生物理地址（帧号、相对地址）。图8-16显示了这样的一个例子。

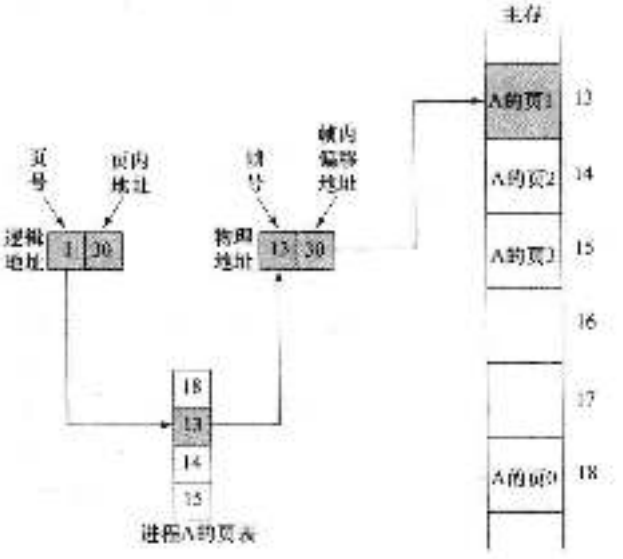


图8-16 逻辑地址与物理地址

这种方法解决了前面提出的问题。主存划分成许多小的大小相等的帧，每个进程划分成与帧大小一致的许多页。较小的进程需要较少的页，较大的进程需要较多的页。当一个进程调入时，其页被装入空闲帧中，并建立页表。

8.3.4 虚拟存储器

1. 请求分页

分页技术使多道程序设计系统变得真正有效，而且把进程分页这一简单策略导致了另一重要概念的产生——虚拟存储器。

为了理解虚拟存储器，我们对刚才讨论的分页方案进行改进。改进的方案称为请求分页，即一个进程的每个页只有在需要时（即请求时）才调入。

考虑一个大的进程，它由一段长的程序和许多组数据组成。在任何一小段时间内，程序执行时只需使用其中的一小部分（例如一个子过程）或一到两组数据。这就是局部性原理，它在附录4A中介绍过。很显然，在程序挂起之前只用到其中很少的页时，装入进程的全部页是一种浪费，装入适当的页才能更好地利用主存。因此，当程序转去执行一个不在主存中的页的指令，或者程序访问不在主存中的页的数据时，则会触发页失效。这就告诉操作系统该调入所需的页。

于是，在任何时候，进程中只有一小部分页在主存中，因而可以使更多的进程同时占用主存，由于未使用的页不用进行换入换出操作，因此节省了系统时间。然而，操作系统必须管理这种管理机制，当它调入一页时，必须把另一页换出去，这称为页替换。如果它换出的页正好是将要使用的页，则又要立即把它调入主存，如果这种现象频繁发生则称为抖动，这时，处理器花费大量的时间处理页替换而不是执行指令。避免系统抖动是20世纪70年代主要研究的课题，并产生了许多有效但复杂的算法。基本上，操作系统是基于最近的页访问历史，认为最近最少使用的页将来也不大可能使用。



页替换算法超出了本章的范围，目前比较有效的技术是最近最少使用算法（LRU），在第4章高速缓存的替换算法中会介绍这个算法。实际上，LRU算法在虚拟存储的页替换机制中很难实现，因此通常是使用一些性能近似于LRU的算法，具体请参见附录F。

采用了请求分页，就没有必要将整个进程装入主存，这将产生一个惊人的结论：一个进程可能比主存所有的空间都大。程序设计中最基本的限制之一已经被打破了。若不采用请求分页，则程序员必须知道有多大的存储空间可用，如果编制的程序太长，程序员必须修改结构使程序变成几段，一次装入一段。而采用了请求分页，这些工作交给了操作系统和硬件去做，就程序员而言，他所看到的是一个大的存储器，其容量与磁盘存储器有关。

因为进程只在主存中执行，所以主存被称为**实存储器**。但是程序员或用户看到了一个大得多的存储器，它分配在磁盘上，后者称为**虚拟存储器**。虚拟存储器使多道程序设计更为有效，同时消除了用户使用主存的限制。

2. 页表结构

从存储器中读取一个字的基本机制包括：通过页表把虚拟或逻辑地址（由页号和偏移量组成）转换成物理地址（由帧号和偏移量组成）。因为页表是可变长的，与进程长度有关，所以我们不能期望将它存入寄存器中，而必须将它存入主存中。图 8-16 说明了这个方案的硬件实现。当运行一个特定进程时，寄存器保存这个进程页表的起始地址，虚拟地址的页号用于检索页表并寻找相应的帧号，它与虚拟地址的偏移部分结合来形成物理实地址。

大多数系统中，每个进程对应一个页表，但每个进程能占据大量的虚拟存储器。例如 VAX 体系结构中，每个进程能拥有高达 $2^{31}=2$GB 的虚拟存储，使用 $2^{9}=512$B 的页。这意味着每个进程需要 2^{22} 个页表项，显然，单纯分配给页表的存储器容量就高得惊人。为了解决这个问题，许多虚拟存储器方案把页表存储在虚拟存储器中，而不是实存中。这表明对页表也要进行与其他页一样的分页。当一个进程正在运行时，至少有一部分页表（包括当前正在运行页的页表项）必须在主存中。一些处理器使用二级方案组织大的页表。在此方法中，有一个页目录，每项指向一个页表。因此，如果页目录表长度是 X，每个页表的最大长度为 Y，则一个进程包括 X × Y 页。通常一个页表的最大长度可以达到一页。在这章的后面介绍 Pentium II 时，我们将看到二级方案的例子。

实现一级或二级页表的另一种方法是倒置页表结构（如图 8-17 所示），这种办法的几种不同版本已用于 PowerPC，UltraSPARC 和 IA-64 体系结构。IBM-PC 上的 Mach 操作系统的实现亦使用了这种技术。

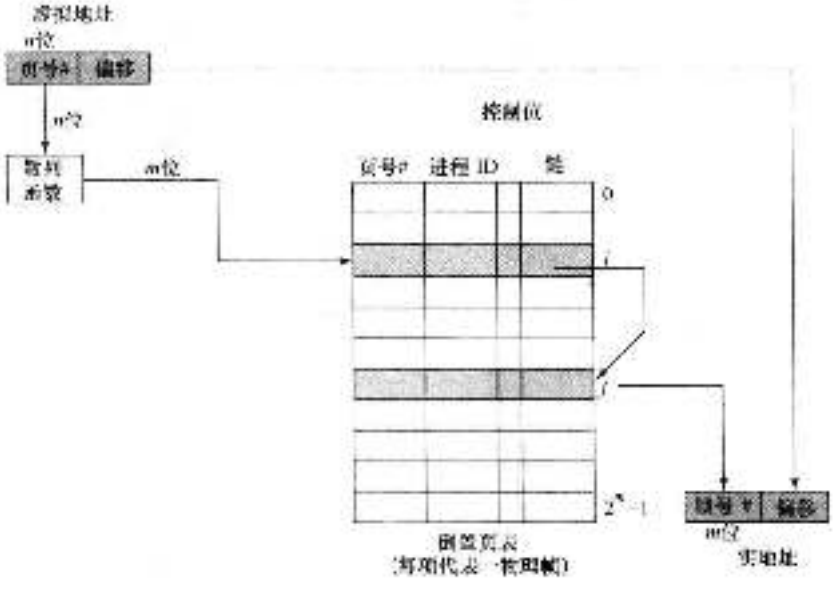


图 8-17　倒置页表结构

在这种技术中，虚拟地址的页号部分采用简单的散列函数[1]映射到散列表中，散列值是一个指向由页表项组成的倒置页表的指针。采用这种结构，每个实存帧在散列表和倒置页表中都有对应项，但虚拟页不需要，因此，表只需要实存的固定部分，而不考虑所支持的进程数或虚拟页数。因为多个虚拟地址可以映射到同一散列表项，所以采用链技术来管理溢出，散列技术通常采用由一项或两项组成的短链。此页表的结构称为倒置，因为它用帧号而不是虚拟页号来检索页表项。

8.3.5 快表

原则上，每次虚拟存储器的访问能引起两次物理存储器的存取：一次是获得相应的页表项，另一次是获得所需的数据。因此，即使一个简单的虚拟存储器方案也将使存储器存取时间加倍。为了解决这个问题，许多虚拟存储器方案使用一个特殊的高速缓存来存放页表项，通常称为快表（TLB）。这个高速缓存功能与存储器中高速缓存的相同，它用来存储最近使用的那些页的页表项。图 8-18 描述了使用 TLB 的流程图。按照局部性原理，大部分虚拟存储器的访问局限在最近使用过的那些页，因此，多数访问都是此高速缓存中记录的页。对 VAX 机中 TLB 的研究表明，这个方案能有效地改善性能［CLAR85，SATY81］。

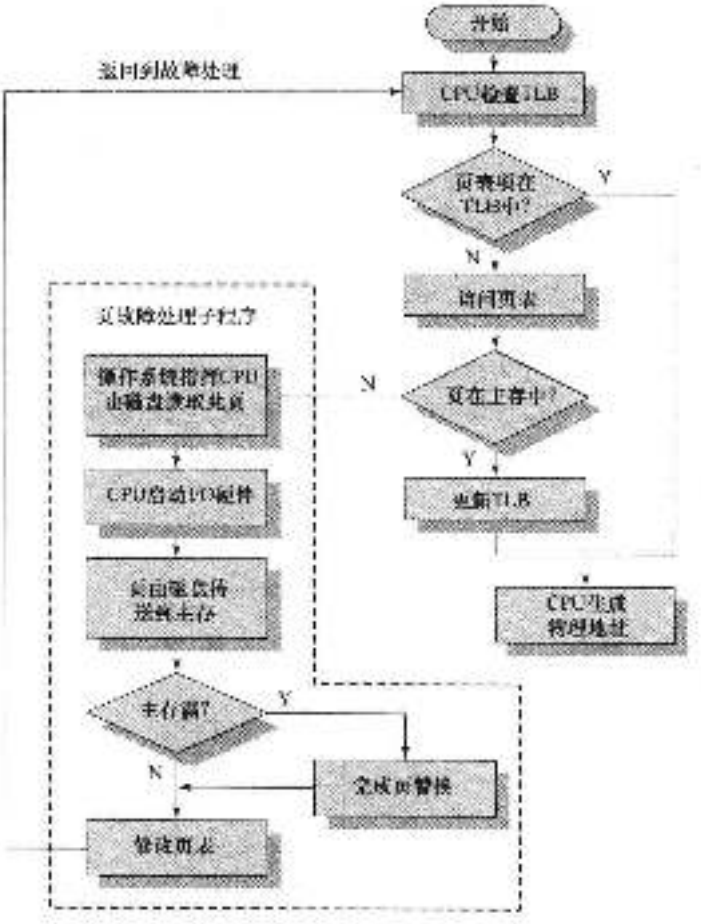


图 8-18 分页操作和快表（TLB）

[1] 散列函数是将范围 0～M 的数映射到范围 0～N 的数，这里 M＞N。散列函数的输出被用作为对散列表的索引。因为多于一个的输入映射到同一输出，因此可能存在一个输入项映射到已被占据的散列表项。在这种情况下，新项必须"溢出"到另外的散列表位置上。一般来说，新项被放在每一个后端的空项中，由原位置提供一个指针指向它，从而把它们链接在一起。附录 C 提供了有关散列函数的更多内容。

注意，虚拟存储器机制必须与高速缓存系统（不是 TLB 的高速缓存，而是主存高速缓存）交互，如图 8-19 所示。虚拟地址通常由页号和偏移量组成，首先存储器系统查询 TLB 是否有匹配的页表项，如果有，则实（物理）地址通过帧号和偏移量组合产生；如果没有，则从页表中取项。一旦由标志项和剩余项组成（如图 4-5 所示）的实地址产生，先查询高速缓存中是否有包含该字的块。如果有，字返回给处理器，反之，则从主存中检索此字。

读者可能体会到单个存储器访问所涉及的处理器硬件的复杂性。虚拟地址转换成实地址，这涉及访问页表，这个页表可能在 TLB、主存或磁盘上。然后要访问字，这字也可能在高速缓存、主存或磁盘上，如果是在磁盘上，则还需要将包含该字的页调入主存，把它的块装入高速缓存中。此外，还要更新该页的页表项。

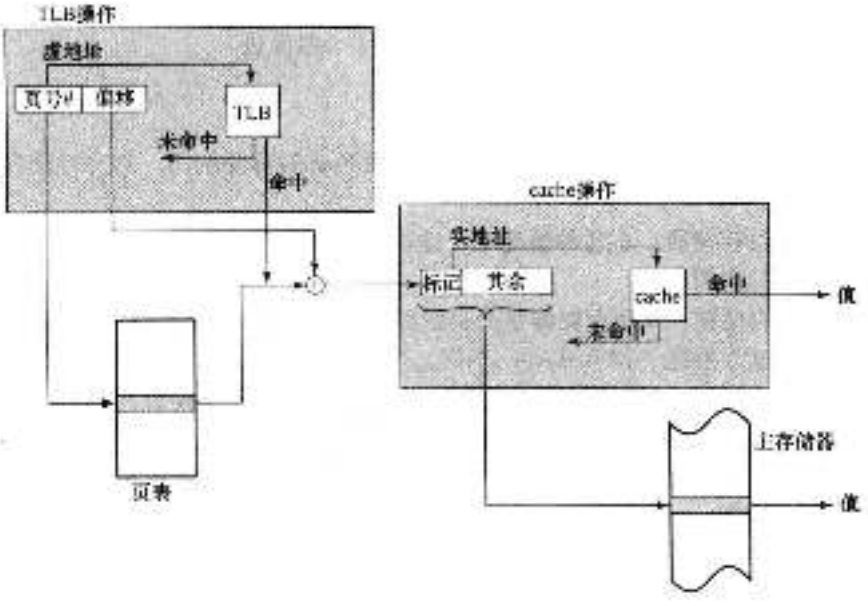


图 8-19 TLB 和 cache 操作

8.3.6 分段

另一种划分可寻址存储器的方法是分段（segmentation）。分页对程序员是不可见的，其目的是为程序员提供较大的地址空间。而分段通常对程序员是可见的，它使组织程序和数据更方便，能将特权、保护属性与指令、数据联系起来。

分段后，存储器由多个地址空间或段组成。段长度是可变、可动态分配的。通常，程序员或操作系统为程序和数据分配不同的段。各程序可以有许多程序段和数据段，各段可指定分配存取权和使用权，存储器访问地址由段号和偏移量组成。

对程序员来讲，分段的地址空间有许多优点：

（1）它简化了对数据结构的处理。如果程序员预先不知道其数据结构有多大，他也不必去猜测。该数据结构可以分配它自己的段，并且操作系统会按数据结构的需要动态地扩大或缩小这个段。

（2）将程序分段允许每段程序独立地修改和重编译，不需要整个程序重新连接和重装入，即使用多个段来完成。

（3）可以实现进程共享。程序员将某程序或数据表放入一段，其他进程也可以访问该段内容。

（4）可以实现段保护。一个段可能包含一个完好定义的程序或数据集，程序员或系统管理

员能方便地赋予该段存取特权。

这些优点在分页中是没有的，因为分页对程序员是不可见的。另一方面，分页提供了存储器管理的有效方式。为了兼并这两者的优点，有些操作系统同时提供了分页和分段两种方式的硬件和软件支持。

8.4 Pentium 存储器管理

引入了 32 位体系结构后，微处理器已经可以实现复杂的存储器管理机制，这些机制只在大、中规模系统中实现。在很多情况下，微处理器比其先前的较大系统版本更胜一筹。由于存储器管理方案是由微处理器硬件厂家开发的，因而可适应于各种操作系统。随着微处理器版本的不断升级，它们趋向于通用。一个典型的例子是 Pentium II 的存储器管理，Pentium II 的存储管理硬件基本上与 Intel 80386 和 80486 处理器相同，只是稍做了一些改进。

8.4.1 地址空间

Pentium II 包含分段和分页的硬件支持，同时，两种机制都能禁用。它允许用户从如下 4 种不同方式中任选一种：

- **不分段不分页存储器**：在这种情况下，虚拟地址和物理地址相同，适用于低复杂性、高性能控制器的应用。
- **分页不分段存储器**：这些存储器是分页的线性地址空间，存储器的保护和管理通过分页实现。某些操作系统（如 Berkeley UNIX）采用这种方法。
- **分段不分页存储器**：可以把这种存储器看成逻辑地址空间的集合。相对于分页，其优点是：如果需要，它可以提供低至字节级的保护机制；而且，它可以保证当段在存储器中时，需要的转换表（段表）在芯片上。因此，分段不分页存储器可以预测存取时间。
- **分段分页存储器**：分段用于存储器逻辑分区，分页用于逻辑分区的物理再分配，如同 UNIX System V 的操作系统就采用这种方式。

8.4.2 分段

分段时，每个虚拟地址（在 Pentium II 中称为逻辑地址）由 16 位段号和 32 位偏移量组成，段号中有两位用于保护机制，余下的 14 位表示一个具体的段。因此，对于无分段的存储器，用户的虚拟存储空间是 2^{32} =4GB。而在有分段的存储器中，用户总的虚拟存储空间为 2^{46} =64TB。物理地址空间采用 32 位地址，最大为 4GB。

虚拟存储器容量实际上能大于 64TB，这是因为处理器对虚拟地址的译码取决于当前那个进程是活动的。虚拟地址空间被分成两部分：一半（8K 段 ×4GB）是全局的，被所有进程共享；另一半是局部的，每个进程都不同。

段的两种保护机制是：优先级和存取属性。优先级有 4 种，从最高（0 级）到最低（3 级）。与数据段有关的优先级称为“等级”，与程序段有关的优先级称为“许可”。当正在执行的程序许可级低于（更高特权）或等于（相同特权）数据段的优先级时，它可存取此数据段。

硬件没有规定这些优先级的分配，它取决于操作系统的设计和实现。特权级 1 通常用于大多数操作系统，第 0 级只有少量操作系统用于负责存储器管理、保护和存取控制，剩下的两级用于应用。在许多系统中，应用处于第 3 级，而第 2 级不用。一些特殊的应用子系统，由于它们要实现自己的安全机制，因此必须受到保护，它们会用到第 2 级，例如数据库管理系统、办公自动化系统和软件工程环境系统。

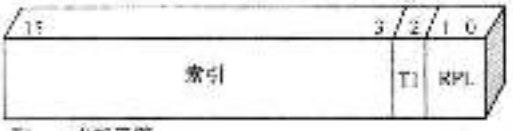
除了约束数据段的存取外，优先级机制还用于某些指令。一些指令（如处理存储管理寄存器的指令）只能在 0 级执行。I/O 指令只能在操作系统指定的某一级上执行，通常是第 1 级。

数据段的存取属性表明是否允许读/写或只读。对于程序段，存取属性表示读/执行或只读。

段地址转换机制把虚拟地址映射成线性地址（如图 8-20b 所示）。虚拟地址由 32 位的偏移量和 16 位的段选择符组成（如图 8-20a 所示），段选择符又包含下列域：

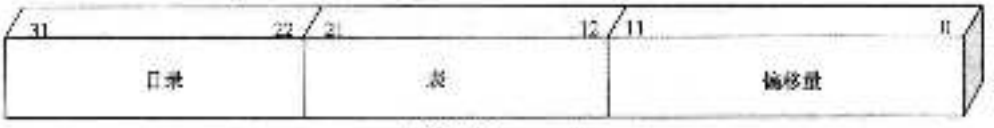
- 段表指示符（TL）：指示转换使用的是全局段表或者局部段表。
- 段号：表示第几段，在段表中用作索引。
- 请求优先级（RPL）：存取请求的优先级。

段表中的每一项由 64 位组成，如图 8-20c 所示，域定义如表 8.5 所示。

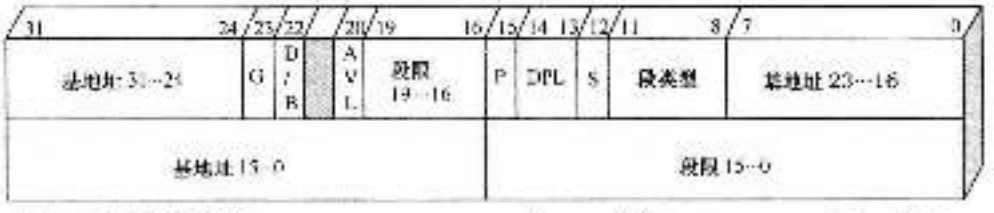


TI　－表指示符
RPL　－请求优先级

a) 段选择符

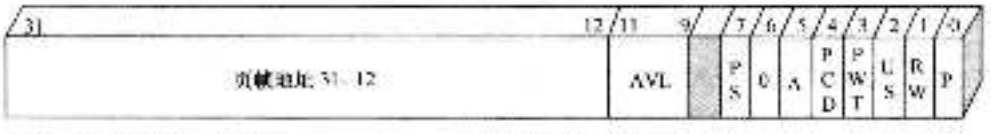


b) 线性地址



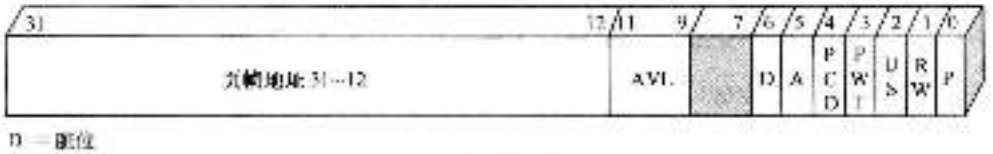
AVL　－可被系统软件使用
Base　－段基址
D/B　－默认的操作大小
DPL　－描述符优先级
G　－粒度
Limit　－段限
P　－段出现
S　－描述符类型
□ －保留

c) 段描述符（段表项）



AVL　－可被系统程序员使用
PS　－页大小
A　－访问过
PCD　－cache 禁止
PWT　－写直达
US　－用户 / 监督
RW　－读 / 写
P　－出现

d) 页目录项



D　－脏位

e) 页表项

图 8-20　Pentium 存储器管理格式

8.4.3 分页

分段是一种可选特性，可以禁止。使用分段时，程序使用的是虚拟地址，它要转换成线性地址。分段禁止时，在程序中直接使用线性地址。因此，无论是否分段，下一步的工作都是将线性地址转换成32位的实际地址。

为了理解线性地址的结构，必须知道 Pentium II 的分页机制实际上是一种二级表的检索操作。第1级是一个包含1024个项的页目录，它将4GB的线性存储空间分成1024个页组，每个组有自己的页表。每个页组长度为4MB。每个页表包含1024个项，每项对应一个4KB的页。存储器管理有选择地使用页目录，由所有进程共用一个页目录，或者每个进程一个页目录，或者两者的组合。当前任务的页目录总是在主存中，页表可以存储在虚拟存储器中。

图8-20表示页目录项和页表项的格式，域定义如表8-5所示。注意，存储器控制机制可用于一个页或一个页组。

表8-5 Pentium II存储管理参数

段描述符（段表项）
基（base） 定义段在4GB的线性地址空间中的起始地址。
D/B位 在代码段中，这是D位，表示操作数和地址模式是16位还是32位。
描述符优先级（DPL） 指定此段描述符指向的段的优先级。
粒度位（G） 表示段界域是以1B还是4KB为单位来说明。
段限 定义段的大小，处理器以两种方法之一说明段界域。它的大小取决于粒度位：在以字节为单位时，段段长度为1MB；以4KB为单位时，段限长度最大为4GB。
S位 确定某一给定的段是系统段，代码或是数据段。
段出现位（P） 用于不分页的系统中时，它指明段是否在主存中；对于分页系统，这一位总是设置为1。
类型 区别各种段，表示存取属性。
页目录项和页表项
存取位（A） 当读或写操作出现在相应页时，处理器将两级页表中的此位设置为1。
脏位（D） 当写操作出现在相应页时，处理器将此位设置为1。
页帧地址 如果出现位（P）置位，则提供该页在存储器中的物理地址。由于页帧分成4KB大小，因此低12位为0，只有高20位包含在项中。在页目录中，该地址是页表地址。
页cache禁止位（PCD） 说明页中的数据是否可经过高速缓存。
页大小位（PS） 表示页大小为4KB或4MB。
页写直达位（PWT） 表示此页的数据是否采用写直达或回写的cache写策略。
出现位（P） 表示此页表或页是否已在主存中。
读/写位（RW） 对于用户级的页，它表示用户级程序的页是只读存取或是读/写存取。
用户/管理位（US） 表示该页是只在操作系统级（管理级）可用还是在操作系统和应用级（用户级）都可用。

Pentium II也使用快表，该快表能保存32个页表项。每当页目录变化时，快表被清空。图8-21表示分段和分页的组合。为了清晰，快表和存储器高速缓存机制未显示出来。

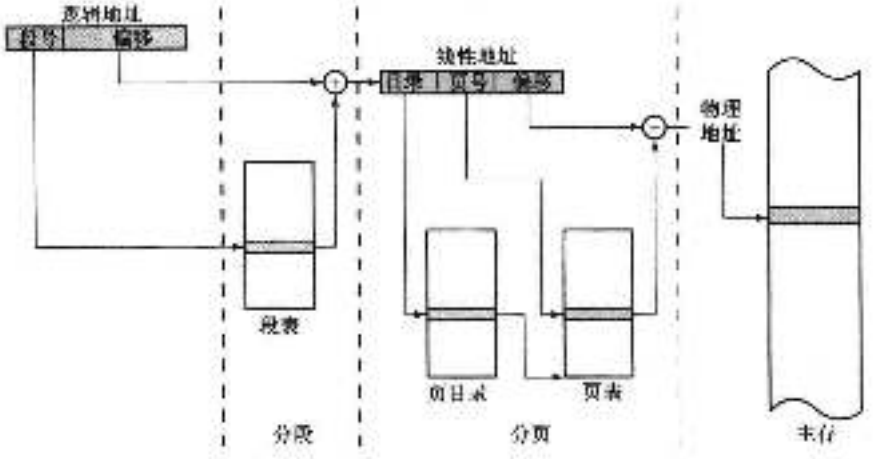


图8-21 Pentium存储器地址转换机制

最后，Pentium II扩展了80386或80486，它提供了两种页长度。如果在控制寄存器4中的页长度扩展位（PSE）置为1，则分页单元允许操作系统程序员定义页长度为4KB或4MB。

使用4MB的页时，只有一级表检索。当硬件存取页目录时，页目录项（如图8-20d所示）使PS位置为1。在这种情况下，第9到21位被忽略，而第22到31位定义存储器中一个4MB页的基地址。从而，这是一个单页表。

使用4MB的页可减少存储器管理本身对大量主存的需求。使用4KB的页，整个4GB的主存大约需要4MB的存储器来存放页表；而使用4MB的页，单个4KB长度的表便足以满足页存储管理的存储需求。

8.5 ARM存储器管理

ARM提供了一种通用的虚拟存储器系统结构，它通过裁剪满足嵌入式系统设计者的需要。

8.5.1 存储器系统组织

图8-22是ARM中虚拟存储器的存储器管理硬件的概述图。如随后所述，虚拟存储器转换硬件采用一级或二级表将虚拟地址转换成物理地址。快表（TLB）存储最近使用的页表项，如果将要调用的页表项在TLB中，则TLB直接将该页的物理地址发给主存用于读/写操作。正如第4章所述，处理器与主存之间的数据交换是通过高速缓存进行的。如果采用了逻辑高速缓存（如图4-7a所示），则当高速缓存访问失效时，ARM直接提供地址给高速缓存同时也提供给TLB；如果采用了物理高速缓存（如图4-7b所示），则TLB必须将物理地址提供给高速缓存。

转换表中的项也包含存取控制位，它决定进程是否可以存取某一段主存。如果不允许存取，则访问控制硬件向ARM处理器发送存取终止信号。

8.5.2 虚拟存储器地址转换

ARM存储器存取基于分节或分页来支持存储器访问：

- 超级节（可选）：由主存的16MB块组成。
- 节：由主存的1MB块组成。
- 大页：由主存的64KB块组成。

- 小页：由主存的4KB块组成。

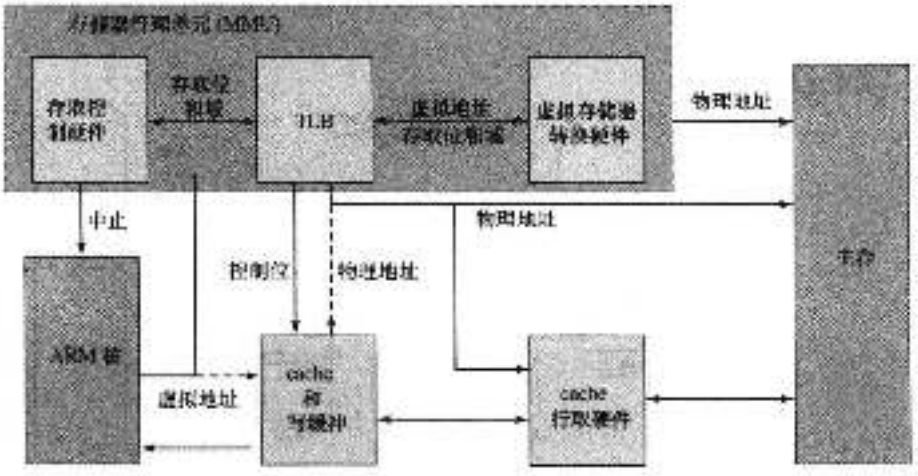


图 8-22 ARM 存储器系统

节和超级节可以使用 TLB 中单个表项映射一大片主存区域，而且存取控制机制可以将小页扩展到 1KB 的子页，将大页扩展到 16KB 的子页。保存在主存中的转换表有两级：

- **第一级表**：包含节和超级节的转换地址，以及指向第二级表的指针。
- **第二级表**：包含大页和小页的转换地址。

存储器管理单元（MMU）将处理器产生的虚拟地址转换为访问主存的物理地址，并驱动和检查存取许可位。如果 TLB 发生页缺失，则需要转换地址，首先在第一级表中取转换地址。节映射存取只需要在第一级表中取转换地址，而页映射存取还需要在第二级表中取转换地址。

图 8-23 表示小页的两级地址转换过程。一级（L1）页表包含 4K 个 32 位的表项，每个 L1 表项指向一个二级（L2）页表；二级页表包含 255 个 32 位的表项。每个 L2 表项指向主存中一个大小为 4KB 的页。32 位的虚拟地址结构如下：最高 12 位是指向 L1 页表的索引，接下来的 8 位指向相应的 L2 页表地址，最低的 12 位指向主存中相应页的一个字节。

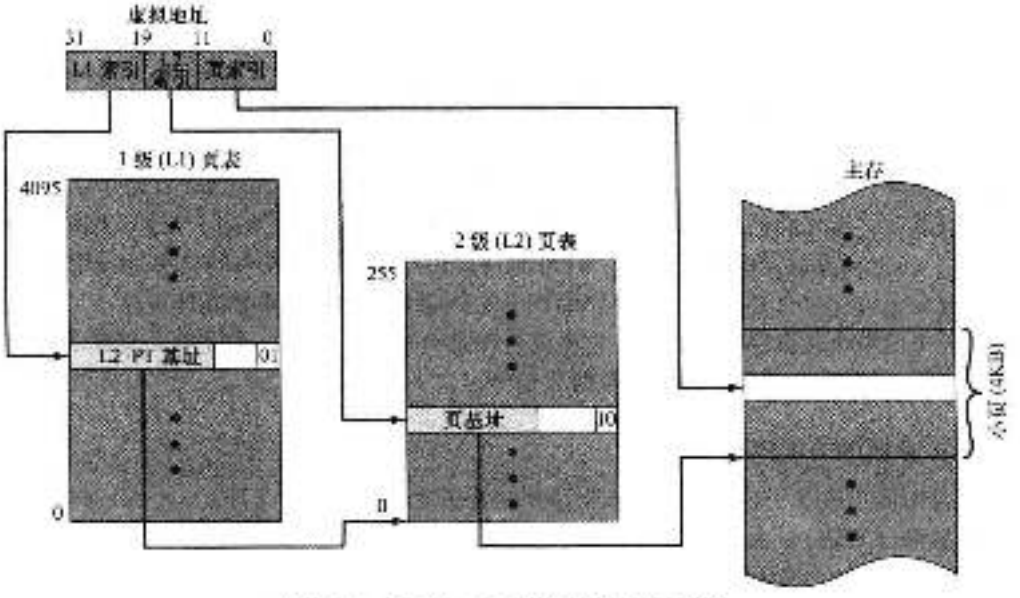


图 8-23 小页的 ARM 虚拟存储器地址转换

一个类似的二页地址转换适用于大页，而节和超级节只需查找 L1 页表。

8.5.3　存储器管理格式

为了更好地理解 ARM 的存储器管理机制，我们考虑其主要的存储器管理格式，如图 8-24 所示。图中控制位的定义参见表 8-6。

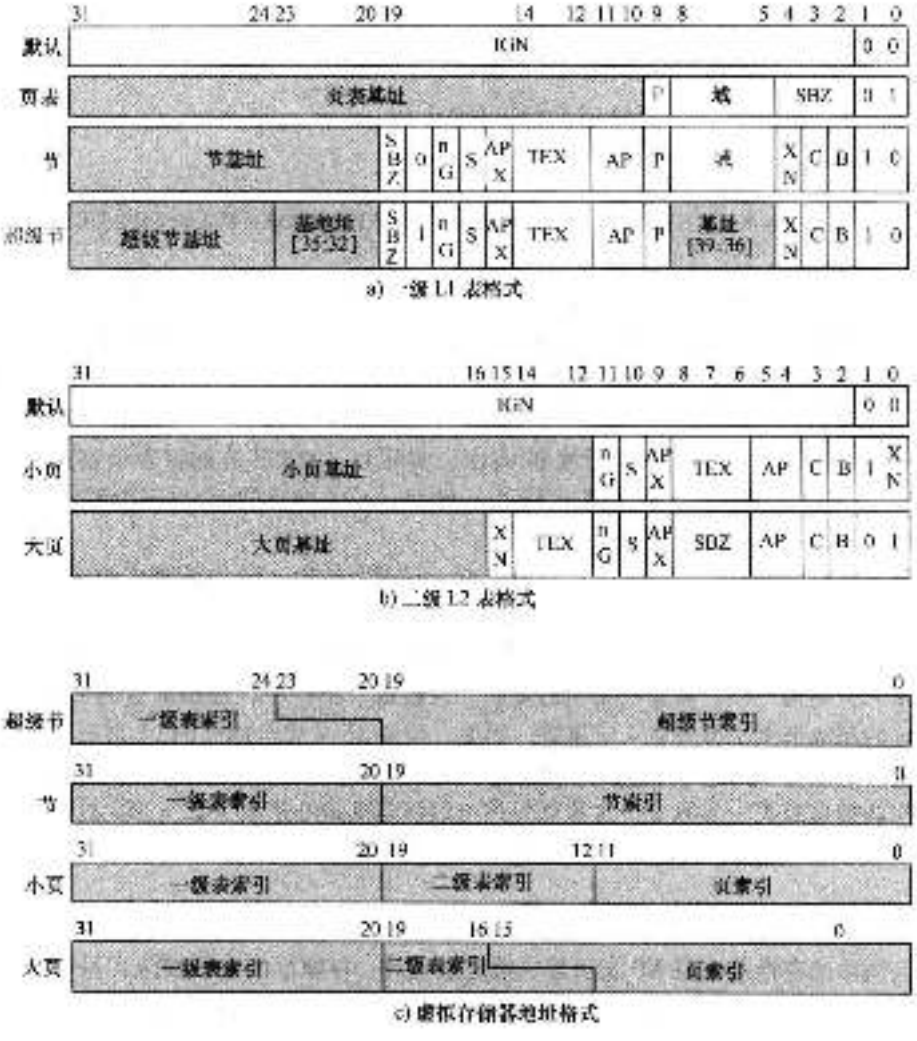


图 8-24　ARMv6 存储器管理格式

对于 L1 页表，每一个表项表示它与 1MB 的虚拟地址的映射关系。每个表项有 4 种可选格式之一：

- 位[1:0] =00：对应的虚拟地址没有被映射，试图存取时会产生地址转换错误。
- 位[1:0] =01：表项给出 L2 页表的物理地址，它说明了对应虚拟地址的映射关系。
- 位[1:0] =01 且第 19 位为 0：表项表示的是对应虚拟地址的节映射。
- 位[1:0] =01 且第 19 位为 1：表项表示的是对应虚拟地址的超级节映射。

位[1:0] =11 的表项被保留。

对于分页的存储器结构，需要一个二级页表存取，L1 页表项的位[31:30]包含了一个指针指向一个 L1 页表。对于小页，L2 页表项中包含一个 20 位的指针指向主存中一个 4KB 页的基地址。

表 8-6 ARM 存储管理参数

访问允许（AP）位，访问允许扩展（APX）位： 控制程序的存储器访问。如果访问没有得到允许，则会引起FP可获取。
可缓冲（B）位： 和TEX位一起决定带高速缓存的存储器的写缓冲方式。
可高速缓存（C）位： 决定某一块存储区是否可以映射到高速缓存。
域（Domain）： 若干存储区的集合，存储控制可应用于基本域。
非全局（nG）位： 决定某一地址转换是否为全局（0）或为特定进程（1）。
共享（S）位： 决定某一地址转换是共享存储器（1）或是不共享存储器（0）。
SBZ位： 应该为0。
类型扩展（TEX）位： 与B和C位一起控制高速缓存的读取、写缓冲方式，以及如果某存储区域是可共享的则必须保证其一致性。
永不执行（XN）位： 决定某存储区是可执行的（0）还是不可执行的（1）。

大页的页表结构更为复杂。与小页的虚拟地址结构一样，大页的虚拟地址结构中有 12 位指向 L1 页表，8 位指向 L2 页表。对于 64KB 的大页，其虚拟地址的页索引部分必须是 16 位。为了使这种结构适用于 32 位的格式，4 位页索引域与 L2 页表域重叠。为了实现这种重叠，ARM 要求 L2 页表中的每一个支持大页的页表项复制 16 次。实际上，如果所有的页表项都指向大页，那么 L2 页表中的页表项就从 256 个减少到了 16 个。然而，一个给定的 L2 页表能够支持大页和小页的混合，因此需要进行大页项的复制。

节和超级节的存储器结构需要一级页表存取。对于节结构，L1 页表项中的位 [31:20] 包含一个 12 位的指针，指向主存中 1MB 节的基地址。

对于超级节结构，L1 页表项的位 [31:24] 包含一个 8 位的指针，指向主存中 16MB 节的基地址。就像大页结构一样，需要页表项的复制。在超级节结构中，虚拟地址的 L1 页表索引部分与虚拟地址的超级节索引部分有 4 位重叠。因此，需要 16 个完全相同的 L1 页表项。

物理地址范围可以通过 8 个额外的地址位（位 [23:20] 和位 [8:5]）进行扩展。扩展的范围与额外位的数量有关，实际上，这些额外位可以将物理地址扩展为 $2^8 = 256$ 的倍数。因此，对于每一个进程，可寻址的物理存储范围可以变成了原来的 256 倍。

8.5.4 存取控制

每个表项中的存取控制位 AP 表示某一给定进程对一存储区的存取权限，每块存储区都可以设置为不可访问、只读或者读写。进一步说，还可以将它设置为特权访问，即只有操作系统才可以访问，用户不可以访问。

ARM 也引入了“域”的概念，域是一批具有特殊访问权限的节和/或页。ARM 体系结构支持 16 个域，域特征允许多个进程使用同一个转换表而不会相互影响。

每个页表项和 TLB 项都包含一个字段，用来指明该项是包含在哪个域中。域存取控制寄存器中一个 2 位的字段控制访问每个域，每个字段可以迅速地标志访问一个域是允许还是禁止，以便整个存储区域可以非常有效地换入和换出虚拟存储区。可支持的域存取权限有两种：

- **客户**：域的用户（执行程序和存取数据），它必须获得构成该域每一节和每一页的存取允许。
- **管理员**：控制域（该域的当前节和当前页，以及该域的存取）的行为，页不会管理域中表项的存取允许权限。

一个程序可以是一些域的客户端，同时也是另外一些域的管理员，而对剩下的域没有访问权限，这样在一个程序访问不同的存储器资源时可以实现对存储器的灵活保护。

8.6 推荐的读物和 Web 站点

[STAL09] 包含了这章相关的详细内容。

STAL09 Stallings, W. *Operating System, Internal and Design Principles, Sixth Edition*. Upper Saddle River, NJ: Prentice Hall, 2009.



推荐的 Web 站点

- Operating System Resource Center：收集了许多有用的关于操作系统的文档资料和期刊。
- ACM Special Interest Group on Operating System：提供 SIGOPS 的出版资料和相关参考文献。
- IEEE Technical Committee on Operating System and Applications：包括在线时讯和到其他网站的链接。

8.7 关键词、思考题和习题

关键词

batch system：批处理系统
demand paging：请求分页
interactive operating system：交互式操作系统
interrupt：中断
job control language（JCL）：作业控制语言
kernel：内核
logical address：逻辑地址
long-term scheduling：长期调度
medium-term scheduling：中期调度
memory management：存储器管理
memory protection：存储器保护
multiprogramming：多道程序设计
multitasking：多任务化
nucleus：核，核心
operating system（OS）：操作系统
paging：分页
page table：页表
partitioning：分区
physical address：物理地址
privileged instruction：特权指令
process：进程
process control block：进程控制块
process state：进程状态
real memory：实存储器
resident monitor：驻留的监控程序
segmentation：分段
short-term scheduling：短期调度
swapping：交换
thrashing：抖动
time-sharing system：分时系统
translation lookaside buffer（TLB）：快表
utility：实用程序
virtual memory：虚拟存储器

思考题

8.1 什么是操作系统？
8.2 列出并简要定义操作系统提供的主要服务。
8.3 列出并简要定义操作系统的主要调度类型。
8.4 进程和程序有什么不同？
8.5 交换的目的是什么？
8.6 如果进程可以动态地分配到主存中的不同位置，这对寻址机制有何影响？
8.7 在进程执行期间，此进程的所有页都必须在主存中吗？
8.8 一进程的页在主存中必定是连续的吗？
8.9 一进程的页在主存中必须按顺序排列吗？
8.10 快表的作用是什么？

习题

8.1 假设我们有一台能进行多道程序设计的计算机，每个作业有一个标识符。在一个计算周期 T 内，一个

作业的一半时间花在 I/O 上，另一半时间花在处理器处理上。每个作业总共运行 N 个周期。假设使用简单的轮转优先权算法，I/O 操作能与处理器操作重叠。定义以下变量：

- 周转时间 = 完成一个作业的实际时间。
- 吞吐率 = 平均每个时间周期 T 内完成的作业数。
- 处理器利用率 = 处理器活动（不是等待）时间的百分数。

假设周期 T 以下列方式分配，同时有 1 个、2 个和 4 个作业时，计算周转时间，吞吐率和处理器利用率。

(a) I/O 占用第 1 个半周期，处理器占用第 2 个半周期。

(b) I/O 占用第 1、第 4 个 1/4 周期，处理器占用第 2、第 3 个 1/4 周期。

8.2 I/O 受限的程序定义为：如果单独运行，则等待 I/O 的时间比使用处理器的时间多；处理器受限的程序则相反。假设短期调度算法适合那些最近使用较少处理器时间的程序，解释为什么这个算法适合于 I/O 受限的程序，而并不始终拒绝被处理器受限的程序使用？

8.3 一个程序的功能是计算数组 A（大小为 100 × 100）中某一行的和：

$$C_i = \sum_{j=1}^{n} a_{ij}$$

假设计算机采用请求分页机制，每页大小为 1000 个字，主存分配给它们 5 个页帧。如果数组 A 按行存储或按列存储在虚拟存储器中，页失效率有区别吗？请解释原因。

8.4 一个容量为 2^n 字节的存储器采用等长度的分区方案，每个分区的大小是 2^m 字节，所维护的进程表包括一个指向各驻留进程分区的指针。此指针需要多少位？

8.5 考虑动态分区策略，请说明：平均而言，存储器拥有的空块数据是段数的一半。

8.6 假设处理器当前正在执行的进程的页表如下，所有数据是十进制，用数字表示每个事情均从 0 开始，所有地址都是存储器的字节地址。一个页的大小为 1024B。

虚拟页号	有效位	访问位	修改位	页帧号
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

(a) 准确描述 CPU 生成的虚拟地址如何转换成主存的物理地址。

(b) 虚拟地址：（Ⅰ）1052，（Ⅱ）2221，（Ⅲ）5499 对应的物理地址是什么（不考虑页故障）？

8.7 说出在虚拟存储器系统中，页大小既不应该很小也不应该很大的理由。

8.8 处理器以如下顺序访问 A、B、C、D、E 5 个页：

A，B，C，D，A，B，E，A，B，C，D，E

假定开始前主存有 3 个空页帧并采用先进先出替换算法，请指出此访问序列下主存传送（换入换出）页的页号序列。若有 4 个空页帧，重复此问题。

8.9 在带有虚拟存储器的计算机的执行过程中，遇到如下序列的虚拟页号：

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

假设采用最近最少使用的页替换策略，主存初始时为空，画出页命中率（访问的页已在主存中的百分比）与主存页容量 n（$1 \le n \le 8$）函数的图。假设开始时主存为空。

8.10 在 VAX 机中，用户的页表被放置在系统空间的虚拟地址处。问：用户的页表存储在虚拟空间而不存储在主存中有什么优点？又有什么缺点？

8.11 若程序语句

```
for (i=1; i<=n; i++)
    a[i]=a[i]+c[i];
```

在一个页大小为 1000 字的存储器上执行，请写一段机器指令程序来实现它，令 $n = 1000$，机器具有全范围的寄存器到寄存器的指令并可使用变址寄存器。然后写出执行期间页访问的序列。

8.12 IBM370体系结构使用了段和页的两级存储器结构，但它们的分段法缺少许多本章前面所描述的特征。对于基本的370体系结构，页大小是2KB或4KB，固定的段大小是64KB或1MB。对于370/XA和370/ESA体系结构，页大小是4KB，段大小是1MB。这种简单缺乏分段法的什么优点？370的分段法又有什么优点？

8.13 考虑一个既有分段又有分页的计算机系统，当段在主存中时，其最后一页总有些字是浪费的。另外，当段大小为s，页大小为p时，应有s/p个页表项。页越小，段中最后一页的浪费就越少，但页表却增大了。那么，多大的页能使总开销最小？

8.14 计算机有一个cache、主存和用于虚拟存储器的磁盘。如果一个字在cache中，则需20ns的时间来存取它。如果字在主存而不在cache中，则首先需60ns的时间将它调入cache，然后再开始存取。如果字不在主存中，则需12ms的时间从磁盘中获取，再用60ns将它存入cache中。如果cache的命中率为0.9，主存的命中率为0.6，问：存取一个字的平均存取时间是多少？

8.15 假设把一个任务分成4个大小相等的段，系统为每个段建立一个8项的页描述符表。于是，系统是分段和分页的组合，同时假设页长度为2 KB，问：

(a) 每个段的最大长度是多少？

(b) 此任务的最大逻辑地址空间是多少？

(c) 假设物理单元00021ABC中的一个元素被此任务存取，则任务产生的逻辑地址格式是什么？系统的最大物理地址空间是多少？

8.16 假设某个微处理器能存取多达2^{32}B的物理主存，它采用分段逻辑地址空间，最大长度为2^{16}B。每条指令包含那个两部分地址，采用外部存储管理单元（MMU），它的管理方案是把固定长度2^{12}B的物理存储中的相邻块分配给段，一个段的起始物理地址总是1024的倍数。画出采用合适的MMU数值并将逻辑地址转换成物理地址的外部块射机制的详细连接图，以及MMU的内部结构图（假设每个MMU包含128项直接映射段描述符cache）。每个MMU怎样选择？

8.17 考虑一个分页逻辑地址空间（包含32个页，每页2KB）映射成1MB的物理存取空间，问：

(a) 处理器的逻辑地址格式是什么？

(b) 页表的长度和宽度是什么？（不考虑"存取权"位。）

(c) 如果物理存储空间减少一半，则对页表有什么样的影响？

8.18 在IBM大型操作系统OS/390中，内核的一个重要模块是系统资源管理器（SRM），这个模块负责在地址空间（进程）之间分配资源。在各种操作系统中，SRM给予OS/390独特的精致度。没有任何其他类型的大型操作系统，甚至可以说，没有任何其他类型的操作系统，能与SRM所提供的功能相匹配。资源包括处理器、实存储器和I/O通道。SRM统计处理器、通道和各种关键数据结构的利用率，其目的是在性能监督和分析的基础上优化性能。建立后面的各种性能目标并把这些作为服务的指导，SRM基于系统的利用率动态地修改安装和作业性能特征。SRM本身也提供报告以允许训练有素的操作员改进配置和参数设置来改善用户业务。

这个问题是关于SRM活动的一个例子。实存储器被分成等长的块，称为帧，这里可能有成千上万个块。每个帧能容纳虚拟存储器的一个块，称为页。SRM大约每秒读取20次控制信息，每次都会检查所有页帧。如果页没有被访问过或修改过，计数器加1。过一段时间，SRM取这些数的平均值来确定系统中一页帧未被触动过的平均秒数。SRM这样做的目的是什么？SRM会采取什么动作？

8.19 根据图8-24所示的每个ARM的虚拟地址格式，写出其物理地址格式。

8.20 当主存划分为多个节时，画出类似图8-23的ARM虚拟存储器转换。

第三部分　中央处理器

到目前为止，我们基本上把处理器（processor）看作是一个黑盒子，考察了它与输入/输出设备，以及存储器的相互关系和操作。本部分将考察处理器的内部结构和功能。处理器由寄存器组、算术逻辑单元、指令执行单元、控制单元，以及这些部件间的互连结构所组成。本部分介绍诸如指令集设计和数据类型这样的体系结构问题，还考察像流水线这样的组织结构问题。下面对本部分的各章予以简述。

第 9 章　计算机算术

第 9 章考察算术逻辑单元（Arithmetic and Logic Unit，ALU）的功能，并关注数的表示方法和实现算术运算的技术。一般来说，处理器支持定点数（或整数）和浮点数两类算术运算。对这两种数据类型，本章都是先考察数的表示，然后讨论算术运算。本章还详细讨论了重要的 IEEE 754 浮点标准。

第 10 章　指令集：特征和功能

从程序员的视角来看，理解处理器操作的最佳途径是学习它所执行的机器指令集。指令集设计这个复杂的主题占据了第 10、第 11 两章的内容。第 10 章重点讨论指令集设计的功能方面。该章首先考察计算机指令所指定的功能类型，然后详细地考察操作数类型（即被操作的数据）和操作类型。最后，该章简要说明了处理器指令与汇编语言的关系。

第 11 章　指令集：寻址方式和指令格式

第 10 章主讨论指令集的语义学问题；第 11 章则是讨论指令集的语法学问题。具体而言，本章考察了指定存储器地址的方式和计算机指令的完整格式。

第 12 章　CPU 结构和功能

第 12 章集中讨论处理器的内部结构和功能。本章首先介绍了用作中央处理单元（Central Processing Unit，CPU）内部存储结构的寄存器组，然后把前面介绍过的所有内容综合起来，提供一个对 CPU 结构和功能的概述。CPU 的整体组织（ALU，寄存器组，控制器）在此被重新审视。接下来，讨论寄存器组的组成。本章的其余部分描述了处理器如何完成机器指令的执行。通过考察指令周期，说明了取指令、译码、执行和中断周期的功能及其相互关系。本章最后深入探讨如何使用流水（pipelining）技术来提高性能。

第 13 章　精简指令集计算机

第三部分的最后两章详细地考察了 CPU 设计的重要趋势。第 13 章介绍与精简指令集计算机（Reduced Instruction Set Computers，RISC）概念相关的设计方法。RISC 是计算机组织与体系结构领域近年来最显著的革新之一。RISC 体系结构急剧地改变了处理器体系结构的发展趋势。对 RISC 设计方法的分析可以使我们明了计算机组织和体系结构的许多重要问题。本章首先考察采用 RISC 设计方法的动力，然后详细地考察 RISC 指令集设计和 RISC CPU 体系结构，并对 RISC 与复杂指令集计算机（Complex Instruction Set Computer，CISC）设计方法做比较。

第 14 章　指令级并行性和超标量处理器

本章考察一个更新近的并且是同等重要的设计革新：超标量（superscalar）处理器。虽然超标量技术能用于任何类型的处理器，但它特别适合 RISC 体系结构的处理器使用。本章首先考察指令级并行性（instruction level parallelism）的有关性问题。

第9章

Computer Organization and Architecture: Designing for Performance, 8E

计算机算术

本章要点

- 计算机算术涉及的两个基本方面是数值表示方式（二进制格式）和基本算术运算（加、减、乘、除）的算法。这两个方面既适用于整数算术，也适用于浮点算术。
- 浮点数表示成一个数（有效值，significant）乘以一个定值（基值，base）的某个幂数（指数，exponent）。浮点数能够表示很大的数和很小的数。
- 大多数处理器都实现了 IEEE 754 标准，用于浮点表示和浮点运算。IEEE 754 定义了 32 位和 64 位两种浮点数格式。

下面以算术逻辑单元（ALU）的概述开始讨论处理器。简要介绍 ALU 之后，本章重点放在 ALU 的最复杂方面计算机算术。作为 ALU 一部分的逻辑功能将在第 10 章介绍，简单逻辑和运算功能的数字逻辑实现则在第 20 章介绍。

计算机算术一般要对两种很不相同的数值类型（整数和浮点数），完成算术运算。无论何种类型，表示法的选择都是关键的设计出发点。我们首先讨论数的表示法，然后再讨论其算术运算。

本章给出了一些实例，每个实例利用灰色方框突出显示。

9.1 算术逻辑单元

算术逻辑单元（Arithmetic and Logic Unit，ALU）是计算机实际完成数据算术逻辑运算的部件。计算机系统的其他部件（控制器、寄存器、存储器、输入/输出），主要是为 ALU 传入数据，待 ALU 处理后取回运算结果。在某种意义上，考察 ALU 涉及的是计算机的核心或本质。

算术逻辑单元，以及计算机所有电子部件实际上都是基于简单数字逻辑器件的应用，这些器件可以保存二进制数字，并完成简单的布尔逻辑（Boolean logic）操作。对此有兴趣的读者，可以阅读第 20 章的内容，它提供了算术逻辑运算的数字逻辑实现方面的知识。

图 9-1 以不失一般性的方式指出了 ALU 如何与处理器的其余部分互连。数据由寄存器提交给 ALU，运算结果也存于寄存器。这些寄存器是处理器内的临时存储位置，它们通过信号路径连接到 ALU（参见图 2-3）。ALU 可能根据运算结果设置一些标志。例如，如果计算结果超出了要保存它的寄存器位宽，那么上溢（overflow）标志将被置为 1。标志值也保存在处理器内的寄存器中。控制器提供控制 ALU 操作和数据传入送出 ALU 的信号。



图 9-1 ALU 的输入和输出

9.2 整数表示

在二进制数制系统中[⊖]，仅用数字 0 和 1、负号和小数点（radix point）表示任意一个数。例如，

$$-1101.0101_2 = -13.3125_{10}$$

对于计算机存储和处理，负号和小数点是不方便的，因为只能用二进制数字（0 和 1）来表示数。如果只使用非负整数，那么其表示是直截了当的。

⊖ 参见第 19 章关于数制系统（二进制、十进制、十六进制）的基本介绍。

一个8位的字能表示从0～255的数。例如：

$$00000000 = 0$$
$$00000001 = 1$$
$$00101001 = 41$$
$$10000000 = 128$$
$$11111111 = 255$$

通常，如果以一个n位二进制数字序列 $a_{n-1}a_{n-2}\cdots a_1a_0$ 表示一个无符号整数A，那么A的值是：

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

9.2.1 符号-幅值表示法

有几种可选的方式来表示负数和正数。这些表示方式都涉及将字的最高位（最左位）作为符号位对待：若最左位是0，则为正数；若最左位是1，则为负数。

采用符号位表示正负数的最简单的表示法是符号-幅值表示法（sign-magnitude representation）。以一个n位字为例，最左位为符号位。其余n－1位为整数的幅值（绝对值）。

$$+18 = 00010010$$
$$-18 = 10010010 \text{（符号-幅值）}$$

一般情况下，符号-幅值可表示为：

$$\text{符号-幅值} \qquad A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{如果 } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{如果 } a_{n-1} = 1 \end{cases} \qquad (9.1)$$

符号-幅值表示法有几个缺点。一个缺点是加减运算时既要考虑数的符号，又要考虑幅值，才能进行所要求的运算。在9.3节的讨论中，这点将会变得更清楚。另一个缺点是，0有两种表示：

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \text{（符号-幅值）}$$

这样之所以不方便，因为相对于单一的零的表示，它会使判断是否为零的操作（计算机经常需要进行的一种操作）稍微困难一点。

因为这些缺点，符号-幅值表示法很少用于ALU中的整数表示，而最常用的方案是2的补码表示法。

9.2.2 2的补码表示法

与符号-幅值表示法类似，2的补码表示法（twos complement representation）也使用最高位作为符号位，从而很容易判断一个整数是正还是负。其不同点在于其他位的解释方式。表9-1说明了2的补码表示法和算术的关键特征，这是本节和下一节所要阐述的。

表9-1 2的补码表示法和算术的主要特征

范围	-2^{n-1} 到 $2^{n-1}-1$
表示零的数	1个
取负	将此数对应的二进制串中各位取反，将此结果作为一个无符号数对待，再加1
位长度扩展	在此数的左边添加附加位的位置，并以原符号位的值填充这些位置
加减规则	若两个同符号数相加（两个正数或两个负数），则当且仅当结果的符号位变反时才出现上溢
减法规则	由A减B，则先取B的2的补，然后与A相加

大多数关于2的补码表示法的介绍都把重点放在生成负数的规则上，但并没给出为什么这些规则能够成立的证明。而本节和9.3节对于2的补码的介绍是基于参考文献[DATT93]的。它建议，通过以位加权取和的方式来定义2的补码，正如上面对无符号和符号-幅值表示法所做的那样，这有助于更好地理解2的补码表示法。这种解释的优点在于，它可以消除2的补码算术规则可能不适于某些特殊情况的任何疑虑。

考虑以2的补码形式来表示一个n位整数A。若A是正的，则符号位"a_{n-1}"是0；其余位表示此数的幅值，如同符号-幅值法一样，因此有：

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{对于 } A \geqslant 0$$

数零被认为是正的，因此表示为符号位值为0和所有幅值位都为0。可见，正整数可表示的范围是由0（所有幅值位全为0）到$2^{n-1}-1$（所有幅值位全为1）。再大的数将需要更多的位。

现在，对于一个负数A（$A<0$），其符号位a^{n-1}是1。其余$n-1$位能取2^{n-1}个值中的某个值。于是，负整数可表示的范围是由-1到-2^{n-1}。对于$n-1$位值与负整数值的对应，我们希望以这样一种方式来指派负整数的位值，它能使算术运算能直截了当地处理，类似于无符号整数算术那样。在无符号整数表示中，要从n位值的表示计算得到整数的值，是由各位乘以位权值取和而得到的，最高有效位的权是$+2^{n-1}$。对于一个有符号位的表示法，我们将会在9.3节看到，如果最高位的权是-2^{n-1}，那么上文所要求的算术运算性质将得到满足。这就是2的补码表示法中的约定，这个约定会产生如下的负数计算表达式：

$$\text{2的补码} \qquad A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \tag{9.2}$$

在正整数情况下，$a_{n-1}=0$，故$-2^{n-1}a_{n-1}=0$，这样该表达式定义了一个非负整数。因此，式（9.2）定义了正数和负数的2的补码表示法。

表9-2比较了4位二进制整数的符号-幅值和2的补码两种表示法。虽然2的补码表示法看似与人们的习惯有些别扭，但我们将会看到对于大多数最重要的运算，加法和减法，它是极其方便的。正因为如此，几乎所有的处理器都是采用此法来表示整数。

表9-2 4位二进制整数的各种表示法

十进制表示	符号-幅值表示	2的补码表示	移码表示（Biased Representation）
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011

（续）

十进制表示	符号-幅值表示	2的补码表示	移码表示（Biased Representation）
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	—	1000	—

值盒子（value box）是说明2的补码表示法的一个很有用的方法。盒子的最右端是1（2^0），往左每一个邻接的位置其值加倍，直到最左端。但最左端的值是负的。正如图9-2a所示，它能以2的补码表示的最小负数是 -2^{n-1}；如果非符号位的其他位有的是1，那么就表示要把对应的某个正数加到最小负数 -2^{n-1} 上。还有，负数的最左位必定是1，正数的最左位必定是0，这一点是很清楚的。于是，最大的正数是一个0开头，后面跟着全是1的数，即等于 $2^{n-1}-1$。

图9-2的其余部分用于说明，可以使用值盒子将2的补码转换成十进制，以及由十进制转换成2的补码。



图9-2 用于2的补码与十进制相互转换的值盒子

9.2.3 不同位长间的转换

有时取来一个 n 位整数需要以 m 位来保存，这里 $m>n$，对于符号-幅值表示法，这是很容易完成的：简单地将符号位移到新的最左位置上，多余出的空位全填充为0。例如：

$+18 = 00010010$（符号-幅值，8位）

$+18 = 0000000000010010$（符号-幅值，16位）

$-18 = 10010010$（符号-幅值，8位）

$-18 = 1000000000010010$（符号-幅值，16位）

这种做法对于2的补码的负数则不行，使用同样的例子：

$+18 = 00010010$（2的补码，8位）

$+18 = 0000000000010010$（2的补码，16位）

$-18 = 11101110$（2的补码，8位）

$-32\,658 = 1000000001101110$（2的补码，16位）

倒数第2行使用图9-2所示的值盒子很容易判断，而最后一行则可用等式（9.2）或16位值盒子来检验。

因此，与符号-幅值的扩展规则不同，2的补码整数的扩展规则是，移符号位到新的最左位，其余空出位均以符号位的值填充。即对于正数填充0，对于负数填充1。这种方式被称为符号扩展（sign extension）。

$-18 = 11101110$（2的补码，8位）

$-18 = 1111111111101110$（2的补码，16位）

为了解释这个规则为什么能工作，我们再看看把一个 n 位二进制数字序列 $a_{n-1}a_{n-2}\cdots a_1a_0$ 当作一个2的补码表示的整数 A，那么 A 的值是：

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

如果 A 是一个正数，那么规则肯定是正确的。现在假定 A 是一个负数，并且想要构成一个 m 位表示，$m > n$，则有：

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

这两个值必须相等：

$$-2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i = -2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

$$-2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{m-1}$$

$$2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=n-1}^{m-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow \quad a_{m-2} = \cdots = a_{n-2} = a_{n-1} = 1$$

从上面的第一个等式走到第二个等式，要求最低的 $n-1$ 位在两种表示中保持不变。再看倒数第二个等式，只有从第 $n-1$ 位到第 $m-2$ 位上的位置全为 1，等式才成立。所以，规则是正确的。研究一下 9.3 节开头的 2 的补码取负的讨论，你可能更容易理解这个符号扩展的规则。

9.2.4 定点表示法

最后应指出，本节所讨论的表示法有时称为定点（fixed point）表示法。这是因为小数点（二进制小数点）是固定的，并且被假定为在最低位数字的右边。程序员可使用定点表示法来表示二进制小数，方法是适当地降低数量级，使得小数点隐含地设置在某个其他位置上。

9.3 整数算术

本节考察 2 的补码表示的数的常用算术功能。

9.3.1 取负

在符号-幅值表示法中，求一个整数的负数的规则是简单的：只需将符号位取反。在 2 的补码表示法中，求一个整数的负数可用如下规则：

（1）将整数的每一位（包括符号位）取反（布尔反），即把每个 1 变为 0，每个 0 变为 1。

（2）将此取反结果作为一个无符号二进制整数对待，加 1。

上述两步骤称作 2 的求补运算（twos complement operation），或求整数的 2 的补，例如：

$+18$ = 00010010（2 的补码）

按位取反 = 11101101

$$\begin{array}{r} 11101101 \\ +\quad\quad 1 \\ \hline 11101110 \end{array} = -18$$

若对此数再取负，则正如我们预料到的，取负再取负将是原来那个数：

-18 = 11101110（2 的补码）

按位取反 = 00010001

$$\begin{array}{r} 00010001 \\ +\quad\quad 1 \\ \hline 00010010 \end{array} = +18$$

可使用2的补码表示定义式（9.2）来说明刚才介绍的操作的有效性。用 n 位二进制数字序列 $a_{n-1}a_{n-2}\cdots a_1a_0$ 来表示一个2的补码整数 A。它的值是

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

现在构造按位取反的位串，$\overline{a_{n-1}}\ \overline{a_{n-2}}\cdots\overline{a_0}$。然后把这个取反后的位串当作一个非负整数，并加1。最后，把得到的 n 位二进制数字当作一个2的补码表示的整数 B。于是，B 的值是

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

这么做之后我们期望等式 $A = -B$ 成立，即 $A + B = 0$。这很容易验证是正确的。

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i(a_i + \overline{a_i})\right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

上述推导过程假设了我们可以把 A 取反后的数作为一个无符号整数，以便完成加1的操作，然后又把加的结果作为一个2的补码表示的整数。这里有两个特殊情况需要考虑。第一种情况，考虑 $A = 0$。此时，对于一个8位的表示：

$$\begin{array}{rcl} 0 & = & 00000000 \text{（2的补码）} \\ \text{按位取反} & = & 11111111 \\ & & +\quad\ 1 \\ & & 100000000 \quad = \quad 0 \end{array}$$

上面加1时，有一个从最高位发出的进位（carry），可以忽略该进位。结果是0求负还是0，正如期望的那样。

第二种情况就更不是问题了。如果我们对位串1后面跟 $n-1$ 个0取负，我们得到的是原来的数。例如，对于一个8位的字（word）：

$$\begin{array}{rcl} -128 & = & 10000000 \text{（2的补码）} \\ \text{按位取反} & = & 01111111 \\ & & +\quad\ 1 \\ & & 10000000 = -128 \end{array}$$

这样的一些意外情况是难免的。n 位字可表示的不同位串数目是 2^n，这是个偶数。我们希望用这些位串来表示正数、负数和零。如果能表示的正数和负数的数目一样（符号-幅值），那么0就会有两种表示。如果0只有一种表示（2的补码），那么能表示的正数和负数的数目就必然不一样。在2的补码表示法中，一个 n 位长度的位串可以表示 -2^{n-1}，却不能表示 $+2^{n-1}$。

9.3.2 加法和减法

图9-3表示了2的补码的加法，加法执行过程与无符号整数加法一样，好像2的补码表示的数与无符号整数是一样的。图中前面的4个例子展示了无异常的运算过程。如果操作的结果为正，那么得到2的补码表示会与无符号整数表示是一样的。如果结果为负，那么会得到负数的2的补码形式。注意，在某些情况下，最高位会有一个进位，它超出了字的长度（图中以阴影表示），将被丢弃，从而忽略不计。

对于任一加法操作，如果出现结果的长度大于正被使用的字的长度，那么这种状况被称为上溢或溢出（overflow）。当上溢出现时，ALU必须指出这个事实，以通知其他部件不要试图使用

此结果。判断上溢的规则如下所述：

上溢规则（overflow rule）：两个数相加，若它们同为正数或同为负数，则当且仅当结果的符号位变为相反时才出现上溢。

图 9-3e 和图 9-3f 显示的是上溢例子。注意，不论是否有进位，都可能出现上溢。

减法也是很容易处理的。减法可用如下的规则实现：

减法规则（subtraction rule）：若由一个数（被减数）减去另一个数（减数），则只需求由减数的 2 的补（取负），并把它加到被减数上。

于是，减法可以用加法实现，如图 9-4 所示。此图最后两个例子说明上溢规则仍是适用的。

1001 = −7 −0101 = −5 1110 = −2 a) 7+(+5)	1100 = −4 +0100 = 4 10000 = 0 b) (−4)+(+4)
0011 = 3 +0100 = 4 0111 = 7 c) (3)+(4)	1100 = −4 +1111 = −1 11011 = −5 d) (−4)+(−1)
0101 = 5 +0100 = 4 1001 = 溢出 e) (5)+(4)	1001 = −7 +1010 = −6 10011 = 溢出 f) (−7)+(−6)

图 9-3 2 的补码表示的数相加

0010 = 2 +1001 = −7 1011 = −5 a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 −1110 = −2 10011 = 3 b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 −0010 = −2 0011 = 7 d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = 溢出 e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = 溢出 f) M = −6 = 1010 S = 4 = 0100 −S = 1100

图 9-4 两个 2 的补码表示的数相减（M − S）

考察图 9-5 所示的几何描述法［BENH92］，就能够加深对 2 的补码加减法的理解。图中上部的两个圆是将相应的数轴段以端到端的方式连接而构成的。注意，当数在圆上时，任一数的求补数（即取负）是其水平方向上相对的那个数（用水平破折线指示）。从圆上的任一数开始，我们通过顺时针方向移动 k 个位置来表示加正数 k（或减负数 k），逆时针方向移动 k 个位置来表示减正数 k（或加负数 k）。如果算术运算导致经过端到端的连接点，则产生一个不正确的答案（溢出）。

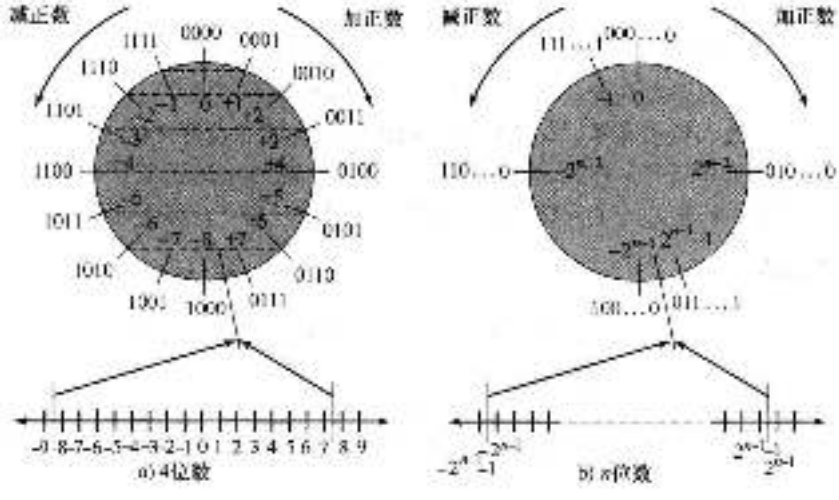


图 9-5 2 的补码整数的几何表示

图9-3和图9-4中的所有例子都能很容易地使用图9.5的图来描绘。

图9-6给出了实现加法和减法的数据通路和所需的硬件元件。中心元件是一个二进制加法器，它对输入的两个数进行相加，产生一个和以及一个上溢指示。二进制加法器将两个数看作是无符号数（二进制加法器的数字逻辑实现请见第20章）。对于加法，提交给加法器的两个数来自寄存器，在图9-6中是A和B寄存器。结果通常是存于这两个寄存器中的某一个或是另外的第三个寄存器。上溢指示保存在一个1位的上溢标志（overflow flag，0=无上溢，1=上溢）中。对于减法，减数（B寄存器）要通过一个2的补码求补器（complementer），产生减数的2的补，并提交给加法器。注意图9-6只是显示了数据通路。还需要一些控制信号，根据当前执行的操作是加法还是减法，来控制是否需要使用求补器。

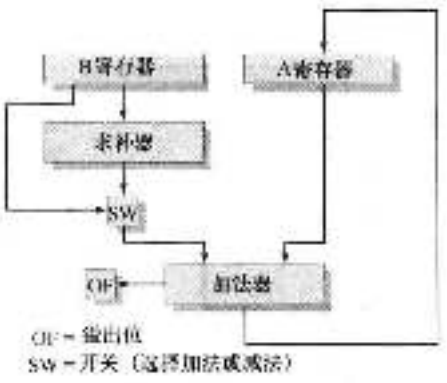


图9-6 加减法硬件框图

9.3.3 乘法

与加法和减法相比，无论是以硬件还是以软件来完成，乘法都是一个复杂的操作。各种各样的算法已用于各类计算机中。本小节的目的在于给读者某些关于常用乘法算法的感性认识。首先，我们介绍如何实现两个无符号（非负）整数相乘的简单方法，然后再关注实现两个2的补码表示数乘法的最通用技术。

1. 无符号整数乘法

图9-7说明了无符号二进制整数的乘法，就像我们用笔和纸手工演算的那样。由此可以得出几点重要发现：



图9-7 无符号二进制整数乘法

（1）乘法涉及部分积的生成，乘数的每一位对应一个部分积。然后，部分积相加得到最后的乘积。

（2）部分积是容易确定的。当乘数的位是0，其部分积也是0；当乘数的位是1，其部分积是被乘数。

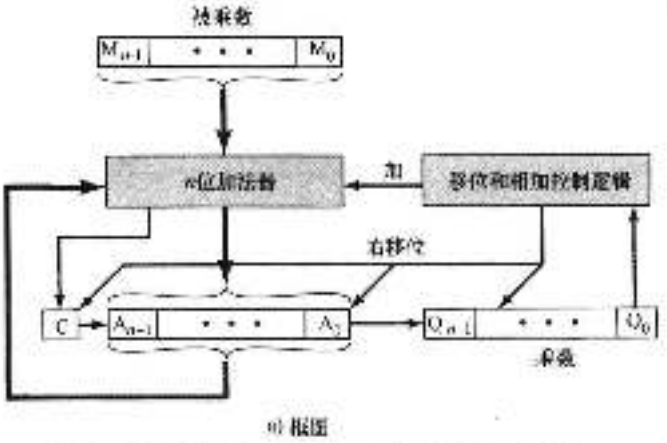
（3）部分积通过求和而得到最后乘积。因此，后面的部分积总要比它前面的部分积左移一个位置。

（4）两个n位二进制整数的乘法可产生最大长度为$2n$位的积（如$11 \times 11 = 1001$）。

与笔纸手工演算相比，计算机能做一些改进使乘法操作更有效。首先，可以边产生部分积边做加法，而不是等到最后再相加。这就消除了存储所有部分积的需求，从而减少了需要的寄存器数目。其次，能节省某些部分积的生成时间，对于乘数的每个1，需要执行加和移位两个操作；但对于每个0，则只执行移位操作就够了。

图9-8a表示了一种采用上述改进的实现方案。乘数和被乘数分别装入两个寄存器（Q和M）。保存部分积需要第三个寄存器，寄存器A，初始设置为0。还需要一个1位寄存器C，初始为0，用于保存加法可能产生的进位。

乘法器的操作如下。控制逻辑每次读乘数的一位。若Q_0是1，则被乘数与A寄存器相加，并将结果存于A寄存器。然后，C、A和Q各寄存器的所有位向右移一位，于是C位进入A_{n-1}，A_0进入Q_{n-1}而Q_0丢失。若Q_0是0，则只需要移位，不需要进行加法运算。对原始的乘数每一位重复上述过程。产生的$2n$位积存于A和Q寄存器。这种操作的流程图显示于图9-9中，图9-8b给出了一个例子。注意图9-8b中例子的第2周期，因为乘数当前位是0，所以该周期没有加法运算。



a) 框图

C	A	Q	M	
0	0000	1101	1011	初始值
0	1011	1101	1011	加
0	0101	1110	1011	移位
0	0010	1111	1011	移位
0	1101	1111	1011	加
0	0110	1111	1011	移位
1	0001	1111	1011	加
0	1000	1111	1011	移位

b) 图9-7中的例子（乘法A、Q）

图 9-8 无符号二进制乘法的硬件实现

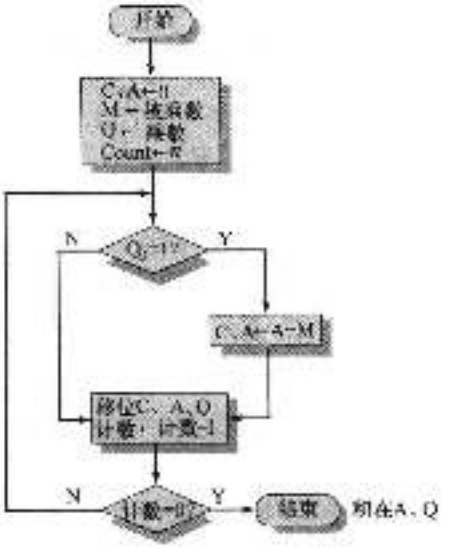


图 9-9 无符号二进制乘法流程图

2. 2 的补码乘法

我们已经看到，对 2 的补码表示的数能将它们看作是无符号数来完成加减法运算。现在考虑：

$$\begin{array}{r} 1001 \\ +0011 \\ \hline 1100 \end{array}$$

若将这些数看成是无符号整数，则是 9（1001）加 3（0011）得到 12（1100）。若看成是 2 的补码整数，则是 −7（1001）加 3（0011）得到 −4（1100）。

遗憾的是，这种简单做法不能用于乘法。为说明这点，再看图 9-7。我们是将 11（1011）乘以 13(1101)得到 143(10001111)。若将其解释为补码数，则是 −5(1011)乘以 −3(1101)得到的却是 −113(10001111)。这个例子说明，如果被乘数和乘数都是负数，简单直接的乘法将不能使用。实际上，被乘数和乘数只要有一个是负数就不行。为说明这种状况，需要返回到图 9-7 的例子，借助 2 的幂操作看看实际在做什么。回想，任何一个无符号二进制数都可表示成 2 的幂之和。于是：

$$\begin{aligned} 1101 &= 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 \\ &= 2^3 + 2^2 + 2^0 \end{aligned}$$

而且，一个数乘以 2^k 可通过左移此数 k 位来完成。理解了这一点，可以看出图 9-10 是对图 9-7 的改造，以使部分积变得更显和完整。唯一的不同在于，图9-10 把由 n 位被乘数产生的部分积当作是一个 $2n$ 位的数。

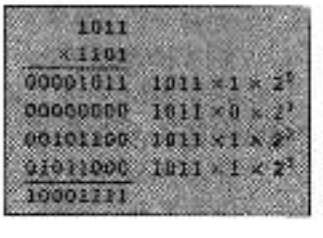
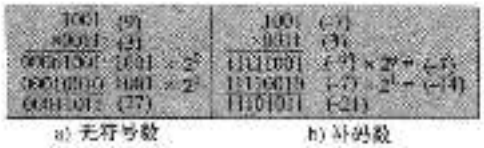


图 9-10 两个无符号 4 位整数相乘产生 8 位结果

于是，作为一个无符号数，4 位被乘数 1011 是以一个 00001011 的 8 位字来保存的。每个部分积（对应除了 2^0 这一位之外的其他

非0乘数位）由这个数左移，并且空出位以0填充而组成（例如，此数左移两次产生00101100）。

现在我们来说明，若被乘数是负数，为什么简单直接的乘法是不能工作的。问题在于，作为负的被乘数，其每次得出的部分积必须是2n位字长的负数；部分积的符号位必须一同设置。这可由图9-11说明，它表示的是1001乘以0011。若这些数被看作是无符号数，则是9×3=27，处理很简单。然而，若把1001看作是补码数-7，则每个部分积必须是2n位（8位）的负的补码数，如图9-11b所示。注意，这要求用部分积左边填充1来完成。



a) 无符号数　　b) 补码数

图9-11　无符号数和补码数的整数乘法比较

应该清楚，若乘数是负数，那种简单直接的乘法也是不能工作的。理由是乘数的各位不再对应于必须发生的移位或乘法操作。例如，十进制数-3的四位补码表示为1011。如果采用简单的按位操作来取部分积，则会有如下的对应结果：

$$1101 \longleftrightarrow -(1\times2^3+1\times2^2+0\times2^1+1\times2^0)=-(2^3+2^2+2^0)$$

实际上，1101对应的是$-(2^3+2^0)$。因此负的乘数不能直接用于上面所描述的方式。

有几种摆脱这种困境的方法。一是把被乘数和乘数都转变成正数再相乘，然后当且仅当两个原始数的符号不同时，其结果取其2的补（即取负）。乘法电路实现者更喜欢采用上述最后转换步骤的方法。其中一种广为使用的方法是布斯（Booth）算法。这种方法还有一个好处，与前面介绍的无符号数直接乘法相比它能加速乘法过程。

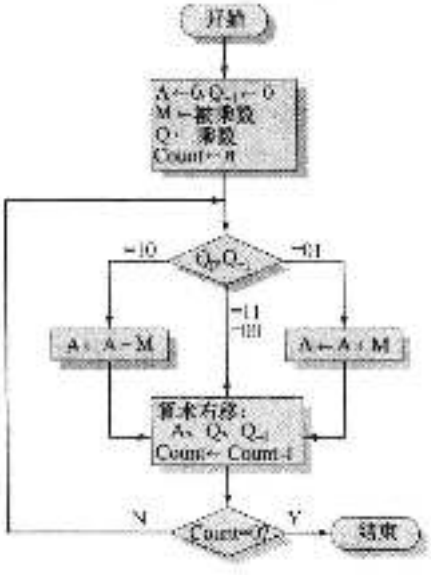


图9-12　2的补码乘法的布斯算法

图9-12给出了布斯算法框图，可做如下描述。与前文相同，乘数和被乘数分别放入Q和M寄存器内。这里也有一个1位寄存器，逻辑上位于Q寄存器最低位（Q_0）的右边，并命名为Q_{-1}；它的用途下面即将说明。乘法的结果将保存在A和Q寄存器中。A和Q_{-1}初始化为0。与前文相同，控制逻辑也是每次扫描乘数的一位。只不过现在是检查某一位时，它右边的一位也同时被检查。若两位相同（1—1或0—0），则A、Q和Q_{-1}寄存器的所有位向右移一位。若两位不同，根据两位是0—1或1—0，则被乘数被加到A寄存器或由A寄存器减去，加减之后再右移。无论哪种情况，右移是这样进行的：A的最左位，即A_{n-1}位，不仅移入A_{n-2}，而且仍保留在A_{n-1}中。这要求保留A和Q中数的符号。这种移位称为**算术移位**（arithmetic shift），因为它保留了符号位。

图9-13为7乘以3时布斯算法的操作顺序。图9-14a给出同样操作的更紧凑表示。图9-14其余部分给出其他例子。正如所示，对于任何正负数的组合，布斯算法都能工作得很好。同时注意此算法的效率，连续的1串或0串都可以跳过，只在每串开头和结尾有加法或减法，平均而言每串一次。

A	Q	Q_{-1}	M		
0000	0011	0	0111	初始化	
1001	0011	0	0111	$A \leftarrow A-M$	第1周期
1100	1001	1	0111	移位	
1110	0100	1	0111	移位	第2周期
0101	0100	1	0111	$A \leftarrow A+M$	第3周期
0010	1010	0	0111	移位	
0001	0101	0	0111	移位	第4周期

图9-13　布斯算法举例（7×3）

布斯算法为什么能得到正确结果？首先可以考虑正乘数的情况。具体来说，考虑一个由全为1的块两边是0的乘数，例如00011110。如前文所述，乘法能通过将被乘数

适当左移的副本相加来实现：

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

这种操作数目能减少到两个，如果我们能发现：

$$2^4 + 2^{3-1} + \cdots + 2^{2-5} = 2^{4+1} - 2^{1-5} \quad (9.3)$$

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

因此可通过被乘数的一次加法和一次减法来得到积。这种策略可以扩展到乘数中右任何数目（连续）的 1 的块，包括把单个 1 也看作一个块的情况。于是有：

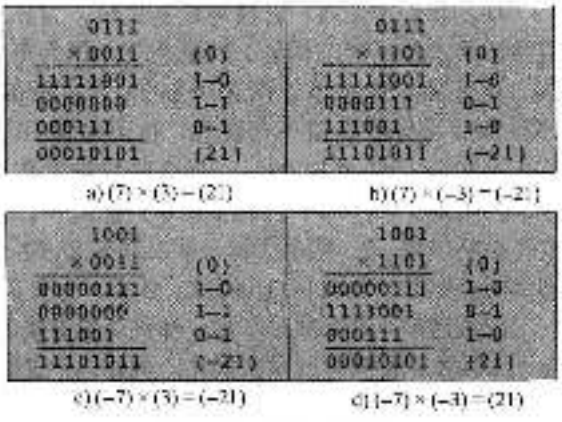


图 9-14 使用布斯算法举例

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

布斯算法以下述方式执行这种策略，当遇到一串 1 的第一个 1 时（1 – 0），执行一次减法，当遇到一串 1 的最后一个 1 时（0 – 1），执行一次加法。

为说明对于一个负的乘数这样的策略照样能工作，我们需要做如下的推导。假设 X 是一个 2 的补码表示的负数，其表示为：

$$X \text{ 的表示} = |1x_{n-2}x_{n-3}\cdots x_1x_0|$$

X 的值则可表示成：

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0) \quad (9.4)$$

读者可将式（9.4）应用到表 9-2 中的数来验证。

现在我们知道 X 的最左位是 1，因为 X 是一个负数。假定最右的 0 在第 k 位上，于是 X 的表示为：

$$X \text{ 的表示} = |111\cdots10x_{k-1}x_{k-2}\cdots x_1x_0| \quad (9.5)$$

其值为：

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k-1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \quad (9.6)$$

现在，由式（9.3）可知：

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

重新排列：

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \quad (9.7)$$

将式（9.7）代入式（9.6），得到：

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \quad (9.8)$$

终于返回到布斯算法了。记住 X 的表示（式（9.5）），这一点是清楚的：从 x_0 起到最左的 0 的所有位都能被这样处理，因为它们产生式（9.8）中除去（-2^{k+1}）项之外的所有项。当算法扫描经过最左的 0 而遇到 1（2^{k+1}）时，出现一个 1 – 0 变化，于是一个减法发生（-2^{k+1}）。这是式（9.8）的最后一个剩余项。

作为一个例子，让我们考察某个数 M 乘以（–6）。若字长为 8 位，则（–6）的补码表示式为 11111010。由式（9.4）可知：

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

这个读者可以很容易验证。于是有：

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2)$$

使用式（9.7），

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

读者不难验证这还是 $M \times (-6)$。最后，依据前面阐述的理由，最终得出：

$$M \times (11111010) = M \times (-2^7 + 2^2 - 2^1)$$

现在，我们清楚地看出布斯算法是遵守上述操作方式的。它从最低位开始，当遇到第一个 1 时（1−0），它执行一次减法；当遇到（0−1）时，它执行一次加法；最后遇到下一个全是 1 串的第一个 1 时，又完成另一次减法。于是，布斯算法与直接移位相加的算法相比只需完成较少的加法和减法。

9.3.4 除法

除法要比乘法更复杂，但也是基于同样的通用原则。同前述一样，算法的基础是纸和笔的演算方法，并且操作涉及重复的移位和加或减。

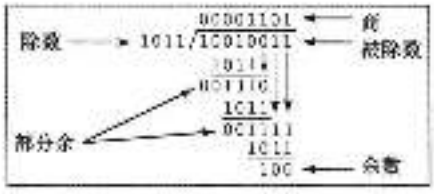


图 9-15 无符号二进制整数的除法举例

图 9-15 表示的是一个无符号二进制整数长除（long division）的例子。详细描述这个过程是有指导意义的。首先，从左到右检查被除数的位，直到被检查的位所表示的数大于或等于除数；这被称为除数能去“除”此数。直到这个事件发生之前，一串 0 从左到右被放入到商中。当上述事件发生时，一个 1 被放入商，并且从这个部分被除数中减去除数。结果称为部分余（partial remainder）。由此开始除法呈现一种循环样式。在每一次循环中，被除数的其他位被加到部分余上，直到所构成的数大于或等于除数。同前面一样，除数由这个数中减去并产生新的部分余。此过程继续下去，直到被除数的所有位都被用完。

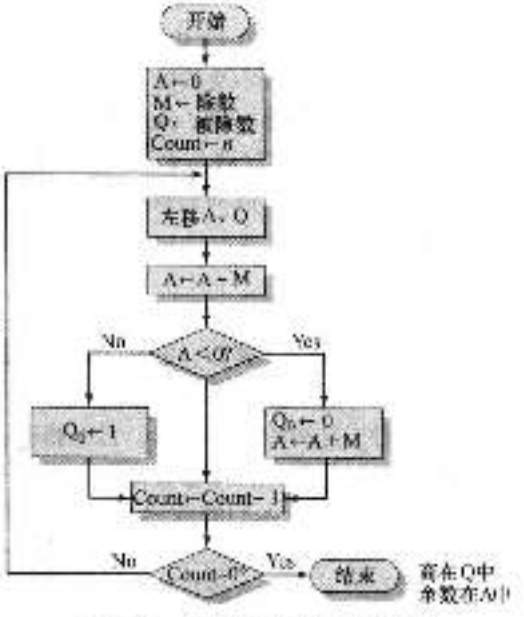


图 9-16 无符号二进制除法流程

图 9-16 表示了对应此长除过程的机器算法。除数放入 M 寄存器，被除数放在 Q 寄存器中。每一步 A 和 Q 寄存器一起左移 1 位。然后 A 减 M 以确定 A 是否能分出部分余[⊖]来。若够减，则 Q_0 位变为 1。否则，Q_0 位为 0，并且 M 必须被返加到 A 以恢复原先的值。计数值然后减 1，此过程持续进行 n 步。结束时，商保存在 Q 寄存器中，余数保存在 A 寄存器中。

这个过程能扩展到用于负数，但有一些难度。我们给出一个用于 2 的补码数的方法，这种方法的几个例子显示于图 9-17 中。

该算法假设除数 V 和被除数 D 都是正数，且 $|V| < |D|$。如果 $|V| = |D|$，那么商 $Q = 1$，且余数 $R = 0$。如果 $|V| > |D|$，那么商 $Q = 0$，且余数 $R = D$。算法可概括如下：

（1）把除数的 2 的补装入 M 寄存器，实际上是把除数的相反数装入 M 寄存器。被除数装入 A，Q 寄存器。被除数必须以 $2n$ 位的正数来表示。例如，4 位 0111 变成 00000111。

⊖ 这是一个无符号减法，若结果产生了最高位向上的借位，则是不够减。

(2) A，Q 左移 1 位。

(3) 执行 A←A − M。这个操作从寄存器 A 的值中减去除数。

(4) a) 若上一步的减法结果为非负（寄存器 A 的最高位 = 0），则置 Q_0←1。

b) 若上一步减法结果为负（寄存器 A 的最高位 = 1），则置 Q_0←0，并恢复寄存器 A 的原值。

(5) 重复 (2) 到 (4) 步，Q 有多少位就重复多少次。

(6) 余数在 A 中，商在 Q 中。

A	Q	
0000	0111	初始值
0000 1101 1101 0000	1110	移位 使用 0011 的补码做减法 恢复余数，置 Q_0=0
0001 1101 1110 0001	1100	移位 相减 恢复余数，置 Q_0=0
0011 1101 0000	1000	移位 相减，置 Q_0=1
0001 1101 1110 0001	0010 0010	移位 相减 恢复余数，置 Q_0=0

图 9-17 恢复余数（Restoring）的 2 的补码除法举例（7/3）

为了处理负数，我们考虑余数定义为：

$$D = Q \times V + R$$

（这里 D = 被除数，Q = 商，V = 除数，R = 余数）

考虑下面的整数除法例子，其中包括 D 和 V 的所有可能的符号组合。

$$\begin{aligned}
&D=7 \quad V=3 \quad \Rightarrow \quad Q=2 \quad R=1\\
&D=7 \quad V=-3 \quad \Rightarrow \quad Q=-2 \quad R=1\\
&D=-7 \quad V=3 \quad \Rightarrow \quad Q=-2 \quad R=-1\\
&D=-7 \quad V=-3 \quad \Rightarrow \quad Q=2 \quad R=-1
\end{aligned}$$

读者可能注意到图 9-17 中 (−7) / (3) 和 (7) / (−3) 产生不同的余数。不过可以发现，Q 和 R 的绝对值并不受被除数和除数符号的影响。而 Q 和 R 的符号不难从被除数 D 和除数 V 的符号推导出来。具体而言，$\text{sign}(R) = \text{sign}(D)$，而 $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$。因此，实现 2 的补码除法的一种方法就是把操作数都转换为无符号绝对值，进行除法运算，最后，根据被除数和除数的符号来设置商和余数的符号。这正是恢复余数除法算法所选取的方式 [PARH00]。

9.4 浮点表示

9.4.1 原理

使用定点表示法（例如，2 的补码）能表示以 0 为中心一定范围内的正和负的整数。通过重新设定小数点的位置，这种格式也能用来表示带有小数部分的数。

不过这种方法有明显的限制，它不能表示很大的数，也不能表示很小的分数。而且当两个大数相除时，商的小数部分可能会丢失。

对于十进制数，人们解除这种限制的方法是使用科学计数法（ecientific notation）。于是，$976\,000\,000\,000\,000$ 可表示成 9.76×10^{14}，而 $0.000\,000\,000\,000\,097\,6$ 可表示成 9.76×10^{-14}。实际上我们所做的只是动态地移动十进制小数点到一个合适的位置，并使用 10 的指数来保持对此小数点位置的跟踪。这就允许只使用少数几个数字来表示很大范围的数和很小的数。

这样的方法也可用于二进制数。可以使用如下形式表示一个数：

$$+S \times B^{-E}$$

这样的数保存在一个二进制字的三个字段中。

- 符号：正或负。
- 有效值 S（significand）。
- 指数或者称为阶 E（exponent）。

指数的底或基 B（base）是隐含的，因此是不需要存储的，因为对所有的数它都是相同的。

通常，小数点位置被约定在最左（最高）有效位的右边，即小数点左边有1位。

最好以例子来说明用于表示二进制浮点数的原则。图9-18a表示了一个典型的32位浮点格式。最左位保存数的符号（0 = 正，1 = 负）。阶值存于位1到位8，所用的表示法是称为移码（biased）的表示法。从字段中减去一个称为偏移量（bias）的固定值，才得到真正的指数。通常，偏移量等于$2^{k-1}-1$，k是二进制指数的位数。在此例子中，一个8位字段能表示的数是0～255。取偏移量为127（即2^7-1），则真实阶值的范围是－127～＋128。此例中阶值的底被认为是2。

表9-2给出了4位指数的移码表示。注意，当移码表示法的各位被作为一个无符号整数对待时，其数的大小相对关系并不改变。例如，在无符号数和移码两种表示法中，都是1111最大，0000最小；然而，这在符号-幅值或2的补码表示法中却不是真的了。使用移码表示法的好处在于，非负的浮点数能作为整数对待，便于进行比较。



a) 格式

$1.1010001 \times 2^{(127)} = 0.10011011\ 10100010000000000000000 = 1.6328125 \times 2^{27}$
$-1.1010001 \times 2^{(102)} = 1.00110011\ 10100010000000000000000 = -1.6328125 \times 2^{-25}$
$1.1010001 \times 2^{(10100)} = 0.01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-107}$
$-1.1010001 \times 2^{(-10100)} = 1.01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

b) 举例

图9-18 典型的32位浮点格式

字的最后一部分是有效值[⊖]（上例中是23位）。

> 任一浮点数都能以多种样式来表示。以下各式是等价的，这里的有效数以二进制格式表示：
>
> 0.110×2^5
>
> 110×2^2
>
> 0.0110×2^6

为了简化浮点数的操作，一般需要对它们进行规格化（normalize）。一个规格化的数是一个有效值的最高有效位为非零的数。对于二进制表示法，一个规格化数是它的有效值的最高有效位是1。正如前面所述，通常约定小数点左边有1位。于是，一个规格化的非零数具有如下格式：

$$+1.\text{bbb}\cdots\text{b} \times 2^{\pm E}$$

这里的b是二进制数字（0或1）。这意味着有效值的最左位必须总是1。因此也没必要总存储这个1，所以它成为隐含的。于是23位有效值字段能用于存储24位有效数字，其值范围在半开区间[1, 2)。对于一个非规格化的数，通过移动小数点直到最左一个1的右边并相应调整阶值，就可以将此数规格化。

图9-18b给出几个以这种规格化形式存储的数的例子。其中每个例子左边是二进制数值，中间是对应的二进制位串表示，右边是十进制数值。注意如下特征：

- 符号总是位于字的第1位。
- 真实有效值的第1位总是1，并且不需要存于有效值字段中。
- 值127加到真实阶值后再存入阶值字段中。
- 阶的底是2。

为便比较，图9-19显示了这种表示法的32位字能表示的数的范围。使用2的补码整数表示

⊖ 用于替代有效值的术语是尾数（mantissa），但尾数有些过时了，尾数实用于表示对数的小数部分，因此在本书中尽量避免使用。

法，由 -2^{31} 到 $2^{31}-1$ 的所有整数都能被表示，总计 2^{32} 个不同的数。以图 9-18 的浮点格式为例，可以表示如下范围的数：

- 介于 $-(2-2^{-23})\times 2^{128}$ 和 -2^{-127} 之间的负数。
- 介于 2^{-127} 和 $(2-2^{-23})\times 2^{128}$ 之间的正数。

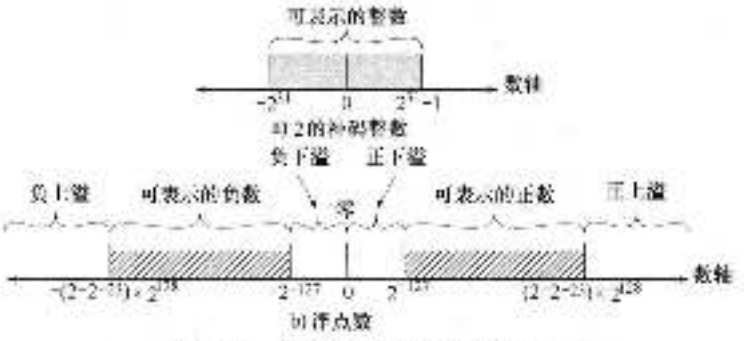


图 9-19 共显 32 位格式可表示的数

数轴上有 5 个区间不包括在这些范围内：

- 比 $-(2-2^{-23})\times 2^{128}$ 还小的负数，叫做负上溢（negative overflow）。
- 比 -2^{-127} 还大的负数，叫做负下溢（negative underflow）。
- 零。
- 比 2^{-127} 还小的正数，叫做正下溢（positive underflow）。
- 比 $(2-2^{-23})\times 2^{128}$ 还大的正数，叫做正上溢（positive overflow）。

正如上面提到的，这种表示法不适合值 0 的表示。然而，下面将会看到，可以把一个专门的位串定义成零。当算术运算的结果其幅值比指数 128 能表示的幅值还大时，则出现上溢（例如，$2^{100}\times 2^{100}=2^{200}$）。当小数的幅值太小时（例如，$2^{-100}\times 2^{-100}=2^{-200}$），则出现下溢。下溢不是一个严重问题，因为其结果通常足够小而可以近似成 0。

注意，使用浮点表示法并不能使我们表示出更多的值。以 32 位二进制位串能表示不同值的最大数目仍是 2^{32}。我们所做的，实际上只是把这些数沿数轴正负两个方向在更大范围内分布。在实际应用中，大多数浮点数只是用户真正想表示数值的一个近似。不过，对于不是很大的指数而言，浮点表示还是精确的。

还应注意，浮点表示的数不再像定点数那样沿数轴等距分布，而是越靠近原点，数越密集，越远离原点，数越稀疏，如图 9-20 所示。这是浮点算术的重要特点之一：多数计算的结果并不是严格精确的，必须进行某种舍入，以使结果达到所能表示的最近似值。



图 9-20 浮点数的密度

以图 9-18 所示的格式类型为例，范围（range）和精度（precision）是要权衡考虑的问题。例子是 8 位用于阶，23 位用于有效数。若增加阶的位数，就扩充了可表示数的范围。但是，总的位数是不变的，因此能表示的数的总数也是固定不变的，于是实际效果是减少了这些数的密度，因而也就降低了精度。既能增加范围，又能增加精度的唯一途径是使用更多的位。于是，大多数计算机都至少提供单精度和双精度两种浮点数。例如，单精度格式是 32 位，双精度格式是 64 位。

这还只是一个阶值位数和有效值位数之间的折中问题。还有一个问题比它要复杂，隐含的阶值的底并不需要总是 2。IBM S/390 结构就是一个使用底为 16 的例子［ANDE67b］，其浮点数格式由 7 位阶值和 24 位有效值组成。

按照 IBM 底为 16 的格式：

$$0.11010001 \times 2^{1000} = 0.11010001 \times 16^{100}$$

于是所存的阶值是 5 而不再是 20。

使用较大底的优点在于同样数目的阶值位能表示的数的范围更大。但是，要记住，我们并没有增加所能表示的数的总数。因此，对一种固定格式而言，更大的底能给出更大的表示范围，但以精性精度为代价。

9.4.2 二进制浮点表示的 IEEE 标准

最重要的浮点表示法是在 1985 年通过的 IEEE 754 标准中所定义的浮点表示法［IEEE85］。开发这个标准是为了提高程序从一种处理器移植到另一种处理器上的可移植性，也为了促进研制更为复杂的数值运算程序。这个标准获得了广泛的认可，并已经用于当代各种处理器和算术协处理器中。

IEEE 754 标准定义了 32 位的单精度和 64 位的双精度两种格式（见图 9-21），它们的阶值字段分别是 8 位和 11 位，隐含的底是 2。另外，标准还定义了单精度、双精度的扩展格式，但它们的具体格式是与实现相关的。扩展格式包括在阶值字段提供更多的位（扩展范围）和在有效值字段提供更多的位（扩展精度）。扩展格式将被用于中间计算过程。由于它们有更高精度，扩展格式使得最终结果避免被过量舍入，从而减少加大误差的机会；由于它们有更大表示范围，扩展格式也使计算过程中出现上溢的机会减少，这样如果计算的最终结果能以基本格式表示，那么就不会因为中间计算过程中的上溢而提前终止了。对于单精度扩展格式而言，它的一个推动力是它能呈现双精度格式的某些优点又不导致计算耗时过长。因为通常是精度越高，计算耗时越长。表 9-3 总结了 4 种格式的特点。

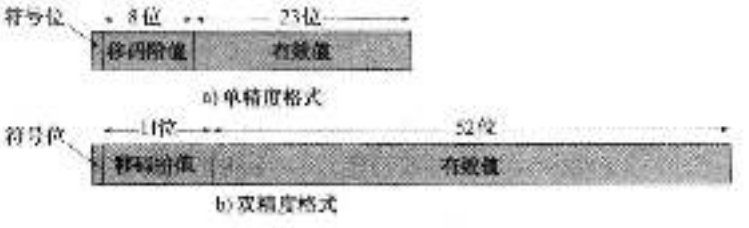


图 9-21 IEEE 754 格式

表 9-3 IEEE 754 格式参数

参数	单精度	单精度扩展	双精度	双精度扩展
字宽（位数）	32	≥43	64	≥79
阶值位宽（位数）	8	≥11	11	≥15
阶值偏移量	127	未指定	1023	未指定
最大阶值	127	≥1023	1023	≥16383
最小阶值	−126	≤−1022	−1022	≤−16382
数的范围（底为 10）	10^{-38}，10^{+38}	未指定	10^{-308}，10^{+308}	未指定
有效值位宽（位数）	23	≥31	52	≥63
阶值的数目	254	未指定	2046	未指定
小数的数目	2^{23}	未指定	2^{52}	未指定
值的数目	1.98×2^{30}	未指定	1.99×2^{62}	未指定

注：不包括隐含位。

注意，IEEE 754 格式的所有位模式（bit pattern）并不是都以通常方式解释，某些位模式用来表示特殊值。表 9-4 指出各种位模式对应的值。全 0（0）和全 1（单精度是 255，双精度是 2047）这种极端阶值用来定义特殊值。数的分类如下所示。

表 9-4 IEEE 754 浮点数的说明

	单精度（32 位）				双精度（64 位）			
	符号	移码阶值	小数	值	符号	移码阶值	小数	值
正零	0	0	0	0	0	0	0	0
负零	1	0	0	−0	1	0	0	−0
正无穷大	0	255（全 1）	0	∞	0	2047（全 1）	0	∞
负无穷大	1	255（全 1）	0	−∞	1	2047（全 1）	0	−∞
静默式非数（quiet NaN）	0 或 1	255（全 1）	≠0	NaN	0 或 1	2047（全 1）	≠0	NaN
通知式非数（signaling NaN）	0 或 1	255（全 1）	≠0	NaN	0 或 1	2047（全 1）	≠0	NaN
正的规格化非零数	0	0 < e < 255	f	$2^{e-127}(1.f)$	0	0 < e < 2047	f	$2^{e-1023}(1.f)$
负的规格化非零数	1	0 < e < 255	f	$-2^{e-127}(1.f)$	1	0 < e < 2047	f	$-2^{e-1023}(1.f)$
正的非规格化数	0	0	f≠0	$2^{e-126}(0.f)$	0	0	f≠0	$2^{e-1022}(0.f)$
负的非规格化数	1	0	f≠0	$-2^{e-126}(0.f)$	1	0	f≠0	$-2^{e-1022}(0.f)$

- 如果阶值范围在 1～254（单精度）和 1～2046（双精度），那么位模式表示了一个规格化的非零浮点数。阶值是移码表示的，故真正阶值范围是 −126～+127（单精度）和 −1022～+1023（双精度）。一个规格化的数要求二进制小数点左边有一个 1；这位是隐藏的，使得有效值的总位数实际为 24 位或 53 位（标准中称有效值为小数，fraction）。
- 0 阶值与 0 有效值一起表示正零或负零，取决于它的符号位。正如我们曾提到的，有精确的 0 值表示是有益的。
- 全 1 阶值与 0 有效值一起表示正无穷大或负无穷大，取决于它的符号位。能表示无穷大也是有用的。把上溢看成一个错误条件而停止程序执行，还是把它看成是一个 ∞ 值带入程序并继续处理，这样的决定权留给用户。
- 0 阶值与非 0 有效值一起表示一个非规格化（denormalized）数。这种情况下，二进制小数点左边的隐藏位是 0，并且真实阶值是 −126 或 −1022。数的正负取决于它的符号位。
- 全 1 阶值与非 0 有效值一起给出非数（NaN）值，它意味着不是一个数（Not a Number）。非数（NaN）用来表示出现了各种异常情况。

9.5 节将讨论非规格化数和 NaN 的意义。

9.5 浮点算术

表 9-5 总结了浮点算术的基本操作。对于加、减法，必须保证两个操作数具有相同的阶。这可能要求移动一个操作数的小数点以达到对齐。乘、除法反而更简单些。

浮点运算可能会产生如下几种特殊情况：

- **阶值上溢**（exponent overflow）：一个正阶值超出了最大允许阶值。某些系统将其设计成 +∞ 或 −∞。
- **阶值下溢**（exponent underflow）：一个负阶值小于最小允许阶值（如 −200 小于 −127）。这意味着那个数太小无法表示，一般可报告成 0。

- **有效值下溢**（significand underflow）：处理有效值对齐时，可能有数字被移出右端最低位而丢失。下面将要讨论，此时需要某种形式的舍入。
- **有效值上溢**（significand overflow）：同符号的两个有效值相加可能导致最高有效位的进位。这可通过重新对齐来修补，后面将说明。

表9-5 浮点数和算术操作

浮点数	算术操作
$X=X_S\times B^{X_E}$ $Y=Y_S\times B^{Y_E}$	$X+Y=(X_S\times B^{X_E-Y_E}+Y_S)\times B^{Y_E}$ $X-Y=(X_S\times B^{X_E-Y_E}-Y_S)\times B^{Y_E}$ 当 $X_E\leq Y_E$ $X\times Y=(X_S\times Y_S)\times B^{X_E+Y_E}$ $\frac{X}{Y}=\left(\frac{X_S}{Y_S}\right)\times B^{X_E-Y_E}$

举例：$X=0.3\times10^2=30$

$Y=0.2\times10^3=200$

$X+Y=(0.3\times10^{2-3}+0.2)\times10^3=0.23\times10^3=230$

$X-Y=(0.3\times10^{2-3}-0.2)\times10^3=(-0.17)\times10^3=-170$

$X\times Y=(0.3\times0.2)\times10^{2+3}=0.06\times10^5=6000$

$X\div Y=(0.3\div0.2)\times10^{2-3}=1.5\times10^{-1}=0.15$

9.5.1 浮点加法和减法

浮点算术中，加、减法要比乘、除法更复杂，因为它需要对齐。加、减法有4个基本阶段：

（1）检查0。

（2）对齐有效值。

（3）加或减有效值。

（4）规格化结果。

图9-22是一个典型的流程图。下面逐步说明浮点加、减法所需要的主要操作。假定格式类似于图9-21中的格式。为了加或减操作，两个操作数必须传送到可被算术逻辑单元（ALU）使用的寄存器中。若格式包括一个隐藏有效位，则此位要先变成显式再操作。

步骤1：检查0。因为加法和减法除了符号不同外基本上是相同的。因此若是一个减法，过程一开始就改变减数的符号。接着，若有一个操作数是0，那么另一个操作数就是结果。

步骤2：对齐有效值。下一步是操纵数使两个阶值相等。

要说明为什么需要这样做，可考虑十进制加法：

$$(123\times10^0)+(456\times10^{-2})$$

很明显，不能仅加有效值。数字首先要设置成对等位置，即第二个数的4需与第一个数的3对齐。在两个阶值相等的条件下两个数才能相加，这是数学的基本要求。于是有：

$$(123\times10^0)+(456\times10^{-2})=(123\times10^0)+(4.56\times10^0)=127.56\times10^0$$

实现有效值对齐，或右移较小的数（增加它的阶值）或左移较大的数（减少它的阶值）。无论哪种操作都可能导致数字丢失，一般来说，右移较小的数而丢失的数字，所造成的影响要相对小些。因此，对齐通过重复右移较小数有效值的幅值部分1位，并将其阶值加1，直到两个阶值相等来实现（注意，若隐含的底是16，移动一个数字相当于移动4位二进制值）。若此过程导致有效值变为0，则另一个数即为结果。于是，若两个数的阶值差别非常大，则较小的数丢失。

步骤3：加法。将两个有效值相加，相加时要考虑它们的符号。因为符号可能不同，结果有可能是0。这里也可能出现有效值上溢1个数字，若是这样，则有效值要右移，阶值增加1。阶值加1又可能发生阶值上溢；如果发生阶值上溢，此时应终止操作并报告。

步骤4：规格化。最后一步是规格化结果。左移有效值直到最高有效数字（位，或4位——对底为16而言）为非零。每次左移都引起阶值相应减1。这种情况有可能出现阶值下溢。最后，必须对结果进行舍入，然后报告结果。我们将舍入的讨论推迟到乘除法讨论之后再进行。

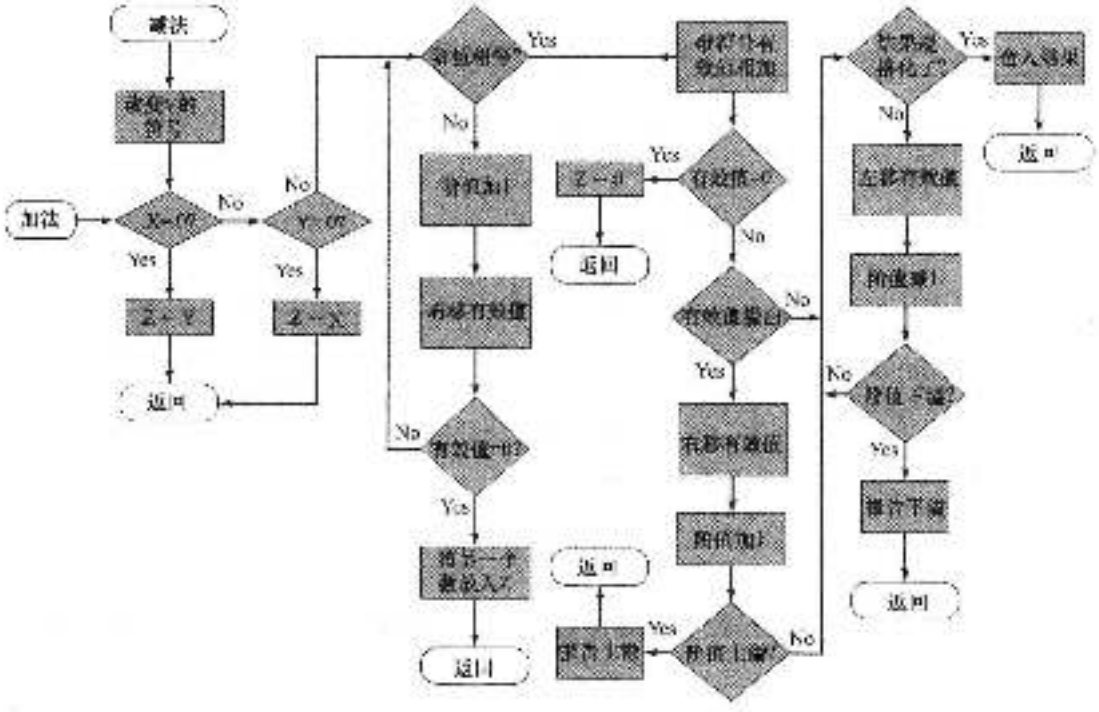


图9-22 浮点加法和减法（Z← X+Y）

9.5.2 浮点乘法和除法

浮点乘、除法要比加、减法简单，由如下讨论可看出。

我们首先考虑乘法，其过程如图9-23所示。无论哪个操作数是0，乘积即为0。下一步是阶值相加。若阶值是移码表示的形式，两个阶值的和将会包含两倍的偏移量，故应从和中减去一个偏移量。阶值相加可能会出现阶值上溢或下溢，此时应结束乘法并报告。

若积的阶值在一个恰当的范围内，则下一步应是有效值相乘，包括它们的符号一起考虑。有效值相乘与整数乘法的完成方式相同。此例中我们使用的是“符号-幅值”表示法，不过对于2的补码表示法而言，其细节也是类似的。积的长度将是被乘数和乘数的长度的两倍，多余的位将在舍入期间丢失掉。

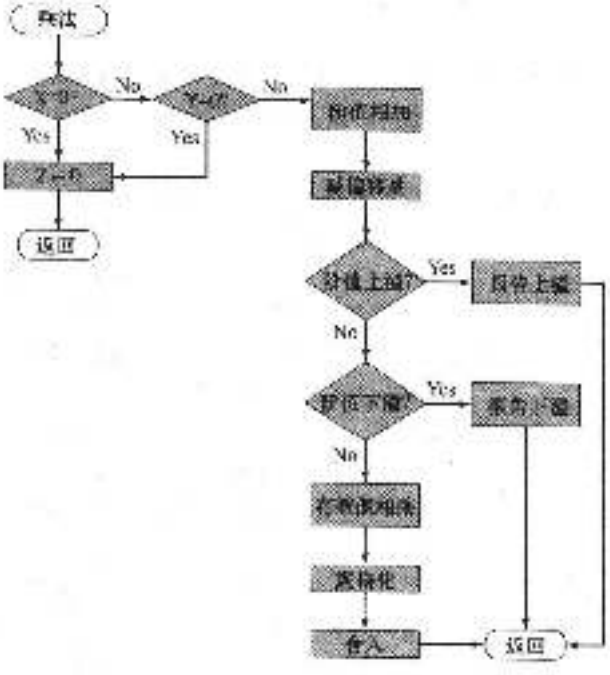


图9-23 浮点乘法（Z← X×Y）

得出乘积之后，下一步则是结果的规

格化和舍入处理，同加、减法时的一样。注意，规格化可能导致阶值下溢。

最后考虑图 9-24 所示的除法流程图。第一步是测试0。若除数是0，或报告出错，或认为商是一个无穷大，这取决于具体的实现。若被除数是0，则结果是0。下一步是被除数的阶值减除数的阶值。这个过程把偏移量减掉了，故必须在阶值相减后再加上偏移量。然后检查阶值是否出现了上溢或下溢。

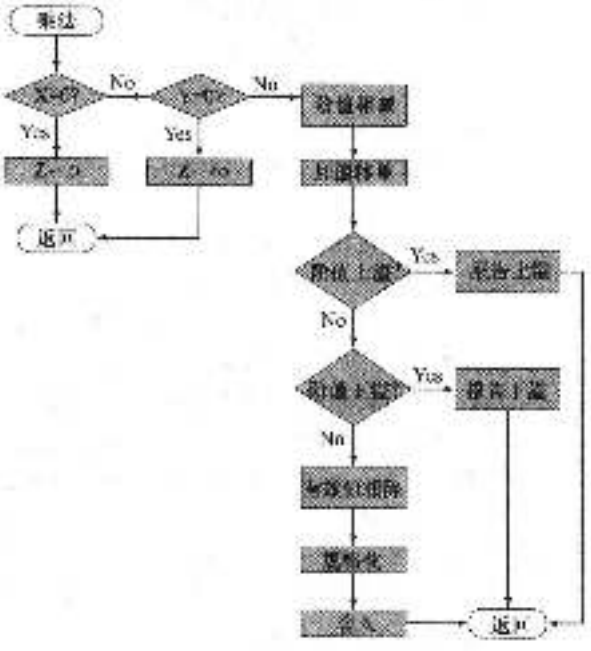


图 9-24 浮点除法（Z←X/Y）

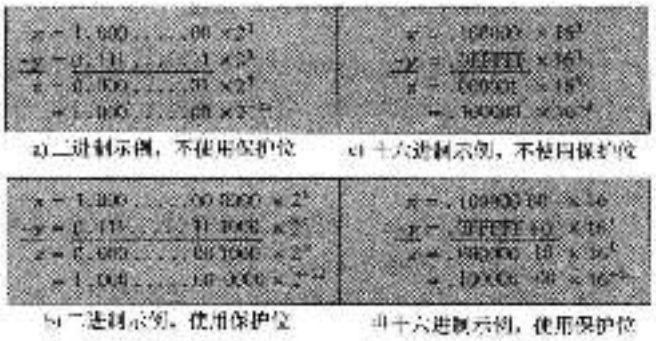
下一步是有效值相除。接下去是规格化和舍入处理。

9.5.3 精度考虑

1. 保护位

我们提到过，在浮点运算之前，每个操作数的阶值和有效值要装入算术逻辑单元的寄存器中。有效值的装入情况是，寄存器的长度几乎总是大于有效值位长与一个隐藏位（若使用）之和。寄存器包含的这些附加位叫做保护位（guard bits），有效值装入时，这些位以0填充，用于扩展有效值的右端。

使用保护位的理由说明于图 9-25。考虑 IEEE 754 格式的数，它有 24 位有效值，包括了二进制小数点左边的一个隐藏位。$x = 1.00\cdots00 \times 2^1$ 和 $y = 1.11\cdots11 \times 2^0$ 是两个值很靠近的数。$x - y$ 时，较小的数 y 必须右移一位以对齐阶值。这示于图 9-25a。此过程中，y 丢失了一位有效数；结果是 2^{-23}。同样的过程重复于图 9-25b，但此时附加有 4 位保护位。现在，最低有效值位不会由于对齐而丢失了，并且结果是 2^{-23}，与前一答案相比差了一半。当基数是 16 时，精度的



a) 二进制示例，不使用保护位　c) 十六进制示例，不使用保护位

b) 二进制示例，使用保护位　d) 十六进制示例，使用保护位

图 9-25 保护位的使用

损失可能更大。正如图 9-25c 和图 9-25d 所示，结果差了 16 倍。

2. 舍入

影响结果精度的另一细节是舍入策略（rounding policy）。对有效值操作的结果通常保存在更长的寄存器中。当结果转换回浮点格式时，必须要去掉多余的位。

已开发出几种技术用于舍入处理。实际上，IEEE 754 标准已列出 4 种可供选择的方法。

- 就近舍入（round to nearest）：结果被舍入成最近的可表示的数。
- 朝 $+\infty$ 舍入（round toward $+\infty$）：结果朝正无穷大方向向上舍入。
- 朝 $-\infty$ 舍入（round toward $-\infty$）：结果朝负无穷大方向向下舍入。
- 朝 0 舍入（round toward 0）：结果朝 0 舍入。

让我们依次考察这些策略。就近舍入是标准列出的默认舍入方式，其定义如下：最最近无限精度结果的可表示值将被提交。

例如，如果超出可保存的 23 位的多余位是 10010。则多余位的值超过了最低可表示位值的一半。这种情况下，正确的答案是最低可表示位加 1，即“入”到下一个可表示的数。现在，考虑多余位是 01111。这种情况下，多余位的值小于最低可表示位值的一半。正确的答案是简单去掉多余位（截断，truncate），这具有舍到下一个可表示数的效果。

标准也规定了多余位是 10000……这种特殊情况的处理。此时结果位于两个可表示数值的严格中点。一种可选的方法是截断，因为该操作最简单。但这个简单方法的缺点是，它会给一个计算序列带来小的但可累积的偏差效应。另一种可选方法是，基于一随机数来决定是舍还是入，于是平均而言无偏差积累效应。反对这种方法的意见是，它不能产生一个可预期的确定的结果。IEEE 采取的方法是强迫结果是偶数：若计算结果是严格位于两个可表示数的正中间，则当结果的最低可表示位是 1 时，结果向上入；当最低可表示位是 0，结果向下舍。

接下来两个可选方法是朝正或负的无穷大方向舍入。它们在实现一种称为区间算术（interval arithmetic）的技术中是有用的。通过为每个结果产生两个值，区间算术为监视和控制浮点运算错误提供了一种方法。这两个值对应于含有真正结果的区间的上下两端。区间宽度，即上下两端之差，指示结果的精确度。若区间端点不可表示，则对它要进行舍入处理。虽然区间宽度随实现的不同而变动，但已有许多算法能产生窄的区间。如果上、下边界的范围足够窄，则得到了一个足够精确的结果。如果不是这样，那么至少我们能知道这个事实，并能基于此进行进一步的分析。

标准描述的最后一种舍入技术是朝 0 舍入。它实际上是简单的截断，不管多余位。这确实是最简单的技术。然而，被截断值的幅值总是小于或等于更精确原值的幅值，这会在计算中产生一致的向下偏差。这是比我们讨论过的任何偏差都更为严重的偏差，因为这种偏差对任何产生非零多余位的运算都有影响。

9.5.4 二进制浮点算术的 IEEE 标准

IEEE 754 远超出格式的简单定义，它还规定了特殊情况及相应处理方法，以使浮点算术能产生一致的、可预期的结果，并且与硬件平台无关。其中一个方面是我们已讨论过的舍入处理。本小节介绍其他三个论题：无穷大、非数（NaN）和非规格化数。

1. 无穷大

无穷大在实数算术中作为限界来对待，对无穷大可给出如下解释：

$-\infty <$（任何有限的数）$< +\infty$

除后面讨论的几种特殊情况之外，任何涉及无穷大的算术运算都将产生明确的结果。

例如：

$5 + (+\infty) = +\infty \qquad 5 \div (-\infty) = +0$

$5-(+\infty)=-\infty$　　$(+\infty)+(+\infty)=+\infty$

$5+(-\infty)=-\infty$　　$(-\infty)+(-\infty)=-\infty$

$5-(-\infty)=+\infty$　　$(-\infty)-(+\infty)=-\infty$

$5\times(+\infty)=+\infty$　　$(+\infty)-(-\infty)=+\infty$

2. 静默式和通知式非数

非数（NaN）是以浮点格式编码的符号实体，它有两种类型：静默式和通知式。通知式（signaling）NaN在每次它作为一个操作数出现时，就产生一个无效操作异常的通知。通知式NaN可为未初始化的变量，以及不属于标准的算术类增强，提供赋值。静默式（quiet）NaN可以通过几乎所有算术操作而不给出异常通知。表9-6指出了那些能产生静默式NaN的操作。

注意，两类NaN具有同样的一般格式（见表9-4）：全1的阶值和非0的有效值。非0有效值的实际位模式取决于具体实现；有效值能用来区分通知式NaN和静默式NaN，以及指定具体的异常条件。

表9-6　产生静默式NaN的操作

运算类型	产生静默式NaN的操作	运算类型	产生静默式NaN的操作
任何	对通知式NaN的任何操作	乘	$0\times\infty$
加或减	无穷大幅值相减： $(+\infty)+(-\infty)$ $(-\infty)+(+\infty)$ $(+\infty)-(+\infty)$ $(-\infty)-(-\infty)$	除	$0/0$ 或 ∞/∞
		求余	x REM 0 或 ∞ REM y
		平方根	$\sqrt{x}$ 其中，$x<0$

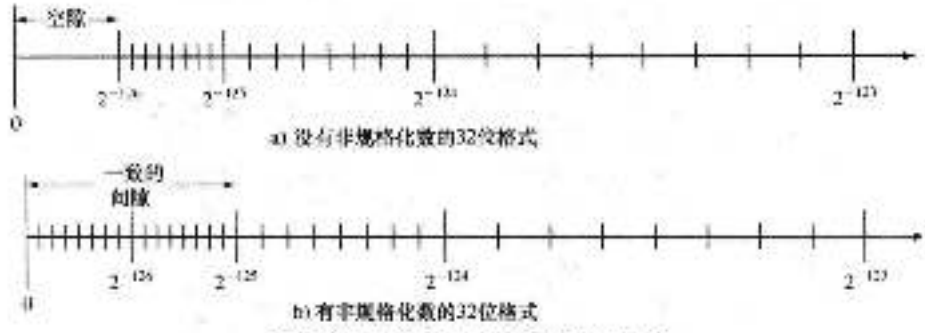


图9-26　IEEE 754非规格化数的效应

3. 非规格化数

IEEE 754包括了一种非规格化数（denormalized number），用于处理阶值下溢情况。当结果的阶值太小（大幅值的负阶值）时，通过右移进行非规格化：每次右移阶值增1，直到阶值落在可表示范围之内。

图9-26说明了加入非规格化数后的效果。可表示的数能以 $[2^n, 2^{n+1}]$ 的区间分组，每个区间内，数的阶值部分保持不变而有效值变动，在区间内产生一致间隔的可表示数。当向0靠近时，每一后向区间是前一区间宽度的一半，但可表示数的数目是相同的。于是，随着向0逼近，可表示数的密度增加了。然而，若只使用规格化数，则在最小规格化数和0之间有一个空隙被浪费了。以IEEE的32位格式而言，每个区间有 2^{23} 个可表示的数，最小可表示的正数是 2^{-126}。使用非规格化数后，$2^{23}-1$ 个附加的数以一致的密度添加到0与 2^{-126} 之间。

非规格化数的使用，被称为逐级下溢（gradual underflow）[COON81]。若无非规格化数，则最小可表示的非0数与0之间的间隙远宽于最小可表示的非0数与下一个更大非0数之间的间

腔。逐级下溢填充了这个间隙，并将阶值下溢的影响降低到与规格化数向舍入相当的级别上。

9.6 推荐的读物和 Web 站点

[ERCE04] 和 [PARH00] 是计算机算术的两本优秀著作，它们全面详尽地介绍了本章所有主题。[FLYN01] 提供了实际设计和实现问题的有益讨论。对于一个想认真学习计算机算术的学生来说，一份必备的参考书是两卷的 [SWAR90]，其卷 I 是 1980 年出版的，提供了计算机算术基础的很多重要论文（其中一些是其他地方很难得到的）；卷 II 包含了更近的论文，覆盖了计算机算术的理论、设计和实现等诸多方面。

对于浮点算术，[GOLD91] 正如其书名 "每个计算机科学家所应该知道的浮点算术知识（What Every Computer Scientist Should Know About Floating-Point Arithmetic）" 所暗，是任何计算机科学工作者都应具备的浮点算术知识。另外，[KNUT98] 含有此主题的优秀论述，亦介绍了计算机整数算术。[OVER01，EVEN00a，OBER97a，OBER97b，SODE96] 等对于更深入探讨计算机算术是有价值的。[KUCK77] 给出了浮点算术舍入方法的良好讨论。[EVEN00b] 考察了 IEEE 754 的舍入处理。

[SCHW99] 介绍了将基数 16 和 IEEE 754 浮点算术集成到同一浮点单元的 IBM S/390 处理器。

ERCE04 Ercegovac, M., and Lang, T. *Digital Arithmetic*. San Francisco: Morgan Kaufmann, 2004.

EVEN00a Even, G., and Paul, W. "On the Design of IEEE Compliant Floating-Point Units." *IEEE Transactions on Computers*, May 2000.

EVEN00b Even, G., and Seidel, P. "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication." *IEEE Transactions on Computers*, July 2000.

FLYN01 Flynn, M., and Oberman, S. *Advanced Computer Arithmetic Design*. New York: Wiley, 2001.

GOLD91 Goldberg, D. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys*, March 1991.

KNUT98 Knuth, D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1998.

KUCK77 Kuck, D.; Parker, D.; and Sameh, A. "An Analysis of Rounding Methods in Floating-Point Arithmetic." *IEEE Transactions on Computers*, July 1977.

OBER97a Oberman, S., and Flynn, M. "Design Issues in Division and Other Floating-Point Operations." *IEEE Transactions on Computers*, February 1997.

OBER97b Oberman, S., and Flynn, M. "Division Algorithms and Implementations." *IEEE Transactions on Computers*, August 1997.

OVER01 Overton, M. *Numerical Computing with IEEE Floating Point Arithmetic*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2001.

PARH00 Parhami, B. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford: Oxford University Press, 2000.

SCHW99 Schwarz, E., and Krygowski, C. "The S/390 G5 Floating-Point Unit." *IBM Journal of Research and Development*, September/November 1999.

SODE96 Soderquist, P., and Leeser, M. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys*, September 1996.

SWAR90 Swartzlander, E., editor. *Computer Arithmetic, Volumes I and II*. Los Alamitos, CA: IEEE Computer Society Press, 1990.

推荐的 Web 站点

- IEEE 754：IEEE 754 文档，包括相关论文和出版物以及对有关计算机算术的一些有用链接。

9.7 关键词、思考题和习题

关键词

Arithmetic and Logic Unit (ALU)：算术逻辑单元
arithmetic shift：算术移位
base：基值（或称为底）
biased representation：移码表示法
denormalized number：非规格化数
dividend：被除数
divisor：除数
exponent：指数（阶值）
exponent overflow：阶值上溢
exponent underflow：阶值下溢
fixed-point representation：定点表示法
floating-point representation：浮点表示法
guard bits：保护位
mantissa：尾数（有效值）
minuend：被减数
multiplicand：被乘数
multiplier：乘数
negative overflow：负上溢
negative underflow：负下溢
normalized number：规格化数
ones complement representation：1 的补码（反码）表示法
overflow：上溢（或溢出）
partial product：部分积
positive overflow：正上溢
positive underflow：正下溢
product：积
quotient：商
radix point：小数点
remainder：余数
rounding：舍入
sign bit：符号位
significand：有效值
significand overflow：有效值上溢
significand underflow：有效值下溢
sign-magnitude representation：符号-幅值表示法
subtrahend：减数
twos complement representation：2 的补码表示法

思考题

9.1 简要解释如下表示法：符号-幅值，2 的补码，移码。
9.2 下面三种表示法中如何确定一个数是否为负数：符号-幅值，2 的补码，移码？
9.3 对于 2 的补码数，符号扩展规则是什么？
9.4 在 2 的补码表示法中，如何求得一个整数的负数？
9.5 一般而言，什么情况对一个 n 位整数求（2 的）补时能产生同一整数？
9.6 一个数的 2 的补码表示与对一个数求（2 的）补有何不同？
9.7 如果将 2 的补码数看作是无符号整数来进行加法运算，则以 2 的补码数来解释此“和”数，其结果是正确的。但这不能用于乘法运算。为何？
9.8 浮点表示法中，数的 4 个基本元素是什么？
9.9 浮点数的阶值部分采用移码表示法有什么好处？
9.10 正上溢、阶值上溢和有效值上溢，三者的区别是什么？
9.11 浮点加减法的基本要素是什么？
9.12 给出使用保护位的理由。
9.13 列出浮点数运算结果的 4 种舍入方法。

习题

9.1 以 16 位长的二进制符号-幅值法和 2 的补码表示法分别表示如下十进制数：+512；-29。
9.2 给出如下 2 的补码数的十进制值：1101011；0101101。
9.3 有时会碰到的另一种二进制整数表示法是 1 的补码表示法。正整数以与“符号-幅值法”相同的方法来表示，负整数以对应正数的各位取反来表示。
(a) 以类似于等式（9.1）和（9.2）的步骤，使用位的加权和给出 1 的补码表示法的定义。
(b) 1 的补码表示法所能表示的数的范围是什么？
(c) 若以 1 的补码完成加法运算，请定义算法。
注意：虽然 1 的补码算术在 20 世纪 60 年代就从计算机硬件中消失了，但还在 IP（互联网协议，Inter-

net Protocol）和 TCP（传输控制协议，Transmission Control Protocol）中用于校验和的计算。

9.4 将符号-幅值和 1 的补码两种表示法的特征加入到表 9-1 中。

9.5 考虑对一个二进制字的如下操作：从最低有效位开始，逐个复制值为 0 的位，直到遇到第一个值为 1 的位，并也将它复制过来；然后对此位之后的各位取反再复制。生成的结果是什么？

9.6 在 9.3 节我们曾将 2 的补操作定义成：为求 X 的 2 的补，将 X 的各位取反然后加 1。

（a）请说明如下定义是等价的：对一个 n 位二进制整数 X，X 的 2 的补可通过将 X 看作是一个无符号整数然后计算 $(2^n - X)$ 来获得。

（b）请通过用顺时针的移动来验证减法，展示图 9-5 也能对（a）中所提出定义类似图形化的支持。

9.7 对于 n 位数字的 r 进制数 N，"r 的补" 定义为：当 $N \neq 0$ 时，N 的 r 的补为 $r^n - N$，当 $N = 0$ 时，N 的 r 的补为 0。根据此定义求十进制数 13 250 的 10 的补。

9.8 使用 10 的补码算术计算 72 530 − 13 250。假定其规则类似于 2 的补码算术规则。

9.9 考虑如下的两个 n 位 2 的补码数相加：

$$x_{n-1}x_{n-2}\cdots x_0 = x_{n-1}x_{n-2}\cdots x_0 + y_{n-1}y_{n-2}\cdots y_0$$

假定上式按位相加时使用了一个位间进位 C_i，它是由 x_i、y_i 和 C_{i-1} 相加产生的。令 v 是指示溢出 $(n-1)$ 的二进制变量。现只考虑首位相加的各种情况，请填写下表中的输出值。

输入	x_{n-1}	0	0	0	0	1	1	1	1
	y_{n-1}	0	0	1	1	0	0	1	1
	c_{n-2}	0	1	0	1	0	1	0	1
输出	z_{n-1}								
	v								

9.10 将下面的数用 8 位 2 的补码表示，再进行计算：

（a）6 + 13　　（b）−6 + 13　　（c）6 − 13　　（d）−6 − 13

9.11 使用 2 的补码算法，求如下的差：

（a）111000 − 110011　　（b）11001100 − 101110　　（c）111100001111 − 110011110011　　（d）11000011 − 11101000

9.12 下面这个十溢的定义对于 2 的补码运算是否有效？

若向最左位的进位与最左位向上的进位，其异或值是 1 时，则有上溢；否则便没有上溢。

9.13 比较图 9-9 和图 9-12，为什么后者不使用 C 位？

9.14 已知以 2 的补码表示有 $x = 0101$ 和 $y = 1010$（即 $x = 5$，$y = -6$）。请以布斯算法计算积 $p = x \times y$。

9.15 使用布斯算法计算 23（被乘数）×29（乘数），这里每个数用 6 位表示。

9.16 试证明，基值为 B 的两个 n 位数字的数相乘，其积不会多于 $2n$ 位数字。

9.17 通过说明图 9-15 的除法计算各步，验证图 9-16 无符号数除法算法的有效性。请使用类似于图 9-17 的说明格式。

9.18 在 9.3 节中描述的除法算法称为恢复余数算法，因为一个不成功的减之后必须恢复 A 寄存器中的值。另一较为复杂的方法，称为不恢复余数除法，它避免了不必要的减和加。请为后一种方法设计算法。

9.19 在计算机整数运算时，两个整数 J 和 K 的商 J/K 小于或等于平常（手工运算）的商，是真还是假？

9.20 以 2 的补码表示法完成 −145 除以 13，使用 12 位字长。应用 9.3 节描述的算法。

9.21 （a）考虑一个十进制数的定点表示法，其隐含的小数点能在任何位置（如，在最低有效数字的右边，在最高有效数字的右边等）。若近似表示普朗克（Planck）常数（6.63×10^{-27}）和阿伏伽德罗（Avogadro）常数（6.02×10^{23}），并要求两数的隐含小数点在同一位置。请问这需要多少位十进制数？

（b）考虑一个十进制数的浮点表示法，其阶值以移码表示。偏移量是 80。假定表示已是规格化了。请问以这种浮点表示法表示以上两常数需要多少位十进制数？

9.22 假定阶值 e 限定在 $0 \leq e \leq x$ 范围内，偏移量是 q，底是 b，有效值长度是 p 个数字。

（a）能表示的最大和最小正值是什么？

（b）作为规格化的浮点数，能表示的最大和最小正值是什么？

9.23 以 IEEE 32 位浮点格式表示如下的数：

(a) −5 (b) −6 (c) −1.5 (d) 384 (e) 1/16 (f) −1/32

9.24 下列各数使用了IEEE 32位浮点格式，对应的十进制值是什么？

(a) 1 10000011 110 0000 0000 0000 0000 0000

(b) 0 0111110 101 0000 0000 0000 0000 0000

(c) 0 1000 0000 000 0000 0000 0000 0000 0000

9.25 考虑一个简化的7位IEEE浮点格式，3位阶值，3位有效值。请列出全部的127个值。

9.26 IBM 32位浮点格式为：7位阶值，隐含的底是16，阶值的偏移量是64（十六进制的40），规格化的浮点数要求最左的十六进制数字不为零，小数点隐含位于最左位的左边。请以该格式表示如下的数：

a. 1.0	c. 1/64	e. −15.0	g. 7.2×10^{75}
b. 0.5	d. 0.0	f. 5.4×10^{-79}	h. 65535

9.27 5BCA000是以十六进制表述的IBM格式的浮点数，它的十进制值是多少？

9.28 下列情况偏移量应为多少？

(a) 6位字段中底为2的阶值。

(b) 7位字段中底为8的阶值。

9.29 为图9-21b的浮点格式画出类似于图9-19b那样的数轴表示。

9.30 考虑一种浮点格式，它有8位移码阶值和23位有效值。请以此格式表示下列数：

(a) −720 (b) 0.645

9.31 正文中提到，32位格式最多能表示2^{32}个不同的数。用IEEE 32位浮点格式最多能表示多少不同的数，为什么？

9.32 任何浮点表示法在计算机中只能精确地表示某些实数，其他所有数必定是近似的。若A'是存储的近似值，A是原实际值，则相对误差r可表示成：

$$r = \frac{A - A'}{A}$$

请以如下浮点格式表示十进制数+0.4，并计算它的相对误差。

- 底为2。
- 阶值：移码，4位。
- 有效值：7位。

9.33 设$A = 1.427$，若A被截断到1.42，求其相对误差；若A被入到1.43，求其相对误差。

9.34 当人们说到浮点运算的不精确性时，常把误差归结于两个相近数相减时出现的抵消（cancellation）上。但实际上x和y近似相等时，其$x - y$的差可没有误差地精确得到。那么人们实际指的是什么呢？

9.35 数值A和B在计算机中是以其近似值A'和B'来存储的，忽略任何可能有的截断或舍入误差，说明积的相对误差大约是因子相对误差之和。

9.36 在计算机计算中，当两个几乎相等的数相减时会出现最严重的误差。考虑$A = 0.22288$和$B = 0.22211$。计算机将所有值都截断到4位十进制数字。于是，$A' = 0.2228$和$B' = 0.2221$。

(a) A'和B'的相对误差各是多少？

(b) 对于$C' = A' - B'$，其相对误差又是多少？

9.37 为获得对非规格化和逐级下溢效应的某些感性认识，请考虑一个十进制系统。它的有效数是6位十进制数，它的最小规格化数是10^{-99}。一个规格化的数要求在十进制小数点左边有一位非0的十进制数。完成如下计算并将结果非规格化。对结果进行解释。

(a) $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-43})$

(b) $(2.50000 \times 10^{-96}) \times (3.50000 \times 10^{-90})$

(c) $(5.67834 \times 10^{-97}) - (5.67812 \times 10^{-97})$

9.38 展示如何完成以下浮点加法（有效值截断到4位十进制数）。以规格化的形式表示结果。

(a) $5.566 \times 10^{2} + 7.777 \times 10^{2}$ (b) $3.344 \times 10^{1} + 8.877 \times 10^{-1}$

9.39 展示如何完成以下浮点减法（有效值截断到4位十进制数）。以规格化的形式表示结果。

(a) $7.744 \times 10^{-3} - 6.666 \times 10^{-3}$ (b) $8.844 \times 10^{-1} - 2.233 \times 10^{1}$

9.40 展示如何完成以下浮点计算（有效值截断到4位十进制数）。以规格化的形式表示结果。

(a) $(2.255 \times 10^{1}) \times (1.234 \times 10^{2})$ (b) $(8.833 \times 10^{2}) \div (5.555 \times 10^{4})$

第10章

Computer Organization and Architecture: Designing for Performance, 8E

指令集：特征和功能

本章要点

- 计算机指令最重要的元素是操作码（opcode）。它指明将完成的操作，源和目的操作数的引用方式，并通常隐式指明下一条指令的来源。
- 操作码指定的操作，一般可有如下类型：算术和逻辑运算，在两个寄存器、寄存器和存储器或存储器两个位置之间传送数据，输入/输出（I/O），控制。
- 操作数引用方式指定如何寻找被操作数据的寄存器或存储器的位置。数据类型可以是地址、数值、字符或逻辑数据。
- 各类处理器中的一个普遍的体系结构特征是栈（stack）的使用，栈对程序员来可见的或是不可见的。栈用于管理过程的调用和返回，也可用来提供另一种寻址存储器的方式。栈的基本操作是 PUSH（入栈）和 POP（出栈），以及在栈顶部一或两个位置上完成的操作。一般来说，栈都实现为从高地址向低地址增长。
- 字节可寻址的处理器可分为大端（big endian）、小端（little endian）、双端（bi-endian）这几类。如果多字节的数值是以最高有效字节存于最低地址值的字节来顺序存储，则称为大端；如果它们是以最低有效字节存于最低地址值的字节来顺序存储，则称为小端。既支持大端又支持小端的处理器是双端处理器。

本书讨论的大部分内容对于计算机用户或程序员来说，实际上是看不到的。像使用 Pascal 或 Ada 那样，高级语言的编程人员看到的只是机器低层结构的很少一部分。

对一台计算机而言，把计算机设计人员和计算机编程人员衔接而又区分开的分界是机器指令集。以设计者观点看，机器指令集提出了对中央处理器（CPU）的功能性需求；实现 CPU 的任务大部分都涉及机器指令集的实现。从用户观点来看，选取以机器语言（实际上以汇编语言；见附录 B）编程的用户必定要通晓机器所直接支持的寄存器和存储器结构、数据类型，以及算术逻辑单元（ALU）的功能。

计算机机器语言的描述将大大有助于了解计算机的 CPU。因此，本章和下一章集中讨论机器指令。

10.1 机器指令特征

CPU 的操作由它所执行的指令确定。这些指令称为机器指令（machine instruction）或计算机指令。CPU 能执行的各种不同指令的集合称为 CPU 的指令集（instruction set）。

10.1.1 机器指令要素

每条机器指令必定包含处理器执行该指令所需的信息。图 10-1 是图 3-6 的复制，它表示了指令执行步骤，也隐含地定义了机器指令要素。这些要素如下所示。

- **操作码**（operation code）：指定将要完成的操作（如 ADD、I/O 等）。这些二进制代码常被称为操作码（opcode）。
- **源操作数引用**（source operand reference）：操作会涉及一个或多个源操作数，这是操作所需的输入。
- **结果操作数引用**（result operand reference）：操作可能产生一个结果。

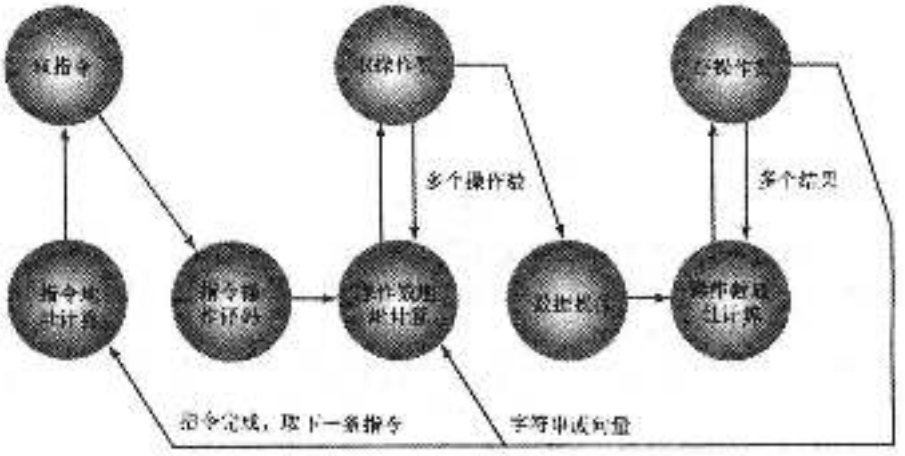


图 10-1　指令周期状态图

- 下一指令引用（next instruction reference）：它告诉处理器这条指令执行完成后到哪儿去取下一条指令。

待取的下一条指令地址可能是一个实地址（real address），也可能是一个虚地址（virtual address），这取决于具体的计算机体系结构。通常来说，这个问题跟指令集无关。大多数情况下，待取的下一条指令就位于当前指令之后，此时指令中没有显式引用。当需要显式引用时，则指令中必须提供主存或虚拟存储器的地址。地址提供的形式在第 11 章讨论。

源和结果操作数可能位于如下 4 个范围内：

- **主存或虚存**：与下一条指令的引用一样，必须提供主存或虚存的地址。
- **处理器寄存器**：除极少数例外，一个处理器总有一个或多个能被机器指令所访问的寄存器。若只有一个寄存器，对它的引用可以是隐式的；若不止一个寄存器，则每个寄存器要指定一个唯一的名字或编号，指令提供所需寄存器的寄存器号。
- **立即数**：操作数的值直接保存在当前执行指令的某个字段中。
- **I/O 设备**：需要 I/O 操作的指令必须指定 I/O 模块或设备。若使用存储器映射（memory-mapped）I/O 方式，那么形式上将是另一个主存或虚存地址。

10.1.2　指令表示

在计算机内部，指令由一个位串来表示。对应于指令的各要素，这个位串划分成几个字段。图 10-2 显示了一个简单的指令格式（instruction format）例子。另一个例子显示于图 2-2 的 TAS 指令格式。大多数指令集使用不止一种指令格式。指令执行期间，一条指令被读入到处理器的指令寄存器（IR）中，处理器必须能从各个指令字段中提取数据来完成所需操作。

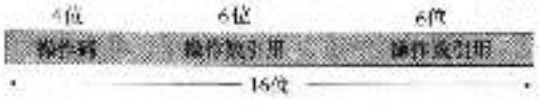


图 10-2　一种简单的指令格式

无论是编程人员还是相关图书的读者，都难与机器指令的二进制表示法直接打交道。于是，普遍使用的是机器指令符号表示法（symbolic representation）。表 2-1 中的 TAS 指令就是这样一个例子。

操作码被缩写成助记符（mnemonic）来表示，一般的例子如下所示：

ADD　加
SUB　减
MUL　乘

DIV 除
LOAD 由存储器装入
STOR 保存到存储器

操作数也可用符号表示。例如，指令：

ADD R, Y

会意味着，它将存储器 Y 位置中的数据值加到寄存器 R 上。在这个例子中，Y 是存储器某位置的地址，R 指的是一个具体寄存器。注意，操作是对位置的内容来完成的，而不是对它的地址。

于是，以符号形式写一个机器语言程序是可能的。每个符号化的操作码都有一个固定的二进制表示，程序员指明每个符号化操作数的位置。例如，程序员可在程序开始处写出下列定义：

X = 513

Y = 514

等等。一个简单的程序能专门用来接收这些符号输入，把操作码和操作数引用转换成对应的二进制形式，并构成二进制的机器指令。

机器语言程序员已很少到几乎没有的程度。今天大多数程序是以高级语言编写的，或至少也是以汇编语言（附录 B 将讨论）来编写的。然而，符号机器语言在描述机器指令方面仍是有用的工具，因此我们还将使用它来学习有关机器指令的知识。

10.1.3 指令类型

考虑像能表示成 BASIC 或 FORTRAN 这样的高级语言指令。例如：

X = X + Y

这条语句指挥计算机将存于 Y 的值加到存于 X 的值并将结果放入 X 中。用机器指令如何完成它？让我们假定 X 和 Y 变量相应于位置 513 和 514。若我们假定有一简单指令集，这个操作能以如下三条指令完成：

（1）将存储器位置 513 的内容装入一个寄存器；

（2）把存储器位置 514 的内容与上述寄存器的内容相加并保存到该寄存器中；

（3）将此寄存器内容存入存储器的 513 位置中。

正如所见，一条单一的 BASIC 指令可能需要三条机器指令。这是高级语言和机器语言之间的典型关系。高级语言使用变量，以简明的代数形式来表达操作。而机器语言是以涉及数据移入移出寄存器的基本形式来表达操作。

按照这个简单例子给我们的启示，让我们考虑一个具体的计算机中必须包括的指令类型。计算机应有能允许用户表达任何数据处理任务的一组指令。另一方面是考虑高级语言的编程能力。任何以高级语言编写的程序，都必须转换成机器语言才能执行。于是，机器指令的集合必须充分，足以表示任何高级语言的指令形式。基于这些考虑，可将指令分类成：

- **数据处理**：算术和逻辑指令；
- **数据存储**：存储器指令；
- **数据传送**：I/O 指令；
- **控制**：测试和分支（branch）指令。

算术指令提供了处理数值型数据的计算能力。逻辑（布尔）指令是对字中的位进行操作，这些位不再看成是数值位，因此提供了处理任何用户想使用的其他数据类型的能力。这些操作主要是对处理器寄存器中数据进行的。因此，必须有存储器指令，以便在存储器和寄存器之间传送数据。需要有 I/O 指令将程序和数据装载到存储器，并将计算结果返给用户。测试指令用于测试数据字的值或计算的状况。分支指令则用于依据判定条件是否成立，转移到另一组指令上去。

本章稍后将更详细地考察各类指令。

10.1.4　地址数目

描述处理器结构的一种传统方式是依据每条指令包含的地址数。随着日益增长的处理器设计的复杂性，这种量度已变得没什么太大意义。然而，对于规划和分析来说，无论如何它还是有用的。

一条指令需要的最大地址数是多少？很显然，算术和逻辑指令要求的操作数最多。实际上，所有算术和逻辑运算或是一元的（一个源操作数）或是二元的（两个源操作数）。于是，我们将最多需要两个地址来访问源操作数。运算的结果必须被存储，这暗示需要第三个地址，用以定义目的操作数（destination operand）。最后，完成一条指令后必须取得下一条指令，这又需要指令地址。

上述推理过程暗示，指令貌似需要有 4 个地址引用：两个源操作数，一个目的操作数，以及下一指令地址。实际上，大多数系统中，指令使用一个、两个或三个操作数地址，下一指令地址为隐含的（由程序计数器得到）。很多系统包含一些特殊指令，它们有更多的操作数。例如，第 11 章所描述的 ARM 处理器，其装载（load）和保存（store）指令组可以在单个指令中使用多达 17 个寄存器操作数。

图 10-3 比较了能用来完成 $Y = (A - B)/(C + D \times E)$ 计算的典型的单地址、双地址和三地址的指令。使用三地址，每个指令可指定两个源操作数位置和一个目的操作数位置。因为我们希望不更改任一操作数的值，故使用 T 为中间结果的暂存。注意，原始的表达式有 5 个操作数，而使用三地址只需要 4 条指令。

三地址的指令格式不普遍，因为指令格式要容纳 3 个地址引用，所以指令格式相对要更长。如果采用双地址指令完成一次二元运算，那么一个地址必须承担双重任务，既用做源操作数又用做结果。于是，SUB Y, B 指令执行 Y − B 计算并将结果保存于 Y。双地址格式降低了空间需求，但也引入了一些麻烦。为避免更改一个操作数的值，要使用 MOVE 指令在完成运算之前将一个值传到最终或临时位置上去。图 10-3 示范例子中，使用两地址指令的话需要 6 条指令。

指令		注释
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

a) 三地址指令

指令		注释
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

b) 两地址指令

指令		注释
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

c) 单地址指令

图 10-3　执行 $Y = (A - B) \div (C + D \times E)$ 的程序

最简单的还是单地址指令。为使它能工作，第二个地址必须是隐含的。这在早先机器中是很普遍的，其隐含地址是被称为累加器（Accumulator，AC）的 CPU 寄存器。累加器提供一个操作数，且结果被保存回累加器。图 10-3 的例子，完成同样的任务需要 8 条单地址指令。

事实上，对于某些指令还可能有“零地址”格式。零地址指令可用于称为栈（stack）的专门存储器组织中。栈是一种后入先出的结构，位于某个已知的位置，并且通常栈顶部至少两个元素是放在 CPU 寄存器中的。于是，零地址指令能访问栈顶两个元素。附录 10A 给出了栈的描述，本章稍后以及第 11 章将进一步说明栈的使用。

表 10-1 总结了具有 0、1、2 或 3 个地址的指令的解释。表中的每种情况都假定了下一指令地址是隐含的，指令的运算都是对两个源操作数进行，并产生一个结果操作数。

每条指令的地址数目是基本的设计决策。每条指令中的地址数目越少，则指令的长度越短，指令也更原始（不需要复杂的 CPU）。另一方面，它又会使程序的总的指令条数更多，而导致执

行时间更长，程序也更长更复杂。还有，在单地址指令和多地址指令之间存在一个重要分界点。对单地址指令来说，程序员通常只有一个通用寄存器（即累加器）可利用。而对于多地址指令，普遍具有多个通用寄存器可用。这就允许某些运算只使用寄存器即可完成。因为寄存器访问要比存储器访问快得多，从而使执行加快。为了灵活性和能够使用多个寄存器，大多数当代计算机采用了双地址和三地址指令的混合方式。

表 10-1 指令地址的利用（非分支指令）

地址数	符号表示	解释	地址数	符号表示	解释
3	OP A, B, C	A← B OP C	1	OP A	AC← AC OP A
2	OP A, B	A← A OP B	0	OP	T←(T-1) OP T

注：AC = 累加器

T = 栈顶

(T-1) = 栈第二个元素的内容

A, B, C = 存储器或寄存器位置

涉及每条指令地址数目选择的设计权衡，被其他因素复杂化了。一个需要考虑的设计问题是引用到存储器位置还是寄存器。因为寄存器的数目总是不太多，因此寄存器引用只需要少数几位即可。还有，正如下一章将会看到的那样，一个机器会有各种寻址方式，为指定这些方式也要占用指令的一位或几位。结果，大多数处理器设计成有多种指令格式。

10.1.5 指令集设计

计算机设计的一个最有趣，也最受关注的方面是指令集的设计。指令集的设计是一件很复杂的事情，因为它影响计算机系统的诸多方面。指令集定义了处理器应完成的多数功能，于是它对处理器的实现有着显著的影响。指令集也是程序员控制处理器的方式。于是，设计指令集时必须考虑程序员的要求。

当你知道，在有关指令集设计的最根本出发点方面还存在某些争议时，你一定会感到惊奇。事实确实如此。最近几年这种分歧的程度实际还在增长。这些最基础的设计出发点中，最重要的包括：

- **操作指令表**（operation repertoire）：应提供多少和什么样的操作，操作将是何等复杂。
- **数据类型**（data type）：对哪几种数据类型完成操作。
- **指令格式**（instruction format）：指令的（位）长度、地址数目、各个字段的大小等。
- **寄存器**（register）：能被指令访问的处理器寄存器数目以及它们的用途。
- **寻址**（addressing）：指定操作数地址的产生方式。

这些出发点是紧密相关的，设计指令集时必须一起考虑。当然，本书只能依次考察它们，但我们也努力说明其相关性。

因为这些主题的重要性，第三部分的许多篇幅将用于介绍指令集设计。在此节综述之后，本章将考察数据类型和操作清单。第 11 章考察寻址模式（其中也包括寄存器的考虑）和指令格式。第 13 章考察精简指令集计算机（RISC）。RISC 体系结构对很多商业计算机中指令集设计的传统方式提出了质疑。

10.2 操作数类型

机器指令对数据进行操作，数据通常分为：地址、数值、字符、逻辑数据。

在本章讨论寻址方式时将看到，地址实际也是一种形式的数据。多数情况下，必须对指令中的操作数引用完成某些计算，才能确定主存或虚拟地址。此时，地址将被看成是无符号整数。

其他普通数据类型是数值、字符和逻辑数据，本章将分别予以简要的考察。除这些之外，某些机器定义了专门的数据类型或数据结构。例如，有些机器的指令可对字符列表或字符串直接操作。

10.2.1　数值

所有机器语言都包括有数值型数据类型。即使在一个非数值数据的处理中，也需要某些数值用做计数、字段宽度等。一般数学所用的数值与计算机存储的数值之间存在一个重大不同，这就是后者是受限的。这表现在两个方面：首先是机器可表示数值的幅值是有限的；其次是在浮点数情况下数值精度是有限的。于是，程序员必须理解溢入、上溢、下溢的意义。

计算机中普遍使用三种类型的数值数据：二进制整数或定点数、二进制浮点数、十进制数。

我们已在第 9 章较详细地考察过前两种类型的数值数据。这里有必要介绍一下十进制数。

虽然计算机内部操作都有二进制属性，但作为系统的使用者——人，却是与十进制打交道。于是，有必要在输入时将十进制转换成二进制，而在输出时将二进制转换成十进制。对于有大量 I/O 需处理而计算相对较少、较简单的应用来说，以十进制形式的数来存储、操作更为合适。为此，一个最普遍的表示法是压缩的或称打包的十进制数（packed decimal number）。[⊖]

压缩的十进制数用 4 位二进制代码表示每个十进制数字。0 = 0000，1 = 0001，…，8 = 1000 和 9 = 1001。注意，4 位二进制代码可有 16 个不同的值，但只上述 10 个代码是有效的。另外 6 个是无效的。为构成一个数，4 位代码紧密排列在一起，通常为多个 8 位的字。例如，246 的编码是 0000 0010 0100 0110。很清楚，这种表示法没有二进制表示法那样紧凑，但它避免了转换开销。负数可以通过在整个压缩十进制数字串的前或后加上一个 4 位的符号数字来表示。标准的符号数字值是 1100，表示正数（+）；1101 表示负数（-）。

多数机器都提供了直接对压缩十进制数进行操作的算术指令，算法类似于 9.3 节的介绍，但必须注意十进制的进位操作。

10.2.2　字符

另一种常用的数据类型是文本或字符串。文本数据对于人来说使用是很方便的，但在数据处理和通信系统中，却不便于以字符形式存储或发送。因为数据处理和通信系统都是被设计来处理二进制数据的。于是，研究人员研制了几种编码方式将字符表示成二进制的位序列。最早的编码或许是莫尔斯电码（Morse Code）。今天使用最广泛的字符编码是国际参考字母表（International Reference Alphabet，IRA），在美国称为 ASCII 码（美国信息交换标准码，American Standard Code for Information Interchange），见附录 F。在这种编码中，每个字符被表示成唯一的 7 位二进制中，于是具有 128 个可表示字符。这个数量比可打印字符数量多，故某些“位样式”（Bit Pattern）用来代表“控制字符”（control character）。一些控制字符用来控制字符的按页打印；而另一些则用来控制通信过程。IRA 编码的字符几乎总是以每字符 8 位来存储和发送的。第 8 位可设置成 0 或用做错误检测的奇偶校验位。后一种情况下，要这样设置该位，使得在每个 8 位中二进制 1 的数目或者总是奇数个（奇校验），或者总是偶数个（偶校验）。

注意，在表 F-1（附录 F）中数字 0 ~ 9 的 IRA 代码的位样式是 011 × × × ×，其中最后 4 位 × × × × 恰恰正是 0000 到 1001，即压缩十进制数的编码。因此在 7 位 IRA 代码和 4 位压缩十进制表示之间转换是非常方便的。

⊖ 教科书经常将其称为二进制编码的十进制数（Binary Coded Decimal，BCD）。严格地说，BCD 指的是用 4 位编码十进制数字，而压缩的十进制数指的是用一字节存入两个 BCD 编码的十进制数。

用于字符编码的另一类代码是 EBCDIC（Extended Binary Coded Decimal Interchange Code，扩展的二进制编码的十进制交换码）。它是一种 8 位编码，被用于 IBM 制造的大型机中。与 IRA 一样，EBCDIC 与压缩十进制是有互换性的，代码 11110000 到 11111001 表示数字 0～9。

10.2.3 逻辑数据

正常情况下，每个字或其他可寻址单元（字节、半字等）是作为一个单一数据单元看待的。然而，某些时候需要将一个 n 位单元看成是由 n 个 1 位项组成，每项有值 0 或 1。当以这种方式看待数据时，该数据就被认为是逻辑数据。

这种位排列观点有两个优点。第一，有时我们希望存储一个布尔或二进制数据项序列，序列中的每项只能取值 1（真）或 0（假）。逻辑数据对于这种情况而言能实现存储器最有效的使用。第二，逻辑数据有利于实现对数据项的具体位进行操纵。例如，如果浮点运算是以软件实现的，那么我们需要能在某些操作中移动有效位。另一例子是，由 IRA 转换到压缩十进制时，我们需要提取出每字节的最右边 4 位。

注意，在上面例子中，同一个数据有时看作是逻辑数据，而有时看作是数值或文本。数据单元的类型由当前在它上面正在完成的操作所确定。在高级语言中（如 Pascal 语言），一般不是这种情况，而在机器语言中几乎总是这种情况。

10.3 Intel x86 和 ARM 数据类型

10.3.1 x86 数据类型

x86 能处理 8 位（字节）、16 位（字）、32 位（双字）、64 位（四字）和 128 位（双四字）各种长度的数据类型。为允许最大的数据结构灵活性和最有效地使用存储器，字不需要在偶数地址上对齐，双字也不需要在 4 倍整数地址上对齐，四字也不需要在 8 倍整数地址上对齐，其他类推。然而，当经由 32 位总线存取数据时，数据传送是以双字为单位进行的，双字的起始地址是能被 4 整除的。处理器要将对于未对齐值的访问请求，转换成一系列的总线传送请求。像所有的 80x86 机器一样，x86 也采用小端序（little-endian ordering）风格，即最低有效字节存于最低地址中（见本章后面的附录 10B 对端序的讨论）。

字节、字、双字、四字和双四字称为常规数据类型。另外，x86 支持一系列的特殊数据类型，这些特殊数据类型只被特殊指令所识别和操作。表 10-2 总结了所有这些数据类型。

表 10-2 x86 数据类型

数据类型	说 明
常规	字节、字（16 位）、双字（32 位）、四字（64 位）和双四字（128 位）可包含任意二进制数据
整数	字节、字或双字中的有符号二进制值，使用 2 的补码表示法
序数（Ordinal）	字节、字或双字中的无符号整数
未压缩的二进制编码的十进制数（BCD）	范围 0～9 的 BCD 数字表示，每字节一个数字
压缩的 BCD	每字节表示两个 BCD 数字，每字节值是 0～99
近指针	表示段内偏移的 16 位、32 位或 64 位有效地址，可用于不分段存储器中的所有指针和分段存储器中的段内访问
远指针	由 16 位段选择符（segment selector）和 16 位、32 位或 64 位偏移量组成的逻辑地址。远指针可用于分段存储中的内存引用，其中被访问的段标识必须被显式指定
位字段（Bit field）	一个连续的位序列，每位位置都认为是一个独立的单位。像以任何字节的任何位位置开始一个位字段，位字段最长可为 32 位

（续）

数据类型	说　明
位串	一个连续位的序列，可包含 0 到 $2^{32}-1$ 个位
字节串（Byte string）	一个连续的字节、字或双字的序列，可包含 0 到 $2^{32}-1$ 个字节
浮点数	见列 10-4
压缩的 SIMD（单指令多数据）	压缩的 64 位或 128 位数据类型

图 10-4 展示了 x86 数值数据类型。有符号整数是以 2 的补码表示的，可以是 16、32 或 64 位长。浮点数类型实际指可被浮点运算单元使用和可被浮点指令操作的一组数据类型。三种浮点数表示都是符合 IEEE 754 标准的。

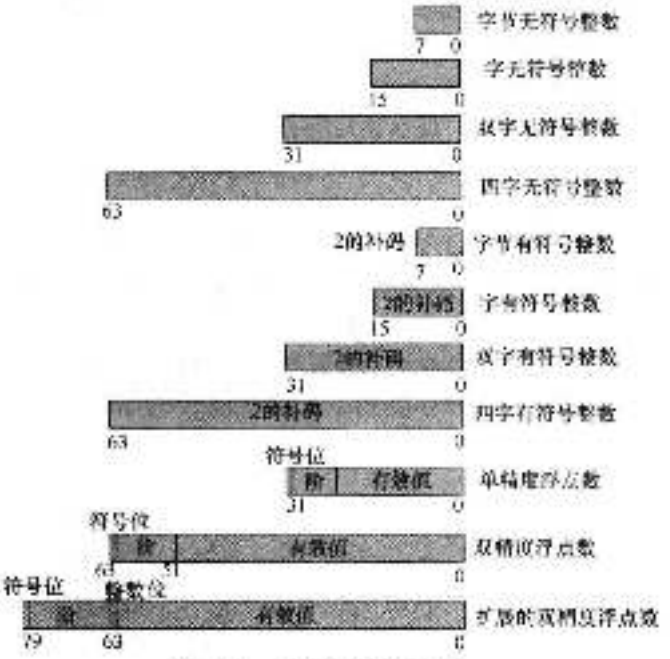


图 10-4　x86 数值数据格式

压缩的 SIMD（single-instruction-multiple-data，单指令多数据）数据类型是为了优化多媒体应用的性能，作为一种对指令集的扩展，添加到 x86 体系结构中来的。这些扩展包括 MMX（multimedia extension，多媒体扩展）和 SSE（streaming SIMD extension，流式 SIMD 扩展）。基本的思想是把多个操作数打包为一个内存引用项，并且并行地操作这些操作数，从而提高性能。压缩的 SIMD 数据类型包括：

- **压缩的字节和压缩的字节整数**：多个字节被打包到一个 64 位的四字（quadword）或 128 位的双四字（double quadword）中，并被当作位字段或整数。
- **压缩的字和压缩的字整数**：多个 16 位的字被打包到一个 64 位的四字（quadword）或 128 位的双四字（double quadword）中，并被当作位字段或整数。
- **压缩的双字和压缩的双字整数**：多个 32 位的字被打包到一个 64 位的四字（quadword）或 128 位的双四字（double quadword）中，并被当作位字段或整数。
- **压缩的四字和压缩的四字整数**：两个 64 位的字被打包到一个 128 位的双四字（double quadword）中，并被当作位字段或整数。
- **压缩的单精度浮点数和压缩的双精度浮点数**：四个 32 位的单精度浮点数或两个 64 位双精度浮点数被打包到一个 128 位的双四字（double quadword）中。

10.3.2　ARM 数据类型

ARM 处理器支持 8 位（字节）、16 位（半字）、32 位（字）各种长度的数据类型。通常，半字数据的访问要对齐到半字地址，字数据的访问要对齐到字地址。对于未对齐的访问请求，ARM 提供了 3 种可选方案：

- 默认情况：
  - 非对齐的地址将被截断，即当访问字时，地址位 bits [1:0] 会被当作 0，而访问半字

时，地址位 bits [0] 被当作 0。

- 载入单个字的 ARM 指令遇到非字对齐的地址时，会以此地址对字对齐的数据循环右移一个，两个或三个字节，具体操作根据非对齐地址最低两位的值而定。

- **对齐检查**：如果设置了相应的控制位，试图访问非对齐地址时，会产生一个数据取消（data abort）信号，表明发生了一个非对齐访问错误。
- **非对齐访问**：如果设置了允许非对齐访问这一选项，处理器将通过一次或多次内存访问，获得非对齐地址所指定的字节，并返回给程序。

所有这三种数据类型（字节、半字、字）都支持使用无符号表示。此时，数据所表示的值是一个无符号，非负的整数。所有数据类型也支持使用 2 的补码表示有符号整数。

大部分的 ARM 处理器实现都不支持浮点硬件，这可以节省功耗和芯片面积。如果这些处理器需要浮点运算，那么就只能通过软件来提供。ARM 可以带一个浮点协处理器，利用浮点协处理器可以支持 IEEE 754 标准所规定的单精度和双精度浮点数据类型。

端序支持

程序可以通过 SETEND 指令设置和清除 ARM 处理器中系统控制寄存器的端序状态位（E 位）。E 位定义了采用哪种端序来装载和保存数据。图 10-5 通过字装载和保存操作显示了 E 位的功能。当系统设计人员需要使用不同端序来访问操作系统和环境系统数据结构时，端序选择机制为系统设计人员在动态装载和保存数据时提供了方便。注意，每个数据字节的地址在内存中是固定的，不过寄存器中的字节排列在不同端序下是不同的。

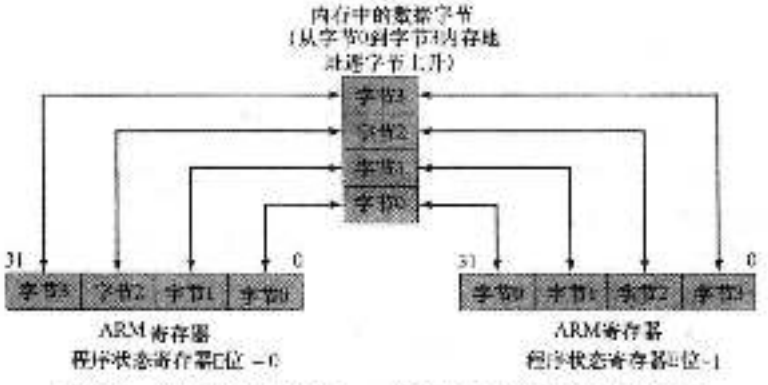


图 10-5 ARM 的端序支持——不同 E 位值时的字装载和保存

10.4 操作类型

不同的机器，其操作码的数目变动是很大的。然而，在所有机器上都会发现相同的常用操作类型。操作的典型分类包括：数据传送、算术、逻辑、转换（Conversion）、输入/输出、系统控制、控制转移。

表 10-3（以 [HAYE98] 为基础）列出了每一类操作的常见指令类型。本节对这些各种类型的操作提供一个简短的综述，并结合处理器执行具体操作类型时所采取的动作予以简要讨论（总结见表 10-4）。后一主题还要在第 12 章进行更详细的考察。

表 10-3 常用指令集操作

类型	操作名	说明
数据传送	Move（transfer）	自源向目标传送字或块
	Store	自处理器向存储器传送字
	Load（fetch）	自存储器向处理器传送字
	Exchange	源和目标交换内容

（续）

类型	操作名	说　明
数据传送	Clear(reset)	传送全 0 字到目标
	Set	传送全 1 字到目标
	Push	申请向栈顶传送字
	Pop	由栈顶向目标传送字
算术运算	Add	计算两个操作数的和
	Subtract	计算两个操作数的差
	Multiply	计算两个操作数的积
	Divide	计算两个操作数的商
	Absolute	以其绝对值替代操作数
	Negate	改变操作数的符号
	Increment	操作数加 1
	Decrement	操作数减 1
逻辑运算	AND	执行逻辑与操作
	OR	执行逻辑或操作
	NOT(complement)	执行逻辑非操作
	Exclusive-OR	执行逻辑异或操作
	Test	测试确定的条件；根据结果设置标志
	Compare	对两个或多个操作数进行逻辑的或算术的比较；根据结果设置标志
	Set 控制变量	由于保护目的、中断管理、时间控制等原因进行设置控制的一类指令
	Shift	左（右）移位操作数，一端引入常数
	Rotate	循环左（右）移位操作数
控制转移	Jump(branch)	无条件转移：向 PC 装入指定地址
	Jump 条件	测试指定的条件，根据条件或将指定地址装入 PC 或什么也不做
	Jump 子程序	将当前程序控制信息放到一个已知位置；转移到指定地址
	Return	用已知位置内容替代 PC 和其他寄存器的内容
	Execute	由指定位置取操作数并作为指令执行；不修改 PC
	Skip	PC 加 1 以跳过下一指令
	Skip 条件	测试指定条件；基于条件或跳步或不跳步
	Halt	停止程序执行
	Wait(hold)	暂停程序执行；重复测试指定条件；当条件满足时恢复执行
	No operation	无操作完成；但程序执行继续
输入/输出（I/O）	Input(read)	由确定的 I/O 端口或设备传送数据到目标（如主存或处理器寄存器）
	Output(write)	由确定的源传送数据到 I/O 端口或设备
	Start I/O	向 I/O 处理器传送指令以初始化 I/O 操作
	Test I/O	由 I/O 系统向指定目标传送状态信息
转换	Translate	根据一个对应表转换某一段内存的值
	Convert	将字的内容由一种形式转换到另一种形式（如压缩的十进制转换到二进制）

表 10-4 各种操作类型的 CPU 动作

数据传送	从一个位置向另一个位置传送数据
	若涉及存储器： 确定存储器地址 进行虚地址到实地址的转换 检查 cache 发出存储器读写命令
算术运算	在此之前和（或）之后，可能有数据传送
	在 ALU 内完成运算
	设置条件代码和标志
逻辑运算	与算术运算相同
转换	类似于算术和逻辑，为完成转换可能涉及特殊逻辑
控制传送	修改程序计数器（PC），为了程序调用/返回，还需管理参数传送和链接
输入/输出(I/O)	向 I/O 模块发命令
	若是存储器映射式 I/O，确定存储器映射地址

10.4.1 数据传送

最基础的机器指令类型是数据传送指令。数据传送指令必须指明几件事情。第一，源和目标操作数的位置必须指明。每个位置可能是存储器、寄存器或栈顶。第二，必须指明将要传送数据的长度。第三，像所有带操作数的指令一样，必须为每个操作数指明寻址方式。这最后一点将在第 11 章讨论。

选取什么样的数据传送指令包括在指令集内，是设计人员必须进行权衡考虑的范例。例如，操作数的通常位置（存储器或寄存器）是以操作数或操作码的类型来指明。表 10-5 列出了最常用的 IBM S/390 的数据传送指令例子。注意，这里有用于表示将要传送数据长度（8、16、32 或 64 位）的不同版本。还有，对寄存器到寄存器，寄存器到存储器和存储器到寄存器的传送，采用的是不同的指令。与之对比，VAX 机是使用带有不同传送数据长度版本的 MOV 指令，不过它把指示操作数是在寄存器还是在存储器中的信息，作为操作数的一部分。VAX 的方法对于程序员来说是相对容易的，他只需要与较少的助记符打交道。然而，它没有 IBM S/370 方法那样紧凑，因为每个操作数的位置（寄存器还是存储器）必须在指令中分别指定。当下一章讨论指令格式时我们再返回到这个问题上来。

表 10-5 IBM S/390 数据传送操作例子

操作助记符	名字	传送的位数	说 明
L	Load	32	由存储器传送到寄存器
LH	Load Halfword	16	由存储器传送到寄存器
LR	Load	32	由寄存器传送到寄存器
LER	Load (Short)	32	浮点寄存器间的传送
LE	Load (Short)	32	由存储器传送到浮点寄存器
LDR	Load (Long)	64	浮点寄存器间的传送
LD	Load (Long)	64	由存储器传送到浮点寄存器
ST	Store	32	由寄存器传送到存储器
STH	Store Halfword	16	由寄存器传送到存储器
STC	Store Character	8	由寄存器传送到存储器
STE	Store (Short)	32	由浮点寄存器传送到存储器
STD	Store (Long)	64	由浮点寄存器传送到存储器

以处理器动作而言，数据传送操作也许是最简单的类型。若源和目标都是寄存器，则处理器只要使数据从一个寄存器传送到另一个即可；这是处理器内部操作。若一个或两个操作数是在存储器中，则处理器必须完成如下某些或全部动作：

(1) 根据寻址方式（第 11 章讨论），计算存储器地址。

(2) 若地址指的是虚拟存储器，则将虚地址转换成物理存储器地址。

(3) 确定所寻找的项是否在高速缓存（cache）中。

(4) 如果不是，向存储器模块发命令。

10.4.2　算术运算

大多数机器都提供了加、减、乘、除这样的基本算术指令。这些操作总是为有符号整数（定点数）提供，也经常为浮点数和压缩十进制数提供。

其他可能有的操作包括各种单操作数指令，例如：

- Absolute：取一个操作数的绝对值。
- Negate：取一操作数的负数。
- Increment：操作数加 1。
- Decrement：操作数减 1。

一条算术指令的执行会涉及数据传送操作，来为算术和逻辑单元（ALU）准备输入，并传送 ALU 的输出。图 3.5 说明了在算术运算中涉及的数据传送活动。当然，处理器的 ALU 部分还要完成所要求的运算。

10.4.3　逻辑运算

大多数机器也提供处理一个字或其他可寻址单元中的个别位的操作，这常被称为“位操纵”（bit twiddling）。它们的基础是布尔运算（见第 20 章）。

能在布尔或二进制数据上完成的某些基本逻辑操作列于表 10-6 中。NOT 类作取一位的反。与或、异或（XOR）是有两个操作数的最常见逻辑功能。EQUAL 是一个有用的二进制测试。

表 10-6　基本的逻辑操作

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P = Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

这些逻辑操作能被加到 n 位的逻辑数据单元上。于是，若两个寄存器含有如下数据

$$(R1) = 10100101$$
$$(R2) = 00001111$$

则

$$(R1)\,AND\,(R2) = 00000101$$

其中（X）表示位置 X 中的内容。从这个例子可以看到，AND 类作能用来屏蔽（mask）一个字，选出字中的某些位，而其他位为 0。作为另一例子，若两个寄存器含有：

$$(R1) = 10100101$$
$$(R2) = 11111111$$

则

$$(R1)\,XOR\,(R2) = 01011010$$

对于一个全1的字，则XOR操作会将另一字的各位取反（1的补码）。

除了按位的逻辑操作，大多数机器也提供了移位和旋转（rotating）功能。最基本的操作说明在图10-6中。**逻辑移位**（logical shift），一个字的各位左移或右移，一端移出的位丢失，另一端则是移入0。逻辑移位主要用于隔离字中的各位段。移入字中的那些0修走了不需要的信息，它们从另一端被移去。

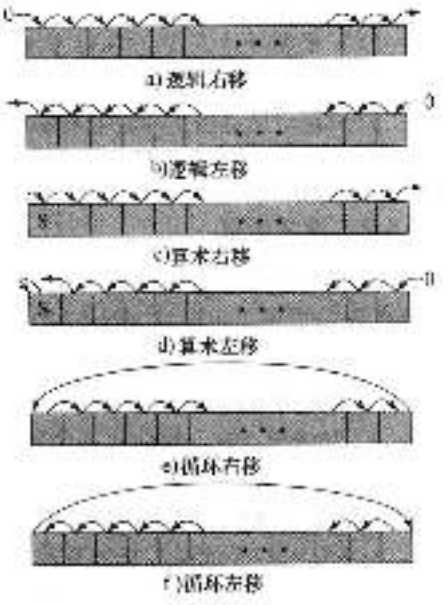


图10-6 移位和旋转操作

举个例子，假设我们希望每次1个字符地将数据字符发送到I/O设备。若每个存储器字是16位长含有两个字符，则在发送之前必须先拆包字符。送出字中的两个字符：

（1）将此字装入一个寄存器。

（2）右移8次，这将把剩下的字符移到寄存器的右半部。

（3）执行I/O操作。此时I/O模块将读取数据总线的低8位。

上述步骤发送了该字左半部的字符。要发送右半部的字符：

（1）再一次将此字装入寄存器。

（2）与（AND）0000000011111111。这将屏蔽掉左半部的字符。

（3）执行I/O操作。

算术移位（arithmetic shift）操作是将数据看作有符号整数而不移符号位。对于算术右移，符号位一般是复制到它右边空出的位。对于算术左移，除了符号位不变外，其余位的操作和逻辑左移一样。这些操作能加速某些算术运算。对于以2的补码表示的数，算术右移相当于除以2；原数为奇数的话，则被截断。无溢出时，算术左移和逻辑左移都相当于乘以2。如果出现溢出，算术左移和逻辑左移将产生不同的结果，但算术左移仍保留数的符号不变，因为有潜在的溢出问题，许多处理器，包括PowerPC和Itanium，不提供这种左移指令。其他，如IBM EAS/390，提供这种指令。奇怪的是，x86体系结构提供了一条算术左移指令，但将它定义成等同于逻辑左移指令。

旋转（Rotate）或**循环移位**（cyclic shift）操作保留了被操作数的所有位。旋转的一个可能用途是连续地将各个位移到最左位，从而可通过测试符号位（将其当作一个数）来识别各个位。

与算术操作一样，逻辑操作涉及ALU动作并且会涉及数据传送操作。表10-7给出了本小节讨论过的所有移位和旋转操作的例子。

表10-7 移位和旋转操作举例

输入	操作	结果	输入	操作	结果
10100110	逻辑右移（3位）	00010100	10100110	逻辑左移（3位）	00110000
10100110	算术右移（3位）	11110100	10100110	算术左移（3位）	10110000
10100110	循环右移（3位）	11010100	10100110	循环左移（3位）	00110101

10.4.4 转换

转换指令改变数据格式或对数据格式进行操作。一个转换例子是由十进制转到二进制。一个更复杂的编辑转换指令例子是EAS/390的翻译（Translate，TR）指令。这条指令能用来将一种8位编码转换成另一种形式编码，它有二个操作数：

TR R1 (L), R2

R2 包含一个 8 位编码表的起始地址。指令从 R1 指定地址开始，翻译连续的 L 个字节，每个字节被以此字节为索引的编码表项内容所替代。例如，由 EBCDIC 转换成 IRA。我们首先要在存储位置，例如 1000 – 10FF，生成一个 256 字节的表。表项的内容是字符的 IRA 编码，表项的排序是按 EBCDIC 编码的二进制表示值的大小，由小到大排列；即某字符的 IRA 编码放入表中的相对位置等于此字符的 EBCDIC 编码值。例如，数字 0 到 9 的 IRA 编码值为 30 到 39，其 EBCDIC 编码值为 F0 到 F9，故在 10F0 到 10F9 的表项位置中有值 30 到 39。现在，假定在 2100 位置处开始有数字 1984 的 EBCDIC 码。我们要求把它们转换成 IRA 编码。假定：

- 位置 2100 – 2103 处含有 F1 F9 F8 F4
- R1 中有 2100
- R2 中有 1000

若我们执行：

TR R1 (4), R2

位置 2100 – 2103 将会有 31 39 38 34。

10.4.5　输入/输出

第 7 章已经较详细地讨论了输入/输出（I/O）指令。正如我们看到的，输入/输出可选取几种方式，包括分立的编程控制的 I/O、存储映射的编程控制的 I/O、DMA 和使用 I/O 处理器。多数实现是只提供少数几条 I/O 指令，由参数、代码或命令字来指定所要求的动作。

10.4.6　系统控制

系统控制指令通常是特权指令，仅当处理器正处于某种特权状态时，或程序正在一个专门的特权存储区域中执行时才能执行它。一般而言，这些指令保留给操作系统使用。

系统控制操作的某些例子如下，一条系统控制指令可读取或更改控制寄存器的内容，我们将在第 12 章讨论控制寄存器。另一例子是，像在 EAS/390 存储系统中使用的，读或修改一个存储保护键的指令。再一个例子是多道程序系统中存取进程控制块。

10.4.7　控制转移

至此所讨论过的所有操作类型，指令指定的是将要完成的操作和操作数。隐含地，下一条将要执行的指令，是在存储器中当前指令之后的那条指令。然而，任何程序都有相当一部分指令具有改变指令执行顺序的功能。对于这些指令，CPU 所完成的操作是将程序计数器修改为某条指令的存储器地址。

有几个理由说明为何需要控制转移操作，其中最重要的是：

（1）在计算机的实际应用中，可以不止一次甚至上千次地重复执行某些指令。这种能力是至关重要的。实现一个应用需要上千条甚至上百万条指令。若每条指令必须分别写出，这将是不可思议的事情。若一个表或列表需要处理，则可使用程序循环的方法，一个指令序列重复执行直到所有数据被处理完毕。

（2）实际上，所有程序都涉及某种裁决。我们希望计算机能在满足某种条件下做某件事情，另一种条件满足时做另一件事情。例如，一个指令序列计算数的平方根。在序列开始时应该测试数的符号位，若是负数，则不进行计算而报告出错。

（3）正确地编写一个大型的即使是中等规模的计算机程序，也是一个极其困难的任务。若能将此任务分成小的片段，每次只工作在一个片段上，将是有益的。

现在，可以开始指令集中最常见的控制转移操作的讨论了，它们是：分支（branch）；跳步

(skip)；过程调用。

1. 分支指令

分支（Branch）指令亦称为跳转（Jump）指令，它把将要执行的下一条指令的地址作为它的一个操作数。使用最多的是条件分支（conditional branch）指令，即仅当某种条件被满足时才进行转移（将程序计数器的值更改为操作数指定的地址）；否则，顺序执行下一条指令（像通常那样程序计数器加1）。无条件分支（unconditional branch）指令则总是转移。

有两种常用的方式来生成条件分支指令中将要测试的条件。首先，大多数机器提供了一位或多位的条件码，它作为某种操作的结果被设置。这些条件码可以想象成一个用户可见的短寄存器。例如，算术运算（加法、减法等等）可能以4种值（0，正、负，溢出）来相应设置2位条件代码。在这样的机器上，可能有如下4种不同的条件转移指令：

BRP X　若结果是正，则转移到X位置

BRN X　若结果是负，则转移到X位置

BRZ X　若结果是零，则转移到X位置

BRO X　若结果出现上溢，则转移到X位置

在所有这些情况中，设置条件码的操作结果指的是最近一次操作的结果。

另一种方法是在同一条指令内完成比较和转移，这可用于三地址指令格式的指令。例如：

BRE R1，R2，X　若R1内容=R2内容，则转移到X

图10-7给出了这些操作的例子。注意，转移可以是向前（forward，具有更高地址的指令），也可以是向后（backward，较低地址）。例子中显示了无条件和条件分支指令能用来产生一个重复的指令循环。位置202到210的指令将重复执行，直到X减Y的结果是0。

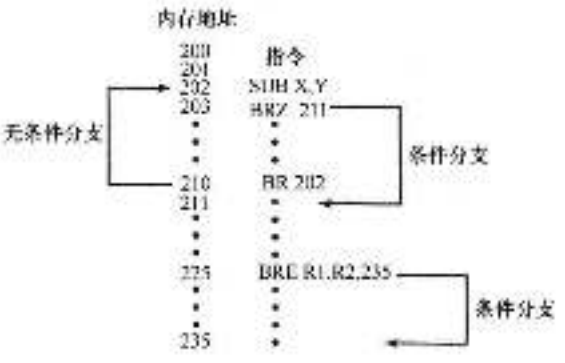


图10-7　分支指令

2. 跳步指令

另一类常用的控制转移指令是跳步（Skip）指令。跳步指令包括一个隐含地址。一般，跳步是指下一指令将被跳过；于是，隐含地址等于下一指令地址加上该指令长度之和。

因为跳步指令不要求目标地址字段，所以做其他事情更有自由。一个典型的例子是"加1并且若为0则跳步"（increment-and-skip-if-zero，ISZ）指令。考虑如下的程序片段：

```
301
 ⋮
309 ISZ R1
310 BR 301
311
```

在这个程序片段中，使用了两条控制转移指令来实现一个重复循环。初始时R1设置成欲重复次数的负数。在循环终端，R1被加1。若它不是0，则程序向后转移到循环开始处，循环体再次被执行。否则，此转移指令被跳过，接续执行循环之后的指令。

3. 过程调用指令

在编程语言的发展历程中，也许最重要的革新就是过程（Procedure）了。过程是一个自包含（self-contained）的计算机程序，能被并入到一个大的程序中。可在程序的任一点上激活或调用过程，即在那一点上，处理器被指示转到过程起始处，执行完整个过程，然后再返回到调用发生点。

使用过程的两个基本理由是经济性和模块化。过程允许多次使用同一代码片段，这能大大

节省编程工作量，并且也使系统的存储空间得到最有效的利用（程序必须被存储）。过程亦允许大的编程任务分解成较小的任务。这种模块化方式极大地减轻了编程任务。

过程机制涉及两类基本指令：由目前位置转移到过程的调用指令；由过程返回到调用发生位置的返回指令。两者都是转移指令形式。

图 10-8a 说明了如何使用过程来构造程序。此例中，有一个起始位置为 4000 的主程序。这个主程序包括了一条调用 Proc1 过程的指令，Proc1 的起始位置为 4500。当遇到这条调用指令时，CPU 挂起主程序的执行，并由位置 4500 处取出下一条指令，开始 Proc1 的执行。在 Proc1 内，有两次对位于 4800 开始的 Proc2 的调用，每次 Proc1 被挂起而 Proc2 被执行。返回（Return）指令使 CPU 返回到调用程序，并接续执行相应调用（Call）指令之后的指令。图 10-8b 显示了上述的执行顺序。

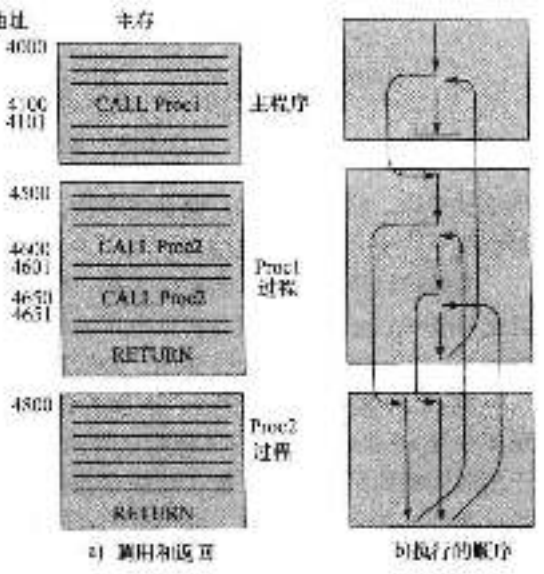


图 10-8　过程嵌套

有几点是值得注意的：

（1）可从多个位置调用过程。

（2）过程中能出现过程的调用，并允许过程嵌套（nesting）到任意深度。

（3）每一次过程调用与被调用程序中的一次返回相匹配。

因为能从不同位置点调用过程，所以处理器必须以某种方式保存返回地址，以使返回能相应发生。有三个常用的保存返回地址的位置：寄存器、被调过程开始处、栈顶部。

考虑一条机器语言指令 CALL X，它代表调用 X 处的过程。若使用寄存器方法，则 CALL X 指令引起如下动作：

$$RN \leftarrow PC + \Delta$$
$$PC \leftarrow X$$

这里的 RN 是用于此目的的寄存器。PC 是程序计数器，Δ 是指令长度。被调用的过程需要保护 RN 的内容，以便稍后的返回使用。

第二种可能的方法是将返回地址存于过程开始处。这种情况下，CALL X 引起：

$$X \leftarrow PC + \Delta$$
$$PC \leftarrow X + 1$$

这是很方便的，返回地址被很安全地保存。

前两种方法都能工作并已被采用。这些方法的唯一限制是使重入（reentrant）过程的使用变得复杂。重入过程是这样一种过程，它准许几个对它的调用同时存在。递归过程（自己调用自己的过程）就是使用这一特征的例子（见附录 H）。如果重入过程的参数通过寄存器或内存来传递，必须有代码负责保存参数，这样这些寄存器和内存空间就可以被其他过程调用使用了。

一个更通用更强有力的方法是使用栈（栈的定义见附录 10A）。当 CPU 执行一次调用时，它将返回地址放到栈上。当它执行一次返回时，它使用栈上的地址。图 10-9 说明了栈的使用。

过程调用除需要提供返回地址外，经常还需要传送参数。可以使用寄存器传送参数，也可以将参数存入恰好在 CALL 指令之后的存储器位置中；而返回必须是返回到这些参数之后的位置上。这两种方法都有缺点。若使用寄存器，则主调程序和被调程序都要小心编写，以使寄存器恰

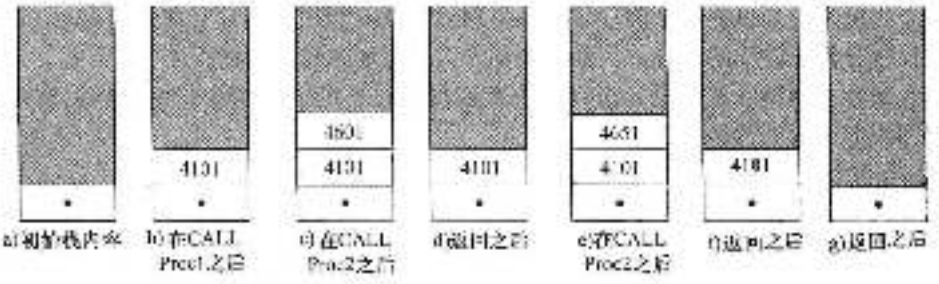


图 10-9 实现图 10-8 嵌套过程对栈的使用

当地被使用。将参数存于存储器的方法，会使传送可变数目的参数变得很困难。而且这两种方法都妨碍了可重入过程的使用。

参数传送更灵活的方法是使用栈。当处理器执行一次调用时，它不仅把返回地址压入栈，而且把将要传送给被调过程的参数也压入栈中。被调过程能由栈存取参数。返回时，返回参数也能放到栈中。为过程调用而存储的，包括返回地址和全部参数的栈内容称为栈帧（stack frame）。

图 10-10 提供了一个例子。此例有两个子过程 P 和 Q，主程序调用过程 P 时向其提供了两个局部变量 x1 和 x2，P 过程又调用 Q 并向 Q 过程提供了局部变量 y1 和 y2。在此图中，返回点是存于相应栈帧的第一项，接着存储的是指向先前栈帧起点的指针。如果压入栈的参数长度或数目是可变的，那么保存指向先前栈帧起点的指针是必要的。

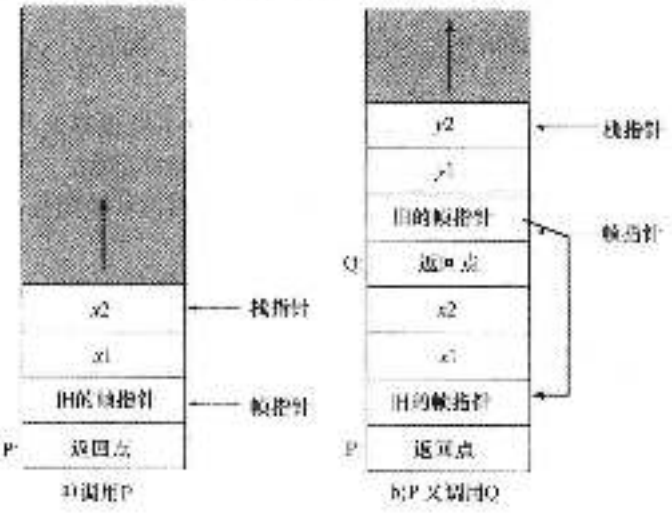


图 10-10 示例过程 P 和 Q 的栈帧增长

10.5 Intel x86 和 ARM 操作类型

10.5.1 x86 操作类型

x86 提供了一系列复杂的操作类型，包括几种特殊的指令，其目的是为程序员提供一种强有力的工具，以便将高级语言程序翻译成优化的机器语言程序。表 10-8 列出这些类型并予以简要说明。这些指令的大多数也是能在其他机器指令集中找到的常规指令，但有几种指令类型是为 x86 体系结构精心设计的，很值得考察。[CART06] 的附录 A 列出了各种 x86 指令，以及每条指令的操作数和指令的执行对条件码的影响。NASM 汇编语言手册的附录 B 提供了每条 x86 指令较详细的描述。这两个参考文档都在本书网站上可查到。

表 10-8　x86 操作类型（附有典型操作示例）

指　令		说　明
数据传送	MOV	在寄存器之间或寄存器与存储器之间传送类操作数
	PUSH	将操作数压入栈
	PUSHA	将所有寄存器的内容压入栈
	MOVSX	传送字节、字、双字；符号被扩展。字节传送到字或字节传送到双字，以 2 的补码符号扩展方式进行
	LEA	装入有效地址，将源操作数当做地址而不是它的值送到目标操作数
	XLAT	表查找翻译。以用户编制的翻译表的一字节替代 AL 中的字节，当 XLAT 执行时，AL 中应有表的无符号数的索引值。XLAT 以被索引项的内容替代 AL 的内容
	IN、OUT	由 I/O 空间输入，或输出到 I/O 空间
算术	ADD	加操作数
	SUB	减操作数
	MUL	无符号带乘法运算，操作数是字节、字或双字，结果是字、双字或四字
	IDIV	有符号数除法
逻辑	AND	按位与操作数
	BTS	位测试和置位，对位字段（域）操作数进行操作。指令将一位当前值复制到 CF 标志，并设置原来位为 1
	BSF	位向前扫描。扫描一个字或双字，当发现第一个 1 时，将度位对应的位号保存到一个寄存器
	SHL/SHR	左或右的逻辑移位
	SAL/SAR	左或右的算术移位
逻辑	ROL/ROR	左或右的循环移位
	SETcc	根据状态标志定义的 16 个条件的某一个，设置一字节为 0 或 1
控制传送	JMP	无条件转移
	CALL	转移控制到另一位置。在转移之前，此 CALL 指令之后的指令地址压入栈
	JE/JZ	若相等/为 0 则跳转
	LOOPE/LOOPZ	若相等/为 0 则循环。这是一个使用 ECX 值的条件跳转指令。指令先递减 ECX，然后为转移条件测试 ECX 值
	INT/INTO	中断/若上溢则中断。转移控制到一个中断服务子程序
字符串操作	MOVS	传送字节、字或双字的字符串。它对由 ESI 和 EDI 寄存的字符串元素进行操作，每次字符串操作之后，这两个寄存器自动被递增或递减以指向下一个字符串元素
	LODS	装入字节、字或双字的字符串
高级语言支持	ENTER	创建一个能用来实现块结构化（block-structured）高级语言规则的栈帧
	LEAVE	上述 ENTER 动作的逆动作
	BOUND	检查数组边界。验证第一个操作数是否在一个上下限范围之内，这上下限值存于紧第二个操作数所引出的两个相邻存储器中。若值不在边界内，产生中断。此指令用来检查数组索引是否越界
标志控制	STC	置位进位标志
	LAHF	将标志装入 A 寄存器。复制 SF、ZF、AF、PF 和 CF 位到 A 寄存器
段寄存器	LDS	装入指针到 DS 段寄存器和另一个寄存器
		系统控制
系统控制	HLT	暂停
	LOCK	对共享存储器提出加锁要求，这样在立即跟随 LOCK 之后的那条指令期间，处理器可独占式地使用共享存储器

（续）

指令		说明
系统控制	ESC	处理器扩展换码。此换码指示后面的指令，将在支持高精度整数和浮点数计算的数字协处理器上执行
	WAIT	等待，直到 BUSY#信号撤除。挂起处理器当前执行的程序，直到处理器已测出 BUSY 引脚信号无效，这表明数字协处理器已执行结束
保护	SGDT	保存全局描述符表
	LSL	装入段限。将段限装入一个用户指定的寄存器中
	VERR/VERW	验证段可被读/写
高速缓存管理	INVD	清空内部高速缓存
	WBINVD	在将已修改的行写回存储器之后，清空内部高速缓存
	INVLPG	使一个页表高速缓存（Translation Lookaside Buffer，TLB）项无效

1. 调用/返回指令

x86 为支持过程调用/返回提供了 4 条指令：CALL、ENTER、LEAVE、RETURN。考察这些指令所提供的支持是有指导意义的。回想图 10-10，实现过程调用/返回机制的普通方式是使用栈帧。当调用一个新过程时，在进入新过程之前必须完成如下步骤：

- 将返回点压入栈。
- 将当前帧指针（frame pointer）压入栈。
- 将栈指针拷贝到帧指针作为新的帧指针值。
- 调整栈指针以分配帧。

CALL 指令将当前指令指针值压入栈，并且通过将过程入口地址放入指令指针，引起控制转移到过程入口。在 8088 和 8086 机器中，典型的过程开始处有如下指令序列：

```
PUSH EBP
MOV  EBP,ESP
SUB  ESP,space-for-locals
```

这里的 EBP 是帧指针，ESP 是栈指针。在 80286 及其后的机器中，ENTER 指令以一条单独的指令完成上述全部操作。

ENTER 指令加入到指令集中为编译程序提供了直接的支持。这条指令也能支持像 Pascal、COBOL 和 Ada 语言中（C 和 FORTRAN 中未发现）的嵌套过程（nested procedure）。不过后来发现，对这些语言，有其他更好的方式来处理嵌套过程调用。另外，虽然 ENTER 指令（4 字节）与 PUSH、MOV、SUB 指令序列（6 字节）相比节省了存储器少数几个字节，但它实际要使用更长的时间来执行（10 个时钟周期与 6 个时钟周期相比）。于是，加入 ENTER 指令对指令集设计者来说，看起来是一个好想法，但它使处理器的实现变得复杂，而带来很少或没什么好处。我们将看到，与之对比，处理器设计的 RISC 方法会避免像 ENTER 这样的复杂指令，而以更简单指令序列来做到更有效的实现。

2. 存储管理

另一类指令专门用来与存储器分段打交道。这些是特权指令，仅能由操作系统来使用。它们允许局部或全局段表（叫做描述符表）被装入和读取，并可检查和更改段的优先级别。

与片内高速缓存打交道的专门指令已在第 4 章讨论过。

3. 状态标志和条件码

状态标志（status flag）是专门寄存器中的位，它可被特定操作所设置并用于条件转移指令。条件码（condition code）指的是一个或多个状态标志的设置。在 x86 和很多其他的体系结构中，状态标志由算术和比较操作设置。在大多数语言中，比较操作使两个操作数相减，与减法操作

样；但不同的是，比较操作只设置条件码，而减法操作还要将结果存入目标操作数中。

表 10-9 列出了用于 x86 的状态标志。每个状态标志或这些状态标志的组合可为条件转移所测试。表 10-10 列出了条件转移操作码所定义的条件码（状态标志值的组合）。

表 10-9　x86 状态标志

状态位	名字	说　明
C	进位	在算术运算之后，此位为 1 指出在最左位位置有向上的进位或借位。某些移位或循环移位操作也修改进位位
P	奇偶	一个算术或逻辑操作结果的奇偶性。1 指示偶，0 指示奇
A	辅助进位	8 位算术或逻辑运算后，它指示半字节之间的进位或借位，用于二进制编码的十进制数的算术中
Z	零	指示算术或逻辑运算结果是 0
S	符号	指示算术或逻辑运算结果的符号
O	溢出	指示 2 的补码加法或减法之后的算术溢出

表 10-10　x86 中条件转移和 SETcc 指令使用的条件码

符号	测试的条件	注　释
A, NBE	C = 0 AND Z = 0	高于；不低于或等于（大于，无符号）
AE, NB, NC	C = 0	高于或等于；不低于（大于或等于，无符号）；无进位
B, NAE, C	C = 1	低于；不高于或等于（小于，无符号）；进位置位
BE, NA	C = 1 OR Z = 1	低于或等于；不高于（小于或等于，无符号）
E, Z	Z = 1	等于；为零（无符号或有符号）
G, NLE	[(S = 1 AND O = 1)OR(S = 0 and O = 0)] AND [Z = 0]	大于，不小于或等于（有符号）
GE, NL	(S = 1 AND O = 1)OR(S = 0 AND O = 0)	大于或等于；不小于（有符号）
L, NGE	(S = 1 AND O = 0)OR(S = 0 AND O = 1)	小于；不大于或等于（有符号）
LE, NG	(S = 1 AND O = 0)OR(S = 0 AND O = 1)OR(Z = 1)	小于或等于，不大于（有符号）
NE, NZ	Z = 0	不等于，不为零（无符号或有符号）
NO	O = 0	无溢出
NS	S = 0	符号表示复位（不为负）
NP, PO	P = 0	奇偶标志复位（奇偶为奇）
O	O = 1	溢出
P	P = 1	奇偶标志置位（奇偶为偶）
S	S = 1	符号标志置位（为负）

对于此表有几个需要注意的地方。首先，我们可能想测试两个操作数，以确定一个是否比另一个更大些。但这取决于数是无符号还是有符号数。例如，8 位数 11111111 大于 00000000 是在把它们看成是无符号数时才成立（255 > 0），若把它们看成是补码数则前者小于后者（−1 < 0）。于是，多数汇编语言都是引入两组术语来区别这两种情况：若我们比较的是作为有符号整数的两个数，则使用小于（lose than）和大于（greater than）术语；若作为无符号整数来比较，则使用低于（below）和高于（above）术语。

第二个需关注的是比较有符号整数的复杂性。如果（1）符号位为 0 而且没有上溢（S = 0 AND O = 0），结果是大于或等于 0；或者（2）符号位是 1 但有上溢（S = 1 AND O = 1）。研究一

下图9-4应该会使你确信，为各种带符号操作所测试的条件是合适的。

4. x86 SIMD 指令

1996 年，Intel 开始将 MMX 技术引入 Pentium 产品系列。MMX 是一组用于多媒体任务的优化指令，共有 57 条新指令。这些新指令以一种 SIMD（single-instruction-multiple-data，单指令多数据）方式来处理数据。于是，它能一次在多个数据元素上同时完成加、乘这样的运算。一般，每条指令执行只用一个时钟周期。对于合适的应用，与不使用 MMX 指令相比，这些快速的并行操作能产生 2～8 倍的加速效果［ATKI96］。随着 x86 体系结构推出 64 位的处理器，Intel 也扩展了这些指令，使它们能处理双四字（128 位）操作数和浮点运算。在本小节中，我们将介绍 MMX 的特点。

MMX 主要是为多媒体程序设计而设置的。常规的指令是面向 32 位和 64 位数据操作的，而视频和音频数据一般是由 8 位或 16 位这样小的数据类型构成的大的阵列。例如，在图形或图像中，每一屏都是由像素（pixel）[⊖]点所组成，每个像素，或者每个像素的每个颜色分量（红、绿、蓝）都由 8 位数据表示。声音采样一般量化成 16 位数据。对于某些 3D 图形，基本数据类型为 32 位是普通的情况。为对这些长度的数据提供并行操作的方便，MMX 中定义了 3 种新数据类型。每种数据类型都是 64 位长，由多个小的定点整数字段所组成。这 3 种新类型是：

- 压缩字节型：8 个字节打包成一个 64 位长的数据。
- 压缩字型：4 个字打包成一个 64 位长的数据。
- 压缩双字型：2 个 32 位的双字打包成一个 64 位长的数据。

表 10-11 列出了 MMX 指令集，其中大多数指令涉及字节、字、双字上的并行操作。例如，PSLLW 指令完成压缩字型每个字的逻辑左移；PADDB 指令以两个压缩字节型数据为输入，对它们进行字节对字节加法，输出一个压缩字节型数据。

表 10-11 Pentium MMX 指令集

种类	指令	说明
算术	PADD[B,W,D]	并行完成［字节、字、双字］的环绕（wraparound）加法
	PADDS[B,W]	饱和带符号加法
	PADDUS[B,W]	饱和无符号加法
	PSUB[B,W,D]	环绕减法
	PSUBS[B,W]	饱和带符号减法
	PSUBUS[B,W]	饱和无符号减法
	PMULHW	并行完成16位有符号字的乘法，选取32位结果的高序16位
	PMULLW	并行完成16位有符号字的乘法，选取32位结果的低序16位
	PMADDWD	并行完成16位有符号字的乘法，两对相邻32位结果再相加比较
比较	PCMPEQ[B,W,D]	并行完成等于比较：以全"1"表示真，以全"0"表示假
	PCMPGT[B,W,D]	并行完成大于比较：以全"1"表示真，以全"0"表示假
转换	PACKUSWB	压缩字到字节（饱和无符号）
	PACKSS[WB,BW]	压缩字到字节或双字到字（饱和有符号）
	PUNPCKH[BW,WD,DQ]	从MMX寄存器解压缩（交叉）高序字节、字、双字
	PUNPCKL[BW,WD,DQ]	从MMX寄存器解压缩（交叉）低序字节、字、双字

⊖ 像素是能被指定一定灰度的数字图像的最小元素；它也等效于点阵图形表示中的像点。

（续）

种类	指令	说　明
逻辑	PAND	64 位按位与
	PNDN	64 位按位与非
	POR	64 位按位或
	PXOR	64 位按位异或
移位	PSLL[W,D,Q]	并行逻辑左移［字、双字、四字］，移位量由 MMX 寄存器的值或立即数指定
	PSRL[W,D,Q]	并行逻辑右移［字、双字、四字］，移位量由 MMX 寄存器的值或立即数指定
	PSRA[W,D]	并行算术右移［字、双字、四字］，移位量由 MMX 寄存器的值或立即数指定
数据传送	MOV［D，Q］	移入或移出 MMX 寄存器［双字、四字］
状态管理	EMMS	清除 MMX 状态（清除 FP 寄存器的标记位）

注：若一条指令支持多种数据类型［字节（B）、字（W）、双字（D）、四字（Q）］，则数据类型由方括号中字母指出。

新指令的突出特性是对字节或 16 位字操作数，引入了**饱和算术**（saturation arithmetic）。在通常的无符号算术中，当运算出现上溢时（即最高有效位有向上进位），则此额外位被舍弃。这称为环绕（wraparound）运算，因为从效果上看，舍弃进位使两个数加法之和会小于被加的两个数。例如，考虑两个十六进位制数 F000h 和 3000h 相加，则和能表示成：

$$\begin{aligned} \text{F000h} &= 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} &= \underline{0011\ 0000\ 0000\ 0000} \\ & \quad 10010\ 0000\ 0000\ 0000 \ = \ 2000\text{h} \end{aligned}$$

若这些数表示图像像素的亮度（数值越大越亮），则两个亮点的组合反而产生一个暗点，显然这不是所预期的。在饱和算术中，如果加法导致上溢，减法导致下溢，那么结果分别被设置成可表示的最大值或最小值。对上面这个例子，使用饱和算术则有：

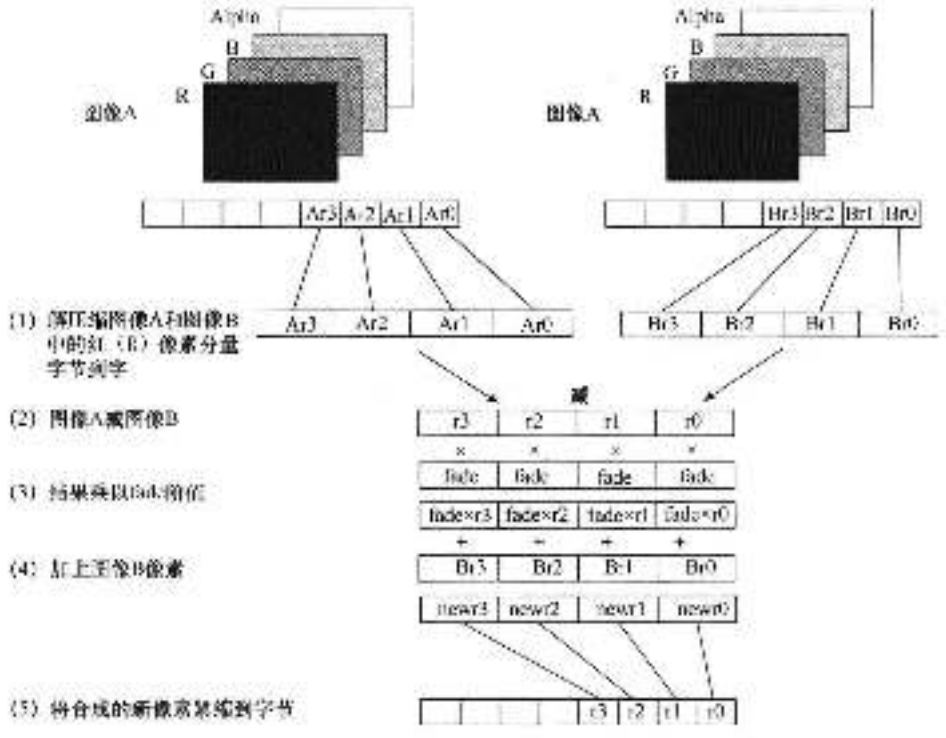
$$\begin{aligned} \text{F000h} &= 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} &= \underline{0011\ 0000\ 0000\ 0000} \\ & \quad 10010\ 0000\ 0000\ 0000 \\ & \quad 1111\ 1111\ 1111\ 1111 \ = \ \text{FFFFh} \end{aligned}$$

为感受一下使用 MMX 指令的好处，让我们考察一个选自［PELE97］的例子。一般视频应用程序常提供淡出、淡入效果，即一屏图像逐渐溶解成另一屏图像。两个图像以一种加权平均组合：

$$\text{Result\_pixel} = \text{A\_pixel} \times \text{fade} + \text{B\_pixel} \times (1 - \text{fade})$$

对 A、B 两图像的每个像素位置进行上述计算。当 fade 值由 1 逐渐变为 0（可用相应的 8 位整数分成 255 阶），则产生一系列的图像帧，实现了由 A 图像淡化到图像 B 的效果。

图 10-11 表示一组像素所需的步骤序列。8 位像素分量被转换成 16 位元素以适应 MMX 的 16 位乘法指令。若这些图像使用 640 × 480 分辨率，溶解技术使用全部的 255 个阶值，则使用 MMX 只需 5.35 亿条指令，而不使用 MMX 却要 14 亿条指令才能完成同样的计算［INTE98b］。



完成这个运算的MMX指令序列：

```
pxor      mm7, mm7      ;将mm7寄存器清除为全0
movq      mm3, fad_val  ;将4个同值的fade阶值装入mm3寄存器
movd      mm0, imageA   ;将图像A的4个红像素分量装入mm0寄存器
movd      mm1, imageB   ;将图像B的4个红像素分量装入mm1寄存器
punpcklbw mm0, mm7      ;将像素分量由字节解压缩到16位
punpcklbw mm1, mm7      ;将像素分量由字节解压缩到16位
psubw     mm0, mm1      ;图像A减图像B
pmulhw    mm0, mm3      ;差值乘以阶值
paddsw    mm0, mm1      ;和值加上图像B
packuswb  mm0, mm7      ;压缩到16位到字节
```

图 10-11 颜色平面表示法的图像合成[PELI97]

10.5.2 ARM 操作类型

ARM 提供了大量的操作类型。主要的类型如下所列：

- **装载和保存指令**：在 ARM 体系结构中，只有装载（Load）和保存（Store）指令能访问内存，算术和逻辑指令只对寄存器和指令中的立即数进行操作。这一限制是精简指令集计算机（RISC）的设计特征。在第 13 章我们将进一步探讨这点。ARM 体系结构支持两大类装载和保存指令，一种转载和保存单个寄存器数据到内存，另一种装载和保存一对寄存器的数据到内存：（1）装载或保存一个 32 位字或一个 8 位无符号字节；（2）装载或保存一个 16 位无符号半字，同时装载并符号扩展一个 16 位的半字或一个 8 位字节。
- **分支指令**：ARM 支持分支指令，允许条件分支指令向前或向后跳转最多 32MB。由于程序计数器使用了通用寄存器 R15，直接改写 R15 的值也可以产生分支或跳转。子程序调用是通过修改标准的分支指令来实现。因为可以向前或向后跳转 32MB 的距离，分支链接（Branch and Link，BL）指令把本指令的后继指令地址（返回地址）保存到 LR（R14）寄存器。分支的确定是通过指令中的 4 位条件码字段完成的。
- **数据处理指令**：这一类指令包括逻辑指令（AND、OR、XOR），加法和减法指令，以及

测试和比较指令。

- **乘法指令**：整数乘法指令对字或半字操作数进行计算，并产生普通或较长的结果。例如，以 32 位操作数为输入，产生 64 位结果的乘法指令。
- **并行加法和减法指令**：除了普通的数据处理和乘法指令外，还有一组并行加法和减法指令。其中指令两个操作数的对应部分同时进行运算。例如，ADD16 把两个寄存器的上半字相加，产生结果的上半字，同时把这两个寄存器的下半字相加，产生结果的下半字。这些指令类似于 x86 的 MMX 指令。对于图像处理应用很有用。
- **扩展指令**：这是用于解压缩数据的指令，它们通过符号扩展或填零扩展的方式，把字节扩展为半字或字，把半字扩展为字。
- **状态寄存器存取指令**：ARM 提供了读写状态寄存器中特定位的指令。

条件码（Condition Code）

ARM 体系结构定义了 4 个条件标志（condition flag），这些标志保存在程序状态寄存器中：N，Z，C 和 V（负标志，零标志，进位标志和溢出标志），其含义与 x86 中的 S，Z，C 和 V 标志一样。这 4 个标志的值构成了 ARM 处理器的条件码。表 10-12 显示了条件执行所依赖的条件组合。

ARM 中使用条件码有两个不同寻常之处：

（1）所有的指令，不仅仅是分支指令，都有条件码字段。这意味着所有的指令都可以根据条件决定是否执行。实际上，除了 1110 和 1111 这两个条件标志的组合以外，其他任何指令条件码字段的条件标志组合都意味着，该指令仅当条件满足时才能执行。

（2）所有的数据处理指令（算术、逻辑）都包含一个 S 位，指出该指令是否会修改条件标志。

按条件执行以及条件标志设置控制，有利于设计更短的程序，从而占用更少的内存。另一方面，由于所有指令都带 4 位的条件码字段，这意味着 32 位指令的操作码和操作数字段的可用位数就少了。不过 ARM 是一种 RISC 方式设计的处理器，更多地使用寄存器寻址，因此条件码的开销应该是可接受的。

表 10-12　ARM 指令条件执行的条件码

条件码	符号	被测试的条件标志	说明
0000	EQ	Z = 1	相等
0001	NE	Z = 0	不相等
0010	CS/HS	C = 1	进位设置/无符号大于或等于
0011	CC/LO	C = 0	进位清除/无符号小于
0100	MI	N = 1	-/负
0101	PL	N = 0	+/正或零
0110	VS	V = 1	溢出
0111	VC	V = 0	无溢出
1000	HI	C = 1 AND Z = 0	无符号大于
1001	LS	C = 0 OR Z = 1	无符号小于或等于
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	有符号大于或等于
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	有符号小于
1100	GT	(Z = 0) AND (N = V)	有符号大于

（续）

条件码	符号	被测试的条件标志	说明
1101	LE	(Z=1) OR (N≠V)	有符号小于或等于
1110	AL	—	总是（无条件）
1111			该指令以相反条件执行

10.6 推荐的读物

[BREY03] 全面介绍了 x86 指令集，ARM 指令集的介绍在 [SLOS04] 和 [KNAG04] 中。[INTE04b] 介绍了与微处理器程序结构有关的软件设计考虑，并讨论了开发程序无关代码的一些准则。

BREY09 Brey, B. *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4 and Core2 with 64-bit Extensions.* Upper Saddle River, NJ: Prentice Hall, 2009.

INTE04b Intel Corp. *Endianness White Paper.* November 15, 2004.

KNAG04 Knaggs, P., and Welsh, S. *ARM: Assembly Language Programming.* Bournemouth University, School of Design, Engineering, and Computing, August 31, 2004. www.freetechbooks.com/arm-assembly-language-programming-t729.html

SLOS04 Sloss, A.; Symes, D.; and Wright, C. *ARM System Developer's Guide.* San Francisco: Morgan Kaufmann, 2004.

10.7 关键词、思考题和习题

关键词

accumulator：累加器
address：地址
arithmetic shift：算术移位
bi-endian：双端
big endian：大端
branch：分支
conditional branch：条件分支
instruction set：指令集
jump：跳转
little endian：小端
logical shift：逻辑移位
machine instruction：机器指令
operand：操作数
operation：操作，运算
packed decimal：压缩的十进制数
pop：出栈
procedure call：过程调用
procedure return：过程返回
push：入栈
reentrant procedure：可重入过程
reverse Polish notation：逆波兰表示法
rotate：循环移位，旋转
skip：跳步
stack：栈

思考题

10.1 机器指令的典型元素是什么？
10.2 什么类型的位置能保存源和目的操作数？
10.3 若一指令包含 4 个地址，每个地址的用途是什么？
10.4 列出并简要介绍指令集设计的 5 个重要问题。
10.5 在机器指令集内，典型的操作数类型是什么？
10.6 压缩十进制表示法与 IRA 字符代码之间的关系是什么？
10.7 算术移位与逻辑移位有何区别？
10.8 为何需要控制转移指令？
10.9 列出并简要说明生成（将要条件分支指令测试的）条件的两种常见方式。
10.10 过程嵌套（nesting of procedure）是何意思？
10.11 列出为过程返回保存返回地址的三种可能位置。
10.12 什么是可重入过程？

10.13　汇编语言与机器语言有何不同？
10.14　什么是逆波兰表示法？
10.15　大端与小端有何不同？

习题

10.1　以十六进制数表示：
(a) 压缩十进制数 23
(b) ASCII 字符 25

10.2　对如下压缩十进制数给出其十进制值：
(a) 0111　0011　0000　1001
(b) 0101　1000　0010
(c) 0100　1010　0110

10.3　若给定的被处理器的字长为 n 字节，则如下表示法所能表示的最大和最小整数各是什么？
(a) 无符号　　(b) 符号-幅值
(c) 1 的补码　　(d) 2 的补码
(e) 无符号压缩十进制　　(f) 有符号压缩十进制

10.4　多数 CPU 都提供对压缩十进制数完成算术的逻辑电路。虽然十进制算术规则类似于二进制操作，但使用二进制逻辑时，十进制结果的个别数字会需要做某种修正。
考虑两个无符号数的十进制加法。若每个数有 n 个数字，则每个数有 4n 位。两个数相加使用一个二进制加法器。请提出修正此结果的简单规则，并对 1698 和 1786 这两个数完成加法运算。

10.5　十进制数 X 的 10 的补码定义成 $10^N - X$。这里的 N 是十进制数字的数目。请描述使用 10 的补码表示法完成十进制减法的过程，以 $(0736)_{10}$ 减 $(0326)_{10}$ 为例。

10.6　以零地址、一地址、二地址、三地址法分别编写程序来计算：

$$X = (A + B \times C)/(D - E \times F)$$

据此对 4 种地址机制进行比较。4 种地址机器可使用的指令是：

零地址	一地址	二地址	三地址
PUSH M	LOAD M	MOVE(X←Y)	MOVE(X←Y)
POP M	STORE M	ADD(X←X + Y)	ADD(X←Y + Z)
ADD	ADD M	SUB(X← X − Y)	SUB(X← Y − Z)
SUB	SUB M	MUL(X← X × Y)	MUL(X← Y × Z)
MUL	MUL M	DIV(X←X/Y)	DIV(X←Y/Z)
DIV	DIV M		

10.7　考虑一个只有两条指令的假想计算机。n 位指令的第 1 位用于指定操作码，其余 n − 1 位用于指定主存 2^{n-1} 个 n 位字的另一个。这两条指令是：
SUBS X 累加器减去位置 X 处的内容，结果存入累加器和位置 X 处。
JUMP X 将地址 X 放入程序计数器。
主存中的每个字，或是一条指令，或是一个 2 的补码表示的数。通过编程以实现如下操作，来证明这个指令清单是相当完整的。
(a) 数据传送：位置 X 到累加器，累加器到位置 X。
(b) 加法：将位置 X 的内容加到累加器。
(c) 条件转移。
(d) 逻辑 OR。
(e) I/O 操作。

10.8　多数指令集都有一条空操作（NOOP）指令。除了递增程序计数器之外，它对 CPU 状态没有任何影响。请给出这条指令的一些使用示例。

10.9　在 10.4 节已经看到：当无溢出时，算术左移和逻辑左移相当于乘以 2；出现溢出时，二者产生不同的结果，但算术左移必须保持符号位。试用两个 5 位 2 的补码整数来验证此语是对的。

10.10 算术右移时，数以何种方式舍入（例如，朝 $+\infty$ 舍入，朝 $-\infty$ 舍入，朝0舍入，远离0舍入）？

10.11 假设一个栈被处理器用来管理过程调用和返回。能否以栈顶作为程序计数器从而取消原程序计数器？

10.12 在 x86 体系结构中有一条称作“加后十进制调整”（decimal adjust after addition，DAA）指令。DAA 指令完成如下操作顺序：

```
if ((AL AND 0FH) >9) OR (AF = 1) then
    AL= AL + 6;
    AF= 1;
else
    AF=0;
endif;
if(AL>9FH)OR(CF = 1)then
    AL=AL +60H;
    CF= 1;
else
    CF=0;
endif.
```

上述程序中，“H”表示十六进制。AL是一个8位寄存器，它可容纳两个无符号8位整数的相加结果。AF是辅助进位标志。若加法的结果中有位3向位4的进位，它被置位。CF是进位标志，若有位7向位8的进位，它被置位。说明DAA指令所完成的功能。

10.13 x86 的比较指令 CMP 由目标操作数减源操作数；它改动状态标志（C、P、A、Z、S、O），但不改变两个操作数。CMP 指令可以用来确定目的操作数是否大于、等于或小于源操作数。

(a) 假设两个操作数被当作无符号整数。指出哪些状态标志与确定两个整数绝对值相对大小有关，以及大于、等于或小于分别对应于什么标志值。

(b) 假设两个操作数被当作2的补码表示的有符号整数。指出哪些状态标志与确定两个整数绝对值相对大小有关，以及大于、等于或小于分别对应于什么标志值。

(c) CMP 指令之后常是一条条件转移（Jcc）或条件设置（SETcc）指令，这里的 cc 指的是表 10-11 所列的 16 个条件之一。说明对有符号数比较所测试的条件是正确的。

10.14 若希望将 x86 的 CMP 指令用于 32 位浮点数操作，要得到正确的结果。下面提到的这些部分必须满足什么要求？

(a) 有效值、符号、阶值各字段的相对位置。

(b) 值0的表示。

(c) 阶值的表示。

(d) IEEE 754 格式满足这些要求吗？请说明。

10.15 大多数微处理器指令集都包括这样一条指令：它测试条件，并当条件为真时设置目标操作数。这样的例子有 x86 上的 SETcc，Motorola MC68000 上的 Scc 和 National NS32200 上的 Scond。

(a) 这些指令中有几点不同：

- SETcc 和 Scc 只对字节操作，而 Scond 可对字节、字、双字操作。
- SETcc 和 Scond 是当条件为真时设置操作数为整数1，为假时设置为0。Scc 则是为真时设置字节为全1，为假时设置为全0。

这些不同的相对优缺点是什么？

(b) 这些指令本身都不修改任何条件代码标志，因此确定这些指令所设置的值时需要显式测试指令结果，而不能借助条件代码。讨论，是否应该允许这种指令修改条件标志。

(c) 像 IF a > b THEN 这样的简单 IF 语句能使用一种数值表示法来实现，它使布尔表达式的值显式表示出来。这种方法与控制流（flow of control）方法不同，控制流方法中布尔表达式的值是通过程序控制流到达点的不同来体现的。一个编译器可用如下 x86 代码来实现 IF a > b THEN 语句：

```
        SUB   CX, CX  ; 置 CX 寄存器为 0
        MOV   AX, a   ; a 的内容传送到 AX 寄存器
        CMP   AX, b   ; AX 的内容与 b 的内容比较
        JLE   TEST    ; 若 a<=b 则转移
        INC   CX      ; 否则,CX 寄存器加 1
TEST    JCXZ  OUT     ; 若 CX 等于 0 则转移
THEN    ...
OUT
```

(A > B) 的结果作为一个布尔值，被保持到了 CX 寄存器中，因此在上面所示代码范围之外还可用。这种情况下使用 CX 保存布尔值非常方便的，因为多数转移和循环指令都有测试 CX 的功能。

请用 SETcc 指令来实现上述功能，以便节省存储空间和执行时间（提示：除 SETcc 之外，不需要再添加其他的 x86 指令）。

(d) 现在考虑高级语言语句：

$$A := (B > C)\ OR\ (D = F)$$

编译器可生成如下代码：

```
    MOV  EAX,B  ;传送位置 B 的内容到 EAX 寄存器
    CMP  EAX,C  ;比较寄存器 EAX 和位置 C 中的内容
    MOV  BL,0   ;0 表示假
    JLE  N1     ;若 B<=C 则转移
    MOV  BL,1   ;1 表示真
N1  MOV  EAX,D
    CMP  EAX,F
    MOV  DL,0
    JNZ  N2
    MOV  DL,1
N2  OR   BL,DL
```

请用 SETcc 指令来实现，从而节省存储器空间和执行时间。

10.16　假设两寄存器含有的十六进制值分别为 AB0890C2 和 4598EE50。使用 MMX 指令相加的结果是什么，假设不采用饱和算术：

(a) 作为压缩字节

(b) 作为压缩字

10.17　附录 10A 指出，若栈仅被处理器用于过程调用处理这样的目的，则指令集中可以没有面向栈的指令。没有这些面向栈的指令，处理器是如何做到对栈的使用的？

10.18　将如下算式由逆波兰（reverse Polish）表达式转换为中缀表达式（infix）：

(a) AB + C + D ×

(b) AB/CD/ +

(c) ABCDE + × ×/

(d) ABCDE − F/ + G − H/ × +

10.19　将下列算式由中缀表示法转换成逆波兰表示法：

(a) A + B − C + D + E

(b) (A + B) × (C + D) + E

(c) (A × B) + (C × D) + E

(d) (A − B) × (((C − D × E)/F)/G) × H

10.20　将表达式 A + B − C 使用 Dijkstra 算法转换成后缀表示法，显示所涉及的步骤。结果是等价于 (A + B) − C，还是 A + (B − C)？这要紧吗？

10.21　使用附录 10A 中定义的中缀法到后缀法（postfix）转换的算法，显示将图 10-15 中的计算公式转换成后缀法的各步，使用类似于图 10-17 的显示格式。

10.22　显示图 10-17 中的表达式计算，使用类似于图 10-16 的格式。

10.23　重画图 10-18 中的小端排列，按字节按大端的方式进行编号和显示，即把存储器表示为 64 位的行，字节排列是从左到右、从上到下。

10.24　对如下数据结构，请使用图 10-18 的格式画出小端的排列情况和大端的排列情况，并对其注解。

```
(a)struct {
      double i ; //0x1112131415161718
   } s_;
(b)struct {
      int i; //0x11121314
      int j; //0x15161718
   } s2;
```

```
(c)struct {
        short i; //0x1112
      short j; //0x1314
      short k; //0x1516
        short l; //0x1718
    } s2;
```

10.25 IBM Power 体系结构规范中不规定处理器应如何实现小端模式。它只指定当以小端模式操作时，处理器所看到的存储器形式。当数据结构由大端格式转换到小端格式时，处理器有选择的自由，它既可用真的字节交换机制来实现，也可用某种地址修改机制来实现。当前的 Power 处理器的默认方式全是大端机制，并使用地址修改法来对待小端的数据。

考虑图 10-18 中定义的结构 s，图的右下部表示的处理器所看到的结构 s。实际上，若结构 s 以小端模式被访问时，它在存储器中的排列情况如图 10-12 所示。解释它所涉及的映射，描述实现此映射的简易方式，并讨论这种方法的效率。



图 10-12 Power 体系结构内存中的小端数据结构 s

10.26 编写一个测试机器端序模式并报告结果的小程序，并上机运行。

10.27 MIPS 处理器能设置成以大端或小端的模式来操作。考察它的装入无符号字节（load byte unsigned，LBU）指令，它将存储器一字节装入寄存器的低位，寄存器的高 24 位填充 0。MIPS 参考手册使用一种寄存器传送语言来描述 LBU 指令：

```
mem ← load Memory(...)
byte ← Virtual Address<...>
if CONDITION then
    GPR[rt] ← 0^24 // mem<(...) * 8+byte.size-1 : (...) * 8+byte>
else
    GPR[rt] ← 0^24 // mem<(...) * 8+byte.size-1 : (...) * 8+byte>
endif
```

这里的 byte 指的是有效地址的低两位，mem 指的是取自存储器的装入值。手册中替代 CONDITION 的字是 BigEndian（大端），LittleEndian（小端）。你认为应使用哪个字？

10.28 大多数处理器，但不是全部，在字节中使用大端或小端的位序。与在多字节标量中大端或小端的字节排序是一致的。让我们考察 Motorola 68030，它使用大端的字节排序。然而关于格式的 68030 文档有些混乱。用户手册说明位域的位序与整数的位序相反。大多数的位域操作以一种端序来操作，但少数的位域操作又要求相反的端序。用户手册这样描述了大多数的位域操作：

位操作数由基地址和位号指定，基地址选择存储器一字节（基字节），位号选择此字节中的一位。最高有效位的位号是位 7。位域操作数由如下指定：(1) 选择存储器一字节的基地址。(2) 位域偏移（bit field offset），它指示位域最左位（基位）相对于基字节最高有效位的偏移。(3) 位域宽度，它指出基字节右起的多少位在此位域中。基字节最高有效位的位域偏移是 0，基字节最低有效位的位域偏移是 7。

这些指令使用的是大端位序还是小端位序？

附录 10A 栈

栈（stack）是个有序元素组，一次仅能存取它的一个元素。此存取点称为栈顶（top）。栈中元素的数目或说栈的长度（length），是可变的。栈中最后一个元素称为栈底（base）。栈的元素只能由栈顶添加或删除。因此，栈也常被看作是下压式列表（push down list 或后进先出（LIFO）列表）[1]。

图 10-13 表示了栈能完成的基本操作。我们以栈已含有某些元素的时刻开始。一个 PUSH（入栈）操作将一个新元素添加到栈顶。一个 POP（出栈）操作移去栈顶元素。在这两种情况下，栈顶都要相应进行移

[1] 更好的术语是顶上放列表（place-on-top of list），因为添加新元素时，栈中现有元素并不向下移动，而是新元素放入下一个可用的存储器地址中。

动。此外，要求两个操作数的二元操作（如加、减、乘、除）。使用顶部的两个元素作为操作数，POP 两个元素，然后再将结果 PUSH 到栈上。只要求一个操作数的一元操作（如逻辑 NOT）只使用栈顶元素。所有这些操作总结于表 10-13。

表 10-13　与栈有关的操作

PUSH	栈顶上添加一个新元素
POP	移走栈顶元素
一元操作	对栈顶元素完成操作，以结果替换栈顶元素
二元操作	对栈顶元素操作，删除栈顶两元素，操作结果放到栈顶

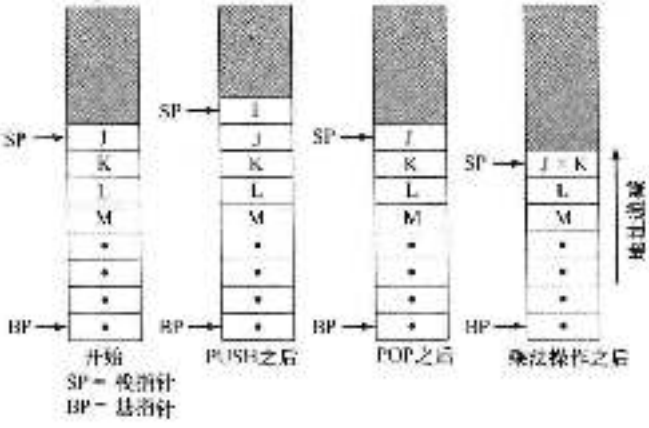


图 10-13　基本栈操作（栈向下增长）

10A.1　栈的实现

栈是一种有用的结构，作为 CPU 实现的部分来提供。在 10.4 节讨论过的一种用途是管理过程的调用和返回。栈对程序员亦是有用的，例如表达式求值，本节稍后将讨论。

栈的实现，部分取决于它的可能的用途。如果要求栈操作可被程序员使用，则指令集将包括与栈有关的操作，如 PUSH、POP 和使用一个或两个栈顶元素作为操作数的操作。因为所有这些操作指的都是唯一的位置，即栈顶。故操作数地址或操作数是隐含的，并不需要在指令中明确指出。这些是 10.1 节所指的零地址指令。

如果栈机制将只由 CPU 使用，用于过程管理这类目的，那么指令集中将没有显式的与栈有关指令。无论是哪种情况，栈的实现都要求有一组位置用来保存栈元素。一个典型的方法示于图10-14。主存（或虚存）中的连续位置块保留给栈。大多数时间，块只是部分地被栈元素所填充，剩余的部分可用于栈增长。

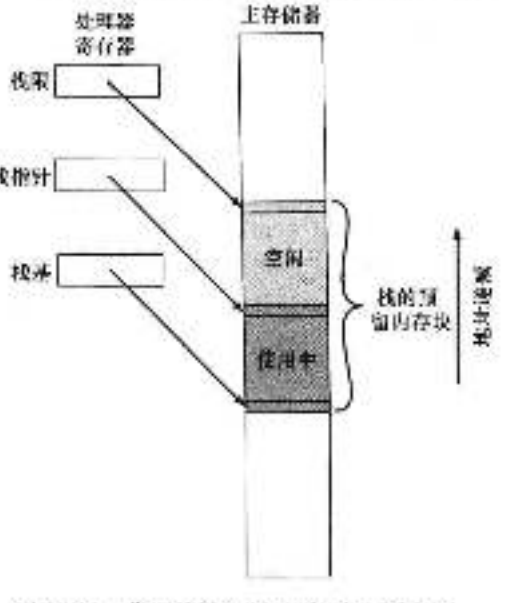


图 10-14　典型的栈组织（栈向下增长）

有三个与栈相关的地址需要恰当地操作，并且它们通常存于 CPU 寄存器中。

- **栈指针**（stack pointer）：保存着栈顶的地址。若一元素被添加到栈或自栈移走，则此指针相应地被递减或递增，以保存新栈顶的地址。
- **栈基址**（stack base）：保存为栈保留的内存块底部位置的地址。当栈是空时，试图进行 POP 操作将报告出错。
- **栈界限**（stack limit）：保存为栈保留的内存块另一端的地址。当内存块全被栈使用时，若试图进行 PUSH 操作则报告出错。

栈的实现有两个关键的属性：

- 向上增长/向下增长：一个向上增长（ascending）的栈向地址增大方向增长，从低地址开始并向高地址推进。也就是说，一个向上增长的栈，当有元素入栈时，其 SP 是递增的，而当元素出栈时，SP 是递减的。一个向下增长的栈（descending）向地址下降的方向增长，从高地址向低地址推进。大多数机器默认栈由高地址向低地址增长。
- 满/空：这是一个有误导性的术语，因为它不是指栈完全空或完全满。相反，SP 可以指向栈的顶部元素（满，full），或者栈的下一个空闲位置（空，empty）。对于满方式，当栈真正的完全填满了，SP 指向栈的上限。对于空方式，当栈全空时，SP 指向栈的基（栈底）。

图 10-13 是向下增长/空方式栈实现的例子（假设数值低的地址在图的上方）。ARM 体系结构允许系统程序员指定是使用向上增长还是向下增长的栈，以及空方式或满方式的栈操作。x86 体系结构采用向下增长/空方式的栈。

10A.2 表示式求值

数学表示式通常以一种称为**中缀**（infix）表示法来表达。按照这种形式，二元操作符出现在两个操作数之间（如 a + b）。对于复杂的表达式，要使用括号来确定表达式求值顺序。例如，a + (b × c) 将产生不同于 (a + b) × c 的结果。为减少括号的使用，操作具有一个隐含的优先顺序，通常是乘法先于加，于是 a + b × c 等同于 a + (b × c)。

另一种方法是称为逆波兰（reverse polish）或后缀（postfix）表示法。以这种形式，操作符是跟随在它的两个操作数之后。例如：

a - b	变成 ab -
a - (b × c)	变成 abc × -
(a + b) × c	变成 ab + c ×

注意，不管一个表示式是多么复杂，使用逆波兰表示法都不需要括号。

后缀表示法的优点是，采用这种形式的表达式很容易使用栈来完成求值计算。后缀表示法的算式是由左向右被扫描，对算式的每个元素值加如下的法则：

（1）若元素是一个变量或常数，把它压入栈。

（2）若元素是一个操作符，则弹出栈顶部两个元素，完成操作并将结果压入栈。

整个表达式被扫描之后，结果已在栈顶上。

这种算法的简明性使它求值表达式很方便。于是，多数编译器都是先将高级语言的表达式转换成后缀表示式，然后再产生相应机器指令。图 10-15 表示了使用与栈有关指令对 f = (a - b)/(c + d × e) 求值的机器指令序列。图10-15也表示了单地址和双地址指令的使用。注意，尽管后两种情况没使用与栈有关的规则，但后缀表示法仍用做生成机器指令的基础。栈程序的手件序列显示于图 10-16 中。

栈	通用寄存器	单个寄存器
Push a Push b Subtract Push c Push d Push e Multiply Add Divide Pop f	Load R1, a Subtract R1, b Load R2, d Multiply R2, e Add R2, c Divide R1, R2 Store R1, f	Load d Multiply e Add c Store f Load a Subtract b Divide f Store f
指令条数 10	7	5
内存访问 10 op + 6 d	7 op + 6 d	5 op + 5 d

图 10-15 计算 f = (a - b)/(c + d × e) 的三个程序比较

将中缀表示式转换成后缀表示式的过程自表，使用栈也是容易完成的。如下算法归功于 Dijkstra（参见 DIJK63）。算法从左到右扫描中缀表示式，随着扫描的进行，后缀表示式生成并输出。步骤如下：

（1）读者输入的下一个元素。

（2）若它是一个操作数，则输出它。

（3）若它是一个开括号（左括号），把它压入栈。

（4）若它是一个操作符，则有如下情况：

- 若栈顶是一个开括号，把操作符压入栈。
- 若它比栈顶操作符有更高优先权（乘和除优先于加和减），则把此操作符压入栈。
- 否则，从栈弹出操作符到输出，并重复步骤（4）。

（5）若它是一个闭括号（右括号），则弹出操作符到输出，直到遇上一个开括号。弹出并作废此开括号。

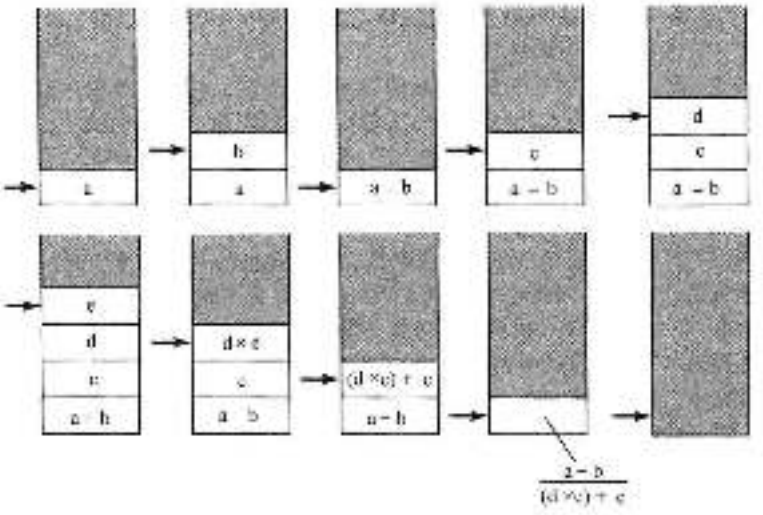


图 10-16　计算 f=(a−b)/[(d×e)+c] 时栈的使用

(6) 若还有输入，回到步骤 (1)。

(7) 若没有输入了，则弹出所有剩余操作符到输出。

图 10-17 说明了这种算法的使用。这个例子应能给读者一点对基于栈算法效力的感受。

输入	输出	栈（顶在左）
A + B × C + (D + E) × F	空	空
+ B × C + (D + E) × F	A	空
B × C + (D + E) × F	A	+
× C + (D + E) × F	A B	+
C + (D + E) × F	A B	× +
+ (D + E) × F	A B C	× +
(D + E) × F	A B C × +	+
D + E) × F	A B C × +	+ (
+ E) × F	A B C × + D	+ (
E) × F	A B C × + D	+ (+
) × F	A B C × + D E	+ (+
× F	A B C × + D E +	+
F	A B C × + D E +	× +
空	A B C × + D E + F	× +
空	A B C × + D E + F × +	空

图 10-17　由中缀表示法到后缀表示法的算式转换

附录 10B　小端、大端和双端

一些奇怪并令人气馁的现象，涉及字中的字节和字节中的位如何表示及引用的问题。我们首先查看字节排序问题，然后再考虑位排序。

10B.1　字节排序

端序（endianness）概念首先是在 Cohen 的著作中讨论的（参见 [COHE81]）。就字节而言，端序与多字节标量值的字节排序相关。用例子更能说明问题之所在。假定我们有 32 位的十六进制值 12345678，并且

它以一个32位字存于字节可寻址的存储器字节位置184处。此值由4个字节组成，最低有效字节的值是78，最高有效字节的值是12。存储此值有两种方式：

地址	值	地址	值
184	12	184	78
185	34	185	56
186	56	186	34
187	78	187	12

左边的映射方式是将最高有效字节存于最低数值的字节地址中，称为**大端序**（big-endian ordering），它等同于西方文化语言中的从左到右书写顺序。右边的映射方式是将最低有效字节存于最低数值的字节地址中，称为**小端序**（little-endian ordering），它使我们联想起算术单元中的从右到左的算术运算次序。[1]对于一个给定的多字节标量值，大端和小端映射方式的字节排列彼此正好相反。

端序概念的出现，是在必须将多个字节项看作是具有单一地址的单个数据项时，尽管它是由更小的可寻址单元组成。某些机器，像Intel 80x86、x86、VAX和Alpha，是小端的机器，而像IBM5370/390、Motorola的680x0、Sun SPARC和大多数RISC机器采用的是大端。当数据由一种端序类型的机器传送到另一类机器时，和当程序试图操纵多字节标量的个别字节或个别位时，就会出现问题。

端序特性不扩展到单个数据单元之外。在任何机器中，像文件、数据结构和阵列这类的集合都是由多个数据单元组成的，每个数据单元都有自己的端序。于是，将存储器数据块由一种风格的端序转换成另一种风格时，需要了解数据的结构。

图10-18说明端序如何确定寻址和字节次序。图中顶部的C结构含有几种数据类型。左下部的存储器排放情况来自大端机器编译这个结构的结果，右下部是小端机器的编译结果。无论是哪种情况，存储器显示为一系列64位的行。对于大端情况，存储器安排成从左到右、从上到下；而对于小端情况，存储器安排成从右到左、从上到下。注意，这些安排是任意的。无论使用哪种映射方式，在一行内可使用从左到右也可使用从右到左安排；这只是一种表述的形式，不是存储器指定的。实际上，查看某类机器的编程手册时，可发现令人困惑的各种表述，甚至在同一本手册中也是这样。

```
struct{
  int     a;    //0x1112_1314              word
  int     pad;  //
  double  b;    //0x2122_2324_2526_2728    doubleword
  char*   c;    //0x3132_3334              word
  char    d[7]; //'A','B','C','D','E','F','G'  byte array
  short   e;    //0x5152                   halfword
  int     f;    //0x6162_6364              word
};
```

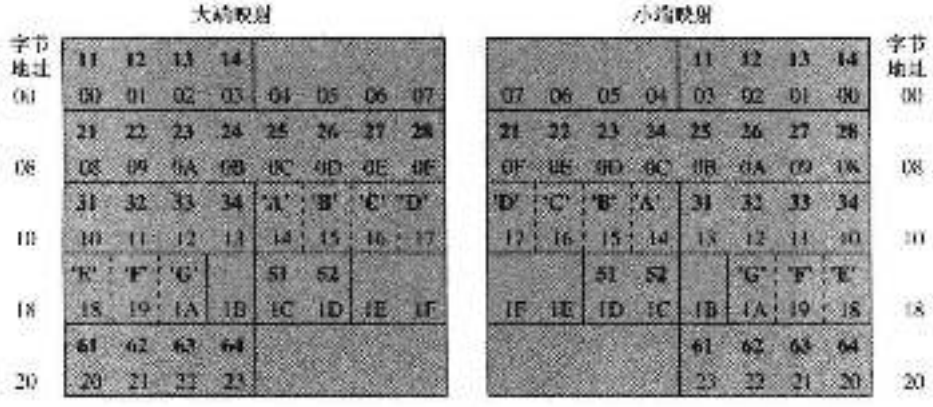


图10-18　C数据结构及其端序映射例子

我们能观察到有关这个数据结构的几点结论：

- 在两种策略中每个数据项有同样地址。例如，有十六进制值2122232425262728的双字的地址是08。
- 在任何一个给定的多字节标量值中，小端的字节排序是大端的反序，反之亦然。
- 端序不影响结构中数据项的次序。于是，4字符的字c展示出字节的反序，但7字符的字节数组d却不是。因此，d的各个元素的地址在两种结构中是相同的。

[1] 术语big-endian和little-endian来自Jonathan Swift的《Gulliver's Travels》第1部分的第4章，它们指的是两个组织间的一场宗教争论，一方说应在小头打破鸡蛋，另一方说应在大头打破鸡蛋。

当我们把存储器视作一个垂直的字节队列时，或许更能清楚地说明端序的效果，如图 10-19 所示。

哪种端序更优一点，没有普遍一致的意见[⊖]。如下观点偏爱大端风格：

- **字符串排序（sorting）**：在比较整数排列的字符串时，大端机器更容易些；整数 ALU 能并行比较多个字节。
- **十进制/IRA 打印输出**：所有值能从左到右打印而不会引起混乱。
- **一致的次序**：大端处理器以同样的次序存储它的整数和字符串（最高有效字节最先存储）。

以下观点偏爱小端风格：

- **整数地址转换**：一个大端处理器当需要转换 32 位整数地址到 16 位整数地址时，它必须完成加法，以便于使用最低有效字节。
- **算术**：采用小端风格完成高精度算术更容易些；你不必找到最低有效字节和反向移动。

两种风格难分高低，并且端序风格的选取经常更多考虑兼容以前的机器而不是其他因素。

PowerPC 是一个既支持大端模式又支持小端模式的双序机器。这种结构允许软件开发人员在将操作系统和应用由另一种机器迁入时，选取某种端模式。操作系统建立处理器执行时使用的端模式。一旦模式被选定，所有后续的装载和保存都由这种模式下的存储器寻址方式所确定。为支持硬件这种特点，机器状态寄存器（Machine State Register，MSR）中的对应两位，由操作系统视作进程状态的一部分来维护。一位用来指定内核运行时所采用的端模式；另一位用来指定处理器的当前操作的模式。于是，模式可依每个进程来改变。

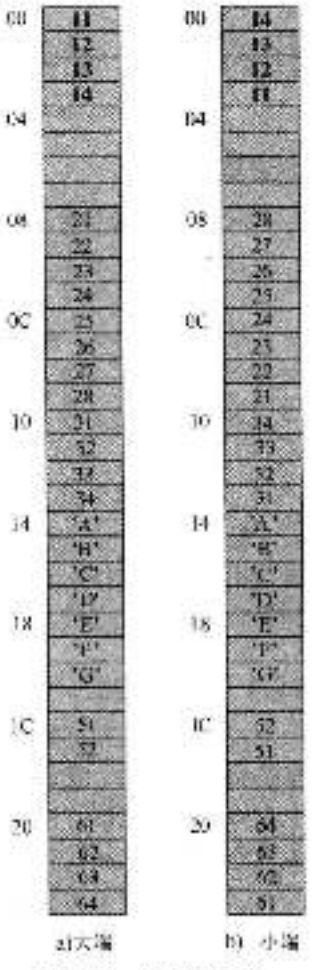


图 10-19　图 10-18 的另一种视图

10B.2　位排序

对于字节中的位排序，我们立刻面对两个问题：

（1）把第 1 位是计为位 0 还是计为位 1？

（2）把此最低位号是指派给字节的最低有效位（小端）还是指派给字节的最高有效位（大端）？

这些问题在所有机器中不是以同样方式回答的。实际上，在某些机器上，在不同的环境下也有不同的答案。而且，字节内大端或小端位次序的选取与多字节标量中的大端或小端字节次序的选取并不总是一致的。当操作个别位时，程序员必须关注这些问题。

需关注的另一领域是，在数据经由位串行线路发送情况下，当发送各个字节时，系统是先发送最高有效位还是最低有效位？设计者必须确保收到的位能被正确地处理。这个问题的讨论请参见［JAME90］。

⊖ 在《Gulliver's Travels》一书中，争论的两个组织都尊敬的旅行家说："所有真实信徒都将在他方便的一端打破他的鸡蛋。"这不是很有帮助吗？！

第11章

Computer Organization and Architecture: Designing for Performance, 8E

指令集：寻址方式和指令格式

本章要点

- 指令的操作数引用有两种形式：一是指令中含有操作数的实际值（立即数），二是指令中含有对操作数地址的引用。多种指令集使用类型广泛的寻址方式。这包括直接寻址（操作数地址在指令的地址字段中）、间接寻址（地址字段指向一个存储位置，此位置含有操作数地址）、寄存器寻址、寄存器间接寻址，以及各种形式的偏移寻址（寄存器值加上地址值产生操作数地址）。
- 指令格式定义了指令中字段的布局。指令格式设计是一项十分复杂的事情，要考虑到诸多因素，如指令长度是定长还是变长，指派给操作码和每个操作数字段的位数，以及如何确定寻址方式等。

第10章重点讨论了指令集做什么，特别是考察了机器指令可指定的操作和操作数类型。本章讨论如何指定指令的操作和操作数。这里要解决两个问题，首先是如何指定操作数地址，其次是指令的位如何组织，以定义操作数地址和操作。

11.1 寻址方式

正如以前提到过的，指令格式中的地址字段通常是相对较小的。我们希望，有能力大范围地访问主存或虚拟存储器。为实现此目标，指令采用了各类寻址技术。它们都涉及地址范围和寻址灵活性之间，以及存储器引用数和地址计算复杂性之间的权衡考虑。本节将考察最常用的寻址技术：立即寻址、直接寻址、间接寻址、寄存器寻址、寄存器间接寻址、偏移寻址、栈寻址。

图11-1显示了这些方式。在本节中，我们将使用如下表示法：

A = 指令中地址字段的内容

R = 指向寄存器的指令地址字段内容

EA = 被访问位置的实际（有效）地址

(X) = 存储器位置X或寄存器X的内容

表11-1列出了每种寻址方式所进行的地址计算。

在讨论之前需要说明两点。首先，实际上所有计算机结构都提供不止一种寻址方式。这也就提出一个问题，处理器如何确认什么样的寻址方式正被一条具体指令所使用。这有几种方法。通常是不同的操作码使用不同的寻址方式。另外，指令格式中的一位或几位能用做方式字段（mode field），方式字段的值确定使用哪种寻址方式。

第二点说明是关于有效地址（EA）的解释。

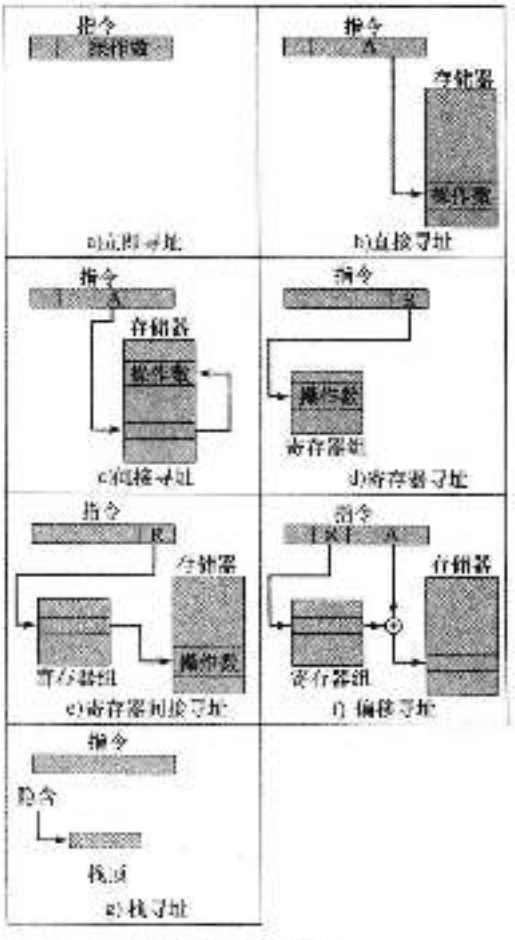


图11-1 寻址方式

在没有虚拟存储器的系统中，有效地址（effective address）将是主存地址或是寄存器。在虚拟存储器系统中，有效地址是虚拟地址或是寄存器。把虚拟地址映射到物理地址实际上是内存管理单元（MMU）的功能，并且是程序员不可见的。

表 11-1　基本寻址方式

方式	算法	主要优点	主要缺点
立即寻址	操作数 = A	无存储器访问	操作数的值有限
直接寻址	EA = A	简单	地址范围有限
间接寻址	EA = (A)	大的地址范围	多重存储器访问
寄存器寻址	EA = R	无存储器访问	地址范围有限
寄存器间接寻址	EA = (R)	大的地址范围	额外存储器访问
偏移寻址	EA = A + (R)	灵活	复杂
栈寻址	EA = 栈顶	无存储器访问	应用有限

11.1.1　立即寻址

寻址的最简单形式是立即寻址（immediate addressing）。在这种方式下，操作数实际出现在指令中：

$$\text{操作数} = A$$

这种方式能用于定义和使用常数或者设置变量的初始值。一般地，数以 2 的补码形式存储，最左位是符号。当操作数装入数据寄存器时，符号位向左扩展来填充数据字的字长。在某些情况下，立即二进制值被当作无符号非负整数。

立即寻址的优点是，除了取指令之外，获得操作数不要求另外的存储器访问，于是节省了一个存储器或高速缓存（cache）周期。其缺点是数的大小受限于地址字段的长度，而在大多数指令集中此字段长度与字长度相比是比较短的。

11.1.2　直接寻址

直接寻址（direct addressing）也是一种很简单的寻址形式，这种方式下地址字段含有操作数的有效地址：

$$EA = A$$

此技术在早期计算机中是很普遍的，但在当代计算机体系结构中就不多见了。它只要求一次存储器访问，而且不需要为生成地址的专门计算。正如前面所述，明显的不足是只能提供有限的地址空间。

11.1.3　间接寻址

直接寻址的问题是地址字段的长度通常小于字长度，这样寻址范围就很有限。一个解决方法是，让地址字段指示一个存储器字地址，而此地址处保存有操作数的全长度地址。这被称为间接寻址（indirect addressing）：

$$EA = (A)$$

正如前面所说明的，括号解释成“其内容”（content of）。这种方法的明显优点是，对于 N 位字长来说能有 2^N 个地址可用。缺点是为取一操作数，指令执行需要两次访问存储器，第一次为得到地址，第二次才是得到它的值。

虽然能被寻址的字的数目现在等于 2^N，但一次能被引用的不同有效地址数目限制到 2^K，这里的 K 是地址字段的位长度。一般而言，这并不是一个严重的限制，而且可能是有益的。在虚拟存储器环境中，所有的有效地址位置都能限定放到任何远程的第 0 页内。因为指令的地址字段

通常较小，自然会形成数目不多且数值不大的直接地址，这些地址对应的存储位置可以放在第0页中（唯一的限制是页的大小必须大于或等于2^5）。当一个进程激活时，将会有对第0页的重复访问，这将使它保留在物理存储器中。于是，一次间接存储器访问最多只涉及一次缺页而不是两次。

一种很少使用的间接寻址的变体是多级或级联的间接寻址：

$$EA = (\cdots(A)\cdots)$$

这种情况下，全字地址中有一个间接标志位（I）。若此位是0，则此全字地址中的其余位是有效地址EA。若I位是1，则要求另一级的间接。这种方法没什么特别的好处，缺点是为取一操作数要求三次甚至更多次的存储器访问。

11.1.4 寄存器寻址

寄存器寻址（register addressing）类似于直接寻址。唯一的不同是地址字段指的是寄存器而不是一个主存地址：

$$EA = R$$

举例说明一下，假设指令中寄存器地址字段的值是5，那么寄存器R5就是所指定的地址，操作数的值就是R5的内容。一般地，引用寄存器的地址字段有3到5位，因此能访问总计8～32个通用寄存器。

寄存器寻址的优点，一是指令中仅需要一个较小的地址字段，二是不需要存储器访问。正如第4章讨论过的，对CPU内部寄存器存取的时间是远小于主存存取时间的。寄存器寻址的缺点是地址空间十分有限。

若指令集中大量地使用了寄存器寻址，这意味着CPU寄存器将被大量使用。因为寄存器数量极其有限（与主存位置相比），所以只有它们能得到有效使用的应用才有意义。若是每个操作数都由主存来装入寄存器，操作一次后又送回主存，则暂存这些内容实际上是一种浪费。然而，若是留在寄存器中的操作数能为多个操作所使用，则实现了有效的节省，例如计算的中间结果。具体地，假设2的补码数的乘法是以软件实现的，那么，图9-12流程图中标记为A的位置是要多次访问的，因此应以寄存器而不是主存位置来实现。

哪些值应保留在寄存器中，哪些值应存于存储器，这个判决应由程序员或编译器完成。大多数当代CPU都使用多个通用寄存器，如何有效地使用它们就成为汇编语言编程人员（例如，编译器的编写者）的责任。

11.1.5 寄存器间接寻址

正如寄存器寻址类似于直接寻址一样，寄存器间接寻址（register indirect addressing）也类似于间接寻址。两种情况的唯一不同是，地址字段指的是存储位置还是寄存器。于是，对于寄存器间接寻址，

$$EA = (R)$$

寄存器间接寻址的优点和不足基本上与间接寻址类似。二者的地址空间限制（有限的地址范围）都通过将地址字段指向一个保存有全长地址的位置而被克服了。另外，寄存器间接寻址比间接寻址少一次存储器访问。

11.1.6 偏移寻址

一种强有力的寻址方式是直接寻址和寄存器间接寻址能力的结合。根据上下文的不同，它有几种名称，但基本的机制是相同的。我们将它称为偏移寻址（displacement addressing）：

$$EA = A + (R)$$

偏移寻址要求指令有两个地址字段，至少其中一个是显式的。保存在一个地址字段中的值

(值=A)直接被使用。另一个地址字段，或者一个基于操作码的隐含引用，指向一个寄存器。此寄存器的内容加上A产生有效地址。

我们将介绍最流行的二种偏移寻址：

- 相对寻址
- 基址寄存器寻址
- 变址

1. 相对寻址

对于相对寻址，隐含引用的寄存器是程序计数器（PC），因此也叫PC相对寻址。即当前PC的值（此指令后续的下一条指令的地址），加上地址字段的值（A），产生有效地址。一般地，地址字段的值在这种操作下被看成2的补码数的值。于是，有效地址是对当前指令地址的一个前后范围的偏移。

相对寻址利用了第4章和第8章讨论过的局部性概念。若大多数存储器访问都相对靠近正在执行的指令，则使用相对寻址可节省指令中的地址位数。

2. 基址寄存器寻址

对于这种寻址方式，其解释如下：被引用的寄存器含有一个存储器地址，地址字段含有一个相对于那个地址的偏移量（通常是无符号整数表示）。寄存器引用可以是显式的，也可以是隐式的。

基址寄存器寻址也利用了存储器访问的局部性。用它来实现第8章讨论过的段是一种方便的方式。在某些方案中，采用了一个单一的段基址寄存器，并且是隐含使用。而在其他情况中，程序员可选取一个寄存器来保存段的基地址，并且指令对它必须显式引用。在后一种情况下，若地址字段的位长度是K，并且可选取的寄存器有N个，则一条指令能访问$N\times 2^K$个字的块中的任一字。

3. 变址

对于变址，典型的解释如下：指令地址字段引用一个主存地址，被引用的寄存器含有对于那个地址的一个正的偏移量。注意，这种用法正好和基址寄存器寻址方式的解释相反。当然，两者的区别并不只是一个用法解释的问题。因为变址中的地址字段将被看作一个存储器地址，所以该地址字段通常要比基址寄存器指令中的地址字段包含更多的位。还有，我们将会看到对于变址将会有某些改进，而这些改进不能用于基址寄存器的方式。不过无论如何，基址寄存器寻址和变址二者的EA计算方法是相同的，而且两种情况下的寄存器引用都是有时是显式，有时是隐式的（对于不同类型的CPU）。

变址的一个重要用途是为重复操作的完成提供一种高效机制。例如，在位置A处开始有一数值列表，我们准备为表的每个元素加1。这需要取出表中每个数值，对它加1，然后再存回。需要的有效地址序列是A，A+1，A+2，…，直至表的最后一个位置。使用变址，这很容易完成。将值A存入指令的地址字段，并选取一个寄存器，叫做变址寄存器（index register），初始化为0。每次操作之后，变址寄存器加1。

因为变址寄存器普遍用于这种重复任务，故一般都需要在对它每次引用之后将它递增或递减。某些系统是自动完成这种递增、递减的操作，并将其作为同一指令周期的一部分。这称为自动变址（autoindexing）。若某个寄存器专门用于变址，则自动变址能隐含地自动启动。若使用通用寄存器作为变址寄存器，则自动变址操作需要用指令中的某一位来通知。采用递增的自动变址可描述成如下：

$$EA = A + (R)$$
$$(R) \leftarrow (R) + 1$$

某些机器既提供了间接寻址又提供了变址，在同一指令中使用两种寻址方式是准许的。这有两种可能的方式：在间接寻址之前或之后进行变址寻址。

如果在间接寻址之后进行变址，这称为后变址（post-indexing）：

$$EA = (A) + (R)$$

首先，地址字段的内容用来访问一个存储器位置取得直接地址。然后，这个地址被寄存器值变址。对于存取若干具有固定格式数据块中的某一个，这种技术是很有用的。例如，在第 8 章曾讨论过，操作系统需要为每个进程访问进程控制块。不管当前正在操作哪个块，完成的操作都是相同的。于是，访问这些块的指令中的那些地址（值 = A）可以都指向一个保存有一个可变指针的位置，此可变指针指向具体进程控制块的起点。变址寄存器中保存有块内偏移量。

预变址（pre-indexing）是变址完成在间接寻址之前：

$$EA = (A + (R))$$

像简单变址一样，完成一次地址计算，然而所计算出的地址含有的不是操作数而是操作数的地址。这种技术使用的一个例子是构建多路转去（multiway branch table）。在程序的某一点上，可能要根据一些条件转移到几个位置中的某一个。可以用位置 A 作为起点建立一个地址表。通过变址到这个表中，来找到所要求的位置。

正常情况下，指令集将不会同时包括前变址和后变址。

11.1.7 栈寻址

最后考虑栈寻址（stack addressing）方式。正如附录 9A 中所定义的，栈是一种位置的线性序列。它有时称为下推表（pushdown list）或后进先出队列（last-in-first-out queue）。栈是一个预留的位置块。数据项是被陆续加到栈顶，因此在任一给定时刻，栈对应的位置块是部分被填充的。与栈相关的是一个指针，它的值是栈顶地址。或者，当栈顶的两个元素已在 CPU 寄存器内，此时栈指针指向栈顶的第三个元素（参见图 10-14b）。栈指针保存在寄存器中，于是对存储器中栈位置的访问实际上是一种寄存器间接寻址方式。

栈寻址方式是一种隐含寻址形式。机器指令不需要指明存储器引用，而是隐含地指示操作发生在栈顶。

11.2 x86 和 ARM 寻址方式

11.2.1 x86 寻址方式

回顾图 8-21，x86 的地址转换机制产生一个地址，称为虚拟地址或有效地址，它是一个段内位移（offset）。段的起始地址和这个有效地址之和就构成了一个线性地址（linear address）。如果采用了分页，这个线性地址必须通过一个页转换机制来生成一个物理地址。下面暂不管这最后一步，因为它对于指令集和程序员都是透明的。

x86 配备了各种寻址方式，目的是使得高级语言能有效地执行。图 11-2 指出了所涉及的硬件逻辑。被访问的对象是段，它由段寄存器所确定。有 6 个段寄存器，具体的访问使用哪一个段寄存器，这取决于执行的上下文和指令。每个段寄存器保存一个指向段描述符表（segment descriptor table）的索引（见图 8-20），段描述符表保存了各个段的起始地址。每个段寄存器（用户可见的）与一个段描述符寄存器（程序员不可见的）关联，段描述符寄存器记录了此段的访问权限，以及段的起始地址和段的界限（段的长度）。此外，还有两个寄存器（基址寄存器和变址寄存器），可用于构造地址。

表 11-2 列出了 x86 的寻址方式，让我们依序考察每一种方式。

对于**立即方式**（immediate mode），操作数被包含在指令中。操作数可以是字节、字或双字的数据。

对于**寄存器操作数方式**（register operand mode），操作数位于寄存器中。对于像数据传送、算术和逻辑指令这样的一般指令，操作数可以是一个 32 位通用寄存器（EAX、EBX、ECX、EDX、ESI、EDI、ESP 和 EBP）、一个 16 位通用寄存器（AX、BX、CX、DX、SI、DI、SP 和

BP），或是一个 8 位通用寄存器（AH、BH、CH、DH、AL、BL、CL 和 DL）。还有访问段寄存器（CS、DS、ES、SS、FS 和 GS）的一些指令。

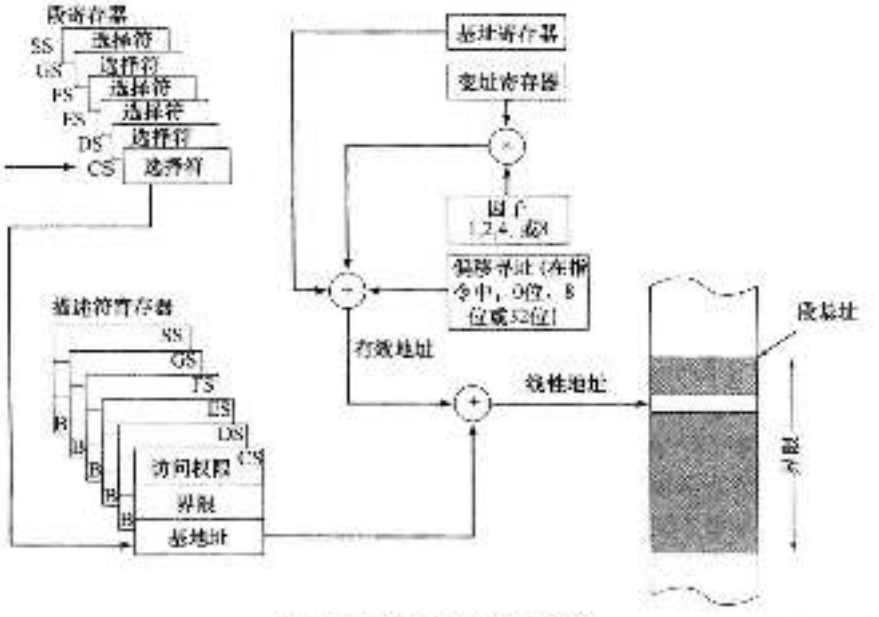


图 11-2　x86 寻址方式的计算

表 11-2　x86 寻址方式

方式	算法
立即寻址	操作数 = A
寄存器操作数寻址	LA = R
偏移量寻址	LA = (SR) + A
基址寻址	LA = (SR) + (B)
基址带偏移量寻址	LA = (SR) + (B) + A
比例变址带偏移量寻址	LA = (SR) + (I) × S + A
基址带变址和偏移量寻址	LA = (SR) + (B) + (I) + A
基址带比例变址和偏移寻址	LA = (SR) + (I) × S + (B) + A
相对寻址	LA = (PC) + A

注：LA = 线性地址
(X) = X 的内容
SR = 段寄存器
PC = 程序计数器
A = 指令中地址字段的内容
R = 寄存器
B = 基址寄存器
I = 变址寄存器
S = 比例因子

其余的寻址方式引用的是存储器中的位置，通过指定包含有此位置的段和距离段起点的位移来说明存储器位置。某些情况下，段是显式指定的。在其他情况下，段通过一个简单的段默认指派规则来隐式指定。

在偏移量方式（displacement mode）中，指令中的 8 位、16 位或 32 位偏移量就是操作数在段起点的位移。偏移量寻址方式只在少数几种机器中能找到，因为它会导致较长的指令。在 x86 机器中，偏

移量值可长达32位，这就使指令多达6字节长。偏移量寻址方式对于访问全局变量很有用。

剩下的寻址方式是指令的地址部分告诉处理器到何处去找地址，因此，它们是间接方式。**基址方式**（base mode）指定一个8位、16位或32位的寄存器，其中包含有效地址。这等同于我们已说过的寄存器间接寻址。

在基址带偏移量方式（base with displacement mode）中，指令包括一个将被加到基址寄存器的偏移量，基址寄存器可是任意一个通用寄存器。这种方式的使用例子有：

- 编译器用来指向一个局部变量域的开始。例如，基址寄存器能用来指向栈帧的起点，而栈帧含有相应过程的局部变量。
- 当数组元素大小不是1、2、4或8字节，从而不能使用变址寄存器来索引时，可用这种方式来索引数组。此时，偏移量指向数组起点，基址寄存器保存指定数组元素距数组起点的位移。
- 用于访问记录中的字段，基址寄存器指向记录的起点，而偏移量是到此字段的位移。

在比例变址带偏移量方式（scaled index with displacement mode）中，指令包括一个将加到变址寄存器的偏移量。除ESP通常用于栈处理之外，其他任何通用寄存器都可以作为变址寄存器。计算有效地址时，变址寄存器的内容乘以1、2、4或8的比例因子，然后加上偏移量。对于索引一个数组，这种方式是很方便的。比例因子为2能用于一个16位整数数组，比例因子为4能用于32位整数或浮点数。最后，比例因子为8能用于一个双精度浮点数的数组。

基址带变址和偏移量方式（base with index and displacement mode）是将基址寄存器内容、变址寄存器内容和偏移量三者求和，得到有效地址。同样，除ESP通常用于栈处理之外，其他任何通用寄存器都可以作为基址和变址寄存器。作为一个例子，这种寻址方式可以用于存取栈帧中的局部数组。除此之外，它也可以用于寻址二维数组，此时偏移量指向数组的起点，基址和变址寄存器分别处理二维数组中一个维的地址。

基址的比例变址带偏移量方式（based scaled index with displacement mode）是将变址寄存器内容乘以比例因子、基址寄存器内容和偏移量三者求和。若一个数组存于栈帧中，这种寻址方式是很有用的，此时数组元素可以是2、4或8字节长。这种方式亦能对数组元素是2、4或8字节长的二维数组提供有效的索引。

最后，**相对寻址**（relative addressing）能用于控制转移指令。一个偏移量加到指向下一条指令的程序计数器的值上。此时，偏移量被看作是一个有符号的字节、字或双字值，于是此值能增加或减少程序计数器中的地址。

11.2.2 ARM寻址方式

不像CISC机器，RISC机器一般都普遍采用简单和相对直截了当的一组寻址方式。不过ARM与这个传统有些不同，它有相对比较丰富的寻址方式。这些寻址方式基本是根据指令类型区分的。[1]

1. 装载/保存寻址

ARM中只有装载/保存（Load/Store）指令能访问内存。内存地址通常由一个基址寄存器加上一个偏移量来得到。考虑到变址的情况，则可分为3种不同方式（见图11-3）。

- **偏移**（offset）：对于这种寻址方式，变址不被使用。内存地址通过基址寄存器的值加上或减去偏移量而得到。例如，图11-3a显示了采用这种寻址方式的汇编语言指令 STRB r0, [r1, #12]。这条指令保存一个字节到内存中。在这个例子中，基址在寄存器r1中，偏移量是一个立即值，十进制数12。得到的地址（基址加上偏移量）就是r0的最低有效字节所要保存数据的位置。

---

[1] 如同讨论x86的寻址方式时一样，我们在接下来的讨论中也忽略从虚拟地址到物理地址的转换。

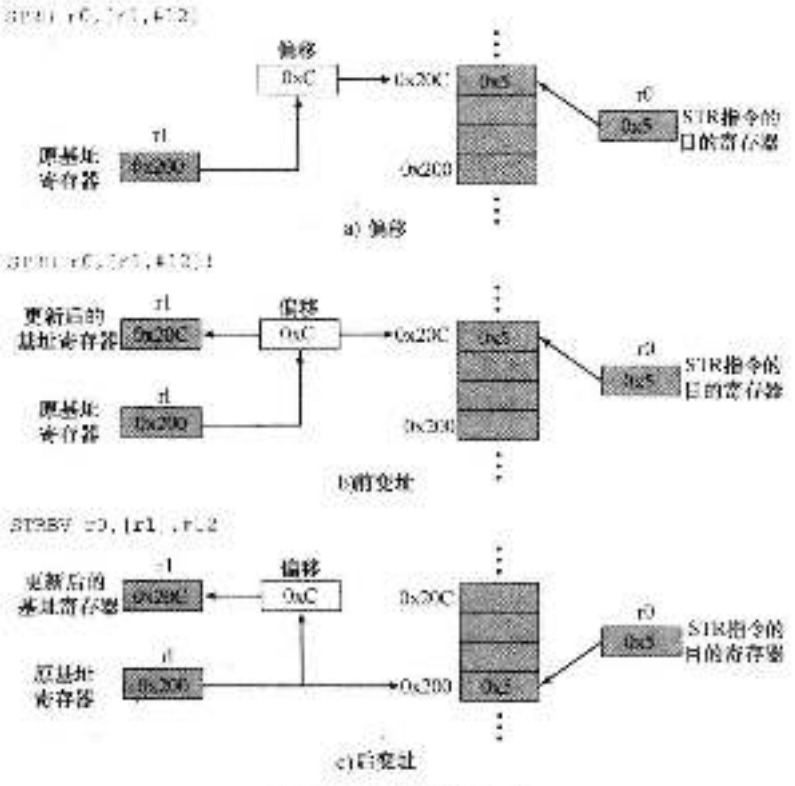


图 11-3　ARM 变址方式

- 前变址（preindex）：内存地址的计算与上面偏移寻址的方式一样。然后，计算得到的内存地址被写回到基址寄存器。也就是说，基址寄存器的值增加或减少了一个偏移量的值。图 11-3b 显示了采用这种寻址方式的汇编语言指令 `STR r0, [r1, #12]!`。其中感叹号“!”表示要进行前变址。
- 后变址（postindex）：内存地址就是基址寄存器的值。然后，偏移量被加到基址寄存器值中，或从中减去，结果再保存回基址寄存器。图 11-3c 显示了采用这种寻址方式的汇编语言指令 `STR r0, [r1], #12`。

可以注意到，ARM 中的基址寄存器在前变址和后变址寻址方式时，更像一个变址寄存器。偏移量可以是一个存于指令中的立即数，也可以是另一个寄存器的值。如果偏移量在寄存器中，那么就可以获得一个有用的特性：带比例的寄存器寻址。在偏移量寄存器中保存的值可以通过移位操作按比例放大或缩小。移位操作可以是逻辑左移，逻辑右移，算术右移，循环右移，以及扩展的循环右移（在循环移位中包括进位位）。移位的位数可以通过指令中的立即值给出。

2. 数据处理指令的寻址

数据处理指令使用寄存器寻址，或者寄存器和立即数混合寻址。采用寄存器寻址时，操作数寄存器中提供的值可以采用上文介绍的 5 种移位操作，来进行按比例的放大和缩小。

3. 分支指令

分支指令只有一种寻址方式，立即寻址。分支指令中带有一个 24 位的立即值。在地址计算时，这个立即值被左移 2 位，使得地址对齐到 32 位字的边界。这样，24 位的立即值可以产生 26 位的地址偏移量，相对于程序计数器来说，有效地址的范围为 ±32MB。

4. 多装/多存寻址

多装（Load Multiple）指令从内存装载多个数据到多个（可能全部）寄存器。多存（Store

Multiple）指令把多个（可能全部）寄存器的内容保存到内存中。用作装载或保存的寄存器列表由指令中一个 16 位的字段给出，其中每一位对应 16 个寄存器中的一个。多装和多存指令寻址方式会产生一组连续的内存地址。编号最小的寄存器对应最低的内存地址，编号最大的寄存器对应最高的内存地址。内存地址的产生有 4 种方式（见图 11-4）：后递增（increment after）、先递增（increment before）、后递减（decrement after）、先递减（decrement before）。指令用一个基址寄存器指定一个内存地址，寄存器的值从这个地址装载，或向这个地址存入，方向可以按字位置上升（递增），或下降（递减）。递增或递减可以发生在第一个值装载或保存完之后或之前。

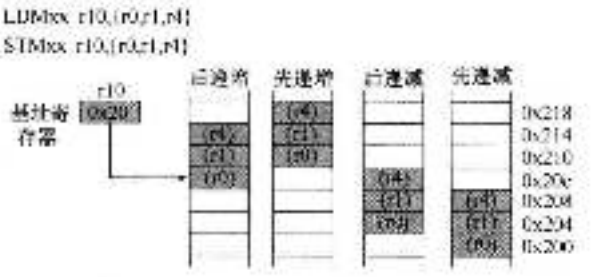


图 11-4 ARM 多装/多存寻址

这些指令对于数据块装载和保存、栈操作以及过程退出操作都是很有用的。

11.3 指令格式

指令格式通过它的各个构成部分来定义指令的位安排。一个指令格式必须包括一个操作码，以及隐式或显式的、零个或多个操作数。每个显式操作数使用 11.1 节描述的某种寻址方式来引用。指令格式必须显式或隐式地为每个操作数指定其寻址方式。大多数指令集使用不止一种指令格式。

指令格式设计是一种复杂的技术，已有的指令格式种类之繁多令人吃惊。本节考察关键的设计出发点，通过简要地查看某些设计来说明这些出发点。然后，再详细考察 x86 和 ARM 的做法。

11.3.1 指令长度

设计人员面对的最基本设计出发点，是指令格式的长度。这个决定与存储器尺寸、存储器组织、总线结构、CPU 复杂程度和 CPU 速度等相互影响。它决定了汇编语言编程人员所看到的机器指令的丰富性和灵活程度。

此时最明显的权衡考虑是在强有力的指令清单和必须节省空间之间进行。编程人员希望更多的操作码、更多的操作数、更多的寻址方式和更大的地址范围。更多的操作码和更多的操作数可使编程人员的日子更好过，他们能写出较短的程序来完成给定的任务。类似地，更多的寻址方式可给编程人员在实现某些像表处理和跳转这样的功能时有更大的灵活度。还有，随着主存容量的增加和虚拟存储器的使用，编程人员自然希望能寻址更大的范围。所有这些事情（操作码、操作数、寻址方式和地址范围）都需要更多指令的位，并使指令长度趋向更长的方向。但过长的指令长度将是浪费。一个 64 位指令占据 32 位指令的两倍空间，但很可能功能没有两倍那样多。

除了这个基本权衡之外，还有其他的考虑。指令长度或者应该等于存储器的传送长度（在总线系统中，是数据总线宽度），或者这两个值其中一个是另一个的整数倍。否则，会在取指周期得不到整齐数目的指令。一个相关考虑是存储器传送速度。这个速度的提高并不与处理器速度提高保持一致。于是，若处理器执行指令的速度快于取指令的速度，则存储器传送就变成一个瓶颈。这个问题的一种解决方法是使用 cache（参见 4.3 节）。另一种方法是使用较短的指令。这样，16 位指令能以 32 位指令的两倍速率来取指，但执行速度可能不会是两倍那样快。

一个看起来有些平凡的但依然重要的特征是，指令长度应当是字符长度（通常是 8 位）或定点数长度的整数倍。为表明这一特征，人们使用了一个不幸被含混定义的名词“字”（word）

[FRAI 83]。在某种意义上，字的长度是存储器组织的“自然”单位。字的长度通常也就确定了定点数的长度（一般二者是相等的）。字的长度一般也是等于，或至少整齐地相对于存储器传送的宽度。因为数据的普遍形式是字符数据，我们希望一个字能保存整数倍的字符。否则，当存储多个字符时每个字会有浪费的位，或者一个字符必须跨越字的边界。说明其重要性的一个例子是，当 IBM 推出 System/360 并打算使用 8 位字符时，它毅然做出转向的决定，由 700/7000 系列的 36 位的科学计算体系结构改为 32 位结构。

11.3.2　位的分配

我们已经讨论了影响指令格式长度确定的某些因素。一个同样困难的设计出发点是如何分配指令中的位，这个问题的权衡考虑是复杂的。

对于一个给定的指令长度，显然在操作码数目和寻址能力之间有一个权衡考虑的问题。越多的操作码明显地意味着操作码字段要有更多的位，这就减少了寻址可用的位数。对于这种折中考虑有一个有趣的改进，是使用变长的操作码。按照这种方法，有一个最小操作码长度，但是对于某些操作码，可通过使用指令附加位的方法来指定附加的操作。对于一个固定长度的指令来说，就使留给寻址使用的位减少了。于是，这种方法只适用于要求较少操作数和（或）不太强的寻址方式的指令。

下面一些相互关联的因素，在确定如何使用寻址位时是需要考虑的。

- **寻址方式的数目**：有时，一种寻址方式能隐含指定，例如，某些操作码会总是使用变址。其他情况下，寻址方式必须是显式指定的，这将需要一位或多位的寻址方式位。
- **操作数数目**：我们已看到过（例如，图 10-3），操作数少会使程序变长而且难于编写。当今的机器上的典型指令都提供两个操作数，每个操作数可能都要求有自己的寻址方式指示位，或者限制只允许两个操作数其中一个使用寻址方式指示位。
- **寄存器与存储器比较**：一个机器必须有寄存器，这样数据才能装入到 CPU 进行处理。对于只有一个单一的用户可见的寄存器（通常叫做累加器），一个操作数的地址是隐含的因而不占用指令位。然而，单一寄存器的编程很棘手，而且要求较多的指令。尽管是有多个寄存器，也只需要较少的位来指定寄存器。能用于操作数引用的寄存器越多，指令需要的位数越少。多项研究结果都指出，总计 8 到 32 个用户可见寄存器是比较合适的[LUND77，HEAK83]。大多数当代处理器体系结构至少有 32 个寄存器。
- **寄存器组的数目**：大多数当代机器只有一组通用寄存器，通常是 32 个或更多寄存器。这些寄存器既能用于保存数据，也能用于保存偏移寻址方式的地址。包括 x86 在内的一些体系结构具有两个或多个专用寄存器组（像数据和偏移量）。这种方法的优点之一是，对于同定数目的寄存器，功能上的分开将使指令只需较少的位数。例如，有两个 8 寄存器的组，只需 3 位来标识一个寄存器；操作码将隐式地确定那一组寄存器被引用。
- **地址范围**：对于引用内存位置的地址来说，地址范围与指令的地址位数有关。因为指令中地址位数严重受到限制，所以很少使用直接寻址。使用偏移寻址，范围问题就出现在地址寄存器长度上。尽管如此，允许对地址寄存器做相当大的偏移在应用中还是很方便的，这就要求指令中有比较多的地址位数。
- **地址粒度**：对于引用到存储器而不是寄存器的地址，另一考虑因素是寻址的粒度。在一个有 16 位或 32 位字的系统中，一个地址是能引用到一个字还是一个字节。具体由设计者选择。字节寻址对于字符处理是方便的，但对于一个固定大小的存储器来说却要求更多地址位。

于是，设计人员面对许多因素需要考虑和权衡。然而，各种选择的重要程度却不是很清楚。作为一个例子，我们援引一份研究报告[CRAG79]，它比较了各种指令格式的构成方法，包括

使用栈、通用寄存器、一个累加器和仅存储器到寄存器方法。在一致的一组假设条件下，观察到在代码空间或执行时间方面并没有显著的不同。

让我们简要地检查两类机器是如何权衡这些不同因素的。

1. PDP-8

对于通用计算机，最简单的指令设计实例之一是 PDP-8 [BELL78b]，它使用 12 位指令和 12 位的字，有一个单一的通用寄存器，即累加器。

尽管有这种设计限制，但它的寻址还是非常灵活的。每个存储器引用由 7 位加上两个 1 位的修饰符（modifier）组成。存储器分成固定长度的页，每页有 $2^7 = 128$ 字。地址计算是基于对页 0 或当前页（含有这条指令的页）的引用。两个修饰符其中一个是页位，它确定是引用到页 0 还是当前页。第 2 个修饰符指示是直接地址还是间接地址。这两种方式能组合使用，故一个间接地址可以是第 0 页或者当前页中的一个字所容纳的 12 位地址。另外，第 0 页上的 8 个专用字是自动变址的“寄存器”。当对这些位置上的某一个进行间接访问时，就相当于前变址（preindexing）。

图 11-5 表示 PDP-8 的指令格式。它有 3 位操作码和 3 类指令。对于操作码值 0～5，指令格式是一种单地址存储器引用指令，带有一个页位和一个间接位。于是，这种指令格式仅有 6 种基本操作。为扩大操作种类，操作码 7 定义了一种寄存器引用指令或称微指令（microinstruction）。以这种格式，其余位用于编码其他的操作。通常是，每位定义一种专门操作（例如，清除累加器），而且这些位能组合在一条指令中。采用微指令的设计思路可回溯到 DEC 的 PDP-1。在某种意义上，它是当今微程序式机器的先驱，这种机器将在第四部分讨论。操作码 6 是 I/O 指令；6 位用于选择 64 个设备中的某一个，3 位用于指定一个具体的 I/O 命令。

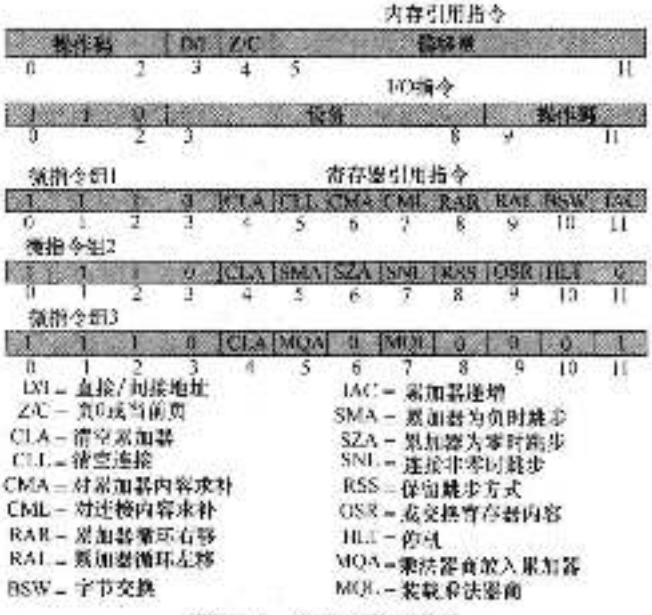


图 11-5 PDP-8 指令格式

PDP-8 指令格式是非常有效的，它支持间接寻址、偏移寻址和变址。包括对操作码扩展的操作在内，它能支持约 35 种指令。对于给定的 12 位指令长度的限制来说，设计人员已做得很不错了。

2. PDP-10

PDP-8 指令集的一个鲜明对照是 PDP-10 指令集。PDP-10 的设计目的是为大型分时系统所使用，并强调编程的易用性，尽管为此会付出更多的硬件代价。

设计该指令集时所采用的设计原则主要有［BELL78c］：

- **正交性**（orthogonality）：它是指两个变量相互独立的一种原理。在指令集语境中，此术语指的是指令中其他元素独立于操作码，即不被操作码所确定。PDP-10 设计者使用这个术语来描述这样一个事实：地址总是以独立的方式来计算，与操作码无关。与之对照的是，不少机器的寻址模式有时是隐含地取决于操作码。
- **完整性**（completeness）：每种算术数据类型（整数、浮点数、实数）都应有一组完整的和等效的操作。
- **直接寻址**（direct addressing）：提倡直接寻址方式，避免使用基址加偏移量寻址方式，因为这种方式把存储器组织的负担放到了程序员身上。

这些原则的每一个都是为了提高编程易用性这一主要目标。

PDP-10 有 36 位字长和 36 位指令长度，图 11-6 给出了固定的指令格式。操作码占据 9 位，允许多达 512 种操作，实际上 PDP-10 定义了总共 365 种指令。大多数指令有双地址，其中一个是 16 个通用寄存器中的某一个。于是，这个寄存器操作数的引用占据了 4 位。另一个操作数的引用包含一个 18 位的存储器地址字段。它既可用作立即数也可用作存储器地址。在后一种方案中，允许变址和间接寻址。变址寄存器使用的就是第一个操作数引用的同一通用寄存器。

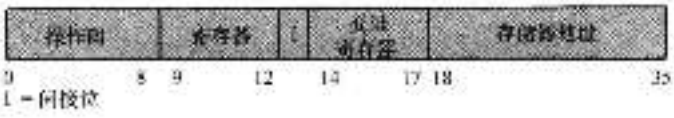


图 11-6　PDP-10 指令格式

36 位的指令长度确实过于奢侈。不管需要这么多操作码做什么，9 位操作码字段也显得太多。寻址是直截了当的。18 位地址字段使直接寻址更为合理。对于大于 2^{18} 的存储器容量，提供了间接寻址。为使编程容易，还为表处理和迭代程序提供了变址。另外，18 位的操作数字段，使立即寻址变得有吸引力。

PDP-10 指令集设计实现了前面所列的目标［LUND77］。它以空间利用率低为代价，使得程序员编程相对容易。这是设计人员有意而为之，不能把它错看成一种糟糕的设计。

11.3.3　变长指令

至此，我们已考察了单一固定指令长度的使用，并在上下文中间隐含地讨论了所做的权衡考虑。但设计者可选取另一种替代方法，即提供不同长度的各种指令格式。这种策略易于提供大的操作码清单，而操作码具有不同的长度。寻址方式也能更灵活，指令格式能将各种寄存器和存储器引用加上寻址方式予以组合。使用变长指令，能有效和紧凑地提供这些众多变化。

变长指令的主要代价是增加了 CPU 的复杂程度。硬件价格的降低，微程序设计方式的使用（第四部分将讨论）以及对 CPU 设计原则理解的普遍提高，这一切都使所付的代价变小。但是，我们会看到 RISC 和超标量机器能利用固定长度的指令来提高性能。

使用变长指令并没有消除所有指令相对于机器字长，其长度整齐的到来。因为 CPU 不知道下一条待取指令的长度，典型的策略是取至少等于最长指令长度的几个字节或几个字。这意味着有时一次能取来多条指令。不过，我们会在第 12 章看到，在任何情况下，这都是一个好的策略。

1. PDP-11

PDP-11 设计是在 16 位小型计算机范畴内提供了功能最强和最灵活的指令集［BELL70］。

PDP-11 使用了一组 8 个 16 位通用寄存器。其中两个有特殊的作用，一个用作栈指针，支持专用的栈操作。一个用作程序计数器，保存下一条指令的地址。

图 11-7 给出了 PDP-11 的指令格式。PDP-11 使用了 13 种不同格式，包括零地址、单地址和双地址指令类型。操作码的长度由 4 位到 16 位。寄存器引用使用 6 位，其中 3 位用于指定寄存

器，其余3位用于指定寻址方式。PDP-11具有丰富的寻址方式。将寻址方式关联到操作数而不是操作码，其优点是任何寻址方式能与任何操作码一起使用。正如前面提到的，这种独立性称为正交性。

PDP-11指令通常是1字（16位）长。对于某些指令，又续加了一个或两个存储器地址，这样32位和48位指令也成为指令清单的一部分。这进一步提供了寻址的灵活性。

PDP-11指令集和寻址能力是复杂的。这增加了硬件成本和编程的复杂性。优点是能开发更有效更紧凑的程序。

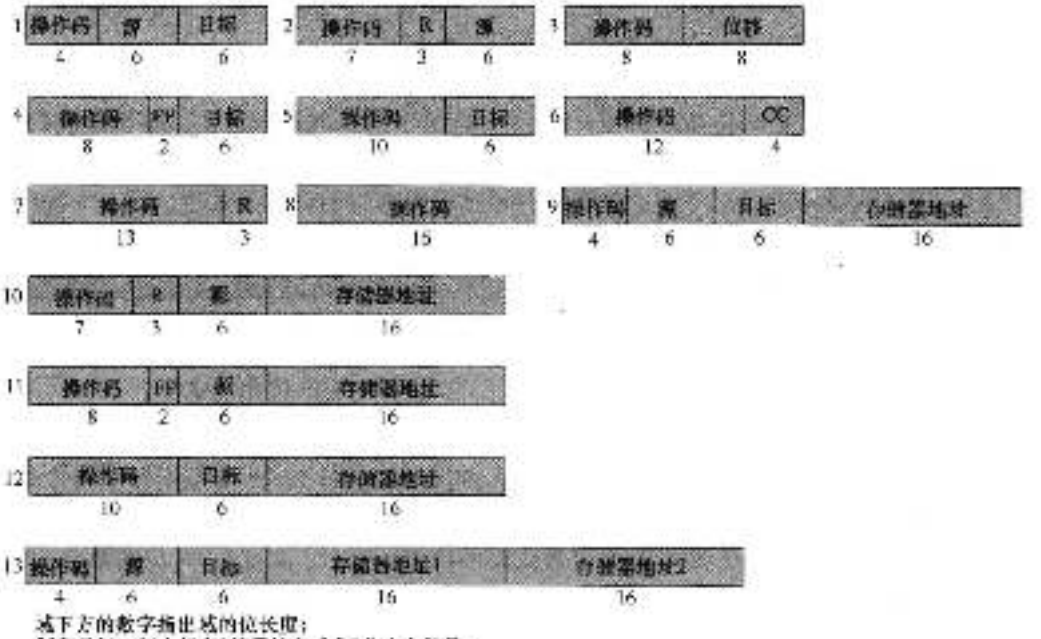


图11-7 PDP-11指令格式

2. VAX

大多数处理器体系结构提供相对比较少的固定指令格式。这对程序设计人员来说，导致了两个问题。首先，寻址方式和操作码是非正交的。例如，对某个操作，其操作数必须来自寄存器，另一个来自存储器，或者两个都来自寄存器等等。其次，一条指令只能使用有限数目的操作数，一般最多只有2～3个。由于某些运算本身就要求多个操作数，这就要使用某种技巧以两条或更多条指令来实现所要求的运算。

为避免这些问题，设计VAX指令格式时确定了两条规则［STRE78］：

（1）所有指令都应该具有“天生”数目的操作数。

（2）所有操作数都应该具有同样的规范通则。

结果是高度可变的指令格式。一条指令由1或2字节的操作码后跟随0～6个操作数规定符（specifier）组成，规定符的具体数目取决于操作码。最短的指令是1字节长，但也能有多达37字节的指令。图11-8给出VAX指令的若干例子。

VAX指令以1字节操作码开始。对于大多数指令，1字节操作码足够了，然而VAX指令有300多条。于是，少数指令要使用2字节操作码：第1字节为FD或FF，指示第2字节为实际操作码。

操作码之后是可多达6个的操作数规定符。最小操作数规定符是1字节，其格式最左4位用

于寻址方式说明。唯一例外是立即方式，此时第 1 字节最左两位是 00，留出 6 位用于指定立即数。由于这个例外，4 位字段总计能说明 12 种寻址方式。



图 11-8　VAX 指令举例

操作数规定符经常只由 1 个字节组成，此时最右 4 位用于指定 16 个通用寄存器之一。操作数规定符能以如下两种方式之一来扩展长度。第一种方式是，一个或多个字节的常数值可紧跟操作数规定符第 1 字节之后。例如，偏移寻址方式中的 8 位、16 位或 32 位的偏移量。第二种方式是用变址方式，此时第 1 字节由 0100 变址方式码和 4 位变址寄存器标识符组成，操作数规定符的其余字节用于指明基地址。

读者可能会感到惊讶，什么指令需要 6 个操作数，而 VAX 还确实有这样的指令。例如：

```
ADDP6 OP1, OP2, OP3, OP4, OP5, OP6
```

这条指令是将两个压缩十进制数相加，OP1 和 OP2 指明一个十进制串的长度和起始地址，OP3 和 OP4 指明另一个串，相加结果串的长度和存放始地址由 OP5 和 OP6 指示。

VAX 指令集有范围广泛的操作类型和寻址方式，这为程序设计人员，尤其是编写编译程序的程序员，提供了一种强有力的、灵活的编程工具。从理论上讲，这将有利于高效地把高级语言程序编译为机器语言，以及对 CPU 资源的有效利用。但为此付出的代价是，与具有简单指令集和格式的处理器相比，VAX CPU 的复杂性是大幅度地增加了。

第 13 章考察精简指令集的情况，届时我们将继续这个问题的讨论。

11.4　x86 和 ARM 指令格式

11.4.1　x86 指令格式

x86 配备了各种指令格式。下面介绍的指令各元素中，只有操作码字段是必出现的，其他都是可选的。图 11-9 说明了通常的指令格式。指令由 0 到 4 个字节的可选指令前缀、1 或 2 字节的操作码、一个由 Mod R/m 字节和比例变址（Scale Index）字节组成的可选地址指定符、一个可选

的偏移量以及一个可选的立即数字段等组成。

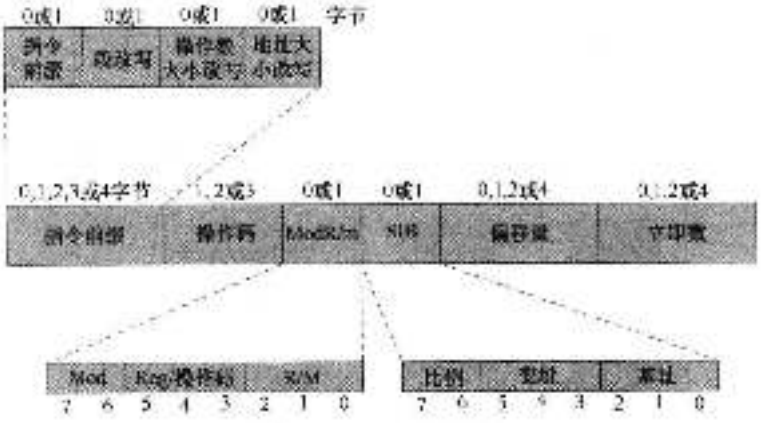


图 11-9 x86 指令格式

下面首先考察前缀字节。

- **指令前缀**（Instruction Prefix）：指令前缀若出现，则由 LOCK（锁定）前缀或一个重复前缀所组成。LOCK 前缀用于多处理器环境，保证对共享存储器的独占式访问。重复前缀指定串的重复操作，这就使 x86 处理串要比普通的软件循环快得多。有 5 种重复前缀：REP、REPE、REPZ、REPNE 和 REPNZ。当无条件的 REP 前缀出现时，指令中指定的操作对串中连续元素重复执行，重复的次数由 CX 寄存器指定。条件 REP 前缀出现时，引起指令重复执行直到 CX 变为 0 或指定的条件被满足。
- **段取代**（Segment Override）：显式地指定这条指令应使用哪个段寄存器，取代 x86 为那条指令规定的默认段寄存器。
- **操作数大小**（Operand Size）：指令默认的操作数大小是 16 位或 32 位，操作数大小前缀用于在 32 位和 16 位的操作数大小之间的切换。
- **地址大小**（Address Size）：处理器能使用 16 位或 32 位地址来寻址存储器。地址大小确定了指令格式中偏移量的大小和在有效地址计算中生成的位移量大小。其中一种被设计成默认值。地址大小前缀还用于 32 位和 16 位地址生成之间的切换。

指令本身包括如下字段。

- **操作码**：操作码字段长度是 1、2 或 3 字节。操作码可能包括一些位，这些位指定数据是字节还是全尺寸（16 位或 32 位，取决于上下文），数据操作方向（送至或来自存储器），以及一个立即数字段是否要进行符号扩展。
- **Mod R/m**：这个字节和下一个字节提供寻址信息。Mod R/m 字节指定操作数是在寄存器中还是在存储器中。若在存储器中，则该字节中的一个字段指定将使用的寻址方式。Mod R/m 字节由三个字段组成：Mod 字段（2 位），与 R/m 字段组合构成 32 个可能的值：8 个寄存器和 24 个变址方式。Reg/操作码字段（3 位）指定一个寄存器号或者用做操作码信息的 3 个补充位；R/m 字段（3 位）能指定一个寄存器作为一个操作数的位置，或者它构成寻址方式的一部分，与 Mod 字段组合来编码。
- **SIB**：Mod R/m 字节的某些编码要求包含另一个称为 SIB 字节来完成寻址方式的指定。SIB 字节由三个字段组成：比例（Scale）字段（2 位）指定用于比例变址的比例因子；变址（Index）字段（3 位）用于指定变址寄存器；基址（Base）字段（3 位）用于指定基址寄存器。
- **偏移量**：当地址方式指定符指出使用一个偏移量时，一个 8 位、16 位或 32 位有符号整数

的偏移量被添加到指令中。

- 立即数：在指令中提供一个 8 位、16 位或 32 位的操作数值。

在这里做几个比较可能是有益的。在 x86 格式中，寻址方式是作为操作码序列的一部分来提供的，而不是与每个操作数一起提供。因为只允许一个操作数有寻址方式信息，所以，x86 指令中也就只能引用到一个存储器操作数。相对比，VAX 机是每个操作数都可携带寻址方式信息，从而允许存储器到存储器的操作。因此，x86 的指令更紧凑。然而，若要求存储器到存储器的操作，VAX 使用一条指令就能实现。

x86 格式允许变址使用不仅是 1 字节，而且是 2 字节或 4 字节的位移。虽然使用较长的变址位移会导致指令更长，但这个特点能提供所需的灵活性。例如，在寻址大的数组或大的栈帧时它就很有用。与之对比，IBM S/370 指令格式只允许位移不大于 4KB（12 位的位移信息），并且位移必须是正值。当一位置不在此位移范围内时，编译器必须生成额外的代码来产生所需的地址。当与局部变量超过 4KB 的栈帧打交道时，这个问题变得尤为明显。正如［DEWA90］对它的描述："由于那个限制，为 370 生成代码非常费劲，导致有的 370 编译器简单地选择把栈帧的大小限定到 4KB。"

正如我们已看到的，x86 指令集的编码是很复杂的。其部分原因是与 8086 向下兼容的需要，部分原因是设计者打算为编译器设计者提供尽可能多的支持，以产生更有效的代码。然而，是像这样的复杂指令集还是另一极端的 RISC 指令集更合适一些，还是一个有争议的事情。

11.4.2　ARM 指令格式

ARM 的所有指令都是 32 位长，并在规整的格式（见图 11-10）。指令的前 4 位是条件码。正如在第 10 章中讨论的，实际上 ARM 的所有指令都是条件执行的。指令接下来的 3 位指定了指令的一般类型。对除分支指令之外的大多数指令而言，接下来的 5 位构成了操作码，和（或）操作的修订码。剩下的 20 位用于操作数寻址。ARM 指令这种规整的格式使得指令译码单元的工作变得比较轻松。



S = 对于数据处理指令，该位标示指令将更新条件码
S = 对于多载/多存指令，该位标示指令只在特权模式才允许执行
P, U, W = 用于区分不同寻址模式类型的位
B = 区分无符号字节访问（B=1），与字访问（B=0）的位
L = 对于装载/保存指令，区分装载（L=1）和保存（L=0）
L = 对于分支指令，确定一个返回地址是否保存在连接寄存器（link register）中

图 11-10　ARM 指令格式

1. 立即常数

为获得取值范围较大的立即数，采用立即数的数据处理指令不但指定了立即数值，还指定了一个循环移位值。8 位的立即数值被扩展到 32 位，然后循环右移若干次，次数等于 4 位循环移位值的两倍。图 11-11 显示了几个这方面的例子。



图 11-11 使用 ARM 立即常数的例子

2. 压缩指令集

压缩指令集（thumb instruction set）是 ARM 指令集中一个重新编码的子集。设计压缩指令集的目的是提高使用 16 位或更窄内存数据总线的 ARM 实现的性能，使其相对于普通 ARM 指令集来说有更高的代码密度。压缩指令集包含了 ARM 的 32 位指令集的子集，并重新编码为 16 位指令。下面列出了压缩指令集采取的精简措施：

（1）压缩指令都是无条件的，因此条件码字段都被省去。而且，所有的压缩算术和逻辑指令都更新条件标志，因此标志更新位也被省去。这样总计省去 5 位。

（2）压缩指令只包含了全部指令集中一部分操作，只用到 2 位的操作码字段，加上一个 3 位的类型字段。这样又省去 2 位。

（3）接下来通过对操作数字段的精简，又省去了 9 位，从而总计省去了 16 位。具体来说，例如，压缩指令只引用寄存器 r0 到 r7，因此只需要 3 位的寄存器引用字段，而不是 4 位。立即数字段中也省去了 4 位的循环移位量字段。

ARM 处理器可以执行压缩指令和普通 32 位 ARM 指令混合在一起的程序。处理器控制寄存器中的一位用于确定当前要运行的指令是哪种类型的指令。图 11-12 给出了这样的例子。图中给出两种类型指令的一般格式，以及 16 位和 32 位格式指令的具体示例。

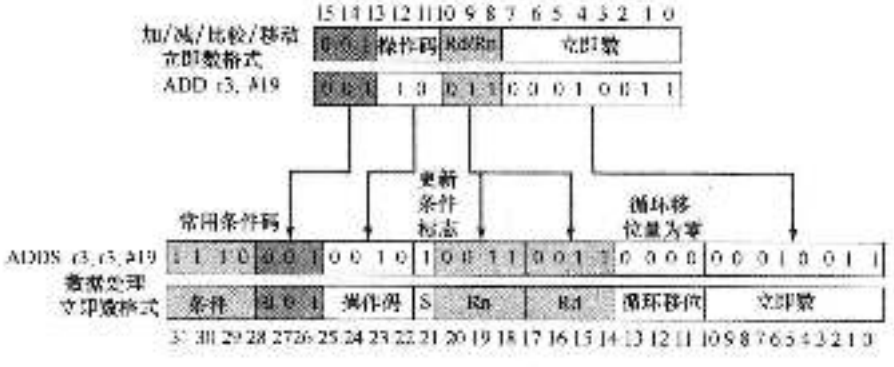


图 11-12 将一条压缩 ADD 指令扩展为对应的普通 ARM 指令

11.5 汇编语言

处理器可以识别和执行机器指令。这些指令都是以二进制形式存储在计算机中的。如果一个程序员想直接用机器语言编程，那么他必须用二进制来编写程序。

考虑下面这个简单的 BASIC 语句：

$$N = I + J + K$$

假设我们想用机器语言来编写这个语句，并且把 I、J 和 K 分别初始化为 2、3 和 4。图11-13a

给出了机器语言的程序。程序从内存位置 101（十六进制）开始。所用到的 4 个变量在内存中保留在从位置 201 开始的地方。程序包括下面这 4 条指令：

(1) 把位置 201 的内容装载到累加器 AC 中。

(2) 把位置 202 的内容加到累加器 AC 中。

(3) 把位置 203 的内容加到累加器 AC 中。

(4) 把累加器 AC 中的内容保存到位置 204 中。

这样编程很明显十分繁琐，而且容易出错。

一个稍微的改进是用十六进制，而不是二进制来编写程序（图 11-13b）。我们可以把程序写成一系列的行。每行对应着一个内存位置的地址，以及要保存到这个位置的二进制内容的十六进制表示。然后，我们需要一个程序接收这些行作为输入，把每行的十六进制数转换为二进制数，并把它们保存到所指定的位置。

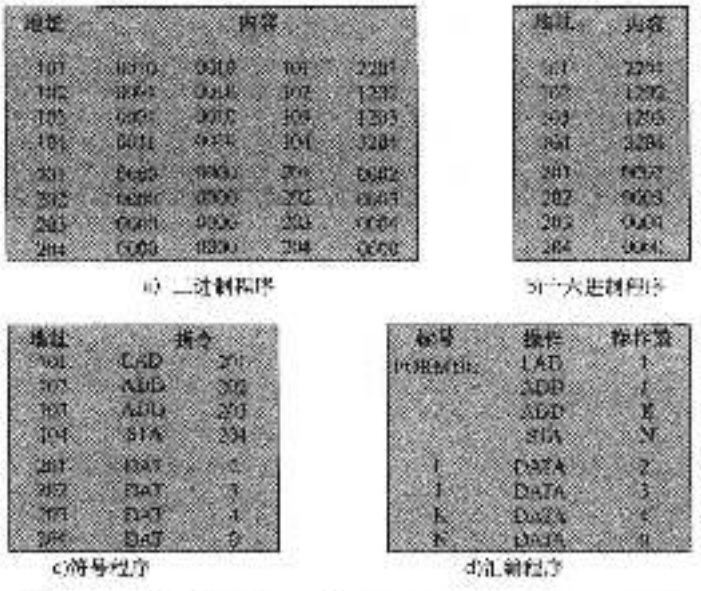


图 11-13　将一条压缩 ADD 指令扩展为对应的普通 ARM 指令

更多的改进是，我们可以利用符号名称或助记符来表示每条指令。这就导致了图 11-13c 所示的符号程序。输入的每一行仍然对应着一个内存位置。每行包括三个字段，以空格隔开。第一个字段是该内存位置的地址。第二个字段是每条指令操作码对应的三个字母的符号。如果一条指令是内存引用指令，那么第三个字段就包含所引用的内存地址。要保存任意的数据到某个位置，我们发明了一条伪指令（pseudoinstruction），其符号是 DAT。该伪指令只是用来指明该行第三字段的十六进制数要被保存到该行第一字段所指定的内存位置。

对于这种类型的输入程序，我们需要一个稍微复杂一些的程序。该程序接收输入的每一行，根据每行第二字段和第三字段（如果有的话），生成相应的二进制数，并保存到第一字段指定的地址中。

使用符号程序可使编程工作轻松些，但还是比较笨拙。尤其是，必须为每个字指定一个绝对地址时。这意味着程序指令和数据都只能被装载到内存中一个固定的地方，而且还必须预先确定。更糟糕的是，假如我们某天想修改程序，添加或删除一行，那么我们必须更新被修改处之后所有行的地址。

更好的一种方式也是普遍使用的一种方式，是使用符号地址，如图 11-13d 所示。每行还是

包含3个字段。第一个字段仍然是地址，但是不再使用绝对的数值地址，而是使用符号。有些行没有地址，意味着这些行的地址是按顺序接着上一行的地址递增。对于引用内存的指令，第二个字段也使用的是符号地址。

经过最后的改进，我们得到了所谓的汇编语言（assembly language）。用汇编语言编写的程序（汇编程序），通过汇编器（assembler）翻译为机器语言。汇编器不但进行上面介绍过的符号操作码翻译，而且对符号地址进行存储器地址分配。

汇编语言的出现是计算机技术发展的一个重要里程碑。它迈出了走向今天我们所使用的高级语言的第一步。虽然现在很少有程序员使用汇编语言了，但所有的机器都提供了对应的汇编语言。对于系统程序，例如编译器和输入/输出例程，它们还是需要的。

11.6 推荐的读物

第10章所援引的那些文献同样可用于本章。[BLAA97]有对指令格式和寻址方式的详细讨论。另外，读者可参考[FLYN85]的关于指令集设计出发点，特别是有关格式的讨论和分析。

BLAA97 Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

FLYN85 Flynn, M.; Johnson, J.; and Wakefield, S. "On Instruction Sets and Their Formats." *IEEE Transactions on Computers*, March 1985.

11.7 关键词、思考题和习题

关键词

autoindexing：自动变址
base-register addressing：基值寄存器寻址
direct addressing：直接寻址
displacement addressing：偏移寻址
effective address：有效地址
immediate addressing：立即寻址
indexing：变址
indirect addressing：间接寻址
instruction format：指令格式
postindexing：后变址
preindexing：前变址
register addressing：寄存器寻址
register indirect addressing：寄存器间接寻址
relative addressing：相对寻址
word：字

思考题

11.1 简要定义立即寻址。
11.2 简要定义直接寻址。
11.3 简要定义间接寻址。
11.4 简要定义寄存器寻址。
11.5 简要定义寄存器间接寻址。
11.6 简要定义偏移寻址。
11.7 简要定义相对寻址。
11.8 自动变址的优点是什么？
11.9 前变址和后变址的区别何在？
11.10 确定寻址位的使用受什么因素影响？
11.11 使用变长指令的优缺点是什么？

习题

11.1 给定如下存储器值，并使用有一累加器的单地址机器，如下指令将把什么值装入累加器？

- 字20含40
- 字30含50

- 字 40 含 60
- 字 50 含 70

(a) LOAD IMMEDIATE 20
(b) LOAD DIRECT 20
(c) LOAD INDIRECT 20
(d) LOAD IMMEDIATE 30
(e) LOAD DIRECT 30
(f) LOAD INDIRECT 30

11.2　若存于程序计数器中的地址标记为 X1，存于 X1 中的指令的地址部分（操作数引用）是 X2，执行此指令所需的操作数存于地址为 X3 的存储器字中。变址寄存器有值 X4。若此指令的寻址方式是（a）直接寻址，（b）间接寻址，（c）PC 相对寻址，（d）变址寻址，那么 X1，X2，X3，X4 应该如何组合，从而得到需要的地址？

11.3　某指令的地址字段含有十进制值 14。对下列寻址方式，其相应的操作数位于何处？
(a) 立即寻址　(b) 直接寻址　(c) 间接寻址
(d) 寄存器寻址　(e) 寄存器间接寻址

11.4　考虑一个 16 位处理器，它的一条装载指令以如下形式出现在主存，起始地址为 200。

200	装载到 AC　Mode
201	500
202	下一条指令

第一字的第一部分指出此指令是将一个值装入累加器。Mode 字段用于指定一种寻址方式。对于某些寻址方式，Mode 字段还指定了一个操存器，并假设指定的操存器是 R1，有值 400。还有一个基址寄存器，它有值 100；地址 201 处的值 500 可以是地址计算的一部分。假定位置 399 处有值 999，位置 400 处有值 1000，如此等等。请对如下寻址方式确定有效地址和将被装入的操作数：
(a) 直接 (b) 立即　(c) 间接　(d) PC 相对寻址
(e) 偏移 (f) 寄存器 (g) 寄存器间接 (h) 用 R1，自动递增变址（autoindexing with increment）

11.5　某 PC 相对方式的分支指令是 3 字节长。此指令的地址是 256028（十进制）；若指令中的带符号偏移量是 −31，请确定转移的目标地址。

11.6　某条 PC 相对方式的分支指令存于地址为 620_{10} 的存储器位置中。它要转移到 530_{10} 位置上。指令中的地址字段是 10 位长，其二进制值是什么？

11.7　若 CPU 取并执行一条间接地址方式指令，该指令是：(a) 一个要求单操作数的计算；(b) 一个分支，那么 CPU 分别需要访问存储器几次？

11.8　IBM 370 不提供间接寻址。假定一个操作数的地址是在主存中，你如何存取此操作数？

11.9　在［COOK82］中，作者建议取消 PC 相对寻址方式，替或使用其他寻址方式，例如栈寻址方式。这个建议有什么缺点？

11.10　x86 包括如下指令：

IMUL op1，op2，立即数

这条指令将操作数 op2（可以是寄存器或存储器）乘以立即操作数值，结果放入 op1（必须是寄存器）中。指令集中再没有其他这类的 3 操作数指令。这种指令的可能用途是什么（提示：考虑变址）？

11.11　考虑一个包括基址带变址（base with indexing）寻址方式的处理器。假设遇到使用这种寻址方式的一条指令，指令给定的偏移量是 1970（十进制）。当前的基址和变址寄存器分别有十进制数 48022 和 8。那么操作数的地址是什么？

11.12　定义：EA = (X) + 为有效地址等于位置 X 的内容，并在有效地址计算后 X 递增一个字长；
EA = −(X) 为有效地址等于位置 X 的内容，并在有效地址计算前 X 递减一字长；
EA = (X) − 为有效地址等于位置 X 的内容，并在有效地址计算后 X 递减一字长。
考虑如下指令，它们都有（操作、源操作数、目的操作数）的格式，并且操作结果放入目的操作数。
(a)　OPX，(X)　　(b) OP (X)，(X) +

(c) OP (X) +, (X) (d) OP -(X), (X)
(e) OP (X), (X) + (f) OP (X) +, (X) +
(g) OP (X) -, (X)

使用 X 作为栈指针，上述哪些指令能自栈弹出顶部两元素，完成所要求的操作（例如，ADD 源到目的并存入目的），并将结果压回栈？对这样的每条指令，栈是朝存储器位置 0 方向还是朝相反方向增长？

11.13 假定有一个面向栈的处理器，包括有 PUSH 和 POP 栈操作。算术运算自动涉及栈顶的 1 或 2 个元素。开始时栈为空。下述指令执行后栈中保留下来的栈元素是什么？

```
PUSH  4
PUSH  7
PUSH  8
ADD
PUSH  10
SUB
MUL
```

11.14 证明说法：32 位指令的功能不会有 16 位指令的功能两倍那样多，是正确的。

11.15 为什么 IBM 决定将每字 36 位转向到每字 32 位的结构，并且是针对什么做出这一决定的？

11.16 假定有一指令集，其指令长度是固定的 16 位长，其中操作数指定符是 6 位长。若有 K 条双操作数指令，L 条零操作数指令，那么能支持的单操作数指令的最大数目为多少？

11.17 设计一种变长操作码，以允许如下指令全都能编码成 36 位指令：

- 指令有两个 15 位地址和一个 3 位寄存器号。
- 指令有一个 15 位地址和一个 3 位寄存器号。
- 指令没有地址或寄存器。

11.18 考虑习题 10.6 的结果。假定，M 是一个 16 位存储器地址，X、Y、Z 或是 16 位地址，或是 4 位寄存器号。单地址机器使用一个累加器。双地址和三地址机器有 16 个寄存器。并且指令能在存储器位置和寄存器的各种组合上操作。并假定指令长度是 4 位的整倍数，操作码是 8 位。为计算 X，每种机器各需要多少位？

11.19 一条指令有两个操作码，它有无任何可能的存在理由？

11.20 16 位的 Zilog Z8001 通常有如下指令格式：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
模式		操作码					w/b	操作数 2				操作数 1			

其中，模式字段指示如何由操作数字段找到操作数。w/b 字段用在某些指令中指示操作数是字节（B）还是 16 位的字（W）。操作数 1 字段可以（取决于模式字段内容）指定 16 个通用寄存器之一。操作数 2 字段可指定除寄存器 0 之外的任一通用寄存器。若此字段为全 0，则原操作码有新的意义。

(a) Z8001 提供了多少的操作码？

(b) 建议一种有效方式来提供更多操作码并指出所涉及的权衡考虑。

CPU 结构和功能

本章要点

- 处理器包括用户可见的寄存器和控制/状态寄存器。用户可见寄存器是指，用户使用机器指令显式或隐式可访问的寄存器。它们可以是通用寄存器，也可以是用于定点或浮点数、地址、变址和段指针这样的专用寄存器。控制和状态寄存器用于控制 CPU 的操作。一个明显的例子是程序计数器。另一重要的例子是程序状态字（PSW），它包含各种状态和条件位，例如反映最近一次算术运算结果的标志位、中断允许位和指示 CPU 当前运行于特权模式下还是用户模式下的状态位。
- 处理器采用指令流水方式来加速指令的执行。从本质上讲，流水是将指令周期分解成几个连续出现的阶段，如取指令、译码指令、确定操作数地址、取操作数、执行指令和写结果操作数。指令向前移动通过这些段，就像车间的一条装配线一样。于是，不同的指令能同时在各个段上工作。不过，转移和指令间相关性的出现，使流水线的设计和使用变得复杂了。

本章继续第三部分未完成的讨论，并为第 13 章和第 14 章关于 RISC 和超标量结构的讨论打基础。

本章以 CPU 组成为开始，然后分析构成处理器内部存储器的寄存器，接着再返回到指令周期的讨论（始于 3.2 节）。在那里将完整地说明指令周期和被称为指令流水线的通用技术。最后以考察 x86 和 ARM 处理器组成的各方面情况来结束本章。

12.1 CPU 组成

为理解 CPU 的组成，让我们考虑对 CPU 的要求，它必须完成以下任务：

- **取指令**：CPU 必须从存储器（寄存器、cache、主存）读取指令。
- **解释指令**：必须对指令进行译码，以确定所要求的动作。
- **取数据**：指令的执行可能要求从存储器或输入/输出（I/O）模块读取数据。
- **处理数据**：指令的执行可能要求对数据完成某些算术或逻辑运算。
- **写数据**：执行的结果可能要求写数据到存储器或 I/O 模块。

显然，为了能做这些事情，CPU 需要暂时存储某些数据。CPU 必须记住当前执行的指令的位置，以便知道从何处得到下一条指令。CPU 还需要在指令执行期间暂时保存指令和数据。换句话说，CPU 需要一个小的内部存储器。

图 12-1 是一个简化的 CPU 视图，指出了 CPU 经由系统总线与系统其余部分的连接。对第 3 章所描述的任一种互连结构，将需要一个类似的接口。读者会回忆起，CPU 的主要部件是一个算术逻辑单元（ALU）和一个控制器（CU）。ALU 完成数据的实际计算或处理。控制器控制数据和指令移入移出 CPU，并控制 ALU 的操作。另外，此图还表示了由一组存储位置组成的极小的内部存

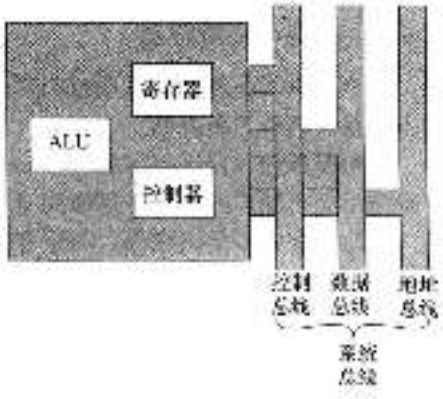


图 12-1 CPU 与系统总线

储器，称为寄存器。

图 12-2 是一个更详细的 CPU 视图，指出了数据传送和逻辑控制的路径，包括一个标记为“CPU 内部总线”的组件。需要有这么一个组件以在各寄存器和 ALU 间传送数据，因为 ALU 实际上只对 CPU 内部存储器中的数据操作。此图还表示了 ALU 典型的基本组件。请注意在作为一个整体的计算机内部结构与 CPU 内部结构之间的相似性，两种结构中都有一个主要组件的小集合（计算机的 CPU、I/O、存储器，CPU 的控制器、ALU、寄存器）通过数据通路连接在一起。

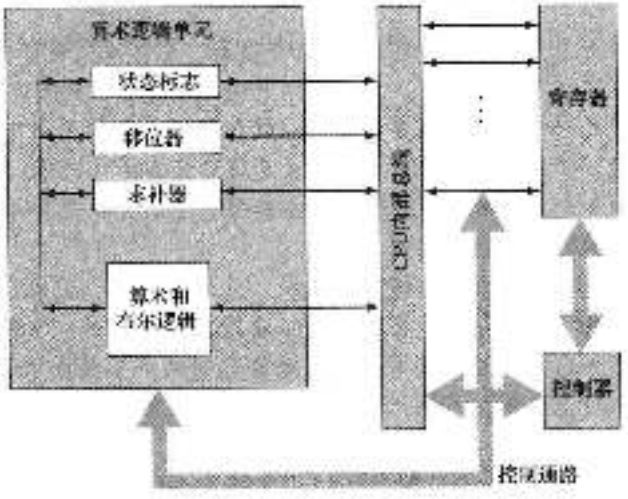


图 12-2 CPU 内部结构

12.2 寄存器组成

正如第 4 章所述，计算机系统采用了存储器分级系统。此分级系统的级别越高，存储器越快、越小，也越昂贵（每位）。CPU 内有一组寄存器，它们的存储器级别在分级系统中位于主存和 cache 之上。CPU 中的寄存器可分为两类：

- **用户可见寄存器**（user-visible register）：允许机器语言或汇编语言的编程人员通过优化寄存器的使用而减少对主存的访问。
- **控制和状态寄存器**（control and status register）：由控制器来控制 CPU 的操作，并由拥有特权的操作系统程序来控制程序的执行。

这两类寄存器的划分界限并不明确。例如，在某些机器上程序计数器是一个用户可见寄存器（如 x86），但在很多机器上却不是。为了顺利进行下面的讨论，我们将采用上述分类。

12.2.1 用户可见寄存器

用户可见寄存器是指可通过机器语言方式访问的寄存器。这些用户可见寄存器可分为：通用、数据、地址、条件码。

通用寄存器（general-purpose register）可被程序员指派各种用途。有时，它们在指令集中的使用是正交于操作的，即任何通用寄存器能为任何操作码容纳操作数。这表现了真正通用的意义。然而，常常不是这样的，而是有某些限制。例如，可能有专用于浮点操作和栈操作的通用寄存器。

某些情况下，通用寄存器可用作寻址功能（如寄存器间接寻址、偏移寻址）。在其他情况下，**数据寄存器**（data register）和**地址寄存器**（address register）之间有部分或完全的区分。数据寄存器仅可用于保持数据而不能用于操作数地址的计算。地址寄存器可以是自身有某些通用性，或是专用于某种具体的寻址方式。例如：

- **段指针**（segment pointer）：在提供分段寻址的机器中（见 8.3 节），段寄存器保持着该段的基地址。可以有多个段寄存器，例如，一个是操作系统的，一个是当前进程的。
- **变址寄存器**（index register）：这些寄存器用于变址寻址，并可能是自动变址的。
- **栈指针**（stack pointer）：若有用户可见的栈寻址方式，则一般来说栈会被分配在存储器中，而 CPU 内有一专用的寄存器指向栈顶。这允许隐含寻址，即 push、pop 和其他不需要显式地指定栈操作数的栈指令。

这里有几个设计出发点。重要的一点是，使用完全通用的寄存器还是规定各寄存器的用途。在上一章我们已接触到这个问题，因为它影响指令集的设计。对专用寄存器的使用，一个操作数指定符所引用的寄存器类型通常能隐含在操作码中。操作数指定符必须做的只是标识这一组专用寄存器中的某一个将被使用，而不是所有寄存器中的某一个，于是节省了位数。另一方面，这种规定又限制了程序员的灵活性。

另一设计出发点是寄存器数量。同样，这影响指令集的设计，因为寄存器越多，需要的操作数指定符的位数也越多。正如前面所讨论的，某些机器有 8 到 32 个寄存器是适宜的［LUND77］。太少的寄存器会导致更多的存储器访问，太多的寄存器又不能显著地减少存储器访问（如［WILL90］）。然而，一种新的方法使得在某些 RISC 系统中展示出了使用上百个寄存器的优点，这些将在第 13 章讨论。

最后，这里还有一个寄存器长度问题。用于保存地址的寄存器明显要求其长度足以容纳最长的地址。数据寄存器应能保存大多数数据类型的值。某些机器允许两个相邻的寄存器作为一个寄存器来保持两倍长度的值。

最后一类寄存器是用于保存**条件码**（condition code），亦称为**标志**（flag）的寄存器，它们至少是部分用户可见的。CPU 硬件设置这些条件位作为操作的结果。例如，一个算术运算可能产生一个正的、负的、零或溢出的结果。除结果本身存于寄存器或存储器之外，一个条件码亦相应被设置。这些条件码可被后面的条件分支指令所测试。

条件码通常被收集到一个或多个寄存器中。通常，它们构成控制寄存器的一部分。机器指令允许这些位以隐含引用的方式读出，但它们不能被程序员更改。

许多处理器，包括那些基于 IA-64 体系结构的处理器和 MIPS 处理器，根本不使用条件码。相反，它们采用测试条件分支指令，这种指令指定一种比较操作，并根据比较的结果产生控制转移动作，不保存条件码。基于［DERO87］的表 12-1 列出了条件码的主要优缺点。

表 12-1 条件码

优点	缺点
1. 因条件码自设置的算术和数据传送指令建立，故它们能减少对比较和测试类指令的需求。 2. 条件指令，例如“BRANCH”这样的条件分支指令要比测试条件分支（“TEST AND BRANCH”这样的复合指令简单。 3. 条件码便利了多路分支选择。例如，一条测试指令之后可带两条分支，一条按小于或等于 0 的条件码进行，另一条按大于 0 进行。	1. 条件码增加了硬软件的复杂性。条件码位被不同的指令以不同的方式频繁修改，使编程序编程人员和编译器设计人员的工作更为困难。 2. 条件码不正交，通常它们不是主数据路径的一部分，故要求额外的硬件连线。 3. 使用条件码的机器经常需要为诸如位检查、循环控制、原子式信号量操作等这类特殊情况添加专门的非条件码指令。 4. 在流水实现中，条件码要求专门的同步化，以避免冲突。

某些机器中，一个子程序调用将导致自动保存所有用户可见的寄存器，在返回时自动取回。这些保存和恢复是作为调用和返回指令执行功能的一部分由 CPU 完成的。这就允许各个子程序独立地使用用户可见寄存器。而在其他一些机器上，子程序调用之前保存相关用户可见寄存器的内容是程序员的责任，他们要在程序中为此专门安排一些指令。

12.2.2 控制和状态寄存器

有一类寄存器在CPU中起着控制操作的作用。它们中的大多数，在大多数机器上，是用户不可见的。某些在控制或操作系统模式下执行的机器指令是用户可见的。

当然，不同的机器将有不同的寄存器组织并使用不同的术语。这里，我们列出一个相对完整的寄存器类型列表，并予以简短描述。

对于指令执行，有4种寄存器是至关重要的。

- 程序计数器（PC）：存有待取指令的地址。
- 指令寄存器（IR）：存有最近取来的指令。
- 存储器地址寄存器（MAR）：存有存储器位置的地址。
- 存储器缓冲寄存器（MBR）：存有将被写入存储器的数据字或最近从存储器读出的字。

不是所有的处理器都有专门称为MAR和MBR的寄存器，不过还是会有某种等价的缓冲机制，其中要被写到系统总线上的数据位，以及从系统总线上读到的数据位，都会被暂时保留或存储起来。

通常，在每次取指令之后，PC的内容即被CPU更改，故它总是指向将被执行的下一条指令。转移或跳步指令亦修改PC的内容。取来的指令装入IR，在那里分析操作码和操作数指定符。与存储器的数据交换使用MAR和MBR。在总线组织的系统中，MAR直接与地址总线相连，MBR直接与数据总线相连。然后，用户可见寄存器再与MBR交换数据。

刚才提到的4个寄存器用于CPU和存储器之间的数据传送。在CPU内，数据必须提交给ALU来处理。ALU可对MBR和用户可见寄存器直接存取。相应地也可在ALU的边界上有另外的缓冲寄存器，这些寄存器能作为ALU的输入和输出，可与MBR和用户可见的寄存器交换数据。

很多CPU设计都包括常称为程序状态字（program status word，PSW）的一个或一组寄存器。PSW一般含有条件码加上其他状态信息。通常PSW包括下列字段或标志：

- 符号（sign）：容纳最后算术运算结果的符号位。
- 零（zero）：当结果是0时被置位。
- 进位（carry）：若操作导致最高位有向上的进位（加法）或借位（减法）时被置位。用于多字算术运算。
- 等于（equal）：若逻辑比较的结果相等，则置位。
- 溢出（overflow）：用于指示算术溢出。
- 中断允许/禁止：用于允许或禁止中断。
- 监管（supervisor）：指出CPU是执行在监管模式中还是在用户模式中。某些特权的指令只能在监管模式中执行，某些存储器区域也只能在监管模式中被访问。

此外具体CPU设计中可能还会有其他额外的有关状态和控制的寄存器。例如，除了PSW之外，可能有一个指向存储器块（例如进程控制块PCB）的指针寄存器，而此存储块含有附加的状态信息。在使用向量式中断的机器中，可能提供有一个中断向量寄存器。若栈用于实现某些功能（例如子程序调用），则需要有一个系统栈指针。对于虚拟存储器系统，可能会有一个页表指针寄存器。最后，在I/O操作控制方面也可能需要有专门的寄存器。

设计控制和状态寄存器组织时有几个因素需考虑。一个关键的考虑是对操作系统的支持。某些类型的控制信息是专门为操作系统使用的。若CPU设计者对将要使用的操作系统有基本的了解，则寄存器的组织可能在一定程度上为该操作系统定制。

另一个关键的考虑是控制信息在寄存器和存储器之间的分配。一种普遍的做法是将存储器最前面（最低地址）的几百或几千个字用于控制目的。设计者必须决定多少控制信息应在寄存器中，多少应在存储器中。这通常要在成本和速度之间进行权衡。

12.2.3　微处理器寄存器组成的例子

考察和比较一些可比系统的寄存器组成是有指导意义的。本节我们考察大约在同一时期设计出的 2 个 16 位微处理器：Motorola MC68000［STRI79］和 Intel 8086［MORS78］。图 12-3a 和图 12-3b 分别给出了上述两种处理器的寄存器组成，像存储器地址寄存器这样的纯内部寄存器未在图中示出。

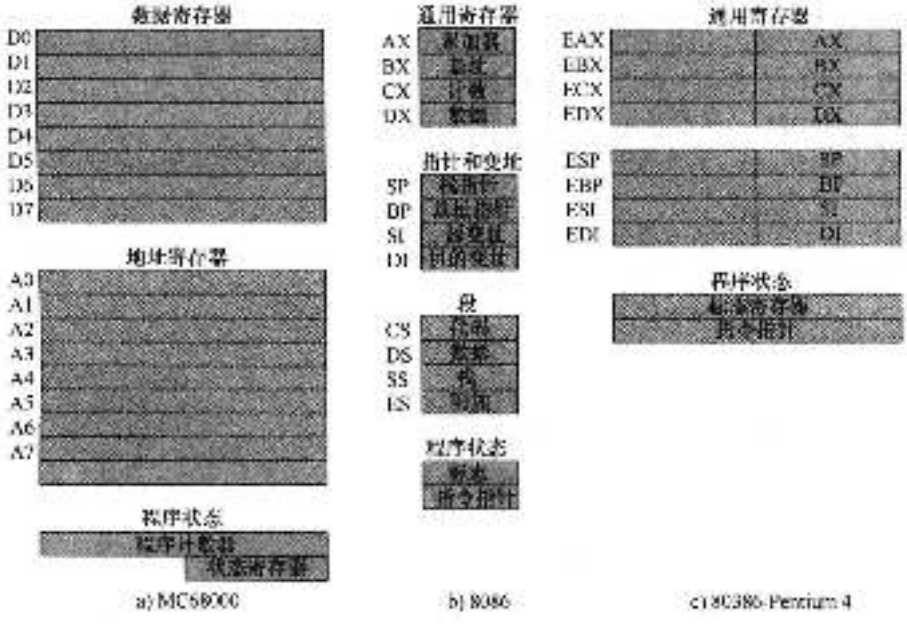


图 12-3　微处理器寄存器组成的例子

MC68000 将它的 32 位寄存器分成 8 个数据寄存器和 9 个地址寄存器。8 个数据寄存器主要用于数据操作，并在寻址方式中用作变址寄存器。寄存器的宽度允许 8 位、16 位或 32 位数据操作，具体取决于操作码。地址寄存器包含 32 位（不分段）地址，其中两个寄存器亦用作栈指针，一个用于操作系统，一个用于用户，这取决于当前的执行模式。这两个寄存器都编号为 7，因为任何时刻只能一个在使用。MC68000 还包括一个 32 位程序计数器和一个 16 位状态寄存器。

Motorola 设计小组也希望有一个很规整的指令集而不带有专门的目的寄存器。对代码效率的考虑使他们将寄存器分成两个功能组件，在每个寄存器指定符上节省了 1 位。这看起来是在完全通用性和代码紧凑性之间的一个合理折中。

Intel 8086 采取另一种不同的方法来组织寄存器，每个寄存器都有专门的用途，虽然某些寄存器也可通用。8086 含有 4 个 16 位数据寄存器，它们亦可按字节（8 位）来使用；还含有 4 个 16 位的指针和变址寄存器。数据寄存器在某些指令中可用作通用寄存器，而在另一些指令中它们是隐含被使用的。例如，乘法指令总是使用累加器。4 个指针寄存器亦是在几种操作中隐含被使用，每个用于保存段内位移。8086 还有 4 个段寄存器，其中三个以一种专门的隐含方式来使用，分别指向一个包含当前指令的段（对转移指令特别有用），一个包含数据的段，一个包含栈的段。这些专门的隐含方式的使用，以减少灵活性为代价，提供了编码的紧凑性。8086 还包括一个指令指针寄存器和一组状态和控制标志，其中每个状态和控制标志都是 1 位。

通过这个比较可以看清楚的一点是，到目前为止，关于组织 CPU 寄存器的最好方式还没有一个普遍接受的原则［TOON81］。正如对整个指令集设计的情况一样，也有众多的 CPU 设计观点，这些都还是有待品评的事情。

关于寄存器组织设计的第二个有指导意义的观点说明如图 12-3c 所示。此图表示了 Intel 80386 [ELAY85] 的用户可见寄存器的组织。80386 是 32 位微处理器，并设计成 8086 的扩展[⊖]。80386 使用 32 位寄存器。然而，为了向在早先机器上写成的程序提供向上兼容，80386 将原先的寄存器组织嵌入到新组织中。给定这种设计限制，这个 32 位微处理器的寄存器组织设计明显在灵活性上受到制约。

12.3 指令周期

前面 3.2 节已描述过 CPU 的指令周期（参见图 3.9）。指令周期包括如下子周期：

- **取指**（fetch）：将下一条指令由存储器读入 CPU。
- **执行**（execute）：解释操作码并完成指定的操作。
- **中断**（interrupt）：若中断是允许的并且有中断发生，则保存当前进程的状态并为此中断服务。

现在我们来详细描述指令周期。首先，必须引入一个另外的子周期，称为间接周期（indirect cycle）。

12.3.1 间接周期

在第 11 章我们已看到，指令的执行可能涉及一个或多个存储器中的操作数，它们每个都要求一次存储器访问。而且，若使用间接寻址，则还需要额外的存储器访问。

可把间接地址的读取看成是一个额外的指令子周期，其过程显示于图 12-4。动作的主线由交替的取指令和指令执行动作组成。取来一条指令之后，要对它进行检查以确定是否需要间接寻址。如果是，则所要求的操作数使用间接寻址方式取来。在执行之后，可能有一个中断在取下一条指令之前被处理。

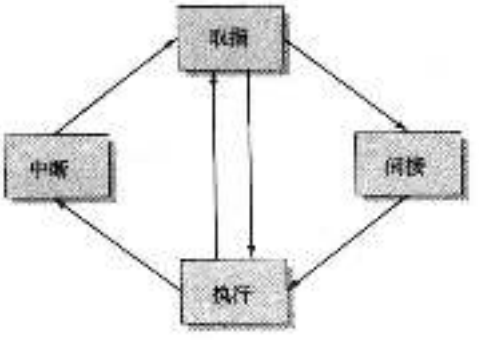


图 12-4 指令周期

观察此过程的另一种方式如图 12-5 所示，它是图 3-12 的改进版。它更准确地说明了指令周期的实质。一旦取来一条指令，它的操作数指定符必须被识别。然后读取存储器中的每个操作数，这个过程可能要求间接寻址。寄存器操作数不需要从存储器读取。一旦操作被执行，可能需要一个类似的过程将结果存入主存。

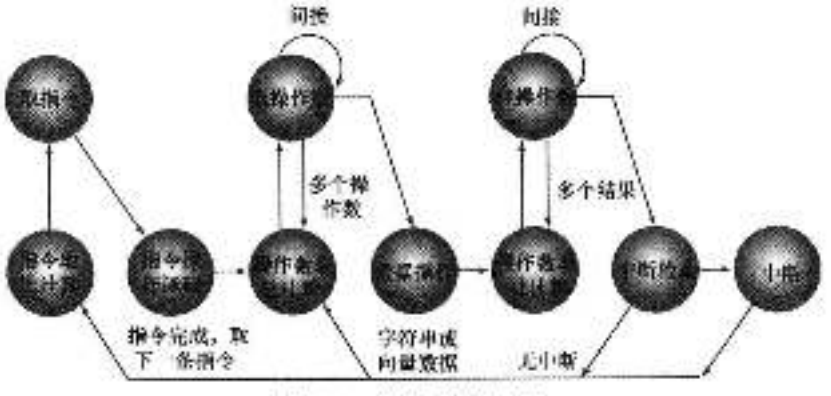


图 12-5 指令周期状态图

⊖ 因 MC68000 已使用了 32 位寄存器，所以它为全 32 位扩展 MC68020 [MACG84] 采用的是相似的寄存器组织。

12.3.2　数据流

指令周期期间，所发生事件的实际序列取决于 CPU 的具体设计。不过我们还是可以从一般意义上列出哪些事件是应该要发生的。假定一个 CPU 有一个存储器地址寄存器（MAR）、一个存储器缓冲寄存器（MBR）、一个程序计数器（PC）和一个指令寄存器（IR）。

在取指周期，一条指令由存储器读入。图 12-6 显示了此期间的数据流动。开始时，PC 存有待取的下一条指令的地址。这个地址被传送到 MAR 并放到地址总线上。控制器发出一个存储器读的请求，存储器把结果放到数据总线上，CPU 将其复制到 MBR，然后传送到 IR。在此期间，PC 增 1，为下次取指令做好准备。

一旦经历过取指周期，控制器检查 IR 的内容，以确定是否有一个使用间接寻址的操作数指定符。若是，则进入间接周期，如图 12-7 所示，这是一个简单周期。MBR 最右边的 N 位是一个地址引用，被传送到 MAR。然后，控制器请求一个存储器读，得到所要求的操作数地址，并送入 MBR。

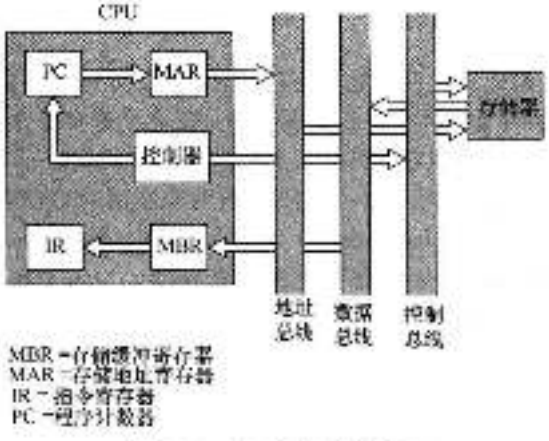


图 12-6　数据流与取指周期

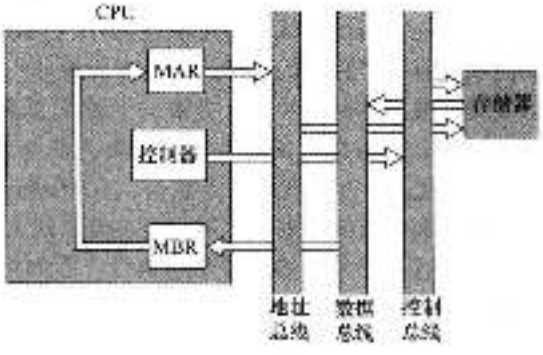


图 12-7　数据流与间接周期

取指和间接周期是简单的，并且是可预测的。执行周期则有多种表现形式，具体是哪种形式取决于各种不同的指令里哪一条当前在 IR 中。这个周期可能涉及寄存器间的数据传送，对存储器或 I/O 设备的读或写，以及 ALU 的功能使用。

像取指和间接周期一样，中断周期是简单的并可预测的（见图 12-8）。PC 的当前内容必须被保存，以便在中断之后 CPU 能恢复先前的动作。于是，PC 的内容传送到 MBR，将被写入存储器。为此目的，一个专门的存储器位置被控制器装入 MAR。它可能是一个栈指针。随后，中断子程序的地址装入 PC。结果是，下一指令周期将以取此相应的指令而开始。

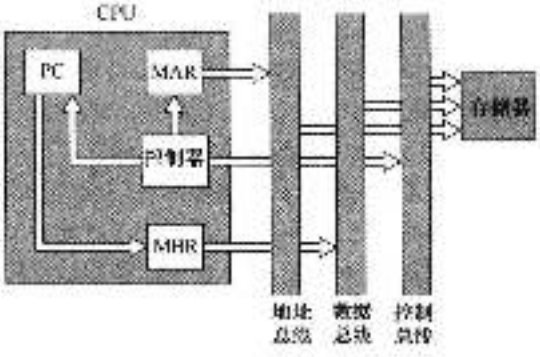


图 12-8　数据流与中断周期

12.4　指令流水线技术

随着计算机系统的发展，特别是集成电路工艺上的改进，例如更快的电路，可以实现更高的性能。另外，CPU 组织的改进也能改善性能。我们已经看到过这样的例子，如使用多个寄存器而不是单一的累加器，又如使用高速缓存等。另外一种使用非常普遍的组织方法是指令流水线技术。

12.4.1 流水线策略

指令流水线类似于工厂中装配线的使用。装配线利用了这样一个事实，即一个产品要经过几个制作步骤。通过把制作过程安排在一条装配线上，多个产品能在各个阶段同时被加工。这种过程称为流水处理（pipelining），因为在一条流水线上，当先前接收的输入已成为加工的结果出现在另一端时，新的输入又在一端被接收进来。

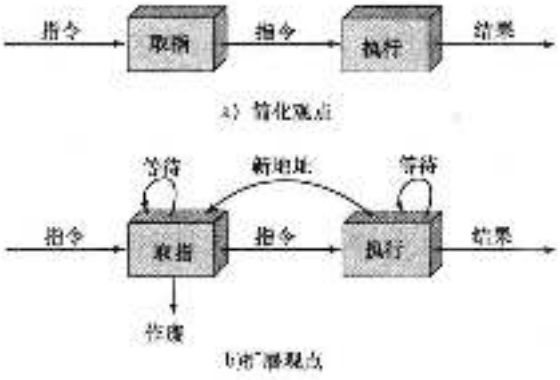


图 12-9 两阶段指令流水线

将这种概念施加到指令的执行上，我们必须认识到，事实上一条指令的执行也是分成几个步骤。图 12-5 就是一个例子，它将指令周期分成 10 个顺序的任务。很清楚，这里应有实施流水线技术的某种机会。

作为一种简化的方法，考虑将指令处理分成两个阶段：取指令和执行指令。在一条指令执行期间，主存可能没有存取的操作。此时主存能用于取下一条指令，从而这个取指操作与当前指令的执行并行工作。图 12-9a 描述了这种方法。此流水线有两个独立的阶段。第一个阶段取一条指令并缓存它，当第二个阶段空闲时，将第一个阶段缓存的指令输送给它。当第二个阶段正在执行此指令时，第一个阶段利用未使用的存储器周期读取下一条指令并缓存它。这称为指令预取（instruction prefetch）或取指交叠（fetch overlap）。注意这种方式涉及指令缓存（instruction buffering），因此需要更多的寄存器。一般来说，流水处理都需要额外的寄存器用以在流水段之间保存数据。

显然这种处理将加快指令的执行。若取指和执行这两个阶段是相等的时间，则指令周期时间将是原来的一半。然而，若我们更仔细地查看这个流水线（见图 12-9b），将会看到实现这种执行速度的翻倍是不太可能的，理由有二：

（1）执行时间一般要长于取指时间。执行将涉及读取和保存操作数以及完成某些操作。于是，取指阶段可能必须等待一定的时间才能排空它的缓冲器。

（2）条件分支指令使得待取的下一条指令的地址是未知的。于是，取指阶段必须等待，直到它能由执行阶段得到下一条指令地址。而在取下一条指令时执行阶段又可能必须等待。

由第二种情况造成的时间损失可通过推测来减少。一个简单的规则如下：当一条条件分支指令由取指阶段传送到执行阶段时，取指阶段读取存储器中此分支指令之后的指令。于是，若转移未发生，则没有时间损失；若转移发生，则已读取的指令要作废，并读取新的指令。

虽然这些因素降低了两段流水线的潜在效率，但还是带来某种加速。为获得进一步的加速，流水线必须有更多的阶段。让我们考虑指令处理的如下分解。

- **取指令（FI）**：读下一条预期的指令到缓冲器。
- **译码指令（DI）**：确定操作码和操作数指定符。
- **计算操作数（CO）**：计算每个源操作数的有效地址，这可能涉及偏移寻址、寄存器间接寻址、间接寻址或其他形式的地址计算。
- **取操作数（FO）**：从存储器取出每个操作数。寄存器中的操作数不需要取。
- **执行指令（EI）**：完成指定的操作。若有指定的目的操作数位置，则将结果写入此位置。
- **写操作数（WO）**：将结果存入存储器。

按照这种分解，各个阶段所需要的时间几乎是相等的。为便于说明，假定是相等的时间。图 12-10 则表示了一个这样的 6 段流水线。它能将 9 条指令的执行时间由 54 个时间单位减少到 14

个时间单位。

以下是几点说明：此图假设每条指令都通过流水线的 6 个阶段，但并不总是这种情况。例如，一条 LOAD 指令就不需要 WO 阶段。然而，为简化流水线硬件设计，就在假定每条指令都要求这 6 个阶段的基础上来建立时序。还有，此图是假定所有阶段都能并行完成，具体地说，是假定没有存储器冲突。如 FI、FO 和 WO 都涉及存储器访问，此图暗示它们是能同时进行的。大多数存储器系统不准许这样，然而可能所要求的值在 cache 中，或者 FO 或 WO 阶段是个空操作。于是，多数情况下，这种存储器冲突并不会减慢流水处理速度。

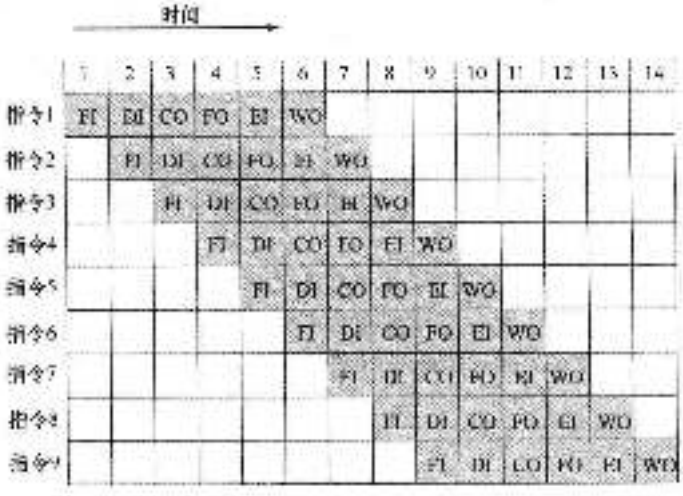


图 12-10　指令流水线操作时序图

有几个因素限制了性能提升。若 6 个阶段不全是相等的时间，则正如我们在前面讨论的两阶段流水线那样，会在各个流水阶段涉及某种等待。另一困难是条件转移指令，它能使若干指令的读取变为无效。另一不可预料的事件是中断。图 12-11 说明了条件转移的影响，其中使用与图 12-10 同样的程序。假定，指令 3 是一个可能转到指令 15 的条件转移指令。直到指令执行之前，没办法知道下一条指令到底是哪条指令。此例中，流水线只是简单地按顺序装入下一条指令（指令 4）并继续执行。图 12-10 表示转移未发生，我们得到了全面的性能提升。图 12-11 表示的是转移发生的情况，但这直到时间单位 7 结束时才能确定。此时流水线必须清除那些已取来的无用指令。这样，在时间单位 8，指令 15 进入流水线。在时间单位 9～12 期间没有指令完成，这是由于我们不能预测转移是否发生所导致的性能下降。图 12-12 指出考虑到转移和中断，流水线所需要的逻辑。

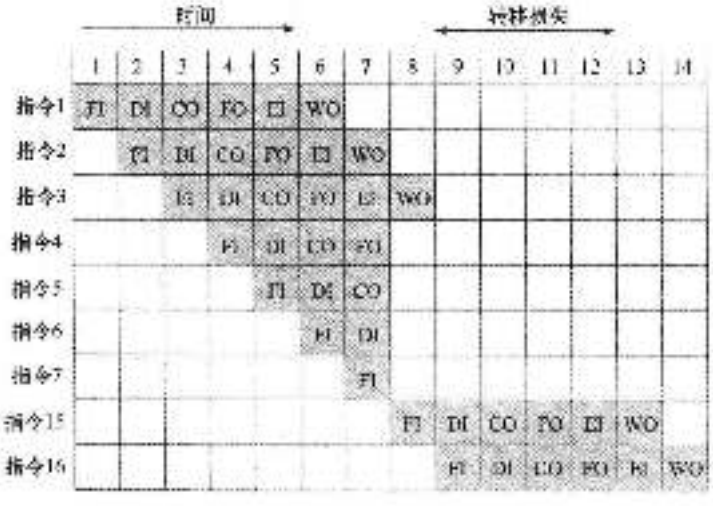


图 12-11　条件转移对指令流水线操作的影响

还有一些问题，它们不出现在简单的两阶段流水线中。CO 阶段可能需要某个寄存器的内容，而此值可能被仍在流水线中的完前指令所修改。其他的寄存器和存储器冲突也可能出现。系统必须考虑到处理这类冲突的逻辑。

为清晰起见，换一种方式来查看流水线操作将是有益的。图 12-10 和图 12-11 是水平方向表示时间，每行表示一条指令的执行。新给出的图 12-13 是垂直向下表示时间的进行，而每一行表示的是给定时间点的流水线状态。在图 12-13a（对应于图 12-10）中，时间 6 时流水线已满，有 6 条不同的指令正在流水线各阶段进行，并一直保持满负荷运行，直到时间 9；假定 19 是最后一条待执行指令。在图 12-13b（对应于图 12-11）中，时间 6 和 7 时流水线是满的。在时间 7，一

条转移指令 I3 正处于执行阶段，它将引发转移到指令 I15。此时，指令 I4～I7 都要由流水线中被清除出去，于是，时间 8 时只有 I3 和 I15 两条指令在流水线中。

从前面的讨论，我们可能会认为流水线中阶段数目越多，执行速度越快。但 IBM S/370 的设计者指出，有两个因素将使这种看似简单的高性能设计失败［ANDE67a］，而且这些观点至今仍是有效的。

（1）流水线的每一阶段，都会有某些开销涉及数据在缓冲器间的传送，以及涉及完成各种准备和递交功能。这些开销能明显地使单一指令总的执行时间变长。当顺序指令之间发生逻辑依赖（这种逻辑依赖或是因为大量使用了分支指令，或是因为存储器访问的相关性）时，这个问题就变得特别严重。

（2）优化流水线的使用和处理存储器及寄存器相关性所需的控制逻辑总量，会随着流水线阶段数的增长而急剧增长。这将导致这样一种情形，阶段之间的门控逻辑比这些阶段本身的逻辑还要复杂。

另一个要考虑的因素是锁存延迟（latching delay），即流水线阶段之间的缓冲需要一定时间来完成其操作，而这会增加指令周期的时间。

指令流水线是一种提高性能的强有力技术，但需要精心设计，以合理的复杂性达到最优的效果。

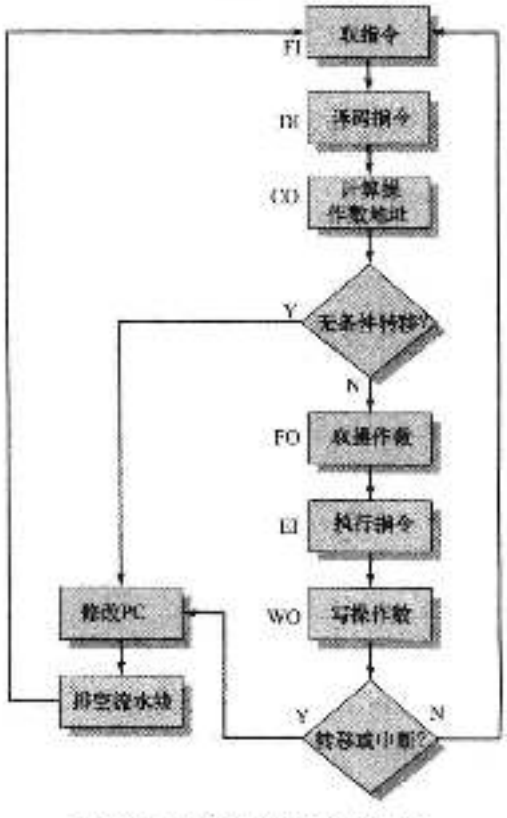


图 12-12 6 阶段 CPU 指令流水线

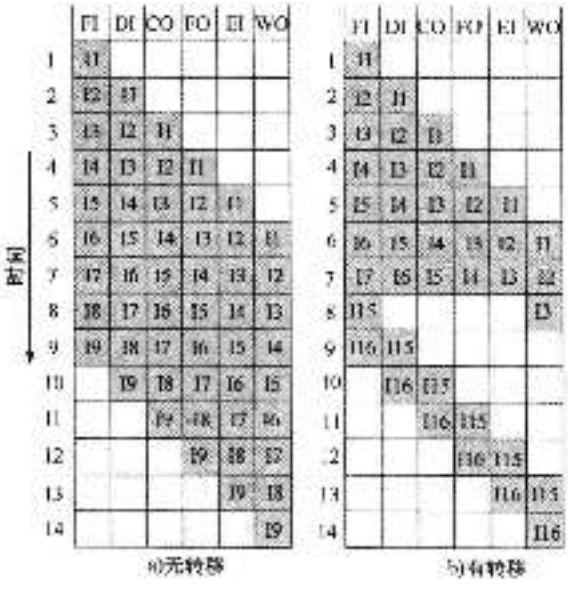


图 12-13 流水线的另一种描述方式

12.4.2 流水线性能

下面我们对流水线性能和相对加速予以简单度量（基于［HWAN93］中的讨论）。指令流水线的周期时间 τ，是在流水线中将一组指令推进一段所需的时间，图 12-10 和图 12-11 中的每一列都代表一个周期时间，周期时间能表示成：

$$\tau = \max_i[\tau_i] + d = \tau_m + d \qquad 1 \leqslant i \leqslant k$$

这里　τ_i = 流水线第 i 段的电路延迟时间

τ_m = 最大段延迟（通过耗时最长段的延迟）

k = 指令流水线段数

d = 锁存延时（数据和信号从上一段送到下一段所需的段间锁存接收时间）

通常，延时 d 等于时钟脉冲的宽度并且 $\tau_m >> d$。现假设有 n 条指令在进行，无转移发生。令 $T_{k,n}$ 为 k 阶段流水线执行所有 n 条指令所需的总时间，则有：

$$T_{k,n} = [k + (n-1)]\tau \tag{12.1}$$

完成第 1 条指令执行需要 k 个周期。完成其余 $n-1$ 条指令需要 $n-1$ 个周期[⊖]。这个等式很容易由图 12-10 得到验证，第 9 条指令在周期 14 时完成，于是有：

$$14 = [6 + (9-1)]$$

使用指令流水线相对于不使用流水线的加速比定义为：

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{k + (n-1)} \tag{12.2}$$

图 12-14a 给出无分支情况下加速比随指令数的变化关系，当 $n \to \infty$ 时，$S_k \to k$，即我们获得 k 倍加速。图 12-14b 给出加速比随指令流水线段数的变化关系[⊖]。在这种情况下，加速比能接近进入流水线而且无转移的指令数。于是，更多的流水线段数能带来更大的潜在加速比。然而，增加更多的段所带来的增益，必须考虑到成本的增加、段间延时的增多，以及遇到转移指令而要求清空流水线的这些事实。

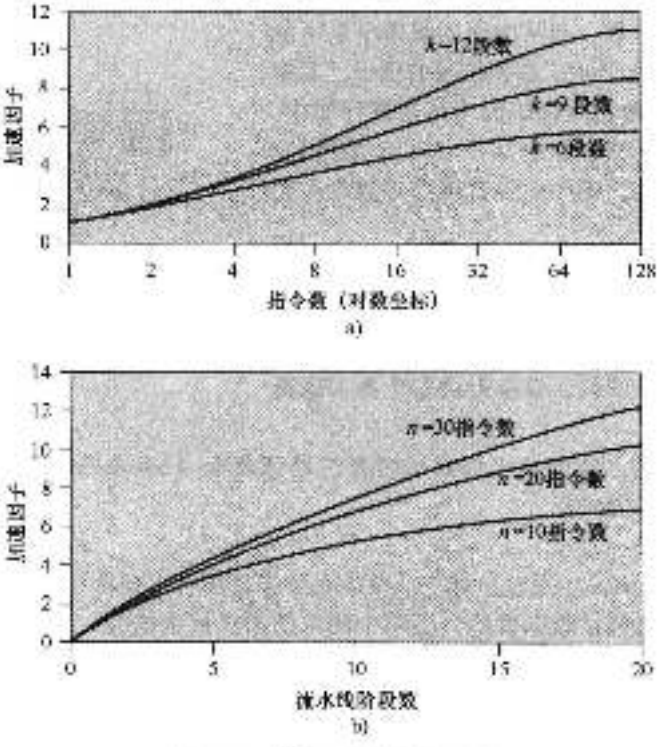


图 12-14　指令流水的加速因子

⊖ 这里有点儿不精确，仅当所有段都满时，周期时间才等于 τ 的最大值；而在开始的 1 个或少数几个流周内周期时间稍要小一些。

⊖ 注意，图 12-14a 的 x 轴是对数坐标，图 12-14b 的 x 轴是线性标度。

12.4.3 流水线冒险

在前一小节，我们提到某些情况会导致流水线不能达到理想加速比。在本小节中，我们将更加系统地考察这个问题。第 14 章介绍了超标量流水线组织中的复杂性后，也将更细致地回顾这个问题。

流水线冒险（pipeline hazard）发生在流水线或流水线的某个部分，因为某些条件不允许流水线继续运行，而必须停顿（stall）的时候。这样的流水线停顿也通常被称为流水线空泡（pipeline bubble）。有 3 种类型的流水线冒险：资源冒险、数据冒险和控制冒险。

1. 资源冒险

资源冒险（resource hazard）发生在两条（或多条）已进入流水线的指令需要使用相同资源的时候。结果就是在流水线某个部分，这些指令必须串行执行，而不是并行执行。资源冒险在有些时候也称为结构冒险（structural hazard）。

让我们考察一个简单的资源冒险例子。假设有一个简单的 5 阶段流水线，其中每个阶段的执行时间是一个时钟周期。图 12-15a 显示了理想的运行情况，其中每个时钟周期都有一条新指令进入流水线。现在假设主存只有一个端口，所有的指令读取以及数据装载和保存都一次只能有一个单独操作。另外，假设没有高速缓存。在这种情况下，从内存装载一个操作数，或向内存保存一个操作数，与从内存读取指令是不能同时进行的。图 12-15b 显示了这种情况的例子，其中假设指令 I1 的源操作数在内存中而不是在寄存器中。这样，流水线的取指阶段在进行指令 I3 的读取之前，必须空闲一个时钟周期。图中假设其他操作数都在寄存器中。

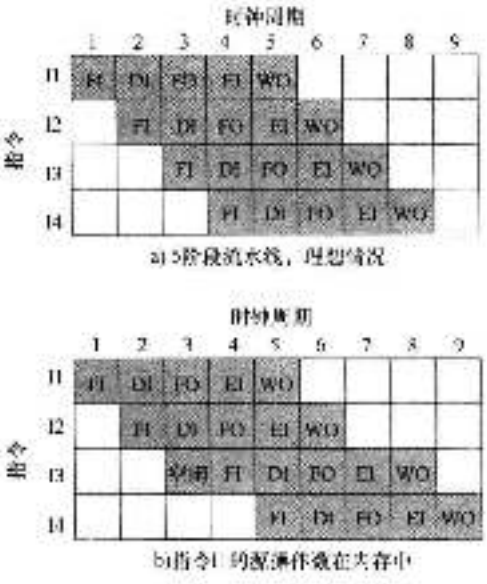


图 12-15 资源冒险的例子

另一个资源冲突的例子是当多条指令可以进入执行阶段，但只有一个算术逻辑单元的时候发生的情况。资源冒险的一种解决办法是增加可用的资源，例如提供访问主存的多个端口，以及提供多个算术逻辑单元。

一种分析资源冲突，辅助流水线设计的方法是预约表（reservation table）。我们会在附录 I 中考察预约表。



2. 数据冒险

数据冒险（data hazard）发生在对一个操作数位置的访问出现冲突的时候。通常可以用如下形式来描述数据冒险：程序中的两条指令依次执行，并且都将访问同一个内存或寄存器操作数。如果两条指令的执行是严格串行的，那么没有问题发生。但是，如果这两条指令在流水线中运行，那么有可能操作数会不按次序更新，从而导致与严格串行执行不一样的结果。换句话说，就是由于使用了流水线，导致程序运行产生了不正确的结果。

作为一个例子，考虑下面的 x86 机器指令序列：

```
ADD EAX, EBX        /* EAX = EAX + EBX
SUB ECX, EAX        /* ECX = ECX - EAX
```

第一条指令把 32 位寄存器 EAX 和 EBX 中的内容相加，把结果保存回 EAX 寄存器。第二条

指令从ECX中减去EAX的值，并把结果保存回ECX。图12-16显示了流水线的行为。ADD指令在第5阶段结束之前才会更新寄存器EAX，这发生在时钟周期5。但是SUB指令在它执行的第2阶段就需要EAX的最新值，这发生在时钟周期4。要保证正确的操作，流水线必须停顿2个时钟周期。这样，在缺乏专门硬件和特殊的规避算法条件下，这一数据冒险导致流水线的运行效率降低。

有3种类型的数据冒险：

- 写后读（Read After Write，RAW）或真相关（true dependency）：一条指令改写一个寄存器或内存位置，而后续的指令从所改写的寄存器或内存位置读取数据。如果在写操作完成之前，读操作就开始进行，那么就会发生冒险。
- 读后写（Write After Read，WAR）或反相关（antidependency）：一条指令读一个寄存器或内存位置，而后续的指令将改写该寄存器或内存位置的内容。如果在读操作完成之前，写操作就开始进行，那么就会发生冒险。
- 写后写（Write After Write，WAW）或输出相关（output dependency）：两条指令要改写同一个寄存器或内存位置。如果这两条指令的写操作发生次序与期望的次序相反，那么就会发生冒险。

图12-16显示了一个RAW型的冒险。另外两种类型的冒险在第14章有细致深入的讨论。



图12-16 数据冒险的例子

3. 控制冒险

控制冒险（control hazard），又称为分支冒险（branch hazard），发生在流水线对分支转移做出了错误的预测，因此读取了在后期必须取消的指令之时。我们接下来讨论处理控制冒险的各种办法。

12.4.4 处理分支指令

设计指令流水线的一个主要问题是，保证有一个稳定的指令流进入流水线的最初几个阶段。正如我们已看到的，主要的障碍是条件分支指令。这种指令的特点是，直到指令实际被执行之前，不可能确定转移是否发生。

已有几种方法用于处理条件分支指令：

- 多个指令流（multiple streams）
- 预取分支目标（prefetch branch target）
- 循环缓冲器（loop buffer）
- 分支预测（branch prediction）
- 延迟分支（delayed branch）

1. 多个指令流

一个简单的流水线之所以蒙受分支指令带来的惩罚，在于它必须在取下一条指令时做出二选一的选择，而且其选择可能是错的。一个强制的方法是复制流水线的开始部分，并允许流水线同时取这两条指令，使用两个指令流。这种方法有两个问题：

- 使用多个流水线，会有对寄存器和存储器访问的竞争延迟。
- 在原先的分支判断还没解决之前，可能又有另外的分支指令进入流水线（不管哪一路）。这样又需要添加指令流。

尽管有这些缺点，这个策略也能改善性能。使用两条或多条指令流水线的例子是 IBM 370/168 和 IBM 3033 机。

2. 预取分支目标

识别出一个条件分支指令时，除了取此分支指令之后的指令外，分支目标处的指令也被取来。这个目标被保存直到分支指令被执行。若是分支发生，则目标已经被预取来了。

IBM 360/91 使用这种方法。

3. 循环缓冲器

循环缓冲器是由流水线指令取指阶段维护的一个小的但极高速的存储器，含有 n 条最近顺序取来的指令。若一个转移将要发生，硬件首先检查转移目标是否在此缓冲器中。若是，则下一条指令由此缓冲器取得。循环缓冲器有三个好处：

（1）采用指令预取，循环缓冲器将含有某些地址在当前指令地址之前的指令。于是，顺序取来的指令都可能被使用而不需要通常的存储器访问时间。

（2）若一个转移的目标恰恰是在此转移指令之前的少数几个位置上，则目标已在缓冲器中。这对于相当普通的 IF-THEN 和 IF-THEN-ELSE 序列特别有用。

（3）这一策略非常适合处理循环或迭代，因此命名为循环缓冲器。若循环缓冲器充分大，足以容纳循环的全部指令，则这些指令只需要第一次循环时由存储器取来，后面的循环不需要再取指令。

从原理上讲，循环缓冲器类似于指令高速缓存。不同在于，循环缓冲器只保留顺序的指令，因而容量较小，成本也较低。

图 12.17 给出一个循环缓冲器的例子。若此缓冲器容纳 256 字节，并且使用字节寻址，则转移地址的低 8 位可用于对缓冲器的索引。而其余的高有效位被检查，以确定转移目标是否已在此缓冲器所捕获的上下文中。

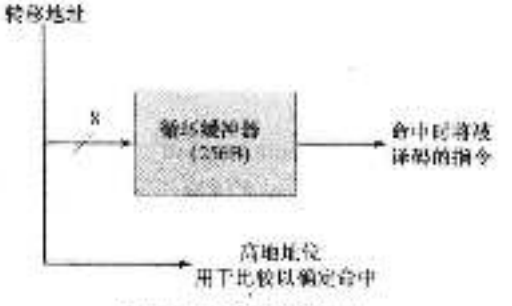


图 12.17 循环缓冲器

使用循环缓冲器的机器，有 CDC 的某些机器（Star-100、6600、7600）和 CRAY-1。Motorola 68010 使用了一种特殊形式的循环缓冲器，用于执行涉及 DBcc（递减并依条件转移）指令的 3 指令循环（见习题 12.14）。此时维持着一个 3 字缓冲器，处理器重复执行这些指令直到满足循环结束条件。

4. 分支预测

已有几种不同技术用于预测转移是否将发生。其中应用最普遍的是：

- 预测绝不发生（predict never taken）
- 预测总是发生（predict always taken）
- 依操作码预测（predict by opcode）
- 发生/不发生切换（taken/not taken switch）
- 转移历史表（branch history table）



前三种方法是静态的，它们不取决于条件转移指令的过去执行历史。后两种方法是动态的，它们取决于执行的历史。

前两种方法最简单，它们或者总是假定转移不发生而继续顺序取指令，或者总是假定转移发生而从转移目标处取指令。其中预测绝不发生转移时所有分支预测方法中最广泛使用的方法。

分析程序行为的研究已说明，条件分支的转移发生概率高于 50%［LILJ8］。于是，若由每条路径预取的代价都是相同的，那么总是由转移目标地址的预取，应当比总是由顺序路径预取能给出更好的性能。然而，在一个分页的机器中，由转移目标的预取要比顺序预取下一条指令更可能引起缺页，故必须考虑到这种性能的损害。可使用一种规避机制来减少这种损失。

最后一种静态方法是依据转移指令的操作码进行判定。处理器假定，对某些条件分支指令的操作码将总是发生转移，对另外的总是不发生转移。［LILJ88］报告这种策略的成功概率大于 75%。

动态分支预测策略，试图通过记录条件分支指令在程序中的历史来改善预测的准确度。例如，每个条件分支指令可有与之相关联的一位或几位，它们反映此指令的最近历史。这些位称为发生/不发生切换，它们指挥处理器在下次遇到此指令时产生具体的判定。一般，这些位不是保存在主存中，而是保存在一个暂时的高速存储装置中。一种可能是把这些位与对应的已在高速缓存（cache）中的条件分支指令相关联，当这种指令由 cache 替换出时其历史位也相应丢失。另一种可能是维护一个小型表，其中每个表项有最近执行的转移指令及其一位或几位相关位。处理器能像对待 cache 一样地访问这种关联表，或者通过使用分支指令地址的低序位来访问表。

以单个位所能记录的，只是这条指令最后一次执行是否发生转移。这种单个位方法的不足，表现在像循环指令这样的几乎总是发生转移的条件分支指令使用时。对一个历史位，预测错误将出现两次，一次是进入循环时，一次是退出循环时。

若使用两位，则它们能用来记录相关指令的最后两次执行情况或是记录某种其他样式的状态。图 12-18 表示了一种典型的方法（另一种可能的样式见习题 12.5）。假定算法由流程图的左上角开始。对一给定的条件分支指令，只要连续两次遇到的都是发生转移，则算法预测下一次转移要发生。如果一次预测失误，算法继续预测下一次转移将发生。仅当连续两次都不发生转移，算法才走到流程图的右边部分。接下来预测转移不发生，直到连续两次发生转移。于是，算法是连续两次失误才更改预测判定。

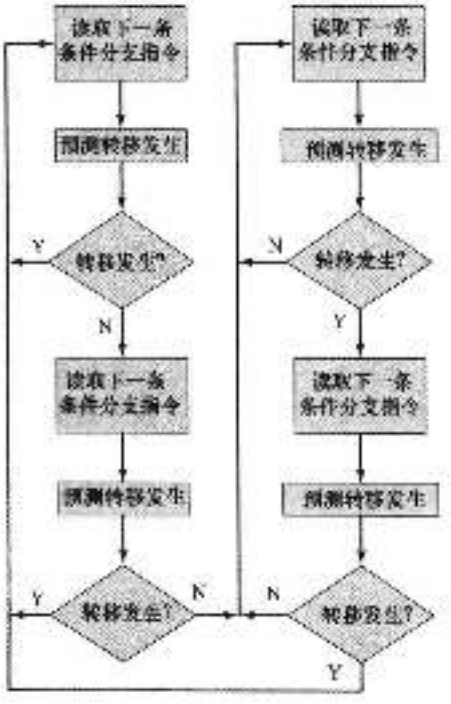


图 12-18　分支预测流程图

判定过程用有限状态机法能表示得更紧凑，如图 12-19 所示。有限状态机表示法为正式文献普遍采用。

使用刚才描述的历史位方法有一个缺点：若判定转移发生，转移目标指令并不能马上取得，直到作为条件分支指令操作数的目标地址被解析后才能取到。若判定一经做出，就能立刻开始取指令，可实现更高的效率。为此，必须保存更多的信息于一种称为分支目标缓冲器（branch target buffer）或分支历史表（branch history table）的结构中。

转移历史表是一个与流水线取指令阶段相关联的小型高速缓冲存储器。每个表项由三个元素组成：分支指令的地址，记录这条指令转移状况的历史位，有关它的目标指令的信息。在多数的建议和实现中，表项第三个字段保存的是目标指令的地址。另一种可能是直接保存目标指令。这里的权衡考虑是清楚的：保存目标地址可使表的规模较小，但与保存目标指令相比却要花费较多的取指令时间［RECH98］。

图 12-20 将这种策略与预测绝不发生策略做了对照。对后一种策略，取指阶段总是由顺序下一地址取指令。若一个转移发生，处理器中的某种逻辑就检测到这个事件发生，于是指挥由目标地

址处取下一指令（另外还要清空流水线）。转移历史表作为 cache 对待，每次预取触发一次转移历史表中的查找。若未发现匹配，则下一顺序地址用于取指。若发现匹配，依据指令状态进行预测；或是下一顺序地址，或是转移目标地址，将被送给选择逻辑。

当分支指令执行时，执行阶段将其结果通知转移历史表的控制逻辑。指令状态被修改以反映正确或不正确的预测。若预测不正确，则选择逻辑重定向到正确地址以便下次取指。当碰到的一个条件分支指令不在表中时，则它被添加到表中，而一个现有项被删除。这里会使用第 4 章讨论过的某种 cache 替换算法。

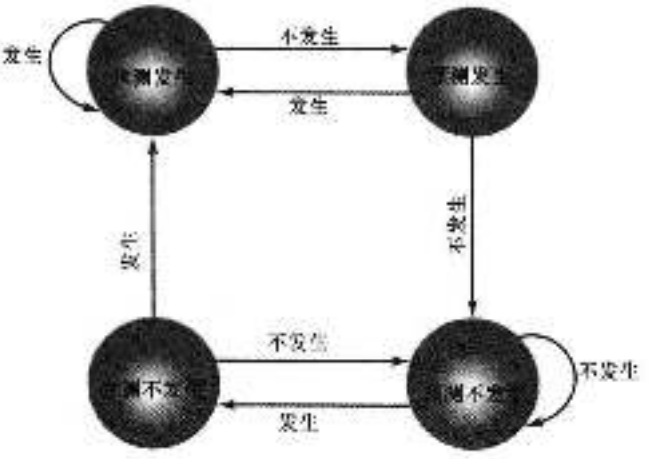


图 12-19 转移预测状态图

转移历史方法的一个改进版本是称为两级（two-level）或基于关联（correlation-based）的转移历史方法 [YEH91]。这种方法基于如下假设：在循环结束处的分支，一个特定分支指令的过去历史，将是其未来行为的很好预测器。对于更为复杂的控制流结构，分支的方向经常与相关的分支指令方向有关。一个例子是 if-then-else 或 case 结构。可能采取的策略很多。一种典型的做法是，联合使用全局转移历史（即最近执行的分支指令的历史，而不仅仅是当前执行的分支指令历史）与当前分支指令的历史。通常的结构是定义为一个 (m, n) 关联预测器。该预测器使用最近 m 个分支指令的行为，从 2^m 个 n 位分支预测器中选择一个，作为当前分支指令的预测器。换句话说，对于一个给定的分支指令，最近 m 条分支指令可能发生的每个组合都保留了一个 n 位历史记录。

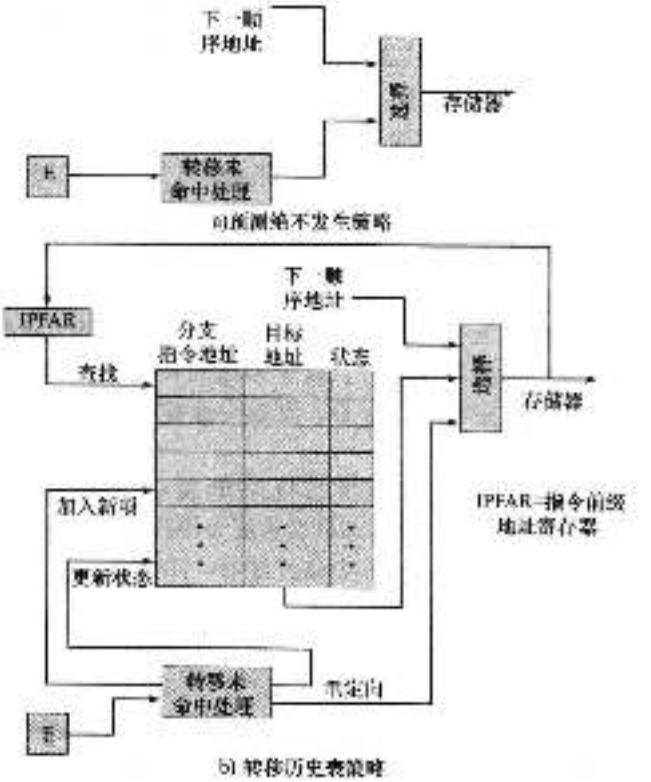


图 12-20 处理分支指令

5. 延迟分支

改进流水性能的另一可能方法是自动重排程序中的指令，这样可以把一条分支指令移到实际所期望的位置之后。这种方法将在第 13 章讨论。

12.4.5 Intel 80486 的流水线

一个有指导意义的指令流水线例子是 Intel 80486 处理器的指令流水线。80486 处理器实现了一个 5 段流水线，如下所示：

- **取指**（fetch）：指令由 cache 或外部存储器取来，并被放入两个 16 字节预取缓冲器中的某一个。取指阶段的目标是，只要 BIU 的数据被指令译码器用掉，立即以新数据填充预取缓冲器。因为指令是可变长的（不计前缀，1 至 11 字节），故预取器的状态相对于流水线的其他阶段是由指令到指令变动的。平均而言，每个 16 字节的缓冲器大约装入 5 条指令[CRAW90]。取指阶段的操作独立于其他阶段以保持预取缓冲器的满载。
- **译码阶段 1**（decode stage 1，D1）：所有的操作码和寻址方式信息在 D1 阶段被译码，所要求的信息以及指令长度信息最多也只占据指令的前 3 个字节。于是，3 字节由预取缓冲器传送到 D1 阶段，D1 译码器则能指挥 D2 阶段计算其余的信息（偏移量和立即数）。这些是 D1 译码所不涉及的。
- **译码阶段 2**（decode stage 2，D2）：D2 阶段将每个操作码扩展成对 ALU 的控制信号。它还控制史复杂寻址方式的计算。
- **执行**（execute，EX）：这一阶段包括 ALU 运算，cache 访问和寄存器修改。
- **写回**（write back，WB）：如果需要，这一阶段更改寄存器和在前面执行阶段修改过的状态标志。若当前指令要更改存储器，则计算结果值同时被送到 cache 和总线接口的写缓冲中。

通过采用两个译码段，该流水线能支持接近每时钟周期一条指令的吞吐率。复杂指令和条件分支指令会降低这个吞吐率。

图 12-21 显示了此流水线操作的例子。图 12-21a 表示，当要求存储器访问时，并没有延时引入到流水线中，然而，正如图 12-21b 表示的那样，对于一个用于计算存储器地址的值，可能引入延时。也就是说，若一个值由存储器装入寄存器，而后个寄存器又要用作下一条指令的基址寄存器，则处理器要停顿一个时钟周期。在此例中，第一条指令的 EX 阶段处理器访问 cache，然后在 WB 阶段将所取得的值存入寄存器。然而，下一条指令在它的 D2 阶段就需要这个寄存器的值。当它的 D2 阶段与前一指令的 WB 阶段对齐时，一个旁路信号通路允许 D2 阶段去访问正被 WB 阶段所使用的同一数据，这样节省了一个流水段时间。



图 12-21　80486 的指令流水线示例

图 12-21c 说明了一条分支指令的时序。假定转移发生，比较指令在其 WB 阶段更新条件码，一个旁路信号通路允许分支指令的 EX 阶段能同时使用这些新条件码。在分支指令的 EX 阶段，处理器并行地运行一个猜测取指周期来读取转移目标指令。若处理器确认转移条件不成立，它将作废这个预取来的转移目标指令，并继续执行下一顺序指令（已被另一路预取缓冲器预先取来，并被译码）。

12.5　x86 系列处理器

x86 的组成结构随着时间不断发生显著变化。本节我们考察最新 x86 系列处理器组成的一些细节，并集中介绍各个处理器的共同结构要素。第 14 章会考察 x86 系列处理器超标量方面的特点。第 18 章考察多核组成结构。图 4-18 曾给出 Pentium 4 处理器组织结构的概要。

12.5.1 寄存器组成

x86 系列处理器的寄存器组织包括如下类型的寄存器（见表 12-2）。

- **通用寄存器**（general）：x86 有 8 个 32 位通用寄存器（见图 12-3c）。它们可由所有类型的 x86 指令使用，也可用来保存用于地址计算的操作数。此外，某些寄存器亦服务于专门目的。例如，字符串指令使用 ECX、ESI 和 EDI 寄存器的内容作为操作数，无需指令中有对这些寄存器的显式引用。这样做的结果是，某些指令能更紧凑地编码。在 64 位模式中，有 16 个 64 位的通用寄存器。
- **段寄存器**（segment）：x86 有 6 个 16 位段寄存器，正如第 8 章所讨论的，它们容纳索引段表的段选择符。代码段（CS）寄存器引用含有正被执行指令的段。栈段（SS）寄存器引用含有用户可见栈的段。其余的段寄存器（DS、ES、FS、GS）允许用户一次可访问多达 4 个分开的数据段。
- **标志寄存器**（flag）：32 位的 EFLAGS 寄存器容纳条件码和各种模式位。在 64 位模式中，该寄存器被扩展到 64 位，并被称为 RFLAGS 寄存器。在当前的 x86 体系结构定义中，RFLAGS 寄存器的高 32 位未被使用。
- **指令指针寄存器**（instruction pointer）：存有当前指令的地址。

表 12-2 x86 处理器的寄存器

a）32 位模式的整数单元

类型	数目	长度（位）	目的
通用	8	32	通用用户寄存器
段	6	16	含有段选择符
标志（EFLAGS）	1	32	状态和控制位
指令指针	1	32	指令指针

b）64 位模式的整数单元

类型	数目	长度（位）	目的
通用	16	32	通用用户寄存器
段	6	16	含有段选择符
标志（RFLAGS）	1	64	状态和控制位
指令指针	1	64	指令指针

c）浮点单元

类型	数目	长度（位）	目的
数值	8	80	保持浮点数
控制	1	16	控制位
状态	1	16	状态位
标记字	1	16	标示数值寄存器的内容
指令指针	1	48	指向被异常中断的数据
数据指针	1	48	指向被异常中断的数据

还有一些寄存器专门用于浮点单元：

- **数值寄存器**（numeric）：每个寄存器保存一个扩展精度的 80 位浮点数。这样的寄存器有 8 个，它们的功能像一个栈，指令集中有相应的压入和弹出指令可用于对它们进行操作。
- **控制寄存器**（control）：16 位的控制寄存器含有控制浮点单元操作的控制位，包括舍入类型控制，单、双或扩展精度控制，以及禁止或允许各种异常条件的位。
- **状态寄存器**（status）：16 位状态寄存器包含反映浮点单元当前状态的位，包括一个指向

数值寄存器栈顶的 3 位指针、报告最后运算结果的条件码，以及异常标志。

- 标记字寄存器（tag word）：这个 16 位寄存器为 8 个浮点数值寄存器每个保留 2 位标记，它们用于指示相应寄存器内容的属性，四种可能值是有效、零、特殊数（NaN、无穷，非规格化）和空。这些标记允许程序在不对寄存器中的实际数据进行复杂解析的情况下，检查数值寄存器中的内容。例如，进行现场切换时，处理器不需要保存那些标记为"空"的浮点寄存器。

上述寄存器的大多数是很好理解的，下面简要地对几个寄存器做进一步说明。

1. EFLAGS 寄存器

EFLAGS 寄存器（参见图 12-22）指出处理器的状态，并帮助控制处理器的操作。它包括表 10-9 所定义的 6 个条件码（进位、奇偶、辅助进位、零、符号、溢出）。它们报告一次整数运算的结果。另外，此寄存器还有一些位可看作是控制位，它们是：

- 自陷标志（trap flag）：当此 TF 位置位时，每条指令执行后都引起一个中断，这可用于调试。
- 中断允许标志（interrupt enable flag）：当此 IF 位置位时，处理器将响应外部中断。
- 方向标志（direction flag）：此 DF 位确定串行处理指令是递增还是递减 16 位寄存器 SI 和 DI（16 位操作）或 32 位寄存器 ESI 和 EDI（32 位操作）。
- I/O 特权级（I/O privilege flag）：当此 IOPL 位置位时，保护模式期间所有对 I/O 设备的访问都将引起处理器产生一个异常。
- 重新开始标志（resume flag）：此 RF 位允许程序员禁止调试异常，这样在一次调试异常之后指令能重新开始，不会立即又引起另一次调试异常。
- 对齐检查（alignment check）：若一个字或双字被寻址在一个非字或非双字边界上，此 AC 位将被激活。
- 标识标志（identification flag）：若此 ID 位能置位和复位，则表明这个处理器支持处理器 ID 指令。这条指令能提供处理器的有关厂商、系列、型号等信息。



图 12-22　Pentium II EFLAGS 寄存器

此外，还有 4 位标志涉及操作模式。嵌套任务（NT）标志指示，保护模式下的当前任务嵌套在另一任务中。虚拟模式（VM）位使程序员能允许或禁止虚拟 8086 模式，在这种模式下处理器作为一个 8086 机器来运行。虚拟中断标志（VIF）和虚拟中断未决标志（VIP）用于多任务环境。

2. 控制寄存器

x86 使用了 4 个 32 位控制寄存器（寄存器 CR1 不使用）来控制处理器操作的各个方面（参见图 12-23）。除了 CR0 寄存器之外，其余寄存器都可以是 32 位或 64 位宽度，具体位宽根据后台支持 x86 的 64 位体系结构而定。CR0 寄存器包含系统控制标志，这些标志通常是控制处理器工作模式或指示其工作状态，而不是控制个别任务执行的。标志包括：

- **保护模式使能**（protection enable）：此 PE 位允许/禁止保护工作模式。
- **监控协处理器**（monitor coprocessor）：只有在早期的 x86 机器上运行程序时才需要关注此 MP 位。该位与数字协处理器的使用有关。
- **模拟**（emulation）：当处理器不具有一个浮点单元时，此 EM 位置位，试图执行一条浮点指令时将引起一个中断。
- **任务切换**（task switched）：此 TS 位指出处理器具有切换的任务。
- **扩展类型**（extension type）：不在 Pentium 及后续机器上使用，在早先的机器上此 ET 位用于指示对数字协处理器指令的支持。
- **数值错误**（numeric error）：此 NE 位允许在外部数据总线上报告浮点错误的标准机制。
- **写保护**（write protect）：当此 WP 位被清零时，一个监管进程（supervisor process）可向用户级只读页写入。在某些操作系统中，此特征可用于支持进程生成。
- **对齐屏蔽**（alignment mask）：此 AM 位允许/禁止对齐检查。
- **非写直通**（not write through）：此 NW 位选择数据 cache 的操作模式，当它置位时，禁止数据 cache 的写直通操作。
- **高速缓存禁止**（cache disable）：此 CD 位允许/禁止使用内部 cache 填充机制。
- **分页**（paging）：此 PG 位允许/禁止分页。

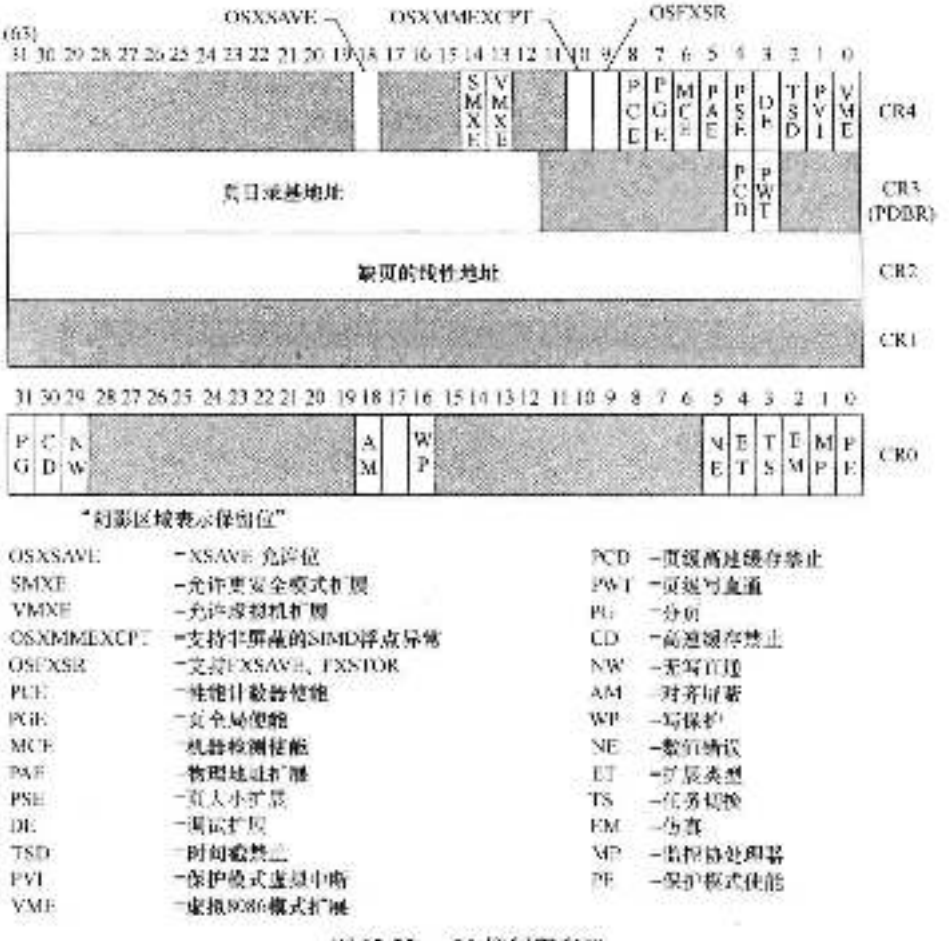


图 12-23 x86 控制寄存器

当允许分页时，CR2 和 CR3 寄存器有效。CR2 寄存器保存缺页中断之前最后访问页的 32 位

线性地址。CR3 最左 20 位保存页目录基地址的有效的高 20 位（低位为 0 不需保存）。CR3 的两位用于驱动控制外部 cache 操作的引脚。页级高速缓存禁止（PCD）位允许或禁止外部 cache。页级写直通（PWT）位控制外部 cache 的写直通。

CR4 中定义了 9 个附加控制位：

- **虚拟 8086 模式扩展**（virtual-8086 mode extension）：此 VME 标志允许在虚拟 8086 模式中支持虚拟中断标志。
- **保护模式虚拟中断**（protected-mode virtual interrupt）：此 PVI 位允许在保护模式中支持虚拟中断标志。
- **时间戳禁止**（time stamp disable）：此 TSD 位用于禁止读时间戳计数器（RDTSC）指令。这用于调试目的。
- **调试扩展**（debugging extension）：此 DE 位用于允许 I/O 断点，这允许处理器中断 I/O 的读和写。
- **页大小扩展**（page size extension）：此 PSE 位被置位时，允许使用更大的页（2MB 或 4MB 页）。清零时只允许使用 4KB 的页。
- **物理地址扩展**（physical address extension）：此 PAE 位用于控制一种新的寻址模式。每当该模式被允许时，地址线 A35 到 A32 将允许被使用。
- **机器检查使能**（machine check enable）：此 MCE 位允许机器检查中断。当读总线周期出现奇偶校验错误，或当一个总线周期未能成功完成时，会出现此种中断。
- **页全局使能**（page global enable）：此 PGE 位用于允许使用全局页。当 PGE = 1 及任务切换时，所有的 TLB 表项要被清除，但那些标志为全局（G 位 = 1）页的 TLB 项被保留。
- **性能计数器使能**（performance counter enable）：此 PCE 位用于允许以任何特权级执行 RDPMC（读性能计数器）指令。两个性能计数器用于测量指定类型事件的持续时间和出现次数。

3. MMX 寄存器

回顾 10.3 节，我们知道 x86 的 MMX 功能利用了几种 64 位数据类型。MMX 指令使用 3 位寄存器地址字段，故可以支持 8 个 MMX 寄存器的使用。实际上，处理器并未专门设置专用的 MMX 寄存器，而是使用了一种别名技术（aliasing technique），如图 12-24 所示，利用现有的浮点寄存器来保存 MMX 值。更准确地说，各浮点寄存器的低 64 位（尾数）用来构成 8 个 MMX 寄存器。于是，现有的 32 位 x86 体系结构很容易扩展成支持 MMX 功能。MMX 使用这些寄存器的关键特征如下所示。

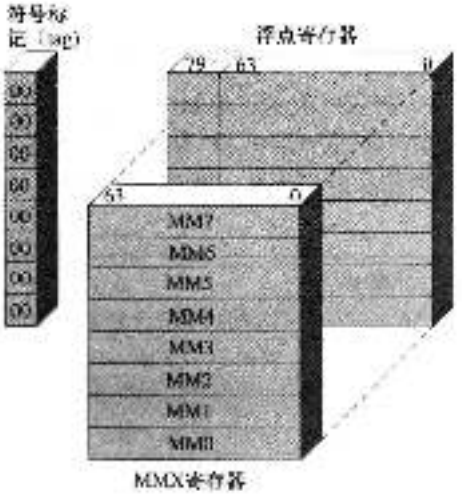


图 12-24　MMX 寄存器映射到浮点寄存器

- 回想一下，浮点寄存器在浮点运算中是作为栈来对待的。但对于 MMX 操作，同样这些寄存器却是直接可寻址的。
- 在任何浮点运算之后最初执行 MMX 指令时，FP 的标记（tag）字段是有效的。这反映了由栈操作到直接寄存器寻址的改变。
- EMMS（清除 MMX 状态）指令设置 FP 标记字段的各位，令其指示所有寄存器都是空的。程序员在一个 MMX 代码块结束时插入这条指令是至关重要的，这样才能使后面的浮点运算正常运行。

- 当一个值写入 MMX 寄存器时，FP 寄存器的［79：64］位（符号和阶值位）被置为全 1。这种设置使得以浮点数来看待这些寄存器时，这些 FP 寄存器的值是无穷大或 NaN（非数）。这保证了任何 MMX 数据值都不会被误认为是有效的浮点值。

12.5.2 中断处理

处理器中的中断处理是为支持操作系统提供的一种便利。它允许一个应用程序被挂起，以使各种中断事件能及时得到处理，然后再恢复应用程序的运行。

1. 中断和异常

有两类事件能引起 x86 挂起当前指令流的执行并响应事件：中断（interrupt）和异常（exception）。在这两种情况下处理器都要保存当前进程的上下文，并将转至一个预先定义的子程序来执行特殊的服务。中断通常是由硬件信号产生的，并出现在程序执行期间内的任何时刻。异常是由软件产生的，由执行指令所引发。有两类中断源和两类异常源。

（1）中断

- **可屏蔽中断**（maskable interrupt）：由处理器的 INTR 引脚接收此信号。除非中断允许标志（IF）被置位，否则处理器不响应可屏蔽中断。
- **不可屏蔽中断**（nonmaskable interrupt）：由处理器的 NMI 引脚接收其信号。这类中断的响应不能被阻止。

（2）异常

- **处理器检测的异常**（processor-detected exceptions）：当试图执行一条指令而处理器遇到一个错误时此异常发生。
- **程序异常**（programmed exceptions）：有一些指令（INTO、INT3、INT 和 BOUND）能产生异常。

2. 中断向量表

x86 的中断处理使用了中断向量表（interrupt vector table）。每一类中断都被指派了一个中断号，此号用于对中断向量表的索引。该表包含有 256 个 32 位中断向量，它们存储着中断服务程序的地址（段基址和偏移量）。

表 12-3 表示了中断向量号的指派情况，有阴影的项表示是中断，无阴影的项是异常。NMI 硬件中断是类型 2。INTR 硬件中断号的范围在 32-255。当一个 INTR 中断产生时，在总线上需要同时传送对应于此中断的中断向量号。其余的中断向量号用于异常。

表 12-3 x86 的异常和中断向量表

向量号	说明
0	除法错：除法上溢或被零除
1	调试异常：包括与调试有关的各种自陷和故障
2	NMI 引脚引发的中断：信号送至 NMI 引脚
3	断点：由 INT3 指令引起的，它是一条用于调试的 1 字节指令
4	INTO 检测到的上溢：处理器执行 INTO 的时若 OF 标志置位将发生
5	BOUND 范围超出：BOUND 指令比较一个寄存器的值与保存于存储器的边界值，若寄存器的值不在边界指定的范围内则产生一个异常
6	未定义的操作码
7	设备不可用：由于外部设备不存在，试图使用 ESC 或 WAIT 指令而失败
8	双重故障：在同一指令期间出现两个中断而且不能串行处理
9	保留
10	无效任务状态段：指定一个将求任务的段地址未被初始化或是无效

（续）

向量号	说明
11	段不存在：要求的段不存在
12	栈故障：超过了栈段的界限或者栈段不存在
13	常规保护：不引起其他异常的保护违约（如向只读段的写）
14	缺页
15	保留
16	浮点错：由浮点算术指令产生
17	对齐检查，以一个奇数字节地址存取一个字或以一个非 4 倍字节地址存取一个双字
18	机器检查，型号说明
19～31	保留
32～255	用户中断向量：当 INTR 信号激活时响应

注：不带阴影的是异常；右阴影的是中断。

若不止一个异常或中断是悬而未决的，则处理器以一个预先指定的顺序为它们服务。向量号在表中的位置不反映它们的优先级，异常和中断的优先级分为 5 类。以优先级降序排列的这 5 类优先级是：

- 类 1：先前指令上的中断（向量号 1）；
- 类 2：外部中断（2，32～255）；
- 类 3：取下一指令的故障（3，14）；
- 类 4：下一指令的译码故障（6，7）；
- 类 5：执行指令的故障（0，4，5，8，10～14，16，17）。

3. 处理中断

正如使用 CALL 指令的转移执行流程一样，一个到中断处理子程序的控制转移也使用系统栈保存处理器的状态。当一个中断出现并被处理器响应时，如下事件序列发生：

（1）若转移涉及特权级改变，则当前栈段寄存器和当前扩展的栈指针（ESP）寄存器的内容被压入栈。

（2）EFLAGS 寄存器的当前值被压入栈。

（3）中断（IF）和自陷（TF）两个标志被清除。这就禁止了 INTR 中断、自陷或单步中断。

（4）当前代码段（CS）寄存器和当前指令指针（IP 或 EIP）寄存器的内容被压入栈。

（5）若中断伴随有错误代码，则错误代码也要压入栈。

（6）读取中断向量表对应项的内容，将其装入 CS 和 IP（或 EIP）寄存器。控制转移到中断服务子程序继续执行。

为从中断返回，中断服务子程序执行一条 IRET 指令。这使得所有保存在栈上的值被取回，并由中断点恢复执行。

12.6　ARM 处理器

本节我们考察 ARM 处理器的组成和体系结构中一些关键的要素。我们把其中较复杂的内容以及流水线组织安排到第 14 章进行讨论。对于本节和第 14 章的讨论，记住 ARM 体系结构的关键特征是有好处的。ARM 处理器是一个 RISC 处理器，并有如下值得注意的特征：

- 中等规模、结构规整的寄存器组。寄存器数量比一些 CISC 机器多，但少于多数的 RISC 机器。
- 数据处理遵循装载/保存模式。其中各种运算只对寄存器中的操作数进行操作，而不直接访问内存。运算前，所需要的所有数据要先从内存装载到寄存器。运算结果可以继续被

后继运算使用，或保存回存储器。

- 定长的，格式统一的32位标准指令集，以及一个16位的压缩指令集。
- 为使每条数据处理指令更为灵活，可以对一个源操作数进行移位或循环移位的预处理，为有效支持这一功能，设计了单独的算术逻辑单元和移位单元。
- 只提供了少数几种寻址模式，应用于所有装载/保存地址的确定。这些地址由指令中立即数或指定的寄存器操作数来计算得到。间接寻址，以及变址寻址时要用到内存中的值，故不被使用。
- 使用了自动递增和自动递减寻址模式，以便提高程序中循环的操作性能。
- 所有指令的执行都可带条件，这降低了条件分支指令的使用，从而减少了流水线清空，提高了流水线的效率。

12.6.1 处理器组成

ARM处理器的组织随着实现的不同版本之间差别较大，尤其当ARM处理器的实现是基于不同版本的ARM体系结构的时候。不过，对于本节的讨论，一个简单的一般化的ARM组织结构还是有用的，该组织结构如图12-25所示。图中，带箭头的线表示数据流。每个方框表示一个硬件功能单元或一个存储单元。

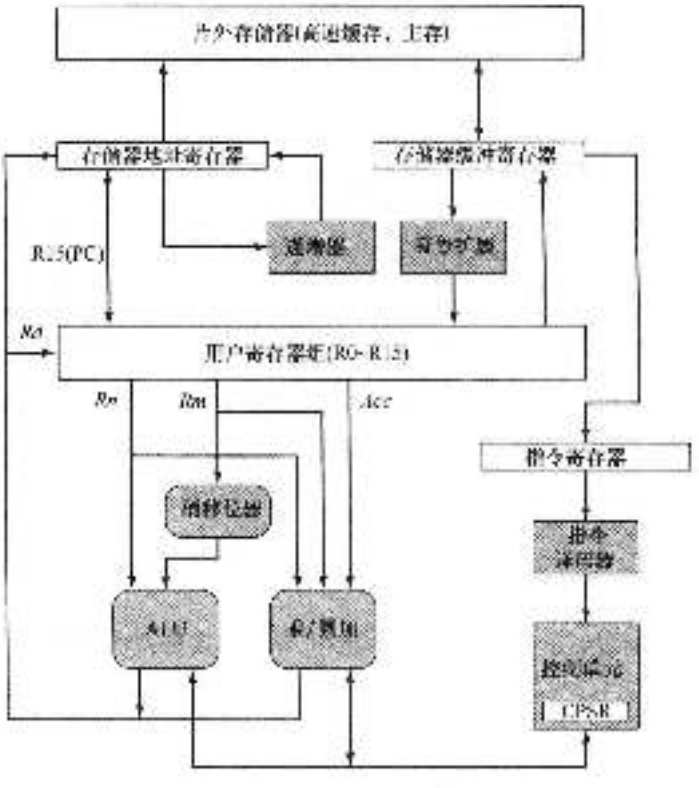


图12-25 简化的ARM处理器组织

数据经由数据总线在处理器外的存储器和处理器之间传递。所传送的数据元素或者是装载/保存指令要作的数据项，或者是取指的指令。读取到的指令在控制单元的控制下，经过指令译码器，然后执行。控制单元包括流水线逻辑电路，并产生控制信号（图中未显示），送到处理器各个硬件单元。读取到的数据项放入由一组32位寄存器组成的寄存器组。以2的补码表示的字节或半字数据项会通过符号扩展到32位。

ARM的数据处理指令通常有两个源寄存器（*Rn* 和 *Rm*），一个目的或结果寄存器（*Rd*）。源寄存器值被送到算术逻辑单元，或单独的乘法单元进行计算。乘法单元中有额外的寄存器，以便累加部分积。ARM处理器还包含了一个单独的硬件单元，该单元可用于对源寄存器 *Rm* 的值在送到算术逻辑单元之前，做移位或循环移位操作。移位和循环移位操作可以在指令周期中完成，这样能提高数据处理的能力和灵活性。

操作的结果会送回到目的寄存器中。装载/保存指令可能也会使用算术单元的结果来生成要装载或保存的存储器地址。

12.6.2 处理器模式

处理器只支持几种处理器模式是很常见的。例如，很多操作系统只使用两种模式：用户模式（user mode）和内核模式（kernel mode）。其中内核模式用来执行特权系统程序。ARM体系结构与上述方式不同，它提供了一个灵活的平台，以便操作系统实施不同的保护策略。

ARM处理器提供了7种运行模式。大多数应用程序在用户模式下运行。当处理器处于用户模式时，运行的程序不能访问受保护的系统资源，也不能改变模式，除非发生了异常。

其他6种运行模式称为特权模式，这些模式用于运行系统软件。定义这么多不同的特权模式有两个主要的好处：（1）操作系统可以对系统软件进行定制，以适应不同的情况；（2）特定的寄存器将用于特定的特权模式，使得上下文的切换更为便利。

异常模式可以访问所有的系统资源，并随意更改运行模式。特权模式中有5种是异常模式。当特定的异常发生时，就会进入到对应的异常模式。每种异常模式有一些专用的寄存器，它们取代了一些用户模式下的寄存器，这样做的目的是为了避免破坏在异常发生时的用户模式状态信息。异常模式如下所示：

- **监管模式**（supervisor mode）：这通常是操作系统运行的模式。当处理器碰到一条软件中断指令时，将进入这种模式。软件中断是ARM中一个调用操作系统服务的标准办法。
- **取消模式**（abort mode）：当出现内存错误时，将进入到这种模式。
- **未定义模式**（undefined mode）：当处理器试图执行一条既不被整数主处理器也不被协处理器支持的指令时，就进入到这种模式。
- **快速中断模式**（fast interrupt mode）：当处理器从指定的快速中断源接收到一个中断信号时，就进入到这种模式。快速中断服务程序是不能被中断的，但快速中断可以中断一个普通的中断服务程序。
- **中断模式**（interrupt mode）：当处理器从任何其他中断源（快速中断除外）接收到一个中断信号时，就进入到这种模式。只有快速中断可以中断一个中断服务程序。

最后一种特权模式是系统模式，任何异常都不会进入这种模式。它与用户模式使用相同的寄存器。系统模式用于运行特定的特权操作系统任务，这些任务可以被上述5种异常模式中断。

12.6.3 寄存器组成

图12-26显示了ARM处理器对用户可见的寄存器。ARM处理器总计有37个32位的处理器，分类如下：

- ARM处理器用户手册中介绍有31个通用寄存器。实际上，其中一些寄存器是有专门用途的，例如程序计数器。
- 6个程序状态寄存器。

寄存器分成若干组，组与组之间有部分重叠。当前运行模式决定哪个组的寄存器是可见的。在任何时候，有16个寄存器、1到2个程序状态寄存器是始终可见的，这样总计有17～18个软件可见的寄存器。下面给出图12-26的说明。

模式						
	特权模式					
		异常模式				
用户	系统	监管	终止	未定义	中断	快速中断
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13(SP)	R13(SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14(LR)	R14(LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

注：图中阴影表示用户或系统模式下使用的寄存器被替换为异常模式下的专用寄存器。

SP=栈指针　　CPSR=当前程序状态寄存器

LR=连接寄存器　　SPSR=已保存的程序状态寄存器

PC=程序计数器

图 12-26　ARM 寄存器组织

- 寄存器 R0 到 R7，寄存器 R15（程序计数器），以及当前程序状态寄存器（CPSR），对所有模式可见，并被所有模式共享。
- 寄存器 R8 到 R12 由除了快速中断模式以外的模式共享。快速中断模式有它自己专用的寄存器 R8_fiq 和 R12_fiq。
- 所有的异常模式都有它们自己版本的寄存器 R13 和 R14。
- 所有的异常模式都有它们自己专用的已存程序状态寄存器（SPSR）。

（1）通用寄存器：寄存器 R13 通常被用作栈指针，因此常常称为 SP。因为每个异常模式下有自己单独的 R13，因此每个异常模式都有自己专用的程序栈。R14 被用作连接寄存器，用于保存子过程的返回地址以及异常模式退出时返回的结果。寄存器 R15 是程序计数器（PC）。

（2）程序状态寄存器：CPSR 对于所有处理器模式都可访问。每个异常模式也有自己专用的 SPSR 寄存器。该寄存器用于保存异常发生时 CPSR 的值。

CPSR 的高 16 位包含了用户模式下可见的那些标志。这些标志可以影响程序的操作（见图 12-27）。下面列出了这些标志的说明：

- 条件码标志：第 10 章介绍过的 N，Z，C 和 V 标志。
- Q 标志：用于指示在一些面向 SIMD 的指令中是否发生了溢出和/或饱和（saturation）。
- J 位：表示现在在使用特殊的 8 位指令，即所谓的 Java 加速指令（Jazelle），这种指令的讨论不在本书范围内。
- GE［3:0］位：SIMD 指令使用 bits［19:16］作为运算结果各个字节或半字的大于或等于

（GE）标志。

CPSR 的低 16 位包含了系统控制标志，它们只能在处理器处于特权模式时被修改。这些标志如下所示：

- E 位：控制数据装载和保存的顺序；对于指令读取，该位被忽略。
- 中断禁止位：A 位置位时，非精确地数据取消异常是被禁止的。I 位置位时，IRQ 中断被禁止。F 位置位时，FIQ 中断被禁止。
- T 位：指示指令是否应该被当作普通的 ARM 指令，还是被当作压缩指令。
- 模式位：指示当前处理器的运行模式。

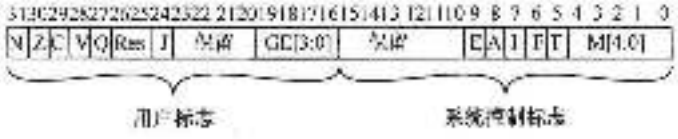


图 12-27　ARM 处理器 CPSR 和 SPSR 寄存器的格式

12.6.4　中断处理

与其他处理器类似，ARM 也提供了中断机制，允许处理器中断当前正在执行的程序，转而处理异常情况。异常由内部或外部的中断源产生，使得处理器应对某个事件。在处理异常前，处理器的状态通常要被保存，这样当异常处理完后，可以恢复执行被中断的程序。同一时间可以发生多于一个的异常。ARM 处理器支持 7 种类型的异常，表 12-4 列出了这些异常类型，以及每种异常对应的处理器运行模式。当一个异常发生时，处理器的运行被强制转向到对应于该异常类型的某个内存固定地址处开始执行。这些固定的内存地址被称为异常向量。

表 12-4　ARM 中断向量

异常类型	运行模式	通常的入口地址	说　明
重启（Reset）	监管	0x00000000	当系统被初始化时发生
数据放弃	放弃	0x00000010	当访问一个无效内存地址时发生，例如一个地址没有页面物理映射，或没有正确的访问许可
快速中断（FIQ）	快速中断	0x0000001C	当一个外部设备置位了处理器快速中断引脚时发生。一个中断处理程序一般不能被中断，除非该中断是一个快速中断。最快快速中断是为了支持数据传送或通道处理。快速中断模式提供了足够的私有寄存器，这样就不用考虑寄存器的使用，从而能使上下文切换当开销等至最低。快速中断服务程序是不能被中断的
中断（FIQ）	中断	0x00000018	当一个外部设备置位了处理器中断引脚时发生。中断服务程序不能被中断，除了快速中断以外
指令取错	放弃	0x0000000C	当试图读取一条指令，却导致内存错误时发生。该异常指令进入流水线的执行阶段时被标出
未定义指令	未定义	0x00000004	当一条不属于指令集的指令进入流水线的执行阶段时发生
软件中断	监管	0x00000008	通常用于允许用户模式的程序调用操作系统服务。在这种情况下，用户程序执行一条 SWI 指令，并带上相应参数，指出想要调用的系统服务

如果有多个中断等待处理，那么它们将按照优先级依次处理。表 12-4 就是按照异常的优先级次序从高到低排列的。

当异常发生时，处理器在执行完当前指令后，中止正在运行的程序。处理器的状态被保存到对应该异常的 SPSR 寄存器中。这样当异常处理程序执行完后，可以恢复原来程序的运行。处理器在异常发生时本来要执行的原程序的指令被保存在与该异常模式对应的连接寄存器（R14）中。异常处理完后，SPSR 寄存器的内容将移到 CPSR 中，而 R14 的内容移到程序计数器中，从

们返回原程序继续运行。

12.7 推荐的读物

[PATT01] 和 [MOSH01] 对本章所讨论的流水线问题提供了很好的概述。[HENN91] 包含了流水线的详细讨论。[SOHI90] 提供了一个有关指令流水线硬件设计出发点的详细并优秀的讨论。

[EVER01] 考察了分支预测策略的进展。[CRAG92] 是一个指令流水线中分支预测的详尽研究。[DUBE91] 和 [LILJ88] 考察了能用于提高指令流水线性能的各种分支预测策略。[KAEL91] 考察了由于转移目标地址步骤，给分支预测带来的困难。

[BREY09] 提供了 x86 中央处理的很好介绍。[FOG08b] 提供了 x86 系列流水线结构的详细讨论。

BREY09 Brey, B. *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4 and Core2 with 64-bit Extensions.* Upper Saddle River, NJ: Prentice Hall, 2009.

CRAG92 Cragon, H. *Branch Strategy Taxonomy and Performance Models.* Los Alamitos, CA: IEEE Computer Society Press, 1992.

DUBE91 Dubey, P., and Flynn, M. "Branch Strategies: Modeling and Optimization." *IEEE Transactions on Computers*, October 1991.

EVER01 Evers, M., and Yeh, T. "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors." *Proceedings of the IEEE*, November 2001.

FOG08b Fog, A. *The Microarchitecture of Intel and AMD CPUs.* Copenhagen University College of Engineering, 2008. www.agner.org/optimize/

HENN91 Hennessy, J., and Jouppi, N. "Computer Technology and Architecture: An Evolving Interaction." *Computer*, September 1991.

KAEL91 Kaeli, D., and Emma, P. "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns." *Proceedings, 18th Annual International Symposium on Computer Architecture*, May 1991.

LILJ88 Lilja, D. "Reducing the Branch Penalty in Pipelined Processors." *Computer*, July 1988.

MOSH01 Moshovos, A., and Sohi, G. "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling." *Proceedings of the IEEE*, November 2001.

PATT01 Patt, Y. "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution." *Proceedings of the IEEE*, November 2001.

RAMA77 Ramamoorthy, C. "Pipeline Architecture." *Computing Surveys*, March 1977.

SOHI90 Sohi, G. "Instruction Issue Logic for High-Performance Interruptable, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

12.8 关键词、思考题和习题

关键词

branch prediction：分支预测
condition code：条件码
delayed branch：延迟分支
flag：标志
instruction cycle：指令周期
instruction pipeline：指令流水线
instruction prefetch：指令预取
program status word：程序状态字

思考题

12.1 CPU 寄存器通常起什么作用？
12.2 用户可见寄存器普遍支持的数据类型是什么？
12.3 条件码的功能是什么？
12.4 什么是程序状态字？
12.5 与不使用流水线相比，为什么一个两阶段流水线不可能将指令周期时间缩短到原来的一半？
12.6 列出并简要说明指令流水线处理条件分支指令的几种方式。
12.7 分支预测中如何使用历史信？

习题

12.1　(a) 若在一个 8 位字的计算机上完成的最后操作是两个操作数 2 和 3 的加法，如下标志应该有何值？

- 进位　　• 符号　　• 零
- 偶校验　　• 上溢　　• 半进位

(b) 若两个操作数是 −1（2 的补码）和 +1，又应该为何值？

12.2　若 A 含有 11110000，B 含有 00010100，请对 A − B 操作重复习题 12.1 问题。

12.3　某微处理器的时钟频率是 5GHz。

(a) 时钟周期是多长？

(b) 由 3 个时钟周期组成的某特定类型的机器指令周期有多长时间？

12.4　某微处理器提供了能将字节串由内存一区域传送到另一区域的指令。取指令和指令译码用了 10 个时钟周期，此后每传送一字节用 15 个时钟周期。微处理器的时钟频率是 5GHz。

(a) 请对 64 字节的串，确定指令周期长度。

(b) 若此指令是不可中断的，那么最坏情况下中断响应的最大延迟是多少？

(c) 若此指令在每字节传送开始前能被中断，重复 (b) 问。

12.5　Intel 8088 由总线接口单元 BIU 和执行单元 EU 两部分组成，它们构成一个两段流水线。BIU 负责取指令，放入一个 4 字节指令队列，并依 EU 请求参与地址计算、由内存取操作数和写回结果。若没有待处理的这类请求，并且总线空闲，则 BIU 填充指令队列的空位置。当 EU 完成一条指令的执行，它将结果传送给 BIU（最终到存储器或 I/O），然后再处理一下指令。

(a) 假定 BIU 和 EU 完成各自的任务用相等的时间，流水线能提高 8088 性能多少倍？不考虑分支指令的影响。

(b) 假定 EU 用时是 BIU 的两倍长，重复上问。

12.6　假定 8088 正在执行的程序中跳转指令出现的概率是 0.1。为简化，认为所有指令都是 2 字节长。

(a) 多大比率的指令读取总线周期被浪费了？

(b) 若指令队列是 8 字节长，重复上问。

12.7　考虑图 12-10 的时序图。假定只是一个两阶段流水线（取指，执行）。重画此图，显示如果有 4 条指令的话，现在需要多少时间单位。

12.8　假定一流水有 4 段：取指（FI）、译码指令和地址计算（DA）、取操作数（FO）和执行（EX）。请为 7 条指令序列画出类似于图 12-10 的图，并假定此指令序列中的第 3 条指令是一条分支指令。另外，此序列不存在数据相关性。

12.9　某时钟速率为 2.5GHz 的流水式处理器执行一个有 150 万条指令的程序。流水线有 5 段并以每时钟周期 1 条的速率发射指令。不考虑分支指令和乱序（out-of-sequence）执行所带来的性能损失。

(a) 同样执行这个程序，该处理器比非流水式处理器加速了多少？此处采用与 12.4 节相同的假设。

(b) 此流水式处理器的吞吐率是多少（以 MIPS 为单位）？

12.10　某时钟频率为 2.5GHz 的非流水式处理器，其平均 CPI（每指令周期数）是 4。此处理器的升级版本引入了 5 段流水。然而，由于加锁存延迟这样的流水线内部延迟，使新版处理器的时钟频率必须降低到 2GHz。

(a) 对典型程序，新版处理器所实现的加速比是多少？

(b) 新、旧两版处理器的 MIPS 速率是多少？

12.11　考虑通过指令流水线来执行的一个长度为 n 的指令序列。假设遇到一条有条件的或无条件的分支指令的概率为 p，并假设执行分支指令 I，而转移到非后继连续地址的概率是 q。请利用这些概率重写等式（12.1）和等式（12.2）。为使问题简化，假设对于发生转移的分支指令 I，当它在流水线最后一段执行时，将清空流水线并撤销线上其他正在进行的指令。

12.12　对于处理流水线中分支指令的多个指令就方法，它的一个局限是，在最初分支指令是否转移还没确定之前又遇到另一个分支指令，请举出另外两个局限或缺点。

12.13　考虑图 12-28 所示的状态图。

(a) 描述每个状态图的行为。

(b) 将它们与 12.4 节分支预测的状态图做比较。讨论包括上述两种状态图在内的三种分支预测策略

的相对优缺点。

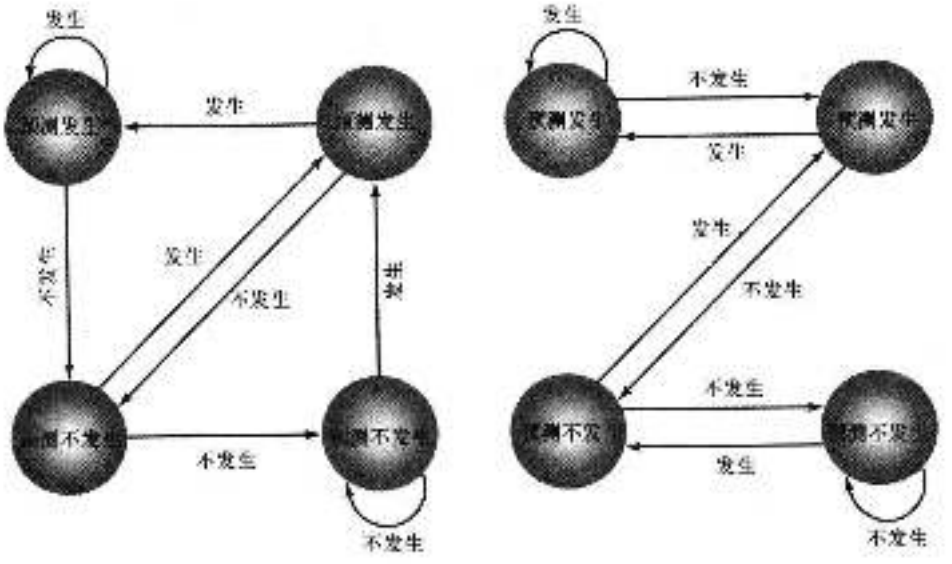


图 12-28 两种转移处理状态图

12.14 Motorola 68000 机器包括有“递减并根据条件转移”（decrement and branch according to condition）指令，它具有如下形式：

DBcc Dn, displacement

这里的 cc 是一个可测试条件，Dn 是一个通用寄存器，displacement（偏移量）则指定相对于当前指令地址的目标地址。此指令能定义成：

```
if !cc = false!
then begin
     Dn := (Dn) - 1;
     if Dn ≠ -1 then PC := (PC) + displacement  end
else PC := (PC) + 2;
```

当指令执行时，首先测试条件以确定循环结束条件是否被满足。若是，则不执行任何操作，并继续执行顺序的下一条指令。若条件是假，则指定的数据寄存器被减 1，并检查其值是否小于零。若是小于零，则循环结束，并继续执行顺序的下一条指令。否则，程序转移到指定的位置。

现考虑如下的汇编语言程序段：

```
AGAIN    CMPM.L     (A0)+, (A1)+
         DBNE       D1, AGAIN
         NOP
```

其中 A0 和 A1 是两个字串地址，代码对这两个串做比较，检查它们是否相等；每次访问了两个串中的对应元素，串指针都被递增。D1 最开始含有待比较的长字（4 字节）的数量。

(a) 寄存器的初始值是：A0 = $00004000，A1 = $00005000，D1 = $0000001F（$ 表示十六进制数）。地址 $4000 和 $6000 之间的存储器全部以 $AAAA 字装入。若运行上述程序，请指出 DBNE 循环执行的次数和当达到 NOP 指令时三个寄存器的内容。

(b) 重复 (a)，但现在是假定存储器 $4000 和 $4FFE 之间是以 $0000 字装入，而 $5000 和 $6000 之间是以 $AAAA 装入。

12.15 假定条件转移未发生，请重画图 12-19c。

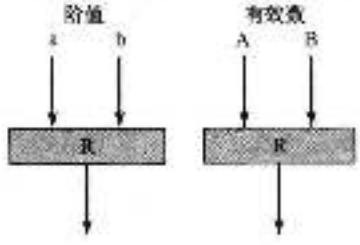
12.16 摘自 [MACD84] 的表 12-5 对各类应用的转移行为进行了统计。除 1 类转移行为外，各类应用之间没有明显不同。请确定科学应用环境中，转向转移目标地址的转移占全部转移的比率。对于商业应用和系统应用环境，重复上一问题。

表 12-5 示例应用程序中的转移行为

转移出现的类型：			
1 类：转移指令	72.5%		
2 类：循环控制	9.8%		
3 类：过程调用，返回	17.7%		
1 类转移：转向何处	**科学**	**商业**	**系统**
无条件——100% 转向目标	20%	40%	35%
条件——转向目标	43.2%	24.3%	32.5%
条件——不转向目标（顺序）	36.8%	35.7%	32.5%
2 类转移（所有环境）			
转向目标	91%		
不转向目标	9%		
3 类转移			
100% 转向目标			

12.17 流水化亦能应用到 ALU 内部以加速浮点运算。考虑浮点加减法的情况。简洁地说，流水线可以包含 4 段：(1) 比较阶值；(2) 选择阶值并对齐有效数；(3) 加或减有效数；(4) 规格化结果。假设流水有两个并行线程并能像这样着手进行：一个处理阶值，一个处理有效数。

图中标记 R 的方框指的是用于保持临时结果的一组寄存器。完善此框图，使其顶层视图表示出流水线的结构。



第13章

Computer Organization and Architecture: Designing for Performance, 8E

精简指令集计算机

本章要点

- 对设计新型处理器体系结构来说，高级语言程序行为的研究具有指导意义，成果之一就是产生了精简指令集计算机（RISC）。程序中赋值语句占最大份额，这暗示着简单的数据传送应当优化。程序中还有许多 IF 和 LOOP 语句，意味着基本的顺序控制机制需要进行优化，以便有效地使用流水技术。操作数引用特点的研究表明，在寄存器中保持适当数量的操作数会有助于性能的提高。
- 这些研究已导出 RISC 机器的关键特征：其一，有限的指令集并具有固定格式；其二，大量的寄存器或利用编译器来优化寄存器的使用；其三，强调对指令流水线的优化。
- RISC 实现简单指令集便于有效的流水化，因为每条指令只有少数几种操作，并且这些操作定义容易确定统一。RISC 指令集体系结构自身也有助于实施延迟分支（delayed branch）技术，这种技术将分支指令和其他指令重排从而提高流水线效率。

自 1950 年研制出程序存储式计算机以来，在计算机组织和体系结构领域中，只有少量引人瞩目的真正变革。以下虽然不是一个完整列表，但给出了自计算机诞生以来某些最主要的进步。

- **系列概念**（family concept）：1964 年 IBM 在它的 System/360 机器上引入此概念。接着又有 DEC 的 PDP-8 使用了此概念。系列这个概念将机器的结构与它的具体实现分离开来。以不同的价格/性能特征提供的一组计算机，对用户来说具有同样结构。它们被称为一个系列。性能和价格方面的差异在于同样结构的不同实现。
- **微程序式控制器**（microprogrammed control unit）：它是 Wilkes 于 1951 年首先提出的，并在 1964 年被 IBM 引入到它的 S/360 生产线。微程序设计使控制器的设计和实现变得更容易，并提供了对系列概念的支持。
- **高速缓存存储器**（cache memory）：商品化使用首先是在 1968 年的 IBM S/360 型号 85 机器上实现的。在存储器层次结构中插入 cache 这个层次，极大地改善了系统性能。
- **流水**（pipelining）：将并行性引入机器指令程序顺序本性的一种方式。例子是指令流水和向量处理。
- **多个处理器**（multiple processors）：这一类包含了几种不同组织和目标机器。
- **精简指令集计算机**（reduced instruction set computer，RISC）结构：这是本章的焦点。

RISC 结构是一个对 CPU 传统趋势的重大背离。RISC 结构分析将我们带入计算机组织和体系结构热点的讨论。

虽然，已有不同的团体以各种方式定义和设计了 RISC 系统，然而有些关键点是大多数（不是所有）设计都采用的。它们是：

- 通过大量的通用寄存器和（或）使用编译器技术来优化寄存器的使用。
- 一个有限且简单的指令集。
- 强调指令流水的优化。

表 13-1 比较了 RISC 和非 RISC 系统。

本章以简要综述指令集的几个研究结果开始，然后考察上述的三个主题，最后介绍 RISC 设计中的两个实例。

表 13-1 一些 CISC、RISC 和超标量处理器的特征

特征	CISC			RISC			超标量	
	IBM370/168	VAX11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
开发年份	1973	1978	1989	1987	1991	1993	1996	1996
指令数量	208	303	235	69	94	225		
指令长度(字节)	2 ~ 6	2 ~ 57	1 ~ 11	4	4	4	4	4
寻址方式	4	22	11	1	1	2	1	1
通用寄存器数	16	16	8	40 ~ 520	32	32	40 ~ 520	32
控制存储器大小(Kb)	420	480	246	—	—	—	—	—
cache 大小(KB)	64	64	8	32	128	16 ~ 32	32	64

13.1 指令执行特征

计算机发展的最易见形式是编程语言。随着硬件成本的下降，软件的成本相对上升。另一方面，编程人员的长期缺乏也驱使软件成本在绝对意义上上升。因此，一个系统生存期的主要成本是软件而不是硬件。除成本和不便利之外，还有不可靠因素：不论是系统程序还是应用程序，运行多年之后虽经不断修正仍继续出现新的故障。

研究人员和业界对此的响应是，开发出了功能更强、更复杂的高级程序设计语言。这些高级语言（High-Level Language，HLL）允许编程人员能更简明地表示算法，更关注细节，并通常支持结构化程序设计或面向对象的程序设计。

然而，这种解决方法又提出了一个称为语义差距（semantic gap）的问题，即 HLL 中提供的操作与计算机硬件结构提供的操作间的差异。这种差距被认为是执行的低效、过长的机器程序和编译器复杂性的缘由。设计者试图以结构的改进来减小这个差距。关键的做法包括大的指令集、众多的寻址方式和硬件实现的各种 HLL 语句。最后一种做法的例子是 VAX 机上的 CASE 机器指令。这种复杂指令集希望：

- 使编译器编写者的任务变得容易。
- 提高执行效率，因为复杂操作序列能以微代码实现。
- 提供更复杂更精致的 HLL 支持。

与此同时，确定 HLL 程序生成的机器指令执行的特征和样式，这样的研究已进行多年。研究结果促使设计人员寻找一种截然不同的方法：使支持 HLL 的硬件结构更简单而不是更复杂。

因此，为理解主张 RISC 的理由，我们先简单回顾一下指令执行特征。所关注的涉及计算的方面如下所示：

- **执行的操作**：这些操作确定了 CPU 及其与存储器相互作用所能完成的功能。
- **所用的操作数**：操作数类型和它们使用的频度，确定了存储它们的存储器组织和访问它们的寻址方式。
- **执行顺序**：这确定了控制和流水线的组织。

下面，我们总结几个有关高级语言研究的报告，所有这些都是动态测量结果。动态测量是通过运行程序，并统计所出现的某个特征的次数，或者某个特征为真的次数，来收集得到的。相对地，静态测量只是在源程序文本上进行统计，这不会给出很有用的性能信息，因为它们没有对每条语句的执行次数加权。

13.1.1 操作

已有不少研究者分析了 HLL 程序的行为。第 4 章讨论过，表 4-7 包括了这些研究的重要结

论。在混合的语言和应用的研究中，结论也具有相当的趋同性。赋值语句在程序中很显著，这暗示简单的数据传送非常重要。条件语句亦在程序中占有优势。这些语句（IF、LOOP）是用一些比较和分支的机器指令来实现的。这表明指令集的顺序控制机制是关键。

这些研究成果对机器指令集设计人员具有指导意义，指出了什么类型语句出现最频繁，因而应以一种“优化”形式来支持它们。然而，这些成果未揭示什么样的语句在一个典型程序的执行中占用了最大时间。也就是说，对于一个给定的编译后机器语言程序，源语言中的什么语句可能占有最多的机器语言指令执行次数。

为寻找这种潜在的规律性，研究者曾在 VAX、PDP-11 和 Motorola 68000 上编译 Patterson 程序［PATT82a］，其描述可参见附录 4A，以确定每类语句的平均机器指令数和平均存储器访问数。表 13-2 的第 2、第 3 两列是程序中各类 HLL 语句出现的相对频度。这些数据是通过观察程序运行得到的，故它们是动态频度统计。将这两列数据乘以编译器为各语句产生的机器指令数。再将乘积规范化就得到表中第 4、第 5 两列数据，这给它们是机器指令加权后各类 HLL 语句的相对出现频度。类似地，将第 2、第 3 两列数据乘以各语句引起的存储器访问相对次数，就得到第 6、第 7 两列数据。第 4～7 列数据提供了执行各类语句所花费时间度量的一种替代测量值。此结果指出，过程调用/返回是典型 HLL 程序中最耗时的操作。

读者应清楚地了解表 13-2 中的含义。该表指出当 HLL 程序被编译到典型的当代指令集结构时，HLL 中各类语句的相对分量。某些其他结构肯定会产生不同的结果。不过，这个表给出的结果是以当代复杂指令集计算机（CISC）结构为代表的。于是，它们能为寻找支持 HLL 的更有效方式提供指导。

表 13-2 HLL 操作的加权相对动态频度［PATT82a］

	动态出现频度		机器指令加权		存储器访问加权	
	Pascal	C	Pascal	C	Pascal	C
赋值语句	45%	38%	13%	13%	14%	15%
循环语句	5%	3%	42%	32%	33%	26%
调用语句	15%	12%	31%	33%	44%	45%
其他语句	29%	43%	11%	21%	7%	13%
直接转移语句		3%	—	—	—	—
其他	6%	1%	3%	1%	2%	1%

13.1.2 操作数

虽然操作数类型的出现率也是一个重要课题，但在这方面所做工作很少。操作数有几个方面是比较重要的。

前面提到过的 Patterson 研究报告［PATT82a］也查看了各类变量的动态出现频度（见表 13-3）。Pascal 和 C 程序的结论是一致的：主要使用的是简单标量变量，而且 80% 以上的标量是（过程的）局部变量。另外，访问数组/结构也要求先取得它们的索引或指针，而这些通常也是局部标量。于是，程序中大量访问的是标量，而且它们是高度局部化的。

表 13-3 操作数的动态百分比

	Pascal	C	平均
标量变量	16%	23%	20%
标量指针	58%	53%	55%
数组/结构	26%	24%	25%

Patterson 研究以独立于底层结构的方式考察了 HLL 程序的动态行为。正如前面讨论的，考

必要针对实际结构来更深入地考察程序行为。一项研究［LUND77］动态地考察了 DEC-10 指令，平均的统计数据显示，每条指令访问 0.5 个存储器操作数和 1.4 个寄存器操作数。［HUCK83］报告对于运行在 S/370、PDP-11 和 VAX 机上的 C、Pascal 和 FORTRAN 程序也有类似的结果。当然，这种状况很大程度上取决于体系结构和编译器，但它们说明了操作数存取的频度。

这些后来的研究显示，因为操作数存取如此频繁，所以采用快速存取的结构将起重要作用。Patterson 研究揭示，优化的主选方向应是对局部标量变量的存储和访问。

13.1.3　过程调用

我们已经看到，过程调用和返回是 HLL 程序的一个重要部分。表 13-2 指出，它们是编译后的 HLL 程序中最耗时的操作。于是，考虑高效地实现这些操作的方式将是有益的。其中两个方面有显著意义：过程使用的参数及变量的数量和嵌套的深度。

在 Tanenbaum 的研究［TANE78］中指出，98% 的动态调用过程中传送的参数少于 6 个，而且其中 92% 使用少于 6 个局部标量变量。Berkeley 的 RISC 小组报告了类似的结果［KATE83］，如表 13-4 所示。这些结论表明，每个过程调用所需字的数目是不多的。前面介绍的一些研究报告曾指出，操作数访问的绝大部分是对局部标量变量的访问。这些研究报告又指出，这些访问实际上只是限定在相对少数变量上。

表 13-4　过程参数和局部标量变量

带下列参数数目的过程调用所占的百分比	编译器、解释器和排版程序	小型非数值程序
>3 参数	0% ~ 7%	0% ~ 5%
>5 参数	0% ~ 3%	0%
>8 字的参数和局部标量	1% ~ 20%	0% ~ 6%
>12 字的参数和局部标量	1% ~ 6%	0% ~ 3%

Berkeley 小组也研究了 HLL 程序中过程调用和返回的样式。他们发现很少出现这种情况：一系列长的不被打断的调用后面跟着一系列相应的返回。确切地说，他们发现程序保持在相当窄的过程调用窗口区域内。这已在图 4-21 中说明过。这些成果进一步证实了操作数访问是“高度局部化”的这一结论。

13.1.4　推论

一些研究组考察了上述这些报告的结果后认为：试图让指令集结构更接近 HLL 并不是一个有效的策略。相反，通过优化典型 HLL 程序中最耗时操作的性能，能更好地支持 HLL。

由不少研究者的工作可发现，总的来说，RISC 结构特征通常体现在以下三点：首先是使用大量的寄存器，这样可以优化操作数的访问。正如前面讨论所示，每个 HLL 指令都有几次操作数访问，并且传送（赋值）语句在程序中占有很高的份额。这些再结合局部性和标量访问的主导性，表明可以通过采取更多寄存器访问的方式，来降低内存访问次数，从而提高性能。由于这些访问的局部性，一个可扩展的寄存器组看起来是符合实际的。

其次，要精心谨慎地设计指令流水线。由于条件分支和过程调用指令的高比例，一个过于简单的指令流水线将是低效的。因为它本身可能表现出大量的指令被预取但却永不执行。

最后，研究报告也指出了对简单（减少）的指令集的需求。这点目前虽不像其他问题那样明显，但在随后的讨论中变得更加明确。

13.2　大寄存器组方案的使用

13.1 节概述的研究成果指出了对操作数快速存取的要求。我们已经看到，在 HLL 程序中有

大量的赋值语句，其中多数是简单的 A← B 形式。还有，每个 HLL 语句都有一定数量的操作数访问。若再考虑到大多数访问的是局部标量，则倾向于寄存器访问应是推荐使用的。

采用寄存器的理由是，寄存器是比主存和 cache 还要快的最快可用存储装置。寄存器组从物理上讲是小的，通常是与 ALU 和控制器在同一芯片上，并且它们使用比主存和 cache 地址还要短的地址。于是，需要一种策略能使最频繁访问的操作数保持在寄存器中，并减少“寄存器-存储器”操作。

有两种基本途径可实现这个目标，一种是基于软件，另一种是基于硬件。软件方法是依赖编译器使寄存器的使用率最大化。编译器将试图把寄存器分配给那些在一给定时间期内使用最多的变量。这种方法要求使用复杂的程序分析算法。硬件方法是简单地装备更多的寄存器，使得更多的变量能更长时间地保持在寄存器中。

本节将讨论硬件方法，这种方法是 Berkley RISC 小组首先提出的［PATT82a］，并用于最初的 RISC 商业产品 Pyramid 中［RAGA83］。

13.2.1 寄存器窗口

就表面上来判断，使用一大组寄存器应能减少对存储器访问的需求。因此，设计的任务就是很好地组织寄存器来实现这个目标。

因为大多数操作数是局部标量，一种明显的方法是使用寄存器来保存它们，或许再将少量寄存器保留给全局变量。问题是这个“局部的”定义是随着每次过程调用和返回而改变的，而过程调用和返回又是频繁出现的操作。每次调用时，寄存器中的局部变量必须被送到存储器保存，以使这些寄存器能由调用程序再次使用。而且，还需要传送过程调用的参数。返回时，父程序的变量必须恢复（装载回寄存器），并且结果也要返回到父程序。

解决的方法是基于 13.1 节报告过的另外两个结论。第一，一个典型的过程只使用少数传送参数和局部变量。第二，过程调用的深度仅限定在一个相对窄的范围内（见至 4-16）。为利用这些性质，使用多个小的寄存器组，每个小组指派给一个不同的过程。过程调用时自动地切换来使用不同的但大小固定的寄存器窗口，而不再在存储器保存寄存器内容。相邻过程的窗口是（部分）重叠的，以允许参数传递。

图 13-1 说明了这个概念。任何时刻，只有一个寄存器窗口是可见和可寻址的，就像它是唯一的一组寄存器一样（例如，地址 0 至 $N-1$）。窗口分成三个固定大小区域：参数寄存器域、局部寄存器域和临时寄存器域。参数寄存器用来保存调用当前过程的过程（即父过程）向下传递的参数和将被返回的结果。局部寄存器用于局部变量，这由编译器指派。临时寄存器用于当前过程与下一级过程（被当前过程调用的过程，即子过程）交换参数和结果。某一级的临时寄存器与下一级的参数寄存器是物理同一的，这种重叠准许不用实际移动数据就能传递参数。记住，除了重叠的情况之外，两个不同级的寄存器窗口在物理上是完全不同的。也就是说，第 J 级的参数和局部寄存器与第 $J+1$ 级的局部和临时寄存器是不相交的。

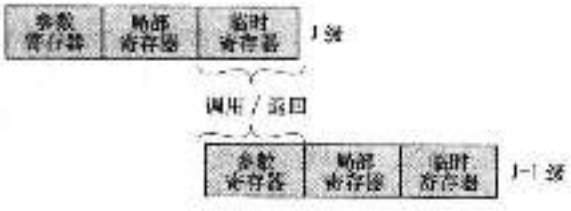


图 13-1 重叠的寄存器窗口

为管理任何可能的调用和返回的模式，寄存器窗口的数目应该是不受限制的。不过这是不可能的，替代的方法是，寄存器窗口只用于保持少数最近过程的调用。更早的过程调用必须保存到存储器中，当嵌套深度减少时再恢复。于是，寄存器组的实际组织是一个由重叠窗口组成的环形缓冲器。这种方式值得一提的例子是 Sun 的 SPARC 体系结构，13.7 节将有其描述，以及 Intel 公司 Itanium 处理器所采用的 IA-64 体系结构，第 21 章将有其描述。

图 13-2 说明了这种环形组织形式，它描述的是一个 6 窗口的环形缓冲器。缓冲器已填充到深度 4（A 调用 B，B 调用 C，C 调用 D），而过程 D 是当前活动的过程。当前窗口指针（current-window pointer，CWP）指向当前活动过程的窗口。机器指令的寄存器引用是一个对此指针的位移，以此来确定实际使用的物理寄存器。保存窗口指针（saved-window Pointer，SWP）标识最近保存在存储器的窗口。若当前过程 D 又调用过程 E，E 的初始参数放在 D 窗口的临时寄存器中（图中 w3 和 w4 的重叠部分），CWP 前进一个窗口。

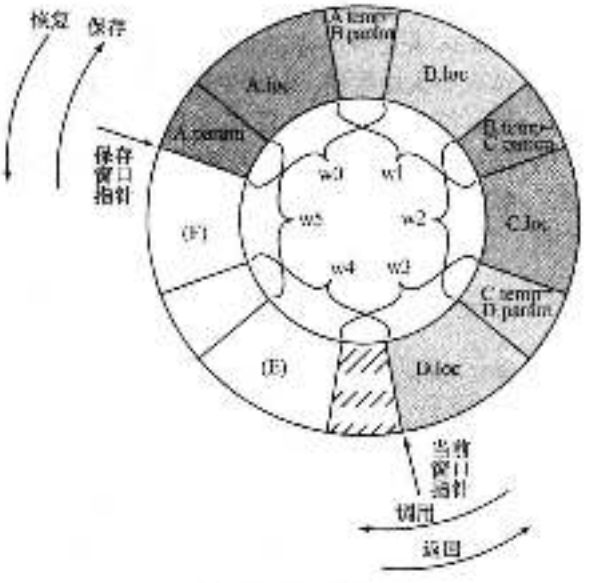


图 13-2　重叠窗口的环形缓冲组织

若过程 E 又调用过程 F，则以目前的缓冲器状况，此调用不能立即进行。这是因为 F 的窗口重叠了 A 的窗口。若 F 开始对它的临时寄存器装入数据，准备一个调用，就会改写 A 的参数寄存器（A.in）。于是，当 CWP 递增（模 6）变成等于 SWP 时，一个中断就会发生。在中断处理中，A 的窗口被保存。保存时只需要保存窗口的前两部分（A.in 和 A.loc）需要保存。然后，SWP 递增，现在可调用 F 过程了。返回时也会出现类似的中断。例如，在 F 过程完成之后逐级地返回，当 B 返回到 A 时，CWP 被递减变成等于 SWP，这将引起中断，导致 A 窗口的恢复。

由此可见，N 个窗口的寄存器组仅能用于 $N-1$ 个过程的调用。N 值不需要很大，正如前面（附录 4A）提到过的，［TAMI83］研究报告指出仅有 1% 的过程调用和返回需要 8 个窗口。Berkeley RISC 计算机使用 8 个窗口，每个窗口有 16 个寄存器。Pyramid 计算机使用 16 个窗口，每个窗口有 32 个寄存器。

13.2.2　全局变量

刚才介绍的窗口策略为在寄存器中存储局部标量变量提供了一种有效的组织形式。然而，这种策略没有解决存储全局变量的需求。全局变量由多个过程所使用，解决它有两种方法。首先，由编译器为高级程序设计语言（HLL）中声明的全局变量指派存储器位置。所有访问这些变量的机器指令将使用存储器引用的操作数。无论从硬件观点还是从软件（编译器）观点看，这种方法都是直截了当的。然而，对于频繁访问的全局变量来说，这种策略是低效的。

替代的方法是，CPU 中包括有一组全局寄存器，这些寄存器的数量是固定的并可被所有过程使用。一种统一编号的方法能用来简化指令格式。例如，寄存器引用号 0～7 指的是唯一一组全局寄存器，对寄存器 8～31 的访问指的是当前窗口内的具体寄存器。对于分立的寄存器寻址而言，这会增加硬件负担。另外，编译器也必须决定哪些全局变量应指派到全局寄存器。

13.2.3　大寄存器组与高速缓存的对比

组织成窗口的寄存器组，其作用就像一个小的快速的缓冲器，保持着可能多次使用的所有变量的一个子集。从这个意义上讲，寄存器组的作用非常像一个高速缓存存储器。于是，就引出一个问题，使用 cache 还是使用小的传统的寄存器组，哪一种更简单、更好。

表 13-5 比较了两种方法的特征。基于窗口的寄存器组保持着最近 $N-1$ 个过程调用的所有局部标量变量（除少有的窗口上溢情况之外）。cache 是有选择地保持最近使用过的标量变量。寄

存器组节省了时间，因为它保留了所有局部标量变量。而 cache 能更有效地利用空间，因为它能对动态变化的情况做出反应。而且，cache 通常是将所有的存储器调用，包括指令和其他数据类型一样地对待。于是，使用 cache 做到其他方面的节省是可能的，而寄存器组却不行。

寄存器组在空间利用方面比较低效，因为不是所有过程都使用分配给它们的全部窗口空间。另外，cache 承受另一类的低效：数据是成块读入 cache 的，而寄存器组仅容纳有用的变量。cache 读入一大块数据，但其中部分甚至更多的数据将不会使用。

cache 能处理局部变量和全局变量。通常有很多全局标量，但只有少数是频繁使用的[KATE83]。cache 将动态地发现这些变量并保持它们。若基于窗口的寄存器组补充有全局寄存器，则它也能保持某些全局标量。然而，让编译器来确定全局标量的使用频率却是一件困难的事情。

表 13-5 大寄存器组和 cache 组织的特征

大寄存器组	cache
所有局部标量	最近使用过的局部标量
各个变量	方便替换
编译器确定的全局变量	最近使用过的全局变量
保存/恢复基于过程的嵌套深度	保存/恢复基于 cache 替换算法
寄存器寻址	存储器寻址

使用寄存器组，寄存器和存储器间的数据传送由过程嵌套深度所确定。因为这个深度通常是在一个窄的范围内摆动，故存储器的使用相对不太频繁。大多数 cache 是一种组关联结构，组的容量较小。于是，存在一种危险，其他的数据或指令可能会排挤走那些要频繁使用的变量。

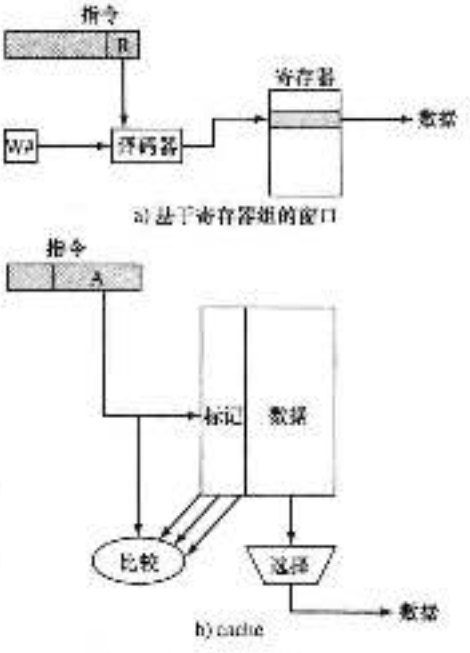


图 13-3 访问一个标量

讨论至此，在大的基于窗口的寄存器组与 cache 之间应选择谁，还不是很清楚。然而，有一个特征能说明寄存器方法占有明显优势——基于 cache 的系统是明显较慢的。这个区别在于两种方法的寻址开销总量上。

图 13-3 说明了这种区别。为访问基于窗口寄存器组中的一个局部标量，要使用一个窗口号和一个“虚拟的”寄存器号。这些通过一个相对简单的译码器来选择某一个具体的寄存器。为访问 cache 存储器中的一个位置，必须生成全宽度的地址。这种操作的复杂性取决于寻址方式。在一个组关联的 cache 中，地址的一部分用于读取数目等同于组长度的几个字和标记（tag）。地址的另一部分用于与标记进行比较，以选择所读的一个字。这一点应是很清楚的，尽管 cache 能与寄存器组一样地快，但 cache 的存取时间肯定要长。于是，从性能观点看，基于窗口的寄存器组对于局部标量而言是更优化的。通过加入专门的指令 cache，能进一步改善性能。

13.3 基于编译器的寄存器优化

我们现在假定目标 RISC 机器上只有少量寄存器可用（如 16～32 个）。这种情况下，优化寄存器的使用就是编译器的责任了。用高级语言编写的程序自然没有对寄存器的显式引用，程序中的量是以符号来表示的。编译器的目标就是，尽可能在寄存器中而不是在主存中为多数计算

保持操作数，并且减少装载和保存操作。

通常，所采取的方法如下所述。准备驻留在寄存器中的每个程序量先被指派到一个符号的或虚假的寄存器，然后编译器再将这些未限定数目的符号寄存器映射到固定数目的真实寄存器上。那些使用不重叠的符号寄存器能共享同一真实寄存器。若在程序具体运行的某个期间，需要处理的量多于真实寄存器数目，则某些量被指派到存储器位置上。装载和保存指令能把要计算的量暂时放置到寄存器中。

优化任务的本质是，判定在程序的任何给定时间点，什么样的量应指派到寄存器中。在 RISC 编译器中普遍使用一种称为图着色（graph coloring）的技术，这是由拓扑学借用过来的技术［CHAI82，CHOW86，COUT86，CHOW90］。

图着色的做法是这样的。对于一个由结点和边组成的给定图，为结点指定颜色，并使相邻结点不同色，而且要使颜色的数目最少。这个问题以如下方式转换成编译器问题。首先，分析程序并构成一个寄存器相关图。图的结点是符号寄存器，若两个符号寄存器同时“生存”于同一程序段，则相应的两个结点用一条边连接起来以指示它们相关。尝试用 n 种颜色给图上色。这里的 n 是真实寄存器的数目。若这个过程不能完全成功，那么这些不能上色的结点必须放入存储器中，并且当需要它们时，必须使用装载和保存操作给它们开辟寄存器空间。

图 13-4 是这种做法的一个简单例子。假定程序有 6 个符号寄存器将被编译到 3 个实际寄存器上。图 13-4a 表示每个符号寄存器有效使用的时间顺序，图 13-4b 表示寄存器相关图。这里指出了一种使用三种颜色给图着色的可能方法。一个符号寄存器 F 未能上色，必须使用装载和保存操作来处理。

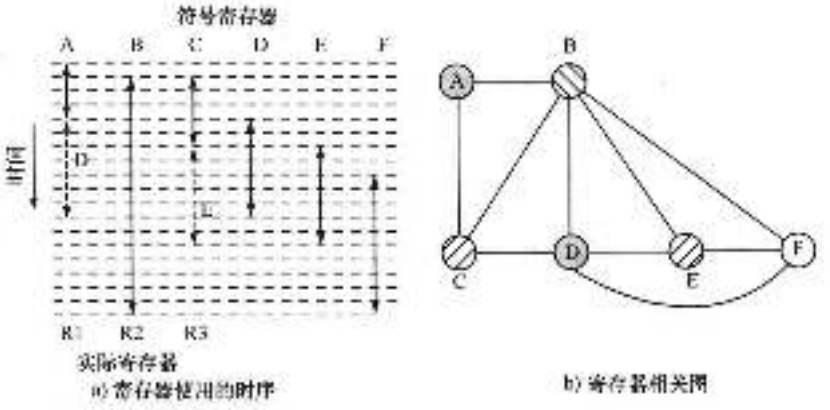


图 13-4　图着色法

通常，在使用大量的寄存器和基于编译器的寄存器优化之间有一个权衡考虑的问题。例如，［BRA1991a］是在一个具有类似于 Motorola 88000 和 MIPS R2000 特色的 RISC 结构的模型机上所做的研究报告。他们选取了不同的寄存器的数目（从 16 ~ 128），既考虑到所有寄存器都作为通用寄存器使用，也考虑到将寄存器分成整数寄存器和浮点寄存器的使用。他们的研究表明，若只有相当简单的寄存器优化，那么使用多于 64 个寄存器几乎不带来好处。使用相当精致的寄存器优化技术，当使用多于 32 个寄存器时，带来的性能改善也是不明显的。最后，他们指出，具有少量的寄存器（如 16 个）时，具有共享寄存器组织的机器要比具有分立寄存器组织的机器执行得更快。类似的结论由［HUGU91］得出，他们的研究主要关心的是使用少量寄存器的优化问题，而不是将大量寄存器的使用与优化来做比较。

13.4　精简指令集体系结构

本节将考察精简指令集体系结构的一般特征及其发展的原因。本章稍后将给出具体的例子。首先，我们讨论当代复杂指令集结构的发展原因。

13.4.1　采用 CISC 的理由

我们已指出过朝更丰富指令集发展的趋势，这包括更大数量的指令和更复杂的指令。推动

这一趋势的两个基本理由是：要求简化编译器和改善性能。这两个理由之下的根本性原因是，大部分程序员已转移到高级语言（HLL）上，厂家试图设计能对 HLL 提供更好支持的机器。

本章并不是说 CISC 设计人员选错了方向。的确，技术还在发展，处理器结构存在着广泛类型而不是两种纯类型，因此想做一个黑白分明的评定是不太可能的。于是，下面的解释只是指出 CISC 方法的某些潜在缺陷，并提供对 RISC 发展动因的一些理解。

采用 CISC 的第一个理由——简化编译器，看起来是很明了的。编译器的任务就是为每个 HLL 语句产生一个机器指令序列。若有类似于 HLL 语句的机器指令，那任务就简单多了。但 RISC 研究者对这个理由提出了异议（[HENN82，RAD183，PATT82b]）。他们发现复杂指令难以使用，因为编译器必须找到严格满足限制的情况。像优化生成的代码，以达到减小代码长度、减少指令执行数目和增强流水这样的任务，使用复杂指令集也是非常困难的。作为这个观点的证据，本章前面所援引的一些研究报告曾指出，编译后程序中的大多数指令都是相当简单的。

前面提到的采用 CISC 的另一个理由是，CISC 可生成更小、更快的程序。我们考察这个主张的两个方面：程序将更小并且执行得更快。

小程序有两个优点。首先，程序占用内存少，这就节省了资源。但今天的存储器价格已如此廉价，以至于这个潜在的优点不再令人信服。更重要的是，较小的程序能改善性能。这表现在两个方面：一是较少的指令意味着待取的指令字节也少；二是内存分页环境下，较小的程序占据较少的页，减少了缺页中断。

与这个理由并存的问题是：CISC 程序将小于相应的 RISC 程序，但这一点远不像看起来那么肯定。多数情况下，以符号机器指令表示的 CISC 程序是会“短”些（即较少的指令），但它们所占据的存储器位数却不见得更“小”。表 13-6 列出了来自三个研究的结果，比较了几类机器上的编译后 C 程序的大小，其中包括精简指令集结构的 RISC I。注意，CISC 比 RISC 没有或只有少许的节省。另一点也是值得注意的，VAX 比 PDP－11 的程序并无大量减少，而前者比后者采用了更为复杂的 CISC 结构。这些结果也得到 IBM 研究人员的赞同［RAD183］，他们发现 IBM 801（一台 RISC 机器）产生的代码只是 IBMS/370 产生代码的 0.9 倍。他们的研究使用了一组 PL/1 程序。

表 13-6 相对于 RISC I 的代码大小

	[PATT82a] 11 个 C 程序	[KATE83] 12 个 C 程序	[HEAT84] 5 个 C 程序
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

有几个理由可解释这些令人惊奇的结果。我们曾指出过，CISC 上的编译器有偏爱简单指令的倾向，结果使得复杂指令所提供的简洁性很少能发挥作用。还有，CISC 上更多的指令数要求较长的操作码，这使指令较长。最后，RISC 强调寄存器而不是存储器的访问，因而要求的指令位数也少。最后一种效应的例子，见下一小节对图 13.5 的讨论。

因此，期望 CISC 能产生较小的程序并带有其他优点，是不太现实的。增加指令集复杂性的第二个动因是它的指令执行可能会更快。看起来这个观点是言之有理的，一个复杂的 HLL 操作，作为一条机器指令将会比作为一串更原始的指令执行得更快。然而，因为实际情况是偏向使用较为简单指令，因此上述的想法不见得成立。为适应丰富的指令集，每个控制器必须做得更复杂，而且微程序控制存储也必须做得更大。不论哪种因素都增加了简单指令的执行时间。

实际上，某些研究者已发现，加速复杂函数的执行不在于复杂的机器指令是多么强有力，而

在于它们驻留在高速控制存储中［RAD183］。该控制存储实际上起到指令 cache 的作用。于是，硬件结构研究者面临的任务便是，确定什么样的子程序或者函数将使用得最频繁，然后将它们指派到控制存储中，通过微代码实现它们。然而结果并不那么令人鼓舞。于是，在 S/390 系统上，像翻译（translate）和扩展精度的浮点除法（extended-precision-floating-point-divide）这样的指令驻留在高速存储中，而涉及建立过程调用或初始中断处理程序这样重要的指令序列反而放在较慢的主存中。

于是，朝更加复杂指令集方向发展是否合适，远不是那么清楚，这导致了几个研究组朝相反的方向探索。

13.4.2　精简指令集体系结构特征

虽然精简指令集结构可能采取各种不同的方法，但某些特征对它们都是共同的。这些特征包括：

每周期一条指令、寄存器到寄存器的操作、简单的寻址方式、简单的指令格式。

下面我们简要介绍这些特征，稍后再给出具体的例子。

第一个特征是，**每机器周期一条机器指令**。机器周期（machine cycle）被定义成由寄存器取两个操作数，完成一个 ALU 操作，然后再将结果写入寄存器所用的时间。于是，RISC 机器指令不会比 CISC 机器上的微指令复杂，执行大约也是一样快。简单的单周期指令很少或没有对微代码的需求，机器指令能以硬布线方式实现。这样的指令应当比其他机器上的类似指令执行得更快，因为在指令执行期间它不必去访问微程序控制存储器。

第二个特征是，大多数操作应是寄存器到寄存器的，只以简单的 LOAD 和 STORE 操作访问存储器。这个设计特点简化了指令集，进而也简化了控制器。例如，一个 RISC 指令集可只包括一条或两条 ADD 指令（如整数加、带进位加），而 VAX 有 25 种不同的 ADD 指令。这种结构的另一好处是：它鼓励寄存器的优化使用，使频繁存取的操作数保留在高速存储中。

这种对寄存器到寄存器操作的强调，对于 RISC 设计是特别值得注意的。虽然其他当代机器也提供这种指令，但它们还包括存储器到存储器和混合的寄存器/存储器操作。对这些不同设计方法的比较是在 1970 年完成的，在 RISC 出现之前。图 13-5a 说明了所采取的方法。评价原型结构只在于比较程序的大小和传输的存储器位数。这样的结果曾使一个研究人员建议，将来的结构应完全不含有任何寄存器［MYER78］。人们奇怪他怎么会这样想，那个时候（1978 年），RISC 机器 Pyramid 已经面市，它含有不少于 528 个寄存器。

这些研究遗漏的是，对少量局部标量的频繁访问，和以大量的寄存器或优化的编译器能使大多数操作数长时间地保持在寄存器中。于是，前面的图 13-5b 可能是更公平的比较。

第三个特征是使用简单的寻址方式。几乎全部指令都使用寄存器寻址方式，其他几种寻址方式，像偏移寻址和 PC 相对寻址，也可能包括进来。其他的更为复杂的寻址方式可由这些简单方式用软件来合成。同样，这个设计特征简化了指

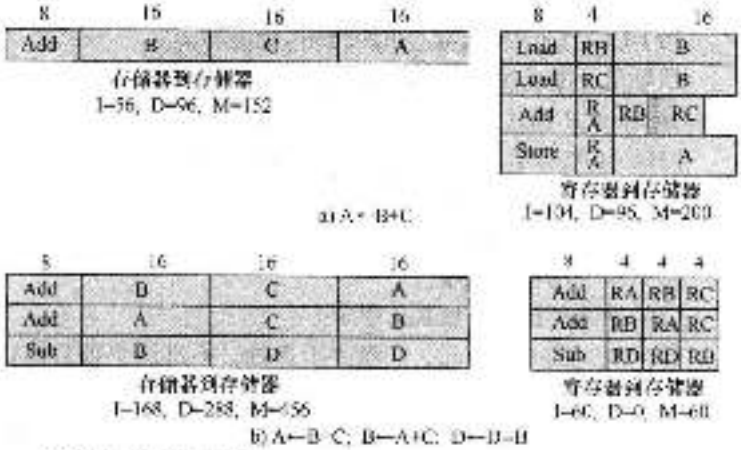


图 13-5　对寄存器到寄存器和存储器到存储器方法的两个比较

令集和控制器。

最后一个公共特征是使用简单的指令格式，而且通常仅使用一种或少数几种格式。指令长度固定并且在字边界上对齐。字段位置，特别是操作码字段位置是固定的。这个设计特点有多个优点。对于固定的字段，操作码的译码和寄存器操作数的访问能同时出现。简化了格式也就简化了控制器。因为以字长单位来取指令和数据，取指令也就被优化了。这还意味着，单一指令不会跨越内存分页的边界。

将这些特征综合在一起进行评估，就能确定 RISC 方法的潜在优势。这也有相当数量的“间接证据”。首先，能开发出更有效的优化编译器。利用更原始的指令，对于无循环的传送功能、有效地重组代码、最大化寄存器的使用等，都会有更多的机会，甚至能在编译时求解复杂指令的作用。例如，S/390 的传送字符（move characters，MVC）指令能将字符串由一个位置传送到另一位置上。每当执行该指令时，传送将取决于串的长度、是否或在什么方向上位置有重叠，以及排列的特征是什么。大多数情况下，这些在编译时都将是已知的。于是，编译器能为这种操作生成一个优化的原始指令序列。

其次，前面已指出，是编译器生成的大多数指令从任何方面讲都是相对简单的指令。专门为这些指令来构造控制器看起来是有道理的，并且很少或根本不使用微代码来执行它们，要比相应的 CISC 快得多。

另外，与指令流水的使用有关。RISC 研究者们发现，精简指令集能非常有效地应用指令流水技术。我们即将更详细地考察这一点。

最后一点有时不太明显的是，RISC 程序应能更好地响应中断，因为中断是相当于在基本操作之间检查的。使用复杂指令的结构要么将中断限定在指令边界上，要么定义专门的中断点，并为重启动一条指令实现一种结构。

精简指令集结构的性能改善情况还没有完全验证。已有几个这方面的研究，但不是在技术和功能方面可比的机器上进行的。而且，大多数的研究还没有将精简指令集的效应与大寄存器组的效应分开。然而，上述的“间接证据”还是有启发的。

13.4.3 CISC 与 RISC 特征对比

在对 RISC 机器的最初热情之后，人们逐渐认识到：

（1）RISC 设计包括某些 CISC 特色会有好处。

（2）CISC 设计包括某些 RISC 特色也会有益。

结果是，最近的 RISC 设计，以 PowerPC 为代表，不再是“纯”RISC 了。而最近的 CISC 设计以 Pentium II 和其后的型号为代表，也结合了某些 RISC 特征。

[MASH 95] 给出了一个令人感兴趣的比较，提供了对这些观点的某些见识（参见表 13-7）。此表列出了几种处理器，并对几个特征进行了比较。作为这个比较的目的，可以列出如下一些被认为是典型的 RISC 特征。

（1）单一的指令长度。

（2）典型的指令长度是 4 字节。

（3）较少的寻址方式，一般少于 5 种。不过这个数目难以限定。表中未计入寄存器和立即数的方式，而带有不同位移大小的不同格式却分别予以统计了。

（4）无间接寻址。间接寻址要求先进行一次存储器访问来得到操作数的存储器地址。

（5）装载/保存操作不与算术操作混在一起（例如，由存储器加或加到存储器）。

（6）每条指令不会有多于一个的存储器操作数。

（7）对装载/保存操作，不支持数据的任意对齐。

（8）对指令中的数据地址，最大化存储管理单元（Memory Management Unit，MMU）的使用。

(9) 整数寄存器指定符的位数等于 5 或更多。这意味着，至少有 32 个整数寄存器能被显式地引用。

(10) 浮点寄存器指定符的位数等于 4 或更多。这意味着，至少有 16 个浮点寄存器也能被显式地引用。

(1) ~ (3) 项是指令译码复杂性的标示；(4) ~ (8) 项揭示了流水线技术实现的难易，特别是当出现虚拟存储器要求时；(9) 和 (10) 项关系到使用编译器的能力。

表 13-7 中的前 8 个处理器是明显的 RISC 结构，下面 5 个是明显的 CISC，最后两个处理器常被看作是 RISC，但事实上它们有不少 CISC 特征。

表 13-7　某些处理器特征

处理器	不同长度的指令数	最大指令长度字节	寻址方式数目	间接寻址	LOAD/STORE 结合算术	最多的存储器操作数	未对齐寻址允许	使用的 MMU 最大数量	整数寄存器指定符的位数	浮点寄存器指定符的位数
AMD29000	1	4	1	无	无	1	否	1	8	3①
MIPS R3000	1	4	1	无	无	1	否	1	5	4
SPARC	1	4	2	无	无	1	否	1	5	4
MC88000	1	4	3	无	无	1	否	1	5	4
HP PA	1	4	10①	无	无	1	否	1	5	4
IBM RT/PC	2①	4	1	无	无	1	否	1	4②	3②
IBM RS/6000	1	4	4	无	无	1	是	1	5	5
Intel i860	1	4	4	无	无	1	否	1	5	4
IBM 3090	4	8	2②	无②	有	3	是	4	4	2
Intel 80486	12	12	15	无②	有	2	是	4	3	3
NSC 32016	21	21	23	有	有	2	是	4	3	3
MC 68040	11	22	44	有	有	3	是	8	4	3
VAX	56	56	22	有	有	6	是	24	4	0
Clipper	4②	8②	9②	无	无	1	0	2	4②	3②
Intel 80960	2②	8②	9②	无	无	1	是①	—	5	3②

① 六符合该特征为 RISC。
② 六符合该特征为 CISC。

13.5　RISC 流水线技术

13.5.1　使用规整指令的流水线技术

正如 12.4 节所讨论过的，指令流水线技术经常被使用以提高性能。让我们再在 RISC 结构前提下考虑这个问题。大多数指令是寄存器到寄存器的，并且指令周期有如下两个阶段。

- I：取指令。
- E：执行。带寄存器输入和输出，完成一个 ALU 操作。

对于装载和保存操作，需要三个阶段。

- I：取指令。
- E：执行（计算存储器地址）。
- D：存储（寄存器到存储器或存储器到寄存器操作）。

图 13-6a 描述了不使用流水线技术的一个指令序列的操作顺序。很清楚，这个过程有浪

费，即使一个很简单的流水线技术都能实质性地改善其性能。图 13-6b 表示一种两段流水处理策略，在此流水线处理方式中，两个不同指令的 I 和 E 同时完成。这种策略能产生串行策略两倍的执行速率。有两个问题妨碍了这种最大速率的达到。第一个问题是，如果假定使用单端口存储器，那么每个阶段只准一次存储器访问，这就要求在某些指令的执行中插入等待状态。第二个问题是，一条分支指令能打断顺序的执行流。为了以尽量少的电路来应对这种情况，可通过编译器或汇编器将 NOOP 指令插入指令流中。

通过允许每个阶段有两次存储器访问，能进一步改进流水线的性能。这产生了如图 13-6c 所示的序列。现在，能重叠执行多达 3 条指令，改善的比例最大可达到 3 。同样，分支指令使加速不能达到最大允许值。还有，注意数据相关性也有影响。若一个指令需要某个操作数，而该操作数会由前面指令所更新，则需要一个延迟。同样，这能用插入 NOOP 来实现。

至此，若 3 个阶段有大致相等的期间，所讨论的流水线就能很好地工作。因为 E 阶段通常涉及一个 ALU 操作，它可能会更长一些。这种情况下，我们能将它分成两个子阶段。

- E_1：寄存器组读。
- E_2：ALU 操作和寄存器写。

由于指令集的简单性和规整性，设计 3 个或 4 个阶段的流水线很容易。图 13-6d 表示使用 4 段流水的结果。多达 4 条指令能同时进行，最大可能的加速比是 4。请再次注意，考虑到因数据和分支的延迟而插入 NOOP 指令。

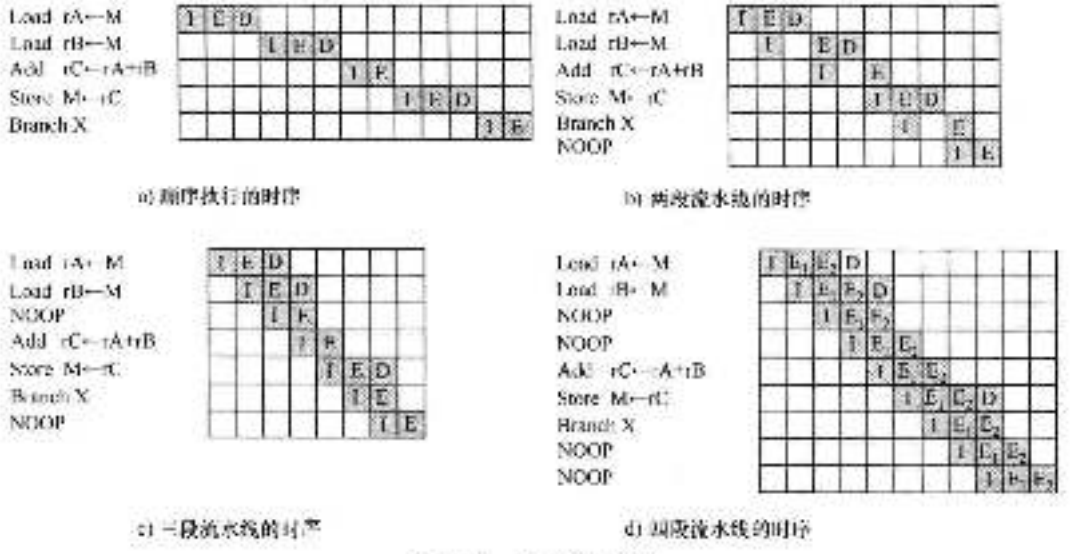


图 13-6 流水线的效果

13.5.2 流水线的优化

由于 RISC 指令集的简单性和规整性，能有效地使用流水线策略。指令的执行时间没有多少变动，并且可以对流水线进行改进以反映这些变动。然而，我们已看到，数据的相关性和分支指令会降低整体的执行速率。

1. 延迟分支

为抵消这些相关性带来的性能损失，开发人员采用了代码重组（code reorganization）技术。提高流水线效率的一种方式是延迟分支（delayed branch）。它利用了分支指令直到下面一条指令之后才产生影响这一特点，在分支指令之后安排一条有用指令来替代仅为延迟的空操作。分支指令之后的指令位置被称为延迟槽（delay slot）。这种奇特的处理可由表 13-8 说明。表中的第一列，我们看到的是以符号指令表述的正常的机器语言程序。在 102 处的分支指令执行之后，将要

执行的下一条指令在 105 处。为规范流水线，在分支指令之后插入了一条 NOOP 指令。然而，若将 101 处和 102 处的指令进行交换，则能实现性能的提升。

表 13-8　正常的和延迟的转移

地址	正常分支	延迟分支	优化的延迟转移
100	LOAD X，rA	LOAD X，rA	LOAD X，rA
101	ADD 1，rA	ADD 1，rA	JUMP 105
102	JUMP 105	JUMP 106	ADD 1，rA
103	ADD rA，rB	NOOP	ADD rA，rB
104	SUB rC，rB	ADD rA，rB	SUB rC，rB
105	STORE rA，Z	SUB rC，rB	STORE rA，Z
106		STORE rA，Z	

图 13-7 表示了上述处理的结果。图 13-7a 展示的是传统流水线方案，这种方案已在第 12 章讨论过（参见图 12-11 和图 12-12）。在时间 3 取来 JUMP 指令。在时间 4，此 JUMP 指令在执行，与此同时指令 103（ADD 指令）已被取来。因为 JUMP 的出现，修改了程序计数器，流水线必须清除指令 103。在时间 5，JUMP 的目标，即指令 105 被取来。图 13-7b 展示的是典型 RISC 组织来处理同样的流水线，时序相同。只不过由于插入 NOOP 指令，我们不再需要清除流水线的专门电路了；此 NOOP 简单地被执行，无任何影响。图 13-7c 展示的是延迟分支方法的使用。此 JUMP 指令现在是在时间 2 取来。注意，JUMP 指令将修改程序计数器，然而在此之前，ADD 指令已在时间 3 取来了。于是在时间 4，ADD 指令在执行，同时指令 105 被取来。这样，既保持程序的原语义，又使指令总执行时间减少了至少一个时钟周期。

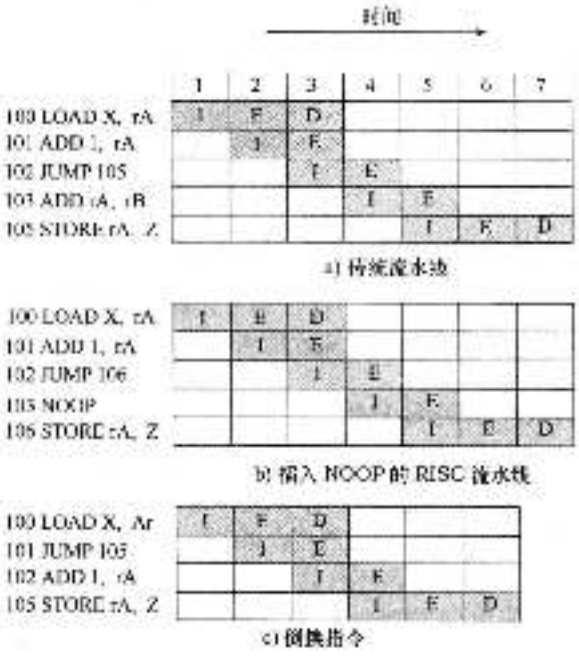


图 13-7　延迟分支的使用

对于无条件跳转、调用和返回，都能成功地进行这种交换。然而，不能自动地将其施加到条件分支指令上。若分支所测试的条件会被前面这条指令所修改，则编译器应避开这种交换，而插入 NOOP 指令。否则，编译器能在分支指令之后插入有用指令。以 Berkeley RISC 和 IBM 801 系统二者的经验来看，大多数的条件分支指令都能以这种交换方式得到优化（[PATT82a]，[RADI83]）。

2. 循环展开

另一种提高指令并行性的编译技术是循环展开（loop unrolling）[BACO94]。它把一个循环的循环体复制若干次，其次数被称为展开因子（a），从而以步长 a 来执行循环，而不是步长 1。



循环展开是通过以下方式来提高性能的：

- 降低循环开销。
- 通过提升流水线性能来提高指令并行性。
- 提高寄存器、数据高速缓存或页表快速缓存（TLB）的局部性。

图 13-8 用例子描述了这 3 种性能改进的效果。循环开销降低为原来的一半，因为在循环结尾的测试和分支之前，一次执行了两个迭代。指令并行性的提高是因为第二个赋值的执行可以与第一个赋值的结果保存和循环变量的更新同时进行。如果数组元素是被赋值到寄存器中，寄存器的局部性可获得提升，因为 a [i] 和 a [i+1] 在展开后的循环体中被用了两次，把每次循环迭代中的内存装载次数从 3 次降低到了 2 次。

```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

a) 原来的循环

```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = 1)  then
   a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

b) 循环展开两次

图 13-8 循环展开

最后应当指出，指令流水线的设计不应与其他适用于系统的优化技术隔离开进行。例如，[BRAD91b] 指出，流水的指令调度策略应与寄存器的动态分配一起考虑，以提高效率。

13.6 MIPS R4000

最早商品化的一种 RISC 芯片组是 MIPS Technology 公司开发的。此系统受到一个斯坦福研制的一个实验系统（也叫 MIPS）的启发 [HENN84]。MIPS 系列最新产品号是 R4000，它实质上与 MIPS 设计的早先产品：R2000、R3000 具有相同的体系结构和指令集。最显著的不同是 R4000 使用的是 64 位而不再是 32 位位宽，用于所有内部和外部数据路径和地址、寄存器以及 ALU。

使用 64 位要比 32 位结构有几个好处。它允许更大的地址空间：大到能使操作系统将比太字节还大的文件直接映射到虚拟存储器，使存取变得很容易。现在普遍使用的 1TB 或更大的磁盘驱动器，32 位机器的 4GB 地址空间变成了一种限制。另外，64 位能允许 R4000 处理像 IEEE 双精度浮点数这样的数据，以及处理字符串数据时能一次处理多达 8 个的字符。

R4000 处理器芯片分成两个部分，一部分含有 CPU，另一部分含有存储管理单元。CPU 是个很简单的结构，其设计思想是，尽可能使指令执行逻辑简单，留出空间用于增强性能的逻辑（例如，完整的存储管理单元）。

处理器支持 32 个 64 位寄存器。它还提供多达 128KB 的高速 cache，一半用于指令，一半用于数据。这种相对大的 cache（IBM 3090 提供 128 – 256KB 的 cache）使系统能保持更多的程序代码和数据在处理器内，从而减轻了主存总线的负荷，也避免了对大寄存器组及其配套窗口逻辑的需求。

13.6.1 指令集

表 13-9 列出了所有 MIPS R 系列处理器的基本指令集。所有指令都以单一 32 位字格式来编码。所有数据操作都是寄存器到寄存器，仅纯装载/保存操作有存储器访问。

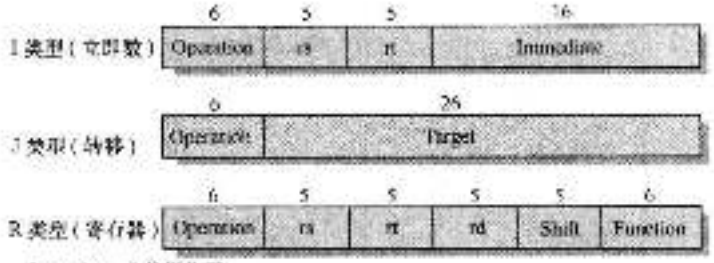
R4000 没使用条件码。若一条指令产生某个条件，其相应的标志存于一个通用寄存器中。这就避免了专门用于处理条件码的逻辑，因为它们影响流水线机制和编译器对指令的重排序。流

水线已经实现了处理寄存器值相关的机制。而且，映射到寄存器组的条件在分配与再使用上，与存于寄存器其他值都一样可由编译器在编译时间优化。

表 13-9　MIPS R 系列指令集

操作	操作码	说明	操作	操作码	说明
装载/保存指令	LB	装载字节	移位指令	SRLV	逻辑右移变量
	LBU	装载无符号字节		SRAV	算术右移变量
	LH	装载半字	乘/除指令	MULT	乘
	LHU	装载无符号半字		MULTU	无符号乘
	LW	装载字		DIV	除
	LWL	装载左字		DIVU	无符号除
	LWR	装载右字		MFHI	由 HI 送出
	SB	保存字节		MTHI	送至 HI
	SH	保存半字		MFLO	由 LO 送出
	SW	保存字		MTLO	送至 LO
	SWL	保存左字	跳转和分支指令	J	跳转
	SWR	保存右字		JAL	跳转并链接
算术指令（ALU 立即数）	ADDI	加立即数		JR	跳转并链接
	ADDIU	加无符号立即数		JALR	跳转并链接寄存器
	SLTI	小于立即数置位		BEQ	相等分支
	SLTIU	小于无符号立即数置位		BNE	不等分支
	ANDI	AND 立即数		BLEZ	小于或等于零分支
	ORI	OR 立即数		BGTZ	大于零分支
	XORI	XOR 立即数		BLTZ	小于零分支
	LUI	装入上部立即数		BGEZ	大于或等于零分支
算术指令（3 操作数，R 类型）	ADD	加		BLTZAL	小于零分支并链接
	ADDU	无符号加		BGEZAL	大于或等于零分支并链接
	SUB	减	协处理器指令	LWC_z	装载字到协处理器
	SUBU	无符号减		SWC_z	保存字到协处理器
	SLT	小于置位		MTC_z	传送到协处理器
	SLTU	无符号小于置位		MFC_z	由协处理器传出
	AND	与		CTC_z	传送控制到协处理器
	OR	或		CFC_z	由协处理器传出控制
	XOR	异或		COP_z	协处理器操作
	NOR	或非		BC_zT	协处理器 z 真时分支
移位指令	SLL	逻辑左移		BC_zF	协处理器 z 假时分支
	SRL	逻辑右移	专门指令	SYSCALL	系统调用
	SRA	算术右移		BREAK	断点
	SLLV	逻辑左移变量			

与大多数 RISC 类机器一样，R4000 使用单一的 32 位指令长度。这既简化了取指令和译码，又简化了取指令与虚拟存储管理单元的相互作用（即指令不穿越字或页的边界）。它的三种指令格式（见图 13-9）共享操作码和寄存器引用的公共格式，简化了指令译码，可在编译时间以简单指令的合成实现更复杂指令的效果。



Operation 6 位操作码
rs 5 位源寄存器指定符
rt 5 位源/目标寄存器指定符
Immediate 16 位立即数，转移或地址偏移
Target 26 位跳转目标地址
rd 5 位目标寄存器指定符
Shift 3 位移位总量
Function 6 位 ALU/shift 功能指定符

图 13-9 MIPS 的指令格式

只有最简单的和最经常使用的存储器寻址方式是以硬件实现的。所有的存储器引用由一个 32 位寄存器和一个 16 位的对此寄存器基址的偏移量组成。例如，"装载字"指令最如下形式：

```
lw r2, 128(r3) /* 装载一个字到寄存器 r2，该字的地址是寄存器 r3 的值加上偏移量 128
```

32 个通用寄存器的任意一个都可被用作基址寄存器。有一个寄存器，r0，所包含的值总是 0。

编译器使用多条机器指令的合成，来实现普通机器中的典型寻址方式。下面给出了一个来自［CHOW87］的例子。其中使用了 lui（load upper immediate，装载上部立即数）指令，这条指令将 16 位立即数装入寄存器高半部，低半部会置为 0。考虑如下使用一个 32 位立即数作为参数的汇编语言指令。

```
lw r2, 4[imm](r4) /* 装载一个字到寄存器 r2，该字的地址是寄存器 r4 的值加上 32 位的立即数偏移量 4[imm]
```

上面这条指令将被编译为下列 MIPS 指令。

```
lui r1, %imm. hi /* 装载偏移量 %imm 的高 16 位 %imm. hi 到寄存器 r1
add r1, r1, r4 /* 把寄存器 r1 中的 4[imm.-hi] 与 r4 的值相加，结果保存回寄存器 r1
lw r2, %imm.-lo(r1) /* %imm. lo 是偏移量 %imm 的低 16 位。
```

13.6.2 指令流水线

以其简化的指令集结构，MIPS 能实现很有效的流水。考察 MIPS 流水线的演变情况是有指导意义的，因为它在大体上说明了 RISC 流水技术的改进。

最初实验 RISC 系统和第一代商品化的 RISC 处理器，实现了大约每系统时钟周期 1 条指令的执行速度。为改善这种性能，两类处理器：超标量和超级流水线体系结构，已发展到能每时钟周期执行多条指令。从本质上讲，超标量体系结构（super scalar architecture）复制每个流水线段，使得流水线的同一阶段可以同时处理两条或多条指令。超级流水线体系结构（super pipelined architecture）是使用更多更细致的流水阶段。对这种更多的段，更多的指令能同时处于流水线中，从而提高并行度。

这两种方法都有限制。对超标量流水线技术，不同流水线中指令间的相关性会减慢系统。还有，为协调这些相关性也要求一些辅助逻辑。对超级流水线，指令由一个阶段传送到下一阶段的

开销也会增加。

第 14 章将专门研究超标量体系结构。MIPS R4000 是一个基于 RISC 的超级流水线体系结构的好例子。

图 13-10a 显示了 R3000 的指令流水线，指令在流水线中每时钟周期前进一步。MIPS 编译器能重排序指令，在 70% ~ 90% 的情况下能填充分支延迟槽。所有指令都流经如下 5 个流水阶段：

- 取指令
- 由寄存器组取源操作数
- ALU 运算或数据操作数地址生成
- 数据存储器访问
- 写回到寄存器组

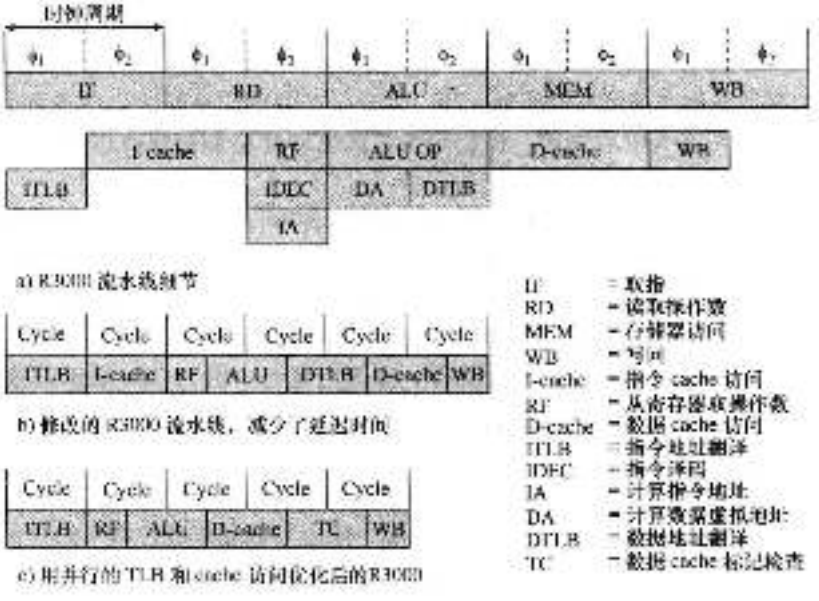


图 13-10　增强 R3000 流水线

正如图 13-10a 所显示的，这里不仅有流水线的并行性，也有单一指令执行内的并行性。60ns 的时钟周期分成两个 30ns 的相位。外部指令和数据的 cache 存取操作，每项操作都需要 60ns，主要的内部操作（OP、DA、IA）也需要同样的时间。指令译码是一个较简单操作，只要求一个 30ns 相位，能与同一指令的寄存器读取重叠。一个分支指令的地址计算亦与指令译码和寄存器读取重叠，于是一条分支指令 i 可提供指令 $i+2$ 访问 ICACHE 的地址。类似地，指令 i 可装载会被指令 $i+1$ 的操作紧接着使用的操作数，与此同时，一个 ALU/shift 的结果能直接传递给指令 $i+1$ 而没有任何延迟。这种指令间的紧耦合有利于高效流水。

每个时钟周期分成的两个相位分别标为 $\phi 1$ 和 $\phi 2$。每个相位所完成的功能总结见表 13-10。

表 13-10　R3000 流水段

流水段	相　位	功　能
IF	$\phi 1$	使用 TLB，将指令的虚拟地址转换成物理地址（在分支转移判定之后）
	$\phi 2$	送出物理地址到指令 cache

（续）

流水段	相 位	功 能
RD	$\phi 1$	上指令 cache 送回一个指令，比较标记确认所取指令有效性
	$\phi 2$	译码指令，读寄存器堆，旁转移，计算转移目标地址
ALU	$\phi 1+\phi 2$	若是寄存器到寄存器操作，则逻辑或算术运算完成
	$\phi 1$	若是一个分支，判定转移是否发生
		若是一个存储器访问（装载或保存），计算数据的虚拟地址
	$\phi 2$	若是一个存储器访问，使用 TLB 将虚拟地址转换成物理地址
MEM	$\phi 1$	若是一个存储器访问，送物理地址到数据 cache
	$\phi 2$	若是一个存储器访问，从数据 cache 返回数据并检查标记
WB	$\phi 1$	写到寄存器堆

R4000 比 R3000 又有几点技术改进。使用更先进的技术，使时钟周期缩短到原来的一半，即 30ns，寄存器的存取时间也缩短到原来的一半。另外，芯片的密度更大，指令和数据 cache 能集成到芯片上。在最后考察 R4000 之前，让我们先考虑 R3000 应如何修改以使用 R4000 技术提高性能。

图 13-10b 表示第一步。注意此图中的周期已是图 13-10a 中周期的一半长。因为指令和数据 cache 在处理器同一芯片上，它们的存取时间也是原来的一半长。故它们的流水阶段仍占据一个时钟周期。同样，因为寄存器堆存取的加速，寄存器读和写仍占据时钟周期的一半。

因为 cache 在芯片上，虚拟地址到物理地址的转换会延迟 cache 访问。可通过虚拟地址索引的 cache，从而使 cache 存取和地址转换的并行，以缩短延迟。图 13-10c 表示以这种改进而优化的 R3000 流水线。因为事件的密集，数据 cache 标记的检查放在 cache 存取之后的下一周期来完成。

在超级流水线式系统中，通过插入流水线的寄存器，每个流水阶段被细分开来，使得原有硬件在每周期可被多次使用。基本上，超流水线的每个阶段是以基本时钟频率的几倍来操作，倍数取决于超流水程度。R4000 所具有的速度和密度，准许它有级别为 2 的超级流水线。图 13-11a 表示使用这种超级流水线的优化的 R3000。注意，它基本上同于图 13-10c 所示的动态结构。

为了进一步改进 R4000，在其上设计了一个更大的专门的加法器，这使它能以两倍的速率来执行 ALU 操作。另外的改进允许装载和保存以两倍的速率来执行。改进后的流水线示于图 13-11b。



图 13-11 R3000 的理论超级流水线与 R4000 的实际超级流水线

R4000 有 8 个流水段，意味着多达 8 条指令能同时在流水线中。流水以每时钟周期两段的速率向前推进。8 个流水段如下所述。

- 取指令的前一半（instruction fetch first half）：虚拟地址提交给指令 cache 和转换后备缓冲器（TLB）。

- **取指令的后一半**（instruction fetch second half）：指令 cache 送出指令和 TLB 生成物理地址。
- **寄存器组**（register file）：三个动作并行出现。
  - 指令被译码，并对联锁条件进行检查（即这条指令取决于前面指令的执行结果）。
  - 进行指令 cache 标记（tag）检查。
  - 由寄存器组读取操作数。
- **指令执行**（instruction execute）：可能出现下列三个动作之一。
  - 若指令是寄存器到寄存器的操作，ALU 完成此算术或逻辑操作。
  - 若指令是装载/保存指令，计算数据的虚拟地址。
  - 若指令是分支指令，计算转移目标的虚拟地址并检查转移条件。
- **数据 cache 前一半**（data cache first half）：虚拟地址提交给数据 cache 和 TLB。
- **数据 cache 后一半**（data cache second half）：数据 cache 输出数据，TLB 生成物理地址。
- **标记检查**（tag check）：为装载/保存完成 cache 标记检查。
- **写回**（write back）：指令结果写回到寄存器组。

13.7　SPARC

SPARC（scalable processor architecture，可扩展处理器体系结构）是指一种由 Sun Microsystems 公司定义的处理器结构。Sun 已开发了自己的 SPARC 实现，而且也许可其他厂商生产 SPARC 兼容机。SPARC 结构的开发从 Berkeley 的 RISC 机器得到了许多启发，它的指令集和寄存器组织也紧密基于 Berkeley RISC 模型之上。

13.7.1　SPARC 寄存器组

与 Berkeley RISC 一样，SPARC 也使用了寄存器窗口。每个窗口由 24 个寄存器组成；总的窗口数是 2 到 32 个，实际数目取决于具体实现。图 13-12 展示了一个 8 窗口的实现，总共使用了 136 个物理寄存器。正如 13.2 节讨论中所指出的，这是一个合理的窗口数。物理寄存器 0 到 7，是所有过程共享的全局寄存器。每个过程见到逻辑寄存器号从 0 到 31。逻辑寄存器 24 到 31 标记为输入（ins），是与调用过程（父过程）共享的。逻辑寄存器 8 到 15 标记为输出（outs），是与被调用过程（子过程）共享的。这两部分与其他窗口重叠。逻辑寄存器 16 到 23 标记为局部的，是本过程使用的局部寄存器，既不与其他过程共享，也不与其他窗口重叠。再次，如同 13.1 节讨论中所指出的，8 个寄存器用于参数传送，在大多

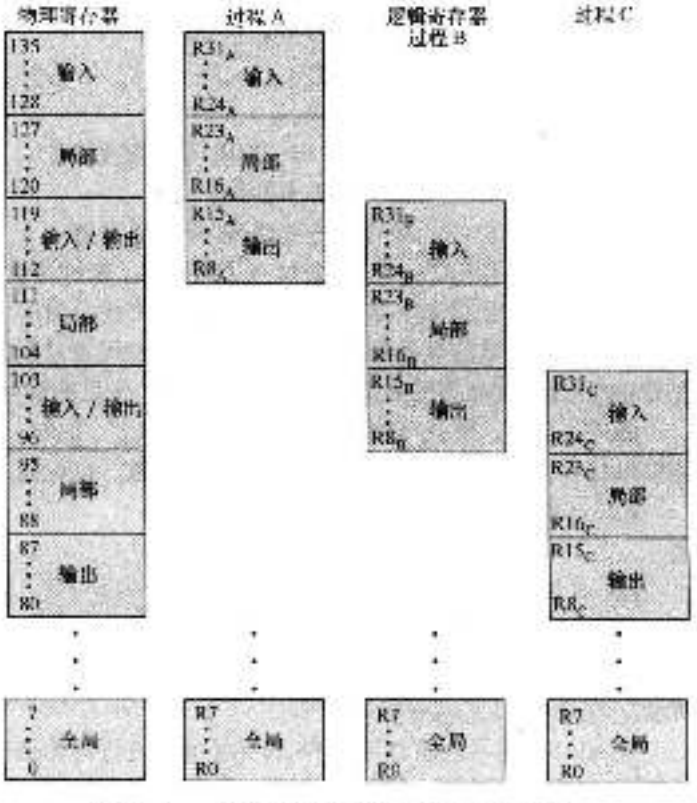


图 13-12　三个过程的 SPARC 寄存器窗口布局

数情况下，应该就足够了（见表13-4）。

图13-13是寄存器重叠使用的另一种视图。调用过程将要传送的参数放入它的outs寄存器中，被调用过程将这同一组物理寄存器看作它的ins寄存器。处理器维护一个指向当前执行过程窗口的指针，称为当前窗口指针（current window pointer，CWP）。CWP位于处理器状态寄存器（PSR）中，此寄存器中还有一个窗口无效屏蔽WIM（window invalid mask），它指示哪个窗口无效。

使用SPARC的寄存器结构，过程调用通常没必要保存和恢复寄存器。因为编译器只需关心以有效方式为过程分配局部寄存器，而无需关心过程间的寄存器分配，故编译器大大简化了。

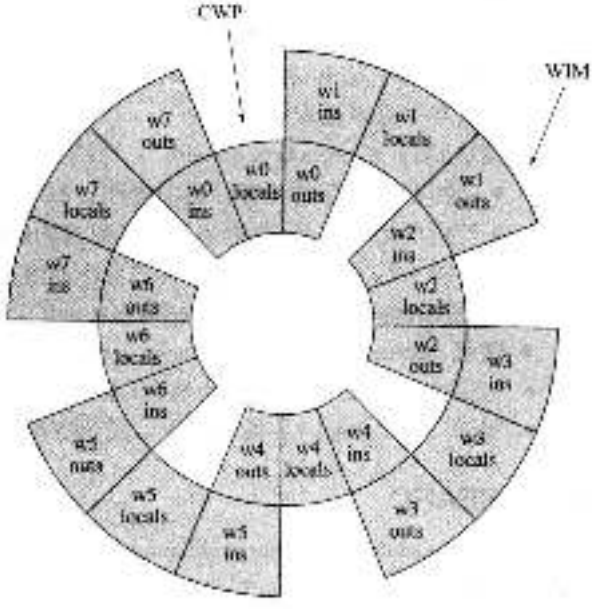


图13-13 SPARC十8个寄存器窗口构成了一个环形栈

13.7.2 指令集

表13-11列出了SPARC结构的指令集。大多数指令只使用寄存器操作数。寄存器到寄存器指令有三个操作数，并能表示成：

$R_d \leftarrow R_s$ op S2

其中，R_d和R_s是寄存器，S2或者是寄存器，或者是一个13位立即数。寄存器零（R_0）已被硬布线为0值。这种指令形式非常适合于具有高比例的局部标量和常数的程序。

可由ALU实现操作的指令分成如下几组：

- 整数加法（带或不带进位）。
- 整数减法（带或不带借位）。
- 按位的布尔运算AND、OR、XOR及其取反操作。
- 逻辑左移、逻辑右移和算术右移。

除移位指令之外，所有这些指令都可选择设置4个条件代码：零（ZERO）、负（NEGATIVE）、上溢（OVERFLOW）、进位（CARRY）。带符号整数以32位的2的补码形式表示。

只有简单的装载和保存指令访问存储器，并区分对字（32位）、双字、半字、字节的装载或保存。半字或字节的装载指令还可按有符号数和无符号数区别对待。对有符号数，符号位被扩展填充32位目的寄存器的高位。对无符号数，32位目的寄存器的高位将被0填充。

除寄存器外，唯一可用的寻址方式是“偏址”方式，即操作数的有效地址（EA）是基址和偏移量之和。基址来自寄存器，偏移量是立即数，也可能来自于寄存器，即可表示为：

$EA = (R_s) + S2$

或

$EA = (R_s) + (R_s)$

为完成装载或保存，指令周期需添加一个额外相位，在第2步使用ALU完成存储器地址计算，在第3步装载或保存。这种单一寻址方式非常灵活，能综合形成其他寻址方式，如表13-12所示。

将SPARC的寻址能力与MIPS的寻址能力进行对比是有益的。MIPS使用16位偏移，SPARC使用13位偏移。另外，MIPS不准许一个地址由两个寄存器的内容构成，而SPARC准许。

表 13-11　SPARC 指令集

指令	助记符	说　明	指令	助记符	说　明
装载/保存指令	LDSB	装载有符号字节	算术指令	ADD	加
	LDSH	装载有符号半字		ADDCC	加，设置条件码
	LDUB	装载无符号字节		ADDX	带进位加
	LDUH	装载无符号半字		ADDXCC	带进位加，设置条件码
	LD	装载字		SUB	减
	LDD	装载双字		SUBCC	减，设置条件码
	STB	保存字节		SUBX	带借位减
	STH	保存半字		SUBXCC	带借位减，设置条件码
	ST	保存字		MULSCC	多步，设置条件码
	STDD	保存双字	跳转/转移指令	BCC	依条件转移
移位指令	SLL	逻辑左移		FBCC	依浮点条件转移
	SRL	逻辑右移		CBCC	依协处理器条件转移
	SRA	算术右移		CALL	调用过程
布尔运算指令	AND	AND		JMPL	跳转并链接
	ANDCC	AND，设置条件码		TCC	依条件转移
	ANDN	NAND		SAVE	前移寄存器窗口
	ANDNCC	NAND，设置条件码		RESTORE	后移寄存器窗口
	OR	OR		RETT	由自陷返回
	ORCC	OR，设置条件码	其他指令	SETHI	设置高 22 位
	ORN	NOR		UNIMP	未实现的指令（自陷）
	ORNCC	NOR，设置条件码		RD	读专门寄存器
	XOR	XOR		WR	写专门寄存器
	XORCC	XOR，设置条件码		FLUSH	指令 cache 清空
	XNOR	异或非			
	XNORCC	异或非，设置条件码			

表 13-12　以 SPARC 寻址方式综合成其他寻址方式

指令类型	寻址方式	算法	SPARC 对应表示
寄存器到寄存器	立即寻址	操作数 = A	S2
装载，保存	直接寻址	EA = A	R_2 + S2
寄存器到寄存器	寄存器寻址	EA = R	R_{S1}，R_{S2}
装载，保存	寄存器间接寻址	EA = (R)	R_{S1} + 0
装载，保存	偏移寻址	EA = (R) + A	R_{S1} + S2

13.7.3　指令格式

与 MIPS R4000 一样，SPARC 使用了一组简单的 32 位指令格式（见图 13-14）。所有指令都以 2 位操作码开始，某些指令在指令格式中的其他位置还有操作码。对于 Call 指令，一个 30 位的立即数用右方加两个零位的方法扩展成 32 位，构成以 2 的补码格式表示的 PC 相对地址。指令

对齐在32位边界上，故这种地址形式[○]满足寻址需要了。

分支指令包括一个4位条件字段，它对应于4个标准条件码，因此可以测试这些条件的任何组合。22位PC相对地址以右方加两位0而扩展，形成24位的2的补码形式的相对地址。分支指令的一个不寻常特点是它的注销（Annul）位。若此注销位未置位时，则分支指令的直接后继指令总是被执行，不论是否发生转移。这是在许多RISC机器中都可以找到的典型延迟分支法策略，已在13.5节介绍过了（见图13-7）。但是，若注销位置位时，则仅当发生转移时才执行此分支指令之后的指令。当转移实际未发生时，处理器会注销已经取到流水线中的延迟槽指令。这个注销位很有用，因为它使编译器填充条件分支指令之后的延迟槽变得很容易。转移目标指令总放在延迟槽内，因为如果不发生转移，能自动注销此指令。采用这种办法的理由是，条件分支多半是要发生转移的。

SETHI指令是一条用于装载或保存32位值的特殊指令，这对于装载或保存地址或大的常数是一个很有用的特征。SETHI指令以它的22位立即数设置寄存器的高22位，以0填充寄存器的低10位。一条普通指令格式的指令能指定多达13位的立即数，这样的指令能用于填注寄存器剩余的低10位。装载或保存指令也可以用来实现直接寻址方式。假设要从内存位置K取一个值装入寄存器，我们可以使用如下SPRAC指令：

```
sethi   %hi(K), %r8        /* 将位置K的高22位地址装入寄存器r8
ld[%r8+%lo(K)], %r8        /* 将位置K的内容装入寄存器r8
```

宏%hi和%lo用于定义由位置的相应地址位组成的立即数。SETHI的这种使用类似于MIPS上LUI指令的使用。

浮点指令格式用于浮点运算，它要求两个源寄存器和一个目的寄存器。

最后，包括装载、保存、算术和逻辑运算的所有其他指令，都使用图13-14最下方所列的两种普通指令格式之一。一种格式使用两个源寄存器和一个目的寄存器；另一种使用一个源寄存器，一个13位立即数和一个目的寄存器。

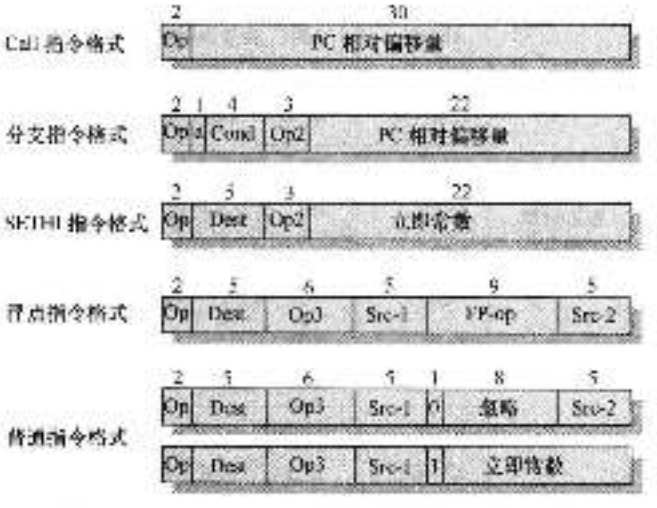


图13-14 SPARC指令格式

13.8 RISC与CISC的争论

多年来，计算机组织和体系结构朝着增加CPU复杂性方向发展：更多的指令，更多的寻址方式，更多专用寄存器等。RISC结构表示了与这种趋势背后根本原理的彻底决裂。显然，RISC系统的出现和赞美RISC优点文章的发表，遭到了参与CISC体系结构设计人员的反对。

在评价RISC方法动过方面所做的工作可分成两类。

- 定量化（quantitative）：试图比较采用了可对比技术的RISC和CISC机器上的程序大小及程序执行速度。
- 定性化（qualitative）：考察诸如高级语言支持和VLSI资源的优化使用等设计出发点。

[PATT82b，HEAT84，PATT84]这些在RISC系统上所做的研究已完成定量评价的大量工作。正如已看到的，他们原则上赞同RISC方法。其他一些人考察了设计问题，对RISC路线还是

○ 意为后补两位0，即4字节边界。——译者注

抱着一些怀疑［COLW85a，FLYN87，DAVI87］。在试图进行这类比较时，存在几个问题［SERL86］。

- 没有这样成对的 RISC 和 CISC 机器，它们在生存期、技术水平、成本、门电路的复杂性、编译器的精致性、操作系统支持等各方面都是可比较的。
- 不存在一组正式的测试程序。实际性能总是随所用测试程序不同而改变。
- 难于将硬件效应与编译器编写技巧的效应分开。
- 对 RISC 的分析比较，大都是在模拟机上完成的而不是在商品机上。更何况作为 RISC 广告的大多数商品机又都具有 RISC 和 CISC 的混合特征。于是，与标榜为纯的 CISC 商品机（例如 VAX、Pentium）进行公平比较是困难的。

定性评估几乎都是通过定义来完成的，因而是主观观点。几个研究者已将他们的注意力转到这种评估［COI，W85a，wALL85］上，但结论仍存在模糊性并遭到批驳［PATT85b］，当然，也有反批驳［COLW85b］。

近几年来，RISC 与 CISC 的争论在很大程度上已平息下来。这是因为已出现技术的逐渐交融。随着芯片密度和硬件速度的提高，RISC 系统已变得更复杂。与此同时，在达到极限性能的努力中，CISC 设计已关注通常是与 RISC 相联系的一些特点，如增加通用寄存器数量和更加强调指令流水线设计等。

13.9 推荐的读物

两篇经典的 RISC 综述文章是［PATT85a］和［HENN84］，另一篇论文章是［STAL88］。［RAD183］和［PATT82a］提供了关于 RISC 两个开拓性工作的说明。［KANE92］详细介绍了商品化的 MIPS 机器。［MIRA92］提供了很好的 MIPS R4000 综述。［BASH91］讨论了由 R3000 流水线到 R4000 超流水线的演变。［DEWA90］中详细介绍了 SPARC。

BASH91 Bashteen, A.; Lui, I.; and Mullan, J. "A Superpipeline Approach to the MIPS Architecture." *Proceedings, COMPCON Spring '91*, February 1991.

DEWA90 Dewar, R., and Smosna, M. *Microprocessors: A Programmer's View*. New York: McGraw-Hill, 1990.

HENN84 Hennessy, J. "VLSI Processor Architecture." *IEEE Transactions on Computers*, December 1984.

KANE92 Kane, G., and Heinrich, J. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1992.

MIRA92 Mirapuri, S.; Woodacre, M.; and Vasseghi, N. "The MIPS R4000 Processor." *IEEE Micro*, April 1992.

PATT82a Patterson, D., and Sequin, C. "A VLSI RISC." *Computer*, September 1982.

PATT85a Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM*. January 1985.

RAD183 Radin, G. "The 801 Minicomputer." *IBM Journal of Research and Development*, May 1983.

STAL88 Stallings, W. "Reduced Instruction Set Computer Architecture." *Proceedings of the IEEE*, January 1988.

13.10 关键词、思考题和习题

关键词

complex instruction set computer (CISC)：复杂指令集计算机
delayed branch：延迟分支
delayed load：延迟装载
high-level language：高级语言
reduced instruction set computer (RISC)：精简指令集

计算机
register file：寄存器组
register window：寄存器窗口
SPARC：可扩展处理器体系结构

思考题

13.1 RISC 组织的典型特征是什么？
13.2 简要说明 RISC 机器上用于减少寄存器-存储器操作的两种基本方法。
13.3 若用一个环形寄存器缓冲器来管理嵌套过程的局部变量，请描述管理全局变量的两种办法。
13.4 RISC 指令集体系结构的共同特征是什么？
13.5 什么是延迟分支？

习题

13.1 考虑图 4-21 中的调用-返回样式，以如下窗口尺寸将出现多少上溢和下溢（它们每个都引起一个寄存器保存/恢复）：
(a) 5? (b) 8? (c) 16?
13.2 在讨论图 13-2 时曾说过，仅窗口的前两部分需要保存或恢复。为什么没必要保存临时寄存器？
13.3 我们希望确定一给定程序的执行时间，它使用 13.5 节讨论过的各种流水策略。令：
N = 已执行指令数
D = 存储器访问次数
J = 转移指令数
对于简单的顺序策略，见图 13-6a，执行时间是 $2N + D$ 个阶段。求出两段、三段和四段流水的执行时间公式。
13.4 重新组织图 13-6d 中的代码顺序以减少 NOOP 的数目。
13.5 考虑高级语言的如下代码片段：

```
for I in 1...100 loop
  S:= S + Q(I).VAL
end loop;
```

假定 Q 是一个 32 字节记录的数组，VAL 字段是每个记录的前 4 个字节。使用 x86 代码，能将这个程序段编译成：

```
        MOV     ECX,1           ;使用寄存器 ECX,初始值为 1
LP:     IMUL    EAX,ECX,32      ;得到位移量在 EAX 中
        MOV     EBX,Q[EAX]      ;将 VAL 字段装入 EBX
        ADD     S,EBX           ;加到 S 中
        INC     ECX             ;递增 1
        CMP     ECX,101         ;与 101 比较
        JNE     LP              ;循环直到 I=100
```

这个程序使用了 IMUL 指令，它将第二个操作数乘以第三个操作数中的立即值，乘积放入第一个操作数中（见习题 10.13）。一个 RISC 拥护者证明，一个灵巧的编译器能取消像 IMUL 这样的不必要的复杂指令。通过重写一个不使用 IMUL 指令的 x86 程序来提供此证明。
13.6 考虑如下循环：

```
S:=0;
for K:=1 to 100 do
    S:=S-K;
```

将这些语句翻译成通常的汇编语言，直截了当的做法可以是这样：

```
        LD      R1,0            ;S 值保持在 R1 中
        LD      R2,1            ;K 值保持在 R2 中
LP      SUB     R1,R1,R2        ;S:=S-K
        BEQ     R2,100,EXIT     ;若 K=100,则结束
        ADD     R2,R2,1         ;否则,递增 K
        JMP     LP              ;回到循环开始处
```

RISC 机器的编译器将在这段代码中引入延迟槽，下是处理器能使用延迟分支机制。JMP 指令好处理，

因这条指令后总是跟着一条 SUB 指令，因此我们可以简单地将 SUB 指令的副本放入 JMP 之后的延迟槽。BEQ 指令处理就有些困难，我们不能让代码就这样运行，否则 ADD 指令会执行太多次。于是，需要 NOP 指令。请给出使用延迟分支法的最终代码。

13.7　为提高流水效率，RISC 机器可将符号寄存器映射到实际寄存器，并重排指令顺序。这就提出了一个有趣问题：这两个操作有没有先后次序。考虑如下程序段：

```
LD      SR1, A              ; A 装入符号寄存器 1
LD      SR2, B              ; B 装入符号寄存器 2
ADD     SR3, SR1, SR2       ; SR1,SR2 两内容相加，以及存入符号寄存器 3
LD      SR4, C
LD      SR5, D
ADD     SR6, SR4, SR5
```

(a) 先进行寄存器映射，后进行指令重排序，使用了多少机器寄存器？有流水性能的任何改进吗？

(b) 仍以原程序开始，现在是先做指令重排序，后做寄存器映射，使用了多少机器寄存器？有流水性能的任何改进吗？

13.8　请在表 13-7 中加入这两项：

(a) Pentium II　(b) AHM

13.9　多数情况下，本列为 MIPS 指令集一部分的普通机器指令能以单个 MIPS 指令来合成。请表示出如下的各 MIPS 指令序列。

(a) 寄存器到寄存器的传送　(b) 递增，递减

(c) 求补　(d) 求负　(e) 清除

13.10　一个 SPARC 实现中有 K 个寄存器窗口，它的物理寄存器数目 N 是多少？

13.11　SPARC 缺乏几条 CISC 机器上普遍有的指令，其中某些指令可使用寄存器 R0（它的值总为 0）或常数操作数来模拟而成。这些被模拟的指令称为伪指令（pseudo instruction），并被 SPARC 编译器所承认。请表示如何模拟出如下伪指令，每个只使用单一 SPARC 指令。所有这些伪指令中，src 和 dst 分别指的是源寄存器和目的寄存器（提示：保存到 R0 对 R0 无影响）：

(a) MOV src, dst　(b) COMPARE src1, src2　(c) TEST src1

(d) NOT dst　(e) NEG dst　(f) INC dst

(g) DEC dst　(h) CLR dst　(i) NOP

13.12　考虑如下程序段：

```
if  K>10
    L=X+1
else
    L=X-1;
```

这些语句翻译后，能以如下形式进入 SPARC 编译器：

```
    sethi %hi(K), %r8               ; 将位置 K 的高 22 位地址装入寄存器 r8
    ld    [%r8 + %lo(K)], %r8       ; 将位置 K 的内容装入寄存器 r8
    cmp   %r8, 10                   ; r8 的内容与 10 相比较
    ble   L1                        ; 若 (r8)≤10 则转移
    nop
    sethi %hi(X), %r9
    ld    [%r9 + %lo(X)], %r9       ; 将位置 X 的内容装入寄存器 r9
    inc   %r9                       ; 将 (r9) 加 1
    sethi %hi(L), %r10
    st    %r9, [%r10 + %lo(L)]      ; 存 (r9) 到位置 L
    b     L2
    nop
L1: sethi %hi(X), %r11
    ld    [%r11 + %lo(X)], %r12     ; 将位置 X 的内容装入存器 r12
    dec   %r12                      ; 从 (r12) 减 1
    sethi %hi(L), %r13
    st    %r12, [%r13 + %lo(L)]     ; 存 (r12) 到位置 L
L2:
```

在每个分支指令后都有一个 nop 指令，因此这些代码准许以延迟分支法来运行。

(a) 上面汇编指令代码经编译器后就可以在 SPARC 机器上运行，无需再进行其他操作。但优化的编译器对高级语言程序能完成两次翻译，先翻译成汇编语言，然后对它进行优化处理。请注意，上述汇编语言程序中两个装载指令是不必要的，并且如果保存指令挪移到程序中另一位置，则两次保存可合并成一次。请写出完成这些修改之后的程序。

(b) 如果编译器现在能够完成针对 SPARC 的特有优化，请考虑使用设置注销位的 ble 指令（表示成 ble,a L1），并将其他有用的指令移入它之后的延迟槽内，而取代 nop 指令。写出这一改动之后的程序。

(c) 现在还有两条不必要的指令，请将它们移走，写出最终优化的汇编语言程序。

指令级并行性和超标量处理器

本章要点

- 超标量处理器是一种使用多条相互独立的指令流水线的处理器。每条流水线由多个段（stage）组成，因此每条流水线能同时处理多条指令。多流水线引入了新一级并行性，允许同时处理多个指令流。超标量处理器利用了所谓**指令级并行性**（instruction-level parallelism），指令级并行性指的是程序中的指令可以并行执行。
- 超标量处理器一次取多条指令，然后试图找出几条彼此不相关因而能够并行执行的指令。如果一条指令的输入取决于前面指令的输出，则这条指令不能同时，更不能先于前面指令完成执行。一旦这种相关性被确认，处理器可以以不同于原来代码的顺序发射和完成指令。
- 通过使用更多的寄存器，或对原代码中的寄存器引用换名，处理器可取消某些不必要的相关性。
- 纯 RISC 处理器经常使用延迟分支来最大程度地利用指令流水线，然而，这种方法不太适用于超标量处理器，大多数超标量机器使用了传统的分支预测技术来提高流水线效率。

超标量实现的处理器结构是指，在这样的结构中，包括整数和浮点运算、装载、保存以及条件分支之类的普通指令，能同时启动并独立执行。这种实现引出了涉及指令流水线的几个复杂设计问题。

超标量设计紧跟 RISC 体系结构的脚步。虽然 RISC 机器的精简指令集体系结构自身已倾向于应用超标量技术，但超标量方法既能用于 RISC 也能用于 CISC 体系结构。

其实，以 IBM 801 和 Berkeley RISC I 开始的 RISC 研究到 RISC 商品机的推出，其孕育期长达 7～8 年，而最初成为商业可用的超标量机器只是超标量这个概念提出后一两年的事。超标量方法现在已成为实现高性能微处理器的标准方法。

本章先是概述超标量方法，将它与超级流水线对照。接着提出与超标量实现相关的主要设计考虑。然后考察几个最具代表性的超标量处理器实例。

14.1 概述

超标量（superscalar）这一术语最早是在 1987 年提出的［AGER87］，它指的是为改善标量指令执行性能而设计的机器。在大多数应用中，大量操作都是对标量进行的。因而，超标量方法代表了高性能通用处理器的进一步发展。

超标量方法的本质是，在不同流水线中独立执行指令的能力。此概念可进一步发展为，允许指令以不同于原程序顺序的次序来执行。图 14-1 说明了超标量方法的通用模式。这里有多个功能单元，每个单元以流水线方式来实现，它们支持几条指令的并行执行。此例中，两条整数指令、两条浮点指令和一条存储器操作（装载或保存）指令能同时来执行。

不少研究人员考察了类似超标量处理器，他们的研究指出某种程度的性能改善是可能的。表 14-1 列出了所报告的性能改进，结果的不同起因于被模拟机器硬件和被模拟应用两方面的不同。

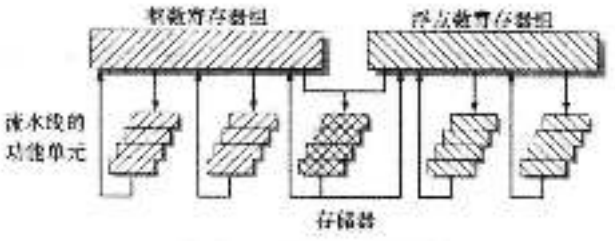


图 14-1 通用超标量结构

表 14-1 超标量机器的速度提高情况

参考文献	加速比
[TJAD70]	1.8
[KLCK77]	8
[WEIS84]	1.58
[ACOS86]	2.7
[SOHI90]	1.8
[SMIT89]	2.3
[JOUP89b]	2.2
[LEE91]	7

14.1.1 超标量与超级流水线的对比

实现更高性能的另一种方法是超级流水线（super pipelining），这一术语最早提出是在 1988 年[JOUP88]。超级流水线是利用了如下事实：多数流水阶段所完成的任务只需要比时钟周期一半还少的时间。于是，双倍的内部时钟速率允许在一个外部时钟周期内完成两个任务。我们已看过这种方法的一个例子，MIPS R4000。

图 14-2 比较了这两种方法。图的上部显示了一个普通的流水线，用做比较的基础。它是每时钟周期发出一条指令，并能每时钟周期完成一个流水段。此流水线有 4 段：取指令、操作译码、操作执行和结果写回。为清楚起见，执行段以阴影表示，注意，虽有几条指令并行执行，但任何时刻只有一条指令处于执行段。

图的中间部分表示一种超级流水线实现，它能每个时钟周期完成两个流水阶段。查看它的另一种方式是，每个流水段所完成的任务能分成两个不重叠的部分并且每个能在半个时钟周期内执行完。这种样式的超级流水线被称作级别为 2。最后，图的最下部表示的是一种超标量实现，它能并行执行每阶段的两个实例。自然，更高程度的超级流水线和超标量实现也完全是可能的。

图 14-2 所描述的超级流水线和超标量实现，二者在稳定状态下具有相同的指令数同时在执行。在程序开始和每次转移到目标时，超级流水线处理器落后于超标量处理器。

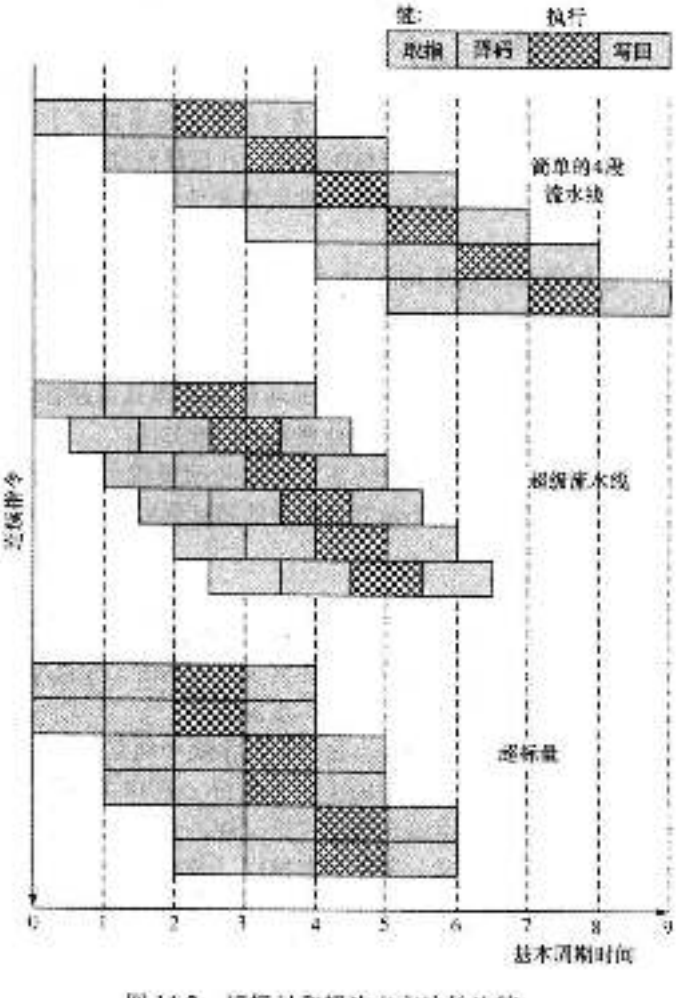


图 14-2 超标量和超流水方法的比较

14.1.2　限制

超标量方法依赖于并行执行多条指令的能力。指令级并行性（instruction-level parallelism）指的是程序指令能并行执行的程度。硬件技术与编译器优化技术的结合能够达到最大程度的指令级并行性。在考察超标量机器用于提高指令级并行性所采用的设计技术之前，我们需要仔细看并行性的基本限制。这些限制是系统必须认真对待的。参考文献［JOHN91］中列出 5 种限制：

- 真实数据相关性（true data dependency）
- 过程相关性（procedural dependency）
- 资源冲突（resource conflict）
- 输出相关性（output dependency）
- 反相关性（antidependency）

本节先考察前二个限制，后两个限制留待下一节讨论。

1. 真实数据相关性

考虑如下指令序列[⊖]：

```
ADD  EAX, ECX; 将寄存器 EAX 的内容和 ECX 的内容相加，结果保存到 EAX 寄存器中
MOV  EBX, EAX; 将寄存器 EAX 的内容保存到 EBX 寄存器中
```

第二条指令能取指并译码，但在到第一条指令执行完成之前不能被执行。原因在于第二条指令需要第一条指令产生的数据。这种情况称为真实数据相关性，也称为流相关性（flow dependency）或写后读相关性（read after write dependency，RAW）。

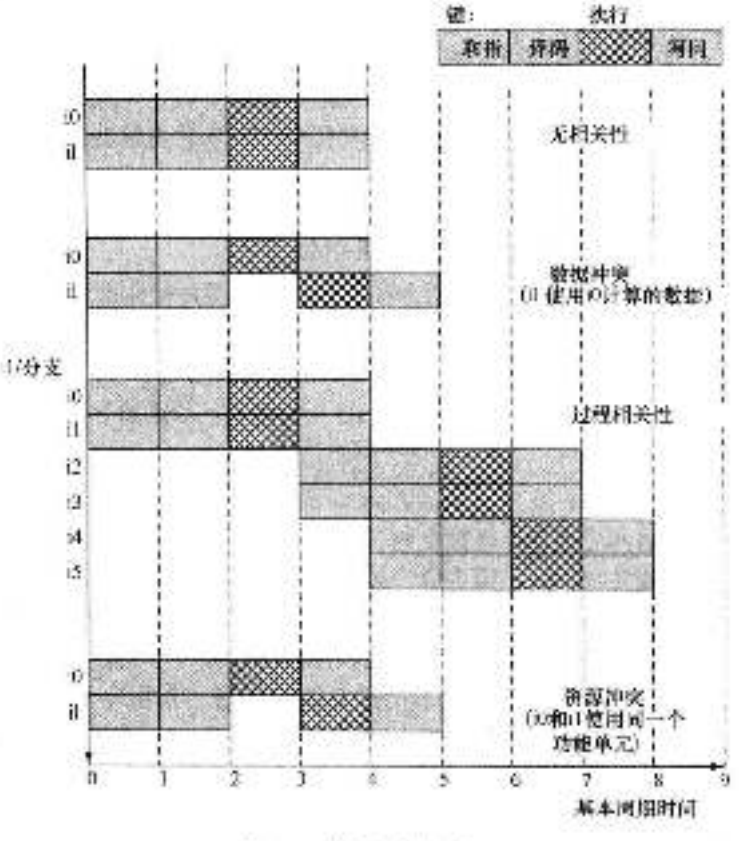


图 14-3　相关性的影响

图 14-3 说明了级别为 2 的超标量机器中的这种相关性。若没有相关性，两条指令能并行地取指和执行。若第一、第二条指令间有数据相关性存在，则第二条指令要延迟一定时钟周期以待相关性消除。通常，任何指令直到它的所有输入值都已产生之前必须被延迟。

一个简单的流水线中，例如图 14-2 上部所示的流水线，上述指令序列可能不会引起延迟。然而，考虑如下指令序列，其中一条指令从内存装载一个数，而不是从寄存器读一个数：

```
MOV EAX,eff ; 将有效存储器地址 eff 处的内容装到寄存器 EAX
MOV EBX,EAX ; 将 EAX 的内容传送到寄存器 EBX
```

⊖ 对于 Intel x86 汇编语言来说，每行的分号后为注释。

一个典型的 RISC 处理器要用 2 个或更多时钟周期来完成由存储器取数的操作，这是由于储存器不在处理器芯片上导致的延迟，或 cache 的存取延迟。补偿这种延迟的一种方法是编译器重排指令顺序，让不必等待存储器装载数据的一条或多条后续指令开始进入流水线。这种策略在超标量流水线情况下不太有效。在装载期间执行的这种不相关指令很可能在装载的第一个周期执行完毕，留下处理器无事可做直到装载完成。

2. 过程相关性

正如第 12 章所讨论的，指令序列中出现分支指令把流水操作弄复杂了。分支（发生或不发生转移）之后的指令有对分支指令的过程相关性，而且直到分支指令被执行之前它们不能去执行。图 14-3 说明了分支对 2 度超标量流水线的影响。

我们已经看过，这类过程相关性也影响简单的标量流水线。但结果对于超标量流水来说要更严重，因为每个延迟会丢失更多的机会。

若使用变长指令，会出现另一类过程相关性。因为任何指令的具体长度不是事先已知的，在取后续指令之前，它必须至少部分地被译码。这就妨碍了超标量流水所要求的指令同时取指。这也是超标量技术更适合用于 RISC 或类 RISC 结构的理由之一，因为它们的指令长度固定。

3. 资源冲突

资源冲突是两个或多个指令同时竞争同一资源。资源的例子包括存储器、cache、总线、寄存器组端口和功能单元（如 ALU 加法器）。

对于流水线而言，资源冲突展示出类似于数据相关性的行为（参见图 14-3）。然而也有些不同，资源冲突可通过复制资源来克服，而真实数据相关性是不能被取消的。还有，当操作需要较长时间来完成时，通过将相应的功能单元流水化可减轻资源冲突。

14.2 设计考虑

14.2.1 指令级并行性和机器并行性

文献［JOUP89a］对指令级并行性和机器并行性这两个相关概念指出了一个重要的区别。当指令序列中的指令是独立的，并因此能通过重叠来并行执行时，则存在**指令级并行性**（instruction-level parallelism）。

作为说明指令级并行性概念的一个例子，考虑如下两个代码片段［JOUP89b］：

```
Load R1←R2          Add R3←R3,"1"
Add R3←R3,"1"       Add R4←R3,R2
Add R4←R4,R2        Store [R4]←R3
```

左边的三条指令是独立的，并且从理论上讲这三条是可以并行执行的。相对照，右边的三条指令不能并行执行，因为第二条指令使用了第一条的结果，第三条指令又使用了第二条的结果。

代码中的真实数据相关性和过程相关性的频繁程度决定了指令级的并行性。这些因素本身又取决于指令集体系结构和应用程序。指令级并行性也可由操作延迟时间（operation latency）所确定［JOUP89a］。操作延迟时间是指，等到一条指令的结果可作为后续指令的操作数使用时，所需的等待时间称为操作延迟时间。它确定了一个数据或过程相关性将引起多长的延迟。

机器并行性（machine parallelism）是指处理器获取指令级并行性好处的能力程度。机器并行性由下面这些因素决定：它能同时取指和执行的指令数（并行流水线数），以及处理器用于找出独立指令所使用结构的速度及精巧程度。

指令级并行性和机器并行性都是提高性能的重要因素。一个不具有充分指令级并行性的程序也能取得机器并行性的个部好处。像 RISC 那样使用固定长度的指令集结构，增强了指令级并行性。从另一方面讲，有限的机器并行性将限制无论什么性质的程序的性能。

14.2.2　指令发射策略

正如所提到过的，机器并行性并不只是使每个流水段能容纳多条指令这样简单的事情。处理器必须能识别出指令级并行性，并指挥流水线并行地去取指、译码和执行。文献［JOHN91］使用了术语指令发射（instruction issue），它是指启动指令去处理器功能单元执行的过程，并用指令发射策略（instruction-issue policy）这个术语来表示启动指令执行时所采用的协议。通常，我们说指令发射是在指令从流水线的译码阶段，向流水线的执行阶段前进时发生的。

实际上，指令发射就是处理器试图在当前执行点之前查找能进入流水线并执行的指令。因此，三种类型的排序是重要的：

- 取指令的顺序。
- 指令执行的顺序。
- 指令改变寄存器和存储器位置内容的顺序。

处理器越精巧，对这些顺序间严格关系的限制就越少。对那些在严格顺序执行中所见到的次序，处理器可能需要更改一个或多个指令的次序以求做到各个流水线部件的最大化利用。对此的唯一限制是，处理器必须保证结果是正确的。于是，处理器必须协调以前所讨论的各种相关性和冲突。

通常，我们能把超标量指令发射策略分为下面这几种：

- 按序发射按序完成（in-order issue with in-order completion）。
- 按序发射乱序完成（in-order issue with out-of-order completion）。
- 乱序发射乱序完成（out-of-order issue with out-of-order completion）。

1. 按序发射按序完成

最简单的指令发射策略是，严格地按顺序执行的那个顺序发射指令（按序发射），并以同样顺序写结果（按序完成）。即使标量流水线也不遵循这种简单方式的策略。然而将它作为更复杂指令发射方法的一个比较底线还是有用的。

图 14-4a 给出这种策略的一个例子。假定超标量流水线一次能取并译码两条指令，有三个分立的功能单元（如整数算术、浮点算术等），有两个流水写回段的部件。例子是一个 6 条指令的代码片段，并假定有如下限制：

- I1 执行要求两个执行周期。
- I3 和 I4 为使用同一功能单元而发生冲突。
- I5 依赖于 I4 产生的值。
- I5 和 I6 为使用同一功能单元而发生冲突。

指令是一次取两条并传送到译码单元。因为指令是成对取，所以下两条指令必须等待，直到译码流水段已完成上次所取指令的译码。为保证按序完成，当有功能单元冲突或功能单元产生结果需要不止一个周期时，指令发射必须停止。

在这个例子中，由译码第一条指令到写回最后结果总共花费的时间是 8 个时钟周期。

2. 按序发射乱序完成

乱序完成在标量 RISC 机器中用来改善需要执行多个时钟周期指令的性能。图 14-4b 说明了它在超标量处理器上的使用。指令 I2 被允许先于 I1 完成。这就允许 I3 也能更早完成，从而节省了一个时钟周期。

采用乱序完成，任何时候可能有多条指令在流水线执行阶段运行，最大数目取决于各个功能单元之间的最大机器并行度。如果发生资源冲突、出现数据相关性或过程相关性，指令发射将被迫等候。

除上面的限制外，一种新的相关性。前面曾称为输出相关性，也称为“写后写”相关性

(write after write dependency，WAW) 出现了。以下代码片段说明了这种相关性（其中 OP 表示任何一种操作）。

```
I1: R3←R5 op R6
I2: R4←R3 + 1
I3: R3←R5 + 1
I4: R7←R3 op R4
```

指令 I2 不能先于 I1 执行，因为 I2 需要 I1 在 R3 产生的结果，这是 14.1 节所描述过的真实数据相关性的例子。类似地，I4 必须等待 I3，因为它使用 I3 产生的结果。那么 I1 和 I3 之间有什么关系呢？这里没有我们已定义的那种数据相关性。然而，若 I3 的执行先于 I1 完成，则 R3 内容的错误值将被 I4 的执行所取用。于是，I3 必须在 I1 之后完成，以产生正确的结果。如果它的结果可能会被一条需要较长时间完成的较早指令改写的话，为保证结果正确，第 3 条指令的发射必须停止。

乱序完成比按序完成要求更复杂的指令发射逻辑。另外，在处理中断和异常时也更困难。当一个中断出现时，当前点的指令执行被挂起，中断处理后再恢复。处理器必须保证这个恢复操作已考虑到下述情况：中断发生时，那些位于引起此中断的指令之后的指令可能已先行完成。

3. 乱序发射乱序完成

如果按序发射，那么处理器对指令进行译码时，遇到相关点或冲突点即停顿，这期间没有另外的指令被译码，直到冲突解决。于是，处理器不能向前查看冲突点的后续指令，而这些后续指令可能独立于已在流水线中的指令，因而可以引入流水线中。

为允许乱序发射，有必要解耦流水线的译码段和执行段。这是通过使用一个称为指令窗口 (instruction window) 的缓冲器来完成的。在这种组织方式下，处理器译完一条指令就把它放入指令窗口，只要缓冲器未满，处理器就继续取指和译码新指令。当执行阶段中的功能单元变成可用时，需要此功能单元的指令就会由指令窗口发射到执行段。只要（1）指令所需的具体功能单元是可用的，以及（2）没有冲突或相关性阻塞这条指令，那任何指令都可以被发射。

这种组织方式的结果是，处理器有先行查找的能力，允许它识别那些能放入执行段的独立指令。指令向指令窗口发射出去的次序很少遵循它们原来的程序顺序。同前面一样，唯一的限制是程序执行的结果是正确的。

图 14-4c 说明了这种策略。每周期两条指令取入译码段。由于缓冲器大小限制，每周期两条指令由译码段进入指令窗口。在这个例子中，指令 I6 先于 I5 被发射是可能的（回想一下，I5 依赖于 I4，但 I6 不这样）。于是，执行和写回两段都节省了一个周期，与图 14-4b 相比较，端到端节省了一个周期。

图 14-4c 中所表示的指令窗口仅在于说明它的作用。注意，它并

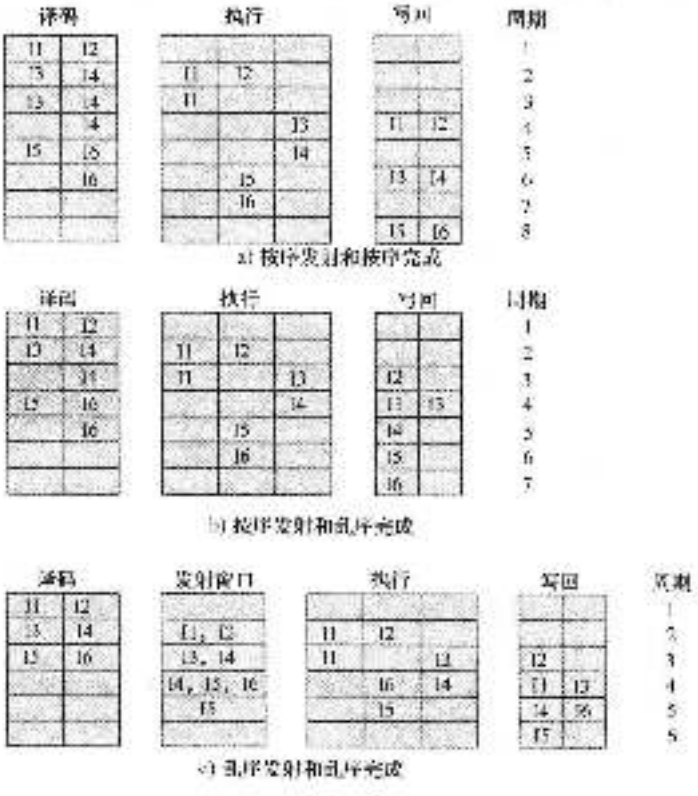


图 14-4 超标量流水线指令发射和完成策略

不是一个附加的流水线阶段。一条指令位于窗口中简单地意味着，处理器具有关于那条指令应何时发射的充足信息。

乱序发射乱序完成策略也要服从前面所描述过的限制。若一条指令违背相关性或冲突，那它不能进行发射。不同在于，有更多的指令可用来发射，减少了流水线阶段不得不停顿的发生概率。另外，一种前面称为反相关性（也称为“读后写”相关性，WAR）的问题出现了。前面曾考察过的代码段可说明这种相关性。

```
I1 :R3←R3 op R5
I2 :R4←R3 + 1
I3 :R3←R5 + 1
I4 :R7←R3 op R4
```

在指令 I2 开始执行并已取得它的操作数之前，指令 I3 不能完成执行。这是因为 I3 修改寄存器 R3，而 R3 是 I2 的源操作数。这里使用术语反相关性（antidependency），因为这一限制与真实数据相关性类似，但正好相反。真实数据相关性是前一条指令产生的值会被后一条指令使用，而反相关性是后一条指令破坏前一条指令所使用的数据值。



14.2.3　寄存器重命名

当允许乱序指令发射和/或乱序指令完成时，我们已看到，这有输出相关性（写后写相关性，WAW）和反相关性（读后写相关性，WAR）的可能性。这些相关性不同于真实数据相关性和资源冲突，后者反映了通过程序的数据流和执行的顺序，而输出相关性和反相关性的出现，从另一方面看，是因为寄存器的值可能不再反映被程序流指定的值顺序。

当指令顺序发射顺序完成时，在程序的每个执行点上确定每个寄存器的内容是可能的。当采用乱序技术时，仅考虑程序指定的指令顺序，则每点上的寄存器值不能完全已知。实际上，值对于寄存器的使用是存在冲突的，处理器必须偶尔停顿一个流水线阶段来解决这些冲突。

反相关性和输出相关性都是寄存器存储冲突的例子。多个指令为使用同一寄存器位置而竞争，产生了妨碍性能的流水限制。当寄存器优化技术被采用时，问题变得更严重，因为这些编译器技术力图最大限度地使用寄存器，于是也使寄存器存储冲突最大化。

对付这种类型的存储冲突的一种方法是基于传统的资源冲突解决方法：资源复制。在我们的语境中，此技术称为**寄存器重命名**（register renaming）。本质上，寄存器由处理器硬件动态分配，并且它们与各时间点指令所需值相关。当一个新寄存器值产生时（即当一条以寄存器为目标操作数的指令执行时），一个新寄存器分配给那个值。作为源操作数访问那个寄存器值的后续指令必须通过一个重命名过程：这些指令中的寄存器引用部分必须修改成对含有所需值寄存器的引用。于是，若准备使用不同值，不同指令中的同样原始寄存器引用可能引用到不同的实际寄存器。

让我们考虑寄存器重命名如何用于已考察过的那个代码段上。

```
I1 :R3b ← R3a op R5a
I2 :R4b ← R3b + 1
I3 :R3c ← R5a + 1
I4 :R7b ← R3c op R4b
```

不带下标的寄存器引用指的是指令中找到的逻辑寄存器。带下标的寄存器引用指向被分配用来保存新值的硬件寄存器。当对一具体逻辑寄存器进行新的分配后，作为源操作数访问那个逻辑寄存器的后续指令要修改成对最近被分配的硬件寄存器的引用（最近是依据程序的指令顺序而定）。

在这个例子中，指令 I3 中的寄存器 R3c 的生成，避免了对第一条指令的反相关性和对第一

条指令的输出相关性，而且它不影响正被 I4 访问的正确值。结果是 I3 能立即被发射；没有重命名，直到第一条指令完成和第二条指令已发射之前，I3 不能发射。

不同于寄存器重命名的另一种允许指令乱序发射的技术是记分牌（scoreboarding）。本质上讲，记分牌是一种寄存器使用登记技术，该技术允许指令乱序执行，只要指令不依赖于前面的指令，而且不存在结构冒险的时候，该指令就可以被发射。请参考附录 I 对记分牌技术的介绍。



14.2.4 机器并行性

前面已查看了能用在超标量处理器中提高性能的三种硬件技术：资源复制、乱序发射和重命名。SMIT89] 中的研究报告说明了这些技术之间的相互关系。这个研究是在有 MIPS R2000 特征的模拟器上进行的，并带有各种超标量特性的增强。研究者对几种不同的程序指令序列进行了模拟。

图 14-5 表示其结果。在每个图中，纵轴表示超标量机器与标量机器相对比的加速程度。横轴对应着 4 种不同的处理器组织。基本（base）机器不复制任何功能单元，但它可以乱序地发射指令。第二种配置（+装入/存储）是复制了访问数据 cache 的装载/保存功能单元。第三种配置（+ALU）复制了 ALU。第四种配置（+两者）是装载/保存和 ALU 都复制。每个图中的三个图柱分别对应的是 8、16 和 32 条指令的指令窗口，它们指出处理器能先行查找的指令总量。左右两图的不同在于右图允许寄存器重命名。这等于说，左图反映的机器受限于所有相关性，而右图所对应的机器只受限于真实相关性。

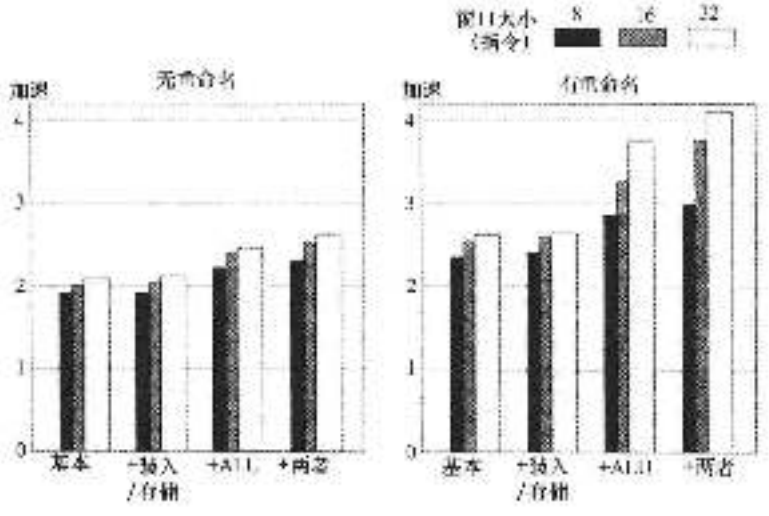


图 14-5 无过程相关性的不同机器组织结构的加速比

结合两图，能得出某些重要的结论。首先，没有寄存器重命名而添加功能单元可能不会很有价值。此时会有某些少许的性能改善，但要付出增加硬件复杂性的代价。使用寄存器重命名，取消了反相关性和输出相关性，通过添加更多的功能单元能实现显著的加速。还要注意，在实现加速的总量方面，使用 8 指令的指令窗口与更大指令窗口之间也有明显的不同。这指出，若指令窗口太小，数据相关性将妨碍额外功能单元的有效利用；处理器必须有能力足够超前地找出独立的指令，才能更全面地利用硬件。

14.2.5 分支预测

任何高性能的流水式机器必须解决分支处理问题。例如，Intel 80486 解决这个问题的方法是，既读取位于分支指令之后的下一顺序指令，又推测地读取转移目标处的指令。然而，由于取指和执行之间有两个流水段，当转移发生时这种策略会导致两个周期的延迟。



基于 RISC 机器的先进性，可采用延迟分支策略。这允许处理器在实取一些无用指令之前，先计算条件分支指令的结果。通过这种方法，处理器总是执行紧跟在分支

指令之后的那条指令。这样，在处理器读取新的指令流的同时，可保持流水线满载。

随着超标量机器的开发，延迟分支策略反而较少采用了。原因在于，多条指令需要在延迟槽中执行，会引起一些指令相关性问题。于是，超标量机器又转回到 RISC 出现以前使用的分支预测技术。某些机器，像 PowerPC 601，采用简单的静态分支预测技术。更为复杂的机器，像 PowerPC 620 和 Pentium 4，采用基于转移历史分析的动态分支预测技术。

14.2.6　超标量执行

现在，我们可对超标量的执行提供一个概述，如图 14-6 所示。将被执行的程序由一个线性指令序列组成，这是程序员编写的或编译器生成的静态程序。包括分支预测在内的取指令过程，用来形成一个动态的指令流。对此指令流进行相关性检查，处理器会解除某些人为的相关性。然后处理器派发指令进入执行窗口，在此窗口中指令不再是顺序流，而是依据它们的真实数据相关性来排序。处理器以真数据相关性和资源可用性所确定的顺序来完成每条指令的执行阶段。最后，指令的结果被登记。从概念上讲，它们是放回到原顺序序列。

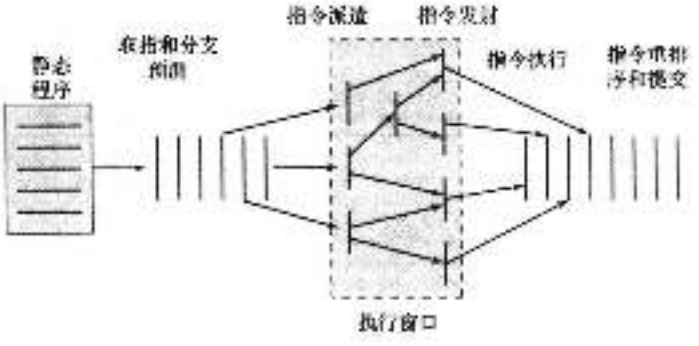


图 14-6　超标量执行的概念图

上面所提到的最后一步称为**提交**（committing）或**回收**（retiring）指令。需要此步有如下理由：首先，由于使用并行的多条流水线，指令会以不同于静态程序的顺序来完成；其次，分支预测和推测执行的使用，意味着某些指令会已完成执行但其结果需要被放弃，因为它们所在的分支没有真正发生。于是，当一条指令执行完，不能立即修改固有存储位置和程序可见的寄存器，而应将结果暂存到一个相关指令可使用的临时存储位置中。当确认顺序模型应执行此指令时，再使其结果固定化。

14.2.7　超标量实现

依上述的全面讨论，我们能对超标量方式所需要的处理器硬件予以某些一般性评论。在［SMIT95］中列出了如下关键部件：

- 同时取多条指令的取指策略，经常要有预测条件分支指令结果和超前取指的功能，这要求使用多个取指和译码流水线段，以及分支预测逻辑。
- 确定有关寄存器值真相关性的逻辑，以及执行期间把这些值与需要它们的位置之间相互联系起来的机制。
- 并行启动或发射多条指令的机制。
- 多条指令并行执行所需的资源，这包括多个流水式的功能单元，以及为多个存储器访问同时提供服务的存储器层次结构。
- 以正确顺序提交处理状态的机制。

14.3　Pentium 4

虽然超标量设计这一概念通常是与 RISC 体系结构联系在一起的，但是同样的超标量原则也能应用到 CISC 机器上。也许，这方面最著名的例子要属 Pentium 了。考察超标量概念在 Intel 产品系列中的发展情况是有益的。386 是一个传统的非流水 CISC 机器。486 是 x86 系列处理器中第

一个引入流水线的，从而使得整数操作的平均延迟从 2～4 个时钟周期减少到了 1 个时钟周期。不过 486 仍然被限制为一个时钟周期只能执行一条指令，没有超标量的部件。最初的 Pentium 有了一定的超标量能力，它使用了两个分立的整数执行单元。Pentium Pro 引入了全面的超标量设计理念。后续的 Pentium 系列处理器具有更精进、功能更强大的超标量设计。

图 4-18 给出 Pentium 4 的普通框图。图 14-7 描述同样的结构，但更适合于本节将要进行的 Pentium 4 流水线讨论。Pentium 4 的操作总结成如下：

（1）处理器以静态程序的顺序由存储器取指令。

（2）每条指令被译成一个或多个定长的 RISC 指令，称为微操作（micro-operation，micro-op）。

（3）处理器在超标量流水组织上执行微操作，因此微操作可能以乱序方式来执行。

（4）处理器以原程序流的顺序将每个微操作的执行结果提交到处理器的寄存器组。

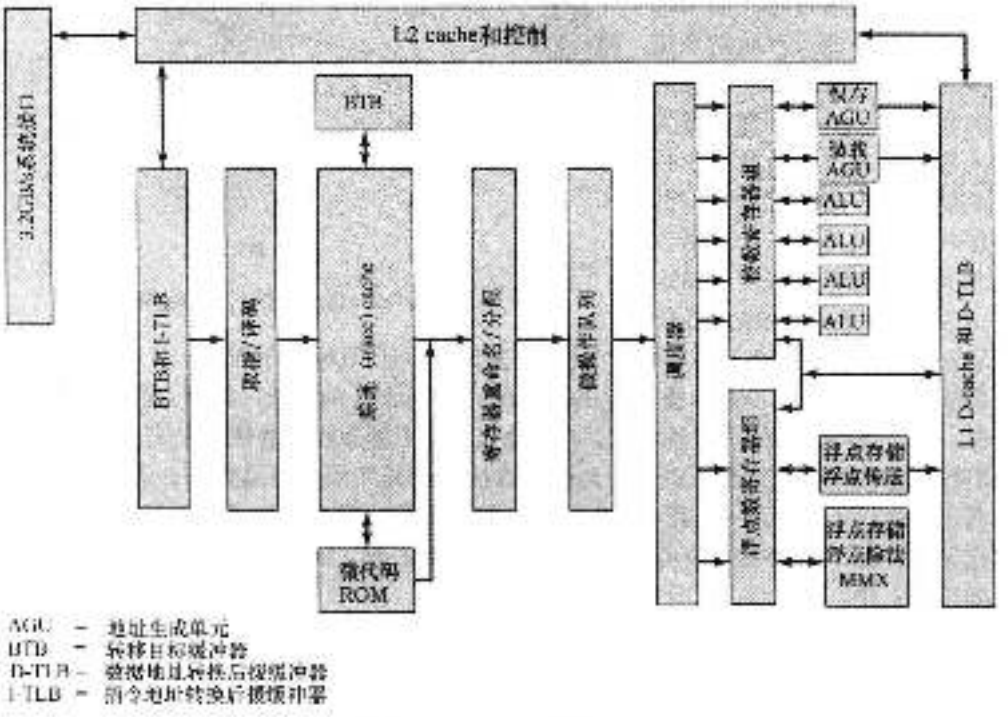


图 14-7 Pentium 4 结构图

从效果上看，Pentium 4 体系结构由外层的 CISC 壳和内部的 RISC 核所组成。内部的 RISC 微操作通过至少有 20 段（stage）的流水线（见图 14-8）。在某些情况下，微操作要求多个执行段，这导致流水线更长。这可与早期 Intel x86 处理器和 Pentium 上使用的 5 段流水线（见图 12-19）做一对照。

现在让我们使用图 14-9 逐步说明 Pentium 4 的流水线操作。

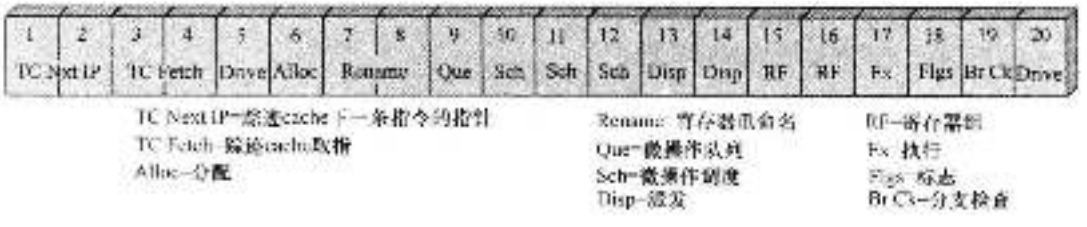


图 14-8 Pentium 4 流水线

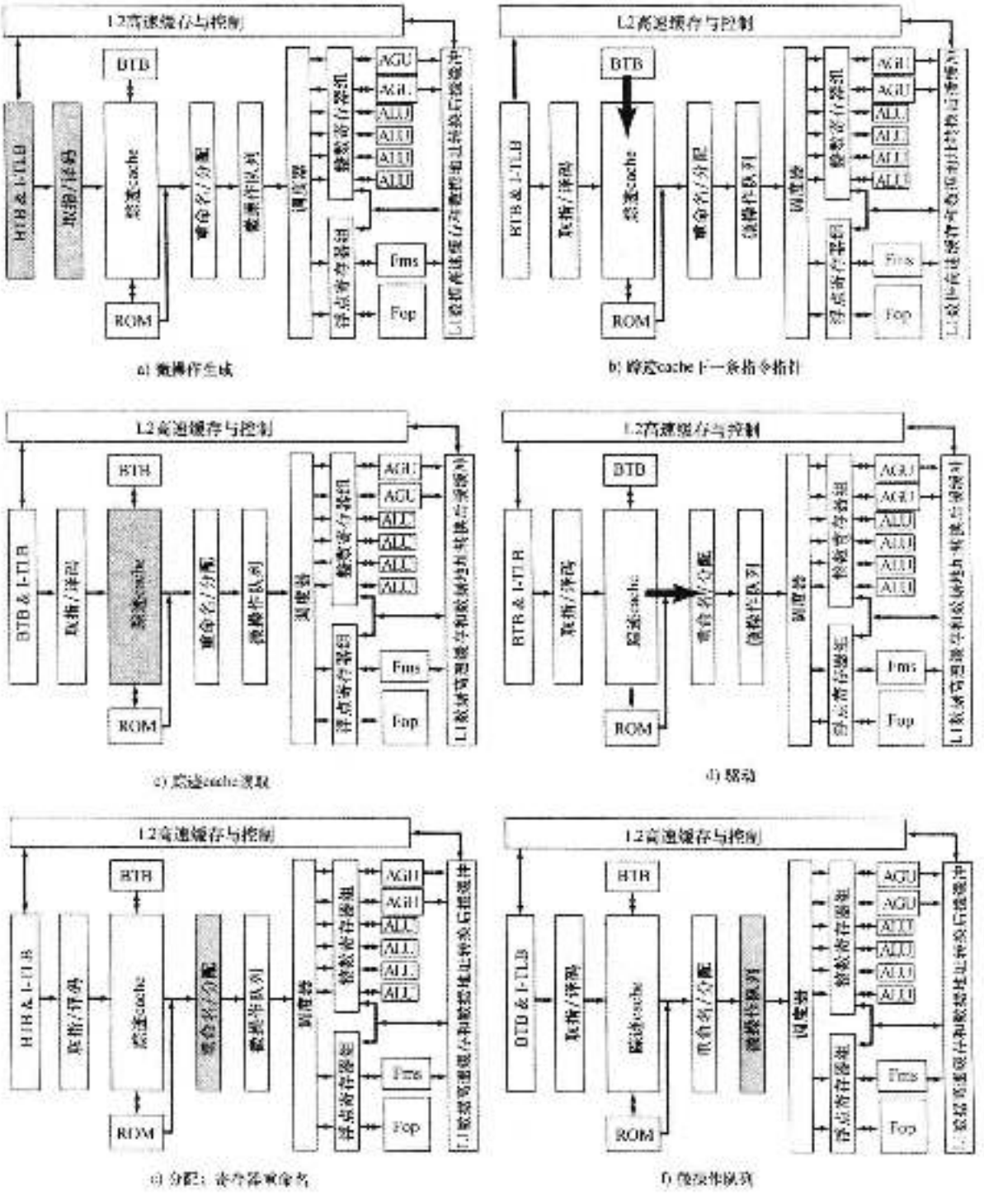
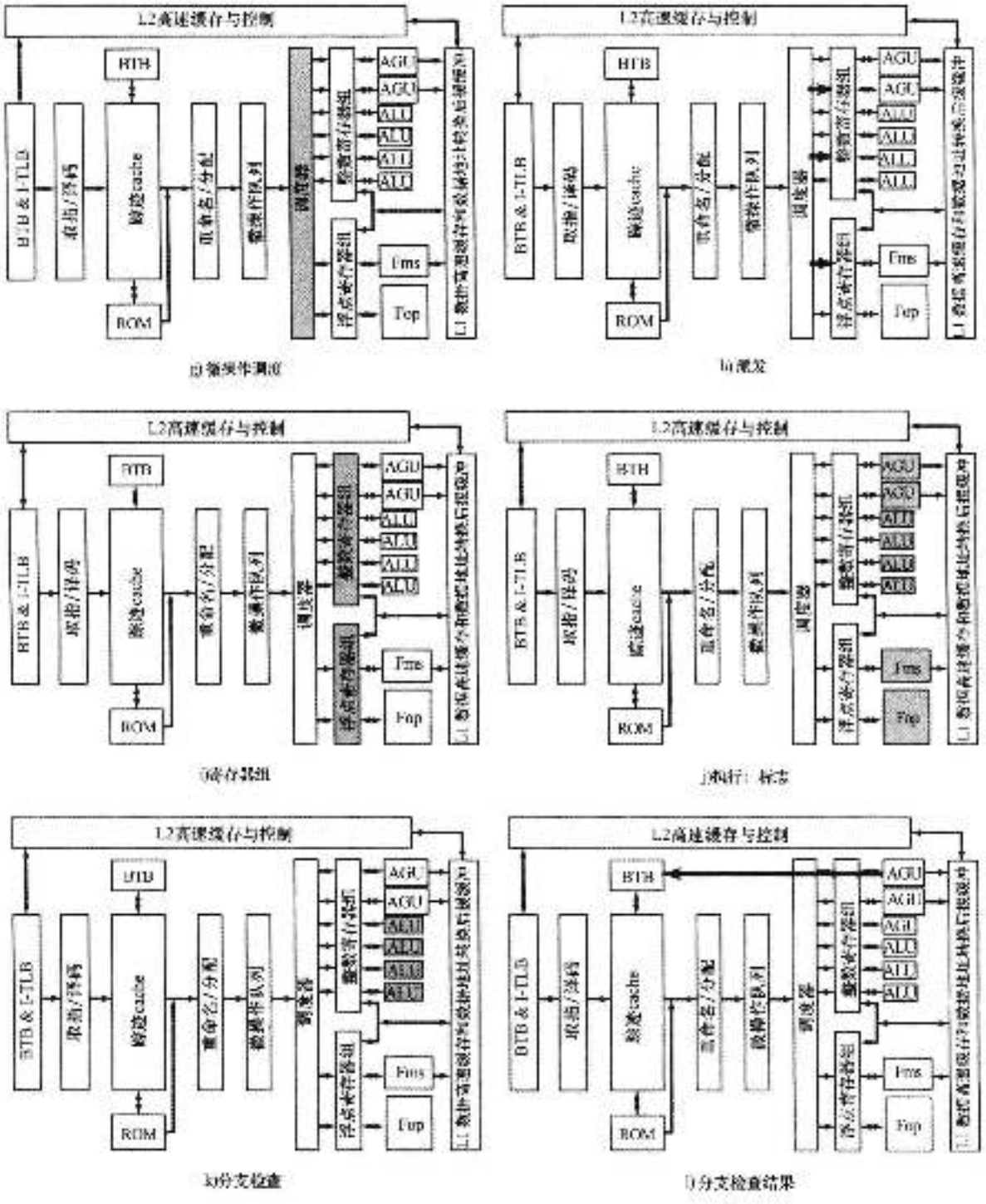


图 14-9　Pentium 4 流水线操作



i) 微操作调度

h) 派发

j) 寄存器组

j) 执行：标志

k) 分支检查

l) 分支检查结果

图 14-9 （续）

14.3.1　前端

1. 微操作生成

Pentium 4 的组织包括了一个按序操作的前端（front end）（见图 14-9a），但这部分被认为在图 14-8 所示的 Pentium 4 流水线范围之外。这个前端向一个称为踪迹 cache（trace cache）的 L1 指令 cache 提供指令。由踪迹 cache 起，才算是流水线的正式开始。通常，处理器由踪迹 cache 取操作命令；当踪迹 cache 未命中时，有序前端向踪迹 cache 提供新指令。

通过转移目标缓冲器和指令地址转换后备缓冲器（BTB&I-TLB）的支援，取指/译码单元由 L2 cache 读取 Pentium 4 的机器指令，每次 64 字节。作为默认情况，指令是顺序取指，包含下一条指令的每个 cache 行被一并取来。分支预测逻辑经由 BTB 和 I-TLB 单元可更改取指操作的顺序。I-TLB 将指令指针的线性地址转换成物理地址，以便访问 L2 cache。前端中的这个 BTB 使用静态分支预测来确定下次取哪条指令。

一旦指令被取来，先由取指/译码单元扫描字节以确定指令边界。这是必要的，因为 Pentium 4 机器指令是变长的。译码器再将每条机器指令翻译成 1 ~ 4 个微操作，每个微操作是 118 位的 RISC 指令。相比之下大多数纯 RISC 机器的指令长度只有 32 位。为适应复杂的 Pentium 4 机器指令，其微操作要求有较长的长度。尽管较长，但定长的微操作要比原来变长的机器指令易于管理。

生成的微操作存储于踪迹 cache 中。

2. 踪迹 cache

Pentium 4 流水线开始的两段是踪迹 cache 下一指令指针（TC Next IP），它负责在踪迹 cache 中选择指令，这涉及另一个不同于上述前端所采用的分支预测结构（见图 14-9b）。Pentium 4 使用基于分支指令最近执行历史的动态分支预测策略。Pentium 4 维护有一个转移目标缓冲器（BTB），它缓存了最近遇到的分支指令执行情况的相关信息。每当在指令流中遇到一条分支指令，该结构就去检查 BTB。若相应的项已在 BTB 中，则以此项的历史信息为指导确定是否应预测转移发生。若是，则与此项相关联的转移目标地址作为下一指令指针；若否，则顺序取下一指令。

一旦转移指令被执行，相应项的历史信息被修改以反映该指令的本次执行结果。如果所遇到的分支指令在 BTB 中没有相应项，则这条指令的地址装入 BTB 中的一项，如果有必要，则先删除一个旧的项。

一般来说，上面的描述符合最初的 Pentium 型号以及包括 Pentium 4 在内的后来型号所使用的分支预测策略。然而，最初的 Pentium 只使用相对简单的 2 位历史位；后来的 Pentium 型号由于有更长的流水线（Pentium 4 有 20 段，Pentium 只有 5 段），预测失误所带来的性能损失也就更大。因此后来的 Pentium 4 型号使用了更多的历史位，以更精细的分支预测算法来降低预测失误率。

Pentium 4 的 BTB 组成一个 4 路组关联 cache，共有 512 行。每项使用分支指令地址作为标记（tag），每项还包括 4 位历史字段，以及转移最后一次发生时的转移目标地址。Pentium 4 使用 4 位历史位，就能在转移预测时考虑到更长的历史状况，所用的算法称为 Yeh 算法［YEH91］。此算法的研发者已用实例验证了，与仅使用 2 位历史位的算法相比，该算法能显著地降低预测失误率［EVER98］。

在 BTB 中无历史记录的条件分支指令，采用静态预测算法。转移与否根据如下规则来预测：

- 对于转移地址不是 IP 相对寻址的条件分支指令，如果该分支指令是一个返回，则预测发生，否则预测不发生。
- 对于 IP 相对寻址的后向条件分支指令，预测转移发生。这个规则反映了典型的循环行为。
- 对于 IP 相对寻址的前向条件分支指令，预测转移不发生。

3. 踪迹 cache 取指

踪迹 cache（见图 14-9c）读取已由指令译码器翻译成的微操作，并将它们装配成按程序顺序的微操作序列。此序列被称为踪迹。踪迹 cache 按顺序取微操作，服从于分支预测逻辑。

少数指令要求多于 4 个的微操作，这些指令被传送到微代码 ROM。它为每条复杂的机器指令保存了一个对应的微操作串组（5 个或更多的微操作）。例如，一条串操作机器指令可由 ROM 翻译成很多（甚至上百次）重复的微操作串组。于是，微代码 ROM 实际上是一个微程序式控制器（它将在第 4 部分介绍）。在微代码 ROM 完成为当前复杂指令产生微操作后，又恢复到由踪迹 cache 读取微操作。

4. 驱动

Pentium 4 流水线的第 5 段（见图 14-9d）称为驱动（drive），它负责将译码后的指令由踪迹 cache 送交给重命名/分配模块。

14.3.2 乱序执行逻辑

处理器的这部分将重排序微操作，以允许它们只要输入操作数一旦就绪就可快速地被执行。

1. 分配

流水线的分配（allocate）段（见图 14-9e）为微操作的执行分配资源。它完成如下功能：

- 每时钟周期有三个微操作到达分配器。如果其中某个微操作所需的寄存器这类资源不可用，则分配器停顿流水线，直到三者所需资源都可用。
- 分配器要为微操作在重排序缓冲器（reorder buffer，ROB）中分配一项。此 ROB 共有 126 项，每项跟踪一个微操作执行过程中的完成状况。[⊖]
- 分配器要为微操作的结果数据在 128 个整数或浮点寄存器组中分配一项，以及可能为流水线中的装载（可多达 48 个）和保存（可多达 24 个）微操作分配一个装载或保存缓冲器。
- 分配器要为微操作在调度器前沿的两个微操作队列中的一个分配队列项。

ROB 是一个环形缓冲器，能保存多达 126 个的微操作，并含有 128 个硬件寄存器。每个缓冲器项由下列字段组成。

- 状态（State）：指示此微操作是否已被调度、派发、完成执行、回收（retirement）就绪等。
- 存储器地址（Memory Address）：产生此微操作的 Pentium 4 指令地址。
- 微操作（Micro - op）：实际的操作。
- 别名寄存器（Alias Register）：若微操作引用了机器体系结构 16 个寄存器的某一个，则此字段将该引用重定向到 128 硬件寄存器的某一个。

微操作按序进入 ROB，然后由它出发去排队、被调度、被派发以及去执行，这些都将是乱序的。最后 ROB 的微操作登记项要按序回收。为实现按序回收，已完成的微操作项打上回收就绪标志。然后由最早微操作最先回收的顺序回收这些已标记的微操作。

2. 寄存器重命名

在寄存器重命名（register renaming）段（见图 14-9e）将对 16 个体系结构寄存器（8 个浮点寄存器加上 EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP）的引用重新映射到 128 个物理寄存器。这样，就解除了由体系结构寄存器数量有限引起的虚假数据相关性，与此同时仍保留了真实数据相关性（写后读，RAW）。

⊖ 关于重排序缓冲器的介绍请见附录 I。

3. 微操作排队

在资源分配和寄存器重命名之后，微操作进入流水线的第 9 段：微操作排队（micro－op queuing）。见图 14-9f。微操作被放入两个队列之一，然后保持在那里直到调度器去取出它们。两个队列一个用于存储器操作（装载和保存），一个用于不涉及存储器访问的其他微操作。每个队列遵循先进先出（FIFO）规则，但队列间不维护一个次序。也就是说，一个微操作是否出队与另一队列的微操作没有次序关系，这给调度器提供了更大的灵活性。

4. 微操作调度和派发

调度器（见图 14-9g）负责由队列取出微操作并派发它们去执行。调度器查找那些其状态指明已具备自己全部操作数的微操作，若它所需的执行单元可用，则调度器取出此微操作，并将它派发到相应的执行单元（见图 14-9h）。每周期能派发多达 6 个的微操作。如果多个微操作要使用同一个执行单元，调度器将按队列顺序逐个派发它们。这也是一种 FIFO 规则，确保按序执行；但此时指令流已被相关性和分支重新排列了，实际上它已是乱序了。

调度器有 4 个端口与执行单元连接。端口 0 用于整数和浮点运算，但简单整数运算不在其内。端口 1 用于简单整数运算和分支预测失误处理。另外，几个 MMX 执行单元有的在端口 0，有的在端口 1，余下的两个端口分别用于存储器装载和保存。

14.3.3　整数和浮点执行单元

整数和浮点寄存器组是执行单元待完成操作的数据源之一（见图 14-9i）。执行单元由寄存器组以及 L1 数据 cache 取出所需的值。一个单独的流水线阶段（见图 14-9j）专门用于计算标志（如零、负等），这些值一般都是分支指令所需要使用的值。

下一个流水线阶段完成分支检查（见图 14-9k），它将分支的实际结果与预测进行比较。在最后的驱动（drive）阶段期间实现分支检查结果（见图 14-9l）。如果预测是错的，那么在各个阶段正在进行的微操作必须从流水线中清除掉。正确的目标地址提供给分支预测器（branch predictor），从而由新的目标地址重新启动整个流水线。

14.4　ARM CORTEX-A8

ARM 体系结构最近的实现已经开始在指令流水线中采用超标量技术。在本节中，我们将集中讨论 ARM Cortex－A8，它提供了一个基于 RISC 的超标量设计的很好实例。

Cortex-A8 在 ARM 系列处理器中被称为是应用处理器。ARM 公司的应用处理器是指那些运行复杂操作系统的嵌入式处理器，它们主要的应用是无线通信、消费电子以及图像处理等。Cortex-A8 的目标定位于各种移动和消费电子应用，包括手机、机顶盒、游戏机以及汽车导航/娱乐系统。

图 14-10 显示了 Cortex-A8 处理器的逻辑结构，突出了其中各个功能单元之间的指令流。主要的指令流在 3 个功能单元之间，这 3 个功能单元实现了一个有 13 个阶段，按序发射的双流水线。Cortex 设计人员决定采用按序发射是为了使所需的功耗保持最低。乱序发射和完成需要大量的逻辑电路来实现，从而消耗更多的电能。

图 14-11 显示了 Cortex-A8 主流水线的详细结构。Cortex-A8 另外还有一个单独的 SIMD（单指令多数据，single-instruction-multiple-data）单元。该单元被实现为一条 10 阶段的流水线。

14.4.1　指令取指单元

指令取指单元（instruction fetch unit）预测指令流，从 L1 指令高速缓存中取指，并把取来的指令放到缓冲器中，以便译码流水线对指令进行译码。L1 指令高速缓存包含在指令取指单元中。由于流水线中允许有若干未确定的分支指令，因此指令取指是推测性的。这意味着取来的指令

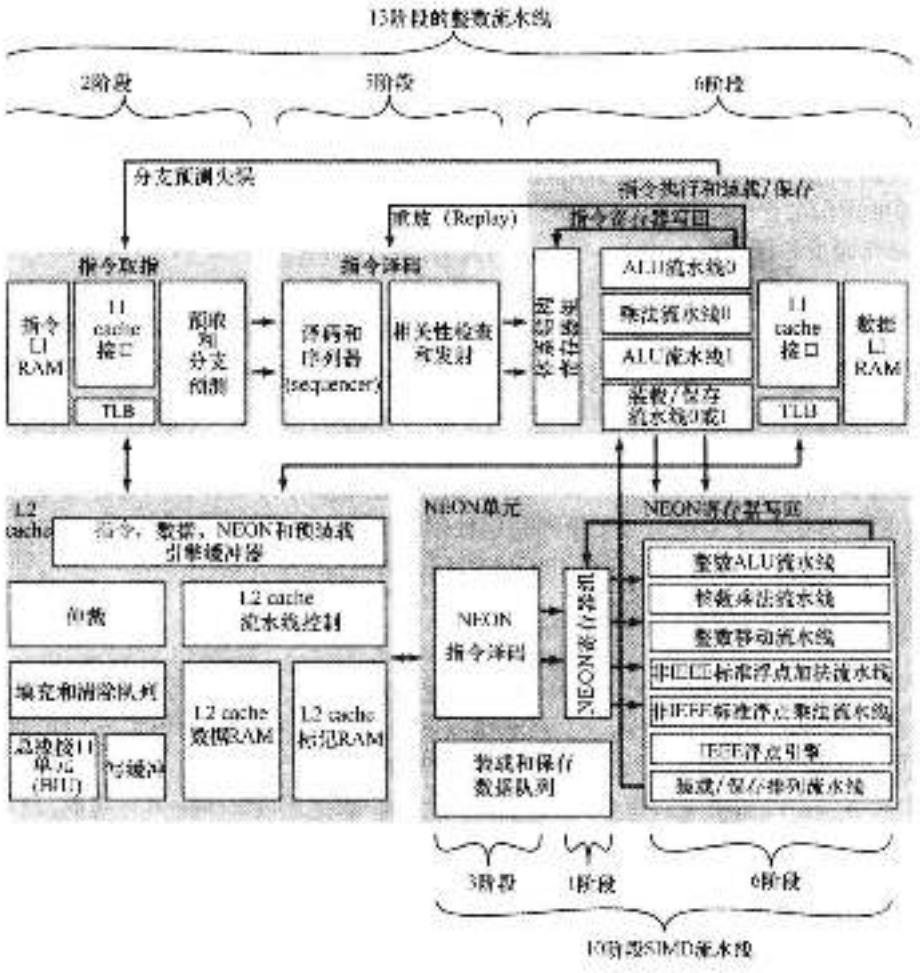


图 14-10 ARM Cortex-A8 处理器的结构图

不一定会被执行。代码流中的分支指令和发生异常的指令会导致流水线清空，丢弃当前取来的指令。指令取指单元每周期可以取来多达 4 条指令，取指操作经过如下这些阶段：

F0 地址生成单元（Address Generation Unit，AGU）生成一个新的虚拟地址。通常这个地址是上一个地址的顺序后继地址。它也可以是分支转移目标地址，该地址由分支预测器对前一条指令的预测而产生。F0 不作为一个阶段计入 13 阶段流水线，因为 ARM 处理器传统上把指令高速缓存访问当作流水线的第一阶段。

F1 计算得到的地址用于从 L1 指令高速缓存中取指。与此同时，该地址也用于访问分支预测阵列，以便确定下一个取指地址是否应该基于分支预测产生。

F3 取来的指令被放到指令队列中。如果一条指令引起了分支预测动作，那么新的目标地址会被送到地址生成单元。

为尽量减少由较深流水线带来的较大转移开销，Cortex-A8 处理器实现了一个两级全局分支预测器。该预测器由转移目标缓冲器（Branch Target Buffer，BTB）和全局历史缓冲器（Global History Buffer，GHB）组成。这些数据结构在指令取指的同时被并行访问。BTB 指出当前取指地址是否会是一个分支指令，并给出对应的分支转移目标地址。BTB 包含 512 项。如果取指地址与其中一项匹配，就触发一个分支预测动作，并将使用到 GHB。GHB 包含 4096 个 2 位的计数器。这些计数器记录了分支转移的方向及强度信息。GHB 使用最近 10 次分支转移方向的一个 10 位

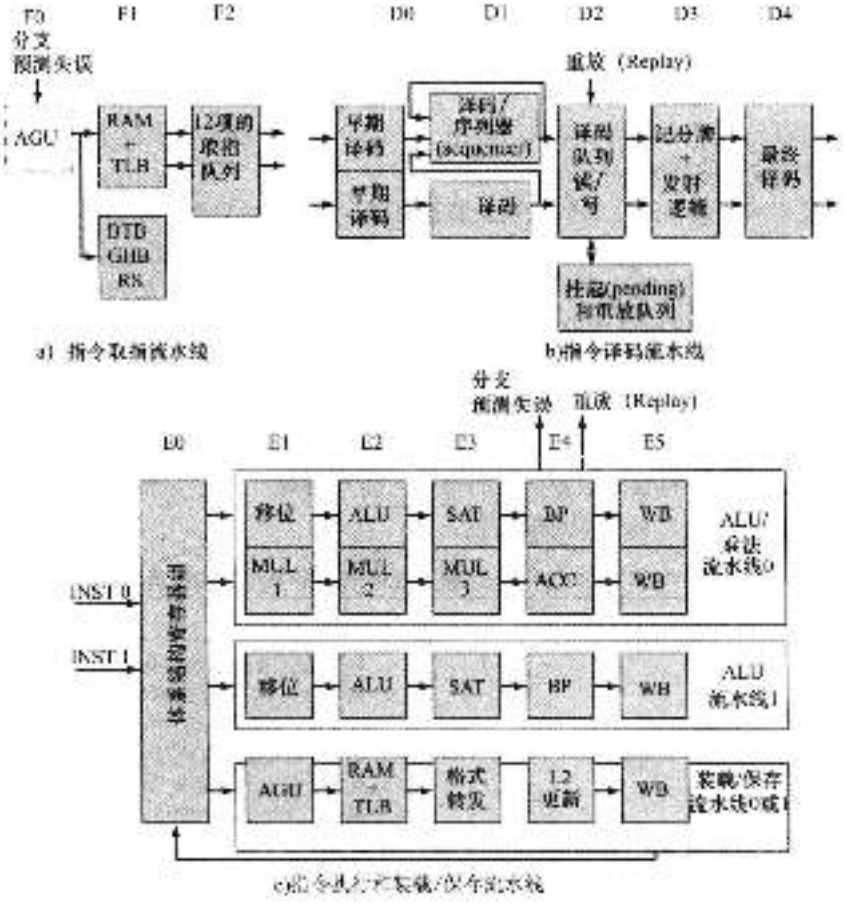


图 14-11　ARM Cortex-A8 整数流水线

历史记录和 PC 中的 4 位一起作为索引。除了动态分支预测器之外，取指单元还使用了一个返回栈，来预测子过程返回地址。返回栈有 8 个 32 位的项，每项保存了一个连接寄存器 r14 中的值，以及调用函数的 ARM 指令或压缩（Thumb）指令状态。当一个返回类型指令被预测为要发生转移时，返回栈就提供最后被压入栈的地址和状态。

指令取指单元可以取指和入队多达 12 条指令，并能在同一时间发射两条指令到译码单元。指令队列使得指令取指单元能够先于整数流水线其他阶段而预取指令，形成一批积压的指令等待译码。

14.4.2　指令译码单元

指令译码单元对所有的 ARM 指令和压缩指令进行译码并排序。译码单元有一个双流水线结构，称为流水线 0 和流水线 1。这样在同一时间，可以有两条指令通过译码单元。指令译码单元发出两条指令时，流水线 0 总是包含了程序顺序中靠前的那条指令。这意味着如果流水线 0 中的指令不能发射的话，那么流水线 1 中的指令也不会发射。一旦发射，所有被发射指令按序进入到执行流水线，并在执行流水线末尾把结果写入到寄存器组中。这种按序发射，按序完成的方式避免了 WAR 冒险，同时能直接记录 WAW 冒险和从流水线清空条件中恢复。这样，指令译码单元主要的考虑就是如何避免 RAW 冒险了。

每条指令将通过如下 5 个阶段的操作。

D0　压缩指令被解压缩到 32 位的 ARM 指令。初始的译码功能被执行。

D1 继续完成指令译码功能。

D2 这一阶段把译码后的指令写入等待/重放指令队列，并从等待/重放队列中读出指令送往下一阶段。

D3 这一阶段包含了指令调度逻辑。其中一个记分牌根据静态调度技术[2]预测寄存器的可用性。本阶段同时检查各种情况的冒险。

D4 完成最后的译码，产生整数执行及装载/保存单元需要的所有控制信号。

在最开始的两个阶段，将确定指令类型、源操作数和目的操作数，以及指令的资源需求。ARM 指令中有一些称为多周期指令的不常使用的指令。D1 阶段会把这些指令分开成为多个指令操作码，这些指令操作码将被分别排序通过执行流水线。

等待（pending）队列起到两个作用。首先，它避免来自 D3 阶段的流水线停顿信号进一步扩散从而影响流水线的运行。指令在 D3 阶段进行静态调度，调度是基于对源操作数何时可用的预测。存储系统的任何停顿会导致一个不少于 8 个周期的延迟。这个最小 8 周期的延迟对应于 L1 装载缺失时，从 L2 高速缓存接收数据所需的最少可能周期数。表 14-2 给出了因为存储系统停顿可能导致指令重放（replay）的最常见的情况。

表 14-2 Cortex-A8 存储系统停顿对指令时序的影响

重放事件	延迟	说明
装载数据缺失	8 周期	1. L1 数据高速缓存的装载指令缺失 2. 接下来向 L2 数据高速缓存发出访问请求 3. 如果 L2 数据高速缓存也发生缺失，那么会导致第二个重放。停顿周期数取决于外部系统存储器的时序。当 L2 高速缓存缺失时，接收关键字所需的最小时间大约是 25 个周期，不过这个时间可能会因为 L2 高速缓存的延迟而更长
数据 TLB 缺失	24 周期	1. 由于 L1 TLB 缺失导致的页表查找操作需要 24 个周期的延迟，如果要查找的地址转换表项不在 L2 高速缓存中的话 2. 如果要查找的地址转换表项不在 L2 高速缓存中，停顿的周期数将取决于外部系统存储器的时序
保存缓冲满	8 周期加上清空缓冲的延迟	1. 保存指令的缺失不会导致任何停顿，除非保存缓冲满了 2. 当保存缓冲满的时候，延迟至少是 8 个周期，如果清空保存缓冲中某些项的时间花费较多，那么延迟时间就会更长
未对齐的数据或保存请求	8 周期	1. 如果数据指令使用的地址是未对齐的，并且整个访问不包含在 128 位的界限内，那么延迟是 8 个流周期 2. 如果保存指令使用的地址是未对齐的，并且整个访问不包含在 64 位的界限内，那么延迟是 8 个流周期

为了处理这些停顿，指令译码单元使用了一个恢复机制，该机制先清空执行流水线中所有的后续指令，然后重新发射（重放）它们。为支持重放，指令在发射之前会被拷贝到重放队列中，直到它们写回了它们的执行结果并完成，才会从重放队列中删除。如果重放信号被置位，指令就从重放队列中读出，并重新放入流水线。

译码单元并行地发射两条指令到执行单元，除非译码单元碰到了发射限制情况。表 14-3 显示了最常见的限制情况。

---

[2] 记分牌技术的讨论请见附录 I。

表 14-3　Cortex-A8 双发射的限制

限制类型	说　明	示　例	周　期	原　因

装载/保存资源冒险	只有一条 LS（加载/保存）流水线。每个时钟周期只能发射一条 LS 指令。该指令可以进入流水线 0 或流水线 1	LDR r5, [r6]
		STR r7, [r8]
		MOV r9, r10
			1
			2
			2
				
				等待装载/保存单元
				双发射是可能的

乘法资源冒险	只有一条乘法流水线，而且只在流水线 0 中	ADD r1, r2, r3
		MUL r4, r5, r6
		MUL r7, r8, r9
			1
			2
			3
				
				等待流水线 0
				等待乘法单元

分支资源冒险	每个时钟周期只能执行一条分支指令。该分支指令可进入流水线 0 或流水线 1。分支指令是所有那些改变 PC 寄存器的指令	BX r1 BEQ 0x1000 ADD r1, r2, r3	1 2 2	 等待分支 双发射是可能的

数据输出冒险	有相同目标的指令不能在同一个时钟周期发射。有相同目标的情况可能发生在条件码的执行中	MOVEQ r1, r2
		MOVNE r1, r3
		LDR r5, [r6]
			1
			2
			2
				
				因输出相关而等待
				双发射是可能的

数据冒险	即使指令所需的数据还未就绪，指令也可以被发射。此时将检查调度表，看看源操作数需求和各个流水线结果	ADD r1, r2, r3
		ADD r4, r1, r6
		LDR r7, [r4]
			1
			2
			4
				
				等待 r1
				等待 r4 两个周期

多周期指令	多周期指令必须发射到流水线 0 中，而且只能在这些指令的最后一次重复执行时进行双发射	MOV r1, r2
		LDM r3, {r4-r7}
		LDM（周期 2）
		LDM（周期 3）
		
		ADD r8, r9, r10
			1
			2
			3
			4
			
			4
				等待流水线 0，传送 r4
				传送 r5，r6
				传送 r7
				对最后一次传送双发射是可能的

14.4.3　整数执行单元

指令执行单元由两个对称的算术逻辑单元（ALU）流水线，其中一个服务于装载/保存指令的地址生成器，以及乘法流水线组成。执行流水线也进行寄存器写回操作。下面是指令执行单元的功能：

- 执行所有的整数 ALU 和乘法操作，包括标志的生成。
- 为装载和保存指令生成虚拟地址。如果需要，生成写回基值（base write-back value）。
- 为保存指令提供格式化后的数据，并转发数据及标志。
- 处理分支以及其他对指令流的改变，并计算指令条件码。

对于 ALU 指令，可以使用两条流水线的任何一条，其执行包括如下阶段：

E0　访问寄存器组。对于两条指令，最多需要从寄存器组读出 6 个寄存器值。

E1　如果需要的话，桶移位器（见图 12-25）执行移位操作。

E2　ALU（见图 12-25）执行算术逻辑运算。

E3　如果需要的话，该阶段完成某些 ARM 数据处理指令所使用的饱和运算。

E4　如果控制流发生任何改变，包括分支预测失误、异常以及存储系统重放，那么该阶段保证这些情况被优先处理。

E5　ARM 指令的执行结果被写回到寄存器组。

使用乘法单元（见图 12-25）的指令被放入到流水线 0 中处理。乘法操作在 E1 阶段到 E3 阶段进行，乘积累加在 E4 阶段执行。

装载/保存流水线与整数流水线并行运行。装载/保存流水线包括如下阶段：

E1　存储地址从基址和变址寄存器产生。

E2　地址被用于高速缓存索引的访问。

E3 对于装载指令，数据被返回并被格式化，以使转发给 ALU 或乘法单元。对于保存指令，数据被格式化，以便写入到高速缓存中。

E4 如果需要，对 L2 高速缓存进行更新。

E5 ARM 指令的执行结果被写回到寄存器组。

表 14-4 显示了一个示例代码片段，并指出了处理器可能会如何来调度它。

表 14-4 Cortex-A8 整数流水线的示例双发射指令序列

周 期	程序计数器	指 令	时 序 说 明
1	0x00000e80	BX r14	双发射流水线 0
1	0x00000ee4	CMP r0, #0	双发射流水线 1
2	0x00000ee8	MOV r3, r5	双发射流水线 0
2	0x00000eec	MOV r0, #0	双发射流水线 1
3	0x00000ef0	STREQ r8, [r1, #0]	双发射流水线 0，r8 直到 E3 阶段才需要
3	0x00000ef4	CMP r2, #4	双发射流水线 1
4	0x00000ef8	LDRLS pc, [pc, r2, LSL, #2]	单发射流水线 0，+1 周期装载到 PC，由 LSL #2 后无额外移位周期
5	0x00000f2c	MOV r0, #1	双发射，流水线 1 中是装载的第 2 次迭代
6	0x00000f30	B [pc] +8	#0x2f38 双发射流水线 0
7	0x00000f38	STR r0, [r1, #0]	双发射流水线 1
7	0x00000f3c	LDR pc, [r13], #4	单发射流水线 0，+1 周期装载到 PC
8	0x0000017c	ADD r2, r4, #0xc	双发射，流水线 1 中是装载的第 2 次迭代
9	0x00000180	LDR r0, [r6, #4	双发射流水线 0
9	0x00000184	MOV r1, #0xa	双发射流水线 1
12	0x00000188	LDR r0, [r0, #0]	单发射流水线 0，r0 在 E3 阶段产生，而 E1 阶段就需要，因此 +2 周期停顿
13	0x0000018c	STR r0, [r4, #0]	单发射流水线 0，原因是装载/保存资源冒险，从 r0 在 E3 阶段产生并被使用之后无额外延迟
14	0x00000190	LDR r0, [r4, #0xc]	单发射流水线 0，原因是装载/保存资源冒险
15	0x00000194	LDMFD r13!, {r4 - r6, r14}	多次装载操作，第 1 周期装载 r4，第 2 周期装载 r5 和 r6，第 3 周期装载 r14，共计 3 周期
17	0x00000198	B [pc] +0xdb8	#0x1f40 双发射流水线 1，其中 LDM 指令处于第 3 周期
18	0x00000f40	ADD r0, r0, r2 ARM	单发射流水线 0
19	0x00000f44	ADD r0, r1, r0 ARM	单发射流水线 0，由于 r0 在 E2 才能产生，而同时在 E2 阶段需要用到，因此无法进行双发射

14.4.4 SIMD 和浮点流水线

所有的 SIMD 和浮点指令都通过整数流水线并由一个单独的 10 阶段流水线处理（见图 14-12）。这个称为 NEON 的单元，能处理压缩的（packed）SIMD 指令，并提供了两种类型的浮点处理支持。根据实际实现，该单元可能带有一个向量浮点（vector floating-point，VFP）协处理器，负责完成遵循 IEEE 754 标准的浮点操作。如果实际实现中不带这个协处理器，那么会有一条单独的乘法和加法流水线来实现浮点运算。

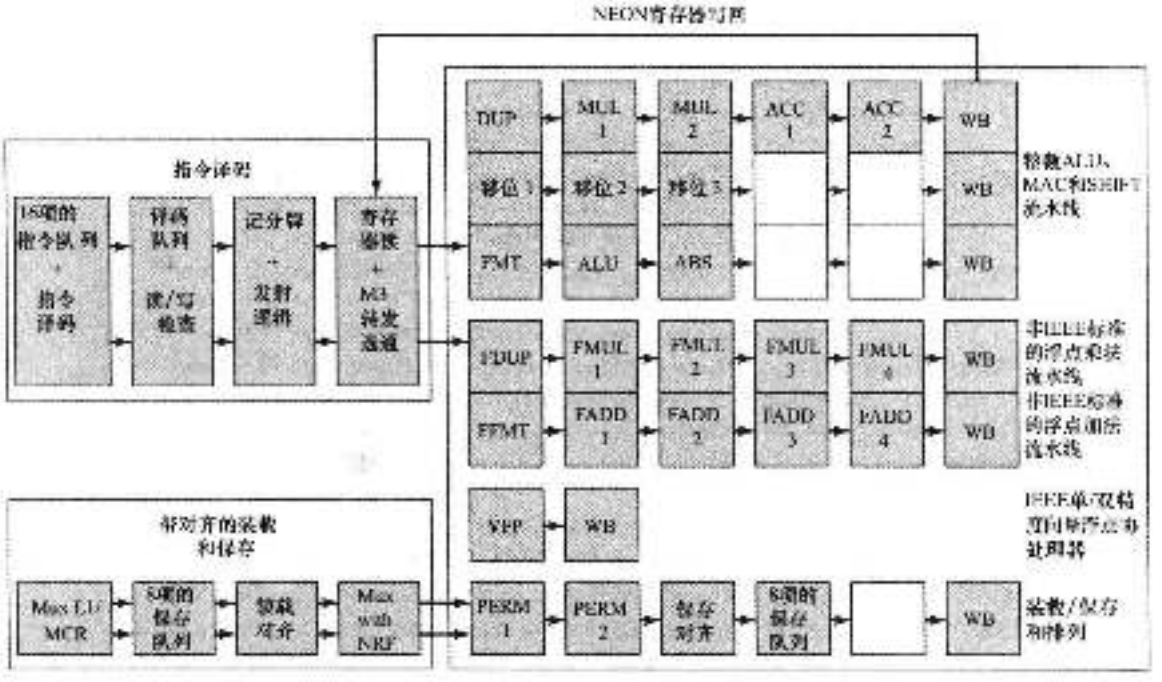


图 14-12　ARM Cortex-A8 的 NEON 和浮点流水线

14.5　推荐的读物

[SHEN05] 和 [OMON99] 两本书论述了超标量设计。这一主题的相当好的综述文章是 [SMIT95] 和 [SIMA97]。[JOUP89a] 考察了指令级并行性，查看了最大化并行度的各种技术，并使用模拟比较了超标量和超级流水线方法。[SIMA04]、[PATT01] 和 [MOSH01] 是三篇介绍超标量设计的近期文章。

[POPE91] 提供了对一个推荐的超标量处理器的详细介绍，还提供了关于乱序指令发射设计考虑的一个很有益的指导。其他对推荐系统的考察可在 [KuGA91] 中找到，这篇文章提出并考虑了超标量实现的最重要的设计观点。[LEE91] 考察了能用来增强超标量性能的软件技术。[WALL91] 是一个关于超标量处理器中端开发指令级并行性范围的有意义研究。

[INTE04] 的卷 1 提供了对 Pentium 4 流水线的一般性描述。更详细的了解请参见 [INTE01a] 和 [INTE01b]。另外还有 [FOGG08b]。

[JOHN08] 和 [ARM08a] 提供了对 ARM Cortex-A8 流水线的详细介绍。[RICH07] 是一个很好的概述。

ARM08a　ARM Limited. *Cortex-A8 Technical Reference Manual.* ARM DDI 0344E, 2008. www.arm.com

FOG08b　Fog, A. *The Microarchitecture of Intel and AMD CPUs.* Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

HINT01　Hinton, G., et al. "The Microarchitecture of the Pentium 4 Processor." *Intel Technology Journal,* Q1 2001. http://developer.intel.com/technology/itj/

INTE01a　Intel Corp. *Intel Pentium 4 Processor Optimization Reference Manual.* Document 248966-04 2001. http://developer.intel.com/design/Pentium4/documentation.htm.

INTE01b　Intel Corp. *Desktop Performance and Optimization for Intel Pentium 4 Processor.* Document 248966-04 2001. http://developer.intel.com/design/Pentium4/documentation.htm.

INTE04a　Intel Corp. *IA-32 Intel Architecture Software Developer's Manual (4 volumes).* Document 253665 through 253668, 2004. http://developer.intel.com/design/Pentium4/documentation.htm.

JOHN08 John, E., and Rubio, J. *Unique Chips and Systems.* Boca Raton, FL: CRC Press, 2008.

JOUP89a Jouppi, N., and Wall, D. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines." *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

KUGA91 Kuga, M.; Murakami, K.; and Tomita, S. "DSNS (Dynamically-hazard resolved, Statically-code-scheduled, Nonuniform Superscalar): Yet Another Superscalar Processor Architecture." *Computer Architecture News,* June 1991.

LEE91 Lee, R.; Kwok, A.; and Briggs, F. "The Floating Point Performance of a Superscalar SPARC Processor." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991.

MOSH01 Mostowos, A., and Sohi, G. "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling." *Proceedings of the IEEE,* November 2001.

OMON99 Omondi, A. *The Microarchitecture of Pipelined and Superscalar Computers.* Boston: Kluwer, 1999.

PATT01 Patt, Y. "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution." *Proceedings of the IEEE,* November 2001.

POPE91 Popescu, V., et al. "The Metaflow Architecture," *IEEE Micro,* June 1991.

RICH07 Riches, S., et al. "A Fully Automated High Performance Implementation of ARM Cortex-A8." *IQ Online,* Vol. 6, No. 3, 2007. www.arm.com/iqonline

SHEN05 Shen, J., and Lipasti, M. *Modern Processor Design: Fundamentals of Superscalar Processors.* New York: McGraw-Hill, 2005.

SIMA97 Sima, D. "Superscalar Instruction Issue." *IEEE Micro,* September/October 1997.

SIMA04 Sima, D. "Decisive Aspects in the Evolution of Microprocessors." *Proceedings of the IEEE,* December 2004.

SMIT95 Smith, J., and Sohi, G. "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE,* December 1995.

WALL91 Wall, D. "Limits of Instruction-Level Parallelism." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991.

14.6 关键词、思考题和习题

关键词

antidependency：反相关生
branch prediction：分支预判
commit：并位（或提交）
flow dependency：流相关性
in-order issue：按序发射
in-order completion：按序完成
instruction issue：指令发射
instruction-level parallelism：指令级并行性
instruction window：指令窗口
machine parallelism：机器并行性
micro-operations：微操作
micro-ops：微操作
out-of-order completion：乱序完成
out-of-order issue：乱序发射
output dependency：输出相关性
procedural dependency：过程相关性
read-write dependency：读写相关性
register renaming：寄存器重命名
resource conflict：资源冲突
retire：回收
superpipelined：超级流水线式
superscalar：超标量
true data dependency：真数据相关性
write-read dependency：写读相关性
write-write dependency：写写相关性

思考题

14.1　处理器超标量设计方法的本质特征是什么？
14.2　超标量与超级流水线的区别是什么？
14.3　什么是指令级并行性？
14.4　简要定义如下术语：
- 真数据相关性
- 过程相关性
- 资源冲突
- 输出相关性
- 反相关性

14.5　指令级并行性与机器并行性有何区别？
14.6　列出并简要定义超标量指令的三种发射策略。
14.7　指令窗口的用途是什么？
14.8　什么是寄存器重命名？它的目的何在？
14.9　超标量机器组织的关键部件是什么？

习题

14.1　当超标量处理器采用乱序完成时，中断处理后的恢复复杂化了。因为检测到异常的条件可能会是一条乱序完成指令的结果，程序不能以异常指令之后的顺序指令来重新启动，因为该后续指令可能已完成过，如果这样做，该指令就执行两次了。请描述一种机制来处理这种情况。

14.2　考虑如下指令序列，它的句法是：操作码之后是一个目标寄存器，再其后是一个或两个源寄存器：

```
0     ADD     R3, R1, R2
1     LOAD    R5, [R3]
2     ADD     R7, R5, 3
3     ADD     R1, R6, R0
4     SUB     R7, R0, R
5     OR      R2, R4, R7
6     SUB     R5, R3, R4
7     ADD     R9, R2, R10
8     LOAD    R6, [R5]
9     SUB     R2, R2, R6
10    AND     R3, R7, 15
```

假定使用 4 段流水线：取指、译码/发射、执行、写回，并假定除执行段以外，所有流水段都花费 1 个时钟周期。对于简单的算术和逻辑指令，执行段花费 1 个时钟周期；但对于由存储器的装载（LOAD），执行段要花费 5 个时钟周期。

若此简单的标量流水线具有乱序执行能力，则对于前 7 条指令我们能构成下表：

指令编号	取指	译码	执行	写回
0	0	1	2	3
1	1	2	4	9
2	2	3	5	6
3	3	4	10	11
4	4	5	6	7
5	5	6	8	10
6	6	7	9	12

表中 4 个流水段下的项，指示每条指令在每个阶段开始的时钟周期。在这个指令序列中，第二个 ADD 指令（指令 3 的一个源操数 6 依赖于 LOAD 指令（指令 1））。因为 LOAD 指令的执行需要 5 个时钟周期，发射逻辑在 2 个时钟周期后遇到这条相关的 ADD 指令时，它必须延迟 3 个时钟周期再发射 ADD 指令去执行。使用它的乱序完成能力，处理器在时钟周期 4 停止指令 3 发射时，转而发射下面 3 条独立的指令并使它们分别以时钟 6、8、9 进入执行段。在时钟 9，LOAD 指令结束后，相关的 ADD 指令就能在时钟 10 被发射去执行了。

(a) 请对上述的 11 条指令完成比表。

(b) 若没有乱序完成能力，请重做此表；乱序完成能力节省了多少时间？

(c) 假定它是一个超标量实现，能每流水段同时处理两条指令，请重做此表。

14.3 考虑如下汇编语言程序：

```
I1: Move R3, R7        /R3 ← (R7)/
I2: Load R8, (R3)      /R8 ← Memory (R3)/
I3: Add R3, R3, 4      /R3 ← (R3) + 4/
I4: Load R9, (R3)      /R9 ← Memory (R3)/
I5: BLR R8, R9, L3     /Branch if (R8) > (R9)/
```

这个程序包括了写后写（WAW）、写后读（RAW）、读后写（WAR）相关性，请指出。

14.4 (a) 在下面的指令序列中找到写后读相关、写后写相关和读后写相关：

```
I1: R1 = 100
I2: R1 = R2 + R4
I3: R2 = R4 - 25
I4: R4 = R1 + R2
I5: R1 = R1 + 20
```

(b) 重命名以上 (a) 中寄存器，消除相关问题。对于原始寄存器值的引用，在寄存器引用中加以下标 "a" 进行标示。

14.5 考虑图 14-13 中所示的 "按序发射/按序完成" 执行序列：

(a) 指出指令 I2 不能在第 4 周期前进入执行阶段的最可能原因。如果采用 "按序发射/乱序完成" 或 "乱序发射/乱序完成" 策略，能解决这个问题么？如果能，是哪种策略解决了这个问题？

(b) 指出指令 I6 在第 9 周期前不能进入写阶段的原因。如果采用 "按序发射/乱序完成" 或 "乱序发射/乱序完成" 策略，能解决这个问题吗？如果能，是哪种策略解决了这个问题？

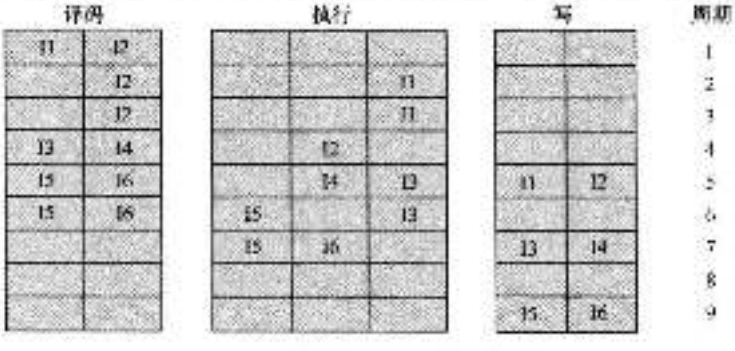


图 14-13 一个按序发射/按序完成的执行序列

14.6 图 14-14 显示了一个超标量处理器组织的例子。如果无资源冲突和数据相关问题，处理器能每周期发射两条指令。这里基本上有两条流水线，每条流水线有 4 段（取指、译码、执行、保存），并有自己的取指、译码和保存单元。4 个功能单元（乘法器、加法器、逻辑单元、装载单元）被两条流水线在执行段动态共享。两个保存单元能被两条流水线动态使用，取决于具体周期时的可用性。这里还有一个先行窗口（lookahead window），它有自己的取指和译码逻辑。这个窗口用于乱序发射指令的先行查找。

考虑在此处理器上执行如下程序：

```
I1: Load R1, A         /R1 ← Memory (A)/
I2: Add R2, R1         /R2 ← (R2) + R(1)/
I3: Add R3, R4         /R3 ← (R3) + R(4)/
I4: Mul R4, R5         /R4 ← (R4) + R(5)/
I5: Comp R6            /R6 ← (R6)/
I6: Mul R6, R7         /R3 ← (R3) + R(4)/
```

(a) 程序中存在什么相关性？

(b) 请给出这个程序在图 14-14 的处理器上运行时的流水线动作，使用类似于图 14-2 的表示法。首先考虑采用按序发射，按序完成策略。

(c) 再考虑采用按序发射，乱序完成策略。

(d) 最后，考虑采用乱序发射，乱序完成策略。

14.7　图 14-15 摘自于一篇超标量设计方面的论文，请解释图中的 3 个子图，并定义 w，x，y 和 z。

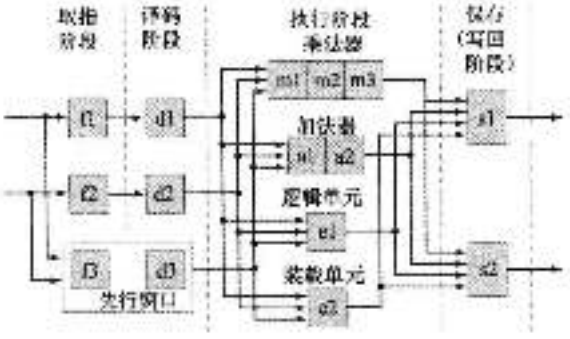


图 14-14　一个双流水线的超标量处理器

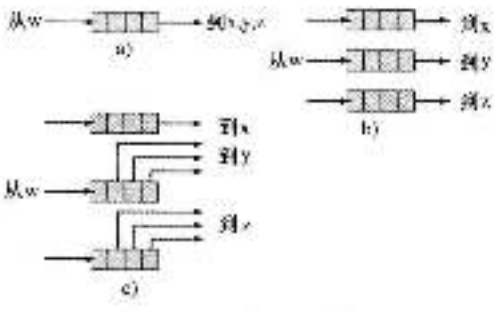


图 14-15　习题 14.7 的图

14.8　用于 Pentium 4 上的 Yeh 动态分支预测算法是一个两级分支预测算法。第一级是最后 n 次转移的历史，第二次是此 n 次转移最后出现 s 次的转移样式的转移行为。可按如下方法实现。程序中的每个条件分支指令在转移历史表（Branch History Table，BHT）都有一对应项，每项由 n 位组成，它相应于该分支指令最后 n 次的执行：若 i 次转移发生，则 i 位置 1；若没发生，则 i 位置 0。每个 BHT 项可索引到一个样式表（Pattern Table，PT），PT 有 2^n 项，每种可能的 n 位样式有一项。每个 PT 项由 s 位组成。这些 s 位在分支预测中的使用如在第 12 章所述的那样（例如，图 12-19）。在指令取指和译码期间遇到一个条件分支指令时，此指令的地址用于取出一个相对应的 BHT 项，它表示该指令的历史信息。然后，该 BHT 项用于取出相对应的 PT 项，来进行转移预测。此分支指令执行后，此 BHT 项更新，然后相对应的 PT 项也更新。

（a）测试这一策略的性能时，Yeh 曾尝试了图 14-16 所示 5 种不同预测方案。请指出其中某一种方案相应于图 12-19 和图 12-28 所示的方案，并请描述其余两种方案。

（b）按这种算法，预测不是只基于某条分支指令的历史，而且还要基于匹配该指令 BHT 表项 n 位样式的所有转移样式的最近历史。请说明这种策略的原理。

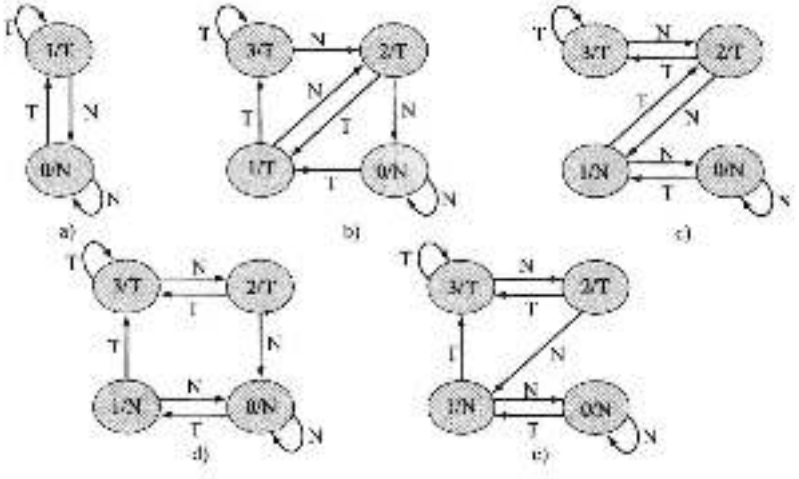


图 14-16　习题 14.8 的图

第四部分 控 制 器

在第三部分，我们集中介绍了机器指令以及每条指令是如何由处理器操作执行的。但没有仔细讨论各个操作是如何发生的，这便是控制单元（Control Unit，也称控制器）的任务。

控制器是处理器中引发上述操作的部分。控制器负责向处理器外部发出控制信号，从而控制与存储器或I/O模块间的数据交换；控制器还需要向处理器内部发送控制信号，以在寄存器间移动数据，并引发ALU完成指定功能以及其他内部调整操作。控制器的输入包括指令寄存器、标志和来自外部的控制信号（例如中断信号等）。

第15章 控制器操作

第15章将讨论处理器功能是如何实现的，更明确的说是处理器的各个部件如何在控制器的控制下提供这些功能的。这一章将说明每个指令周期实际上由一系列微操作（micro-operation）组成，这些微操作产生控制信号。执行就是由控制信号作用产生的效果来完成的。这些控制信号从控制器发出到ALU、寄存器组以及系统互连结构。最后，本章给出了一种控制器实现的方法，称为硬布线（hardwired）实现。

第16章 微程序控制

在第16章中，我们将看到如何实现优雅而功能强大的控制器，这便是人们常说的"微程序设计"（microprogramming）。从本质上讲，微程序语言是底层编程语言。每条处理器的机器指令将会翻译成控制器内底层指令序列。这些底层指令称为"微指令"（microinstruction），而翻译过程称为"微程序设计"。本章讨论了控制存储器的情况，它为每条机器指令保存一个微程序。然后，阐述微程序控制器的结构和功能。

第15章

Computer Organization and Architecture: Designing for Performance, 8E

控制器操作

本章要点

- 一条指令的执行涉及一系列统称为周期的子步骤。例如，一条指令的执行可由取指、间接寻址、执行和中断周期组成。每个周期又是由一系列更基本的操作（称为微操作）组成。一个单一的微操作可以完成寄存器间的一次传送，寄存器与外部总线的一次传送，或一个简单的 ALU 操作。
- 处理器的控制器完成两项任务：(1) 它使得处理器以正在运行的程序所确定的次序来执行微操作；(2) 它产生引起微操作执行的控制信号。
- 控制器产生的控制信号引起逻辑门的打开与关闭，从而导致寄存器数据的传送和 ALU 的操作。
- 一种控制器的实现技术是硬布线技术，采用此技术实现的控制器是一个组合电路。当前机器指令支配的输入逻辑信号被转换为一组输出控制信号。

我们在第 10 章讲过，机器指令集很大程度上决定了 CPU 的功能。若我们知道了机器指令集，包括理解每种操作码的效果，理解寻址方式，以及知道用户可见的寄存器组，也就知道了 CPU 必须完成的功能。但事情还不止于此。我们必须理解外部接口，这通常是总线，还要知道中断是如何处理的。根据以上的推断，可以定义 CPU 所需处理的事项如下：

(1) 操作（操作码）
(2) 寻址方式
(3) 寄存器组
(4) I/O 模块接口
(5) 内存模块接口
(6) 中断

这个列表尽管很普通，但是相当完整。第 (1) ~ (3) 项是由指令集定义的；第 (4) ~ (5) 项一般是由系统总线定义的；第 (6) 项部分由系统总线定义，部分由 CPU 对操作系统的支持类型所定义。

这个 6 项的列表可以看作是对 CPU 的功能要求，它们定义了 CPU 必须做什么。这些也是本书第二部分和第三部分的主要内容。现在，我们转到这些功能是如何完成的问题上来，更具体地，CPU 的各个部件是如何受控来提供这些功能的。下面讨论控制 CPU 操作的控制器。

15.1 微操作

我们已经看到，执行程序时，计算机操作是由一系列指令周期组成，每个周期执行一条机器指令。当然，应记住这个指令周期顺序没必要等同于程序的指令编写顺序，因为存在着分支指令。这里我们所说的顺序指的是指令执行的时间顺序。

进而我们可以看到，每个指令周期又可以看作是由几个更小的单位组成。一种通常的方法是分解为取指、间接、执行和中断，其中取指周期和执行周期总是必有的。

然而，为设计控制器需要将此描述进一步向下分解。在第 12 章讨论流水线时，我们已看到进一步分解是可能的。事实上，我们将会看到，这每个较小周期又由一系列涉及 CPU 寄存器操作的更小步骤组成，人们把这些步骤称为微操作（micro-operation）。“微”是指这些操作很微小，

很简单。图 15-1 描述了前面讨论的各个概念之间的关系。总而言之，一个程序的执行是由指令的顺序执行组成。每条指令的执行是一个指令周期，每个指令周期由更短的子周期（如取指、间接、执行、中断）组成。每个子周期的完成又涉及一个或多个更短的操作，即微操作。

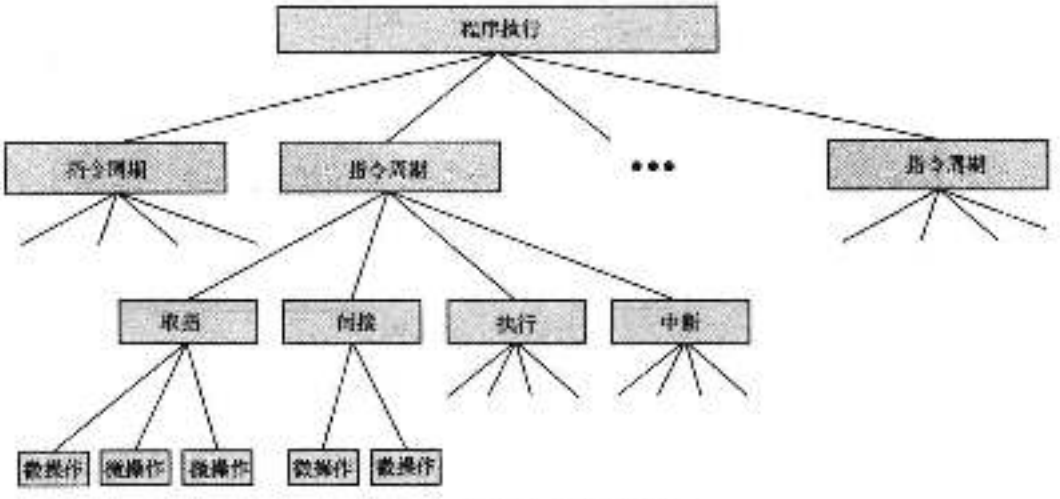


图 15-1　程序执行的组成元素

微操作是 CPU 基本的或者说是原子的操作。本节将考察微操作，以理解任何指令周期的事件是如何被描述成这样的微操作序列。我们将使用一个简单的例子来说明。本章的其余部分用于说明微操作概念如何用于控制器设计。

15.1.1　取指周期

首先查看取指周期，它出现在每个指令周期的开始，并使指令从存储器中取出。为便于讨论，我们假设使用的是图 12-6 所描述的组织。它涉及 4 个寄存器。

- **存储器地址寄存器**（memory address register，MAR）：连接到系统总线的地址线。它指定了读、写操作的内存地址。
- **存储器缓冲寄存器**（memory buffer register，MBR）：连接到系统总线的数据线。它存放将被存入内存的值或最近从内存读出的值。
- **程序计数器**（program counter，PC）：保存待取的下一条指令的地址。
- **指令寄存器**（instruction register，IR）：保存最近取来的指令。

下面从对 CPU 寄存器影响的角度来查看取指周期的事件顺序。图 15-2 给出一个例子。取指周期开始时，下一条将被执行的指令的地址存放在程序计数器 PC 中；此例中的地址是 1100100。第一步是将此地址送到 MAR，因为这是与地址总线相连的唯一寄存器。第二步是装入指令。所要求的地址（在 MAR 中）放到地址总线上，控制器发出一个读（READ）命令到控制总线上，于是结果出现在数据总线上并复制到 MBR 内。我们需对 PC 递增一个指令的长度，以便为取下一条指令做准备。因为这两个动作（由内存读一个字和递增 PC）彼此不相干，故可同时完成以节省时间。第三步是将 MBR 的内容传送到 IR，这也释放了 MBR，使其可用于下面可能有的间接周期。

于是，简单的取指周期实际上由三步和四个微操作组成。每个微操作涉及数据在寄存器之间的传送。只要这些传送彼此互不干扰，那么这几个微操作就可在一步之内同时完成。下面用符号描述此事件顺序。

t_1：MAR←(PC)

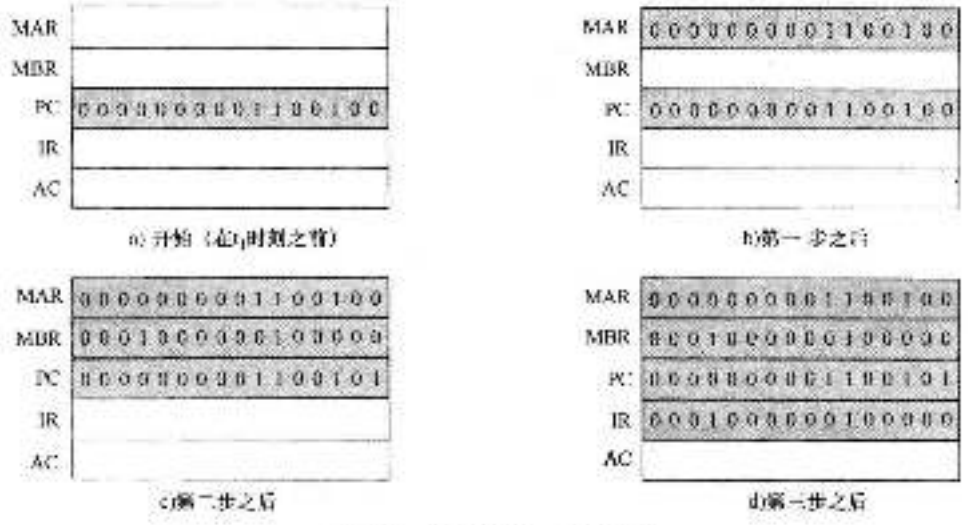


图 15-2 事件顺序，取指周期

t_2：MBR←内存
AC←(AC) + 1
t_3：IR←(MBR)

这里 *I* 表示指令长度。我们需要对这个操作序列做些解释。为了定时，假定有时钟装置可用，它发出等距的时钟脉冲，每个时钟脉冲定义一个时间单位。于是所有的时间单位都是等长的。每个微操作都能在一个时间单位内完成。（t_1，t_2，t_3）代表连续的时间单位。换句话说，我们有：

- **第 1 个时间单位** PC 内容传送到 MAR。
- **第 2 个时间单位** 被 MAR 指定的内存中的内容存放到 MBR 中，PC 递增 *I*。
- **第 3 个时间单位** 传送 MBR 的内容到 IR。

注意，第 2 个和第 3 个微操作都是在第 2 个时间单位同时发生的。第 3 个微操作也能与第 4 个微操作组合在一起，并且不影响取指操作。

t_1：MAR←(PC)
t_2：MBR←内存
t_3：AC←(PC) + *I*
IR←(MBR)

微操作的分组必须遵守下面两个简单的原则：

(1) 事件的先后顺序必须是恰当的。于是，（MAR←(PC)）必须先于（MBR←内存），因为内存读操作要使用 MAR 中的地址。

(2) 必须避免冲突。不要试图在一个时间单位里去读、写同一个寄存器，否则结果是不可预料的。例如，（MBR←内存）和（IR←MBR）这两个微操作不应出现在同一时间单位里。

最后值得注意的是，如果有一个微操作涉及加法运算，为避免电路的重复，这个加法应通过 ALU 完成。根据 ALU 的功能和 CPU 的组织，这个 ALU 的使用可能引起另外的微操作。我们把这个问题放在本章后面讨论。

将这里和下面所讨论的事件与图 3-5 进行对比是有益的。图 3-5 未涉及微操作，这里的讨论表示了完成指令周期的了周期所需的操作。

15.1.2 间接周期

在取到指令后，下一步是取源操作数。继续上述简单例子，假设使用单地址的指令格式并且支持直接与间接寻址方式。若指令指定是间接寻址，则在指令执行前有一个间接周期。数据流与

图 12-7 有所不同，它包括下述微操作：

t_1：MAR←(IR(地址))
t_2：MBR←内存
t_3：IR(地址)←(MBR(地址))

指令的地址字段被传送到 MAR，然后用于读取操作数的地址。最后，MBR 修改 IR 的地址字段，于是它现在容纳的是操作数的直接地址而不是间接地址。

现在，IR 的状态与不使用间接寻址方式时的状态是相同的，并且它已为执行周期准备就绪了。先将执行周期的讨论放在一边，下面考虑中断周期。

15.1.3　中断周期

在执行周期完成时，要进行测试以确定是否有允许的中断产生。若是，则出现一个中断周期。这个周期的特性对于不同的机器差异很大。我们给出一个很简单的事件序列，正如图 12-8 所示，其操作步骤为：

t_1：MBR←(PC)
t_2：MAR←保存地址
　　PC←子程序地址
t_3：内存←(MBR)

在第一步，PC 的内容传送到 MBR，这样它可作为中断返回地址而保存起来。然后，把 MBR 中 PC 内容将要保存到的内存地址装入到 MAR，同时中断处理子程序的起始地址装入 PC。这两个操作可以是单一的微操作。然而，因为大多数 CPU 提供了多种或多级中断，故可能会使用一个或多个另外的微操作来取得该保存地址和子程序起始地址。任何情况下，一旦得到这两个地址，并分别装入到 PC 和 MAR 中，最后一步是将保存有 PC 旧值的 MBR 装入内存。现在，CPU 开始为下一个指令周期做好准备了。

15.1.4　执行周期

取指、间接和中断周期是简单并可预先确定的。每个包括一系列小的、固定序列的微操作，并且每当某周期出现时其相应的一组微操作就被重复一次。

但执行周期不是这样。因为有不同的操作码，所以就可能会出现不同的微操作序列。让我们来思考几个假想的例子。

首先，考虑一条加法指令：

ADD R1，X

它将存储器 X 位置的内容加到寄存器 R1。该加法指令可能产生如下的微操作序列：

t_1：MAR←(IR(地址))
t_2：MBR←内存
t_3：R1←(R1) + (MBR)

开始时，IR 中装有 ADD 指令。第一步是 IR 的地址部分装入到 MAR，然后读出被引用到的存储器位置中的内容。最后，由 ALU 将 R1 和 MBR 的内容相加。同样，这是一个简化的例子。如从 IR 中提取出寄存器的引用，以及用某个中间寄存器对 ALU 的输入和输出进行暂存等，都可能要求另外的微操作。

让我们再来看两个更复杂的例子。一个常用的指令是“递增，若为 0 则跳步”的指令：

ISZ X

X 位置的内容递增 1，若结果是 0，则跳过下一条指令。可能的微操作序列为：

t_1：MAR←(IR(地址))
t_2：MBR←内存
t_3：MBR←(MBR) + 1
t_4：内存←(MBR)
　　If (MBR) = 0 Then (PC←(PC) + 1)

这里引入的新特点是条件操作。若(MBR)=0，则PC递增一个指令长度。这个测试和递增1操作可作为一个微操作来实现。还要注意，这个微操作能与将MBR中的修改值写回内存的微操作同时完成。

最后，考虑了程序调用指令。作为例子，考虑一个"转移并保存地址"指令：

BSA X

此BSA指令之后的指令地址被保存于X位置中，并由X+1位置继续执行。然后这个被保存地址用于返回。这种指令提供对于程序调用点嵌套的支持。对应的微操作是：

t_1：MAR←(IR(地址))
　　MBR←(PC)
t_2：PC←IR(地址)
　　内存←(MBR)
t_3：PC←(PC)+1

指令开始时PC中的地址是下一条指令的地址。它被保存在IR指定的地址位置中。此后，PC中的地址也递增1，以提供下一指令周期的指令地址。

15.1.5 指令周期

我们已看到，指令周期的每个阶段都可分解为一系列的微操作。在本例中，取指、间接、中断周期都各有一个序列，而对于执行周期则是每一操作码有一个序列。

为完善此模型，需要将微操作序列连接在一起，如图15-3所示。我们假设了一个2位的新寄存器，叫做指令周期代码（Instruction Cycle Code，ICC）。此ICC定义了CPU处于周期哪一部分的状态：

00：取指
01：间接
10：执行
11：中断

这4个周期每个结束时会对ICC进行相应的设置。间接周期之后总是执行周期，中断周期之后总是取指周期（参见图12-4）。而执行周期和取指周期之后应是什么周期，这取决于系统的状态。

于是，图15-3的流程图定义了微操作的完整顺序，它仅取决于指令序列和中断模式。当然，这是一个简化的例子，实际的CPU流程会更复杂。但无论如何，我们的讨论已说明：CPU的操作被定义为微操作序列的执行。现在，我们可以来考虑控制器如何发起这个序列的执行。

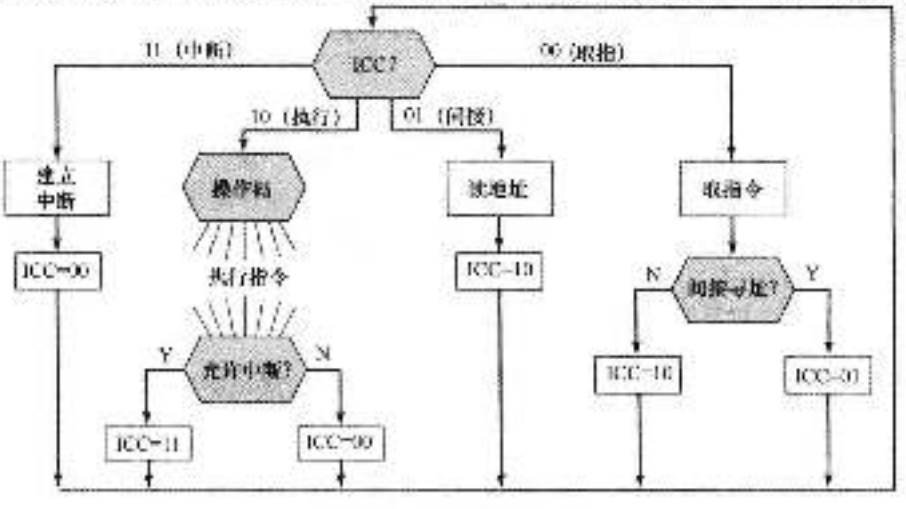


图15-3 指令周期流程图

15.2　处理器控制

15.2.1　功能需求

根据前面的分析结果，可将 CPU 的行为（功能）分解为称作微操作的基本操作。下一个目标是：确定控制器的特性。通过把 CPU 的操作分解到它的最基本级，就能严格定义控制器必须引起什么动作发生。于是，就可以定义控制器的功能需求，即控制器必须完成的功能。这些功能需求的定义是设计和实现控制器的基础。

根据以上信息，如下三步过程能表征控制器：

(1) 定义 CPU 的基本元素。

(2) 描述 CPU 完成的微操作。

(3) 确定为了使微操作完成，控制器必须具备的功能。

至此，已完成第 1 步和第 2 步，让我们总结其结果。首先，CPU 的基本功能元素有：ALU、寄存器组、内部数据通路、外部数据通路、控制器。

下述思考应该可使读者信服，这是一个完整的列表。ALU 是计算机的功能精髓。寄存器组用于 CPU 内的数据存储。某些寄存器包含用于管理指令顺序执行所需的状态信息（如程序状态字）。其他寄存器含有来自或去往 ALU、内存或 I/O 模块的数据。内部数据通路用于寄存器之间或寄存器与 ALU 之间的数据传输。外部数据通路用于将寄存器连接到内存和 I/O 模块，这通常借助于系统总线。控制器引起 CPU 内的操作发生。

程序执行由涉及这些 CPU 元素的操作组成。正如我们已看到的，这些操作由微操作序列组成。回顾 15.1 节可知，所有的微操作可按如下分类：

- 在寄存器之间传送数据。
- 将数据由寄存器传送到外部接口（如系统总线）。
- 将数据由外部接口传送到寄存器。
- 将寄存器作为输入、输出，完成算术或逻辑运算。

完成一个指令周期所需的各种微操作，包括执行指令集中任何指令的微操作，它们都属于上述类型之一。

现在我们能更明确地说明控制器的功能方式，控制器完成两项基本任务：

- 定序（sequencing）：根据正被执行的程序，控制器使 CPU 以恰当的顺序一步步通过一系列微操作。
- 执行（execution）：控制器使每个微操作得以完成。

以上是控制器所完成任务的功能描述，控制器如何实现这些功能的关键是对控制信号的使用。

15.2.2　控制信号

我们已经定义了 CPU 的组成（ALU、寄存器、数据通路等）及其完成的微操作。为使控制器实现其功能，就必须提供允许它确定系统状态的输入和允许它控制系统行为的输出。这些是控制器的外部规范。至于内部规范，控制器必须包含完成它的定序和执行功能的逻辑。我们把控制内部操作的讨论留到 15.3 节和第 16 章。本节的剩余部分用于讨论控制器和 CPU 其他元素的交互作用。

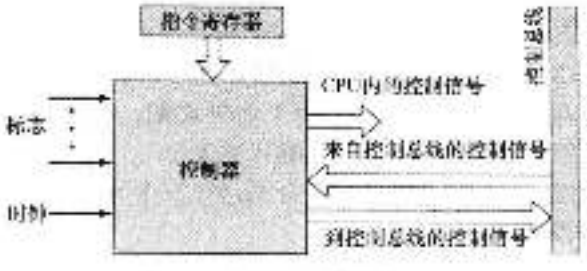


图 15-4　控制器模型结构图

图 15-4 是一个控制器的一般模型，图中显示了控制器的所有输入和输出。输入是：

- **时钟**：这是控制器如何“遵守时间”的关键。控制器要在每个时钟脉冲完成一个（或一组同时的）微操作。这有时称为处理器周期时间（processor cycle time）或时钟周期时间（clock cycle time）。
- **指令寄存器**：当前指令的操作码和寻址方式用于确定在执行周期内完成何种微操作。
- **标志**：控制器需要一些标志来确定 CPU 的状态和前一个 ALU 操作的结果。例如，对那个“递增，若为 0 则跳步”（ISZ）指令来说，控制器将依据零标志位是否置位来决定 PC 是否递增一个指令长度。
- **来自控制总线的控制信号**：系统总线的控制线部分向控制器提供的控制信号。

输出是：

- **CPU 内的控制信号**：这有两类，一类用于寄存器与其他寄存器之间传送数据，另一类用于启动特定的 ALU 功能。
- **到控制总线的控制信号**：这亦有两类，即到存储器的控制信号和到 I/O 模块的控制信号。

控制信号可分为三类，它们分别是：启动 ALU 功能的；控制数据路径的；外部系统总线上的或其他外部接口上的控制信号。所有这些信号最终作为二进制输入量直接输入到各个逻辑门上。

让我们再次考察取指周期，看看控制器如何维护控制。控制器保持着当前处于指令周期何处的记录。在一个给定时间点，控制器知道下面将要完成的是取指周期。第一步是传送 PC 的内容到 MAR。控制器完成这个任务是通过发出控制信号，打开 PC 各位与 MAR 各位之间的门。下一步是由存储器读一个字装入 MBR 并递增 PC。控制器通过发出如下控制信号来完成这个任务：

- 控制信号打开逻辑门，以便允许 MAR 的内容送到地址总线上。
- 存储器读控制信号送到控制总线上。
- 控制信号打开逻辑门，允许数据总线上的内容存入 MBR。
- 控制信号对 PC 内容加 I（指令长度）并把结果存回 PC。

接着，控制器发出打开 MBR 和 IR 之间门的控制信号。

这就完成了取指周期，除了一件事：控制器必须判断下面是要完成一个间接周期还是要完成一个执行周期。为此，它要检查 IR，看看此指令是否要进行间接存储器访问。

间接周期和中断周期的工作类似。对于执行周期，控制器要检查指令的操作码，并由此判断此周期要完成什么微操作。

15.2.3 控制信号举例

为说明控制器的功能，让我们考察图 15-5 所示的例子。这是具有单一累加器（AC）的简单 CPU，图中显示了部件之间的数据通路。控制器发出信号的控制通路未画出，但控制信号的终端用一个小圆圈指示并标记有 C_x。此控制器接受来自时钟、指令寄存器和标志的输入。每个时钟周期，控制器读取所有的这些输入并发出一组控制信号。控制信号分别送往三个目标。

- **数据通路**：控制器控制 CPU 内部的数据流。例如，取指令

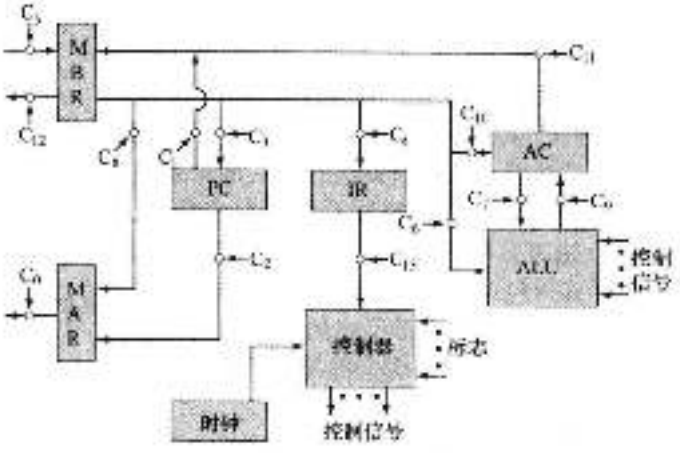


图 15-5 数据通路和控制信号

时，MBR 的内容传送到 IR。为了能控制每条通路，通路上都有逻辑门（图中以圆圈表示）。来自控制器的控制信号可临时打开逻辑门让数据通过。

- ALU：控制器以一组控制信号控制 ALU 的操作。这些信号作用于 ALU 内的各种逻辑电路和门。
- 系统总线：控制器发送控制信号到控制总线上（如存储器读信号）。

控制器必须总是知晓它处于指令周期的什么阶段。利用这一信息，并通过读取所有的输入，控制器发送一系列的控制信号使微操作得以产生。控制器使用时钟脉冲确定事件顺序，允许事件之间有一定的时间间隔以使信号电平得以稳定。表 15-1 表示的控制信号，是前面描述过的某些微操作序列所需的。为了简化，递增 PC 的数据和控制通路，以及固定地址装入 PC 和 MAR 的数据和控制通路没有给出。

表 15-1　微操作和控制信号

	微操作	有效的控制信号
取指	t_1：MAR← (PC)	C_2
	t_2：MBR← Memory PC← (PC) +1	C_5，C_R
	t_3：IR← (MBR)	C_4
间接	t_1：MAR← (IR (地址))	C_8
	t_2：MBR← Memory	C_R，C_E
	t_3：IR (地址) ← (MBR (地址))	C_4
中断	t_1：MBR← (PC)	C_1
	t_2：MAR←保存 地址 PC←子程序 地址	
	t_3：Memory← (MBR)	C_{12}，C_W

注：C_R = 到系统总线的读控制信号
C_W = 到系统总线的写控制信号

考虑控制器的最小特性是有益的。控制器是整个计算机运行的引擎。它只需要知道将被执行的指令和算术、逻辑运算结果的性质（例如，正的、上溢等）。它不需要知道正被处理的数据或得到的实际结果具体是什么。并且，它控制任何事情只是以少量的送到 CPU 内的和送到系统总线上的控制信号来实现。

15.2.4　处理器内部的组织

图 15-5 中使用了不同的数据通路。这类组织的复杂性可以清楚地看到。更典型的是使用某种内部总线的组织结构，如图 12-2 所描存的那样。

使用一个 CPU 的内部总线，可将图 15-5 重安排成如图 15-6 所示。ALU 和所有的 CPU 寄存器都连接到单一的内部总线上。为将数据由各寄存器传送到此总线上，或从此总线上接收，内部总线和寄存器之间提供了门和控制信号。另外的控制线控制着数据和系统（外部）总线的交换以及 ALU 的操作。

图中所示的内部组织已添加了两个新寄存器，分别标记为 Y 和 Z，这是 ALU 的一些相应操作所需要的。当 ALU 的操作涉及两个操

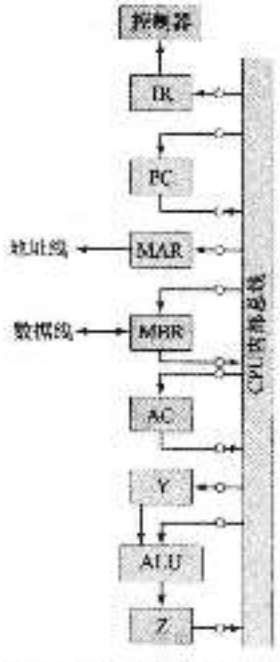


图 15-6　有内部总线的 CPU

作数时，一个可由内部总线得到，但另一个必须要从另外的源得到。累加器 AC 能用于这个目的，但这限制了系统的灵活性。而且，对于有多个通用寄存器的 CPU，使用累加器的方案是不可行的。寄存器 Y 提供了另一个输入的暂时存储。ALU 是一个组合逻辑电路（见第 20 章），其内部无存储电路。这样，当控制信号激活 ALU 的某个功能时，ALU 输入通过 ALU 的组合逻辑电路被转换为 ALU 的输出。因此，ALU 的输出不能直接连到内部总线上，因为这个输出会又反馈为输入。为此，寄存器 Z 提供了这个输出的暂时存储。通过这样的安排，把存储器的值加到 AC 的操作将有如下步骤：

t_1：MAR←(IR(地址))
t_2：MBR←内存
t_3：Y←(MBR)
t_4：Z←(AC) + (Y)
t_5：AC←(Z)

其他的组织形式也是允许的，但通常是要使用某种内部总线或一组内部总线。使用公共数据通路，简化了互连布局和 CPU 的控制。使用内部总线的另一实际理由是节省芯片面积。

15.2.5 Intel 8085

为说明本章至此所介绍的这些概念，让我们考虑 Intel 8085，它的内部结构如图 15-7 所示。几个关键的组件需要稍作解释。

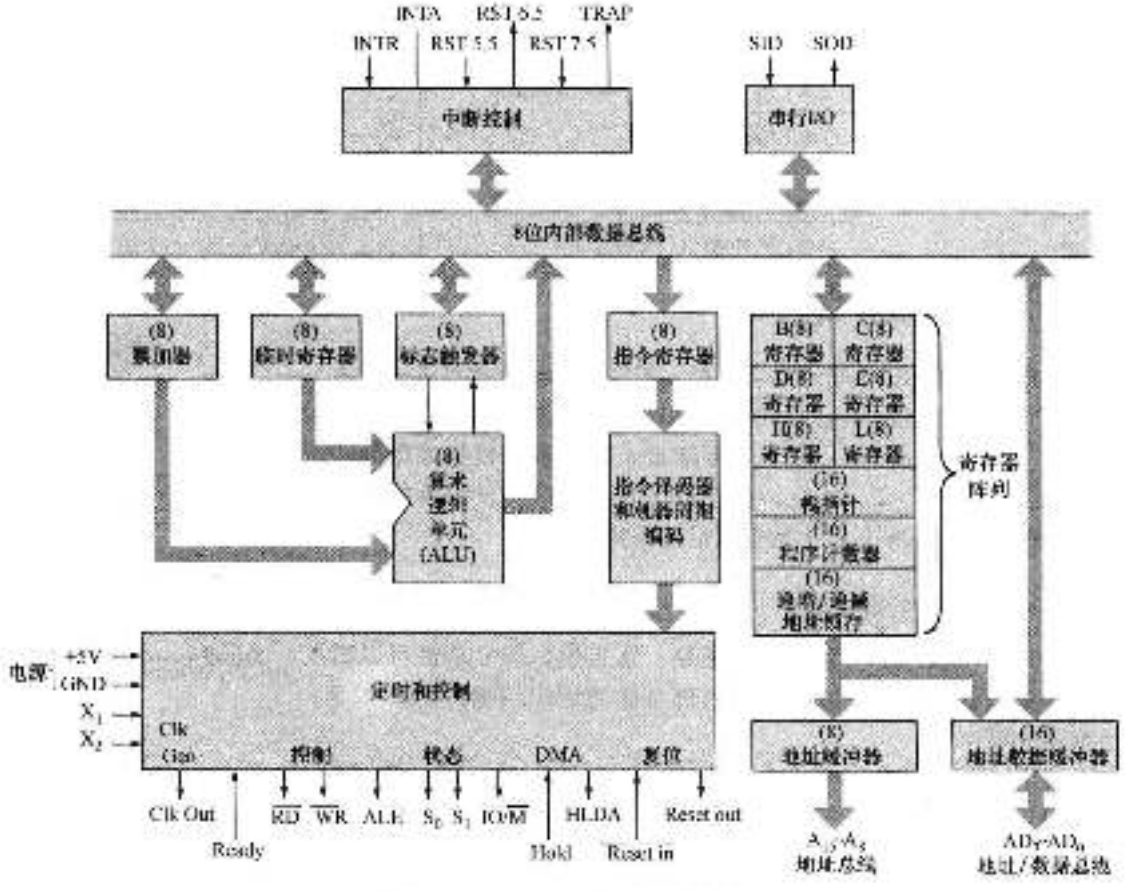


图 15-7 Intel 8085 处理器结构图

- **递增/递减地址锁存**（incrementer/decrementer address latch）：这是能对栈指针或程序计数

器的内容进行加 1 或减 1 的逻辑。这避免了为这些加减法运算使用 ALU，从而节省了时间。

- **中断控制**（interrupt control）：这个模块管理多级中断信号。
- **串行 I/O 控制**（serial I/O control）：这个模块控制与串行设备的接口。串行设备是以每次 1 位的方式进行通信的设备。

表 15-2 说明了 8085 的外部信号，它们连接到外部系统总线上。这些信号是 8085 处理器与系统其余部分的联系（见图 15-8）。

表 15-2　Intel 8085 外部信号

地址和数据信号	高位地址（$A_{15} \sim A_8$）　16 位地址的高 8 位
	地址/数据（$AD_7 \sim AD_0$）　引脚复用，或是 16 位地址的低 8 位，或是 8 位数据
	串行输入数据（SID）　适合串行发送（一次一位）设备的单个位输入
	串行输出数据（SOD）　适合串行接收设备的单个位输出
定时和控制信号	时钟（CLK（Out））　系统时钟，CLK 信号送到外围芯片并同步它们的时序
	X_1，X_2 这些信号来自外部石英晶体或其他设备，用于驱动内部时钟发生器
	地址锁存使能（ALE）　出现于机器周期的第一个时钟周期，并使外围芯片存储地址线上的内容。这就允许地址模块（例如内存，I/O）知道它们被寻址了
	状态（S_0，S_1）　用于指示处理有读或写操作正在发生的控制信号
	IO/$\overline{M}$　用于指示读、写操作的对象是 I/O 模块还是存储器
	读控制（RD）　指示被选中的存储器或 I/O 模块将被读，并且数据总线可用于传输数据
	写控制（WR）　指示数据总线上的数据将被写入到被选中的 I/O 模块或存储器位置中去
存储器和 I/O 发出的信号	保持请求（HOLD）　请求 CPU 放弃对外部系统总线的控制和使用。CPU 将完成当前 IR 中的指令然后进入保持态。在保持态期间，CPU 没有信号去单数据、地址、控制总线，此期间可用于 DMA 操作
	保持确认（HOLDA）　控制器输出此信号表示它已确认了 HOLD 请求信号，并指示总线现在可用
	就绪（Ready）　用于 CPU 与较慢的存储器或 I/O 设备同步。当一个被寻址的设备使 Ready 有效时，CPU 就开始一个输入（DBIN）或输出（WR）操作。否则，CPU 进入等待状态直至此设备就绪
有关中断的信号	自陷（TRAP）　最高优先级可屏蔽中断（RST 7.5、6.5、5.5）
	中断请求（INTR）　为 8 根线可被外部设备使用以中断 CPU。若 CPU 处于保持态或禁止中断，CPU 将不理会中断请求。只在一条指令执行完时中断才被受理。中断以优先权序序排队
	中断确认（INTA）　确认一个中断
CPU 初始化信号	复位输入（Reset in）　使 PC 内容将被置为 0，CPU 将从位置 0 处恢复执行
	复位输出（Reset out）　CPU 已被复位的确认信号，它能用于复位系统的其余部分
电源和地	V_{CC}　+5V 电源
	V_{SS}　电气地

控制器由两个组件组成：指令译码器和机器周期编码（instruction decoder and machine cycle encoding）、定时和控制（timing and control）。第一个组件的讨论推迟到下一节。控制器功能主要在于定时和控制各个模块。这个模块包括一个时钟并接受当前指令和某些外部控制信号。它的控制部分输出信号，这包括送到 CPU 其他部件的控制信号和送到外部系统总线的控制信号。

CPU 操作的时序与时钟同步，并受控于控制器的控制信号。每个指令周期又被分为 1 - 5 个机器周期，每个机器周期又细分为 3 - 5 个状态。每个状态持续一个时钟周期。在一个状态周期内，CPU 完成由控制信号所确定的一个或多个同时的微操作。

机器周期数目，对于给定的一条指令而言是固定的；但从一条指令到另一条指令，它是不同的。机器周期定义为等同于总线访问。于是，一条指令的机器周期数取决于 CPU 必须与外部设备通信的次数。例如，若一条指令由两个 8 位字节组成，则取此指令需要两个机器周期。若此指令涉及 1 字节的存储器或 I/O 操作，则还需要第三个机器周期用于执行。

图 15-9 给出 8085 时序的一个例子，表示了外部控制信号值的变化。当然，与此同时，内部控制信号也正由控制器产生，用于控制内部的数据传输。此图表示的是一条 OUT 指令的指令周期，它需要三个机器周期（M_1，M_2，M_3）。在第一个机器周期，此 OUT 指令的前一半被取出来；第二个机器周期取来指令的后一半，它含有用于输出的 I/O 设备号。在第三个机器周期，AC 的内容经由数据总线写入所选择的设备。

由发自控制器的地址锁存使能 ALE（address latch enabled）脉冲来指示每一机器周期的开始。此 ALE 脉冲使外部电路处于待命状态。在机器周期 M_1 的状态 T_1 期间，控制器设置 $IO/\overline{M}$ 信号指出这是一个存储器操作。控制器亦使 PC 内容放到地址总线（$A_{15} \sim A_8$）和地址/数据总线（$AD_7 \sim AD_0$）上。在 ALE 脉冲的下降沿，总线上的其他模块会保存地址。

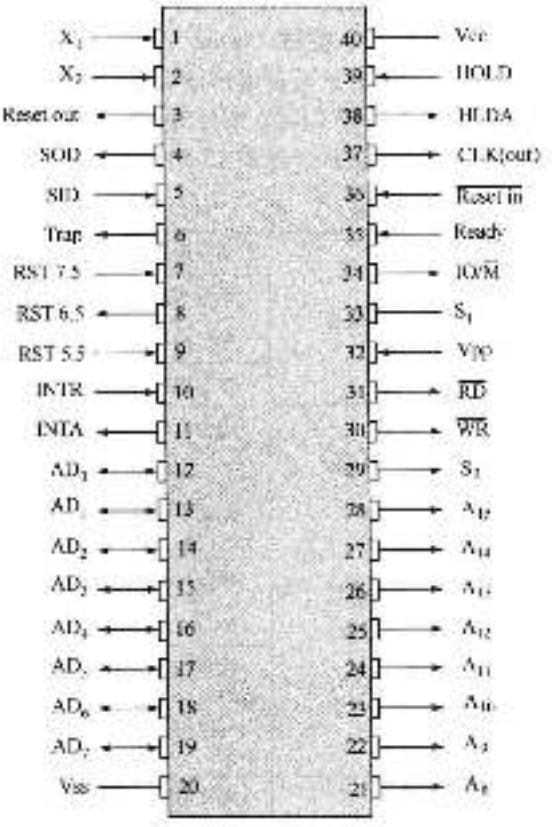


图 15-8 Intel 8085 引脚分布图

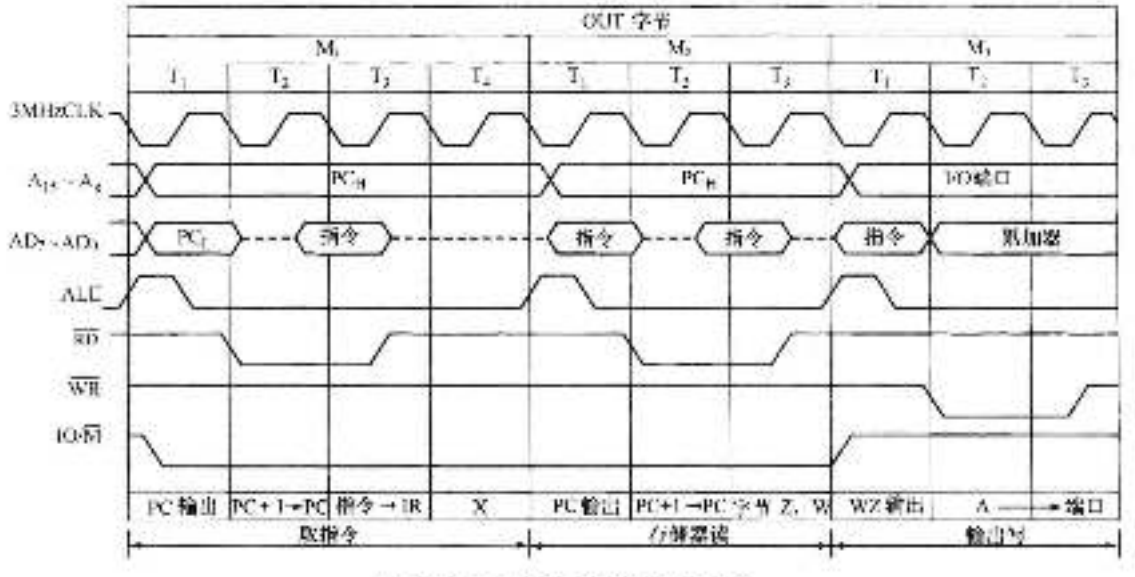


图 15-9 Intel 8085 OUT 指令时序图

在状态 T_2 期间，存储器模块将被寻址的位置中的内容放到地址/数据总线上。控制器设置读控制 RD（read control）信号以指示一个读，但它等待着直到 T_3 才复制总线上的数据。这给存储器模块提供一个时间，使它放数据于总线上并使信号电平得以稳定。最后一个状态 T_4 是一个总线空闲（bus idle）状态，在此期间 CPU 译码此指令。其余的机器周期也以类似方式进行。

15.3　硬布线实现

通过对控制器的输入/输出和功能的介绍，我们详细讨论了控制器。现在是讨论控制器实现问题的时候了。已采用的各种技术可分为两大类：

- 硬布线实现（hardwired implementation）
- 微程序实现（microprogrammed implementation）

通过硬布线方式实现，控制器本质上是一个组合电路。它把输入逻辑信号转换为一组输出逻辑信号，即控制信号。本节考察此方法。微程序实现是第 16 章的主题。

15.3.1　控制器输入

图 15-4 给出的是我们至此已讨论过的控制器，关键输入是指令寄存器、时钟、标志和控制总线信号。对于标志和控制总线信号来说，一般是每个位都有某种意义（例如溢出位）。而另两个输入对于控制器来说其用处不是那么直接明了。

首先考虑指令寄存器。控制器使用指令的操作码，并将为不同的指令完成不同的动作（发出不同的控制信号组合）。为简化控制器逻辑，应使每一操作码有一个唯一的逻辑输入。译码器（decoder）能完成这个功能，它接收一个编码了的输入并产生单一的输出。通常，译码器有 n 个输入和 2^n（2 的 n 次方）个输出。2^n 个不同输入之一将产生唯一的一个输出。表 15-3 是当 n = 4 的一个例子。控制器的译码器更复杂些，它要考虑变长的操作码。第 20 章给出了使用数字逻辑实现译码器的一个例子。

表 15-3　一个 4 输入 16 输出的译码器

I1	I2	I3	I4	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

控制器的时钟部分发出一个重复的脉冲序列。这对于度量微操作的持续时间是有用的。本质上讲，时钟脉冲周期要足够的长，以允许信号能沿着数据通路传播和通过CPU电路。然而，正如我们已看到的，在一个指令周期内，控制器要在不同时间单位发送不同的控制信号。于是，我们希望有一个计数器作为控制器的输入，在不同计时步骤 T_1，T_2 等发出不同的控制信号。在指令周期结束时，控制器必须通知计数器，以使它从 T_1 重新开始。

通过这两点改进，控制器能表示成如图15-10所示的结构。

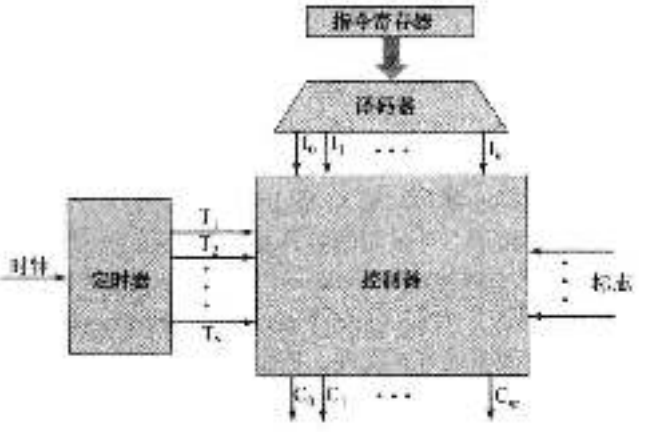


图15-10 带有译码输入的控制器

15.3.2 控制器逻辑

为定义控制器的硬布线实现方案，所剩下的全部事情就是讨论控制器的内部逻辑了，它产生作为输入信号函数的输出控制信号。

基本上，我们必须做的是为每个控制信号得到一个布尔表达式。这最好用例子来说明。让我们再一次考察图15-5说明的简单例子。在表15-1中已看到，微操作序列和控制信号都需要对指令周期4个阶段中的3个进行控制。

让我们考虑一个简单控制信号 C_2，这个信号使外部数据总线上的数据读入MBR。可看出，它在表15-1使用了两次。让我们重新定义两个新的控制信号P和Q，它们具有如下的解释：

PQ = 00	取指周期
PQ = 01	间接周期
PQ = 10	执行周期
PQ = 11	中断周期

则如下的表达式定义了 C_2：

$$C_2 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2$$

即控制信号 C_2 在取指和间接周期的第二个时间单位有效。

这个表达式还不完整，执行周期也需要 C_2。在这个简单例子中假定只有LDA、ADD和AND三条指令还需要在执行时读内存。现可将 C_2 定义为：

$$C_2 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2 + P \cdot \overline{Q} \cdot (\mathrm{LDA} + \mathrm{ADD} + \mathrm{AND}) \cdot T_2$$

对CPU产生的任何控制信号重复这样的过程，结果将是有一组布尔等式，他们定义了控制器的行为，从而定义了CPU的行为。

为将这些组织在一起，控制器必须控制指令周期的状态。正如我们曾提到过的，在每个子周期（取指、间接、执行、中断）结束时，控制器都要发出一个信号，以使时序发生器重新初始并发出 T_1。控制器还必须设置P、Q的相应电平值来定义下面将要完成子周期。

读者应该能想得到，对于当代复杂的CPU而言，其控制器实现所需的布尔表达式数量将非常大。实现这样一个组合电路来满足所有这些布尔表达式的任务将变得异常困难。结果是，普遍使用的是一种更为简洁的方法：微程序设计方式。下一章我们会来讨论这一主题。

15.4 推荐的读物

包括「FARH04」和「MANO04」在内的几本书介绍了控制器功能的基本原理。

FARH04 Farhat, H. *Digital Design and Computer Organization*. Boca Raton, FL: CRC Press, 2004.

MANO04 Mano, M. *Logic and Computer Design Fundamentals*. Upper Saddle River, NJ: Prentice Hall, 2004.

15.5 关键词、思考题和习题

关键词

control bus：控制总线
control path：控制通路
control signal：控制信号
control unit：控制器
hardwired implementation：硬布线实现
microoperation：微操作

思考题

15.1 说明指令的编写顺序（written sequence）与时间顺序（time sequence）的区别。
15.2 指令和微操作的关系是什么？
15.3 CPU 控制器的总体功能是什么？
15.4 概述表征控制器的三步骤。
15.5 控制器要完成的基本任务有哪些？
15.6 给出一个控制器输入输出的典型列表。
15.7 列出三类控制信号。
15.8 简要说明控制器的硬布线实现是什么意思？

习题

15.1 假设有一个 ALU 不能做减法运算，但它能加两个输入寄存器，并能对两个寄存器的各位取逻辑反。数是以 2 的补码形式存储的。请列出实现减法的控制器必须完成的微操作。
15.2 若使图 15-5 中的 CPU 完成如下指令，请以与表 15.1 相同的方式列出其微操作和控制信号：
- 装载一个数到累加器
- 保存累加器内容到存储器
- 加一个数到累加器
- AND 一个数到累加器
- 跳转
- 若 AC = 0 则跳转
- 累加器取反

15.3 假设图 15-6 中的总线传播，以及通过 ALU 的信号其传播延迟分别是 20ns 和 100ns。由总线将数据拷贝到寄存器需要 10ns。那么，必须允许多少时间才能：
（a）从一个寄存器到另一寄存器传送数据。
（b）递增程序计数器。

15.4 以图 15-6 的内部总线结构为例，考虑加一个数到 AC，若该数是：
（a）一个立即数
（b）一个直接寻址的操作数
（c）一个间接寻址的操作数
请写出所需的微操作序列。

15.5 考虑一个按照图 10-14 来实现的栈，请为（a）由栈弹出（pop）和（b）压入（push）栈，给出相应的微操作序列。

第16章

Computer Organization and Architecture: Designing for Performance, 8E

微程序控制

本章要点

- 硬布线控制器的一种替代方案是微程序控制器，它的控制逻辑是微程序指定的。微程序由以微程序设计语言编写的微指令序列组成，这些微指令是用于指定微操作的非常简单的指令。
- 微程序控制器是一个相对简单的逻辑电路，(1) 它能够定序各个微指令，(2) 又能生成执行每条微指令的控制信号。
- 像硬布线控制器那样，微指令产生的控制信号几乎完全等效于 ALU 操作。

20 世纪 50 年代，M. V. Wilkes 最先提出了微程序（microprogram）这个术语［WILK51］。Wilkes 提出了一种控制器的设计方法，它是有组织而又有体系的，避免了硬布线实现的复杂性。这个思想引起了很多研究人员的注意，但由于它要求一个快速且不太昂贵的控制存储器而显得不太实际。

Datamation 在 1964 年 2 月刊上对微程序的技术状况进行了评价。因为那时没有任何微程序式系统在广泛使用，于是一篇论文［HILL64］总结了相当流行的看法，说微程序“是有些前途暗淡，没有任何主要厂商表明对此技术感兴趣，尽管可推测他们都曾考察过此技术。”

但是，这种情况在几个月后就发生了急剧变化。IBM 的 System/360 于 4 月公布，虽然不是全部，但除了它的最大型号之外，其余都是微程序的。尽管 360 系列是在半导体 ROM 可以使用之前，但微程序的优点足够令人信服地使 IBM 这样做。从此，微程序设计成为实现 CISC 处理器的流行技术。近几年来，微程序设计已较少使用，但它仍是计算机设计的有用工具。例如，正如我们看到的，Pentium 4 的机器指令转换为类 RISC 形式，它们中的大多数不使用微程序来执行；然而，仍有某些指令使用微程序来执行。

16.1 基本概念

16.1.1 微指令

正如前面刚描述过的，控制器像是一个相当简单的设备。但是，若以基本逻辑元件来连来实现一个控制器，却不是一个简单的任务。设计必须包括定序微操作、执行微操作、解释操作码以及根据 ALU 的标志来决策等逻辑。设计和测试这样一片硬件是困难的，并且这种设计也不太灵活。例如，改动设计来增加新的机器指令就相当困难。

有一种替代方法，就是微程序控制器设计方法，它是很多 CISC 处理器普遍采用的设计方法。

参见表 16-1，除使用控制信号外，每个微操作都以符号表示来描述。这种表示看起来像是一种编程语言。实际上，它确实是一种编程语言，叫做**微程序设计语言**（microprogramming language）。每行描述一个时间内出现的一组微操作，并称为一条**微指令**（microinstruction）。这种微指令序列被称为**微程序**或**固件**（firmware）。后一术语反映了这样的事实：微程序是介于硬件与软件之间的。以固件进行设计要比硬件容易，但写一个固件程序要比一个软件程序困难得多。

如何使用微程序概念来实现控制器呢？考虑到对于每个微操作，控制器所做的全部事情就是产生一组控制信号。对任一微操作，控制器发出的每根控制线或开或关。自然，对应这种情况，每根控制线可由一个二进制数字表示。这样，我们就构造了一个控制字（control word），其中每位代表一根控制线。从而每个微操作能用控制字中的不同的 0 和 1 的式样来表示。

表 16-1　Wilkes 示例的机器指令集

命　令	命令的效果
An	$C(Acc)+C(n)$存到 Acc_1
Sn	$C(Acc)-C(n)$存到 Acc_1
Hn	$C(n)$移到 Acc_2
Vn	$C(Acc_2)\times C(n)$存到 Acc，其中 $C(n)\geq 0$
Tn	$C(Acc_1)$移到 n，0 移到 Acc
Cn	$C(Acc_1)$移到 n
Rn	$C(Acc)\times 2^{-(n+1)}$移到 Acc
Ln	$C(Acc)\times 2^{n+1}$移到 Acc
En	如果 $C(Acc)<0$，将控制转移到 n；如果 $C(Acc)\geq 0$，忽略（即继续正常前进）
In	从输入读出下一个字符，并存到 n
On	发送 $C(n)$ 到输出设备

注：*Acc* = 累加器
Acc_1 = 累加器高半部分
Acc_2 = 累加器低半部分
n = 存储位置 n
$C(X)$ = X 中的内容（X 可以是寄存器，也可以是存储器位置）

若能将这些控制字串在一起，就能表示控制器完成的微操作序列了。然而，必须认识到微操作序列不是固定的。有时有间接周期，有时又没有。让我们把控制字放入一个存储器单元中，每个字有自己唯一的地址。现在再给每个控制字添加一个地址字段，以指示若某种条件为真时（例如，存储器访问指令中的间接位为 1），将要执行的下一控制字的位置。还有，添加少数几位用于指示条件的真假。

结果是在图 16-1a 中的所谓水平微指令（horizontal microinstruction）。此微指令或控制字的格式说明如下。这里对每一 CPU 内控制线和每一系统控制总线都有相应 1 位。它还有一个指示转移发生条件的条件字段（condition field）和转移目标地址字段（address field），指示将要执行的微指令地址。这样的微指令以如下的方式解释执行：

（1）执行这条微指令的效果是，打开所有位值为 1 的控制线，关闭所有位值为 0 的控制线。生成的控制信号会使得一个或多个微操作被完成。

（2）若条件位指示的条件为假，则顺序执行下一条指令。

（3）若条件位指示的条件为真，则地址字段指向的微指令是将被执行的下一条微指令。

图 16-2 说明这些控制字或微指令在控制存储器（control memory）中是如何安排的。在每个例程中微指令

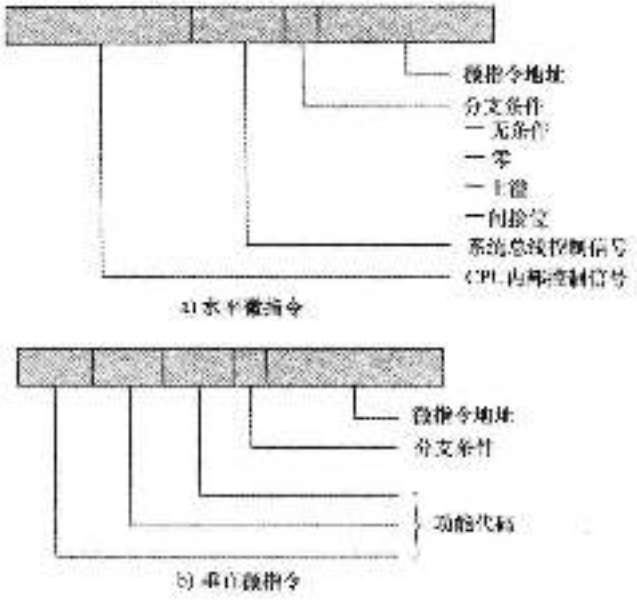


图 16-1　典型的微指令格式

是顺序执行的，在例程的终端有一个分支或转移微指令，指出下面将到何处执行。这里还有一个专门的执行周期例程，它的目的仅在于，根据当前的操作码指明那一个机器指令（AND、ADD等）例程将被执行。

图16-2的控制存储器是控制器整体操作的简明描述，它定义了在每个周期（取指、间接、执行、中断）内将要完成的微操作序列。它也指定了这些周期的顺序。如果没有别的事情，这个表示可作为一个说明具体计算机控制器功能的有用工具。但它还不仅只能作为一个设计说明工具，它亦是实现控制器的一种方式。

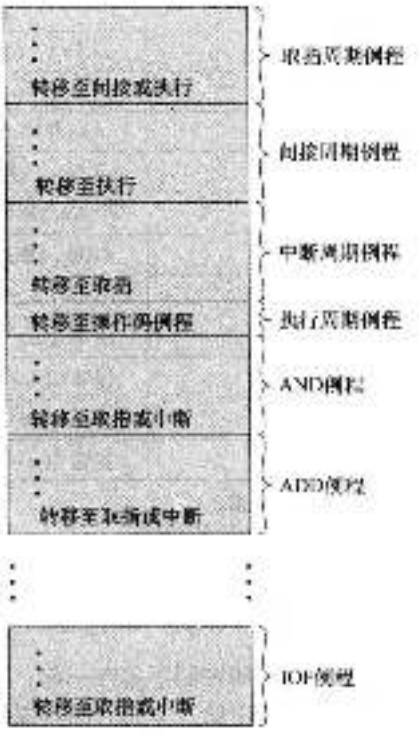


图16-2 控制存储器组织

16.1.2 微程序控制器

图16-2的控制存储器含有描述控制器行为的微程序，下面以执行这个微程序的简单方式来讨论控制器的实现。

图16-3表示了这种控制器实现方式的关键部件：控制存储器（control memory）存有一组微指令；控制地址寄存器（control address register）含有下面即将被读取的微指令地址；当一条微指令由控制存储器读出后，即被传送到控制缓冲寄存器（control buffer register）。此寄存器的左半部分与控制器发出的控制线相连接。于是，由控制存储器读一条微指令等同于执行这条微指令。图中所示的第三个部件是定序器（sequencing unit），它向控制地址寄存器装入地址并发出读命令。

让我们更详细地考察这种结构，如图16-4所示。将此图与图15-4相比，此控制器仍有相同的输入（IR，ALU标志、时钟）和输出（控制信号）。此控制器的功能如下所述。

（1）为执行一条指令，定序逻辑发出一个读命令给控制存储器。

（2）控制地址寄存器指定的一个字读入到控制缓冲寄存器。

（3）控制缓冲寄存器的内容生成控制信号，并为定序逻辑提供下一条地址信息。

（4）定序逻辑根据这个地址信息和ALU标志，将一个新地址装入到控制地址寄存器。所有这些事情都发生在一个时钟周期内。

上述的第4个步骤还需进一步说明。在每条微指令结束时，定序逻辑都要将一新的地址装入到控制地址寄存器。这取决于ALU标志和控制缓冲寄存器的内容，它要进行二选一的决策。

- **取顺序下一条微指令**：加1到控制地址寄存器。
- **基于跳转微指令转移到一个新的例程**：将控制缓冲寄存器的地址字段装入控制地址寄存器。
- **转移到一个机器指令例程**：根据IR中的操作码向控制地址寄存器装入机器指令例程的第一条微指令。

图16-4显示了两个标记有译码器（decoder）的模块。上方的译码器将IR中的地址码翻译为一个控制存储器地址。下方的译码器不用于水平微指令，而是用于垂直微指令（vertical microinstruction）（图16-1b）。正如前面所提到过的，以水平微指令方式，微指令的控制字段中的每一位都接到控制线。以垂直指令方式，一个代码用于表示将被完成的一项动作。例如MAR←(PC)，而由此译码器将这个代码转换为对应的控制信号。垂直微指令的优点在于它比水平微指令更紧凑（位数少），代价是一个小量的附加逻辑和时间延迟。

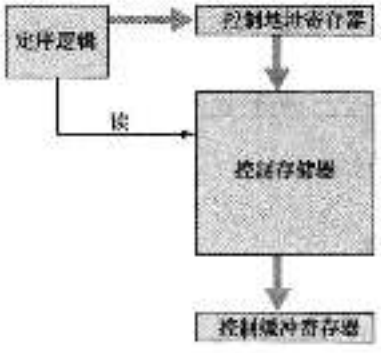


图 16-3　控制器级结构

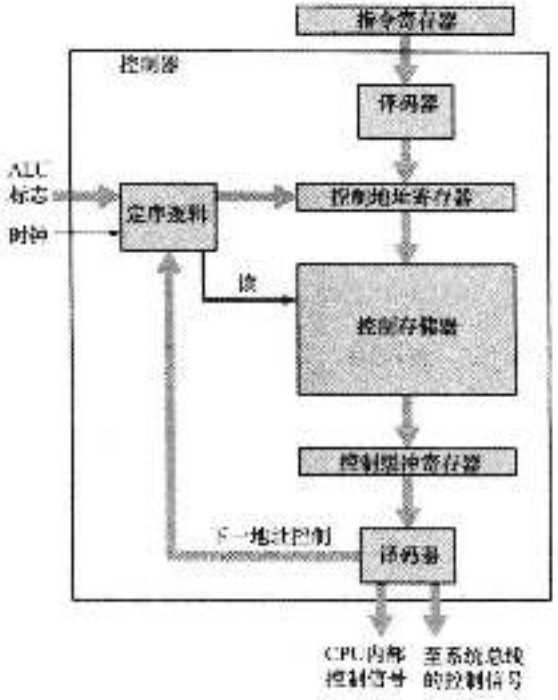


图 16-4　微程序控制器的运作

16.1.3　Wilkes 控制

正如前面所说过的，Wilkes 在 1951 年最早提出使用微程序控制器。这个方案接着被修正并形成更详尽的设计［WILK53］。现在考察这个最初的方案仍是具有指导意义的。

Wilkes 关注的是为设计控制器开发出一种系统化的方法。他提出的配置方案如图 16-5 所示。系统的核心是一个阵列，其中有些部分连接着二极管。在一个机器周期内，阵列的一行被激活。阵列中连接着二极管的地方（图中以圆点所示）产生信号。每行的前一部分产生控制 CPU 操作的控制信号，后一部分产生下一周期将要激活的行地址。于是，阵列的每一行是一条微指令，整个阵列则是控制存储器。

机器周期开始时，将要激活的行地址保存在寄存器 I 中。这个地址输入到译码器，当它被一个时钟脉冲启动时，它激活阵列的某一行。在此周期，或是指令寄存器中的操作码，或是行的后一部分，被传送到寄存器 II。这取决于相应的控制信号。然后由一个时钟脉冲打开寄存器 II 到寄存器 I 的门。交替的时钟脉冲用于启动阵列行和寄存器 II 到寄存器 I 的传送。因为译码器是一个简单的组合电路，故需要这种双寄存器的安排；否则，只使用一个寄存器的话，在一个周期内输出可能会反馈到输入，引起一种不稳定的状况。

这种思想很类似于前面（见图 16-1a）所介绍的水平微程序设计方法。主要的不同在于：在前面的介绍中，控制地址寄存器能递增 1，得到下一顺序地址；而在 Wilkes 方案中，下一地址是在微指令中。为准许转移，一行必须包含两个地址部分，受控于一个条件信号（例如标志），如图 16-5 所示。

为证实所提出的思想，Wilkes 提供了一个以这种方式实现控制器的简单机器。这个例子是已知的第一个微程序 CPU 设计。这里再一次重复它是有益的，因为它说明了许多当代微程序设计的原则。

此原机型 CPU 包括如下寄存器：

A　被乘数

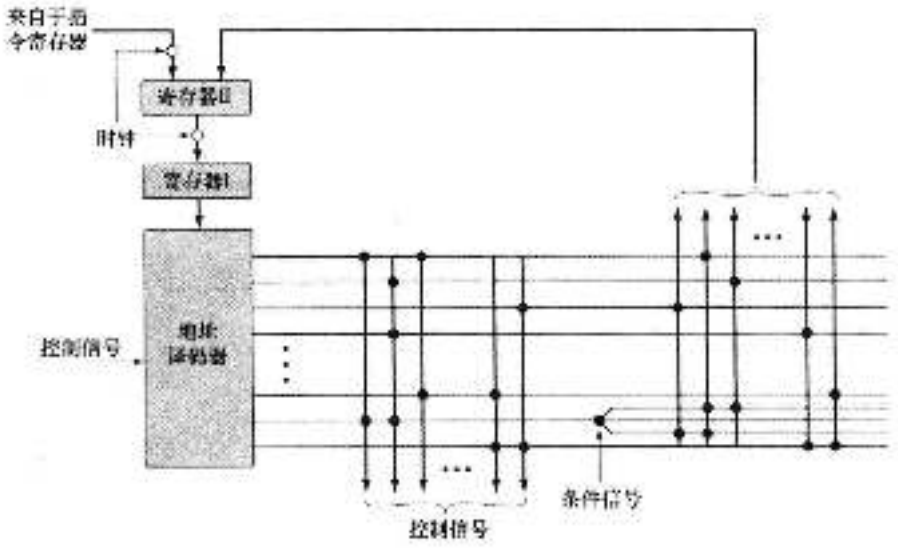


图 16-5 Wilkes 微程序控制器

B 累加器（低半部分）

C 累加器（高半部分）

D 移位寄存器

另外，还有只由控制器可访问的三个寄存器和两个 1 位的标志。三个寄存器是：

E 用作存储器地址寄存器（MAR）和暂时存储的寄存器

F 程序计数器

G 另一个暂时寄存器，用作计数

表 16-1 列出了这个例子的机器指令集。表 16-2 是实现控制器的全部微指令集，以符号形式表示。于是，完整定义此系统需要 38 条微指令。

表中第 1 列给出了每个微指令的地址（行号）。那些对应着操作码的地址被标记出来。例如，当遇到加法指令时，位置 5 处的微指令被执行。第 2 列与第 3 列分别表示 ALU 和控制寄存器单元发生的动作。每个符号表示都必须翻译为一组控制信号（微指令的位）。第 4 列与第 5 列表示两个标志（触发器）的设定和使用。第 4 列指定此标志的信号置位。例如，（1）C_s 意味着由寄存器 C 的符号位来置位标志 1。若第 5 列有一标志识别符，那么第 6 列与第 7 列就会含有两个可选的微指令地址。否则，第 6 列指定将被取的下一条微指令地址。

微指令 0 到 4 构成了取指周期。微指令 4 将操作码提交给译码器，译码器产生对应于待取机器指令的微指令地址。如果细心研究表 16-2，读者不难推导出控制器的全部功能。

表 16-2 Wilkes 例子的微指令

标记法：A, B, C, … 表示算术逻辑单元中不同的寄存器。C 到 D 表示开关电路把寄存器 C 的输出连接到寄存器 D 的输入。$(D+A)$ 到 C 表示寄存器 A 的输出被连接到加法器的一个输入上（寄存器 D 被假定连接到加法器的另一个输入上），而加法器的输出连接到寄存器 C。单引号括起来的数值符号（例如 'n'）表示一个常数，它的输出是以最低有效数字为单位的数值 n。

	算术单元	控制寄存器单元	条件触发器		下一条微指令	
			置位	复用	0	1
0		F 到 G 和 E			1	
1		$(G$ 到 $')'$ 到 F			2	

（续）

		算术单元	控制寄存器单元	条件触发器	下一条微指令	
	2		存储器到 G		3	
	3		G 到 K		4	
	4		K 到译码器		—	
A	5	C 到 D			16	
S	6	C 到 D			17	
H	7	存储器到 B			0	
V	8	存储器到 A			27	
T	9	C 到存储器			25	
U	10	C 到存储器			0	
B	11	B 到 D	E 到 C		19	
L	12	C 到 D	E 到 C		22	
G	13		K 到 G	(1)C_5	18	
I	14	输入到存储器			0	
O	15	存储器到输出			0	
	16	(D + 存储器) 到 C			0	
	17	(D − 存储器) 到 C			0	
	18			1	0	1
	19	D 到 $B(R)$ *	(G − '1') 到 K		20	
	20	C 到 D		(1)E_2	21	
	21	D 到 $C(R)$		1	11	0
	22	D 到 $C(L)$ *	(G − '1') 到 K		33	
	23	B 到 D		(1)E_2	24	
	24	D 到 $B(L)$		1	12	0
	25	'0' 到 B			26	
	26	B 到 C			0	
	27	'0' 到 C	'18' 到 K		28	
	28	B 到 D	K 到 G	(1)B_1	29	
	29	D 到 $B(R)$	(G − '1') 到 K		30	
	30	C 到 $D(R)$		(2)E_3 1	31	32
	31	D 到 C		2	28	33
	32	(D − A) 到 C		2	28	33
	33	B 到 D		(1)B_1	34	
	34	D 到 $B(R)$			35	
	35	C 到 $D(R)$		1	36	37
	36	D 到 C			0	
	37	(D − A) 到 C			0	

注：右移，算术单元中的开关电路的设计为：寄存器 C 的最低位放入到寄存器 B 的最高位中（在右移操作期间），而寄存器 C 的最高位（符号位）被重复（这样，对下一个负数，右移结果也是正确的）。

左移，开关电路板简单地设计为使得，在左移操作期间，寄存器 B 的最高位移到寄存器 C 的最低位。

16.1.4 优缺点

使用微程序实现控制器的优点在于，简化了控制器的设计任务，实现起来既成本较低，也能减少出错机会。硬布线控制器需要一个复杂的逻辑，用来使指令周期的众多微操作按序执行。而微程序控制器的译码器和定序逻辑单元是很简单的逻辑电路。

微程序控制器的主要缺点是：要比采用相同或相近半导体工艺的硬布线控制器慢一些。尽管如此，由于它的易实现性，使微程序设计成为当今CISC控制器的主导技术。而对于RISC处理器，由于它们的简单指令格式，一般使用硬布线控制器。下面更详尽地讨论微程序方法。

16.2 微指令定序

微程序控制器的两个基本任务是：

- **微指令定序**（microinstruction sequencing）：由控制存储器得到下一条微指令。
- **微指令执行**（microinstruction execution）：产生执行微指令的控制信号。

设计控制器时这些任务必须一起考虑，因为二者都影响微指令格式和控制器时序。这一节将重点关注定序问题，而尽可能少提及有关格式与时序的问题，后两个问题留待下一节详细讨论。

16.2.1 设计考虑

设计微指令定序要考虑到两个问题：微指令的大小和地址生成时间。第一个问题是明显的，减小微指令的大小就能节省控制存储器的成本；第二个问题是尽可能快地执行微指令的最简单的要求。

执行微程序时，获得下面将要执行的微指令的地址有如下三种情况：

- 由指令寄存器确定
- 下一顺序地址
- 转移

第一种情况在每指令周期中只出现一次，发生在指令刚刚取来之后。第二种情况在大多数情况下是最普遍的，然而设计不能只为顺序存取来优化。转移，无论是有条件还是无条件的，都是微程序必不可少的部分。而且，微指令序列倾向于短小，每3～4条微指令之后就可能有一个转移发生［SIEW82］。于是，为微指令转移设计一种紧缩的、高时效的技术是重要的。

16.2.2 定序技术

必须根据当前的微指令、条件标志和指令寄存器的内容，产生下一微指令的控制存储器地址。已有很多技术被采用，我们以图16-6到图16-8来说明这些技术的常规分类。以微指令中的地址信息为例，可以大致分为双地址字段、单地址字段、可变格式三类。

最简单的方法是在每条微指令中提供两个地址字段。图16-6展示了如何使用这些地址字段，图中使用了一个多路选择器（或多路器），两个地址字段和指令寄存器的内容连接到这个多路选择器作为输入。多路选择器根据地址选择输入，发送四个地址之中的某一个或操作码到控制地址寄存器（CAR）。CAR接着被译码以产生下一微指令地址。由一转移逻辑（branch logic）模块接收控制器标志和微指令控制部分的输入，向多路选择器提供多路选择信号。

虽然双地址字段方法简单，但它的微指令比其他方法需要更多的位。添加一些逻辑就可以做到节省。一种普遍的方法是只有单地址字段（参见图16-7）。按照这种方法，下一地址的选择项是：

- 地址字段
- 指令寄存器代码
- 下一顺序地址

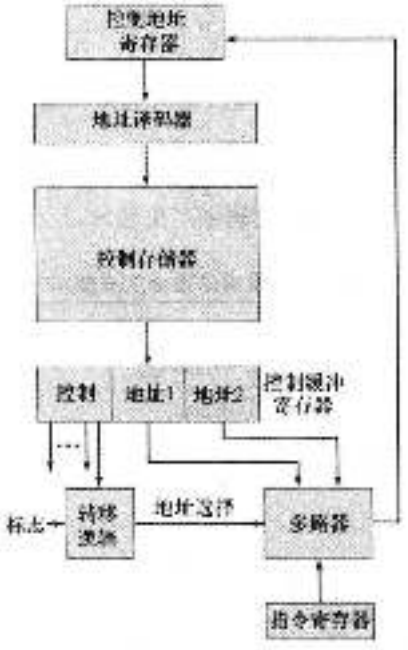


图 16-6　转移控制逻辑：双地址字段

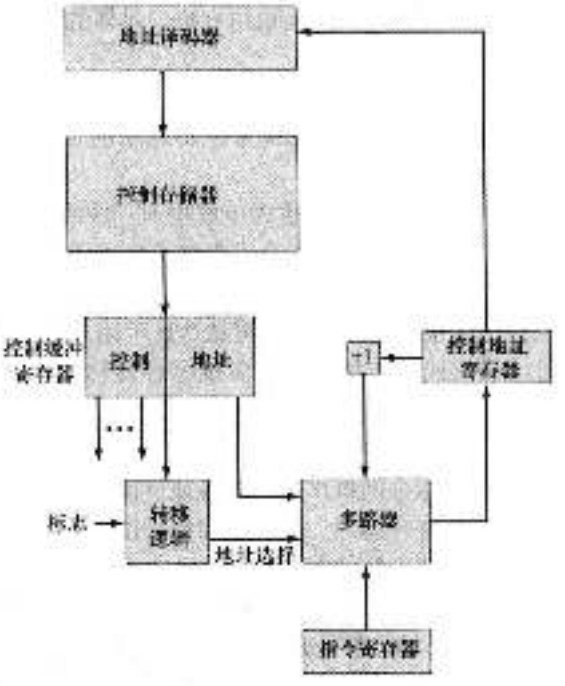


图 16-7　转移控制逻辑：单地址字段

地址选择信号确定哪项被选中，这种方法将地址字段减少到 1 个。然而要注意，一般不会经常用到此地址字段。因此，在微指令编码策略中存在着某种低效因素。

另外的方法是提供两种完全不同的指令格式（见图 16-8）。一位字段用于指定哪种格式正被使用。在一种格式中，其余的位用于产生控制信号。在另一种格式中，某些位用于启动转移逻辑模块，剩余的位用来提供地址。以第一种格式，下一地址或是下一顺序地址，或是由指令寄存器来获取的地址。在第二种格式中，指定了一个有条件或无条件转移。这种方法的缺点之一是每一条转移微指令将耗费一个时钟周期。其他方法的地址产生，与控制信号生成在同一周期，从而减少了控制存储器的存取。

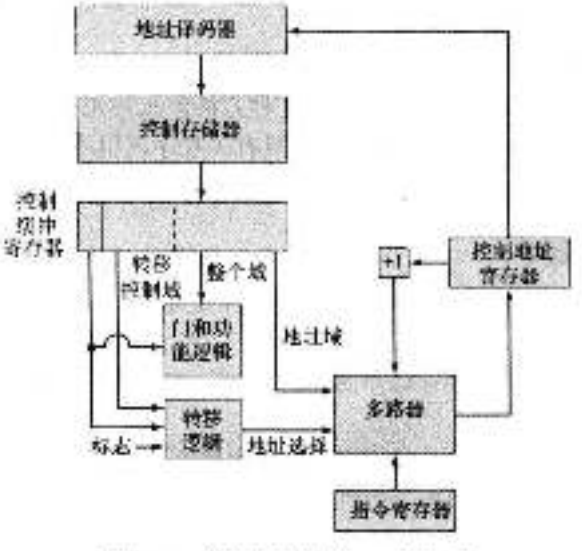


图 16-8　转移控制逻辑：可变格式

刚才介绍的方法只是通用方法。具体的实现常常是这些技术的组合或变异。

16.2.3 地址生成

我们已从格式考虑和通常逻辑需求的观点查看了定序问题，另一观点是考虑取得或计算出下一地址的各种方式。

表 16-3 列出了各种格式生成技术。这些可分成显式（explicit）技术和隐式（implicit）技术两大类。显式技术在微指令中直接给出可用地址，隐式技术要求附加逻辑来产生地址。

我们已知显式技术打过交道了。对于双地址字段法，每条微指令都有两个可选地址可用。使用单地址字段法或可变格式，能实现各种转移微指令。一个条件转移微指令依赖于如下类型信息：

- ALU 标志；
- 机器指令的操作码部分或地址模式字段；
- 选定寄存器的一部分，如符号位；
- 控制器内的状态位。

表 16-3 微指令地址生成技术

显式	隐式
双地址字段	映射（mapping）
无条件转移	加（addition）
条件转移	剩余控制（residual control）

几种隐式技术也得到普遍采用。其中之一，映射，实际上是所有设计都需要的。机器指令的操作码部分必须映射成微指令地址。这在每指令周期中仅出现一次。

普通的隐式技术是将两个地址部分相加或组合来形成一个完整的地址。IBM S/360 系列［TUCK67］以及 S/370 多种型号都采用这种技术。下面以 IBM 3033 作为例子来说明此技术。

IBM 3033 的控制地址寄存器是 13 位长，如图 16-9 所示。它能分成两个地址部分，高 8 位（00－07）一般在一个指令周期到下一个指令周期之间都不会发生改变。微指令执行期间，此 8 位直接由微指令的 8 位字段（BA 字段）复制到控制地址寄存器的高 8 位。这就在控制存储器中选定了一个 32 条微指令的块。控制地址寄存器的剩余 5 位用于指定此块中待取微指令的具体地址。这 5 位的每一位都由当前微指令的一个 4 位字段（有一位是 7 位字段）所确定；这些字段指定了设置相应位的条件。例如，根据最近一次的 ALU 运算是否出现进位，将控制地址寄存器的一位设置为 0 或 1。

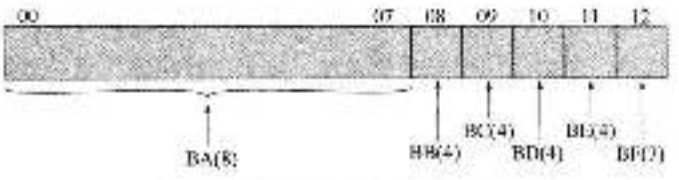


图 16-9 IBM 3033 控制地址寄存器

表 16-3 所列的最后一种方法是剩余控制。这种方法涉及使用先前已保存在控制器内的一个暂存装置中的微指令地址。例如，某些微指令集具有子程序调用功能，一个内部寄存器或寄存器栈用于暂存子程序返回地址。使用这一方法的例子是 LSI-11 机，下面来考察它。

16.2.4 LSI-11 微指令定序

LSI-11 是 PDP-11 的微型机版本，系统的主要部件安装在单板上。LSI-11 的实现使用了微程序控制器［SEBU76］。

LSI-11 的微指令有 22 位，控制存储器的容量是 2K × 22 位字。以如下 5 种方式来确定下一微指令的地址。

- 下一顺序地址：在不出现另外的情况时，控制器的控制地址寄存器的内容增 1。
- 操作码映射：在每个指令周期开始时，由操作码确定下一微指令地址。
- 子程序机制：后文有说明。
- 中断测试：某些微指令有测试中断功能，若中断已出现，它确定下一条微指令的地址。

• 转移：使用了有条件和无条件的转移微指令。

LSI-11 控制器提供了单层（one－level）子程序调用的机制。每个微指令都有一位用于此目的。当此位置位时，控制地址寄存器更新过的内容装入一个 11 位的返回寄存器。子程序调用结束时，指示返回的微指令将使返回寄存器的内容装入控制地址寄存器。

此返回是无条件转移指令的一种形式。无条件转移指令的另一种形式是把微指令中的 11 位地址装入控制地址寄存器中。条件转移指令使用了微指令内的一个 4 位测试码，以对 ALU 各种条件码的测试进行决策。若条件不成立，下一顺序地址将被选择；若成立，微指令中的 8 位被装入控制地址寄存器的低 8 位。这允许在一个 256 字的存储器页内进行跳转。

由上可见，LSI－11 的控制器内包括了一个很有力的地址定序机制。这就允许微程序设计人员有相当的灵活性，并使微程序设计任务能变得容易些。另一方面，这种方法要比功能简单的控制器需要更多的控制逻辑。

16.3　微指令执行

微指令周期是一个微程序 CPU 最基本的事件。每个周期有两部分组成：取指和执行。取微指令这部分前面已介绍过，它由微指令地址的生成时间所确定。本节介绍微指令的执行。

请思考一下，微指令的执行会引起什么发生。基本上，执行的作用在于产生控制信号，一些控制信号发往 CPU 内部，另一些送往外部控制总线或其他外部接口。作为一种附带功能，下一条微指令地址也在微指令执行期间被确定。

前面描述建议的一种控制器组织见图 16-10。它是为强调本节内容对图 16-4 稍加改进的版本。此图的主要模块现在都应是清楚的。定序逻辑模块包含了完成上节所述功能的逻辑。它为生成下一微指令地址，使用了指令寄存器、ALU 标志、控制地址寄存器（为增 1）和控制缓冲寄存器作为输入。最后一项可提供一个实际地址、控制位或者二者皆有。此模块由时钟驱动，时钟确定了微指令周期的定时。

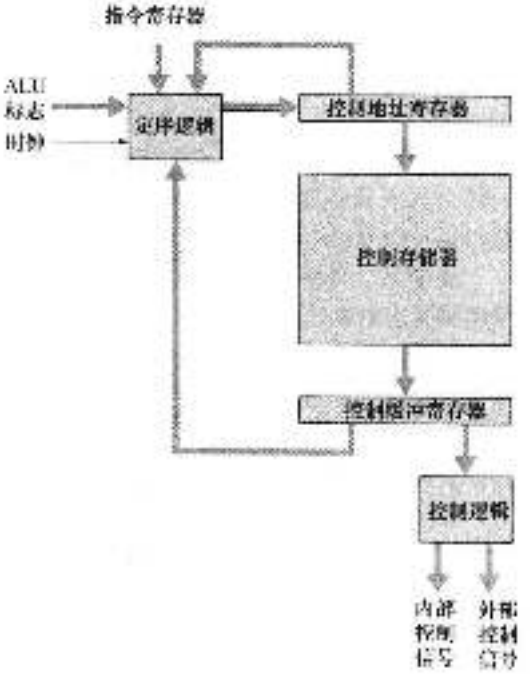


图 16-10　控制器组织

控制逻辑模块生成控制信号，这些控制信号可以看作是微指令某些位的逻辑函数。应当清楚，微指令的格式和内容将决定控制逻辑模块的复杂程度。

16.3.1　微指令的分类法

微指令可用几种方式分类。文献上普遍使用的分类方法包括：

• 垂直/水平（vertical/horizontal）
• 压缩/非压缩（packed/unpacked）
• 硬/软微程序设计（hard/soft microprogramming）
• 直接/间接编码（direct/undirect）

所有的这些都依赖于微指令格式。这些术语每一个都不会在文献上以完全一致的方式使用。然而，考察这些分类的特征能用来说明微指令设计可供选择的方法。下面首先看看支持所有这些分类特征的关键设计出发点，然后查看每一分类特征所表示的概念。

在 Wilkes 策略中［WILK51］，微指令的每一位要么直接产生一个控制信号，要么直接产生下一条微指令地址的一位。我们已经在前一节中看到，越复杂的地址定序方式，就可能使用越少

的微指令位。类似的设计选择也存在于微指令中产生控制信号部分的那些位。通过编码控制信息，而后又解码产生控制信号，可以减少微指令中涉及控制信号产生所需的位数。

这种编码怎样来实现？要回答这个问题，可以假设一个控制器要驱动 K 个不同的内部和外部控制信号。在 Wilkes 设计策略中，微指令中需要分配 K 个位专门用于此目的。这就允许在任一指令周期内产生 2^K 个可能的控制信号组合。如果我们发现不是所有的组合都是要用到的，那么我们可能做得更节省。例如：

- 两个源不能被打开通向到同一目的（例如图 16.5 中的 C_2 和 C_3）。
- 一个寄存器不能同时既作为源又作为目的（例如图 16-5 中的 C_5 和 C_6）。
- 一次只能向 ALU 提交一种控制信号样式。
- 一次只能向外部控制信号提交一种控制信号样式。

对一个给定的 CPU，可列出所有可能的有效控制信号组合，假设有 Q 种，而 $Q<2^K$。这些组合能以 $\log_2 Q$ 来编码，而 $(\log_2 Q)<K$。这将是保留了所有允许的控制信号组合的最紧凑的编码格式。实际上不会使用这种编码格式，理由如下：

- 编程像纯译码式（Wilkes）策略那样困难。关于这点下面还要进一步说明。
- 它要求更复杂的控制逻辑模块，因而速度太慢。

替代的是使用某些折中方案，这有两种：

- 使用比最低要求的位数更多的位来编码。
- 某些物理上允许的组合实际上却是不可能编码的。

后一种折中方式有减少位数的效应，但结果是使用比 $\log_2 Q$ 更多的位。

下一小节将讨论一些具体的编码技术。本小节的其余部分介绍编码效果，以及描述它所用的术语。

据上所述，我们能看出微指令格式中的控制信号能形成一个谱系。一端是每个控制信号有一位，另一端是使用高度编码格式。表 16-4 列出了微程序控制器的其他一些特征，这些特征也是沿谱系分布的。这些特征谱系一般说来是由编码谱系所确定的。

表 16-4 微指令谱系

特 征	
不编码	高度编码
多位	少数位
硬件细节观点	硬件总体观点
编程困难	编程容易
完全利用了并发性	没有完全利用并发性
很少或没有逻辑控制	复杂的逻辑控制
执行快	执行慢
优化性能	优化编程
术 语	
非压缩的（unpacked）	压缩的（packed）
水平的（horizontal）	垂直的（vertical）
硬的（hard）	软的（soft）

表中第二对特征项是相当明白的。纯 Wilkes 策略将要求更多的位。显然，这个极端设计方案反现了硬件最详尽的细节。任何控制信号都分别由微指令各位控制。编码的方案是以一种功能或资源整合的方式来进行的，于是微程序设计人员是在更高、较少细节的级别上看待 CPU 的。而且，编码也是设计用来减轻微程序设计负担的。很显然，理解并指挥好所有控制信号是一件困难的任务。正如前面所提到的，编码的一个通常结果是：禁止使用某些理论上可允许的但实际上不允许的控制信号组合。

前面一段是以微程序设计人员的观点来讨论微指令的。但是，编码程度亦可由它的硬件效应来观察。使用一种纯非编码格式，不需要或较少需要译码逻辑；每一位产生一个具体控制信号。随着使用的编码策略越紧缩、越总体化，所需要的译码逻辑越复杂。这转而又影响性能，因为需要更长的时间，使信号通过更复杂的控制逻辑模块的各个门传播到目的地。于是，执行编码了的微指令要比执行非编码的微指令花费更多的时间。

于是，表 16-4 所列个部特征也归入了设计策略谱系。通常，旧项靠近谱系左端的设计着眼

于优化控制器的性能，归属最近右端的设计则是更关注优化微程序设计的过程。事实的确如此。最近谱系右端的微指令看起来很像机器指令集。这种情况的一个好的例子是本节稍后要介绍的 LSI-11 设计。一般当目标是简化控制器实现时，设计将靠近谱系的左端。本节最后讨论的 IBM 3033 设计就属于这一类。后面将会看到，某些系统准许各类用户使用同样的微指令来构造不同的微程序。这种情况下，设计很可能靠近谱系的右端。

现在能介绍前面引入的术语了。表 16-4 指出这三“对”（pair）术语与微指令集的关系。实际上，所有这些“对”描述的都是同样的事情，只是强调的是不同的设计特征。

微指令压缩的程度关系到给定控制任务和指定微指令位之间的等同程度。随着位变得更紧缩，给定数目的位将提供更多的信息。于是，压缩暗示着编码。水平和垂直这两个术语与微指令相对宽度有关。[SIEW82] 推荐了一个经验规则，垂直微指令的长度在 16～40 位范围内，水平微指令的长度在 40～100 位范围内。硬和软的微程序编程这两个术语，用来暗示与控制信号和硬件布局相关的紧密程度。硬的微程序通常是不变的，并保存在一个只读存储器中。软的微程序是可变的，意味着是可由用户微程序编程所改变的。

另外一对术语是本小节开头提到的直接编码和间接编码，下面就转向这个主题。

16.3.2　微指令编码

实际上，微程序控制器的设计没有使用纯非编码的或水平的微指令格式，但至少要使用某种程度的编码方式，以减小控制存储器的宽度和简化微程序设计任务。

编码的基本技术如图 16-11a 所示。微指令由一组字段组成，每个字段有一代码，经译码产生一个或多个控制信号。

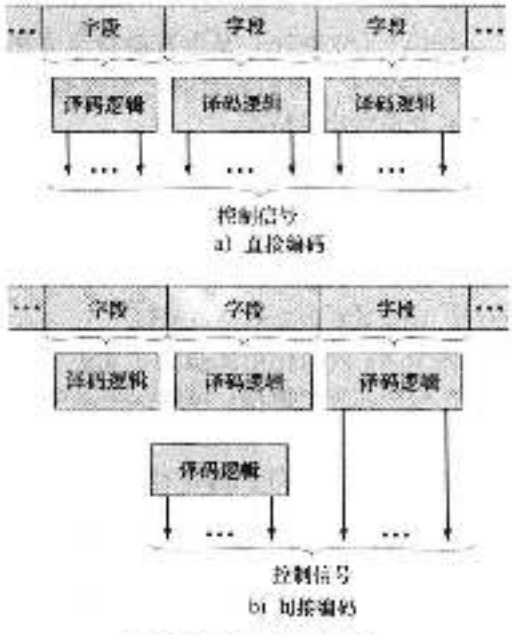


图 16-11　微指令编码

让我们考虑这种情况的含义。当微指令执行时，每个字段被译码并生成控制信号。于是，用 N 个字段，N 个同时发生的动作被指定。每个动作导致一个或多个控制信号被激活。但不总是如此，通常我们希望将格式设计成每个控制信号只被一个字段所激活。然而很清楚，必须允许每个控制信号能被至少一个字段所激活。

再考虑各个字段。一个由 L 位组成的字段可能有 2^L 个不同代码，每个代码对应着不同的控制信号样式。因为每次只能有一个代码出现在字段中，故代码是互斥的，从而它们引起的动作也是互斥的。

设计一个编码的微指令格式，现在可以简单地表示为：

- 格式由独立的字段组成，即每个字段描述这样一组动作（控制信号样式），能使来自不同字段的动作同时实现。
- 这样定义每个字段，即可被此字段指定的各种动作是互斥的。对于一给定字段，每次只能出现一个所指定的动作。

将编码微指令组织成字段格式有两种方法可采用：功能和资源编码方法。功能编码方法识别机器内的功能，并以功能类型来设计字段。例如，若几个数据源都能用来将数据传送到累加器，那么一个字段能设计成用于此目的，用不同代码来指定不同的源。资源编码方法是把机器看成是由一些独立的资源组成，并为每个资源分配一个字段。

编码的另一个方面是，它是直接的还是间接的（见图 16-11b）。如果是间接编码，一个字段会被用来确定另一字段的解释。例如，有一个 ALU，它能完成 8 种不同的算术操作和 8 种不同的移位操作。一个 1 位字段能用来指示将要发生的是算术操作还是移位操作；一个 3 位字段用来指示相应的 8 种操作中的某一个。这种技术通常含有两级译码，因此会增加信号的传播延迟（propagation delay）。

图 16-12 是这些概念的一个简单例子。假定 CPU 有一个单一累加器和几个内部寄存器，如程序计数器、ALU 输入暂存器。图 16-12a 表示了一种高度垂直格式，前 3 位指示操作类型，接着 3 位编码此具体操作，最后 2 位选择一个内部存储器。图 16-12b 是一个水平的格式，虽然仍使用了编码。这种情况下，不同的功能出现在不同的字段中。



图 16-12 简单机器可采用的微指令格式

16.3.3 LSI-11 微指令执行

LSI-11［SEBE76］是垂直微指令方法的一个很好的例子。我们先来看它的控制器组织，然后再来看微指令格式。

1. LSI-11 控制器组织

LSI-11 是 PDP-11 系列以单板处理器面市的第一个成员。此板上有三个 LSI 芯片，一个称为微指令总线（microinstruction bus，MIB）的内部总线，以及某些辅助的接口逻辑。

图 16-13 以简化形式描述了 LSI-11 的 CPU 组织。三个芯片是数据、控制和控制存储芯片。数据芯片含有一个 8 位 ALU、26 个 8 位寄存器和用于保存几个条件码的寄存器，这些寄存器中的 16 个寄存器用来实现 PDP-11 的 8 个 16 位通用寄存器。其余包括有一个程序状态字、存储器地址寄存器（MAR）和存储器缓冲寄存器（MBR）。因为 ALU 每次只能处理 8 位，为实现 PDP-11 的一次 16 位算术运算，数据必须两次通过 ALU，这由微程序控制。

控制存储芯片或芯片组含有一个 22 位宽的控制存储器。控制芯片含有定序和控制微指令的逻辑，控制地址寄存器、控制数据寄存器以及一个机器指令寄存器的副本。

MIB 将所有的这些组件“捆绑”在一起。在微指令取指周期时，控制芯片产生一个 11 位地址放到 MIB 上。于是，控制存储芯片被访问，送出一个 22 位微指令到 MIB 上。此指令的低端 18 位送到控制芯片，同时低端 16 位送到数据芯片。高端 4 位控制专门的 CPU 板功能。

图 16-14 显示了仍是简化但更详细的 LSI-11 控制器组成框图，此图不考虑各个芯片的边界。16.2 节介绍的地址定序逻辑在这里由两个模块实现。微程序定序控制模块提供了全面的定序控制，它能递增微指令地址寄存器或完成无条件转移。另一模块是转换阵列，它能进行其他形式的地址计算。它是一个组合电路，依据微指令、机器指令、微指令程序计数器和中断寄存器来产生地址。

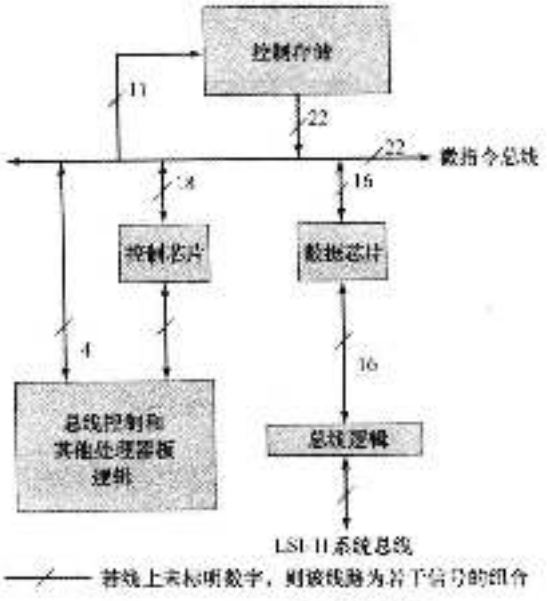


图 16-13　LSI-11 处理器简化图

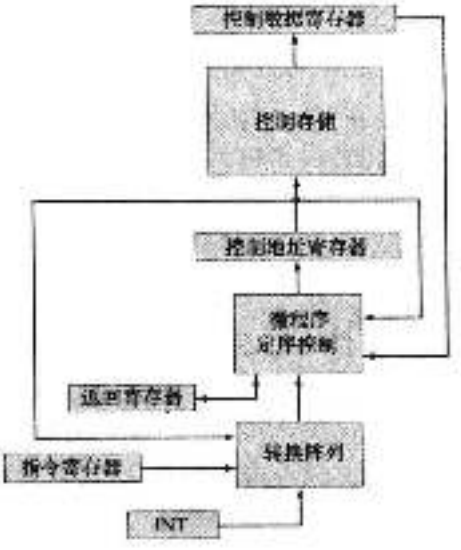


图 16-14　LSI-11 控制器组织

此转换阵列在如下时机起作用：

- 使用操作码确定一个微程序的开始。
- 在适当时机，依指令的地址模式位被测试，以完成相应的寻址。
- 周期性地测试中断条件。
- 检查条件转移指令。

2. LSI-11 微指令格式

LSI-11 使用了一种非常垂直化的微指令格式，它只有 22 位宽。微指令集极其相似于它所实现的 PDP-11 机器指令集。这种设计目的在于，在选用了垂直式、易编程设计的微指令格式前提下，优化控制器的性能。表 16-5 列出了某些 LSI-11 微指令。

表 16-5　某些 LSI-11 微指令

算术操作	逻辑操作	一般指令	输入输出指令
加字(字节,字符) 测试字(字节,字符) 字(字节)增 1 字(字节)增 2 取负字(字节) 条件清零(或减减)字节 条件加字(字节) 带进位的字(字节)加 条件加数字 减字(字节) 比较字(字节,字符) 带进位的字(字节)减 字(字节)减 1	"与"字(字节,字符) 测试字(字节) "或"字(字节) "异或"字(字节) "位清除"字(字节) 带(不带)进位右(左)移字(字节) 字(字节)取反	传送字 转移 返回 条件转移 置位(复位)标志 装入 G 状位 条件传送字(字节)	输入字(字节) 输入状态字(字节) 读 写 读(写)字(字节)并增 1 读(写)字(字节)并增 2 读(写)认可 输出字(字节,状态)

图 16-15 表示了 22 位 LSI-11 的微指令格式。高 4 位控制 CPU 板上的专门功能。转换位允许转换阵列来检查未处理的中断。"装入返回寄存器"位用于在微子程序结束时，由返回寄存器装

入下一微指令地址。其余 16 位用于微操作的高度编码。微指令格式非常像 PDP-11 机器指令，有变长操作码以及一个或多个操作数。

16.3.4 IBM 3033 微指令执行

标准 IBM 3033 控制存储器由 4K 字组成。它的前一半（0000～07FF）含有的是 108 位的微指令；而后一半（0800～0FFF）用于存储 126 位的微指令。图 16-16 描述了这种指令格式。虽然这是一种相当水平的微指令格式，但仍使用了编码技术，格式中的关键字段总结在表 16-6 中。

ALU 对来自 4 个专用的、用户不可见的寄存器 A、B、C、D 的输入进行操作。微指令格式有这样的字段，用来将用户可见的寄存器的内容装入这些用户不可见的寄存器，完成一个 ALU 运算，并指定一个用户可见的寄存器存储此运算结果。微指令中亦有这样的字段，用来在寄存器和存储器之间装载和保存数据。

IBM 3033 的定序机制已在 16.2 节讨论。

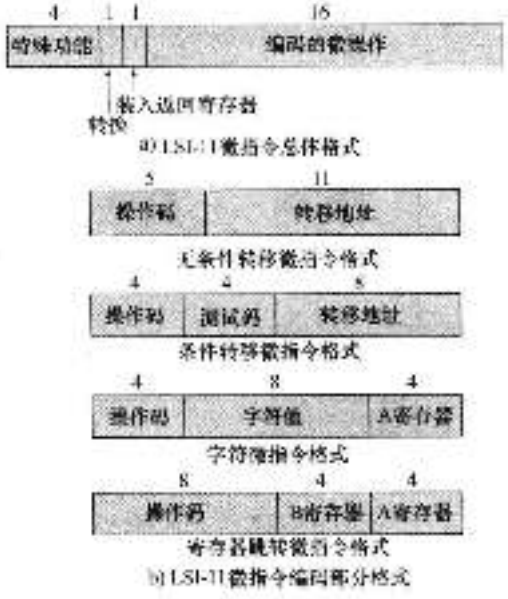


图 16-15 LSI-11 微指令格式

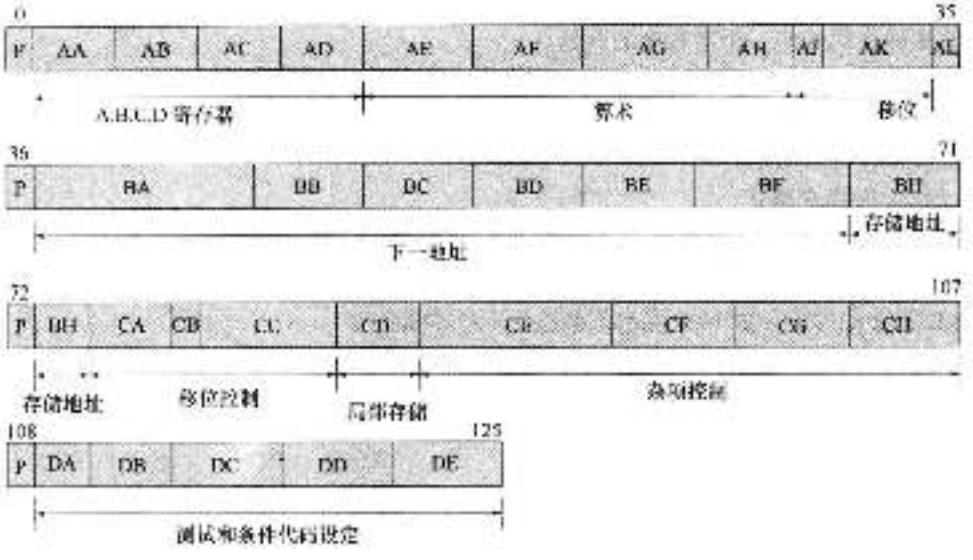


图 16-16 IBM 3033 微指令格式

表 16-6 IBM 3033 微指令控制字段

ALU 控制字段		ALU 控制字段	
AA (3)	由一数据寄存器输入 A 寄存器	AH (4)	指定 ALU 对 B 输入进行算术操作
AB (3)	由一数据寄存器装入 B 寄存器	AJ (1)	指定 0 或 B 输入到 ALU 的 B 侧
AC (3)	由一数据寄存器输入 C 寄存器	AK (4)	传送算术结果输出到移位器
AD (3)	由一数据寄存器装入 D 寄存器	CA (3)	装入 F 寄存器
AE (4)	传送指定的 A 位到 ALU	CB (1)	启动移位器
AF (4)	传送指定的 B 位到 ALU	CC (5)	指定屏蔽和进位功能
AG (5)	指定 ALU 对 A 输入进行算术操作	CE (7)	指定移位总量

（续）

排序和转移字段		排序和转移字段	
AL (1)	结束操作并完成转移 (00 – 07)	BD (4)	指定设置控制地址寄存器位 10 的条件
BA (8)	设置控制地址寄存器的高 8 位	BE (4)	指定设置控制地址寄存器位 11 的条件
BB (4)	指定设置控制地址寄存器位 8 的条件	BF (7)	指定设置控制地址寄存器位 12 的条件
BC (4)	指定设置控制地址寄存器位 9 的条件		

16.4　TI 8800

德州仪器公司的 8800 软件开发板（software development board，SDB）是一个可编程的 32 位计算机卡。该系统有一个以 RAM 而不是 ROM 实现的可写式控制存储器。这样的系统达不到具有 ROM 控制存储器的微程序系统的速度和密度。然而，它对只需要开发原型机和教学应用是很有用的。

TI 8800 SDB 的组件包括：微代码存储器、微定序器、32 位 ALU、浮点和整数处理器、局部数据存储器。

两条总线（DA 和 System Y）连接着系统内部组件。DA 总线将微指令数据字段的数据传到 ALU、浮点处理器或微定序器。此总线也可用于 ALU 或者微定序器向其他组件提供数据。System Y 总线提供了 ALU 和浮点处理器与局部存储器，以及经由 PC 接口与外部模块的连接。

TI 8800 开发板可以嵌入一个 IBM PC 兼容的计算机。主机提供一个可用的平台，来汇编和调试微代码。

16.4.1　微指令格式

8800 的微指令有 128 位，分成 30 个功能字段，其指令格式如表 16-7 所示。这些字段可分成板控制、8847 浮点和整数处理器芯片、8832 寄存器式 ALU、8818 微定序器、WCS 数据字段 5 种类型。

表 16-7　TI 8800 微指令格式

	字 段 号	位　数	说　明
板控制	1	5	选择条件码输入
	2	1	允许/禁止外部 I/O 请求信号
	3	2	允许/禁止局部数据存储器读/写操作
	4	1	装入状态/不装入状态
	5	2	确定驱动 Y 总线的组件
	6	2	确定驱动 DA 总线的组件
8847 浮点和整数处理器芯片	7	1	C 寄存器控制：要时钟或不要时钟
	8	1	为 Y 总线选择最高位或最低位
	9	1	C 寄存器数据源：ALU、多路器
	10	4	为 ALU 和 MUL 选择 IEEE 或 FAST 模式
	11	8	为数据操作数选择源：RA、RB、P、S、C 寄存器
	12	1	RB 寄存器控制：要时钟或不要时钟
	13	1	RA 寄存器控制：要时钟或不要时钟
	14	2	数据源确认
	15	2	允许/禁止流水线寄存器
	16	11	8847 ALU 功能

（续）

	字 段 号	位 数	说 明
8832 寄存器式 ALU	17	2	写允许/禁止数据输出到所选寄存器：高位一半，低位一半
	18	2	选择寄存器集数据源：DA 总线、DB 总线、ALU Y MUX 输出、System Y 总线
	19	3	移位指令修改符
	20	1	进位输入：强制、不强制
	21	2	配置 ALU 模式：32、16 或 8 位
	22	2	选择输入到 S 多路器：寄存器集、DB 总线、MQ 寄存器
	23	1	选择输入到 R 多路器：寄存器集、DA 总线
	24	6	为写选择文件 C 中的寄存器
	25	6	为写选择文件 B 中的寄存器
	26	6	为写选择文件 A 中的寄存器
	27	8	ALU 功能
8818 微定序器	28	12	控制输入信号到 8818
WCS 数据字段	29	16	可写控制存储器字段的高位
	30	16	可写控制存储器字段的低位

如图 16-17 所示，WCS（可写式控制存储器）的 32 位数据字段送入 DA 总线，并将作为数据送给 ALU、浮点处理器或微定序器。微指令的其余 96 位（字段 1～27）是控制信号，直接送至相应的模块。为简化，其他连接未在图 16-17 中给出。

前 6 个字段处理有关板控制的操作，而不是控制个别的组件。控制操作包括：

（1）为定序器选择条件码。字段 1 的第 1 位指示条件标志是将被设定为 1 还是 0，字段 1 的其余 4 位用于指示哪一个条件标志被设定。

（2）发送一个 I/O 请求到 PC/AT。

（3）允许局部数据存储器的读/写操作。

（4）确定驱动 System Y 总线的组件，接到此总线的 4 个组件中的某个被选中（见图 16-17）。

微指令的最后 32 位是数据字段，它含有对应于具体指令的信息。

对于微指令的其他字段，最好在介绍它们控制的设备时，结合上下文一起讨论。下面讨论微定序器和寄存器式 ALU。对于浮

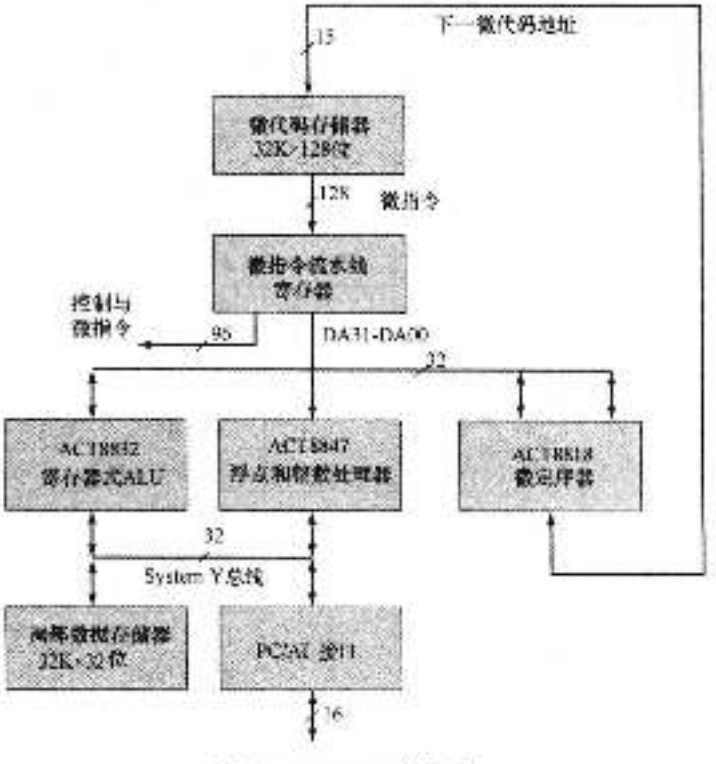


图 16-17 TI 8800 结构图

点处理器，因为它没引入什么新概念，故不予讨论了。

16.4.2　微定序器

8818 微定序器的基本功能是为微程序产生下一个微指令地址。它的 15 位地址提供给微代码存储器（microcode memory），如图 16-17 所示。

微定序器能从如下 5 个来源选取下一地址：

（1）微程序计数器 MPC（microprogram counter），用于重复（重用相同地址）和后继指令（增 1 地址）。

（2）栈，它支持微程序的子程序调用，以及重复循环和中断返回。

（3）DRA 和 DRB 端口，它提供了来自外部硬件的连接，通过这一连接可以生成微程序地址。这两个端口分别连接到 DA 总线的高 16 位和低 16 位。这允许微定序器由当前微指令的 WCS 数据字段或由 ALU 计算结果来获得下一指令地址。

（4）寄存器计数器（register counter）RCA 和 RCB，它们能用于另外的地址存储。

（5）一个对双向 Y 端口的外部输入，以便支持外部中断。

图 16-18 是 8818 的逻辑框图，8818 由如下主要功能组件组成：

- 一个 16 位微程序计数器（MPC），它包括一个计数器和一个递增器。
- 两个寄存器计数器 RCA 和 RCB，用来为循环和重复计数、保存转移地址或驱动外部设备。
- 一个 65 字 × 16 位的栈，用于微程序的子程序调用和中断。
- 一个中断返回寄存器（interrupt return register）和 Y 输出，以允许微指令集的中断处理。
- 一个 Y 输出多路器，通过它能由 MPC、RCA 和 RCB，外部总线 DRA 和 DRB，或栈来选择下一地址。

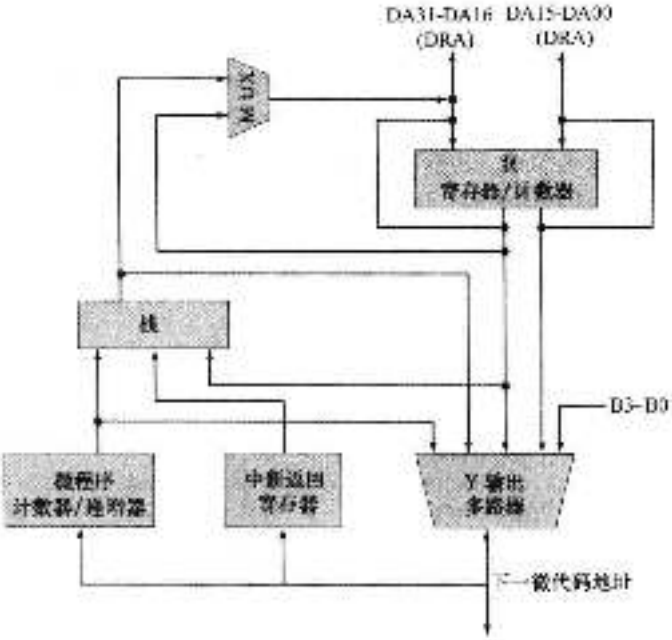


图 16-18　TI 8818 微定序器

1. 寄存器/计数器

寄存器 RCA 和 RCB 可由 DA 总线装入，或装载来自当前微指令或来自 ALU 的输出。其值可

用作计数器来控制执行流，并且当被访问时能自动递减。其值还可用作提供给 Y 输出多路器的微指令地址。除同时递减两个寄存器外，还支持在单一微指令周期内两个寄存器的独立控制。

2. 栈

此栈允许子程序调用或中断的多级嵌套，也能用于支持转移和循环。但应清楚，这些操作是对控制器而言，而不是对整个处理器。这些被涉及的地址是控制存储器中的微指令地址。

6 种栈操作是允许的：

(1) 清除 (clear)，它置栈指针为零，清空栈。

(2) 弹出 (pop)，它递减栈指针。

(3) 压入 (push)，它将 MPC、中断返回寄存器或 DRA 总线的内容放到栈上，并递增栈指针。

(4) 读 (read)，它使被读指针指示的地址在 Y 输出多路器上可用。

(5) 保持 (hold)，它使栈指针的地址保持不变。

(6) 装入栈指针 (load stack pointer)，它使 DRA 的低 7 位装入到栈指针。

3. 微程序控制

从根本上讲，微定序器是被当前微指令字段 28 的 12 位所控制（见表 16-7）。这个字段分成如下子字段。

- OSEL (1 位)：输出选择。它确定哪个值放到连接 DRA 总线的多路器 (MUX) 输出上 (见图 16-18 的左上角)，选择的值来自栈或寄存器 RCA。DRA 则作为 Y 输出多路器或寄存器 RCA 的输入。
- SELDR (1 位)：选择 DR 总线。若此位为 1，则选择外部 DA 总线作为 DRA/DRB 总线的输入。若此位为 0，则选择 DRA 多路器的输出到 DRA 总线（再由 OSEL 控制输出选择）和 RCB 的内容到 DRB 总线。
- ZEROIN (1 位)：用于指示一个条件转移。微定序器的行为，将取决于字段 1 所选的条件码（见表 16-7）。
- RC2 – RC0 (3 位)：寄存器控制。这些位确定寄存器 RCA 和 RCB 中内容的改变。每个寄存器都可以保持不变、递减或由 DRA/DRB 总线装入。
- S2 – S0 (3 位)：栈控制。这些位确定什么样的栈操作将要被完成。
- MUX2 – MUX0：输出控制。这些位与条件码一起控制 Y 输出多路器，从而也控制了下一微指令地址。该多路器能从栈、DRA、DRB 或 MPC 来选择输入作为它的输出。

这些位可被程序员分别设置，但一般不这样做。程序员一般使用等价于所要求位样式的助记符。表 16-8 列出用于字段 28 的 15 个助记符。一个微代码的汇编器将这些助记符转换为相应的位样式。

表 16-8 TI 8818 微定序器的微指令位（字段 28）

助记符	值	说明
RSTS818	000000000110	复位指令
BRA88181	011000111000	转移到 DRA 指令
BRA88180	010000111110	转移到 DRA 指令
INC88181	000000111110	继续执行指令
INC88180	001000001000	继续执行指令
CAL88181	010000110000	转移到 DRA 指定地址处执行子程序
CAL88180	010000101110	转移到 DRA 指定地址处执行子程序
RETS818	000000011010	由子程序返回
PUSH8818	000000110111	将中断返回地址压入栈
POP8818	100000010000	由中断返回
LOADDRA	000010111110	由 DA 总线装入 DRA 计数器
LHADDRB	000110111110	由 DA 总线装入 DRB 计数器

（续）

助记符	值	说明
LDADDRAB	000110111100	装入 DRA/DRB
DECRDRA	010001111100	递减 DRA 计数器并且不为零则转移
DECRDRB	010101111100	递减 DRB 计数器并且不为零则转移

作为一个例子，若当前所选的条件码是 1，助记符 INC 88181 将使顺序的下一微指令被选中。由表 16-8 可知：

INC88181 = 000000111110

直接译码成：

- OSEL = 0　选择 RCA 作为 DRA 的输出送到 MUX；对本示例助记符，该选择是不相干的。
- SELDR = 0　正如上面所定义的；同样，该助记符与此选择不相干。
- ZEROIN = 0　与 MUX 值相结合，指示无转移发生。
- R = 000　保留 RA 和 RC 的当前值。
- S = 111　保留当前栈的状态。
- MUX = 110　当条件码为 1 时选取 MPC，当条件码为 0 时选取 DRA。

16.4.3　寄存器式 ALU

8832 是一个 32 位的 ALU，它带有 64 个寄存器，这些寄存器能配置成 4 个 8 位 ALU、2 个 16 位 ALU 或一个 32 位 ALU 来操作。

微指令的字段 17 – 27 的 39 位控制着 8832（参见表 16-7），这些位作为控制信号提供给 ALU。另外，8832 还有外部连接到 32 位 DA 总线和 32 位系统 Y 总线，见图 16-17。来自 DA 总线的输入能作为输入数据同时提供给 64 字的寄存器组和 ALU 逻辑模块。ALU 和移位操作的结果能输出到 DA 总线或系统 Y 总线。结果亦能反馈给其内部寄存器组。

三个 6 位地址端口允许在寄存器组内部同时完成读取两个操作数和写入一个操作数。MQ 移位器和 MQ 寄存器亦能配置成独立实现双精度的 8 位、16 位和 32 位的移位操作。

每条微指令的字段 17 – 26 控制 8832 内部以及 8832 与外部环境之间的数据流。这些字段是：

- 17——写使能（write enable）。这两位指定写 32 位，写高 16 位，或不写到寄存器组。目标寄存器由字段 24 定义。
- 18　选择寄存器组数据源（select register file data source）。若出现对寄存器组的写操作，这两位指定源为：DA 总线、DB 总线、ALU 输出或系统 Y 总线。
- 19——移位指令修改符（shift instruction modifier）。指定在移位指令期间有关提供终端填入位和读取移出位的选项。
- 20——进位输入（carry in）。这个位指定是否为当前操作将一位进位输入到 ALU。
- 21——ALU 配置模式（ALU configuration mode）。8832 能配置成以单一 32 位 ALU、2 个 16 位 ALU 或 4 个 8 位 ALU 来计算。
- 22——S 输入（S input）。有两个称为 S 和 R 的多路器向 ALU 逻辑模块提供输入。这个字段选择由 S 多路器提供的输入为：寄存器组、DB 总线或 MQ 寄存器。寄存器组中的源由字段 25 定义。
- 23——R 输入（R input）。选择由 R 多路器提供的输入为：寄存器组或 DA 总线。
- 24——目标寄存器（destination register）。指定被目标操作数使用的寄存器组中寄存器的地址。
- 25——源寄存器（source register）。指定被源操作数使用的寄存器组中的寄存器地址，由 S 多路器送出。
- 26——源寄存器。指定被源操作数使用的寄存器组中的寄存器地址，由 R 多路器送出。

最后，字段 27 是一个 8 位操作码用来指定将被 ALU 完成的算术或逻辑运算。表 16-9 列出了它能完成的运算。

表 16-9 TI 8832 寄存器式 ALU 指令字段（字段 27）

组	指 令		功 能	组	指 令		功 能
1	ADD	H#01	R + S + Cn	3	ADDI	H#68	加立即数
	SUBR	H#02	(NOT R) − S + Cn		SUBI	H#78	减立即数
	SUBS	H#03	R = (NOT S) + Cn		BADD	H#88	字节加 R 到 S
	INSC	H#04	S + Cn		BSUBS	H#98	字节减，R 减 S
	INCNS	H#05	(NOT S) + Cn		BSUBR	H#A8	字节减，S 减 R
	INCR	H#06	R + Cn		BINCS	H#B8	字节递增 S
	INCNR	H#07	(NOT R) + Cn		BINCNS	H#C8	字节求增负 S
	XOR	H#09	R XOR S		BXOR	H#D8	字节 R XOR S
	AND	H#0A	R AND S		BAND	H#E8	字节 R AND S
	OR	H#0B	R OR S		BOR	H#F8	字节 R OR S
	NAND	H#0C	R NAND S	4	CRC	H#00	循环冗余字符累加
	NOR	H#0D	R NOR S		SEL	H#10	选择 S 或 R
	ANDNR	H#0F	(NOT R) AND S		SNORM	H#20	单长度规格化
2	SRA	H#00	算术右移单精度		DNORM	H#30	双长度规格化
	SRAD	H#10	算术右移双精度		DIVRF	H#40	除法，余数定位
	SRL	H#20	逻辑右移单精度		SDIVQF	H#50	有符号除法，余数定位
	SRLD	H#30	逻辑右移双精度		SMULI	H#60	有符号乘法重复
	SLA	H#40	算术左移单精度		SMULT	H#70	有符号乘法结束
	SLAD	H#50	算术左移双精度		SDIVIN	H#80	有符号除法初始化
	SLC	H#60	循环左移单精度		SDIVIS	H#90	有符号除法开始
	SLCD	H#70	循环左移双精度		SDIVI	H#A0	有符号除法重复
	SRC	H#80	循环右移单精度		UDIVIS	H#B0	无符号除法开始
	SRCD	H#90	循环右移双精度		UDIVI	H#C0	无符号除法重复
	MQSRA	H#A0	算术右移 MQ 寄存器		UMULI	H#D0	无符号乘法重复
	MQSRL	H#B0	逻辑右移 MQ 寄存器		SDIVIT	H#E0	有符号除法结束
	MQSLL	H#C0	逻辑左移 MQ 寄存器	5	UDIVIT	H#F0	无符号除法结束
	MQSLC	H#D0	循环左移 MQ 寄存器		LOADFP	H#0F	装入除法/BCD 触发器
	LOADMQ	H#E0	装入 MQ 寄存器		CLR	H#1F	清除
	PASS	H#F0	通过 ALU 到 Y(无移位操作)		DUMPFF	H#5F	输出除法/BCD 触发器
3	SETI	H#08	置位位 1		BCDBIN	H#7F	BCD 到二进制
	SET0	H#18	置位位 0		EX3BC	H#8F	超 3 字节修正
	TBI	H#28	测试位 1		EX3C	H#9F	超 3 字修正
	TB0	H#38	测试位 0		SDIV0	H#AF	有符号溢出测试
	ABS	H#48	绝对值		BINEX3	H#DF	二进制转换到超 3
	SMIC	H#58	有符号增除以 2 的补码		NOP32	H#FF	空操作

作为说明字段的 17 - 27 使用的例子，让我们考虑这样一条指令：它将寄存器 1 的内容加到寄存器 2，并将结果放到寄存器 3。此助记符指令是：

CONT11 [17], WELH, SELRYFYMX, [24], R3, R2, R1, PASS + ADD

然后由汇编器转换成相应的位样式。此助记符指令的各部分描述如下：

- CONT11 是一个基本的 NOP 指令。
- 字段 17 被修改为 WELH（写使能，低到高），一个 32 寄存器被写入。
- 字段 18 被修改为 SELRFYMX，选择由 ALU Y 多路器输出的反馈。
- 字段 24 被修改为指定寄存器 R3 用作目标寄存器。
- 字段 25 被修改为指定寄存器 R2 用作源寄存器之一。
- 字段 26 被修改为指定寄存器 R1 用作源寄存器之一。
- 字段 27 被修改为指定一个 ADD 的 ALU 操作。ALU 的移位指令是 PASS，于是，ALU 的输出不被移位器移动。

有关助记符表示法有几点要说明一下。对于连续的字段没必要指出字段号，即

CONT11 [17], WELH, [18], SELRFYMX

可写成：

CONT11 [17], WELH, SELRFYMX

因为字段 17 之后是字段 18。

表 16-9 中组 1 的 ALU 指令必须总是与组 2 相结合使用。组 3 - 5 的 ALU 指令不能与组 2 一起使用。

16.5　推荐的读物

有几本书专门论述了微程序设计，或许最全面的应该是 [LYN93]。[SEGE91] 给出了微代码设计的基础，并以逐步设计一个简单的 16 位微处理器的方式来说明微代码系统的设计。[CART96] 也使用了一个轮到机器来说明基本概念。[PARK89] 和 [TI90] 提供了对 TI 8800 软件开发板的详细描述。

[VASS03] 讨论了计算机设计中的微代码应用的演变和它的当前状况。

CART96　Carter, J. *Microprocesser Architecture and Microprogramming.* Upper Saddle River, NJ: Prentice Hall, 1996.

LYNC93　Lynch, M. *Microprogrammed State Machine Design.* Boca Raton, FL: CRC Press, 1993.

PARK89　Parker, A., and Hamblen, J. *An Introduction to Microprogramming with Exercises Designed for the Texas Instruments SN74ACT8800 Software Development Board.* Dallas, TX: Texas Instruments, 1989.

SEGE91　Segee, B., and Field, J. *Microprogramming and Computer Architecture.* New York: Wiley, 1991.

TI90　Texas Instruments Inc. *SN74ACT880 Family Data Manual.* SCSS006C, 1990.

VASS03　Vassiliadis, S.; Wong, S.; and Cotofana, S. "Microcode Processing: Positioning and Directions." *IEEE Micro*, July-August 2003.

16.6　关键词、思考题和习题

关键词

control memory：控制存储器
control word：控制字
firmware：固件
hard microprogramming：硬微程序设计
horizontal microinstruction：水平微指令
microinstruction encoding：微指令编码
microinstruction execution：微指令执行
microinstruction sequencing：微指令定序
microinstruction：微指令
microprogram：微程序

microprogrammed control unit：微程序控制器
microprogramming language：微程序设计语言
soft microprogramming：软微程序设计
unpacked microinstruction：非压缩微指令
vertical microinstruction：垂直微指令

思考题

16.1 控制器的硬布线实现方式与微程序实现方式有何不同？
16.2 如何解释水平微指令？
16.3 控制存储器有何作用？
16.4 水平微指令执行的典型顺序是什么？
16.5 水平微指令与垂直微指令有何不同？
16.6 微程序控制器完成的基本任务是什么？
16.7 压缩与非压缩的微指令有何不同？
16.8 硬微程序设计和软微程序设计有何不同？
16.9 功能编码和资源编码有何不同？
16.10 列出微程序设计的一些普遍应用。

习题

16.1 描述 Wilkes 原机型中乘法指令的执行过程并画出流程图。
16.2 假设一微指令集有这样一条微指令，以符号形式表示其功能为：

$$\text{IF}(AC_0 = 1)\ \text{THEN CAR} \leftarrow (C_{0-8})\ \text{ELSE CAR} \leftarrow (\text{CAR}) + 1$$

其中，AC_0 是累加器的符号位，C_{1-8} 是微指令的前 7 位。使用这种微指令，写一个实现“寄存器为负转移”（BRM）机器指令的微程序，该微程序检查 AC 寄存器中的内容，若是 AC 为负则转移。假定微指令的 C_1 - C_4 指定一组并行的微操作，用符号形式表示此微程序。

16.3 一个简单 CPU 其指令周期有 4 个主要阶段：取指、间接、执行和中断。硬布线实现方式时，有两个 1 位标志用来指示当前的状态。
(a) 为何需要这些标志？
(b) 为什么微程序控制器不需要这些标志？

16.4 考虑图 16-7 的控制器，假定它的控制存储器是 24 位宽。微指令格式的控制部分分成两个字段。一个 13 位的微操作字段用来指定将要完成的微操作，一个地址选择字段用来指明能引起微指令转移的条件，这些条件是基于 8 个标志来建立的。
(a) 地址选择字段有多少位？
(b) 地址字段有多少位？
(c) 存储控制器的容量有多大？

16.5 在上一问题的环境下，无条件转移指令应如何完成？如何避免转移，即描述一条不指定任何（有条件和无条件的）转移的微指令。

16.6 我们希望为每个机器指令例程能提供 8 个控制字。机器指令的操作码有 8 位，控制存储器有 1024 字。请举出一种由指令寄存器到地址寄存器的映射方式。

16.7 使用一种编码式微指令格式，说明如何划分 9 位的微操作字段来指定 46 种不同动作。

16.8 CPU 有 16 个寄存器，一个 ALU 有 16 种逻辑功能和 16 种算术功能，一个移位器有 8 种操作，所有这些操作都与一个 CPU 内部总线相连。设计一个微指令格式能指定此 CPU 的各种微操作。

第五部分　并行组织

本书的最后部分考察并行组织这个日益重要的领域：在并行组织中，多个处理单元相互协作来执行程序。超标量处理器在指令级发掘并行执行的机会，而并行处理组织寻找更高级别的并行性，使得任务能并行完成，而且是多个处理器协作完成。这种组织方式提出了几个问题。例如，如果多个处理器每个都有自己的高速缓存（cache），它们共享对同一存储器的存取，那么，就必须采用某种硬件或软件机制，以保证这些处理器共享有效的主存映像。这就是所谓的 cache 一致性问题。第五部分将讨论并行组织设计考虑及其他问题。

第 17 章　并行处理

第 17 章先是对并行处理设计进行了综述，然后考察组织多处理器的三种方法：对称多处理器（SMP）、集群系统和非均匀存储器访问（NUMA）机器。SMP 和集群系统是两种最普遍采用的，组织多个处理器来改善性能和可用性的方式。NUMA 系统是近来才提出的概念，还未实现广泛的商业成功，但表现出强劲的发展态势。最后，考察称为向量处理器的专门组织。

第 18 章　多核计算机

多核计算机是包含超过一个以上处理器（核）的计算机芯片。多核芯片使得计算能力相对单个处理器而言将有显著提升，从而使程序能运行得更快。第 18 章介绍了多核计算机设计的一些基本问题，并讨论了来自 Intel x86 和 ARM 体系结构的多核处理器实例。

第17章

Computer Organization and Architecture: Designing for Performance, 8E

并行处理

本章要点

- 提高系统性能的传统方式是使用多个处理器，它们能并行执行，以支持给定的工作负载。现对最普遍的多处理器组织方式是对称多处理器（symmetric multiprocessor，SMP）和集群系统（cluster）。最近，非均匀存储器访问（nonuniform memory access，NUMA）系统已有产品推出。
- SMP由多个相同或相类似的处理器组成，以总线或某种开关阵列互连成一台计算机。SMP所需解决的最重要问题是cache一致性。每个处理器都有自己的cache，于是某一行数据可能出现在不止一个cache中，如果该行在一个cache中被修改，则主存和其他cache保存的将是此行的无效版本。cache一致性协议设计用来解决这个问题。
- 当不止一个处理器实现在单个芯片上时，这种配置就称为**片上多处理**（chip multiprocessing）。一种相关的设计策略是复制单个处理器的某些部件，使处理器能并发地执行多个线程，这称为**多线程处理器**（multithreaded processor）。
- 集群系统由一组“完整的计算机”互连而成，作为统一的计算资源一起工作，并能产生整个集群是作为一台机器在工作的印象。术语“完整的计算机”意味着集中每个计算机可脱离集群系统而独立完成自己的工作。
- NUMA系统是一种共享存储器的多处理器，系统中某个处理器对存储器字的访问时间是随存储器字所在位置的不同而不同的。
- 另一类并行组织是向量机数，专用于处理向量和数据阵列。

传统上，计算机被看成是一个顺序机器。大多数计算机程序设计语言要求程序员指定作为指令顺序的算法。处理器执行程序是顺序地并且每次一条地执行机器指令。每条指令又是顺序地执行一系列操作（取指令、取操作数、完成运算、保存结果）。

这个计算机的顺序观点已不完全是真的了。在微操作级，多个控制信号是同时产生的。指令流水线中至少也要将取指和执行操作相重叠，这已不是新鲜事了。这两个例子都说明了功能的并行执行。这种方法进一步扩展到超标量组织，它发掘了指令级并行性，单个处理器内有多个执行单元，可并行执行同一程序的多条指令。

随着计算机技术的发展和硬件成本的下降，计算机设计人员开始寻求越来越多的实现并行的机会，通常用于提高性能，某些情况下用于改善可用性。在进行概述之后，本章将考察并行组织的几个最著名的方法。首先考察对称多处理器（SMP），这是最早采用、并且至今仍广泛使用的并行组织方法。在SMP组织中，多个处理器共享一个公共的存储器，由此带来了cache一致性问题，我们将用一节的篇幅专门讨论此问题。然后介绍集群系统，它是多个独立的计算机以一种协作风格组织而成。接着，本章考察多线程处理器和片上多处理器。

对于工作负载超出单个SMP能力的情况，采用集群系统方法变得日益普遍。使用多个处理器的另一种方法是非均匀存储器访问（NUMA）式机器，这是一种比较新的方法，有待市场的进一步验证，但它经常被看作是SMP或集群系统的替代方式。本章最后考察向量计算的硬件组织方法。这些方法为处理向量或浮点数矩阵而优化ALU，它们常用在称为超级计算机（supercomputer）的系统实现中。

17.1 多处理器组织

17.1.1 并行处理器系统的类型

最初由 Flynn 提出的一种分类法［FLYN72］至今仍是最普遍使用的，是对具有并行处理能力的系统进行分类的方法。Flynn 提出下列计算机系统类型：

- **单指令单数据（SISD）流**：单一处理器执行单一指令流来操作保存于单一存储器上的数据。单处理器系统属于这一类。
- **单指令多数据（SIMD）流**：一条机器指令控制几个处理部件基于锁步方式同时执行，每个处理部件有一个相关的数据存储器，故每条指令在不同的数据组上执行。17.7 节讨论的向量和阵列处理器属于这一类。
- **多指令单数据（MISD）流**：一系列数据被发送到一组处理器，每个处理器执行不同的指令序列，这种结构从来没有在商业上被实现过。
- **多指令多数据（MIMD）流**：一组处理器同时执行不同的指令序列，对不同的数据集进行操作。SMP、集群系统和 NUMA 系统都属于这一类。

对于 MIMD 组织，处理器是通用的。每个处理器都能处理所有完成相应数据变换的必要指令。MIMD 能以处理器的通信方式来进一步划分（见图 17-1）。如果处理器共享一个公共的存储器，则每个处理器存取共享存储器中的程序和数据，并经由此存储器相互通信。这种系统的最普通形式是对称多处理器（symmetric multiprocessor，SMP），我们将在 17.2 节考察它。在 SMP 中，通过共享总线或其他互连结构，多个处理器共享单一存储器或存储器池；一个明显特征是，存储器任何区域的存取时间，对各个处理器大致是相同的。当前的一种新的实现方法是非均匀存储器访问（non-uniform memory access，NUMA）组织，我们将在 17.5 节介绍它。顾名思义，存储器中不同区域的存取时间，对一个 NUMA 的处理器是不同的。

一组独立的单处理器或 SMP，可互连成集群系统（cluster），计算机之间的通信或是经由固定的路径，或是经由某些网络设施。

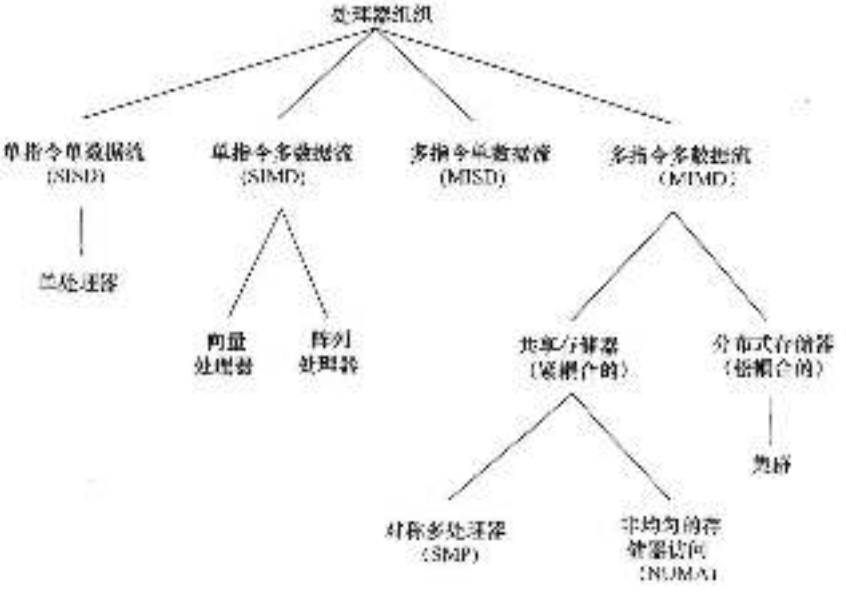


图 17-1 并行处理器体系结构分类

17.1.2 并行组织

图 17-2 说明了图 17-1 分类的通常组织。图 17-2a 表示一个 SISD 的结构。这里有某种控制单元（Control Unit，CU）向处理单元（Processing Unit，PU）提供一个指令流（Instruction Stream，IS），该处理单元对来自一存储单元（Memory Unit，MU）的单一数据流（Data Stream，DS）进行操作。对 SIMD 而言，还是一个单一控制单元，不过现在是向多个处理部件提供单一指令流，每个处理部件可有自己的专用存储器（见图 17-2b）或者有一个共享的存储器。最后是 MIMD，它有多个控制单元，每个向自己的处理部件提供一个独立的指令流。MIMD 可以是共享存储器的多处理器（见图 17-2c），或是一个分布式存储器的多计算机（multicomputer）（见图 17-2d）。

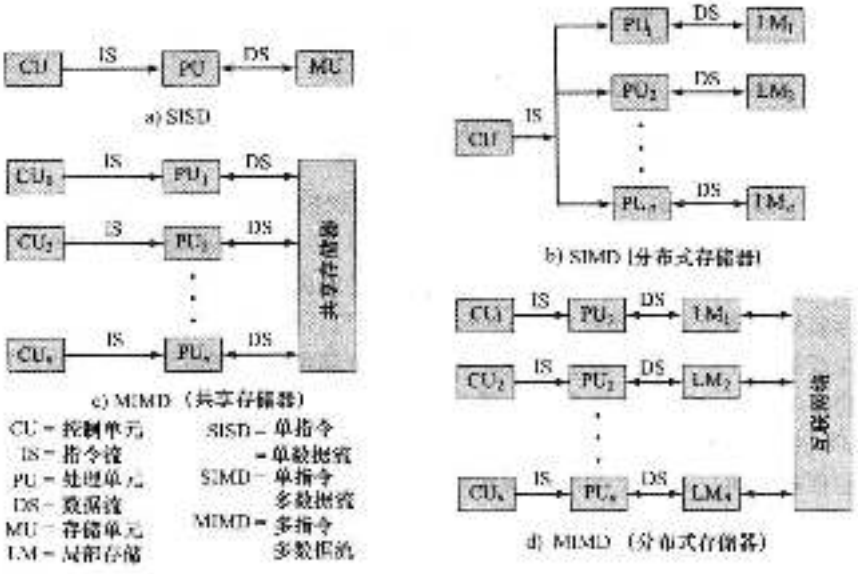


图 17-2 可选的计算机组织

SMP、集群系统和 NUMA 机器的设计是一个涉及物理组织、互连结构、处理器间通信、操作系统设计和应用软件技术的复杂问题。这里，我们关注的主要是组织，不过也将简要介绍操作系统设计问题。

17.2 对称多处理器

直到前不久，所有的单用户个人计算机和大多数工作站还只含有单一通用微处理器。对性能需求的增长和微处理器价格的持续下跌，促使厂商推出了 SMP 系统。SMP 既指计算机硬件体系结构，也指反映此体系结构的操作系统行为，可把 SMP 定义为具有如下特征的独立计算机系统：

（1）有两个或更多功能相似的处理器。

（2）这些处理器共享同一主存和 I/O 设备，以总线或其他内部连接机制互连在一起；这样，存储器的存取时间对每个处理器大致都是相同的。

（3）所有处理器共享对 I/O 设备的访问，或通过同一通道，或通过提供到同一设备路径的不同通道。

（4）所有处理器能完成同样的功能（术语“对称”的由来）。

（5）系统被一个集中式操作系统（OS）控制，OS 提供各处理器及其程序之间的作业级、任务级、文件级和数据元素级的交互。

第（1）点到第（4）点是不言而喻的，第（5）点表示了与集群系统之类的松耦合多处理系统的对照。后者所交互操作的物理单位通常是消息或整个文件；而 SMP 中，各个的数据元素能成为一个交互级别，并且处理器间能够有高度的相互协作。

SMP 的操作系统能跨越所有处理器来调度进程或线程。SMP 有如下几个超过单处理器的潜在优点：

- **性能**（performance）：如果可以对一台计算机完成的工作进行组织，使得某些工作部分能够并行完成，则具有多个处理器的系统与具有同样类型的单处理器的系统相比，将产生更高的性能（见图 17-3）。
- **可用性**（availability）：在一个对称多处理器中，所有处理器都能完成同样的功能，故单个处理器的故障不会造成系统的停机，系统可在性能降低的情况下继续运行。
- **增量式增长**（incremental growth）：用户可以通过在系统中添加处理器来提高系统性能。
- **可扩展**（scaling）：厂商能提供一个产品范围，它们基于系统中配置的处理器数目不同而有不同的价格和性能特征。



图 17-3 多道程序和多处理

应当注意，这些只是潜在的优点，而不是保证必有的，操作系统必须提供相应工具和功能，以开发 SMP 系统中的并行性。

SMP 的一个有吸引力的特点是，多个处理器的存在对用户是透明的：由操作系统实际管理各个处理器上的进程或线程的调度，以及处理器间的同步。

17.2.1 组织

图 17-4 说明了多处理器系统的一般组成情况，其中可以有两个或更多的处理器。每个处理器是自包含的，即包括有控制器、ALU、寄存器等。通常，还会有一级或多级 cache。通过某种形式的互连结构，每个处理器能访问共享的主存储器和 I/O 设备。各处理器之间通过存储器（位于公共数据区中的消息或状态信息）能相互通

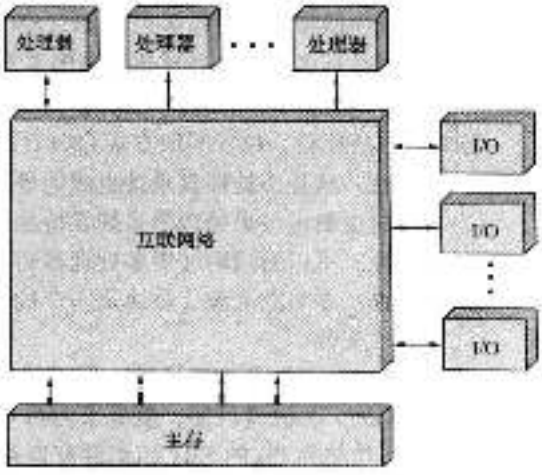


图 17-4 紧耦合多处理器的常规结构图

信。处理器之间直接交换信号是可能的。存储器经常组织成允许对存储器各个块的多重同时存取。在某些配置中，每个处理器除了共享资源之外，还允许有它专用的主存储器和 I/O 通道。

对个人计算机、工作站和服务器而言，最普通的组织是分时共享总线。分时共享总线是构成一个多处理器系统的最简单结构（见图 17-5）。分时共享总线系统的结构和接口基本上同于使用总线互连的单处理器系统。总线由控制、地址和数据线组成。为便于来自 I/O 处理器的 DMA 传送，应提供如下特征：

- **寻址**：必须能区别总线上各模块，以确定数据的源和目标。
- **仲裁**：任何 I/O 模块都能临时行使主控器（master）功能。因此需要提供一种机制来对总线控制的竞争请求进行仲裁，这可使用某种类型的优先级策略。
- **分时复用**：当一个模块正在控制总线时，其他模块是被锁住的，而且需要的话，应能挂起它的操作直到当前的总线访问被完成。

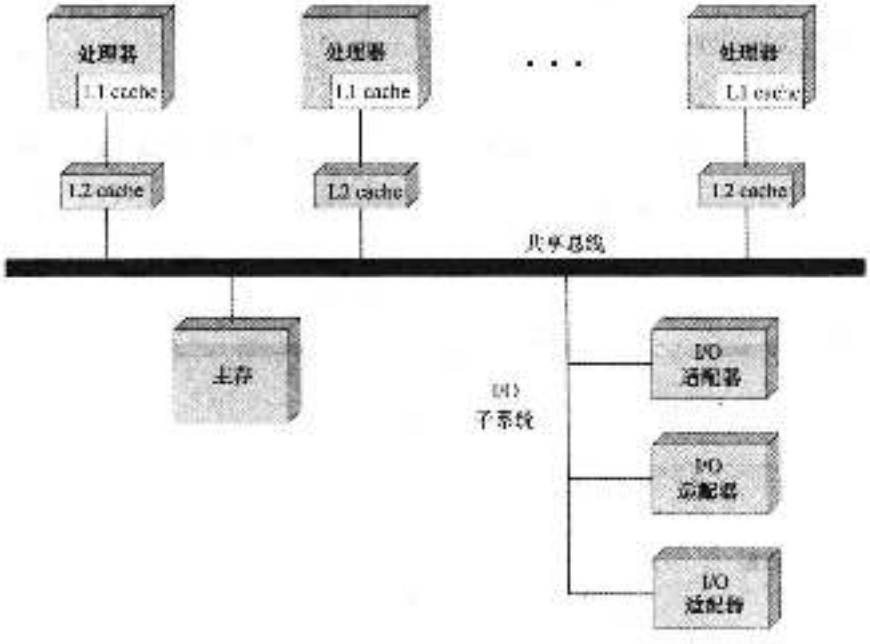


图 17-5 对称多处理器组织

这些单处理器特征在对称多处理器（SMP）配置中是直接可用的。稍后将会看到，SMP 中有多个处理器以及多个 I/O 处理器都试图掌管总线，对一个或多个存储器模块进行访问的更为复杂的情况。

与其他方法比较，总线组织方式有如下几个优点：

- **简易性**：这是多处理器系统组成的最简单方式。物理接口以及每个处理器的寻址、仲裁和分时逻辑可与单处理器系统保持相同。
- **灵活性**：可以通过附加更多处理器到总线的方式来扩充系统，这一般来说也是容易的。
- **可靠性**：本质上来说，总线是一个被动介质，并且总线上任一设备的故障不会引起整个系统的失败。

总线组织的主要缺点在于性能。所有的存储器访问都要通过公共总线，于是系统速度受限于总线周期时间。为改善性能，就要求为每个处理器配置 cache，这将有效地减少总线访问次数。一般来说，工作站和 PC 的 SMP 组织都有两级 cache。L1 cache 是内部的（与处理器同一芯片），L2 cache 或是内部的或是外部的。现在，某些处理器还使用了 L3 cache。

cache 的使用导致某些新的设计考虑，因为每个局部 cache 只保存部分存储器的映像，如果在某个 cache 中修改了一个字，可想象出其他 cache 中的此字将会是无效的。为防止这个问题，必须通知其他处理器：已经发生了修改。这个问题称为 cache 一致性（cache coherence）问题，并且一般是用硬件解决，而不是由操作系统去解决，17.4 节将专门讨论这个问题。

17.2.2 多处理器操作系统设计考虑

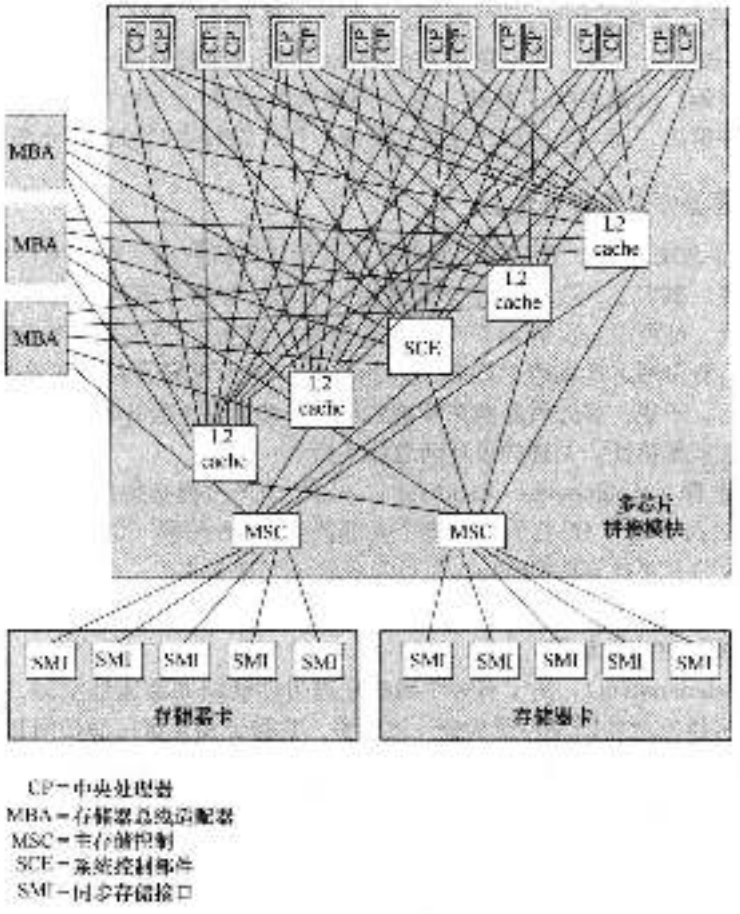
一个 SMP 操作系统（OS）管理多个处理器和其他计算机资源，而使用户感觉到单一操作系统控制着系统资源。事实上，这种配置应呈现为一个单处理器多道程序设计系统。不论是 SMP 还是单处理器情况，都可以一次启动多个作业或进程；调度它们的执行和分配资源是操作系统的责任。用户可构造使用多进程或多线程的应用程序，而无需顾及程序是在单一处理器还是在多个处理器上进行。于是，多处理器操作系统必须提供多道程序设计系统的所有功能，以及伴随多个处理器而来的其他特征。关键的设计问题如下所示：

- **同时并发进程**（simultaneous concurrent process）：OS 例程必须是可重入的，以允许几个处理器可同时执行同一 OS 代码。当多个处理器执行 OS 的同一部分或不同部分时，要恰当地管理 OS 的表和其他数据结构，以避免死锁或无效操作。
- **调度**（scheduling）：任何处理器都可完成调度，因此必须避免冲突。调度程序必须将就绪进程指派给可用处理器。
- **同步**（synchronization）：由于有多个活动进程可能访问共享地址空间，或共享 I/O 资源，因此必须提供有效的同步。同步是一种机制，它保证相互排斥的访问和事件的次序。
- **存储管理**（memory management）：正如第 8 章所讨论的，多处理器上的存储管理必须解决单机系统上出现的所有问题。此外，操作系统还要发掘可用的硬件并行性，如多端口存储器，以实现最佳性能。操作系统必须协调不同处理器上的分页机制，以保证几个处理器共享页（段）时的一致性，并在发生页替换时决定合适的操作。
- **可靠性和容错**（reliability and fault tolerant）：面对处理器故障，操作系统应提供优雅的降级使用。调度程序和操作系统的其他部分必须识别处理器的失效，并相应地重构管理表。

17.2.3 大型机 SMP

大多数 PC 和工作站 SMP 使用总线互连策略，如图 17-5 所示。考察另一种方法是有指导意义的，它已用于 IBM zSeries 大型机系列的最近实现 z990 中［SIEG04，MAK04］。这个系列的产品散布在一个较宽的范围内，由带一个主存储器卡的单处理器到具有 48 个处理器和 8 个存储器卡的高端系统。配置（图 17-6）的主要部件如下：

- **双核处理器芯片**（dual-core processor chip）：每个处理器芯片都包含两个等同的中央处理器（CP）。此 CP 是一个 CISC 超标量微处理器，它的大多数指令以硬布线式实现，其余的指令由垂直微代码执行。每个 CP 包括一个 256KB 的 L1 指令 cache 和一个 256KB 的 L1 数据 cache。
- **L2 cache**：每个 L2 cache 芯片的容量是 32MB。这些 L2 cache 排列成 5 个一组，每组支持 8 个处理器芯片，并提供对整个主存空间的访问。
- **系统控制部件**（system control element，SCE）：该 SCE 仲裁系统通信并在维护 cache 一致性方面起核心作用。
- **主存储控制**（main store control，MSC）：MSC 与 L2 cache 和主存互连。
- **存储器卡**（memory card）：每卡有 32GB 的存储容量。最大可配置的存储器由 8 个存储器卡组成，总容量为 256GB。存储器卡经由同步存储接口（synchronous memory interface，



CP＝中央处理器
MBA＝存储器总线适配器
MSC＝主存储控制
SCE＝系统控制部件
SMI＝同步存储接口

图 17-6 IBM z990 多处理器结构

SMI）连接到 MSC。

- **存储器总线适配器**（memory bus adapter，MBA）：MBA 提供到各类 I/O 通道的接口。去往/来自通道的通信量都要直接走到 L2 cache。

z990 中的核处理器与当代其他处理器相比有些不一般。虽然它是一个超标量处理器，但它以严格的体系结构次序执行指令。不过，它通过具有更短的流水线，相比其他处理器更多、更大的 cache 和 TLB，以及其他性能增强的特色弥补了这个不足。

z990 系统组成 1～4 个**叠箱**（book）。每箱是一个可插拔单元，可容多达 12 个处理器，高达 64GB 的存储器、I/O 适配器和一个连接所有这些部件的系统控制部件（SCE）。每箱中的 SCE 含有一个 32MB 的 L2 cache，它起着该箱的中央一致性维护点的作用。L2 cache 和主存二者都可由此箱或系统中其他三箱的处理器或 I/O 适配器来存取。SCE 和 L2 cache 芯片亦与环形配置（ring configuration）中其他箱的相应部件连接。

IBM z990 SMP 配置有几个值得注意的特点。下面依次对其进行讨论：

- 交换式互连。
- 共享的 L2 cache。

1. 交换式互连

对于 PC 和工作站的 SMP，使用单一共享总线是常见的做法（图 17-5），此时单一总线变成影响可扩展性（扩展到更大规模的能力）的瓶颈。z990 采用了两种方法来解决这个问题。首先，主存被分成多个卡，每个卡有自己能高速处理存储器存取的存储控制器。主存的平均流通负荷

已被减轻，因为到存储器各个部分有独立的路径。每箱包括两个存储器卡，对于4箱的最高配置总共可有8个卡。其次，由处理器（实际上由L2 cache）到单个存储器卡的连接不再是共享总线形式，而是点对点的链路。每个处理器芯片到同箱的每个L2 cache都有一条链路；每个L2 cache都有一链路连接到MSC，再经由它连接到同箱两个存储器卡的每一个。

每个L2 cache仅连接到同箱的两个存储器卡。系统控制部件提供到配置中其他箱的链路（图中未画），故全部的主存可被所有处理器存取。

用点到点的链路而不用总线，亦提供了对I/O通道的连接。箱上的每个L2 cache连接到该箱上的各个MBA，MBA再连接到I/O通道。

2. 共享的L2 cache

在SMP的典型两级cache策略中，每个处理器有自己专用的L1 cache和专用的L2 cache。近年来，对于共享的L2 cache的关注日益增长。在第三代（G3）大型机SMP的早期版本中，IBM使用的是专用L2 cache，在后来的版本（G4、G5和z900系列）中使用的是共享L2 cache。进行这种改变有两个考虑：

（1）由G3到G4，IBM将微处理器的速度提高了一倍。如果仍维持G3的组织，则现有总线的流量会显著增加。与此同时，又要求尽量使用G3部件。如果不显著提升总线性能，BSN将变成瓶颈。

（2）典型的工作负载分析揭示出，处理器之间存在大量共享的指令和数据。

这些因素使G4设计小组决定使用一个或多个L2 cache，每个被多个处理器共享（每个处理器仍有一个专用的片内L1 cache）。乍一看，共享一个L2 cache可能是个糟糕的想法，因为处理器现在必须竞争对单个L2 cache的存取，而使处理器对存储器的访问变慢。然而，如果有足够多的数据实际上是被多个处理器共享，那么共享的L2 cache能提高吞吐率，而不是降低它。如果数据是共享的，并在共享cache中找到了，那么得到它们要比经由总线得到它们快得多。

17.3 cache一致性和MESI协议

当代多处理器系统中，通常是每个处理器都有一级或两级cache。这种组织是出于性能的考虑，然而也产生了一个称为cache一致性的问题。此问题是：同一数据的多个副本可能同时存在于不同的cache中。若允许处理器自由地修改它们自己的副本，就会导致不同cache对存储器中同一数据的反映不一致。第4章曾定义了两种通用的写策略。

- **回写**（write back）：写操作通常只是对cache进行，仅当此cache数据行由cache换出时，相应的主存内容才被修改。
- **写直达**（write through）：所有写操作既对cache也对主存进行，保证主存总是有效的。

很显然，回写法会导致不一致。如果两个cache保存同一行数据，一个cache修改了此行，另一个cache将不知道它保存的已是过时数据，不后读此行数据，将产生一个无效结果。即使采用写直达法，也会出现不一致，除非其他cache监督存储器的访问情况，或接收某些直接的修改指示。

本节我们先简要介绍解决cache一致性问题的几种方法，然后集中介绍最广泛使用的方法：MESI协议。这个协议的不同版本已分别用于Pentium 4和PowerPC的实现中。

对任何cache一致性协议，目标都是让近期使用的局部变量进入cache，并保留在那里，经历多次读和写，而使用协议来维护同时可出现在多个cache中的共享变量的一致性。cache一致性方法通常可分为软件方法和硬件方法。某些实现采用了二者相结合的策略。无论如何，分类成软件方法和硬件方法是有指导意义的，并广泛用于评价cache一致性。

17.3.1 软件解决方案

cache 一致性问题的软件解决方法的思路是：依赖于编译程序和操作系统来解决问题，力图避免附加硬件电路和逻辑。因为确定潜在不一致问题的开销由运行时转换成编译时，设计的复杂性由硬件转移到软件，故软件解决方法是很有吸引力的。从另一方面看，编译时软件方法一般采取的是保守性的判决，这会导致 cache 利用率的下降。

基于编译程序的一致性机制通过进行代码分析，来确定什么样的数据项对于高速缓存可能会变成不安全的，然后相应地标记出这些项。操作系统或硬件来防止这些不可高速缓存的项进入 cache。

最简单的方法是不准任何共享数据变量被高速缓存。但这种方法太保守了，因为一个共享数据结构可以在某些时期内是排他性使用的，并可在某些其他时期内是只读的。只有在如下的期间内：即至少一个进程去修改变量而同时至少有另一个进程会去访问此变量，才会出现 cache 一致性问题。

更有效的方法是分析代码来确定共享变量的安全期间。然后，编译程序在生成代码中插入指令，以在临界期间实施 cache 一致性的维护。已研制了几种技术用于代码分析和保证结果的正确性，详见［LILJ93］和［STEN90］。

17.3.2 硬件解决方案

基于硬件的解决方法通常称为 cache 一致性协议，它提供了对运行时潜在的不一致情况的动态识别。因为是在问题实际发生时及时解决，所以 cache 的使用更有效，性能的改善要比软件解决方法好。另外，这些方法对程序员和编译程序都是透明的，也减轻了软件开发负担。

不同的硬件解决方法在几个具体细节上有所不同，包括数据行状态信息保存在何处，信息是如何组织的，在何处实施一致性维护以及实施结构又是怎样的等。通常，硬件解决方法分成两大类：目录协议（directory protocol）和监听协议（snoopy protocol）。

1. 目录协议

目录协议收集并维护有关数据块副本驻存在何处的信息。典型地，系统有一集中式控制器，它是主存控制器的一部分，目录就存于主存中。目录含有关于各个局部 cache 内容的全局性状态信息。当某个 cache 控制器产生一个访问请求时，集中式控制器检查此请求并发出必要的命令，以在存储器和 cache 之间，或 cache 相互之间传送数据。它亦负责保持状态信息的更新。于是，任何一个能影响 cache 数据行全局状态的局部动作必须报告给中央控制器。

一般来说，控制器维护着关于哪个处理器拥有哪个数据行副本的信息。在处理器向局部的 cache 行副本写入之前，它必须向控制器请求排他性存取权。在同意这次排他性存取之前，控制器发送一个消息给所有包含该行副本高速缓存的处理器，以强迫每个处理器使它的副本无效。接收到这些处理器返回的认可后，控制器才将排他性存取权授给请求的处理器。当一行已授权给某处理器专有，而另外的处理器企图读此行时，它将送出一个未命中提示给控制器。控制器则向拥有此行的处理器发出命令，要求它将此行写回到主存。于是，现在此行可被原先的处理器和请求的处理器读共享了。

目录协议有中央瓶颈的缺点，另外各 cache 控制器和中央控制器之间的通信开销也大。然而，在采用了多条总线或某种另外的复杂互连结构的大型系统中，它们是很有效的。

2. 监听协议

监听协议将维护 cache 一致性的责任分布到多处理器中各个 cache 控制器上。一个 cache 必须识别出何时它保有的一行是与其他 cache 共享的。当对共享的 cache 行完成一个修改动作时，它必须通过一种广播机制通知所有其他 cache。各个 cache 控制器应能监听互联网络，以得到这些

广播通知，并做出相应的反应。

监听协议非常适合于基于总线的多处理器。因为共享的总线能为广播和监听提供简洁的方式。然而，使用局部 cache 的一个目标就是希望避免或减少总线访问，因此必须小心地设计，不使由于广播和监听而增加的总线传输开销抵消掉使用局部 cache 的获益。

监听协议已开发出两种基本方法：写-作废（write-invalidate）和写-更新（write-update）方法。使用写-作废协议，系统任一时刻可有多个读取者，但只有一个写入者。最初，一个数据行可能在几个 cache 中为读目的而共享。当某个 cache 要对此行完成一个写操作时，它首先发出一个通知，以使其他 cache 中此行变为无效。使此行成为要执行写操作的 cache 专有。一只 cache 行变为专有，拥有者处理器就可进行本地写操作，直到某些其他处理器要求此同一数据行。

写-更新协议又称为写-广播（write-broadcast）协议。根据这样的协议，系统可有多个写入者以及多个读取者。当一个处理器打算修改一个共享行时，欲被修改的特定数据字亦被广播到所有其他 cache，于是包含此行的 cache 能同时进行写入修改。

两种方法谈不上哪个更好些，性能取决于局部 cache 的数量和存储器读、写的样式。某些系统的实现既可适用写-作废协议，也可适用写-更新协议。

写-作废协议已广泛用于像 Pentium 4 和 PowerPC 这样的商业多处理器系统中。它（使用 cache 标记中的额外两位）标出每个 cache 行的状态是修改（Modified）、专有（Exclusive）、共享（Shared）还是无效（Invalid）。因此，写-作废协议也被称为 MESI 协议。本节的剩余部分，我们将考察它在多处理器的各局部 cache 中的使用。出于简化的考虑，我们将不考察涉及局部的以及经过分布式多处理器的 L1 和 L2 cache 协调机制。如果对这些协调机制进行考察，也不会使我们了解更多的原理，仅会使讨论变得很复杂。

17.3.3 MESI 协议

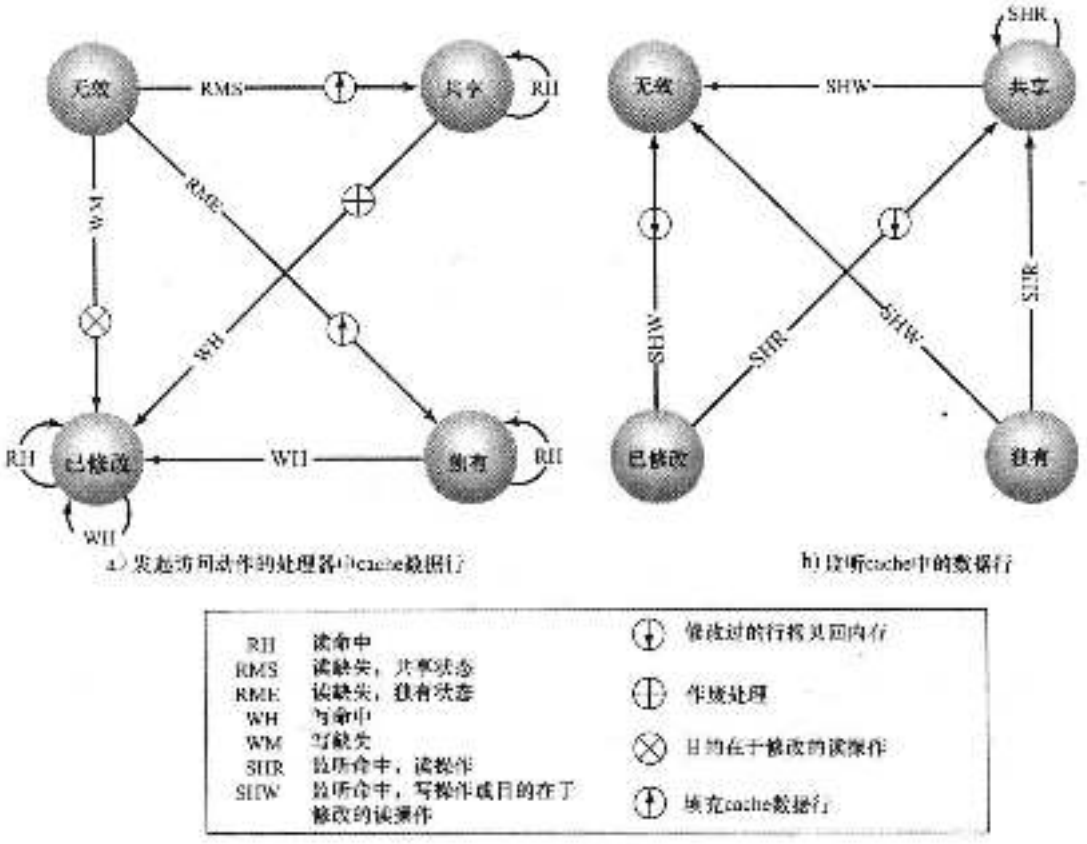
为维护 SMP 中的 cache 一致性，数据 cache 通常支持 MESI 的协议。根据 MESI 协议，数据 cache 的每行包括一个两个状态位的标记（tag），这样每行可处于下列 4 种状态之一：

- **修改态**（modified）：此 cache 行已被修改（不同于主存），并仅在这个 cache 中。
- **专有态**（exclusive）：此 cache 行同于主存，但不出现于任何其他 cache 中。
- **共享态**（shared）：此 cache 行同于主存，并可出现于另外的 cache 中。
- **无效态**（invalid）：此 cache 行不含有效数据。

表 17-1 总结了 4 种状态的意义。图 17-7 展示了 MESI 协议的状态图。注意，每个 cache 行有自己的状态位，因此每个 cache 行本身的状态变化就实现了所示的状态图。图 17-7a 表示的是，由于连接这个 cache 的处理器发起动作而发生的状态转换。图 17-7b 表示的是，由于监听到公共总线上的事件而发生的状态转换。对处理器发起的和总线发起的动作，分别给出状态图，这样有助于弄清楚 MESI 协议的逻辑。注意，任何时候 cache 行只处于单一状态，如果下一个事件来自所连接的处理器，那么状态转换由图 17-7a 给出；如果下一个事件来自总线，那么状态转换由图 17-7b 给出。下面让我们更详细地考察这些转换。

表 17-1 MESI cache 数据行状态

	M 已修改	E 独有	S 共享	I 无效
该高速缓存数据行有效	是	是	是	否
内存副本是……	旧的	有效	有效	—
其他 cache 有此数据行的副本	否	否	可能	可能
写入到该数据行	不发送到总线	不发送到总线	发送到总线并更新 cache	直接发送到总线


无效
RMS
共享
RH
RME
WM
WH
已修改
WH
RH
WH
SHR
无效
SHW
共享
SHW
SHR
SHW
SHR
已修改
独有
RH
WH
RH
RH
WH
RH
WH
WH
RH
WH
RH
WH
R

2. 读命中

当读命中出现在当前局部 cache 行时，处理器简单地读取所需数据项，没有任何状态改变；仍保留在修改、共享或专有状态。

3. 写缺失

局部 cache 出现写缺失时，处理器启动一个存储器读，来读取含有此未命中地址的主存行。为了写入修改的目的，处理器在总线上发出一个意为“用于修改的读”（read; with-intent-to-modify，RWITM）信号。当此行装入后，它立即标记为修改态，并进行写入修改。依据其他 cache 状况，在进行数据行装入时会有两种情况。

第一种情况，其他某个 cache 有此行的修改副本（状态为修改态）。这种情况下，被通知的处理器告诉启动处理器，它有此行的修改过的副本；启动处理器放弃总线并等待；而被通知的处理器获得总线访问权，并将它修改过的副本写回主存，并将此 cache 行状态转换成无效（因为启动处理器正要修改此行）。接着，启动处理器再一次在总线上发出 RWITM 信号，然后从主存中读取该行，修改 cache 中的行，并把该行标识为修改状态。

第二种情况，任何其他 cache 都没有所要求行修改过的副本，也就没有信号返回，启动处理器着手它请求的行读入并修改的操作。在此期间，若一个或多个处理器有此行的干净副本并处于共享状态，则每个处理器都要将它的副本变为无效；若一个处理器有此行的干净副本而处于专有状态，则它同样要将此副本作废。

4. 写命中

当写命中出现在当前局部 cache 行上时，效应取决于局部 cache 中此行的当前状态：

- **共享**：完成写修改之前，处理器必须先得到此行的专有权。处理器在总线上通告它的打算，cache 中含有此行共享副本的各处理器将各自的 cache 行状态由共享变成无效。然后，启动处理器完成写修改，并将它的 cache 行副本状态由共享转换成修改。
- **专有**：处理器已有对此行的排他性控制，故它只需要简单地完成写修改，并将此行的状态由专有变为修改。
- **修改**：处理器已有对此行的排他性控制，并已标志此行为修改态。它只需要简单地完成写修改。

5. L1-L2 cache 的一致性

至此，本书所介绍的 cache 一致性是通过连接到同一总线或其他 SMP 互连结构上的各 cache 之间的协同动作来实现的。一般而言，这些 cache 是 L2 cache，而每个处理器还有一个 L1 cache，它不直接连到总线上，因而也不能直接参与监听协议的活动。于是，需要某种策略来维护 SMP 配置中跨越两级 cache 和跨越所有 cache 的数据完整性。

策略是扩展 MESI 协议（或任何其他 cache 一致性协议）到 L1 cache。于是，每个 L1 cache 行都包括指示状态的位。本质上，目标如下：对于既出现在 L2 cache 中又出现在相应 L1 cache 中的任何行，L1 行的状态应追踪 L2 行的状态。实现此目标的最简单方式是在 L1 cache 使用写直达策略；这种情况下，写直达是到 L2 cache 而不是到存储器。L1 写直达策略迫使对 L1 行的任何修改都送出到 L2 cache；于是，使这一修改对其他 L2 cache 都是可见的。L1 使用写直达策略要求 L1 的内容必须是 L2 内容的子集。这又暗示 L2 cache 的关联度应等于或大于 L1 cache 的关联度。IBM S/390 SMP 采用了 L1 写直达策略。

如果 L1 cache 使用了回写策略，两个 cache 间的关系将会更复杂。已有几种方法用于维护这种情况下的 cache 一致性。其中一种方法用于 Pentium II，详见[SHAN05]。

17.4 多线程和片上多处理器

处理器性能一个重要的评测指标是它执行指令的速率。这可表示成：

$$\text{MIPS 速率} = f \times IPC$$

这里，f 是处理器时钟频率，单位为 MHz；IPC（instructions per cycle）是平均每周期执行指令数。于是，设计者在两个方面追求增强性能的目标：提高时钟频率和提高平均每周期执行的指令数，或严格地说，即提高在单个处理器周期期间所完成的指令数。正如在前面几章所看到的，设计者通过使用指令流水线和在超标量体系结构中使用多条并行的指令流水线办法来提高 IPC。在流水线或多条流水线设计中，其原理性问题是各个流水段利用的最大化。为提高吞吐率，设计人员提出了更为复杂的机制，例如以不同于指令流顺序的次序执行某些指令，以及猜测执行一些可能不会被执行的指令。但正如2.2节所讨论过的那样，由于复杂性和所涉及的功率消耗，这种办法最终会达到一个限制点。

一种替代的办法是多线程化（multithreading），它允许高度的指令级并行而不增加电路复杂性和功率消耗。从本质上讲，这种办法是将指令流分成几个更细小的流，称为线程；这样，这些线程可能并行执行。

在商业系统和实验系统中所实现的各种专用多线程设计类型非常丰富。本节简要介绍其主要概念。

17.4.1 隐式和显式多线程

多线程处理器讨论中所使用的线程概念，可能同于也可能不同于多道程序操作系统设计中的软件线程概念。简要定义如下术语将是有用的：

- **进程**（process）：在计算机上运行的一个程序实例，它体现出如下两个主要特征。
  - **资源占有**（resource ownership）：进程包括一个包含进程映像的虚拟地址空间；进程映像是定义进程的程序、数据、栈和属性的集合。随着时间的推移，进程会分配到如主存、I/O 设备和文件这样资源的控制或占有权。
  - **调度/执行**（scheduling/execution）：一个进程的执行沿一条执行路径（踪迹）流经一个或多个程序。这个执行可能与其他进程的执行相交错。于是，一个进程具有各种执行状态（运行、就绪等）和派发（dispatch）优先权。进程是操作系统调度和派发的一个实体。
- **进程切换**（process switch）：将处理器由一个进程转换到另一个进程的操作。它通过保存第一个进程的所有控制数据、寄存器和其他信息，并以第二个进程的信息来替换它们而完成。[⊖]
- **线程**（thread）：进程内可派发的任务单位。它包括一个处理器上下文（程序计数器和栈指针等）和它专有的栈数据区域（以允许子例程转移）。线程顺序执行并可中断，这样处理器能转向另一线程。
- **线程切换**（thread switch）：在同一进程中，将处理器控制由一线程转换到另一线程的动作。通常，这类切换的代价要比进程切换少得多。

于是，线程关注的只是调度/执行，而进程关注的是调度/执行和资源占有。一个进程中的多个线程共享同样资源。这是为什么线程切换的耗时要比进程切换少得多的原因。像 UNIX 早期版本这样的传统操作系统不支持线程。大多数当代操作系统，如 Linux、UNIX 后期版本、Windows 等，都支持线程。应区分用户级线程和内核级线程，前者对应用程序是可见的，后者仅对操作系统是可见的。两者都称为显式线程，因它们是软件中定义的。

⊖ 术语“上下文切换”（context switch）通常出现在有关操作系统的文献和教材中。不幸的是，虽然大多数文献使用该术语表示这里所说的进程切换，但是另外一些文献使用该术语来表示线程切换。因此，为避免歧义，此处使用进程切换而非上下文切换。

所有商业处理器和大多数实验处理器至今使用的都是显式多线程。这些系统并发执行来自不同显式线程的指令，或通过在共享流水线上交错执行来自不同线程的指令，或通过在并行流水线上的并行执行来实现多线程。隐式多线程化指的是，从单个顺序程序中提取多个线程来并发执行。这些隐式线程可静态地被编译器定义或动态地被硬件定义。本节的其余部分只考虑显式多线程化。

17.4.2 显式多线程的方式

首先，多线程式处理器必须为并发执行的每个线程提供一个分立的程序计数器。不同的设计区别在于为支持并发多线程执行所添加的硬件类型和总量。通常，以线程为基础来进行取指令操作。处理器分别对待每个线程，并可用几种技术来优化单线程的执行，这包括分支预测、寄存器重命名和超标量技术。我们要实现的是线程级并行性，它与指令级并行性结合后会提供极大的性能改善。

概括来说，多线程化有如下4种办法：

- **交错式多线程**（interleaved multithreading）：这亦称为**细粒度多线程**（fine-grained multithreading）。处理器同时处理两个或多个线程上下文，每个时钟周期由一个线程切换到另一个线程。若由于数据相关性或存储器等待而使一个线程阻塞，则跳过此线程去执行另一个就绪线程。
- **阻塞式多线程**（blocked multithreading）：这亦称为**粗粒度多线程**（coarse-grained multithreading）。线程的指令连续地执行，直到如cache缺失这类引起延迟的事件出现。这类事件引发处理器切换到另一线程。这种办法在按序执行的处理器上是比较有效的，因为像cache缺失这类的延迟事件会使流水停顿。
- **同时多线程**（simultaneous multithreading，SMT）：来自多个线程的指令同时发射到超标量处理器的执行单元。它将超标量多条指令发射的能力与多个线程上下文的使用相结合。
- **片上多处理**（chip multiprocessing）：这种情况是完整处理器在单一芯片上进行复制，每个处理器处理一个分立线程。这种办法的优点是，能有效地使用芯片上可用逻辑面积而不依赖复杂性日益增长的流水线设计。

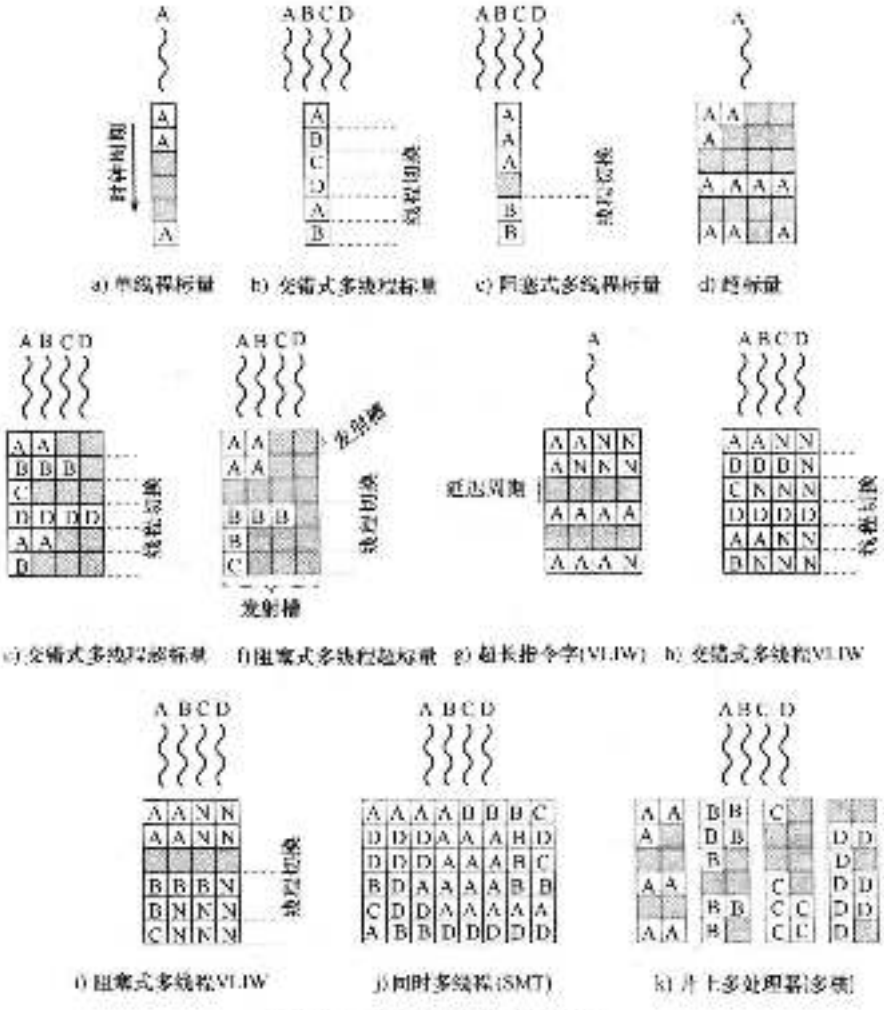
对于前两种办法，来自不同线程的指令不是同时被执行的，而是通过使用不同的寄存器组和其他上下文信息，处理器能快速地由一个线程切换到另一个线程。这使得处理器的执行资源更好地得到利用，并避免了由于cache缺失和其他延迟事件所造成的较大性能损失。SMT办法利用被复制的执行资源，完成了来自不同线程的指令真正地同时被执行。片上多处理亦允许来自不同线程的指令同时执行。

基于［UNGE02］的图17-8展示了一些涉及多线程化的可能的流水体系结构，并将它们与不使用多线程的办法相对照。每个水平行代表一个执行周期可有的发射槽[⊖]；即每行的宽度对应于单一时钟周期所能发射的最大指令数。垂直方向表示时钟周期的时间顺序。一个空槽（画有阴影）代表在流水线中未被用于执行的槽。用N表示无操作（no-op）。

图17-8的前三个说明了标量（即单发射）处理器所使用的不同办法：

- **单线程标量**（single-threaded scalar）：这是在传统RISC和CISC机器上见到的简单流水线，没有多线程。

⊖ 发射槽（issue slot）是在一给定时钟周期内可发射指令所来自的位置。回顾第14章中指令发射的介绍，指令发射操作处理器功能单元上启动指令执行的过程。这发生在指令从流水线的译码阶段向流水线执行阶段第一级前进的时候。



a) 单线程标量 b) 交错式多线程标量 c) 阻塞式多线程标量 d) 超标量

e) 交错式多线程超标量 f) 阻塞式多线程超标量 g) 超长指令字(VLIW) h) 交错式多线程VLIW

i) 阻塞式多线程VLIW j) 同时多线程(SMT) k) 片上多处理器(多核)

图 17-8 执行多线程的不同方法

- **交错式多线程标量**（interleaved multithreaded scalar）：这是最易实现的多线程化办法。通过每时钟周期切换线程，使流水线各段保持或接近满载。硬件必须在周期之间能由一个线程上下文切换到另一线程上下文。
- **阻塞式多线程标量**（blocked multithreaded scalar）：这种情况下，单线程连续执行，直到使流水停顿的等待事件出现，此时处理器切换到另一线程。

图 17-8c 表示的完成线程切换时间是一个周期的情况，而图 17-8b 表示线程切换以零周期实现。在交错式多线程情况中，假定了线程间无数据相关性和控制相关性，这就简化了流水线设计，并允许线程切换无任何延迟。然而，这取决于特定的设计和实现，阻塞式多线程可能会要求以一个时钟周期来完成线程切换，正如图 17-8 所示。若取来的指令触发线程切换，且必须由流水线逐出，就发生这种情况［UNGE03］。

虽然交错式多线程办法比阻塞式多线程呈现出更好的处理器利用率，但它这样做是以牺牲单线程性能为代价。多个线程竞争 cache 资源，这会使其中某些线程的 cache 缺失概率升高。

若处理器每周期能发射多条指令，则有更多并行执行机会可用。图 17-8d 至图 17-8i 说明了几种类型的处理器，它们都具有每周期发射 4 条指令的硬件。在所有这些情况中，一个周期仅能同时发射来自单个线程的多条指令。下面分别予以说明：

- **超标量**（superscalar）：这是基本的超标量，无多线程。直到前不久，这还是在处理器内提供并行性的最强有力办法。注意在某些周期期间，不是全部的可用发射槽都被使用。在一些周期，少于最大数的指令被发射，这称为水平损失（horizontal loss）。在另一些周期，无发射槽可用；这些是有指令却不能被发射的周期，这称为垂直损失（vertical loss）。
- **交错式多线程超标量**（interleaved multithreading superscalar）：每个周期期间，来自单一线程的尽可能多的指令被发射。正如前面所讨论过的，使用这种技术可消除由线程切换所带来的潜在延迟。然而，任一给定周期发射的指令数仍受限于给定线程内所具有的指令间相关性。
- **阻塞式多线程超标量**（blocked multithreaded superscalar）：再一次，任何周期仅来自单个线程的指令会被发射，并使用了阻塞式多线程。
- **超长指令字**（very long instruction word，VLIW）：一个 VLIW 体系结构，如 IA－64，是将多条指令放在单个字中。典型地，由编译器构造 VLIW，它把可并行执行的操作放在同一字中。在一个简单的 VLIW 机器中（图 17-8g），如果不能以可并行发射的指令来满字，就使用无操作（no-op）填充。
- **交错式多线程超长指令字**（interleaved multithreading VLIW）：这个办法所提供的效能类似于交错式多线程在超标量体系结构上所提供的效能。
- **阻塞式多线程超长指令字**（blocked multithreaded VLIW）：这个办法所提供的效能类似于阻塞式多线程在超标量体系结构上所提供的效能。

图 17-8 中所示的最后两种办法允许多个线程同时并行执行：

- **同时多线程**（simultaneous multithreading）：图 17-8j 表示一个每次能发射 8 条指令的系统。若一个线程只有高度的指令级并行性，它可在某些周期填满水平槽。在另一些周期，来自两个或多个线程的指令可被发射。若有充足的活跃线程，每周期发射最大数目的指令通常是可能的，这提供了更高级的效能。
- **片上多处理器**（chip multiprocessor）或**多核**（multicore）：图 17-8k 表示一个芯片含有 4 个处理器，其中每个都是一个双发射的超标量处理器。每个处理器各指派给一个线程，它能每周期发射来自此线程的两条指令。

比较图 17-8j 和图 17-8k，我们看到一个与 SMT 有同样指令发射能力的片上多处理器，却不能达到与 SMT 同等程度的指令级并行性。这是因为片上多处理器不能隐藏由发射来自其他线程的指令所导致的延迟。另一方面，片上多处理器应胜出有同样指令发射能力的超标量处理器，因为超标量处理器的水平损失将更大。此外，片上多处理器上的每个处理器使用多线程是可行的，当今某些机器正是这样做的。

17.4.3 示例系统

1. Pentium 4

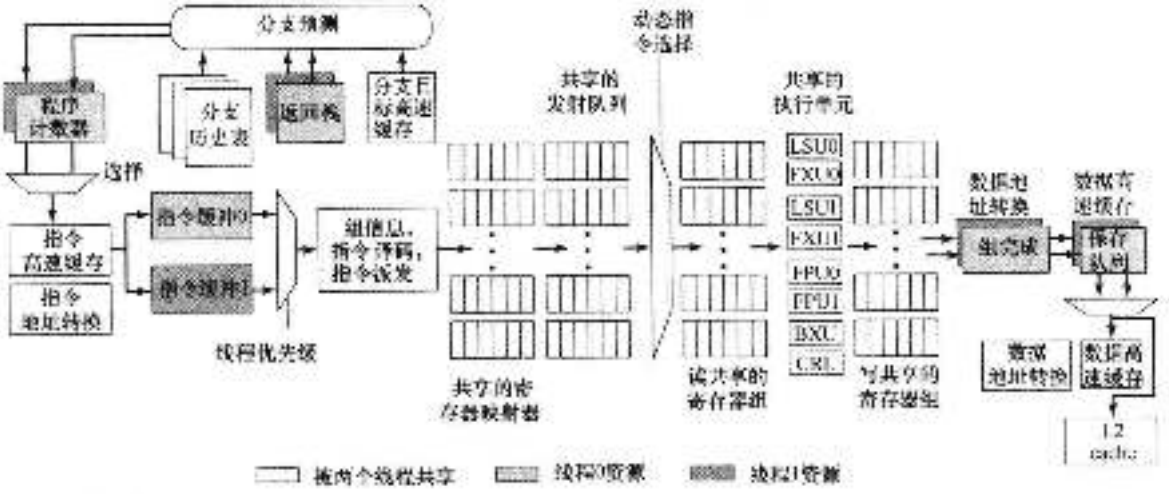
Pentium 4 的最新型号使用了一种在 Intel 文档中称为超线程（hyperthreading）的多线程技术[MARR02]。Pentium 4 的方法实际上使用了支持两个线程的 SMT。这样，单个多线程处理器逻辑上是两个处理器。

2. IBM Power 5

用于 PowerPC 高端产品的 IBM Power 5 芯片，是将片上多处理器与 SMT 相结合的产物[KALI04]。芯片具有两个分立的处理器，每个处理器是一个使用 SMT 并发地支持两个线程的多线程处理器。有趣的是，设计者对各种替代方案进行了模拟，发现单个芯片上具有两个两路的 SMT 处理器比单个的四路 SMT 处理器能提供更优异的性能。模拟研究表明，支持超出两个线程

的额外多线程化会降低性能，因为来自一个线程的数据替换另一个线程所需数据时会导致 cache 内容的频繁切换。

图 17-9 给出 Power 5 的指令流图。只是处理器中少数部件需要复制，以分立的部件支持分立的线程，其中使用了两个程序计数器。处理器交替地在两个线程间取指令，一次多达 8 条。取来的所有指令保存在一个公用的指令 cache，并共享指令翻译机构，此机构进行指令部分译码。当遇到一个条件分支指令时，分支预测机制预测转移方向，并且可能计算出转移目标地址。为预测子例程返回目标，处理器使用了返回栈，每个线程一个。



BXU = 分支执行单元
CRL = 条件寄存器逻辑执行单元
FPU = 浮点执行单元
FXU = 定点执行单元
LSU = 装载/保存单元

图 17-9 Power 5 指令数据流

然后指令被送入两个分立的指令缓冲器。接着，基于线程优先级，选择一组指令且并行译码。指令流再以程序的顺序通过寄存器换名机构，把逻辑寄存器映射到物理寄存器。Power 5 具有 120 个物理通用寄存器和 120 个物理浮点寄存器。指令然后被送入发射队列。发射队列使用对称多线程化（symmetric multithreading）方式发射指令。这意味着，处理器具有超标量体系结构，并能并行发射来自一个或两个线程的指令。流水线结束处，需要有分立的线程资源来提交指令结果。

17.5 集群

计算机系统设计的一个重要而且较新的发展是集群化（clustering）。集群化作为一种提供高性能和高可用性的方法，是对称多处理器（SMP）的替代物，并对服务器应用特别有吸引力。我们将集群系统（cluster）定义成一组完整的计算机相互连接，它们作为一个统一的计算资源一起工作，并能产生好像是一台机器的假象。术语“完整计算机”（whole computer）意指一台计算机离开集群系统仍能执行自己的任务。在文献中，集群系统中的每台计算机一般称为结点（node）。

[BREW97] 列出了集群化能提供的 4 个好处，它们也可看成是集群设计的要求或目标：

- **绝对的可扩展性**（absolute scalability）：组建一个大的集群系统，使它的能力远超过甚至最大的独立计算机是完全可能的。一个集群系统能有上百甚至几千台机器，每台机器是一个多处理器。
- **增量的可扩展性**（incremental scalability）：集群系统能以这样的方式来配置：以少量增加方式把新结点添加到集群系统中。于是，用户开始只需要一个适度规模的系统，随着需

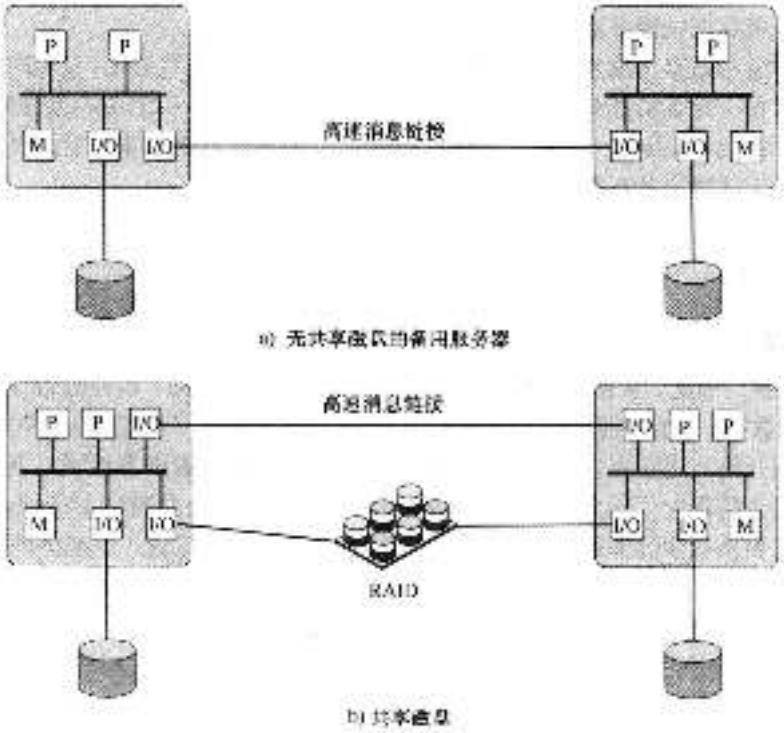
求的增长再扩展它，而无需采用以大系统替代原小系统的主要升级方式。

- **高可用性**（high availability）：因为集群系统的每个结点都是一台独立计算机，因此一个结点的故障不意味着业务的丢失。多数产品中，容错是由软件自动完成的。
- **优异的价格/性能比**（superior price/performance）：通过将商售的构件组合在一起构成一个集群系统，其计算能力等于或大于单一大型机器，而其成本却低得多。

17.5.1 集群配置

在文献中，集群系统有几种分类方式。也许最简单的分类法是，根据集群系统中的计算机是否共享对同一磁盘的存取来分类。图17-10a表示了一个两结点的集群系统，高速链路是两结点的唯一互连机制，它能用于高速消息交换，以协调系统的动作。此链路可以是与其他非集群系统计算机共享的LAN，也可以是专用的互连设施。在后一种情况下，集群系统中的一个或多个计算机将有一个到LAN或WAN的链路，这样就提供了服务器集群系统与远程客户系统之间的连接。注意，图中每台计算机被描述为一台多处理器，实际上不是必须如此，但这样做会增强性能和可用性。

在至17-10描述的简单分类中，另一种方法是共享磁盘的集群系统。这种情况下，通常在结点间仍有一条消息链路；此外，还有一个直接连接到系统内多台计算机的磁盘子系统。图中的公共磁盘子系统是一个RAID系统。使用RAID或某种类型的冗余磁盘技术，在集群系统中是很普通的，这样可使多台计算机所实现的高可用性不被作为单故障点的共享磁盘所损害。



a) 无共享磁盘的集群服务器

b) 共享磁盘

图17-10 集群配置

考察功能的可替换性可得到集群系统更清楚的分类图谱。表17-2提供了一个活动能主线的有效分类法，下面就对它进行讨论。

表 17-2 集群方法：优点和缺点

集群方法	说 明	优 点	缺 点
被动式备用	当主服务器发生故障时，次服务器接管任务处理	容易实现	成本较高，因为次服务器对其他处理任务是不可用的
主动式辅助	次服务器也用于任务处理	降低了成本，因为次服务器可用于任务处理	增加了复杂度
分立的服务器	分立的服务器有各自的磁盘。数据从主服务器连续拷贝到此服务器	高可用性	因为大量的拷贝操作，网络和服务器开销大
连接磁盘的服务器	服务器被连接到同一组磁盘，但每个服务器在磁盘组中有各自磁盘。如果一个服务器发生故障，它的磁盘被其他服务器接管	降低了网络和服务器开销，因为不需要拷贝操作	通常需要采用磁盘镜像或 RAID 技术，以便抵消磁盘故障带来的影响
共享磁盘的服务器	多个服务器同时共享磁盘访问	网络和服务器开销较低。降低了因磁盘故障而导致的停机风险	需要锁管理软件，通常会同时使用磁盘镜像或 RAID 技术

一种通用的老式方法称为**被动式备用**（passive standby），其原理很简单，一台计算机处理所有工作，而另一台计算机不工作，但时刻准备着在主服务器出故障时去接管工作。为协调机器动作，活动的主服务器要周期地发送“心跳”（heartbeat）消息给备用服务器。一旦这些消息停止到达，备用服务器将认为主服务器失败了，并立即投入运行。这种方法提高了可用性，但没有改善性能。而且，两者之间交换的信息只是心跳信息，并且二者不共享磁盘，即备用者提供了功能上的备份，但不能存取主服务器管理的数据库。

这种被动式备用通常不被看成是集群系统。术语“集群系统”是专门用于这样的多个计算机互连的系统：每个计算机都是主动地进行处理，而对外维护了一个单一系统映像。术语“主动式辅助”（active secondary）经常用于指这种配置。它又分成 3 种集群化方式，分别标识为：分立的服务器、无共享和共享存储。

一种集群化方式是，每个计算机是带有自己磁盘的分立的服务器（separate server），计算机之间无共享磁盘（见图 17-10a），这种安排提供了高性能以及高可用性。这种情况下，需要某种管理或调度软件，以将到来的客户请求指派给适当的服务器，而实现负载平衡和高利用率。同时要求系统具有故障接管能力，这意味着一台计算机在执行应用程序期间出现故障，另一台计算机能接管它并完成此应用服务。因此，数据必须在计算机之间连续不断地被复制，以使每台计算机都能存取其他计算机的当前数据。这种数据交换开销保证了高可用性，却付出了性能损失的代价。

为减少通信开销，现在大多数集群系统由连接到公共磁盘的服务器组成（见图 17-10b）。这种方法的另外说法简称为“无共享”（shared nothing），集群的公共磁盘划分成卷，每卷被单一计算机所拥有。如果某台计算机出了故障，则集群系统必须重新配置，以使另外某台计算机拥有失效计算机的卷。

使多台计算机同时共享同一磁盘也是可能的，这称为“共享磁盘”（shared disk）方法，这样每台计算机可存取所有磁盘上的所有卷。这种方法要求使用某种类型的锁机制，以保证数据一次仅能被一台计算机存取。

17.5.2 操作系统设计问题

全面发挥集群系统的硬件潜力，要求对单机的操作系统进行某些增强。

1. 故障管理（failure management）

集群系统如何管理故障，取决于所采用的集群化方法（见表 17-2）。通常，对待故障有两种方法：高可用的集群系统和容错集群系统。高可用的集群系统呈现出所有资源都正在使用的高

概率。如果出现像系统停机或磁盘卷丢失这样的故障，那么正在进行的查询可能丢失。如果进行重试，丢失的查询将由集群系统中另一台计算机提供服务。然而，集群操作系统并不保证部分已执行但未完成事务的状态。这需要应用级处理。

容错的集群系统保证所有资源总是可用的。这是通过使用冗余共享磁盘技术、回退未提交事务、提交已完成事务的机制来实现的。

应用程序和数据资源从一台失效计算机切换到集群系统中的另一台计算机上，称为故障接管（failover）。一旦故障被修复了，应用程序和数据恢复到原计算机，称为故障返回（failback）。故障返回能自动进行，但这要求故障是真正修复了，并且不会再发生；如果不是这样，那么自动故障返回会造成随后各计算机随机发生资源故障，导致性能下降和难以恢复的问题。

2. 负载平衡（load balancing）

集群系统要求一种在可用计算机之间有效平衡负载的能力，对增量式扩展的集群系统也要求具备此能力。当一台新计算机加入到集群系统中，负载平衡机制应该自动将这台计算机列入调度应用程序可使用的资源。中间件机构需要识别集群系统不同成员上的服务，并可将服务由一个成员迁移到另一个成员上。

3. 并行化计算（parallelingcomputation）

某些情况下，集群的有效利用要求并行执行来自单个应用的软件。[KAPP00] 对此问题列出了三种通常方法：

- **并行化编译器**（parallelizing compiler）：并行化编译器在编译时确定应用的哪些部分可并行执行。然后，将它们离析出来指派给集群中不同的计算机。性能取决于问题的性质和编译器设计得是否恰当。
- **并行式应用**（parallelized application）：在这种方法中，程序员从一开始就按照程序在集群上运行的设想来编写程序，并使用消息传递方式在各结点间传送所要求的数据。这对于程序员来说，负担不轻；对于某些应用来说，却是利用集群能力的最好办法。
- **参数式计算**（parametric computing）：如果应用的实质是一个算法或程序的多次执行，每次有一组不同的起始条件或参数，那么可以使用这种方法。一个典型的例子是模型模拟，它将多次运行不同的模拟方案，然后再对结果进行分析。为有效地使用参数式计算方法，需要使用参数处理工具软件以便有序地组织、运行和管理作业。

17.5.3 集群计算机体系结构

图17-11给出一个典型的集群体系结构。各个计算机通过高速LAN或开关硬件互连，每个计算机都能独立运行。各计算机都安装中间件（middleware）软件层，以允许集群操作。集群的中间件对用户提供了一个统一的系统映像，称为**单一系统映像**（single-system image）；它还通过负载平衡和响应个别部件故障的方式，来负责提供高可用性。[HWAN99] 列出如下作为集群中间件的服务和功能要求：

- **单一入口点**（single entry point）：用户登录到集群上而不是登录到个别计算机上。
- **单一文件层次**（single file hierarchy）：用户看到的是，同一根目录下的单一文件目录层次体系。
- **单一控制点**（single control point）：默认的工作站用于整个集群的管理和控制。
- **单一虚拟网络**（single virtual networking）：尽管实际的集群配置可能由多个互连的网络组成，但任一结点都能访问集群中任何其他结点。这形成单一虚拟网络的操作。
- **单一存储空间**（single memory space）：分布式共享存储器允许程序共享变量。
- **单一作业管理系统**（single job-management system）：在集群作业调度程序管理下，用户提交作业无需指定执行此作业的宿主计算机。

- 单一用户接口（single user interface）：一个公共图形接口支持所有用户，不管用户由哪台工作站进入集群。
- 单一I/O空间（single I/O space）：任一结点都能远程访问任何I/O外围或磁盘设备，而无需知晓它们的物理位置。
- 单一进程空间（single process space）：使用一致的进程标识方式，任何结点上的进程都能在远程结点上生成另一进程并与之通信。
- 检查点（checkpointing）：这个功能周期性地保存进程状态和中间计算结果，以使在故障修复之后重新运算。
- 进程迁移（process migration）：这个功能可使负载平衡。

上述的最后4项用于提高集群的可用性。其余所有项都涉及提供单一系统映像。

再返回到图17-11，集群还要包括软件工具，以使并行程序能有效地运行。

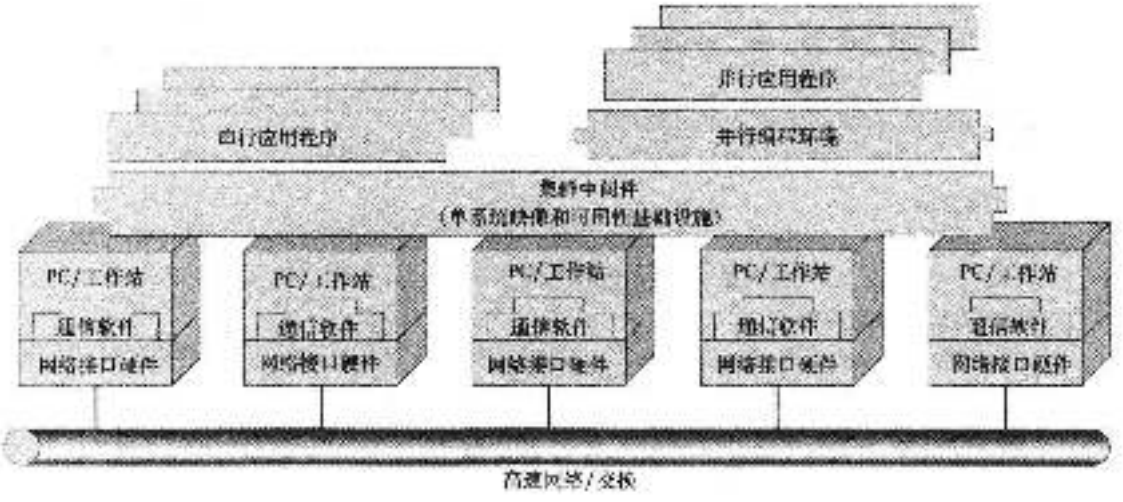


图17-11 集群计算机体系结构［BUYY99a］

17.5.4 刀片服务器

实现集群的一种常用方式是刀片服务器。刀片服务器是一种服务器体系结构，它把多个服务器模块（刀片）装入到单个机箱中。刀片服务器广泛应用于数据中心，以便节省空间，并提高系统管理效率。刀片服务器可能采用塔式或机架式安装，无论哪一种，电源都是由机箱统一提供。机箱内每个"刀片"包含它自己的处理器、内存和硬盘。

刀片服务器的一个应用例子如图17-12所示，该例子来源于［NOWE07］。大型数据中心通常装备着大量的刀片服务器，其发展趋势是给各个服务器装配10Gb/s的网络接口，以便处理这些服务器上大规模多媒体数据交换。这样的配置给连接这些众多服务器的以太网交换机带来了巨大的压力。100Gb/s的带宽能提供处理高速增长的数据交换要求的能力。于是，100Gb/s的以太网交换机就被安

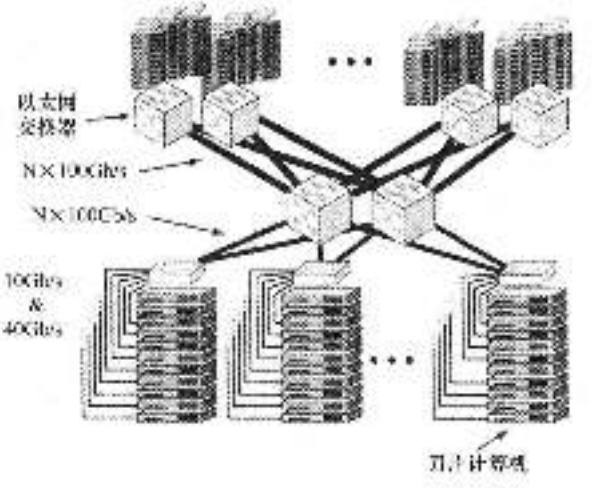


图17-12 大规模刀片服务器
站点的100Gb/s以太网配置示例

装到数据中心的上行交换链路（switch uplink）上，以提供楼宇之间、园区之间以及广域的企业网络连接。

17.5.5 集群与SMP的对比

集群系统和对称多处理器都提供了一种使用多个处理器来支持高需求应用的配置，二者都已有商业化产品可用，虽然SMP产品出现要早得多。

SMP方法的主要优势在于，SMP比集群系统更容易配置和管理。SMP更接近原来的单处理器模型，而几乎所有的应用都是以此模型来编写的。从单处理器到SMP主要的变化在于调度程序的功能增强。SMP的另一优点是，与具有可比性的集群系统相比，SMP占用较少的物理空间，并且耗电较少。另一重要优点是，SMP产品已很好地建立起来了，并且很稳定。

然而，经过长时间的竞争后，集群系统方法的优势将使它占领高性能服务器市场。在增量和绝对可扩展性方面，集群系统的优势是无可匹敌的。同时，在可用性方面，集群系统也是大大强于SMP的，因为集群系统中的所有部件都能很容易地做到高冗余度。

17.6 非均匀存储器访问

以商售产品形式提供多处理器系统的两种通常方法是SMP和集群系统。近年来，一种被称为非均匀存储器访问（NUMA）的方法已成为研究的热点，并且NUMA商业产品最近也已问世。

开始介绍之前，先定义经常出现在NUMA文档中的一些术语。

- **均匀存储器访问**（uniform memory access，UMA）：所有处理器都可使用装载和保存指令存取主存的所有部分。一个处理器对所有存储区域的访问时间是相同的，不同处理器所进行的存储器访问时间也是相同的。17.2节和17.3节所讨论的SMP组织就是一种UMA。
- **非均匀存储器访问**（nonuniform memory access，NUMA）：所有处理器使用装载和保存指令能访问所有主存部分。根据正被访问的存储器区域，一个处理器所用的存储器访问时间是不同的。这种情况对所有处理器也是真的；然而，哪个区域快些，哪个区域慢些，对不同处理器是不同的。
- **cache一致的NUMA**（cache-coherent NUMA，CC-NUMA）：在各处理器的cache之间有维护cache一致性机制的NUMA系统。

一个无cache一致性维护的NUMA系统或多或少等同于一个集群系统。令人关注的商业产品是CC-NUMA系统，它与SMP和集群系统有着明显区别。通常但不总是，这样的系统在商业文档中实际被称为CC-NUMA系统。本节也仅关注CC-NUMA系统。

17.6.1 动机

在SMP系统中，可以使用的处理器数目是有实际限制的。有效的cache策略能减少任一处理器与主存之间的总线流通量。随着处理器数目的增长，这个总线流通量也增长。还有，总线被用于传递cache一致性信号，进一步增加了总线的负担。到某一点，总线变成一个性能瓶颈。性能的降低限制了SMP中可配置的处理器数目大致在16～64之间。例如，Silicon Graphics公司的Power Challenge SMP被限制为单一系统中有64个R10000处理器；超过这个数目性能将显著降低。

SMP中的处理器数目限制也是开发集群系统的一个推动因素。然而，集群系统中是每个结点有自己的私有主存；应用程序看不到大的全局存储器。实际上，集群以软件而不是硬件维护一致性。这种存储器粒度影响着性能。为实现最大性能，必须为这种环境专门定制软件。一种达到大规模多处理而又保持SMP风格的方法是NUMA。例如，Silicon Graphics公司的Origin NUMA系

统被设计成能支持多达 1024 个 MIPS R10000 处理器［WHIT97］，Sequent NUMA-Q 系统被设计成支持多达 252 个 Pentium Ⅱ处理器［LOVE96］。

NUMA 的目标是维护一个透明的、系统范围的存储器，并准许有多个多处理器结点，每个结点有自己的总线或其他内部互连系统。

17.6.2 组织

图 17-13 描述了一种典型的 CC-NUMA 组织。这里有多个独立的结点，每个结点实际上是一个 SMP 组织。于是，每个结点有多个处理器和主存，每个处理器有自己的 L1 和 L2 cache。结点是整个 CC-NUMA 组织的基本构造块，如每个 Origin 结点包括两个 MIPS R10000 处理器，又如每个 Sequent NUMA-Q 结点包括 4 个 Pentium Ⅱ处理器。结点通过某种通信设施互连，这些设施可以是开关机构，环或其他某种网络设施。

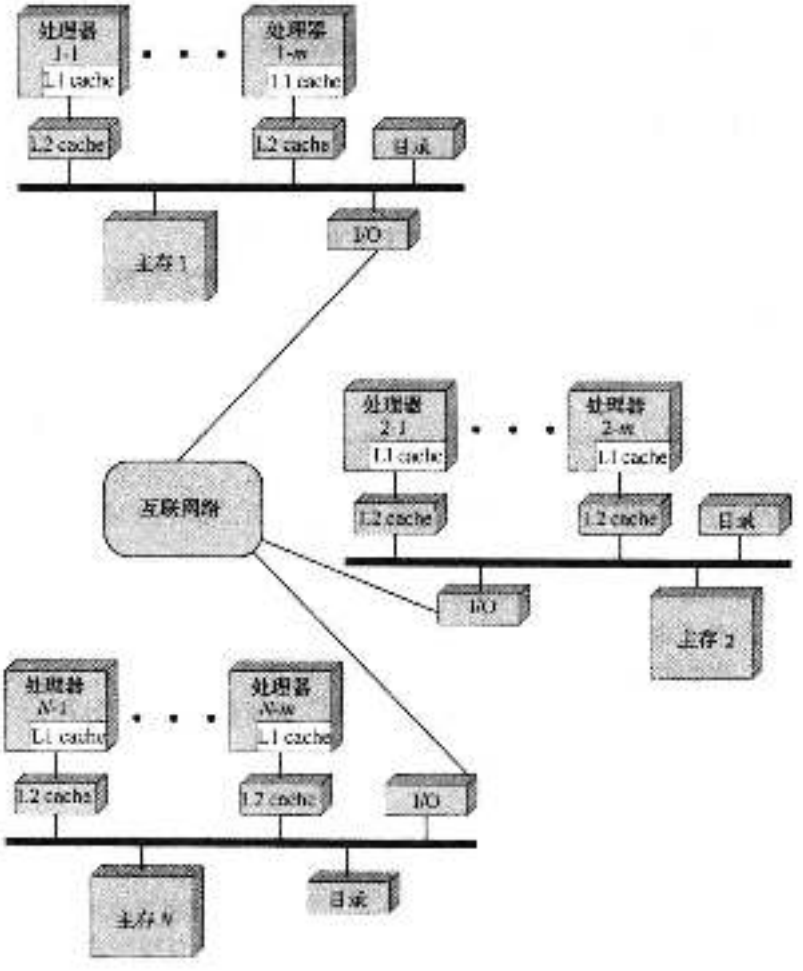


图 17-13 CC-NUMA 组织

CC-NUMA 系统内的每个结点都包含某些主存。然而，从处理器的角度看，这里只有每一位置都具有唯一系统范围地址的单一可寻址的存储器。当一个处理器启动一个存储器访问时，如果所请求的存储位置不在此处理器的 cache 中，那么 L2 cache 启动一个取操作。如果所请求的行在主存的本地部分中，经由本地总线把此行取来。如果所请求的行在远程的主存区域中，那么一个请求会由 cache 自动发出，以便通过互联网络从远端获取该行，然后将该行放到内部总线上，

并且由发送请求的 cache 从内部总线上读取。所有这些动作是自动进行的，对处理器及其 cache 是透明的。

在这种配置中，cache 一致性是主要要考虑的问题，虽然实现在细节上有所不同，但一般情况下我们可认为每个结点必须维护某种类型的目录，它给出各存储器部分的位置指示和 cache 状态信息。为看清这种策略如何起作用，我们给出一个取自［PFIS98］的例子。假设，结点 2 上的处理器 3（P2－3）要求访问存储器位置 798，而此位置在结点 1 的存储器内，那么会发生如下序列的操作：

（1）P2-3 在结点 2 的监听总线发出一个对位置 798 的读请求。

（2）结点 2 上的目录看见这个请求，并识别出此位置在结点 1 中。

（3）结点 2 的目录把一个请求发送到结点 1。结点 1 的目录收到此请求。

（4）结点 1 的目录，现着 P2-3 代理的作用，请求读取 798 的内容，就像它是一个处理器一样。

（5）结点 1 的主存响应此请求，将所要的数据放到总线上。

（6）结点 1 的目录由总线读取数据。

（7）数据被传回到结点 2 的目录。

（8）结点 2 的目录将数据放回到总线，它起着代理保存此数据的原存储器的作用。

（9）数据被读取并放入 P2-3 的 cache 中，然后递交给 P2-3。

上述顺序说明了如何由远程存储器读数据。它使用了硬件机制，从而使事务过程对处理器是透明的。在这种机制的背后，需要有某种类型的 cache 一致性协议。各种系统具体如何实现 cache 一致性协议是不同的。这里我们只进行简要的一般性讨论。首先，作为上述序列操作的一部分，结点 1 的目录应保持一个记录，记录着某些远程 cache 保存着位置 798 所在的行。其次，需要一种协同操作的协议来处理可能发生的修改。例如，一个 cache 中发生了修改，这个事实能广播到其他 cache。各个结点的目录收到这一广播后，确定本结点的各局部 cache 中是否有此行的副本，如果有，就放弃它。此修改的实际存储位置所在的结点收到此广播通知后，该结点目录维护的一个记录项指示此存储器行已无效，并一直保持下去，直到回写发生。如果任何其他处理器（本地或远程的）请求此无效行，则本地目录必须强迫发生一个回写过程，修改存储器之后，再提供此数据。

17.6.3 NUMA 的优缺点

CC-NUMA 系统的主要优点是，它能在比 SMP 更高的应用级别上提供有效性能，而不要求软件进行大的修改。由于有多个 NUMA 结点，因此任何个别结点上的总线流通量被限制到总线能处理的负载程度上。然而，如果多数存储器访问是对远程结点的，性能就开始变差。有两个理由可相信这种性能变差可以避免。第一，L1 和 L2 cache 的使用被设计成减少所有的存储器访问，包括远程的访问。如果大多数的软件都有好的时间局部性，则远程存储器访问应该不会过多。第二，如果使用了虚拟存储器，并且软件有好的空间局部性，则应用所需的数据将驻留在经常使用的有限几页上；那么，这些页可以初始装入到运行应用程序所在结点的本地存储器上。Sequent 研究人员报告，这种空间局部性出现在代表性应用中［LOVE96］。最后，通过在操作系统中包括页移植机制，能增强虚拟存储器的能力，这种机制将虚拟存储页迁移到频繁使用此页的结点上。Silicon Graphics 公司的设计人员报告了这种方法的成功应用［WHIT97］。

虽然远程访问导致的性能下降可通过上述方法弥补，但 CC-NUMA 方法也还是有缺点，文献［PFIS98］中详细讨论了其中两点。第一点，CC-NUMA 不像 SMP 那样透明，操作系统和应用由 SMP 移植到 CC－NUMA 系统上，将要求软件做些改变。这包括上面已提到过的页分配、进程分配和由操作系统完成的负载平衡。第二点与可用性有关，这是一个相当复杂的问题，并取决于

CC-NUMA 系统的确切实现。有兴趣的读者请参阅［PFIS98］。

17.7 向量计算



尽管通用的大中型计算机仍在持续地改进，然而应用软件也在持续发展，有些已超出当代大中型计算机的能力范围。包括像空气动力学、地震学、气象学、原子物理、核物理和离子物理等这类学科要求计算机解决这些物理过程中的数学计算问题。

一般而言，这些问题是以要求高精度和对大数据阵列重复完成浮点算术操作为特征的，这些问题的大多数都属于连续场模拟（continuous-field simulation）范畴。从本质上讲，一个物理现象能用二维的表面或层来描述（如邻近火箭的空气流）。这个表面可用点的网格来近似。表面上每一点的物理行为可用一组微分方程来定义。方程式用值和系数的阵列来表示，并且方程式的解涉及对数据阵列的重复算术运算。

为解决这类问题研制出了超级计算机，这类机器一般都能每秒钟完成上亿次计算。可做个对比，大中型计算机是为多道程序和密集的 I/O 操作而设计的，而超级计算机是面向刚才介绍的数值计算问题。

超级计算机应用有限，并且由于它们价格昂贵，市场也较小，主要是一些研究中心和某些尖端科学或工程研发任务的政府部门使用这类机器。与计算机技术的其他领域一样，对超级计算机性能也有一个持续增长的需求。于是，超级计算机的技术和性能也要持续发展。

还有另一类系统设计成用来解决向量计算需求，被称为阵列处理器（array processor）。虽然，超级计算机是为向量计算而优化的，但它是一个通用计算机，它也能处理标量计算和一般数据处理任务。而阵列处理器不包括标量处理；它们被大中型计算机和小型计算机用户作为外围设备配置到系统中，只运行程序中的向量计算部分。

17.7.1 向量计算的方法

超级计算机或阵列处理器设计的关键，是要认识到它们的主要任务是完成浮点数数组或向量的算术运算。在通用计算机中，这将要求迭代计算数组的每一元素。例如，考虑两个向量（一维数组）A 和 B，把它们相加结果存于 C。在图 17-14 所示的例子中，这将要求 6 次分别的相加。我们能加速这个计算吗？答案是要引入某种形式的并行性。

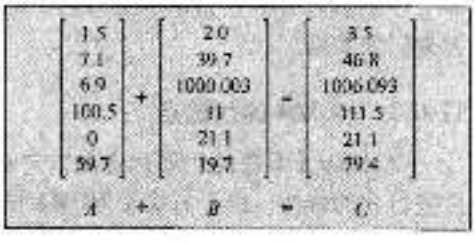


图 17-14 向量加示例

已有几种方法能实现向量计算的并行性。我们用一个例子来说明。考虑向量乘法 $C = A \times B$，这里的 A、B 和 C 都是 $N \times N$ 矩阵。C 每个元素的计算公式是：

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j}$$

这里 A、B 和 C 分别有 $a_{i,j}$、$b_{i,j}$ 和 $c_{i,j}$ 元素。图 17-15a 表示的是能在一般标量处理器上运行的，完成这个计算的 FORTRAN 程序。

改进性能的一种方法称为向量处理（vector processing）。这里假定对一维的数据向量操作是准许的。图 17-15b 是一个带有新指令格式的 FORTRAN 程序，新指令能用来进行向量计算，其中（J = 1，N）表示在给定间隔中的所有 J 下标上的运算将作为单一操作来执行。下面简要说明它是如何实现的。

图 17-15b 程序指出，第 i 行所有元素是并行计算的。行中的每个元素是一个累加，但此累加（经由 K）是串行而不是并行完成的。尽管如此，这个算法仅需要 N^2 次向量乘法，而与之相应的

标量算法却需要 N^3 次标量乘法。

另一方法是并行处理（parallel processing），由图 17-15c 说明。这种方法假定有 N 个独立的处理器能并行工作。为有效利用处理器，我们必须以某种方式将计算任务分发到各处理器。这里使用了两个原语，FORK n 原语引发一个独立的进程，并将由位置 n 启动。同时，原进程继续执行 FORK 之后的指令。每执行一次 FORK 就产生一个新进程。JOIN 原语基本上是 FORK 的反。JOIN N 语句使 N 个独立进程合并成一个进程，并继续执行 JOIN 之后的指令。操作系统必须调整好这个合并，以免执行不再继续，直到所有 N 个进程都已到达此 JOIN 指令。

```
      DO 100 I = 1, N
      DO 100 J = 1, N
      C(I, J) = 0.0
      DO 100 K = 1, N
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
100   CONTINUE
```

a) 标量处理

```
      DO 100 I = 1, N
      C(I, J) = 0.0 (J = 1, N)
      DO 100 K = 1, N
      C(I, J) = C(I, J) + A(I, K) * B(K, J) (J = 1, N)
100   CONTINUE
```

b) 向量处理

```
      DO 50 J = 1, N - 1
      FORK 100
50    CONTINUE
      J = N
100   DO 200 I = 1, N
      C(I, J) = 0.0
      DO 200 K = 1, N
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
200   CONTINUE
```

c) 并行处理

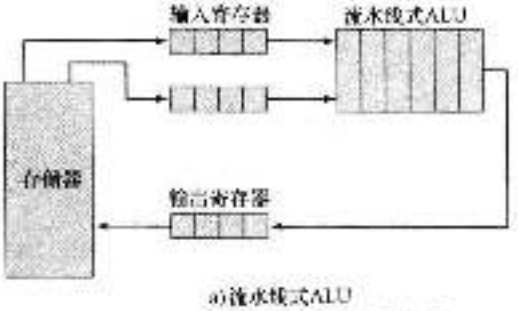
图 17-15 矩阵相乘（$C = A \times B$）

图 17-15c 中的程序是模仿向量处理程序的行为而写成的。在并行处理程序中，C 的每一列是由一个分立的进程来计算，于是，C 的给定行中各元素是并行计算的。

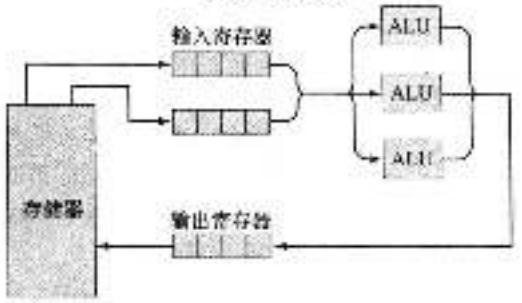
前面的讨论以逻辑的或结构的术语描述了向量计算的方法。现在，让我们转而考虑能用来实现这些方法的处理器组织类型。目前已有，并在继续推出各种组成方式，以下面三种主要类型为代表：

- 流水线式 ALU。
- 并行 ALU。
- 并行处理器。

图 17-16 描述了前两类。在第 12 章已讨论过流水线，这里把流水的概念扩展到了 ALU 操作。因为浮点运算是相当复杂的，所以这就有将一个浮点运算分解成几段的机会。这样不同的段能并发地对不同数据组操作。这由图 17-16a 来说明。浮点加法分成 4 段（参见图 9-22）：比较阶值、有效值移位、有效值相加和规格化。数据向量顺序提交给第 1 段。随着处理的进行，4 组不同的数据将在流水线中被并发操作。



a) 流水线式ALU



b) 并行ALU

图 17-16 向量计算的方法

这种组织方式适合于向量处理，这一点应是很明显的。为此，可考虑第 12 章所描述的指令流水线。处理器重复地通过取指和执行周期。在无转移时，处理器从顺序位置连续地取指令。于是流水线可以保持满载，从而实现了时间的节省。类似地，若由顺序位置向流水线式 ALU 提供数据流，它也将节省时间。一个单个的、孤立的浮点操作是不会被流水线加速的。当一个向量操作数提交给 ALU 时就能实现加速。控制器轮流地计数源通过 ALU 直到整个向量被处理。

如果向量元素是保存在寄存器中而不是存储器中，那么流水线操作能进一步增强。实际上，图 17-16a 就已暗示了这一点。向量操作数的各元素作

为一个块来装入一个向量寄存器（vector register），结果也被放入一个向量寄存器，向量寄存器只不过是同类型寄存器组成的体（bank）。下是大多数操作仅涉及寄存器的使用，只是装载和保存操作及向量运算的开始和结尾要求访问存储器。

图 17-17 所说明的机制可被看作是一个运算内的流水化（pipelining within an operation），即我们有一个单一的算术运算（如 $C = A + B$）。这个运算将被实施到向量操作数，并且流水化允许多个向量元素被并行处理。这个机制能被增强到跨越运算的流水化（pipelining across operations）。在后一情况中，有一个向量算术运算序列，并且指令流水线用于加速处理。这种跨越运算的流水化的一种方法是所谓的链接法（chaining）。Cray 超级计算机采用了这一方法。链接法的基本规则是：一个向量运算可立即开始，只要其操作数向量的第一个元素是可用的，并且功能单元（如加法器、减法器、乘法器、除法器）是空闲的。关键是，链接使一个功能单元发送出的结果，将立即被送入另一个功能单元。如果使用向量寄存器，中间结果没必要存入存储器，甚至在产生它们的向量运算完成前就可被使用。

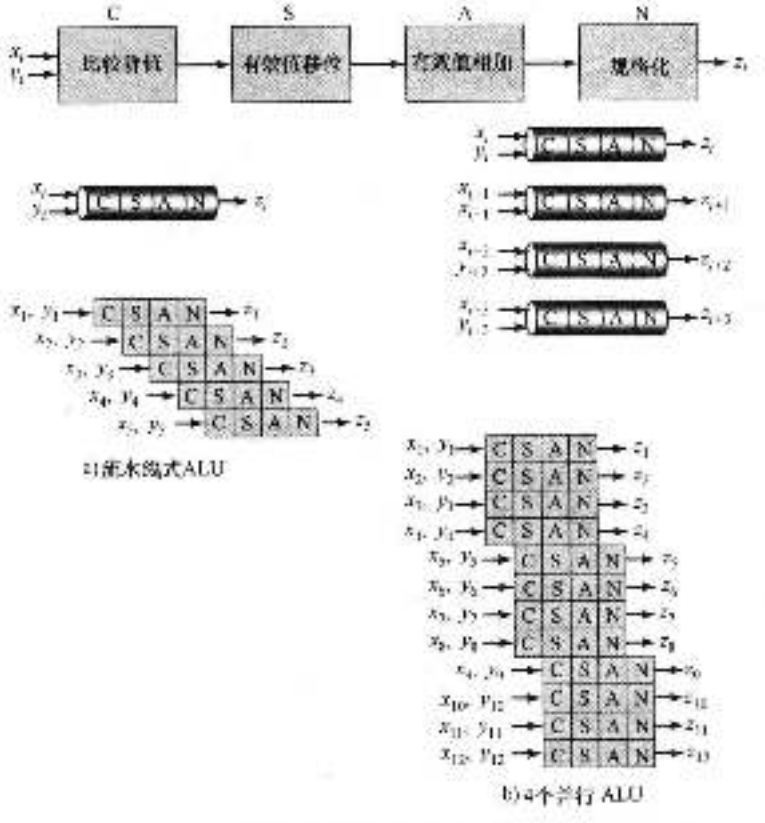


图 17-17 浮点运算的流水处理

例如，当计算 $C = (s \times A) + B$ 时，这里的 A、B、C 是向量而 s 是标量，Cray 可一次执行三条指令。装载（load）指令取来的元素，可立即进入一个流水线式的乘法器，积被送入流水线式的加法器；只要加法器一完成，其和立即被放入向量寄存器：

（1）向量装载 A→向量寄存器（VR1）；
（2）向量装载 B→VR2；
（3）向量乘法 $s \times$ VR1→VR31；
（4）向量加法 VR3 + VR2→VR4；
（5）向量保存 VR4→C。

指令2和指令3能被链接（流水化），因为它们涉及不同的存储器位置和寄存器。指令4需要指令2和指令3的结果，但它完全能与它们链接。只要向量寄存器2和寄存器3的最初元素是可用的，指令4的操作即可开始。

实现向量处理的另一方式是在一个单一处理器中使用多个ALU，在单一控制器控制下并行操作。在这种情况下，控制器指派数据到各个ALU，使它们能并行工作。每个并行的ALU亦可使用流水线。这由图17-17b说明，此例表示了4个ALU并行操作的情况。

正如流水线式组织一样，并行的ALU组织适于向量处理。控制器以一种轮转方式指派向量元素到各ALU，直到所有元素都被处理。这种组织类型要比单一ALU流水线更复杂。

最后，向量处理也能使用多个并行的处理器来实现。这种情况下，必须将任务分解成多个进程以使并行执行。仅当用于有效协调并行处理器的硬、软件是可用的，这种组织才是有效的。

现在，我们来扩充17.1节所介绍的分类以反映三种新结构，如图17-18所示。计算机组织能以一个或多个控制单元来区分。多个控制单元暗示着多个处理器。遵循我们先前的讨论，若多个处理器能在一个给定任务上协同操作，它们被称为并行处理器（parallel processor）。



图17-18 计算机组织分类

读者应知晓某些不太恰当的术语的真正意义，这些术语会在一些文献上碰到。术语“向量处理器”（vector processor）经常是等同于一个流水线式ALU组织，虽然正如我们讨论过的，并行ALU组织和并行处理器组织也是为向量处理而设计的。术语“阵列处理”（array processing）某些时候指的是并行ALU。再一次强调，尽管上述三种组织的任一种对处理阵列都是优化了的，把事情弄糟的是，“阵列处理器”（array processor）通常指的是一个辅助处理器，它接到一个通用处理器来完成指定的向量计算。阵列处理器可使用流水线式或并行式ALU方法。

目前，流水线式ALU占据市场的主导地位。流水线式系统比其他两种方法要简单些。它们的控制器和操作系统都已很好地开发出来了，能实现有效地分配资源和高性能。本节的剩余部分以一个专门例子更详细地考察这种方法。

17.7.2 IBM 3090向量机制

用于向量处理的流水线ALU组织的一个好的例子是，为IBM 370结构开发的并在高端3090系列实现的向量部件——PADE88，TUCK87]。这个部件可添加到基本系统上，但此可选部件能与基本系统高度集成在一起。它类似于像Cray系列这样的超级计算机上的向量部件。

这个IBM的向量机制使用了几个向量寄存器。每个向量寄存器实际上是一组标量寄存器，或称一个体（bank）。为计算$C = A + B$的向量和，向量A和B被装入两个向量寄存器，来自这些寄存器的数据尽可能快地被传送通过ALU，结果存于第三个向量寄存器。计算的元素和输入数据以块的形式装入寄存器，导致明显超过普通ALU操作的加速。

1. 组织

此IBM向量结构以及类似的流水线式向量ALU，提供了超越标量算术指令循环的性能提高，原因在于如下三种方法：

- 固定的并预先确定的向量数据结构，准许循环中的辅助指令（housekeeping instruction）

被快速的内部（硬件或微代码）机器操作所代替。

- 对几个连续向量元素的数据存取和算术运算能并发进行，或通过在流水线式设计中重叠这样的操作或通过多元素操作的并行完成。
- 向量寄存器用于中间结果的暂存，避免了额外的存储器访问。

图 17-19 表示了此向量机制的通常组织。虽然此向量部件看起来与主处理器是物理分立的，但它的结构是 System/370 结构的一个扩展，并与 System/370 兼容。向量部件以如下方式集成到 System/370 结构中：

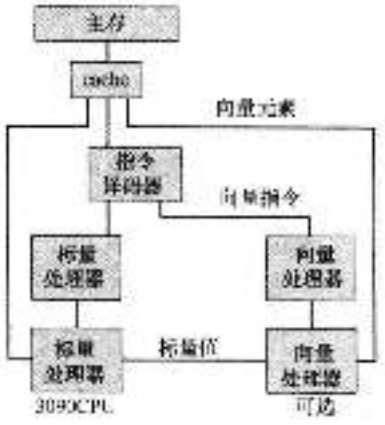


图 17-19 带向量机制的 IBM 3090

- 已有的 System/370 指令用于所有标量运算。
- 各个向量元素的算术运算产生与相应的 System/370 标量指令严格相同的结果。例如，一个设计决策涉及在一个浮点 DIVIDE 运算中结果的定义。结果是应当与标量浮点除法严格一致；还是应当允许一个近似值，使得可以高速实现但会在一位或多位低序位上引入误差？决策是这样做出的，完全保持与 System/370 结构兼容，即使为此会付出少许性能降低的代价。
- 向量指令是可中断的，并且它们的执行在相应的中断处理完成之后能由中断点恢复，以与 System/370 程序中断策略兼容的方式工作。
- 算术异常与 System/370 的标量算术指令异常相同，或是它的扩展，并且类似的处理子程序也能使用。为适应这点，使用了一个中断向量索引（vector interruption index），它能指示一个向量寄存器中受异常（如溢出）影响的位置。于是，当向量指令恢复执行时，向量寄存器中相应位置能被访问。
- 向量数据驻存在虚拟存储器中，而且以标准方式处理页故障。

这种集成程度提供了几个好处。现有的操作系统能支持这个带少许扩展的向量部件。原有的应用程序、语言编译器程序和其他软件不加修改即可运行。要利用向量部件优势的软件则可以根据需要修改。

2. 寄存器

向量机制一个重要的设计出发点是，操作数是位于寄存器还是位于存储器。IBM 组织是被称为寄存器到寄存器的，因为输入和输出的向量操作数都能在向量寄存器中按段暴示。这种方法亦用于 Cray 超级计算机上。一个用于控制数据（Control Data）机器的替代方法，是直接由存储器得到操作数。使用向量寄存器的主要缺点是程序员或编译程序必须考虑到它们。例如，假设向量寄存器的长度是 K，待处理的向量长度是 N，而且 $N > K$。在这种情况下，必须完成一个向量循环，一次完成 K 个元素的操作，而循环需重复 N/K 次。向量寄存器方法的主要优点是，操作与较慢的主存解耦了，操作主要在寄存器中发生。

使用寄存器能实现的加速可用图 17-20 说明。此 FORTRAN 例程将向量 A 乘以向量 B 产生向量 C。这里的每个向量都有一个实部（AR，BR，CR）和一个虚部（AI，BI，CI）。IBM 3090 能在每个处理器时钟或每个周期之内完成一次主存存取（读或写）。若有寄存器，它能支持每周期两次读和一次写，并能在它的算术单元中每周期产生一个结果。让我们假设能使用一种可指派两个源操作数和一个结果的指令[⊖]。图 17-20a 表示，以存储器到存储器指令，完成每一次迭代的

⊖ 对于 370/390 体系结构，所有的二操作数指令（寄存器和存储器指令，RS）都使用了两个寄存器操作数和一个存储器操作数。在例子的 a 部分中，我们假设三操作数指令中所有的操作数都来自于存储器。这一假设是出于便于比较的考虑。实际上，这种格式的指令未被向量体系结构机器采用。

计算需要总计18个周期。而采用纯寄存器到寄存器结构（见图17-19b），这个时间减少到12个周期。当然，采用寄存器到寄存器运算，在计算之前必须把向量装入向量寄存器，而在计算之后要存入存储器。对大的向量而言，这个固定的耗费是相对比较小的。图17-20c表示，在一条指令中指派存储器和寄存器两种操作数，能将每次迭代进一步减少到10个周期。这后一种指令包括在向量部件中[1]。

```
FORTRAN ROUTINE:
        DO 100 J = 1, 50
      CR(J) = AR(J) * BR(J) - AI(J) * BI(J)
  100 CI(J) = AR(J) * BI(J) + AI(J) * BR(J)
```

操作	周期
AR(J) * BR(J) → T1(J)	3
AI(J) * BI(J) → T2(J)	3
T1(J) - T2(J) → CR(J)	3
AR(J) * BI(J) → T3(J)	3
AI(J) * BR(J) → T4(J)	3
T3(J) + T4(J) → CI(J)	3
TOTAL	18

a) 存储器到存储器

操作	周期
AR(J) → V1(J)	1
BR(J) → V2(J)	1
V1(J) * V2(J) → V3(J)	1
AI(J) → V4(J)	1
BI(J) → V5(J)	1
V4(J) * V5(J) → V6(J)	1
V3(J) - V6(J) → V7(J)	1
V7(J) → CR(J)	1
V1(J) * V5(J) → V8(J)	1
V4(J) * V2(J) → V9(J)	1
V8(J) + V9(J) → V0(J)	1
V0(J) → CI(J)	1
TOTAL	12

b) 寄存器到寄存器

操作	周期
AR(J) → V1(J)	1
V1(J) * BR(J) → V2(J)	1
AI(J) → V3(J)	1
V3(J) * BI(J) → V4(J)	1
V2(J) - V4(J) → V5(J)	1
V5(J) → CR(J)	1
V1(J) * BI(J) → V6(J)	1
V4(J) * BR(J) → V7(J)	1
V6(J) + V7(J) → V8(J)	1
V8(J) → CI(J)	1
TOTAL	10

c) 存储器到寄存器

操作	周期
AR(J) → V1(J)	1
V1(J) * BR(J) → V2(J)	1
AI(J) → V3(J)	1
V2(J) - V3(J) * BI(J) → V2(J)	1
V2(J) → CR(J)	1
V1(J) * BI(J) → V4(J)	1
V4(J) + V3(J) * BR(J) → V5(J)	1
V5(J) → CI(J)	1
TOTAL	8

d) 混合指令

V - 向量寄存器
AR,BR,AI,BI - 存储器中的操作数
Ti - 存储器中的临时位置

图17-20 向量计算的可选存储

图17-21阐明了IBM3090向量机制的寄存器部分。这里有16个32位向量寄存器，它们亦能组成8个64位向量寄存器。每个寄存器元素能保存整数或浮点数。于是，向量寄存器可用32位与64位整数值以及32位与64位浮点值。

3090体系结构规定，每个寄存器包含8～512个标量元素，实际长度的选取涉及权衡。基本上，完成一次向量运算的时间由流水线启动和寄存器填充的开销，加上每个向量元素一个周期的计算所组成。于是，大的寄存器元素数能减少启动时间在计算中所占的比例。然而，这个效应必须与在进程切换时保存和恢复向量寄存器所要求的附加时间相权衡，并且要考虑到实际的成本和芯片面积限制。这些考虑导致在当前的3090实现中，每个向量寄存器有128个元素。

[1] 随后将讨论的复合指令可以进一步减少所需的时钟周期数。

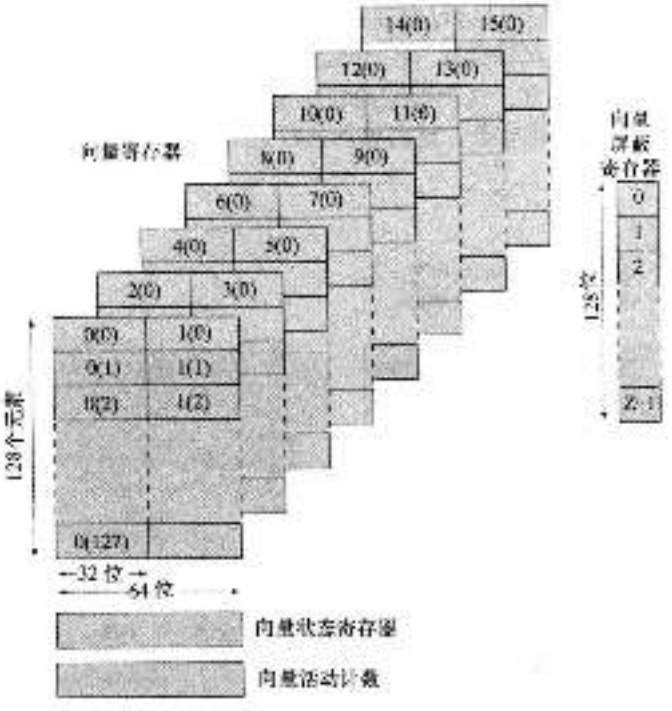


图 17-21 IBM 3090 向量机制的寄存器

向量部件需要三个另外的寄存器。向量屏蔽寄存器（vector-mask register）包含用于选择向量寄存器中的哪些元素将被一个具体运算所处理的屏蔽位。向量状态寄存器（vector-status register）包含像向量计数这样的控制字段。向量计数确定向量寄存器中有多少元素将被处理。向量活动计数（vector-activity count）保持着执行向量指令花费时间的记录。

3. 复合指令

如前面所述，为改善性能可使用链接来使指令执行重叠。IBM 向量部件的设计者未选择包含这种能力，这出于几方面的考虑。要支持复合指令，System/370 结构将被扩展到能处理复杂的中断（包括它们对虚拟存储器的影响），并在软件中需做相应的修改。一个更基本的出发点是，为实现链接而在向量部件中添加控制和寄存器存取通路的成本。

替代的是，将向量计算中最普遍的三种操作序列，即乘后加、乘后减和乘后累加，组合成单一指令（一个操作码）的形式来提供。例如，存储器到寄存器的 MULTIPLY-AND-ADD 指令由存储器读取一个向量，它被来自寄存器的一个向量乘，然后再把此积与第二个向量相加并存入第三个向量的寄存器。在图 17-20 的例子中，通过使用复合指令 MULTIPLY-AND-ADD 以及 MULTIPLY-AND-SUBTRACT，一次迭代的总时间由 10 个周期减少到 8 个周期。

不像链接，复合指令不要求使用另外的寄存器来暂存中间结果，并且它们要求较少的寄存器访问。例如，考虑如下的链：

A →VR1

VR1 + VR2→VR1

这种情况下，要求对 VR1 向量寄存器的两次存入。在 IBM 结构中，有存储器到寄存器的 ADD 指令，利用这条指令，只需要把“和”放到 VR1 中。复合指令亦避免了对反映几条指令并发执行时机器状态描述的需求，这就简化了操作系统和中断管理中的状态保存和恢复工作。

4. 指令集

表 17-3 总结了为 IBM 3090 向量机制定义的算术逻辑操作。另外，有存储器到寄存器的装载

(load) 指令和寄存器到存储器的保存 (store) 指令。注意，多数指令使用三操作数格式。还有，多数指令有几个变体，分别取决于操作数的位置。源操作数可以是向量寄存器 (V)、存储器 (S) 或一个标量寄存器 (Q)；目标操作数总是向量寄存器，除了比较指令以外。比较指令的结果要进入向量屏蔽寄存器。算上这些变体，总的类操码数量 (不同的指令数) 是171。然而，这个相当大的数量，实现起来并不像想象中那样昂贵。一旦机器提供了算术单元和将操作数由存储器、标量寄存器以及向量寄存器输送给向量流水线的数据路径，主要的硬件成本就已成形。体系结构只是以较小的附加成本，就能对这些寄存器和流水线提供丰富的指令变体集。

表 17-3 IBM 3090 向量机制：算术和逻辑指令

操作	数据类型			操作数位置			
	浮点						
	长	短	二进制或逻辑				
加	FL	FS	BI	V + V →V	V + S →V	Q + V →V	Q + S →V
减	FL	FS	BI	V − V→V	V − S→V	Q − V →V	Q − S →V
乘	FL	FS	BI	V × V→V	V × V→V	Q × V→V	Q × S→V
除	FL	FS	—	V/V →V	V/S→V	Q/V→V	Q/S→V
比较	FL	FS	BI	V · V →V	V · S →V	Q · V→V	Q · S→V
乘加	FL	FS	—		V + V × S →V	V + Q × V →V	V + Q × S →V
乘减	FL	FS	—		V − V × S→V	V − Q × V →V	V − Q × S→V
乘累加	FL	FS		P + · V→V	P + · S→V		
求补	FL	FS	BI	−V→V			
求绝对值	FL	FS	BI	|V| →V			
负绝对值	FL	FS	BI	−|V| →V			
最大	FL	FS	—			Q · V →Q	
最大绝对值	FL	FS	—			Q · V→Q	
最小	FL	FS	—			Q · V →Q	
逻辑左移	—	—	LO	· V →V			
逻辑右移	—	—	LO	· V→V			
与	—	—	LO	V&V→V	V&S→V	Q&V→V	Q&S→V
或	—	—	LO	V|V →V	V|S →V	Q|V →V	Q|S →V
异或	—	—	LO	V⊕V →V	V⊕S →V	Q⊕V→V	Q⊕S→V

说明：数据类型
- FL 长浮点数
- FS 短浮点数
- BI 二进制整数
- LO 逻辑型

操作数位置
- V 向量寄存器
- S 存储器
- Q 标量 (通用或浮点寄存器)
- P 向量寄存器中的部分和
- • 特殊操作

表 17-3 中的大多数指令不需要再说明，但两个求和指令需要进一步说明。将一个单个向量各元素加在一起的 ACCUMULATE (累加) 指令，或将两个向量的积的各元素累加的 MULTIPLY AND ACCUMULATE (乘累加) 指令，提出了一个有趣的设计问题。当然，我们希望取得 ALL 流水线的全部优点，尽可能迅速地完成这些累加运算。问题是两个数放入流水线后，其和不是立即可用的，而是要延迟几个周期。于是，当到前两个元素已全部通过整个流水线，第二个

元素才能与其和相加。为了克服这个问题，采用了产生4个部分和的方式来将向量元素相加。具体而言，元素0，4，8，12，…，124相加产生部分和0，元素1，5，9，13，…，125产生部分和1，元素2，4，10，14，…，126产生部分和2，元素3，7，11，15，…，127产生部分和3。因为流水线中的延迟大约是4个周期，故每个部分和都能在流水线中以最大速度快速进行。一个分立的向量寄存器用于保持此部分和。当原先向量的所有元素都被处理之后，4个部分和再一起相加产生最终结果。这里第二步的性能不是十分重要的，因为只涉及4个向量元素。

17.8 推荐的读物和Web站点

[CATA94]综述了多处理器原理并详细考察了基于SPARC的SMP。[STON93]和[HWAN93]中亦有SMP系统细节的介绍。

[MILE00]概述了用于多处理器的cache一致性算法和技术，重点讨论性能。[LILJ93]是有关多处理器中cache一致性问题的优秀综述。[TOMA93]包含了此课题许多重要论文的转载。

[UNGE02]是多线程处理器和片上多处理器概念的优秀综述。[UNGE03]是提出的和目前使用的显式多线程处理器的长篇综述。

[BUYY99a]和[BCYY99b]中有对集群系统的详尽论述。[WEYG01]重点不是对集群系统的技术性综述，而是对各种商业产品予以评价。[DESA05]描述了IBM的刀片服务器体系结构。

在[STON93]和[HWAN93]中可以找到有关向量计算的有益讨论。

BUYY99a Buyya, R. *High-Performance Cluster Computing: Architectures and Systems.* Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High-Performance Cluster Computing: Programming and Applications.* Upper Saddle River, NJ: Prentice Hall, 1999.

CATA94 Catanzaro, B. *Multiprocessor System Architectures.* Mountain View, CA: Sunsoft Press, 1994.

DESA05 Desai, D., et al. "BladeCenter System Overview." *IBM Journal of Research and Development*, November 2005.

HWAN93 Hwang, K. *Advanced Computer Architecture.* New York: McGraw-Hill, 1993.

LILJ93 Lilja, D. "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Computing Surveys*, September 1993.

MILE00 Milenkovic, A. "Achieving High Performance in Bus-Based Shared-Memory Multiprocessors." *IEEE Concurrency*, July-September 2000.

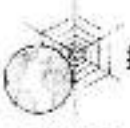
STON93 Stone, H. *High-Performance Computer Architecture.* Reading, MA: Addison-Wesley, 1993.

TOMA93 Tomasevic, M., and Milutinovic, V. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions.* Los Alamitos, CA: IEEE Computer Society Press, 1993.

UNGE02 Ungerer, T.; Rubic, B.; and Silc, J. "Multithreaded Processors." *The Computer Journal*, No. 3, 2002.

UNGE03 Ungerer, T.; Rubic, B.; and Silc, J. " A Survey of Processors with Explicit Multithreading." *ACM Computing Surveys*, March, 2003.

WEYG01 Weygant, P. *Clusters for High Availability.* Upper Saddle River, NJ: Prentice Hall, 2001.



推荐的Web站点

- IEEE Computer Society Task Force on Cluster Computing：一个促进集群计算研究和教育的国际论坛。

17.9 关键词、思考题和习题

关键词

active standby：主动式备用
cache coherence：cache 一致性
cluster：集群
directory protocol：目录协议
failback：故障返回
failover：故障接管
MESI protocol：MESI 协议
multiprocessor：多处理器
nonuniform memory access（NUMA）：非均匀存储器访问
passive standby：被动式备用
snoopy protocol：监听协议
symmetric multiprocessor（SMP）：对称多处理器
uniform memory access（UMA）：均匀存储器访问
uniprocessor：单处理器
vector facility：向量机制

思考题

17.1 列出并简要定义计算机系统组织的三种类型。
17.2 SMP 的主要特征是什么？
17.3 与单处理器相比，SMP 有哪些潜在优势？
17.4 为 SMP 设计 OS，某些关键考虑是什么？
17.5 软件的和硬件的 cache 一致性方案有何不同？
17.6 MESI 协议有 4 种状态，每种状态的意思是什么？
17.7 集群化的主要好处是什么？
17.8 故障接管和故障返回有何不同？
17.9 UMA、NUMA 和 CC－NUMA 三者之间的区别是什么？

习题

17.1 假定一个计算机系统中有 n 个处理器，每个处理器的执行速率是 x MIPS。程序代码有 α 比例部分能在 n 个处理器上同时执行，其余部分必须在单处理器上顺序执行。
(a) 请给出程序在这一系统上运行时的有效执行速率，用 n、x 和 α 表示。
(b) 若 $n=16$，$x=4$MIPS，则 α 应为何值才能产生 40MIPS 的系统性能。

17.2 一台有 8 个处理器的多处理器连接 20 台磁带机。有大量的作业提交给系统，每个作业最大要求 4 台磁带机才能完成执行。假定每个作业开始运行时只需要 3 台磁带机，长时间运行之后到结束前才短时间需要第 4 台磁带机，并假设这些作业是循环不断地被提交的。
(a) 如果 OS 中的调度程序是这样工作：没有 4 台磁带机可用，不启动作业；一旦启动作业，4 台磁带机立即被指派，直到作业结束才释放。问：一次能进行的最大作业数是多少？作为这种调度策略的结果，处于空闲的磁带机最大数和最小数各是多少？
(b) 请推荐一种策略，它能改善磁带机的利用率，而又避免系统死锁。此时，一次能进行的最大作业数是多少，空闲磁带机上限和下限又是多少？

17.3 你能预见在基于总线的多处理器上采用 cache 写一次（write-once）方法会有什么问题吗？若有，请推荐一种解决方法。

17.4 考虑一个 SMP 配置中只有两个处理器的情况，每个处理器有一个 cache，并使用 MESI 协议。经过一段时间后，两个处理器都要访问存储器的同一数据行 x，并假定最初两 cache 中有此行的无效副本。图 17-22 给出处理器 P1 读 x 行后的结果。请以此为初始情况，画出如下操作序列的后续图：
(1) P2 读 x。
(2) P1 写 x（为清楚起见，此 P1 cache 行标记 x'）。
(3) P1 写 x（此 P1 cache 行标记 x''）。
(4) P2 读 x。

17.5 图 17-23 表示两种可能的 cache 一致性协议的状态图，请给出并说明每种协议，并将其与 MESI 协议做比较。

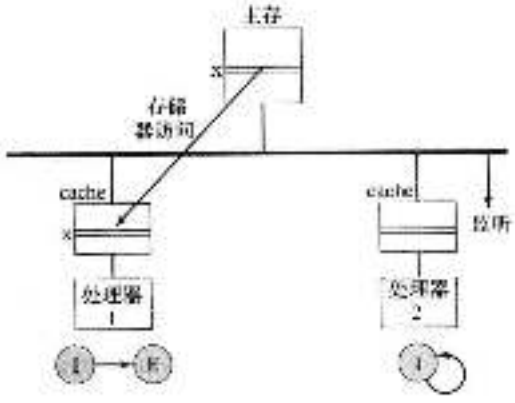


图 17-22 MESI 例子：处理器 1 读高速缓存行 x

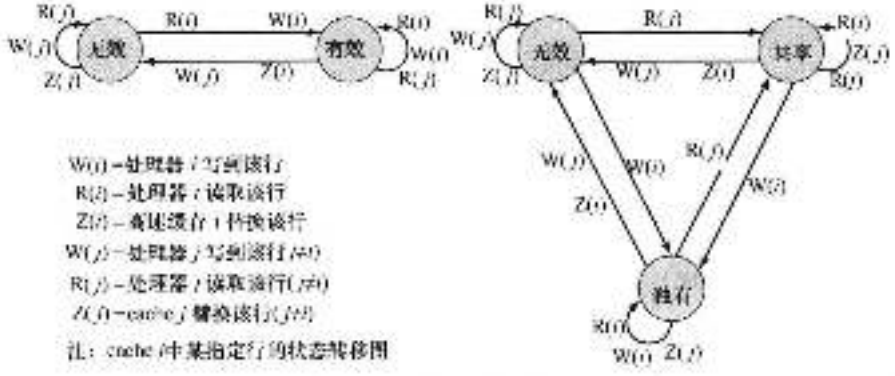


图 17-23 两种 cache 一致性协议

17.6 考虑一个有 L1 和 L2 cache 并使用 MESI 协议的 SMP。正如 17.3 节所述，L2 cache 的每行有 4 种可能的状态。L1 cache 行是否也需要 4 种状态？如果不是，请说明什么状态能取消。

17.7 IBM 大型机的一个早期版本（S/390 G4），使用了三级 cache，与 z990 相同，仅 L1 cache 在处理器芯片（称为处理器单元，PU）上。L2 cache 类似于 z990。L3 cache 在一分立芯片上，位于 L2 cache 和存储器卡之间，起着存储控制器作用。表 17-4 表示了 IBM S/390 三级 cache 配置情况下的性能。这个问题的目的在于确认包括第 3 级 cache 是否有价值。请确定只有 L1 cache 的系统的存取开销（平均 PU 周期数）；并将得到的值归一化到 1.0。然后，再确定使用 L1 和 L2 cache 以及使用全部三级 cache 的系统存取开销。记下每种情况下的改进总量，并表明你对 L3 cache 价值的看法。

表 17-4 S/390 SMP 配置上典型的高速缓存命中率 [MAK97]

存储系统	访问开销（PU 时钟周期）	存储容量	命中率（%）
L1cache	1	32KB	89
L2cache	5	256KB	5
L3cache	14	2MB	3
存储器	32	8GB	3

17.8 (a) 考虑一个带有指令和数据分立 cache 的单处理器，其命中率分别是 H_i 和 H_d。由处理器到 cache 的存取时间是 c 个时钟周期，存储器和 cache 间的块传送时间是 b 个时钟周期。令 f_i 是取指占存储器访问的比例，f_d 是数据 cache 中脏行占被替换行的比例。假设采用回写策略。请用上述定义的参数确定有效的存储器访问时间。

(b) 假定有一个基于总线的 SMP，其中每个处理器都有 (a) 部分所述的特征。除了存储器读写之外，每个处理器还必须完成 cache 作废处理，这将影响有效的存储器访问时间。令 f_w 是引发作废信号

的数据访问的比例，此信号被传送到其他数据 cache。发送此信号的处理器要用 t 时钟周期完成作废操作，其他处理器不参与此次作废操作。请确定有效的存储器访问时间。

17.9 图 17-24a～图 17-24d 分别暗示的是什么组织方案？

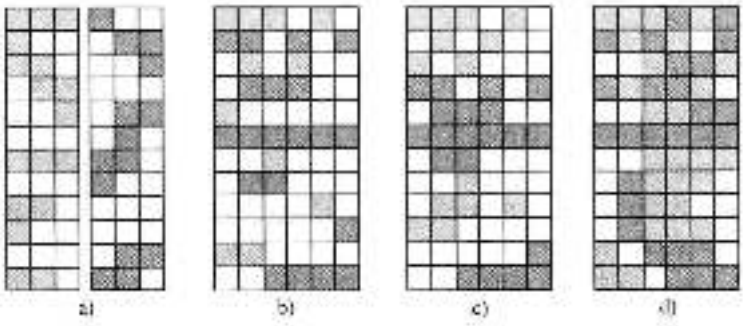


图 17-24 习题 17.9 的图

17.10 在图 17-8 中，某些图表示出水平行部分填充，另一种情况是整行空白。请说明这代表哪两种效率损失。

17.11 考虑图 12-13b 所描述的流水。现在不管指令取指和译码两阶段，将它改画成图 17-25a 来代表线程 A 的执行。图 17-24b 代表另一线程 B 的执行。两种情况处理器使用的都是简单流水。

(a) 为每个线程画出类似图 17-8a 的指令发射图。

(b) 假定在片上多处理器上并行执行这两个线程，芯片上两个处理器核每个都使用简单流水。请画出类似图 17-8k 的指令发射图，并再画出有图17-25风格的流水执行图。

(c) 假定有一个双发射超标量体系结构为交错式多线程超标量的实现。重复（b）问题，假定无数据相关性。注意：这里没有唯一的答案，你需对延迟和优先级进行假设。

(d) 对于阻塞式多线程超标量的实现，重复（c）问题。

(e) 对于 4 发射 SMT 体系结构，重复（c）问题。

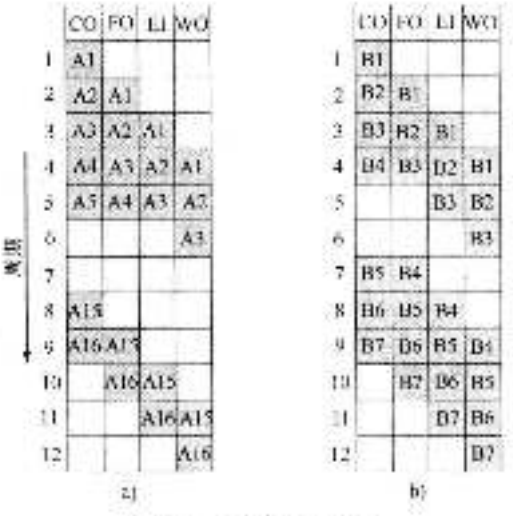


图 17-25 执行 2 个线程

17.12 为计算向量算术表达式：$D(I) = A(I) + B(I) \times C(I)$，$0 \leq I \leq 63$

如下代码段需要执行 64 次：

```
Load R1, B(I)          ;R1←Memory(α + I);
Load R2, C(I)          ;R2←Memory(β + I);
Multiply R1, R2        ;R1←(R1) × (R2);
Load R3, A(I)          ;R3←Memory(γ + I);
Add R3, R1             ;R3←(R3) + (R1);
Load D(I), D(I)        ;Memory(θ + I)←(R3);
```

这里 R1、R2 和 R3 是处理器的寄存器，α、β、γ、θ 分别是数组 B(I)、C(I)、A(I) 和 D(I) 的主存起始地址。假定每个 Load 或 Store 用 4 个时钟周期，Add 用 2 个时钟周期，Multiply 用 8 个时钟周期，这些指单处理器或在 SIMD 机器单个处理器上执行的情况。

(a) 请计算：这个代码段在一个 SISD 处理器上重复执行 64 次所需的总的处理器时钟周期数，不考虑其他所有延时。

(b) 考虑使用一台有 64 个处理器的 SIMD 机器，这 6 条指令经过向量化后能同时对 64 个分量的向量数据进行操作，处理单元和数据部件是按同一频率时钟驱动的。请计算在此 SIMD 机器上的总的执行时间，不考虑指令广播和其他延迟。

(c) SIMD 计算机与 SISD 计算机相比，其加速比是多少？

17.13 为如下程序制作一个向量化版本：

```
      DO 20 I = 1, N
      B(I, 1) = 0
      DO 10 J = 1, N
      A(I) = A(I) + B(I, J) * C(I, J)
10    CONTINUE
      D(I) = E(I) + A(I)
20    CONTINUE
```

17.14 一个应用程序在 9 台计算机的集群上执行，花费了 T 时间。测试进一步表明，T 的 25% 时间是此应用程序在 9 台计算机上同时运行，其余时间应用程序仅在单一计算机上运行。

(a) 请计算，上述情况比之程序在单计算机上运行所获得的有效加速比。再计算上述程序中并行化代码（编程或编译时使用了集群模式）所占的百分比 α。

(b) 假设我们能有效地使用 18 台而不是 9 台计算机来运行并行化代码部分，请计算所实现的有效加速比。

17.15 在一计算机上执行的 FORTRAN 程序如下，其并行化版本将在一个由 32 台计算机组成的集群系统上运行。

```
1:    DO 10 I = 1, 1024
2:        SUM(I) = 0
3:        DO 20 J = 1, I
4:20          SUM(I) = SUM(I) + I
5:10    CONTINUE
```

假设行 2 和行 4 每个都用 2 个机器周期，包括所有的处理器和存储器访问动作。本计算机循环控制语句（行 1，3，5）引起的开销和其他所有系统开销以及资源冲突。试问：

(a) 程序在单个计算机上总的执行时间是多少（以机器周期为单位）？

(b) 将 I 循环的迭代以如下方式在 32 个计算机中分摊：计算机 1 执行最初的 32 次迭代（I = 1 到 32），计算机 2 执行下一个 32 次迭代，如此继续分摊下去。总的执行时间是多少？与 (a) 比较，速度提高了多少？（注意，J 循环支配的计算负载在各计算机中未做均衡）

(c) 说明应如何修改上述并行化方式，以使计算负载在 32 个计算机上分布均衡地并行执行。负载均衡意味着对 I、J 两个循环都要将相等数量的加法运算指派到各计算机上。

(d) 32 台计算机并行执行所需要的最小执行时间是多少？与单个计算机相比，速度提高多少？

17.16 考虑两向量相加的如下两个程序形式：

```
1:    DO 10 I = 1, N
2:      A(I) = B(I) + C(I)
3:10  CONTINUE
4:    SUM = 0
5:    DO 20 J = 1, N
6:      SUM = SUM + A(J)
7:20  CONTINUE
```

```
      DOALL K = 1, X
        DO 10 I = L(K - 1) + 1, KL
          A(I) = B(I) + C(I)
10    CONTINUE
      SUM(K) = 0
      DO 20 J = 1, M
        SUM(K) = SUM(K) + A(L(K - 1) + J)
20    CONTINUE
      ENDALL
```

(a) 左边的程序在单一处理器上执行。假设执行 12、14 和 16 每行用 1 个处理器时钟周期。为简化，不考虑其他代码行所要求的时间。最初，所有数组已装入主存就绪，并且此短程序段已在指令 cache 中。执行这个程序要用多少时钟周期？

(b) 右边的程序打算在一个有 M 个处理器的多处理器上执行。我们将循环操作部分划分成 M 个段，每段 $L = N/M$ 个元素。DOALL 声明，所有 M 个段并行执行。执行的结果是产生 M 个部分和。假定，经由共享存储器的处理器之间的通信操作每次都需要 k 个时钟周期，这样每次部分和的加法要求 k 个时钟周期。一个 l 级二进制加法器树能合并所有的部分和，这里 $l = \log_2 M$。产生最终结果需要多少周期？

(c) 假定数组有 $N = 2^{20}$ 个元素，并且 $M = 256$。使用多处理器能实现的加速比是多少？假定 $k = 200$。这个加速比是因子为 256 的理想加速比的百分之几？

多核计算机

本章要点

- 多核计算机或者多核处理器，在一个计算机芯片上有两个或者多个处理器。
- 由于硬件性能问题，使用更多单个处理器芯片已达到一个极限，包括指令层并行和功耗问题。
- 另一方面，多核体系结构给软件开发者探索多核上的多线程能力提出挑战。
- 多核结构的主要变量是芯片处理器的数目、cache 存储器级数、cache 共享的程度。
- 多核系统中另一个结构上的设计决策是单个内核是超标量或者实现并发多线程（SMT）。

多核计算机也称为**单芯片多处理器**，指在一个单独的硅片上结合两个或者多个处理器（称为核）。典型地，每个核由一个独立处理器的所有组件构成，如寄存器组、ALU，流水线硬件以及控制单元，加上 L1 指令和数据 cache。除了多个核，当代多核芯片也包括 L2 cache，并在一些例子中，还有 L3 cache。

这一章对多核系统进行综述。首先考察多核计算机发展的硬件性能因素，探索多核系统功耗方面的软件挑战。接下来，考察多核组织结构。最后，研究两个多核产品实例，即 Intel 和 ARM。

18.1 硬件性能问题

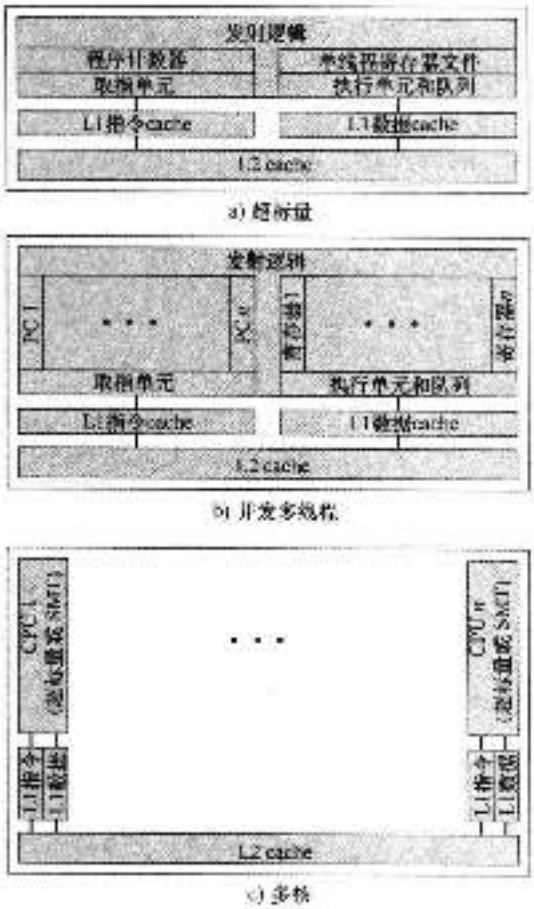
正如我们在第 2 章所讨论的，微处理器系统在执行性能上已经历了十多年稳定的指数增长。图 2-12 显示了这个增长部分是由于芯片处理器结构上的精炼，另一部分是由于时钟频率的提高。

18.1.1 增加并行

处理器设计结构的改变已主要聚焦在增加指令级并行上，从而更多工作能在每个时钟周期完成。按照时间排序这些改变，包括（参见图 18-1）：

- **流水线**：各条指令通过多段流水线被执行，因而，在一条指令执行时，另一条指令在流水线的另一个阶段同时被执行。
- **超标量**：通过复制执行资源来构造多个流水线，只要不出现冲突，指令就能够在多个流水线上并发地执行。
- **并发多线程（SMT）**：寄存器体（register bank）被复制，从而多线程能共享流水线资源。

对于以上这些创新，设计者多年来试图通过



a) 超标量

b) 并发多线程

c) 多核

图 18-1 可供选择的芯片组织

增加复杂度来提高系统性能。在流水线例子中，简单的二段流水被五段甚至更多段流水替代，有些实现了 12 个以上流水段。实际上这个走势能扩展多远是有限制的，因为如果拥有较多流水段，则需要更多逻辑、内部连接和控制信号。如果拥有超标量结构，性能提高能通过增加并行流水线的数目获得。随着流水线的数量增加，这里同样存在性能反而下降的问题，需要更多逻辑来管理冲突以及分段指令资源。最终，当一个单线程运行到某点时，资源冲突和相关性将阻止系统充分利用所有可获得的流水线资源。同样，拥有 SMT 的性能遵循也会达到这一点，因为一套流水线上多个线程管理的复杂度限制了线程数量以及能够被有效利用的流水线数量。

在图 18-2 OLUK05 中，上图显示近年来 Intel 处理器性能的指数增长。中图是结合 Intel 发布 SPEC CPU 数据和处理器时钟频率计算得出的，以给出一个提高指令级并行开发获得性能提高程度的测评。在广泛开发并行之前，20 世纪 80 年代后期曲线存在一个平缓区。接下来，随着设计者能够不断开发流水线、超标量技术以及 SMT，曲线出现了陡然升高。但是，从 2000 年开始，随着有效指令级并行开发达到一个极限，曲线出现了一个新的平缓区。

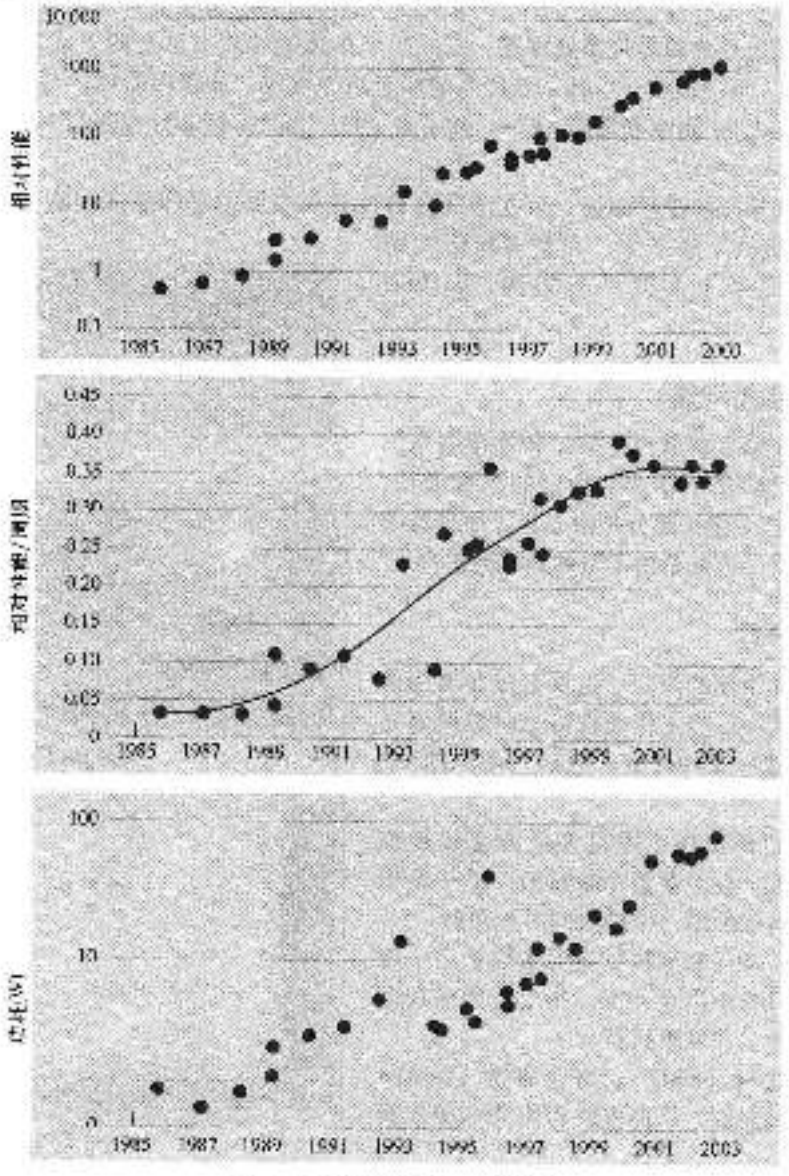


图 18-2 Intel 硬件的一些趋势

18.1.2　功耗

随着芯片上晶体管数目的增加，为了保持更高性能的发展趋势，设计者采取更精细处理器设计（流水线、超标量、SMT）以及高时钟频率。不幸的是，在芯片密度和时钟频率增加的同时，对功耗的需求呈指数增长。这一点体现在图 18-2 最下面。

控制功耗密度的一种方法是为 cache 存储器使用更多的芯片面积。内存晶体管较小且其功耗密度的数量级低于逻辑的（见图 18-3a）。如图 18-3b 所示［BORK03］，当芯片晶体管密度增加时，内存所占芯片面积的百分比已上升超过 50%。

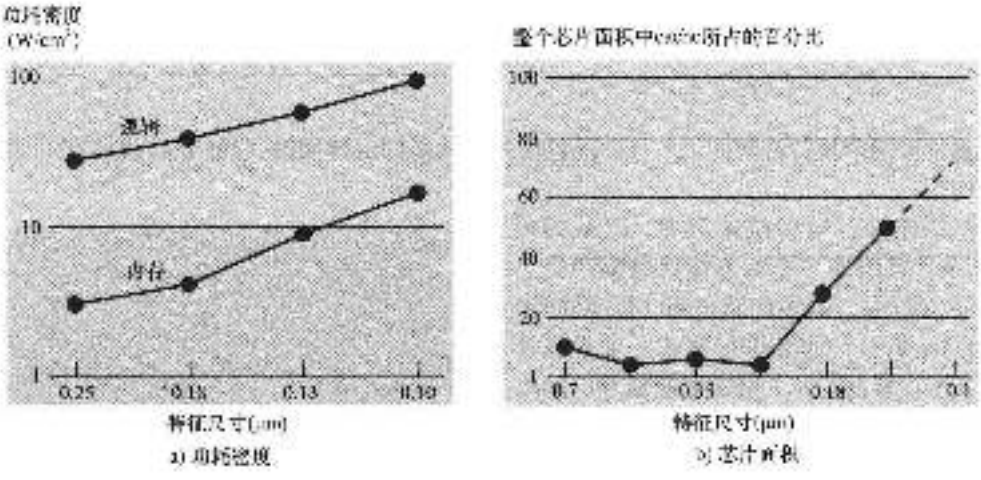


图 18-3　功耗和存储器的关系

图 18-4 显示了功耗趋势的方向［BORK07］。到 2015 年，我们希望能看到在一个 300mm² 的裸片上有大约 1000 亿晶体管的微处理器芯片。假设内存占芯片面积 50% – 60% 左右，那么芯片将支持大约 100MB cache 存储器，给逻辑部分留下 10 亿晶体管。

如何利用这些逻辑晶体管是设计的一个关键问题。如前面讨论那样，有效地使用一些方法（如超标量和 SMT）是有限制的。总的来说，近十年的经验已精练为 Pollack 规则［POLL99］的经验法则，它声称性能增长与复杂度增加的平方根严格地成比例。换句话说，如果增加两倍处理器核中的逻辑部分，那么它的性能仅增加 40%。原则上，随着核数目增加，使用多核器在理论上提供了性能的近线性（near-linear）提高。

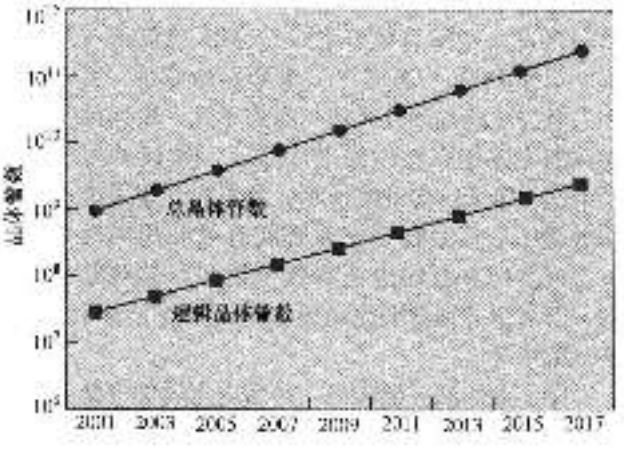


图 18-4　芯片上晶体管的使用

功耗是转向多核架构的另一个原因。因为当一块芯片拥有如此丰富的 cache 存储器时，任何一个执行的线程都不可能有效利用所有内存。即使拥有 SMT，由于你是在一个相对有限的方式中运行多线程开发，也不能全向开发一个巨大的 cache，而许多相对独立的线程或过程则有较大机会完全利用 cache 存储器。

18.2 软件性能问题

多核结构软件性能的详细考察超出了我们的范围。本节我们首先给出这些问题的一个概述，然后看一个挖掘多核能力应用设计的例子。

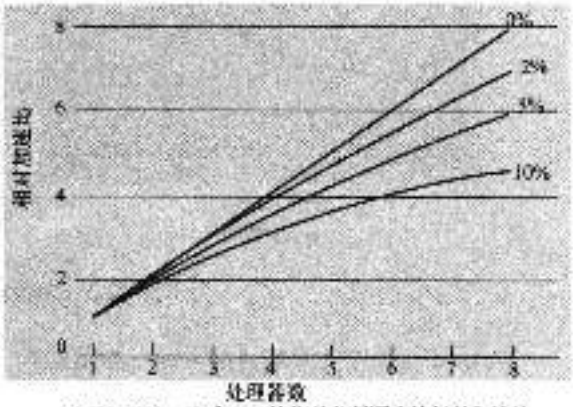
18.2.1 多核软件

多核结构的潜在性能受益于有效开发应用程序并行资源的能力。我们首先来关注一个运行在多核系统上的简单应用程序。回顾第 2 章阿姆达尔定律：

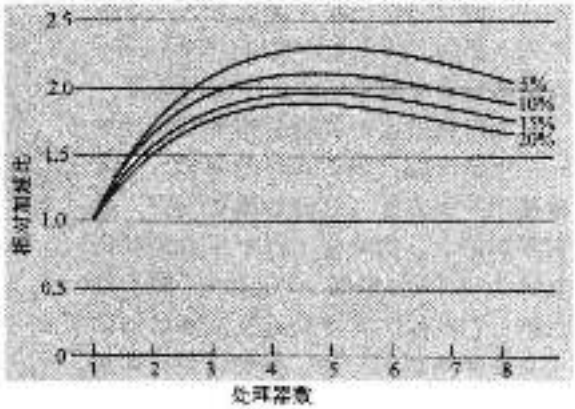
$$\text{加速比} = \frac{\text{程序在单个处理器上的执行时间}}{\text{程序在 } N \text{ 个并行处理器上的执行时间}} = \frac{1}{(1-f) + \frac{f}{N}} \tag{18.1}$$

该定律假定程序中执行时间的 $(1-f)$ 段包含的代码是固定连续的，f 段包含无调度开销的无限可并行代码。

该定律使多核结构前景可观。但是如图 18-5a 所示，甚至连一小段连续代码都产生了值得关注的影响。如果代码只有 10% 是固定连续的（$f=0.9$），那么该程序在一个 8 核处理器系统上仅能获得 4.7 倍的性能提升。除此之外，由于多处理器上通信和任务分配会导致软件开销以及 cache 一致性开销，导致曲线中性能峰值因使用多处理器的开销增加开始下降。来自 [MCD005] 的图 18-5b 是一个具有代表性的例子。



a) 0%、2%、5%和10%的代码必须顺序执行的加速比



b) 考虑软件和外开销后的加速比

图 18-5 多核对性能的影响

可是，软件工程师们还在关注这个问题，目前存在大量能有效开发一个多核系统的应用。[MCD005] 报道了一套数据库应用，其中主要关注减少硬件结构、操作系统、中间件和数据库应用软件内的串行部分。图 18-6 显示了这个结果。如此例所示，数据库管理系统和数据库应用是在多核系统能有效利用的地方。许多种服务器也能有效使用并行多核结构，因为服务器典型地就是并行处理多个相对独立的事务。

除了通用服务器软件，许多应用软件也能随处理器核数量的变化而实现性能的伸缩。[MCD006] 列出了下面的例子：

- **多线程本地应用**：多线程应用以拥有小部分高线程处理器为特征。线程应用（threaded application）的例子包括 Lotus Domino 或 Siebel CRM（客户关系管理器）。
- **多处理应用**：多处理应用以存在许多单线程处理为特征。多处理应用的例子包括 Oracle 数据库、SAP 以及 PeopleSoft。
- **Java 应用**：Java 应用包含一个基本方式运行的线程。Java 语言不但对于多线程应用十分

便捷，而且 Java 虚拟机是一个提供 Java 应用的调度和内存管理的多线程处理。直接得益于多核资源的 Java 应用包括应用服务器，如 Sun 的 Java 应用服务器、BEA 的 Weblogic、IBM 的 Websphere 以及开源的 Tomcat 应用服务器。所有使用 Java 2 平台的应用、企业版（J2EE 平台）应用服务器能快速受益于多核技术。

- **多实例应用**：即使单个应用不能伸缩以利用大量的线程，但它仍然可通过并行运行多个应用实例以获得多核结构。如果多个应用实例需要一定程度的独立，可以采用虚拟技术给每个实例提供单独和安全的环境。

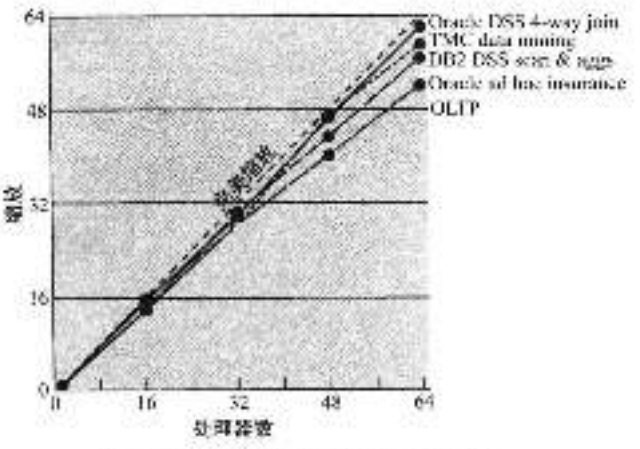


图 18-6　多处理器硬件平台上数据库工作负载的缩放

18.2.2　应用实例：Valve 游戏软件

Valve 是一家娱乐技术公司，已开发了许多大众游戏以及 Source 引擎，它是最广泛应用的游戏引擎之一。Source 是 Valve 为其游戏以及其他授权游戏开发者使用的一个动画引擎。

近年来，Valve 为利用 Intel 和 AMD 多核处理器芯片［REIM06］，使用多线程重新编写了 Source 引擎软件。修订的 Source 引擎代码为 Valve 游戏（如 Half Life 2）提供了更强大的支持。

以 Valve 视角来看，线程粒度选项定义如下［HARR06］：

- **粗粒度线程**：被称为“系统”的每个模块都会被分配到一个单独的处理器上运行。以 Source 引擎为例，这意味着：图像计算被放到某个处理器上，AI（人工智能）计算被放到另一个处理器上，动作处理则又是一个不同的处理器，如此类推。该方案非常直接。本质上，每个重要的模块都是一个单线程，而系统协调工作主要是将所有的线程与一个时间轴线程同步。
- **细粒度线程**：许多类似或者同样的任务调布在多个处理器上。例如，在一个数组上迭代的循环可以分成许多小的在单个线程内并行调度的并行循环。
- **混合线程**：这包括为某些系统选取细粒度线程，以及为另一些系统选取单线程。

Valve 发现通过粗粒度线程，在两个处理器上能获得与在一个单处理器上执行相比两倍的性能，但是这仅是理想情况。对于真实世界游戏，性能提高大约为 1.2 倍。Valve 也发现有效利用细粒度线程十分困难。每个工作单元的时间是变化的，并且维护各种结果及其后续事件在时间轴上的位置也需要复杂的编程。

Valve 发现混合线程的方法最有前景并且在多核处理器上具有最好的性能伸缩性，特别是对于即将出现的 8 核或 16 核处理器而言。Valve 标记出那些被固定分配到某个处理器时才能高效运行的系统。一个例子是混音系统，它几乎没有用户交互，不受窗口帧配置的限制，工作在它自己的数据集上。其他模块（如场景绘制）可以组织为许多线程，从而模块能在一个处理器上执行，并且随着调布到越来越多的处理器上而获得更高性能。

图 18-7 显示了绘制模块的线程结构。在这个层级结构，根据需要高层线程产生底层线程。该绘制模块依赖 Source 引擎的关键部分——世界列表（world list），它是游戏世界中可视化元素的一个数据库表示。第一项任务是确定需要绘制的世界区域是什么。下一步工作是确定从多个角度看到场景中的对象是什么。接下来是加速处理器工作。该绘制模块必须完成每个对象从多

个视角（如玩家角度、TV 监视器角度、水中反射角度）的视图。

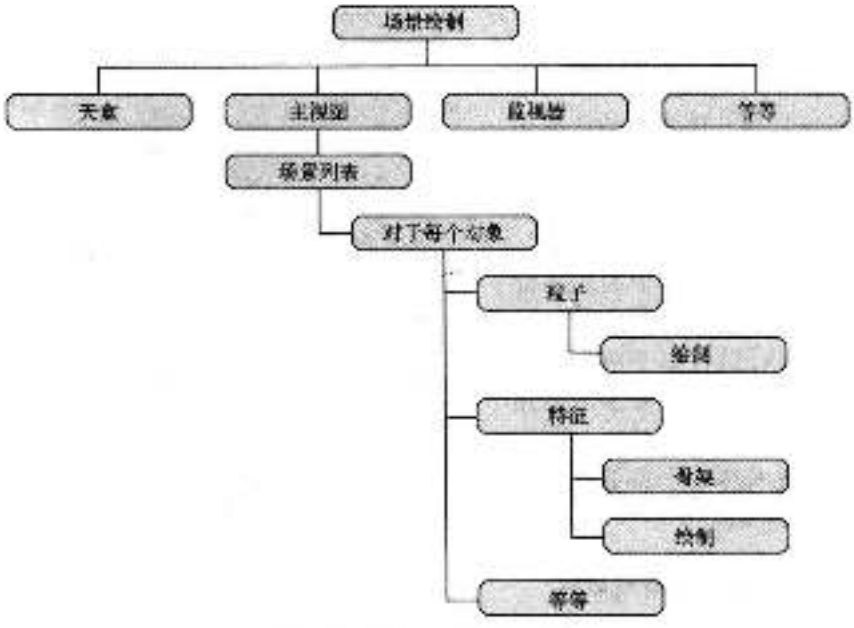


图 18-7 绘制模块的混合线程

绘制模块的线程策略的一些关键元素如 [LENON07] 中所列，包括：

- 为多场景并行构造场景绘制列表。
- 交叠图形模拟。
- 并行计算全部场景中所有角色的骨架变换。
- 并行多线程绘制。

设计者发现简单锁定关键数据库，如世界列表，对于线程太低效。在超过 95% 的时间内，线程试图从数据集中读数据，而最多 5% 的时间用来写入数据集。这样一来，可以有效利用目前一个称为单写者多读者的模块机制。

18.3 多核组织结构

在开始描述顶层结构之前，多核结构中的主要变量如下：

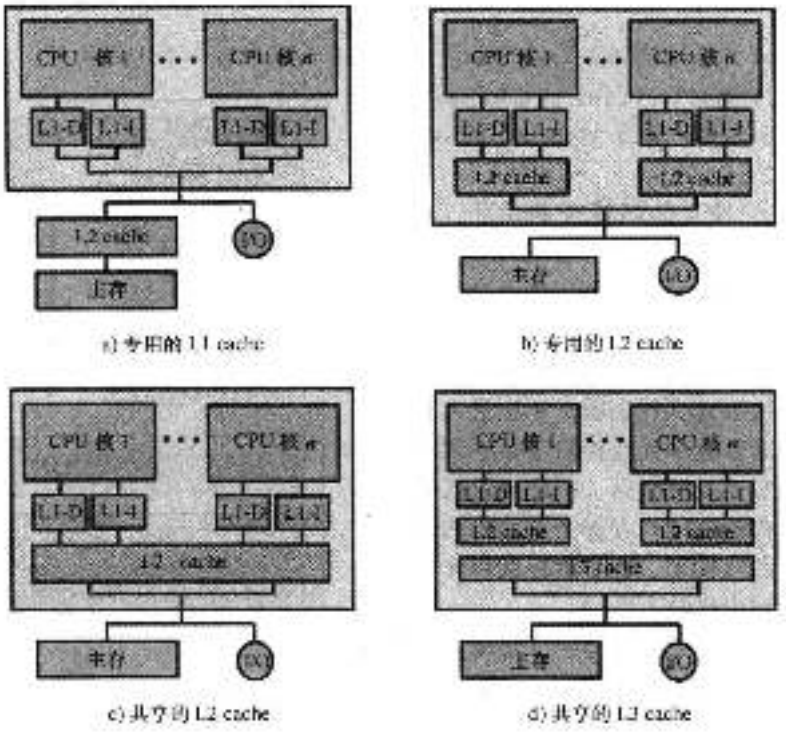
- 芯片上核处理器的数目。
- cache 存储器的级数。
- 共享 cache 存储器的数目。

图 18-8 显示了多核系统的四种常见结构。图 18-8a 出现在早期多核计算机芯片结构中，现在在嵌入式芯片中仍可见。该结构唯一的片上 cache 是 L1 cache，每个核拥有自己专门的 L1 cache。L1 cache 几乎固定被划分为指令 cache 和数据 cache。该结构的一个实例就是 ARM11 MPCore。

图 18-8b 的结构也是一个没有片上 cache 共享的。其中芯片上有足够的面积来支持 L2 cache。该结构的一个实例是 AMD Opteron。图 18-8c 显示了一个类似的芯片空间对存储器的分配，但使用了一个共享的 L2 cache。Intel Core Duo 就是这种结构。最后，随着芯片上可利用的 cache 存储器资源的增多，出于性能考虑，可以设置一个 L3 cache，并由多个核共享，同时每个处理器核仍拥有一个专用的 L1 cache 和 L2 cache。Intel Core i7 就是这种结构。

芯片上使用一个共享的 L2 cache 比独靠一个专门 cache 有许多优势：

(1) 结构干涉能够减少整体失效率。这是因为：当某个核上的线程访问主存某个地址之后，



a) 专用的 L1 cache　　b) 专用的 L2 cache

c) 共享的 L2 cache　　d) 共享的 L3 cache

图 18-8　几种多核组织结构

该地址所在的存储块就被调入共享 cache 中，若随后另一个核上的线程要访问同一个存储块，则可以直接从共享的片上 cache 中读取。

（2）一个相关的优势是被多核共享的数据在共享的 cache 级上不会被复制。

（3）利用合适的替换算法，分配给每个核的共享 cache 数目是动态的，因此，那些存储器访问局部性不强的线程能够占用更多的 cache 空间。

（4）通过共享存储器空间，容易实现处理器内部通信。

（5）使用一个共享的 L2 cache 使 cache 一致性问题限制在 L1 cache 级，从而提供一些额外的性能优势。

芯片上拥有专用 L2 cache 的一个潜在优势是每个核享有对其私有 L2 cache 更快速的访问。这是线程展示强大局部性的优势所在。

随着可获得的存储器数和核数的增加，使用一个共享 L3 cache，附加一个共享 L2 cache 或每个核专用的 L2 cache 似乎比简单共享一个大块 L2 cache 能提供更好的性能。

多核系统中另一个结构设计决策是单个核能否超标量或实现并发多线程（SMT）。例如，Intel Core Duo 采用超标量核，而 Intel Core i7 使用 SMT 核。SMT 能按比例提高多核系统支持的硬件层上线程的数目。如此一来，一个拥有四核和 SMT 的多核系统，其中每个核支持四个并发线程，在应用层看来与一个拥有 16 个核的多核系统相同。随着软件开发对并行资源更全面的利用，SMT 方法比超标量方法更有吸引力。

18.4　Intel x86 多核结构

Intel 近来推出了许多多核产品。本节我们主要来看两个例子：Intel Core Duo 和 Intel Core i7。

18.4.1　Intel Core Duo

Intel Core Duo 在 2006 年推出，实现了两个 x86 超标量处理器，以及一个共享 L2 cache（见

图 18-8c）。

Intel Core Duo 总体结构如图 18-9 所示。我们从上至下来看一下主要组件。正如多核系统中常见的一样，每个核有它自己的专门 L1 cache。本例中，每个核有 32KB 指令 cache 和 32KB 数据 cache。

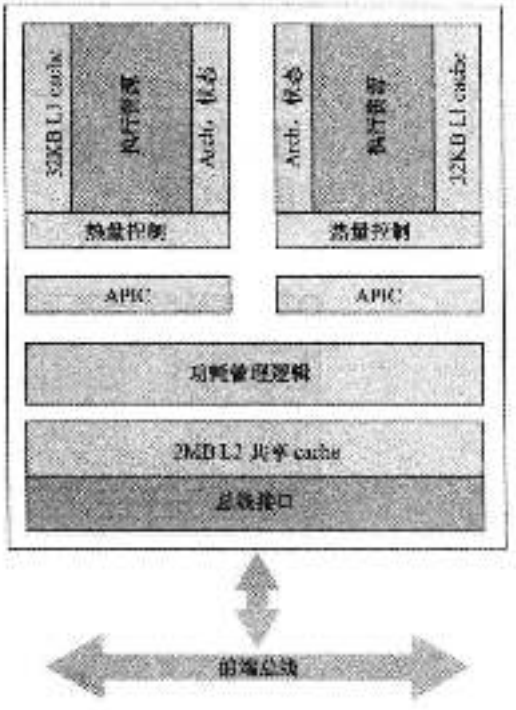


图 18-9 Intel Core Duo 总体结构

每个核有一个独立的**热量控制单元**（thermal control unit）。由于当今芯片晶体管的高密度，热量管理是基本能力，特别是对于笔记本电脑和移动系统。Core Duo 热量控制单元用来在热量界限下管理芯片散热以获取最大处理器性能。同时，热量管理通过一个较冷系统和较低风扇噪声来改进人类工程学。本质上，热量管理单元监控数字传感器。每个核被定义在一个独立的热量区。每个热量区的最大温度通过软件选择的专门寄存器来单独报告。如果一个核内的温度超过阈值，则热量控制单元减小该核的时钟频率以减少热的产生。

Core Duo 结构的下一个关键组件是**高级可编程中断控制器**（advanced programmable interrupt controller，APIC）。APIC 可完成很多功能，包括：

（1）APIC 支持处理器间的中断，使得某个处理器上的进程能够中断其他（单个或一组）处理器的执行。对于 Core Duo 而言，某个处理器核上运行的线程产生的中断将首先被该核的本地 APIC 所接受，然后传递到其他核上的 APIC，就好像该中断是在其他处理器核上发生一样。

（2）APIC 接受 I/O 中断，发送这些消息给合适的核。

（3）每个 APIC 含有一个定时器，它能通过 OS 设置以产生一个中断给本地核。

功耗管理逻辑（power management logic）负责尽可能降低功耗，从而增加移动平台（如笔记本电脑）的电池寿命。本质上，功耗管理逻辑监视热量条件以及 CPU 活动，并适当调节电压和功耗。它包括一个高级的功耗门控能力，允许一个过细粒度的逻辑控制。仅当需要时，该逻辑控制打开单个处理器的逻辑子系统。而且，很多总线和数组被分离，从而某些操作模式下所需数据能在不需要时处于一个低功耗状态。

Core Duo 芯片包括一个共享的 2MB L2 cache。此 cache 逻辑允许根据当前核所需的 cache 空间动态分配，因此一个核能被分配达到 100% 的 L2 cache。该 L2 cache 包含支持其附带 L1 cache 的 MESI 协议的逻辑。需考虑的一个关键点是在 L1 cache 上 cache 写入何时完成。当一个处理器写入时，cache 行得到了 M 状态。如果这行在写之前没在 E 或 M 状态，cache 发送一个 RFO（Read-For-Ownership）需求，以确保该行在此 L1 cache 中且在其他 cache 中处于 I 状态。Intel Core Duo 扩展了这个协议，以考虑出现多个 Intel Core Duo 芯片被组织为一个对称多处理器（SMP）系统的情况。L2 cache 使得系统区分两种情况，一种情况是，数据被两个本地核共享，而与处理器系统的其他部分无关；另一种情况是，数据被模片上的一个或多个 cache 以及一个外部总线代理（可以是另一个处理器）共享。当一个核发出一个 RFO 时，如果 cache 行仅被本地裸片上另一个 cache 共享，可以在内部迅速解决该 RFO，根本不通过外部总线。仅当此行被外部总线上的另一个代理共享时，才需要外部发出 RFO。

总线接口连接外部总线，称为 FSB（front side bus），它连接主存、I/O 控制器和其他处理器

芯片。

18.4.2　Intel Core i7

Intel Core i7 在 2008 年推出，实现了 4 个 x86 SMT 处理器，每个处理器包括一个专用 L2 cache 以及一个共享的 L3 cache（见图 18-8d）。

Intel Core i7 的总体结构如图 18-10 所示。每个核拥有它自己专用的 L2 cache，并且四个核共享一个 8MB L3 cache。Intel 用来使其 cache 更高效的一个机制是预取，其中硬件检查内存访问模式，试图用不久可能需要的数据去填充 cache。拿芯片上这个三层 cache 结构的性能与 Intel 的两层组织结构进行比较是颇有意思的。表 18-1 显示了运行在同一时钟频率下的两个 Intel 多核系统，就其时钟频率而言各自的 cache 访问延时。Core 2 Quad 拥有一个共享的 L2 cache，类似于 Core Duo。Core i7 通过采用专用的 L2 cache 以及提供对 L3 cache 相对高速的访问，使其在 L2 cache 性能上有了提高。

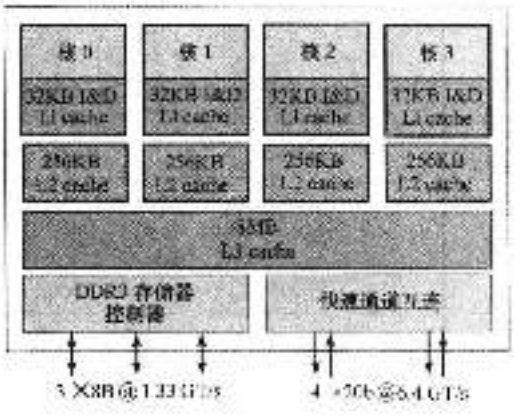


图 18-10　Intel Core i7 模块图

表 18-1　cache 延迟（时钟周期数）

CPU	时钟频率	L1 cache	L2 cache	L3 cache
Core 2 Quad	2.66 GHz	3 个时钟周期	15 个时钟周期	—
Core i7	2.66 GHz	4 个时钟周期	11 个时钟周期	39 个时钟周期

Core i7 支持两种与其他芯片进行外部通信的形式。DDR3 存储器控制器将 DDR 主存控制器放在芯片上。该接口支持三个 8 字节的通道，因此总的总线宽度为 192 位，总数据速率达到 32GB/s。芯片上由于有存储器控制器，因此无需 FSB。

快速通道互连（quickpath interconnect，QPI）是一种基于 Intel 处理器和芯片组电气互连规范，具备 cache 一致性的点到点通信链路。它使得在连接的芯片之间进行高速通信。QPI 链路工作于6.4GT/s（每秒传输量）。每传送 16 位，增加到 12.8GB/s。由于 QPI 链路是一对双向的，所以总的带宽达到 25.6GB/s。

18.5　ARM11 MPCore

ARM11 MPCore 是 ARM11 处理器家族的一款多核产品。ARM11 MPCore 可以在每个芯片上配置多达 4 个处理器，每个处理器拥有自己的 L1 指令和数据 cache。表 18-1 列出了系统包括默认值的配置选项。

图 18-11 给出了 ARM11 MPCore 的一个结构图。该系统的主要元素如下：

- 分布式中断控制器（DIC）：处理中断检测和中断优先级。DIC 将中断分配给各个处理器。
- 定时器：每个 CPU 都有其自己私有的能产生中断的定时器。
- 看门狗：软件出错时发出警报。如果启用看门狗，则设置一个预定的值，然后计数器减直到 0。它被周期性重置。如果看门狗的值达到 0，则发出警报。
- CPU 接口：处理中断响应、中断屏蔽（masking）以及中断竞争响应。
- CPU：一个 ARM11 处理器。各个 CPU 都作为一个 MPU CPU。
- 向量浮点（VFP）单元：在硬件上实现浮点运算的协处理器。
- L1 cache：每个 CPU 都拥有自己专用的 L1 数据 cache 和指令 cache。

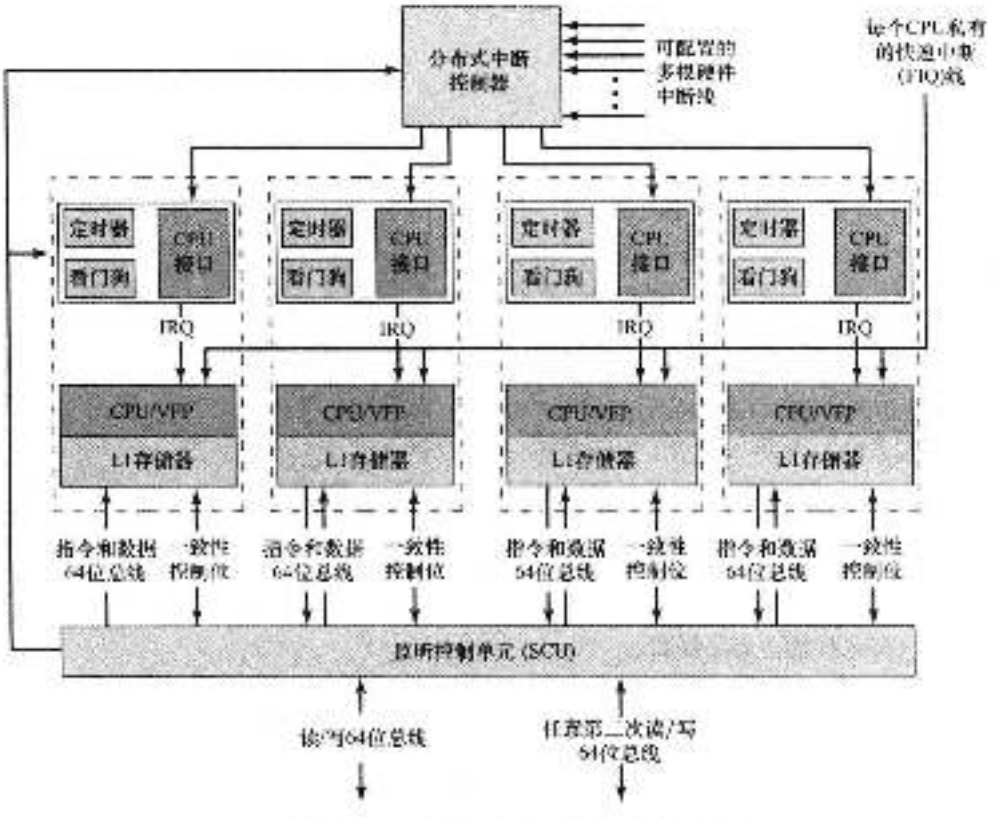


图 18-11 ARM11 MPCore 处理器的模块图

- 监听控制单元（SCU）：负责保持 L1 数据 cache 的一致性。

18.5.1 中断处理

DIC 比较来自大量资源中的中断。它提供以下功能：

- 屏蔽中断
- 区分中断优先级
- 分配中断给目标 MP11 CPU
- 跟踪中断状态
- 通过软件产生中断

DIC 是一个单独功能单元，放置在系统 MP11 CPU 旁。这使得系统中所支持的中断数目独立于 MP11 CPU。DIC 是内存映射，即 DIC 的控制寄存器被定义成与主存基址相关。MP11 CPU 通过 SCU 使用一个私有接口访问 DIC。

DIC 设计用来满足两个功能需求：

- 提供发送中断需求给单个 CPU 或多个 CPU 的一种方法。
- 提供一种处理器间通信的方法，使一个 CPU 上的线程能被另一个 CPU 的线程激活。

作为同时利用上述两个要求的例子，让我们考虑一个在多处理器上运行的多线程应用程序。假定该应用程序已经分配了一些虚拟内存。为了维护存储一致性，操作系统必须更新所有处理器上的内存映射表。操作系统首先将更新那个实际分配虚拟内存的处理器上的映射表，然后向运行该应用程序的其他处理器发出一个中断。其他处理器将利用该中断的编号来确定它们是否需要更新内存映射表。

DIC 能以如下三种方法来发送中断给一个或更多 CPU：

- 中断仅能被直接发送给一个特定处理器。
- 中断被直接发送给一个定义的处理器组。MPCore 把第一个接收该中断的处理器视为处理该中断的最佳位置，其中典型的就是将最近加载的（处理器）作为中断接收处理器。
- 中断能被直接发送给所有处理器。

从运行在一个特定 CPU 上的软件的角度来看，OS 能产生中断给所有或特定的其他 CPU。运行在不同 CPU 上的线程通信，中断机制为消息传递采用了共享内存。因此，当一个线程被一个处理器间通信中断给打断时，它会读取合适的共享内存块，以从触发该中断的线程中找回消息。

从 MP11 CPU 的角度来看，中断可以是：

- **非活动**：非活动中断是未声明的，或在多处理环境已完全被 CPU 处理，但仍然在某些 CPU 内等待或活动，因而在中断资源上可能未被清除的中断。
- **就绪**：就绪中断是已声明，且在一 CPU 上没有开始处理的中断。
- **活动**：活动中断是在 CPU 上已开始处理，但未处理完的中断。当一个新的高优先级中断打断 MP11 CPU 中断处理时，它能抢占一个活动中断。

中断来源如下：

- **处理器间的中断（IPI）**：每个 CPU 都有私有中断（如 ID0 - ID15），它们仅能被软件触发。一个 IPI 的优先级由接收的 CPU 而不是发送的 CPU 确定。
- **私有定时器和（或）看门狗中断**：这些使用中断 ID29 和 30。
- **继承 FIQ 行**：在继承 IRQ 方式中，继承的 FIQ 引脚，在每个 CPU 基础上，绕过中断分配逻辑，并直接驱动中断需求给 CPU。
- **硬件中断**：硬件中断被相关中断输入行的可编程事件触发。CPU 最大能支持 224 个中断输入行。硬件中断从 ID32 开始。

图 18-12 是 DIC 的一个结构图。DIC 可配置为支持 0 - 255 个硬件中断输入。DIC 维护一张中断表，上面显示了中断的优先级和状态。中断分配器发送给每个 CPU 接口最高优先级就绪中断，它接收中断响应的反馈信息，然后改变相应中断的状态。CPU 接口还发送 EOI（end of interrupt information），它使中断分配器将该中断的状态从活动更新为非活动。

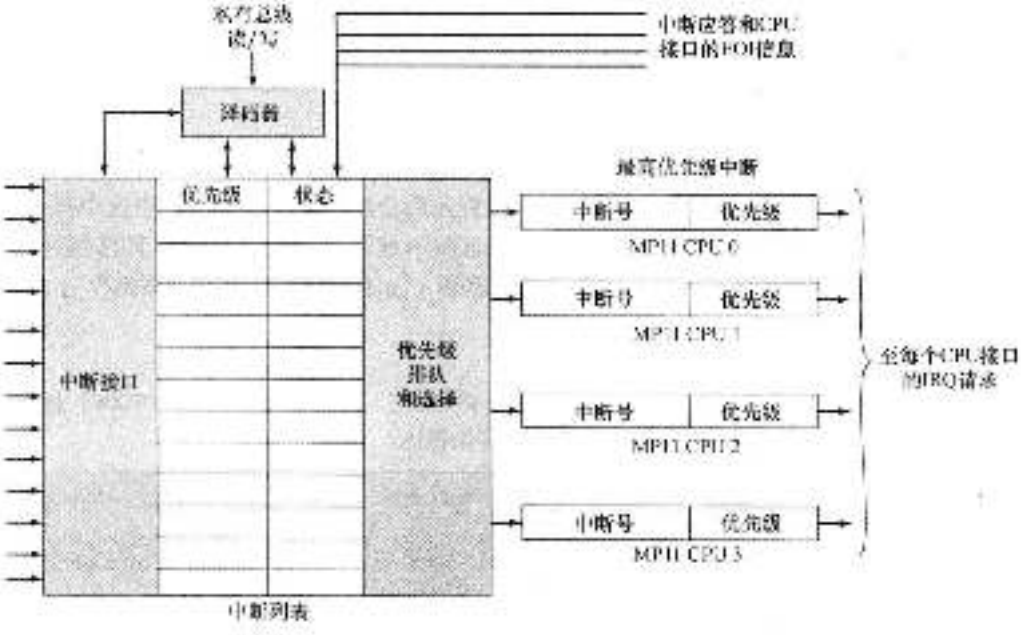


图 18-12　中断分配器的模块图

表 18-2 ARM11 MPCore 配置选项

特征	选择范围	默认值
处理器	1～4	4
每个处理器的指令 cache 大小	16KB、32KB 或 64KB	32KB
每个处理器的数据 cache 大小	16KB、32KB 或 64KB	32KB
主端口	1 或 2	2
中断总线宽度	0～224，以 32 递增	32 个引脚
每个处理器的向量浮点协处理器	包含或不包含	包含

18.5.2 cache 一致性

MPCore 监听控制单元（SCU）设计用来解决许多与共享数据访问和一致性阻塞带来的可伸缩性限制相关的传统瓶颈问题。

L1 cache 一致性方案是基于 MESI 协议的，这在第 17 章已阐述过。SCU 监视器操作共享数据以优化 MESI 状态迁移。SCU 给出了三种优化：直接数据插入、标签 RAM 副本以及迁移行。

直接数据插入（DDI）可不访问外部存储器，就从 CPU 的 L1 cache 复制清除的数据到另一 CPU 的 L1 cache。这样减少了从 L1 cache 到 L2 cache 的读行为。于是，一个本地 L1 cache 的失效被一个远程 L1 cache 解决，而不是通过访问共享 L2 cache 得到解决。

回顾 cache 内每行的主存位置是被该行一个标签所定义。这个标签可以作为一个与 cache 中行数同样长度的独立 RAM 块来实现。在 SCU 中，标签 RAM 副本是 SCU 使用的 L1 标签 RAM 的副本，用来在发送一致性命令给相关 CPU 之前检查数据的可获得性。一致性命令仅发送给需更新数据 cache 一致性的 CPU。这样减少了功耗，降低了在每个内存更新上监听和操作每个处理器 cache 的性能影响。本地拥有可获得的标签数据使得 SCU 限制了对拥有公共 cache 行的处理器的 cache 操作。

迁移行的特点是允许脏数据从一个 CPU 移到另一个，而不用写入 L2 或从外部存储器读回数据。此操作可描述如下。在一个典型的 MESI 协议中，一个处理器有一个修改的行，另一个处理器试图读此行，将发生下面的行为：

（1）该行内容被从修改的行传送到初始化读的处理器。

（2）该行内容被读回到主存。

（3）该行在两个 cache 中处于共享状态。

MPCore SCU 实现着处理这种情况。SCU 监视系统的一个迁移行。如果一处理器有一个修改行，且另一个处理器读然后写它，SCU 假定这个位置此后会经历相同的操作。当这个操作又开始时，SCU 将自动把该 cache 行直接移到一个无效状态而不是浪费能量去首先将其移到共享状态。这个优化也使得处理器直接传送 cache 行给其他处理器，而不用插入外部存储器操作。

18.6 推荐的读物和 Web 站点

两本很好涉及本章讨论问题的书是 [OLUK07] 和 [JERR05]。[GOCH06] 和 [MEND06] 描述了 Intel Core Duo。[FOG08b] 提供了对 Core Duo 流水线结构的详细描述。

ARM08b ARM Limited. *ARM11 MPCore Processor Technical Reference Manual.* ARM DDI 0360E, 2008. www.arm.com

FOG08b Fog, A. *The Microarchitecture of Intel and AMD CPUs.* Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

GOCH06 Gochman, S., et al. "Introduction to Intel Core Duo Processor Architecture." *Intel Technology Journal*, May 2006.

GOOD05　Goodacre, J., and Sloss, A. "Parallelism and the ARM Instruction Set Architecture." *Computer*, July 2005.

HIRA07　Hirata, K., and Goodacre, J. "ARM MPCore: The Streamlined and Scalable ARM11 processor core." *Proceedings, 2007 Conference on Asia South Pacific Design Automation*, 2007.

JERR05　Jerraya, A., and Wolf, W., eds. *Multiprocessor Systems-on-Chips*. San Francisco: Morgan Kaufmann, 2005.

MEND06　Mendelson, A., et al. "CMP Implementation in Systems Based on the Intel Core Duo Processor." *Intel Technology Journal*, May 2006.

OLUK07　Olukotun, K.; Hammond, L.; and Laudon, J. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. San Rafael, CA: Morgan & Claypool, 2007.



推荐的 Web 站点

- **多核协会**（Multicore Association）：厂商组织促进了多核技术的发展和使用。

18.7　关键词、思考题和习题

关键词

Amdahl's law：阿姆达尔定律
chip multiprocessor：芯片多处理器
multicore：多核
simultaneous multithreading（SMT）：并发多线程
superscalar：超标量

思考题

18.1　总结简单指令流水线、超标量和并发多线程的区别。
18.2　给出设计者选择转向多核结构，而不在一个单处理器内增加并行性的理由。
18.3　为什么现在有一种将越来越多的芯片面积分配给高速缓存的趋势？
18.4　列出几个能够根据核数目伸缩并直接受益的例子。
18.5　在顶层上，一个多核结构中主要的设计变量是什么？
18.6　列出多核之间共享 L2 cache 与每个核拥有专门 L2 cache 相比的一些优势。

习题

18.1　考虑这样一个问题：假定设计者有一个芯片并且需要决定用该芯片上的多少资源来实现高速缓存（L1，L2，L3）。芯片的剩余部分可用来实现一个复杂的超标量和（或）一个 SMT 处理器核，或多个相对简单的核。定义如下参数：

n = 芯片上能包含的最大核数目。

k = 实际上实现的核数目（$1 \le k \le n$，$r = n/k$ 是一个整数）。

$perf(r)$ = 用 r 个处理器资源实现单个处理器所带来的连续性能提升，当 $r = 1$ 时，$perf(1) = 1$。

f = 多核上可并行的软件部分。

如果构造一个包含 n 个核的芯片，我们期望每个处理器核能带来 1 倍的连续性能提升，并且 n 个核能利用多达 n 个并行线程的并行性。同样，如果芯片有 k 个核，那么每个核应表现出一个 $perf(r)$ 性能，并且芯片能利用多达 k 个并行线程的并行性。我们可以修改阿姆达尔定律（式（18.1））来说明这个情况，表达如下：

$$\text{加速比} = \frac{1}{\dfrac{1-f}{perf(r)} + \dfrac{f \times r}{perf(r) \times n}}$$

(a) 判断这个修改的阿姆达尔定律。

(b) 利用 Pollack 规则，设定当 $n=16$ 时 $perf(r)=\sqrt{r}$。绘出当 $f=0.5$，$f=0.975$，$f=0.99$，$f=0.999$ 时加速比作为 r 的函数。

(c) 当 $n=256$ 时重复 (b) 问题。

18.2 ARM11 MPCore 的技术参考手册介绍 DIC 是一个内存映射。也就是说，核处理器使用内存映射 I/O 与 DIC 通信。回顾第 7 章内存映射 I/O 中，内存位置和 I/O 设备有独立地址空间。处理器把 I/O 模块的状态和数据寄存器视为内存位置，并采用同样的机器指令来访问内存和 I/O 设备。基于这些知识，图 18-11 的结构图使用什么路径进行核处理器与 DIC 通信？

计算机组成与体系结构的教学课题

许多教师都相信，研究或实践课题对于清楚理解计算机组成和结构的概念是至关重要的。不进行课题训练，让学生把握住某些基本概念和部件之间的相互关系是非常困难的。课题深化了书中引出的概念，使学生更好地了解处理器内部的工作，能激励学生并给予其信心：他们能完全掌握这些知识。

在本书的正文中，作者已试图尽可能简洁地给出了概念并提供了大量的习题来加深这些概念。许多教师还希望为这些内容补充一些课题，这个附录对教师手册中的支持资料予以某些指导。这些支持材料包括6类课题和其他的练习：

- 交互式模拟
- 研究类课题
- 模拟类课题
- 汇编语言课题
- 阅读/报告作业
- 写作作业
- 试题库

A.1 交互式模拟

本版的新增内容是交互式模拟。这些模拟为理解现代计算机系统的复杂设计特征提供了强有力的工具。如今的学生希望能在他们自己的电脑屏幕上展示各种复杂的计算机系统机制。总计20个模拟用来说明计算机组成和体系结构设计中的关键功能和算法。本书中在相关的地方给出了图标，表示对应的内容在本书网页上有相应的交互式模拟可供学生使用。

模拟允许用户设置初始条件，因此这些模拟可以作为学生作业。本书的教师资源中心（Instructor's Resource Center，IRC）提供了一些作业，每个交互式模拟一个。每份作业包括若干问题，可以布置给学生。

交互式模拟由马萨诸塞大学电气和计算机工程系的Israel Koren教授指导开发。马萨诸塞大学的Aswin Sreedhar设计了交互式模拟的作业。

表A-1 计算机组成与体系结构——各章的交互式模拟

第4章 cache存储器	
高速缓存模拟器	根据用户输入的高速缓存规范进行小容量高速缓存的模拟，根据用户输入或随机生成的输入序列，在模拟周期结束后显示高速缓存的内容
高速缓存的时间分析模拟	展示在用户指定的参数下，高速缓存的平均存取时间
多任务的高速缓存模拟	对十支持多任务的系统上的高速缓存建模
获得性的Victim高速缓存模拟器	比较二种不同的高速缓存策略
第5章 内部存储器	
多体交叉存储器模拟器	展示交叉内存的效果
第6章 外部存储器	
独立磁盘冗余阵列（RAID）模拟器	确定外存（Storage）效率和可靠性
第7章 输入/输出	
I/O系统设计工具	计算不同输入/输出系统的代价和性能

（续）

第 8 章 操作系统支持	
页替换算法模拟器	比较最近最少使用（LRU）、先进先出（FIFO）和最优替换算法
其他页替换算法	比较更多页面替换策略
第 12 章 CPU 结构和功能	
预约表模拟器	对作为一种流水线系统任务流表示的预约表进行评价
分支预测模拟器	展示三种不同的分支预测方案
分支目标缓冲	分支预测/分支目标缓冲的集成模拟器
第 13 章 精简指令集计算机	
MIPS R3000 五级流水线模拟器	模拟流水线
循环展开模拟器	模拟循环展开的软件技术，循环展开用于发掘指令级并行性
第 14 章 指令级并行性和超标量处理器	
使用静态/动态指令调度的流水线模拟器	MIPS 流水线更为复杂的模拟
重排序缓冲器模拟器	模拟 RISC 流水线中的指令重排序
记分牌模拟器	模拟用于不少处理器中的一种指令调度技术
Tomasulo 算法模拟器	模拟指令调度的另一种技术
另一个 Tomasulo 算法模拟器	Tomasulo 算法的另一种模拟
第 17 章 并行处理	
向量处理器模拟器	展示向量处理指令的执行

A.2 研究类课题

巩固基本概念和向学生传授研究技巧的一种有效的方式是，给学生指派研究课题。这种课题应该涉及文献检索、产品的万维网检索，实验室的研究活动和标准化的努力。课题应该指派给小组，简单的课题可指派给个人。不论哪种情况，最好尽快要求递交某种课题申请，以使教师有充裕的时间评价学生的决策是否合适，准备是否充分。课题的学生公告应包括：课题申请格式、最终报告格式、中间和最后期限的时间表、课题可选题目的列表。

学生可选择所列的题目之一，也可提交他们自己拟定的程度相当的课题。教师资源中心提供了课题申请和最终报告的推荐格式，以及可行的研究课题列表。

A.3 模拟类课题

理解处理器的内部操作、学习和评价某些设计权衡及其性能影响的最好方式是，模拟处理器的关键部件。SimpleScalar 和 SMP cache 是服务于此目的的两个有用工具。

与实际硬件实现相比，模拟为研究和教学提供如下好处：

- 模拟很容易做到修改一个体系的不同元素，改变各种部件的性能特征，然后分析这些修改的效果。
- 模拟可提供详细的性能统计资料，这些有助于对性能权衡的理解。

SimpleScalar

SimpleScalar［BURG97，MANJ01a，MANJ01b］是一组工具软件，它们可用于当代处理器和系统上模拟实际程序的运行。这组工具软件包括编译器、汇编器、链接器以及模拟与可视化工具。SimpleScalar 提供的

处理器模拟器，包括从极其快速的功能模拟器到详细的乱序发射的超标量处理器的模拟器，后者能支持无阻塞的 cache 和猜测执行。指令集体系结构和组成的参数可修改，以进行各种实验。

本书的教师资源中心提供了 SimpleScalar 的简要介绍，它指导学生如何安装和启动 SimpleScalar。手册还包括了一些推荐的课题作业。

SimpleScalar 是一款可运行于大多数 Unix 平台上的可移植软件包。可从 SimpleScalar 官方网站下载此软件，非商业用途可免费使用。

SMP cache

SMP cache 是一款跟迹驱动的模拟器，用于对称多处理器上的高速缓存系统的分析与教学［RODRI1］。该模拟器根据建立在这些系统的体系结构基本原理上的模型进行模拟。它具有用户友好的全图形界面。利用该模拟器能学习的内容包括：程序局部性行为，处理器数目的影响，cache 一致性协议、总线仲裁策略、映射、替换策略、高速缓存大小（高速缓存中块数），高速缓存组数（针对组相联高速缓存）、每块的字数（存储器块的大小）。

本书的教师资源中心提供了 SMP cache 的简要介绍，它指导学生如何安装和启动 SMP cache。手册中还包括一些推荐的课题作业。

SMP cache 是一款可在装有 Windows 操作系统的个人电脑上运行的可移植软件包，可从 SMP cache 官方网站下载此软件，非商业用途可免费使用。

A.4　汇编语言课题

汇编语言程序设计通常用于教授学生底层硬件部件和计算机系统结构的基础知识。CodeBlue 是美国空军科学院（U.S. Air Force Academy）开发的一种简化的汇编语言程序，目的在于利用可视化的模拟器，帮助学生在一课时内了解和学习汇编语言的概念。CodeBlue 的开发人员也希望通过这个简化的汇编语言使学生们对使用汇编语言更感兴趣。CodeBlue 汇编语言比其他简化的机器指令集（如 SC123），更为简单，但学生们使用它还是能开发出有意思的汇编级程序，并进行比赛，其能力与远为复杂的 SPIMbot 模拟器相当。最重要的是，通过 CodeBlue 编程，学生们可以学到计算机系统结构的基本概念，例如指令和数据在内存中的编排，程序控制结构的实现，以及各种寻址方式等。

为了便于课题的设计，CodeBlue 的开发人员提供了一个可视化的开发环境，以便学生们创建自己的程序，查看代码在内存中的表示，单步执行程序，并在虚拟内存环境中模拟不同对战程序之间的战斗。

设计课题可以利用“核心大战”（Core War）的概念来组织。核心大战是 20 世纪 80 年代早期出现的一种程序设计比赛，并持续流行了大约 15 年之久。核心大战包括 4 大主要要素：一个包含 8000 个地址的内存空间，一个简化的汇编语言 Redcode，一个叫做 MARS（Memory Array Redcode Simulator）的可执行程序，以及一组相互对战的程序。核心大战的过程是，两个对战程序被装载到随机选择的内存地址上；对战程序相互之间不知道另一个程序在内存什么位置。MARS 以一种简单的方式分时执行各个程序的代码，两个对战程序交替执行：首先第一个程序的一条指令被执行，然后第二个程序的一条指令被执行，循环往复。每个对战程序在所分配的执行周期中做什么事情则完全由程序设计者决定。程序的目标是通过破坏另一个程序的指令，从而摧毁另一个程序的执行。CodeBlue 与核心大战不同之处就是，用 CodeBlue 交互式执行界面替代了 Redcode。

教师资源中心提供了 CodeBlue 执行环境、学生使用手册以及其他的支持资料和建议的课程作业。

A.5　阅读/报告作业

另一种强化课程概念和给学生某些研究经验的好方式是，要求学生阅读并分析文献中的论文。教师资源中心中提供了推荐的论文列表，每章一或两篇文章。所有的这些文章可在教师资源中心下载。教师资源中心也提供了这些作业的建议。

A.6　写作作业

写作作业对于工程技术类专业的学习过程而言，如数据通信和网络工程，是一个有力的效果倍增器。

"跨课程写作（Writing Across Curriculum）"运动（http://wac.colostate.edu/）的支持者报告描述，写作作业对于促进学习有着显著的好处。写作作业促使学生对特定问题进行更为细致和全面的思考。另外，写作作业有助于解决学生学习时的一个不良习惯，即总是倾向于花最少的努力去解决一个问题，通常表现是了解了一些问题描述和求解方法后，就草草收工，而不求甚解。

教师资源中心提供了一些建议的写作作业，分章布置。教师可能会发现这是教授本书内容最重要的资源。作者非常感谢对此提出任何反馈意见，并提出更多其他写作作业的教师。

A.7 试题库

本书的教师资源中心网站提供了针对本书的试题库。对于书中的每一章，试题库提供了适用问题、多项选择题、填空题等题型的习题。利用试题库，可以有效地评估学生对书中内容的掌握情况。

汇编语言及相关主题

要点

- 汇编语言是针对特定处理器机器语言的符号化表示，带有增强了的附加语句类型，以使程序员输入并控制汇编器的命令。
- 汇编器是将汇编语言转换成机器代码的程序。
- 产生活动进程的第一步是把程序装入主存并且创建一个进程映像。
- 链接器用于解决被装载模块之间的所有引用。

汇编语言的概念在第 11 章进行了简单的介绍，本附录提供更多的细节并且涵盖了很多相关主题。为什么要学习汇编语言程序（相对其他高级语言程序）有很多的理由，下面列出了其中一些理由：

(1) 它阐明了指令的执行。

(2) 它表明数据在内存中是如何表示的。

(3) 它显示了一个程序如何与操作系统、处理器、输入输出系统交互。

(4) 它阐明了程序如何访问外部设备。

(5) 掌握了汇编语言编程，可以使学生成为更好的高级语言程序员，因为他们能更好地理解目标语言，而每个高级语言程序最终都要翻译为对应的目标语言执行。

我们将从汇编语言的基本要素来开始这章的学习，并以 x86 体系结构作为我们的例子[1]。然后我们将学习汇编器（Assembler）的操作，接下来再讨论链接器（Linker）和装载器（Loader）。

表 B-1 定义了一些在本附录中使用的关键术语。

表 B-1 本附录中使用的关键术语

汇编器
将汇编语言转换成机器代码的程序。

汇编语言
是针对特定处理器机器语言的符号化表示，带有增强了的附加语句类型，以使程序的编写并提供控制汇编器的命令。

编译器
编译器是一种将其他程序从源程序（或程序设计语言）转化到机器语言（目标代码）的程序。一些编译器输出汇编语言，然后汇编语言被单独的汇编器翻译成机器语言。编译器与汇编器不同，通常它的每个输入语句不对应单一机器指令或固定的指令序列。编译器可能支持这样的特征：自动申请变量、任意算术表达式、像 for 和 while 这样的循环控制结构、变量可见范围、输入/输出操作、高级函数以及源代码可移植性。

可执行代码
由源语言处理器（比如汇编器和编译器）生成的机器代码，是一种可以在计算机上运行的软件。

指令集
特定机器的所有可能的指令集合，它是特定处理器的机器语言指令的集合。

链接器
把一个或多个包含目标代码的文件，从已编译的独立的模块合并为可装载或可执行的单一文件的实用程序。

装载器
把一个可执行程序拷贝到内存执行的例行程序。

机器语言或机器代码
实际上被计算机读取和执行的计算机程序的二进制表示。机器代码程序由一系列机器指令（可能带有数据）组成。指令是大小相同或者不同的二进制串（例如，许多现代 RISC 微处理器的一个 32 位字）。

目标代码
程序源代码的机器语言的表示。目标代码由编译器或汇编器生成。然后由链接器转化成可执行代码。

---

[1] 有很多针对 x86 的汇编器，我们的例子使用 NASM（Netwide Assembler），一个开放源代码的汇编器。本书的 Web 站点提供了 NASM 使用手册的副本。

B.1 汇编语言

汇编语言是一种与机器语言具有一步之遥的程序设计语言。通常，每条汇编语言指令被汇编器转换成一条机器指令。汇编语言依赖于硬件，每种处理器都有一种不同的汇编语言。尤其是，汇编语言指令能指定访问处理器的特定寄存器，可使用处理器的所有操作码，并使用处理器的不同寄存器的位长和机器语言的操作数。因此一位汇编语言程序员必须懂计算机体系结构。

程序员很少用汇编语言编写应用程序甚至系统程序。高级语言提供更强的表达力和简洁性，大大简化了程序员的任务。使用汇编语言而不是高级语言的缺点有［FOG08a］：

（1）开发时间。用汇编语言写代码比用高级语言写代码要花更长的时间。

（2）可靠性和安全性。使用汇编代码更容易犯错。汇编器不检查调用约定（calling conventions）与寄存器保存约定（register save conventions）是否一致，也不会检查入栈（push）和出栈（pop）指令的数量对于所有可能的分支和执行路径都一样。在汇编代码中有非常多类似的可能出错之处，以至于采用汇编语言会影响项目的可靠性和安全性，除非你有一种非常系统的方法去测试和验证汇编语言程序。

（3）调试和验证。汇编代码更难于调试和验证，因为它比高级语言代码有更多可能的错误。

（4）可维护性。汇编代码更难修改和维护，因为汇编语言允许出现意大利面条式的非结构化代码以及各种让人难以理解的技巧。因此详细的文档和一致的编程风格对于汇编语言程序设计来说是十分必要的。

（5）可移植性。汇编代码是基于特定平台的，将其移植到另一不同的平台是困难的。

（6）系统代码能使用其内建函数，而汇编代码不能。最优秀的现代C++编译器具有内建函数（intrinsic function）来访问系统控制寄存器和其他系统指令。当内建函数可用时，设备驱动程序和其他的系统代码不再需要使用汇编代码。

（7）应用代码可使用内建函数或向量类取代汇编代码。最好的现代C++编译器具有内建函数来实现向量运算和执行其他特殊指令。这些内建函数以前是用汇编语言编写的。

（8）近些年编译器的功能提高了很多。如今最好的编译器功能很强大。想利用汇编语言对程序进行优化，除非有非常丰富的专业知识和实践经验，否则不会比最好的C++编译器做得更好。

但是偶尔使用汇编语言仍有一些优点，包含如下所列出的［FOG08a］：

（1）调试与验证。查看编译器产生的汇编代码或调试器中的反汇编窗口，对于查错和检验编译器是如何优化一段特定代码是非常有用的。

（2）编写编译器。理解汇编语言编码技术对于编写编译器、调试器和其他开发工具来说是很有必要的。

（3）嵌入式系统。小型嵌入式系统的软硬件资源远比个人计算机和大型机少。如果在嵌入式系统上优化代码的运行时间和存储空间，用汇编语言编程是必需的。

（4）硬件驱动程序和系统代码。访问硬件、系统控制寄存器等，可能有时候很难或者不能使用高级程序设计语言来实现。

（5）使用那些从高级程序语言中无法访问的指令。一些特定的汇编指令是没有等价的高级语言形式的。

（6）自修改代码。通常而言，自修改代码（self-modifying code）并不是很有用，因为它会影响代码缓存的运作效率。不过，在特定条件下，比如某个数学程序允许用户定义数学函数，这些数学函数要被计算多次，这就需要一个小型编译器来产生相应代码，那么这时自修改代码就有优势了。

（7）优化代码的存储空间。如今，存储器是如此便宜，以至于无需使用汇编语言来减少代码存储空间。但是，高速缓存的容量仍是很关键的资源，因此，为了使代码的存储空间适应高速缓存，使用汇编语言优化一些关键的代码片段是很有用的。

（8）优化代码的运行时间。通常，现代C++编译器能在绝大多数情况下很好地优化代码，但仍有个别情况下优化得很糟。而在这些情况下，小心使用汇编语言编程可以极大地提升（运行）速度。

（9）函数库。对函数库的代码进行优化所带来的总效益，要高于对应用程序代码进行的优化，因为函数库会被程序员大量使用。

（10）使函数库能和多种编译器、多种操作系统相兼容。通过为函数库中的函数建立多个调用入口，

可以使这些函数与不同的编译器和不同的操作系统相兼容。而这就需要汇编语言编程。

汇编语言和机器语言有时候被错误地当作同义词。机器语言是由能被处理器直接执行的指令组成。每一条机器指令都是二进制串，它包含相关的操作码、操作数以及其与执行相关的位（例如标志位）。为了方便，可以将操作码和寄存器符号化来取代二进制指令。汇编语言大量使用符号名称，包括给特定的内存地址和指令地址分配名称。此外，汇编语言还包括这样的语句，它们不可以直接执行，但可以作为汇编器的指令，指示如何把汇编语言程序转换为机器代码。

B.1.1 汇编语言要素

典型的汇编语言语句格式如图 B-1，它包含四个组成部分：标号（Label）、助记符（Mnemonic）、操作数（Operand）和注释（Comment）。



图 B-1 汇编语言语句结构

标号 如果标号存在，汇编器就把标号定义为地址的等价物，该地址为此标号对应指令的地址，就是这条指令所产生的目标代码第一个字节被装入内存空间的地址。随后，程序员可能使用标号作为地址或者作为数据填在另一指令的地址字段中。汇编器把汇编程序转换为目标程序时，会用实际地址值替换标号。标号在分支指令中使用得最频繁。

这里有一程序片段作为例子：

```
L2:   SUB    EAX,    EDX    ; 寄存器EAX的值减去寄存器EDX的值，结果
                            ; 存入寄存器EAX

      JG     L2             ; 如果上一条减法指令运算结果为正，则跳转到
                            ; 标号L2的地址执行
```

程序将从 L2 开始不断循环，直到减法指令的结果为零或负。因为，如果结果为正，当 jg 指令被执行时，将执行跳转，处理器会将标号 L2 对应的地址放入程序计数器中，从而发生跳转。

使用标号的理由如下：

(1) 标号使程序中指令的位置更容易找到和记住。

(2) 当修改一个程序导致指令移动时，标号名称可以保持不变，但地址变化。当程序被重新汇编时，汇编器将在所有指令中自动改变标号对应的地址，从而反映指令的移动。

(3) 程序员不必去计算相对或绝对的内存地址，而只需使用标号替代地址。

助记符 助记符是汇编语言语句功能或者操作码的代名。正如接下来讨论的，一条汇编语言语句可以对应一条机器指令、一条汇编指令或者一个宏。就机器指令而言，助记符是和特定操作码相关的符号名。

表 10-8 列出了很多 x86 指令助记符或者指令名。文献［CART06］的附录 A 列出了 x86 指令和何条指令的操作数，以及指令执行结果对条件码的影响。NASM 使用手册的附录 B 提供更多对每条 x86 指令的详尽描述。这些资料都可以在本书的网站上查到。

操作数 汇编语言语句包括零个或更多操作数。每个操作数可以是一个立即数、或一个寄存器值、或一个内存地址。通常，汇编语言对于如何区别三种类型的操作数，以及如何指定寻址方式制订了一些约定。

对 x86 体系结构，汇编语言语句可能通过名字来标识寄存器操作数。图 B-2 显示了 x86 通用寄存器组，各个寄存器的符号名，及其位编码。汇编器可将寄存器符号名转换成对应的二进制识别码。

正如第 11.2 节所讨论的，x86 体系结构有丰富的寻址方式，每种寻址方式在汇编语言中都必须符号化表示。这里列举一些常见的例子，比如寄存器寻址方式，通过在指令中使用寄存器名来表示。例如，

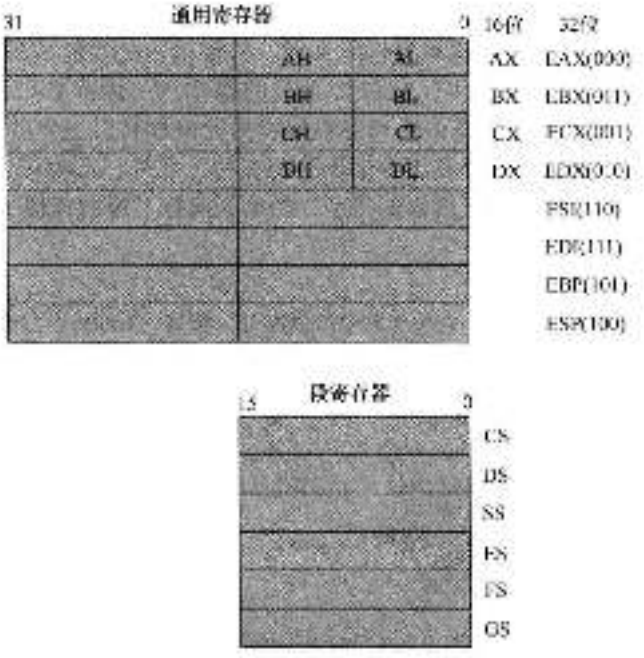


图 B-2 Intel x86 程序运行可使用的寄存器

MOV ECX, EBX，该指令将寄存器 EBX 的内容拷贝到 ECX 中。对于**立即数寻址方式**，立即数的值直接保存在指令编码中。例如，MOV EAX, 100H，该指令将十六进制值 100 拷贝到寄存器 EAX 中。立即数值也可以用带后缀 B 的二进制数表示或者没有前缀的十进制数表示。因此，MOV EAX, 100000000B 和 MOV EAX, 256 以及 MOV EAX, 100H 是等价的表示。对于直接寻址方式，指令需要指向一个内存地址，该地址在 x86 中以相对于 DS 段寄存器的偏移量来表示。下面的例子很好地解释了直接寻址方式。假设 16 位数据段寄存器 DS 包含值 1000H，而我们准备执行下列指令：

```
MOV  AX, 1234H
MOV  [3518H], AX
```

首先 16 位寄存器 AX 初始化为 1234H，然后，在第二行，AX 的内容被移动到逻辑地址 DS:3518H。这个地址是这样形成的，把 DS 的内容左移 4 位加上 3518H 形成 32 位逻辑地址 13518H。

注释 所有汇编语言都允许在程序中添加注释。注释可以放在汇编语句的右边，也可以单独占据整个文本行。无论哪种情况，注释都开始于一个特殊的字符，该字符告诉汇编器，该行的其余部分是注释，这些注释也就会被汇编器忽略。通常，x86 体系结构的汇编语言使用分号（;）作为注释符。

B.1.2 汇编语言语句的类型

汇编语言语句有四种类型：指令，指导语句（directives），宏定义（macro definition），注释。注释语句就是整个语句都是注释的语句。本节将简单介绍其他类型的语句。

指令 在汇编语言程序中，大部分非注释语句都是指令语句，即机器语言指令的符号化表示。几乎没有例外，汇编语言指令和机器语言指令一一对应。汇编器可解析各种符号引用，并且把汇编语言指令转化成组成机器指令的二进制串。

指导语句 指导语句又叫伪指令（pseudo－instruction），这些汇编语言语句不能被直接转换成机器指令。不过，指导语句作为给汇编器的命令，指示了汇编器在汇编过程中要进行的一些特殊操作。下面是些特殊操作的例子：

- 定义常量
- 指定数据存储的内存区域
- 初始化内存区域

- 将表或其他已定数据放置在内存中
- 允许对其他程序引用

表 B-2 列出了一些 NASM 伪指令。作为一个例子，考虑下面的语句序列：

```
L2      DB      'A'             ;初始化为 ASCII 码 A (65) 的字节值量
        MOV     AL,[L2]         ;把位于 L2 的字节内容拷贝到 AL
        MOV     RAX,L2          ;把 L2 对应的地址存储到 EAX
        MOV     [L2],AL         ;把 AL 的内容拷贝到位于 L2 的字节中
```

表 B-2 一些 NASM 汇编语言伪指令

a) RESx 和 Dx 伪指令字母

单位	字母
字节	B
字（2 字节）	W
双字（4 字节）	D
四字（8 字节）	Q
十字节	T

b) 伪指令

名称	描述	例子
DB, DW, DD, DQ, DT	初始化的内存位置	L6 DD 1A92H ;位于 L6 的双字初始化为 1A92H
RESB, RESW, RESD, RESQ, REST	预留不进行初始化的内存位置	BUFFER RESB 64 ;以 BUFFER 为起始位置，预留 64 字节
INCBIN	在汇编器最后输出的目标文件中插入指定的二进制文件	INCBIN "file.dat" ;包含"file.dat"这个文件
EQU	把符号定义为常量值	MSGLEN EQU 25 ;常量 MSGLEN 等于十进制 25
TIMES	重复若干次指令	ZEROBUF TIMES 64 DB 0 ;把 64 字节缓存初始化为 0

如果标号被使用，它解释为数据的地址（或偏移量）。如果标号被放在方括号里，它被解释为这个地址的数据。

宏定义 宏定义在好几个方面类似于子程序（subroutine）。一个子程序是一个程序段，这个程序段写一次能使用多次，子程序能从程序的任何地方调用。当程序被编译或汇编，子程序仅被装载一次。与子程序类似，宏定义也是程序员可以写一次能使用多次的代码段。子程序和宏定义主要的不同在于，当汇编器遇到一个宏调用，它用宏本身来取代宏调用，这个过程称作宏扩展（macro expansion）。因此，如果一个宏在一段汇编语言程序中被定义并且被调用 10 次，那么宏的 10 个实体将在汇编代码中出现。本质上，子程序在运行时由硬件处理，而宏由汇编器在汇编的时候处理。就模块化程序而言，宏提供了和子程序一样的优点，但是没有子程序被调用和返回的运行开销。缺点是宏在目标代码中消耗了更多的空间。

在 NASM 和很多其他汇编器中，有单行（single-line）宏和多行（multiple-line）宏区别。在 NASM 中，用%DEFINE 伪指令定义单行宏。下面的例子展示了单行宏及其扩展。首先，定义两个宏：

```
%DEFINE B(X) =2*X
%DEFINE A(X) =1 + B(X)
```

在汇编语言程序中某处可能有下面的语句：

```
MOV  AX,  A(3)
```

汇编器会把这个语句扩展成：

```
MOV  AX,  1+2*3
```

在汇编时，这条语句将被转换为机器指令，它把立即数 17 赋给寄存器 AX。

多行宏用助记符 &MACRO 定义。下面是一个多行宏定义的例子：

```
%MACRO  PROLOGUE  1
        PUSH EBP            ;把 EBP 的内容放入
                            ;由 ESP 指向的栈顶，
                            ;并且 ESP 的内容减 4
        MOV  EBP,  ESP      ;ESP 的内容拷贝给 EBP
        SUB  ESP,  %1       ;从 ESP 中减去该宏第一个参数的值
```

在%MACRO 行宏名后的数字 1 定义宏希望接收的参数数目。在宏定义里使用%1 表示宏调用的第一个参数。

宏调用

```
MYFUNC:  PROLOGUE  12
```

将被扩展成下面几行代码：

```
MYFUNC:  PUSH  EBP
         MOV   EBP, ESP
         SUB   ESP, 12
```

B.1.3 示例：最大公约数程序

作为使用汇编语言的例子，我们看一个计算两个整数的最大公约数的程序。我们定义整数 a 和 b 最大公约数如下：

$$\gcd(a,b) = \max[k, a \text{ 整除 } k \text{ 且 } b \text{ 整除 } k]$$

所谓 a 整除 k，就是说 a 除以 k 不会有余数。求最大公约数的欧几里得（Euclid）算法基于下述定理：对任何非负整数 a 和 b，有

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

下面是实现欧几里得算法的 C 程序：

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    if (a==0&&b==0)
        a=1;
    else if (b==0)
        b=a;
    else if (a!=0)
        while (a!=b)
            if(a<b)
                b-=a;
            else
                a-=b;
    return b;
}
```

图 B.3 显示了上面程序的两个汇编语言版本。左边程序是 C 编译器生成的；右边程序是直接手写的汇编语言。后者使用了大量编程技巧，使程序更紧凑，执行更有效。

```
gcd:    mov     ebx, eax
        mov     ecx, ecx
        test    ebx, ebx
        jne     L1
        test    edx, edx
        jne     L2
        mov     eax, 1
        ret
L1:     test    eax, eax
        jns     L2
        mov     eax, ebx
        ret
L2:     test    ebx , ebx
        je      L5
L3:     cmp     ebx, eax
        je      L5
        jae     L4
        sub     ecx, ebx
        jmp     L3
L4:     sub     ecx, eax
        jne     L3
L5:     ret
```

a) 编译器生成的程序

```
gcd:    neg     eax
        je      L3
L1:     neg     eax
        xchg    eax, ecx
L2:     sub     eax, edx
        jz      L2
        jns     L1
L3:     add     eax,ecx
        jne     L4
        inc     eax
L4:     ret
```

b) 用汇编语言来手工编写

图 B.3 求最大公约数的汇编程序

B.2 汇编器

汇编器是一个以汇编语言程序为输入，输出目标代码（object code）的应用程序。目标代码是二进制文件。汇编器把这个文件看作从地址 0 开始的内存块。

汇编器有两种设计方式：两趟（two-pass）汇编器和单趟（one-pass）汇编器。

B.2.1 两趟汇编器

我们先看下两趟汇编器，它更通用且更易理解。两趟汇编器扫描源代码两次来生成目标代码（见图 B-4）。

第一趟 在第一趟中，汇编器仅处理标号的定义。汇编器扫描源代码，建立符号表（symbol table）。符号表中包含所有标号和标号位置计数器（location counter，LC）的值。目标代码第一个字节初始化为 LC = 0。在第一趟扫描中，汇编器检查每条汇编语句。虽然此时汇编器不准备转换指令，但它还是要充分检查每条指令来确定每条机器指令的长度，从而确定 LC 的增量。这可能不仅要检查操作码，而且要检查操作数和寻址方式。

像 DQ 和 REST 这样的伪指令（见表 B-2），它们指定了要预留多少内存空间，因此位置计数器 LC 会按照这些伪指令指定的内存空间大小进行累加。

当汇编器遇到有标号的语句的时候，它把标号和当前 LC 的值放入符号表。汇编器会继续扫描完所有汇编语句，并对所有遇到的标号做同样操作。

第二趟 第二趟扫描时，汇编器又从头至尾地把源程序读一遍，每一条指令都翻译为对应的二进制机器指令。翻译过程包括如下操作：

(1) 把助记符翻译为二进制操作码。

(2) 用操作码决定指令的格式，以及指令中各个字段的位置和长度。

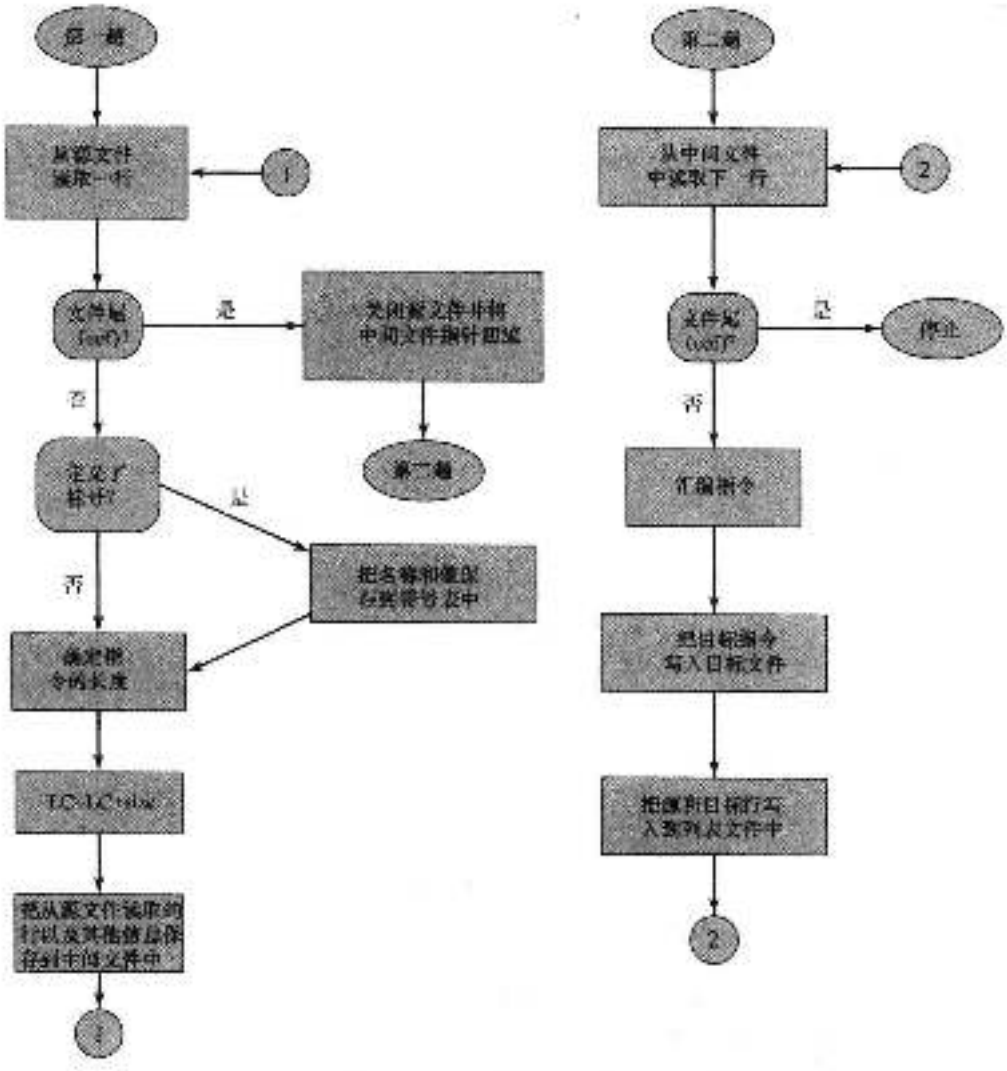


图 B-4 两趟汇编器流程图

(3) 把每一个操作数名称翻译为对应的寄存器或内存代码。

(4) 把每一个立即数转化为二进制串。

(5) 利用符号表把任何对标号的引用转换为对应的 LC 值。

(6) 在指令中设置其他任何需要的二进制位，包括寻址方式指示位、条件码位等。

以图 B-5 中的 ARM 汇编语言指令为例，ARM 汇编语言指令 ADDS r3，r3，#19 转换为了二进制机器指令 1110 0010 0101 0011 0000 0001 0011。



图 B-5 把 ARM 汇编指令翻译为二进制机器指令

第零趟（zeroth pass）　绝大多数的汇编语言具有定义宏的功能。如果汇编程序中定义了宏，那么汇编器必须要在第一趟扫描前还多加一次扫描，以便处理所有的宏。因此，汇编语言通常都要求把宏定义置于程序文件的开头。

汇编程序在第零趟遍历中会读取所有的宏定义。一旦所有的宏都被识别后，汇编器就继续扫描源代码，每当遇到宏调用时，就按照调用的参数把宏展开。这个宏处理的遍历过程最后生成了一份新的源代码，其中所有的宏调用都被展开，而程序开头的所有宏定义就都可以删除了。

B.2.2　单趟汇编器

实现一个只对源代码做一次扫描的汇编器是可以做到的（不包括宏遍历）。单趟汇编器的主要困难是对标号的提前引用（forward reference）。指令中操作数使用的标号，可能是在已扫描的源代码中还未定义的符号。因此，汇编器不能确定在翻译指令时，应该插入的相对地址是什么。

从根本上来说，解决提前引用的过程是这样的：当汇编器遇到了一个未定义的指令操作数符号，那汇编程序就会采取如下步骤：

(1) 在翻译二进制指令时将指令操作数字段全部置空（全零）。

(2) 把作为操作数的符号输入到符号表中。符号表的对应项标记这个符号是未定义的。

(3) 把涉及未定义符号操作数的指令的地址，添加到一个与符号表对应项相关的提前引用列表中。

当在后续扫描中遇到一个符号定义时，即可确定与它有关的位置计数器 LC 值。汇编程序把 LC 值插入到符号表中的该符号对应的项。如果有和该符号相关的提前引用表，那么汇编程序就在以前生成的指令中的合适位置插入该符号的地址。

B.2.3　示例：求素数程序

现在我们来看一个带指导语句（directive）的例子。这个程序用于查找素数。素数是只能被 1 和自身整除的数。没有现成的公式可以用来查找素数。这个程序所采用的基本方法是对于一个给定的上限，找到在上限之下所有奇数的因子。如果奇数没有因子，那么该奇数就是素数。图 B-6 用 C 语言展示了其基本算法，图 B-7 使用 NASM 汇编语言展示了同样的算法。

```
unsigned guess;                         /* 当前考察是否为素数的数 */
unsigned factor;                        /* 当前被考察数的可能的因子 */
unsigned limit;                         /* 以该值为寻找素数的上限 */

printf ("Find primes up to : ");
scanf("%u", &limit);
printf ("2\n");                         /* 对于最开始的 2 和 3 这两个素数 */
printf ("3\n");                         /* 做特殊处理 */
guess = 5;                              /* 从 5 开始进行考察 */
while (guess <= limit) {                /* 寻找当前被考察数的因子 */
    factor = 3;
    while (factor * factor < guess && guess% factor != 0)
    factor += 2;
    if (guess % factor != 0)
        printf ("%d\n", guess);
    guess += 2;                         /* 只考虑奇数 */
}
```

图 B-6　求素数的 C 程序

```
%include "asm_io.inc"
segment .data
Message db "Find primes up to: ", 0

segment .bss
Limit resd 1                            ; 以此为寻找素数的上限
Guess resd 1                            ; 当前考察的数

segment .text
     global _asm_main
_asm_main:
     enter 0,0                          ; 程序运行的启动例程
     pusha

     mov eax, Message
     call print_string
     call read_int                      ; scanf("%d", &limit);
     mov [Limit], eax
     mov eax, 2                         ; printf("2\n");
     call print_int
     call print_nl
     mov eax, 3                         ; printf("3\n");
     call print_int
     call print_nl

     mov dword [Guess], 5               ; Guess = 5;
while_limit:                            ; while (Guess <= Limit)
     mov eax, [Guess]
     cmp eax, [Limit]
     jnbe end_while_limit               ; 因为使用了无符号整数，故此处使用jnbe

     mov ebx, 3                         ; ebx保存可能的因子，此处相当于 factor=3;
while_factor:
     mov eax,ebx
     mul eax                            ; edx:eax = eax*eax
     jo end_while_factor                ; 如果上一条乘法指令的运算结果超出了eax的长度
     cmp eax, [Guess]
     jnb end_while_factor               ; 如果(factor * factor<guess)
     mov eax,[Guess]
     mov edx,0
     div ebx                            ; edx-edx:eax% ebx
     cmp edx, 0
     je end_while_factor                ; 如果((guess% factor )=0)

     add ebx,2; factor += 2;
     jmp while_factor
end_while_factor:
     je end_if                          ; 如果((guess% factor )=0)
     mov eax,[Guess]                    ; printf("%u\n")
     call print_int
     call print_nl
end_if:
     add dword [Guess], 2               ; guess += 2
     jmp while_limit
end_while_limit:

     popa
     mov eax, 0                         ; 退出返回C
     leave
     ret
```

图 B-7 求素数的汇编程序

B.3 装载和链接

创建一个活动进程（active process）的第一步是装载一个程序到主存储器，并创建一个进程映像（process image，见图 B-8）。图 B-9 描述了对于大多数系统来说典型的链接和装载过程。图中的应用程序由许多不同的模块组成，这些模块的目标代码或者通过编译生成，或者通过汇编生成。链接器将这些模块链

接起来，解决模块之间的相互调用。如果这些模块中还有对于库例程（library routine）的调用，链接器也会对此进行解析。库例程的代码可能成为最后可执行程序的一部分，也可能作为运行时被操作系统装载的共享代码被调用。在本节，我们会介绍链接器和装载器的主要功能。首先，我们讨论重定位（relocation）的概念。接下来，为清楚起见，我们描述一个单一程序模块程序执行时的装载情况，此时不需要用到链接。然后，我们再考察同时涉及链接和装载的情况。

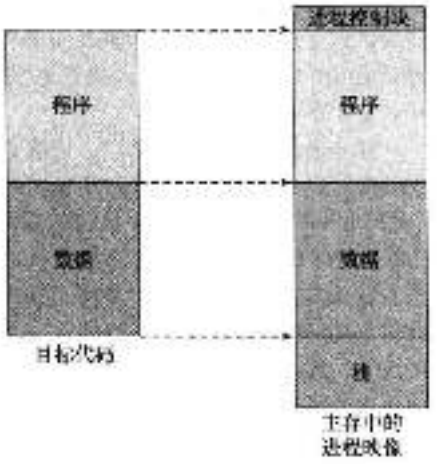


图 B-8 装载过程示意图

B.3.1 重定位

在多道程序（multiprogramming）系统中，可用的主存通常被多个进程共享。通常来说，程序员不可能事先知道，在运行自己的程序时，主存中还驻留有哪些其他的程序。另外，为了能充分利用处理器，多道程序系统都能够把活动进程调（swap）入，或者调出主存。这一般通过提供一个大的进程映像，其中存放准备好要执行的进程来实现。一旦一个程序被调出主存到硬盘，那么很难保证在下一次把这个程序重新调入主存时，操作系统还把它放

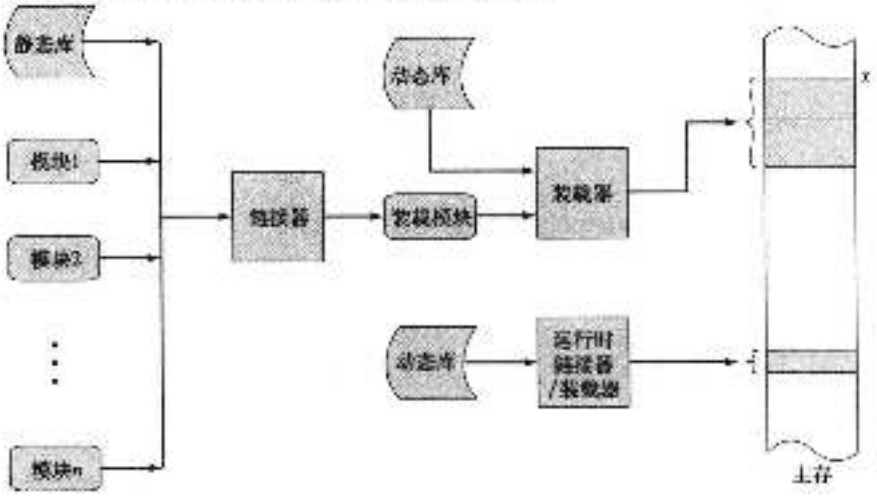


图 B-9 链接与装载过程

置到上次调出前所在的主存位置。相反，我们需要能够把这个进程重新定位到新的主存位置上。

因此，我们事先无法确定一个程序在装入主存时，它会被放置到什么位置。我们必须允许程序在被调入和调出时，被操作系统移动到主存中其他地方。这些事实导致了一些技术上需要考虑的问题，如图 B-10 所示。图中显示了一个进程在主存中的映像。简单起见，假设进程映像占据了主存中一段连续的区域。显然，操作系统需要知道进程控制信息的位置，程序运行栈的位置，以及该进程开始执行的入口地址。因为操作系统管理着全部存储器，而且负责把进程导入到主存中，所以这些地址信息对于操作系统而言是容易获得的。不过，处理器必须解决程序自身内部的内存地址引用。就像分支指令，其中包含了在分支跳转时要执行的下一条指令的地址。而数据访问指令中可能包含被访问数据的字节或字地址。因此，处理器硬件和操作系统软件必须能够根据当前程序在主存中的位置，把程序代码中对内存中指令和数据的引用，翻译为真实的物理内存地址。

B.3.2 装载

在图 B-9 中，装载器把装载模块放置在内存中以 a 为起始的地址上。在装载程序的过程中，图 B-10 中显示的进程寻址要求必须满足。一般有三种方法可供选择：绝对装载、可重定位装载、动态运行时装载。

绝对装载 绝对装载器要求给定的装载模块始终被载入到内存中的相同位置。因此，呈现在装载器中的装载单元，其地址必须是明确的，或者说是绝对的内存地址。例如，如果在图 B-9 中 x 的地址是 1024，那么，在装入内存的装载模块中，首个字的地址是 1024。

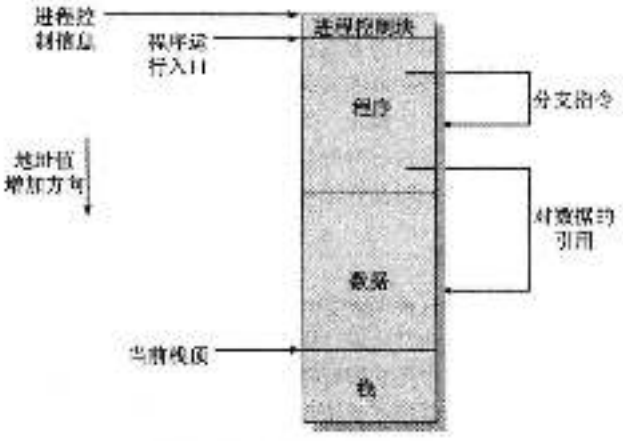


图 B-10 进程的寻址需求

在一段程序内的内存引用，其特定地址值的分配可以由程序员或编译时或汇编阶段完成（见表 B-3a）。不过，由程序员确定地址分配的方法有几个缺点。首先，每个程序员得知道将模块装入内存的预期的分配策略。其次，如果程序有任何涉及在模块中进行插入和删除之类的改变，那么此插入或删除点后所有的地址都将改变。因此，更可取的做法是，允许把在程序中的内存引用表述符号化，并且在编译和汇编时解析这些符号的引用。图 B-11 体现了这一方法。每一个指令或数据项的引用最初由符号表示。在准备输入绝对装载器时，汇编器或编译器将把所有的符号引用转换为明确的地址（在这个例子中，模块将载入到起始地址为 1024 的内存中），正如图 B-11b 所示那样。

表 B-3 地址绑定

a）装载器

绑定时间	功 能
编程时间	在程序中所有的实际物理地址都由程序员直接指定
编译或汇编时间	程序中包含符号地址引用，并且这些符号地址引用由编译器或汇编者转换为实际的物理地址
装载时间	编译器或汇编器产生相对地址，装载器在装载程序时将相对地址转换为绝对地址
运行时间	装载程序保留相对地址，它们由处理器硬件动态转换为绝对地址

b）链接器

链接时间	功 能
编程时间	不允许引用外部的程序或数据，程序员必须把所有的子程序所引用的源代码输入程序中
编译或汇编时间	汇编器必须获取每个被调用的子例程的源代码，并与调用它们的代码一起汇编为一个单元
创建装载模块	所有的目标模块都使用相对地址进行汇编，这些模块都链接在一起，并且所有的引用都以相对于最终装载模块的起始地址进行重定位
装载时间	外部引用只有在装载模块载入到内存时才进行解析。在那时，被引用的动态链接模块被添加到装载模块，并且整个程序包都载入到主存或虚拟内存
运行时间	外部引用只有在处理器执行到一个外部调用时才进行解析，在那时，进程被中断，并且所需要的模块将被链接到调用程序

可重定位装载 在装载之前，将内存引用和特定地址绑定的缺点是，将导致装载模块只能被装入内存中的一个指定区域。然而，当内存中有很多程序时，不应该事先决定一个特定的模块应放入内存中的哪个区域，更好的做法是在装载的时候来决定。因此，我们需要使装载模块能放入到内存中的任何位置。

为了满足这个新的要求，汇编器或编译器不产生实际的内存地址（绝对地址），而产生相对于某起已知点的地址，例如程序的起点。这一技术如图 B-11c 所示。装载模块的起点分配相对地址 0，在此模块内，所有的内存引用都表述为相对于模块起点的地址。

当所有的内存引用都表述为相对地址格式，那么装载器将模块装入到预期位置将变成一件易事。如果模块要装入到以 x 为起始地址的区域，那装载器给每一个内存引用简单地加上 x。为了协助这个任务，装

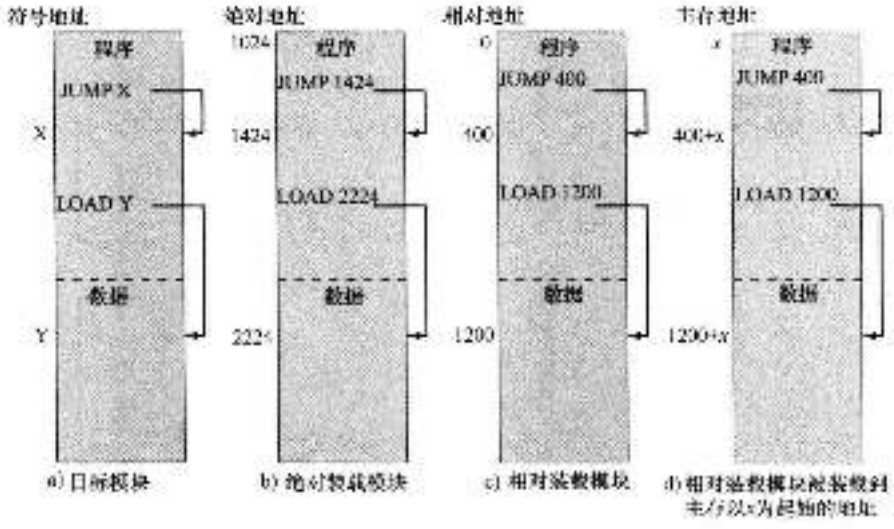


图 B-11 绝对与可重定位的装载模块

载模块必须包含一些信息，这些信息告诉装载器该模块的地址引用在哪儿，以及它们应该如何被解析（通常来说，地址引用采用相对于程序起点的相对地址，但是也可能相对于程序中的其他点，如当前位置）。这些信息由编译器或汇编器提供，并且通常涉及重定位地址目录（relocation directory）。

动态运行时装载 可重定位装载器已很普遍，并且比绝对地址装载器优势更明显。但是，在多道程序运行环境中，即使该系统不依靠虚拟内存，可重定位装载方式仍然不足。我们看到过，之所以需要把进程映像换进换出内存，目的是最大程度地利用处理器。为了最大限度地利用内存，我们希望能将进程映像在不同时间放入内存中的不同位置。因此，程序一旦载入，可能会换出到磁盘，然后换入内存中的不同位置。如果内存引用在刚开始装载时就和绝对地址绑定，以上操作将不可能做到。

另一种替代方法就是延迟计算绝对地址，直到程序运行时真正需要它才做。为了达到这个目的，装载模块把所有内存引用以相对地址的形式装入内存，直到指令真正执行时才计算绝对地址。为了确保这种机制不会降低性能，它必须由专用的硬件而不是软件来处理。在本书第 8 章中就提到了这种硬件。

动态地址计算机制提供了完全的灵活性。一个程序可装载到内存中的任意区域。程序的执行可以中断，程序可以换出内存，后面又可以换入到内存中的不同位置。

B.3.3 链接

链接器的功能是收集目标模块并产生一个装载模块，这个装载模块由程序模块和数据模块集合组成。程序运行时，装载器将装载这个集成的装载模块。在每一个目标模块中，都可能有引用到其他模块的地址。每一个这样的引用都会在一个未链接模块中以符号形式表示。每一个模块间的引用都必须被解析，即把符号地址转换为在最终装载模块内的地址。例如，在图 B-12a 的模块 A 中含有一个调用模块 B 的程序。当这些模块在装载模块结合时，这个模块 B 的符号引用将转换为到模块 B 起点地址的特定引用。

链接编辑器 地址链接的本质取决于所创建的装载模块的类型以及链接什么时候发生（见表 B-3b）。就像在通常情况下那样，如果想要可重定位载入模块，那么链接通常是按如下方式进行：每一个被编译或汇编的目标模块在创建时，其中的地址引用都以相对于目标模块起始地址的形式表示。所有的这些模块都一起放在一个可重定位载入模块中，所有的引用都转换为相对于装载模块起始地址的引用。这个装载模块即可用作可重定位装载器或动态运行时装载器的输入。

能产生可重定位载入模块的链接器通常被称为是链接编辑器（linkage editor）。图 B-12 显示了链接编辑器的功能。

动态链接器 了解了装载过程，可以发现我们可以推迟其中链接功能。动态链接（dynamic linking）用来表述在装载模块创建之前，延迟链接一些外部模块的技术。采用动态链接的话，装载模块会包含未解析

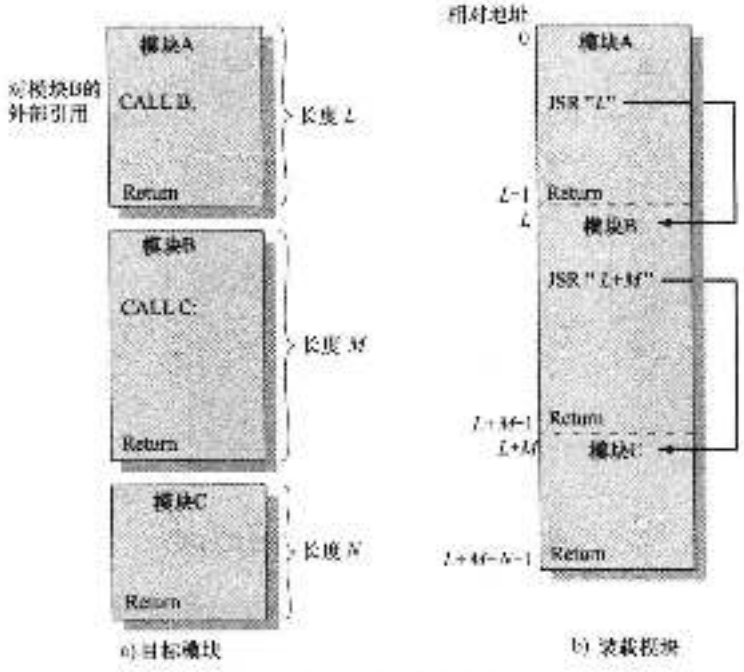


图 B-12 链接动态模块

的其他程序的引用。这些引用将在装载时或运行时进行解析。

若是**装载时动态链接**（它和图 B-9 中上半部的动态链接库有关），它的步骤如下：将被载入的装载模块（应用模块，application module）读入内存。任何到外部模块（目标模块，target module）的引用将引起装载器寻找并载入目标模块，且把引用转化为相对于应用模块起始地址的相对地址。这种方法与所谓的静态链接相比，有下述若干优势：

- 这使得合并修改过的或升级的目标模块很容易实现，而这一目标模块可能是操作系统的常用功能或其他某个调用了程序。如果采用静态链接，一旦该目标模块稍有变化将会导致整个应用模块需要重新链接。这样做不但低效，而且在某些情况下根本不能实现。例如，在个人电脑领域，绝大多数商业软件是以装载模块形式发行，源代码和目标版本却不发行。
- 动态链接文件中的目标代码为自动代码共享做了准备。操作系统能判断出超过一个应用在使用相同的目标代码。因为操作系统装载并链接这些代码，这样一来，它可以利用新知信息，只载入一份目标代码的拷贝来链接两个应用，而不是为每个应用程序各装载一个拷贝。
- 这使得独立的软件开发者能很轻松地扩展广为使用的操作系统的功能，例如 Linux。开发者能以动态链接模块的形式提供一项新功能，这项功能可能对许多应用非常有用。

对运行时动态链接（与图 B-9 中下半部的动态链接库有关）来说，有些链接推迟到执行时才进行。到目标模块的外部引用在装载程序保留不变，不做解析。当程序执行到对目标模块的调用时，由于被调用的模块缺失，操作系统将发到这个缺失模块，载入它，并链接到调用模块。通常这种模块是典型的共享模块。在 Windows 操作系统中，这种模块称为动态链接库（Dynamic-Linked Library，DLL）。因此，如果一个进程使用了共享的动态链接模块，那么一个新进程可以简单地链接到已存在的载入模块。

使用 DLL（动态链接库）可能会导致动态链接库灾难（DLL Hell）。动态链接库灾难发生在当两个或更多的进程共享一个 DLL 模块，而每个进程却期望使用该共享 DLL 模块的不同版本。例如，一个系统或应用程序可能被重装，从而带入老版本的 DLL 文件。

我们已经看到动态装载允许一个完整的装载模块在内存中移动到其他位置。但是，模块的内部结构是静态的，无论程序执行了多少次，都不会改变。然而，在某些情况下，没有办法在执行之前确定哪些目标模块会被用到。这种情形的一个典型例子是事务处理应用（transaction-processing application）。像航空订票

系统或银行应用系统。事务本身的特性决定了那些程序模块要被调用，这些模块会在适当的时机装载并链接到主程序中。使用运行时动态链接器的优点在于，不用在一开始就为所有程序单元分配存储空间，除非该单元被引用过。运行时动态链接也支持分段（segmentation）程序系统。

另一个改进是，应用程序不需要知道那些可能被调用的模块的名称或人口地址。例如，一个报表程序可能被设计为能使用不同的绘图仪，每一种绘图仪都有不同的驱动程序来驱动。该报表程序可以通过系统中某个独立程序或绘图仪来获得绘图仪的名称，或者从一个配置文件中查找到绘图仪的名称。这样就允许用户安装新的绘图仪驱动程序，即使这些绘图仪在该报表程序设计时甚至都还未出现。

B.4 推荐的读物和 Web 站点

[SALO93] 包含了汇编器与装载器的设计与实现。

链接和装载这两个主题在很多程序开发、计算机体系结构，以及操作系统的书籍上都有提到。一个特别详细的论述是 [BECK97]，[CLAR98] 也有很好的论述。在 [LEVI00] 中，使用很多操作系统的实例，对这个主题进行了透彻且具有实用性的论述。

[BART03] 对学习 x86 处理器的汇编语言做出了精彩的论述，很适合自学。[CART06] 涵盖了 x86 汇编语言。对于严谨的 x86 汇编语言程序员来说，[FOG08a] 很有用。[KNAG04] 透彻地论述了 ARM 汇编语言。

BART03 Bartlett, J. *Programming from the Ground Up*. 2003. Available at this book's Web site.

BECK97 Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1997.

CART06 Carter, P. *PC Assembly Language*. July 23, 2006. Available at this book's Web site.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

FOG08a Fog, A. *Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms*. Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

KNAG04 Knaggs, P., and Welsh, S. *ARM: Assembly Language Programming*. Bournemouth University School of Design, Engineering & Computing, August 31, 2004. Available at this book's Web site.

LEVI00 Levine, J. *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 2000.

SALO93 Salomon, D. *Assemblers and Loaders*. Ellis Horwood Ltd, 1993. Available at this book's Web site.

推荐的 Web 站点

- Gavin 的 80x86 汇编指南（Gavin's Guide to 80x86 Assembly）：一个关于 x86 汇编语言精彩简明的概述。
- 汇编语言编程艺术（The Art of Assembly Language Programming）：它提供多达 1500 页，内容无所不包的在线文档。这对学习汇编语言的学生来说是矣。

B.5 关键词、思考题和习题

关键词

assembler：汇编器
assembly language：汇编语言
comment：注释
directive：指导语句
dynamic linker：动态链接器
instruction：指令
label：标号
linkage editor：链接编辑器
linking：链接
load-time dynamic linking：装载时动态链接
loading：装载
macro：宏

mnemonic：助记符
one-pass assembler：单趟汇编器
operand：操作数
relocation：重定位
run-time dynamic linking：运行时动态载入
two-pass assembler：两趟汇编器

思考题

B.1 请给出为什么要学习汇编语言程序设计的理由。
B.2 什么是汇编语言？
B.3 与高级程序语言相比，汇编语言有哪些缺点？
B.4 与高级程序语言相比，汇编语言又有哪些优点？
B.5 汇编语言语句的典型组成部分有哪些？
B.6 列举并定义四种不同的汇编语言语句。
B.7 单趟汇编器与两趟汇编器有什么区别？

习题

B.1 Core War（核心大战）是20世纪80年代早期走向大众的编程游戏，曾流行15年之久。Core War由四个主要部分组成：8000个地址组成的内存空间，一个简化的汇编语言Redcode，执行程序MARS（它是Memory Array Redcode Simulator的首字母缩写）以及对战程序。两个对战程序在进入内存时随机选择位置，任一程序都不知道另一程序在哪里。MARS以简单的分时形式执行程序，两个程序轮流执行：一个程序在执行完一条指令后，另一程序也执行一条指令，依次轮流执行。对战程序在执行周期中做些什么操作完全由程序员决定。对战程序的目标就是通过摧毁另一程序的指令来打败对方。在这个问题以及接下来的几个中，我们使用一个叫CodeBlue的更为简单的语言，来阐述一些Core War的概念。CodeBlue仅有5个汇编语句，使用三种寻址方式（表B-4）。地址空间首尾相接，即最后一个单元的地址加1，则回到了地址空间的第一个单元。例如，ADD #4，6把相对地址为6的内存单元中的值加4并存储在地址为6的内存单元中；JUMP @5将执行转移到当前位置后5个地址单元所在的指令。

表 B-4 CodeBlue 汇编语言

a) 指令集

格　式	意　义
DATA <value>	在当前地址设置值<value>
COPY A, B	将数据从A拷贝到B
ADD A, B	A加B，结果放在B
JUMP A	转移去执行A
JUMPZ A, B	如果B=0，转去执行A

b) 寻址方式

寻址方式	格　式	意　义
立即寻址	# 后面跟值	这是立即寻址，操作数在指令中
相对寻址	值	这个值表示了一个从当前地址的开始的偏移量，它包含操作数
间接寻址	@ 后面跟值	这个值表明了一个从当前地址开始的移动值，这个含有相对地址的地址中，又含有操作数

(a) 程序Imp就是一条指令 COPY 0,1 这条指令会做什么？
(b) 程序Dwarf是一个指令序列：

```
ADD  #4,  3
COPY  2,  @2
JUMP  -2
DATA  0
```

那么这个程序会做什么？
(c) 使用符号重写Dwarf，使它更像典型的汇编语言程序。

B.2 如果Imp和Dwarf相互对战，会发生什么事？

B.3　使用 CodeBlue 写一个“地核式蠢虫”程序，把所有的内存单元清零（可能的例外是程序本身占据的内存单元）。

B.4　下面这个程序与 Imp 对战会怎么样？

```
Loop   COPY #0,    1
       JUMP   -1
```

提示：在两个程序对战时，它们的指令交替执行。

B.5　(a) 在执行完以下指令后，状态标志 C 的值是什么？

```
mov al, 3
add al, 4
```

(b) 在执行完以下指令后，状态标志 C 的值又会是什么？

```
mov al, 3
sub al, 4
```

B.6　思考下面这个 NASM 指令：

```
cmp vleft, vright
```

对于有符号整数，有三个相关的状态标志位：ZF、SF、OF。如果 vleft = vright，那么 ZF 被置位。如果 vleft > vright，那么 ZF 被清零，而 SF = OF。如果 vleft < vright，那么 ZF 被清零，而 SF≠OF。为什么当 vleft > vright 时，SF = OF？

B.7　思考下列 NASM 代码片段：

```
mov  al, 0
cmp  al, al
je   next
```

请只用一条指令编写一个与之等价的程序。

B.8　思考下面的 C 程序代码：

```
/* 一个简单的 C 程序，求一组整数的平均值 */
main()
{int ave;
 int i1 = 20;
 int i2 = 19;
 int i3 = 82;
 ave = (i1 + i2 + i3) /3;
}
```

请用 NASM 编写一个与之等价的汇编语言程序。

B.9　思考下列 C 语言代码片段：

```
if (EAX == 0) EDX = 1;
else EBX = 2;
```

请编写与之等价的 NASM 代码。

B.10　初始化数据指导语句可用于初始化多个地址的内容。例如：

```
db 0x55, 0x56, 0x57
```

保留三个字节，并初始化它们的值。NASM 支持特殊标记符 $，用于涉及当前汇编位置的计算。当 $ 出现在表达式起始时，它表示当前的汇编位置。根据以上叙述，考虑下面的指导语句序列：

```
message db 'hello, world'
msglen equ $ -message
```

那么符号 msglen 被赋予的值会是多少？

B.11　假设有 3 个符号变量 V1、V2、V3，它们都存储整型值。编写一段 NASM 代码，把这三个变量中的最小值移到整型变量 ax 中。只准使用指令 mov、cmp、jbe。

B.12　请描述下面这条指令的执行效果：cmp eax, 1

假设前一条指令已经更新了 eax 中的内容。

B.13　xchg 指令用于交换两个寄存器中的内容。假设 x86 指令集不支持这条指令。

(a) 仅使用指令 push 和 pop 来实现指令 xchg ax, bx。

(b) 仅使用指令 xor（不使用其他的寄存器）来实现 xchg ax, bx。

B.14 在下列程序中，假设 a, b, x, y 是主存地址的符号。那该程序会做什么？你可以用 C 语言编写等价程序。

```
    mov  eax, a
    mov  ebx, b
    xor  eax, x
    xor  ebx, y
    or   eax, ebx
    jnz  L2;
L1 :              ; 一个指令序列(具体内容略)…
    jmp  L3;
L2 :              ; 另一个指令序列(具体内容略)…
L3 :
```

B.15 B.1 节中有一个求两整数最大公约数的 C 程序。

(a) 用文字描述欧几里得算法并解释这个程序如何使用欧几里得算法求最大公约数。

(b) 给图 B-3a 中的汇编程序添加注释来说明它和 C 程序有同样的功能。

(c) 给图 B-3b 中的程序也添加注释。

B.16 (a) 两趟汇编器可以处理未定义符号，因此指令可以把未定义符号用作操作数。但这对指导语句来说并不总是正确。例如 EQU 指导语句就不能使用未定义符号。指导语句 'A EQU B+1' 在 B 已定义时可以很容易执行，但如果 B 是未定义符号，那就不行。请问这是为什么？

(b) 为汇编器设计一个方法解决这个限制，使得汇编程序中任何语句都可以使用未定义符号。

B.17 思考下列形式的符号指导语句 MAX：

符号 MAX 表达式列表

其中“符号”是必需的，并且将被赋值为指导语句的操作数中各个表达式结果的最大值。例如：

MBIGLER MAX A, B, C ；其中 A, B, C 都是已定义的符号

那么汇编器会如何执行 MAX 指导语句的？在第几趟扫描中？

absolute address（绝对地址） 计算机语言中，用来标识一个存储单元或一个设备的地址，且无需其他中间访问。

accumulator（累加器） CPU 中的寄存器名（AC），用来表示单地址指令格式中被隐含的某一个操作数。

address bus（地址总线） 系统总线的一部分，用于地址的传输。通常，该地址识别一个主存单元或一个 I/O 设备。

address space（地址空间） 能被访问的主存或 I/O 设备的地址范围。

arithmetic and logic unit（ALU，算术逻辑单元） 计算机中执行算术运算、逻辑运算和关系运算的部件。

ASCII（ASCII 码） 美国信息交换标准代码，是一种 7 位二进制编码，用来表示数字、字母和一些可以输出的符号；也包括一些不能输出或显示，但说明一些控制功能的控制符。

assembly language（汇编语言） 一种面向计算机的语言，其指令通常与计算机的机器指令一一对应，且提供了一些便利，如宏指令的使用。同义于 computer-dependent language。

associative memory（相联存储器） 一种依靠其内容（或一部分内容），而不是依靠其名称或其位置来识别存储单元的存储器。

asynchronous timing（异步时序） 总线上一个事件的发生跟随且依赖于其前面的事件的一种技术。

autoindexing（自动变址） 变址寻址的一种形式，其变址寄存器随着每次存储器的访问而自动地增加或减少。

base（基） 常用于科学论文中表示数据，其值等于该基的指数次方乘以尾数（例如，表达式 $2.7'10^2=270$ 中的数字 10 即为基）。

base address（基址） 计算机程序执行时用来计算地址的一种参考值。

binary operator（二元运算符） 表示两个且仅仅两个操作数运算的一种运算符。

bit（位） 在纯二进制数字系统中，每一位只能为 0 或为 1。

block multiplexor channel（块多路通道） 一种交叉传送数据块的多路通道。参考 byte multiplexor channel，与 selector channel 进行比较。

branch prediction（分支预测） 处理器在程序分支执行前预测其结果的一种机制。

buffer（缓冲器） 一种存储装置，当数据从一个设备传送到另一个设备时，用来弥补数据流的速度差别或事件发生的时间差别。

bus（总线） 一种分时通信的通路，由一根或一组线组成。在一些计算机系统中，CPU、存储器和 I/O 设备由公共总线相连。由于这些线由所有部件共享，因此在任何时刻只能有一个部件能成功传送信息。

bus arbitration（总线仲裁） 决定那个竞争总线的设备被允许占用总线的过程。

bus master（总线主控） 占用总线、具有初始化和控制通信的设备。

byte（字节） 8 位二进制序列，也称为 8 位位组。

byte multiplexor channel（字节多路通道） 一种交叉传送数据字节的多路通道。参考 block multiplexor channel，与 selector channel 进行比较。

cache（高速缓存） 一种相对小但快速的存储器，置于主大、更慢的存储器与访问更大存储器的逻辑之间。它存放最近被访问的数据，以加速这些数据的后继访问。

cache coherence protocol（cache 一致性协议） 在多个 cache 之间保持数据有效性的一种机制，以便每次访问都将得到主存数据的最近版本。

cache line（cache 行） 与 cache 标识有关的一个数据块，也是 cache 与主存之间的传送单位。

cache memory（cache 存储器） 一种特殊的缓冲存储器，比主存小而快，用来保存主存中可能很将被处理器需要的指令和数据的一个副本，并且是从主存中自动获取信息。

CD-ROM 只读光盘，用来存放计算机数据的一种不可擦写磁盘。标准系统用 12cm 磁盘，能存放多于

550MB 的信息。

central processing unit（CPU，中央处理器） 计算机中取指令和执行指令的部件，它包括算术逻辑单元（ALU）、控制部件和寄存器组，常常简称为处理器。

cluster（集群） 一组互连的计算机，全体计算机作为一个统一的计算资源一起工作，能产生是一台计算机的幻象。术语“全体计算机”意味着一个系统可以远离集群而独立运行。

combinational circuit（组合电路） 一种逻辑设备，对于任何给定的状态，其输出值只依赖于输入值。组合电路是时序电路的一种特殊形式，它没有存储功能；同义于 combinatorial circuit。

compact disk（CD，光盘） 一种不可擦写的磁盘，用来存放数字音频信息。

computer architecture（计算机体系结构） 程序员所能看见得一些系统属性，或者说，能直接影响程序逻辑执行的一些属性。通常，系统的体系结构属性包括指令集、用来表示各种数据类型的二进制位（例如，数字、字符）、I/O 机制、存储器寻址技术等。

computer instruction（计算机指令） 一条为计算机所设计的、能被计算机中的处理单元识别的指令。同义于 machine instruction。

computer instruction set（计算机指令集） 计算机指令的一整套操作集合，以及对其操作数类型的描述。同义于 machine instruction set。

computer organization（计算机组成，计算机组织） 为实现系统结构规范所涉及的操作部件及其互连一组成属性包括一些对程序员透明的硬件细节，如控制信号、计算机与外设之间的接口，使用的存储器技术等。

conditional jump（条件转移） 仅当条件指令被执行且指定的条件满足时，跳转才发生。与 unconditional jump 进行比较。

condition code（条件码） 反映前一个操作（如算术运算）结果的编码。CPU 可以包含一个或多个条件码，条件码可以分开地存放在 CPU 内部，也可以作为更大的控制寄存器的一部分。也称为“标志”。

control bus（控制总线） 系统总线中用来传输控制信号的部分。

control registers（控制寄存器） 用来控制 CPU 操作的 CPU 寄存器，该寄存器的大部分是用户不可见的。

control storage（控制存储器） 存储器的一部分，用来存放微代码。

control unit（控制单元） CPU 的一部分，控制 CPU 的操作，包括 ALU 操作、CPU 内部的数据传送、数据交换和跨外部界面（例如，系统总线）的控制信号。

daisy chain（菊花链） 一种设备互连方法，通过串行地连接中断源来决定中断优先级。

data bus（数据总线） 系统总线中用来传输数据的部分。

data communication（数据通信） 设备之间的数据传输。该术语通常将 I/O 排斥在外。

decoder（译码器） 一种包含多个输入线和多个输出线的设备，其任何输入线都可以携带信号，但其输出线中最多只能一根携带信号，在输出与输入信号的组合之间有一一对应关系。

demand paging（请求页面） 需要时，一个页面从辅助存储器到实际存储器的传送。

direct access（直接访问） 以独立于数据的相对位置次序，依靠指出数据的物理位置的地址，从存储器设备中获取数据或输入数据到存储器设备中的能力。

direct address（直接地址） 指示作为操作数的数据项在存储器中位置的地址。同义于 one-level address。

direct memory access（DMA，直接存储器存取） 一种 I/O 形式，它由一种特殊的模块（称为 DMA 模块）来控制主存与 I/O 模块之间的数据交换。CPU 发送一个需要传送数据块的请求给 DMA 模块，然后一直到整个数据块传送完成后才被中断。

disabled interrupt（不允许中断） 通常是由 CPU 创造的一个环境，在此期间，CPU 将忽略指定类型的中断请求信号。

diskette（磁盘） 一种包装在保护套中的软件磁盘。同义于 flexible disk。

disk pack（磁盘组套） 一组可从磁盘驱动器中整体卸载的磁盘，该组磁盘装在一个盒套中，使用时要将磁盘组从盒套中取出。

disk stripping（磁盘条带） 一种磁盘阵列映射方法，其中逻辑上连续的数据块，或称条带（strip），被循环地映射到连续的磁盘号上。而映射到各个磁盘号的一组连续逻辑条带称为一条（stripe）。

dynamic RAM（动态 RAM） 一种使用电容来实现其存储元件的 RAM。除非进行周期性刷新，否则，动态 RAM 将逐渐丢失其数据。

emulation（模拟） 一个系统对另一个系统的全部或部分所进行的模仿（主要通过硬件），即模仿系统像被模仿系统一样，接受相同的数据、执行相同的程序和实现相同的结果。

enabled interrupt（允许中断） 通常是由 CPU 创造的一个环境，在此期间，CPU 将响应指定类型的中断请求信号。

erasable optical disk（可擦写光盘） 一种使用光学技术、能容易擦除和重写的磁盘。常用的有 3.25 英寸和 5.25 英寸的磁盘，典型的容量为 650MB。

error-correcting code（纠错码） 每个符号或信号都符合指定的一些结构规则的一种编码，如果背离了这些规则，则表明存在错误，并且可以自动纠正部分或全部错误。

error-detecting code（错误检测码） 每个符号或信号都符合指定的一些结构规则的一种编码，如果背离了这些规则，则表明存在错误。

execute cycle（执行周期） 指令周期的一部分，在此期间，CPU 执行指令操作码规定的操作。

fetch cycle（取指周期） 指令周期的一部分，在此期间，CPU 从存储器中取得将要执行的指令。

firmware（固件） 存放在只读存储器中的程序代码。

fixed-point representation system（定点表示系统） 一种小数表示系统，其中小数点隐含地固定在数字序列中某些由达成一致的约定所规定的位置上。

flip-flop（触发器） 一种具有滞后元件、在给定时刻能处于两种稳态之一的电路或设备。同义于 bistable circuit、toggle。

floating-point representation system（浮点表示系统） 一种计数系统，所表示的实数是下述两部分的积。一个是定点部分，即其中一个数字，另一个是通过计算隐含的浮点数基的幂方所得到的值，幂方次数就是第一个数字所表示的浮点数的阶（exponent）值。

G 缩写，意味着 2^{30}。

gate（门） 一种产生输出信号的电路，其输出信号为输入信号的简单布尔运算。

general-purpose register（通用寄存器） 通常为具有明确地址的寄存器。在一组寄存器中，可以用于不同的目的。例如，作为累加器、索引寄存器或数据的特殊处理者。

global variable（全局变量） 程序中，由一部分定义，并且至少能被另一部分使用的变量。

high-performance computing（HPC，高性能计算） 涉及超级计算机及其软件的研究领域，重点是科学计算，包括向量的大量使用、矩阵计算和并行算法。

immediate address（立即数寻址） 地址部分的内容，它包含操作数的值，而不是地址。同义于 zero-level address。

indexed address（变址寻址） 在指令执行前或执行过程中，根据变址寄存器内容而修改的地址。

indexing（变址） 一种利用变址寄存器进行地址修改的技术。

index register（变址寄存器） 在计算机指令的执行期间，其内容能用来更改操作数地址的寄存器，它也可以用作计数器。变址寄存器可以用来控制循环的执行、控制数组的使用，作为查找表的引用或指针。

indirect address（间接寻址） 一个包含地址的存储单元的地址。

input-output（I/O，输入/输出） 属于输入、输出或二者。涉及计算机与其直接附带的外设之间的数据传送。

instruction address register（指令地址寄存器） 一种特殊用途的寄存器，用来存放即将执行的下一条指令的地址。

instruction cycle（指令周期） CPU 执行一条指令的过程。

instruction format（指令格式） 计算机指令作为一序列二进制位的布局。指令格式将指令分为不同的域，以对应于指令的各组成元素（例如，操作码、操作数）。

instruction issue（指令流出） 在处理器的功能单元中，初始化指令执行的过程，它发生在指令从流水线的译码段移动到第一执行段的时候。

instruction register（指令寄存器） 用来存放指令以便译码的寄存器。

integrated circuit（IC，集成电路） 一个小的固体金属（如硅）片，其上蚀刻或印刷有大量的电子器件和它们的互连。

interrupt（中断） 一个如同计算机程序执行一样的处理过程的中止，它由外部事件引起和执行，被中止的处理过程是能够恢复的。同义于 interruption。

interrupt cycle（中断周期） 指令周期的一部分，在此期间 CPU 检查中断。如果存在一个未处理的允许中断，则 CPU 保存现行程序的状态，并开始执行中断服务程序。

interrupt-driven I/O（中断驱动式 I/O） 一种 I/O 形式。CPU 发出 I/O 命令后，继续执行其后续的指令，直到 I/O 模块完成它的任务后才中断 CPU。

I/O channel（I/O 通道） 一种相对复杂的 I/O 模块，它将 CPU 从 I/O 操作的细节中解放出来。I/O 通道将执行来自主存的一系列 I/O 命令，而无需 CPU 的干预。

I/O controller（I/O 控制器） 一种相对简单的 I/O 模块，它需要来自 CPU 或 I/O 通道的详细控制。同义于 device controller。

I/O module（I/O 模块） 计算机的主要部件类型之一，它负责一台或多台外部设备（外设）的控制、外设与主存或外设与 CPU 的寄存器之间的数据交换。

I/O processor（I/O 处理器） 自带处理器的 I/O 模块，能执行它自己的特殊 I/O 指令，甚至在某些情况下能执行通用的机器指令。

isolated I/O（分立的 I/O） 一种对 I/O 模块和外部设备进行编址的方法，其 I/O 地址空间与主存地址空间分开处理，需要专用的 I/O 机器指令。与 memory-mapped I/O 进行比较。

K 缩写，意味着 $2^{10}=1024$，因此，2Kb = 2048 位。

local variable（局部变量） 仅仅在计算机程序的一个特殊部分进行定义和使用的变量。

locality of reference（访问的局部性） 在一个短的时期内，处理器重复访问同一局部存储区域的趋势。

M 缩写，意味着 $2^{20}=1\,048\,576$，因此，2Mb = 2 097 152 位。

magnetic disk（磁盘） 带磁性表面层的扁平圆盘，其一面或双面能存放数据。

magnetic tape（磁带） 带磁性表面层的带子，它利用磁记录来存放数据。

mainframe（大型机） 最初定义为包含大型电脑中的中央处理器或主机机框的术语。在 20 世纪 70 年代早期较小的小型机设计出现之后，传统的大机器被描述为大型计算机或大型机。主机典型的特征是支持大型数据库，有精细的 I/O 硬件，并被用作为主要的数据处理工具。

main memory（主存） 程序可编址的存储器，其中的指令和其他数据可以直接装载寄存器以便后续的执行或处理。

memory address register（MAR，存储器地址寄存器） 处理单元中的寄存器，存放即将访问的存储单元的地址。

memory buffer register（MBR，存储器缓冲寄存器） 存放从主存中读出的数据或将写进存储单元的数据的寄存器。

memory cycle time（存储周期时间） 能够访问存储器的速度的倒数。它是上一次访问请求（读或写）响应到下一次访问请求响应之间的最小时间间隔。

memory-mapped I/O（存储器映射 I/O） 一种对 I/O 模块和外部设备进行编址的方法。单一的地址空间既可用于主存地址，也可用于 I/O 地址。相同的机器指令既可用于存储器的读/写，也可用于 I/O 操作。

microcomputer（微计算机） 处理单元是微处理器的计算机系统。基本的微计算机包括微处理器、存储器和输入/输出设备，它们既可以集成在一片芯片上，也可以在多块芯片上。

microinstruction（微指令） 控制数据流的指令，它在处理器中处于比机器指令更基本的级别。单个机器指令和其他功能可以由微程序来实现。

micro-operation（微操作） 最基本的 CPU 操作，能在一个时钟脉冲内执行。

microprocessor（微处理器） 其元件已经被小型化到一个或很少几个集成电路中的处理器。

microprogram（微程序） 特殊存储器中的一系列微指令，它们能被动态访问以执行各种功能。

microprogrammed CPU（微程序 CPU） 利用微程序设计来实现控制单元的 CPU。

microprogramming language（微程序设计语言） 用于说明微程序的指令集。

multiplexer（多路选择器） 连接多个输入到一个单独的输出的组合电路。任何时候，仅仅选择一个输入通过，作为输出。

multiplexor channel（多路通道） 设计用来同时处理多个 I/O 设备的通道。通过交叉传送数据块，多个 I/O 设备能够同时传输记录。参考 byte multiplexor channel 和 block multiplexor channel。

multiprocessor（多处理器） 具有两个或多个处理器的计算机，各处理器能对主存共同访问。

multiprogramming（多道程序设计） 一种单个处理器能够交叉执行两个或多个程序的处理模式。

multitasking（多道任务） 一种提供并发执行或交叉执行两个或多个计算机任务的处理模式。与 multiprogramming 相同，只是使用了不同的术语。

nonuniform memory access（NUMA）multiprocessor（非均匀存储器访问（NUMA）的多处理器） 一种共享存储器的多处理器，其中，其一给定处理器对存储器中字的访问时间随着存储器字的位置不同而变化。

nonvolatile memory（非易挥发性存储器） 其内容是固定的且不需要持续电源的存储器。

nucleus（原子） 操作系统的一部分，它包含基本的、最常用的功能。通常，原子常驻在主存中。

ones complement representation（1 的补码表示） 用来表示二进制整数。正整数表示为带符号的数值；负整数表示为将等量正整数表示的各位取反。

opcode（操作码） operation code 的缩写。

operand（操作数） 一个实体，其上执行操作。

operating system（操作系统） 控制程序执行，并且提供程序定位、调度、I/O 控制和数据管理等服务的软件。

operation code（操作码） 用来表示计算机操作的代码，常缩写为 opcode。

orthogonality（正交性） 两个变量或实体相互独立的原则。在指令集知识点中，该术语常用来说明指令中的某些元素（寻址方式、操作数个数、操作数长度）是独立于（不决定于）操作码的。

page（页面） 在虚拟存储器系统中，具有虚拟地址的固定大小的数据块，它是物理存储器与辅助存储器之间数据传输的基本单位。

page fault（页缺失） 发生在包含访问字的页面不在主存中时，它将引起一个中断，并需要操作系统将需要的页面调进主存。

page frame（页帧） 主存中用来保存一个页面的区域。

parity bit（奇偶位） 添加在一组二进制数字中使整个数据的和总是为偶（偶校验）或总是为奇（奇校验）。

peripheral equipment（外部设备） 在计算机系统中，相对于特殊的处理单元来说，给处理单元提供外部通信的任何设备。同义于 peripheral device。

pipeline（流水线） 一种处理器组织，其中的处理器包含多个段，允许多条指令同时执行。

predicated execution（预测执行） 一种支持个别指令条件执行的机制，它使预测性地执行分支指令的两个不同分支且保留分支指令最后的执行结果成为可能。

process（进程） 执行中的程序。进程的控制与调度由操作系统完成。

process control block（进程控制模块） 操作系统中进程的表现，它是包含进程特征和状态信息的数据结构。

processor（处理器） 计算机中解释并执行指令的功能单元。处理器至少包含有指令控制单元和算术单元。

processor cycle time（处理器周期时间） 最短的定义明确的 CPU 操作所需要的时间。它是测量所有 CPU 行为的基本数据单位。同义于 machine cycle time。

program counter（程序计数器） 指令地址寄存器。

programmable logic array（PLA，可编程逻辑阵列） 门之间的互连能被编程来实现特定逻辑功能的门阵列。

programmable read-only memory（PROM，可编程只读存储器） 其内容仅仅能被设置一次的半导体存储器。写过程由电实现，且可以在原始芯片制作完成后由用户实现。

programmed I/O（编程式I/O） 一种I/O形式，此时，CPU给I/O模块发送一条I/O命令，并且必须等待该I/O操作完成之后才能进行后续工作。

program status word（PSW，程序状态字） 存储器中的一个区域，用来指明指令执行的顺序，保持和指示计算机系统的状态。同义于processor status word。

random-access memory（RAM，随机访问存储器） 每个地址单元都有唯一的编址机制的存储器。访问一给定单元的时间与前一次访问的次序无关。

read-only memory（ROM，只读存储器） 除非破坏存储单元以外，其内容不能改变的半导体存储器。也是非易挥发性存储器。

redundant array of independent disks（RAID，独立冗余磁盘阵列） 一种磁盘阵列，其中一部分物理存储空间用来存放关于用户数据的冗余信息，而用户数据则存放在存储空间的其余位置。当一个阵列磁盘或访问它的路径遭破坏时，冗余信息能够再生用户数据。

registers（寄存器） CPU内部的高速存储器。一些寄存器是用户可见的，即程序员可以通过机器指令使用它们；而另一些只能由CPU使用，用于控制机器。

scalar（标量） 具有单个数值的量。

secondary memory（辅助存储器） 处于计算机系统本身之外的存储器，即处理器不能直接访问它，必须首先将其拷贝到主存。例如，磁盘和磁带。

selector channel（选择通道） 一种设计在任何时刻，只能与一个I/O设备进行操作的I/O通道。一旦选择了某个I/O设备，则其全部记录以字节为单位传输完。对比于block multiplexor channel和multiplexor channel。

semiconductor（半导体） 一种固态晶体物质（硅、锗），它们的导电性能介于绝缘体和导体之间，用来制成晶体管和固体元件。

sequential circuit（时序电路） 一种数字逻辑电路，其输出依赖于电路输入和电路的状态，因此，时序电路具有存储器的功能。

sign-magnitude representation（符号-幅值表示） 用来表示二进制整数。在一个N位字中，最左的位是符号（0表示正，1表示负），其余$N-1$位表示这个数的幅值。

solid-state component（固体元件） 一种元件，其操作依赖于固体（如晶体二极管、铁氧体磁心）中电现象或磁现象的控制。

speculative execution（猜测执行） 指令沿着分支的一条路径的执行，如果后来这个分支没有按猜测的情况出现，则猜测执行的结果应该被丢弃。

stack（栈） 一种有序序列，其项目的添加和删除都从序列的同一固定端进行，该固定端被称为栈顶。也就是说，下一个将添加到序列的项目被置于栈顶，而下一个将被移出序列的项目是处于序列中最短时间的那个项目。这种方法被称为“后进先出”。

static RAM（静态RAM） 由触发器实现其单元的RAM存储器。只要供电，静态RAM将一直保存其数据，不需要周期性刷新。

superpipelined processor（超级流水线处理器） 一种处理器，其指令流水线被设计成包含很多非常小的段，以至于在一个时钟周期内能够执行多个流水线段，也就是，大量的指令可以同时处于流水线中。

superscalar processor（超标量处理器） 一种处理器设计，它包含多条指令流水线，因此在同一流水线段中能够同时执行多条指令。

symmetric multiprocessing（SMP，对称多处理器） 一种多处理器形式，它允许操作系统运行在任意一个可用的处理器上，或者同时运行在几个可用的处理器上。

synchronous timing（同步时序） 总线上事件的发生由时钟来决定的一种技术。时钟定义了等宽度的时间相，并且事件仅仅发生在时钟相的起始端。

system bus（系统总线） 用来连接计算机的主要部件（CPU、主存、I/O）的总线。

truth table（真值表） 一种描述逻辑功能的表，它列出输入值的所有可能组合，并且说明每个输入组合所对应的输出值。

twos complement representation（2的补码表示） 用来表示二进制整数。正整数的表示与符号幅值表示一

效；负整数表示为与之对应正数的各位取逻辑补码（取反）后加 1，使结果的位模式看起来像一个无符号整数一样。

unary operator（一元操作符） 表示一个或仅仅一个操作数运算的一种操作。

unconditional jump（无条件转移） 当指定它的指令被执行时随时就发生的跳转。

uniprocessing（单机处理） 指令的顺序执行，由单个处理器单元或多处理器系统中的一个独立的处理器单元来完成。

user-visible registers（用户可见寄存器） CPU 中可以由程序员定义的寄存器。指令集格式允许指定一个或多个寄存器作为操作数或操作数地址。

vector（向量） 通常由一组有序标量构成的量。

very long instruction word（VLIW，超长指令字） 定义为包含多个操作的指令的使用。相当于，将多条指令包含在一个单独的字中，特别是，VLIW 由编译器构造，它将可以并行执行的操作安排在同一字中。

virtual storage（虚拟存储器） 用户可以认为它是可编址主存的存储空间。在计算机系统中，虚拟地址被映射成真实地址，虚拟存储器的尺寸受限于计算机系统的编址方式和可用的虚拟存储器容量，而不受实际主存单元数量的限制。

volatile memory（易挥发式存储器） 需要连续的电能来维持其内容的存储器，如果关掉电能，则存放的信息丢失。

word（字） 有序字节或位的集合。在给定的计算机内，它是可以存储、传送或处理信息的正常单位。其规模，如果处理器有一定长指令集，则其指令长等于字长。

ABBREVIATIONS

ACM	Association for Computing Machinery
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers

ABBO04 Abbot, D. *PCI Bus Demystified.* New York: Elsevier, 2004.

ACOS86 Acosta, R.; Kjelstrup, J.; and Torng, H. "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors." *IEEE Transactions on Computers,* September 1986.

ADAM91 Adamek, J. *Foundations of Coding.* New York: Wiley, 1991.

AGAR89 Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Boston: Kluwer Academic Publishers, 1989.

AGER87 Agerwala, T., and Cocke, J. *High Performance Reduced Instruction Set Processors.* Technical Report RC12434 (#55845). Yorktown, NY: IBM Thomas J. Watson Research Center, January 1987.

AMDA67 Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capability." *Proceedings of the AFIPS Conference,* 1967.

ANDE67a Anderson, D.; Sparacio, F.; and Tomasulo, F. "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling." *IBM Journal of Research and Development,* January 1967.

ANDE67b Anderson, S., et al. "The IBM System/360 Model 91: Floating-Point Execution Unit." *IBM Journal of Research and Development,* January 1967. Reprinted in [SWAR90, Volume 1].

ANDE03 Anderson, D. "You Don't Know Jack About Disks." *ACM Queue,* June 2003.

ANDE98 Anderson, D. *FireWire System Architecture.* Reading, MA: Addison-Wesley, 1998.

ANTH08 Anthes, G. "What's Next for the x86?" *ComputerWorld,* June 16, 2008.

ARM08a ARM Limited. *Cortex-A8 Technical Reference Manual.* ARM DDI 0344E, 2008. www.arm.com

ARM08b ARM Limited. *ARM11 MPCore Processor Technical Reference Manual.* ARM DDI 0360E. 2008. www.arm.com

ASH90 Ash, R. *Information Theory.* New York: Dover, 1990.

ATKI96 Atkins, M. "PC Software Performance Tuning." *IEEE Computer,* August 1996.

AZIM92 Azimi, M.; Prasad, B.; and Bhat, K. "Two Level Cache Architectures." *Proceedings COMPCON '92,* February 1992.

BACO94 Bacon, F.; Graham, S.; and Sharp, O." Compiler Transformations for High-Performance Computing." *ACM Computing Surveys,* December 1994.

BAIL93 Bailey, D. "RISC Microprocessors and Scientific Computing." *Proceedings, Supercomputing '93,* 1993.

BART03 Bartlett, J. *Programming from the Ground Up.* 2003. Available at this book's Web site.

BASH81 Bashe, C.; Bucholtz, W.; Hawkins, G.; Ingram, J.; and Rochester, N. "The Architecture of IBM's Early Computers." *IBM Journal of Research and Development,* September 1981.

BASH91 Bashteen, A.; Lui, I.; and Mullan, J. "A Superpipeline Approach to the MIPS Architecture." *Proceedings, COMPCON Spring '91,* February 1991.

BECK97 Beck, L. *System Software.* Reading, MA: Addison-Wesley, 1997.

BELL70 Bell, C.; Cady, R.; McFarland, H.; Delagi, B.; O'Loughlin, J.; and Noonan, R. "A New Architecture for Minicomputers—The DEC PDP-11." *Proceedings, Spring Joint Computer Conference,* 1970.

BELL71 Bell, C., and Newell, A. *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

BELL74 Bell, J.; Casasent, D.; and Bell, C. "An Investigation into Alternative Cache Organizations." *IEEE Transactions on Computers*, April 1974. http://research.microsoft.com/users/GBell/gbvita.htm

BELL78a Bell, C.; Mudge, J.; and McNamara, J. *Computer Engineering: A DEC View of Hardware Systems Design*. Bedford, MA: Digital Press, 1978.

BELL78b Bell, C.; Newell, A.; and Siewiorek, D. "Structural Levels of the PDP-8." In [BELL78a].

BELL78c Bell, C.; Kotok, A.; Hastings, T.; and Hill, R. "The Evolution of the DEC System-10." *Communications of the ACM*, January 1978.

BENH92 Benham, J. "A Geometric Approach to Presenting Computer Representations of Integers." *SIGCSE Bulletin*, December 1992.

BETK97 Betker, M.; Fernando, J.; and Whalen, S. "The History of the Microprocessor." *Bell Labs Technical Journal*, Autumn 1997.

BEZ03 Bez, R.; et al. Introduction to Flash Memory. *Proceedings of the IEEE*, April 2003.

BHAR00 Bharandwaj, J., et al. "The Intel IA-64 Compiler Code Generator." *IEEE Micro*, September/October 2000.

BLAA97 Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

BLAH83 Blahut, R. *Theory and Practice of Error Control Codes*. Reading, MA: Addison-Wesley, 1983.

BOHR98 Bohr, M. "Silicon Trends and Limits for Advanced Microprocessors." *Communications of the ACM*, March 1998.

BOHR03 Bohr, M." High Performance Logic Technology and Reliability Challenges." *International Reliability Physics Symposium*, March 2003. http://www.irps.org/03-41st

BORK03 Borkar, S. "Getting Gigascale Chips: Challenges and Opportunities in Continuing Moore's Law." *ACM Queue*, October 2003.

BORK07 Borkar, S. "Thousand Core Chips—A Technology Perspective." *Proceedings, ACM/IEEE Design Automation Conference*, 2007.

BRAD91a Bradlee, D.; Eggers, S.; and Henry, R. "The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy." *Proceedings, 18th Annual International Symposium on Computer Architecture*, May 1991.

BRAD91b Bradlee, D.; Eggers, S.; and Henry, R. "Integrating Register Allocation and Instruction Scheduling for RISCs." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

BREW97 Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.

BREY09 Brey, B. *The Intel Microprocessors: 8086/8066, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4 and Core2 with 64-bit Extensions*. Upper Saddle River, NJ: Prentice Hall, 2009.

BROW96 Brown, S., and Rose, S. "Architecture of FPGAs and CPLDs: A Tutorial." *IEEE Design and Test of Computers*, Vol. 13, No. 2, 1996.

BURG97 Burger, D., and Austin, T. "The SimpleScalar Tool Set, Version 2.0." *Computer Architecture News*, June 1997.

BURK46 Burks, A.; Goldstine, H.; and von Neumann, J. *Preliminary Discussion of the Logical Design of an Electronic Computer Instrument*. Report prepared for U.S. Army Ordnance Dept., 1946, reprinted in [BELL71].

BUYY99a Buyya, R. *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications*. Upper Saddle River, NJ: Prentice Hall, 1999.

CANT01 Cantin, J., and Hill, H. "Cache Performance for Selected SPEC CPU2000 Benchmarks." *Computer Architecture News*, September 2001.

CART96 Carter, J. *Microprocesser Architecture and Microprogramming*. Upper Saddle River, NJ: Prentice Hall, 1996.

CART06 Carter, P. *PC Assembly Language*, July 23, 2006. Available at this book's Web site.

CATA94 Catanzaro, B. *Multiprocessor System Architectures*. Mountain View, CA: Sunsoft Press, 1994.

CEKL97 Cekleov, M., and Dubois, M. "Virtual-Address Caches, Part 1: Problems and Solutions in Uniprocessors." *IEEE Micro*, September/October 1997.

CHAI82 Chaitin, G. "Register Allocation and Spilling via Graph Coloring." *Proceedings, SIGPLAN Symposium on Compiler Construction*, June 1982.

CHAS00 Chasin, A. "Predication, Speculation, and Modern CPUs." *Dr. Dobb's Journal*, May 2000.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; and Patterson, D. "RAID: High-Performance, Reliable Secondary Storage." *ACM Computing Surveys*, June 1994.

CHEN96 Chen, S., and Towsley, D. "A Performance Evaluation of RAID Architectures." *IEEE Transactions on Computers*, October 1996.

CHOW86 Chow, F.; Himmelstein, M.; Killian, E.; and Weber, L. "Engineering a RISC Compiler System." *Proceedings, COMPCON Spring '86*, March 1986.

CHOW87 Chow, F.; Correll, S.; Himmelstein, M.; Killian, E.; and Weber, L. "How Many Addressing Modes Are Enough?" *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

CHOW90 Chow, F., and Hennessy, J. "The Priority-Based Coloring Approach to Register Allocation." *ACM Transactions on Programming Languages*, October 1990.

CLAR85 Clark, D., and Emer, J. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems*, February 1985.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

CLEM00 Clements, A. "The Undergraduate Curriculum in Computer Architecture." *IEEE Micro*, May/June 2000.

COHE81 Cohen, D. "On Holy Wars and a Plea for Peace." *Computer*, October 1981.

COLW85a Colwell, R.; Hitchcock, C.; Jensen, E.; Brinkley-Sprunt, H.; and Kollar, C. "Computers, Complexity, and Controversy." *Computer*, September 1985.

COLW85b Colwell, R.; Hitchcock, C.; Jensen, E.; and Sprunt, H. "More Controversy About 'Computers, Complexity, and Controversy.'" *Computer*, December 1985.

COME00 Comerford, R. "Magnetic Storage: The Medium that Wouldn't Die." *IEEE Spectrum*, December 2000.

COOK82 Cook, R., and Dande, N. "An Experiment to Improve Operand Addressing." *Proceedings, Symposium on Architecture Support for Programming Languages and Operating Systems*, March 1982.

COON81 Coonen J. "Underflow and Denormalized Numbers." *IEEE Computer*, March 1981.

COUT86 Coutant, D.; Hammond, C.; and Kelley, J. "Compilers for the New Generation of Hewlett-Packard Computers." *Proceedings, COMPCON Spring '86*, March 1986.

CRAG79 Cragon, H. "An Evaluation of Code Space Requirements and Performance of Various Architectures." *Computer Architecture News*, February 1979.

CRAG92 Cragon, H. *Branch Strategy Taxonomy and Performance Models*. Los Alamitos, CA: IEEE Computer Society Press, 1992.

CRAW90 Crawford, J. "The i486 CPU: Executing Instructions in One Clock Cycle." *IEEE Micro*, February 1990.

CRIS97 Crisp, R. "Direct RAMBUS Technology: The New Main Memory Standard." *IEEE Micro*, November/December 1997.

CUPP01 Cuppu, V., et al. "High Performance DRAMS in Workstation Environments." *IEEE Transactions on Computers*, November 2001.

DATT93 Dattatreya, G. "A Systematic Approach to Teaching Binary Arithmetic in a First Course." *IEEE Transactions on Education*, February 1993.

DAVI87 Davidson, J., and Vaughan, R. "The Effect of Instruction Set Complexity on Program Size and Memory Performance." *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

DENN68 Denning, P. "The Working Set Model for Program Behavior." *Communications of the ACM*, May 1968.

DERO87 DeRosa, J., and Levy, H. "An Evaluation of Branch Architectures." *Proceedings, Fourteenth Annual International Symposium on Computer Architecture*, 1987.

DESA05 Desai, D., et al. "BladeCenter System Overview." IBM Journal of Research and Development." November 2005.

DEWA90 Dewar, R., and Smosna, M. *Microprocessors: A Programmer's View.* New York: McGraw-Hill, 1990.

DEWD84 Dewdney, A. "In the Game Called Core War Hostile Programs Engage in a Battle of Bits." *Scientific American,* May 1984.

DIJK63 Dijkstra, E. "Making an ALGOL Translator for the X1." In *Annual Review of Automatic Programming, Volume 4.* Pergamon, 1963.

DOWD98 Dowd, K., and Severance, C. *High Performance Computing.* Sebastopol, CA: O'Reilly, 1998.

DUBE91 Dubey, P., and Flynn, M. "Branch Strategies: Modeling and Optimization." *IEEE Transactions on Computers,* October 1991.

DULO98 Dulong, C. "The IA-64 Architecture at Work." *Computer,* July 1998.

ECKE90 Eckert, R. "Communication Between Computers and Peripheral Devices—An Analogy." *ACM SIGCSE Bulletin,* September 1990.

ELAY85 El-Ayat, K., and Agarwal, R. "The Intel 80386—Architecture and Implementation." *IEEE Micro,* December 1985.

ERCE04 Ercegovac, M., and Lang, T. *Digital Arithmetic.* San Francisco: Morgan Kaufmann, 2004.

EISC07 Eischen, C. "RAID 6 Covers More Bases." *Network World,* April 9, 2007.

EVAN03 Evans, J., and Trimper, G. *Itanium Architecture for Programmers.* Upper Saddle River, NJ: Prentice Hall, 2003.

EVEN00a Even, G., and Paul, W. "On the Design of IEEE Compliant Floating-Point Units." *IEEE Transactions on Computers,* May 2000.

EVEN00b Even, G., and Seidel, P. "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication." *IEEE Transactions on Computers,* July 2000.

EVER98 Evers, M., et al. "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work." *Proceedings, 25th Annual International Symposium on Microarchitecture,* July 1998.

EVER01 Evers, M., and Yeh, T. "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors." *Proceedings of the IEEE,* November 2001.

FARH04 Farhat, H. *Digital Design and Computer Organization.* Boca Raton, FL: CRC Press, 2004.

FARM92 Farmwald, M., and Mooring, D. "A Fast Path to One Memory." *IEEE Spectrum,* October 1992.

FLEM86 Fleming, P., and Wallace, J. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results." *Communications of the ACM,* March 1986.

FLYN72 Flynn, M. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers,* September 1972.

FLYN85 Flynn, M.; Johnson, J.; and Wakefield, S. "On Instruction Sets and Their Formats." *IEEE Transactions on Computers,* March 1985.

FLYN87 Flynn, M.; Mitchell, C.; and Mulder, J. "And Now a Case for More Complex Instruction Sets." *Computer,* September 1987.

FLYN01 Flynn, M., and Oberman, S. *Advanced Computer Arithmetic Design.* New York: Wiley, 2001.

FOG08a Fog, A. *Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms.* Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

FOG08b Fog, A. *The Microarchitecture of Intel and AMD CPUs.* Copenhagen University College of Engineering, 2008. http://www.agner.org/optimize/

FRAI83 Frailey, D. "Word Length of a Computer Architecture: Definitions and Applications." *Computer Architecture News,* June 1983.

FRIE96 Friedman, M. "RAID Keeps Going and Going and . . ." *IEEE Spectrum,* April 1996.

FURB00 Furber, S. *ARM System-On-Chip Architecture.* Reading, MA: Addison-Wesley, 2000.

FURH87 Furht, B., and Milutinovic, V. "A Survey of Microprocessor Architectures for Memory Management." *Computer,* March 1987.

FUTR01 Futral, W. *InfiniBand Architecture: Development and Deployment.* Hillsboro, OR: Intel Press, 2001.

GENU04 Genua, P. *A Cache Primer.* Application Note AN2663. Freescale Semiconductor, Inc., 2004. www.freescale.com/files/32bit/doc/app_note/AN2663.pdf

GHAI98 Ghai, S.; Joyner, J.; and John, L. *Investigating the Effectiveness of a Third Level Cache.* Technical Report TR-980501-01, Laboratory for Computer Architecture, University of Texas at Austin. http://lca.ece.utexas.edu/pubs-by-type.html

GIBB04 Gibbs, W. "A Split at the Core." *Scientific American.* November 2004.

GIFF87 Gifford, D., and Spector, A. "Case Study: IBM's System/360-370 Architecture." *Communications of the ACM,* April 1987.

GOCH06 Gochman, S., et al. "Introduction to Intel Core Duo Processor Architecture." *Intel Technology Journal,* May 2006.

GOLD91 Goldberg, D. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys,* March 1991.

GOOD83 Goodman, J. "Using Cache Memory to Reduce Processor-Memory Bandwidth." *Proceedings, 10th Annual International Symposium on Computer Architecture,* 1983. Reprinted in [HILL00].

GOOD05 Goodacre, J., and Sloss, A. "Parallelism and the ARM Instruction Set Architecture." *Computer,* July 2005.

GREG98 Gregg, J. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets.* New York: Wiley, 1998.

GRIM05 Grimheden, M., and Torngren, M. "What is Embedded Systems and How Should It Be Taught?—Results from a Didactic Analysis." *ACM Transactions on Embedded Computing Systems,* August 2005.

GUST88 Gustafson, J. "Reevaluating Amdahl's Law." *Communications of the ACM,* May 1988.

HALF97 Halfhill, T. "Beyond Pentium II." *Byte,* December 1997.

HAMM97 Hammond, L.; Nayfay, B.; and Olukotun, K. "A Single-Chip Multiprocessor." *Computer,* September 1997.

HAND98 Handy, J. *The Cache Memory Book.* San Diego: Academic Press, 1993.

HARR06 Harris, W. "Multi-core in the Source Engine." bit-tech.net technical paper, November 2, 2006. bit-tech.net/gaming/2006/11/02/Multi_core_in_the_Source_Engin/1

HAUE07 Haeusser, B., et al. *IBM System Storage Tape Library Guide for Open Systems.* IBM Redbook SG24-5946-05, October 2007. ibm.com/redbooks

HAYE98 Hayes, J. *Computer Architecture and Organization.* New York: McGraw-Hill, 1998.

HEAT84 Heath, J. "Re-Evaluation of RISC I." *Computer Architecture News,* March 1984.

HENN82 Hennessy, J., et al. "Hardware/Software Tradeoffs for Increased Performance." *Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems,* March 1982.

HENN84 Hennessy, J. "VLSI Processor Architecture." *IEEE Transactions on Computers,* December 1984.

HENN91 Hennessy, J., and Jouppi, N. "Computer Technology and Architecture: An Evolving Interaction." *Computer,* September 1991.

HENN06 Henning, J. "SPEC CPU2006 Benchmark Descriptions." *Computer Architecture News,* September 2006.

HENN07 Henning, J. "SPEC CPU Suite Growth: An Historical Perspective." *Computer Architecture News,* March 2007.

HIDA90 Hidaka, H.; Matsuda, Y.; Asakura, M.; and Kazuyasu, F. "The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory." *IEEE Micro,* April 1990.

HIGB90 Highie, J. "Quick and Easy Cache Performance Analysis." *Computer Architecture News,* June 1990.

HILL64 Hill, R. "Stored Logic Programming and Applications." *Datamation,* February 1964.

HILL89 Hill, M. "Evaluating Associativity in CPU Caches." *IEEE Transactions on Computers,* December 1989.

HILL00 Hill, M.; Jouppi, N.; and Sohi, G. *Readings in Computer Architecture.* San Francisco: Morgan Kaufmann, 2000.

HINT01 Hinton, G., et al. "The Microarchitecture of the Pentium 4 Processor." *Intel Technology Journal*, Q1 2001. http://developer.intel.com/technology/itj/

HIRA07 Hirata, K., and Goodacre, J. "ARM MPCore: The Streamlined and Scalable ARM11 processor core." *Proceedings, 2007 Conference on Asia South Pacific Design Automation*, 2007.

HUCK83 Huck, T. *Comparative Analysis of Computer Architectures*. Stanford University Technical Report No. 83-243, May 1983.

HUCK00 Huck, J., et al. "Introducing the IA-64 Architecture." *IEEE Micro*, September/October 2000.

HUGU91 Huguet, M., and Lang, T. "Architectural Support for Reduced Register Saving/Restoring in Single-Window Register Files." *ACM Transactions on Computer Systems*, February 1991.

HUTC96 Hutcheson, G., and Hutcheson, J. "Technology and Economics in the Semiconductor Industry." *Scientific American*, January 1996.

HWAN93 Hwang, K. *Advanced Computer Architecture*. New York: McGraw-Hill, 1993.

HWAN99 Hwang, K., et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January–March 1999.

HWU98 Hwu, W. "Introduction to Predicated Execution." *Computer*, January 1998.

HWU01 Hwu, W.; August, D.; and Sias, J. "Program Decision Logic Optimization Using Predication and Control Speculation." *Proceedings of the IEEE*, November 2001.

IBM01 International Business Machines, Inc. *64 Mb Synchronous DRAM*. IBM Data Sheet 364164, January 2001.

INTE98 Intel Corp. *Pentium Pro and Pentium II Processors and Related Products*. Aurora, CO, 1998.

INTE00a Intel Corp. *Intel IA-64 Architecture Software Developer's Manual (4 volumes)*. Document 245317 through 245320. Aurora, CO, 2000.

INTE00b Intel Corp. *Itanium Processor Microarchitecture Reference for Software Optimization*. Aurora, CO, Document 245473. August 2000.

INTE01a Intel Corp. *Intel Pentium 4 Processor Optimization Reference Manual*. Document 248966-04 2001. http://developer.intel.com/design/Pentium4/documentation.htm

INTE01b Intel Corp. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Document 248966-04 2001 http://developer.intel.com/design/Pentium4/documentation.htm

INTE04a Intel Corp. *IA-32 Intel Architecture Software Developer's Manual (4 volumes)*. Document 253665 through 253668. 2004. http://developer.intel.com/design/Pentium4/documentation.htm

INTE04b Intel Research and Development. *Architecting the Era of Tera*. Intel White Paper, February 2004. http://www.intel.com/labs/teracra/index.htm

INTE04b Intel Corp. *Endianness White Paper*. November 15, 2004.

INTE08 Intel Corp. Intel 64 and *IA-32 Intel Architectures Software Developer's Manual (3 volumes)*. Denver, CO, 2008. intel.com/products/processor/manuals

JACO08 Jacob, B.; Ng, S.; and Wang, D. *Memory Systems: Cache, DRAM, Disk*. Boston: Morgan Kaufmann, 2008.

JAME90 James, D. "Multiplexed Buses: The Endian Wars Continue." *IEEE Micro*, September 1983.

JARP01 Jarp, S. "Optimizing IA-64 Performance." *Dr. Dobb's Journal*, July 2001.

JERR05 Jerraya, A., and Wolf, W., eds. *Multiprocessor Systems-on-Chips*. San Francisco: Morgan Kaufmann, 2005.

JOHN91 Johnson, M. *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

JOHN08 John, E., and Rubio, J. *Unique Chips and Systems*. Boca Raton, FL: CRC Press, 2008.

JOUP88 Jouppi, N. "Superscalar versus Superpipelined Machines." *Computer Architecture News*, June 1988.

JOUP89a Jouppi, N., and Wall, D. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines." *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

JOUP89b Jouppi, N. "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance." *IEEE Transactions on Computers*, December 1989.

KAEL91 Kaeli, D., and Emma, P. "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns." *Proceedings, 18th Annual International Symposium on Computer Architecture*, May 1991.

KAGA01 Kagan, M. "InfiniBand: Thinking Outside the Box Design." *Communications System Design*, September 2001. www.csdmag.com

KALL04 Kalla, R.; Sinharoy, B.; and Tendler, J. "IBM Power5 Chip: A Dual-Core Multithreaded Processor." *IEEE Micro*, March–April 2004.

KANE92 Kane, G., and Heinrich, J. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1992.

KAPP00 Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.

KATE83 Katevenis, M. *Reduced Instruction Set Computer Architectures for VLSI*. PhD dissertation, Computer Science Department, University of California at Berkeley, October 1983. Reprinted by MIT Press, Cambridge, MA, 1985.

KATH01 Kathail, B.; Schlansker, M.; and Rau, B. "Compiling for EPIC Architectures." *Proceedings of the IEEE*, November 2001.

KATZ89 Katz, R.; Gibson, G.; and Patterson, D. "Disk System Architecture for High Performance Computing." *Proceedings of the IEEE*, December 1989.

KEET01 Keeth, B., and Baker, R. *DRAM Circuit Design: A Tutorial*. Piscataway, NJ: IEEE Press, 2001.

KHUR01 Khurshudov, A. *The Essential Guide to Computer Data Storage*. Upper Saddle River, NJ: Prentice Hall, 2001.

KNAG04 Knaggs, P., and Welsh, S. *ARM: Assembly Language Programming*. Bournemouth University, School of Design, Engineering, and Computing, August 31, 2004. www.freetechbooks.com/arm-assembly-language-programming-t729.html

KNUT71 Knuth, D. "An Empirical Study of FORTRAN Programs." *Software Practice and Experience*, vol. 1, 1971.

KNUT98 Knuth, D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1998.

KOOP96 Koopman, P. "Embedded System Design Issues (the Rest of the Story). *Proceedings, 1996 International Conference on Computer Design*, 1996.

KUCK77 Kuck, D.; Parker, D.; and Sameh, A. "An Analysis of Rounding Methods in Floating-Point Arithmetic." *IEEE Transactions on Computers*, July 1977.

KUGA91 Kuga, M.; Murakami, K.; and Tomita, S. "DSNS (Dynamically-hazard resolved, Statically-code-scheduled, Nonuniform Superscalar): Yet Another Superscalar Processor Architecture." *Computer Architecture News*, June 1991.

LEE91 Lee, R.; Kwok, A.; and Briggs, F. "The Floating Point Performance of a Superscalar SPARC Processor." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

LEON07 Leonard, T. "Dragged Kicking and Screaming: Source Multicore." *Proceedings, Game Developers Conference 2007*, March 2007.

LEON08 Leong, p. "Recent Trends in FPGA Architectures and Applications." *Proceedings, 4th IEEE International symposium on Electronic Design, Test, and Applications*, 2008.

LEVI00 Levine, J. *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 2000.

LILJ88 Lilja, D. "Reducing the Branch Penalty in Pipelined Processors." *Computer*, July 1988.

LILJ93 Lilja, D. "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Computing Surveys*, September 1993.

LOVE96 Lovett, T., and Clapp, R. "Implementation and Performance of a CC-NUMA System." *Proceedings, 23rd Annual International Symposium on Computer Architecture*, May 1996.

LUND77 Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures." *Communications of the ACM*, March 1977.

LYNC93 Lynch, M. *Microprogrammed State Machine Design*. Boca Raton, FL: CRC Press, 1993.

MACD84 MacDougall, M. "Instruction-level Program and Process Modeling." *IEEE Computer*, July 1984.

MAHL94 Mahlke, S., et al. "Characterizing the Impact of Predicated Execution on Branch Prediction." *Proceedings, 27th International Symposium on Microarchitecture*, December 1994.

MAHL95 Mahlke, S., et al. "A Comparison of Full and Partial Predicated Execution Support for ILP Processors." *Proceedings, 22nd International Symposium on Computer Architecture*, June 1995.

MAK04 Mak, P., et al. "Processor Subsystem Interconnect for a Large Symmetric Multiprocessing System." *IBM Journal of Research and Development*, May/July 2004.

MANJ01a Manjikian, N. "More Enhancements of the SimpleScalar Tool Set." *Computer Architecture News*, September 2001.

MANJ01b Manjikian, N. "Multiprocessor Enhancements of the SimpleScalar Tool Set." *Computer Architecture News*, March 2001.

MANO04 Mano, M. *Logic and Computer Design Fundamentals*. Upper Saddle River, NJ: Prentice Hall, 2004.

MANS97 Mansuripur, M., and Sincerbox, G. "Principles and Techniques of Optical Data Storage." *Proceedings of the IEEE*, November 1997.

MARC90 Marchant, A. *Optical Recording*. Reading, MA: Addison-Wesley, 1990.

MARK00 Markstein, P. *IA-64 and Elementary Functions*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

MARR02 Marr, D.; et al. "Hyper-Threading Technology Architecture and Microarchitecture." *Intel Technology Journal*, First Quarter, 2002.

MASH95 Mashey, J. "CISC vs. RISC (or what is RISC really)." *USENET comp.arch newsgroup, article 46782*, February 1995.

MAYB84 Mayberry, W., and Efland, G. "Cache Boosts Multiprocessor Performance." *Computer Design*, November 1984.

MAZI03 Mazidi, M., and Mazidi, J. *The 80x86 IBM PC and Compatible Computers: Assembly Language, Design and Interfacing*. Upper Saddle River, NJ: Prentice Hall, 2003.

MCDO05 McDougall, R. "Extreme Software Scaling." *ACM Queue*, September 2005.

MCDO06 McDougall, R., and Laudon, J. "Multi-Core Microprocessors are Here." *;login*. October 2006.

MCEL85 McEliece, R. "The Reliability of Computer Memories." *Scientific American*, January 1985.

MCNA03 McNairy, C., and Soltis, D. "Itanium 2 Processor Microarchitecture." *IEEE Micro*. March-April 2003.

MEE96a Mee, C., and Daniel, E. eds. *Magnetic Recording Technology*. New York: McGraw-Hill, 1996.

MEE96b Mee, C., and Daniel, E. eds. *Magnetic Storage Handbook*. New York: McGraw-Hill, 1996.

MEND06 Mendelson, A., et al. "CMP Implementation in Systems Based on the Intel Core Duo Processor." *Intel Technology Journal*, May 2006.

MILE00 Milenkovic, A. "Achieving High Performance in Bus-Based Shared-Memory Multiprocessors." *IEEE Concurrency*, July-September 2000.

MIRA92 Mirapuri, S.; Woodacre, M.; and Vasseghi, N. "The MIPS R4000 Processor." *IEEE Micro*, April 1992.

MOOR65 Moore, G. "Cramming More Components Onto Integrated Circuits." *Electronics Magazine*, April 19, 1965.

MORS78 Morse, S.; Pohlman, W.; and Ravenel, B. "The Intel 8086 Microprocessor: A 16-bit Evolution of the 8080." *Computer*, June 1978.

MOSH01 Moshovos, A., and Sohi, G. "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling." *Proceedings of the IEEE*, November 2001.

MYER78 Myers, G. "The Evaluation of Expressions in a Storage-to-Storage Architecture." *Computer Architecture News*, June 1978.

NAFF02 Naffziger, S., et al. "The Implementation of the Itanium 2 Microprocessor." *IEEE Journal of Solid-State Circuits*, November 2002.

NOER05 Noergaard, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. New York: Elsevier, 2005.

NOVI93 Novitsky, J.; Azimi, M.; and Ghaznavi, R. "Optimizing Systems Performance Based on Pentium Processors." *Proceedings COMPCON '92*, February 1993.

NOWE07 Nowell, M.; Vusirikala, V.; and Hays, R. "Overview of Requirements and Applications for 40 Gigabit and 100 Gigabit Ethernet." *Ethernet Alliance White Paper*, August 2007.

OBER97a Oberman, S., and Flynn, M. "Design Issues in Division and Other Floating-Point Operations." *IEEE Transactions on Computers*, February 1997.

OBER97b Oberman, S., and Flynn, M. "Division Algorithms and Implementations." *IEEE Transactions on Computers*, August 1997.

OLUK96 Olukotun, K., et al. "The Case for a Single-Chip Multiprocessor." *Proceedings, Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

OLUK05 Olukotun, K., and Hammond, L. "The Future of Microprocessors." *ACM Queue*, September 2005.

OLUK07 Olukotun, K.; Hammond, L.; and Laudon, J. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. San Rafael, CA: Morgan & Claypool, 2007.

OMON99 Omondi, A. *The Microarchitecture of Pipelined and Superscalar Computers*. Boston: Kluwer, 1999.

OVER01 Overton, M. *Numerical Computing with IEEE Floating Point Arithmetic*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2001.

PADE81 Padegs, A. "System/360 and Beyond." *IBM Journal of Research and Development*, September 1981.

PADE88 Padegs, A.; Moore, B.; Smith, R.; and Buchholz, W. "The IBM System/370 Vector Architecture: Design Considerations." *IEEE Transactions on Communications*, May 1988.

PARH00 Parhami, B. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford: Oxford University Press, 2000.

PARK89 Parker, A., and Hamblen, J. *An Introduction to Microprogramming with Exercises Designed for the Texas Instruments SN74ACT8800 Software Development Board*. Dallas, TX: Texas Instruments, 1989.

PATT82a Patterson, D., and Sequin, C. "A VLSI RISC." *Computer*, September 1982.

PATT82b Patterson, D., and Piepho, R. "Assessing RISCs in High-Level Language Support." *IEEE Micro*, November 1982.

PATT84 Patterson, D. "RISC Watch." *Computer Architecture News*, March 1984.

PATT85a Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM*, January 1985.

PATT85b Patterson, D., and Hennessy, J. "Response to 'Computers, Complexity, and Controversy.'" *Computer*, November 1985.

PATT88 Patterson, D.; Gibson, G.; and Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings, ACM SIGMOD Conference of Management of Data*, June 1988.

PATT01 Patt, Y. "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution." *Proceedings of the IEEE*, November 2001.

PEIR99 Peir, J.; Hsu, W.; and Smith, A. "Functional Implementation Techniques for CPU Cache Memories." *IEEE Transactions on Computers*, February 1999.

PELE97 Peleg, A.; Wilkie, S.; and Weiser, U. "Intel MMX for Multimedia PCs." *Communications of the ACM*, January 1997.

PFIS98 Pfister, G. *In Search of Clusters*. Upper Saddle River, NJ: Prentice Hall, 1998.

POLL99 Pollack, F. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)." *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

POPE91 Popescu, V., et al. "The Metaflow Architecture." *IEEE Micro*, June 1991.

PRES01 Pressel, D. "Fundamental Limitations on the Use of Prefetching and Stream Buffers for Scientific Applications." *Proceedings, ACM Symposium on Applied Computing*, March 2001.

PRIN97 Prince, B. *Semiconductor Memories*. New York: Wiley, 1997.

PRIN02 Prince, B. *Emerging Memories: Technologies and Trends*. Norwell, MA: Kluwer, 2002.

PRZY88 Przybylski, S.; Horowitz, M.; and Hennessy, J. "Performance Trade-offs in Cache Design." *Proceedings, Fifteenth Annual International Symposium on Computer Architecture*, June 1988.

PRZY90 Przybylski, S. "The Performance Impact of Block Size and Fetch Strategies." *Proceedings, 17th Annual International Symposium on Computer Architecture*, May 1990.

RADD08 Radding, A. "Small Disks, Big Specs." *Storage Magazine*, September 2008.

RADI83 Radin, G. "The 801 Minicomputer." *IBM Journal of Research and Development*, May 1983.

RAGA83 Ragan-Kelley, R., and Clark, R. "Applying RISC Theory to a Large Computer." *Computer Design*, November 1983.

RAMA77 Ramamoorthy, C. "Pipeline Architecture." *Computing Surveys*, March 1977.

RECH98 Reches, S., and Weiss, S. "Implementation and Analysis of Path History in Dynamic Branch Prediction Schemes." *IEEE Transactions on Computers*, August 1998.

REDD76 Reddi, S., and Feustel, E. "A Conceptual Framework for Computer Architecture." *Computing Surveys*, June 1976.

REIM06 Reimer, J. "Valve Goes Multicore." ars technica, November 5, 2006. arstechnica.com/articles/paedia/cpu/valve-multicore.ars

RICH07 Riches, S., et al. "A Fully Automated High Performance Implementation of ARM Cortex-A8." *IQ Online*, Vol. 6, No. 3, 2007. www.arm.com/iqonline

RODR01 Rodriguez, M.; Perez, J.; and Pulido, J. "An Educational Tool for Testing Caches on Symmetric Multiprocessors." *Microprocessors and Microsystems*, June 2001.

ROSC03 Rosch, W. *Winn L. Rosch Hardware Bible*. Indianapolis, IN: Que Publishing, 2003.

SAKA02 Sakai, S. "CMP on SoC: architect's view." *Proceedings, 15th International Symposium on System Synthesis*, 2002.

SALO93 Salomon, D. *Assemblers and Loaders*. Ellis Horwood Ltd, 1993. Available at this book's Web site.

SATY81 Satyanarayanan, M., and Bhandarkar, D. "Design Trade-Offs in VAX 11 Translation Buffer Organization." *Computer*, December 1981.

SCHA97 Schaller, R. "Moore's Law: Past, Present, and Future." *IEEE Spectrum*, June 1997.

SCHL00a Schlansker, M.; and Rau, B. "EPIC: Explicitly Parallel Instruction Computing." *Computer*, February 2000.

SCHL00b Schlansker, M.; and Rau, B. *EPIC: An Architecture for Instruction-Level Parallel Processors*. HPL Technical Report HPL-1999-111, Hewlett-Packard Laboratories (www.hpl.hp.com), February 2000.

SCHW99 Schwarz, E., and Krygowski, C. "The S/390 G5 Floating-Point Unit." *IBM Journal of Research and Development*, September/November 1999.

SEAL00 Seal, D., ed. *ARM Architecture Reference Manual*. Reading, MA: Addison-Wesley, 2000.

SEBE76 Sebern, M. "A Minicomputer-compatible Microcomputer System: The DEC LSI-11." *Proceedings of the IEEE*, June 1976.

SEGA95 Segars, S.; Clarke, K.; and Goudge, L. "Embedded Control Problems, Thumb, and the ARM7TDMI." *IEEE Micro*, October 1995.

SEGE91 Segee, B., and Field, J. *Microprogramming and Computer Architecture*. New York: Wiley, 1991.

SERL86 Serlin, O. "MIPS, Dhrystones, and Other Tales." *Datamation*, June 1, 1986.

SHAN38 Shannon, C. "Symbolic Analysis of Relay and Switching Circuits." *AIEE Transactions*, vol. 57, 1938.

SHAN99 Shanley, T., and Anderson, D. *PCI Systems Architecture*. Richardson, TX: Mindshare Press, 1999.

SHAN03 Shanley, T. *InfiniBand Network Architecture*. Reading, MA: Addison-Wesley, 2003.

SHAN05 Shanley, T. *Unabridged Pentium 4, The: IA32 Processor Genealogy*. Reading, MA: Addison-Wesley, 2005.

SHAR97 Sharma, A. *Semiconductor Memories: Technology, Testing, and Reliability*. New York: IEEE Press, 1997.

SHAR00 Sharangpani, H., and Arona, K. "Itanium Processor Microarchitecture." *IEEE Micro*, September/October 2000.

SHAR03 Sharma, A. *Advanced Semiconductor Memories: Architectures, Designs, and Applications*. New York: IEEE Press, 2003.

SHEN05 Shen, J., and Lipasti, M. *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005.

SIEG04 Siegel, T.; Pfeffer, E.; and Magee, A. "The IBM z990 Microprocessor." *IBM Journal of Research and Development*, May/July 2004.

SIEW82 Siewiorek, D.; Bell, C.; and Newell, A. *Computer Structures: Principles and Examples*. New York: McGraw-Hill, 1982.

SIMA97 Sima, D. "Superscalar Instruction Issue." *IEEE Micro*, September/October 1997.

SIMA04 Sima, D. "Decisive Aspects in the Evolution of Microprocessors." *Proceedings of the IEEE*, December 2004.

SIMO96 Simon, H. *The Sciences of the Artificial*. Cambridge, MA: MIT Press, 1996.

SLOS04 Sloss, A.; Symes, D.; and Wright, C. *ARM System Developer's Guide*. San Francisco: Morgan Kaufmann, 2004.

SMIT82 Smith, A. "Cache Memories." *ACM Computing Surveys*, September 1992.

SMIT95 Smith, J., and Sohi, G. "The Microarchitecture of Superscalar Processors." *Proceedings of the IEEE*, December 1995.

SMIT87 Smith, A. "Line (Block) Size Choice for CPU Cache Memories." *IEEE Transactions on Communications*, September 1987.

SMIT88 Smith, J. "Characterizing Computer Performance with a Single Number." *Communications of the ACM*, October 1988.

SMIT89 Smith, M.; Johnson, M.; and Horowitz, M. "Limits on Multiple Instruction Issue." *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

SMIT08 Smith, B. "ARM and Intel Battle over the Mobile Chip's Future." *Computer*, May 2008.

SODE96 Soderquist, P., and Leeser, M. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys*, September 1996.

SOHI90 Sohi, G. "Instruction Issue Logic for High-Performance Interruptable, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

STAL88 Stallings, W. "Reduced Instruction Set Computer Architecture." *Proceedings of the IEEE*, January 1988.

STAL07 Stallings, W. *Data and Computer Communications, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2007.

STAL09 Stallings, W. *Operating Systems, Internals and Design Principles, Sixth Edition*. Upper Saddle River, NJ: Prentice Hall, 2009.

STEN90 Stenstrom, P. "A Survey of Cache Coherence Schemes of Multiprocessors." *Computer*, June 1990.

STEV64 Stevens, W. "The Structure of System/360, Part II: System Implementation." *IBM Systems Journal*, Vol. 3, No. 2, 1964. Reprinted in [SIEW82].

STON93 Stone, H. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1993.

STON96 Stonham, T. *Digital Logic Techniques*. London: Chapman & Hall, 1996.

STRE78 Strecker, W. "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family." *Proceedings, National Computer Conference*, 1978.

STRE83 Strecker, W. "Transient Behavior of Cache Memories." *ACM Transactions on Computer Systems*, November 1983.

STRI79 Stritter, E., and Gunter, T. "A Microprocessor Architecture for a Changing World: The Motorola 68000." *Computer*, February 1979.

SWAR90 Swartzlander, E., editor. *Computer Arithmetic, Volumes I and II*. Los Alamitos, CA: IEEE Computer Society Press, 1990.

TAMI83 Tamir, Y., and Sequin, C. "Strategies for Managing the Register File in RISC." *IEEE Transactions on Computers*, November 1983.

TANE78 Tanenbaum, A. "Implications of Structured Programming for Machine Architecture." *Communications of the ACM*, March 1978.

TANE97 Tanenbaum, A., and Woodhull, A. *Operating Systems: Design and Implementation.* Upper Saddle River, NJ: Prentice Hall, 1997.

THOM00 Thompson, D. "IEEE 1394: Changing the Way We Do Multimedia Communications." *IEEE Multimedia,* April–June 2000.

TI90 Texas Instruments Inc. *SN74ACT880 Family Data Manual.* SCSS006C, 1990.

TJAD70 Tjaden, G., and Flynn, M. "Detection and Parallel Execution of Independent Instructions." *IEEE Transactions on Computers,* October 1970.

TOMA93 Tomasevic, M., and Milutinovic, V. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions.* Los Alamitos, CA: IEEE Computer Society Press, 1993.

TOON81 Toong, H., and Gupta, A. "An Architectural Comparison of Contemporary 16-Bit Microprocessors." *IEEE Micro,* May 1981.

TRIE01 Triebel, W. *Itanium Architecture for Software Developers.* Intel Press, 2001.

TUCK67 Tucker, S. "Microprogram Control for System/360." *IBM Systems Journal,* No. 4, 1967.

TUCK87 Tucker, S. "The IBM 3090 System Design with Emphasis on the Vector Facility." *Proceedings, COMPCON Spring '87,* February 1987.

UNGE02 Ungerer, T.; Rubic, B.; and Silc, J. "Multithreaded Processors." *The Computer Journal,* No. 3, 2002.

UNGE03 Ungerer, T.; Rubic, B.; and Silc, J. "A Survey of Processors with Explicit Multithreading." *ACM Computing Surveys,* March 2003.

VASS03 Vassiliadis, S.; Wong, S.; and Cotofana, S. "Microcode Processing: Positioning and Directions." *IEEE Micro,* July–August 2003.

VOEL88 Voelker, J. "The PDP-8." *IEEE Spectrum,* November 1988.

VOGL94 Vogley, B. "800 Megabyte Per Second Systems Via Use of Synchronous DRAM." *Proceedings, COMPCON '94,* March 1994.

VONN45 Von Neumann, J. *First Draft of a Report on the EDVAC.* Moore School, University of Pennsylvania, 1945. Reprinted in *IEEE Annals on the History of Computing,* No. 4, 1993.

VRAN80 Vranesic, Z., and Thurber, K. "Teaching Computer Structures." *Computer,* June 1980.

WALL85 Wallich, P. "Toward Simpler, Faster Computers." *IEEE Spectrum,* August 1985.

WALL91 Wall, D. "Limits of Instruction-Level Parallelism." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* April 1991.

WANG99 Wang, G., and Tafti, D. "Performance Enhancement on Microprocessors with Hierarchical Memory Systems for Solving Large Sparse Linear Systems." *International Journal of Supercomputing Applications,* vol. 13, 1999.

WEIC90 Weicker, R. "An Overview of Common Benchmarks." *Computer,* December 1990.

WEIN75 Weinberg, G. *An Introduction to General Systems Thinking.* New York: Wiley, 1975.

WEIS84 Weiss, S., and Smith, J. "Instruction Issue Logic in Pipelined Supercomputers." *IEEE Transactions on Computers,* November 1984.

WEYG01 Weygant, P. *Clusters for High Availability.* Upper Saddle River, NJ: Prentice Hall, 2001.

WHIT97 Whitney, S., et al. "The SGI Origin Software Environment and Application Performance." *Proceedings, COMPCON Spring '97,* February 1997.

WICK97 Wickelgren, I. "The Facts About FireWire." *IEEE Spectrum,* April 1997.

WILK51 Wilkes, M. "The Best Way to Design an Automatic Calculating Machine." *Proceedings, Manchester University Computer Inaugural Conference.* July 1951.

WILK53 Wilkes, M., and Stringer, J. "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer." *Proceedings of the Cambridge Philosophical Society,* April 1953. Reprinted in [SIEW82].

WILK65 Wilkes, M. "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers,* April 1965. Reprinted in [HILL00].

WILL90 Williams, F., and Steven, G. "Address and Data Register Separation on the M68000 Family." *Computer Architecture News,* June 1990.

YEH91 Yeh, T., and Patt, N. "Two-Level Adaptive Training Branch Prediction." *Proceedings, 24th Annual International Symposium on Microarchitecture,* 1991.

ZHAN01 Zhang, Z.; Zhu, Z.; and Zhang, X. "Cached DRAM for ILP Processor Memory Access Latency Reduction." *IEEE Micro,* July–August 2001.

计算机科学丛书特别推荐









作者：Bob Hughes，Mike Cotterell 著
译者：[illegible]
书号：978-7-111-30984-2
定价：39.00

[illegible]

作者：Bjarne Stroustrup 著
作者：[illegible]
书号：978-7-111-29986-4
定价：99.00

作者：Thomas H. Cormen 著
[illegible]

